Here is what I've learn about groupBy operator by reading RxJS sources

Hidden gotchas when building a pipeline to group data based on criteria

Kiran Holla Follow
Oct 19, 2018 · 7 min read

. . .

Most developers have probably already heard of Reactive Programming, and in the JavaScript world, specifically of **RxJS**. **RxJS** is an extremely powerful library that allows us to deal with and manipulate streams of data. This pattern can be used to deal with asynchronous events as well as to create data processing *pipelines* that may not necessarily be asynchronous. The library provides many features/operators that allow developers to break down business logic into simple pieces of code that can be composed together to achieve extremely complex data transformations.

. . .

The day RxJS left me scratching my head…

For all the power that RxJS affords us, it can also be frustratingly hard to learn. While good documentation in the form of detailed operator-level documentation and API Reference manuals do exist, tricky scenarios still can catch you off-guard. I recently experienced this first hand when setting up a data processing *pipeline* to transform a stream of data into categorized sets.

Here is an example of some input data:

```
const records = [
  { id: 'a', category: 1 },
  { id: 'b', category: 2 },
  { id: 'c', category: 3 },
  { id: 'd', category: 1 },
  { id: 'e', category: 2 },
  { id: 'f', category: 3 },
  { id: 'g', category: 1 },
  { id: 'h', category: 2 }
];
```

I needed the output to look like this:

```
const result = [
  // Category 1
  { key: 1, value: { id: 'a', category: 1 } },
  { key: 1, value: { id: 'd', category: 1 } },
  { key: 1, value: { id: 'g', category: 1 } },

  // Category 2
  { key: 2, value: { id: 'b', category: 2 } },
  { key: 2, value: { id: 'e', category: 2 } },

  // Category 3
  { key: 3, value: { id: 'c', category: 3 } },
  { key: 3, value: { id: 'f', category: 3 } }
];
```

• • •

The data pipeline

I set out to create a *pipeline* that would take a series of data points, classify them based on some attributes, and then output the classified sets sequentially. While such functionality can easily be achieved using simple functional programming techniques or libraries, I chose RxJS because the data points were expected to arrive asynchronously.

My proposed solution was simple: an observable stream piped into the `groupBy` operator, followed by the `concatMap` operator to get the sequential data sets at the end. I thought this was a simple solution based on the documentation for the `groupBy` and `concatMap` operators at Reactivex.io.

The `groupBy` operator would create individual Observable streams for each *key* within the source data. Each of these Observable streams would then have the individual data elements that belonged to that *key*. The `concatMap` operator, which followed, would then merge all of these individual Observable streams together, one after another, into one single stream that would then form the output.

A proof of concept

To check if the approach I had in mind would actually work, I created some simple proof-of-concept code based on examples I had seen online.

```
const records = ['a', 'b', 'c', 'd'];
const pipedRecords = new Subject();


const result = pipedRecords.pipe(
  concatMap(
    group => group.subject$.pipe(
      take(2),
      map(x => ev.key + x)
    )
  )
);

const subscription = result.subscribe(x => console.log(x));

records.forEach(
  x => pipedRecords.next({key: x, subject$: interval(1000)})
);
pipedRecords.complete();

// Expected & Actual Output:
// a0
// a1
// b0
// b1
// c0
// c1
// d0
// d1
```

The results from the proof-of-concept were encouraging. I seemed to be getting exactly what I needed, and I felt that the approach of using the `concatMap` operator to collate values from a series of Observables would work.

. . .

Now for the real deal

Here is an over-simplified version of the final code I came up with:

```
const pipedRecords = new Subject();
const result = pipedRecords.pipe(
  groupBy(
    x => x.category
  ),
  concatMap(
    group$ => group$.pipe(
      map(obj => ({ key: group$.key, value: obj }))
    )
  )
);

const subscription = result.subscribe(x => console.log(x));

records.forEach(x => pipedRecords.next(x));
pipedRecords.complete();
```

The output

Here's the output I got from my code:

```
const result = [
  { key: 1, value: { id: 'a', category: 1 } },
  { key: 1, value: { id: 'd', category: 1 } },
  { key: 1, value: { id: 'g', category: 1 } }
];
```

I had obviously gone wrong somewhere, because my *pipeline* failed to return all the rows that I had put into it.

. . .

Some detective work

After my search within the RxJS API refrence and on ReactiveX.io drew a blank, I decided dive into the RxJS code to try and understand what was going on here.

Reading through the code for the `groupBy` operator, I learned that `groupBy` internally creates a new `Subject` instance for each *key* that is found in the source stream. All values belonging to that key are then immediately emitted by that `Subject` instance.

All the `Subject` instances are wrapped into `GroupedObservable` instances and emitted downstream by the `groupBy` operator. This stream of `GroupedObservable` instances is the input to the `concatMap` operator.

The `concatMap` operator internally calls the `mergeMap` operator with a value of 1 for `concurrency` , which means only one source Observable is subscribed to concurrently. The `mergeMap` operator in this case will subscribe to only one Observable, as is allowed by the `conccurency` parameter we passed in, and all other Observables will be held in a *buffer* till the first one has completed.

