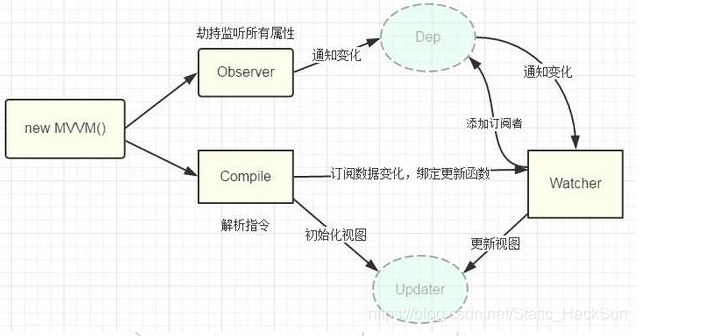
# **Vue的MVVM原理及其实现**

## 一. 什么是mvvm

MVVM是Model-View-ViewModel的简写。它本质上就是MVC 的改进版。 MVVM分为三个部分：分别是M（Model，模型层 ），V（View，视图层），VM（ViewModel，V与M连接的桥梁，也可以看作为控制器） 1、 M：模型层，主要负责业务数据相关； 2、 V：视图层，顾名思义，负责视图相关，细分下来就是html+css层； 3、 VM：V与M沟通的桥梁，负责监听M或者V的修改，是实现MVVM双向绑定的要点

## 二.为什么使用mvvm

MVVM 就是将其中的View 的状态和行为抽象化，让我们将视图 UI 和业务逻辑分开。 MVVM支持双向绑定，意思就是当M层数据进行修改时，VM层会监测到变化，并且通知V层进行相应的修改，反之修改V层则会通知M层数据进行修改，以此也实现了视图与模型层的相互解耦；



MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果。 1.实现compile,进行模板的编译，包括编译元素（指令）、编译文本等，达到初始化视图的目的，并且还需要绑定好更新函数； 2.实现Observe,监听所有的数据，并对变化数据发布通知； 3.实现watcher,作为一个中枢，接收到observe发来的通知，并执行compile中相应的更新方法。 4.结合上述方法，向外暴露mvvm方法。

## **MVVM原理及其实现**

<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <title>MVVM原理及其实现</title>  
</head>  
<body>  
<div id="app">  
  <input type="text" v-model="message">  
  <div>{{message}}</div>  
  <ul><li></li></ul>  
</div>  
<script src="watcher.js"></script>  
<script src="observe.js"></script>  
<script src="compile.js"></script>  
<script src="mvvm.js"></script>  
<script>  
  let vm = new MVVM({  
    el: '#app',  
    data: {  
      message: 'hello world',  
      a: {  
        b: 'bbb'  
     }  
   }  
 })  
</script>  
</body>  
</html>  
​

## 1.实现一个mvvm类（入口）

新建一个mvvm.js,将参数通过options传入mvvm中，并取出el和data绑定到mvvm的私有变量和data中。

// mvvm.js  
class MVVM {  
  constructor(options) {  
    this.$el = options.el  
    this.$data = options.data  
 }  
}  
​  
​  
​

## 2.实现compile（编译模板）

新建一个compile.js文件，在mvvm.js中调用compile。compile.js接收mvvm中传过来的el和vm实例。

// mvvm.js  
class MVVM {  
  constructor(options) {  
    this.$el = options.el  
    this.$data = options.data  
    // 如果有要编译的模板 =>编译  
    if(this.$el) {  
      // 将文本+元素模板进行编译  
      new Compile(this.$el, this)  
   }  
 }  
}  
​

### **（1）初始化传值**

// compile.js  
export default class Compile {  
  constructor(el, vm) {  
    // 判断是否是元素节点，是=》取该元素 否=》取文本  
    this.el = this.isElementNode(el) ? el:document.querySelector(el)  
    this.vm = vm  
 },  
  // 判断是否是元素节点  
  isElementNode(node) {  
    return node.nodeType === 1  
 }  
}  
​

