7.JS的编程技巧

学习高阶编程总结

模块化编程

单例设计模式

对象的另一个作用: 把描述相同事物的方法和属性,归纳到同一个对象下, 可起到防止变量污染的作用

```
// 基于对象防止变量污染
```

```
var person = {
  name: 'link',
```

```
var name = 'link'
var age = 24
```

```
  age: 24
}
```

实现模块间的相互调用

1. 通过全局对象挂载 window.xxxx (挂在过多,仍然会造成变量污染的问题)
2. 基于闭包 + 单例模式 ⇒ 高级单例模式 (早期的模块化思想)

由于闭包, 会产生内存占用的问题, 所以这种模式也不太好.

```
// 基于闭包防止变量污染
let module = (function () {
  let wd = ''
  function query () {//...}
  return {
    query
  }

})()

let module2 = (function () {
    module.query()

})()
```

1. **webpack环境**是支持一下两种规范的, 打包后的内容, 可以放在浏览器下运行
2. **纯node环境**是不支持ES6Module规范的
3. 纯浏览器环境两者都不支持, 但是有window对象

commonJS

基于commonJS规范导出的模块, 也可以基于ES6Module导入

```
// a.js
moudle.exports = {
```

```
  fn () {}

}
// b.js
let a = require('./a.js')
fn()
// ES6Module
import a from './a.js'
```

ES6Module

基于ES6Module规范导出的模块, 基于commonJS导出会有问题

```
// a.js
fn(){}
export default = {
  fn,
}
// b.js
import b from './a.js'
let b = require('./a.js') // 用commonJS的方式导入会有问题 如下
```

高级函数编程技巧

惰性函数

惰性函数: 能只执行一次的代码,绝不执行多次.

```
// 标准:getComputedStyle
// IE6~8:currentStyle
// 常规情况:
// 这样的话就必须每次调用该函数就判断是否兼容新方法
let box = document.querySelector('.box')

let compatible = typeof getComputedStyle !== undefined ? true : false;
let getCss = function (element, attr) {
  if (compatible) {
    return window.getComputedStyle(element)[attr]
  }
  return element.currentStyle[attr]

}
```

```
// 通过惰性函数思想
// 首次执行函数判断完浏览器兼容器后即不再需要判断
let getCss = function (element, attr) {
  if (typeof getComputedStyle !== undefined) {
    getCss = function (element, attr) {
      return window.getComputedStyle(element)[attr]
    }
  } else {
    getCss = function (element, attr) {
      return element.currentStyle[attr]
    }
  }
  // 保证第一次执行
  return getCss(element, attr)

}
console.log(getCss(box, 'width'));
console.log(getCss(box, 'backgroundColor'));
console.log(getCss(box, 'height'));
```

柯理化函数

函数柯里化: 闭包的高阶引用, 预处理/预存储, 通过形成闭包,存储变量或者数据, 供下级上下文调用的都叫做柯理化思想

```
// 面试题
let add = curring();
let res = add(1)(2)(3);
console.log(res); //->6

add = curring();
```

```
res = add(1, 2, 3)(4);
console.log(res); //->10

add = curring();
res = add(1)(2)(3)(4)(5);
console.log(res); //->15

// curring 不指定参数
const curring = () => {
  let arr = []
  const add = (...params) => {
    arr = arr.concat(parmas)
    return add;
  }
  add.toString() = () => {
    return arr.reduce( (total, item) =>{
      return total + item
    })
  }
  return add;
}
```

```
// 指定参数
let add = curring(5);
res = add(1)(2)(3)(4)(5);
console.log(res); //->15

const curring = n => {
  let arr = [],
      index = 0
  const add = (...params) => {
    index++
    arr = arr.concat(params)
    if (index >= n) {
      return arr.reduce( (total, item) => {
        return total + item;
      })
    }
    return add
  }
  return add
}
```

