设计模式

设计模式总的来说是一个抽象的概念，前人通过无数次的实践总结出的一套写代码的方式，通过这种方式写的代码可以让别人更加容易阅读、维护以及复用。

这一章节我们将来学习几种最常用的设计模式。

工厂模式

工厂模式分为好几种，这里就不一一讲解了，以下是一个简单工厂模式的例子

```js
class Man {
  constructor(name) {
    this.name = name
  }
  alertName() {
    alert(this.name)
  }
}

class Factory {
  static create(name) {
    return new Man(name)
  }
}

Factory.create('yck').alertName()
```

当然工厂模式并不仅仅是用来 new 出**实例**。

可以想象一个场景。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码，只需要你提供给我一个接口去调用，用户只负责传递需要的参数，至于这些参数怎么使用，内部有什么逻辑是不关心的，只需要你最后返回我一个实例。这个构造过程就是工厂。

工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰。

在 Vue 源码中，你也可以看到工厂模式的使用，比如创建异步组件

```js
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {

    // 逻辑处理...

  const vnode = new VNode(
    `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
    data, undefined, undefined, undefined, context,
    { Ctor, propsData, listeners, tag, children },
    asyncFactory
  )

  return vnode
}
```

在上述代码中，我们可以看到我们只需要调用 `createComponent` 传入参数就能创建一个组件实例，但是创建这个实例是很复杂的一个过程，工厂帮助我们隐藏了这个复杂的过程，只需要一句代码调用就能实现功能。

单例模式

单例模式很常用，比如全局缓存、全局状态管理等等这些只需要一个对象，就可以使用单例模式。

单例模式的核心就是保证全局只有一个对象可以访问。因为 JS 是门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中，我们只需要用一个变量确保实例只创建一次就行，以下是如何实现单例模式的例子

```js
class Singleton {
  constructor() {}
}

Singleton.getInstance = (function() {
  let instance
  return function() {
    if (!instance) {
      instance = new Singleton()
    }
    return instance
  }
})()

let s1 = Singleton.getInstance()
let s2 = Singleton.getInstance()
console.log(s1 === s2) // true
```

在 Vuex 源码中，你也可以看到单例模式的使用，虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 Vuex

```js
let Vue // bind on install

export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    // 如果发现 Vue 有值，就不重新创建实例了
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

适配器模式

适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。

以下是如何实现适配器模式的例子

```js
class Plug {
  getName() {
    return '港版插头'
  }
}

class Target {
  constructor() {
    this.plug = new Plug()
  }
  getName() {
    return this.plug.getName() + ' 适配器转二脚插头'
  }
}

let target = new Target()
target.getName() // 港版插头 适配器转二脚插头
```

在 Vue 中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属性，组件内部需要将时间戳转为正常的日期显示，一般会使用 `computed` 来做转换这件事情，这个过程就使用到了适配器模式。

装饰模式

装饰模式不需要改变已有的接口，作用是给对象添加功能。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身，给手机添加了保护套提供防摔功能。

以下是如何实现装饰模式的例子，使用了 ES7 中的装饰器语法

```js
function readonly(target, key, descriptor) {
  descriptor.writable = false
  return descriptor
}

class Test {
  @readonly
  name = 'yck'
}

let t = new Test()

t.yck = '111' // 不可修改
```

在 React 中，装饰模式其实随处可见

```js
import { connect } from 'react-redux'
class MyComponent extends React.Component {
    // ...
}
export default connect(mapStateToProps)(MyComponent)
```

代理模式

代理是为了控制对对象的访问，不让外部直接访问到对象。在现实生活中，也有很多代理的场景。比如你需要买一件国外的产品，这时候你可以通过代购来购买产品。

在实际代码中其实代理的场景很多，也就不举框架中的例子了，比如事件代理就用到了代理模式。

```html
<ul id="ul">
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
</ul>
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

因为存在太多的 `li`，不可能每个都去绑定事件。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。

发布-订阅模式

发布-订阅模式也叫做观察者模式。通过一对一或者一对多的依赖关系，当对象发生改变时，订阅方都会收到通知。在现实生活中，也有很多类似场景，比如我需要在购物网站上购买一个产品，但是发现该产品目前处于缺货状态，这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。

在实际代码中其实发布-订阅模式也很常见，比如我们点击一个按钮触发了点击事件就是使用了该模式

```html
<ul id="ul"></ul>
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

在 Vue 中，如何实现响应式也是使用了该模式。对于需要实现响应式的对象来说，在 `get` 的时候会进行依赖收集，当改变了对象的属性时，就会触发派发更新。

外观模式

外观模式提供了一个接口，隐藏了内部的逻辑，更加方便外部调用。

举个例子来说，我们现在需要实现一个兼容多种浏览器的添加事件方法

