

陣列 (Array)

Eddy Chang

✉️ hello@eddychang.me



建立、頭尾加入新成員

```
// 陣列為有序資料集合
// 以"常數"來宣告文字字面，索引自0開始由左往右遞增0,1,2...
const aArray = []

// (★常用必會) `[]` <-- 1` 從後面加入新成員，原陣列 => [1]
aArray.push(1)

// `2 --> []` 從前面加入新成員，原陣列 => [2, 1]
aArray.unshift(2)
```

頭尾移除成員

```
const aArray = [1, 2, 3]

// `[] --> 3` 從後面移除成員，回傳3。原陣列 => [1, 2]
let popValue = aArray.pop()

// `1 <-- []` 從前面移除成員，回傳1。原陣列 => [2]
let shiftValue = aArray.shift()
```

存取成員

```
const aArray = [1, 2, 3]
```

// 用索引存取成員值(索引自0開始由左往右遞增0,1,2...)

```
const a = aArray[2]
```

// 對成員指定一個新值，也是使用方括號([])，使用索引來存取成員的值

```
aArray[2] = 99
```

// 解構賦(指定)值語法 (destructuring assignment)

// 類似鏡子的1對1映對作指定運算，把陣列中的資料解開擷取成為獨立變數

// 執行後結果`x=1, y=2, z=3`

```
const [x, y, z] = aArray
```

對每個成員處理 - for

★ 常用必會

範例：對每個成員乘 2，最後要得到 [2, 4, 6, 8]

使用 for 迴圈語法，自索引為 0 開始依序遞增處理

```
const aArray = [1, 2, 3, 4]

// for 迴圈
for (let i = 0; i < aArray.length; i++) {
  aArray[i] = aArray[i] * 2
}
```

對每個成員處理 - forEach

使用 forEach 迭代語法，由陣列呼叫開始，依次輪流進入其中的回調(呼)函式中處理

```
const aArray = [1, 2, 3, 4]

// forEach 迭代(iteration)
aArray.forEach(function (v, i, array) {
  // callback function(回調、回呼函式)
  // 每個成員的索引、值，會依序在這區塊裡面被得到，視情況運算執行
  array[i] = v * 2
})
```

對每個成員處理 - map

★ 常用必會

使用 map 迭代語法，由陣列呼叫開始，依次輪流進入其中的回調(呼)函式中處理，之後回傳處理過的值。map 會終會回傳一個新陣列。

```
const aArray = [1, 2, 3, 4]

// map 迭代(iteration)
const bArray = aArray.map(function (v, i, array) {
  return v * 2
})
```

callback 是什麼

回調或回呼函式 callback function，常簡稱為 callback，「是一個函式以傳入參數值的方式，傳遞到另一個函式中，準備接下來要繼續執行」

- 函式(function)在 JS 中有頭等函式與高等函式(HOF)的設計
- 與 JS 中一種程式碼編寫風格 CPS(延續傳遞風格) 有關，CPS 主要是用來解決阻塞，也就是要達成非阻塞(non-block)、可異步(asnyc)執行
- ! 並非所有的 callback 都是異步執行的

註：CPS - 延續傳遞風格(Continuation-passing style)

拷貝陣列成員(淺拷貝 shallow copy)

```
const aArray = [1, 2, 3, 4]  
  
// (★常用必會) 方式一：ES6 - 展開運算符  
const bArray = [...aArray]  
  
// 方式二：slice  
const cArray = aArray.slice()
```

組合多個陣列為一個陣列

```
const aArray = [1, 2, 3]
const bArray = [99, 100]

// 用展開運算子為最方便
const cArray = [...aArray, ...bArray]
```

清空陣列

```
// 方式一：使用length屬性(又稱truncate(截短)語法)
```

```
const aArray = [1, 2, 3, 4]  
aArray.length = 0
```

```
// 方式二：直接指定空白陣列
```

```
// ⚠️ 注意要用 let 告白才行  
let bArray = [7, 8, 9, 10]  
bArray = []
```

