

Vue 响应式原理模拟

课程目标

- 模拟一个最小版本的 Vue
- 响应式原理在面试的常问问题
- 学习别人优秀的经验，转换成自己的经验
- 实际项目中出问题的原理层面的解决
 - 给 Vue 实例新增一个成员是否是响应式的？
 - 给属性重新赋值成对象，是否是响应式的？
- 为学习 Vue 源码做铺垫

准备工作

- 数据驱动
- 响应式的核心原理
- 发布订阅模式和观察者模式

数据驱动

- 数据响应式、双向绑定、数据驱动
- 数据响应式
 - 数据模型仅仅是普通的 JavaScript 对象，而当我们修改数据时，视图会进行更新，避免了繁琐的 DOM 操作，提高开发效率
- 双向绑定
 - 数据改变，视图改变；视图改变，数据也随之改变
 - 我们可以使用 v-model 在表单元素上创建双向数据绑定
- 数据驱动是 Vue 最独特的特性之一
 - 开发过程中仅需要关注数据本身，不需要关心数据是如何渲染到视图

数据响应式的核心原理

Vue 2.x

- [Vue 2.x深入响应式原理](#)
- [MDN - Object.defineProperty](#)
- 浏览器兼容 IE8 以上（不兼容 IE8）

```
1 // 模拟 vue 中的 data 选项
2 let data = {
3   msg: 'hello'
4 }
5
6 // 模拟 vue 的实例
7 let vm = {}
8
9 // 数据劫持：当访问或者设置 vm 中的成员的时候，做一些干预操作
```

```

10 Object.defineProperty(vm, 'msg', {
11     // 可枚举（可遍历）
12     enumerable: true,
13     // 可配置（可以使用 delete 删除，可以通过 defineProperty 重新定义）
14     configurable: true,
15     // 当获取值的时候执行
16     get () {
17         console.log('get: ', data.msg)
18         return data.msg
19     },
20     // 当设置值的时候执行
21     set (newValue) {
22         console.log('set: ', newValue)
23         if (newValue === data.msg) {
24             return
25         }
26         data.msg = newValue
27         // 数据更改，更新 DOM 的值
28         document.querySelector('#app').textContent = data.msg
29     }
30 })
31
32 // 测试
33 vm.msg = 'Hello world'
34 console.log(vm.msg)

```

- 如果有一个对象中多个属性需要转换 getter/setter 如何处理？

Vue 3.x

- [MDN - Proxy](#)
- 直接监听对象，而非属性。
- ES 6中新增，IE 不支持，性能由浏览器优化

```

1 // 模拟 vue 中的 data 选项
2 let data = {
3     msg: 'hello',
4     count: 0
5 }
6
7 // 模拟 vue 实例
8 let vm = new Proxy(data, {
9     // 当访问 vm 的成员会执行
10    get (target, key) {
11        console.log('get, key: ', key, target[key])
12        return target[key]
13    },
14    // 当设置 vm 的成员会执行
15    set (target, key, newValue) {
16        console.log('set, key: ', key, newValue)
17        if (target[key] === newValue) {
18            return
19        }
20        target[key] = newValue
21        document.querySelector('#app').textContent = target[key]
22    }
23 })

```

```

24
25 // 测试
26 vm.msg = 'Hello world'
27 console.log(vm.msg)

```

发布订阅模式和观察者模式

发布/订阅模式

- 发布/订阅模式
 - 订阅者
 - 发布者
 - 信号中心

我们假定，存在一个"信号中心"，某个任务执行完成，就向信号中心"发布" (publish) 一个信号，其他任务可以向信号中心"订阅" (subscribe) 这个信号，从而知道什么时候自己可以开始执行。这就叫做"发布/订阅模式" (publish-subscribe pattern)

- Vue 的自定义事件
 - <https://cn.vuejs.org/v2/guide/migration.html#dispatch-%E5%92%8C-broadcast-%E6%9B%BF%E6%8D%A2>

```

1  let vm = new Vue()
2
3  vm.$on('dataChange', () => {
4    console.log('dataChange')
5  })
6
7  vm.$on('dataChange', () => {
8    console.log('dataChange1')
9  })
10
11 vm.$emit('dataChange')

