

30 天精通RxJS (02) : Functional Programming 基本观念

Dec 18th, 2016 . 4 mins read

Functional Programming 是Rx 最重要的观念之一，基本上只要学会FP 要上手Rx 就不难了！Functional Programming 可以说是近年来的显学，各种新的函式编程语言推出之外，其他旧有的语言也都在新版中加强对FP 的支援！

这是【30天精通RxJS】的02篇，如果还没看过01篇可以往这边走：[30天精通RxJS \(01\)：认识RxJS](#)

什么是Functional Programming ?



Series / 30 天精通RxJS



Functional Programming 是一种编程范式(programming paradigm)，就像Object-oriented Programming(OOP)一样，就是一种写程式的方法论，这些方法论告诉我们如何思考及解决问题。

简单说Functional Programming 核心思想就是做运算处理，并用function 来思考问题，例如像以下的算数运算式：

$$(5 + 6) - 1 * 3$$

我们可以写成

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



我们把每个运算包成一个个不同的function，并用这些function 组合出我们要的结果，这就是最简单的Functional Programming。

Functional Programming 基本要件

跟OOP一样不是所有的语言都支持FP，要能够支持FP的语言至少需要符合**函式为一等公民**的特性。

函式为一等公民(First Class)

一等公民就是指跟其他资料型别具有同等地位，也就是说函式能够被赋值给变数，函式也能够被当作参数传入另一个函式，也可当作一个函式的回传值

函式能够被赋值给变数

```
var hello = function() {}
```

函式能被当作参数传入

```
fetch('www.google.com')  
.then(function(response) {}) // 匿名 function 被传入 then()
```

函式能被当作回传值

```
var a = function(a) {  
  return function(b) {  
    return a + b;  
  };  
  // 可以回传一个 function  
}
```

Functional Programming 重要特性



Series / 30 天精通RxJS

表达式是一个运算过程，一定会有返回值，例如执行一个function

```
add(1, 2)
```

- 陈述式则是表现某个行为，例如一个赋值给一个变数

```
a = 1;
```

有时候表达式也可能同时是合法的陈述式，这里只讲基本的判断方法。如果想更深入了解其中的差异，可以看这篇文章 [Expressions versus statements in JavaScript](#)

由于Functional Programming 最早就是为了做运算处理不管I/O，而Statement 通常都属于对系统I/O 的操作，所以FP 很自然的不会是Statement。

当然在实务中不可能完全没有I/O 的操作，Functional Programming 只要求对I/O 操作限制到最小，不要有不必要的I/O 行为，尽量保持运算过程的单纯。

Pure Function

Pure function 是指一个function 给予相同的参数，永远会回传相同的返回值，并且没有任何显著的副作用(Side Effect)

举个例子：

```
var arr = [1, 2, 3, 4, 5];  
  
arr.slice(0, 3); // [1, 2, 3]  
  
arr.slice(0, 3); // [1, 2, 3]  
  
arr.slice(0, 3); // [1, 2, 3]
```

这里可以看到slice不管执行几次，返回值都是相同的，并且除了返回一个值(value)之外并没有做任何事，所以slice 就是一个pure function。

```
var arr = [1, 2, 3, 4, 5];
```



Series / 30 天精通RxJS

```
arr.slice(0, 3); // []
```

这里我们换成用 `splice`，因为 `splice` 每执行一次就会影响 `arr` 的值，导致每次结果都不同，这就很明显不是一个 pure function。

Side Effect

Side Effect 是指一个 function 做了跟本身运算返回值没有关系的事，比如说修改某个全域变数，或是修改传入参数的值，甚至是执行 `console.log` 都算是 Side Effect。

Functional Programming 强调没有 Side Effect，也就是 function 要保持纯粹，只做运算并返回一个值，没有其他额外的行为。

这里列举几个前端常见的 Side Effect，但不是全部

- 发送 http request
- 在画面印出值或是 log
- 获得使用者 input
- Query DOM 物件

Referential transparency

前面提到的 pure function 不管外部环境如何，只要参数相同，函式执行的返回结果必定相同。这种不依赖任何外部状态，只依赖于传入的参数的特性也称为引用透明 (Referential transparency)

利用参数保存状态

由于最近很红的 Redux 使我能很好的举例，让大家了解什么是用参数保存状态。了解 Redux 的开发者应该会知 Redux 的状态是由各个 reducer 所组成的，而每个 reducer 的状态就是保存在参数中！

```
function countReducer(state = 0, action) {  
  // ...  
}
```

如果你跟 Redux 不熟可以看下面递归的例子

```
function findIndex(arr, predicate, start = 0) {  
  if (0 <= start && start < arr.length) {  
    if (predicate(arr[start])) {
```



Series / 30 天精通RxJS

```
}  
findIndex(['a', 'b'], x => x === 'b'); // 找陣列中 'b' 的 index
```

这里我们写了一个findIndex用来找阵列中的元素位置，我们在 `findIndex` 中故意多塞了一个参数用来保存当前找到第几个index的**状态**，这就是利用参数保存状态！

这边用到了递归，递归会不断的呼叫自己，制造多层stack frame，会导致运算速度较慢，而这通常要靠编译器做优化！

那JS有没有做递归优化呢？恭喜大家，ES6提供了**尾呼优化(tail call optimization)**，让我们有一些手法可以让递归更有效率！

Functional Programming 优势

可读性高

当我们透过一系列的函式封装资料的操作过程，程式码能变得非常的简洁且可读性极高，例如下面的例子

```
[9, 4].concat([8, 7]) // 合併陣列  
    .sort() // 排序  
    .filter(x => x > 5) // 過濾出大於 5 的
```

可维护性高

因为Pure function 等特性，执行结果不依赖外部状态，且不会对外部环境有任何操作，使Functional Programming 能更好的除错及撰写单元测试。

易于并行/平行处理

Functional Programming 易于做并行/平行(Concurrency/Parallel)处理，因为我们基本上只做运算不碰I/O，再加上没有Side Effect 的特性，所以较不用担心deadlock 等问题。

今日小结



Series / 30 天精通RxJS

FP 的基本观念有助于我们在学习其他Library 更容易上手，也能使我们撰写出更好的程式码，希望各位读者有所收获，若有任何疑问欢迎在下方留言给我！

Tags

- JavaScript
- RxJS
- Functional Programming
- RxJS 30 Days

⬅️ Prev



Next ➡️