一、为什么要学习函数式编程

函数式编程是非常古老的一个概念，早于第一台计算机的诞生，函数式编程的历史。

那我们为什么现在还要学函数式编程?

1、函数式编程是随着react的流行受到越来越多的关注

2、vue3也开始拥抱函数式编程

3、函数式编程可以抛弃this

4、打包过程中可以更好的利用tree shaking过滤无用代码

5、方便测试，方便并行处理

6、有很多库可以帮助我们进行函数是开发：lodash、underscore、ramda

二、函数式编程的概念

函数式编程（Functional Programming，FP），FP是编程范式之一，我们常听说的编程范式还有面向过程编程、面向对象编程。

面向对象编程的思维方式：把现实世界中的事物抽象成程序世界中的类和对象，通过封装、继承和多态来演示事物事件的联系。

函数式编程的思维方法：把现实世界的事物和事物之间的联系抽象到程序世界（对运算过程进行抽象）

程序的本质：根据输入通过某种运算获得相应的输出，程序开发过程中会涉及很多很有输入和输出的函数

x -> f（联系、映射）->y，y=f(x)

函数式编程中的函数指的不是程序中的函数(方法)，而是数学中的函数即映射关系，例如：y=sin(x)，x和y的关系

相同的输入始终要得到相同的输出(纯函数)

函数式编程用来描述数据(函数)之间的映射

```
1 //非函数式
2 let num1 = 2
3 let num2 = 3
```

```
let sum = num1+num2
console.log(sum)

//函数式
function add (n1, n2){
 return n1 + n2
}
let sum = add(2,3)
console.log(sum)
```

三、函数是一等公民

MDN First-class Function

- 函数可以存储在变量中
- 函数作为参数
- 函数作为返回值

在JavaScript中函数就是一个普通的对象（可以通过new Funciton()），我们可以把函数存储到变量/数组中，它还可以作为另一个函数的参数和返回值，甚至我们可以在程序运行的时候通过new Function('alert(1)')来构造一个新的函数。

把函数赋值给变量：

```
let fn = function() {
console.log('Hello First-class Function')
}
fn()

//一个示例
const BlogController = {
 index (posts) { return Views.index(posts) },
 show (post) { return Views.show(post) },
  create (attrs) { return Db.create(attrs) },
```

```
  update (post, attrs) { return Db.update(post, attrs) },
  destroy (post) { return Db.destroy(post) }
}

//优化
const BlogController = {
  index:Views.index,
  show:Views.show,
  create:Db.create,
  update:Db.update,
  destroy:Db.destroy
}
```

四、高阶函数

什么是高阶函数（higher-order function）

- 可以把函数作为参数传递给另一个函数
- 可以把函数作为另一个函数的返回结果

```
// 作为参数
function forEach(array, fn){
  for(let i = 0; i < array.length; i++){
  fn(array[i])
  }
}
let arr = [1, 2, 3, 4]
forEach(arr, function(item){
  console.log(item)
})


function filter(array, fn){
  let results = [];
  for (let i = 0; i < array.length; i++) {
```

```
 if (fn(array[i])) {
 results.push(array[i]))
 }
 }
 return results
}

let arr = [1, 2, 3, 4]
let r = filter(arr, function(item){
 return item % 2 === 0
})
cosnole.log(r)
```

```
// 作为返回值
function makeFn() {
 let msg = 'Hello function'
 return function() {
 console.log(msg)
 }
}
const fn = makeFn();
fn()

//调用内部返回的函数
makeFn()()

//模拟函数，只执行一次
function once(fn){ //传入一个函数fn
 let done = false;
 return function(){ // 返回一个函数，闭包？，不能修改done。
 if(!done){
 done = true;
 return fn.apply(this,arguments) //
 }
 }
```

```
}
let pay = once(function(money){
 console.log(`支付: ${money}RMB`)
})
pay(5) //直接执行once里的函数
pay(5) //二次执行done已经修改为true，无法执行第二次
//两个不一样的定义就可以执行多次
```

