

# 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

```
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

```

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

```
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

```
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

```
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

```
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

```
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。

```
<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



```
<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

```
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](https://book.douban.com/subject/26382780/) (<https://book.douban.com/subject/26382780/>)。