```

- 兄弟组件通信过程

```

1  // EventBus.js
2  // 事件中心
3  let eventHub = new Vue()
4
5  // ComponentA.vue
6  // 发布者
7  addTodo: function () {
8    // 发布消息(事件)
9    eventHub.$emit('add-todo', { text: this.newTodoText })
10   this.newTodoText = ''
11 }
12 // ComponentB.vue
13 // 订阅者
14 created: function () {
15   // 订阅消息(事件)
16   eventHub.$on('add-todo', this.addTodo)
17 }

```

- 模拟 Vue 自定义事件的实现

```

1  class EventEmitter {
2      constructor () {
3          // { eventType: [ handler1, handler2 ] }
4          this.subs = {}
5      }
6      // 订阅通知
7      $on (eventType, handler) {
8          this.subs[eventType] = this.subs[eventType] || []
9          this.subs[eventType].push(handler)
10     }
11     // 发布通知
12     $emit (eventType) {
13         if (this.subs[eventType]) {
14             this.subs[eventType].forEach(handler => {
15                 handler()
16             })
17         }
18     }
19 }
20
21 // 测试
22 var bus = new EventEmitter()
23
24 // 注册事件
25 bus.$on('click', function () {
26     console.log('click')
27 })
28
29 bus.$on('click', function () {
30     console.log('click1')
31 })
32
33 // 触发事件
34 bus.$emit('click')

```

观察者模式

- 观察者(订阅者) -- Watcher
 - update(): 当事件发生时，具体要做的事情
- 目标(发布者) -- Dep
 - subs 数组：存储所有的观察者
 - addSub(): 添加观察者
 - notify(): 当事件发生，调用所有观察者的 update() 方法
- 没有事件中心

```

1  // 目标(发布者)
2  // Dependency
3  class Dep {
4      constructor () {
5          // 存储所有的观察者
6          this.subs = []
7      }
8      // 添加观察者
9      addSub (sub) {
10         if (sub && sub.update) {

```

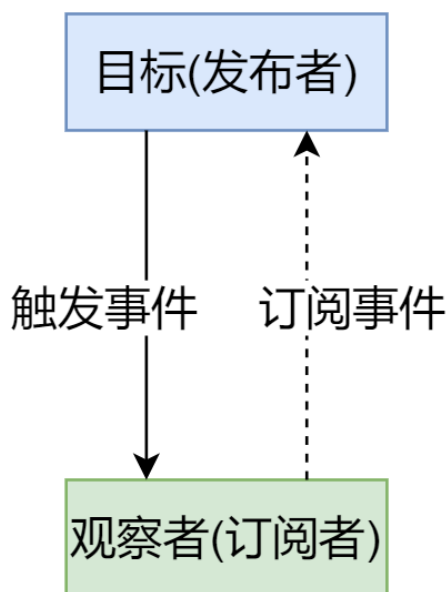
```

11     this.subs.push(sub)
12   }
13 }
14 // 通知所有观察者
15 notify () {
16   this.subs.forEach(sub => {
17     sub.update()
18   })
19 }
20 }
21
22 // 观察者(订阅者)
23 class watcher {
24   update () {
25     console.log('update')
26   }
27 }
28
29 // 测试
30 let dep = new Dep()
31 let watcher = new watcher()
32 dep.addSub(watcher)
33 dep.notify()

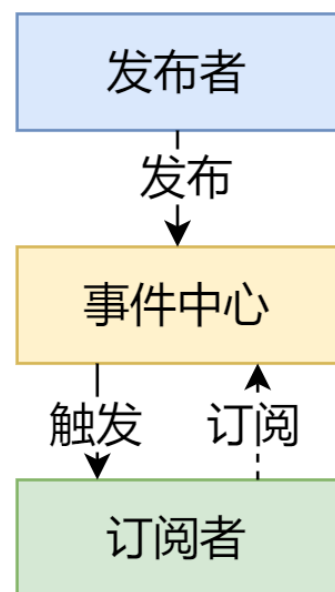
```

总结

- **观察者模式**是由具体目标调度，比如当事件触发，Dep 就会去调用观察者的方法，所以观察者模式的订阅者与发布者之间是存在依赖的。
- **发布/订阅模式**由统一调度中心调用，因此发布者和订阅者不需要知道对方的存在。



