ES6篇: Side Effects(副作用)與Pure Functions(純粹函式)

本章的目標是對Side Effect(副作用)與(Pure Functions)純粹函式的概念提供一些簡要的說明。Pure Functions(純粹函式)的概念是來自於Side Effects(副作用)，這是一個電腦科學(程式語言)中經常看到的名詞，本章的內容與ES6標準雖無關係，但也有運用一些ES6中的語法。本章的內容會與開發者如何撰寫功能性比較有關。

React在一開始學習時，這個概念還不明顯，實際上React的內部設計是有純粹函式的概念。但只要一學到Flux與Redux，純粹函式並不是一個選項，是必定要使用的，這個部份大概是所有學過Redux的初學者，最難理解的其中一個主軸。但如果你不先學習什麼是純粹函式，什麼是副作用，馬上就學Flux與Redux的話，相信一定很難一下子進入狀況。

> 註: 本文章同步放置於Github庫的這裡。

副作用是什麼？

你如果有去醫院看過醫生拿過藥，在藥包上都會註明吃了這個藥物後會產生的副作用，Side Effect(副作用)就是這個意思，也就是除了主要的作用外，會產生額外的其他作用。Side Effect(副作用)這個名詞，現在經常會出現在很多進階使用的JavaScript框架或函式庫之中。實際上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程式語言，最重視的其實是程式效率與相容性，因為有可能同樣的功能，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。在受限的執行環境之中(瀏覽器)，原本可以作的程式應用就很有限，反而不太會去對應用程式的整體設計模式或樣式太講究。

但現在的電腦硬體與網路速度已經進步太多，所謂的執行資源的限制早就與10多年前完全不同，許多大公司的紛紛投入，現在的JS引擎已經從10多年前的50cc小綿羊，進步到跑車等級的速度。效率在今天的JavaScript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。純粹函式的確是現在與未來的主流想法，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。

上面都在講概念的東西，來看個簡單的實例，副作用可以在很多情況下使用，並沒有很嚴格的說什麼就是什麼，這是個概念性的講法。下面這段話摘譯於這篇文章中:

> 在JavaScript中習慣地最好使用嚴格相等比較 `===` 與 `!==` ，而不要使用值相等比較 `==` 與 `!=` ，是因為要避免任何不經意的"副作用"

為何？說值相等比較會有不經意的副作用？主要的原因是在ES的標準中，值相等比較會作類型的強制轉換，也就是在進行不同類型的兩個值比較時，它會在執行時的內部作你看不到的資料類型的強制轉換，除非你很理解這轉換的步驟與結果是什麼，不然這個隱藏在內部的轉換機制，很容易產生所謂不經意的副作用。

所以程式碼到處都有副作用？廣義的來說，的確如此。這種講法到處都可以在網路上找到。尤其是在JavaScript這個語言中，因為它有一個非常特殊的設計，就是單一個執行緒，有可能某個方法因為產生了副作用，所以會阻擋到其他方法的執行等等，JavaScript引入了異步執行的方式來解決單執行緒的並行問題。

要一個很清楚的概念是，Side Effect(副作用)並不是指"好"或者"不好"的意思，而是它有可能會影響到其他環境的使用情況。對於具有副作用的情況(表達式或函式)在使用時要特別小心注意。在程式語言中，副作用除了在表達式中有這個概念，在函式中也會見到它。

> 註: 維基百科中關於副作用的中文翻譯是有問題的，請主要看英文說明部份。

Side Effects(副作用)於表達式中

在表達式中就有使用這個概念來進行分類。Expression(表達式)就上面所說的，是用來求出值的，但常見的一些Expression(表達式)的作用也有可能指定值，而當在指定原本的變數/常數的表達式時，如果會變動原本的變數/常數的值，就稱為是"具有Side Effect(副作用)"的Expression(表達式)，例如像下面這些都是具有副作用的表達式範例:

```
counter++
x += 3
y = "Hello " + name
```

沒有副作用的表達式的範例，它們大概都只是字面量(literal)或變數/常數名稱，或是會產生一個新的運算結果值:

```
3 + 5
true
1.9
x
x > y
'Hello World'
```

上面的表達式的分類會比較容易理解，上一節的那個嚴格相等比較與值相等比較的例子，是屬於語言標準中原本的內建設計部份，會比較難理解些。不過你可以仔細的思考一下，其實也不難。差不多就是這個意思吧，用語言說的確不容易說得清。

Side Effects(副作用)於函式中 & Pure Functions(純粹函式)

