

JavaScript

深入 JAVASCRIPT 程式設計

DARREN YANG 楊德倫

內容

Module 1. JavaScript 基本語法	1
1-1: 基本類型	1
1-2: 運算子和運算元	7
1-3: 流程控制	21
Module 2. 基本集合類型	35
2-1: Object.....	35
2-2: Array.....	41
2-3: for/in 迴圈	50
Module 3. 操作 DOM	52
3-1: 取得 DOM 元素.....	52
3-2: 建立 DOM 元素.....	55
3-3: 刪除 DOM 元素.....	60
Module 4. 深入 DOM 元素	61
4-1: 屬性操作	61
4-2: 自訂屬性	62
4-3: 元素與樣式	63
Module 5. DOM 元素集合.....	67
5-1: querySelectorAll 與 NodeList	67
5-2: document 的子物件與 HTMLCollection	70

5-3: NodeList 不是 Array	70
Module 6. 函式	72
6-1: 函式定義	72
6-2: 呼叫	88
6-3: 綁定	88
Module 7. 表單	95
7-1: 文字欄	95
7-2: 單選項目	95
7-3: 多選項目	95
Module 8. 深入表單	97
8-1: 檔案上傳	97
8-2: 上傳前預覽	98
8-3: 自訂上傳按鈕	100
Module 9. 事件處理器	103
9-1: 不同形式的事件處理器	103
9-2: 滑鼠事件	104
9-3: 事件模型	105
Module 10. 正規表示法	113
10-1: 單一字元表示法	116

10-2: 多字元表示法	117
10-3: 表單送出前的檢查	118
Module 11. 網頁測試伺服器	119
11-1: 安裝 node 和 http-server.....	119
11-2: 測試頁面	125
11-3: 測試送出 querystring.....	126
Module 12. Promise 類型	127
12-1: unblock 與 callback hell.....	127
12-2: Promise 的優點	131
12-3: Promise 建議的使用方式	132
Module 13. AJAX	135
13-1: 什麼是 AJAX	135
13-2: XMLHttpRequest.....	135
13-3: fetch 函式	137
Module 14. 物件導向使用 prototype.....	139
14-1: 自訂功能物件	139
14-2: 使用 function 自訂類型.....	140
14-3: 擴展類型功能	141
Module 15. 物件導向使用 class	142
15-1: 類別基本定義方式	142

15-2: 建構函式	142
15-3: 屬性與方法	143
Module 16. 使用 babel	144
16-1: 使用 npm 建立專案	144
16-2: 安裝和設定 babel	145
16-3: 引入和匯出模組	146
Module 17. 在專案上使用 webpack	148
17-1: 安裝 webpack	148
17-2: 安裝 loader	149
17-3: 功能模組化	151
Module 18. ES7 的 await 和 async	153
18-1: 改善 Promise 缺點	153
18-2: await 使用時機	154
18-3: async 的使用	154

Module 1. JavaScript 基本語法

1-1: 基本類型

Number、Boolean 和 String

在 JavaScript 中有八種主要的型別：

- 三種基本型別：
 - 布林(Boolean)
 - 數值(Number)
 - 字串(String)
- 兩種複合的型別：
 - 陣列(Array)
 - 物件(Object)
- 兩種簡單型別：
 - 空值(null)
 - 未定義(undefined)
- 一種特殊型別：
 - 函式(Function)

以下我們透過簡單的例子，來介紹數值；我們也可以透過變數之間的運算，來建立新的變數：

範例 1-1-1.html

```
<html>
<head>
  <title>範例 1-1-1.html</title>
  <script>
    let secondsInMinute = 60; //一分鐘 60 秒
    let minutesInAnHour = 60; //每小時 60 分鐘
    let secondsInAnHour = secondsInMinute * minutesInAnHour;
    console.log(secondsInAnHour);
    // 輸出 3600 (秒)

    let hoursInADay = 24;
    let secondsInADay = secondsInAnHour * hoursInADay;
```

```

    console.log(secondsInADay);
    // 輸出 86400
  </script>
</head>
<body>

</body>
</html>

```

布林提供了 true (真) 與 false (假) 的判斷值：

範例 1-1-2.html

```

<html>
<head>
  <title>範例 4-1-2.html</title>
  <script>
    //設定變數為 true (真)
    let bool = true;
    console.log(bool);

    //設定變數為 false (假)
    bool = false;
    console.log(bool);
  </script>
</head>
<body>

</body>
</html>

```

Javascript 中的字串有序列的概念，可以包括文字、數字、標點與空格等，例如「Hello World!」。字串與相關處理，非常重要，我們下面使用一些具體的範例，供大家參考。

範例

```

let myStr = "Hello World!";
console.log(myStr);
// 輸出 Hello World!

```

```
let myStr01 = "Hello";
let myStr02 = " "; //裡面是空格
let myStr03 = "World";
let myStr04 = "!";
let myStr05 = myStr01 + myStr02 + myStr03 + myStr04;
console.log(myStr05);
// 輸出 Hello World!

let fname = 'Darren';
let lname = 'Yang';
console.log(fname + ' ' + lname);
// 輸出 Darren Yang

let numberNine = 9;
let stringNine = "9";
console.log(numberNine + numberNine); // 輸出 18
console.log(stringNine + stringNine); // 輸出 99
console.log(numberNine + stringNine); // 輸出 99，數值將會自動轉型，與 "9"
前後串接
```

範例 1-1-3.html

```
<html>
<head>
  <title>範例 1-1-3.html</title>
  <script>
    let myStr = "Hello World!";
    console.log(myStr);
    // 輸出 Hello World!

    let myStr01 = "Hello";
    let myStr02 = " "; //裡面是空格
    let myStr03 = "World";
    let myStr04 = "!";
    let myStr05 = myStr01 + myStr02 + myStr03 + myStr04;
    console.log(myStr05);
    // 輸出 Hello World!
```



```

let fname = 'Darren';
let lname = 'Yang';
console.log(fname + ' ' + lname);
// 輸出 Darren Yang

let numberNine = 9;
let stringNine = "9";
console.log(numberNine + numberNine); // 輸出 18
console.log(stringNine + stringNine); // 輸出 99
console.log(numberNine + stringNine); // 輸出 99，數值將會自動轉型，
與 "9" 前後串接
</script>
</head>
<body>

</body>
</html>

```

轉換為 **Number**

常見的有：

- parseInt()
- parseFloat()
- Number()

parseInt()可以傳回由字串轉換而成的整數：

範例	
parseInt("abc")	// 傳回 NaN
parseInt("123abc")	// 傳回 123
parseInt("abc123")	// 傳回 NaN
parseInt(" 123abc")	// 傳回 123

parseFloat()可以傳回由字串轉換而成的浮點數。

- parseFloat 會傳回一個在 String 中之的數值。如果沒有任何可以傳回的浮點數值，則會傳回 NaN (使用 isNaN()可以判斷是否為 NaN)。
- parseFloat 只傳回第一個數字。前後空格會被省略。

範例	
parseFloat("20");	//傳回 20
parseFloat("30.00");	//傳回 30

parseFloat("10.68");	//傳回 10.68
parseFloat("12 22 32");	//傳回 12
parseFloat("80 ");	//傳回 80
parseFloat("378abc");	//傳回 378
parseFloat("abc378");	//傳回 NaN

Number() 可以將物件轉化成數值。

- 若無法轉成數字則傳回 NaN

範例	
Number(true);	//傳回 1
Number(false);	//傳回 0
Number(new Date());	//傳回 1970/1/1 到現在的毫秒數
Number("123");	//傳回 123
Number("123 456");	//傳回 NaN

範例 1-1-4.html

```
<html>
<head>
  <title>範例 1-1-4.html</title>
  <script>
    //parseInt()可以傳回由字串轉換而成的整數。
    console.log( parseInt("abc") );           // 傳回 NaN
    console.log( parseInt("123abc") );         // 傳回 123
    console.log( parseInt("abc123") );         // 傳回 NaN
    console.log( parseInt("123abc") );         // 傳回 123

    //parseFloat()可以傳回由字串轉換而成的浮點數。
    //parseFloat 會傳回一個在 String 中之的數值。
    //沒有任何可以傳回的浮點數值，則傳回 NaN(使用 isNaN()可判斷是否為 NaN)。
    //parseFloat 只傳回第一個數字。前後空格會被省略。
    console.log( parseFloat("20") );           //傳回 20
    console.log( parseFloat("30.00") );         //傳回 30
    console.log( parseFloat("10.68") );         //傳回 10.68
    console.log( parseFloat("12 22 32") );       //傳回 12
    console.log( parseFloat("80 ") );           //傳回 80
    console.log( parseFloat("378abc") );         //傳回 378
    console.log( parseFloat("abc378") );         //傳回 NaN
```

```

//Number() 可以將物件轉化成數值。
//若無法轉成數字則傳回 NaN
console.log( Number(true) );           //傳回 1
console.log( Number(false) );          //傳回 0
console.log( Number(new Date()) );     //傳回 1970/1/1 至今的毫秒數
console.log( Number("123") );          //傳回 123
console.log( Number("123 456") );      //傳回 NaN
</script>
</head>
<body>

</body>
</html>

```

轉換為 String

將變數轉為字串的方式有三種：

- 變數.toString()
- "" + 變數
- String(變數)

範例 1-1-5.html

```

<html>
<head>
  <title>範例 1-1-5.html</title>
  <script>
    let myNum = 9487;

    //使用 toString()
    console.log( myNum.toString() );

    //使用字串轉型合併
    console.log( "" + myNum );

    //將各種類型的值，轉成字串
    console.log( String(myNum) );
  </script>
</head>
<body>

```

```
</body>
</html>
```

1-2: 運算子和運算元

算術運算子

JavaScript 可以進行加 (+)、減 (-)、乘 (*)、除 (/) 的基本運算，傳統的數學計算概念，也可以在程式當中運作，例如先乘除、後加減等等的概念；關於下面的基本運算，我們稱為「算數運算子」(Arithmetic operators)。

運算子	範例
加 (+)	$1 + 2 = 3$
減 (-)	$3 - 1 = 2$
乘 (*)	$3 * 4 = 12$
除 (/)	$8 / 2 = 4$
取餘數 (%)	$12 \% 5 = 2.$
遞增 (++)	假如 x 是3，那 ++x 將把x設定為 4 並回傳 4，而 x++ 會回傳 3，接著才把 x 設定為4。
遞減 (--)	假如 x 是 3，那 --x 將把x設定為 2 並回傳 2，而 x-- 會回傳 3，接著才把 x 設定為 2。
(一元運算子) 減號 (-)	假如 x 是 3，-x 回傳 -3。
(一元運算子) 加號 (+)	+"3" 回傳 3；+true 回傳1。
指數運算子 (**)	$2 ** 3$ 回傳 8。

範例 1-2-1.html

```
<html>
<head>
  <title>範例 1-2-1.html</title>
  <script>
    console.log(5566 + 3312);
    // 輸出 8878
```

```
console.log(11 + 22 + 33 + 44);  
// 輸出 110  
  
console.log(7 - 3 + 11);  
// 輸出 15  
  
console.log(8 * 3 + 12);  
// 輸出 36  
  
console.log(4 * 12 + 12 / 6);  
// 輸出 50  
  
console.log(7 - 8 / 2 + 23 * 3);  
// 輸出 72  
  
console.log(7 - 8 / (2 + 23) * 3);  
// 加上括號，輸出 6.04  
</script>  
</head>  
<body>  
  
</body>  
</html>
```

關係運算子

以相互比較的方式，來呈現布林邏輯運算結果，稱之為「關係運算子」或「比較運算子」，例如常常提到的「大於 >、大於等於 >=、等於 ==（或 ===）、小於 <、小於等於 <=、不等於 !=」等概念。

範例
<p>判斷身高與身高限制的比較</p> <pre>let height = 171; let heightRestriction = 140; console.log(height > heightRestriction); // 輸出 true console.log(height >= heightRestriction); // 輸出 true console.log(height == heightRestriction); // 輸出 false console.log(height < heightRestriction); // 輸出 false</pre>

```
console.log(height <= heightRestriction); // 輸出 false  
console.log(height != heightRestriction); // 輸出 true
```

範例 1-2-2.html

```
<html>  
<head>  
  <title>範例 1-2-2.html</title>  
  <script>  
    let hadShoes = true;  
    let hadBackpack = false;  
    console.log(hadShoes && hadBackpack);  
    //輸出 false，代表還沒準備好出門  
  
    let hasApple = true;  
    let hasBanana = false;  
    console.log(hasApple || hasBanana);  
    // 輸出 true  
  
    let isWeekend = true;  
    let workAllDay = !isWeekend;  
    console.log(workAllDay);  
    // 輸出 false，代表不需要工作整天  
  
    let height = 171;  
    let heightRestriction = 140;  
    console.log(height > heightRestriction); // 輸出 true  
    console.log(height >= heightRestriction); // 輸出 true  
    console.log(height == heightRestriction); // 輸出 false  
    console.log(height < heightRestriction); // 輸出 false  
    console.log(height <= heightRestriction); // 輸出 false  
    console.log(height != heightRestriction); // 輸出 true  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

補充說明

「==」 vs. 「===」 相等運算子

有時候閱讀程式設計教課書，在討論是否等價這件事，有著不同的寫法，如同上面的「==」與「===」，兩個的差別在於：

「`x == y`」：若是型態不相等，變數會先強制轉換成相同的型態，再進行嚴格比對。

```
console.log(1 == 1); // 輸出 true
console.log("1" == "1"); // 輸出 true
console.log(1 == "1"); // 輸出 true
console.log("1" == 1); // 輸出 true
```

註：若是比對的是「物件」，會比對變數佔用的記憶體位置是否相同。

```
var object1 = {'key': 'value'}, object2 = {'key': 'value'};
console.log(object1 == object2);
//輸出 false，變數宣告時，會各自佔用不同的記憶體位置（不同的物件）。
```

```
var object1 = {'key': 'value'};
var object2 = object1;
console.log(object1); // 輸出 { key: 'value' }
console.log(object2); // 輸出 { key: 'value' }
object2.key = 'vvv'; //隨意改值
console.log(object1); //輸出 { key: 'vvv' }
console.log(object2); //輸出 { key: 'vvv' }
console.log(object1 == object2); //輸出 true
```

「`===`」：兩個型態不相等，不轉換型態，直接嚴格比對。**最常用的等價邏輯判斷方式**，若為 `true`，意味著兩個值相同，類型也相等。

```
console.log(1 === 1); // 輸出 true
console.log("1" === "1"); // 輸出 true
console.log(1 === "1"); // 輸出 false
console.log("1" === 1); // 輸出 false
```

以下表格，為大家分析一下不同類型的值，用相等運算子(==)比較後的結果：

類型 (x)	類型 (y)	結果
--------	--------	----

null	undefined	true
undefined	null	true
數字	字串	<code>x == toNumber(y)</code>
字串	數字	<code>toNumber(x) == y</code>
布林值	任何類型	<code>toNumber(x) == y</code>
任何類型	布林值	<code>x == toNumber(y)</code>
字串或數字	物件	<code>x == toPrimitive(y)</code>
物件	字串或數字	<code>toPrimitive(x) == y</code>

`toPrimitive`（轉為基本類型 `Number` 或 `String`）通常用在「物件」上，它的轉換邏輯為：

- 如果 `input` 是基本類型，則直接回傳 `input`。
- `PreferredType` 為 `Number` 首選類型時（在物件前面直接放個「+」號，`+obj`），優先使用 `valueOf`，然後再呼叫 `toString`。
- `PreferredType` 為 `String` 首選類型時（用 `template strings` 進行轉換，``${obj}``），優先使用 `toString`，然後再呼叫 `valueOf`。
- 預設呼叫方式則是先呼叫 `valueOf` 再呼叫 `toString`，否則，拋出 `TypeError` 錯誤。
- 兩個例外，一個是 `Date` 物件預設首選類型是字串(`String`)，另一是 `Symbol` 物件，它們覆蓋了原來的 `PreferredType` 行為。

補充說明

若是我們今天要將「物件」進行轉換，可以用「`+obj`」取得 `valueOf` 的結果，用「``${obj}``」取得 `toString` 的結果（指定 `Preferred Type`）：

```
let tmp = {
  toString: function() { return 'foo' },
  valueOf: function() { return 123 }
}
```

```
// default, 優先使用 valueOf()
console.log( tmp == 123 ); // true
console.log( tmp == 'foo' ); // false
```

```
// number, 優先使用 valueOf()
console.log( +tmp == 123 ); // true
console.log( +tmp == 'foo' ); // false
```



```
// string，優先使用 toString()
console.log( `${tmp}` == 123 ); // false
console.log( `${tmp}` == 'foo' ); // true
```

範例 1-2-3.html

```
<html>
<head>
  <title>範例 1-2-3.html</title>
  <script>
    let tmp = {
      toString: function() { return 'foo' },
      valueOf: function() { return 123 }
    }
    // default，優先使用 valueOf()
    console.log( tmp == 123 ); // true
    console.log( tmp == 'foo' ); // false

    // number，優先使用 valueOf()
    console.log( +tmp == 123 ); // true
    console.log( +tmp == 'foo' ); // false

    // string，優先使用 toString()
    console.log( `${tmp}` == 123 ); // false
    console.log( `${tmp}` == 'foo' ); // true
  </script>
</head>
<body>

</body>
</html>
```

補充說明

date 物件是個例外，當我們宣告 date 實體的時候，預設會以字串作為預設首選類型。

```
let date = new Date();
console.log(date);
```

```
> let date = new Date();
  console.log(date);
Tue Mar 24 2020 17:40:56 GMT+0800 (台北標準時間)
```

(圖) 宣告 date 物件時，預設值以字串作為回傳結果

有個問題來了，若是單純直接進行 `{ } + []`，而非透過 `console.log` 輸出，它應該是什麼？

說明

如果 `{ }` (空物件) 在前面，而 `[]` (空陣列) 在後面時，前面那個會被認為是區塊而不是物件。

所以 `{ } + []` 相當於 `+ []` 語句，也就是相當於強制求出數字值的 `Number([])` 運算，相當於 `Number("")` 運算，最後得出的是 0 數字。

```
> { } + [ ]
< 0
```

(圖) 被視為 `+ []` 的結果，會輸出 0

參考資料：

JS 中的 `{ } + { }` 與 `{ } + []` 的結果是什麼？

<https://eddychang.me/js-object-plus-object/>

那麼 `===` 運算子呢？一切變得簡單多了。

類型 (x)	類型 (y)	結果
數字	x 和 y 數值相同(非 NaN)	true
字串	x 和 y 是相同的字元	true
布林值	x 和 y 都是 true 或 false	true
物件	x 和 y 引用同一個物件	true

什麼是「x 和 y 引用同一個物件」？

範例

```
let obj1 = {age: 20};
let obj2 = obj1;
console.log(obj1 === obj2); // 輸出 true
```

牛刀小試

// 請問下列這四行的執行結果會得到什麼？

```
[] + []  
[] + {}  
{ } + []  
{ } + { }
```

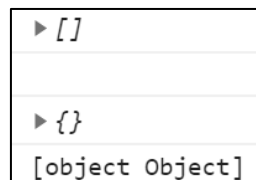
範例 1-2-4.html

```
<html>  
<head>  
  <title>範例 1-2-4.html</title>  
  <script>  
    console.log( [] + [] );  
    console.log( [] + {} );  
    console.log( {} + [] );  
    console.log( {} + {} );  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

說明

第一步先確認 [] 及 {} 的 valueOf 與 toString 分別是什麼：

```
[].valueOf() // []，回傳自己  
[].toString() // ""  
{ }.valueOf() // { }，回傳自己  
{ }.toString() // "[object Object]"
```



(圖) 確認結果

第一個是 [] + []，由於沒有指定 PreferredType，預設會呼叫 valueOf 方法轉型，但回傳的是本身，仍是物件，還沒轉變為基本類型；接著嘗試呼叫 toString 方法進行轉型，得到的是 "" 空字串，兩個空字串相加後仍是空字

串。

第二個是 `[] + {}`，同樣的先嘗試呼叫兩者的 `valueOf`，都仍然得到物件，還沒轉變為基本類型；接著呼叫 `toString`，得到 `"" + [object Object]`，最後答案是 `[object Object]`。

第三個 `{ } + []`，同樣的先嘗試呼叫兩者的 `valueOf`，都仍然得到物件，還沒轉變為基本類型；接著呼叫 `toString`，得到 `[object Object] + ""`，最後答案是 `[object Object]`。

第四個 `{ } + { }`，同樣的先嘗試呼叫兩者的 `valueOf`，都仍然得到物件，還沒轉變為基本類型，把兩個物件 `toString` 後加起來（合併起來），得到 `[object Object][object Object]`。

<code>[object Object]</code>
<code>[object Object]</code>
<code>[object Object][object Object]</code>

(圖) 牛刀小試的結果

邏輯運算子

布林提供了 `true` (真) 與 `false` (假) 的判斷值，透過邏輯運算子（`&&`、`||`、`!`）來進行操作。

範例

`&&` (and 符號)：

幼兒園學生出門前，檢查是否有鞋子跟背包，兩個東西都很重要；全部為真，結果才為真。

```
let hadShoes = true;
let hadBackpack = false;
console.log(hadShoes && hadBackpack);
//輸出 false，代表還沒準備好出門
```

若是把 `hadBackpack` 的值，改成 `true`

```
let hadShoes = true;
let hadBackpack = true;
```

```
console.log(hadShoes && hadBackpack);  
//輸出 true，代表準備好了，可以出門囉！
```

範例

|| (or 符號)：

幼兒園學生出門前，選擇水果作為飯後甜點，至少帶一樣；一個為真，結果為真。

```
let hasApple = true;  
let hasBanana = false;  
console.log(hasApple || hasBanana);  
// 輸出 true
```

範例

! (not 符號)：

假設現在是週末，那麼，我們不需要工作整天；真變假、假變真。

```
let isWeekend = true;  
let workAllDay = !isWeekend;  
console.log(workAllDay);  
// 輸出 false，代表不需要工作整天
```

範例 1-2-5.html

```
<html>  
<head>  
  <title>範例 1-2-5.html</title>  
  <script>  
    let hadShoes = true;  
    let hadBackpack = false;  
    console.log(hadShoes && hadBackpack);  
    //輸出 false，代表還沒準備好出門  
  
    let hasApple = true;  
    let hasBanana = false;  
    console.log(hasApple || hasBanana);  
    // 輸出 true  
  
    let isWeekend = true;
```

```
    let workAllDay = !isWeekend;
    console.log(workAllDay);
    // 輸出 false，代表不需要工作整天
  </script>
</head>
<body>

</body>
</html>
```

賦值運算子的種類，常見的有以下幾種，提供給大家參考：

名稱	簡化後運算子	說明
賦值	$x = y$	$x = y$
加法賦值	$x += y$	$x = x + y$
減法賦值	$x -= y$	$x = x - y$
乘法賦值	$x *= y$	$x = x * y$
除法賦值	$x /= y$	$x = x / y$
餘數賦值	$x \% = y$	$x = x \% y$
指數賦值	$x ** = y$	$x = x ** y$
左移賦值	$x << = y$	$x = x << y$
右移賦值	$x >> = y$	$x = x >> y$
無號右移賦值	$x >>> = y$	$x = x >>> y$
位元 AND 賦值	$x \& = y$	$x = x \& y$
位元 XOR 賦值	$x \wedge = y$	$x = x \wedge y$
位元 OR 賦值	$x = y$	$x = x y$

```
範例 1-2-6.html
<html>
<head>
  <title>範例 1-2-6.html</title>
  <script>
    //宣告變數
    let x = 20;
    let y = 10;

    //輸出初始值
    console.log(x, y);
```

```
//將 x 加上 y 的結果，再賦值到 x 當中
x += y; //可以寫成 x = x + y;
console.log(x);

//變數初始化
x = 2, y = 8;
y *= x; //可以寫成 y = y * x;
console.log(y);

//變數初始化
x = 15, y = 4;
x %= y; //可以寫成 x = x % y;
console.log(x);
</script>
</head>
<body>

</body>
</html>
```

以下表格提供給大家參考，無須特別記憶，常用的幾個運算子先後順序的概念（例如先乘除、後加減）有印象，其它透過未來實作或是工作上用到，再了解即可。以下表格參考網址：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence，表格愈上面的優先權愈高，愈下面優先權愈低：

運算子的型態	相依性	運算子用法
群組	n/a	(...)
成員存取	left-to-right
計算的成員存取	left-to-right	... [...]
new(帶有參數)	n/a	new ... (...)
函式呼叫	left-to-right	... (...)
new(不帶參數)	right-to-left	new ...
後綴增加	n/a	... ++

後綴減少		... --
邏輯 NOT	right-to-left	! ...
位元 NOT		~ ...
一元運算的加號		+ ...
一元運算的負號		- ...
前綴增加		++ ...
前綴減少		-- ...
typeof		typeof ...
void		void ...
delete		delete ...
await		await ...
指數運算	right-to-left	... ** ...
乘法運算	left-to-right	... * ...
減法運算		... / ...
餘數運算		... % ...
加法運算	left-to-right	... + ...
減法運算		... - ...
位元左移	left-to-right	... << ...
位元右移		... >> ...
位元無號右移		... >>> ...
小於	left-to-right	... < ...
小於等於		... <= ...
大於		... > ...
大於等於		... >= ...
in		... in ...
instanceof		... instanceof ...