观察者模式

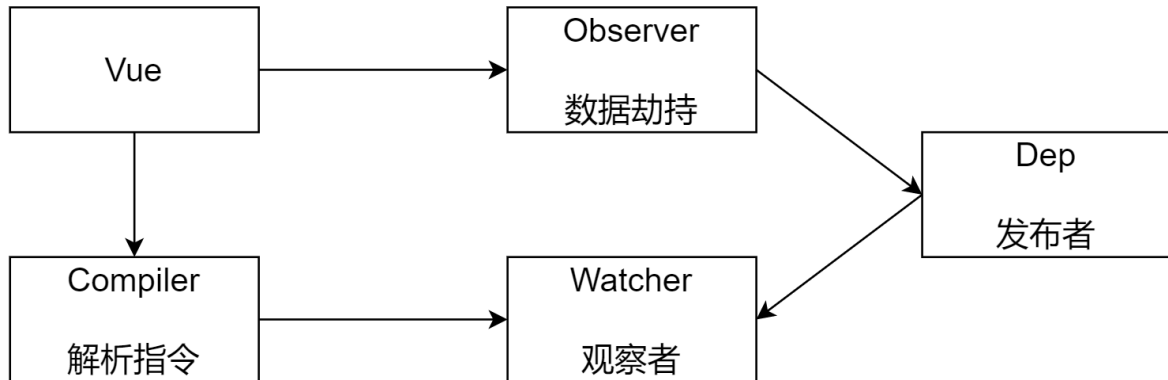


发布订阅模式

Vue 响应式原理模拟

整体分析


- Vue 基本结构
- 打印 Vue 实例观察
- 整体结构



- Vue
 - 把 data 中的成员注入到 Vue 实例，并且把 data 中的成员转成 getter/setter
- Observer
 - 能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知 Dep
- Compiler
 - 解析每个元素中的指令/插值表达式，并替换成相应的数据
- Dep
 - 添加观察者(watcher)，当数据变化通知所有观察者
- Watcher
 - 数据变化更新视图

Vue

- 功能
 - 负责接收初始化的参数(选项)
 - 负责把 data 中的属性注入到 Vue 实例，转换成 getter/setter
 - 负责调用 observer 监听 data 中所有属性的变化
 - 负责调用 compiler 解析指令/插值表达式
- 结构

<div></div> <div>Vue</div>
<div>+ \$options</div> <div>+ \$el</div> <div>+ \$data</div>
<div>- _proxyData()</div>

- 代码

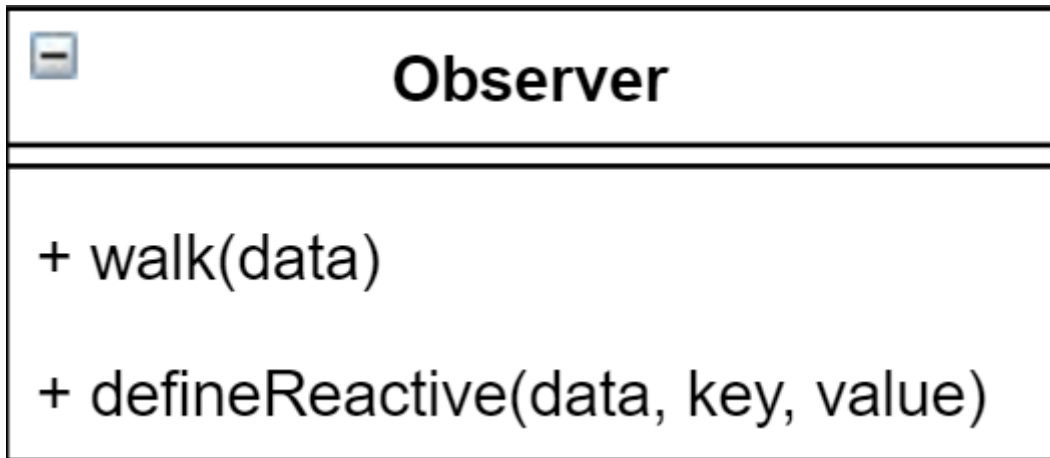
```

1  class Vue {
2    constructor (options) {
3      // 1. 保存选项的数据
4      this.$options = options || {}
5      this.$data = options.data || {}
6      const el = options.el
7      this.$el = typeof options.el === 'string' ? document.querySelector(el)
      : el
8
9      // 2. 负责把 data 注入到 vue 实例
10     this._proxyData(this.$data)
11     // 3. 负责调用 Observer 实现数据劫持
12     // 4. 负责调用 Compiler 解析指令/插值表达式等
13   }
14
15   _proxyData (data) {
16     // 遍历 data 的所有属性
17     Object.keys(data).forEach(key => {
18       Object.defineProperty(this, key, {
19         get () {
20           return data[key]
21         },
22         set (newValue) {
23           if (data[key] === newValue) {
24             return
25           }
26           data[key] = newValue
27         }
28       })
29     })
30   }
31 }

