# <https://segmentfault.com/a/1190000008464065>

# [通俗的方式理解RxJS](https://segmentfault.com/a/1190000008464065)

* [前端](https://segmentfault.com/t/%E5%89%8D%E7%AB%AF/blogs)
* [javascript](https://segmentfault.com/t/javascript/blogs)
* [rxjs](https://segmentfault.com/t/rxjs/blogs)

[**mrcode**](https://segmentfault.com/u/mrcode) 2017年02月23日发布

* **21.8k** 次浏览

## 通俗的方式理解Rx.js

### 序言

今早看[民工叔](https://zhuanlan.zhihu.com/p/25383159)的文章的时候， 发现对Rxjs所知甚少， 于是去官方看了下教程， 整理出一些东西， 写成此文。  
Rxjs据说会在2017年流行起来， 因为其处理异步逻辑，数据流， 事件非常擅长。 但是其学习曲线相比Promise， EventEmitter陡峭了不少。 而且民工叔也说:"由于RxJS的抽象程度很高，所以，可以用很简短代码表达很复杂的含义，这对开发人员的要求也会比较高，需要有比较强的归纳能力。" 本文就Rx.js的几个核心概念做出阐述。 尽可能以通俗易懂的方式解释这些概念。要是本文有误或不完善的地方，欢迎指出。

### Observable到底是什么

先上代码:

let foo = Rx.Observable.create(observer => {

console.log('Hello');

observer.next(42);

});

foo.subscribe(x => console.log(x));

foo.subscribe(y => console.log(y));

输出

"Hello"

42

"Hello"

42

这里可以把foo想象成一个函数，**这意味着你每次调用foo都会导致传入Rx.Observable.create里的回调函数重新执行一次**, 调用的方式为foo.subscribe(callback), 相当于foo()。 接收函数返回值的方式也从var a = foo()改为通过传入回调函数的方式获取。第三行的observer.next表示返回一个值, 你可以调用多次，**每次调用observer.next后， 会先将next里的值返回给foo.subcribe里的回调函数, 执行完后再返回**。observer.complete, observer.error来控制流程。 具体看代码:

var observable = Rx.Observable.create(observer => {

try {

observer.next(1);

console.log('hello');

observer.next(2);

observer.next(3);

observer.complete();

observer.next(4);

} catch (err) {

observer.error(err);

}

});

let = subcription = observable.subscribe(value => {

console.log(value)

})

运行结果：

1

hello

2

3

如上的第一个回调函数里的结构是推荐的结构。 当observable的执行出现异常的时候，通过observer.error将错误返回, 然而observable.subscribe的回调函数无法接收到.因为observer.complete已经调用, 因此observer.next(4)的返回是无效的. **Observable不是可以返回多个值的Promise** 虽然获得Promise的值的方式也是通过then函数这种类似的方式, 但是new Promise(callback)里的callback回调永远只会执行一次！因为**Promise的状态是不可逆的**。

可以使用其他方式创建Observable， 看代码：

var clicks = Rx.Observable.fromEvent(document, 'click');

clicks.subscribe(x => console.log(x));

当用户对document产生一个click行为的时候， 就会打印事件对象到控制台上。

### Observer是什么

先看代码：

let foo = Rx.Observable.create(observer => {

console.log('Hello');

observer.next(42);

});

let observer = x => console.log(x);

foo.subscribe(observer);

代码中的第二个变量就是observer. 没错， observer就是**当Observable"返回"值的时候接受那个值的函数!**第一行中的observer其实就是通过foo.subscribe传入的callback. 只不过稍加封装了。 怎么封装的？ 看代码：

let foo = Rx.Observable.create(observer => {

try {

console.log('Hello');

observer.next(42);

observer.complete();

observer.next(10);

} catch(e) { observer.error(e) }

});

let observer = {

next(value) { console.log(value) },

complete() { console.log('completed'),

error(err) { console.error(err) }

}

foo.subscribe(observer);