等於	left-to-right	$\cdots == \cdots$
不等於		$\cdots != \cdots$
嚴格等於		$\cdots === \cdots$
嚴格不等於		$\cdots !== \cdots$
位元 AND	left-to-right	$\cdots \& \cdots$
位元 XOR	left-to-right	$\cdots \wedge \cdots$
位元 OR	left-to-right	$\cdots \cdots$
邏輯 AND	left-to-right	$\cdots \&\& \cdots$
邏輯 OR	left-to-right	$\cdots \cdots$
條件式	right-to-left	$\cdots ? \cdots : \cdots$
賦值	right-to-left	$\cdots = \cdots$
		$\cdots += \cdots$
		$\cdots -= \cdots$
		$\cdots **= \cdots$
		$\cdots *= \cdots$
		$\cdots /= \cdots$
		$\cdots \%= \cdots$
		$\cdots <<= \cdots$
		$\cdots >>= \cdots$
		$\cdots >>>= \cdots$
		$\cdots \&= \cdots$
		$\cdots \wedge= \cdots$
		$\cdots = \cdots$
yield	right-to-left	yield \cdots
yield*		yield* \cdots
逗號 / 序列	left-to-right	\cdots , \cdots

1-3: 流程控制

我們把條件與迴圈，稱之為「流程控制」或「控制結構」，對於任何程式，都扮演著重要的角色。它們讓你定義的特定條件，控制在何時、何種頻率來執行哪些部分的程式碼。

選擇敘述

if 陳述句、if ... else 陳述句、if...else 陳述句鏈，它是依據條件做二選一區塊執行時，所使用的陳述句：

範例

```
let condition = true;
if(condition) {
    console.log('Hello World!');
}
//輸出 Hello World!
```

範例

```
let condition = false;
if(condition) {
    //若條件為真，則執行這個區塊的程式碼
    console.log('Hello World!');
} else {
    //若條件為假，則執行這個區塊的程式碼
    console.log('May you have a nice day!');
}
//輸出 May you have a nice day!
```

範例

```
let number = 7;
if(number > 10) {
    console.log(1);
} else if(number < 3){
    console.log(2);
} else if(number > 5) {
    console.log(3);
}
```

```
} else if(number === 6) {  
    console.log(4);  
}
```

下面的情境，請問輸出是多少？

```
let number = 7;  
if(number > 10) {  
    console.log(1);  
} else if(number === 7){  
    console.log(2);  
} else if(number > 5) {  
    console.log(3);  
} else if(number === 6) {  
    console.log(4);  
}  
// 輸出 2
```

if ... else 陳述句鏈會依序進行判斷，縱然有多個判斷符合條件，依然會以第一個符合條件的區塊來執行。

補充說明

當 if 判斷只需要單獨執行一行，可以不用加「{...}」

```
let num = 10;  
if(num === 8) console.log("It's 8.");  
if(num === 9) console.log("It's 9.");  
if(num === 10) console.log("It's 10.");  
  
//輸出 It' s 10.
```

補充說明

三元條件運算子 (?:)

(condition) ? 條件為真才執行 : 條件為假才執行;

```
(3 > 2) ? console.log('true') : console.log('false');  
// 輸出 true
```

```
(1 > 2) ? console.log('true') : console.log('false');  
// 輸出 false
```

可以將判斷結果帶到變數中

```
let bool = (1 > 2) ? true : false;  
console.log(bool);  
// 輸出 false  
let x = 3;  
let y = 4;  
let answer = (x > y) ? 'x is greater than y' : 'x is less than y';  
console.log(answer);  
// 輸出 x is less than y
```

範例 1-3-1.html

```
<html>  
<head>  
  <title>範例 1-3-1.html</title>  
  <script>  
    let condition = false;  
    if(condition) {  
      //若條件為真，則執行這個區塊的程式碼  
      console.log('Hello World!');  
      document.write('Hello World!');  
    } else {  
      //若條件為假，則執行這個區塊的程式碼  
      console.log('May you have a nice day!');  
      document.write('May you have a nice day!');  
    }  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

範例 1-3-2.html

```
<html>
```

```
<head>
  <title>範例 1-3-2.html</title>
  <script>
    let number = 7;
    if(number > 10) {
      console.log(1);
    } else if(number === 7){
      console.log(2);
    } else if(number > 5) {
      console.log(3);
    } else if(number === 6) {
      console.log(4);
    }
  </script>
</head>
<body>

</body>
</html>
```

範例 1-3-3.html

```
<html>
<head>
  <title>範例 1-3-3.html</title>
  <script>
    let bool = (1 > 2) ? true : false;
    console.log(bool);
    // 輸出 false

    let x = 3;
    let y = 4;
    let answer = (x > y) ? 'x is greater than y' : 'x is less than y';
    console.log(answer);
    // 輸出 x is less than y
  </script>
</head>
<body>
```

```
</body>
</html>
```

switch 判斷

範例

```
let number = 7;
switch(number){
  case '7':
    console.log('string 7');
    break;

  case 7:
    console.log('number 7');
    break;

  case 'number':
    console.log('string number');
    break;

  default:
    console.log('string default');
}
```

// 輸出 number 7

```
let number = 7;
switch(number){
  case '7':
  case 7:
  case 'number':
    console.log('number');
    break;

  default:
    console.log('default');
}
```

// 輸出 number

範例 1-3-4.html

```
<html>
<head>
  <title>範例 1-3-4.html</title>
  <script>
    let number = 7;
    switch(number){
      case '7':
        console.log('string 7');
        break;

      case 7:
        console.log('number 7');
        break;

      case 'number':
        console.log('string number');
        break;

      default:
        console.log('string default');
    }
    // 輸出 number 7
  </script>
</head>
<body>

</body>
</html>
```

範例 1-3-5.html

```
<html>
<head>
  <title>範例 1-3-5.html</title>
  <script>
    let number = 7;
    switch(number){
      case '7':
```

```

        case 7:
        case 'number':
            console.log('number');
            break;

        default:
            console.log('default');
    }
    // 輸出 number
</script>
</head>
<body>

</body>
</html>

```

迴圈

while 迴圈結合了條件判斷的概念，符合條件，則繼續執行區塊內的程式，直到條件不成立，才跳出程式區塊。以下提供範例來說明：

while 迴圈

範例

印出 1 到 7

```
let count = 1;
```

```
while(count <= 7){
```

```
    console.log(count);
```

```
    count++;
```

```
}
```

// 輸出 1 2 3 4 5 6 7

// count 遞增到 8 的時候，count <= 7 的條件不成立，則跳出 while() {…}

另一種 while 迴圈使用形式 do{…}while()

```
let count = 1;
```

```
do{
```

```
    console.log(count);
```

```
    count++;
```

```
}
```

```
while(count <= 7)
```



```
// 輸出 1 2 3 4 5 6 7，與 while(){...} 差異在於 do 區塊會先執行完，才進行 while 判斷
```

補充說明

無限迴圈：

```
while(1){  
    console.log('running');  
}
```

```
console.log('Done');
```

//會不斷輸出 running，沒有輸出 Done 的時候，我們需要使用 Ctrl + C 來終止程式運作。

通常 while(1){...} 的程式設計方式，會建立起類似 Listener 的程式，隨時監聽資料的接受狀態，例如 Socket 網路程式設計。

範例 1-3-6.html

```
<html>  
<head>  
    <title>範例 1-3-6.html</title>  
    <script>  
        let count = 1;  
        while(count <= 7){  
            console.log(count);  
            count++;  
        }  
        // 輸出 1 2 3 4 5 6 7  
    </script>  
</head>  
<body>  
  
</body>  
</html>
```

範例 1-3-7.html

```
<html>  
<head>  
    <title>範例 1-3-7.html</title>  
    <script>
```

```

    let count = 1;
    do{
        console.log(count);
        count++;
    }
    while(count <= 7)
    // 輸出 1 2 3 4 5 6 7，與 while(){...} 差異在於 do 區塊會先執行完，才進行 while 判斷
</script>
</head>
<body>

</body>
</html>

```

for 迴圈

範例

for 迴圈起手式

```

for(setup; condition; increment) {
    //statements
}

```

setup 是變數初始宣告的地方

condition 是變數狀態與判斷條件

increment 是變數賦值的方式

```

for(let i = 0; i < 10; i++) {
    console.log("The value is " + i);
}
// 輸出 The value is 0 ... The value is 9

```

迴圈內部也可以使用迴圈，通稱為「巢狀迴圈」

```

for(let i = 1; i <= 9; i++) {
    for(let j = 1; j <= 9; j++){
        console.log(i + ' * ' + j + ' = ' + (i*j));
    }
}

```

```
}
```

由於 `console.log()` 有尾端斷行的特性，所以我們使用「`process.stdout.write()`」，來達到輸出卻不斷行的效果。

for 的無限迴圈形式：

```
for(;;){  
    console.log('Hello World!');  
}
```

範例 1-3-8.html

```
<html>  
<head>  
    <title>範例 1-3-8.html</title>  
    <script>  
        for(let i = 0; i < 10; i++) {  
            console.log('The value is ' + i);  
        }  
        // 輸出 The value is 0 .... The value is 9  
    </script>  
</head>  
<body>  
  
</body>  
</html>
```

範例 1-3-9.html

```
<html>  
<head>  
    <title>範例 1-3-9.html</title>  
    <script>  
        for(let i = 1; i <= 9; i++) {  
            for(let j = 1; j <= 9; j++){  
                //在 Console 面板輸出結果  
                console.log(i + ' * ' + j + ' = ' + (i*j));  
            }  
        }  
    }  
}
```

```

    for(let i = 1; i <= 9; i++) {
        for(let j = 1; j <= 9; j++){
            //在網頁輸出結果
            document.write(i + ' * ' + j + ' = ' + (i*j) + '&nbsp;');
        }
        document.write('<br />');
    }
</script>
</head>
<body>

</body>
</html>

```

補充說明

迴圈也有 block-scoped

在 for 迴圈以 var 關鍵字宣告 i

```

for(var i = 0; i < 10; i++){
    console.log(i);
}

```

console.log(i);

//輸出 0, 1, ... , 8, 9, 10

從上面的例子可以發現，i 被抬升（Hoisting）到 for(){...} 外的上一行了，縱然迴圈執行完，變數 i 理應在 {...} 結束時被消滅，卻因為變數抬升的關係，導致 for(){...} 以外的作用域，可以存取到變數 i。

在 for 迴圈以 let 關鍵字宣告 i

```

for(let i = 0; i < 10; i++){
    console.log(i);
}

```

console.log(i);

// 拋出錯誤訊息 ReferenceError: i is not defined

因為在這裡用 let 關鍵字宣告的 i，不會被抬升到 for(){...} 區塊的外部，所

以變數 `i` 不會被外部存取，可以確實掌握變數的生命週期。

補充說明

「迭代器 (Iterator)」

一般走訪 `for` 迴圈，是透過來實現迴圈：

```
let items = [1,2,3,4,5];
for(let i = 0; i < items.length; i++){
    console.log(`The item is ${items[i]}`);
}
//輸出 The item is 1
//輸出 The item is 2
//輸出 The item is 3
//輸出 The item is 4
//輸出 The item is 5
```

然而 `Iterator` 簡化了這個過程。

我們在下面模擬一個自訂的 `Iterator`

```
function createIterator(items){
    let i = 0;
    return {
        next: function(){
            let done = (i >= items.length);
            let value = done ? undefined : items[i++];
            return {
                done: done,
                value: value
            }
        }
    }
}
```

```
let iterator = createIterator([1, 2, 3]);
```

```
console.log(iterator.next()); //輸出 { done: false, value: 1 }
console.log(iterator.next()); //輸出 { done: false, value: 2 }
```

```
console.log(iterator.next()); //輸出 { done: false, value: 3 }  
console.log(iterator.next()); //輸出 { done: true, value: undefined }  
console.log(iterator.next()); //輸出 { done: true, value: undefined }
```

迭代器透過類似指標的方式，在每一次 next() 後，將 key 自動往下遞增計算，直到 done 為 true，迴圈即執行完畢。

補充說明

Iterable Proctol

必須有一個 [@@iterator] 屬性，並且回傳一個 iterator

Iterator Proctol

必須有一個 next 屬性，呼叫該屬性每次必須回傳一個 { value: any, done: boolean } 的物件

break 和 continue

通常我們使用 break 來停止、跳出迴圈運作，直接往 for 或 while 迴圈 block 結尾以後的程式區塊繼續執行；使用 continue 直接跳往下一個索引值的步驟繼續執行。

範例 1-3-10.html

```
<html>  
<head>  
  <title>範例 1-3-10.html</title>  
  <script>  
    //i 到 4 的時候，跳出迴圈  
    for(let i = 1; i <= 9; i++) {  
      if(i == 4){  
        break;  
      }  
      console.log(`i = ${i}`);  
    }  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

範例 1-3-11.html

```
<html>
<head>
  <title>範例 1-3-11.html</title>
  <script>
    /**
     * j 為 4 的時候，只跳出內部迴圈，
     * 但外部迴圈依然會繼續執行，
     * 只有內部迴圈的 j 走到 4 時候，
     * 自行跳出
     */
    for(let i = 1; i <= 9; i++) {
      for(let j = 1; j <= 9; j++){
        if(j == 4){
          break;
        }
        console.log(`i = ${i}, j = ${j}`);
      }
    }
  </script>
</head>
<body>

</body>
</html>
```

範例 1-3-12.html

```
<html>
<head>
  <title>範例 1-3-12.html</title>
  <script>
    let count = 0;
    while(count <=7){
      //count 遞增到 5 的時候，跳出迴圈
      if(count == 5){ break; }
      console.log(`count = ${count}`);
      count++; //千萬記得要遞增，不然會變成無限迴圈
    }
  </script>
</head>
<body>

</body>
</html>
```

```
    </script>
</head>
<body>

</body>
</html>
```

範例 1-3-13.html

```
<html>
<head>
  <title>範例 1-3-13.html</title>
  <script>
    //i 為 4 的時候略過，直接往下一個 i (即是 5) 繼續執行
    for(let i = 1; i <= 9; i++) {
      if(i == 4){ continue; }
      console.log(`i = ${i}`);
    }
  </script>
</head>
<body>

</body>
</html>
```

Module 2. 基本集合類型

2-1: Object

Object 類型的特點

物件和陣列很類似，但是物件使用字串作為屬性來存取不同元素，而非數字。這個字串，叫作「鍵」(Key) 或是屬性 (property)，它所指向的元素叫作「值」(Value)。通常把這兩個合在一起，稱為「鍵值對」(Key-Value pair)，最主要的目的，在於儲存更多特定對象的資訊

建立物件

範例

物件的宣告：

```
let obj = {};
```

也有人這麼寫

```
let obj = new Object();
```

Object 表達式

範例

我們會以 {key01: value01, key02: value02, ..., keyN: valueN}

例如

```
{name: 'bill', age: 25, id: 'A001'}
```

若是對物件進行簡的排版，可能會長成

```
{  
  key01: value01,  
  key02: value02,  
  ...,  
  keyN: valueN  
}
```

例如

```
{  
  name: 'bill',  
  age: 25,  
  id: 'A001'  
}
```

自訂物件屬性

```
let cat = {  
  legs: 4,  
  name: 'Darren',  
  color: 'Tangerine',  
  ability: {  
    jump: function(){  
      console.log('Jump!');  
    },  
  },  
}
```

```

        sound: function(){
            console.log('Meow~');
        },
        sleep: function() {
            console.log('ZZZzzz...');
        }
    }
};

```

有些人會對物件屬性，以字串型式呈現

```

let cat = {
    'legs': 4,
    'name': 'Darren',
    'color': 'Tangerine',
    'ability': {
        'jump': function(){
            console.log('Jump!');
        },
        'sound': function(){
            console.log('Meow~');
        },
        'sleep': function() {
            console.log('ZZZzzz...');
        }
    }
};

```

存取物件

範例

```

let cat = {
    legs: 4,
    name: 'Darren',
    color: 'Tangerine',
    ability: {
        jump: function(){
            console.log('Jump!');
        },
        sound: function(){

```

```

        console.log('Meow~');
    },
    sleep: function() {
        console.log('ZZZzzz...');
    }
}
};

```

使用「[]」來存取屬性或函式：

```

console.log( cat['legs'] ); // 輸出 4
console.log( cat['name'] ); // 輸出 Darren
cat['ability'].sleep(); // 輸出 ZZZzzz...
cat['ability'].sound(); // 輸出 Meow~

```

使用「.」來存取屬性或函式：

```

console.log( cat.legs ); // 輸出 4
console.log( cat.color ); // 輸出 Tangerine
cat.ability.sound(); // 輸出 Meow~
cat.ability.sleep(); // 輸出 ZZZzzz...

```

增加物件屬性：

```

cat['eye_color'] = 'black';
cat.habbit = 'play with slave';

```

console.dir(cat , {depth: null}); // console.dir 可以列出物件（或陣列）的內容
//輸出：

```

// { legs: 4,
//   name: 'Darren',
//   color: 'Tangerine',
//   ability:
//     { jump: [Function: jump],
//       sound: [Function: sound],
//       sleep: [Function: sleep] },
//   eye_color: 'black',
//   habbit: 'play with slave' }

```

可以使用 delete，來刪除物件的屬性

```
delete cat.color;
```

範例 2-1-1.html

```
<html>
<head>
  <title>範例 2-1-1.html</title>
  <script>
let cat = {
  legs: 4,
  name: 'Darren',
  color: 'Tangerine',
  ability: {
    jump: function(){
      console.log('Jump!');
    },
    sound: function(){
      console.log('Meow~');
    },
    sleep: function() {
      console.log('ZZZzzz...');
    }
  }
};

//使用「[ ]」來存取屬性或函式：
console.log( cat['legs'] ); // 輸出 4
console.log( cat['name'] ); // 輸出 Darren
cat['ability'].sleep(); // 輸出 ZZZzzz...
cat['ability'].sound(); // 輸出 Meow~

//使用「.」來存取屬性或函式：
console.log( cat.legs ); // 輸出 4
console.log( cat.color ); // 輸出 Tangerine
cat.ability.sound(); // 輸出 Meow~
cat.ability.sleep(); // 輸出 ZZZzzz...
```

```
</script>
</head>
<body>

</body>
</html>
```

範例 2-1-2.html

```
<html>
<head>
  <title>範例 2-1-2.html</title>
  <script>
    let cat = {
      legs: 4,
      name: 'Darren',
      color: 'Tangerine',
      ability: {
        jump: function(){
          console.log('Jump!');
        },
        sound: function(){
          console.log('Meow~');
        },
        sleep: function() {
          console.log('ZZZzzz...');
        }
      }
    };

    //增加物件屬性：
    cat['eye_color'] = 'black';
    cat.habbit = 'play with slave';

    // console.dir 可以列出物件（或陣列）的內容
    console.dir( cat , {depth: null} );
  </script>
</head>
<body>
```

```
</body>
</html>
```

補充說明

物件屬性可以使用有「單/雙引號」括住的字串，或是以「點」作為屬性存取的方式，若是選擇使用「點」來存取屬性，若是屬性的文字有「-」、「.」等特殊用途的字，會產生錯誤。

最好的方式，就是在屬性文字當中，有「-」、「.」等字，一定要用「[]」以及「單/雙引號」的格式，來存取屬性的值。

```
let student = {
  id: '10801003',
  name: 'Cook',
  age: 18,
  phone_number: '0933333333',
  'test-test': 'test'
};

console.log( student['test-test'] ); // 輸出 test
```

2-2: Array

Array 類型的特點

在沒有使用陣列的情況下，我們需要這樣記錄資料：

```
let name1 = "Alex";
let name2 = "Bill";
let name3 = "Cook";
let name4 = "Darren";
.
.
.
let name9999 = 'Somebody';
```

上面這種列表會變得很不好用，假設每一個人的名字都需要一張紙來記錄，這要浪費多少紙張？於是我們可以使用類似「清單」概念的陣列，將所有名字都

記錄在同一張紙上，這樣就簡單多了。

建立陣列

範例

陣列的宣告：

```
let arr = [];
```

也有人這麼寫

```
let arr = new Array();
```

Array 表達式

範例

宣告陣列時，建立初始值：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
```

有時候為了排版好看，會將陣列中的元素，透過鍵盤的 Enter，讓每個元素對齊：

```
let listOfName = [  
    'Alex',  
    'Bill',  
    'Cook',  
    'Darren'  
];
```

很重要的觀念是，陣列中每一個值的索引（index），都是從「0」開始。

索引\	0	1	2	3
值	'Alex'	'Bill'	'Cook'	'Darren'
概念	{0: 'Alex'}	{1: 'Bill'}	{2: 'Cook'}	{3: 'Darren'}

Array 的方法

存取陣列

範例

一般來說，陣列的索引，從 [0] 開始，到 [n - 1] 結束。

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];  
console.log( listOfName[0] ); // 輸出 Alex
```

```
console.log( listOfName[3] ); // 輸出 Darren  
console.log( listOfName[4] ); // 輸出 Undefined
```

