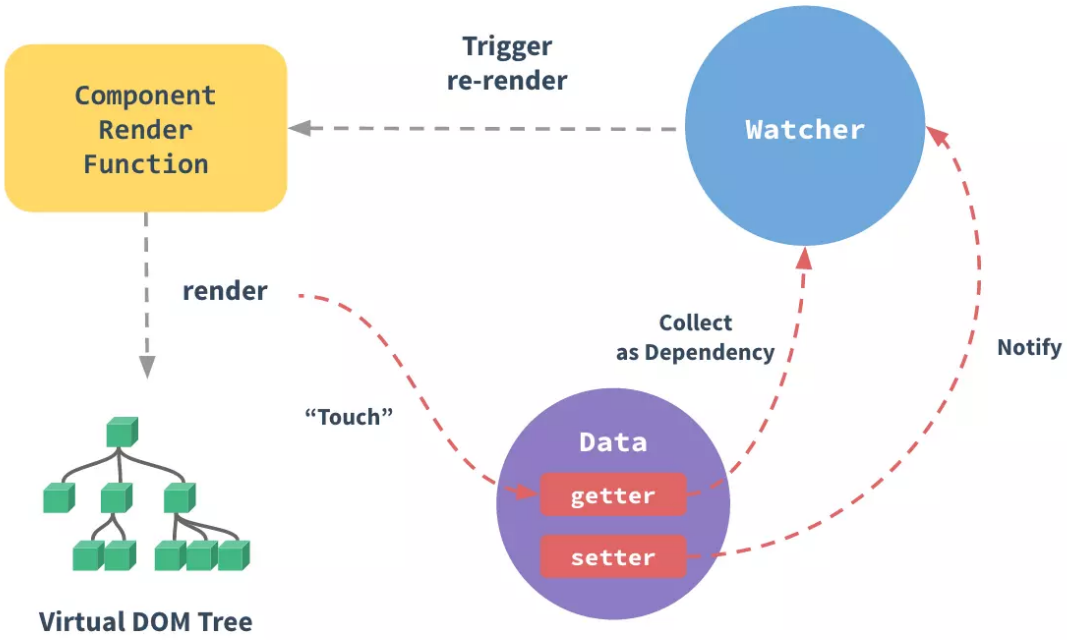
上节我们说过，观察者模式作为一个超高频考点，在设计模式中具有举足轻重的地位。单从面试的维度来说，说它是最重要的设计模式也不为过。正因为如此，它的面试题变体可以说是五花八门。

不过面试题这东西，和数学题一样，看似变化多端，实则大同小异。大家如果经历的面试足够多，会发现观察者模式考来考去也就是那么几种考法，所谓的“变化多端”，也无非是改个条件改个变量的事情。本节在梳理了大量相关面试题的基础上，为大家总结了观察者模式的四个出题方向。相信有了本节的加持和下节的强化，大家再和面试官聊到观察者模式时，一定可以滔滔不绝、轻松拿下~

Vue数据双向绑定（响应式系统）的实现原理

解析

Vue 框架是热门的渐进式 JavaScript框架。在 Vue 中，当我们修改状态时，视图会随之更新，这就是Vue的数据双向绑定（又称响应式原理）。数据双向绑定是Vue 最独特的特性之一。如果读者没有接触过Vue，强烈建议阅读**Vue官方对响应式原理的介绍**。此处我们用官方的一张流程图来简要地说明一下Vue响应式系统的整个流程：

