

JavaScript 重要基礎

彭哥^先讀：

https://www.youtube.com/watch?v=xbJPFMCiLHM&list=PL-g0fdC5RMbqW54tWQPIVbhyl_Ky6a2VI&index=8

目錄

- ❖ FlexBox (p. 4)
- ❖ Responsive Web Design(RWD) (p. 8)
- ❖ JavaScript 基本語法 (p. 13)
- ❖ HTML DOM (p. 19)
- ❖ AJAX 網路連線 (p. 29)
- ❖ Arrow Function (p. 35)
- ❖ 解構賦值 (p. 38)
- ❖ 其餘運算符號 Rest Operator (p. 42)
- ❖ 模組基礎 Module (p. 46)
- ❖ 模組輸出 / 輸入 (export/import) (p. 53)
- ❖ Proxy 代理物件基礎 (p. 59)
- ❖ JavaScript 物件的拷貝 (p. 68)

基礎提醒

- ❖ 只有 block 類型元件(會自動換行的那種)可以設定長、寬
 - ❖ <div> 可以
 - ❖ 不行
- ❖ <div> 基本上是切版的重要工具，很多設定都是在此標籤上做
- ❖ 容器：父元素；項目：子元素
- ❖ 子元素的長寬預設基本都是隨父元素變化
- ❖ <script> 標籤通常寫在 <body> 的最下面
- ❖ Javascript 的空值有 undefined(真 . 空) 、 null(有個東西是空)

```
const hobbies = ["Sports", "Cooking"];  
  
for (const hobby of hobbies) {  
    console.log(hobby);  
}
```



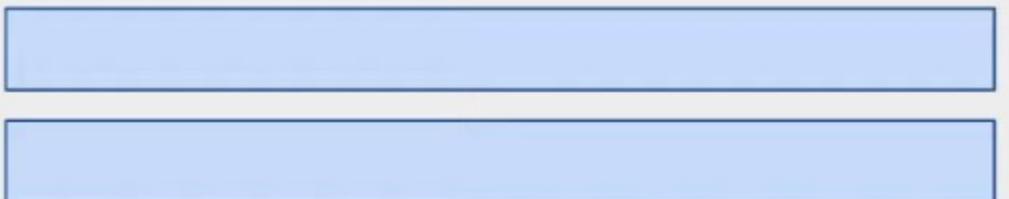
FlexBox

元件的水平排列

(在父元素設置，子元素生效)

https://www.youtube.com/watch?v=xbJPFMCiLHM&list=PL-g0fdC5RMbqW54tWQPIVbhy1_Ky6a2VI&index=8

display:block



display:flex



固定寬度 · 各 50% 的配置

display:flex

flex:none;
width:50%

flex:none;
width:50%

左邊固定 100px · 右邊彈性縮放配置

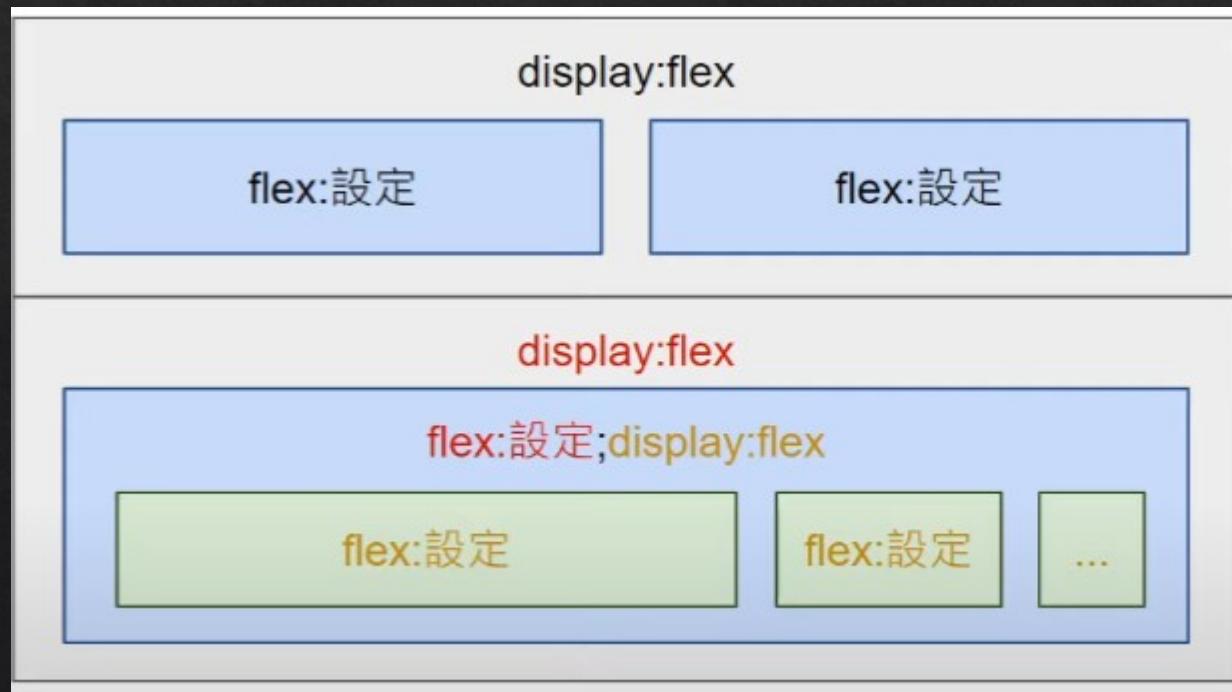
display:flex

flex:none;
width:100px

flex:auto

多層次結構

- ❖ 每個層次要獨立看待。
- ❖ `display: flex` → 在父元素寫，設定子元素要水平排列
- ❖ `flex`: 設定 → 在子元素寫，設定自己(子元素本人)要怎麼排