範例 2-2-1.html

```
<html>  
<head>  
  <title>範例 2-2-1.html</title>  
  <script>  
    //一般來說，陣列的索引，從 [0] 開始，到 [n - 1] 結束。  
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];  
    console.log( listOfName[0] ); // 輸出 Alex  
    console.log( listOfName[3] ); // 輸出 Darren  
    console.log( listOfName[4] ); // 輸出 Undefined  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

設定、修改陣列元素

範例

設定（新增）元素：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];  
listOfName[4] = 'Ellen'; // 指定索引位置來新增元素  
console.log(listOfName); //輸出 [ 'Alex', 'Bill', 'Cook', 'Darren', 'Ellen' ]
```

使用 **push()**，將資料加到陣列尾端：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];  
listOfName.push('Ellen');  
listOfName.push('Fox');  
console.log(listOfName);
```

修改元素：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];  
listOfName[0] = 'Allen';  
listOfName[2] = 'Carl';  
console.log(listOfName[0]); // 輸出 Allen
```



```
console.log(listOfName[2]); // 輸出 Carl
console.log(listOfName); // 輸出 [ 'Allen', 'Bill', 'Carl', 'Darren' ]
```

刪除元素，使用 **pop()**，將會刪除陣列尾端的資料：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
listOfName.pop();
console.log(listOfName); // 輸出 [ 'Alex', 'Bill', 'Cook' ]
listOfName.pop();
console.log(listOfName); // 輸出 [ 'Alex', 'Bill' ]
```

可以透過變數賦值的方式，取得被 **pop()** 刪除的資料：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
let name = listOfName.pop();
console.log(name); // 輸出 Darren
```

另一種刪除元素的方式，使用 **delete**，但會保持原先索引的位置：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
delete listOfName[3];
console.log(listOfName); // 輸出 [ 'Alex', 'Bill', 'Cook', <1 empty item> ]
```

```
for(let i = 0; i < listOfName.length; i++){
  console.log(`index: ${i}, value: ${listOfName[i]}`);
}
// 輸出 index: 0, value: Alex
// 輸出 index: 1, value: Bill
// 輸出 index: 2, value: Cook
// 輸出 index: 3, value: undefined
```

刪除陣列第一個元素，使用 **shift()**：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
listOfName.shift();
console.log(listOfName); // 輸出 [ 'Bill', 'Cook', 'Darren' ]
```

有時候從尾端刪除的陣列資料，需要放在陣列前端，使用 **unshift()**：

```
let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
let name = listOfName.pop(); // 從陣列尾端刪除 Darren
listOfName.unshift(name); // 將刪除的陣列尾端資料放到陣列前端
console.log(listOfName); // 輸出 [ 'Darren', 'Alex', 'Bill', 'Cook' ]
```

範例 2-2-2.html

```
<html>
<head>
  <title>範例 2-2-2.html</title>
  <script>
    //設定（新增）元素：
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    listOfName[4] = 'Ellen'; // 指定索引位置來新增元素
    console.log(listOfName); //輸出 [ 'Alex', 'Bill', 'Cook', 'Darren', 'Ellen' ]
  </script>
</head>
<body>

</body>
</html>
```

範例 2-2-3.html

```
<html>
<head>
  <title>範例 2-2-3.html</title>
  <script>
    //使用 push()，將資料加到陣列尾端：
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    listOfName.push('Ellen');
    listOfName.push('Fox');
    console.log(listOfName);
    //輸出 ["Alex", "Bill", "Cook", "Darren", "Ellen", "Fox"]
  </script>
</head>
<body>

</body>
</html>
```

範例 2-2-4.html

```
<html>
```

```

<head>
  <title>範例 2-2-4.html</title>
  <script>
    //修改元素：
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    listOfName[0] = 'Allen';
    listOfName[2] = 'Carl';
    console.log(listOfName[0]); // 輸出 Allen
    console.log(listOfName[2]); // 輸出 Carl
    console.log(listOfName); // 輸出 [ 'Allen', 'Bill', 'Carl', 'Darren' ]
  </script>
</head>
<body>

</body>
</html>

```

範例 2-2-5.html

```

<html>
<head>
  <title>範例 2-2-5.html</title>
  <script>
    //刪除元素，使用 pop()，將會刪除陣列尾端的資料：
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    listOfName.pop();
    console.log(listOfName); // 輸出 [ 'Alex', 'Bill', 'Cook' ]
    listOfName.pop();
    console.log(listOfName); // 輸出 [ 'Alex', 'Bill' ]

    //重新初始化陣列
    listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    let name = listOfName.pop();
    console.log(name); // 輸出 Darren
  </script>
</head>
<body>

</body>

```

```
</html>
```

範例 2-2-6.html

```
<html>
<head>
  <title>範例 2-2-6.html</title>
  <script>
    //刪除陣列第一個元素，使用 shift():
    let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    listOfName.shift();
    console.log(listOfName); // 輸出 [ 'Bill', 'Cook', 'Darren' ]

    //有時候從尾端刪除的陣列資料，需要放在陣列前端，使用 unshift():
    listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];
    let name = listOfName.pop(); // 從陣列尾端刪除 Darren
    listOfName.unshift(name); // 將刪除的陣列尾端資料放到陣列前端
    console.log(listOfName); // 輸出 [ 'Darren', 'Alex', 'Bill', 'Cook' ]
  </script>
</head>
<body>

</body>
</html>
```

補充說明

二維陣列：

```
let arr = [
  ['a0', 'a1', 'a2', 'a3'],
  ['b0', 'b1', 'b2'],
  ['c0', 'c1', 'c2', 'c3', 'c4'],
];

console.log( arr[0][1] ); // 輸出 a1
console.log( arr[2][4] ); // 輸出 c4
```

它的概念如下表格：

二維陣列	0	1	2	3	4
0	a0	a1	a2	a3	
1	b0	b1	b2		
2	c0	c1	c2	c3	c4

左側索引代表每一列的陣列資料，上方索引代表每一列當中特定欄位的位置。

建立二維陣列：

```
let arr1d = [];
for(let i = 1; i <= 9; i++){
    //先建立一維陣列
    arr1d.push(i);
}

let arr2d = [];
for(let j = 1; j <= 9; j++){
    //連續新增先前建立的一維陣列，便可成為二維陣列
    arr2d.push(arr1d);
}
```

```
console.log(arr2d);
```

輸出:

```
[
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
  [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
]
```

範例 2-2-7.html

```
<html>
<head>
  <title>範例 2-2-7.html</title>
  <script>
    //二維陣列
    let arr = [
      ['a0', 'a1', 'a2', 'a3'],
      ['b0', 'b1', 'b2'],
      ['c0', 'c1', 'c2', 'c3', 'c4'],
    ];

    console.log( arr[0][1] ); // 輸出 a1
    console.log( arr[2][4] ); // 輸出 c4
  </script>
</head>
<body>

</body>
</html>
```

範例 2-2-8.html

```
<html>
<head>
  <title>範例 2-2-8.html</title>
  <script>
    //建立二維陣列：
    let arr1d = [];
    for(let i = 1; i <= 9; i++){
      //先建立一維陣列
      arr1d.push(i);
    }

    let arr2d = [];
    for(let j = 1; j <= 9; j++){
      //連續新增先前建立的一維陣列，便可成為二維陣列
      arr2d.push(arr1d);
    }
  </script>
</head>
<body>

</body>
</html>
```

```

        console.log(arr2d);
    </script>
</head>
<body>

</body>
</html>

```

陣列混合其它資料類別

```

let arr = [
    3,
    'Darren',
    ['aaa', 'bbb', 3.14],
    10
];
console.log( arr[0] ); // 輸出 3
console.log( arr[2][2] ); // 輸出 3.14
console.log( arr[3] ); // 輸出 10

```

2-3: for/in 迴圈

for 迴圈的形式，常見還有其它幾種。若是要取得物件/陣列當中的「索引（index）/鍵（key）」，可以使用「for(property/key/index in dataSet) {…}」（就是 for/in 可以查看 Object 類型物件的屬性和屬性值）。

補充說明

```

let obj = {
    fname: 'Darren',
    lname: 'Yang',
    age: null,
    lineId: 'telunyang'
};
for(let attr in obj) {
    console.log(`The property in object variable is ${attr}`);
}
//輸出 The property in object variable is fname
//輸出 The property in object variable is lname

```

```
//輸出 The property in object variable is age
//輸出 The property in object variable is lineId

let arr = [
  'a',
  'b',
  'c'
];
for(let key in arr){
  console.log(`The index number in array variable is ${key}`);
}
//輸出 The index number in array variable is 0
//輸出 The index number in array variable is 1
//輸出 The index number in array variable is 2
```

若是要取得**陣列**當中的「值」，可以使用「for(value of dataSet) {…}」：

```
let arr = [
  'a',
  'b',
  'c'
];
for(let value of arr){
  console.log(`The value in array variable is ${value}`);
}
```

但是取得**物件**當中的值，則無法使用「for(value of dataSet) {…}」，因為 dataSet 是需要可以迭代的（Iterable），是指變數是可以透過 [0], [1], [2], … , [n] 方式來取得相對應的值，物件內部成員以**屬性（Property）**的方式存在，沒有「有序的鍵（Key）」，故無法迭代：

```
let obj = {
  fname: 'Darren',
  lname: 'Yang',
  age: null,
  lineId: 'telunyang'
};

for(let value of obj){
```



```
    console.log(`The value in object variable is ${value}`);  
  }  
  // 拋出錯誤 TypeError: obj is not iterable
```

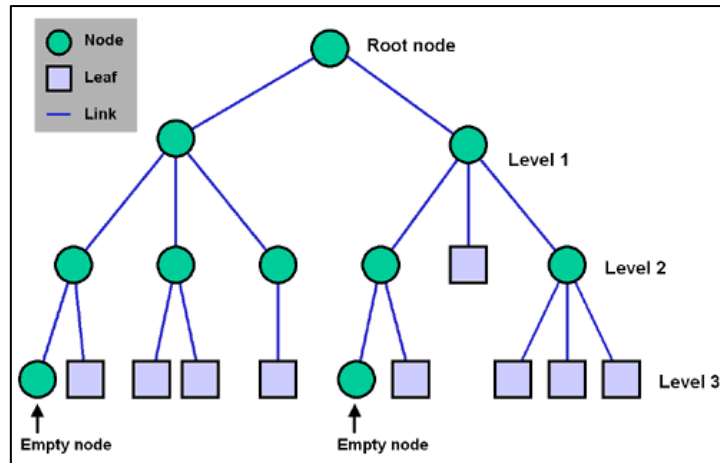
範例 2-3-1.html

```
<html>  
<head>  
  <title>範例 2-3-1.html</title>  
  <script>  
    let obj = {  
      fname: 'Darren',  
      lname: 'Yang',  
      age: null,  
      lineId: 'telunyang'  
    };  
  
    //查看所有屬性  
    for(let attr in obj) {  
      console.log(`The property in object variable is ${attr}`);  
    }  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

Module 3. 操作 DOM

3-1: 取得 DOM 元素

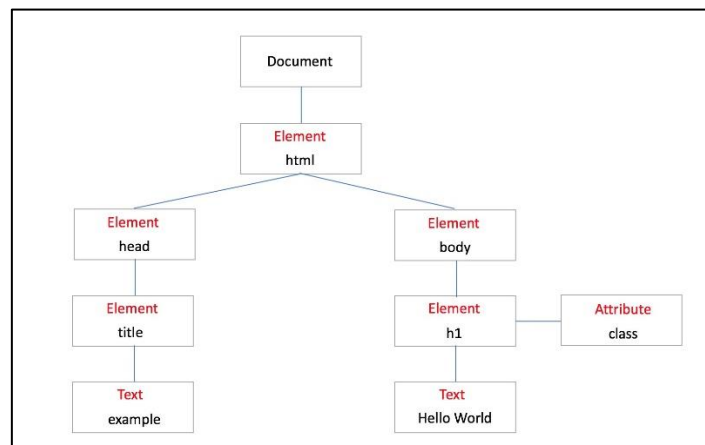
我們來解析一下 DOM。



(圖) DOM 的樹狀結構，裡面是由節點、樹葉、連結所組成

在 DOM 中，每個元素(element)、文字(text) 等等都是一個節點(node)，而節點通常分成以下四種：

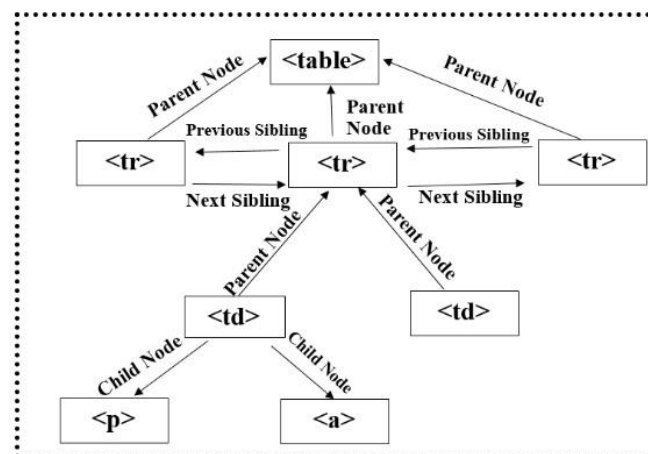
- Document
 - Document 就是指這份文件，也就是這份 HTML 檔的開端，所有的一切都會從 Document 開始往下進行。
- Element
 - Element 就是指文件內的各個標籤，因此像是 <div>、<p> 等等各種 HTML Tag 都是被歸類在 Element 裡面。
- Text
 - Text 就是指被各個標籤包起來的文字，舉例來說在 <h1>Hello World</h1> 中，Hello World 被 <h1> 這個 Element 包起來，因此 Hello World 就是此 Element 的 Text
- Attribute
 - Attribute 就是指各個標籤內的相關屬性，例如 class、id、data-* 等。



(圖) DOM 圖解

由於 DOM 為樹狀結構，樹狀結構最重要的觀念就是 Node 彼此之間的關係，這邊可以分成以下兩種關係：

- 父子關係(Parent and Child)
 - 簡單來說就是上下層節點，上層為 Parent Node，下層為 Child Node。
- 兄弟關係(Siblings)
 - 簡單來說就是同一層節點，彼此間只有 Previous 以及 Next 兩種。



(圖) table 元素 - 節點間的關係

補充說明

window 物件（可以想成瀏覽器本身）代表了一個包含 DOM 文件的視窗，其中的 document 屬性指向了視窗中載入的 Document 物件：

- 代表所有在瀏覽器中載入的網頁，也是作為網頁內容 DOM 樹（包含如 <body>、<table> 與其它的元素）的進入點。Document 提供了網頁文件所需的通用函式，例如取得頁面 URL 或是建立網頁文件中新的元素節點等。
- 描述了各種類型文件的共同屬性與方法。根據文件的類型（如 HTML、XML、SVG 等），也會擁有各自的 API：HTML 文件（content type 為 text/html）實作了 HTMLDocument 介面，而 XML 及 SVG 文件實作了 XMLDocument 介面。

以下為常用的 DOM API（document）：

- document.getElementById('idName')
 - 找尋 DOM 中符合此 id 名稱的元素，並回傳相對應的 element。
- document.getElementsByTagName('tag')
 - 找尋 DOM 中符合此 tag 名稱的所有元素，並回傳相對應的 element

集合，集合為 `HTMLCollection` 。

- `document.getElementsByTagName('className')`
 - 找尋 DOM 中符合此 class 名稱的所有元素，並回傳相對應的 element 集合，集合為 `HTMLCollection` 。
- `document.querySelector(selectors)`
 - 利用 selector 來找尋 DOM 中的元素，並回傳相對應的第一個 element 。
- `document.querySelectorAll(selectors)`
 - 利用 selector 來找尋 DOM 中的所有元素，並回傳 `NodeList` 。

補充說明

HTMLCollection

集合內元素為 HTML element 。

NodeList

集合內元素為 Node，全部的 Node 存放在 `NodeList` 內。

3-2: 建立 DOM 元素

在先前的介紹中，我們已經理解了 **DOM Node** 的類型、以及節點之間的查找與關係。那麼在今天的介紹裡我們將繼續來說明，如何透過 **DOM API** 來建立新的節點、修改以及刪除節點。

DOM 節點的新增：

API	說明
<code>document.createElement(tagName)</code>	建立一個新的元素
<code>document.createTextNode()</code>	建立文字節點
<code>document.createDocumentFragment()</code>	建立最小化 document 物件（可以視為虛擬容器）
<code>document.write()</code>	將內容寫入網頁

範例 3-2-1.html

```
<html>
<head>
  <title>範例 3-2-1.html</title>
</head>
<body>
```

```

<script>
//建立新的 div 元素 newDiv
var newDiv = document.createElement('div');

//指定屬性
newDiv.id = "myNewDiv";
newDiv.className = "box";

//建立 textNode 文字節點
var textNode = document.createTextNode("Hello world!");

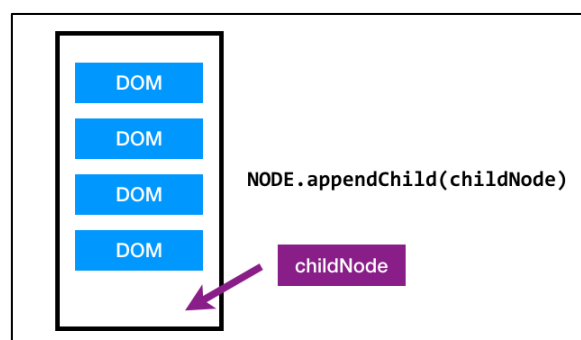
//透過 newDiv.appendChild 將 textNode 加入至 newDiv
newDiv.appendChild(textNode);

//透過 document.body.appendChild 來加入 newDiv 到網頁當中
document.body.appendChild(newDiv);
</script>
</body>
</html>

```

DOM 節點的修改：

API	說明
NODE.appendChild(childNode)	可以將指定的 childNode 節點，加入到 NODE 父容器節點的末端
NODE.insertBefore(newNode, refNode)	將新節點 newNode 新增至指定的 refNode 節點的前面
NODE.replaceChild(newChildNode, oldChildNode)	將原本的 oldChildNode 替換成指定的 newChildNode



(圖) 將指定的 childNode，加入到 NODE 父容器節點的末端

範例 3-2-2.html

```
<html>
<head>
  <title>範例 3-2-2.html</title>
</head>
<body>
  <ul id="myList">
    <li>Item 01</li>
    <li>Item 02</li>
    <li>Item 03</li>
  </ul>

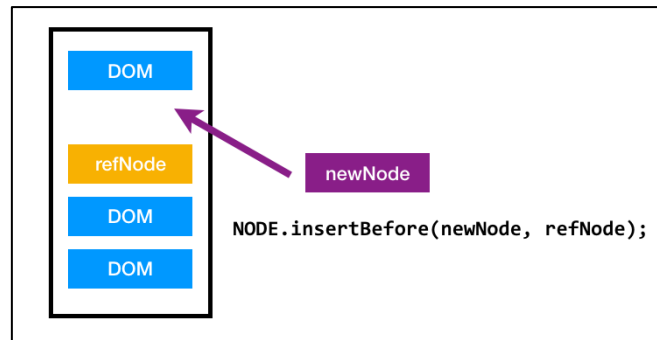
  <script>
    // 取得容器
    var myList = document.getElementById('myList');

    // 建立新的 <li> 元素
    var newList = document.createElement('li');

    // 建立 textNode 文字節點
    var textNode = document.createTextNode("Hello world!");

    // 透過 appendChild 將 textNode 加入至 newList
    newList.appendChild(textNode);

    // 透過 appendChild 將 newList 加入至 myList
    myList.appendChild(newList);
  </script>
</body>
</html>
```



(圖) 將 newNode 新增到指定的 refNode 的前面

範例 3-2-3.html

```
<html>
<head>
  <title>範例 3-2-3.html</title>
</head>
<body>
  <ul id="myList">
    <li>Item 01</li>
    <li>Item 02</li>
    <li>Item 03</li>
  </ul>

  <script>
    // 取得容器
    var myList = document.getElementById('myList');

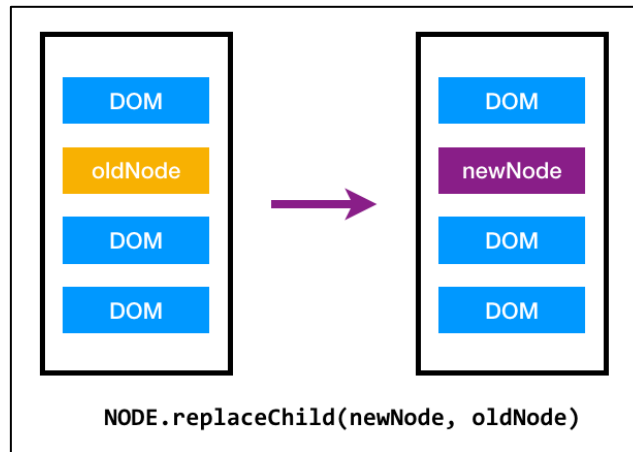
    // 取得 "<li>Item 02</li>" 的元素
    var refNode = document.querySelectorAll('li')[1];

    // 建立 li 元素節點
    var newNode = document.createElement('li');

    // 建立 textNode 文字節點
    var textNode = document.createTextNode("Hello world!");
    newNode.appendChild(textNode);

    // 將新節點 newNode 插入 refNode 的前方
    myList.insertBefore(newNode, refNode);
```

```
</script>
</body>
</html>
```



(圖) 將原本的 oldChildNode 替換成指定的 newChildNode

範例 3-2-4.html

```
<html>
<head>
  <title>範例 3-2-4.html</title>
</head>
<body>
  <ul id="myList">
    <li>Item 01</li>
    <li>Item 02</li>
    <li>Item 03</li>
  </ul>

  <script>
    // 取得容器
    var myList = document.getElementById('myList');

    // 取得 "<li>Item 02</li>" 的元素
    var oldNode = document.querySelectorAll('li')[1];

    // 建立 li 元素節點
    var newNode = document.createElement('li');
```

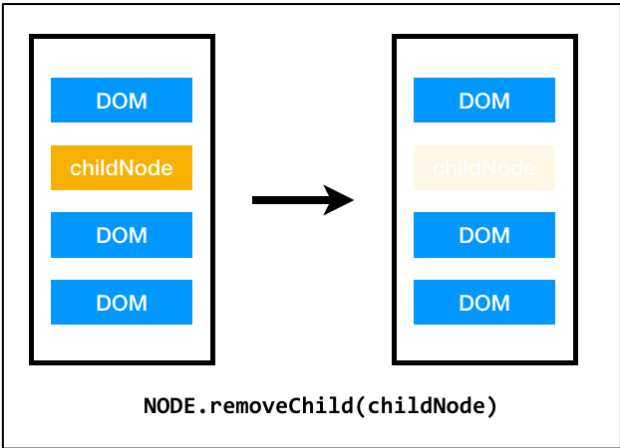


