1.设计准则

1. 小即美
2. 让每个程序只做好一件事
3. 快速建立原型
4. 舍弃高效率而取可移植性
5. 采用纯文本来存储数据
6. 充分利用软件的杠杆效应（软件复用）
7. 使用shell脚本来提高杠杆效应和可移植性
8. 避免强制性的用户界面
9. 让每个程序都称为过滤器

2.SOLID五大设计原则

1. S：单一职责
    a. 一个程序只做好一件事
    b. 功能过于复杂就拆分
2. O：开放封闭原则
    a. 对扩展开放，对修改封闭
    b. 增加需求时，扩展新代码，而非修改原有代码
3. L：里式置换原则
    a. 子类能覆盖父类
    b. 父类能出现的地方子类就能出现
4. I：接口独立原则
    a. 保持接口的单一独立
5. D：依赖导致原则
    a. 面向接口编程，依赖抽象而不是具体
    b. 只关注接口不关注类的具体实现

3.23种设计模式

- 创建型
    - 工厂模式
    - 单例模式
    - 原型模式
- 结构性
    - 适配器模式
    - 装饰器模式
    - 代理模式

    - 外观模式
    - 桥接模式
    - 组合模式
    - 享元模式
- 行为型
    - 策略模式
    - 模板方法模式
    - 观察者模式
    - 迭代器模式
    - 职责连模式
    - 命令模式
    - 备忘录模式
    - 状态模式
    - 访问者模式
    - 中介者模式
    - 解释者模式

4.工厂模式

介绍

- 将new操作单独封装
- 遇到new时，就要考虑是否该使用工厂模式

```
class Product {
  constructor (name) {
    this.name = name
  }

  init () {}

  fun1 () {}

}

class Creator {
  create (name) {
    return new Product(name)
  }
}
```

```
17
18 let creator1 = new Creator()
19
```

实例：

- Jquery - $('div')
- React.createElement
- vue异步组件

5.单例模式

介绍

- 系统中唯一使用
- 一个类只有一个实例

```
1  class SingleObject {
2    login(){
3      console.log('login....');
4    }
5  }
6
7  SingleObject.getInstance = (function(){
8    let instance
9    return function(){
10     if(!instance){
11       instance = new SingleObject()
12     }
13     return instance
14   }
15 })();
16
17 let obj1 = SingleObject.getInstance()
18 let obj2 = SingleObject.getInstance()
19 console.log(obj1 === obj2);
```

实例：

vuex

6.适配器模式

介绍

- 旧接口格式和使用者不兼容
- 中间加一个适配转换接口

```
class Adaptee {
  specificRequest () {
    return '德国标准插头'
  }
}

class Target {
  constructor () {
    this.adaptee = new Adaptee()
  }

  request () {
    let info = this.adaptee.specificRequest()
    return `${info} -> 转换器 -> 中国标准插头`
  }
}

let target = new Target()
target.request()
```

实例

vue computed \ 旧接口封装

7.装饰器模式

简介

- 为对象添加新功能
- 不改变原有的结构和功能

场景

- ES7装饰器
- core-decorators

默认不支持装饰器，需要增加插件。

```
@testDec(false)
class Demo{

}
```

```
5
6  function testDec(isDec){
7    return function(target){
8      target.isDec = isDec
9    }
10 }
11
12 alert(Demo.isDec)
```

mixins

```
1  function mixins(...list){
2    return function(target){
3      Object.assign(target.prototype,...list)
4    }
5  }
6
7  const Foo = {
8    foo(){
9      alert('foo')
10   }
11 }
12
13 @mixins(Foo)
14 class MyClass{
15
16 }
17
18 let obj = new MyClass()
19 obj.foo()
```

设计验证原则

- 将现有对象和装饰器进行分离，两者独立存在
- 符合开放封闭原则

8.代理模式

介绍

- 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

```
class ReadImg{
  constructor(fileName){
    this.fileName = fileName
    this.loadFromDisk() //初始化即从硬盘加载,模拟
  }

  display(){
    console.log('display...' + this.fileName);
  }

  loadFromDisk(){
    console.log('loading...' + this.fileName);
  }
}

class ProxyImg {
  constructor(fileName){
    this.realImg = new ReadImg(fileName)
  }

  display(){
    this.realImg.display()
  }
}

let proxyImg = new ProxyImg('1.png')
proxyImg.display()
```

