Rxjs 简易实现

大纲

- 📝 什么是 rxjs
- 🎨 rxjs 的编程范式
- 🤖 rxjs 的设计模式
- 👷 rxjs 的基本概念
- 📹 Observable简单实现
- 📩 手写一些 operator
- 🛠️ QA

业务代码

```
// DOMRouter.ts
const routerInfoSubject = new BehaviorSubject<RouterInfo | null>(null);

routerInfoSubject.next({
  pathname: relativePath,
  query,
  layerIdx,
  state,
});

DOMRouter.routeInfo$.subscribe((routerInfo) => {
  if (routerInfo.query.type === 'success') {
    this.isHide = true;
  }
});
```

什么是 rxjs

RxJS 是一套借由 Observable sequences 来组合非同步行为和事件基础程序的 Library

> Think of RxJS as Lodash for events.

这也被称为 Functional Reactive Programming，更切确地说是指 Functional Programming 及 Reactive Programming 两个编程思想的结合。

rxjs 的编程范式 - Functional Programming

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。

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
const mul = (a, b) ⇒ a * b
const sub = (a, b) ⇒ a - b

sub(add(5, 6), mul(1, 3))
```

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。

rxjs 的编程范式 - Reactive Programming

Reactive Programming 简单来说就是 当变数或资源发生变动时，由变数或资源自动告诉我发生变动了

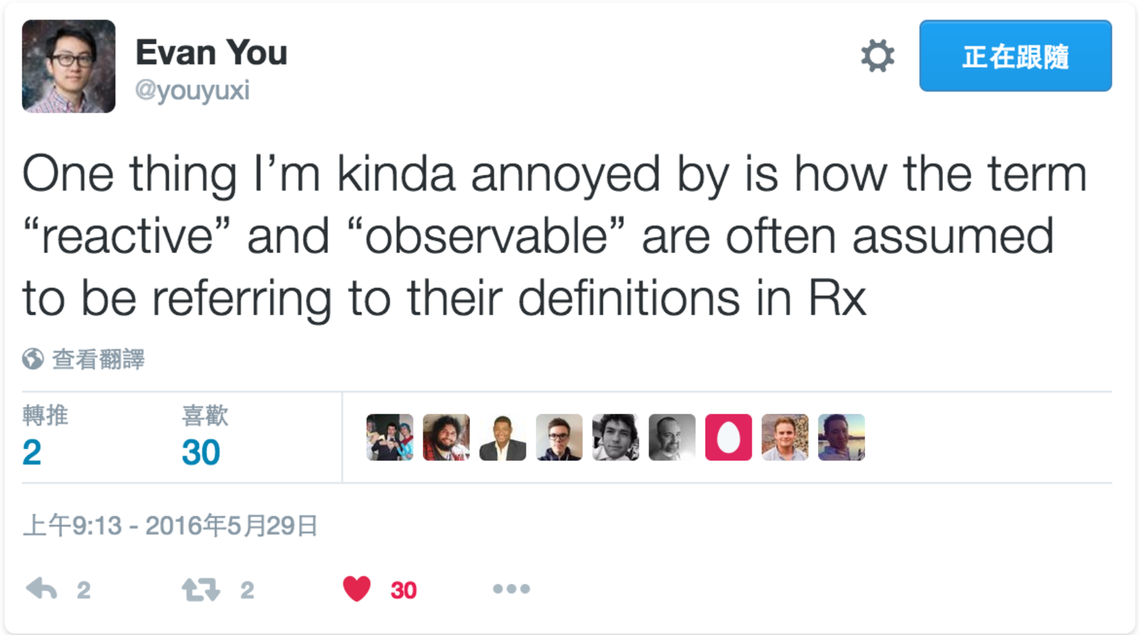
这句话看似简单，其实背后隐含两件事

1. 当发生变动 ⇒ 非同步：不知道什么时候会发生变动，反正变动时要跟我说
2. 由变数自动告知我 ⇒ 我不用写通知我的每一步代码

rxjs 的编程范式 - Reactive Programming

很多人一谈到 Reactive Programming 就会直接联想到是在讲 RxJS，但实际上 Reactive Programming 仍是一种编程范式，在不同的场景都有机会遇到，而非只存在于 RxJS。

尤雨溪(Vue 的作者)就曾在 twitter 对此表达不满！



比如 Vue.js 底层就是用 Reactive Programming 的概念实例

设计模式（Observer Pattern）

谈谈两个设计模式(Design Pattern)， Iterator Pattern 跟 Observer Pattern

Observer Pattern(观察者模式)

在许多 API 的设计上都用了 Observer Pattern 实例，最简单的例子就是 DOM 事件的事件监听，代码如下

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

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

设计模式（Iterator Pattern）

Iterator 是一个事件，它的就像是一个指针(pointer)，指向一个数据结构并产生一个序列(sequence)，这个序列会有数据结构中的所有元素(element)。

