王熙： JavaScript函数式编程之Compose

由 Xi Wang 王熙创建, 最后修改于二月 13, 2018

函数式编程概览

JavaScript近些年从原先的浏览器领域，已经慢慢渗透到越来越多的方面，比如手机原生开发（React-Native, 微信小程序，MIUI直达服务）， 桌面客户端（Electron），服务器开发（NodeJS）等。随着需要解决问题的复杂度越来越高，更好的使用好JavaScript也越来越引起大家的关注。

同时， JavaScript作为一门支持多范式的的语言，天然的支持函数式编程。相比于已经广范流行的面向过程或者面向对象编程，个人认为函数式编程一个最大的优点是能让数据变动限制在有限的范围内（Limited Mutation）。软件的执行可以理解为数据流通过一个一个预设好的函数，每次流出一个函数产生一份新的数据（而不是对原有数据的修改），做为下一个函数的输入，如此依次，最终完成目标。从执行流程上来看，有点像工厂里的流水线，每一个环节的机器完成一个特定的任务。

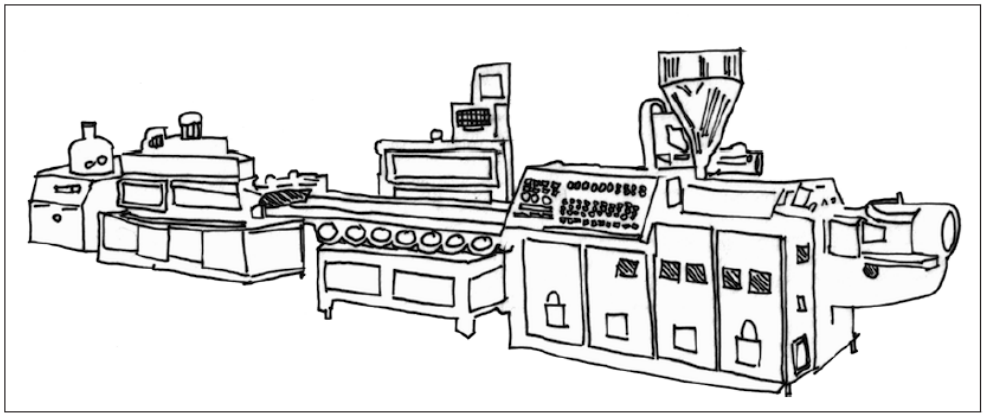


Figure 1-7. A functional program is a machine for transforming data

限制数据变动的好处有很多，这样代码的关注点可以只放在有限的范围内，一方面不会让自己的修改引起系统别的方面的变化，同时也不用担心自己在用的数据会被别的某个地方的代码修改引起自己意想不到的后结，从而大大提高软件系统的可靠性。 想更深入了解函数式编程的同学可以看一下这本书 https://drboolean.gitbooks.io/mostly-adequate-guide （VPN 自备）， 讲得比较通俗易懂。

正因为函数式编程能降低代码的复杂度，提高可维护性，近些来也越来越受到大家的推崇，很多框架也开始固化函数式编程的思想。

通过上面对函数式编程简单的介绍，大家大体上了解到函数式编程的基本流程是构建函数（产生各个处理环节的机器），编排好函数，然后让数据（原料）流过以产生结果。函数是这个流程中最重要的组成部分，如何更好的创建函数值得我们好好研究一下：除了可以全新定义一个函数，也可以基于已有的函数创建新的函数，从而提高开发效率，也达到更好的复用。

本文中我给大家介绍一种比较流行的基于已有函数构建新函数的方式， 一般业内称之为Compose。

Compose的数学基础

在数学里，大家都学过如果有函数

f(x)：x → y

g(x)：y → z

那么我们可以组合这两个函数，产生个新函数, 可以写成

g∘f(x) = g(f(x)) : x → z

同时，如果还有一个函数

h(z): z → a

我们可以在g∘f的基础上再组合上h

h∘(g∘f)(x) = h(g∘f(x)) = h(g(f(x))) : x → a

这里我们也可以先结合h∘g，达到的效果是相同的

(h∘g)∘f(x) = (h∘g)(f(x)) = h(g(f(x))) : x → a

所以可以用更抽象一点的表示方式：

(h∘g)∘f=h∘(g∘f)

由此可见，我们可以通过组合(Compose)得到新的函数。 同时在编排函数时，如果有需要，我们可以合理调整函数的组合顺序。

在JavaScript函数式编程中使用Compose

有了上面的理论基础，下面开始我们在JavaScript中实战

你可以会问，什么时候需要compose呢？ 简单来说，我们已经有一些零碎的功能函数，但要满足特定的数据处理的时候，我们需要把这些功能进行组合(compose), 这样才能完成目的。 很多函数库都支持compose这个操作，比如lodash，其实compose的实现也很简单，大体思路如下：

```
function compose(g, f) {
   return function(input) {
     return g(f(input))
   }
}
```

下面说一下怎么用这个compose函数，假设我们已经有两个函数

```
let add1 = function(input) {
    return input + 1
}
let multiple2 = function(input) {
    return input * 2
}
```

现在有输入数据 4， 先对它先加1再乘2， 那我们可以

```
let add1ThenMultiple2 = compose(multiple2, add1)
add1ThenMultiple2(4)
```

聪明的你估计又要问， 为啥要这样复杂，我可以直接如下调用

```
multiple2(add1(4))
```

不就可以了吗?

