RxJS: How to make an ordered mergeMap operator

An operator for when both concurrency control and ordering are important

Tamás Sallai

7 mins

An example is available on GistRun

The problem to solve

In one of my projects, I needed to assemble a PDF document where **different pages came from different sources and the result document had to contain all of them in order**. I used the pdf-lib for everything related to documents. With this library, I can create a new document, copy pages from another one then save the result as a buffer with some simple commands:

```
// create a document
const pdfDoc = await PDFDocument.create();

// source document
const pdf = // ...

// copy pages from pdf to pdfDoc
(await pdfDoc.copyPages(pdf, pdf.getPageIndices())).forEach((page) => pdfDoc.addPage(page));

// save the result document
await pdfDoc.save();
```

I had an **async function** that generated part of the result based on a config object. By calling this function with the different configs, I can generate the fragments, then use the `copyPages` + `addPage` to combine them:

```
const generatePages = async (pageConfig) => {
  // generate and return a PDF
  const pdf = ...;

  return await PDFDocument.load(pdf);
}

const pdfDoc = await PDFDocument.create();

// generate multiple pages
const pdf1 = await generatePages(pageConfig1);
const pdf2 = await generatePages(pageConfig2);
// ...

// copy all of the to pdfDoc
(await pdfDoc.copyPages(pdf1, pdf1.getPageIndices())).forEach((page) => pdfDoc.addPage(page));
(await pdfDoc.copyPages(pdf2, pdf2.getPageIndices())).forEach((page) => pdfDoc.addPage(page));
// ...

await pdfDoc.save();
```

The problem was that the `generatePages` ran slow and I wanted to parallelize it. The obvious solution is to use a `Promise.all` with an async `map` and run all the `generatePages` concurrently.

```
const pageConfigs = [...];

// generate all the input pages
const pdfs = await Promise.all(pageConfig.map(generatePages))

// build the result
const result = await pdfs.reduce(async (pdfDocProm, pdf) => {
  const pdfDoc = await pdfDocProm;
  (await pdfDoc.copyPages(pdf, pdf.getPageIndices())).forEach((page) => pdfDoc.addPage(page));
  return pdfDoc;
}, PDFDocument.create());

await result.save();
```

The above solution also takes advantage of an async `reduce` to build the result page in a serialized way.

Related

How to use async functions with Array.map in Javascript

Return Promises in map and wait for the results

How to use async functions with Array.reduce in Javascript

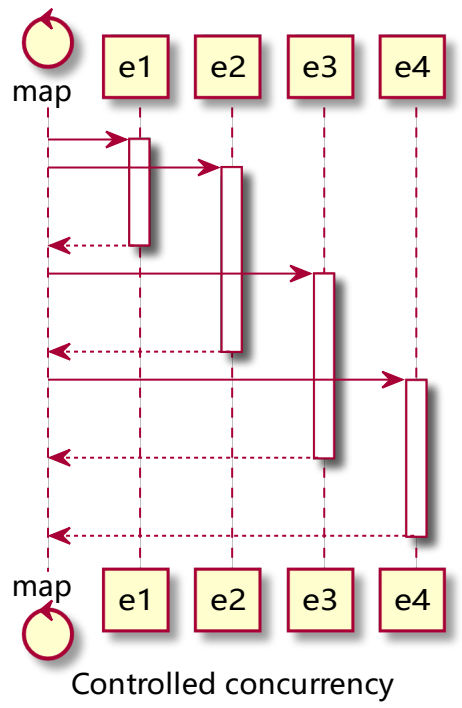
How to use Promises with reduce and how to choose between serial and parallel processing

There are two problems with this solution. First, **it runs all `generatePages` in parallel**. I used Puppeteer to generate the PDF from HTML and that is memory-heavy. In practice, I could run 3-5 at a time, any more would use up all the available memory and the whole process would be a lot slower due to swapping.

The second problem is that **it generates all the input PDFs before it starts to assembly the result document.** This added more memory pressure, so I was looking for a solution that has a controllable amount of concurrency and produces elements when they become ready and in order.