```
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 } */
```

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

  // 一个数据结构只要具有Symbol.iterator属性，就可以认为是“可遍历
  [Symbol.iterator]() {
    return this
  }

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

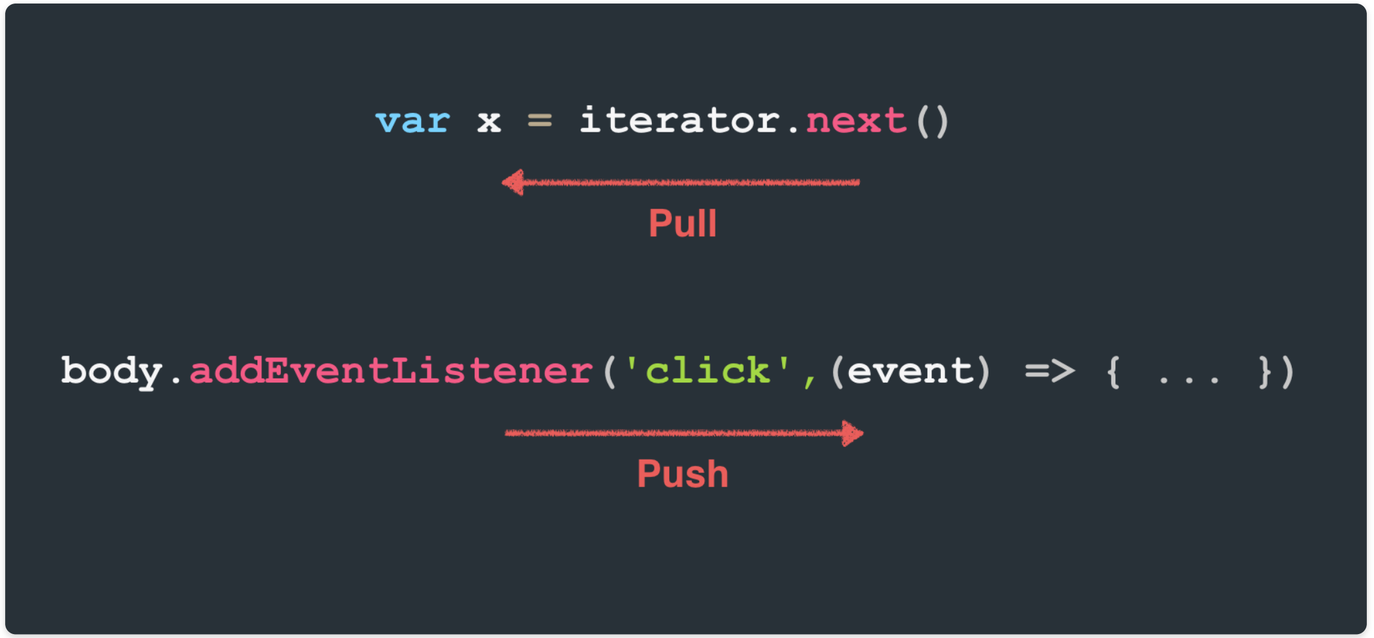
Iterator Pattern

Iterator Pattern 虽然很简单，但同时带来了两个优势，

1. 渐进式取得数据的特性可以拿来做延迟运算(Lazy evaluation)，让我们能用它来处理大数据结构。
2. iterator 本身是序列，所以可以实例所有数组的运算方法像 map, filter... 等！

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

    }
```

```
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 }
```



Observable 其实就是这两个 Pattern（模式） 思想的结合

1. Observable 具备生产者推送数据的特性

2. 同时能像序列，拥有序列处理数据的方法(map, filter...)!

rxjs 的基本概念

1. observable ➡️ 数据源
2. observer ➡️ 消费者
3. operator ➡️ 加工数据源

Observer：有3种callback，只关心如何处理数据

```
const observer = {
  next: (value) ⇒ console.log(value),
  error: (error) ⇒ console.log(error),
  complete: () ⇒ console.log('completed')
}
```

observable 就像个豌豆射手，不断推送数据

```
const x$ = (observer) ⇒ {
  observer.next('I\'m the 1st go');
  observer.next('I\'m the 2nd go');
  // observer.complete('complete');
  observer.error('something went wrong');
}
```

