一、观察者模式、发布订阅模式

·观察者模式概念：

当对象间存在一对多的关系时，使用观察者模式，来达到松耦合。当一个对象(被观察者)被修改时，则会自动通知依赖(观察)它的对象(观察者)。

·手动实现观察者模式

在被观察者中收集所有的观察者。

在被观察者中通知观察者其属性值被修改了，在被观察者中可以获取到最新被修改的值。

一个属性被修改了，可以实现多个操作。在观察者中要设置一个用于收集操作的值，并且设置一个通信方式update()方法，用来执行这些操作。

// 被观察者

class Observered {

    constructor(name, age, state) {

        this.name = name

        this.age = age

        this.state = state

        this.observers = []

    }

    addOb(observer) {

        this.observers.push(observer)

    }

    setState(newVal) {

        this.state = newVal

        this.observers.forEach((ob) => {

            ob.update(this.name, this.age, newVal)

        })

    }

}

// 观察者

class Observer {

    constructor() {

        this.callback = []

    }

    add(fn) {

        this.callback.push(fn)

    }

    update(...args) {

        this.callback.forEach((fn) => {

            fn(...args)

        })

    }

}

let ob1 = new Observered('小明', 14, '吃饭')

let fn1 = (name, age, state) => {

    console.log(`${name}今年${age}岁，正在${state}`)

}

let fn2 = (name, age, state) => {

    console.log(`你好，我是${name}，我今年${age}，正在${state}`)

}

let watcher = new Observer()

watcher.add(fn1)

watcher.add(fn2)

ob1.addOb(watcher)

ob1.setState('学习')

//小明今年14岁，正在学习

//你好，我是小明，我今年14，正在学习

·发布订阅模式概念：

在发布订阅模式中，发布者并不会直接通知订阅者，会通过消息调度中心来通知订阅者。来实现发布者和订阅者的完全解耦合。

发布者只管向调度中心通知，并不知道会怎么处理数据，有多少人响应。

而订阅者只管在调度中心订阅，由调度中心来调用

·手动实现发布订阅模式

实现一个消息调度中心类EventEmitter

收集多个订阅事件，及其不同的回调函数

包含供订阅者添加订阅on、取消订阅off的方法，供发布者发布事件emit的方法

class EventEmitter {

    constructor() {

        //同名事件多个回调函数

        this.events = {}

    }

    // 订阅事件

    on(event, fn) {

        if (!this.events[event]) {

            this.events[event] = [fn]

        } else {

            this.events[event].push(fn)

        }

    }

    // 取消订阅事件

    off(event,fn){

        if(this.events[event]){

            let index = this.events[event].indexOf(fn)

            this.events[event].splice(index,1)

        }

    }

    // 发布事件

    emit(event,...args){

        if(this.events[event]){

            this.events[event].forEach((fn)=>{

                fn(...args)

            })

        }

    }

}

let eventBus = new EventEmitter()

let fn1 = function(name, age) {

    console.log(`${name} ${age}`)

}

let fn2 = function(name, age) {

    console.log(`hello, ${name} ${age}`)

}

eventBus.on('changeName', fn1)

eventBus.on('changeName', fn2)

eventBus.emit('changeName', '布兰', 12)

//布兰 12

//hello, 布兰 12

eventBus.off('changeName',fn1)

eventBus.emit('changeName', 'who am i', 21)

//hello, who am i 21

二、对MVVM的理解

MVVM框架包含三层view视图层、viewmodel视图模型层、model模型层。

在viewmodel中实现view和model的双向数据绑定，将view和model完全解耦。

通过view层可以修改model层的数据，当model层的数据修改后，也可以直接修改view层的显示效果。

实现双向数据绑定的方法：

数据劫持

脏值检查

发布-订阅模式

在vue中采用数据劫持结合发布订阅模式的方式实现双向数据绑定。

View层是html中的DOM元素

ViewModel是vue实例

Model层是实例化vue时的data配置项。

三、Array.prototype.reduce()的用法

使用：下一次操作的初始值，依赖于上一次操作的返回值。

功能：数组中每个元素按升序执行一个reducer函数。每次运行reducer会将先前元素的计算结果作为参数传入，最后将结果汇总为单个返回值。

语法：arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

返回值：使用reducer回调函数遍历整个数组后的结果

·数组中元素累加

let arr = [1,2,3,4]

let ans = arr.reduce((a,b)=>a+b)

console.log(ans);   //10

·链式获取嵌套对象中的元素值

let obj = {

    name:'zs',

    info:{

        address:{

            location:'南京'

