https://zhuanlan.zhihu.com/p/240287653

阅读完这篇文章，可以帮助你收获这些知识：

* RxJS 是如何构建出一个**响应式**编程库的？
* RxJS 是如何构建出一个**函数式**编程库的？
* 深入 RxJS 的核心概念
* 深入 RxJS 的操作符实现
* 深入 RxJS 的异常处理
* 探索 RxJS 的测试、文档和调试

如果你还不熟悉 RxJS，这篇文章可以很好帮助你认识和了解 RxJS，如果你是有一定 RxJS 使用经验的开发者，那么就和笔者一起更深入的掌握 RxJS 吧！

## **一、RxJS 是如何构建出一个响应式编程库的**

### **理解响应式**

在正式讨论 RxJS 之前，我们需要先明确**响应式编程**到底指的是什么，staltz 在 [The Introduction to Reactive Programming you've been missing](https://link.zhihu.com/?target=https://gist.github.com/staltz/868e7e9bc2a7b8c1f754" \t "https://zhuanlan.zhihu.com/p/_blank) 这篇文章里认为 **Reactive programming is programming with asynchronous data streams （响应式编程就是使用异步数据流进行编程）。**

如果你和我一样刚开始读到这句话时会心存疑惑，那不妨来看看响应式编程最著名的库 Reactive Extensions（后续简称 Rx） 的起源。微软的 DevLabs 在设计 Rx 时就是为了处理异步的数据，这些异步的数据可能来源于网络中的异步请求、UI 的点击事件等，这些都是异步的，例如 A = B + C，在同步执行编程模型下，A 的值可以立即被计算出来，如果 B 与 C 都是未来才可以确定的，那么 A 的值必须**响应** B 和 C 的变化，或者说 A 的值必须响应 B 数据流和 C 数据流的变化。

响应式编程里一个非常重要的概念就是流，可以说响应式编程的世界就是流的世界，而 RxJS （Reactive Extensions for JavaScript）作为一个响应式编程库，核心自然就是对流的控制，那么在这一节里，我们一起探索 RxJS 是如何**操作流**的。

### **操作流**

这是一段传统的原生 JS 写法，点击 document，打印 “Clicked”。

document.addEventListener('click', () => console.log('Clicked!'));

用 RxJS 改写这段代码：

import { fromEvent } from 'rxjs';

fromEvent(document, 'click').subscribe(() => console.log('Clicked!'));

fromEvent 会将 document 上的 click 事件转为一个点击事件流，并订阅它，这样也可以实现点击 document 打印 "Clicked"。那么我们就来看看 RxJS 内部是如何将普通的点击事件转换为点击事件流的。

我们查看 RxJS 关于这部分的源码：

**fromEvent.ts**

export function fromEvent<T>(

target: FromEventTarget<T>,

eventName: string,): Observable<T> {

// fromEvent 会返回一个 Observable

return new Observable<T>(subscriber => {

function handler(e: T) {

if (arguments.length > 1) {

subscriber.next(Array.prototype.slice.call(arguments) as any);

} else {

subscriber.next(e);

}

}

setupSubscription(

target, eventName, handler, subscriber, options as EventListenerOptions);

});}

function setupSubscription<T>(

sourceObj: FromEventTarget<T>,

eventName: string,

handler: (...args: any[]) => void,

subscriber: Subscriber<T>,

options?: EventListenerOptions) {

let unsubscribe: (() => void) | undefined;

// ... const source = sourceObj;

sourceObj.addListener(eventName, handler as NodeEventHandler);

unsubscribe = () => source.removeListener(eventName, handler as NodeEventHandler);

// ... subscriber.add(unsubscribe);

}

**Observable.ts**

export class Observable<T> implements Subscribable<T> {

// ... // Observable 将传入的函数存为内部

constructor(subscribe?: (this: Observable<T>, subscriber: Subscriber<T>) => TeardownLogic) {

if (subscribe) {

this.\_subscribe = subscribe;

