link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=133 sentences=266, words=2142

1.函数式编程优势

- 更少的时间
- 更少的BUG
- 更好的测试性
- 更方便调试
- 适合并发执行
- 更高的复用性
- 支持tree-shaking
- React和Vue3大量使用函数式编程

2.什么是函数式编程

- 函数式编程是一种编程范式

```
let a=1;
let b=2;
let result = a+b;

class Sum{
    add(a,b){
        return a+b;
    }
}
let sum = new Sum();
sum.add(1,2);

function add(a,b){
    return a+b;
}
add(1,2);
```

3.First-class Function(头等函数)

- 函数是头等函数 (https://developer.mozilla.org/zh-CN/docs/Glossary/First-class_Function)
    - 函数可以赋值给变量
    - 函数可以作为参数
    - 函数可以作为返回值
- 作为参数和返回值的函数被称为高阶函数，高阶函数是函数式编程的基础

```
function add(a,b){
    return a+b;
}

let add1 = add;

function exec(fn,a,b){
    fn(a,b);
}

function exec(fn){
    return function(a,b){
        return fn(a,b);
    }
}
```

4.闭包(closure)

- 一个函数和对其周围状态的引用捆绑在一起,这样的组合就是闭包 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Closures)
- 闭包让你可以在一个内层函数中访问到其外层函数的作用域的变量

```
function init() {
    var name = "hello";
    function showName() {
        debugger
        console.log(name);
    }
    return showName;
}
let showName = init();
showName();
```

5. 纯函数

5.1 什么是纯函数

- 函数的返回结果只依赖于它的参数，相同的输入始终得到相同的输出
- 函数执行过程里面没有副作用(一个函数执行过程对产生了外部可观察的变化那么就可以说这个函数是有副作用)

```
function add(a,b){
    return a+b;
}

let c = 1;
let d =2;
function add2(a,b){
    d++;
    return a+b+c;
}
add2();
console.log(d);
```

5.2 优点

5.2.1 可缓存

- lodash.memoize (https://www.lodashjs.com/docs/lodash.memoize)
- 创建一个会缓存 func 结果的函数。 如果提供了 resolver ，就用 resolver 的返回值作为 key 缓存函数的结果。 默认情况下用第一个参数作为缓存的 key。 func 在调用时 this 会绑定在缓存函数上

```
cnpm i lodash -S
```

```
let _ = require('lodash');
const add = (a, b) => {
    console.log('add');
    return a + b;
}
const resolver = (...args)=>JSON.stringify(args)

var memoize = function (func,resolver) {
    let cache = {};
    let memoized =  (...args) => {
        const key = resolver?resolver(...args):JSON.stringify(args);
        if (typeof cache[key] !== 'undefined') {
            return cache[key];
        } else {
            cache[key] = func(...args);
            return cache[key];
        }
    }
    memoized.cache = cache;
    return memoized;
};

let memoizedAdd = memoize(add,resolver);
console.log(memoizedAdd(1,2));
console.log(memoizedAdd(1,2));
console.log(memoizedAdd.cache);
module.exports = memoizedAdd;
```

5.2.2 可测试

- jestjs (https://www.jestjs.cn/)

```
cnpm install jest --save-dev
```

5.test.js

```
const sum = require('./5.js');
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
  expect(sum(1, 2)).toBe(3);
});
```

```
{
  "scripts": {
    "test": "jest"
  }
}
```

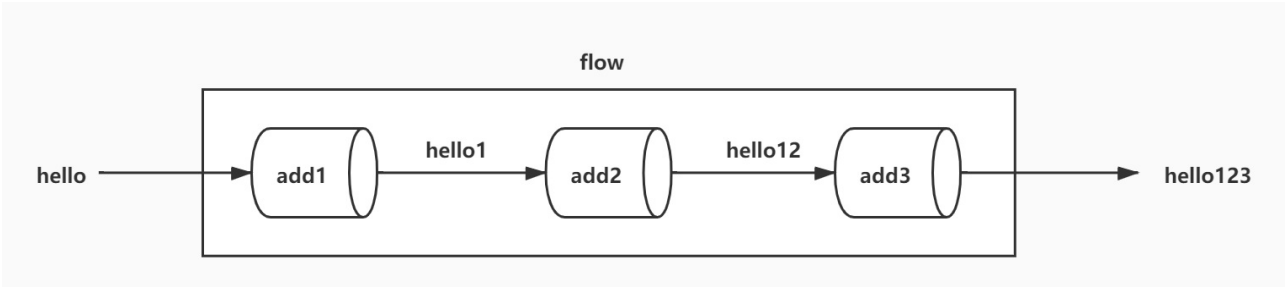
6. 柯里化

- lodash.curry (https://www.lodashjs.com/docs/lodash.curry)创建一个函数，该函数接收 func 的参数，要么调用func返回的结果，如果 func 所需参数已经提供，则直接返回 func 所执行的结果。或返回一个函数，接受余下的func 参数的函数，可以使用 func.length 强制需要累积的参数个数

```
let _ = require('lodash');
function add(a, b, c) {
    return a + b + c;
}
function curry(func) {
    let curried = (...args) => {
        if(args.length < func.length){
            return (...rest)=>curried(...args,...rest);
        }
        return func(...args);
    }
    return curried;
}
let curriedAdd = curry(add);
console.log(curriedAdd(1, 2, 3));
console.log(curriedAdd(1)(2, 3));
console.log(curriedAdd(1)(2)(3));
```