尋找成員

(★ 常用必會)

```
const aArray = [1, 2, 3, 4]

// find index，如果有找到回傳索引，沒找到會回傳 -1
const index = aArray.findIndex((v, i, array) => v === 2)

// find value，如果有找到回傳值，沒找到會回傳 undefined
const value = aArray.find((v, i, array) => v === 2)
```

註: 另有 `indexof` 方法，只能用來尋找一般基礎值(數字/字串/布林...)，所以用途很有限。

分割一個陣列為多個 - slice

語法: `slice(開始索引, [結束索引(不包含)])`

```
const aArray = [1, 2, 3, 4]
// slice(start, [end(不包含)]) 會回傳一個新陣列
// 結束索引沒給定時，會分割到最後一個。結束索引負數-1 代表從最後面算起第 1 個
// 得到[1, 2]
const bArray = aArray.slice(0, 2)
// 得到[2, 3]
const cArray = aArray.slice(1, -1)
// 得到[3, 4]
const dArray = aArray.slice(2)
```

從中移除成員 - filter

(★ 常用必會)

```
const aArray = [1, 2, 3, 4]
```

// 移除 index=1 -> 相當於新建立一個陣列不要有索引 1 的成員

```
const bArray = aArray.filter((v, i, array) => i !== 1)
```

// 移除 value=3 -> 相當於新建立一個陣列不要有值 3 的成員

```
const cArray = aArray.filter((v, i, array) => v !== 3)
```

從中移除成員 - splice

⚠️ 有副作用，會改變呼叫它的陣列

語法: `splice(要移除的成員索引, 1)`

```
const aArray = [1, 2, 3, 4]
// 移除自索引 2 的成員，第二傳入參數1代表移除一個
// 結果: `aArray = [1,2,4]`
aArray.splice(2, 1)
```

從中插入新成員 - slice

```
const aArray = [1, 2, 3, 4]

// 範例1：在索引 1 處插入新的成員 99，其它成員往後
// 先用slice切割出兩個陣列
const bArray = aArray.slice(0, 1)
const cArray = aArray.slice(1)

const newArray = [...bArray, 99, ...cArray]

// 範例2：在索引 1 處插入一個陣列的中的值
const newArray = [...bArray, ...[99, 100, 101], ...cArray]
```

從中插入新成員 - slice 函式

```
// insertItem(aArray, 1, 99)
const insertItem = (arr, index, newItem) => [
  ...arr.slice(0, index),
  newItem,
  ...arr.slice(index),
]

// insertItems(aArray, 1, [99, 100, 101])
const insertItems = (arr, index, newItems) => [
  ...arr.slice(0, index),
  ...newItems,
  ...arr.slice(index),
]
```

從中插入新成員 - splice

⚠️ 有副作用，會改變呼叫它的陣列

語法: `splice(要插入值在它後面的成員索引, 0, [值])`

```
const aArray = [1, 2, 3]
```

// 範例1：在索引 2 後插入新的成員 99，其它成員往後

// 結果: aArray = [1, 2, 999, 3]

// splice 第二傳入參數要保持為0，它是刪除成員用的，在這不用這個。

```
aArray.splice(2, 0, 99)
```

排序成員

⚠️ 有副作用，會改變呼叫它的陣列

```
const aArray = [2, 1, 3, 4, 5]

// 由小到大排序(只針對數字)
aArray.sort(function (a, b) {
  return a - b
})

// 由大到小排序(只針對數字)
aArray.sort(function (a, b) {
  return b - a
})
```

與字串交互應用

```
const aString = '1,2,3,4,5'  
const aArray = myString.split(',')
```

```
const bArray = ['a', 'b', 'c']  
const bString = bArray.join('-')
```

物件 (Object)

