Sujetos / Subjects

¿Qué es un Sujeto? Un Sujeto RxJS es un tipo especial de Observable que permite la multidifusión de valores a muchos Observadores. Mientras los Observables simples son de monodifusión (cada Observador suscrito es propietario de una ejecución independiente del Observable), los Sujetos son de multidifusión.

> *Un Sujeto es como un Observable, pero permite la multidifusión a muchos Observadores. Los Sujetos son como EventEmitters: mantienen un registro de múltiples listeners.*

Cada Sujeto es un Observable. Dado un Sujeto, se puede suscribir a él, proporcionando un Observador, que empezará a recibir valores. Desde la perspectiva del Observador, no se puede saber si la ejecución Observable viene de un Observable monodifusión simple o de un Sujeto.

Internamente en el Sujeto, `susbscribe` no invoca una nueva ejecución que emite valores. Simplemente registra el Observador proporcionado en una lista de Observadores, de manera similar a cómo funciona `addListener` en otras bibliotecas y lenguajes.

Cada Sujeto es un Observador. Es un objeto con los métodos `next(v)`, `error(e)`, y `complete()`. Para proporcionarle un nuevo valor al Sujeto, basta con hacer una llamada a `next(value)`, y este será proporcionado a los Observadores registrados en el Sujeto, mediante multidifusión.

En el siguiente ejemplo, se tienen dos Observadores vinculados a un Sujeto, y se le proporcionan varios valores al Sujeto:

```
import { Subject } from 'rxjs';

const subject = new Subject<number>();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

subject.next(1);
subject.next(2);

// Salida:
```

Copy

```
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

Dado que un Sujeto es un Observador, quiere decir que se puede proporcionar un Sujeto como argumento a la función `subscribe` de cualquier Observable, tal y como se muestra a continuación:

```
import { Subject, from } from 'rxjs';

const subject = new Subject<number>();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`)
});
subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`)
});

const observable = from([1, 2, 3]);

observable.subscribe(subject); // Nos podemos suscribir mediante un Sujeto

// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

Utilizando el enfoque anterior, se ha convertido una ejecución Observable monodifusión a multidifusión, mediante el Sujeto. Esto demuestra que los Sujetos son la única forma de conseguir que una ejecución Observable pueda ser compartida entre múltiples Observadores.

Existen varias especializaciones del tipo `Subject`: `BehaviorSubject`, `ReplaySubject` y `AsyncSubject`.

Observables Multidifusión

Un "Observable multidifusión" envía notificaciones a través de un Sujeto que puede tener múltiples suscriptores, mientras que un "Observable monodifusión" simple envía notificaciones a un único Observador.

> *Un Observabe multidifusión utiliza un Sujeto internamente para hacer que varios Observadores vean la misma ejecución Observable.*

Internamente, así es como funciona el operador `multicast`: los Observadores se suscriben a un Sujeto subyacente, y el Sujeto se suscribe al Observable fuente. El siguiente ejemplo es similar al anterior que utilizaba `observable.subscribe(subject)`:

```
import { from, Subject } from "rxjs";
import { multicast } from "rxjs/operators";

const source = from([1, 2, 3]);
const subject = new Subject();
const multicasted = source.pipe(multicast(subject));

// Estos son, internamente, `subject.subscribe({...})`:
multicasted.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});
multicasted.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

// Este es, internamente, `source.subscribe(subject)`:
multicasted.connect();
```

Copy

`multicast` retorna un Observable que parece un Observable normal, pero que funciona como un Sujeto a la hora de realizar una suscripción.

`multicast` retorna un `ConnectableObservable`, que es simplemente un Observable con el método `connect()`.

El método `connect()` es importante para determinar exactamente cuándo se da comienzo a la ejecución Observable compartida. Dado que `connect()` hace `source.subscribe(subject)` internamente, devuelve una Suscripción, que podemos cancelar, para asimismo cancelar la ejecución Observable compartida.

Recuento de referencias

Realizar llamadas a `connect()` de forma manual y manejar la Suscripción suele ser un tanto engorroso. Normalmente, queremos conectar *automáticamente* cuando se recibe el primer Observador, y cancelar la ejecución compartida automáticamente en cuanto el último Observador cancele su suscripción.

Se considera el siguiente ejemplo donde las suscripciones ocurren según se indica en esta lista:

El primer Observador se suscribe al Observable multidifusión.

Se conecta el Observable multidifusión.

El valor `next 0` se envía al primer Observador.

El segundo Observador se suscribe al Observable multidifusión.

El valor `next 1` se envía al primer Observador.

El valor `next 1` se envía al segundo Observador.

El primer Observador cancela la suscripción al Observable multidifusión.

El valor `next 2` se envía al segundo Observador.

El segundo Observador cancela la suscripción al Observable multidifusión.

Se cancela la suscripción a la conexión del Observable multidifusión.

Para poder implementar este escenario con llamadas explícitas a `connect()`, hacemos lo siguiente:

```
import { interval, Subject } from "rxjs";
import { multicast } from "rxjs/operators";

const source = interval(500);
const subject = new Subject();
const multicasted = source.pipe(multicast(subject));
let subscription1, subscription2, subscriptionConnect;

subscription1 = multicasted.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});
// Aquí se debe llamar a `connect()`, ya que el primer suscriptor
// a `multicasted` quiere consumir valores
subscriptionConnect = multicasted.connect();
```

```
setTimeout(() => {
  subscription2 = multicasted.subscribe({
    next: (v) => console.log(`observerB: ${v}`),
  });
}, 600);

setTimeout(() => {
  subscription1.unsubscribe();
}, 1200);

// Aquí se debe cancelar la suscripción a la ejecución compartida del Observ
setTimeout(() => {
  subscription2.unsubscribe();
  subscriptionConnect.unsubscribe(); // Para la ejecución compartida del Obs
}, 2000);
```

Si se quieren evitar llamadas explícitas a `connect()`, se puede utilizar el método `refCount()` (recuento de referencias) del `ConnectableObservable`, que retorna un Observable que lleva la cuenta de cuántos suscriptores tiene. Cuando el número de suscriptores aumente de `0` a `1`, llamará al método `connect()` automáticamente, lo que dará comienzo a la ejecucion compartida. Cuando el número de suscriptores baje de `0` a `1`, se cancelará la suscripción, parando así la ejecución.

