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

protected source: Observable<any> | undefined;

protected operator: Operator<any, T> | undefined;

//Observable 将传入的函数存为内部

//当可观察对象最初被订阅时调用的函数。这个函数被赋予了一个订阅者，新的值可以被//“next”，或者调用“error”方法来引发错误，或者调用“complete”来通知成功完成。

constructor(

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

) {

if (subscribe) { this.\_subscribe = subscribe; }

}

static create:

(...args: any[]) => any = <T>(subscribe?: (subscriber: Subscriber<T>) => TeardownLogic)

=> { return new Observable<T>(subscribe); };

protected lift<R>(operator?: Operator<T, R>): Observable<R> {

const observable = new Observable<R>();

observable.source = this;

observable.operator = operator;

return observable;

}

subscribe(observer?: PartialObserver<T>): Subscription;

subscribe(next: null | undefined, error: null | undefined, complete: () => void): Subscription;

Subscribe(

observerOrNext?: PartialObserver<T> | ((value: T) => void) | null,

error?: ((error: any) => void) | null,

complete?: (() => void) | null

): Subscription {

const subscriber = isSubscriber(observerOrNext) ?

observerOrNext : new SafeSubscriber(observerOrNext, error, complete);

const { operator, source } = this;

subscriber.add(

operator

? operator.call(subscriber, source)

: source || config.useDeprecatedSynchronousErrorHandling

? this.\_subscribe(subscriber)

: this.\_trySubscribe(subscriber)

);

return subscriber;

}

protected \_trySubscribe(sink: Subscriber<T>): TeardownLogic {

try {

return this.\_subscribe(sink);

} catch (err) {

if (config.useDeprecatedSynchronousErrorHandling) {

throw err;

}

sink.error(err);

}

}

forEach(next: (value: T) => void): Promise<void>;

forEach(next: (value: T) => void, promiseCtor: PromiseConstructorLike): Promise<void>;

ForEach(next: (value: T) => void, promiseCtor?: PromiseConstructorLike): Promise<void> {

promiseCtor = getPromiseCtor(promiseCtor);

return new promiseCtor<void>((resolve, reject) => {

let subscription: Subscription;

subscription = this.subscribe(

(value) => {

try {

next(value);

} catch (err) {

reject(err);

subscription?.unsubscribe();

}

},

reject,

resolve

);

}) as Promise<void>;

}

protected \_subscribe(subscriber: Subscriber<any>): TeardownLogic {

return this.source?.subscribe(subscriber);

}

[Symbol\_observable]() { return this; }

pipe(): Observable<T>;

toPromise(): Promise<T | undefined>;

ToPromise(promiseCtor?: PromiseConstructorLike): Promise<T | undefined> {

promiseCtor = getPromiseCtor(promiseCtor);

return new promiseCtor((resolve, reject) => {

let value: T | undefined;

this.subscribe(

(x: T) => (value = x),

(err: any) => reject(err),

() => resolve(value)

);

}) as Promise<T | undefined>;

}

}

function getPromiseCtor(promiseCtor: PromiseConstructorLike | undefined) {

return promiseCtor ?? config.Promise ?? Promise;

}

function isObserver<T>(value: any): value is Observer<T> {

return value && isFunction(value.next) && isFunction(value.error) && isFunction(value.complete);

}

function isSubscriber<T>(value: any): value is Subscriber<T> {

return (value && value instanceof Subscriber) || (isObserver(value) && isSubscription(value));

}

### ***深入了解***

### 接下来我们来深入了解一下我们上面学习到的一些内容，首先是[Observable](https://rxjs-dev.firebaseapp.com/api/index/class/Observable" \t "https://juejin.im/entry/_blank)， 这是一个类，用来创建Observable；我们一般会使用它的[create](https://rxjs-dev.firebaseapp.com/api/index/class/Observable" \l "create" \t "https://juejin.im/entry/_blank) 方法来创建一个Observable对象；create方法其实调用的是Observable的构造器[constructor()](https://rxjs-dev.firebaseapp.com/api/index/class/Observable" \l "constructor()" \t "https://juejin.im/entry/_blank)