五、高阶函数的意义

- 抽象可以帮我们屏蔽细节，只需要关注与我们的目标
- 高阶函数是用来抽象通用的问题

六、常用的高阶函数

```
// 模拟常用高阶函数：map、every、some

//map
const map = (array,fn) => {
 let results = [];
 for (let value of array) {
 results.push(fn(value));
 }
 return results;
}
//测试
let arr = [1,2,3,4]
arr = map(arr,v => v*v);
console.log(arr)

//every
const every = (array, fn) => {
 let result = true;
 for (let value of array) {
```

```
 result = fn(value)
 if(!result){
 break;
 }
 }
 return result;
}
//测试
let arr = [11, 12, 9];
let r = every(arr,v => v > 10);
console.log(r)

//some
const some = (array, fn) => {
 let result = false;
 for (let value of array) {
 result = fn(value)
 if(result) {
 break;
 }
 return result;
 }
}
//测试
let arr = [1, 2, 3, 4]
let r = some(arr, v => v % 2 === 0);
console.log(r)
```

七、闭包

- 闭包（Closure）：函数和其周围的状态（词法环境）的引用捆绑在一起形成闭包
- 可以在另一个作用于中调用一个函数的内部函数并访问到该函数的作用域中的成员

- 闭包的本质：函数在执行的时候会放到一个执行栈上当函数执行完毕之后会从执行栈上移除，但是堆上的作用域成员因为被外部引用不能释放，因此内部函数依然可以访问外部函数的成员。

```
function makePower(power){
 return function(number){
 return Math.pow(number,power)
 }
}
//求平方
let power2 = markePower(2) //平方
let power3 = makePower(3) //3次方
console.log(power2(4))
 console.log(power2(5))
 console.log(power3(4))


function makeSalary(base){//传入基本工资
 return function(performance){//传入绩效工资
 return base + performance
 }
}
let salaryLevel1 = makeSalary(12000)
let salaryLevel2 = makeSalary(15000)
console.log(salaryLevel1(2000))
console.log(salaryLevel2(3000))
```

八、纯函数 pure functions

- **纯函数的概念：相同的输入永远会得到相同的输出**，而且没有任何可观察的副作用。
    - 纯函数就类似数学中的函数(用来描述输入和输出之间的关系)，y=f(x)

- **lodash**是一个纯函数的功能库，提供了对数组、数字、对象、字符串、函数等操作的一些方法
- 数组的slice和splice分别是：纯函数和不纯的函数
    - slice返回数组中的指定部分，不会改变原数组。
    - splice对数组进行操作返回该数组，会改变原数组。

```
let array = [1, 2, 3, 4, 5];
//纯函数
console.log(array.slice(0, 3))
console.log(array.slice(0, 3))
console.log(array.slice(0, 3))

//不纯的函数
console.log(array.splice(0, 3))
console.log(array.splice(0, 3))
 console.log(array.splice(0, 3))

 //纯函数
 function getSum(n1, n2){
  return n1 + n2;
 }
 console.log(getSum(1, 2))
 console.log(getSum(1, 2))
 console.log(getSum(1, 2))
```

- 函数式编程不会保留计算中间的结果，所以变量是不可变的（无状态的）
- 我们可以把一个函数的执行结果交给另一个函数去处理
- 纯函数的好处：
    - 可缓存

- 因为纯函数对相同的输入始终有相同的结果，所有可以把纯函数的结果缓存起来。

```
//记忆函数
const _ = require('lodash');
function getArea(r) {
 return Math.PI * r * r;
}
let getAreaWithMemory = _.memoize(getArea);
console.log(getAreaWithMemory(4))

//模拟memoize方法的实现
function memoize(fn){
 let cache = {}
 return function(){
 let key = JSON.stringify(arguments)
 cache[key] = cache[key] || fn.apply(fn, arguments)
 return cache[key]
 }
}
let getAreaWithMemory = memoize(getArea);
console.log(getAreaWithMemory(4))
console.log(getAreaWithMemory(4))
console.log(getAreaWithMemory(4))
```

- 可测试：
  - 纯函数让测试更方便
- 并行处理
  - 在多线程环境下并行操作共享的内存数据很可能会出现意外情况

- 纯函数不需要访问共享的内存数据，所以在并行环境下可以任意运行纯函数(Web Worker)

- 副作用：
  - 纯函数：对于相同的输入永远会得到相同的输出，而且没有任何可观察的副作用
  - 副作用让一个函数变的不纯，纯函数的根据相同的要入返回相同的输出，如果函数依赖于外部的状态就无法保证输出相同，就会带来副作用。
  - 副作用来源：
    - 配置文件
    - 数据库
    - 获取用户的输入
    - ……

