

## 2. 为什么学习函数式编程?

→ 为什么要学?

- React 很流行，开发者关注
- Vue 3.0 的趋势
- 可以抛弃 this
- 打包过程中可以更好的利用 tree shaking 过滤无用代码
- 方便调试、方便并行处理
- 有一些库帮助我们：lodash underscore ramda

→ 存在的问题

- 可读性的问题（对新手不好）
- 用的人不多
- 需要使用大量的“柯里化”
- 需转换习惯

## 3. 函数式编程概念

→ 定义

函数式编程 (Functional Programming, FP) 是一种编程范式。

常见的编程范式：面向对象，面向过程

- 面向对象的思维方式：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件事件之间的联系。
- 函数式编程思维：把现实世界中的事物和事物之间的事物关系抽象到程序世界

◦ 关键：同样的输入获得同样的输出

◦ 注意：“函数式”是指程序中的 function，而不是数学中的“映射”关系

→ 前置知识

- 函数是一等公民、高阶函数、闭包

补充：

- 降低认知成本
- 削减了“副作用”

概念：副作用是在计算结果

的进程中导致状态

的一种变化，或与世

界进行双向影响交互。

## 4. 函数是一等公民

→ MDN First-class Function

- 函数可存储于变量中
- 函数作为参数
- 函数作为返回值

→ 在 JS 中函数就是一个普通对象 (new Function())

```
1 // 把函数赋值给变量
2 let fn = function () {
3   console.log('Hello First-class Function')
4 }
5 fn()
6
7 // 一个示例
8 const BlogController = {
9   index (posts) { return Views.index(posts) },
10  show (post) { return Views.show(post) },
11  create (attrs) { return Db.create(attrs) },
12  update (post, attrs) { return Db.update(post, attrs) },
13  destroy (post) { return Db.destroy(post) }
14 }
15 // 优化
16 const BlogController = {
17   index: Views.index,
18   show: Views.show,
19   create: Db.create,
20   update: Db.update,
21   destroy: Db.destroy
22 }
```

← 只操作函数方法而非返回值

## 5.6.7.8 高阶函数

→ 什么是高阶函数 Higher-order Function

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

→ 高阶函数的意义

- 抽象可以帮我们屏蔽细节，只关心我们的目标
- 高阶函数是用来抽象通用的问题

→ 常用的高阶函数

- filter, map, every, some

### ◦ map

```
const map = (array, fn) => {
  let result = []
  for(let value of array){
    result.push(fn(value))
  }
  return result
}
```

### ◦ every

```
const every = (array, fn) => {
  let result = true
  for(let value of array){
    result = fn(value)
    if(!result){
      return result
    }
  }
  return result
}
```

### ◦ some

```
const some = (array, fn) => {
  let result = false
  for(let value of array){
    result = fn(value)
    if(result){
      return result
    }
  }
}
```

## 9.10 闭包

→ 概念.

- Closure 闭包：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
- 可以在另一个作用域中调用函数内部方法

→ 例子

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

```
// once function 只会执行一次
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`)
})
```

→ 本质

- 函数在执行的时候会放到一个执行线上，当函数执行完毕之后会从执行线上移除，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成员。

→ 例子

```
function makePower (power) {  
  return function (number){  
    return Math.pow(number, power)  
  }  
  
let power2 = makePower(2)  
let power3 = makePower(3)  
  
console.log(power2(4))  
console.log(power3(3))
```

browser dev tool → source → 断点分析 | call stack

Scope

## 11 12 13 14 函数式编程

→ 概念

- 纯函数 Pure Function: 相同的输入，始终会得到相同的输出
- 无任何可观察副作用
- 函数式编程不会保留任何计算中间的结果，所以变量是不可变的（无状态的）
- 我们可以把一个函数执行结果交给另一个函数处理
- 例子

◦ slice(3, 6) 和 splice(3, 6)

→ Lodash

• 工具库 辅助 → FP

• 安装: npm i lodash

• 引用: const \_ = require('lodash')