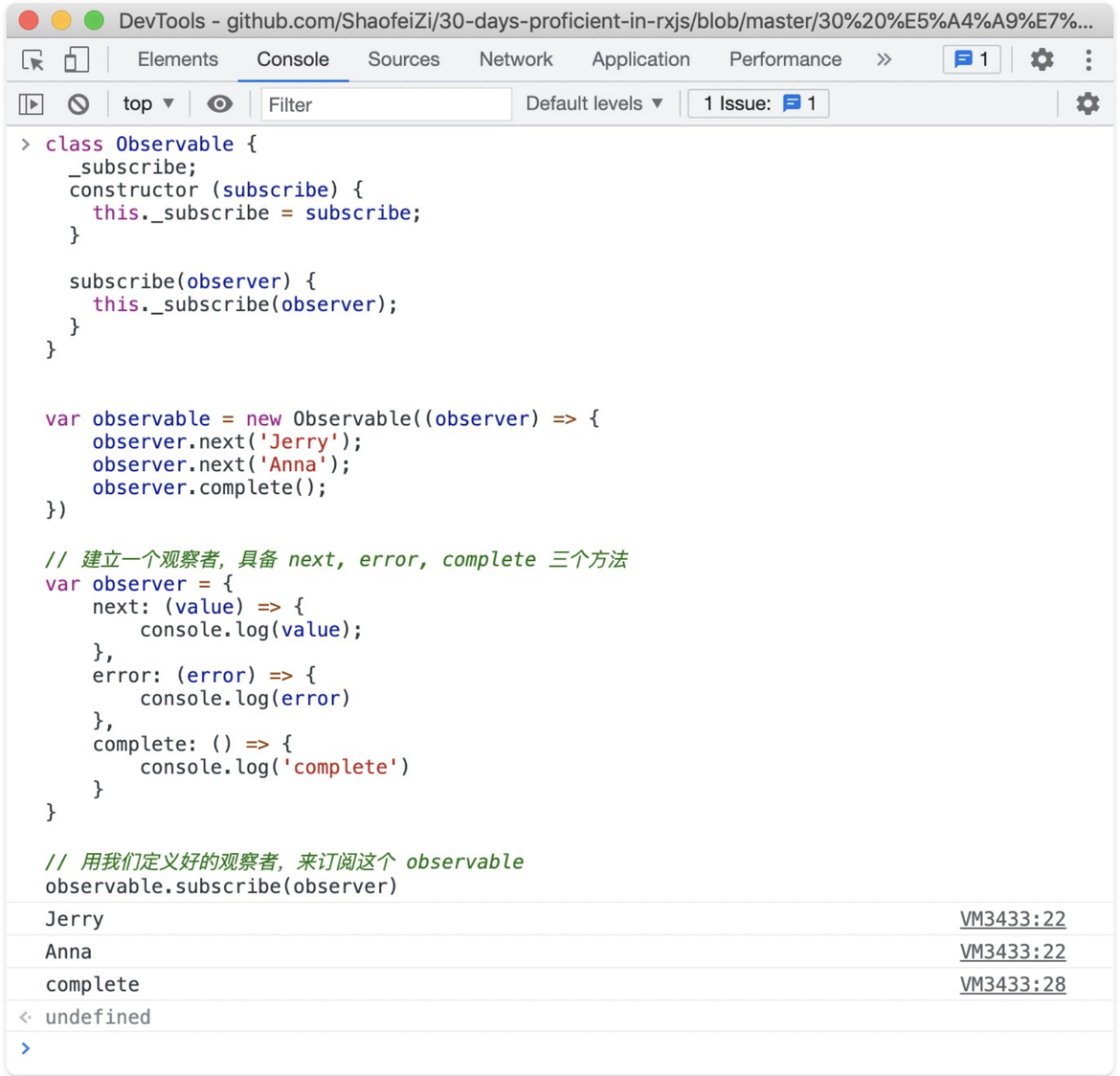
一个例子

```
import { Observable } from 'rxjs';

var observable = new Observable((observer) => {
    observer.next('Jerry');
    observer.next('Anna');
    observer.complete();
})

// 建立一个观察者，具备 next, error, complete 三个方法
var observer = {
    next: (value) => {
        console.log(value);
    },
    error: (error) => {
        console.log(error)
    },
    complete: () => {
        console.log('complete')
    }
}

// 用我们定义好的观察者，来订阅这个 observable
observable.subscribe(observer)
```

