# **为什么学习函数式编程**

1. 函数编程是非常古老的一个概念，早于第一台计算的诞生，函数式编程的历史。
2. 函数式编程是跟随这React的流行收到越来越多的关注
3. Vue3也开始拥抱函数式编程
4. 函数式编程可以抛弃this
5. 打包过程中可以更好的利用tree shaking过滤不用代码
6. 方便测试，方便并行处理 有很多库可以帮助我们进行函数式开发：loash、underscore、ramda

# **什么是函数式编程**

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式：

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

*// 非函数式* **let** num1 = 2 **let** num2 = 3 **let** **sum** = num1 + num2

console.**log**(**sum**)*// 函数式 相同的输入要有相同的输出*

function add (n1, n2) { **return** n1 + n2 }**let** **sum** = add(2, 3)

console.**log**(**sum**)

# **函数是一等公民**

* 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 Function() )，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

*// 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

}

# **高阶函数**

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

数作为参数

*// 高阶函数-函数作为参数***function** **forEach** (array, fn) {

**for** (**let** i = 0; i < **array**.length; i++) {

fn(**array**[i])

}

}

*// 测试// let arr = [1, 3, 4, 7, 8]// forEach(arr, function (item) {// console.log(item)// })*

*// filter***function** **filter** (array, fn) {

**let** results = []

**for** (**let** i = 0; i < **array**.length; i++) {

**if** (fn(**array**[i])) {

results.push(**array**[i])

}

}

**return** results

}

*// 测试***let** arr = [1, 3, 4, 7, 8]**let** r = filter(arr, **function** (item) {

**return** item % 2 === 0

})

console.log(r)

函数作为返回值

*// 高阶函数-函数作为返回值// function makeFn () {// let msg = 'Hello function'// return function () {// console.log(msg)// }// }// // const fn = makeFn()// // fn()// makeFn()()// once*

**function** **once** (fn) {

**let** done = false

**return** **function** () {

**if** (!done) {

done = true

**return** fn.apply(**this**, arguments)

}

}

}**let** pay = once(**function** (money) {

console.log(`支付: ${money} RMB`)

})

pay(5)

pay(5)

pay(5)

pay(5)

# **高阶函数的意义**

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

*// 面向过程的方式*

let array = [1, 2, 3, 4] **for** (let i = 0; i < array.length; i++) {

console.log(array[i])

}*// 高阶高阶函数*

let array = [1, 2, 3, 4]

forEach(array, item => {

console.log(item)

})

let r = filter(array, item => { **return** item % 2 === 0

})

# **常用高阶函数**

* forEach
* map
* filter
* every
* some
* find/findIndex reduce
* sort

*// 模拟常用高阶函数：map、every、some// map //作用：对数组中的元素遍历 并对每个每个元素进行处理 最后返回一个新的数组***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// 判断数组中每个元素是否 匹配指定的条件***const** every = (array, fn) => {

**let** result = true

**for** (**let** value **of** array) {

result = fn(value)

**if** (!result) {

**break**

}

}

**return** result

}

*// 测试// let arr = [9, 12, 14]// 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, 3, 5, 9]**let** r = some(arr, v => v % 2 === 0)console.log(r)

# **高阶函数**

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

数作为参数

*// 高阶函数-函数作为参数***function** **forEach** (array, fn) {

**for** (**let** i = 0; i < **array**.length; i++) {

fn(**array**[i])

}

}

*// 测试// let arr = [1, 3, 4, 7, 8]// forEach(arr, function (item) {// console.log(item)// })*

*// filter***function** **filter** (array, fn) {

**let** results = []

**for** (**let** i = 0; i < **array**.length; i++) {

**if** (fn(**array**[i])) {

results.push(**array**[i])

}

}

**return** results

}

*// 测试***let** arr = [1, 3, 4, 7, 8]**let** r = filter(arr, **function** (item) {

**return** item % 2 === 0

})

console.log(r)

函数作为返回值

*// 高阶函数-函数作为返回值// function makeFn () {// let msg = 'Hello function'// return function () {// console.log(msg)// }// }// // const fn = makeFn()// // fn()// makeFn()()// once*