        }

    }

}

let attrs = ['info','address','location']

let res = attrs.reduce((a,b)=>a[b],obj)

console.log(res);   // 南京

四、如何实现数据劫持及数据代理？

使用Object.defineProperty()来完成。

数据劫持：在修改和读取data配置项的属性时，设置setter和getter都可以拦截这个操作。

let obj = {

    name: 'wan',

    age: 12,

    gender: 'male'

}

Object.keys(obj).forEach((key) => {

    let value = obj[key]

    Object.defineProperty(obj, key, {

        set(newVal) {

            console.log(`设置了${key}属性为${newVal}`);

            value = newVal

        },

        get() {

            console.log(`读取了${key}属性`);

            return value

        }

    })

})

console.log(obj.gender)

obj.gender = 'female'

数据代理：在vue中，vue实例vm上的$data属性中包含了data配置项的所有属性和属性值，但在插值语法中使用必须包含$data的前缀，为了方便使用，将data配置项的所有属性和属性值直接挂载到vm上，但是当访问和修改这些属性时，实际修改访问的还是vm.$data上的属性。

let data = {

    name: 'wan',

    age: '14',

    info: {

        address: 'nj',

        school: 'cc'

    }

}

let vm = {}

function proxyKeys(dest, source) {

    Object.keys(source).forEach((key) => {

        Object.defineProperty(dest, key, {

            enumerable: true,

            configurable: true,

            set(newVal) {

                source[key] = newVal

            },

            get() {

                return source[key]

            }

        })

    })

}

proxyKeys(vm,data)

console.log(vm);

五、Vue的双向数据绑定原理

·仅使用Object.defineProperty()完成简易版的双向数据绑定

        const app = document.getElementById('app')

        const box = document.getElementById('box')

        const btn = document.getElementById('btn')

        const obj = {}

        Object.defineProperty(obj, 'value', {

            enumerable: true,

            configurable: true,

            set(newVal) {

                app.value = newVal

                box.innerHTML = newVal

            }

        })

        app.addEventListener('input', (e) => {

            obj.value = e.target.value

        })

        btn.addEventListener('click', () => {

            setTimeout(() => {

                obj.value = 'chaos'

            }, 3000)

        })

·Object.defineProperty()结合发布订阅模式完成双向数据绑定

1·先构造index.html，写好构造Vue实例的函数，添加配置项el和data。

2·创建一个Vue类，在初始化时，要传入options配置项，将其data挂载到vm.$data上。

3·使用Observer()函数，将vm.$data上的所有属性都监听

递归遍历vm.$data上的属性。对嵌套对象上的属性也添加setter和getter

4·实现数据代理，将vm.$data上的所有属性添加到vm本身上。方便使用。

        // 做数据代理

        this.proxykeys(this.$data, this)

    proxykeys(source, dest) {

        Object.keys(source).forEach(key => {

            Object.defineProperty(dest, key, {

                enumerable: true,

                configurable: true,

                set(newVal) {

                    source[key] = newVal

                },

                get() {

                    return source[key]

                }

            })

        })

    }

5·实现用于模板编译的Compile函数，要介绍两个参数，一个指向真实DOM，一个接收vm对象获取最新的数据。

判断文本节点，以及input元素节点。

// 对HTML结构进行模板编译

function compile(el, vm) {

    // 取得真实的DOM对象，并且挂载到vue实例上

    vm.$el = document.querySelector(el)

    //通过文档碎片DocumentFragment来操作DOM，提高DOM操作的性能

    // 开辟一块内存区域，dom元素移入到文档片段时，就不可以存在于DOM树中

    const fragment = document.createDocumentFragment()

    while (child = vm.$el.firstChild) {

        fragment.appendChild(child)

    }

    // 进行模板编译

    compileText(fragment)

    // 将fragment中所有子节点添加到DOM树中，只渲染一次，提升性能

    vm.$el.appendChild(fragment)

    // 负责对DOM模板进行编译的方法

    function compileText(node) {

        // 进行插值语法的匹配正则

        const reg = /\{\{\s\*(\S+)\s\*\}\}/

        // 证明当前node节点是文本节点

        if (node.nodeType === 3) {

            let text = node.nodeValue

            if (reg.exec(text) !== null) {

                let arr = reg.exec(text);

                let str = arr[1].split('.').reduce((obj, key) => obj[key], vm);

