陣列

陣列是一種有順序的複合式的資料結構，用於定義、存放複數的資料類型，在JavaScript的陣列中，並沒有規定它能放什麼資料進去，可以是原始的資料類型、其他陣列、函式等等。

陣列是用於存放大量資料的結構，要如何有效地處理資料，需要更加注意。它的搭配方法與語法很多，也有很多作相同事情的不同方式，並不是每樣都要學，有些只需要用到再查詢相關用法即可。

> 註: 雖然陣列資料類型是屬於物件，但Array這個包裝物件的 `typeof Array` 也是回傳'function'

陣列定義

陣列定義有兩種方式，一種是使用陣列字面文字，以下說明定義的方式。

陣列的索引值(index)是從0開始的順序整數值，陣列可以用方括號([])來取得成員的指定值，用這個方式也可以改變成員包含的值:

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

console.log(aArray.length) //0

aArray[0] = 1
aArray[1] = 2
aArray[2] = 3
aArray[2] = 5

console.log(typeof aArray) // object
console.log(aArray) // [1,2,5]
console.log(aArray.length) //3
console.log(aArray[3]) //undefined
```

> 註: 陣列為參照的(reference)資料類型，其中包含的值是可以再更改的，這與const的常數宣告無關。

另一種是使用Array包裝物件的預先分配空間的方式，但這種方式並不建議使用，這種定義語法除了容易搞混之外，經測試過它對效能並沒有太大幫助。而且這語法在分配後，一定會把長度值(成員個數)固定住，除非你百分之百確定陣列裡面的成員個數，不然千萬不要用。以下為範例:

```
//警告: 不要使用這種定義方式

//預先分配的定義
const aArray = new Array(10)
//混淆語法，這是定義三個陣列值
const bArray = new Array(1, 2, 3)
console.log(aArray.length) //10
```

> 註: JavaScript的內建物件都不建議 `new` 作初始定義的。不過有一些特例是一定要的，例如Date、Error等等。

陣列定義注意事項

一開始就搞混的語法

注意初學者很容易犯的一個錯誤，就是用下面這種陣列定義語法，這語法並不會導致執行錯誤，這是很怪異的合法定義語法，特別在這裡指出來就是希望你不要搞混了:

```
//這是錯誤示範
const aArray = [10]
```

實際上這相當於:

```
const aArray = []
aArray[0] = 10
```

多維陣列

對JavaScript中的陣列結構來說，多維陣列指的是"陣列中的陣列"結構，只是在陣列中保存其他的陣列值，例如像下面的範例:

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

magicMatrix[2][1]
```

維數過多時在處理上會愈複雜，一般常見的只有二維。通常需要另外撰寫專屬的處理函式，或是搭配額外的函式庫在容易上會較為方便。

關聯陣列

JavaScript中並沒有關聯陣列(Associative Array)這種資料結構，關聯陣列指的是"鍵-值"的資料陣列，也常被稱為字典(dictionary)的資料陣列，有很多程式語言有這樣的資料結構。

基本上在JavaScript中有複合類型只有物件和陣列，物件屬性的確是"鍵-值"對應的，但它並不是陣列，也沒有像陣列有這麼多方法可使用。雖然在ES6標準加入幾個特殊的物件結構，例如Set與Map，但目前的使用還不廣泛，其中支援的方法也少。在處理大量複合資料時，陣列的處理效率明顯高出物件許多，陣列的用途相當廣泛。

儲存多種資料類型

雖然並沒有規定說，你只能在同一個陣列中使用單一種資料類型。但是，在陣列中儲存多種不同的資料類型，絕對是個壞主意。在包含有大量資料的陣列中會嚴重影響處理效能，例如像下面這樣的例子。如果是多種資料類型，還不如先直接都用字串類型，需要取值時再作轉換。

```
var arr = [1, '1', undefined, true, 'true']
```