**function** **once** (fn) {

**let** done = false

**return** **function** () {

**if** (!done) {

done = true

**return** fn.apply(**this**, arguments)

}

}

}**let** pay = once(**function** (money) {

console.log(`支付: ${money} RMB`)

})

pay(5)

pay(5)

pay(5)

pay(5)

# **高阶函数的意义**

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

*// 面向过程的方式*

let array = [1, 2, 3, 4] **for** (let i = 0; i < array.length; i++) {

console.log(array[i])

}*// 高阶高阶函数*

let array = [1, 2, 3, 4]

forEach(array, item => {

console.log(item)

})

let r = filter(array, item => { **return** item % 2 === 0

})

# **常用高阶函数**

* forEach
* map
* filter
* every
* some
* find/findIndex reduce
* sort

*// 模拟常用高阶函数：map、every、some// map //作用：对数组中的元素遍历 并对每个每个元素进行处理 最后返回一个新的数组***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// 判断数组中每个元素是否 匹配指定的条件***const** every = (array, fn) => {

**let** result = true

**for** (**let** value **of** array) {

result = fn(value)

**if** (!result) {

**break**

}

}

**return** result

}

*// 测试// let arr = [9, 12, 14]// 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, 3, 5, 9]**let** r = some(arr, v => v % 2 === 0)console.log(r)

# **纯函数概念**

纯函数：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 是一个一致性、模块化、高性能的 JavaScript 实用工具库(lodash 的 fp 模块提供了对函数 式编程友好的方法)，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 numbers = [1, 2, 3, 4, 5] *// 纯函数*

numbers.slice(0, 3) *// => [1, 2, 3]*

numbers.slice(0, 3) *// => [1, 2, 3]*

numbers.slice(0, 3) *// => [1, 2, 3] // 不纯的函数*

numbers.splice(0, 3) *// => [1, 2, 3] numbers.splice(0, 3) // => [4, 5] numbers.splice(0, 3) // => []*

* 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，所以可以把纯函数的结果缓存起来

1

2

3

4

**const** \_ = require('lodash') **function** **getArea** (r) { **return** Math.PI \* r \* r

}**let** getAreaWithMemory = \_.memoize(getArea) console.log(getAreaWithMemory(4))

自己模拟一个 memoize 函数

**function** **memoize** (f) {**let** cache = {} **return** **function** () { **let** arg\_str = JSON.stringify(arguments) cache[arg\_str] = cache[arg\_str] || f.apply(f, arguments) **return** cache[arg\_str]

}

}

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

// 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

}

* 副作用让一个函数变的不纯(如上例)，纯函数的根据相同的输入返回相同的输出，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带来副作用，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。

# **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）
* 然后返回一个新的函数接收剩余的参数，返回结果

*// 柯里化演示// function checkAge (age) {// let min = 18// return age >= min// }*

*// 普通的纯函数// function checkAge (min, age) {// return age >= min// }// console.log(checkAge(18, 20))// console.log(checkAge(18, 24))// console.log(checkAge(22, 24))*

*// 函数的柯里化// function checkAge (min) {// return function (age) {// return age >= min// }// }*

*// ES6***let** checkAge = min => (age => age >= min)**let** checkAge18 = checkAge(18)**let** checkAge20 = checkAge(20)console.log(checkAge18(20))console.log(checkAge18(24))

lodash 中的柯里化函数

* \_.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))*// 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)**const** filter = \_.curry(**function** (func, array) {

**return** array.filter(func)

})

**const** findSpace = filter(haveSpace)*// console.log(haveSpace('helloworld'))// console.log(haveNumber('abc'))*

console.log(filter(haveSpace, ['John Connor', 'John\_Donne']))

console.log(findSpace(['John Connor', 'John\_Donne']))*// 模拟实现 lodash 中的 curry 方法*

**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))

**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)))

}

}

**return** func(...args)

}

}

# **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
* 可以把多元函数转换成一元函数，可以组合使用函数产生强大的功能

# **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]))

# **Lodash组合函数**

* lodash 中组合函数 flow()或者flowRight()，他们都可以组合多个函数
* flow() 是从左到右运行
* 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']))

