**vue断点调试**

|  |
| --- |
|  |

**指令：当表达式的值发生改变时，将这个变化反应到DOM上，指令绑定的值都是字符串**

**console.log(app.$data); // 会给data里的属性添加get、set方法**

|  |
| --- |
| vue vue-router vuex vue-loader vue-cli axios rxjs mobx lodash element-ui iview weex 小程序  nuxt webpack gulp css html es6 typescript 数据结构和算法 设计模式 npm yarn  git svn mock echart jquery bootstrap zepto underscore eslint |

# vue组件与自定义元素的关系

|  |
| --- |
| <https://cn.vuejs.org/v2/guide/index.html#与自定义元素的关系> |

# 核心思想：数据驱动 组件化 虚拟dom

|  |
| --- |
| 双向数据绑定  view <=*=>* viewModel <=*=>* model  dom vue pojo（原生js对象）  如何实现双向数据绑定  *Object*.defineProperty()  <input type="text", id="a">  <span id="b"></span>  *var* obj = {};  *Object*.defineProperty(obj, 'hello', {  set: *function*(*newVal*) {  document.getElementById('a').value = newVal;  document.getElementById('b').innerHtml = newVal;  }  })  document.addEventListener('keyup', *function*(*e*){  obj.hello = e.terget.value;  }) |

# 创建一个Vue实例

|  |
| --- |
| 通过Vue构造函数创建： <b>*var* vm = new Vue({ // 选项 })</b>  一个Vue应用应由<b>Vue根实例</b>，以及<b>可选的、嵌套的、可复用的</b>组件树构成  所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象 (*一些根实例特有的选项除外*) |

# 数据与方法

|  |
| --- |
| 一个Vue实例被创建时，会把data对象里的属性加入Vue的响应式系统，属性值改变时，视图会进行重渲染  只有当实例被创建时data中存在的属性才是响应式的，如下  // 添加新的属性,对b的改动不触发视图的更新  // 解决：一开始为空或不存在，后面再赋值  new Vue({  data: {  b: 'hello'  }  })  vm.b = 'hi';  b // hello  *Object*.freeze()会阻止修改现有的属性，也意味着响应系统无法再追踪变化  <p>{{ foo }}</p>  <!-- 这里的 `foo` 不会更新！ -->  <button v-on:click="foo = 'baz'">Change it</button>  *var* obj = {  foo: 'bar'  }  *Object*.freeze(obj)  new Vue({  el: '#app',  data: obj  }) |

# npm全局安装路径

|  |
| --- |
| npmq全局安装路径 |

# vue-cli使用scss

|  |
| --- |
| <https://www.jianshu.com/p/8e60048baeb7>  <https://www.cnblogs.com/rainheader/p/6505366.html> |

# vue多环境打包和多环境api配置

|  |
| --- |
| <https://www.jb51.net/article/149741.htm> |

# elementui按需引入

|  |
| --- |
| <https://www.jianshu.com/p/40da0ba35ac1>  <https://github.com/bill-mark/elementui-demand>  {  "presets": [  ["env", {  "modules": false,  "targets": {  "browsers": ["> 1%", "last 2 versions", "not ie <= 8"]  }  }],  ["es2015", { "modules": false }],  "stage-2"  ],  "plugins": [  "transform-vue-jsx",  "transform-runtime",  [  "component",  {  "libraryName": "element-ui",  "styleLibraryName": "theme-chalk"  }  ]  ],  "env": {  "test": {  "presets": ["env", "stage-2"],  "plugins": ["transform-vue-jsx", "transform-es2015-modules-commonjs", "dynamic-import-node"]  }  }  } |

# Echarts引入

|  |
| --- |
|  |

# vue使用Nprogress进度条

|  |
| --- |
| <https://blog.csdn.net/qq_35844177/article/details/70171054>  <https://blog.csdn.net/qq_36410795/article/details/85086621>  <https://www.helloweba.net/javascript/438.html>  1.main.js引入  import 'nprogress/nprogress.css'  2.在使用的文件中引入  import NProgress from 'nprogress'  3.使用 |

# vue引入Normalize.css初始化

|  |
| --- |
| <https://www.ixiqin.com/2019/02/vue-project-introduced-the-normalize-css-to-format-the-initialization/> |

# vue-I18n国际化

|  |
| --- |
| <http://kazupon.github.io/vue-i18n/zh/introduction.html>  <https://blog.csdn.net/fabulous1111/article/details/79493825> |

# vue引入图片

|  |
| --- |
| data () {  return {  locationIcon: require('../../assets/images/location.jpeg'),  }  }, |

# mustache插值

|  |
| --- |
| mustache插值在网络差时，会显示出来，v-text不会  <div v-text="msg">{{msg}}</div>  mustache标签 可以接受表达式的值  {{cents/10}}  {{true? 0: 1}}  {{example.split(",")}}  {{example | filterA | filterB}}  {{example | filter a b}}  {{var a = 10}} 错  {{if(true) return "haha"}} 错 |

# v-text

|  |
| --- |
| 更新部分的 textContent ，需要使用 {{ mustache }} 插值  <span v-text="msg"></span>  等价于  <span>{{msg}}</span> |

# v-html

|  |
| --- |
| 更新元素的 innerHTML,会解析html代码,容易导致 XSS 攻击  <span v-html="html"></span> |

# v-show

|  |
| --- |
| 注意: v-show 不支持<template>语法, 也不支持 v-else  据表达式之真假值，切换元素的 display CSS 属性  <input type="button" value="按钮" @click="flag=!flag">  <span v-show="flag"> HTML </span>  <span v-else> CSS </span>  data:{  flag:true  } |

# v-if

|  |
| --- |
| 1.在切换时,元素及它的数据绑定 / 组件被销毁并重建。如果元素是 <template> ，将提出它的内容作为条件块。  2.v-else 元素必须紧跟在 v-if 元素或者 v-else-if的后面——否则它不能被识别  <span v-if="flag1 = num>11?true:false">HTML</span>  <span v-else-if="flag2 = num>12?true:false">CSS</span>  <span v-else>JS</span>  data:{  num:120,  }  <template v-if="ok">  <h1>HTML</h1>  <p>CSS</p>  <p>JS</p>  </template> |

# v-bind

|  |
| --- |
| v-bind:class  值为数组时，数组的值可以是对象(多个)或字符串(多个)，推荐在data写好类样式，再赋值,  值为对象时，也推荐在data写好，再赋值  .red {*color*:red;}  .font {*font-size*:24px;}  .bg {*background*:blue;}  <span v-bind:class="[red,font]">HTML</span> <!--:class值为数组-->  <span v-bind:class="arr">HTML</span> <!--:class值为数组-->  <span v-bind:class="[json,json1]">HTML</span> <!--:class值为数组，数组值是对象-->  <span v-bind:class="arr1">HTML</span> <!--:class值为数组，数组值是对象-->  <span v-bind:class="{red:true, font:false}">HTML</span> <!--:class值为对象-->  <span v-bind:class="{red:aaa, font:bbb}">HTML</span> <!--:class值为对象-->  <span v-bind:class="json">HTML</span> <!--:class值为对象,不能多个对象-->  选中isOk则为true  <input type="checkbox" id="isTrue" v-model="isOk">  <label for='isTrue'>isOk={{isOk}}</label>  <span :class="isOk?classA:classB">绑定class中的三元表达式判断</span>  data:{  red:"red",  font:"font",  arr:["red","font"],  aaa:true,  bbb:false  json:{red:true,font:false},  json1:{bg:true},  arr1:[  {red:true, font:false},  {bg:true}  ]  }  ======================================================================================  v-bind:style  值为数组时，数组的值只能是对象(多个)，推荐在data写好类样式，再赋值,  值为对象时，也推荐在data写好，再赋值  注意：复合样式，采用驼峰命名法  <span v-bind:style="{color:'red',fontSize:'24px'}">HTML</span> <!--值为对象-->  <span v-bind:style="{color:'red',fontSize:fontSize+'px'}">HTML</span> <!--值为对象-->  <span v-bind:style="json">HTML</span> <!--值为对象-->  <span v-bind:style="[json,json1]">HTML</span> <!--值为数组-->  data:{  fontSize:24,  json:{color:'red',fontSize:"24px"},  json1:{background:"blue"}  }  ===================================================================================  注意：  <button v-bind:disabled="flag">Button</button> <!—flag为false的时，该属性会被移除-->  <img src="{{url}}"/> <!—效果能出来，但是会报一个404错误-->  <img :src="url"/>  <img v-bind:src="url" v-bind:width="width" :title="title"/>  data:{  url:'lyf.jpg',  width:"200px",  title: "vue"  } |

# v-for

|  |
| --- |
| 可以用带有 v-for 的 <template> 标签来渲染多个元素块  v-for 默认行为试着不改变整体，而是替换元素。迫使其重新排序的元素,您需要提供一个 key 的特殊属性,  数组或对象有重复的值时要添加该值才输出  vue2.0里面默认就可以添加重复数据,去掉了隐式一些变量($index $key)  track-by="id" 变成 :key="index" 提高性能  之前:v-for="(index,val) in array"  现在:v-for="(val,index) in array"  <li v-for="(val,index) in arr" :key="index">{{val}} {{index}}</li>  <li v-for="(val,key,index) of obj">{{val}} {{key}} {{index}}</li>  <button @click="add">按钮</button>  data: {  arr:['apple','banana','orange','pear','pear'],  obj:{a:'apple',b:'banana',c:'orange'}  },  methods: {  add() {  this.arrs.push('vue2') //vue2可以无限添加  }  } |

|  |
| --- |
| push() pop() shift() unshife() splice() sort() reverse() |

**v-for 练习**

|  |
| --- |
| <li v-for="item in sortItems">{{item}}</li>  <li v-for="(student,index) in sortStudent">  {{index}}：{{student.name}} - {{student.age}}  </li>  data:{  items:[20,23,18,65,32,19,5,56,41],  students:[  {name:'jspang',age:32},  {name:'Panda',age:30},  {name:'PanPaN',age:21},  {name:'King',age:45}  ]  },  computed:{  sortItems:*function*(){  return this.items.sort(sortNumber);  },  sortStudent:*function*(){  return sortByKey(this.students,'age');  }  }  //数组对象方法排序:  *function* sortByKey(*array*,*key*){  return array.sort(*function*(*a*,*b*){  *var* x=a[key];  *var* y=b[key];  return ((x<y)?-1:((x>y)?1:0));  });  }  *function* sortNumber(*a*,*b*){  return a-b  } |

# v-ref

# v-cloak

|  |
| --- |
| Vue遇到v-cloak时，会先去掉，等实例准备完毕才显示  v-cloak 防止闪烁, 比较大段落v-text v-html  <style>  [v-cloak]{  *display*: none;  }  </style>  <div v-cloak>HTML</div> |

# v-pre

|  |
| --- |
| <span v-pre>{{msg}}</span> 原样输出 |

# v-once

|  |
| --- |
| 只渲染元素和组件一次  <input type="text" v-model="msg">  <div>{{msg}}</div>  <div v-once>{{msg}}</div> 继续输入不会再渲染  <my-component v-once :comment="msg"></my-component> |

# v-model

|  |
| --- |
| 修饰符：lazy number debounce  <input type="text" v-model="message">  <input type="text" v-model.lazy="message"> 失去焦点时才更新  <input type="text" v-model.number="message"> 开始输入是string，后面再输入number，number无效  <input type="text" v-model.trim="message"> |

# v-slot

技术胖

|  |
| --- |
| v-slot旧版本  slot:位置、槽口，作用: 占个位置  <div id="app">  <jspang>  <span slot="bolgUrl">{{jspangData.bolgUrl}}</span>  <span slot="netName">{{jspangData.netName}}</span>  <span slot="skill">{{jspangData.skill}}</span>  </jspang>  </div>  <template id="tmp">  <div>  <p>博客地址：<slot name="bolgUrl">默认显示我</slot></p>  <p>网名：<slot name="netName">默认显示我</slot></p>  <p>技术类型：<slot name="skill">默认显示我</slot></p>  </div>  </template>  *var* jspang={  template:'#tmp'  }  *var* app=new Vue({  el:'#app',  data:{  jspangData:{  bolgUrl:'http://jspang.com',  netName:'技术胖',  skill:'Web前端'  }  },  components:{  "jspang":jspang  }  }) |

# v-on

## 鼠标事件

|  |
| --- |
| 缩写 @click  click/mouseout/mouseover/dblclick/mousedown/contextmenu.....  <input type="button" v-on:click="add" value="按钮"/>  <input type="button" v-on:click="add($event,10)" value="按钮"/> 传参  add:*function*(*ev*,*num*){  *console*.log(ev,num)  } |

## 键盘事件

|  |
| --- |
| 常用键:  keyup, keydown 区别，keydown按下就触发  @keyup.13 或 @keyup.enter 回车  @keyup/keydown.left  @keyup/keydown.right  @keyup/keydown.up  @keyup/keydown.down  <input type="text" @keyup="show($event)"/>  <input type="text" @keyup.enter="show($event)"/>等价于  <input type="text" @keyup.13="show($event)"/>  <input type="text" @keydown.c="show($event)"/> 等价于  <input type="text" @keydown.67="show($event)"/>  show:*function*(*ev*){  *console*.log(ev.keyCode);  } |

## 冒泡 阻止默认行为

|  |
| --- |
| <input type="text" @click.stop="show($event)"/><!-- vue阻止冒泡 -->  <input type="text" @contextmenu.prevent="show($event)"/> <!-- 阻止默认行为(默认事件) -->  <form v-on:submit.prevent="onSubmit"></form> <!-- 提交事件不再重载页面 -->  show:*function*(*ev*){  ev.cancelBubble = true;//js阻止冒泡  ev.preventDefault(); // js阻止默认行为  *console*.log(ev.keyCode);  } |

## 自定义键盘信息

|  |
| --- |
| <input type="text" @keyup.ctrl="change($event)">  Vue.directive('on').keyCodes.ctrl=17; //等价于  Vue.config.keyCodes.ctrl=13;  change(*ev*){  *console*.log(ev);  }  =========================================== |

|  |
| --- |
| Vue1.0  <!-- 要自定义指令才会输出 -->  <input type="text" @keydown.ctrl="show1"/>  <input type="text" @keydown.myenter="show2 | debounce 2000"/>  Vue.directive('on').keyCodes.ctrl=17;  Vue.directive('on').keyCodes.myenter=13;  methods:{  show1:*function*(*ev*){  *console*.log(ev.keyCode);  },  show2:*function*(*ev*){  *console*.log(ev.keyCode);  }  } |

# 全局配置

## Vue.config

|  |
| --- |
| Vue.config 是一个对象，包含 Vue 的全局配置。可以在启动应用之前修改下列属性  IMG_256 |

## silent

|  |
| --- |
| <div id="app">  <span>{{msg}}</span>  </div>  // Vue.config.silent = true; // 取消 Vue 所有的日志与警告。  *var* vm = new Vue({  el: '#app',  data: {  //msg:'hello world' // 使用了未定义的data属性msg  }  });  IMG_256 |

