

JavaScript

網頁程式設計

DARREN YANG 楊德倫

內容

Module 1. 測試及除錯工具.....	1
1-1: Chrome 開發者工具.....	1
1-2: Console 面板.....	5
1-3: Network 面板.....	8
Module 2. 使用 script 標籤.....	14
2-1: DOM 簡介.....	14
2-2: 動態新增頁面標籤內容.....	16
2-3: 取得標籤元素.....	16
Module 3. 常數和變數宣告.....	22
3-1: var、let 和 const.....	23
3-2: 識別字的規則.....	24
3-3: var 和 let 的主要差異.....	27
Module 4. 基本類型.....	28
4-1: Number、Boolean 和 String.....	28
4-2: 轉換為 Number.....	32
4-3: 轉換為 String.....	34
Module 5. 運算子.....	35
5-1: 算術運算子.....	35
5-2: 關係運算子.....	37
5-3: 邏輯運算子.....	43
Module 6. String.....	49
6-1: 字串的標示方式.....	49
6-2: 字串的跳脫表示法.....	52
6-3: 字串的常用方法.....	53
Module 7. 取得標籤元素.....	57
7-1: 使用 ES3 的方法.....	57
7-2: querySelector().....	61
7-3: querySelectorAll().....	61
Module 8. 流程控制.....	62
8-1: 選擇敘述.....	63
8-2: 迴圈.....	69
8-3: break 和 continue.....	75
Module 9. Object 類型.....	77
9-1: Object 類型的特點.....	77
9-2: Object 表達式.....	78
9-3: for/in 迴圈.....	83
Module 10. Array 類型.....	85

10-1: Array 類型的特點	85
10-2: Array 表達式	86
10-3: Array 的方法	87
Module 11. JSON.....	96
11-1: JSON 字串規則	96
11-2: Object 和 Array 的複製	98
11-3: 編輯 JSON 檔.....	100
Module 12. 函式的定義.....	101
12-1: 基本型函式.....	101
12-2: 匿名函式.....	107
12-3: 箭頭函式.....	110
Module 13. Scope 變數領域	118
13-1: 全域變數.....	118
13-2: 區域變數.....	119
13-3: closure.....	121
Module 14. 物件導向使用 prototype	122
14-1: 自訂功能物件.....	122
14-2: 使用 function 自訂類型.....	123
14-3: 擴展類型功能.....	124
Module 15. 時間與計時器.....	125
15-1: Date 物件	125
15-2: setTimeout 用法	132
15-3: setInterval 用法	136
Module 16. 數學物件.....	137
16-1: 亂數.....	139
16-2: 三角函數.....	141
16-3: 環狀排列物件.....	143
Module 17. window 物件	146
17-1: window 物件的方法	146
17-2: window 的子物件	152
17-3: document 的常用屬性	155
Module 18. 事件處理.....	158
18-1: 標籤內的事件處理器.....	158
18-2: addEventListener.....	160
18-3: 滑鼠事件.....	162
Module 19. AJAX.....	164
19-1: 不刷新頁面更新內容.....	164
19-2: XMLHttpRequest.....	166

19-3: fetch()方法.....	170
Module 20. 操作 DOM.....	172
20-1: 取得 DOM 元素.....	172
20-2: 建立 DOM 元素.....	175
20-3: 刪除 DOM 元素.....	180
Module 21. 深入 DOM 元素.....	181
21-1: 屬性操作.....	181
21-2: 自訂屬性.....	182
21-3: 元素與樣式.....	183
Module 22 正規表示法.....	187
22-1: 單一字元表示法.....	190
22-2: 多字元表示法.....	191
22-3: 表單送出前的檢查.....	192
Module 23. Canvas.....	193
23-1 繪製圖片.....	194
23-2 動態效果.....	204
23-3 碰撞偵測.....	211

課程範例網址:

<https://github.com/telunyang/javascript-uiux-class>

Module 1. 測試及除錯工具

1-1: Chrome 開發者工具

Chrome 開發者工具是內建於 Google Chrome 中的 Web 開發和測試工具

網址：<https://developers.google.com/web/tools/chrome-devtools?hl=zh-tw>



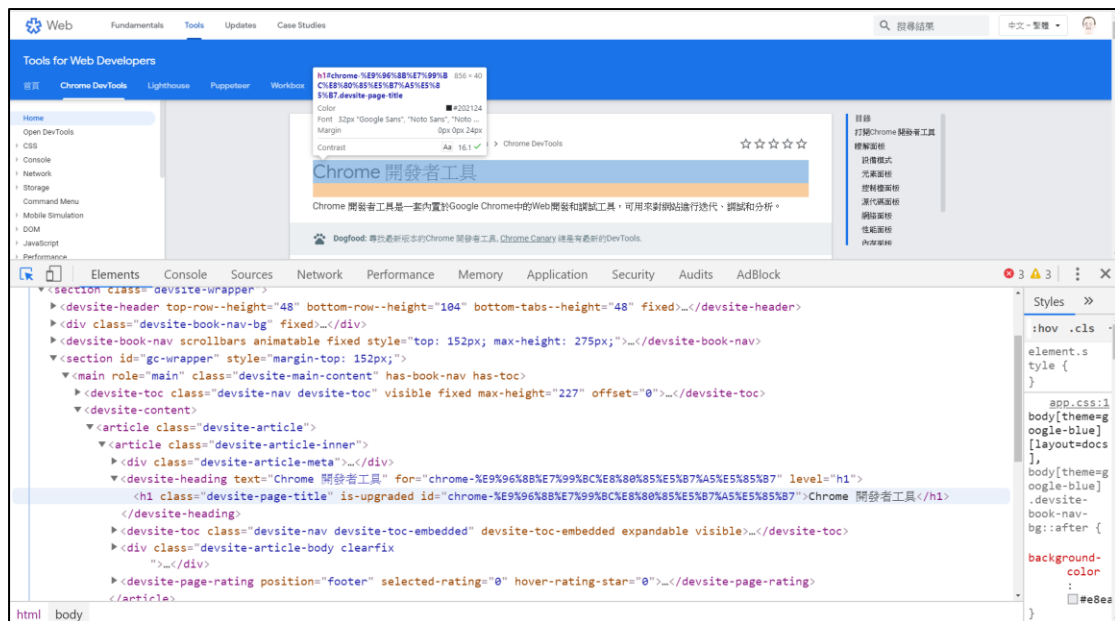
圖 Chrome 開發者工具的說明網頁

開啟開發工具(dock)

- F12
- Ctrl + Shift + I

檢查 HTML 元素

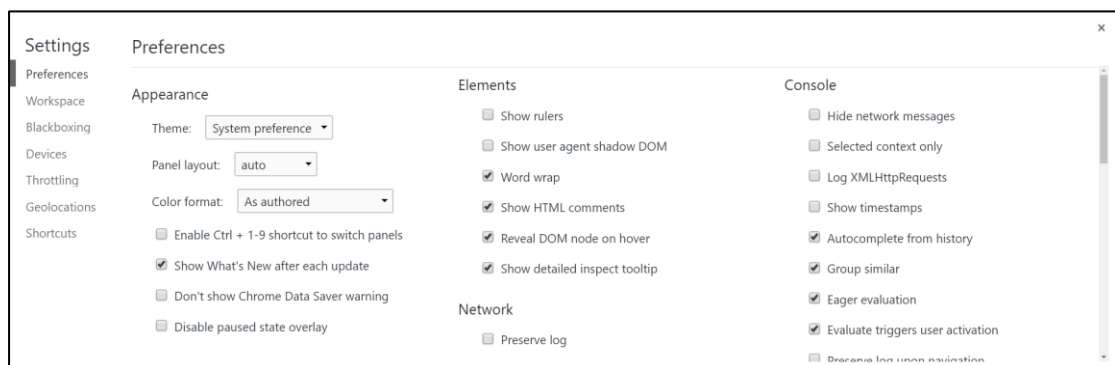
- Ctrl + Shift + C (追蹤滑鼠移過網頁元素所在位置的狀態)
- 網頁內容任意處按滑鼠右鍵→檢查



(圖) 檢查元素

補充說明

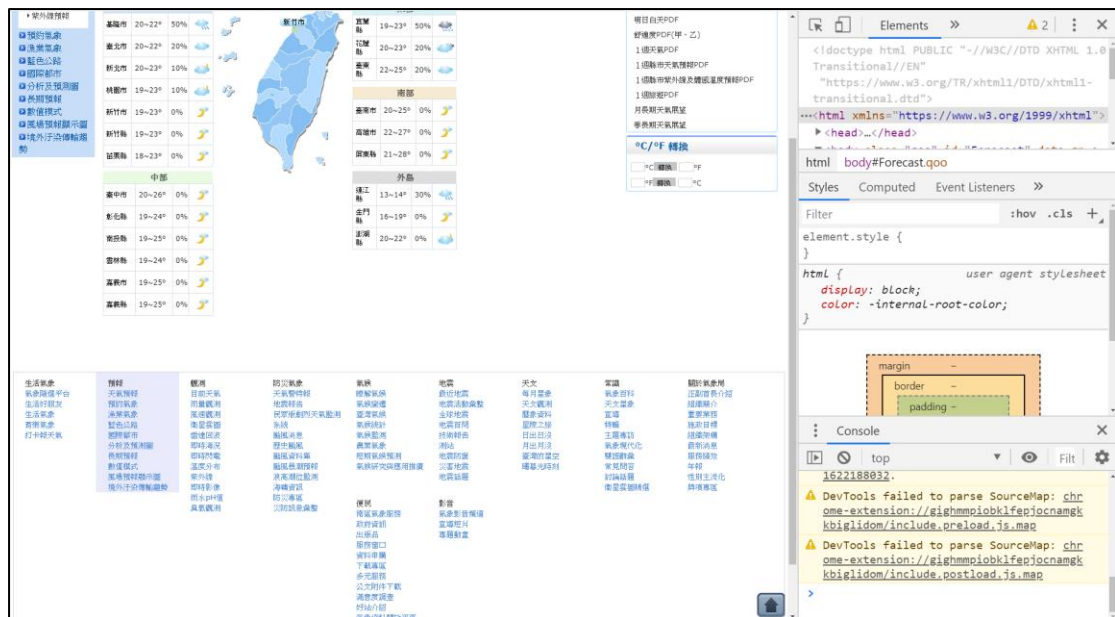
開啟 Chrome 開發者工具以後，按下 F1，可以看到一些偏好設定，方便我們設定開發工具，例如顯示外觀、模擬裝置、自訂地理位置、快捷鍵等。



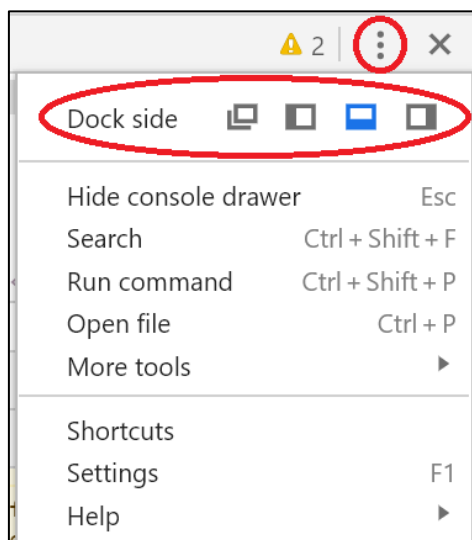
(圖) Chrome 開發者工具偏好設定

開啟開發工具後，常用快速鍵：

- Ctrl + Shift + D 切換檢查元素的 dock side

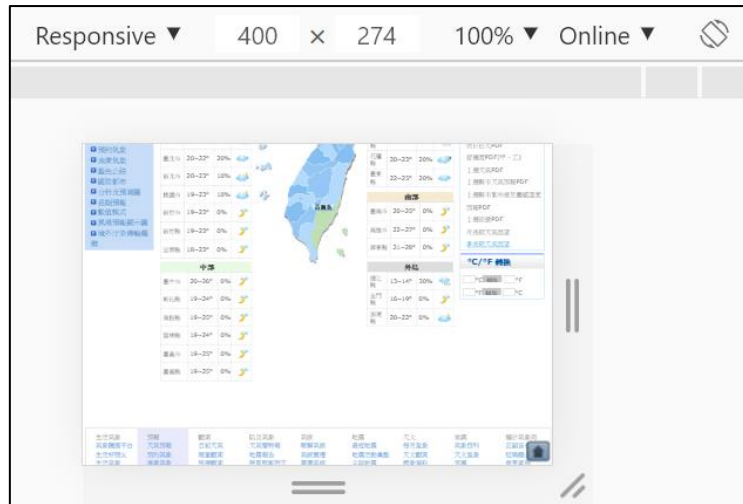


(圖) 切换 dock side，從下方到右側

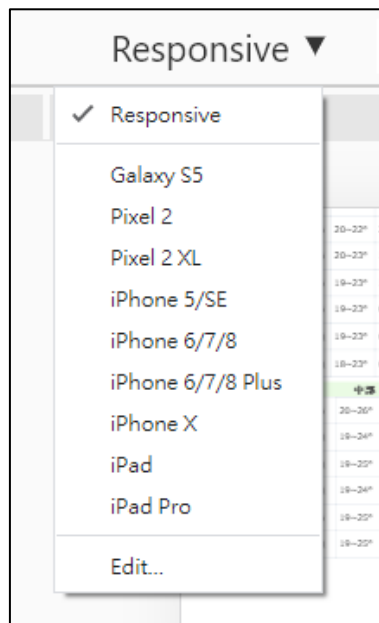


(圖) 按下三個點的圖示，也可以選擇 dock side

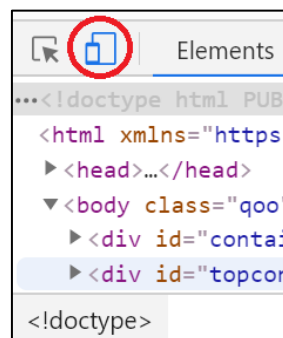
- Ctrl + Shift + M 開啟模擬裝置模式(切换裝置工具欄)



(圖) 可選擇不用的行動裝置，或自訂寬高，來顯示網頁



(圖) 選擇裝置來觀看網頁

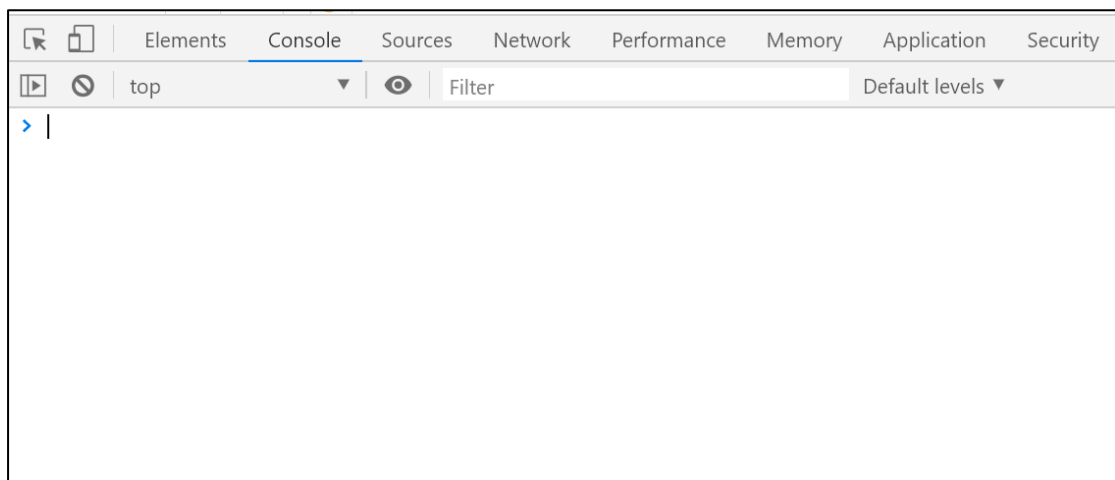


(圖) 等同按下切換裝置工具欄

- Ctrl + O 尋找 HTML 當中的檔名
- Ctrl + R 或 F5 刷新頁面
- Ctrl + F5 清除快取後，刷新頁面(重新從伺服器端請求下載 HTML)
- Ctrl + L 清除 Console
- Shift + Enter 在 Console 中斷行(或多行)

1-2: Console 面板

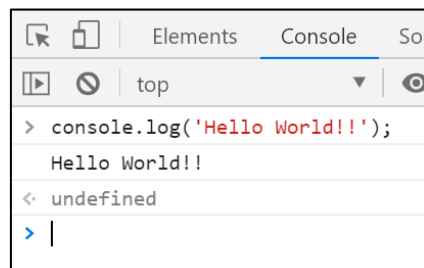
我們可以使用 Console 面板，進行 JavaScript 的開發測試與除錯。



(圖) Console 面板

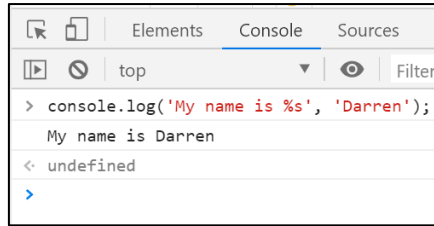
以下是使用的範例：

- 使用 console.log 直接輸出



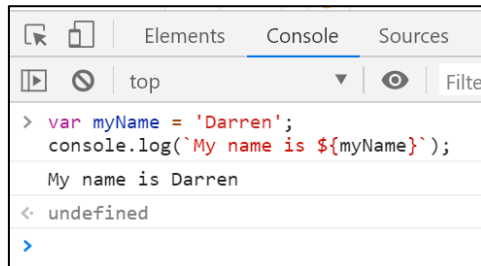
(圖) 輸出結果

- 使用 %s 輸入字串



(圖) 第二個字串引數值，可以帶入 %s 當中

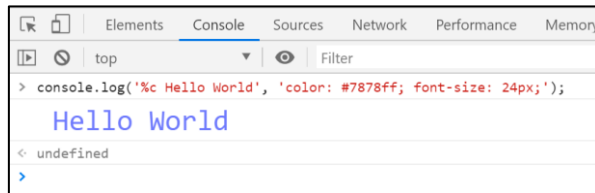
- 使用 es6 的 template strings



(圖) 透過 template 進行字串輸出

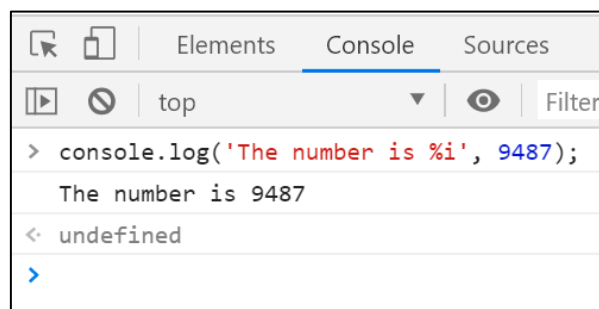
備註: 若需要換行，可以在輸入第一行以後，按下 Shift + Enter，即可換行

- 使用 %c 輸出時，加入樣式 (css style)



(圖) 設定 css style 的文字輸出

- 使用 %i 伴隨字串格式來輸出數字的值



(圖) 輸出數字的值

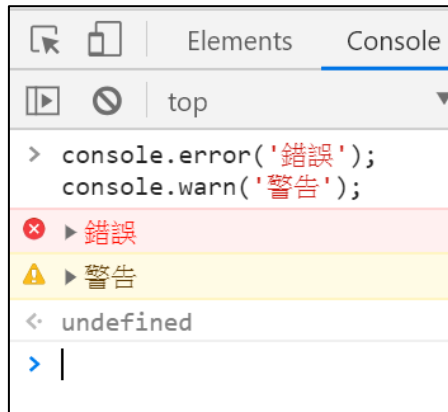
常見的 % (格式化字串所使用的符號):

- %s: 輸入字串

- %i 或 %d: 輸入數值
- %f: 輸入浮點數
- %c: css style

若有特殊的訊息(例如錯誤、警告等)要顯示出來，可以使用 `console.error`、`console.warn`：

- `console.error()` JavaScript 出現錯誤訊息時，會出現紅色的文字訊息與圖示
- `console.warn()` JavaScript 出現警告訊息時，會出現黃色的文字訊息與圖示



(圖) 顯示結果會包括圖示

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

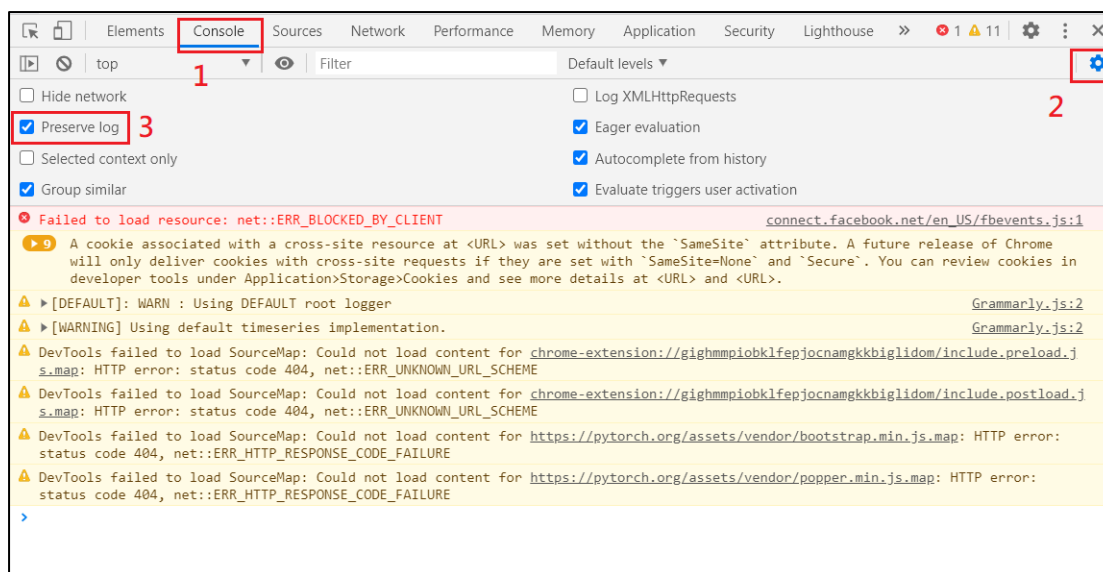
    console.log(7 - 3 + 11);
    // 輸出 15