模拟实现 lodash 的 flowRight 方法

*// 模拟 lodash 中的 flowRight***const** reverse = arr => arr.reverse()**const** first = arr => arr[0]**const** toUpper = s => s.toUpperCase()*// function compose (...args) {// return function (value) {// return args.reverse().reduce(function (acc, fn) {// return fn(acc)// }, value)// }// }***const** compose = (...args) => value => args.reverse().reduce((acc, fn) => fn(acc), value)**const** f = compose(toUpper, first, reverse)console.log(f(['one', 'two', 'three']))

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

// 函数组合要满足结合律

const \_ = require('lodash')

// const f = \_.flowRight(\_.toUpper, \_.first, \_.reverse)

// const f = \_.flowRight(\_.flowRight(\_.toUpper, \_.first), \_.reverse)

const f = \_.flowRight(\_.toUpper, \_.flowRight(\_.first, \_.reverse))

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

# **调试**

// 函数组合 调试

// NEVER SAY DIE --> never-say-die

const \_ = require('lodash')

// const log = v => {

// console.log(v)

// return v

// }

const trace = \_.curry((tag, v) => {

console.log(tag, v)

return v

})

// \_.split()

const split = \_.curry((sep, str) => \_.split(str, sep))

// \_.toLower()

const join = \_.curry((sep, array) => \_.join(array, sep))

const map = \_.curry((fn, array) => \_.map(array, fn))

const f = \_.flowRight(join('-'), trace('map 之后'), map(\_.toLower), trace('split 之后'), split(' '))

console.log(f('NEVER SAY DIE'))

# **lodash/fp**

* lodash 的 fp模块提供了实用的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
* 函数优先 数据之后

// lodash 的 fp 模块

// NEVER SAY DIE --> never-say-die

const fp = require('lodash/fp')

const f = fp.flo**wRight(**fp.jo**in(**'-'), fp.map(fp.toLower), fp.split(' '))

console**.log(**f('NEVER SAY DIE'))

// lodash 和 lodash/fp 模块中 map 方法的区别

// const \_ = require('lodash')

// console**.log(**\_.map(['23', '8', '10'], parseInt))

// // pars**eInt(**'23', 0, **array**)

// // pars**eInt(**'8', 1, **array**)

// // pars**eInt(**'10', 2, **array**)

const fp = require('lodash/fp')

console**.log(**fp.map(parseInt, ['23', '8', '10']))

# **Point Free**

* Point Free：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。

1. 不需要指明处理的数据
2. 只需要合成运算过程
3. 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** (word) { **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'))

*//使用 Point Free 的模式，把单词中的首字母提取并转换成大写***const** fp = require('lodash/fp')**const** firstLetterToUpper = fp.flowRight(join('. '), fp.map(fp.flowRight(fp.first,

fp.toUpper)), split(' '))

console.log(firstLetterToUpper('world wild web')) *// => W. W. W*

# **Functor 函子**

* 为什么要学函子

1. 到目前为止已经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

1. 容器：包含值和值的变形关系(这个变形关系就是函数)
2. 函子：是一个特殊的容器，通过一个普通的对象
3. 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)

Functor 函子

*// 一个容器，包裹一个值* **class** Container { *// of 静态方法，可以省略 new关键字创建对象*

static **of** (**value**) {

**return** **new** Container(**value**)

}

constructor (**value**) { **this**.\_**value** = **value** }*// map 方法，传入变形关系，将容器里的每一个值映射到另一个容器*

map (fn) {

**return** Container.**of**(fn(**this**.\_**value**)) }

}*// 测试*

Container.**of**(3)

.map(x => x + 2)

.map(x => x \* x)

## **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）
* 在 Functor 中如果我们传入 null 或 undefined

*// 值如果不小心传入了空值(副作用)*Container.**of**(null)

.map(x => x.**to**UpperCase()) *// TypeError: Cannot read property 'toUpperCase' of null*

# **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**.\_value))

}

isNothing () {

**return** **this**.\_value === null || **this**.\_value === undefined

}

}

*// let r = MayBe.of('Hello World')// .map(x => x.toUpperCase())// console.log(r)*

