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

/\*\* 内部实现细节，不要直接使用 \*/

public \_isScalar: boolean = false;

/\*\* @deprecated 这是一个内部实现细节，不要使用. \*/

source: Observable<any>;

/\*\* @deprecated 这是一个内部实现细节，不要使用 \*/

operator: Operator<any, T>;

/\*\*

\* @constructor

\* @param {Function}

当Observable最初被订阅时调用的函数。此函数被赋予一个订阅服务器，可以对其`next`ed `新值，或调用`error`方法引发错误，或调用`complete`方法来通知成功完成。

\*/

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

if (subscribe) {

this.\_subscribe = subscribe;

}

}

// HACK: 因为TypeScript也继承了静态属性，所以我们必须与TypeScript对抗，这样Subject就可以有一个不同的静态创建签名

/\*\*

\*通过调用Observable的构造函数来创建一个新的冷Observable

\* @static true

\* @owner Observable

\* @method create

\* @param {Function} subscribe? 要传递给Observable构造函数的订阅函数

\* @return {Observable} 返回一个新的 cold observable

\* @nocollapse

\* @deprecated use new Observable() instead

\*/

static create: Function = <T>(subscribe?: (subscriber: Subscriber<T>) => TeardownLogic) => {

return new Observable<T>(subscribe);

}

/\*\*

\* 创建一个新的Observable，将这个Observable作为源，并将传递的操作符定义为这个新Observable的操作符。

\* @method lift

\* @param {Operator} 操作符:定义接受可观察对象的操作符

\* @return {Observable} 一个应用了Operator的新可观察对象

\*/

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;

/\*\* @deprecated 使用一个观察者而不是一个完整的回调\*/

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

/\*\* @deprecated 使用观察者而不是错误回调\*/

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

/\*\* @deprecated 使用一个观察者而不是一个完整的回调\*/

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

subscribe(next?: (value: T) => void, error?: (error: any) => void, complete?: () => void): Subscription;

/\*\*

\* 调用一个可观察对象的执行，并注册它将发出的通知的Observer处理程序.

\*

\* <span class="informal">当你拥有所有这些可观察对象，但仍然什么都没有发生时使用它.</span>

\*

\* ' subscribe '不是一个普通的操作符，而是一个调用Observable内部的' subscribe '函数的方法。例如，它可能是一个你传递给Observable构造函数的函数，但大多数情况下，它是一个库实现，它定义了Observable会发出什么，以及什么时候会发出。这意味着调用' subscribe '实际上是在Observable开始工作的时候，而不是像通常想的那样，是在它被创建的时候。 \*

\* 除了启动Observable的执行之外，这个方法还允许你监听Observable发出的值，以及它完成或出错的时候。您可以通过以下两种方式实现此目的。

\*

\*第一种方法是创建一个实现{@link Observer}接口的对象。它应该有该接口定义的方法，但请注意，它应该只是一个常规的JavaScript对象，你可以以任何你想要的方式创建自己(ES6类，经典函数构造器，对象字面量等)。特别是不要尝试使用任何RxJS实现细节来创建观察者——你不需要它们。还请记住，您的对象不必实现所有方法。如果您发现自己创建了一个什么都不做的方法，您可以简单地忽略它。但是请注意，如果没有提供' error '方法，所有的错误都不会被捕获。 \*

\*第二种方法是完全放弃Observer对象，简单地提供回调函数来代替其方法。这意味着你可以为' subscribe '提供三个函数作为参数，其中第一个函数相当于' next '方法，第二个是' error '方法，第三个是' complete '方法。就像在观察者的情况下，如果你不需要监听一些东西，你可以省略一个函数，最好通过传递' undefined '或' null '，因为' subscribe '识别这些函数在函数调用中放置的位置。当涉及到' error '函数时，就像之前一样，如果没有提供，由Observable触发的错误将被抛出。

