面试题

目录

[第一章 ：vue 2](#_Toc1426883796)

[1. 什么是vue? 2](#_Toc686073323)

[2. MVVM 与MVC? 2](#_Toc994638918)

[a) Mvvm ： 2](#_Toc883586578)

[b) Mvc 3](#_Toc590197441)

[M:业务模型 3](#_Toc221425394)

[V:用户界面 3](#_Toc2054920354)

[C:控制器 3](#_Toc1214378624)

[3. vue如何实现mvvm?双向数据绑定? 3](#_Toc376952480)

[4. 常见vue的指令 3](#_Toc363572710)

[5. 自定义指令v-big v-focus? 4](#_Toc975561255)

[6. watch 侦听属性与 computed 的区别? 4](#_Toc220367940)

[7.vue中 key有什么作用？（内部原理） 6](#_Toc1462160152)

[a) 虚拟dom中key的作用 6](#_Toc870302043)

[b) 对比规则 6](#_Toc655316984)

[c) 用index做key可能会引发的问题 6](#_Toc1616408272)

[d) 开发中如何选择key? 6](#_Toc1305692954)

[8. 过滤器 filter :对某些数据进行处理 例如性别等 6](#_Toc1793572832)

[9. vue的生命周期? 6](#_Toc350634485)

[10. 组件通信? (10种) 7](#_Toc418662027)

[1）:props 7](#_Toc1296260217)

[a):布尔值类型，把路由中params参数映射为组件props数据 7](#_Toc23868304)

[b):对象，静态数据，很少用 7](#_Toc1722626986)

[c):函数，可以把路由中params|query参数映射为组件props数据 8](#_Toc1964708495)

[2）:自定义事件 $emit $on[简写@] 8](#_Toc1147119193)

[3）：$bus 全局事件总线----【万能】任意组件 8](#_Toc1671577632)

[12.vue监视数据的原理？ 20](#_Toc824190970)

[1）对像中后追加的属性，vue默认不做响应式处理。 20](#_Toc908109640)

[2）如需给后追加的属性做响应式，用如下api 20](#_Toc432440251)

[1）调用原生对应的方法对数组进行更新。 20](#_Toc938637109)

[2）重新解析模版，进而更新页面。 20](#_Toc259020101)

[13. 数据代理？vue的数据代理？ 20](#_Toc401485038)

[14.vue双向绑定的原理？ 21](#_Toc365414792)

[15. 浏览器本地存储---webStorage 21](#_Toc1870662371)

[16. Vue与VueComponent的关系？ 22](#_Toc1061877317)

[17. $nextTick 22](#_Toc1382960249)

[18. vue中的 ajax---axios 以及二次封装 22](#_Toc1197393462)

[19. 跨域？ 23](#_Toc522659797)

[（1） cors（不推荐） 23](#_Toc1135091949)

[（2） jsonp 24](#_Toc1393150542)

[（3） 配置代理服务器 可以使用nginx;借用vue-cli vue脚手架中 24](#_Toc666956153)

[20. 路由 vue-router （一个vue插件，用来实现spa 单页面应用） 25](#_Toc1814909778)

[1） 下载 npm install vue-router --save 25](#_Toc330916858)

[2） 什么是路由？ 25](#_Toc1884470323)

[一个路由是一组映射关系 （key-value） key为路径 ，value可能是function也可能是component 25](#_Toc1203892705)

[3） 路由的分类 25](#_Toc233770901)

[（1） 后端路由：value是function用来处理客户端发送的请求；服务器收到一个请求，根据请求的路径找到匹配的函数来处理，返回响应数据。 25](#_Toc1239942744)

[（2） 前端路由：value是component，用于展示页面内容；当浏览器的路径改变时，显示对应的组件。 25](#_Toc536387920)

[4） 25](#_Toc2082904981)

[21. js中 Set和map的区别？（es6提出） 25](#_Toc1253085920)

[1） Set 25](#_Toc242931311)

[（1） 本身就是构造函数，用来生成Set数据结构。类似于数组，但是成员的值都是唯一的，没有重复的，默认可遍历它的默认遍历器生成函数就是它的values方法 25](#_Toc580131030)

[（2）Set函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。 25](#_Toc686463830)

[（3）向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是向 Set 加入值时认为NaN等于自身，而精确相等运算符认为NaN不等于自身。 25](#_Toc1115439126)

[（4）Set 结构的实例有以下属性。 25](#_Toc1800636019)

[（5）遍历操作，四个遍历方法，可以用于遍历成员。 26](#_Toc950017809)

[22. 数组去重的方法 26](#_Toc408400418)

[const arr = [1,2,3,3,3,4,5] 26](#_Toc628089514)

[1) const s = new Set(arr); //Set(5) {1, 2, 3, 4, 5} 26](#_Toc1418336793)

[[...s] // [1, 2, 3, 4, 5] 27](#_Toc917998251)

[也可用于字符串去重 27](#_Toc1274084509)

[[...new Set('ababbc')].join('') // "abc" 27](#_Toc978898526)

[2) function dedupe(array) { 27](#_Toc475306815)

[return Array.from(new Set(array)); //Array.from可以将set结构转化为数组 27](#_Toc1989956512)

[} 27](#_Toc288778806)

[dedupe([1, 1, 2, 3]) // [1, 2, 3] 27](#_Toc192350222)

[23. 单页面 与 多页面应用？ 27](#_Toc867292419)

[1）单页面：一个应用程序中，只有一个完整的HTML页面，其它所谓的页面，其实都是一些较大的组件而已。所谓的页面跳转，其实是更换不同的组件加载而已。 27](#_Toc1612173944)

[2）两者的区别？ 27](#_Toc1006302609)

[24. 27](#_Toc1494229338)

[25. 27](#_Toc838715748)

[26. 27](#_Toc212917728)

1. ：vue2
2. 什么是vue?

一套基于MVVM设计模式，用于构建用户界面的渐进式框架，其核心库只关注视图库。

1. MVVM 与MVC?
   1. Mvvm ：

视图/界面(view) : html + css 增强两者的功能，提供了变量 循环等

模型数据(Model)：data对象 统一保存页面所需的全部变量

控制器(ViewModel)：自动实现data中的数据与视图的实时同步更新，控制器已经封装了传统的增删改查的功能

总结：它有两个方向：一是将【模型】转化成【视图】，通过数据绑定的方式将后端传递的数据转化成所看到的页面。二是将【视图】转化成【模型】，通过DOM事件监听将所看到的页面转化成后端的数据。这两个方向都实现的，我们称之为数据的双向绑定。在MVVM的框架下视图和模型是不能直接通信的。它们通过ViewModel来通信，ViewModel通常要实现一个observer观察者，当数据发生变化，ViewModel能够监听到数据的这种变化，然后通知到对应的视图做自动更新，而当用户操作视图，ViewModel也能监听到视图的变化，然后通知数据做改动，这实际上就实现了数据的双向绑定。并且MVVM中的View 和 ViewModel可以互相通信。

* 1. Mvc

M:业务模型

V:用户界面

C:控制器

优点：

1. 耦合性低：视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码。
2. 重用性高：
3. 可维护性高

缺点：

1. 调试困难：因为模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难，每个构件在使用之前都需要经过彻底的测试。
2. 不适合小型，中等规模的应用程序
3. 增加系统结构和实现的复杂性
4. 视图与控制器间的过于紧密的连接并且降低了视图对模型数据的访问：视图与控制器是相互分离，但却是联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。
5. vue如何实现mvvm?双向数据绑定?

vue中的两大系统：响应系统和虚拟dom树

响应系统：将data数据放入vue中之后，vue会自定给每个变量添加访问器属性（Object.defineProperty，getter setter）,对数据进行监听

虚拟dom树：vue内存中仅保存可能发生变化的dom元素和可能变化的属性。数据相对更少，查询遍历更快捷。

1. 常见vue的指令

v-if:满足条件的就保留，否则就删除

v-show：控制元素的display:none;或display:block;（性能更好一些）

v-if v-else-if v-else

v-for 其中 :key只能是string 或 nubmer

v-on 或者 @

v-html html片段

v-cloak 避免由于网络加载缓慢用户看到 {{}} 是彻底隐藏，连结构和样式也看不见 自己不带隐藏样式 添加隐藏样式 等vue在加载完之后，会自动移除所有元素上的v-cloak属性

<元素 v-text="表达式"></元素> 保持其它样式和结构，仅内容文本暂时隐藏

v-once 只在首次扫描真实DOM树时，替换一次内容，且并没有将这个元素添加到虚拟DOM树。所以，今后变量变化，不会再扫描到这个元素。

v-pre 保持内容中的{{}}不被编译

|  |
| --- |
| 特殊： 双向数据绑定 v-model  1) 文本框或文本域 <input v-model:value=‘变量’ /> 或者 <input v-model=‘变量’ />  2) 单选框 <input type="radio" value="1" v-model:checked="变量">男 3) 复选框单用 <input type="checkbox" v-model:checked="变量">同意  4) 单选列表: <select>  <select v-mode:value="变量">  <option value="写死的value">  <option value="写死的value">  <select />  原理:  凡是带有v-model:的元素，都自动添加事件处理函数: oninput或onchange。在自动绑定的事件处理函数中，内置了自动修改data中变量的方法。 |

1. 自定义指令v-big v-focus?

|  |
| --- |
| //两种写法 函数  directives:{  // 自定义指令何时会被调用？1.指令与元素绑定成功后 2.当前绑定指令的模版再次解析时  big(element,binding){  //element: 真的dom元素  element.innerText = binding.value \* 10;  }  }  //对象  directives:{  focus:{  bind(element,binding){ //1.指令与元素绑定成功后  element.value = binding.value \* 10;  },  inserted(element,binding){ //指令所在元素被插入页面时  element.focus();  },  update(element,binding){ //指令所有在模版被再次解析时  element.value = binding.value \*10 ;  }  }  } |

1. watch 侦听属性与 computed 的区别?

a) watch 当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的。

|  |
| --- |
| watch:{  isHot:{ //监视谁 写谁名  Immediate:true, // 是否立即执行  handler(oldValue, newValue){  coosole.log(‘isHot被修改了’)  }  }  } 或者 vm.$watch(‘isHot’,{  Immediate:true, // 是否立即执行  handler(oldValue, newValue){  coosole.log(‘isHot被修改了’)  })  或者 简写  watch:{  isHot(oldValue, newValue){  coosole.log(‘isHot被修改了’)  }  }  或者  vm.$watch(‘isHot’,function(oldValue, newValue){  coosole.log(‘isHot被修改了’))  深度监听 ：  data(){  return{  number:{a:1,b:1} }  }, watch:{  deep:true ,// 检测多级结构所有属性的变化  //监视多级结构中的某个属性变化 ’number.b‘:{  handler(){ console.log(‘number中的b变化了’) }  } } |

b) computed 计算属性 （****计算属性是基于它们的响应式依赖进行缓存的****。）

|  |
| --- |
| data(){  return{  firstName:"张",  lastName:'位'  }  },  computed:{ //完整写法  fullName:{  get(){  return this.firstName + '-' +lastName;  },  set(value){  const arr = value.split('-');  this.firstName = arr[0]  }  }  }  或者 简写  computed:{ //简写  fullName(){  console.log('get被调用了')  return this.firstName + '-' +lastName;  }  } |

c) 总结： computed能完成的 watch都可以完成，反之则不行（watch可以执行异步操作 例如开启一秒的定时器执行）  
 两个小点：

1. 所有被vue管理的函数，最好写成普通函数，这样this的指向为vm或组件实例对象
2. 所有不被vue管理的函数（定时器函数，ajax的回调函数等）最好写成箭头函

7.vue中 key有什么作用？（内部原理）

* 1. 虚拟dom中key的作用

是虚拟dom的标识，刚数据发生变化时，vue会将新旧虚拟dom进行比较，相同的保留，不一样的进行替换

b) 对比规则

1. 旧虚拟dom找到与新虚拟dom一样的key

新旧比较其内容，相同的保留，不一样的使用新的，后替换页面真实dom

1. 旧虚拟dom没有找到与新虚拟dom一样的key

创建新的真实dom并渲染到页面

c) 用index做key可能会引发的问题

1. 若对数据进行逆序添加，逆序删除等破坏顺序的操作 => 会产生没有必要的真实dom，效率低
2. 如果结构中还包含输入类的dom =>会产生错误的dom更新

d) 开发中如何选择key?