    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>
```

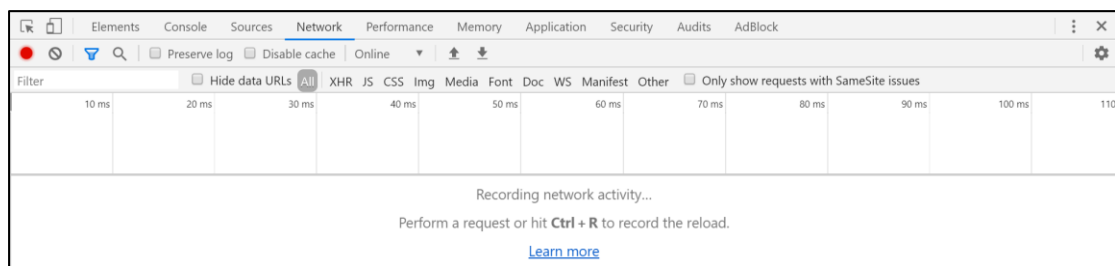
我們可以開啟 Preserve log，讓每次刷新頁面時，還能保有先前的輸出訊息。這個部分很重要，因為未來在使用非同步傳輸（Ajax、Fetch）結合動態網頁程式語言（例如 PHP）的時候，在資料傳輸上有問題，可以在刷新頁面、重新執行程式碼後，保留先前的輸出結果，讓我們在 debug 上更為便利。



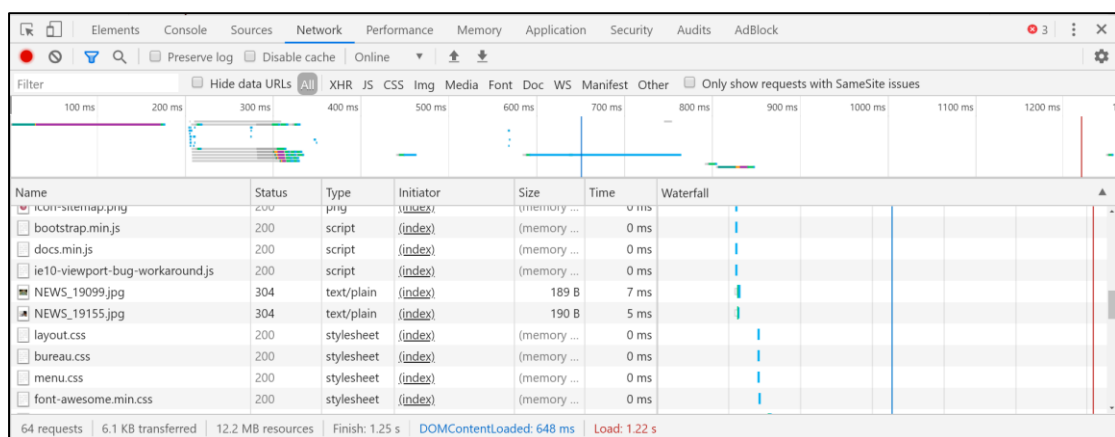
圖：開啟 Preserve log

1-3: Network 面板

Network 面板會顯示出所有網路請求的詳細訊息記錄，包括狀態、資源類型、大小、所需時間、HTTP request header 和 response header 等等，明確找出哪些請求比預期還要耗時，並加以調整，是優化網頁的重要工具。



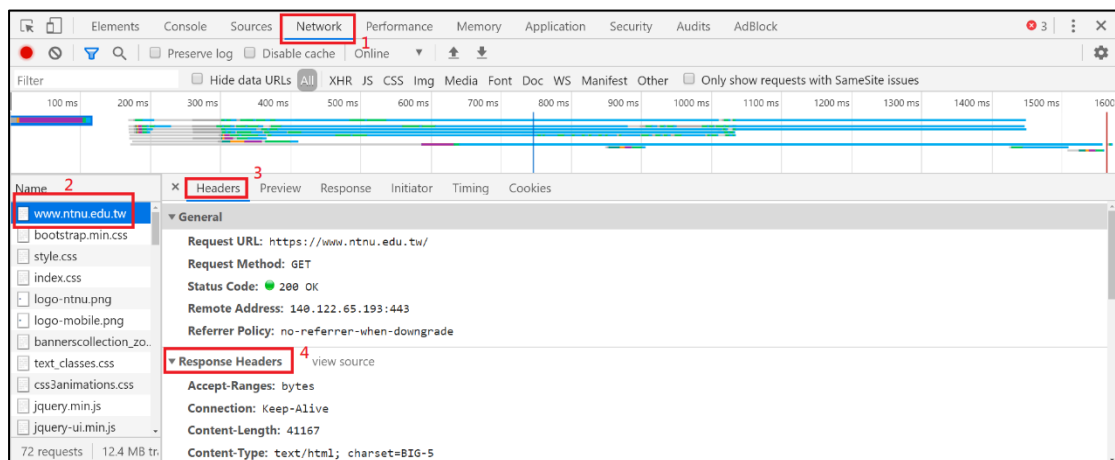
(圖) Network 面板會記錄任何的網路活動



(圖) 記錄網頁讀取的資訊與下載順序

我們可以透過 Headers，來了解網頁請求的狀況。開啟 Headers 的流程為：

1. 開啟 Network 面板
2. Ctrl + R 或是 F5 刷新頁面
3. 點選左側的檔案名稱
4. 點選 Headers



(圖) 觀看檔案的 Headers 內容

最後，讓我們來看一下 Request Headers 與 Response Headers。
(下列表格參考[維基百科](#))

Request Headers (請求標頭)

標頭欄位	說明	範例
Accept	能夠接受的回應內容類型 (Content-Types)。	Accept: text/plain
Accept-Encoding	能夠接受的編碼方式列表。參考 HTTP 壓縮 。	Accept-Encoding: gzip, deflate
Accept-Language	能夠接受的回應內容的自然語言列表。	Accept-Language: en-US
Authorization	用於超文字傳輸協定的認證資訊	Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Cache-Control	用來指定在這次的請求/回應鏈中的所有快取機制 都必須遵守的指令	Cache-Control: no-cache
Connection	該瀏覽器想要優先使用的連接類型	Connection: keep-alive Connection: Upgrade
Cookie	之前由伺服器通過 Set-Cookie 傳送的一個 超文字傳輸協定 Cookie	Cookie: _ga=GA1.3.1322956465.1572335045; locale=zh_TW;

標頭欄位	說明	範例
		<pre>_gid=GA1.3.1110994946.1584940974; _gat_gtag_UA_141775379_1=1</pre>
Content-Length	以 八位位元組陣列（8 位元的位元組）表示的請求體的長度	Content-Length: 348
Content-Type	請求多媒體類型（用於 POST 和 PUT 請求中）	Content-Type: application/x-www-form-urlencoded
Host	伺服器的域名(用於虛擬主機)，以及伺服器所監聽的埠號。如果所請求的埠是對應的服務的標準埠，則埠號可被省略。	<pre>Host: en.wikipedia.org:80 Host: en.wikipedia.org</pre>
Origin	發起一個針對 跨來源資源共享 的請求（要求伺服器在回應中加入一個『存取控制-允許來源』（'Access-Control-Allow-Origin'）欄位）。	Origin: http://www.example-social-network.com
Pragma	每次發出請求，確認有沒有新的檔案；若有 Cache-Control，則直接看 Cache-Control。	Pragma: no-cache
Referer	表示瀏覽器所存取的前一個頁面，正是那個頁面上的某個連結將瀏覽器帶	<pre>Referer: http://en.wikipedia.org/wiki/Main_Page</pre>

標頭欄位	說明	範例
	到了目前所請求的這個頁面。	
User-Agent	瀏覽器的瀏覽器身分標識字串	User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/21.0
Upgrade	要求伺服器升級到另一個協定。	Upgrade: HTTP/2.0, SHTTP/1.3, IRC/6.9, RTA/x11

Response Headers (回應標頭)

標頭名	說明	例子
Access-Control-Allow-Origin	指定哪些網站可參與到跨來源資源共享過程中	Access-Control-Allow-Origin: *
Accept-Ranges	這個伺服器支援哪些種類的部分內容範圍	Accept-Ranges: bytes
Age	這個物件在代理快取中存在的時間，以秒為單位	Age: 12
Allow	對於特定資源有效的動作。針對 HTTP/405 這一錯誤代碼而使用	Allow: GET, HEAD
Cache-Control	向從伺服器直到客戶端在內的所有快取機制告知，它們是否可以快取這個物件。其單位為秒	Cache-Control: max-age=3600
Connection	針對該連接所預期的選項	Connection: close
Content-Disposition	一個可以讓客戶端下載檔案並建議檔	Content-Disposition: attachment; filename="fname.ext"

標頭名	說明	例子
	名的頭部。檔名需要用雙引號包裹。	
Content-Encoding	在資料上使用的編碼類型。	Content-Encoding: gzip
Content-Language	內容所使用的語言	Content-Language: da
Content-Length	回應訊息體的長度，以 位元組（8 位元為一位元組）為單位	Content-Length: 348
Content-Location	所返回的資料的一個候選位置	Content-Location: /index.htm
Content-Type	目前內容的 MIME 類型	Content-Type: text/html; charset=utf-8
Date	此條訊息被傳送時的日期和時間	Date: Tue, 15 Nov 1994 08:12:31 GMT
ETag	對於某個資源的某個特定版本的一個識別碼，通常是一個訊息雜湊	ETag: "737060cd8c284d8af7ad3082f209582d"
Expires	指定一個日期/時間，超過該時間則認為此回應已經過期	Expires: Thu, 01 Dec 1994 16:00:00 GMT
Last-Modified	所請求的物件的最後修改日期	Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT
Location	用來進行重新導向，或者在建立了某個新資源時使用。	Location: http://www.w3.org/pub/WWW/People.html
Pragma	每次發出請求，確認有沒有新的檔案；若有 Cache-	Pragma: no-cache

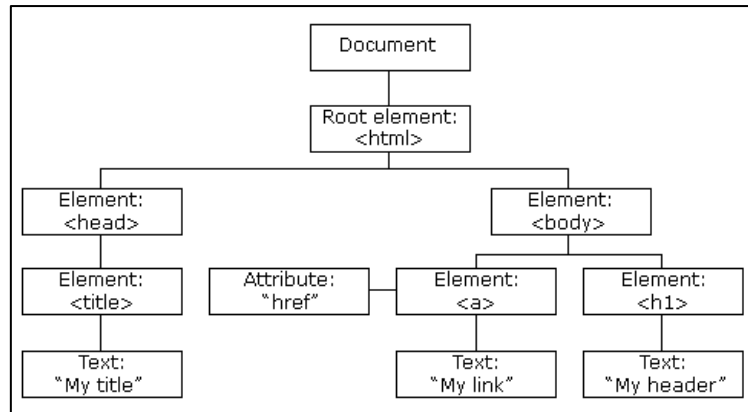
標頭名	說明	例子
	Control，則直接看 Cache-Control。	
Refresh	用於設定可定時的重新導向跳轉。	Refresh: 5; url=http://www.w3.org/pub/WWW/People.html
Server	伺服器的名字	Server: Apache/2.4.1 (Unix)
Set-Cookie	HTTP cookie	Set-Cookie: UserID=JohnDoe; Max-Age=3600; Version=1
Status	用來說明目前這個超文字傳輸協定回應的狀態。	Status: 200 OK

Module 2. 使用 script 標籤

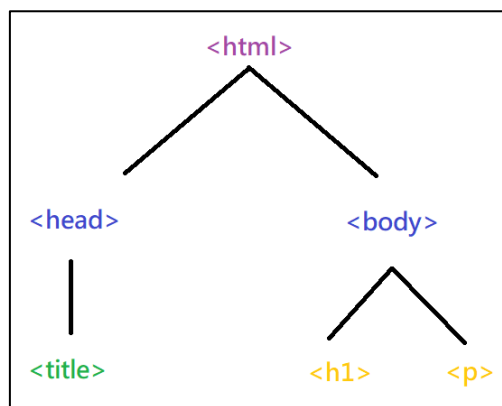
2-1: DOM 簡介

文件物件模型（Document Object Model, DOM）是 HTML、XML 和 SVG 文件的程式介面。它提供了一個文件（樹）的結構化表示法，並定義讓程式可以存取並改變文件架構、風格和內容的方法。DOM 提供了文件以擁有屬性與函式的節點與物件組成的結構化表示。節點也可以附加事件處理程序，一旦觸發事件就會執行處理程序。本質上，它將網頁與腳本或程式語言連結在一起。

簡而言之，就是把一份 HTML 文件內的各個元素，包括 html 字串、文字、圖片等等，全都載入到瀏覽器中，瀏覽器會把這些元素都定義成物件，這些物件最終會變成一種樹狀結構，再透過 JavaScript 來調用 HTML API，進而操控 HTML 元素。



(圖) DOM 示意圖



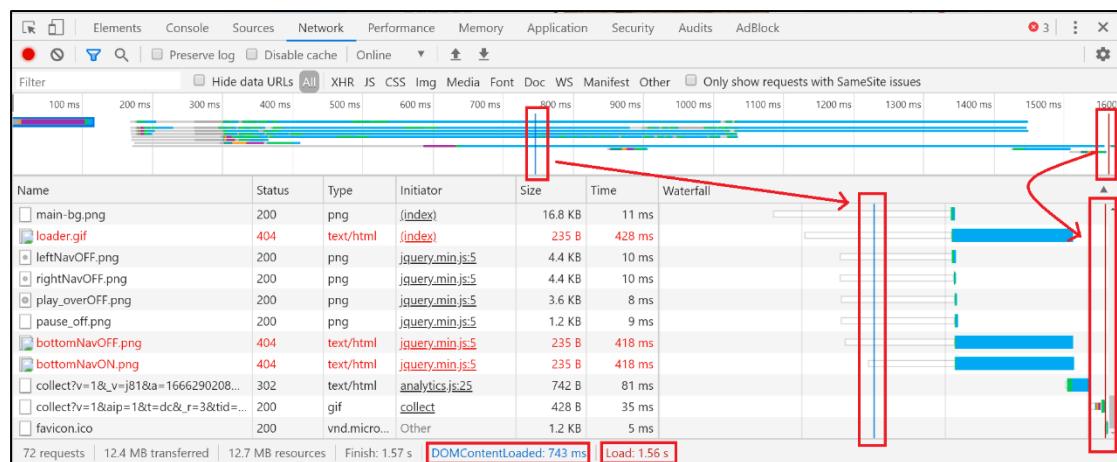
(圖) 簡化版 DOM 示意圖

補充說明

DOMContentLoaded 是在 DOM 被載入完畢時的事件，它會計算 HTML 被載入的時間。例如我們先在瀏覽器輸入 URL，網頁一開始是空白的，過一陣子，網頁會出現基本的架構和格局，我們可以想像成所有 HTML 被下載到我們的本機瀏覽器中。

隨著載入時間的增加，最後慢慢可以看到完整的文字、圖片、音檔等，這時候 Load 便會被觸發，同時顯示載入時間。

DOMContentLoaded 會先被觸發，之後才會觸發 Load。



(圖) DOMContentLoaded 與 Load

2-2: 動態新增頁面標籤內容

我們使用 `document.write()` 方法，在 `<body>` 裡新增文字。

範例 2-2-1.html

```
<html>
<head>
  <title>2-2-1.html</title>
  <script>
    document.write("Hello World<br />My name is XXX")
  </script>
</head>
<body>

</body>
</html>
```

2-3: 取得標籤元素

取得元素的方式有三種：

- Id (元素 Id) : `<div id="foo">`
- ClassName (元素樣式名稱) : `<div class="foo">`
- TagName (元素名稱) : 就是一般的 HTML 元素，如 `div`, `h1`, `h2`, `h3` ... 等
- Name (屬性有 name 的值)

取得元素內部文字的方式有三種：

- innerHTML：標籤開頭與結尾間的 html 字串（包括文字）
- innerText：標籤開頭與結尾間的純文字
- value：元素中，有 value 屬性的值

我們可以透過 `document.getElementById("元素 ID")` 來取得標籤元素。

範例 2-3-1.html

```
<html>
<head>
  <title>2-3-1.html</title>
</head>
<body>
  <p id="myContent">Hello World</p>

  <script>
    var p = document.getElementById('myContent');
    alert( p.innerHTML );
  </script>
</body>
</html>
```

補充說明

上面的範例，有沒有發現 `<script>` 標籤被放到 `<body>` 裡面的最後面了？那是因為在 `DOMContentLoaded` 觸發的時候，若有 `<script>` 在 `<head>` 裡面，則會影響到 `DOMContentLoaded` 的運作，非得要等 `<script>` 執行完，才能繼續載入 HTML，所以實務上來說，我們會把自訂的 `<script>` 放到 `<body>` 裡面的最後面，這樣就可以等 HTML 都載入後，才開始執行 `<script>`。

下面的範例是用函式(function)先將要做的事情先行定義，再透過對元素註冊事件(event)，透過事件觸發來運作函式裡面所定義的內容。

範例 2-3-2.html

```
<html>
<head>
  <title>2-3-2.html</title>

  <script>
    function getValue(){
```

```
var p = document.getElementById('myContent');
alert( p.innerText );
}
</script>
</head>
<body>
  <p id="myContent" onclick="getValue()">Hello World</p>
</body>
</html>
```

範例 2-3-3.html
<pre><html> <head> <title>2-3-3.html</title> <script> function getValue(){ var p = document.getElementById('myContent'); alert(p.innerHTML); } </script> </head> <body> <p id="myContent" onclick="getValue()"> Hello World </p> </body> </html></pre>

範例 2-3-4.html
<pre><html> <head> <title>2-3-4.html</title> <script> function getValue(){ var btn = document.getElementById('myBtn'); alert(btn.value); } </script></pre>

```
</head>
<body>
  <button id="myBtn" value="Hi!" onclick="getValue()">按鈕</button>
</body>
</html>
```

document.getElementById() 通常會取得單一元素，但元素若是一個集合呢？我們通常會使用以下幾個方法：

- document.getElementsByTagName(元素名稱)
- document.getElementsByClassName(元素 class 名稱)
- document.getElementsByName(元素 name 屬性值)

取得集合後，還可以：

- .length：取得元素數量

document.getElementsByTagName(元素名稱)

範例 2-3-5.html

```
<html>
<head>
  <title>2-3-5.html</title>
  <style>
    <!--
      table, tr, th, td{
        border: 1px solid;
      }
    -->
  </style>
</head>
<body>
  <table>
    <thead>
      <tr>
        <th>流水編號</th><th>姓名</th><th>年紀</th>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td>1</td><td>張○○</td><td>25</td>
```

```
</tr>
<tr>
  <td>2</td><td>王○○</td><td>33</td>
</tr>
<tr>
  <td>3</td><td>林○○</td><td>19</td>
</tr>
</tbody>
</table>

<br /><br />

<table>
  <thead>
    <tr>
      <th>流水編號</th><th>姓名</th><th>年紀</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>4</td><td>江○○</td><td>24</td>
    </tr>
    <tr>
      <td>5</td><td>沈○○</td><td>42</td>
    </tr>
    <tr>
      <td>6</td><td>陳○○</td><td>23</td>
    </tr>
  </tbody>
</table>

<script>
var tables = document.getElementsByTagName("table");
alert("網頁中包含 " + tables.length + " 個 tables");
</script>
</body>
</html>
```


document.getElementsByClassName(元素 class 名稱)

範例 2-3-6.html

```
<html>
<head>
  <title>範例 2-3-6.html</title>
</head>
<body>
  <button class="btn">按鈕 1</button>
  <button class="btn">按鈕 2</button>
  <button class="btn">按鈕 3</button>

  <script>
    var buttons = document.getElementsByClassName("btn");
    alert("網頁中包含 " + buttons.length + " 個 buttons");
  </script>
</body>
</html>
```

document.getElementsByName(元素 name 屬性值)

範例 2-3-7.html

```
<html>
<head>
  <title>範例 2-3-7.html</title>
</head>
<body>
  <input type="radio" name="gender" value="男" /> 男
  <input type="radio" name="gender" value="女" /> 女

  <script>
    var radios = document.getElementsByName("gender");
    alert("網頁中包含 " + radios.length + " 個 radios");
  </script>
</body>
</html>
```

若是我們要改變標籤內部文字，我們可以透過以下方法來設定：

- document.getElementById(元素).innerText = '你的文字';
- document.getElementById(元素).innerHTML = 'html 字串';

範例 2-3-8.html

```
<html>
<head>
  <title>範例 2-3-8.html</title>
</head>
<body>
  <p id="paragraph01">金庸小說</p>
  <p id="paragraph02">金光布袋戲</p>

  <script>
    document.getElementById("paragraph01").innerText = '飛雪連天射白鹿，笑書神俠倚碧鴛';
    document.getElementById("paragraph02").innerHTML = '<span style="color: #7878ff;">南宮恨（黑白郎君）：別人的失敗，就是我的快樂啦！</span>';
  </script>
</body>
</html>
```

補充說明：

- 有時候我們會將 `<script></script>` 當中的程式碼，以模組化的概念，存放在一個 .js 格式的檔案當中，並且在 `<head></head>` 當中，以「`<script src="你的程式碼.js"></script>`」存放。
- 有時候會在 `<script>` 標籤中看到 `defer` 屬性，其功用類似將所有 HTML 都讀完後，才執行 `<script>` 當中的程式碼。

Module 3. 常數和變數宣告

我們可以透過變數名稱的宣告，來為「值」（文字、數值、運算結果、暫存資料等）取個名字。我們使用關鍵字 `var`、`let`、`const` 來宣告變數，例如 `var myName`，而所謂「關鍵字」，是指在程式語言中具有意義的單詞，通常不適合拿來作為變數命名使用。原則上，變數的賦值，通常由右往左，我們稱之為「賦值運算子」（Assignment operators）。

變數命名的規則，有很多種方式，例如駱駝命名法、匈牙利命名法等。駱駝命名法是一種慣例，看起來像駱駝的駱峰，變數以小寫字母開頭，除了第一個單詞外，其它單詞的首個字母都大寫，例如 `numberOfCandies`，或是我們上方的 `myName`，或是 `numberOfDays`。課程中，原則上使用駱駝命名法，但沒有特別強制規定，你也可以自由選擇自己慣用的方式。

3-1: var、let 和 const

var、let、const 都是宣告變數的關鍵字，var 跟 let 都是宣告一般自訂變數，而 const 是用來宣告自訂常數（顧名思義，常數原則上不可修改它的值）。

範例 3-1-1.html