使用场景

- 网页事件代理
- jquery： $.proxy
- ES6 proxy
- vue3

代理模式VS适配器模式

- 适配器模式：提供一个不同的接口
- 代理模式：提供一模一样的接口

代理模式VS装饰器器模式

- 装饰器模式：扩展功能，原有功能不变且可以直接使用

- 代理模式：显示原有功能，但是是经过限制或者阉割之后的

9.外观模式

介绍

- 为子系统的一组提供了一个高层接口
- 使用者使用这个高层接口

```
function bindEvent (elem, type, selector, fn) {
  if (fn == null) {
    fn = selector
    selector = null
  }
}

bindEvent(elem, 'click', '#div1', fn)
bindEvent(elem, 'click',  fn)
```

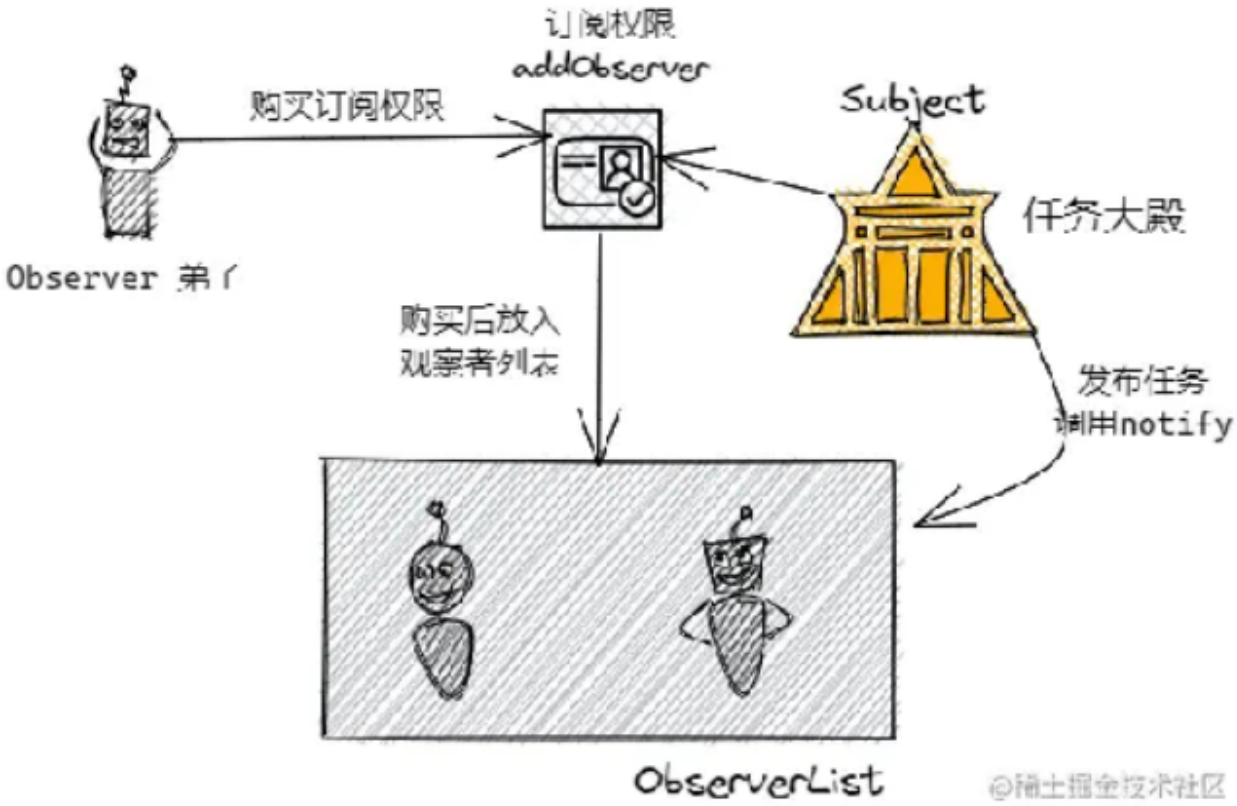
不符合单一职责原则和开放封闭原则，谨慎使用，不要滥用

10.观察者模式与发布订阅模式

介绍

- 发布&订阅
- 一对多

观察者模式：

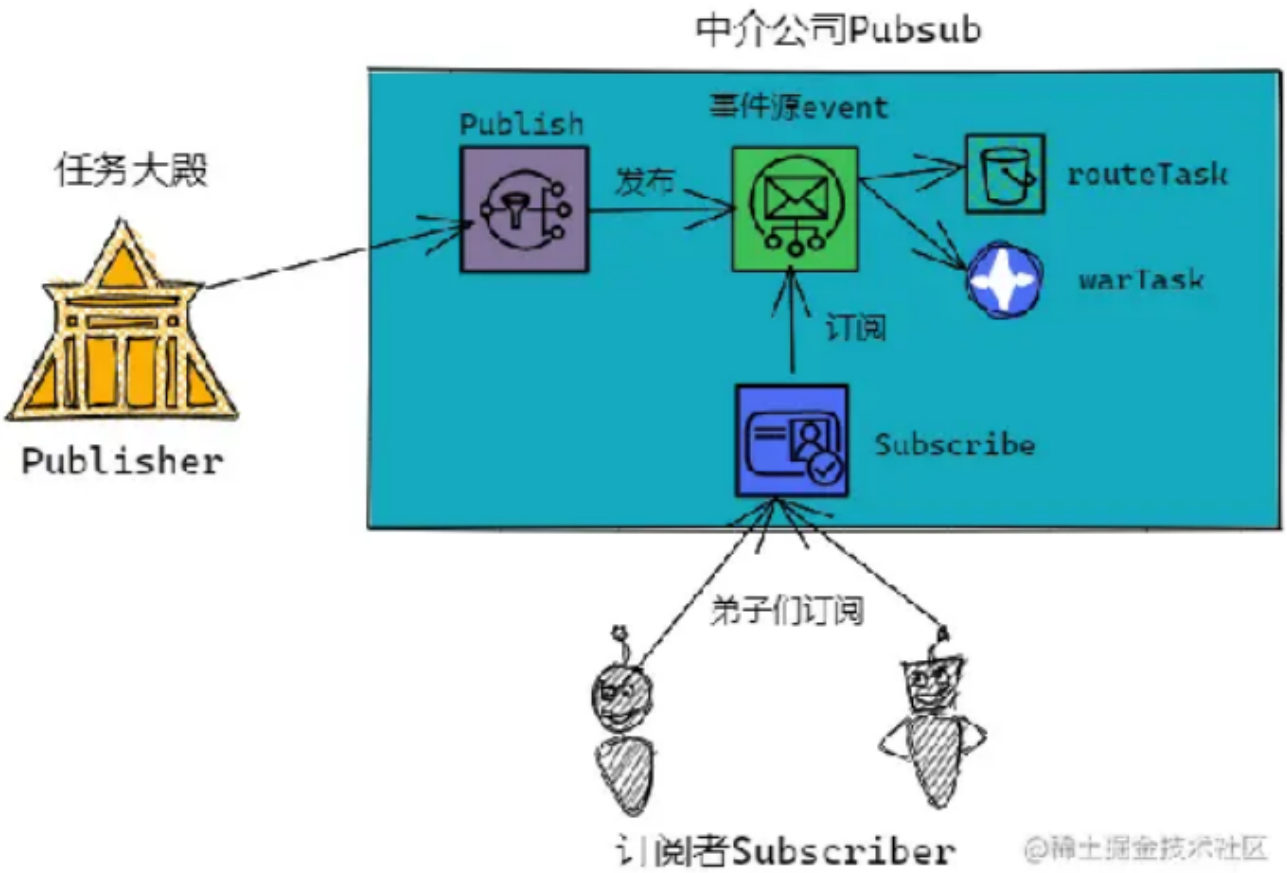


1. 当对象存在一对多的关系，其中一个对象的状态发生改变，所有依赖他的对象都会收到通知。
2. 在观察者模式中，只有两个主体：目标主体（object）和观察者（observer）。
3. 目标对象：
    a. 维护观察者列表
    b. 定义添加观察者的方法
    c. 当自身发生变化后，通过调用自己的notify方法依次通知每个观察者执行update方法。
4. 观察者需要实现update方法，供目标对象调用。

```
class Observer {
    constructor(name){
        this.name = name
    }

    update(task){
        console.log(task);
    }
}
class Subject {
    constructor(){
        this.observerList = []
    }
    addObserver(observer){
        this.observerList.push(observer)
```

```
    }
    notify(task){
        this.observerList.forEach(observer=>observer.update(task))
    }
}
```

发布订阅模式：