\*

\* 无论您使用哪种类型的调用' subscribe '，在这两种情况下它都会返回一个Subscription对象。这个对象允许你对它调用' unsubscribe '，这反过来会停止Observable所做的工作，并清理Observable所使用的所有资源。请注意，取消订阅不会调用提供给' subscribe '函数的' complete '回调函数，这是为来自Observable的常规完成信号保留的。

\*

请记住，提供给' subscribe '的回调不能保证被异步调用。可观察对象本身决定什么时候调用这些函数。例如，{@link of}默认同步发送它的所有值。一定要检查文档，看看给定的Observable在订阅时是如何表现的，以及它的默认行为是否可以通过' scheduler '来修改。 \*

## Example

### Subscribe with an Observer

```ts

import { of } from 'rxjs';

const sumObserver = {

sum: 0,

next(value) {

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

this.sum = this.sum + value;

},

error() {

// 实际上我们可以去掉这个方法，因为我们现在并不关心错误。

},

complete() {

console.log('Sum equals: ' + this.sum);

}

};

of(1, 2, 3) // 同步发出1,2,3，然后完成。

.subscribe(sumObserver);

// Logs:

// "Adding: 1"

// "Adding: 2"

// "Adding: 3"

// "Sum equals: 6"

```

### Subscribe with functions

import { of } from 'rxjs'

let sum = 0;

of(1, 2, 3).subscribe(

value => {

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

sum = sum + value;

},

undefined,

() => console.log('Sum equals: ' + sum)

);

// Logs:

// "Adding: 1"

// "Adding: 2"

// "Adding: 3"

// "Sum equals: 6"

```

### Cancel a subscription

```ts

import { interval } from 'rxjs';

const subscription = interval(1000).subscribe(

num => console.log(num),

undefined,

() => {

// 不会被召唤, 甚至在取消订阅的时候.

console.log('completed!');

}

);

setTimeout(() => {

subscription.unsubscribe();

console.log('unsubscribed!');

}, 2500);

// Logs:

// 0 after 1s

// 1 after 2s

// "unsubscribed!" after 2.5s

```

\* @param {Observer|Function} observerOrNext(可选)一个有方法被调用的观察者，或者三个可能的处理程序中的第一个，它是每个从订阅的Observable中发出的值的处理程序。

\* @param {Function} error (可选)由错误导致的终端事件的处理程序。如果没有提供错误处理程序，则将错误作为未处理抛出。

\* @param {Function} complete (可选)由于成功完成而产生的终端事件的处理程序。

\* @return {ISubscription} 对已注册处理程序的订阅引用

\* @method subscribe

\*/

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

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

complete?: () => void): Subscription {

const { operator } = this;

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

if (operator) {

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

} else {

sink.add(

this.source || (config.useDeprecatedSynchronousErrorHandling && !sink.syncErrorThrowable) ?

this.\_subscribe(sink) :

this.\_trySubscribe(sink)

);

}

if (config.useDeprecatedSynchronousErrorHandling) {

if (sink.syncErrorThrowable) {

sink.syncErrorThrowable = false;

if (sink.syncErrorThrown) {

throw sink.syncErrorValue;

}

}

}

return sink;

}

/\*\* @deprecated This is an internal implementation detail, do not use. \*/

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

try {

return this.\_subscribe(sink);

} catch (err) {

if (config.useDeprecatedSynchronousErrorHandling) {

sink.syncErrorThrown = true;

sink.syncErrorValue = err;

}

if (canReportError(sink)) {

sink.error(err);

} else {

console.warn(err);

}

}

}

/\*\*

\* @method forEach

\* @param {Function} 接下来是可观察对象发出的每个值的处理程序

\* @param {PromiseConstructor} [promiseCtor] 用于实例化Promise的构造函数

\* @return {Promise} 一个承诺，它要么解析可观察完成，要么用已处理的错误拒绝

\*/

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

promiseCtor = getPromiseCtor(promiseCtor);

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

// 必须在单独的语句中声明，以避免在访问由于时间死区而导致的闭包中的订阅时出现ReferenceError。

let subscription: Subscription;

subscription = this.subscribe((value) => {

try {

next(value);

} catch (err) {

reject(err);

if (subscription) {

subscription.unsubscribe();

}

}

}, reject, resolve);

}) as Promise<void>;

}

/\*\* @internal This is an internal implementation detail, do not use. \*/

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

const { source } = this;

return source && source.subscribe(subscriber);

}

//' if '和' throw '是特殊的雪花，编译器将它们视为保留字。不赞成使用iif和throwError函数。

/\*\*

\* @nocollapse

\* @deprecated In favor of iif creation function: import { iif } from 'rxjs';

\*/

static if: typeof iif;

/\*\*

\* @nocollapse

\* @deprecated In favor of throwError creation function: import { throwError } from 'rxjs';

\*/

static throw: typeof throwError;

/\*\*

\* An interop point defined by the es7-observable spec https://github.com/zenparsing/es-observable

\* @method Symbol.observable

\* @return {Observable} this instance of the observable

\*/

[Symbol\_observable]() {

return this;

}

/\* tslint:disable:max-line-length \*/

pipe(): Observable<T>;

pipe<A>(op1: OperatorFunction<T, A>): Observable<A>;

pipe<A, B>(op1: OperatorFunction<T, A>, op2: OperatorFunction<A, B>): Observable<B>;

pipe<A, B, C, D, E, F, G, H, I>(op1: OperatorFunction<T, A>, op2: OperatorFunction<A, B>, op3: OperatorFunction<B, C>, op4: OperatorFunction<C, D>, op5: OperatorFunction<D, E>, op6: OperatorFunction<E, F>, op7: OperatorFunction<F, G>, op8: OperatorFunction<G, H>, op9: OperatorFunction<H, I>, ...operations: OperatorFunction<any, any>[]): Observable<{}>;

/\* tslint:enable:max-line-length \*/

/\*\*

\* 用于将函数运算符缝合成一条链。

\* @method pipe

\* @return {Observable}所有被调用的操作符的可观察结果都是按照它们被传入的顺序调用的。

\*

\* ### Example

\* ```ts

\* import { interval } from 'rxjs';

\* import { map, filter, scan } from 'rxjs/operators';

\*

\* interval(1000)

\* .pipe(

\* filter(x => x % 2 === 0),

\* map(x => x + x),

\* scan((acc, x) => acc + x)

\* )

\* .subscribe(x => console.log(x))

\* ```

\*/

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

if (operations.length === 0) {

return this as any;

}

return pipeFromArray(operations)(this);

}

/\* tslint:disable:max-line-length \*/

toPromise<T>(this: Observable<T>): Promise<T>;

toPromise<T>(this: Observable<T>, PromiseCtor: typeof Promise): Promise<T>;

toPromise<T>(this: Observable<T>, PromiseCtor: PromiseConstructorLike): Promise<T>;

/\* tslint:enable:max-line-length \*/

toPromise(promiseCtor?: PromiseConstructorLike): Promise<T> {

promiseCtor = getPromiseCtor(promiseCtor);

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

let value: any;

this.subscribe((x: T) => value = x, (err: any) => reject(err), () => resolve(value));

}) as Promise<T>;

}

}

export interface Subscribable<T> {

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

/\*\* @deprecated 使用一个观察者而不是一个完整的回调 \*/

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

/\*\* @deprecated 使用观察者而不是错误回调 \*/

subscribe(next: null | undefined, error: (error: any) => void, complete?: () => void): Unsubscribable;

/\*\* @deprecated 使用一个观察者而不是一个完整的回调 \*/

subscribe(next: (value: T) => void, error: null | undefined, complete: () => void): Unsubscribable;

subscribe(next?: (value: T) => void, error?: (error: any) => void, complete?: () => void): Unsubscribable;

}