```
<html>
<head>
  <title>範例 3-1-1.html</title>
  <script>
    var myName = 'Darren';
    console.log('我的名字:' + myName);

    let myAge = 18;
    console.log('我的年紀:' + myAge);

    const PI = 3.1415926;
    console.log('圓周率 = ' + PI);
  </script>
</head>
<body>

</body>
</html>
```

若是使用 const 宣告出來的變數，其值被修改，會發生什麼事？

範例 3-1-2.html

```
<html>
<head>
  <title>範例 3-1-2.html</title>
  <script>
    const PI = 3.1415926;
    PI = 3.14;
    console.log('圓周率 = ' + PI);
  </script>
</head>
<body>
```

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

```
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at 3-1-2.html:6
```

(圖) 型別錯誤: 對常數賦值。

`const` 關鍵字用於「常數」的宣告，有別於我們先前看到的「變數」，原則上不宜重覆賦值，也常用大寫的方法與變數加以區隔，例如 `const MAX_LENGTH`、`DB_HOST`、`DB_PASSWORD` 等。

3-2: 識別字的規則

識別字 (identifier) 為程式語言中依程式需求自行定義的名稱，舉凡程式中所用的各種名稱都屬於識別字，內建物件 (built-in object) 及 DOM 其內所用的名稱也屬於識別字，自行定義名稱時應該避免與其相衝突，同時，識別字也不可與保留字 (reserved words) 的名稱相同。

JavaScript 定義識別字可用任何 Unicode 符號，但是習慣上仍是以英文二十六的大小寫字母為主，另加上數字、底線符號及 dollar sign，如下

_	\$											
a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9			

範例 3-2-1.html

```
<html>
<head>
  <title>範例 3-2-1.html</title>
  <script>
    var _b;
    var $a;
    var $3;
```

```
var 3cats;

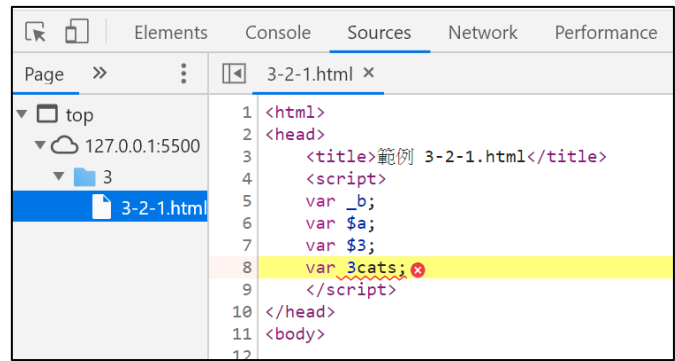
</script>

</head>

<body>


</body>

</html>
```



(圖) 3cats 是不合法的變數名稱

補充說明

我們另外補充說明「保留字」。在 JavaScript 中，有些識別字是保留給程式語言用的關鍵字，不能拿來作為變數名稱或是函式名稱。

我們應該避免使用 JavaScript 關鍵字：

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return

short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

我們應該避免使用 JavaScript 的物件、屬性和方法名稱：

Array	Date	eval	function	hasOwnProperty
Infinity	isFinite	isNaN	isPrototypeOf	length
Math	NaN	name	Number	Object
prototype	String	toString	undefined	valueOf

我們在之後會應用到 Window 物件，我們也要避免使用到它的屬性和方法：

alert	all	anchor	anchors	area
assign	blur	button	checkbox	clearInterval
clearTimeout	clientInformation	close	closed	confirm
constructor	crypto	decodeURI	decodeURIComponent	defaultStatus
document	element	elements	embed	embeds
encodeURI	encodeURIComponent	escape	event	fileUpload
focus	form	forms	frame	innerHeight
innerWidth	layer	layers	link	location
mimeType	navigate	navigator	frames	frameRate
hidden	history	image	images	offscreenBuffering
open	opener	option	outerHeight	outerWidth

packages	pageXOffset	pageYOffset	parent	parseFloat
parseInt	password	pkcs11	plugin	prompt
propertyIsEnumerable	radio	reset	screenX	screenY
scroll	secure	select	self	setInterval
setTimeout	status	submit	taint	text
textarea	top	unescape	untaint	window

我們應該避免使用 HTML 的事件名稱：

onblur	onclick	onerror	onfocus
onkeydown	onkeypress	onkeyup	onmouseover
onload	onmouseup	onmousedown	onsubmit

3-3: var 和 let 的主要差異

Javascript 透過 var 關鍵字宣告的變數，具有抬升 (Hoisting) 的性質，不易掌握變數的生命週期，同時 var 所宣告的全域變數，會成為 window 物件的屬性，我們日後會在課堂中提到。近年的 Javascript 版本，增加了 let 與 const 關鍵字，解決了 var 關鍵字的抬升問題 (Hoisting)。

我們來看看 var 在區域變數中的特性：

一般來說，我們會這樣宣告變數、初始化變數（給變數初始值）、輸出變數內容

```
var name = 'Darren';
console.log(name);
//輸出 Darren
```

換個方式

```
console.log(name);
var name = 'Darren';
```

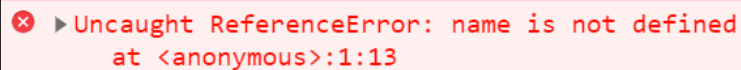
//這裡會輸出 undefined (Console 面板會不顯示輸出)

若是這樣子呢

```
console.log(name);
```

```
let name = 'Darren';
```

//應該輸出 undefined，卻拋出 **ReferenceError: name is not defined** 的訊息



✖ ▶ Uncaught ReferenceError: name is not defined
at <anonymous>:1:13

(圖) 參考錯誤: name 變數沒有被定義

這是因為 Javascript 在使用 var 宣告變數時，會將變數抬升 (hoisting) 到作用域 (程式區塊) 的前面，作用域內外的每一行都有機會使用到。

Module 4. 基本類型

4-1: 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 4-1-1.html

```
<html>
<head>
  <title>範例 4-1-1.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 4-1-2.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 4-1-3.html

```
<html>
<head>
```

```
<title>範例 4-1-3.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>
```

範例 4-1-4.html

```
<html>
<head>
  <title>範例 4-1-4.html</title>
  <script>
    //樣版字面值 (Template literals)
```

```

console.log(`你好！我是\nDarrenYang`);
console.log(`你好！我是
Darren Yang`);

//輸出 你好! 我是 Darren Yang
let fname = 'Darren';
let lname = 'Yang';
console.log(`你好! 我是 ${fname} ${lname}`);

//若是嵌入變數計算，可以這麼做
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b}, not ${2 * a + b}.`);
</script>
</head>
<body>

</body>
</html>

```

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

```

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>

```

4-3: 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 4-3-1.html

```

<html>
<head>
  <title>範例 4-2-1.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>
```

Module 5. 運算子

5-1: 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.

運算子	範例
指數運算子 (**)	2 ** 3 回傳 8.

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

5-2: 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

判斷身高與身高限制的比較

```
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
```

範例 5-2-1.html

```
<html>
<head>
  <title>範例 5-2-1.html</title>
  <script>
    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。

什麼？看不懂？再次說明如下：

- 如果 `input` 是原始資料類型，則直接回傳 `input`。
- 否則，如果 `input` 是個物件時，則呼叫物件的 `valueOf()` 方法，如果能得到原始資料類型的值，則回傳這個值。
- 否則，如果 `input` 是個物件時，呼叫物件的 `toString()` 方法，如果能得到原始資料類型的值，則回傳這個值。
- 否則，拋出 `TypeError` 錯誤。

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 5-2-2.html

```
<html>
<head>
  <title>範例 5-2-2.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

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 5-2-3.html

<html>

<head>

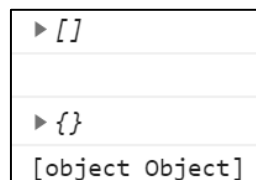
```
<title>範例 5-2-3.html</title>
<script>
console.log( [] + [] );
console.log( [] + {} );
console.log( {} + [] );
console.log( {} + {} );
</script>
</head>
<body>

</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]。

[object Object]
[object Object]
[object Object][object Object]

(圖) 牛刀小試的結果

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/>

5-3: 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 5-3-1.html

```
<html>
<head>
  <title>範例 5-3-1.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 5-3-2.html

```
<html>
<head>
  <title>範例 5-3-2.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，再去了解即可。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	... == ...
不等於		... != ...
嚴格等於		... === ...
嚴格不等於		... !== ...
位元 AND	left-to-right	... & ...
位元 XOR	left-to-right	... ^ ...
位元 OR	left-to-right
邏輯 AND	left-to-right	... && ...
邏輯 OR	left-to-right
條件式	right-to-left	... ? ... : ...
賦值	right-to-left	... = ...
		... += ...
		... -= ...
		... **= ...
		... *= ...
		... /= ...
		... %= ...
		... <<= ...

		... >>= ...
		... >>>= ...
		... &= ...
		... ^= ...
		... = ...
yield	right-to-left	yield ...
yield*		yield* ...
逗號 / 序列	left-to-right	... , ...

Module 6. String

6-1: 字串的標示方式

字串標式方式有：

- 透過兩個「'」將字串圍起來
- 透過兩個「"」將字串圍起來
- 透過 Template Literals 或 Template Strings 合併字串

範例 6-1-1.html
<pre> <html> <head> <title>範例 6-1-1.html</title> <script> //宣告字串 let strName = 'Darren'; //用單引號圍住字串的方式輸出 console.log('Hello World! ' + strName + '...'); //用雙引號圍住字串的方式輸出 console.log("I love javascript! " + strName + "..."); </pre>

```
//用 template strings 的方式輸出
console.log(`My name is ${strName} ...`);
</script>
</head>
<body>

</body>
</html>
```

補充說明

有關字串合併的方法，除了單/雙引號前面加號（+）來串接，還有提供樣版字面值（Template literals），讓字串合併與變數內嵌字串等問題，變得簡單許多。

使用單/雙引號來合併兩個句子，同時實現句子斷行

```
console.log('你好！我是' + '\n' + 'Darren Yang');
```

有的人會這麼做

```
console.log('你好！我是' +
'\n' +
'Darren Yang');
```

在上述的例子中，可以使用樣版字面值，即是使用「`」符號（一般鍵盤ESC 下面、數字 1 波浪符號的按鍵），放在字串前後，取代單/雙引號的位置。

```
console.log(`你好！我是\nDarren Yang`);
```

你也可以這麼使用

```
console.log(`你好！我是
Darren Yang`);
```

輸出的結果與上面幾個例子一樣！

還有一個很重要的用法，就是把變數放在樣版字面值當中來顯示

```
let fname = 'Darren';
let lname = 'Yang';
console.log(`你好! 我是 ${fname} ${lname}`);
//輸出 你好! 我是 Darren Yang

若是嵌入變數計算，可以這麼做
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b}, not ${2 * a + b}.`);
```

範例 4-1-4.html

```
<html>
<head>
  <title>範例 4-1-4.html</title>
  <script>
    //樣版字面值 (Template literals 或 Template strings)
    console.log(`你好! 我是\nDarrenYang`);
    console.log(`你好! 我是
Darren Yang`);

    //輸出 你好! 我是 Darren Yang
    let fname = 'Darren';
    let lname = 'Yang';
    console.log(`你好! 我是 ${fname} ${lname}`);

    //若是嵌入變數計算，可以這麼做
    let a = 5;
    let b = 10;
    console.log(`Fifteen is ${a + b}, not ${2 * a + b}.`);
  </script>
</head>
<body>

</body>
</html>
```

有關字串的進階運用，我們之後將在「正規表達式」或「正規表示法」的章節中詳細討論。

6-2: 字串的跳脫表示法

我們有時需要在字串中放置特殊字元，例如換行（\n）、向右縮排（\t）。除了常規的、可印出來的字元，特殊字元也可以被跳脫字元（符號）來表示編碼：

字元	意義
\b	後退一格
\f	換頁
\n	換行
\r	打字游標歸位
\t	Tab
\v	垂直 Tab
\'	單引號
\"	雙引號
\\	反斜線 (\)。
\XXX	使用介於 0 至 377 之間的三個八進制數 XXX 來表示以 Latin-1 編碼的字元。例如，\251 是版權符號的八進制內碼序列。
\xXX	使用介於 00 至 FF 之間的兩個十六進制數 XX 來表示以 Latin-1 編碼的字元。例如，\xA9 是版權符號的十六進制內碼序列。
\uXXXX	使用四個十六進制數 XXXX 來表示 Unicode 字元。例如，\u00A9 是版權符號的 Unicode 內碼序列。參閱 Unicode 的跳脫序列 。

範例 6-2-1.html

```
<html>
<head>
  <title>範例 6-2-1.html</title>
  <script>
    //為了避免字串當中有單引號
    console.log('Did you talk to Mary\'s brother?');

    //綜合使用
    console.log("When\tyou\nsay\nnothing\bat all.");
```



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

</body>
</html>
```

補充說明

雙引號、單引號所括括的字串，可以連同跳脫字元的效果與特性，一同輸出。以字串「When you say nothing at all.」為例。

```
console.log("When\\tyou\\nsay\\nnothing\\bat all.");
```

輸出結果為

```
When    you
say
nothinat all.
```

要注意的是，字串前後的引號，使用要一致，不能一個單、一個雙，只能單單、雙雙。

```
console.log('Hello World!'); // 正確輸出 Hello World!
```

```
console.log("Hello World!"); // 拋出 SyntaxError 語法錯誤的訊息
```

和其他語言不同，JavaScript 將單引號字串和雙引號字串是做相同；因此，上述的序列可以在單引號或雙引號中作用。

6-3: 字串的常用方法

性質或方法	說明
length	傳回字串的長度
big()	增大字串的字型
small()	減小字串的字型
bold()	變黑體
italics()	變斜體
fixed()	變等寬字體

strike()	槓掉字串
sub()	變下標
sup()	變上標
fontcolor()	設定字串的顏色
fontSize()	設定字串的字型大小
toUpperCase()	換成大寫字母
toLowerCase()	換成小寫字母
concat()	字串並排（等效於使用加號）
charAt(n)	抽出第 n 個字元（n=0 代表第一個字元）
charCodeAt(n)	抽出第 n 個字元（n=0 代表第一個字元），並轉換成 Unicode
substr(m, n)	傳回一個字串，從位置 m 開始，且長度為 n
substring(m, n)	傳回一個字串，從位置 m 開始，結束於位置 n-1
indexOf(str)	尋找子字串 str 在原字串的第一次出現位置
lastIndexOf(str)	尋找子字串 str 在原字串的最後一次出現位置
slice(start, end) slice(start)	傳回從 start 到 end（不包括）之間的字元；若是沒有指定 end，則會從 start 處開始到最後一個字元，一起傳回。
replace(substr, newSubStr) replace(regex, newSubStr)	替換 str 的子字串
search(regex)	查詢 str 與一個正規表示式是否匹配。如果匹配成功，則返回正規表示式在字串中 <u>首次匹配項的索引</u> ；否則，返回 -1。如果引數傳入的是一個非正規表示式物件，則會使用 new RegExp(obj) 隱式地將其轉換為正規表示式物件
match(regex)	返回一個包含匹配結果的 <u>陣列</u> ，如果沒有匹配項，則返回 null。如果引數傳入的是一個非正規表示式物件，則會使用 new RegExp(obj) 隱式地將其轉換為正規表示式物件
split(separator)	返回一個陣列， <u>分隔符 separator 可以是一個字串或正規表示式</u> 。
trim()	去除 str 開頭和結尾處的空白字元，返回 str 的一個副本，不影響字串本身的值

範例 6-3-1.html

```

<html>

<head>

  <title>範例 6-3-1.html</title>

  <script>

let myStr = "Tang Poem: 年年歲歲花相似，歲歲年年人不同。";

document.write("原字串：myStr = " + myStr + "<br>");

document.write("字串長度：myStr.length = " + myStr.length + "<br>");

document.write("增大字型：myStr.big() = " + myStr.big() + "<br>");

document.write("減小字型：myStr.small() = " + myStr.small() + "<br>");

document.write("變黑體：myStr.bold() = " + myStr.bold() + "<br>");

document.write("變斜體：myStr.italics() = " + myStr.italics() + "<br>");

document.write("變等寬字體：myStr.fixed() = " + myStr.fixed() + "<br>");

document.write("槓掉字串：myStr.strike() = " + myStr.strike() + "<br>");

document.write("變下標：myStr.sub() = " + myStr.sub() + "<br>");

document.write("變上標：myStr.sup() = " + myStr.sup() + "<br>");

document.write("設定顏色：myStr.fontcolor(\"salmon\") = " + myStr.fontcolor("salmon") + "<br>");

document.write("設定字型大小：myStr.fontSize(5) = " + myStr.fontSize(5) + "<br>");

  </script>

</head>

<body>


</body>

</html>

```

補充說明

檢查字串的長度

```

console.log("Hello World!".length);
// 輸出 12

```

在 ECMAScript 5 中被提到，可以從字串中取得單個字元（字串索引從 0 開始起算）

```

let myName = "Darren";
console.log(myName[0]); // 輸出 D
console.log(myName[2]); // 輸出 r
console.log(myName[5]); // 輸出 n
console.log(myName[6]); // 輸出 undefined

```

截取字串

```
"Darren Yang".slice(0,6);
```

// 0 是起始位置，6 是結束位置；位置 6 之前的字串才會被取得，故截取 0 到 5 位置的字串

D	a	r	r	e	n		Y	a	n	g
<u>0</u>	1	2	3	4	5	<u>6</u>	7	8	9	10

轉換所有字串為大寫

```
console.log( "Darren Yang".toUpperCase() );
```

// 輸出 DARREN YANG

轉換所有字串為小寫

```
console.log( "Darren Yang".toLowerCase() );
```

// 輸出 darren yang

你也可以透過變數來輸出大（小）寫結果

```
let id = 'a123456789';
```

```
console.log( id.toUpperCase() );
```

// 輸出 A123456789

範例 6-3-2.html

```
<html>
<head>
<title>範例 6-3-2.html</title>
<script>
let myStr = "Tang Poem: 年年歲歲花相似，歲歲年年人不同。";
document.write("原字串：myStr = " + myStr + "<br>");
document.write("換成大寫字母：myStr.toUpperCase() = " + myStr.toUpperCase() + "<br>");
document.write("換成小寫字母：myStr.toLowerCase() = " + myStr.toLowerCase() + "<br>");
document.write("字串並排：myStr.concat(\"新加的\") = " + myStr.concat("新加的") + "<br>");
document.write("抽出字元：myStr.charAt(13) = " + myStr.charAt(13) + "<br>");
document.write("抽出 Unicode 字元：myStr.charCodeAt(13) = " + myStr.charCodeAt(13) + "<br>");
document.write("抽出子字串：myStr.substr(13, 5) = " + myStr.substr(13, 5) + "<br>");
document.write("抽出子字串：myStr.substring(13, 15) = " + myStr.substring(13, 15) + "<br>");
document.write("尋找子字串：myStr.indexOf(\"年\") = " + myStr.indexOf("年") + "<br>");
document.write("尋找子字串：myStr.lastIndexOf(\"年\") = " + myStr.lastIndexOf("年") + "<br>");
document.write("抽出子字串：myStr.slice(13, 15) = " + myStr.slice(13, 15) + "<br>");
document.write("取代字串：myStr.replace('不同','相同') = " + myStr.replace('不同','相同') + "<br>");
document.write("搜尋字串：myStr.search(/歲歲/) = " + myStr.search(/歲歲/) + "<br>");
```

```

document.write("匹配字串：myStr.match(/歲歲/) = " + myStr.substr(/歲歲/) + "<br>");

console.log("切割字串：myStr.split('.') = ");
console.log(myStr.split('.'));

console.log("匹配字串：myStr.trim() = " + myStr.trim());

</script>
</head>
<body>

</body>
</html>

```

Module 7. 取得標籤元素

7-1: 使用 ES3 的方法

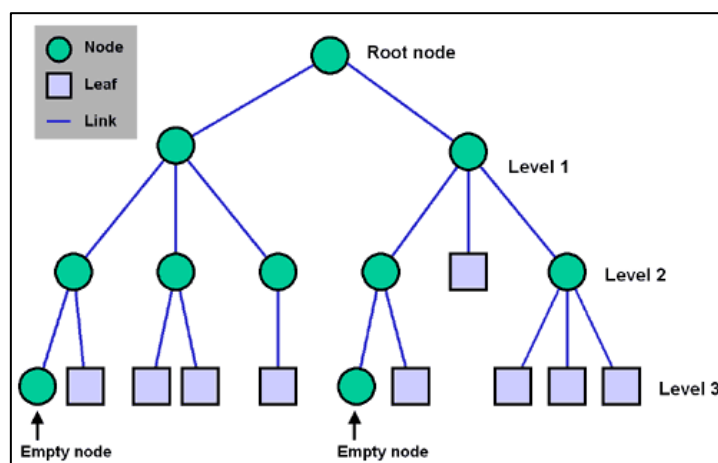
現今所有的瀏覽器都支援 ES3。JavaScript 核心的語言特性，由 ECMA-262 標準來定義，而該標準中定義的語言，又被稱為 ECMAScript，可以視為 JavaScript 的一種子集合（意思指 JavaScript 可以做得更多，例如在某些瀏覽器裡，可以實現多於 ECMAScript 定義的功能）。

