**1、active-class是哪个组件的属性？嵌套路由怎么定义？**  
答：vue-router模块的router-link组件。

**2、怎么定义vue-router的动态路由？怎么获取传过来的动态参数？**  
答：在router目录下的index.js文件中，对path属性加上/:id。  使用router对象的params.id

**3、vue-router有哪几种导航钩子？**  
答：三种，一种是全局导航钩子：router.beforeEach(to,from,next)，作用：跳转前进行判断拦截。第二种：组件内的钩子；第三种：单独路由独享组件

**4、scss是什么？安装使用的步骤是？有哪几大特性？**  
答：预处理css，把css当前函数编写，定义变量,嵌套。 先装css-loader、node-loader、sass-loader等加载器模块，在webpack-base.config.js配置文件中加多一个拓展:extenstion，再加多一个模块：module里面test、loader

**4.1、scss是什么？在vue.cli中的安装使用步骤是？有哪几大特性？**  
答：css的预编译。

使用步骤：

第一步：用npm 下三个loader（sass-loader、css-loader、node-sass）

第二步：在build目录找到webpack.base.config.js，在那个extends属性中加一个拓展.scss

第三步：还是在同一个文件，配置一个module属性

第四步：然后在组件的style标签加上lang属性 ，例如：lang=”scss”

有哪几大特性:

1、可以用变量，例如（$变量名称=值）；

2、可以用混合器，例如（）

3、可以嵌套

**5、mint-ui是什么？怎么使用？说出至少三个组件使用方法？**

答：基于vue的前端组件库。npm安装，然后import样式和js，vue.use（mintUi）全局引入。在单个组件局部引入：import {Toast} from ‘mint-ui’。组件一：Toast(‘登录成功’)；组件二：mint-header；组件三：mint-swiper

**6、v-model是什么？怎么使用？ vue中标签怎么绑定事件？**

答：可以实现双向绑定，指令（v-class、v-for、v-if、v-show、v-on）。vue的model层的data属性。绑定事件：<input @click=doLog() />

**7、axios是什么？怎么使用？描述使用它实现登录功能的流程？**

答：请求后台资源的模块。npm install axios -S装好，然后发送的是跨域，需在配置文件中config/index.js进行设置。后台如果是Tp5则定义一个资源路由。js中使用import进来，然后.get或.post。返回在.then函数中如果成功，失败则是在.catch函数中

**8、axios+tp5进阶中，调用axios.post(‘api/user’)是进行的什么操作？axios.put(‘api/user/8′)呢？**

答：跨域，添加用户操作，更新操作。

**9、什么是RESTful API？怎么使用?**

答：是一个api的标准，无状态请求。请求的路由地址是固定的，如果是tp5则先路由配置中把资源路由配置好。标准有：.post .put .delete

**10、vuex是什么？怎么使用？哪种功能场景使用它？**

答：vue框架中状态管理。在main.js引入store，注入。新建了一个目录store，….. export 。场景有：单页应用中，组件之间的状态。音乐播放、登录状态、加入购物车

**11、mvvm框架是什么？它和其它框架（jquery）的区别是什么？哪些场景适合？**

答：一个model+view+viewModel框架，数据模型model，viewModel连接两个

区别：vue数据驱动，通过数据来显示视图层而不是节点操作。

场景：数据操作比较多的场景，更加便捷

**12、自定义指令（v-check、v-focus）的方法有哪些？它有哪些钩子函数？还有哪些钩子函数参数？**

答：全局定义指令：在vue对象的directive方法里面有两个参数，一个是指令名称，另外一个是函数。组件内定义指令：directives

钩子函数：bind（绑定事件触发）、inserted(节点插入的时候触发)、update（组件内相关更新）

钩子函数参数：el、binding

**13、说出至少4种vue当中的指令和它的用法？**

答：v-if：判断是否隐藏；v-for：数据循环出来；v-bind:class：绑定一个属性；v-model：实现双向绑定

**14、vue-router是什么？它有哪些组件？**

答：vue用来写路由一个插件。router-link、router-view

**15、导航钩子有哪些？它们有哪些参数？**

答：导航钩子有：a/全局钩子和组件内独享的钩子。b/beforeRouteEnter、afterEnter、beforeRouterUpdate、beforeRouteLeave

参数：有to（去的那个路由）、from（离开的路由）、next（一定要用这个函数才能去到下一个路由，如果不用就拦截）最常用就这几种

**16、Vue的双向数据绑定原理是什么？**

答：vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

**具体步骤：**

**第一步：需要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  
这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化

**第二步：compile解析模板指令**，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图

**第三步：Watcher订阅者是Observer和Compile之间通信的桥梁**，主要做的事情是:  
1、在自身实例化时往属性订阅器(dep)里面添加自己  
2、自身必须有一个update()方法  
3、待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

**第四步：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。

ps：16题答案同样适合”**vue data是怎么实现的？”**此面试题**。**

**17、请详细说下你对vue生命周期的理解？**

答：总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后。

**创建前/后：** 在beforeCreated阶段，vue实例的挂载元素$el和数据对象data都为undefined，还未初始化。在created阶段，vue实例的数据对象data有了，$el还没有。

**载入前/后：**在beforeMount阶段，vue实例的$el和data都初始化了，但还是挂载之前为虚拟的dom节点，data.message还未替换。在mounted阶段，vue实例挂载完成，data.message成功渲染。

**更新前/后：**当data变化时，会触发beforeUpdate和updated方法。

**销毁前/后：**在执行destroy方法后，对data的改变不会再触发周期函数，说明此时vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在

**18、请说下封装 vue 组件的过程？**

答：首先，组件可以提升整个项目的开发效率。能够把页面抽象成多个相对独立的模块，解决了我们传统项目开发：**效率低**、**难维护**、**复用性**等问题。

然后，使用Vue.extend方法创建一个组件，然后使用Vue.component方法注册组件。子组件需要数据，可以在props中接受定义。而子组件修改好数据后，想把数据传递给父组件。可以采用emit方法。

**19、你是怎么认识vuex的？**

答：vuex可以理解为一种开发模式或框架。比如PHP有thinkphp，java有spring等。  
通过状态（数据源）集中管理驱动组件的变化（好比spring的IOC容器对bean进行集中管理）。