你看到observer被定义成了一个对象， 其实这才是完整的observer. 传入一个callback到observable.subcribe相当于传入了{ next: callback }。

### Subcription里的陷阱

Subscription是什么， 先上代码：

var observable = Rx.Observable.interval(1000);

var subscription = observable.subscribe(x => console.log(x));

setTimeout(() => {

subscription.unsubscribe();

}, 3100)

运行结果：

0

1

2

Rx.Observable.interval可以返回**一个能够发射(返回)0， 1， 2， 3...， n数字的Observable**， 返回的时间间隔这里是1000ms。 第二行中的变量就是subscription。 subscription有一个unsubscribe方法, 这个方法可以让**subscription订阅的observable发射的数据被observer忽略掉**.通俗点说就是取消订阅。

unsubscribe存在一个陷阱。 先看代码：

var foo = Rx.Observable.create((observer) => {

var i = 0

setInterval(() => {

observer.next(i++)

console.log('hello')

}, 1000)

})

const subcription = foo.subscribe((i) => console.log(i))

subcription.unsubscribe()

运行结果：

hello

hello

hello

......

hello

unsubscribe只会让observer忽略掉observable发射的数据，但是setInterval依然会继续执行。 这看起来似乎是一个愚蠢的设计。 所以不建议这样写。

### Subject

**Subject是一种能够发射数据给多个observer的Observable**, 这让Subject看起来就好像是EventEmitter。 先上代码：

var subject = new Rx.Subject();

subject.subscribe({

next: (v) => console.log('observerA: ' + v)

});

subject.subscribe({

next: (v) => console.log('observerB: ' + v)

});

subject.next(1);

subject.next(2);

运行结果：

observerA: 1

observerB: 1

observerA: 2

observerB: 2

与Observable不同的是， Subject发射数据给多个observer。 其次， 定义subject的时候并没有传入callback， 这是因为subject自带next, complete, error等方法。从而可以发射数据给observer。 这和EventEmitter很类似。observer并不知道他subscribe的是Obervable还是Subject。 对observer来说是透明的。 而且Subject还有各种派生， 比如说：

BehaviorSubject 能够保留最近的数据，使得当有subscribe的时候，立马发射出去。看代码：

var subject = new Rx.BehaviorSubject(0); // 0 is the initial value

subject.subscribe({

next: (v) => console.log('observerA: ' + v)

});

subject.next(1);

subject.next(2);

subject.subscribe({

next: (v) => console.log('observerB: ' + v)

});

subject.next(3);

运行结果：

observerA: 0

observerA: 1

observerA: 2

observerB: 2

observerA: 3

observerB: 3

ReplaySubject 能够保留最近的一些数据， 使得当有subscribe的时候，将这些数据发射出去。看代码：

var subject = new Rx.ReplaySubject(3);

subject.subscribe({

next: (v) => console.log('observerA: ' + v)

});

subject.next(1);

subject.next(2);

subject.next(3);

subject.next(4);

subject.subscribe({

next: (v) => console.log('observerB: ' + v)

});

subject.next(5);

输出结果：

observerA: 1

observerA: 2

observerA: 3

observerA: 4

observerB: 2

observerB: 3

observerB: 4

observerA: 5

observerB: 5

第一行的声明表示ReplaySubject最大能够记录的数据的数量是3。

AsyncSubject 只会发射结束前的一个数据。 看代码：

var subject = new Rx.AsyncSubject();

subject.subscribe({

next: (v) => console.log('observerA: ' + v)

});

subject.next(1);

subject.next(2);

subject.next(3);

subject.next(4);

subject.subscribe({

next: (v) => console.log('observerB: ' + v)

});

subject.next(5);

subject.complete();

输出结果：

observerA: 5

observerB: 5

既然subject有next, error, complete三种方法， 那subject就可以作为observer！ 看代码：

var subject = new Rx.Subject();

subject.subscribe({

next: (v) => console.log('observerA: ' + v)