                // 更新DOM中的元素

                node.nodeValue = text.replace(reg, str)

                // 创建一个watcher实例

                new Watcher(vm, arr[1], (newVal) => {

                    node.nodeValue = text.replace(reg, newVal)

                })

            }

            return

        }

        // 判断有没有v-model属性

        if (node.nodeType === 1 && node.nodeName.toUpperCase() === 'INPUT') {

            let propVmodel = Array.from(node.attributes).find(key => key.name == 'v-model')

            if (propVmodel) {

                let expStr = propVmodel.value

                // 获取value中绑定属性的值

                let value = expStr.split('.').reduce((obj, key) => obj[key], vm)

                // 给input输入框的value赋值

                node.value = value

                // 创建一个订阅者watcher，再次修改其中值时可以获得消息

                new Watcher(vm, expStr, (newValue) => {

                    node.value =  newValue

                })

在操作DOM时，要创建一个文档碎片DocumentFragment来进行操作，这样可以提升性能。防止重绘重排。

6·构建收集依赖的Dep类。

包含两个方法，一个用于添加订阅者watcher的方法add()，一个是用于通知订阅者数据发生改变的notify()方法。

// 依赖收集的类

class Dep {

    constructor() {

        this.watchers = []

    }

    // 追加订阅者

    add(watcher) {

        this.watchers.push(watcher)

    }

    // 通知订阅者数据发生了变化

    notify() {

        this.watchers.forEach(watcher => watcher.update())

    }

}

7·构建Watcher类。

在订阅时要获取最新数据，传入vm；要知道对哪个具体的属性进行订阅的，传入key；还要由修改DOM的具体方法，传入回调函数cb。

// 订阅者

class Watcher {

    constructor(vm, key, cb) {

        // 传入回调函数，在回调中用来修改DOM上的数据

        this.cb = cb

        // vm中的$data保存着最新的数据，用来拿到最新的数据

        this.vm = vm

        // 在vm上众多数据中，订阅了哪个数据

        this.key = key

        // 在Dep中添加watcher

        // Watcher是一个构造函数，this指向新创建的watcher实例

        Dep.target = this

        this.key.split('.').reduce((obj, propName) => obj[propName], vm)

        Dep.target = null

    }

    update() {

        // 获取到新值

        const val = this.key.split('.').reduce((obj, propName) => obj[propName], this.vm)

        // 利用回调函数cb来修改DOM

        this.cb(val)

    }

}

此时考虑一个问题：在什么时候要创建Watcher类？

在知道如何修改DOM时，就可以获取这个回调函数cb，以及对应的属性，此时应该创建一个Watcher实例，对该属性进行订阅。

如何将Watcher加入到Dep中，使其收集依赖？

在new一个Watcher实例时，就应该将Watcher添加到Dep中去。

        Dep.target = this

        this.key.split('.').reduce((obj, propName) => obj[propName], vm)

        Dep.target = null

第一行是使用Dep.target来获取Watcher实例this，第二行代码则是触发对应属性的getter，在getter中，调用Dep实例的add方法，将Watcher实例加入到Dep中去。

            Dep.target && dep.add(Dep.target)

最后再初始化Dep.target，方便下次添加Watcher实例。

8·此时已经实现了单向数据绑定，通过修改data中的属性，可以实现修改view层的显示。对input元素绑定对应事件，可以实现双向数据绑定，view==>model。

// 双向数据绑定，监听输入框的input事件

                node.addEventListener('input',(e)=>{

                    let newVal = e.target.value

                    // 获取对应属性的值

                    let propArr = expStr.split('.')

                    let property = propArr.slice(0,-1).reduce((obj,key)=>obj[key],vm)

                    property[propArr[propArr.length - 1]] = newVal

六、使用到的一些API

·console.dir( object ) ：打印对象的所有属性以及属性值，通过类似文件树状的交互列表显示。

·childNodes：

每个节点都有一个childNodes属性，其中包含一个NodeList的实例，是一个类数组，是一个实时的活动对象，虽然是类数组但是其包含forEach()方法，可以遍历其中每个元素。

·Attr类型：

元素可以访问attributes属性，是一个类数组，没有实现数组的方法。使用Array.from()转换为真正的数组。

·DocumentFragment类型

是内存中的一块区域，如果文档中的一个节点添加到文档碎片中，则在原来的文档树中则会消失，不会在被浏览器渲染。

为避免多次渲染，造成重排重绘，使用文档碎片可以一次性将其所有子节点添加到DOM树中。提升了性能。