设计模式

- 设计模式是我们在 **解决问题的时候针对特定问题给出的简洁而优化的处理方案**
- 我们有很多的设计模式
  - 单例模式
  - 组合模式
  - 观察者模式
  - …
- 今天我们就聊一下这三个设计模式
  - 单例模式 / 组合模式 / 观察者模式

单例模式

- 什么是单例模式呢？
- 我们都知道，构造函数可以创造一个对象
- 我们 new 很多次构造函数就能得到很多的对象
- 单例模式： 就是使用构造函数实例化的时候，不管实例化多少回，都是同一个对象
  - 也就是一个构造函数一生只能 new 出一个对象
- 也就是说，当我们使用构造函数，每一次 new 出来的对象 属性/功能/方法 **完全一样** 的时候，我们把他设计成单例模式

核心代码

- 单例模式的核心代码很简单
- 其实就是判断一下，他曾经有没有 new 出来过对象
- 如果有，就还继续使用之前的那个对象，如果没有，那么就给你 new 一个

```
// 准备一个构造函数
// 将来要 new 的
function Person() {}

// 准备一个单例模式函数
// 这个单例模式函数要把 Person 做成一个单例模式
// 将来再想要 new Person 的时候只要执行这个 singleton 函数就可以了
function singleton () {
 let instance

```

```
  if (!instance) { // 如果 instance 没有内容
  // 来到这里，证明 instance 没有内容
  // 给他赋值为 new Person
  instance = new Person()
  }

  // 返回的永远都是第一次 new Person 的实例
  // 也就是永远都是一个实例
  return instance
}

const p1 = singleton()
const p2 = singleton()
console.log(p1 === p2) // true
```

应用

- 我们就用这个核心代码简单书写一个 demo

```
// 这个构造函数的功能就是创建一个 div，添加到页面中
function CreateDiv() {
 this.div = document.createElement('div')
 document.body.appendChild(this.div)
}

CreateDiv.prototype.init = function (text) {
 this.div.innerHTML = text
}

// 准备把这个 CreateDiv 做成单例模式
// 让 singleton 成为一个闭包函数
const singleton = (function () {

 let instance

 return function (text) {
 if (!instance) {
 instance = new CreateDiv()
 }
 instance.init(text)
 return instance
```

```
  }
})()

singleton('hello') // 第一次的时候，页面中会出现一个新的 div ，内容是 hello
singleton('world') // 第二次的时候，不会出现新的 div，而是原先的 div 内容变成了 world
```

组合模式

- 组合模式，就是把几个构造函数的启动方式组合再一起
- 然后用一个 ”遥控器“ 进行统一调用

```
class GetHome {

  init () {
  console.log('到家了')
  }
}

class OpenComputer {

  init () {
  console.log('打开电脑')
  }
}

class PlayGame {

  init () {
  console.log('玩游戏')
  }
}
```

  - 上面几个构造函数的创造的实例化对象的 **启动方式** 都一致
  - 那么我们就可以把这几个函数以组合模式的情况书写
  - 然后统一启动

- 准备一个 **组合模式** 的构造函数

```
class Compose {
  constructor () {
  this.compose = []
```

```
    this.compose = []
  }

  // 添加任务的方法
  add (task) {
  this.compose.push(task)
  }

  // 一个执行任务的方法
  execute () {
  this.compose.forEach(item => {
  item.init()
  })
  }
}
```

- 我们就用我们的组合模式构造函数来吧前面的几个功能组合起来

```
const c = new Compose()
// 把所有要完成的任务都放在队列里面
c.add(new GetHome())
c.add(new OpenComputer)
c.add(new PlayGame)

// 直接器动任务队列
c.execute()
// 就会按照顺序执行三个对象中的 init 函数
```

观察者模式

- 观察者模式，通常也被叫做 **发布-订阅模式** 或者 **消息模式**
- 英文名称叫做 `Observer`
- 官方解释： 当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，解决了主体对象与观察者之间功能的耦合，即一个对象状态改变给其他对象通知的问题
- 听起来很迷糊，但是其实没有很难

一个例子

- 当你想去书店买书，但是恰巧今天你要买的书没有了
- 我们又不能总在书店等着，就把我们的手机留给店员
- 当你需要的书到了的时候，他会打电话通知你，你去买了就好了

- 你买到数了以后，就告诉他，我买到了，那么以后再来了书就不会通知你了

addEventListener

- 上面的例子可能还不是很明确
- 但是 `addEventListener` 是一个我们都用过的东西
- 这个东西其实就是一个标准的 **观察者模式**

```
btn.addEventListener('click', function () {
  console.log('btn 被点击了')
})
```

  - 上面这个就是有一个 **无形的观察者** 再观察着 `btn` 的一举一动
  - 当这个 `btn` 被点击的时候，就会执行 对应的函数
  - 我们也可以多绑定几个函数
- 说白了： 观察者模式就是我们自己实现一个 `addEventListener` 的功能
  - 只不过 `addEventListaner` 只有固定的一些事件，而且只能给 dom 元素绑定
  - 而我们自己写的可以随便绑定一个事件名称，自己选择触发时机而已