应用级的状态集中放在store中； 改变状态的方式是提交mutations，这是个同步的事物； 异步逻辑应该封装在action中。

**20、vue-loader是什么？使用它的用途有哪些？**

答：解析.vue文件的一个加载器，跟template/js/style转换成js模块。

用途：js可以写es6、style样式可以scss或less、template可以加jade等

**21、请说出vue.cli项目中src目录每个文件夹和文件的用法？**

答：assets文件夹是放静态资源；components是放组件；router是定义路由相关的配置;view视图；app.vue是一个应用主组件；main.js是入口文件

**22、vue.cli中怎样使用自定义的组件？有遇到过哪些问题吗？**

答：第一步：在components目录新建你的组件文件（smithButton.vue），script一定要export default {

第二步：在需要用的页面（组件）中导入：import smithButton from ‘../components/smithButton.vue’

第三步：注入到vue的子组件的components属性上面,components:{smithButton}

第四步：在template视图view中使用，<smith-button>  </smith-button>  
问题有：smithButton命名，使用的时候则smith-button。

**23、聊聊你对Vue.js的template编译的理解？**

答：简而言之，就是先转化成AST树，再得到的render函数返回VNode（Vue的虚拟DOM节点）

详情步骤：

首先，通过compile编译器把template编译成AST语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），compile是createCompiler的返回值，createCompiler是用以创建编译器的。另外compile还负责合并option。

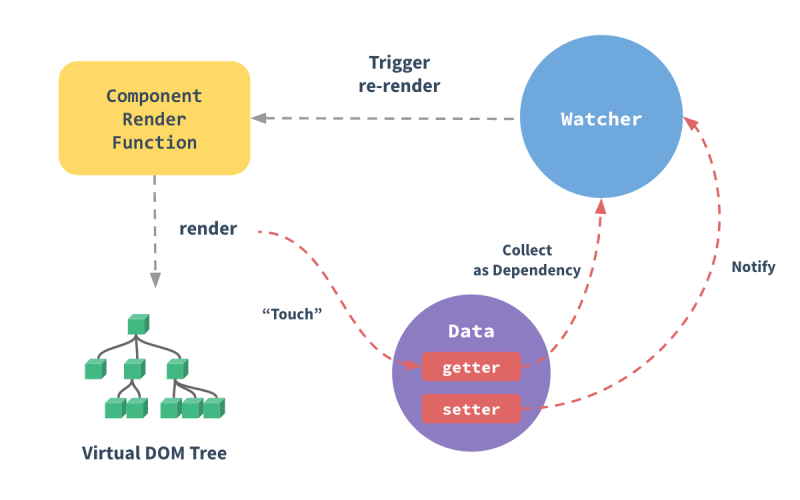
然后，AST会经过generate（将AST语法树转化成render funtion字符串的过程）得到render函数，render的返回值是VNode，VNode是Vue的虚拟DOM节点，里面有（标签名、子节点、文本等等）  
  
**挑战一下：  
1、vue响应式原理？**

1.把一个普通 JavaScript 对象传给 Vue 实例的 data 选项，Vue 将遍历此对象所有的属性，并使用 [Object.defineProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些属性全部转为 **getter/setter**。

2.组件实例的 **watcher** 实例对象，

**初步**

最近一段时间在阅读Vue源码，从它的核心原理入手，开始了源码的学习，而其核心原理就是其数据的响应式，讲到Vue的响应式原理，我们可以从它的兼容性说起，Vue不支持IE8以下版本的浏览器，因为Vue是基于 [Object.defineProperty](https://funteas.com/go/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fen-US%2Fdocs%2FWeb%2FJavaScript%2FReference%2FGlobal_Objects%2FObject%2FdefineProperty) 来实现数据响应的，而 Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器的原因；Vue通过Object.defineProperty的 **getter/setter** 对收集的依赖项进行监听,在属性被访问和修改时通知变化,进而更新视图数据；

受现代JavaScript 的限制 (以及废弃 **Object.observe**)，Vue不能检测到对象属性的添加或删除。由于 Vue 会在初始化实例时对属性执行 **getter/setter** 转化过程，所以属性必须在 **data** 对象上存在才能让Vue转换它，这样才能让它是响应的。 <a id=“more”></a> 

我们这里是根据Vue2.3源码进行分析,Vue数据响应式变化主要涉及 **Observer**, **Watcher** , **Dep** 这三个主要的类；因此要弄清Vue响应式变化需要明白这个三个类之间是如何运作联系的；以及它们的原理，负责的逻辑操作。那么我们从一个简单的Vue实例的代码来分析Vue的响应式原理

var vue = new Vue({

el: "#app",

data: {

name: 'Junga'

},

created () {

this.helloWorld()

},

methods: {

helloWorld: function() {

console.log('my name is' + this.name)

}

}

...

})

**Vue初始化实例**

根据Vue的[生命周期](https://funteas.com/go/?target=https%3A%2F%2Fcn.vuejs.org%2Fv2%2Fguide%2Finstance.html%23%E5%AE%9E%E4%BE%8B%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E9%92%A9%E5%AD%90)我们知道，Vue首先会进行init初始化操作；源码在[src/core/instance/init.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Finstance%2Finit.js)中

/\*初始化生命周期\*/

initLifecycle(vm)

/\*初始化事件\*/

initEvents(vm)Object.defineProperty

/\*初始化render\*/

initRender(vm)

/\*调用beforeCreate钩子函数并且触发beforeCreate钩子事件\*/

callHook(vm, 'beforeCreate')

initInjections(vm) // resolve injections before data/props

/\*初始化props、methods、data、computed与watch\*/

initState(vm)

initProvide(vm) // resolve provide after data/props

/\*调用created钩子函数并且触发created钩子事件\*/

callHook(vm, 'created')

以上代码可以看到 **initState(vm)** 是用来初始化props,methods,data,computed和watch;

[src/core/instance/state.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Finstance%2Fstate.js)

/\*初始化props、methods、data、computed与watch\*/

export function initState (vm: Component) {

vm.\_watchers = []

const opts = vm.$options

/\*初始化props\*/

if (opts.props) initProps(vm, opts.props)

/\*初始化方法\*/

if (opts.methods) initMethods(vm, opts.methods)

/\*初始化data\*/

if (opts.data) {

initData(vm)

} else {

/\*该组件没有data的时候绑定一个空对象\*/

observe(vm.\_data = {}, true /\* asRootData \*/)

}

/\*初始化computed\*/

if (opts.computed) initComputed(vm, opts.computed)

/\*初始化watchers\*/

if (opts.watch) initWatch(vm, opts.watch)

}

...

