## 设计原则SOLID

S：单一职责 一个函数或接口只实现一个功能 代码拆分

O：封闭开放原则 对修改封闭 扩展开放。有新需求，原始代码不动，增加新代码

L：李式置换原则，子类覆盖父类，父类有的 子类一定有

I：接口独立

D：依赖倒置原则

# 常用模式

设计模式是解决某个特定场景下对某种问题的解决方案。分穿创建型、行为型、结构型

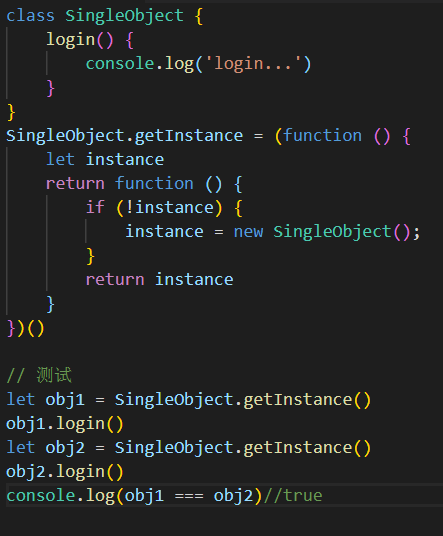
## 工厂模式

**创建者和构造函数分离，对构造函数进行封装：**



## 单例模式

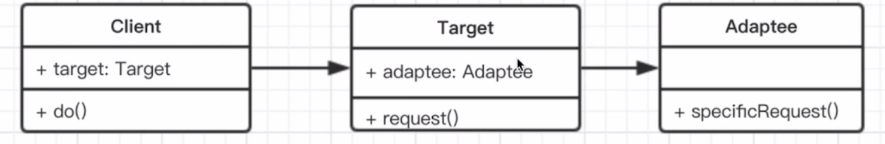
**系统中被唯一使用，一个类只有一个实例，一旦存在，直接返回实例而不在新建，常见用于购物车、登录窗、状态 store。由于js没有private，所以可以通过模块化处理，定一个class 和getInstance函数，然后只导出getInstance函数，不导出class。**



## 适配器模式

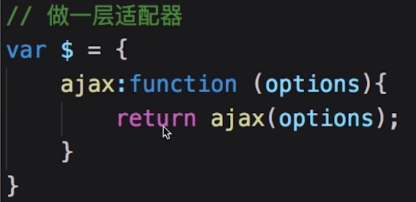
**将一个类的接口适配成用户所期待的，这就是适配器。替换老项目中 jquery 的 $.ajax ，vue中计算机属性 computed 的使用，都可以看作是适配器模式。旧接口格式和使用者不兼容，中间加一个适配转换接口。常用于新老接口不兼容，可封装旧接口，或者使用旧接口的名字和使用方法，但里面已经封装成新功能。原则：将旧接口和使用者进行分离，符合开放封闭原则，不修改旧接口而是新增一个适配器。**





用户----适配器-----老的接口（待适配的类或者是接口）

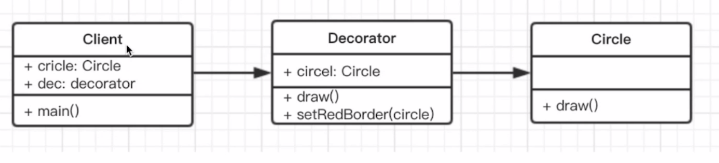




## 装饰器模式

**为对象添加新功能 而不改变其原有的结构和功能，将现有的对象和装饰器进行分离解耦。如react-redux的connnect、部分HOC、包括ES7装饰器、core-decorators模块。**



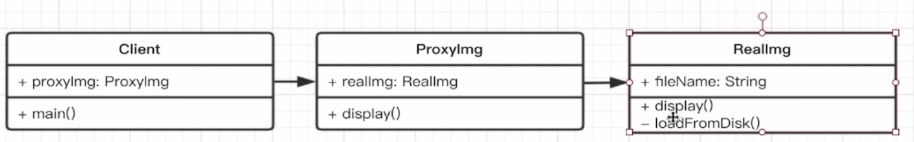


用户----装饰器-----原始类（待扩展的类）

|  |  |
| --- | --- |
|  |  |

## 代理模式

**使用者无权访问目标对象，中间加代理，通过代理做授权和控制。网络代理，科学上网，明星经纪人，es6的 proxyl、dom事件代理**



|  |  |
| --- | --- |
|  |  |

* **代理模式说明2**

为一个对象提供一个代用品或占位符，以便控制对它的访问。常用的虚拟代理形式：某一个花销很大的操作，可以通过虚拟代理的方式延迟到这种需要它的时候才去创建（例：使用虚拟代理实现图片懒加载）