}

}

// ...}

可以看到 fromEvent 会返回 Observable 实例，并且 Observable 的构造函数会接收一个函数类型的参数作为 Observable 对象内部的 subscribe，而点击事件需要执行的操作则会被包裹在 Subscription 内，每次点击事件不会直接触发回调函数，而是执行 subscriber.next()，这样 RxJS 就将点击事件转换为流了。

现在我们继续为上例中的点击事件添加 throttle 功能。

原生 JS 的写法：

**c**onst rate = 1000;

let lastClick = Date.now() - rate;

document.addEventListener('click', () => {

// 1000 ms 内只会触发一次 if (Date.now() - lastClick >= rate) {

console.log(`Clicked`);

lastClick = Date.now();

}});

而 RxJS 可以通过 throttleTime 操作符声明式的控制流：

import { fromEvent } from 'rxjs';import { throttleTime } from 'rxjs/operators';

fromEvent(document, 'click')

.pipe(

// 在 pipe 当中添加操作符即可

throttleTime(1000)

)

.subscribe(count => console.log(`Clicked`));

查看 pipe 的源代码：

export function pipe(...fns: Array<UnaryFunction<any, any>>): UnaryFunction<any, any> {

return pipeFromArray(fns);

}

/\*\* @internal \*/

export function pipeFromArray<T, R>(fns: Array<UnaryFunction<T, R>>): UnaryFunction<T, R> {

if (fns.length === 0) { return identity as UnaryFunction<any, any>; }

if (fns.length === 1) { return fns[0]; }

return function piped(input: T): R {

return fns.reduce((prev: any, fn: UnaryFunction<T, R>) => fn(prev), input as any);

};}

简单理解就是 pipe 中传入的函数会被一层层包裹起来，例如 pipe(fn1, fn2, fn3) 内部 reduce 后会组合成类似于 fn3(fn2(fn1(...))) 的效果。

接着查看 throttleTime ，源码中 lift 函数会在后面的 **深入 RxJS 操作符实现** 中介绍，这里我们主要查看 ThrottleTimeSubscriber 的实现：

**c**lass ThrottleTimeSubscriber<T> extends Subscriber<T> {

protected \_next(value: T) {

// ... // 满足条件后 this.destination.next(value);

// ... }

我们只需要关注两点：

1. ThrottleTimeSubscriber 会继承 Subscriber，从而拥有 Subscriber 的方法和属性
2. 在满足条件的时候将流继续向后传递

**小结：**

* 响应式编程就是使用异步数据流进行编程，响应式编程的世界就是流的世界。
* RxJS 通过引入和实现 Observable, Subscription, Observer 和 Operators 等概念来达到生成和操作流的效果。

## **二、RxJS 是如何构建出一个函数式编程库的**

要点：

* 理解函数式
* 声明式的代码（Declarative）
* 减少副作用（Side Effects）
* 无参数风格（Point-free）
* 不可变性（Immutability）
* 函子
* 函数组合

### **理解函数式编程**

在 JS 中学习和理解函数式编程，笔者推荐使用**轻量级**的方式，“monad 是一个自函子范畴上的幺半群” 这种说法并不会帮助你和你的组员在函数式编程交流上和平时的 code review 中提供太多的帮助，如果你是精通 HasKell 和函数式编程的高手，就当没看到这段话 [狗头]。

我会从下面这几个部分来谈谈 RxJS 是如何实现一个函数式编程库的：

* 声明式的代码 （Declarative）
* 纯函数（Pure Function）与减少副作用 （Side Effect）
* 无参数风格（Point-free）
* 不可变性（Immutability）
* 函子（Functor）
* 组合（Compose）

### **声明式的代码**

在上一小节的例子中，笔者提到 RxJS 中的节流操作是通过**声明式**添加 throttleTime 操作符的方式，而声明式的编写代码是函数式编程的特点之一。

同样以点击按钮节流和每次计数 + 1 为例：

**原生 JS 命令式的代码：**

let count = 0;

const rate = 1000;

let lastClick = Date.now() - rate;

document.addEventListener('click', event => {

if (Date.now() - lastClick >= rate) {

count += event.clientX;

console.log(count);

lastClick = Date.now();

