04 | 如何通过组合、管道和reducer让函数抽象化？

2022-09-27 石川 来自北京

《JavaScript进阶实战课》

课程介绍 >

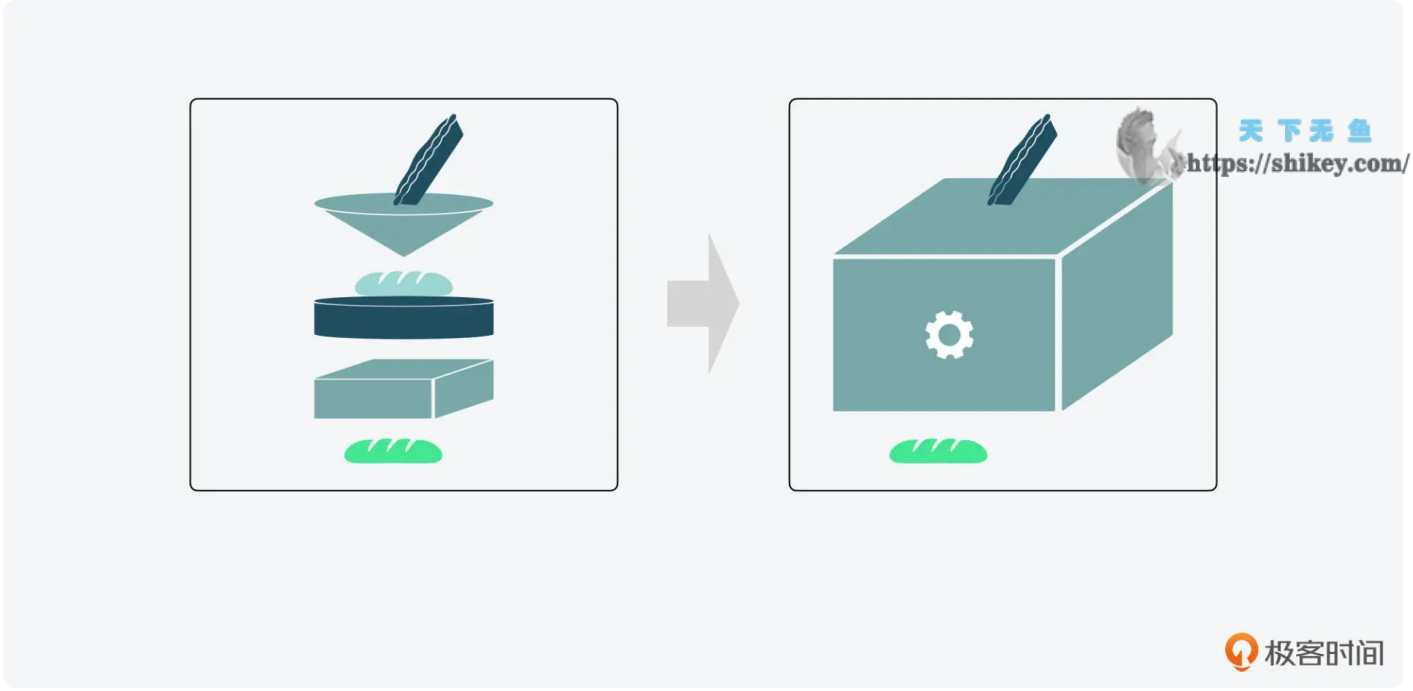
讲述：石川

时长 10:53　大小 9.94M

你好，我是石川。

上节课我们讲到，通过部分应用和柯里化，我们做到了从抽象到具象化。那么，今天我们要讲的组合和管道，就是反过来帮助我们把函数**从具象化变到抽象化**的过程。它相当于是系统化地把不同的组件函数，封装在了只有一个入口和出口的函数当中。

其实，我们在上节课讲处理函数输入问题的时候，在介绍 unary 的相关例子中，已经看到了组合的雏形。在函数式编程里，组合（Composition）的概念就是把组件函数组合起来，形成一个新的函数。



我们可以先来看个简单的组合函数例子，比如要创建一个“判断一个数是否为奇数”的 isOdd 函数，可以先写一个“计算目标数值除以 2 的余数”的函数，然后再写一个“看结果是不是等于 1”的函数。这样，isOdd 函数就是建立在两个组件函数的基础上。

```
1 var isOdd = compose(equalsToOne, remainderOfTwo);
```