图片懒加载的方式：先通过一张loading图占位，然后通过异步的方式加载图片，等图片加载好了再把完成的图片加载到img标签里面。

**代理模式 VS 适配器模式**

适配器：提供一个不同的接口（如不同的版本的插头）、一般原有功能不能用了

代理模式：提供一模一样的接口

**代理模式 VS 装饰器模式**

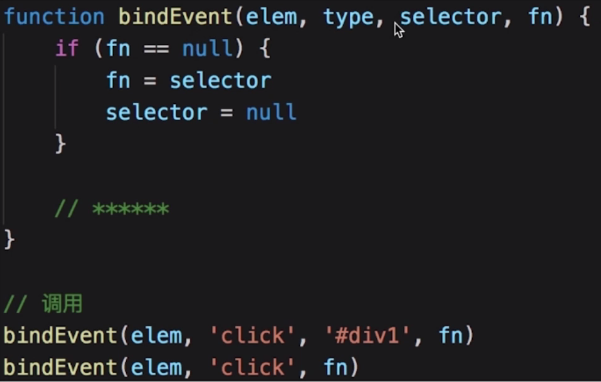
装饰器器：扩展功能，原有功能不变可直接使用

代理模式：显示原有功能，但是经过限制或者阉割之后

## 外观模式

|  |  |
| --- | --- |
|  |  |

**外观模式不符合单一接口原则，出现了“胖接口”**



## 观察者模式

发布订阅，一对N，主题发布者和观察者分离，不是主动触发而是被动监听，二者解耦

点咖啡，做好被叫

实例：**网页事件绑定、redux 、promise、 jquery、 callback**

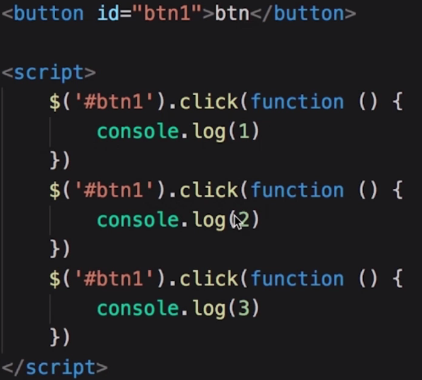
观察者：先从发布者中订阅，等待发布者发布状态，然后自己更新 等待被触发

发布者，在数据更新时候，触发所有订阅者进行更新回调

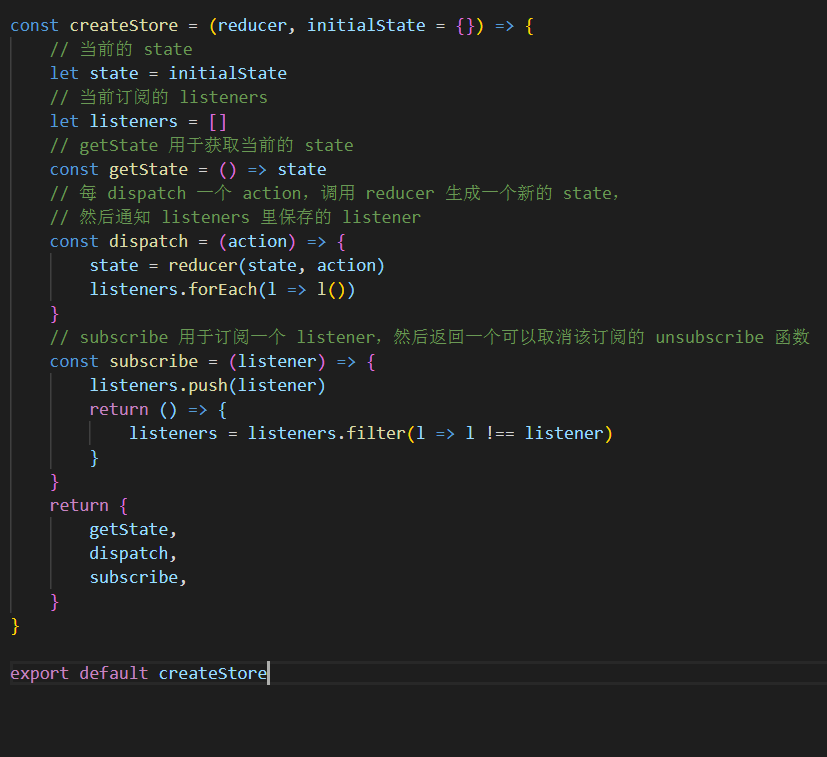


|  |  |
| --- | --- |
|  |  |

* **网页事件绑定**



* **Redux也是观察者模式**



* **Promise**

|  |  |
| --- | --- |
|  |  |

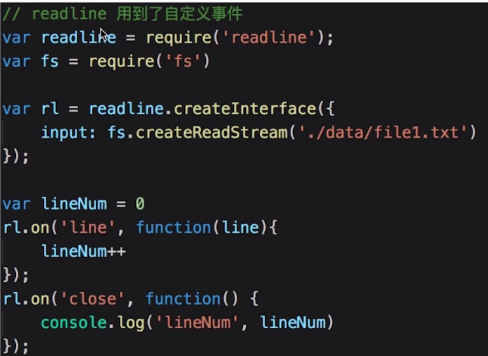
* **Jquery 的 callback**



* **NodeJS 自定义事件**

|  |  |
| --- | --- |
|  |  |





**其他实例：**

**Nodejs中：处理 http 请求；多进程通讯 req res**

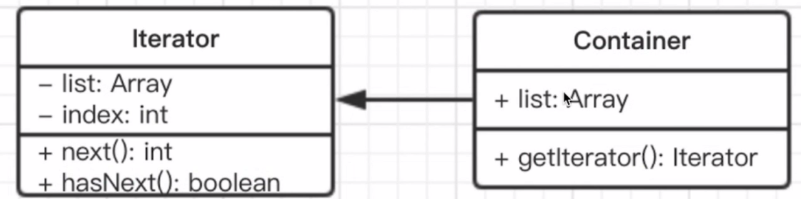