## optionMergeStrategies

|  |
| --- |
| // 全局注册混入对象,将会影响到 所有 之后创建的 Vue 实例  *let* globalMinin = Vue.mixin({  created: *function* () {  *var* haha = this.$options.haha || 'xixi'  *console*.log(haha)  },  myOption: {  getters: 'getToVal',  state: 'staToVal',  actions: 'actToVal'  }  })  *const* merge = Vue.config.optionMergeStrategies.computed // 使用computed合并策略  // 将globalMinin的myOption和myVue的myOption按照 自定义选项合并策略myOption进行合并  Vue.config.optionMergeStrategies.myOption = *function* (*toVal*, *fromVal*, *vm*) {  *console*.dir(toVal)  *console*.dir(fromVal)  *console*.dir(vm)  *console*.dir('-----------------')  if (!toVal) return fromVal  if (!fromVal) return toVal  return {  getters: merge(toVal.getters, fromVal.getters),  state: merge(toVal.state, fromVal.state),  actions: merge(toVal.actions, fromVal.actions)  }  }  /\* eslint-disable no-new \*/  *let* myVue = new Vue({  el: '#app',  haha: 'haha',  myOption: {  getters: 'getFromVal',  state: 'staFromVal',  actions: 'actFromVal'  },  router,  store,  mock,  components: { App },  template: '<App/>'  })  IMG_256 |

## devtools

|  |
| --- |
| // 务必在加载 Vue 之后，立即同步设置以下内容  Vue.config.devtools = true  /\* 配置是否允许 vue-devtools 检查代码。开发版本默认为 true，生产版本默认为 false。  生产版本设为 true 可以启用检查。\*/  IMG_256 |

## errorHandler

|  |
| --- |
| // 指定组件的 渲染和观察期间 未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue 实例。  Vue.config.errorHandler = *function* (*err*, *vm*, *info*) {  // handle error  // info 是 Vue 特定的错误信息，比如错误所在的生命周期钩子  // 只在 2.2.0+ 可用  *console*.log('---------------err-----------------');  *console*.log(err);  *console*.log('---------------info---------------');  *console*.log(info);  *console*.log('---------------vm------------------');  *console*.log(vm);  *console*.log('---------------arguments------------------');  *console*.log(arguments)  }  *var* vm = new Vue({  el: '#app',  data: {  msg:'hello world'  },  created: *function* () {  *var* x = '';  if(x === '') throw "值为空";  else if(isNaN(x)) throw "不是数字";  else if(x>10) throw "太大";  /\*try {  if(x === '') throw "值为空";  else if(isNaN(x)) throw "不是数字";  else if(x>10) throw "太大";  } catch(err) {  console.log('err: ' + err);  }\*/  }  });  IMG_256 |

## warnHandler

|  |
| --- |
| <div id="app">  <span>{{msg}}</span>  </div>  // 为 Vue 的运行时警告赋予一个自定义处理函数。注意这只会在开发者环境下生效，在生产环境下它会被忽略。  // msg trace 跟[Vue warn] 一样  Vue.config.warnHandler = *function* (*msg*, *vm*, *trace*) {  *console*.log('---------------msg-----------------');  *console*.log(msg);  *console*.log('---------------trace---------------');  *console*.log(trace); // `trace` 是组件的继承关系追踪  *console*.log('---------------vm------------------');  *console*.log(vm);  }  *var* vm = new Vue({  el: '#app',  data: {  //msg:'hello world' // 使用了未定义的data属性msg  }  });  IMG_256 |

## ignoredElements

## keyCodes

|  |
| --- |
| <input type="text" @keydown.media-play-pause="change($event)">  // 按下keyCose为179键才触发change  Vue.config.keyCodes = {  v: 86,  f1: 112,  // camelCase 不可用  mediaPlayPause: 179,  // 取而代之的是 kebab-case 且用双引号括起来  "media-play-pause": 179,  up: [38, 87]  }  *var* vm = new Vue({  el: '#app',  data: {  msg:'hello world'  },  methods:{  change:*function*(*ev*){  *console*.log(ev.keyCode);  *console*.log('改变了');  }  }  }); |

## performance

|  |
| --- |
| // <my-aaa></my-aaa>  // 设置为 true 以在浏览器开发工具的 性能/时间线 面板中启用对组件初始化、编译、渲染和打补丁的性能追踪。  // 只适用于开发模式和支持 performance.mark API 的浏览器上。  Vue.config.performance = true;  //全局组件  Vue.component('my-aaa',{  template:'<h3 @click="change">{{msg}}</h3>',  data(){  return { msg:'html'}  },  methods:{  change() {this.msg='javascript'; }  }  })  *var* vm = new Vue({  el: '#app',  data: {  msg:'hello world'  }  });  查看自定义组件<my-aaa></my-aaa>的各个阶段花费的时间：  IMG_256 |

## productionTip

|  |
| --- |
| Vue.config.productionTip = false; // 设置为 false 以阻止 vue 在启动时生成生产提示。  IMG_256  IMG_256 |

# 全局 API

## Vue.extend

|  |
| --- |
| <author></author>  <div id="author"></div>  /\*  扩展实例构造器，创建一个“子类”，data是函数，返回一个对象,经常和组件使用  \*/  *var* authorExtend = Vue.extend({  template:"<p><a :href='authorUrl'>{{authorName}}</a></p>",  data:*function*(){  return{  authorName:'xiaoming',  authorUrl:'http://www.com'  }  }  });  *console*.log(new authorExtend());  new authorExtend().$mount('author');  new authorExtend().$mount('#author'); |

案例

|  |
| --- |
| <shi-peng></shi-peng>  <div id="shi-peng"></div>  /\*  Vue.extend 返回的是一个“扩展实例构造器”,也就是预设了 "部分" 选项的Vue实例构造器。  经常服务于Vue.component用来生成组件，理解为在模板中遇到该组件名称作为标签的自定义元素时，  会自动调用“扩展实例构造器”来生产组件实例，并挂载到自定义元素上,  但是他没有传递任何参数，只是个静态标签。  \*/  *var* Profile = Vue.extend({  template:'<p @click="change">{{msg}}</p>',  data:*function*(){  return {  msg:'html'  }  },  methods:{  change: *function*() {  this.msg = 'javascript'  }  }  })  // 直接挂载时click不可以  new Profile().$mount('shi-peng');  new Profile().$mount('#shi-peng'); //挂载到id为shi-peng的div上  //全局组件 此时click可用  //Vue.component('shi-peng',Profile);  IMG_256 |

技术胖

|  |
| --- |
| {{message}}  <button @click="add">add</button>  // extends 只能拓展构造器里原生的方法  *var* bbb={  created:*function*(){  *console*.log("我是被扩展出来的"); // 执行  },  methods:{  add:*function*(){  *console*.log('我是被扩展出来的方法！'); // 不执行  }  }  };  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  methods:{  add:*function*(){  *console*.log('我是原生方法'); // 点击执行  }  },  extends:bbb, // 只能是对象或方法，不能是数组  delimiters:['${','}'] // ${num}  }) |

## Vue.nextTick

|  |
| --- |
| <https://www.cnblogs.com/first-time/p/7087830.html>  <https://www.cnblogs.com/xujiazheng/p/6852124.html>  <https://www.cnblogs.com/douyaer/p/7650140.html>  Vue.nextTick([callback,context])  在下次 DOM 更新循环结束之后执行延迟回调。  在修改数据之后立即使用这个方法，获取更新后的 DOM  <shi-peng></shi-peng>  *var* Profile = Vue.extend({  template:'<p >{{msg}}</p>',  data:*function*(){  return {  msg:'htmlxxxx'  }  },  created(){  //this.change();  },  beforeMount() {  //this.change();  },  mounted () {  //this.change();  },  beforeUpdate() {  this.change();  },  methods:{  change: *function*() {  this.msg = 'javascriptxxx';  *console*.log(this.$el.textContent); // => '没有更新'  //console.log(this.msg); // => '没有更新'  this.$nextTick(*function* () {  *console*.log(this.$el.textContent); // => '更新完成'  //console.log(this.msg); // => '更新完成'  })  }  }  }) |

## Vue.set

技术胖

|  |
| --- |
| <div id="app">  <p>{{count}}</p>  <ul>  <li v-for="value in arr">{{value}}</li>  </ul>  </div>  <button onclick="add()">外部添加</button>  *function* add() {  /\*  Vue.set(outData,'count',4);  app.count++;  outData.count++;  **使用外部outData时，以上3种方法都可以让基本类型修改成功**  \*/  **// 有以上其中一句都可以使**app.arr[1]='ddd' **成功，因为触发了虚拟dom更新**  **// 只有该句时，使用下标修改数组的值时，不成功，js的限制**  //app.arr[1]='ddd';  Vue.set(app.arr,1,'ddd'); // 成功  }  // data在外部声明  *var* outData={  count:1,  arr:['aaa','bbb','ccc']  };  *var* app=new Vue({  el:'#app',  data:outData //引用外部数据  }) |

案例

|  |
| --- |
| <http://jspang.com/2017/03/14/vue2_02/#3vueset>  <https://cn.vuejs.org/v2/guide/reactivity.html>  Vue.set(target,key,value)普通的情况下对Vue实例里面的数据进行更改，数据改掉了，  但是呈现在页面的视图并没有发生变化，所以借用该方法视图也会跟着刷新    <li v-for="item in arr">{{item}}</li>  var app=new Vue({  el:"#app",  data:{  arr:['北京','上海','天津','重庆']  }  });  app.arr[0]='北京天安门';// 不会刷新  Vue.set(app.arr,0,'北京天安门'); // set方式视图会刷新 |

## Vue.delete

|  |
| --- |
| <https://blog.csdn.net/sinat_17775997/article/details/75950335>  <https://segmentfault.com/a/1190000011512960>  Vue.delete(target,key)用法和原理与set添加元素是一样的道理  *var* data = {  msg:'xcssdsds',  arr:['aaa','bbb','ccc']  }  Vue.delete(data, 'msg');  Vue.delete(data.arr, 0);  *var* vm = new Vue({  el: '#app',  data: data,  }); |

## Vue.directive自定义指令-属性指令

|  |
| --- |
| 作用：验证数据  <span v-color>HTML</span>  <span v-color="'red'">HTML</span> <!-- 传入的是字符串，双引号包裹单引号 -->  // 自定义指令color要添加前缀v-color,my-color => v-my-color  Vue.directive('color',*function*(*el*,*binding*){  binding.value = binding.value ? binding.value : 'blue';  el.style.background = binding.value;  });  // 第二参数是对象  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）  Vue.directive('color',{  bind:*function*(*el*,*binding*,*vnode*,*oldVnode*) {  binding.value = binding.value ? binding.value : 'blue';  el.style.background = binding.value;  *console*.log('bind',el,binding,vnode,oldVnode);  },  inserted:*function*(*el*,*binding*) {  *console*.log('inserted',el,binding);  },  update:*function*() {  *console*.log('update');  },  componentUpdated:*function*() {  *console*.log('componentUpdated');  },  unbind:*function*() {  *console*.log('unbind');  },  }); |

技术胖directive

|  |
| --- |
| <div id="app">  <div v-mydirective="color" id="aaa">{{num}}</div>  <button @click='jia'>加分</button>  <button onclick='unbind()'>解绑</button>  </div>  *function* unbind(){  app.$destroy();  }  //自定义指令  Vue.directive('mydirective',{  bind:*function*(*el*,*binding*,*vnode*){//被绑定  /\*\*  var s=JSON.stringify;  el.innerHTML =  'name:' + s(binding.name) +'<br>' +  'value:' + s(binding.value) +'<br>' +  'expression:' + s(binding.expression) +'<br>' ;  \*\*/  el.style='color:'+binding.value;  *console*.log('1 - bind');  },  inserted:*function*(){//绑定到节点  *console*.log('2 - inserted');  },  update:*function*(){//组件更新  *console*.log('3 - update'); // 元素的值更新或dom更新是调用  },  componentUpdated:*function*(){//组件更新完成  *console*.log('4 - componentUpdated');  },  unbind:*function*(){//解绑  *console*.log('5 - unbind');  }  })  *var* app=new Vue({  el:'#app',  data:{  color:'green',  num:10  },  methods:{  jia:*function*(){  this.num++;  }  }  }) |

## Vue.directive自定义指令-元素指令

|  |
| --- |
| <style>  my-red {  *width*: 100px;  *height*: 100px;  *background*: blue;  *display*:block;  }  </style>  <my-red></my-red>  // vue1.0注册或取得一个全局自定义元素指令  Vue.elementDirective('my-red',{ //不支持function语法  bind:*function*(){  this.el.style.background='black'; //注释该句为blue  }  }) |

## Vue.directive自定义指令-拖拽

|  |
| --- |
| <div v-drag :style="{width:'100px',height:'100px',background:'blue',  position:'absolute',right:0,top:0}"></div>  <div v-drag :style="{width:'100px', height:'100px', background:'red',  position:'absolute', left:0, top:0}"></div>  Vue.directive('drag',*function*(*el*){  el.onmousedown = *function*(*ev*){  *var* disX = ev.clientX-el.offsetLeft;  *var* disY = ev.clientY-el.offsetTop;  document.onmousemove=*function*(*ev*){  *var* l=ev.clientX-disX;  *var* t=ev.clientY-disY;  el.style.left=l+'px';  el.style.top=t+'px';  };  document.onmouseup=*function*(){  document.onmousemove=null;  document.onmouseup=null;  };  };  }); |

## Vue.filter过滤器vue1.0

### 常用过滤器

|  |
| --- |
| 常用过滤器  {{'welcome'|uppercase}} 转大写  {{'WELCOME'|lowercase}} 转小写  {{'WELCOME'|lowercase|capitalize}} 转小写,再首字母大写  {{12|currency}} $12  {{12|currency '￥'}} ￥12  {{json | json}} 等价于JSON.stringify |

### debounce

|  |
| --- |
| debounce 配合事件，延迟执行，lodash \_.debounce(fn,时间)  <input type="text" @keyup="show | debounce 2000">  methods:{  show:function(){  console.log(123);  }  } |

### limitBy

|  |
| --- |
| limitBy数据配合使用过滤器: limitBy  限制几个 limitBy 取几个 从哪开始  filterBy 过滤数据  orderBy 排序 1/-1 1 -> 正序 -1 -> 倒序 按照名字排  <li v-for="val in arr | limitBy 2">{{val}}</li>  <li v-for="val in arr | limitBy 2 arr.length-2">{{val}}</li>  data:{  arr:[1,2,3,4,5]  } |

### filterBy

|  |
| --- |
| filterBy  <input type="text" v-model="char">  <li v-for="val in arr | filterBy char">{{val}}</li>  data:{  arr:['width','height','background','orange'],  char:''  } |

### orderBy

|  |
| --- |
| orderBy  <input type="text" v-model="char"/>  <li v-for="val in arr | orderBy char">{{val}}</li> 排序  <li v-for="val in arr | orderBy -1">{{val}}</li>  <li v-for="val in arr | orderBy 'o'">{{val}}</li>  data:{  arr:['width','height','background','orange'],  char:''  } |

## Vue.filter过滤器vue2.0

|  |
| --- |
| 过滤器用在两个地方：**mustache 插值和 v-bind表达式**, 不能用在v-model、v-html、v-text  为了在其他指令中实现更复杂的数据变换，应该使用计算属性 |

### 自定义过滤器

|  |
| --- |
| ***model ->过滤 -> view***  vue2.0，删除全部内置过滤器,推荐使用工具库lodash \_.debounce(fn,200)  {{msg}} // 2.0可以直接解析json  data:{msg:{a:1,b:2,c:3} }  自定义过滤器——还保留，但是,自定义过滤器传参改变  之前: {{msg | toDou '12' '5'}}  现在: {{msg | toDou('12','5')}}  {{num | toDou(1,2)}}  Vue.filter('toDou',*function*(*num*,*a*,*b*){  return num<10? '0'+num:''+num; //转化成字符串  })  new Vue({  data:{  num:90  }  }).$mount('#app'); // vm.$mount('#box'); //手动挂载 |