對齊方式

水平靠右

display:flex;justify-content:flex-end

flex:none;
width:30%

flex:none;
width:30%

水平置中

display:flex;justify-content:center

flex:none;
width:30%

flex:none;
width:30%

水平對齊 :justify-content

垂直靠中

display:flex;align-items:center



垂直靠下

display:flex;align-items:flex-end



垂直對齊 :align-items

垂直靠上

display:flex;
justify-content:center;align-items:center

flex:none;
width:50%

完美置中



Responsive Web Design RWD

在不同大小螢幕都可以正常顯示

Media Query

- ❖ 根據螢幕寬度，調整 CSS 設定

- 基本語法

滿足螢幕條件，就
會套用下面設定

@media (螢幕條件){

套用符合條件時的 CSS 設定

}



- 簡單範例

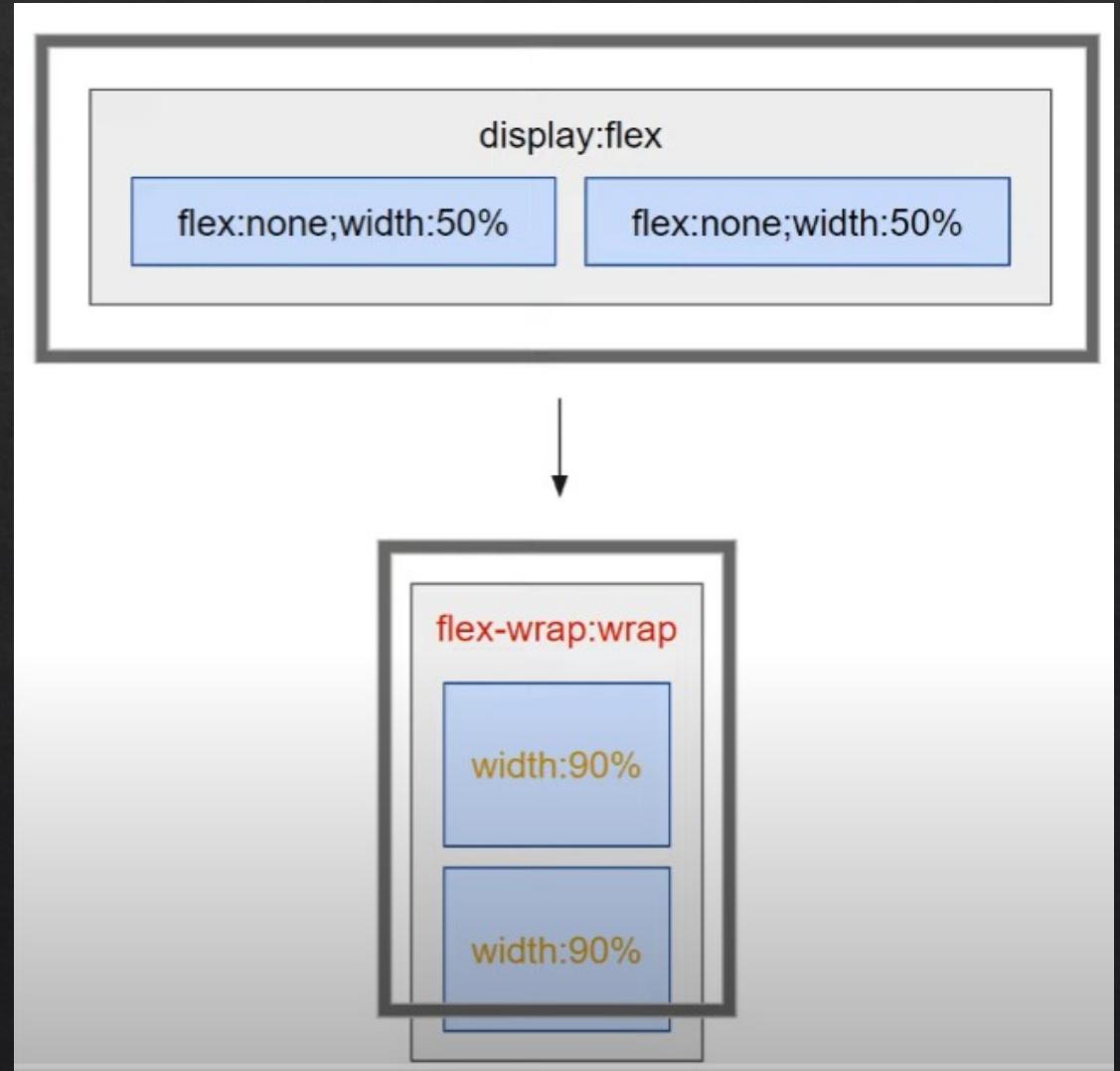
@media (max-width:1200px){

螢幕寬度在 1200px 以下

}

多個排 / 少量並排 / 單一排列

- ❖ 使用 flex-wrap + 寬度調整
 - ❖ Flex-wrap 才能使元素有機會往下掉
 - ❖ 右圖兩個 90% 會超過螢幕寬度，所以就自動變直向排列



控制內容是否換行

- HTML 程式碼

```
<div>大家好<br/>我是彭彭</div>
```

- CSS 換行

```
br{display:block}
```

- CSS 不換行

```
br{display:none}
```

寫好兩份，視情況切換

- ❖ 利用顯示和隱藏決定意1顯示的版面

- HTML 程式碼

```
<div class="desktop">電腦版</div>
```

```
<div class="mobile">手機版</div>
```

- CSS 的設定

```
.desktop{display:block}
```

```
.mobile{display:none}
```

```
media (max-width:500px){
```

// 螢幕寬度小於 500px



```
.desktop{display:none}
```

```
.mobile{display:block}
```

```
}
```



JavaScript 基本語法

惡補拉

基礎 (基本上跟 c++ 很像)

- ◆ 變數
 - ◊ let name;
- ◆ 常數
 - ◊ const x=3;
- ◆ 加減乘除 餘數 布林反運算 (!) 跟 python 差不多
- ◆ += 、 %= 、 ++ 、 -- 那些也可用
- ◆ 等於 : ==
- ◆ and: && 、 or: ||
- ◆ 要求使用者輸入 : num=prompt(" 請輸入一個數字" , " 預設值") // 「預設值」會出現在輸入框

函式

- ❖ 第一種寫法

- ❖ function add(num1, num2){
 result = num1+num2;
 return result;
}

- ❖ 把函數當作變數

- ❖ let add=function(num1, num2){
 result = num1+num2;
 return result;
}

- ❖ 兩種寫法呼叫的方式都一樣

物件 Object – 用來封裝其他資料！

- ❖ 函式在裡面稱為方法
- ❖ 變數稱為屬性

程式碼範例

```
let obj=new Object();  
  
obj.x=3;  
  
obj.y=4;  
  
obj.show=function(){  
    console.log(this.x, this.y);  
};
```

