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/162685223>

本篇是 RxJS 源码解析的第三篇文章，使用源码的版本是 6.6.0 。本篇文章的内容可能会比较多，请耐心阅读。为了方便阅读，文中的相关代码均经过裁剪和处理。如有不妥，还请指正。

在 RxJS 中，Creation Operator 主要分为以下两类：

执行一般创建操作的 Normal Creation Operator。

执行复杂的创建操作的 Join Creation Operator。

在 pipe 中使用的 operator ，我称之为 Pipe Operator ，它主要分为以下几类：

用于数据映射的 Transformation Operators

过滤用的 Filtering Operators

将当前的 Observable 多播化的 Multicasting Operators

处理错误的 Error Handling Operators

工具操作函数 Utility Operators

Conditional and Boolean Operators

Mathematical and Aggregate Operators

限于篇幅本篇将先介绍 Normal Creation Operator ，它的主要作用是帮助开发者快速创建 Observable。

**一、of , empty & throwError**

of 、empty 、throwError ，首先讲这三个 operator 的重要原因是，它提供了一系列基础的操作：next、complete、以及 error。

const observableA = of(1);

const observableB = empty();

const observableC = throwError(Error('test'));

observableA.subscribe({

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

complete: () => console.log('A: complete');

error: (e) => console.log('A: error is ' + e);

});

observableB.subscribe({

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

complete: () => console.log('B: complete');

error: (e) => console.log('B: error is ' + e);

});

observableC.subscribe({

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

complete: () => console.log('C: complete'),

error: (e) => console.log(`C: error is (${e}).`),

});

// 打印结果

// A: 1

// A: complete

// B: complete

// C: error is Error: test

**1 of source code**

它的构建方式如下，其中，调度器是最后一个参数。

export function of<T>(...args: Array<T | SchedulerLike>): Observable<T> {

let scheduler = args[args.length - 1] as SchedulerLike;

if (isScheduler(scheduler)) {

args.pop();

return scheduleArray(args as T[], scheduler);

} else {

return fromArray(args as T[]);

}

}

of 由两个函数 fromArray 和 scheduleArray。fromArray 是一个简单循环的函数，它将数据循环发送给 Observable 的订阅者。

export function fromArray<T>(input: ArrayLike<T>) {

return new Observable<T>((subscriber: Subscriber<T>) => {

// 循环获取数据

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

subscriber.next(array[i]);

}

subscriber.complete();

});

}

**2 empty source code**

这部分的代码很简单，scheduler 部分可以忽略。实际上就是在 Observable 中调用 subscriber.complete()。

export function empty(scheduler?: SchedulerLike) {

if (scheduler) {

return new Observable<never>(subscriber => scheduler.schedule(

() => subscriber.complete()

));

} else {

return new Observable<never>(subscriber => subscriber.complete());

}

}

**3 throwError source code**

throwError 跟 empty 的实现是一致的，只不过 complete 换成了 error 。

export function throwError(error: any, scheduler?: SchedulerLike): Observable<never> {

if (!scheduler) {

return new Observable(subscriber => subscriber.error(error));

} else {

return new Observable(subscriber => scheduler.schedule(

dispatch,

0,

{ error, subscriber }

));

}

}

// 以下是 调度器中想要执行的状态。

interface DispatchArg {

error: any;

subscriber: Subscriber<any>;

}

// 最终执行的是 subcriber 的 error 方法。

function dispatch({ error, subscriber }: DispatchArg) {

subscriber.error(error);

}

**二、iif & defer**

iif 和 defer 的表现是一致的。

defer 的主要作用是延后了具体 Observable 的生成，是一个 Lazy Observable Factory。

iif 则是缩小了 defer 的表达范围，主要作用是增强了Rx 的命令式的语义。

let test = false;

const observableA = iif(

() => test,

of('1'),

of('2'),

);

const observableB = defer(function() {

return test ? of('1') : of('2');

});

