day24 设计模式

回顾

ajax

ajax是一个异步的请求工具，主要提供对应的客户端和服务端之间的数据交互。它可以完成对应的局部刷新（同步加载页面整体会刷新，异步加载对应的页面不会进行刷新）核心对象（XMLHttpRequest）。

ajax的流程

- 创建xhr对象
- 打开连接
- 设置对应请求头
- 发送请求
- 监听事件接收数据（onreadystatechange）

get请求

get请求数据携带在对应的url中

post请求

post需要设置对应的请求头

```
xhr.setRequestHeader('content-type','x-www-form-urlencoded') //以表单数据提交
```

post请求携带在请求体中

```
xhr.send(body)
```

JSONP

解决对应的跨域的方案 （主要利用script标签不受跨域的影响 通过回调函数来接收对应的数据）实际上是一个get请求

步骤

- 准备一个回调函数（全局的）
- 使用script标签链入对应的地址 传入对应参数及回调函数
- 加载完成 删除对应的script标签

设计模式

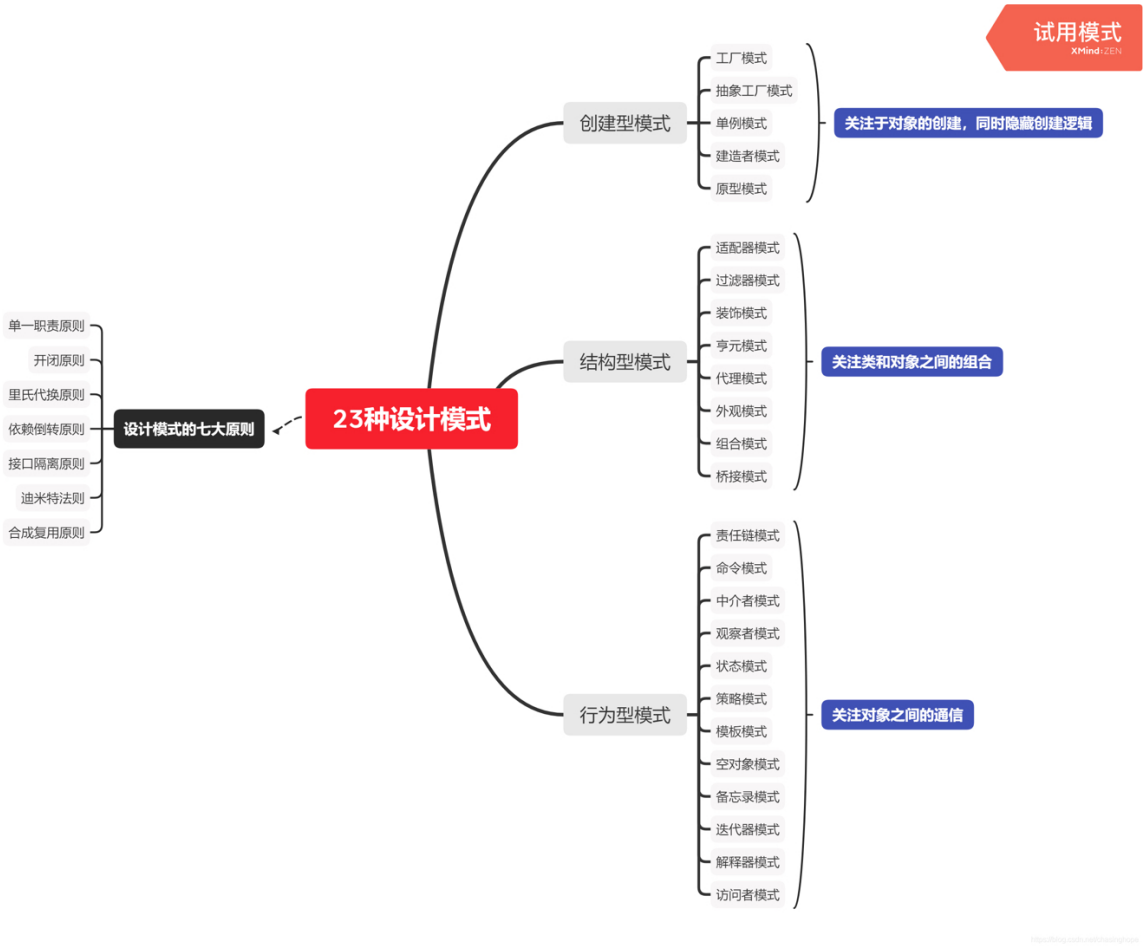
概述

设计模式是对应的一些处理方案形成的思想所构成的模式。主要针对是类和对象设计和构造。它区分语言，总共有23种设计模式（架构设计 ， 源码设计）。

设计模式的分类

- 创建型模式 （关于对象的创建）
- 结构型模式 (将多个小结构并入一个大结构)
- 行为型模式 （对象的通信逻辑）

设计模式分类图



设计模式的七大原则

- 单一责任原则
- 开闭原则
- 里氏置换原则
- 依赖倒转原则
- 接口隔离原则
- 迪米特法则
- 合成复用原则

主要讲解的模式

- 创建型模式 （工厂模式、单例模式）
- 结构型模式 （代理模式、装饰器模式、组合模式）
- 行为型模式 （观察者模式）

工厂模式

以工厂的形式来生产对象 （不关注对象的细节）

```
function factory(name){
    //创建对象
    let obj = {}
    //属性赋值
    obj.name = name
    //返回对象
    return obj
}
```

单例模式（饿汉 、懒汉）

保证产生的对象只有一个 （不会被污染）

利用闭包实现

```
function singletonClouse(){
    let single = null
    return function(){
        //判断对应的对象是否为null
        //如果为null产生新的对象
        if(!single){
            single = new Object()
        }
        return single
    }
}
let singletonFn = singletonClouse()
let obj = singletonFn()
let obj1 = singletonFn()
console.log(obj === obj1)
```

利用原型实现

```
function singletonPrototype(){
    //判断是否存在这个对象
    //不存在给产生一个新的对象
    if(!Object.prototype.single){
        Object.prototype.single = new Object()
    }
