> 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。 —— Graphic Design Patterns

观察者模式，是所有 JavaScript 设计模式中**使用频率**最高，**面试频率也最高**的设计模式，所以说它**十分重要**——如果我是面试官，考虑到面试时间有限、设计模式这块不能多问，我可能在考查你设计模式的时候只会问观察者模式这一个模式。该模式的权重极高，我们此处会花费两个较长的章节把它**掰碎嚼烂**了来掌握。

重点不一定是难点。观察者模式十分重要，但它并不抽象，理解难度不大。这种模式不仅在业务开发中遍地开花，在日常生活中也是非常常见的。为了帮助大家形成初步的理解，在进入代码世界之前，我们照例来看一段日常：

生活中的观察者模式

周一刚上班，前端开发李雷就被产品经理韩梅梅拉进了一个钉钉群——“员工管理系统需求第99次变更群”。这个群里不仅有李雷，还有后端开发 A，测试同学 B。三位技术同学看到这简单直白的群名便立刻做好了接受变更的准备、打算撸起袖子开始干了。此时韩梅梅却说：“别急，这个需求有问题，我需要和业务方再确认一下，大家先各忙各的吧”。这种情况下三位技术同学不必立刻投入工作，但他们都已经做好了**本周需要做一个新需求**的准备，时刻等待着产品经理的号召。

一天过去了，两天过去了。周三下午，韩梅梅终于和业务方确认了所有的需求细节，于是在“员工管理系统需求第99次变更群”里大吼一声：“需求文档来了！”，随后甩出了"需求文档.zip"文件，同时@所有人。三位技术同学听到熟悉的“有人@我”提示音，立刻点开群进行群消息和群文件查收，随后根据群消息和群文件提供的需求信息，投入到了各自的开发里。上述这个过程，就是一个典型的**观察者模式**。

重点角色对号入座

观察者模式有一个“别名”，叫 `发布 - 订阅模式` （之所以别名加了引号，是因为两者之间存在着细微的差异，下个小节里我们会讲到这点）。这个别名非常形象地诠释了观察者模式里两个核心的角色要素——**“发布者”与“订阅者”**。在上述的过程中，需求文档（目标对象）的发布者只有一个——产品经理韩梅梅。而需求信息的接受者却有多个——前端、后端、测试同学，这些同学的共性就是他们需要根据需求信息开展自己后续的工作、因此都非常关心这个需求信息，于是不得不时刻关注着这个群的群消息提醒，他们是实打实的**订阅者**，即观察者对象。

现在我们再回过头来看一遍开头我们提到的略显抽象的定义：

> 观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个目标对象，当这个目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。

在我们上文这个钉钉群里，一个需求信息对象对应了多个观察者（技术同学），当需求信息对象的状态发生**变化**（从无到有）时，产品经理通知了群里的所有同学，以便这些同学接收信息进而开展工作：角色划分 --> 状态变化 --> 发布者通知到订阅者，这就是观察者模式的“套路”。

在实践中理解定义

结合我们上面的分析，现在大家知道，在观察者模式里，至少应该有两个关键角色是一定要出现的——发布者和订阅者。用面向对象的方式表达的话，那就是要有**两个类**。

首先我们来看这个代表发布者的类，我们给它起名叫Publisher。这个类应该具备哪些“基本技能”呢？大家回忆一下上文中的韩梅梅，韩梅梅的基本操作是什么？首先是拉群（增加订阅者），然后是@所有人（通知订阅者），这俩是最明显的了。此外作为群主&产品经理，韩梅梅还具有踢走项目组成员（移除订阅者）的能力。OK，产品经理发布者类的三个基本能力齐了，下面我们开始写代码：

```
// 定义发布者类
class Publisher {
  constructor() {
    this.observers = []
    console.log('Publisher created')
  }
  // 增加订阅者
  add(observer) {
    console.log('Publisher.add invoked')
    this.observers.push(observer)
  }
  // 移除订阅者
  remove(observer) {
    console.log('Publisher.remove invoked')
    this.observers.forEach((item, i) => {
      if (item === observer) {
        this.observers.splice(i, 1)
      }
    })
  }
  // 通知所有订阅者
  notify() {
    console.log('Publisher.notify invoked')
    this.observers.forEach((observer) => {
      observer.update(this)
    })
  }
}
```