那我们来看一下这个构造器需要传递的参数是什么，看下面的[代码](https://github.com/ReactiveX/rxjs/blob/master/src/internal/Observable.ts" \l "L37" \t "https://juejin.im/entry/_blank)

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

我们可以知道，传递的参数是一个subscribe，它是一个函数；这个函数会在Observable对象调用subscribe方法的时候执行；这个函数有一个[Subscriber](https://rxjs-dev.firebaseapp.com/api/index/class/Subscriber" \t "https://juejin.im/entry/_blank)类型的值， 然后我们可以通过这个subscriber，发送三种类型的通知，并且可以传递值给外面的 [observer](https://rxjs-dev.firebaseapp.com/api/index/class/Observable" \l "subscribe-" \t "https://juejin.im/entry/_blank)， 或者相应接收值的函数。

Observable对象的[subscribe](https://rxjs-dev.firebaseapp.com/api/index/class/Observable" \l "subscribe-" \t "https://juejin.im/entry/_blank)方法的参数可以是一个[Observer](https://rxjs-dev.firebaseapp.com/api/index/interface/Observer" \t "https://juejin.im/entry/_blank)类型的对象， 或者一到三个函数；如果是一个对象的话，需要满足[Observer](https://rxjs-dev.firebaseapp.com/api/index/interface/Observer" \t "https://juejin.im/entry/_blank)接口的一些属性； 一般情况下这个对象上面至少有一个next方法，来接收相应的Observable传递过来的值。如果给subscribe方法传递的参数是函数的话，那么可以传递一到三个， 第一个接收Next类型通知，第二个接收Error类型的通知，第三个接收Complete类型的通知；如果相应的通知可以传递值的话，那么我们函数的参数就是相应要传递的值。

Observable对象调用subscribe方法之后，返回的是一个[Subscription](https://rxjs-dev.firebaseapp.com/api/index/class/Subscription" \t "https://juejin.im/entry/_blank)类型的对象，它有一个[unsubscribe](https://rxjs-dev.firebaseapp.com/api/index/class/Subscription" \l "unsubscribe" \t "https://juejin.im/entry/_blank)方法，可以取消Observable execution的执行过程。

<https://zhuanlan.zhihu.com/p/157897829>

***Observable***

可观察对象是整个 Rx 的核心，主要的作用就是提供了一个观察者模式，使得调用者可以通过响应式的方式获取数据。

Observable 实际上就是一个单向链表，基本的数据结构如下：

class Observable<T> {

source: Observable<any>;

}

其构造方法与 Promise 类似，通过传入一个函数包裹操作，并让这个函数来决定数据传递，这个函数的参数包含了一个订阅器。

const observable = new Observable((subscriber) => {

subscriber.next(1);

subscriber.error(Error('error message'));

subscriber.complete();

});

订阅器提供了三个主要方法：next，error，complete。订阅器的实现很巧妙，其内部实现是一个链表。

跟 Promise 不同，Observable 不会立刻运行这个函数，而是等到它被订阅后，这个函数才会被执行，这种惰性求值的特性使得 Observable 可以在它仅被需要的地方进行计算。

***lift***

lift 方法提供了一个这样的功能，传入一个映射函数，并返回一个新的 Observable，这个新的 Observable 的 source 会指向创建它的 Observable。实际上，这种做法就是将这个映射函数用一个外覆类包裹起来，这个外覆类，正是 Observable。那么，看看它是如何实现。

lift<R>(operator: Operator<T, R>): Observable<R> {

const observable = new Observable<R>();

observable.source = this;

observable.operator = operator;

return observable;