    return Object.prototype.single
}
let obj = singletonPrototype()
let obj1 = singletonPrototype()
console.log(obj === obj1)
```

利用静态属性实现

```
function singletonStatic(){
    //判断是否存在 不存在赋值
    if(!Object.single){
        Object.single = new Object()
    }
    return Object.single
}
let obj = singletonStatic()
let obj1 = singletonStatic()
console.log(obj === obj1)
```

使用global对象实现

```
function singletonGlobal(){
    //globalThis指向当前global对象 window
    if(!globalThis.single){
        globalThis.single =  new Object()
    }
    return globalThis.single
}
let obj = singletonGlobal()
let obj1 = singletonGlobal()
console.log(obj === obj1)
```

组合模式

将多个小结构组合成一个大结构 (将共有的函数放在一起调用)

示例

```
function PlayGame(){
    this.init = function(){
        console.log('开机 登录')
    }
}
function GoHome(){
    this.init = function(){
        console.log('收拾 背包')
    }
}
function Run(){
    this.init = function(){
        console.log('拉伸 穿鞋')
    }
}
//跑步回家玩电脑
new Run().init()
new GoHome().init()
new PlayGame().init()
```

将init方法 一起执行 利用组合模式来执行多个同名方法

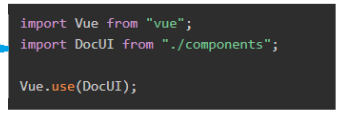
```
function Combination(){
    this.objs = []
    //传入对应的对象
```

```
    this.add = function(...objs){
        //接收对应的对象数组
        this.objs = this.objs.concat([...objs])
    }
    //执行对应的里面的init方法 以参数传递函数名的形式来执行
    this.exec = function(fnName){
        //遍历对应的对象数组 调用里面相关的方法
        this.objs.forEach((obj)=>{
            //调用对应的方法
            obj[fnName].call(this)
        })
    }
}
let combination = new Combination()
combination.add(new Run(),new GoHome())
combination.exec('init')
```