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

**let** r = MayBe.of('hello world')

.map(x => x.toUpperCase())

.map(x => null)

.map(x => x.split(' '))console.log(r)

# **Either 函子**

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

**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)*

<!--Either 用来处理异常-->**function** parseJSON (str) {

**try** {

**return** Right.**of**(JSON.parse(str))

} **catch** (e) {

**return** Left.**of**({ error: e.message })

}

}

*// let r = parseJSON('{ name: zs }')// console.log(r)*

**let** r = parseJSON('{ "name": "zs" }')

.map(x => x.name.toUpperCase())

console.log(r)

# **Either 函子**

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

**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)*

<!--Either 用来处理异常-->**function** parseJSON (str) {

**try** {

**return** Right.**of**(JSON.parse(str))

} **catch** (e) {

**return** Left.**of**({ error: e.message })

}

}

*// let r = parseJSON('{ name: zs }')// console.log(r)*

**let** r = parseJSON('{ "name": "zs" }')

.map(x => x.name.toUpperCase())

console.log(r)

# **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

**const** fp = require('lodash/fp')**class** **IO** {

**static** **of** (value) {

**return** **new** IO(**function** () {

**return** value

})

}

**constructor** (fn) {

**this**.\_value = fn

}

map (fn) {

**return** **new** IO(fp.flowRight(fn, **this**.\_value))

}

}

*// 调用***let** r = IO.of(process).map(p => p.execPath)*// console.log(r)*console.log(r.\_value())

# **IO 函子的问题**

*// IO 函子的问题***const** fs = require('fs')**const** fp = require('lodash/fp')**class** **IO** {

**static** **of** (value) {

**return** **new** IO(**function** () {

**return** value

})

}

**constructor** (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(x))***let** r = cat('package.json').\_value().\_value()console.log(r)

# **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文 Context（把值放到容器中，使用 map 来处理值）

**class** **Container** { **static** **of** (**value**) { **return** **new** Container(**value**)

}……

}

Contanier.of(2)

.map(x => x + 5)

# **Monad（单子）**

* 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** fs = require('fs')

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

**class** **IO** {

**static** **of** (value) {

**return** **new** IO(**function** () {

**return** value

})

}

**constructor** (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()

}

}

**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** r = readFile('package.json')

*// .map(x => x.toUpperCase())*

.map(fp.toUpper)

.flatMap(print)

.join()console.log(r)

函数组合

* 纯函数和柯里化很容易写出洋葱代码 h(g(f(x)))
* 获取数组的最后一个元素再转换成大写字母， *.toUpper(*.first(\_.reverse(array)))
* 函数组合可以让我们把细粒度的函数重新组合生成一个新的函数



管道



**JavaScript性能优化**

**性能优化**

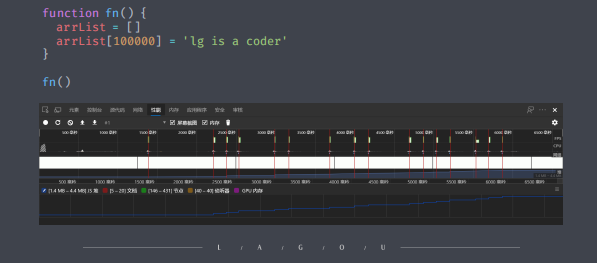
* 性能优化是不可避免的
* 那些内容可以看作是性能优化
* 无处不在的前端性能优化

**内容概要**

* 内存管理
* 垃圾回收与常见的GC算法
* V8引擎的垃圾回收
* Performance工具
* 代码优化实例

**JavaScript内存管理**

内存为什莫需要管理



**内存管理介绍**

* 内存：由可读写单元组成，表示一片可操作空间
* 管理：认为的去操作一片工间的申请、使用和释放
* 内存管理：开发者主动哦申请空间、使用空间、释放空间
* 管理流程：申请-使用-释放