### 时间转化器

|  |
| --- |
| {{num | date}}  Vue.filter('date',*function*(*input*){  *var* oDate = new Date(input);  return oDate.getFullYear()+'-'+(oDate.getMonth()+1)+'-'+oDate.getDate()+  ' '+oDate.getHours()+':'+oDate.getMinutes()+':'+oDate.getSeconds();  });  data:{  num:*Date*.now() //1970 年 1 月 1日午夜与当前日期和时间之间的毫秒数  }, |

### 过滤html标记

|  |
| --- |
| {{msg | fiterHtml}}  Vue.filter('fiterHtml',*function*(*input*){  return input.replace(/<[^<]+>/g,'');  });  data:{  msg:'<strong>welcome</strong>'  } |

### 双向过滤器vue1.0

|  |
| --- |
| {{{msg | filterHtml}}}  Vue.filter('filterHtml',{  read:*function*(*input*){ //model-view  return input.replace(/<[^<]+>/g,'');  },  write:*function*(*val*){ //view -> model  *console*.log(val);  return val;  }  });  data:{  msg:'<strong>welcome</strong>'  }, |

## Vue.component

### 全局组件

|  |
| --- |
| <aaa></aaa>  var Aaa = Vue.extend({  data(){ **//必须是函数形式,函数必须返回一个对象(json)**  return { msg:'刘亦菲' }  },  methods:{  **change**(){ this.**msg** = '大魔王'}  },  template:'<h3 @click="change">{{msg}}</h3>'  // template:'<h3>刘亦菲</h3>' });  Vue.**component**('aaa',Aaa); //全局组件  var a = new Aaa(); **console**.log(Aaa); // function VueComponent(options){...} **console**.log(a); // VueComponent{...}  =========================================================================================  <my-aaa></my-aaa>  Vue.**component**('my-aaa',{ //全局组件  **template**:'<h3 @click="change">{{msg}}</h3>',  data(){  return { msg:'刘亦菲'}  },  methods:{  **change**() {this.**msg**='大魔王'; }  } }) |

### 局部组件

|  |
| --- |
| <aaa></aaa>  <my-aaa></my-aaa>  var Aaa = Vue.extend({  **template**:'<h3>{{msg}}</h3>',  data(){  **// data必须是函数的形式，函数必须返回一个对象(json)**  return {  msg:'刘亦菲'  }  } });  var vm=new Vue({  **el**:'#app',  **components**:{ //局部组件  **aaa**:Aaa,  **'my-aaa'**: Aaa ***// 有 – 连接时，是字符串***  }  });  =============================================================================  <my-aaa></my-aaa>  var vm=new Vue({  **el**:'#box',  components:{ //局部组件  'my-aaa':{  data(){  return { msg:'welcome vue'}  },  methods:{  **change**(){ this.**msg**='changed'; }  },  template:'<h2 @click="change">标题2->{{msg}}</h2>'  }  } }); |

|  |
| --- |
| <world></world>  *var* city={  template:`<div>this is city!</div>`  }  *var* world={  template:`<div>  <p>局部注册的- &lt;world&gt;</p>  <city></city>  </div>`,  components:{  "city":city  }  }  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  components:{  "world":world  }  }) |

### 组件模板

|  |
| --- |
| 模板定义有4种：  script标签模板  template 标签模板  在构造器的template属性里定义：template:` <h1>我是选项模板</h1>`  .vue文件里定义 |

|  |
| --- |
| <my-aaa></my-aaa>  <!--script标签模板-->  <script type="x-template" id="aaa" src="demo.js"></script> 引入外部文件，形成模板  <script type="x-template" id="aaa">   <h2 @click="change">{{msg}}</h2>  <ul>  <li v-for="val in arr">{{val}}</li>  </ul> </script>  ***或***  <!--template 标签模板 -->  <**template** id="aaa">  <h2 @click="change">{{msg}}</h2>  <ul>  <li v-for="val in arr">{{val}}</li>  </ul> </**template**>  new Vue({  **el**:"#app",  components:{  'my-aaa':{  data(){  return {  msg:'刘亦菲',  arr:['apple','banana','orange']  }  },  methods:{  **change**(){  this.**msg** = '大魔王';  }  },  template:"#aaa"  }  } }) |

### vue2.0-组件模板

|  |
| --- |
| <div id="app">  <my-aaa></my-aaa>  </div>  <!--  vue2.0必须包含一个根标签,vue1.0可以不包含根标签，在每个组件模板，不在支持片段代码  否则报错:Component template should contain exactly one root element  -->  <template id="aaa">  <div>  <h3>刘亦菲</h3>  <strong>大魔王</strong>  </div> </template>  Vue.**component**('my-aaa',{  //template:'<h3>刘亦菲</h3><strong>大魔王</strong>'  **template**:'#aaa' }); |

### vue2.0-组件定义

|  |
| --- |
| Vue.extend 这种方式，在2.0里面有，但是有一些改动，这种写法，即使能用，咱也不用——废弃  <div id="app">  <my-aaa></my-aaa> </div> <template id="aaa">  <div>  <h3>刘亦菲</h3>  <strong>大魔王</strong>  </div> </template>  var Home= {  **template**:"#aaa",  }; Vue.component('my-aaa',Home); // 全局组件  new Vue({  **el**:'#app',  **components**:{  **'my-aaa'**:Home //局部组件  }  })  =========================================================  <hello></hello>  <helloworld></helloworld>  Vue.component('hello',{  template:`<div>全局化注册的：&lt;hello&gt;标签</div>`  });  *var* helloworld = { template:`<div>局部注册的：&lt;helloworld&gt;标签</div>`}  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  components:{  "helloworld":helloworld  }  }) |

### 动态组件

|  |
| --- |
| <input type="button" @click="a='aaa'" value="aaa组件"/> <input type="button" @click="a='bbb'" value="bbb组件"/> <component :is="a"></component>  new Vue({  **el**:"#app",  **data**:{  **a**:'aaa'  },  **components**:{  **'aaa'**:{  **template**:'<h2>我是aaa组件</h2>'  },  **'bbb'**:{  **template**:'<h2>我是bbb组件</h2>'  }  } }) |

|  |
| --- |
| <component v-bind:is="who"></component>  <button @click="changeComponent">changeComponent</button>  *var* componentA={  template:`<div style="color:red;">I'm componentA</div>`  }  *var* componentB={  template:`<div style="color:green;">I'm componentB</div>`  }  *var* componentC={  template:`<div style="color:pink;">I'm componentC</div>`  }  *var* app=new Vue({  el:'#app',  data:{  who:'componentA'  },  components:{  "componentA":componentA,  "componentB":componentB,  "componentC":componentC,  },  methods:{  changeComponent:*function*(){  if(this.who=='componentA'){  this.who='componentB';  }else if(this.who=='componentB'){  this.who='componentC';  }else{  this.who='componentA';  }  }  }  }) |

## Vue.use

|  |
| --- |
| 1.采用ES6的import ... from ...语法或CommonJSd的require()方法引入插件  2.使用全局方法Vue.use( plugin )使用插件,可以传入一个选项对象Vue.use(MyPlugin, { someOption: true }) |

## Vue.mixin

|  |
| --- |
| {{ message }}  // 混入2个用途，  // 1.构造器写好了，项目写好了，临时需求，不想污染现有的构造器  // 2.公用的方法  //全局混入  Vue.mixin({  created:*function*(){  *console*.log('我是全局被混入的');  }  })  //局部混入  *var* aaa={  created:*function*(){  *console*.log('我是外部被混入的');  }  }  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  created:*function*(){  *console*.log('我是原生的');  },  mixins:[aaa] // 只能是数组  }) |

案例

|  |
| --- |
| <http://jspang.com/2017/03/26/vue3/5/>  // 全局混入  Vue.mixin({  created(){  this.change1();  },  beforeMount() {  this.change2();  },  mounted () {  this.change3();  },  beforeUpdate() {  this.change4();  },  data: *function* () {  return {  message: 'hello',  foo: 'abc',  arr:['a','b','c']  }  },  methods:{  hello: *function*() {  *console*.log('这是混合器添加的hello方法');  },  lsit:*function*(){  *console*.log('这是混合器添加的lsit方法')  },  say: *function*() {  *console*.log('这是混合器添加的say方法')  },  change1: *function*() {  *console*.log('change1');  },  change2: *function*() {  *console*.log('change2');  },  change3: *function*() {  *console*.log('change3');  },  change4: *function*() {  *console*.log('change4');  }  }  })  // 注册组件 返回的Profile 是个构造函数  *var* Profile = Vue.extend({  template:'<p >{{msg}}</p>',  data:*function*(){  return {  message: 'xhello',  foo: 'xabc',  arr:['xa','xb','xc']  }  },  created(){  this.change1();  },  beforeMount() {  this.change2();  },  mounted () {  this.change3();  },  beforeUpdate() {  this.change4();  },  methods:{  change1: *function*() {  *console*.log('xchange1');  },  change2: *function*() {  *console*.log('xchange2');  },  change3: *function*() {  *console*.log('xchange3');  },  change4: *function*() {  *console*.log('xchange4');  }  }  })  Vue.component('shi-peng',Profile);  // 在Profile 的原型对象上定义myapp  *Profile* .prototype.myapp = *function*() {  *console*.log('我是外来入侵者');  }  *console*.log(Profile);  *console*.log(new Profile());  *console*.log(Profile.\_\_proto\_\_);  /\*  全局混入的执行顺序要前于混入和构造器里的方法。  Profile 实例对象上有就调用实例对象上的，没有才去mixin里找  \*/  IMG_256  IMG_256 |

## Vue.compile

## Vue.observable

## Vue.version

# 选项 / 数据

## data

## props

## propsData

|  |
| --- |
| <h1>Vue.extend的propsData 用得很少，仅用于开发环境</h1>  <div id="head2"></div>  *var* CommonHeader = Vue.extend({  template:"<header>我是网站头部 {{ message }} -by {{ username }}</header>",  data:*function* () {  return{  message:"I am message!"  }  },  props:['username']  });  // 向Vue.extend传参propsData属性,实际开发中我们使用的并不多  *var* header = new CommonHeader({  propsData:{  username:"wss"  }  }).$mount("#head2"); |

## computed计算属性

|  |
| --- |
| 需求：num1的值变化，num2的值跟着变化，用计算属性  {{num1}} {{num2}}  data:{  num1:10  },  computed:{  num2:*function*(){ // 是属性，只是写法是函数，不用事件触发就执行,默认调用get  return this.num1+20; //num2的值取决于返回值 业务逻辑代码  },  // 或  num2:{ //一开始只执行get方法，get给num2赋值，以后num2值改变时，先set再get  set:*function*(*val*){  this.num1 = val;  },  get:*function*(){  return this.num1 + 20;  }  }  }  document.onclick = *function*(){  vm.num1 = 100;  } |

技术胖

|  |
| --- |
| <p>{{newPrice}}</p>  <li v-for="item in reverseNews">{{item.title}}-{{item.date}}</li>  *var* newsList=[  {title:'香港或就“装甲车被扣”事件追责 起诉涉事运输公司',date:'2017/3/10'},  {title:'日本第二大准航母服役 外媒：针对中国潜艇',date:'2017/3/12'},  {title:'中国北方将有明显雨雪降温天气 南方阴雨持续',date:'2017/3/13'},  {title:'起底“最短命副市长”：不到40天落马，全家被查',date:'2017/3/23'},  ];  *var* app=new Vue({  el:'#app',  data:{  price:100,  newsList:newsList  },  computed:{  newPrice:*function*(){  return this.price='￥' + this.price + '元';  },  reverseNews:*function*(){  return this.newsList.reverse();  }  }  }) |

## methods

|  |
| --- |
| <div id="app">  {{a}}  <p><button @click="add(2)">add</button></p>  <!-- 构造器里的组件上的事件需添加native -->  <btn @click.native="add(2)"></btn>  </div>  <button onclick="app.add(3)">外部访问构造器里的方法</button>  *var* btn={  template:'<button>外部组件</button>'  }  *var* app=new Vue({  el:'#app',  data:{  a:1  },  methods:{  add:*function*(*num*){  return this.a+=num;  }  },  components:{  "btn":btn  }  }) |

## watch

|  |
| --- |
| <p>今日温度：{{temperature}}°C</p>  <p>穿衣建议:{{this.suggestion}}</p>  <p>  <button @click="add">添加温度</button>  <button @click="reduce">减少温度</button>  </p>  *var* suggestion=['T恤短袖','夹克长裙','棉衣羽绒服'];  *var* app=new Vue({  el:'#app',  data:{  temperature:14,  suggestion:'夹克长裙'  },  methods:{  add:*function*(){  this.temperature+=5;  },  reduce:*function*(){  this.temperature-=5;  }  },  watch:{  temperature:*function*(*newVal*,*oldVal*){  if(newVal>=26) {  this.suggestion=suggestion[0];  }else if(newVal<26 && newVal >=0) {  this.suggestion=suggestion[1];  }else{  this.suggestion=suggestion[2];  }  }  }  })  // 全局的  app.$watch('temperature',*function*(*newVal*,*oldVal*){  if(newVal>=26){  this.suggestion=suggestion[0];  }else if(newVal<26 && newVal >=0) {  this.suggestion=suggestion[1];  }else{  this.suggestion=suggestion[2];  }  }) |

# 选项/其它

## name

## delimiters

|  |
| --- |
| <hello></hello>  <world></world>  Vue.component('hello',{  template:`<div>{this.$parent.message}我是全局化注册的-&lt;hello&gt;</div>`,  //替换插值符号  delimiters:['{','}']  })  *var* world={  template:'<div>{{this.$parent.message}}局部注册的-&lt;world&gt</div>'  }  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  components:{  "world":world  }  }) |

## functional

## model

## inheritAttrs

## comments

# vue实例属性

|  |
| --- |
| vue实例属性可以在new Vue() 构造函数外访问其属性和方法  <https://cn.vuejs.org/v2/api/#实例属性>  注意：vm.$options 获取自定义属性  *var* vm = new Vue({  el:"#app",  num :12, // 自定义属性  show:*function*(){ // 自定义方法  *console*.log(123);  }  })  vm.$el.style.background='red'; // vm.$el等价于 el  vm.$options.show(); //123  *console*.log(vm.$options.num); // 12 |

# vue实例方法/数据

## vm.$watch监听数据变化

|  |
| --- |
| 根据  <input type="text" v-model="num">  <input type="text" v-model="json.name">  <div>{{num}} </div>  <div>{{json}}</div>  data:{  num:10,  json:{name:'HTML',age:18}  },  vm.$watch('num',*function*(*newVal*,*oldVal*){  *console*.log(newVal,oldVal);  })  vm.$watch('json',*function*(*newJson*,*oldJson*){  *console*.log(newJson);//等价于oldJson  *console*.log(oldJson);  },{deep:true}); //深度监视 |

## vm.$set

## vm.$delete

# 实例方法/事件

## vm.$on

在构造器的外部增加构造器内部的事件，可以在构造器外部调用内部的数据

|  |
| --- |
| <div id="app">  <p>{{num}}</p>  <button @click="add">add</button>  </div>  <p><button onclick="reduce()">reduce</button></p>  <p><button onclick="reduceOnce()">reduceOnce</button></p>  <p><button onclick="off()">off</button></p>  *var* app=new Vue({  el:'#app',  data:{num:1},  methods:{  add:*function*(){  this.num++;  }  }  })  //实例事件  app.$on('reduce',*function*(){  *console*.log('执行了reduce()');  this.num--;  });  //只使用一次的实例方法  app.$once('reduceOnce',*function*(){  *console*.log('只执行一次的方法');  this.num--;  });  //关闭事件  *function* off(){  app.$off('reduce');  }  //外部调用内部事件  *function* reduce(){  app.$emit('reduce');  }  *function* reduceOnce(){  app.$emit('reduceOnce');  } |