```

Observer

- 功能
 - 负责把 data 选项中的属性转换成响应式数据
 - data 中的某个属性也是对象，把该属性转换成响应式数据
 - 数据变化发送通知
- 结构



- 代码

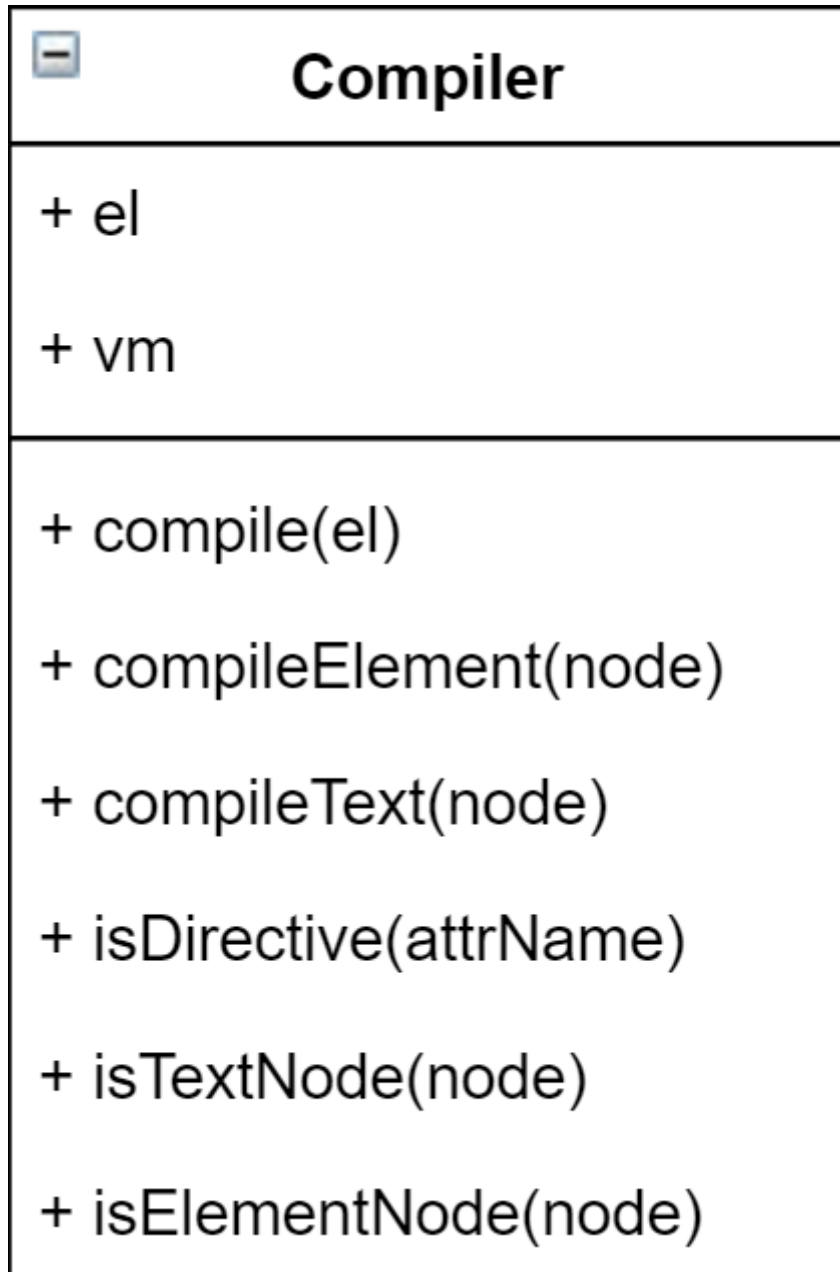
```
1 // 负责数据劫持
2 // 把 $data 中的成员转换成 getter/setter
3 class Observer {
4   constructor (data) {
5     this.walk(data)
6   }
7   // 1. 判断数据是否是对象，如果不是对象返回
8   // 2. 如果是对象，遍历对象的所有属性，设置为 getter/setter
9   walk (data) {
10    if (!data || typeof data !== 'object') {
11      return
12    }
13    // 遍历 data 的所有成员
14    Object.keys(data).forEach(key => {
15      this.defineReactive(data, key, data[key])
16    })
17  }
18  // 定义响应式成员
19  defineReactive (data, key, val) {
20    const that = this
21    // 如果 val 是对象，继续设置它下面的成员为响应式数据
22    this.walk(val)
23    Object.defineProperty(data, key, {
24      configurable: true,
25      enumerable: true,
26      get () {
27        return val
28      },
29      set (newValue) {
30        if (newValue === val) {
31          return
32        }
33        // 如果 newValue 是对象，设置 newValue 的成员为响应式
34        that.walk(newValue)
35        val = newValue
36      }
37    })
38  }
39 }
```



```
36     }  
37   })  
38 }  
39 }
```