### **（2）先把真实DOM移入到内存中 fragment，因为fragment在内存中，操作比较快**

// compile.js  
class Compile {  
  constructor(el, vm) {  
    // 判断是否是元素节点，是=》取该元素 否=》取文本  
    this.el = this.isElementNode(el) ? el:document.querySelector(el)  
    this.vm = vm  
    // 如果这个元素能获取到 我们才开始编译  
    if(this.el) {  
      // 1. 先把真实DOM移入到内存中 fragment  
      let fragment = this.node2fragment(this.el)  
   }  
 },  
  // 判断是否是元素节点  
  isElementNode(node) {  
    return node.nodeType === 1  
 }  
  // 将el中的内容全部放到内存中  
  node2fragment(el) {   
    let fragment = document.createDocumentFragment()  
    let firstChild  
    // 遍历取出firstChild，直到firstChild为空  
    while (firstChild = el.firstChild) {  
      fragment.appendChild(firstChild)  
   }  
    return fragment // 内存中的节点  
 }  
}  
​

### **（3）编译 =》 在fragment中提取想要的元素节点 v-model 和文本节点**

// compile.js  
class Compile {  
  constructor(el, vm) {  
    // 判断是否是元素节点，是=》取该元素 否=》取文本  
    this.el = this.isElementNode(el) ? el:document.querySelector(el)  
    this.vm = vm  
    // 如果这个元素能获取到 我们才开始编译  
    if(this.el) {  
      // 1. 先把真实DOM移入到内存中 fragment  
      let fragment = this.node2fragment(this.el)  
      // 2. 编译 =》 在fragment中提取想要的元素节点 v-model 和文本节点  
      this.compile(fragment)  
      // 3. 把编译好的fragment在放回到页面中  
      this.el.appendChild(fragment)  
   }  
 }  
  // 判断是否是元素节点  
  isElementNode(node) {  
    return node.nodeType === 1  
 }  
  // 是不是指令  
  isDirective(name) {  
    return name.includes('v-')  
 }  
  // 将el中的内容全部放到内存中  
  node2fragment(el) {  
    let fragment = document.createDocumentFragment()  
    let firstChild  
    // 遍历取出firstChild，直到firstChild为空  
    while (firstChild = el.firstChild) {  
      fragment.appendChild(firstChild)  
   }  
    return fragment // 内存中的节点  
 }  
  //编译 =》 提取想要的元素节点 v-model 和文本节点  
  compile(fragment) {  
    // 需要递归  
    let childNodes = fragment.childNodes  
    Array.from(childNodes).forEach(node => {  
      // 是元素节点 直接调用文本编译方法 还需要深入递归检查  
      if(this.isElementNode(node)) {  
        this.compileElement(node)  
        // 递归深入查找子节点  
        this.compile(node)  
      // 是文本节点 直接调用文本编译方法  
     } else {  
        this.compileText(node)  
     }  
   })  
 }  
  // 编译元素方法  
  compileElement(node) {  
    let attrs = node.attributes  
    Array.from(attrs).forEach(attr => {  
      let attrName = attr.name  
      // 判断属性名是否包含 v-指令  
      if(this.isDirective(attrName)) {  
        // 取到v-指令属性中的值（这个就是对应data中的key）  
        let expr = attr.value  
        // 获取指令类型  
        let [,type] = attrName.split('-')  
        // node vm.$data expr  
        compileUtil[type](node, this.vm, expr)  
     }  
   })  
 }  
  // 这里需要编译文本  
  compileText(node) {  
    //取文本节点中的文本  
    let expr = node.textContent  
    let reg = /\{\{([^}]+)\}\}/g  
    if(reg.test(expr)) {  
      // node this.vm.$data text  
      compileUtil['text'](node, this.vm, expr)  
   }  
 }  
}  
// 解析不同指令或者文本编译集合  
const compileUtil = {  
  text(node, vm, expr) { // 文本  
    let updater = this.updater['textUpdate']  
    updater && updater(node, getTextValue(vm, expr))  
 },  
  model(node, vm, expr){ // 输入框  
    let updater = this.updater['modelUpdate']  
    updater && updater(node, getValue(vm, expr))  
 },  
  // 更新函数  
  updater: {  
    // 文本赋值  
    textUpdate(node, value) {  
      node.textContent = value  
   },  
    // 输入框value赋值  
    modelUpdate(node, value) {  
      node.value = value  
   }  
 }  
}  
// 辅助工具函数  
// 绑定key上对应的值，从vm.$data中取到  
const getValue = (vm, expr) => {  
  expr = expr.split('.') // [message, a, b, c]  
  return expr.reduce((prev, next) => {  
    return prev[next]  
 }, vm.$data)  
}  
// 获取文本编译后的对应的数据  
const getTextValue = (vm, expr) => {  
  return expr.replace(/\{\{([^}]+)\}\}/g, (...arguments) => {  
    return getValue(vm, arguments[1])  
 })  
}  
​

