30 天精通 RxJS(21)：深入 Observable

2017年1月1日。6 分钟阅读

> 我们已经把绝大部分的operators 都介绍完了，但一直没有机会好好的解释Observable 的 operators 运作方式。

在系列文章的一开头是以阵列(Array)的operators(map, filter, concatAll) 作为切入点，让读者们在学习observable 时会更容易接受跟理解，但实际上observable 的operators 跟阵列的有很大的不同，主要差异有两点

1. 延迟运算
2. 渐进式取值

延迟运算

window 是一整个家族，总共有五个相关的operators

```
var source = Rx.Observable.from([1,2,3,4,5]);
var example = source.map(x => x + 1);
```

上面这段程式码因为Observable 还没有被订阅，所以不会真的对元素做运算，这跟阵列的操作不一样，如下

```
var source = [1,2,3,4,5];
var example = source.map(x => x + 1);
```

上面这段程式码执行完，example 就已经取得所有元素的返回值了。

延迟运算是Observable跟阵列最明显的不同，延迟运算所带来的优势在之前的文章也已经提过这里就不再赘述，因为我们还有一个更重要的差异要讲，那就是**渐进式取值**

渐进式取值

阵列的operators 都必须完整的运算出每个元素的返回值并组成一个阵列，再做下一个operator

```
                .filter(x => x % 2 === 0) // 這裡會運算並返回一個完整的陣列
                .map(x => x + 1) // 這裡也會運算並返回一個完整的陣列
```

上面这段程式码，相信读者们都很熟悉了，大家应该都有注意到 `source.filter(...)` 就会返回一整个新阵列，再接下一个operator又会再返回一个新的阵列，这一点其实在我们实作map跟filter时就能观察到

```
Array.prototype.map = function(callback) {
    var result = []; // 建立新陣列
    this.forEach(function(item, index, array) {
        result.push(callback(item, index, array))
    });
    return result; // 返回新陣列
}
```

每一次的operator 的运算都会建立一个新的阵列，并在每个元素都运算完后返回这个新阵列，我们可以用下面这张动态图表示运算过程

Observable operator 的运算方式跟阵列的是完全的不同，虽然Observable 的operator 也都会回传一个新的observable，但因为元素是渐进式取得的关系，所以每次的运算是一个元素运算到

```
                .filter(x => x % 2 === 0)
                .map(x => x + 1)

example.subscribe(console.log);
```

上面这段程式码运行的方式是这样的

1. 送出 1 到filter被过滤掉
2. 送出 2 到filter在被送到map转成 3 ，送到observer `console.log` 印出
3. 送出 3 到filter被过滤掉

每个元素送出后就是运算到底，在这个过程中不会等待其他的元素运算。这就是渐进式取值的特性，不知道读者们还记不记得我们在讲Iterator 跟Observer 时，就特别强调这两个Pattern 的共同特性是渐进式取值，而我们在实作Iterator 的过程中其实就能看出这个特性的运作方式