版本	發表日期	與前版本差異
1	1997 年 6 月	首版。
2	1998 年 6 月	格式修正，以使得其形式與 ISO/IEC16262 國際標準一致。
3	1999 年 12 月	強大的正規表示式，更好的詞法作用域鏈處理，新的控制指令，例外處理，錯誤定義更加明確，資料輸出的格式化及其它改變。
4	放棄	由於關於語言的複雜性出現分歧，第 4 版本被放棄，其中的部分成為了第 5 版本及 Harmony 的基礎。
5	2009 年 12 月	新增「嚴格模式（strict mode）」，一個子集用作提供更徹底的錯誤檢查，以避免結構出錯。澄清了許多第 3 版本的模糊規範，並適應了與規範不一致的真實世界實現的行為。增加了部分新功能，如 getters 及

		setters，支援 JSON 以及在物件屬性上更完整的反射。
5.1	2011 年 6 月	ECMAScript 標 5.1 版形式上完全一致於國際標準 ISO/IEC 16262:2011。
6	2015 年 6 月	ECMAScript 2015 (ES2015)，第 6 版，最早被稱作是 ECMAScript 6 (ES6)，添加了類和模組的語法，其他特性包括疊代器，Python 風格的生成器和生成器表達式，箭頭函式，二進位資料，靜態型別陣列，集合 (maps, sets 和 weak maps)，promise，reflection 和 proxies。作為最早的 ECMAScript Harmony 版本，也被叫做 ES6 Harmony。
7	2016 年 6 月	ECMAScript 2016 (ES2016)，第 7 版，多個新的概念和語言特性。
8	2017 年 6 月	ECMAScript 2017 (ES2017)，第 8 版，多個新的概念和語言特性。
9	2018 年 6 月	ECMAScript 2018 (ES2018)，第 9 版，包含了非同步迴圈，生成器，新的正規表示式特性和 rest/spread 語法。
10	2019 年 6 月	ECMAScript 2019 (ES2019)，第 10 版

參考網址：<https://zh.wikipedia.org/wiki/ECMAScript>

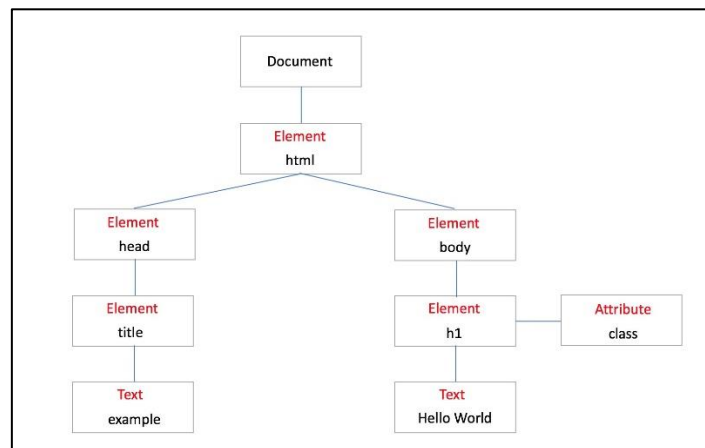
我們來解析一下 DOM。



(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：

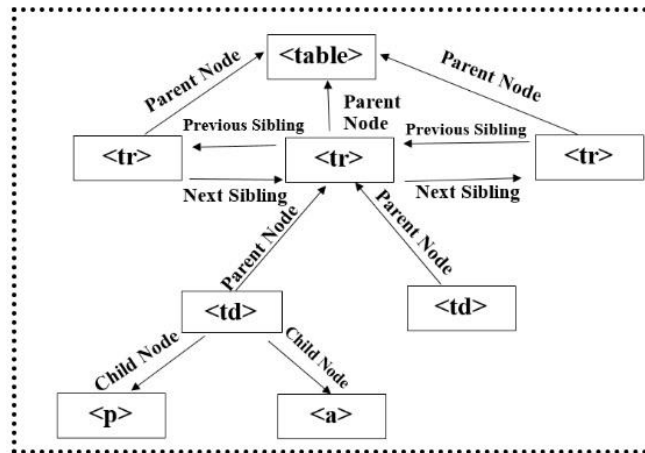
- Document
 - Document 就是指這份文件，也就是這份 HTML 檔的開端，所有的一切都會從 Document 開始往下進行。
- Element
 - Element 就是指文件內的各個標籤，因此像是 <div>、<p> 等等各種 HTML Tag 都是被歸類在 Element 裡面。
- Text
 - Text 就是指被各個標籤包起來的文字，舉例來說在 <h1>Hello World</h1> 中，Hello World 被 <h1> 這個 Element 包起來，因此 Hello World 就是此 Element 的 Text
- Attribute
 - Attribute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

7-2: querySelector()

用法：element = document.querySelector(selectors);

說明：在 document 中指定 CSS selectors 的第一個 element。

範例：var el = document.querySelector(".myclass");

範例 7-2-1.html

```
<html>
<head>
  <title>範例 7-2-1.html</title>
</head>
<body>
  <p class="myClass">問世間，</p>
  <p class="myClass">情是何物，</p>
  <p class="myClass">直教生死相許。</p>

  <script>
    //這裡是回傳第一個 p，是一種 element
    let p = document.querySelector('p');
    p.style.backgroundColor = '#7878ff';
  </script>
</body>
</html>
```

7-3: querySelectorAll()

用法：elementList = baseElement.querySelectorAll(selectors);

說明：在 document 中指定 CSS selectors 所有元素，回傳 NodeList 物件。

- NodeList 物件表示節點的集合，可以透過索引來取得元素資料。
- 可以透過 .length 來確認在 NodeList 中的元素數量。

範例：`let matches = document.body.querySelectorAll('p');`

範例 7-3-1.html

```
<html>
<head>
  <title>範例 7-3-1.html</title>
</head>
<body>
  <p class="myClass">問世間，</p>
  <p class="myClass">情是何物，</p>
  <p class="myClass">直教生死相許。</p>

  <script>
    //回傳的是 NodeList
    let p = document.querySelectorAll('p');

    //可以使用類似陣列的操作方式，提供索引值來存取
    p[0].style.backgroundColor = '#ff0000';
    p[1].innerHTML = '<span style="color: #00ff00">...情是何物...</span>';
    p[2].style.lineHeight = '24px';
    p[2].style.backgroundColor = '#0000ff';
    p[2].style.color = '#ffffff';
  </script>
</body>
</html>
```

Module 8. 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。

8-1: 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 8-1-1.html

```
<html>  
<head>  
  <title>範例 8-1-1.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 8-1-2.html

```
<html>  
<head>  
  <title>範例 8-1-2.html</title>  
  <script>  
    let number = 7;  
    if(number > 10) {  
      console.log(1);  
    }  
  </script>  
</head>  
<body>  
  
</body>  
</html>
```

```
    } 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 8-1-3.html
<pre><html> <head> <title>範例 8-1-3.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></pre>

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 8-1-4.html

```
<html>
<head>
  <title>範例 8-1-4.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 8-1-5.html

```
<html>
<head>
  <title>範例 8-1-5.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>
```

8-2: 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 8-2-1.html

```
<html>  
<head>  
    <title>範例 8-2-1.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 8-2-2.html

```
<html>  
<head>  
    <title>範例 8-2-2.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
<p>for 迴圈起手式</p> <pre>for(setup; condition; increment) { //statements }</pre> <p>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</p> <pre>for(let i = 0; i < 10; i++) { console.log('The value is ' + i); }</pre> <p>// 輸出 The value is 0 The value is 9</p> <p>迴圈內部也可以使用迴圈，通稱為「巢狀迴圈」</p> <pre>for(let i = 1; i <= 9; i++) { for(let j = 1; j <= 9; j++){ console.log(i + ' * ' + j + ' = ' + (i*j)); } }</pre> <p>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。</p>

for 的無限迴圈形式：

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

範例 8-2-3.html

```
<html>  
<head>  
    <title>範例 8-2-3.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 8-2-4.html

```
<html>  
<head>  
    <title>範例 8-2-4.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

8-3: 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 8-3-1.html

```
<html>
<head>
  <title>範例 8-3-1.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 8-3-2.html

```

<html>
<head>
  <title>範例 8-3-2.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 8-3-3.html

```

<html>
<head>
  <title>範例 8-3-3.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>
```

範例 8-3-4.html

```
<html>
<head>
  <title>範例 8-3-4.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 9. Object 類型

9-1: 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();
```

9-2: 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 9-2-1.html

```
<html>
<head>
  <title>範例 9-2-1.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 9-2-2.html

```
<html>
<head>
  <title>範例 9-2-2.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
```

9-3: 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 9-3-1.html

```
<html>
<head>
  <title>範例 9-3-1.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 10. Array 類型

10-1: 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
<p>陣列的宣告：</p> <pre>let arr = [];</pre> <p>也有人這麼寫</p> <pre>let arr = new Array();</pre>

10-2: Array 表達式

範例					
<p>宣告陣列時，建立初始值：</p> <pre>let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];</pre> <p>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：</p> <pre>let listOfName = ['Alex', 'Bill', 'Cook', 'Darren'];</pre> <p>很重要的觀念是，陣列中每一個值的索引（index），都是從「0」開始。</p> <table><tr><td>索引</td><td>0</td><td>1</td><td>2</td><td>3</td></tr></table>	索引	0	1	2	3
索引	0	1	2	3	

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

10-3: 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 10-3-1.html

```
<html>
<head>
  <title>範例 10-3-1.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 10-3-2.html

```
<html>
<head>
  <title>範例 10-3-2.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 10-3-3.html

```
<html>
<head>
  <title>範例 10-3-3.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 10-3-4.html

```

<html>
<head>
  <title>範例 10-3-4.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 10-3-5.html

```

<html>
<head>
  <title>範例 10-3-5.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 10-3-6.html

```

<html>
<head>
  <title>範例 10-3-6.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 10-3-7.html

```
<html>  
<head>  
  <title>範例 10-3-7.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 10-3-8.html

```
<html>  
<head>
```

```

<title>範例 10-3-8.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);
}

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

```

補充說明：

- 陣列、字串具有「序列化」的特性，例如陣列 `arr = ["Alex", "Bill"]`；可透過 `arr[0]` 來取得 "Alex"；字串 `str = "人生好難"`；可透過 `str[2]` 來得取「好」。

- 針對這種序列化的結構，若是要搜尋當中是否有相等的值，可以使用「.indexOf()」，來與序列化資料當中的值進行比較，若有相等的值，則回傳該值位於序列化結構的「索引」（0 到 n-1，找不到的話，回傳 -1）。

範例 10-3-9.html

```
<!DOCTYPE html>
<html>
<head>
  <title>範例 10-3-9.html</title>
  <script>
    //宣告一個陣列
    let arr = ["Alex", "Bill"];

    //宣告一個字串
    let str = "人生好難";

    //判斷 Bill 是否存在於 arr 當中
    if( arr.indexOf("Bill") !== -1 ){
      alert(`有找到 Bill`);
    } else {
      alert(`沒找到 Bill ...`);
    }

    //判斷「好」是否存在於 str 當中
    if( str.indexOf("好") !== -1 ){
      alert(`有找到「好」`);
    } else {
      alert(`沒找到「好」 ...`);
    }
  </script>
</head>
<body>

</body>
</html>
```

Module 11. JSON

JSON (JavaScript Object Notation, JavaScript 物件表示法, 讀作/¹dʒeɪ sən/), 是一種輕量級的資料交換語言, 該語言以易於讓人閱讀的文字為基礎, 用來傳輸由屬性、值組成的資料物件。JSON 資料格式與語言無關。即便它源自 JavaScript, 但目前很多程式語言都支援 JSON 格式資料的生成和解析。JSON 的官方 MIME 類型是 application/json, 副檔名是 .json。

大致要注意的是 JSON：

- 名稱為字串, 必須用"雙引號"包起來。
- 值可以是"雙引號"包括的字串, 或者是數字、true、false、null、物件或陣列。
- 不支援 JavaScript 的 Data、Error、正規表達式或函式表示。

11-1: JSON 字串規則

範例

它的呈現方式類似 JavaScript 裡面的 Object, 基本結構與概念都差不多, 只是它以字串的格式儲存和運用, 屬性也是以字串方式定義。

簡單結構

```
'{"name": "bill", "age": 25}'
```

json 轉物件的結構

```
{  
  "name": "bill",  
  "age": 25  
}
```

json 結構

```
'{"firstName": "Darren", "lastName": "Yang", "gender": "male", "age": 18, "address": {"streetAddress": "台北市復興南路一段 390 號 2 樓、3 樓", "city": "Taipei", "phoneNumber": [{"type": "office", "number": "6631-6666"}, {"type": "fax", "number": "6631-6598"}]}'
```

json 轉物件的結構

```

{
  "firstName": "Darren",
  "lastName": "Yang",
  "gender": "male",
  "age": 18,
  "address": {
    "streetAddress": "台北市復興南路一段 390 號 2 樓、3 樓",
    "city": "Taipei",
  },
  "phoneNumber": [
    {
      "type": "office",
      "number": "6631-6666"
    },
    {
      "type": "fax",
      "number": "6631-6598"
    }
  ]
}

```

範例 11-1-1.html

```

<html>
<head>
  <title>範例 11-1-1.html</title>
  <script>
    //輸出 json 內容
    let str = '{"firstName": "Darren", "lastName": "Yang", "gender": "male", "age": 18, "address": {"streetAddress": "台北市復興南路一段 390 號 2 樓、3 樓", "city": "Taipei"}, "phoneNumber": [{"type": "office", "number": "6631-6666"}, {"type": "fax", "number": "6631-6598"}]}';
    console.log(str);
  </script>
</head>
<body>

</body>
</html>

```

11-2: Object 和 Array 的複製

Array 的複製

範例 11-2-1.html

```
<html>
<head>
  <title>範例 11-2-1.html</title>
  <script>
    //用 for 迴圈實現陣列的複製
    let arr01 = [1,2,3,4,5];
    let arr02 = [];
    for(let i = 0; i < arr01.length; i++){
      arr02.push(arr01[i]);
    }
    console.log(arr01);
    console.log(arr02);

    //用 slice 方法實現陣列的複製
    let arr03 = [9,4,8,7];
    let arr04 = arr03.slice(0); //從索引 0 開始取到最後一個元素，一次回傳
    console.log(arr03);
    console.log(arr04);

    //用 concat 方法實現陣列的複製
    let arr05 = [6,6,6];
    let arr06 = arr05.concat(); //結合空的東西，代表依然保留原先陣列內容
    console.log(arr05);
    console.log(arr06);

    //用 ... 實現陣列的複製
    let arr07 = [9,8,7,6,5];
    let [...arr08] = arr07;
    console.log(arr07);
    console.log(arr08);
  </script>
</head>
```

```
<body>
```

```
</body>
```

```
</html>
```

物件的複製

說明

將物件轉成 json

```
let str = JSON.stringify( object );
```

將 json 轉成物件 (將 json 字串轉換成 JavaScript 的原生類型)

```
let obj = JSON.parse(str);
```

範例 11-2-2.html

```
<html>
```

```
<head>
```

```
  <title>範例 11-2-2.html</title>
```

```
  <script>
```

```
    //用 for 迴圈實現物件的複製
```

```
    let obj01 = {name: "Bill", age: 25};
```

```
    let obj02 = {};
```

```
    for(let key in obj01){
```

```
      obj02[key] = obj01[key];
```

```
    }
```

```
    console.log(obj01);
```

```
    console.log(obj02);
```

```
    //轉成 json 後，再轉成 object，來實現物件的複製
```

```
    let obj03 = {name: "Alex", age: 18};
```

```
    let strJson = JSON.stringify(obj03);
```

```
    let obj04 = JSON.parse(strJson);
```

```
    console.log(obj03);
```

```
    delete obj04.name;
```

```
    console.log(obj04);
```

```
  /**
```

```
    * 上面的流程等同於
```

```
    * let obj03 = {name: "Alex", age: 18};
```

```

    * let obj04 = JSON.parse( JSON.stringify(obj03) );
    */

    //用 ... 實現物件的複製
    let obj05 = { name: "Cook", age: 22 };
    let {...obj06} = obj05;
    console.log(obj05);
    console.log(obj06);
  </script>
</head>
<body>

</body>
</html>

```

11-3: 編輯 JSON 檔

我們可以透過物件屬性的新增、修改、刪除：

範例 11-3-1.html

```

<html>
<head>
  <title>範例 11-3-1.html</title>
  <script>
    //物件初始化
    let objPerson = {
      name: "Bill",
      age: 25,
      hairColor: 'black',
      skinColor: 'beige'
    };

    //新增屬性
    objPerson.hadShoes = true;
    objPerson.isWoken = true;

    //修改屬性
    objPerson.age = 26;
  </script>

```



```
//暫時顯示物件內容
console.log(objPerson);

//刪除屬性
delete objPerson.skinColor;

//顯示物件內容
console.log(objPerson);
</script>
</head>
<body>

</body>
</html>
```

透過上述的操作，可以得知 Object、Array、String 等，都可以轉成 JSON，但是 HTML element 不行。

Module 12. 函式的定義

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）。

12-1: 基本型函式

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 12-1-1.html

```
<html>  
<head>  
    <title>範例 12-1-1.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!';
}

//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 ($10 + 9 + 8 + \dots + 2 + 1$)：

```
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。

12-2: 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 12-2-1.html

```
<html>  
<head>  
    <title>範例 12-2-1.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);  
        }  
    </script>  
</head>  
</html>
```

```
//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 12-2-2.html

```
<html>
<head>
  <title>範例 12-2-2.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>
```

12-3: 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 12-3-1.html

```
<html>
<head>
  <title>範例 12-3-1.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，若是在區塊外嘗試輸出，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 var 來宣告 value	使用 let 來宣告 value
<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 else {...} 也能存取 value</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 value 在 else {...} //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; } swap(a, b); console.log(a, b); // 輸出 10, 20</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' } function setName(objX){ objX.name = 'Alex'; //類似傳址呼叫 // objX = {name: 'Doris'} //類似傳值呼叫</pre>

	<pre> } setName(objA); console.log(objA, objB); // 輸出 { name: 'Alex' } { name: 'Alex' } 若是把 setName 當中的兩行程式註解互換， objA 跟 objB 還各是什麼？ </pre>
<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

Module 13. Scope 變數領域

13-1: 全域變數

Global Variable 全域變數：

能夠在函式內或函式外宣告（可以想成所有 `{...}` 結構外所宣告的變數），可於整個網頁範圍內調用，所以整個網頁中雖然可以有無數個不同名稱的 Global Variable，但僅會有一個獨立的 Global Variable 名稱，若重覆名稱則會覆蓋變數值，網頁關閉時，Global Variable 亦失效。

範例 13-1-1.html

```
<html>
<head>
  <title>範例 13-1-1.html</title>
  <script>
    //定義一個函式
    function test(){
      //宣告全域變數
      var01 = 'A';

      document.write('var01 = ' + var01 + '<br>'); // 輸出 A
    }

    //在 function 外定義變數
    var var02 = 'C';

    //執行函式
    test();

    //在 function 外輸出變數的值
    document.write('var01 = ' + var01 + '<br>'); // 輸出 A
    document.write('var02 = ' + var02 + '<br>'); // 輸出 B
  </script>
</head>
<body>
```

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

範例 13-1-2.html

```
<html>
<head>
  <title>範例 13-1-2.html</title>
  <script>
    //定義一個函式
    function test(){
      //宣告一個全域變數
      var01 = 'A';

      document.write('var01 = ' + var01 + '<br>'); // 輸出 A
    }

    //設定 var01 的值
    function setVar(){
      var01 = 'B';
    }

    //執行函式
    test();
    setVar();

    //在 function 外輸出變數的值
    document.write('var01 = ' + var01 + '<br>'); // 輸出 B
  </script>
</head>
<body>

</body>
</html>
```

13-2: 區域變數

Local Variable 區域變數：

僅能夠在函式中透過關鍵字 `var` 宣告 (可以想成在 `{...}` 區塊內使用 `var` 宣告的變數)，每個不同的函式可以有相同的 Local Variable 變數名稱，換句話說，每個函式間的 Local Variable 互不干涉，也無法在函式外其他地方調用，當函式結束工作後，Local Variable 亦失效。

範例 13-2-1.html

```
<html>
<head>
  <title>範例 13-2-1.html</title>
  <script>
    //定義一個函式
    function test(){
      //宣告區域變數
      var var01 = 'A';

      //宣告全域變數
      var02 = 'B';

      document.write('var01 = ' + var01 + '<br>'); // 輸出 A
      document.write('var02 = ' + var02 + '<hr>'); // 輸出 B
    }

    //在 function 外定義變數
    var var03 = 'C';

    //執行函式
    test();

    //document.write('var01 = ' + var01 + '<br>'); // 拋出錯誤
    document.write('var02 = ' + var02 + '<br>'); // 輸出 B
    document.write('var03 = ' + var03 + '<hr>'); // 輸出 C
  </script>
</head>
<body>

</body>
</html>
```

13-3: closure

通常定義了一個函式之後，在 functionA 作用域（{...}）的內部區域中，又定義了一個內部的函式，這種透過作用域的特性，來存取內外部變數，稱之為閉包（closure）。

13-3-1.html

```
<html>
<head>
  <title>範例 13-3-1.html</title>
  <script>
    //定義函式
    function makeFunc(){
      //宣告變數
      var strName = 'Darren';

      //內部函式
      function displayName(){
        console.log(strName);
      }

      //執行內部函式
      displayName();
    }

    //執行函式
    makeFunc();
  </script>
</head>
<body>

</body>
</html>
```