});

subject.subscribe({

next: (v) => console.log('observerB: ' + v)

});

var observable = Rx.Observable.from([1, 2, 3]);

observable.subscribe(subject);

输出结果：

observerA: 1

observerB: 1

observerA: 2

observerB: 2

observerA: 3

observerB: 3

也就是说， observable.subscribe可以传入一个subject来订阅其消息。 这就好像是Rxjs中的一颗语法糖， Rxjs有专门的实现。

Multicasted Observables 是一种借助Subject来将数据发射给多个observer的Observable。 看代码：

var source = Rx.Observable.from([1, 2, 3]);

var subject = new Rx.Subject();

var multicasted = source.multicast(subject);

multicasted.subscribe({

next: (v) => console.log('observerA: ' + v)

});

multicasted.subscribe({

next: (v) => console.log('observerB: ' + v)

});

multicasted.connect();

Rx.Observable.from能够逐一发射数组中的元素， 在multicasted.connect()调用之前的任何subscribe都不会导致source发射数据。multicasted.connect()相当于之前的observable.subscribe(subject)。因此不能将multicasted.connect()写在subscribe的前面。因为这会导致在执行multicasted.connect()的时候source发射数据， 但是subject又没保存数据， 导致两个subscribe无法接收到任何数据。

最好是第一个subscribe的时候能够得到当前已有的数据， 最后一个unsubscribe的时候就**停止Observable的执行**， 相当于Observable发射的数据都被忽略。

refCount就是能够返回这样的Observable的方法

var source = Rx.Observable.interval(500);

var subject = new Rx.Subject();

var refCounted = source.multicast(subject).refCount();

var subscription1, subscription2, subscriptionConnect;

console.log('observerA subscribed');

subscription1 = refCounted.subscribe({

next: (v) => console.log('observerA: ' + v)

});

setTimeout(() => {

console.log('observerB subscribed');

subscription2 = refCounted.subscribe({

next: (v) => console.log('observerB: ' + v)

});

}, 600);

setTimeout(() => {

console.log('observerA unsubscribed');

subscription1.unsubscribe();

}, 1200);

setTimeout(() => {

console.log('observerB unsubscribed');

subscription2.unsubscribe();

}, 2000);

输出结果：

observerA subscribed

observerA: 0

observerB subscribed

observerA: 1

observerB: 1

observerA unsubscribed

observerB: 2

observerB unsubscribed

### What's Operators?

Observable上有很多方法， 比如说map, filter, merge等等。 他们基于调用它们的observable，返回一个**全新的observable**。 而且他们都是纯方法。 operators分为两种， instance operators 和 static operators。 instance operators是存在于observable实例上的方法， 也就是实例方法； static operators是存在于Observable这个类型上的方法， 也就是静态方法。Rxjs拥有很多强大的[operators](http://reactivex.io/rxjs)。

自己实现一个operators：

function multiplyByTen(input) {

var output = Rx.Observable.create(function subscribe(observer) {

input.subscribe({

next: (v) => observer.next(10 \* v),

error: (err) => observer.error(err),

complete: () => observer.complete()

});

});

return output;

}

var input = Rx.Observable.from([1, 2, 3, 4]);

var output = multiplyByTen(input);

output.subscribe(x => console.log(x));

输出结果：

10

20

30

40

### Rx.js实践

import React from 'react';

import ReactDOM from 'react-dom';

import Rx from 'rx';

class Main extends React.Component {

constructor (props) {

super(props);

this.state = {count: 0};

}

// Click events are now observables! No more proactive approach.

componentDidMount () {

const plusBtn = document.getElementById('plus');

const minusBtn = document.getElementById('minus');

const plus$ = Rx.Observable.fromEvent(plusBtn, 'click').map(e => 1);

const minus$ = Rx.Observable.fromEvent(minusBtn, 'click').map(e => -1);