使用每条数据的唯一标识作为key

在不对数据进行逆序添加，逆序删除等破坏顺序的操作的时候，可以使用index

1. 过滤器 filter :对某些数据进行处理 例如性别等

在绑定语法中: {{变量 | 过滤器(实参值, )}}

多个过滤器可以连用:

使用过滤器时，多个过滤器可用|连接

强调: 后一个过滤器收到的不一定是原始值，而是前一个过滤器处理后的中间产物

filters:{

过滤器名(oldval,其他形参){

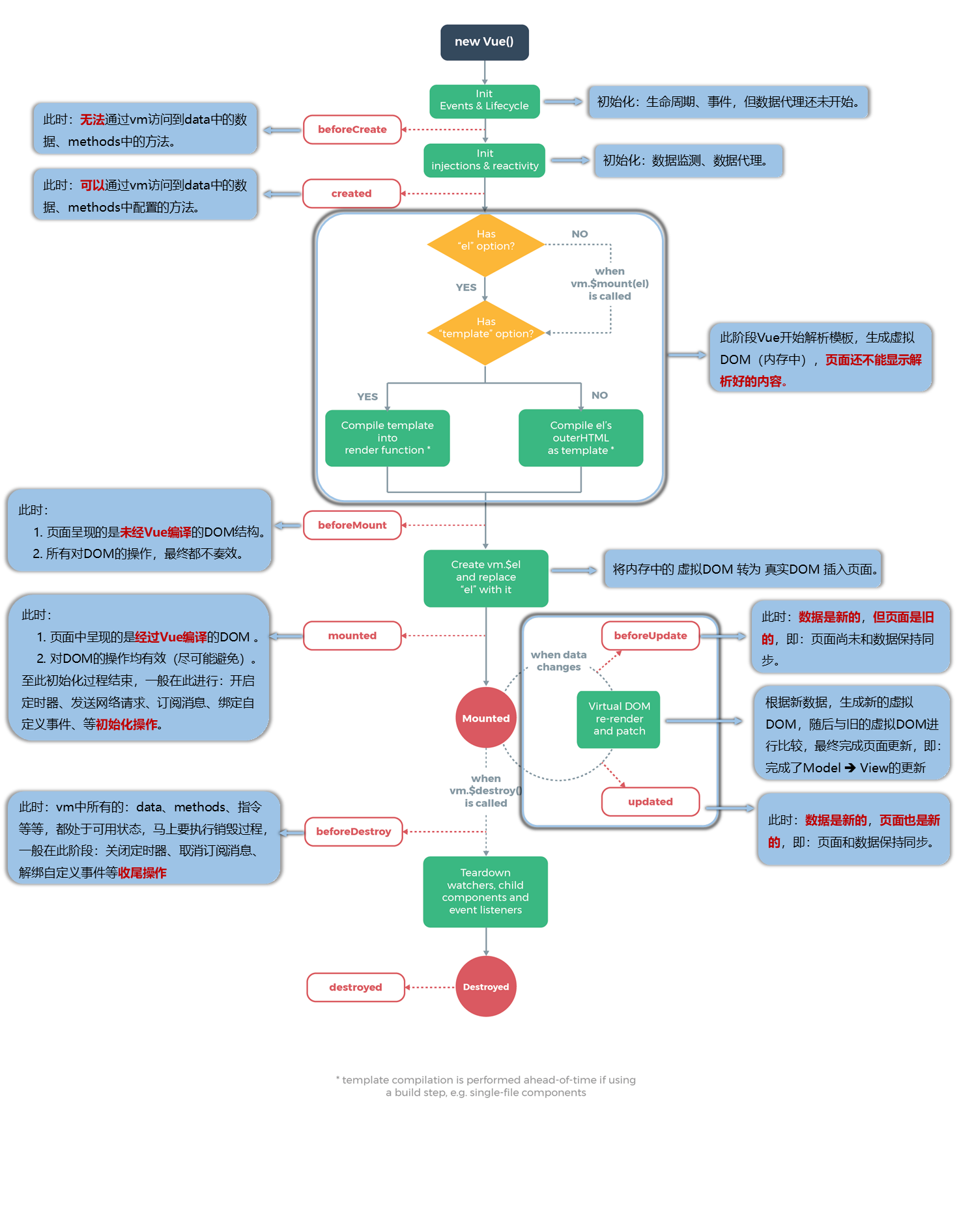
//强调: 第一个实参值，就写在实参列表的第一个位置，也不会覆盖过滤器函数定义时的第一个形参oldVal。而是自动传给过滤器函数定义时的第二个形参变量。过滤器第一个形参，永远接到的是变量的原始值。

return 根据不同实参值返回不同的结果

}

}

1. vue的生命周期?



1. 组件通信? (10种)

1):props

使用场景：父-> 子

传递的数据类型：

函数 ---- 子想传给父的数据

非函数 --- 父传给子的数据

<TodoList :todos="123" updateChecked="hander">

接收书写方式：3种

['todos'], {type:Array}, {type:Array,default:[]}

特殊情况：路由传递props

a):布尔值类型，把路由中params参数映射为组件props数据

b):对象，静态数据，很少用

c):函数，可以把路由中params|query参数映射为组件props数据

2）:自定义事件 $emit $on[简写@]

第一种原生DOM事件（即系统事件click ，双击等），第二种自定义事件。

<Event1 @click="handler1"></Event1>

组件绑定原生DOM事件，并非原生DOM事件，而是所谓的自定义事件。

如果你想把自定义事件变为原生DOM事件，需要加上修饰符.native修饰

<!-- 当其加了.native 后 是给子组件的根节点绑定了点击事件 ----- 利用到事件委派 -->

<Event1 @click.native="handler1"></Event1>(vuex已取消，新增了emits配置项,接收自定义事件)

自定义事件-----[子给父传递数据]

父组件中： <Event2 @click="handler3" @xxx='handler3'>Event2组件的自定义事件回调</Event2>

子组件中：<button @click="$emit('xxx','自定义事件xxx')">自定义事件xxxx</button>

两种【简单写法|复杂写法】

<Todos @erha="handler">

<Todos ref='erha'>

mounted(){

this.$refs.erha.$on('xx',callBack)

}

3）：$bus 全局事件总线----【万能】任意组件

（1） $on(eventName, listener): 绑定自定义事件监听

（2） $emit(eventName, data): 触发自定义事件

（3） $off(eventName): 解绑自定义事件监听

（4） $once(eventName, listener): 绑定事件监听, 但只能处理一次

组件实例的原型的原型指向的Vue.prototype

new Vue({

// 配置全局时间总线

beforeCreate() {

Vue.prototype.$bus = this;

},

render: h => h(App),

router,

store

}).$mount('#app')

|  |
| --- |
| // 点击小图将 通过全局事件总线 $bus 图片的索引值 传递到 Zoom组件  //  changeZoomImg(index) {  this.currentIndex = index;  this.$bus.$emit("changImg", index);  },  ///////////////  // 通过全局事件总线 $bus 接收ImageList 传递的图片索引值 并修改当前 .big 的图片  //  this.$bus.$on("changImg", (index) => {  this.currentIndex = index;  });  beforeDestroy(){  //解绑当前组件所用的事件  this.$bus.$off('changImg')  } |

4）：pubsub-js【发布订阅消息】\**\*\**\*\*在vue中根本不用【React】 ----万能

|  |
| --- |
| 下载: npm install -S pubsub-js  相关语法  (1) import PubSub from 'pubsub-js' // 引入  (2) PubSub.subscribe(‘msgName’, functon(msgName, data){ })  (3) PubSub.publish(‘msgName’, data): 发布消息, 触发订阅的回调函数调用  (4) PubSub.unsubscribe(token): 取消消息的订阅 |

1. :Vuex[仓库] -----数据非持久化----万能的

采用MVC模式中的[Model](https://so.csdn.net/so/search?q=Model&spm=1001.2101.3001.7020" \t "/Users/wang/Documents\\x/_blank)层，规定所有的数据必须通过action—>mutaion—>state这个流程进行来改变状态的。再结合Vue的数据视图双向绑定实现页面的更新，统一页面状态管理。

核心概念：Vuex是通过全局注入store对象，来实现组件间的状态共享。在大型复杂的项目中（多级组件嵌套），需要实现一个组件更改某个数据，多个组件自动获取更改后的数据进行业务逻辑处理，这时候使用vuex比较合适。

state ：唯一的，vuex管理的状态对象

actions ：值为一个对象，包含多个响应用户动作的回调函数；通过commit()来触发mutations中函数的调用，从而更新state；通过在组件中this.$store.dispatch(‘actions中回调函数名’)来触发；可以包含异步代码。

mutations：值为一个对象，包含多个直接更新state的方法；在 action 中使用：**commit('对应的 mutations 方法名')** 触发；不能写异步代码，只能单纯的操作state。

getters ：值为一个对象，包含多个用于返回数据的方法；如何使用？—— **$store.getters.xxx**

modules：包含多个module，一个module是一个store的配置对象，与每个组件对应。

|  |
| --- |
| 1.概念：专门在 Vue 中实现集中式状态（数据）管理的一个 Vue 插件，对 vue 应  用中多个组件的共享状态进行集中式的管理（读/写），也是一种组件间通信的方  式，且适用于任意组件间通信。   1. 什么时候使用？   多个组件依赖于一个状态；来自不同组件的行为需要变更同一个状态   1. 使用步骤   1)下载 npm install vuex --save  2) 创建store    3) 创建小仓库 search    4) 使用store main.js    5) 组件search中触发 actions中的 回调 getSearchList  this.$store.dispatch("getSearchList", this.searchParams);  6) 获取search store的数据  import { mapGetters, mapState } from "vuex";  computed: {  ...mapState({  searchList: (state) => state.search.searchList,  }),  ...mapGetters(["goodsList"]),  }  7) 原理图     1. 四个map方法的使用        1. 模块化 + 命名空间 |