}});

**RxJS 声明式的代码：**

import { fromEvent } from 'rxjs';

import { throttleTime, map, scan } from 'rxjs/operators';

fromEvent(document, 'click')

.pipe(

throttleTime(1000),

map(event => event.clientX),

scan((count, clientX) => count + clientX, 0)

)

.subscribe(count => console.log(count));

通过对比，我们不难发现 RxJS 声明式的代码可读性更高，可维护性也更高，数据流的组织也更清晰。其他人在阅读这段代码时也能很快掌握含义，这种思维方式也是函数式编程的重要思维方式之一。

### **纯函数与减少副作用**

我们举一个简单的例子来理解副作用：

let x = 1;

let y = 2;

let z = calc(x, y);

function add(a: number, b: number): number {

return a + b;

}

z; // 3

let x = 1;

let y = 2;

let z = 0;

function add(a: number, b: number): number {

z = a + b;}

z; // 3

上面两段代码都可以正确的计算出 z 的值，但是两段代码的可维护性却相差很多，区别就在于在第二段代码中，直接修改了外部的 z ，如果这段代码稍微复杂一点，那么可读性和可维护性会降低更多，这种情况我们就可以理解为一种函数的**副作用**，而没有副作用的函数我们就可以称为**纯函数**。或者从另一方面来说：“**给定相同的输入（一个或多个），它总是产生相同的输出**”。

介绍完纯函数与副作用，我们来看看 RxJS 是如何在运算符中减少副作用并保持尽可能“纯”的。

首先在源码的组织结构里，每个操作符都会书写在单独的文件中，并且这些操作符之间几乎不会引用其它操作符（除了功能上的复用之外）。

fromEvent(document, 'click')

.pipe(

throttleTime(1000),

map(event => event.clientX),

scan((count, clientX) => count + clientX, 0)

)

.subscribe(count => console.log(count));

其次在操作符的内部计算中，RxJS 也会尽量避免代码里出现对外部变量的引用，以操作符 map和 filter 为例：

// filter.ts 核心代码

class FilterSubscriber<T> extends Subscriber<T> {

count: number = 0;

constructor(

destination: Subscriber<T>,

private predicate: (value: T, index: number) => boolean,

private thisArg: any

) {

super(destination);

}

protected \_next(value: T) {

let result: any;

try {

result = this.predicate.call(this.thisArg, value, this.count++);

} catch (err) {

this.destination.error(err);

return;

}

if (result) {

this.destination.next(value);

}

}}

// map.ts 核心代码

class MapSubscriber<T, R> extends Subscriber<T> {

count: number = 0;

private thisArg: any;

constructor(

destination: Subscriber<R>,

private project: (value: T, index: number) => R,

thisArg: any

) {

super(destination);

this.thisArg = thisArg || this;

}

protected \_next(value: T) {

let result: R;

try {

result = this.project.call(this.thisArg, value, this.count++);

} catch (err) {

this.destination.error(err);

return;

}

this.destination.next(result);

}}

可以看到 filterSubscriber 和 mapSubscriber 中都只有对类内部属性的操作

### **无参数风格（point-free）**

先通过一个简单地例子来理解无参数风格：

JS 的 map 可以这样使用：

// 过滤出偶数

[1, 2, 3, 4].filter( e => e % 2 === 0 ); // 2, 4

也可以这样用：

// 过滤出偶数

function even(a: number): number {

return a % 2 === 0;

}

[1, 2, 3, 4].filter(even); // 2, 4

第二种做法就是一种无参数风格（point-free），这种风格的好处就是可以能更大的限度的简洁代码，提高代码的可读性。有些读者可能不太同意这种观点，但是实际在代码中践行起来，无参数风格确实会让你的代码看起来更优雅。