用 JSON 建立物件的簡單寫法

◆ 這樣寫更快 ~

程式碼範例

```
let obj1={};  
  
let obj2={  
    x:3,  
    y:4,  
    show:function(){  
        console.log(this.x, this.y);  
    }  
};
```

陣列

建立空白陣列，可放進變數中做後續操作

`new Array()` 或 `[]`

程式碼範例

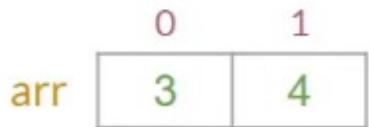
`let arr=[];` 傾向使用這個

使用陣列物件的 `push` 方法

`陣列.push(資料)`

程式碼範例

```
let arr=[];
arr.push(3);
arr.push(4);
```





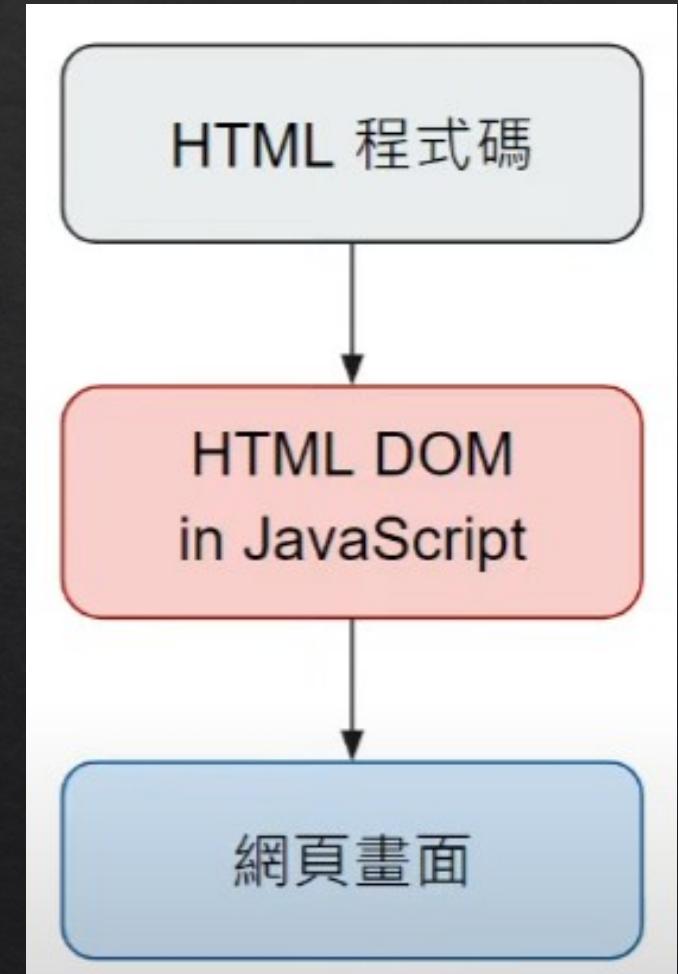
HTML DOM

根據 HTML 標籤，建立的物件結構

https://www.youtube.com/watch?v=8OejlO7N_vw&list=PL-g0fdC5RMbqW54tWQPIVbhyl_Ky6a2VI&index=22

- 每個 HTML 標籤在 JavaScript 引擎中都有對應的**標籤物件**，稱為 **HTML Element**。
- 把 HTML **標籤物件**，串接在一起，成為物件結構，就是 **HTML DOM**。
- **HTML DOM** (Document Object Model)。

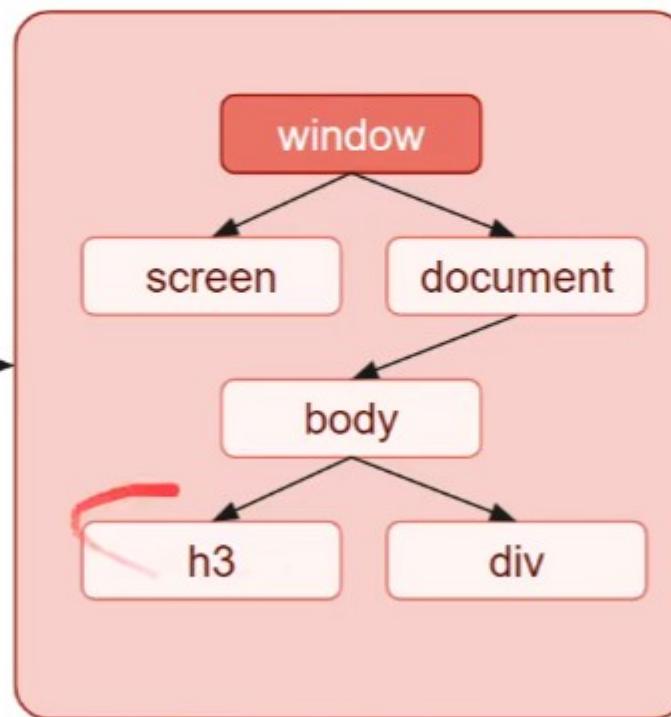
例如 <div> 標籤在 javascript 終究會有對應的 div HTML Element



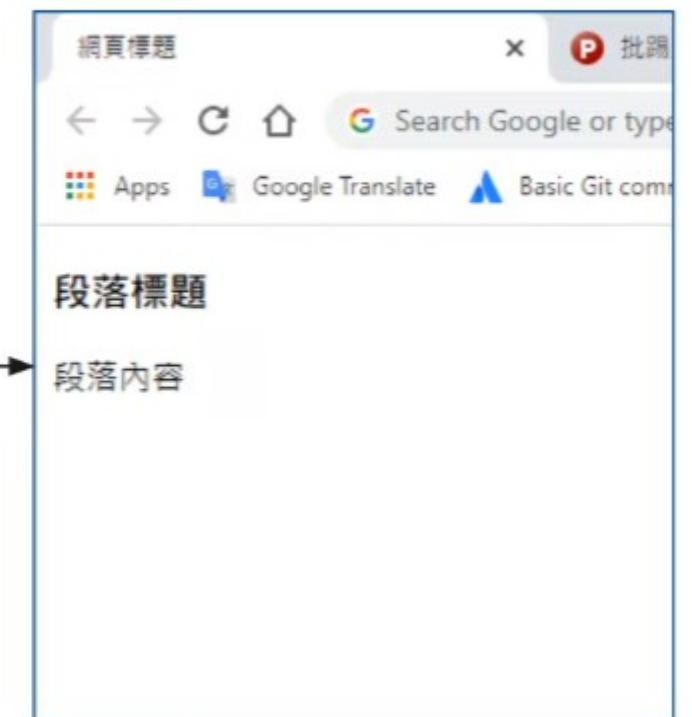
HTML 程式碼

```
<!DOCTYPE html>
<html>
<head>
    <title>網頁標題</title>
</head>
<body>
    <h3>段落標題</h3>
    <div>段落內容</div>
</body>
</html>
```