6）：插槽-----父子通信【结构】

（1）作用：让父组件可以向子组件指定位置插入html结构，也是通信中的一种方式。 父->子

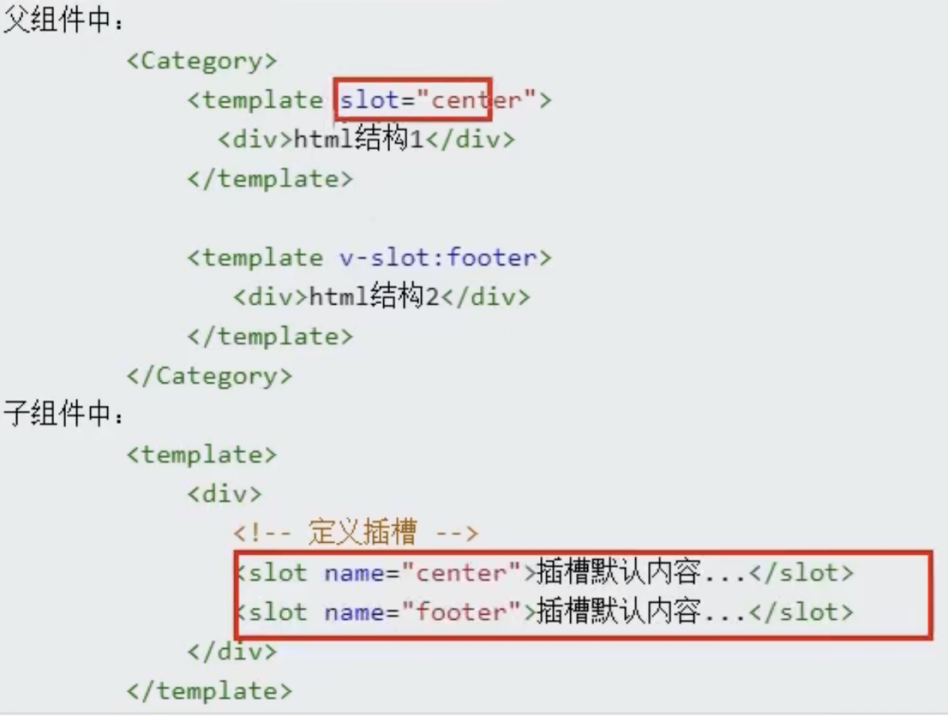
slot

（2）使用方式

1. 默认插槽



1. 具名插槽



1. 作用域插槽

数据在组件自身，但根据数据生成的结构需要组件的使用者开决定。（games数据在Category,但使用数据所遍历出来的结构由父组件决定）

|  |
| --- |
|  |

7）:v-model实现组件通信？

v-model：指令，可以收集表单数据【text、radio、checkbox、range】等等

切记：v-model收集checkbox需要用数组收集

v-model:实现原理 :value @input 还可以实现父子数据同步。

<CustomInput v-model="msg"></CustomInput>

|  |
| --- |
| 父组件：<h2>v-model实现原理(vue2)</h2>  <!-- 原生的dom中有 oninput事件，常结合表单元素一起使用，，当表单内容发生变化的时候就触发一次回调  vue2: 可以通过value与input事件实现v-model功能  -->  <input type="text" :value="msg" @input="msg = $event.target.value">  <span>{{msg}}</span>  <hr>  <!-- 深入学习v-model：实现父子组件数据同步（父子组件通信）  :value => props ，父子组件通信  @input => 自定义事件  -->  <CustomInput :value='msg' @input="msg = $event"/>  <!-- 上述 CustomInput 可简写为 如下  v-model 实现的原理 ： value与input事件实现的，可以使用v-model实现父子组件间的数据同步  -->  <CustomInput v-model="msg"/> |
| 子组件CustomInput：<div>  <h2>深入v-model</h2>  <!--  :value => 动态的属性  @input => 给原生dom绑定原生dom事件 -->  <input type="text" :value="value" @input="$emit('input',$event.target.value)"> <span>{{value}}</span>  </div>  <script>  export default {  name:'CustomInput',  props:['value']  }  </script> |

8）:属性修饰符.sync，可以实现父子数据同步。

以后在elementUI组件中出现，实现父子数据同步。

<!-- :money.sync => 1.父组件给子组件传递了prop（需要接收） 2.给当前子组件绑定了一个自定义事件，事件名称为 update:money -->

父组件：<Child2 :money.sync='money'/>

|  |
| --- |
| 子组件中：  <button @click="$emit('update:money',money - 100)">花钱</button>  export default {  name:'Child2',  props:['money']  } |

9）：$attrs与$listeners ----vue-helper 父子组件通信

$attrs：组件实例的属性，可以获取到父亲传递的props数据（前提子组件没有通过props接受）

$listeners：组件实例的属性，可以获取到父亲传递自定义事件（对象形式呈现）

|  |
| --- |
| 父组件： |
| 子组件： |

10）:$children与$parent 可以实现父子组件通信

ref:可以在父组件内部获取子组件---实现父子通信

$children:可以在父组件内部获取全部的子组件【返回数组】

$parent:可以在子组件内部获取唯一的父组件【返回组件实例】

===> 其上的功能可 完善

混入 mixin ： 项目中出现很多结构 功能 类似，---组件的复用

很多组件的js业务逻辑相似 -- 使用mixin

|  |
| --- |
| 父组件： |

|  |
| --- |
| 子组件son 使用mixin |
| Mixin.js |

11.模块、组件、模块化、组件化?

1）模块: 向外提供特定功能的 js 程序, 一般就是一个 js 文件

2）组件：用来实现局部(特定)功能效果的代码集合(html/css/js/image…..)

3）模块化：当应用中的 js 都以模块来编写的, 那这个应用就是一个模块化的应用。

4）组件化：当应用中的功能是多组件的方式来编写的，那这个应用就是一个组件化的应用。

|  |
| --- |
| 前端模块化的规范： AMD CMD CommonJS ES6  AMD ： RequireJS 在推广过程中对模块定义的规范化产出；提前执行；推荐的风格通过返回一个对象做为模块对象  CMD ： SeaJS 在推广过程中对模块定义的规范化产出；延迟执行；  CMD模块方式  define(function(require, exports, module) {  // 模块代码  });  CommonJS：通过对module.exports或exports的属性赋值来达到暴露模块对象的目的。    ES6：export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。使用import命令的时候，用户需要知道所要加载的变量名或函数名，否则无法加载。为了给用户提供方便，让他们不用阅读文档就能加载模块，就要用到export default命令，为模块指定默认输出。 |

12.vue监视数据的原理？

（1）vue会监视data中所有层次的数据

（2）如何监测对象中的数据？

通过setter实现监视，且要在vue时就传入要监测的数据。

1）对像中后追加的属性，vue默认不做响应式处理。

2）如需给后追加的属性做响应式，用如下api

Vue.set(target,propertyName/index,value) 或者

this.$set(target,propertyName/index,value)

1. 如需给删除的属性做响应式，用如下api

Vue.delete(target,propertyName/index) 或者

this.$delete(target,propertyName/index)

（3）如何监测数组中的数据？ 通过包裹数组更新元素的方法，本质时做了两件事。

1）调用原生对应的方法对数组进行更新。

2）重新解析模版，进而更新页面。

（4）vue中修改数组中的某个元素的方法。

1）push(),pop(),shift(),unshift(),sort(),splice,reverse()

2）Vue.set()或者vm.$set()

注意：Vue.set()，vm.$set() 不能给vm或者vm的根数据对象 添加属性！

1. 数据代理？vue的数据代理（也就是数据劫持）？
2. 数据代理：通过一个对象代理对另一个对象中属性值的 读/写 操作。 Object.defineProperty

|  |
| --- |
| // 通过obj2对obj中的 x 进行读 / 修改  let obj = { x:100}  let obj2 = {y:200}  Object.defineProperty(obj2,'x',{  // value:1 , // 当前所添加属性的属性值  // enumerable:true, //该属性是否可被枚举（遍历）默认值 false  // writable:true, // 是否可以被修改 默认值 false  // configurable:true, //是否可被删除，是否可修改enumerable writable 这两个属性，一旦为false 不可逆  get(){ //读取 obj中的 x  return obj.x  },  set(val){ //修改obj中的 x  obj.x=val  }  }) |

2）vue的数据代理

|  |
| --- |
| 通过vm对象来代理data对象属性的操作 读/写 更加方便的操作data中的数据  基本原理：通过Object.defineProperty()把data中的所有属性都添加到vm上，为每一个添加到vm上的属性，都指定一个getter / setter，在getter /setter 内部操作（读/写）data中的对应属性。 |

14.vue双向绑定的原理？

|  |
| --- |
| 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。 |

1. 浏览器本地存储---webStorage
2. 存储内容大小一般为5mb
3. 浏览器通过Window.sessionStorage和Window.localStorage属性来实现
4. 相关api

xxxxStorage.setItem(‘key’,’value’)

xxxxStorage.getItem(‘key’)

xxxxStorage.removeItem(‘key’)

xxxxStorage.clear() 会清空存储中的所有数据

1. 比较（ cookies会发送到服务器端。其余两个不会）

A.sessionStorage会随着浏览器窗口的关闭而消失。同时引入了一个“浏览器窗口”的概念。sessionStorage是在同源的同窗口（或tab）中，始终存在的数据。也就是说只要这个浏览器窗口没有关闭，即使刷新页面或进入同源另一页面，数据仍然存在。关闭窗口后，sessionStorage即被销毁。同时“独立”打开的不同窗口，即使是同一页面，sessionStorage对象也是不同的

B.localStorage需要手动清除

C.xxxxStorage.getItem(‘key’,’value’)，若key对应的value没有，返回值为null

D.JSON.parse(null)的结果依旧是null

1. Vue与VueComponent的关系？

内置关系 VueComponent.prototype.\_\_proto\_\_ === Vue.prototype

组件实例对象vc可以访问到Vue原型上的属性和方法

|  |
| --- |
|  |

1. $nextTick
2. this.$nextTick(回调函数)
3. 作用：在下一次dom更新结束后执行指定的回调
4. 何时用：当改变数据后，基于更新后的新dom进行某些操作时，要在nextTick所指定的回调函数中执行。
5. 原理：vue在更新dom是异步执行的，可以联系到event loop上
6. vue中的 ajax---axios 以及二次封装
7. 下载 npm install axios --save
8. src目录下 api目录 重写 axios 配置 响应拦截器与 请求拦截器（request.js）

// 二次封装 axios

import axios from 'axios';

// nprogress 进度条的使用

import nprogress from "nprogress";

// 引入进度条的样式（也可以在原css样式中进行样式的修改）

import "nprogress/nprogress.css";