}

***pipe***

Rxjs 跟其他语言实现的 ReactiveX 不一样的地方就是在于，它的映射方法不再是放在 Observable 内部，而是通过参数的形式传入到一个管道函数pipe中，在这个函数中，通过对管道函数的数组进行 reduce 后，就能够得到最终的Observable。这个 reduce 的过程也很巧妙，传入的函数的参数就是上游的 Observable，返回的就是一个给下游接收的 Observable，那么就可以把一个又一个的 Observable 串联起来

pipe(...operations: OperatorFunction<any, any>[]): Observable<any> {

if (operations.length === 0) {

return this as any;

}

if (operations.length == 1) {

return operation[0];

}

return operations.reduce((prev, fn) => fn(prev), this);

}

那么在使用过程中，pipe 通过重载给传入的函数提供类型信息。

export function pipe<T>(): UnaryFunction<T, T>;

export function pipe<T, A>(fn1: UnaryFunction<T, A>): UnaryFunction<T, A>;

export function

pipe<T, A, B>(fn1: UnaryFunction<T, A>, fn2: UnaryFunction<A, B>): UnaryFunction<T, B>;

// ...

其中 UnaryFunction 表示一元函数，通过这种链式操作，使得链条上的所有函数都可以拿到上游的类型，并把类型转化传递给下游。

***subscribe***

当 Observable 一旦调用 subscribe，那么就意味着其开始执行链条中的所有函数。subscribe 传入的参数是一个包含了 next ，error ， complete 三个属性的对象；也可以是三个函数，分别对应 next，error，complete。

observable.subscribe(

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

(error) { console.error(error);},

() { console.log('complete');});

observable.subscribe({

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

error: (error) { console.error(error); },

complete: () { console.log('complete'); },

});

其具体实现是通过将传入的函数（对象）参数转化成 Subscriber 对象，而 Subscriber 继承了 Subscription。最后，返回的就是一个 subscription 给到调用者。

subscribe(

observerOrNext?: PartialObserver<T> | ((value: T) => void),

error?: (error: any) => void,

complete?: () => void): Subscription

) {

// operator 是一个映射函数

const {operator} = this;

const sink = new Subscriber(observerOrNext, error, complete);

if (operator) {

sink.add(operator.call(sink, this.source));

} else {

sink.add(this.source || !sink.syncErrorThrowable ?

this.\_subscribe(sink) :

this.\_trySubscribe(sink)

);

}

// 省略了错误处理

return sink;

}

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

const { source } = this;

return source && source.subscribe(subscriber);

}

\_trySubscribe(sink: Subscriber<T>): TeardownLogic {

try {

return this.\_subscribe(sink);

} catch (err) {

// 此处省略了源码中的一些判断，不影响阅读

sink.error(err);

}

Subscriber 的 add 方法下面会讲。总之，Observable 就像一串或者一个爆竹，只有当它被点燃（subscribe）的时候，才会把一个又一个的 Observable 点着，最终迸发出巨大声响，而 subscribe 就是一个找到引线并点燃它们的过程。

***Subscription***

Subscription 则是通过一种树结构，它包含了叶节点和一个父节点或者父节点的集合。

class Subscription {

\_parentOrParents: Subscription;

\_subscriptions: Subscription[];

}

***add***

add 方法主要的功能是连接不同的订阅，配合注释，其逻辑就是将函数或者订阅对象包裹后放入成员变量 subscriptions 中，并将这个包裹对象的父订阅对象设置为当前对象。

add(logic: Function | Subscription | void): Subscription {

let subscription = logic;

if (typeof logic === 'object') {

// 如果添加进来订阅已经被取消了，则不进行设置。

// 如果当前的订阅已经被取消，添加进来的订阅也应该要被取消。

if (subscription === this || subscription.closed ||

typeof subscription.unsubscribe !== 'function') {

return subscription;

} else if (this.closed) {

subscription.unsubscribe();

return subscription;

} else if (!(subscription instanceof Subscription)) {

const tmp = subscription;

subscription = new Subscription();

subscription.\_subscriptions = [tmp];

