EBook Gratis

APRENDIZAJE

rxjs

Free unaffiliated eBook created from
Stack Overflow contributors.

#rxjs

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rxjs

It is an unofficial and free rxjs ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rxjs.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con rxjs

Observaciones

Esta sección proporciona una descripción general de qué es rxjs y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de rxjs, y vincular a los temas relacionados. Dado que la Documentación para rxjs es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
RxJS 4	2015-09-25
RxJS 5	2016-12-13
RxJS 5.0.1	2016-12-13
RxJS 5.1.0	2017-02-01

Examples

Instalación o configuración

Utilizando un CDN:

```
<!DOCTYPE html>
<head>
  <script src="https://cdn.jsdelivr.net/rxjs/4.1.0/rx.min.js"></script>
</head>
<body>

  <script>
    // `Rx` is available
    var one$ = Rx.Observable.of(1);
    var onesub = one$.subscribe(function (one) {
      console.log(one); // 1
    });
    // alternatively: subscribe(console.log)
  </script>
</body>
</html>
```

CDN si usa RxJS 5 (RC) :

```
<script src="https://npmcdn.com/@reactivex/rxjs@5.0.0-rc.1/dist/global/Rx.js"></script>
```

Utilizando un bundler:

Primero instale en su directorio de proyecto con npm :

```
npm install rx
```

O utilizando RxJS 5 (RC) :

```
npm install rxjs
```

Luego, en tu archivo JavaScript:

```
var Rx = require('rx');
//var Rx = require('rxjs/Rx'); // v5beta

var one$ = Rx.Observable.of(1);
var onesub = one$.subscribe(function (one) {
  console.log(one); // 1
});
```

Si usa un bundle compatible con es6 / 2015:

```
import Rx from 'rx';
//import Rx from 'rxjs/Rx'; // v5beta

const one$ = Rx.Observable.of(1);
const onesub = one$.subscribe(one => console.log(one)); // 1
```

Lea Empezando con rxjs en línea: https://riptutorial.com/es/rxjs/topic/2361/empezando-con-rxjs

Capítulo 2: Operador: mapa / seleccionar

Sintaxis

- **Rx.Observable.prototype.map (selector, [thisArg])**
- **Rx.Observable.prototype.select (selector, [thisArg])**

Parámetros

Parámetro, Tipo	Detalles
`selector` , `Function` u `Object`	Transformar la función para aplicar a cada elemento fuente o un elemento para producir. Si selector es una función, se llama con la siguiente información: 1. el valor del elemento, 2. el índice del elemento, 3. el objeto observable que se está suscribiendo.
`[thisArg]` , `Any`	Objeto para usar como `this` al ejecutar el predicado.

Observaciones

`map` y `select` son alias.

Producen una secuencia observable que emite un elemento cada vez que la fuente observable emite un elemento.

Si el `selector` no es una función, su valor se emite para cada elemento fuente.

Si el `selector` es una función, el elemento emitido es el resultado de ejecutar el `selector` en el elemento fuente, y posiblemente puede usar su posición.

Examples

Usando un elemento para ceder

```
const md = Rx.Observable.fromEvent(document, 'mousedown').map(true);
// `md` will emit `true` whenever the mouse is pressed
const mu = Rx.Observable.fromEvent(document, 'mouseup').map(false);
// `mu` will emit `false` whenever the mouse is depressed
```

Usando una función de transformación

```
const source = Rx.Observable.range(1, 3)
  .map(x => x * x);
```

```
const subscription = source.subscribe(
  x => console.log(`Next: ${x}`),
  err => console.log(`Error: ${err}`),
  () => console.log(`Completed`)
);

// => Next: 1
// => Next: 4
// => Next: 9
// => Completed
```

Usando una función de transformación y el índice de elementos

```
const source = Rx.Observable.range(1, 3)
  .map((x, idx, obs) => `Element ${x} was at position ${idx}`);

const subscription = source.subscribe(
  x => console.log(`Next: ${x}`),
  err => console.log(`Error: ${err}`),
  () => console.log(`Completed`)
);

// => Next: Element 1 was at position 0
// => Next: Element 2 was at position 1
// => Next: Element 3 was at position 2
// => Completed
```

Lea Operador: mapa / seleccionar en línea: https://riptutorial.com/es/rxjs/topic/5419/operador--mapa---seleccionar

Capítulo 3: Operador: PublishReplay

Examples

¿Cómo funciona PublishReplay?