(4) 将编译后的fragment放回到dom中

 let fragment = this.node2fragment(this.el)  
  this.compile(fragment)  
  // 3. 把编译好的fragment在放回到页面中  
  this.el.appendChild(fragment)  
​  
​

## 3.实现observe（数据监听/劫持）

不同于发布者-订阅者模式和脏值检测，vue采用的observe + sub/pub 实现数据的劫持，通过js原生的方法Object.defineProperty()来劫持各个属性的setter，getter，在属性对应数据改变时，发布消息给订阅者，然后触发相应的监听回调。 主要内容：observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter。

// observe.js  
class Observe {  
  constructor(data) {  
    this.observe(data)  
 }  
  // 把data数据原有的属性改成 get 和 set方法的形式  
  observe(data) {  
    if(!data || typeof data!== 'object') {  
      return  
   }  
    console.log(data)  
    // 将数据一一劫持  
    // 先获取到data的key和value  
    Object.keys(data).forEach((key) => {  
      // 数据劫持  
      this.defineReactive(data, key, data[key])  
      this.observe(data[key]) // 深度递归劫持，保证子属性的值也会被劫持  
   })  
 }  
  // 定义响应式  
  defineReactive(obj, key, value) {  
    let \_this = this  
    Object.defineProperty(obj, key, {  
      enumerable: true,  
      configurable: true,  
      get() { // 当取值时调用  
        return value  
     },  
      set(newValue) { //当data属性中设置新值得时候 更改获取的新值  
        if(newValue !== value) {  
          \_this.observe(newValue) // 如果是对象继续劫持  
          console.log('监听到值变化了,旧值：', value, ' --> 新值：', newValue);  
          value = newValue  
       }  
     }  
   })  
 }  
}  
​

完成observe.js后，修改mvvm.js文件,将属性传入observe中

// mvvm.js  
class MVVM {  
  constructor(options) {  
    console.log(options)  
    this.$el = options.el  
    this.$data = options.data  
    // 如果有要编译的模板 =》编译  
    if(this.$el) {  
      // 数据劫持 就是把对象的所有属性改成 get 和 set方法  
      new Observe(this.$data)  
      // 将文本+元素模板进行编译  
      new Compile(this.$el, this)  
   }  
 }  
}  
​

可以在控制台查看到以下信息，说明劫持属性成功。



实现数据劫持后，接下来的任务怎么通知订阅者了，我们需要在监听数据时实现一个消息订阅器，具体的方法是：定义一个数组，用来存放订阅者，数据变动通知（notify）订阅者，再调用订阅者的update方法。 在observe.js添加Dep类：