}

} else if (typeof logic === 'function' ) {

subscription = new Subscription(<(() => void)>teardown);

} else {

// 抛出错误。 }

// 设置父对象的过程采用懒加载模式。

let { \_parentOrParents } = subscription;

if (\_parentOrParents === null) {

// 如果没有设置父对象，则设置当前对象为父对象。

subscription.\_parentOrParents = this;

} else if (\_parentOrParents instanceof Subscription) {

// 如果父对象已经是当前的对象，直接返回。

if (\_parentOrParents === this) {

return subscription;

}

// 添加进来的订阅的父对象已经存在，那么用一个数组保存。

subscription.\_parentOrParents = [\_parentOrParents, this];

} else if (\_parentOrParents.indexOf(this) === -1) {

// 如果已经是数组对象了，并且不存在当前订阅对象，则设置当前订阅对象

\_parentOrParents.push(this);

} else {

// 已经设置当前订阅对象为父对象

return subscription;

}

// 同样，设置叶子结点的过程也是用懒加载

const subscriptions = this.\_subscriptions;

if (subscriptions === null) {

this.\_subscriptions = [subscription];

} else {

subscriptions.push(subscription);

}

return subscriptio

***unsubscribe***

取消订阅是订阅对象的主要功能，它为观察者模式提供了终结观察的方法。

unsubscribe(): void {

// 已经取消订阅了。

if (this.closed) {

return;

}

// 拿到当前想要取消订阅的相关的对象。

// 这样做的目的是防止loop

let { \_parentOrParents, \_unsubscribe, \_subscriptions } = (<any> this);

// 设置取消订阅

this.closed = true;

// 设置父对象为空

this.\_parentOrParents = null;

// 设置订阅为空

this.\_subscriptions = null;

// 父对象可能是数组，也可能是订阅对象

if (\_parentOrParents instanceof Subscription) {

\_parentOrParents.remove(this);

} else if (\_parentOrParents !== null) {

for (let index = 0; index < \_parentOrParents.length; ++index) {

const parent = \_parentOrParents[index];

parent.remove(this);

}

}

// \_unsubscribe 是一个外部传入的函数.

if (isFunction(\_unsubscribe)) {

try {

\_unsubscribe.call(this);

} catch (e) {

errors = e instanceof UnsubscriptionError ? flattenUnsubscriptionErrors(e.errors) : [e];

}

}

// 将所有的子订阅取消订阅

if (isArray(\_subscriptions)) {

let len = \_subscriptions.length;

for (const sub of \_subscriptions) {

if (isObject(sub)) {

try {

sub.unsubscribe();

} catch (e) {

// 省略错误处理

}

}

}

}

//

结语

第一篇就先介绍这两个重要的组成类，由这两个类引申出来的组合才是 Rx 的精华，之后会陆续介绍这些操作函数。

*Observable*（多播观察源），这些 Observable 在 RxJS 中主要是以 Subject 命名，它们有以下几种不同的实现：

1. Subject AnonymousSubject BehaviorSubject
2. ReplaySubject AsyncSubject

所谓 *Muticasted Observable*，就是这个 Observable 可以持续的发送数据给到订阅它的订阅者们。 注：文中 RxJS 所使用的源码版本为 6.6.0

## **Subject**

Subject 是最基础的 *Muticasted Observable*，订阅者对其进行订阅后，将会拿到 Subject 之后发送的数据。但是，如果订阅者在数据发送后再订阅，那么它将永远都拿不到这条数据。用一下例子简单说明一下：

const subject = new Subject<number>();

// 订阅之前调用是不会打印

subject.next(1);

// 订阅数据

const subscription = subject.subscribe((value) => {

console.log('订阅数据A：' + value);

});

subject.next(2);

// 打印结果 订阅数据A：2

Subject 的实现通过将观察员们放入数组中，如果有事件即将到来，通知当前所有已经在位的观察员们。

class Subject<T> extends Observable<T> {

observers: Observer<T>[] = [];