Crea internamente un `ReplaySubject` y lo hace compatible con `multicast` . El valor mínimo de reproducción de `ReplaySubject` es 1 emisión. Esto resulta en lo siguiente:

- La primera suscripción activará `publishReplay(1)` para suscribirse internamente al flujo fuente y canalizará todas las emisiones a través de `ReplaySubject` , almacenando efectivamente las últimas *n* (= 1) emisiones
- Si se inicia una segunda suscripción mientras la fuente aún está activa, la `multicast()` nos conectará al mismo objeto de `replaySubject` y recibiremos todas las próximas emisiones hasta que se complete la transmisión de la fuente.
- Si se inicia una suscripción después de que ya se haya completado la fuente, replaySubject ha guardado las últimas *n* emisiones y solo las recibirá antes de completarlas.

```
const source = Rx.Observable.from([1,2])
  .mergeMap(i => Rx.Observable.of('emission:'+i).delay(i * 100))
  .do(null,null,() => console.log('source stream completed'))
  .publishReplay(1)
  .refCount();

// two subscriptions which are both in time before the stream completes
source.subscribe(val => console.log(`sub1:${val}`), null, () => console.log('sub1
completed'));
source.subscribe(val => console.log(`sub2:${val}`), null, () => console.log('sub2
completed'));

// new subscription after the stream has completed already
setTimeout(() => {
  source.subscribe(val => console.log(`sub_late-to-the-party:${val}`), null, () =>
console.log('sub_late-to-the-party completed'));
}, 500);
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/5.0.3/Rx.js"></script>
```

Emisiones inesperadas al usar publishReplay

Basado en una pregunta . Los siguientes fragmentos de código no almacenan en caché la emisión esperada y previene más llamadas. En su lugar, vuelve a suscribirse a *realSource* para cada suscripción.

```
var state = 5
var realSource = Rx.Observable.create(observer =>  {
  console.log("creating expensive HTTP-based emission");
  observer.next(state++);
//  observer.complete(); //absent on purpose
```

```
  return () => {
    console.log('unsubscribing from source')
  }
});


var source = realSource
.do(null, null, () => console.log('stream completed'))
.publishReplay()
.refCount();

subscription1 = source.subscribe({next: (v) => console.log('observerA: ' + v)});
subscription1.unsubscribe();

subscription2 = source.subscribe(v => console.log('observerB: ' + v));
subscription2.unsubscribe();
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/5.1.0/Rx.js"></script>
```

Al ejecutar este fragmento de código, podemos ver claramente que no está emitiendo valores duplicados para el *Observador B* , de hecho, está creando nuevas emisiones para cada suscripción. ¿Cómo?

Cada suscripción se cancela antes de que se realice la próxima suscripción. Esto efectivamente hace que la refCount disminuya de nuevo a cero, no se está haciendo multidifusión.

El problema reside en el hecho de que la secuencia `realSource` *no se completa* . Debido a que no estamos realizando una multidifusión, el siguiente suscriptor obtiene una nueva instancia de `realSource` través del ReplaySubject y las nuevas emisiones están precedidas por las emisiones ya emitidas.

Lea Operador: PublishReplay en línea: https://riptutorial.com/es/rxjs/topic/8987/operador--publishreplay

Capítulo 4: Programador

Examples

Usando un TestScheduler para avanzar el tiempo manualmente

La mayoría de los operadores de Rx utilizan un programador opcional para programar sus futuras iteraciones. Si no se suministra, utilizarán su programador configurado predeterminado. El suministro de un planificador puede ser útil para propósitos de prueba en los que nos gusta hablar de tiempo virtual en lugar de tiempo real para la velocidad de ejecución de la prueba.

```
const scheduler = new Rx.TestScheduler();
scheduler.stop();
Rx.Observable.interval(100, scheduler)
  .do(i => console.log(i))
  .subscribe();
scheduler.advanceBy(10 * 100);
```

Lea Programador en línea: https://riptutorial.com/es/rxjs/topic/7991/programador

Capítulo 5: Recetas comunes

Introducción

Una colección de casos de uso comunes y su implementación en RxJS.

Examples

Cachear las respuestas HTTP

Un caso de uso típico para RxJS es crear solicitudes HTTP y almacenar en caché sus resultados durante un período de tiempo. Además, siempre queremos ejecutar solo una solicitud a la vez y compartir su respuesta.

Por ejemplo, el siguiente código almacena en caché **1** elemento para máx. **1000ms** :

```
var updateRequest = Observable.defer(() => makeMockHttpRequest())
    .publishReplay(1, 1000)
    .refCount()
    .take(1);

var counter = 1;
function makeMockHttpRequest() {
    return Observable.of(counter++)
        .delay(100);
}

function requestCachedHttpResult() {
    return updateRequest;
}
```

La función `makeMockHttpRequest()` simula una solicitud HTTP que llega con un retraso de `100ms` .