• 例子

◦ first、toUpper、reverse、  
includes、find、findIndex、etc

→ 纯函数好处

• 可缓存

◦ 因为纯函数输入始终有相同的结果

可以把纯函数结果缓存

◦ 例子

### 1. 调用 lodash 中的 memoize.

```
const _ = require('lodash')
function getArea(r){
  console.log('Run one time')
  return Math.PI * r * r
}
let getAreaWithMemory = _.memoize(getArea)
console.log(getAreaWithMemory(4)) //Run one time 50.26548245743669
console.log(getAreaWithMemory(4)) //50.26548245743669
console.log(getAreaWithMemory(4)) //50.26548245743669
```

### 2. 自建 memoize()

```
function memorize (f) {
  let cache = {}
  return function () {
    let key = JSON.stringify(arguments)
    cache[key] = cache[key] || f.apply(f, arguments)
    return cache[key]
  }
}
```

- 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有意外

- 纯函数不需要共享内存数据，所以在并行环境下可以运行任意纯函数

(通过 Web Worker 并行编程)

## → 副作用问题

- 什么是副作用

- 副作用让一个函数变的不纯，如果函数依赖于外部的状态，那就无法保证输出相同，进而带来副作用

- 来源

- 配置文件

- 外部交互

- 数据库

- 硬编码

- 获取用户的输入

- 带来的影响

- 通用性↓

- 安全性↓ (跨站脚本攻击)

- 重用性
- 和函数结合?

◦ 纯函数

## 16 17 18 19 框型化 Currying

→ 用途

- 解决硬编码问题

→ 例子

```
//Original
function OcheckAge(age){
  const min = 18
  return age >= min
}

function OcheckAge(min, age){
  return age >= min
}

//Currying
function checkAge(min){
  return function(age){
    return age >= min
  }
}

//Curry with ES6
let CheckAge = (min) => ((age) => age >= min )
```

→ 概念

- 当一个函数有多个参数调用的时候先传递一部分参数调用它这部分以后不变)
- 然后返回一个新函数接着剩余的参数，返回结果。

→ Lodash 中的通用柯里化方法

- .curry(func)

◦ 功能：创建一个函数，该函数接收一个或者多个func参数，如果func所需要的参数都已被提供则执行func并返回执行的结果。否则继续返回函数并等待待接收剩余参数。

◦ 参数：需柯里化的函数

◦ 返回：柯里化后的函数。

◦ 例子：

```
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))
```

◦ 例子：

```
// .match(/\s+/g)  
// .match(/\d+/g)
```

↓ 柯里化

```
let test = function (reg, str){  
    return str.match(reg)  
}
```

↑ 柯里化

```
const _ = require('lodash')  
const match = _.curry(function (reg, str){  
    return str.match(reg)  
})  
const haveSpade = match [/^\s+/g]
```

→ 柯里化原理模块

```
function curry(func){  
    return function curriedfn(...args){  
        if(args.length < func.length){  
            return function (){  
                return curriedfn(...args.concat(Array.from(arguments)))  
            }  
        }  
        return func(...args)  
    }  
}
```

◦ 基础回顾

◦ Array.from 用来将类数组转为数组

◦ Array.concat 用来合并数组的

◦ 从左到右解析

→ 总结

◦ 柯里化内部可以让我们给一个函数传递较少参数得到一个已经记住某些固定参数的新函数。

◦ 这是一种对函数参数的缓存

◦ 让函数变得更灵活，让函数粒度更小

◦ 把多元函数转换为一元函数，可以组合使用函数产生强大的功能

20.21.22.23.24 组合函数

→ 一个问题

◦ 问题：柯里化和纯函数很容易写成洋葱代码

◦ 解决方案：函数组合可以让我们把纯函数重新组合成一个新函数

→ 管道

$\xrightarrow{a} (\quad) f_n (\quad) \xrightarrow{b}$

$\xrightarrow{a} (\quad f_3 (\quad) \xrightarrow{m} (\quad f_2 (\quad) \xrightarrow{n} (\quad f_1 (\quad) \xrightarrow{b}$

$f_n = \text{compose}(f_1, f_2, f_3)$

→ 概念.

• 函数组合 (compose): 如果一个函数需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成为一个函数

◦ 函数就像是数据流管, 函数组合就是把这些管道连接起来

◦ 这样就穿过多个管道形成最终结果

◦ 函数组合默认从右到左执行

→ 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 中的组合函数.