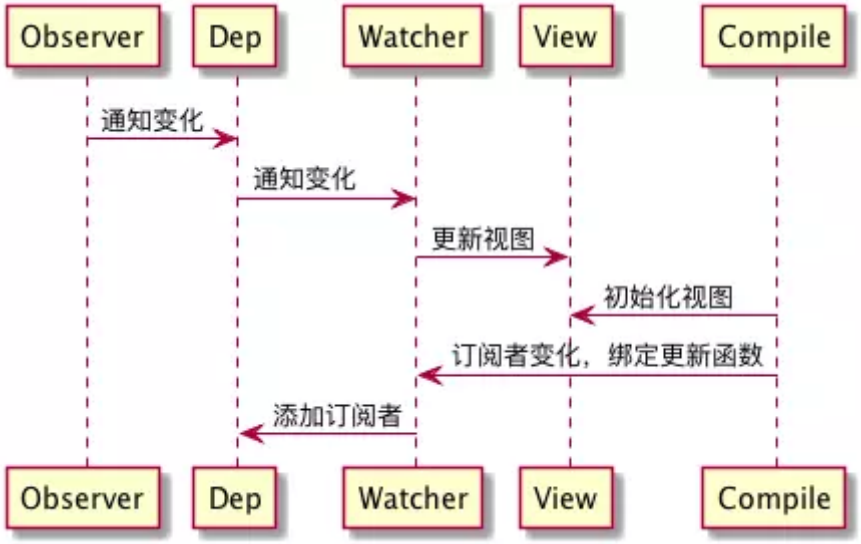


在 Vue 中，每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新——这是一个典型的观察者模式。这道面试题考察了受试者对Vue底层原理的理解、对观察者模式的实现能力以及一系列重要的JS知识点，具有较强的综合性和代表性。

值得注意的是，在面试过程中，面试官多数情况下不会要求大家写出完整的响应式原理实现代码，而是要求你“说说自己的理解”。在本节，我们不会带大家一行一行写代码（具体深入Vue框架的相关知识，建议大家阅读**Vue源码**及这本**专门写Vue的小册**。），而是针对Vue响应式系统中与观察者模式紧密关联的这部分知识作讲解，帮助大家捋清楚整套流程里的来龙去脉、加深对观察者模式的理解。

在Vue数据双向绑定的实现逻辑里，有这样三个关键角色：

- observer（监听器）：注意，此 observer 非彼 observer。在我们上节的解析中，observer 作为设计模式中的一个角色，代表“订阅者”。但在Vue数据双向绑定的角色结构里，所谓的 observer 不仅是一个数据监听器，它还需要对监听到的数据进行**转发**——也就是说它**同时还是一个发布者**。
- watcher（订阅者）：observer 把数据转发给了**真正的订阅者**——watcher对象。watcher 接收到新的数据后，会去更新视图。
- compile（编译器）：MVVM 框架特有的角色，负责对每个节点元素指令进行扫描和解析，指令的数据初始化、订阅者的创建这些“杂活”也归它管~ 这三者的配合过程如图所示：



OK，实现方案搞清楚了，下面我们给整个流程中**涉及到发布-订阅这一模式的代码**来个特写：

核心代码

实现observer

首先我们需要实现一个方法，这个方法会对需要监听的数据对象进行遍历、给它的属性加上定制的 getter 和 setter 函数。这样但凡这个对象的某个属性发生了改变，就会触发 setter 函数，进而通知到订阅者。这个 setter 函数，就是我们的监听器：

```
// observe方法遍历并包装对象属性
function observe(target) {
    // 若target是一个对象，则遍历它
    if(target && typeof target === 'object') {
        Object.keys(target).forEach((key)=> {
            // defineReactive方法会给目标属性装上“监听器”
            defineReactive(target, key, target[key])
        })
    }
}

// 定义defineReactive方法
function defineReactive(target, key, val) {
    // 属性值也可能是object类型，这种情况下需要调用observe进行递归遍历
    observe(val)
    // 为当前属性安装监听器
    Object.defineProperty(target, key, {
         // 可枚举
        enumerable: true,
        // 不可配置
```

```
        configurable: false,
        get: function () {
            return val;
        },
        // 监听器函数
        set: function (value) {
            console.log(`${target}属性的${key}属性从${val}值变成了了${value}`)
            val = value
        }
    });
}
```