复制代码

不过，你会看到这个组合的顺序是反直觉的，因为如果按照正常的顺序，应该是先把 remainderByTwo 放在前面来计算余数，然后再执行后面的 equalsToOne， 看结果是不是等于 1。

那么，这里为什么会有一个反直觉的设计呢？今天这节课，我们就通过回答这个问题，来看看组合和管道要如何做到抽象化，而 reducer 又是如何在一系列的操作中，提高针对值的处理性能的。

组合 Compose

在讲组合前，我想先带你来看看 Point-Free 和函数组件。这里，我们还是用刚刚提到的“判断一个值是不是奇数”的 isOdd 函数，来一步步看下它的实现。

Point-Free

那么首先，什么是 Point-Free 呢？实际上，Point-Pree 是函数式编程中的一种编程风格，其中的 Point 是指参数，free 是指没有。加在一起，Point-Free 的意思就是**没有参数的函数**。

而这样做的目的是什么呢？其实通过这种方式，就可以将一个函数和另外一个函数结合起来，形成一个新函数。比如，为了要创建 isOdd 函数，通过这种方式，我们就可以把这两个函数“组合”在一起，得到 isOdd。

复制代码

```
var isOdd = (x) => equalsToOne(remainderOfTwo(x));
```

函数组件

接着，我们再来看函数组件。

在以下的代码示例当中，我们先定义了两个函数：第一个是 dividedBy，它的作用是计算 x 除以 y 的余数；第二个是 equalsTo，它是用来看余数是否等于 1。

这两个函数其实就是我们用到的组件函数。你可以发现，这两个组件的特点都是努力专注做好一件小事。

复制代码

```
var dividedBy = (y) => {
    return function forX(x) {
        return x % y;
    }
}
var equalsTo = (y) => {
    return function forX(x) {
        return x === y;
    }
}
```

然后，在 dividedBy 和 equalsToOne 的基础上，我们就可以创建两个 Point-Free 的函数，remainderOfTwo 和 equalsToOne。

复制代码

```
var remainderOfTwo = dividedBy(2);
var equalsToOne = equalsTo(1);
```

最后，我们只需要传入参数 x，就可以计算相应的 isOdd 的结果了。


```
var isOdd = (x) => equalsToOne(remainderOfTwo(x));
```

好了，现在我们知道了，函数是可以通过写成组件来应用的。这里其实就是用到了函数式编程**声明式**的思想，equalsToOne 和 remainderByTwo，不仅把过程进行了封装，而且把参数也去掉了，暴露给使用者的就是**功能本身**。所以，我们只需要把这两个函数组件的功能结合起来，就可以实现 isOdd 函数了。

独立的组合函数

下面我们再来看看独立的组合函数。

其实从上面的例子里，我们已经看到了组合的影子。那么更进一步地，我们就可以把组合抽象成一个独立的函数，如下所示：

```
function compose(...fns) {
    return fns.reverse().reduce( function reducer(fn1,fn2){
        return function composed(...args){
            return fn2( fn1( ...args ) );
        };
    } );
}
```

也就是说，基于这里抽象出来的 compose 功能，我们可以把之前的组件函数组合起来。

```
var isOdd = compose(equalsToOne, remainderOfTwo);
```

所以，回到课程一开始提到的问题：为什么组合是反直觉的？因为它是按照**传参顺序**来排列的。

前面讲的这个组合，其实就是 `equalsToOne(remainderOfTwo(x))`。在数学中，组合写成 `fog`，意思就是一个函数接收一个参数 x，并返回成一个 `f(g(x))`。

好，不过看到这里，你可能还是觉得，即使自己理解了它的概念，但是仍然觉得它反直觉，因此想要一种更直观的顺序来完成一系列操作。这个也有相应的解决方案，那就是用函数式编程中的**管道**。