```
// 建立 textNode 文字節點
var textNode = document.createTextNode("Hello world!");
newNode.appendChild(textNode);

// 將原有的 oldNode 替換成新節點 newNode
myList.replaceChild(newNode, oldNode);
</script>
</body>
</html>
```

3-3: 刪除 DOM 元素

API	說明
NODE.removeChild(childNode)	將指定的 childNode 子節點移除



(圖) 將指定的 childNode 移除

```
範例 3-3-1.html
<html>
<head>
  <title>範例 3-3-1.html</title>
</head>
<body>
  <ul id="myList">
    <li>Item 01</li>
    <li>Item 02</li>
    <li>Item 03</li>
```

```
</ul>

<script>
// 取得容器
var myList = document.getElementById('myList');

// 取得 "<li>Item 02</li>" 的元素
var removeNode = document.querySelectorAll('li')[1];

// 將 myList 下的 removeNode 節點移除
myList.removeChild(removeNode);
</script>
</body>
</html>
```

參考資料：
重新認識 JavaScript: Day 13 DOM Node 的建立、刪除與修改
<https://ithelp.ithome.com.tw/articles/10191867>

Module 4. 深入 DOM 元素

4-1: 屬性操作

方法	說明
Element.setAttribute(name, value)	設定指定元素上的某個屬性值。若屬性已經存在，則更新該值；否則，使用指定的名稱和值添加一個新的屬性。
Element.getAttribute(attributeName)	回傳元素所擁有的屬性值，若是指定的屬性不存在，則回傳 null 或 ""。
Element.removeAttribute(attrName)	從指定的元素中，刪除一個屬性。

範例 4-1-1.html
<pre><html> <head> <title>範例 4-1-1.html</title></pre>

```

</head>
<body>
  <button id="btn01">按鈕 01</button>
  <button id="btn02" name="btn02">按鈕 02</button>
  <button id="btn03" onclick="javascript: alert('btn03');">按鈕
03</button>

  <script>
    //取得第一個按鈕元素，並設定屬性
    let btn01 = document.querySelectorAll("button")[0];
    btn01.setAttribute("name", "btn01"); //設定 name 屬性
    btn01.setAttribute("disabled", ""); //設定 disabled 屬性

    //取得第二個按鈕元素，並取得屬性值
    let btn02 = document.querySelectorAll("button")[1];
    let strName = btn02.getAttribute("name");
    document.write(strName);

    //取得第三個按鈕，刪除 onclick 屬性
    let btn03 = document.querySelectorAll("button")[2];
    btn03.removeAttribute("onclick");
  </script>
</body>
</html>

```

4-2: 自訂屬性

HTML5 支援自訂屬性「data-*」，「*」由兩個部分組成：

- 不應該包含任何大寫字母，同時必須至少有 1 個字元在「data-」後面。
- 自訂屬性的值可以是任何字串。

範例 4-2-1.html

```

<html>
<head>
  <title>範例 4-2-1.html</title>
</head>
<body>

```

```

<input type="text" id="txt" value="" />

<script>
//取得文字欄位的元素，並設定自訂屬性
let txt = document.querySelector("input#txt");
txt.setAttribute("name", "txt");
txt.setAttribute("value", "1234");
txt.setAttribute("data-price", "10000");
txt.setAttribute("data-title", "文字欄位");
txt.setAttribute("data-description", "可以輸入任何文字");

//輸出自訂屬性到網頁上
document.write("<hr />");
document.write( txt.getAttribute('data-title') + "<br />" );
document.write( txt.getAttribute('data-price') + "<br />" );
document.write( txt.getAttribute('data-description') + "<br />" );
</script>
</body>
</html>

```

4-3: 元素與樣式

HTMLElement.style 屬性用於取得與設定元素的 inline 樣式，我們可以使用下面的方式來進行設定：

範例

取得 style 屬性值

```
let colorValue = p.style.color;
```

設定 style 屬性值

```
p.style.color = '#ff0000';
```

```
p.style['font-size'] = '80px'; //屬性可以用 css 格式
```

```
p.style['backgroundColor'] = '#CFEE99'; //屬性可以用 javascript 格式
```

範例 4-3-1.html

```
<html>
```

```
<head>
```

```

<title>範例 4-3-1.html</title>
</head>
<body>
  <p style="color: #7878ff; line-height: 3; font-size: 30px;">Hello World<br />Good job!</p>

  <script>
    //取得 p 元素，並取得 style 屬性中的 css 屬性值
    let p = document.getElementsByTagName('p')[0];
    document.write("<br />");
    document.write(p.style.color + "<br />");
    document.write(p.style.lineHeight + "<br />");
    document.write(p.style.fontSize + "<br />");

    //設定 p 的 color 為 #ff0000
    p.style.color = '#ff0000';
    p.style['font-size'] = '80px';
    p.style['backgroundColor'] = '#CFEE99';
  </script>
</body>
</html>

```

以下是 CSS 與 JavaScript 轉換的列表：

CSS	JavaScript
background	background
background-attachment	backgroundAttachment
background-color	backgroundColor
background-image	backgroundImage
background-position	backgroundPosition
background-repeat	backgroundRepeat
border	border
border-bottom	borderBottom
border-bottom-color	borderBottomColor

CSS	JavaScript
border-bottom-style	borderBottomStyle
border-bottom-width	borderBottomWidth
border-color	borderColor
border-left	borderLeft
border-left-color	borderLeftColor
border-left-style	borderLeftStyle
border-left-width	borderLeftWidth
border-right	borderRight
border-right-color	borderRightColor
border-right-style	borderRightStyle
border-right-width	borderRightWidth
border-style	borderStyle
border-top	borderTop
border-top-color	borderTopColor
border-top-style	borderTopStyle
border-top-width	borderTopWidth
border-width	borderWidth
clear	clear
clip	clip
color	color
cursor	cursor
display	display
filter	filter
float	cssFloat

CSS	JavaScript
font	font
font-family	fontFamily
font-size	fontSize
font-variant	fontVariant
font-weight	fontWeight
height	height
left	left
letter-spacing	letterSpacing
line-height	lineHeight
list-style	listStyle
list-style-image	listStyleImage
list-style-position	listStylePosition
list-style-type	listStyleType
margin	margin
margin-bottom	marginBottom
margin-left	marginLeft
margin-right	marginRight
margin-top	marginTop
overflow	overflow
padding	padding
padding-bottom	paddingBottom
padding-left	paddingLeft
padding-right	paddingRight
padding-top	paddingTop

CSS	JavaScript
page-break-after	pageBreakAfter
page-break-before	pageBreakBefore
position	position
stroke-dasharray	strokeDasharray
stroke-dashoffset	strokeDashoffset
stroke-width	strokeWidth
text-align	textAlign
text-decoration	textDecoration
text-indent	textIndent
text-transform	textTransform
top	top
vertical-align	verticalAlign
visibility	visibility
width	width
z-index	zIndex

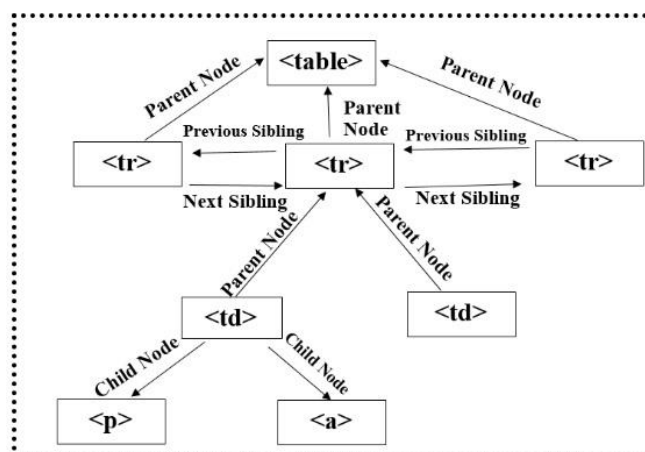
Module 5. DOM 元素集合

5-1: querySelectorAll 與 NodeList

NodeList 物件是節點（Node）的集合，可藉由 Node.childNodes 屬性以及 document.querySelectorAll() 方法來取得：

- document.querySelector(selectors)
 - 利用 selector 來找尋 DOM 中的元素，並回傳相對應的第一個 element。
- document.querySelectorAll(selectors)

- 利用 selector 來找尋 DOM 中的所有元素，並回傳 NodeList。



(圖) table 元素 - 節點間的關係

補充說明

NodeList 不是 Array，它的是節點的集合，所以它無法使用 push、pop 等方法。

範例

```
<html>
<head>
  <title>範例 5-1-1.html</title>
</head>
<body>
  <p style="color: #7878ff; line-height: 3; font-size: 30px;">Hello World<br />Good job!</p>
  <button id="btn01">按鈕 01</button>
  <button id="btn02">按鈕 02</button>
  <button id="btn03">按鈕 03</button>

  <script>
    //取得所有 body 裡面的 nodes
    var children = document.body.childNodes;

    //輸出現有的 nodes 有多少個
    console.log(children.length); // 10

    //新增 p 元素，加入內部文字，再加入到 body 當中
    let p = document.createElement('p');
```

```

let txtNode = document.createTextNode("JavaScript");
p.appendChild(txtNode);
document.body.appendChild(p);

//輸出現有的 nodes 有多少個
console.log(children.length); // 11

//取得 body 底下所有 node 的 iterator
for (var entry of children.entries()) {
    console.log(entry);
}

//取得 p node
let pNode = document.body.childNodes.item(1);
//let pNode = document.body.childNodes[1]; //這種格式也可以用
console.log(pNode);
</script>
</body>
</html>

```

以下是以 NODE.nodeType 值的對照表：

Name	Value
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10

DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

5-2: document 的子物件與 HTMLCollection

HTMLCollection 代表 Element 物件的集合，並提供了選取集成員的方法與屬性：

- document.getElementById('idName')
 - 找尋 DOM 中符合此 id 名稱的元素，並回傳相對應的 element。
- document.getElementsByTagName('tag')
 - 找尋 DOM 中符合此 tag 名稱的所有元素，並回傳相對應的 element 集合，集合為 HTMLCollection。
- document.getElementsByClassName('className')
 - 找尋 DOM 中符合此 class 名稱的所有元素，並回傳相對應的 element 集合，集合為 HTMLCollection。

5-3: NodeList 不是 Array

NodeList 是一個集合的類型，擁有 length 的屬性，但它不是 Array，更不是 Array 的延伸類型。

方法	說明
NodeList.item()	按照索引取得 NodeList 中的一個項目，若是超出範圍，則會回傳 null。可以使用 NodeList[index] 的格式來存取項目。
NodeList.entries()	回傳一個 iterator，允許透過迴圈走訪每一個項目的 key/value。
NodeList.forEach()	按照項目的順序，對 NodeList 中每一個 key/value 提供一個 callback 函式，來進行操作。
NodeList.keys()	回傳一個 iterator，允許透過迴圈走訪所有的 key。
NodeList.values()	回傳一個 iterator，允許透過迴圈走訪所有的 value。

範例 5-3-1.html
<pre><html> <head></pre>

```

<title>範例 5-3-1.html</title>
</head>
<body>
  <p>Hello World</p>
  <p>Good job!</p>
  <button id="btn01">按鈕 01</button>
  <button id="btn02">按鈕 02</button>
  <button id="btn03">按鈕 03</button>

  <script>
    var p = document.getElementsByTagName("p");
    var pGJ = p.item(1); //也可以用 p[1]

    //取得所有 body 裡面的 nodes
    var children = document.body.childNodes;

    //取得 body 底下所有 node 的 iterator
    for(var entry of children.entries()) {
      console.log(entry);
    }

    //取得 body 底下所有 node 的 key
    for(var key of children.keys()) {
      console.log(key);
    }

    //取得 body 底下所有 node 的 value
    for(var value of children.values()) {
      console.log(value);
    }

    //取得每個 node 的值、索引與
    children.forEach(
      function(currentValue, currentIndex, childNodeList) {
        console.log(currentValue + ', ' + currentIndex + ', ' + this);
      },
      '[用在 this 的參數，可以是任何值]'
    );
  </script>

```

```
</script>
</body>
</html>
```

Module 6. 函式

6-1: 函式定義

函式（Function，又稱函數），是把程式碼集合在一起，以便能夠重複使用它們的一種方法。原則上，函式是有名字的（函式名稱）。

基本結構

```
function 函式名稱() {
    //程式執行區域
}
```

建立函式

```
function say() {
    console.log('Hello World!');
}
```

呼叫函式

接續建立函式的內容，我們使用在函式名稱後面加上「()」，便能執行函式內容。

```
say(); //輸出 Hello World!
```

參數傳遞

將參數設定在「函式的括號內」，同時在外部使用「賦予參數特定值」的函式，便能將「參數的值」傳遞給區塊當中的程式碼使用。

```
function say(title){
    console.log(`Hello ${title}!`);
}
```

```
say('Darren Yang'); //輸出 Hello Darren Yang!
```

補充說明

多個參數傳遞時的作法

```
function say (greeting, title) {  
    console.log('You said: ' + greeting + ' ' + title);  
}
```

```
say('Hello', 'World'); //輸出 You said: Hello World
```

回傳值

```
function say(greeting, title) {  
    return 'You said: ' + greeting + ' ' + title;  
}
```

```
console.log( say('Hello', 'World') ); // 輸出 You said: Hello World
```

範例 6-1-1.html

```
<html>  
<head>  
    <title>範例 6-1-1.html</title>  
    <script>  
        //建立基本函式  
        function say() {  
            console.log('Hello World!');  
        }  
  
        //帶有參數的函式  
        function greet(name){  
            console.log('Hello, ' + name);  
        }  
  
        //有回傳值的函式  
        function getMessage(){  
            return 'Good job!';  
        }  
    </script>  
</head>  
</html>
```

```

    //依序執行各個函式
    say();
    greet('Alex');
    console.log(getMessage());

    </script>
</head>
<body>

</body>
</html>

```

回呼函式

常稱為回呼函數、回調函式，為一種「延續傳遞風格」(Continuation-passing style) 的函式程式寫法，它的對比的是前面我們所提供的基本函式範例（直接風格，Direct style）。回呼函數可以將特定函式作為傳遞參數，在該函式中呼叫執行，將原本應該在該函式中回傳的值，交給下一個函式來執行。

範例

建立一個 say 函式，其中的第二個參數也是函式：

```

//主程式
say('Darren Yang', function(result){
    console.log(result);
});

//建立 say 函式
function say(name, callback_function){
    //say 函式會處理特定程式碼後，透過 callback_function 把結果或訊息回傳到主程式
    callback_function('Hi, [{name}] ... how have you been ?');
}

```

1. 主程式呼叫 say 函式的時候，除了第 1 個參數 name，在第 2 個參數放置一個 callback_function 函式，一起傳遞到 say 函式。
2. say 程式區塊中，使用主程式傳遞到 say 函式當中的 name 跟 callback_function 參數。

3. 經過處理，將結果作為 `callback_function` 函式的參數，再透過 `callback_function` 送回主程式，變成主程式的第二個參數。
4. 此時 `callback_function` 展開變成一般的函式「`function(result) { ... }`」，此時該函式的「`result`」就是在 `say` 函式中處理的結果，被 `callback_function` 作為參數，帶回主程式。

流程

1. 想像主程式原先的樣子：

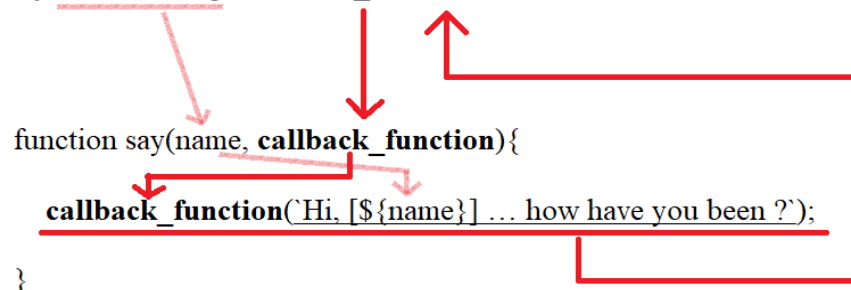
```
say('Darren Yang', callback_function);
```

2. 想像建立 `say` 函式的樣子：

```
function say(name, callback_function){
    callback_function('Hi, [{name}] ... how have you been ?');
}
```

3. 經過一連串的傳遞與處理：

```
say('Darren Yang', callback_function);
```



```
function say(name, callback_function){
    callback_function('Hi, [{name}] ... how have you been ?');
}
```

4. 主程式變成這個樣子，`result` 是「`Hi, xxx ... how have you been ?`」的文字處理結果，可以在「`{ ... }`」當中使用：

```
say('Darren Yang', function(result){ ... });
```

5. 我們可以自訂「`function(result) { ... }`」的內容：

```
say('Darren Yang', function(result){
    console.log('=====> {result}');
});
```



```
//輸出 =====> Hi, [Darren Yang] ... how have you been ?
```

使用回呼函數的目的，在於確保程式運作流程的明確性（另一種說法是指移交程式執行的控制權），例如一個回呼函式用於讀取檔案內容，主程式為了確保檔案內容被完整讀取出來，使用回呼函數，等待回呼函數執行完畢後，再透過主程式帶入的函式參數，將檔案內容回傳到主程式當中。

遞迴函式

舉個例子，知名漫畫/卡通《哆啦 A 夢》裡面的大雄，在房間裡面，用時光電視看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況。電視畫面中的那個時候，他正在用時光電視，看著未來的情況...

上述的例子說明了，遞迴是一種類似鏡中巢狀的圖像，電視中有電視，該電視中又有電視，如此重複著。以程式的角度來看，是指在函式當中，使用函式自身的方法。

範例

自己呼叫自己：

```
function go(){
    go();
}
go();
```

遞減數字相加（ $10 + 9 + 8 + \cdots + 2 + 1$ ）：

```
function add(num){
    if(num === 0) return 0;
    if(num === 1) return 1;
    return num + add(num-1);
}
console.log( add(10) );
// 輸出 55
```

計算 n 階乘：

```
function factorial(num){  
    if(num === 0) return 1;  
    return num * factorial(num-1);  
}  
console.log( factorial(5) );  
// 輸出 120
```

Fibonacci 數列：

```
function fibonacci(num){  
    if(num === 0) return 0;  
    if(num === 1) return 1;  
    return fibonacci(num-1) + fibonacci(num-2);  
}  
console.log( fibonacci(10) );  
//數列從第 0 項開始，所以 fibonacci(10) 是指第 11 項，輸出 55
```

使用遞迴時，需要有結束程式的時候，例如加上條件判斷，否則會不斷執行，造成資源浪費。

匿名函式

把一個函數複製給變數，而這個函數是沒有名字的，即匿名函數。

```
let say = function() {  
    console.log('Hello World!');  
}
```

我們也可以有參數

```
let say = function(name) {  
    console.log('Hello, ' + name);  
}
```

也可以有回傳值

```
let say = function(name) {  
    return 'Hello, ' + name;
```

}

範例 6-1-2.html

```
<html>
<head>
  <title>範例 6-1-2.html</title>
  <script>
    //建立匿名函式
    let say = function() {
      console.log('Hello World!');
    }

    //帶有參數的匿名函式
    let greet = function(name) {
      console.log('Hello, ' + name);
    }

    //有回傳值的匿名函式
    let getMessage = function(name) {
      return 'Good job!';
    }

    //依序執行各個函式
    say();
    greet('Alex');
    console.log(getMessage());
  </script>
</head>
<body>

</body>
</html>
```

補充說明

IIFE (Immediately Invoked Function Expression) 是一個定義完馬上就執行的 JavaScript function。

IIF 起手式

```
(  
    function(){  
        console.log('Hello World!');  
    }  
)();
```

帶參數的 IIFE

```
(  
    function (name) {  
        alert('Good job! ' + name);  
    }  
)('Darren');
```

他又稱為 Self-Executing Anonymous Function，也是一種常見的設計模式，包含兩個主要部分：第一個部分是使用 Grouping Operator () 包起來的 anonymous function。這樣的寫法可以避免裡面的變數污染到 global scope。

第二個部分是馬上執行 function 的 expression ()，JavaScript 引擎看到它就會立刻轉譯該 function。

範例 6-1-3.html

```
<html>  
<head>  
    <title>範例 6-1-3.html</title>  
    <script>  
        //顯示在 Console 面板  
        (  
            function () {  
                console.log('Hello World!');  
            }  
        )();  
  
        //顯示在 <body> 當中  
        (  
            function () {  
                document.write('Good job!');  
            }  
        )
```

```

    )();

    //跳出訊息
    (
        function (name) {
            alert('Good job! ' + name);
        }
    )('Darren');
</script>
</head>
<body>

</body>
</html>

```

箭頭函式

補充說明

箭頭函數：

我們從基本的函式結構

```

var 函式名稱 = function() {
    //程式執行區域
}

```

變成

```

var 函式名稱 = () => {
    //程式執行區域
}

```

不使用「function」關鍵字來定義函式，「){」之間，也由「=>」來代替。

```

let reflect = value => value;
           函式參數 回傳值

```

相當於

```
let reflect = function(value) {  
    return value;  
}  
console.log( reflect(20) );  
// 輸出 20
```

```
let sum = (num1, num2) => num1 + num2;
```

相當於

```
let sum = function(num1, num2){  
    return num1 + num2;  
}  
console.log( sum(20,50) );  
// 輸出 70
```

```
let getName = () => "Darren Yang";
```

相當於

```
let getName = function() {  
    return "Darren Yang";  
}  
console.log( getName() );  
// 輸出 Darren Yang
```

下面這一種比較常見：

```
let sum = (num1, num2) => {  
    return num1 + num2;  
}
```

相當於

```
let sum = function(num1, num2){  
    return num1 + num2;  
}  
console.log( sum(8,9) );  
// 輸出 17
```

範例 6-1-4.html

```
<html>  
<head>  
    <title>範例 6-1-4.html</title>  
    <script>  
        //匿名函式風格的箭頭函式  
        let say = () => {  
            console.log('Hello World!');  
        }  
  
        //帶參數的箭頭函式  
        let greet = (name) => {  
            return 'Hello, ' + name;  
        };  
  
        //帶兩個參數的箭頭函式  
        let sum = (num1, num2) => num1 + num2;  
  
        //帶參數，未加 () 的箭頭函式  
        let getValue = value => value + ', ' + value;  
  
        //依序執行各個函式  
        say();  
        console.log(greet('Bill'));  
        console.log(sum(10, 20));  
        console.log(getValue('Ha'));  
  
        //箭頭函式版本 IIFE  
        (() => {  
            let strName = "Beryl";  
            console.log(`Hello, ${strName}`);  
        })();
```