在 closure 內，可以取用全域變數：

範例 13-3-2.html

```
<html>
```

```
<head>
  <title>範例 13-3-2.html</title>
  <script>
    //宣告全域變數
    strNum = 9487;

    //定義函式
    function makeFunc(){
      //宣告變數
      var strName = 'Bill';

      //內部函式(使用箭頭函式)
      let displayName = () => {
        //存取全域變數
        console.log(strName + ' 說: ' + strNum);
      }

      //執行內部函式
      displayName();
    }

    //執行函式
    makeFunc();
  </script>
</head>
<body>

</body>
</html>
```

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;
    };
  </script>
</body>
</html>
```

```
//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. 時間與計時器

15-1: Date 物件

在 Javascript 的互動應用中，經常使用「Date 物件」當中的類時間函式，來判斷、比較、計算時間的差距，或是使用特定的時間格式，例如 timestamp、ISO 8601 等。以下提供幾個範例：

範例

我們可以直接使用「new Date()」來使用時間相關函式，並取得相對應的結果；以下請用 console.log() 輸出。

宣告 Date 物件：

```
let date = new Date();
```

今天星期幾：

`date.getDay();` // 0，指星期天；數值範例是 0 - 6

你也可以直接使用時間物件，例如看看今天星期幾：

`new Date().getDay();` // 0，指星期天；數值範例是 0 - 6

其它範例：

`date.getFullYear()` // 2019，西元年

`date.getMonth()` // 月份，6 月時執行會出現 5，比正常值少 1，記得加 1

`date.getDate()` // 26，當月 26 號

`date.getHours();` // 幾時，例如 14 時

`date.getMinutes();` // 幾分，例如 58 分

`date.getSeconds();` // 幾秒，例如 53 秒

還有許多應用，可查閱 MDN web docs。

[https://developer.mozilla.org/en-](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)

[US/docs/Web/JavaScript/Reference/Global_Objects/Date](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)

補充說明

判斷今天星期幾（0 為星期日，1 為星期一，2 為星期二，...，6 為星期六）

```
let day;
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;

  case 1:
    day = "Monday";
    break;

  case 2:
    day = "Tuesday";
    break;

  case 3:
    day = "Wednesday";
    break;
```

```

    case 4:
        day = "Thursday";
        break;

    case 5:
        day = "Friday";
        break;

    case 6:
        day = "Saturday";
        break;

    default:
        day = "What?!";
}
console.log(day);

```

補充說明

若是時間物件的函式（方法）回傳，只有一位數的結果（例如六月只回傳 6，而非 06），我們可以用一些小技巧來輸出自訂的格式。

```

let month = (new Date().getMonth() + 1); // getMonth() 範圍 0 到 11，所以要加 1

if( month < 10 ){ // 數值小於 10，則左側加個 '0'
    month = '0' + month;
}

console.log(month); // 輸出 06

```

範例 15-1-1.html

```

<html>
<head>
    <title>範例 15-1-1.html</title>
    <script>
        //宣告 Date 物件：
        let date = new Date();
    </script>

```

```

//字串變數，預設空值
let strTime = "";

//串接時間
strTime += date.getFullYear() // 2020，西元年
strTime += date.getMonth() + 1 // 月份，6月時執行會出現5，比正常值少1，記得加1
strTime += date.getDate() // 26，當月 26 號
strTime += date.getHours(); // 幾時，例如 14 時
strTime += date.getMinutes(); // 幾分，例如 58 分
strTime += date.getSeconds(); //幾秒，例如 53 秒

//輸出結果
console.log(strTime);
</script>
</head>
<body>

</body>
</html>

```

顯示當前的「西元年-月-日 時:分:秒」。(例如：2020-03-31 11:22:33)

參考作法 (?: 三元條件運算子版本)

```

let date = new Date();

let year, month, day, hour, minute, second;

// 2019，西元年
year = date.getFullYear()

// 月份，10 月時執行會出現 9，比正常值少 1，記得加 1
month = ( ( date.getMonth() + 1 ) < 10 ) ? '0' + (date.getMonth() + 1) : (date.getMonth() + 1);

// 25，當月 25 號
day = ( date.getDate() < 10 ) ? '0' + date.getDate() : date.getDate();

// 幾時，例如 14 時
hour = ( date.getHours() < 10 ) ? '0' + date.getHours() : date.getHours();

```

```
// 幾分，例如 58 分
minute = ( date.getMinutes() < 10 ) ? '0' + date.getMinutes() : date.getMinutes();

//幾秒，例如 53 秒
second = ( date.getSeconds() < 10 ) ? '0' + date.getSeconds() : date.getSeconds();

console.log(` ${year}-${month}-${day} ${hour}:${minute}:${second} `);
```

參考作法 (if else 版本)

```
let date = new Date();

let year, month, day, hour, minute, second;

// 2019，西元年
year = date.getFullYear()

// 月份，10 月時執行會出現 9，比正常值少 1，記得加 1
if( (date.getMonth() + 1) < 10 ) {
    month = '0' + (date.getMonth() + 1);
} else {
    month = (date.getMonth() + 1);
};

// 25，當月 25 號
if( date.getDate() < 10 ) {
    day = '0' + date.getDate();
} else {
    day = date.getDate();
};

// 幾時，例如 14 時
if( date.getHours() < 10 ) {
    hour = '0' + date.getHours();
} else {
    hour = date.getHours();
};
```

```

// 幾分，例如 58 分
if( date.getMinutes() < 10) {
    minute = '0' + date.getMinutes();
} else {
    minute = date.getMinutes();
};

//幾秒，例如 53 秒
if( date.getSeconds() < 10) {
    second = '0' + date.getSeconds();
} else {
    second = date.getSeconds();
};

console.log(` ${year}-${month}-${day} ${hour}:${minute}:${second} `);

```

範例 15-1-2.html

```

<html>
<head>
  <title>範例 15-1-2.html</title>
  <script>
    /**
     * if else 版本
     */

    //宣告 Date 物件
    let date = new Date();

    //宣告變數
    let year, month, day, hour, minute, second;

    // 2020，西元年
    year = date.getFullYear()

    // 月份，10 月時執行會出現 9，比正常值少 1，記得加 1
    if( (date.getMonth() + 1) < 10 ) {
        month = '0' + (date.getMonth() + 1);
    }
  </script>

```

```
} else {
    month = (date.getMonth() + 1);
};

// 25，當月 25 號
if( date.getDate() < 10 ) {
    day = '0' + date.getDate();
} else {
    day = date.getDate();
};

// 幾時，例如 14 時
if( date.getHours() < 10 ) {
    hour = '0' + date.getHours();
} else {
    hour = date.getHours();
};

// 幾分，例如 58 分
if( date.getMinutes() < 10 ) {
    minute = '0' + date.getMinutes();
} else {
    minute = date.getMinutes();
};

//幾秒，例如 53 秒
if( date.getSeconds() < 10 ) {
    second = '0' + date.getSeconds();
} else {
    second = date.getSeconds();
};

//輸出結果
console.log(`${year}-${month}-${day} ${hour}:${minute}:${second}`);
</script>
</head>
<body>
```

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

15-2: setTimeout 用法

用於在指定的毫秒數後，執行 1 次函式，後方參數會作為函式參數使用：

```
setTimeout(函式, 毫秒數[, 參數 1, 參數 2, ... 參數 N]);
```

```
let instanceId = setTimeout(函式, 毫秒數[, 參數 1, 參數 2, ... 參數 N]);
```

例如 `setTimeout(function () { alert('Hello World'); }, 3000);`

也可以事先定義函式後，再放到 `setTimeout()` 當中作為參數：

範例 15-2-1.html

```
<html>
<head>
  <title>範例 15-2-1.html</title>
  <script>
    //三秒後跳出訊息 Hello World
    setTimeout(function () { alert('Hello World'); }, 3000 );

    //自訂函式後，使用 setTimeout 來調用自訂函式
    function getMessage(){
      alert('Good job!');
    }

    //十秒後跳出訊息 Good job!
    setTimeout(getMessage, 10000);
  </script>
</head>
<body>

</body>
</html>
```

範例 15-2-2.html

```
<html>
```



```

<head>
  <title>範例 15-2-2.html</title>
</head>
<body>
  <p id="txt">時間開始</p>

  <script>
    /* 每 2 秒更新一次 p 元素當中的內文 */

    //取得 p 元素
    var p = document.getElementById("txt");

    //可以對 setTimeout 裡面的 function block 內容進行排版
    setTimeout(function(){
      p.innerText = "2 秒";
    }, 2000);

    setTimeout(function(){ p.innerText = "4 秒" }, 4000);
    setTimeout(function(){ p.innerText = "6 秒" }, 6000);
  </script>
</body>
</html>

```

範例 15-2-3.html

```

<html>
<head>
  <title>範例 15-2-3.html</title>
  <script>
    var myWindow = window.open(
      "https://www.google.com/",
      "_blank",
      "width=800, height=600"
    );
    document.write("<p>開新視窗</p>");
    setTimeout(function(){ myWindow.close() }, 3000);
  </script>
</head>
<body>

```

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

範例 15-2-4.html

```
<html>
<head>
  <title>範例 15-2-4.html</title>
  <script>
    //宣告變數
    var myVar;

    //定義函式
    function myStartFunction() {
      myVar = setTimeout(alertFunc, 3000, "Runoob", "Google");
    }

    //此函式用在 myStartFunction 當中
    function alertFunc(param1, param2){
      alert( 你的參數: ${param1}, ${param2} );
    }

    //執行函式
    myStartFunction();
  </script>
</head>
<body>

</body>
</html>
```

clearTimeout() - 停止 setTimeout() 執行的函式：

我們可以使用 clearTimeout(透過 setTimeout() 建立的變數) 來結束 setTimeout() 的運作。

範例 15-2-5.html

```
<html>
<head>
  <title>範例 14-2-5.html</title>
```

```
</head>
<body>
  <button onclick="startCount()">開始累加</button>
  <input type="text" id="txt" value="">
  <button onclick="stopCount()">結束累加</button>

  <script>
    //累加數字的變數
    var countNum = 0;

    //作為 clearTimeout() 用的變數
    var t;

    //累加用的函式
    function count() {
      //將當前的 count 值，放到 id=txt 的元素 value 屬性當中
      document.getElementById("txt").value = countNum;

      //進行數字的累加
      countNum = countNum + 1;

      //運用遞迴的概念，在指定毫秒後進行累加
      t = setTimeout(function(){ count() }, 1000);
    }

    //開始累加
    function startCount() {
      //執行累加用的函式
      count();
    }

    //結束累加
    function stopCount() {
      clearTimeout(t);
    }
  </script>
</body>
</html>
```

15-3: setInterval 用法

與 `setTimeout` 不同（僅執行 1 次），`setInterval` 方法可以週期性地（一樣指定毫秒）執行自訂的函式。

執行 `setInterval()`

```
setInterval(function(){ alert("Hello"); }, 3000);
```

執行 `setInterval()`，並返回實體 `id`

```
let instanceId = setInterval(function(){ alert("Hello"); }, 3000);
```

停止 `setInterval`

```
clearInterval(myVar);
```

範例 15-3-1.html

```
<html>
<head>
  <title>範例 15-3-1.html</title>
</head>
<body>
  <button onclick="startCount()" id="btn_start">開始累加</button>
  <input type="text" id="txt" value="">
  <button onclick="stopCount()" id="btn_end">結束累加</button>

  <script>
    //累加數字的變數
    var countNum = 0;

    //作為 clearInterval() 用的變數
    var t;

    //累加用的函式
    function count() {
      //將當前的 count 值，放到 id=txt 的元素 value 屬性當中
      document.getElementById("txt").value = countNum;
```

```

    //進行數字的累加
    countNum = countNum + 1;
}

//開始累加
function startCount() {
    //週期性在指定毫秒後，進行累加
    t = setInterval(function(){ count() }, 1000);

    //將開始按鈕設定為不可用
    document.getElementById("btn_start").setAttribute("disabled", true);

    //將結束鈕按移除不可用的屬性
    document.getElementById("btn_end").removeAttribute("disabled");
}

//結束累加
function stopCount() {
    //結束 setInterval
    clearInterval(t);

    //將開始鈕按移除不可用的屬性
    document.getElementById("btn_start").removeAttribute("disabled");

    //將結束按鈕設定為不可用
    document.getElementById("btn_end").setAttribute("disabled", true);
}
</script>
</body>
</html>

```

Module 16. 數學物件

Math 是一個擁有數學常數及數學函數（非函式物件）屬性及方法的內建物件。不像其他的全域物件，Math 並非建構函式。所有 Math 的屬性及方法皆為靜態。你可以使用 Math.PI 來參考到圓周率 pi 的常數值，以及可以呼叫

Math.sin(x) 函式來計算三角函數正弦曲線 sine (x 為方法的引數)。常數是由 JavaScript 中實數的完整精度來定義。

以下資料參考 MDN web docs - Math

https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Reference/Global_Objects/Math

屬性	說明
Math.E	歐拉常數 (此指自然常數)，也是自然對數的底數，約為 2.718。
Math.LN2	2 的自然對數，約為 0.693。
Math.LN10	10 的自然對數，約為 2.303。
Math.LOG2E	以 2 為底的 E 的對數，約為 1.443。
Math.LOG10E	以 10 為底的 E 的對數，約為 0.434。
Math.PI	一個圓的圓周和其直徑比值，約為 3.14159。
Math.SQRT1_2	1/2 的平方根；也就是 1 除以 2 的平方根，約為 0.707。
Math.SQRT2	2 的平方根，約為 1.414。

方法	說明
Math.abs(x)	回傳 x 的絕對值。
Math.acos(x)	回傳 x 的反餘弦值。
Math.acosh(x)	Returns the hyperbolic arccosine of a number.
Math.asin(x)	回傳 x 的反正弦值。
Math.asinh(x)	Returns the hyperbolic arcsine of a number.
Math.atan(x)	回傳 x 的反正切值。
Math.atanh(x)	Returns the hyperbolic arctangent of a number.
Math.atan2(y, x)	Returns the arctangent of the quotient of its arguments.
Math.cbrt(x)	回傳 x 的立方根值。
Math.ceil(x)	回傳不小於 x 的最小整數值。
Math.clz32(x)	Returns the number of leading zeroes of a 32-bit integer.
Math.cos(x)	回傳 x 的餘弦值。
Math.cosh(x)	Returns the hyperbolic cosine of a number.
Math.exp(x)	回傳 E^x ，x 為給定數值，E 為歐拉常數 (自然對數的底數)。
Math.expm1(x)	回傳 $\exp(x)$ 減去 1 的值。

Math.floor(x)	回傳不大於 x 的最大整數值。
Math.fround(x)	Returns the nearest single precision float representation of a number.
Math.hypot([x[, y[, ...]]])	回傳參數平方之和的平方根。
Math.imul(x, y)	Returns the result of a 32-bit integer multiplication.
Math.log(x)	回傳 x 的自然對數值。
Math.log1p(x)	回傳 1 + x 的自然對數值。
Math.log10(x)	回傳以 10 為底，x 的對數值。
Math.log2(x)	回傳以 2 為底，x 的對數值。
Math.max([x[, y[, ...]]])	回傳給定數值中的最大值。
Math.min([x[, y[, ...]]])	回傳給定數值中的最小值。
Math.pow(x, y)	回傳 x 的 y 次方，也就是 xy 。
Math.random()	回傳一個 0 到 1 之間的偽隨機值。
Math.round(x)	回傳 x 的四捨五入值。
Math.sign(x)	回傳 x 的正負號，也就是回傳 x 的正負。
Math.sin(x)	回傳 x 的正弦值。
Math.sinh(x)	Returns the hyperbolic sine of a number.
Math.sqrt(x)	回傳 x 的正平方根。
Math.tan(x)	回傳 x 的正切值。
Math.tanh(x)	Returns the hyperbolic tangent of a number.
Math.toSource()	回傳字串 "Math"。
Math.trunc(x)	Returns the integral part of the number x, removing any fractional digits.

16-1: 亂數

回傳於 0 (包含) ~ 1 (不包含) 之間的隨機數 (浮點數)，只是通常不太可能產生剛好為 1 的值。

範例 16-1-1.html
<pre> <html> <head> <title>範例 15-1-1.html</title> <script> //隨機產生 0 (包含) 到 1 (不包含) 之間的隨機數 console.log(Math.random()); // 輸出 0.02529304315381542 console.log(Math.random()); // 輸出 0.8048744968811596 </pre>

```
console.log( Math.random() ); // 輸出 0.3708412087199269
console.log( Math.random() ); // 輸出 0.9822622936344187

</script>
</head>
<body>

</body>
</html>
```

補充說明
<p>取整數（取得小於等於 x 的最大整數）：</p> <p>Math.floor(x);</p> <p>例如</p> <p>console.log(Math.floor(3.1415926)); // 輸出 3</p> <p>建立隨機索引：</p> <p>let index = Math.floor(Math.random() * 4);</p> <p>說明：</p> <p>因為 Math.random() 產生 0 - 1 之間的浮點數，0.01 乘上 4 的整數為 0，0.9 乘上 4 的整數為 3，所以可以確定建立的索引範例，會在 0 - 3 之間（[0],[1],[2],[3]，共四個索引）。</p> <p>隨機選擇餐廳</p> <p>let arrMeal = ['麥當勞', '肯德基', '摩斯漢堡', '頂呱呱', '漢堡王'];</p> <p>let idxRandom = Math.floor(Math.random() * 5);</p> <p>console.log(`今天中餐，我選擇 \${arrMeal[idxRandom]}`);</p> <p>// 輸出 今天中餐，我選擇 麥當勞</p> <p>取得介於 1 到 100 之間的隨機數</p> <p>Math.floor((Math.random() * 100) + 1);</p>

範例 16-1-2.html
<html>


```

<head>
  <title>範例 16-1-2.html</title>
  <script>
    //隨機選擇餐廳
    let arrMeal = ['麥當勞', '肯德基', '摩斯漢堡', '頂呱呱', '漢堡王'];
    let idxRandom = Math.floor( Math.random() * 5 );
    document.write(`我可以選擇的餐廳有: <br />`);

    for(let i = 0; i < arrMeal.length; i++){
      document.write('● ' + arrMeal[i] + '<br />');
    }

    document.write(`今天中餐，我選擇 ${arrMeal[idxRandom]}`);
  </script>
</head>
<body>

</body>
</html>

```

16-2: 三角函數

方法	說明
Math.sin(弧度 radian)	回傳正弦值
Math.cos(弧度 radian)	回傳餘弦值
Math.tan(弧度 radian)	回傳正切值

因為

- 單位弧度 (radian) 定義為「圓弧長度 (arc length) 等於半徑 (radius)」時的圓心角。
- $2 * \pi * \text{radius} = 360 \text{ degree}$
 - 圓弧長為 1 個 radius 的長度 $\Rightarrow 1 \text{ radian}$
 - $2 * \pi$ 個 radian = 360 degree
 - 同除以 2
 - ◆ π 個 radian = 180 degree

- 1 degree = $\pi / 180$ radian => 約為 0.01745329 radian
- 1 radian = degree * $180 / \pi$ => 約為 57.29578 degree

所以

- 弧度 = 角度 * $\pi / 180$

相同角度的轉換表								
角度單位	值							
轉	0	$\frac{1}{12}$	$\frac{1}{8}$	$\frac{1}{6}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	1
角度	0°	30°	45°	60°	90°	180°	270°	360°
弧度	0	$\frac{\pi}{6}$	$\frac{\pi}{4}$	$\frac{\pi}{3}$	$\frac{\pi}{2}$	π	$\frac{3\pi}{2}$	2π
梯度	0 ^g	$33\frac{1}{3}^g$	50 ^g	$66\frac{2}{3}^g$	100 ^g	200 ^g	300 ^g	400 ^g

(圖) 角度轉換表

範例 16-2-1.html

```
<html>
<head>
  <title>範例 16-2-1.html</title>
  <script>
    //角度
    let degree = 60;

    //正弦值
    console.log( Math.sin(degree * Math.PI / 180) );

    //餘弦值
    console.log( Math.cos(degree * Math.PI / 180) );

    //正切值
    console.log( Math.tan(degree * Math.PI / 180) );
  </script>
</head>
<body>

</body>
</html>
```

範例 16-2-2.html

```

<html>
<head>
  <title>範例 16-2-2.html</title>
  <script>
    //假設角度為 60 度
    var degree = 60;

    //算出弧度
    var radian = degree * Math.PI / 180;
    console.log(radian);

    //輸出 cos 60 度的值
    console.log( Math.cos(radian) );
  </script>
</head>
<body>

</body>
</html>

```

對三角函數有興趣的人，可以參考這個網址：

<https://www.shuxuele.com/sine-cosine-tangent.html>

16-3: 環狀排列物件

接下來，我們使用 `Math.cos()` 和 `Math.sin()` 進行環狀排列物件。

範例 16-3-1.html

```

<!DOCTYPE html>
<html>
<head>
  <title>範例 16-3-1.html</title>
  <style>
    /* 小圓圈 */
    .circle {
      position: absolute;
      left: 300px;
      top: 300px;

```

```

        display: none;
        width: 10px;
        height: 10px;
        background-color: gray;
        border-radius: 50%;
    }

    /* 圓形邊界 */
    .wrap {
        position: absolute;
        left: 100px;
        top: 100px;
        width: 500px;
        height: 500px;
        border: 1px solid red;
        border-radius: 50%;
    }
</style>
</head>
<body>
    <!-- div.wrap 是大的圓形邊界 -->
    <div class="wrap" id="wrap"></div>

    <!-- div.circle 是之後要連續環狀複製的小圓圈 -->
    <div class="circle" id="circle"></div>

    <script>
        //環狀排列的半徑
        var radius = 250;

        //取得繪製用圓形的元素
        var circle = document.getElementById('circle');

        //取得既有圓形（小圓）的元素
        var wrap = document.getElementById('wrap');

        //角度
        var degree = 0
    </script>

