1.介绍上家公司项目

vue项目总结，项目期间遇到的问题、难点等。【不定期更新】

近期一直在做一个xxx中心的项目，先来吐槽下内心的想法，

要开发的项目需求很不明确，需求两周两周的更改，感觉每天并没有特别多实际的产出，总是感觉有点儿浪费时间。

虽然这样，但是作为开发，我们只能服从上级命令，好了，进入正题。

但是对我来说，感觉还好，需求的更改，对我来说又是新的开发，新的接触，同时也是在锻炼我的耐心和耐性。

第一次使用vue开发实际的项目，虽然之前也有写过，但是都是自己练手的，并没有实际的后端接口，这次对我自己来说也算是个小小的锻炼。项目开发过程中遇到的坑，以及bug，以及自己不会的地方挺多的，下面我就一一列举下。

（下面所有的问题，在我博客里面都会有）

（1）axios-请求中post请求的坑。刚开始的坑是，使用axios的post方法请求数据，数据被拦截，数据一直传不到后端那边。后来查文档才得知 axios对于post请求是有拦截功能的，需要自己判断，或者使用提请的 qs 方法，将传给后端的数据进行下处理。

（2）路由传参的功能的坑。之前一直使用路由传参，但是当本页面刷新的时候，页面上是没有参数的，因为参数是从上个页面传入进来的。 解决办法：使用了缓存，和vuex状态管理。但是由于项目并不是很大型的项目，所以使用最多的是缓存。

（3）页面缓存的坑。有个填写信息的页面，需要填写一部分信息，进入查新协议页面，返回的时候，页面上填写的信息还需要留存。 解决办法：使用vue提供的keep-alive，来完成页面的缓存的。

（4）vue组件动态加载的坑。由于首页的排版不确定，然后想着，让组件动态显示，根据后端传入的数据，传入那个组件的数据，就显示那个组件。解决办法：和后端商量好，做个标识。前端根据标识判断，动态显示组件。 使用到了vue中的<component :is=""></component> , 刚开始想着是不是和原生js一样使用append直接可以插入进入呢，但是后来发现根本不可以，思路是可以的，但是实现起来是行不通的。因为append后面插入的必须是个节点，而不是组件。后来就去查阅vue文档。

但是现在还有个问题，首页是通过动态组件添加的，数据得从后端接口返回，但是接口请求也是需要时间，所以，刚开始进入页面的是，页面先会是空白，但是这样的体验并不友好，会让用户感觉到页面就是一片空白，但是好的解决办法现在暂时没有想出来。好的解决办法还在寻找ing。【师傅有建议添加个加载的gif图，但是，感觉。。。，我在找下看还有别的办法没，实在没有的话，就只能添加个gif图了。】

（5）解析后端返回的map格式数据的坑。 之前解析数据的时候，直接就可以拿去，然后直接渲染页面使用即可。但是这次遇到后端返回的是map格式的数据，这就得解析下了。 例如：body['1'] 。根据返回的格式，自己解析成自己需要的数据格式。

（6）更新文件缓存的坑。每次打包好文件给后端更新的时候，用户手机上总会留下，上次版本的信息，而且每次都得清下缓存，才会显示最新版本的数据。后来，我师傅提了个建议，让后端返回一个更新版本的接口，前端每次更新版本的时候，都会给后端传入时间戳，然后后端接收后判断和库里的时间戳是否相同，相同的返回不需要更新，不相同的话，返回要更新，然后前端这边的处理方法是：需要更新的话，清除掉缓存，刷新页面即可。

虽然说给.js .css文件后缀加上时间戳也是可以的，但是页面的入口index.html每次都是一样的，所以。。。就不会更新，，百度一些说在nginx服务器上，写上强制更新，但是由于公司服务器上的文件很多，万一操作失误那就麻烦了。

（7）h5页面打开调试日志。h5页面不像小程序那样，直接可以打开控制台，在手机上查看日志，得需要自己安装vConsole的插件来实现。 详见博客：https://blog.csdn.net/Miss\_liangrm/article/details/100867112

（8）获取首页链接里面的参数问题。获取是可以获取到，只要不跳转出这个项目的页面，都是可以的，但是该项目链接了许多外链，所以，有时候返回的时候，页面就会显示空白，因为获取的参数出了问题。解决办法：将参数设置成了缓存，但是返回的速度快了，首页同样还是会出现拿不到参数，的问题。

解决办法还在寻找ing。

（9）h5里面的搜索。h5里面input实现在手机上按下“搜索”，“go”，“前往”等按钮的时候，同时会触发像PC端的Enter。 input标签需要设置属性：type="search" 。 详见博客 https://blog.csdn.net/Miss\_liangrm/article/details/102520925

（10）登录接口bug。需要判断 errCode 10001状态的情况。如果出现 errCode出现 10001，则清空原来的session，重新请求该网络请求。

（11）封装的请求方法不需要在传入相同的参数。 封装的方法，每个方法里面都得传入sessoin，但是里面需要有个版本的方法不需要传入session，那么就得在封装的方法里面进行判断。详见博客：https://blog.csdn.net/Miss\_liangrm/article/details/102593402

(config.method == 'get') {

console.log('config.params.version',config.params.version)

if(config.params.version == 'v'){ // 在更新版本的接口里面会用到

config.url = config.url

}else{

config.url = config.url + '?session='+localStorage.getItem('session');

}

}

（12）省份地区判断的话，尽量不使用name判断，会有bug的。通过市区匹配省份的话，使用areaCode，有些文件是不一样的。

（13）进来不在index.html文件里面引入公共的文件，因为每次更新版本的时候，index.html 都是相同的，如果修改了公共文件，是会有缓存，不会更新。因为该公共文件后面没有添加时间戳。

----------未结束。下次继续

待优化的：

首页空白。【已解决】【添加了loading】

页面更新版本缓存. 【已解决】（将获取连接的templateId写在Home页面）

2.用node.js做过项目么

3.对关系型数据库了解多少

# 4.移动端的布局方式

**常见的8种移动端界面布局方式包括8种：. 1、列表式布局. 2、宫格式布局. 3、仪表布局. 4、卡片布局. 5、瀑布流布局. 6、Gallery布局. 7、手风琴布局. 8、多面板布局**.

# 4.PC端的布局方式

**总共有6大 布局方式** **双飞翼、多栏、弹性、流式、瀑布流、响应式 布局**

# react生命周期

1、mounting: 插入真实 DOM </br>

2、updating: 正在被**重新**渲染 props 或 state 改变 </br>

3、unmounting: 卸载 移除真实 DOM

第一阶段：装载阶段3

constructor()

componentWillMount()

render()

componentDidMount()

第二阶段：更新阶段2

componentWillReceiveProps()

shouldComponentUpdate()

componentWillUpdate()

render()

componentDidUpdate()

第三阶段：卸载阶段1

componentWillUnmount()

​

constructor生命周期：

  (1)当react组件实例化时，是第一个运行的生命周期；

  (2)在这个生命周期中，不能使用this.setState()；

  (3)在这个生命周期中，不能使用副作用(调接口、dom操作、定时器、长连接等)；

  (4)不能把props和state交叉赋值；

​

componentDidMount生命周期：

  (1)相当于是vue中的mounted；

  (2)它表示DOM结构在浏览器中渲染已完成；

  (3)在这里可以使用任何的副作用；

​

shouldComponentUpdate(nextProps,nextState)生命周期：

  (1)相当于一个开关，如果返回true则更新机制正常执行，如果为false则更新机制停止；

  (2)在vue中是没有的；

  (3)存在的意义：可以用于性能优化，但是不常用，最新的解决方案是使用PureComponent；

  (4)理论上，这个生命周期的作用，用于精细地控制声明式变量的更新问题，如果变化的声明式变量参与了视图渲染则返回true，如果被变化的声明式变量没有直接或间接参与视图渲染，则返回false；

​

componentDidUpdate生命周期：

  (1)相当于vue中的updated()；

  (2)它表示DOM结构渲染更新已完成，只发生在更新阶段；

  (3)在这里，可以执行大多数的副作用，但是不建议；

  (4)在这里，可以使用this.setState()，但是要有终止条件判断。

​

componentWillUnmount生命周期：

  (1)一般在这里清除定时器、长连接等其他占用内存的构造器；

​

render生命周期：

  (1)render是类组件中唯一必须有的生命周期，同时必须有return（return 返回的jsx默认只能是单一根节点，但是在fragment的语法支持下，可以返回多个兄弟节点）；

  (2)Fragment碎片写法: <React.Fragment></React.Fragment> 简写成<></>；

  (3)return之前，可以做任意的业务逻辑，但是不能使用this.setState()，会造成死循环；

  (4)render()在装载阶段和更新阶段都会运行；

  (5)当render方法返回null的时候，不会影响生命周期函数的正常执行。

# Vue和React的区别

1，react整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入，所以在react中，是单向数据流；

2，vue的思想是响应式的，是基于是数据可变的，通过对每一个属性Watcher来监听，当属性变化的时候，响应式的更新对应的虚拟dom。

3，vue表单可以使用v-model支持双向绑定，相比于react来说开发更简单。

4，改变数据的方式不同。react需要通过setState

5，react使用jsx，有一定的上手成本，并且需要一整套工具链支持。vue使用模版语法，完全脱离工具链。

​

1. **react的优势**
2. React 速度很快：它并不直接对 DOM 进行操作，引入了一个叫做虚拟 DOM 的概念，安插在 javascript 逻辑和实际的 DOM 之间，性能好。
3. 跨浏览器兼容：虚拟 DOM 帮助我们解决了跨浏览器问题，它为我们提供了标准化的 API，甚至在 IE8 中都是没问题的。
4. 一切都是 component：代码更加模块化，重用代码更容易，可维护性高。
5. 单向数据流：Flux 是一个用于在 JavaScript 应用中创建单向数据层的架构，它随着 React 视图库的开发而被 Facebook 概念化。
6. 同构、纯粹的 javascript：因为搜索引擎的爬虫程序依赖的是服务端响应而不是 JavaScript 的执行，预渲染你的应用有助于搜索引擎优化。
7. 兼容性好：比如使用 RequireJS 来加载和打包，而 Browserify 和 Webpack 适用于构建大型应用。它们使得那些艰难的任务不再让人望而生畏。

# 6.你理解的闭包

**闭包是什么?**

**闭包(closure)就是能够读取其他函数内部变量的函数。**

**在javascript中，只有函数内部的子函数才能读取局部变量，所以闭包可以理解成 “定义在一个函数内部的函数”。**

**在本质上，闭包是将函数内部和函数外部连接起来的桥梁。(闭包的最典型的应用是实现回调函数（callback) )。**

**闭包的作用：**

**1. 在外部访问函数内部的变量**

**2. 让函数内的局部变量可以一直保存下去**

**3. 模块化私有属性和公共属性**

参数和变量不会被垃圾回收机制回收

**闭包的使用场景**

1.setTimeout

　　原生的setTimeout传递的第一个函数不能带参数，通过闭包可以实现传参效果。

2.回调

　　定义行为，然后把它关联到某个用户事件上（点击或者按键）。代码通常会作为一个回调（事件触发时调用的函数）绑定到事件。

3.函数防抖

 　　在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

　　 实现的关键就在于setTimeOut这个函数，由于还需要一个变量来保存计时，考虑维护全局纯净，可以借助闭包来实现。

4.封装私有变量

**闭包的原理：**

**全局变量生存周期是永久，局部变量生存周期随着函数的调用结束而销毁。**

**闭包就是 在函数中定义且成为该函数内部返回的函数的自由变量 的变量，该变量不会随着外部函数调用结束而销毁。**

**（注：不光是变量，函数内声明的函数也可以形成闭包）**

**当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。**

**闭包的this指向问题：**

**1.this指向window对象（因为匿名函数的执行具有全局性，所以其this对象指向window）；**

**2.不能实现value加1（每个函数在被调用时都会自动取得两个特殊变量，this和arguments，内部函数在搜索这两个对象时，只会搜索到其活动对象为止，所以不能实现访问外部函数的this对象）；**

**3.修改代码实现正确功能**

**第一种解决方法：**

**/定义变量that用于保存上层函数的this对象**

**第二种解决方法：**

**//使用apply改变helper的this对象指向，使其指向myNumber对象**

**第三种解决方法**

**//使用bind绑定，和apply相似，只是它返回的是对函数的引用，不会立即执行**

**闭包的应用场景：**

**// 1. 返回值 最常见的一种形式**

**// 2. 函数赋值 一种变形的形式是将内部函数赋值给一个外部变量**

**// 3. 函数参数 通过函数参数引用内部函数产生闭包**

**// 4. IIFE(自执行函数)**

**// 5. 循环赋值**

**// 6. getter和setter**

**// getter和setter函数来将要操作的变量保存在函数内部，防止暴露在外部**

**// 7.迭代器（计数器）**

**// 8.触发事件**

**简述闭包的问题以及优化**

**闭包的缺点：占用内层空间 大量使用闭包会造成 栈溢出**

**由于闭包会一直占用内存空间，直到页面销毁，我们可以主动将已使用的闭包销毁：**

**将闭包函数赋值为null 可以销毁闭包**

**JS中闭包的优缺点及特性**

**→ 优点：**

**1.保护函数内的变量安全**

**2.在内存中维持一个变量(用的太多就变成了缺点，占内存) ；**

**3. 逻辑连续，当闭包作为另一个函数调用的参数时，避免你脱离当前逻辑而单独编写额外逻辑。**

**4. 方便调用上下文的局部变量。**

**5. 加强封装性，可以达到对变量的保护作用。**

**→ 缺点：**

**1.常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。**

**2.还有有一个非常严重的问题，那就是内存浪费问题，这个内存浪费不仅仅因为它常驻内存，更重要的是，对闭包的使用不当会造成无效内存的产生。**

**→ 特性：**

**1. 函数嵌套函数**

**2. 内部函数可以访问外部函数的变量**

**3. 参数和变量不会被回收。**

**在哪里用到了闭包**

**1.匿名自执行函数**

**我们创建了一个匿名的函数，并立即执行它，由于外部无法引用它内部的变量，因此在函数执行完后会立刻释放资源，关键是不污染全局对象。**

**2.结果缓存**

**我们开发中会碰到很多情况，设想我们有一个处理过程很耗时的函数对象，每次调用都会花费很长时间，**

**那么我们就需要将计算出来的值存储起来，当调用这个函数的时候，首先在缓存中查找，如果找不到，则进行计算，**

**然后更新缓存并返回值，如果找到了，直接返回查找到的值即可。闭包正是可以做到这一点，因为它不会释放外部的引用，**

**从而函数内部的值可以得以保留。**

**3.封装**

**4.实现类和继承**

# ****8.你理解的JS的继承****

什么是继承？

简单说就是：在js中获取存在对象已有属性和方法的一种方式.下面我总结了几个继承：

一丶原型链继承

基本原理是：将父类的实例赋值给子类的原型。

这种继承方式的缺点：子类的实例可以访问父类的私有属性，子类的实例还可以更改该属性，这样不安全。

二丶构造函数继承

原理：在子类构造函数中，使用call来将子类的this绑定到父类中去

优点：

借用构造函数法可以解决原型中引用类型值被修改的问题；

缺点：

只能继承父对象的实例属性和方法，不能继承父对象原型属性和方法

三丶组合继承

将原型继承和借用构造函数两种方式组合起来

优点：

可以保证每个函数有自己的属性，可以解决原型中引用类型值被修改的问题；

子类的实例可以继承父类原型上面的属性和方法

缺点：

在实例化子类的过程中，父类构造函数调用了两次

四丶寄生组合式继承（推荐）

所谓寄生继承：通过 Object.create() 将子类的原型继承到父类的原型上。

特点：

这种方式的高效率体现在纸雕用了一次父类构造函数，并且因此避免了在父类的prototype 上面创建不必要的、多余的属性。

同时，原型链还能保持不变，可以正常使用 instanceof 和 isPrototypeOf 。

开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。

五丶class extend 继承

ES6 中有了类的概念，可以通过 class 声明一个类，通过 extends 关键字来实现继承关系。

class 与 ES5 构造函数的主要区别：

class 只能通过 new 来调用，而构造函数则可以直接调用；

class 内部所有定义的方法，都是不可枚举的（non-enumerable）

值得注意的是：

super 关键字表示父类的构造函数，相当于 ES5 的 Parent.call(this)。

子类必须在 constructor 方法中调用 super 方法，否则新建实例时会报错。这是因为子类没有自己的 this 对象，而是继承父类的 this 对象，然后对其进行加工，如果不调用 super 方法，子类就得不到 this 对象。

正是因为这个原因，在子类的构造函数中，只有调用 super 之后，才可以使用 this 关键字，否则会报错。

子类的 proto 属性指向父类因此，子类可以继承父类的静态方法。子类的原型的 **proto**，总是指向父类的 prototype 属性

# 9.JS的原型与原型链

你知道什么是原型吗？我们为什么要用原型呢？或者说原型为我们提供了什么？

什么是原型：

Javascript规定，每一个函数都有一个prototype对象属性，指向另一个对象（原型链上面的）。

prototype(对象属性)的所有属性和方法，都会被构造函数的实例继承。这意味着，我们可以把那些不变(公用)的属性和方法，直接定义在prototype对象属性上。

prototype就是调用构造函数所创建的那个实例对象的原型（proto）。

prototype可以让所有对象实例共享它所包含的属性和方法。也就是说，不必在构造函数中定义对象信息，而是可以直接将这些信息添加到原型中。

为什么要用原型：使用原型对象解决浪费内存

你了解原型链吗 你能说说 prototype 与 proto 的区别吗？

1.对象有属性\_\_proto\_\_,指向该对象的构造函数的原型对象。

2.方法除了有属性\_\_proto\_\_,还有属性prototype，prototype指向该方法的原型对象。

1. **react中不同组件之间如何做到数据交互？**
2. 父组件向子组件通信：使用 props
3. 子组件向父组件通信：使用 props 回调
4. 跨级组件间通信：中间组件层次传递props，使用 context 对象
5. 非嵌套组件间通信：使用事件订阅

# 13、说一下Redux

**Redux三大原则**

**1.单一数据源：**

**整个应用的 state 被储存在一棵对象结构中，并且这个对象结构只存在于唯一一个 store 中**

**2.State 是只读的：**

**redux中的state只读的不可以直接修改**

**3.使用纯函数(reducer)来执行修改state**

**为了修改了state数据，redux定义了一个reducer函数来完成state数据的修改，reducer会接收先前的 state 和 action，并返回新的 state**

**操作原理图**

**①、store通过reducer创建了初始状态**

**②、component通过store.getState()获取到了store中保存的state挂载在了自己的状态上**

**③、用户产生了操作，调用了actions 的方法**

**④、actions的方法被调用，创建了带有标示性信息的action（描述对象）**

**⑤、actions将action通过调用store.dispatch方法发送到了reducer中**

**⑥、reducer接收到action并根据标识信息判断之后返回了新的state**

**⑦、store的state被reducer更改为新state的时候，store.subscribe方法里的回调函数会执行，此时就可以通知component去重新获取state**

# 两边固定，中间自适应

## 绝对定位法

    绝对定位法原理是将左右两边使用absolute定位，因为绝对定位使其脱离文档流，后面的center会自然流动到他们上面，然后使用margin属性，留出左右元素的宽度，既可以使中间元素自适应屏幕宽度。

## 使用自身浮动法

自身浮动法的原理就是使用对左右使用分别使用float:left和float:right，float使左右两个元素脱离文档流，中间元素正常在正常文档流中，使用margin指定左右外边距对其进行一个定位。

# 圣杯布局

先定义好header和footer的样式，使之横向撑满。

在container中的三列设为浮动和相对定位(后面会用到)，center要放在最前面，footer清除浮动。

三列的左右两列分别定宽200px和150px，中间部分center设置100%撑满

这样因为浮动的关系，center会占据整个container，左右两块区域被挤下去了

接下来设置left的 margin-left: -100%;，让left回到上一行最左侧

但这会把center给遮住了，所以这时给外层的container设置 padding-left: 200px;padding-right: 150px;，给left和right空出位置

这时left并没有在最左侧，因为之前已经设置过相对定位，所以通过 left: -200px; 把left拉回最左侧

同样的，对于right区域，设置 margin-left: -150px; 把right拉回第一行

这时右侧空出了150px的空间，所以最后设置 right: -150px;把right区域拉到最右侧就行了。

**防抖和节流**

防抖是控制次数，节流是控制频率。

在前端开发的过程中，我们经常需要绑定一些持续触发的事件，而当一个函数（事件）被频繁的调用的时候，会造成浏览器卡顿的现象，这时就要用到防抖和节流。

防抖：

点击事件时，点多少次就会执行多少次，防抖就是让代码在最后一次点击的时候去执行一次。（点击结束之后去执行）

例：搜索框，鼠标滚动会用到防抖

节流：

在快速点击的过程中降低日至打印的频率。（设置多长时间执行一次）

节流解决：每次触发事件前都判断当前是否有等待执行的延时函数

# 11、深拷贝是什么？项目哪里是用到了深拷贝？

**浅拷贝：**

当对某个数据（数组或对象）进行拷贝后，修改新数据（拷贝后的数据）里面第1层的数据是不会影响老数据（被拷贝的数据）的，但是如果还要第2层 或 更深层次的数据（复杂数据类型），它仍然是有关联的，如果修改了新数据，那么老数据也会被修改。

深拷贝：

\* 就是在拷贝数据（数组或对象）时，不管数据里面有多少层，是简单 还是 复杂数据类型，只要进行深拷贝后，和老数据（之前被拷贝的数据）就毫无关联，相互独立，互不影响！在修改新数据，对老数据毫无影响。

\* 作用：打断两个对象或数组之间的引用关系！

**1，在拷贝构造函数中假如只完成了数据成员本身的赋值则称为“浅拷贝”；编译器提供的默认拷贝构造函数就已经可以完成这个任务。**

**而假如要复制的数据除了属性值本身以外，还要复制附加在数据属性值上的额外内容，那就要自己来写拷贝构造函数了，来完成所谓的“深拷贝”。**

**把一个对象里面的每一个成员都复制一份到新的对象里面**

**=> 当对象中的某一个成员是复杂数据类型的时候**

**=> 继续循环遍历这个复杂数据类型**

**=> 在新的对象里面也创建一个复杂数据类型继续复制进去**

**实现深拷贝:**

**通过递归函数来实现深拷贝**

**除了递归，我们还可以借用JSON对象的parse和stringify**

**deepCopy() 自定义方法**

**=> 在函数内部判断, 如果你是一个对象或者数组**

**=> 那么就再次调用函数**

**举个例子：**

**若在构造函数中new了一个新的空间存放数据，并且用指针记录了首地址；若是浅拷贝，则在拷贝构造函数中指针值将复制给另一个数据成员，这样就会有两个指针指向同一个空间；这样的话在析构函数里将会对指针所指向的空间进行释放，由于两个指针指向的是同一个空间，在释放第一个指针指向的空间时不会出现什么问题，而释放第二个指针指向的空间时就会因为空间已经被解析过而导致解析的空间不存在的情况，就会造成程序无法终止。**

**而解决上面这种情况的办法就是使用“深拷贝”，深拷贝是在拷贝构造函数里再new一个新的空间。将数据复制在新空间里，并将拷贝的指针记录这个新空间的首地址，这样在析构函数里就不会有问题了。**

**2，在某些引用类型值不更新的情况下用深拷贝**

# 17. ★★★ async 和 await 、promise的区别 和 这两个的本质

**/\*---------Promise概念：---------\*/**

**Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大，简单地说，Promise好比容器，里面存放着一些未来才会执行完毕（异步）的事件的结果，而这些结果一旦生成是无法改变的**

**/\*---------async await概念：---------\*/**

**async await也是异步编程的一种解决方案，他遵循的是Generator 函数的语法糖，他拥有内置执行器，不需要额外的调用直接会自动执行并输出结果，它返回的是一个Promise对象。**

**两者的区别：**

**Promise的出现解决了传统callback函数导致的“地域回调”问题，但它的语法导致了它向纵向发展行成了一个回调链，遇到复杂的业务场景，这样的语法显然也是不美观的。而async await代码看起来会简洁些，使得异步代码看起来像同步代码，await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，只有这一句代码执行完，才会执行下一句。**

**async await与Promise一样，是非阻塞的。**

**async await是基于Promise实现的，可以说是改良版的Promise，它不能用于普通的回调函数。**

# ****什么是Vue.nextTick()？？****

定义：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。

所以就衍生出了这个获取更新后的DOM的Vue方法。所以放在Vue.nextTick()回调函数中的执行的应该是会对DOM进行操作的 js代码；

理解：nextTick()，是将回调函数延迟在下一次dom更新数据后调用，简单的理解是：当数据更新了，在dom中渲染后，自动执行该函数，

# 12.事件循环（宏任务，微任务）

微任务和宏任务有什么区别？

|  | **宏任务（macrotask）** | **微任务（microtask）** |
| --- | --- | --- |
| 谁发起的 | 宿主（Node、浏览器） | JS引擎 |
| 具体事件 | 1. script (可以理解为外层同步代码) 2. setTimeout/setInterval 3. UI rendering/UI事件 4. postMessage，MessageChannel 5. setImmediate，I/O（Node.js） | 1. Promise 2. MutaionObserver 3. Object.observe（已废弃；Proxy 对象替代） 4. process.nextTick（Node.js） |
| 谁先运行 | 后运行 | 先运行 |
| 会触发新一轮Tick吗 | 会 | 不会 |

微任务先执行，宏任务后执行，微任务不会触发新一轮的tick，宏任务会触发新一轮的tick



# 13.浏览器渲染原理，什么情况下会触发重绘重排

**浏览器会把HTML解析成DOM，把CSS解析成CSSOM，DOM和CSSOM合并就产生了RenderTree(渲染树)。有了Render Tree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上。**

**DomTree用来存放元素，渲染树(Render Tree)用来展示元素相应的样式.**

**重排(回流)和重绘是什么？**

**1、回流：当Render Tree中部分或全部元素的尺寸、结构、或某些属性发生改变时，浏览器重新渲染部分或全部文档的过程称为回流。每个页面至少需要一次回流，就是在页面第一次加载的时候。**

**2、重绘：当页面中元素样式的改变并不影响它在文档流中的位置时（例如：color、background-color、visibility等），这些属性只是影响元素的外观，风格，而不会影响布局的，浏览器会将新样式赋予给元素并重新绘制它，这个过程称为重绘。**

**回流必将引起重绘，而重绘不一定会引起回流。**

**因为：如果一个元素引发了回流，它就影响了父级以及兄弟节点的位置，它们都要重新渲染了，所以回流必定引发重绘。**

**触发重排和重绘：**

**• 页面首次渲染**

**• 浏览器窗口大小发生改变**

**• 元素尺寸或位置发生改变**

**• 元素内容变化（文字数量或图片大小等等）**

**• 元素字体大小变化**

**• 添加或者删除可见的DOM元素**

**减少重绘和重排：**

**避免频繁的样式操作，最好一次性重写style，或者一次性更改class，避免频繁操作dom，对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。**

# 15.vue监听不到data中的数组变化的情况

对于普通的js对象

- Vue 将遍历此对象所有的 property，

- 并使用 Object.defineProperty 把这些 property 全部转为getter/setter，

- 使得vue可以在内部对这些数据进行追踪依赖， 在property被访问和修改时通知变更，重新渲染相关联的组件。

关于对象

Vue 不允许动态添加根级别的响应式 property。但是我们可以利用vue的内置方法，

- 使用Vue.set(object, propertyName, value)方法

- 该方法向嵌套对象添加响应式property，此时vue可以监听到对象的变化。

关于数组

Vue 不能检测以下数组的变动，因为这些都是不是响应式的。

- 如利用索引直接设置一个数组项，即使用下标修改某个元素（vm.arr[indexOfItem] = newValue）

- 直接修改数组的长度（vm.arr.length = newLength）

# 16.虚拟dom的理解

**虚拟 dom 和原生 dom**

**/\***

**（1）原生dom是浏览器通过dom树渲染的复杂对象，属性非常多；**

**（2）虚拟dom是存在于内存中的js对象，属性远少于原生的dom对象，它用来描述真实的dom，并不会直接在浏览器中显示；**

**（3）原生dom操作、频繁排版与重绘的效率是相当低的，虚拟dom则是利用了计算机内存高效的运算性能减少了性能的损耗；**

**（4）虚拟DOM进行频繁修改，然后一次性比较并修改真实DOM中需要改的部分，最后并在真实DOM中对修改部分进行排版与重绘，减少过多DOM节点排版与重绘损耗**

**\*/**

**虚拟DOM的优劣如何？实现原理？**

**/\***

**虚拟dom是用js模拟一颗dom树,放在浏览器内存中，相当于在js和真实dom中加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。**

**优点：**

**（1）虚拟DOM具有批处理和高效的Diff算法,最终表现在DOM上的修改只是变更的部分，可以保证非常高效的渲染,优化性能；**

**（2）虚拟DOM不会立马进行排版与重绘操作，对虚拟DOM进行频繁修改，最后一次性比较并修改真实DOM中需要改的部分；**

**（3）虚拟 DOM 有效降低大面积真实 DOM 的重绘与排版，因为最终与真实 DOM 比较差异，可以只渲染局部；**

**缺点：**

**（1）首次渲染大量DOM时，由于多了一层虚拟DOM的计算，会比innerHTML插入慢；**

**React组件的渲染过程：**

**（1）使用JSX编写React组件后所有的JSX代码会通过Babel转化为 React.createElement执行;**

**（2）createElement函数对 key和 ref等特殊的 props进行处理，并获取 defaultProps对默认 props进行赋值，并且对传入的子节点进行处理，最终构造成一个 ReactElement对象（所谓的虚拟 DOM）。**

**（3）ReactDOM.render将生成好的虚拟 DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实 DOM。**

**虚拟DOM的组成——ReactElementelement对象结构：**

**（1）type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）**

**（2）key：组件的唯一标识，用于Diff算法，下面会详细介绍**

**（3）ref：用于访问原生dom节点**

**（4）props：传入组件的props，chidren是props中的一个属性，它存储了当前组件的孩子节点，可以是数组（多个孩子节点）或对象（只有一个孩子节点）**

**（5）owner：当前正在构建的Component所属的Component**

**（6）self：（非生产环境）指定当前位于哪个组件实例**

**（7）\_source：（非生产环境）指定调试代码来自的文件(fileName)和代码行数(lineNumber)**

**\*/**

# 10、v-if和v-show的区别

1、v-if只有在判断为true的时候才会对数据进行渲染,false的时候把包含的代码进行删除。除非再次进行数据渲染，v-if才会重新判断。可以说是用法比较倾向于对数据一次操作。

2、v-show是无论判断是什么都会先对数据进行渲染，只是false的时候对节点进行display.none;的操作。所以再不重新渲染数据的情况下，改变数据的值可以使数据展示或隐藏。

使用 频繁切换时用v-show，运行时较少改变时用v-if

（从Vue源码的角度上来看，v-if的判断应该是发生在template编译成render function的过程中）

# 11、vue的生命周期，什么时候使用，哪个使用的比较频繁

**常用的生命周期有，beforeCreate，created，beforeMount，mounted，beforeUpdate，updated，beforeDestroy，destroyed**

**你的接口请求一般放在哪个生命周期中？**

**接口请求一般放在mounted中，但需要注意的是服务端渲染时不支持mounted，需要放到created中。**

# 7、 怎么理解MVVM

对MVC，MVP，MVVM的理解

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 。

MVVM 是 Model-View-ViewModel 的缩写。mvvm 是一种设计思想。

1：Model 层代表数据模型，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 是一个同步 View 和 Model 的对象。

2：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。

3：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。

8、 Vue是工作中学的还是大学课程

工作中学的 大学哪讲这些啊 大学静的都是html、css写静态页面 用js渲染都没讲的太深

# 10、v-if和v-show的区别

1、手段：v-if是动态的向DOM树添加或者删除DOM元素；v-show是通过设置DOM元素的display样式属性控制显示和隐藏。

2、编译过程：v-if切换有一个局部编译/卸载的过程， 切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换。

3、编译条件：v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译（编译被缓存？编译被缓存后，然后再切换的时候进行局部卸载); v-show是在任何条件下（首次条件是否为真）都被编译，然后被缓存，而且DOM元素保留

4、性能消耗：v-if有更高的切换消耗；v-show有更高的初始渲染消耗 使用场景：v-if适合运营条件不大可能改变；v-show适合频繁切换

6、相同点： v-show 都可以动态控制着dom元素的显示隐藏 不同点： v-if 的显示隐藏是将DOM元素整个添加或删除，v-show 的显示隐藏是为DOM元素添

7、加css的样式display，设置none或者是block，DOM元素是还存在的

8、在渲染多个元素的时候，可以把一个 元素作为包装元素，并使用v-if 进行条件判断，最终的渲染不会包含这个元素，v-show是不支持 语法

# 15、vue中用过哪些性能优化，遇到的性能问题一般怎么解决，前端展示后台数据慢的问题怎么解决方案

**1、首屏加载优化**

**2、路由懒加载**

**3、开启服务器 Gzip**

**4、启动 CDN 加速**

**5、代码层面优化**

**6、Webpack 对图片进行压缩**

**7、避免内存泄漏**

**8、减少 ES6 转为 ES5 的冗余代码**

# 16、怎么解决跨域问题

**在AJAX应用环境中，由于安全的原因，浏览器不允许XMLHttpRequest组件请求跨域资源。在很多情况下，这个限制给我来带来的诸多不 便。很多同行，研究了各种各样的解决方案：**

**1. 通过修改document.domain和隐藏的IFrame来实现跨域请求。这种方案可能是最简单的一种跨域请求的方案，但是它同样是一种限制最大的方 案。首先，它只能实现在同一个顶级域名下的跨域请求；另外，当在一个页面中还包含有其它的IFrame时，可能还会产生安全性异常，拒绝访问。**

**2.通过请求当前域 的代理，由服务器 代理去访问另一个域的资源。XMLHttpRequest通过请求本域内的一个服务器资源 ，将要访问的目标资源提供给服务器，交由服务器 去代理访问目标资源。这种方案，可以实现完全的跨域访问，但是开发，请求过程的消费会比较大。**

**3. 通过HTML中可以请求跨域资源的标签引用来达到目的，比如Image,Script,LINK这些标签。在这些标签中，Script无疑是最合适的。在 请求每一个脚本资源时，浏览器都会去解析并运行脚本文件内定义的函数，或需要马上执行的JavaScript代码，我们可以通过服务器返回一段脚本或 JSON对象，在浏览器解析执行，从而达到跨域请求的目的。使用script标签来实现跨域请求，只能使用get方法请求服务器资源。**

阿里问了ts与js的不同，hooks里的use effect的参数，react代码优化的方法，数组长度可以赋值吗，数组遍历方法有哪些以及区别，webpack的使用，箭头题怎么解决的，最近在看的技术书籍和新技术，umijs了解吗，git 里的cherry-pick用过吗，git平时用的指令，build后的上线流程了解吗

Arry.isarry 判断是否是数组

object.prototype.tostring.call(obj)==='[object object]' 判断是否是对象

清除浮动

content: ".";

clear: both;

height: 0;

visibility: hidden;

display: block;

数组里面是对象然后去重

# 简述cookie和localStorage以及sessionStorage的区别。

**关于cookie、sessionStorage、localStorage三者的区别主要有以下几点：**

**存储大小：cookie数据大小不能超过4k，sessionStorage和localStorage虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大**

**有效时间：localStorage存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；sessionStorage数据在当前浏览器窗口关闭后自动删除；cookie设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭**

**数据与服务器之间的交互方式，cookie的数据会自动的传递到服务器，服务器端也可以写cookie到客户端；sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存**

**antDesign 和 element-ui 哪个好用**

我更加倾向于elementUI,**UI**上更加漂亮，**使用**起来更加容易上手。 一开始，我最新接触的就是elementUI，感觉elementUI这个框架更加适合于面向外部开发。而作为对比的**Ant Design**，也有一定的优势。 从功能上来讲，后者更加齐全。

**遍历数组的方法**

**For、 while 、for…in 、for…of 、for…each 、 map、do…while**

为什么使用use Memo记忆组件

**set.state第二个参数是用来干什么的**

setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

git命令

**git config**。配置 Git 的相关参数。Git 一共有3个配置文件：1.仓库级的配置文件：在仓库的 ...

**git clone**。从远程仓库克隆一个版本库到本地。# 默认在当前目录下创建和版本库名相同的文件夹并 ...

**git init**。初始化项目所在目录，初始化后会在当前目录下出现一个名为 .git 的目录。# 初始化本地仓 ...

**git status**。查看本地仓库的状态。# 查看本地仓库的状态 $ git status # 以简短模式查看本地仓库的

git init # 初始化本地代码仓

git add . # 添加本地代码

git commit -m "add local source" # 提交本地代码

git pull origin master # 下载远程代码

git merge master # 合并master分支

git push -u origin master # 上传代码

**http状态码**

| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| --- | --- |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |
| 状态码 | 含义 |
| 100 | 客户端应当继续发送请求。这个临时响应是用来通知客户端它的部分请求已经被服务器接收， … |
| 101 | 服务器已经理解了客户端的请求，并将通过Upgrade ... |
| 102 | 由WebDAV（RFC ... |
| 200 | 请求已成功，请求所希望的响应头或数据体将随此响应返回。 ... |

400 Bad Request：表示请求报文中存在语法错误；

401 Unauthorized：未经许可，需要通过HTTP认证；

403 Forbidden：服务器拒绝该次访问（访问权限出现问题）

404 Not Found：表示服务器上无法找到请求的资源，除此之外，也可以在服务器拒绝请求但不想给拒绝原因时使用；

display有哪些属性

**3、数组和字符串的相互装换**

1.字符串变成数组,split

2.数组变字符串,join

# 11.你理解的Promise

**一、Promise是什么？**

**Promise是最早由社区提出和实现的一种解决异步编程的方案，比其他传统的解决方案（回调函数和事件）更合理和更强大。**

**ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。**

**ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。**

**二、Promise是为解决什么问题而产生的？**

**promise是为解决异步处理回调金字塔问题而产生的**

**三、Promise的两个特点**

**1、Promise对象的状态不受外界影响**

**1）pending 初始状态**

**2）fulfilled 成功状态**

**3）rejected 失败状态**

**Promise 有以上三种状态，只有异步操作的结果可以决定当前是哪一种状态，其他任何操作都无法改变这个状态**

**2、Promise的状态一旦改变，就不会再变，任何时候都可以得到这个结果，状态不可以逆，只能由 pending变成fulfilled或者由pending变成rejected**

**四、Promise的三个缺点**

**1）无法取消Promise,一旦新建它就会立即执行，无法中途取消**

**2）如果不设置回调函数，Promise内部抛出的错误，不会反映到 外部**

**3）当处于pending状态时，无法得知目前进展到哪一个阶段，是刚刚开始还是即将完成**

3、es6新增的语法**、数组方法，箭头函数等**

ES6新增的常用语法. **- let，const，export/import** (ES6的模块化)，class (类)

ES6 的新特性

const与let

ES6箭头函数

for...of循环

展开运算符

模板字符串

类和继承

解构赋值

对象简写法

剩余参数(可变参数)

参数默认值

模块化规范

# ES6 箭头函数和普通函数有什么差异？

1. 相比普通函数更简洁的语法

2. 没有this,捕获其所在上下文的 this 值，作为自己的 this 值

3. 不能使用new,箭头函数作为匿名函数,是不能作为构造函数的,不能使用new

4. 不绑定arguments，用rest参数...解决

let test3=(...a)=>{console.log(a[1])} //22

5. 使用call()和apply()调用:由于 this 已经在词法层面完成了绑定，通过 call() 或 apply() 方法调用一个函数时，只是传入了参数而已，对 this 并没有什么影响：

6. 箭头函数没有原型属性

7. 不能简单返回对象字面量

let fun5 = ()=>({ foo: x }) //如果x => { foo: x } //则语法出错

8. 箭头函数不能当做Generator函数,不能使用yield关键字

9. 箭头函数不能换行

let a = ()

=>1; //SyntaxError: Unexpected token =>

1. **用箭头函数会遇到哪些问题。**
2. this问题

箭头函数没有它自己的this值，箭头函数内的this值继承自外围作用域，谁定义的函数，this指向谁

箭头函数要实现类似纯函数的效果，必须剔除外部状态。所以箭头函数不具备普通函数里常见的 this、arguments 等，当然也就不能用 call()、apply()、bind() 去改变 this 的指向

对于箭头函数来说，并没有自己的 this ，它的 this 将始终指向让它生效的对象，即它的外部调用者：

1. arguments 关键字

在函数中，可以通过 arguments 关键字来获取到当前函数中传入的参数，但是在箭头函数中是没有这个关键字的，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。

不可以使用yield命令，因此箭头函数不能用作 Generator 函数。

不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误。

**4.用到哪些方法实现路由懒加载。**

什么是路由懒加载？

也叫延迟加载，即在需要的时候进行加载，随用随载。

1

官方解释：

1：当打包构建应用时，JavaScript 包会变得非常大，影响页面加载。

2：如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了。

1

2

官方在说什么呢？为什么需要懒加载？

1：首先，我们知道路由中通常会定义很多不同的页面。

2：这个页面这项目build打包后，一般情况下，会放在一个单独的js文件中

3：但是，如果很多的页面都放在同一个js文件中，必然会造成这个页面非常大

4：如果我们一次性的从服务器中请求下来这个页面，可能会花费一定时间，用户体验不好

5：如何避免这种情况发生呢？使用路由懒加载就可以了

1

2

3

4

5

继续解释原由？

1：像vue这种单页面应用，如果没有应用懒加载，运用webpack打包后的文件将会异常的大。

2：造成进入首页时，需要加载的内容过多，时间过长，会出啊先长时间的白屏，即使做了loading也是不利于用户体验。

3：而运用懒加载则可以将页面进行划分，需要的时候加载页面，可以有效的分担首页所承担的加载压力，减少首页加载用时

1

2

3

也就是说：进入页面不用也不需要一次性加载过多资源造成加载时间过程！

路由懒加载做了什么事情？

1：主要作用是将路由对应的组件打包成一个个的js代码块

2：只有在这个路由被访问到的时候，才加载对应的组件，否则不加载！

1

2

即：只有在这个路由被访问到的时候，才加载对应的组件，否则不加载！

1:.如何实现路由懒加载？？

vue项目实现路由按需加载（路由懒加载）的三种方式：

1：Vue异步组件

2：ES6标准语法import（）---------推荐使用！！！！！

3：webpack的require，ensure()

1

2

3

4

2.Vue异步加载技术

1：vue-router配置路由，使用vue的异步组件技术，可以实现懒加载，此时一个组件会生成一个js文件。

2：component: resolve => require(['放入需要加载的路由地址'], resolve)

1

5

3.ES6推荐方式imprort ()----推荐使用

1：直接将组件引入的方式，import是ES6的一个语法标准，如果需要浏览器兼容，需要转化成es5的语法。

2：推荐使用这种方式，但是注意wepack的版本>2.4

3：vue官方文档中使用的也是import实现路由懒加载

4：上面声明导入，下面直接使用

4.webpack提供的require.ensure()实现懒加载：

1：vue-router配置路由，使用webpack的require.ensure技术，也可以实现按需加载。

2：这种情况下，多个路由指定相同的chunkName，会合并打包成一个js文件。

3：require.ensure可实现按需加载资源，包括js,css等。他会给里面require的文件单独打包，不会和主文件打包在一起。

4：第一个参数是数组，表明第二个参数里需要依赖的模块，这些会提前加载。

5：第二个是回调函数,在这个回调函数里面require的文件会被单独打包成一个chunk,不会和主文件打包在一起，这样就生成了两个chunk,第一次加载时只加载主文件。

6：第三个参数是错误回调。

7：第四个参数是单独打包的chunk的文件名

4.import和require的比较（了解）

1：import 是解构过程并且是编译时执行

2：require 是赋值过程并且是运行时才执行，也就是异步加载

3：require的性能相对于import稍低，因为require是在运行时才引入模块并且还赋值给某个变量

**5.你接触过哪些浏览器，出现过哪些兼容问题，css兼容，event等等**

**一、css兼容**

**1. 不同浏览器的标签默认的margin和padding不同**

**问题症状： 随便写几个标签，不加样式控制的情况下，各自的margin 和padding差异较大。**

**碰到频率: 100%**

**解决方案：**

**CSS里 \*{margin:0;padding:0;} 但是性能不好**

**一般我们会引入reset.css样式重置；**

**2. css3新属性，加浏览器前缀兼容早期浏览器**

**-moz- /\* 火狐浏览器 /**

**-webkit- / Safari, 谷歌浏览器等使用Webkit引擎的浏览器 /**

**-o- / Opera浏览器(早期) /**

**-ms- / IE \*/**

**哪些css3属性需要加：**

**定义关键帧动画 @keyframes**

**css3中的变形（transform）、过渡(transtion)、动画(animation)**

**border-radius 圆角**

**box-shadow 盒子阴影**

**flex 弹性布局**

**3. 块属性标签float后，又有横行的margin情况下，IE 浏览器margin加倍的问题**

**问题症状: 常见症状是IE6中后面的一块被顶到下一行**

**解决方案： 设置为float的div在ie下设置的margin会加倍。这是一个ie6都存在的bug。解决方案是在这个div里面加上display:inline;**

**4. 设置较小高度标签（一般小于10px），在IE6，IE7，遨游中高度超出自己设置高度**

**问题症状： 设置div高度小于10px，IE6、7和遨游里div的高度，超出自己设置的10px.**

**给超出高度的标签设置overflow:hidden;**

**或者设置行高line-height 小于你设置的高度。**

**5. 行内属性标签，设置display:block后采用float布局，又有横行的margin的情况，IE6间距bug**

**问题症状： IE6里的间距比超过设置的间距**

**解决方案： 在display:block;后面加入display:inline;display:table;**

**6. IE浏览器div最小宽度和高度的问题**

**问题症状： IE浏览器div最小宽度和高度不生效**

**IE不认得min-这个定义，但实际上它把正常的width和height当作有min的情况来使。这样问题就大了，如果只用宽度和高度，正常的浏览器里这两个值就不会变，如果只用min-width和min-height的话，IE下面根本等于没有设置宽度和高度。**

**比如要设置背景图片，这个最小宽度是比较重要的。要解决这个问题，可以这样：**

**7. 超链接访问过后hover样式就不出现的问题**

**被点击访问过的超链接样式不在具有hover和active了,很多人应该都遇到过这个问题,解决技巧是改变CSS属性的排列顺序: L-V-H-A**

**8. 图片默认有间距**

**问题症状： 几个img标签放在一起的时候，有些浏览器会有默认的间距，通配符清除间距也不起作用。**

**解决方案： 使用float属性为img布局(所有图片左浮)**

**9. css hack解决浏览器兼容性**

**不同浏览器，识别不同的样式，csshack本身就是处理浏览器兼容的。**

**下面是css hack写法：**

**background-color:yellow0; 0 是留给ie8的**

**+background-color:pink; + ie7定了；**

**\_background-color:orange; \_专门留给神奇的ie6；**

**二、js兼容**

**1. 事件绑定**

**IE: dom.attachEvent();**

**标准浏览器： dom.addEventListener(‘click',function(event){},false);**

**标准浏览器采用事件捕获的方式对应IE的事件冒泡机制（即标准由最外元素至最内元素或者IE由最内元素到最外元素）最后标准方亦觉得IE这方面的比较合理，所以便将事件冒泡纳入了标准，这也是addEventListener第三个参数的由来，而且事件冒泡作为了默认值第三值默认false，表示事件冒泡方式。**

**如果浏览器不支持 addEventListener()方法, 你可以使用 attachEvent()方法替代。**

**2. event事件对象问题**

**document.onclick=function(ev){//谷歌火狐的写法，IE9以上支持，往下不支持；**

**var e=ev;**

**console.log(e);**

**}**

**document.onclick=function(){//谷歌和IE支持，火狐不支持；**

**var e=event;**

**console.log(e);**

**}**

**document.onclick=function(ev){//兼容写法；**

**var e=ev||window.event;**

**var mouseX=e.clientX;//鼠标X轴的坐标**

**var mouseY=e.clientY;//鼠标Y轴的坐标**

**}**

**1**

**2. event.srcElement(事件源对象)问题**

**IE： event对象有srcElement属性，但是没有target属性；**

**Firefox: event对象有target属性，但是没有srcElement属性。**

**解决方法：**

**srcObj = event.srcElement?event.srcElement:event.target;**

**1**

**3. 获取元素的非行间样式值：**

**IE: dom.currentStyle[‘width’] 获取元素高度**

**标准浏览器： window.getComputedStyle(obj, null)['width'];**

**跨浏览器兼容解决方法：**

**// 获取元素属性值的兼容写法**

**function getStyle(obj,attr){**

**if(obj.currentStyle){**

**//兼容IE**

**obj.currentStyle[attr];**

**return obj.currentStyle[attr];**

**}else{**

**//非IE，**

**return window.getComputedStyle(obj, null)[attr];**

**}**

**}**

**4. 阻止事件冒泡传播：**

**5. 阻止事件默认行为：**

**//js阻止默认事件 一般阻止a链接href，form表单submit提交**

**6. ajax兼容问题**

**IE： ActiveXObject**

**其他： xmlHttpReuest**

**在IE6以前不是用XMLHttpRequest创建的，所以我们要兼容ie6以前的浏览器要判断他有没有XMLHttpRequest()**

**6.性能优化方面，提升页面的响应速度，考虑过哪些关键的点，**

1、首屏加载优化

2、路由懒加载

{

path: '/',

name: 'home',

component: () => import('./views/home/index.vue'),

meta: { isShowHead: true }

}

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再进行解析。首先安装 webpack 提供的compression-webpack-plugin进行压缩,然后在 vue.config.js：

const CompressionWebpackPlugin = require('compression-webpack-plugin')

const productionGzipExtensions = ['js', 'css']......plugins: [

new CompressionWebpackPlugin(

{

algorithm: 'gzip',

test: new RegExp('\\.(' + productionGzipExtensions.join('|') + ')$'),

threshold: 10240,

minRatio: 0.8

}

)]....

4、启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理是将我们的压力分给其他服务器点。

5、代码层面优化

computed 和 watch 区分使用场景

computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；

watch：类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 ( 访问一个 API )，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

v-if 和 v-show 区分使用场景 v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。这里要说的优化点在于减少页面中 dom 总数，我比较倾向于使用 v-if，因为减少了 dom 数量。

v-for 遍历必须为 item 添加 key，且避免同时使用 v-if v-for 遍历必须为 item 添加 key，循环调用子组件时添加 key，key 可以唯一标识一个循环个体，可以使用例如 item.id 作为 key 避免同时使用 v-if，v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度。

6、Webpack 对图片进行压缩

7、避免内存泄漏

8、减少 ES6 转为 ES5 的冗余代码

**7.webpack打包，压缩，都用过哪些插件**

**Plugin**

**plugin 是一个扩展器，它丰富了 webpack 本身，针对是 loader 结束后，webpack 打包的整个过程，它并不直接操作文件，而是基于事件机制工作，会监听 webpack 打包过程中的某些节点，执行广泛的任务。**

**Plugin 的特点**

**是一个独立的模块**

**模块对外暴露一个 js 函数**

**函数的原型 (prototype) 上定义了一个注入 compiler 对象的 apply方法 apply 函数中需要有通过 compiler 对象挂载的 webpack 事件钩子，钩子的回调中能拿到当前编译的 compilation 对象，如果是异步编译插件的话可以拿到回调 callback**

**完成自定义子编译流程并处理 complition 对象的内部数据**

**如果异步编译插件的话，数据处理完成后执行 callback 回调。**

**下面介绍 18 个常用的 webpack 插件**

**HotModuleReplacementPlugin**

**模块热更新插件。Hot-Module-Replacement 的热更新是依赖于 webpack-dev-server，后者是在打包文件改变时更新打包文件或者 reload 刷新整个页面，HRM 是只更新修改的部分。**

**HotModuleReplacementPlugin是webpack模块自带的，所以引入webpack后，在plugins配置项中直接使用即可。**

**html-webpack-plugin**

**生成 html 文件。将 webpack 中entry配置的相关入口 chunk 和 extract-text-webpack-plugin抽取的 css 样式 插入到该插件提供的template或者templateContent配置项指定的内容基础上生成一个 html 文件，具体插入方式是将样式link插入到head元素中，script插入到head或者body中。**

**8.小程序都用过哪些框架，对小程序的理解**

**在微信开放文档中的**

**WeUI 是一套由微信原生提供的一套基础样式库，由微信官方设计团队为微信内网页和微信小程序量身设计，令用户的使用感知更加统一。小程序开发中最常用到的一款框架，受广大开发人员的欢迎。**

**美团小程序框架mpvue**

**官方介绍：mpvue是一个使用 Vue.js开发小程序的前端框架。框架基于 Vue.js核心，mpvue修改了 Vue.js的 runtime和 compiler实现，使其可以运行在小程序环境中，从而为小程序开发引入了整套Vue.js开发体验。**

**3：组件化开发框架wepy**

**官方介绍：组件化开发，完美解决组件隔离，组件嵌套，组件通信等问题,支持使用第三方 npm 资源，自动处理 npm 资源之间的依赖关系，完美兼容所有无平台依赖的 npm 资源包.**

**4：官方框架MINA**

**官方介绍：框架提供了自己的视图层描述语言 WXML 和 WXSS，以及基于 JavaScript 的逻辑层框架，并在视图层与逻辑层间提供了数据传输和事件系统，可以让开发者可以方便的聚焦于数据与逻辑上。**

**5：Tina.js 一款轻巧的渐进式微信小程序框架**

**官方介绍：是一款轻巧的渐进式微信小程序框架，保留 MINA (微信小程序官方框架) 的大部分 API 设计；无论你有无小程序开发经验，都可以轻松过渡上手。**

**6：前端框架weweb**

**官方介绍：weweb是一个兼容小程序语法的前端框架，你可以用小程序的写法，来写web应用。如果你已经有小程序了，通过它你可以将你的小程序运行在浏览器中。**

**7：微信UI组件库 iView Weapp**

**介绍：iView Weapp 提供了与 iView 一致的 UI 和尽可能相同的接口名称，大幅度降低了学习成本，是一套一套高质量的微信小程序 UI 组件库。**

**8：ZanUI-WeApp – 一个颜值高、好用、易扩展的微信小程序 UI 库**

**官方介绍：ZanUI-WeApp结合了微信的视觉规范，为用户提供更加统一的使用感受。 包含 badge、btn、等共计 17 类组件或元素。**

**9：Taro**

**Taro 是由京东 - 凹凸实验室打造的一套遵循 React 语法规范的多端统一开发框架。我要没记错的话，是最近刚刚开源的。**

**使用 Taro，我们可以只书写一套代码，再通过 Taro 的编译工具，将源代码分别编译出可以在不同端（微信小程序、H5、App 端等）运行的代码。同时 Taro 还提供开箱即用的语法检测和自动补全等功能，有效地提升了开发体验和开发效率。**

# 9、说一下 微信小程序 与 Vue 的区别

1、生命周期：

小程序的钩子函数要简单得多 。 vue的钩子函数在跳转新页面时，钩子函数都会触发，但是小程序的钩子函数，页面不同的跳转方式，触发的钩子并不一样。

在页面加载请求数据时，两者钩子的使用有些类似，vue一般会在created或者mounted中请求数据，而在小程序，会在onLoad或者onShow中请求数据。

2、数据绑定：

vue动态绑定一个变量的值为元素的某个属性的时候，会在变量前面加上冒号：

<img :src="imgSrc"/>

小程序 绑定某个变量的值为元素属性时，会用两个大括号括起来，如果不加括号，为被认为是字符串

<image src="{{imgSrc}}"></image>

3、列表循环

4、显示与隐藏元素

vue中，使用v-if 和v-show控制元素的显示和隐藏

小程序中，使用wx-if和hidden控制元素的显示和隐藏

5、事件处理

vue：使用v-on:event绑定事件，或者使用@event绑定事件

小程序中，全用bindtap(bind+event)，或者catchtap(catch+event)绑定事件

6、数据的双向绑定

在vue中,只需要再表单元素上加上v-model,然后再绑定data中对应的一个值，当表单元素内容发生变化时，data中对应的值也会相应改变 。

当表单内容发生变化时，会触发表单元素上绑定的方法，然后在该方法中，通过this.setData({key:value})来将表单上的值赋值给data中的对应值 。

7、绑定事件传参

在vue中，绑定事件传参挺简单，只需要在触发事件的方法中，把需要传递的数据作为形参传入就可以了

在小程序中，不能直接在绑定事件的方法中传入参数，需要将参数作为属性值，绑定到元素上的data-属性上，然后在方法中，通过e.currentTarget.dataset.\*的方式获取

8、父子组件通信

父组件向子组件传递数据，只需要在子组件通过v-bind传入一个值，在子组件中，通过props接收，即可完成数据的传递

父组件向子组件通信和vue类似，但是小程序没有通过v-bind，而是直接将值赋值给一个变量 在子组件properties中，接收传递的值

# 11、computed和watcher的区别？watch实现原理？watch有几种写法？

使用场景：computed----当一个属性受多个属性影响的时候，使用computed-------购物车商品结算。watch----当一条数据影响多条数据的时候，使用watch-------搜索框。

**计算属性computed :**

**1. 支持缓存，只有依赖数据发生改变，才会重新进行计算**

**2. 不支持异步，当computed内有异步操作时无效，无法监听数据的变化**

**3.computed 属性值会默认走缓存，计算属性是基于它们的响应式依赖进行缓存的，也就是基于data中声明过或者父组件传递的props中的数据通过计算得到的值**

**4. 如果一个属性是由其他属性计算而来的，这个属性依赖其他属性，是一个多对一或者一对一，一般用computed**

**5.如果computed属性属性值是函数，那么默认会走get方法；函数的返回值就是属性的属性值；在computed中的，属性都有一个get和一个set方法，当数据变化时，调用set方法。**

**侦听属性watch：**

**1. 不支持缓存，数据变，直接会触发相应的操作；**

**2.watch支持异步；**

**3.监听的函数接收两个参数，第一个参数是最新的值；第二个参数是输入之前的值；**

**4. 当一个属性发生变化时，需要执行对应的操作；一对多；**

**5. 监听数据必须是data中声明过或者父组件传递过来的props中的数据，当数据变化时，触发其他操作，函数有两个参数，**

**immediate：组件加载立即触发回调函数执行，**

**deep: 深度监听，为了发现对象内部值的变化，复杂类型的数据时使用，例如数组中的对象内容的改变，注意监听数组的变动不需要这么做。**

**注意：deep无法监听到数组的变动和对象的新增，参考vue数组变异,只有以响应式的方式触发才会被监听到。**

**watch工作原理:**

**watch在一开始初始化的时候，会读取一遍监听的数据的值，此时那个数据就收集到watch的watcher了然后你给watch设置的handler，watch 会放入watcher的更新函数中，当数据改变时，通知watch的watcher进行更新，于是你设置的handler就被调用了。**

**nodejs**

**1、什么是nodejs？我们在哪里使用它？**

Nodejs是服务器端的一门技术。它是基于Google V8 JavaScript引擎而开发的。用来开发可扩展的服务端程序。

**2、为什么要使用node js？**

nodejs会让我们的编程工作变得简单，它主要包含如下几点几个好处:

执行快速。

永远不会阻滞。

JavaScript是通用的编程语言。

异步处理机制。

避免并行所带来的问题。

**3、nodejs有哪些特点？**

是单线程的，但是有很高的可扩展性，使用JavaScript作为主流编程语言。使用的是异步处理机制和事件驱动。处理高效。

**4、如何更新nodejs的版本?**

npm install npm -g

**5、为什么nodejs是单线程的？**

Nodejs使用的是单线程没错，但是通过异步处理的方式，可以处理大量的数据吞吐量，从而有更好的性能和可扩展性。

**6、你做的项目中都使用过那些中间件呢？**

中间件可以理解为在请求和响应之间进行响应数据的处理。 分类： 内置中间件

app.use(express.static('托管目录地址')) //静态资源管理中间件 ​ 第三方中间件 body-parser// cookie-parser// cookie-seesion ¬ 自定义中间件 记录访问日志

**7、nodejs的接口(加分项)**

**react**

1. **react生命周期（初始化阶段、运行中状态、销毁状态）**
2. **componentWillMount** 在渲染前调用,在客户端也在服务端。
3. **componentDidMount** : 在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异步操作阻塞UI)。
4. **componentWillReceiveProps** 在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。
5. **shouldComponentUpdate** 返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。
6. 可以在你确认不需要更新组件时使用。
7. **componentWillUpdate**在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。
8. **componentDidUpdate** 在组件完成更新后立即调用。在初始化时不会被调用。
9. **componentWillUnmount**在组件从 DOM 中移除之前立刻被调用。

**初始化阶段：**

getDefaultProps:获取实例的默认属性

getInitialState:获取每个实例的初始化状态

componentWillMount：组件即将被装载、渲染到页面上

render:组件在这里生成虚拟的 DOM 节点

componentDidMount:组件真正在被装载之后

**运行中状态：**

componentWillReceiveProps:组件将要接收到属性的时候调用

shouldComponentUpdate:组件接受到新属性或者新状态的时候（可以返回 false，接收数据后不更新，阻止 render 调用，后面的函数不会被继续执行了）

componentWillUpdate:组件即将更新不能修改属性和状态

render:组件重新描绘

componentDidUpdate:组件已经更新

**销毁阶段：**

componentWillUnmount:组件即将销毁

**2、shouldComponentUpdate 是做什么的？**

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

**3、React 优势**

1、React 速度很快：它并不直接对 DOM 进行操作，引入了一个叫做虚拟 DOM 的概念，安插在 javascript 逻辑和实际的 DOM 之间，性能好。

2、跨浏览器兼容：虚拟 DOM 帮助我们解决了跨浏览器问题，它为我们提供了标准化的 API，甚至在 IE8 中都是没问题的。

3、一切都是 component：代码更加模块化，复用代码更容易，可维护性高。

4、单向数据流：Flux 是一个用于在 JavaScript 应用中创建单向数据层的架构，它随着 React 视图库的开发而被 Facebook 概念化。

5、同构、纯粹的 javascript：因为搜索引擎的爬虫程序依赖的是服务端响应而不是 JavaScript 的执行，预渲染你的应用有助于搜索引擎优化。

6、兼容性好：比如使用 RequireJS 来加载和打包，而 Browserify 和 Webpack 适用于构建大型应用。它们使得那些艰难的任务不再让人望而生畏。

**4、React 中 refs 的作用是什么？**

* Refs 是 React 提供给我们的安全访问 DOM元素或者某个组件实例的句柄
* 可以为元素添加ref属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回

**5、高阶组件(higher order component)**

高阶组件是一个以组件为参数并返回一个新组件的函数。HOC 运行你重用代码、逻辑和引导抽象。最常见的可能是 Redux 的 connect 函数。除了简单分享工具库和简单的组合，HOC 最好的方式是共享 React 组件之间的行为。如果你发现你在不同的地方写了大量代码来做同一件事时，就应该考虑将代码重构为可重用的 HOC。