在表達式中，我們有講到在表達式中有無副作用(Side Effect)的分別，函式也有這種區分方式。對於函式來說，具有副作用代表著可能會更動到外部環境，或是更動到傳入的參數值。函式的區分是以 純粹(pure)函式 與 不純粹(impure)函式 兩者來區分，但這不光只有無副作用的差異，還有其他的條件。

純粹函式(pure function)即滿足以下三個定義的函式(以下的定義是來自於Redux的概念):

- 給定相同的輸入(傳入值)，一定會回傳相同輸出值結果(回傳值)
- 不會產生副作用
- 不依賴任何外部的狀態

一個典型的純粹函式的例子如下:

```
const sum = function(value1, value2) {
  return value1 + value2
}
```

套用上面說的定義，你可以用下面這樣理解它是不是一個純粹函式:

- 只要每次給定相同的輸入值(例如1與2)，就一定會得到相同的輸出值(例如3)
- 不會改變原始輸入參數，或是外部的環境，所以沒有副作用
- 不依賴其他外部的狀態(變數之類的)

那什麼又是一個不純粹的函式？看以下的範例就是，它需要依賴外部的狀態值(變數值):

```
let count = 1

let increaseAge = function(value) {
  return count += value
}
```

在JavaScript中不純粹函式很常見，像我們一直用來作為輸出的 `console.log` 函式，或是你可能會在很多範例程式看到的 `alert` 函式，都是"不"純粹函式，這類函式通常沒有回傳值，都是用來作某件事，像 `console.log` 會更動瀏覽器的主控台(外部環境)的輸出，也是一種副作用。

每次輸出值都不同的不純粹函式一類，最典型的就是 `Math.random` ，這是產生隨機值的內建函式，既然是隨機值當然每次執行的回傳值都不一樣。

例如在陣列的內建方法中，有一些是有副作用，而有一些是無副作用的，這個部份需要查對應表才能夠清楚。不會改變傳入的陣列的，會在作完某件事後回傳一個新陣列的方法，就是無副作用的純粹函式(方法)，而會改變原陣列就算是不純粹函式(方法)了。

下面是兩個在陣列中作同樣事情的不同方法，都是要取出只包含陣列的前三個成員的陣列。一個用splice，另一用是slice，看起來都很像，連這兩個方法的名稱都很像，但卻是完全屬於不同的種類:

```
// 不純粹(impure)，splice會改變到原陣列
const firstThree = function(arr) {
  return arr.splice(0,3)
}

// 純粹(pure)，slice會回傳新陣列
const firstThree = function(arr) {
  return arr.slice(0,3)
}
```

其他有許多內建的或常用的函式都是免不了有副作用的，例如這些應用:

- 會改變傳入參數(物件、陣列)的函式(方法)
- 時間性質的函式，setTimeout等等
- I/O相關
- 資料庫相關
- AJAX

純粹函式當然有它的特別的優點:

- 程式碼閱讀性提高
- 較為封閉與固定，可重覆使用性高
- 輸出輸入單純，易於測試、除錯
- 因為輸入->輸出結果固定，可以快取或作記憶處理，在高花費的應用中可作提高執行效率的機制

最後，並不是說有副作用的函式就不要使用，而且要很清楚的理解這個概念，然後儘可能在你自己的撰寫的一般功能函式上使用純粹函式，以及讓必要有副作用的函式得到良好的管控。現在已經有一些新式的函式庫或框架(例如Redux)，會特別要求在某些地方只能使用純粹函式，而具有副作用的不純粹函式只能在特定的情況下才能使用。

陣列相關純粹函式

因為陣列的處理方法很常使用到，而且對初學者來說會比較困難，所以我把在網路上目前找到的改寫過的純粹函式列出來。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，通常會使用ES6的展開運算符(...)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: 以下範例來自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

```
//注意它並非像pop是回傳成員(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

```
//注意它並非像shift是回傳成員(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，再組合所有子成員:

```
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
```

結論

本章是一個概念性的說明章節，這個概念相當的簡單，但非常的重要。在React中其實處處可見到純粹函式的設計，但可能沒那麼顯眼，我們在其中都有使用到，只是身在其中有可能不自知。你可能沒注意到像React官網的這個網頁Components and Props中就有這一句粗體的文字，這是在講解元件對於自己本身props的嚴格規則:

> All React components must act like pure functions with respect to their props.

> 所有的React元件必須運作得就像相對於它們props(屬性)的純粹函式

Redux並非是純粹函式的提倡者，其實它只是順應了React的設計，並且更進化應用了這些設計。現在我們離React愈來愈近了，再過幾天我們將進入真正的React中開始學習。