**JavaScript中的内存管理**

* 申请内存空间
* 使用内存空间
* 释放内存空间

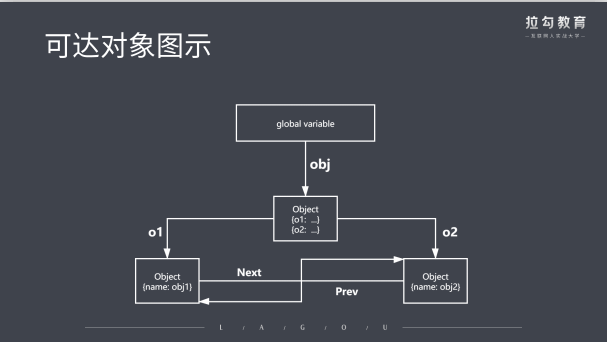
**JavaScript 中的垃圾回收**

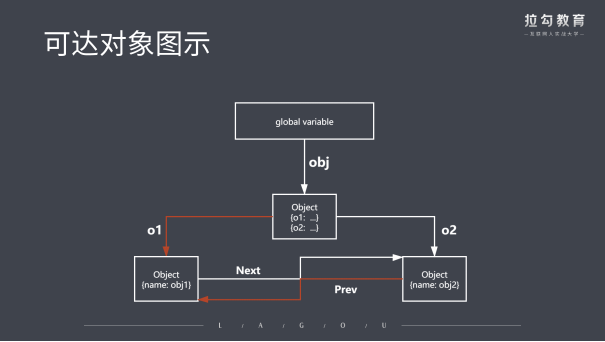
* JavaScript中内存管理是自动的
* 对象不再被引用时是垃圾
* 对象不能从跟上访问到时是垃圾

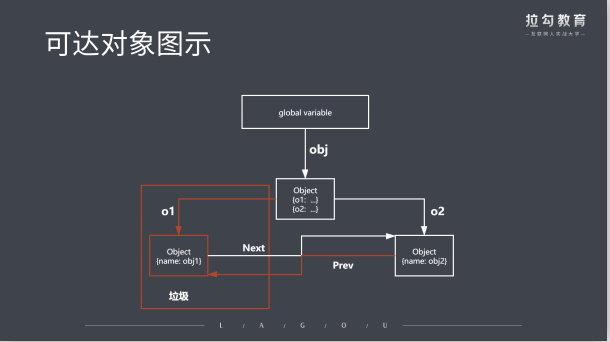
**JavaScripe 中的可达对象**

* 可以访问到对象就是可达对象（引用、作用区域链）
* 可达的标准就是从根本出发是否能够被找到
* JavaScripe中的根就可以里理解为全局变量对象

**JavaScript 中的引用与可达**





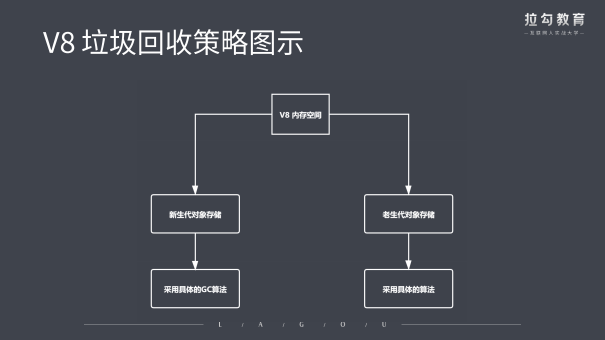


**认识V8**

* V8是一款主流JavaScript执行引擎
* V8采用即使编译
* V8内存设限

**V8垃圾回收策略**

* 采用分代回收的思想
* 内存分为新生代、老生代
* 针对不同对象采用不同算法



**V8中常用GC算法**

* 分代回收
* 空间复制
* 标记清楚
* 标记整理
* 标记增量



**V8内存分配**

* V8内存空间一分为二
* 小空间用于储存新生代对象（32M|16M）
* 新生代指的是存活时间较短的对象

**新生代对象回收实现**

* 回收过程采用复制算法+标记整理
* 新生代内存区分为两个等大小空间
* 使用空间为From，空闲空间为To
* 活动对象储存From空间
* 标记整理后将活对象拷贝值To
* From与To交换换成空间释放