• lodash 中组合函数 flow() 或 flowRight(), 它们都可以组合多个函数.

◦ flow 是从右向左

◦ flowRight 是从左向右

→ 组合函数实现模拟.

```
const compose = function(...args){  
    return function(value){  
        return args.reverse().reduce(function(acc, fn){  
            return fn(acc)  
        }, value)//value是acc初始值  
    }  
}  
  
//ES6写法  
const compose = (...args) => value => args.reverse().reduce((acc, fn) => fn(acc), value)
```

• JavaScript 中 reduce() 方法不完全指南 (对基础型)

◦ 语法: arr.reduce(callbackI, initialValueI)

## ◦ 参数:

### 1. callback

a. previousValue 前值

b. currentValue 正在处理的数组元素

c. currentIndex 当前正在处理的数组元素下标

d. array (调用 reduce 的方法的参数)

### 2. initialValue (可选值。如果第一次调用函数时提供 previousValue 则值)

→ 结合律 (associativity)

•  $g(f(x))$  和  $f(g(x))$  是等效的

→ 调试

• [示例]

```
// 函数组合 调试
// NEVER SAY DIE --> neve-say-die

const _ = require('lodash')

// 调试关键点
const log = v => {
  console.log(v)
  return v
}

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

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

const f = _.flowRight(join('-'), _.toLowerCase, log, split(' '))
```

## 25 lodash-fp 模块

→ lodash-infp 模块提供了实用的对函数式编程友好的方法

→ 提供了可变 auto-curried iteratee-first data-last 方法

→ lodash-infp 模块 vs. lodash 原来

柯里化

无柯里化

函数优先, 数据滞后

数据优先, 函数滞后

→ 伎俩

```

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'])

```

→ lodash 中的 map 和 lodash/fp 的 map  
 • 区别：接收的函数参数不一样

27 28 Point Free

→ 概念

• 我们可以把数据处理的过程变成与数据无关的合成运算，不需要引用到代表数据的那个参数、只要把简单的运算聚合成一起。在使用这种模式之前我们需要定义一些辅助的基本运算符

- 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

```

// 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.toLowerCase)

console.log(f('Hello World'))

```

## 29.30 Functor 函数

→ 什么函数

- 把副作用限制在可控范围内，处理异常，异步操作

→ 什么是函数

- 容器：包含值和值的变形关系

- 函数：是一个特殊的容器，通过一个有 map 方法的对像来实现

map 方法运行一个函数来进行变形处理

```
2 class Container {  
3   constructor (value) {  
4     this._value = value  
5   }  
6  
7   map (fn) {  
8     return new Container(fn(this._value))  
9   }  
10 }  
11 // 避免出现  
12 let r = new Container(5)  
13   .map(x => x + 1)  
14   .map(x => x * x)  
15  
16 console.log(r)
```

无化

```
class Container {  
  static of (value) {  
    return new Container(value)  
  }  
  
  constructor (value) {  
    this._value = value  
  }  
  
  map (fn) {  
    return Container.of(fn(this._value))  
  }  
}  
  
let r = Container.of(5)  
  .map(x => x + 2)  
  .map(x => x * x)  
  
console.log(r)
```

· 注意：返回的始终是函数对象，并不是具体的值

→ 总结

· 函数式编程以运算值操作值，而不是由函数完成

· 函数就是一个实现了 map 约束的对象

· 我们可以把函数想象成一个盒子，这个盒子里面装了一个值

· 想要处理盒子中的值，我们需要给盒子传递一个纯函数，在这个函数进行处理

· 最终 map 方法返回一个包含新值的盒子

## 31. Maybe 函数

→ 介绍

· Maybe 函数的作用就是可以对外部的空值情况进行处理（限制在可控范围内）

→ 例子

```

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 n = Maybe.of(null)
    .map(x => x.toUpperCase())
console.log(n)

let d = Maybe.of('Hello World')
    .map(x => x.toUpperCase())
    .map(x => null)
    .map(x => x.split(''))
console.log(d)

```

一个问题

如果map方法中有一个

执行到了错误但又不

报错会很麻烦，

找不到出错点。

## 32. Either

→ 介绍

- Either两者之间一个类似于 if ... else ... 语句处理
- 异常会让函数变得不纯，Either可以用来异常处理

→ 例子

```

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

let r = parseJSON('{ "name" : "zs" }').map(x => x.name.toUpperCase())
console.log(r)