```

```

//幾個小圓
var qty = 6;

//每次小圓的移動次數
var moveLength = 360 / qty;

//給 clearInterval 用的實體 id
var t;

//定義繪圖函式
function draw() {
    //當前角度加上最後一步的距離，大於等於 360，則結束繪圖
    if(degree + moveLength >= 360) {
        clearInterval(t);
    };

    //計算當前的角度
    degree += moveLength;

    //計算
    var x = Math.cos(degree * (Math.PI / 180)) * radius;
    var y = Math.sin(degree * (Math.PI / 180)) * radius;

    //取得既有圓形的 left 與 top
    var wrap_left = wrap.offsetLeft;
    var wrap_top = wrap.offsetTop;

    //計算小圓的相對位置。element.style.left 需要填寫數值加 px 字串
    circle.style.left = (wrap_left + radius + x) + "px";
    circle.style.top = (wrap_top + radius - y) + "px";

    //left = 100 + 250 + 250
    //top = 100 + 250 - 0

    //顯示小圓
    circle.style.display = 'block';

```

```

//複製小圓
var newCircle = circle.cloneNode(true);

//將複製小圓放到 <body> 當中
document.body.appendChild(newCircle);
}

//透過 setInterval 來週期性執行繪圖函式
t = setInterval(function(){
    draw();
}, 500);
</script>
</body>
</html>

```

參考資料：

基於 javascript 實現按圖形排列 DIV 元素(一)

<https://codertw.com/%E5%89%8D%E7%AB%AF%E9%96%8B%E7%99%BC/252042/>

Module 17. window 物件

17-1: window 物件的方法

window 物件表示瀏覽器中開啟的視窗。

基本方法

方法	作用
alert()	顯示一個警示對話方塊，包含一條資訊和一個確定按鈕
confirm()	顯示一個確認對話方塊
prompt()	顯示一個提示對話方塊，提示使用者輸入資料

open()	開啟一個已存在的視窗，或者建立一個新視窗，並在該視窗中載入一個文件
close()	關閉一個開啟的視窗
navigate()	在當前視窗中顯示指定網頁
setTimeout()	設定一個定時器，在經過指定的時間間隔後呼叫一個函式
clearTimeout()	給指定的計時器復位
focus()	使一個 Window 物件得到當前焦點
blur()	使一個 Window 物件失去當前焦點

常用事件列表

事件	作用
onload	HTML 檔案載入瀏覽器時發生
onunload	HTML 檔案從瀏覽器刪除時發生
onfocus	視窗獲得焦點時發生
onblur	視窗失去焦點時發生
onhelp	使用者按下 F1 鍵時發生
onresize	使用者調整視窗大小時發生
onscroll	使用者滾動視窗時發生
onerror	載入 HTML 檔案出錯時發生

常見屬性

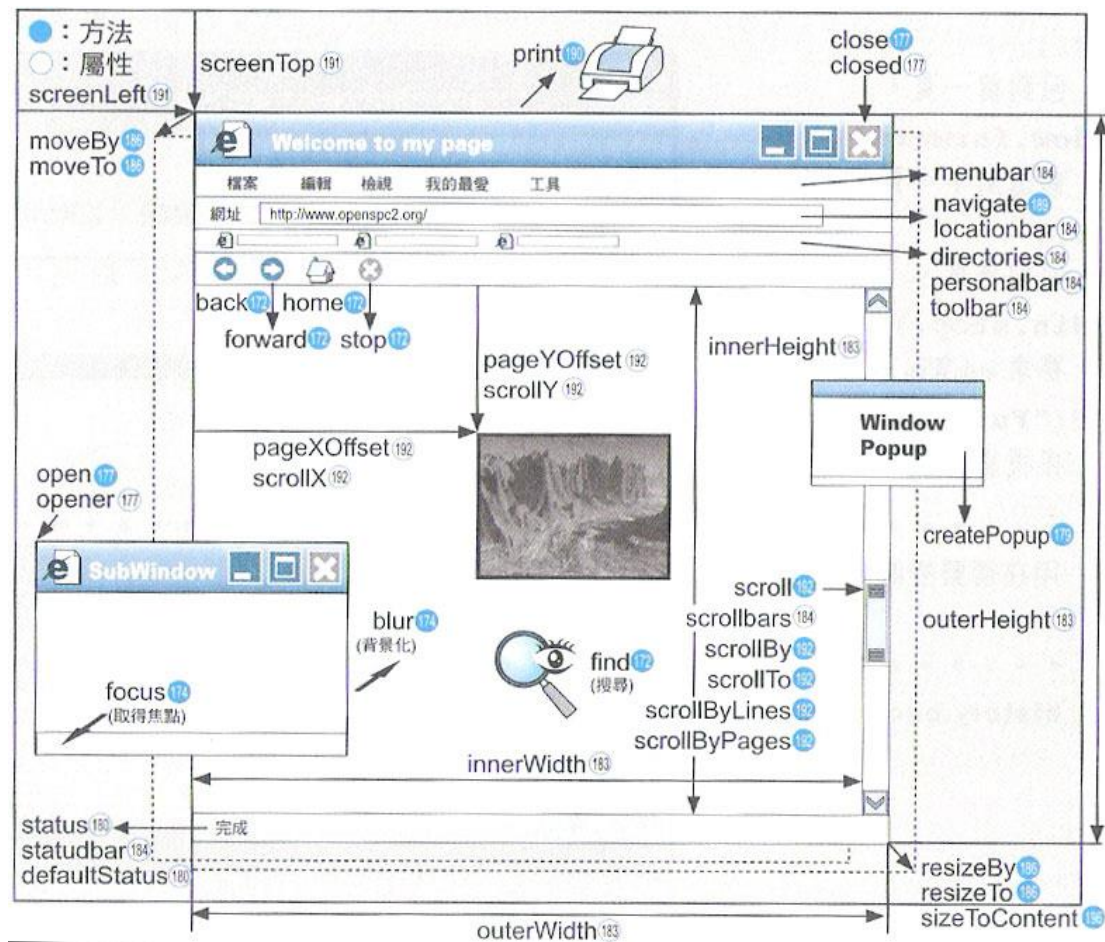
屬性	說明
name	指定視窗的名稱

parent	當前視窗（框架）的父視窗，使用它返回物件的方法和屬性
opener	返回產生當前視窗的視窗物件，使用它返回物件的方法和屬性
top	代表主視窗，是最頂層的視窗，也是所有其他視窗的父視窗。可通過該物件訪問當前視窗的方法和屬性
self	返回當前視窗的一個物件，可通過該物件訪問當前視窗的方法和屬性
defaultstatus	返回或設定將在瀏覽器狀態列中顯示的預設內容
status	返回或設定將在瀏覽器狀態列中顯示的指定內容

調整視窗的尺寸和位置

方法	用法
window.moveBy(dx,dy)	將瀏覽器視窗移動到指定位置（相對定位）
window.moveTo(x,y)	將瀏覽器視窗移動到指定位置（絕對定位）
window.resizeBy(dw,dh)	將瀏覽器視窗的尺寸改變指定的寬度和高度（相對調整視窗大小）
window.resizeTo(w,h)	將瀏覽器視窗的尺寸改變指定的寬度和高度（絕對調整視窗大小）
window.scrollTo(x,y) window.scrollTo({ top: y-coord, left: x-coord, behavior: "smooth" })	移到捲軸到指定的地方。 x 代表 left，y 代表 top。 若引數為物件，則： top: y 座標， left: x 座標， behavior: 字串格式，代表滾動行為 <ul style="list-style-type: none"> ● smooth（平滑滾動） ● instant（瞬間滾動）

	<ul style="list-style-type: none"> ● auto (預設值, 與 instant 一樣)
--	------------------------------------------------------------------------------



(圖) Windows 物件圖解

參考資料：

ddmg 筆記本

<http://www.ddmg.com.tw/WebApp/Learn/JavaScript/Base/window.html>

範例 17-1-1.html

```

<html>
<head>
  <title>範例 17-1-1.html</title>
</head>
<body>
  <a href="javascript:OpenNewWindow();">開啟新視窗</a>
  <a href="javascript:CloseNewWindow();">關閉新視窗</a>

```

```

<script>
//宣告變數，給 window 物件用
var myWin = null;

//定義開啟新視窗的函式
function OpenNewWindow(){
    myWin = window.open(
        "https://www.iii.org.tw/",
        "_blank",
        "height=600, width=800");
}

//定義開啟新視窗的函式
function CloseNewWindow(){
    myWin.close();

    //若是視窗已關閉，則跳出訊息
    if (myWin.closed) {
        alert('關閉了新視窗!!');
    }
}
</script>
</body>
</html>

```

範例 17-1-2.html

```

<html>
<head>
    <title>範例 17-1-2.html</title>
</head>
<body>
    <p>臣亮言：先帝創業未半而中道崩殂，今天下三分，益州疲弊，此誠危急存亡之秋也。然侍衛之臣不懈於內，忠志之士忘身於外者，蓋追先帝之殊遇，欲報之於陛下也。誠宜開張聖聽，以光先帝遺德，恢弘志士之氣，不宜妄自菲薄，引喻失義，以塞忠諫之路也。</p>
    <p>宮中府中，俱為一體；陟罰臧否，不宜異同：若有作奸犯科及為忠善者，宜付有司論其刑賞，以昭陛下平明之理；不宜偏私，使內外異法也。</p>

```

<p>侍中、侍郎郭攸之、費禕、董允等，此皆良實，志慮忠純，是以先帝簡拔以遺陛下：愚以為宮中之事，事無大小，悉以諮之，然後施行，必能裨補闕漏，有所廣益。

</p>

<p>將軍向寵，性行淑均，曉暢軍事，試用於昔日，先帝稱之曰“能”，是以眾議舉寵為督：愚以為營中之事，悉以諮之，必能使行陣和睦，優劣得所。</p>

<p>親賢臣，遠小人，此先漢所以興隆也；親小人，遠賢臣，此後漢所以傾頹也。先帝在時，每與臣論此事，未嘗不嘆息痛恨於桓、靈也。侍中、尚書、長史、參軍，此悉貞良死節之臣，願陛下親之、信之，則漢室之隆，可計日而待也。</p>

<p>臣本布衣，躬耕於南陽，苟全性命於亂世，不求聞達於諸侯。先帝不以臣卑鄙，猥自枉屈，三顧臣於草廬之中，諮臣以當世之事，由是感激，遂許先帝以驅馳。後值傾覆，受任於敗軍之際，奉命於危難之間：爾來二十有一年矣。</p>

<p>先帝知臣謹慎，故臨崩寄臣以大事也。受命以來，夙夜憂嘆，恐託付不效，以傷先帝之明；故五月渡瀘，深入不毛。今南方已定，兵甲已足，當獎率三軍，北定中原，庶竭駑鈍，攘除奸兇，興復漢室，還於舊都。此臣所以報先帝而忠陛下之職分也。至於斟酌損益，進盡忠言，則攸之、禕、允之任也。</p>

<p>願陛下託臣以討賊興復之效，不效，則治臣之罪，以告先帝之靈。若無興德之言，則責攸之、禕、允等之慢，以彰其咎；陛下亦宜自謀，以諮諏善道，察納雅言，深追先帝遺詔。臣不勝受恩感激。</p>

<p>今當遠離，臨表涕零，不知所言。</p>

<hr />

.
. .

<hr />

<p>臣亮言：先帝創業未半而中道崩殂，今天下三分，益州疲弊，此誠危急存亡之秋也。然侍衛之臣不懈於內，忠志之士忘身於外者，蓋追先帝之殊遇，欲報之於陛下也。誠宜開張聖聽，以光先帝遺德，恢弘志士之氣，不宜妄自菲薄，引喻失義，以塞忠諫之路也。</p>

<p>宮中府中，俱為一體；陟罰臧否，不宜異同：若有作奸犯科及為忠善者，宜付有司論其刑賞，以昭陛下平明之理；不宜偏私，使內外異法也。</p>

<p>侍中、侍郎郭攸之、費禕、董允等，此皆良實，志慮忠純，是以先帝簡拔以遺陛下：愚以為宮中之事，事無大小，悉以諮之，然後施行，必能裨補闕漏，有所廣益。

</p>

<p>將軍向寵，性行淑均，曉暢軍事，試用於昔日，先帝稱之曰“能”，是以眾議舉寵為督：愚以為營中之事，悉以諮之，必能使行陣和睦，優劣得所。</p>

<p>親賢臣，遠小人，此先漢所以興隆也；親小人，遠賢臣，此後漢所以傾頹也。先帝在時，每與臣論此事，未嘗不嘆息痛恨於桓、靈也。侍中、尚書、長史、參軍，此悉貞良死節之臣，願陛下親之、信之，則漢室之隆，可計日而待也。</p>

<p>臣本布衣，躬耕於南陽，苟全性命於亂世，不求聞達於諸侯。先帝不以臣卑鄙，猥自枉屈，三顧臣於草廬之中，諮臣以當世之事，由是感激，遂許先帝以驅馳。後值傾覆，受任於敗軍之際，奉命於危難之間：爾來二十有一年矣。</p>

<p>先帝知臣謹慎，故臨崩寄臣以大事也。受命以來，夙夜憂嘆，恐託付不效，以傷先帝之明；故五月渡瀘，深入不毛。今南方已定，兵甲已足，當獎率三軍，北定中原，庶竭駑鈍，攘除奸兇，興復漢室，還於舊都。此臣所以報先帝而忠陛下之職分也。至於斟酌損益，進盡忠言，則攸之、禕、允之任也。</p>

<p>願陛下託臣以討賊興復之效，不效，則治臣之罪，以告先帝之靈。若無興德之言，則責攸之、禕、允等之慢，以彰其咎；陛下亦宜自謀，以諮諏善道，察納雅言，深追先帝遺詔。臣不勝受恩感激。</p>

<p>今當遠離，臨表涕零，不知所言。</p>

移動到 y = 100 的位置

移動到 y = 400 的位置

平滑移動到 y = 100 的位置

<script>

//平滑移動

```
function move() {  
    window.scrollTo({  
        top: 100,  
        behavior: "smooth"  
    });  
}
```

</script>

</body>

</html>

17-2: window 的子物件

- document 物件：表示瀏覽器中載入頁面的文件物件；
- location 物件：包含了瀏覽器當前的 URL 資訊；
- navigation 物件：包含了瀏覽器本身的資訊；
- screen 物件：包含了客戶端螢幕及渲染能力的資訊；
- history 物件：包含了瀏覽器訪問網頁的歷史資訊。

範例 17-2-1.html

```
<html>
<head>
  <title>範例 17-2-1.html</title>
</head>
<body>
  <a href="https://tw.yahoo.com/" target="_blank">yahoo!</a> -
  <a href="https://zh.wikipedia.org/" target="_blank">維基百科</a> -
  <a href="https://www.cwb.gov.tw/" target="_blank">中央氣象局</a>

  <script>
    //window.document 的範例
    alert('網頁標題: ' + window.document.title);
    alert('網頁連結數: ' + window.document.links.length);
  </script>
</body>
</html>
```

範例 17-2-2.html

```
<html>
<head>
  <title>範例 17-2-2.html</title>
  <script>
    //window.location 的範例
    document.write('目前頁面: ' + window.location.href + '<br />');
    document.write('目前網域名稱: ' + window.location.hostname + '<br />');
    document.write('目前頁面路徑: ' + window.location.pathname + '<br />');
    document.write('目前通訊協定: ' + window.location.protocol + '<br />');
    document.write('目前 port 號: ' + window.location.port + '<br />');
  </script>
</head>
<body>

</body>
</html>
```

範例 17-2-3.html

```
<html>
<head>
  <title>範例 17-2-3.html</title>
  <script>
    //window.navigator 的範例
    var sBrowser, sUsrAg = navigator.userAgent;

    if(sUsrAg.indexOf("Chrome") > -1) {
      sBrowser = "Google Chrome";
    } else if (sUsrAg.indexOf("Safari") > -1) {
      sBrowser = "Apple Safari";
    } else if (sUsrAg.indexOf("Opera") > -1) {
      sBrowser = "Opera";
    } else if (sUsrAg.indexOf("Firefox") > -1) {
      sBrowser = "Mozilla Firefox";
    } else if (sUsrAg.indexOf("MSIE") > -1) {
      sBrowser = "Microsoft Internet Explorer";
    }

    document.write("您正在使用: " + sBrowser);
  </script>
</head>
<body>

</body>
</html>
```

範例 17-2-4.html

```
<html>
<head>
  <title>範例 17-2-4.html</title>
  <script>
    //window.screen 的範例
    if (window.screen.pixelDepth < 8) {
      // use low-color version of page
      alert('您使用低彩的畫質觀看此頁面');
    } else {
      // use regular, colorful page
    }
  </script>

```

```

    alert('您使用一般、高彩的畫質觀看此頁面');
}
</script>
</head>
<body>

</body>
</html>

```

範例 17-2-5.html
<pre> <html> <head> <title>範例 17-2-5.html</title> </head> <body> .back() 上一頁
 .go(-1) 上一頁
 .forward() 下一頁
 .go(1) 下一頁 </body> </html> </pre>

17-3: document 的常用屬性

屬性	說明
document.anchors	對文件中所有 Anchor 物件的引用
document.links	回傳一個 document 中所有具有 href 屬性值的 <a> 元素與 <a> 元素的集合。
document.forms	回傳目前 document 中 <form>元素的集合。
document.images	回傳目前 document 中所有 image 元素的集合。
document.URL	回傳當前文件的 url
document.title	回傳當前文件的標題
document.body	回傳 body 元素節點

補充說明

創造元素

document.createElement 建立 html 元素

document.createTextNode 建立 text 節點

document.createAttribute 建立屬性

```
var btn = document.createElement("button");
var txt = document.createTextNode("CLICK ME");
btn.appendChild(txt);

var h = document.createElement("h1")
var t = document.createTextNode("Hello World");
h.appendChild(t);

var att = document.createAttribute("class");
att.value="myClass";
document.getElementsByTagName("H1")[0].setAttributeNode(att);
```

範例 17-3-1.html

```
<html>
<head>
  <title>範例 17-3-1.html</title>
</head>
<body>
  <a href="https://tw.yahoo.com/" target="_blank">yahoo!</a> <br />
  <a href="https://zh.wikipedia.org/" target="_blank">維基百科</a> <br />
  <a href="https://www.cwb.gov.tw/" target="_blank">中央氣象局</a> <br />

  <script>
var links = document.links;
for(var i = 0; i < links.length; i++) {
  var linkHref = document.createTextNode(links[i].href);
  var lineBreak = document.createElement("br");
  document.body.appendChild(linkHref);
  document.body.appendChild(lineBreak);
}
</script>
</body>
</html>
```

範例 17-3-2.html


```

<html>
<head>
  <title>範例 17-3-2.html</title>
</head>
<body>
  <form id="robby">
    <input type="button" onclick="alert(document.forms[0].id);"
      value="robby's form" />
  </form>

  <form id="dave">
    <input type="button" onclick="alert(document.forms[1].id);"
      value="dave's form" />
  </form>

  <form id="paul">
    <input type="button" onclick="alert(document.forms[2].id);"
      value="paul's form" />
  </form>
</body>
</html>

```

範例 17-3-3.html

```

<html>
<head>
  <title>範例 17-3-3.html</title>
</head>
<body>
  
  <hr />
  
  <hr />
  
  <hr />

  <script>
    var imgList = document.images;
    for(var i = 0; i < imgList.length; i++) {

```

```
document.write(imgList[i].src);
document.write("<br />");
}
</script>
</body>
</html>
```

Module 18. 事件處理

事件 (Events) 是由特定動作發生時所引發的訊號，舉例來說，使用者點選或移動滑鼠，或是瀏覽器的載入網頁，都可以看成是事件的產生。對於特定的事件，我們可以在瀏覽器內偵測得之，並以特定的程式來對此事件做出反應，此程式即稱為「事件處理器」(Event handlers)。

若是想要對事件處理有更深的了解，可以參考以下的連結：

w3schools - HTML Event Attributes

https://www.w3schools.com/tags/ref_eventattributes.asp

18-1: 標籤內的事件處理器

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。

範例 18-1-1.html

```
<html>
<head>
  <title>範例 18-1-1.html</title>
</head>
<body>
  <a href="https://tw.yahoo.com/" onclick="javascript: alert(document.links[0].href);">奇摩連結</a>
</body>
</html>
```

範例 18-1-2.html

```
<html>
<head>
  <title>範例 18-1-2.html</title>
</head>
<body>
  <input type="text" name="txt" id="txt" onfocus="showFocus()" />

  <script>
    //在文字欄位中，寫入文字
    function showFocus(){
      document.getElementById('txt').value = '你好!!';
    }
  </script>
</body>
</html>
```

範例 18-1-3.html

```
<html>
<head>
  <title>範例 18-1-3.html</title>
</head>
<body>
  <div id="box" style="width: 300px; height: 100px; background-color: #9999ff;"
    onmousemove="move()"
    onmouseout="out()">請用滑鼠移過此元素</div>

  <script>
    //onmousemove 事件所使用的函式
    function move(){
      document.getElementById("box").style.backgroundColor = '#33ffff';
    }

    //onmouseout 事件所使用的函式
    function out(){
      document.getElementById("box").style.backgroundColor = '#ff0000';
    }
  </script>
</body>
</html>
```

18-2: addEventListener

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

範例

將 id 為 myBtn 的元素，註冊一個 click 事件，事件運作是將 id 為 demo 的元素的節點內文，設定成 Hello World：

```
document.getElementById("myBtn").addEventListener("click", function(){
    document.getElementById("demo").innerHTML = "Hello World";
});
```

移除已註冊事件

```
document.getElementById("myBtn").removeEventListener("click", function(){});
```