**6、setState 和 replaceState 的区别**

setState 是修改其中的部分状态，相当于 Object.assign，只是覆盖， 不会减少原来的状态

replaceState 是完全替换原来的状态，相当于赋值，将原来的 state 替换为另一个对象，如果新状态属性减少，那么 state 中就没有这个状态了

**7、调用 super(props) 的目的是什么**

在 super() 被调用之前，子类是不能使用 this 的，在 ES2015 中，子类必须在 constructor 中调用 super()。

传递 props 给 super() 的原因则是便于(在子类中)能在 constructor 访问 this.props。

**8、态(state)和属性(props)之间有何区别**

State 是一种数据结构，用于组件挂载时所需数据的默认值。

State 可能会随着时间的推移而发生突变，但多数时候是作为用户事件行为的结果。

Props(properties 的简写)则是组件的配置。props 由父组件传递给子组件，并且就子组件而言，props 是不可变的(immutable)。

组件不能改变自身的 props，但是可以把其子组件的 props 放在一起(统一管理)。Props 也不仅仅是数据--回调函数也可以通过 props 传递。

**9、应该在 React 组件的何处发起 Ajax 请求**

在 React 组件中，应该在 componentDidMount 中发起网络请求。这个方法会在组件第一次“挂载”(被添加到 DOM)时执行，在组件的生命周期中仅会执行一次。

更重要的是，你不能保证在组件挂载之前 Ajax 请求已经完成，如果是这样，也就意味着你将尝试在一个未挂载的组件上调用 setState，这将不起作用。 在 componentDidMount 中发起网络请求将保证这有一个组件可以更新了。

**10、Axios写在哪个生命周期**

componentDidMount

**11、React组件之间的传值**

1.向下传值--父组件向子组件传值 父组件通过props传递一个不是方法的数据，给子组件；

2.向上传值--子组件向父组件传值 父组件通过props向子组件传入一个方法，子组件在通过调用该方法，并将数据以参数的形式传给父组件，父组件可以在该方法中对传入的数据进行处理；

按传值方法分：props，ref，context，updater，redux

**12、render函数什么时候会执行？**

当this.state或者this.props发生改变的时候，render函数会执行

**React.js 面试真题**

**★★★ React 事件绑定原理**

/\*

一、react并没有使用原生的浏览器事件，而是在基于Virtual DOM的基础上实现了合成事件。

二、react合成事件主要分为以下三个过程：

1、事件注册

document上注册、存储事件回调。

2、事件合成

事件触发后，会执行一下过程：

（1）进入统一的事件分发函数dispatchEvent；

（2）找到触发事件的 ReactDOMComponent；

（3）开始事件的合成；

—— 根据当前事件类型生成指定的合成对象

—— 封装原生事件和冒泡机制

—— 查找当前元素以及他所有父级

—— 在listenerBank根据key值查找事件回调并合成到 event(合成事件结束)

3、批处理

批量处理合成事件内的回调函数

\*/

**★★★ React中的 setState 缺点是什么呢**

setState执行的时候可以简单的认为，隶属于原生js执行的空间，那么就是属于同步，被react处理过的空间属于异步，这其实也是一种性能的优化，如果多次使用setState修改值，那么在异步中会先进行合并，再进行渲染，降低了操作dom的次数

**★★★ React组件通信如何实现**

/\*

react本身:

(1)props——父组件向子组件通过props传参

(2)实例方法——在父组件中可以用 refs 引用子组件，之后就可以调用子组件的实例方法了

(3)回调函数——用于子组件向父组件通信，子组件调用props传递过来的方法

(4)状态提升——两个子组件可以通过父组件定义的参数进行传参

(5)Context上下文——一般用作全局主题

(6)Render Props——渲染的细节由父组件控制

状态管理:

(1) mobx/redux/dva——通过在view中触发action，改变state，进而改变其他组件的view

\*/

**★★★ 类组件和函数组件的区别**

/\*

（1）语法上：函数组件是一个函数，返回一个jsx元素，而类组件是用es6语法糖class定义，继承component这个类

（2）类组件中可以通过state进行状态管理，而在函数组件中不能使用setState()，在react16.8以后，函数组件可以通过hooks中的useState来模拟类组件中的状态管理；

（3）类组件中有一系列的生命周期钩子函数，在函数组件中也需要借助hooks来使用生命周期函数；

（4）类组件能够捕获\*\*最新\*\*的值（永远保持一致），这是因为当实例的props属性发生修改时，class组件能够直接通过this捕获到组件最新的props；而函数式组件是捕获\*\*渲染\*\*所使用的值，已经因为javascript\*\*闭包\*\*的特性，之前的props参数保存在内存之中，无法从外部进行修改。

\*/

**★★★ 请你说说React的路由是什么？**

/\*

路由分为前端路由和后端路由，后端路由是服务器根据用户发起的请求而返回不同内容，前端路由是客户端根据不同的URL去切换组件；在web应用前端开发中，路由系统是最核心的部分，当页面的URL发生改变时，页面的显示结果可以根据URL的变化而变化，但是页面不会刷新。

react生态中路由通常是使用react-router来进行配置，其主要构成为：

（1）Router——对应路由的两种模式，包括<BrowsersRouter>与<HashRouter>；

（2）route matching组件——控制路径对应的显示组件，可以进行同步加载和异步加载，<Route>；

（3）navigation组件——用做路由切换和跳转，<Link>；

BrowserRouter与HashRouter的区别：

（1）底层原理不一样：BrowserRouter使用的是H5的history API，不兼容IE9及以下版本；HashRouter使用的是URL的哈希值；

（2）path表现形式不一样：BrowserRouter的路径中没有#,例如：localhost:3000/demo/test；HashRouter的路径包含#,例如：localhost:3000/#/demo/test；

（3）刷新后对路由state参数的影响：BrowserRouter没有任何影响，因为state保存在history对象中；HashRouter刷新后会导致路由state参数的丢失；

\*/

**★★★★★ React有哪些性能优化的手段？**

/\*

1. 使用纯组件；

5、使用 React Fragments 避免额外标记；

2、使用 React.memo 进行组件记忆，对于相同的输入，不重复执行；

3、如果是类组件，使用 shouldComponentUpdate生命周期事件，可以利用此事件来决定何时需要重新渲染组件；

8、如果是类组件，事件函数在Constructor中绑定bind改变this指向；

14、类组件中使用immutable对象；

4、路由懒加载；

10、优化 React 中的条件渲染；

12、列表渲染的时候加key；

6、不要使用内联函数定义

9、避免使用内联样式属性；

11、不要在 render 方法中导出数据；

7、避免在Willxxx系列的生命周期中进行异步请求，操作dom等；

13、在函数组件中使用useCallback和useMemo来进行组件优化，依赖没有变化的话，不重复执行；

\*/

**★★★★ React hooks 用过吗，为什么要用？**

/\*

Hooks 是React在16.8版本中出的一个新功能，本质是一种函数，可以实现组件逻辑复用，Hook 使你在无需修改组件结构的情况下复用状态逻辑，Hook 将组件中相互关联的部分拆分成更小的函数，组件树层级变浅。组件逻辑耦合度低，组件颗粒度小，从而从整体上实现性能优化让我们在函数式组件中使用类组件中的状态、生命周期等功能，hooks的名字都是以use开头。

react：

1、useState——创建状态

接收一个参数作为初始值；返回一个数组，第一个值为状态，第二个值为改变状态的函数

2、useEffect——副作用（数据获取、dom操作影响页面——在渲染结束之后执行

(1)第一个参数为函数，第二个参数为依赖列表，只有依赖更新时才会执行函数；返回一个函数，当页面刷新的时候先执行返回函数再执行参数函数

(2)如果不接受第二个参数，那么在第一次渲染完成之后和每次更新渲染页面的时候，都会调用useEffect的回调函数

3、useRef

返回一个可变的ref对象，此索引在整个生命周期中保持不变。可以用来获取元素或组件的实例，用来做输入框的聚焦或者动画的触发。

4、useMemo——优化函数组件中的功能函数——在渲染期间执行

（1）接收一个函数作为参数，同样接收第二个参数作为依赖列表，返回值可以是任何，函数、对象等都可以

（2）这种优化有助于避免在每次渲染时都进行高开销的计算，仅会在某个依赖项改变时才重新计算

5、useContext——获取上下文注入的值

(1)接受一个context 对象，并返回该对象<MyContext.Provider> 元素的 value值；

const value = useContext(MyContext)；

6、useLayoutEffect——有DOM操作的副作用——在DOM更新之后执行

和useEffet类似，但是执行时机不同，useLayoutEffect在DOM更新之后执行，useEffect在render渲染结束后执行，也就是说useLayoutEffect比useEffect先执行，这是因为DOM更新之后，渲染才结束或者渲染还会结束

7、useCallback——与useMemo类似

useMemo与useCallback相同，接收一个函数作为参数，也同样接收第二个参数作为依赖列表；useCallback是对传过来的回调函数优化，返回的是一个函数

react-router:

被route包裹的组件，可以直接使用props进行路由相关操作，但是没有被route包裹的组件只能用withRouter高阶组件修饰或者使用hooks进行操作

1、useHistory——跳转路由

2、useLocation——得到url对象

3、useParams——得到url上的参数

react-redux:

1、useSelector——共享状态——从redux的store中提取数据

2、useDispatch——共享状态——返回redux的store中对dispatch的引用

\*/

**★★★★ 虚拟DOM的优劣如何？实现原理？**

/\*

虚拟dom是用js模拟一颗dom树,放在浏览器内存中，相当于在js和真实dom中加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。

优点：

（1）虚拟DOM具有批处理和高效的Diff算法,最终表现在DOM上的修改只是变更的部分，可以保证非常高效的渲染,优化性能；

（2）虚拟DOM不会立马进行排版与重绘操作，对虚拟DOM进行频繁修改，最后一次性比较并修改真实DOM中需要改的部分；

（3）虚拟 DOM 有效降低大面积真实 DOM 的重绘与排版，因为最终与真实 DOM 比较差异，可以只渲染局部；

缺点：

（1）首次渲染大量DOM时，由于多了一层虚拟DOM的计算，会比innerHTML插入慢；

React组件的渲染过程：

（1）使用JSX编写React组件后所有的JSX代码会通过Babel转化为 React.createElement执行;

（2）createElement函数对 key和 ref等特殊的 props进行处理，并获取 defaultProps对默认 props进行赋值，并且对传入的子节点进行处理，最终构造成一个 ReactElement对象（所谓的虚拟 DOM）。

（3）ReactDOM.render将生成好的虚拟 DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实 DOM。

虚拟DOM的组成——ReactElementelement对象结构：

（1）type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）

（2）key：组件的唯一标识，用于Diff算法，下面会详细介绍

（3）ref：用于访问原生dom节点

（4）props：传入组件的props，chidren是props中的一个属性，它存储了当前组件的孩子节点，可以是数组（多个孩子节点）或对象（只有一个孩子节点）

（5）owner：当前正在构建的Component所属的Component

（6）self：（非生产环境）指定当前位于哪个组件实例

（7）\_source：（非生产环境）指定调试代码来自的文件(fileName)和代码行数(lineNumber)

\*/

**★★★★ React 和 Vue 的 diff 时间复杂度从 O(n^3) 优化到 O(n) ，那么 O(n^3) 和 O(n) 是如何计算出来的？**

/\*

react的diff算法只需要O(n)，这是因为react对树节点的比较做了一些前提假设，限定死了一些东西，不做过于复杂的计算操作，所以降低了复杂度。react和vue做了以下的假设，这样的话diff运算时只进行同层比较，每一个节点只遍历了一次。

（1）Web UI中DOM节点跨层级的移动操作特别少，可以忽略不计；

（2）拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构；

（3）对于同一层级的一组子节点，它们可以通过唯一 id 进行区分。

而传统的diff运算时间复杂度为O(n^3)，这是因为传统的树节点要做非常完整的检查，首先需要节点之间需要两两比较，找到所有差异，这个对比过程时间复杂度为O(n^2)，找到差异后还要计算出最小的转换方式，最终复杂度为O(n^3)

\*/

**★★★ 聊聊 Redux 和 Vuex 的设计思想**

Redux设计和使用的三大原则：

（1）单一的数据源：整个应用的 state被储存在唯一一个 store中；

（2）状态是只读的：Store.state不能直接修改（只读），必须调用dispatch(action) => store.reducer => return newState；action是一个对象，有type（操作类型）和payload（新值）属性；

（3）状态修改均由纯函数完成：在Redux中，通过纯函数reducer来确定状态的改变，因为reducer是纯函数，所以相同的输入，一定会得到相同的输出，同时也不支持异步；返回值是一个全新的state；

vuex由State + Muatations(commit) + Actions(dispatch) 组成：

（1）全局只有一个Store实例（单一数据源）；

（2）Mutations必须是同步事务，不同步修改的话，会很难调试，不知道改变什么时候发生，也很难确定先后顺序，A、B两个mutation，调用顺序可能是A -> B，但是最终改变 State的结果可能是B -> A；

（3）Actions负责处理异步事务，然后在异步回调中触发一个或多个mutations，也可以在业务代码中处理异步事务，然后在回调中同样操作；

（4）模块化通过module方式来处理，这个跟Redux-combineReducer类似，在应用中可以通过namespaceHelper来简化使用；

\*/

**★★★ React中不同组件之间如何做到数据交互？**

**★★★ React中refs的作用是什么？**

// ref是React提供的用来操纵React组件实例或者DOM元素的接口。主要用来做文本框的聚焦、触发强制动画等；

// 类组件

class Foo extends React.Component {

constructor(props) {

super(props)

this.myRef = React.createRef()

}

render() {

return

<div>

<input ref={ this.myRef } />

<button onClick = {()=>this.handle()}>聚焦</button>

</div>

}

handle() {

// 通过current属性访问到当前元素

this.myRef.current.focus()

}

}

// 函数组件

function Foo() {

const inputEl = useRef(null)

const handle = () => {

inputEl.current.focus()

}

return

<div>

<input type="text" ref={ inputEl }/>

<button onClick = {handle}>聚焦</button>

</div>

}

**★★★★ 请列举react生命周期函数。**

/\*

第一阶段：装载阶段3

constructor()

render()

componentDidMount()

第二阶段：更新阶段2

[shouldComponentUpdate()]

render()

componentDidUpdate()

第三阶段：卸载阶段1

componentWillUnmount()

constructor生命周期：

(1)当react组件实例化时，是第一个运行的生命周期；

(2)在这个生命周期中，不能使用this.setState()；

(3)在这个生命周期中，不能使用副作用(调接口、dom操作、定时器、长连接等)；

(4)不能把props和state交叉赋值；

componentDidMount生命周期：

(1)相当于是vue中的mounted；

(2)它表示DOM结构在浏览器中渲染已完成；

(3)在这里可以使用任何的副作用；

shouldComponentUpdate(nextProps,nextState)生命周期：

(1)相当于一个开关，如果返回true则更新机制正常执行，如果为false则更新机制停止；

(2)在vue中是没有的；

(3)存在的意义：可以用于性能优化，但是不常用，最新的解决方案是使用PureComponent；

(4)理论上，这个生命周期的作用，用于精细地控制声明式变量的更新问题，如果变化的声明式变量参与了视图渲染则返回true，如果被变化的声明式变量没有直接或间接参与视图渲染，则返回false；

componentDidUpdate生命周期：

(1)相当于vue中的updated()；

(2)它表示DOM结构渲染更新已完成，只发生在更新阶段；

(3)在这里，可以执行大多数的副作用，但是不建议；

(4)在这里，可以使用this.setState()，但是要有终止条件判断。

componentWillUnmount生命周期：

(1)一般在这里清除定时器、长连接等其他占用内存的构造器；

render生命周期：

(1)render是类组件中唯一必须有的生命周期，同时必须有return（return 返回的jsx默认只能是单一根节点，但是在fragment的语法支持下，可以返回多个兄弟节点）；

(2)Fragment碎片写法: <React.Fragment></React.Fragment> 简写成<></>；

(3)return之前，可以做任意的业务逻辑，但是不能使用this.setState()，会造成死循环；

(4)render()在装载阶段和更新阶段都会运行；

(5)当render方法返回null的时候，不会影响生命周期函数的正常执行。

\*/

**★★★ 组件绑定和js原生绑定事件哪个先执行？**

// 先执行js原生绑定事件，再执行合成事件，因为合成事件是发生在冒泡阶段

**★★★ React中的keys的作用是什么？**

##### Answer

key 是用来帮助 react 识别哪些内容被更改、添加或者删除。key 需要写在用数组渲染出来的元素内部，并且需要赋予其一个稳定的值。稳定在这里很重要，因为如果 key 值发生了变更，react 则会触发 UI 的重渲染。这是一个非常有用的特性。

* key 的唯一性

在相邻的元素间，key 值必须是唯一的，如果出现了相同的 key，同样会抛出一个 Warning，告诉相邻组件间有重复的 key 值。并且只会渲染第一个重复 key 值中的元素，因为 react 会认为后续拥有相同 key 的都是同一个组件。

* key 值不可读

虽然我们在组件上定义了 key，但是在其子组件中，我们并没有办法拿到 key 的值，因为 key 仅仅是给 react 内部使用的。如果我们需要使用到 key 值，可以通过其他方式传入，比如将 key 值赋给 id 等

**★★ fetch的延时操作**

// fetch语法：fetch(resource, config).then( function(response) { ... } )；resource为要获取的资源，config是配置对象，包含method请求方法，headers请求头信息等

// 定义一个延时函数，返回一个promise

const delayPromise = (timeout=5000) => {

return new Promise((resolve, reject) => {

setTimeout(()=>{

reject(new Error("网络错误"))

}, timeout)

})

}

// 定义一个fetch网络请求，返回一个promise

const fetchPromise = (resource, config) => {

return new Promise((resolve, reject)=>{

fetch(resource, config).then(res=>{

resolve(res)

})

})

}

// promise的race静态方法接受多个promise对象组成的数组，该数组中哪个promise先执行完成，race方法就返回这个promise的执行结果

const fetchRequest = (resource, config, timeout) => {

Promise.race([

delayPromise(timeout),

fetchPromise(resource,config)

])

}

**★★ A 组件嵌套 B 组件，生命周期执行顺序**

/\*

父组件创建阶段的生命周期钩子函数 constructor

父组件创建阶段的生命周期钩子函数 render

子组件创建阶段的生命周期钩子函数 constructor

子组件创建阶段的生命周期钩子函数 render

子组件创建阶段的生命周期钩子函数 componentDidMount

父组件创建阶段的生命周期钩子函数 componentDidMount

\*/

**★★★ diff 和 Key 之间的联系**

/\*

diff算法即差异查找算法，对于DOM结构即为tree的差异查找算法，只有在React更新阶段才会有Diff算法的运用；react的diff运算为了降低时间复杂度，是按层比较新旧两个虚拟dom树的。diff运算的主要流程见下：

1、tree diff : 新旧两棵dom树，逐层对比的过程就是 tree diff, 当整棵DOM树逐层对比完毕，则所有需要被按需更新的元素，必然能够被找到。

2、component diff ： 在进行tree diff的时候，每一层中，都有自己的组件，组件级别的对比，叫做 component diff。如果对比前后，组件的类型相同，则暂时认为此组件不需要更新；如果对比前后，组件的类型不同，则需要移除旧组件，创建新组件，并渲染到页面上。

React只会匹配类型相同的组件，也就是说如果<A>被<B>替换，那么React将直接删除A组件然后创建一个B组件；如果某组件A转移到同层B组件上，那么这个A组件会先被销毁，然后在B组件下重新生成，以A为根节点的树整个都被重新创建，这会比较耗费性能，但实际上我们很少跨层移动dom节点，一般都是同层横向移动；

3、element diff ：在进行组件对比的时候，如果两个组件类型相同，则需要进行元素级别的对比，这叫做element diff。

对于列表渲染，react会在创建时要求为每一项输入一个独一无二的key，这样就能进行高效的diff运算了。比如我们要在b和c节点中间插入一个节点f，jquery会将f这个节点后面的每一个节点都进行更新，比如c更新成f，d更新成c，e更新成d，这样操作的话就会特别多，而加了key的react咋不会频繁操作dom，而是优先采用移动的方式，找到正确的位置去插入新节点；所以我们不能省略key值，因为在对比两个新旧的子元素是，是通过key值来精确地判断两个节点是否为同一个，如果没有key的话则是见到谁就更新谁，非常耗费性能。

当我们通过this.setState()改变数据的时候，React会将其标记为脏节点，在事件循环的最后才会重新渲染所有的脏节点以及脏节点的子树；另外我们可以使用shouldComponentUpdate这个生命周期来选择性的渲染子树，可以基于组件之前的状态或者下一个状态来决定它是否需要重新渲染，这样的话可以组织重新渲染大的子树。

\*/

**★★★ 虚拟 dom 和原生 dom**

/\*

（1）原生dom是浏览器通过dom树渲染的复杂对象，属性非常多；

（2）虚拟dom是存在于内存中的js对象，属性远少于原生的dom对象，它用来描述真实的dom，并不会直接在浏览器中显示；

（3）原生dom操作、频繁排版与重绘的效率是相当低的，虚拟dom则是利用了计算机内存高效的运算性能减少了性能的损耗；

（4）虚拟DOM进行频繁修改，然后一次性比较并修改真实DOM中需要改的部分，最后并在真实DOM中对修改部分进行排版与重绘，减少过多DOM节点排版与重绘损耗

\*/

**★★★★ 新出来两个钩子函数？和砍掉的will系列有啥区别？**

// react16 中废弃了三个钩子

componentWillMount // 组件将要挂载的钩子

componentWillReceiveProps // 组件将要接收一个新的参数时的钩子

componentWillUpdate // 组件将要更新的钩子

// 新增了方法

getDerivedStateFromProps // 静态方法

getSnapshotBeforeUpdate

/\*

在16.8版本以后，react将diff运算改进为Fiber，这样的话当我们调用setState方法进行更新的时候，在reconciler 层中js运算会按照节点为单位拆分成一个个小的工作单元，在render前可能会中断或恢复，就有可能导致在render前这些生命周期在进行一次更新时存在多次执行的情况，此时如果我们在里面使用ref操作dom的话，就会造成页面频繁重绘，影响性能。

所以废弃了这几个will系列的勾子，增加了 getDerivedStateFromProps这个静态方法，这样的话我们就不能在其中使用this.refs以及this上的方法了；getSnapshotBeforeUpdate 这个方法已经到了commit阶段，只会执行一次，给想读取 dom 的用户一些空间。

\*/

**★★★ react中如何打包上传图片文件**

**★★★ 对单向数据流和双向数据绑定的理解，好处？**

/\*

react的单向数据流是指只允许父组件向子组件传递数据，子组件绝对不能修改父组件传的数据，如果想要修改数据，则要在子组件中执行父组件传递过来的回调函数，提醒父组件对数据进行修改。数据单向流让所有的状态改变可以追溯，有利于应用的可维护性；

angular中实现了双向数据绑定，代码编写方便，但是不利于维护

\*/

**★★ React 组件中 props 和 state 有什么区别？**

/\*

1、props是从外部传入组件的参数，一般用于父组件向子组件通信，在组件之间通信使用；state一般用于组件内部的状态维护，更新组建内部的数据，状态，更新子组件的props等

2、props不可以在组件内部修改，只能通过父组件进行修改；state在组件内部通过setState修改；

\*/

**★★ react中组件分为那俩种？**

// 类组件和函数组件

**★★ react中函数组件和普通组件的区别？**

见上

**★★★★ react中 setState 之后做了什么？**

/\*

如果是在隶属于原生js执行的空间，比如说setTimeout里面，setState是同步的，那么每次执行setState将立即更新this.state，然后触发render方法，渲染数据；

如果是在被react处理过的空间执行，比如说合成事件里，此时setState是异步执行的，并不会立即更新this.state的值，当执行setState的时候，会将需要更新的state放入状态队列，在这个空间最后再合并修改this.state，触发render；

\*/

**★★★★ redux本来是同步的，为什么它能执行异步代码？中间件的实现原理是什么？**

/\*

当我们需要修改store中值的时候，我们是通过 dispatch(action)将要修改的值传到reducer中的，这个过程是同步的，如果我们要进行异步操作的时候，就需要用到中间件；中间件其实是提供了一个分类处理action的机会，在 middleware 中，我们可以检阅每一个流过的action，并挑选出特定类型的 action进行相应操作，以此来改变 action；

Redux中间件原理:是对dispatch()函数能力的一个扩展

applyMiddleware 是个三级柯里化的函数。它将陆续的获得三个参数：第一个是 middlewares 数组，第二个是 Redux 原生的 createStore，最后一个是 reducer；然后applyMiddleware会将不同的中间件一层一层包裹到原生的 dispatch 之上；

redux-thunk 中间件的作用就是让我们可以异步执行redux，首先检查参数 action 的类型，如果是函数的话，就执行这个 action这个函数，并把 dispatch, getState, extraArgument 作为参数传递进去，否则就调用next让下一个中间件继续处理action。

\*/

// redux-thunk部分源码

function createThunkMiddleware(extraArgument) {

return ({ dispatch, getState }) => next => action => {

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

return action(dispatch, getState, extraArgument)

}

return next(action)

}

}

const thunk = createThunkMiddleware()

thunk.withExtraArgument = createThunkMiddleware

export default thunk

**★★★★ 列举重新渲染 render 的情况**

this.setState()

this.forceUpdate()

// 接受到新的props

// 通过状态管理，mobx、redux等

// 改变上下文

**★★★ React 按需加载**

// 1、使用React.lazy， 但是React.lazy技术还不支持服务端渲染

const OtherComponent = React.lazy(() => import('./OtherComponent'))

// 2、使用Loadable Components这个库

import loadable from '@loadable/component'

const OtherComponent = loadable(() => import('./OtherComponent'))

**★★★ React 实现目录树（组件自身调用自身）**

**★★★ React组件生命周期按装载，更新，销毁三个阶段分别都有哪些？**

见上

**★★★★★ 调用this.setState之后，React都做了哪些操作？怎么拿到改变后的值？**

/\*

如果是在隶属于原生js执行的空间，比如说setTimeout里面，setState是同步的，那么每次执行setState将立即更新this.state，然后触发render方法；因为是同步执行，可以直接获取改变后的值；

如果是在被react处理过的空间执行，比如说合成事件里，此时setState是异步执行的，并不会立即更新this.state的值，当执行setState的时候，会将需要更新的state放入状态队列，在这个空间最后再合并修改this.state，触发render；setState接受第二个参数，是一个回调函数，可以在这里获取改变后的state值；

触发render执行后，会生成一个新的虚拟dom结构，然后触发diff运算，找到变化的地方，重新渲染；

\*/

**★★★ 如果我进行三次setState会发生什么**

// 看情况，如果是在原生js空间，则会同步执行，修改三次state的值，调用三次render函数；如果是在react函数空间下，则会进行合并，只修改一次state的值，调用一次render。

**★★★ 循环执行setState组件会一直重新渲染吗？为什么？**

见上

**★★★ 渲染一个react组件的过程**

1、babel编译、

2、生成element、

3、生成真实节点(初次渲染)、

4、生命周期

1、babel编译

当我们对代码进行编译的时候，babel会将我们在组件中编写的jsx代码转化为React.createElement的表达式，createElement方法有三个参数，分别为type(元素类型）、attributes(元素所有属性)、children(元素所有子节点)；

2、生成element

当render方法被触发以后，createElement方法会执行，返回一个element对象，这个对象描述了真实节点的信息，其实就是虚拟dom节点；

3、生成真实节点(初次渲染)

这时候我们会判断element的类型，如果是null、false则实例一个ReactDOMEmptyComponent对象; 是string、number类型的话则实例一个ReactDOMTextComponent对象； 如果element是对象的话，会进一步判断type元素类型，是原生dom元素，则实例化ReactDOMComponent； 如果是自定义组件，则实例化ReactCompositeComponentWrapper；

在这些类生成实例对象的时候，在其内部会调用 mountComponent方法，这个方法里面有一系列浏览器原生dom方法，可以将element渲染成真实的dom并插入到文档中；

4、生命周期

componentDidMount：会在组件挂载后(插入DOM树中) 立即调用。一般可以在这里请求数据；

componentDidUpdate：会在数据更新后立即调用，首次渲染不会执行此方法；可以在其中直接调用 setState，但必须用if语句进行判断，防止死循环；

conponentWillUnmount：会在组件卸载及销毁之前调用，在此方法中执行必要的清理操作，如清除timer；

static getDerivedStateFromProps(prps,state)：这个生命周期函数代替了componentWillMount和componentWillUpdate生命周期；props和state发生改变则调用，在初始化挂载及后续更新时都会被调用，返回一个对象来更新state，如果返回null则不更新任何内容；

shouldComponentUpdate(nextProps,nextState)：这个生命周期函数的返回值用来判断React组件是否因为当前 state 或 props 更改而重新渲染，默认返回值是true；这个方法在初始化渲染或使用forceUpdate()时不会调用；当将旧的state的值原封不动赋值给新的state（即不改变state的值，但是调用了setState）和 无数据交换的父组件的重新渲染都会导致组件重新渲染，这时候都可以通过shouldComponentUpdate来优化。

\*/

**★★★ React类组件，函数组件，在类组件修改组件对象会使用。**

**★★★★ 类组件怎么做性能优化？函数组件怎么做性能优化？**

/\*

类组件：

（1）使用shouldComponentUpdate：这个生命周期可以让我们决定当前状态或属性的改变是否重新渲染组件，默认返回ture，返回false时不会执行render，在初始化渲染或使用forceUpdate()时不会调用；如果在shouldComponentUpdate比较的值是引用类型的话，可能达不到我们想要的效果，因为引用类型指向同一个地址；

当将旧的state的值原封不动赋值给新的state（即不改变state的值，但是调用了setState）和 无数据交换的父组件的重新渲染都会导致组件重新渲染，这时候都可以通过shouldComponentUpdate来优化；

（2）React.PureComponent：基本上和Component用法一致，不同之处在于 PureComponent不需要开发者自己设置shouldComponentUpdate，因为PureComponent自带通过props和state的浅对比来实现 shouldComponentUpate；但是如果props和state对象包含复杂的数据结构，它可能会判断错误(表现为对象深层的数据已改变，视图却没有更新）；

（4）使用Immutable：immutable是一种持久化数据，一旦被创建就不会被修改，修改immutable对象的时候返回新的immutable；也就是说在使用旧数据创建新数据的时候，会保证旧数据同时可用且不变；为了避免深度复制所有节点的带来的性能损耗，immutable使用了结构共享，即如果对象树中的一个节点发生变化，只修改这个节点和受他影响的父节点，其他节点仍然共享；

（5）bind函数：在react中改变this的指向有三种方法，a)constructor中用bind绑定; b)使用时通过bind绑定; 3)使用箭头函数；选择第一种只在组件初始化的时候执行一次，第二种组件在每次render都要重新绑定，第三种在每次render时候都会生成新的箭头函数，所以选择第一种；

函数组件：

（1）useCallback：接收一个函数作为参数，接收第二个参数作为依赖列表，返回值为函数，有助于避免在每次渲染时都进行高开销的计算，仅会在某个依赖项改变时才重新计算；可以使用useCallback把要传递给子组件的函数包裹起来，这样父组件刷新的时候，传递给子组件的函数指向不会发生改变，可以减少子组件的渲染次数；

const handleUseCallback=useCallback(handleClick,[])

<Child handleClick={handleUseCallback} />

（2）useMemo：useMemo的使用和useCallback差不多，只是useCallback返回的是一个函数，useMemo返回值可以是函数、对象等都可以；

两者都可使用：

（1）React.memo：React.memo 功能同React.PureComponent，但React.memo是高阶组件，既可以用在类组件中也可以用在函数组件中；memo还可以接收第二个参数，是一个可定制化的比较函数，其返回值与 shouldComponentUpdate的相反；

（2）使用key：在列表渲染时使用key，这样当组件发生增删改、排序等操作时，diff运算后可以根据key值直接调整DOM顺序，避免不必要的渲染而避免性能的浪费；

（3）不要滥用props：尽量只传需要的数据，避免多余的更新，尽量避免使用{…props}；

\*/

**★★★ useEffect 和 useLayoutEffect 的区别**

/\*

2、useEffect和useLayout都是副作用hooks，两则非常相似，同样都接收两个参数：

(1)第一个参数为函数，第二个参数为依赖列表，

useEffect和 useLayout的主要区别就是他们的执行时机不同，在浏览器中js线程与渲染线程是互斥的，当js线程执行时，渲染线程呈挂起状态，只有当js线程空闲时渲染线程才会执行，将生成的 dom绘制。useLayoutEffect在js线程执行完毕即dom更新之后立即执行，而useEffect是在渲染结束后才执行，也就是说 useLayoutEffect比 useEffect先执行。

\*/

**★★★ hooks 的使用有什么注意事项**

/\*

（1）只能在React函数式组件或自定义Hook中使用Hook。

（2）不要在循环，条件或嵌套函数中调用Hook，必须始终在React函数的顶层使用Hook。这是因为React需要利用调用顺序来正确更新相应的状态，以及调用相应的钩子函数。一旦在循环或条件分支语句中调用Hook，就容易导致调用顺序的不一致性，从而产生难以预料到的后果。

\*/

**★★★ 纯函数有什么特点，副作用函数特点**

/\*

纯函数与外界交换数据只有一个唯一渠道——参数和返回值；函数从函数外部接受的所有输入信息都通过参数传递到该函数内部；函数输出到函数外部的所有信息都通过返回值传递到该函数外部。

纯函数的优点：无状态，线程安全；纯函数相互调用组装起来的函数，还是纯函数；应用程序或者运行环境可以对纯函数的运算结果进行缓存，运算加快速度。

函数副作用是指当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响。比如调接口、修改全局变量、抛出一个异常或以一个错误终止、打印到终端或读取用户输入、读取或写入一个文件等，所以说副作用是编程中最关键的部分，因为我们需要跟用户、跟数据进行交互。

\*/

**★★ React 中 refs 干嘛用的？如何创建 refs？**

见上

**★★★ 在构造函数调用 `super` 并将 `props` 作为参数传入的作用是啥？**

/\*

ES6 中在调用 super()方法之前，子类构造函数无法使用this引用，在react的类组件中也是如此；将 props 参数传递给 super() 调用的主要原因是在子构造函数中能够通过this.props来获取传入的 props。

\*/

**★★★ 如何 React.createElement ？**

见上

**★★★ 讲讲什么是 JSX ？**

/\*

JSX全称为JavaScript XML，是react中的一种语法糖，可以让我们在js代码中脱离字符串直接编写html代码；本身不能被浏览器读取，必须使用@babel/preset-react和webpack等工具将其转换为传统的JS。

主要有以下特点：

（1）类XML语法容易接受，结构清晰；

（2）增强JS语义；

（3）抽象程度高，屏蔽DOM操作，跨平台；

（4）代码模块化；

\*/

**★★★ 为什么不直接更新 `state` 呢？**

// 如果试图直接更新 state ，则不会重新渲染组件；需要使用setState()方法来更新 state这样组件才会重新渲染；

**★★★ React 组件生命周期有哪些不同阶段？React 的生命周期方法有哪些？**

见上

**★★★ 这三个点(...)在 React 干嘛用的？**

// ...是es6语法新出的规范，叫做展开运算符；在react中可以将对象或数组进行展开，让我们操作改变数据结构非常方便。

**★★★ React 中的 `useState()` 是什么？**

// useState是一个内置的React Hook，可以让我们在函数组件中像类组件一样使用state并且改变state的值。

**★★★ React 中的StrictMode(严格模式)是什么？**

/\*

React的StrictMode是一种辅助组件，用<StrictMode />包装组件，可以帮助我们编写更好的react组件，不会渲染出任何可见的ui；仅在开发模式下运行，它们不会影响生产构建，可以做以下检查：

（1）验证内部组件是否遵循某些推荐做法，如果没有，会在控制台给出警告；

（2）验证是否使用的已经废弃的方法，如果有，会在控制台给出警告；

（3）通过识别潜在的风险预防一些副作用。

\*/

**★★★ 为什么类方法需要绑定到类实例？**

// 在 JS 中，this 值会根据当前上下文变化。在 React 类组件方法中，开发人员通常希望 this 引用组件的当前实例，因此有必要将这些方法绑定到实例。通常这是在构造函数中完成的:

**★★★★ 什么是 prop drilling，如何避免？**

/\*

从一个外部组件一层层将prop传递到内部组件很不方便，这个问题就叫做 prop drilling；主要缺点是原本不需要数据的组件变得不必要地复杂，并且难以维护，代码看起来也变得冗余，不优雅；

为了避免prop drilling，一种常用的方法是使用React Context。通过定义提供数据的Provider组件，并允许嵌套的组件通过 Consumer组件或 useContext Hook 使用上下文数据。

\*/

**★★ 描述 Flux 与 MVC？**

/\*

传统的 MVC 模式在分离数据(Model)、UI(View和逻辑(Controller)方面工作得很好，但是 MVC 架构经常遇到两个主要问题:

数据流不够清晰——跨视图发生的级联更新常常会导致混乱的事件网络，难于调试。

缺乏数据完整性——模型数据可以在任何地方发生突变，从而在整个UI中产生不可预测的结果。

使用 Flux 模式的复杂用户界面不再遭受级联更新，任何给定的React 组件都能够根据 store 提供的数据重建其状态。Flux 模式还通过限制对共享数据的直接访问来加强数据完整性。

\*/

**★★★ 这段代码有什么问题吗？**

this.setState((prevState, props) => {

return {

streak: prevState.streak + props.count

}

})

// 没有问题

**★★★★ 什么是 React Context?**

[React Context](https://zh-hans.reactjs.org/docs/context.html" \l "gatsby-focus-wrapper" \t "_blank)

***[源码解析](https://react.jokcy.me/book/features/context.html" \t "_blank)***

##### What

**Context**提供了一个无需为每层组件手动添加props，就能在**组件树**间进行数据传递的功能

##### Why

某些全局属性，通过父子props传递太过繁琐，Context提供了一种组件之间共享此类值的方式，而不必显式的通过组件树逐层传递props

##### When

共享那些对于一个组件树而言是**全局**的数据，例如当前认证的用户、主题或者首选语言等

##### Where

* Context应用场景在于很多不同层级的组件访问同样的数据，这样也使得组件的复用性变差。
* **如果你只是想避免层层传递一些属性，组件组合有时候是一个比Context更好的方案**，也就是直接传递组件
* 所以一个技术方案的选定需要针对不同的场景具体分析，采取合适的方案

##### How

// ①创建

const ThemeContext = React.createContext('xxx')

// ②注入---提供者 在入口或者你想要注入的父类中，且可以嵌套，里层覆盖外层

return (

<ThemeContext.Provider value="yyy">

{children}

<ThemeContext.Provider>

)

// ③使用---消费者 需要使用共享数据的子类中

// 方式一

static contextType = ThemeContext

// 方式二

Class.contextType = ThemeContext

render() {

let value = this.context

/\* 基于这个值进行渲染工作 \*/

}

// 方式三

return(

<ThemeContext.Consumer>

{ value => /\* 基于 context 值进行渲染\*/ }

</ThemeContext.Consumer>

)

##### More

**动态Context**---类似父子组件

// ①创建

const ThemeContext = React.createContext({

value: 'xxx',

changeFunc: () => {} //通过context传递这个函数，让consumers组件更新context

})

// ②注入

return (

<ThemeContext.Provider value="yyy">

<Child changeFunc={this.changeFunc}>

<ThemeContext.Provider>

)

// ③消费

return(

<ThemeContext.Consumer>

{ ({value, changeFunc}) => /\* 基于 context 值进行渲染,同时把changeFunc绑定\*/ }

</ThemeContext.Consumer>

)

**消费多个Context、注意事项等**[参考React中文网](https://zh-hans.reactjs.org/docs/context.html" \l "consuming-multiple-contexts" \t "_blank)

**★★★★★ 什么是 React Fiber?**

##### What

[React Fiber En](https://github.com/acdlite/react-fiber-architecture" \t "_blank)

* Fiber是React16中新的协调引擎，它的主要目的是使Virtual DOM可以进行增量式渲染，让界面渲染更流畅
* 一种流程控制原语，也称为协程，可以类比es6中的generator函数；React渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复渲染。
* 一个执行单元，每次执行完一个“执行单元”，React就会检查现在还剩多少时间，如果没有时间就将控制权让出去。
* 目标
* 把可中断的工作拆分成小任务
* 对正在做的工作调整优先次序、重做、复用上次（做了一半的）成果
* 在父子任务之间从容切换（yield back and forth），以支持React执行过程中的布局刷新
* 支持render()返回多个元素
* 更好地支持error boundary
* 特性
* 增量渲染（把渲染任务拆分成块，匀到多帧）
* 更新时能够暂停，终止，复用渲染任务
* 给不同类型的更新赋予优先级
* 并发方面新的基础能力

**★★★ 如何在 React 的 Props 上应用验证？**

使用PropTypes进行类型检查

[PropTypes](https://zh-hans.reactjs.org/docs/typechecking-with-proptypes.html" \l "gatsby-focus-wrapper" \t "_blank)自React v15.5起，请使用这个库[prop-types](https://www.npmjs.com/package/prop-types" \t "_blank)

##### What & Why & When

* 随着应用的不断增长，也是为了使程序设计更加严谨，我们通常需要对数据的类型（值）进行一些必要的验证
* 出于性能方面的考虑，propTypes仅在开发模式下进行检测,在程序运行时就能检测出错误，不能使用到用户交互提醒用户操作错误等
* 也可以使用[Flow](https://flow.org/" \t "_blank)或者[TypeScript](https://www.typescriptlang.org/" \t "_blank)做类型检查，后期建议用typescript进行替代更好

##### Where

* class组件
* 函数组件
* [React.memo](https://zh-hans.reactjs.org/docs/react-api.html" \l "reactmemo" \t "_blank)高阶组件 **可自行扩展**
* [React.forwardRef](https://zh-hans.reactjs.org/docs/react-api.html" \l "reactforwardref" \t "_blank)组件 **可自行扩展**

##### How

我们在组件类下添加一个静态属性 propTypes (属性名不能更改)，它的值也是一个对象，用来设置组件中props的验证规则，key 是要验证的属性名称，value 是验证规则。

// 类组件

import PropTypes from 'prop-types';

class Greeting extends React.Component {

render() {

return (

<h1>Hello, {this.props.name}</h1>

);

}

}

// 指定 props 的默认值：

Greeting.defaultProps = {

name: 'Stranger'

};

// 类组件在这里做检测

Greeting.propTypes = {

// v15.4 and below

// name: React.PropTypes.string

name: PropTypes.string

};

// 函数组件

unction HelloWorldComponent({ name }) {

return (

<div>Hello, {name}</div>

)

}

// 函数组件在这里做检测

HelloWorldComponent.propTypes = {

name: PropTypes.string

}

export default HelloWorldComponent

[更多检测的类型](https://zh-hans.reactjs.org/docs/typechecking-with-proptypes.html" \l "proptypes" \t "_blank)

##### More

* 限制单个元素 [PropTypes.element](https://zh-hans.reactjs.org/docs/typechecking-with-proptypes.html" \l "requiring-single-child" \t "_blank)

**★★ 在 React 中使用构造函数和 getInitialState 有什么区别？**

[参考-stackoverflow](https://stackoverflow.com/questions/30668326/what-is-the-difference-between-using-constructor-vs-getinitialstate-in-react-r" \t "_blank)

// ES6

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = { /\* initial state \*/ };

}

}

// ES5

var MyComponent = React.createClass({

getInitialState() {

return { /\* initial state \*/ };

},

});

* 区别在于ES6和ES5本身，**getInitialState** 是搭配 **React.createClass** 使用的， **constructor** 是搭配 **React.Component** 使用的
* 在React组件的生命周期中 **constructor** 先于 **getInitialState**

**★★★ 如何有条件地向 React 组件添加属性？**

* 对于某些属性，React足够智能可以忽略该属性，比如值为boolean值属性的值
* 也可以写控制语句管理是否给组件添加属性

**★★★★ Hooks 会取代 `render props` 和高阶组件吗？**

* 可以取代，但没必要
* 在Hook的渐进策略中也有提到，没有计划从React中移除class，在新的代码中同时使用Hook和class，所以这些方案目前还是可以有勇武之地

##### What

* 为什么要把这3种技术拿过来对比？

都在处理同一个问题，**逻辑复用**

* [高阶组件HOC](https://zh-hans.reactjs.org/docs/higher-order-components.html" \t "_blank)---不是 React API 的一部分，是基于 React 的**组合特性**形成的设计模式。

高阶组件是参数为组件，返回值为新组件的**函数**（将组件转换为另一个组件，纯函数，无副作用）

* [Render Props](https://zh-hans.reactjs.org/docs/render-props.html" \l "gatsby-focus-wrapper" \t "_blank)

是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的 **简单技术？**

* [Hooks](https://zh-hans.reactjs.org/docs/hooks-intro.html" \t "_blank)

React16.8新增的特性，是一些可以让你在函数组件里“钩入”React state及生命周期等特性的函数，

##### Why

虽然 **HOC** & **Render Props** 能处理逻辑复用的问题，但是却存在各自的问题。

**HOC** 存在的问题

* 写法破坏了原来组件的结构，DevTools中组件会形成“嵌套地狱”
* 不要在 **render** 方法中使用 **HOC** 每次调用render函数会创建一个新的高阶组件导致该组件及其子组件的状态丢失
* 需要修复静态方法，即拷贝原组件的静态方法到高级组件中
* 如需传递Ref则需要通过React.forwardRef创建组件

**Render Props** 存在的问题

* 同样的写法会破坏原来组件的结构，DevTools中组件会形成“嵌套地狱”
* 与React.PureComponent组件使用有冲突

**Hook** 目前最优雅的实现，React为共享状态逻辑提供最好的原生途径

* 没有破坏性改动，完全可选，100%向后兼容
* 解决复杂组件，中逻辑状态、副作用和各种生命周期函数中逻辑代码混在一起，难以拆分，甚至形成bug的问题
* 处理class组件中

##### When

* 在函数组件中意识到要向其添加一些state---useState
* 有副作用的行为时

##### Where

* 只能在函数最外层调用Hook，不要在循环、条件判断或者子函数中调用
* 只能在函数组件或者自定义Hook中调用Hook

##### How

**★★★ 如何避免组件的重新渲染？**

##### Answer

当porps/state改变时组件会执行render函数也就是重新渲染

* class组件中 使用shouldComponentUpdate钩子函数
* PureComponent默认有避免重新渲染的功能
* 函数组件使用高阶组件memo处理

**★★★ 什么是纯函数？**

##### Answer

一个不会更改入参，且多次调用下相同的入参始终返回相同的结果

一个函数的返回结果只依赖于它的参数，并且在执行的过程中没有副作用，我们就把该函数称作纯函数

特点：

**1. 函数的返回结果只依赖于它的参数。**

1. **函数执行过程里面没有副作用。（**除了修改外部的变量，一个函数在执行过程中还有很多方式产生外部可观察的变化，比如说调用 DOM API 修改页面，或者你发送了 Ajax 请求，还有调用 window.reload 刷新浏览器，甚至是 console.log 往控制台打印数据也是副作用。**）**

**3.没有额外的状态依赖（**指方法内的状态都只在方法的生命周期内存活，这意味着不能在方法内使用共享变量，因为会带来不可知因素。**）**

**★★★★ 当调用`setState`时，React `render` 是如何工作的？**

##### Answer

**调用setState()**

* 检查上下文环境生成更新时间相关参数并判定事件优先级（fiber，currenttime，expirationtime等…）
* 根据优先级相关参数判断更新模式是sync（同步更新）或是batched（批量处理）
* 加入执行更新事件的队列，生成事件队列的链表结构
* 根据链表顺序执行更新

setState既是同步的，也是异步的。同步异步取决于setState运行时的上下文。且setState 只在合成事件和钩子函数中是“异步”的，在原生DOM事件和 setTimeout 中都是同步的

**render如何工作**

* React在props或state发生改变时，会调用React的render方法，创建一颗不同的树
* React需要基于这两颗不同的树之间的差别来判断如何有效的更新UI
* diff算法，将两颗树完全比较更新的算法从O(n^3^),优化成O(n)；
* 同层节点之间相互比较，不会跨节点比较
* 不同类型的节点，产生不同的树结构
* 设置key来指定节点在不同的渲染下保持稳定

**★★★ 如何避免在React重新绑定实例？**

##### Answer

* 将事件处理程序定义为内联箭头函数
* 使用箭头函数来定义方法
* 使用带有 Hooks 的函数组件

**★★★ 在js原生事件中 onclick 和 jsx 里 onclick 的区别**

##### Answer

**js原生中**

* onclick添加事件处理函数是在全局环境下执行，污染了全局环境，
* 且给很多dom元素添加onclick事件，影响网页的性能，
* 同时如果动态的从dom树种删除了该元素，还要手动注销事件处理器，不然就可能造成内存泄露

**jsx里的onClick**

* 挂载的函数都控制在组件范围内，不会污染全局空间
* jsx中不是直接使用onclick，而是采取了事件委托的方式，挂载最顶层DOM节点，所有点击事件被这个事件捕获，然后根据具体组件分配给特定函数，性能当然比每个onClick都挂载一个事件处理函数要高
* 加上React控制了组件的生命周期，在unmount的时候自然能够清楚相关的所有事件处理函数，内存泄露不再是一个问题

**★★★★ diff复杂度原理及具体过程画图**

##### Answer

* 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；
* React 通过分层求异的策略，对 tree diff 进行算法优化；
* 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；
* React 通过设置唯一 key的策略，对 element diff 进行算法优化；
* 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；
* 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。

**★★★★ shouldComponentUpdate的作用是什么？**

[shouldComponentUpdate的作用](https://zh-hans.reactjs.org/docs/optimizing-performance.html" \l "shouldcomponentupdate-in-action" \t "_blank)

##### What

* 不常用的生命周期方法，能影响组件是否重新渲染
* 在更新阶段，当有new props 或者 调用了 setState()方法，在render方法执行前会执行到，默认返回值为true，如果返回false则不刷新组件

##### Why & When & Where

* 如果你知道在什么情况下组件不需要更新，你可以让其返回值为false跳过整个渲染过程
* 次方法仅作为 **性能优化方式** 而存在，不要企图靠此方法来阻止渲染，
* 大部分情况下，使用PureComponent代替手写shouldComponentUpdate，仅浅层对比
* 不建议在shoulComponentUpdate中进行深层或者使用JSON.stringify()，这样非常影响效率和性能

##### Answer

* 作为React组件中不常用的生命周期函数，能影响组件是否重渲染
* 建议做浅层次的比较，来优化性能，当然这里也可以用PureComponent组件代替
* 如果有较深层次的比较则可能会导致更严重的性能问题，因此在这种情况下不要靠手动管理组件的重新渲染来优化性能，要找其他方式
* **比如？**

**★★★ React组件间信息传递**

##### Answer

* 1.（父组件）向（子组件）传递信息 : **porps传值**
* 2.（父组件）向更深层的（子组件） 进行传递信息 : **context**
* 3.（子组件）向（父组件）传递信息：**callback**
* 4.没有任何嵌套关系的组件之间传值（比如：兄弟组件之间传值）: **利用共同父组件context通信、自定义事件**
* 5.利用react-redux进行组件之间的状态信息共享 : **组件间状态信息共享：redux、flux、mobx等**

**★★★ React状态管理工具有哪些？redux actionCreator都有什么？**

##### Answer

* 简单状态管理：组件内部state、基\*于Context API封装、
* 复杂状态管理：redux(单项数据流)、mobx(响应式数据流)、RxJS(stream)、dva
* 创建各种action，包含同步、异步，然后在组件中通过dispatch调用

**★★★★ 什么是高阶组件、受控组件及非受控组件？都有啥区别**

##### Answer

**定义**

* [高阶组件HOC](https://zh-hans.reactjs.org/docs/higher-order-components.html" \t "_blank)---不是 React API 的一部分，是基于 React 的**组合特性**形成的设计模式。

高阶组件是参数为组件，返回值为新组件的**函数**（将组件转换为另一个组件，纯函数，无副作用）

* [受控组件](https://zh-hans.reactjs.org/docs/forms.html" \l "controlled-components" \t "_blank)

在表单元素中，state是唯一数据源，渲染表单的React组件控制着用户输入过程中表单发生的操作。被React以这种方式控制取值的表单输入元素叫做受控组件

* [非受控组件](https://zh-hans.reactjs.org/docs/uncontrolled-components.html" \t "_blank)

表单数据由DOM节点来处理，而不是用state来管理数据，一般可以使用ref来从DOM节点中获取表单数据

**区别**

* 受控组件和非受控组件是表单中的组件，高阶组件相当于对某个组件注入一些属性方法
* 高阶组件是解决代码复用性问题产生的技术
* 受控组件必须要有一个value，结合onChange来控制这个value，取值为event.target.value/event.target.checked
* 非受控组件相当于操作DOM，一般有个defaultValue，通过onBlur触发响应方法

**★★★ vuex 和 redux 的区别？**

##### Answer

**Redux**

* Redux 是 JavaScript 状态容器，提供可预测化的状态管理。

**Redux**

* 随着JS单页面应用日趋复杂，JS需要管理比任何时候都要多的state（服务器响应，缓存数据，本地生成尚未持久化到服务器的数据，UI状态等）
* 伴随的现象就是models和views相互影响，你难以弄清楚变化的源头
* 也就是变化和异步让我们的state变得一团糟

**★★★ Redux遵循的三个原则是什么？**

##### Answer

**单一数据源**

* 整个应用的state被存储在一棵object tree中，并且整个 object tree 只存在于唯一一个 store 中

**State是只读的**

* 唯一改变state的方法就是触发 action，action是一个描述已发生事件的普通对象
* 这样确保视图和网络请求不能直接修改state

**使用纯函数来执行修改**

* 为了描述action如何改变state tree，你需要编写reducers

**★★★ redux中使用setState不能立刻获取值，怎么办**

##### Answer

setState 只在合成事件和钩子函数中是异步的，在原生事件和 setTimeout 中都是同步

* ①addeventListener添加的事件或者dom事件中触发
* ②setState接收的参数还可以是一个函数，在这个函数中可以拿先前的状态，并通过这个函数的返回值得到下一个状态。

this.setState((preState) => {

return {

xxx: preState.xxx + yyy

}

})

* ③async/await 异步调用处理

**★★ 什么是JSX**

##### Answer

当 Facebook 第一次发布 React 时，他们还引入了一种新的 JS 方言 JSX，将原始 HTML 模板嵌入到 JS 代码中。JSX 代码本身不能被浏览器读取，必须使用Babel和webpack等工具将其转换为传统的JS。很多开发人员就能无意识使用 JSX，因为它已经与 React 结合在一直了

* 是一个 JavaScript 的语法扩展
* 具有 JavaScript 的全部功能
* 可以生成 React “元素”
* JSX 也是一个表达式

**★★★ React新老版生命周期函数**

[生命周期](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/" \t "_blank)

##### Answer

**New Version**

* 挂载：constructor --> getDerivedStateFromProps --> render --> componentDidMount
* 更新：
* New props、setState() --> getDerivedStateFromProps --> shouldComponentUpdate --> render --> getSnapshotBeforeUpdate --> componentDidUpdate
* forceUpdate() --> getDerivedStateFromProps --> render --> getSnapshotBeforeUpdate --> componentDidUpdate
* 卸载： componentWillUnmount

**Old Version\***

* 挂载：constructor --> getDerivedStateFromProps --> render --> ComponentDidMount
* 更新：
* New props --> getDerivedStateFromProps --> shouldComponentUpdate --> render --> getSnapshotBeforeUpdate --> componentDidUpdate
* setState() --> shouldComponentUpdate --> render --> getSnapshotBeforeUpdate --> componentDidUpdate
* forceUpdate() --> render --> getSnapshotBeforeUpdate --> componentDidUpdate
* 卸载：componentWillUnmount

**★★★★ vue react都怎么检测数据变化**

##### Answer

* React

React默认是通过比较引用的方式（diff）进行的，不精确监听数据变化，如果不优化可能导致大量不必要的VDOM重新渲染

* 16之前 componentWillReveiveProps 监听 props 变化
* 16之后 getDerivedStateFromProps 监听 props
* Vue
* vue监听变量变化依靠 watch Object.defineProperty，Vue通过“getter/setter”以及一些函数的劫持，能精确知道数据变化

**★★★ React中怎么让 setState 同步更新？**

##### Answer

* setState 回调，setState,第二个参数是一个回调函数，可实现同步
* 引入 Promise 封装 setState，在调用时我们可以使用 Async/Await 语法来优化代码风格

setStateAsync(state) {

return new Promise((resolve) => {

this.setState(state, resolve)

});

}

* 传入状态计算函数， setState 的第一个参数，

this.setState((prevState, props) => ({

count: prevState.count + 1

}));

* 在 setTimeout 函数中调用 setState
* more？

**★★★★ 什么是 immutable？为什么要使用它？**

##### Answer

immutable是一种持久化数据。一旦被创建就不会被修改。修改immutable对象的时候返回新的immutable。但是原数据不会改变。

在Rudux中因为深拷贝对性能的消耗太大了（用到了递归，逐层拷贝每个节点）。 但当你使用immutable数据的时候：只会拷贝你改变的节点，从而达到了节省性能。 总结：immutable的不可变性让纯函数更强大，每次都返回新的immutable的特性让程序员可以对其进行链式操作，用起来更方便。

因为在react中，react的生命周期中的setState()之后的shouldComponentUpdate()阶段默认返回true，所以会造成本组件和子组件的多余的render，重新生成virtual dom，并进行virtual dom diff，所以解决办法是我们在本组件或者子组件中的shouldComponentUpdate()函数中比较，当不需要render时，不render。

当state中的值是对象时，我们必须使用深拷贝和深比较！

如果不进行深拷贝后再setState，会造成this.state和nextState指向同一个引用，所以shouldComponentUpdate()返回值一定是false，造成state值改了，而组件未渲染（这里不管shouldComponentUpdate中使用的是深比较还是浅比较）。所以必须深拷贝。

如果不在shouldComponentUpdate中进行深比较，会造成即使state中的对象值没有改变，因为是不同的对象，而在shouldComponentUpdate返回true，造成不必要的渲染。

所以只能是深拷贝和深比较。

**★★★ 为什么不建议在 componentWillMount 做AJAX操作**

##### Answer

* Fiber原因，React16之后，采用了Fiber架构，只有componentDidMount的生命周期函数确定会执行一次，其他像componentWillMount可能会执行多次
* render 阶段 可能会被React暂停，中止或重启

**★★★★ 如何在React中构建一个弹出的遮罩层**

##### Answer

//css部分

.mask{

background: rgba(0,0,0,0.4) !important;

z-index: 10;

height: 100vh;

position: fixed;

width: 100vw;

}

.selectMask\_box{

background: rgba(0,0,0,0);

transition: all .2s linear

}

//js部分

handleMask=()=>{

this.setState({

dateSelected: !this.state.dateSelected

})

}

<div

onClick={this.handleMask}

className={`selectMask\_box ${this.state.dateSelected ? "mask" : ""} `} >

//这里是待展示的内容，<div>...</div>

//你可以自己设置dataSelected的初始值，同时请注意注意三元运算的顺序。

</div>

**★★★★★ React中的Context的使用**

[React Context](https://zh-hans.reactjs.org/docs/context.html" \l "gatsby-focus-wrapper" \t "_blank)

***[源码解析](https://react.jokcy.me/book/features/context.html" \t "_blank)***

##### Answer

// ①创建

const ThemeContext = React.createContext('xxx')

// ②注入---提供者 在入口或者你想要注入的父类中，且可以嵌套，里层覆盖外层

return (

<ThemeContext.Provider value="yyy">

{children}

<ThemeContext.Provider>

)

// ③使用---消费者 需要使用共享数据的子类中

// 方式一

static contextType = ThemeContext

// 方式二

Class.contextType = ThemeContext

render() {

let value = this.context

/\* 基于这个值进行渲染工作 \*/

}

// 方式三

return(

<ThemeContext.Consumer>

{ value => /\* 基于 context 值进行渲染\*/ }

</ThemeContext.Consumer>

)

**动态Context**---类似父子组件

// ①创建

const ThemeContext = React.createContext({

value: 'xxx',

changeFunc: () => {} //通过context传递这个函数，让consumers组件更新context

})

// ②注入

return (

<ThemeContext.Provider value="yyy">

<Child changeFunc={this.changeFunc}>

<ThemeContext.Provider>

)

// ③消费

return(

<ThemeContext.Consumer>

{ ({value, changeFunc}) => /\* 基于 context 值进行渲染,同时把changeFunc绑定\*/ }

</ThemeContext.Consumer>

)

**消费多个Context、注意事项等**[参考React中文网](https://zh-hans.reactjs.org/docs/context.html" \l "consuming-multiple-contexts" \t "_blank)

**★★★★ React路由懒加载的实现**

##### Answer

* **原理**
* webpack代码分割
* React利用 React.lazy与import()实现了渲染时的动态加载
* 利用Suspense来处理异步加载资源时页面应该如何显示的问题
* 1.React.lazy
* 通过lazy() api来动态import需要懒加载的组件
* import的组件目前只支持export default的形式导出
* Suspense来包裹懒加载的组件进行加载，可以设置fallback现实加载中效果
* React.lazy可以结合Router来对模块进行懒加载。

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'))

const AnyComponent = lazy(() => import('./routes/AnyComponent'))

...

return (

<Router>

<Suspense fallback={<div>Loading...</div>}>

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/anyManage" component={AnyComponent}/>

...

</Switch>

</Suspense>

</Router>

* 2.react-loadable

react-loadable是以组件级别来分割代码的，这意味着，我们不仅可以根据路由按需加载，还可以根据组件按需加载，使用方式和路由分割一样，只用修改组件的引入方式即可

// 路由懒加载（异步组件）

import Loadable from 'react-loadable';

//通用过场组件

const LoadingComponent = () => {

return (

<div>loading</div>

)

}

...

export default (loader, loading=LoadingComponent) => {

return Loadable({

loader,

loading

})

}

//Route中调用

import { BrowserRouter, Route } from 'react-router-dom'

const loadable from './loadable';

const AnyComponent = loadable(() => import('./AnyComponent'))

const Routes = () => (

<BrowserRouter>

<Route path="/home" component={AnyComponent}/>

</BrowserRouter>

);

export default Routes;

##### 以下是老版中的方法

* 3.webpack配置中使用lazyload-loader

// webpack 配置中

module: {

rules: [

{

test: /.(js|jsx)$/,,

use: [

'babel-loader',

'lazyload-loader'

]

},

// 业务代码中

// 使用lazy! 前缀 代表需要懒加载的Router

import Shop from 'lazy!./src/view/Shop';

// Router 正常使用

<Route path="/shop" component={Shop} />

* 4.import() webpack v2+

符合ECMAScript提议的import()语法，该提案与普通 import 语句或 require 函数的类似，但返回一个 Promise 对象

function component() {

return import( /\* webpackChunkName: "lodash" \*/ 'lodash').then(\_ => {

var element = document.createElement('div');

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}).catch(error => 'An error occurred while loading the component');

}

// 或者使用async

async function getComponent() {

var element = document.createElement('div');

const \_ = await import(/\* webpackChunkName: "lodash" \*/ 'lodash');

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

* 5.requre.ensure webpack v1 v2

require.ensure([], function(require){

var list = require('./list');

list.show();

，'list');

<!-- Router -->

const Foo = require.ensure([], () => {

require("Foo");

}, err => {

console.error("We failed to load chunk: " + err);

}, "chunk-name");

//react-router2 or 3

<Route path="/foo" getComponent={Foo} />

**★★★★ React-router-dom内部是怎么样实现的，怎么做路由守卫？**

##### Answer

##### 内部实现

* 总

react-router-dom利用了Context API，通过上下文对象将当前路由信息对象注入到 Router 组件中，所以 Router 组件中 render() 渲染的内容就是 ContextAPI 提供的 Provider 组件，然后接收 Router 组件中的当前路由信息对象。 这样 Router 组件下的所有组件都能通过上下文拿到当前路由信息对象，即其中的Switch 、 Route 、 Link 、Redirect 等组件都可以拿到当前路由信息对象，然后通过改变当前路由信息来实现动态切换 Route 组件的渲染。

* 分
* RouterContext：react-router使用context实现跨组件间数据传递，所以react-router定义了一个routerContext作为数据源，
* Router：BrowserRouter和HashRouter将当前路由注入到上下文中，同时路由信息包含location、match、history
* Route：路由规则，获取RouterContext的信息(location对象)，获取path和component属性，判断path和当前的location是否匹配，如果匹配，则渲染component，否则返回null，不渲染任何内容
* Switch：遍历所有子元素(Route)，判断Route的path和location是否匹配，如果匹配，则渲染，否则不渲染
* Redireact：未能配则重定向到指定页面
* Link/NavLink: Link组件本质就是a标签，它修改了a标签的默认行为，当点击Link时，会导航到对应的路由，导致locaiton对象的改变，出发组件的更新
* withRouter：对传入的组件进行加强，功能就是获取routerContext上面的信息，然后作为props传给需要加强的组件

##### 怎么做路由守卫

* 路由里设置meta元字符实现路由拦截
* React Router 4.0之前也像vue中一样有个钩子函数 onEnter 可实现
* ReactRouter 4.0开始自己实现如下

// routerMap.js中

import Index from './page/index'

export default [

{ path:'/', name: 'App', component:Index, auth: true },

...