1. 基于一个事件通道，希望接受通知的对象Subscriber通过自定义事件订阅主题，被激活事件的对象Publisher通过发布主题事件的方式通知各个订阅该主题的Subscriber对象。
2. 发布订阅模式包含：发布者Publisher，事件调度中心Event Channel，订阅者Subscriber。
    a. 发布者：发布任务
    b. 事件调度中心：
        i. 维护任务类型，以及每种任务下的订阅情况。
        ii. 给订阅者提供订阅功能。
        iii. 当发布者发布任务之后，事件调度中心给订阅者发布任务。
    c. 订阅者：任务接受。

```
lass PubSub {
    constructor(){
        this.events = {}
    }

```

```
    // 订阅方法
    subscribe(type,cb){
        if(!this.events[type]){

            this.events[type] = []
        }
        this.events[type].push(cb)
    }
    // 发布方法
    publish(type,...args){
        this.events[type] && this.events[type].forEach(cb=>cb(...args))
    }
    // 取消订阅方法
    unsubscribe(type,cb){
        if(this.events[type]){
            const cbIndex = this.events[type].findIndex(fn=>fn===cb);
            if(cbIndex > -1){
                this.events[type].splice(cbIndex,1)
            }
            if(this.events[type].length === 0){
                delete this.events[type]
            }
        }
    }
    unsubscribeAll(type){
        if(this.events[type]){
            delete this.events[type]
        }
    }
}
```

设计模式	观察者模式	发布订阅模式
主体	Object观察者、Subject目标对象	Publisher发布者、Event Channel事件中心、Subscribe订阅者
主体关系	Subject中通过observerList记录ObServer	Publisher和Subscribe不想不知道对方，通过中介联系