> *`refCount` hace que el Observable multidifusión comience a ejecutarse automáticamente en cuanto llega el primer suscriptor, y que deje de ejecutarse cuando el último suscriptor se vaya.*

A continuación, un ejemplo:

Copy

```
import { interval, Subject } from "rxjs";
import { multicast, refCount } from "rxjs/operators";

const source = interval(500);
const subject = new Subject();
const refCounted = source.pipe(multicast(subject), refCount());
let subscription1, subscription2;

// Esto llama a `connect()`, ya que
// es el primer suscriptor de `refCounted`
console.log("observerA suscrito");
```

```
subscription1 = refCounted.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});

setTimeout(() => {
  console.log("observerB suscrito");
  subscription2 = refCounted.subscribe({
    next: (v) => console.log(`observerB: ${v}`),
  });
}, 600);

setTimeout(() => {
  console.log("Suscripción de observerA cancelada");
  subscription1.unsubscribe();
}, 1200);

// En este momento se parará la ejecución compartida del Observable, ya que
// `refCounted` no tendrá más suscriptores a partir de este momento
setTimeout(() => {
  console.log("Suscripción de observerB cancelada");
  subscription2.unsubscribe();
}, 2000);

// Logs
// observerA suscrito
// observerA: 0
// observerB suscrito
// observerA: 1
// observerB: 1
// Suscripción de observerA cancelada
// observerB: 2
// Suscripción de observerB cancelada
```

El método `refCount()` existe únicamente en `ConnectableObservable`, y no retorna otro `ConnectableObservable`, sino un `Observable`.

BehaviorSubject

Una de las variantes del Sujeto es el `BehaviorSubject`, que tiene la noción del "valor actual". Almacena el último valor emitido a sus consumidores, y cuandoquiera que se suscriba un Observador nuevo, este recibirá inmediatamente el "valor actual" del `BehaviorSubject`.

> *Los `BehaviorSubjects` son útiles para representar "valores a lo largo del tiempo". Por ejemplo, un flujo de eventos de cumpleaños es un Sujeto normal, pero el flujo de la edad de una persona sería un `BehaviorSubject`.*

En el siguiente ejemplo, el `BehaviorSubject` se inicializa con el valor `0`, que es el que recibe el primer Observador cuando se suscribe. El segundo Observador recibe el valor `2` a pesar de haberse suscrito *después* de que el valor `2` fuese emitido.

```
import { BehaviorSubject } from "rxjs";
const subject = new BehaviorSubject(0); // 0 es el valor inicial

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

subject.next(3);

// Logs
// observerA: 0
// observerA: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

Copy

ReplaySubject

El `ReplaySubject` es similar al `BehaviorSubject` en el sentido de que puede enviar valores antiguos a los suscriptores nuevos, pero a diferencia de este último,

el `ReplaySubject` también puede *almacenar* una parte de la ejecución Observable.

> *Un `ReplaySubject` almacena múltiples valores de la ejecución Observable, y los repite a los nuevos suscriptores.*

Cuando se crea un `ReplaySubject`, se puede especificar cuántos valores se repetirán:

```
import { ReplaySubject } from "rxjs";
const subject = new ReplaySubject(3); // Almacena 3 valores para los nuevos

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

subject.next(5);

// Logs:
// observerA: 1
// observerA: 2
// observerA: 3
// observerA: 4
// observerB: 2
// observerB: 3
// observerB: 4
// observerA: 5
// observerB: 5
```

Copy

También se puede especificar el parámetro `windowTime` en milisegundos, además del tamaño del buffer, para determinar cuán antiguos pueden ser los valores almacenados. En el siguiente ejemplo utilizamos un tamaño bastante mayor de buffer, `100`, pero un parámetro `windowTime` de solo `500` milisegundos.

```
import { ReplaySubject } from "rxjs";
const subject = new ReplaySubject(100, 500 /* El parámetro windowTime */);

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});

let i = 1;
setInterval(() => subject.next(i++), 200);

setTimeout(() => {
  subject.subscribe({
    next: (v) => console.log(`observerB: ${v}`),
  });
}, 1000);

// Logs
// observerA: 1
// observerA: 2
// observerA: 3
// observerA: 4
// observerA: 5
// observerB: 3
// observerB: 4
// observerB: 5
// observerA: 6
// observerB: 6
// ...
```

AsyncSubject

El `AsyncSubject` es una variante donde únicamente se envía el último valor de la ejecución Observable a sus Observadores, y esto ocurre solo cuando la ejecución se ha completado.

```
import { AsyncSubject } from "rxjs";
const subject = new AsyncSubject();

subject.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
```

```
});

subject.next(1);
subject.next(2);
subject.next(3);
subject.next(4);

subject.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});

subject.next(5);
subject.complete();

// Salida:
// observerA: 5
// observerB: 5
```

El `AsyncSubject` es similar al operador last(), ya que espera a haber recibido la notificación `complete` para enviar el valor último.