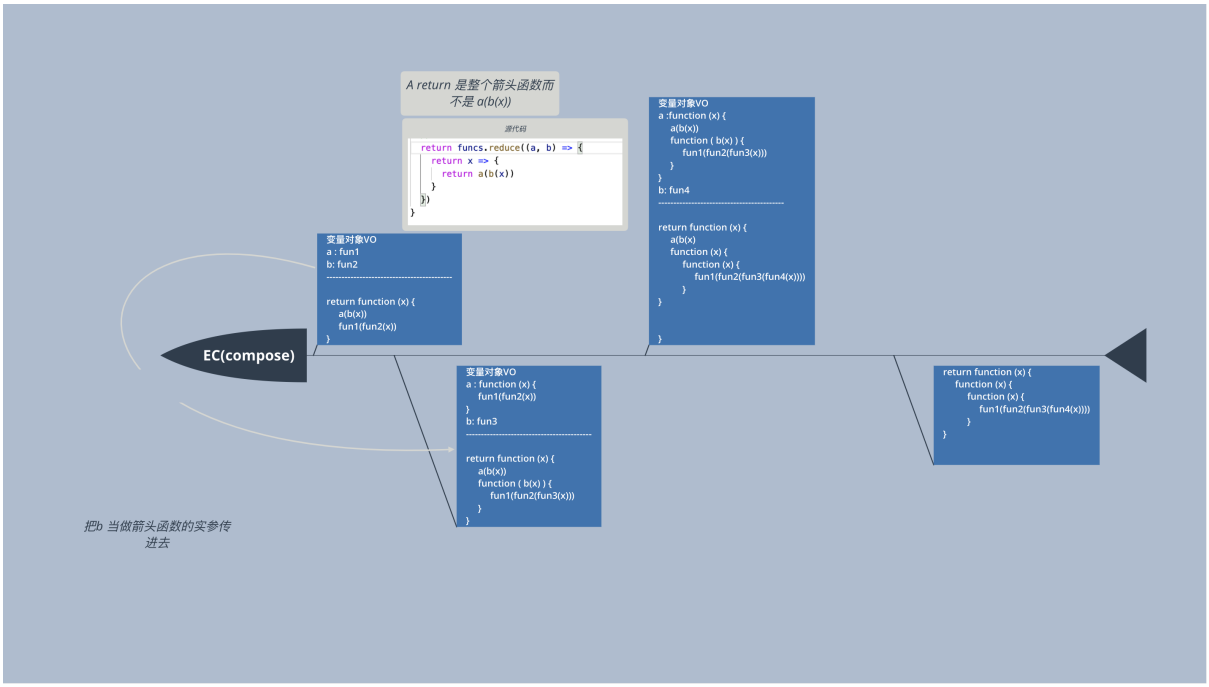
组合函数

compose函数: 即管道函数, 就是把要处理数据的函数串成一条管道, 让数据穿过所有的函数

```
/*
    在函数式编程当中有一个很重要的概念就是函数组合， 实际上就是把处理数据的函数像管道一样连接起来，然后让数据穿过管道得到最终的结果。 例如：
    const add1 = x => x + 1;
    const mul3 = x => x * 3;
    const div2 = x => x / 2;
    div2(mul3(add1(add1(0)))); //=>3

    而这样的写法可读性明显太差了，我们可以构建一个compose函数，它接受任意多个函数作为参数(这些函数都只接受一个参数)，然后compose返回的也是一个函数，达到以下的效果：
    const operate = compose(div2, mul3, add1, add1)
    operate(0) //=>相当于div2(mul3(add1(add1(0))))
    operate(2) //=>相当于div2(mul3(add1(add1(2))))

    简而言之：compose可以把类似于f(g(h(x)))这种写法简化成compose(f, g, h)(x)，请你完成 compose函数的编写
*/
const add1 = x => x + 1
const mul3 = x => x * 3
const div2 = x => x / 2
```




```
// redux 源码的形式
/*
  1. compose会返回一个函数
  2. 没有传处理函数,或者只传一个的情况(*)
*/
const compose = (..funcs) => {
  if (funcs.length === 0) return x => x  // (*)
  if (funcs.length === 1) return func[0] // (*)

  // div(mul3(add1(add1(x))))
```

```
  return funcs.reduce((a, b) => {
    return x => {
      return a(b(x))
    }
  })
}
const operate = compose(div2, mul3, add1, add1)
operate(0) //=>相当于div2(mul3(add1(add1(0))))
operate(2) //=>相当于div2(mul3(add1(add1(2))))
```

```
// 这个方法比redux更优, 因为不会创造那么多作用域
const compose = (...funcs) => {
  if (funcs.length === 0) return x => x  // (*)
  if (funcs.length === 1) return func[0] // (*)

  // div(mul3(add1(add1(x))))
  return (...args) => {
    return funcs.reduceRight(result,item => {
      if (Array.isArray(result)) {
        return item(...result)
      }
      return item(result)
    },args)
  }

}
```

reduce 源码解析

```
Array.prototype.reduce = function redunce(callback, initial) {
  /*
    将处理结果作为后一项的参数

  */
  let self = this,
    len = self.length,
    i = 0,
    result
  if (typeof callback !== 'function') throw new TypeError('callback must be a function')
  if (typeof initial === 'undefined') {
    result = self[0]
    i = 1
  } else {
    result = initial
  }

  for (; i < len; i++) {
    item = self[i]
    result = callback(result, item)
```

```
  }
  return result
}
let arr = [10, 20, 30, 40]
console.log(
  arr.reduce((result, item, index) => {
    return result + item
  })
)
console.log(
  arr.reduce((result, item) => {
    return result + item
  }, 0)
)
```

call原理分析

bind原理分析