// 引入 store

import store from '@/store';

// 1.利用axios对象的方法create，去创建一个axios实例

// 2.requsets就是axios，稍微配置一下

const requsets = axios.create({

baseURL:'/api',

timeout:5000

})

// 请求拦截器:再发请求之前，请求拦截器可以检测到，可以在请求发出去之前做一些事情

requsets.interceptors.request.use((config) => {

// 开启进度条

if(store.state.detail.uuid\_token){

// 请求头 添加一个字段 （userTempId）与后端配合

config.headers.userTempId = store.state.detail.uuid\_token;

}

//携带token 给服务器

if(store.state.user.token){

config.headers.token = store.state.user.token;

}

nprogress.start();

return config;

});

// 响应拦截器

requsets.interceptors.response.use(

(res) => {

// 关闭进度条

nprogress.done();

return res.data;

} ,

(error)=> {

nprogress.done();

return Promise.reject(new Error('faile'));

})

export default requsets;

1. 跨域？

违背同源策略：协议（http或https）、主机名、端口号 其中一个或者都不同

常见的解决跨域的方式

1. cors（不推荐）

由后端人员在服务器响应的时候配置几个特殊的响应头。服务器端对于CORS的支持，主要就是通过设置Access-Control-Allow-Origin来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问。

1. 通过修改document.domain来跨子域

将子域和主域的document.domain设为同一个主域.前提条件：这两个域名必须属于同一个基础域名!而且所用的协议，端口都要一致，否则无法利用document.domain进行跨域

主域相同的使用document.domain

1. 使用window.name来进行跨域

window对象有个name属性，该属性有个特征：即在一个窗口(window)的生命周期内,窗口载入的所有的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是持久存在一个窗口载入过的所有页面中的

1. 使用HTML5中新引进的window.postMessage方法来跨域传送数据

还有flash、在服务器上设置代理页面等跨域方式。个人认为window.name的方法既不复杂，也能兼容到几乎所有浏览器，这真是极好的一种跨域方法。

1. jsonp

通过script标签的src属性在引入外部资源的时候不受同源策略限制的特点来实现的，前后端一起配合，兼容性好，简单易用，支持浏览器与服务器双向通信，但是只能解决get请求。

原理是：动态插入script标签，通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。通过script标签实现跨域请求，然后在服务端输出JSON数据并执行回调函数，从而解决了跨域的数据请求。

JSONP：json+padding（内填充），顾名思义，就是把JSON填充到一个盒子里

<script>

function createJs(sUrl){

var oScript = document.createElement('script');

oScript.type = 'text/javascript';

oScript.src = sUrl;

document.getElementsByTagName('head')[0].appendChild(oScript);

}

createJs('jsonp.js');

box({

'name': 'test'

});

function box(json){

alert(json.name);

}

</script>

1. 配置代理服务器 可以使用nginx;借用vue-cli vue脚手架中

1）方式一 vue.config.js 文件中

devServer{

proxy:”http://loaclhost:5000”

}

A.优点：配置简单，请求直接资源时直接发给前端（8080）。

B.缺点：不能配置多个代理，不能灵活的控制请求是否走代理

C.工作方式:若按照上述配置代理，当请求 了前端不存在的资源时，那么该请求才转发给服务器（优先匹配前端资源）

1. 方式二 vue.config.js 文件 编写具体的配置代理规则

devServer: {

proxy: {

'/api1': { //匹配所有以 /api1 开头的 请求路径

target: 'http://localhost:5000', //代理目标的基础路径

changeOrigin:true, // 默认值为 true

pathRewrite:{'^/api1':''}

},

'/api2': { //匹配所有以 /api2 开头的 请求路径

target: 'http://localhost:5001', //代理目标的基础路径

changeOrigin:true,

pathRewrite:{'^/api2':''}

},

},

/\*\*

\* changeOrigin : true 时，服务器收到的请求头中host为 ：localhost:5000

\* changeOrigin : false 时，服务器收到的请求头中host为 ：localhost:8080

\*/

}

A.优点：可以配置多个代理，且可以灵活的控制请求是否走代理

B.缺点：配置略复杂，请求资源时必须加前缀

1. 路由 vue-router （一个vue插件，用来实现spa 单页面应用）
2. 下载 npm install vue-router --save
3. 什么是路由？

一个路由是一组映射关系 （key-value） key为路径 ，value可能是function也可能是component

1. 路由的分类
2. 后端路由：value是function用来处理客户端发送的请求；服务器收到一个请求，根据请求的路径找到匹配的函数来处理，返回响应数据。
3. 前端路由：value是component，用于展示页面内容；当浏览器的路径改变时，显示对应的组件。
4. 基本的使用

|  |
| --- |
| 1. 编写router文件夹 index.js   import Vue from 'vue';  import VueRouter from 'vue-router';  Vue.use(VueRouter);  import routes from './routes';  // 引入 store  import store from '@/store';  // 重写$router.push|replace 为解决路由跳转是传相同的参数，出现 NavigationDuplicated 的错误  // 参数1：location--往哪跳转， resolve--成功的回调， reject--失败的回调  // 1.重写push:先保存一份原有的push|replace方法  let originPush = VueRouter.prototype.push;  let originReplace = VueRouter.prototype.replace;  VueRouter.prototype.push = function (location, resolve, reject) {  if (resolve && reject) {  originPush.call(this, location, resolve, reject);  } else {  originPush.call(this, location, () => { }, () => { });  }  };  // \*\*\*\*call 与 apply 的区别\*\*\*\*  /\*  相同点:都可以调用函数一次，都可以篡改函数的上下文一次  不同点：传递的参数，call传递参数用逗号隔开；apply方法执行，传递数组  \*/  VueRouter.prototype.replace = function (location, resolve, reject) {  if (resolve && reject) {  originReplace.call(this, location, resolve, reject)  } else {  originReplace.call(this, location, () => { }, () => { })  }  }  let router = new VueRouter({  mode:'history', //hash 路由器的工作模式  // 配置路由  routes,  // 滚动行为设置，使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样  scrollBehavior(to, from, savedPosition) {  return { x: 0, y: 0 }  }  });  export default router; |
| 1. main.js 中   import router from './router';  new Vue({  render: h => h(App),  router,  }).$mount('#app') |
| 1. 编写routes 配置   //路由的懒加载  /\*  component: () => import('@/pages/Search')  1. import(modulePath): 动态import引入模块, 被引入的模块会被单独打包  2. 组件配置的是一个函数, 函数中通过import动态加载模块并返回,  初始时函数不会执行, 第一次访问对应的路由才会执行, 也就是说只有一次请求对应的路由路径才会请求加载单独打包的js  作用: 用于提高首屏的加载速度  \*/  export default [{  // 代表 params参数可传不可传  path: '/search/:keyword?',  name: 'search', //===>这是命名路由，可以简化跳转  component: () => import ('@/pages/Search'),  meta: {  isShow: true  }  },  {  path:'/communication',  component: () => import('@/pages/Communication'),  children:[ // 配置children 嵌套路由  {  path:'event',//一定不要写 /  name:’event’  // 这是命名路由 可以简化跳转 <router-link to='/communication/event'>event深入</router-link> 直接简化为 <router-link :to=”{name:’event’}}”>event深入</router-link>    component:()=>import('@/pages/Communication/EventTest'),  meta:{  isHideFooter:true  }  },  {  path:'model',  component:()=>import('@/pages/Communication/ModelTest'),  meta:{  isHideFooter:true  }  },] |
| 1. 实现切换   <router-link to="/center/myorder" active-class="active"> 我的订单 </router-link>  active-class： 可配置高亮样式 |
| 1. 指定展示位置   <router-view></router-view> |

