RxJS: Understanding the publish and share Operators

Nicholas Jamieson Follow
Aug 23, 2017 · 10 min read

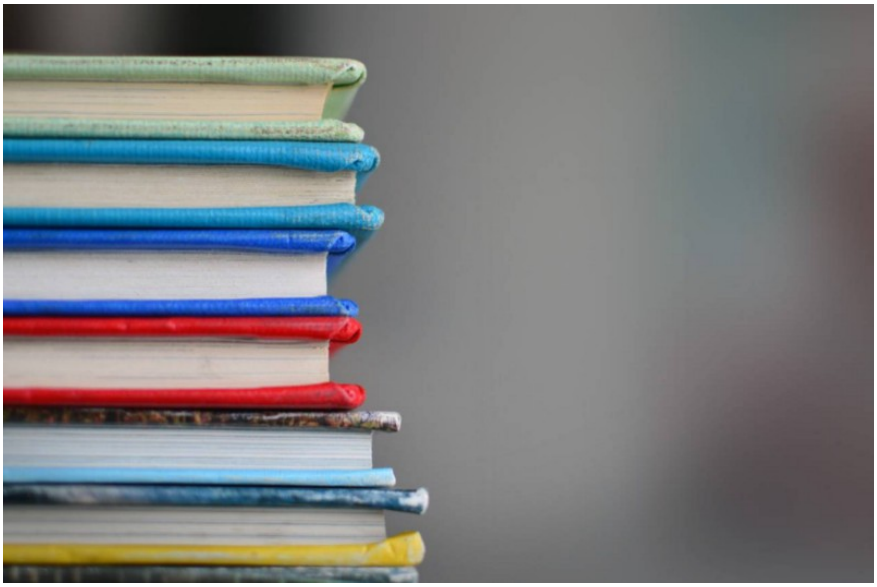

Photo by Kimberly Farmer on Unsplash

I'm often asked questions that relate to the `publish` operator:

> *What's the difference between publish and share?*

> *How do I import the refCount operator?*

> *When should I use an AsyncSubject?*

Let's answer these questions—and more—by starting with the basics.

A mental model for multicasting

Multicasting is the term used to describe the situation in which each notification emitted by a single observable is received by multiple observers. Whether or not an observable is capable of multicasting depends upon whether that observable is hot or cold.

Hot and cold observables are characterised by where the producer of the observable's notifications is created. In his article *Hot vs. Cold Observables,* Ben Lesh discusses the differences in detail, and the differences can be summarised as follows:

- An observable is cold if the producer of its notifications is created whenever an observer subscribes to the observable. For example, a `timer` observable is cold; each time a subscription is made, a new timer is created.
- An observable is hot if the producer of its notifications is not created each time an observer subscribes to the observable. For example, an observable created using `fromEvent` is hot; the element that produces the events exists in the DOM—it's not created when the observer is subscribed.

Cold observables are unicast, as each observer receives notifications from the producer that was created when the observer subscribed.

Hot observables are multicast, as each observer receives notifications from the same producer.

Sometimes, it's desirable to have multicast behaviour with a source observable that is cold, and RxJS includes a class that makes this possible: the `Subject`.

A subject is both an observable and an observer. By subscribing observers to a subject and then subscribing the subject to a cold observable, a cold observable can be made hot. RxJS includes subjects primarily for this purpose; in his *On the Subject of Subjects* article, Ben Lesh states that:

> *[multicasting] is the primary use case for Subjects in RxJS.*

Let's look at an example:

```
import { Observable } from "rxjs/Observable";
import { Subject } from "rxjs/Subject";
import "rxjs/add/observable/defer";
import "rxjs/add/observable/of";

const source = Observable.defer(() => Observable.of(
  Math.floor(Math.random() * 100)
));

function observer(name: string) {
  return {
    next: (value: number) => console.log(`observer ${n
    complete: () => console.log(`observer ${name}: com
  };
```

The source in the example is cold. Each time an observer subscribes to the source, the factory function passed to `defer` will create an observable that emits a random number and then completes.

To multicast the source, the observers are subscribed to the subject, and the subject is then subscribed to the source. The source will see only one subscription, will produce only one `next` notification—containing the random number—and will produce only one `complete` notification. The subject will send those notifications to its observers and the output will look something like this:

```
observer a: 42
observer b: 42
observer a: complete
observer b: complete
```

The example can be used as a basic mental model for RxJS multicasting: a source observable; a subject subscribed to the source; and multiple observers subscribed to the subject.