管道 Pipeline

函数式编程中的管道，是另外一种函数的创建方式。这样创建出来的函数的特点是：**一个函数的输出会作为下一个函数的输入，然后按顺序执行**。

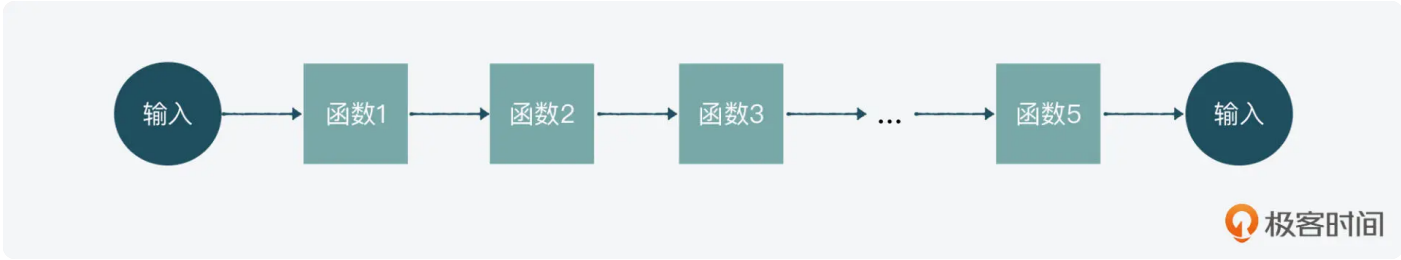
所以，管道就是以组合反过来的方式来处理的。

Unix/Linux 中的管道

其实管道的概念最早是源于 Unix/Linux，这个概念的创始人道格拉斯·麦克罗伊（Douglas McIlroy）在贝尔实验室的文章中，曾经提到过两个很重要的点：

- 一是让每个程序只专注做好一件事。如果有其它新的任务，那么应该重新构建，而不是通过添加新功能使旧程序复杂化。
- 二是让每个程序的输出，可以成为另一个程序的输入。

感兴趣的话你也可以读一下这篇杂志文章，虽然这是 1978 年的文章，但是它的设计思想到现在都不算过时。



好，那么现在，我们就来看一个简单的管道例子，在这个例子里，我们可以找到当前目录下面所有的 JavaScript 文件。

复制代码

```
$ ls -1 | grep "js$" | wc -l
```

你能发现，这个管道有竖线“|”隔开的三个部分。第一个部分 `ls -1`，列出并返回了当前目录下所有的文件，这个结果作为了第二步 `grep "js$"` 的输入；第二个部分会过滤出所有的以 `js` 结尾的文件；然后第二步的结果会作为第三部分的输入，在第三步，我们会看到最后计算的结果。

JavaScript 中的管道

回到 JavaScript 中，我们也可以用 isOdd 的例子，来看看同样的功能要如何通过管道来实现。

其实也很简单，我们只需要通过一个 reverseArgs 函数，将 compose 中接收参数的顺序反过来即可。

你可能会想到我们在上节课讲 unary 的时候，是把函数的输入参数减少到 1，而这里是**把参数做倒序处理，生成一个新的函数**。在函数式编程中，这算是一个比较经典的高阶函数的例子。

复制代码

```
function reverseArgs(fn) {
    return function argsReversed(...args){
        return fn( ...args.reverse() );
    };
}

var pipe = reverseArgs( compose );
```