1. 路由传参 query参数



1. 路由传参 params参数



1. 路由的props配置



1. router-link的replace属性

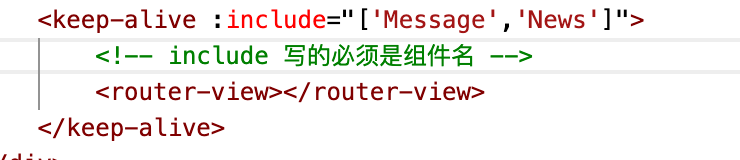


1. 编程式路由导航

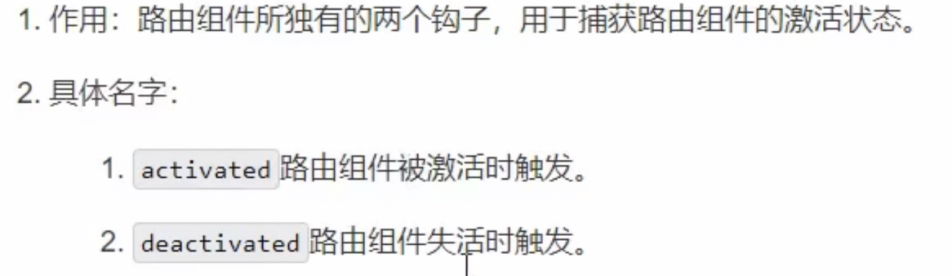


1. 缓存路由组件 <keep-alive></keep-alive>

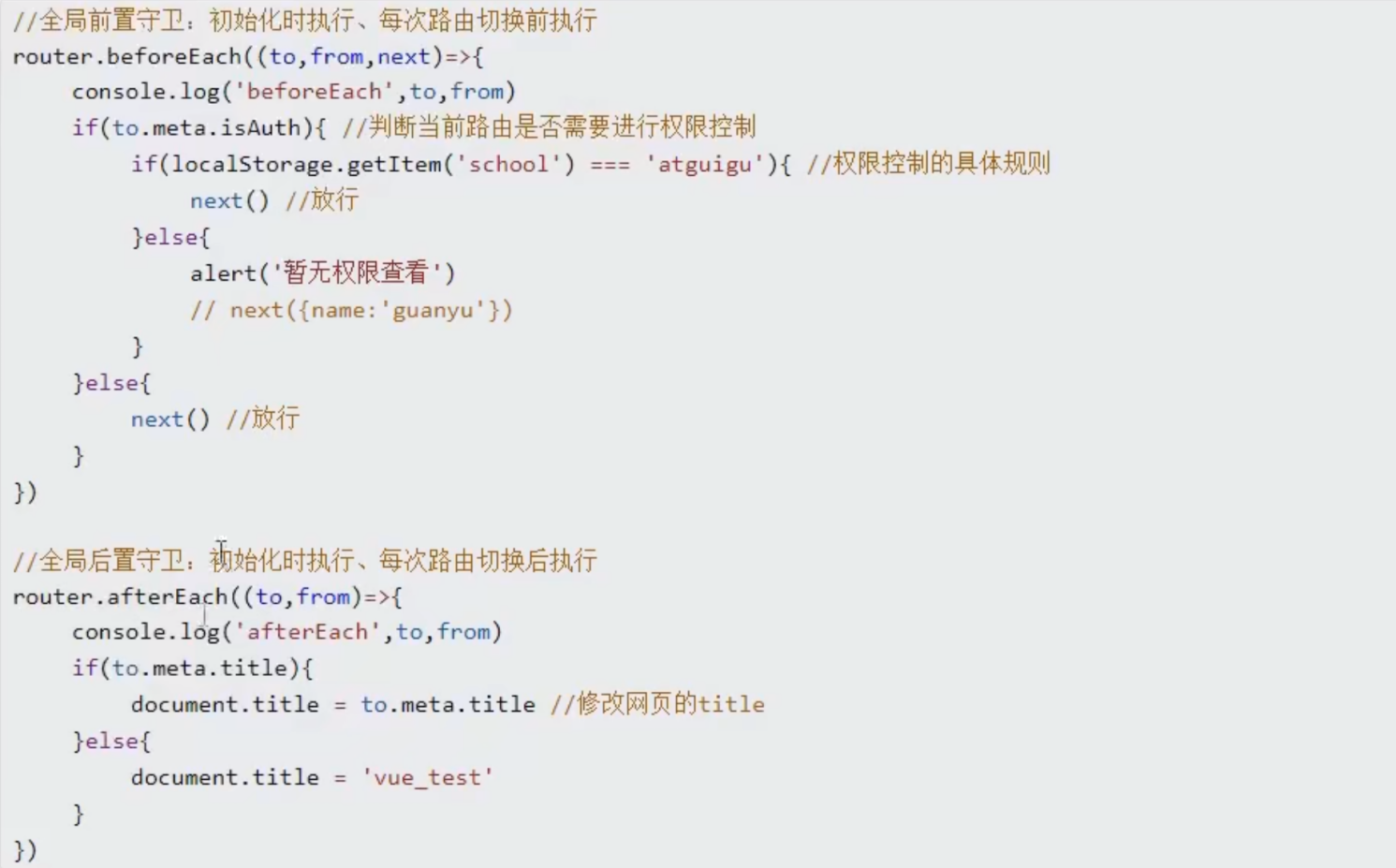




1. 两个新的生命周期钩子 activated deactivated



1. 路由守卫---对路由权限进行控制（全局守卫、独享守卫、组件内守卫）
2. 全局守卫



1. 独享守卫：某一个路由所独享的



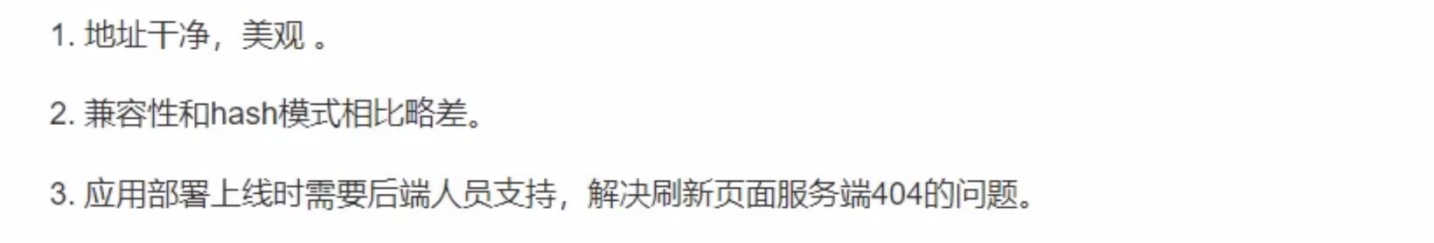
1. 组件内守卫



1. 路由器的两种工作模式
2. hash模式



1. history模式



1. js中 Set和map的区别？（es6提出）
2. Set
3. 本身就是构造函数，用来生成Set数据结构。类似于数组，但是成员的值都是唯一的，没有重复的，默认可遍历它的默认遍历器生成函数就是它的values方法

（2）Set函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化。

（3）向 Set 加入值的时候，不会发生类型转换，所以5和"5"是两个不同的值。Set 内部判断两个值是否不同，使用的算法叫做“Same-value-zero equality”，它类似于精确相等运算符（===），主要的区别是向 Set 加入值时认为NaN等于自身，而精确相等运算符认为NaN不等于自身。

（4）Set 结构的实例有以下属性。

* Set.prototype.constructor：构造函数，默认就是Set函数。
* Set.prototype.size：返回Set实例的成员总数。

Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

* Set.prototype.add(value)：添加某个值，返回 Set 结构本身。
* Set.prototype.delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
* Set.prototype.has(value)：返回一个布尔值，表示该值是否为Set的成员。
* Set.prototype.clear()：清除所有成员，没有返回值。

（5）遍历操作，四个遍历方法，可以用于遍历成员。

* Set.prototype.keys()：返回键名的遍历器
* Set.prototype.values()：返回键值的遍历器
* Set.prototype.entries()：返回键值对的遍历器
* Set.prototype.forEach()：使用回调函数遍历每个成员

需要特别指出的是，Set的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

keys方法、values方法、entries方法返回的都是遍历器对象（详见《Iterator 对象》一章）。由于 Set 结构没有键名，只有键值（或者说键名和键值是同一个值），所以keys方法和values方法的行为完全一致。entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等。Set

结构的实例默认可遍历，它的默认遍历器生成函数就是它的values方法。直接用for...of循环遍历 Set。Set 结构的实例与数组一样，也拥有forEach方法，用于对每个成员执行某种操作，没有返回值。另外，forEach方法还可以有第二个参数，表示绑定处理函数内部的this对象。

1. 遍历的应用

扩展运算符（...）内部使用for...of循环，所以也可以用于 Set 结构。

let set = new Set(['red', 'green', 'blue']);let arr = [...set];

// ['red', 'green', 'blue']

扩展运算符和 Set 结构相结合，就可以去除数组的重复成员。

let arr = [3, 5, 2, 2, 5, 5];let unique = [...new Set(arr)];

// [3, 5, 2]

而且，数组的map和filter方法也可以间接用于 Set 了。

let set = new Set([1, 2, 3]);set = new Set([...set].map(x => x \* 2));

// 返回Set结构：{2, 4, 6}let set = new Set([1, 2, 3, 4, 5]);set = new Set([...set].filter(x => (x % 2) == 0));

// 返回Set结构：{2, 4}

因此使用 Set 可以很容易地实现并集（Union）、交集（Intersect）和差集（Difference）。

let a = new Set([1, 2, 3]);let b = new Set([4, 3, 2]);

// 并集let union = new Set([...a, ...b]);

// Set {1, 2, 3, 4}

// 交集let intersect = new Set([...a].filter(x => b.has(x)));

// set {2, 3}

// （a 相对于 b 的）差集let difference = new Set([...a].filter(x => !b.has(x)));

// Set {1}

如果想在遍历操作中，同步改变原来的 Set 结构，目前没有直接的方法，但有两种变通方法。一种是利用原 Set 结构映射出一个新的结构，然后赋值给原来的 Set 结构；另一种是利用Array.from方法。

// 方法一let set = new Set([1, 2, 3]);set = new Set([...set].map(val => val \* 2));

// set的值是2, 4, 6

// 方法二let set = new Set([1, 2, 3]);set = new Set(Array.from(set, val => val \* 2));

// set的值是2, 4, 6

上面代码提供了两种方法，直接在遍历操作中改变原来的 Set 结构。

1. Map

（1）

1. 数组去重的方法

const arr = [1,2,3,3,3,4,5]

|  |
| --- |
| 1. const s = new Set(arr); //Set(5) {1, 2, 3, 4, 5}   [...s] // [1, 2, 3, 4, 5]  也可用于字符串去重 [...new Set('ababbc')].join('') // "abc"  2) function dedupe(array) {  return Array.from(new Set(array)); //Array.from可以将set结构转化为数组  }  dedupe([1, 1, 2, 3]) // [1, 2, 3] |

1. 单页面 与 多页面应用？

1）单页面：一个应用程序中，只有一个完整的HTML页面，其它所谓的页面，其实都是一些较大的组件而已。所谓的页面跳转，其实是更换不同的组件加载而已。

2）两者的区别？

（1）. 请求次数：

多页面应用，每更换一次页面都需要重新请求服务器，重新下载网页文件——增加了请求次数，增大了服务器的负担，同时也增加了用户等待

单页面应用，在首次请求页面时，就把所有的组件一起下载下来。今后，切换页面其实就是在客户端本地切换不同组件显示而已，不需要反复请求服务器——减少了请求次数，减轻了服务器的负担，避免了切换页面等待的事件，提高了用户体验。

（2）. 加载效率:

多页面应用，每更换一次页面都需要重建整棵DOM树，效率低！

单页面应用，因为只替换页面中部分内容，不需要重建整棵DOM树，所以效率高！

（3）. 公共资源: bootstrap.css bootstrap.js jquery.js

多页面应用，因为每个页面都包含<link><script>，所以，每更换一次页面都重新请求这些公共资源——也增加了请求次数，增大了服务器的负担，也增加了用户等待的时间

单页面应用，因为唯一完整的index.html文件只在首次记载时，下载一次，<link>和<script>也仅在首次加载时下载一次。之后更换页面组件，不会重新加载<link>和<script>——减少了请求次数，减轻了服务器的负担，避免了切换页面等待的时间，提高了用户体验。

（4）. 页面过渡效果:

多页面应用不可能实现页面过渡效果，因为两个.html文件不可能并存在一个浏览器窗口内！

单页面应用，因为所有组件都是以元素方式，保存在客户端，所以，完全可以通过定位，CSS3变换的方式，实现页面组件之间的切换效果。

（5）. 首屏加载效率: ——SPA的缺点

多页面应用，首屏只需要下载第一个页面的内容，其它页面内容暂时不需要下载，所以，首屏加载快！

单页面应用，首屏就要把所有组件内容都下载下载，所以首屏加载慢——已解决: 配合打包工具webpack，让组件的js文件异步延迟加载: 先加载首屏组件用着，其它组件js在后台异步下载，不影响首屏下载速度。

（补充）**首屏加载慢**的解决方案

|  |
| --- |
| 1. 首屏时间   首屏：首页打开完成，第一个可视区域的所有内容。（1.5s左右加载完最好，）   1. 网站**输入url**到实现页面跳转经历了哪些事？   协议及端口号  （1）浏览器输入url点击跳转或回车  （2）浏览器查找当前URL是否存在缓存，并比较缓存是否过期  （3）DNS解析当前URL对应的IP  （4）根据IP建立TCP连接（三次握手）   1. http发起请求 2. 服务器处理请求，浏览器接收http响应 3. 渲染页面，构建dom树 4. 关闭TCP连接（四次挥手） 5. 计算首屏加载时间   （1）标签标记法：在不考虑下拉获取新数据和图片加载的情况下  window.fristScreen = Date.now();  首屏加载时间 = fristScreen - performance.timing.navigationStart   1. 解决首屏加载慢（差不多就是前端性能优化） |

1. Vue UI组件库
2. 移动端

Vant ,Cube UI ,Mint UI

1. pc端

Element UI , IView UI

1. $router与$route的区别
2. $router

VueRouter的实例，可以导航到不同的url，常见的$router.push(),$router.replace()

1. $route

为当前 router 跳转对象里面可以获取 name 、 path 、 query 、 params 等

26.