La función `requestCachedHttpResult()` es donde nos suscribimos para obtener una respuesta real o en caché.

Con `.publishReplay(1, 1000)` utilizamos la multidifusión RxJS para usar `ReplaySubject` y mantener `1` elemento por un máximo de `1000ms` . Luego, `refCount()` se usa para mantener siempre solo una suscripción a la `source` que es `Observable.defer()` . Este observable se utiliza para crear un nuevo `counter` solicitudes e incrementos para demostrar que los valores almacenados en caché y las nuevas suscripciones comparten los mismos datos.

Cuando queremos obtener datos actuales, llamamos `requestCachedHttpResult()` . Para garantizar que el Observer se complete correctamente después de emitir los datos, utilizamos el operador `take(1)` .

```
requestCachedHttpResult()
    .subscribe(val => console.log("Response 0:", val));
```

Esto crea una solicitud única con `mockDataFetch()` y se imprime en la consola:

```
1
```

Un ejemplo más complicado llamará a múltiples solicitudes en diferentes momentos en los que queremos probar que las conexiones y respuestas HTTP simuladas están compartidas.

```
requestCachedHttpResult()
    .subscribe(val => console.log("Response 0:", val));

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 50:", val))
, 50);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 200:", val))
, 200);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 1200:", val))
, 1200);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 1500:", val))
, 1500);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 3500:", val))
, 3500);
```

Vea una demostración en vivo: https://jsbin.com/todude/5/edit?js,console

Cada solicitud se envía con retraso y se realiza en el siguiente orden:

`0` : primera solicitud que hace que `refCount()` suscriba a su `source` que hace que la llamada `mockDataFetch()` . Su respuesta va a ser retrasada por `100ms` . En este momento, el operador `ConnectableObservable` inside `publishReplay()` tiene **un** observador.

`50` - La segunda solicitud también se suscribe a `ConnectableObservable` . En este momento, el operador `ConnectableObservable` inside `publishReplay()` tiene **dos** observadores. No crea otra solicitud con `makeMockHttpRequest()` porque `refCount()` ya está suscrito.

`100` - La primera respuesta está lista. Primero se almacena en caché en `ReplaySubject` y luego se `ReplaySubject` a los **dos** Observadores suscritos a `ConnectableObservable` . Ambos observadores se completaron gracias a `take(1)` y anular la suscripción.

`200` : se suscribe a `ReplaySubject` que emite de inmediato su valor almacenado en caché, lo que hace que `take(1)` complete el Observer y cancele la suscripción de inmediato. No se realizan solicitudes HTTP y no queda ninguna suscripción.

`1200` - Lo mismo que el primer evento en `0` . En este punto, el valor almacenado en caché se ha descartado porque es más antiguo que `1000ms` .

`1500`

- Igual que el cuarto evento en `200` .

`3500` - Lo mismo que el primer evento a las `1200` .

La salida en consola es la siguiente:

```
Response 0: 1
Response 50: 1
Response 200: 1
Response 1200: 2
Response 1500: 2
Response 3500: 3
```

En RxJS 5 una funcionalidad similar fue cubierta por el operador `cache()` . Sin embargo, se eliminó en `5.0.0-rc.1` debido a su funcionalidad limitada.

Errores de manejo

Si queremos manejar los errores producidos por el servicio remoto (la función `makeMockHttpRequest`), debemos capturarlos antes de que se fusionen en la cadena principal de Observable porque cualquier error recibido por `ReplaySubject` en `publishReplay()` marcaría su estado interno como `stopped` (Lea más aquí Sujeto y su estado interno) que definitivamente no es lo que queremos.

En el siguiente ejemplo, simulamos un error cuando `counter === 2` y lo detectamos con el operador `catch()` . Estamos usando `catch()` para transformar solo la notificación de `error` en un `next` regular, por lo que podemos manejar el error en los observadores:

```
function makeMockHttpRequest() {
    return Observable.of(counter++)
        .delay(100)
        .map(i => {
            if (i === 2) {
                throw new Error('Invalid URL');
            }
            return i;
        })
        .catch(err => Observable.of(err));
}
```

Vea una demostración en vivo: https://jsbin.com/kavihu/10/edit?js,console

Esto imprimirá a la consola la siguiente salida. Note que los errores son recibidos en los `next` manejadores:

```
Response 0: 1
Response 50: 1
Response 200: 1
Response 1200: [object Error] { ... }
Response 1500: [object Error] { ... }
Response 3500: 3
```