**Vue 和 react 组件的生命周期触发**

**Vue watch**

**Html5 new worker postMessage onMessage**

## 迭代器模式

**顺序访问一个集合，使用者无需知道集合内部结构（封装），从而实现迭代器对象与目标对象分离。 es6数组、类数组有内置的迭代：Symbol.iterator Array String Set Object.keys/values/entries for...of nodeList 用一个统一的接口实现不同类型的有顺序集合的遍历。Array.prototype[Symbol.iterator] 有一个next方法。**



|  |  |
| --- | --- |
|  |  |

**Data可以是数组或是类数组，只要其内置迭代器Symbol.iterator 就可以用它进行遍历，迭代器有next函数，next执行返回{values：’值’，done:bool}**

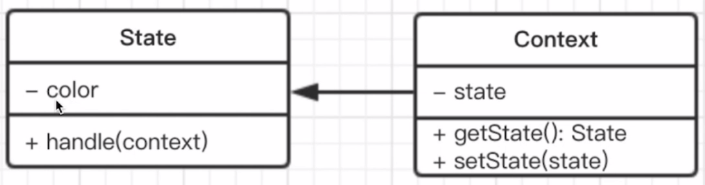


**上述也可以用 for of 取代。\*\*generator 返回的数据 符合Interator接口的要求 es6还有 function \*fn（）{yield value;yield value2} for of fn() 不过感觉要过时了**

## 状态模式

一个对象有状态变化，每次状态变化都会触发一个逻辑，不能总用if…else来控制。能将状态对象和主题对象分离，状态的变化逻辑单独处理。红绿灯，**有限状态机**、promise

状态变化（pending—>fulfilled pending—>rejected）做**promise的简单实现。**



* **红绿灯**

|  |  |
| --- | --- |
|  |  |

Context 是共用的 用来保持和存储当前的状态，一共有红绿黄三个状态，每个状态都有一个切换动作。状态切换会触发主体状态更改。实现状态以及状态管理的分离 多个状态共用一个状态管理和存储。

