Planificador / Scheduler

¿Qué es un Planificador? Un Planificador controla cuándo comienza una suscripción, y cuándo se envían las notificaciones. Está compuesto por tres componentes:

Un Planificador es una estructura de datos. Sabe cómo almacenar y poner tareas a la cola basándose en la prioridad u otro criterio.

Un Planificador es un contexto de ejecución. Denota dónde y cuándo se ejecuta una tarea (ej: inmediatamente, o mediante otro mecanismo de *callback* tal como `setTimeout`, `process.nextTick` o el Animation frame.)

Un Planificador tiene un reloj (virtual.) Este proporciona la noción de "tiempo" mediante un método de acceso `now()` en el Planificador. Las tareas que se planifican con un Planificador particular se adherirán únicamente a la noción de tiempo proporcionada por su reloj.

> *Un `Scheduler` permite definir en qué contexto de ejecución se enviarán las notificaciones de un Observable a su Observador.*

En el ejemplo mostrado a continuación, se parte del ejemplo simple de un Observable que emite los valores `1`, `2` y `3` de forma síncrona, y se utiliza el operador `observeOn` para especificar que el Planificador `async` será utilizado para enviar esos valores.

```
import { Observable, asyncScheduler } from "rxjs";
import { observeOn } from "rxjs/operators";

const observable = new Observable((observer) => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
}).pipe(observeOn(asyncScheduler));

console.log("Justo antes de subscribe");
observable.subscribe({
  next(x) {
    console.log("Obtenido valor " + x);
  },
  error(err) {
    console.error("Ha ocurrido algo : " + err);
  },
  complete() {
    console.log("Completado");
  },
});
console.log("Justo después de subscribe");
```

Al ejecutar el código, se recibe la siguiente salida:

```
Justo antes de subscribe
Justo después de subscribe
Obtenido valor 1
Obtenido valor 2
Obtenido valor 3
Completado
```

Nótese que las notificaciones `Obtenido valor...` se enviaron después de `Justo después de subscribe`. Este comportamiento es distinto al comportamiento por defecto que hemos podido ver hasta ahora. Esto es debido a que `observeOn(asyncScheduler)` introduce un Observador proxy entre `new Observable` y el Observador final. Se pueden renombrar los identificadores para hacer que esta distinción sea obvia, como podemos ver en el siguiente código:

```
import { Observable, asyncScheduler } from "rxjs";
import { observeOn } from "rxjs/operators";

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("Obtenido valor " + x);
  },
  error(err) {
    console.error("Ha ocurrido algo: " + err);
  },
  complete() {
    console.log("Completado");
  },
};

console.log("Justo antes de subscribe");
observable.subscribe(finalObserver);
console.log("Justo después subscribe");
```

El `proxyObserver` se crea en `observeOn(asyncScheduler)`, y su función `next(val)` es aproximadamente la que vemos a continuación:

```
const proxyObserver = {
  next(val) {
    asyncScheduler.schedule(
      (x) => finalObserver.next(x),
      0 /* Retardo */,
      val /* Será la x para la función anterior */
    );
  },

  // ...
};
```

El Planificador `async` opera con un `setTimeout` o `setInterval`, aunque el `delay` proporcionado sea cero. Como es habitual en JavaScript, `setTimeout(fn, 0)` ejecuta la función `fn` lo más pronto posible en la siguiente iteración del bucle de eventos. Esto explica por qué `Obtenido valor 1` se envía al `finalObserver` después de que ocurra `Justo después de subscribe`.

El método `schedule()` de un Planificador recibe un argumento `delay`, que se refiere a la cantidad de tiempo relativo al reloj interno propio del Planificador. El reloj del Planificador no tiene por qué tener relación alguna al tiempo de un reloj tradicional. Así es como los operadores temporales como `delay` operan según el tiempo dictaminado por el reloj del Planificador, en lugar de según el tiempo real. Esto es especialmente útil a la hora de hacer tests, ya que puede utilizarse un Planificador de tiempo virtual para falsificar el tiempo real, mientras se ejecutan tareas planificadas de forma síncrona.

Tipos de Planificadores

El Planificador `async` es uno de los Planificadores que proporciona RxJS. Cada uno de estos Planificadores se puede crear y retornar mediante las propiedades estáticas del objeto `Scheduler`.

