RxJS - Subject

rxjs.dev/guide/subject

Sujet

Qu'est-ce qu'un sujet ? Un sujet RxJS est un type spécial d'observable qui permet aux valeurs d'être multidiffusées à de nombreux observateurs. Alors que les Observables simples sont unicast (chaque Observateur abonné possède une exécution indépendante de l'Observable), les Sujets sont multicast.

Un sujet est comme un observable, mais peut être multidiffusé à plusieurs observateurs. Les sujets sont comme les EventEmitters : ils maintiennent un registre de nombreux auditeurs.

Chaque Sujet est un Observable. Étant donné un sujet, vous pouvez `subscribe`le faire, en fournissant un observateur, qui commencera à recevoir des valeurs normalement. Du point de vue de l'Observer, il ne peut pas dire si l'exécution d'Observable provient d'un Observable monodiffusion simple ou d'un Sujet.

En interne au sujet, `subscribe`n'invoque pas une nouvelle exécution qui délivre des valeurs. Il enregistre simplement l'observateur donné dans une liste d'observateurs, de la même manière que cela `addListener`fonctionne habituellement dans d'autres bibliothèques et langages.

Chaque sujet est un observateur. C'est un objet avec les méthodes `next(v)`, `error(e)`et `complete()`. Pour fournir une nouvelle valeur au sujet, appelez simplement `next(theValue)`, et elle sera multidiffusée aux observateurs enregistrés pour écouter le sujet.

Dans l'exemple ci-dessous, nous avons deux observateurs attachés à un sujet, et nous alimentons le sujet en valeurs :

```
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);
// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

Puisqu'un Sujet est un Observateur, cela signifie également que vous pouvez fournir un Sujet comme argument à `subscribe`n'importe quel Observable, comme le montre l'exemple ci-dessous :

```
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); // You can subscribe providing a Subject
// Logs:
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
// observerA: 3
// observerB: 3
```

With the approach above, we essentially just converted a unicast Observable execution to multicast, through the Subject. This demonstrates how Subjects are the only way of making any Observable execution be shared to multiple Observers.

There are also a few specializations of the `Subject` type: `BehaviorSubject`, `ReplaySubject`, and `AsyncSubject`.

Multicasted Observables

A "multicasted Observable" passes notifications through a Subject which may have many subscribers, whereas a plain "unicast Observable" only sends notifications to a single Observer.

A multicasted Observable uses a Subject under the hood to make multiple Observers see the same Observable execution.

Under the hood, this is how the `multicast` operator works: Observers subscribe to an underlying Subject, and the Subject subscribes to the source Observable. The following example is similar to the previous example which used `observable.subscribe(subject)`:

```
const source = from([1, 2, 3]);
const subject = new Subject();
const multicasted = source.pipe(multicast(subject));
// These are, under the hood, `subject.subscribe({...})`:
multicasted.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});
multicasted.subscribe({
  next: (v) => console.log(`observerB: ${v}`),
});
// This is, under the hood, `source.subscribe(subject)`:
multicasted.connect();
```

`multicast` returns an Observable that looks like a normal Observable, but works like a Subject when it comes to subscribing. `multicast` returns a `ConnectableObservable`, which is simply an Observable with the `connect()` method.

The `connect()` method is important to determine exactly when the shared Observable execution will start. Because `connect()` does `source.subscribe(subject)` under the hood, `connect()` returns a Subscription, which you can unsubscribe from in order to cancel the shared Observable execution.

Reference counting

Calling `connect()` manually and handling the Subscription is often cumbersome. Usually, we want to *automatically* connect when the first Observer arrives, and automatically cancel the shared execution when the last Observer unsubscribes.

Consider the following example where subscriptions occur as outlined by this list:

1. First Observer subscribes to the multicasted Observable
2. **The multicasted Observable is connected**
3. The `next` value `0` is delivered to the first Observer
4. Second Observer subscribes to the multicasted Observable
5. The `next` value `1` is delivered to the first Observer
6. The `next` value `1` is delivered to the second Observer
7. First Observer unsubscribes from the multicasted Observable
8. The `next` value `2` is delivered to the second Observer
9. Second Observer unsubscribes from the multicasted Observable
10. **The connection to the multicasted Observable is unsubscribed**

To achieve that with explicit calls to `connect()`, we write the following code:

```
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}`),
});
// We should call `connect()` here, because the first
// subscriber to `multicasted` is interested in consuming values
subscriptionConnect = multicasted.connect();
setTimeout(() => {
  subscription2 = multicasted.subscribe({
    next: (v) => console.log(`observerB: ${v}`),
  });
}, 600);
setTimeout(() => {
  subscription1.unsubscribe();
}, 1200);
// We should unsubscribe the shared Observable execution here,
// because `multicasted` would have no more subscribers after this
setTimeout(() => {
  subscription2.unsubscribe();
  subscriptionConnect.unsubscribe(); // for the shared Observable execution
}, 2000);
```

If we wish to avoid explicit calls to `connect()`, we can use ConnectableObservable's `refCount()` method (reference counting), which returns an Observable that keeps track of how many subscribers it has. When the number of subscribers increases from `0` to `1`, it will call `connect()` for us, which starts the shared execution. Only when the number of subscribers decreases from `1` to `0` will it be fully unsubscribed, stopping further execution.

`refCount` makes the multicasted Observable automatically start executing when the first subscriber arrives, and stop executing when the last subscriber leaves.

Below is an example:

```
import { interval, Subject, multicast, refCount } from 'rxjs';
const source = interval(500);
const subject = new Subject();
const refCounted = source.pipe(multicast(subject), refCount());
let subscription1, subscription2;
// This calls `connect()`, because
// it is the first subscriber to `refCounted`
console.log('observerA subscribed');
subscription1 = refCounted.subscribe({
  next: (v) => console.log(`observerA: ${v}`),
});
setTimeout(() => {
  console.log('observerB subscribed');
  subscription2 = refCounted.subscribe({
    next: (v) => console.log(`observerB: ${v}`),
  });
}, 600);
setTimeout(() => {
  console.log('observerA unsubscribed');
  subscription1.unsubscribe();
}, 1200);
// This is when the shared Observable execution will stop, because
// `refCounted` would have no more subscribers after this
setTimeout(() => {
  console.log('observerB unsubscribed');
  subscription2.unsubscribe();
}, 2000);
// Logs
// observerA subscribed
// observerA: 0
// observerB subscribed
// observerA: 1
// observerB: 1
// observerA unsubscribed
// observerB: 2
// observerB unsubscribed
```

The `refCount()` method only exists on ConnectableObservable, and it returns an `Observable`, not another ConnectableObservable.

BehaviorSubject

One of the variants of Subjects is the `BehaviorSubject`, which has a notion of "the current value". It stores the latest value emitted to its consumers, and whenever a new Observer subscribes, it will immediately receive the "current value" from the `BehaviorSubject`.

BehaviorSubjects are useful for representing "values over time". For instance, an event stream of birthdays is a Subject, but the stream of a person's age would be a BehaviorSubject.

In the following example, the BehaviorSubject is initialized with the value `0` which the first Observer receives when it subscribes. The second Observer receives the value `2` even though it subscribed after the value `2` was sent.

```
import { BehaviorSubject } from 'rxjs';
const subject = new BehaviorSubject(0); // 0 is the initial value
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
```

ReplaySubject

A `ReplaySubject` is similar to a `BehaviorSubject` in that it can send old values to new subscribers, but it can also *record* a part of the Observable execution.

A `ReplaySubject` records multiple values from the Observable execution and replays them to new subscribers.

When creating a `ReplaySubject`, you can specify how many values to replay:

```
import { ReplaySubject } from 'rxjs';
const subject = new ReplaySubject(3); // buffer 3 values for new subscribers
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
```

You can also specify a *window time* in milliseconds, besides of the buffer size, to determine how old the recorded values can be. In the following example we use a large buffer size of `100`, but a window time parameter of just `500` milliseconds.

```
import { ReplaySubject } from 'rxjs';
const subject = new ReplaySubject(100, 500 /* 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

The AsyncSubject is a variant where only the last value of the Observable execution is sent to its observers, and only when the execution completes.

```
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();
// Logs:
// observerA: 5
// observerB: 5
```

The AsyncSubject is similar to the `last()` operator, in that it waits for the `complete` notification in order to deliver a single value.

Void subject

Sometimes the emitted value doesn't matter as much as the fact that a value was emitted.

For instance, the code below signals that one second has passed.

```
      const subject = new Subject<string>();
setTimeout(() => subject.next('dummy'), 1000);
```

Passing a dummy value this way is clumsy and can confuse users.

En déclarant un *void subject* , vous signalez que la valeur n'est pas pertinente. Seul l'événement lui-même compte.

```
      const subject = new Subject<void>();
setTimeout(() => subject.next(), 1000);
```

Un exemple complet avec contexte est présenté ci-dessous :

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

const subject = new Subject(); // Shorthand for Subject<void>

subject.subscribe({
  next: () => console.log('One second has passed'),
});

setTimeout(() => subject.next(), 1000);
```