**1 iif Source Code**

看到 iif 的源码的那一刻我震惊了，什么叫大道至简（战术后仰）。

export function iif<T = never, F = never>(

condition: () => boolean,

trueResult: SubscribableOrPromise<T> = EMPTY,

falseResult: SubscribableOrPromise<F> = EMPTY

): Observable<T|F> {

// 直接调用了 defer

return defer(() => condition() ? trueResult : falseResult);

}

**2 defer Source Code**

defer 原理上比较简单：在构造 Observable 的时候，在传入的订阅函数中返回一个 Subscription。那么在这个传入的订阅函数中，defer 的过程分为以下三步：

（1）调用工厂，获取输入数据。

（2）调用 from 将数据转换成一个 observable

（3）返回这个 observable 的订阅。

export function defer<R extends ObservableInput<any> | void>(observableFactory: () => R): Observable<ObservedValueOf<R>> {

return new Observable<ObservedValueOf<R>>(subscriber => {

let input: R | void;

try {

// 调用工厂函数，获取输入的数据。

input = observableFactory();

} catch (err) {

subscriber.error(err);

return undefined;

}

// 通过 from 将 input 转换为 observable。

const source = input ? from(input as ObservableInput<ObservedValueOf<R>>) : empty();

// 返回一个订阅器到外部。

return source.subscribe(subscriber);

});

}

其中的 ObservedValueOf 是这样定义的，使用了 ts 的 infer 来推导出 ObservableInput\<T> 中 T 的具体类型。

export type ObservedValueOf<OV> = OV extends ObservableInput<infer T> ? T : never;

**from**

from 提供了一种映射的功能，可以将传入的数据映射成 Observables 。它可以接受以下参数：

原生数组 和 Iterable<T>

dom 迭代器

Promise<T>

Observable<T>

稍微的修剪了一下，源码如下：

export function from<T>(input: ObservableInput<T>): Observable<T> { return new Observable<T>(subscribeTo(input)); }

它直接创建一个新的 Observable，并且调用了 subscribeTo ，根据输入类型，对输入进行不同的处理。

如果输入是 Observable，调用 subscribeToObservable。

如果输入是原生数组，调用 subscribeToArray。

如果输入是 Promise，调用 subscribeToPromise。

如果输入是生成器，调用 subscribeToIterable。

**subscribeToArray**

如果输入是原生数组或者是实现了数组功能的数据结构，那么直接调用 subscriber.next 把所有数据依次发送给订阅者。

export const subscribeToArray = <T>(array: ArrayLike<T>) => (subscriber: Subscriber<T>) => {

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

subscriber.next(array[i]);

}

subscriber.complete();

};

**subscribeToObservable**

如果输入是 Obervable，那么要通过一个特定的 Symbol 取出 Observable，然后再订阅它。

（基于 Symbol 的特性，当前很多项目都会使用一个固定的 Symbol 对特定数据取值，来验证这个数据是不是符合类型）。

export const subscribeToObservable = <T>(obj: any) => (subscriber: Subscriber<T>) => {

const obs = obj[Symbol\_observable]();

if (typeof obs.subscribe !== 'function') {

throw new TypeError('Provided object does not correctly implement Symbol.observable');

} else {

return obs.subscribe(subscriber);

}

};

**subscribeToPromise**

如果输入是一个 Promise，那么通过 then 获取到 Promise 的内容，并将内容发送给订阅者。

export const subscribeToPromise = <T>(promise: PromiseLike<T>) => (subscriber: Subscriber<T>) => {

promise.then(

(value) => {

if (!subscriber.closed) {

subscriber.next(value);

subscriber.complete();

}

},

(err: any) => subscriber.error(err)

);

return subscriber;

};

**subscribeToIterable**

生成器跟数组的方式类似，也是通过循环的方式将数据发送给订阅者。

export const subscribeToIterable = <T>(iterable: Iterable<T>) => (subscriber: Subscriber<T>) => {

const iterator = (iterable as any)[Symbol\_iterator]();

do {

let item: IteratorResult<T>;

try {

item = iterator.next();

} catch (err) {

subscriber.error(err);

return subscriber;

}

if (item.done) {

subscriber.complete();

break;