範例 18-2-1.html

```
<html>
<head>
  <title>範例 18-2-1.html</title>
</head>
<body>
  <button id="btn" style="width: 200px; height: 100px;">按下後改變文字</button>
  <br /><br /><br /><br />
  <div style="width: 150px; height: 100px; border: 1px solid;">請用滑鼠移過此元素</div>
  <br /><br /><br /><br />
  

  <script>
    //對 button#btn 註冊 click 事件
    document.getElementById('btn').addEventListener('click', function(event){
      //跳出文字訊息
      alert(this.innerText);
```

```

    //修改文字
    this.innerText = '已改變';
});

//對 div 註冊兩個事件，分別是滑鼠移入和移出
let div = document.getElementsByTagName('div')[0];
div.addEventListener('mousemove', function(event){
    this.style.backgroundColor = '#33ffff';
});
div.addEventListener('mouseout', function(event){
    this.style.backgroundColor = '#ff0000';
});

//對 img.puppy 註冊滑鼠移入事件
document.querySelector('img.puppy').addEventListener('mousemove', function(event){
    this.setAttribute(
        "src",
        "https://ballparkdigest.com/wp-content/uploads/2020/02/New-Trenton-Thunder-bat-dog-300x300.jpg"
    );
});
</script>
</body>
</html>

```

18-3: 滑鼠事件

on 事件	事件
onclick	click
ondblclick	dblclick
onmousedown	mousedown
onmousemove	mousemove
onmouseout	mouseout

onmouseover	mouseover
onmouseup	mouseup

範例 18-3-1.html

```

<html>
<head>
  <title>範例 18-3-1.html</title>
</head>
<body>
  <button id="btn" style="width: 100px; height: 50px;">連點兩下: 0</button>

  <script>
    //計數用的變數
    let count = 0;
    let width = 100;
    let height = 50;

    //註冊雙擊事件，每點兩下，按鈕會慢慢變大
    document.getElementById('btn').addEventListener('dblclick', function(){
      count += 1; //等於 count++
      width += 20;
      height += 20;
      this.innerText = '連點兩下: ' + count;
      this.style.width = width;
      this.style.height = height;
    });
  </script>
</body>
</html>

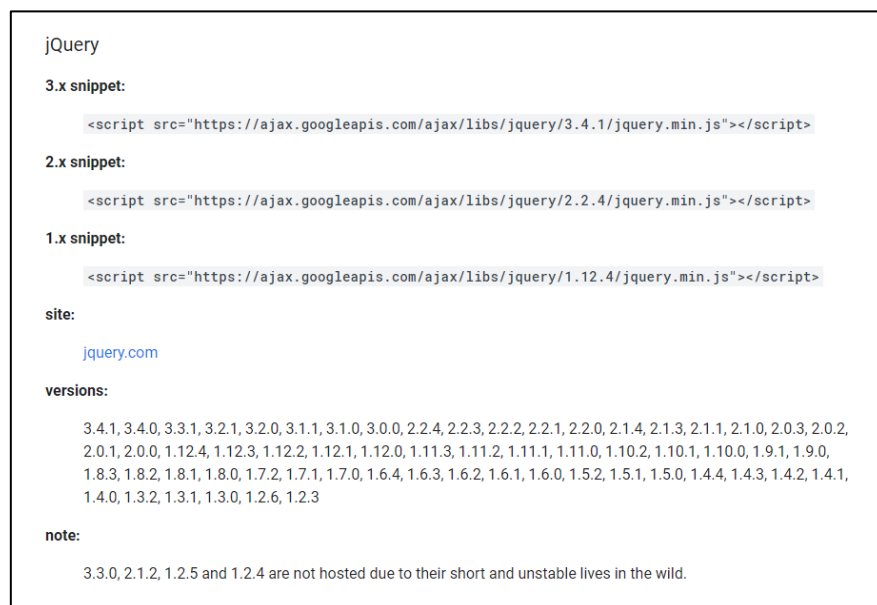
```

Module 19. AJAX

19-1: 不刷新頁面更新內容

AJAX 為 Asynchronus(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，以下以 jQuery 的 Ajax 作為範例。

jQuery 是一個以 Javascript 來編寫的函式庫，簡單來說就是先幫你實作了很多 Javascript 的函數功能，用途是讓開發者可以更輕鬆方便的製作網站功能，最重要的是它是免費的。我們先到 <https://developers.google.com/speed/libraries>，去尋找 jQuery 的線上函式庫檔案，放到我們的 HTML head 當中。



(圖) jQuery 的 CDN

範例 19-1-1.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
```



```

<title>範例 19-1-1.html</title>
</head>
<body>
  <button name="btn_get" id="btn_get">使用 GET 傳遞</button>
  <br />
  <button name="btn_post" id="btn_post">使用 POST 傳遞</button>
  <br />
  <button name="btn_ajax" id="btn_ajax">使用 Ajax 傳遞</button>

  <div id="message"></div>

  <!-- 引入 jQuery 的函式庫 -->
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
  <script>
    //取得 document 當中的 div#message 元素
    let div = document.querySelector("div#message");

    $(document).ready(function(){
      /**
       * 如果使用 GET 方式，練習時可以使用下列的網址：
       * https://data.taipei/opendata/datalist/apiAccess?scope=datasetMetadataSearch
       */

      //使用 GET 傳遞
      $(document).on('click', 'button#btn_get', function(){
        $.get("https://httpbin.org/get", function( objJson ) {
          div.innerHTML = JSON.stringify(objJson);
        });
      });

      //使用 POST 傳遞
      $(document).on('click', 'button#btn_post', function(){
        $.post("https://httpbin.org/post ", { postMethod: "1" } )
        .done(function(objJson){
          div.innerHTML = JSON.stringify(objJson);
        });
      });
    });
  </script>

```

```

});

//使用 Ajax 傳遞
$(document).on('click', 'button#btn_ajax', function(){
    $.ajax({
        method: 'POST', // GET
        url: "https://httpbin.org/post",
        data: {
            postMethod: "1"
        }
    }).done(function(objJson) {
        div.innerHTML = JSON.stringify(objJson);
    });
});
});
</script>
</body>
</html>

```

19-2: XMLHttpRequest

剛剛有提到 AJAX 為 Asynchronus(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

<code>var req = new XMLHttpRequest();</code>

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

<code>var req = new XMLHttpRequest();</code>

<code>//建立連線資訊</code>

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

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

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

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

```
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
```

範例 19-2-1.html

```
<html>  
<head>  
    <title>範例 19-2-1.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();
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>

```

範例 19-2-2.html

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>範例 19-2-2.html</title>
</head>
<body>
    <input type="file" name="myFile" id="myFile" onchange="uploadFile()" />

```

```

<br />
<progress id="p" value="0" max="100" style="width: 500px; height: 50px"></progress>
<br />
<div id="msg"></div>

<script>
function uploadFile(){
    //取得 progress 物件
    var progressBar = document.getElementById('p');
    var myFile = document.getElementById('myFile').files[0];
    var div = document.getElementById('msg');

    //建立 XMLHttpRequest 物件，來進行 ajax 非同步傳輸
    var req = new XMLHttpRequest();
    req.open('post','https://httpbin.org/post');
    req.setRequestHeader("Content-Type", "multipart/form-data");

    //建立 FormData 物件，夾帶變數(或檔案)，搭配 req.send( formdata ) 送到後端頁面去處理
    var formdata = new FormData();
    formdata.append("myFile", myFile);

    //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');
    }

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

    //顯示當前上傳的進度
    req.upload.onprogress = function(event) {

```

```

//計算上傳百分比
var percent = (event.loaded / event.total) * 100;

//顯示在 div 的內文中
div.innerHTML = percent + '%';

//將目前上傳的進度，分別放到 progress 元素的屬性當中
progressBar.max = event.total;
progressBar.value = event.loaded;
};

req.send(formdata); //執行連線
}

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

```

參考資料：

AJAX 利用 XHR2 Progress Event 實作下載進度列

<https://blog.toright.com/posts/3585/ajax-%E5%88%A9%E7%94%A8-xhr2-%E5%AF%A6%E4%BD%9C%E4%B8%8B%E8%BC%89%E9%80%B2%E5%BA%A6%E5%88%97-progress-event.html>

19-3: fetch() 方法

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 19-3-1.html

```

<html>
<head>
<title>範例 19-3-1.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 19-3-2.html

```

<html>
<head>
  <title>範例 19-3-2.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：

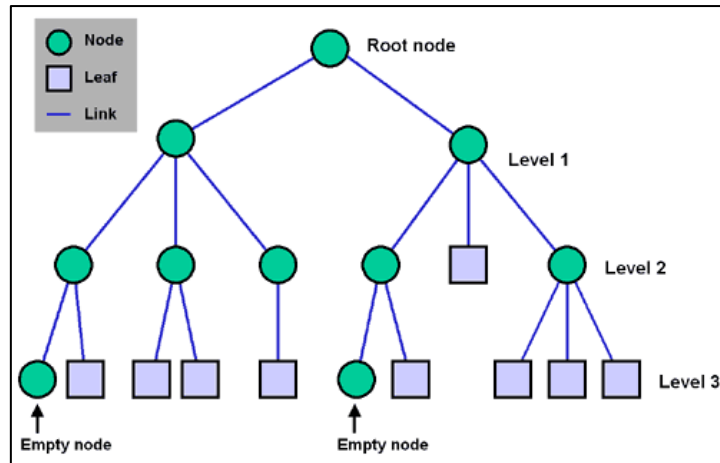
鐵人賽：ES6 原生 Fetch 遠端資料方法

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

Module 20. 操作 DOM

20-1: 取得 DOM 元素

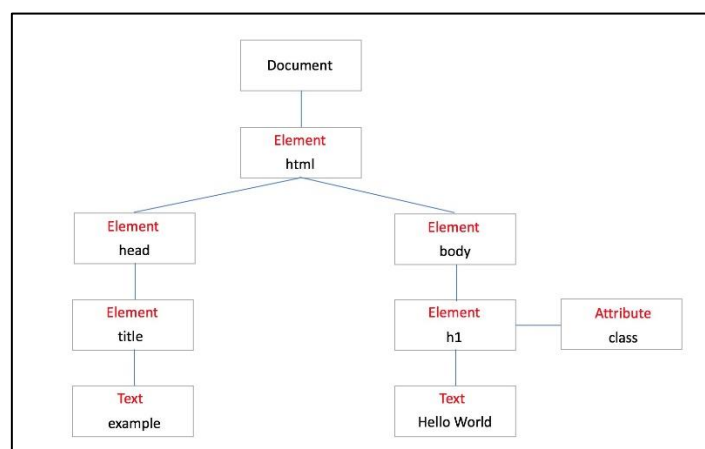
我們來解析一下 DOM。



(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：

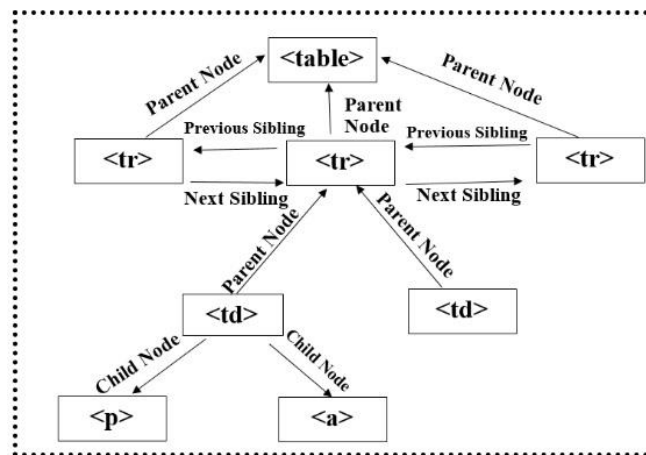
- Document
 - Document 就是指這份文件，也就是這份 HTML 檔的開端，所有的一切都會從 Document 開始往下進行。
- Element
 - Element 就是指文件內的各個標籤，因此像是 <div>、<p> 等等各種 HTML Tag 都是被歸類在 Element 裡面。
- Text
 - Text 就是指被各個標籤包起來的文字，舉例來說在 <h1>Hello World</h1> 中，Hello World 被 <h1> 這個 Element 包起來，因此 Hello World 就是此 Element 的 Text
- Attribute
 - Attribute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

20-2: 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 20-2-1.html

```
<html>
<head>
  <title>範例 20-2-1.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";

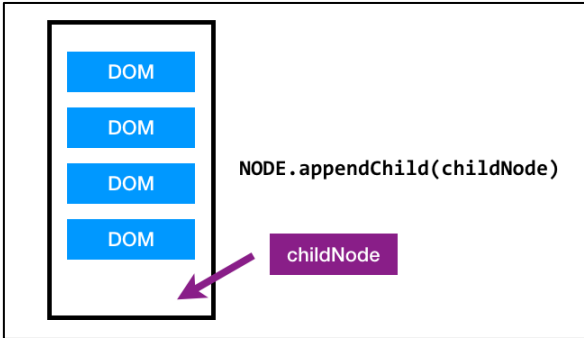
//建立 textNode 文字節點
var textNode = document.createTextNode("Hello world!");

//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 20-2-2.html

```
<html>
<head>
  <title>範例 20-2-2.html</title>
</head>
<body>
  <ul id="myList">
    <li>Item 01</li>
    <li>Item 02</li>
    <li>Item 03</li>
  </ul>

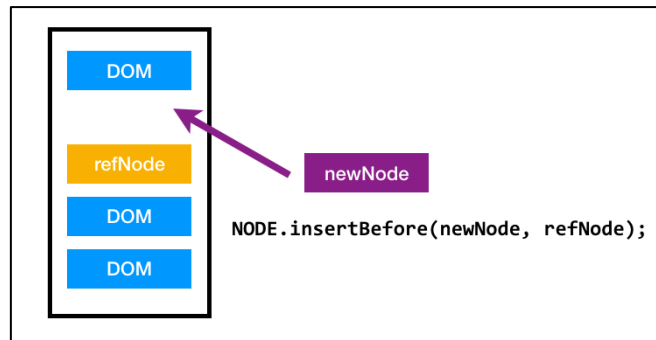
  <script>
    // 取得容器
    var myList = document.getElementById('myList');

    // 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 20-2-3.html

```
<html>
<head>
  <title>範例 20-2-3.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');

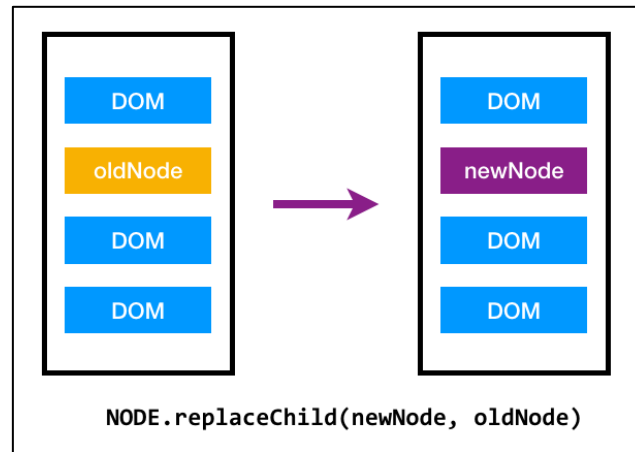
    // 取得 "<li>Item 02</li>" 的元素
    var refNode = document.querySelectorAll('li')[1];

    // 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 20-2-4.html

```
<html>
<head>
  <title>範例 20-2-4.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');
```

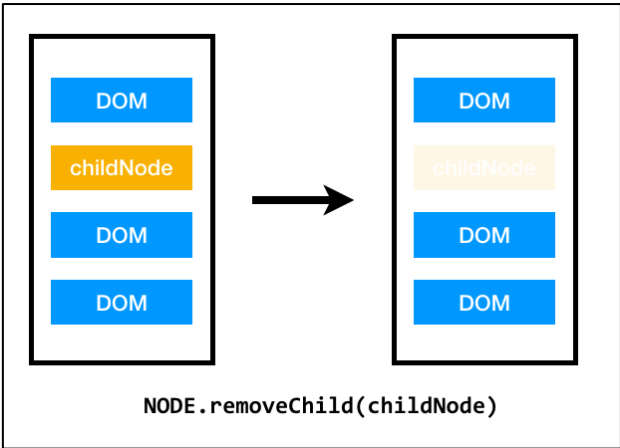
```
// 建立 textNode 文字節點
var textNode = document.createTextNode("Hello world!");
newNode.appendChild(textNode);

// 將原有的 oldNode 替換成新節點 newNode
myList.replaceChild(newNode, oldNode);

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

20-3: 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 20-3-1.html

<html>
<head>
  <title>範例 20-3-1.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 21. 深入 DOM 元素

21-1: 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 21-1-1.html
<pre><html> <head> <title>範例 21-1-1.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>

```

21-2: 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 21-2-1.html

```

<html>
<head>
  <title>範例 21-2-1.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>

```

21-3: 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 21-3-1.html

```
<html>
```

```
<head>
```

```
<title>範例 21-3-1.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
border-bottom-style	borderBottomStyle

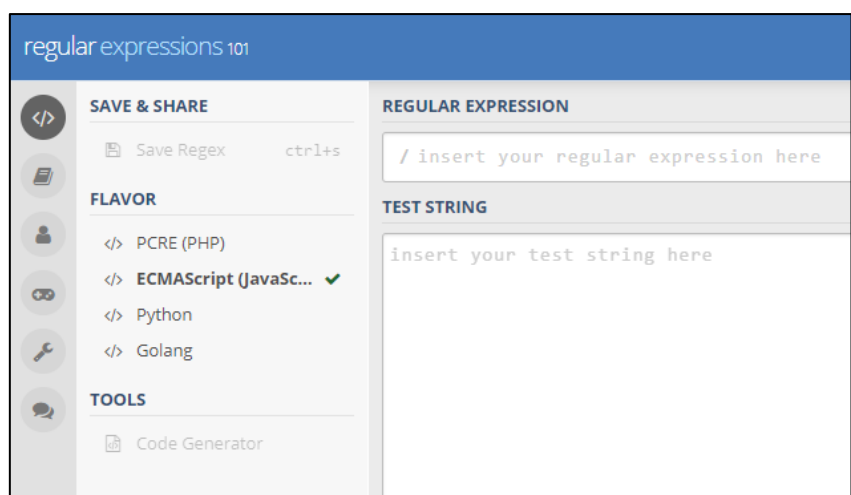
CSS	JavaScript
border-bottom-width	borderBottomWidth
border-color	borderColor
border-left	borderLeft
border-left-color	borderLeftColor
border-left-style	borderLeftStyle
border-left-width	borderLeftWidth
border-right	borderRight
border-right-color	borderRightColor
border-right-style	borderRightStyle
border-right-width	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
font	font

CSS	JavaScript
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
page-break-after	pageBreakAfter

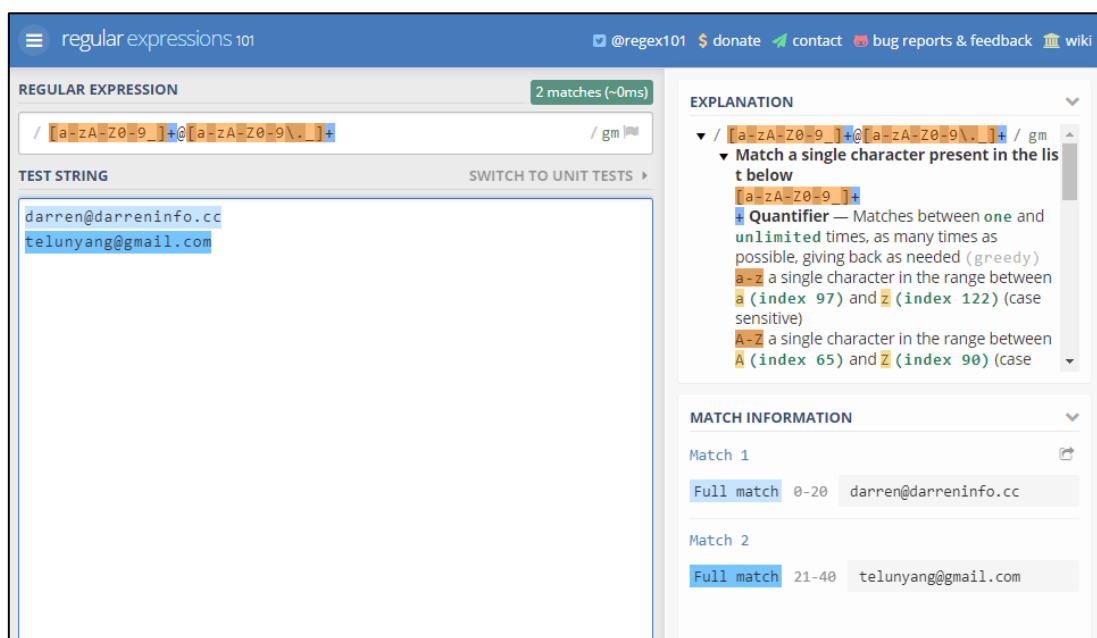
CSS	JavaScript
page-break-before	pageBreakBefore
position	position
stroke-dasharray	strokeDasharray
stroke-dashoffset	strokeDashoffset
stroke-width	strokeWidth
text-align	textAlign
text-decoration	textDecoration
text-indent	textIndent
text-transform	textTransform
top	top
vertical-align	verticalAlign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

Module 22 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

說明	正規表達式	範例
避開特殊字元	\ ex. \?	?
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	a b	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	(?<!=John) Walker	Johnnie Walker

說明	正規表達式	範例
(這位置左邊不能出現什麼)		

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

22-1: 單一字元表示法

範例 22-1-1.html
<pre> <html> <head> <title>範例 20-1-1.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); } </pre>

```

//任何數字
str = 'AB123';
pattern = /\d/g;
match = null;
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>

```

22-2: 多字元表示法

範例 22-2-1.html

```

<html>
<head>
    <title>範例 22-2-1.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
    </script>

```

```

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 = 'addbaaaaaccbaa';
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>

```

22-3: 表單送出前的檢查

範例 22-3-1.html

```

<html>
<head>
  <title>範例 22-3-1.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>