]

//入口文件 app.js中

import { BrowserRouter as Router, Switch } from "react-router-dom";

import FrontendAuth from "./FrontendAuth";

import routerMap from "./routerMap";

...

return (

<Router>

<div>

<Switch>

<FrontendAuth routerConfig={routerMap} />

</Switch>

</div>

</Router>

)

// 高阶组件FrontendAuth 处理路由跳转，即路由守卫功能

//FrontendAuth.js

import React, { Component } from "react";

import { Route, Redirect } from "react-router-dom";

class FrontendAuth extends Component {

// eslint-disable-next-line no-useless-constructor

constructor(props) {

super(props);

}

render() {

const { routerConfig, location } = this.props;

const { pathname } = location;

const isLogin = sessionStorage.getItem("username");

console.log(pathname, isLogin);

console.log(location);

// 如果该路由不用进行权限校验，登录状态下登陆页除外

// 因为登陆后，无法跳转到登陆页

// 这部分代码，是为了在非登陆状态下，访问不需要权限校验的路由

const targetRouterConfig = routerConfig.find(

(item) => item.path === pathname

);

console.log(targetRouterConfig);

if (targetRouterConfig && !targetRouterConfig.auth && !isLogin) {

const { component } = targetRouterConfig;

return <Route exact path={pathname} component={component} />;

}

if (isLogin) {

// 如果是登陆状态，想要跳转到登陆，重定向到主页

if (pathname === "/login") {

return <Redirect to="/" />;

} else {

// 如果路由合法，就跳转到相应的路由

if (targetRouterConfig) {

return (

<Route path={pathname} component={targetRouterConfig.component} />

);

} else {

// 如果路由不合法，重定向到 404 页面

return <Redirect to="/404" />;

}

}

} else {

// 非登陆状态下，当路由合法时且需要权限校验时，跳转到登陆页面，要求登陆

if (targetRouterConfig && targetRouterConfig.auth) {

return <Redirect to="/login" />;

} else {

// 非登陆状态下，路由不合法时，重定向至 404

return <Redirect to="/404" />;

}

}

}

}

export default FrontendAuth;

* 总结一下，实现路由守卫需要考虑到以下的问题：
* 未登录情况下，访问不需要权限校验的合法页面：允许访问
* 未登录情况下，访问需要权限校验的页面：禁止访问，跳转至登陆页
* 未登录情况下，访问所有的非法页面：禁止访问，跳转至 404
* 登陆情况下，访问登陆页面：禁止访问，跳转至主页
* 登陆情况下，访问除登陆页以外的合法页面：允许访问
* 登陆情况下，访问所有的非法页面：禁止访问，跳转至 404

**★★★★ redux中sages和thunk中间件的区别，优缺点**

##### Answer

* 区别
* redux-thunk异步采取 async/await redux-saga采取generate函数
* 优缺点
* redux-thunk

优点： 库小，代码就几行 缺点：代码臃肿，reducer不再是纯粹函数，直接返回对象，违背了当初的设计原则；action的形式不统一，异步操作太为分散，分散在了各个action中

* redux-saga

优点： 将异步与reducer区分开了，更加优雅，适合大量APi请求，而且每个请求之间存在复杂的以来关系 缺点：学习曲线比较陡，理解async await；而且库也比较大，即使发布的最小也有25kb,gzip压缩后也有7KB,React压缩后才45kb

**★★ 为什么说React是view（视图层）**

##### Answer

* react, 是 Facebook 推出的一个用来构建用户界面的 JavaScript 库. React 主要用于构建 UI
* React被认为是视图层的框架是因为它是基于组件的，一切都是组件，而组件就是渲染页面的基础。不论组件中包含的jsx，methods，state，props，都是属于组件内部的
* View（视图）是应用程序中处理数据显示的部分。视图层主要包括二个部分：1.视图层显示及交互逻辑；2.视图层的数据结构ViewObj， 包括React中的props和stats；

**★★★ 怎么用useEffect模拟生命周期函数？**

[useEffect](https://react.docschina.org/docs/hooks-effect.html" \t "_blank)

###### Answer

* ①默认情况下，它在第一次渲染之后和每次更新之后都会执行，无需清除的effect

// 在函数式组件中 在 return之前

// Similar to componentDidMount and componentDidUpdate:

useEffect(() => {

// Update the document title using the browser API

document.title = `You clicked ${count} times`;

});

* ②需要清除的effect：React 会在组件卸载的时候执行清除操作

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

// Specify how to clean up after this effect:

return function cleanup() {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

* ③使用多个effect实现关注点的分离

把类组件中的分散在多个生命周期中的同一件事件的处理，合并到同一个effect中处理

* ④通过跳过effect进行性能优化

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [count]); // 仅在 count 更改时更新

**★★★ useCallback是干什么的？使用useCallback有什么好处？**

[useCallback](https://zh-hans.reactjs.org/docs/hooks-reference.html" \l "usecallback" \t "_blank)

##### Answer

把内联回调函数及依赖项数组作为参数传入 useCallback，它将返回该回调函数的 memoized(缓存)版本，该回调函数仅在某个依赖项改变时才会更新 **好处** 当你把回调函数传递给经过优化的并使用引用相等性去避免非必要渲染（例如 shouldComponentUpdate）的子组件时，它将非常有用

**★★★ 能简单说一下redux-sage的使用流程吗？**

[redux-saga](https://redux-saga.js.org/" \t "_blank)

##### Answer

redux-saga 是一个用于管理 Redux 应用异步操作的中间件（又称异步 action）。 redux-saga 通过创建 Sagas 将所有的异步操作逻辑收集在一个地方集中处理，可以用来代替 redux-thunk 中间件。

* Reducers 负责处理 action 的 state 更新
* Sagas 负责协调那些复杂或异步的操作
* ①connet to the store：本质是管理 Redux 应用异步操作的中间件

import { createStore, applyMiddleware } from 'redux'

import createSagaMiddleware from 'redux-saga'

import reducer from './reducers'

import mySaga from './sagas'

// Create the saga middleware

const sagaMiddleware = createSagaMiddleware()

// Mount it on the Store

const store = createStore(

reducer,

applyMiddleware(sagaMiddleware)

)

// Then run the saga

sagaMiddleware.run(mySaga)

// Render the application

* ②initiate a side effect：初始化副作用

import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'

import Api from '...'

// Worker saga will be fired on USER\_FETCH\_REQUESTED actions

function\* fetchUser(action) {

try {

const user = yield call(Api.fetchUser, action.payload.userId);

yield put({type: "USER\_FETCH\_SUCCEEDED", user: user});

} catch (e) {

yield put({type: "USER\_FETCH\_FAILED", message: e.message});

}

}

// Starts fetchUser on each dispatched USER\_FETCH\_REQUESTED action

// Allows concurrent fetches of user

function\* mySaga() {

yield takeEvery("USER\_FETCH\_REQUESTED", fetchUser);

}

* ③dispath an action：组件中使用

class UserComponent extends React.Component {

...

onSomeButtonClicked() {

const { userId, dispatch } = this.props

dispatch({type: 'USER\_FETCH\_REQUESTED', payload: {userId}})

}

...

}

* ④more：takeEvery、takeLatest、take、put、call、fork、select

**★★★★ React复用组件的状态和增强功能的方法**

##### Answer

* ①render props模式
* 创建Mouse组件，在组件中提供复用的状态逻辑代码
* 将要复用的状态作为 props.render(state) 方法的参数，暴露到组件外部
* 使用 props.render() 的返回值作为要渲染的内容

// 子组件

class Mouse extends React.Component{

// mouse本组件的数据

state={

x:0,

y:0

}

// mouse本组件的方法

handleMouse=(e)=>{

this.setState({

x:e.clientX,

y:e.clientY

})

}

componentDidMount(){

window.addEventListener('mousemove',this.handleMouse)

};

render(){

// 在这里用props接收从父传过来的render函数，再把state数据作为实参传递出去

// 其实渲染的就是从父传过来的UI结构，只是公用了Mouse组件的数据和方法

return this.props.render(this.state)

}

}

// 父组件

class App extends React.Component{

render(){

return(<div>

<h1>app组件:{this.props.name}</h1>

{/\* 在使用mouse组件时，给mouse传递一个值（父传子），

只不过这里的props是函数，这个函数将要用形参接受从mouse组件传递过来的实参（state数据） \*/}

<Mouse render={(mouse)=>{

return (<div>

<h5>我用的是Mouse的state和方法：X坐标{mouse.x}-Y坐标{mouse.y}</h5>

</div>)

}} />

</div>)

}

}

* ②高阶组件 HOC

const EnhancedComponent = withHOC(WrappedComponent)

// 高阶组件内部创建的类组件：

class Mouse extends React.Component {

render() {

return <WrappedComponent {...this.state} />

}

}

* ③hooks：自定义hook

// 自定义hook

function useFriendStatus(friendID) {

const [isOnline, setIsOnline] = useState(null);

// ...

return isOnline;

}

// 使用

function FriendStatus(props) {

const isOnline = useFriendStatus(props.friend.id);

if (isOnline === null) {

return 'Loading...';

}

return isOnline ? 'Online' : 'Offline';

}

**★★★ redux 和 mobx 的区别**

##### Answer

* ①Redux的编程范式是函数式的而Mobx是面向对象的
* ②因此数据上来说Redux理想的是immutable的，每次都返回一个新的数据，而Mobx从始至终都是一份引用。因此Redux是支持数据回溯的
* ③然而和Redux相比，使用Mobx的组件可以做到精确更新，这一点得益于Mobx的observable；对应的，Redux是用dispatch进行广播，通过Provider和connect来比对前后差别控制更新粒度，有时需要自己写SCU；Mobx更加精细一点
* ④Mobx-react vs React-rdux：

redux,采取Provider和connect方式，mobx采取Provider和inject、observer

**★★★ react中如何实现命名插槽**

##### Answer

import React, { Component } from 'react';

import ReactDOM from 'react-dom';

class ParentCom extends React.Component {

constructor(props) {

super(props)

console.log(props);

}

render() {

return (

<div>

<h1>组件插槽</h1>

{this.props.children}

<ChildCom>

<h1 data-position="header">这是放在头部的内容</h1>

<h1 data-position="main">这是放在主体的内容</h1>

<h1 data-position="footer">这是放在尾部的内容</h1>

</ChildCom>

</div>

)

}

}

class ChildCom extends React.Component {

render() {

console.log(this.props);

let headerCom, mainCom, footerCom

this.props.children.forEach((item,index) => {

if (item.props['data-position'] === 'header') {

headerCom = item

}else if (item.props['data-position'] === 'main') {

mainCom = item

}else {

footerCom = item

}

})

return (

<div>

<div className="header">

{headerCom}

</div>

<div className="main">

{mainCom}

</div>

<div className="footer">

{footerCom}

</div>

</div>

)

}

}

ReactDOM.render(

<ParentCom>

<h2>子组件1</h2>

<h2>子组件2</h2>

<h2>子组件3</h2>

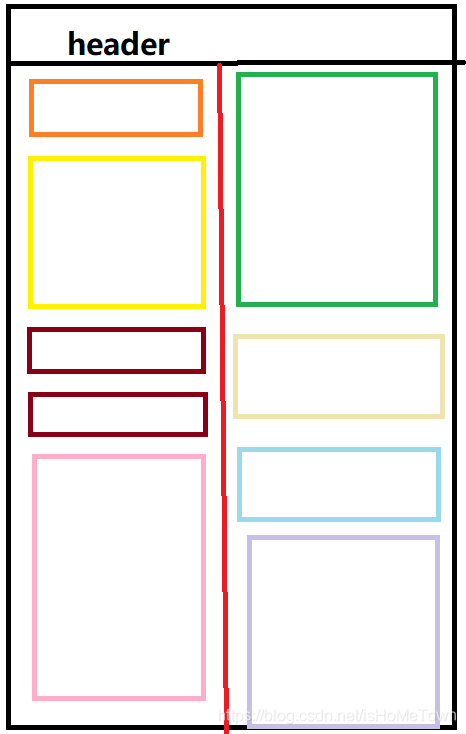
</ParentCom>,

document.getElementById('root')

)

**★★★ 简单说一下，如何在react中实现瀑布流加载？（左右两列的一个商品长列表）**

##### Answer



根据红线，将数据分为两部分，然后根据两边的高度（哪边少往那边加内容）去渲染两个盒子，然后达到一个瀑布流的效果

import React, { Component,Fragment } from 'react';

import {connect} from'react-redux'

import Axios from '\_axios@0.19.0@axios';

class Waterfall extends Component {

constructor(props) {

super(props);

this.state = {

data:[],//整体的数据

leftData:[],//左边的数据

rightData:[]//右边的数据

}

}

getHW(data){

let heightDate = [0,0];//接收累计高度的容器数组

let rightData =[]//渲染右侧盒子的数组

let leftData = []//渲染左侧盒子的数组

data.forEach(item => {

let height = item.src.replace('http://dummyimage.com/','').substr(0,7).split('x')[1]\*1;//对url地址进行一个截取，拿到高度

let minNum = Math.min.apply(null,heightDate)// 从heighetData筛选最小项

let minIndex = heightDate.indexOf(minNum);// 获取 最小项的小标 准备开始进行累加

heightDate[minIndex] = heightDate[minIndex] + height;//从 heightData 中找到最小的项后进行累加，

if(minIndex===0){//[0]加到left [1]加到 right

leftData.push(item)

}else{

rightData.push(item)

}

})

this.setState({ leftData,rightData });//重新set state

}

render() {

let {leftData,rightData} = this.state;

console.log(leftData,rightData)

return (

<Fragment>

<div className='left'>

{

leftData && leftData.map((item,index)=>{

return <img src={item.src} alt={index} key={index}/>

})

}

</div>

<div className='right'>

{

rightData && rightData.map((item,index)=>{

return <img src={item.src} alt={index} key={index}/>

})

}

</div>

</Fragment>

);

}

componentDidMount(){

Axios.get('/api/data').then(res=>{

this.props.dispatch({

type:'SET\_DATA',

data:res.data.data

})

this.getHW(this.props.data) //调用

})

}

}

export default connect(

(state)=>{

return{

data:state.data,

}

}

)(Waterfall);

**8、link和@import是什么，区别是什么**

是外部引入css的两种方式

**区别：**

**1.从属关系区别**

@import是 CSS 提供的语法规则，link是HTML提供的标签

**2.加载顺序区别**

加载页面时，link标签引入的 CSS 被同时加载；@import引入的 CSS 将在页面加载完毕后被加载。

**3.兼容性区别**

@import有兼容问题；link标签作为 HTML 元素，不存在兼容性问题。

**4.DOM可控性区别**

**5.权重区别**

link引入的样式权重大于@import引入的样式。

**9、image 标签的 alt与title**

alt是在图片不能正常加载时候显示的提示语

title属性是鼠标划上去显示的内容,title是全局属性

**10、HTML全局属性 global attribute**

**class**

为元素设置类标识，多个类名用空格分开，CSS和javascript可通过class属性获取元素

**contenteditable**

指定元素内容是否可编辑

**contextmenu**

自定义鼠标右键弹出菜单内容

**11、元素垂直居中**

1、**通过verticle-align:middle实现CSS垂直居中。**

2、**通过display:flex实现CSS垂直居中。**

**3、定位到垂直居中的位置。**

4、**通过line-height实现CSS垂直居中。**

**12、fouc**

FOUC（浏览器样式闪烁或者叫做无样式内存闪烁）

由于css引入使用了@import 或者存在多个style标签以及css文件在页面底部引入使得css文件加载在html之后导致页面闪烁、花屏 用link加载css文件，放在head标签里面

**13、BFC是什么？**

**概念：是块级格式上下文，它是一个独立的渲染区域，该区域内所有元素的布局不会影响到区域外元素的布局，这个渲染区域只对块级元素起作用。**

**布局规则：**

1， 内部的BOX会独占一行，

2， 两个Box垂直方向上的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠。

3， 计算BFC的高度时，浮动元素也参与计算。

4， BFC的区域不会与float元素重叠

**BFC的作用：**

1， 清除内部浮动，

2， 防止margin上下重叠，

3， 自适应两栏布局

**14、rgba()和opacity的透明效果有什么不同？**

共同点：都能实现透明效果

区别是：

opacity 作用于元素内的所有内容的透明度

rgba() 只作用于元素的颜色或其背景色

**15、CSS3新特性**

新增选择器 p:nth-child（n）{color: rgba（255, 0, 0, 0.75）}

弹性盒模型 display: flex;

多列布局 column-count: 5;

媒体查询 @media （max-width: 480px） {.box: {column-count: 1;}}

个性化字体 @font-face{font-family:BorderWeb;src:url（BORDERW0.eot）；}

颜色透明度 color: rgba（255, 0, 0, 0.75）；

圆角 border-radius: 5px;

渐变 background:linear-gradient（red, green, blue）；

阴影 box-shadow:3px 3px 3px rgba（0, 64, 128, 0.3）；

文字溢出 text-overflow:ellipsis;

背景效果 background-size: 100px 100px;

边框效果 border-image:url（bt\_blue.png） 0 10;

旋转 transform: rotate（20deg）；

倾斜 transform: skew（150deg, -10deg）；

位移 transform:translate（20px, 20px）；

缩放 transform: scale（。5）；

平滑过渡 transition: all .3s ease-in .1s;

**16、如何垂直居中一个浮动元素**

1、给父元素设置一个相对定位，然后把浮动元素设置margin：auto、绝对定位并且设置left:0,right:0,top:0,bottom:0;

2、给父元素设置一个相对定位，然后给这个浮动元素设置绝对定位并设置top：50%，left：50%，margin-top和margin-left为二分之一的宽和高

**17、js延迟加载的方式（有利于提高页面的加载速度）**

1、defer和async（只适用于外部脚本文件）

2、动态创建DOM方式（创建script，插入到DOM中·，加载完毕后执行callback）

3、按需异步载入js

给script标签添加了defer属性之后，会告知浏览器立即下载，但脚本会被延迟到整个页面都解析完毕之后再执行（load之后执行），脚本会按顺序执行

给script标签添加了async属性之后，不让页面等待脚本下载和执行，从而异步加载页面其他内容，（在load事件之前执行），不能保证脚本会按照顺序执行

**18、Ajax的工作原理**

Ajax的原理简单来说通过XmlHttpRequest对象来向服务器发送异步请求，从服务器获得数据，然后用javascript来操作DOM而更新页面。

XmlHttpRequest是Ajax的核心机制，它是在IE5中首先引入的，是一种支持异步请求的技术。简单的说，也就是JavaScript可以 及时向服务器提出请求和处理响应，而不阻塞用户，达到无刷新的效果。

**19、get和post请求的区别**

1、get一般用于获取数据，post一般用于提交或修改数据，

2、get使用Url传递参数，所以没有请求体，而post会把数据放在HTMLHEADER内，所以post请求有请求体。

3、相比在地址栏传输数据，post会安全一些。

4、get方式需要使用Request.QueryString来取的变量的值，而post方式通过Request.Form来获取变量的值

**20、sass，less 等 CSS 预处理器**

它们是动态的样式语言，是css预处理器，它们是一种特殊的语法而编译成css

**为什么使用它们？**

结构清晰，便于扩展。

可以轻松实现多重继承。

完全兼容 CSS 代码，可以方便地应用到老项目中。

**21、如何用css实现一个三角形？**

边框颜色：transparent是透明色

利用边框斜切实线三角形

border:30px solid transparent;颜色设置为透明

border-top-color:red; 某一个颜色不透明

div {

        width: 0px;

        height: 0px;

        border: 30px solid transparent;

        border-bottom-color: red;

    }

**22、html+css经典布局**

左列定宽，右列自适应

.left{width:100px;float:left}

.right{overflow:hidden}

**23、em,rem的区别**

em就是根据父元素来设置字体的大小

rem是相对于根元素<html>字体大小来显示

**24、什么是置换元素和非置换元素？**

置换元素：就是内联元素，但是可以直接设置宽度和高度

input,img,select,textarea

除了置换元素其它都是非置换元素

**25、清除浮动的几种方式？**

1、给父元素添加声明overflow:hidden;

2、给父元素添加height

3、在浮动元素下方添加空的div,并给该元素添加div{clear:both;}

4、万能清除法：

p:after{

content:".";

clear:both;

display:block;

height:0;

overflow:hidden;

visibility:hidden;

}

**webpack**

**1、webpack五个核心概念？**

**entry入口**

本项目应该使用哪个模块，来作为构建其内部依赖图的开始（指定打包入口文件）

**output输出**

在哪里输出它所创建的 bundles，以及如何命名这些文件，默认值为 ./dist

**loader**

loader让webpack能够去处理那些非js文件(webpack自身只理解js)

**plugins**

插件则可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量。插件接口功能极其强大，可以用来处理各种各样的任务。

**mode 模式**

通过选择 development 或 production 之中的一个，来设置 mode 参数，你可以启用相应模式下的 webpack 内置的优化

**2、webpack与grunt、gulp的不同？**

三者都是前端构建工具，grunt和gulp在早期比较流行，现在webpack相对来说比较主流，不过一些轻量化的任务还是会用gulp来处理，比如单独打包CSS文件等。

grunt和gulp是基于任务和流（Task、Stream）的。类似jQuery，找到一个（或一类）文件，对其做一系列链式操作，更新流上的数据， 整条链式操作构成了一个任务，多个任务就构成了整个web的构建流程。

webpack是基于入口的。webpack会自动地递归解析入口所需要加载的所有资源文件，然后用不同的Loader来处理不同的文件，用Plugin来扩展webpack功能。

所以总结一下：

* 从构建思路来说

gulp和grunt需要开发者将整个前端构建过程拆分成多个Task，并合理控制所有Task的调用关系

webpack需要开发者找到入口，并需要清楚对于不同的资源应该使用什么Loader做何种解析和加工

* 对于知识背景来说

gulp更像后端开发者的思路，需要对于整个流程了如指掌

webpack更倾向于前端开发者的思路

**3、与webpack类似的工具还有哪些？谈谈你为什么最终选择（或放弃）使用webpack？**

同样是基于入口的打包工具还有以下几个主流的：

* webpack
* rollup
* parcel

**从应用场景上来看：**

* webpack适用于大型复杂的前端站点构建
* rollup适用于基础库的打包，如vue、react
* parcel适用于简单的实验性项目，他可以满足低门槛的快速看到效果

**4、webpack的构建流程是什么?从读取配置到输出文件这个过程尽量说全**

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
3. 确定入口：根据配置中的 entry 找出所有的入口文件；
4. 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
5. 完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
7. 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感兴趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变 Webpack 的运行结果。

**5、如何利用webpack来优化前端性能？（提高性能和体验）**

用webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运行快速高效。

* 压缩代码。删除多余的代码、注释、简化代码的写法等等方式。可以利用webpack的UglifyJsPlugin和ParallelUglifyPlugin来压缩JS文件， 利用cssnano（css-loader?minimize）来压缩css
* 利用[CDN](https://cloud.tencent.com/product/cdn?from=10680)加速。在构建过程中，将引用的静态资源路径修改为CDN上对应的路径。可以利用webpack对于output参数和各loader的publicPath参数来修改资源路径
* 删除死代码（Tree Shaking）。将代码中永远不会走到的片段删除掉。可以通过在启动webpack时追加参数--optimize-minimize来实现
* 提取公共代码。

**6、npm打包时需要注意哪些？如何利用webpack来更好的构建？**

Npm是目前最大的 JavaScript 模块仓库，里面有来自全世界开发者上传的可复用模块。你可能只是JS模块的使用者，但是有些情况你也会去选择上传自己开发的模块。 关于NPM模块上传的方法可以去官网上进行学习，这里只讲解如何利用webpack来构建。

NPM模块需要注意以下问题：

1. 要支持CommonJS模块化规范，所以要求打包后的最后结果也遵守该规则。
2. Npm模块使用者的环境是不确定的，很有可能并不支持ES6，所以打包的最后结果应该是采用ES5编写的。并且如果ES5是经过转换的，请最好连同SourceMap一同上传。
3. Npm包大小应该是尽量小（有些仓库会限制包大小）
4. 发布的模块不能将依赖的模块也一同打包，应该让用户选择性的去自行安装。这样可以避免模块应用者再次打包时出现底层模块被重复打包的情况。
5. UI组件类的模块应该将依赖的其它资源文件，例如.css文件也需要包含在发布的模块里。

基于以上需要注意的问题，我们可以对于webpack配置做以下扩展和优化：

1. CommonJS模块化规范的解决方案： 设置output.libraryTarget='commonjs2'使输出的代码符合CommonJS2 模块化规范，以供给其它模块导入使用
2. 输出ES5代码的解决方案：使用babel-loader把 ES6 代码转换成 ES5 的代码。再通过开启devtool: 'source-map'输出SourceMap以发布调试。
3. Npm包大小尽量小的解决方案：Babel 在把 ES6 代码转换成 ES5 代码时会注入一些辅助函数，最终导致每个输出的文件中都包含这段辅助函数的代码，造成了代码的冗余。解决方法是修改.babelrc文件，为其加入transform-runtime插件
4. 不能将依赖模块打包到NPM模块中的解决方案：使用externals配置项来告诉webpack哪些模块不需要打包

**7、如何在vue项目中实现按需加载？**

Vue UI组件库的按需加载 为了快速开发前端项目，经常会引入现成的UI组件库如ElementUI、iView等，但是他们的体积和他们所提供的功能一样，是很庞大的。 而通常情况下，我们仅仅需要少量的几个组件就足够了，但是我们却将庞大的组件库打包到我们的源码中，造成了不必要的开销。

不过很多组件库已经提供了现成的解决方案，如Element出品的babel-plugin-component和AntDesign出品的babel-plugin-import 安装以上插件后，在.babelrc配置中或babel-loader的参数中进行设置，即可实现组件按需加载了。

**8、webpack的概念？**

*webpack* 是一个现代 JavaScript 应用程序的*静态模块打包器(module bundler)*。

先理解四个**核心概念**：

* 入口(entry)
* 出口(output)
* loader
* 插件(plugins)

**9、webpack的原理及配置？**

**原理**

从启动构建到输出结果一系列过程：

（1）初始化参数：解析webpack配置参数，合并shell传入和webpack.config.js文件配置的参数，形成最后的配置结果。

（2）开始编译：上一步得到的参数初始化compiler对象，注册所有配置的插件，插件监听webpack构建生命周期的事件节点，做出相应的反应，执行对象的 run 方法开始执行编译。

（3）确定入口：从配置的entry入口，开始解析文件构建AST语法树，找出依赖，递归下去。

（4）编译模块：递归中根据文件类型和loader配置，调用所有配置的loader对文件进行转换，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理。

（5）完成模块编译并输出：递归完事后，得到每个文件结果，包含每个模块以及他们之间的依赖关系，根据entry配置生成代码块chunk。

（6）输出完成：输出所有的chunk到文件系统。

**配置**

1、首先安装Node.js，有node环境

2、创建一个文件夹，在指定目录打开命令行，项目初始化 npm init -y，生成一个package.json

**10、loader的配置做到样式模块化？**

需要在我们的webpack.config.js中对css-loader进行一些额外设置

**vue**

**1、axios网络请求**

axios其实就是在ajax的基础上加了promise

**介绍：**

axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，它本身具有以下特征：

优点：

* 创建 XMLHttpRequest
* 从 node.js 发出 http 请求
* 支持 Promise API
* 拦截请求和响应
* 转换请求和响应数据
* 取消请求
* 自动转换JSON数据

**使用：**

先安装npm i -S axios，在main.js中引入，比如axios请求本地json数据，

**2、vue-router路由**

就是进行页面跳转的

**声明式（标签跳转）**

<router-link :to="index">

**编程式（ js跳转）**

router.push('index')

**3、-状态管理**

**Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。Vue的状态管理就是vuex，通俗的说就是存储一些公用的数据，提供给各个组件使用。Vuex包含五个基本对象：state、getters、mutations、actions、modules**

**在组件中通过ajax请求获得数据，把数据存储在vuex中，就要先通过dispatch来通知actions来工作，actions里面的commit在通知mutations进行工作，最后mutations把数据放在state中。**

用来读取的状态集中放在store中；改变状态的方式是提交mutations，这个是同步的；异步逻辑应该封装在action中

**应用场景：Vue多个组件之间需要共享数据或状态。**

**Vuex有几个核心概念：State、Getter、Mutation、Action、Module。**

**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**

**流程：**

新建store.js文件，然后将该store实例注入到所有子组件，接下来就可以在vue的各个子组件中通过下面的方式访问vuex对象实例。

**4、vue生命周期**

**什么是vue生命周期？**

就是Vue从创建到销毁的过程，从开始创建，初始化数据，编译模板，挂载Dom -> 渲染，更新 -> 渲染，销毁等一系列过程

**vue生命周期的作用是什么？**

它的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

**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生命周期总共有几个阶段？**

**总共分为8个阶段，分别是**

**1、beforeCreate(创建前)**

在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用

**2、created(创建后)**

在实例创建完成后被立即调用。然而，挂载阶段还没开始，$el 属性目前不可见

**3、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

**4、mounted(载入后)**

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

**6、updated(更新后)**

由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子

**7、beforeDestory(销毁前)**

实例销毁之前调用。

**5、beforeUpdate(更新前)**

数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器

**8、destroyed(销毁后)**

Vue 实例销毁后调用。

1.什么是 vue 生命周期？ 答： 就是描述Vue 实例从创建到销毁的过程，就是生命周期。从开始创建、初始化数据、 编译模板、挂载 Dom→渲染、更新→渲染、销毁等一系列过程，称之为 Vue 的生 命周期。 2.vue 生命周期的作用是什么？ 答：它的生命周期中有多个事件钩子，利用钩子函数，我们可以完成一些业务逻辑，比如发送ajax获取数据还有就是利用生命周期函数实现底部菜单的隐藏与现实，以及关闭动画等。 3.vue 生命周期总共有几个阶段？ 答：它可以总共分为 8 个阶段：创建前/后, 载入前/后,更新前/后,销毁前/销毁后。 4.第一次页面加载会触发哪几个钩子？（前四个） 答：会触发 下面这几个 beforeCreate, created, beforeMount, mounted 。 5.DOM 渲染在 哪个周期中就已经完成？ 答：DOM 渲染在 mounted 中就已经完成了。

**5、ajax**

应用在axios里面

**6、vue中组件间传值的方法？vue父子组件通信的方法有什么**

bus总线，props，vuex，$emit，children

**1、vue父子组件通信的方法有什么**

**方法一：props**

使用[props](https://cn.vuejs.org/v2/guide/components.html" \l "Prop)，父组件可以使用props向子组件传递数据。

**2、子组件向父组件通信**

使用[$children](https://cn.vuejs.org/v2/api/" \l "vm-children)可以在父组件中访问子组件。

**方法一:使用[vue事件](https://cn.vuejs.org/v2/guide/components.html" \l "%E4%BD%BF%E7%94%A8-v-on-%E7%BB%91%E5%AE%9A%E8%87%AA%E5%AE%9A%E4%B9%89%E4%BA%8B%E4%BB%B6)**

父组件向子组件传递事件方法，子组件通过$emit触发事件，回调给父组件。

**方法二： 通过修改父组件传递的props来修改父组件数据**

**方法三：使用$parent**

使用[$parent](https://cn.vuejs.org/v2/api/" \l "vm-parent)可以访问父组件的数据。

**3、非父子组件、兄弟组件之间的数据传递**

非父子组件通信，Vue官方推荐[使用一个Vue实例作为中央事件总线](https://cn.vuejs.org/v2/guide/components.html" \l "%E9%9D%9E%E7%88%B6%E5%AD%90%E7%BB%84%E4%BB%B6%E9%80%9A%E4%BF%A1) bus总线。

Vue内部有一个事件机制，可以参考[源码](https://github.com/vuejs/vue/blob/dev/src/core/instance/events.js)。

$on方法用来监听一个事件。

$emit用来触发一个事件。

**4、复杂的单页应用数据管理**

当应用足够复杂情况下，请使用[vuex](https://cn.vuejs.org/v2/guide/state-management.html)进行数据管理。

**7、VUE和REACT有什么区别？**

react整体是函数式的思想，把组件设计成纯组件，状态和逻辑通过参数传入，所以在react中，是单向数据流；

vue的思想是响应式的，也就是基于是数据可变的，通过对每一个属性建立Watcher来监听，当属性变化的时候，响应式的更新对应的虚拟dom。

**8、V-model的是什么？**

Vue的双向数据绑定是由数据劫持结合发布者订阅者实现的。 数据劫持是通过Object.defineProperty()来劫持对象数据的setter和getter操作。

**9、MVC 框架**

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

* **Model（模型）** - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
* **View（视图）** - 视图代表模型包含的数据的可视化。
* **Controller（控制器）** - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

**10、MVP框架**

* M-Model : 业务逻辑和实体模型(biz/bean)
* V-View : 布局文件(XML)和Activity
* P-Presenter : 完成View和Model的交互

**11、MVVM框架**

* M-Model : 实体模型(biz/bean)
* V-View : 布局文件(XML)
* VM-ViewModel : DataBinding所在之处，对外暴露出公共属性，View和Model的绑定器

在 MVVM架构下，View层和 Model层 之间并没有直接的联系，而是通过 ViewModel 进行交互，Model层 和 ViewModel 之间的交互是双向的， 因此 View层 数据的变化会 同步到 Model层 中，而 Model 层数据的变化也会立即反应到 View层 上。 ViewModel 通过双向数据绑定把 View 层和 Model 层连接了起来，而 View层和 Model 层之间的同步工作完全是自动的，无需人为干涉，因此作为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。

**12、对keep-alive的了解？**

keep-alive是vue内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染

**13、Vue中一句话就能回答的面试题？**

1、css只在当前组件起作用

答：在style标签中写入 scoped 即可。例如 <style scoped><style>

2、v-if 和 v-show 的区别？

答：v-if按照条件是否渲染，v-show是display的 block 或 none

3、$route 和 $router 的区别？

答：$route 是路由信息对象，包括path,params,hash,query,fullPath,matched,name等路由信息参数。

$router是路由实例对象，包括了路由的跳转方法，钩子函数等。

4、vue.js的两个核心是什么？

答：数据驱动，组件系统

5、vue几种常用的指令？

答：v-for, v-if, v-else, v-bind, v-on, v-show

6、v-on 可以绑定多个方法吗？

答：可以

7、vue中key值的作用？

答：key的作用主要是为了高效的更新虚拟 DOM

8、什么是vue的计算属性？

答：computed,适用于复杂数据处理，便于维护

**14、v-if 和 v-show 区别**

v-if按照条件是否渲染，v-show是display的block或none

**15、computed 和 watch 区别**

* 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。
* 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。
* 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。
* 另外 computerd 和watch` 还都支持对象的写法，这种方式知道的人并不多。

**16、Vue的路由实现：hash模式 和 history模式**

* hash模式：在浏览器中符号“#”，#以及#后面的字符称之为hash，用 window.location.hash 读取。特点：hash虽然在URL中，但不被包括在HTTP请求中；用来指导浏览器动作，对服务端安全无用，hash不会重加载页面。
* history模式：history采用HTML5的新特性；且提供了两个新方法： pushState()， replaceState()可以对浏览器历史记录栈进行修改，以及popState事件的监听到状态变更,刷新的时候会出现404

**前端路由的hash模式和history模式有什么区别？**

1、地址栏展示不同 hash模式地址栏有# ​ 2、history在服务器端有配置，如果不配置，一刷新就会出现404，hash模式不用配置。 ​ 3、所监听的事件也不一样 hash监听hashchange事件 history监听popstate事件 ​ 4、hash模式不会刷新页面，history模式会刷新页面

**17、vue项目中的性能优化**

1.不要在模板里面写过多表达式

2.循环调用子组件时添加key

3.频繁切换的使用v-show，不频繁切换的使用v-if

4.尽量少用float，可以用flex布局

5.按需加载，可以用require或者import()按需加载需要的组件

6.路由懒加载

**18、VUE数据双向绑定的原理**

Vue内部通过Object.defineProperty（）属性拦截的方式， 具体来说：当把一个普通的JavaScript对象传给Vue实例的data选项，Vue将遍历此对象所有的属性，使用Object.defineProperty把这些属性全部转为getter/setter(数据劫持/数据映射)。在属性被访问和修改时通知视图变化。每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新。

**19、了解一下vue3和vue2有啥区别，性能为啥能提升**

1、使用了 proxy 替代 Object.defineProperty

2、压缩包体积更小

3、目录结构有一些改变

### 1.重构响应式系统，使用Proxy替换Object.defineProperty，使用Proxy优势：

•可直接监听数组类型的数据变化

•监听的目标为对象本身，不需要像Object.defineProperty一样遍历每个属性，有一定的性能提升

•可拦截apply、ownKeys、has等13种方法，而Object.defineProperty不行

•直接实现对象属性的新增/删除

### 2.新增Composition API，更好的逻辑复用和代码组织

### 3.重构 Virtual DOM

•模板编译时的优化，将一些静态节点编译成常量

•slot优化，将slot编译为lazy函数，将slot的渲染的决定权交给子组件

•模板中内联事件的提取并重用（原本每次渲染都重新生成内联函数）

4.代码结构调整，更便于Tree shaking，使得体积更小

5.使用Typescript替换Flow

## 为什么要新增Composition API，它能解决什么问题？

Vue2.0中，随着功能的增加，组件变得越来越复杂，越来越难维护，而难以维护的根本原因是Vue的API设计迫使开发者使用watch，computed，methods选项组织代码，而不是实际的业务逻辑。

另外Vue2.0缺少一种较为简洁的低成本的机制来完成逻辑复用，虽然可以minxis完成逻辑复用，但是当mixin变多的时候，会使得难以找到对应的data、computed或者method来源于哪个mixin，使得类型推断难以进行。

所以Composition API的出现，主要是也是为了解决Option API带来的问题，第一个是代码组织问题，Compostion API可以让开发者根据业务逻辑组织自己的代码，让代码具备更好的可读性和可扩展性，也就是说当下一个开发者接触这一段不是他自己写的代码时，他可以更好的利用代码的组织反推出实际的业务逻辑，或者根据业务逻辑更好的理解代码。

第二个是实现代码的逻辑提取与复用，当然mixin也可以实现逻辑提取与复用，但是像前面所说的，多个mixin作用在同一个组件时，很难看出property是来源于哪个mixin，来源不清楚，另外，多个mixin的property存在变量命名冲突的风险。而Composition API刚好解决了这两个问题。

**20、为什么要使用路由懒加载？**

像vue这种单页面应用，如果没有应用懒加载，webpack打包后的文件将会非常大，在成进入首页需要加载的内容过多，时间过长

简单来说就是：进入首页不用一次加载过多资源造成用时过长

**21、彻底理解vue中render函数与template的区别？**

vue中的template用来创建html,有的时候需要用js来创建html，这个时候需要使用render函数

**相同之处：**

render函数跟remplate一样都是创建html模板

**不同之处：**

template逻辑简单，render适合复杂逻辑

template灵活性低，render灵活性高

template性能低，render性能高

render的优先级高于template

**22、methods、computed、watch三者的区别？**

**共同点：**

三者都是以函数为基础的

computed与watch都是以vue的依赖为基础，当所依赖的数据发生变化的时候会触发相关的函数去实现数据的变动

methods里面是用来定义函数的，需要手动才能执行，不想computed与watch“自动执行”函数

**不同点：**

computed是一个计算属性

watch类似事件监听，watch的方法默认不会执行，只有依赖属性才会执行，设计deep为true可以进入深度监听，但是性能也会变大

**watch与computed的使用场景**

watch：网络请求，模糊查询，浏览器自适应，监控路由对象

computed：购物车结算， 过滤某些商品数据

**23、如何检测路由参数的变化？**

通过属性监听来实现或者beforeRouterUpdate()

**24、什么是路由守卫？路由的钩子函数有哪些？分别说出使用场景？**

路由守卫就是路由跳转前后的做的一些验证

钩子函数：

beforeRouteEnter 使用场景：登录验证

beforeRouteUpdate 使用场景：路由更新的时候

beforeRouteLeave 使用场景：路由离开的时候

**25、什么是全局守卫？**

beforeEach：全局守卫，验证用户是否登录

**26、vue中路由传参的方式有哪几种？**

1、动态路由传参数

2、query传参

3、路由解耦

**27、说一下vue.extend的理解？**

vue.extend()返回的是一个扩展实例构造器，主要用来服务vue.component，用来生成组件

**28、简述每个生命周期具体适合的场景**

beforecreate：可以在里面加入loading事件，在加载实例的时候触发

create：初始化完成时的事件写在这里，在这结束loading事件，异步请求也适合用在这里

mounted：挂载元素，获取到DOM节点

updated：如果对数据统一处理，在这里写上相应函数

beforeDestroy：可以做一个确认停止事件的确认框

**29、前端路由和后端路由的区别？**

前端路由很重要的一点是页面不刷新，把不同路由对应的不同内容或页面交给前端来做

后端路由：通过用户请求的url导航到具体的html页面，每跳转到不同的url都是重新访问服务器，然后服务端返回页面

**30、vuex中怎么异步请求的？**

vuex中的异步方法，例如请求都要在actions中声明，通过解构出commit参数，再将请求返回的数据通过commit提交到mutations中声明的方法进行修改state中的数据

**31、什么时候使用vuex更好一些？**

1、当一个组件需要多次派发事件时

2、跨组件共享数据、跨页面共享数据

**32、你工作中使用vuex的场景？**

购物车的数据共享，登入注册

**33、vue中$router和$route的区别？**

**this.$route**：当前激活的路由的信息对象。每个对象都是局部的，可以获取当前路由的 path, name, params, query 等属性。

**this.$router**：全局的 router 实例。通过 vue 根实例中注入 router 实例，然后再注入到每个子组件，从而让整个应用都有路由功能。其中包含了很多属性和对象（比如 history 对象），任何页面也都可以调用其 push(), replace(), go() 等方法。

**34、怎么样在路由中传递参数或者数据？**

**35、vue3.0和vue2.0的区别，有哪些创新？**

1. **3.0 新加入了 TypeScript 以及 PWA 的支持**
2. **默认项目目录结构也发生了变化：**
   * 移除了配置文件目录，config 和 build 文件夹

**36、watch深度监听？**

添加deep为true

**37、global,，eventbus原理？**

**global**

JS时面向对象的语言，或者说它本身就是一个大对象，就像一个大箱子里面装着很多大盒子，每个大盒子里面都装着小盒子…那么最顶层的箱子是什么呢？这个对象在ES标准中叫做Global对象。ES标准中规定Global对象要在进入执行环境前就已创建，它是所有对象的根对象，其他对下个你都是它的属性或者属性的属性。

**eventbus**

eventbus是一款在 Android 开发中使用的**发布/订阅事件**总线框架，基于观察者模式，将事件的接收者和发送者分开，简化了组件之间的通信，使用简单、效率高、体积小

**javascipt**

**1、javascript如何实现继承？**

**构造继承法**

在子类中执行父类的构造函数。

**原型继承法**

通过改变原型链的方式来达到继承 让子类的 prototype 指向 父类的实例

**实例继承法**

**组合继承**

组合继承: 就是把 原型继承 和 借用构造函数继承 合在一起使用

其中原型继承: 为了把父类的 prototype 上的内容继承下来

其中借用构造函数继承: 为了把父类的构造函数体内的内容继承到子类的身上

**2、说一下cookie,sessionStorage和localStorage的区别？**

Cookie: 关闭浏览器就失效了，通过浏览器记录信息确认用户身份，最大4kb,这也就限制了传输的数据，请求的性能会受到影响

sessionStorage：可以自己设置储存的时间，过了时间才失效，用于本地储存一个会话数据，不是一种持久化的本地储存

localStorage用于持久化的本地储存，除非主动删除，否则永远不会过期

**3、http常见的状态码？**

200-客户端请求成功

301-永久重定向

302-暂时重定向

400-客户端请求语法有错误，不能被服务器理解

401-用户没有权限

403-服务器端拒绝提供服务

404-请求地址有误

500-服务器端错误

**4、什么是闭包(closure),为什么要用它？**

概念：函数 A 内部有一个函数 B，函数 B 可以访问到函数 A 中的变量，那么函数 B 就是闭包。

闭包就是能够读取其他函数内部变量的函数。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

例如在javascript中，只有函数内部的子函数才能读取[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F/9844788)，所以闭包可以理解成“定义在一个[函数](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0/301912)内部的函数“。

使用闭包可以形成独立的空间，延长变量的生命周期，保存中间状态值

**闭包的优缺点？**

**优点：**

1. 延长了变量的声明周期 => 一个不会被销毁的函数执行空间(始终占用内存)
2. 可以在函数外部通过闭包函数访问函数内部的变量 => 一个不会被销毁的函数执行空间(始终占用内存)

**缺点致命:**

缺点总结：我们的电脑和浏览器内存空间有限，一个闭包就占用一个内存空间

-> 因为我们的电脑还是浏览器, 内存空间是有限的

-> 有一段东西始终占用内存, 那么我最大内存空间就会被压缩

-> 如果我有 10000000 个闭包空间, 占用相当大一段内存

-> 一旦内存被占满了, 那么就挂了

**闭包的应用场景：**

定时器 回调 函数防抖

**闭包面试题（必考）？**

+ 在面试笔试中必然出现的一种题

+ 再做闭包面试题的时候

=> 注意: 你调用的是闭包函数还是原始函数

function fn(num) {

  var a = num

  return function (n) {

    a += n

    console.log(a)

  }

}

 var f = fn(10)

 f(20)    // 30 调用闭包函数

 fn(30)(40)  // 70 调用原始函数

 fn(40)(50)  // 90 调用原始函数

 f(60)    // 90 调用闭包函数

**5、什么是防抖和节流，它们的区别**

**防抖是控制次数，节流是控制频率。**

**为什么要使用防抖和节流呢？**

在前端开发的过程中，我们经常需要绑定一些持续触发的事件，而当一个函数（事件）被频繁的调用的时候，会造成浏览器卡顿的现象，这时就要用到防抖和节流。

什么是防抖和节流？

防抖：

在持续触发事件之后，一段时间没有再次触发该事件，事件函数才会执行一次。

防抖解决：每次触发事件时都取消之前的延时调用方法

节流：

在持续触发事件时，每隔指定时间触发一次事件处理函数。

节流解决：每次触发事件前都判断当前是否有等待执行的延时函数

**6、常用的数组方法：**

pop() 尾部删除 push() 尾部添加 shift() 头部删除 unshift()头部添加

array.concat() 连接数组

array.join() 将数组连接起来变成一个字符串

array.reverse() 反转数组

array.slice() 切割数组

array.splice() 截取数组

array.sort() 对数组进行排序

array.toString() 将数组转换成一个字符串

**7、节点的方法**

appendChild() 添加节点

removeChild 删除节点

insertBefore 插入节点

replaceChild 替换节点

**8、async、await 优缺点**

async 和 await 相比直接使用 Promise 来说，

优势在于处理 then 的调用链，能够更清晰准确的写出代码。

缺点在于滥用 await 可能会导致性能问题，因为 await 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成，导致代码失去了并发性

**9、Promise**

**解决回调地狱的终极解决办法是async,await**

* Promise 是 ES6 新增的语法，解决了回调地狱的问题。
* 可以把 Promise看成一个状态机。初始是 pending 状态，可以通过函数 resolve 和 reject，将状态转变为 resolved 或者 rejected 状态，状态一旦改变就不能再次变化。
* then 函数会返回一个 Promise 实例，并且该返回值是一个新的实例而不是之前的实例。因为 Promise 规范规定除了 pending 状态，其他状态是不可以改变的，如果返回的是一个相同实例的话，多个 then 调用就失去意义了。 对于 then 来说，本质上可以把它看成是 flatMap
* Promise 的特点是什么？

有三种状态：1，等待中（pedding）2，完成了（resolved）3，拒绝了（rejected）

这三种状态一经改变就永远不能更改了。

Promise 实现了链式调用，也就是说每次调用 then 之后返回的都是一个 Promise

Promise 构造函数执行和 then 函数执行有什么区别？

当我们在构造 Promise 的时候，构造函数内部的代码是立即执行的，如果在then函数中使用了return，return的值会被Promise.resolve（）包装

**9、什么是回调地狱？**

多个回调函数嵌套出现的代码书写结构叫做 回调地狱 + 代码的阅读性和维护性要低了, 代码书写的不是不对, 是不好

**10、涉及面试题：什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛？**

* 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域，Ajax 请求会失败。
* 那么是出于什么安全考虑才会引入这种机制呢？ 其实主要是用来防止 CSRF 攻击的。简单点说，CSRF 攻击是利用用户的登录态发起恶意请求。
* 也就是说，没有同源策略的情况下，A 网站可以被任意其他来源的 Ajax 访问到内容。如果你当前 A 网站还存在登录态，那么对方就可以通过 Ajax 获得你的任何信息。当然跨域并不能完全阻止 CSRF。

解决跨域问题的方式？

1、JSONP 使用简单且兼容性不错，但是只限于 get 请求。利用 src 不受同源策略的影响 => 利用 script 标签会把所有内容当作 JS 代码来执行 => 我们就绕开了浏览器同源策略的影响

2、CORS(跨域资源共享) 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。

3、服务器代理(常用) 切换到 nginx 服务器, 使用 nginx 服务器进行代理配置 -> 配置好以后, 请求的时候只要请求代理标识符就可以了

**11、谈谈js中你对call（）方法和apply（）方法的理解**

1每个函数都包含两个非继承而来的方法call（）和apply（）。call（）和apply（）都可以改变this的指向。

2相同点：call（）和apply（）的第一个参数都是this值，

3区别：call（）的第二、三参数可以是任意类型，而apply（）的第二参数必须是数组或arguments对象。

**12.递归函数，回调函数**

**什么是递归函数？**

定义：在函数体内部直接或间接的调用自己。如果不停的执行递归函数，就会造成死循环，所以递归函数必须要有一个条件判断的出口，而且必须使用return跳出。

a. 递归解决的是深度遍历，循环解决的是广度遍历

**递归使用：**

90% ~ 95% 递归能做的事情循环也能 推荐使用循环，不推荐使用递归 慎用!!!慎用!!!慎用!!!

**什么是回调函数？**

将这个函数作为参数传到另一个函数里面，当那个函数执行完之后，再执行传进去的这个函数，需要使用异步代码的时候在使用回调函数

**回调函数有什么缺点？**

容易出现回调地狱

**如何解决回调地狱问题？**

1. 不使用匿名函数
2. Promise

**13、break，continue，return**

**三者区别**

**break**

break用于完全结束一个循环，跳出循环体。

**continue**

continue的功能和break有点类似，区别是continue只是中止本次循环，接着开始下一次循环。而break则是完全中止循环。

**return**

与continue和break不同的是，return直接结束整个方法，不管这个return处于多少层循环之内。

**14、如何理解原型？如何理解原型链？**

原型：每个js对象都有一个**proto**属性，这个属性指向原型，原型也是一个对象，

每个函数的都有一个prototype（原型）属性，prototype属性是一个指针，指向的是原型对象，原型对象中的方法和属性都可以被函数实例共享。

原型链：JS每一次获取对象中的属性，都是一次查询的过程，首先在自身上进行查找，如果在自身属性中找不到，就会去原型对象中去查找，如果原型对象还是查不到就去原型对象的原型中去查找，直到查找到原型链的顶端，也就是Object原型。（Object的原型为null）

**原型链常常用在继承中**

**15、作用域链是什么，你是如何理解的·？**

作用域链的作用主要用于查找标识符，

一般情况下，当作用域需要查询变量的时候会在当前作用域中查找该值，如果没有在当前作用域中查找到该值，就会向上级作用域中查找，直到查到全局作用域，这个查询过程形成的链条就叫作用域链。

**16、import和require的区别？**

1、require是CommonJs的语法（AMD规范引入方式）import是es6的一个语法标准

2、 require是运行时加载整个模块，import具有提升效果，会提升到模块的头部（编译时执行）

**17、字符串常用方法**

concat() 用于连接两个或多个字符串。按顺序

indexOf()可返回某个指定的字符串值在字符串中首次出现的位置。

replace(regexp/substr，replacement) 用于在字符串中用一些字符替换另一些字符

split(分隔符，howmany) 用于把一个字符串分割成字符串数组。

toLowerCase()：将字符串转为小写

toUpperCase()：将字符串转为大写

slice(start，end)可提取字符串的某个部分，并以新的字符串返回被提取的部分。

包括start但是不包括end（截取字符串建议使用）

charAt(index)可返回指定位置的字符。

charCodeAt()可返回指定位置的字符的 Unicode 编码

fromCharCode() 可接受一个指定的 Unicode 值，然后返回一个字符串。

substr(start，length) 可在字符串中抽取从 start 下标开始的指定数目的字符。（不）

substring(start，stop)用于提取字符串中在start处到stop-1的位置的字符。

**18、重绘和回流**

什么是重绘？

由于几何属或样式发生改变不影响布局而需要重新渲染的过程，就叫重绘。

什么是回流？

由于布局发生改变需要重建就叫做回流

什么原因会导致回流

添加或者删除可见的DOM元素；

元素位置改变；

元素尺寸改变——边距、填充、边框、宽度和高度

页面渲染初始化；

浏览器窗口尺寸改变——resize事件发生时；

回流必将引起重绘，而重绘不一定会引起回流。

**19、谈谈你对Proxy(代理)的认知**

Proxy就是在我们访问对象前添加了一层拦截，可以过滤很多操作，而这些过滤，由咱自己来定义。可以帮助我们完成对数据的处理，对构造函数的处理，对数据的验证等。

**20、javascript的本地对象，内置对象和宿主对象有哪些？**

本地对象：（引用类型）

Object、Function、Array、String、Boolean、Number、Date、RegExp等等。

内置对象：（只有两个，并且内置对象也是本地对象）

Global 和 Math（不可以实例化）

宿主对象：（可以理解为浏览器提供的对象）

所有的DOM和BOM都是宿主对象

DOM和BOM的区别：

BOM是浏览器对象模型，是操作浏览器的行为，window是他的一个对象

DOM是文档对象模型，是操作浏览器的内容（HTML元素）document是他的一个对象

由于BOM对象的window包含了document，document又是DOM对象的根节点，所以BOM对象包含了DOM对象。

BOM包含对象：window、document、location、navigator、history

BOM常用方法：alert()、confirm()（询问框）、open()、close()、onload()、prompt()（输入框）

**21、什么是伪数组？**

概念：

具有length属性，不具有数组的pop、push等方法，但仍能用遍历数组的方法来遍历他们。（例如：函数内的arguments、通过document.getElemnetByTagName（）获取到的dom元素）

如何判断数组是否为伪数组

Array.isArray(arr) arr instanceof Array

将伪数组转为标准数组

Array.from(arr) Array.prototype.slice.call(arr)

**22、谈谈你对同步和异步的理解，并说一下他们的区别**

异步执行代码: 当代码执行到异步的时候, 先暂时不执行, 继续向后，直到所有的同步代码都执行完毕以后, 再回来执行这个异步代码

同步执行代码: 按照代码从上到下的顺序执行

我们 JS 里面大部分都是同步代码

**定时器是一个异步执行的代码**

**23、ES6新特性**

1、可以使用let和const定义变量（常量），他们都是块级作用域，不存在变量声明提升 2、模板字符串（倒引号）让我们不再使用“+”或者concat（）连接字符串 3、解构赋值（对象和数组的解构赋值） 4、for of：循环，要注意的是for of不可以循环对象 5、展开运算符（…）可以将数组或对象展开为多个元素 6、ES6箭头函数：当箭头函数的参数只有一个时可以省略小括号，当函数体只有一条语句的话，不写大括号会默认return 7、ES6类只是一个语法糖使用class来定义一个类 8、使用新的super和extends关键字来扩展类（super必须在this之前被调用） 9、新增了数组方法 forEach()、map()、filter()、reduce()、every()、some()等等 10、新增改变this指向的方法 call()、 apply()

**24、什么是EvenLoop（事件循环）**

EvenLoop：是指浏览器提供的一种解决javascript单线程运行时不阻塞的一种机制，

在浏览器中，专门有一个线程负责执行js代码，这个线程叫做主线程，他会从上到下执行js代码，在碰到异步任务的时候，会将异步任务放在异步任务队列中，会继续向下执行js代码，等到所有的同步代码都执行完毕之后，就会把存在异步任务队列中的异步任务一个一个取出来，放入代码栈中执行。

网络事件===axios

**25、设计模式**

**单例模式:**

一个构造函数一生只有一个实例, 不管你 new 多少次, 出来的都是一个实例

**观察者模式**

1、实时监控一个状态的改变, 一旦发生改变, 马上触发技能

2、被观察者

3、观察者 => 技能: 当被观察者通知你的时候你要触发的能力

4、需要依靠构造函数来实现观察者和被观察者的实现

**发布订阅模式**

也称作观察者模式，定义了对象间的一种一对多的依赖关系，当一个对象的状态发 生改变时，所有依赖于它的对象都将得到通知

取代对象之间硬编码的通知机制，一个对象不用再显式地调用另外一个对象的某个接口。

与传统的发布-订阅模式实现方式（将订阅者自身当成引用传入发布者）不同，在JS中通常使用注册回调函数的形式来订阅

**26、数组去重的方法**

<script>

       var arr = [4,2,4,5,7,2,6,2,6,6,5]

       // ES6的写法，把数组写成Set.Set是ES6新增的数据类型，默认不允许重复的

       // 但是把数组变成Set了，在使用Array,from重新转换成数组，就可以了

       var arr1 = Array.from(new Set(arr))

       console.log(arr1)

</script>

**27、对this指向的理解？**

全局的this 指向的是window

函数中的this 指向的是函数所在的对象

对象中的this 指向其本身

究其原因，是因为箭头函数的this指向的是当前的作用域，而非运行时调用函数的对象：在传统函数中，这里的this.age是指向windows的，但是当我们使用箭头函数处理时，由于箭头函数处于obj的作用域中，所以其this指向了obj而非windows。

**28、预解析是什么？**

JS代码在执行之前，会对代码进行预解析，寻找作用域中的var 和function ，然后对其进行事先声明，在从上到下执行代码。这就是一个预解析的过程。

**29、事件委托是什么?**

利用事件冒泡的原理，让自己所触发的事件，让它的父元素代替执行

**30、事件冒泡是什么？**

当一个元素接收到事件的时候 会把他接收到的事件传给自己的父级，一直到window

**31、cookie的缺点？**

1、有数量和长度的限制，不能超过4KB

2、安全性低，容易被拦截

3、不能长久保存在客户端

**32、js中的垃圾回收机制？**

在js中，如果一个对象不再被引用，那么这个对象就会被回收，

如果两个对象互相引用，而不在被第三者引用，那么这个互相引用的对象也会被回收。

**33、BOM对象有哪些，列举window对象？**

window对象，document对象，location对象，navigator对象，screen对象，history对象

**34、token的含义和使用？**

cookie存不上，选用token而不选用cookie和seesion，因为token只需要有加密口令就可以进行验证，而session只能单对单，如果你返回的session不存在于这台服务器中，将无法进行正常验证

**含义：**

1、Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

2、Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。

3、使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

**如何使用Token？**

这是本文的重点，在这里我就介绍常用的两种方式。

1、用设备号/设备mac地址作为Token（推荐）

客户端：客户端在登录的时候获取设备的设备号/mac地址，并将其作为参数传递到服务端。

服务端：服务端接收到该参数后，便用一个变量来接收同时将其作为Token保存在数据库，并将该Token设置到session中，客户端每次请求的时候都要统一拦截，并将客户端传递的token和服务器端session中的token进行对比，如果相同则放行，不同则拒绝。

分析：此刻客户端和服务器端就统一了一个唯一的标识Token，而且保证了每一个设备拥有了一个唯一的会话。该方法的缺点是客户端需要带设备号/mac地址作为参数传递，而且服务器端还需要保存；优点是客户端不需重新登录，只要登录一次以后一直可以使用，至于超时的问题是有服务器这边来处理，如何处理？若服务器的Token超时后，服务器只需将客户端传递的Token向数据库中查询，同时并赋值给变量Token，如此，Token的超时又重新计时。

2、用session值作为Token

客户端：客户端只需携带用户名和密码登陆即可。

客户端：客户端接收到用户名和密码后并判断，如果正确了就将本地获取sessionID作为Token返回给客户端，客户端以后只需带上请求数据即可。

分析：这种方式使用的好处是方便，不用存储数据，但是缺点就是当session过期后，客户端必须重新登录才能进行访问数据。

**35、说一下bind,apply,call的区别？**

1每个函数都包含两个非继承而来的方法call（）和apply（）。call（）和apply（）都可以改变this的指向。

2相同点：call（）和apply（）的第一个参数都是this值，

3区别：call（）的第二、三参数可以是任意类型，而apply（）的第二参数必须是数组或arguments对象。

bind方法跟call方法一样

**36、说一下解构赋值的规则？**

**数组的解构赋值**

=> 快速从 对象 或者 数组 中获取成员

=> 就是把 对象 或者 数组 中的成员单独得到

数组的解构赋值

=> 快速从数组中获取成员

=> 在定义变量的时候直接使用 [] 将变量包裹起来

=> 等于号 左边的顺序 和 等于号右边的顺序是一一对应的

// 用数组解构赋值的方式

   let [n1, n2, n3] = arr

   // n1 变量获取的就是 arr[0]

   // n2 变量获取的就是 arr[1]

   // n3 变量获取的就是 arr[2]

   console.log(n1, n2, n3)

**对象的解构赋值**

=> 快速从对象中获取成员

=> 直接在定义变量的时候使用 {} 包裹起来

=> 这个 {} 里面写的名字是你定义的变量名, 也是对象内部的成员名称

// 对象的解构赋值, 从对象中获取多个成员

   let { gender, age, name } = obj

   // 他就等价于一次从对象中获取三个成员, 分别定义三个名称接收

   console.log(name, age, gender)

**37、ES6对象的常用方法？**

1、for...in循环遍历对象自身的和继承的可枚举属性

2、Object.keys返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

**3、Object.getOwnPropertyNames(obj)**

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

**4、Object.getOwnPropertySymbols(obj)**

Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。

**5、Reflect.ownKeys(obj)**

Reflect.ownKeys返回一个数组，包含对象自身的（不含继承的）所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

**38、promise的三种状态？**

1、初始化状态：pending

2、当调用resolve成功的状态：pengding=>fulfilled

3、当调用reject失败的状态：pending=>rejected

**39、闭包为什么会造成内存泄露？**

因为它把外部的包含它的函数的活动对象也包含在自己的作用域链中了，会比一般的函数占用更多内存。

**40、jsonp的返回格式及原理？**

**返回的格式**

jsonp主要是为了解决跨域访问问题而产生的，其返回的数据格式一般就是一个js脚本，返回的js脚本通常是服务端动态生成的。

**原理**

ajax 请求受同源策略影响，不允许进行跨域请求，而 script 标签 src 属性中的链接却可以访问跨域的js脚本，利用这个特性，服务端不再返回JSON格式的数据，而是返回一段调用某个函数的js代码，在src中进行了调用，这样实现了跨域。

**41、jsonp的优缺点？**

**1.优点** ​ 1.1它不像XMLHttpRequest对象实现的Ajax请求那样受到同源策略的限制，JSONP可以跨越同源策略； ​ 1.2它的兼容性更好，在更加古老的浏览器中都可以运行，不需要XMLHttpRequest或ActiveX的支持 ​ 1.3在请求完毕后可以通过调用callback的方式回传结果。将回调方法的权限给了调用方。这个就相当于将controller层和view层终于分开了。我提供的jsonp服务只提供纯服务的数据，至于提供服务以 后的页面渲染和后续view操作都由调用者来自己定义就好了。如果有两个页面需要渲染同一份数据，你们只需要有不同的渲染逻辑就可以了，逻辑都可以使用同 一个jsonp服务。 ​ **2.缺点** ​ 2.1它只支持GET请求而不支持POST等其它类型的HTTP请求 ​ 2.2它只支持跨域HTTP请求这种情况，不能解决不同域的两个页面之间如何进行JavaScript调用的问题。 ​ 2.3 jsonp在调用失败的时候不会返回各种HTTP状态码。 ​ 2.4缺点是安全性。万一假如提供jsonp的服务存在页面注入漏洞，即它返回的javascript的内容被人控制的。那么结果是什么？所有调用这个 jsonp的网站都会存在漏洞。于是无法把危险控制在一个域名下…所以在使用jsonp的时候必须要保证使用的jsonp服务必须是安全可信的。

**42、如何判断一个对象是否属于某个类？**

instanceof constructor

**43、new操作符具体干了什么？**

1、创建一个空对象，并且this变量引用该对象，同时还继承了该函数的原型

2、属性和方法被加入到this引用的对象中

3、新创建的对象由this所引用，并且最后隐士的返回this

**44、eval是做什么的？**

它的功能是把对应的字符串解析成JS代码并运行，应该避免使用eval，不安全，非常消耗性能

**45、js中作用域和变量声明提升？**

1、所有申明都会被提升到作用域的最顶上，变量声明提升仅仅是声明提升，定义并不提升；函数声明会连带 定义一起被提升

2、同一个变量申明只进行一次，并且因此其他申明都会被忽略,变成赋值操作

如：var i = 0 ; var i = 1 相当于 var i ; i = 0 ; i = 1;

3、函数声明的优先级优于变量申明，且函数声明会连带定义一起被提升

**46、浏览器标签页操作**

开启一个新的标签页, 关闭当前标签页

1. window.open()

作用: 再当前浏览器开启一个新的标签页

语法: window.open(你要再新标签页显示的地址)

1. window.close()

作用: 关闭当前标签页

语法: window.close()

**47、浏览器的常见事件**

由浏览器的行为来触发的事件

1. window.onload 事件

会在浏览器所有资源加载完毕后触发

所有资源(标签, css, js, 图片, 视频, 音频, ...)