组合模式的应用 vue里面 use 和 install

这个use它会将对应的你传入的对象里面install方法调用
如果没有install方法 它会将当前传入的内容识别为install 再调用

声明一个对象 对象里面有一个install方法
在install方法中
遍历所有的component（组件） 加载到vue中 成为一个组件对象



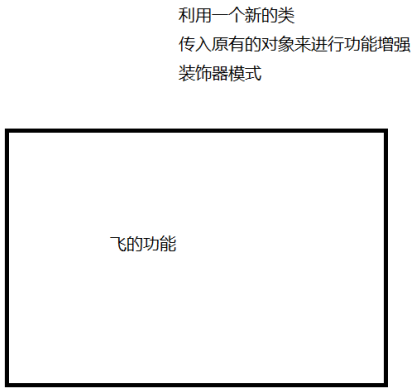
导入对应导出对象
使用vue去调用use方法传入对应这个对象

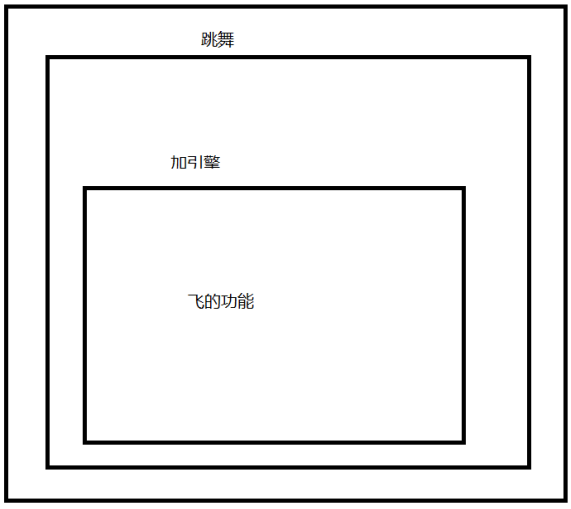
```
class Vue {
    //解析对应的对象 执行对应的install
    static use(...objs) {
        objs = objs.map(v => {
            //如果没有install方法
            if (!v.install) {
                if (typeof v != 'function') {
                    throw new Error('传入内容出错')
                }
                //将本身当作insatll函数
                let fn = v
                v = {
                    install() {
                        fn()
                    }
                }
            }
            return v
        })
        Vue.exec(objs)
    }
    //传入对象进行执行
```

```
    static exec(arr) {
        arr.forEach(obj => {
            obj['install'].call(this, Vue)
        })
    }
}
Vue.use({
    install() {
        console.log('吃饭了吗')
    }
}, {
    install() {
        console.log('手机一直响 吵死了')
    }
})
Vue.use({
    install(vue) {
        console.log(vue)
        console.log('哈哈哈哈')
    }
})
Vue.use(()=>{
    console.log('函数执行了')
})
```

装饰器模式

用一个新的类将对应的原本的对象进行包装再进行加强 （在不改变原有对象的基础上增强对象）






```
function Person() {
    this.run = function () {
        console.log('跑')
    }
}
//传入要增强的对象 返回一个新的对象
function Stronger(person) {
    this.person = person
    this.run = function () {
        this.person.run()
        console.log('我会飞')
```

```
    }
}
//基础对象
let person = new Person()
person.run() //跑
//增强的对象
let stronger = new Stronger(person)
stronger.run() //跑 我会飞
```

Ts内置有对应的装饰器 Decorator TS中使用注解来进行对应的装饰器添加 @decorator

代理模式

概述

代理模式是在原有对象的基础上增强对应的对象 （利用代理对象来增强）代理对象通常访问的是实际的对象。

示例

我（原本的对象）请了会计（代理），会计给我管钱 (功能增强)，当会计（代理）把钱花完了（我（原本对象）的钱也没有）

- 代理对象对原本的对象进行了功能增强
- 代理对象影响的是实际的对象

ES7新增对应的Proxy的类 来帮助我们进行代理

Proxy (vue3的底层实现)

实例化 （传入对应的被代理对象 处理对象 产生一个代理对象）

```
var proxy = new Proxy(target,handler)
```

示例

```
//被代理对象
var object = {
    name: '张三',
    age: 18
}
//代理对象 传入被代理对象 及 处理对象
//在proxy里面的set和get不要使用proxy
var proxy = new Proxy(object, {
    //获取相关的属性 被代理对象 属性名  代理对象
    get(targetObj,attributeName,proxyObj) {
        // console.log(targetObj,attributeNmae,proxyObj)
        let result = targetObj[attributeName]
        if(attributeName == 'age'){
            result += '岁'
        }else if(attributeName == 'name'){
            result = '我的名字叫'+result
        }
        return result
        // return proxyObj[attributeNmae] 造成栈溢出
    },
    //设置相关的属性 被代理对象 属性名 属性值  代理对象
```

```
    set(targetObj,attributeName,attributeValue,proxyObj) {
        if(attributeName == 'age' && typeof attributeValue != 'number'){
            throw new Error('你传入的不是一个年纪')
        }else{
            targetObj[attributeName] = attributeValue
        }
        // console.log(arguments)
    },
    //目标对象 对应的属性 默认返回false 使用in关键词调用
    has(targetObj,p){
        console.log(targetObj,p)
        return p in targetObj
    },
    deleteProperty(targetObj,attribute){
        // console.log(attribute)
        delete targetObj[attribute]
    }
})
//使用代理对象
console.log(proxy.age) //访问get方法
proxy.name = 'jack' //访问set
console.log(proxy.name) //访问get方法
// proxy.age = 'abc' 报错
console.log(object.name)
console.log('name' in proxy) //调用has
delete proxy.age
console.log(object)
```

proxy的handler相关属性方法

`handler.apply()`

A trap for a function call. 使用对应的call方法及apply方法调用

`handler.construct()` 使用new关键词的时候调用的

A trap for the `new` operator.

