模块2 串讲

函数式编程

- lodash 和 lodash/fp 模块

```
const _ = require('lodash')

let arr = [1, 2, 3, 4]
let r = _.reverse(arr)

_.map(data, callback)

// -------fp 模块，迭代优先/数据置后---------
const fp = require('lodash/fp')

let arr = [1, 2, 3, 4]
let r = fp.reverse(arr)

fp.map(callback, data)
```

- 函子在开发中的实际使用场景
  - 作用是控制副作用 (IO)、异常处理 (Either)、异步任务 (Task)

```
class Functor {
  constructor (value) {
    this._value = value
  }

  map (f) {
    return new Functor(f(this._value))
  }

  value (f) {
    return f(this._value)
  }
}

const toRMB = s => new Functor(s)
    .map(s => s.replace(/\$/, ''))
    .map(parseFloat)
    .map(n => n / 7)
    .value(x => x.toFixed(2))

console.log(toRMB('$299.99'))
```

  - folktale
    - https://folktale.origamitower.com/

```
const Maybe = require('folktale/maybe')

```

```
// const data = [{ name: 'zs', age: 20 }]
// const user = data.find(u => u.name === 'tom')
// let age = 'No Age'
// if (user && user.age) {
//   age = user.age
// }
// console.log(age)

const data = [{ name: 'zs', age: 20 }]
const age = Maybe.fromNullable(data.find(u => u.name === 'zs'))
  .map(u => u.age)
  .getOrElse('No Age')
console.log(age)
```

- 对函数编程整体还是比较模糊 需要补充哪些基础知识理念
  - 纯函数
  - 柯里化
  - 函数组合
- 柯里化概念意义和用法
  - 柯里化：把多个参数的函数转换成只有一个参数的函数，可以给函数组合提供细粒度的函数
  - 应用：
    - Vue 源码中使用柯里化的位置
      - src/platform/web/patch.js

```
function createPatch(obj) {
  return function patch(vdom1, vdom2) {
    ..
  }
}

const patch = createPatch(...)


patch
```

    - 固定不常变化的参数

```
// 方法1
function isType (type) {
  return function (obj) {
    return Object.prototype.toString.call(obj) === `[object ${type}]`
  }
}

const isObject = isType('Object')
const isArray = isType('Array')


// 方法2
function isType (type, obj) {
```

```
  return Object.prototype.toString.call(obj) === `[object ${type}]`
}

let isTypeCurried = curry(isType)

const isObject = isTypeCurried('Object')
// isObject(obj)

// 柯里化通用函数
function curry (func) {
  return function curriedFn(...args) {
    // 判断实参和形参的个数
    if (args.length < func.length) {
      // 实参个数小于形参个数，继续柯里化
      return function () {
        return curriedFn(...args.concat(Array.from(arguments)))
      }
    }
    // 形参个数等于实参个数执行 func
    return func(...args)
  }
}
```

- 延迟执行(模拟 bind 方法)

```
// rest 参数
Function.prototype.mybind = function (context, ...args) {
  return (...rest) => this.call(context, ...args, ...rest)
}

function t (a, b, c) {
  return a + b + c
}

const sumFn = t.mybind(this, 1, 2)
const sum = sumFn(3)
console.log(sum)
```

- 请扩展讲下 Lodash 中 chain tap thru 3个函数

```
var users = [
  { 'user': 'barney',  'age': 36 },
  { 'user': 'fred',    'age': 40 },
  { 'user': 'pebbles', 'age': 1 }
]
// chain 链式调用
let youngest = _.chain(users)
  .sortBy('age')
  .map(user => user.user + 'is ' + user.age)
  .head()
  .value()

console.log(youngest)

// tap 方法在链式调用过程中插入一个拦截器，修改当前的 value
```

```
let r = _.chain(users)
  .tap(users => {
    users.push({ 'user': 'tom', 'age': 28 })
  })
  .sortBy('age')
  .value()

console.log(r)

// thru 方法的拦截器中改变当前的 value 并返回，用于后续的链式调用
let r = _.chain(users)
  .thru(value => {
    value.push({ 'user': 'tom', 'age': 28 })
    return value
  })
  .sortBy('age')
  .value()

console.log(r)
```