1. window.onresize 事件

会在浏览器窗口尺寸改变的时候触发

1. window.onscroll 事件

会在浏览器的滚动条滚动的时候触发

**48、prototype - 原型**

**定义(背下来):** 每一个函数都天生自带一个属性, 叫做 prototype, 它是一个对象

构造函数也是函数, 所以构造函数也天生自带这个 prototype 属性

当你想访问这个对象空间的时候, 就直接访问 函数名.prototype

**49、proto(原型属性)**

**定义(背下来):** 每一个对象天生自带一个属性, 叫做 **proto**, 指向所属构造函数的 prototype

构造函数创造出来的对象, 也是对象

所以也有 **proto** 属性

一个对象, 如果由 xxx 构造函数 new 出来的,

那么我们就说这个对象所属的构造函数就是 xxx

也就是说, new 出来的对象属于 xxx 构造函数

**50、cookie，session，token的定义与区别？**

我的理解： cookie存在浏览器，不安全； session存在服务器，可能影响性能； token身份认证安全性好，是唯一的。 若存在第三方，用token，不用session。

**51、token的原理及使用**

**登录的验证流程**

当用户请求登录的时候，如果没有问题，我们就在服务端生成一条记录，然后把这条记录的id发给客户端，客户端收到以后把id号储存在cookie里，下次这个用户在向服务器发送请求的时候可以带着这个cookie,服务器端如果可以找到相应的记录，说明用户通过了身份验证，就把请求的数据返回给客户端

**基于token的身份验证方法**

1、用户向服务器发送用户名和密码

2、服务端收到请求，验证用户名和密码

3、验证成功后，服务端会发送一个token给客户端

4、客户端收到token以后可以把它储存起来，比如放在cookie或loaclStorage

5、用户随后的每一次请求都会通过cookie，将token传回服务器

6、服务端收到请求，然后去验证客户端请求里的cookie,如果验证成功，就向客户端返回请求的数据

**jwt的原理**

服务器认证后，生成一个json对象，发送给用户

**52、块级作用域**

在ES6中凡是{}包裹的代码都是块级作用域，凡是在块级作用域中使用let,const生命的变量都有一个暂时性死去

**53、var,let,const**

var 支持变量声明与解析 不支持块级作用域 允许重复声明

let 不支持变量的声明与解析 支持块级作用域 不允许重复声明

const 不支持变量的声明与解析 支持块级作用域 不允许重复声明 声明常量，一旦确定不能修改 声明后必须赋值

**54、扩展运算符(...)**

将数组或者对象转换成参数序列，使用逗号分隔的序列

作用：

1、将数组、对象合并

2、函数剩余参数

3、替代arguments

**HTML,CSS**

**1、ES5,ES6知识**

1.不一样的变量声明：const和let

2.模板字符串

3.箭头函数（Arrow Functions）

1. 函数的参数默认值

5.Spread / Rest 操作符

6.二进制和八进制字面量

7.对象和数组解构

8.对象超类

9.for...of 和 for...in

10.ES6中的类

**2、页面渲染流程**

1、构建DOM树及CSSOM树

2、构建渲染树

##### 3、布局与绘制渲染树

浏览器构建的步骤，大概如下：

1、处理HTML标记并构建DOM树

2、处理CSS标记并构建CSSOM树

3、将DOM与CSSOM合并成一个渲染树

4、根据渲染树来布局，以计算每个节点的集合信息

5、将各个节点绘制到屏幕

## url页面渲染

1. DNS 解析
2. HTTP三次握手 -> TCP/IP连接
3. 浏览器发送请求
4. 服务器返回请求的文件 (html)
5. 浏览器渲染

### 1. DNS 解析

1. 查找缓存
   * Chrome搜索自身的DNS缓存，看有没有该域名对应的IP地址（chrome://net-internals/#dns）
   * Chrome会搜索操作系统自身的DNS缓存（浏览器没有找到缓存或者缓存已经失效）
   * 读取本地的HOST文件（操作系统的缓存没有找到）（mac: etc/host）
2. 浏览器发起一个DNS的一个系统调用（向本地主控DNS服务器，一般由宽带运营商提供的，发起的一个域名解析请求）
   * 宽带运营商服务器查看本身缓存，有没有过期。
   * 运营商服务器代替浏览器发起一个迭代DNS解析的请求（baidu.com的IP地址是多少？1、寻找根域的DNS IP地址，2、COM域的顶级域的IP地址，3、baidu.com域的IP地址）（以123.com为例，COM 是根域名，123.com是人们经常提到的顶级域名，主域名可以理解为你的网站的访问域名，[一般是www.123.com](http://xn--www-p18dp91ijd9a.123.com/" \o "http://xn--www-p18dp91ijd9a.123.com/)）
   * 运营商服务器把结果返回操作系统内核同时缓存以来
   * 操作系统内核把结果返回浏览器
   * 最终浏览器拿到了www.imooc.com对应的IP地址

### 2. HTTP三次握手 -> TCP/IP连接

1. 浏览器获得域名对应的IP地址后，发起HTTP“三次握手”
   * 浏览器会以一个随机端口，向服务器的web程序发起TCP连接请求
   * 应用层会下发数据给传输层，然后下发给网络层。网络层中的 IP 协议会确定 IP 地址，并且指示了数据传输中如何跳转路由器。然后包会再被封装到数据链路层的数据帧结构中，最后就是物理层面的传输
   * TCP链接请求通过层层的路由设备，到达服务器端以后，首先进入网卡，然后进入内核的TCP/IP协议栈，经过防火墙的过滤，最后到达服务端，建立起TCP/IP的连接
   * 数据在进入服务端之前，可能还会先经过负责负载均衡的服务器，它的作用就是将请求合理的分发到多台服务器上

### 3. 浏览器发送请求

1. TCP/IP连接建立起来以后，浏览器就可以向服务器发送HTTP请求了使用了。（例如：用HTTP的GET方法请求一个根域里的一个协议，协议可以采用HTTP 1.0的一个协议）
   * 浏览器在每次发起请求时，都会现在浏览器缓存中查找该请求的结果以及标识
   * 如果没有请求的缓存结果和缓存标识，那么就会发起HTTP请求
   * 如果该请求的缓存结果失效，只有缓存标识，则携带该资源的缓存标识，发起HTTP请求,若协商缓存生效，则返回304
   * 如果存在请求的缓存结果，且未失效，则直接返回。（强制缓存可以在network中的size查看，代表该缓存存放的位置。from memory cache内存中的缓存[图片、js] ,from disk cache硬盘中的缓存[css]）

### 4. 服务器返回请求的文件 (html)

1. 服务器端接受到了这个请求，根据路径参数，经过后端的一些处理以后，把处理后的一个结果的数据返回给浏览器。
   * 服务器会将缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器，控制强制缓存的字段分别是Expires和Cache-Control,控制协商缓存的字段分别是Last-Modified/if-Modified-Since和Etag/if-None-Match
   * 浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器中缓存中。

### 5. 浏览器渲染

1. 浏览器拿到了服务器返回的HTML页面代码，在解析和渲染这个页面时候，里面的JS/CSS、图片静态资源，他们同样也是HTTP请求，都需要经过上面的步骤。
2. 浏览器根据拿到的资源对页面进行渲染，最终把一个完整的页面呈现给了用户。
   * HTML parser：根据 HTML 构建 DOM 树。
   * CSS parser：根据 CSS 构建 CSSOM 树。
   * 如果遇到 script 标签的话，会判断是否存在 async 或者 defer ，前者会并行进行下载并执行 JS，后者会先下载文件，然后等待 HTML 解析完成后顺序执行。如果以上都没有，就会阻塞住渲染流程直到 JS 执行完毕。遇到文件下载的会去下载文件，这里如果使用 HTTP/2 协议的话会极大的提高多图的下载效率。
   * Attachment：CSSOM 树和 DOM 树构建完成后会开始生成 Render 树。
   * Layout: 布局。确定页面元素的布局、样式等等诸多方面的东西。
   * GPU painting: 像素绘制页面，将内容显示在屏幕上了。

**3、最大最小宽高与正常宽高执行优先级**

min-width 优先级高于 max-width

**4、float position display 区别，同时使用的优先级**

**5、display visibility区别**

display 隐藏但不挤占原来的空间

visibility 隐藏但是占据原来的空间

**6、标签切换相互传值（两种，不能用点击事件）**

1、全局变量 2、通过共同的父窗口

**7、display inline与block**

block 将元素转换为块级元素(还有显示元素的功能)

inline-block 将元素转换为行内块元素

**jQuery**

**1、jQuery 框架中$.ajax()的常用参数有哪些？**

1、url：发送请求的地址

2、type：请求方式，（默认为GET方式）

3、data：发送到服务器的数据

4、dataType：预期服务器返回的数据（xml，html，script，json，jsonp，text）

5、contentType：发送给服务器数据的编码类型

6，success：请求成功的回调函数

**2、JQuery的源码看过没，简单概况一下它的实现原理**

我了解过jq的源码，但是没有花大把时间来研究，只是略微看了一下 1、 简单来说，jq的原理就是对原生函数的封装，然后链式调用。就是把函数都挂载在jquery对象的prototype下，然后每个函数结尾都会return this（jq对象本身），这样就可以一直在后面调用函数。 2、 还有就它的无new化，当我们传入一个选择器时，它在内部会返回new jQuery.fn.init（）,

**小程序**

**1. 请谈谈微信小程序作用？**

* project.config.json 项目配置文件，用得最多的就是配置是否开启https校验；
* App.js 设置一些全局的基础数据等；
* App.json 底部tab, 标题栏和路由等设置；
* App.wxss 公共样式，引入iconfont等；
* pages 里面包含一个个具体的页面；
* index.json (配置当前页面标题和引入组件等)；
* index.wxml (页面结构)；
* index.wxss (页面样式表)；
* index.js (页面的逻辑，请求和数据处理等)；

**2. 简单描述下微信小程序的相关文件类型？**

1. wxml 模板文件，是框架设计的一套标签语言，结合基础组件、事件系统、可以构建出页面的结构
2. wxss 样式文件，是一套样式语言，用于描述WXML的组件样式
3. js 脚本逻辑文件，逻辑处理网络请求
4. json 配置文件，小程序设置，如页面注册，页面标题及tabBar
5. app.json 整个小程序的全局配置，包括：
   * pages:[所有页面路径]
   * 网络设置（网络超时时间）
   * 界面表现（页面注册）
   * window:{背景色、导航样式、默认标题}
   * 底部tab等
6. app.js 监听并处理小程序的生命周期函数、声明全局变量
7. app.wxss 全局配置的样式文件

**3. 请谈谈wxml与标准的html的异同？**

* 都是用来描述页面的结构；
* 都由标签、属性等构成；
* 标签名字不一样，且小程序标签更少，单一标签更多；
* 多了一些 wx:if 这样的属性以及 {{ }} 这样的表达式
* WXML仅能在微信小程序开发者工具中预览，而HTML可以在浏览器内预览
* 组件封装不同， WXML对组件进行了重新封装，
* 小程序运行在JS Core内，没有DOM树和window对象，小程序中无法使用window对象和document对象。

**4. 请谈谈WXSS和CSS的异同？**

* 都是用来描述页面的样子；
* WXSS 具有 CSS 大部分的特性，也做了一些扩充和修改；
* WXSS新增了尺寸单位，WXSS 在底层支持新的尺寸单位 rpx；
* WXSS 仅支持部分 CSS 选择器；
* WXSS 提供全局样式与局部样式
* WXSS 不支持window和dom 文档流

**5. 你是怎么封装微信小程序的数据请求的？**

1. 将所有的接口放在统一的js文件中并导出
2. 在app.js中创建封装请求数据的方法
3. 在子页面中调用封装的请求数据

或

1. 在根目录下创建utils目录及api.js文件和apiConfig.js文件；
2. 在appConfig.js封装基础的get\post\put\upload等请求方法，设置请求体，带上token和异常处理等；
3. 在api.js中引入apiConfig.js封装好的请求方法，根据页面数据请求的urls，设置对应的方法并导出；
4. 在具体页面导入；

**6. 小程序页面间有哪些传递数据的方法？**

1. 给html元素添加data-\*属性来传递值，然后通过e.currentTarget.dataset或onload的param参数获取。注：data-名称不能有大写字母、不可以存放对象
2. 设置id的方法标识来传值，通过e.currentTarget.id获取设置的id的值，然后通过设置全局对象的方式来传递数值
3. 在navigator中添加参数数值

或

1. 使用全局变量实现数据传递
2. 页面跳转或重定向时，使用url带参数传递数据
3. 使用组件模板template传递参数
4. 使用缓存传递参数
5. 使用数据库传递参数

**7. 请谈谈小程序的双向绑定和vue的异同？**

大体相同，但小程序直接this.data的属性是不可以同步到视图的，必须调用this.setData()方法！

​

1.双向绑定：vue默认支持双向绑定，微信小程序需要借助data。

2.取值：vue中，通过this.reason取值。小程序中，通过this.data.reason取值。

3.定义方法：小程序使用 在app.js中定义即可，vue的方法通过写在method中进行定义。

​

1. 去变量的时候：

  - 小程序  wx:for = "lists"

  - Vue是  v-for = "item in lists"

2. 调用data模型（赋值）的时候：

  - 小程序：this.data.item //调用  this.setData({item:1}) //赋值

  - vue: this.item //调用  this.item =1 //赋值

小程序的双向绑定实际上并不是双向绑定 如果在小程序.js文件中改变了某个变量的值 那么页面上的值并不会跟着改变 如果想要页面上的值也跟着改变的话 需要通过setData来操作 而Vue默认就是双向绑定 只改变了某个变量的值 页面上也会跟着改变

**8. 请谈谈小程序的生命周期函数？**

* onLoad() 页面加载时触发，只会调用一次，可获取当前页面路径中的参数。
* onShow() 页面显示/切入前台时触发，一般用来发送数据请求；
* onReady() 页面初次渲染完成时触发, 只会调用一次，代表页面已可和视图层进行交互。
* onHide() 页面隐藏/切入后台时触发, 如底部 tab 切换到其他页面或小程序切入后台等。
* onUnload() 页面卸载时触发，如redirectTo或navigateBack到其他页面时。
* onPullDownRefresh() 下拉刷新的钩子函数 用户下拉刷新时会自动走到这个函数中
* onReachBottom() 上翻到底的钩子函数

**9. 简述微信小程序原理？**

* 小程序本质就是一个单页面应用，所有的页面渲染和事件处理，都在一个页面内进行，但又可以通过微信客户端调用原生的各种接口；
* 它的架构，是数据驱动的架构模式，它的UI和数据是分离的，所有的页面更新，都需要通过对数据的更改来实现；
* 它从技术讲和现有的前端开发差不多，采用JavaScript、WXML、WXSS三种技术进行开发；
* 功能可分为webview和appService两个部分；
* webview用来展现UI，appService有来处理业务逻辑、数据及接口调用；
* 两个部分在两个进程中运行，通过系统层JSBridge实现通信，实现UI的渲染、事件的处理等。

或

1. 微信小程序采用JavaScript、wxml、wxss三种技术进行开发，与现有前端开发的区别：
   * JavaScript的代码是运行在微信APP中的，因此一些h5技术的应用需要微信APP提供对应的API支持；
   * wxml微信自己基于xml语法开发的，因此在开发时只能使用微信提供的现有标签，html的标签是无法使用的；
   * wxss具有css的大部分特性，但并不是所有都支持没有详细文档（wxss的图片引入需使用外链地址，没有body，样式可直接使用import导入）。
2. 微信的架构，是数据驱动的架构模式，它的UI和数据是分离的，所有的页面更新，都需要通过对数据的更改来实现。
3. 小程序功能分为webview和APPservice，webview主要用来展示UI，appservice用来处理业务逻辑、数据及接口调用。它们在两个进程中进行，通过系统层JSBridge实现通信，实现UI的渲染、事件处理。

**10. 哪些方法来提高微信小程序的应用速度？**

1. 提高页面的加载速度
2. 用户行为预测 # 什么叫行为预测
3. 减少默认data的大小
4. 组件化方案 # 使用

**11. 分析微信小程序的优劣势？**

* 优势：
  1. 容易上手，基础组件库比较全，基本上不需要考虑兼容问题；
  2. 开发文档比较完善，开发社区比较活跃，支持插件式开发；
  3. 良好的用户体验：无需下载，通过搜索和扫一扫就可以打开，打开速度快，安卓上可以添加到桌面，与原生APP差不多；
  4. 开发成本比APP要低；
  5. 为用户提供良好的安全保障（小程序发布 严格的审查流程）
* 劣势：
  1. 限制较多，页面大小不能超过1M，不能打开超过5个层级的页面；
  2. 样式单一，部分组件已经是成型了的，样式不可修改，例如：幻灯片、导航
  3. 推广面窄，不能分享朋友圈，只能通过分享给朋友，附近小程序推广
  4. 依托于微信，无法开发后台管理功能
  5. 后台调试麻烦，因为api接口必须https请求且公网地址
  6. 真机测试，个别功能安卓和苹果表现迥异，例如安卓的定位功能加载很慢

**12. 怎么解决微信小程序的异步请求问题？**

小程序支持 ES6 语法 在返回成功的回调里面处理逻辑 Promise 异步 async/await

**13. 小程序关联微信公众号如何确定用户的唯一性？**

使用wx.getUserInfo方法 withCredentials为true时，可获取encryptedData，里面有union\_id.后端需要进行对称解密。

**14. 使用webview直接加载要注意哪些事项？**

1. 必须要在小程序后台使用管理员添加业务域名；
2. h5页面跳转至小程序的脚步必须是1.3.1以上；
3. 微信分享只可以是小程序的主名称，如要自定义分享内容，需小程序版本在1.7.1以上；
4. h5的支付不可以是微信公众号的appid，必须是小程序的appid，而且用户的openid也必须是用户和小程序的

**15. 小程序调用后台接口遇到哪些问题？**

1. 数据的大小限制，超过范围会直接导致整个小程序崩溃，除非重启小程序；
2. 小程序不可以直接渲染文章内容这类型的html文本，显示需借助插件

注：插件渲染会导致页面加载变慢，建议在后台对文章内容的html进行过滤，后台直接处理批量替换p标签div标签为view标签。然后其他的标签让插件来做。

**16. 微信小程序如何实现下拉刷新？**

用view代替scroll-view，设置onPullDownRefresh函数实现

**17. 简述五个路由的区别**

wx.navigateTo()

wx.redirectTo()

wx.switchTab()

wx.navigateBack()

wx.reLaunch()

1. 在wxml页面中：
   * 跳转新页面
   * 在当前页打开
   * 切换到首页Tab
2. 在js页面中：分为‘应用内的页面’和‘tabBar页面’；
3. 如果上述跳转遇到跳转失败或者无效的问题，请访问：wx.navigateTo/wx.redirectTo无效

wx.navigateTo()：保留当前页面，跳转到应用内的某个页面。但是不能跳到 tabbar 页面

wx.redirectTo()：关闭当前页面，跳转到应用内的某个页面。但是不允许跳转到 tabbar 页面

wx.switchTab()：跳转到 tabBar 页面，并关闭其他所有非 tabBar 页面

wx.navigateBack()关闭当前页面，返回上一页面或多级页面。可通过 getCurrentPages() 获取当前的页面栈，决定需要返回几层

wx.reLaunch()：关闭所有页面，打开到应用内的某个页面

**18. 微信小程序与H5的区别？**

1. 运行环境不同（小程序在微信运行，h5在浏览器运行）；
2. 开发成本不同（h5需要兼容不同的浏览器）；
3. 获取系统权限不同（系统级权限可以和小程序无缝衔接）；
4. 应用在生产环境的运行流畅度（h5需不断对项目优化来提高用户体验）；

**19. 小程序如何更新页面中的值 ?**

可以通过this.setData来进行改变

**20. 如何实现登录数据的持久化 ?**

可以通过wx.setStorageSync('键名', 对应的值)来进行数据持久化

**21. 微信小程序和app有什么不同之处**

微信小程序属于轻量级的app 但是限制在微信中，开发周期短，功能较少，占用空间少,

app就相反 需要占用额外内存 开发周期长

【【我理解的意思相当于flask和django的区别】】

**22. 微信小程序如何进行双向绑定**

1. 通过bind-tap点击事件 向app.js 定义的方法中获取回执
2. 设置data的值 实现双向绑定

**23. 如何自定义toobar**

1. 取消当前的toobar
2. 插入自定义toobar组件
3. app.json调用组件
4. 页面显示tootar组件

**B类问题(业务)**

**1. 你在开发过程中有什么困难点(或者使用了什么技术)**

在开发的过程中我没有太多的问题但是我的同事遇到了一个问题，怎么在序列化器中获取request的值。通过源码的研究发现self.context["reqeust"].xx 可以获取到值 大大提高了数据的粘性和开发效率 在小程序端 使用模块的引用设置url值 方便url路径的更改

**2. 会写接口吗？ 项目你负责什么？**

会写drf源码 在小程序的项目中我通过自定义方法 重构drf内部方法 实现更高的扩展性。

开始：小程序、api（主）

**3. 你之前做过小程序吗？主要包括哪些功能？**

1. 没做过小程序，这次公司需求。
2. 了解vue.js ，类似。
3. 小程序就是前端技术：html、css、js

**4. 这个小程序的具体设计逻辑是怎么样的? 有几个人开发？周期多长？在你离职时， 这个小程序项目是否还在开发？或者你离 职时，你们公司在开发什么？**

他分为两大模块，有用户动态模块，拍卖模块，用户动态模块又分为发布功能，动态展示功能以及常见的用户交互功能，拍卖模块又分为专场-拍品，由后台管理人员维护更新，连我总共有3个人开发，总共开发了三个月，我离职时，项目还在开发，正在开发xx

**5. 你说你负责支付环节，那么微信支付的 流程是否可以简单说一下？**

* 临时凭证 获取 wxid
* 生成repayid
* json，返回给小程序，小程序调支付窗口。
* 支持成功提示 + POST回调。

**6. 那好，既然流程说完了，那么，我问你， 你这个项目涉及到了rmb，它在用户点击 支付之后的逻辑关系和表关系你是怎么 理解的？ 有多少张表？表关系？**

首先，在做这个功能的时候，会先在纸上画出来其中的逻辑，然后再把后端需要的值传过去。做这个后端api时，需要拿到前端的数据，判断它是否使用优惠券了，是否使用保证金了，是否有地址了，选择什么方式支付了，余额还是微信支付，还有该用户的支付价格，传到后端之后进行数据校验，首先判断地址是否存在，然后判断订单是否合法，是否使用了优惠券，是否使用保证金，用户的支付价格和后端计算的价格是否相符，由于支付环节必须保证一次完成，所以给它加了事务，订单表的查询加了锁，通过一系列校验，然后进行支付。支付成功之后，订单表记录，保证金抵扣记录，优惠券使用记录，退保证金记录，都需要进行相应的修改

**7. 你觉得这个小程序项目的细节之处有哪些？**

* 闭包
* drf，认证组件自定义。
* drf序列化 嵌套
* 重复订单处理 & 数据库锁（InnoDB 行级锁）

**8. 如何自定义tabbar？**

创建一个compent文件夹，里面创建一个tabbar页面，取消原有的tabbar页面。需要在app.json中输入compent:ture，在自定义的pages页面中写入<tabbar selected="{{0}}"></tabbar>进行选中效果显示

**9. 你刚才提到了异步，在这个项目中你是否被异步坑过，最后又是如何解决的？**

算是坑过吧，我在做发布功能时，需要将用户选中的图片和其他信息发送上去，图片发送到桶中，其他信息需要放到数据库中，而其他信息需要包含图片的路径地址，这个路径地址需要上传到桶中返回，而异步执行会造成图片上传和其他信息发送时间不一致，造成的结果是图片路径不能保存到数据库中。

**npm**

**npm是什么**

* 想用什么插件, 就打开什么插件的官网, 点击下载
  + npm 是一个大型的 包管理器

=> 其实就是一个大型的 插件 库 的商场

=> 我们所有需要的 JS 相关的第三方, 都可以直接在 npm 来下载

**其它面试题**

**1、从浏览器输入URl到显示出页面的过程**

**2、TCP三次握手**

客服端发c起请求连接服务器端s确认，服务器端也发起连接确认客服端确认。 第一次握手：客服端发送一个请求连接，服务器端只能确认自己可以接受客服端发送的报文段

第二次握手： 服务端向客服端发送一个链接，确认客服端收到自己发送的报文段 第三次握手： 服务器端确认客服端收到了自己发送的报文段

简写：

第一次：客户端向服务端发送一个请求，服务端确认接收

第二次：服务端发送向客户端发送一个链接，确认已经接收到报文

第三次：服务端确认客户端已经接收到报文

**3、TCP四次挥手**

* 客户端 -- FIN --> 服务端， FIN—WAIT
* 服务端 -- ACK --> 客户端， CLOSE-WAIT
* 服务端 -- ACK,FIN --> 客户端， LAST-ACK
* 客户端 -- ACK --> 服务端，CLOSED

**4、首屏优化**

**什么是首屏优化？**

用户体验对web网页来说很重要，而首屏加载是用户体验中最重要的环节之一，可能没有之一。然而现在的很多前端开发者，都没能重视对首屏加载的优化。

正常来说，首屏加载时间不应该超过2s。有些成熟的网页比如京东和淘宝，虽然页面模块非常多且复杂，但通过优化，首屏加载时间却不长。

**如何优化首屏加载时间**

1. cdn分发: 通过在多台服务器部署相同的副本，当用户访问时，服务器根据用户跟哪台服务器地理距离小或者哪台服务器此时的压力小，来决定哪台服务器去响应这个请求。
2. 静态文件缓存方案
3. 利用好script标签的[async和defer](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/10808109/script-tag-async-defer)这两个属性：功能独立且不要求马上执行的js文件，可以加入async属性。如果是优先级低且没有依赖的js，可以加入defer属性。
4. 前端做一些接口缓存：前端也可以做接口缓存，缓存的位置有两个，一个是内存，即赋值给运行时的变量，另一个是localStorage。
5. 页面使用骨架屏
6. 选择先进的图片格式：使用 WebP 的图片格式来代替现有的jpeg和png，也能缩小体积。
7. 利用好http压缩

**5、什么是深浅拷贝，jquery如何实现深浅拷贝？**

这个问题通常可以通过 JSON.parse(JSON.stringify(object)) 来解决

**浅拷贝:**

把一个对象里面的每一个成员都复制一份放到新的对象里面

=> 两个对象是两个对象空间 ，互不影响

=> 通过 for in 循环来实现的

如果对象中的某一个成员的值是一个复杂数据类型的时候

=> 你浅拷贝完毕以后, 你操作这个复杂数据类型的成员, 依旧是在操作一个内容

**深拷贝:**

把一个对象里面的每一个成员都复制一份到新的对象里面

=> 当对象中的某一个成员是复杂数据类型的时候

=> 继续循环遍历这个复杂数据类型

=> 在新的对象里面也创建一个复杂数据类型继续复制进去

通过递归函数来实现深拷贝

=> 在函数内部判断, 如果你是一个对象或者数组

=> 那么就再次调用函数

**6、什么是堆和栈？**

**堆（heap）是为动态分配预留的内存空间**。

**栈是为执行线程留出的内存空间**。

数据存储

1、基本数据类型就是直接存储在 栈内存 里面的

2、复杂数据类型

数据存储在 堆内存 里面，会有一个空间地址

地址存储在 栈内存 里面的变量身上

**7、你能描述一下渐进增强和优雅降级之间的不同吗?**

渐进增强：主要是针对低版本浏览器进行页面构建，保证最基本的功能，然后再针对高级浏览器进行效果或交互功能的改进来达到更好的用户体验。

优雅降级：主要是针对高级浏览器构建完整的功能，并视图降低用户的体验。然后再考虑低版本的兼容问题。

**8. 为什么要使用模块化？模块化有哪几类，各有什么特点？**

**原因：**

提供代码的复用性，便于代码维护和项目拓展

**分类：AMD，CMD，CommonJS**

**AMD：**

异步模块定义，它是一个在浏览器端模块化开发的规范，

**优点：**

适合在浏览器环境中异步加载模块。可以并行加载多个模块。

**缺点：**

提高了开发成本，并且不能按需加载，而是必须提前加载所有的依赖。

**CMD：**

同步模块定义，它也是一个在浏览器端模块化开发的规范

优点：

可以按需加载模块，依赖就近。

**AMD和CMD的区别：**

1. 对于依赖的模块AMD是提前执行，CMD是延迟执行。
2. AMD推崇依赖前置（在定义模块的时候就要声明其依赖的模块），CMD推崇依赖就近（只有在用到某个模块的时候再去require——按需加载）。

**CommonJS:**

是服务器端模块的规范，用module.export导出模块，使用require导入

特点：CommonJS 加载模块是同步的，所以只有加载完成才能执行后面的操作。

**9、内存泄漏和内存溢出，，垃圾回收机制**

**什么是内存泄漏？**

内存泄漏是指一块被分配的内存既不能使用也不能回收，

内存泄漏严重就会导致内存溢出（程序要求的内存，超出了系统所能分配的范围）

后果：轻则影响系统性能，重则导致程序崩溃

**哪些操作会造成内存泄漏**

1、意外的全局变量（在函数内部直接使用一个未声明的变量）

2、闭包

3、 dom清空或删除时，事件没有清除

4、被遗忘的定时器或回调函数

5、 setTimeout的第一个参数使用字符串而不是函数

**垃圾回收机制（没有被引用的对象就是垃圾）**

Js引擎中有一个后台进程为垃圾回收器，他会定期的监视所有对象，一旦发现有无用的对象，就会对其进行标记，然后就把它们删除掉

**10、为什么利用多个域名来存储网站资源会更有效？**

1，节省主域名的连接数，使页面响应速度提高。

2，由于同一时刻对同一域名的请求数有一定限制，使用多个域名就可以突破浏览器的并发限制。

3，静态和动态请求分别放在不同的服务器上更加方便CDN缓存

4，避免不必要的安全问题( 上传js窃取主站cookie之类的)

**11、一个页面上有大量的图片（大型电商网站），加载很慢，你有哪些方法优化这些图片的加载，给用户更好的体验。**

我对不同的情况进行不同的处理。

1，如果一个页面图片很多，并超出了浏览器窗口的可视区域，我会采用图片懒加载。

图片**懒加载**，在页面上的未可视区域可以添加一个滚动条事件，判断图片位置与浏览器顶

端的距离与页面的距离，如果前者小于后者，优先加载。

2，如果为幻灯片、相册等，可以使用图片**预加载**技术，将当前展示图片的前一张和后一张优先下载

3， 果图片为 css 图片，比如一些小图片，logo，二维码可以整合到一起，使用**精灵图**技术。

4，如果图片过大，可以使用特殊编码的图片，**加载时会先加载一张压缩的特别厉害的缩略图**，以提高用户体验。

**12、前端项目中教你如何优化自己的代码**

1、HTML结构要清晰，尽可能使用语义化标签

2、压缩 CSS, JS 和 HTML

3、使用 CDN 和缓存提高速度

4、尽量使用轻量级框架

5、使用异步加载，延迟加载依赖

6、尽可能使用CSS3动画，相比JavaScript驱动的动画效率更高

7、使用事件委托，防止绑定不必要的大量事件

8、CSS Sprites雪碧图、icon图标，base64图片

9、尽量合并样式表和js，减少请求次数

10、将样式表放在头部,脚本放在底部

什么是CDN

CDN是构建在现有网络基础之上的智能虚拟网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。

**13、一个页面从输入 URL 到页面加载显示完成，这个过程中都发生了什么？**

过程：浏览器输入域名->浏览器查询dns是否有缓存->DNS查询到域名->TCP/IP链接（三次握手）->建立连接->浏览器发出请求->服务器响应（1.2.3.4）->浏览器会先获得响应头然后在获得响应体

1. 当发送一个 URL 请求时，浏览器都会开启一个线程来处理这个请求，同时在远程 DNS 服务器上启动一 个 DNS 查询，看这个请求的url是否存在，这能使浏览器获得请求对应的 IP 地址。
2. 浏览器与远程 Web 服务器通过 TCP 三次握手来建立一个 TCP/IP 连接。
3. 一旦 TCP/IP 连接建立，浏览器会通过该连接向远程服务器发送 HTTP 的 GET 请求。远 程服务器收到请求并找到资源并使用 HTTP 响应返回该资源。
4. 响应返回之后，收到客户端的应答之后，服务器会关闭TCP/IP连接
5. 客户端接受到响应之后就会开始解析下载响应数据

**14、前端如何进行登录身份的判断**

前台发送登录请求 后台返回 token，前台得到后台返回的 token，将其写入到 localStorage 中，后续请求中都携带 token 后台判断 token 是否过期，如果过期就对前台的请求响应过期标识或者状态码 前台得到过期标识后，清除 localStorage 中的 token，然后重定向到 login 路由

**15、如何确定一个元素得包含块是谁**

元素的位置(position)和大小（dimensions） 需要根据元素的包含块（containing block）来确定。

**16、电商项目跟其它项目有什么不同？**

电商网站重点在于支付购物环节 其他类型的网站更多在于内容上面的建设

**17、浏览器缓存和回收机制**

**浏览器缓存**

简单来说，浏览器缓存就是把一个已经请求过的Web资源（如html页面，图片，js，数据等）拷贝一份副本储存在浏览器中。

(1）减少网络带宽消耗

（2）降低服务器压力

（3）减少网络延迟，加快页面打开速度

**回收机制**

在js中，如果一个对象不再被引用，那么这个对象就会被回收，

如果两个对象互相引用，而不在被第三者引用，那么这个互相引用的对象也会被回收。

**18、简述一下你对web性能优化的方案？**

1、减少http请求 2、使用浏览器缓存 3、使用压缩 4、将脚本放在底部 5、使用外部的js和css 6、精简代码

**19、什么是ts？**

TypeScript 是 Javascript 的最正确的用法，遵循最新的 ES6、Es5 规范。TypeScript 扩展了 JavaScript 的语法。

**20、ECharts**

ECharts简单来说是互联网开发程bai序过程中，后台数据du库用以实现数据到图形的映射的zhi一个插件。

具体来说一dao个使用 JavaScript 实现的开源可视化库，可以流畅的运行在 PC 和移动设备上，兼容当前绝大部分浏览器，底层依赖轻量级的矢量图形库 ZRender，提供直观，交互丰富，可高度个性化定制的数据可视化图表。

**数据库**

**1、mongodb和MySQL的区别是什么？**

mongodb是非关系型数据库。MySQL是关系型数据库。

**一、特点不同**

1、关系型数据库：传统的关系型数据库采用表格的储存方式，数据以行和列的方式进行存储，要读取和查询都十分方便；

2、非关系型数据库：一个集群是一个分布式系统；个集群是线性可伸缩的；一个Hibari集群是高度可用；所有的更新都是持久的；所有的更新都是强一致性；所有客户端操作是无锁的。

**二、功能不同**

1、关系型数据库：关系型数据库十分强调数据的一致性，并为此降低读写性能付出了巨大的代价，虽然关系型数据库存储数据和处理数据的可靠性很不错，但一旦面对海量数据的处理的时候效率就会变得很差，特别是遇到高并发读写的时候性能就会下降得非常厉害。

2、非关系型数据库：可用于云计算应用，如Web电子邮件、社交网络服务，以及其它日常需要储存TB和PB级规模数据的服务。

**笔试题**

**1、数组去重的方法**

<script>

       var arr = [4,2,4,5,7,2,6,2,6,6,5]

       // ES6的写法，把数组写成Set.Set是ES6新增的数据类型，默认不允许重复的

       // 但是把数组变成Set了，在使用Array,from重新转换成数组，就可以了

       var arr1 = Array.from(new Set(arr))

       console.log(arr1)

</script>

**2、闭包面试题（必考）？**

+ 在面试笔试中必然出现的一种题

+ 再做闭包面试题的时候

=> 注意: 你调用的是闭包函数还是原始函数

function fn(num) {

  var a = num

  return function (n) {

    a += n

    console.log(a)

  }

}

 var f = fn(10)

 f(20)    // 30 调用闭包函数

 fn(30)(40)  // 70 调用原始函数

 fn(40)(50)  // 90 调用原始函数

 f(60)    // 90 调用闭包函数

**3、递归手写**

// 递归求4的阶乘

function fn(n) {

  // 当 数字===1 的时候，结束我们递进来的

  if(n === 1) return 1

  // 把每一个数字的阶乘转换为 数字 \* 数字-1 的阶乘

     return n \* fn(n - 1)

}

var res = fn(4)

console.log(res)

**4、冒泡排序**

口诀:

先给一个数组

双层 for 循环, 一层减 1 次

里层再减外层变量

用里层变量交换变量

var arr = [1,5,3,4,2,8,5,89,54,12]

for(var i = 0; arr.length-1; i++) {

for(var j = 0; arr.length-i; j++) {

if (arr[j] > arr[j + 1]) {

         var tmp = arr[j]

         arr[j] = arr[j + 1]

         arr[j + 1] = tmp

      }

}

}

★★★ 实践题

**经过二个月“测评项目“开发，测试，上线，”测评项目“服务于学校实施的第一站，参与测评学校的学生上午8点集中开测。市场反馈有如下情况：**

**1）打开网站出现502**

**2）登录进不去系统**

**3）提交数据一致反复**

**4）有时出现白屏现象**

**对于市场反馈情况谈谈你的认识？**

首先，502错误，意思是worker忙不过来，参与测评学校的学生上午8点集中开测，在短时间内访问人数过多导致积累了大量的请求，后台服务器忙不过来。解决方法贵公司可以看一下PHP到底在忙些啥，如果是 CPU 密集型计算（应该不会），看看CPU和内存满没满，没满就多开一些worker，满了就多加一些机子。我猜测可能是数据库响应缓慢，然后人多了数据库是否能撑住也是一个考虑，多用一些缓存吧。防流量攻击真没啥好办法，找牛逼一些的机房/CDN 吧。

第二个问题，结合502报错来看，登录不进系统的原因估计也是短时间内访问人数过多导致服务器崩了。

提交数据一直反复就相对复杂了，可能出现的使用场景也有很多。

比如说我们在提交表单的时候，有可能是网络延迟或内存不足，导致点击一次提交按钮，页面没有反应，结果强迫症就犯了，就疯狂的点提交恨不得把手机屏幕点烂来。也有可能是有些毛孩子，点完提交按钮立马点击刷新页面，或直接返回上层页面想卡bug，不为别的，哎~就是玩儿。

对于这种情况，我以前工作的做法就是设置蒙版层，不管你怎么点，只要你点提交，我就给你弹出蒙版层。其实这个问题最根本的原因是程序没有进行重复判断，导致数据库重复写入。前端后端都可以添加一个重复判断逻辑，判断后台数据库中是否已经存在当前提交的数据，避免重复添加。但是这种判断只解决了两次表单依次提交的问题，如果不同用户同时提交表单，数据就不一定会正确了。这种问题就是我们常说的并发问题，并发问题的解决方案也有很多，比如：加锁排队处理等。

最后提到解决白屏问题，我个人在白屏优化实践上尝试过SSR、预渲染、还有骨架屏等一些方案，每个方案都个有自己的优劣，需要根据实际的业务场景进行取舍。

SSR服务端渲染，这个方案可以让页面直接在服务端渲染，但是不利于前后端分离，开发的效率也比客户端渲染低，同时也加大了服务器的压力。而且由于地理位置的不同，不同用户看到的页面也是不一样的，也就是所谓的千人千面，这也为缓存造成了一定困难。

使用预渲染的话，预渲染插件会在编译阶段就将对应的路由编译好插入到app节点，这样就能在js文件解析过程中有内容展示，js解析完成后，项目使用的单页面应用框架会将app节点内的内容替换成它渲染好的内容。但是当真实访问页面的时候，真实数据可能已经和预渲染的数据有了很大的出入，会导致涉及到一些价格、金额、地理位置的地方甚至会导致用户做出一些错误的决定。

使用骨架屏的话，实现原理和预加载类似，也可以很好的完善预渲染的不足。但是也存在与业务深度耦合，页面复杂度变高的问题。当应对需求变更时，骨架屏结构成本，工具实现成本，配置成本，项目的维护成本，都比较高。

**二、项目中的挑战 ★★★**

★★★ 项目开发中有遇到什么挑战没？

1.问题描述：1）简单介绍这个项目规模、背景2）什么情况下遇到什么样的问题

2.你处理这个问题的过程及结果：1）遇到问题你如何思考；2）你如何执行的；3）处理结果如何

3.通过处理这个问题，你学到了什么或者说通过这个问题，你看到了你们什么不足，后续动作（采用什么样的方式，在以后的项目中避免再出现这类问题）

1.（使用缓存处理）

在做Vue开发移动端APP时，有个页面比较常见，左边是对所有菜谱品类的展示，右边是对对应菜谱的展示，一开始在开发的时候没注意，就直接在mounted()前面使用async,里面使用await调了两边接口后，又通过watch监听了品类索引变化，调了一遍接口。后面发现每次切换品类时，页面都有一闪而过的感觉，发现每次都调了一遍接口，这对性能消耗挺大。所以就尝试的走了Vuex做缓存处理。当时解决这个缓存问题用了一些巧妙的方法，首先缓存的数据，采用的是对象形式，不是数组。这样写起来更快，因为用品类的下标直接做缓存数据的key值，非常好写。不过后面又有点坑，就是我监听的是引用数据类型，不管是vue还是react，引用数据类型发生变化时，页面可能不会更新。后面又在mutation中对vuex中的数据更新时，做了一下深复制，就做好了缓存处理。因为我是根据判断这个对象中有没有数据去调接口的，如果存在，就不调接口。现在做了缓存没错，那如果以后后台的数据发生了变化的话，那我这里也不调接口了，后台数据就没有在页面上实现更新，所以在跳出这个页面的时候还要清下缓存。清缓存要在生命周期结束的时候清，如果碰到了动态组件把APP页面下的 Tap 栏包住的时候， Keep-alive。就不能用 destroyed 生命周期清除，要用 deactivated 生命周期去清除。这样才有始有终，完成缓存处理。

2.（解决关键词高亮问题---->原理和字符串敏感词替换一样，但是用在关键词高亮上算是一个技术亮点吧）

用户配置一堆关键词，在页面上将这些关键词高亮，也许你会觉得这有什么难度？用正则匹配一下出来高亮不就行了吗？但是，一开始，用户的词不多，我确实使用的是遍历，时间复杂度为n2。后来用户会配置100w量级的词，使用遍历就会使页面卡死崩溃。解决的方法就是：优化性能，高亮分三步，生成字典树，遍历页面文字，取出文字进行匹配。使用字典树代替遍历，整个页面100w量级的词绘制可以实现在1秒以内。

3.（封装自定义组件）

在一个小程序项目需求中，要求页面头部tab栏切换的同时，对应tab栏品类的页面也要展示出来。你可能会觉得这个需求用个taroUI组件库中的tabs标签页组件不就可以完成吗？但是这个需求想满足用户的沉浸式体验，切换页面时需要有独特丝滑的专场特效。所以当时面临的问题就是，使用的UI库的组件默认样式生硬，满足不了需求。我当时是对小程序原生swiper组件进行了二次封装，主要实现了几点自定义需求：头部Tab栏品类样式使用flex动态布局，实现品类数量可变；使用 slot 插槽来动态渲染 Tab区块中的内容，配合原生swiper组件使用定义插槽；小程序原生组件<swiper>是有默认高度的，必须手动设置其高度，这里使用wx.getSystemInfo来动态获取屏幕尺寸。自己封装组件，踩了不少坑，但从中我学习到了：使用小程序的原生组件，并修改其默认的样式；学会使用 slot 插槽，实现组件内容的差异化；学会了使用小程序原生 api 获取手机信息，用 js 改变组件样式等等

4.（遇到了难用的轮子）

在写XXX小程序项目中需要实现音频播放的功能，官方已经推荐使用新的API了，然而这个小程序官方文档文档写得并不好，很多时候我们会遇到一些需求，或者改变而这些需求文档里又写得非常模糊，这就比较头疼了。小程序官方文档更新十分频繁，坑非常多。所以很多时候，我就要去不断地从文档的字里行间猜测，并结合源码一步一步地去跟踪，去尝试解决这个音频播放问题。但有时候确实超出了我的能力范围，那么我就会把我的问题提炼成一个小demo，到知乎、segmentfault思否去问，或者提问一些同样用这个轮子的作者，最终找到了新API的使用规范。

**注意**：这个时候你就要说你最擅长的，一定是你最擅长的点，告诉他这个问题怎么解决，一定是你最懂的，而且经得起推敲的解决方案。**你要准备的不仅仅是你的项目中有哪些难点，更重要的是和这个难点相关的知识都要准备充分。**所以，**你的答案并不是这个项目中最难的点，而是一个最能让你回答好整套相关问题的难点**。

**三、最难得项目 ★★★ (同上)**

★★★ 对哪个项目印象比较深刻深刻，遇到最难的项目是啥？

**【同上】**

**1.问题描述：1）简单介绍这个项目规模、背景2）什么情况下遇到什么样的问题**

**2.你处理这个问题的过程及结果：1）遇到问题你如何思考；2）你如何执行的；3）处理结果如何**

**3.通过处理这个问题，你学到了什么或者说通过这个问题，你看到了你们什么不足，后续动作（采用什么样的方式，在以后的项目中避免再出现这类问题）**

1.（使用缓存处理）

在做Vue开发移动端APP时，有个页面比较常见，左边是对所有菜谱品类的展示，右边是对对应菜谱的展示，一开始在开发的时候没注意，就直接在mounted()前面使用async,里面使用await调了两边接口后，又通过watch监听了品类索引变化，调了一遍接口。后面发现每次切换品类时，页面都有一闪而过的感觉，发现每次都调了一遍接口，这对性能消耗挺大。所以就尝试的走了Vuex做缓存处理。当时解决这个缓存问题用了一些巧妙的方法，首先缓存的数据，采用的是对象形式，不是数组。这样写起来更快，因为用品类的下标直接做缓存数据的key值，非常好写。不过后面又有点坑，就是我监听的是引用数据类型，不管是vue还是react，引用数据类型发生变化时，页面可能不会更新。后面又在mutation中对vuex中的数据更新时，做了一下深复制，就做好了缓存处理。因为我是根据判断这个对象中有没有数据去调接口的，如果存在，就不调接口。现在做了缓存没错，那如果以后后台的数据发生了变化的话，那我这里也不调接口了，后台数据就没有在页面上实现更新，所以在跳出这个页面的时候还要清下缓存。清缓存要在生命周期结束的时候清，如果碰到了动态组件把APP页面下的 Tap 栏包住的时候， Keep-alive。就不能用 destroyed 生命周期清除，要用 deactivated 生命周期去清除。这样才有始有终，完成缓存处理。

2.（解决关键词高亮问题---->原理和字符串敏感词替换一样，但是用在关键词高亮上算是一个技术亮点吧）

用户配置一堆关键词，在页面上将这些关键词高亮，也许你会觉得这有什么难度？用正则匹配一下出来高亮不就行了吗？但是，一开始，用户的词不多，我确实使用的是遍历，时间复杂度为n2。后来用户会配置100w量级的词，使用遍历就会使页面卡死崩溃。解决的方法就是：优化性能，高亮分三步，生成字典树，遍历页面文字，取出文字进行匹配。使用字典树代替遍历，整个页面100w量级的词绘制可以实现在1秒以内。

3.（封装自定义组件）

在一个小程序项目需求中，要求页面头部tab栏切换的同时，对应tab栏品类的页面也要展示出来。你可能会觉得这个需求用个taroUI组件库中的tabs标签页组件不就可以完成吗？但是这个需求想满足用户的沉浸式体验，切换页面时需要有独特丝滑的专场特效。所以当时面临的问题就是，使用的UI库的组件默认样式生硬，满足不了需求。我当时是对小程序原生swiper组件进行了二次封装，主要实现了几点自定义需求：头部Tab栏品类样式使用flex动态布局，实现品类数量可变；使用 slot 插槽来动态渲染 Tab区块中的内容，配合原生swiper组件使用定义插槽；小程序原生组件<swiper>是有默认高度的，必须手动设置其高度，这里使用wx.getSystemInfo来动态获取屏幕尺寸。自己封装组件，踩了不少坑，但从中我学习到了：使用小程序的原生组件，并修改其默认的样式；学会使用 slot 插槽，实现组件内容的差异化；学会了使用小程序原生 api 获取手机信息，用 js 改变组件样式等等

4.（遇到了难用的轮子）

在写XXX小程序项目中需要实现音频播放的功能，官方已经推荐使用新的API了，然而这个小程序官方文档文档写得并不好，很多时候我们会遇到一些需求，或者改变而这些需求文档里又写得非常模糊，这就比较头疼了。小程序官方文档更新十分频繁，坑非常多。所以很多时候，我就要去不断地从文档的字里行间猜测，并结合源码一步一步地去跟踪，去尝试解决这个音频播放问题。但有时候确实超出了我的能力范围，那么我就会把我的问题提炼成一个小demo，到知乎、segmentfault思否去问，或者提问一些同样用这个轮子的作者，最终找到了新API的使用规范。

**注意**：这个时候你就要说你最擅长的，一定是你最擅长的点，告诉他这个问题怎么解决，一定是你最懂的，而且经得起推敲的解决方案。**你要准备的不仅仅是你的项目中有哪些难点，更重要的是和这个难点相关的知识都要准备充分。**所以，**你的答案并不是这个项目中最难的点，而是一个最能让你回答好整套相关问题的难点**。

**四、前端扮演的角色 ★★★**

★★★ 项目研发流程中作为前端开发一般扮演的啥角色？

前端开发一般扮演着一个“团队核心废物”的角色。为什么这么说，首先在项目的研发中，UI小姐姐总感觉前端开发的页面满足不了她们的设计理念，说你没品味；后端小哥哥觉得前端开发就像只会写写样式的js交互工程师，像极了一个破美工的；测试小哥哥拿着测试报告说：这锅谁他娘的来背一下。还记得当年那个请你根据手机壳的颜色，来实现APP启动的颜色的产品经理嘛？这些对前端开发的误会难道不能折射出前端是团队里最应该学会沟通的人嘛？

界面有问题需要和UI沟通,数据有问题需要和后台沟通,功能有问题需要和产品沟通,测试的时候给你提bug你还需要和测试沟通……毕竟前端是最接近用户的人,用户对一个网站,软件最直观的感受是反映到前端；交互体验更是前端项目的核心点。

和UI的沟通,在工作中我们不应该是被动的实现UI的设计,而是应该合理化的提出自己的想法,不然日后返工浪费的是双方的时间。比如通用组件的设计，每次页面的提示弹窗设计，再比如你需要做一个图表,用到了echarts,你完全可以让UI基于echarts去设计样式,而不是让她在那里自由发挥，因为你永远不知道设计师的脑子里装了多少创意,这样节省的是两个人的时间,不会出现他做好样式而你实现不了的尴尬。

和后端联调接口前，先要对业务需求了解透彻，需要哪些数据，有时候明明后台来处理某个事件很简单,后台非要你来做,这就需要我们对一个需求,一个任务的要有清晰认识了,如果对任务含糊不清,自己都没搞明白,你只能受后台摆布了.最后可能也会因为任务没有完成而备受责难了。有理有据,后台开发人员是不会说什么的,否则,后台会很不耐烦的,甚至骂你的可能都有,本身做后台比较难,尤其在查询数据,取数据,封装数据方面都比较难处理。

面对产品经理的需求，前端应该深刻理解需求，毕竟工作性质影响了一个人的思维逻辑，前端能站在一个产品经理的角度去思考每一个需求，便显得尤其重要。不放过每一个细节也很重要。产品经理在设计一个产品的时候，都是从大方向去想问题的，大方向没有错就行了，细节脱离不了大方向。这是他们想的。但是对于程序来说，却万万不能。因为一个细节的逻辑往往决定了整个大方向。

举个例子：有一个需求，用户的作品需要提交审核，经过审核才可以让所有人看到。当产品经理交这个需求给你的时候，你能察觉到什么问题了吗？这里面有几个细节：1.用户提交审核后，用户可以不可以再编辑作品；2.作品是否会多次审核；3.需不需要记录审核历史；4.用户作品是否需要有版本的控制，如要产生版本，版本又是如何产生的；5.审核通过后，用户可以不可以再修改作品，若不可以，那么是不是其他人就看不见用户作品等等。

我认为前端开发是团队里最应该学会沟通的人，一个好的前端开发模式可以推动整个项目的进步的，尽管有时可能会被误解成废物，但是这何尝不是大家公认的核心呢？

**五、项目有啥优化 ★★★**

★★★ 现在有的项目中觉得哪些项目可以继续优化，为啥没有优化？

之前用vue做了一个动态官网项目，后期客户要求seo，百度上之前搜索不到官网地址，后来在项目的入口文件index.html页面加上了，固定的meta标签，加上name名为keywords、description的meta标签。以上做了个简单的seo优化，这个项目有几个官网，但是其中只有一个官网要求seo，也就是在百度能够搜索到，当时为了应急，就写死了，但是，其它的网站也就会受到干扰了，也就是对于一个项目对应几个官网，写死的meta标签做seo是不科学的。

如果这个项目要解决seo优化，可以用服务端渲染（ssr），如果项目刚开始就考虑到seo，采用服务端渲染，那么就用服务端渲染就得了。但是一般来讲，项目做到后期才会考虑到seo的问题，这时再去搞服务端渲染，相当于重头写项目，非常耗费人力物力。

所以先只考虑在首页加入 meta 标签提供一些元数据，使用简单、具有表意性的 title 以及使用 h5 提供的具有语义化的标签（不要一堆 div），生成对 search engine 友好的 sitemap，使用合理的 html 结构（比如按标题、内容、页脚这样的顺序、或者将重要的内容放在 html 前，其他放在后）

**六、总结过项目吗？★★★**

★★★ 平时写项目总结么，一般总结哪些东西？

平时在写项目的时候，会写项目总结，常常会遇到什么技术难点后，先用自己积累的知识点去尝试解决，如果遇到不会的，我就会在网上查看文档，通过翻阅资料解决问题。在解决问题后，一般我会将我认为有坑的地方记录下来。

比如，最近我做的一个比较有特点的项目，用的是 React + taro + koa框架，完成类似听书小程序。里面有个书城页面，是自己封装了品类筛选的组件。用的是小程序 scroll-view 的原生API，这里面有好多个坑，首先他有一个API，是scrollintoView,这个API的值不能数字开头，但是后端给的数字一般都是id，数字嘛。这只能自己把后端传的数字处理成string类型。最后还有一个样式的坑，你一定要按照官方文档中给的CSS样式来，给scroll-wrap加上white-space：nowrap和宽度百分百，才能实现滚动的效果。

除了总结遇到的问题，还会总结一下自己负责模块的一些规范，比如打包技术，图片的使用，自己使用了那些UI库的组件实现了什么具体的功能。自己负责的模块中联调了哪些接口。最后总结下大概完成了哪些需求。

**七、持续学习么？★★★**

★★★ 工作中能够持续学习么？

工作中当然可以持续学习，我认为每次对自己的总结就是很好学习机会，可以是工作上的总结，生活上的总结，某一件小事的总结都可以学习。

总结自己掌握的前端知识体系，总结知识体系的过程是一个查缺补漏的过程。遇到没掌握或者了解不深的知识点，务必去了解、搞清楚，否则就不算掌握。还可以，总结自己做过的项目中有哪些难点，或者技术架构。涉及到的所有技术点都可以挖一下，看看有没有某个点是自己描述不清的。

有空闲时间，会去leetcode社区上找点算法题，可以去bestofjs社区看看社区又出了什么比较前沿的技术，框架 。还可以去Git上看看别人写的代码，或者翻翻Vue，React源码研究研究。

**八、学习动力 ★★★**

★★★ 学习的动力怎么来的，如何维持？

我对前端开发学习动力来自对解决问题的成就感，这一个过程，是让人享受的。比如刚步入前端的时候，当我刚学会怎么使用css属性，去给静态页面添加旋转，缩放，背景颜色渐变的时候，那时候满脑子都是花里花哨的效果，在练习的demo中，一些地方本来朴实无华，但我偏偏想用用动画，渐变去玩一玩。后面学到用js操作dom元素，自己玩玩计时器，页面上能玩的更花里胡哨了，就感觉非常舒服。在又学会了jq,ajax配合表单的使用，向后端传递数据时，感觉就非常到位。慢慢就觉得前端开发非常有成就感。现在前端开发，知识体系庞大，各种框架组件，组件写法百花齐放，这难道不是一种让人兴奋的事嘛？我在工作中不但可以做着让人兴奋的事，还可以拿工资，这难道不舒服嘛？

**九、未来规划 ★★★★**

★★★★ 未来会有什么样的规划？

我近期的规划首先先把咱们公司的项目做好。另外夯实我自己的技术，多学一些组件、插件，再多学一到两个框架。

这是我近期的规划，对于中长期的规划。我打算将来我研究研究Vue，react，源码。深入了解一下前端工程化和实践，我会持续关注业界的新话题和新技术，并锻炼一下自己进行项目或者业务的技术选型的能力，尝试制定前端的技术规范，制定文档。前段时间也研究过一下webpack，用webpack自己手动搭建了一个React脚手架，把以前工作写的后台管理系统重新在在自己搭的脚手架中简单的跑了一下。我觉得这让我对前端工程化思想有很大的帮助。所以未来希望自己的技术水平能再上一个台阶，也为了更好的为我们公司服务。

**十、你对加班的看法 ★★★★**

★★★★ 对于加班你是怎么看的？

现在每个行业都竞争激烈。企业要想在竞争中占据主动地位，就必须付出更多的努力。比如华为提倡的奋斗者精神。所以我认为正常的加班是可以理解的，如果是工作需要，我会义不容辞的加班。我现在单身，没有任何的家庭负担，可以全身心的投入工作，但是同时，我会提升工作效率，不至于存在加班很严重的情况。如果公司遇到了紧急的任务，我肯定是全力配合公司来完成。

**十一、学习前端的历程 ★★★**

★★★ 说下你学习前端的历程吧？

【按照个人简历上历程的来回答，不穿帮即可】

**十二、前端的未来 ★★★**

★★★ 前端未来展望？

看好，非常看好。很多人以为，前端入门门槛比较低，随便学学HTML,CSS就能只做网页，随便网上看看教程，看几本书就能学会了。这样的话，岂不是随便在家看看，就能入职前端开发工程师了？那当然不是，前端虽然入门容易，但是越往后学的越慢。很多人只是看一点点基础，随便学点东西，只会一点点皮毛就不再深入拓展。那这样的人，市场当然早就饱和了。其实每个行业都是这样，金字塔原理，顶尖的人群可能只占百分之几，而下面的百分之八十，可能都是往这个门里挤的人。这之中，能力强的挤上去了，而能力不足又不努力的，只能被拒之门外。在未来，专业的前端开发工程师才是企业真正争夺的香饽饽。而被淘汰的不是前端开发，而是淘汰技术落后和技术不精的开发者。③关于这个行业的未来刚刚也有提到过，这个行业正处于一个高速发展的阶段，短期内还看不到任何衰退的迹象。随着前端的不断发展，移动端应用、小程序、H5游戏等全新产品的出现，前端开发应用场景不断拓展。走向更专业和工程化的发展。前端还有太多的道路没有探索，没有挖掘，很难判断前端没有现在火的时候行业会怎么样。

前端工程师面向的范围很广，学习的知识也很广泛。不过，要想成为全栈工程师，还是有很长一段距离的，有很多全新知识需要学习。会不会变成全栈工程师，可能更多跟你自己的想法相关吧。不成为全栈工程师，做专一的前端开发工作，也不是不行。

**十三、前端的发展 ★★★ (同上)**

★★★ 你看好前端发展吗？

【同上】

看好，非常看好。很多人以为，前端入门门槛比较低，随便学学HTML,CSS就能只做网页，随便网上看看教程，看几本书就能学会了。这样的话，岂不是随便在家看看，就能入职前端开发工程师了？那当然不是，前端虽然入门容易，但是越往后学的越慢。很多人只是看一点点基础，随便学点东西，只会一点点皮毛就不再深入拓展。那这样的人，市场当然早就饱和了。其实每个行业都是这样，金字塔原理，顶尖的人群可能只占百分之几，而下面的百分之八十，可能都是往这个门里挤的人。这之中，能力强的挤上去了，而能力不足又不努力的，只能被拒之门外。在未来，专业的前端开发工程师才是企业真正争夺的香饽饽。而被淘汰的不是前端开发，而是淘汰技术落后和技术不精的开发者。③关于这个行业的未来刚刚也有提到过，这个行业正处于一个高速发展的阶段，短期内还看不到任何衰退的迹象。随着前端的不断发展，移动端应用、小程序、H5游戏等全新产品的出现，前端开发应用场景不断拓展。走向更专业和工程化的发展。前端还有太多的道路没有探索，没有挖掘，很难判断前端没有现在火的时候行业会怎么样。