`handler.defineProperty()` 定义属性的时候调用的

A trap for `Object.defineProperty`.

`handler.deleteProperty()` 使用删除关键词的时候调用的

A trap for the `delete` operator.

`handler.get()` 获取属性值的时候调用的

A trap for getting property values.

`handler.getOwnPropertyDescriptor()` 获取属性详情的时候调用的

A trap for `Object.getOwnPropertyDescriptor`.

`handler.getPrototypeOf()` 获取原型的时候调用的

A trap for `Object.getPrototypeOf`.

`handler.has()` 判断属性是否存在调用

A trap for the `in` operator.

`handler.isExtensible()` 判断当前对象是否可以扩展的时候调用

A trap for `Object.isExtensible`.

`handler.ownKeys()` 获取对应全部属性名的时候调用

A trap for `Object.getOwnPropertyNames` and `Object.getOwnPropertySymbols`.

`handler.preventExtensions()` 判断当前是否是否禁止扩展的时候调用的

A trap for `Object.preventExtensions`.

`handler.set()` 设置属性值的时候调用的

A trap for setting property values.

`handler.setPrototypeOf()` 设置原型的时候调用的

A trap for `Object.setPrototypeOf`.

观察者模式

概述

观察者模式是前端最常用的模式，它相当于对应的监听和处理执行的机制。观察者模式别名obServer又被称为发布者-订阅者模式。

常见的发布订阅者模式

addEventListener （事件监听器）

```
//element.addEventListener(事件名,handler)
box.addEventListener('click',function(){
    //处理
    console.log('处理了')
})
```

- 你 发布者 （box添加点击事件）
- js事件处理线程 订阅者 （添加事件监听 保持监听）
- 处理 处理函数 订阅者 （订阅者进行处理）

生活中的发布订阅者

- 小王关注了一个美女主播 （订阅了）
- 美女主播上线 （发布了）
- 小王充钱（处理了）

实现发布订阅者模式

模仿eventListener来实现

- 事件监听 on

```
//监听事件传入一个事件名和对应的处理函数
on(eventName,handler){
    //判断是否存储了对应的事件
    if(eventName in this.events){
        //如果存储了就给他加到对应的处理函数数组内
        this.events[eventName].add(handler)
    }else{
        //如果没有的话先需要开劈一个数组 将处理函数装入
        this.events[eventName] = new Set([handler])
    }
}
```

- 事件执行 emit

```
//执行对应的处理函数 传入对应的事件执行对应的处理函数  传入对应的参数传递给处理函数
emit(eventName,...args){
    //不存在对应的事件 退出
    if(!this.events[eventName]) {
        retrun;
    }
    //存在 遍历对应的集合 调用的对应的处理函数
    this.events[eventName].forEach(handler=>{
        //调用对应的函数传递参数
        handler.apply(this,args)
    })
}
```

- 事件取消 off

```
//移出事件监听
off(eventName,handler){
    //查询是否具备对应的事件
    //不具备结束对应的事件
    if(!this.events[eventName]) {
        retrun;
    }
    //具备的话 删除对应的事件
    this.events[eventName].delete(handler)
}
```

完整版实现

```
class Observer{
    constructor(){
        //存储事件的容器和处理函数的容器 {click:[handler],mosuedown:[handler]}
        this.events = {}
    }
    //监听事件传入一个事件名和对应的处理函数
    on(eventName,handler){
        //判断是否存储了对应的事件
        if(eventName in this.events){
            //如果存储了就给他加到对应的处理函数数组内
            this.events[eventName].add(handler)
        }else{
            //如果没有的话先需要开劈一个数组 将处理函数装入
            this.events[eventName] = new Set([handler])
        }
    }
    //执行对应的处理函数 传入对应的事件执行对应的处理函数  传入对应的参数传递给处理函数
    emit(eventName,...args){
        //不存在对应的事件 退出
        if(!this.events[eventName]) {
            retrun;
        }
        //存在 遍历对应的集合 调用的对应的处理函数
        this.events[eventName].forEach(handler=>{
            //调用对应的函数传递参数
            handler.apply(this,args)
        })
    }
    //移出事件监听
    off(eventName,handler){
        //查询是否具备对应的事件
        //不具备结束对应的事件
        if(!this.events[eventName]) {
            retrun;
        }
        //具备的话 删除对应的事件
        this.events[eventName].delete(handler)
    }
}
```

总结

- off是用于取消事件

- on是用于监听事件
- emit是用于执行事件
- 存储使用的对象进行存储 （事件名为key 处理函数集合为value）
- emit传入的参数可以被对应的处理函数接收
- 观察者模式是vue2底层实现之一 （对应的数据双向绑定必须使用观察者模式）