RxJS - Scheduler

rxjs.dev/guide/scheduler

Planificateur

Qu'est-ce qu'un planificateur ? Un planificateur contrôle le moment où un abonnement démarre et quand les notifications sont envoyées. Il se compose de trois éléments.

- **Un Scheduler est une structure de données.** Il sait comment stocker et mettre en file d'attente les tâches en fonction de la priorité ou d'autres critères.
- **Un Scheduler est un contexte d'exécution.** Il indique où et quand la tâche est exécutée (par exemple immédiatement, ou dans un autre mécanisme de rappel tel que setTimeout ou process.nextTick, ou la trame d'animation).
- **Un planificateur a une horloge (virtuelle).** Il fournit une notion de "temps" par une méthode getter `now()`sur le planificateur. Les tâches planifiées sur un planificateur particulier respecteront uniquement l'heure indiquée par cette horloge.

Un Scheduler vous permet de définir dans quel contexte d'exécution un Observable délivrera des notifications à son Observer.

Dans l'exemple ci-dessous, nous prenons l'Observable simple habituel qui émet des valeurs `1`, `2`, `3`de manière synchrone, et utilisons l'opérateur `observeOn`pour spécifier le `async`planificateur à utiliser pour fournir ces valeurs.

```
const observable = new Observable((observer) => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
}).pipe(
  observeOn(asyncScheduler)
);
console.log('just before subscribe');
observable.subscribe({
  next(x) {
    console.log('got value ' + x);
  },
  error(err) {
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  },
});
console.log('just after subscribe');
```

Qui s'exécute avec la sortie :

```
      just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

Remarquez comment les notifications `got value...`ont été livrées après `just after subscribe`, ce qui est différent du comportement par défaut que nous avons vu jusqu'à présent. En effet, introduit un observateur proxy entre et l'observateur final. Renommez certains identifiants pour rendre cette distinction évidente dans l'exemple de code :`observeOn(asyncScheduler)new Observable`

```
import { Observable, observeOn, asyncScheduler } from 'rxjs';
const observable = new Observable((proxyObserver) => {
  proxyObserver.next(1);
  proxyObserver.next(2);
  proxyObserver.next(3);
  proxyObserver.complete();
}).pipe(
  observeOn(asyncScheduler)
);
const finalObserver = {
  next(x) {
    console.log('got value ' + x);
  },
  error(err) {
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  },
};
console.log('just before subscribe');
observable.subscribe(finalObserver);
console.log('just after subscribe');
```

Le `proxyObserver`est créé dans , et sa fonction est approximativement la suivante :`observeOn(asyncScheduler)next(val)`

```
    const proxyObserver = {
  next(val) {
    asyncScheduler.schedule(
      (x) => finalObserver.next(x),
      0 /* delay */,
      val /* will be the x for the function above */
    );
  },

  // ...
};
```

The `async` Scheduler operates with a `setTimeout` or `setInterval`, even if the given `delay` was zero. As usual, in JavaScript, `setTimeout(fn, 0)` is known to run the function `fn` earliest on the next event loop iteration. This explains why `got value 1` is delivered to the `finalObserver` after `just after subscribe` happened.

The `schedule()` method of a Scheduler takes a `delay` argument, which refers to a quantity of time relative to the Scheduler's own internal clock. A Scheduler's clock need not have any relation to the actual wall-clock time. This is how temporal operators like `delay` operate not on actual time, but on time dictated by the Scheduler's clock. This is specially useful in testing, where a *virtual time Scheduler* may be used to fake wall-clock time while in reality executing scheduled tasks synchronously.

Scheduler Types

The `async` Scheduler is one of the built-in schedulers provided by RxJS. Each of these can be created and returned by using static properties of the `Scheduler` object.

Scheduler	Purpose
`null`	By not passing any scheduler, notifications are delivered synchronously and recursively. Use this for constant-time operations or tail recursive operations.
`queueScheduler`	Schedules on a queue in the current event frame (trampoline scheduler). Use this for iteration operations.
`asapScheduler`	Schedules on the micro task queue, which is the same queue used for promises. Basically after the current job, but before the next job. Use this for asynchronous conversions.
`asyncScheduler`	Schedules work with `setInterval`. Use this for time-based operations.
`animationFrameScheduler`	Schedules task that will happen just before next browser content repaint. Can be used to create smooth browser animations.

Using Schedulers

You may have already used schedulers in your RxJS code without explicitly stating the type of schedulers to be used. This is because all Observable operators that deal with concurrency have optional schedulers. If you do not provide the scheduler, RxJS will pick a default scheduler by using the principle of least concurrency. This means that the scheduler which introduces the least amount of concurrency that satisfies the needs of the operator is chosen. For example, for operators returning an observable with a finite and small number of messages, RxJS uses no Scheduler, i.e. `null` or `undefined`. For operators returning a potentially large or infinite number of messages, `queue` Scheduler is used. For operators which use timers, `async` is used.

Because RxJS uses the least concurrency scheduler, you can pick a different scheduler if you want to introduce concurrency for performance purpose. To specify a particular scheduler, you can use those operator methods that take a scheduler, e.g., `from([10, 20, 30], asyncScheduler)`.

Static creation operators usually take a Scheduler as argument. For instance, `from(array, scheduler)` lets you specify the Scheduler to use when delivering each notification converted from the `array`. It is usually the last argument to the operator. The following static creation operators take a Scheduler argument:

Use `subscribeOn` to schedule in what context will the `subscribe()` call happen. By default, a `subscribe()` call on an Observable will happen synchronously and immediately. However, you may delay or schedule the actual subscription to happen on a given Scheduler, using the instance operator `subscribeOn(scheduler)`, where `scheduler` is an argument you provide.

Use `observeOn` to schedule in what context will notifications be delivered. As we saw in the examples above, instance operator `observeOn(scheduler)` introduces a mediator Observer between the source Observable and the destination Observer, where the mediator schedules calls to the destination Observer using your given `scheduler`.

Instance operators may take a Scheduler as argument.

Les opérateurs liés au temps comme `bufferTime`, `debounceTime`, `delay`, `auditTime`, `sampleTime`, `throttleTime`, `timeInterval`, `timeout`, `timeoutWith`, `windowTime`prennent tous un planificateur comme dernier argument, et fonctionnent autrement par défaut sur le `asyncScheduler`.

Autres opérateurs d'instance qui prennent un Scheduler comme argument : `cache`, `combineLatest`, `concat`, `expand`, `merge`, `publishReplay`, `startWith`.

Notez que les deux `cache`et `publishReplay`acceptent un Scheduler car ils utilisent un ReplaySubject. Le constructeur d'un ReplaySubjects prend un Scheduler facultatif comme dernier argument car ReplaySubject peut gérer le temps, ce qui n'a de sens que dans le contexte d'un Scheduler. Par défaut, un ReplaySubject utilise le `queue`Scheduler pour fournir une horloge.