The multicast operator and connectable observables

RxJS includes a `multicast` operator that can be applied to an observable to make it hot. The operator encapsulates the infrastructure that's involved when a subject is used to multicast an observable.

Before looking at the `multicast` operator, let's replace the subject in the above example with a naive implementation of a `multicast` function:

```
function multicast<T>(source: Observable<T>) {
  const subject = new Subject<T>();
  source.subscribe(subject);
  return subject;
}

const m = multicast(source);
```

With this change, the example's output is this:

```
observer a: complete
observer b: complete
```

Which isn't what's wanted. Subscribing the subject inside the function sees the subject receive the `next` and `complete` notifications before the observers are subscribed—so the observers receive only a `complete` notification.

For this to be avoidable, the caller of any function that connects the multicasting infrastructure needs to be able to control when the subject subscribes to the source. RxJS's `multicast` operator enables this by returning a special type of observable: a `ConnectableObservable`.

A connectable observable encapsulates the multicasting infrastructure, but does not immediately subscribe to the source. It subscribes to the source when its `connect` method is called.

Let's change the example to use the `multicast` operator:

```
import { Observable } from "rxjs/Observable";
import { Subject } from "rxjs/Subject";
import "rxjs/add/observable/defer";
import "rxjs/add/observable/of";
import "rxjs/add/operator/multicast";

const source = Observable.defer(() => Observable.of(
  Math.floor(Math.random() * 100)
));

function observer(name: string) {
  return {
    next: (value: number) => console.log(`observer ${n
    complete: () => console.log(`observer ${name}: com
```

With this change, the `next` notifications are now received by the observers:

```
observer a: 54
observer b: 54
observer a: complete
observer b: complete
```

When `connect` is called, the subject passed to the `multicast` operator is subscribed to the source and the subject's observers receive the multicast notifications—which fits our basic mental model of RxJS multicasting.

Connectable observables have another method that can be used to determine when subscriptions to the source are made: the `refCount` method.

`refCount` looks like an operator—that is, it's a method that's called on an observable that returns another observable—but it's a `ConnectableObservable` method and does not need to be imported. As its name suggests, `refCount` returns an observable that maintains a reference count of the subscriptions that have been made.

When an observer is subscribed to the reference-counted observable, the reference count is incremented and if the prior reference count was zero, the multicasting infrastructure's subject is subscribed to the source observable. And when an observer is unsubscribed, the

reference count is decremented and if the reference count drops to zero, the subject is unsubscribed from the source.

Let's change the example to use `refCount`:

```
const m = source.multicast(new Subject<number>()).refCo
m.subscribe(observer("a"));
m.subscribe(observer("b"));
```

With this change, the output is something like this:

```
observer a: 42
observer a: complete
observer b: complete
```

Only the first observer receives a `next` notification. Let's look at why.

The source observable in the example emits its notifications immediately. That is, as soon as a subscription is made, the source emits a `next` notification and then a `complete` notification and the `complete` notification results in the first observer unsubscribing before the second has subscribed. When the first unsubscribes, the reference count drops to zero and the multicasting infrastructure's subject is unsubscribed from the source.

When the second observer subscribes, the subject is again subscribed to the source, but the subject has already received a `complete` notification and subjects cannot be reused.

Passing a subject factory function to `multicast` will solve the problem:

```
const m = source.multicast(() => new Subject<number>())
m.subscribe(observer("a"));
m.subscribe(observer("b"));
```

With this change, a subject is created each time a subscription is made to the source observable, and the output is something like this:

```
observer a: 42
observer a: complete
observer b: 54
observer b: complete
```

Because the source observable emits its notifications immediately, the observers receive separate notifications. If the source is modified so that the notifications are delayed:

```
1  import { Observable } from "rxjs/Observable";
2  import { Subject } from "rxjs/Subject";
3  import "rxjs/add/observable/defer";
4  import "rxjs/add/observable/of";
5  import "rxjs/add/operator/delay";
6  import "rxjs/add/operator/multicast";
7
```

The observers will receive multicast notifications and the output will look something like this:

```
observer a: 42
observer b: 42
observer a: complete
observer b: complete
```

To summarise, the examples have shown that the `multicast` operator:

- encapsulates multicasting that fits our mental model;
- provides a `connect` method that can be used to determine when the subscription to the source is made;
- provides a `refCount` method that can be used to automatically manage subscriptions to the source observable; and
- if `refCount` is used, a subject factory function must be specified —instead of a `Subject` instance.

