

Aunque los Observables constituyen sus cimientos, la verdadera utilidad de RxJS es gracias a sus Operadores. Los Operadores son las piezas esenciales que permiten la composición de código complejo y asíncrono, de manera declarativa.

Qué son los Operadores

Los Operadores son funciones. Hay dos tipos de operadores:

Los Operadores de tubería se pueden utilizar mediante la sintaxis `observableInstance.pipe(operator())`. Entre ellos se incluyen `filter()` y `mergeMap()`. Cuando son llamados, *no modifican* la instancia del Observable existente. En su lugar, devuelven un Observable nuevo, cuya lógica de suscripción está basada en la del primer Observable.

Un Operador de tubería es una función que recibe un Observable y devuelve otro Observable. Es una operación pura: el Observable anterior no se modifica.

Un Operador de tubería es esencialmente una función pura que recibe un Observable como entrada y genera otro Observable como salida. Una suscripción al Observable de salida supone también una suscripción al Observable de entrada.

El otro tipo de operador es el de Creación. Estos operadores pueden llamarse como si fuesen funciones independientes para crear un nuevo Observable. Por ejemplo: `of(1, 2, 3)` crea un Observable que emitirá los valores 1, 2 y 3, de forma consecutiva. Entraremos en más detalle sobre de los Operadores de Creación en una sección posterior.

Por ejemplo, el operador `map` es análogo al método de Array que lleva el mismo nombre. Al igual que `[1, 2, 3].map(x => x * x)` produce `[1, 4, 9]`, el Observable creado de la siguiente manera:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

map(x => x * x)(of(1, 2, 3)).subscribe((v) => console.log(`valor: ${v}`));

// Logs:
// valor: 1
// valor: 4
// valor: 9
```

Emitirá 1, 4, 9.

Otro operador muy útil es `first`:

```
import { of } from "rxjs";
import { first } from "rxjs/operators";

first()(of(1, 2, 3)).subscribe((v) => console.log(`valor: ${v}`));

// Logs:
// valor: 1
```

Hay que tener en cuenta que `map` tiene que ser construido en el momento, ya que tiene que recibir la función de proyección. Por el contrario, `first` podría ser una constante, aunque también se construye en el momento. Como norma general, todos los operadores se construyen, independientemente de que necesiten recibir o no argumentos.

Piping

Los Operadores de tubería son funciones, por lo que pueden utilizarse como funciones normales: `op()(obs)`. Sin embargo, en la práctica, tienden a utilizarse muchos operadores al mismo tiempo, por lo que hacer esto hará que nuestro código sea ilegible: `op4()(op3()(op2()(op1()(obs))))`. Por esta razón, los Observables tienen un método llamado `.pipe()` que cumple esta misma función, de una forma mucho más legible:

```
obs.pipe(op1(), op2(), op3(), op3());
```

Copy

Por motivos estilísticos, `op()(obs)` nunca se utiliza, aunque solo se utilice un operador. `obs.pipe(op())` es universalmente preferible.

Operadores de Creación

¿Qué son los Operadores de Creación? Diferentes a los Operadores de tubería, los Operadores de Creación son funciones que se pueden utilizar para crear un Observable con un comportamiento predeterminado común, o mediante la unión de otros Observables.

Un ejemplo clásico de un Operador de Creación es la función `interval`. Recibe un número (no un Observable) como argumento de entrada, y produce un Observable como salida:

```
import { interval } from "rxjs";  
  
const observable = interval(1000 /* número de milisegundos */);
```

Copy

Se puede ver la [lista completa de Operadores estáticos de Creación aquí](#).

Observables de Orden Superior

Los Observables suelen emitir valores ordinarios como cadenas o números, pero a veces (y más a menudo de lo que pudiera parecer), es necesario manejar un Observable que emite Observables, también conocido como un Observable de orden superior.

Por ejemplo, se podría tener un Observable que emite las URLs de unos ficheros que se quieren ver. El código sería algo parecido a lo siguiente:

```
const fileObservable = urlObservable.pipe(map((url) => http.get(url)));
```

Copy

`http.get()` retorna un Observable (probablemente de cadenas o de arrays de cadenas) por cada URL individual. Esto es un Observable de Observables, un Observable de orden superior.

Pero, ¿Cómo se trabaja con un Observable de orden superior? Normalmente, se 'aplasta', para convertirlo en un Observable normal. Por ejemplo:

```
const fileObservable = urlObservable.pipe(  
  map((url) => http.get(url)),  
  concatAll()  
);
```

Copy

El operador `concatAll()` operator se suscribe cada Observable 'interno' producido por el Observable 'externo', y copia todos los valores que emite hasta que el Observable interno se completa. Entonces, se suscribe al siguiente Observable interno y repite el proceso. De esta manera, se concatenan todos los valores emitidos por todos los Observables internos. Otros operadores de combinación útiles son:

- `mergeAll()` – se suscribe a cada Observable interno en cuanto lo recibe, emitiendo cada uno de sus valores en cuanto lo recibe.
- `switchAll()` – se suscribe al primer Observable interno en cuanto lo recibe y emite cada uno de sus valores en cuanto lo recibe, pero en cuanto recibe el siguiente Observable interno, cancela la suscripción al Observable interno anterior y se suscribe al nuevo.
- `exhaust()` – se suscribe al primer Observable interno en cuanto lo recibe, y emite cada uno de sus valores en cuanto lo recibe, descartando todos los Observables internos nuevos que recibe hasta que el primer Observable interno se haya completado.

Al igual que hacen muchas bibliotecas de array, combinando `map()` y `flat()` (o `flatten()`) en un solo `flatMap()`, en RxJS también existen los equivalentes de proyección de todos los operadores de combinación:

`concatMap()`
`mergeMap()`

[switchMap\(\)](#)[exhaustMap\(\)](#)

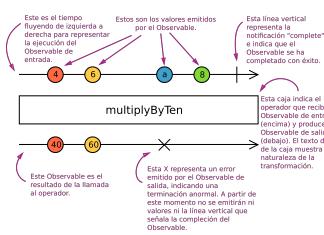
Diagramas de canicas

Para explicar el funcionamiento de los operadores, una descripción por escrito suele ser insuficiente. Muchos operadores están relacionados con el tiempo, como por ejemplo [delay](#), [sample](#), [throttle](#) o [debounce](#). Cada uno de estos operadores trata las emisiones de manera diferente.

Los diagramas de canicas son representaciones visuales de cómo funcionan los operadores, incluyendo el Observable de entrada, el operador y sus parámetros y el Observable de salida. El diagrama describe cómo se emiten los valores (canicas) al ejecutarse el Observable.

En un diagrama de canicas, el tiempo fluye hacia la derecha.

A continuación se puede ver la anatomía de un diagrama de canicas:



En esta documentación se utilizan los diagramas de canicas para explicar el funcionamiento de los operadores. También son muy útiles en otros contextos, como por ejemplo en una pizarra o incluso a la hora de hacer tests unitarios (en forma de diagramas ASCII.)

Categorías de operadores

Hay muchos tipos diferentes de operadores, divididos en las siguientes categorías: creación, transformación, filtración, combinación, multidifusión, gestión de errores, utilidad, etc. A continuación están listados por categoría todos los operadores:

Operadores de creación

- [ajax](#)
- [bindCallback](#)
- [bindNodeCallback](#)
- [defer](#)
- [empty](#)
- [from](#)
- [fromEvent](#)
- [fromEventPattern](#)
- [generate](#)
- [interval](#)
- [of](#)
- [range](#)
- [throwError](#)
- [timer](#)
- [iif](#)

Operadores de Combinación-Creación

Estos son operadores de creación que también tienen funcionalidad de combinación -- emitiendo valores de varios Observables fuente.

- [combineLatest](#)
- [concat](#)

forkJoin
merge
race
zip

Operadores de Transformación

buffer
bufferCount
bufferTime
bufferToggle
bufferWhen
concatMap
concatMapTo
exhaust
exhaustMap
expand
groupBy
map
mapTo
mergeMap
mergeMapTo
mergeScan
pairwise
partition
pluck
scan
switchMap
switchMapTo
window
windowCount
windowTime
windowToggle
windowWhen

Operadores de Filtración

audit
auditTime
debounce
debounceTime
distinct
distinctKey
distinctUntilChanged
distinctUntilKeyChanged
elementAt
filter
first
ignoreElements
last

- sample
- sampleTime
- single
- skip
- skipLast
- skipUntil
- skipWhile
- take
- takeLast
- takeUntil
- takeWhile
- throttle
- throttleTime

Operadores de Combinación

Ver también la sección anterior de operadores de Combinación-Creación.

- combineAll
- concatAll
- exhaust
- mergeAll
- startWith
- withLatestFrom

Operadores de Multidifusión

- multicast
- publish
- publishBehavior
- publishLast
- publishReplay
- share

Operadores de Gestión de Errores

- catchError
- retry
- retryWhen

Operadores de Utilidad

- tap
- delay
- delayWhen
- dematerialize
- materialize
- observeOn
- subscribeOn
- timeInterval
- timestamp
- timeout

`timeoutWith`
`toArray`

Operadores Condicionales y Booleanos

`defaultIfEmpty`
`every`
`find`
`findIndex`
`isEmpty`

Operadores Matemáticos y de Agregación

`count`
`max`
`min`
`reduce`

Creación de operadores personalizados

Se puede utilizar la función `pipe()` para crear operadores nuevos

En el caso de tener una secuencia de operadores que se reutilice en distintas partes del código, se puede utilizar la función `pipe` para extraer dicha secuencia a un operador nuevo. Incluso aunque la secuencia no se reutilice muy a menudo, extraerla a un operador puede mejorar la legibilidad.

Por ejemplo, se puede crear una función que ignore los valores impares y multiplique por dos los valores pares:

```
import { pipe } from "rxjs";
import { filter, map } from "rxjs";

function discardOddDoubleEven() {
  return pipe(
    filter((v) => !(v % 2)),
    map((v) => v + v)
  );
}
```

[Copy](#)

(La función `pipe()` es [análoga](#) al método `.pipe()` de un Observable.)

Creación de un nuevo operador desde 0

Es más complicado, pero si se necesita un operador que no se pueda crear combinando operadores ya existentes (raramente ocurre esto), se puede crear un operador desde 0 utilizando el constructor Observable:

```
import { Observable } from "rxjs";

function delay(delayInMillis) {
  return (observable) =>
    new Observable((observer) => {
      // Esta función se llamará con cada suscripción a este Observable
      const allTimerIDs = new Set();
      const subscription = observable.subscribe({
        next(value) {
```

[Copy](#)

```
const timerID = setTimeout(() => {
  observer.next(value);
  allTimerIDs.delete(timerID);
}, delayInMillis);
allTimerIDs.add(timerID);
},
error(err) {
  observer.error(err);
},
complete() {
  observer.complete();
},
});
// El valor del return es la función teardown, que se invoca cada vez que se cancela la suscripción al Observable
return () => {
  subscription.unsubscribe();
  allTimerIDs.forEach((timerID) => {
    clearTimeout(timerID);
  });
};
});
}
}
```

Es importante que:

1. Se implementen las tres funciones Observer, `next()`, `error()`, y `complete()` a la hora de suscribirse al Observable
2. Se implemente una función *teardown* que se encargue de limpiar (en este caso, cancelando la suscripción y encargándose de liberar el ID)
3. La función que se le pasa al constructor del Observable retorne la función *teardown*.

Por supuesto, este ejemplo es simplemente para mostrar cómo utilizar el constructor del Observable; el operador `delay` ya existe.