RxJS: How to Use refCount

Nicholas Jamieson Follow
Sep 3, 2017 · 7 min read

Photo by Mike Wilson on Unsplash

My previous article—*Understanding the publish and share Operators*—looked only briefly at the `refCount` method. Let's look at it more closely here.

What does refCount do?

To recap, the basic mental model for multicasting in RxJS involves: a source observable; a subject subscribed to the source; and multiple observers subscribed to the subject. The `multicast` operator encapsulates the subject-based infrastructure and returns a `ConnectableObservable` —upon which either the `connect` or `refCount` method can be called.

As its name suggests, `refCount` returns an observable that maintains a reference count of subscribers.

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.

This reference counting behaviour can be used in two ways:

- to automate the unsubscription of the subject from the source observable—when all observers have unsubscribed; or
- to automate both the unsubscription of the subject from the source—when all observers have unsubscribed—and the re-subscription of the subject to the source—when further observers subscribe to the reference counted observable.

Let's look at each of these in detail and then establish some general guidelines for the use of `refCount`.

Automating unsubscription with refCount

The `publish` operator returns a connectable observable. Calling `connect` on the returned observable subscribes the multicasting infrastructure's subject to the source observable and returns a subscription. The subject will remain subscribed to the source until `unsubscribe` is called on the subscription.

Let's look at an example in which the observers take a single value—and then unsubscribe (implicitly) from the published observable:

```
1  const source = instrument(Observable.interval(100));
2  const published = source.publish();
3  const a = published.take(1).subscribe(observer("a"));
4  const b = published.take(1).subscribe(observer("b"));
```

The examples in this article use the following utility functions to instrument the source observables with logging and to create named observers:

```
function instrument<T>(source: Observable<T>) {
  return new Observable<T>(observer => {
    console.log("source: subscribing");
    const subscription = source
      .do(value => console.log(`source: ${value}`))
      .subscribe(observer);
    return () => {
      subscription.unsubscribe();
      console.log("source: unsubscribed");
    };
  });
}

```

The example's output will be:

```
source: subscribing
source: 0
observer a: 0
observer a: complete
observer b: 0
observer b: complete
source: 1
source: 2
source: 3
...
```

The two observers each take a single value and then complete—unsubscribing from the published observable. However, the multicasting infrastructure remains subscribed to the source.

Instead of having to decide when to perform an explicit unsubscription, `refCount` could be used:

```
const source = instrument(Observable.interval(100));
const counted = source.publish().refCount();
const a = counted.take(1).subscribe(observer("a"));
const b = counted.take(1).subscribe(observer("b"));
```

With the observers subscribed to the reference counted observable, the multicasting infrastructure's subject will be unsubscribed from the source observable when the reference count drops to zero and the example's output will be:

```
source: subscribing
source: 0
observer a: 0
observer a: complete
observer b: 0
observer b: complete
source: unsubscribed
```

Re-subscribing to observables that complete

In addition to unsubscribing from the source observable when the reference count drops to zero, the multicasting infrastructure will re-subscribe to the source when further subscriptions are made to the reference counted observable.

Let's look at what this means when used with a source observable that completes, using this example:

```
1  const source = instrument(Observable.timer(100));
2  const counted = source.publish().refCount();
3  const a = counted.subscribe(observer("a"));
4  setTimeout(() => a.unsubscribe(), 110);
```

In the example, a `timer` observable is used as the source. It will wait the specified number of milliseconds and will then emit a `next` notification and a `complete` notification. There are two observers: `a`, which subscribes before the source completes and unsubscribes after the source completes; and `b`, which subscribes after `a` has unsubscribed.

The example's output will be:

```
source: subscribing
source: 0
observer a: 0
source: unsubscribed
observer a: complete
observer b: complete
```

When `b` subscribes, the reference count will be zero, so the multicasting infrastructure looks to re-subscribe the subject to the source. However, the subject will have received a `complete`

notification from the source and completed subjects are not reusable, so there is no re-subscription and `b` receives only the `complete` notification.

If `publish()` is replaced with `publishBehavior(-1)`, the output is similar, but includes the `BehaviorSubject`'s initial value:

```
observer a: -1
source: subscribing
source: 0
observer a: 0
source: unsubscribed
observer a: complete
observer b: complete
```

Again, `b` receives only the complete notification.

If `publish()` is instead replaced with `publishReplay(1)`, the situation is a little different and the output will be:

```
source: subscribing
source: 0
observer a: 0
source: unsubscribed
observer a: complete
observer b: 0
observer b: complete
```

Again, there is no re-subscription to the source, as the subject has completed. However, a completed `ReplaySubject` replays its notifications to late subscribers, so `b` receives the replayed `next` notification and the `complete` notification.

If `publish()` is instead replaced with `publishLast()`, the situation is a little different and the output will be:

```
source: subscribing
source: 0
source: unsubscribed
observer a: 0
observer a: complete
observer b: 0
observer b: complete
```

Again, there is no re-subscription to the source, as the subject has completed. However, the `AsyncSubject` used by the multicasting infrastructure emits the last-received `next` notification to its subscribers, so, like `a`, `b` receives the `next` notification and the `complete` notification.