优点	角色明确，Subject和Object要遵循约定的成员方法	松散耦合，灵活度高，通常应用在异步编程中
缺点	紧耦合	当事件类型变多时，会增加维护成本
使用案例	双向数据绑定	事件总线EventBus

场景

- 网页事件绑定
- Promise
- nodejs自定义事件
- vue

11.迭代器模式

介绍

- 顺序访问一个集合
- 使用者无需知道集合的内部结构（封装）

```
class Iterator {
  constructor(container){
    this.list = container.list
    this.index = 0
  }

  next(){
    if(this.hasNext()){
      return this.list[this.index++]
    }
    return null
  }

  hasNext(){
    if(this.index >= this.list.length){
```

```
      return false
    }
    return true
  }
}

class Container {
  constructor(list){
    this.list = list
  }

  // 生成遍历器
  getIterator(){
    return new Iterator(this)
  }
}

let arr = [1,2,3,4,5,6]
let container = new Container(arr)
let iterator =  container.getIterator()
while(iterator.hasNext()){
  console.log(iterator.next());
}
```

场景

- jquery each
- ES6 Iterator

```
function each(data){
  let iterator = data[Symbol.iterator]()
  let item = {done:false}
  while(!item.done){
    item = iterator.next()
    if(!item.done){
      console.log(item.value);
    }
  }
}
```

上述代码可以直接使用for of来遍历

ES6 Iterator 与 Generator

- Iterator的价值不限于几个类型的遍历，还有Generator可以使用
- 只要返回的数据符合Iterator接口的要求，即可以使用Iterator语法，这就是迭代器模式