```
</script>
</head>
<body>

</body>
</html>
```

補充說明

let 與 const，都在程式區塊內有效（{...}，block-scoped），例如在某個函式或某個判斷式裡面宣告常數，若是在區塊外嚐試輸出，則會出現錯誤訊息（因為解決了抬升問題）。在實務案例當中，會頻繁用到 let 與 const 關鍵字。

我們使用函式來說明變數 hoisting 的情況：

```
function getValue(condition){
  if(condition) {
    var value = "Yellow";
    return value;
  } else {
    return null;
  }
}
```

```
console.log( getValue(true) );
// 輸出 "Yellow"
```

上面的範例裡，var 會因為編譯器的關係，被提升（Hoisting）放到 if(...) 的上一行來進行宣告，讓 else {...} 區塊內也可以使用 value 這個變數。

```
function getValue(){
  var value; //被編譯器放到這裡
  if(condition) {
    value = "Yellow";
    return value;
  } else {
    //原先的概念上不會使用到 value 這個變數，
    //卻因為 hoisting 的關係，讓 else {...} 區塊內，也能使用到 value 變數
    return null;
  }
}
```



```

    }
}

```

在這個情況下，程式設計人員無法精確地掌控變數的生命週期（宣告、使用、消滅等），於是近年的 Javascript 版本增加了 `let` 與 `const` 關鍵字，來強化對變數生命週期的控制。

大家可以試試，以下四個 IIFE（Immediately Invoked Function Expression，是一個定義完馬上就執行的 JavaScript function），各自會出現什麼訊息？

使用 <code>var</code> 來宣告 <code>value</code>	使用 <code>let</code> 來宣告 <code>value</code>
<pre> (function getValue(condition){ if(condition) { var value = "Yellow"; console.log(value); } else { console.log(value); } })(true); </pre> <p>//輸出 Yellow</p>	<pre> (function getValue(condition){ if(condition) { let value = "Yellow"; console.log(value); } else { console.log(value); } })(true); </pre> <p>//輸出 Yellow</p>
<pre> (function getValue(condition){ if(condition) { var value = "Yellow"; console.log(value); } else { console.log(value); } })(false); </pre> <p>//輸出 Undefined //因為 <code>else {…}</code> 也能存取 <code>value</code></p>	<pre> (function getValue(condition){ if(condition) { let value = "Yellow"; console.log(value); } else { console.log(value); } })(false); </pre> <p>//拋出錯誤訊息，因為 <code>value</code> 在 <code>else {…}</code> //裡面尚未被宣告，可以從這裡控制變數的生命週期</p>

範例：

```
var count = 30;
```

let count = 40; //這裡會拋出錯誤，因為重覆宣告

範例：

```
var count = 30;
```

```
if(condition) {
```

```
    let count = 40; // 這裡不會拋出錯誤
```

```
    // if 區塊內部的 count，會遮住外部的 count，
```

```
    // 外部的 count 只有 if 區塊「外面」才能存取得到
```

```
    //其它程式碼...
```

```
}
```

以上討論完 let 關鍵字，下面讓我們聊聊 const 關鍵字。

const 關鍵字是常數，其值一旦被設定後，**原則上不可任何更改**，每一個常數宣告時，都要初始化（就是一宣告就賦予初始值）：

範例：

```
const maxItems = 30;
```

```
const name; //拋出錯誤，常數尚未初始化
```

範例：

```
let condition = true;
```

```
if(condition) {
```

```
    const maxItems = 5;
```

```
}
```

```
console.log(maxItems); // 拋出錯誤，因為 if 區塊以外，無法存取 maxItems
```

範例：

```
var message = "Hello Word!";
```

```
let age = 25;
```

```
// 下面兩行都會拋出錯誤，因為重複宣告
```

```
const message = "Good Job!";
```

```
const age = 30;
```

雖然常數原則上重新賦值，但是當 const 宣告的變數，是「陣列」或是「物

件」，便會產生例外。

範例：

```
const arr = ["Hello", "World"];
arr.push("!");
console.log(arr);
// 輸出 [ 'Hello', 'World', '!' ]
```

範例：

```
const obj = {
  name: "Darren Yang",
};
obj.lineId = "telunyang";
obj.website = "https://darreninfo.cc";
console.log(obj);
// 輸出
// { name: 'Darren Yang', lineId: 'telunyang', website: 'https://darreninfo.cc' }
```

簡而言之，上面宣告為常數的陣列、物件變數，在 Javascript 的概念裡，之所以能夠增修，是因為陣列、物件變數擁有「感覺像是」 call by reference 的特性，在新增內部元素或屬性時，都在同一個記憶體位置作業，不會額外佔用記憶體位置，但實際上，Javascript 用的是 call by sharing，當參數物件修改的是「屬性」，則類似 call by reference；當參數物件修改的是本身的「值」，則類似 call by value。

call by value	call by reference → call by sharing
<pre>let a = 10; let b = 20; console.log(a, b); // 輸出 10, 20 function swap(x, y){ let tmp = x; x = y; y = tmp; }</pre>	<pre>let objA = {name: 'Darren'}; let objB; objB = objA; console.log(objA, objB); // 輸出 { name: 'Darren' } { name: 'Darren' } objA.name = 'Bill'; console.log(objA, objB); // 輸出 { name: 'Bill' } { name: 'Bill' }</pre>

<pre> swap(a, b); console.log(a, b); // 輸出 10, 20 </pre>	<pre> function setName(objX){ objX.name = 'Alex'; //類似傳址呼叫 // objX = {name: 'Doris'} //類似傳值呼叫 } setName(objA); console.log(objA, objB); // 輸出 { name: 'Alex' } { name: 'Alex' } </pre> <p>若是把 setName 當中的兩行程式註解互換， objA 跟 objB 還各是什麼？</p>
<pre> let a = 30; let b; b = a; a = 20; console.log(a); // 輸出 20 console.log(b); // 輸出 30 </pre>	<pre> let a = {greeting: "Hi"}; let b; b = a; b.greeting = "Hello World!"; console.log(a); // 輸出{ greeting: 'Hello World!' } console.log(b); // 輸出{ greeting: 'Hello World!' } b = {greeting: 'Hello!'}; console.log(a); // 輸出 { greeting: 'Hello World!' } console.log(b); // 輸出 { greeting: 'Hello!' } </pre>
call by value	call by reference
布林值、字串、數值、空值（null）、未定義（undefined）	物件、陣列、函式

6-2: 呼叫

即為函式的使用（通常是函式名稱後面加上一組小括號），可參考 6-1 的內容。

6-3: 綁定

在談綁定前，我們來聊聊 this 關鍵字。

何謂 this？通常是指向它所歸屬的「物件」。它在不同的使用情境下，會有不同的值：

- 在物件（object）的方法（method，它是 function，在物件導向的世界裡叫作 method）中，this 指向自己所在的物件（object）。
- 單獨使用，this 指向的是全域物件（global object），也就是 window。
- 在函式中（function）中，this 指向的是全域物件（global object），也就是 window。
- 在嚴格模式（strict mode）下的函式中（function）中，this 是 undefined。
- 在事件（event）裡，this 指向接收事件的元素（element）。
- 其它方法像是 call()、apply()，this 指向的是任何的物件（any object）。

範例 6-3-1.html

```
<html>
<head>
  <title>範例 6-3-1.html</title>
</head>
<body>
  <p id="demo"></p>

  <script>
    //建立一個物件，裡面有定義方法（method），這是先決條件；
    //在這裡的 this，是指 person = {...} block 裡面的成員
    var person = {
      firstName: "大 Guy",
```

```

        lastName : "4 John",
        id       : 5566,
        fullName : function() {
            return this.firstName + " " + this.lastName;
        }
    };

    //使用物件的方法，來取得回傳的資料
    document.getElementById("demo").innerHTML = person.fullName();
</script>
</body>
</html>

```

範例 6-3-2.html

```

<html>
<head>
    <title>範例 6-3-2.html</title>
</head>
<body>
    <p id="demo"></p>

    <script>
        //this 在這裡指的是 window
        var x = this;

        // x 執行 .toString() 後，回傳 [object Window]
        document.getElementById("demo").innerHTML = x;
    </script>
</body>
</html>

```

範例 6-3-3.html

```

<html>
<head>
    <title>範例 6-3-3.html</title>
</head>
<body>
    <p id="demo"></p>

```

```

<script>
document.getElementById("demo").innerHTML = myFunction();

//在函式中 (function) 中，this 指向的是全域物件 (global object) ，
//也就是 window，所以它會顯示 [object Window]
function myFunction() {
    return this;
}
</script>
</body>
</html>

```

範例 6-3-4.html

```

<html>
<head>
    <title>範例 6-3-4.html</title>
</head>
<body>
    <p id="demo"></p>

    <script>
        "use strict"; //嚴格模式
        document.getElementById("demo").innerHTML = myFunction();
        function myFunction() {
            return this;
        }
    </script>
</body>
</html>

```

範例 6-3-5.html

```

<html>
<head>
    <title>範例 6-3-5.html</title>
</head>
<body>
    <!-- onclick 事件裡面的 this，代表元素自己 -->

```

```
<button onclick="this.style.display='none'">按我隱藏</button>
</body>
</html>
```

在這裡的綁定，稱之為顯式函式綁定（Explicit Function Binding），用來預先定義 JavaScript 的方法。常用的綁定方式有三個：

方法	說明
Function.call(this, arg1, arg2, ..., argN)	this：輸入的物件會被指定為目標函式中的 this；其它 arg 都是類似 function(arg1, arg2, ..., argN) 的操作方式。
Function.apply(this, [arg1, arg2, ..., argN])	與 .call() 類似，差別在於 arg 的引用，是以「[arg1, arg2, ..., argN]」的陣列形式，作為引數。
Function.bind(this, arg1, arg2, ..., argN)	與 .call() 類似，差別在於回傳的是一個 function，我們可以在 .bind() 後面加上 () 來直接執行，或是賦予回傳結果到一個變數上，再透過「變數名稱」來執行。

以下是一般的物件方法（method）：

```
範例 6-3-6.html
<html>
<head>
  <title>範例 6-3-6.html</title>
</head>
<body>
  <p id="demo"></p>

  <script>
    var name = '小王';
    var age = 17;

    //以下使用一般的物件方法
    var obj = {
      name: '小強',
      objAge: this.age,
      myFunc: function(){
```



```

        //在 function 中，
        //this.age 是指 obj 本身 {...} 當中的 age，
        //但我們沒有定義 age，所以會顯示 undefined
        console.log(this.name, this.age);
    }
};

console.log(obj.objAge); //輸出 17
obj.myFunc(); //輸出 小強 undefined
</script>
</body>
</html>

```

使用 call()、apply()、bind()：

範例 6-3-7.html

```

<html>
<head>
    <title>範例 6-3-7.html</title>
</head>
<body>
    <p id="demo"></p>

    <script>
        var name = '小王';
        var age = 17;

        //以下使用一般的物件方法
        var obj = {
            name: '小強',
            objAge: this.age,
            myFunc: function(){
                console.log(this.name, this.age);
            }
        };

        //當作 this 參數的物件
        var objArg = {
            name: '小倫',

```

```

        age: 99
    };

    //call() 可以把 objArg 當作 obj.myFunc() 當中的變數，
    //讓 myFunc() 可以把 objArg 裡面的 key: value 組合，
    //當成自己的內部變數來使用
    obj.myFunc.call(objArg);

    //apply() 原則上跟 .call() 用法一樣，
    //差異在於後面有 arg1,...,argN 的時候，
    //要用 [] 把其它 arg1,...argN 包進去
    obj.myFunc.apply(objArg);

    //bind() 也跟 call() 用法差不多，
    //差異在於回傳的是新的 function，
    //我需要在 .bind() 後面，再加上 () 來執行
    obj.myFunc.bind(objArg)();

    //或是 .bind() 賦予回傳結果到一個變數上，再透過變數名稱後面加上 () 來執行
    let newFunc = obj.myFunc.bind(objArg);
    newFunc();
</script>
</body>
</html>

```

範例 6-3-8.html

```

<html>
<head>
    <title>範例 6-3-8.html</title>
</head>
<body>
    <p id="demo"></p>

    <script>
        var name = '小王';
        var age = 17;

        //以下使用一般的物件方法
    </script>

```

```

var obj = {
  name: '小強',
  objAge: this.age,
  myFunc: function(nickname, identity){
    console.log(`我是 ${this.name}, 年紀是 ${this.age};`);
    console.log(`暱稱是 ${nickname}, 身分是 ${identity}`);
  }
};

//當作 this 參數的物件
var objArg = {
  name: '小倫',
  age: 99
};

obj.myFunc.call(objArg, '擺渡人', '學生'); //引數逐個置入
obj.myFunc.apply(objArg, ['擺渡人', '學生']); //引數透過 [] 包裝後，再置入
obj.myFunc.bind(objArg, '擺渡人', '學生')(); //引數逐個置入

let newFunc = obj.myFunc.bind(objArg, '擺渡人', '學生'); //引數逐個置入
newFunc();
</script>
</body>
</html>

```

參考資料：

The JavaScript **this** Keyword

https://www.w3schools.com/js/js_this.asp

JavaScript 中 call()、apply()、bind() 的用法

<https://www.runoob.com/w3cnote/js-call-apply-bind.html>

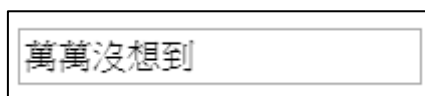
Module 7. 表單

7-1: 文字欄

在 HTML 的 input 元素中，type 屬性為 text 時，是一般文字輸入欄。

範例

```
<input type="text" name="txt" id="txt" value="萬萬沒想到" />
```



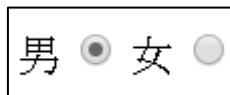
(圖) 文字欄位

7-2: 單選項目

在 HTML 的 input 元素中，type 屬性為 radio 時，是單選項目。

範例

```
<label>男</label>  
<input type="radio" name="radioGender" value="男" />  
  
<label>女</label>  
<input type="radio" name="radioGender" value="女" />
```



(圖) 多個元素，只能選一個

7-3: 多選項目

在 HTML 中，可使用 select 元素，加上 multiple 屬性，透過 Ctrl + 滑鼠左鍵，進行多個項目選擇。

範例

```
<select multiple>
  <option value="外帶">外帶</option>
  <option value="內用">內用</option>
  <option value="宅配">宅配</option>
</select>
```



(圖) 使用 select 元素，加上 multiple 屬性，可進行多選

範例 7-3-1.html

```
<html>
<head>
  <title>範例 7-3-1.html</title>
</head>
<body>
  <input type="text" name="txt" id="txt" value="萬萬沒想到" />

  <hr />

  <label>男</label>
  <input type="radio" name="radioGender" value="男" />

  <label>女</label>
  <input type="radio" name="radioGender" value="女" />

  <hr />

  <select multiple>
    <option value="外帶">外帶</option>
    <option value="內用">內用</option>
    <option value="宅配">宅配</option>
  </select>
</body>
```

```
</html>
```

Module 8. 深入表單

8-1: 檔案上傳

上傳檔案的表單，有幾個重要的注意事項：

- 在 form 標籤中，action 屬性必須設定為接收檔案的路由（routing）或後端程式（例如 .php 檔案）。
- 在 form 標籤中，method 屬性必須設定為 POST (大小寫皆可)。
- 在 form 標籤中，因為有檔案上傳，所以要設定資料的編碼方式，這時候要加上 `enctype="multipart/form-data"`，才能讓檔案正確地送出。
- 上傳檔案的 input 標籤，type 屬性必須設定為 `type="file"`，才能在使用時，出現瀏覽鈕，讓使用者選擇上傳的檔案。

範例 8-1-1.html

```
<html>
<head>
  <title>範例 8-1-1.html</title>
</head>
<body>
  <form name="myForm"
    action="指定後端網頁伺服器的路由或程式"
    method="POST"
    enctype="multipart/form-data">
    <h3>請選擇所要上傳的檔案</h3>
    <input type="file" name="fileUpload" /><br />
    <input type="submit" value="送出資料" />
  </form>
</body>
</html>
```

8-2: 上傳前預覽

藉由 `FileReader` 物件，Web 應用程式能以非同步（asynchronously）方式讀取儲存在用戶端的檔案（或原始資料暫存）內容，可以使用 `File` 或 `Blob` 物件指定要讀取的資料。

`File` 物件可以從使用者於 `<input>` 元素選擇之檔案所回傳的 `FileList` 物件當中取得，或是來自拖放操作所產生的 `DataTransfer` 物件之中，也能由 `HTMLCanvasElement` 物件（元素物件）執行 `mozGetAsFile()` 方法後回傳。

`FileReader` 有四個方法來讀取檔案資料：

- `readAsBinaryString(BlobFile)`
- `readAsDataURL(BlobFile)`
- `readAsText(BlobFile)`
- `readAsArrayBuffer(BlobFile)`

`FileReader` 在讀取檔案時，有幾個事件可用：

- `.onload()`：
 - `load` 事件處理器，於讀取完成時觸發。
- `.onerror()`：
 - `error` 事件處理器，於讀取發生錯誤時觸發。
- `.onprogress()`：
 - `progress` 事件處理器，於讀取 `Blob` 內容時觸發。
- `.onloadstart()`：
 - `loadstart` 事件處理器，於讀取開始時觸發。
- `.onloadend()`：
 - `loadend` 事件處理器，於每一次讀取結束之後觸發（不論成功或失敗），會於 `onload` 或 `onerror` 事件處理器之後才執行。
- `.onabort()`：
 - `abort` 事件處理器，於讀取被中斷時觸發。

範例 8-2-1.html

```
<html>
<head>
  <title>範例 8-2-1.html</title>
  <style>
    .thumb {
```

```

        height: 75px;
        border: 1px solid #000;
        margin: 10px 5px 0 0;
    }
</style>
</head>
<body>
    <form name="myForm"
        action=""
        method="POST"
        enctype="multipart/form-data">
        <h3>請選擇所要上傳的檔案</h3>

        <input type="file" name="fileUpload[]" id="files" multiple />
    </form>

    <hr />

    <ul id="list"></ul>

    <script>
    function handleFileSelect(evt) {
        var files = evt.target.files; // FileList 物件

        //個別走訪多重選擇的檔案
        for (var i = 0, f; f = files[i]; i++) {

            // 僅處理 image/jpeg、image/png 等格式的圖片檔案
            if (!f.type.match('image.*')) {
                continue;
            }

            //建立 FileReader 物件
            var reader = new FileReader();

            //透過閉包將 f 引數帶到 IIFE 的 function 作為參數
            reader.onload = (function(theFile) {
                return function(e) {

```



```

        //產生縮圖
        var li = document.createElement('li');

        //透過陣列結合的方式，將各種陣列元素中的字串合併
        li.innerHTML = [
            ''
        ].join('');

        //新增 li 元素
        document
            .getElementById('list')
            .insertBefore(li, null);
    };
})(f);

    // 以 data URL 的格式來讀取圖檔
    reader.readAsDataURL(f);
}
}

//監聽多選檔案之後的事件
document
    .getElementById('files')
    .addEventListener('change', handleFileSelect, false);
</script>
</body>
</html>

```

8-3: 自訂上傳按鈕

原則上，自訂上傳按鈕需要表單或 FormData 物件，也需要一個 input 標籤，type 屬性為 file，同時將標籤隱藏起來；當自訂按鈕觸發時，會呼叫 input 標籤的 click 方法。

範例 8-3-1.html

```
<html>
<head>
  <title>範例 8-2-1.html</title>
  <style>
    .thumb {
      height: 75px;
      border: 1px solid #000;
      margin: 10px 5px 0 0;
    }
    input#files{
      display: none;
    }
  </style>
</head>
<body>
  <form name="myForm"
    action=""
    method="POST"
    enctype="multipart/form-data">
    <h3>請選擇所要上傳的檔案</h3>

    <!-- input[type=file] 的標籤已經被隱藏起來了 -->
    <input type="file" name="fileUpload[]" id="files" multiple />

    <!-- 自訂的按鈕會有 onclick 屬性，註冊了點選 input[type=file] 標籤的功能 -->
    <input type="submit" name="smb" id="smb"
      onclick="document.getElementById('files').click(); return false;"
      value="選擇檔案" />

  </form>

  <hr />

  <ul id="list"></ul>

  <script>
    function handleFileSelect(evt) {
```

```

var files = evt.target.files; // FileList 物件

//個別走訪多重選擇的檔案
for (var i = 0, f; f = files[i]; i++) {

    // 僅處理 image/jpeg、image/png 等格式的圖片檔案
    if (!f.type.match('image.*')) {
        continue;
    }

    //建立 FileReader 物件
    var reader = new FileReader();

    //透過閉包將 f 引數帶到 IIFE 的 function 作為參數
    reader.onload = (function(theFile) {
        return function(e) {
            //產生縮圖
            var li = document.createElement('li');

            //透過陣列結合的方式，將各種陣列元素中的字串合併
            li.innerHTML = [
                '<img class="thumb" src=',
                e.target.result,
                '" title=',
                escape(theFile.name),
                '"/>'
            ].join('');

            //新增 span 元素到
            document
                .getElementById('list')
                .insertBefore(li, null);
        };
    })(f);

    // 以 data URL 的格式來讀取圖檔
    reader.readAsDataURL(f);
}

```

```
}

//監聽多選檔案之後的事件
document
.getElementById('files')
.addEventListener('change', handleFileSelect, false);
</script>
</body>
</html>
```

Module 9. 事件處理器

9-1: 不同形式的事件處理器

我們可以將事件當作屬性，放在元素（html element）當中。

以下是 JavaScript 常用事件表（事件會以 on 開頭）：

事件	說明
onclick	當使用者產生點擊某元素時，例如選擇某的選項或是按鈕（button）。
onchange	當元素發生改變時，例如選擇下拉選單（select option）中的其他項目時。
onblur	當游標失去焦點時，也就是點選其他區域時，通常用於填完表單的一個欄位。
ondblclick	連續兩次 click 某特定元素，通常用於需要特定確認的情況。
onfocus	當網頁元素被鎖定的時候，例如textarea、input text。
onload	當頁面載入完成後立即觸發 function。
onmousedown	滑鼠事件，當滑鼠的按鍵被按下的時候。
onmouseover	滑鼠事件，當滑鼠游標移經某個元素或區塊時。
onmousemove	滑鼠事件，當滑鼠游標移動時。

onmouseout	滑鼠事件，當滑鼠移出某個元素或區塊時。
onmouseup	滑鼠事件，當滑鼠的按鍵被放開的時候。
onunload	當使用者要準備離開網頁的時候。

我們也可以透過 `addEventListener(event, function)` 方法用於指定元素註冊（添加）事件。

參數	描述
event	事件類型的字串。
function	當你觸發事件時，所要執行的函式

事件	描述
focus	當網頁元素被鎖定的時候，例如textarea、input text。
change	下拉式選單被切換選項的時候
mouseover	滑鼠游標移入元素上方時
mouseout	滑鼠游標移出元素時
mousemove	滑鼠游標移入元素當中任何一個位置時
mousedown	滑鼠點擊元素時
mouseup	滑鼠點擊元素後，放開點擊按鍵時
click	點擊元素
dblclick	快速兩次點擊元
keydown	鍵盤按下按鍵的時候
keypress	鍵盤按下按鈕後，放開按鍵時
submit	表單送出
load	網頁讀取完畢
unload	離開網頁時

9-2: 滑鼠事件

on 事件	事件
onclick	click
ondblclick	dblclick
onmousedown	mousedown

onmousemove	mousemove
onmouseout	mouseout
onmouseover	mouseover
onmouseup	mouseup

9-3: 事件模型

Web 應用程式本身是事件驅動（Event-driven）的。它在適當的時間，透過用戶的操作或系統事件，來執行某些操作。在事件標準化之前存在基本事件模型（Basic Event Model），即使沒有標準化，它也具有更好的瀏覽器兼容性。因此，這是一個眾所周知且最受歡迎的事件模型。基本事件模型通常以「on」開頭，例如「onclick」、「onfocus」等。

範例 9-3-1.html