- 显示用 chain 链式调用和隐世链式调用是否都是惰性调用？还有什么具体区别？印象里出现过链式上有某些方法的时候必须用 chain 函数，要不就报错，但是想不起来了。

```
const _ = require('lodash')

let employees = [
  { name: 'Jack', age: 25, salary: 20000 },
  { name: 'Tom', age: 30, salary: 30000 },
  { name: 'Jim', age: 26, salary: 25000 },
  { name: 'Carl', age: 25, salary: 22000 },
  { name: 'Abel', age: 32, salary: 32000 },
  { name: 'Gary', age: 31, salary: 28000 },
  { name: 'Kevin', age: 23, salary: 27000 }
]
// 显示链式调用，接触链式调用必须使用 value 方法，获取链式调用的结果
let salary = _.chain(employees)
  .filter(e => e.age >= 30)
  .map(e => e.salary)
  .sum()
  .value()

let salary = _(employees)
  .chain()
  .filter(e => e.age >= 30)
  .map(e => e.salary)
  .sum()
  .value()

console.log(salary)

// 隐式链式调用，调用返回单个值的方法，会自动解除链式调用，否则需要调用 value 方法
// 隐式链式调用依然是惰性求值，当隐05-式或者显示调用了 value 方法才会执行计算
let salary = _(employees)
  .filter(e => e.age >= 30)
  .reduce((sum, e) => sum += e.salary, 0)

let salary = _(employees)
```

```
  .filter(e => e.age >= 30)
  .map(e => e.salary)
  .sum()

console.log(salary)
```

- **Chain函数的惰性调用是否就是一个函子包含一个 flowRight 函数，此函数包含我所有链式上的方法?**

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

class MyWrapper {
  constructor (value) {
    this._wrapped = value
    this._actions = []
  }

  chain (value) {
    this._wrapped = value
    return this
  }

  filter (fn) {
    this._actions.push(fp.filter(fn))
    return this
  }

  map (fn) {
    this._actions.push(fp.map(fn))
    return this
  }

  sum () {
    this._actions.push(fp.sum)
    return this
  }

  value () {
    let fn = fp.compose(...this._actions.reverse())
    return fn(this._wrapped)
  }
}

let _ = {
  chain: function chain (value) {
    return new MyWrapper(value)
  }
}

let employees = [
  { name: 'Jack', age: 25, sex: 'male', salary: 20000 },
  { name: 'Tom', age: 30, sex: 'male', salary: 30000 },
  { name: 'Jim', age: 26, sex: 'male', salary: 25000 },
  { name: 'Carl', age: 25, sex: 'male', salary: 22000 },
  { name: 'Abel', age: 32, sex: 'male', salary: 32000 },
```

```
  { name: 'Gary', age: 31, sex: 'male', salary: 28000 },
  { name: 'Kevin', age: 23, sex: 'male', salary: 27000 }
]

let salary = _.chain(employees)
  .filter(e => e.age >= 30)
  .map(e => e.salary)
  .sum()
  .value()
console.log(salary)
```

- **请问如果函数有3个或3个以上参数，应该怎样在函数式编程中处理，是必须使用io函子处理？还是先柯里化后再怎么做？**

```
fn(a, b, c)
```

- 想了解下 Task 函子的具体实现
  - https://github.com/origamitower/folktale/tree/master/packages/base/source/concurrency

JavaScript 性能优化

- 性能优化回顾
  - 我们讲的是 JavaScript 语言本身的一些内容，课程中主要介绍的是内存管理、垃圾回收机制、常见的GC算法。你相对语言本身进行优化，得先了解这些概念。
  - 至于 Web 性能优化，在最后一个阶段会讲
- 希望扩展下垃圾回收、V8
- 《垃圾回收的算法与实现》
- 新生代存储区回收机制， 复制算法+标记整理，很多相关文档说并没有使用标记整理；
- 上面的书中对标记整理有相关介绍，然后 V8 引擎中使用了标记整理对垃圾回收做了优化
- JS性能优化那一块，只讲了怎么去优化，但是并没有讲为什么这样去优化，原理是什么？例如使用遍历方法的时候为什么foreach就比其它两种要快，为什么使用直接使用方法字面量去申明的对象要比使用new更快？
  - https://jsperf.com/forEach-for-demo
  - https://jsperf.com/object-new-Object

```
[1,2,  ,3]
arr.forEach(() => {

})

07
for (let i = arr.length; i >= 0; i--) {

}
```

```
10
11
12 var o = {}
13
14 var o = new Object()
```

- 函数的活动对象
  - 执行上下文（Execution Context）
    - 全局执行上下文
    - 函数级执行上下文
    - eval 执行上下文
  - 函数执行的阶段可以分文两个：函数建立阶段、函数执行阶段
    - 函数建立阶段：当调用函数时，还没有执行函数内部的代码
      - 创建执行上下文对象

```
1 fn.EC = {
2   variableObject:  // 函数中的 arguments、参数、局部成员
3   scopeChains:  // 当前函数所在的父级作用域中的活动对象
4   this: {}      // 当前函数内部的 this 指向
5 }
```

    - 函数执行阶段