Si queremos manejar los errores como notificaciones de `error` regulares en cada observador, tenemos que volver a `publishReplay()` después del operador `publishReplay()` por los motivos

explicados anteriormente.

```
var updateRequest = Observable.defer(() => makeMockHttpRequest())
    .publishReplay(1, 1000)
    .refCount()
    .take(1)
    .map(val => {
        if (val instanceof Error) {
            throw val;
        }
        return val;
    });
```

Vea la demostración en vivo: https://jsbin.com/fabosam/5/edit?js,console (observe que también tuvimos que agregar devoluciones de llamada de error para cada observador).

```
Response 0: 1
Response 50: 1
Response 200: 1
error callback: Error: Invalid URL
error callback: Error: Invalid URL
Response 3500: 3
```

Descartando llamadas de descanso lento / obsoleto

Un caso de uso común es descartar ciertas llamadas de descanso, que ya no son necesarias después de ciertas entradas de usuario. El ejemplo más destacado sería, cuando un usuario utiliza alguna función de búsqueda, realiza una solicitud, realiza otra solicitud y, por algún motivo, la primera solicitud llega después de la segunda solicitud y la aplicación muestra los datos desactualizados de la solicitud anterior.

Este es un caso de uso perfecto para el `switchMap` switchMap.

```
searchInput$
    .switchMap(() => makeRestCall());
```

En este caso, la transmisión `switch` a la llamada de descanso, pero solo hasta que se emitan nuevos datos en `searchInput$` , luego se descarta la transmisión dentro del `switchMap` y se realiza una nueva llamada de descanso. Por lo tanto, un resultado de reposo solo se considerará si finalizó antes del siguiente clic.

Y aquí hay un ejemplo completamente burlado:

```
// some initial data-mocking
const Observable = Rx.Observable;
var counter = 1;
function mockDataFetch() {
    return Observable.of(counter++)
        .delay(500);
}

// the recipe
```

```
const searchInput$ = new Rx.Subject();
searchInput$
    .do(searchInput => console.log("Searching for " + searchInput))
    .switchMap(searchInput => mockDataFetch()
              .map(result => ({result, searchInput}))
     )
    .do(data => console.log("Received result for " + data.searchInput + ": " + data.result))
    .subscribe();

// simulating search inputs
searchInput$.next("google");
setTimeout(() => searchInput$.next("froogle"), 600);
setTimeout(() => searchInput$.next("doodle"), 800);
setTimeout(() => searchInput$.next("poodle"), 1000);
setTimeout(() => searchInput$.next("noodle"), 1600);
```

Vea la demostración en vivo: https://jsbin.com/suzakahoro/1/edit?js,console

Enviando múltiples solicitudes HTTP paralelas

Un caso de uso muy común en aplicaciones web es realizar múltiples solicitudes asíncronas (por ejemplo, HTTP) y recopilar sus resultados a medida que llegan o todos a la vez (por ejemplo, en Angular2 con el servicio HTTP).

1. Recopilando respuestas asíncronas una por una a medida que llegan

Esto normalmente se hace con el operador `mergeMap()` que toma una función de proyección que tiene que devolver un Observable. El operador `mergeMap()` suscribe internamente a cada Observable inmediatamente, incluso si el Observable anterior aún no se ha completado.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url))
  .subscribe(val => console.log(val));
```

Esto imprime las respuestas a la consola en un orden diferente debido al retraso aleatorio:

```
Response from url-3
Response from url-4
Response from url-2
Response from url-1
```

Vea una demostración en vivo: https://jsbin.com/xaqudan/2/edit?js,console

Cada respuesta (elemento emitido a través de la `next` llamada) es reemitida por `mergeMap()` inmediatamente.

Para nuestro propósito de enviar múltiples solicitudes HTTP, es útil mencionar que `mergeMap()` puede tomar tres argumentos en total:

1. La función de proyección que necesita para devolver un observable.
2. La función de selección de resultados que nos permite modificar el resultado antes de emitirlo.
3. El número de Observables suscritos concurrentemente.

Controlando el número de peticiones paralelas.

Con el tercer argumento podemos controlar cuántas solicitudes paralelas manejaremos (asumiendo que cada Observable que realiza una solicitud HTTP es "frío").

En el siguiente ejemplo, ejecutaremos solo 2 solicitudes al mismo tiempo.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url), undefined, 2)
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

Vea una demostración en vivo: https://jsbin.com/sojejal/4/edit?js,console

Observe que las dos primeras solicitudes se completaron después de 1 s mientras que las otras dos después de 2s.

```
[1004, "Response from url-1"]
[1010, "Response from url-2"]
[2007, "Response from url-3"]
[2012, "Response from url-4"]
```

Errores de manejo

Si cualquiera de los observables origen fracasan (envía `error` de notificación) del `mergeMap()` reenvía el error más como `error` . En caso de que queramos que cada Observable falle correctamente, necesitamos usar, por ejemplo, el operador `catch()` .

