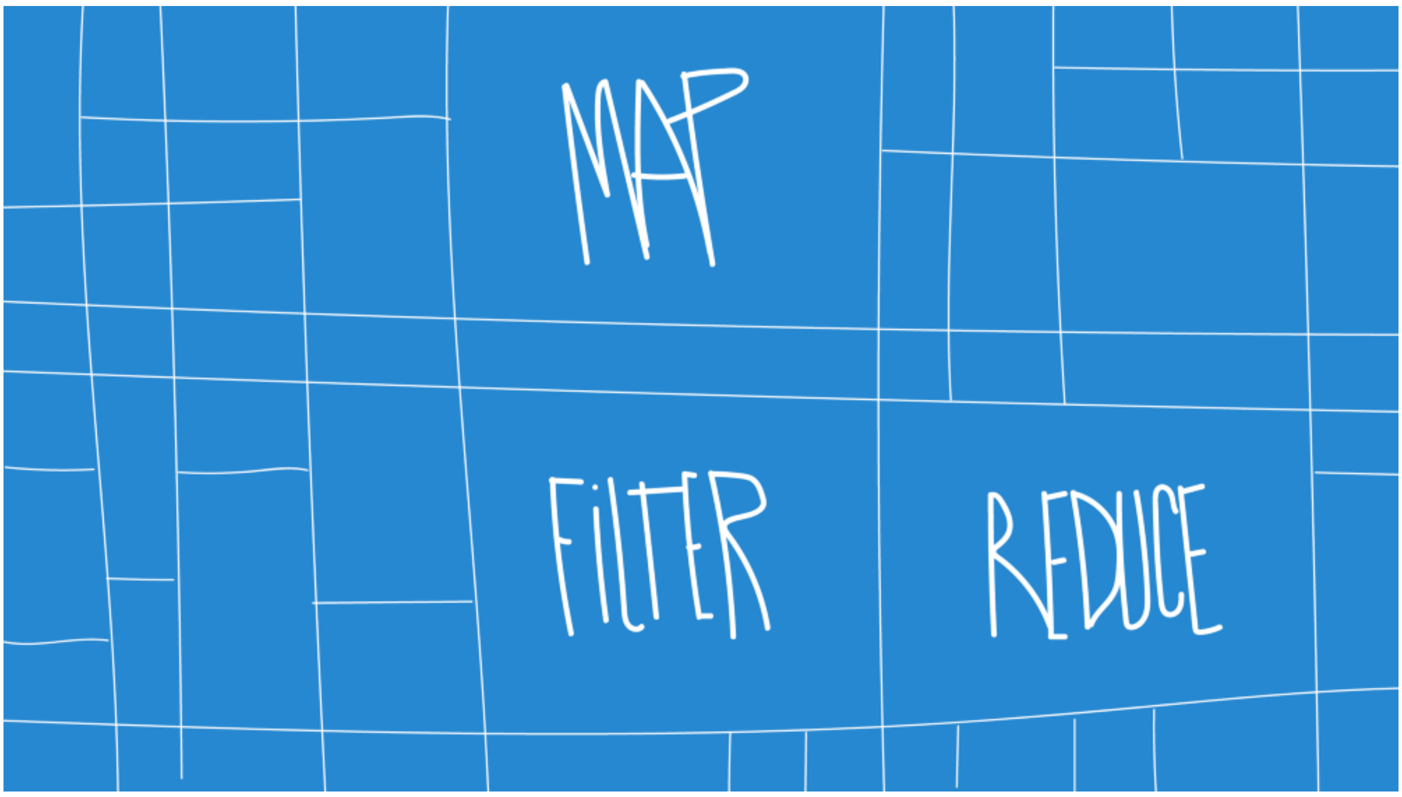
Currying and Composing your own versions of Reduce, Filter and Map

This is a follow up on my previous post, Reduce, Filter and Map without Reduce, Filter and Map in which I showed how to implement your own versions of the array functions. Here we'll alter those functions a bit so that we can make curried versions of them and later on compose them.

Gustaf Holm Follow
Aug 7, 2019 · 6 min read



So if you don't know how to implement reduce, map and filter or if you just wan't to take a look at how I did it. Head over to Filter and Map without Reduce, Filter and Map, read it, understand it, come back here 😊.

Reduce, Filter and Map without Reduce Filter and Map

Write your own Reduce, Filter and Map without using the built in reduce() filter() and map().

codeburst.io

So in the previous post I got a question on how to chain the self implemented versions of reduce, filter and map 🤔. I thought it was a good question and my way of combining the functions would be to compose them instead of chaining them.

✍*Also the way I've implemented them, as basic functions, there's no* `this` *keyword so chaining as the JavaScript built in versions wouldn't really be an option. I could rewrite all of it as a module pattern but prefer the basic functions. Another way is to override the built in Array methods but that's a BIG no no!*

So to compose these functions we need to make sure they all accept arrays as arguments and also returns arrays, so that the result from on function can be the input to the next one.