前端工程师面向的范围很广，学习的知识也很广泛。不过，要想成为全栈工程师，还是有很长一段距离的，有很多全新知识需要学习。会不会变成全栈工程师，可能更多跟你自己的想法相关吧。不成为全栈工程师，做专一的前端开发工作，也不是不行。

**十四、登录的流程 ★★★**

★★★ 请简单绘制登录场景的业务流程图，如不熟悉登录业务，也可以选择自己之前项目的业务简单说明。

React做的后台管理系统的登录。【鉴权，不同角色的权限管理】

**十五、项目上线 ★★★**

★★★ 项目上线后，会将 index.html 给后端，在地址栏上输入 www.abc.com，当在地址后面缀上 /layout 回车后，页面会报 404，是否遇见过这个问题，又该如何去解决？

修改配置文件，把默认首页从index.html改成5ox.html就可以了。

**十六、项目流程 ★★★**

★★★ 项目中由谁定接口，公司文档如何管理，由谁负责上传代码，怎么上传代码的，项目发布都是怎么做的？

首先，接口文档不应以前端作为主导方提出，而是由服务端进行接口文档的编写工作，在编写完成后，和前端一起协商补充，这样做是最好的。前端只负责数据的展示，所以没必要由前端进行接口定义，因为服务端的逻辑处理如果完全按照前端提出的来，那还咋设计。。最重要的是沟通，并且熟悉业务逻辑。

公司文档，

项目上传由运维上传，

。。。

**十七、前端工程师区别 ★★★**

★★★ 请你说说高级前端工程师和初级以及中级有什么区别？

初级前端工程师：能熟练使用HTML、CSS、JS主要工作还是搭建静态页面。一套代码能适配PC+手机端。以外，还需要会使用一些框架之类的东西，像bootstrap、jquery之类的。会ajax了解怎么与后台交互是学习Ajax的关键点。

中级前端工程师：首先就是前端工程化有一定的了解，框架angular、vue、react 。那它和jquery有着很大区别。vue是数据控制页面渲染及状态，而jquery是DOM节点控制渲染，vue渲染页面更容易更优雅。vue能够把前端项目彻底工程化，有配置文件、可以安装第三方模块、配合webpack打包、可以实现模块化开发..等等，当然简单是它最大的优势。熟练es6 7 语法、vuex、Element\_ui （开发pc端框架）、vux（开发手机端框架）、Mint UI（开发手机端框架）、Nodejs（后端语言，js语法）。

高级前端工程师：要求编写的模块更具有可复用性，不用怎么修改，后续要加新功能时，也不需要改动原有代码，就能让项目逐渐的稳定下来。有卓越的技术方案设计和架构能力了。能从原理层面来解决问题。对前端工程化、数据可视化、Webkit、可视化搭建、跨端、移动端动画等领域极为精通。来自于深耕，基本上有什么问题、需求或者技术评审，叫上他们就行。钻研和解决问题的能力。时刻关注业界新动态，能快速研究一项新技术，了解和思考如何应用以及对已有业务的帮助。如何提升体验以及赋能。遇到问题就解决问题，不抱怨。执行力和沟通能力。跨部门、团队协调合作，做一件事情能有始有终做好并且拿到成绩，拿到之后还可以汇报展示出来。

**十八、使用 echars ★★★**

★★★ 用过echars与highchars么，你遇到哪些问题及如何解决的

在用React项目中集成过echars图标表，首先引入之后，用window来访问，如果访问成功，就说明安装成功。指的注意的是echars图表初始化实例其实是dom操作，当用React的hooks写法时，这个初始化实例的操作要放在副作用里，就是用useEffect包起来，不然的话就影响性能。有时候还要注意样式，如果不给，图表就会没有高度，就不会显示出来。一般是从后端动态数据，先做一层处理，在一般是叫 options 生成器，把后端数据处理成 echars 的规范的数据，在引入到视图展示。还有注意调数据是异步的，所以要确保拿到数据后在显示图表。如果做大量图表，我们就需要封装一个个的 option ，不然数据会非常复杂。我们调接口肯定是走状态管理，一般的做法会使用两个useEffect，一个用来初始化，一个用来触发调接口。因为图表的实例化是dom操作，所以我们还有可以用hooks中useRef这个API去优化一下。

在工作中也是用过Ant-V，不过大都是看着文档来，思路都是一样的，实在有搞不定的问题，只能面向百度编程了。

**十九、使用加密 ★★★**

★★ 你写过的项目中有没有使用过加密

没有使用过加密

**二十、包管理工具 ★★★**

★★★ 项目开发中是用什么工具来管理代码的；说一下你是用过的工具用法（git、svn）

git包管理工具，加入一个项目组，正式开始开发，一般有以下几个步骤：

1、git clone git@gitlab.com ——使用git clone克隆远程项目到本地，克隆到本地后只有一个分支master

2、git checkout -b test ——本地创建一个test分支并为测试环境代码分支并切换至test，这里根据原项目仓库的测试环境代码分支命名，有些使用dev，这里假定原项目测试环境代码分支为test，测试test分支代码跟master分支代码相同

3、git pull origin test ——拉取远程test分支代码并合并到本地test分支，经过本步操作，本地test分支就是测试环境代码了，以后开发分支的代码都要先合并到test分支推到远程测试环境经过测试后才能进行线上部署

4、git checkout master ——再切回master分支

5、git checkout -b develop-branch ——创建开发分支develop-branch并切换至develop-branch分支，这一步就是要进入开发了，一个需求一个开发分支，直到功能开发完毕

6、git add new\_file.php ——把开发过程中新建的文件添加进版本管理

7、git commit -m '备注' ——提交所有变更

8、git checkout test ——切换至测试分支

9、git merge develop-branch ——合并新开发的变更到test分支

10、git push origin test ——test推到远程分支，此时测试环境可以拉取test分支代码，然后进行测试

11、git checkout master && git merge develop-branch ——测试通过后，切换到master分支然后合并develop-branch到master分支

12、git push origin master ——master分支推到远程仓库，进行后续的部署上线

**二十一、前后端分离 ★★★**

★★★ 谈谈对前后端分离的理解。

在我看来，前后端分离给开发工作带来了很多好处，在以前，没有这个概念的时候，一种经典的设计模式MVC模式。前端开发很不盛行，几乎很多后端程序员就兼顾了前端开发工作，前后端耦合性极强，在jsp时代，前端写好的页面最后要和后端实现交互，需要程序员手动的更改代码，这是很大的一个工作量，这种事情交给前端还是后端？谁都不愿意做吧！

前后端分离过后，通过预先定义好接口规范，前端后端独立部署独立开发，后端只需要提供接口供调用即可，解耦效果是真的强，这对工作效率的提升是巨大的，对于后期的维护只需要前后端单独完成即可！

【可引导拓展至对 MVC、MVVM模式的理解】

**二十二、路由跳转 ★★★**

请你说一下我从A页面路由跳到B页面如何让它不记录路由跳转

this.$router.replace： 跳转到指定url路径，但是history栈中不会有记录，点击返回会跳转到上上个页面 (就是直接替换了当前页面)

**二十三、为什么辞职 ★★★★**

★★★★ 简单的自我介绍为什么辞职？

在上一家公司，我也想过要早些辞职，但是考虑某个未完成的重要项目、或是继任者短期内还不能胜任角色所以晚了一些。主要的原因是在上一家公司待的时间比较长了，公司的各种业务已经非常熟练了，感觉工作成了流水线般的生产工具，技术达到了一定的瓶颈，我觉得我应该跳出舒适圈去接触更多的新技术与新的业务从而扩展自己的技术广度。我觉得贵公司的技术要求与我非常符合，有一些我自己非常熟练或精通的技术，更重要的事有一些我之前只是自己业余时间研究过但一直没有几乎用于实战的技术。我相信贵公司的岗位对我虽然有一定的挑战信但是会给我带来更多的提升机会。同时我上一家公司中用的React加koa框架做的项目也有很多设计上的亮点，我也可以将上一家优秀完整的项目开发流程与管理经验，项目设计思路与贵公司分享。完成互利互惠共同成长。

**二十四、项目负责什么 ★★★**

★★★ 讲一下最近的这个项目中都负责什么

1. 这个项目的列表展示与详情页是我负责的。我对于后端传递的数据进行了怎样的处理，在异步请求中选择了一定的异步分割处理数据，拆分一次性阻塞主线程的时间，可以减少用户的等待，页面滚动时选择节流，减少无效的axios请求等等，对自己模块所负责的内容，进行梳理。

2. 这个项目的登录，注册，模块是我负责的。我对不同角色的鉴权是怎么实现的。。。

3. 这个项目的表单提交页是我负责的。。。。用了什么UI组件。。。

【问法同一、二大题类似，展开自己最拿手的回答即可】

**二十五、判断成产、打包 ★★★**

★★★ 怎么判断是开发环境 生产环境

在node中，我们有一个对象process对象，它里面包括的一些信息，env和它的一些属性，当然NODE\_ENV是我们自己加上去的自定义属性，用来区分环境变量，也就是通过这个变量来进行区别是开发环境还是生产环境；但是有个问题，不同电脑上设置的方式是不一样的，所以cross-env就来了，它可以跨平台设置环境和使用环境变量。

npm install cross-env

我们在webpack.base.conf.js文件中修改代码：

const NODE\_ENV=process.env.NODE\_ENV;

console.log(NODE\_ENV);

然后我们修改package.json文件：

//--config是可以设置我们执行哪个webpack文件，默认是执行webpack.config.js,但是我们现在修改文件名了，所以我们要设置一下

"build": "cross-env NODE\_ENV=production webpack --config webpack.config.prod.js",

"dev": "cross-env NODE\_ENV=development webpack-dev-server --config webpack.config.dev.js"

就这样，我们就实现了利用webpack来区分开发环境和生产环境，当我们用npm run dev运行的时候就是开发环境，当我们运行npm run build的时候就是构建生产环境打包；

**1、作用域：**

作用域就是它定义了变量的可访问范围，控制变量的可见性和生命周期。

作用域链的作用主要用于查找标识符

作用域链的原理和原型链很类似，如果这个变量在自己的作用域中没有，那么它会寻找父级的，直到最顶层。 注意：JS没有块级作用域，若要形成块级作用域，可通过（function（）｛｝）（）；立即执行的形式实现。

**4、什么是闭包(closure),为什么要用它？**

闭包就是能够读取其他函数内部变量的函数。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

常见的构造方法，是在一个函数内部定义另外一个函数。内部函数可以引用外层的变量；外层变量不会被垃圾回收机制回收。 注意，闭包的原理是作用域链，所以闭包访问的上级作用域中的变量是个对象，其值为运算结束后的最后一个值。

优点：避免全局变量污染。缺点：容易造成内存泄漏。

使用闭包可以形成独立的空间，延长变量的生命周期，保存中间状态值

闭包的应用场景：定时器 回调 函数防抖

**5、async、await 优缺点**

async 和 await 相比直接使用 Promise 来说，

优势在于处理 then 的调用链，能够更清晰准确的写出代码。

缺点在于滥用 await 可能会导致性能问题，因为 await 会阻塞代码，也许之后的异步代码并不依赖于前者，但仍然需要等待前者完成，导致代码失去了并发性

**6、谈谈this的理解**

1. this总是指向函数的直接调用者（而非间接调用者）
2. 如果有new关键字，this指向new出来的那个对象
3. 在事件中，this指向目标元素，特殊的是IE的attachEvent中的this总是指向全局对象window。
4. 普通函数的this：指向它的调用者，如果没有调用者则默认指向window.
5. 箭头函数的this: 指向箭头函数定义时所处的对象，而不是箭头函数使用时所在的对象，默认使用父级的this.

**7、普通函数和箭头函数的区别？**

1. 外形不同，箭头函数是【 () => {()}】，普通函数是function（）
2. 箭头函数是匿名函数，普通函数有匿名函数和具名函数
3. 箭头函数不能作为构造函数，不能使用new
4. 普通函数有构造函数，可以用new创造实例化对象
5. This指向不同，箭头函数本身没有this，但是它在声明的时候，默认使用的是富集的this，普通函数中的this总是指向调用它的那个对象，默认指向window
6. 箭头函数没有原型属性
7. 箭头函数通过call()或apply()方法调用一个函数时，只传入了一个参数，对this并没有影响。

**8、ES6的新特性**

1. 可以使用let和const定义一个变量，都是块级作用域
2. 模板字符串（倒引号）
3. 解构赋值
4. For of循环，但是他不能循环对象
5. 展开运算符（...）他可以展开数组和对象的多个元素
6. 箭头函数
7. 用class定义一个类
8. 定义一个函数function
9. 引入import
10. 新增了this的方法call()，apply()
11. 新增了数组的方法forEach()，map()，filter()，reduce()
12. Symbol用来做对象的key值

**9、原型链**

原型链就是多个对象通过 \_\_proto\_\_ 的方式连接了起来。每个 JS 对象都有 \_\_proto\_\_ 属性，对象可以直接访问 \_\_proto\_\_ ⾥⾯的属性和⽅法，这个属性指向了原型

**prototype：**

每⼀个函数都有⼀个prototype这个属性，⽽这个属性指向⼀个对象，这个对象我们叫做原型对象，对象没有prototype属性

作⽤：节约内存，扩展属性和⽅法，可以实现类之间的继承

**10、微任务和宏任务**

1. 宏任务：当前栈中执行的代码是宏任务
2. 微任务：当前宏任务执行完，在下一个宏任务开始之前需要执行的任务，就是微任务，又被称为回调事件。
3. 运行：从宏任务开始，先执行一次宏任务，再询问一次微任务，微任务队列有任务就清空微任务队列，循环往复，直到无宏任务和微任务时事件轮询结束。轮流询问宏任务队列和微任务队列。
4. **微任务**：promise nextTick
5. **宏任务：原生的js就是宏任务，如**setTimeout setInterval
6. JS执行顺序：先执行宏任务列，微任务队列

**11、为什么使用key?**

需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点。作用主要是为了高效的更新虚拟DOM。

**12、axios及安装?**

请求后台资源的模块。npm i axios -S装好，

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。

**13、$nextTick的使用**

当你修改了data的值然后马上获取这个dom元素的值，是不能获取到更新后的值，

你需要使用$nextTick这个回调，让修改后的data值渲染更新到dom元素之后在获取，才能成功。

**14、并行和并发的区别**

并发是指一个处理器同时处理多个任务。

并行是指多个处理器或者是多核的处理器同时处理多个不同的任务。

并发是逻辑上的同时发生（simultaneous），而并行是物理上的同时发生。

来个比喻：并发是一个人同时吃三个馒头，而并行是三个人同时吃三个馒头。

**15、new操作符具体干了什么呢?**

1. 创建一个空对象，并且 this 变量引用该对象，同时还继承了该函数的原型。
2. 属性和方法被加入到 this 引用的对象中。
3. 新创建的对象由 this 所引用，并且最后隐式的返回 this 。

**16、为什么操作dom慢**

JS引擎和和渲染引擎的模块化设计，使得它们可以独立优化，运行速度更快，但是这种设计带来的后果就是DOM操作会越来越慢

1. **了解promise吗？**
2. Promise就是异步操作的同步代码，是ES6新增的一个构造函数。构造函数内部的代码是立即执行的。
3. Promise是解决异步以及回调地狱的问题（回调函数：回调函数就是一个被作为参数传递的函数）（嵌套太多导致代码维护成本增加，代码可读性降低的情况）
4. promise的基本使⽤

通过new promise创建⼀个promise对象，⾥⾯有⼀个参数，参数是⼀个回调函数，回调函数中 有2个参数，resolve，reject ：

完成了（resolve）当异步执⾏成功的时候调⽤的⽅法

拒绝了（reject）当异步失败的时候调⽤的⽅法

1. resolved是一个函数，直接调用会把本次的状态改为成功，rejected也是一个函数直接调用会讲本次promise的状态改为失败，成功后执行.then里面的函数，失败后执行.catch里面的函数；每次return出来的都是一个新的promise。原因也是因为状态不可变。

**二、优点和缺点**

优点：Promise 实现了链式调用，也就是说每次调用 then 之后返回的都是一个 Promise，并且是一个全新的 Promise，原因也是因为状态不可变。

缺点：无法取消 Promise，错误需要通过回调函数捕获。

promise过程：

（1）生成Promise实例

（2）执行一系列同步操作

（3）使用resolve函数将异步操作的结果传递出去, reject函数传递异步操作的错误

（4）用then方法分别指定Resolve状态和Reject状态的函数,then方法返回一个新的Promise实例,因此可以采用链式写法

**18、”==”和“===”的不同**

前者会自动转换类型，后者不会

**19、请说出call、apply、bind的区别**

1. bind:bind绑定完this的指向后会返回⼀个新的函数体，不会被⽴即调⽤
2. call和apply:绑定完this的指向后会⽴即调⽤
3. call与apply的区别：

call:第⼀个参数是this的指向，第⼆个以及后⾯的所有参数需要⼀个个进⾏传递

apply:第⼀个参数是this

**20、shouldComponentUpdate 是做什么的？**

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

1. **虚拟DOM和DIFF算法？**

虚拟DOm是相对于浏览器所渲染出来的真实DOM，将DOM抽象为虚拟DOM, 然后通过新旧虚拟DOM 这两个对象的差异(Diff算法),最终只把变化的部分重新渲染,提高渲染效率的过程;

diff 是通过JS层面的计算，返回一个patch对象，即补丁对象，在通过特定的操作解析patch对象，完成页面的重新渲染

1. **函数组件和类组件的区别**
2. 类组件有this指向，生命周期，有状态state。函数组件没有
3. 函数组件接收一个单一的 props 对象并返回了一个React元素

**类组件**

1. 无论是使用函数或是类来声明一个组件，它决不能修改它自己的 `props`。
2. 所有 React 组件都必须是纯函数，并禁止修改其自身 `props` 。
3. React是单项数据流，父组件改变了属性，那么子组件视图会更新。
4. 属性 `props` 是外界传递过来的，状态 `state` 是组件本身的，状态可以在组件中任意修改
5. 组件的属性和状态改变都会更新视图。
6. **v-if 和 v-show 区别**

v-if按照条件是否渲染，v-show是display的block或none

1. **let，const，var的区别**
2. let没有变量提升，但是他有作用域，不可以重复命名，他只有跳出作用域时才可以重复命名
3. Const他就是一个常量
4. Var他可以变量提升
5. **Vue的路由实现：hash模式 和 history模式**
6. hash模式：在浏览器中符号“#”，#以及#后面的字符称之为hash，用 window.location.hash 读取。特点：hash虽然在URL中，但不被包括在HTTP请求中；用来指导浏览器动作，对服务端安全无用，hash不会重加载页面。
7. history模式：history采用HTML5的新特性；且提供了两个新方法： pushState()， replaceState()可以对浏览器历史记录栈进行修改，以及popState事件的监听到状态变更,刷新的时候会出现404
8. **前端路由的hash模式和history模式有什么区别？**
9. 地址栏展示不同 hash模式地址栏有# ​
10. history在服务器端有配置，如果不配置，一刷新就会出现404，hash模式不用配置。 ​
11. 所监听的事件也不一样 hash监听hashchange事件 history监听popstate事件 ​
12. hash模式不会刷新页面，history模式会刷新页面
13. **computed 和 watch 区别**
14. 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。是一个多对一或者一对一。
15. 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。
16. 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。
17. 另外 computerd 和watch` 还都支持对象的写法，这种方式知道的人并不多。
18. 当一个属性发生变化时，需要执行对应的操作；一对多；
19. **你都做过哪些Vue的性能优化**
20. 尽量减少data中的数据，data中的数据都会增加getter和setter，会收集对应的watcher
21. v-if和v-for不能连用
22. 如果需要使用v-for给每项元素绑定事件时使用事件代理
23. SPA页面采用keep-alive缓存组件
24. 在更多的情况下，使用v-if替代v-show
25. key保证唯一
26. 使用路由懒加载、异步组件
27. 防抖、节流
28. 第三方模块按需导入
29. 长列表滚动到可视区域动态加载
30. 图片懒加载
31. SEO优化
32. 预渲染
33. 服务端渲染SSR
34. 打包优化
35. 压缩代码
36. Tree Shaking/Scope Hoisting
37. 使用cdn加载第三方模块
38. 多线程打包happypack
39. splitChunks抽离公共文件
40. sourceMap优化
41. 用户体验
42. 骨架屏
43. PWA
44. **Vue 等单页面应用的优缺点**
45. 单页面应用（SPA），通俗一点说就是指只有一个主页面的应用，浏览器一开始要加载所有必须的 html, js, css。所有的页面内容都包含在这个所谓的主页面中。
46. 多页面（MPA），就是指一个应用中有多个页面，页面跳转时是整页刷新。

**单页面的优点和缺点**

**优点**：

1. 用户体验好，快，内容的改变不需要重新加载整个页面，对服务器压力较小。
2. 前后端分离，比如vue项目
3. 完全的前端组件化，前端开发不再以页面为单位，更多地采用组件化的思想，代码结构和组织方式更加规范化，便于修改和调整；

**缺点：**

1. 首次加载页面的时候需要加载大量的静态资源，这个加载时间相对比较长。
2. 不利于 SEO优化，单页页面，数据在前端渲染，就意味着没有 SEO。
3. 页面导航不可用，如果一定要导航需要自行实现前进、后退。（由于是单页面不能用浏览器的前进后退功能，所以需要自己建立堆栈管理）
4. **说一下Vue的双向绑定数据的原理**

vue.js 则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。

**33、foreach和map区别**

forEach方法中是一个回调函数，然后回调函数有三个参数，第一个参数就是数组中的遍历值，第二个参数就是数组中的对应遍历的索引值，第三个参数就是数组本身，没有返回值；

Map会映射出一个新的数组，和原始数组是一样的长度。map方法和forEach方法都是遍历数组的方法，唯一不同的是map方法可以返回值，foreach的返回值是undefiend

## 深拷贝和浅拷贝以及原理和代码实现

浅拷贝：只能拷贝一层，如果对象内还有复杂数据类型，那么没效果 object.assign()；for in循环

深拷贝：不管有多少层数据，都能一次性拷贝过来，不管多少层的数据，互相之间操作都不会影响；深拷贝后的对象与原来的对象是完全隔离的，互不影响，

对一个对象的修改并不会影响另一个对象。JSON.parse(JSON.stringify( ))

深拷贝目的：同一个Array或者Object赋值给两个不同变量时，变量指向的是同一个内存地址，所以就会造成其中一个变量改变属性值，同时改变了另外一个变量的对应属性值。如果想要两个变量之间互不影响就会用到深拷贝。

**34、token的含义和使用？**

cookie存不上，选用token而不选用cookie和seesion，因为token只需要有加密口令就可以进行验证，而session只能单对单，如果你返回的session不存在于这台服务器中，将无法进行正常验证

**含义：**

1、Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

2、Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。

3、使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

**如何使用Token？**

这是本文的重点，在这里我就介绍常用的两种方式。

1、用设备号/设备mac地址作为Token（推荐）

客户端：客户端在登录的时候获取设备的设备号/mac地址，并将其作为参数传递到服务端。

服务端：服务端接收到该参数后，便用一个变量来接收同时将其作为Token保存在数据库，并将该Token设置到session中，客户端每次请求的时候都要统一拦截，并将客户端传递的token和服务器端session中的token进行对比，如果相同则放行，不同则拒绝。

分析：此刻客户端和服务器端就统一了一个唯一的标识Token，而且保证了每一个设备拥有了一个唯一的会话。该方法的缺点是客户端需要带设备号/mac地址作为参数传递，而且服务器端还需要保存；优点是客户端不需重新登录，只要登录一次以后一直可以使用，至于超时的问题是有服务器这边来处理，如何处理？若服务器的Token超时后，服务器只需将客户端传递的Token向数据库中查询，同时并赋值给变量Token，如此，Token的超时又重新计时。

2、用session值作为Token

客户端：客户端只需携带用户名和密码登陆即可。

客户端：客户端接收到用户名和密码后并判断，如果正确了就将本地获取sessionID作为Token返回给客户端，客户端以后只需带上请求数据即可。

分析：这种方式使用的好处是方便，不用存储数据，但是缺点就是当session过期后，客户端必须重新登录才能进行访问数据。

## 登录token

客户端使用用户名跟密码请求登录；

服务端收到请求，去验证用户名与密码；

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端；

客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里；

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token；

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据；

**1. ★★★ 如何再Vue的单文件组件里的样式定义全局CSS？**

在style标签上不加上scoped的属性，默认为全局css样式

**2. ★★ vue-router 3.1.0 <router-link>新增的v-slot属性怎么用？**

router-link 通过一个作用域插槽暴露底层的定制能力。这是一个更高阶的 API，主要面向库作者，但也可以为开发者提供便利，多数情况用在一个类似 NavLink 这样的自定义组件里。

在使用 v-slot API 时，需要向 router-link 传入一个单独的子元素。否则 router-link 将会把子元素包裹在一个 span 元素内。

**3. ★★★ 如何实现一个路径渲染多个组件？**

可以通过命名视图(router-view)，它容许同一界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。通过设置components即可同时渲染多个组件。

**4. ★★★ 如何实现多个路径共享一个组件？**

只需将多个路径的component字段的值设置为同一个组件即可。

**5. ★★★ 如何监测动态路由的变化**

可以通过watch方法来对$route进行监听，或者通过导航守卫的钩子函数beforeRouteUpdate来监听它的变化。

**6. ★★★ 对MVC，MVP，MVVM的理解**

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 。

MVVM 是 Model-View-ViewModel 的缩写。mvvm 是一种设计思想。

1：Model 层代表数据模型，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 是一个同步 View 和 Model 的对象。

2：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。

3：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。

**7. ★★ 说一下$root，$parent，$refs**

$root，和$parent都能访问父组件的属性和方法，区别在于如果存在多级子组件，通过parent 访问得到的是它最近一级的父组件，通过root 访问得到的是根父组件。通过在子组件标签定义 ref 属性，在父组件中可以使用$refs 访问子组件实例。

**8. ★★★★★ 你知道Vue响应式数据原理吗？Proxy 与 Object.defineProperty 优劣对比？**

**// 响应式原理**

**vue的响应式实现主要是利用了Object.defineProperty的方法里面的setter 与getter方法的观察者模式来实现。在组件初始化时会给每一个data属性注册getter和setter，然后再new 一个自己的Watcher对象，此时watcher会立即调用组件的render函数去生成虚拟DOM。在调用render的时候，就会需要用到data的属性值，此时会触发getter函数，将当前的Watcher函数注册进sub里。当data属性发生改变之后，就会遍历sub里所有的watcher对象，通知它们去重新渲染组件。**

// proxy的优势如下：

Proxy 可以直接监听对象而非属性，可以直接监听数组的变化；

Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；

Proxy 返回的是一个新对象,我们可以只操作新的对象达到目的,而 Object.defineProperty 只能遍历对象属性直接修改；

// Object.defineProperty 的优势如下:

兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题,而且无法用 polyfill(垫片)来弥补

9. ★★★★ Composition API 的出现带来哪些新的开发体验，为啥需要这个？

1：在Composition API 中时根据逻辑相关组织代码的，提高可读性和可维护性，类似于react的hook写法。

2：更好的重用逻辑代码，在Options API中通过MIxins重用逻辑代码，容易发生命名冲突且关系不清。

3：解决在生命周期函数经常包含不相关的逻辑，但又不得不把相关逻辑分离到了几个不同方法中的问题，如在mounted中设置定时器，但需要在destroyed中来清除定时器，将同一功能的代码拆分到不同的位置，造成后期代码维护的困难。

**10. ★★★ 什么情况下使用 Vuex**

如果应用够简单，最好不要使用 Vuex，一个简单的 store 模式即可，需要构建一个中大型单页应用时，使用Vuex能更好地在组件外部管理状态

**42. ★★★★ computed和watcher的区别？watch实现原理？watch有几种写法？**

计算属性computed :

1. 支持缓存，只有依赖数据发生改变，才会重新进行计算

2. 不支持异步，当computed内有异步操作时无效，无法监听数据的变化

3.computed 属性值会默认走缓存，计算属性是基于它们的响应式依赖进行缓存的，也就是基于data中声明过或者父组件传递的props中的数据通过计算得到的值

4. 如果一个属性是由其他属性计算而来的，这个属性依赖其他属性，是一个多对一或者一对一，一般用computed

5.如果computed属性属性值是函数，那么默认会走get方法；函数的返回值就是属性的属性值；在computed中的，属性都有一个get和一个set方法，当数据变化时，调用set方法。

侦听属性watch：

1. 不支持缓存，数据变，直接会触发相应的操作；

2.watch支持异步；

3.监听的函数接收两个参数，第一个参数是最新的值；第二个参数是输入之前的值；

4. 当一个属性发生变化时，需要执行对应的操作；一对多；

5. 监听数据必须是data中声明过或者父组件传递过来的props中的数据，当数据变化时，触发其他操作，函数有两个参数，

immediate：组件加载立即触发回调函数执行，

deep: 深度监听，为了发现对象内部值的变化，复杂类型的数据时使用，例如数组中的对象内容的改变，注意监听数组的变动不需要这么做。

注意：deep无法监听到数组的变动和对象的新增，参考vue数组变异,只有以响应式的方式触发才会被监听到。

watch工作原理:

watch在一开始初始化的时候，会读取一遍监听的数据的值，此时那个数据就收集到watch的watcher了然后你给watch设置的handler，watch 会放入watcher的更新函数中，当数据改变时，通知watch的watcher进行更新，于是你设置的handler就被调用了。

**11. ★★★ Vuex可以直接修改state的值吗？**

可以直接修改，但是极其不推荐，state的修改必须在mutation来修改，否则无法被devtool所监测，无法监测数据的来源，无法保存状态快照，也就无法实现时间漫游/回滚之类的操作。

**12. ★★★★ 为什么Vuex的mutation不能做异步操作**

Vuex中所有的状态更新的唯一途径都是mutation，异步操作通过 Action 来提交 mutation实现，这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。每个mutation执行完成后都会对应到一个新的状态变更，这样devtools就可以打个快照存下来，否则无法被devtools所监测。如果mutation支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难。

**13. ★★★ v-model和vuex有冲突吗？**

// 答案：

**14. ★★★ 解释单向数据流和双向数据绑定**

对于 Vue 来说，组件之间的数据传递具有单向数据流这样的特性称为单向数据流，单向数据流（Unidirectional data flow）方式使用一个上传数据流和一个下传数据流进行双向数据通信，两个数据流之间相互独立，单向数据流指只能从一个方向来修改状态。

而双向数据绑定即为当数据发生变化的时候，视图也就发生变化，当视图发生变化的时候，数据也会跟着同步变化，两个数据流之间互为影响。

**15. ★★ Vue 如何去除url中的 #**

将路由模式改为history

**16. ★★★★★ vue-router 路由实现原理**

**17. ★★★ $route 和 $router 的区别**

$route用来获取路由的信息的，它是路由信息的一个对象，里面包含路由的一些基本信息，包括name、meta、path、hash、query、params、fullPath、matched、redirectedFrom等。而$router主要是用来操作路由的，它是VueRouter的实例，包含了一些路由的跳转方法，钩子函数等

**18. ★★ 对比 jQuery，Vue 有什么不同**

jQuery 专注视图层，通过直接操作 DOM 去实现页面的一些逻辑渲染；Vue 专注于数据层，通过数据的双向绑定，最终表现在 DOM 层面，减少了 DOM 操作。Vue 使用了组件化思想，使得项目子集职责清晰，提高了开发效率，方便重复利用，便于协同开发

**19. ★★★ Vue 中怎么自定义指令**

通过directive来自定义指令，自定义指令分为全局指令和局部指令，自定义指令也有几个的钩子函数，常用的有bind和update，当 bind 和 update 时触发相同行为，而不关心其它的钩子时可以简写。一个表达式可以使用多个过滤器。过滤器之间需要用管道符“|”隔开。其执行顺序从左往右。

**20. ★★★ Vue 中怎么自定义过滤器**

通过filter来定义过滤器，过滤器分为全局和局部过滤器，过滤器的主体为一个普通的函数，来对数据进行处理，可以传递参数。当有局部和全局两个名称相同的过滤器时候，会以就近原则进行调用，即：局部过滤器优先于全局过滤器被调用。

**21. ★★★ Vue 等单页面应用的优缺点**

// 优点

1单页应用的内容的改变不需要重新加载整个页面，web应用更具响应性和更令人着迷。

2、单页应用没有页面之间的切换，就不会出现“白屏现象”,也不会出现假死并有“闪烁”现象

3、单页应用相对服务器压力小，服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍。

4、良好的前后端分离。后端不再负责模板渲染、输出页面工作，后端API通用化，即同一套后端程序代码，不用修改就可以用于Web界面、手机、平板等多种客户端。

// 缺点

1、首次加载耗时比较多。

2、SEO问题，不利于百度，360等搜索引擎收录。

3、容易造成Css命名冲突。

4、前进、后退、地址栏、书签等，都需要程序进行管理，页面的复杂度很高，需要一定的技能水平和开发成本高。

**22. ★★★ Vue-router 使用params与query传参有什么区别**

// 用法上

1：query要用path来引入，params要用name来引入，接收参数都是类似的，分别是this.$route.query.name和this.$route.params.name。

// 展示上

2：query更加类似于我们ajax中get传参，params则类似于post，说的再简单一点，前者在浏览器地址栏中显示参数，后者则不显示

3：params是路由的一部分,必须要有。query是拼接在url后面的参数，没有也没关系。

4：params、query不设置也可以传参，params不设置的时候，刷新页面或者返回参数会丢失

**23. ★★★ Vue中 keep-alive 的作用**

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。一旦使用keepalive包裹组件，此时mouted，created等钩子函数只会在第一次进入组件时调用，当再次切换回来时将不会调用。此时如果我们还想在每次切换时做一些事情，就需要用到另外的周期函数，actived和deactived，这两个钩子函数只有被keepalive包裹后才会调用。

**24. ★★★ Vue如何实现单页面应用**

通常的url 地址由以下内容构成：协议名 域名 端口号 路径 参数 哈希值，当哈希值改变，页面不会发生跳转，单页面应用就是利用了这一点，给window注册onhashchange事件，当哈希值改变时通过location.hash就能获得相应的哈希值，然后就能跳到相应的页面。

**25. ★★★★ 说明一下封装vue组件的原则和方法**

// 答案：

**26. ★★★ 说出至少4种Vue当中的指令和它的用法？**

v-if(判断是否隐藏，用来判断元素是否创建)

v-show(元素的显示隐藏，类似css中的display的block和hidden)

v-for(把数据遍历出来)

v-bind(绑定属性)

v-model(实现双向绑定)

**27. ★★★ Vuex是什么？怎么使用？描述使用它实现登录功能的流程？**

**28. ★★★ 谈谈vue生命周期函数？vue的实例选项对象？谈谈动态数据绑定的原理是什么？如何切换路由。有哪些方法传值到下一个页面，区别是什么？**

**29. ★★★ Vue-loader解释一下**

解析和转换 .vue 文件，提取出其中的逻辑代码 script、样式代码 style、以及 HTML 模版 template，再分别把它们交给对应的 Loader 去处理。

**30. ★★ 用 Vue 封装个组件，可以自定义里面有多少个input**

// 答案：

**31. ★★★ 用过插槽吗？用的是具名插槽还是匿名插槽**

用过，都使用过。插槽相当于预留了一个位置，可以将我们书写在组件内的内容放入，写一个插槽就会将组件内的内容替换一次，两次则替换两次。为了自定义插槽的位置我们可以给插槽取名，它会根据插槽名来插入内容，一一对应。

**32. ★★★ Vue-cli运行的命令行**

// 答案：

**33. ★★★ Vue 路由守卫**

vue-router 提供的导航守卫主要用来对路由的跳转进行监控，控制它的跳转或取消，路由守卫有全局的, 单个路由独享的, 或者组件级的。导航钩子有3个参数：

1、to:即将要进入的目标路由对象；

2、from:当前导航即将要离开的路由对象；

3、next ：调用该方法后，才能进入下一个钩子函数（afterEach）。

**34. ★★ Vue中如何实现子组件内的css样式名在项目中绝对唯一性**

在style标签上加上scoped属性

**35. ★★★★★ Vue3中的双向数据绑定proxy**

Proxy相当于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,我们可以这样认为,Proxy是Object.defineProperty的全方位加强版，它解决了之前defineProperty无法监听到数组变化等缺点。

**36. ★★★★★ 55、Vue和React中diff算法区别**

vue和react的diff算法，都是忽略跨级比较，只做同级比较。vue diff时调动patch函数，参数是vnode和oldVnode，分别代表新旧节点。

1.vue对比节点。当节点元素相同，但是classname不同，认为是不同类型的元素，删除重建，而react认为是同类型节点，只是修改节点属性。

2.vue的列表对比，采用的是两端到中间比对的方式，而react采用的是从左到右依次对比的方式。当一个集合只是把最后一个节点移到了第一个，react会把前面的节点依次移动，而vue只会把最后一个节点移到第一个。总体上，vue的方式比较高效。

**37. ★★★ 请你说一下 Vue 中 create 和 mount 的区别**

create为组件初始化阶段，在此阶段主要完成数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始，此时还未生成真实的DOM，也就无法获取和操作DOM元素。而mount主要完成从虚拟DOM到真实DOM的转换挂载，此时html已经渲染出来了，所以可以直接操作dom节点。

**38. ★★★★ Object.defineProperty有什么缺点**

1：无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应；

2：只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果，属性值是对象，还需要深度遍历。

**39. ★★★★ axios是什么？怎么使用？描述使用它实现登录功能的流程?**

axios 是请求后台资源的模块。 通过npm install axios -S来安装，在大多数情况下我们需要封装拦截器，在实现登录的过程中我们一般在请求拦截器中来加入token，在响应请求器中通过判断后端返回的状态码来对返回的数据进行不同的处理。如果发送的是跨域请求，需在配置文件中 config/index.js 进行代理配置。

**40. ★★ v-model是什么？Vue中标签怎么绑定事件？**

v-model是一个语法糖，这一个指令可以分为几个指令，它内部已经帮我们处理整合了。对于普通的文本框来说，v-model = v-bind:value + @input。对单选框和复选框来说，v-model = v-bind:checked + @change，并且它还有一个非常重要的功能，就是解决父子组件之间的通讯问题，可以提升我们的开发效率。在vue中通过使用<v-on:事件名 = 函数名>的方式来绑定事件。

**97. ★★★ 什么是v-model？它的原理是什么？**

1、v-model本质上是一个语法糖，可以看成是value + input 方法的语法糖。可以通过model的prop属性和event事件来进行自定义。

2、v-model是vue的双向绑定的指令，能将页面上控件输入的值同步更新到相关绑定的data属性， 也会在更新data绑定属性时候，更新页面上输入控件的值。

**41. ★★★ 路由懒加载**

把不同路由对应的组件分割成不同的代码块，然后当路由被访问时才加载对应的组件即为路由的懒加载，可以加快项目的加载速度，提高效率。通过这种格式来导入组件const foo = () => import('./foo.vue');

**43. ★★★ Vue中 computed 的原理，怎么追踪到它的依赖的？还是怎么确定是他的依赖变更 他才更新**

// 答案：

**44. ★★★★ 如果你是leader，做管理系统项目 Vue和React 怎么选择？**

评估项目成员的水平，如果成员js基础较好、编码能力较强则选择React，否则Vue。

评估系统的大小，如果想构建生态系统，则选择React，如果要求而快，简单和“能用就行",则选择Vue。

评估系统运行环境，如果你想要一个同时适用于Web端和原生APP的框架，请选择React(RN)。

**45. ★★★ Vuex的缺点**

如果您不打算开发大型单页应用，使用 Vuex 可能是繁琐冗余的，并且state中的值会伴随着浏览器的刷新而初始化，无缓存。

**46. ★★★★ Vue和React区别**

1：Vue 使用的是 web 开发者更熟悉的模板与特性，Vue的API跟传统web开发者熟悉的模板契合度更高，比如Vue的单文件组件是以模板+JavaScript+CSS的组合模式呈现，它跟web现有的HTML、JavaScript、CSS能够更好地配合。React 的特色在于函数式编程的理念和丰富的技术选型，Vue更加注重web开发者的习惯。

2：Vue跟React的最大区别在于数据的reactivity，就是反应式系统上。Vue提供反应式的数据，当数据改动时，界面就会自动更新，而React里面需要调用方法SetState。我把两者分别称为Push-based和Pull-based

**47. ★★★ Vue路由传参，刷新后还有吗**

通过params传参会出现参数丢失的情况，可以通过query的传参方式或者在路由匹配规则加入占位符即可以解决参数丢失的情况。

**48. ★★★ Vue深层次嵌套传值方法**

利用$attrs 和 $listeners

**49. ★★★ Vue组件如何引入使用**

1. 定义组件并抛出

2. import引入，并在component里面定义

3. 使用组件（注意首字母大写）

**50. ★★★★ Vue路由实现的底层原理**

在Vue中利用数据劫持defineProperty在原型prototype上初始化了一些getter,分别是router代表当前Router的实例 、 router代表当前Router的实例、router代表当前Router的实例、route 代表当前Router的信息。在install中也全局注册了router-view,router-link,其中的Vue.util.defineReactive, 这是Vue里面观察者劫持数据的方法，劫持\_route，当\_route触发setter方法的时候，则会通知到依赖的组件。

接下来在init中，会挂载判断是路由的模式，是history或者是hash,点击行为按钮，调用hashchange或者popstate的同时更\_route,\_route的更新会触发route-view的重新渲染。

**51. ★★★★ 如何封装一个通用组件**

通用组件的封装就是对可复用组件的解耦和样式复用，为了解耦一般数据都是通过父组件传递过来，在子组件中进行数据处理，对于一些较为复杂的数据可能还需要做数据验证，为了避免高耦合，逻辑最好放在父组件中，通过自定义事件将数据回传，子组件只是一个承载体，这样既降低耦合，保证子组件中数据和逻辑不会混乱。如果同一组件需要适应不同需求时，我们需要配合slot来使用，可以通过具名插槽灵活地解决了不同场景同一组件不同配置的问题。

**52. ★★ Vue 生命周期通常使用哪些**

常用的生命周期有，beforeCreate，created，beforeMount，mounted，beforeUpdate，updated，beforeDestroy，destroyed

**53. ★★ Vue 深层次的组件怎么和父组件通讯**

1、使用$attrs和$listeners

Vue.component('C', {

template: `

<div>

<p>我是C组件</p>

<input

type='text'

v-model='$attrs.msgc'

@input='$emit("getC", $attrs.msgc)'

/>

</div>

`

})

Vue.component('B', {

/\*\*

给C组件绑定$attrs属性和$listeners事件，C组件可以直接获取到A组件中传递下来的props（除了B组件中props声明的）

\*/

template: `

<div>

<p>我是B组件</p>

<input

type='text'

v-model='mymsg1'

@input="$emit('getChild', mymsg1)"

/>

<C v-bind='$attrs' v-on='$listeners'/>

</div>

`,

props: ['msg1'],

data () {

return {

mymsg1: this.msg1

}

}

})

Vue.component('A', {

template: `

<div id='app'>

<p>我是A组件</p>

<B

:msg1='msg1'

:msgc='msgc'

@getChild='getChild'

@getC='getC'

/>

</div>

`,

data () {

return {

msg1: 'A',

msgc: 'hello c!'

}

},

methods: {

getChild (val) {

console.log( val )

},

getC (val) {

console.log( val )

}

}

})

const app = new Vue({

el: '#app',

template: `

<A />

`

})

**54. ★★★★★ Vue 响应式原理**

1.观察者observer：首先通过观察者对data中的属性使用object.defineproperty劫持数据的getter和setter，通知订阅者，触发他的update方法，对视图进行更新

2.Compile：用来解析模板指令，并替换模板数据，初始化视图，初始化相应的订阅器

3.订阅者Watcher：订阅者接到通知后，调用update方法更新对应的视图

4.订阅器Dep：订阅者可能有多个，因此需要订阅器Dep来专门接收这些订阅者，并统一管理

但在vue3中抛弃了object.defineproperty方法，因为

1.Object.defineproperty无法监测**对象属性的添加和删除**、**数组索引和长度的变更**，因此vue重写了数组的push/pop/shift/unshift/splice/sort/reverse方法

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，这样很消耗性能

vue3中实现数据双向绑定的原理是数据代理，使用proxy实现。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。

**55. ★★★★ Vue proxy的原理**

主要通过Proxy对对象进行绑定监听处理，通过new Map对对象的属性操作进行处理，将要执行的函数匹配到存到对应的prop上面，通过每次的访问触发get方法，进行存方法的操作，通过修改触发set的方法，此时执行回调监听的函数，这样达到修改数据和视图的

[参考](https://www.yuque.com/zhongyangweizao/wheel/off7wo" \t "_blank)

**56. ★★★ Vue $forceUpdate的原理**

**1、作用：**

迫使 Vue 实例重新渲染。注意它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

**2、内部原理：**

Vue.prototype.$forceUpdate = function () {

const vm: Component = this

if (vm.\_watcher) {

vm.\_watcher.update()

}

}

实例需要重新渲染是在依赖发生变化的时候会通知watcher，然后通知watcher来调用update方法，就是这么简单。

**57. ★★★ v-for key**

* key是为Vue中的vnode标记的唯一id,通过这个key,我们的diff操作可以更准确、更快速
* diff算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的key与旧节点进行比对,然后超出差异.

diff程可以概括为：oldCh和newCh各有两个头尾的变量StartIdx和EndIdx，它们的2个变量相互比较，一共有4种比较方式。如果4种比较都没匹配，如果设置了key，就会用key进行比较，在比较的过程中，变量会往中间靠，一旦StartIdx>EndIdx表明oldCh和newCh至少有一个已经遍历完了，就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.

准确: 如果不加key,那么vue会选择复用节点(Vue的就地更新策略),导致之前节点的状态被保留下来,会产生一系列的bug. 快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度O(n),Map的时间复杂度仅仅为O(1)

[建议使用id，不建议使用索引](https://www.jianshu.com/p/4bd5e745ce95" \t "_blank)

**58. ★★★★ defineProperty在数据劫持后是如何通知数据的更新和视图的更新的**

vue的双向绑定是由数据劫持结合发布者－订阅者模式实现的，那么什么是数据劫持？vue是如何进行数据劫持的？说白了就是通过Object.defineProperty()来劫持对象属性的setter和getter操作，在数据变动时做你想要做的事情

我们已经知道实现数据的双向绑定，首先要对数据进行劫持监听，所以我们需要设置一个监听器Observer，用来监听所有属性。如果属性发生变化了，就需要告诉订阅者Watcher看是否需要更新。因为订阅者是有很多个，所以我们需要有一个消息订阅器Dep来专门收集这些订阅者，然后在监听器Observer和订阅者Watcher之间进行统一管理的。接着，我们还需要有一个指令解析器Compile，对每个节点元素进行扫描和解析，将相关指令（如v-model，v-on）对应初始化成一个订阅者Watcher，并替换模板数据或者绑定相应的函数，此时当订阅者Watcher接收到相应属性的变化，就会执行对应的更新函数，从而更新视图。因此接下去我们执行以下3个步骤，实现数据的双向绑定：

1.实现一个监听器Observer，用来劫持并监听所有属性，如果有变动的，就通知订阅者。

2.实现一个订阅者Watcher，每一个Watcher都绑定一个更新函数，watcher可以收到属性的变化通知并执行相应的函数，从而更新视图。

3.实现一个解析器Compile，可以扫描和解析每个节点的相关指令（v-model，v-on等指令），如果节点存在v-model，v-on等指令，则解析器Compile初始化这类节点的模板数据，使之可以显示在视图上，然后初始化相应的订阅者（Watcher）。

**60. ★★★★ Vuex如何实现跨组价的数据监听**

**61. ★★★★ 中间件机制是怎么生效的**

**62. ★★★★ axios谁封装的，怎么封装的**

// 使用axios用于对数据的请求

import axios from 'axios'

// 创建axios实例

const instance = axios.create({

baseURL: baseURL + version,

timeout: 5000

})

// 创建请求的拦截器

instance.interceptors.request.use(config => {

config.headers['Authorization'] = localStorage.getItem('token')

return config

}, error => {

return Promise.reject(error)

})

// 创建响应的拦截器

instance.interceptors.response.use(response => {

let res = null

// 对相应的数据进行过滤

if (response.status === 200) {

if (response.data && response.data.err === 0) {

res = response.data.data

} else if (response.data.err === -1) {

return alert('token无效')

}

} else {

return alert('请求失败')

}

return res

}, error => {

return Promise.reject(error)

})

export default instance

**63. ★★★ 为什么要设置key值，可以用index吗？为什么不能？**

vue中列表循环需加:key="唯一标识" 唯一标识可以是item里面id index等，因为vue组件高度复用增加Key可以标识组件的唯一性，为了更好地区别各个组件 key的作用主要是为了高效的更新虚拟DOM

**65. ★★★ diff复杂度原理及具体过程画图**

diff算法是一种通过同层的树节点进行比较的高效算法，避免了对树进行逐层搜索遍历，所以时间复杂度只有 O(n)。

diff算法有两个比较显著的特点：

1、比较只会在同层级进行, 不会跨层级比较。

2、在diff比较的过程中，循环从两边向中间收拢。

diff流程： 首先定义 oldStartIdx、newStartIdx、oldEndIdx 以及 newEndIdx 分别是新老两个 VNode 的两边的索引。

接下来是一个 while 循环，在这过程中，oldStartIdx、newStartIdx、oldEndIdx 以及 newEndIdx 会逐渐向中间靠拢。while 循环的退出条件是直到老节点或者新节点的开始位置大于结束位置。

while 循环中会遇到四种情况：

情形一：当新老 VNode 节点的 start 是同一节点时，直接 patchVnode 即可，同时新老 VNode 节点的开始索引都加 1。

情形二：当新老 VNode 节点的 end 是同一节点时，直接 patchVnode 即可，同时新老 VNode 节点的结束索引都减 1。

情形三：当老 VNode 节点的 start 和新 VNode 节点的 end 是同一节点时，这说明这次数据更新后 oldStartVnode 已经跑到了 oldEndVnode 后面去了。这时候在 patchVnode 后，还需要将当前真实 dom 节点移动到 oldEndVnode 的后面，同时老 VNode 节点开始索引加 1，新 VNode 节点的结束索引减 1。

情形四：当老 VNode 节点的 end 和新 VNode 节点的 start 是同一节点时，这说明这次数据更新后 oldEndVnode 跑到了 oldStartVnode 的前面去了。这时候在 patchVnode 后，还需要将当前真实 dom 节点移动到 oldStartVnode 的前面，同时老 VNode 节点结束索引减 1，新 VNode 节点的开始索引加 1。

while 循环的退出条件是直到老节点或者新节点的开始位置大于结束位置。

情形一：如果在循环中，oldStartIdx大于oldEndIdx了，那就表示oldChildren比newChildren先循环完毕，那么newChildren里面剩余的节点都是需要新增的节点，把[newStartIdx, newEndIdx]之间的所有节点都插入到DOM中

情形二：如果在循环中，newStartIdx大于newEndIdx了，那就表示newChildren比oldChildren先循环完毕，那么oldChildren里面剩余的节点都是需要删除的节点，把[oldStartIdx, oldEndIdx]之间的所有节点都删除

**66. ★★★ 怎么修改Vuex中的状态？Vuex中有哪些方法**

* 通过**this.$store.state.属性** 的方法来访问状态
* 通过**this.$store.commit(‘mutation中的方法’)** 来修改状态

**67. ★★★ vue-router路由传参的方式**

1、路由传参

1. query
2. // 方法一
3. <template>
4. <router-link
5. :to="{
6. path: 'blogDetail',
7. query: { id: item.id, views: item.views }
8. }"
9. tag="h2"
10. >
11. </router-link>
12. </template>
13. // 方法二
14. this.$router.push({
15. path: 'blogDetail',
16. query: {
17. id: item.id,
18. views: item.views
19. }
20. })
21. params
22. <template>
23. <router-link
24. :to="{
25. name: 'blogDetail',
26. params: { id: item.id, views: item.views }
27. }"
28. tag="h2"
29. >
30. </router-link>
31. </template>
32. this.$router.push({
33. name: 'blogDetail',
34. params: {
35. id: item.id,
36. views: item.views
37. }
38. })

**68. ★★★ hash history区别，怎么去解决history回退问题**

|  | **hash** | **history** |
| --- | --- | --- |
| url显示 | 有#，很Low | 无#，好看 |
| 回车刷新 | 可以加载到hash值对应页面 | 一般就是404掉了 |
| 支持版本 | 支持低版本浏览器和IE浏览器 | HTML5新推出的API |

**69. ★★★ 用过beforeEach吗？**

每次通过vue-router进行页面跳转，都会触发beforeEach这个钩子函数，这个回调函数共有三个参数，to，from，next这三个参数，to表示我要跳转的目标路由对应的参数，from表示来自那个路由，就是操作路由跳转之前的，即将离开的路由对应的参数，next是一个回调函数，一定要调用next方法来resolve这个钩子函数；

**70. ★★★ jQuery写的页面，切换到vue的页面，有卡顿吗？**

**71. ★★★ Vnode的缺点**

**72. ★★★ Vue中的单项数据流**

单向数据流指只能从一个方向来修改状态。

数据从父级组件传递给子组件，只能单向绑定。

子组件内部不能直接修改从父级传递过来的数据。

**73. ★★★ Vue组件中的Data为什么是函数，根组件却是对象呢？**

综上可知，如果data是一个函数的话，这样每复用一次组件，就会返回一份新的data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份data，就会造成一个变了全都会变的结果。

所以说vue组件的data必须是函数。这都是因为js的特性带来的，跟vue本身设计无关。

**74. ★★★ 你做过哪些Vue的性能优化？**

1、首屏加载优化

2、路由懒加载

{

path: '/',

name: 'home',

component: () => import('./views/home/index.vue'),

meta: { isShowHead: true }

}

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再进行解析。首先安装 webpack 提供的compression-webpack-plugin进行压缩,然后在 vue.config.js：

const CompressionWebpackPlugin = require('compression-webpack-plugin')

const productionGzipExtensions = ['js', 'css']......plugins: [

new CompressionWebpackPlugin(

{

algorithm: 'gzip',

test: new RegExp('\\.(' + productionGzipExtensions.join('|') + ')$'),

threshold: 10240,

minRatio: 0.8

}

)]....