在 RxJS 中，几乎每个操作符的第一个参数都为函数类型，这样你可以很方便地践行无参数风格的代码。

### **不可变性**

来看下面的这段代码示例：

// 为数组中的每个对象的 a 属性值 + 1

function foo(arr: {a: number}[]) {

for(let i = 0; i < arr.length; i++) {

arr[i].a ++;

}

return arr;

}

let arrayA = [{a: 1}, {a: 2}, {a: 3}];

let arrayB = foo(arrayA);

console.log(arrayA); // 2, 3, 4

console.log(arrayB); // 2, 3, 4

可以发现由于 arrayA 是引用类型，导致其在函数计算的过程中自身也被改变了，但是这并不是我们的本意。

再来对比下面这段代码：

// 为数组中的每个对象的 a 属性值 + 1

function foo(arr: {a: number}[]) {

let ret: {a: number}[] = [];

for(let i = 0; i < arr.length; i++) {

ret.push({

a: arr[i].a + 1

})

}

return arr;}

let arrayA = [{a: 1}, {a: 2}, {a: 3}];

let arrayB = foo(arrayA);

console.log(arrayA); // 1, 2, 3

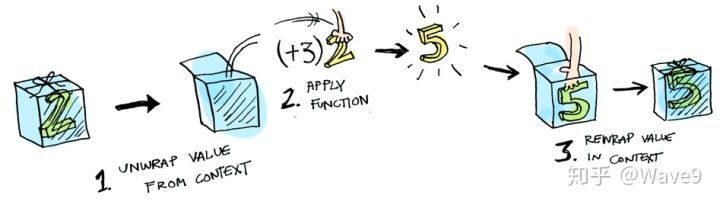
console.log(arrayB); // 2, 3, 4

第二段代码中创建了一个新的数组去接收改变后的值，而不是直接改变传入的参数，这就一种“不可变性”的实践，对比这两段代码，可以明显的感知到第二类代码带来的 bug 率会更低。

RxJS 基于 JavaScript ，只要我们在使用 RxJS 的过程中遵循这些编码规则，就可以达到函数式编程的“不可变性”。

### **函子**

函子是函数式编程中一个比较拗口的概念，笔者这里引用一张图



这个包装起来的就是函子 functor，我们类比 JS 中的概念，例如：

let arrayA = [1, 2, 3, 4];

let arrayB = arrayA.map( e => e + 2 );

arrayA 作为数组拥有 map 的能力， map 后仍然类型不变（具有相同的规则），我们就可以将其称为函子。

同样的在 RxJS 中，Observable 也有 map 的能力，并且 map 后仍然是个 Observable，我们也可以将其称为函子。

### **组合**

思考这样一个场景，假使我们需要对数字做先过滤出偶数再加一的操作

function plusOne(arr) {

let ret = [];

for(let i = 0; i < arr.length;i ++) {

ret.push(arr[i] + 1);

}

return ret;

}

function even(arr) {

let ret = [];

for(let i = 0; i < arr.length;i ++) {

if(arr[i] % 2 === 0) {

ret.push(arr[i]);

}

}

return ret;

}

const source = [1, 2, 3, 4];

const target = plusOne(even(source));

console.log(target); // [3, 5]

如果处理的流程越来越多，那么函数嵌套的层级也会越来深，最后可能会出现 fn5(fn4(fn3(fn2(fn1(arr))))) 这类情况。

我们可以通过一种组合函数 pipe 来让这种情况下的嵌套变得更易读，可维护性也更高。

*/*/ 利用 reduce 来嵌套执行

function pipe(...fns) {

return (prev) => fns.reduce( (prev, cur) => cur(prev), prev );

}

function plusOne(arr) {

let ret = [];

for(let i = 0; i < arr.length;i ++) {

ret.push(arr[i] + 1);

}

return ret;

}

function even(arr) {

let ret = [];

for(let i = 0; i < arr.length;i ++) {

if(arr[i] % 2 === 0) {

ret.push(arr[i]);

}

}

return ret;

}

let source = [1, 2, 3, 4];

let target = pipe(even, plusOne)(source);

console.log(target); // [3, 5]