/\*初始化data\*/

function initData (vm: Component) {

/\*得到data数据\*/

let data = vm.$options.data

data = vm.\_data = typeof data === 'function'

? getData(data, vm)

: data || {}defi

...

//遍历data中的数据

while (i--) {

/\*保证data中的key不与props中的key重复，props优先，如果有冲突会产生warning\*/

if (props && hasOwn(props, keys[i])) {

process.env.NODE\_ENV !== 'production' && warn(

`The data property "${keys[i]}" is already declared as a prop. ` +

`Use prop default value instead.`,

vm

)

} else if (!isReserved(keys[i])) {

/\*判断是否是保留字段\*/

/\*这里是我们前面讲过的代理，将data上面的属性代理到了vm实例上\*/

proxy(vm, `\_data`, keys[i])

}

}

// observe data

/\*这里通过observe实例化Observe对象，开始对数据进行绑定，asRootData用来根数据，用来计算实例化根数据的个数，下面会进行递归observe进行对深层对象的绑定。则asRootData为非true\*/

observe(data, true /\* asRootData \*/)

}

**1、initData**

现在我们重点分析下**initData**，这里主要做了两件事，一是将\_data上面的数据代理到vm上，二是通过执行 observe(data, true / *asRootData* /)将所有data变成可观察的，即对data定义的每个属性进行getter/setter操作，这里就是Vue实现响应式的基础；**observe**的实现如下 [src/core/observer/index.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Fobserver%2Findex.js)

/\*尝试创建一个Observer实例（\_\_ob\_\_），如果成功创建Observer实例则返回新的Observer实例，如果已有Observer实例则返回现有的Observer实例。\*/

export function observe (value: any, asRootData: ?boolean): Observer | void {

if (!isObject(value)) {

return

}

let ob: Observer | void

/\*这里用\_\_ob\_\_这个属性来判断是否已经有Observer实例，如果没有Observer实例则会新建一个Observer实例并赋值给\_\_ob\_\_这个属性，如果已有Observer实例则直接返回该Observer实例，这里可以看Observer实例化的代码def(value, '\_\_ob\_\_', this)\*/

if (hasOwn(value, '\_\_ob\_\_') && value.\_\_ob\_\_ instanceof Observer) {

ob = value.\_\_ob\_\_

} else if (

/\*这里的判断是为了确保value是单纯的对象，而不是函数或者是Regexp等情况。而且该对象在shouldConvert的时候才会进行Observer。这是一个标识位，避免重复对value进行Observer

\*/

observerState.shouldConvert &&

!isServerRendering() &&

(Array.isArray(value) || isPlainObject(value)) &&

Object.isExtensible(value) &&

!value.\_isVue

) {

ob = new Observer(value)

}

if (asRootData && ob) {

/\*如果是根数据则计数，后面Observer中的observe的asRootData非true\*/

ob.vmCount++

}

return ob

}

这里 **new Observer(value)** 就是实现响应式的核心方法之一了，通过它将data转变可以成观察的，而这里正是我们开头说的，用了 **Object.defineProperty** 实现了data的 **getter/setter** 操作，通过 **Watcher** 来观察数据的变化，进而更新到视图中。

**2、Observer**

Observer类是将每个目标对象（即data）的键值转换成getter/setter形式，用于进行依赖收集以及调度更新。

[src/core/observer/index.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Fobserver%2Findex.js)

export class Observer {

value: any;

dep: Dep;

vmCount: number; // number of vms that has this object as root $data

constructor (value: any) {

this.value = value

this.dep = new Dep()

this.vmCount = 0

/\* 将Observer实例绑定到data的\_\_ob\_\_属性上面去，之前说过observe的时候会先检测是否已经有\_\_ob\_\_对象存放Observer实例了，def方法定义可以参考/src/core/util/lang.js\*/

def(value, '\_\_ob\_\_', this)

if (Array.isArray(value)) {

/\*如果是数组，将修改后可以截获响应的数组方法替换掉该数组的原型中的原生方法，达到监听数组数据变化响应的效果。这里如果当前浏览器支持\_\_proto\_\_属性，则直接覆盖当前数组对象原型上的原生数组方法，如果不支持该属性，则直接覆盖数组对象的原型。\*/

const augment = hasProto

? protoAugment /\*直接覆盖原型的方法来修改目标对象\*/

: copyAugment /\*定义（覆盖）目标对象或数组的某一个方法\*/

augment(value, arrayMethods, arrayKeys)

/\*如果是数组则需要遍历数组的每一个成员进行observe\*/

this.observeArray(value)

} else {

/\*如果是对象则直接walk进行绑定\*/

this.walk(value)

},

walk (obj: Object) {

const keys = Object.keys(obj)

/\*walk方法会遍历对象的每一个属性进行defineReactive绑定\*/

for (let i = 0; i < keys.length; i++) {

defineReactive(obj, keys[i], obj[keys[i]])

}

}

}

1. 首先将Observer实例绑定到data的**ob**属性上面去，防止重复绑定；
2. 若data为数组，先实现对应的[变异方法](https://funteas.com/go/?target=https%3A%2F%2Fcn.vuejs.org%2Fv2%2Fguide%2Flist.html%23%E5%8F%98%E5%BC%82%E6%96%B9%E6%B3%95)（这里变异方法是指Vue重写了数组的7种原生方法，这里不做赘述，后续再说明），再将数组的每个成员进行observe，使之成响应式数据；
3. 否则执行walk()方法，遍历data所有的数据，进行getter/setter绑定，这里的核心方法就是 **defineReative(obj, keys[i], obj[keys[i]])**

export function defineReactive (

obj: Object,

key: string,

val: any,

customSetter?: Function

) {

/\*在闭包中定义一个dep对象\*/

const dep = new Dep()

const property = Object.getOwnPropertyDescriptor(obj, key)

if (property && property.configurable === false) {

return

}

/\*如果之前该对象已经预设了getter以及setter函数则将其取出来，新定义的getter/setter中会将其执行，保证不会覆盖之前已经定义的getter/setter。\*/