Compiler

- 功能
 - 负责编译模板，解析指令/插值表达式
 - 负责页面的首次渲染
 - 当数据变化后重新渲染视图
- 结构



compile()

```
1 // 负责解析指令/插值表达式  
2 class Compiler {  
3   constructor (vm) {  
4     this.vm = vm  
5     this.el = vm.$el
```

```

6      // 编译模板
7      this.compile(this.el)
8  }
9  // 编译模板
10 // 处理文本节点和元素节点
11 compile (el) {
12     const nodes = el.childNodes
13     Array.from(nodes).forEach(node => {
14         // 判断是文本节点还是元素节点
15         if (this.isTextNode(node)) {
16             this.compileText(node)
17         } else if (this.isElementNode(node)) {
18             this.compileElement(node)
19         }
20
21         if (node.childNodes && node.childNodes.length) {
22             // 如果当前节点中还有子节点，递归编译
23             this.compile(node)
24         }
25     })
26 }
27 // 判断是否是文本节点
28 isTextNode (node) {
29     return node.nodeType === 3
30 }
31 // 判断是否是属性节点
32 isElementNode (node) {
33     return node.nodeType === 1
34 }
35 // 判断是否是以 v- 开头的指令
36 isDirective (attrName) {
37     return attrName.startsWith('v-')
38 }
39
40 // 编译文本节点
41 compileText (node) {
42 }
43
44 // 编译属性节点
45 compileElement (node) {
46 }
47 }

```

compileText()

- 负责编译插值表达式

```

1 // 编译文本节点
2 compileText (node) {
3   const reg = /\{\{(.+)\}\}/
4   // 获取文本节点的内容
5   const value = node.textContent
6   if (reg.test(value)) {
7     // 插值表达式中的值就是我们要的属性名称
8     const key = RegExp.$1.trim()
9     // 把插值表达式替换成具体的值
10    node.textContent = value.replace(reg, this.vm[key])
11  }
12 }

```

compileElement()