Let's now look at the `publish` and `share` operators—and their variants—to see how they build upon what the `multicast` operator

provides.

The publish operator

Let's use the following example to look at the `publish` operator:

```
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/defer";
import "rxjs/add/observable/of";
import "rxjs/add/operator/delay";
import "rxjs/add/operator/publish";

function random() {
  return Math.floor(Math.random() * 100);
}

const source = Observable.concat(
  Observable.defer(() => Observable.of(random())),
  Observable.defer(() => Observable.of(random())).dela
);

function observer(name: string) {
  return {
    next: (value: number) => console.log(`observer ${n
```

The example's source observable immediately emits a random number, then after a short delay emits another random number and completes. The example will allow us to look at what happens with observers that subscribe before `connect` is called, after `connect` is called and after the published observable completes.

The `publish` operator is a thin wrapper around the `multicast` operator. It calls `multicast`, passing a `Subject`.

The example's output will be something like this:

```
observer a: 42
observer a: 54
observer b: 54
observer a: complete
observer b: complete
observer c: complete
```

The notifications received by the observables can be summarised as follows:

- `a` subscribes before the connect call, so it receives both of the `next` notifications and the `complete` notification;
- `b` subscribes after the connect call, by which time the first, immediate `next` notification has already been emitted, so it receives only the second `next` notification and the `complete` notification;
- `c` subscribes after the source observable has completed, so it receives only a `complete` notification.

If the example is changed to use `refCount` instead of `connect`:

```
const p = source.publish().refCount();
p.subscribe(observer("a"));
p.subscribe(observer("b"));
setTimeout(() => p.subscribe(observer("c")), 10);
```

The example's output will be something like this:

```
observer a: 42
observer a: 54
observer b: 54
observer a: complete
observer b: complete
observer c: complete
```

The output will be similar to that received when `connect` was used. Why?

Observable `b` does not receive the first next notification because the source's first `next` notification is immediate, so that notification is received only by `a`.

Observable `c` subscribes after the published observable completes, so the subscription reference count will have dropped to zero and another subscription to the source will be made. However, `publish` passes a subject to `multicast` —not a factory function—and subjects cannot be reused, so observable `c` receives only a `complete` notification.

The `publish` operator—and the `multicast` operator, too—takes an optional `selector` function and the operator's behaviour differs significantly if the function is specified. This is covered in more detail in a separate article: *multicast's Secret*.

Specialised subjects

The `publish` operator has several variants and they all wrap `multicast` in a similar manner, passing subjects rather than factory functions. However, they pass different kinds of subjects.

The specialised subjects that the `publish` variants use include:

- the `BehaviorSubject`;
- the `ReplaySubject`; and
- the `AsyncSubject`.

The answer to questions regarding where—or how—these specialised subjects should be used is: whenever you require behaviour similar to that of the `publish` variant that's associated with the specialised subject. Let's look at how the variants behave.

The publishBehavior operator

Instead of passing a `Subject` to `multicast`, `publishBehavior` passes a `BehaviorSubject`. A `BehaviorSubject` is similar to a `Subject`, but if a subscription is made to the subject before the source observable emits a `next` notification, the subject emits a `next` notification containing its initial value.

Let's change the example to briefly delay the random-number-generating source so that it does not immediately emit a random number:

```
const delayed = Observable.timer(1).switchMapTo(source)
const p = delayed.publishBehavior(-1);
p.subscribe(observer("a"));
p.connect();
p.subscribe(observer("b"));
```

The example's output will be something like this:

```
observer a: -1
observer b: -1
observer a: 42
observer b: 42
observer a: 54
observer b: 54
observer a: complete
observer b: complete
observer c: complete
```

The notifications received by the observables can be summarised as follows:

- `a` subscribes before the `connect` call, so it receives a `next` notification with subject's initial value, both of the `next` notifications from the source and the `complete` notification;
- `b` subscribes after the connect call, but before the subject receives the source's first `next` notification, so it receives a `next` notification with subject's initial value, both of the `next` notifications from the source and the `complete` notification;
- `c` subscribes after the source observable has completed, so it receives only a `complete` notification.