//observe.js  
​  
// ...  
    let \_this = this  
    let dep = new Dep()  
    Object.defineProperty(obj, key, {  
      enumerable: true,  
      configurable: true,  
      get() { // 当取值时调用  
        return value  
     },  
      set(newValue) { //当data属性中设置新值得时候 更改获取的新值  
        if(newValue !== value) {  
          \_this.observe(newValue) // 如果是对象继续劫持  
          console.log('监听到值变化了,旧值：', value, ' --> 新值：', newValue);  
          value = newValue  
          dep.notify() //通知所有人 数据更新了  
       }  
     }  
   })  
// ...  
// 消息订阅器Dep()  
class Dep {  
  constructor() {  
    // 订阅的数组  
    this.subs = []  
 }  
  addSub(watcher) {  
    // push到订阅数组  
    this.subs.push(watcher)  
 }  
  notify() {  
    // 通知订阅者，并执行订阅者的update回调  
    this.subs.forEach(watcher => watcher.update())  
 }  
}  
​  
​

实现了消息订阅器，并且能够执行订阅者的回调，那么订阅者怎么获取，并push到订阅器数组中呢？这个要和watcher结合。

## **4.实现watcher（订阅中心）**

Observer和Compile之间通信的桥梁是Watcher订阅中心，其主要职责是: 1、在自身实例化时往属性订阅器(Dep)里面添加自己，与Observer建立连接； 2、自身必须有一个update()方法，与Compile建立连接； 3、当属性变化时，Observer中dep.notice()通知，然后能调用自身（Watcher）的update()方法，并触发Compile中绑定的回调，实现更新。

// watcher.js  
// 订阅中心（观察者）： 给需要变化的那个元素 增加一个观察者， 当数据变化后，执行对应的方法  
class Watcher {  
  constructor(vm, expr, cb) {  
    this.vm = vm  
    this.expr = expr  
    this.cb = cb  
    // 先获取一下老值  
    this.value = this.get()  
 }  
  getValue(vm, expr) { // 获取实例上对应的数据  
    expr = expr.split('.') // [message, a, b, c]  
    return expr.reduce((prev, next) => {  
      return prev[next]  
   }, vm.$data)  
 }  
  get() { // 获取文本编译后的对应的数据  
    // 获取当前订阅者  
    Dep.target = this  
    // 触发getter，当前订阅者添加订阅器中 在 劫持数据时，将订阅者放到订阅者数组  
    let value = this.getValue(this.vm, this.expr)  
    // 重置订阅者  
    Dep.target = null  
    return value  
 }  
  // 对外暴露的方法  
  update() {  
    let newValue = this.getValue(this.vm, this.expr)  
    let oldValue = this.value  
    // 更新的值 与 以前的值 进行比对， 如果发生变化就更新方法  
    if(newValue !== oldValue) {  
      this.cb(newValue)  
   }  
 }  
}  
​  
// observe.js   
// ... 省略  
Object.defineProperty(data, key, {  
    get: function() {  
        // 在取值时将订阅者push入订阅者数组  
        Dep.target && dep.addDep(Dep.target);  
        return val;  
   }  
    // ... 省略  
});  
// ... 省略  
​

上面步骤搭建了watcher与observe之间的连接，还需要搭建watcher与之间的连接。 我们需要在compile中解析不同指令或者文本编译集合的时候绑定watcher.

// compile.js  
// ...省略  
 model(node, vm, expr){ // 输入框  
    let updater = this.updater['modelUpdate']  
    // 这里加一个监控 数据变化了 应该调用这个watcher的callback  
    new Watcher(vm, expr, (newValue) => {  
      // 当值变化后 会调用cb ，将新值传递过来  
      updater && updater(node, this.getValue(vm, expr))  
   })  
    node.addEventListener('input', (e) => {  
      let newValue = e.target.value  
      this.setVal(vm, expr, newValue)  
   })  
    updater && updater(node, this.getValue(vm, expr))  
 },  
// ...省略  
​

此时，在浏览器控制台执行下图操作，手动改变 message 属性的值，发现输入框的值也随之变化，v-model 绑定完成。