**回收明细说明**

* 拷贝过程中可能出现晋升
* 晋升就是将新生代对象移至老生代
* 一轮GC还存活的新生代需要晋升
* To空间的使用率超过25%

**V8如何回收老生代对象**

**老年代对象说明**

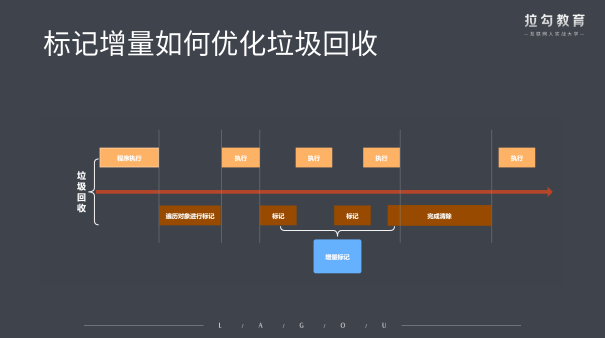
* 老年代对象存放在右侧老生代区域
* 64位操作系统1.4G，32操作系统700M
* 老年代对象就是值存活时间较长的对象

**老年代对象回收**

* 主要采用标记清除、标记整理、增量标记算法
* 首先使用标记清除完成垃圾空间的回收
* 采用标记整理进行空间优化
* 采用增量标记进行效率优化

**细节对比**

* 新生代区域垃圾回收使用空间换时间
* 老生代区域垃圾回收不适合复制算法



**V8垃圾回收总结**

* V8是一款主流的JavaScript执行引擎
* V8内存设置上限
* V8采用基于分代回收思想实现垃圾回收
* V8内存分为新生代和老生代
* V8垃圾回收常见的GC算法

**为什莫使用Performance**

* GC的目的是为了实现内存空间的良性循环
* 良性循环的基石是合理使用
* 时刻关注才能确定是否合理
* Performance提供多种监控方式

**Performance使用步骤**

* 打开浏览器输入目标网址
* 进入开发人员工具面板，选择性能
* 开启录制功能，访问具体界面
* 执行用户行为，一段时间后停止录制
* 分析界面中记录的内存信息

**内存问题的外在表现**

* 页面出现延迟加载或者经常性暂停
* 页面持续性出现糟糕的性能
* 页面的型嫩通过随着时延长越来越差

**界定内存问题的标准**

* 内存泄漏：内存使用持续升高
* 内存膨胀：在多数设备上都存在性能问题
* 频繁垃圾回收：通过内存变化图进行分析

**监控内存的几种方式**

* 浏览器任务管理器
* Timeline时序图记录
* 堆快照查找分离DOM
* 判断是否存在频繁的垃圾回收

**什么是分离DOM**

* 界面元素存活在DOM树上
* 垃圾对象时的DOM节点
* 分离状态的DOM节点

**为什么确定频繁垃圾回收**

* GC工作时应用程序是停止的
* 频繁且过长的GC会导致应用假死
* 用户使用感知应用卡顿

**Performance使用**

* Performance使用流程
* 内存问题的相关解析
* Performance时序图监控内存变化
* 任务管理器监控内存变化
* 堆快照查找分离DOM

**代码优化介绍**

* JavaScript中的内存管理自动完成
* 执行引擎会使用不同的GC算法
* 算法工作的目的是为了实现内存空间良性循环
* Performance工具检测内存变化
* JavaScript是单线程机制的解析型语言

**避免全局变量**

**全局变量特点**

* 全局变量挂载在window下
* 全局变量至少有一个引用计数
* 全局变量存活更久，但持续占用内存

**避免全局查找**

**全局查找相关**

* 目标变量不存在于当前作用内，通过作用域链向上查找
* 减少全局查找降低时间消耗
* 减少不必要的全局变量定义
* 全局变量数据局部化

**避免循环引用**

**全局引用指多个对象间存在相互引用**

**采用字面量替换New操作**

**setTimeout替换setInterval**

**合并循环变量和条件**

**代码优化介绍**

**代码为什莫需要优化**

* WEB应用日益丰富，用户体验至关重要，前端性能备受关注
* 实现相同结果，那种类型JavaScript代码具有更高性能
* 高性能的背后是数据快速存取，也是优秀内存管理的体现