然后我们可以测试下管道是否“畅通”。这次，我们把 remainderOfTwo 和 equalsToOne 按照比较直观的方式进行排序。

可以看到，isOdd(1) 返回的结果是 true，isOdd(2) 返回的结果是 false，和我们预期的结果是一样的。

复制代码

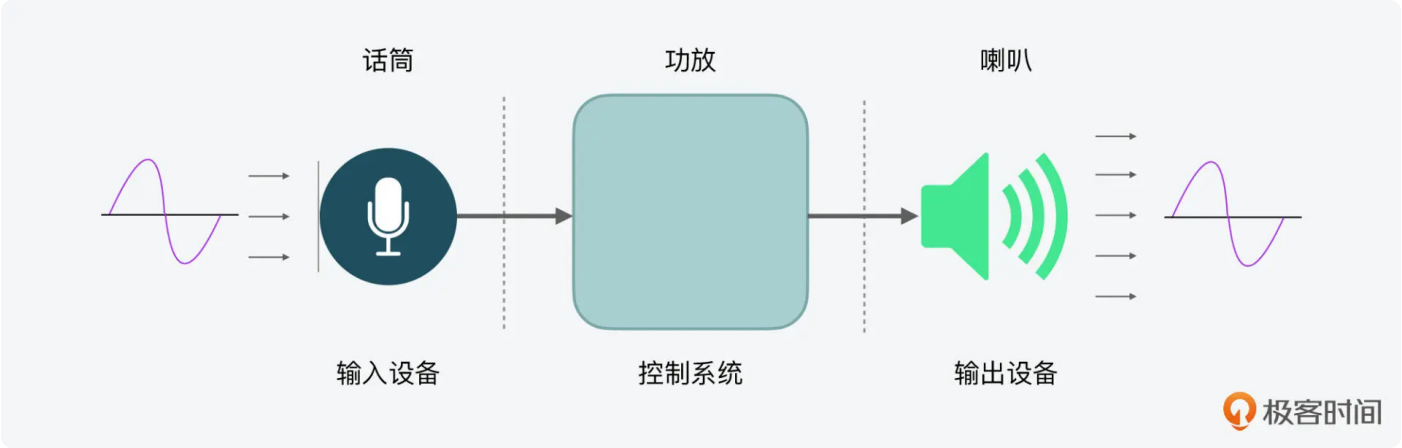
```
const isOdd = pipe(remainderOfTwo, equalsToOne);

isOdd(1); // 返回 true
isOdd(2); // 返回 false
```

Transduction

讲完了组合和管道之后，还有一个地方想再跟你强调下。

我一再说过，函数式编程中的很多概念，都来自于对复杂、动力系统研究与控制等领域。而通过组合和管道，我们可以再延伸来看一下转导（transducing）。

转导主要用于控制系统（Control System），比如声波作为输入，通过麦克风进入到一个功放，然后功放进行能量转换，最后通过喇叭传出声音的这样一个系统，就可以成为转导。



当然，单独看这个词，你或许并没有什么印象，但是如果说 React.js，你应该知道这是一个很著名的前端框架。在这里面的 reducer 的概念，就用到了 transducing。

在后面的课程中，我们讲到响应式编程和观察者模式的时候，还会更深入了解 reducer。这里，我们就先来看看 transduce 和 reducer 的作用以及原理。

