# 30 天精通RxJS (27)：简易实作 Observable(二)

Jan 11th, 2017 . 4 mins read

前一篇文章我们已经完成了基本的observable 以及Observer 的简易实作，这篇文章我们会接续上一篇的内容实作简易的Observable 类别，以及一个creation operator 和一个transform operator。

## 建立简易Observable 类别

这是我们上一篇文章写的建立observable 物件的函式

```
function create(subscribe) {
    const observable = {
        subscribe: function() {
            const realObserver = new Observer(...arguments);
            subscribe(realObserver);
            return realObserver;
        }
    };
    return observable;
}
```

JSBin

上面这段程式码就可以做最简单的订阅，像下面这样

```
class Observable {
  subscribe() {
    // ...做某些事
  }
}
```

observable

另外create 的函式在执行时会传入一个subscribe 的function，这个function 会决定observable 的行为

```
var observable = create(function(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
  observer.next('not work');
})
```

把上面这一段改成下面这样

```
var observable = new Observable(function(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
  observer.next('not work');
})
```

所以我们的Observable 的建构式应该会接收一个subscribe function

```
class Observable {
  constructor(subscribe) {
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到屬性中
    }
  }
  subscribe() {
    // ...做某些事
  }
}
```

接着我们就能完成subscribe 要做的事情了





```
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到 _subscribe 屬性中
    }
  }
  subscribe() {
    const observer = new Observer(...arguments);
    this._subscribe(observer); // 就是執行一個 function 對吧
    return observer;
  }
}
```

到这里我们就成功的把create 的函式改成Observable 的类别了，我们可以直接来使用看看

```
class Observable {
  constructor(subscribe) {
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到屬性中
    }
  }
  subscribe() {
    const observer = new Observer(...arguments);
    this._subscribe(observer);
    return observer;
  }
}

var observable = new Observable(function(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
  observer.next('not work');
})

var observer = {
  next: function(value) {
```

```
    }
  }

  observable.subscribe(observer);
```

JSBin

當然我们可以仿RxJS 在静态方法中加入create，如下

```
class Observable {
  constructor(subscribe) {
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到屬性中
    }
  }
  subscribe() {
    const observer = new Observer(...arguments);
    this._subscribe(observer);
    return observer;
  }
}

Observable.create = function(subscribe) {
    return new Observable(subscribe);
}
```

这样一来我们就可以用`Observable.create`建立observable物件实例。

```
var observable = Observable.create(function(observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
  observer.next('not work');
});
```

JSBin

**Series / 30 天精通RxJS**

当我们有Observable 类别后要建立creation operator 就不难了，这里我们建立一个fromArray 的方法，可以接收array 来建立observable，算是Rx 的Observable.from 的简化版本，记得 creation operators 都属于static 方法

```
class Observable {
  constructor(subscribe) {
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到屬性中
    }
  }
  subscribe() {
    const observer = new Observer(...arguments);
    this._subscribe(observer);
    return observer;
  }
}

// 建立靜態方法
Observable.fromArray = function(array) {
    if(!Array.isArray(array)) {
        // 如果傳入的參數不是陣列，則拋出例外
        throw new Error('params need to be an array');
    }
    return new Observable(function(observer) {
        try{
            // 遍歷每個元素並送出
            array.forEach(value => observer.next(value))
            observer.complete()
        } catch(err) {
            observer.error(err)
        }
    });
}

var observable = Observable.fromArray([1,2,3,4,5]);
```

JSBin

# 建立transform operator - map

相信很多人在实作Observable 都是卡在这个阶段，因为operators 都是回传一个新的observable 这中间有很多细节需要注意，并且有些小技巧才能比较好的实现，在开始实作之前先让我们厘清几个重点

- operators(transform, filter, conditional...) 都是回传一个新个observable
- 大部分的operator 其实就是在原本observer 外包裹一层物件，让执行next 方法前先把元素做一次处理
- operator 回传的observable 订阅时，还是需要执行原本的observable(资料源)，也就说我们要想办法保留原本的observable

让我们一步一步来，首先operators 执行完会回传一个新的observable，这个observable 在订阅时会先去执行operator 的行为再发送元素，所以observable 的订阅方法就不能像现在这样直接把observer 传给subscribe执行

```
class Observable {
  constructor(subscribe) {
    if(subscribe) {
      this._subscribe = subscribe; // 把 subscribe 存到屬性中
    }
  }
  subscribe() {
    const observer = new Observer(...arguments);
    // 先做某個判斷是否當前的 observable 是具有 operator 的
    if(??) {
      // 用 operator 的操作
    } else {
      // 如果沒有 operator 再直接把 observer 丟給 _subscribe
      this._subscribe(observer);
    }
    return observer;
  }
}
```