## vm.$once

## vm.$off

## vm.$emit

# 实例方法/生命周期

## vm.$mount

|  |
| --- |
| new Vue({  data:{msg:'vue'}  }).$mount('#app'); // 手动挂载 |

|  |
| --- |
| 使用构造器后挂载  <div id="app"></div>  <p><button onclick="destroy()">卸载</button></p>  <p><button onclick="reload()">刷新</button></p>  <p><button onclick="tick()">修改</button></p>  *var* jspang = Vue.extend({  template:`<p>{{message}}</p>`,  data:*function*(){  return {  message:'Hello ,I am JSPang'  }  },  destroyed:*function*(){  *console*.log('10-destroyed 销毁之后');  },  mounted:*function*(){  *console*.log("4-mounted 被创建");  },  updated:*function*(){  *console*.log("6-updated 被更新后");  }  })  *var* vm = new jspang().$mount("#app"); // 会打印4-mounted 被创建"，**说明在mounted钩子里挂载的**  *function* destroy(){  vm.$destroy(); // destroyed**钩子执行**  }  *function* reload(){  vm.$forceUpdate(); // 更新/刷新方法， updated**钩子执行**  }  *function* tick(){  // 修改数据  vm.message="update message info ";  // DOM 还没有更新  vm.$nextTick(*function*(){ // 是个回调，dom更新后执行  // DOM 现在更新了  // `this` 绑定到当前实例  *console*.log('message更新完后我被调用了');  })  } |

## vm.$forceUpdate

## vm.$nextTick

## vm.$destroy

|  |
| --- |
| vm.$destroy(); //解绑 |

# transition vue1.0

|  |
| --- |
| #div{  *width*:100px;  *height*:100px;  *background*: red;   }  .fade-transition{*transition*: 1s all ease;}  .fade-enter{*opacity*: 0;} /\*进入 \*/  .fade-leave{ /\*离开\*/  *opacity*: 0;  *transform*: translateX(200px);  }  <input type="button" value="按钮" @click="toggle">  <div id="div" v-show="bSign" transition="fade"></div>  toggle(){      this.bSign=!this.bSign;  }  ===========================================  <link rel="stylesheet" href="bower\_components/animate.css/animate.css">  #div{  *width*:100px;  *height*:100px;  *background*: red;  }  <input type="button" value="按钮" @click="toggle">  <div id="div" class="animated" v-show="bSign" transition="bounce"></div>  data:{bSign:true},  methods:{      toggle(){this.bSign=!this.bSign;}  },  transitions:{ //定义所有动画名称      bounce:{             enterClass:'zoomInLeft',             leaveClass:'zoomOutRight'       }  } |

# transition vue2.0

|  |
| --- |
| .fade-enter{} //初始状态  .fade-enter-active{} //变化成什么样 -> 当元素出来(显示)  .fade-leave{}  .fade-leave-active{} //变成成什么样 -> 当元素离开(消失)  //点击显示和隐藏  p{  *width*:300px;  *height*:300px;  *background*: red;  }  .fade-enter-active, .fade-leave-active{  *transition*: 1s all ease;  }  .fade-enter-active{  *opacity*:1;  *width*:300px;  *height*:300px;  }  .fade-leave-active{  *opacity*:0;  *width*:100px;  *height*:100px;  }  .fade-enter,.fade-leave{  *opacity*:0;  *width*:100px;  *height*:100px;  }  <input type="button" value="点击显示隐藏" @click="show=!show">  <transition name="fade">  <p v-show="show"></p>  </transition> |

# transition配合animate

|  |
| --- |
| p{  *width*:100px;  *height*:100px;  *background*: red;  *margin*:0 auto;  }  <link rel="stylesheet" href="animate.css">  <input type="button" value="点击显示隐藏" @click="show=!show">  <transition enter-active-class="animated zoomInLeft"  leave-active-class="animated zoomOutRight">  <p v-show="show"></p>  </transition>  ==================================================  <input type="button" value="点击显示隐藏" @click="show=!show">  <transition name="transition">  <p v-show="show" class="animated"></p>  </transition>  data:{  show:false  },  transitions:{ //错的  enterActiveClass:'zoomInLeft',  leaveActiveClass:'zoomOutRight'  } |

# transition vue2.0-多个元素

|  |
| --- |
| p{  *width*:100px;  *height*:100px;  *background*: red;  *margin*:10px auto;  }  <link rel="stylesheet" href="animate.css">  <input type="text" v-model="show">  <transition-group enter-active-class="zoomInLeft"  leave-active-class="zoomOutRight">  <p v-show="show" class="animated" v-for="(val,index) in lists" :key="index">  {{val}}  </p>  </transition-group>  data:{  show:'',  list:['apple','banana','orange','pear']  },  computed:{  lists:*function*(){  *var* arr=[];  this.list.forEach(*function*(*val*){  if(val.indexOf(this.show)!=-1){  arr.push(val);  }  }.bind(this));  return arr;  }  } |

# transition vue2.0相关函数

|  |
| --- |
| p{  *width*:300px;  *height*:300px;  *background*: red;  }  .fade-enter-active, .fade-leave-active{  *transition*: 1s all ease;  }  .fade-enter-active{  *opacity*:1;  *width*:300px;  *height*:300px;  }  .fade-leave-active{  *opacity*:0;  *width*:100px;  *height*:100px;  }  .fade-enter,.fade-leave{  *opacity*:0;  *width*:100px;  *height*:100px;  }  <input type="button" value="点击显示隐藏" @click="show=!show">  <transition name="fade"  @before-enter="beforeEnter"  @enter="enter"  @after-enter="afterEnter"  @before-leave="beforeLeave"  @leave="leave"  @after-leave="afterLeave">  <p v-show="show"></p>  </transition>  data:{  show:false  },  methods:{  beforeEnter(*el*){  *console*.log('动画enter之前');  },  enter(*el*){  *console*.log('动画enter进入');  },  afterEnter(*el*){  *console*.log('动画进入之后');  el.style.background='blue';  },  beforeLeave(*el*){  *console*.log('动画leave之前');  },  leave(*el*){  *console*.log('动画leave');  },  afterLeave(*el*){  *console*.log('动画leave之后');  el.style.background='red';  }  } |

# 路由配合运动

|  |
| --- |
| .router-link-active{  *font-size*: 20px;  *color*:#f60;  }  <transition enter-active-class="animated bounceInLeft"  leave-active-class="animated bounceOutRight">  <router-view></router-view>  </transition>  router.push({path:'home'});  router.replace({path:'user'}); |

# 父子组件

**vm.$dispatch(事件名,数据) 子级向父级发送数据**

**vm.$broadcast(事件名,数据) 父级向子级广播数据**

**配合: event:{}，但在vue2.0里面已经，报废了**

**vue默认情况下，子组件没法访问父组件数据，子组件获取父组件data**

|  |
| --- |
| <aaa></aaa>  new Vue({  **el**:"#app",  components:{  'aaa':{  data(){  return{ msg:'父组件数据', info:123 }  },  // 给子组件绑定一个属性，指向父组件的msg，在子组件通过props属性，  template:'<h2>aaa组件{{msg}}</h2><bbb :mmm="msg" :my-msg="info"></bbb>',  **components**:{  'bbb':{  // props:{ 'mmm':String, 'myMsg':Number },  props:['mmm','myMsg'],  template:'<h3>bbb组件-->{{mmm}}哈{{myMsg}}</h3>'  }  }  }  }  }) |

**父组件获取子组件data**

|  |
| --- |
| ***子组件把自己的数据，发送到父级,在父组件通过 @child-msg事件名接收***  ***vm.$emit(事件名,数据);***  <div id="box">  <aaa></aaa> </div>  <template id="aaa">  <span>我是父级 -> {{msg}}</span>  <bbb @child-msg="get"></bbb> </template> <template id="bbb">  <h3>子组件-</h3>  <input type="button" value="send" @click="send"> </template>  var vm=new Vue({  **el**:'#box',  components:{  'aaa':{  data(){  return { msg:'父组件的数据'}  },  template:'#aaa',  methods:{  get(**msg**){ this.**msg**=**msg**; }  },  components:{  'bbb':{  data(){  return {a:'子组件的数据'}  },  template:'#bbb',  methods:{  send(){  this.$emit('child-msg',this.**a**);//把子组件的a数据发送给父组件  }  }  }  }  }  } }); |

# vue2.0父子组件

|  |
| --- |
| vm.$emit() vm.$on();  vue1.0，子组件加sync可以同步更改父组件信息， vue2.0，不管是否有sync,都不允许子组件直接改父级的数据，做赋值操作  vue2.0想更改父组件数据  a). 父组件每次传一个对象给子组件, 对象之间引用 √  b). 只是不报错, mounted中转  <script src="vue1.0.js"></script>  <div id="box">  父级: ->{{a}}  <br>  <child-com :msg.sync="a"></child-com> </div>  <template id="child">  <div>  <span>我是子组件</span>  <input type="button" value="按钮" @click="change">  <strong>{{msg}}</strong>  </div>  </template>  new Vue({  **el**:'#box',  **data**:{ **a**:'我是父组件数据'},  components:{  'child-com':{  **props**:['msg'],  **template**:'#child',  methods:{  change(){  this.**msg**='被更改了'  }  }  }  } }); |

vue2.0如下同步父组件数据: a). 父组件每次传一个对象给子组件, 对象之间引用 √

|  |
| --- |
| <div id="box">  父级: ->{{giveData.a}}  <br>  <child-com :msg="giveData"></child-com> </div>  <template id="child">  <div>  <span>我是子组件</span>  <input type="button" value="按钮" @click="change">  <strong>{{msg.a}}</strong>  </div>  </template>  new Vue({  **el**:'#box',  **data**:{  **giveData**:{**a**:'我是父组件数据'}  },  components:{  'child-com':{  **props**:['msg'],  **template**:'#child',  methods:{  change(){  this.**msg**.**a**='被改了';  }  }  }  } }); |

vue2.0如下同步父组件数据: **b). 只是不报错, mounted中转**

|  |
| --- |
| <div id="box">  父级: ->{{giveData.a}}  <br>  <child-com :msg="giveData"></child-com> </div> <template id="child">  <div>  <span>我是子组件</span>  <input type="button" value="按钮" @click="change">  <strong>{{b.a}}</strong>  </div> </template>  new Vue({  **el**:'#box',  **data**:{  **giveData**:{ **a**:'我是父组件数据'}  },  components:{  'child-com':{  **props**:['msg'],  **template**:'#child',  data(){ return{ **b**:'' } },  mounted(){ this.**b**=this.**msg**;},  methods:{  **change**(){  this.**b**.**a**='被改了'; // 等价于this.msg.a  }  }  }  } });  ***如下是否加sync都不能同步***  <div id="box">  父级: ->{{a}}  <br>  <child-com :msg.sync="a"></child-com> </div> <template id="child">  <div>  <span>我是子组件</span>  <input type="button" value="按钮" @click="change">  <strong>{{b}}</strong> <!--b子组件的数据-->  </div>  </template>  new Vue({  **el**:'#box',  **data**:{ **a**:'我是父组件数据'},  components:{  'child-com':{  **props**:['msg'],  **template**:'#child',  data(){ return { **b**:''} },  mounted(){ this.**b**=this.**msg**;},  methods:{  **change**(){ this.**b**='被改了';}  }  }  } }); |

# 单一事件中心管理组件通信

|  |
| --- |
| Event.$emit(事件名称, 数据)  Event.$on(事件名称,function(data){ //data }.bind(this));  <div id="box">  <com-a></com-a>  <com-b></com-b>  <com-c></com-c>  </div>  //准备一个空的实例对象  *var* Event=new Vue();  *var* A={  template:`  <div>  <span>我是A组件</span> -> {{a}}  <input type="button" value="把A数据给C" @click="send">  </div>`,  methods:{  send(){  Event.$emit('a-msg',this.a); // 把数据给别人  }  },  data(){  return {a:'我是a数据'}  }  };  *var* B={  template:`  <div>  <span>我是B组件</span> -> {{a}}  <input type="button" value="把B数据给C" @click="send">  </div>`,  methods:{  send(){  Event.$emit('b-msg',this.a);  }  },  data(){  return {  a:'我是b数据'  }  }  };  *var* C={  template:`  <div>  <h3>我是C组件</h3>  <span>接收过来的A的数据为: {{a}}</span>  <br>  <span>接收过来的B的数据为: {{b}}</span>  </div>`,  data(){  return {  a:'',  b:''  }  },  mounted(){  //var \_this=this;  //接收A组件的数据  Event.$on('a-msg',*function*(*a*){  this.a=a;  }.bind(this));  //接收B组件的数据  Event.$on('b-msg',*function*(*a*){  this.b=a;  }.bind(this));  }  }; |

# vue全局组件use使用

|  |
| --- |
| Loading.vue 写组件代码  index.js  import LoadingComponent from './Loading.vue'  *const* Loading = {    install: *function*(*Vue*) {      Vue.component('Loading', LoadingComponent)    }  };  export default Loading  main.js 引入  import Loading from './components/loading'  Vue.use(Loading)  App.vue 使用  <Loading></Loading> |

# element-ui

|  |
| --- |
| element-ui.js 按需引入  // 按钮和单选....  import {Button,Radio,DatePicker,Rate,Pagination} from 'element-ui'  Vue.use(Button);  Vue.use(Radio);  // tabs  import {TableColumn,Table,Switch,Badge,TabPane,Tabs} from 'element-ui'  Vue.use(Badge);  Vue.use(Table);  main.js // 直接引入  import './element-ui.js' |

## el-scrollbar

|  |
| --- |
| <https://juejin.im/post/5c83d5ac5188257e1c4dc9e7>  <https://juejin.im/post/5b0c0adb6fb9a009ec7e9734>  <div style="width:300px;height:500px;">  <el-scrollbar>  <div style="height: 1500px;width: 100%;background:#ccc;"></div>  </el-scrollbar>  </div>  .el-scrollbar {  *height*: 100%;  }  .el-scrollbar\_\_wrap {  *overflow-x*:hidden;  }  IMG_256 |

# 面试题

<https://segmentfault.com/a/1190000016344599#articleHeader2>

## Vue与Angular以及React的区别

|  |
| --- |
| **1.与AngularJs的区别** 相同点：都支持指令，内置指令和自定义指令；都支持过滤器：内置过滤器和自定义过滤器；都支持双向数据绑定，都不支持低端浏览器；  不同点：AngularJs的学习成本高，比如Dependency、Injecttion特性，而vue.js本身提供的API都比较简单、直观；在性能上，AngularJs依赖对数据做脏检查，所以watcher越多越慢；Vue.js使用基于依赖追踪的观察并且使用异步队列更新，所以的数据都是独立触发的。  **2. 与React的区别** 相同点： React采用独特的JSX语法，Vue.js 在组件开发中也推崇编写.vue特殊文件格式，对文件内容都有一些约定，两者都需要编译后使用;中心实现相同：一切都是组件，组件实例之间可以嵌套；都提供合理的钩子函数，可以让开发者定制化底去处理需求，都不内置列数Ajax，Route等功能到核心包，而是以插件的方式加载;在组件开发中都支持mixins的特性了；  不同点： React采用Virtual DOM会对渲染出来的结果做脏检查；Vue.js在模板中提供了指令，过滤器等，可以非常方便，快捷的操作Virtual DOM。 |