ok，搞定了发布者，我们一起来想想订阅者能干啥——其实订阅者的能力非常简单，作为被动的一方，它的行为只有两个——被通知、去执行（本质上是接受发布者的调用，这步我们在Publisher中已经做掉了）。既然我们在Publisher中做的是方法调用，那么我们在订阅者类里要做的就是**方法的定义**：

```
// 定义订阅者类
class Observer {
    constructor() {
        console.log('Observer created')
    }

    update() {
        console.log('Observer.update invoked')
    }
}
```

以上，我们就完成了最基本的发布者和订阅者类的设计和编写。在实际的业务开发中，我们所有的定制化的发布者/订阅者逻辑都可以基于这两个基本类来改写。比如我们可以通过拓展发布者类，来使所有的订阅者来**监听某个特定状态的变化**。仍然以开篇的例子为例，我们让开发者们来监听需求文档（prd）的变化：

```
// 定义一个具体的需求文档（prd）发布类
class PrdPublisher extends Publisher {
    constructor() {
```

```
        super()
        // 初始化需求文档
        this.prdState = null
        // 韩梅梅还没有拉群，开发群目前为空
        this.observers = []
        console.log('PrdPublisher created')
    }

    // 该方法用于获取当前的prdState
    getState() {
        console.log('PrdPublisher.getState invoked')
        return this.prdState
    }

    // 该方法用于改变prdState的值
    setState(state) {
        console.log('PrdPublisher.setState invoked')
        // prd的值发生改变
        this.prdState = state
        // 需求文档变更，立刻通知所有开发者
        this.notify()
    }
}
```

作为订阅方，开发者的任务也变得具体起来：接收需求文档、并开始干活：

```
class DeveloperObserver extends Observer {
    constructor() {
        super()
        // 需求文档一开始还不存在，prd初始为空对象
        this.prdState = {}
        console.log('DeveloperObserver created')
    }

    // 重写一个具体的update方法
    update(publisher) {
        console.log('DeveloperObserver.update invoked')
        // 更新需求文档
        this.prdState = publisher.getState()
        // 调用工作函数
        this.work()
    }

    // work方法，一个专门搬砖的方法
    work() {
        // 获取需求文档
        const prd = this.prdState
        // 开始基于需求文档提供的信息搬砖。。。
        ...
        console.log('996 begins...')
    }
}
```

下面，我们可以 new 一个 PrdPublisher 对象（产品经理），她可以通过调用 setState 方法来更新需求文档。需求文档每次更新，都会紧接着调用 notify 方法来通知所有开发者，这就实现了定义里所谓的：

> 目标对象的状态发生变化时，会通知所有观察者对象，使它们能够自动更新。

OK，下面我们来看看韩梅梅和她的小伙伴们是如何搞事情的吧：

```
// 创建订阅者：前端开发李雷
const liLei = new DeveloperObserver()
// 创建订阅者：服务端开发小A（sorry。。。起名字真的太难了）
const A = new DeveloperObserver()
// 创建订阅者：测试同学小B
const B = new DeveloperObserver()
// 韩梅梅出现了
const hanMeiMei = new PrdPublisher()
// 需求文档出现了
const prd = {
    // 具体的需求内容
    ...
}
// 韩梅梅开始拉群
hanMeiMei.add(liLei)
hanMeiMei.add(A)
hanMeiMei.add(B)
// 韩梅梅发送了需求文档，并@了所有人
hanMeiMei.setState(prd)
```

以上，就是观察者模式在代码世界里的完整实现流程了。

相信走到这一步，大家对观察者模式的核心思想、基本实现模式都有了不错的掌握。下面我们趁热打铁，一起来看看如何凭借观察者模式，在面试中表演真正的技术~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）