// cater for pre-defined getter/setters

const getter = property && property.get

const setter = property && property.set

/\*对象的子对象递归进行observe并返回子节点的Observer对象\*/

let childOb = observe(val)

Object.defineProperty(obj, key, {

enumerable: true,

configurable: true,

get: function reactiveGetter () {

/\*如果原本对象拥有getter方法则执行\*/

const value = getter ? getter.call(obj) : val

if (Dep.target) {

/\*进行依赖收集\*/

dep.depend()

if (childOb) {

/\*子对象进行依赖收集，其实就是将同一个watcher观察者实例放进了两个depend中，一个是正在本身闭包中的depend，另一个是子元素的depend\*/

childOb.dep.depend()

}

if (Array.isArray(value)) {

/\*是数组则需要对每一个成员都进行依赖收集，如果数组的成员还是数组，则递归。\*/

dependArray(value)

}

}

return value

},

set: function reactiveSetter (newVal) {

/\*通过getter方法获取当前值，与新值进行比较，一致则不需要执行下面的操作\*/

const value = getter ? getter.call(obj) : val

/\* eslint-disable no-self-compare \*/

if (newVal === value || (newVal !== newVal && value !== value)) {

return

}

/\* eslint-enable no-self-compare \*/

if (process.env.NODE\_ENV !== 'production' && customSetter) {

customSetter()

}

if (setter) {

/\*如果原本对象拥有setter方法则执行setter\*/

setter.call(obj, newVal)

} else {

val = newVal

}

/\*新的值需要重新进行observe，保证数据响应式\*/

childOb = observe(newVal)

/\*dep对象通知所有的观察者\*/

dep.notify()

}

})

}

其中getter方法：

1. 先为每个data声明一个 **Dep** 实例对象，被用于getter时执行dep.depend()进行收集相关的依赖;
2. 根据Dep.target来判断是否收集依赖，还是普通取值。Dep.target是在什么时候，如何收集的后面再说明，先简单了解它的作用，

那么问题来了，我们为啥要收集相关依赖呢？

new Vue({

template:

`<div>

<span>text1:</span> {{text1}}

<span>text2:</span> {{text2}}

<div>`,

data: {

text1: 'text1',

text2: 'text2',

text3: 'text3'

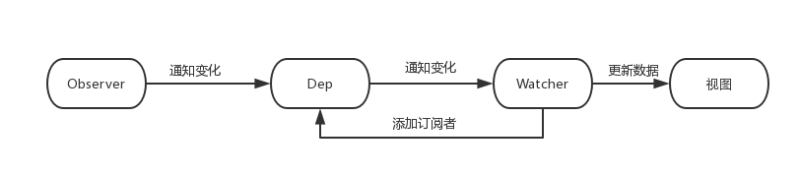
}

});

我们可以从以上代码看出，data中text3并没有被模板实际用到，为了提高代码执行效率，我们没有必要对其进行响应式处理，因此，依赖收集简单点理解就是收集只在实际页面中用到的data数据，然后打上标记，这里就是标记为Dep.target。

在setter方法中:

1. 获取新的值并且进行observe，保证数据响应式；
2. 通过dep对象通知所有观察者去更新数据，从而达到响应式效果。

在Observer类中，我们可以看到在getter时，dep会收集相关依赖，即收集依赖的watcher，然后在setter操作时候通过dep去通知watcher,此时watcher就执行变化，我们用一张图描述这三者之间的关系： 

从图我们可以简单理解：Dep可以看做是书店，Watcher就是书店订阅者，而Observer就是书店的书，订阅者在书店订阅书籍，就可以添加订阅者信息，一旦有新书就会通过书店给订阅者发送消息。

**3、Watcher**

Watcher是一个观察者对象。依赖收集以后Watcher对象会被保存在Dep的subs中，数据变动的时候Dep会通知Watcher实例，然后由Watcher实例回调cb进行视图的更新。

[src/core/observer/watcher.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Fobserver%2Fwatcher.js)

export default class Watcher {

constructor (

vm: Component,

expOrFn: string | Function,

cb: Function,

options?: Object

) {

this.vm = vm

/\*\_watchers存放订阅者实例\*/

vm.\_watchers.push(this)

// options

if (options) {

this.deep = !!options.deep

this.user = !!options.user

this.lazy = !!options.lazy

this.sync = !!options.sync

} else {

this.deep = this.user = this.lazy = this.sync = false

}

this.cb = cb

this.id = ++uid // uid for batching

this.active = true

this.dirty = this.lazy // for lazy watchers

this.deps = []

this.newDeps = []

this.depIds = new Set()

this.newDepIds = new Set()

this.expression = process.env.NODE\_ENV !== 'production'

? expOrFn.toString()

: ''

// parse expression for getter

/\*把表达式expOrFn解析成getter\*/

if (typeof expOrFn === 'function') {

this.getter = expOrFn

} else {

this.getter = parsePath(expOrFn)

if (!this.getter) {

this.getter = function () {}

process.env.NODE\_ENV !== 'production' && warn(

`Failed watching path: "${expOrFn}" ` +

'Watcher only accepts simple dot-delimited paths. ' +

'For full control, use a function instead.',

vm

)

}

}

this.value = this.lazy

? undefined

: this.get()

}

/\*\*

\* Evaluate the getter, and re-collect dependencies.

\*/

/\*获得getter的值并且重新进行依赖收集\*/

get () {

/\*将自身watcher观察者实例设置给Dep.target，用以依赖收集。\*/

pushTarget(this)

let value

const vm = this.vm

/\*执行了getter操作，看似执行了渲染操作，其实是执行了依赖收集。

在将Dep.target设置为自生观察者实例以后，执行getter操作。

譬如说现在的的data中可能有a、b、c三个数据，getter渲染需要依赖a跟c，

那么在执行getter的时候就会触发a跟c两个数据的getter函数，

在getter函数中即可判断Dep.target是否存在然后完成依赖收集，

将该观察者对象放入闭包中的Dep的subs中去。\*/

if (this.user) {

try {

value = this.getter.call(vm, vm)

} catch (e) {

handleError(e, vm, `getter for watcher "${this.expression}"`)

}

} else {

value = this.getter.call(vm, vm)

}

// "touch" every property so they are all tracked as

// dependencies for deep watching

/\*如果存在deep，则触发每个深层对象的依赖，追踪其变化\*/

if (this.deep) {

/\*递归每一个对象或者数组，触发它们的getter，使得对象或数组的每一个成员都被依赖收集，形成一个“深（deep）”依赖关系\*/

traverse(value)

}

/\*将观察者实例从target栈中取出并设置给Dep.target\*/

popTarget()

this.cleanupDeps()

return value

}

/\*\*

\* Add a dependency to this directive.