## vue

### MVVM

|  |
| --- |
| MVVM 是 Model-View-ViewModel 的缩写。  Model代表数据模型，也可以在Model中定义数据修改和操作的业务逻辑。  View 代表UI 组件，它负责将数据模型转化成UI 展现出来。  ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步View 和 Model的对象，连接Model和View。  在MVVM架构下，View 和 Model 之间并没有直接的联系，而是通过ViewModel进行交互，Model 和 ViewModel 之间的交互是双向的， 因此View 数据的变化会同步到Model中，而Model 数据的变化也会立即反应到View 上。  ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而View 和 Model 之间的同步工作完全是自动的，无需人为干涉，因此开发者只需关注业务逻辑，不需要手动操作DOM, 不需要关注数据状态的同步问题，复杂的数据状态维护完全由 MVVM 来统一管理。 |

### mvc和mvvm

|  |
| --- |
| MVVM原理：  MVVM是一种软件架构模式，MVVM有助于前后端分离。  View:视图层，粗略理解为DOM。  Model:与数据库对应的model，一般为json格式，作为req的body通过http(s)与数据库实现通信。  ViewModel:View与Model通过ViewModel实现双向绑定。  核心是提供对View和ViewModel的双向数据绑定，这样使得ViewModel的改变View立即变化，MVVM在前端的实现有：angular，vue，react。  vue中的常用数据双向绑定。  view:{{message}}  viewModel v-model="message"  model:message  <div id="app-6">  <p>{{ message }}</p>  <input v-model="message">  </div>  var app6 = new Vue({  el: '#app-6',  data: {  message: 'Hello Vue!'  }  })  单文件组件中的话，就多了一个用html5的template标签将view和viewModel包裹起来，model部分停留在script标签部分。  <template>  view  viewModel  </tamplate>  <script>  model  </script>  <styles>  为了让view好看点  </styles>  react的话，我在使用的过程中，没有听说过双向绑定的东西，对redux reducers推荐写为纯函数印象深刻，纯函数的话，感觉应该有点单项数据流的意思。  既然说到框架了，说一个最让我感觉有趣的点，那就是组件间的通信，对于简单组件，只涉及父子级别的通信的，vue使用on emit的方式，react使用props。对于复杂级别通信，爷爷父亲儿子孙子等等时，vue推荐使用vuex，react推荐使用redux，统一的全局状态树用来做状态管理非常好，可以使得逻辑非常清晰。vue项目文件结构研究不深，react的项目文件结构的话，presentational和containers的设计方法感觉非常有道理，一个负责视图一个负责数据，非常清爽。 |

|  |
| --- |
| mvc和mvvm其实区别并不大。都是一种设计思想。主要就是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。和当 Model 频繁发生变化，开发者需要主动更新到View 。 |

### 什么是vue生命周期

|  |
| --- |
| Vue 生面周期总结的文章: <https://jkchao.cn/article/59d6e93c7e2ee06d412efef9>  答： Vue 实例从创建到销毁的过程，就是生命周期。也就是从开始创建、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，称这是 Vue 的生命周期 |

### vue生命周期的作用是什么

|  |
| --- |
| 生命周期中有多个事件钩子，让我们在控制整个Vue实例的过程时更容易形成好的逻辑 |

### 第一次页面加载会触发哪几个钩子

|  |
| --- |
| 会触发 下面这几个beforeCreate, created, beforeMount, mounted |

### DOM 渲染在哪个周期中就已经完成

|  |
| --- |
| DOM 渲染在 mounted 中就已经完成了 |

### vue生命周期(10个)

|  |
| --- |
| IMG_256  beforeCreate created beforeMount mounted beforeUpdate updated beforeDestroy destroyed  activated: keep-alive 组件激活时调用  deactivated: keep-alive 组件停用时调用  errorCaptured：当捕获一个来自子孙组件的错误时被调用  总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后  beforeCreate 创建前执行（vue实例的挂载元素$el和数据对象data都为undefined，还未初始化）  created 完成创建 （完成了data数据初始化，el还未初始化）  beforeMount 载入前（vue实例的$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。）  mounted 载入后html已经渲染(vue实例挂载完成，data.message成功渲染。)  beforeUpdate 更新前状态（view层的数据变化前，不是data中的数据改变前）  updated 更新状态后  beforeDestroy 销毁前  destroyed 销毁后 （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）  说一下每一个阶段可以做的事情   * beforeCreate:可以在这里加一个loading事件，在加载实例时触发。 * created:初始化完成时的事件写这里，如果这里结束了loading事件，异步请求也在这里调用。 * mounted:挂在元素，获取到DOM节点 * updated:对数据进行处理的函数写这里。 * beforeDestroy:可以写一个确认停止事件的确认框。     ------------   **created:** 实例已经创建完成之后调用,在这一步,实例已经完成数据观测, 属性和方法的运算, watch/event事件回调. 然而, 挂载阶段还没有开始, $el属性目前还不可见   **mounted:** el被新创建的 vm.$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.$el 也在文档内。   **activated:**:keep-alive组件激活时调用  。  ---------  **创建前/后：** 在beforeCreated阶段，vue实例的挂载元素$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  ---------  beforeCreate（创建前） 在数据观测和初始化事件还未开始  created（创建后） 完成数据观测，属性和方法的运算，初始化事件，$el属性还没有显示出来  beforeMount（载入前） 在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。注意此时还没有挂载html到页面上。  mounted（载入后） 在el 被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html页面中。此过程中进行ajax交互。  beforeUpdate（更新前） 在数据更新之前调用，发生在虚拟DOM重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。  updated（更新后） 在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。  beforeDestroy（销毁前） 在实例销毁之前调用。实例仍然完全可用。  destroyed（销毁后） 在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。  ===================================================  每个Vue实例在被创建时都要经过一系列的初始化过程(设置数据监听、编译模板、将实例挂载到DOM并在数据变化时更新DOM等。  初始化过程中会运行生命周期钩子的函数，可以在不同阶段添加的代码  生命周期钩子的this上下文指向调用它的Vue实例，固不要在选项属性或回调上使用箭头函数，如下  // 因为箭头函数是和父级上下文绑定在一起的，this不会是如你所预期的Vue 实例  // 经常导致 Uncaught TypeError: Cannot read property of undefined 或  // Uncaught TypeError: this.myMethod is not a function 之类的错误  created: () *=>* *console*.log(this.a) 或  vm.$watch('a', *newValue* *=>* this.myMethod()) |

|  |
| --- |
| <div id="app">  <input type="button" value="更新数据" @click="update">  <input type="button" value="销毁组件" @click="destroy">  </div>  <button onclick="app.$destroy()">销毁</button>  {{msg}}  new Vue({  el:'#app',  data:{  msg:'welcome vue2.0'  },  methods:{  update(){  this.msg='大家好';  },  destroy(){  this.$destroy();  }  },  beforeCreate(){  *console*.log('组件实例刚刚被创建,属性都没有');  },  created(){  *console*.log('实例已经创建完成，属性已经绑定');  },  beforeMount(){  *console*.log('模板编译之前');  },  mounted(){  *console*.log('模板编译完成，代替之前ready ');  },  beforeUpdate(){  *console*.log('组件更新之前，数据更新前');  },  updated(){  *console*.log('组件更新完毕，被更新后');  },  activated:*function*(){  *console*.log('activated',keep-alive 组件激活时调用);  },  deactivated:*function*(){  *console*.log('deactivated',keep-alive 组件停用时调用);  },  beforeDestroy(){  *console*.log('组件销毁之前');  },  destroyed(){  *console*.log('组件销毁之后');  }  }); |

### vue组件通信

|  |
| --- |
| 父-> 子: props  父组件通过绑定属性把数据传递给子组件，子组件通过props选项接收数据，  props可以接受function，也可以做些类型检查  <Child-one :parentMessage="parentMessage"></Child-one> 父  props: ['parentMessage'], 子  ---------------------------------------------------  子-> 父: on+emit  子组件通过$emit自定义事件并传递数据，父组件使用v-on监听子组件上的自定义事件接收数据  mounted() {  this.$emit('childEvent', { name: 'zhangsan', age: 10 }); // 子  },  // 父  <Child-one @childEvent="parentMethod"></Child-one>  methods: {  parentMethod({ *name*, *age* }) {  *console*.log(this.parentMessage, name, age);  }  }  父<->子: on.sync(语法糖)来的  ---------------------------------------------------  其他：不推荐  因为vue提倡单向数据流，只有父组件交给子组件的数据子组件才有使用的权限  this.$parent、this.$refs 不属于数据的传递而是一种主动的查找，利用了父子组件关系传递数据  父->子：this.$parent， this为子组件实例 获取父组件然后使用父组件中的数据  子->父：可以通过在子组件添加ref属性，然后可以通过ref属性名称获取到子组件的实例  <Child-one ref="child"></Child-one> // 父  // 子  mounted(){  *console*.log(this.$refs['child']);  },  ---------------------------------------------------  兄弟组件之间的通信  单一事件中心管理: EventBus，就是创建一个事件中心，相当于中转站，可以用它来传递事件和接受事件  let vm = new Vue(); // 实例化一个vue实例，相当于一个第三方  组件他哥  <div @click="ge"></div>  methods: {  ge() {  vm.$emit('blur','sichaoyun'); //触发事件  }  }  组件小弟接受大哥命令  <div></div>  created() {  vm.$on('blur', (*arg*) *=>* {  this.test= arg; // 接收  });  }  ---------------------------------------------------  VueX建一个数据仓库，整个项目全局都可以往这个仓库存放数据和读取数据  ---------------------------------------------------  其他不限于直接父子级 localstorage sessionstorage cookies |

### Proxy 相比于 defineProperty 的优势

|  |
| --- |
| *Proxy*：数组变化也能监听到，不需要深度遍历监听  *let* data = { a: 1 }  *let* reactiveData = new Proxy(data, {    get: *function*(*target*, *name*){      // ...    },    // ...  }) |

### vue响应式原理(数据劫持)Object.defineProperty

|  |
| --- |
| 1.Object.defineProperty(obj, prop, descriptor)  2.obj 是要在其上定义属性的对象；prop 是要定义或修改的属性的名称；descriptor 是将被定义或修改的属性描述符。  比较核心的是 descriptor，它有很多可选键值，具体的可以去参阅它的文档。  这里我们最关心的是 get 和 set，get 是一个给属性提供的 getter 方法，当我们访问了该属性的时候会触发 getter 方法；set 是一个给属性提供的 setter 方法，当我们对该属性做修改的时候会触发 setter 方法。  一旦对象拥有了 getter 和 setter，我们可以简单地把这个对象称为响应式对象  a.对象递归调用  b.数组变异方法的解决方法：代理原型/实例方法  3.observe  4.observe 方法的作用就是给非 VNode 的对象类型数据添加一个 Observer，如果已经添加过则直接返回，否则在满足一定条件下去实例化一个 Observer 对象实例。  5.observe 的功能就是用来监测数据的变化.  6.Observer 是一个类，它的作用是给对象的属性添加 getter 和 setter，用于依赖收集和派发更新：  7.依赖收集和派发更新  8.收集依赖的目的是为了当这些响应式数据发生变化，触发它们的 setter 的时候，能知道应该通知哪些订阅者去做相应的逻辑处理，我们把这个过程叫派发更新，其实 Watcher 和 Dep 就是一个非常经典的观察者设计模式的实现  9.派发更新就是数据发生变化的时候，触发 setter 逻辑，把在依赖过程中订阅的的所有观察者，也就是 watcher，都触发它们的 update 过程，这个过程又利用了队列做了进一步优化，在 nextTick 后执行所有 watcher 的 run，最后执行它们的回调函数  10.vue编译Compile的过程主要分以下几步 parse(生成AST)*=>* optimize(优化静态节点) *=>* generate(生成render *function*)  // 解析模板字符串生成 AST  const ast = parse(template.trim(), options)  //优化语法树  optimize(*ast*, *options*)  //生成代码  const code = generate(*ast*, *options*) |

|  |
| --- |
| 看完生命周期后，里面的watcher等内容其实是数据响应中的一部分。  数据响应的实现由两部分构成: **观察者( watcher )** 和 **依赖收集器( Dep )**，其核心是 defineProperty这个方法，它可以 **重写属性的 get 与 set** 方法，从而完成监听数据的改变   Observe (观察者)观察 props 与 state   * 遍历 props 与 state，对每个属性创建独立的监听器( watcher )    使用 defineProperty 重写每个属性的 get/set(defineReactive）   * get: 收集依赖   + Dep.depend()     - watcher.addDep() * set: 派发更新   + Dep.notify()   + watcher.update()   + queenWatcher()   + nextTick   + flushScheduleQueue   + watcher.run()   + updateComponent()   大家可以先看下面的数据相应的代码实现后，理解后就比较容易看懂上面的简单脉络了。  let data = {a: 1}  // 数据响应性  observe(data)  // 初始化观察者  new Watcher(data, 'name', updateComponent)  data.a = 2  // 简单表示用于数据更新后的操作  function updateComponent() {  vm.\_update() // patchs  }  // 监视对象  function observe(obj) {  // 遍历对象，使用 get/set 重新定义对象的每个属性值  Object.keys(obj).map(key => {  defineReactive(obj, key, obj[key])  })  }  function defineReactive(obj, k, v) {  // 递归子属性  if (type(v) == 'object') observe(v)    // 新建依赖收集器  let dep = new Dep()  // 定义get/set  Object.defineProperty(obj, k, {  enumerable: true,  configurable: true,  get: function reactiveGetter() {  // 当有获取该属性时，证明依赖于该对象，因此被添加进收集器中  if (Dep.target) {  dep.addSub(Dep.target)  }  return v  },  // 重新设置值时，触发收集器的通知机制  set: function reactiveSetter(nV) {  v = nV  dep.nofify()  },  })  }  // 依赖收集器  class Dep {  constructor() {  this.subs = []  }  addSub(sub) {  this.subs.push(sub)  }  notify() {  this.subs.map(sub => {  sub.update()  })  }  }  Dep.target = null  // 观察者  class Watcher {  constructor(obj, key, cb) {  Dep.target = this  this.cb = cb  this.obj = obj  this.key = key  this.value = obj[key]  Dep.target = null  }  addDep(Dep) {  Dep.addSub(this)  }  update() {  this.value = this.obj[this.key]  this.cb(this.value)  }  before() {  callHook('beforeUpdate')  }  } |