4、启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理是将我们的压力分给其他服务器点。

5、代码层面优化

* computed 和 watch 区分使用场景
* computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
* watch：类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 ( 访问一个 API )，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。
* v-if 和 v-show 区分使用场景 v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。这里要说的优化点在于减少页面中 dom 总数，我比较倾向于使用 v-if，因为减少了 dom 数量。
* v-for 遍历必须为 item 添加 key，且避免同时使用 v-if v-for 遍历必须为 item 添加 key，循环调用子组件时添加 key，key 可以唯一标识一个循环个体，可以使用例如 item.id 作为 key 避免同时使用 v-if，v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度。

6、Webpack 对图片进行压缩

7、避免内存泄漏

8、减少 ES6 转为 ES5 的冗余代码

**75. ★★★★★ nextTick知道吗、实现的原理是什么？是宏任务还是微任务？**

微任务

原理：

nextTick方法主要是使用了宏任务和微任务，定义了一个异步方法，多次调用nextTick会将方法存入队列中，通过这个异步方法清空队列。

作用： nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick用于下次Dom更新循环结束之后执行延迟回调，在修改数据之后使用nextTick,则可以在回调中获取更新后的DOM。

**76. ★★★ 介绍下vue单页面和多页面区别**

单页应用 页面跳转---->js渲染 优点：页面切换快 缺点：首屏加载稍慢，seo差

多页应用 页面跳转---->返回html 优点：首屏时间快，seo效果好 缺点：页面切换慢

**77. ★★★ 介绍下vue父子组件生命周期的执行顺序**

[参考](https://blog.csdn.net/weixin_30616969/article/details/94973817" \t "_blank)

**78. ★★★★ 虚拟 dom 为什么会提高性能？**

虚拟DOM其实就是一个JavaScript对象。通过这个JavaScript对象来描述真实DOM，真实DOM的操作，一般都会对某块元素的整体重新渲染，采用虚拟DOM的话，当数据变化的时候，只需要局部刷新变化的位置就好了 ,

虚拟dom相当于在js和真实dom中间加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能

**具体实现步骤如下**

* 用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中
* 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异
* 把2所记录的差异应用到步骤1所构建的真正的DOM树上，视图就更新

**79. ★★★★ Vue 的 computed 的原理**

[参考](https://segmentfault.com/a/1190000010408657" \t "_blank)

**80. ★★★ Vue中key的作用？不加会怎么样？**

vue中列表循环需加:key="唯一标识" 唯一标识可以是item里面id index等，因为vue组件高度复用增加Key可以标识组件的唯一性，为了更好地区别各个组件 key的作用主要是为了高效的更新虚拟DOM

**81. ★★★ Vue的常用修饰符**

**一、v-model修饰符**

1、.lazy：

输入框改变，这个数据就会改变，lazy这个修饰符会在光标离开input框才会更新数据：

2、.trim：

输入框过滤首尾的空格：

3、.number：

先输入数字就会限制输入只能是数字，先字符串就相当于没有加number，注意，不是输入框不能输入字符串，是这个数据是数字：

**二、事件修饰符**

4、.stop：

阻止事件冒泡，相当于调用了event.stopPropagation()方法：

5、.prevent：

阻止默认行为，相当于调用了event.preventDefault()方法，比如表单的提交、a标签的跳转就是默认事件：

6、.self：

只有元素本身触发时才触发方法，就是只有点击元素本身才会触发。比如一个div里面有个按钮，div和按钮都有事件，我们点击按钮，div绑定的方法也会触发，如果div的click加上self，只有点击到div的时候才会触发，变相的算是阻止冒泡：

7、.once：

事件只能用一次，无论点击几次，执行一次之后都不会再执行

8、.capture：

事件的完整机制是捕获-目标-冒泡，事件触发是目标往外冒泡

9、.sync

对prop进行双向绑定

10、.keyCode：

监听按键的指令，具体可以查看vue的键码对应表

**82. ★★★ Vue计算属性和 Watch 的区别**

**methods（方法）：**

只要进行调用就会执行，不管依赖的值有没有改变。无缓存。

**computed（计算属性)：**

监听其所有依赖的变化，如果有变化会执行，没有变化不执行。有缓存，不用每次重新算。不支持异步。

详解：在vue的 模板内（{{}}）是可以写一些简单的js表达式的 ，很便利。但是如果在页面中使用大量或是复杂的表达式去处理数据，对页面的维护会有很大的影响。这个时候就需要用到computed 计算属性来处理复杂的逻辑运算。

1.优点：

在数据未发生变化时，优先读取缓存。computed 计算属性只有在相关的数据发生变化时才会改变要计算的属性，当相关数据没有变化是，它会读取缓存。而不必想 motheds方法 和 watch 方法是否每次都去执行函数。

2.setter 和 getter方法：（注意在vue中书写时用set 和 get）

setter 方法在设置值是触发。

getter 方法在获取值时触发。

**watch（侦听属性）:**

观察某一个变量，发生变化会执行。支持异步。Vue 实例将会在实例化时调用 $watch()，遍历 watch 对象的每一个属性。

一个对象，键是需要观察的表达式，值是对应回调函数。值也可以是方法名，或者包含选项的对象。

小结：

1.主动调用的方法写在methods里，依据某些变量的更新进行某种操作用computed或者watch。

2.computed和watch：如果要异步，只能用watch。如果是计算某个值推荐用computed，比如购物车全选单选功能，购物车计算总价小计功能。

**83. ★★★ Vue 中 v-on 可以绑定多个方法吗？**

<p v-on="{click:dbClick,mousemove:MouseClick}"></p>

**84. ★★★ Vue 中 template 的编译过程**

vue template模板编译的过程经过parse()生成ast(抽象语法树),optimize对静态节点优化，generate()生成render字符串 之后调用new Watcher()函数，用来监听数据的变化，render 函数就是数据监听的回调所调用的，其结果便是重新生成 vnode。 当这个 render 函数字符串在第一次 mount、或者绑定的数据更新的时候，都会被调用，生成 Vnode。 如果是数据的更新，那么 Vnode 会与数据改变之前的 Vnode 做 diff，对内容做改动之后，就会更新到 我们真正的 DOM

[参考](https://blog.csdn.net/qq_47008195/article/details/109148309" \t "_blank)

|  | **hash** | **history** |
| --- | --- | --- |
| url显示 | 有#，很Low | 无#，好看 |
| 回车刷新 | 可以加载到hash值对应页面 | 一般就是404掉了 |
| 支持版本 | 支持低版本浏览器和IE浏览器 | HTML5新推出的API |

**85. ★★★★ Vue中子组件是否可以修改props，如果想修改的话如何修改**

**86. ★★★ 说一下 VueRouter 的 hash 模式和 history 模式区别**

**87. ★★★ Vue中 router-link 和传统 a 链接的区别？**

组件支持用户在具有路由功能的应用中 (点击) 导航。 通过 to 属性指定目标地址，默认渲染成带有正确链接的 标签，可以通过配置 tag 属性生成别的标签.。 通过router-link进行跳转不会跳转到新的页面，也不会重新渲染，它会选择路由所指的组件进行渲染，避免了重复渲染的“无用功”。 总结：对比，router-link组件避免了不必要的重渲染,它只更新变化的部分从而减少DOM性能消耗

**88. ★★★ Vue插槽是什么？怎么使用？**

slot用于封装组件中，写在子组件 接收父组件动态传递子组件内容片断

slot插槽的使用方法其实就是类似于一个子组件或者标签的引用的过程，在父组件里面定义一个slot，给她起个name，然后组件引入到子组件，子组件里面有个元素的属性slot值等于name，然后父组件里面没有值的时候就可以显示子组件里面的信息了（切记：插槽slot要写在父组件里面！！！）

vue slot用法1：

slot主要是让组件的可扩展性更强

1.匿名slot使用

//定义组件

//定义组件

<div class="Component>

<slot></slot>

</div>

//使用方法

<component>

<p>slot内容，可以放任何标签</p>

</component>

2.具名slot使用

//定义组件

<div class="Component>

<slot name="Slot"></slot>

</div>

//使用方法

<component>

<p slot="Slot>可以放任何标签，但必须加上slot="Slot",不然会报错</p>

</component>

如果不在slot里加入任何标签，slot什么都不会显示。

**89. ★★★ 说一下Vue-router守卫有哪些。如何实现路由懒加载？**

* router.beforeEach 全局前置守卫 进入路由之前
* router.beforeResolve 全局解析守卫(2.5.0+) 在beforeRouteEnter调用之后调用
* router.afterEach 全局后置钩子 进入路由之后

1、 Vue异步加载技术

1：vue-router配置路由，使用vue的异步组件技术，可以实现懒加载，此时一个组件会生成一个js文件。

2：component: resolve => require(['放入需要加载的路由地址'], resolve)

2、 ES6推荐方式imprort ()----推荐使用

import Vue from 'vue';

import Router from 'vue-router';

// 官网可知：下面没有指定webpackChunkName，每个组件打包成一个js文件。

const Foo = () => import('../components/Foo')

const Aoo = () => import('../components/Aoo')

// 下面2行代码，指定了相同的webpackChunkName，会合并打包成一个js文件。

// const Foo = () => import(/\* webpackChunkName: 'ImportFuncDemo' \*/ '../components/Foo')

// const Aoo = () => import(/\* webpackChunkName: 'ImportFuncDemo' \*/ '../components/Aoo')

export default new Router({

routes: [

{

path: '/Foo',

name: 'Foo',

component: Foo

},

{

path: '/Aoo',

name: 'Aoo',

component: Aoo

}

]

})

3、 webpack提供的require.ensure()实现懒加载

1：vue-router配置路由，使用webpack的require.ensure技术，也可以实现按需加载。

2：这种情况下，多个路由指定相同的chunkName，会合并打包成一个js文件。

3：require.ensure可实现按需加载资源，包括js,css等。他会给里面require的文件单独打包，不会和主文件打包在一起。

4：第一个参数是数组，表明第二个参数里需要依赖的模块，这些会提前加载。

5：第二个是回调函数,在这个回调函数里面require的文件会被单独打包成一个chunk,不会和主文件打包在一起，这样就生成了两个chunk,第一次加载时只加载主文件。

6：第三个参数是错误回调。

7：第四个参数是单独打包的chunk的文件名

import Vue from 'vue';

import Router from 'vue-router';

const HelloWorld=resolve=>{

require.ensure(['@/components/HelloWorld'],()=>{

resolve(require('@/components/HelloWorld'))

})

}

Vue.use('Router')

export default new Router({

routes:[{

{path:'./',

name:'HelloWorld',

component:HelloWorld

}

}]

})

**90. ★★★ 什么是 Vue.js 动态组件与异步组件？**

**91. ★★★ 说一下Vue中路由跳转和传值的方式**

1、路由跳转

1. router-link组件 默认会被渲染成一个 标签，进行跳转，在组件中可以通过绑定to属性来指定要跳转的链接；tag属性指本来的标签
2. <router-link :to="/foo" tag="h2">Foo</router-link>
3. $router.push()方法

2、路由传参

1. query
2. // 方法一
3. <template>
4. <router-link
5. :to="{
6. path: 'blogDetail',
7. query: { id: item.id, views: item.views }
8. }"
9. tag="h2"
10. >
11. </router-link>
12. </template>
13. // 方法二
14. this.$router.push({
15. path: 'blogDetail',
16. query: {
17. id: item.id,
18. views: item.views
19. }
20. })
21. params
22. <template>
23. <router-link
24. :to="{
25. name: 'blogDetail',
26. params: { id: item.id, views: item.views }
27. }"
28. tag="h2"
29. >
30. </router-link>
31. </template>
32. this.$router.push({
33. name: 'blogDetail',
34. params: {
35. id: item.id,
36. views: item.views
37. }
38. })

**92. ★★★ 最近关注了什么新技术吗，简单说下你的理解（Vue3.0与Vue2.0对比)**

参考100题

**93. ★★★★ Vuex刷新页面数据会丢失吗？咋解决的？**

1、问题描述：页面刷新的时候vuex里的数据会重新初始化，导致数据丢失。因为vuex里的数据是保存在运行内存中的，当页面刷新时，页面会重新加载vue实例，vuex里面的数据就会被重新赋值。

2、解决思路：

办法一：将vuex中的数据直接保存到浏览器缓存中（sessionStorage、localStorage、cookie） 办法二：在页面刷新的时候再次请求远程数据，使之动态更新vuex数据 办法三：在父页面向后台请求远程数据，并且在页面刷新前将vuex的数据先保存至sessionStorage（以防请求数据量过大页面加载时拿不到返回的数据）

3、 解决过程：

3.1、选择合适的浏览器存储

3.2、解决方案

1. 由于state里的数据是响应式，所以sessionStorage存储也要跟随变化，而且只能通过mutations来改变state中的值。 首先在用户登录成功之后，然后把用户信息，菜单信息通过actions分发保存至vuex中。然后在菜单页面计算vuex中state的菜单数据，将数据解析组装成前端组件所需的格式，然后渲染组件，生成菜单树。如果页面没有刷新，则一切正常。
2. 监听浏览器刷新前事件，在浏览器刷新之前就把vuex里的数据保存至sessionStorage中，刷新成功后如果异步请求的数据还没返回则直接获取sessionStorage里的数据，否则获取vuex里的数据。

**94. ★★★★ 什么是 Vue.nextTick()？**

1、$nextTick 是在下次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，意思是 等你dom加载完毕以后再去调用nextTick()里面的数据内容

**95. ★★★ v-if 和 v-show的区别**

1、手段：v-if是动态的向DOM树添加或者删除DOM元素；v-show是通过设置DOM元素的display样式属性控制显示和隐藏。

2、编译过程：v-if切换有一个局部编译/卸载的过程， 切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换。

3、编译条件：v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译（编译被缓存？编译被缓存后，然后再切换的时候进行局部卸载); v-show是在任何条件下（首次条件是否为真）都被编译，然后被缓存，而且DOM元素保留

4、性能消耗：v-if有更高的切换消耗；v-show有更高的初始渲染消耗 使用场景：v-if适合运营条件不大可能改变；v-show适合频繁切换

6、相同点： v-show 都可以动态控制着dom元素的显示隐藏 不同点： v-if 的显示隐藏是将DOM元素整个添加或删除，v-show 的显示隐藏是为DOM元素添

7、加css的样式display，设置none或者是block，DOM元素是还存在的

8、在渲染多个元素的时候，可以把一个 元素作为包装元素，并使用v-if 进行条件判断，最终的渲染不会包含这个元素，v-show是不支持 语法

**96. ★★★ 简述Vue每个生命周期具体适合哪些场景**

| **生命周期** | **发生了什么** |
| --- | --- |
| beforeCreate | 初始化界面前 : 在当前阶段data、methods、computed以及watch上的数据和方法都不能被访问，都还没有完成初始化。 |
| created | 初始化界面后 : 在实例创建完成后发生，当前阶段已经完成了数据观测，也就是可以使用数据，更改数据，在这里更改数据不会触发updated函数,也就是不会更新视图。实例的data数据和methods方法都已经被初始化完毕了，可以正常访问 |
| beforeMount | 挂载前 ：完成模板编译，虚拟Dom已经创建完成，即将开始渲染。在此时也可以对数据进行更改，不会触发updated。数据还没有更新到页面上去。当编译完成之后，只是在内存中已经有了编译好的页面，但并未渲染。 |
| mounted | 挂载完成 ： 将编译好的模板挂载到页面 (虚拟DOM挂载) ，可以在这进行异步请求以及DOM节点的访问，在vue用$ref操作 |
| beforeUpdate | 更新数据前 ： 组件数据更新之前调用，数据都是新的,页面上数据都是旧的。将要根据最新的data数据，重新解析所有指令，从而重新渲染浏览器页面。 |
| updated | 组件更新后 ： render重新渲染 , 此时数据和界面都是新的 ,要注意的是避免在此期间更改数据，因为这可能会导致无限循环的更新 |
| beforeDestroy | 组件卸载前 : 实例销毁之前，在当前阶段实例完全可以被使用，我们可以在这时进行善后收尾工作，比如清除计时器 |
| destroyed | 组件卸载后 ： 组件已被拆解，数据绑定被卸除，监听被移出，子实例也统统被销毁。 |
| activited | keep-alive 专属 , 组件被激活时调用 |
| deactivated | keep-alive 专属 , 组件被销毁时调用 |

**98. ★★★ 简单说一下 微信小程序 与 Vue 的区别**

1、生命周期：

小程序的钩子函数要简单得多 。 vue的钩子函数在跳转新页面时，钩子函数都会触发，但是小程序的钩子函数，页面不同的跳转方式，触发的钩子并不一样。

在页面加载请求数据时，两者钩子的使用有些类似，vue一般会在created或者mounted中请求数据，而在小程序，会在onLoad或者onShow中请求数据。

2、数据绑定：

vue动态绑定一个变量的值为元素的某个属性的时候，会在变量前面加上冒号：

<img :src="imgSrc"/>

小程序 绑定某个变量的值为元素属性时，会用两个大括号括起来，如果不加括号，为被认为是字符串

<image src="{{imgSrc}}"></image>

3、列表循环

4、显示与隐藏元素

vue中，使用v-if 和v-show控制元素的显示和隐藏

小程序中，使用wx-if和hidden控制元素的显示和隐藏

5、事件处理

vue：使用v-on:event绑定事件，或者使用@event绑定事件

小程序中，全用bindtap(bind+event)，或者catchtap(catch+event)绑定事件

6、数据的双向绑定

在vue中,只需要再表单元素上加上v-model,然后再绑定data中对应的一个值，当表单元素内容发生变化时，data中对应的值也会相应改变 。

当表单内容发生变化时，会触发表单元素上绑定的方法，然后在该方法中，通过this.setData({key:value})来将表单上的值赋值给data中的对应值 。

7、绑定事件传参

在vue中，绑定事件传参挺简单，只需要在触发事件的方法中，把需要传递的数据作为形参传入就可以了

在小程序中，不能直接在绑定事件的方法中传入参数，需要将参数作为属性值，绑定到元素上的data-属性上，然后在方法中，通过e.currentTarget.dataset.\*的方式获取

8、父子组件通信

父组件向子组件传递数据，只需要在子组件通过v-bind传入一个值，在子组件中，通过props接收，即可完成数据的传递

父组件向子组件通信和vue类似，但是小程序没有通过v-bind，而是直接将值赋值给一个变量 在子组件properties中，接收传递的值

**99. ★★★ Vue的组件通信**

1、props和$emit

父组件向子组件传递数据是通过prop传递的，子组件传递数据给父组件是通过$emit触发事件

2、$attrs和$listeners

3、中央事件总线 bus

上面两种方式处理的都是父子组件之间的数据传递，而如果两个组件不是父子关系呢？这种情况下可以使用中央事件总线的方式。新建一个Vue事件bus对象，然后通过bus.$emit触发事件，bus.$on监听触发的事件。

4、provide和inject

父组件中通过provider来提供变量，然后在子组件中通过inject来注入变量。不论子组件有多深，只要调用了inject那么就可以注入provider中的数据。而不是局限于只能从当前父组件的prop属性来获取数据，只要在父组件的生命周期内，子组件都可以调用。

5、v-model

父组件通过v-model传递值给子组件时，会自动传递一个value的prop属性，在子组件中通过this.$emit(‘input’,val)自动修改v-model绑定的值

6、$parent和$children

7、boradcast和dispatch

8、vuex处理组件之间的数据交互 如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候才有上面这一些方法可能不利于项目的维护，vuex的做法就是将这一些公共的数据抽离出来，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

[vue中八种通信方式](https://blog.csdn.net/zhoulu001/article/details/79548350" \t "_blank)

**100. ★★★ 谈谈你对Vue3.0有什么了解？**

**一、六大亮点**

1. 性能比vue2.x快1.2~2倍
2. 支持tree-shaking，按需编译，体积比vue2.x更小
3. 支持组合API
4. 更好的支持TS
5. 更先进的组件

**二、性能比vue2.x快1.2~2倍如何实现的呢**

1.diff算法更快

vue2.0是需要全局去比较每个节点的，若发现有节点发生变化后，就去更新该节点

vue3.0是在创建虚拟dom中，会根据DOM的的内容会不会发生内容变化，添加静态标记， 谁有flag！比较谁。

2、静态提升

vue2中无论元素是否参与更新，每次都会重新创建，然后再渲染 vue3中对于不参与更新的元素，会做静态提升，只被创建一次，在渲染时直接复用即可

3、事件侦听缓存

默认情况下，onclick为动态绑定，所以每次都会追踪它的变化，但是因为是同一函数，没有必要追踪变化，直接缓存复用即可

在之前会添加静态标记8 会把点击事件当做动态属性 会进行diff算法比较， 但是在事件监听缓存之后就没有静态标记了，就会进行缓存复用

**三、为什么vue3.0体积比vue2.x小**

在vue3.0中创建vue项目 除了vue-cli，webpack外还有 一种创建方法是Vite Vite是作者开发的一款有意取代webpack的工具，其实现原理是利用ES6的import会发送请求去加载文件的特性，拦截这些请求，做一些预编译，省去webpack冗长的打包时间

**四、vue3.0组合API**

3.说一说vue3.0的组合API跟之前vue2.0在完成业务逻辑上的区别：

在vue2.0中： 主要是往data 和method里面添加内容，一个业务逻辑需要什么data和method就往里面添加，而组合API就是 有一个自己的方法，里面有自己专注的data 和method。

4.再说一下组合API的本质是什么： 首先composition API（组合API） 和 Option API（vue2.0中的data和method）可以共用 composition API（组合API）本质就是把内容添加到Option API中进行使用

**五、ref和reactive的简单理解**

1.ref和reactive都是vue3的监听数据的方法，本质是proxy 2.ref 基本类型复杂类型都可以监听(我们一般用ref监听基本类型)，reactive只能监听对象（arr，json） 3.ref底层还是reactive

**六、对生命周期的监听（ref获取属性）**

**七、proxy响应式本质**

**101. ★★★ Vue如何在用户没登陆的时候重定向登录界面？**

现在 我们需要实现这样 一个功能，登录拦截，其实就是 路由拦截，首先在定义路由的时候就需要多添加一个自定义字段requireAuth，用于判断该路由的访问是否需要登录。如果用户已经登录，则顺利进入路由， 否则就进入登录页面。在路由管理页面添加meta字段

{

path:'/manage',

name:'manage',

component:manage,

meta:{requireAuth:true}

}

在入口文件中，添加路由守卫

1. 先判断该路由是否需要登录权限
2. 判断本地是否存在token，如果存在token就next()，不存在token重定向到登录页

**102. ★★★★★ Vue项目常见优化点**

1、首屏加载优化

2、路由懒加载

{

path: '/',

name: 'home',

component: () => import('./views/home/index.vue'),

meta: { isShowHead: true }

}

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再进行解析。首先安装 webpack 提供的compression-webpack-plugin进行压缩,然后在 vue.config.js：

const CompressionWebpackPlugin = require('compression-webpack-plugin')

const productionGzipExtensions = ['js', 'css']......plugins: [

new CompressionWebpackPlugin(

{

algorithm: 'gzip',

test: new RegExp('\\.(' + productionGzipExtensions.join('|') + ')$'),

threshold: 10240,

minRatio: 0.8

}

)]....

4、启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理是将我们的压力分给其他服务器点。

5、代码层面优化

* computed 和 watch 区分使用场景
* computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
* watch：类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 ( 访问一个 API )，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。
* v-if 和 v-show 区分使用场景 v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。这里要说的优化点在于减少页面中 dom 总数，我比较倾向于使用 v-if，因为减少了 dom 数量。
* v-for 遍历必须为 item 添加 key，且避免同时使用 v-if v-for 遍历必须为 item 添加 key，循环调用子组件时添加 key，key 可以唯一标识一个循环个体，可以使用例如 item.id 作为 key 避免同时使用 v-if，v-for 比 v-if 优先级高，如果每一次都需要遍历整个数组，将会影响速度。

6、Webpack 对图片进行压缩

7、避免内存泄漏

8、减少 ES6 转为 ES5 的冗余代码

**103. ★★★ Vuex和redux有什么区别？他们的共同思想。**

**Redux和Vuex区别**

* Vuex改进了Redux中的Action和Reducer函数，以mutations变化函数取代Reducer，无需switch，只需在对应的mutation函数里改变state值就可以
* Vuex由于Vue自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的state就可以
* Vuex数据流的顺序是:View调用store.commit提交对应的请求到Store中对应的mutation函数 -- store改变(vue检测到数据变化自动渲染)

**共同思想**

* 单一的数据源
* 变化可以预测
* 本质上:Redux和Vuex都是对MVVM思想的服务，将数据从视图中抽离的一种方案
* 形式上:Vuex借鉴了Redux，将store作为全局的数据中心，进行数据管理

# JavaScript面试真题

**1. ★★ 介绍一下JS的内置类型有哪些？**

1. 空类型:null

2. 未定义:undefined

3. 布尔:boolean

4. 数字:number

5. 字符串:string

6. 符号:symbol(ES6新增)

7. 对象:object

除了对象之外,其他为基本类型.

**2. ★★★★ 介绍一下 typeof 区分类型的原理**

typeof原理： 不同的对象在底层都表示为二进制，在Javascript中二进制前（低）三位存储其类型信息。

000: 对象

010: 浮点数

100:字符串

110: 布尔

1: 整数

/\*----------------------------------------------\*/

typeof null 为"object", 原因是因为 不同的对象在底层都表示为二进制，在Javascript中二进制前（低）三位都为0的话会被判断为Object类型，null的二进制表示全为0，自然前三位也是0，所以执行typeof时会返回"object"

**3. ★★★ 介绍一下类型转换**

/\*-------------------显式转换---------------------\*/

1. toString() // 转化为字符串，不可以转null和underfined

2. Number() // 转换为数字，字符串中有一个不是数值的字符，返回NaN

3. parseInt() // 转换为数字，第一个字符不是数字或者符号就返回NaN

4. String() // 转换为字符串，

5. Boolean() // 转换为布尔值

/\*-------------------隐式转换(+-)---------------------\*/

当 JavaScript 尝试操作一个 "错误" 的数据类型时，会自动转换为 "正确" 的数据类型

1. num + "" -> String

2. num + bool -> num

// 当加号运算符时，String和其他类型时，其他类型都会转为 String；其他情况，都转化为Number类型

3. string - num -> num

// 其他运算符时， 基本类型都转换为 Number，String类型的带有字符的比如：

4. 'a1' - num -> NaN

// 与undefined 一样。

/\*-------------------隐式转换(逻辑表达式)---------------------\*/

1. 对象和布尔值比较

对象和布尔值进行比较时，对象先转换为字符串，然后再转换为数字，布尔值直接转换为数字

[] == true; //false []转换为字符串'',然后转换为数字0,true转换为数字1，所以为false

2. 对象和字符串比较

对象和字符串进行比较时，对象转换为字符串，然后两者进行比较。

[1,2,3] == '1,2,3' // true [1,2,3]转化为'1,2,3'，然后和'1,2,3'， so结果为true;

3. 对象和数字比较

对象和数字进行比较时，对象先转换为字符串，然后转换为数字，再和数字进行比较。

[1] == 1; // true `对象先转换为字符串再转换为数字，二者再比较 [1] => '1' => 1 所以结果为true

4. 字符串和数字比较

字符串和数字进行比较时，字符串转换成数字，二者再比较。

'1' == 1 // true

5. 字符串和布尔值比较

字符串和布尔值进行比较时，二者全部转换成数值再比较。

'1' == true; // true

6. 布尔值和数字比较

布尔值和数字进行比较时，布尔转换为数字，二者比较。

true == 1 // true

**4. ★★★★ 说说你对 JavaScript 的作用域的理解。什么是作用域链？**

在 JavaScript 中有两种作用域类型：

1. 局部作用域:只能在函数内部访问它们

2. 全局作用域:网页的所有脚本和函数都能够访问它

JavaScript 拥有函数作用域：每个函数创建一个新的作用域。

作用域决定了这些变量的可访问性（可见性）。

函数内部定义的变量从函数外部是不可访问的（不可见的）。

作用域链：

当查找变量的时候，会先从当前上下文的变量对象中查找，

如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。

这样由多个执行上下文的变量对象构成的链表就叫做作用域链

**5. ★★ 解释下 let 和 const 的块级作用域**

/\*------------let-----------\*/

1. let声明的仅在块级作用域内有效，

2. let不会发生变量提升的现象，所以一定要在定义后使用，否则报错。

3. 暂时性死区：只要块级作用域内存在let命令，它所声明的变量就绑定这个区域，不再受外部影响。

4. 不允许重复声明，let不允许在相同作用域内，重复声明同一个变量：

/\*-----------const----------\*/

1. 声明一个只读的常量。一旦声明，常量的值就不能改变。

2. 一旦声明，就要立即初始化，否则也报错。

3. const命令声明的常量也不提升，同样存在暂时性死区，只能在声明的位置后使用。

4. 也不可以重复声明。

**6. ★★★★ 说说你对执行上下文的理解**

执行上下文有且只有三类，全局执行上下文，函数上下文，与eval上下文(eval一般不会使用)

1. 全局执行上下文：

全局执行上下文只有一个，也就是我们熟知的window对象，我们能在全局作用域中通过this直接访问到它

2. 函数执行上下文

函数执行上下文可存在无数个，每当一个函数被调用时都会创建一个函数上下文；

需要注意的是，同一个函数被多次调用，都会创建一个新的上下文。

3. 执行上下文栈(下文简称执行栈)也叫调用栈，

执行栈用于存储代码执行期间创建的所有上下文，具有LIFO（Last In First Out后进先出，也就是先进后出）的特性。

JS代码首次运行，都会先创建一个全局执行上下文并压入到执行栈中，之后每当有函数被调用，都会创建一个新的函数执行上下文并压入栈内；由于执行栈LIFO的特性，所以可以理解为，JS代码执行完毕前在执行栈底部永远有个全局执行上下文。

**7. ★★★ 对闭包的看法，为什么要用闭包？说一下闭包的原理以及应用场景？闭包的 this 指向问题？**

闭包的作用：

1. 在外部访问函数内部的变量

2. 让函数内的局部变量可以一直保存下去

3. 模块化私有属性和公共属性

闭包的原理：

全局变量生存周期是永久，局部变量生存周期随着函数的调用介绍而销毁。

闭包就是 在函数中定义且成为该函数内部返回的函数的自由变量 的变量，该变量不会随着外部函数调用结束而销毁。

（注：不光是变量，函数内声明的函数也可以形成闭包）

当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。

闭包的应用场景：

// 1. 返回值 最常见的一种形式

var fun\_1 = function () {

var name = "limo";

return function () {

return name;

}

}

var fun\_1 = function () {

var name = "limo";

return name;

}

var fu\_1 = fun\_1();

console.log("fu\_1():" + fu\_1());

// 2. 函数赋值 一种变形的形式是将内部函数赋值给一个外部变量

var f2;

var fun\_2 = function () {

var name = "limop"

var a = function () {

return name;

}

f2 = a;

}

f2();

console.log(f2);

// 3. 函数参数 通过函数参数引用内部函数产生闭包

var fn\_3 = function (f3) {

console.log(f3);

}

function fun\_3() {

var name = "limo";

var a = function () {

return name;

}

fn\_3(a)

}

fun\_3();

// 4. IIFE(自执行函数)

var fn\_4 = function (f4) {

console.log(f4);

};

(function fun\_4() {

var name = "limo";

var a = function () {

return name;

}

fn\_3(a)

})();

// 5. 循环赋值

function foo(){

var arr = [];

for(var i = 0; i < 10; i++){

arr[i] = (function(n){

return function(){

return n;

}

})(i)

}

return arr;

}

var bar = foo();

console.log(bar[3]());

// 6. getter和setter

// getter和setter函数来将要操作的变量保存在函数内部，防止暴露在外部

var getValue, setValue;

(function () {

var num = 0

getValue = function () {

return num

}

setValue = function (v) {

if (typeof v === 'number') {

num = v

}

}

})();

console.log(getValue()); //0

setValue(10);

console.log(getValue()) //10

// 7.迭代器（计数器）

var add = function(){

var num = 0;

return function(){

return ++num;

}

}();

console.log(add());

console.log(add());

function setUp(arr){

var i = 0;

return function(){

return arr[i++];

}

}

var next = setUp(['Steve','Alex','LingYi']);

console.log(next());

console.log(next());

console.log(next());

// 8.触发事件

window.onload = function (){

var btn = document.querySelector('.btn')

btn.onclick = function (){//事件相当于在全局触发

btn.style.color = 'red'//保持对上层作用域的引用 btn

console.log('abc')

// this

}

}

闭包的this指向问题：

var myNumber = {

value: 1,

add: function(i){

var helper = function(i){

console.log(this);

this.value += i;

}

helper(i);

}

}

myNumber.add(1);

1.this指向window对象（因为匿名函数的执行具有全局性，所以其this对象指向window）；

2.不能实现value加1（每个函数在被调用时都会自动取得两个特殊变量，this和arguments，内部函数在搜索这两个对象时，只会搜索到其活动对象为止，所以不能实现访问外部函数的this对象）；

3.修改代码实现正确功能

第一种解决方法：

var myNumber={

value:1,

add:function(i){

var that=this;//定义变量that用于保存上层函数的this对象

var helper=function(i){

console.log(that);

that.value+=i;

}

helper(i);

}

}

myNumber.add(1);

第二种解决方法：

var myNumber={

value:1,

add:function(i){

var helper=function(i){

this.value+=i;

}

helper.apply(this,[i]);//使用apply改变helper的this对象指向，使其指向myNumber对象

}

}

myNumber.add(1);

第三种解决方法

var myNumber={

value:1,

add:function(i){

var helper=function(i){

this.value+=i;

}.bind(this,i);//使用bind绑定，和apply相似，只是它返回的是对函数的引用，不会立即执行

helper(i);

}

}

myNumber.add(1);

**8. ★★★ 简述闭包的问题以及优化**

闭包的缺点：占用内层空间 大量使用闭包会造成 栈溢出

由于闭包会一直占用内存空间，直到页面销毁，我们可以主动将已使用的闭包销毁：

将闭包函数赋值为null 可以销毁闭包

**9. ★★★ 如何确定 this 指向？改变 this 指向的方式有哪些？**

this 指向：

1. 全局上下文（函数外）

无论是否为严格模式，均指向全局对象。注意:严格模式下全局对象为undifined

2. 函数上下文（函数内）

默认的，指向函数的调用对象，且是最直接的调用对象：

简单调用，指向全局对象注意:严格模式下全局对象为undifined，某些浏览器未实现此标准也可能会是window

改变this指向的方式：

1. 第一种： new关键字改变this指向

//构造函数版this

function Fn(){

this.user = "李某";

}

var a = new Fn();

console.log(a.user); //李某

/\*----------------------------------------\*/

2. 第二种： call()

// 把b添加到第一个参数的环境中，简单来说，this就会指向那个对象

var a = {

user:"李某",

fn:function(){

console.log(this.user); //李某

}

}

var b = a.fn;

b.call(a); //若不用call，则b()执行后this指的是Window对象

/\*----------------------------------------\*/

3. 第三种：apply()

// apply方法和call方法有些相似，它也可以改变this的指向，也可以有多个参数，但是不同的是，第二个参数必须是一个数组

var a = {

user:"李某",

fn:function(){

console.log(this.user); //李某

}

}

var b = a.fn;

b.apply(a);

/\*----------------------------------------\*/

4. 第四种：bind()

// bind方法返回的是一个修改过后的函数,

// bind也可以有多个参数，并且参数可以执行的时候再次添加，但是要注意的是，参数是按照形参的顺序进行的。

var a = {

user:"李某",

fn:function(){

console.log(this.user); //李某

}

}

var b = a.fn;

var c = b.bind(a);

c();

**10. ★★★ 介绍箭头函数的 this**

由于箭头函数不绑定this， 它会捕获其所在（即定义的位置）上下文的this值， 作为自己的this值

1. 所以 call() / apply() / bind() 方法对于箭头函数来说只是传入参数，对它的 this 毫无影响。

2. 考虑到 this 是词法层面上的，严格模式中与 this 相关的规则都将被忽略

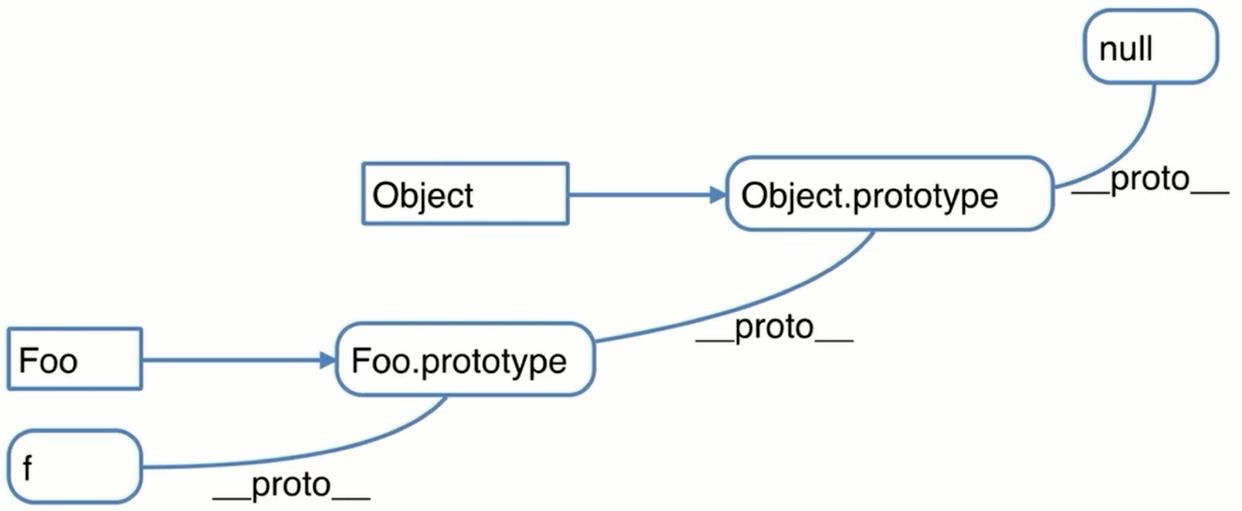
作为方法的箭头函数this指向全局window对象，而普通函数则指向调用它的对象

**11. ★★★ 谈一下你对原型链的理解，画一个经典的原型链图示**

原型链:

因为每个对象和原型都有原型，对象的原型指向原型对象，

而父的原型又指向父的父，这种原型层层连接起来的就构成了原型链。



**12. ★★★ ES5/ES6 的继承除写法以外还有什么区别？**

1. ES5 的继承实质上是先创建子类的实例对象，然后再将父类的方法添加 到 this 上(Parent.apply(this)).

2. ES6 的继承机制完全不同，实质上是先创建父类的实例对象 this(所以必 须先调用父类的super()方法)，然后再用子类的构造函数修改 this。

3. ES5 的继承时通过原型或构造函数机制来实现。

4. ES6 通过 class 关键字定义类，里面有构造方法，类之间通过 extends 关 键字实现继承。

5. 子类必须在 constructor 方法中调用 super 方法，否则新建实例报错。因 为子类没有自己的 this对象，而是继承了父类的 this 对象，然后对其进行加工。 如果不调用 super 方法，子类得不到 this 对象。

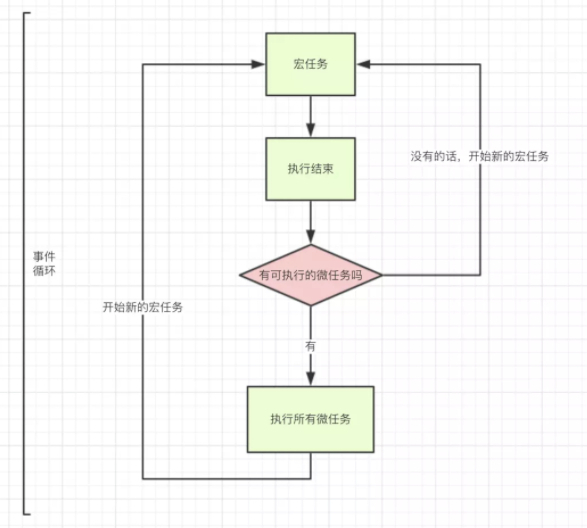
6. 注意 super 关键字指代父类的实例，即父类的 this 对象。 注意:在子类构造函数中，调用 super 后，才可使用 this关键字，否则报错

**13. ★★★★ 你对事件循环有了解吗？说说看**

Event Loop(事件循环)中，每一次循环称为 tick, 每一次tick的任务如下：

* 执行栈选择最先进入队列的宏任务(通常是script整体代码)，如果有则执行
* 检查是否存在 Microtask，如果存在则不停的执行，直至清空 microtask 队列
* 更新render(每一次事件循环，浏览器都可能会去更新渲染)
* 重复以上步骤

宏任务 > 所有微任务 > 宏任务，如下图所示：



从上图我们可以看出：

1. 将所有任务看成两个队列：执行队列与事件队列。

2. 执行队列是同步的，事件队列是异步的，宏任务放入事件列表，微任务放入执行队列之后，事件队列之前。

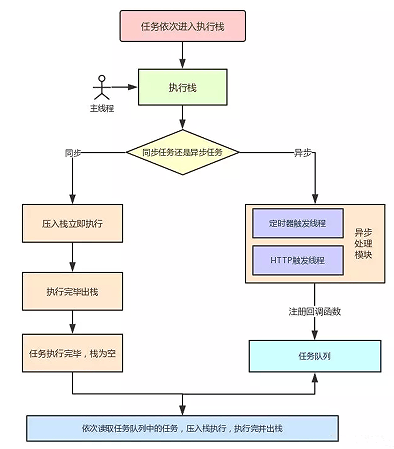
3. 当执行完同步代码之后，就会执行位于执行列表之后的微任务，然后再执行事件列表中的宏任务

**14. ★★★★ 微任务和宏任务有什么区别？**

|  | **宏任务（macrotask）** | **微任务（microtask）** |
| --- | --- | --- |
| 谁发起的 | 宿主（Node、浏览器） | JS引擎 |
| 具体事件 | 1. script (可以理解为外层同步代码) 2. setTimeout/setInterval 3. UI rendering/UI事件 4. postMessage，MessageChannel 5. setImmediate，I/O（Node.js） | 1. Promise 2. MutaionObserver 3. Object.observe（已废弃；Proxy 对象替代） 4. process.nextTick（Node.js） |
| 谁先运行 | 后运行 | 先运行 |
| 会触发新一轮Tick吗 | 会 | 不会 |

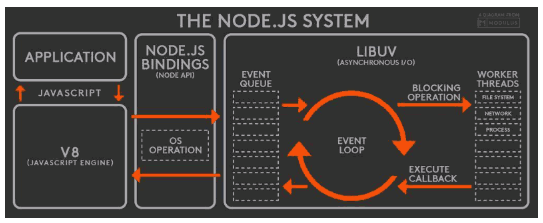
**15. ★★★★★ 浏览器和 Node 事件循环的区别？**

浏览器中的事件循环：



Node中的事件循环：

Node 中的 Event Loop 和浏览器中的是完全不相同的东西。Node.js 采用 V8 作为 js 的解析引擎，而 I/O 处理方面使用了自己设计的 libuv，libuv 是一个基于事件驱动的跨平台抽象层，封装了不同操作系统一些底层特性，对外提供统一的 API，事件循环机制也是它里面的实现（下文会详细介绍）。



Node.js 的运行机制如下:

* V8 引擎解析 JavaScript 脚本。
* 解析后的代码，调用 Node API。
* libuv 库负责 Node API 的执行。它将不同的任务分配给不同的线程，形成一个 Event Loop（事件循环），以异步的方式将任务的执行结果返回给 V8 引擎。
* V8 引擎再将结果返回给用户。

**16. ★★★ 异步解决方案有哪些？**

解决方案：

/\*---------1.回调函数callback：----------\*/

被作为实参传入另一函数，并在该外部函数内被调用，用以来完成某些任务的函数。如setTimeOut，ajax请求，readFile等。

例：

function greeting(name) {

alert('Hello ' + name);

}

function processUserInput(callback) {

var name = prompt('请输入你的名字。');

callback(name);

}

processUserInput(greeting);

优点：

解决了异步的问题。

缺点：

回调地狱：多个回调函数嵌套的情况，使代码看起来很混乱，不易于维护。

/\*---------2.事件发布订阅：---------\*/

当一个任务执行完成后，会发布一个事件，当这个事件有一个或多个‘订阅者’的时候，会接收到这个事件的发布，执行相应的任务，这种模式叫发布订阅模式。如node的events,dom的事件绑定

例：

document.body.addEventListener('click',function(){

alert('订阅了');

},false);

document.body.click();

优点：

时间对象上的解耦。

缺点：

消耗内存，过度使用会使代码难以维护和理解

/\*---------3.Promise：---------\*/

Promise是es6提出的异步编程的一种解决方案。

Promise 对象有三种状态：

pending: 初始状态，既不是成功，也不是失败状态。

fulfilled: 意味着操作成功完成。

rejected: 意味着操作失败。

promise的状态只能从pending变成fulfilled，和pending变成rejected，状态一旦改变，就不会再改变，且只有异步操作的结果才能改变promise的状态。

例：

let promise = new Promise(function (resolve, reject) {

fs.readFile('./1.txt', 'utf8', function (err, data) {

resolve(data)

})

})

promise

.then(function (data) {

console.log(data)

})

优点：

解决了回调地狱的问题，将异步操作以同步操作的流程表达出来。

缺点：

无法取消promise。如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。当执行多个Promise时，一堆then看起来也很不友好。

/\*---------4.Generator：---------\*/

Generator是es6提出的另一种异步编程解决方案，需要在函数名之前加一个\*号，函数内部使用yield语句。Generaotr函数会返回一个遍历器，可以进行遍历操作执行每个中断点yield。

例：

function \* count() {

yield 1

yield 2

return 3

}

var c = count()

console.log(c.next()) // { value: 1, done: false }

console.log(c.next()) // { value: 2, done: false }

console.log(c.next()) // { value: 3, done: true }

console.log(c.next()) // { value: undefined, done: true }

优点：

没有了Promise的一堆then(),异步操作更像同步操作，代码更加清晰。

缺点：

不能自动执行异步操作，需要写多个next()方法，需要配合使用Thunk函数和Co模块才能做到自动执行。

/\*---------5.async/await：---------\*/

async是es2017引入的异步操作解决方案，可以理解为Generator的语法糖，async等同于Generator和co模块的封装，async 函数返回一个 Promise。

例：

async function read() {

let readA = await readFile('data/a.txt')

let readB = await readFile('data/b.txt')

let readC = await readFile('data/c.txt')

console.log(readA)

console.log(readB)

console.log(readC)

}

read()

优点：

内置执行器，比Generator操作更简单。async/await比\*yield语义更清晰。返回值是Promise对象，可以用then指定下一步操作。代码更整洁。可以捕获同步和异步的错误。

**17. ★★★ async 和 await 、promise的区别 和 这两个的本质**

/\*---------Promise概念：---------\*/

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大，简单地说，Promise好比容器，里面存放着一些未来才会执行完毕（异步）的事件的结果，而这些结果一旦生成是无法改变的

/\*---------async await概念：---------\*/

async await也是异步编程的一种解决方案，他遵循的是Generator 函数的语法糖，他拥有内置执行器，不需要额外的调用直接会自动执行并输出结果，它返回的是一个Promise对象。

两者的区别：

Promise的出现解决了传统callback函数导致的“地域回调”问题，但它的语法导致了它向纵向发展行成了一个回调链，遇到复杂的业务场景，这样的语法显然也是不美观的。而async await代码看起来会简洁些，使得异步代码看起来像同步代码，await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，只有这一句代码执行完，才会执行下一句。

async await与Promise一样，是非阻塞的。

async await是基于Promise实现的，可以说是改良版的Promise，它不能用于普通的回调函数。

**18. ★★★ 简述 aync await 的好处**

1. async/await最重要的好处是同步编程风格

2. async/await有本地浏览器支持。截至今天，所有主流浏览器 查看都完全支持异步功能。

3. async关键字。它声明 getBooksByAuthorWithAwait()函数返回值确保是一个 promise，以便调用者可以安全调用 getBooksByAuthorWithAwait().then(...)或 await getBooksByAuthorWithAwait()

**19. ★★★ 移动端点击事件 300ms 延迟如何去掉？原因是什么？**

300毫秒原因：

当用户第一次点击屏幕后，需要判断用户是否要进行双击操作，于是手机会等待300毫秒，

解决方法：FastClick.js

FastClick 是 FT Labs 专门为解决移动端浏览器 300 毫秒点击延迟问题所开发的一个轻量级的库。FastClick的实现原理是在检测到touchend事件的时候，会通过DOM自定义事件立即出发模拟一个click事件，并把浏览器在300ms之后的click事件阻止掉。

**20. ★★★ Cookie 有哪些属性？其中HttpOnly，Secure，Expire分别有什么作用？**

Cookie属性：

name　　字段为一个cookie的名称。

value　　字段为一个cookie的值。

domain　　字段为可以访问此cookie的域名。

path　　字段为可以访问此cookie的页面路径。 比如domain是abc.com,path是/test，那么只有/test路径下的页面可以读取此cookie。

expires/Max-Age 　　字段为此cookie超时时间。若设置其值为一个时间，那么当到达此时间后，此cookie失效。不设置的话默认值是Session，意思是cookie会和session一起失效。当浏览器关闭(不是浏览器标签页，而是整个浏览器) 后，此cookie失效。

Size　　字段 此cookie大小。

http　　字段 cookie的httponly属性。若此属性为true，则只有在http请求头中会带有此cookie的信息，而不能通过document.cookie来访问此cookie。

secure　　 字段 设置是否只能通过https来传递此条cookie

1 secure属性

当设置为true时，表示创建的 Cookie 会被以安全的形式向服务器传输，也就是只能在 HTTPS 连接中被浏览器传递到服务器端进行会话验证，如果是 HTTP 连接则不会传递该信息，所以不会被窃取到Cookie 的具体内容。

2 HttpOnly属性

如果在Cookie中设置了"HttpOnly"属性，那么通过程序(JS脚本、Applet等)将无法读取到Cookie信息，这样能有效的防止XSS攻击。

3 Expire属性

设置Cookie的失效时间：

**21. ★★★★ 如何实现函数的柯里化？比如 add(1)(2)(3)**

/\*-----解决方法1：-----\*/

function add () {

var args = Array.prototype.slice.call(arguments);

var fn = function () {

var sub\_arg = Array.prototype.slice.call(arguments);

　　 // 把全部的参数聚集到参数的入口为一个参数： args.concat(sub\_arg)

return add.apply(null, args.concat(sub\_arg));

}

fn.valueOf = function () {

return args.reduce(function(a, b) {

return a + b;

})

}

return fn;

}

console.log(add(1,2)) // 3

console.log(add(1)(2)) // 3

console.log(add(1)(2)(3)) // 6

console.log(add(1,2,3)(4)) // 10

/\*-----解决方法2：-----\*/

function add () {

var args = Array.prototype.slice.call(arguments);

var fn = function () {

// 把参数都放在一个相当于全局变量的 args 里面

args.push(...arguments)

return fn;

}

fn.valueOf = function () {

return args.reduce(function(a, b) {

return a + b;

})

}

return fn;

}

console.log(add(1,2)) // 3

console.log(add(1)(2)) // 3

console.log(add(1)(2)(3)) // 6

console.log(add(1,2,3)(4)) // 10

**22. ★★★★ 什么是反柯里化**

在JavaScript中，当我们调用对象的某个方法时，其实不用去关心该对象原本是否被设计为拥有这个方法，这是动态类型语言的特点。可以通过反柯里化(uncurrying)函数实现，让一个对象去借用一个原本不属于他的方法。

**23. ★★ 将 [1,2] 与 [3,[4]] 合并为 [1,2,3,[4]]**

JS数组合并方法:

let arr3 = [1,2].concat([3,[4]]); //[1,2,3,[4]]

**24. ★★ Array.forEach() 与 Array.map() 的区别，Array.slice() 与 Array.splice() 的区别？**

/\*-----forEach-----\*/

forEach不支持return,对原来的数组也没有影响。但是我们可以自己通过数组的索引来修改原来的数组

var ary = [12,23,24,42,1];

var res = ary.forEach(function (item,index,input) {

input[index] = item\*10;

})

console.log(res);//-->undefined;

console.log(ary);//-->会对原来的数组产生改变；

/\*-----map-----\*/

map支持return返回值，也不影响原数组，但是会返回一个新的数组

var ary = [12,23,24,42,1];

var res = ary.map(function (item,index,input) {

return item\*10;

})

console.log(res);//-->[120,230,240,420,10];

console.log(ary);//-->[12,23,24,42,1]；

array.slice(start,end)函数是取array数组中指定的一些元素：

根据数组下标start和end，两个参数为取值的开始和结束下标，取出的值不包括end位置的值，生成一个新数组作为返回值；

这里end可以为空，为空则取从start位置到数组结尾的元素，生成新数组。

array.splice(start,length,insert\_one......)函数则是直接在原数组进行删除、添加、替换元素的操作：

start为数组删除元素的开始下标，

length为从start位置开始array删除元素的个数，

后面参数为删除之后array重新插入的数据内容，插入位置为删除位置，而非数组开头或末尾，

返回值为array删除的元素组成的数组。

显而易见，splice函数可以用来对数组元素进行替换。由splice操作后的数组array，数组中内容如果已经改变，就再也找不回array在splice之前的模样。

**25. ★★ 将 1234567 转换为 1,234,567**

function fun(n){

return String(n).replace(/(?!^)(?=(\d{3})+\.)/g, ",")

}

**26. ★★★ bind 的作用是什么？**

bind()方法主要就是将函数绑定到某个对象，

bind()会创建一个函数，函数体内的this对象的值会被绑定到传入bind()第一个参数的值

**27. ★★ Promise.resolve(Promise.resolve(1)).then(console.log) 输出？**

// 答案：1

**28. ★★★ var let const的区别**

1. var声明的变量会挂载在window上，而let和const声明的变量不会

2. var声明变量存在变量提升，let和const不存在变量提升

3. let和const声明形成块作用域

4. 同一作用域下let和const不能声明同名变量，而var可以

5. 使用let/const声明的变量在当前作用域存在暂存死区

6. const一旦声明必须赋值,不能使用null占位,声明后不能再修改,如果声明的是复合类型数据，可以修改其属性

**29. ★★★ document load 和 documen ready的区别**

DOM文档解析：

解析html结构

加载脚本和样式文件

解析并执行脚本

构造html的DOM模型 //ready

加载图片等外部资源文件

页面加载完毕 //load

document load:

load是当页面所有资源全部加载完成后（包括DOM文档树，css文件，js文件，图片资源等），执行一个函数，load方法就是onload事件。

documen ready:

构造html的DOM模型加载完毕后触发

**30. ★★★ 如何自定义事件？**

自定义事件

事件是与DOM交互的最常见的方式。通过实现自定义事件，可以让事件用于非DOM代码中。

思想：创建一个管理事件的对象，让其他对象监听那些事件。

基本模式：

function EventTarget(){

this.handlers = {};

}

EventTarget.prototype = {

constructor:EventTarget,

addHandler:function(type,handler){

if(typeof this.handlers[type] === "undefined"){

this.handlers[type] = [];

}

this.handlers[type].push(handler);

},

fire:function(event){

if(!event.target){

event.target = this;

}

if(this.handlers[event.type] instanceof Array){

const handlers = this.handlers[event.type];

handlers.forEach((handler)=>{

handler(event);

})

}

},

removeHandler:function(type,handler){

if(this.handlers[type] instanceof Array){

const handlers = this.handlers[type];

for(var i = 0,len = handlers.length; i < len; i++){

if(handlers[i] === handler){

break;

}

}

handlers.splice(i,1);

}

}

}

**31. ★★★ 如何用 setTImeout 来实现 setInterval？**

1.不去关心回调函数是否还在运行

在某些情况下，函数可能需要比间隔时间更长的时间去完成执行。比如说是用setInterval每隔5秒对远端服务器进行轮询，网络延迟，服务器无响应以及其他因素将会阻止请求按时按成。结果会导致返回一串无必要的排成队列请求。

2.忽视错误

因为某些原因，setInterval调用的代码中会出现一个错误，但是代码并不会中止执行而是继续执行错误的代码。

3.缺乏灵活性

除了前面提到的缺点之外，我非常希望setInterval方法能有一个表明执行次数的参数而不是无休止的执行下去。

function interval(func, w, t){

var interv = function(){

if(typeof t === "undefined" || t-- > 0){

setTimeout(interv, w);

try{

func.call(null);

}

catch(e){

t = 0;

throw e.toString();

}

}

};

setTimeout(interv, w);

};

**32. ★★★ 如何判断 user 对象里有没有 a 这个属性？如果把user对象中所有的属性都输出出来？**

(var user = {'a': '19', 'b': '18', 'c': '16'})

如何判断 user 对象里有没有 a 这个属性？

js对象的Object.hasOwnProperty()方法

返回一个布尔值，判断对象是否包含特定的自身（非继承）属性。

let obj = new Object();

obj.a = "123";

console.log(obj.hasOwnProperty('a')) // true

console.log(obj.hasOwnProperty('b')) // false

把user对象中所有的属性都输出出来

for(item for user){

console.log(item)

}

**33. ★★ 使用 setTimeout 模拟 setInterval 的功能做一个60秒的倒数计时**

function setInter(s,fn){

let timeOut = (s,fn)=>{

setTimeout(()=>{

fn();

timeOut(s,fn);

},s)

}

timeOut(s,fn);

}

//调用上面的方法

setInter(60000,()=>{console.log("hello world!")})

**34. ★★★ 实现一个函数 add()，运算结果可以满足如下预期结果**

function add () {

var args = Array.prototype.slice.call(arguments);

var fn = function () {

var sub\_arg = Array.prototype.slice.call(arguments);

　　 // 把全部的参数聚集到参数的入口为一个参数： args.concat(sub\_arg)

return add.apply(null, args.concat(sub\_arg));

}

fn.valueOf = function () {

return args.reduce(function(a, b) {

return a + b;

})

}

return fn;

}

add(1,2,3)(10) //16

add(1)(2)(3,4)(5) //15

**35. ★★★ 如何避免回调地狱？**

1. Promise 对象就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的嵌套，改成链式调用。

promise只有两个状态resolve和reject，当它触发任何一个状态后，它会将当前的值缓存起来，并在有回调函数添加进来的时候尝试调用回调函数，如果这个时候还没有触发resolve或者reject，那么回调函数会被缓存，等待调用，如果已经有了状态(resolve或者reject)，则立刻调用回调函数。并且所有回调函数在执行后都立即被销毁。

2. ES6 co/yield方案

yield: Generator 函数是协程在 ES6 的实现，而yield是 Generator关键字， 异步操作需要暂停的地方，都用yield语句注明。

co: co 模块是著名程序员 TJ Holowaychuk 于 2013 年 6 月发布的一个小工具，用于 Generator 函数的自动执行。

3. ES7 async/await 方案

async/await是es7的新标准，并且在node7.0中已经得到支持。

它就是 Generator 函数的语法糖，async函数就是将 Generator 函数的星号（\*）替换成async，将yield替换成await，仅此而已。可以理解官方对co和Generator 封装方案。

**36. ★★ 写一个 function，清除字符串前后的空格。（兼容所有的浏览器）**

function trim(str) {

if (str && typeof str === "string") {

return str.replace(/(^\s\*)|(\s\*)$/g,""); //去除前后空白符

}

}

**37. ★★ 使用正则表达式验证邮箱格式**

function fChkMail(emailAddress){

var reg = new RegExp("^[a-z0-9]+([.\_\\-]\*[a-z0-9])\*@([a-z0-9]+[-a-z0-9]\*[a-z0-9]+.){1,63}[a-z0-9]+$");

var bChk=reg.test(emailAddress);

return bChk;

}

**38. ★★★ 简述同步和异步的区别**

同步：

同步的思想是：所有的操作都做完，才返回给用户。这样用户在线等待的时间太长，给用户一种卡死了的感觉（就是系统迁移中，点击了迁移，界面就不动了，但是程序还在执行，卡死了的感觉）。这种情况下，用户不能关闭界面，如果关闭了，即迁移程序就中断了。

异步：

将用户请求放入消息队列，并反馈给用户，系统迁移程序已经启动，你可以关闭浏览器了。然后程序再慢慢地去写入数据库去。这就是异步。但是用户没有卡死的感觉，会告诉你，你的请求系统已经响应了。你可以关闭界面了。

同步和异步本身是相对的:

同步就相当于是 当客户端发送请求给服务端，在等待服务端响应的请求时，客户端不做其他的事情。当服务端做完了才返回到客户端。这样的话客户端需要一直等待。用户使用起来会有不友好。

异步就是，当客户端发送给服务端请求时，在等待服务端响应的时候，客户端可以做其他的事情，这样节约了时间，提高了效率。

存在就有其道理 异步虽然好 但是有些问题是要用同步用来解决，比如有些东西我们需要的是拿到返回的数据在进行操作的。这些是异步所无法解决的。

**39. ★★ JavaScript 中 callee 和 caller 的作用**

1.callee

callee是对象的一个属性，该属性是一个指针，指向参数arguments对象的函数

作用：就是用来指向当前对象

返回正被执行的 Function 对象，也就是所指定的 Function 对象的正文.

callee是arguments 的一个属性成员，它表示对函数对象本身的引用，这有利于匿名

函数的递归或者保证函数的封装性

2.caller

caller是函数对象的一个属性，该属性保存着调用当前函数的函数的引用（指向当前函数的直接父函数）

返回一个对函数的引用，该函数调用了当前函数。

functionName.caller

functionName 对象是所执行函数的名称。

注意：

对于函数来说，caller 属性只有在函数执行时才有定义。 如果函数是由 Javascript 程序的顶层调用的，那么 caller 包含的就是 null 。

**40. ★★ 统计字符串中字母个数或统计最多的字母数**

function count(str) {

var obj = {}; // 统计对象

var i = 0;

var len = str.length;

for (; i < len; i++){

var curChar = str.charAt(i);

// 如果结果对象存在该字符的属性，则自增，否则置为1

if (obj[curChar]) {

obj[curChar]++;

} else {

obj[curChar] = 1;

}

}

// 返回结果

return obj;

}

var str = "javaScript";

console.log(count(str));

**41. ★★★ jQuery 的事件委托方法 on，live，delegate之间有区别？**

live 把事件委托交给了document（根节点），document 向下去寻找符合条件的元素（）， 不用等待document加载结束也可以生效。

delegate可指定事件委托对象，相比于live性能更优，直接锁定指定选择器；

on事件委托对象选填，如果不填，即给对象自身注册事件，填了作用和delegate一致。

**42. ★★★ 简述下 Promise 对象**

Promise是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理更强大。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件 (通常是一个异步操作)的结果。从语法上说，Promise是一个对象，从它可以获取异步操作的消息。

Promise对象有以下2个特点：

1.对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态：Pending(进行中)、Resolved(已完成)和Rejected(已失败)。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

2.一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved；从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对Promise对象田静回调函数，也会立即得到这个结果。这与事件(Event)完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了Promise对象，就可以把异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供了统一的接口，使得控制异步操作更加容易。

**43. ★★★ 数组扁平化，不用 api**

function myFlat(arr){

let res = [];

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

if(arr[i] instanceof Array){

res = res.concat(myFlat(arr[i]));

}else {

res.push(arr[i]);

}

}

return res;

}

let arr = [1,[2,3,[4,5]]];

console.log(myFlat(arr))

**44. ★★★ 用 JavaScript 实现观察者模式**

function BusinessOne(name){

this.name = name;

//订阅者的集合

this.subscribers = new Array();

}

//订阅者的发送消息的方法(推模式)

BusinessOne.prototype.delive = function(news){

var self = this;

//给每一个订阅者发送消息

this.subscribers.forEach(

function(fn){

//调用接受者处理信息的函数

fn(news,self);

}

)

}

//扩展公共订阅的函数,和取消订阅的函数

Function.prototype.subscribe = function(publisher){

var that = this;

//some 访问数组度i型并且以参数的形式传回回调函数中

//只要至少有一次返回是true那么some就是true

var alreadyExists = publisher.subscribers.some(

function(el){

//处理不能重复订阅的功能

if(el == that){

return;

}

}

);

//没用订阅你就可以订阅

if(!alreadyExists){

publisher.subscribers.push(that);

}

return this;

}

//取消

Function.prototype.unsubscribe = function(publisher){

var that = this;

publisher.subscribers = publisher.subscribers.filter(

function(el){

if(el !== that){

return el;

}

}

);

return this;

};

**45. ★★ 简述一下面象对象的六法则**

1. 单一职责原则：一个类只做它该做的事情

2. 开闭原则：软件实体应当对扩展开放，对修改关闭

3. 依赖倒转原则：面向接口编程

4. 接口隔离原则：接口要小而专，绝不能大而全

5. 合成聚合复用原则：优先使用聚合或合成关系复用代码

6. 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解(低耦合)

**46. ★★★ 谈谈垃圾回收机制方法以及内存管理**

垃圾回收方式

① 标记清除

工作原理：是当变量进入环境时，将这个变量标记为“进入环境”。当变量离开环境时，则将其标记为“离开环境”。标记“离开环境”的就回收内存。

② 引用计数

工作原理：跟踪记录每个值被引用的次数。一旦没有引用，内存就直接释放了。

内存管理

什么时候触发垃圾回收？

垃圾回收器周期性运行，如果分配的内存非常多，那么回收工作也会很艰巨，确定垃圾回收时间间隔就变成了一个值得思考的问题。

1、合理的GC方案：(1)、遍历所有可访问的对象; (2)、回收已不可访问的对象。

2、GC缺陷： (1)、停止响应其他操作；

3、GC优化策略： (1)、分代回收（Generation GC）;(2)、增量GC

**47. ★★★ 开发过程中遇到内存泄漏的问题都有哪些？**

1. 当页面中元素被移除或替换时，若元素绑定的事件仍没被移除，在IE中不会作出恰当处理，此时要先手工移除事件，不然会存在内存泄露。

2. 由于是函数内定义函数，并且内部函数--事件回调的引用外暴了，形成了闭包。闭包可以维持函数内局部变量，使其得不到释放。

**48. ★★★ 请编写获取当前窗口地址中查询参数name的值，当前窗口地址为：[https://foo.com/?id=1&name=tom](https://foo.com/?id=1&name=tom" \t "_blank)**

function GetQueryString(name){

var reg = new RegExp("(^|&)"+ name +"=([^&]\*)(&|$)");

var r = window.location.search.substr(1).match(reg);

if(r!=null)

return unescape(r[2]);

return null;

}

**49. ★★★ 已知a，b两个构造函数，现在 let c = new a()，如何在c的存储地址不变的情况下，改变c的继承（c->a 转为 c->b）**

1. 改变原型链：通过改变C的prototype为b,实现内存地址不动，改变继承

**50. ★★★ 浏览器有哪些兼容问题，你封装过什么插件**

//1.滚动条到顶端的距离（滚动高度）

var scrollTop = document.documentElement.scrollTop || document.body.scrollTop;

//2.滚动条到左端的距离

var scrollLeft = document.documentElement.scrollLeft || document.body.scrollLeft;

//3. IE9以下byClassName

function byClassName(obj,className){

//判断是否支持byClassName

if(obj.getElementsByClassName){

//支持

return obj.getElementsByClassName(className);

}else{

//不支持

var eles = obj.getElementsByTagName('\*'); //获取所有的标签

var arr = []; //空数组，准备放置找到的对象

//遍历所有的标签

for(var i = 0,len = eles.length;i < len;i ++){

//找出与我指定class名相同的对象

if(eles[i].className === className){

arr.push(eles[i]); //存入数组

}

}

return arr; //返回

}

}

//4. 获取非行内样式兼容 IE:currentStyle 标准：getComputedStyle

function getStyle(obj,attr){

return window.getComputedStyle ? getComputedStyle(obj,true)[attr] : obj.currentStyle[attr];

}

//div.style.width = '';设置样式

//obj['属性']： 对象是变量时，必须用对象['属性']获取。

//5. 获取事件对象的兼容

evt = evt || window.event

//6. 获取鼠标编码值的兼容

function getButton(evt){

var e = evt || window.event;

if(evt){

return e.button;

}else if(window.event){

switch(e.button){

case 1 : return 0;

case 4 : return 1;

case 2 : return 2;

}

}

}

//7. 获取键盘按键编码值的兼容

var key = evt.keyCode || evt.charCode || evt.which;

//8. 阻止事件冒泡的兼容

e.stopPropagation ? e.stopPropagation() : e.cancelBubble = true;

//9. 阻止超链接的默认行为的兼容

evt.preventDefault ? evt.preventDefault() : evt.returnValue = false;

//10. 添加事件监听器的兼容

function addEventListener(obj,event,fn,boo){

if(obj.addEventListener){

obj.addEventListener(event,fn,boo);

}else if(obj.attachEvent){

obj.attachEvent('on' + event,fn);

}

}

//11. 移除事件监听器的兼容

function removeEventListener(obj,event,fn,boo){

if(obj.removeEventListener){

obj.removeEventListener(event,fn,boo);

}else if(obj.detachEvent){

obj.detachEvent('on' + event,fn);

}

}

//12. 获取事件源的兼容

var target = event.target || event.srcElement;

**51. ★★★ 如何判断一个对象是否为数组，函数**

方法一： instanceof:

var arr=[];

console.log(arr instanceof Array) //返回true

方法二： constructor:

console.log(arr.constructor == Array); //返回true

方法三： Array.isArray()

console.log(Array.isArray(arr)); //返回true

**52. ★★★ 写一个函数，接受可变个数参数，且每个参数均为数字，返回参数的最大值**

function myMax(){

return Math.max(arguments)

}

**53. ★★★ 请写出 ES6 Array.isArray()**

if (!Array.isArray){

Array.isArray = function(arg){

return Object.prototype.toString.call(arg) === '[object Array]';

};

}

**54. ★★★ 实现一个函数 clone，可以对 JavaScript 中的5种主要数据类型进行值复制**

// 方法一：

Object.prototype.clone = function() {

var o = this.constructor === Array ? [] : {};

for (var e in this) {

o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];

}

return o;

};

//方法二：

/\*\*

\* 克隆一个对象

\* @param Obj

\* @returns

\*/

function clone(Obj) {

var buf;

if (Obj instanceof Array) {

buf = []; //创建一个空的数组

var i = Obj.length;

while (i--) {

buf[i] = clone(Obj[i]);

}

return buf;

} else if (Obj instanceof Object) {

buf = {}; //创建一个空对象

for (var k in Obj) {

//为这个对象添加新的属性

buf[k] = clone(Obj[k]);

}

return buf;

} else {

//普通变量直接赋值

return Obj;

}

}

**55. ★★★ 假如A页面我定义了一个定时器，然后跳到B页面如果让A页面的定时器暂停**

方法1：在beforeDestroy()等生命周期结束阶段内清除定时器：

beforeDestroy() {

clearInterval(this.timer);

this.timer = null;

}

方法2：通过$once这个事件侦听器器在定义完定时器之后的位置来清除定时器。

const timer = setInterval(() =>{

// 某些定时器操作

}, 500);

// 通过$once来监听定时器，在beforeDestroy钩子可以被清除。

this.$once('hook:beforeDestroy', () => {

clearInterval(timer);

})

**56. ★★★ promise的实现原理，如果我现在向服务器发送一个请求，但是我后悔了，不想让服务器返回数据，去实现一个delay**

取消结束Promise的方法？

1. 返回一个pending状态的Promise，原Promise链会终止

Promise.resolve().then(() => {

console.log('ok1')

return new Promise(()=>{}) // 返回“pending”状态的Promise对象

}).then(() => {

// 后续的函数不会被调用

console.log('ok2')

}).catch(err => {

console.log('err->', err)

})

2. Promise.race竞速方法

let p1 = new Promise((resolve, reject) => {

resolve('ok1')

})

let p2 = new Promise((resolve, reject) => {

setTimeout(() => {resolve('ok2')}, 10)

})

Promise.race([p2, p1]).then((result) => {

console.log(result) //ok1

}).catch((error) => {

console.log(error)

})

3. 当Promise链中抛出错误时，错误信息沿着链路向后传递，直至捕获

Promise.resolve().then(() => {

console.log('ok1')

throw 'throw error1'

}).then(() => {

console.log('ok2')

}, err => {

// 捕获错误

console.log('err->', err)

}).then(() => {

// 该函数将被调用

console.log('ok3')

throw 'throw error3'

}).then(() => {

// 错误捕获前的函数不会被调用

console.log('ok4')

}).catch(err => {

console.log('err->', err)

Axios如何取消请求？

第一种通过CancelToken.source工厂方法创建cancel token

var CancelToken = axios.CancelToken;

var source = CancelToken.source();

axios.get('/user/12345', {

cancelToken: source.token

}).catch(function(thrown) {

if (axios.isCancel(thrown)) {

console.log('Request canceled', thrown.message);

} else {

// 处理错误

}

});

// 取消请求（message 参数是可选的）

source.cancel('Operation canceled by the user.');

第二种通过传递executor函数到CancelToken的构造函数来创建cancel token

var CancelToken = axios.CancelToken;

var cancel;

axios.get('/user/12345', {

cancelToken: new CancelToken(function executor(c) {

// executor 函数接收一个 cancel 函数作为参数

cancel = c;

})

});

// 取消请求

cancel();

**57. ★★★ CommonJS 和 RequireJS 的实现原理**

commonjs是通过module.exports导出模块,用require引入一个模块，原理：闭包

requirejs是通过define定义导出模块，用require引入模块。

define('name',[],function(){

return 'requirejs'

})

define('say',['name'].function(name){

return "my name is" + name

})

require(['say'],function(text){

console.log(text)

})

**58. ★★★ 面向对象编程与面向过程编程的区别？**

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

**59. ★★★ eval 是做什么的？性能怎么样？安全如何？**

它的功能是把对应的字符串解析成js代码并运行，

应该避免使用eval,因为不安全，非常耗性能（2次，一次解析成js语句，一次执行）

注意：在项目里写js代码的时候，禁止使用的，因为有安全因素。

**60. ★★★★★ 函数节流、防抖。scroll resize 使用函数节流实现不要频繁触发事件的需求**

// 答案：

防抖：

//scroll方法中的do somthing至少间隔500毫秒执行一次

window.addEventListener('scroll',function(){

var timer;//使用闭包，缓存变量

return function(){

if(timer) clearTimeout(timer);

timer = setTimeout(function(){

console.log('do somthing')

},500)

}

}());//此处()作用 - 立即调用return后面函数，形成闭包

节流：

//scroll方法中当间隔时间大于2s，do somthing执行一次

window.addEventListener('scroll',function(){

var timer ;//使用闭包，缓存变量

var startTime = new Date();

return function(){

var curTime = new Date();

if(curTime - startTime >= 2000){

timer = setTimeout(function(){

console.log('do somthing')

},500);

startTime = curTime;

}

}

}());//此处()作用 - 立即调用return后面函数，形成闭包

**61. ★★★ 数据类型（判断，==和===）堆栈、内存泄漏及垃圾回收机制**

// 答案：

1，== 判断值是否相等； === 判断值和数据类型是否严格相等

2，Javascript堆栈和垃圾回收机制

堆栈溢出

​ 当储存的数据导到某一限制时就会造成堆栈溢出

内存泄漏

​ 当不断向堆中存储数据，而不进行清理，这就是内存泄漏

垃圾回收机制（清除孤儿机制）

​ 语言当中一般分两种，一种是自动清理，一种是手动清理（GC），js中只有自动清理

​ 垃圾回收机制就是将引用对中的地址的对象设置为null，并且将所有引用该地址的对象都设置为null，并且移除事件侦听

​ 不会即时清除,垃圾回收车会根据内存的情况在适当的时候进行清除堆中的对象 内存到达一定程度了才会进行回收

**62. ★★★★ 了解 ES6 的 Proxy 吗？**

// 答案：

Proxy，代理，是ES6新增的功能，可以理解为代理器（即由它代理某些操作）。

Proxy 对象用于定义或修改某些操作的自定义行为，可以在外界对目标对象进行访问前，对外界的访问进行改写。

new Proxy()表示生成一个 Proxy 实例

-target：目标对象

-handler：一个对象，其属性是当执行一个操作时定义代理的行为的函数。

注意：要实现拦截操作，必须是对 Proxy 实例进行操作，而不是针对目标对象 target 进行操作。

**63. ★★★★ 深拷贝是什么？项目哪里是用到了深拷贝？**

// 答案：

1，在拷贝构造函数中假如只完成了数据成员本身的赋值则称为“浅拷贝”；编译器提供的默认拷贝构造函数就已经可以完成这个任务。

而假如要复制的数据除了属性值本身以外，还要复制附加在数据属性值上的额外内容，那就要自己来写拷贝构造函数了，来完成所谓的“深拷贝”。

举个例子：

若在构造函数中new了一个新的空间存放数据，并且用指针记录了首地址；若是浅拷贝，则在拷贝构造函数中指针值将复制给另一个数据成员，这样就会有两个指针指向同一个空间；这样的话在析构函数里将会对指针所指向的空间进行释放，由于两个指针指向的是同一个空间，在释放第一个指针指向的空间时不会出现什么问题，而释放第二个指针指向的空间时就会因为空间已经被解析过而导致解析的空间不存在的情况，就会造成程序无法终止。

而解决上面这种情况的办法就是使用“深拷贝”，深拷贝是在拷贝构造函数里再new一个新的空间。将数据复制在新空间里，并将拷贝的指针记录这个新空间的首地址，这样在析构函数里就不会有问题了。

2，在某些引用类型值不更新的情况下用深拷贝

**64. ★★★ swiper 插件从后台获取数据没问题，css 代码啥的也没问题，但是图片不动，应该怎么解决？**

// 答案：

主要原因：

swiper提前初始化了，而这个时候，数据还没有完全出来。

解决方法

从swiper 入手，在swiper中写 observer:true/observeParents:true

let myswiper = new Swiper(".swiper-container" , {

autoplay: true,

loop: true,

// observer 修改swiper子元素时自动初始化swiper

observer:true,

// observeParents 包括当前父元素的swiper发生变更时也会初始化swiper

observeParents:true,

})

从 Vue 入手，vue中专门提供了提供了一个方法nextTick() 用于解决dom的先后执行问题。

mounted(){

this.$nextTick(function(){

// ...操作

let myswiper = new Swiper(".swiper-container" , {

autoplay: true,

loop: true

})

})

}

**65. ★★★★ ES6 中，数组监测怎么实现的（代理）**

// 通过ES6的关键字extends实现继承完成Array原型方法的重写

class NewArray extends Array {

constructor(...args) {

// 调用父类Array的constructor()

super(...args)

}

push (...args) {

console.log('监听到数组的变化啦！');

// 调用父类原型push方法

return super.push(...args)

}

// ...