```
function mockHTTPRequest(url) {
  return Observable.of(`Response from ${url}`)
    .delay(Math.random() * 1000)
    .map(value => {
      if (url === 'url-3') {
        throw new Error(`Error response from ${url}`)
      }
      return value;
```

```
    });
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url).catch(() => Observable.empty()))
  .subscribe(val => console.log(val));
```

La respuesta para `url-3` arroja un error que se envía como notificación de `error` . Más tarde, el operador `catch()` y lo reemplaza con `Observable.empty()` que es solo una notificación `complete` . Por esta razón esta respuesta es ignorada.

La salida para este ejemplo es la siguiente:

```
Response from url-4
Response from url-1
Response from url-2
```

Vea una demostración en vivo: https://jsbin.com/kuqumud/4/edit?js,console

2. Recopilando todas las respuestas asíncronas a la vez.

Siguiendo los ejemplos anteriores, podríamos reunir todas las respuestas con el operador `toArray()` .

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url))
  .toArray()
  .subscribe(val => console.log(val));
```

Sin embargo, el uso del operador `toArray()` tiene una consecuencia importante. El hecho de que el suscriptor reciba los resultados no solo se controla al completar todas las solicitudes HTTP, sino también al completar la fuente Observable (`Observable.from` en nuestro caso). Esto significa que no podemos usar Observables de origen que nunca se completan (por ejemplo, `Observable.fromEvent`).

Otra forma de lograr el mismo resultado es usar `Observable.forkJoin()` que toma como argumento un conjunto de Observables a los que queremos suscribirnos y esperar hasta que todos ellos **emitan al menos un valor y estén completos** .

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}
```

```
var urls = ['url-1', 'url-2', 'url-3', 'url-4'];
var observables = urls.map(url => mockHTTPRequest(url));

Observable.forkJoin(observables)
  .subscribe(val => console.log(val));
```

Esto imprime todas las respuestas como una sola matriz:

```
["Response from url-1", "Response from url-2", "Response from url-3", "Response from url-4"]
```

Ver demostración en vivo: https://jsbin.com/fomoye/2/edit?js,console

`Observable.forkJoin()` también toma como argumento opcional una función de selección de resultados que nos permite modificar el resultado final antes de emitirlo:

```
Observable.forkJoin(observables, (...results) => {
    return results.length;
  })
  .subscribe(val => console.log(val));
```

Esto imprime a la consola:

```
4
```

Vea una demostración en vivo: https://jsbin.com/muwiqic/1/edit?js,console

Tenga en cuenta que los argumentos para la función selectora de resultados están desempaquetados.

Errores de manejo

Para el manejo de errores podemos usar el mismo enfoque que en el ejemplo anterior con el operador `catch()` .

Sin embargo, hay una cosa importante a tener en cuenta. El `forkJoin()` requiere que cada fuente observable emita al menos un valor. Si utilizamos `catch(() => Observable.empty())` como hicimos antes, `forkJoin()` nunca emitiría nada porque `Observable.empty()` es solo una notificación `complete` .

Esta es la razón por la que necesitamos usar, por ejemplo, `Observable.of(null)` que es un valor `null` seguido de `complete` notificación `complete` .

```
function mockHTTPRequest(url) {
  return Observable.of(`Response from ${url}`)
    .delay(Math.random() * 1000)
    .map(value => {
      if (url === 'url-3') {
        throw new Error(`Error response from ${url}`)
      }
      return value;
    });
}
```

```
var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

var observables = urls.map(url => mockHTTPRequest(url).catch(() => Observable.of(null)));

Observable.forkJoin(observables)
  .subscribe(val => console.log(val));
```

Ver demostración en vivo: https://jsbin.com/yidiked/2/edit?js,console

Esto imprime a la consola:

```
["Response from url-1", "Response from url-2", null, "Response from url-4"]
```

Observe que el error es reemplazado por `null` . Si solo `forkJoin() Observable.empty() forkJoin()` nunca emitiría nada.

Enviando múltiples solicitudes HTTP secuenciales

Hacer una secuencia de solicitudes HTTP tiene dos razones principales:

- Las solicitudes dependen una de la otra (el resultado de una solicitud se requiere para una solicitud consecutiva).
- Queremos repartir la carga del servidor en múltiples solicitudes.

1. Hacer múltiples solicitudes dependientes.

Esto se puede realizar utilizando el operador `concatMap()` para transformar una respuesta a los parámetros requeridos para la solicitud consecutiva.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

function timestamp() {
  return (new Date()).getTime() - start;
}

var start = (new Date()).getTime();

Observable.of('url-1')
  // first request
  .concatMap(url => {
    console.log(timestamp() + ': Sending request to ' + url);
    return mockHTTPRequest(url);
  })
  .do(response => console.log(timestamp() + ': ' + response))

  // second request
  .concatMap(response => {
    console.log(timestamp() + ': Sending request to ' + response);
    let newUrl = 'url-' + response.length; // create new requests url
    return mockHTTPRequest(newUrl);
  })
```

```
  .do(response => console.log(timestamp() + ': ' + response))

  // third request
  .concatMap(response => {
    console.log(timestamp() + ': Sending request to ' + response);
    let newUrl = 'url-' + response.length; // create another requests url
    return mockHTTPRequest(newUrl);
  })
  .subscribe(response => console.log(timestamp() + ': ' + response));
```

El operador `concatMap()` suscribe internamente al Observable devuelto desde su función de proyección y espera hasta que se complete mientras se reemiten todos sus valores.

Este ejemplo marca la hora de cada solicitud y respuesta:

```
3: Sending request to url-1
1010: Response from url-1
1011: Sending request to Response from url-1
2014: Response from url-19
2015: Sending request to Response from url-19
3017: Response from url-20
```

Vea la demostración en vivo: https://jsbin.com/fewidiv/6/edit?js,console

Errores de manejo

Si alguna de las solicitudes HTTP falla, obviamente no queremos continuar porque no podemos construir la siguiente solicitud.

2. Hacer solicitudes consecutivas

En caso de que no estemos interesados en la respuesta de la solicitud HTTP anterior, podemos tomar solo una matriz de URL y ejecutarlas una tras otra.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .concatMap(url => mockHTTPRequest(url))
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

Este ejemplo imprime respuestas con marca de tiempo:

```
[1006, "Response from url-1"]
[2012, "Response from url-2"]
[3014, "Response from url-3"]
[4016, "Response from url-4"]
```

Vea una demostración en vivo: https://jsbin.com/kakede/3/edit?js,console

Retrasando llamadas consecutivas

También podríamos querer hacer un pequeño retraso entre cada solicitud. En tal caso, debemos agregar el `delay()` después de cada llamada `mockHTTPRequest()` .

```
Observable.from(urls)
  .concatMap(url => {
    return mockHTTPRequest(url)
      .do(response => console.log(((new Date()).getTime() - start) + ': Sending request to ' +
url))
      .delay(500);
  })
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

Esto imprime para consolar la siguiente salida:

```
2024: Sending request to url-1
[2833, "Response from url-1"]
4569: Sending request to url-2
[5897, "Response from url-2"]
7880: Sending request to url-3
[8674, "Response from url-3"]
9789: Sending request to url-4
[10796, "Response from url-4"]
```

Vea una demostración en vivo: https://jsbin.com/kakede/4/edit?js,console

Errores de manejo

Si simplemente queremos ignorar cuando falla alguna de las solicitudes HTTP, tenemos que encadenar `catch()` cada `mockHTTPRequest()` .

```
function mockHTTPRequest(url) {
  if (url == 'url-3') {
    return Observable.throw(new Error(`Request ${url} failed.`));
  } else {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
  }
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .concatMap(url => mockHTTPRequest(url).catch(obs => Observable.empty()))
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

Esto simplemente ignora la llamada `url-3` :

```
[1004, "Response from url-1"]
[2012, "Response from url-2"]
[3016, "Response from url-4"]
```

Ver demostración en vivo: https://jsbin.com/jowiqo/2/edit?js,console

Si no utilizáramos el operador `catch()` , la `url-3` causaría que la cadena enviara una notificación de error y la última `url-4` no se ejecutaría.

Vea una demostración en vivo: https://jsbin.com/docapim/3/edit?js,console

Límite de velocidad

Un problema común con los servicios remotos es la limitación de velocidad. El servicio remoto nos permite enviar solo un número limitado de solicitudes o cantidad de datos por período de tiempo.

En RxJS 5, el operador `bufferTime` proporciona una funcionalidad muy similar, y especialmente si dejamos el segundo parámetro sin especificar (define con qué frecuencia queremos crear un nuevo búfer. Si lo dejamos sin definir / nulo, creará un nuevo búfer correcto). después de emitir el actual).

Un uso típico de `bufferTime` se verá así:

```
bufferTime(1000, null, 5)
```

Esto almacenará elementos hasta que se cumpla una de las dos condiciones. Luego emitirá el búfer y arrancará otro:

- El operador ha estado recogiendo artículos por `1000ms`
- El operador ya ha recogido `5` artículos.