那么，**reducer 是做什么用的呢？**它最主要的作用其实是解决在使用多个 map、filter、reduce 操作大型数组时，可能会发生的性能问题。

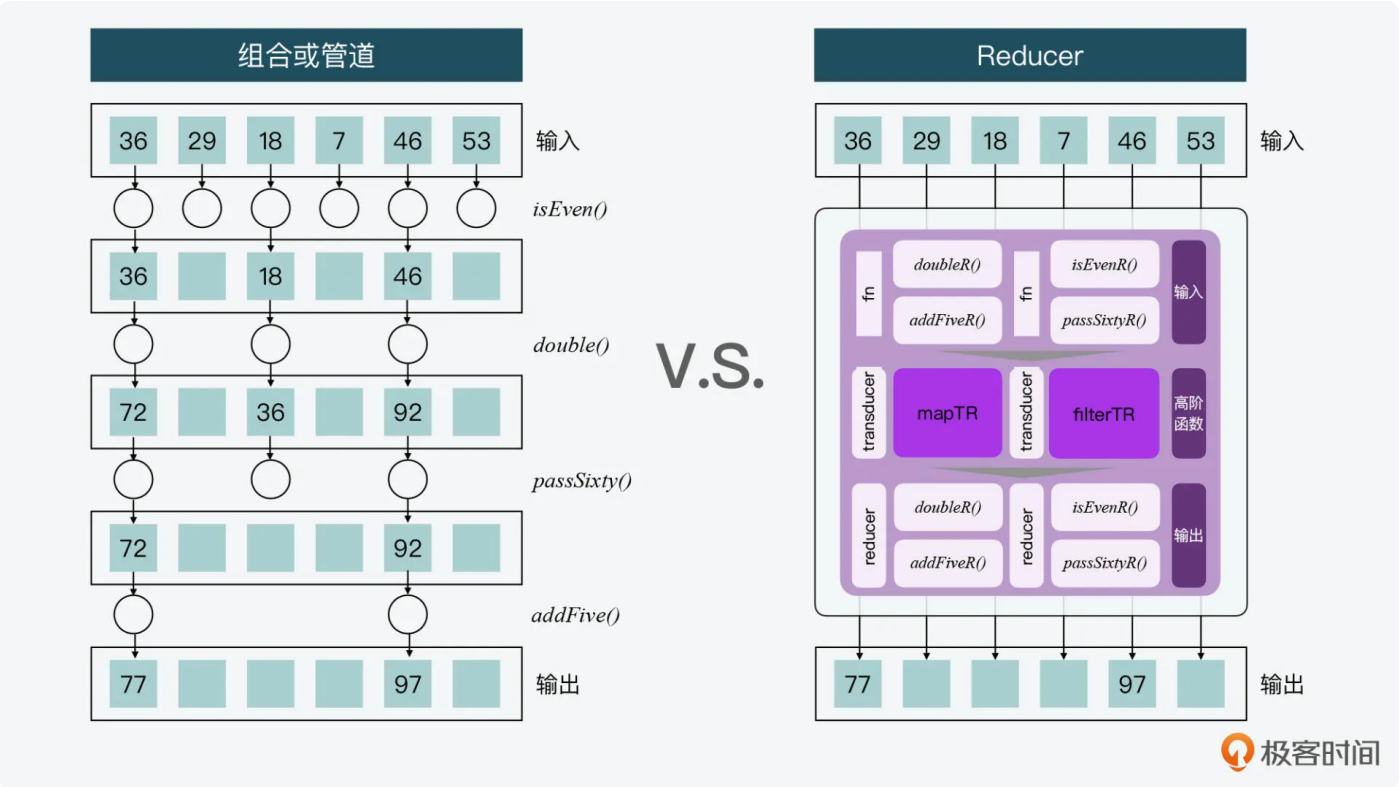
而通过使用 transducer 和 reducer，我们就可以优化一系列 map、filter、reduce 操作，使得输入数组只被处理一次并直接产生输出结果，而不需要创建任何中间数组。

可能我这么讲，你还是不太好理解，这里我们先来举一个不用 tansducer 或 reducer 例子吧。

```
var oldArray = [36, 29, 18, 7, 46, 53];
var newArray = oldArray
  .filter(isEven)
  .map(double)
  .filter(passSixty)
  .map(addFive);

console.log (newArray); // 返回：[77,97]
```


在这个例子里，我们对一组数组进行了一系列的操作，先是筛选出奇数，再乘以二，之后筛出大于六十的值，最后加上五。在这个过程中，会不断生成中间数组。

这个实际发生的过程如下图左半部分所示。



而如果使用 reducer 的话，我们对每个值只需要操作一次，就可产出最终的结果。如上图的右半部分所示。

那么它是如何实现的呢？在这里，我们是先将一个函数，比如 isEven 作为输入，放到了一个 transducer 里，然后作为输出，我们得到的是一个 isEvenR 的 reducer 函数。

是的，这里的 transducer 其实也是一个经典的高阶函数（即输入一个函数，得到一个新的函数）的例子！

实际上，像 **double 和 addFive 都具有映射类的功能**，所以我们可以通过一个类似 mapReducer 这样的一个 transducer，来把它们转换成 reducer。而像 **isEven 和 passSixty 都是筛选类的功能**，所以我们可以通过一个类似 filterReducer 这样的一个 transducer，来把它们转换成 reducer。

如果我们抽象化来看，其代码大致如下。它的具体实现这里我卖个关子，你可以先自己思考下，我们下节课再探讨。


```
var oldArray = [36, 29, 18, 7, 46, 53];

var newArray = composeReducer(oldArray, [
  filterTR(isEven),
  mapTR(double),
  filterTR(passSixty),
  mapTR(addfive),
]);

console.log (newArray); // 返回: [77,97]
```