\*/

/\*添加一个依赖关系到Deps集合中\*/

addDep (dep: Dep) {

const id = dep.id

if (!this.newDepIds.has(id)) {

this.newDepIds.add(id)

this.newDeps.push(dep)

if (!this.depIds.has(id)) {

dep.addSub(this)

}

}

}

/\*\*

\* Clean up for dependency collection.

\*/

/\*清理依赖收集\*/

cleanupDeps () {

/\*移除所有观察者对象\*/

...

}

/\*\*

\* Subscriber interface.

\* Will be called when a dependency changes.

\*/

/\*

调度者接口，当依赖发生改变的时候进行回调。

\*/

update () {

/\* istanbul ignore else \*/

if (this.lazy) {

this.dirty = true

} else if (this.sync) {

/\*同步则执行run直接渲染视图\*/

this.run()

} else {

/\*异步推送到观察者队列中，下一个tick时调用。\*/

queueWatcher(this)

}

}

/\*\*

\* Scheduler job interface.

\* Will be called by the scheduler.

\*/

/\*

调度者工作接口，将被调度者回调。

\*/

run () {

if (this.active) {

/\* get操作在获取value本身也会执行getter从而调用update更新视图 \*/

const value = this.get()

if (

value !== this.value ||

// Deep watchers and watchers on Object/Arrays should fire even

// when the value is the same, because the value may

// have mutated.

/\*

即便值相同，拥有Deep属性的观察者以及在对象／数组上的观察者应该被触发更新，因为它们的值可能发生改变。

\*/

isObject(value) ||

this.deep

) {

// set new value

const oldValue = this.value

/\*设置新的值\*/

this.value = value

/\*触发回调\*/

if (this.user) {

try {

this.cb.call(this.vm, value, oldValue)

} catch (e) {

handleError(e, this.vm, `callback for watcher "${this.expression}"`)

}

} else {

this.cb.call(this.vm, value, oldValue)

}

}

}

}

/\*\*

\* Evaluate the value of the watcher.

\* This only gets called for lazy watchers.

\*/

/\*获取观察者的值\*/

evaluate () {

this.value = this.get()

this.dirty = false

}

/\*\*

\* Depend on all deps collected by this watcher.

\*/

/\*收集该watcher的所有deps依赖\*/

depend () {

let i = this.deps.length

while (i--) {

this.deps[i].depend()

}

}

/\*\*

\* Remove self from all dependencies' subscriber list.

\*/

/\*将自身从所有依赖收集订阅列表删除\*/

teardown () {

...

}

}

**4、Dep**

被Observer的data在触发 **getter** 时，**Dep** 就会收集依赖的 **Watcher** ，其实 **Dep** 就像刚才说的是一个书店，可以接受多个订阅者的订阅，当有新书时即在data变动时，就会通过 **Dep** 给 **Watcher** 发通知进行更新。

[src/core/observer/dep.js](https://funteas.com/go/?target=https%3A%2F%2Fgithub.com%2Fhuangzhuangjia%2FVue-learn%2Fblob%2Fmaster%2Fcore%2Fobserver%2Fdep.js)

export default class Dep {

static target: ?Watcher;

id: number;

subs: Array<Watcher>;

constructor () {

this.id = uid++

this.subs = []

}

/\*添加一个观察者对象\*/

addSub (sub: Watcher) {

this.subs.push(sub)

}

/\*移除一个观察者对象\*/

removeSub (sub: Watcher) {

remove(this.subs, sub)

}

/\*依赖收集，当存在Dep.target的时候添加观察者对象\*/

depend () {

if (Dep.target) {

Dep.target.addDep(this)

}

}

/\*通知所有订阅者\*/

notify () {

// stabilize the subscriber list first

const subs = this.subs.slice()

for (let i = 0, l = subs.length; i < l; i++) {

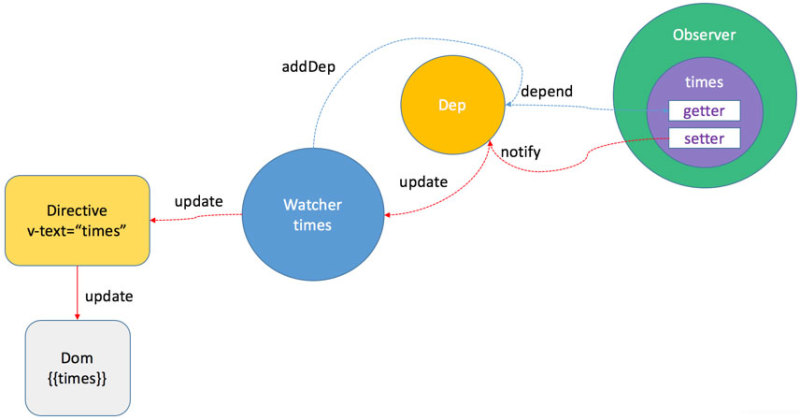
subs[i].update()

}

}

}

**总结**

其实在 **Vue** 中初始化渲染时，视图上绑定的数据就会实例化一个 **Watcher**，依赖收集就是是通过属性的 **getter** 函数完成的，文章一开始讲到的 **Observer** 、**Watcher** 、**Dep** 都与依赖收集相关。其中 **Observer** 与 **Dep** 是一对一的关系， **Dep** 与 **Watcher** 是多对多的关系，**Dep** 则是 **Observer** 和 **Watcher** 之间的纽带。依赖收集完成后，当属性变化会执行被 **Observer** 对象的 **dep.notify()** 方法，这个方法会遍历订阅者（Watcher）列表向其发送消息， **Watcher** 会执行 **run** 方法去更新视图，我们再来看一张图总结一下： 