实现一个Observable

```
class Observable {
  _subscribe;
  constructor (subscribe) {
    this._subscribe = subscribe;
  }

  subscribe(observer) {
    this._subscribe(observer);
  }
}
```

运行


DevTools - github.com/ShaofeiZi/30-days-proficient-in-rxjs/blob/master/30%20%E5%A4%A9%E7%...
Elements
Console
Sources
Network
Application
Performance
top
Filter
Default levels
1 Issue: 1
class Observable {
_subscribe;
constructor (subscribe) {
this._subscribe = subscribe;
}
subscribe(observer) {
this._subscribe(observer);
}
}
var observable = new Observable((observer) => {
observer.next('Jerry');
observer.next('Anna');
observer.complete();
})
// 建立一个观察者，具备 next, error, complete 三个方法
var observer = {
next: (value) => {
console.log(value);
},
error: (error) => {
console.log(error)
},
complete: () => {
console.log('complete')
}
}
// 用我们定义好的观察者，来订阅这个 observable
observable.subscribe(observer)
Jerry
VM3433:22
Anna
VM3433:22
complete
VM3433:28
undefined

手写操作符 - 实现创建类操作符 of

```
const demo$ = of(1,2,3);
```

实现代码

```
export function of( ...args) {
  return new Observable(observer => {
    args.forEach(arg => {
      observer.next(arg);
    })
    observer.complete();
    return {
      unsubscribe: () => { }
    }
  })
}
```

手写操作符 - 实现创建类操作符 fromEvent

```
export function fromEvent(element, event) {
  return new Observable(observer ⇒ {
    const handler = e ⇒ observer.next(e);
    element.addEventListener(event, handler);
  });
}
```

```
// 添加unsubscribe功能
export function fromEvent(element, event) {
  return new Observable(observer ⇒ {
    const handler = e ⇒ observer.next(e);
    element.addEventListener(event, handler);
    return {
      unsubscribe: () ⇒ element.removeEventListener(event, handler)
    };
  });
}
```

手写操作符 - 实现转换类操作符 map

js实现map

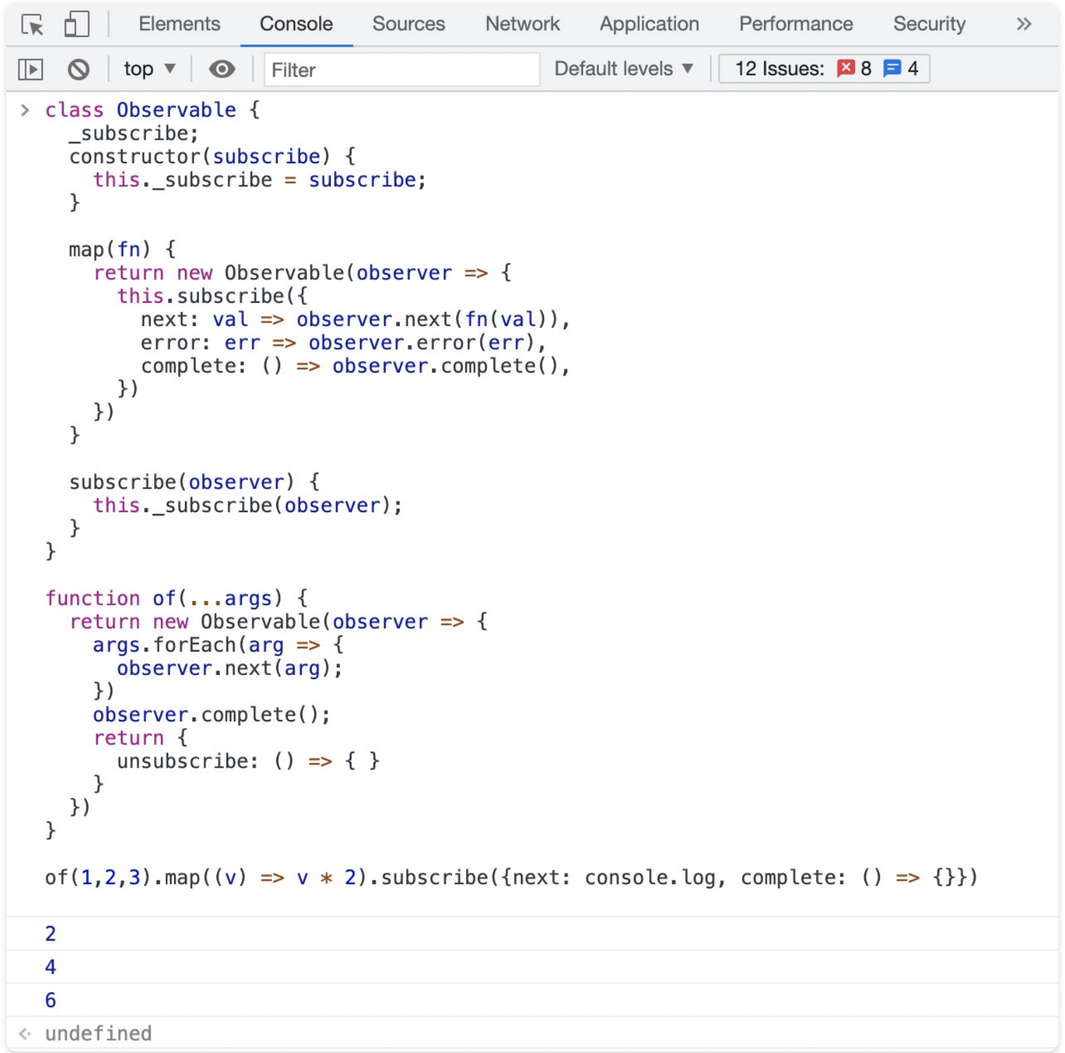
```
function map(array,fn) {
  const res = [];
  array.forEach(item => res.push(fn(item)))
  return res;
}
```