// 省略了一些内容

next(value?: T) {

if (!this.isStopped) {

...

const { observers } = this;

const len = observers.length;

const copy = observers.slice();

for (let i = 0; i < len; i++) {

copy[i].next(value);

}

}

}

// error 类似于 next

error(err: any) {

...

this.hasError = true;

this.thrownError = err;

this.isStopped = true;

const { observers } = this;

const len = observers.length;

const copy = observers.slice();

for (let i = 0; i < len; i++) {

copy[i].error(err);

}

this.observers.length = 0;

}

// complete 类似于 next

complete() {

...

this.isStopped = true;

const { observers } = this;

const len = observers.length;

const copy = observers.slice();

for (let i = 0; i < len; i++) {

copy[i].complete();

}

this.observers.length = 0;

}

}

通过重写了 \_subscribe ，将观察员在订阅时保存到 observers 数组中。

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

if (this.hasError) {

subscriber.error(this.thrownError);

return Subscription.EMPTY;

} else if (this.isStopped) {

subscriber.complete();

return Subscription.EMPTY;

} else {

// 如果都没有问题，在这里将观察员保存到 observers 数组。

this.observers.push(subscriber);

// 提供一个指向于当前观察者的订阅对象。

return new SubjectSubscription(this, subscriber)

}

}

Subject 通过创建一个新的指向于它的 observable，完成和 Observable 之间的转换。

asObservable(): Observable<T> {

const observable = new Observable<T>();

(<any>observable).source = this;

return observable;

}

## **AnonymousSubject**

AnonymousSubject 是一个 Subject 的 wrapper，它拥有一个 名为 destination 的 Observer 成员。 Observer 提供了三个方法接口，分别是 next，error 和 complete。

export interface Observer<T> {

closed?: boolean;

next: (value: T) => void;

error: (err: any) => void;

complete: () => void;

}

AnonymousSubject 通过重载 Subject 的 next，error，complete 将调用转发到 destination 。由于其重载这三个重要的方法，其本身并不具备 Subject 所提供的功能。AnonymousSubject 重载这些方法的主要作用是为了将调用转发到 destination ，也就是提供了一个

export class AnonymousSubject<T> extends Subject<T> {

constructor(protected destination?: Observer<T>, source?: Observable<T>) {

super();

this.source = source;

}

next(value: T) {

const { destination } = this;

if (destination && destination.next) {

destination.next(value);

}

}

error(err: any) {

const { destination } = this;

if (destination && destination.error) {

this.destination.error(err);

}

}

complete() {

const { destination } = this;

if (destination && destination.complete) {

this.destination.complete();

}

}

}

它也重载 \_subscribe，那么也就不具备 Subject 的保存订阅者的功能了。

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

const { source } = this;

if (source) {

return this.source.subscribe(subscriber);

} else {

return Subscription.EMPTY;

}

}

通过阅读源码使用到 AnonymousSubject 的地方，我认为 AnonymousSubject 主要的功能还是为 Subject 的 lift 方法提供一个封装，lift 需要返回的是一个符合当前类的同构对象。

export class Subject<T> extends Observable<T> {

lift<R>(operator: Operator<T, R>): Observable<R> {

const subject = new AnonymousSubject(this, this);

subject.operator = <any>operator;

return <any>subject;

}

}

如果直接重新构造一个 Subject 虽然符合同构，但是存储了过多的冗余数据，比如，订阅的时候就会重复把订阅者添加到 observers 中；如果直接使用 Observable ，那么又不符合同构，因为 Observable 并不具备 next，error 和 complete 等功能，那么这就是一种比较稳妥的做法，通过重载复写 Subject 的一些方法，使得其既具备同构，也不会重复保存冗余数据。

## **BehaviorSubject**

BehaviorSubject 为 Subject 提供了数据持久化（相对于 Subject 本身）功能，它本身存储了已经到来的数据，可以看看以下例子。

const subject = new BehaviorSubject<number>(0);