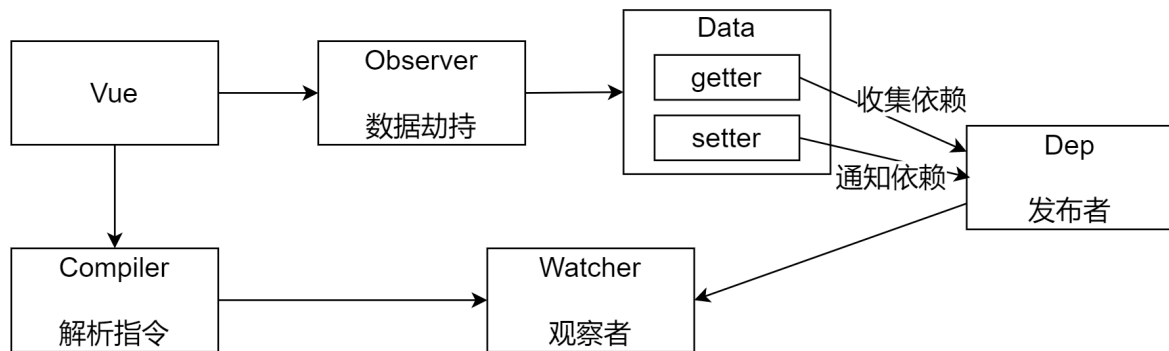
- 负责编译元素的指令
- 处理 v-text 的首次渲染
- 处理 v-model 的首次渲染

```

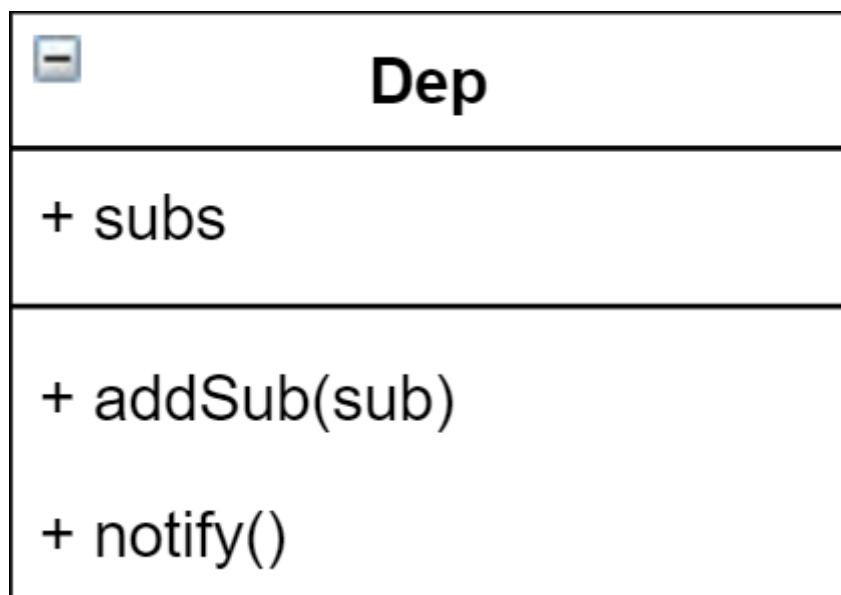
1 // 编译属性节点
2 compileElement (node) {
3   // 遍历元素节点中的所有属性，找到指令
4   Array.from(node.attributes).forEach(attr => {
5     // 获取元素属性的名称
6     let attrName = attr.name
7
8     // 判断当前的属性名称是否是指令
9     if (this.isDirective(attrName)) {
10      // attrName 的形式 v-text v-model
11      // 截取属性的名称，获取 text model
12      attrName = attrName.substr(2)
13      // 获取属性的名称，属性的名称就是我们数据对象的属性 v-text="name", 获取的是
name
14      const key = attr.value
15      // 处理不同的指令
16      this.update(node, key, attrName)
17    }
18  })
19 }
20 // 负责更新 DOM
21 // 创建 watcher
22 update (node, key, dir) {
23   // node 节点, key 数据的属性名称, dir 指令的后半部分
24   const updaterFn = this[dir + 'Updater']
25   updaterFn && updaterFn(node, this.vm[key])
26 }
27
28 // v-text 指令的更新方法
29 textUpdater (node, value) {
30   node.textContent = value
31 }
32 // v-model 指令的更新方法
33 modelUpdater (node, value) {
34   node.value = value
35 }

```

Dep(Dependency)



- 功能
 - 收集依赖，添加观察者(watcher)
 - 通知所有观察者
- 结构



- 代码

```
1 class Dep {
2   constructor () {
3     // 存储所有的观察者
4     this.subs = []
5   }
6   // 添加观察者
7   addSub (sub) {
8     if (sub && sub.update) {
9       this.subs.push(sub)
10    }
11  }
12  // 通知所有观察者
13  notify () {
14    this.subs.forEach(sub => {
15      sub.update()
16    })
17  }
18 }
```