```

## 33 IO 面子不太懂

→ 介绍

- IO 面子中调用 value 是一个函数，这里是对函数作流值权处理
- IO 面子可以对纯副作用操作（副作用）进行包装，当前通过操作
- 把不纯副作用操作交给调用者来处理

→ 例子

```
// IO 面子
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._value)
```

## 34. folktale

→ Task 异步执行

- 异步任务过于复杂，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
  - 只是提供了一些函数式处理的操作，例如：compose, curry 等

Task, Either, Maybe 等面子

→ 例子

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

let f = compose(toUpper, first)
console.log(f['one', 'two'])
```

## 35. Task 面子

→ forkTask (2.3.2) 2.x 中的 Task 和 1.0 中的 Task 差异很大

→ 例子 (2.3.2 版本) (Task 处理异步)

```
// Task 处理异步任务
const fs = require('fs')
const { task } = require('../folktale/concurrency/task')
const { split, find } = require('lodash/fp')

function 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)
    }
})
```

### 36 Pointed 函数

- Pointed 函数实现了静态的派发函数
- 方法是通过避免使用 new 对象，更深层次的含义是方法用来把值放到上下文 Context  
(把值放到容器中，用 map 来处理值)

### 37 38 Monad 函数

```
// IO 函数的问题
const fs = require('fs')
const fp = require('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)
```

• Monad 函数是一个可以变扁的函数

• 一个函数如果具有 join 和 of 两种方法并遵守一些  
定律就是个 Monad