12.状态模式

介绍

- 一个对象有状态变化
- 每次状态变化都会触发一个逻辑
- 不能总是用if...else来控制

```
class State {
  constructor (color) {
    this.color = color
  }

  handle (context) {
    context.setState(this)
  }
}

class Context {
  constructor () {
    this.state = null
  }

  getState () {
    return this.state
  }

  setState (state) {
    this.state = state
  }
}

let context = new Context()

let green = new State('green')
let yellow = new State('yellow')
let red = new State('red')

```

```
green.handle(context)
console.log(context.getState())

yellow.handle(context)
console.log(context.getState())

red.handle(context)
console.log(context.getState())
```

场景

- Promise
- 有限状态机

13.其它设计模式

1.原型模式

概念

- clone一个自己，生成一个新对象

JS中应用

- JS中的Object.create

```
let prototype = {
  getName: function () {
    return this.first + '' + this.last
  },
  say: function () {
    console.log('hello')
  }
}

let x = Object.create(prototype)
x.first = 'A'
x.last = 'B'
console.log(x.getName())
x.say()
```

2.桥接模式

概念

- 用于把抽象化与实现化解耦，使得二者可以独立变化

- 比如画画，画一个红色的圆一个绿色的圆，一个红色的矩形一个绿色的矩形。可以变成画形状和选择颜色两个功能实现。

3.组合模式

概念

- 生成树形结构，表示“整体-部分”关系
- 让整体和部分有统一的操作方式

虚拟DOM中的vnode

4.享元模式

概念

- 共享内存，相同的数据，共享使用

使用

事件绑定，将div内部的元素的事件绑定在div上，事件代理。减少内存销毁。

5.策略模式

概念

- 不同策略分开处理
- 避免出现大量if...else 或者 switch...case

```
class User {
  constructor (type) {
    this.type = type
  }

  buy () {
    if (this.type === 'ordinary') {
      console.log('普通用户')
    }else if (this.type === 'member') {
      console.log('会员购买')
    }else if (this.type === 'vip') {
      console.log('vip')
    }
  }
}

let u1 = new User('ordinary')
u1.buy()
```

```
19
20 let u2 = new User('member')
21 u2.buy()
22
23 let u3 = new User('vip')
24 u3.buy()
25
```

上述代码修改为

```
1  class Ordinary {
2    buy () {
3      console.log('ordinary')
4    }
5  }
6
7  class Member {
8    buy () {
9      console.log('member')
10   }
11 }
12
13 class Vip {
14   buy () {
15     console.log('vip')
16   }
17 }
18
19 let o = new Ordinary()
20 o.buy()
21
22 let m = new Member()
23 m.buy()
24
25 let v = new Vip()
26 v.buy()
```

6.模板方法模式

```
1  class Action {
```

```
  handle () {
    this.handle1()
    this.handle2()
    this.handle3()
  }

  handle1 () {}
  handle2 () {}
  handle3 () {}
}

```

7.职责链模式

概念

- 一步操作可能分为多个职责角色来完成
- 把这些角色都分开，然后用一个链串起来
- 将发起者和各个处理者进行隔离

```
class Action {
  constructor(name){
    this.name = name
    this.nextAction = null
  }

  setNextAction(action){
    this.nextAction = action
  }
  handle(){
    console.log(`${this.name} 处理`);
    if(this.nextAction){
      this.nextAction.handle()
    }
  }
}

let zz = new Action('组长')
let zg = new Action('主管')
let zj = new Action('总监')

```

```
zz.setNextAction(zg)
zg.setNextAction(zj)
zz.handle()
```

8.命令模式

概念

- 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。

```
class Receiver {
  exec(){
    console.log('执行');
  }
}

class Command{
  constructor(receiver){
    this.receiver = receiver
  }
  cmd(){
    console.log('触发命令');
    this.receiver.exec()
  }
}

class Invoker{
  constructor(command){
    this.command = command
  }
  invoke(){
    console.log('开始');
    this.command.cmd()
  }
}

// 士兵
let soldier = new Receiver()
// 小号手
let trumpeter = new Command(soldier)
```

```
// 将军
let general = new Invoker(trumpeter)
general.invoke()
```

js中的应用

- 网页富文本编辑器操作，浏览器封装了命令对象
- document.execCommand('bold')
- document.execCommand('undo')