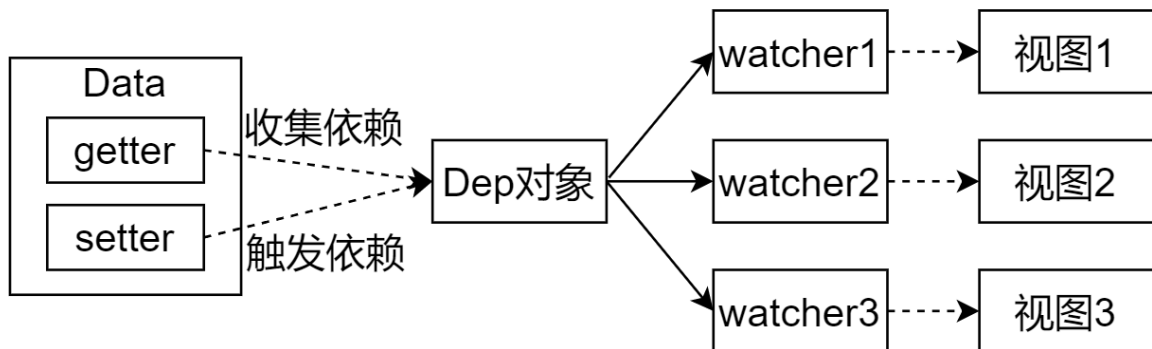
- 在 compiler.js 中收集依赖，发送通知

```

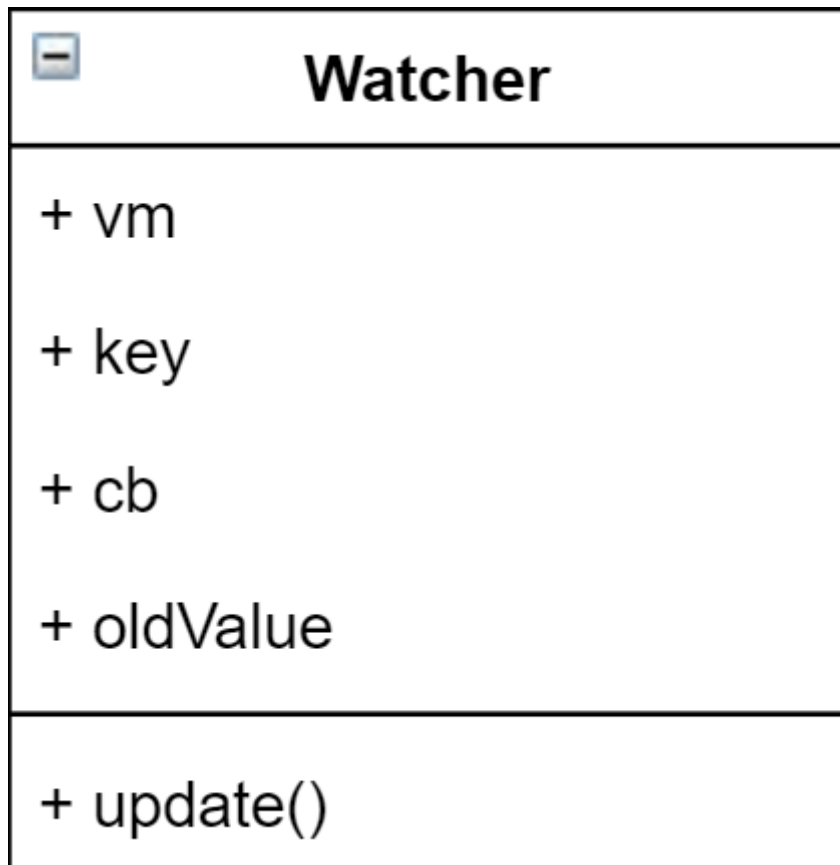
1 // defineReactive 中
2 // 创建 dep 对象收集依赖
3 const dep = new Dep()
4
5 // getter 中
6 // get 的过程中收集依赖
7 dep.target && dep.addSub(Dep.target)
8
9 // setter 中
10 // 当数据变化之后，发送通知
11 dep.notify()

```

Watcher



- 功能
 - 当数据变化触发依赖，dep 通知所有的 Watcher 实例更新视图
 - 自身实例化的时候往 dep 对象中添加自己
- 结构



- 代码

```

1  class Watcher {
2    constructor (vm, key, cb) {
3      this.vm = vm
4      // data 中的属性名称
5      this.key = key
6      // 当数据变化的时候, 调用 cb 更新视图
7      this.cb = cb
8      // 在 Dep 的静态属性上记录当前 watcher 对象, 当访问数据的时候把 watcher 添加到
      dep 的 subs 中
9      Dep.target = this
10     // 触发一次 getter, 让 dep 为当前 key 记录 watcher
11     this.oldValue = vm[key]
12     // 清空 target
13     Dep.target = null
14   }
15
16   update () {
17     const newValue = this.vm[this.key]
18     if (this.oldValue !== newValue) {
19       return
20     }
21     this.cb(newValue)
22   }
23 }