1. 在 **Vue** 中模板编译过程中的指令或者数据绑定都会实例化一个 **Watcher** 实例，实例化过程中会触发 **get()**将自身指向 **Dep.target**;
2. data在 **Observer** 时执行 **getter** 会触发 **dep.depend()** 进行依赖收集;依赖收集的结果：1、data在 **Observer** 时闭包的dep实例的subs添加观察它的 **Watcher** 实例；2. **Watcher** 的deps中添加观察对象 **Observer** 时的闭包dep；
3. 当data中被 **Observer** 的某个对象值变化后，触发subs中观察它的watcher执行 **update()** 方法，最后实际上是调用watcher的回调函数cb，进而更新视图。

**2、vue-router实现原理？**

**深入Vue-Router源码分析路由实现原理**

**使用Vue开发SPA应用，离不开vue-router，那么vue和vue-router是如何协作运行的呢，下面从使用的角度，大白话帮大家一步步梳理下vue-router的整个实现流程。**

**到发文时使用的版本是：**

**- vue (v2.5.0)**

**- vue-router (v3.0.1)**

**一、vue-router 源码结构**

**github 地址：https://github.com/vuejs/vue-router**

**components下是两个组件<router-view> 和 <router-link>**

**history是路由方式的封装，提供三种方式**

**util下主要是各种功能类和功能函数**

**create-matcher和create-router-map是生成匹配表**

**index是VueRouter类，也整个插件的入口**

**Install 提供安装的方法**

**先整体展示下vue-router使用方式，请牢记一下几步哦。**

**import Vue from 'vue'**

**import VueRouter from 'vue-router'**

**//注册插件 如果是在浏览器环境运行的，可以不写该方法**

**Vue.use(VueRouter)**

**// 1. 定义（路由）组件。**

**// 可以从其他文件 import 进来**

**const User = { template: '<div>用户</div>' }**

**const Role = { template: '<div>角色</div>' }**

**// 2. 定义路由**

**// Array,每个路由应该映射一个组件。**

**const routes = [**

**{ path: '/user', component: User },**

**{ path: '/home', component: Home }**

**]**

**// 3. 创建 router 实例，并传 `routes` 配置**

**const router = new VueRouter({**

**routes**

**})**

**// 4. 创建和挂载根实例。**

**// 记得要通过 router 对象以参数注入Vue，**

**// 从而让整个应用都有路由功能**

**// 使用 router-link 组件来导航.**

**// 路由出口**

**// 路由匹配到的组件将渲染在这里**

**const app = new Vue({**

**router,**

**template: `**

**<div id="app">**

**<h1>Basic</h1>**

**<ul>**

**<li><router-link to="/">/</router-link></li>**

**<li><router-link to="/user">用户</router-link></li>**

**<li><router-link to="/role">角色</router-link></li>**

**<router-link tag="li" to="/user">/用户</router-link>**

**</ul>**

**<router-view class="view"></router-view>**

**</div>**

**`**

**}).$mount('#app')**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**32**

**33**

**34**

**35**

**36**

**37**

**38**

**39**

**40**

**41**

**42**

**43**

**分析开始**

**第一步**

**Vue是使用.use( plugins )方法将插件注入到Vue中。**

**use方法会检测注入插件VueRouter内的install方法，如果有，则执行install方法。**

**注意：如果是在浏览器环境，在index.js内会自动调用.use方法。如果是基于node环境，需要手动调用。**

**if (inBrowser && window.Vue) {**

**window.Vue.use(VueRouter)**

**}**

**1**

**2**

**3**

**Install解析 (对应目录结构的install.js)**

**该方法内主要做了以下三件事：**

**1、对Vue实例混入beforeCreate钩子操作（在Vue的生命周期阶段会被调用）**

**2、通过Vue.prototype定义router、router、route 属性（方便所有组件可以获取这两个属性）**

**3、Vue上注册router-link和router-view两个组件**

**export function install (Vue) {**

**if (install.installed && \_Vue === Vue) return**

**install.installed = true**

**\_Vue = Vue**

**const isDef = v => v !== undefined**

**const registerInstance = (vm, callVal) => {**

**let i = vm.$options.\_parentVnode**

**if (isDef(i) && isDef(i = i.data) && isDef(i = i.registerRouteInstance)) {**

**i(vm, callVal)**

**}**

**}**

**Vue.mixin({**

**//对Vue实例混入beforeCreate钩子操作**

**beforeCreate () {**

**if (isDef(this.$options.router)) {**

**this.\_routerRoot = this**

**this.\_router = this.$options.router**

**this.\_router.init(this)**

**Vue.util.defineReactive(this, '\_route', this.\_router.history.current)**

**} else {**

**this.\_routerRoot = (this.$parent && this.$parent.\_routerRoot) || this**

**}**

**registerInstance(this, this)**

**},**

**destroyed () {**

**registerInstance(this)**

**}**

**})**

**//通过Vue.prototype定义$router、$route 属性（方便所有组件可以获取这两个属性）**

**Object.defineProperty(Vue.prototype, '$router', {**

**get () { return this.\_routerRoot.\_router }**

**})**

**Object.defineProperty(Vue.prototype, '$route', {**

**get () { return this.\_routerRoot.\_route }**

**})**

**//Vue上注册router-link和router-view两个组件**

**Vue.component('RouterView', View)**

**Vue.component('RouterLink', Link)**

**const strats = Vue.config.optionMergeStrategies**

**// use the same hook merging strategy for route hooks**

**strats.beforeRouteEnter = strats.beforeRouteLeave = strats.beforeRouteUpdate = strats.created**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**32**