// 初始化后直接订阅

const subscriptionA = subject.subscribe((value) => {

console.log('订阅数据A：' + value);

});

// 订阅之前调用是不会打印

subject.next(1);

const subscriptionB = subject.subscribe((value) => {

console.log('订阅数据B：' + value);

});

// 订阅后调用会打印数据。

subject.next(2);

// 打印结果

// 订阅数据A：0

// 订阅数据A：1

// 订阅数据B：1

// 订阅数据A：2

//

BehaviorSubject 拥有一个 \_value 成员，每次调用 next 发送数据的时候，BehaviorSubject 都会将数据保存到 \_value 中。

export class BehaviorSubject<T> extends Subject<T> {

constructor(private \_value: T) {

super();

}

get value(): T {

return this.getValue();

}

getValue(): T {

if (this.hasError) {

throw this.thrownError;

} else if (this.closed) {

throw new ObjectUnsubscribedError();

} else {

return this.\_value;

}

}

}

调用 next 的时候，会把传入的 value 保存起来，并交由 Subject 的 next 来处理。

next(value: T): void {

super.next(this.\_value = value);

}

当 BehaviorSubject 被订阅的时候，也会把当前存储的数据发送给订阅者，通过重写 \_subscribe 实现这个功能。

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

const subscription = super.\_subscribe(subscriber);

// 只要订阅器没有关闭，那么就将当前存储的数据发送给订阅者。

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

subscriber.next(this.\_value);

}

return subscription;

}

## **AsyncSubject**

AsyncSubject 并没有提供相应的异步操作，而是把控制最终数据到来的权力交给调用者，订阅者只会接收到 AsyncSubject 最终的数据。正如官方例子所展示的的，当它单独调用 next 的时候，订阅者并不会接收到数据，而只有当它调用 complete 的时候，订阅者才会接收到最终到来的消息。以下例子可以说明 AsyncSubject 的运作方式。

const subject = new AsyncSubject<number>();

const subscriptionA = subject.subscribe((value) => {

console.log('订阅数据A：' + value);

});

// 此处不会触发订阅

subject.next(1);

subject.next(2);

subject.next(3);

subject.next(4);

const subscriptionB = subject.subscribe((value) => {

console.log('订阅数据B：' + value);

});

// 同样，这里不会触发订阅

subject.next(5);

// 但是完成方法会触发订阅

subject.complete();

// 打印结果

// 订阅数据A：5

// 订阅数据B：5

AsyncSubject 通过保留发送状态和完成状态，来达到以上目的。

export class AsyncSubject<T> extends Subject<T> {

private value: T = null;

private hasNext: boolean = false;

private hasCompleted: boolean = false;

}

AsyncSubject 的 next 不会调用 Subject 的 next，而是保存未完成状态下最新到来的数据。

next(value: T): void {

if (!this.hasCompleted) {

this.value = value;

this.hasNext = true;

}

}

那么 Subject 的 next 会在 AsyncSubject 的 complete 方法中调用。

complete(): void {

this.hasCompleted = true;

if (this.hasNext) {

super.next(this.value);

}

super.complete();

}

## **ReplaySubject**

ReplaySubject 的作用是在给定的时间内，发送所有的已经收到的缓冲区数据，当时间过期后，将销毁之前已经收到的数据，重新收集即将到来的数据。所以在构造的时候，需要给定两个值，一个是缓冲区的大小（bufferSize），一个是给定缓冲区存活的窗口时间（windowTime），需要注意的是 ReplaySubject 所使用的缓冲区的策略是 FIFO。

下面举出两个例子，可以先感受一下 ReplaySubject 的行为。第一个如下：

const subject = new ReplaySubject<string>(3);

const subscriptionA = subject.subscribe((value) => {

console.log('订阅数据A：' + value);

});

subject.next(1);

subject.next(2);

subject.next(3);

subject.next(4);

const subscriptionB = subject.subscribe((value) => {

console.log('订阅数据B：' + value);