We also want to be able to "prep" our composable functions. For example if I want to filter out odd number from an array and then double the remaining numbers I would like it to look something like this when calling.

```
const double = x => x * 2
const even = x => x % 2 === 0;

const removeEvenAndDouble = compose(map(double), filter(even))
removeEvenAndDouble([1,2,3,4,5])
```

These composed functions create a flow where I would expect the returned value to be an array of `[4,8]` . So when I say "prep" functions I mean that I've already told map to use the `double` callback and filter to use the `even` callback. So when I call `removeEvenAndDouble` with the `[1,2,3,4,5]` array the array should be passed to the filter function, which runs the `even` function on each item, then returns the filtered array to the map function which runs the `double` function on each item.

So the array needs to be passed as the last argument to map and filter (the same goes for reduce). The arguments must also be passed one by one, so as long as the function doesn't have all the arguments it need it will just sit there and relax. In other words we need curried functions.

I was also asked how to debug when combining / chaining / composing the self implemented functions. You could just use the browser dev tools and set breakpoints like any other program. But if you want a more "console.log fashion" we can use a "trace" function, it takes a label, returns a function that takes a value (it's curried 😉), it prints the label and the value and returns the value. It looks like this.

```
const trace = label => value => {
  console.log(label, value)
  return value
}
```

Really simple! The trace function above is already a curried function as it takes one argument at the time and the "data" as it's last argument. We could take all arguments at once and then write another function to curry the trace function. That way we have both a regular and a curried version of the trace.

A Function to Curry Functions

This function might give you a headache 🤯, it's very compressed. Read it twice.

```
const curry = func =>
  (...args) =>
    (args.length < func.length)
    ? (...moreArgs) => curry(func)(...args, ...moreArgs)
    : func(...args);
```

Basically it takes a function, checks how many arguments that function needs to run. Compare the needed number of arguments to the arguments given. If it's enough it runs the function with the given arguments. If it's not enough arguments it returns a anonymous function that takes more arguments and then recalls the curry function with the old arguments plus the new ones and keeps doing that until it has all the arguments.

Here’s an example with the trace function.

```
const trace = (label, value => {
  console.log(label, value)
  return value
}

const curriedTrace = curry(trace)

trace("hello", "world") // logs "hello world"
curriedTrace("hello")("world") // logs "hello world"
```

As you’ve already figured out we’ll use this function to make curried versions of our reduce, filter and map from the previous post. But first we need to switch the arguments order.

New versions of Reduce, Map and Filter with switched arguments

Reduce

This is the same reduce as in the previous post, but with the array argument last. This also affects the internal argument order in map and filter (as we call reduce inside those functions) which we’ll handle below.

```
const reduce = (cb, initialValue, array) => {
  let result = initialValue;
  array.forEach(item =>
    result = cb.call(undefined, result, item, array));
  return result;
};
```

Map and Filter

Same goes here, array is the last argument. We’re also calling our reduce function with the array argument last.

```
const map = (func, array) =>
  reduce((result, item) =>
    result.concat(func(item)), [], array);
```

```
const filter = (func, array) =>
  reduce((result, item) =>
    func(item) ? result.concat(item) : result, [], array);
```

Make the curry

Now it’s dead simple to create curried versions of Reduce, Map and Filter, or any other function for that matter.

```
const curriedMap = curry(map);
const curriedFilter = curry(filter);
const curriedReduce = curry(reduce);
```

Composing

So now we got tracing to log the value with a label. We got the new versions of our functions with the array argument last and a function to curry it all! But we still need a function to compose our curried functions with. And as we saw in the beginning of this little text it should take some functions as argument, return another function which takes the initial value as it’s argument, which is passed to the first (actually the last, it runs from the bottom up) function and the result from the first function should be the input to the next and so on until we reach out last (first 😉) function. Then we get the result of the chain/compose flow. So here’s a implementation of the compose function.

```
const compose = (...funcs) =>
  initValue =>
    reduce(
     (funcResult, func) => func(funcResult),
     initValue,
     funcs.reverse()
   );
```

It takes functions as it’s argument and turns it into a array of function with the rest “...” operator. This returns a function which takes a initial value. Then it calls our implementation of reduce with a callback function, the initial value and the functions array, reversed as it’s how compose should work, bottom up, right to left, backwards.

Dummy Functions

We need some dummy functions to create a working example of composing our curried functions. Here they are.

```
const double = x => x * 2;
const even = x => x % 2 === 0;
const sum = (sum, x) => sum + x;
```

Put it together

Only thing left now is to Create the chain of functions, it runs from the bottom up, as “compose” functions should. Call the chain of function with a initial array.

```
const chain = compose(
 trace('after curried reduce'), // logs 12
  curriedReduce(sum, 0),        // array gets summed
  trace('after curried map'),   // logs [4,8]
  curriedMap(double),           // array gets doubled
  trace('after curried filter'),// logs [2,4]
  curriedFilter(even)           // array gets filtered
 );

chain([1,2,3,4,5]);
```

That’s it. Thanks for reading. Hope you learned something. Bye! 👋

JavaScript Functional Programming Programming