```
====>
// IO 函数的实现
1 const fs = require('fs')
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
11   constructor (fn) {
12     this._value = fn
13   }
14
15   map (fn) {
16     return new IO(fp.flowRight(fn, this._value))
17   }
18
19   join () {
20     return this._value()
21   }
22
23   flatMap (fn) {
24     return this.map(fn).join()
25   }
26
27 }
28
29 let readFile = function (filename) {
30   return new IO(function () {
31     return fs.readFileSync(filename, 'utf-8')
32   })
33 }
34
35 let print = function (x) {
36   return new IO(function () {
37     console.log(x)
38     return x
39   })
40 }
41
42 let r = readFile('package.json').map(fp.toUpper).flatMap(print).join()
43 console.log(r)
```

39总结

思维导图

# 任务 = JavaScript 中的异步编程

## 1 概述

→ 采用单线程的原因

- 防止多个线程操作数据导致混乱

→ 优点:

- 安全简单

→ 缺点:

- 阻塞

→ 内存泄漏

- 同步 vs 异步

- 事件 loop 和消息队列

- 异步编程的几种方式

- Promise 异步方案、宏任务 / 微任务队列

- Generator 异步方案，Async/Await 语法糖

## 2 同步模式 Synchronous

→ 是一个动态图，看视频

## 3 异步模式 Asynchronous

→ 说明

- 不会等待

- 开启后立即执行下一个函数

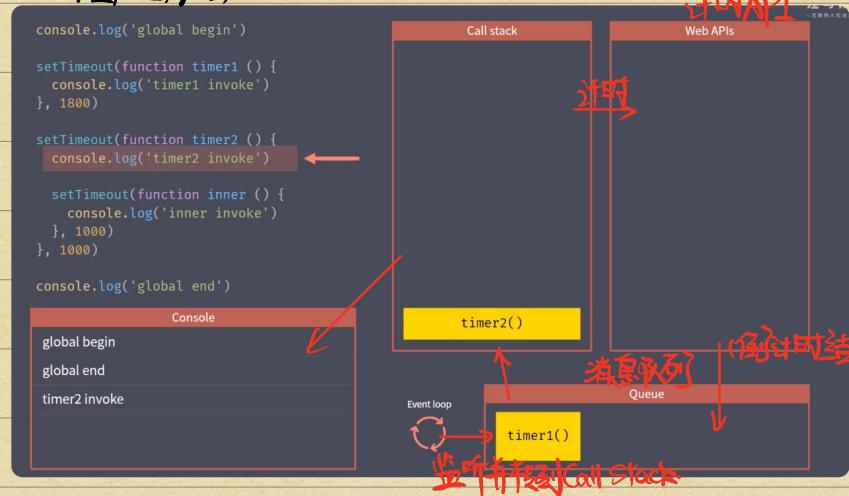
- 后续逻辑一般同回调函数的定义

- 单线程 JS 无法执行大型任务

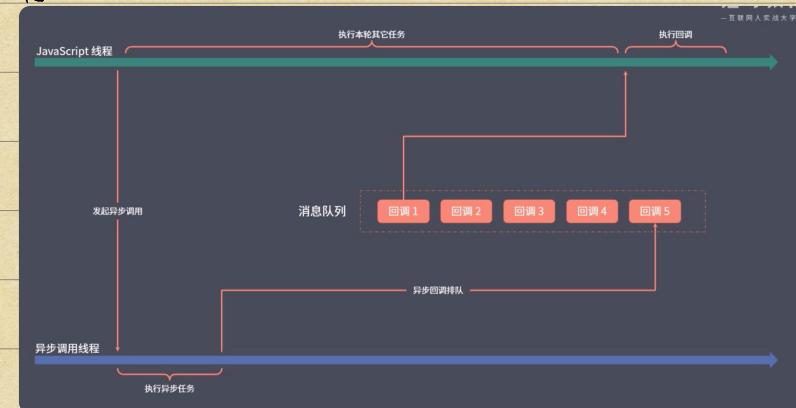
→ 又缺点

· 逻辑混乱

· 例子(动图,看视频)



→ 注意:



◦ JavaScript 基于单线程, 但浏览器并非单线程

◦ JavaScript 调用 API 都有自己线程

◦ 执行代码的线程是单线程

◦ 运行环境提供的 API 是以同步或异步模式完成工作

4 回调函数(所有异步操作的根基)

→ 定义

◦ 由调用者定义, 交给执行者执行的函数

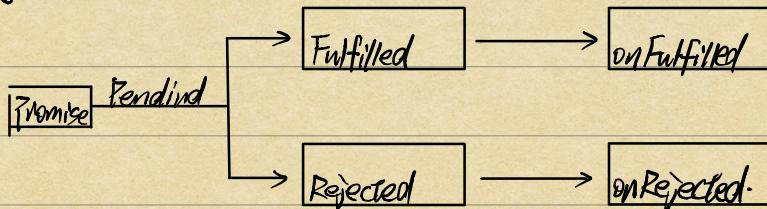
## 5.6.7.8.9.10.11.12.13. Promise

→ 引入

- CommonJS社区提出Promise规范并返回规范 Hell

- ES2015 官方规范规范

→ 状态



→ 基本用法

- 它是一个类型

- resolve() 作用是将状态改为 Fulfilled

◦ 给到一个值

- reject() 作用是将状态改为 Rejected

◦ 给到一个 Error

- 状态一旦被修改，不可回退

◦ 意味着 resolve 和 reject 只能进一个

- Promise 实例的 then 方法

◦ 状态改之后调用

◦ 第一个参数：onFulfilled 的回调函数

◦ 第二个参数：onRejected 的回调函数

→ 例子

```
//Promise 方式的 AJAX
function ajax(url){
    return new Promise(function(resolve, reject){
        var xhr = new XMLHttpRequest()
        xhr.open('GET', url)
        xhr.responseType = 'json'
        xhr.onload = function() {
            if (this.status === 200) {
                resolve(this.response)
            } else {
                reject(new Error(this.statusText))
            }
        }
        xhr.send()
    })
}
ajax('/api/user.json').then(function(res) {
    console.log(res)
}, function(error) {
    console.log(error)
})
```

## → 常见误区

- 嵌套使用方法是Promise最常见错误。应借用Promise方法链式调用的特点，尽可能避免嵌套。
- 每一个then方法实际上都是在上一个then返回的Promise对象上添加的，只有明确后加回调。  
这些Promise会依次执行，回调函数从前往后依次执行

## • 返回

1. 有值 → Promise 返回该值

非Promise then参数返回参数就是该值

2. 无值 → 返回 undefined.

## • 总结

○ Promise对象的then方法返回一个全新的Promise对象。

○ 后面的then方法就是在上一个then返回的Promise上物归原主

○ 前面then方法中回调函数的返回值会作为后面then方法回调的参数。

○ 如果回调中返回的是Promise，那后面then方法内部将公等待它的结果。

## → 异步处理

• catch方法相当于then方法的别名

○ catch(function A) 相当于 then(undefined, function A)

## • 注意

○ 在链式调用中用onFulfilled和onRejected是可传递的，直至被捕获

○ 所以建议使用：

xxx

.then(function() {})

.catch(function() {})

• Aside: 在全局对象上捕获异常

unhandledrejection

• 浏览器上实现

