# 谈一谈你对MVVM的理解。

MVVM就是 model view viewModel 的缩写。Model代表了数据层，只关心数据。view代表了视图层，只关心视图。而viewModel代表的就是view 和 model的桥梁，是业务逻辑。viewModel提供view所需要的数据，响应view中的用户操作带来的变化。利用双向绑定，似的model数据变化时，viewModel自动更新，，而viewModel更新后，view也会自动变化。总结一句话，MVVM简化了界面与业务的依赖，解决了数据频繁更新。

# V-if和v-show的区别。

1. dom层：v-if操作dom添加删除，为false时，不在html中生成dom。v-show只会控制css display，始终存在于html中 。
2. 原理层：v-if 逻辑生成的render函数时 (show)?\_c(xxx):\_e();v-show只会处理style

# 双向绑定的原理

Vue.js是数据劫持+发布订阅者模式实现双向绑定。通过Object.defineProperty()来劫持各个属性的getter和setter，在数据发生变化时，发布消息给订阅者，触发响应的监听回调。

具体实现分为Object.defineProperty，complie，watcher

1. vue遍历data属性，给属性加上getter，setter，这样在改变属性时就可以触发setter，实现监听数据变化。
2. Complie解析模板指令，将模板中的变量替换为数据，然后初始化渲染页面视图，并为每个对应的节点绑定对应的更新函数，添加监听数据的订阅者，一旦setter触发，则调用对应的更新函数，更新视图。
3. Watcher是Observer和complie的桥梁，主要功能：1）把自己添加到属性订阅器（dep）中添加自己。2）包含一个update方法。3）当属性发生变化时dep.notice()通知时，调用自生的update方法，触发complie绑定的回调，实现视图更新。

总结一句话：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

# vue.use(xx)

做了什么?

挂载插件，调用插件中install方法，比如this.$store 为什么能在所有组件的this中访问到，就是在install中做的。

const install = (\_Vue)=>{

Vue = \_Vue;

Vue.mixin({  
 beforeCreate(){

if(this.options && this.options.state){  
 this.$store = this.options.store  
 }else{

this.$store = this.parent && this.parent.$store;

}  
 })

}

# vuex原理

## 关键属性 state getters mutations actions

1. State

实现监听 通过 vm = new Vue({  
 data:{state:options.state}  
})  
get state(){  
 return vm.state

}

1. getters

Options.getters.forEach(getterName =>{  
 Object.defineProperty(this.getters,getterName,{

get(){

return getters[getterName](this.state)

}

})  
})

1. mutations

mutations:{

synAdd(state,dt){

state.num += dt;

}

}  
  
options.mutations.forEach(mutationName=>{   
 this.mutations[mutationName] = (dt)=>{ //实际上这里是数组  
 options.mutations[mutationName](this.state,dt)  
 }  
})

this.commit(“synAdd”,dt)

Commit(type,dt){

this.mutations[type](dt)

}

1. actions

actions:{

asynAdd({commit,dispatch},dt){

setTimout(()=>{commit(“synAdd”,dt)})

}

}

options.actions.forEach(actionName=>{   
 this.actions[actionName] = (dt)=>{ //实际上这里是数组  
 options.actions[actionName](this,dt)  
 }  
})

this.dispatch(“asynAdd”,dt)

dispatch(type,dt){

this.actions[type](dt)

}

# vue-router原理

1. 两种模式

hash history（刷新页面test.html/home =>/home可能会导致404）

* 1. hash: location.hash onhashchange
  2. history history.pushState({},null,pathname) 前进后退监听事件popstate 当前路径location.pathname

1. 监听路由变化

Vue.util.defineReactive(this,”xxx”,当前路由)

1. 两个全局组件

router-link router-view

* 1. Vue.component(‘router-link ’,{  
      props:{  
      to:String

tag:String  
 },

methods:{  
 fn(){  
 hash跳转 || history跳转

}  
 },

render(h)=>{  
 this指向一个proxy对象，this.\_self拿到组件实例

return <tag on-click={this.fn}>{this.} </tag>

})

* 1. Vue.component(‘router-view’,{  
      render(h)=>{  
      return h(路由表[当前页面路由])  
      }  
     })

# diff算法

O(n3) => O(n) 因为dom操作很少跨层级操作 所以vue采用同级比较。

逻辑：

oldVnode Vnode

先比较根节点 如果根节点相同 Vnode.elm= oldVnode

再比较有没有子节点 如果一方有一方没有 只需要删除或者赋值即可，如果两个都有子节点，这就是diff的核心，updateChildren

采用双指针的方式。

1. 头和头比，如果头头相等，那么头头指针双双后移
2. 以上不成立，尾和尾比，如果尾尾相等，那么尾尾指针双双前移
3. 以上不成立，头和尾比，如果头尾相等，则移尾到头，顺次比较
4. 以上不成立，尾和头比，同上
5. 如果都不是，key比较，old(ABCD) new(CAMDB) 第一次比较 new中C的key在old中找的到，那么old=>CABD ，指针后移，CABD ，指针后移，将M插入old CAMBD ,CAMD,CAMDB.