HTML DOM in JavaScript



網頁畫面



瀏覽器內建的 window 物件 (1/2)

操作 window 物件

- 是 HTML DOM 的入口

- 隨時可以用內建的 window 變數取得
`console.log(window)`
- window 物件包含有用的屬性
`window.innerWidth`
`window.innerHeight`
- window 物件包含有用的方法
`window.prompt("輸入資料", "預設值")`
`window.alert("彈出警告視窗")`

操作 screen 物件

- 紀錄使用者螢幕資訊

- screen 物件是 window 物件的一個屬性
`window.screen`
- screen 物件包含有用的屬性
`window.screen.width`
`window.screen.height`

瀏覽器內建的 window 物件 (2/2)

- document 物件是 window 物件的一個屬性
`window.document`
可簡寫為 `document`
- document 物件包含有用的屬性
`document.title`
`document.body` 就是我們用的 `<body>`
- document 物件包含有用的方法
`document.querySelector("CSS 選擇器")`

操作 document 物件
- 代表網頁主畫面

- body 物件是 document 物件的一個屬性
`document.body`
- body 物件包含有用的屬性
`document.body.innerHTML`
`document.body.className`
`document.body.id`

取得標籤物件

- 第一步：在 HTML 想要操作的標籤加上 id 屬性

```
<div id="content">這是一段字</div>
```

```
<span id="keyword">這是一段字</span>
```

- 第二步：在 JavaScript 程式中利用

```
document.querySelector() 方法取得標籤物件
```

```
let divElement=document.querySelector("#content")
```

```
let spanElement=document.querySelector("#keyword")
```

- ❖ 使用 id 屬性建立連結
- ❖ document.querySelector()

- 第一步：先取得標籤物件

```
<div id="content">這是一段字</div>
```

```
<script>
```

```
let divElement=document.querySelector("#content")
```

```
</script>
```

- 第二步：操作標籤物件的 HTML 和 CSS

```
<div id="content">這是一段字</div>
```

```
<script>
```

```
let divElement=document.querySelector("#content")
```

```
divElement.innerHTML="這是新的字";
```

```
divElement.className="welcome";
```

```
divElement.style.fontSize="30px";
```

```
divElement.style.color="blue";
```

```
</script>
```

藍色 : html 屬性

黃色 : CSS 屬性

使用者點擊

- 在希望被點擊的標籤加上 onclick 屬性

```
<div id="content">這是一段字</div>
<button onclick="change();">點我</button>
<script>
function change(){
    let divElm=document.querySelector("#content")
    divElm.innerHTML="這是新的字";
    divElm.style.color="blue"; // 變藍色
}
</script>
```

事件種類

- 以下常見的事件種類
 - click 滑鼠點擊
 - mouseover 滑鼠移入
 - mouseout 滑鼠移出
 - mousedown 滑鼠按下
 - mouseup 滑鼠放開

註冊事件處理函式

- 基本語法

```
<div 事件名稱="程式碼">標籤內文</div>
    .
<button 事件名稱="程式碼">標籤內文</button>
```

- 範例程式

```
<div onclick="change();>點擊</div>
<script>
    function change(){
        document.body.innerHTML="Hello";
    }
</script>
```

- 點擊改變標籤本身的內文

◆ 這裡的 this 表示 <div>

```
<div onclick="change(this);>原本的內文</div>
```

```
<script>
```

```
function change(elem){
```

```
    elem.innerHTML="新的內文";
```

```
}
```

```
</script>
```

- 在某標籤上，將函式與事件作對應
- this 代表觸發事件的標籤

事件處理 - 多個事件搭配使用範例

- 滑鼠按住、放開同時運作的範例程式

```
<button  
    onmousedown="down(this);"  
    •  
    onmouseup="up(this);"  
>按鈕</button>  
  
<script>  
    function down(elem){  
        elem.style.color="red";  
    }  
    function up(elem){  
        elem.style.color="black";  
    }  
</script>
```



AJAX 網路連線

https://www.youtube.com/watch?v=6X8sDGFRss&list=PL-g0fdC5RMbqW54tWQPIVbhyl_Ky6a2VI&index=26

網址組成

通訊協定://主機名稱/路徑

- 通訊協定 (Protocol)
- 主機名稱 (Hostname)
- 路徑 (Path)

前後端互動基礎

- 實際的情境如下：
 - 在瀏覽器網址列上輸入網址，按下 Enter 取得網頁
 - 可以採用 **AJAX / XHR** 技術執行網址連線、取資料的動作
- ◆ 用 javascript 網頁前端的 **AJAX / XHR** 技術作網址連線、取資料



AJAX / XHR

利用 JavaScript 程式進行網路連線



- 初期稱作 **Asynchronous JavaScript And XML** 技術
- 中期使用 **XMLHttpRequest** 物件，稱為 **XHR** 技術
- 近年，建議採用最新的 **fetch** 函式來執行網路連線功能

fetch 函式 (javascript 內建的連線用函式)

- 基本語法詳解

```
fetch(網址).then(function(回應物件){  
    console.log(回應物件);  
});
```

- 取得純文字的回應

```
fetch(網址).then(function(response){  
    return response.text();  
}).then(function(data){  Promise 觀念  
    console.log(data); // 純文字資料  
});
```

- 取得 JSON 格式的回應

```
fetch(網址).then(function(response){  
    return response.json();  
}).then(function(data){  
    console.log(data); // JSON 格式資料  
});
```

```
function getData(){
    // 利用 fetch 進行連線並取得資料
    fetch("https://cwpeng.github.io/live-records-samples/data/products.json").then(function(response){
        return response.json();
    }).then(function(data){
        // 已經取得資料，把資料呈現到畫面上
        let result=document.querySelector("#result");
        result.innerHTML="";
        // 先把畫面清空
        for(let i=0;i<data.length;i++){
            let product=data[i];
            result.innerHTML+="

"+product.name+","+product.price+","+product.description+"

";
        }
    });
}
```