1. ：Vue3
2. ：常见的js笔试题
   * + 1. 深克隆与浅克隆
3. 浅克隆：只克隆第一级属性，若某个属性又是一个内嵌的子对象，不会进入子对象克隆子对象的内容

//1.浅克隆

function shallowClone(oldObj){

// 判断其传入的oldObj的类型：防 null 数值 数组

if(oldObj === null){

return null; //原路返回 退出函数

}

if(typeof oldObj !== 'object'){ //对象和数组 typeof 都是object

return oldObj;

}

var newObj = Array.isArray(oldObj)? []:{} //判断传入的是否是数组

for (var key in oldObj) {

newObj[obj] = oldObj[obj];

}

return newObj;

}

1. 深克隆：既可以克隆第一级属性，若某个属性又是一个内嵌的子对象，还会进入子对象克隆子对象的内容.(递归)

// 深克隆

function deepClone(oldObj){

if (oldObj === null) {

return null

};

if(typeof oldObj !== 'objecct'){

return oldObj

};

var newObj = Array.isArray(oldObj)? []:{};

for (var key in oldObj) {

newObj[key] = deepClone(oldObj[key]);

};

return newObj;

}

补充：for of可以遍历数字下标的数组，类数组对象，字符串中的每个字符。

for in 专门遍历自定义下标的关联数组和对象。

* + - 1. 判断一个对象是否为数组（7种）

var obj = [1, 2, 4];

console.log(obj.\_\_proto\_\_ == Array.prototype)

console.log(Object.getPrototypeOf(obj) == Array.prototype);

console.log(Array.prototype.isPrototypeOf(obj));

console.log(obj.constructor == Array);

console.log(Array.isArray(obj));

// 当强行给该判断对象更换了父对象，可以用以下两种方法判断

console.log(obj instanceof Array);

console.log(Object.prototype.toString.call(obj) == '[object Array]');

* + - 1. typeof可以判断哪些？

几乎所有值类型都可以, 特殊的 typeof null // ‘object’ typeof NaN // ‘number’

typeof undefined // ‘undefined’ 也可以识别function

* + - 1. js里面的数据类型？

1)原始数据类型：number，string，boolean，null（空指针对象），undefined，symbol（es6）,bigint（es10）

2)引用数据类型：object（包含了 array，object，function）

补充

3)es6 symbol:对象的属性名容易产⽣命名冲突，为保证键名的唯⼀性，故es6引⼊Symbol这种新的原始数

据类型，确保创建的每个变量都是独⼀⽆⼆的。特点：类似于字符串的数据类型；不可用new创建，可穿参数

|  |
| --- |
| 1. 定一个对象的属性名   //定义个对象的属性名 方法一  let a3 = Symbol.for("kk")  let obj = {  kk : 123,  a3 : 123,  [a3] : 123  }  console.log(obj) // { kk: 123, a3: 123, [Symbol(kk)]: 123 }  // 定义个对象的属性名 方法二  let obj2 = {}  obj2[a3] = 4  console.log(obj2) //{ [Symbol(kk)]: 4 }  // 定义个对象的属性名 方法三  Object.defineProperty( obj, a3, {value: 456})  console.log(obj) //{ kk: 123, a3: 123, [Symbol(kk)]: 456 }   1. 定义一个字符串做常量   // 当做字符串常量使用  const name = Symbol('Coco')  console.log(name) //Symbol(Coco) |

3)es10 中bigint:BigInt是一种新的数据类型，用于当整数值大于Number数据类型支持的范围时。这种数据类型允许我们安全地对大整数执行算术操作，表示高分辨率的[时间戳](https://so.csdn.net/so/search?q=%E6%97%B6%E9%97%B4%E6%88%B3&spm=1001.2101.3001.7020" \t "/Users/wang/Documents\\x/_blank)，使用大整数id，等等，而不需要使用库。BigInt不支持一元加号运算符，BigInt可以正常地进行位运算，如|、&、<<、>>和^

|  |
| --- |
| console.log(99999999999999999999) //100000000000000000000  解决：i.数值后面加n  console.log(99999999999999999999n) //99999999999999999999n  Ii. 构造函数  BigInt('99999999999999999999') // 99999999999999999999n |

* + - 1. === == ！== ！=

1）== 只判断值是否相等（里面包含隐性转换）

2）=== 值与类型 都必须一样

* + - 1. var 、 const 、 let 的区别

1. var 声明提前，赋值在后；在全局作用域创建；可重复声明同一个变量名，以最后一个赋值的为准
2. const 块级作用域，声明常量，必须有初始值，不可重复赋值，一个作用域内标识符不可一致，若值为对象则对象的所有属性可以修改
3. let 块级作用域，阻止声明提前（与var 的区别就是其实质是匿名函数自调），同一个作用域，let不能重复定义两个同名的标识符

* + - 1. 节流和防抖？ lodash插件

1. 节流：\_.throttle(func, [wait=0], [options=]) 在规定的间隔时间内不会重复触发回调，只有大于这个时间间隔才会触发回调，把频繁触发变为少量触发，也就是说如果连续快速的触发 只会执行一次

|  |
| --- |
| 例如：商城菜单的三级联动，鼠标进入全部商品分类，分类栏显示（过渡动画transition【包裹的该元素或组件必须有v-show或v-if】，name属性为sort，鼠标进入sort-enter过渡动画进入的开始状态，高度为0px,sort-enter-to进入结束状态高度为分类栏总高度，sort-enter-active整体动画的时间wei 0.5s 匀速） |

2）防抖\_.debounce(func, [wait=0], [options=]) 前面的所有的触发都被取消，最后一次执行在规定的时间之后才会触发

·总结： 用户操作很频繁，防抖只是执行一次，节流是把频繁的操作变为少量操作

* + - 1. 事件处理（事件委托）

|  |
| --- |
| 就是在原本绑定在子元素身上的事件绑定在父元素身上（利用的是dom元素的事件冒泡）  适用于：多个平级子元素都要绑定相同的事件时，只要将事件在父元素绑定一次，所有子元素都可通过冒泡原理共享父元素的事件使用！  优点：   1. 尽量减少事件监听个数；因为: 浏览器触发事件时采用的是遍历队列的方式。队列中监听对象多，遍历查找的速度就可能慢，事件相应的速度就可能延迟。队列中监听对象少，遍历查找的速度就快，事件响应的速度就快！   2.即使动态添加的元素，也可自动获得处理函数  dom事件：一个事件触发后有三个阶段。   1. 捕获：有外向内，只获取记录各父元素身上的事件但不触发； 2. 目标触发：先触发目标元素身上的处理函数； 3. 冒泡：由内到外，逐级触发捕获阶段获取到的各级父元素的处理函数   取消冒泡：e.stopPropagation(); |

* + - 1. Event loop（事件循环） ?

1. 实质：是计算机系统的一种运行机制

javascript语言是单线程，作为浏览器的脚本语言，主要用途是与用户会动，以及操作dom。HTML5提出Web Worker标准，允许JavaScript脚本创建多个线程，但是子线程完全受主线程控制，且不得操作DOM。

2）任务列队：分为同步任务（主线程上排队执行）和异步任务（不进入主线程而进入"任务队列"（task queue）的任务，只有"任务队列"通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

3）event loop：主线程从"任务队列"中读取事件，这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）。

* + - 1. 同步与异步

1. 同步：js是单线程，同时只能做一件事。按照顺序做每一件事。JS引擎中负责解释和执行JavaScript代码的线程只有一个。我们叫它**主线程**。
2. 异步：不完全按照顺序执行；主线程发起一个异步请求，相应的工作线程接收请求并告知主线程已收到(异步函数返回)；主线程可以继续执行后面的代码，同时工作线程执行异步任务；工作线程完成工作后，通知主线程；主线程收到通知后，执行一定的动作(调用回调函数)。（ajaxi请求，dom事件，定时器）
   * + 1. 调用构造函数class的new做了哪些事？ var obj = new 类型名(属性值1,....)
3. 创建了一个新对象
4. 让子对象的 \_\_proto\_\_指向构造函数(constructor)的原型对象（prototype）

3）用new调用构造函数。将构造函数中的this，都吸引到new上！然后通过强行赋值的方式！给新对象new，添加规定好的属性！凡是构造函数中规定的属性和方法，最终都会添加到孩子身体里！