```
1 // 把变量对象转换为活动对象
2 fn.EC = {
3   activationObject:  // 函数中的 arguments、参数、局部成员
4   scopeChains:  // 当前函数所在的父级作用域中的活动对象
5   this: {}      // 当前函数内部的 this 指向
6 }
```

  - [[Scopes]] 作用域链，函数在创建时就会生成该属性，js 引擎才可以访问。这个属性中存储的是所有父级中的变量对象

- 介绍下 Performance 相关的工具、库或更方便的性能监控工具，菜鸟只看懂了可以用开发者工具监控分析性能，感觉有更专业的工具可以替换手动操作
  - https://developer.mozilla.org/zh-CN/docs/Web/API/Performance
  - https://developers.google.com/speed/pagespeed/insights/
  - lighthouse

其他问题

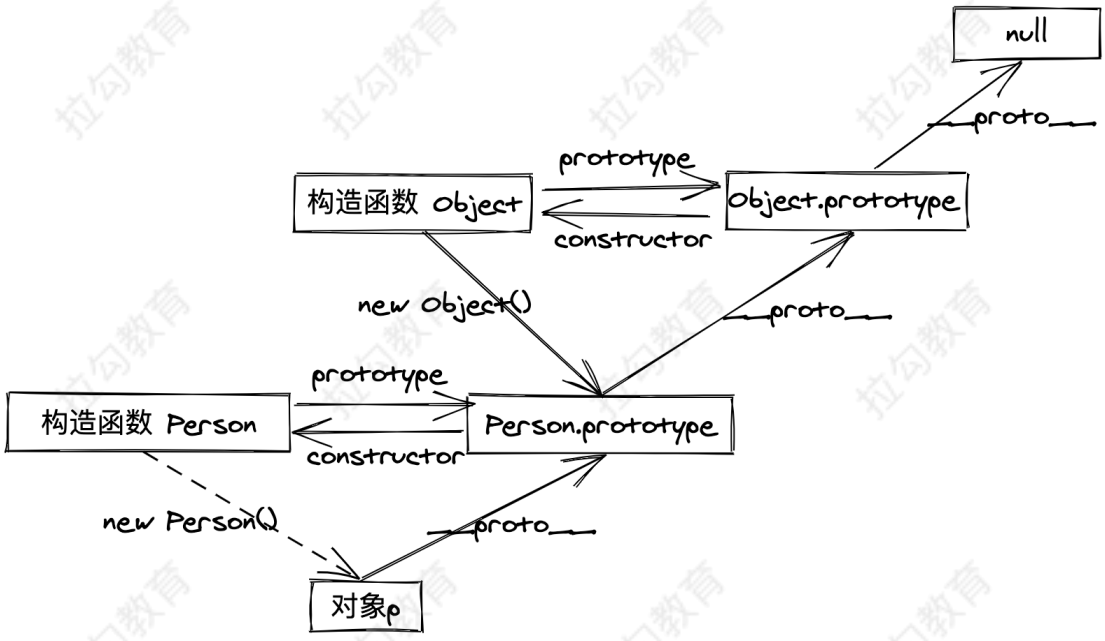
- 闭包：函数作为参数为什么会产生闭包？？？希望能举个例子

```
function fn (08-) {
  var name = 'zs'
  return function () {
    callback(name)
  }
}

let func = fn (function (name) {
  console.log(name)
})