```
<html>
<head>
  <title>範例 9-3-1.html</title>
</head>
<body>
  <button id="btn1" onclick="handler(this)">Button 1</button><br>
  <button id="btn2" onclick="handler(this)">Button 2</button><br>
  <div id="console"></div>

  <script>
    //自訂事件處理器
    function handler(e) {
      document.getElementById('console').innerHTML
        = 'Who\'s clicked: ' + e.id;
    }
  </script>
</body>
</html>
```

上面的範例，將 onclick 屬性直接寫在 HTML 當中，違反了 HTML 與 JavaScript 程式碼分離的原則，將兩者寫在一起，不利於程式碼分工。

接下來我們來看看下面的範例：

範例 9-3-2.html

```
<html>
<head>
  <title>範例 9-3-2.html</title>
</head>
<body>
  <button id="btn1">Button 1</button><br>
  <button id="btn2">Button 2</button><br>
  <div id="console"></div>

  <script>
    //第一個 window.onload 事件
    window.onload = function() {
      //自訂事件處理器
      function handler() {
        document.getElementById('console').innerHTML
          = 'Who\'s clicked: ' + this.id;
      }

      //為 btn1 註冊事件 handler
      document.getElementById('btn1').onclick = handler;
    };

    //下面的 window.onload 會覆蓋上面的 window.onload
    window.onload = function() {
      //自訂事件處理器
      function handler() {
        document.getElementById('console').innerHTML
          = 'Who\'s clicked: ' + this.id;
      }

      //為 btn2 註冊事件 handler
      document.getElementById('btn2').onclick = handler;
    };
  </script>
</body>
</html>
```

```
</script>
</body>
</html>
```

上面的範例，第二個 window.onload 會蓋過第一個 window.onload，只有二個有效。

範例 9-3-3.html

```
<html>
<head>
  <title>範例 9-3-3.html</title>
</head>
<body>
  <button id="btn1">Button 1</button><br>
  <button id="btn2">Button 2</button><br>
  <div id="console"></div>

  <script>
    //自訂事件處理器
    function handler1() {
      document.getElementById('console').innerHTML
        = 'Who\'s clicked: ' + this.id;
    }
    function handler2() {
      document.getElementById('console').innerHTML
        = 'Who\'s clicked: ' + this.id;
    }

    //當網頁讀取完畢後
    window.onload = function() {
      //為 btn1 註冊事件 handler1
      document.getElementById('btn1').onclick = handler1;

      //為 btn2 註冊事件 handler2
      document.getElementById('btn2').onclick = handler2;
    };
  </script>
</body>
```



```
</html>
```

上面的範例，雖然可以在 `window.onload` 中使用不同事件處理器，但定義這麼多事件處理器，會造成管理上的困難（例如你有 100 的事件要處理，你就要寫 100 個 functions）。

DOM Level 2 Event Model 標準化了事件的註冊方式，使用「`addEventListener`」來針對註冊不同事件，來使用不同的事件處理器，我們也稱之為「標準事件模型」（Standard Event Model）：

範例 9-3-4.html

```
<html>
<head>
  <title>範例 9-3-4.html</title>
</head>
<body>
  <button id="btn1">Button 1</button><br>
  <button id="btn2">Button 2</button><br>
  <div id="console"></div>

  <script>
    //當網頁讀取完畢後
    window.addEventListener('load', function() {
      //自訂事件處理器
      function handler() {
        document.getElementById('console').innerHTML
          = 'Who\'s clicked: ' + this.id;
      }
      document.getElementById('btn1')
        .addEventListener('click', handler, false);
      document.getElementById('btn2')
        .addEventListener('click', handler, false);
    }, false);
  </script>
</body>
</html>
```

再來我們聊聊事件氣泡（Event Bubbling）、事件目標（Event Target）與事件捕捉

(Event Capturing)。

Event Bubbling：

範例 9-3-5.html

```
<html>
<head>
  <title>範例 9-3-5.html</title>
  <style>
    body * {
      margin: 10px;
      border: 1px solid blue;
    }
  </style>
</head>
<body>
  <form onclick="alert('form')">FORM
    <div onclick="alert('div')">DIV
      <p onclick="alert('p')">P</p>
    </div>
  </form>
</body>
</html>
```

按下 p 的時候，除了 p 的事件觸發外，還會遞迴地依序往父節點（parent node）觸發各自的 onclick 事件，這就叫事件氣泡（Event Bubbling）。

我們可以使用以下方式來停止 Event Bubbling：

範例 9-3-6.html

```
<html>
<head>
  <title>範例 9-3-6.html</title>
</head>
<body onclick="alert(`the bubbling doesn't reach here`)">
  <button onclick="event.stopPropagation()">點我一下</button>
</body>
</html>
```

不過，通常不建議停止 Event Bubbling，可能會影響某些巢狀元素（例如

select、以許多父子 div 組成的選單）的事件註冊與觸發。

Event target：

範例 9-3-7.html

```
<html>
<head>
  <title>範例 9-3-7.html</title>
  <style>
    form {
      background-color: green;
      position: relative;
      width: 150px;
      height: 150px;
      text-align: center;
      cursor: pointer;
    }

    div {
      background-color: blue;
      position: absolute;
      top: 25px;
      left: 25px;
      width: 100px;
      height: 100px;
    }

    p {
      background-color: red;
      position: absolute;
      top: 25px;
      left: 25px;
      width: 50px;
      height: 50px;
      line-height: 50px;
      margin: 0;
    }

    body {
```

```

        line-height: 25px;
        font-size: 16px;
    }
</style>
</head>
<body>
    點擊一下來對 event.target 和 this 來進行比較：

    <form id="form">FORM
        <div>DIV
            <p>P</p>
        </div>
    </form>

    <script>
        form.onclick = function(event) {
            event.target.style.backgroundColor = 'yellow';

            setTimeout(() => {
                alert("target = " + event.target.tagName + ", this=" + this.tagName);
                event.target.style.backgroundColor = ''
            }, 0);
        };
    </script>
</body>
</html>

```

`event.target` 是指觸發事件的目標元素，它不會受到 Event Bubbling 的影響；`this` 是註冊 `onclick` 事件的當前元素。

Event Capturing：

範例 9-3-8.html

```

<html>
<head>
    <title>範例 9-3-8.html</title>
</head>
<body>
    <div id="outA" style="width:400px; height:400px; background:#CDC9C9;position:relative;">

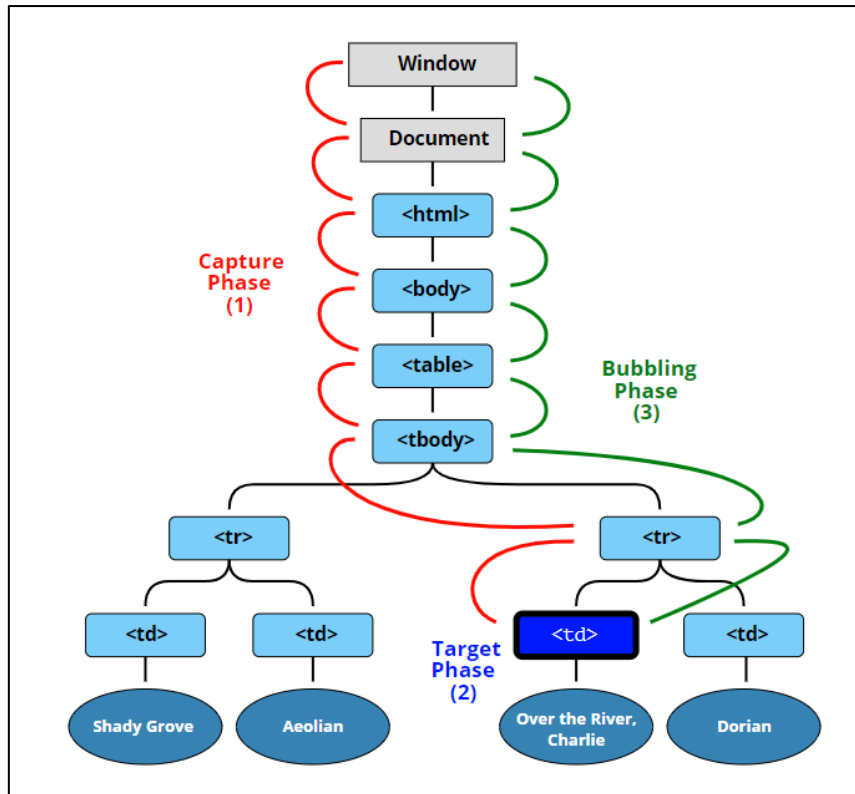
```

```
<div id="outB" style="height:200; background:#0000ff;top:100px;position:relative;">
  <div id="outC" style="height:100px; background:#FFB90F;top:50px;position:relative;"></div>
</div>
</div>

<script>
window.onload = function(){
  var outA = document.getElementById("outA");
  var outB = document.getElementById("outB");
  var outC = document.getElementById("outC");

  //addEventListener 第三個引數：
  // false 是支援 Event Bubbling
  // true 是指援 Event Capturing
  outA.addEventListener('click', function(){ alert(1); }, false);
  outB.addEventListener('click', function(){ alert(2); }, false);
  outC.addEventListener('click', function(){ alert(3); }, false);
};
</script>
</body>
</html>
```

若 `addEventListener` 第三個引數設定 `false`，則以 `Event Bubbling` 為主（預設），事件觸發後，依序跳出訊息為 3、2、1；若設定成 `true`，則以 `Event Capturing` 為主，事件觸發後，依序跳出訊息為 1、2、3。



(圖) 在 table 元素裡的 Bubbling、Target、Capturing 三個階段

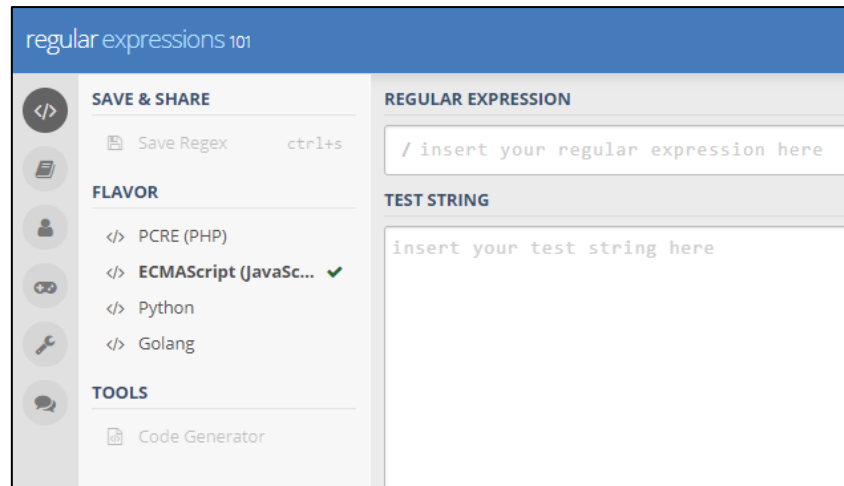
參考資料：

Bubbling and capturing

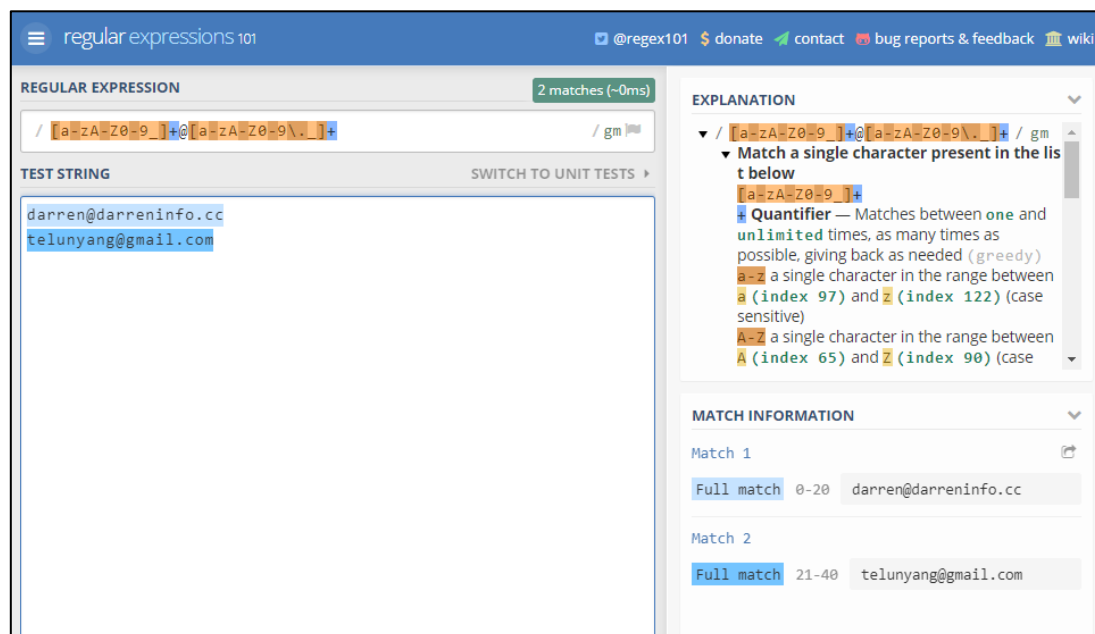
<https://javascript.info/bubbling-and-capturing>

Module 10. 正規表示法

又稱正規表達式 (Regular Expression)，是用來配對、過濾、替換文字的一種表示法。請先進入「<https://regex101.com/>」頁面，我們之後測試正規表達式，都會透過這個網頁的功能。大家，正規表達式是**需要大量練習才能了解的知識**，希望大家都能透過頻繁地練習，慢慢感受到正規表達式在文字處理上的便捷。



(圖) 網頁的樣式，請記得選擇 FLAVOR 為 ECMAScript (JavaScript)



(圖) 使用正規表達式，來判斷字串是否符合文字格式或條件

下面表格為快速參考的範例：

說明	正規表達式	範例
一個字元: a, b or c	[abc]	abcdef
一個字元，除了: a, b or c	[^abc]	abcdef
一個字元，在某個範圍內: a-z	[a-z]	abcd0123
一個字元，不在某個範圍內: a-z	[^a-z]	abcd0123
一個字元，在某個範圍內: a-z or A-Z	[a-zA-Z]	abcdXYZ0123
避開特殊字元	\	ex. ?

說明	正規表達式	範例
	\?	
任何單一字元	.	任何字元
任何空白字元 (\f \r \n \t \v)	\s	空格、換行、換頁等
任何非空白字元 (不是 \f \r \n \t \v)	\S	非空格、非換行、非換頁等
任何數字	\d	10ab
任何非數字	\D	10ab
任何文字字元	\w	10ab/*AZ*\$
任何非文字字元	\W	10ab/*AZ*\$
以群組的方式配對，同時捕捉被配對的資料	(...) ex. (1[0-9]{3} 20[0-9]{2})	1992, 2019, 1789, 1776, 1024, 3000, 4096, 8192
配對 a 或 b	alb	addbeeeaaccbaa
0 個或 1 個 a	a?	addbeeeaaccbaa
0 個或更多的 a	a*	addbeeeaaccbaa
1 個或更多的 a	a+	aaa, aaaaa
完整 3 個 a	a{3}	aaa, aaaaa
3 個以上的 a	a{3,}	aa, aaa, aaaaa
3 個到 6 個之間的 a	a{3,6}	aaa, aaaaaa, aaaa, aaaaaaaa
字串的開始	^ ex. ^Darren	^DarrenYang
字串的結束	\$ ex. Yang\$	DarrenYang\$
位於邊界的字元	\b ex. \bD	DarrenYang
非位於邊界的字元	\B ex. \Ba	DarrenYang
配對卻不在群組裡顯示	John(?:Cena)	John Cena
正向環視 (這位置右邊要出現什麼)	John(?=Cena)	John Cena
正向環視否定 (這位置右邊不能出現什麼)	Johnnie(?!Cena)	Johnnie Walker
反向環視 (這位置左邊要出現什麼)	(?<=Johnnie) Walker	Johnnie Walker
反向環視否定 (這位置左邊不能出現什麼)	(?<!=John) Walker	Johnnie Walker

範例說明：

用途	正規表達式	範例
E-mail	[a-zA-Z0-9_]+@[a-zA-Z0-9\._]+	darren@darreninfo.cc telunyang@gmail.com
英文名字	[a-zA-Z]+	Darren Alex
網址	https://[a-zA-Z0-9\./?_]+=	https://darreninfo.cc/?page_id=10
實數與小數	[0-9]{1,4}\.[0-9]+	9487.94

10-1: 單一字元表示法

範例 10-1-1.html
<pre> <html> <head> <title>範例 10-1-1.html</title> </head> <body> <script> //任何單一字元(任何字元) let str = 'AB123'; let pattern = /. /g; let match = null; while((match = pattern.exec(str)) !== null){ console.log(match); } //任何空白字元 (\f \r \n \t \v) 空格、換行、換頁等 str = ' AB123 '; pattern = /\s/g; match = null; while((match = pattern.exec(str)) !== null){ console.log(match); } //任何數字 str = 'AB123'; pattern = /\d/g; match = null; </pre>

```

while( (match = pattern.exec(str)) !== null ){
    console.log(match);
}

//任何文字
str = 'AB123';
pattern = /\w/g;
match = null;
while( (match = pattern.exec(str)) !== null ){
    console.log(match);
}
</script>
</body>
</html>

```

10-2: 多字元表示法

範例 10-2-1.html

```

<html>
<head>
    <title>範例 10-2-1.html</title>
</head>
<body>
    <script>
        //0 個或 1 個 a
        let str = 'addbaccbaa';
        let pattern = /ba?/g;
        let match = null;
        while( (match = pattern.exec(str)) !== null ){
            console.log(match);
        }

        //0 個或更多的 a
        str = 'addbaaaaaccbaa';
        pattern = /ba*/g;
        match = null;
        while( (match = pattern.exec(str)) !== null ){

```

```

        console.log(match);
    }

    //1 個或更多的 a
    str = 'addbaccbaaaaaa';
    pattern = /ba+/g;
    match = null;
    while( (match = pattern.exec(str)) !== null ){
        console.log(match);
    }

    //完整 3 個 a
    str = 'addbaaaaaaccbaa';
    pattern = /a{3}/g;
    match = null;
    while( (match = pattern.exec(str)) !== null ){
        console.log(match);
    }

    //3 個以上的 a
    str = 'a{3,}';
    pattern = /a{3}/g;
    match = null;
    while( (match = pattern.exec(str)) !== null ){
        console.log(match);
    }
    </script>
</body>
</html>

```

10-3: 表單送出前的檢查

範例 10-3-1.html

```

<html>
<head>
    <title>範例 10-3-1.html</title>
</head>

```

```

<body>
  <form id="myForm" method="post" action="">
    <label>您的身分證字號：</label>
    <input type="text" id="idNum" value="" />
    <input type="submit" id="btn" value="檢查" />
  </form>