RxJS `mergeMap`

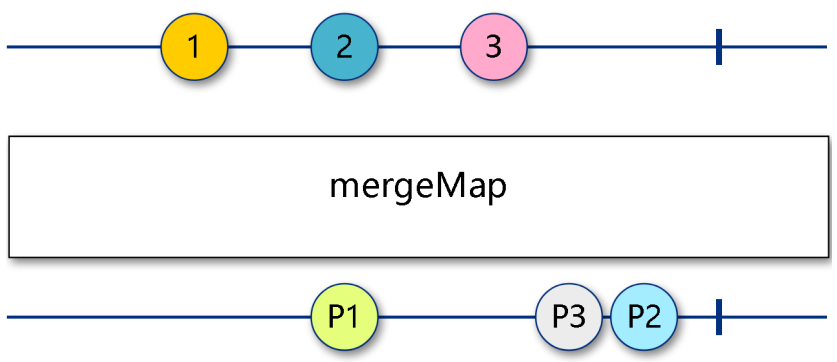
The `mergeMap`, also known as `flatMap`, is a natural choice that came into my mind for this. It **supports Promises and thus async functions nicely**, and it has **a third parameter that sets the number of parallel calls**. Whenever a task is finished, it starts a new one immediately, but never more than the concurrency cap.



This yields a nice structure:

```
// whops: not ordered

const result = await rxjs.from(pageConfigs).pipe(
  // generate pages, 3 at a time
  rxjs.operators.mergeMap(generatePages, 3),

  // build the result document as inputs become available
  rxjs.operators.reduce(async (pdfDocProm, pdf) => {
    const pdfDoc = await pdfDocProm;
    (await pdfDoc.copyPages(pdf, pdf.getPageIndices())).forEach((page) => pdfDoc.addPage(page));
    return pdfDoc;
  }, PDFDocument.create()),
).toPromise();
```

The above solution ticks the managed concurrency checkbox as well as it builds the result document when the inputs become available. But unfortunately, `mergeMap` emits elements *as they are ready* and **does not keep the original ordering of elements**.



This results in a document that has its pages out-of-order, which is wrong.

Another RxJS operator, `concatMap` preserves the ordering, but it has no support for setting the amount of concurrency. It seems like there is no built-in operator to support this use-case.

Fortunately, it's not hard to use existing operators to create new ones. The operator below is an `orderedMergeMap` that emits the elements in-order and also supports setting the concurrency.

orderedMergeMap

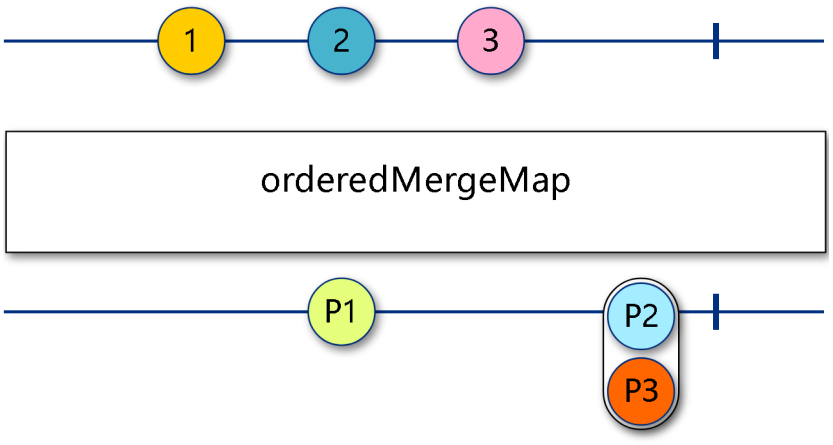
Here's the implementation that gets a Promise-producing mapper (an async function, usually) with a concurrency setting:

```
const orderedMergeMap = (mapper, concurrency) => rxjs.pipe(
  rxjs.operators.mergeMap(async (input, index) => [await mapper(input), index], concurrency),
  rxjs.operators.scan(({lastEmittedIndex, results}, [result, idx]) => {
    const emit = [...results, {result, index: idx}].map((result, _, list) => [
      result,
      [...Array(result.index - lastEmittedIndex - 1).keys()].map((i) => i + lastEmittedIndex + 1)
        .every((i) => list.find(({index}) => index === i) !== undefined)
    ]);

    return {
      emitting: emit
        .filter(([result, emit]) => emit)
        .map(([result]) => result)
        .sort((a, b) => a.index - b.index)
        .map(({result}) => result),
      lastEmittedIndex: Math.max(
        lastEmittedIndex,
        ...emit.filter(([result, emit]) => emit).map(([{index}]) => index)
      ),
      results: emit
        .filter(([result, emit]) => !emit)
        .map(([result]) => result),
    };
  }, {emitting: [], lastEmittedIndex: -1, results: []}),
  rxjs.operators.flatMap(({emitting}) => emitting)
)
```