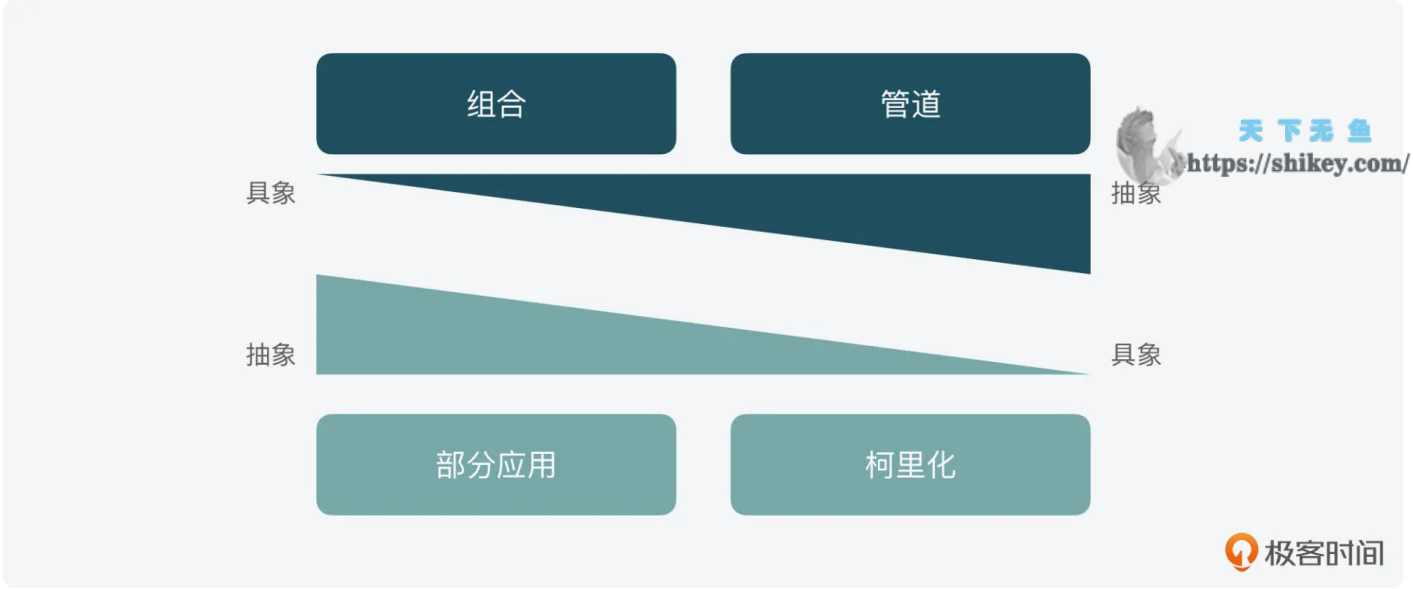
总而言之，从上面的例子中，我们可以看出来 composeReducer 用的就是一个类似组合的功能。

总结

这节课通过对组合和管道的了解，相信你可以看出来，它们和上节课我们讲到的部分应用和柯里化正好相反，一个是从具象走向抽象，一个是从抽象走向具象。

不过，虽然说它们的方向是相反的，但有一条原则是一致的，那就是**每个函数尽量有一个单一职责，只专注做好一件事**。

值得注意的是，这里的方向不同，并不是指我们要用抽象取代具象，或者是用具象取代抽象。而是说它们都是为了**单一职责函数**的原则，相辅相成地去具象化或抽象化。



另外，通过 reducer 的例子，我们也知道了如何通过 reducer 的组合，做到普通的组合达不到的**性能提升**。

在这节课里，我们是先从一个抽象层面理解了 reducer，不过你可能仍然对 map、filter、reduce 等概念和具体实现感到有些陌生。不用担心，下节课我就带你来进一步了解这一系列针对值的操作工具的机制，以及 functor 和 monad。

思考题

我们讲到 reduce 可以用来实现 map 和 filter ，那么你知道这背后的原理吗？欢迎在留言区分享你的答案，或者你如果对此并不十分了解，也希望你能找找资料，作为下节课的预习内容。

当然，你也可以在评论区交流下自己的疑问，我们一起讨论、共同进步。

上一篇　03 | 如何通过部分应用和柯里化让函数具象化？

精选留言 (7)