所有的外部交互都有可能代理副作用，副作用也使得方法通用性下降不适合扩展和可重用性，同时副作用会给程序中带来安全隐患给程序带来不确定性，但是副作用不可能完全禁止，尽可能控制它们在可控范围内发生。

```
//不纯的
let mini = 18;
function checkAge(age){
 return age >= mini
}

//纯的(有硬编码，后续可以通过柯里化解决)
function checkAge(age){
 let mini = 18;
 return age >= mini
}
```

九、Lodash

```
//演示lodash
//first / last / toUpper / reverse / each / includes / find / findIndex
const _ = require('lodash');
const array = ['jack', 'tom', 'lucy', 'kate'];
console.log(_.first(array))
console.log(_.last(array))
console.log(_.toUpper(array))
console.log(_.reverse(array))
const r = _.each(array,(item, idx) => {
    console.log(item, idx)
})
console.log(r)
```

十、柯里化（Haskell Brooks Curry）

使用柯里化解决第八章案例中硬编码的问题

```
function checkAge(age){
 let min = 18
 return age >= min
}

//普通纯函数
function checkAge(min,age){
 return age >= min
}
checkAge(18,24)
checkAge(18,20)
checkAge(20,30)
//18这个数字重复了

//柯里化
function checkAge(min) {
```

```
 return function(age) {
 return age >= min
 }
}
let checkAge18 = checkAge(18);
let checkAge20 = checkAge(20);
console.log(checkAge18(20));
console.log(checkAge20(24));

//ES6写法
let checkAge = min => (age => age >= min);
```

柯里化（currying）：

- 当一个函数有多个参数的时候先传递一部分参数调用它（这部分参数以后永远不变）
- 返回一个新的函数接收剩余的参数，返回结果。

十一、Lodash中的柯里化

- _.curry（func）
  - 功能：创建一个函数，该函数接收一个或多个func的参数，如果func所需要的参数都被提供则执行func并返回执行的结果。否则继续返回该函数并等待接收剩余的参数。
  - 参数：需要柯里化的函数
  - 返回值：柯里化后的函数

```
//lodash中的curry基本使用
const _ = require('lodash');

function getSum(a, b ,c){
 return a + b + c;
}

```

```
const curried = _.curry(getSum);
console.log(curried(1, 2, 3))

 console.log(curried(1)(2, 3))

 console.log(curried(1, 2)(3))
```

十二、柯里化案例

```
//柯里化案例
''.match(/\s+/g)
''.match(/\d+/g)

const _ = require('lodash');

const match = _.curry(function(reg, str){
 return str.match(reg)
})

 const haveSpace = match(/\s+/g);
 const haveNumber = match(/\d+/g)
 console.log(haveSpace('hello world'))
 console.log(haveNumber('abc'))

 const filter = _.curry(function(func, array)){
  return array.filter(func)
 }
 console.log(filter(haveSpace, ['John Conor', 'John_Donne']))

 const findSpace = filter(havaSpace);
 console.log(findSpace(['John Conor', 'John_Donne']))
```

十三、柯里化实现原理

```
function curry (func){
 return function curriedFn(...args){
 //判断实参和形参的个数
 if(args.length < func.length){
 return function(){
 return curriedFn(...args.concat(Array.from(arguments)))
 //Array.from()方法就是将一个类数组对象或者可遍历对象转换成一个真正的数组。
 }
 }
  return func(...args)
  }
}
```

十四、柯里化总结

- 柯里化可以让我们给一个函数传递较少的参数得到一个已经记住了某些固定参数的新函数
- 这是一种对函数参数的‘缓存’
- 让函数变的更灵活，让函数的粒度更小
- 可以把多元函数转成一元函数，可以组合使用函数产生强大的功能

十五、函数组合

- 函数组合（compose）：如果一个函数要经过多个函数处理才能得到最终值，这个时候可以把中间过程的函数合并成一个函数
    - 函数就像是数据的管道，函数组合就是把这些管道连接起来，让数据穿过多个管道形成最终结果
    - 函数组合默认是从右到左执行。

```
// 函数组合演示
function compose(f, g){
 return function(value){
 return f(g(value))
```

```
 }
}

function reverse(array){
 return array.reverse()
}

function first(array){
 return array[0]
}

const last = compose(first, reverse)
console.log(last([1, 2, 3, 4])) //4
```

十六、Lodash中的组合函数

- lodash中组合函数flow()或者flowRight()，她们都可以组合多个函数
- flow()是从左到右
- flowRight()是从右到左运行，使用的更多一些

```
//lodash中的函数组合的方法 _.flowRight()
const _ = require('lodash');

const reverse = arr => arr.reverse();
const first = arr => arr[0];
const toUpper = s => s.toUpperCase()
const f = _.flowRight(toUpper,first,reverse)

console.log(f(['one', 'two', 'three'])) //THREE
```