});

// 打印结果：

// 订阅数据A: 1

// 订阅数据A: 2

// 订阅数据A: 3

// 订阅数据A: 4

// 订阅数据B：2

// 订阅数据B：3

// 订阅数据B：4

下面是第二个例子，这个 ReplaySubject 带有一个窗口时间。

const subject = new ReplaySubject<string>(10, 1000);

const subscriptionA = subject.subscribe((value) => {

console.log('订阅数据A：' + value);

});

subject.next('number');

subject.next('string');

subject.next('object');

subject.next('boolean');

setTimeout(() => {

subject.next('undefined');

const subscriptionB = subject.subscribe((value) => {

console.log('订阅数据B：' + value);

});

}, 2000);

// 打印结果

// 订阅数据A：number

// 订阅数据A：string

// 订阅数据A：object

// 订阅数据A：boolean

// 订阅数据A：undefined

// 订阅数据B：undefined

其实 ReplaySubject 跟 BehaviorSubject 很类似，但是不同的点在于，ReplaySubject 多了缓冲区和窗口时间，也算是扩展了 BehaviorSubject 的使用场景。

在源码中，还有第三个参数，那就是调度器（scheduler），一般来说，使用默认调度器已经可以覆盖大部分需求，关于调度器的部分会在之后讲到。

export class ReplaySubject<T> extends Subject<T> {

private \_events: (ReplayEvent<T> | T)[] = [];

private \_bufferSize: number;

private \_windowTime: number;

private \_infiniteTimeWindow: boolean = false;

constructor(bufferSize: number = Number.POSITIVE\_INFINITY,

windowTime: number = Number.POSITIVE\_INFINITY,

private scheduler?: SchedulerLike) {

super();

this.\_bufferSize = bufferSize < 1 ? 1 : bufferSize;

this.\_windowTime = windowTime < 1 ? 1 : windowTime;

if (windowTime === Number.POSITIVE\_INFINITY) {

this.\_infiniteTimeWindow = true;

this.next = this.nextInfiniteTimeWindow;

} else {

this.next = this.nextTimeWindow;

}

}

}

上面的源码中，ReplaySubject 在构造时会根据不同的窗口时间来设置 next 具体的运行内容，主要以下两种方式。

* nextInfiniteTimeWindow
* nextTimeWindow

### **nextInfiniteTimeWindow**

如果窗口时间是无限的，那么就意味着缓冲区数据的约束条件只会是将来的数据。

private nextInfiniteTimeWindow(value: T): void {

const \_events = this.\_events;

\_events.push(value);

// 根据数据长度和缓冲区大小，决定哪些数据留在缓冲区。

if (\_events.length > this.\_bufferSize) {

\_events.shift();

}

super.next(value);

}

### **nextTimeWindow**

如果窗口时间是有限的，那么缓冲区的约束条件就由两条组成：窗口时间和将来的数据。这时，缓冲区数据就由 ReplayEvent 组成。ReplayEvent 保存了到来的数据的内容和其当前的时间戳。

class ReplayEvent<T> {

constructor(

readonly public time: number,

readonly public value: T

) {}

}

那么通过 \_trimBufferThenGetEvents 对缓冲区数据进行生死判断后，再把完整的数据交由 Subject 的 next 发送出去。

private nextTimeWindow(value: T): void {

this.\_events.push(new ReplayEvent(this.\_getNow(), value));

this.\_trimBufferThenGetEvents();

super.next(value);

}

\_trimBufferThenGetEvents 这个方法是根据不同的 event 对象中的时间戳与当前的时间戳进行判断，同时根据缓冲区的大小，从而得到这个对象中的数据是否能够保留的凭证。

private \_trimBufferThenGetEvents(): ReplayEvent<T>[] {

const now = this.\_getNow();

const \_bufferSize = this.\_bufferSize;

const \_windowTime = this.\_windowTime;

const \_events = <ReplayEvent<T>[]>this.\_events;

const eventsCount = \_events.length;

let spliceCount = 0;