书写代码

- 首先我们分析功能
  - 我们要有一个观察者（这里抽象为一个对象 `{}` ）
  - 需要有一个属性，存放消息的盒子（把你绑定的所有事件放在里面）
  - 需要一个 on 方法，用于添加事件
  - 需要一个 emit 方法，用于发布事件（触发）
  - 需要一个 off 方法，把已经添加的方法取消

```
const observer = {
  message: {},
  on: function () {},
  emit: function () {},
  off: function () {}
}
```

  - 我们把它写成一个构造函数的形式

```
class Observer {
 constructor () {
 this.message = {}
 }

 on () {}

 emit () {}

 off () {}
}
```

- 现在，一个观察者的雏形就出来了
- 接下来完善方法就可以了

ON

- 先来写 ON 方法
- 添加一个事件
- 我们的 on 方法需要接受 两个参数
  - 事件类型
  - 事件处理函数

```
class Observer {
 constructor () {
 this.message = {}
 }

 on (type, fn) {
 // 判断消息盒子里面有没有设置事件类型
 if (!this.message[type]) {
 // 证明消息盒子里面没有这个事件类型
 // 那么我们直接添加进去
 // 并且让他的值是一个数组，再数组里面放上事件处理函数
 this.message[type] = [fn]
 } else {
 // 证明消息盒子里面有这个事件类型
 // 那么我们直接向数组里面追加事件处理函数就行了
 this.message[type].push(fn)
```

```
  }
  }

  emit () {}

  off () {}
 }
```

EMIT

- 接下来就是发布事件
- 也就是让我们已经订阅好的事件执行一下
- 同样需要接受两个参数
  - 要触发的事件类型
  - 给事件处理函数传递的参数

```
 class Observer {
  constructor () {
  this.message = {}
  }

  on (type, fn) {
  // 判断消息盒子里面有没有设置事件类型
  if (!this.message[type]) {
  // 证明消息盒子里面没有这个事件类型
  // 那么我们直接添加进去
  // 并且让他的值是一个数组，再数组里面放上事件处理函数
  this.message[type] = [fn]
  } else {
  // 证明消息盒子里面有这个事件类型
  // 那么我们直接向数组里面追加事件处理函数就行了
  this.message[type].push(fn)
  }
  }

  emit (type, ...arg) {
  // 判断你之前有没有订阅过这个事件
  if (!this.message[type]) return

  // 如果有，那么我们就处理一下参数
```

```
25.   const event = {
26.   type: type,
27.   arg: arg || {}
28.   }
29.
30.   // 循环执行为当前事件类型订阅的所有事件处理函数
31.   this.message[type].forEach(item => {
32.   item.call(this, event)
33.   })
34.   }
35.
36.   off () {}
37.  }
```

OFF

- 最后就是移除事件
- 就是把已经订阅的事件处理函数移除掉
- 同样需要接受两个参数
  - 要移除的事件类型
  - 要移除的事件处理函数

```
1.  class Observer {
2.   constructor () {
3.   this.message = {}
4.   }
5.
6.   on (type, fn) {
7.   // 判断消息盒子里面有没有设置事件类型
8.   if (!this.message[type]) {
9.   // 证明消息盒子里面没有这个事件类型
10.   // 那么我们直接添加进去
11.   // 并且让他的值是一个数组，再数组里面放上事件处理函数
12.   this.message[type] = [fn]
13.   } else {
14.   // 证明消息盒子里面有这个事件类型
15.   // 那么我们直接向数组里面追加事件处理函数就行了
16.   this.message[type].push(fn)
17.   }
18.   }
```

```
19.
20.   emit (type, ...arg) {
21.   // 判断你之前有没有订阅过这个事件
22.   if (!this.message[type]) return
23.
24.   // 如果有，那么我们就处理一下参数
25.   const event = {
26.   type: type,
27.   arg: arg || {}
28.   }
29.
30.   // 循环执行为当前事件类型订阅的所有事件处理函数
31.   this.message[type].forEach(item => {
32.   item.call(this, event)
33.   })
34.   }
35.
36.   off (type, fn) {
37.   // 判断你之前有没有订阅过这个事件
38.   if (!this.message[type]) return
39.
40.   // 如果有我们再进行移除
41.   for (let i = 0; i < this.message[type].length; i++) {
42.   const item = this.message[type][i]
43.   if (item === fn) {
44.   this.message[type].splice(i, 1)
45.   i--
46.   }
47.   }
48.   }
49.  }
```

- 以上就是最基本的 **观察者模式**
- 接下来我们就使用一下试试看

使用一下

```
1.  const o = new Observer()
2.
3.  // 准备两个事件处理函数
4.  function a(e) {
```

```
  console.log('hello')
}

function b(e) {
  console.log('world')
}

// 订阅事件
o.on('abc', a)
o.on('abc', b)

// 发布事件（触发）
o.emit('abc', '100', '200', '300') // 两个函数都回执行

// 移除事件
o.off('abc', 'b')

// 再次发布事件（触发）
o.emit('abc', '100', '200', '300') // 只执行一个 a 函数了
```