* **有限状态机 github javascript-state-machine 包**

|  |  |
| --- | --- |
|  |  |

**类似红绿灯，也是有不同的状态 transition，每一个状态变化都有自己的状态变化时触发的函数 method（监听，属观察模式）。状态管理为放在 fsm 的 state 上，处理函数抽离出来为 updateText**

|  |  |
| --- | --- |
|  |  |

**Then要return this 才能连缀。Promise的实现需要使用观察者模式以及状态模式 this.successList.push就是一个订阅，然后在状态变化时候，触发状态变化的setState执行，通过参数把订阅的函数 数组传递过去，执行所有订阅的函数。**

# 其他模式

## 原型模式

重新创建一个对象开销比较大，可以根据用已有的对象克隆出一个新对象。

* **如Object.create(prototype)**

|  |  |
| --- | --- |
|  |  |

* **对比JS中的原型prototype**

1. Prototye 可以理解为ES6 class 的一种底层原理
2. 而class是实现面向对象的基础，而不是服务于某个模式

## 桥接模式

**用于把抽象化和实现化解耦，使得二者可以独立变化。**

|  |  |
| --- | --- |
|  |  |

将画图和形状混合在一起，不利于抽象。

|  |  |
| --- | --- |
|  |  |

## 组合模式

**生成树形结构，表示“整体—部分”关系，让整体和部分都具有一致的操作型。JS经典应用中，未找到这么复杂的数据类型，虚拟dom中的VNode是这种形式或者文件夹系统，但数据结构简单。**

|  |  |
| --- | --- |
|  |  |

整体和单个节点的操作是一致的，整体和单个节点的数据结构也保持一致。将整体和单个节点的操作抽象出来

## 享元模式

**共享内存（主要是考虑内存，而非效率），相同的数据，共享使用，将相同的部分抽象出，符合开放封闭原则。类似例子（不是很符合，一定程度上可以做例子），如dom事件代理、sso、node中间件**



## 策略模式

不同的策略分开处理，避免出现大量的if…else或者switch…case。策略模式的目的就是将算法的使用算法的实现分离开来。

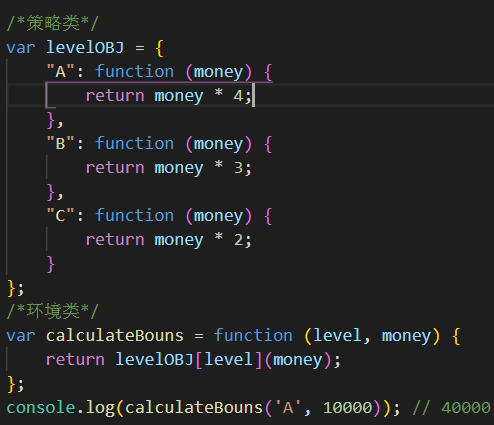
* **说明1**

左边是普通代码，右边是策略模式，每个策略有自己的实现逻辑，不同策略分开处理。

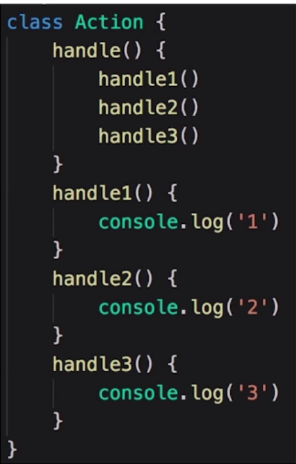
|  |  |
| --- | --- |
|  |  |

* **说明2**

策略模式的目的就是将算法的使用算法的实现分离开来。一个基于策略模式的程序至少由两部分组成。第一个部分是一组策略类（可变），策略类封装了具体的算法，并负责具体的计算过程。第二个部分是环境类Context（不变），Context接受客户的请求，随后将请求委托给某一个策略类。要做到这一点，说明Context中要维持对某个策略对象的引用。



## 模板方法模式



## 职责连模式

一步操作可能分为多个职责角色来完成（一步一个操作），然后把这些角色都分开，然后用一个链串起来。将发起者好各个处理者进行隔离（处理者之间也进行隔离）

|  |  |
| --- | --- |
|  | 每个角色有自己的处理函数，每一个角色都通过一个nextAction,来链接，然后只要最开始执行 handle 时候，整个链条都会顺序执行。类似有jq的链式操作、promise、node 的流操作 pipe，数据结构中的链表 |