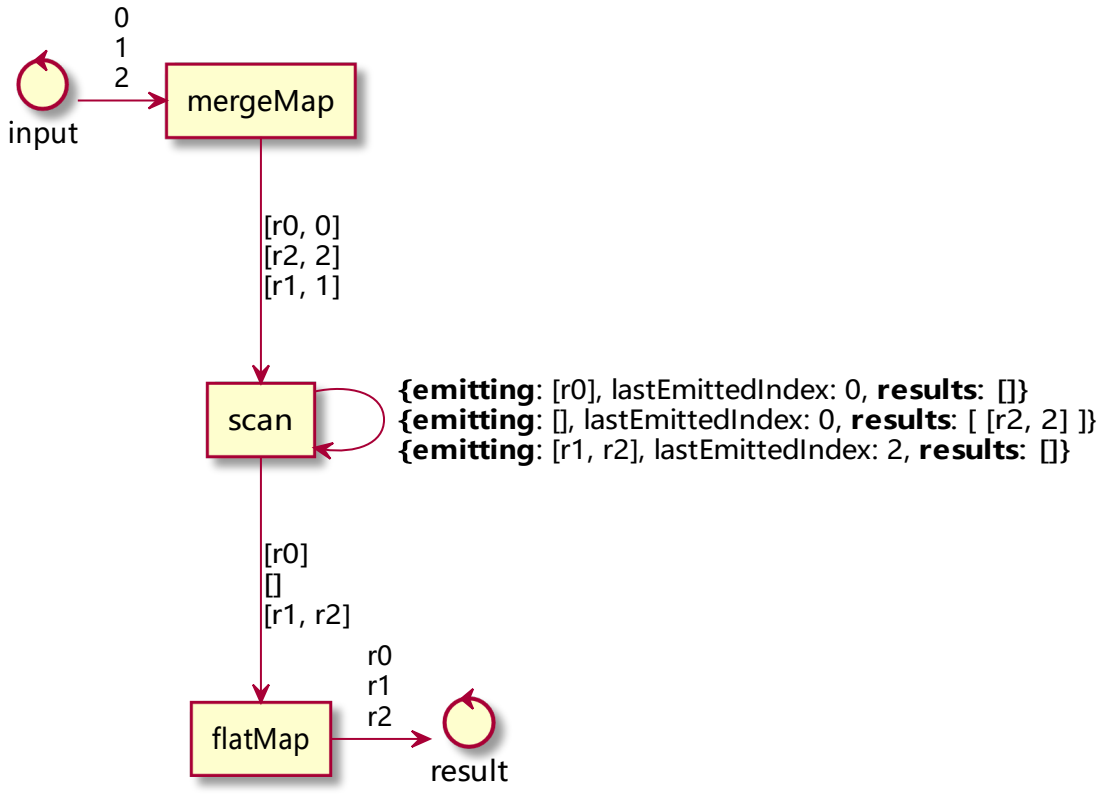
Since it's a pipeable operator, use it like any built-in one, passing the function and parallelization setting:

```
const result = await rxjs.from(pageConfigs).pipe(
  // generate pages, 3 at a time
  orderedMergeMap(generatePages, 3),

  // build the result document as inputs become available
  rxjs.operators.reduce(async (pdfDocProm, pdf) => {
    const pdfDoc = await pdfDocProm;
    (await pdfDoc.copyPages(pdf, pdf.getPageIndices())).forEach((page) => pdfDoc.addPage(page));
    return pdfDoc;
  }, PDFDocument.create()),
).toPromise();
```

In the above example, it feeds the `reduce` operator with the results in-order, no matter how long it takes for the `generatePages` to provide them.