其实调用compose的话，主要有两点好处

1）用户可以给新生成的函数起一个更贴近业务的名字，可以让代码更易读，提高可维护性
2）新生成的函数 add1ThenMultiple2 可以在多个地方使用，甚至可以再被compose， 比如后面还有需要要减2的处理， 则可以 compose(minus2, add1ThenMultiple2)

在很多库里面，都可以看到compose的使用，比如在redux里面定义了compose函数

```
/*
 * @returns {Function} A function obtained by composing the argument functions
 * from right to left. For example, compose(f, g, h) is identical to doing
 * (...args) => f(g(h(...args)))
*/
export default function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }

  funcs = funcs.filter(func => typeof func === 'function')

  if (funcs.length === 1) {
    return funcs[0]
  }

  const last = funcs[funcs.length - 1]
  const rest = funcs.slice(0, -1)
  return (...args) => rest.reduceRight((composed, f) => f(composed), last(...args))
}
```

下面我们看看compose是如何被使用的。

Redux中有一个middleware的概念，是一种对已经有的dispatch能力加强的方式。 每个middleware函数大体

```
middlewere =  ({dispatch, getState}) => next => action => {
   ....
}
```

redux初始化时可以接受一组完成特定任务的middlewares, 在使用前对这组middlewares进行如下处理（注入对dispatch, getState的引用）

```
var middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
}
chain = middlewares.map(middleware => middleware(middlewareAPI))
```

经过如上处理， chain数组中每一个函数签名如下

```
next => action => {
   ....
}
```

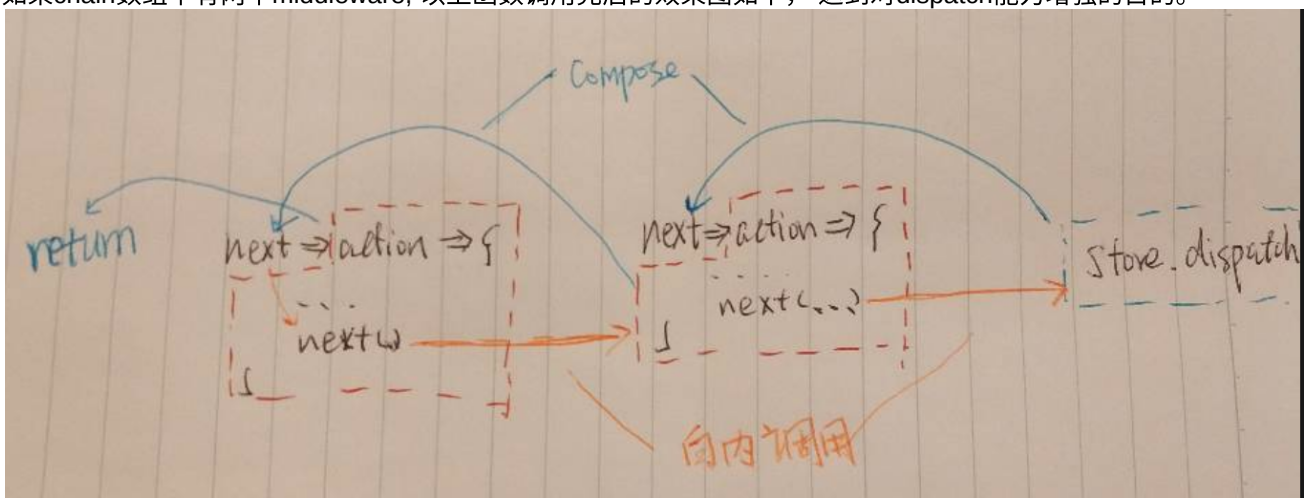
注意： 它接受一个next函数， 返回一个函数签名是 action => {...}，跟redux自带的dispatch的签名是完全一样的

最后就可以使用compose来依次组合上面chain数组中的函数了（让后面函数的返回做为前面函数的输入），同时马上调用compose生成的函数，传入的函数就是redux自带的dispatch

```
dispatch = compose(...chain)(store.dispatch)
```

compose的过程是在原有函数上依次包一层新功能，最终返回函数被调用时通过函数体中的next(...)调用内部封装的功能。

如果chain数组中有两个middleware, 以上函数调用完后的效果图如下， 达到对dispatch能力增强的目的。



最终返回的函数跟自带的dispatch的函数的签名是一样的 （action => {...}），从而在不影响用户调用方式的情况下，达到往原来方法中添加能力的目的。

```
const connectedComment = connect(commentSelector, commentActions)(CommentList);
```

其中connect会先产生一个HOC，然后可以再接受一个新的Component

这里大家会想为啥不直接把 CommentList直接放到connect的调用里作为第三个参数？

主要原因还是为了后面方便compose，因为函数一般只是一个输出（Component），如果connect一次一定要三个输入，它就很难在别的函数上进行compose了

第二个例子：

http://jlongster.com/Transducers.js--A-JavaScript-Library-for-Transformation-of-Data

```
function mapper(f) {
    return function(combine) {
        return function(result, x) {
//这里返回的签名可以被 1）处用
        //从而达到composition的目的
            return combine(result, f(x));
         }
    }
```

```
}
function filterer(f) {
    return function(combine) {
        return function(result, x) {
            return f(x) ? combine(result, x) : result; // 1)
         }
     }
}
// [1, 2, 3, 4]
arr.reduce(
 filterer(function(x) { return x > 2; })(
     mapper(function(x) { return x * 2; })(append)
 )
 ,
 []
);
```

Todo: 解释一下上面的代码

参考文献：

https://www.amazon.com/Functional-JavaScript-Introducing-Programming-Underscore-js/dp/1449360726

https://drboolean.gitbooks.io/mostly-adequate-guide

https://reactjs.org/docs/higher-order-components.html

无标签