RxJS - RxJS Operators

rxjs.dev/guide/operators

Opérateurs RxJS

RxJS est surtout utile pour ses *opérateurs* , même si l'Observable en est la base. Les opérateurs sont les éléments essentiels qui permettent de composer facilement du code asynchrone complexe de manière déclarative.

Que sont les opérateurs ?

Les opérateurs sont **des fonctions** . Il existe deux types d'opérateurs :

Les opérateurs pipeables sont du genre qui peuvent être redirigés vers Observables en utilisant la syntaxe `observableInstance.pipe(operator)`ou, plus communément, `observableInstance.pipe(operatorFactory())`. Les fonctions d'usine de l'opérateur incluent, `filter(...)`et `mergeMap(...)`.

Lorsque les opérateurs Pipeable sont appelés, ils ne *modifient* pas l'instance Observable existante. Au lieu de cela, ils renvoient un *nouvel* Observable, dont la logique d'abonnement est basée sur le premier Observable.

Un Pipeable Operator est une fonction qui prend un Observable comme entrée et renvoie un autre Observable. C'est une opération pure : l'Observable précédent reste inchangé.

Une usine d'opérateurs pipeables est une fonction qui peut prendre des paramètres pour définir le contexte et renvoyer un opérateur pipeable. Les arguments de la fabrique appartiennent à la portée lexicale de l'opérateur.

Un Pipeable Operator est essentiellement une fonction pure qui prend un Observable en entrée et génère un autre Observable en sortie. S'abonner à la sortie Observable s'abonnera également à l'entrée Observable.

Les opérateurs de création sont l'autre type d'opérateur, qui peut être appelé en tant que fonctions autonomes pour créer un nouvel Observable. Par exemple : `of(1, 2, 3)`crée un observable qui émettra 1, 2 et 3, l'un après l'autre. Les opérateurs de création seront discutés plus en détail dans une section ultérieure.

Par exemple, l'opérateur appelé `map`est analogue à la méthode Array du même nom. Tout comme `[1, 2, 3].map(x => x * x)`donnera `[1, 4, 9]`, l'Observable créé comme ceci :

émettra `1`, `4`, `9`. Un autre opérateur utile est `first`:

Note that `map` logically must be constructed on the fly, since it must be given the mapping function to. By contrast, `first` could be a constant, but is nonetheless constructed on the fly. As a general practice, all operators are constructed, whether they need arguments or not.

Piping

Pipeable operators are functions, so they *could* be used like ordinary functions: `op()(obs)` — but in practice, there tend to be many of them convolved together, and quickly become unreadable: `op4()(op3()(op2()(op1()(obs))))`. For that reason, Observables have a method called `.pipe()` that accomplishes the same thing while being much easier to read:

```
obs.pipe(op1(), op2(), op3(), op4());
```

As a stylistic matter, `op()(obs)` is never used, even if there is only one operator; `obs.pipe(op())` is universally preferred.

Creation Operators

What are creation operators? Distinct from pipeable operators, creation operators are functions that can be used to create an Observable with some common predefined behavior or by joining other Observables.

A typical example of a creation operator would be the `interval` function. It takes a number (not an Observable) as input argument, and produces an Observable as output:

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

const observable = interval(1000 /* number of milliseconds */);
```

See the list of all static creation operators here.

Higher-order Observables

Observables most commonly emit ordinary values like strings and numbers, but surprisingly often, it is necessary to handle Observables *of* Observables, so-called higher-order Observables. For example, imagine you had an Observable emitting strings that were the URLs of files you wanted to see. The code might look like this:

```
const fileObservable = urlObservable.pipe(map((url) => http.get(url)));
```

`http.get()` returns an Observable (of string or string arrays probably) for each individual URL. Now you have an Observable *of* Observables, a higher-order Observable.

But how do you work with a higher-order Observable? Typically, by *flattening*: by (somehow) converting a higher-order Observable into an ordinary Observable. For example:

```
    const fileObservable = urlObservable.pipe(
  map((url) => http.get(url)),
  concatAll()
);
```

The `concatAll()` operator subscribes to each "inner" Observable that comes out of the "outer" Observable, and copies all the emitted values until that Observable completes, and goes on to the next one. All of the values are in that way concatenated. Other useful flattening operators (called *join operators*) are

- `mergeAll()` — subscribes to each inner Observable as it arrives, then emits each value as it arrives
- `switchAll()` — subscribes to the first inner Observable when it arrives, and emits each value as it arrives, but when the next inner Observable arrives, unsubscribes to the previous one, and subscribes to the new one.
- `exhaustAll()` — subscribes to the first inner Observable when it arrives, and emits each value as it arrives, discarding all newly arriving inner Observables until that first one completes, then waits for the next inner Observable.

Just as many array libraries combine `map()` and `flat()` (or `flatten()`) into a single `flatMap()`, there are mapping equivalents of all the RxJS flattening operators `concatMap()`, `mergeMap()`, `switchMap()`, and `exhaustMap()`.

Marble diagrams

To explain how operators work, textual descriptions are often not enough. Many operators are related to time, they may for instance delay, sample, throttle, or debounce value emissions in different ways. Diagrams are often a better tool for that. *Marble Diagrams* are visual representations of how operators work, and include the input Observable(s), the operator and its parameters, and the output Observable.

In a marble diagram, time flows to the right, and the diagram describes how values ("marbles") are emitted on the Observable execution.

Below you can see the anatomy of a marble diagram.

Throughout this documentation site, we extensively use marble diagrams to explain how operators work. They may be really useful in other contexts too, like on a whiteboard or even in our unit tests (as ASCII diagrams).

Categories of operators

There are operators for different purposes, and they may be categorized as: creation, transformation, filtering, joining, multicasting, error handling, utility, etc. In the following list you will find all the operators organized in categories.

For a complete overview, see the references page.

Creation Operators

- `ajax`
- `bindCallback`
- `bindNodeCallback`
- `defer`
- `empty`
- `from`
- `fromEvent`
- `fromEventPattern`
- `generate`
- `interval`
- `of`
- `range`
- `throwError`
- `timer`
- `iif`

Join Creation Operators

These are Observable creation operators that also have join functionality -- emitting values of multiple source Observables.

- `combineLatest`
- `concat`
- `forkJoin`
- `merge`
- `partition`
- `race`
- `zip`

Transformation Operators

- `buffer`

- `bufferCount`
- `bufferTime`
- `bufferToggle`
- `bufferWhen`
- `concatMap`
- `concatMapTo`
- `exhaust`
- `exhaustMap`
- `expand`
- `groupBy`
- `map`
- `mapTo`
- `mergeMap`
- `mergeMapTo`
- `mergeScan`
- `pairwise`
- `partition`
- `pluck`
- `scan`
- `switchScan`
- `switchMap`
- `switchMapTo`
- `window`
- `windowCount`
- `windowTime`
- `windowToggle`
- `windowWhen`

Filtering Operators

- `audit`
- `auditTime`
- `debounce`
- `debounceTime`
- `distinct`
- `distinctUntilChanged`
- `distinctUntilKeyChanged`
- `elementAt`
- `filter`
- `first`
- `ignoreElements`
- `last`
- `sample`
- `sampleTime`

- `single`
- `skip`
- `skipLast`
- `skipUntil`
- `skipWhile`
- `take`
- `takeLast`
- `takeUntil`
- `takeWhile`
- `throttle`
- `throttleTime`

Join Operators

Also see the Join Creation Operators section above.

- `combineLatestAll`
- `concatAll`
- `exhaustAll`
- `mergeAll`
- `switchAll`
- `startWith`
- `withLatestFrom`

Multicasting Operators

- `multicast`
- `publish`
- `publishBehavior`
- `publishLast`
- `publishReplay`
- `share`

Error Handling Operators

- `catchError`
- `retry`
- `retryWhen`

Utility Operators

- `tap`
- `delay`
- `delayWhen`
- `dematerialize`
- `materialize`

- `observeOn`
- `subscribeOn`
- `timeInterval`
- `timestamp`
- `timeout`
- `timeoutWith`
- `toArray`

Conditional and Boolean Operators

- `defaultIfEmpty`
- `every`
- `find`
- `findIndex`
- `isEmpty`

Mathematical and Aggregate Operators

- `count`
- `max`
- `min`
- `reduce`

Creating custom operators

Use the `pipe()` function to make new operators

If there is a commonly used sequence of operators in your code, use the `pipe()` function to extract the sequence into a new operator. Even if a sequence is not that common, breaking it out into a single operator can improve readability.

For example, you could make a function that discarded odd values and doubled even values like this:

```
      import { pipe, filter, map } from 'rxjs';

function discardOddDoubleEven() {
  return pipe(
    filter((v) => !(v % 2)),
    map((v) => v + v)
  );
}
```

(The `pipe()` function is analogous to, but not the same thing as, the `.pipe()` method on an Observable.)

Creating new operators from scratch

It is more complicated, but if you have to write an operator that cannot be made from a combination of existing operators (a rare occurrence), you can write an operator from scratch using the Observable constructor, like this:

```
function delay<T>(delayInMillis: number) {
  return (observable: Observable<T>) =>
    new Observable<T>((subscriber) => {
      // this function will be called each time this
      // Observable is subscribed to.
      const allTimerIDs = new Set();
      let hasCompleted = false;
      const subscription = observable.subscribe({
        next(value) {
          // Start a timer to delay the next value
          // from being pushed.
          const timerID = setTimeout(() => {
            subscriber.next(value);
            // after we push the value, we need to clean up the timer timerID
            allTimerIDs.delete(timerID);
            // If the source has completed, and there are no more timers running,
            // we can complete the resulting observable.
            if (hasCompleted && allTimerIDs.size === 0) {
              subscriber.complete();
            }
          }, delayInMillis);
          allTimerIDs.add(timerID);
        },
        error(err) {
          // We need to make sure we're propagating our errors through.
          subscriber.error(err);
        },
        complete() {
          hasCompleted = true;
          // If we still have timers running, we don't want to complete yet.
          if (allTimerIDs.size === 0) {
            subscriber.complete();
          }
        },
      });
      // Return the finalization logic. This will be invoked when
      // the result errors, completes, or is unsubscribed.
      return () => {
        subscription.unsubscribe();
        // Clean up our timers.
        for (const timerID of allTimerIDs) {
          clearTimeout(timerID);
        }
      };
    });
}
// Try it out!
```

Note that you must

1. implémentez les trois fonctions Observer, `next()`, `error()`, et `complete()`lors de l'abonnement à l'entrée Observable.
2. implémentez une fonction de "finalisation" qui nettoie lorsque l'Observable se termine (dans ce cas en vous désabonnant et en effaçant tous les délais d'attente en attente).
3. renvoie cette fonction de finalisation à partir de la fonction transmise au constructeur Observable.

Bien sûr, ce n'est qu'un exemple ; l' `delay()`opérateur existe déjà.