## 命令模式

**执行命令时，发布者和执行者分开，中间加入命令对象，作为中转站。如进攻号角，鸣金收兵就是将军和士兵之间的中转站**

|  |  |
| --- | --- |
|  | 当然这边可以让Command的传入参数是一个数组Array<solder>,类似发布订阅。 |

例子：网页富文本编辑器操作，浏览器封装了一个命令对象

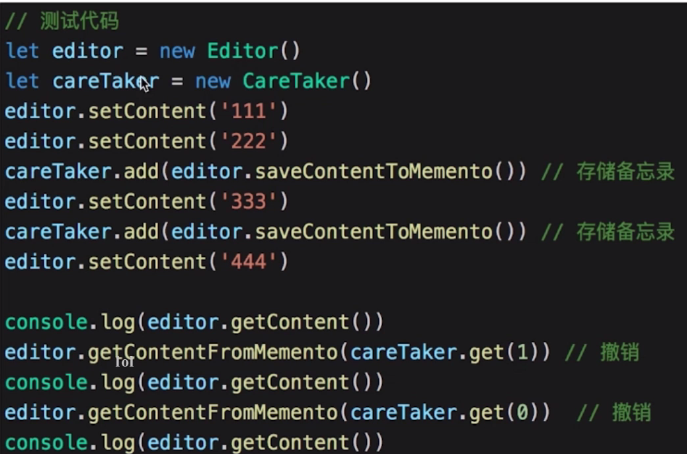
document.ExecCommand(“bold”) document.ExecCommand(“undo”)