}

subscriber.next(item.value);

if (subscriber.closed) {

break;

}

} while (true);

return subscriber;

};

**generate**

generate 可以让你用一种类似 for 循环的方式获得数据流。不过，我目前还没有遇到过非常需要这种方式生成流的方式，如果你遇到这种情况，欢迎交流。一般来说，我习惯于这样调用它。

const observable = generate({

initialState: 1,

condition: x => x < 5,

iterate: x => x + 1,

});

observable.subscribe((value) => {

console.log(value);

});

// 打印结果

// 1

// 2

// 3

// 4

原来的源码包含了较多的参数判断，把内部逻辑梳理一下，实际上就是分为三个大步骤：

判断结束条件， 如果为假代表已经结束，则应该完成订阅，否则进行下一步。

发送数据订阅给到订阅者。

调用迭代方法，生成下一组数据，重复第一步。

export function generate<S>(options: GenerateOptions<S>): Observable<S> {

const initialState = options.initialState;

const condition = options.condition;

const iterate = options.iterate;

// 返回 Observable

return new Observable<S>(subscriber => {

let state = initialState;

try {

while (true) {

// 判断结束条件

if (condition && !condition(state)) {

subscriber.complete();

break;

}

// 发送数据给订阅者

subscriber.next(state);

// 调用迭代，获取下一组数据

state = iterate(state);

if (subscriber.closed) {

break;

}

}

} catch (err) {

subscriber.error(err);

}

return undefined;

});

}

其中 GenerateOptions 包含了三个成员，initialState，condition 以及 iterate 。

export interface GenerateOptions<S> {

// 初始状态

initialState: S;

// 结束条件

condition?: (x: S) => boolean;

// 迭代方式

iterate: (x: S) => S;

}

**range**

range 可以创建一个给定范围的数字流。这个主要就是提供了一个简单的语义化函数，主要就是通过循环给订阅者喂数据。

export function range(start: number = 0, count?: number): Observable<number> {

return new Observable<number>(subscriber => {

if (count === undefined) {

count = start;

start = 0;

}

for (let index = 0; index < count; ++index) {

subscriber.next(start + index);

if (subscriber.closed) {

break;

}

}

return undefined;

});

}

**fromEvent & fromEventPattern**

**fromEvent**

fromEvent 是的 Observable 可以封装一系列的系统事件。既可以接受 NodeJS EventEmitter，也可以接受 DOM EventTarget， JQuery-like event target, NodeList 或者 HTMLCollection 等浏览器对象。

const clicksA = fromEvent(document, 'click');

const clicksB = fromEvent($(document), 'click');

clicksA.subscribe(x => console.log('A: ', x));

clicksB.subscribe(x => console.log('B: ', x));

// 每当点击一下页面，都会打印出 event 。

它的实现很简单，根据 target 的对象类型调用其对应的事件监听函数，然后通过 subscriber 调用 next 获取到订阅的输出。为了方便阅读，我稍微的改了一下，让 fromEvent 只支持 DOM EventTarget。

export interface HasEventTargetAddRemove<E> {

addEventListener(type: string, listener: ((evt: E) => void) | null, options?: boolean | AddEventListenerOptions): void;

removeEventListener(type: string, listener?: ((evt: E) => void) | null, options?: EventListenerOptions | boolean): void;

}

// 一个只支持 DOM EventTarget 的 fromEvent

export function fromEvent<T>(

target: HasEventTargetAddRemove<T>,

eventName: string,

options?: EventListenerOptions,

): Observable<T> {

return new Observable<T>(subscriber => {

// 处理结果

const handler = (...e: T[]) => subscriber.next(e.length === 1 ? e[0] : e);

// 调用 addEventListener，并让其在 handler 中处理。

target.addEventListener(eventName, handler, options);

// 取消订阅的时候，直接调用 removeEventListener 对 dom 取消订阅。

// 返回的是一个函数，这个函数负责了取消订阅的时，所做的内容。

return () => {

target.removeEventListener(eventName, handler, options);