Rx.Observable.merge(plus$, minus$).scan((acc, n) => acc + n)

.subscribe(value => this.setState({count: value}));

}

render () {

return (

<div>

<button id="plus">+</button>

<button id="minus">-</button>

<div>count: {this.state.count}</div>

</div>

);

}

}

ReactDOM.render(<Main/>, document.getElementById('app'));

merge用于合并两个observable产生一个新的observable。 scan类似于Array中的reduce。 [这个例子](https://github.com/belfz/fully-reactive-react-example)实现了点击plus的时候+1， 点击minus的时候-1。

### Rx.js适用的场景

* 多个复杂的异步或事件组合在一起。
* 处理多个数据序列

假如没有被复杂的异步，事件， 数据序列困扰， 如果promise已经足够的话， 就没必要适用Rx.js。

### Summary

* Observable， Observer， Subscription， Subscriber， Subject概念。
* RxJS适用于解决复杂的异步，事件问题。

<https://juejin.im/entry/57e9175cda2f600060de7dcf>

这是一篇 RxJS 初学者教程。

**What Is RxJS**

通过**阅读**[**官方文档**](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html)，不难得出：RxJS 可以很好**解决异步和事件组合的问题**。

这个时候我就有疑问了，异步问题不是用 Promise ( async/await ) 就好了吗？  
至于事件，配合框架 ( React, Vue, Angular2 等 ) 的话不也很容易解决吗？

不管怎样, 让我们先看个 Hello World 吧。( 我要看 [DEMO](https://link.juejin.im?target=http%3A%2F%2Fjsbin.com%2Fsuguho%2Fedit%3Fjs%2Coutput) )

**Rx's Hello World**

// auto-complete

const Observable = Rx.Observable

const input = document.querySelector('input')

const search$ = Observable.fromEvent(input, 'input')

.map(e => e.target.value)

.filter(value => value.length >= 1)

.throttleTime(100)

.distinctUntilChanged()

.switchMap(term => Observable.fromPromise(wikiIt(term)))

.subscribe(

x => renderSearchResult(x),

err => console.error(err)

)

上面的代码做了以下事情：

* 监听 input 元素的 input 事件
* 一旦发生，把事件对象 e 映射成 input 元素的值
* 接着过滤掉值长度小于 1 的
* 并且还设置了一个 throttle ( 节流器 )，两次输入间隔不超过 100 毫秒为有效输入
* 如果该值和过去最新的值相等的话，忽略他
* 最后，拿到值便调用 Wikipedia 的一个 API
* 最后的最后，需要 subscribe 才能拿到 API 返回的数据

是不是看起来就觉得很 cool ，好想学！  
短短几行代码就完成了一个 auto-complete 组件。

**How It Works**

那上面的代码是什么意思？  
RxJS 到底是如何工作的？如何解决异步组合问题的？

**Observable**

Rx 提供了一种叫 **Observable** 的数据类型，兼容 ECMAScript 的 [Observable Spec Proposal](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fzenparsing%2Fes-observable) 草案标准。他是 Rx 最核心的数据类型，结合了 [Observer Pattern](https://link.juejin.im?target=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FObserver_pattern)，[Iterator Pattern](https://link.juejin.im?target=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FIterator_pattern) 。

那到底什么是 Observable ？

Observable 其实就是一个**异步的数组**。*( --->* [*2 minute introduction to rx*](https://link.juejin.im?target=https%3A%2F%2Fmedium.com%2F%40andrestaltz%2F2-minute-introduction-to-rx-24c8ca793877%23.1q1q2mwgq) *)*

不妨想像一下，**数组 + 时间轴 = Observable** 。

数组元素的值是未来某个时间点 *emit* ( 产生 ) 的，但是我们并不关心这个时间点，因为利用了「观察者模式」*subscribe* ( 订阅 ) 了这个数组，只要他 *emit* 了值，就会自动 *push* 给我们。

