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。