func()
```

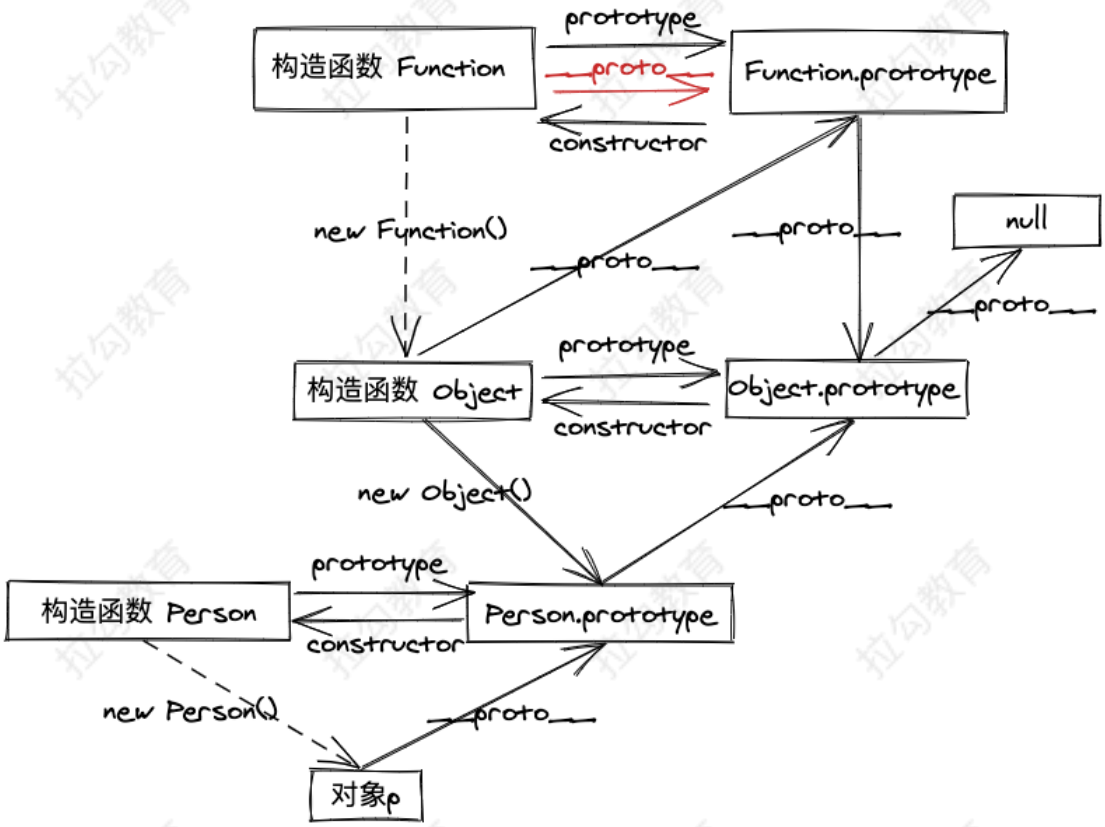
- 仍然希望扩展关于原型原型链构造函数之间的关系
  - 面试题

```
var Person = function () {}
var p = new Person()
// instanceof: 检测构造函数的 prototype 是否出现在对象的原型链上
console.log(p instanceof Object)
console.log(p instanceof Person)
console.log(p instanceof Function)
```

  - 对象的原型链



  - 完整版原型链



- 平时我们使用的都是vue，react项目，课程中的gulp等自动化工作流程对于我们有什么用处
  - gulp：基于流的自动化构建工具，使用 gulpfile 配置文件设置不同的任务，不支持模块化
    - 在 Web 领域主要是对文件进行压缩
    - 综合领域，更强大，需要自己编写任务
    - 比如：electron 的应用，需要签名、需要发布。前后端没有分离的项目
  - webpack：基于模块的自动化构建工具，模块化打包器。基于 loader 对模块处理。
    - 对文件进行压缩
    - 把多个模块打包成单一文件
    - 强大的插件机制，导致我们感觉它无所不能
    - Web 形式的最主要构建工具，最主要的任务是打包

防抖和节流

https://www.lodashjs.com/docs/lodash.debounce09-

debounce (防抖)

把多次函数调用合并成一次，可以用于防止快速按键、快速拖动窗口大小，导致事件处理函数内的复杂操作重复被执行多次降低用于体验

```
// lodash 中的 debounce
// 自己实现一个 debounce 函数
function debounce (fn, delay) {
  let timer = null
  return function (...args) {
    clearTimeout(timer)
    timer = setTimeout(() => {
      fn.apply(this, args)
      timer = null
    }, delay)
  }
```

```
}

function add(n1, n2) {
  console.log(n1 + n2)
}

let fn = debounce(add, 300)

fn(1, 2)
fn(1, 2)
fn(1, 2)
```

throttle (节流)

在一定时间内让指定函数只执行一次，可以用于快速滚动的过程中，检测滚动位置距离底部多远。（要在滚动过程中检测位置是否到达底部，而 debounce 是在事件结束之后才会执行）

```
function add(n1, n2) {
  console.log(n1 + n2)
}

const fn = throttle(add, 3000)

fn(1, 2)
fn(1, 2)
fn(1, 2)
fn(1, 2)

function throttle(fn, delay) {
  let timer = null
  let pre = null

  return function (...args) {
    let now = Date.now()

    if (pre) {
      if (now - pre >= delay) {
        fn.apply(this, args)
        pre = now
      } else {
        if (timer) return
        timer = setTimeout(() => {
          fn.apply(this, args)
          pre = now
          timer = null
        }, delay)
      }
    } else {
      fn.apply(this, args)
      pre = now
    }
  }
}
```

lodash 导出单独防抖和节流函数

- 打开终端，切换到项目中 lodash 的源码目录
- 输入以下命令

```
npm i -g lodash-cli

lodash include=debounce,throttle
```