};

});

}

上面的代码可以分解成这三个步骤：

在闭包中创建一个 handler 函数，handler 函数最终会调用 subscriber.next。

为 target 添加指定事件监听。

为 subscriber 添加一个销毁 target 事件监听的逻辑。

对于其他的事件监听，不再赘述，流程完全是一样。

**fromEventPattern**

fromEventPattern 则是对 fromEvent 的泛化。

function addClickHandler(handler) {

document.addEventListener('click', handler);

}

function removeClickHandler(handler) {

document.removeEventListener('click', handler);

}

const clicks = fromEventPattern(

addClickHandler,

removeClickHandler

);

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

// 点击的时候，就会输出点击事件。

它的源码的与 fromEvent 类似。

export type NodeEventHandler = (...args: any[]) => void;

export function fromEventPattern<T>(

addHandler: (handler: NodeEventHandler) => any,

removeHandler?: (handler: NodeEventHandler, signal?: any) => void,

): Observable<T | T[]> {

return new Observable<T | T[]>(subscriber => {

const handler = (...e: T[]) => subscriber.next(e.length === 1 ? e[0] : e);

// 有一点不同的地方在于，获取了返回值 addHandler 的返回值

let retValue: any;

try {

retValue = addHandler(handler);

} catch (err) {

subscriber.error(err);

return undefined;

}

if (!isFunction(removeHandler)) {

return undefined;

}

// 然后在这里传入 removeHandler 中

return () => removeHandler(handler, retValue);

});

}

**bindCallback， bindNodeCallback**

它们都是一种特殊的 Operator ，思路应该是源于 Function.bind ，提供一种转换操作，将带有回调的函数转换成 Observable Factory。

function setTimeoutWithCallback(callback: () => void) {

setTimeout(() => {

callback();

}, 2000);

}

const obfactory = bindCallback(setTimeoutWithCallback);

const ob1 = obfactory();

const ob2 = obfactory();

const now = Date.now();

ob1.subscribe(() => {

console.log('ob1' + (Date.now() - now));

});

setTimeout(() => {

ob1.subscribe(() => {

console.log('ob1 later: ' + (Date.now() - now));

});

ob2.subscribe(() => {

console.log('ob2: ' + (Date.now() - now));

});

}, 3000);

// 打印结果：

// ob1: 2001

// ob1 later: 3004

// ob2: 5008

以下是 bindNodeCallback 的例子。

/\*

file: ~/desktop/test.json

{ "name": "Hello World" }

\*/

import \* as fs from 'fs';

const readerFactory = bindNodeCallback(fs.readFile);

const reader$ = readerFactory('./src/person.json');

reader$.subscribe({

next: (value) => console.log(value.toString()),

error: (err) => console.log(err),

complete: () => console.log('complete')

});

// 如果没有错误，打印结果如下：

// { name: 'Hello World' }

// complete

// 如果有错误，打印结果如下：

// [Error: ENOENT: no such file or directory, open './src/person.json'] {

// errno: -2,

// code: 'ENOENT',

// syscall: 'open',

// path: './src/person'

// }

**bindCallback 和 bindNodeCallback 的源码非常类似。**

export function bindCallback<T>(

callbackFunc: Function

): (...args: any[]) => Observable<T> {

return function (this: any, ...args: any[]): Observable<T> {

const context = this;

//

let subject: AsyncSubject<T>;

return new Observable<T>(subscriber => {

if (!subject) {

subject = new AsyncSubject<T>();

const handler = (...innerArgs: any[]) => {

subject.next(innerArgs.length <= 1 ? innerArgs[0] : innerArgs);

subject.complete();

};

try {

callbackFunc.apply(context, [...args, handler]);

} catch (err) {

subject.error(err);

}

}

return subject.subscribe(subscriber);

});

};

}

bindCallback 和 bindNodeCallback 的源码唯一不同的地方就是在于 handler 这个函数处理的内容不同，bindNodeCallback 传入的函数的回调，第一个参数为是错误信息。

const handler = (...innerArgs: any[]) => {

const err = innerArgs.shift();