另外你需要注意的是，雖然數字類型在JavaScript中並沒有分浮點數或整數，但實際上在瀏覽器的JavaScript引擎(例如Google Chrome的V8引擎)中，整數的陣列的處理效率高於浮點數的陣列，可見其實引擎可以分辨各種不同的資料類型，然後會作最有效的儲存與運算，比你想像中聰明得很。

在ES6後加入了一種新式的進階資料結構，稱為型別陣列(Typed Arrays)，它是類似陣列的物件，但並非一般的陣列，也沒有大部份的陣列方法。這種資料結構是儲存特定的資料時使用的，主要是為了更有效率的處理二進位的資料(raw binary data)，例如檔案、圖片、聲音與影像等等。(註: Typed Arrays標準)

不過，就像上面一段說明的，聰明的JavaScript引擎在執行時會認得在一般陣列中儲存的資料類型，然後作最有效率的運算處理，在某些情況型別陣列(Typed Arrays)在運算上仍然不見得會比一般陣列還有效率。

多洞的陣列(Holey Arrays)或稀疏陣列(Sparse Arrays)

多洞的陣列代表你在定義陣列時，它的陣列值和索引(index)並沒有塞滿，或是從一個完整的陣列刪除其中一個(留了個空位)。像下面這樣的陣列定義:

```
const aArray = []
aArray[0] = 1
aArray[6] = 3

const bArray = [1, , 3]
```

另一種情況是，陣列大部份的值都是根本不使用的，例如用 `new Array(100)` 先定義出一個很大的陣列，但實際上裡面的值很少，這叫作稀疏陣列(Sparse Arrays)，其實多洞的陣列(Holey Arrays)或稀疏陣列(Sparse Arrays)都差不多指的是這一類的陣列，稀疏陣列可以再重新設計讓它在程式上的效率更高。這種陣列在處理效能上都會有很大的影響，在大的陣列中要避免使用到這樣的情況。

拷貝(copy)陣列

把原有的陣列指定給另一個變數(或常數)並不會讓它成為一個全新的陣列，這是因為當指定值是陣列時，是指定到同一個參照，也就是同一個陣列，看下面的範例:

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

aArray[0] = 100

console.log(bArray) //[100, 2, 3]
```

由範例可以看到， `bArray` 與 `aArray` 是共享同一陣列中的值，就像是不同名字的連體嬰，不論你在 `aArray` 或 `bArray` 中修改其中的值，增加或減少其中的值，都是對同一值作這件事。這種不叫拷貝陣列，只是指向同一個陣列而已。

拷貝陣列並不是像這樣作的，而且它可能是件複雜的事情，複雜的原因是陣列中包含的值，可以是各種不同的值，包含數字、字串、布林這些基本原始資料，也可以是其他陣列、物件、函式、其他特殊物件。而且物件類型的資料，都是使用參照(reference)指向的，裡面的資料也是複合式的，所以有可能也是複雜的。題外話是當你在進行拷貝物件資料時，也是會遇到同樣的複雜的情況。

拷貝陣列的情況，大致可以區分為淺拷貝(shallow copy)與深拷貝(deep copy)兩種。淺拷貝只能完全複製原陣列中，包含像數字、字串之類的基本原始資料值，而且當然是只有一維的平坦陣列，如果其中包含巢狀(多維)陣列、物件、函式、其他物件，只會複製到參照，意思是還是只能指向原來同一個值。

深拷貝(deep copy)反而容易理解，它是真正複製出另一個完全獨立的陣列。不過，深拷貝是一種高花費的執行程序，所以對於效率與精確，甚至能包含的特殊物件範圍都需要考慮。如果要進行深拷貝，一般就不直接用JavaScript內建的語法來達成，而是要用外部的函式庫例如jQuery、understore或lodash來作，而這些函式庫中的深拷貝不是只有支援陣列結構而已，同樣也支援一般物件或特殊物件的資料結構。不過，如果你的陣列結構很簡單，也可以用for或while迴圈自己作深拷貝這件事。

以下的方式主要是針對"淺拷貝"部份，一樣也有很多種方式可以作同樣這件事，以下列出四種:

展開(spread)運算符

> 推薦使用

ES6後的新運算符，長得像之前在函式章節講到的其餘參數，使用的也是三個點的省略符號(ellipsis)(...)，語法相當簡單，現在很常被使用:

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

它也可以用來組合陣列