```

Module 23. Canvas

JavaScript 不僅是只能操作文字和數字，也可以使用 JavaScript 和 HTML canvas 來繪製圖片。將 canvas 元素當作一塊空白畫布或一頁紙，可以將想要的內容繪製在上面。

23-1 繪製圖片

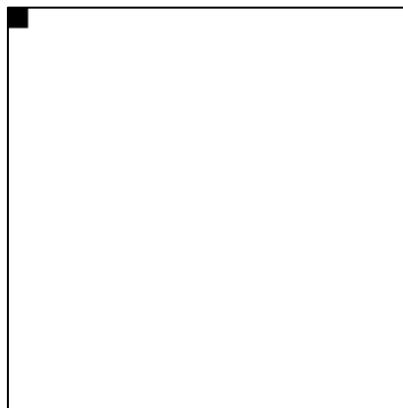
範例 23-1-1.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    //取得 canvas 元素
    let canvas = document.getElementById("canvas");

    //取得二維圖像的繪製環境(drawing context)
    let ctx = canvas.getContext("2d");

    /**
     * 在座標 (0, 0) 繪製一個 10 畫素 x 10 畫素的矩形。
     * void ctx.fillRect(x, y, width, height);
     */
    ctx.fillRect(0, 0, 10, 10);
  </script>
</body>
</html>
```



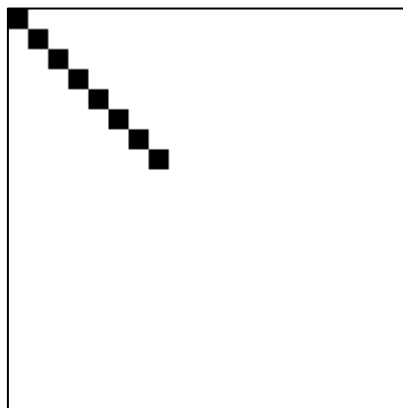
(圖) 在座標 (0, 0) 繪製一個 10 畫素 x 10 畫素的矩形

範例 23-1-2.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //繪製多個方塊
    for(let i = 0; i < 8; i++){
      ctx.fillRect(i * 10, i * 10, 10, 10);
    }
  </script>
</body>
</html>
```



(圖) 繪製多個方塊

範例 23-1-3.html

```
<!DOCTYPE html>
<html>
```

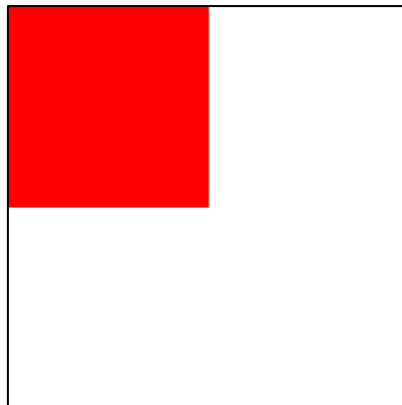
```

<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //更改繪製顏色
    ctx.fillStyle = 'Red'; //也可以用 RGB 碼，例如 #9487ff
    ctx.fillRect(0, 0, 100, 100);
  </script>
</body>
</html>

```



(圖) 更改繪製顏色

範例 23-1-4.html

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

```

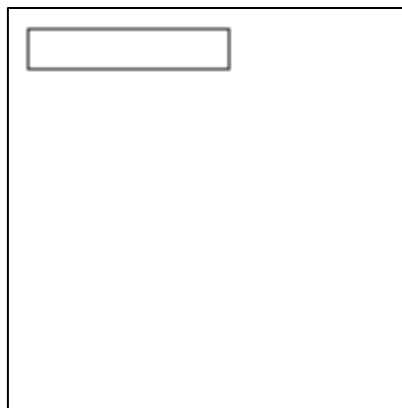


```

<script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    /**
     * 繪製矩形邊框。在畫布左上角 (10,10) 個畫素的位置，
     * 畫了一個寬 100 畫素、高 20 畫素的矩形。
     * void ctx.strokeRect(x, y, width, height);
     */
    ctx.strokeRect(10, 10, 100, 20);
</script>
</body>
</html>

```



(圖) 繪製框線

範例 23-1-5.html

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Canvas</title>
</head>
<body>
    <canvas id="canvas" width="200" height="200"></canvas>

    <script>
        let canvas = document.getElementById("canvas");

```

```

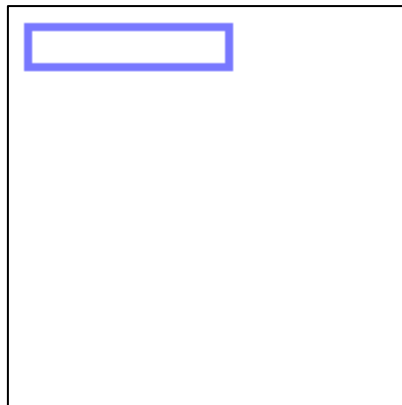
let ctx = canvas.getContext("2d");

//更改邊框顏色
ctx.strokeStyle = '#7878ff';

//更改線條粗細
ctx.lineWidth = 4;

/**
 * 繪製矩形邊框。在畫布左上角 (10,10) 個畫素的位置，
 * 畫了一個寬 100 畫素、高 20 畫素的矩形。
 * void ctx.strokeRect(x, y, width, height);
 */
ctx.strokeRect(10, 10, 100, 20);
</script>
</body>
</html>

```



(圖) 框線變色

範例 23-1-6.html

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

```

```
<script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //更改邊框顏色
    ctx.strokeStyle = '#7878ff';

    //更改線條粗細
    ctx.lineWidth = 4;

    //告訴畫布，我們要開始建立繪製路徑
    ctx.beginPath();

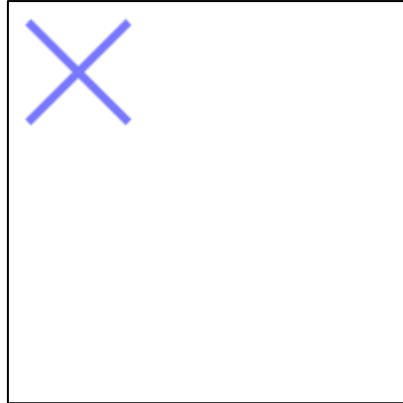
    /**
     * 想像一下，將畫筆提起，移到畫布（10，10）的座標，再放到畫布上
     * void ctx.moveTo(x, y);
     */
    ctx.moveTo(10, 10);

    /**
     * 想像一下，從畫筆目前的位置（10，10），畫一條線到（60，60）的座標
     * void ctx.lineTo(x, y);
     */
    ctx.lineTo(60, 60);

    //想像一下，將畫筆提起，移到畫布（60，10）的座標，再放到畫布上
    ctx.moveTo(60, 10);

    //想像一下，從畫筆目前的位置（60，10），畫一條線到（10，60）的座標
    ctx.lineTo(10, 60);

    //目前為止，還在想像階段，需要使用 ctx.stroke()，才會實際繪製線條
    ctx.stroke();
</script>
</body>
</html>
```



(圖) 繪製交叉線條

範例 23-1-7.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");
    ctx.fillStyle = 'SkyBlue';

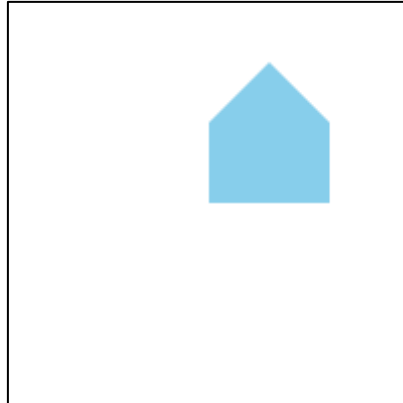
    //在這裡，我們要繪製一個「實心」圖形
    ctx.beginPath();
    ctx.moveTo(100, 100);

    //由於這次要畫的是實心圖形，所以可以連續使用 ctx.lineTo()，把填充
    的框線先畫出
    ctx.lineTo(100, 60);
    ctx.lineTo(130, 30);
    ctx.lineTo(160, 60);
    ctx.lineTo(160, 100);
    ctx.lineTo(160, 100);
```

```

        //繪製實心圖形
        ctx.fill();
    </script>
</body>
</html>

```



(圖) 實心圖形

範例 23-1-8.html

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Canvas</title>
</head>
<body>
    <canvas id="canvas" width="200" height="200"></canvas>

    <script>
        let canvas = document.getElementById("canvas");
        let ctx = canvas.getContext("2d");
        ctx.lineWidth = 2;
        ctx.strokeStyle = 'Green';

        //這個範例，將「各別」繪製圓弧和圓

        /**
         * 繪製弧線
         * void ctx.arc(x, y, radius, startAngle, endAngle [, anticlockwise]);

```

```

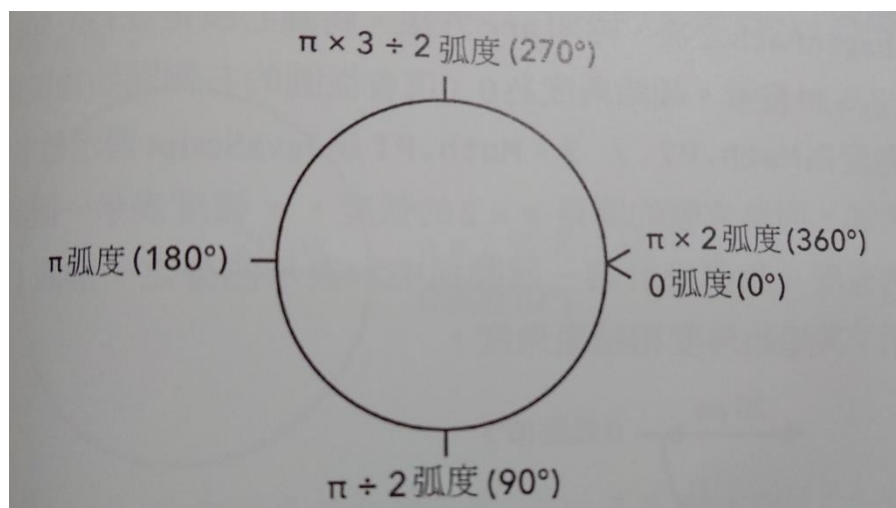
*/

//從 0 弧度開始，畫到  $\pi / 2$  弧度（畫筆從右往下畫弧）
ctx.beginPath();
ctx.arc(50, 50, 20, 0, Math.PI / 2, false);
ctx.stroke();

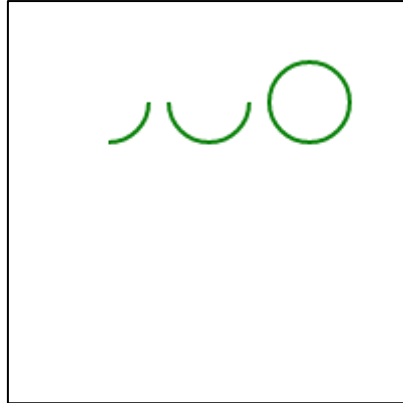
//從 0 弧度開始，畫到  $\pi$  弧度（畫筆從右往下再往左畫弧）
ctx.beginPath();
ctx.arc(100, 50, 20, 0, Math.PI, false);
ctx.stroke();

//從 0 弧度開始，畫到  $\pi * 2$  弧度（畫筆從右開始畫弧，畫成一個圈）
ctx.beginPath();
ctx.arc(150, 50, 20, 0, Math.PI * 2, false);
ctx.stroke();
</script>
</body>
</html>

```



(圖) 角度和弧度，從圓圈的右邊開始，沿順時針方向移動



(圖) 繪製圓弧和圓

範例 23-1-9.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");
    ctx.lineWidth = 4;

    /**
     * 這個範例，將「各別」繪製圓弧和圓
     *
     * 繪製弧線
     * void ctx.arc(x, y, radius, startAngle, endAngle [, anticlockwise]);
     */

    //定義一個函式來繪製多個圓
    let circle = function(x, y, radius){
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2, false);
      ctx.stroke();
    };
  </script>
</body>
</html>
```

```
};

ctx.strokeStyle = 'Red';
circle(100, 100, 10);

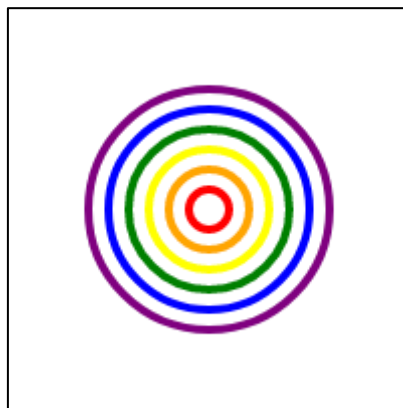
ctx.strokeStyle = 'Orange';
circle(100, 100, 20);

ctx.strokeStyle = 'Yellow';
circle(100, 100, 30);

ctx.strokeStyle = 'Green';
circle(100, 100, 40);

ctx.strokeStyle = 'Blue';
circle(100, 100, 50);

ctx.strokeStyle = 'Purple';
circle(100, 100, 60);
</script>
</body>
</html>
```



(圖) 繪製六個不同半徑、不同顏色的圓圈

23-2 動態效果

將使用 `setInterval()`，執行「畫布清空後再繪製」的流程，且在每一次流程當

中，不斷改變繪製圖形的參值，使其產生變化。

範例 23-2-1.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //移動位置初始值
    let position = 0;

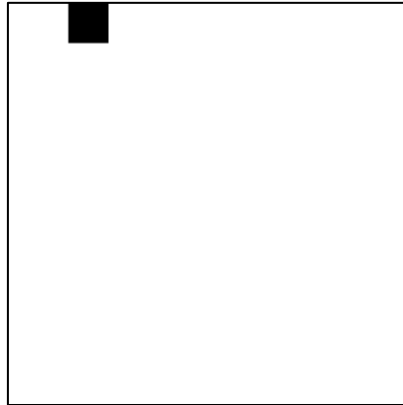
    //設定每 30 毫秒執行一次
    setInterval(function(){
      /**
       * 清空畫布
       * void ctx.clearRect(x, y, width, height);
       */
      ctx.clearRect(0, 0, 200, 200);

      //繪製矩形
      ctx.fillRect(position, 0, 20, 20);

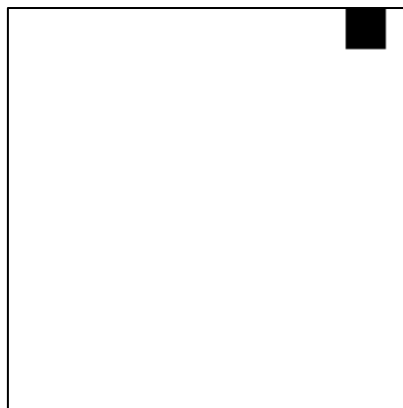
      //每執行 setInterval，就將 position 位置加 1
      position++;

      //若是大於 200（畫布的寬度），則從 0 開始
      if(position > 200){
        position = 0;
      }
    }, 30);
```

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



(圖) 清空畫布後，進行繪製



(圖) 每一次執行，都會改變繪製時 x 的位置，感覺圖形好像往右移動

範例 23-2-2.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
```

```
let ctx = canvas.getContext("2d");

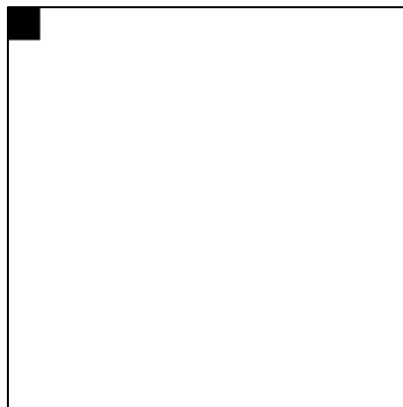
//方塊尺寸初始值
let size = 0;

//設定每 30 毫秒執行一次
setInterval(function(){
    //清空畫布
    ctx.clearRect(0, 0, 200, 200);

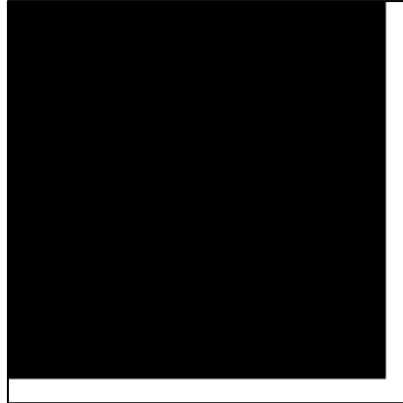
    //繪製矩形
    ctx.fillRect(0, 0, size, size);

    //每執行 setInterval，就將 size 大小加 1
    size++;

    //若是大於 200（畫布的寬度），則從 0 開始
    if(size > 200){
        size = 0;
    }
}, 30);
</script>
</body>
</html>
```



(圖) 圖形會從角落開始慢慢放大



(圖) 放大到邊界的時候，就會從頭開始

範例 23-2-3.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //定義畫「實心圓形」還是畫「圓線」的函式
    let circle = function(x, y, radius, fillCircle){
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2, false);

      //true 就填滿，false 就畫線
      if(fillCircle) {
        ctx.fill();
      } else {
        ctx.stroke();
      }
    }
  </script>
</body>
</html>
```

```

//繪製蜜蜂的函式
let drawBee = function(x, y){
    ctx.lineWidth = 2;

    //畫框線的顏色
    ctx.strokeStyle = 'black';

    //畫實心圓的顏色
    ctx.fillStyle = 'gold';

    //畫黃色實心圓
    circle(x, y, 8, true);

    //畫黑色圓圈，當作黃色實心圓的外框
    circle(x, y, 8, false);

    //畫左上角的圓圈當耳朵
    circle(x - 5, y - 11, 5, false);

    //畫右上角的圓圈當耳朵
    circle(x + 5, y - 11, 5, false);

    //畫小小黑色圓，放在黃色實心圓中間靠左
    circle(x - 2, y - 1, 2, false);

    //畫小小黑色圓，放在黃色實心圓中間靠右
    circle(x + 2, y - 1, 2, false);
}

//隨機更新蜜蜂的位置
let update = function(position){
    //介於 -2 到 1 之間
    let offset = Math.random() * 4 - 2;

    //每次 update() 就從當前的 position 偏移一下
    position += offset;

    //超出右或下邊界，就回到邊界上

```

```
        if(position > 200){
            position = 200;
        }

        //超出左或上邊界，就回到邊界上
        if(position < 0){
            position = 0
        }

        //將計算過的 position 回傳
        return position;
    }

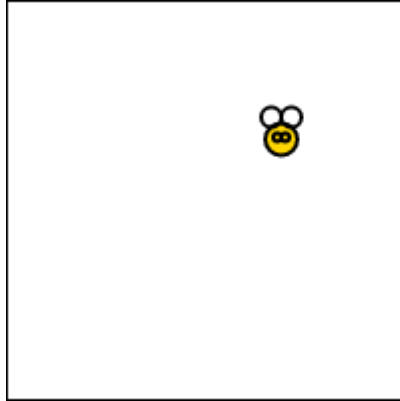
    //主程式區域
    let x = 100;
    let y = 100;

    //設定每 30 毫秒執行一次
    setInterval(function(){
        //清空畫布
        ctx.clearRect(0, 0, 200, 200);

        //畫蜜蜂。初始座標為 (100, 100)
        drawBee(x, y);

        //改變當前座標 x,y 的值
        x = update(x);
        y = update(y);

        //在畫布上繪製邊框，確認蜜蜂是否飛出界
        ctx.strokeRect(0, 0, 200, 200);
    }, 30);
</script>
</body>
</html>
```



(圖) 不斷在飛行的蜜蜂

23-3 碰撞偵測

整合畫圖函式與物件導向的操作方式，讓圖形有移動的效果外，還能藉由判斷 x, y 是否為邊界的值，來將 x 改成 $-x$ ，或是 $-y$ 改成 y ；相對地， $-x$ 會被改成 x ， y 也會被改成 $-y$ ，來進行碰撞偵測。

範例 23-3-1.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Canvas</title>
</head>
<body>
  <canvas id="canvas" width="200" height="200"></canvas>

  <script>
    let canvas = document.getElementById("canvas");
    let ctx = canvas.getContext("2d");

    //定義畫「實心圓形」還是畫「圓線」的函式
    let circle = function(x, y, radius, fillCircle){
      ctx.beginPath();
      ctx.arc(x, y, radius, 0, Math.PI * 2, false);

      //true 就填滿，false 就畫線
```

```

        if(fillCircle) {
            ctx.fill(); //填滿
        } else {
            ctx.stroke(); //畫線
        }
    }
}

//Ball 建構子
let Ball = function(){
    //自訂屬性
    this.x = 100;
    this.y = 100;
    this.xSpeed = -2;
    this.ySpeed = 3;
};

//繪製球
Ball.prototype.draw = function(){
    //畫圖的函式
    circle(this.x, this.y, 2, true);
};

//移動球
Ball.prototype.move = function(){
    /**
     * 在碰撞檢查的時候，透過 xSpeed、ySpeed 的正負值改變，
     * 來決定當前 x 或 y 應該往正或負的值移動
     */
    this.x += this.xSpeed;
    this.y += this.ySpeed;
};

//碰撞檢查
Ball.prototype.checkCollision = function(){
    //如果 this.x < 0，代表碰到左邊界；如果 this.x > 200，代表碰到右邊界
    if(this.x < 0 || this.x > 200){
        this.xSpeed = -this.xSpeed;
    }
}

```



```

    }

    //如果 this.y < 0，代表碰到上邊界；如果 this.y > 200，代表碰到下邊界
    if(this.y < 0 || this.y > 200){
        this.ySpeed = -this.ySpeed;
    }
};

//實體化一個物件
let ball = new Ball();

//設定每 30 毫秒執行一次
setInterval(function(){
    //清空畫布
    ctx.clearRect(0, 0, 200, 200);

    //繪製球
    ball.draw();

    //移動球
    ball.move();

    //碰撞檢查
    ball.checkCollision();

    //在畫布上繪製邊框，確認球是否飛出界
    ctx.strokeRect(0, 0, 200, 200);
}, 30);

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

```



(圖) 球會不斷斜向移動，碰到邊界會反彈