47 设计模式知识脉络梳理+好题精做

更新时间：2020-06-24 10:24:46

> 勤学如春起之苗，不见其增，日有所长。——陶潜

在开始之前

不知不觉，专栏已经接近尾声。站在专栏的第 47 节，修言给持续学习到此处的同学们点个赞。

“设计模式与前端算法”是本专栏具有彩蛋意义的一个知识板块。在笔者最初的课程设计中，并没有考虑将其划进大纲——这两个知识板块，相对来说都比较独立，其本身的系统性太强，有一种“圈地为王”的感觉。如果各自展开来讲，那么这个长达50节的庞大专栏的篇幅估计又要翻一番，不仅会被编辑拉去浸猪笼，也会很大程度上进一步考验大家的学习耐力。

在盯着彼时已经长达 48 节的专栏大纲死磕了数日后，笔者决定保留“设计模式与前端算法”这个模块，并采取不同于前述知识的学习路径：本章不再针对具体的每一个知识点，作详细的解读与归纳。而是给到同学们两样东西：**知识脉络+优质好题**——前者意在帮助大家梳理思路、明确学习路径与学习方法，最终构建起一张知识索引地图；后者则是站在应试的角度，帮助大家把握命题的重点难点（这样即便部分同学对相关知识的学习止步于此，也不至于在面试时完全交白卷）。

设计模式知识脉络梳理

设计模式，本质上就是编程的“套路”。设计模式知识可以分为两大块：设计原则与23种具体设计模式。

其中，设计原则，可以理解为设计模式的指导思想。所有的设计模式，都是设计原则的体现。两者之间是“道”与“术”的关系。

设计原则

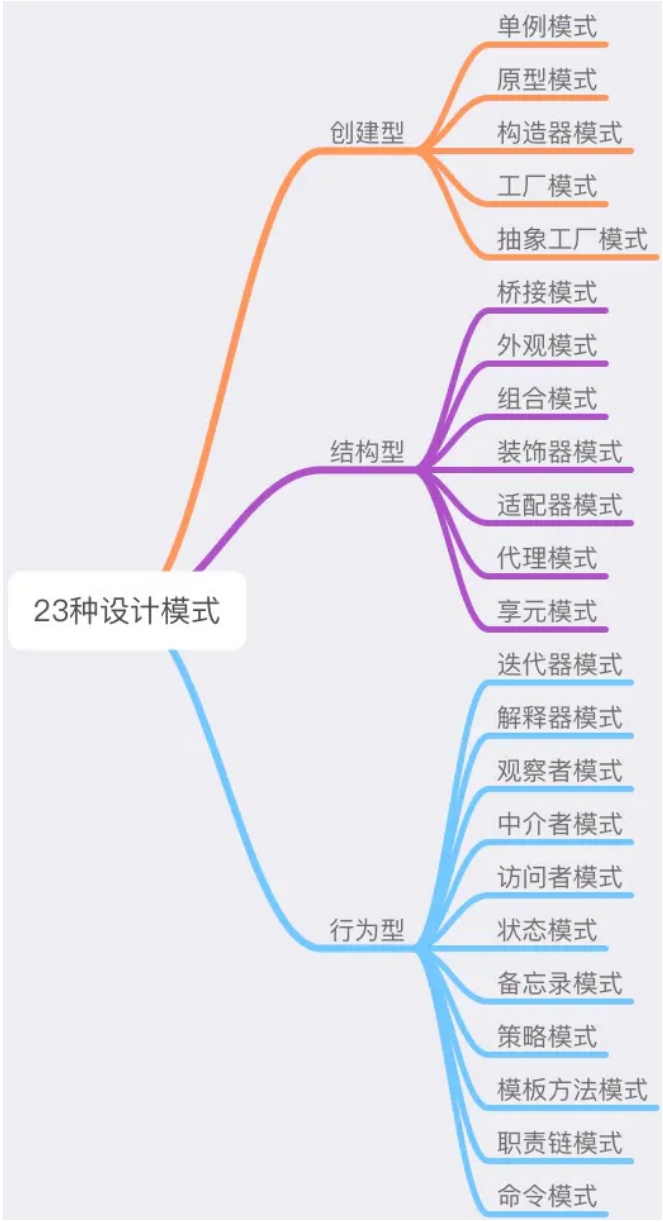
设计原则中最著名的是“SOLID设计原则”，它指代了面向对象编程和面向对象设计的五个基本原则：