markdown组件

## 备忘录模式

随时记录一个对象的状态变化，随时可以恢复之前的某个状态（如撤销功能），未找到JS中的经典应用，除了一些工具（如编辑器）

|  |  |
| --- | --- |
|  |  |

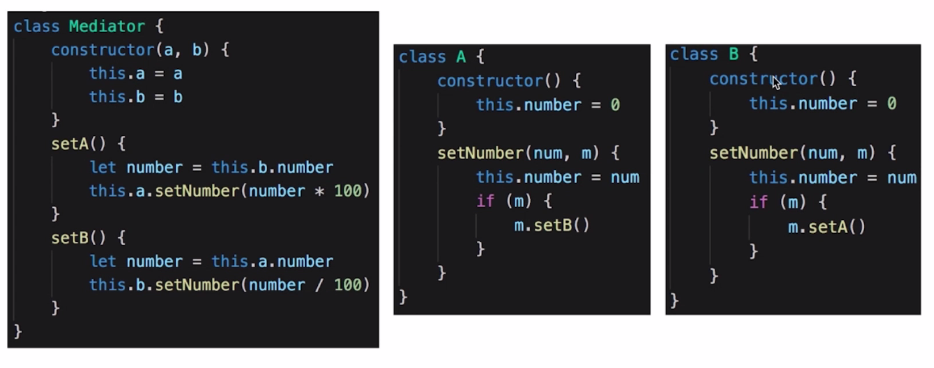


有内容编辑器 Editor ，将内容生成一个备忘录 Memento ，存储备忘录的列表 CareTaker 三者各自分离，各司其职。

## 中介者模式

**通过一个中介者对象，其他所有的相关对象都通过该中介者对象来通信，而不是相互引用，当其中的一个对象发生改变时，只需要通知中介者对象即可。通过中介者模式可以解除对象与对象之间的紧耦合关系。类似早期拔打电话，买房中介，飞机与塔台。**

|  |  |
| --- | --- |
|  |  |



A与B之间不直接交流，要想交流就通过 Mediator ，Mediator 维护两者的信息（中介和A B是有交集通信的），以及对应的逻辑

## 访问者模式

**将数据操作和数据结构进行分离**

## 解释器模式

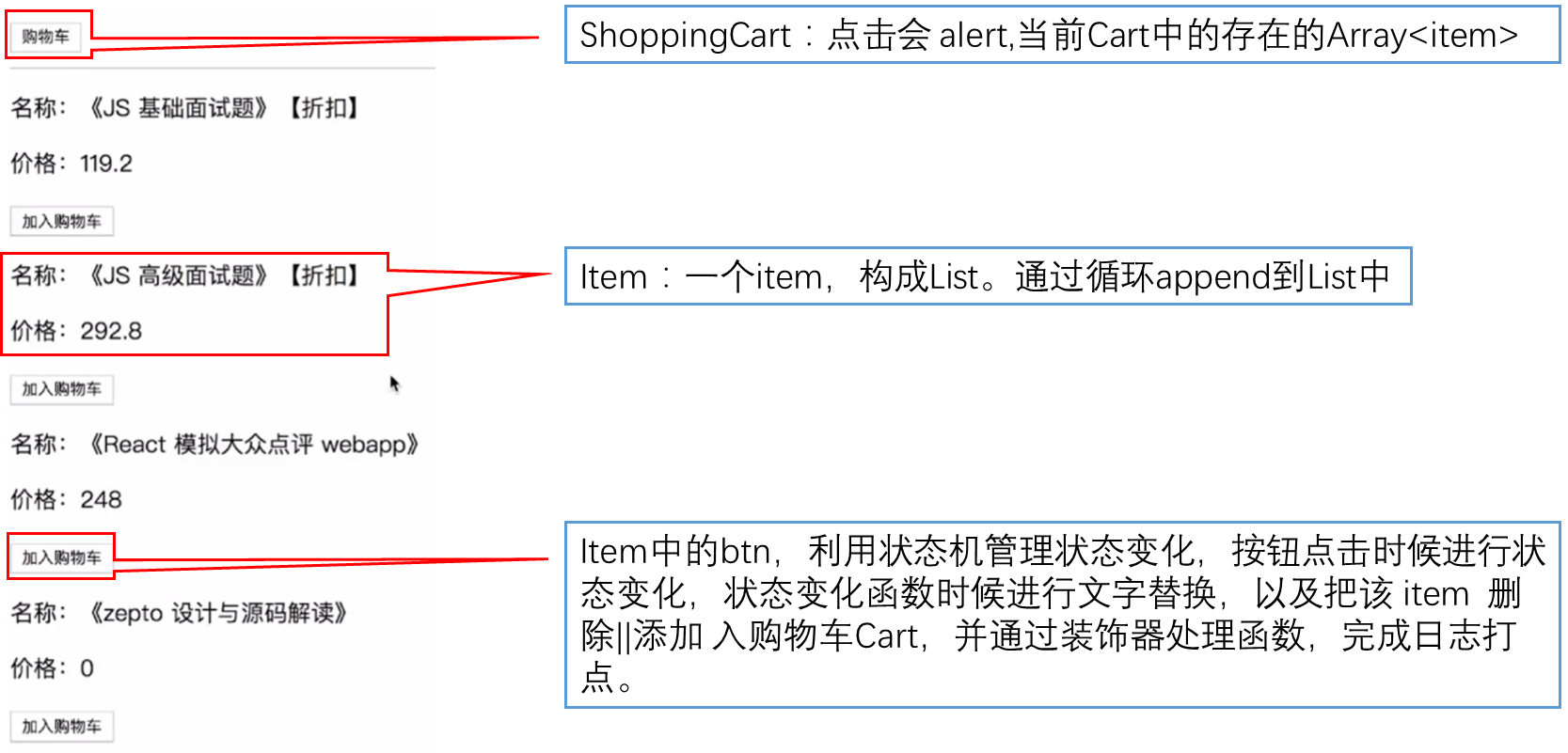