9.备忘录模式

概念

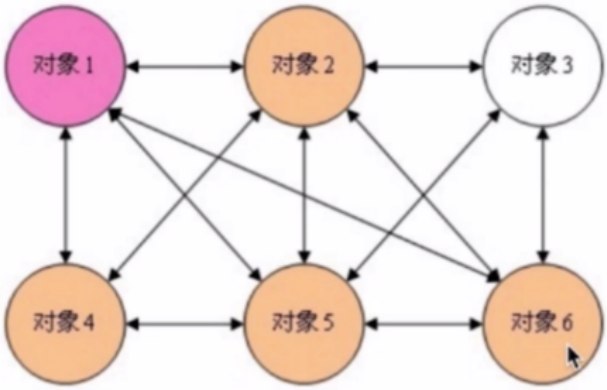
- 随时记录一个对象的状态变化
- 随时可以恢复之前的某个状态（撤销功能）

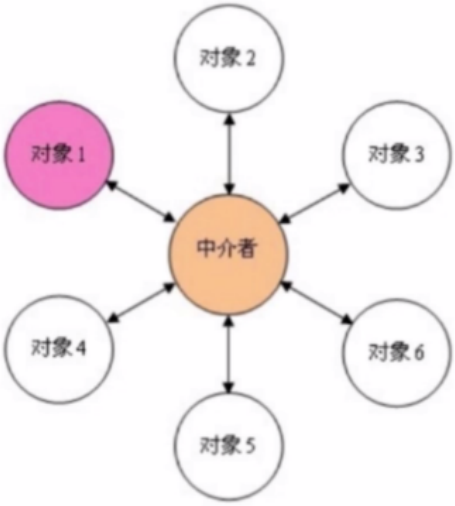
```
// 备忘录状态
class Memento {
  constructor (content) {
    this.content = content
  }

  getContent () {
    return this.content
  }
}

// 备忘录列表
class CareTaker {
  constructor () {
    this.list = []
  }
  add (memento) {
    this.list.push(memento)
  }
  get (index) {
    return this.list[index]
  }
}

// 编辑器
class Editor {
```

```
  constructor () {
    this.content = null
  }

  setContent (content) {
    this.content = content
  }

  getContent () {
    return this.content
  }

  saveContentToMemento () {
    return new Memento(this.content)
  }

  getContentFromMemento (memento) {
    this.content = memento.getContent()
  }
}

let editor = new Editor()
let careTaker = new CareTaker()
editor.setContent('111')
editor.setContent('222')
careTaker.add(editor.saveContentToMemento())
editor.setContent('333')
careTaker.add(editor.saveContentToMemento())
editor.setContent('444')

console.log(editor.getContent())
editor.getContentFromMemento(careTaker.get(1))
console.log(editor.getContent())
editor.getContentFromMemento(careTaker.get(1))
console.log(editor.getContent())
```

10.中介者模式

概念






```
class A {
  constructor () {
    this.number = 0
  }

  setNumber (num, m) {
    this.number = num
    if (m) {
      m.setB()
    }
  }
}

class B {
  constructor () {
    this.number = 0
  }

  setNumber (num, m) {
    this.number = num
    if (m) {
      m.setA()
    }
  }
}
```

```
26
27 // 中介者
28 class Mediator {
29   constructor (a, b) {
30     this.a = a
31     this.b = b
32   }
33
34   setB () {
35     let number = this.a.number
36     // 规则
37     this.b.setNumber(number * 100)
38   }
39
40   setA () {
41     let number = this.b.number
42     this.a.setNumber(number / 100)
43   }
44 }
45
46 let a = new A()
47 let b = new B()
48 let m = new Mediator(a, b)
49
50 a.setNumber(100,m)
51 console.log(a.number , b.number)
52 b.setNumber(100,m)
53 console.log(a.number , b.number)
```

需要修改自己的值同时通过中介去修改需要修改的值。

11.访问者模式

概念

- 将数据操作和数据结构进行分离

12.解释器模式

概念

- 描述语言语法如何定义，如何解释和编译
- 用于专业场景