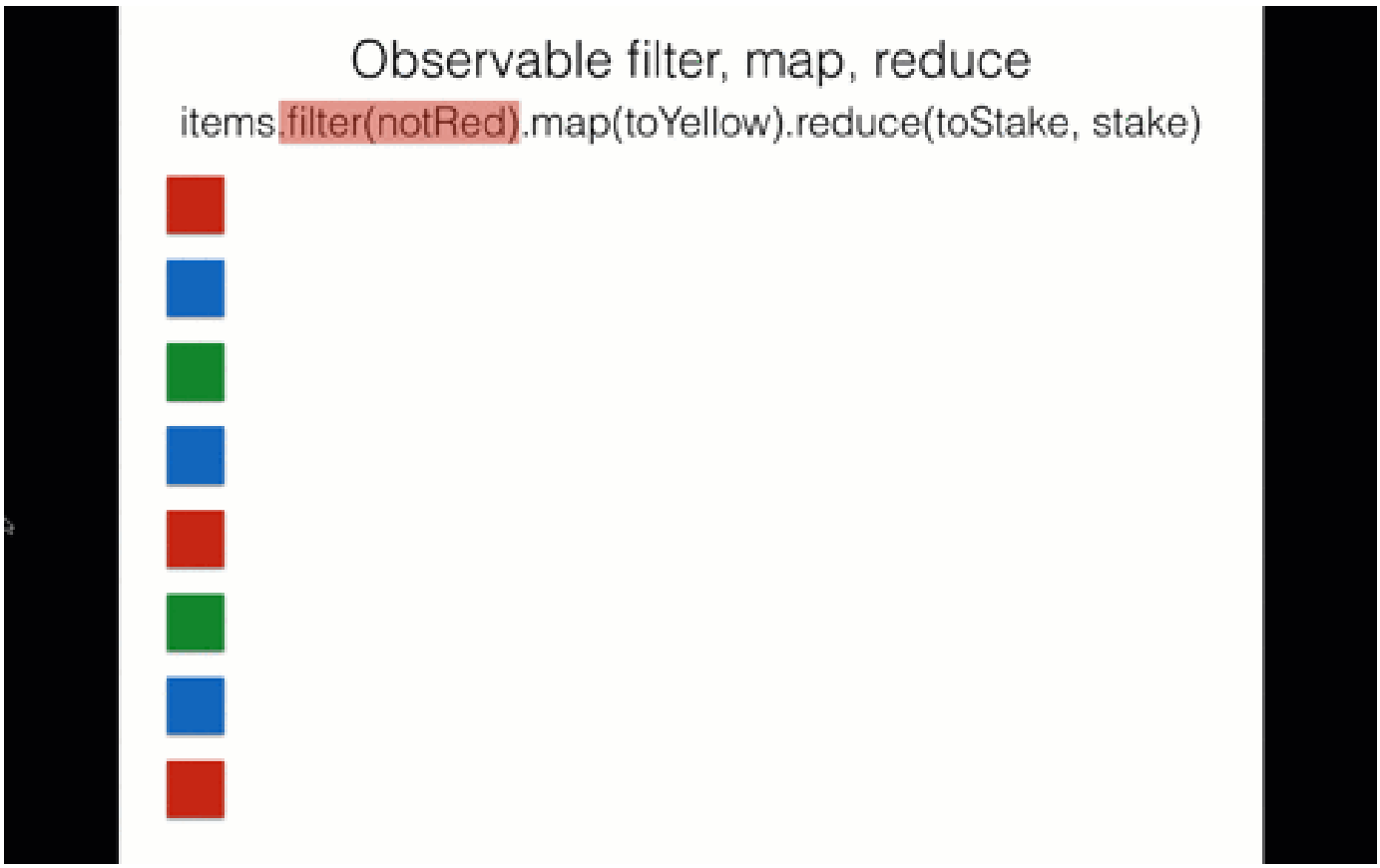
```
class IteratorFromArray {
        constructor(arr) {
                this._array = arr;
                this._cursor = 0;
        }

        next() {
                return this._cursor < this._array.length ?
                { value: this._array[this._cursor++], done: false } :
                { done: true };
        }

        map(callback) {
                const iterator = new IteratorFromArray(this._array);
                return {
                        next: () => {
                                const { done, value } = iterator.next()
                                return {
                                        done: done,
                                        value: done ? undefined : callb
                                }
                        }
```





```
var myIterator = new IteratorFromArray([1,2,3]);
var newIterator = myIterator.map(x => x + 1);
newIterator.next(); // { done: false, value: 2 }
```

虽然上面这段程式码是一个非常简单的示范，但可以看得出来每一次map 虽然都会返回一个新的oterator，但实际上在做元素运算时，因为渐进式的特性会使一个元素运算到底， Observable 也是相同的概念，我们可以用下面这张动态图表示运算过程



渐进式取值的观念在Observable 中其实非常的重要，这个特性也使得Observable 相较于Array 的operator 在做运算时来的高效很多，尤其是在处理大量资料的时候会非常明显！

今日小结

今天我们讲解了Observable 跟阵列各自operators 运作上的差异，这些细微的差异实际上对程式的运行效率有着很大的影响。

从我们一开始从阵列作为obsevable 的切入点，中间介绍了各种常用的operator，到今天我们厘清了阵列跟Observable 运作上的差异，在Observable 这块我们几乎已经完成了，剩下的是一些衍生出来的东西，像是multicast, publish... 等，这些我们会在介绍完Subject 后在做说明！

JavaScript RxJS 可观察的 运算符 RxJS 30天

上一页 下一个