在 RxJS 中，工具操作符 pipe 同样可以将嵌套的操作符转为可读性很高的操作符组合，当然了 pipe 的作用也不只是如此，后面我们会详细的介绍 pipe。

## **三、深入 RxJS 核心概念**

要点：

* Observable 可观察对象（可被观察者）
* Observer 观察者
* Operators 操作符
* Subscription 订阅对象
* Subject 主体

简短描述如下：

const subscription = Observable.pipe(operator).subscribe(observer);

### **Observable**

Observable 是 RxJS 响应式编程的核心概念，我们逐一分析 Observable 中的主要功能。

**实现 Subscribable 接口**

Subscribable 是一个 interface，用于定义 subscribe 函数签名

**constructor**

Observable 会将构造函数中传入的 subscribe 赋值给内部 \_subscribe。

class Observable {

constructor(subscribe) {

this.\_subscribe = subscribe;

} }

大部分情况下，我们并不会直接去调用 Observable 的构造函数，但是如果需要自定义一些创建型操作符时，就需要知道 new Observable 的使用方式。

以 of 为例，例如：

of([1, 2, 3]).subscribe( res => console.log(res) ); // 1// 2// 3

我们这样来模拟 of 调用 Observable 构造函数：

const subscribeToArray = (array) => (subscriber) => {

for (let i = 0, len = array.length; i < len && !subscriber.closed; i++) {

subscriber.next(array[i]);

}

subscriber.complete();

};

new Observable(subscribeToArray([1,2,3]))

.subscribe( res => {

console.log(res);

});

// 1// 2// 3

**subscribe**

class Observable {

// ... // 简化后的subscribe

subscribe(observerOrNext, error, complete) {

// subscriber

const sink = toSubscriber(observerOrNext, error, complete);

sink.add(this.\_subscribe(sink));

}

// ...}

可以看到传入的 observer 会被转为 subscriber，而 subscriber 中可以做到对流的控制，例如在 subscriber 中执行 next 时就会触发 observer 的 next，需要注意的是在 sink.add 这步，this.\_subscribe(sink) 的结果会在 add 方法里被包装成 Subscription 并被添加到 Subscriptions 数组里，这样做的好处是当有 SubScription 执行 unsubscribe 操作时，就会执行整个 Subscription 的 unsubscribe 。

例如这段代码：

import { Subject, of } from 'rxjs';

import { takeUntil } from 'rxjs/operators';

const destroy$ = new Subject<void>();

of([1, 2, 3]).pipe(takeUntil(destroy$)).subscribe( res => console.log(res) );

断点后可以看到：

### **Observer**

Observer 是和 Observable 紧密关联的概念，例如：

const observer = {

next: x => console.log('Observer got a next value: ' + x),

error: err => console.error('Observer got an error: ' + err),

complete: () => console.log('Observer got a complete notification'),};

传递给 Observable 的 subscribe 方法：

of([1, 2, 3]).subscribe(observer);

当然，Observer 并不是一定都要包含 next, error, complete，就算 subscribe 中只传入一个函数，你的代码也不会报错。

### **Operators**

操作符是 RxJS 的重要组成，我们会在下面一节专门探索操作符。

### **Subscription**

Observable 被订阅后的对象就是 Subscription。

const subscription = of([1, 2, 3]).subscribe( res => console.log(res) );

Subscription 包含三个方法 add, unsubscribe, remove：

**add & remove**

Subscripiton 对象包含一个内部私有的 subscriptions 数组，add 和 remove 则是对 subscriptions 的增删操作。

除了 subscriptions 数组外，Subscription 内部还会维护一个 \_parentOrParents ，这是为了在父 Subscription 被取消订阅时，子 Subscription 也会被取消订阅，从而避免内存泄漏。

**unsubscribe**

unsubscribe 是 Subscription 的核心功能，当执行 unsubscribe 时，意味着这段流的终止，Subscription 也会释放相应的资源。