7. 组合

- flow (https://www.lodashjs.com/docs/lodash.flow)创建一个函数,每一个连续调用，传入的参数都是前一个函数返回的结果
- flowRight (https://www.lodashjs.com/docs/lodash.flowRight)类似 `flow`，除了它调用函数的顺序是从右往左的。
- redux compose (https://github.com/reduxjs/redux/blob/master/src/compose.ts)
- lodashjs (https://www.lodashjs.com/)是一个一致性、模块化、高性能的 JavaScript 实用工具库
- lodash/fp (https://github.com/lodash/lodash/tree/4.17.15-npm/fp)中的函数数据放在后后
- ramdajs (https://ramdajs.com/docs/)



手工组合

```
function add1(str) {
    return str + 1;
}
function add2(str) {
    return str + 2;
}
function add3(str) {
    return str + 3;
}

console.log(add3(add2(add1('hello'))));
```

flow

```

function add1(str) {
    return str + 1;
}
function add2(str) {
    return str + 2;
}
function add3(str) {
    return str + 3;
}

function flow(...fns) {
    if (fns.length == 0)
        return fns[0];
    return fns.reduceRight((a, b) => (...args) => a(b(...args)));
}
let flowed = flow(add3, add2, add1);
console.log(flowed('zhufeng'));
```

flowRight

```
let {flowRight} = require('lodash');
function add1(str) {
    return str + 1;
}
function add2(str) {
    return str + 2;
}
function add3(str) {
    return str + 3;
}
function flowRight(...fns) {
    if (fns.length == 0)
        return fns[0];
    return fns.reduce((a, b) => (...args) => a(b(...args)));
}
let flowed = flowRight(add3, add2, add1);
console.log(flowed('zhufeng'));
```

//带参数的函数组合

```
let {split,toUpper,join} = require('lodash');
let str = 'click button';

let r1 = split(str,' ');
console.log(r1);
let r2 = toUpper(r1);
console.log(r2);
let r3 = split(r2,',');
console.log(r3);
let r4 = join(r3,'_');
console.log(r4);
```

数据先放

```
let {split,toUpper,join,flowRight} = require('lodash');
let str = 'click button';

const func = flowRight(join('_'),split(','), toUpper, split(' '));
console.log(func(str));
```

数据后放

```
let {split,toUpper,join,flowRight} = require('lodash/fp');
let str = 'click button';

const func = flowRight(join('_'),split(','), toUpper, split(' '));
console.log(func(str));
```

过程跟踪

```
let {split,toUpper,join,flowRight} = require('lodash/fp');
let str = 'click button';
const logger = (name) => value => {
    console.log(name, value);
    return value;
}
const func = flowRight(join('_'),logger('afterSplit'),split(','),logger('afterToUpper'), toUpper, split(' '));
console.log(func(str));
```

8. Pointfree

- 把数据处理的过程先定义成一种与参数无关的合成运算就叫 Pointfree
- 先想怎么花钱

```
const { compose } = require("lodash/fp");
let num = 1;

function calcu(num){
    return num+1+2+3;
}

function add1(num){
    return num+1
}
function add2(num){
    return num+2
}
function add3(num){
    return num+3
}

let fn = compose(add3,add2,add1);
console.log(fn(1));
```

9. 函子

- 可以用来管理值和值的变化过程
- 把异常和异步操作等副作用控制在可控的范围之内

9.1 Context

- 如果一个对象内部持有一个值那么它就可以称为容器(Container)

```
class Container{
    constructor(value){
        this.value = value;
    }
}
```

9.2 Pointed Container

- 如果它有 of方法可称为有指向的容器

```
class PointedContainer{
    constructor(value){
        this.value = value;
    }
    static of (value){
        return new PointedContainer(value);
    }
}
```

9.3 Functor

- 如果它有 map方法可称为Functor(函子)
- 函子一会有一个静态的 of方法，用来生成实例
- 函子内部会保存一个值 value
- 函子提供 map方法，接入各种运算函数，从而引发值的变化

```
class Functor{
    constructor(value){
        this.value = value;
    }
    static of (value){
        return new Functor(value);
    }
    map(fn){
        return  new Functor(fn(this.value));
    }
}

let result = Functor.of('a')
.map(x=>x+1)
.map(x=>x+2)
.map(x=>x+3);
console.log(result.value);
```