  <script>
document.getElementById('btn').addEventListener('click', function(event){
  //preventDefault() 是讓元素本身的預設功能失去作用，單純作為觸發用的元素
  event.preventDefault();

  let elm = document.getElementById('idNum');
  let pattern = /[A-Z]\d[0-9]{8}/g;
  let match = null;
  if( (match = pattern.exec( elm.value )) !== null ){
    alert('身分證格式正確!');
  } else {
    alert('你的身分證格式有誤...');
  }
});
  </script>
</body>
</html>

```

接下來，我們即將進入部分後端（back-end）的環境。

Module 11. 網頁測試伺服器

11-1: 安裝 node 和 http-server

Node Version Manager（NVM）一款管理 Node.js 版本的工具，可以安裝不同版本，依需求切換。安裝時附帶安裝 Node Package Manager（NPM），用來管理 Node.js 所使用的套件等。安裝流程，以 Windows 版本為例。

Windows 版：<https://github.com/coreybutler/nvm-windows>

Linux / Mac 版：<https://github.com/nvm-sh/nvm>

Node Version Manager (nvm) for Windows

[gitter](#) [join chat](#) (I post development updates here)

[issues](#) [111 open](#) [stars](#) [9k](#) [code helpers](#) [0](#)

Manage multiple installations of node.js on a Windows computer.

`tl;dr nvm`, but for Windows, with an installer. [Download Now!](#) This has always been a node version manager, not an io.js manager, so there is no back-support for io.js. However, node 4+ is supported.

(圖) 按下 Download Now

1.1.7 - Maintenance Release

 coreybutler released this on 2 Aug 2018 · [28 commits](#) to master since this release

Merged several outstanding PR's:






- Silent Setup ([#264](#))
- Comparison Fix ([#269](#))
- Fix for Manually removing folder on nvm uninstall ([#356](#))
- Wilcard major version to latest ([#222](#))

Several documentation PR's were included in the repo as well.

The build process has been updated to provide:

- Code Signed Executables (courtesy Ecor Ventures/Author.io)
- MD5 Checksums

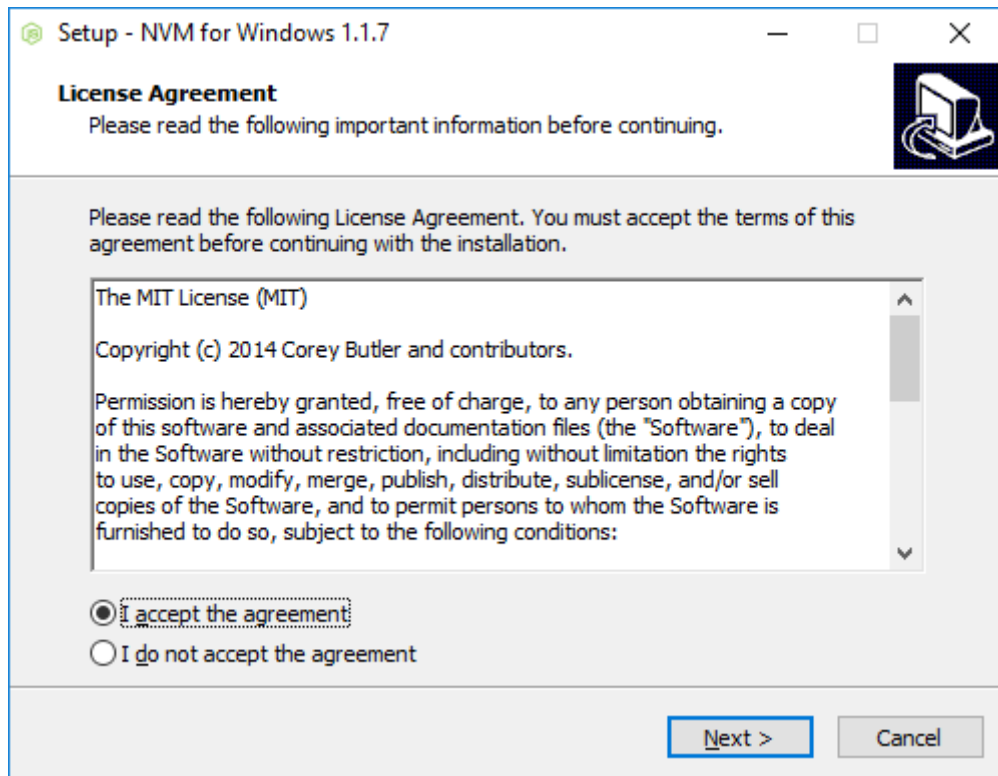
▼ Assets [6](#)

 nvm-noinstall.zip	2.3 MB
 nvm-noinstall.zip.checksum.txt	34 Bytes
 nvm-setup.zip	1.98 MB
 nvm-setup.zip.checksum.txt	34 Bytes
 Source code (zip)	
 Source code (tar.gz)	

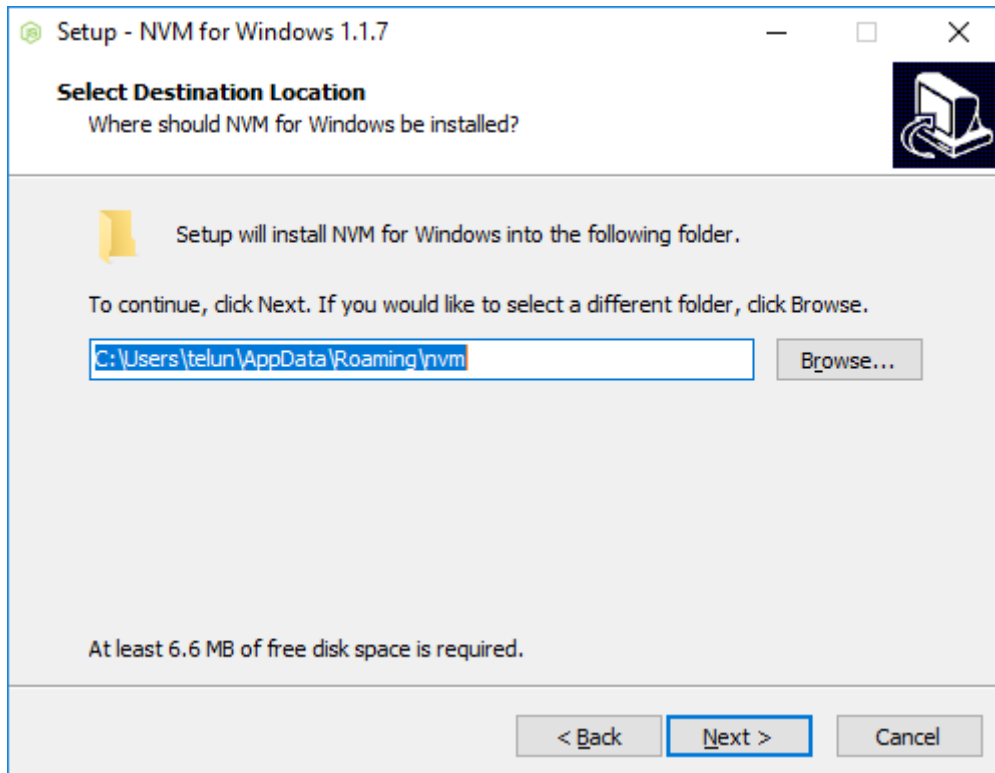
(圖) 本課程使用 1.1.7 版，下載 nvm-setup.zip



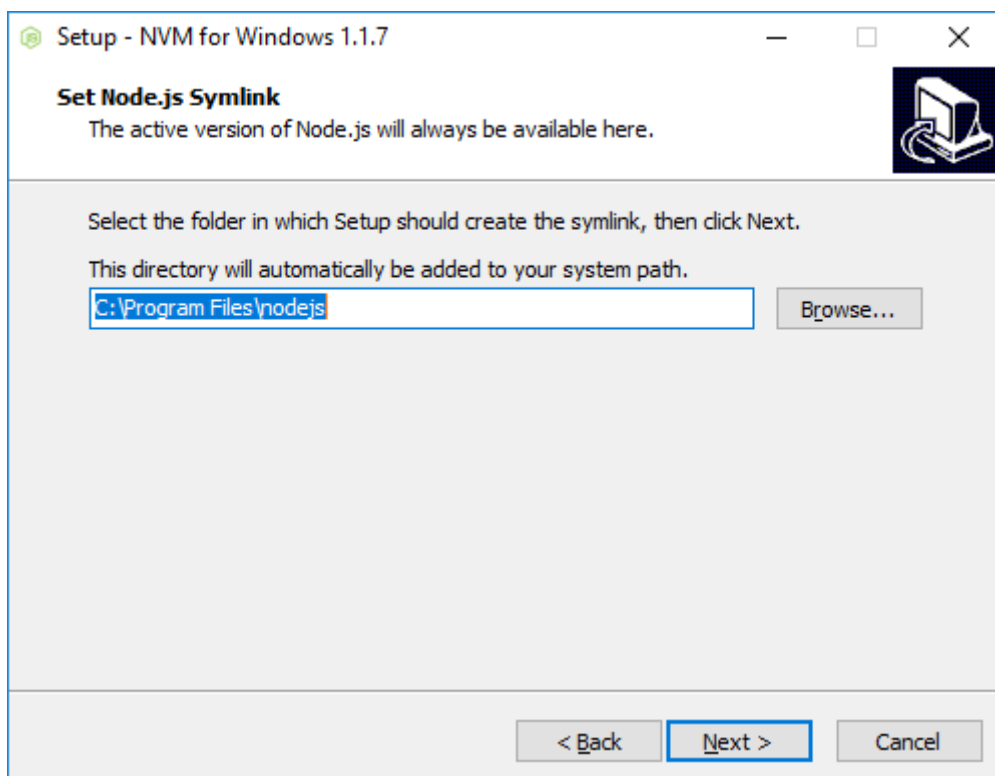
(圖) 壓縮檔解開後的執行檔



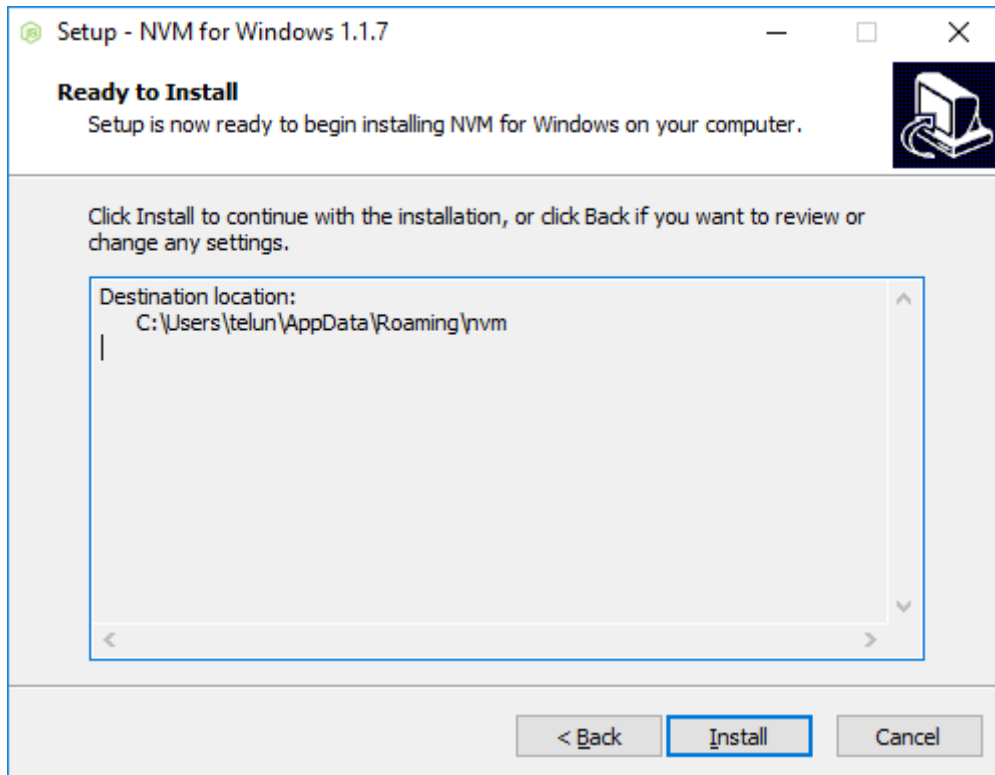
(圖) 同意協議後，按下一步



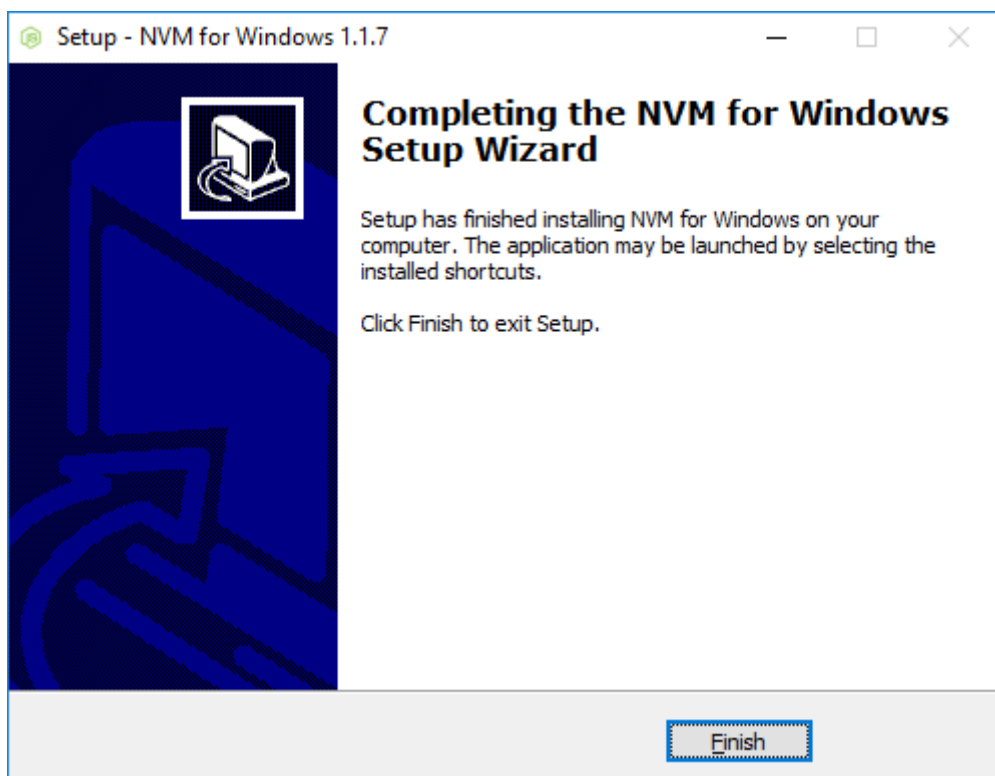
(圖) 沒有特別設定的話，按下一步



(圖) 建立類似軟連結的路徑，沒有特別設定，按下一步



(圖) 確認設定後，開始安裝



(圖) 安裝結束後，按下完成

接下來，我們開啟 Visual Studio Code（若已開啟，請先關閉後再重開 Visual Studio Code），按下「Ctrl + `」，開啟終端機（Terminal），使用指令來進行安裝。

node.js 安裝流程

1. 安裝 nvm（Windows 版或其它作業系統版本皆可；請勿透過系統管理員權限安裝）
2. 打開 Visual Studio Code（若已開啟，請先關閉後再打開，讓其存取 nvm 環境變數）
3. 安裝 node.js 時，會連 npm 一同安裝。

下列流程，每一個輸入完成後，請按下 Enter 執行：

1. 輸入 nvm（確認是否有 Usage 相關說明）
2. 輸入 nvm install 12.16.1
3. 輸入 nvm use 12.16.1
4. 輸入 node -v（確認 node.js 版本，理應輸出 v12.16.1）
5. 輸入 npm -v（確認套件管理工具版本，理應輸出 6.13.4）

補充說明

若是專案壓縮檔有更新的情況，請解壓縮後，透過 Visual Studio Code 開啟專案資料夾，再使用 Ctrl + ` 來開啟終端機，輸入「npm i --save」來安裝套件。

在這裡我們不使用 http-server 這個套件，我們直接使用 node.js express。

指令說明

#express 是 web 應用程式，express-generator 是 express 的指令集；nodemon 是開發環境常用的 node.js 執行程式，會隨時回報 web server 的狀況：

```
> npm i -g express express-generator nodemon
```

安裝結束後，會提供已加入（added）幾個套件（packages）的訊息。我們透過 express 指令，來檢視是否安裝成功，若是出現 Usage 等文字內容，代表安裝成功：

```
> express -h
```

接下來，我們需要建立一個網頁樣版引擎，選擇 ejs：

```
> express --view=ejs
```

此時會看到一堆 create 的訊息出現，例如「`create: web\public\`」等字樣，代表樣版引擎安裝成功。

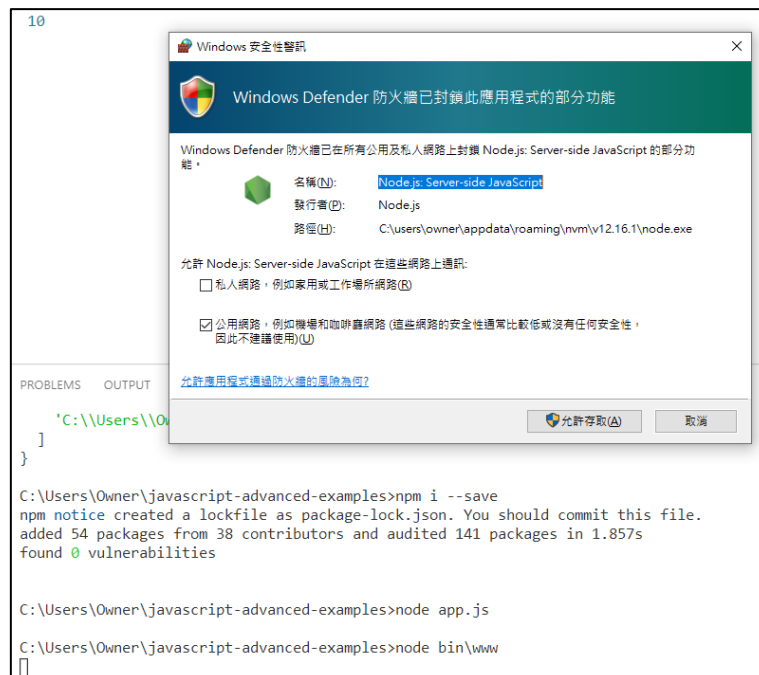
最後再透過 npm 套件管理工具，進行套件安裝，套件列表會放在 `package.json` 的檔案裡面：

```
> npm i --save
```

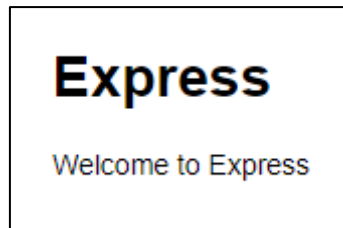
```
> bin
> node_modules
> public
> routes
> views
js app.js
{} package-lock.json
{} package.json
```

(圖) 安裝完成後，基本上會看到新增了圖片中的資料夾和檔案

11-2: 測試頁面



(圖) 到 Terminal 裡面，輸入 `node bin\www`，按下 `enter`，按下允許存取。



(圖) 輸入網址 <http://localhost:3000/>，會看到 Express 預設的頁面。

respond with a resource

(圖) 輸入 <http://localhost:3000/users> 會看到從伺服器回應的文字

在 Terminal 底下，按下 Ctrl + c，可以結束 node.js express 的服務。

11-3: 測試送出 querystring

編輯 routes/user.js：

說明

將原先的內容

```
router.get('/', function(req, res, next) {  
  res.send('respond with a resource');  
});
```

改成

```
router.get('/', function(req, res, next) {  
  let name = req.query.name;  
  let age = req.query.age;  
  res.send(`您傳送的資料為: name = ${name}, age = ${age}`);  
});
```

您傳送的資料為: name = Alex, age = 99

(圖) 輸入 <http://localhost:3000/users?name=Alex&age=99> 的結果

補充說明

req = request，代表 client 端送過來的請求

res = response，代表 server 端要回傳給 client 的回應

req.query.* 是指 網址後面的 ?key01=value01&key02=value02&...
即是剛才輸入在網址後面的 ?name=Alex&age=99

還有 req.params.* 和 req.body.*
將會由 node.js 老師為大家說明。

Module 12. Promise 類型

在討論 Promise 之前，我們先來聊聊同步與非同步。

同步 (synchronous) 的處理方式：

想像我們在銀行、郵局櫃台申辦業務，在你申辦的項目尚未完成前，你會繼續站在櫃台，承辦人員持續幫你處理業務，直到項目完成，你便會離開，櫃台會叫號，換下一位。

非同步 (asynchronous) 的處理方式：

想像我們在正在寫考卷，倘若你先寫完，你可以先繳卷，然後先離開，至於閱卷老師何時改完，我們並不知道，只知道老師改完，會立刻讓你知道成績。

12-1: unblock 與 callback hell

阻塞式 (Blocking) 的程式運作，需要等待程式 I/O 執行完畢，才會執行另一個程式 I/O；非阻塞式 (Non-blocking 或 unblock) 的程式運作，會透過輪詢 (polling) 方式，不斷確認 I/O 過程是否滿足結束條件，滿足以後，才會結束程式。JavaScript 在執行 IO 之類的動作時，不會等待 IO 結束後才往下執行。

範例 12-1-1.html

```
<html>
<head>
  <title>範例 12-1-1.html</title>
</head>
<body>

  <script>
```

```

function run01(){
    setTimeout(function(){
        console.log("Step: setTimeout() in run01()");
    }, 5000);

    console.log("Step: console.log() in run01()");
}

function run02(){
    setTimeout(function(){
        console.log("Step: setTimeout() in run02()");
    }, 3000);

    console.log("Step: console.log() in run02()");
}

run01();
run02();
</script>
</body>
</html>

```

通常在 Network Programming（尤其是 Linux socket 進行文字串接，是透過 Blocking I/O 來運作）的議題裡，討論較多；在 Node.js 裡，為非阻塞式的輸入輸出，通常會在串流（streams）的概念中加以討論。

在以前，為了確保程式進行的流程，我們會使用回呼函式。

範例 callback01.js

```

const fs = require('fs');

//讀取 source.txt 的檔案內容
fs.readFile("source.txt", function(err, contents){
    if(err){
        throw err;
    }
    console.log("讀取 source.txt 完成!");

    //若是讀取沒問題，則寫入 destination.txt 檔案

```

```
fs.writeFile("destination.txt", contents + ' Hello III!', function(error){
    if(error){
        throw error;
    }
    console.log("寫入 destination.txt 成功!");
});
});
```

若是讀寫的次數很多呢？

範例 callback02.js

```
const fs = require('fs');

//讀取 tmp.log 的檔案內容
fs.readFile("tmp.txt", function(err, contents){
    if(err){
        throw err;
    }

    console.log(`讀取 tmp.txt 成功`);

    //若是讀取沒問題，則加上新增的文字，然後寫入 cb1.txt 檔案
    fs.writeFile("cb1.txt", contents + ' cb1', function(err){
        if(err){
            throw err;
        }

        console.log(`寫入 cb1.txt 成功`);

        //讀取 cb1.txt 的檔案內容
        fs.readFile("cb1.txt", function(err, contents){
            if(err) throw err;

            console.log(`讀取 cb1.txt 成功`);

            //若是讀取沒問題，則加上新增的文字，然後寫入 cb2.txt 檔案
            fs.writeFile("cb2.txt", contents + ' cb2', function(err){
                if(err) throw err;
```



```

1  function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SRIPTS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });
23     });
24   });
25 };
26 }

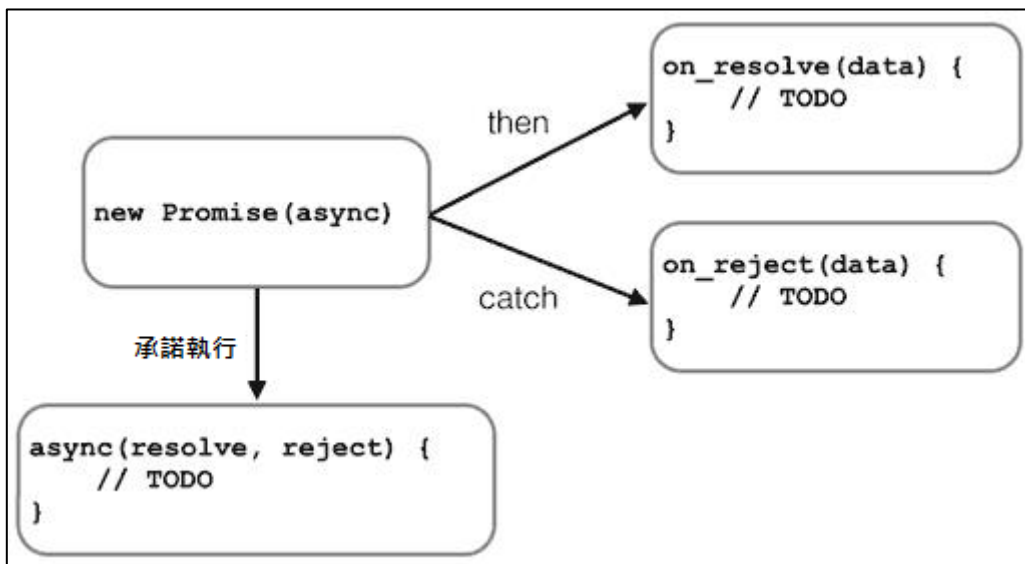
```



(圖) 不斷 callback，形狀會像什麼？

12-2: Promise 的優點

Promise 最大的好處，在於非同步（有人稱為異步）執行的流程中，把執行代碼和處理結果的代碼清晰地分離了。



(圖) Promise 簡化了 callback 的問題

12-3: Promise 建議的使用方式

在 ES6 (ECMAScript 2015) 當中，支援 Promise (還有先前討論到的 `let`、`const`、`arrow function` 等)，將原先回呼函式的格式，以鏈結的方式，來確保程式執行的流程，以及生命週期。

範例 promise01.js

```
const fs = require('fs');

let p = new Promise(function(resolve, reject){
  fs.readFile("tmp.txt", function(err, contents){
    if(err) {
      //讀取錯誤就回傳錯誤的物件到 .catch() 中
      reject(err);
      return;
    }