### vue2.0双向数据绑定原理 或者vue的响应式原理

|  |
| --- |
| 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**。** |

-----------

|  |
| --- |
| <https://juejin.im/post/5acc17cb51882555745a03f8#heading-1>  vue实现数据双向绑定主要是：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，触发相应监听回调。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。用户看不到 getter/setter，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。  vue的数据双向绑定 将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的数据变化，通过Compile来解析编译模板指令（vue中是用来解析 {{}}），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变更双向绑定效果。  <div id="app">  <input type="text" id="txt">  <p id="show"></p>  </div>  *let* obj = {}  *Object*.defineProperty(obj, 'txt', {  get: *function* () {  return obj  },  set: *function* (*newValue*) {  document.getElementById('txt').value = newValue  document.getElementById('show').innerHTML = newValue  }  })  document.addEventListener('keyup', *function* (*e*) {  obj.txt = e.target.value  })  --------------------  vue的双向绑定的原理: Object.defineProperty + Dep + Watcher  --------  用来为input输入框或者单选框、select选择框这类的东西做双向绑定的。  使用栗子：<input v-model="inputData"/>  -------------  原理：利用了 Object.defineProperty() 这个方法重新定义了对象获取属性值(get)和设置属性值(set)的操作来实现的。 缺点：双向数据流是自动管理状态的, 但是在实际应用中会有很多不得不手动处理状态变化的逻辑, 使得程序复杂度上升, 难以调试。  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。  //vue实现数据双向绑定的原理就是用Object.defineproperty()重新定义（set方法）对象设置属性值和（get方法）获取属性值的操纵来实现的。  //Object.property()方法的解释：Object.property(参数1，参数2，参数3) 返回值为该对象obj  //其中参数1为该对象（obj），参数2为要定义或修改的对象的属性名，参数3为属性描述符，属性描述符是一个对象，主要有两种形式：数据描述符和存取描述符。这两种对象只能选择一种使用，不能混合使用。而get和set属于存取描述符对象的属性。  //这个方法会直接在一个对象上定义一个新属性或者修改对象上的现有属性，并返回该对象。  /\* html  <div id="myapp">  <input v-model="message" /><br>  <span v-bind="message"></span>  </div>  \*/  *var* model = {  message: ""  };  *var* models = myapp.querySelectorAll("[v-model=message]");  for (*var* i = 0; i < models.length; i++) {  models[i].onkeyup = *function*() {  model[this.getAttribute("v-model")] = this.value;  }  }  // 观察者模式 / 钩子函数  // defineProperty 来定义一个对象的某个属性  *Object*.defineProperty(model, "message", {  set: *function*(*newValue*) {  *var* binds = myapp.querySelectorAll("[v-bind=message]");  for (*var* i = 0; i < binds.length; i++) {  binds[i].innerHTML = newValue;  };  *var* models = myapp.querySelectorAll("[v-model=message]");  for (*var* i = 0; i < models.length; i++) {  models[i].value = newValue;  };  this.value = newValue;  },  get: *function*() {  return this.value;  }  })  Vue3.0将用原生Proxy替换Object.defineProperty  为什么要替换Object.defineProperty？  1.在Vue中，*Object*.defineProperty无法监控到数组下标的变化，导致直接通过数组的下标给数组设置值，不能实时响应。  2.Object.defineProperty只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历。  什么是Proxy  1.Proxy是 ES6 中新增的一个特性，翻译过来意思是"代理"，用在这里表示由它来“代理”某些操作。 *Proxy* 让我们能够以简洁易懂的方式控制外部对对象的访问。其功能非常类似于设计模式中的代理模式。  2.Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。  3.使用 *Proxy* 的核心优点是可以交由它来处理一些非核心逻辑（如：读取或设置对象的某些属性前记录日志；  设置对象的某些属性值前，需要验证；某些属性的访问控制等）。 从而可以让对象只需关注于核心逻辑，达到关注点分离，降低对象复杂度等目的。  ========================================  Vue 的响应式原理是使用 *Object*.defineProperty 追踪依赖，当属性被访问或改变时通知变化。  有两个不足之处：  不能检测到增加或删除的属性。  数组方面的变动，如根据索引改变元素，以及直接改变数组长度时的变化，不能被检测到。  原因差不多，无非就是没有被 getter/setter 。  第一个比较容易理解，为什么数组长度不能被 getter/setter ？  因为：如果你知道数组的长度，理论上是可以预先给所有的索引设置 getter/setter 的。但是一来很多场景下你不知道数组的长度，二来，如果是很大的数组，预先加 getter/setter 性能负担较大。  *Proxy*：在目标对象之前架设一层拦截 |

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

|  |
| --- |
| // <input id="input"/>  // <div id="txt"></div>  *const* data = {};  *const* input = document.getElementById('input');  *const* txt = document.getElementById('txt');  *Object*.defineProperty(data, 'text', {  set(*value*) {  input.value = value;  this.value = value;  txt.innerHTML = this.value;  }  });  // onchange  input.oninput = *function*(*e*) {  data.text = e.target.value;  }  =================================================  /\*  <div id="app">  <input type="text" id="txt">  <p id="show-txt"></p>  </div>  \*/  *var* obj = {}  *Object*.defineProperty(obj, 'txt', {  get: *function* () {  return obj  },  set: *function* (*newValue*) {  document.getElementById('txt').value = newValue  document.getElementById('show-txt').innerHTML = newValue  }  })  document.addEventListener('keyup', *function* (*e*) {  obj.txt = e.target.value  }) |

### virtual dom实现原理

|  |
| --- |
| 其实 VNode 是对真实 DOM 的一种抽象描述，它的核心定义无非就几个关键属性，标签名、数据、子节点、键值等，其它属性都是都是用来扩展 VNode 的灵活性以及实现一些特殊 feature 的。由于 VNode 只是用来映射到真实 DOM 的渲染，不需要包含操作 DOM 的方法，因此它是非常轻量和简单的。 Virtual DOM 除了它的数据结构的定义，映射到真实的 DOM 实际上要经历 VNode 的 create（用JS对象模拟DOM树）、diff（比较两棵虚拟DOM树的差异）、patch（把差异应用到真正的DOM树上） 等过程。  -----------------------  + 创建 dom 树  + 树的`diff`，同层对比，输出`patchs(listDiff/diffChildren/diffProps)`  + 没有新的节点，返回  + 新的节点`tagName`与`key`不变， 对比`props`，继续递归遍历子树  + 对比属性(对比新旧属性列表):  + 旧属性是否存在与新属性列表中  + 都存在的是否有变化  + 是否出现旧列表中没有的新属性  + tagName和key值变化了，则直接替换成新节点  + 渲染差异  + 遍历`patchs`， 把需要更改的节点取出来  + 局部更新`dom`  ```js  // diff算法的实现  function diff(oldTree, newTree) {  // 差异收集  let pathchs = {}  dfs(oldTree, newTree, 0, pathchs)  return pathchs  }  function dfs(oldNode, newNode, index, pathchs) {  let curPathchs = []  if (newNode) {  // 当新旧节点的 tagName 和 key 值完全一致时  if (oldNode.tagName === newNode.tagName && oldNode.key === newNode.key) {  // 继续比对属性差异  let props = diffProps(oldNode.props, newNode.props)  curPathchs.push({ type: 'changeProps', props })  // 递归进入下一层级的比较  diffChildrens(oldNode.children, newNode.children, index, pathchs)  } else {  // 当 tagName 或者 key 修改了后，表示已经是全新节点，无需再比  curPathchs.push({ type: 'replaceNode', node: newNode })  }  }  // 构建出整颗差异树  if (curPathchs.length) {  if(pathchs[index]){  pathchs[index] = pathchs[index].concat(curPathchs)  } else {  pathchs[index] = curPathchs  }  }  }  // 属性对比实现  function diffProps(oldProps, newProps) {  let propsPathchs = []  // 遍历新旧属性列表  // 查找删除项  // 查找修改项  // 查找新增项  forin(olaProps, (k, v) => {  if (!newProps.hasOwnProperty(k)) {  propsPathchs.push({ type: 'remove', prop: k })  } else {  if (v !== newProps[k]) {  propsPathchs.push({ type: 'change', prop: k , value: newProps[k] })  }  }  })  forin(newProps, (k, v) => {  if (!oldProps.hasOwnProperty(k)) {  propsPathchs.push({ type: 'add', prop: k, value: v })  }  })  return propsPathchs  }  // 对比子级差异  function diffChildrens(oldChild, newChild, index, pathchs) {  // 标记子级的删除/新增/移动  let { change, list } = diffList(oldChild, newChild, index, pathchs)  if (change.length) {  if (pathchs[index]) {  pathchs[index] = pathchs[index].concat(change)  } else {  pathchs[index] = change  }  }  // 根据 key 获取原本匹配的节点，进一步递归从头开始对比  oldChild.map((item, i) => {  let keyIndex = list.indexOf(item.key)  if (keyIndex) {  let node = newChild[keyIndex]  // 进一步递归对比  dfs(item, node, index, pathchs)  }  })  }  // 列表对比，主要也是根据 key 值查找匹配项  // 对比出新旧列表的新增/删除/移动  function diffList(oldList, newList, index, pathchs) {  let change = []  let list = []  const newKeys = getKey(newList)  oldList.map(v => {  if (newKeys.indexOf(v.key) > -1) {  list.push(v.key)  } else {  list.push(null)  }  })  // 标记删除  for (let i = list.length - 1; i>= 0; i--) {  if (!list[i]) {  list.splice(i, 1)  change.push({ type: 'remove', index: i })  }  }  // 标记新增和移动  newList.map((item, i) => {  const key = item.key  const index = list.indexOf(key)  if (index === -1 || key == null) {  // 新增  change.push({ type: 'add', node: item, index: i })  list.splice(i, 0, key)  } else {  // 移动  if (index !== i) {  change.push({  type: 'move',  form: index,  to: i,  })  move(list, index, i)  }  }  })  return { change, list }  } |

### diff算法

|  |
| --- |
| 详解vue的diff算法 <https://juejin.im/post/5affd01551882542c83301da>  1.diff算法比较新旧节点的时候，比较只会在同层级比较，不会跨层级比较  2.当数据发生变化的时候会生成一个新的VNode，然后新VNode和oldNode做对比，发现不一样的地方直接修改在真实的dom上，比较新旧节点，一边比较一边给真是的dom打补丁  3.节点设置key可以高效的利用dom（key最好不要设置成index索引）  4.虚拟DOM diff算法主要就是对以下三种场景进行优化：    1.tree diff  对树进行分层比较，两棵树只会对同一层次的节点进行比较。(因为 DOM 节点跨层级的移动操作少到可以忽略不计)  如果父节点已经不存在，则该节点及其子节点会被完全删除掉，不会用于进一步的比较。  注意：  React 官方建议不要进行 DOM 节点跨层级的操作，非常影响 React 性能。  在开发组件时，保持稳定的 DOM 结构会有助于性能的提升。例如，可以通过 CSS 隐藏或显示节点，而不是真的移除或添加 DOM 节点。  2.component diff  如果是同一类型的组件，按照原策略继续比较 virtual DOM tree（tree diff）。  对于同一类型的组件，有可能其 Virtual DOM 没有任何变化，如果能够确切的知道这点那可以节省大量的 diff 运算时间，因此 React 允许用户通过 shouldComponentUpdate() 来判断该组件是否需要进行 diff。  如果不是，直接替换整个组件下的所有子节点。  3.element diff  对处于同一层级的节点进行对比。  这时 React 建议：添加唯一 key 进行区分。虽然只是小小的改动，性能上却发生了翻天覆地的变化！  如： A B C D --> B A D C  添加 key 之前： 发现 B != A，则创建并插入 B 至新集合，删除老集合 A；以此类推，创建并插入 A、D 和 C，删除 B、C 和 D。  添加 key 之后： B、D 不做任何操作，A、C 进行移动操作，即可。  建议：在开发过程中，尽量减少类似将最后一个节点移动到列表首部的操作，当节点数量过大或更新操作过于频繁时，在一定程度上会影响 React 的渲染性能。  4.总结  1.React 通过制定大胆的 diff 策略，将 O(n3) 复杂度的问题转换成 O(n) 复杂度的问题；  2.React 通过分层求异的策略，对 tree diff 进行算法优化；  3.React 通过相同类生成相似树形结构，不同类生成不同树形结构的策略，对 component diff 进行算法优化；  4.React 通过设置唯一 key的策略，对 element diff 进行算法优化；  5.建议，在开发组件时，保持稳定的 DOM 结构会有助于性能的提升；  6.建议，在开发过程中，尽量减少类似将最后一个节点移动到列表首部的操作，当节点数量过大或更新操作过于频繁时，在一定程度上会影响 React 的渲染性能。 |

### v-if和v-show的相同点和区别

|  |
| --- |
| 相同点：都是用来做条件渲染，控制元素的显示与隐藏。    v-if 是"真正"的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。  v-if 也是惰性的：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。  v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。  一般来说，v-if 有更高的切换开销，而 v-show 有更高的初始渲染开销。  因此，如果需要非常频繁地切换，则使用 v-show 较好；(tabs切换)  如果在运行时条件很少改变，则使用 v-if 较好。(页面初始化时，根据条件显示/隐藏，而后不需切换了) |

### watch 和computed区别

|  |
| --- |
| 1.computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。  2.watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。  3.所以一般来说需要依赖别的属性来动态获得值的时候可以使用computed，对于监听到值的变化需要做一些复杂业务逻辑的情况可以使用 watch。 |

### 实现 vue 中的 on,emit,off,once，手写代码

|  |
| --- |
| // 参照 vue 源码实现  *var* EventEmiter = *function* (){  this.\_events = {};  };  *EventEmiter*.prototype.on = *function* (*event*, *cb*){  if (*Array*.isArray(event)){  for (*let* i = 0, l = event.length; i < l; i++){  this.on(event[i], cb);  }  } else {  (this.\_events[event] || (this.\_events[event] = [])).push(cb);  }  return this;  };  *EventEmiter*.prototype.once = *function* (*event*, *cb*){  *function* on () {  this.off(event, cb);  cb.apply(this, arguments);  }  on.fn = cb;  this.on(event, on);  return this;  };  *EventEmiter*.prototype.off = *function* (*event*, *cb*){  if (!arguments.length){  this.\_events = *Object*.create(null);  return this;  }  if (*Array*.isArray(event)){  for (*let* i = 0, l = event.length; i < l; i++){  this.off(event[i],cb);  }  return this;  }  if (!cb){  this.\_events[event] = null;  return this;  }  if (cb){  *let* cbs = this.\_events[event];  *let* i = cbs.length;  while(i--){  if (cb === cbs[i] || cb === cbs[i].fn){  cbs.splice(i, 1);  break;  }  }  return this;  }  };  *EventEmiter*.prototype.emit = *function* (*event*){  *let* cbs = this.\_events[event];  *let* args = *Array*.prototype.slice.call(arguments, 1);  if (cbs){  for (*let* i = 0, l = cbs.length; i < l; i++){  cbs[i].apply(this,args);  }  }  }; |

### vue 组件data为什么必须是函数

|  |
| --- |
| 每个组件都是 Vue 的实例。  组件共享 data 属性，当 data 的值是同一个引用类型的值时，改变其中一个会影响其他。 |

### <keep-alive></keep-alice>的作用的是什么

|  |
| --- |
| <keep-alive>是Vue的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染DOM。  属性：  include( 包含的组件缓存）  exclude(排除的组件不缓存，优先级大于include）  max:数字。最多可以缓存多少组件实例  首先检查组件自身的 name 选项，如果 name 选项不可用，则匹配它的局部注册名称  <!-- 逗号分隔字符串，只有组件a与b被缓存。 -->  <keep-alive include="a,b">  <component></component>  </keep-alive>  <!-- 正则表达式 (需要使用 v-bind，符合匹配规则的都会被缓存) -->  <keep-alive :include="/a|b/">  <component></component>  </keep-alive>  <!-- Array (需要使用 v-bind，被包含的都会被缓存) -->  <keep-alive :include="['a', 'b']">  <component></component>  </keep-alive> |

### 订阅者-发布者模式和观察者模式有区别吗？

|  |
| --- |
|  |

### nextTick

|  |
| --- |
| nextTick : 更新数据后立即操作dom  在下次dom更新循环结束之后执行延迟回调，可用于获取更新后的dom状态  新版本中默认是microtasks, v-on中会使用macrotasks  macrotasks任务的实现:   * + setImmediate / MessageChannel / setTimeout |

### el 和 $mount 区别

|  |
| --- |
| v-el:提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 HTMLElement 实例, |

### vue 的挂载怎么实现的

|  |
| --- |
|  |

### 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 |

### Vue.js的template编译的理解

|  |
| --- |
| 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） |

### vue如何实现按需加载配合webpack设置?

|  |
| --- |
| webpack中提供了require.ensure()来实现按需加载。以前引入路由是通过import 这样的方式引入，改为const定义的方式进行引入。  不进行页面按需加载引入方式：import home from '../../common/home.vue'  进行页面按需加载的引入方式：const home = r => require.ensure( [], () => r (require('../../common/home.vue'))) |