- 单一功能原则（Single Responsibility Principle）
- 开放封闭原则（Opened Closed Principle）
- 里式替换原则（Liskov Substitution Principle）
- 接口隔离原则（Interface Segregation Principle）
- 依赖反转原则（Dependency Inversion Principle）

设计原则面向的是整个软件领域，对于前端来说，我们需要终端关注的是前两个——“单一功能”和“开放封闭”。它们各自有着以下的含义：

- 单一功能原则：官方的说法是“规定每个类都应该有一个单一的功能，并且该功能应该由这个类完全封装起来”。简单来说，可以理解为不要让一个函数/类做太多事情，而应该进行合理的功能拆分。
- 开放封闭原则：官方的说法是“软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的”。 简单来说，我们设计一个功能时，需要尽可能地区分变化的部分与稳定的部分，进而做到将变与不变分离（尤其是要做到封装变化的部分），达到降低耦合度的目的。

基于这样的设计原则，GOF 提出了最经典、也是如今“设计模式”这个名词普遍指代的23种设计模式：



划重点

23种设计模式，仍然是面向整个软件领域的。对于前端工程师来说，不必照单全收。这里给大家画一下重点，需要你挑出来单独学习一下的设计模式有：

1. 工厂模式
2. 单例模式
3. 原型模式（关键是要理解原型范式）
4. 装饰器模式
5. 适配器模式
6. 代理模式
7. 策略模式
8. 状态模式
9. 观察者模式/发布订阅模式
10. 迭代器模式

以上 10 种设计模式，是长久以来在前端领域应用相对广泛的前十名。在实际面试中，未必会有对应的编码题目，但很有可能考到概念理解（问答题）。

好题精做

设计模式的知识体系虽然庞大，但是命题热点却相对比较稳定。这里我选取了实际面试中考察频率最高的三个考题作展开讲解：

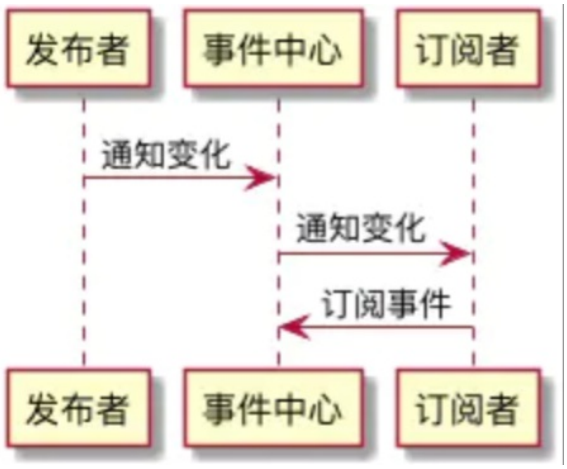
发布-订阅模式真题：实现一个 EventEmitter

> 题目描述：
>
> 1. 创建一个 EventEmitter，承担全局事件总线功能
> 2. 实现 on 事件监听方法
> 3. 实现 emit 事件订阅方法

发布-订阅模式简明解读

发布-订阅模式可以类比我们生活当中订报纸的过程。一般来说，我们订报纸要经过第一个第三方——报社。订阅后，报纸可能不会立刻就到我们手里，而是要等印刷制作完毕之后，先抵达报社。报社的工作人员拿到报纸后，再挨家挨户地分发出去。

在上面这个过程里，涉及了三个角色：订报纸的用户、报社、报纸厂。这三个角色映射到发布-订阅模式里，就分别对应了“发布者”、“事件中心”和“订阅者”，三者的关系如下图所示：