十七、组合函数原理模拟

```
//模拟lodash中的flowRight

function compose(...args){
```

```
 return function(value){
 return args.reverse().reduce(function(acc, fn){
 //此处的fn就是args的其中一个，acc就是上一个函数的运算的结果
 return fn(acc)
 },value)
 }
}

const reverse = arr => arr.reverse();
const first = arr => arr[0];
const toUpper = s => s.toUpperCase()
const f = compose(toUpper,first,reverse)

console.log(f(['one', 'two', 'three'])) //THREE


const compose = (...args) => value => args.reverse().reduce((acc, fn) => fn(acc),value);

```

十八、函数组合-结合律

- 函数的组合要满足结合律（associativity）
  - 我们既可以把g和h组合，还可以把f和g组合，结果都是一样的。

```
//结合律（associativity）
let f = compose(f, g ,h)
let associative = compose(compose(f, g), h) == compose(f, compose(g, h))
//true

//函数组合要满足结合律
const _ = require('lodash');

```

```
//const f = _.flowRight(_.toupper, _.first, _.reverse)
 const f = _.flowRight(_.toupper, _.flowRight(_.first, _.reverse));
 console.log(f(['one', 'two', 'three']))
```

十九、函数组合-如何调试

```
//const f = _.flowRight(_.toupper, _.first, _.reverse)
//console.log(f(['one', 'two', 'three']))

//函数组合，调试
//NEVER SAY DIE --> never-say-die
const _ = require('lodash');

const trace = _.curry((tsg, v) => {
 console.log(tag, v)
})

//_.split()
const split = _.curry((sep, str) => _.split(str, sep));
const join = _.cuury((sep, array) => _.join(array, str));
const map = _.curry((fn, array) => _.map(array, fn));
const log = v => {
 console.log(v);
 //['NEVER', 'SAY', 'DIE'] 第一次调试
 // never,say,die 第二次次调试，此处应该返回的是一个数组，不是字符串
 return v;
}
const f = _.flowRight(join('-'), _.toLower, log, split(' '))//第一次调试
console.log(f('NEVER SAY DIE')) //n-e-v-e-r,s-a-y,d-i-e

const f = _.flowRight(join('-'), log, _.toLower, split(' '))//第二次次调试

```

```
27 const f = _.flowRight(join('-'), trace('map 之后') _.map(_.toLow
er), split(' '))
28 console.log(f('NEVER SAY DIE')) //never-say-die map之后
```

二十、Lodash中的FP模块

- lodash中的fp模块提供了实用对**函数式编程友好**的方法
- 提供了不可变**auto-curried iteratee-first data-last**的方法

```
1 //lodash模块
2 const _ =require('lodash');
3
4 _.map(['a', 'b', 'c'], _.toUpper)
5 // => ['A', 'B', 'C']
6 _.map(['a', 'b', 'c'])
7 // => ['a', 'b', 'c']
8
9 _.split('Hello World', ' ');
10
11 //lodash/fp 模块
12 const fp = require('lodash/fp');
13
14 fp.map(fp.toUpper,['a', 'b', 'c']);
15 fp.map(fp.toUpper)(['a', 'b', 'c']);
16
17 fp.split(' ','Hello World');
18 fp.split(' ')('Hello World');
19
20 //lodash/fp 模块
21 //NEVER SAY DIE --> never-say-die
22 const fp = require('lodash/fp');
23
24 const f = fp.flowRight(fp.join('-'), fp.map(fp.toLower), fp.spl
it(' '));
25 console.log(f('NEVER SAY DIE'))
```

二十一、Lodash中的FP模块-map方法的区别

```
//lodash 和lodash/fp模块中的map方法的区别
const _ = require('lodash');

console.log(_.map(['23', '8', '10'], parseInt)); // [23, NaN, 2]
//parseInt('23', 0, array)
//parseInt('8', 1, array)
//parseInt('10', 2, array)

const fp = require('lodash/fp');

 console.log(fp.map(parseInt, ['23', '8', '10'])); //[23, 8, 10]
 //fp中的map方法中的方法只能接收一个参数
```

二十二、PointFree

PointFree：我们可以把数据处理的过程定义成与数据无关的合成运算，不需要用到代表数据的那个参数，只要把简单的运算步骤合成到一起，在使用这种模式之前我们需要定义一些辅助的基本运算函数。