我们再用图来表示一下的话：

--a---b-c--d-----e--|-->

这种图叫做 [marble diagram](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23marble-diagrams) 。  
我们可以把 ASCII 的 marble 图转成 SVG 的：[ASCII -> SVG](https://link.juejin.im?target=https%3A%2F%2Fjsbin.com%2Fvumepol%2F1%2Fedit%3Fjs%2Coutput) 。

- 表示时间轴，a ~ e 表示 emit 的值，| 则表示这个 stream 已经结束了。  
比方说，click 事件用上图来表示：a 表示第 1 次点击，b 表示第 2 次点击，如此类推。

如果你觉得 Observable 这个名字不够形象不够 cool 的话，你可把他叫做 [stream](https://link.juejin.im?target=https%3A%2F%2Fgist.github.com%2Fstaltz%2F868e7e9bc2a7b8c1f754%23reactive-programming-is-programming-with-asynchronous-data-streams) ，因为他的 marble 图就像 steam 一样。所以啊，下面我都会把 *Observable* 称作 *stream* 。

**Operators**

那么，我们怎么对 stream 进行操作呢？怎么把多个 stream 组合在一起呢？

我们前面不是说了「 Observable 其实就是*异步数组*」吗？在 JavaScript 里的数组不是有很多内置的方法吗？比如 map, filter, reduce 等等。类似地，Observable 也有自己的方法，也就是所谓的 [operator](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23operators) 。比如上面 [Rx's Hello World](https://link.juejin.im?target=https%3A%2F%2Ffe.ele.me%2Flet-us-learn-rxjs%2F%23rx-s-hello-world) 例子中的 map, filter, throttleTime, distinctUntilChanged 等等很多很有用的 operator 。

面对 RxJS 那么多 operator ，我们要怎么学习呢？很简单：

[分类别](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23categories-of-operators) + [画 marble 图](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23marble-diagrams) + [看例子](https://link.juejin.im?target=https%3A%2F%2Fwww.learnrxjs.io%2Foperators%2F) + [选](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23choose-an-operator)

现在，就让我们画出上面 Hello World 例子的 marble 图。

const search$ = Observable.fromEvent(input, 'input')

.map(e => e.target.value)

.filter(value => value.length >= 1)

.throttleTime(100)

.distinctUntilChanged()

.switchMap(term => Observable.fromPromise(wikiIt(term)))

.subscribe(

x => renderSearchResult(x),

err => console.error(err)

)

假设输入了 5 次，每次输入的值一次为：a, ab, c, d, c ，并且第 3 次输入的 c 和第 4 次的 d 的时间间隔少于 100ms ：

---i--i---i-i-----i---|--> (input)

map

---a--a---c-d-----c---|-->

b

filter

---a--a---c-d-----c---|-->

b

throttleTime

---a--a---c-------c---|-->

b

distinctUntilChanged

---a--a---c----------|-->

b

switchMap

---x--y---z----------|-->

如果我告诉你学习 **RxJS 的捷径是「学会看和画 marble 图」**，你信还是不信？

**Learn By Doing**

现在，就让我们结合上面的知识，来实现一个简单的 canvas 画板。

根据 canvas 的 [API](https://link.juejin.im?target=http%3A%2F%2Fdevdocs.io%2Fdom%2Fcanvasrenderingcontext2d%2Fmoveto) ，我们需要知道两个点的坐标，这样才能画出一条线。

**Step 1**

*( 我要看* [*DEMO*](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fa7h4wwcy%2F) *)*

那么，现在我们需要做的是**创建**一个关于鼠标移动的 stream 。于是，我们**去文档找对应的 operator 类别**，也就是 [Creation Operators](https://link.juejin.im?target=http%3A%2F%2Fjsbin.com%2Fgakosun%2Fedit%3Fjs%2Cconsole%2Coutput) ，然后得到 [fromEvent](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23static-method-fromEvent) 。

const canvas = document.querySelector('canvas')

const move$ = Rx.Observable.fromEvent(canvas, 'mousemove')