**描述语言语法如何定义，如何解释和编译，如babel 一般用于编译**

# 日常使用与综合应用

## 心得

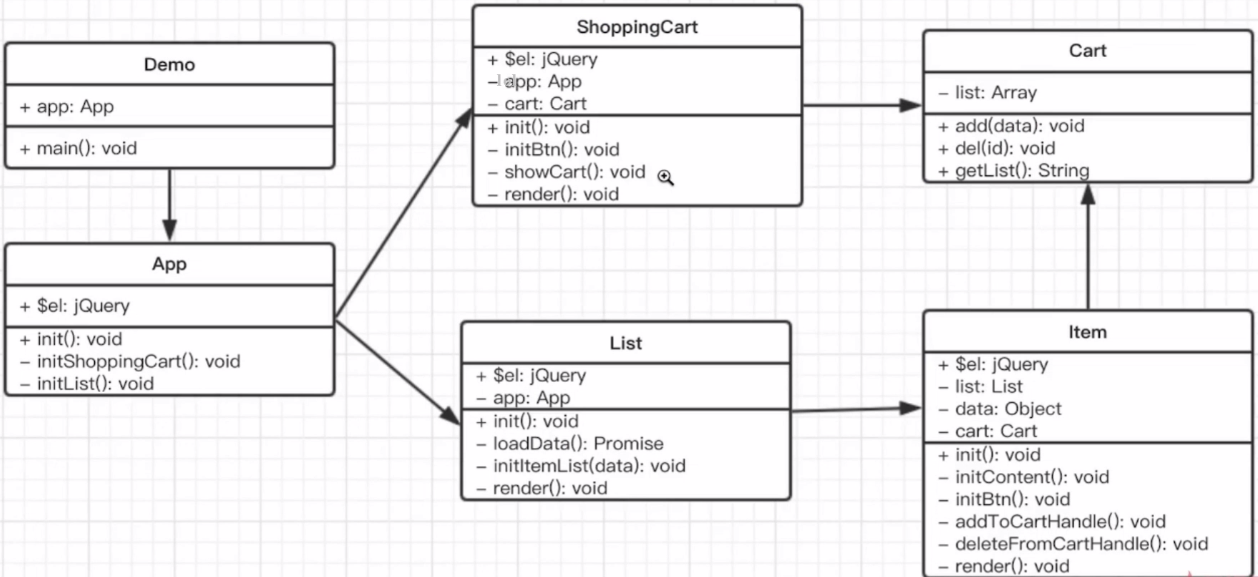
1. 常用的设计模式，要强制自己模仿、掌握
2. 非常用的设计模式，视业务场景 选择性使用

## 综合用例—购物车



**入口文件是index.js 也可以理解成Demo ，ShoppingCart 和 List 在各自组件的render中 将自己的this.$el append到app里面。然后在List中的initItemList 中 将List的实例传入Item，Item render时候 会将Item append 入List。先完成List 再完成 List 插入 App。**

|  |  |  |
| --- | --- | --- |
| 要将list传入Item，然后在List中的  **initItemList 调用item.init（）** | **List与ShoppingCart中的render，** | **App中 初始化List与ShoppingCart** |