A quick note on the RxJS Subject

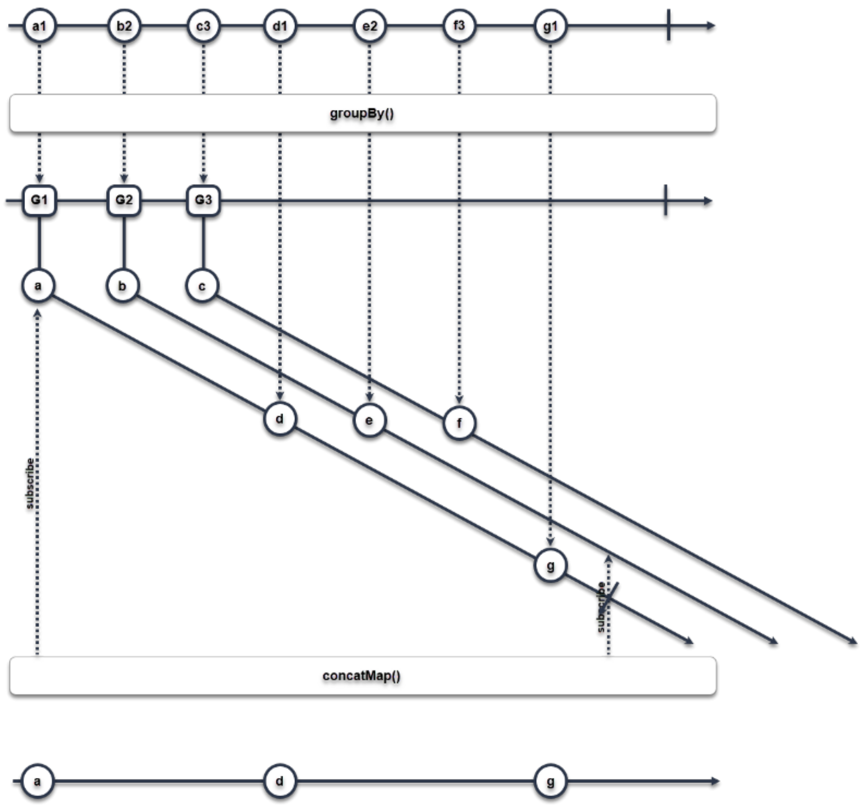
At this stage we must understand that `Subject` instances are what can be considered *hot* Observables. That is, they begin producing notifications irrespective of whether or not there are any subscribers.

Hence, if an observer comes in and subscribes to a `Subject` in the middle of its stream of notifications, then that observer will have missed all values that were produced before the subscription.

So what?

The `groupBy` operator emits `Subject` instances for each *key*, and then uses that `Subject` instance to immediately emit the individual values as well. However, the `mergeMap` operator subscribes to only one of the `Subject` instances at any point in time, starting with the first one.

A quick illustration of this can help make this clear:



Values belonging to the second and third groups: b, c, e, f, are lost as they are never seen by the concatMap() operator

As can be seen in the above diagram, the `mergeMap` operator subscribes to `Subject` s for the second and subsequent keys only after the first `Subject` has completed. However, the `Subject` s for the second and subsequent keys emit their values as soon as they receive them from the source stream.

Since the `mergeMap` operator does not subscribe to any subsequent `Subject` instances till the entire source stream has been exhausted, none of the values from those `Subject` instances are seen by it and are effectively lost.

How do we fix this?

Considering the amount of time I had to spend investigating this problem, the fix turned out to be surprisingly simple. I just forced the `groupBy` operator to use a `ReplaySubject` instead of a `Subject` instance. `ReplaySubject` s are very similar to `Subject` s with one crucial difference. A `ReplaySubject` will always ensure that any new observers will also receive all the values that were emitted before their subscription, in addition to any new values that get emitted in future. Hence, using a `ReplaySubject` helps us ensure that none of the values are lost due to the timing of the subscription.

The `groupBy` accepts a `subjectSelector` parameter that allows us to switch the `Subject` instance with a `ReplaySubject` instance.

The following code works:

```
const pipedRecords = new Subject();
const result = pipedRecords.pipe(
  groupBy(
    x => x.category,
    null,
    null,
    () => new ReplaySubject()  // Use ReplaySubject instead
  ),
  concatMap(
    group$ => group$.pipe(
      map(obj => ({ key: group$.key, value: obj }))
    )
  )
);

const subscription = result.subscribe(x => console.log(x));


records.forEach(x => pipedRecords.next(x));
pipedRecords.complete();
```

. . .

OK, but why did the proof-of-concept work?

The difference between my proof of concept and the real code lies in the difference between *hot* and *cold* Observables.

In the real code, my stream of Observables were being generated from within the `groupBy` operator. The `groupBy` operator made use of `Subject` s to emit values. `Subject` s are *hot* Observables. They produce their notifications irrespective of whether there are subscribers or not.

The proof-of-concept made use of `interval` to simulate a stream of Observables. But `interval` creates a *cold* Observable which does not start producing notifications until someone has actually subscribed to it.

There in lies the difference. In the proof-of-concept the `concatMap` operator did not lose any of the values because they were not emitted until it came around to subscribing to that Observable.

. . .

Conclusion

All in all, this was an experience that was extremely frustrating to begin with, but soon turned out to be a fantastic learning experience.

Reactive Extensions and RxJS are extremely powerful tools, but there can be several nuances of working with Observables that can easily trip up the uninitiated. However, the code for the RxJS library is surprisingly simple to read and understand once you get past the initial disorientation that comes with reading any new code.

Give it a try. You won’t be disappointed; and you might just end up learning a lot more about RxJS than you could possibly learn by just reading their documentation.