主要包含三个部分：

1. 父 Subscription 会移除该 Subscription
2. 被 add 进 subscriptions 数组的所有 Subscription 都会执行 unsubscribe
3. 内部 closed 标识符会置为 true ，释放资源

### **Subject**

Subject 继承于 Observable，并且声明 Subscription 的接口

class Subject<T> extends Observable<T> implements SubscriptionLike {

// ... }

所以它既能被订阅，也能被直接取消订阅，并且它还可以主动触发 next, error, complete，我们可以认为它是类型灵活的 Observable。

Subject 包含四类：

* 普通的 Subject
* 可以存储最后值的 BehaviorSubject
* 可以存储多个值的 ReplaySubject
* 只记录最后值的 AsyncSubject

四类 Subject 的不同主要体现在内部的 \_subscribe 实现方式上：

**BehaviorSubject.ts**

\_subscribe(subscriber: Subscriber<T>): Subscription {

const subscription = super.\_subscribe(subscriber);

if (subscription && !(<SubscriptionLike>subscription).closed) {

subscriber.next(this.\_value);

}

return subscription;}

**ReplaySubject.ts**

\_subscribe(subscriber: Subscriber<T>): Subscription {

const \_infiniteTimeWindow = this.\_infiniteTimeWindow;

const \_events = \_infiniteTimeWindow ?

this.\_events : this.\_trimBufferThenGetEvents();

const len = \_events.length;

let subscription: Subscription;

// ...

if (this.closed) {

throw new ObjectUnsubscribedError();

} else if (this.isStopped || this.hasError) {

subscription = Subscription.EMPTY;

} else {

this.observers.push(subscriber);

subscription = new SubjectSubscription(this, subscriber);

}

// 多个值重新 next if (\_infiniteTimeWindow) {

for (let i = 0; i < len && !subscriber.closed; i++) {

subscriber.next(<T>\_events[i]);

}

} else {

for (let i = 0; i < len && !subscriber.closed; i++) {

subscriber.next((<ReplayEvent<T>>\_events[i]).value);

}

}

// ... return subscription;

}

**AsyncSubject.ts**

\_subscribe(subscriber: Subscriber<any>): Subscription {

if (this.hasError) {

subscriber.error(this.thrownError);

return Subscription.EMPTY;

// 结束时才会 next

} else if (this.hasCompleted && this.hasNext) {

subscriber.next(this.value);

subscriber.complete();

return Subscription.EMPTY;

}

return super.\_subscribe(subscriber);}

## **四、深入 RxJS 操作符实现**

要点：

* 理解 lift
* 设计操作符的基本原则

### **lift**

如果写过 RxJS 5.x 版本的代码的开发者一定知道当时的操作符是直接挂在 Observable 上的。

**RxJS 5.x：**

Rx.Observable

.fromEvent(button, 'click')

.throttleTime(1000)

.subscribe(() => console.log(`Clicked`));

**对比 RxJS 6.x：**

fromEvent(document, 'click')

.pipe(

throttleTime(1000)

)

.subscribe(() => console.log(`Clicked`));

RxJS 6.x 的这项改动带来了三个好处：

* 因为不用直接挂载操作符到 Observable.prototype 上，所以 Observable 干净了很多
* 让 RxJS tree-shakeable
* 编写第三方 operators 更加简单，因为不再需要手动 patch 到 Observable.prototype 上

既然不需要手动挂到 Observable.prototype 上，那一定有其它工具函数帮助解决了这一部分工作，这个工具函数就是 lift。

lift 本意是将某物提升到更高的位置或水平上，以 map 操作符为例：

// 简略后的代码function map(project, thisArg) {

return function mapOperation(source) {

return lift(source, new MapOperator(project, thisArg));

};}

可以看到 map 操作符本身是独立于 Observable 的，lift 会将源 Observable 与 map 操作符联系在一起，我们再来看看 lift 的实现：

*/*/ lift，stankyLift，hasLift 简化后的核心代码

function lift(operator) {

const observable = new Observable();

observable.source = this;

observable.operator = operator;

return observable;