```
window.addEventListener('unhandledrejection', event => {
  const { reason, promise } = event

  console.log(reason, promise)
  // reason => Promise 失败原因, 一般是一个错误对象
  // promise => 出现异常的 Promise 对象

  event.preventDefault()
}, false)
```

### ◦ Node.js

```
process.on('unhandledRejection', (reason, promise) => {
  console.log(reason, promise)
  // reason => Promise 失败原因, 一般是一个错误对象
  // promise => 出现异常的 Promise 对象
})
```

◦ 不推荐使用；应在代码中明确地捕获异常

→ 静态方法

- `Promise.resolve(x)` 方法

◦ 快速把一个值转为 `resolve` 对象，并转为 `fulfilled` 的状态  
并直接返回那个 `x` 值作为 `then` 的参数。

◦ 相当于

```
new Promise(function(resolve, reject) {
  resolve(x);
})
```

◦ 相当于 (这是一个 `thenable` 的特殊情况) (了解即可)

`Promise.resolve()`

`then: function onFulfilled, onRejected()`

`onFulfilled('foo')`

`}`

`)`

◦ `then(function() {} ... {})`

- `Promise.reject()`

◦ 相对应

## → 并行执行

- Promise.all方法

- 当所有Promise对象完成后才结束，结果为数组

- 效果：多个Promise对象

- Promise.race方法

- 跟第一个完成的任务谁结束

- 效果：多个Promise对象

## → 执行顺序/宏任务 vs. 微任务

- 回调队列中执行任务称之为「宏任务」

- 宏任务执行过程中可以临时插入一些额外需求

- 可以选择作为一个新的宏任务放到队列中排队

- 可以作为当前任务的微任务（在当前任务执行结束后立即执行）

- Promise回调是作为微任务执行的

- 提高整体响应能力

- 目前强大数据来源都呈作宏任务执行

- Promise & MutationObserver process.nextTick => 微任务

## 14 15 16 Generator

### → 回顾Generator

- ES2015 / ES6

- 基本语法

- 声明：function \* fnName() { }

- 定值生成器：const generator = fnName()

- 执行：generator.next() // 执行多次

• 返回: function \* fnName() {

    yield 'number'

    }

    // { value: ..., done: true/false }

• 取出值: const result = generator.next(). // 没有 yield.

• 陷入生成器:

function \* fn() {

    const res = yield 'foo'

}

const generator = fn()

const result = generator('bar')

• 陷入生成器异常

generator.throw(new Error('异常'))

→ 举例

```
// Generator 配合 Promise 的异步方案

function ajax(url){
    // ...上面的笔记有这个Promise方法生成的AJAX
}

function * main(){
    const users = yield ajax('api/users.json')
    console.log(users)

    const post = yield ajax('api/posts.json')
}

const g = main()

const result = g.next()

result.value.then(data => {
    const result2 = g.next(data)

    if(result2.done) return

    result2.value.then(data => {
        g.next(data)
    })
})
```

→ 生成器函数执行器

• 库

<http://github.com/tj/co>

## • 自定义