}

let list3 = [1, 2];

let arr = new NewArray(...list3);

console.log(arr)

// (2) [1, 2]

arr.push(3);

// 监听到数组的变化啦！

console.log(arr)

// (3) [1, 2, 3]

**66. ★★ jQuery 优点和缺点**

// 答案：

优点

1.出色的浏览器兼容性

2、出色的DOM操作的封装，使他具备强大的选择器，可以进行快速的DOM元素操作

3、可靠的事件处理机制、jq在处理事件绑定的时候是相当的可靠

4、完善的ajax（对ajax的封装非常好，不需要考虑复杂的浏览器的兼容和XMLhttprequest对象的创建和使用）

5、支持链式操作（什么是链式操作？通过‘.’来操作）和隐士迭代

6、减少服务器的压力和带宽并且加快了加载速度（为什么这么说？原因就是：当你打开网页之前打开了其他的网页，并且该网页也用了cdn的方式来

加载相同版本的jq文件，那么，浏览器就不会加载第二次，为啥舍近求远呢，和生活中的道理一样一样的！）

7、支持丰富的插件，当然你也可以自定义插件，再加上jq的文档也很丰富，对于程序员来说，是一件非常美好的事情

缺点

1.不能向后兼容。

每一个新版本不能兼容早期的版本。举例来说，有些新版本不再支持某些selector，新版jQuery却没有保留对它们的支持，而只是简单的将其移除。这可能会影响到开发者已经编写好的代码或插件。

2.插件兼容性。

与上一点类似，当新版jQuery推出后，如果开发者想升级的话，要看插件作者是否支持。通常情况下，在最新版jQuery版本下，现有插件可能无法正常使用。开发者使用的插件越多，这种情况发生的几率也越高。我有一次为了升级到jQuery 1.3，不得不自己动手修改了一个第三方插件。

3.多个插件冲突。

在同一页面上使用多个插件时，很容易碰到冲突现象，尤其是这些插件依赖相同事件或selector时最为明显。这虽然不是jQuery自身的问题，但却又确实是一个难于调试和解决的问题。

4.jQuery的稳定性。

jQuery没有让浏览器崩溃，这里指的是其版本发布策略。jQuery 1.3版发布后仅过数天，就发布了一个漏洞修正版1.3.1。他们还移除了对某些功能的支持，可能会影响许多代码的正常运行。我希望类似修改不要再出现。

5.对动画和特效的支持差。

在大型框架中，jQuery核心代码库对动画和特效的支持相对较差。但是实际上这不是一个问题。目前在这方面有一个单独的jQuery UI项目和众多插件来弥补此点。

**67. ★★★ ES6 class 关键字原理跟 function 什么区别？**

// 答案：

function 可以用call apply bind 的方式 来改变他的执行上下文

但是class 却不可以 class 虽然本质上也是一个函数 但是 其内（babel）部做了一层代理 来禁止了这种行为

关于构造器constructor

在function定义的构造函数中，其prototype.constructor属性指向构造器自身

在class定义的类中，constructor其实也相当于定义在prototype属性上

重复定义

function会覆盖之前定义的方法

class会报错

原型或者类中方法的枚举

class中定义的方法不可用Object.keys(Point.prototype)枚举到

function构造器原型方法可被Object.keys(Point.prototype)枚举到，除过constructor

所有原型方法属性都可用Object.getOwnPropertyNames(Point.prototype)访问到

**68. ★★★ iframe 跨域问题，页面之间怎么传值？**

// 答案：

一般有两个解决方案，一个是建立一个代理页面，通过代理页面传值，

另一个方法是通过H5的postMessage方法传值，今天用的是第二种。

首先，在父页面A中建立一个iframe，其中src要写好子页面B的地址，然后在A页面中写如下方法：

var iframe = document.getElementById("onemap");

var msg = {loginName:'arcgis',loginPassword:'Esri1234'};

var childDomain = "https://geoplat.training.com";

iframe.contentWindow.postMessage(msg,childDomain);

记住，childDomain与A的iframe的src地址不一样，childDomain是域，而src是域中的一个页面

msg是传输的信息，可以是字符串，也可以是对象。

上面的方法一定要写在一个函数中，并通过点击事件调用，如果希望iframe开始为空，点击后在设置src，

可以在设置src之后，通过setTimeout设置一定时间后在传输信息。

在子页面B中，通过对window添加事件获取传输过来的信息：

window.addEventListener("message",function(obj){

var name = obj.data.loginName;

var password = obj.data.loginPassword;

login.iframeChildLogin(name,password);

},false);

这样就完成了从不同域的父页面向子页面传值的过程

**69. ★★★ 简述 commonJS、AMD 和 CMD**

// 答案：

CommonJS导出模块的方法是exports，导入模块的是require，具体规范如下

1）如果一个JS文件中存在exports或require，该JS文件是一个模块

2）模块内的所有代码均为隐藏代码，包括全局变量、全局函数，这些全局的内容均不应该对全局变量造成任何污染

3）如果一个模块需要暴露一些API提供给外部使用，需要通过exports导出，exports是一个空的对象，你可以为该对象添加任何需要导出的内容

4）如果一个模块需要导入其他模块，通过require实现，require是一个函数，传入模块的路径即可返回该模块导出的整个内容

【注】CommonJS只是一个规范，相当于告诉你按什么标准制造汽车，但是具体怎么制造还是得看生产商。因此，有了规范以后，nodejs就去实现模块化了

AMD

AMD 是 RequireJS 在推广过程中对模块定义的规范化产出。

AMD 推崇依赖前置。

CMD

CMD 是 SeaJS 在推广过程中对模块定义的规范化产出。

CMD 推崇依赖就近

**70. ★★★ require.js 源码看过吗？怎么做到异步加载的**

/\*\*

\* Creates the node for the load command. Only used in browser envs.

\*/

req.createNode = function (config, moduleName, url) {

var node = config.xhtml ?

document.createElementNS('http://www.w3.org/1999/xhtml', 'html:script') :

document.createElement('script');

node.type = config.scriptType || 'text/javascript';

node.charset = 'utf-8';

node.async = true;

return node;

};

　　requirejs 导入模块的方式实际就是创建脚本标签，一切的模块都需要经过这个方法创建。requirejs 使用 onload 事件来处理回调函数：

**71. ★★ jQuery，$() 能传什么参数? html 代码怎么解析? 传 function 呢?**

这个函数接收一个包含 CSS 选择器的字符串，然后用这个字符串去匹配一组元素。

jQuery 的核心功能都是通过这个函数实现的。

jQuery中的一切都基于这个函数，或者说都是在以某种方式使用这个函数。这个函数最基本的用法就是向它传递一个表达式（通常由 CSS 选择器组成），然后根据这个表达式来查找所有匹配的元素。

默认情况下, 如果没有指定context参数，$()将在当前的 HTML document中查找 DOM 元素；如果指定了 context 参数，如一个 DOM 元素集或 jQuery 对象，那就会在这个 context 中查找。在jQuery 1.3.2以后，其返回的元素顺序等同于在context中出现的先后顺序。

**72. ★★ AMD 怎么加载文件的?**

AMD 即Asynchronous Module Definition，中文名是异步模块定义的意思。它是一个在浏览器端模块化开发的规范

由于不是JavaScript原生支持，使用AMD规范进行页面开发需要用到对应的库函数，也就是大名鼎鼎RequireJS，实际上AMD 是 RequireJS 在推广过程中对模块定义的规范化的产出

requireJS主要解决两个问题

多个js文件可能有依赖关系，被依赖的文件需要早于依赖它的文件加载到浏览器

js加载的时候浏览器会停止页面渲染，加载文件越多，页面失去响应时间越长

看一个使用requireJS的例子

// 定义模块 myModule.js

define(['dependency'], function(){

var name = 'Byron';

function printName(){

console.log(name);

}

return {

printName: printName

};

});

// 加载模块

require(['myModule'], function (my){

　 my.printName();

});

语法

requireJS定义了一个函数 define，它是全局变量，用来定义模块

define(id?, dependencies?, factory);

id：可选参数，用来定义模块的标识，如果没有提供该参数，脚本文件名（去掉拓展名）

dependencies：是一个当前模块依赖的模块名称数组

factory：工厂方法，模块初始化要执行的函数或对象。如果为函数，它应该只被执行一次。如果是对象，此对象应该为模块的输出值

在页面上使用require函数加载模块

require([dependencies], function(){});

require()函数接受两个参数

第一个参数是一个数组，表示所依赖的模块

第二个参数是一个回调函数，当前面指定的模块都加载成功后，它将被调用。加载的模块会以参数形式传入该函数，从而在回调函数内部就可以使用这些模块

require()函数在加载依赖的函数的时候是异步加载的，这样浏览器不会失去响应，它指定的回调函数，只有前面的模块都加载成功后，才会运行，解决了依赖性的问题。

**73. ★★ jQuery 怎么找到事件源元素**

$(".btn").click(function(e){

// e 就是事件对象

e.target; // 事件的目标 dom

e.currentTarget; // 事件处理程序正在处理事件的那个元素

e.srcElement; // 事件的目标 ie

});

**74. ★★★★ 模板引擎原理**

模板引擎是通过字符串拼接得到的

let template = 'hello <% name %>!'

let template = 'hello ' + name + '!'

字符串是通过new Function执行的

let name = 'world'

let template = `

let str = 'hello ' + name + '!'

return str

`

let fn = new Function('name', template)

console.log(fn(name)) // hello world!

将模板转换为字符串并通过函数执行返回

let template = 'hello <% name %>!'

let name = 'world'

function compile (template) {

let html = template.replace(/<%([\s\S]+?)%>/g, (match, code) => {

return `' + ${code} + '`

})

html = `let str = '${html}'; return str`

return new Function('name', html)

}

let str = compile(template)

console.log(str(name)) // hello world!

函数只能接收一个name变量作为参数，功能太单一了，一般会通过对象来传参，with来减少变量访问。

with功能

let params = {

name: '张三',

age: 18

}

let str = ''

with (params) {

str = `用户${name}的年龄是${age}岁`

}

console.log(str) // 用户张三的年龄是18岁

实现简单的模板引擎

let template = 'hello <% name %>!'

let name = 'world'

function compile (template) {

let html = template.replace(/<%([\s\S]+?)%>/g, (match, code) => {

return `' + ${code.trim()} + '`

})

html = `'${html}'`

html = `let str = ''; with (params) { str = ${html}; } return str`

return new Function('params', html)

}

let str = compile(template)

console.log(str({ name })) // hello world!

**75. ★★ map 和 foreach 的区别**

forEach()方法不会返回执行结果，而是undefined。也就是说，forEach()会修改原来的数组。

map()方法会得到一个新的数组并返回。

**76. ★★★★ ES6 的新特性**

1. const与let

2. 模板字符串

3. 解构赋值

4. 对象简写法

5. for...of循环

6. 展开运算符

7. 剩余参数(可变参数)

8. ES6箭头函数

9. 参数默认值

10.类和继承

11.模块化规范

**77. ★★ 2018/01/01 转换成 2018年/1月/1日**

function fun(str){

var date = new Date(str)

return date.getFullYear()+'年/'+date.getMonth()+'月/'+date.getDate()+'日'

}

**78. ★★★ 0.1+0.2 等不等于 0.3？自己封装一个让他们相等的方法**

在正常的数学逻辑思维中，0.1+0.2=0.3这个逻辑是正确的，但是在JavaScript中0.1+0.2！==0.3，这是为什么呢？这个问题也会偶尔被用来当做面试题来考查面试者对JavaScript的数值的理解程度。

　　在JavaScript中的二进制的浮点数0.1和0.2并不是十分精确，在他们相加的结果并非正好等于0.3，而是一个比较接近的数字 0.30000000000000004 ，所以条件判断结果为false。

方法1：设置一个误差范围值，通常称为”机器精度“，而对于Javascript来说，这个值通常是2^-52,而在ES6中，已经为我们提供了这样一个属性：Number.EPSILON，而这个值正等于2^-52。这个值非常非常小，在底层计算机已经帮我们运算好，并且无限接近0，但不等于0,。这个时候我们只要判断(0.1+0.2)-0.3小于Number.EPSILON，在这个误差的范围内就可以判定0.1+0.2===0.3为true。

function numbersequal(a,b){

return Math.abs(a-b)<Number.EPSILON;

}

方法2：转为整数运算

**79. ★★★ 跨域是什么？有哪些解决跨域的方法和方案？**

什么是跨域?

所谓的同源是指，域名、协议、端口均为相同。

所谓的跨域，不同的域名、协议、端口皆为不同域

一个域与另一个域名、协议或者端口不同的域的之间访问都叫跨域

解决跨域的方法和方案：

1：通过服务端代理请求。如PHP，服务端语言php是没有跨域限制的，让服务器去别的网站获取内容然后返回给页面。

2：第二种：jsonp跨域

1. jsonp跨域就是利用script标签的跨域能力请求资源

2. jsonp与ajax没有半毛钱关系！！

3. 浏览器的同源策略限制了js的跨域能力，但没有限制link img iframe script 的跨域行为

实现方式：

1. 利用js创建一个script标签，把json的url赋给script的scr属性，

2. 把这个script插入到页面里，让浏览器去跨域获取资源

3. JS先声明好回调函数，插入页面后会代为执行该函数，并且传入json对象为其参数。

注意：

1. jsonp只针对get请求

2. script标签加载回来的资源会被当成js在全局执行

3：CORS 跨域资源共享(xhr2)

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）

它允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制

整个CORS通信过程，都是浏览器自动完成，不需要用户参与

对于开发者来说，CORS通信与同源的AJAX通信没有差别，代码完全一样

实现CORS通信的关键是服务器，只要服务器实现了CORS接口，就可以跨源通信

4：nginx代理跨域

通过nginx服务器转发跨域请求，达到跨域的目的

**80. ★★★ 什么是函数式编程？什么的声明式编程？**

函数式编程：

函数式编程和声明式编程是有所关联的，因为他们思想是一致的：即只关注做什么而不是怎么做。但函数式编程不仅仅局限于声明式编程。

函数式编程最重要的特点是“函数第一位”，即函数可以出现在任何地方，比如你可以把函数作为参数传递给另一个函数，不仅如此你还可以将函数作为返回值。

声明式编程：

声明式编程是以数据结构的形式来表达程序执行的逻辑。它的主要思想是告诉计算机应该做什么，但不指定具体要怎么做。

SQL 语句就是最明显的一种声明式编程的例子，例如：

SELECT \* FROM collection WHERE num > 5

除了 SQL，网页编程中用到的 HTML 和 CSS 也都属于声明式编程。

特点：

1：是它不需要创建变量用来存储数据。

2：不包含循环控制的代码如 for， while。

**81. ★★★ super() 是否必须执行？不执行怎么让它不报错？**

非必须，

在 JavaScript 中，super 指的是父类的构造函数

如果想在构造函数中使用this，你必须首先调用super。 先让父类做完自己的事

不执行无法使用this，

不报错的方法：

1：不使用this

2：手动修正this

**82. ★★★ eventloop 渲染在哪一步？**

任务队列

所有的任务可以分为同步任务和异步任务，同步任务，顾名思义，就是立即执行的任务，同步任务一般会直接进入到主线程中执行；而异步任务，就是异步执行的任务，比如ajax网络请求，setTimeout 定时函数等都属于异步任务，异步任务会通过任务队列( Event Queue )的机制来进行协调。

同步和异步任务分别进入不同的执行环境，同步的进入主线程，即主执行栈，异步的进入 Event Queue 。主线程内的任务执行完毕为空，会去 Event Queue 读取对应的任务，推入主线程执行。 上述过程的不断重复就是我们说的 Event Loop (事件循环)。

在事件循环中，每进行一次循环操作称为tick，通过阅读规范可知，每一次 tick 的任务处理模型是比较复杂的，其关键的步骤可以总结如下：

在此次 tick 中选择最先进入队列的任务( oldest task )，如果有则执行(一次)

检查是否存在 Microtasks ，如果存在则不停地执行，直至清空Microtask Queue

更新 render

主线程重复执行上述步骤

那么，什么是 microtasks ?规范中规定，task分为两大类, 分别是 Macro Task （宏任务）和 Micro Task（微任务）, 并且每个宏任务结束后, 都要清空所有的微任务,这里的 Macro Task也是我们常说的 task 。

(macro)task 主要包含：script( 整体代码)、setTimeout、setInterval、I/O、UI 交互事件、setImmediate(Node.js 环境)

microtask主要包含：Promise、MutaionObserver、process.nextTick(Node.js 环境)

整体 script 作为第一个宏任务进入主线程，遇到 console.log，输出 script start

遇到 setTimeout，其回调函数被分发到宏任务 Event Queue 中

遇到 Promise，其 then函数被分到到微任务 Event Queue 中,记为 then1，之后又遇到了 then 函数，将其分到微任务 Event Queue 中，记为 then2

遇到 console.log，输出 script end

至此，Event Queue 中存在三个任务，如下表：

宏任务 微任务

setTimeout then1

- then2

执行微任务，首先执行then1，输出 promise1, 然后执行 then2，输出 promise2，这样就清空了所有微任务

此时，所有的mircotask执行完毕，本轮事件循环结束，UI 开始 render，当 UI render 完毕，开始下一轮事件循环.

执行 setTimeout 任务，输出 setTimeout, 至此，输出的顺序是：script start, script end, promise1, promise2, setTimeout

UI渲染

根据HTML Standard，一轮事件循环执行结束之后，下轮事件循环执行之前开始进行 UI render。即：macro-task任务执行完毕，接着执行完所有的micro-task任务后，此时本轮循环结束，开始执行UI render。UI render完毕之后接着下一轮循环。

**83. ★★★★ 图片懒加载怎么实现？**

原理：随着滚轮滚动，底部的图片会被不断地加载，从而显示在页面上，按需加载，当页面需要显示图片的时候才进行加载，否则不加载

1. 页面加载完成时记录每个img标签的src值的字符串，

2. 用鼠标滚轮判断图片是否出现在屏幕，如果是，则把记录的src值赋值给src属性

3. 然后让image的src来发起请求，获取对应的图片放置到DOM树的这个位置上，从而实现图片的页面渲染！

于是就可以知道，当进入页面的时候，其实我们已经把所有的图片的这个地址信息拿到了，图片懒加载的作用就是让这个图片的src按需发起请求，获取图片。

**84. ★★ for-in 循环会遍历出原型上的属性吗？怎么避免遍历到原型上的属性**

使用 for in 循环遍历对象的属性时，原型链上的所有属性都将被访问

只遍历对象自身的属性，而不遍历继承于原型链上的属性，需要使用hasOwnProperty 方法过滤一下。

Object.prototype.say="cgl";

var person ={

age: 18

};

for (var key in person) {

if(person.hasOwnProperty(key)){

console.log(key, eval("person."+key));

}

}

**85. ★★★ 简述call、apply、bind，call 和 apply哪个性能更好？**

1、call()

call() 方法调用一个函数, 其具有一个指定的 this值和分别地提供的参数(参数的列表)。 第一个参数：在 fun 函数运行时指定的 this 值;如果指定了 null 或者 undefined 则内部 this 指向 window，后面的参数：指定的参数列表

var fn = function(arg1, arg2) {

};

fn.call(this, arg1, arg2);

var numbers = [5, 458 , 120 , -215 ];

var maxInNumbers = Math.max.call(Math,5, 458 , 120 , -215); //获取数组中的最大值458

2、apply()

apply()方法调用一个函数, 其具有一个指定的 this 值，以及作为一个数组（或类似数组的对象）提供的参数。apply() 与 call() 非常相似，不同之处在于提供参数的方式。apply() 使用参数数组而不是一组参数列表。

var fn = function(arg1, arg2) {

};

fn.apply(this, [arg1, arg2])

var numbers = [5, 458 , 120 , -215 ];

//umber 本身没有 max 方法，但是 Math 有，我们就可以借助 call 或者 apply 使用其方法。

var maxInNumbers = Math.max.apply(Math, numbers), //获取数组中的最大值458

3、bind()

bind() 函数会创建一个新函数（称为绑定函数），新函数与被调函数（绑定函数的目标函数）具有相同的函数体（在 ECMAScript 5 规范中内置的call属性）。

当目标函数被调用时 this 值绑定到 bind() 的第一个参数，该参数不能被重写。绑定函数被调用时，bind() 也接受预设的参数提供给原函数。

一个绑定函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。提供的 this 值被忽略，同时调用时的参数被提供给模拟函数。

/\* ------call 和 apply哪个性能更好？------ \*/

call的性能要比apply好一些，尤其是传递给函数的参数超过3个时所以后期开发的时候，可以使用call多一些

（传参数3个以内的话，call和apply性能差不多，超过3个以上call更好一些）

**86. ★★ ES6 箭头函数和普通函数有什么差异？**

1. 相比普通函数更简洁的语法

2. 没有this,捕获其所在上下文的 this 值，作为自己的 this 值

3. 不能使用new,箭头函数作为匿名函数,是不能作为构造函数的,不能使用new

4. 不绑定arguments，用rest参数...解决

let test3=(...a)=>{console.log(a[1])} //22

5. 使用call()和apply()调用:由于 this 已经在词法层面完成了绑定，通过 call() 或 apply() 方法调用一个函数时，只是传入了参数而已，对 this 并没有什么影响：

6. 箭头函数没有原型属性

7. 不能简单返回对象字面量

let fun5 = ()=>({ foo: x }) //如果x => { foo: x } //则语法出错

8. 箭头函数不能当做Generator函数,不能使用yield关键字

9. 箭头函数不能换行

let a = ()

=>1; //SyntaxError: Unexpected token =>

**87. ★★★ Promise 避免回调地狱的语法糖--实现链式调用的核心点是什么？**

解决回调地狱的终极方法 async/await ES7的语法，可以通过 async/await让代码看起来像同步的

async异步 await等待

await 等待 就是当后面跟的是promise对象，就让他停止 ，先让里面的异步事情做完，在把结果返回给前面的新变量，在继续向后执行

他只生效当前作用域内部，也就是async函数内部。

实现链式调用的核心点:

在 then 中新创建的 Promise，它的状态变为 fulfilled 的节点是在上一个 Promise的回调执行完毕的时候。也就是说当一个 Promise 的状态被 fulfilled 之后，会执行其回调函数，而回调函数返回的结果会被当作 value，返回给下一个 Promise(也就是then 中产生的 Promise)，同时下一个 Promise的状态也会被改变(执行 resolve 或 reject)，然后再去执行其回调,以此类推下去…

**88. ★★★ 进程线程区别是什么？**

什么是进程？什么是线程？

进程是系统中正在运行的一个程序，程序一旦运行就是进程。

进程可以看成程序执行的一个实例。进程是系统资源分配的独立实体，每个进程都拥有独立的地址空间。一个进程无法访问另一个进程的变量和数据结构，如果想让一个进程访问另一个进程的资源，需要使用进程间通信，比如管道，文件，套接字等。

一个进程可以拥有多个线程，每个线程使用其所属进程的栈空间。线程与进程的一个主要区别是，统一进程内的一个主要区别是，同一进程内的多个线程会共享部分状态，多个线程可以读写同一块内存（一个进程无法直接访问另一进程的内存）。同时，每个线程还拥有自己的寄存器和栈，其他线程可以读写这些栈内存。

线程是进程的一个实体，是进程的一条执行路径。

线程是进程的一个特定执行路径。当一个线程修改了进程的资源，它的兄弟线程可以立即看到这种变化。

进程和线程的区别体现在以下几个方面：

1.地址空间和其他资源（如打开文件）：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其他进程内不可见。

2.通信：进程间通信IPC（管道，信号量，共享内存，消息队列），线程间可以直接独写进程数据段（如全局变量）来进程通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。

3.调度和切换：线程上下文切换比进程上下文切换快得多。

4.在多线程OS中，进程不是一个可执行的实体。

**89. ★★★ 禁止事件冒泡，禁止默认事件**

/\*-----禁止事件冒泡:-----\*/

function stopBubble(e) {

//如果提供了事件对象，则这是一个非IE浏览器

if ( e && e.stopPropagation )

//因此它支持W3C的stopPropagation()方法

e.stopPropagation();

else

//否则，我们需要使用IE的方式来取消事件冒泡

window.event.cancelBubble = true;

}

/\*-----阻止浏览器的默认行为-----\*/

function stopDefault( e ) {

//阻止默认浏览器动作(W3C)

if ( e && e.preventDefault )

e.preventDefault();

//IE中阻止函数器默认动作的方式

else

window.event.returnValue = false;

return false;

}

**90. ★★ import export commonJS 对比区别**

ES6和commonJS的一些区别

从语法的角度上看，ES6模块化的import 和 export 是一个内置的标识，而commonJS的module.exports 和 require 分别是js对象和方法。其ES6模块化和commonJS的实现方式不同。

1.ES6是在编译的时候导入文件，而commonJS是编译完成后，在通过require方法导入，并读取文件导出的文件，并返回一个module.exports对象

2.在ES6模块的内部this是问undefined，而commonJS的this为一个空对象

3.ES6模块输出的是一个引用，而commonJS模块输出的是一个值的引用

**91. ★★ 为什么 JavaScript 是单线程**

js作为主要运行在浏览器的脚本语言，js主要用途之一是操作DOM。

举一个栗子，如果js同时有两个线程，同时对同一个dom进行操作，这时浏览器应该听哪个线程的，如何判断优先级？

为了避免这种问题，js必须是一门单线程语言

**92. ★★★ 使用箭头函数应该注意什么？**

1. 不要在对象里面定义函数，对象里面的行数应该用传统的函数方法

2. 不要在对原型对象上定义函数，在对象原型上定义函数也是遵循着一样的规则

3. 不要用箭头定义构造函数

4. 不要用箭头定义事件回调函数

**93. ★★★ 你知道 ES6 中的 Generator 和 yiled 吗？在实际开发中使用过吗？**

Generator 函数是 ES6 提供的一种异步编程解决方案

执行 Generator 函数会返回一个遍历器对象，可以依次遍历 Generator 函数内部的每一个状态

形式上，Generator函数是一个普通函数，但是有两个特征：

1. function关键字与函数名之间有一个星号

2. 函数体内部使用yield表达式，定义不同的内部状态

/\*-----利用Generator函数，在对象上实现Iterator接口-----\*/

function\* iterEntries(obj) {

let keys = Object.keys(obj);

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

let key = keys[i];

yield [key, obj[key]];

}

}

let myObj = { foo: 3, bar: 7 };

for (let [key, value] of iterEntries(myObj)) {

console.log(key, value);

}

**94. ★★★ Cookie、storage 的区别？什么时候使用？**

区别：

1. cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和服务器间来回传递。localStorage不会自动把数据发给服务器，仅在本地保存。

2. cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下。

3. 存储大小限制也不同，cookie数据不能超过4k，同时因为每次http请求都会携带cookie，所以cookie只适合保存很小的数据，如会话标识。localStorage 虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。

4. 数据有效期不同，

localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；

cookie只在设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭。

5. WebStorage 支持事件通知机制，可以将数据更新的通知发送给监听者。

6. WebStorage 的 api 接口使用更方便。

使用场景：

localStorage可以用来统计页面访问次数。

cookie一般存储用户名密码相关信息，一般使用escape转义编码后存储。

**95. ★★★ map、fillter、reduce 各自有什么作用？**

1：map 作用是生成一个新数组，遍历原数组，将每个元素拿出来做一些变换然后放入到新的数组中。

另外 map 的回调函数接受三个参数，分别是当前索引元素，索引，原数组

2：filter 的作用也是生成一个新数组，在遍历数组的时候将返回值为 true 的元素放入新数组，我们可以利用这个函数删除一些不需要的元素

和 map 一样，filter 的回调函数也接受三个参数，用处也相同。

3：reduce 可以将数组中的元素通过回调函数最终转换为一个值。

它接受两个参数，分别是回调函数和初始值，接下来我们来分解上述代码中 reduce 的过程

首先初始值为 0，该值会在执行第一次回调函数时作为第一个参数传入

回调函数接受四个参数，分别为累计值、当前元素、当前索引、原数组，后三者想必大家都可以明白作用，这里着重分析第一个参数

**96. ★★ JS的基本数据类型判断有什么方法？**

1.typeof：

typeof'';// string 有效

typeof 1;// number 有效

typeof Symbol();// symbol 有效

typeof true;//boolean 有效

typeof undefined;//undefined 有效

typeof null;//object 无效

typeof [] ;//object 无效

typeof new Function();// function 有效

typeof new Date();//object 无效

typeof new RegExp();//object 无效

2.instanceof

instanceof 是用来判断 A 是否为 B 的实例，表达式为：A instanceof B，如果 A 是 B 的实例，则返回 true,否则返回 false。 在这里需要特别注意的是：instanceof 检测的是原型

3.constructor

当一个函数 F被定义时，JS引擎会为F添加 prototype 原型，然后再在 prototype上添加一个 constructor 属性

4.toString

toString() 是 Object 的原型方法，调用该方法，默认返回当前对象的 [[Class]] 。这是一个内部属性，其格式为 [object Xxx] ，其中 Xxx 就是对象的类型。

对于 Object 对象，直接调用 toString() 就能返回 [object Object] 。而对于其他对象，则需要通过 call / apply 来调用才能返回正确的类型信息。

Object.prototype.toString.call('') ; // [object String]

Object.prototype.toString.call(1) ; // [object Number]

Object.prototype.toString.call(true) ;// [object Boolean]

Object.prototype.toString.call(Symbol());//[object Symbol]

Object.prototype.toString.call(undefined) ;// [object Undefined]

Object.prototype.toString.call(null) ;// [object Null]

Object.prototype.toString.call(newFunction()) ;// [object Function]

Object.prototype.toString.call(newDate()) ;// [object Date]

Object.prototype.toString.call([]) ;// [object Array]

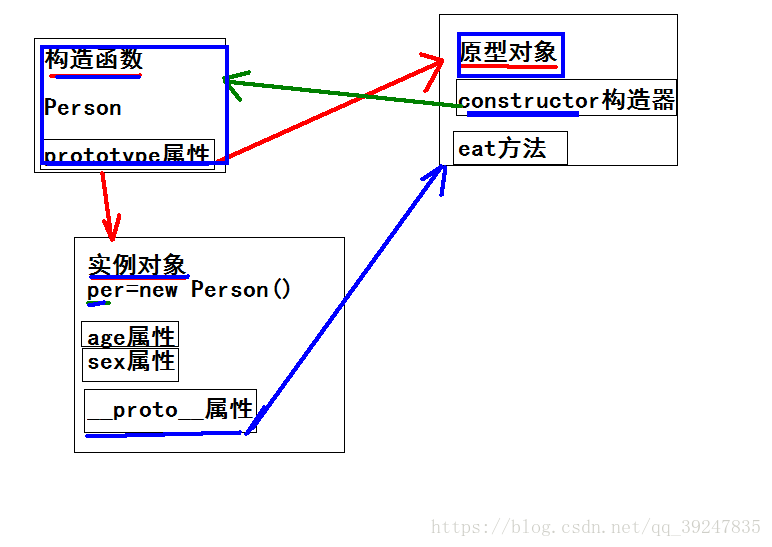
Object.prototype.toString.call(newRegExp()) ;// [object RegExp]

Object.prototype.toString.call(newError()) ;// [object Error]

Object.prototype.toString.call(document) ;// [object HTMLDocument]

Object.prototype.toString.call(window) ;//[object global] window 是全局对象 global 的引用

**97. ★★★ 构造函数、实例对象、原型对象三者的关系是什么？**



**98. ★★★★★ JS中的常见设计模式以及应用场景？**

1、单例模式

单例模式就是一个实例在整个网页的生命周期里只创建一次，后续再调用实例创建函数的时候，返回的仍是之前创建的实例。在实际开发中应用十分广泛，例如页面中的登录框，显示消息的提示窗

2、策略模式

策略模式是指将策略（算法）封装起来，策略的目的是将算法和使用分离开。

3、代理模式

代理模式很好理解，我们不能直接使用目标函数，而是通过调用代理函数来实现对目标函数的使用。

4、发布订阅模式

发布订阅模式在实际应用中非常常见，例如，我们在微信App上关注了某个公众号，当该公众号有新文章发布时，就会通知我们。

发布订阅模式定义了一种一对多的依赖关系，当“一”发生变化，通知多个依赖。

5、命令模式

所谓命令模式就是将下要执行的业务逻辑封装到一个函数或类中，不需要具体谁来执行该命令的

**99. ★★ 介绍下事件代理，主要解决什么问题**

1. 绑定事件太多，浏览器占用内存变大，严重影响性能

2. Ajax出现，局部刷新盛行，每次加载完，都要重新绑定事件

3. 部分浏览器移除元素时，绑定的事件没有被及时移除，导致内存泄漏，严重影响性能

4. Ajax中重复绑定，导致代码耦合性过大，影响后期维护

**100. ★★★★ 异步的解决方案有哪些？**

1.回调函数callback

2.事件发布订阅

3.Promise

4.Generator

5.async/await

**101. ★★ new 的原理是什么？通过 new 的方式创建对象和通过字面量创建有什么区别？**

new操作符的作用如下：

1.创建一个空对象

2.由this变量引用该对象

3.该对象继承该函数的原型

4.把属性和方法加入到this引用的对象中

5.新创建的对象由this引用，最后隐式地返回this。

区别:

字面量创建不会调用 Object构造函数, 简洁且性能更好;

**102. ★★ 数组去重的方法**

/\*

方法一：

双层循环，外层循环元素，内层循环时比较值

如果有相同的值则跳过，不相同则push进数组

\*/

Array.prototype.distinct = function(){

var arr = this,

result = [],

i,

j,

len = arr.length;

for(i = 0; i < len; i++){

for(j = i + 1; j < len; j++){

if(arr[i] === arr[j]){

j = ++i;

}

}

result.push(arr[i]);

}

return result;

}

var arra = [1,2,3,4,4,1,1,2,1,1,1];

arra.distinct(); //返回[3,4,2,1]

/\*

方法二：利用splice直接在原数组进行操作

双层循环，外层循环元素，内层循环时比较值

值相同时，则删去这个值

注意点:删除元素之后，需要将数组的长度也减1.

\*/

Array.prototype.distinct = function (){

var arr = this,

i,

j,

len = arr.length;

for(i = 0; i < len; i++){

for(j = i + 1; j < len; j++){

if(arr[i] == arr[j]){

arr.splice(j,1);

len--;

j--;

}

}

}

return arr;

};

var a = [1,2,3,4,5,6,5,3,2,4,56,4,1,2,1,1,1,1,1,1,];

var b = a.distinct();

console.log(b.toString()); //1,2,3,4,5,6,56

/\*

优点：简单易懂

缺点：占用内存高，速度慢

方法三：利用对象的属性不能相同的特点进行去重

\*/

Array.prototype.distinct = function (){

var arr = this,

i,

obj = {},

result = [],

len = arr.length;

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

if(!obj[arr[i]]){ //如果能查找到，证明数组元素重复了

obj[arr[i]] = 1;

result.push(arr[i]);

}

}

return result;

};

var a = [1,2,3,4,5,6,5,3,2,4,56,4,1,2,1,1,1,1,1,1,];

var b = a.distinct();

console.log(b.toString()); //1,2,3,4,5,6,56

/\*

方法四：数组递归去重

运用递归的思想

先排序，然后从最后开始比较，遇到相同，则删除

\*/

Array.prototype.distinct = function (){

var arr = this,

len = arr.length;

arr.sort(function(a,b){ //对数组进行排序才能方便比较

return a - b;

})

function loop(index){

if(index >= 1){

if(arr[index] === arr[index-1]){

arr.splice(index,1);

}

loop(index - 1); //递归loop函数进行去重

}

}

loop(len-1);

return arr;

};

var a = [1,2,3,4,5,6,5,3,2,4,56,4,1,2,1,1,1,1,1,1,56,45,56];

var b = a.distinct();

console.log(b.toString()); //1,2,3,4,5,6,45,56

//方法五：利用indexOf以及forEach

Array.prototype.distinct = function (){

var arr = this,

result = [],

len = arr.length;

arr.forEach(function(v, i ,arr){ //这里利用map，filter方法也可以实现

var bool = arr.indexOf(v,i+1); //从传入参数的下一个索引值开始寻找是否存在重复

if(bool === -1){

result.push(v);

}

})

return result;

};

var a = [1,1,1,1,1,1,1,1,1,2,2,2,2,2,2,3,3,3,3,3,3,3,2,3,3,2,2,1,23,1,23,2,3,2,3,2,3];

var b = a.distinct();

console.log(b.toString()); //1,23,2,3

方法六：利用ES6的set

Set数据结构，它类似于数组，其成员的值都是唯一的。

利用Array.from将Set结构转换成数组

function dedupe(array){

return Array.from(new Set(array));

}

dedupe([1,1,2,3]) //[1,2,3]

拓展运算符(...)内部使用for...of循环

1

2

3

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

let resultarr = [...new Set(arr)];

console.log(resultarr); //[1,2,3]

**103. ★★★★ 常见内存泄漏**

1、静态集合类，如HashMap、LinkedList等等。如果这些容器为静态的，那么它们的生命周期与程序一致，则容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有短生命周期对象的引用，尽管短生命周期的对象不再使用，但是因为长生命周期对象持有它的引用而导致不能被回收。

2、各种连接，如数据库连接、网络连接和IO连接等。在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用close方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对Connection、Statement或ResultSet不显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。

3、变量不合理的作用域。一般而言，一个变量的定义的作用范围大于其使用范围，很有可能会造成内存泄漏。另一方面，如果没有及时地把对象设置为null，很有可能导致内存泄漏的发生。

4、内部类持有外部类，如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

5、改变哈希值，当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为的参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，造成内存泄露

6、缓存泄漏

内存泄漏的另一个常见来源是缓存，一旦你把对象引用放入到缓存中，他就很容易遗忘，对于这个问题，可以使用WeakHashMap代表缓存，此种Map的特点是，当除了自身有对key的引用外，此key没有其他引用那么此map会自动丢弃此值

7、监听器和回调

内存泄漏第三个常见来源是监听器和其他回调，如果客户端在你实现的API中注册回调，却没有显示的取消，那么就会积聚。需要确保回调立即被当作垃圾回收的最佳方法是只保存他的若引用，例如将他们保存成为WeakHashMap中的键。

**104. ★★★ promise 常见方法和 all 和 race的应用场景**

Promise.race（）：

race的用法：谁跑的快，以谁为准执行回调。

race的使用场景：比如我们可以用race给某个异步请求设置超时时间，并且在超时后执行相应的操作

Promise.all():

all的用法：谁跑的慢，以谁为准执行回调。

在前端的开发实践中，我们有时会遇到需要发送多个请求并根据请求顺序返回数据的需求

**105. ★★★ 介绍一下 ES6 中 Set, Map的区别？**

Map

在JS中的默认对象的表示方式为{}，即一组键值对，但是键必须是字符串。

为了使用Number或者其他数据类型作为键，ES6规范引入了新的数据类型Map。

Map是一组键值对的结构，具有极快的查找速度。初始化Map需要一个二维数组，或者直接初始化一个空Map。

Map 对象是键值对集合，和 JSON 对象类似，但是 key 不仅可以是字符串还可以是其他各种类型的值包括对象都可以成为Map的键

Set

Set也是一组key的集合，与Map类似。但是区别是Set不存储value，并且它的key不能重复。

创建一个Set，需要提供一个Array作为输入，或者直接创建一个空Set：

重复元素会在Set中自动被过滤

Set 对象类似于数组，且成员的值都是唯一的

**106. ★★ 并行和并发的区别是什么？**

并行意味着可以同时取得多个任务，并同时去执行所取得的这些任务。并行模式相当于将长长的一条队列，划分成了多条短队列，所以并行缩短了任务队列的长度

并发表示多个任务同时都要执行

**107. ★★★ 为什么操作 dom 慢？**

DOM对象本身也是一个js对象，所以严格来说，并不是操作这个对象慢，而是说操作了这个对象后，需要经过跨流程通信和渲染线程触发的重新渲染，导致DOM操作慢

JS引擎和和渲染引擎的模块化设计，使得它们可以独立优化，运行速度更快，但是这种设计带来的后果就是DOM操作会越来越慢

**108. ★★★★ 插入几万个 dom ，如何实现页面不卡顿？**

让创建插入节点的工作分批进行：

setTimeout(() => {

// 插入十万条数据

const total = 100000;

// 一次插入 20 条，如果觉得性能不好就减少

const once = 20;

// 渲染数据总共需要几次

const loopCount = total / once;

let countOfRender = 0

let ul = document.querySelector("ul");

function add() {

// 优化性能，插入不会造成回流

const fragment = document.createDocumentFragment();

for (let i = 0; i < once; i++) {

const li = document.createElement("li");

li.innerText = Math.floor(Math.random() \* total);

fragment.appendChild(li);

}

ul.appendChild(fragment);

countOfRender += 1;

loop();

}

function loop() {

if (countOfRender < loopCount) {

window.requestAnimationFrame(add);

}

}

loop();

}, 0);

**109. ★★★ js中的常用事件绑定方法**

1. 在DOM元素中直接绑定

2. 在JavaScript代码中绑定

3. 绑定事件监听函数

**110. ★ 简述 src 和 href 的区别？**

src用于替换当前元素，href用于在当前文档和引用资源之间确立联系

**111. ★★★★ 你知道什么是原型吗？我们为什么要用原型呢？或者说原型为我们提供了什么？**

什么是原型：

Javascript规定，每一个函数都有一个prototype对象属性，指向另一个对象（原型链上面的）。

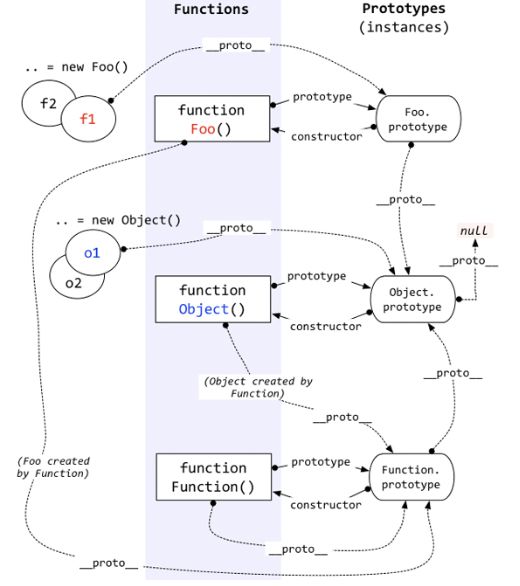
prototype(对象属性)的所有属性和方法，都会被构造函数的实例继承。这意味着，我们可以把那些不变(公用)的属性和方法，直接定义在prototype对象属性上。

prototype就是调用构造函数所创建的那个实例对象的原型（proto）。

prototype可以让所有对象实例共享它所包含的属性和方法。也就是说，不必在构造函数中定义对象信息，而是可以直接将这些信息添加到原型中。

为什么要用原型：使用原型对象解决浪费内存

**112. ★★★ 你了解原型链吗 你能说说 prototype 与 proto 的区别吗？**



1.对象有属性\_\_proto\_\_,指向该对象的构造函数的原型对象。

2.方法除了有属性\_\_proto\_\_,还有属性prototype，prototype指向该方法的原型对象。

**113. ★★★ ts 和 js 的区别**

1.ts是静态类语言，可以做到声明即文档，js是动态类语言相对更灵活。

2.如用ts写一个button组件可以清晰的知道，ButtonProps如是否必传，可选，style是什么类型，disabled是什么类型，较js，ts更易于维护和拓展，可以做到代码即注释，避免一个月不见3，代码自己都忘记自己写了什么的尴尬，

4.ts对比js基础类型上，增加了 void/never/any/元组/枚举/以及一些高级类型

5.js没有重载概念，ts有可以重载

6.vscode/ide对ts有很友好的提示

7.ts更利于重构

**114. ★★★ 简述原生 js 发 ajax 的步骤**

1.创建XMLHTTPRequest对象

2.使用open方法设置和服务器的交互信息

3.设置发送的数据，开始和服务器端交互

4.注册事件

5.更新界面

**115. ★★ 是否所有函数都有 prototype 一说？**

1. 使用Function.prototype.bind创建的函数对象

function abc(){console.log('abc')}

var binded = abc.bind(null)

binded() //abc

console.log(binded.prototype) //undefined

2. 箭头函数也没有

var abc = ()=>{console.log('abc')}

abc() //abc

console.log(abc.prototype) //undefined

**116. ★★ 为什么 await 在 forEach 中不生效？如何解决？**

lodash的forEach和[].forEach不支持await, forEach 只支持同步代码。

解决方法1：使用 for...of

解决方法2：使用 for循环

解决方法3：让orEach支持async await

forEach 在正常情况像下面这么写肯定是做不到同步的，程序不会等一个循环中的异步完成再进行下一个循环。原因很明显，在上面的模拟中，while 循环只是简单执行了 callback，所以尽管 callback 内使用了 await ，也只是影响到 callback 内部。

arr.myforeach(async v => {

await fetch(v);

});

要支持上面这种写法，只要稍微改一下就好

Array.prototype.myforeach = async function (fn, context = null) {

let index = 0;

let arr = this;

if (typeof fn !== 'function') {

throw new TypeError(fn + ' is not a function');

}

while (index < arr.length) {

if (index in arr) {

try {

await fn.call(context, arr[index], index, arr);

} catch (e) {

console.log(e);

}

}

index ++;

}

};

**117. ★ a 标签中，如何禁用 href 跳转页面或定位链接？**

<a href="#" onclick="return false;">return false;</a>

<a href="http://www.baidu.com" onclick="ds(event)">baidu</a>

<script>

function ds(e){

e.preventDefault();

}

</script>

**118. ★★ 请描述一下 cookies，sessionStorage 和 localStorage 的区别？**

不同点：

1.存储大小

cookie数据大小不能超过4k。

sessionStorage和localStorage 虽然也有存储大小的限制，但比cookie大得多，可以达到5M或更大。

2.有效时间

localStorage 存储持久数据，浏览器关闭后数据不丢失除非主动删除数据；

sessionStorage 数据在当前浏览器窗口关闭后自动删除。

cookie 设置的cookie过期时间之前一直有效，即使窗口或浏览器关闭

3. 数据与服务器之间的交互方式

cookie的数据会自动的传递到服务器，服务器端也可以写cookie到客户端

sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。

**119. ★★★ instanceof的原理是什么？**

// instanceof 可以正确的判断对象的类型，是通过判断对象的原型链中是不是能找到类型的 prototype。

function fn(left, right) {

let prototype = right.prototype;

left = left.\_\_proto\_\_;

while (true) {

if (left === undefined || left === null) {

return false;

}

if (left === prototype) {

return true;

}

left = left.\_\_proto\_\_;

}

}

**120. ★★★ 用多种方法实现 JavaScript 继承**

1、原型链继承

核心： 将父类的实例作为子类的原型

特点：

1. 非常纯粹的继承关系，实例是子类的实例，也是父类的实例

2. 父类新增原型方法/原型属性，子类都能访问到

3. 简单，易于实现

缺点：

1. 要想为子类新增属性和方法，必须要在new Animal()这样的语句之后执行，不能放到构造器中

2. 无法实现多继承

3. 来自原型对象的所有属性被所有实例共享（来自原型对象的引用属性是所有实例共享的）（详细请看附录代码： 示例1）

4. 创建子类实例时，无法向父类构造函数传参

推荐指数：★★（3、4两大致命缺陷）

2、构造继承

核心：使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

特点：

1. 解决了1中，子类实例共享父类引用属性的问题

2. 创建子类实例时，可以向父类传递参数

3. 可以实现多继承（call多个父类对象）

缺点：

1. 实例并不是父类的实例，只是子类的实例

2. 只能继承父类的实例属性和方法，不能继承原型属性/方法

3. 无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

推荐指数：★★（缺点3）

3、实例继承

核心：为父类实例添加新特性，作为子类实例返回

特点：

1.不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

1. 实例是父类的实例，不是子类的实例

2. 不支持多继承

推荐指数：★★

4、拷贝继承

特点：支持多继承

缺点：

1. 效率较低，内存占用高（因为要拷贝父类的属性）

2. 无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

推荐指数：★（缺点1）

5、组合继承

核心：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

特点：

1. 弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法

2. 既是子类的实例，也是父类的实例

3. 不存在引用属性共享问题

4. 可传参

5. 函数可复用

缺点：

1. 调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

推荐指数：★★★★（仅仅多消耗了一点内存）

6、寄生组合继承

核心：通过寄生方式，砍掉父类的实例属性，这样，在调用两次父类的构造的时候，就不会初始化两次实例方法/属性，避免的组合继承的缺点

特点：堪称完美

缺点：实现较为复杂

推荐指数：★★★★★

##### ★ 父元素和子元素宽高不固定，如何实现水平垂直居中

- 弹性盒模型

-父元素设置：display:flex; justify-content:center; align-items:center

-额外的骚操作：父元素设置弹性盒display:flex; 子元素可以设置margin: auto; 实现垂直水平居中

- 定位属性（position）配合位移属性（transform）

- 父元素设置：position:relative，

- 子元素设置：position:absolute; top:50%; left:50%; transform:translate(-50%，-50%)

##### 2. ★★ 分别实现骰子中的'一点' 和 '三点' 的布局。

链接：

/\*一点布局：原理就是将单个子元素垂直水平居中，利用弹性盒模型(display:flex)即可\*/

父元素设置：display:flex; justify-content:center; align-items: center

/\*三点布局：同样是使用弹性盒模型(display:flex)，只不过这次需要用到其他属性\*/

父元素设置：display: flex; justify-content: space-between;

子元素设置： .child:nth-child(2){

align-self: center;

}

.child:nth-child(3){

align-self: flex-end;

}

三点布局需要三个子元素，在这里的第一个子元素不需要设置排列方式，默认为 align-self: flex-start

justify-content: space-between; 的作用是使子元素能够在水平方向上两边产生间隔并平均分布空间

align-self: flex-start(默认)/center/flex-end; 该属性作用在父元素设置了display:flex的子元素上，可以调整子元素自身的位置

##### 3. ★★ 简述选择器~和+的区别。

~ 选择器的作用：

1.选择紧跟着当前符合条件元素后面的同级元素

2.可以匹配多个

+ 选择器的作用：

1.选择紧跟在当前符合条件元素后面的同级元素

2.只能匹配一个

例： <div class="box"></div>

<p class="one"></p>

<span class="two"></span>

<p class="three"></p>

<span class="four"></span>

.box ~ p ：可以选中box下的所有p元素，既是 one 和 three

.box + span ：则匹配选中box相邻下符合条件的第一个元素，既是 two

##### 4. ★★ 简述box-sizing的有效值以及所对应的盒模型规则。

/\*box-sizing值\*/

box-sizing: content-box/border-box/inherit

box-sizing属性用于更改用于计算元素的宽度和高度默认的CSS盒子模型，可以使用此属性来模拟不正确支持CSS盒子模型规范的游览器行为。

\*\*content-box：

1.box-sizing的默认属性

2.是CSS2.1中规定的宽度高度的显示行为

3.在CSS中定义的宽度和高度就对应到元素的内容框（即元素容器本身

4.在CSS中定义的宽度和高度之外绘制元素的内边距和边框（即在元素容器本身上增加内边距和边框，容器宽高需要自增计算

容器占据空间大小计算方式：content（容器宽高）+ margin（外边距） + border（边框）

\*\*border-box:

1.在CSS中微元素设定的宽度和高度就决定了元素的边框值

2.元素在设置内边距和边框是在已经设定好的宽度和高度之内进行绘制

3.CSS中设定的宽度和高度减去边框和内间距才能得到元素内容所占的实际宽度和高度

容器占据空间大小计算方式：content（容器宽高）+padding（内边距）+border（边框）

\*\*inherit：

1.规定元素是从父容器继承box-sizing的属性值

##### 5. ★★★ html中元素的margin是否会叠加（合并）？如何解决？

/\* 会叠加 \*/

问题详解1： flex布局对子元素的影响

1.子元素的float、clear和vertical-align属性将会失效

2.解决了margin传递、重叠（叠加）问题

问题详解2：flex布局的margin传递叠加问题主要有以下两种

1.父子间的margin，会由子级传递到父级

—— 解决方法： margin传递的产生的原因是父级的高度没有被自动撑开，所以在父级父级增加属性：overflow: auto 即可解决

2.兄弟间的margin值会重复叠加

—— 解决方法： 浏览器为了保证列表的整齐，上下margin产生了叠加，不能直接解决。只能通过减少一个margin的方式。如只定义margin-top:100px; margin-bottom:0px。的方式解决。

##### 6. ★★ 简述align-items和align-content的区别。

align-items：可以应用于所有的flex容器，它的作用是设置flex子项在每个flex行的交叉轴上的默认对齐方式（相对Y轴

align-content：只适用于多行的flex容器，在使用前需在flex容器设置flex-wrap：wrap;表示子元素超出换行；align-content 它的作用是当flex容器在交叉轴上有多余的空间时，将子项作为一个整体进行对齐。

##### 7. ★★ 简述data-\*属性的用法（如何设置，如何获取），有何优势？

data-\*定义：

1.是用于储存页面或应用程序的私有自定义数据

2.赋予我们在所有html元素上嵌入自定义data属性的能力

data-\*用法：

1.属性名不应该包含任何大写字母，并且在前缀 "data-" 之后必须有至少一个字符

2.属性值可以是任意字符串

3.一个元素可以拥有任意数量的data属性

4.data属性无法储存对象，如需储存，可通过对象序列化

data-\*如何设置、获取：

1.如何设置

通过JavaScript内置的setAttribute('data属性名','新内容')即可设置

通过该数据类型的(dataset) API设置data值，IE10以上才支持；

var button = document.queryselector('button')

button.dataset.data属性名 = '新内容' ; 这里的data属性名是指data-后面的名字

2.如何获取

通过JavaScript内置的getAttribute('data属性名') 即可获取

通过该数据类型的(dataset) API设置data值，IE10以上才支持；

var button = document.queryselector('button')

data = button.dataset.data属性名 ; 这里的data属性名是指data-后面的名字

data-\*优势：

1.其储存的自定义数据能够被页面的JavaScript利用，可以创建更好的用户体验

2.可以通过JavaScript来构造数据、填充数据

3.代码体积小、较为灵活

4.解决网站的外观和实用性之间产生的冲突

##### 8. ★ 简述title与h1的区别，b与strong的区别，i与em的区别。

《title与h1的区别》：

1. 从网站角度看，title 更重于网站信息。title可以直接告诉搜索引擎和用户这个网站是关于什么主题和内容的。

2. 从文章角度看，h1则是用于概括文章主题。

3. 一个网站可以有多个title,最好一个单页用一个title，以便突出网站页面主体信息，从seo看，title权重比h1高，适用性比h1广。

4. 标记了h1的文字页面给予的权重会比页面内其他权重高很多。一个好的网站是h1和title并存，既突出h1文章主题，又突出网站主题和关键字。达到双重优化网站的效果。

《b与strong 的区别》：

1. b 是只是对文本的简单加粗， strong 是一个语义化标签，对相关文本具有强调作用

2. b 标签只是侧重于字体加粗， strong标签加强字体的语气都是通过粗体来实现的，相比之下，搜索引擎更喜欢侧重于strong标签

3. strong标签更注重于内容上的应用，在html中，对关键词的标明，然而还有一些网站上，也有使用strong标签登对小标题进行强调，但是在页面中，如果出现过多的strong标签，可能会对排名不利。

《i 与 em 的区别》：

1. i(italic)是实体标签，用来使字符倾斜，em(emphasis)是逻辑标签，作用是强调文本内容

2. i标签只是斜体的样式，没有实际含义，常用来表达无强调或着重意味的斜体，比如生物学名、术语、外来语；

3. em表示标签内字符重要，用以强调，其默认格式是斜体，但是可以通过CSS添加样式。

建议：为了符合CSS3的规范，i 标签应尽量少用而应改用 em

##### 9. ★ 什么是标准文档流

\*\*\* 标准文档流指的是元素排版布局过程中，元素会默认自动从左往右，从上往下的流式排列方式。当前面内容发生了变化，后面的内容位置也会随着发生变化。

\*\*\* HTML就是一种标准文档流文件。

简单的来说就是各种布局属性在html中所显示的效果，如display（行内元素与块级元素\*非常重要\*）、float、position

##### 10. ★ z-index是什么？在position的值什么时候可以触发?

z-index : 指的是一个元素在当前文档页面定位时重叠层显示的层级等级，默认为0 ，数值不限，越大显示层级越高

触发机制：当position的值设置为absolute、relative和fixed时才能触发

##### 11. ★★ CSS3 如何实现圆角？

border-radius属性

1. 四个值: 第一个值为左上角，第二个值为右上角，第三个值为右下角，第四个值为左下角。

2. 三个值: 第一个值为左上角, 第二个值为右上角和左下角，第三个值为右下角

3. 两个值: 第一个值为左上角与右下角，第二个值为右上角与左下角

4. 一个值： 四个圆角值相同

只需要使用border-radius将四个角设置为相适应的尺寸即可实现圆角

##### 12. ★★ HTML5有哪些缓存方式？

1、localstorege缓存，将数据储存在本地客户端，只有用户手动清除才能清除缓存

API：1.localstorege.setItem(key,value)，键值对的形式缓存

2.localstorege.getItem(key)，根据键名来缓存值

3.localstorege.length ，获取总缓存数量

2、sessionStorege 会话缓存，会话机制是指从打开浏览器开始访问页面的时候，到关闭这个页面的过程成为一个会话，sessionStorege储存的数据会随着页面关闭而销毁

API: 1. sessionStorage.setItem(key,val)，localStorage是以键值对的形式创建的；

2. sessionStorage.getItem(key)，根据键名来获取缓存的值；

3. sessionStorage.length；获取总共缓存值得数量， localStoarge返回的是个对象；

3、离线缓存机制(Application Cache)