9.4 Maybe

- 容器内部的值可能是一个空值,而外部函数未必有处理空值的机制，如果传入空值，很可能就会出错
- Maybe函子可以过滤空值,能过滤空值的函子被称为Maybe函子

```
class Maybe  {
    constructor(value){
        this.value = value;
    }
    static of (value){
        return new Maybe(value);
    }
    map(fn){
        return  this.value?new Maybe(fn(this.value)):this;
    }
}
Maybe.of(null).map(x=>x.toString())
```

9.5 Either

- Either 函子内部有两个值: 左值Left)和右值(Right),右值是正常情况下使用的值，左值是右值不存在时使用的默认值
- 常用来设置默认值和处理异常

```
class Either{
    constructor(left, right) {
        this.left = left;
        this.right = right;
    }
    static of = function (left, right) {
        return new Either(left, right);
    };
    map(fn) {
        return this.right ?
            Either.of(this.left, fn(this.right)) :
            Either.of(fn(this.left), this.right);
    }
    get value(){
        return this.right||this.left;
    }
}

let user = {gender:null};
let result = Either.of('男',user.gender)
.map(x=>x.toUpperCase())
console.log(result.value);

function parse(str){
    try{
        return Either.of(null,{data:JSON.parse(str)});
    }catch(error){
        return Either.of({error:error.message},null);
    }
}
console.log(parse("{}").value);
console.log(parse("{x}").value);
```

9.6 ap

- ap(applicative)的函子拥有ap方法
- ap方法可以让一个函子内的函数使用另一个函子的值进行计算
- ap方法的参数不是函数，而是另一个函子

```
class Ap{
    constructor(value){
        this.value = value;
    }
    static of (value){
        return new Ap(value);
    }
    map(fn){
        return  new Ap(fn(this.value));
    }
    ap(functor) {
        return Ap.of(this.value(functor.value));
    }
}

const A = Ap.of(x=>x+1);
const B = Ap.of(1)
let result = A.ap(B);
console.log(result);
```

9.7 Monad 函子

- 函子的值也可以是函子，这样会出现多层函子嵌套的情况
- Monad(单子[不可分割的实体]) 函子的作用是，总是返回一个单层的函子
- 它有一个 `flatMap`方法，与 `map`方法作用相同，唯一的区别是如果生成了一个嵌套函子，它会取出后者内部的值，保证返回的永远是一个单层的容器，不会出现嵌套的情况

函子嵌套

```
class Functor {
    constructor(value) {
        this.value = value;
    }
    static of(value) {
        return new Functor(value);
    }
    map(fn) {
        return new Functor(fn(this.value));
    }
}

let result = Functor.of('a')
   .map(x=>Functor.of(x+1))
   .map(x=>Functor.of(x.value+2))
   .map(x=>Functor.of(x.value+3))
console.log(result.value.value);
```

```
let r1 = [1,2,3].map(item=>[item+1]);
console.log(r1);
console.log(r1[0][0]);
let r2 = [1,2,3].flatMap(item=>[item+1]);
console.log(r2);
console.log(r2[0]);
```

```
class Monad {
    constructor(value) {
        this.value = value;
    }
    static of(value) {
        return new Monad(value);
    }
    map(fn) {
        return new Monad(fn(this.value));
    }
    join() {
        return this.value;
    }
    flatMap(fn){
        return this.map(fn).join();
    }
}

let result = Monad.of('a')
   .flatMap(x=>Monad.of(x+1))
   .flatMap(x=>Monad.of(x+2))
   .flatMap(x=>Monad.of(x+3))
console.log(result.value);
```

```
let r1 = Array.of(1,2,3).map(x=>x+1);
console.log(r1);
let r2 = Array.of(1,2,3).map(x=>[x+1]);
console.log(r2);
let r3 = Array.of(1,2,3).flatMap(x=>[x+1]);
console.log(r3);
```

9.8 IO函子与副作用

- 副作用就是程序和外部世界的交互，比如读取文件或调用接口
- 由于外部世界不可控，包含副作用的逻辑往往不要预测
- 函数式编程提倡把副作用分离出来，让没有副作用的纯逻辑们放在一起远离包含副作用的逻辑，这时就需要 `IO Monad`
- IO 就是 Input/Output，副作用无非是对外部世界的 Input(读)和 Output(写)
- IO 函子通过推迟执行的方式来实现对副作用的管理和隔离