```
function co(generator) {
  const g = generator()

  function handleResult(result) {
    if (result.done) return // 生成器函数结束
    result.value.then(data => {
      handleResult(g.next(data))
    }, error => {
      g.throw(error)
    })
  }

  handleResult(g.next())
}
```

## 17 Async和Await (最新技术) (ES2017)

→ 介绍/讲解

• Async是Generator更好的语法糖 (不需要co配合)

◦ 生成器函数 → Async函数

◦ yield → await

• 返回Promise对象, 无须对整体操作

• await只能出现在async内部

## 18 小测

→ 第一次

1.以下关于JavaScript的说法中正确的是：（） [分值：5]

您的回答：A.Javascript是单线程语言，方便进行DOM操作。 B.Javascript的异步操作常见的有计时器、事件绑定、Ajax  
✓ (得分：5)

2.下面哪些方法可以实现JavaScript异步编程？（） [分值：5]

您的回答：A.回调函数 B.事件监听 D.Promise对象 X → addEventListener  
正确答案为：A.回调函数|B.事件监听|C.发布/订阅|D.Promise对象

3.关于Promise对象的状态，下列说法错误的是：（） [分值：5]

您的回答：D.Rejected失败可以改变成Fulfilled成功 ✓ (得分：5)

4.关于Generator函数的描述，错误的是：（） [分值：5]

您的回答：C.Generator函数执行后得到的是一个生成器对象 X 还有个Promise对象，错了

正确答案为：D.使用return语句使Generator函数暂停执行，直到next方法的调用  
field

5.下列代码的输出结果 ()

```
1 console.log("1")
2 setTimeout(function () {
3   console.log("2")
4 }, 0)
5 console.log("3")          [分值: 5]
6 setTimeout(function () {
7   console.log("4")
8 }, 1000)
9 console.log("5")
```

您的回答: A.13524 ✓ (得分: 5)

6.以下代码块的输出结果 ()

```
1 const promise = new Promise((resolve, reject) => {
2   console.log(1)    }报错
3   resolve()
4   console.log(2)
5 })
6 promise.then(() => {
7   console.log(3)
8 })
9 console.log(4)
```

您的回答: A.1234 ✗

正确答案为: B.1243

7.下面程序的正确输出结果是 ()

```
1 Promise.resolve(1)
2   .then((res) => {
3     console.log(res)
4     return 2
5   })
6   .catch((err) => {      [分值: 5]
7     return 3
8   })
9   .then((res) => {
10    console.log(res)
11  })
```

您的回答: C.12 ✓ (得分: 5)

8.下列代码的执行结果为

```
1 const promise = new Promise((resolve, reject) => {
2   resolve("success1")
3   reject("error")
4   resolve("success2")
5 })
6 promise
7   .then((res) => {           [分值: 5]
8     console.log("then: ", res)
9   })
10  .catch((err) => {
11    console.log("catch: ", err)
12  })
```

您的回答: A.then: success1 ✓ (得分: 5)

9.下列代码的运行结果是 ()

Promise.resolve(1).then(2).then(Promise.resolve(3)).then(console.log) [分值: 5]

您的回答: D.console.log ✗

正确答案为: A.1

10.Generator函数的yield关键字的作用是: () [分值: 5]

您的回答: C.暂停执行, 等待生成器对象的next()方法调用 ✓ (得分: 5)

11.下列叙述中不正确的是: () [分值: 5]

您的回答: B.next()方法执行的参数是不会当作yield表达式的返回值。| C.next()方法的返回对象中的value是值, done是传递参数。| D.Generator函数的返回值是一个Generator对象。 ✗

正确答案为: B.next()方法执行的参数是不会当作yield表达式的返回值。| C.next()方法的返回对象中的value是值, done是传递参数。

→ 第二次.

大意错了二个

→ 第三次

全对.

## 任务三 手写Promise代码

### 1. Promise类核心逻辑实现

→ Promise就是一个类 在执行这个类的时候 需要传递一个执行器进去 执行器会立即执行

→ 三种状态

- Pending
  - fulfilled
  - rejected
- 一旦确定，不再更改。

→ resolve 和 reject 是用来更改状态的

- resolve fulfilled
- reject rejected

→ then方法内部会根据事态变化的状态。（定义在原型对象中）

• 如果是成功 调用成功回调函数

• 如果是失败 调用失败回调函数

→ then成功回调之后有一个参数，表示成功之后的值

then失败回调有一个参数 表示失败的原因。