// 由于缓冲区的是 FIFO，所以时间的排

// 序一定是从小到大那么，只需要找到分

// 割点，就能决定缓冲数据的最小数据长

// 度。

while (spliceCount < eventsCount) {

if ((now - \_events[spliceCount].time) < \_windowTime) {

break;

}

spliceCount++;

}

// 缓冲区长度对切割的优先级会更高，

// 所以如果超出了缓冲区长度，那么切

// 割点要由更大的一方决定。

if (eventsCount > \_bufferSize) {

spliceCount = Math.max(spliceCount, eventsCount - \_bufferSize);

}

if (spliceCount > 0) {

\_events.splice(0, spliceCount);

}

return \_events;

}

### **订阅过程**

ReplaySubject 的订阅过程比较特殊，因为订阅的时候需要发送缓冲区数据，而且在不同时间进行订阅也会使得缓冲区中的数据变化，所以订阅是需要考虑的问题会比较多。那么，抓住 \_infiniteTimeWindow 这个变量来看代码会变得很容易。

// 以下源码省略了调度器相关的代码

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

const \_infiniteTimeWindow = this.\_infiniteTimeWindow;

// 窗口时间是无限的则不用考虑

// 窗口时间是有限的则更新缓冲区

const \_events = \_infiniteTimeWindow ? this.\_events : this.\_trimBufferThenGetEvents();

const len = \_events.length;

// 创建 subscription

let subscription: Subscription;

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

subscription = Subscription.EMPTY;

} else {

this.observers.push(subscriber);

subscription = new SubjectSubscription(this, subscriber);

}

// 分类讨论不同的约束条件

if (\_infiniteTimeWindow) {

// 窗口时间不是无限的，缓冲区存储直接就是数据

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

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

}

} else {

// 窗口时间不是无限的，缓冲区存储的是 ReplayEvent

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

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

}

}

if (this.hasError) {

subscriber.error(this.thrownError);

} else if (this.isStopped) {

subscriber.complete();

}

return subscription;

}

## **最后**

本章我主要简单分析了 5 种主要的 Subject，这些 Subject 实现了不同类型的 *Muticasted Observable*，对 Observable 进行了扩展。

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

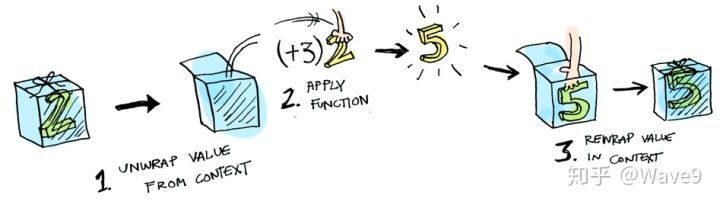
console.log(arrayB); // 2, 3, 4

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;

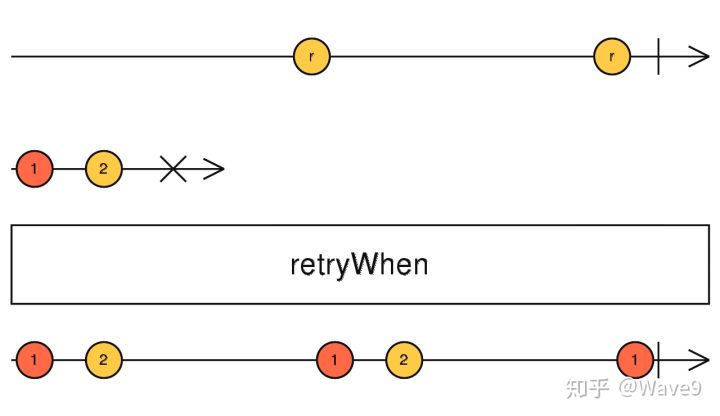
}

// ... }

}}

### **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