事要做

- 建立新的observable
- 保存原本的observable(资料源)，之后订阅时才有办法执行
- 建立并保存operator 本身的行为，等到订阅时执行

```
class Observable {
  constructor(subscribe) {
    // 一些程式碼...
  }
  subscribe() {
    // 一些程式碼...
  }
  map(callback) {
    const observable = new Observable(); // 建立新的 observable

    observable.source = this; // 保存當前的 observable(資料源)

    observable.operator = {
        call: (observer, source) => { // 執行這個 operator 的行為 }
    }; // 儲存當前 operator 行為，並作為是否有 operator 的依據，

    return observable; // 返回這個新的 observable
  }
}
```

上面这三个步骤都是必要的，特别是用到了 `observable.source = this` 这个小技巧，来保存原本的observable。但这里我们还有一个地方没完成就是operator要做的事，这个部分我们等一下再补，先把subscribe写完

```
class Observable {
  constructor(subscribe) {
    // 一些程式碼...
  }
  subscribe() {
    const observer = new Observer(...arguments);
    // 先用 this.operator 判斷當前的 observable 是否具有 operator
```





```
      this._subscribe(observer);
    }
    return observer;
  }
  map(callback) {
    const observable = new Observable(); // 建立新的 observable

    observable.source = this; // 保存當前的 observable(資料源)

    observable.operator = {
        call: (observer, source) => { // 執行這個 operator 的行為 }
    }; // 儲存當前 operator 行為，並作為是否有 operator 的依據，

    return observable; // 返回這個新的 observable
  }
}
```

记得这里补的subscribe 行为，已经是map 回传新observable 的行为，不是原本的observable了。

到这里我们就几乎要完成了，接着只要实作map 这个operator 的行为就可以啰！记得我们在前面讲的operator 其实就是在原本的observer 做一层包裹，让next 执行前先对元素做处理，所以我们改写一下Observer 并建立一个MapObserver 来做这件事

```
class Observer {
  constructor(destinationOrNext, error, complete) {
    switch (arguments.length) {
      case 0:
        this.destination = this.safeObserver(emptyObserver);
        break;
      case 1:
        if (!destinationOrNext) {
          this.destination = this.safeObserver(emptyObserver);
          break;
        }
        // 多一個判斷，是否傳入的 destinationOrNext 原本就是 Observer 的實例，
        if(destinationOrNext instanceof Observer){
          this.destination = destinationOrNext;
```





```
            break;
          }
        default:
          this.destination = this.safeObserver(destinationOrNext, error,
          break;
      }
    }

    // ...下面都一樣
  }

  class MapObserver extends Observer {
    constructor(observer, callback) {
      // 這裡會傳入原本的 observer 跟 map 的 callback
      super(observer); // 因為有繼承所以要先執行一次父層的建構式
      this.callback = callback; // 保存 callback
      this.next = this.next.bind(this); // 確保 next 的 this
    }
    next(value) {
      try {
        this.destination.next(this.callback(value));
        // this.destination 是父層 Observer 保存的 observer 物件
        // 這裡 this.callback(value) 就是 map 的操作
      } catch (err) {
        this.destination.error(err);
        return;
      }
    }
  }
```

上面这段程式码就可以让我们包裹observer 物件，利用物件的继承覆写原本的next 方法。

最后我们就只要补完map 方法就可以了

```
class Observable {
  constructor(subscribe) {
    // 一些程式碼...
```

```
  map(callback) {
    const observable = new Observable();
    observable.source = this;
    observable.operator = {
      call: (observer, source) => {
        // 執行這個 operator 的行為
        const newObserver = new MapObserver(observer, callback);
        // 建立包裹後的 observer
        // 訂閱原本的資料源，並回傳
        return source.subscribe(newObserver);
      }
    };
    return observable;
  }
}
```

这里做的事情就简单很多，我们只要建立包裹过的observer，并用这个包裹后的observer 订阅原本的source。(记得这个function 是在subscribe 时执行的)

这里有 完整的程式码，可以让大家参考。

另外这里有抽出lift 方法的实作，其实跟我们现在的版本很接近了，只是把建立新的observable 封装到lift 而已。

# 今日小结

今天这篇文章介绍了要如何简易的实作Observable，虽然说是简易版本但实际上已经非常非常接近RxJS 官方的实作了，希望读者花点耐心一步一步跟着程式码做，做出来后再慢慢吸收。

不知道今天读者们有没有收获呢？如果有任何问题，欢迎在下方留言给我，谢谢！

Tags

JavaScript RxJS Observable Observer RxJS 30 Days

Prev Next

## Series / 30 天精通RxJS