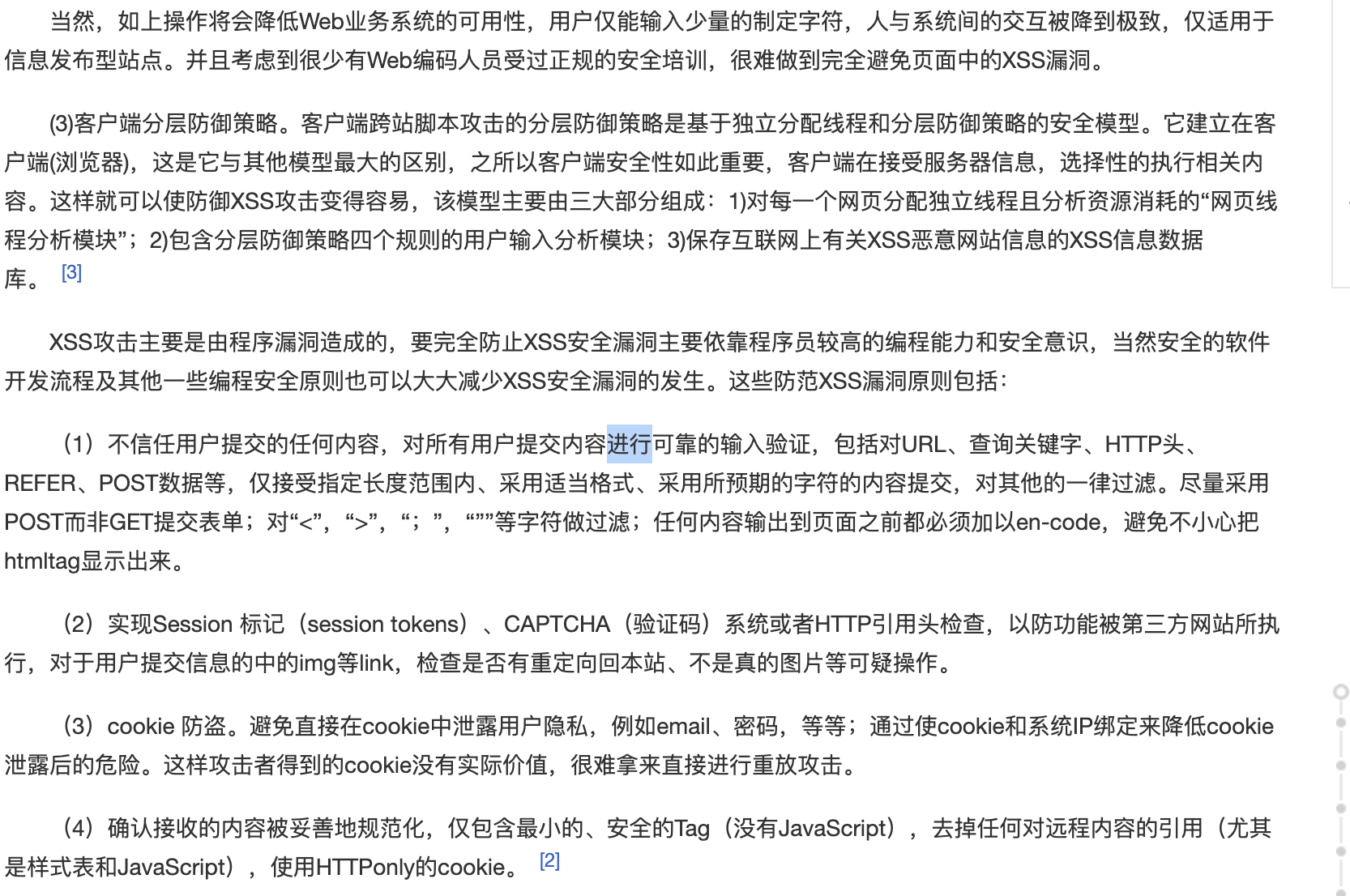
4）返回新对象的地址给变量保存起来。

* + - 1. Vue的data为什么是一个函数？

vue中data必须是函数是为了保证组件的独立性和可复用性，data是一个函数，组件实例化的时候这个函数将会被调用，返回一个对象，计算机会给这个对象分配一个内存地址，你实例化几次，就分配几个内存地址，他们的地址都不一样，所以每个组件中的数据不会相互干扰，改变其中一个组件的状态，其它组件不变。

* + - 1. 纯函数？

一个函数的返回结果只依赖其参数，并且执行过程中没有副作用。副作用（不会改变值  
14.xss攻击(跨站脚本攻击)?

1. 通过利用[网页](https://baike.baidu.com/item/%E7%BD%91%E9%A1%B5/99347" \t "/Users/wang/Documents\\x/_blank)开发时留下的漏洞，通过巧妙的方法注入恶意指令代码到网页，使用户加载并执行攻击者恶意制造的网页程序。攻击成功后，攻击者可能得到包括但不限于更高的权限（如执行一些操作）、私密网页内容、会话和cookie等各种内容。
2. 防御方法：
3. 优化Web应用开发来减少漏洞，避免被攻击:1)用户向服务器上提交的信息要对URL和附带的的HTTP头、POST数据等进行查询，对不是规定格式、长度的内容进行过滤。2)实现Session标记（session tokens）、CAPTCHA系统或者HTTP引用头检查，以防功能被第三方网站所执行。3)确认接收的的内容被妥善的规范化，仅包含最小的、安全的Tag（没有javascript），去掉任何对远程内容的引用（尤其是样式表和javascript），使用HTTP only的cookie。
4. 
5. js的几种设计模式 12种
6. 工厂模式
7. 单例模式
8. 原型模式
9. 适配器模式
10. 代理模式
11. 策略模式
12. 迭代器模式
13. 观察者模式
14. 发布-订阅模式
15. 命令模式
16. 组合模式
17. 建造者模式
18. Js的继承中的继承方式？

|  |
| --- |
| 想要继承，就必须要提供个父类（继承谁，提供继承的属性）  IMG_256  **一、原型链继承**  IMG_257  　　　　重点：让新实例的原型等于父类的实例。  　　　　特点：1、实例可继承的属性有：实例的构造函数的属性，父类构造函数属性，父类原型的属性。（新实例不会继承父类实例的属性！）  　　　　缺点：1、新实例无法向父类构造函数传参。  　　　　　　　2、继承单一。  　　　　　　　3、所有新实例都会共享父类实例的属性。（原型上的属性是共享的，一个实例修改了原型属性，另一个实例的原型属性也会被修改！）  **二、借用构造函数继承**  IMG_258  　　　　重点：用.call()和.apply()将父类构造函数引入子类函数（在子类函数中做了父类函数的自执行（复制））  　　　　特点：1、只继承了父类构造函数的属性，没有继承父类原型的属性。  　　　　　　　2、解决了原型链继承缺点1、2、3。  　　　　　　　3、可以继承多个构造函数属性（call多个）。  　　　　　　　4、在子实例中可向父实例传参。  　　　　缺点：1、只能继承父类构造函数的属性。  　　　　　　　2、无法实现构造函数的复用。（每次用每次都要重新调用）  　　　　　　　3、每个新实例都有父类构造函数的副本，臃肿。  **三、组合继承（组合原型链继承和借用构造函数继承）（常用**）  IMG_259  　　　　重点：结合了两种模式的优点，传参和复用  　　　　特点：1、可以继承父类原型上的属性，可以传参，可复用。  　　　　　　　2、每个新实例引入的构造函数属性是私有的。  　　　　缺点：调用了两次父类构造函数（耗内存），子类的构造函数会代替原型上的那个父类构造函数。  **四、原型式继承**  IMG_260  　　　　重点：用一个函数包装一个对象，然后返回这个函数的调用，这个函数就变成了个可以随意增添属性的实例或对象。object.create()就是这个原理。  　　　　特点：类似于复制一个对象，用函数来包装。  　　　　缺点：1、所有实例都会继承原型上的属性。  　　　　　　　2、无法实现复用。（新实例属性都是后面添加的）  **五、寄生式继承**  IMG_261  　　　　重点：就是给原型式继承外面套了个壳子。  　　　　优点：没有创建自定义类型，因为只是套了个壳子返回对象（这个），这个函数顺理成章就成了创建的新对象。  　　　　缺点：没用到原型，无法复用。  **六、寄生组合式继承（常用）**  　　　　寄生：在函数内返回对象然后调用  　　　　组合：1、函数的原型等于另一个实例。2、在函数中用apply或者call引入另一个构造函数，可传参  IMG_262  IMG_263  　　　　重点：修复了组合继承的问题  　　　　继承这些知识点与其说是对象的继承，更像是函数的功能用法，如何用函数做到复用，组合，这些和使用继承的思考是一样的。上述几个继承的方法都可以手动修复他们的缺点，但就是多了这个手动修复就变成了另一种继承模式。  　　　　这些继承模式的学习重点是学它们的思想，不然你会在coding书本上的例子的时候，会觉得明明可以直接继承为什么还要搞这么麻烦。就像原型式继承它用函数复制了内部对象的一个副本，这样不仅可以继承内部对象的属性，还能把函数（对象，来源内部对象的返回）随意调用，给它们添加属性，改个参数就可以改变原型对象，而这些新增的属性也不会相互影响。 |

1. ：常见的css面试题

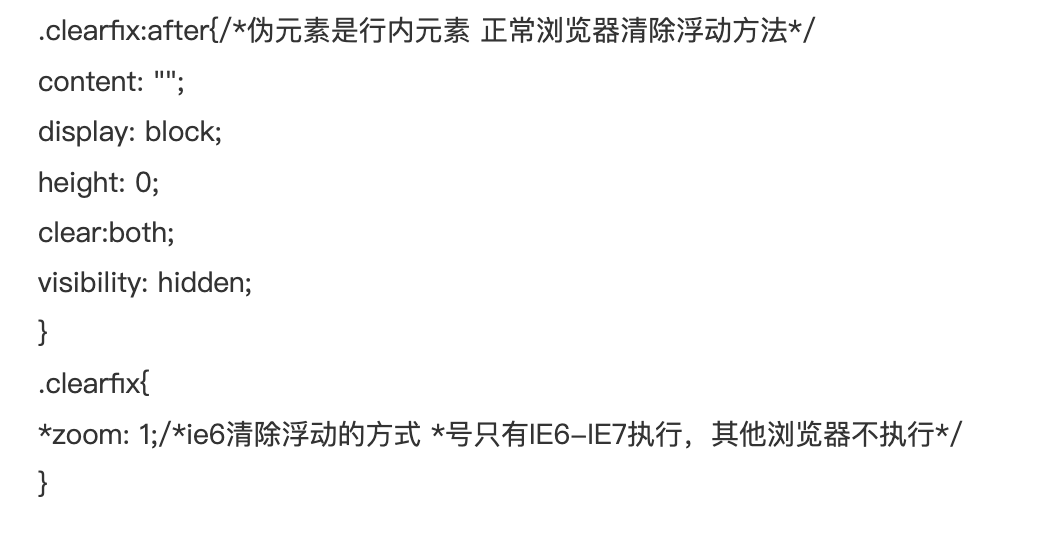
1.rem , em , vh/ vw ，px ,rpx

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| em：1em等于当前元素或父元素的font-size值  rem:1rem等于根元素的font-size值，（注意在使用rem布局时，要动态设置html根元素的font-size值，可使用js或者vw；同时要给body重置一下font-size：16px）  vw /vh :把屏幕分成100份，1vw等于屏幕宽度的1%   |  | | --- | | rem布局 **第一种方式** 通过js动态设置根元素的值: layout viewport: document.documentElement.clientWidth layout viewport（可视窗口）: document.documentElement.clientWidth              iphone 6 :375              iphone 6 plus : 414              思路  若设计稿：以 iphone 6 为准              iphone 6 ：document.documentElement.clientWidth / 3.75  -> 100              iphone 6 plus ：document.documentElement.clientWidth / 3.75  -> 110.4  var fontszie = document.documentElement.clientWidth / 3.75          console.log(fontszie)          // 动态为html根元素设置 fontsize          document.documentElement.style.fontSize = fontszie + 'px' | | **第二种方式** 通过 vw设置根元素的值:  vw：把屏幕分成 100份， 屏幕宽等于100vw          iphone 6 :375  1vw  -> 3.75          iphone 6 plus : 414  1vw -> 4.14          思路  若设计稿：以 iphone 6 为准 100px 设置多少个vw?  1vw / 3.75 = ? / 100px  -> 100 / 3.75 = 26.66667vw   <style>          html{ font-size:26.666667vw}          body{              font-size: 16px;          }          #box1{width: 1.2rem; height: 1.5rem; background: #aaffaa;}          #box2{width: 1rem; height: 1.5rem; background: #102eb1;}      </style> | | **总结**： 两种方式都需要 设置 body的font-size：16px | | **Vscode 插件 px to rem :**  设置： 用户-扩展-          alt+z 快捷键换算          批量换算  ctrl+a alt+z  **另一种rem适配的方法**  <script>        (function ( doc, win) {          var docEl = doc.documentElement,            resizeEvt = 'orientationchange' in window ? 'orientationchange' : 'resize'            recalc = function(){              var clientWidth = docEl.clientWidth              if(!clientWidth) return;              if(clientWidth>=750){//iPhone 6 的比例                docEl.style.fontSize = '100px'              }else{                docEl.style.fontSize = 100 \* (clientWidth / 750) + 'px'              }            }            if(!doc.addEventListener) return;            win.addEventListener(resizeEvt, recalc, false)            doc.addEventListener('DOMContentLoaded', recalc, false)          })(document,window)      </script>  **1rpx = 0.5px** | |