- 不需要指明处理的数据
- 只需要合成运算过程
- 需要定义一些辅助的基本运算函数

```
const f = fp.flowRight(fp.join('-'), fp.map(_.toLower),
fp.split(' '))
```

```
// 非Point Free 模式
//Hello World => hello_world
function f(world){
 return word.toLowerCase().replace(/\s+/g, '_');
}

//Point Free
const fp = require('lodash/fp')
```

```
const f = fp.flowRight(fp.replace(/\s+/g, '_'), fp.toLower);
console.log(f('Hello World'))
```

二十三、Point Free案例

```
// 把一个字符串中的首字母提取并转换成大写，使用. 作为分隔符
// world wild web => W. W. W
const fp = require('lodash/fp');

//const firstLetterToUpper = fp.flowRight(fp.join('. '), fp.map(fp.first), fp.map(fp.toUpper), fp.split(' '))

const firstLetterToUpper = fp.flowRight(fp.join('. '), fp.map(fp.flowRight(fp.first, fp.toUpper)), fp.split(' '))

console.log(firstLetterToUpper('world wild web'))
```

二十四、Functor函子

为什么要学函子

都目前为止已经学习了函数式编程的一些基础，但是我们还没有演示在函数式编程中如何把副作用控制在可控的范围内、异常处理、异步操作等。

什么是Functor

- 容器：包含值和值的变形关系（这个变形关系就是函数的）
- 函子：是一个特殊的容器，通过一个普通的对象来实现，该对象具有map方法，map方法可以运行一个函数对值进行处理（变形关系）

```
class Container {
 static of (value) {
 return new Container(value)
 }

 constructor (value) {
```

```
  this._value = value
  }

   map (fn) {
   return new Container(fn(this._value))
   }
  }

  //let r = new Container(5)
  let r = Container.of(5)
   .map(x => x + 1)
   .map(x => x * x)
  console.log(r)
```

二十五、Functor函子-总结

- 函数式编程的运算不直接操作值，而是由函子完成
- 函子就是一个实现了map契约的对象
- 我们可以把函子想象成一个盒子，这个盒子里封装了一个值
- 想要处理盒子中的值，我们需要给盒子的map方法传递一个处理值的函数(纯函数)，由这个函数来对值进行处理
- 最终map方法返回一个包含新值的盒子（函子）

```
// 演示null undefined 的问题
Container.of(null)
 .map(x => x.toUpperCase()) //报错
```

二十六、MayBe函子

- 我们在编程的过程中可能会遇到很多错误，需要对这些错误做相应的处理
- MayBe函子的作用就是可以对外部的空值情况做处理（控制副作用在允许的范围）

```
// MayBe 函子
```

```
class MayBe {
 static of (value) {
 return new MayBe(value)
 }

 constructor (value) {
 this._value = value;
 }

  map (fn) {
 return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this._v
alue))
 }

 isNothing () {
 return this._value === null || this._value === undefined
 }
}

let r = MayBe.of(null)
 .map(x => x.toUpperCase())
 console.log(r)

 let p = MayBe.of('hello world');
 .map(x => x.toUpperCase());
 .map(x => null);
 .map(x => x.split(' '))
 //这样并不知道到底是哪个map方法传入了null
```

二十七、Either函子

- Either两者中的任何一个，类似于if...else...的处理
- 异常会让函数变得不纯，Either函子可以用来做异常处理

```
// Either函子
class Left {
```

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

class Right {
   static of (value) {
   return new Right(value)
   }
   constructor (value) {
   this._value = value
   }
   map (fn) {
   return Right.of(fn(this._value))
   }
}

let r1 = Right.of(12).map(x => x + 2);
let r2 = Left.of(12).map(x => x + 2);
console.log(r1)
console.log(r2)

function parseJSON (str) {
   try {
   return Right.of(JSON.parse(str))
   } catch (e) {
   return Left.of({ error: e.message })
   }
}
```

```
38
39 let r = parseJSON('{ "name": "ez"}') //Right{_value:{name:'e
z'}}
40 console.log(r)
41
42 let r = parseJSON('{ "name": "ez"}')
43  .map(x => x.name.toUpperCase())
44 console.log(r) //Right{_value:{name:'EZ'}}
```

二十八、IO函子

- IO函子中的_value是一个函数，这里是把函数作为值来处理
- IO函子不可以把不纯的动作存储到_value中，延迟执行这个不纯的操作（惰性执行），包装当前的操作纯
- 把不纯的操作交给调用者来处理