```

- 在 compiler.js 中为每一个指令/插值表达式创建 watcher 对象, 监视数据的变化

```

1  // 因为在 textUpdater 等中要使用 this
2  updaterFn && updaterFn.call(this, node, this.vm[key], key)
3
4  // v-text 指令的更新方法
5  textUpdater (node, value, key) {
6    node.textContent = value
7    // 每一个指令中创建一个 watcher, 观察数据的变化
8    new Watcher(this.vm, key, value => {
9      node.textContent = value
10    })
11  }

```

视图变化更新数据

```

1  // v-model 指令的更新方法
2  modelUpdater (node, value, key) {
3    node.value = value
4    // 每一个指令中创建一个 watcher, 观察数据的变化
5    new Watcher(this.vm, key, value => {
6      node.value = value
7    })
8    // 监听视图的变化
9    node.addEventListener('input', () => {
10      this.vm[key] = node.value
11    })
12  }

```

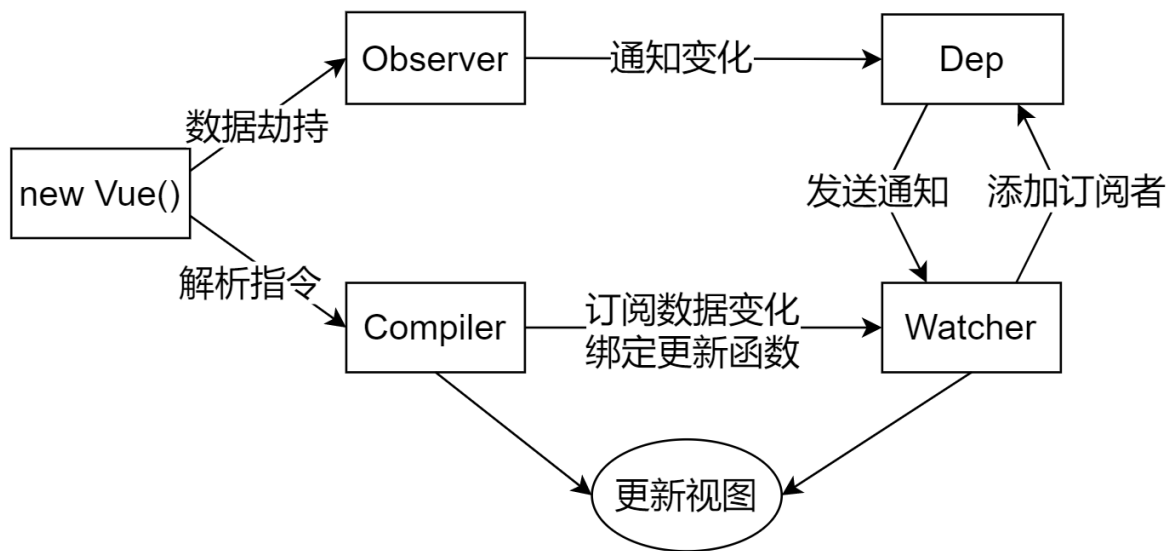
调试

通过调试加深对代码的理解

1. 调试页面首次渲染的过程
2. 调试数据改变更新视图的过程

总结

- 问题
 - 给属性重新赋值成对象，是否是响应式的？
 - 给 Vue 实例新增一个成员是否是响应式的？
- 通过下图回顾整体流程



- Vue
 - 记录传入的选项，设置 `$data/$el`
 - 把 `data` 的成员注入到 Vue 实例
 - 负责调用 `Observer` 实现数据响应式处理（数据劫持）
 - 负责调用 `Compiler` 编译指令/插值表达式等
- Observer
 - 数据劫持
 - 负责把 `data` 中的成员转换成 `getter/setter`
 - 负责把多层属性转换成 `getter/setter`
 - 如果给属性赋值为新对象，把新对象的成员设置为 `getter/setter`
 - 添加 `Dep` 和 `Watcher` 的依赖关系
 - 数据变化发送通知
- Compiler
 - 负责编译模板，解析指令/插值表达式
 - 负责页面的首次渲染过程
 - 当数据变化后重新渲染
- Dep
 - 收集依赖，添加订阅者(`watcher`)
 - 通知所有订阅者
- Watcher
 - 自身实例化的时候往 `dep` 对象中添加自己
 - 当数据变化 `dep` 通知所有的 `Watcher` 实例更新视图

参考

- [深入响应式原理](#)
- <https://github.com/DMQ/mvvm>