```
const aArray = [1, 2, 3]
const bArray = [5, 6, ...aArray, 8, 9]
```

> 註: 展開(spread)運算符目前用babel轉換為ES5相容語法時，是使用 `concat` 方法

slice

slice(分割)原本是用在分割陣列為子陣列用的，當用0當參數或不加參數，相當於淺拷貝，這個方式是目前是效率較好的方式，語法也很簡單:

```
const newArray = oldArray.slice(0)
const newArray = oldArray.slice()
```

concat

concat(串聯)是用於合併多個陣列用的，把一個空的陣列和原先陣列合併，相當於拷貝的概念。在這裡寫出來是為了比較一下展開運算符:

```
const newArray = [].concat(oldArray)
```

for/while迴圈語句

迴圈語句也可以作為淺拷貝，語句寫起來不難也很直覺，只是相較於其他方式要多打很多字，通常不會單純只用來作淺拷貝。以下為範例程式:

```
const newArray = []

for (let i = 0, len = oldArray.length ; i < len ; i++){
    newArray[i] = oldArray[i]
}
```

```
const newArray = []
let i = oldArray.length

while (i--){
    newArray[i] = oldArray[i]
}
```

判別是否為陣列

最常見的情況是，如果有個函式要求它的傳入參數之一為陣列，而且確定不能是物件或其他類型。你要如何判斷傳進來的值是真的一個陣列？

直接使用 `typeof` 來作判斷是沒辦法作這件事的，它對陣列資料類型只會直接回傳 `'object'` 。

在JavaScript中，有很多種方式可以作同一件事，這件事也不意外，不過每種方法都有一些些不同的細節或問題。以下的 `variable` 代表要被判斷的變數(或常數)值。

isArray

> 推薦使用

最簡單的判斷語法應該是這個，用的是內建Array物件中的 `isArray` ，它是個ES5標準方法:

```
Array.isArray(variable)
```

constructor

下面這個是在Chrome瀏覽器中效能最佳的判斷方法，它是直接用物件的建構式來判斷:

```
variable.constructor === Array
```

如果你是要判斷物件中的其中屬性是否為陣列，你可以先判斷這個屬性是否存在，像下面這樣(prop指的是物件屬性):

```
variable.prop && variable.prop.constructor === Array
```

> 失效情況: 當使用在一個繼承自陣列的陣列會失效

instanceof

這也是用物件的相關判別方法來判斷， `instanceof` 是用於判斷是否為某個物件的實例，優點為語法簡潔清楚:

```
variable instanceof Array
```

> 失效情況: 處理不同window或iframe時的變數會失效

toString.call

> 推薦使用

這也是用物件中的 `toString` 方法來判斷，這是所有情況都可以正確判斷的一種。它也是萬用方式，可以判斷陣列以外的其他特別物件，缺點是效率最差:

```
Object.prototype.toString.call(variable) === '[object Array]'
```

> 註: jQuery、underscore函式庫中的判斷陣列的API是用這種判斷方法

> 註: 在JavaScript: The Definitive Guide, 6th Edition書中有提到， `Array.isArray` 其實就是用這個方式的實作方法。

方式結論

這幾個方式的選擇，我的建議是只要學最後一種就行了(不考慮舊瀏覽器就用第一種)，它可以正確判斷並應用在各種情況，有時候正確比再快的效能更重要，更何況它其實是萬用的，除了陣列之外也可以用於其它的判斷情況。雖然它的語法對初學者來說，可能無法在此時完全理解，不過就先知道要這樣用就行了。

參考資料: How do you check if a variable is an array in JavaScript?

陣列屬性與方法

陣列屬性與方法的細節多如牛毛，以下只列出常用到的方法與屬性。

屬性

length長度(成員個數)

`length` 用來回傳陣列的長度(成員個數)，這個屬性有時候是不可信的，就如同上面用 `new Array(10)` 定義時，會被固定住為10，不論現在的裡面的值有多少個。多洞的陣列中也是與目前有值成員的個數不同:

```
const aArray = [1, , undefined, '123'] //有洞的陣列
console.log(aArray.length) //4
```

多維陣列的情況上面有說明過了，只是陣列中的陣列而已，它的 `length` 只會回傳最上層陣列的個數:

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

console.log(magicMatrix.length) //3
```

`length` 的整數值**竟然**是可以更動的，它並不是只能讀不能寫的屬性:

```
const bArray = [1, 2, 3]
console.log(bArray.length)

bArray.length = 4
console.log(bArray.length)
console.log(bArray)

bArray.length = 2
console.log(bArray.length)
console.log(bArray)
```

更動 `length` 經測試過，事實上是從陣列最後面"**截短(truncate)**"的語法，它的效率是所有類似功能語法中最好的:

```
const sArray = ['apple', 'banana', 'orange', 'mongo']

sArray.length = 2
console.log(sArray)
```

另外， `length` 指定為0也可以用於清空陣列，清空陣列一樣也是有好幾種方式，以下為各種清空陣列的範例程式碼，注意第一種的原陣列不能使用 `const` 宣告，就意義上它不是真的把原來的陣列清空。一般情況下第一種效率最好，第四種最差:

```
let aArray = ['apple', 'banana', 'orange', 'mongo']

//第一種
aArray = []

//第二種
aArray.length = 0

//第三種
while(aArray.length > 0) {
    aArray.pop();
}

//第四種
aArray.splice(0, aArray.length)
```

方法

indexOf

`indexOf` 是簡便的搜尋索引值用的方法，它可以給定一個要在陣列中搜尋的成員(值)，如果找到的話就會回傳成員的索引值，沒找到就會回傳"-1"數字。多個成員符合的話，它只會回傳最先找到的那個(一律是從左至右)，它的比對是使用完全符合的比較運算符(===)，可加入第二個參數，這是可選擇的，它是"開始搜尋的索引值"，如果是負整數則從最後面倒過來計算位置的(最後一個索引值相當於 `-1`)。

```
const aArray = ['a', 'b', 'c', 'a', 'c', 'c']

console.log(aArray.indexOf('a')) //0
console.log(aArray.indexOf('c')) //2
console.log(aArray.indexOf('c', 3)) //4
console.log(aArray.indexOf('a', -3)) //3，-3代表要從索引值3開始搜尋
```

pop與push、shift與unshift

> 副作用方法

陣列的傳統處理方法，pop是"砰出"最後面一個值，然後把這個值從陣列移除掉。push是"塞入"一個值到陣列最後面。shift與pop類似，不過它是砰出最前面的值。unshift則與push類似，它是塞到陣列列前面。

`pop` 的例子如下，它會回傳被砰出的值:

```
const aArray = [1, 2, 3]
const popValue = aArray.pop()

console.log(aArray) //[1,2]
console.log(popValue) //3
```

`push` 的例子如下，它則是回傳新的長度值:

```
const aArray = [1, 2, 3]
aArray.push(4)

console.log(aArray) //[1,2,3,4]

const pushValue = aArray.push(5)

console.log(aArray) //[1,2,3,4,5]
console.log(pushValue) //5
```

> 口訣記法: 有"p"的pop與push是針對陣列的"屁股"(最後面)。 `pop-corn` 是爆米花，所以pop用來爆出值的。有u的push與unshift是同一掛的。

concat

`concat` (串聯)是用於合併其他陣列或值，到一個陣列上的方法。它最後會回傳一個新的陣列。

> 語法: array.concat(value1[, value2[, ...[, valueN]]])