```
1 // IO函子
2 const fp = require('lodash/fp');
3
4 class IO {
5  static of (value) {
6  return new IO(function() {
7  return value
8  })
9  }
10  construtor (fn) {
11  this._value = fn
12  }
13
14  map (fn) {
15  return new IO(fp.flowRight(fn, this._value))
16  }
17 }
18
19 //调用
20 let r = IO.of(process).map(p => p.execPath);
```

```
console.log(r); // IO {_value: function}
r._value() // 直接调用;
// 打印结果为：当前执行node的路径
```

二十九、folktale

Task异步执行

- 异步任务的实现过于复杂，我们使用folktale中的Task来演示
- folktale 一个标准的函数式编程库
    - 和lodash、ramda不同的是，他没有提供很多功能函数
    - 只提供了一些函数式处理的操作，例如：compose、curry等，一些函子Task、Either、MayBe等

```
// folktale中的compose、curry
const { compose, curry } = require('folktale/core/lambda');
const { toUpper, first} = require('lodash/fp');

let f = curry(2, (x, y) => {
 return x +y
})

console.log(f(1,2)); //3
 console.log(f(1)(2)); //3

 let f2 = compose(toUpper, first);
 console.log(f2(['one', 'two'])); // ONE
```

三十、Task函子

```
//Task 处理异步任务
const { task } = require('folktale/concurrency/task');
```

```
const fs = require('fs');
const { split, find } = require('lodash/fp');

fucntion readFile(filename) {
 return task(resolver => {
 fs.readFile(filename, 'utf-8', (err, data) => {
 if (err) resolver.reject(err)
  resolver.resolve(data)
  })
  })
}

readFile('package.json')
  .map(split('\n'))
  .map(find(x => x.includes('version')))
  .run()
  .listen({
  onRejected: err => {
  console.log(err)
  },
  onResolved: value => {
  console.log(value)
  }
  })
```

三十一、Pointed函子

- Pointed函子是实现了of静态方法的函子
- of方法是为了避免使用new来创建对象，更深层的含义是of方法用来把值放到上下问Context（把值放到容器中，使用map来处理值）

三十二、Monad（单子）

在使用IO函子的时候，如果我们写出如下代码：

```
//IO函子的问题
const fp = require('lodash/fp');
const fs = require('fs');

class IO {
 static of (value) {
 return new IO(function() {
 return value
 })
  }
  construtor (fn) {
  this._value = fn
  }

  map (fn) {
  return new IO(fp.flowRight(fn, this._value))
  }
}

let readFile = function (filename) {
 return new IO(function () {
 return fs.readFileSync(filename, 'utf-8')
 })
}
let print = function(x) {
 return new IO(function () {
 console.log(x)
 return x;
 })
}

let cat = fp.flowRight(print, readFile)
// IO(IO())
let r = cat('package.json')
console.log(r); // IO { _value: [Function] }
```

```
r._value(); // IO { _value: [Function] }
r._value()._value(); // 文件内容
// 要嵌套调用，很麻烦
```

三十三、Monad函子2

- Monad函子是可以变扁的Pointed函子，IO(IO(x));
- 一个函子如果具有join和of两个方法并遵守一些定律就是一个Monad

```
// IO Monad
const fp = require('lodash/fp');
const fs = require('fs');

class IO {
 static of (value) {
 return new IO(function() {
 return value
 })
  }
  construtor (fn) {
  this._value = fn
  }

  map (fn) {
  return new IO(fp.flowRight(fn, this._value))
  }

  join () {
  return this._value()
  }

  flatMap (fn) {
  return this.map(fn).join()
  }
}
```

```
27
28 let readFile = function (filename) {
29  return new IO(function () {
30  return fs.readFileSync(filename, 'utf-8')
31  })
32 }
33 let print = function(x) {
34  return new IO(function () {
35  console.log(x)
36  return x;
37  })
38 }
39
40 let r = readFile('package.json');
41  //.map(x => x.toUpperCase())
42  .map(fp.toUpper)
43  .flatMap(print)
44  .join()
45 console.log(r)
```

三十四、总结

- 认识函数式编程
- 函数相关复习
  - 函数是一等公民
  - 高级函数
  - 闭包
- 函数式编程基础
  - lodash
  - 纯函数
  - 柯里化
  - 管道

    - 函数组合
- 函子
    - Functor
    - MayBe
    - Either
    - IO
    - Task folktale
    - Monad