下面实现订阅者 Dep：

```
// 定义订阅者类Dep
class Dep {
    constructor() {
        // 初始化订阅队列
        this.subs = []
    }

    // 增加订阅者
    addSub(sub) {
        this.subs.push(sub)
    }

    // 通知订阅者（是不是所有的代码都似曾相识？）
    notify() {
        this.subs.forEach((sub)=>{
            sub.update()
        })
    }
}
```

现在我们可以改写 defineReactive 中的 setter 方法，在监听器里去通知订阅者了：

```
function defineReactive(target, key, val) {
    const dep = new Dep()
    // 监听当前属性
    observe(val)
    Object.defineProperty(target, key, {
        set: (value) => {
            // 通知所有订阅者
            dep.notify()
        }
    })
}
```

实现一个Event Bus/ Event Emitter

Event Bus（Vue、Flutter 等前端框架中有出镜）和 Event Emitter（Node中有出镜）出场的“剧组”不同，但是它们都对应一个共同的角色——**全局事件总线**。 全局事件总线，严格来说不能说是观察者模式，而是发布-订阅模式（具体的概念甄别我们会在下个小节着重讲）。它在我们日常的业务开发中应用非常广。

上节开篇我说过，如果只能考一个设计模式的面试题，我一定会出观察者模式。

这句话接着往下说，如果只能选一道题，那这道题一定是 Event Bus/Event Emitter 的代码实现——我都说这么清楚了，这个知识点到底要不要掌握、需要掌握到什么程度，就看各位自己的了。

在Vue中使用Event Bus来实现组件间的通讯

Event Bus/Event Emitter 作为全局事件总线，它起到的是一个**沟通桥梁**的作用。我们可以把它理解为一个事件中心，我们所有事件的订阅/发布都不能由订阅方和发布方“私下沟通”，必须要委托这个事件中心帮我们实现。 在Vue中，有时候 A 组件和 B 组件中间隔了很远，看似没什么关系，但我们希望它们之间能够通信。这种情况下除了求助于 Vuex 之外，我们还可以通过 Event Bus 来实现我们的需求。

创建一个 Event Bus（本质上也是 Vue 实例）并导出：

```
const EventBus = new Vue()
export default EventBus
```

在主文件里引入EventBus，并挂载到全局：

```
import bus from 'EventBus的文件路径'
Vue.prototype.bus = bus
```

订阅事件：

```
// 这里func指someEvent这个事件的监听函数
this.bus.$on('someEvent', func)
```

发布（触发）事件：

```
// 这里params指someEvent这个事件被触发时回调函数接收的入参
this.bus.$emit('someEvent', params)
```

大家会发现，整个调用过程中，没有出现具体的发布者和订阅者（比如上节的PrdPublisher和DeveloperObserver），全程只有bus这个东西一个人在疯狂刷存在感。这就是全局事件总线的特点——所有事件的发布/订阅操作，必须经由事件中心，禁止一切“私下交易”！

下面，我们就一起来实现一个Event Bus（注意看注释里的解析）：

```
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)
      })
    }
  }

  // 移除某个事件回调队列里的指定回调函数
  off(eventName, cb) {
    const callbacks = this.handlers[eventName]
    const index = callbacks.indexOf(cb)
    if (index !== -1) {
      callbacks.splice(index, 1)
    }
  }

  // 为事件注册单次监听器
  once(eventName, cb) {
    // 对回调函数进行包装，使其执行完毕自动被移除
    const wrapper = (...args) => {
      cb.apply(...args)
      this.off(eventName, wrapper)
    }
    this.on(eventName, wrapper)
  }
}
```

在日常的开发中，大家用到EventBus/EventEmitter往往提供比这五个方法多的多的多的方法。但在面试过程中，如果大家能够完整地实现出这五个方法，已经非常可以说明问题了，因此楼上这个EventBus希望大家可以熟练掌握。学有余力的同学，推荐阅读**FaceBook推出的通用EventEmiiter库的源码**，相信你会有更多收获。

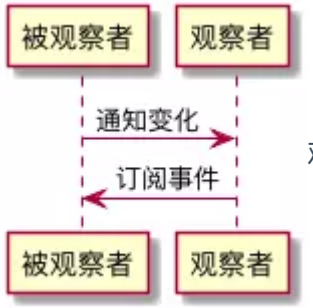
观察者模式与发布-订阅模式的区别是什么?