写留言

卡卡

2022-09-27 来自北京

我的理解是：reduce可以对原集合的每个元素使用map回调函数进行映射或者使用filter回调函数进行过滤，然后将新值放入新的集合

mapReduce的实现：

```
Array.prototype.mapReduce = function (cb, initValue) {
  return this.reduce(function (mappedArray, curValue, curIndex, array) {
    mappedArray[curIndex] = cb.call(initValue, curValue, curIndex, array);
    return mappedArray;
  }, []);
};
```

filterReduce的实现：

```
Array.prototype.filterReduce = function (cb, initValue) {
  return this.reduce(function (mappedArray, curValue, curIndex, array) {
    if (cb.call(initValue, curValue, curIndex, array)) {
      mappedArray.push(curValue);
    }
    return mappedArray;
  }, []);
};
```

作者回复: 是的，这里利用了reduce的第二个参数的初始值可以是一个“空数组”，映射或过滤后，放入“新数组”。

3

鐘

2022-09-27 来自北京

靜下心來重看一次, 好像看懂了, 以下是我對於 composeReducer 的實作：

```
const { filterTR, mapTR, composeReducer } = (() => {
        function applyTypeForFunction(fn, type) {
                fn.type = type;
```

```
        return fn;
    }

    function filterTR(fn) {
        return applyTypeForFunction(fn, "filter");
    }

    function mapTR(fn) {
        return applyTypeForFunction(fn, "map");
    }

    function composeReducer(inputArray, fnArray) {
        return inputArray.reduce((sum, element) => {
            let tmpVal = element;
            let tmpFn;

            for (let i = 0; i < fnArray.length; i++) {
                tmpFn = fnArray[i];
                if (tmpFn.type === "filter" && tmpFn(tmpVal) === false) {
                    console.log(`failed to pass filter: ${element} `);
                    return sum;
                }
                if (tmpFn.type === "map") {
                    tmpVal = tmpFn(tmpVal);
                }
            }

            console.log(`${element} pass, result = ${tmpVal}`);
            sum.push(tmpVal);

            return sum;
        }, []);
    }

    return {
        filterTR,
        mapTR,
        composeReducer
    };
})();
```

```
const isEven = (v) => v % 2 === 0;
const passSixty = (v) => v > 60;
const double = (v) => 2 * v;
const addFive = (v) => v + 5;

var oldArray = [36, 29, 18, 7, 46, 53];
var newArray = composeReducer(oldArray, [
        filterTR(isEven),
        mapTR(double),
        filterTR(passSixty),
        mapTR(addFive)
]);

console.log(newArray);
```


作者回复: 帥！！

1

深山何处钟

2022-09-30 来自北京

请问老师，compose那个函数，直接fns后不接reverse，是不是就是pipe的效果呢？

作者回复: 如果不用reverseArgs，pipe是可以简单理解成这样的：

var pipe = (...fns) => (x) => fns.reduce((v, f) => f(v), x);

I keep my ideals...

2022-09-28 来自北京

想请教一下老师compose组合的新函数里面如果有某一个是异步函数，或者没有返回值的情况下该怎么处理呢。还有多条件分支的情况下又该如何处理呢

作者回复: 1. 异步可以考虑结合CPS的promise/then，或 async/await来解决。

2. 没有返回值，可以考虑用Just和Nothing组成Maybe monad。

3. 多条件分支的情况下可以考虑在Maybe monad中创建orElse的方法。

灯火阑珊

2022-09-28 来自北京

我是从制造业转行的，对pipe和流水线有天然的接受度，上个工序的半成品就是下个工序的入参。

作者回复: 是的，说明生活中的例子无处不在。

天择

2022-09-27 来自北京

最近两篇文章的知识常在框架和库的代码里面见到，也会给我们阅读源码提供帮助。
具体和抽象都是为使用目标服务的，不管是柯里化还是函数组件，都是给使用者提供某种场景下的便利性，只不过有的需要具体的手段，有的需要抽象的手段。

作者回复: 嗯嗯，是这样的，无论具象还是抽象，目的都是学以致用

天择

2022-09-27 来自北京

point free的理解：把参数去掉，是指参数的含义已经体现在函数声明（名字）里面了，比如equalsToOne，那就是说传入的值是否等于1，如果是equalsToA，那么这个A就得传为参数，加上要比较的x就是两个参数了。这就是所谓“暴露给使用者的就是功能本身”。

作者回复: 是这样的