```js
function addEvent(elm, evType, fn, useCapture) {
  if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture)
    return true
  } else if (elm.attachEvent) {
    var r = elm.attachEvent("on" + evType, fn)
    return r
  } else {
    elm["on" + evType] = fn
  }
}
```

对于不同的浏览器，添加事件的方式可能会存在兼容问题。如果每次都需要去这样写一遍的话肯定是不能接受的，所以我们将这些判断逻辑统一封装在一个接口中，外部需要添加事件只需要调用 `addEvent` 即可。

小结

这一章节我们学习了几种常用的设计模式。其实设计模式还有很多，有一些内容很简单，我就没有写在章节中了，比如迭代器模式、原型模式，有一些内容也是不经常使用，所以也就不一一列举了。

如果你还想了解更多关于设计模式的内容，可以阅读这本书。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论

全部评论（33）

棕色玩具熊 JY.1 | 14天前
发布-订阅模式也叫做观察者模式。
就这。。。还写小册子，我真是MMP了 也不能退钱，吃屎了
点赞 回复

于先森 LV.3 JY.4 | 前端工程师 @ 菜的睡不... | 1年前
在修言设计模式那本小册中明确提到了观察者模式和订阅发布模式相对比区别在于是否有第三方介入 发布订阅模式需要第三方去维持触发和注册组件所需事件 而观察者由发布者维持订阅痴
2 回复

理想永远都年轻 LV.3 JY.3 | 前端 @ 去哪儿网 | 2年前
发布-订阅者模式和观察者模式不一样
2 回复

FruitBro LV.5 JY.3 | 前端开发工程师 | 2年前
装饰模式，在在 React 中的例子，connect不是高阶函数么，好像没有涉及到装饰模式...
2 回复

你若像风 LV.2 JY.1 | 前端开发工程师 | 2年前
讲得很好，了解了很多。
1 回复

雪无止尽 JY.4 | 前端工程师 @ 龙湖 | 2年前
那位大佬能给解释一下适配器模式和外观模式的区别?
点赞 回复

Tag JY.5 | Web Developer | 2年前
太水了
3 回复

明天下雨不想说话 JY.1 | 2年前
太垃圾了
2 回复

chiuwingyan LV.3 JY.4 | 前端工程师 | 3年前
发布订阅者模式和观察者模式只能说是类似，但是不是一样的
2 回复

星星_star JY.2 | 3年前
太水了
5 回复

邹小邹 LV.5 JY.5 | 大前端负责人-大量招人... | 3年前
正本看下来，太垃圾了
11 回复

kingll JY.4 | web前端 | 3年前
使用外观模式在增加子系统的时候会修改外观类内部，违反开放封闭原则，具体是否使用还是要结合场景。
点赞 回复

wenting68456 JY.1 | 4年前
适配器模式不是很懂，太抽象了
1 回复

sammui4 JY.3 | 切图仔 | 4年前
居然没有策略模式 = =
点赞 回复

嘿黑 JY.3 | 4年前
发布订阅模式也叫观察者模式？两个并不是一回事吧
13 4

> **卷家老大** | 4年前
> 应该是一回事
> 点赞 回复
>
> **incuisting** | 4年前
> 不是一回事吧
> 点赞 回复
>
> 查看更多回复

泡泡君61109 JY.3 | 4年前
不太明白单例模式中，Singleton.getInstance为什么要是一个IIFE，尝试着改成
Singleton.getInstance = function() {
let instance
return function() {
if (!instance) {
instance = new Singleton()...
展开
点赞 7

> **泡泡君61109** | 4年前
> 是因为要通过IIFE来创建一个作用域吗？这样返回的函数就会形成闭包，拥有对这个作用域中instance的访问权限？
> 点赞 回复
>
> **1同志42022** 回复 **泡泡君61109** | 4年前
> 你说的是对的，其实目的就是不想创建一个全局变量instance而已
> “是因为要通过IIFE来创建一个作用域吗？这样返回的函数就会形成闭包，拥有对这个... ”
> 点赞 回复
>
> 查看更多回复

烛火星光1 JY.4 | 前端工程师 | 4年前
发布订阅模式和代理模式的两端js代码一模一样，那么这个利用父节点作为事件代理的函数，究竟是代理模式还是发布订阅模式？还是两者都是？如果两者都可以，那么这两者的区别究竟是啥呢？
点赞 3

> **yck**（作者） | 4年前
> 发布订阅模式核心是 addEventListener...
> 点赞 回复
>
> **zexiplus** | 4年前
> 本来应该绑定在li元素上的事件，代理到了父元素ul上，这一点是代理模式的体现，而对dom事件的监听（addEventListener）是发布订阅模式的体现，这段代码两个模式都有体现
> 点赞 回复
>
> 查看更多回复

不曾亏欠 LV.3 JY.4 | 胶水工程师 | 4年前
点赞！
1 回复