前面已經有看到它可以作陣列的淺拷貝，它與展開運算符可以互為替代，以下為一個簡單的範例:

```
const strArray = ['a', 'b', 'c']
const numArray = [1, 2, 3]
const aString = 'Hello'

const newArray = strArray.concat(numArray)
console.log(newArray)

//連鎖(chain)運算
const newArray1 = strArray.concat(numArray).concat(aString)
console.log(newArray1)

//展開運算符 相等的作法
const newArray2 = [...strArray, ...numArray]
console.log(newArray2)
```

> 注意: 陣列沒有運算符這種東西。但是字串類型可以用合併運算符(+)，或是用同名方法concat作合併。

slice

slice(分割)是用於分割出子陣列的方法，它會用淺拷貝(shallow copy)的方式，回傳一個新的陣列。這個方法與字串中的分割子字串所使用的的同名稱方法 `slice` ，類似的作法。

> 語法: array.slice(start[, end])

`slice` 使用陣列的索引值作為前後參數，大部份的重點都是這兩個索引值的正負值情況。以下是對於特殊情況的大致規則:

- 當開頭索引值為undefined時(空白沒寫)，它會以0計算，也就是陣列最開頭。
- 當索引值有負值的情況下，它是從最後面倒過來計算位置的(最後一個索引值相當於 `-1`)
- 只要遵守"開頭索引值"比"結束索引值"的位置更靠左，就有回傳值。
- `slice(0)` 相當於陣列淺拷貝

```
const aArray = [1, 2, 3, 4, 5, 6]

const bArray = aArray.slice(1, 3)
const cArray = aArray.slice(0)
const dArray = aArray.slice(-1, -3)
const eArray = aArray.slice(-3, -1)
const fArray = aArray.slice(1, -3)
```

splice

> 副作用方法

splice(粘接)這個字詞與上面的slice(分割)長得很像，但用途不相同，這個方法是用於刪除或增加陣列中的成員，以此來改變原先的陣列的。

為何會有這種用途？主要是要在陣列的中間"插入"幾個新的成員(值)，或是"刪掉"其中的幾個成員(值)用的。

> 語法: array.splice(start, deleteCount[, item1[, item2[, ...]]])

這個方法的參數值會比較多用起來會複雜些，先說明它的參數如下:

- start 是開始要作這件事的索引值，左邊從0開始，如果是負數則從右邊(最後一個)開始計算
- deleteCount 是要刪除幾個成員(值)，最少為0代表不刪除成員(值)
- item1... 這是要加進來的成員(值)，不給這些成員(值)的話，就只會作刪除的工作，不會新增成員(值)。注意如果是陣列值，會變成巢狀(子)陣列成員(值)。

實際上有幾個基本的用法範例，splice通常會用在一些特定的情況，也可能會搭配搜尋語法或迴圈語句。以下為常用的幾個範例:

插入一個新成員(值)在某個值之後

插入某個值之後需要先找出這個某個值的索引值，所以會用 `indexOf` 來找，不過如果是多個值的情況，就要用迴圈語句了，這是只有單個"某個值"的情況。下面的兩個範例的結果是相同的，所以你要在"某個值"的後面插入新成員(值)，"後面"代表新成員(值)的索引值還需要加1才行。

```
const dictionary = ['a', 'b', 'd', 'e', 'f']

//先找到b的位置，等會要在b後面插入c
const bIndex = dictionary.indexOf('b')

//bIndex大於-1代表有存在，插入c，不刪除
if (bIndex > -1) {
    dictionary.splice(bIndex+1, 0, 'c')
}
```

用新成員(值)取代某個值

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要用c來取代x
const xIndex = dictionary.indexOf('x')

//bIndex大於-1代表有存在，插入c，刪除x
if (bIndex > -1) {
    dictionary.splice(xIndex, 1, 'c')
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1, 'c')
    }
}
```

用於刪除成員(值)

> 註: 其實完全與取代範例幾乎一樣

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要刪除
const xIndex = dictionary.indexOf('x')

//xIndex大於-1代表有存在，刪除x
if (xIndex > -1) {
    dictionary.splice(xIndex, 1)
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1)
```

```
    }
}
```