对应的 marble 图：

--m1---m1-m2--m3----m4---|--> (mousemove)

接着，我们需要拿到每次鼠标移动时的坐标。也就是说：需要**变换** stream 。  
对应类别的 operator 文档：[Transformation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23transformation-operators) ---> [map](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-map) 。

const move$ = Rx.Observable.fromEvent(canvas, 'mousemove')

.map(e => ({ x: e.offsetX, y: e.offsetX }))

此时的 marble 图：

--m1---m2-m3--m4----m5---|--> (mousemove)

map

--x1---x2-x3--x4----x5---|--> (点坐标)

然后，怎么拿到两个点的坐标呢？我们需要再**变换**一下 stream 。  
对应类别的 operator 文档：[Transformation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23transformation-operators) ---> [bufferCount](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-bufferCount) 。

const move$ = Rx.Observable.fromEvent(canvas, 'mousemove')

.map(e => ({ x: e.offsetX, y: e.offsetY }))

.bufferCount(2)

marble 图：

--m1---m2-m3--m4----m5---|--> (mousemove)

map

--x1---x2-x3--x4----x5---|--> (点坐标)

bufferCount(2)

-------x1-----x3----x5---|---> (两点坐标)

x2 x4

然而你会发现，此时画出来的[线段是不连续的](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fa7h4wwcy%2F)。为什么？我也不知道！！  
那就让我们看看别人是怎么写的吧：[canvas paint](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FReactive-Extensions%2FRxJS%2Fblob%2Fmaster%2Fexamples%2Fcanvaspaint%2Fcanvaspaint.js) 。

**Step 2**

*( 先让我要看看* [*DEMO*](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F1%2F) *)*

换了一种思路，并没有**变换** stream ，而是把两个 stream **组合**在一起。  
查看文档 [Combination Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23combination-operators) ---> [zip](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23static-method-zip) 以及 [Filtering Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23filtering-operators) ---> [skip](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-skip)

const move$ = Rx.Observable.fromEvent(canvas, 'mousemove')

.map(e => ({ x: e.offsetX, y: e.offsetY }))

const diff$ = move$

.zip(move$.skip(1), (first, sec) => ([ first, sec ]))

此时的 marble 图：

--x1---x2-x3--x4----x5---|--> (move$)

skip(1)

-------x2-x3--x4----x5---|-->

--x1---x2-x3--x4----x5---|--> (move$)

-------x2-x3--x4----x5---|-->

zip

-------x1-x2--x3----x4---|--> (diff$)

x2 x3 x4 x5

这样一来，diff$ emit 的值就依次为 (x1, x2), (x2, x3)，(x3, x4) …… 现在，鼠标移动的时候，就可以[画出美丽的线条](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F1%2F)。

**Step 3**

*( 我想看* [*DEMO*](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F2%2F) *)*

就在此时我恍然大悟，终于知道前面用 bufferCount 为什么不行了。我们不妨来比较一下：

-------x1-----x3----x5---|---> (bufferCount)

x2 x4

-------x1-x2--x3----x4---|--> (diff$)

x2 x3 x4 x5

bufferCount emit 的值依次为：(x1, x2)， (x3, x4) …… x2 和 x3 之间是有间隔的。这就是为什么线段会不连续的原因。

然后看 [bufferCount](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-bufferCount) 文档的话，你会发现**可以使用 bufferCount(2, 1) 实现同样的效果**。这样的话，我们就不需要使用 zip 来组合两个 stream 了。Cool ~

const move$ = Rx.Observable.fromEvent(canvas, 'mousemove')

.map(e => ({ x: e.offsetX, y: e.offsetX }))

.bufferCount(2, 1)

此时的 marble 图：

--m1---m2-m3--m4----m5---|--> (mousemove)

map

--x1---x2-x3--x4----x5---|--> (点坐标)

bufferCount(2, 1)