Planificador	Función
null	Si no se proporciona un planificador, las notificaciones se emiten de forma

	síncrona y recursiva. Se debe utilizar para operaciones de tiempo constante u operaciones recursivas lineales finales.
queueScheduler	Planifica en una cola en el *event frame* actual (planificador trampolín.) Se debe utilizar para operaciones de iteración.
asapScheduler	Planifica en la cola de *micro task*, que es la misma cola que se utiliza para las promesas. Básicamente se planifica después de la tarea actual, pero antes de la siguiente tarea. Se debe utilizar para conversiones asíncronas.
asyncScheduler	Planifica tareas con setInterval. Se debe utilizar para operaciones basadas en el tiempo.
animationFrameScheduler	Planifica una tarea que ocurrirá justo antes del siguiente repintado del contenido del navegador. Se puede utilizar para crear animaciones más fluidas en el navegador.

Usando Planificadores

Es posible haber utilizado Planificadores en código RxJS sin tener que declarar explícitamente el tipo de Planificador que se vaya a usar. Esto es debido a que todos los operadores que tratan con concurrencia pueden recibir Planificadores de forma opcional. En el caso de que no se proporcione un Planificador, RxJS elegirá un Planificador mediante el principio de la concurrencia mínima. Esto quiere decir que se elegirá el

Planificador que introduzca la cantidad mínima de concurrencia que satisfaga las necesidades del operador. Por ejemplo, para los operadores que devuelven un Observable con un número finito y pequeño de mensajes, RxJS no utiliza ningún Planificador, ej: `null` o `undefined`. Para los operadores que devuelvan un número potencialmente grande o infinito de mensajes, se utiliza el Planificador `queue`. Para los operadores que utilizan temporizadores, se utiliza el Planificador `async`.

Dado que RxJS utiliza el Planificador de concurrencia mínima, se puede elegir un Planificador si se quiere introducir concurrencia por razones de rendimiento. Para especificar un Planificador concreto, se pueden utilizar aquellos métodos de operador que reciban un Planificador, ej: `from([10, 20, 30], asyncScheduler)`.

Los operadores de creación estáticos suelen recibir un Planificador como argumento. Por ejemplo, `from(array, scheduler)`, que nos permite especificar el Planificador que vaya a utilizarse para enviar cada notificación convertida del `array`. Suele ser el último argumento que recibe el operador. Los siguientes operadores de creación estáticos reciben un Planificador como argumento:

- `bindCallback`
- `bindNodeCallback`
- `combineLatest`
- `concat`
- `empty`
- `from`
- `fromPromise`
- `interval`
- `merge`
- `of`
- `range`
- `throw`
- `timer`

Se puede utilizar `subscribeOn` para planificar el contexto en el que ocurrirá la llamada `subscribe()`. Por defecto, una llamada a `subscribe()` en un Observable ocurre de forma síncrona e inmediata. Sin embargo, se puede retrasar o planificar la suscripción actual para que ocurra sobre un Planificador que se haya especificado, utilizando el operador de instancia `subscribeOn(scheduler)`, donde `scheduler` es el argumento que se proporciona.

Se puede utilizar `observeOn` para planificar el contexto en el que las notificaciones se entregan. Como se ha visto en los ejemplos anteriores, el operador de instancia `observeOn(scheduler)` introduce un Observador mediador entre el Observable fuente y el Observador destino, donde el mediador planifica las llamadas del Observador destino mediante el `scheduler` que se le haya proporcionado.

Los operadores de instancia pueden recibir un Planificador como argumento.

Los operadores de tiempo como `bufferTime`, `debounceTime`, `delay`, `auditTime`, `sampleTime`, `throttleTime`, `timeInterval`, `timeout`, `timeoutWith` o `windowTime` recibe un Planificador como último argumento opcional, utilizando por defecto el `asyncScheduler`.

Otros operadores de instancia que reciben un Planificador como argumento son `cache`, `combineLatest`, `concat`, `expand`, `merge`, `publishReplay` y `startWith`.

Nótese que tanto `cache` como `publishReplay` aceptan un Planificador porque utilizan un `ReplaySubject`. El constructor de los `ReplaySubjects` recibe un Planificador opcional como último argumento, ya que es posible que el `ReplaySubject` tenga que tratar con tiempo, lo que tiene sentido únicamente en el contexto de un Planificador. Por defecto, un `ReplaySubject` utiliza el Planificador `queue` para proporcionar un reloj.