// 如果第一个参数存在，说明有问题。

if (err) {

subject.error(err);

return;

}

subject.next(innerArgs.length <= 1 ? innerArgs[0] : innerArgs);

subject.complete();

};

源码中比较有趣的地方在于，创建的时候，返回的工厂函数包含了一个 AsyncSubject。这个 AsyncSubject 保存了已经到来数据，可以看看例子中，ob1 被订阅了2次，第二次订阅后实际上是立刻就能拿到返回值；而 ob2 仍要执行一次 setTimeoutWithCallback。这种设计与这个 bind 的语义相吻合。

**interval & timer**

上面的 operators 中，我已经把 scheduler 相关的内容进行了裁剪，基本上与 scheduler 无关。而 interval 和 timer 都必须通过 scheduler 来相应的定时操作，所以这部分放到了最后。它们是用于创建定时数据源的 operators 。

interval： 传入的参数表示每隔指定毫秒发送一条数据。

timer：传入的第一个参数是指第一条发送数据的时间间隔，第二个参数是指后续数据发送的间隔。

const observableA = interval(1000).pipe(take(2));

const observableB = timer(500, 1000).pipe(take(3));

console.log('hello');

observableA.subscribe(value => {

console.log('A: ' + value);

});

observableB.subscribe(value => {

console.log('B: ' + value);

});

// 打印结果

// hello

// B: 0

// A: 0

// B: 1

// A: 1

// B: 2

interval 和 timer 都使用了一个默认的异步调度器，这个异步调度器主要是通过 setInterval 来实现相应的功能，实际上 Rx 把异步调度器通过 interval 和 timer 转化成 Observable 的形式提供到给用户。

**timer Source Code**

timer 的实现如下图所示。它首先创建了一个 Observable ，然后在订阅函数中，返回调度器的订阅。在这里， scheduler 的 schedule 函数返回了一个 Subscription。

export function timer(

dueTime: number | Date = 0,

period: number,

scheduler: SchedulerLike = async

): Observable<number> {

return new Observable(subscriber => {

let due = 0;

// 判断是不是 Date 类型

if (dueTime instanceof Date) {

due = +dueTime - scheduler.now();

}

// 判断是不是 number 类型

if (isNumeric(dueTime)) {

due = dueTime as number;

}

// 此处调用跟 interval 类似。

return scheduler.schedule(dispatch, due, {

index: 0, period, subscriber

});

});

}

dispatch 实际上是一个递归函数，这个函数绑定了 SchedulerAction ，通过传入订阅者，使得 Action 内部的 setInterval 能够一直调用 subscriber.next。

interface TimerState {

index: number;

period: number;

subscriber: Subscriber<number>;

}

function dispatch(this: SchedulerAction<TimerState>, state: TimerState) {

const { index, period, subscriber } = state;

subscriber.next(index);

if (subscriber.closed) {

return;

} else if (period === -1) {

return subscriber.complete();

}

state.index = index + 1;

this.schedule(state, period);

}

**interval Source Code**

以下是 interval 的源码。

export function interval(period = 0): Observable<number> {

if (!isNumeric(period) || period < 0) {

period = 0;

}

const scheduler = async;

return new Observable<number>(subscriber => {

// 订阅器接收 scheduler 的订阅结果。

subscriber.add(

scheduler.schedule(dispatch, period, { subscriber, counter: 0, period })

);

return subscriber;

});

}

仔细的分析上面的代码，我发现 interval 的实现实际上就是 timer 的一个约束版本，它可以改写成这样。

export function interval(

period = 0,

scheduler: SchedulerLike = async,

): Observable<number> {

if (!isNumeric(period) || period < 0) {

period = 0;

}

return timer(period, period, sch);

}

本篇小结

整体而言，这部分的源码并没有写得很绕，删去了 scheduler 相关的内容后，逻辑立刻就变得清晰了起来。同时，从源码的风格上可以看到它们由不同的人来编写。

最后，限于本人的水平，如果文章中有错误的地方，欢迎指正。