30 天精通RxJS (04)：什么是Observable ?

Dec 18th, 2016 . 4 mins read

> 整个RxJS 的基础就是Observable，只要弄懂Observable 就算是学会一半的RxJS 了，剩下的就只是一些方法的练习跟熟悉；但到底什么是Observable 呢？

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

读者可能会很好奇，我们的主题是RxJS 为什么要特别讲Functional Programming 的通用函式呢？实际上，RxJS 核心的Observable 操作观念跟FP 的阵列操作是极为相近的，只学会以下几个基本的方法跟观念后，会让我们之后上手Observable 简单很多！

Observer Pattern

Observer Pattern 其实很常遇到，在许多API 的设计上都用了Observer Pattern 实作，最简单的例子就是DOM 物件的事件监听，程式码如下

```
function clickHandler(event) {
        console.log('user click!');
}

document.body.addEventListener('click', clickHandler)
```

在上面的程式码，我们先宣告了一个 `clickHandler` 函式，再用DOM物件(范例是body)的 `addEventListener` 来监听**点击** (click)事件，每次使用者在body点击滑鼠就会执行一次 `clickHandler` ，并把相关的资讯(event)带进来！这就是观察者模式，我们可以对某件事注册监听，并在事件发生时，自动执行我们注册的监听者(listener)。

Observer 的观念其实就这么的简单，但笔者希望能透过程式码带大家了解，如何实作这样的Pattern！

首先我们需要一个建构式，这个建构式new 出来的实例可以被监听。

> 这里我们先用ES5 的写法，会再附上ES6 的写法

```
            throw new Error('請用 new Producer()!');
            // 仿 ES6 行為可用：  throw new Error('Class constructor Produce
        }

        this.listeners = [];
}

// 加入監聽的方法
Producer.prototype.addListener = function(listener) {
        if(typeof listener === 'function') {
                this.listeners.push(listener)
        } else {
                throw new Error('listener 必須是 function')
        }
}

// 移除監聽的方法
Producer.prototype.removeListener = function(listener) {
        this.listeners.splice(this.listeners.indexOf(listener), 1)
}

// 發送通知的方法
Producer.prototype.notify = function(message) {
        this.listeners.forEach(listener => {
                listener(message);
        })
}
```

> 这里用到了this, prototype等观念，大家不了解可以去看我的一支 影片 专门讲解这几个观念！

附上ES6 版本的程式码，跟上面程式码的行为基本上是一样的

```
class Producer {
        constructor() {
                this.listeners = [];
```

```
                } else {
                        throw new Error('listener 必須是 function')
                }
        }
        removeListener(listener) {
                this.listeners.splice(this.listeners.indexOf(listener),
        }
        notify(message) {
                this.listeners.forEach(listener => {
                        listener(message);
                })
        }
}
```

有了上面的程式码后，我们就可以来建立物件实例了

```
var egghead = new Producer();
// new 出一個 Producer 實例叫 egghead

function listener1(message) {
        console.log(message + 'from listener1');
}

function listener2(message) {
        console.log(message + 'from listener2');
}

egghead.addListener(listener1); // 註冊監聽
egghead.addListener(listener2);

egghead.notify('A new course!!') // 當某件事情方法時，執行
```

当我们执行到这里时，会印出：

```
a new course!! from listener1
a new course!! from listener2
```

Iterator Pattern

Iterator 是一個物件，它的就像是一個指針(pointer)，指向一個資料結構並產生一個序列(sequence)，這個序列會有資料結構中的所有元素(element)。

先讓我們來看看原生的 JS 要怎麼建立 iterator

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

var iterator = arr[Symbol.iterator]();

iterator.next();
// { value: 1, done: false }
iterator.next();
// { value: 2, done: false }
iterator.next();
// { value: 3, done: false }
iterator.next();
// { value: undefined, done: true }
```

> JavaScript 到了 ES6 才有原生的 Iterator

> 在 ECMAScript 中 Iterator 最早其實是要採用類似 Python 的 Iterator 規範，就是 Iterator 在沒有元素之後，執行 `next` 會直接拋出錯誤；但後來經過一段時間討論後，決定採更 functional 的做法，改成在取得最後一個元素之後執行 `next` 永遠都回傳 `{ done: true, value: undefined }`

JavaScript 的 Iterator 只有一個 next 方法，這個 next 方法只會回傳這兩種結果：

1. 在最後一個元素前： `{ done: false, value: elem }`
2. 在最後一個元素之後： `{ done: true, value: undefined }`

當然我們可以自己實作簡單的 Iterator Pattern

```
function IteratorFromArray(arr) {
    if(!(this instanceof IteratorFromArray)) {
```

```
        this._cursor = 0;
}

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

附上 ES6 版本的程式碼，行為同上

```
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 };
        }
}
```

Iterator Pattern 雖然很單純，但同時帶來了兩個優勢，第一它漸進式取得資料的特性可以拿來做延遲運算(Lazy evaluation)，讓我們能用它來處理大資料結構。第二因為 iterator 本身是序列，所以可以實作所有陣列的運算方法像 map, filter... 等！

這裡我們利用最後一段程式碼實作 map 試試

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

        next() {
                return this._cursor < this._array.length ?
```





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

var iterator = new IteratorFromArray([1,2,3]);
var newIterator = iterator.map(value => value + 3);

newIterator.next();
// { value: 4, done: false }
newIterator.next();
// { value: 5, done: false }
newIterator.next();
// { value: 6, done: false }
```