### 路由懒加载

|  |
| --- |
| <https://router.vuejs.org/zh/guide/advanced/lazy-loading.html#路由懒加载> |

### 为什么使用key

|  |
| --- |
| 当有相同标签名的元素切换时，需要通过 key 特性设置唯一的值来标记以让 Vue 区分它们，否则 Vue 为了效率只会替换相同标签内部的内容。 |

### axios、fetch和ajax有什么区别？

|  |
| --- |
|  |

## vue-router

<https://jspang.com/post/vue-router.html>

<https://router.vuejs.org/zh/guide/#html>

|  |
| --- |
| vue路由依赖：vue-router  通过组合组件来组成单页应用程序，只需要将组件映射到路由即可。  前端路由的核心，就在于 —— 改变视图的同时不会向后端发出请求。  $route 当前路由(html/js里调用)this.$route.params.username  $router 路由器(整个应用,js里调用) this.$router.go(-1)  router路由  routes 路线 |

### vue-router如何做用户登录权限等

|  |
| --- |
|  |

### vue-router有哪几种导航钩子

|  |
| --- |
| 三种  全局导航钩子（跳转前进行判断拦截）  beforeEach: 全局前置守卫  beforeResolve: 全局解析守卫  afterEach: 全局后置钩子  beforeRouteEnter  beforeRouteUpdate  beforeRouteLeave  beforeEnter  to : route即将进入的目标路由对象、  from： route当前导航正要离开的路由、  next： function一定要调用该方法resolve这个钩子。执行效果依赖next方法的调用参数。可以控制网页的跳转  --------------------------------------------------------  *const* router = new VueRouter({})  router.beforeEach((*to*, *from*, *next*) *=>* {/\* must call `next` \*/})  router.beforeResolve((*to*, *from*, *next*) *=>* {/\* must call `next` \*/})  router.afterEach((*to*, *from*) *=>* { })  组件内钩子  *const* Foo = {  template: `...`,  beforeRouteEnter(*to*, *from*, *next*) {  // 在渲染该组件的对应路由被 confirm 前调用  // 不！能！获取组件实例 `this`  // 因为当守卫执行前，组件实例还没被创建  },  beforeRouteUpdate(*to*, *from*, *next*) {  // 在当前路由改变，但是该组件被复用时调用  // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，  // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。  // 可以访问组件实例 `this`  },  beforeRouteLeave(*to*, *from*, *next*) {  // 导航离开该组件的对应路由时调用  // 可以访问组件实例 `this`  }  }  单独路由独享组件  *const* router = new VueRouter({  routes: [{  path: '/foo',  component: Foo,  beforeEnter: (*to*, *from*, *next*) *=>* {// ...}  }  ]  }) |

### vue-router的hash 和 history 区别，也是vue-router实现原理

|  |
| --- |
| 利用pushState开发无刷页面切换:<https://www.cnblogs.com/flash3d/p/3384823.html>  router有3种模式 : "hash" | "history" | "abstract"  Router({mode: 'hash'})  hash模式: window对象提供的onhashchange事件监听location.hash值的改变,一旦url中的hash值发生改变,便会触发该事件,达到不刷新跳转的效果.    history模式：用了 HTML5的 historyAPI 来实现,  a. popstate监听历史栈信息变化,变化时重新渲染  b. 使用pushState方法实现添加功能  c. 使用replaceState实现替换功能  hash可以理解为锚点，例如./index.html/#/foo，hash值为#/foo，不会跳转页面。  就相当于统一页面的不同锚点，页面间跳转与 ./index.html/#foo到./index.html/#bar类似。  abstract模式: 支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式。    hash模式url里面永远带着#号，我们在开发当中默认使用这个模式。那么什么时候要用history模式呢？  如果用户考虑url的规范那么就需要使用history模式，因为history模式没有#号，是个正常的url适合推广宣传。当然其功能也有区别，比如我们在开发app的时候有分享页面，那么这个分享出去的页面就是用vue或是react做的，  咱们把这个页面分享到第三方的app里，有的app里面url是不允许带有#号的，所以要将#号去除那么就要使用history模式，  但是使用history模式还有一个问题就是，在访问二级页面的时候，做刷新操作，会出现404错误，那么就需要和后端人配合让他配置一下apache或是nginx的url重定向，重定向到你的首页路由上就ok啦。 |

### vue-router 跳转和 location.href 有什么区别

|  |
| --- |
| router 是hash改变 ,不刷新页面  location.href 是页面跳转，刷新页面 |

### vue-router设置动态路由和获取参数

|  |
| --- |
| 设置：{path: '/home/:name', component: Hmoe}  获取：this.$route.params.name |

### vue-router跳转方式

|  |
| --- |
| <router-link to="/home">about</router-link>  this.$router.go(1)  this.$router.push('/home') 跳转到该页面，有历史记录  this.$router.replace('/home') 替换当前页面，无历史记录 |

### this.$route.query

|  |
| --- |
| this.$route.query.name 访问查询参数 |

### this.$route.params.name

|  |
| --- |
| this.$route.params.name |

### router-link

|  |
| --- |
|  |

### router-view

|  |
| --- |
| <router-view name="home"></router-view> 占位 |

### active-class是哪个组件的属性？嵌套路由怎么定义？

|  |
| --- |
| vue-router模块的router-link组件。  children嵌套路由 |

### scss是什么？安装使用的步骤是？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 |

### 什么是RESTful API？怎么使用

|  |
| --- |
| 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 |

### 前后端路由差别

|  |
| --- |
| 1.后端每次路由请求都是重新访问服务器  2.前端路由实际上只是JS根据URL来操作DOM元素，根据每个页面需要的去服务端请求数据，返回数据后和模板进行组合。 |

### vue-router传递参数

#### name属性传递参数

|  |
| --- |
| routes: [  {  path: '/',  name: 'HelloWorld',  component: HelloWorld  },  {  // 有子路由的在子路由里传递name  path:'/hi',  component:Hi,  children: [  {path:'/', name: hi,component: Hi1},  {path:'/hi1', name: 'hi1',component: Hi1},  ]  }  ]  接收{{$route.name}} |

#### 动态路由传参

|  |
| --- |
| <router-link :to="{name:'hi1',params:{username:'jspang', id:10}}">Hi页面</router-link>  {{$route.params.username}}  {{$route.params.id}} |

#### 查询参数传参

|  |
| --- |
| <router-link :to="{path: 'details', query: {name: '张三'}}">详情</router-link>  {{$route.query.name}} |

## Vuex

<https://jspang.com/post/vuex.html>

|  |
| --- |
| types.js  export *const* INCREMENT = 'INCREMENT';  export *const* DECREMENT = 'DECREMENT';  getters.js  export default {    count: (*state*) *=>* {      return state.count;    },    getOdd: (*state*) *=>* {      return state.count % 2 == 0 ? '偶数' : '奇数'    }  }  mutations.js  import {INCREMENT,DECREMENT} from './types'  import getters from './getters'  *const* state = {    count: 20  };  *const* mutations = { //处理状态(数据)变化    [INCREMENT](*state*) {      state.count++;    },    [DECREMENT](*state*) {      state.count--;    }  };  export default {    state,    mutations,    getters  }  actions.js  import \* as types from './types'  export default {    increment: ({ // 处理你要干什么，异步请求，判断，流程控制      commit    }) *=>* {      commit(types.INCREMENT);    },    decrement: ({      commit    }) *=>* {      commit(types.DECREMENT);    },    clickOdd: ({      commit,      state    }) *=>* {      if (state.mutations.count % 2 == 0) {        commit(types.INCREMENT);      }    },    clickAsync: ({      commit    }) *=>* {      new Promise((*resolve*) *=>* {        setTimeout(*function*() {          commit(types.INCREMENT);        }, 1000);      })    }  }  **index.js**  import Vue from 'vue'  import Vuex from 'vuex'  import mutations from './mutations'  import actions from './actions'  Vue.use(Vuex);  export default new Vuex.Store({    modules:{      mutations    },    actions  });  main.js  import Vue from 'vue'  import App from './App.vue'  import store from './store/'  new Vue({    store,    el: '#app',    render: *h* *=>* h(App)  })  App.vue  <h3>welcome vuex-demo</h3>  <input type="button" value="增加" @click="increment">  <input type="button" value="减少" @click="decrement">  <input type="button" value="偶数才能点击+" @click="clickOdd">  <input type="button" value="点击异步" @click="clickAsync">  <div>  现在数字为: {{count}}, 它现在是 {{getOdd}}  </div>  import {mapGetters, mapActions} from 'vuex'  export default{  computed:mapGetters([  'count',  'getOdd'  ]),  methods:mapActions([  'increment',  'decrement',  'clickOdd',  'clickAsync'  ])  } |

### vuex

|  |
| --- |
| store: 仓库 => 状态中心  $store全局单例对象，只要在main.js里注册就能直接使用  state: 状态 =>访问状态对象  mapState  getters: 获取，获得 => 对state里的数据输出前进行过滤，只有state有变化就触发(增加/删除/修改)  mapGetters  -----------------  mutations: 改变, 变化 => 同步改变state状态，属性键最好为全大写，如：SET\_SIZE  mapMutations  actions: 行动 =>  actions是异步的改变state状态，而mutations是同步改变state状态  actions是可以调用Mutations里的方法的  commit: 提交=>用于调用mutations对象里的方法  <el-button @click="$store.commit('add')">+</el-button>  -----------------  module: 模块 => 模块组,将state分成多个modules，便于管理 |

### vuex的作用

|  |
| --- |
| 作用：集中管理项目的数据状态，方便组件间数据通信，是单向数据流的。  数据的流向是是单向数据流,且数据并不具有持久化的特性(默认情况下刷新就重置所有状态)  至于单向数据流(全局单例模式)怎么理解？  state只能给mutation(同步操作) 改动, action只能反馈给mutation,可以进行异步操作(比如和后端交互拉取数据), state能触发render，action能用dispatch分发  --------------  1.在vue组件里面，通过dispatch来触发actions提交修改数据的操作。  2.然后再通过actions的commit来触发mutations来修改数据。  3.mutations接收到commit的请求，就会自动通过Mutate来修改state（数据中心里面的数据状态）里面的数据。  4.最后由store触发每一个调用它的组件的更新  IMG_256 |

### mutation和action区别

|  |
| --- |
| mutation只变更本地的状态，也就是说，直接只去修改store中的数据。  action包含异步操作，直接调用api，通过api的数据，再提交mutation。  可以说，action只比mutation多了一个异步调用api的操作，因为调用api后，  一般有2种返回结果，succes或者error，或者是promise的多种状态，根据不同的。 |

### 应用场景

|  |
| --- |
| 应用场景：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。 |

### vuex是什么

|  |
| --- |
| Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。  它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化 |

### 解决问题

|  |
| --- |
| 多个视图依赖于同一状态。  来自不同视图的行为需要变更同一状态  对于问题一，传参的方法对于多层嵌套的组件将会非常繁琐，并且对于兄弟组件间的状态传递无能为力。  对于问题二，我们经常会采用父子组件直接引用或者通过事件来变更和同步状态的多份拷贝。  以上的这些模式非常脆弱，通常会导致无法维护的代码 |

### Vuex 流程

|  |
| --- |
| IMG_256  Vuex把组件中共享的State抽离出来,任何组件都可以访问和操作其中的数据中心  用户在组件中的输入操作触发 action 调用；  Actions 通过分发 mutations 来修改 store 实例的状态；  Store 实例的状态变化反过来又通过 getters 被组件获知  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 |

### 父子传递数据

|  |
| --- |
| 1、父子组件直接引用  2、事件来变更和同步状态的多份拷贝 |

### vuex有哪几种属性

|  |
| --- |
| 有五种，分别是 State、 Getter、Mutation 、Action、 Module |

### vuex的State特性是

|  |
| --- |
| 1、Vuex就是一个仓库，仓库里面放了很多对象。其中state就是数据源存放地，对应于与一般Vue对象里面的data  2、state里面存放的数据是响应式的，Vue组件从store中读取数据，若是store中的数据发生改变，依赖这个数据的组件也会发生更新  3、它通过mapState把全局的 state 和 getters 映射到当前组件的 computed 计算属性中 |

### vuex的Getter特性是

|  |
| --- |
| 1、getters 可以对State进行计算操作，它就是Store的计算属性  2、 虽然在组件内也可以做计算属性，但是getters 可以在多组件之间复用  3、 如果一个状态只在一个组件内使用，是可以不用getters |

### vuex的Mutation特性是

|  |
| --- |
| 1、Action 类似于 mutation，不同在于：  2、Action 提交的是 mutation，而不是直接变更状态。  3、Action 可以包含任意异步操作 |

### Vue.js中ajax请求代码应该写在组件的methods中还是vuex的actions中

|  |
| --- |
| 1、如果请求来的数据是不是要被其他组件公用，仅仅在请求的组件内使用，就不需要放入vuex 的state里。  2、如果被其他地方复用，这个很大几率上是需要的，如果需要，请将请求放入action里，方便复用，并包装成promise返回，在调用处用async await处理返回的数据。如果不要复用这个请求，那么直接写在vue文件里很方便。 |

### 不用Vuex会带来什么问题

|  |
| --- |
| 1、可维护性会下降，你要想修改数据，你得维护三个地方  2、可读性会下降，因为一个组件里的数据，你根本就看不出来是从哪来的  3、增加耦合，大量的上传派发，会让耦合性大大的增加，本来Vue用Component就是为了减少耦合，现在这么用，和组件化的初衷相背。  但兄弟组件有大量通信的，建议一定要用，不管大项目和小项目，因为这样会省很多事 |

### state

|  |
| --- |
| state: {  count: 0  },  没有在computed声明count，html中 如下访问  <h3>{{$store.state.count}}</h3>  在computed声明count，就 {{count}}  import store from '@/vuex/store'  computed:{  count(){  return this.$store.state.count; // 不需要 import store from '@/vuex/store'  或  return store.state.count  }  }  或mapState参数：Array/{}  computed:mapState({  count:*state=>*state.count,  // 传字符串参数 'count' 等同于 `state => state.count`  countAlias: 'count',  })  或  computed:mapState(["count"])  或  computed: {  localComputed () { /\* ... \*/ },  // 使用对象展开运算符将此对象混入到外部对象中  ...mapState({  })  } |

### getter

|  |
| --- |
| state: {  todos: [  { id: 1, text: '...', done: true },  { id: 2, text: '...', done: false }  ]  },  getters: {  doneTodosCount: (*state*, *getters*) *=>* {  return getters.doneTodos.length  }  }  访问 store.getters.doneTodos  getter 在通过属性访问时是作为 Vue 的响应式系统的一部分缓存其中的  computed: {  doneTodosCount () {  return this.$store.getters.doneTodosCount  }  }  computed: {  // 使用对象展开运算符将 getter 混入 computed 对象中  ...mapGetters([  'doneTodosCount',  'anotherGetter',  // ...  ])  ...mapGetters(["count"])  ...mapState(["count"]),  } |