在面试过程中，一些对细节比较在意的面试官可能会追问观察者模式与发布-订阅模式的区别。这个问题可能会引发一些同学的不适，因为在大量参考资料以及已出版的纸质书籍中，都会告诉大家“发布-订阅模式和观察者模式是同一个东西的两个名字”。本书在前文的叙述中，也没有突出强调两者的区别。其实这两个模式，要较起真来，确实不能给它们划严格的等号。

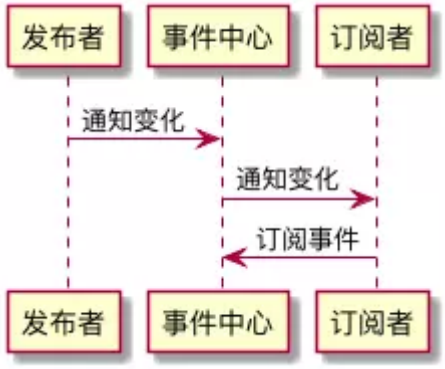
为什么大家都喜欢给它们强行划等号呢？这是因为就算划了等号，也不影响我们正常使用，毕竟两者在核心思想、运作机制上没有本质的差别。但考虑到这个问题确实可以成为面试题的一个方向，此处我们还是单独拿出来讲一下。

回到我们上文的例子里。韩梅梅把所有的开发者拉了一个群，直接把需求文档丢给每一位群成员，这种**发布者直接触及到订阅者**的操作，叫观察者模式。但如果韩梅梅没有拉群，而是把需求文档上传到了公司统一的需求平台上，需求平台感知到文件的变化、自动通知了每一位订阅了该文件的开发者，这种**发布者不直接触及到订阅者、而是由统一的第三方来完成实际的通信的操作，叫做发布-订阅模式。**

相信大家也已经看出来了，观察者模式和发布-订阅模式之间的区别，在于是否存在第三方、发布者能否直接感知订阅者（如图所示）。



观察者模式



发布-订阅模式

在我们见过的这些例子里，韩梅梅拉钉钉群的操作，就是典型的观察者模式；而通过EventBus去实现事件监听/发布，则属于发布-订阅模式。

既生瑜，何生亮？既然有了观察者模式，为什么还需要发布-订阅模式呢？

大家思考一下：为什么要有观察者模式？观察者模式，解决的其实是模块间的耦合问题，有它在，即便是两个分离的、毫不相关的模块，也可以实现数据通信。但观察者模式仅仅是减少了耦合，**并没有完全地解决耦合问题**——被观察者必须去维护一套观察者的集合，这些观察者必须实现统一的方法供被观察者调用，两者之间还是有着说不清、道不明的关系。

而发布-订阅模式，则是快刀斩乱麻了——发布者完全不用感知订阅者，不用关心它怎么实现回调方法，事件的注册和触发都发生在独立于双方的第三方平台（事件总线）上。发布-订阅模式下，实现了完全地解耦。

但这并不意味着，发布-订阅模式就比观察者模式“高级”。在实际开发中，我们的模块解耦诉求**并非总是需要它们完全解耦**。如果两个模块之间本身存在关联，且这种关联是稳定的、必要的，那么我们使用观察者模式就足够了。而在模块与模块之间独立性较强、且没有必要单纯为了数据通信而强行为两者制造依赖的情况下，我们往往会倾向于使用发布-订阅模式。

（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~）