Arrow Function

https://www.youtube.com/watch?v=GzrWyJkD3b8&list=PL-g0fdC5RMbqW54tWQPIVbhyl_Ky6a2VI&index=27

箭頭函式語法

函式的另外一種寫法

省去 function 關鍵字 · 用 => 取代

•

- 普通函式語法一

```
function 函式名稱(函式參數列表){  
    函式區塊內部的程式碼  
    return 回傳值;  
}
```

- 普通函式語法二

```
let 函式名稱=function(函式參數列表){  
    函式區塊內部的程式碼  
    return 回傳值;  
};
```

- 箭頭函式基本語法

```
let 函式名稱=(函式參數列表)=>{  
    函式區塊內部的程式碼  
    return 回傳值;  
};
```

- 普通函式寫法一

```
function add(n1, n2){  
    let result=n1+n2;  
    return result;  
}
```

- 普通函式寫法二

```
let add=function(n1, n2){  
    let result=n1+n2;  
    return result;  
};
```

- 箭頭函式基本寫法

```
let add=(n1, n2)=>{  
    let result=n1+n2;  
    return result;  
};
```

箭頭函式簡化語法

函式本體只有回傳值敘述
沒有其他指令時，可簡化

- 可簡化的箭頭函式

```
let 函式名稱=(函式參數列表)=>{
    return 回傳值;
};
```

- 箭頭函式簡化的語法

```
let 函式名稱=(函式參數列表)=>(回傳值);
```

箭頭函式 簡化程式碼範例

若函式中包含其他指令，無法簡化

- 無法簡化

```
let add=(n1, n2)=>{
    let result=n1+n2;
    return result;
};
```

- 可以簡化

```
let add=(n1, n2)=>{
    return n1+n2;
};
```

- 箭頭函式簡化後的寫法

```
let add=(n1, n2)=>(n1+n2);
```



解構賦值

Destructuring Assignment

https://www.youtube.com/watch?v=AMwRSPPh2G3U&list=PL-g0fdC5RMbqW54tWQPIVbhyl_Ky6a2VI&index=28

先解構，再賦值

- 解構：把陣列或物件中的資料拆開
- 賦值：將拆開的資料分別放入個別的變數中

陣列的解構賦值

按照出現順序作對應
使用中括號

```
// 變數資料交換
let n1=3;
let n2=4;
[n2, n1]=[n1, n2];
console.log(n1, n2);
```

應用例子

- 傳統的做法
 - let arr=[3, 4, 5];
 - let d1=arr[0];
 - let d2=arr[1];
 - let d3=arr[2];
- 解構賦值的語法
 - let arr=[3, 4, 5];
 - let [d1, d2, d3]=arr;
- 宣告與賦值分開
 - let arr=[3, 4, 5];
 - let d1, d2, d3;
 - [d1, d2, d3]=arr;
- 預設值
 - let arr=[3, 4];
 - let d1, d2, d3;
 - [d1, d2=2, d3=5]=arr;

3,4,5 分別放入 d1, d2, d3

可以有預設值

物件的解構賦值

使用花括號

基本操作

將物件中的資料分開賦值給變數
按照物件成員名稱做對應

```
let x, y, z;  
({x, y, z=10}=obj);  
console.log(x, y, z);
```

```
let newX, newY, newZ;  
({x:newX, y:newY, z:newZ=10}=obj);  
console.log(newX, newY, newZ);
```

解構到 x，並賦值給 newX

- 傳統的做法
- 解構賦值的語法

```
let obj={x:3, y:4};
```

```
let x=obj.x;
```

```
let y=obj.y;
```

```
let obj={x:3, y:4};
```

```
let {x, y}=obj;
```

應用例子

- 宣告與賦值分開

```
let obj={x:3, y:4};
```

```
let x, y;
```

({x, y}=obj); // 不和宣告一起執行，要多加()

不然會出現

- 預設值

```
let obj={x:3};
```

```
let x, y;
```

```
({x, y=5}=obj);
```

- 指定新的變數名稱

```
let obj={x:3, y:4};
```

```
let newX, newY;
```

```
({x:newX, y:newY}=obj);
```

在函式的應用

```
// 統整函式的物件參數  
function add(args){  
| console.log(args.n1+args.n2);  
}  
add({n1:3, n2:4});
```



```
// 統整函式的物件參數  
function add({n1, n2} ){  
| console.log(n1+n2);  
}  
add({n1:3, n2:4});
```

更好 !

```
const [ state, setState ] =  
  React.useState(null);
```

React 中 Hook 的寫法就是利用解構賦值完成



其餘運算符號

Rest Operator

把剩餘的東西包在一起

常常跟上一節的解構賦值用在一起

- 運算符號，三個點 ...
- 運用在解構賦值
 - 陣列：把剩餘的資料包在一個新陣列中
 - 物件：把剩餘的成員包在一個新物件中
- 運用在函式參數
 - 把剩餘的參數包在一個新陣列中

在陣列 / 物件解構賦值中應用

未逐一指定名稱的剩餘值
運用其餘運算符號，包在新陣列中

- 陣列解構賦值的運用

```
let arr=[3, 4, 5, 6, 2];
let [d1, d2, ...data]=arr;
console.log(data); // [5, 6, 2]
```

- 限制：必須放在最後面

```
let arr=[3, 4, 5, 6, 2];
let [d1, ...data, d2]=arr; // error
```

未逐一指定名稱的剩餘值
運用其餘運算符號，包在新物件中

- 物件解構賦值的運用

```
let obj={x:3, y:4, z:5};
let {x, ...data}=obj;
console.log(data); // {y:4, z:5}
```

- 限制：必須放在最後面

```
let obj={x:3, y:4, z:5};
let {...data, x}=obj; // error
```

用在函式參數中

未逐一指定參數名稱的參數資料
運用其餘運算符號，包在陣列中

- 其餘參數的運用

```
function test(a, b, ...data){  
    console.log(a, b); // 3 4  
    console.log(data); // [1, 2, 5]
```

```
}
```

```
test(3, 4, 1, 2, 5);
```