-------x1-x2--x3----x4---|---> (两点坐标)

x2 x3 x4 x5

**Step 4**

*( 我就要看* [*DEMO*](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F3%2F) *)*

接下来，我们想实现「只有鼠标按下时，才能画画，否则不能」。  
首先我们需要**创建**两个关于鼠标动作的 stream 。

const down$ = Rx.Observable.fromEvent(canvas, 'mousedown')

const up$ = Rx.Observable.fromEvent(canvas, 'mouseup')

当鼠标按下的时候，我们需要把他**变换**成鼠标移动的 stream ，直到鼠标放开。  
查看文档 [Transformation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23transformation-operators) ---> [switchMapTo](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-switchMapTo) 。

down$.switchMapTo(move$)

此时的 marble 图：

--d---d-d-----d---d--|--> (mousedown)

switchMapTo

--m---m-m-----m---m--|-->

此时，鼠标放开了我们还能[继续画画](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F3%2F)，这显然不是我们想要的。这个时候我们很容易会使用 [takeUntil](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-takeUntil) 这个 operator ，但是这是不对的，因为他会把 *stream* complete 掉。

还是让我们看看别人是怎么写的吧：[canvas paint](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FReactive-Extensions%2FRxJS%2Fblob%2Fmaster%2Fexamples%2Fcanvaspaint%2Fcanvaspaint.js) 。

**Step 5**

*( 我只想看* [*DEMO*](https://link.juejin.im?target=https%3A%2F%2Fjsfiddle.net%2FDrakeLeung%2Fvj368qy7%2F4%2F) *)*

思路是这个样子的：

把 up$ 和 down$ **组合**成一个新的 stream ，但为了分辨他们，我们需要先把他们**变换**成新的 stream 。  
查看文档 [Combination Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23combination-operators) ---> [merge](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-merge) 。  
[Transformation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23transformation-operators) ---> [map](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-map) 。

const down$ = Rx.Observable.fromEvent(canvas, 'mousedown')

.map(() => 'down')

const up$ = Rx.Observable.fromEvent(canvas, 'mouseup')

.map(() => 'up')

const upAndDown$ = up$.merge(down$)

再来看看他们的 marble 图：

--d--d-d----d--d---|--> (down$)

----u---u-u------u-|--> (up$)

merge

--d-ud-du-u-d--d-u-|--> (upAndDown$)

此时，我们再**变换** upAndDown$ 。如果是 down 的话，则变换成 move$ ，否则变换成一个空的 stream 。  
查看文档 [Creation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23categories-of-operators) ---> [empty](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23static-method-empty) 。  
[Transformation Operators](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fmanual%2Foverview.html%23transformation-operators) ---> [switchMap](https://link.juejin.im?target=http%3A%2F%2Freactivex.io%2Frxjs%2Fclass%2Fes6%2FObservable.js~Observable.html%23instance-method-switchMap) 。

upAndDown$

.switchMap(action =>

action === 'down' ? move$ : Rx.Observable.empty()

)

你要的 marble 图：

--d-ud-du-u-d--d-u-|--> (upAndDown$)

switchMap

--m-em-me-e-m--m-e-|-->

其实这个 canvas 画板不用 RxJS 实现也不会很难。但是当我们把他扩展成一个「你画我猜」之后，用 RxJS 处理异步就会变得简单起来。比如，添加新的工具栏 ( 调色板，撤销…… ) ，即时通信 ( 同步画板，聊天 ) ……

另外，如果你想边学习 RxJS 边实现一些小东西的话：

**Production**

怎么把 RxJS 应用到实际生产的 web 应用当中呢？  
怎么结合到当前流行的框架当中呢？

**Vue**

你可以直接在各种 [Lifecycle Hooks](https://link.juejin.im?target=http%3A%2F%2Fvuejs.org%2Fapi%2F%23Options-Lifecycle-Hooks) 中使用 RxJS 。