**如何精准测试JavaScript性能**

* 本质上就是采用大量的执行样本进行数学统计和分析
* 使用基于benchmark.js的https://jsperf.com完成

**Jsperf使用流程**

* 使用Github账号登陆
* 填写个人信息（非必须）
* 填写详细的测试用例信息（title、slug）
* 填写准备代码（DOM操作时经常使用）
* 填写必要有setup与teardown
* 填写测试代码片段

**慎用全局变量**

**为什么要慎用**

* 全局变量定义在全局执行上下文，是所有作用域的顶端
* 全局执行上下文一直存在于上下文执行栈，直到程序退出
* 如果某个全局作用域出现了同名变量则会遮蔽或污染全局

**明确数据作用域的情况下，尽量用局部**

**缓存全局变量**

**将使用中无法避免的全局变量缓存到局部**

**通过原型新增方法**

**在原型对象上新增实例对象需要的方法**

**避开闭包陷阱**

关于闭包

* 闭包是一种强大的语法
* 闭包使用不当容易出现内存泄漏
* 不要为了闭包而闭包

**避免属性访问方法使用**

**JavaScript中的面向对象**

* JS不需属性的访问方法，所有属性都是外部可见的
* 使用属性访问方法指挥增加一层重定义，没有访问的控制力

**For循环优化**

从大到小遍历

**采用最有循环方式**

**节点添加优化**

**文档碎片**

**节点的添加操作必然会有回流和重绘**

**克隆优化节点操作**

**直接量替换object操作**

**GC定于与作用**

* GC就是垃圾回收机制的简称
* GC可以找到内存中的垃圾、并释放和回收空间

**GC里的垃圾是什么**



**GC算法是神魔**

* GC是一种机制，垃圾回收器完成具体的工作
* 工作的内容就是查找垃圾释放空间、回收空间
* 算法就是工作时查找和回收所遵循的规则

**常见GC算法**

* 引用计数