Para fines de demostración, podemos crear una fuente observable que emita muy rápido, por lo que `bufferTime` alcanzará el límite de tamaño (`5`) y emitirá más a menudo que una vez cada `1000ms` :

```
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));
```

Luego lo encadenaremos con `bufferTime` y `concatMap` . El operador `concatMap` es donde `1000ms` retraso de `1000ms` :

```
const startTime = (new Date()).getTime();

const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));

source.bufferTime(1000, null, 5)
  .concatMap(buffer => Observable.of(buffer).delay(1000))
  .timestamp()
```

```
  .map(obj => {
    obj.timestamp = obj.timestamp - startTime;
    return obj;
  })
  .subscribe(obj => console.log(obj));
```

Vea la demostración en vivo: https://jsbin.com/kotibow/3/edit?js,console

También agregamos la `timestamp()` para ver los tiempos de emisión para asegurarnos de que el retraso sea realmente de al menos `1000ms` .

Tenga en cuenta que no tuvimos que usar `Observable.of(buffer)` en absoluto. Lo estamos utilizando aquí solo para verificar manualmente que el número de elementos almacenados en búfer es correcto.

Desde la salida de la consola podemos ver que el retraso entre dos emisiones es de aproximadamente `1000ms` :

```
Timestamp { value: [ 1, 2, 3, 4, 5 ], timestamp: 1475 }
Timestamp { value: [ 6, 7, 8, 9, 10 ], timestamp: 2564 }
Timestamp { value: [ 11, 12, 13, 14, 15 ], timestamp: 3567 }
Timestamp { value: [ 16, 17, 18, 19, 20 ], timestamp: 4572 }
Timestamp { value: [ 21, 22, 23, 24, 25 ], timestamp: 5573 }
Timestamp { value: [], timestamp: 6578 }
```

Ahora también podemos probar una situación en la que la fuente se emite lentamente para que el operador de `bufferTime` la condición de intervalo máximo:

```
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(300));
```

Vea una demostración en vivo: https://jsbin.com/tuwowan/2/edit?js,console

Luego, la salida debería comenzar después de aproximadamente `2s` ya que tomó `1s` para que el operador `bufferTime` emitiera y luego agregamos la demora de `1s` ;

```
Timestamp { value: [ 1, 2, 3 ], timestamp: 2017 }
Timestamp { value: [ 4, 5, 6 ], timestamp: 3079 }
Timestamp { value: [ 7, 8, 9, 10 ], timestamp: 4088 }
Timestamp { value: [ 11, 12, 13 ], timestamp: 5093 }
Timestamp { value: [ 14, 15, 16 ], timestamp: 6094 }
Timestamp { value: [ 17, 18, 19, 20 ], timestamp: 7098 }
Timestamp { value: [ 21, 22, 23 ], timestamp: 8103 }
Timestamp { value: [ 24, 25 ], timestamp: 9104 }
```

Si quisiéramos usar este enfoque en una aplicación del mundo real, pondríamos la llamada remota en el operador `concatMap` . De esta manera podemos controlar si queremos forzar el retraso de `1s` entre solicitudes o respuestas del servicio remoto.

Por ejemplo, podemos forzar la demora mínima de `1s` entre solicitudes usando `forkJoin` en la `concatMap` llamada `concatMap` :

```
function mockHTTPRequest(buffer) {
  return Observable.of(true).delay(Math.random() * 1500)
}

const startTime = (new Date()).getTime();
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));

source.bufferTime(1000, null, 5)
  .concatMap(buffer => Observable.forkJoin(
    mockHTTPRequest(buffer),
    Observable.of(buffer).delay(1000)
  ))
  .timestamp()
  .map(obj => {
    obj.timestamp = obj.timestamp - startTime;
    return obj;
  })
  .subscribe(obj => console.log(obj));
```

Vea una demostración en vivo: https://jsbin.com/xijaver/edit?js,console

Gracias a `forkJoin` el `concatMap` debe esperar a que ambos Observables se completen.

Por otro lado, si quisiéramos forzar el retraso de `1s` entre las respuestas, simplemente `mockHTTPRequest()` operador `delay()` después de `mockHTTPRequest()` :

```
.concatMap(buffer => mockHTTPRequest(buffer).delay(1000))
```

Vea la demostración en vivo: https://jsbin.com/munopot/2/edit?js,console

Lea Recetas comunes en línea: https://riptutorial.com/es/rxjs/topic/8247/recetas-comunes

Capítulo 6: Reintentar y reintentar cuando los operadores

Introducción

Reintentar y Reintentar cuando se puede usar para intentar recuperar los Observables que puedan tener errores en su flujo.