- 限制：必須放在最後面

```
function test(a, ...data, b){ // error  
    console.log(a, b);  
    console.log(data);  
}
```

實例

```
let [n1, n2, ...data]=[3, 2, 1, 5, 4, 0];  
console.log(n1, n2, data);
```

```
3 2 ▼ (4) [1, 5, 4, 0] ⓘ  
 0: 1  
 1: 5  
 2: 4  
 3: 0  
length: 4  
► [[Prototype]]: Array(0)
```

data 會是陣列

```
let {x, y, ...obj}={x:3, y:4, z:5, a:1, b:2};  
console.log(x, y, obj);
```

同理 · obj 會是物件
obj.x 為 3

```
const user = {  
  name: "Max",  
  age: 34  
};  
const newHobbies = ["Reading"];
```

```
const extendedUser = {  
  isAdmin: true,  
  ...user  
};
```

```
console.log(extendedUser);
```

```
▼ {isAdmin: true, na  
me: "Max", age: 34}  
  isAdmin: true  
  name: "Max"  
  age: 34
```



模組基礎 Module

單一程式切割為多個檔案

每個程式檔案稱為模組

(就像 python 自己寫的 .py 可以 import 進來)

vscode 建議安裝 : live server

簡介

- 早期的前端 JavaScript 程式沒有良好的模組支援，可使用 Webpack 解決。
- 現代的前端 JavaScript 擁有完整的模組能力，支援獨立的模組名稱空間，以及 `export/import` 語法。

使用 JavaScript 原生模組系統

要設定 type="module"
並且指定 src

- 檔案名稱 `lib.js`

```
function echo(msg){  
    console.log(msg);  
}
```

- 檔案名稱 `main.js`

```
let name="我又大又肥";  
echo(name);
```

- HTML 中引入主要的程式檔案

```
<script type="module" src="main.js"></script>
```

建立輸出輸入的關聯性

- ❖ 使用 export 輸出 /import 輸入
- ❖ export default <東東>
 - <東東>不一定要是函式，可以是變數

```
JS main.js x ...
1 // import 資料 from "模組檔案的路徑";
2 import x from "./lib.js";
3 console.log(x);
4 /*
5 let name="我又大又肥";
6 echo(name);
7 */
```

```
JS lib.js x ...
1 function echo(msg){
2   console.log(msg);
3 }
4 let x=3;
5 // export default 資料;
6 export default x;
```

- ❖ export 多個物件

```
JS main.js x ...
1 // import 資料 from "模組檔案的路徑";
2 import lib from "./lib.js";
3 console.log(lib);
4 lib.echo("我是彭彭");
5 lib.add(3, 4);
6 /*
7 let name="我又大又肥";
8 echo(name);
9 */
```

```
JS lib.js x ...
1 function echo(msg){
2   console.log(msg);
3 }
4 function add(n1, n2){
5   console.log(n1+n2);
6 }
7 // export default 資料;
8 export default {
9   echo:echo,
10  add:add
11 };
```

- 檔案名稱 lib.js

```
function echo(msg){  
  console.log(msg);  
}  
  
export default echo; // export default 資料
```

- 檔案名稱 main.js

```
// import 資料 from "模組檔案路徑"  
import echo from "./lib.js";  
  
let name="我又大又肥";  
  
echo(name);
```

- HTML 中引入主要的程式檔案

```
<script type="module" src="main.js"></script>
```

模組的獨立性

- ❖ 模組有各自空間，互不影響
- ❖ 模組中的 name 變數與 其他模組中的 name 不影響
- ❖ 但 default 的名子就要注意，不要撞名

- 檔案名稱 lib.js

```
function echo(msg){  
    console.log(msg);  
}  
  
let name="我是彭彭";  
echo(name);  
  
export default echo;
```

- 檔案名稱 main.js

```
import echo from "./lib.js";  
  
let name="我又大又肥";  
echo(name);
```

實際例子 (1/2)

The screenshot shows a code editor with three tabs:

- modules.html**: An HTML file containing the following code:

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta name="viewport" content="width=device-width, initial-scale=1">
5     <meta charset="utf-8" />
6     <title>模組基礎 Modules</title>
7   </head>
8   <body>
9     <script src="lib.js"></script>
10    <script src="main.js"></script>
11  </body>
12 </html>
```

The lines `<script src="lib.js"></script>` and `<script src="main.js"></script>` are highlighted with a red box.
- main.js**: A JavaScript file containing:

```
1 let name="我又大又肥";
2 echo(name);
```
- lib.js**: A JavaScript file containing:

```
1 function echo(msg){
2   console.log(msg);
3 }
```

- ❖ 上面這樣可以跑，但 main 跟 lib 本質上是同源的 (所以會互相影響)
- ❖ 若在 lib.js 中增加 `let name="JJ"`，會出現衝突
- ❖ 這種引入方式不好！

實際例子 (2/2)

The screenshot shows a code editor with three tabs:

- modules.html**: An HTML file containing a script tag with type="module" and src="main.js".
- main.js**: A JavaScript file with three lines of code: import echo from "./lib.js";, let name="我又大又肥";, and echo(name);.
- lib.js**: A JavaScript file with four lines of code: function echo(msg){, console.log(msg);, }, and export default echo;

The code in the main.js and lib.js files is highlighted with red boxes around the imports and exports.

```
modules.html
1  <!DOCTYPE html>
2  <html>
3  | <head>
4  | | <meta name="viewport" content="width=device-width">
5  | | <meta charset="utf-8" />
6  | | <title>模組基礎 Modules</title>
7  | </head>
8  <body>
9  | | <script type="module" src="main.js"></script>
10 </body>
11 </html>
```

```
main.js
1 import echo from "./lib.js";
2 let name="我又大又肥";
3 echo(name);
```

```
lib.js
1 function echo(msg){
2   console.log(msg);
3 }
4 export default echo;
```

❖ 這種方式較佳 ~



模組輸出 / 輸入

分為 預設 (default) 、非預設

```
let x = 3;  
export default x; // export default 3;  
import y from "./libs.js"; // 還是能取到 3
```

```
let x = 3;  
export {x}; // export let x = 3;  
import {x} from "./libs.js";
```

預設

- ❖ 輸出時使用 `default` 關鍵字
- ❖ 輸入時可使用任意名稱（但不建議）

- 預設的輸出語法