Eddy Chang

✉️ hello@eddychang.me



物件 . 車列 vs 物件

	陣列	物件
用途	用於資料集合	用於描述某種物件資料
有無順序	有序集合	無序集合(內部實作也是有序)
宣告運算子	方括號([])	花括號({})
存取成員運算子	方括號([])	點號(.)
建立方式	使用方括號([])宣告 + API	<ol style="list-style-type: none">1. 使用花括號({})宣告 + API2. 使用類別
判斷	Array.isArray	typeof (! 注意要排除 null)

註 1：物件也可以使用方括號([])來存取成員。建議用像變數識別名規則，來命名屬性名稱。

建立新物件

```
// 建立一個空白的物件
const emptyObject = {}

// 建立一個有屬性值的物件
const player = {
  name: 'Inori',
  age: 16,
}

// 用點(.)符號可以存取物件中的屬性
const name = player.name
// 更改屬性的值
player.name = 'Amy'
// 直接增加屬性
player.hairColor = 'pink'
```

建立新物件(工廠模式)

撰寫一個函式，最後回傳一個新物件

```
function createProduct(id, name, price) {  
    // 這裡可以由傳入值來調整所需的屬性值  
    return { id, name, price }  
}  
  
// 呼叫這個工廠函式，建立一個新的物件  
const product = createProduct('PS001', 'Play Station 5', 14990)
```

建立新物件(類別語法)

撰寫一個類別，建構式中設置好物件的對應屬性

```
class Product {  
    constructor(id, name, price) {  
        this.id = id  
        this.name = name  
        this.price = price  
    }  
}  
  
// 使用 new 運算子建立新的物件  
const product = new Product('PS001', 'Play Station 5', 14990)
```

物件拷貝(淺拷貝)

```
const a0bject = { a: 1, b: 2 }

// (★常用必會) 新語法 ES9(ES2018)
const b0bject = { ...a0bject }

// 舊語法用Object.assign
const c0bject = Object.assign({}, a0bject)
```

物件合併

```
const aObject = { a: 1, b: 2 }
const bObject = { x: 1, y: 2 }

// (★常用必會) 新語法 ES9(ES2018)
const cObject = { ...aObject, ...bObject }

// 舊語法用Object.assign
const dObject = Object.assign(aObject, bObject)
```

判斷是否存在屬性

```
const a0bject = { a: 1, b: 2 }

// 屬性名稱需要為字串
console.log(a0bject.a !== undefined)
// in運算子
console.log('a' in a0bject)
// hasOwnProperty方法
console.log(a0bject.hasOwnProperty('a'))
```

遍歷物件屬性(for...in)

```
const aObject = { a: 1, b: 2 }

for (let prop in aObject) {
  if (aObject.hasOwnProperty(prop)) {
    console.log(`aObject.${prop} = ${aObject[prop]}`)
  }
}
```

其它方法

```
const aObject = { a: 1, b: 2 }

// `Object.keys` 回傳一個給定物件所有可列舉屬性的字串陣列
// ['a', 'b']
Object.keys(aObject)

// `Object.values` 回傳一個給定物件所有可列舉屬性值的陣列
// [1, 2]
Object.values(aObject)

// `Object.entries` 回傳一個給定物件自身可列舉屬性的鍵值對陣列
// [['a', 1], ['b', 2]]
for (const [key, value] of Object.entries(aObject)) {
  console.log(` ${key}: ${value}`)
}
```

函式 (Function)

Eddy Chang

✉️ hello@eddychang.me



函式宣告

// 函式定義 - 使用有名稱的函式 Function Declarations (FD)

```
function sum(a, b) {  
    return a + b  
}
```

// 函式表達式 - 常數指定為匿名函式 Function Expressions (FE)

```
const sum = function (a, b) {  
    return a + b  
}
```

// (★常用必會) 箭頭函式，函式表達式的縮寫 Arrow Function

```
const sum = (a, b) => a + b
```

函式呼叫(執行)