Sintaxis

1. .retry (n: número): observable
    - n: reintentar intentará la fuente Observable esta varias veces.
2. .retryWhen (recibe: notificationHandler, the: scheduler): Observable
    - Recibe: un Observable de notificaciones que el uso puede completar o error.
        - Si el observable 'recibe' devuelve limpiamente (completa), se volverá a intentar la fuente observable.
        - Si el observable 'recibe' devuelve un error, la fuente observable se cancela.
    - planificador: el origen observable está suscrito a este planificador.

Examples

Vuelva a intentar con el retroceso, hasta el éxito o el número máximo de intentos

El siguiente código intentará ejecutar `loadFromHttp()` hasta 5 veces (`maxAttempts`), con cada intento retrasado tantos segundos. Si se supera a `maxAttempts` , el observable se rinde.

```
// assume loadFromHttp() returns a Promise, which might fail.
Rx.Observable.from(loadFromHttp())
.retryWhen((attempts) => {
    let maxAttempts = 5;

    Rx.Observable.range(1, maxAttempts+1).zip(attempts, (i, attempt) => [i, attempt])
    .flatMap(([i, attempt]) => {
        if (i <= maxAttempts) {
            console.log(`Retrying in ${i} second(s)`);
            return Rx.Observable.timer(i * 1000);
        } else {
            throw attempt;
        }
    })
})
```

Lea Reintentar y reintentar cuando los operadores en línea: https://riptutorial.com/es/rxjs/topic/9026/reintentar-y-reintentar-cuando-los-operadores

Capítulo 7: Tema

Introducción

Los sujetos son clases que se comportan como observables y observadores al mismo tiempo.

http://reactivex.io/documentation/subject.html

Examples

El sujeto y su estado interno

En Rx los sujetos tienen estados internos que pueden controlar su comportamiento.

Un formulario de caso de uso común El sujeto lo suscribe a múltiples observables. El siguiente ejemplo crea dos Observables diferentes y suscribe un Sujeto a ambos. Luego intenta imprimir todos los valores que pasaron por:

```
let subject = new Subject();
subject.subscribe(val => console.log(val));

Observable.range(1, 5).subscribe(subject);
Observable.from(['a', 'b', 'c']).subscribe(subject);
```

Vea una demostración en vivo: https://jsbin.com/pesumup/2/edit?js,console

Este ejemplo solo imprime los números del `1 - 5` y no imprimió ninguno de los caracteres `a` , `b` , `c` .

```
1
2
3
4
5
```

La pregunta es ¿qué pasó? El problema aquí es el estado interno de la instancia del sujeto cuando recibió la notificación `complete` . Cuando un sujeto recibe un `error` o notificaciones `complete` se marca como detenido y nunca emitirá ninguna otra señal .

Debe ser así porque los sujetos son básicamente observables y los observables solo pueden emitir una notificación `complete` o de `error` al final de la secuencia, pero nunca ambas.

El problema con el ejemplo anterior es que el primer Observable.range `Observable.range()` también emite la notificación `complete` que luego recibe el Asunto y, por lo tanto, no vuelve a emitir ningún valor cuando se suscribe al segundo Observable.

Podemos ver que el Asunto realmente recibe la notificación `complete` configurando también la devolución de llamada completa.

```
subject.subscribe(val => console.log(val), null, () => console.log('complete'));
```

La salida es la misma justo al final, también se imprime `complete` .

```
1
2
3
4
5
complete
```

Entonces, si no queremos que el Asunto reciba la notificación `complete` , podemos enviar manualmente las `next` señales. Esto significa que en lugar de suscribir directamente al sujeto, suscribiremos una devolución de llamada que llame al método `next()` en el sujeto:

```
Observable.range(1, 5).subscribe(val => subject.next(val));
Observable.from(['a', 'b', 'c']).subscribe(val => subject.next(val));
```

Vea una demostración en vivo: https://jsbin.com/funeka/1/edit?js,console

```
1
2
3
4
5
a
b
c
```

Tenga en cuenta que este mismo principio se aplica en todos los lugares donde usamos Sujetos.

Por ejemplo, los operadores como `publish()` , `share()` y todas sus variantes que usan la misma instancia de Subject bajo el capó se ven afectados por esto.

Lea Tema en línea: https://riptutorial.com/es/rxjs/topic/9518/tema

Creditos

S. No	Capítulos	Contributors
1	Empezando con rxjs	bloodyKnuckles, Community, gsc, martin, Ptival
2	Operador: mapa / seleccionar	Ptival
3	Operador: PublishReplay	Mark van Straten
4	Programador	Mark van Straten
5	Recetas comunes	gsc, martin, olsn, vek
6	Reintentar y reintentar cuando los operadores	JBCP
7	Tema	martin