### mutation

|  |
| --- |
| 更改 Vuex 的 store 中的状态的唯一方法是提交 mutation  必须是同步函数  *const* store = new Vuex.Store({  state: {  count: 1  },  mutations: {  increment (*state*) {  // 变更状态  state.count++  }  }  })  store.commit('increment')  mutations: {  increment (state, payload) {  state.count += payload.amount  }  }  store.commit('increment', {  amount: 10  })  或  store.commit({  type: 'increment',  amount: 10  })  在组件中提交 Mutation  this.$store.commit('xxx', {num:10})  或  methods: {  ...mapMutations([  'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`  // `mapMutations` 也支持载荷：  'incrementBy' // 将 `this.incrementBy(amount)` 映射为  `this.$store.commit('incrementBy', amount)`  ]),  ...mapMutations({  add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`  })  }  因为mapActions/mapMutations只是把action/mutation函数绑定到你的methods里了;  固调methods里的方法的时候照常传参就可以了。  如：this.increment({num:10}) |

### action

|  |
| --- |
| Action 提交的是 mutation，而不是直接变更状态。  Action 可以包含任意异步操作。  *const* store = new Vuex.Store({  state: {  count: 0  },  mutations: {  increment (*state*) {  state.count++  }  },  actions: {  increment (*context*) {  context.commit('increment')  }  }  })  在组件中分发 Action  this.$store.dispatch('xxx',{num:10})  或  methods: {  ...mapActions([  'increment', // 将 `this.increment()` 映射为 `this.$store.dispatch('increment')`  // `mapActions` 也支持载荷：  'incrementBy' // 将 `this.incrementBy(amount)` 映射为 `this.$store.dispatch('incrementBy',  amount)`  ]),  ...mapActions({  add: 'increment' // 将 `this.add()` 映射为 `this.$store.dispatch('increment')`  })  }  调用:this.increment({num:10}) |

# vue组件的三种使用方式教程

|  |
| --- |
| ### 1.基本组件四步骤  1. 写好组件（废话~）  2. 在页面中引用组件  3. 在components中声明组件  4. 在页面上使用  ./src/components/button.vue  ```vue  <!-- 1.写好子组件 -->  <template>  <button class="btn" :style="{color:color}">  <slot/> <!-- 插槽 -->  </button>  </template>  <script>  export default {  // 传入子组件的参数写到props  props: {  color: {  type: String, // 颜色参数类型  default: "#000" // 默认黑色  }  }  }  </script>  <style scoped>  .btn {  width: 110px;  height: 60px;  border-radius: 10px;  border: none;  font-size: 15px;  }  </style>  ```  ```vue  // 父组件  <template>  <div id="app">  <!-- 4. 在页面上使用 -->  <Button color="red">我是插槽的值</Button>  </div>  </template>  <script>  // 2. 在页面种引用组件  import Button from '@/components/Button.vue'  export default {  name: "app",  // 3. 在components中声明组件  components: {  Button  }  };  </script>  ```  ### 2.全局组件五步骤  1. 写好组件（还是废话~）  2. 子组件添加install方法  3. 在 main.js 中引用  4. 使用 Vue.use 方法  5. 在页面上使用  \*\*1.子组件还是那样~~：\*\*  \*\*2. 子组件添加install方法\*\*  button.js  ```js  import ButtonComponent from './button.vue'  // 添加install方法 （插件方法）  *const* Button = {  install: *function* (*Vue*) {  Vue.component("Button", ButtonComponent);  }  }  // 导出Button  export default Button    ------------------处理多个全局组件------------------------  import ButtonComponent1 from './Button1.vue'  import ButtonComponent2 from './Button2.vue'  import ButtonComponent3 from './Button3.vue'  *const* buttonList = [  ButtonComponent1,  ButtonComponent2,  ButtonComponent3  ];  // 添加install方法 （插件方法）  *const* Button = {  install: *function* (*Vue*) {  buttonList.forEach(*button=>*{  // 这里 使用每个组件的 name 属性作为组件名  Vue.component(button.name, button);  })  }  }  // 导出Button  export default Button  ```  main.js  ```js  import Vue from 'vue'  import App from './App.vue'  // 3  import Button from '@/components/button.js'  // 4  Vue.use(Button);  new Vue({  render: *h* *=>* h(App),  }).$mount('#app')  ```  \*\*5. 在页面上使用\*\*  app.vue  ```vue  <template>  <div id="app">  <!-- 5. 在页面上使用 -->  <Button color="blue">我是全局按钮</Button>  </div>  </template>  ```  ### 3.构造组件四步骤  1. 写好组件（还\*\*是废话~）  2. vue.extend构建组件  3. 挂载 Vue.prototype  4. 在js中使用  \*\*1.写好子组件：\*\*  ```vue  <template>  <p class="Message">{{value}}</p>  </template>  <script>  export default {  data() {  return {  value: "我是一个弹框"  };  }  };  </script>  <style>  .Message {  position: fixed;  bottom: 30px;  width: 200px;  background-color: rgba(0, 0, 0, 0.5);  color: #fff;  border-radius: 10px;  left: 50%;  transform: translateX(-50%);  line-height: 30px;  text-align: center;  font-size: 15px;  animation: messageFade 3s 1;  }  /\* 加个简单动画 \*/  @keyframes messageFade {  0% {  opacity: 0;  -webkit-transform: translate3d(-50%, 80%, 0);  transform: translate3d(-50%, 80%, 0);  }  16% {  opacity: 1;  -webkit-transform: translate3d(-50%, 0, 0);  transform: translate3d(-50%, 0, 0);  }  84% {  opacity: 1;  -webkit-transform: translate3d(-50%, 0, 0);  transform: translate3d(-50%, 0, 0);  }  100% {  opacity: 0;  -webkit-transform: translate3d(-50%, 80%, 0);  transform: translate3d(-50%, 80%, 0);  }  }  </style>  ```  \*\*2. vue.extend构建组件\*\*  Message.js  ```js  import Vue from 'vue';  import Message from './Message.vue';  // 构造组件  *const* MessageConstructor = Vue.extend(Message);  // 设置删除组件  *const* removeDom = (*target*) *=>* {  target.parentNode.removeChild(target);  };  // 构造组件添加关闭方法  *MessageConstructor*.prototype.close = *function*() {  this.visible = false;  removeDom(this.$el);  };  *const* MessageDiv = (*options*) *=>* {  // 实例化组件  *const* instance = new MessageConstructor({  el: document.createElement('div'),  // 组件参数，运用到组件内的data  data: options,  });  // 在body添加组件  document.body.appendChild(instance.$el);  Vue.nextTick(() *=>* {  instance.timer = setTimeout(() *=>* {  // 定时关闭组件  instance.close();  }, 3000);  });  return instance;  };  export default MessageDiv;  ```  \*\*3. 挂载 Vue.prototype\*\*  main.js  ```js  import Message from '@/components/Message.js'  *Vue*.prototype.$message = Message;  ```  \*\*4. 使用\*\*  ```vue  <template>  <div id="app">  <Button color="blue" @click.native="msg">我是全局按钮</Button>  </div>  </template>  <script>  import Button from "@/components/Button.vue";  export default {  name: "app",  components: {  Button  },  methods: {  msg() {  // 4. 使用构造组件  this.$message({value:'我是构造组件'});  }  }  };  </script>  ``` |

# vue和Angular的区别

|  |
| --- |
| 1.vue仅仅是mvvm中的view层，只是一个如jquery般的工具库，而不是框架，而angular而是mvvm框架。  2.vue的双向邦定是基于ES5中的getter/setter来实现的，而angular而是由自己实现一套模版编译规则，需要进行所谓的“脏”检查，vue则不需要。因此，vue在性能上更高效，但是代价是对于ie9以下的浏览器无法支持。  3.vue需要提供一个el对象进行实例化，后续的所有作用范围也是在el对象之下，而angular是整个html页面。一个页面，可以有多个vue实例，而angular好像不是这么玩的。  4.vue真的很容易上手，学习成本相对低，不过可以参考的资料不是很丰富，官方文档比较简单，缺少全面的使用案例。高级的用法，需要自己去研究源码，至少目前是这样 |

# 说说你对angular脏检查理解

|  |
| --- |
| 在angular中你无法判断你的数据是否做了更改，所以它设置了一些条件，  当你触发这些条件之后,它就执行一个检测来遍历所有的数据，对比你更改的地方，然后执行变化。  这个检查很不科学。而且效率不高，有很多多余的地方，所以官方称为 脏检查 |

# mint-ui是什么？怎么使用？说出至少三个组件使用方法？

|  |
| --- |
| 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 |

# axios是什么？怎么使用？描述使用它实现登录功能的流程？

|  |
| --- |
| 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 |

# 请说出vue.cli项目中src目录每个文件夹和文件的用法？

|  |
| --- |
| 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 |

# 聊聊你对Vue.js的template编译的理解

|  |
| --- |
| 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） |

# 组件化思想

# 写过render函数么，跟template有啥区别 【写过，说了一下区别】

|  |
| --- |
| <https://cn.vuejs.org/v2/guide/render-function.html> |

# Vue computed 实现

|  |
| --- |
| <https://segmentfault.com/a/1190000010408657>  从两个问题出发：  建立与其他属性（如：data、 Store）的联系；  属性改变后，通知计算属性重新计算。  实现时，主要如下  初始化 data， 使用 Object.defineProperty 把这些属性全部转为 getter/setter。  初始化 computed, 遍历 computed 里的每个属性，每个 computed 属性都是一个 watch 实例。每个属性提供的函数作为属性的 getter，使用 Object.defineProperty 转化。  Object.defineProperty getter 依赖收集。用于依赖发生变化时，触发属性重新计算。  若出现当前 computed 计算属性嵌套其他 computed 计算属性时，先进行其他的依赖收集。 |

# vue complier 实现

|  |
| --- |
| <https://link.juejin.im/?target=https%3A%2F%2Fjkchao.cn%2Farticle%2F59d6e93c7e2ee06d412efef9> |

# 怎么快速定位哪个组件出现性能问题

|  |
| --- |
| 当面试官问这个问题，没有 get 到面试官的点，扯了一堆乱七八糟没用的 - -。  后来面试官说主要是用 timeline 工具。  大意是通过 timeline 来查看每个函数的调用时常，定位出哪个函数的问题，从而能判断哪个组件出了问题。  附上两个使用 timeline 的文章：  <https://juejin.im/post/5a6e78abf265da3e3f4cf085>  <https://developers.google.cn/web/tools/chrome-devtools/?hl=zh-cn> |

# vue怎么监听数组

|  |
| --- |
| <https://cn.vuejs.org/v2/guide/list.html#注意事项>  <https://segmentfault.com/q/1010000015949937>  首先，Vue.js是基于Object.defineProperty对对象实现“响应式化”，而对于数组，Vue.js提供的方法是重写push、pop、shift、unshift、splice、sort、reverse这七个数组方法。  如果需要用数组下标修改数组并实现响应式数据变化 vm.set() vue.set()及vm.$remove()方法 |

# reatc 和 angular 的了解和看法？

# vue服务器渲染的好处

# vue的服务端渲染和客户端渲染区别是啥，服务端渲染作用是啥 【SEO友好，首屏渲染速度等】

# vue SSR

# Electron

# vue里使用jquery

|  |
| --- |
| <script type="text/javascript" src="./jquery-3.1.1.min.js"></script>  *var* app=new Vue({  el:'#app',  data:{  message:'hello Vue!'  },  //在Vue中使用jQuery  mounted:*function*(){  // 挂载了之后才能获取到元素  $('#app').html('我是jQuery!');  },  methods:{  add:*function*(){  *console*.log("调用了Add方法");  }  }  })  app.add(); |

# 百度下拉列表

|  |
| --- |
| .gray{*background*: #ccc;}  <input type="text"  v-model="t1"  @keyup="get($event)"  @keydown.down="changeDown()"  @keydown.up.prevent="changeUp()">  <ul>  <li v-for="value in myData" :class="{gray:$index==now}">{{value}} </li>  </ul>  <p v-show="myData.length==0">暂无数据...</p>  <script src="vue.js"></script>  <script src="vue-resource.js"></script>  new Vue({  el:'body',  data:{  myData:[],  t1:'',  now:-1  },  methods:{  get:*function*(*ev*){  if(ev.keyCode==38 || ev.keyCode==40)return; // 上下键不触发请求  if(ev.keyCode==13){ // open() 方法用于打开一个新的浏览器窗口或查找一个已命名的窗口  window.open('https://www.baidu.com/s?wd='+this.t1);  this.t1='';  }  this.$http.jsonp('https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hhy/su',{  wd:this.t1  },{  jsonp:'cb'  }).then(*function*(*res*){  this.myData=res.data.s;  },*function*(){  });  },  changeDown:*function*(){  this.now++;  if(this.now==this.myData.length)this.now=-1;  this.t1=this.myData[this.now];  },  changeUp:*function*(){  this.now--;  if(this.now==-2)this.now=this.myData.length-1;  this.t1=this.myData[this.now];  }  }  }); |

# 简易留言板

|  |
| --- |
| th,td {*width*: 25%;}  <div class="container" id="box">  <form role="form">  <div class="form-group">  <label for="username">用户名:</label>  <input type="text" id="username" class="form-control" placeholder="输入用户名"  v-model="username">  </div>  <div class="form-group">  <label for="age">年　龄:</label>  <input type="text" id="age" class="form-control" placeholder="输入年龄" v-model="age">  </div>  <div class="form-group">  <input type="button" value="添加" class="btn btn-primary" v-on:click="add()">  <input type="reset" value="重置" class="btn btn-danger">  </div>  </form>  <hr>  <table class="table table-bordered table-hover">  <caption class="h3 text-info">用户信息表</caption>  <tr class="text-danger">  <th class="text-center">序号</th>  <th class="text-center">名字</th>  <th class="text-center">年龄</th>  <th class="text-center">操作</th>  </tr>  <tr class="text-center" v-for="item in myData">  <td>{{$index+1}}</td>  <td>{{item.name}}</td>  <td>{{item.age}}</td>  <td>  <button class="btn btn-primary btn-sm" data-toggle="modal" data-target="#layer"  v-on:click="nowIndex=$index">删除</button>  </td>  </tr>  <tr v-show="myData.length!=0">  <td colspan="4" class="text-right">  <button class="btn btn-danger btn-sm" v-on:click="nowIndex=-2" data-toggle="modal"  data-target="#layer" >删除全部</button>  </td>  </tr>  <tr v-show="myData.length==0">  <td colspan="4" class="text-center text-muted">  <p>暂无数据....</p>  </td>  </tr>  </table>  <!--模态框 弹出框-->  <div role="dialog" class="modal fade bs-example-modal-sm" id="layer">  <div class="modal-dialog">  <div class="modal-content">  <div class="modal-header">  <button type="button" class="close" data-dismiss="modal">  <span>&times;</span>  </button>  <h4 class="modal-title">确认删除么?</h4>  </div>  <div class="modal-body text-right">  <button data-dismiss="modal" class="btn btn-primary btn-sm">取消</button>  <button data-dismiss="modal" class="btn btn-danger btn-sm"  v-on:click="deleteMsg(nowIndex)">确认</button>  </div>  </div>  </div>  </div>  </div>  <link rel="stylesheet" href="bootstrap.min.css"/>  <script src="jquery-1.11.1.js"></script>  <script src="bootstrap.js"></script>  <script src="node\_modules/vue/dist/vue.js"></script>  new Vue({  el:'#box',  data:{  myData:[],  username:'',  age:'',  nowIndex:-100  },  methods:{  add:*function*(){  this.myData.push({  name:this.username,  age:this.age  });  this.username='';  this.age='';  },  deleteMsg:*function*(*n*){  if(n==-2){  this.myData=[];  }else{  this.myData.splice(n,1);  }  }  }  }); |

# vue-cli访问静态资源

|  |
| --- |
| <http://localhost:8080/static/mock/index.json> |

# axios访问static里的mock数据

|  |
| --- |
| 返回数据 |