行为型模式，

它是对在不同的对象之间`划分责任`和`算法`的抽象化，行为型模式不仅仅关注类和对象的结构，而且重点关注它们之间的相互作用；行为型模式一共有以下11种

- 模板方法模式（Template Method）
- 策略模式（Strategy）
- 命令模式（Command）
- 中介者模式（Mediator）
- 观察者模式（Observer）
- 迭代器模式（Iteratior）
- 访问者模式（Visiter）
- 责任链模式（Chain of Responsibility）
- 备忘录模式（Memento）
- 状态模式（State）
- 解释器模式（Interpreter）

观察者模式

> 定义：观察者模式定义了对象间的一种`一对多`的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都将得到通知，并自动更新。观察者模式属于行为型模式。

- 理解：一个目标对象对应多个观察者，当目标对象发生改变时，就通知所由的观察者
- 步骤
  - 在目标对象中维护了一个观察者的数组，新增时将观察者向数组中push；
  - 然后通过notify通知所有的观察者；
  - 每个观察者只有一个update函数，用来接收观察者更新后的一个回调；
- 观察者模式把观察者对象维护在目标对象中的，需要发布消息时直接发消息给观察者。在观察者模式中，目标对象本身是知道观察者存在的
- ```
  // 定义一个目标对象
  class Subject {
    constructor() {
      this.Observers = [];
    }
    add(observer) {
      //添加
      this.Observers.push(observer);
    }
    remove(observer) {
      //移除
      this.Observers.filter((item) => item === observer);
    }
    notify() {
      //通知所有观察者
      this.Observers.forEach((item) => {
        item.update();
      });
    }
  }
  //定义观察者对象
  ```
  ```

```
class Observer {
  constructor(name) {
    this.name = name;
  }
  update() {
    console.log(`my name is:${this.name}`);
  }
}

let sub = new Subject();
let obs1 = new Observer("observer11");
let obs2 = new Observer("observer22");
sub.add(obs1);  // 添加到观察者队列
sub.add(obs2);
sub.notify();   // 触发所有的观察者
```

设计模式

订阅发布模式

> 发布订阅模式是基于一个事件（主题）通道，希望接收通知的对象`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。

- 与观察者模式的不同：增加了第三方即`事件中心`；目标对象状态的改变并非直接通知观察者，而是通过第三方的事件中心来派发通知。
- 订阅者订阅主题，发布者推送某个主题时，订阅该主题的所有读者都会被通知到；避免了观察者模式无法进行过滤筛选的缺陷。
- 而发布/订阅模式中，发布者并不维护订阅者，也不知道订阅者的存在，所以也不会直接通知订阅者，而是通知调度中心，由调度中心`(公用的对象实例)`通知订阅者。
- 例：
    - Vue2中的模板语法，实现数据与界面的双向绑定，数据与页面相互更新

单例模式

> 单例就是保证一个类只有一个实例，并提供一个访问它的全局访问点

- 实现方法
    - 一般是先判断实例存在与否，如果存在直接返回，
    - 如果不存在就创建了再返回，这就确保了一个类只有一个实例对象。
    - 在JavaScript里，单例作为一个命名空间提供者，从全局命名空间里提供一个唯一的访问点来访问该对象
- 使用场景
    - 团队开发中，可能会产生命名冲突，这时候单例模式就能很好的解决这个问题。
    - 全局模态框，同一时间只允许弹出一个模态框，内容可以不一样，但都是同一个框，elementUI

代理模式

> 为一个对象提供一个代用品或者占位符，以便控制对它的访问

- 常用虚拟代理形式：某一个花销很大的操作，可以使用虚拟代理的方式延迟到需要它的时候再去创建它，（虚拟代理实现图片懒加载）
- 图片懒加载方式，使用loading图占位，然后通过异步的方式加载图片，等图片记载好了再替换掉

中介者模式

- 通过抽离配置项，进行代码的控制，实现各个事件的解耦，仅仅维护好中介者就行了

装饰器模式

- 不改变原对象的基础上，在程序运行时给对象动态的添加方法。

其他

防抖与节流

- 例如表单的频繁修改操作，可以进行代理，只提交一次，减少提交的次数

设计模式

JS设计模式

- 定义：在面向对象软件设计过程中针对特定问题的简洁而优雅的解决方案
- 作用：在于可复用性、可维护性。
- 原则：找出程序中变化的部分，并将变化封装起来。
    - 设计模式会增加代码量，容易将逻辑混论
    - 但软件的开发成本，不仅在开发阶段，也在后期维护阶段，采用好的设计模式，有利于降低维护成本
- 设计模式分为两部分：
    - 可变：
    - 不可变：将可变的部分封装后，剩下的就是不变和稳定的部分
- 设计模式的误解：
    - 设计模式的滥用，导致不该使用的地方使用，适得其反
    - 模式应该用在正确的地方
- 关注模式的意图而不是结构，模式只用放在具体的环境才有意义

对JS设计模式的误解

- 不同语言的语法特性是不一样的，准确的说，最初的 23 种设计模式是针对 cpp、java 等静态类型、传统面向对象编程语言设计的，而在 JS 中有些模式可能已 经不再需要，而有些模式的实现会有变化。比如有些人为了实现 JS 版的工厂模式，生硬的将创建对象的步骤延迟到子类中，**实际上在静态类 型语言中让子类来“决定”创建对象的原因是为了迎合“依赖倒置”原则，解开对象类型的耦合，让对象表现出多态性，而在 JS 这种动态类型语言中，多态是天生的， JS 不存在类型耦合，不需要将对象延迟到子类中创建，所以 JS 其实是不需要工厂模式的。**

  应用设计模式是为了解决问题，像上述这种牵强的应用只会让人觉得设计模式既难懂又没什么用，影响设计模式在 JS 中的发展。

模式的发展

设计模式在 1995 年最初被提出时有 23 种，但它不应该被局限于这 23 种，从它被发明到现在这些年也许已经有更多的模式被发现并总结了出来，比如有些 JS 书籍会提到模块模式、沙箱模式，这些模式能否经受住时间考验还有待验证，但**设计模式是在发展的，不仅限于最初的 23 种。**