```
const PENDING = 'pending'
const FULFILLED = 'fulfilled'
const REJECTED = 'rejected'

class MyPromise {
    constructor (executor){
        executor(this.resolve, this.reject)
    }

    status = PENDING
    value = undefined //成功之后的值
    reason = undefined //失败之后的值

    resolve = (value) => {
        if(status !== PENDING) return;
        this.status = FULFILLED
        this.value = value
    }

    reject = (reason) => {
        if(status !== PENDING) return;
        this.status = REJECTED
        this.reason = reason
    }

    then (successCallback, failCallback) {
        if (this.status === FULFILLED) {
            successCallback(this.value)
        }else if(this.status === REJECTED){
            failCallback(this.reason)
        }
    }
}

module.exports = MyPromise
```

## 2 在Promise中加入异步逻辑

```
→
const PENDING = 'pending'
const FULFILLED = 'fulfilled'
const REJECTED = 'rejected'

class MyPromise {
  constructor (executor){
    executor(this.resolve, this.reject)
  }

  status = PENDING
  value = undefined //成功之后的值
  reason = undefined //失败之后的值
  successCallback = undefined //成功的回调
  failCallback = undefined //失败的回调

  resolve = (value) => {
    if(status !== PENDING) return;
    this.status = FULFILLED
    value = this.value
    this.successCallback && this.successCallback(this.value) //如果成功回调存在 就调用
  }

  reject = (reason) => {
    if(status !== PENDING) return;
    this.status = REJECTED
    reason = this.reason
    this.failCallback && this.successCallback(this.reason)
  }

  then (successCallback, failCallback) {
    if (this.status === FULFILLED) {
      successCallback(this.value)
    }else if(this.status === REJECTED){
      failCallback(this.reason)
    }else{
      //等待 将成功回调和失败回调存储起来
      this.successCallback = successCallback;
      this.failCallback = failCallback;
    }
  }
}

module.exports = MyPromise
```

## 3 实现then方法多次调用添加多个处理器函数

```
→
class MyPromise {
  constructor (executor){
    executor(this.resolve, this.reject)
  }

  status = PENDING
  value = undefined //成功之后的值
  reason = undefined //失败之后的值
  successCallback = []//成功的回调
  failCallback = []//失败的回调

  resolve = (value) => {
    if(status !== PENDING) return;
    this.status = FULFILLED
    value = this.value
    //this.successCallback && this.successCallback(this.value)//如果成功回调存在 就调用
    while(this.successCallback.length){
      this.successCallback.shift()(this.value);
    }
  }

  reject = (reason) => {
    if(status !== PENDING) return;
    this.status = REJECTED
    reason = this.reason
    //this.failCallback && this.successCallback(this.reason)
    while(this.failCallback.length){
      this.failCallback.shift()(this.value);
    }
  }

  then (successCallback, failCallback) {
    if (this.status === FULFILLED) {
      successCallback(this.value)
    }else if(this.status === REJECTED){
      failCallback(this.reason)
    }else{
      //等待 将成功回调和失败回调存储起来
      this.successCallback.push(successCallback)
      this.failCallback.push(failCallback)
    }
  }
}
```

#### 4.5 实现then方法的链式调用

6. then方法链调用深入到Promise对象返回

7. 捕获错误及then链式调用其它状态代码

8. 将then方法的参数变成可选参数

9. Promise.all方法

10. Promise.resolve方法

11. finally方法

12. catch方法实现

1. 关于 then 方法, 以下选项中说法不正确的是? [分值: 5]

您的回答: C: then 方法的成功回调函数可以继续返回 Promise 对象, 失败回调函数中不可以 ✓ (得分: 5)

2. 执行器函数的 resolve 参数的作用是什么? [分值: 5]

您的回答: B: 创建 Promise 对象 ✗

正确答案为: C: 将 promise 状态更改为成功

3. 关于 Promise.all 方法以下说法不正确的是? [分值: 5]

您的回答: B: 参数为数组, 数组中只能传递返回 promise 对象的异步API ✓ (得分: 5)

4. 以下选项中关于 Promise 状态说法不正确的是? [分值: 5]

您的回答: C: 在 Promise 中状态确定后, 可以再次更改 ✓ (得分: 5)

5. 以下关于 finally 方法说法不正确的是? [分值: 5]

您的回答: C: finally 方法的返回值是 Promise 对象, 所以在调用方法后还可以链式调用 then 方法 ✗

正确答案为: A: 在 finally 方法中, 可以获取到 promise 对象的执行结果