**33**

**34**

**35**

**36**

**37**

**38**

**39**

**40**

**41**

**42**

**43**

**44**

**45**

**46**

**47**

**48**

**第二步 生成router实例**

**const router = new VueRouter({**

**routes**

**})**

**1**

**2**

**3**

**生成实例过程中，主要做了以下两件事**

**1、根据配置数组(传入的routes)生成路由配置记录表。**

**2、根据不同模式生成监控路由变化的History对象**

**注：History类由HTML5History、HashHistory、AbstractHistory三类继承**

**history/base.js实现了基本history的操作**

**history/hash.js，history/html5.js和history/abstract.js继承了base，只是根据不同的模式封装了一些基本操作**

**第三步 生成vue实例**

**const app = new Vue({**

**router,**

**template: `**

**<div id="app">**

**<h1>Basic</h1>**

**<ul>**

**<li><router-link to="/">/</router-link></li>**

**<li><router-link to="/user">用户</router-link></li>**

**<li><router-link to="/role">角色</router-link></li>**

**<router-link tag="li" to="/user">/用户</router-link>**

**</ul>**

**<router-view class="view"></router-view>**

**</div>**

**`**

**}).$mount('#app')**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**代码执行到这，会进入Vue的生命周期，还记得第一步Vue-Router对Vue混入了beforeCreate钩子吗，在此会执行哦**

**Vue.mixin({**

**beforeCreate () {**

**//验证vue是否有router对象了，如果有，就不再初始化了**

**if (isDef(this.$options.router)) { //没有router对象**

**//将\_routerRoot指向根组件**

**this.\_routerRoot = this**

**//将router对象挂载到根组件元素\_router上**

**this.\_router = this.$options.router**

**//初始化，建立路由监控**

**this.\_router.init(this)**

**//劫持数据\_route，一旦\_route数据发生变化后，通知router-view执行render方法**

**Vue.util.defineReactive(this, '\_route', this.\_router.history.current)**

**} else {**

**//如果有router对象，去寻找根组件，将\_routerRoot执行根组件（解决嵌套关系时候\_routerRoot指向不一致问题）**

**this.\_routerRoot = (this.$parent && this.$parent.\_routerRoot) || this**

**}**

**registerInstance(this, this)**

**},**

**destroyed () {**

**registerInstance(this)**

**}**

**})**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**代码执行到这，初始化结束，界面将显示默认首页**

**路由更新方式：**

**一、主动触发**

**router-link绑定了click方法，触发history.push或者history.replace，从而触发history.transitionTo。**

**transitionTo用于处理路由转换，其中包含了updateRoute用于更新\_route。**

**在beforeCreate中有劫持\_route的方法，当\_route变化后，触发router-view的变化。**

**二、地址变化（如：在浏览器地址栏直接输入地址）**

**HashHistory和HTML5History会分别监控hashchange和popstate来对路由变化作对用的处理 。**

**HashHistory和HTML5History捕获到变化后会对应执行push或replace方法，从而调用transitionTo**

**,剩下的就和上面主动触发一样啦。**

**总结**

**1、安装插件**

**混入beforeCreate生命周期处理，初始化\_routerRoot，\_router，\_route等数据**

**全局设置vue静态访问router和router和route，方便后期访问**

**完成了router-link和 router-view 两个组件的注册，router-link用于触发路由的变化，router-view作 为功能组件，用于触发对应路由视图的变化**

**2、根据路由配置生成router实例**

**根据配置数组生成路由配置记录表**

**生成监控路由变化的hsitory对象**

**3、将router实例传入根vue实例**

**根据beforeCreate混入，为根vue对象设置了劫持字段\_route，用户触发router-view的变化**

**调用init()函数，完成首次路由的渲染，首次渲染的调用路径是 调用history.transitionTo方法，根据router的match函数，生成一个新的route对象**

**接着通过confirmTransition对比一下新生成的route和当前的route对象是否改变，改变的话触发updateRoute，更新hsitory.current属性，触发根组件的\_route的变化,从而导致组件的调用render函数，更新router-view**

**另外一种更新路由的方式是主动触发**

**router-link绑定了click方法，触发history.push或者history.replace,从而触发history.transitionTo**

**同时会监控hashchange和popstate来对路由变化作对用的处理**

**---------------------**

**作者：陨落的轨迹**

**来源：CSDN**

**原文：https://blog.csdn.net/u013938465/article/details/79421239?utm\_source=copy**

**版权声明：本文为博主原创文章，转载请附上博文链接！**

# **3、为什么要选vue？与其它框架对比的优势和劣势？**

# **4、vue如何实现父子组件通信，以及非父子组件通信？**

## **5、vuejs与angularjs以及react的区别？**

## **6、vuex是用来做什么的？**

**VueX 是一个专门为 Vue.js 应用设计的状态管理架构，统一管理和维护各个vue组件的可变化状态(你可以理解成 vue 组件里的某些 data )。如下图，所示：**

**一、什么是Vuex**

**用学过react的人来说:Vuex 之于 Vue 就像 Redux 之于 React,这边附上一篇Vuex文档地址http://vuex.vuejs.org/en/intro.html**

**Vuex是集中的应用程序架构为了Vue.js应用程序状态管理。灵感来自Flux 和 Redux,但简化的概念和实现是一个专门为 Vue.js 应用设计的状态管理架构。**

**状态管理: 简单理解就是统一管理和维护各个vue组件的可变化状态(你可以理解成 vue 组件里的某些 data )**

**二、我们为什么需要Vuex**

**如果你的应用程序很简单,你可能不需要Vuex。不要过早地应用它。但是如果你正在构建一个medium-to-large-scale SPA,那么你遇到的情况,让你思考如何更好的结构Vue组件之外的事情。这是Vuex发挥作用的地方。**

**当单独使用Vue.js,我们常常倾向于存储状态我们的组件内。也就是说,每个组件属于我们的应用程序状态,因此结果状态乱扔的到处都是。然而,有时一块状态需要由多个组件共享。常见的做法是让一个组件“发送”一些使用自定义事件系统其他组件。这种模式的问题是内部的事件流大组件树很快就变得复杂,通常很难原因时出现错误。**

**更好地处理共享状态在大型应用程序中,我们需要区分组件本地状态和应用程序级状态。应用程序状态不属于一个特定的组件,但我们的组件还可以观察反应DOM更新。通过集中管理在一个地方,我们不再需要传递事件,因为一切影响多个组件应该属于那里。此外,这让我们记录和检查状态变化的每一个突变更容易理解,甚至实现花哨的东西穿越调试。**