陣列與字串 join與split

join(結合)與split(分離)是相對的兩個方法，join(結合)是陣列的方法，用途是把陣列中的成員(值)組合為一個字串，組合的時候可以加入組合時用的分隔符號(或空白字元)。

```
const aArray = ['Hello', 'Hi', 'Hey']
const aString = aArray.join()   //Hello,Hi,Hey
const bString = aArray.join(', ') //Hello, Hi, Hey
const cString = aArray.join('') //HelloHiHey
```

split(分離)是倒過來，它是字串中的方法，把字串拆解成陣列的成員，拆解時需要給定一個拆解基準的符號(或空白)，通常是用逗號(,)分隔，以下為範例:

```
const monthString = 'Jan,Feb,Mar,Apr,May'
const monthArray1 = monthString.split(',') //["Jan", "Feb", "Mar", "Apr", "May"]

//以下為錯誤示範
const monthArray2 = monthString.split()  //["Jan,Feb,Mar,Apr,May"]
const monthArray3 = monthString.split('') //["J", "a", "n", ",", "F", "e", "b", ",", "M", "a", "r", ",", "A", "p", "r", ",", "M",
```

迭代

> forEach為副作用方法

陣列的迭代可以單純地使用迴圈語句都可以作得到，但在ES6後加入幾個新的方法，例如 `forEach` 、 `map` 與 `reduce` 提供更多彈性的運用。

forEach

`forEach` 類似於for迴圈，但它執行語句的是放在回調函式(callback)的語句中，下面這兩個範例是相等功能的:

```
const aArray = [1, 2, 3, 4, 5]
for (let i = 0, len = aArray.length; i < len; i++) {
    // 對陣列元素作某些事
    console.log(i, aArray[i])
}
```

```
const aArray = [1, 2, 3, 4, 5]
aArray.forEach(function(value, index, array){
// 對陣列元素作某些事
console.log(index, value)
})
```

forEach的回調函式(callback)共可使用三個參數:

- value 目前成員的值
- index 目前成員的索引
- array 要進行尋遍的陣列

forEach雖然可以與for迴圈語句相互替代，但他們兩個設計原理不同，也有一些明顯的差異，在選擇時你需要考慮這些。不同之處在於以下幾點:

- forEach無法提早結束或中斷
- forEach被呼叫時， `this` 會傳遞到回調函式(callback)裡
- forEach方法具有副作用

map(映射)

map(映射)反而是比較常被使用的迭代方法，由於它並不會改變輸入陣列(呼叫map的陣列)的成員值，所以並不會產生副作用，現在有很多程式設計師改用它來作為陣列迭代的首選使用方法。

map(映射)一樣會使用回調函式(callback)與三個參數，而且行為與forEach幾乎一模一樣，不同的地方是它會回傳一個新的陣列，也因為它可以回傳新陣列，所以map(映射)可以用於串接(chain)語法結構。

```
var aArray = [1, 2, 3, 4];
var newArray = aArray.map(function (value, index, array) {
    return value + 100
})

//原陣列不會被修改
console.log(aArray) // [1, 2, 3, 4]
console.log(newArray) // [101, 102, 103, 104]
```

reduce(歸納)

reduce(歸納)這個方法是一種應用於特殊情況的迭代方法，它可以藉由一個回調(callback)函式，來作前後值兩相運算，然後不斷縮減陣列中的成員數量，最終回傳一個值。reduce(歸納)並不會更動作為傳入的陣列(呼叫reduce的陣列)，所以它也沒有副作用。

reduce(歸納)比map又更複雜了一些，它多了一個參數值，代表前一個值，主要就是它至少要兩個值才能進行前後值兩相運算。你也可以給定它一個初始值，這時候reduce(歸納)會從索引值的0接著取第二個值進行迭代，如果你沒給初始值，reduce(歸納)會從索引值1開始開始迭代。可以用下面這個範例來觀察它是如何運作的:

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

const total = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = ', index)
    return pValue + value
})

console.log(aArray) // [0, 1, 2, 3, 4]
console.log(total) // 10

//下面給定初始值10，注意它放的地方是在回調函式之後的參數中
const total2 = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = ', index)
    return pValue + value
}, 10)

console.log(total2) // 20
```

按照這個邏輯，reduce(歸納)具有分散運算的特點，可以用於下面幾個應用之中:

- 兩相比較最後取出特定的值(最大或最小值)
- 計算所有成員(值)，總合或相乘
- 其它需要兩兩處理的情況(組合巢狀陣列等等)

排序與反轉 sort與reverse

sort

> 副作用方法

sort是一個簡單的排序方法，以Unicode字串碼順序來排序，對於英文與數字有用，但中文可能就不是你要的。以下為簡單的範例:

```
const fruitArray = ['apple', 'mongo', 'cherry', 'banana' ]
fruitArray.sort()
console.log(fruitArray) //["apple", "banana", "cherry", "mongo"]


const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.sort()
console.log(fruitArray) //["櫻桃", "芒果", "蘋果", "香蕉"]
```

中文的排序一般來說只會有兩種情況，一種是要依big5編碼來排序，另一種是要依筆劃來排序，這時候需要在sort方法傳入參數中，另外加入比較的回調(callback)函式，這個回調函式中將使用localeCompare這個可以比較本地字串的方法，以下為範例程式，其中本地(locale)的參數請參考locales argument:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉', '大香蕉', '小香蕉' ]

//使用原本的排序
fruitArray.sort()

console.log(fruitArray)
//["大香蕉", "小香蕉", "櫻桃", "芒果", "蘋果", "香蕉"]


fruitArray.sort(function(a, b){
  //zh-Hans-TW、zh-Hans-TW-u-co-big5han、pinyin等等參數同樣結果
  return a.localeCompare(b, 'zh-Hans-TW')
})

console.log(fruitArray)
//["大香蕉", "芒果", "蘋果", "香蕉", "小香蕉", "櫻桃"]

fruitArray.sort(function(a, b){
  //按筆劃從小到大排序
  return a.localeCompare(b, 'zh-Hans-TW-u-co-stroke')
})

console.log(fruitArray)
//["大香蕉", "小香蕉", "芒果", "香蕉", "蘋果", "櫻桃"]
```

> 註: 這個字串比較方法應該可以再最佳化其效率，有需要可以進一步參考其文件的選項設定

reverse

> 副作用方法

reverse(反轉)這語法用於把整個陣列中的成員順序整個反轉，就這麼簡單。以下為範例:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.reverse()

console.log(fruitArray)
//["香蕉", "櫻桃", "芒果", "蘋果"]
```

另一個情況是字串中的字元，如果要進行反轉的話，並沒有字串中的 `reverse` 方法，要用這個陣列的 `reverse` 方法加上字串與陣列的互相轉換的split與join方法，可以使用以下的函式:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}
```

過濾與搜尋 filter與find/findIndex

filter(過濾)是使用一個回調(callback)函式作為傳入參數，將的陣列成員(值)進行過濾，最後回傳符合條件(通過測試函式)的陣列成員(值)的新陣列。它的傳入參數與用法與上面說的迭代方法類似，實際上也是另一種特殊用途的迭代方法:

```
const aArray = [1, 3, 5, 7, 10, 22]

const bArray = aArray.filter(function (value, index, array){
        return value > 6
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bArray) //[7, 10, 22]
```

find與findIndex方法都是在搜尋陣列成員(值)用的，一個在有尋找到時會回傳值，一個則是回傳索引值，當沒找到值會回傳undefined。它們一樣是使用一個回調(callback)函式作為傳入參數，來進行尋找的工作，回調函式的參數與上面說的迭代方法類似。

findIndex與最上面說的indexOf不同的地方也是在於，findIndex因為使用了回調(callback)函式，可以提供更多的在尋找時的彈性應用。以下為範例:

```
const aArray = [1, 3, 5, 7, 10, 22]
const bValue = aArray.find(function (value, index, array){
        return value > 6
    })
const cIndex = aArray.findIndex(function (value, index, array){
        return value > 6
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bValue) //7
console.log(cIndex) //3
```