过程调用

```
const { compose } = require("lodash/fp");
let localStorage = {
    getItem(key){
        if(key === 'data')
            return `{"code":0,"userId":"1"}`;
        else if(key === "1"){
            return `{"id":1,"name":"zhangsan","age":18}`;
        }
    }
}
 function printUsers() {
    const response = localStorage.getItem('data');
    const data = JSON.parse(response);
    const users = data.userId;
    const user = localStorage.getItem(data.userId);
    console.log(user);
}
printUsers();
```

IO函子

```
const { compose } = require("lodash/fp");
let localStorage = {
    getItem(key){
        if(key === 'data')
            return `{"code":0,"userId":"1"}`;
        else if(key === "1"){
            return `{"id":1,"name":"zhangsan","age":18}`;
        }
    }
}
class IO {
    constructor(value) {
        this.value = value;
    }
    map(fn) {
        return new IO(compose(fn, this.value));
    }
    join() {
        return this.value();
    }
    start(callback) {
        callback(this.value());
    }
}
const readByKey = key => new IO(() => localStorage.getItem(key));
const parseJSON = string => JSON.parse(string);
const writeToConsole = console.log;
readByKey('data')
    .map(parseJSON)
    .start(writeToConsole);
```

链式调用

```
const { compose } = require("lodash/fp");
let localStorage = {
    getItem(key){
        if(key === 'data')
            return `{"code":0,"userId":"1"}`;
        else if(key === "1"){
            return `{"id":1,"name":"zhangsan","age":18}`;
        }
    }
}
class IO {
    constructor(value) {
        this.value = value;
    }
    map(fn) {
        return new IO(compose(fn, this.value));
    }
    flatMap(fn) {
        return new IO(compose(x=>x.value(), fn, this.value));
    }
    start(callback) {
        callback(this.value());
    }
}
const readByKey = key => new IO(() => localStorage.getItem(key));
const parseJSON = string =>  JSON.parse(string)
const writeToConsole = console.log;
let ret = readByKey('data')
    .map(parseJSON)
    .map(item=>item.userId)
    .flatMap(readByKey)
    .map(parseJSON)
    .start(writeToConsole);;
```

9.9 task函子

- Task 函子通过类似 Promise 的 resolve 的风格来声明一个异步流程
- FP 中除了容器（Container），也可以用上下文（Context）来称呼包裹了一个值的结构
- Promise的任务是立刻执行的，而Task是在调用的时候才开始执行

异步执行任务

```
const Task = execute => ({
    execute
});
function get(url) {
    return Promise.resolve({ "code": 0, "userId": "1" });
}
const request = url => Task((resolve,reject) => get(url).then(resolve,reject));
request('data')
    .execute(user => console.log(user),error => console.error(error));
```

实现map

```
const Task = execute => ({
    execute,
    map: fn => Task(resolve => execute(x => resolve(fn(x))))
});
function get(url) {
    return Promise.resolve({ "code": 0, "userId": "1" });
}
const request = url => Task((resolve,reject) => get(url).then(resolve,reject));
request('data')
    .map(x => x.userId)
    .execute(user => console.log(user),error => console.error(error));
```

实现 flatMap

```
const Task = execute => ({
    map: fn => Task(resolve => execute(x => resolve(fn(x)))),
    flatMap: fn => Task(resolve => execute(x => fn(x).execute(resolve))),
    execute
});
function get(url) {
    if (url === 'data')
        return Promise.resolve({ "code": 0, "userId": "1" });
    else if (url === "1") {
        return Promise.resolve({ "id": 1, "name": "zhangsan", "age": 18 });
    }
}
const request = url => Task(resolve => get(url).then(resolve));
request('data')
    .map(x => x.userId)
    .flatMap(request)
    .map(x => x.name)
    .execute(user => console.log(user));
```

10.实际应用

10.1 react

- components (https://react.docschina.org/docs/components-and-props.html)
- redux compose (https://redux.js.org/api/compose)

10.2 vue

- function-api (https://github.com/vuejs/rfcs/blob/function-apis/active-rfcs/0000-function-api.md)
- vue2是将mounted，data，computed，watch之类的方法作为一个对象的属性进行导出。
- vue3新增了一个名为setup的入口函数，value, computed, watch, onMounted等方法都需要从外部import

```
<template>
  <div>
    <span>count is {{ count }}span>
    <span>plusOne is {{ plusOne }}span>
    <button @click="increment">count++button>
  div>
template>

<script>
import { value, computed, watch, onMounted } from 'vue'
export default {
  setup() {

    const count = value(0)

    const plusOne = computed(() => count.value + 1)

    const increment = () => { count.value++ }

    watch(() => count.value * 2, val => {
      console.log(`count * 2 is ${val}`)
    })

    onMounted(() => {
      console.log(`mounted`)
    })

    return {
      count,
      plusOne,
      increment
    }
  }
}
script>
```