How it works

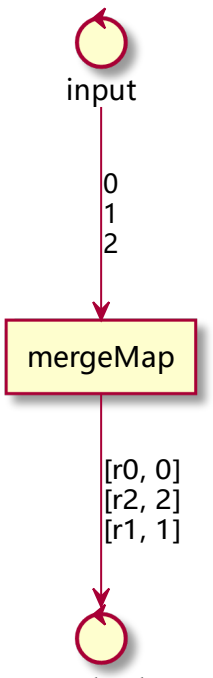


orderedMergeMap

Internally, it uses 3 operators.

```
rxjs.operators.mergeMap(async (input, index) => [await mapper(input), index], concurrency),
```

First, a `mergeMap` calls the mapper function and it controls the concurrency. It also adds the `index` to the result, so that later operators know the original ordering of the elements.

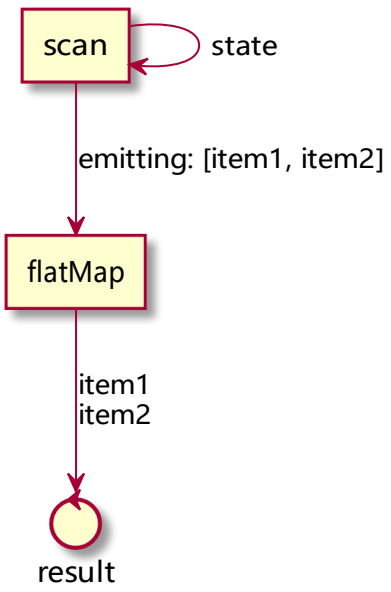


The mergeMap produces the result and adds the original index

Then a `scan` with a `flatMap` (=`mergeMap`) structure that allows storing intermediate values as the result of the `scan` and also emitting multiple items.

```
rxjs.operators.scan(({state}, item) => {
  // use stat

  return {
    emitting: [], // flatMap will emit these elements
    state: ..., // state for the next element
  }
}, {emitting: [], state: undefined}, 3),
  rxjs.operators.flatMap(({emitting}) => emitting)
```

This is a useful structure in many situations where you don’t want to use local variables. The result of the previous `scan` is fed back to the next invocation, and anything that it returns in the `emitting` array will be extracted and emitted by the `flatMap`.



A scan combined with a flatMap provides internal state

This structure moves all interesting logic inside the `scan` operator. It needs to keep track of the last element it emitted (the `lastEmittedIndex`) and all the results that are out-of-order (`results`). The latter is a buffer for elements that need to wait for a previous result before they can be emitted.

```
rxjs.operators.scan(({lastEmittedIndex, results}, [result, idx]) => {
  // mark elements as in-order or out-of-order
  const emit = [...results, {result, index: idx}].map((result, _, list) => [
    result,
    [...Array(result.index - lastEmittedIndex - 1).keys()].map((i) => i + lastEmittedIndex + 1)
      .every((i) => list.find(({index}) => index === i) !== undefined)
  ]);

  return {
    // emit in-order elements sorted by their index
    emitting: emit
      .filter(([result, emit]) => emit)
      .map(([result]) => result).sort((a, b) => a.index - b.index)
      .map(({result}) => result),
    // move the last index
    lastEmittedIndex: Math.max(
      lastEmittedIndex,
      ...emit.filter(([result, emit]) => emit).map(([{index}]) => index)
    ),
    // the buffer of out-of-order elements
    results: emit
      .filter(([result, emit]) => !emit)
      .map(([result]) => result),
  };
}, {emitting: [], lastEmittedIndex: -1, results: []}),
```

The implementation seems complicated, but it boils down to the `emit` array. It contains all results with a flag whether they can be emitted (all previous items are emitted) or not (waiting for a previous result). Then the `emitting` array in the result is the elements sorted by their original index that can be emitted, the `lastEmittedIndex` is the internal state to keep track of which elements were passed to the `flatMap`, and the `results` is the items that are still waiting.

Conclusion

The `orderedMergeMap` operator provides a way to control the concurrency of the `mergeMap` while also preserving the ordering of the elements. The implementation above uses only standard RxJS operators.

There are other solutions for this problem too, most using a local variable to store the out-of-order elements.

Javascript [68] RxJS [4]

01 January 2021

Tamás Sallai

Given a task that requires writing software, an expert provides better and more reliable solutions. I write articles and books to help you be that expert.

I came to believe that great software craftsmanship starts with understanding the underlying technologies better. You can't rely on "easy solutions" and "quick fixes" when you want dependable systems. I write about technology to deepen my knowledge and also to help others solve problems.

Check out the Books & Courses page for the more in-depth content I made.