```
export default 變數名稱;
```

- 預設的輸入語法

```
import 變數名稱 from "模組檔案路徑";
```

這點和 python 不一樣ㄛ

- 模組檔案名稱 `lib.js`

```
let x=3;
```

```
export default x;
```

- 主程式檔案名稱 `main.js`

```
import y from "./lib.js";
```

```
console.log(y);
```

非預設

使用大括號包裹輸出 / 輸入的變數名稱
輸出和輸入的名稱必須相同

- 輸出語法

export {變數名稱, 變數名稱, ...};

- 輸入語法

import {變數名稱, 變數名稱, ...}

from "模組檔案路徑";

非預設在 export 跟 import 都要用 {} 包，且 export 跟 import 使用的名稱需要相同

* 在 export 物件 (object) 時也會用到 {}，但若前面出現 default 表示他用的還是預設值，所以 import 時不需要 {} (詳見 p. 48)

可同時使用預設 / 非預設變數

- 模組檔案名稱 lib.js

```
let x=3;  
let obj={x:3, y:4};  
let data=[5, 6, 7];  
export default x;  
export {obj, data};
```

- 主程式檔案名稱 main.js

```
import x from "./lib.js";  
import {obj, data} from "./lib.js";  
console.log(x);  
console.log(obj, data);
```

```
// import apiKey from "./util.js";  
import { apiKey, abc as content } from "./util.js";  
// import * as util from "./util.js";
```

一些使用例
子

```
export default (userName, message) => {  
    console.log('Hello');  
    return userName + message;  
}
```

也可以傳 function

語法整合 - 範例

- 模組檔案名稱 lib.js

```
let x=3;
```

```
let obj={x:3, y:4};
```

```
let data=[5, 6, 7];
```

```
export {x as default, obj, data};
```

- 主程式檔案名稱 main.js

```
import x, {obj, data} from "./lib.js";
```

```
console.log(x);
```

```
console.log(obj, data);
```

實際範例

JS lib.js X

```
1 let add=function(n1,n2){  
2   console.log(n1+n2);  
3 };  
4 let multiply=function(n1,n2){  
5   console.log(n1*n2);  
6 };  
7 let math={  
8   add:add, multiply:multiply  
9 };  
10 export default math; 包成一個工具箱  
11 export {add, multiply};
```

以單個工具傳送

JS main.js X

```
1 import math from "./lib.js";  
2 math.add(3,4);  
3 math.multiply(-3,4);
```

JS main.js X

```
1 import {multiply} from "./lib.js";  
2 multiply(3, 4);  
3 multiply(-2, 2);  
4 |
```



Proxy 代理物件基礎

現代前端框架經常使用的核心技巧

簡介

- 用來「代理」目標物件：意思是「改變、中介」目標物件的基礎操作。
- 可能的用途：
 1. 取得經過加工處理的屬性資料
 2. 驗證物件的屬性資料
 3. 綁定物件的屬性資料和使用者介面

建立代理物件 → 使用代理物件

- Proxy 代理物件的建構式

`new Proxy(目標物件, 包裝處理函式的物件);`

- 建立 Proxy 代理物件

`let 目標物件=某個物件實體;`

`let ref=new Proxy(目標物件,{`

`get:function(目標物件, 屬性名稱){`

`return 回傳自定義的屬性資料;`

`}`

`});`

- 使用 Proxy 代理物件

`// 取得物件屬性時，執行上述的 get 函式並取得回傳值`

`console.log(ref.x);`

最基礎代理物件處理函式：

`get()`

入門範例 (1/3)

準備一個目標物件

就是一個普通的物件

- 使用 JSON 表達紀錄名字和姓氏的目標物件

```
let profile={  
    firstName:"小明",  
    lastName:"王"  
};
```

入門範例 (2/3)

潛在的處理需求

多個屬性組合後才能滿足的需求

- 使用 JSON 表達紀錄名字和姓氏的目標物件

```
let profile={  
    firstName:"小明",  
    lastName:"王"  
};
```

- 我想要取得完整的中文姓名

```
console.log(profile.lastName+profile.firstName)
```

入門範例 (3/3)

使用 Proxy

包裝屬性組合細節，提供簡要的介面

完整程式範例如下：

```
let profile={  
    firstName:"小明",  
    lastName:"王"  
};  
let proxy=new Proxy(profile,{  
    get:function(target,property){  
        if(property==="chineseName"){  
            return target.lastName+target.firstName;  
        }else{  
            return target[property];  
        }  
    }  
});  
console.log(proxy.chineseName); // 印出：王小明  
console.log(proxy.firstName); // 印出：小明  
          有可能只想要名字的情況
```

實際範例 (1/3)

```
// 建立代理物件  
let data={  
    price:100,  
    count:5  
};  
console.log(data.price*data.count);
```

原始需求：計算總價

看一下內容



```
// 建立代理物件  
let data={  
    price:100,  
    count:5  
};  
let proxy=new Proxy(data, {  
    get:function(target, property){ // 使用代理物件取得屬性資料的時候，會對應的函式  
        console.log("代理的目標物件", target); 輸出上面的 data  
        console.log("想要取得的屬性名稱", property); 輸出 abc  
        return "屬性對應的資料";  
    }  
});  
// 使用代理物件，取得物件的屬性資料  
console.log(proxy.abc); 不管傳過去的 property 是啥，反正  
會先呼叫上面的 get()
```

實際範例 (2/3)

```
// 建立代理物件
let data={
    price:100,
    count:5
};
let proxy=new Proxy(data, {
    get:function(target, property){ // 使用代理物件取得屬性資料的時候，會對應的函式
        return target.price*target.count;
    }
});
// 使用代理物件，取得物件的屬性資料
console.log(proxy.total);
```

可以獲得總價了，但有時候我還是會想要取得單價 price

```
console.log("總價", proxy.total);
console.log("單價", proxy.price);    但發現都會輸出 500
```

實際範例 (3/3)