1. BFC？
2. 概念：block format context块级格式化上下文；一块独立的渲染区，内部元素的渲染不会影响边界外的元素
3. 触发的条件：浮动元素不是float：none）；  
    定位元素position:不是static和relative；  
    display：inline-block，table-celles,flex，inline-flex

overflow不是visible

1. 应用：
   1. 解决边距margin重叠问题，给子元素添加一个父元素overflow:hidden
   2. 清除浮动（左右子元素浮动后，父元素未设置高度会出现高度坍塌的现象，设置父元素overflow：hidden，计算bfc高度的时候，浮动元素也要参与计算）
2. 清除浮动？
   * 1. 在最后一个浮动元素的后面，添加额外的元素，clear:both；（不推荐）
     2. 给父元素添加overflow:hidden;(触发bfc，不推荐，当内容增多时，容易造成不会自动换行导致内容被隐藏掉)
     3. 给父元素添加:after伪元素content: "";display: block;height: 0;clear:both;visibility: hidden;



* + 1. 使用before和after双伪元素清除浮动



1. 盒子模型？
   * 1. 元素在页面实际占地的空间计算方式，分为标准盒模型和IE盒模型。
     2. 标准盒模型（w3c盒子模型)：根据 W3C 的规范，元素内容占据的空间是由 width 属性设置的，而内容周围的 padding 和 border 值是另外计算的；即在标准模式下的盒模型，盒子实际内容（content）的width/height=我们设置的width/height;盒子总宽度/高度=width/height+padding+border+margin。
     3. IE盒模型：浏览器的 width 属性不是内容的宽度，而是内容、内边距和边框的宽度的总和；即在怪异模式下的盒模型，盒子的（content）宽度+内边距padding+边框border宽度=我们设置的width(height也是如此)，盒子总宽度/高度=width/height + margin = 内容区宽度/高度 + padding + border + margin。
     4. 两种盒子的兼容性：不直接给元素添加具有指定宽度的内边距，而且尝试将内外边距添加到父元素和子元素
     5. box-sizing：属性指定哪种盒子
        + 1. 默认box-sizing：content-box；（定义的宽高为内容区域即标准盒子模型）

box-sizing：border-box；（定义的宽高包含border即IE盒子模型；一个容器内在同一行显示多个元素，若子元素的宽度使用百分比定义，基本就要使用border-box了。）

b) box-sizing: inherit;// 规定应从父元素继承 box-sizing 属性的值。

1. position定位？
   * 1. 默认定位 static 静态 默认文档流
     2. 相对定位 position:relative; 不会脱离文档流，相对于自己原始位置进行偏移，若不写偏移量则效果与没写定位一样，一般作为绝对定位元素的祖先元素
     3. 绝对定位 position:absolute; 脱离文档流

|  |
| --- |
| 绝对定位元素，如果祖先级没有已定位元素，那就相对于body左上角，执行偏移量  绝对定位元素，会相对于离自己最近的，祖先级，已定位元素的左上角，进行偏移  绝对定位，由于脱离文档流，产生如下效果  1.页面不占据空间，后续元素上前补位  2.绝对定位的元素，会变成块级  3.没有写宽度元素，发生绝对定位以后，宽度靠内容撑开 |

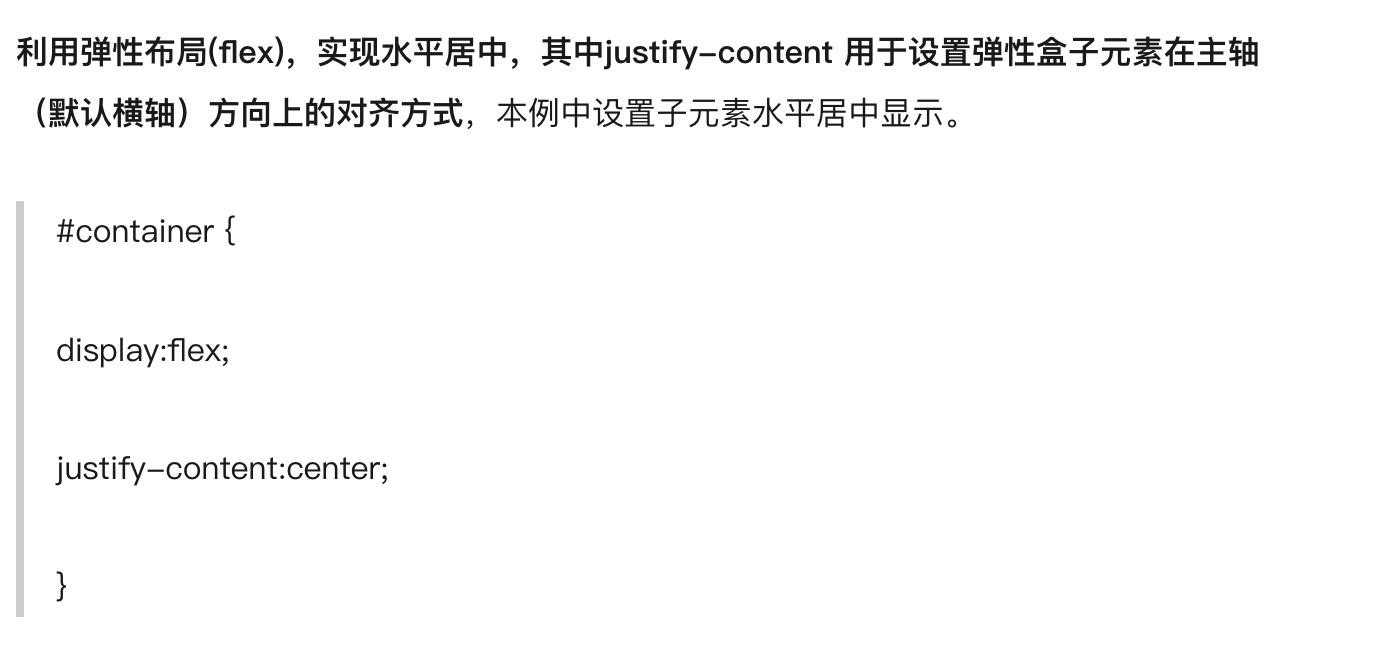
* + 1. 固定定位position:fixed;脱离文档流，位置始终相对于body初始化；将元素固定在页面某区域，不会随滚动条的变化而变化，一般固定在页面的可视区域内
    2. inherit 规定应该从父元素那继承position的属性
    3. 堆叠顺序

|  |
| --- |
| 注意：  1.默认的堆叠顺序，HTML元素后写的堆叠顺序高  2.浮动和定位的堆叠顺序不是同一个体系，尽量避免在一起处理  3.手动调整堆叠顺序  z-index：整数；  4.堆叠顺序对父子关系无效，儿子永远在爹的上面  5.只有已定位元素，可以设置堆叠顺序 |

1. 如何居中一个元素？
   * 1. 水平居中：
        1. **行内元素**水平居中：给父元素设置text-align：center；对此方法对inline、inline-block、inline-table和inline-flex元素水平居中都有效。
        2. 块级元素水平居中
           1. 有宽度：margin：0 auto
           2. table + margin：先将子元素设置为块级表格来显示（类似），再将其设置水平居中display:table在表现上类似block元素，但是宽度为内容宽。子元素=>display:table;margin:0 auto;
           3. absolute + transform:父元素相对定位，子元素绝对定位，向右移动子元素，移动距离为父元素容器的一半left:50%，最后再将子元素向左移动自己宽度的一半transform：translateX(-50%)---transform css3高版本浏览器要加前缀

|  |
| --- |
| .parent {  border: 1px solid red;  width: 500px;  height: 600px;  position: relative;  }  .child{  width: 100px;  height: 100px;  background: #000;  position: absolute;  left: 50%;  transform: translateX(-50%);  } |

* + - * 1. flex + justify-contect (css3 flex): 父元素display:flex; justify-content:center;
        2. flex + margin :父元素display:flex; 子元素 margin:0 auto；
      1. 多级块级元素水平居中
         1. flex :



* + - * 1. inline-block



* + - 1. 浮动元素水平居中
         1. 定宽 子元素relative（left：50%）+负margin（元素本事宽度一半）
         2. 不定宽 子元素 通过父子容器相对定位

.parent{position:relative;float:left;left:50%}

.child{position:relative;float:left;right:50%}

* + - * 1. flex 不管是否定宽与否



* + - 1. 定位元素水平居中 通过子元素绝对定位+margin:0 auto；

父元素 .parent{position:relative;} .child{position:absolute;margin:0 auto; left:0; right:0} /\*left right 不可删除即使为0\*/

* + 1. 垂直居中
       1. 内联元素垂直居中 line-height = height
       2. 多行内联元素垂直居中
          1. flex .parent{display:flex; flex-direction:column; justify-content:center;}
          2. 利用表布局 vertical-align: middle可以实现子元素的垂直居中 .parent{display:table,height:200px;} .child{display:table-cell;vertical:middle;}
       3. 块级元素垂直居中
          1. 已知高度 absolute + 负margin-top50% .child{position:absolute; top:50%;height:100px;margin-top:-50px;}
          2. 元素宽高不知道 可以借助CSS3中的transform属性向Y轴反向偏移50%的方法实现垂直居中。但是部分浏览器存在兼容性的问题。 .child{position:absolute; top:50%;transform:translateY(-50%)}
          3. 设置父元素的flex+align-items:center; .parent{display:flex;align-items:center;}
          4. 父元素 table-cell + vertical-align：middle；
    2. 水平垂直居中
       1. 已知宽高 绝对定位+ 负外边距 .parent{position:relative;} .child{width:100px;height:100px;position:absolute;top:50%;left:50%;margin:-50% 0 0 -50%;}
       2. 绝对定位+ transform .parent{position:relative;} .child{position:absolute;top:50%;left:50%;transform:translate(-50%,-50%)}
       3. 父元素（必须有高度）flex 利用flex布局，其中justify-content 用于设置或检索弹性盒子元素在主轴（横轴）方向上的对齐方式；而align-items属性定义flex子项在flex容器的当前行的侧轴（纵轴）方向上的对齐方式。.parent{height:100vh;display:flex;justify-content:center;align-items:center;}
       4. flex/grid与margin:auto(最简单写法) 父元素 display：flex或者grid； 子元素 margin:auto;

1. 文本溢出隐藏
   * 1. 单行文本： overflow:hidden; withe-space:nowrap;text-overflow:ellipsis;
     2. 多行文本：

overflow: hidden;

text-overflow: ellipsis;

display: -webkit-box;/\*对象作为伸缩盒子模型显示\*/

-webkit-box-orient: vertical;/\*设置或检索伸缩对象的子元素的排列方式\*/

/\*设置对其模式\*/

-webkit-line-clamp: 2;

/\*设置多行的行数\*/

1. Ajax
   * + 1. http与https的区别？