The publishReplay operator

Instead of passing a `Subject` to `multicast`, `publishReplay` passes a `ReplaySubject`. As its name suggests, a `ReplaySubject` will replay the specified number of `next` notifications whenever an observer subscribes.

```
1  const p = source.publishReplay(1);
2  p.subscribe(observer("a"));
3  p.connect();
4  p.subscribe(observer("b"));
```

Using `publishReplay`, the example's output will be something like this:

```
observer a: 42
observer b: 42
observer a: 54
observer b: 54
observer a: complete
```

```
observer b: complete
observer c: 54
observer c: complete
```

The notifications received by the observables can be summarised as follows:

- `a` subscribes before the `connect` call, at which stage the subject has received no `next` notifications, so `a` receives both of the `next` notifications from the source and the `complete` notification;
- `b` subscribes after the `connect` call, at which stage the subject has received the first `next` notification from the source, so `b` receives the replayed `next` notification, the source's second `next` notification and the `complete` notification;
- `c` subscribes after the source observable has completed, so it receives a replayed `next` notification and a `complete` notification.

Looking at the behaviour of observable `c`, it's clear that—unlike the `publish` operator—the `publishReplay` operator is suited for use with the `refCount` method, as observers subscribing after the source completes will receive the any replayed `next` notifications.

The publishLast operator

Instead of passing a `Subject` to `multicast`, `publishLast` passes an `AsyncSubject`. The `AsyncSubject` is the most unusual of the specialised subjects. It does not emit a `next` notification until it completes, at which time it emits the last `next` notification it received from the source observable—if it has received one—and a `complete` notification.

```
1  const p = source.publishLast();
2  p.subscribe(observer("a"));
3  p.connect();
4  p.subscribe(observer("b"));
```

Using `publishLast`, the example's output will be something like this:

```
observer a: 54
observer b: 54
observer a: complete
observer b: complete
observer c: 54
observer c: complete
```

The notifications received by the observables can be summarised as follows:

- `a` and `b` subscribe before the source completes, but receive no notifications until the source has completed, at which time they receive a `next` notification containing the second random number and a `complete` notification.
- `c` subscribes after the source has completed and it, too, receives a `next` notification containing the second random number and a `complete` notification.

Like `publishReplay`, the `publishLast` operator is suited for use with the `refCount` method, as observers subscribing after the source completes will receive the last `next` notification.

The share operator

The `share` operator is similar to using the `publish` operator and calling `refCount`. However, `share` passes a factory function to `multicast`, which means that when a subscription is made after the reference count drops to zero, a new `Subject` will be created and subscribed to the source observable.

The use of a factory function makes observables composed using `share` retry-able: if an error occurs, any subscribers are unsubscribed and the reference count drops to zero. If retried, re-subscription will see a new `Subject` created which will subscribe to the source. With observables composed using `publish`, this will not happen: the `Subject` will simply re-emit the error notification.

```
1  const s = source.share();
2  s.subscribe(observer("a"));
3  s.subscribe(observer("b"));
4  setTimeout(() => s.subscribe(observer("c")), 10);
```

Using `share`, the example's output will be something like this:

```
observer a: 42
observer a: 54
observer b: 54
observer a: complete
observer b: complete
observer c: 6
observer c: 9
observer c: complete
```

The notifications received by the observables can be summarised as follows:

- `a` subscribes and immediately the first `next` notification, followed by the second `next` notification and the `complete` notification;
- `b` receives only the second `next` notification and the `complete` notification;
- `c` subscribes after the source observable has completed; a new subject is created and subscribed to the source, from which it immediately receives the first `next` notification, followed by the second `next` notification and the `complete` notification.

In the examples we've used to look at the `publish` and `share` operators, observers `a` and `b` are unsubscribed automatically when the source observable completes. They would also be unsubscribed automatically if the source were to error. That highlights another difference between the `publish` and `share` operators:

- if the source observable errors, any future subscriber to the observable returned by `publish` will receive the error;
- however, any future subscriber to the observable returned by `share` will effect a new subscription to the source, as the error will have automatically unsubscribed any subscribers, dropping the reference count to zero.

And that's it; we're at the end. We've looked at six operators, but they are all implemented in a similar manner and they all fit the same basic mental model: a source observable; a subject subscribed to the source; and multiple observers subscribed to the subject.

• • •

This article looked briefly at the `refCount` method. For a more in-depth look, see *RxJS: How to Use refCount*.