## 具体代码

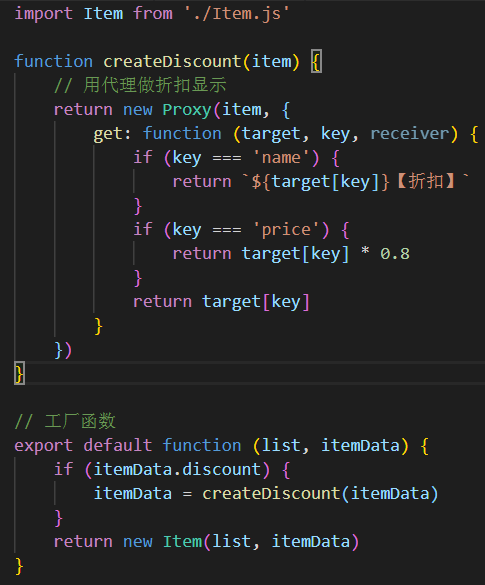
### App.js



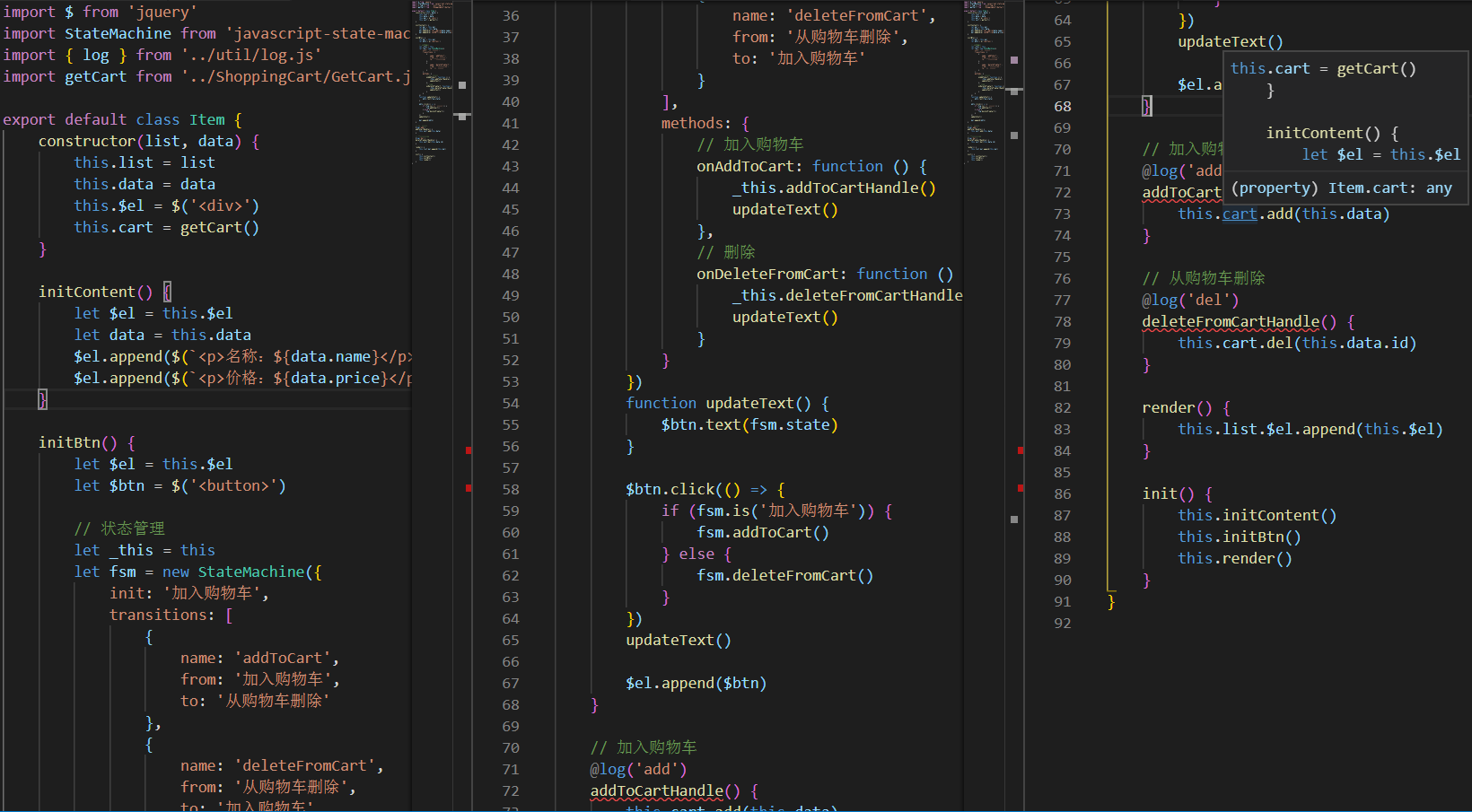
### List.js



### CreateItem.js



### Item.js



### ShoppingCart.js(左)与Cart.js（右）

|  |  |
| --- | --- |
|  |  |

### Log.js与目录结构

|  |  |
| --- | --- |
|  |  |

### 用到的模式

1. 工厂模式：$(‘xxx’)，创建商品；
2. 单例模式：购物车；
3. 装饰器模式：日志打点；
4. 观察者模式：网页事件，promise；
5. 状态模式：添加购物车&从购物车删除；
6. 模板方法模式：渲染有统一的方法init ，内部包含了各模块渲染；
7. 代理模式：打折商品信息处理、网络api接口转发；

## 附项目webpack配置

html-webpack-plugin

webpack-dev-server

babel-loader

babel-preset-latest babel-preset-es2015

babel-plugin-transform-decorators-legacy

core-decorators //常用的装饰器

packjson.json script:

"dev": "webpack-dev-server --config ./webpack.dev.config.js --mode development"

.babelrc

{

"presets": ["es2015", "latest"],

"plugins": ["transform-decorators-legacy"]//实现支持es2017装饰器

}

http-server -p 8880 //多开一个服务器，用作数据mock

const path = require('path')

const HtmlWebpackPlugin = require('html-webpack-plugin')