函式名稱加上圓括號()，其中加入傳入的參數值，即可呼叫(執行)函式

```
const sum = (a, b) => a + b  
sum() // undefined + undefined = NaN  
sum(1, 2) // 3
```

函式傳入參數

```
// a 與 b 為函式的傳入參數
function sum(a, b) {
    return a + b
}

sum(1, 2) // 3
```

```
// (★常用必會) 函式傳入參數預設值 Default parameters
// 紿定 a 與 b 預設值，當沒給傳入參數值時(即 undefined)會套用預設值
function sum(a = 1, b = 1) {
    return a + b
}

sum() // 2
```

函式回傳值

```
// 函式必有回傳值，沒寫就是 undefined 值
function foo() {}
function bar() {
  return
}
// return 會跳出函式中的程序運行，代表函式此次執行結束
function sum(a, b) {
  if (a === 0) return 0
  return a + b
}
// return 後可以加上表達式(一樣會執行求值)，在箭頭函式裡很常用
function log(a) {
  return console.log(a)
}
// 上面函式相當於這個箭頭函式
const log = (a) => console.log(a)
```

以函式作為傳入參數值

```
// 傳入一個函式到另一個函式中，"高階函式(HOF)"特性
const log = (a) => console.log(a)

const sum = (a, b, fn) => {
  fn(a + b)
  return a + b
}

console.log(sum(1, 2, log))
```

函式最後回傳另一個函式

```
// 函式可以最後回傳另一個函式，"高階函式(HOF)"特性
function sum(a) {
    return function (b) {
        return a + b
    }
}
// 箭頭函式簡寫整個語法
const sum = (a) => (b) => a + b
// 這裡是回傳另個函式
const sumA = sum(1)
// 這裡才會計算回傳最後結果值
sumA(2)
// 直接寫成兩個函式呼叫圓括號
sum(1)(2)
```

作用域(作用範圍)

註: ES6 後，作用域是以區塊({})為分界

```
const x = 1

function outer(a) {
  const y = 2
  function inner(b) {
    const z = 3
    // 這個區塊中可以存取得到所有的變數值
    console.log(a, b, x, y, z)
  }
  return inner
}

outer(99)(100) // 99 100 1 2 3
```

閉包 Closure

閉包 Closure 一種資料結構，包含函式以及記住函式被建立時當下環境。每當函式被建立時，一個閉包就會被產生(自然特性)。

- 閉包結構中所記憶的環境值是用參照指向的(傳址)
- 函式作用域連鎖規則：內部函式可以有三個作用域：1.) 自己本身的 2.) 外部函式的 3.) 全域的
- 閉包的記憶環境例外變數：
 - this: 執行外部函式時的 this 值
 - arguments: 函式執行時的一個隱藏偽陣列物件

callback 回調、回呼函式

```
const el = document.getElementById('myButton')

// 函式作為傳入參數的資料
el.addEventListener(
  'click',
  function () {
    console.log('hello!')
  },
  false
)
```

提升(Hoist) - 變數

`var` 變數提升，值為 `undefined`；`let/const` 變數提升，但有 `TDZ` 造成的參照錯誤，相當來說變數使用會更安全

```
// --- file A -----
console.log(x) // undefined
var x = 5

// --- file B -----
console.log(y) // ReferenceError(TDZ)
let y = 5
```

提升(Hoist) - 函式

一般函式定義(FD)可以正常被提升

```
foo() // valid 合法

function foo() {
  console.log('Hello1')
}
```

函式表達式名稱被提升，但函式主體無法提升，TDZ 特性造成參照錯誤

```
bar() // ReferenceError(TDZ)

const bar = function () {
  console.log('Hello2')
}
```

IIFE(立即呼叫函式表達式)

只會在定義時呼叫(執行)一次的函式

```
(function () { //... })()
(function () { //... }())
```