}

这就是 lift 的核心功能。

### **设计操作符的基本规则**

RxJS 的操作符分为九类：

* 创建操作符
* 转换操作符
* 过滤操作符
* 组合操作符
* 多播操作符
* 错误处理操作符
* 工具操作符
* 条件和布尔操作符
* 数学和聚合操作符

由于操作符的数量和分类太多，这里不会讲述每一个操作符的实现原理，但是每个操作符都有着基本的设计规则，我们来一探究竟。

以 filter 操作符为例：

**1.调用时会返回 Observable**

function filter(predicate, thisArg) {

return function filterOperatorFunction(source) {

return lift(source, new FilterOperator(predicate, thisArg));

};}

filter 会被挂载到 Observable 上并返回该 Observable

**2.订阅、取消订阅与资源释放**

每个操作符会有自己的 Subscriber

class FilterOperator implements Operator {

// ...

call(subscriber, source) {

return source.subscribe(

new FilterSubscriber(subscriber, this.predicate, this.thisArg));

}}

取消订阅和资源释放这部分我们在 Subscription 中详细讲述过，这里就不再赘述

**3.异常处理**

当操作符执行出错时，需要捕获异常，并向后继续传递下去。

class FilterSubscriber<T> extends Subscriber<T> {

// ... protected \_next(value: T) {

let result: any;

try { // ... } catch (err) { // 捕获异常

this.destination.error(err);

return;

}

// ... }}

## **五、深入 RxJS 异常处理**

要点：

* catchError
* retry
* retryWhen
* 对比同步的 try/catch 与 Promise 的异常处理

RxJS 操作符计算出错时，不会让代码处于停止运行的状态，而是会将错误随着流一起传递下去，RxJS 提供了三种操作符用于捕获异常。

### **catchError**

of(1, 2, 3, 4, 5).pipe(

map(n => {

if (n === 4) {

throw 'four!';

}

return n;

}),

catchError(err => of('I', 'II', 'III', 'IV', 'V')),).subscribe(x => console.log(x));

// 1, 2, 3, I, II, III, IV, V

前面我们提高每个操作符会有个自己的 Subscriber，catchError 也不例外，不过在 CatchSubscriber 中，只有 error 方法用于捕获异常。

class CatchSubscriber extends OuterSubscriber {

// ... error(err) {

if (!this.isStopped) {

let result: any;

try {

result = this.selector(err, this.caught);

} catch (err2) {

super.error(err2);

return;

}

// ... }

}}

### **retry**

如果想要失败后重试，可以使用 retry 操作符。

const source = interval(1000);

const example = source.pipe(

mergeMap(val => {

if(val > 5){

return throwError('Error!');

}

return of(val);

}),

//retry 2 times on error retry(2));

const subscribe = example.subscribe({

next: val => console.log(val),

error: val => console.log(`${val}: Retried 2 times then quit!`)});

// Output:// 0..1..2..3..4..5..// 0..1..2..3..4..5..// 0..1..2..3..4..5..// "Error!: Retried 2 times then quit!"

retrySubscriber

class RetrySubscriber<T> extends Subscriber<T> {

// ...

next(value?: T): void { // ... }

error(err: any) {

if (!this.isStopped) {

const { source, count } = this;

// 再次失败时，计数会 - 1，直到计数降为 0

if (count === 0) {

return super.error(err);

} else if (count > -1) {

this.count = count - 1;