To summarise, from examples we can see that with `publish` and its variants:

- when a source observable completes, the multicasting infrastructure's subject will complete, too, and this will prevent re-subscription to the source;
- when using `refCount` with `publish` and `publishBehavior`, late subscribers will receive only a `complete` notification—which is unlikely to be what's wanted;
- when using `refCount` with `publishReplay` and `publishLast`, late subscribers will receive the expected notifications.

Re-subscribing to observables that do not complete

We've seen what happens when re-subscribing to a source observable that completes, now let's look at re-subscribing to a source that does not complete.

Instead, of a `timer` observable, this example uses an `interval` observable—which will repeatedly emit `next` notifications containing a incremented number at the specified interval:

```
const source = instrument(Observable.interval(100));
const counted = source.publish().refCount();
const a = counted.subscribe(observer("a"));
setTimeout(() => a.unsubscribe(), 110);
```

The example's output will be:

```
source: subscribing
source: 0
observer a: 0
source: unsubscribed
source: subscribing
source: 0
observer b: 0
```

```
source: 1
observer b: 1
...
```

Unlike the examples with source observables that complete, the multicasting infrastructure's subject is able to be re-subscribed, so a new subscription is made to the source. Evidence of the re-subscription can be seen in the `next` notification received by `b`: the notification contains a value of zero, as the re-subscription has seen a new series of intervals started.

If `publish()` is instead replaced with `publishBehavior(-1)`, the situation is a little different and the output will be:

```
observer a: -1
source: subscribing
source: 0
observer a: 0
source: unsubscribed
observer b: 0
source: subscribing
source: 0
observer b: 0
source: 1
observer b: 1
...
```

The output is similar and it's clear that the re-subscription sees a new interval started. However, before it receives the `next` notification from the interval, `a` also receives a `next` notification containing the `BehaviorSubject`'s initial value of `-1` and, before it receives the `next` notification from the interval, `b` receives a `next` notification containing the `BehaviorSubject`'s current value of `0`.

If `publish()` is instead replaced with `publishReplay(1)`, the situation is a little different and the output will be:

```
source: subscribing
source: 0
observer a: 0
source: unsubscribed
observer b: 0
source: subscribing
source: 0
observer b: 0
source: 1
```

```
observer b: 1
...
```

The output is similar and it's clear that the re-subscription sees a new interval started. However, `b` receives a replayed `next` notification before it receives the first `next` notification from the source.

To summarise, from the examples we can see that when using `refCount` with sources that do not complete, `publish`, `publishBehavior` and `publishReplay` behave as expected. There are no surprises here.

What does shareReplay do?

In RxJS version 5.4.0, the `shareReplay` operator was introduced. It is very similar to `publishReplay().refCount()`, but has one subtle difference.

Like `share`, `shareReplay` passes the `multicast` operator a subject factory. That means that when re-subscribing to the source observable, the factory will be used to create a new subject. However, the factory only returns a new subject if the previously subscribed subject did not complete.

`publishReplay` passes a `ReplaySubject` instance to the `multicast` operator—not a factory—and it's this that effects the differing behaviour.

Re-subscriptions to a `publishReplay().refCount()` observable will see the subject always replay its replay-able notifications. However, re-subscriptions to a `shareReplay()` observable might not—if the subject has not completed, a new subject will be created. So the difference is that with a `shareReplay()` observable, when the reference count drops to zero the replay-able notifications are flushed if the subject has not completed.

Some guidelines

From what we've seen in the examples, some guidelines can be inferred:

- `refCount` can be used with `publish` —or any of its variants—to automatically unsubscribe from source observables.

- When using `refCount` to automate re-subscriptions to source observables that complete, `publishReplay` and `publishLast` behave as you'd expect. However, `publish` and `publishBehavior` don't behave in a useful manner for late subscriptions—so you should use `refCount` with `publish` and `publishBehavior` only to automate unsubscriptions.
- When using `refCount` to automate re-subscriptions to source observables that do not complete, `publish`, `publishBehavior` and `publishReplay` behave as you'd expect.
- The behaviour of `shareReplay()` is similar to that of `publishReplay().refCount()` and which you choose to use depends upon whether or not you want replay-able notifications to be flushed on re-subscription to the source observable.

• • •

The behaviour of `shareReplay` described above applies to versions of RxJS prior to 5.5. In the version 5.5.0 beta, `shareReplay` was changed: when the reference count drops to zero, the operator no longer unsubscribes from the source observable.

This change effectively makes the reference count redundant, as the subscription to the source will only be unsubscribed if the source completes or errors. The change means that `shareReplay` differs from `publishReplay().refCount()` only in its handling of errors:

- if the source errors, any future subscribers to the observable returned by `publishReplay().refCount()` will receive the error;
- however, any future subscriber to the observable returned by `share` will effect a new subscription to the source.

• • •

The behaviour of `shareReplay` was changed again in version 6.4.0. For an explanation of the change, see this article.