補充: 延遲運算(Lazy evaluation)

延遲運算，或說 call-by-need，是一種運算策略(evaluation strategy)，簡單來說我們延遲一個表達式的運算時機直到真正需要它的值在做運算。

以下我們用 generator 實作 iterator 來舉一個例子

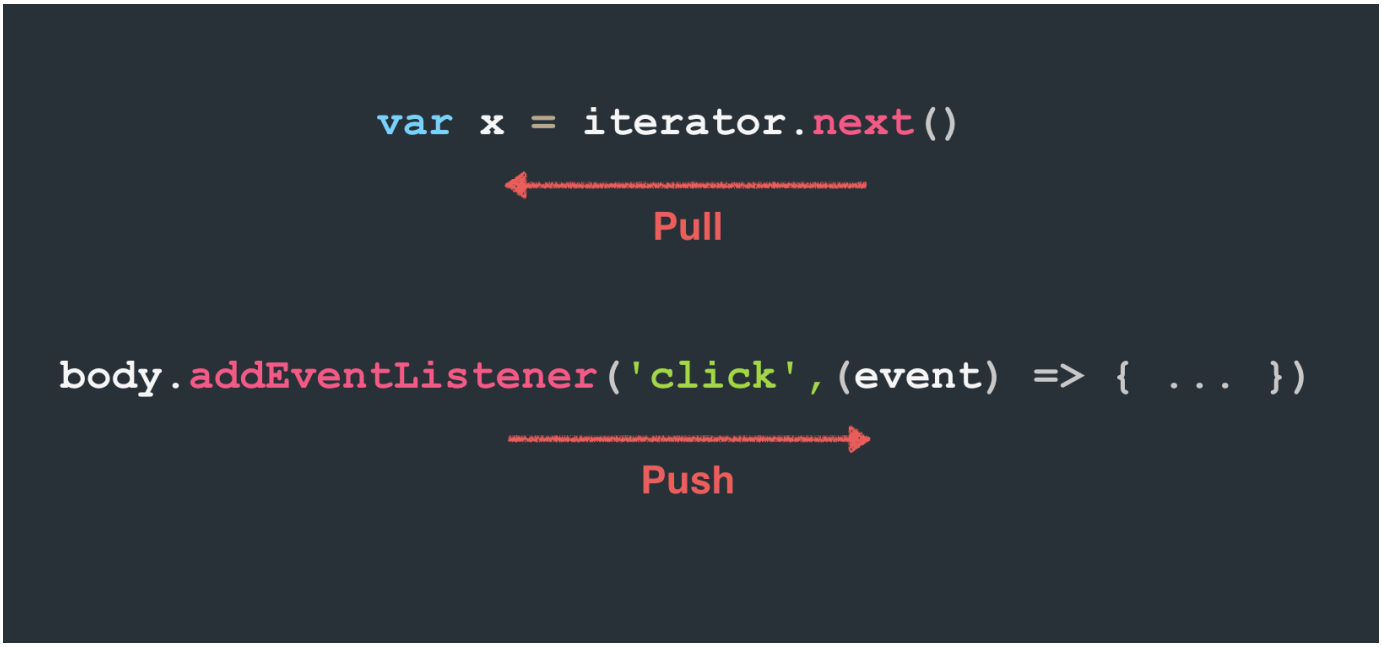
```
        function* getNumbers(words) {
                for (let word of words) {
                        if (/^[0-9]+$/.test(word)) {
                            yield parseInt(word, 10);
                        }
                }
```

```
iterator.next();
// { value: 3, done: false }
iterator.next();
// { value: 0, done: false }
iterator.next();
// { value: 0, done: false }
iterator.next();
// { value: 4, done: false }
iterator.next();
// { value: undefined, done: true }
```

這裡我們寫了一個函式用來抓取字串中的數字，在這個函式中我們用 for...of 的方式來取得每個字元並用正則表示式來判斷是不是數值，如果為真就轉成數值並回傳。當我們把一個字串丟進 `getNumbers` 函式時，並沒有馬上運算出字串中的所有數字，必須等到我們執行 `next()` 時，才會真的做運算，這就是所謂的延遲運算(evaluation strategy)

Observable

在了解 Observer 跟 Iterator 後，不知道大家有沒有發現其實 Observer 跟 Iterator 有個共通的特性，就是他們都是 **漸進式**(progressive) 的取得資料，差別只在於 Observer 是生產者(Producer)推送資料(push)，而 Iterator 是消費者(Consumer)要求資料(pull)!



Observable 其實就是這兩個 Pattern 思想的結合，Observable 具備**生產者推送資料**的特性，同

> 注意這裡講的是 **思想的結合**，Observable 跟 Observer 在實作上還是有差異，這我們在下一篇文章中講到。

今日小結

今天讲了Iterator 跟Observer 两个Pattern，这两个Pattern 都是渐进式的取得元素，差异在于Observer 是靠生产者推送资料，Iterator 则是消费者去要求资料，而Observable 就是这两个思想的结合！

今天的观念需要比较多的思考，希望读者能多花点耐心想一想，如果有任何问题请在下方留言给我。

Tags

JavaScript RxJS Observable Iterator Observer RxJS 30 Days

Prev Next