比如 created 的时候初始化一个 Observable ，beforeDestroy 时就取消订阅 Observable 。( 查看 [DEMO](https://link.juejin.im?target=http%3A%2F%2Fjsbin.com%2Fcafodu%2Fedit%3Fhtml%2Cjs%2Coutput) )

new Vue({

el: '#app',

data: {

time: ''

},

created () {

this.timer$ = Rx.Observable.interval(1000)

.map(() => new Date())

.map(d => moment(d).format('hh:mm:ss'))

.subscribe(t => {

this.time = t

})

},

beforeDestroy () {

this.timer$.unsubscribe()

}

})

其实已经有对应的插件 [vue-rx](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fvuejs%2Fvue-rx) 帮我们干了上面的 dirty work 。他会分别在 init 和 beforeDestroy 的时候自动地订阅和取消订阅 Observable ：[Vue.js + RxJS binding mixin in 20 lines](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fvuejs%2Fvue-rx%2Fblob%2Fmaster%2Fvue-rx.js%23L22-L51) 。

因此，我们可以直接把一个 Observable 写到 data 中：[vue-rx/example.html](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fvuejs%2Fvue-rx%2Fblob%2Fmaster%2Fexample%2Fexample.html%23L26-L65) 。

**React**

类似地，React 也可以在他组件的 [lifecycle hooks](https://link.juejin.im?target=https%3A%2F%2Ffacebook.github.io%2Freact%2Fdocs%2Fcomponent-specs.html%23lifecycle-methods) 里调用 RxJS：[fully-reactive-react](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fbelfz%2Ffully-reactive-react-example) 。 也可以使用 [rxjs-react-component](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fchristianalfoni%2Frxjs-react-component) 把 Observable 绑定到 state 。 如果你结合 Redux 的话，可以使用这个 [redux-oservable](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fredux-observable%2Fredux-observable) 。

**Angular2**

RxJS 已经是 Angular2 的标配，不多说。  
更多可查看对应的文档 [Angular2 - Server Communication](https://link.juejin.im?target=https%3A%2F%2Fangular.io%2Fdocs%2Fts%2Flatest%2Fguide%2Fserver-communication.html%23!%23rxjs) 。

更多关于 RxJS 的集成：[RxJS community](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FReactive-Extensions%2FRxJS%2Fblob%2Fmaster%2Fexamples%2Fcommunity.md) 。

**You Might Not Need RxJS**

根据 [When to Use RxJS](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2FReactive-Extensions%2FRxJS%2Fblob%2Fmaster%2Fdoc%2Fdesignguidelines%2Freadme.md%2321-use-rxjs-for-orchestrating-asynchronous-and-event-based-computations) ，我们可以知道 RxJS 的适用场景是：

* 多个复杂的异步或者事件组合在一起
* 处理多个数据序列（有一定顺序）

我觉得，如果你没被异步问题困扰的话，那就不要使用 RxJS 吧，因为 Promise 已经能够解决简单的异步问题了。至于 Promise 和 Observable 的区别是什么呢？可以看 [Promise VS Observable](https://link.juejin.im?target=https%3A%2F%2Fwww.google.com.hk%2Fsearch%3Fq%3Dpromise%2Bvs%2Bobservable) 。

讲真，**RxJS 在实际生产中适用的业务场景有哪些**？哪些场景是需要多个异步组合在一起的？游戏吗？即时通信？还有一些特殊的业务。是我的写的业务太少了吗？还是我平时写业务的时候，为写而写，没有把他们抽象起来。

另外，我倒是对 Teambition 关于 RxJS 的思路有点感兴趣：[*xufei - 数据的关联计算 -> Brooooooklyn 评论*](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fxufei%2Fblog%2Fissues%2F36%23issuecomment-246662343) & [xufei - 对当前单页应用的技术栈思考](https://link.juejin.im?target=https%3A%2F%2Fgithub.com%2Fxufei%2Fblog%2Fissues%2F37)。

让我们一起来学习 RxJS 吧!