rxjs链式调用map

```
const dataStream1$ = of(1, 2, 3);

dataStream1$
  .map(data => data * 2)
  .filter(data => data > 3)
  .map(data => data + 1)
  .subscribe(console.log)
```

map代码

```
export class Observable {
  _subscribe;
  constructor(subscribe) {
    this._subscribe = subscribe;
  }

  map(fn) {
    return new Observable(observer => {
      this.subscribe({
        next: val => observer.next(fn(val)),
        error: err => observer.error(err),
        complete: () => observer.complete(),
      })
    })
  }

  subscribe(observer) {
    this._subscribe(observer);
  }
}
```



filter

```
export class Observable {
  _subscribe;
  constructor(subscribe) {
    this._subscribe = subscribe;
  }

  filter(fn) {
    return new Observable(observer => {
      this.subscribe({
        next: val => fn(val) ? observer.next(val) : () => { },
        error: err => observer.error(err),
        complete: () => observer.complete(),
      })
    })
  }

  subscribe(observer) {
    this._subscribe(observer);
  }
}
```

Question

observable 是可以多次订阅的

如果我们希望第二次订阅 observable 不会从头开始接收元素，而是从第一次订阅到当前处理的元素开始发送?

Subject的特点

Subject是一个Observable，因为它有subscribe方法；Subject又是一个Observer，因为它有next方法。

可以建立一个中间人来订阅 source 再由中间人转送资料出去，就可以达到我们想要的效果

Subject

我们把这种处理方式称为组播(multicast)，那我们要如何做到组播呢?

```
class Subject {
  _subscribe
  observers = []
  constructor(subscribe) {
    this._subscribe = subscribe
  }

  subscribe(observer) {
    if (typeof observer === 'function') {
      this.observers.push({ next: observer })
    } else {
      this.observers.push(observer)
    }
  }

  next(val) {
    this.observers.forEach((ob) => {
      ob.next(val)
    })
  }
}
```

```
const subject = new Subject()

const a$ = new Observable((observer) => {
  observer.next(1);
  setTimeout(() => {
    observer.next(2);
  }, 2000)
})

subject.subscribe({ next: console.log })
a$.subscribe(subject)

setTimeout(() => {
  subject.subscribe({ next: (val) => {
    console.log(val, 'second')
  } })
}, 1000)

// output:
// 1
// 2


// 2 second
```

业务代码

现在知道是如何实现的吗?

```
// BehaviorSubject 是 Subject 的变体之一。BehaviorSubject 的特性就是它会存储“当前”的值。
// 这意味着你始终可以直接拿到 BehaviorSubject 最后一次发出的值。

// DOMrouter.ts
const routerInfoSubject = new BehaviorSubject<RouterInfo | null>(null);

routerInfoSubject.next({
  pathname: relativePath,
  query,
  layerIdx,
  state,
});

DOMRouter.routeInfo$.subscribe((routerInfo) => {
  if (routerInfo.query.type === 'success') {
    this.isHide = true;
  }
});
```