RxJS: Avoiding takeUntil Leaks

Nicholas Jamieson Follow
May 27, 2018 · 3 min read

Photo by Tim Gouw on Unsplash

Using the `takeUntil` operator to automatically unsubscribe from an observable is a mechanism that's explained in Ben Lesh's *Don't Unsubscribe* article.

It's also the basis of a generally-accepted pattern for unsubscribing upon an Angular component's destruction.

For the mechanism to be effective, operators have to be applied in a specific sequence. And, recently, I've seen code that used the `takeUntil` operator, but applied operators in a sequence that effected subscription leaks.

Let's look at the problem sequence and at the reason for the leaks.

What's the problem sequence?

If the `takeUntil` operator is placed before an operator that involves a subscription to another observable source, the subscription to that source might not be unsubscribed when `takeUntil` receives its notification.

For example, this use of `combineLatest` will leak the subscription to `b`:

```typescript
import { Observable } from "rxjs";
import { combineLatest, takeUntil } from "rxjs/operato

declare const a: Observable<number>;
declare const b: Observable<number>;
declare const notifier: Observable<any>;

const c = a.pipe(
```

And this use of `switchMap` will also leak the subscription to `b`:

```typescript
import { Observable } from "rxjs";
import { switchMap, takeUntil } from "rxjs/operators";

declare const a: Observable<number>;
declare const b: Observable<number>;
declare const notifier: Observable<any>;

const c = a.pipe(
```

Why does it leak?

The reason for the leak is more apparent when the deprecated `combineLatest` operator is replaced with the static `combineLatest` factory function, like this:

```typescript
import { combineLatest, Observable } from "rxjs";
import { takeUntil } from "rxjs/operators";

declare const a: Observable<number>;
declare const b: Observable<number>;
declare const notifier: Observable<any>;

const c = a.pipe(
```

When the `notifier` emits, the observable returned by the `takeUntil` operator completes, automatically unsubscribing any subscribers.

However, the subscriber to `c` is not subscribed to the observable returned by `takeUntil` —it's subscribed to the observable returned by `combineLatest` —so it's not automatically unsubscribed upon the `takeUntil` observable's completion.

The subscriber to `c` will remain subscribed until all of the observables passed to `combinedLast` complete. So, unless `b` completed before the `notifier` emitted, the subscription to `b` would leak.

To avoid the problem, the general rule is that `takeUntil` should be the last operator in the sequence:

```
import { combineLatest, Observable } from "rxjs";
import { takeUntil } from "rxjs/operators";

declare const a: Observable<number>;
declare const b: Observable<number>;
declare const notifier: Observable<any>;

const c = a.pipe(
```

Arranged this way, when the `notifier` emits, the subscriber to `c` will be automatically unsubscribed—as the observable returned by `takeUntil` will complete—and the `takeUntil` implementation will unsubscribe from the observable returned by `combineLatest` which, in turn, will unsubscribe from both `a` and `b` .

Using TSLint to avoid the problem

If you're using the `takeUntil` mechanism to effect indirect unsubscription, you can ensure that it's always the last operator passed to pipe by enabling the `rxjs-no-unsafe-takeuntil` rule that I've added to `rxjs-tslint-rules` package.

. . .

An update

The general rule is for `takeUntil` to be placed last. However, there are some situations in which you might want want use it as the second-last operator.

RxJS includes several operators that emit a value when the source observable to which they are applied completes. For example, when their sources complete, `count` emits a count of the values emitted by the source and `toArray` emits an accumulated array of values.

When an observable completes due to `takeUntil`, operators like `count` and `toArray` will only emit a value if they are placed after the `takeUntil` operator.

Additionally, there is another operator that you should place after `takeUntil`: the `shareReplay` operator.

The current implementation of `shareReplay` has a bug/feature: it will never unsubscribe from its source. It will remain subscribed until its source errors or completes—for more information, see this PR—so placing `takeUntil` after `shareReplay` will be ineffectual.

The TSLint rule mentioned above is aware of these exceptions to the general rule and won't effect unwarranted failures.

· · ·

In RxJS version 6.4.0, `shareReplay` was changed so that its reference counting behaviour can be specified via a `config` parameter. If reference counting is specified, `shareReplay` can be safely placed before `takeUntil`.

For more information, see this article.