**Vuex也执行一些意见如何状态管理逻辑分割成不同的地方,但仍然允许足够的灵活性的实际代码结构。**

**三、单向数据流**

**作用：减少数据传递，数据统一存放管理**

**四、callApi()**

**在action中使用，有五个参数分别是（ApiCode,parent,success,false,fail）**

**success,成功后回调，false更新数据**

**一般用前三个参数**

**五、1、store 2、action 3、Vuex  4、添加到store.js**

**---------------------**

**作者：CURRY\_zhao**

**来源：CSDN**

**原文：https://blog.csdn.net/qq\_24122593/article/details/53129900?utm\_source=copy**

**版权声明：本文为博主原创文章，转载请附上博文链接！  
7、vue源码结构**

.谈谈你对vue的认识

vue概念：是一个构建用户界面的渐进式框架，典型的MVVM框架。

注：模型（Model）只是普通的JavaScript对象，修改它则视图（View）会自动更新。这种设计让状态管理变得非常简单而直观。

vue作用：响应式的数据绑定和组合的视图组件

vue原理：数据双向绑定     模板编译和虚拟dom

Vue实现数据双向绑定的效果，需要三大模块：

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订阅者

Compile：对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数

Watcher：作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图。

Observer的核心是通过Obeject.defineProperty()来监听数据的变动，这个函数内部可以定义setter和getter，每当数据发生变化，就会触发setter。这时候Observer就要通知订阅者，订阅者就是Watcher。

Watcher订阅者作为Observer和Compile之间通信的桥梁，主要做的事情是：

在自身实例化时往属性订阅器(dep)里面添加自己

自身必须有一个update()方法

待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调

Compile主要做的事情是解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图。

vue生命周期：vue 实例从创建到销毁的过程，就是Vue 的生命周期，即开始创建、初始化数据、编译模板、挂载Dom→渲染、                            更新→渲染、卸载等一系列过程。总共分为8个阶段：

beforeCreate----创建前 组件实例更被创建，组件属性计算之前，数据对象data都为undefined，未初始化。

created----创建后 组件实例创建完成，属性已经绑定，数据对象data已存在，但dom未生成，$el未存在

beforeMount---挂载前 vue实例的$el和data都已初始化，挂载之前为虚拟的dom节点，data.message未替换

mounted-----挂载后 vue实例挂载完成，data.message成功渲染。

beforeUpdate----更新前 当data变化时，会触发beforeUpdate方法

updated----更新后 当data变化时，会触发updated方法

beforeDestory---销毁前 组件销毁之前调用

destoryed---销毁后 组件销毁之后调用，对data的改变不会再触发周期函数，vue实例已解除事件监听和dom绑定，但dom结构依然存在 。

vue的优点：低耦合、可重用、性独立开发、可测试 。

vue和其他框架的对比：

前期借鉴了angular和react的一些优秀思想，比如虚拟dom、指令操作等

more

2.MVVM框架是什么？他和其他框架（jQuery）有什么区别，使用场景

MVVM框架：一个model+view+view-model的框架，model 是数据模型，view是视图，view-model连接数据和视图。

视图的输入框绑定了v-model, 用户输入后会改变data；Model改变也会同步视图更新相关的依赖, 双向绑定就是vm起了作用。

区别：vue数据驱动，通过数据来显示视图层而不是节点操作。

使用场景：数据操作比较多的场景更加方便快捷。

3.MVC框架和MVVM框架的区别

mvc和mvvm都是一种设计思想，主要是mvc中Controller演变成mvvm中的viewModel。

mvvm主要解决了mvc中大量的DOM 操作使页面渲染性能降低，加载速度变慢，影响用户体验，以及当 Model 频繁发生变化，开发者需要主动更新到View，这些问题。

4.vuex是什么，它的原理

vuex：状态管理器，实现组件间的数据共享。

原理：一个应用可以看作是由View, Actions,State三部分组成,数据的流动也是从View => Actions => State =>View 以此达到数据的单向流动。但是项目较大的, 组件嵌套过多的时候, 多组件共享同一个State会在数据传递时出现很多问题.Vuex就是为了解决这些问题而产生的.Vuex可以被看作项目中所有组件的数据中心,我们将所有组件中共享的State抽离出来,任何组件都可以访问和操作我们的数据中心。

一个实例化的Vuex.Store由state, mutations和actions三个属性组成:

state中保存着共有数据

改变state中的数据有且只有通过mutations中的方法,且mutations中的方法必须是同步的

如果要写异步的方法,需要些在actions中, 并通过commit到mutations中进行state中数据的更改。

注：官网https://vuex.vuejs.org/

5.列举vue的指令及用法

v-for：遍历循环

v-html、v-text：文本信息

v-model：实现双向绑定

v-if、v-show:  判断是否隐藏显示

v-bind:class  绑定属性

@click    绑定事件

6.vue里面的自定义指令是什么，其中的钩子函数有哪些？

vue.directive,可以写在组件内部，也可以写在外部作为全局的使用。

它的钩子有bind，inserted，update等

7.vue组件之间如何传值通信

父到子：

 子组件在props中创建一个属性，用来接收父组件传过来的值；

在父组件中注册子组件；

在子组件标签中添加子组件props中创建的属性；

把需要传给子组件的值赋给该属性

子到父：

子组件中需要以某种方式（如点击事件）的方法来触发一个自定义的事件；

将需要传的值作为$emit的第二个参数，该值将作为实参传给响应事件的方法；

在父组件中注册子组件并在子组件标签上绑定自定义事件的监听。

平行组件：

$emit推送，$on接收

8.vue组件之间如何跳转

  路由配置好之后，可以使用下面三总方式进行组件的跳转展示

 ① 直接修改地址栏的路由路径

 ② 用router-link标签的to属性配置path即可

 ③通过js编程方式，在事件里面调用this.$router.push("/home")实现

9.vue中跨域问题如何解决

  ① 后台更改header：

header('Access-Control-Allow-Origin:\*');    //允许所有来源访问

header('Access-Control-Allow-Method:POST,GET'); //允许访问的方式

  ② 使用JQuery提供的jsonp :  发起ajax请求，设置dataType为jsonp

  ③ 使用http-proxy-middleware 代理解决

10.es6和es5对比，有何改变

     es6常用语法：

 变量声明const和let

 import导入模块、export导出模块

 class类

 promise

 箭头函数

 模板字符串

---------------------

作者：keep789

来源：CSDN

原文：https://blog.csdn.net/keep789/article/details/82811984?utm\_source=copy

版权声明：本文为博主原创文章，转载请附上博文链接！