陣列處理純粹函式

改寫原有的處理方法(或函式)為純粹函式並不困難，相當於要拷貝一個新的陣列出來，進行處理後回傳它。ES6後的展開運算子(...)可以讓語法更簡潔。

> 註: 以下範例並沒有作傳入參數的是否為陣列的檢查判斷語句，使用時請再自行加上。

> 註: 下列範例來自Pure javascript immutable arrays，更多其他的純粹函式可以參考這裡的範例。

push

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function purePush(aArray, newEntry){
  return [ ...aArray, newEntry ]
}

const purePush = (aArray, newEntry) => [ ...aArray, newEntry ]
```

pop

```
//注意它並非回傳pop的成員(值)，而是回傳最終的陣列結果
function purePop(aArray){
  return aArray.slice(0, -1)
}

const purePush = aArray => aArray.slice(0, -1)
```

shift

```
//注意它並非回傳shift的成員(值)，而是回傳最終的陣列結果
function pureShift(aArray){
  return aArray.slice(1)
}

const pureShift = aArray => aArray.slice(1)
```

unshift

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function pureUnshift(aArray, newEntry){
  return [ newEntry, ...aArray ]
}

const pureUnshift = (aArray, newEntry) => [ newEntry, ...aArray ]
```

splice

這方法完全要使用slice與展開運算符(...)來取代，是所有的純粹函式最難的一個。

```
function pureSplice(aArray, start, deleteCount, ...items) {
  return [ ...aArray.slice(0, start), ...items, ...aArray.slice(start + deleteCount) ]
}

const pureSplice = (aArray, start, deleteCount, ...items) =>
[ ...aArray.slice(0, start), ...items, ...aArray.slice(start + deleteCount) ]
```

sort

```
//無替代語法，只能拷貝出新陣列作sort
function pureSort(aArray, compareFunction) {
  return [ ...aArray ].sort(compareFunction)
}

const pureSort = (aArray, compareFunction) => [ ...aArray ].sort(compareFunction)
```

reverse

```
//無替代語法，只能拷貝出新陣列作reverse
function pureReverse(aArray) {
  return [ ...aArray ].reverse()
}

const pureReverse = aArray => [ ...aArray ].reverse()
```

delete

刪除(delete)其中一個成員，再組合所有子字串:

```
function pureDelete (aArray, index) {
   return aArray.slice(0,index).concat(aArray.slice(index+1))
}

const pureDelete = (aArray, index) => aArray.slice(0,index).concat(aArray.slice(index+1))
```

英文解說

常見問題

為什麼要那麼強調副作用？

副作用的概念早已經存在於程式語言中很久了，但在最近幾年才受到很大的重視。在過去，我們在撰寫這種腳本直譯式程式語言，最重視的其實是程式效率與相容性，因為同樣的功能，不同的寫法有時候效率會相差很多，也有可能這是不同瀏覽器品牌與版本造成的差異。

但現在的電腦硬體已經進步太多，所謂的執行資源的限制早就與10年前不同。效率在今天的程式開發早就已經不是唯一的重點，更多其他的因素都需要加入來一併考量，以前的應用程式也可能只是小小的特效或某個小功能，現在的應用程式將會是很龐大而且結構複雜的。所以，程式碼的閱讀性與語法簡潔、易於測試、維護與除錯、易於規模化等等，都會變成要考量的其他重點。純粹函式的確是未來的主流想法，當一個應用程式慢慢變大、變複雜，純粹函式可以提供的好處會變成非常明顯，所以一開始學習這個概念是必要的。

參考