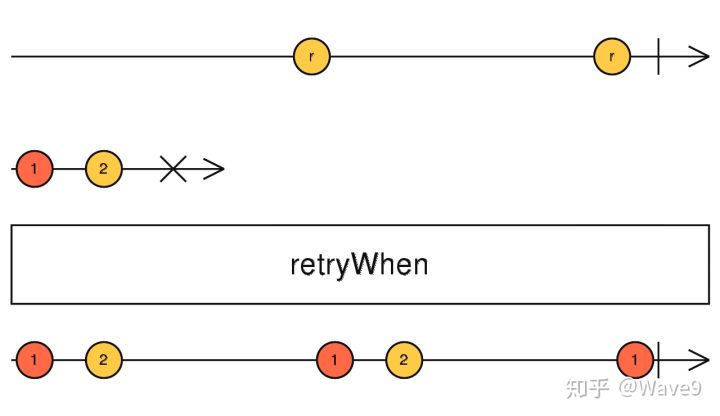
}

// ... }

}}

### **retryWhen**

与 retry 接受重试次数作为不一样，retryWhen 接受 Observable 作为参数，执行流程看弹珠图会很好理解。



当 retryWhen 的 Observable 发送新的值时，源 Observable 会开始重试操作。

例如：

const source = interval(1000);

const example = source.pipe(

map(val => {

if (val > 5) {

// 大于 5 后会报错并重试

throw val;

}

return val;

}),

retryWhen(errors =>

errors.pipe(

tap(val => console.log(`Value ${val} was too high!`)),

// 5s 后重试

delayWhen(val => timer(val \* 1000))

)

));

const subscribe = example.subscribe(val => console.log(val));

打断点到 retrySubscriber 内部，会发现 source 被重新订阅

## **六、探索 RxJS 测试、文档和调试技巧**

要点： 弹珠测试 代码即文档 如何调试 RxJS

### **弹珠测试**

如果你翻开 RxJS 的测试用例，你会发现它的测试用例也是弹珠图的形式

这并不是什么黑魔法，而是因为 RxJS 使用的是独有的 marble-testing 来表述测试语言，不同的符号也代表着不同的含义，如果想要学习弹珠测试，可以在这里了解更多的知识。

[rxjs-marble-testing​github.com](https://link.zhihu.com/?target=https://github.com/cartant/rxjs-marbles" \t "https://zhuanlan.zhihu.com/p/_blank)

### **文档**

RxJS 的文档非常详细，详细到每个 API 的功能都会写在文档里，这并不是因为 RxJS 的作者非常勤奋，不舍昼夜的编写这些文档，而是因为这些文档都是生成的，甚至连文档上的弹珠图都是生成的，你可以在这里查看生成弹珠图的脚本：

[ReactiveX/rxjs​github.com](https://link.zhihu.com/?target=https://github.com/ReactiveX/rxjs/blob/master/docs_app/tools/marbles/scripts/index.ts" \t "https://zhuanlan.zhihu.com/p/_blank)

### **调试**

RxJS 核心成员 Nicholas Jamieson 写过两篇文章专门讲述了 RxJS 的调试技巧：

[Debugging RxJS, Part 1: Tooling​medium.com](https://link.zhihu.com/?target=https://medium.com/angular-in-depth/debugging-rxjs-4f0340286dd3" \t "https://zhuanlan.zhihu.com/p/_blank)[Debugging RxJS, Part 2: Logging​medium.com](https://link.zhihu.com/?target=https://medium.com/angular-in-depth/debugging-rxjs-part-2-logging-56904459f144" \t "https://zhuanlan.zhihu.com/p/_blank)

但是笔者比较喜欢在 stackblitz 上做 debug，因为它可以直接测试到 RxJS 的 TypeScript 代码，特别是在阅读 RxJS 源码的时候，调试步骤很简单：

1.打开 stackblitz 新建一个 RxJS 项目并在合适的位置上输入 debugger

2.打开控制台，刷新 demo 页，你就会发现页面进入 debug 模式

3.点击右侧的调用栈，你会发现可以直接进入到 TypeScript 源文件中

4.接下来你就可以随心所欲的调试 RxJS 了！

## **总结：**

笔者通过如下六个方面对 RxJS 做了比较细致的分析

* RxJS 是如何构建出一个**响应式**编程库的？
* RxJS 是如何构建出一个**函数式**编程库的？
* 深入 RxJS 的核心概念
* 深入 RxJS 的操作符实现
* 深入 RxJS 的异常处理
* 探索 RxJS 的测试、文档和调试技巧