    //成功就往 .then() 傳遞
    resolve(contents);
  });
});

p.then(function(contents){
  fs.writeFile("promise_tmp.txt", `${contents} I'm ironman!!`, function(err){
    if(err){
      throw err;
    }
    console.log("寫入完成");
  });
}).catch(function(err){
  console.error(err);
});
```

另一種 Promise 使用方式 ↓

範例 promise02.js

```
const fs = require('fs');
```

```

new Promise(function(resolve, reject){
  fs.readFile("tmp.txt", function(err, contents){
    if(err) {
      //讀取錯誤就回傳錯誤的物件到 .catch() 中
      reject(err);
      return;
    };

    fs.writeFile("tmp1.txt", `${contents} I'm ironman!!`, function(err){
      if(err){
        throw err;
      }
      resolve('tmp1'); //成功就往 .then() 傳遞
    });
  });
}).then((tmpNum) => {
  console.log(`Write to ${tmpNum} ok`);
  let p2 = new Promise(function(resolve, reject){
    fs.readFile("tmp1.txt", function(err, contents){
      if(err) {
        //讀取錯誤就回傳錯誤的物件到 .catch() 中
        reject(err);
        return;
      };

      //成功就往 .then() 傳遞
      fs.writeFile("tmp2.txt", `${contents} I'm spiderman!!`, function(err){
        if(err){
          throw err;
        }
        resolve('tmp2'); //成功就往 .then() 傳遞
      });
    });
  });
  return p2;
}).then((tmpNum) => {
  console.log(`Write to ${tmpNum} ok`);
  let p3 = new Promise(function(resolve, reject){

```

```

    fs.readFile("tmp2.txt", function(err, contents){
        if(err) {
            //讀取錯誤就回傳錯誤的物件到 .catch() 中
            reject(err);
            return;
        };

        //成功就往 .then() 傳遞
        fs.writeFile("tmp3.txt", `${contents} I'm superman!!`, function(err){
            if(err){
                throw err;
            }
            resolve('tmp3'); //成功就往 .then() 傳遞
        });
    });
});
return p3;
}).then((tmpNum) => {
    console.log(`Write to ${tmpNum} ok`);
    let p4 = new Promise(function(resolve, reject){
        fs.readFile("tmp3.txt", function(err, contents){
            if(err) {
                //讀取錯誤就回傳錯誤的物件到 .catch() 中
                reject(err);
                return;
            };

            //成功就往 .then() 傳遞
            fs.writeFile("tmp4.txt", `${contents} I'm batman!!`, function(err){
                if(err){
                    throw err;
                }
                resolve('tmp4'); //成功就往 .then() 傳遞
            });
        });
    });
});
return p4;
}).then((tmpNum) => {

```

```
    console.log(`Write to ${tmpNum} ok`);
  })
  .catch((err) => {
    console.error(err); //例外處理
  });
```

Module 13. AJAX

13-1: 什麼是 AJAX

AJAX 為 Asynchronous(非同步) JavaScript and XML 的簡稱。AJAX 是在 瀏覽器不需要重整的情境下（不換頁的情況下），直接向伺服器 Server 端取得資料的一種傳輸技術，以達到提高網頁的互動性、速度效率，減少了伺服器的負荷量。

13-2: XMLHttpRequest

剛剛有提到 AJAX 為 Asynchronous(非同步) JavaScript and XML 的簡稱，我們在程式面上來說：使用 JavaScript 與伺服器 Server 端取得 XML（實務上 json 居多）的資料。為了使用傳統的 AJAX，JavaScript 內建提供了一個物件為 XMLHttpRequest，一個專門與伺服器 Server 端溝通的物件，所以我們在建立 AJAX 的連線之前，第一個動作就是建立 XMLHttpRequest 的物件，如下：

範例

var req = new XMLHttpRequest();

接著我們使用 XMLHttpRequest 提供的靜態方法 open() 與伺服器建立連線資訊：

範例

var req = new XMLHttpRequest(); req.open('get','https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch'); //建立連線資訊
--

在這邊 open() 只是設置了連線的資訊，還沒開始進行連線。

最後還需要使用 XMLHttpRequest 的提供的靜態方法 send() 去執行連線：

範例

<pre>var req = new XMLHttpRequest(); req.open('get','https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch'); req.send(); //執行連線</pre>
--

使用 XMLHttpRequest() 物件提供的事件方法（Method）取得伺服器內容。我們在連線中，也就是 onload 的事件中，就可以獲取伺服器 Server 端所請求的資料了：

範例

<pre>var req = new XMLHttpRequest(); req.open('get','https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch '); req.onloadstart = function(){ console.log('連線開始') } req.onload = function(){ console.log('連線中') console.log(this.responseText) //取得回應的內容的屬性 } req.onloadend = function(){ console.log('連線結束') } req.send(); //執行連線</pre>

範例 13-2-1.html

<pre><html> <head> <title>範例 13-2-1.html</title> </head> <body> <pre id="codeArea"></pre> <script> //建立 XMLHttpRequest 物件，來進行 ajax 非同步傳輸 var req = new XMLHttpRequest();</pre>

```

req.open('get', 'https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch');

//連線開始時觸發的事件
req.onloadstart = function(){
    console.log('連線開始')
}

//連線中所觸發的事件
req.onload = function(){
    console.log('連線中')
    document.getElementById("codeArea").innerHTML = JSON.stringify(JSON.parse(this.responseText), null, 4);
}

//連線結束後所觸發的事件
req.onloadend = function(){
    console.log('連線結束')
}

req.send(); //執行連線
</script>
</body>
</html>

```

13-3: fetch 函式

Fetch API 提供了工具使操作 http pipeline 更加容易，像是日常會用到的發送和接送資料都可以使用。並且有 global 的 fetch() 可以直接呼叫，使開發能夠用更簡潔的語法取得非同步資料（可以想成是較新的 AJAX）。

以往都是依賴 XMLHttpRequest。但相較下 Fetch 使用上更容易，並被廣泛使用在 Service Workers。Fetch 在設定 HTTP 相關的設定時，也提供可讀性比較好的方法，這些設定包括 CORS 以及其他 header。

範例 13-3-1.html

```

<html>
<head>
  <title>範例 13-3-1.html</title>

```

```

</head>
<body>
  <pre id="codeArea"></pre>

  <script>
    fetch('https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch')
    .then(function(response) {
      return response.json();
    })
    .then(function(myJson) {
      console.log(myJson);
    });
  </script>
</body>
</html>

```

範例 13-3-2.html

```

<html>
<head>
  <title>範例 13-3-2.html</title>
</head>
<body>
  <pre id="codeArea"></pre>

  <script>
    fetch('https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch', {
      //RESTful 方法，常見的有 GET, POST, PUT, DELETE
      method: 'GET',
      //設定標頭：指明使用者代理為桌面瀏覽器，同時要求後端伺服器回傳的格式是 json
      headers: {
        'user-agent': 'Mozilla/4.0 MDN Example',
        'content-type': 'application/json'
      },
      //傳遞資料的方法若為 POST，需要先設定成物件({...})，加上 body，
      //最後轉成透過 JSON.stringify() 將物件字串化，才能正確執行
      //body: JSON.stringify({})
    })
    .then(function(res) {

```

```

    /**
     * 使用 fetch，會以 ES6 的 Promise 來回應 (res, 即是 response)，
     * 回應的值為 ReadableStream 的實體，我們需要使用 json 的方法，
     * 去取得 json 格式的資料，然而依照 Fetch API 的格式，需要再次
     * return 到下一個 .then() 去接收，此時 .then() 裡面的回呼值，
     * 就會變成帶有實際 json 內容物件，而非 ReadableStream 物件。
     */
    return res.json();
  })
  .then((json) => {
    document.getElementById("codeArea").innerHTML = JSON.stringify(json, null, 4);
  })
  .catch(function(err){
    alert(err);
  });
</script>
</body>
</html>

```

參考資料：

鐵人賽：ES6 原生 Fetch 遠端資料方法

<https://wcc723.github.io/javascript/2017/12/28/javascript-fetch/>

Module 14. 物件導向使用 prototype

14-1: 自訂功能物件

以 function 這個關鍵字的來建立類別 (class)：

範例 14-1-1.html

```

<html>
<head>
  <title>範例 14-1-1.html</title>
</head>
<body>

```



```
<script>
//以 function 這個關鍵字的來建立類別 (class)
var Person = function () {};

//用 new 關鍵字來實體化 (變成物件)
var person1 = new Person();
var person2 = new Person();
</script>
</body>
</html>
```

14-2: 使用 function 自訂類型

以 this 關鍵字作為實體本身，同時存取成員屬性或方法：

範例 14-2-1.html

```
<html>
<head>
  <title>範例 14-2-1.html</title>
</head>
<body>

  <script>
//以 function 這個關鍵字的來建立類別 (class)
var Person = function (firstName) {
  //以 this 關鍵字作為實體本身
  this.firstName = firstName;
};

//用 new 關鍵字來實體化 (變成物件)
var person1 = new Person("Alex");
var person2 = new Person("Bill");

//輸出兩個 person 的 firstName
console.log( person1.firstName );
console.log( person2.firstName );
</script>
```

```
</body>
</html>
```

14-3: 擴展類型功能

範例 14-3-1.html

```
<html>
<head>
  <title>範例 14-3-1.html</title>
</head>
<body>

  <script>
    //以 function 這個關鍵字的來建立類別 (class)
    var Person = function (firstName) {
      //以 this 關鍵字作為實體本身
      this.firstName = firstName;
    };

    //使用 prototype，為 Person 類別定義了方法 sayHello()
    //共同的屬性或方法，不見得一定要在類別中定義
    //也可以透過 prototype 來建立 Person 原型的方法
    Person.prototype.sayHello = function() {
      console.log("Hello, I'm " + this.firstName);
    };

    //用 new 關鍵字來實體化 (變成物件)
    var person1 = new Person("Alex");
    var person2 = new Person("Bill");

    //輸出兩個 person 的 firstName
    console.log( person1.firstName );
    console.log( person2.firstName );

    person1.sayHello(); // "Hello, I'm Alice"
    person2.sayHello(); // "Hello, I'm Bob"
  </script>
```

```
</body>
</html>
```

注意：請勿修改原生原型

在這裡是在設定自己建立的物件的原型，不要嘗試修改預設的原生原型（例如：String.prototype），也不要無條件地擴充原生原型，若要擴充也應撰寫符合規格的測試程式；不要使用原生原型當成變數的初始值，以避免無意間的修改。

Module 15. 物件導向使用 class

15-1: 類別基本定義方式

在 ES6 之後，我們可以直接使用 class 關鍵字，來建立類別：

範例

```
class Person {

}
```

15-2: 建構函式

所謂建構函式（constructor），即是在我們透過 `let person = new Person();` 來實體化（建立物件）的時候，會優先執行 constructor 函式內部的程式碼，猶如初始化一樣，是建立類別時，一定要定義的函式。

我們可以使用 constructor 關鍵字新增建構函式：

範例

```
class Person {
  constructor() {
    console.log('Initialized');
  }
}
```

15-3: 屬性與方法

我們可以在 class 當中，建立自己的屬性與方法，其中方法的定義，不一定要關鍵字 function，在 constructor 中，也可以加入參數，作為屬性的預設值：

範例

```
class Person {
    constructor(height, width) {
        this.height = height;
        this.age = null;
    }

    getInfo(){
        return `Height: ${this.height}, Age: ${this.age}`;
    }

    setAge(age){
        this.age = age;
    }
}
```

範例 15-3-1.html

```
<html>
<head>
    <title>範例 15-1-1.html</title>
</head>
<body>

    <script>
class Person {
    constructor(height, width) {
        this.height = height;
        this.age = null;
    }

    getInfo(){
        return `Height: ${this.height}, Age: ${this.age}`;
    }
}
```

```
        setAge(age){
            this.age = age;
        }
    }

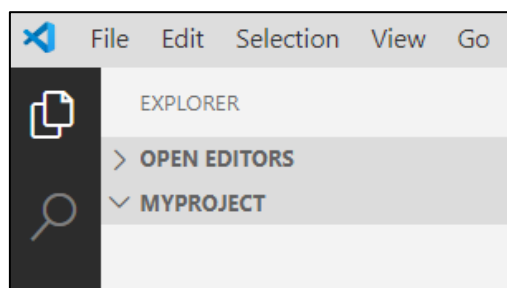
    let obj = new Person(160);
    obj.setAge(17);

    document.write( obj.getInfo() );
</script>
</body>
</html>
```

Module 16. 使用 babel

16-1: 使用 npm 建立專案

我們先在權限充足的地方，建立一個資料夾，例如「myProject」，再由 Visual Studio Code 開新視窗後，開啟 myProject 資料夾。



(圖) 開啟後的樣子

接下來按 `Ctrl + ``，開啟終端機，並輸入 `npm init` 指令，按下 `enter`，來初始化專案，同時建立 `package.json`。

```
Microsoft Windows [版本 10.0.18363.720]
(c) 2019 Microsoft Corporation. 著作權所有，並保留一切權利。
C:\Users\Owner\myProject>npm init
```

(圖) 在終端機輸入 npm init，按下 enter

```
C:\Users\Owner\myProject>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

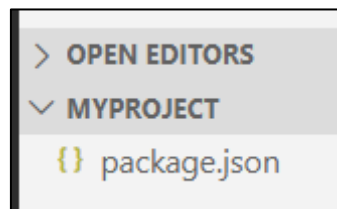
Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (myproject)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\Owner\myProject\package.json:

{
  "name": "myproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this OK? (yes) yes
C:\Users\Owner\myProject>
```

(圖) 會詢問專案的資訊，不確定就一直按 enter，日後可以再修正

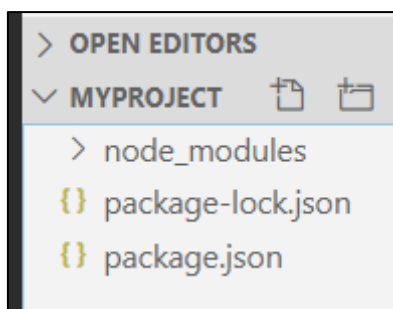


(圖) 會看到新增的 package.json

16-2: 安裝和設定 babel

在終端機輸入指令：

```
> npm install --save-dev @babel/core @babel/preset-env @babel/cli @babel/node
> npm install babel-cli -g
```



(圖) 目前專案資料夾的內容

結構說明：

- node_modules：
 - 就是我們透過 npm 下載下來的套件跟工具，都會放在這個資料夾裡面，通常這包會很大，所以平常不用備份，只要備份 package.json，我們可以透過 npm i --save 將相依的套件陸續安裝回來。
- package.json：
 - 關於這整包專案所有的資訊，包含我們安裝的套件版本，專案版本，npm 指令都可以在這個 json 檔案裡面找得到，之後要搬移專案重新安裝套件也需要靠這個 json 檔案
- package-lock.json：
 - package-lock.json 是 npm5 版本新增的，是專門紀錄 package.json 裡面更細節的內容，例如安裝的套件的詳細版本，或是確認你的 dependency（相依性）是被哪個函式庫所要求的等等。

再新增一個 .babelrc 檔案，裡面寫入：

範例

```
{
  "presets": [
    "@babel/preset-env"
  ],
  "plugins": []
}
```

原則上 babel 的環境建置完成。

16-3: 引入和匯出模組

我們在 myProject 專案資料夾裡面，新增一個檔案，叫作 module.js，再透過

export 關鍵字來匯出模組內容：

範例

```
export function function1() {  
  console.log('f1')  
}  
  
export function function2() {  
  console.log('f2')  
}  
  
export default function1;
```

再新增 test.js，使用 import 關鍵字引入自定義的模組 module.js 來操作：

範例

```
import defaultExport, { function1, function2 } from './module.js';  
  
defaultExport();  
function1();  
function2();
```

再執行指令：

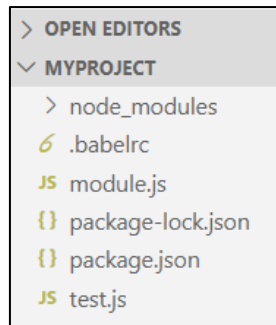
> babel-node test.js

上面是 babel 6 的指令，目前 babel 版本為 7.x，我們要用以下指令：

> npx babel-node test.js

```
C:\Users\Owner\myProject>npx babel-node test.js  
f1  
f1  
f2  
  
C:\Users\Owner\myProject>
```

(圖) 使用 npx babel-node 指令來執行 test.js



(圖) 當前的專案檔案結構

參考資料：

運行 nodejs 項目，npm start 啟動項目 import 報錯，SyntaxError: Cannot use import statement outside a module

https://blog.csdn.net/ll837448792/article/details/103307796?depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2&utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2

Module 17. 在專案上使用 webpack

17-1: 安裝 webpack

在 myProject 專案下，開啟終端機，輸入以下指令：

```
> npm install babel-loader webpack webpack-cli --save-dev
```

```
> npm install webpack -g
```

```
> npm install webpack-cli -g
```

新增一個檔案，叫作 myLib.js，進行 webpack 的 bundle 示範，裡面輸入：

範例

```
console.log("Hello World");
```

建立一個 webpack.config.js 檔案，內容如下：

範例

```
const path = require('path');
module.exports = {
  entry: './myLib.js',
  output: {
```

```

    filename: 'myLib.bundle.js',
    path: path.resolve(__dirname, './'),
  }
};

```

- entry :
 - 是我們取得 js 檔案的路徑，範例是 myLib.js。
- output :
 - 打包輸出後的檔案，檔名在 myLib 後面加了個 bundle，來確認它是打包過後的檔案。

回終端機輸入指令：

> webpack --mode development

```

C:\Users\Owner\myProject>webpack --mode development
Hash: 33959e08bb923e5eaa09
Version: webpack 4.42.1
Time: 45ms
Built at: 2020-04-06 2:27:02

```

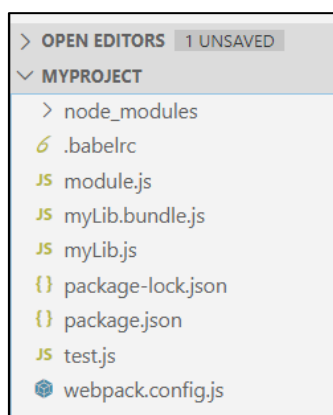
Asset	Size	Chunks	Chunk Names
myLib.bundle.js	3.8 KiB	main [emitted]	main

```

Entrypoint main = myLib.bundle.js
[./myLib.js] 54 bytes {main} [built]

```

(圖) 打包好之後，會產生 myLib.bundle.js 的檔案



(圖) 目前專案的結構

17-2: 安裝 loader

接下來，我們修改 webpack.config.js，加入 module 的設定資訊：

範例

```
const path = require('path');
module.exports = {
  entry: './myLib.js',
  output: {
    filename: 'myLib.bundle.js',
    path: path.resolve(__dirname, './'),
  },
  module: {
    rules: [{
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      },
      exclude: /(node_modules)/,
      test: /\.js$/
    }]
  }
};
```

接下來，我們修改 myLib.js 的內容：

範例

```
let myName = "york",
    numA = 20,
    numB = 7;

let content = `My name is ${myName}, My lucky number is ${2*(numA + numB)}`;

let gretting = ()=>console.log(` Hi, Arrow!, ${content}`)

gretting()
```

回終端機輸入指令：

> webpack -p

17-3: 功能模組化

ES6 的模組自動採用嚴格模式，不管你有沒有在模組頭部加上"use strict";。模組功能主要由兩個命令構成：export 和 import。export 命令用於規定模組的對外介面，import 命令用於輸入其他模組提供的功能。

範例

```
export let name = 'leo';
export let age= 30;

//也可以按照下面的寫法，兩種寫法都一樣
let name= 'leo';
let age= 30;
export {name, age};
```

範例

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

//也可以按照下面的方法
function sum(x, y) {
    return x + y;
};
export {sum}
```

範例

```
let name= 'leo';
let age= 30;
function sum(x,y){
    return x+y;
}
export {
    name as xm,
    sum as qh1,
    sum as qh2
}
```

export default 為模組指定預設輸出：

範例

```
export default function () {  
    console.log('foo');  
} // 第一種寫法  
  
export default function foo() {  
    console.log('foo');  
} // 第二種寫法  
  
// 也可以是下面的寫法  
function foo() {  
    console.log('foo');  
}  
export default foo;
```

export default 命令用於指定模組的預設輸出。顯然，一個模組只能有一個預設輸出，因此 export default 命令只能使用一次。所以，import 命令後面才不用加大括號，因為只可能唯一對應 export default 命令。

範例

```
// 正確  
var a = 1;  
export default a;  
  
// 正確  
export var a = 1;  
  
// 正確  
export default 123;
```

Module 18. ES7 的 await 和 async

18-1: 改善 Promise 缺點

雖然 Promise 以 `.then()` 和 `.catch()` 將 `callback hell` 的問題簡化了，但透過先前範例的展示，Promise 程式的撰寫還是稍嫌冗餘，於是我們可以透過 `async` 關鍵字來建立非同步的函式，在裡面透過 `await` 關鍵字來使用非同步的函式或工具。

在 ES7（ECMAScript 2016）完整加入了 `async`、`await`，讓程式排版看起來更簡潔。

範例 `await.js`

```
const fs = require('fs');
const util = require('util');
const readFile = util.promisify(fs.readFile); // 讓 readFile 可以使用 await 關鍵字
const writeFile = util.promisify(fs.writeFile); // 讓 writeFile 可以使用 await 關鍵字

//IIFE, Immediately-Invoked Function Expressions
(async function(){
  try {
    let contents = await readFile("tmp.txt", "utf8"); //讀取 tmp.log
    await writeFile("await01.txt", `${contents} I'm ironman...`); //寫入 await01.log
    contents = await readFile("await01.txt", "utf8"); //讀取 await01.log
    await writeFile("await02.txt", `${contents} I'm spiderman...`); //寫入 await02.log
    contents = await readFile("await02.txt", "utf8"); //讀取 await02.log
    await writeFile("await03.txt", `${contents} I'm superman...`); //寫入 await03.log
    contents = await readFile("await03.txt", "utf8"); //讀取 await03.log
    await writeFile("await04.txt", `${contents} I'm batman...`); //寫入 await04.log
    console.log(`await.js 執行完成`);
  } catch(err) {
    throw err;
  }
})();
```

註：

使用 `await`，區塊外必須使用 `async` 函式，否則會拋出錯誤。

SyntaxError: await is only valid in async function

18-2: await 使用時機

建立非同步的函式

範例

與一般函式相同，在函式前面加上 `async` 即可：

```
async function 函式名稱() {  
    ...  
}  
  
let func = async function() {  
    ...  
}  
  
let func = async () => {  
    ...  
}
```

18-3: async 的使用

使用非同步函式

範例

```
async function 新函式(){  
    await 既有函式(); // 使用非同步方式開發的函式  
}
```

註：既有函式，也需要是非同步函式