订阅者要想获得报纸，必须经过报社；报纸厂想要派发报纸，也必须经过报社。报社在整个过程中起到“中介”的作用，这也正是事件中心的职责所在。所谓“全局事件总线”，指的正是图中的“事件中心”。

在发布-订阅模式的协助下，全局环境中的任意两个对象都可以实现通信。Vue 中任意两个组件之间若想实现通信，全局事件总线就是一个很好的办法。

编码实现

```
class EventEmitter {
  constructor() {
    // handlers是一个map，用于存储事件与回调之间的对应关系
    this.handlers = {}
  }

  // on方法用于安装事件监听器，它接受目标事件名和回调函数作为参数
  on(eventName, cb) {
    // 先检查一下目标事件名有没有对应的监听函数队列
    if (!this.handlers[eventName]) {
      // 如果没有，那么首先初始化一个监听函数队列
      this.handlers[eventName] = []
    }

    // 把回调函数推入目标事件的监听函数队列里去
    this.handlers[eventName].push(cb)
  }

  // emit方法用于触发目标事件，它接受事件名和监听函数入参作为参数
  emit(eventName, ...args) {
    // 检查目标事件是否有监听函数队列
    if (this.handlers[eventName]) {
      // 如果有，则逐个调用队列里的回调函数
      this.handlers[eventName].forEach((callback) => {
        callback(...args)
      })
    }
  }
}
```

在编码作答过程中，大家把握以下三个要点：

1. 创建一个 Map（对应示例代码中的 handlers），它的作用是保存事件名称和函数之间的映射关系
2. on 事件监听方法的基本逻辑：如果 Map 中已经有此事件，则意味着对应的方法数组已存在，直接push到方法数组中；如果 Map 没有此事件， 则初始化事件监听函数队列。
3. emit 事件订阅方法的基本逻辑：检查对应事件的监听函数队列是否存在，若存在，则逐个调用队列里的回调函数。

单例模式真题：实现一个全局唯一的模态框

> 题目描述：创建一个全局唯一的 Modal 弹层

单例模式浅析

单例模式相对比较好理解，它要求我们保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例模式在前端领域常见的应用有：

1. 实例化一个全局唯一的元素，确保样式一致。比如说全局唯一的 Alert、全局唯一的 Modal 等等。
2. 实例化一个全局唯一的类。比如说上文提及的全局事件总线：一个上下文里只能有一个全局时间总线，否则就不叫“总线”了。

实现单例模式，常见的思路是用闭包来做——借助闭包中的自由变量，保存对单例对象的引用，进而避免对象的重复创建。

编码实现

HTML 部分：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>单例模式真题解读</title>
</head>
<style>
  #modal {
    height: 300px;
    width: 300px;
    line-height: 300px;
    position: fixed;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    border: 1px solid red;
    text-align: center;
  }
</style>
<body>
  <button id='open'>打开 Modal 弹层</button>
  <button id='close'>关闭 Modal 弹层</button>
</body>
```

JS部分：

```
// 点击打开按钮展示模态框
document.getElementById('open').addEventListener('click', function() {
  // 未点击则不创建modal实例，避免不必要的内存占用;此处不用 new Modal 的形式调用也可以，和 Storage 同理
  const modal = createModal()
  modal.style.display = 'block'
})

// 点击关闭按钮隐藏模态框
document.getElementById('close').addEventListener('click', function() {
  const modal = createModal()
  if(modal) {
    modal.style.display = 'none'
  }
})

// 核心逻辑，这里采用了闭包思路来实现单例模式
const createModal = (function() {
  let modal = null
  return function() {
    if(!modal) {
      modal = document.createElement('div')
      modal.innerHTML = 'Modal模态框，全局唯一'
      modal.id = 'modal'
      modal.style.display = 'none'
      document.body.appendChild(modal)
    }
    return modal
  }
})()
```

}