```
// 建立代理物件
let data={
  price:100,
  count:5
};
let proxy=new Proxy(data, {
  get:function(target, property){ // 使用代理物件取得屬性資料的時候，會對應的函式
    if(property==="total"){ // 如果想要取得的屬性名稱是 total，做一個乘法的組合算出總價
      return target.price*target.count;
    }else{ // 如果是其他的屬性，就按照原來的目標物件做回應
      return target[property];
    }
  }
});
// 使用代理物件，取得物件的屬性資料
console.log("總價", proxy.total);
console.log("單價", proxy.price); 取得 100
```

若獲得其他屬性，就回傳原本對應的屬性回去！
(我只是代理，不是要替代掉原本的人)



JavaScript 物件的拷貝

淺拷貝 / 深拷貝

- 淺拷貝：只能複製到一層（內建拷貝方法多屬此種）
 - let b = {...a} 或是下面方式
 - Let b = Object.assign({}, a) // 陣列就用 Object.assign([], a)
- 深拷貝：可以複製整層
 - let str = JSON.stringify(a)
 - let b = JSON.parse(str)

拷貝搶先看

- 物件沒有被實際拷貝 / 複製的狀況

```
let a={x:3, y:4};
```

```
let b=a; // 沒有建立新物件，參考到同一個物件實體
```

```
b.x=5;
```

```
console.log(a.x); // 印出 5
```

- 物件的拷貝 / 複製

```
let a={x:3, y:4};
```

```
let b={...a}; // 建立新物件，並複製原物件的資料
```

```
b.x=5;
```

```
console.log(a.x); // 印出 3
```

python 中也有一樣情形，物件的 assign 就像指標一樣，會將箭頭指到同一個記憶體，若另一個指標操作，會影響到原本的東西

用其餘運算子，並重新賦值給 b

淺拷貝

僅拷貝物件的第一層資料

- 物件 / 陣列中包含其他物件 / 陣列，產生層次
- 常見的拷貝工具都是淺拷貝

```
let a={x:3, y:4, data:[1, 2, 3]};  
// 使用其餘運算符號 ... 拷貝  
  
let b={...a};  
// 使用內建方法 Object.assign() 拷貝  
  
let c=Object.assign({}, a);
```

data 就是兩層

淺拷貝的特性

第二層以上的資料不會真的拷貝
參考到和原來相同的物件或陣列實體

- 淺拷貝特性範例說明

```
let a={x:3, y:4, data:[1, 2, 3];  
// data 中的陣列沒有真的被拷貝  
// a 和 b 物件中的 data 仍參考到同一個陣列實體  
let b={...a};  
b.data[0]=4; // b 對 data 的操作影響到 a 的 data  
console.log(a.data[0]); // 印出 4
```

深拷貝

完全拷貝物件底下所有層次的資料

- 使用 `JSON.stringify()` 將物件字串化 (Serialize)
- 使用 `JSON.parse()` 根據字串化的資料重新建立物件結構，完成深拷貝

```
let a={x:3, y:4, data:[1, 2, 3]};
```

// 將物件結構字串化

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

// 根據字串化的資料 重新建立物件結構

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

深拷貝的特性

建立兩個完全獨立的物件實體

- 深拷貝特性範例說明

```
let a={x:3, y:4, data:[1, 2, 3]};  
let str=JSON.stringify(a);  
let b=JSON.parse(str);  
// 以上完成深拷貝，兩個物件完全獨立不互相影響  
b.data[0]=4; // b 對 data 的操作不影響 a 的 data  
console.log(a.data[0]); // 印出 1
```

JSON 方法的限制

- 無法拷貝函式、Symbol 等等資料。

無法拷貝不能字串化 (Serialize) 的資料

```
let a={x:3, y:4, test:function(){  
    console.log("Hello");  
}};  
  
let str=JSON.stringify(a);  
  
let b=JSON.parse(str);  
// test 是函式，無法拷貝，遺失  
b.test(); // 錯誤
```

留言區討論

@user-lh7pb5he7w 🌟 11 個月前

感謝帶一次理解對於淺拷貝與深拷貝有進一步觀念~~

了解運作原理 想請教這會是運用在真正實作專案中的哪裏 目前我想到的可能是 例如是一個人事薪資系統 要計算人員出勤狀況 並且計算薪水 有分為A出勤的頁面 與 B計薪的頁面 通常 A頁面以原資料進行統計 B頁面可能有套用薪水公式規則去打印報表 在這種Case中 A頁面跳轉B頁面不希望資料跳回來被異動過 就會考慮到用深拷貝的情況將資料完整複製一份到 B頁面座使用嗎？

1 回覆

• 3 則回覆

@cwpeng-course 11 個月前

通常 Single Page Application 比較有機會用到，因為單一頁面的應用對於資料狀態的管理比較講究，例如 React 的狀態更新經常會要求完整的拷貝物件，會讓兩個狀態間的資料，比較不容易互相干擾。



補充

defer 、 type...

defer

- ❖ defer 用來確保導入的 javascript 腳本只在 HTML 文檔的其餘部分被讀取和解析後才執行
 - ❖ 白話 : html 用完了再開始看導入的 javascript
- ❖ 另一種 defer 的替代方案是把 <script> 移到 <body>

```
<script src="assets/scripts/app.js" defer></script>
```

type

- ◆ 確保這個 javascript 文件被視為 javascript 模塊
- ◆ 當 type="module" 後才能做 import export
- ◆ 這些東西在搭建 React 時應該會幫忙做

```
<script src="assets/scripts/app.js" type="module"></script>
```

Javascript 的 class

```
class User {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    greet() {  
        console.log('Hi!');  
    }  
}  
  
const user1 = new User("Manuel", 35);
```

Javascript 的 array 常用功能

```
const hobbies = ["Sports", "Cooking", "Reading"];
console.log(hobbies[0]);

hobbies.push("Working");
console.log(hobbies);

const index = hobbies.findIndex((item) => {
  return item === "Sports";
});
const index = hobbies.findIndex((item) => item === "Sports");

console.log(index);
```

Javascript 的 map

```
const hobbies = ["Sports", "Cooking", "Reading"];
const editedHobbies = hobbies.map((item) => item + "!");
```

```
▼ (4) ["Sports!", "Cooking!", "Reading!", "Working!"]
  0: "Sports!"
  1: "Cooking!"
  2: "Reading!"
  3: "Working!"
```

也可以返回一個物件，記得 {} 外面要多一層()

```
const editedHobbies = hobbies.map((item) => ({text: item}));
```

```
▼ (4) [Object, Object, Object, Object]
  ▼ 0: Object
    text: "Sports"
  ▼ 1: Object
    text: "Cooking"
  ▼ 2: Object
    text: "Reading"
  ▼ 3: Object
    text: "Working"
```