* 标记清除
* 标记整理
* 分代回收

**引用计数算法**

* 核心思想：设置引用数，判断当前引用数是否为0
* 引用计数器
* 引用关系改变时修改引用数字
* 引用数字为0时立即回收

**···引用计数算法优点**

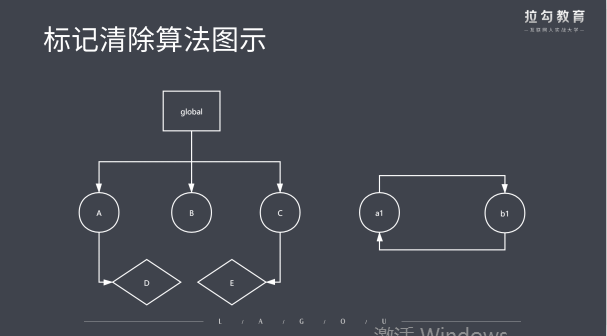
* 发现垃圾时立即回收
* 最大限度减少程序暂停

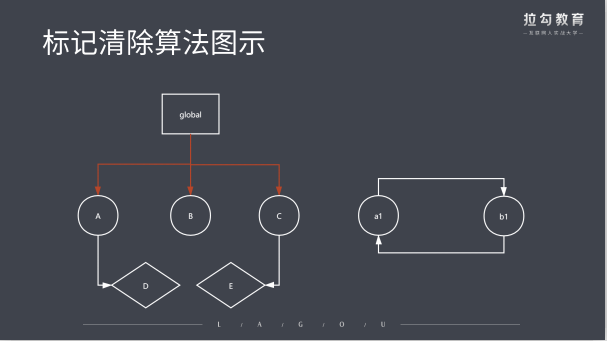
**···引用计数算法缺点**

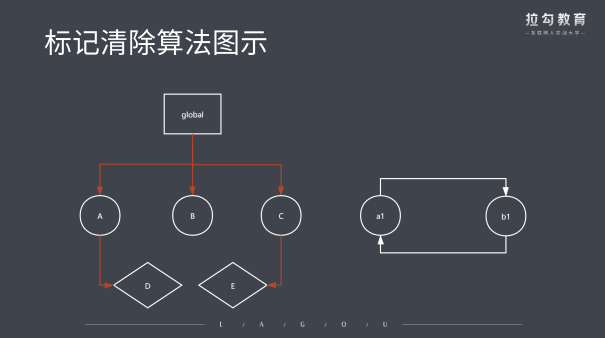
* 无法回收循环引用的对象
* 时间开销大

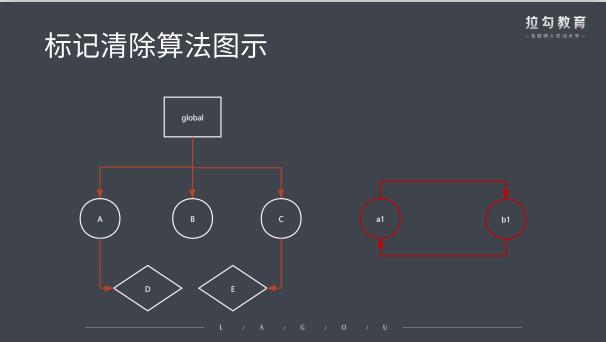
**标记清除算法**

* 核心思想：分标记和清除二个阶段完成
* 遍历所有对象找标记活动对象
* 遍历所有对象清除没有标记对象
* 回收相应的空间







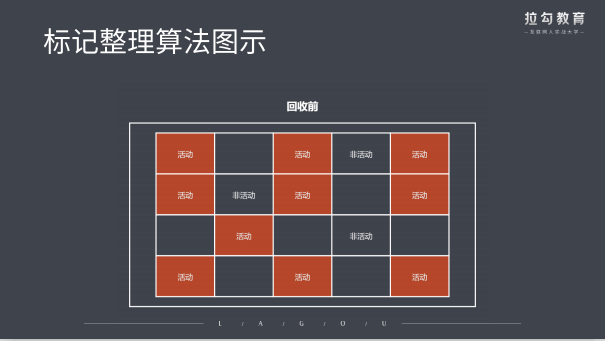


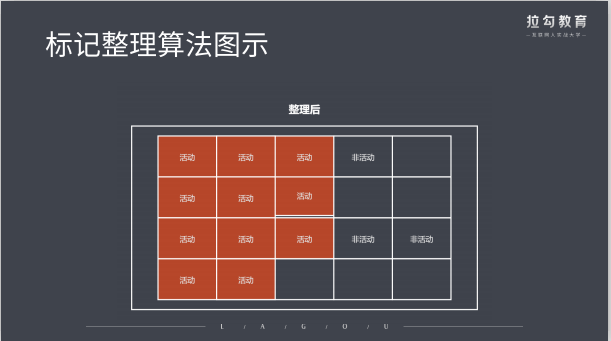
**标记清除算法优点**

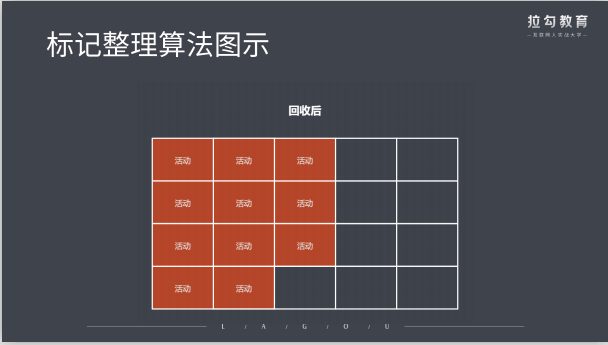
**标记清除算法优点**

**标记整理算法**

* 标记整理可以看作时标记清除的增强
* 标记阶段的操作和清除一致
* 清除阶段会先执行整理，移动对象位置







**常见GC算法**

* 引用计数
* 标记清除
* 标记整理

**引用计数优缺点**

* 可以即时回收垃圾对象
* 减少程序卡顿时间
* 无法回收循环引用的对象
* 资源消耗较大

**标记清除优缺点**

* 以西回收循环引用的对象
* 容易产生碎片化空间、浪费空间
* 不会立即回收垃圾对象

**标记整理优缺点**

* 减少碎片化空间
* 不会立即回收垃圾对象
* 移动对象位置，回收效率慢