1. 配置manifest文件，manifest 文件是简单的文本文件，它告知浏览器被缓存的内容（以及不缓存的内容）

2. manifest 文件可分为三个部分：

1、CACHE MANIFEST - 在此标题下列出的文件将在首次下载后进行缓存

2、NETWORK - 在此标题下列出的文件需要与服务器的连接，且不会被缓存

3、FALLBACK - 在此标题下列出的文件规定当页面无法访问时的回退页面（比如 404 页面）

3. API： 0（UNCACHED） : 无缓存， 即没有与页面相关的应用缓存

1（IDLE） : 闲置，即应用缓存未得到更新

2 （CHECKING） : 检查中，即正在下载描述文件并检查更新

3 （DOWNLOADING） : 下载中，即应用缓存正在下载描述文件中指定的资源

4 （UPDATEREADY） : 更新完成，所有资源都已下载完毕

5 （IDLE） : 废弃，即应用缓存的描述文件已经不存在了，因此页面无法再访问应用缓存

4、web SQL

1. 关系数据库，通过SQL语句访问

2. Web SQL 数据库API并不是HTML5 规范的一部分，但是它是一个独立的规范，引入了一组使用SQL操作客户端数据库的APIs

3. 支持情况：Web SQL 数据库可以在最新版的 Safari, Chrome 和 Opera 浏览器中工作。

4. API：1. openDatabase：这个方法使用现有的数据库或者新建的数据库创建一个数据库对象。

2. transaction：这个方法让我们能够控制一个事务，以及基于这种情况执行提交或者回滚。

3. executeSql：这个方法用于执行实际的 SQL 查询。

5、 IndexDB

1.索引数据库 (IndexedDB) API（作为 HTML5 的一部分）对创建具有丰富本地存储数据的数据密集型的离线 HTML5 Web 应用程序很有用。同时它还有助于本地缓存数据，使传统在线 Web 应用程序（比如移动 Web 应用程序）能够更快地运行和响应。

##### 13. ★★ CSS3新增伪类有那些？

常用的伪类：

1. :link 选择所有未访问的链接

2. :visited 选择所有访问过的链接

3. :active 选择正在活动的链接（或理解为鼠标点击瞬间效果）

4. :hover 鼠标放到链接后的状态

5. :focus 选择元素输入后具有焦点

6. :before 在元素之前插入内容

7. :after 在元素之后插入内容

##### 14. ★ 简述一下src与href的区别，title和alt的区别

href：href表示超文本引用，用来建立当前元素和文档之间的链接，常用在link和a等元素上。

注：当浏览器解析到这一句时会识别该文档为css文件，会下载并不会停止对当前文档的处理，所以建议使用link方式而不是@import加载css。

src：src表示引用资源，替换当前元素，是页面内容不可缺少的一部分，常用在img，script，iframe上。src指向外部资源的位置，指向的内部会迁入到文档中当前标签所在的位置；请求src资源时会将其指向的资源下载并应用到当前文档中,例如js脚本、img图片等。src链接内的地址不会有跨域问题

注：当浏览器解析到这一句时会暂停其他资源的下载和处理，直至将该资源加载、编译、执行完毕。这也是js脚本放在底部而不是头部的问题

title：

1. title属性是为元素提供额外的注释信息，当鼠标放在元素上时会有title文字显示，以达到补充说明或提示。

2. title属性更倾向于用户体验的考虑。

3. title既可以是元素的属性也可以是标签，作为属性可以用在除base,basefont,head,html,meta,param,script和title之外的任何标签上（title常与form以及a标签一同使用，以提供关于输入格式和链接目标的信息），title与style、id、class等一起作为HTML中许多标签共用的标准属性。

alt：

1. alt属性是在你的图片无法显示时的替代文本，它会直接输出在原本加载图片的地方。

2. alt属性有利于SEO，是搜索引擎搜录时判断图片与文字是否相关的重要依据。

3. alt只能是元素的属性，只能用在img、area和input标签中（img,area中alt必须指定）。

注：当a标签内嵌套img标签时，起作用的是img的title属性。

##### 15. ★ 什么是CSS hack？

CSS hack：CSS hack是通过在CSS样式中加入一些特殊的符号，让不同的浏览器识别不同的符号（什么样的浏览器识别什么样的符号是有标准的，CSS hack就是让你记住这个标准），以达到应用不同的CSS样式的目的。

注：CSS属性Hack、CSS选择符Hack以及IE条件注释Hack， Hack主要针对IE浏览器。

例：margin属性在ie6中显示的距离会比其他浏览器中显示的距离宽2倍，也就是说margin-left:20px;在ie6中距左侧对象的实际显示距离是40px，而在非ie6中显示的距左侧对象的距离是设置的值20px;所以要想设置一个对象距离左侧对象的距离在所有浏览器中都显示是20px的宽度的样式应为：.kwstu{margin-left:20px;\_margin-left:20px;}

CSS hack常见的三种形式：

1. 属性级Hack：比如IE6能识别下划线“\_”和星号“\*”，IE7能识别星号“\*”，但不能识别下划线”\_ ”，而firefox两个都不能认识。

2. 选择符级Hack：比如IE6能识别\*html .class{}，IE7能识别\*+html .class{}或者\*:first-child+html .class{}。

3. IE条件注释Hack：IE条件注释是微软IE5开始就提供的一种非标准逻辑语句。比如针对所有IE：&lt;!-[if IE]&gt;&lt;!-您的代码-&gt;&lt;![endif]&gt;，针对IE6及以下版本：&lt;!-[if it IE 7]&gt;&lt;!-您的代码-&gt;&lt;![endif]-&gt;，这类Hack不仅对CSS生效，对写在判断语句里面的所有代码都会生效。

PS：条件注释只有在IE浏览器下才能执行，这个代码在非IE浏览下被当做注释视而不见。可以通过IE条件注释载入不同的CSS、JS、HTML和服务器代码等。

##### 16. ★★ 什么叫做优雅降级和渐进增强？

渐进增强 progressive enhancement：

1. 针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。

2. 渐进增强观点则认为应关注于内容本身。内容是我们建立网站的诱因。有的网站展示它，有的则收集它，有的寻求，有的操作，还有的网站甚至会包含以上的种种，但相同点是它们全都涉及到内容。这使得“渐进增强”成为一种更为合理的设计范例。这也是它立即被 Yahoo! 所采纳并用以构建其“分级式浏览器支持 (Graded Browser Support)”策略的原因所在。

优雅降级 graceful degradation：

1. 一开始就构建完整的功能，然后再针对低版本浏览器进行兼容。

2. 优雅降级观点认为应该针对那些最高级、最完善的浏览器来设计网站。而将那些被认为“过时”或有功能缺失的浏览器下的测试工作安排在开发周期的最后阶段，并把测试对象限定为主流浏览器（如 IE、Mozilla 等）的前一个版本。

区别：

1. 优雅降级是从复杂的现状开始，并试图减少用户体验的供给

2. 渐进增强则是从一个非常基础的，能够起作用的版本开始，并不断扩充，以适应未来环境的需要

3. 降级（功能衰减）意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带

##### 17. ★★★ 移动端适配怎么做？

方法一：@media 媒体查询，通过查询设备的宽度来执行不同的 css 代码，最终达到界面的配置。

方法二：Flex弹性布局

方法三：rem + viewport 缩放，屏幕宽度设定 rem 值，需要适配的元素都使用 rem 为单位，不需要适配的元素还是使用 px 为单位。

##### 18. ★★ 请问苹果原生浏览器中默认样式如何清除，例如button，input默认样式？

清除苹果默认样式： css样式中加入 input,textarea,button { -webkit-appearance: none; border-radius:0px; border:none;}

input、button默认样式： input[type="button"], input[type="submit"], input[type="reset"] {-webkit-appearance: none;}

##### 19. ★ CSS清除浮动的方法。

1. 在标签尾部添加空块级标签，设置样式属性为：clear：both；缺点：如果页面浮动布局多，就要增加很多空div，不利于页面的优化。

2. 父级定义伪类after和zoom，.box:after{display:block; clear:both; content:""; visibility:hidden; height:0;} .box{ zoom: 1 }

3. 简单粗暴，父级设置overflow:hidden，缺点是不能和position配合使用

4. 直接给父元素单独设置高度（height）；缺点：只适合高度固定的布局，要给出精确的高度，如果高度和父级div不一样时，会产生问题。对于响应式布局会有很大影响。

##### 20. ★★ PC端常用的布局方法。

1、利用float+overflow实现

左右定宽度，分布向两边浮动，中间如果没设置溢出处理默认宽度是100%，设置后就会截取两边的宽度从而实现中间自适应

2、flex布局

通过flex相关的属性进行适配性的调整布局

3、Table表格布局

4、float+margin实现三列布局

5、定位absolute实现中间自适应

6、Grid网格布局

7、圣杯布局

8、双飞翼布局

9、等高布局

##### 21. ★★ 布局左边20%，中间自适应，右边200px，不能用定位。

布局： <div class="box">

<div class="left">left</div>

<div class="right">right</div>

<div class="content">content</div>

</div>

样式：\*,html,body{

margin: 0;

padding: 0;

}

.box{

width: 1500px;

height: 500px;

background-color: rgb(215, 221, 221);

margin: 0 auto;

}

.left{

width: 20%;

height: 200px;

background-color: chocolate;

float: left;

}

.right{

width: 200px;

height: 200px;

float: right;

background-color: cornflowerblue;

}

.content{

overflow: hidden;

height: 400px;

background-color: darkblue;

}

##### 22. ★★ 行内元素和块级元素？img算什么？行内元素怎么转化为块元素？

行内元素：1.无法设置宽高；

2. 对margin仅设置左右有效，上下无效；

3. padding上下左右有效；不会自动换行

块级元素：1.可以设置宽高

2. margin和padding的上下左右均对其有效

3. 超出当前行会自动换行

4. 多个块状元素标签写在一起，默认排列方式为从上至下

img：属于行内块元素(inline-block),即有行内元素的属性也有块级元素的属性

元素之间的转化可以通过设置样式：display:block/inline/inline-block来改变自身的元素属性

##### 23. ★★ 将多个元素设置为同一行? 清除浮动的几种方式？

将多个元素设置为同一行：1. 浮动(float) 2.行内元素/行内块元素

清除浮动：详见 19

##### 24. ★ transform与animatio的区别是什么？

transform：1. transform 属性向元素应用 2D 或 3D 转换。该属性允许我们对元素进行旋转、缩放、移动或倾斜。

2. transform属性是静态属性，需要配合transition和animation才能展现出动画效果。你可以把它看成是跟left、top等属性一样，只是一个静态样式而已。

animation：一个css3的动画属性，需要配合@keyframes 使用

区别： 1. transition是css中检测指定属性变化进行自动补间动画的属性。

2. animate是先指定好动画过程中的关键帧属性，进行动画的属性。

# 程序面试真题

**★★ 简单描述下微信小程序的相关文件类型**

微信小程序项目结构主要有四个文件类型

* WXML（WeiXin Markup Language）是框架设计的一套标签语言，结合基础组件、事件系统，可以构建出页面的结构。内部主要是微信自己定义的一套组件
* WXSS (WeiXin Style Sheets)是一套样式语言，用于描述 WXML 的组件样式
* js 逻辑处理，网络请求
* json 小程序设置，如页面注册，页面标题及tabBar

主要文件

* app.json 必须要有这个文件，如果没有这个文件，项目无法运行，因为微信框架把这个作为配置文件入口，整个小程序的全局配置。包括页面注册，网络设置，以及小程序的 window 背景色，配置导航条样式，配置默认标题
* app.js 必须要有这个文件，没有也是会报错！但是这个文件创建一下就行 什么都不需要写以后我们可以在这个文件中监听并处理小程序的生命周期函数、声明全局变量
* app.wxss 可选

**★★★★ 简述微信小程序原理**

微信小程序采用 JavaScript、WXML、WXSS 三种技术进行开发,本质就是一个单页面应用，所有的页面渲染和事件处理，都在一个页面内进行，但又可以通过微信客户端调用原生的各种接口微信的架构，是数据驱动的架构模式，它的 UI 和数据是分离的，所有的页面更新，都需要通过对数据的更改来实现 小程序分为两个部分 webview 和 appService 。其中 webview 主要用来展现 UI ，appService 有来处理业务逻辑、数据及接口调用。它们在两个进程中运行，通过系统层 JSBridge 实现通信，实现 UI 的渲染、事件的处理

**★★★ 小程序的双向绑定和vue哪里不一样**

小程序直接 this.data 的属性是不可以同步到视图的，必须调用：

微信小程序在初始化之后，再对原来的数据对象进行任何更改，都始终不会生效！只能手动调用setData接口明确指明同步哪些键/值至视图层才会触发更新 vue则是对数据对象进行了监听，只要有更改就可以并且会立即触发视图层的更新

this.setData({

// 这里设置

})

设置值

* 在vue中,只需要再表单元素上加上v-model,然后再绑定data中对应的一个值，当表单元素内容发生变化时，data中对应的值也会相应改变，这是vue非常nice的一点。 ```js



new Vue({

el: '#app',

data: {

reason:''

}

})

- 但是在小程序中，却没有这个功能。那怎么办呢？当表单内容发生变化时，会触发表单元素上绑定的方法，然后在该方法中，通过this.setData({key:value})来将表单上的值赋值给data中的对应值。下面是代码，可以感受一下:

```js

<input bindinput="bindReason" placeholder="填写理由" class='reason' value='{{reason}}' name="reason" />

Page({

data:{

reason:''

},

bindReason(e) {

this.setData({

reason: e.detail.value

})

}

})

* 当页面表单元素很多的时候，更改值就是一件体力活了。和小程序一比较，vue的v-model简直爽的不要不要的。

取值

* vue中，通过this.reason取值
* 小程序中，通过this.data.reason取值

**★★ 小程序的 wxss 和 css 有哪些不一样的地方**

WXSS 和 CSS 类似，不过在 CSS 的基础上做了一些补充和修改

* 尺寸单位 rpx

rpx 是响应式像素,可以根据屏幕宽度进行自适应。规定屏幕宽为 750rpx。如在 iPhone6 上，屏幕宽度为 375px，共有 750 个物理像素，则 750rpx = 375px = 750 物理像素

* 使用 @import 标识符来导入外联样式。@import 后跟需要导入的外联样式表的相对路径，用;表示语句结束

/\*\* index.wxss \*\*/

@import './base.wxss';

.container{

color: red;

}

**★★★ 小程序页面间有哪些传递数据的方法**

* 使用全局变量实现数据传递

在 app.js 文件中定义全局变量 globalData， 将需要存储的信息存放在里面

// app.js

App({

// 全局变量

globalData: {

userInfo: null

}

})

使用的时候，直接使用 getApp() 拿到存储的信息

* 使用 wx.navigateTo 与 wx.redirectTo 的时候，可以将部分数据放在 url 里面，并在新页面 onLoad 的时候初始化

//pageA.js

// Navigate

wx.navigateTo({

url: '../pageD/pageD?name=raymond&gender=male',

})

// Redirect

wx.redirectTo({

url: '../pageD/pageD?name=raymond&gender=male',

})

// pageB.js

...

Page({

onLoad: function(option){//option里面可以拿到路由参数

console.log(option.name + 'is' + option.gender)

this.setData({

option: option

})

}

})

需要注意的问题：

wx.navigateTo 和 wx.redirectTo 不允许跳转到 tab 所包含的页面

onLoad 只执行一次

* 使用本地缓存 Storage 相关

**★★★ 小程序的生命周期函数**

* onLoad 页面加载时触发。一个页面只会调用一次，可以在 onLoad 的参数中获取打开当前页面路径中的参数,建议在此调接口
* onShow() 页面显示/切入前台时触发，不建议在此调接口
* onReady() 页面初次渲染完成时触发。一个页面只会调用一次，代表页面已经准备妥当，可以和视图层进行交互
* onHide() 页面隐藏/切入后台时触发。 如 navigateTo 或底部 tab 切换到其他页面，小程序切入后台等
* onUnload() 页面卸载时触发。如 redirectTo 或 navigateBack 到其他页面时

详见 [生命周期回调函数](https://link.zhihu.com/?target=https%3A//developers.weixin.qq.com/miniprogram/dev/framework/app-service/page.html%23%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E5%9B%9E%E8%B0%83%E5%87%BD%E6%95%B0" \t "_blank)

**★★★ 怎么封装微信小程序的数据请求**

参考 [这里](https://link.zhihu.com/?target=https%3A//segmentfault.com/a/1190000014789969" \t "_blank)

**★★★ 哪些方法可以用来提高微信小程序的应用速度**

1、提高页面加载速度

2、用户行为预测

3、减少默认 data 的大小

4、组件化方案

**★★ 微信小程序的优劣势**

优势

* 即用即走，不用安装，省流量，省安装时间，不占用桌面
* 依托微信流量，天生推广传播优势
* 开发成本比 App 低

缺点

* 用户留存，即用即走是优势，也存在一些问题
* 入口相对传统 App 要深很多
* 限制较多,页面大小不能超过2M。不能打开超过10个层级的页面

**★★★ 怎么解决小程序的异步请求问题**

小程序支持大部分 ES6 语法

* 在返回成功的回调里面处理逻辑
* Promise 异步

**★★★ 小程序关联微信公众号如何确定用户的唯一性**

如果开发者拥有多个移动应用、网站应用、和公众帐号（包括小程序），可通过 unionid 来区分用户的唯一性，因为只要是同一个微信开放平台帐号下的移动应用、网站应用和公众帐号（包括小程序），用户的 unionid 是唯一的。换句话说，同一用户，对同一个微信开放平台下的不同应用，unionid 是相同的

**★★★ 如何实现下拉刷新**

* 首先在全局 config 中的 window 配置 enablePullDownRefresh
* 在 Page 中定义 onPullDownRefresh 钩子函数,到达下拉刷新条件后，该钩子函数执行，发起请求方法
* 请求返回后，调用 wx.stopPullDownRefresh 停止下拉刷新

参考 [这里](https://link.zhihu.com/?target=https%3A//juejin.im/post/5a781c756fb9a063606eb742" \t "_blank)

**★★ bindtap 和 catchtap 的区别是什么**

相同点：首先他们都是作为点击事件函数，就是点击时触发。在这个作用上他们是一样的，可以不做区分

不同点：他们的不同点主要是bindtap是不会阻止冒泡事件的，catchtap是阻止冒泡的

**★★★★ 简述微信支付的业务流程**

* 步骤1：用户在商户APP中选择商品，提交订单，选择微信支付。
* 步骤2：商户后台收到用户支付单，调用微信支付统一下单接口。参见[统一下单API](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=9_1" \t "_blank)。
* 步骤3：统一下单接口返回正常的prepay\_id，再按签名规范重新生成签名后，将数据传输给APP。参与签名的字段名为appid，partnerid，prepayid，noncestr，timestamp，package。注意：package的值格式为Sign=WXPay
* 步骤4：商户APP调起微信支付。api参见本章节[app端开发步骤说明](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=8_5" \t "_blank)
* 步骤5：商户后台接收支付通知。api参见[支付结果通知API](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=9_7" \t "_blank)
* 步骤6：商户后台查询支付结果。，api参见[查询订单API](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=9_2" \t "_blank)（查单实现可参考：[支付回调和查单实现指引](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=23_9&index=1" \t "_blank)） 详情参考[这里](https://pay.weixin.qq.com/wiki/doc/api/app/app.php?chapter=8_3" \t "_blank)

**★★★ 什么是小程序自定义组件样式隔离,他有哪几种隔离模式？**

* 默认情况下，自定义组件的样式只受到自定义组件 wxss 的影响。除非以下两种情况：
* app.wxss 或页面的 wxss 中使用了标签名选择器（或一些其他特殊选择器）来直接指定样式，这些选择器会影响到页面和全部组件。通常情况下这是不推荐的做法。
* 指定特殊的样式隔离选项 styleIsolation 。
* styleIsolation 选项从基础库版本 2.6.5 开始支持。它支持以下取值：
* isolated 表示启用样式隔离，在自定义组件内外，使用 class 指定的样式将不会相互影响（一般情况下的默认值）；
* apply-shared 表示页面 wxss 样式将影响到自定义组件，但自定义组件 wxss 中指定的样式不会影响页面；
* shared 表示页面 wxss 样式将影响到自定义组件，自定义组件 wxss 中指定的样式也会影响页面和其他设置了 apply-shared 或 shared 的自定义组件。（这个选项在插件中不可用。） 参考[这里](https://zhuanlan.zhihu.com/p/163190376" \t "_blank)

**★★★ 在小程序中又哪些方法让图片宽高比例保持不变？**

* 使用mode：widthFix

widthFix：宽度不变，高度自动变化，保持原图宽高比不变。 首先我们先设置image的mode为widthFix，然后给图片加一个固定rpx的宽度，比如：730rpx。 这样图片也可以自适应了。。因为小程序的rpx本身就是一个自适应显示的单位

* 使用mode属性为aspectFit,并给图片一个固定rpx宽高
* <image src="/images/1.png" class="img" mode="widthFix"></image>
* 使用bindload绑定函数动态自适应

详情参考[这里](http://www.qianduan8.com/1005.html" \t "_blank)

**★★ 小程序组件传参(父子，子父)**

* 父传子

自定义属性，属性的值来自声明式变量 子组件通过props接收

* 子传父

自定义事件，父组件通过事件接收子组件传过来的值 自定义封装组件的例子：

//父组件使用

//接收子组件数据的事件

ontogg(ele) {

console.log('接收子组件发过来的数据',ele)

}

//自定义横向滚动条

let { box, imgs, cates,toView ,flag,end} = this.state

<ScroLLview

flag={flag}

toView={toView}

cates={cates}

box={box}

onChange={(ele)=>this.ontogg(ele)}

/>

//子组件

export default props=>{

//接收父组件传过来的属性和事件

const {box,flag,toView,cates,onChange}=props

const renderSCview=()=>{

return(

cates.map(ele => (

//view要加id，和scroll-into-view相关联

<View

//传给父组件的事件，要传过去的数据放在onchange参数里面

onClick={() => onChange(ele)}

className={ele.cate === box ? 'mack on' : 'mack'}

key={ele.id}

id={'c'+ele.id}

>

{ele.cate\_zh}

</View>

))

)

}

return (

// 横向滚动条

<View style={{position:flag ? "fixed":"static"}} className="nav fix">

<ScrollView

className="scroll-view\_H"

scrollX

scrollIntoView={'c'+(toView-1)}

>

<View className="list">

{renderSCview()}

</View>

</ScrollView>

</View>

)

}

* 还可以参考[这里](https://zhuanlan.zhihu.com/p/267714541" \t "_blank)

**★★ 小程序组件生命周期**

* 组件的生命周期，指的是组件自身的一些函数，这些函数在特殊的时间点或遇到一些特殊的框架事件时被自动触发

| 生命周期 | 参数 | 描述 | 最低版本 | | -------- | ------------ | ---------------------------------------- | -------- | | created | 无 | 在组件实例刚刚被创建时执行 | 1.6.3 | | attached | 无 | 在组件实例进入页面节点树时执行 | 1.6.3 | | ready | 无 | 在组件在视图层布局完成后执行 | 1.6.3 | | moved | 无 | 在组件实例被移动到节点树另一个位置时执行 | 1.6.3 | | detached | 无 | 在组件实例被从页面节点树移除时执行 | 1.6.3 | | error | Object Error | 每当组件方法抛出错误时执行 | 2.4.1 |

参考[这里](https://blog.csdn.net/weixin_41829477/article/details/104297088" \t "_blank)

**★★ 小程序页面生命周期**

* 1、小程序注册完成后，加载页面，触发onLoad方法，一个页面只会调用一次。
* 2、页面载入后触发onShow方法，显示页面，每次打开页面都会调用一次。
* 3、首次显示页面，会触发onReady方法，渲染页面元素和样式，一个页面只会调用一次。
* 4、当小程序后台运行或跳转到其他页面时，触发onHide方法。
* 5、当小程序有后台进入到前台运行或重新进入页面时，触发onShow方法。
* 6、当使用重定向方法wx.redirectTo(OBJECT)或关闭当前页返回上一页wx.navigateBack()，触发onUnload

参考[这里](https://zhuanlan.zhihu.com/p/28872497" \t "_blank)

**★★ 小程序怎么样实现路由传参**

* wx.navigateTo （非 tabBar 的页面的路径，能返回来，只能有五层）
* wx.navigateTo({
* url: '../placeIntroduce/placeIntroduce?placeId=2',//在路径后面拼接
* })
* 另外一个页面通过onload(options)这个生命周期获取到placeId
* wx.redirectTo （非 tabBar 的页面的路径，不能返回来，不限层）
* wx.redirectTo （非 tabBar 的页面的路径，不能返回来，不限层）
* wx.redirectTo ({
* url: '../placeIntroduce/placeIntroduce?placeId=2',//在路径后面拼接

参考[这里](https://blog.csdn.net/weixin_30276935/article/details/97401135" \t "_blank)

**★★★ 小程序中的路由跳转switchTab navigateTo redirectTo的区别**

* wx.navigateTo()：保留当前页面，跳转到应用内的某个页面。但是不能跳到 tabbar 页面
* wx.redirectTo()：关闭当前页面，跳转到应用内的某个页面。但是不允许跳转到 tabbar 页面
* wx.switchTab()：跳转到 abBar 页面，并关闭其他所有非 tabBar 页面
* wx.navigateBack()关闭当前页面，返回上一页面或多级页面。可通过 getCurrentPages() 获取当前的页面栈，决定需要返回几层
* wx.reLaunch()：关闭所有页面，打开到应用内的某个页面

**★★★ 小程序tabbar实现原理**

* 要实现tabbar的导航条其实很简单，我们要实现全局的tabbar只需要在app.json文件中定义即可，局部的就在局部的tabbar文件中实现。

来看看app.json代码：

export default {

pages: [

'pages/index/index',

"pages/search/search",

'pages/news/news',

'pages/attention/attention',

'pages/collect/collect',

'pages/comment/comment',

'pages/my/my',

"pages/detail/detail",

"pages/chat/chat",

],

window: {

backgroundTextStyle: 'light',

navigationBarBackgroundColor: '#fff',

navigationBarTitleText: '小红书',

navigationBarTextStyle: 'black',

enablePullDownRefresh: true

},

tabBar: {

"list": [

{

"pagePath": "pages/index/index",

"text": "首页",

"iconPath": "assets/tabbar/1.png",

"selectedIconPath": "assets/tabbar/2.png"

},

{

"pagePath": "pages/news/news",

"text": "消息",

"iconPath": "assets/tabbar/news1.png",

"selectedIconPath": "assets/tabbar/news2.png"

},

{

"pagePath": "pages/my/my",

"text": "我的",

"iconPath": "assets/tabbar/3.png",

"selectedIconPath": "assets/tabbar/4.png"

}

]

},

permission: {

"scope.userLocation": {

"desc": "为了更好的为你服务，请允许访问你的地址"

}

}

}

参考[这里](https://www.cnblogs.com/izhaofu/p/6278589.html" \t "_blank)

**★★★ 小程序性能为什么那么好，为什么能做到即用即走的效果**

* 因为是轻量级的，代码包体积限制在2M以内，如果超过2M还可以进行分包，提高性能优化
* 运行在微信端,很多功能只需要使用API来实现，就可以实现跟APP的一样的功能
* 因为他的双线程的模型的原因，让小程序可以快速的一下将数据渲染出来呈现在用户的面前
* 是基于卫星或者支付宝这样的宿主环境的，微信客户端提供双线程去执行wxml, wxss, js文件

参考[这里](https://blog.csdn.net/xfy196/article/details/108232592" \t "_blank)

**★★ 如何自定义小程序的navigationBar**

* 思路

隐藏原生样式 获取胶囊按钮、状态栏相关数据以供后续计算 根据不同机型计算出该机型的导航栏高度，进行适配 编写新的导航栏 引用到页面

参考[这里](https://zhuanlan.zhihu.com/p/117244248" \t "_blank)

**★★ 说说小程序中wx:if和hidden的区别**

相同点：

wx:if 与 hidden 都用来控制小程序元素的显示

不同点

wx:if：

* 1、条件为 true 时显示
* 2、当元素显示时渲染
* 3、元素变为不显示时销毁元素

hidden：

* 1、条件为 false 时显示
* 2、当元素显示时渲染
* 3、元素变为不显示时保留元素
* 4、相当于使用了dispaly

总结： 1、当元素频繁切换是否显示时使用 hidden ，因为 wx:if 会频繁地销毁渲染元素 2、当元素不频繁切换是否显示时使用 wx:if，因为 wx:if 会避免页面加载时渲染过多，导致页面加载缓慢

# HTML + CSS 面试题

1.\*\*父元素和子元素宽高不固定 , 如何实现水平垂直居中

- 弹性盒模型

- 父元素设置 : display:flex; justify-content:center; align-items:center

- 额外的骚操作 : 父元素设置弹性盒display:flex; 子元素可以设置margin:auto; 实现垂直水平居中

- 父元素设置 : position : relative

- 子元素设置 : position : absolute; top:50%;left:50%;trans

form:translate(-50%,-50%)

2.\*\*分别实现色子中“一点”和“三点”的布局。

连接：

/\* 一点布局：原理就是将单个子元素垂直水平居中，利用弹性盒模型(display:flex)即可 \*/

父元素设置 : display:flex; justify-content:center; align-items: center

/\* 三点布局: 同样是使用弹性盒模型(display:flex),只不过这次需要用到其他属性 \*/

父元素设置 : display:flex; justify-content:space-between;

子元素设置 : .child:nth-child(2){

align-self:center;

}

.child:nth-child(3){

align-self:flex-end;

}

3.\*\*简述选择器 ~ 和 + 的区别。

~ 选择器的作用 :

1.选取紧跟当前符合条件元素后面的同级元素

2.可以匹配多个

+ 选择器的作用 :

1.选择紧跟在当前符合条件元素后面的同级元素

2.只能匹配一个

例:<div class="box"></div>

<p class="one"></p>

<span class="two"></span>

<p class="three"></p>

<span class="four"></span>

.box ~ p : 可以选中box下的所有p元素,即是 one 和 three

.box + span : 则匹配选中box相邻下符合条件的第一个元素,即是 two

4.\*\*简述box-sizing的有效值以及所对应的盒模型规则。

/\* box-sizing值 \*/

box-sizing: content-box/border-box/inherit

box-sizing属性用于更改，用于计算元素的高度和高度默认的CSS盒子模型，可以使用此属性来模拟不支持CSS盒子模型规范的浏览器行为。

\*\*content-box:

1.box-sizing的默认属性

2.是CSS中规定的宽度高度的显示行为

3.在CSS中定义的宽度和高度就对应到元素的内容框(即元素容器本身)

4.在CSS中定义的宽度和高度之外绘制元素的内边框和边框(即在元素本身上增加内边框和边框，容器占据空间大小计算方式:content(容器宽高)+margin(外边距)+border(边框))

\*\*border-box:

1.在CSS中微元素设定的宽度和高度就决定了元素的边框值

2.元素在设置内边距和边框是在已经设定好的宽度和高度之内进行绘制

3.CSS中设定的宽度和高度减去边框和内间距才能得到元素内容所占的实际宽度和高度(容器占据空间大小计算方式:content(容器宽高)+padding(内边距)+border(边框)\*\*inherit)

4.规定元素是从父容器继承box-sizing的属性值

5.\*\*\*html中元素的margin是否会叠加(合并)？如何解决？

/\* 会叠加 \*/

问题详解1: flex布局对子元素的影响

1.子元素的float、clear和vertical-align属性将会失效

2.解决了margin传递、重叠(叠加)问题

问题详解2: flex布局的margin传递叠加问题主要有以下两种

1.父子间的margin，会由子级传递到父级

——解决方法: margin传递的产生的原因是父级的高度没有被自动撑开，所以在父级增加属性: overflow:auto 即可解决

2.兄弟间的margin值会重复叠加

——解决方法: 浏览器为了保证列表的整齐，上下margin产生了叠加，不能直接解决。只能通过减少一个margin的方式。例如:margin-top:100px;

margin-bottom:0px的方式解决。

6.\*\*简述aligin-items和aligin-content的区别。

align-items: 可以应用于所有的flex容器，它的作用是设置flex子项在每个flex行的交叉轴上的默认对齐方式(相对Y轴)

align-content: 只适用于多行的flex容器，在使用前需在flex容器设置flex-wrap:wrap;表示子元素超出换行；align-content 它的作用是当flex容器在交叉轴上有多余的空间时，将子项作为一个整体进行对齐。

7.*\*简述data-*属性的用法(如何设置，如何获取)，有何优势？

data-\*定义:

1.是用于储存页面或应用程序的私有自定义数据

2.赋予我们在所有html元素上嵌入自定义data属性的能力

data-\*用法:

1.属性名不应该包含任何大写字母，并且在前缀“data-”之后必须有至少一个字符

2.属性值可以是任意字符串

3.一个元素可以拥有任意数量的data属性

4.data属性无法储存对象，如需储存，可以通过对象序列化

data-\*如何设置、获取:

1.如何设置

通过javascript内置的setAttribute('data属性名'，'新内容')即可设置。

通过改数据类型的(dataset) API设置的data值，IE10以上才支持；

var button = document.queryselector('button')

button.dataset.data属性名 = '新内容'；这里的data属性名是值data-后面的名字。

2.如何获取

通过javascript内置的getAttribute('data属性名')即可获取

通过该数据类型的(dataset)API设置的data值，IE10以上才支持

var button = document.queryselector('button')

data = button.dataset.data属性名；这里的data属性名是指data-后面的名字。

data-\*优势:

1.其储存的自定义数据能够被页面的JavaScript利用，可以创建更好的用户体验。

2.可以通过JavaScript来构造数据、填充数据

3.代码体积小、较为灵活

4.解决网站的外观和实用性之间产生的冲突

9.\*什么是标准文档流

\*\*\* 标准文档流指的是元素排版布局过程中，元素会默认自动从左往右，从上往下的流式排列方式。当前面内容方式了变化，后面的内容位置也会随着方式变化。

\*\*\* HTML就是一种标准文档流文件

\*\*\* 简单来说就是各种布局属性在html中所显示的效果，

10.\*z-index是什么？在position的值什么时候可以触发？

z-index : 指的是一个元素在当前文档页面定位时重叠层显示的层级等级，默认为0，数值不限，越大显示层级越高。

触发机制 : 当position的值设置为absolute、relative和fixed是才能触发。

11.\*\*CSS3如何实现圆角？

border-radius属性

1.四个值 : 左上角，右上角，右下角，左小角。

2.三个值 : 左上角，右上角和左小角，右下角。

3.两个值 : 左上角和右下角，右上角和左小角。

4.一个值 : 四个圆角值相同。

只需要使用border-radius将四个角设置为相适应的尺寸即可实现圆角

12.\*\*localstorage缓存 和 sessionStorage会话缓存

(1)localhostorage缓存，将数据储存在本地客户端，只有用户手动清除才能清除缓存

API : 1.localhostorage.setItem(key,value),键值对的形式缓存

2.localhostorage.getItem(key),根据键名来缓存值

3.localhostorage.length,获取总缓存数量

(2)sesssionStorage 会话缓存，会话机制是指从打开浏览器开始访问页面的时候，到关闭这个页面的过程成为一个会话，sessionStorage储存的数据会随着页面关闭而销毁

API : 1.sessionStorage.setItem(key,value),localStorage是以键值对的形式创建的；

2.sessionStorage.getItem(key),根据键名来获取缓存的值；

3.SessionStorage.length;获取总共缓存值得数量，localStor

age返回的是个对象

13.\*\*CSS3新增伪类有哪些？

常用的伪类:

1. :link 选择所有未访问的链接

2. :visited 选择所有访问过的链接

3. :active 选择正在活动的链接(或理解为鼠标点击瞬间效果)

4. :hover 鼠标放到链接后的状态

5. :focus 选择元素输入后具有焦点

6. :bedore 在元素之前插入内容

7. :after 在元素之后插入内容

16.\*\*什么叫优雅降级和渐进增强?

渐进增强 :

1.针对低版本浏览器进行构建页面，保证最基本的功能，然后在针对高级浏览器进行效果、交互等改进和追加功能达到更好的用户体验。

2.渐进增强观点认为应关注于内容本身。内容是我们建立网站的诱因。有的网站展示它，有的则收集它，有的寻求，有的操作，还有的网站甚至会包含以上种种，但相同点是它们全都涉及到内容。这使得“渐进增强”成为一种更为合理的设计范例。

优雅降级 :

1.一开始就构建完整的功能，然后在针对低版本浏览器进行兼容

2.优雅降级观点应该针对那些最高级、最完善的浏览器来设计网站。而将那些被认为“过时”或有功能缺失的浏览器下的测试工作安排在开发周期的最后阶段，并把测试对象限定为主流浏览器的前一个版本。

区别 :

1.优雅降级是从复杂的现状开始，并视图减少用户体验的供给

2.渐进增强则是从一个非常基础的，能够其作用的版本开始的，并不断的扩充，以适应未来环境的需要

3.降级(功能衰减)意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带

17.\*\*\*移动端适配怎么做？

方法一 : @media 媒体查询，通过查询设备的宽度来执行不同的 css 代码，最终达到界面的配置

方法二 : flex弹性布局

方法三 : rem + viewport 缩放，屏幕宽度设定 rem 值，需要适配的元素都使用 rem 为单位，不需要适配的元素还是使用 px 为单位

18.\*\*请问苹果原生浏览器中默认样式如何清除，例如button，input默认样式？

清除苹果默认样式: css样式中加入 input，textarea,button {-webkit-appearance:none;border-radius:0px;border:none}

input、button默认样式: input[type="button"],input[type="submit"],input[type="reset"]{-webkit-appearance:none}

19.\*CSS清除浮动的方法

1.在标签尾部添加空块级标签，设置样式属性为 : clear:both; 缺点: 如果页面浮动布局多，就要增加很多空div，不利于页面的优化。

2.父级定义伪类after和zoom， .box:after{display:block;clear:

both;content:"";visibility:hidden;height:0;} .box{zoom:1}

3.简单粗暴，父级设置overflow:hidden,缺点是不能和position配合使用

4.直接给父元素单独设置高度(height)；缺点:只适合高度固定的布局，要给出精确的高度，如果高度和父级div不一样时，会产生问题。对于响应式布局会有很大的影响。

20.\*\*PC端常用的布局方法

1.利用float+overflow实现

左右定宽度，分布向两边浮动，中间如果没设置溢出处理默认宽度为100%，设置后就会截取两边的宽度从而实现中间自适应

2.flex布局

通过flex相关的属性进行适配性的调整布局

3.Table表格布局

4.float+margin实现三列布局

5.定位absolute实现中间自适应

6.Grid网格布局

7.圣杯布局

8.双飞翼布局

9.等高布局

21.\*\*布局左边20%，中间自适应，右边200px，不能用定位

布局: <div class ="box">

<div class="left">left</div>

<div class="right">right</div>

<div class="content">content</div>

</div>

样式:\*,html,body{

margin:0;

padding:0;

}

.box{

width:1500px;

height:500px;

background-color:rgb(215.221.221);

margin:0 auto;

}

.left{

width:20%;

height:200px;

background-color:chocolate;

float:left;

}

.right{

width:200px;

height:200px;

float:right;

background-color:cornflowerblue;

}

.content{

overflow:hidden;

height:200px;

background-color:darkblue;

}

22.\*\*行内元素和块级元素？img算什么？行内元素怎么转化为块级元素？

行内元素:1.无法设置宽高；

2.对margin仅设置左右有效，上下无效；

3.padding上下左右有效；不会自动换行；

块级元素:1.可以设置宽高

2.margin和padding的上下左右均对其有效

3.超出当前行会自动换行

4.多个块状元素标签写在一起，默认排列方式为从上到下

img:属于行内块元素(inline-block)，即有行内元素的属性也有块级元素的属性

元素之间的转化可以通过设置样式:display:block/inline/inline-block来改变自身的元素属性

24.\*\*什么是CSS3 transform?animation?区别是什么？

transform:1.transform 属性向元素应用 2D 或者 3D转换。该属性允许我们对元素进行旋转、缩放、移动或倾斜。

2.transform 属性是静态属性，需要配合 transition 和 animation 才能展现动画效果。你可以把它看成跟left 、top等属性一样，只是一个静态样式而已。

animation:一个css3的动画属性，需要配合@keyframes使用

区别:1.transition是css中检测指定属性变化进行自动补间动画的属性。

2.animation是先指定好动画过程中的关键帧属性，进行动画的属性。

28.\*如何理解html结构语义化？

为什么要语义化：

a.为了在没有css的情况下，页面也能呈现出很好的内容结构、代码结构

b.用户体验：列如title、alt用于解释名词或者解释图片信息的标签尽量填写有含义的词语，label标签的活用

c.有利于SEO：和搜索引擎建立良好沟通，有助于爬虫抓取更多有效信息：爬虫依赖于标签来确定上下文和各个关键字的权重

d.方便其他设备解析以有意义的方式来渲染页面

e.便于团队开发和维护，语义化更具可读性，遵循W3C标准的团队都遵循这个标准，可以减少差异化

41.\*\*a标签中active hover link visited 正确的设置顺序是什么？

a标签的设置顺序：

1.link ， 链接平常的状态

2.hover ， 鼠标放置在链接上显示的样式

3.active ， 链接被按下的样式

4.visited ， 链接被访问过后的状态

42.\*\*\*手机端上图片长时间点击会选中图片，如何处理？

img{pointer-events:none},禁止事件，但会把整个标签的事件都禁用掉，不建议使用

img{-webkit-user-select:none},用户选中状态

推荐：

img{

-webkit-touch-callout:none;//触摸

-webkit-user-select:none;

-moz-user-select:none;

-ms-user-select:none;

user-select:none;

}

44.\*\*常见的视频编码格式有几种?视频格式有几种？

常见的视频编码格式：H264,VP8,AVS,RMVB,WMV,QuickTime(mov)

视频格式有：MPEG,AVI,nAVI,ASF,MOV,3GP,WMV,DivX，XviD，RM，RMVB,FLV/F4V

45.\*\*canvas在标签上设置宽高和在style中设置宽高有什么区别？

在canvas标签上设置宽高，canvas画布发生的变化不会影响到画布内容，及画布内容不会发生改变

相反的，style样式中设置宽高则会影响到画布内容的改变

47.\*\*解释css3的怪异盒模型和弹性盒模型

在标准模式下的盒模型:

盒子总宽度=width+padding+margin

盒子总高度=height+padding+border+margin

在怪异模式下的盒模型:

盒子的总宽度和高度是包含内边距padding和边框border宽度在内的，盒子总宽度/高度=width/height + margin = 内容区宽度/高度 + padding + border + margin;

# JavaScript 面试题

4. 说说你对 JavaScript 的作用域的理解。什么是作用域链？

在 JavaScript 中有两种作用域类型:

1、局部作用域:只能在函数内部访问它们

2、全局作用域:网页的所有脚本和函数都能够访问它

JavaScript 拥有函数作用域:每个函数创建一个新的作用域

作用域决定了这些变量的可访问性(可见性)

函数内部定义的变量从函数外部是不可访问的(不可见的)

作用域链:

1、当查找变量的时候，会先从当前上下文的变量对象中查找。

2、如果没有找到，就会从父级执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。

3、这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

5.\*\* 解释下 let 和 const 的块级作用域

*/\* -------- let ---------- \*/*

1.let声明的仅在块级作用域内有效

2.let不会发生变量提升的现象，所以一定要在定义后使用，否则报错

3.暂时性死区:只要块级作用域内存在let命令，它所声明的变量就绑在这个区域，不再受外部影响

4.不允许重复声明，let不允许在相同作用域内，重复声明同一个变量

*/\*---------- const --------\*/*

1.声明一个只读的常量。一旦声明，常量的值就不能改变

2.一旦声明，就要立即初始化，否则也报错

3.const命令声明的常量也不提升，同样存在暂时性死区，只能在声明的位置后使用

4.也不可以重复使用

6. 说说你对执行上下文的理解

1.全局执行上下文:

全局执行上下文只有一个，也就是我们熟知的 window 对象，我们能在全局作用域中通过 this 直接访问到它

2.函数执行上下文:

函数执行上下文可存在无数个，每当一个函数被调用时都会创建一个函数上下文

需要注意的是，同一个函数被多次调用，都会创建一个新的上下文

7.\*\*\* 对闭包的看法，为什么要用闭包?说一下闭包的原理以及应用场景？闭包的this指向问题？

闭包的作用:

1.在外部访问函数内部的变量

2.让函数内的局部变量可以一直保存下去

3.模块化私有属性和公共属性

闭包的原理:

全局变量生存周期是永久，局部变量生存周期随着函数的调用结束而销毁

闭包就是 : 在函数中定义且成为该函数内部返回的函数的自由变量的变量，该变量不会随着外部函数调用结束而销毁

(注:不光是变量，函数内声明的函数也可以形成闭包)

当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这是就产生了闭包

8.\*\*\* 简述闭包的问题以及优化

闭包的缺点:占用内存空间 大量使用闭包会造成 栈溢出

由于闭包会一直占用内存空间，直到页面销毁，我们可以主动将已使用的闭包销毁:将闭包函数赋值为 null 可以销毁闭包

9.\*\*\* 如何确定 this 指向?改变 this 指向的方式有哪些?

this指向:

1、全局上下文(函数外)

无论是否为严格模式，均指向全局对象。注意:严格模式下全局对象为 undifined

2、函数上下文(函数内)

默认的，指向函数的调用对象，且是最直接的调用对象:

简单调用，指向全局对象注意:严格模式下全局对象为 undifined ，某些浏览器未实现此标准也可能是 window

改变 this 指向的方式:

(1) new 关键字改变 this 指向

(2) call( )

(3) apply( )

(4) bind( )

10.\*\*\* 介绍箭头函数的 this

由于箭头函数不绑定 this，它会捕获其所在(即定义的位置)上下文的 this 值，作为自己的 this 值

1、所以 call() / apply() / bind() 方法对于箭头函数来说只是传入参数，对它的 this 毫无影响

2、考虑到 this 是词法层面上的，严格模式中与 this 相关的规则都将会被忽略

作为方法的箭头函数 this 指向全局 window 对象，而普通函数则指向调用它的对象

16.***11.*** 谈一下你对原型链的理解

原型链:

因为每个对象和原型都有原型，对象的原型指向原型对象

而父的原型又指向父的父，这种原型层层连接器起来就构成了原型链

13. 你对事件循环有了解吗？说说看

Event Loop(事件循环)中，每一次循环称为 tick，每一次 tick 的任务如下:

\* 执行栈选择最先进入队列的宏任务(通常是 script 整体代码)，如果 有则执行

\* 检查是否存在 Microtask,如果存在则不停的执行，直至清空 microtask 队列

\* 更新 render(每一次事件循环，浏览器都可能会去更新渲染)

\* 重复以上步骤

16.\*\*\* 异步解决方案有哪些?

1、回调函数 callback

被作为实参传入另一个函数，并在该外部函数内被调用，用以来完成某些任务的函数。如 setTimeOut，ajax请求，readFile等。

function greeting(name){

alert('Hello' + name);

}

function processUserInput(callback){

var name = prompt('请输入你的名字');

callback(name);

}

processUserInput(greeting);

优点:解决了异步的问题

缺点:回调地狱:多个回调函数嵌套的情况，使代码看起来很混乱，不易于维护

2、事件分布订阅

当一个任务执行完成后，会发布一个事件，当这个事件有一个或多个“订阅者”的时候，会接收到这个事件的发布，执行相应的任务，这种模式叫发布式订阅模式。

document.body.addEventListener('click',function(){

alert('订阅了');

},false);

document.body.click();

优点:时间对象上的解耦

缺点:消耗内存，过度使用会使代码难以维护和理解

3、Promise

Promise 是es6提出的异步编程的一种解决方案

Promise 对象有三种状态:

pending:初始状态，既不是成功，也不是失败状态

fulfilled:意味着操作成功完成

rejected:意味着操作失败

promise 的状态只能从 pending 变成 fulfilled ，和 pending 变成 rejected,状态一旦改变，就不会在改变，且只有异步操作的结果才能改变promise的状态

let promise = new Promise(function (resolve,reject){

fs.readFile('./1.text','utf-8',function(err,data){

resolve(data)

})

})

promise.then(function(data){

consele.log(data)

})

优点:解决了回调地狱的问题，将异步操作以同步操作的流程表达出来

缺点:无法取消promise。如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。当处于Pending状态时u，无法得知目前进展到哪一个阶段(刚刚开始还是即将完成)。当执行多个Promise是，一堆 then 看起来也不友好

4、Generator

优点:没有了 Promise 的一堆then(),异步操作更像同步操作，代码更加清晰

缺点:不能自动执行异步操作，需要写多个next()方法，需要配合使用 Thunk函数和 Co模块才能自动执行

5、async/await

优点:内置执行器，比Generator操作更加简单。async/await比yield语义更清晰。返回值是Promise对象。可以用then指定下一步操作。代码更加整洁。可以捕获同步和异步的错误

17.\*\*\* async 和 await、promise 的区别 和 这两个的本质

18.\*\*\* 简述 aync await 的好处

1.async/await 最重要的好处是 同步编程风格

2.async/await 有本地浏览器支持。截止今天，所有主流浏览器 查看都完全支持异步功能

3.async关键字。它声明 getBooksByAuthorWithAwait( )函数返回值确保一个 promise，以便调用者安全调用tgetBooksByAuthorWithAwait( ).then(...)或await getBooksByAuthorWithAwait( )

26.\*\*\* bind 的作用是什么？

bind( )方法主要就是将函数绑定到某个对象

bind( )会创建一个函数，函数体内的 this 对象的值会被绑定到传入bind( )第一个参数的值

28.\*\*\* var let const 的区别、

1.var 声明的变量会挂载在 window 上，而let和const声明的变量不会

2.var 声明变量存在提升，let和const不存在变量提升

3.let 和 const 声明形成块作用域

4.同一作用域下 let 和 const 不能声明同名变量，而 var 可以

5.使用let/const 声明的变量在当前作用域存在暂存死区

6.const 一旦声明必须赋值，不能使用null占位，声明后不能在修改，如果声明的是复合类型数据，可以修改其属性

38.\*\*\* 简述同步和异步的区别

同步:

同步的思想:所有的操作都做完，才返回给用户。这样用户在线等待的事件太长，给用户一种卡死了的感觉(就是系统迁移中，点击了迁移，界面就不动了，但是程序还在执行，卡死了的感觉)。这种情况下，用户不能关闭界面，如果关闭了，即迁移程序就中断了

异步:

将用户请求放入消息队列，并反馈给用户，系统迁移程序已经启动，你可以关闭浏览器了。然后程序在慢慢第写入数据库。这就是异步。但是用户没有卡死的感觉，会告诉你，你的请求系统已经响应了。你可以关闭界面了

同步和异步本身是相对的:

同步就相当于是 当客户端发送请求给服务端，在等待服务器响应的请求时，客户端不做其他的事情。当服务器做完才返回到客户端。这样的话客户端需要一直等待。用户使用起来会有不友好

异步就是，当客户端发送给服务端请求时，在等待服务端响应的时候，客户端可以做其他的事情，这样节约了事件，提高了效率

存在就有其道理，异步虽然好，但是有些问题就要用同步用来解决，比如有些东西我们需要的是拿到返回的数据在进行操作的。这些是异步无法解决的

42.\*\*\* 简述下 Promise 对象

Promise 是异步编程的一种解决方案，比传统的解决方案(回调函数和事件)更合理强大

所谓 Promise 简单说就是一个容器，里面保存着某个未来才会结束的事件(通常是一个异步操作)的结果

Promise对象有以下2个特点:

1.对象的状态不受外界影响。Promise对象代表一个异步操作，有三种状态:Pending(进行中)、Resolved(已完成)和Rejected(已失败)。只有异步操作的结果，可以决定当前是哪一种情况，任务其他操作都无法改变这个状态

2.一旦状态改变，就不会再变，任何时候都可以得到这个结果。promise对象的状态改变，只有两种可能:从Pending变为Resolved；从pending变为Rejected

有了 Promise 对象，就可以把异步操作以不同操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供了统一的接口，是的控制异步操作更加容易

45.\*\* 简述一下面向对象的六法则

1.单一职责原则:一个类只做它该做的事情

2.开闭原则:软件实体应当对扩展开发，对修改关闭

3.依赖倒转原则:面向接口编程

4.接口隔离原则:接口要小而专，绝不能大而全

5.合成聚合复用原则:优先使用聚合或合成关系复用代码

6.迪米特法则:迪米特法则又叫最少知识原则，一个对象应当其他对象有尽可能少的了解(低耦合)

46.\*\*\* 谈谈垃圾回收机制方法以及内存管理

垃圾回收方式

(1) 标记清除

工作原理:是当变量进入环境时，将这个变量标记为“进入环境”。当变量离开环境时，则将其标记为“离开环境”。标记“离开环境”的就回收内存

(2) 引用计数

工作原理:跟踪记录每一个值被引用的次数。一旦没有引用，内存就直接释放了

内存管理

什么时候触发垃圾回收?

垃圾回收器周期性运行，如果分配的内存非常多，那么回收工作也会很艰巨，确定垃圾回收时间间隔就变成了一个值得思考的问题

1、合理的GC方案: (1)遍历所有可访问的对象 (2)回收已不可访问的对象

2、GC缺陷: (1)停止响应其他操作

3、GC优化策略: (1)分代回收 (2)增量GC

# VUE 面试题

HTTP 和 HTTPS 的区别

　超文本传输协议HTTP协议被用于在Web浏览器和网站服务器之间传递信息，HTTP协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，因此，HTTP协议不适合传输一些敏感信息，比如：信用卡号、密码等支付信息。

　　为了解决HTTP协议的这一缺陷，需要使用另一种协议：安全套接字层超文本传输协议HTTPS，为了数据传输的安全，HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。

**一、HTTP和HTTPS的基本概念**

　　HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

　　HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

　　HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

**二、HTTP与HTTPS有什么区别？**

　　HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

　　HTTPS和HTTP的区别主要如下：

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

**三、HTTPS的工作原理**

　　我们都知道HTTPS能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用HTTPS协议。



　客户端在使用HTTPS方式与Web服务器通信时有以下几个步骤，如图所示。

　　（1）客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。

　　（2）Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

　　（3）客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。

　　（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

　　（5）Web服务器利用自己的私钥解密出会话密钥。

　　（6）Web服务器利用会话密钥加密与客户端之间的通信。

1.\*\*\* 如何在vue的单文件组件里的样式定义全局css？

在 style 标签上不加上 scoped 的属性，默认为全局 css 样式

2.\*\* vue-router 3.1.0 新增的v-slot属性怎么用？

router-link 通过一个作用域插槽暴露底层的定制能力。这是一个更高阶的 API ，主要面向库作者，但也可以为开发者提供便利，多数情况下用在一个类似 NavLink 这样的自定义组件里

在使用 v-solt API 时，需要向 router-link 传入一个单独的子元素。否则 router-link 将会把子元素包裹在一个 span 元素内

3.\*\*\* 如何实现一个路径渲染多个组件？

可以通过命名视图(router-view),它允许同一个界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 router-view 没有设置名字，那么默认为 default。通过设置 components 即可同时渲染多个组件

4.\*\*\* 如何实现多个路径共享一个组件？

只需将多个路径的 component 字段的值设置为同一个组件即可。

5.\*\*\* 如何监测动态路由的变化

可以通过 watch 方法来对 $route 进行监听，或者通过导航守卫的钩子函数 beforeRouteUpdate 来监听它的变化

6.\*\*\* 对 MVC , MVP , MVVM 的理解

mvc 和 mvvm 其实区别并不大。都是一种设计思想。

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。

当 Model 频繁发生变化，开发者需要主动更新到 View 。

MVVM 是 Model-View-ViewModel 的缩写。mvvm 是一种设计思想。

1：Model 层代表数据模型，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 是一个同步 View 和 Model 的对象。

2：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。

3：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。

7.\*\* 说一下 $root , $parent , $refs

$root,$parent 都能访问父组件的属性和方法，区别在于如果存在多级子组件，通过 parent 访问得到的是它最近一级的父组件，通过 root 访问得到的是根父组件，通过在子组件标签定义 ref 属性，在父组件中可以使用 $refs 访问子组件实例

8.**\***你知道Vue响应式数据原理吗？Proxy 与 Object.defineProperty 优劣对比？

*// 响应式原理*

vue 的响应式数据原理实现主要是利用了 Object.defineProperty 的方法里面的 setter 与 getter 方法的观察者模式来实现，watcher 调用组件的 render 函数去生成虚拟 DOM 。在调用 render 的时候，就会需要用到 data 的属性值，触发 getter 函数，将当前的 watcher 函数注册进 sub 里。当 data 属性发生改变之后，就会遍历 sub 里所有的 watcher 对象，通知它们去渲染组件。

*// proxy的优势如下:*

Proxy 可以直接监听对象而非属性，可以直接监听数组的变化；

Proxy 有多达 13 种拦截方法，不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；

Proxy 返回的是一个新对象，我们可以只操作新的对象达到目的，而Object.defineProperty 只能遍历对象属性直接修改；

*// Object.defineProperty 的优势如下:*

兼容性好，支持 IE9，而 Proxy 存在浏览器兼容性问题，而且无法用 polyfill(垫片)来弥补

9. Composition API 的出现带来那些新的开发体验，为啥需要这个？

1: 在Composition API 中是根据逻辑相关组织代码的，提高可读性和可维护性，类似于 react 的 hook 写法。

2: 更好的重用逻辑代码，避免发生命名冲突或者关系不清。

3: 解决在生命周期函数经常包含不相关的逻辑，但又不得不把相关逻辑分离到了不同方法中的问题，如在 mounted 中设置定时器，但需要在 destroyed 中来清除定时器，将同一功能的代码拆分到不同的位置，造成后期代码维护的苦难。

10.\*\*\* 什么情况下使用Vuex

如果应用够简单，最好不要使用 vuex ， 一个简单的 store 模式即可，需要构建一个中大型单页面应用时，使用 vuex 能更好的在组件外部管理状态

11.\*\*\* Vuex可以直接修改state的值吗？

可以直接修改，但是极其不推荐，state 的修改必须在 mutation 中修改，否则无法被 devtool 所监测，无法监测数据的来源，无法保存状态快照，也就无法实现时间漫游/回滚之类的操作

12. 为什么 Vuex 的 mutation 不能做异步操作

vuex 中所有的状态更新的唯一途径都是 mutation，异步操作通过 Action 来提交 mutation 实现，这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够使用一些工具帮助我们更好地了解我们的应用。每一个 mutation 执行完成后都会对应到一个新的状态变更，这样 devtools 就可以打个快照存下来。如果 mutation 支持异步操作，无法被 devtools 所监测，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难

13.\*\*\* v-model和vuex有冲突吗？

有冲突

在严格模式中使用vuex，当用户输入时，v-model会试图直接修改属性值，但这个修改不是在 mutation 中修改的，所以会抛出一个错误

当需要在组件中使用 vuex 中的state是，有2中解决方案:

1、在 input 中绑定 value(vuex中的state)，然后监听 input 的 change 或者 input 事件，在事件回调中调用 mutation 修改 state 的值

2、使用带有 setter 的双向绑定计算属性

14.\*\*\* 解释单向数据流和双向数据绑定

对于 vue 来说，组件之间的数据传递具有单向数据流这样的特性称为单向数据流，单向数据流方式使用一个上传数据流和一个下传数据流进行双向数据通信，两个数据流之间相互独立，单向数据流指只能从一个方向来修改状态

而双向数据绑定即为当数据发生变化的时候，视图也相应的发生变化，当视图发生变化的时候，数据也会跟着同步变化，两个数据流之间互为影响

15.\*\* Vue如何去除url中的 #

将路由模式改为history

16. vue-router 路由实现原理

我们使用 vue 来开发单页面应用(SPA)时，通常都会使用 vue-router 来实现页面路由跳转。单页面应用采用前端路由系统，通过改变 URL，在不重新请求页面的情况下，更新页面视图

router.push 添加新路由

router.replace 替换当前路由

router.go 跳转到指定索引路由

router.back 返回上一个路由

router.forward 跳转下一个路由

17.\*\*\* $route 和 $router 的区别

$route 用来获取路由的信息的，它是路由信息的一个对象，里面包含路由的一些基本信息。$router 主要是操作路由的，它是 VueRouter 的实例，包含了一些路由的跳转方法，钩子函数等。

18.\*\* 对比jQuery , Vue有什么不同

jQuery 专注于视图层，通过直接操作 DOM 去实现页面的一些逻辑渲染；Vue 专注于数据层，通过数据的双向绑定，最终表现在 DOM 层面，减少了 DOM 操作。Vue 使用了组件化思想，使项目子集职责清晰，提高了开发效率，方便重复利用，便于协同开发

19.\*\*\* Vue 中怎么自定义指令

通过 directive 来自定义指令，自定义指令分为全局指令和局部指令，自定义指令也有几个钩子函数，常用的有 bind 和 update，当 bind 和 update 触发相同行为，而不关心其他的钩子时可以简写

20.\*\*\* Vue 中怎么自定义过滤器

通过 filter 来定义过滤器，过滤器分为全局过滤器和局部过滤器，过滤器的主体为一个普通的函数，来对数据进行处理，可以传递参数。当有局部和全局两个名称相同的过滤器时候，会以就近原则进行调用，即:局部过滤器优先于全局过滤器被调用

21.\*\*\* Vue 等单页面应用的优缺点

*// 优点*

1、单页应用的内容的改变不需要重新加载整个页面，web应用更具有响应性

2、单页应用没有页面之间的切换，就不会出现“白屏现象”也不会出现假死和“闪烁”现象

3、单页应用相对服务器压力小，服务器只用出数据就可以，不用管展示逻辑和页面合成，吞吐能力会提高几倍

4、良好的前后端分离，后端不再负责模块渲染，输出页面工作，后端API通用化，即同一套后端程序代码

*// 缺点*

1、首次加载耗时较多

2、SEO 问题，不利于百度，360等搜索引擎收录

3、容易造成CSS命名冲突

4、前进，后退，地址栏，书签等都需要进行管理，页面的复杂度很高，需要一定的技能技术和开发水平

22.\*\*\* Vue-router 使用 params 和 query 传参有什么区别

1、query 要用 path 来引入，params 要用 name 来引入

2、query 更加类似于我们ajax中get传参，params则类似于post，query在浏览器地址栏中显示参数，params在浏览器地址栏中不显示参数

3、params 是路由的一部分，必须要有。query 是拼接在 url 后面的参数，没有也没关系

4、params、query不设置也可以传参，params不设置的时候，刷新页面或者返回参数会丢失

23.\*\*\* Vue中keep-alive的作用

keep-alive 是 vue 内置的一个组件，可以使被包含的组件保留状态或者避免重新渲染。一旦使用 keep-alive 包裹组件，此时mouted，created等钩子函数只会在第一次进入组件时调用，当再次切换回来时将不会调用。此时如果我们还想在每一次切换时做一些事情，就需要用到另外的周期函数 actived 和 dectived ，这两个钩子函数只有被 keep-alive 包裹后才会调用

24. Vue 如何实现单页面应用

通常的 url 地址由以下内容构成:协议名 域名 端口名 路径 参数 哈希值，当哈希值改变，页面不会发生跳转，单页面应用就是利用了这一点，当哈希值改变时通过 location.hash 就能获得相应的哈希值，然后就能调用相应的页面

25. 说明一下封装vue组件的原则和方法

封装出来的通用组件叫做子组件，引用通用组件的界面叫做父组件

封装出来的组件必须具有高性能，低耦合的特性

数据从父组件传入:子组件本身不要生成数据，如果需要生成数据，只能在组件内部进行使用，不要传递出去。父组件使用 props 对子组件进行传参。对于通过 props 传入的参数，不建议对其进行操作，因为会同时修改父组件里面的数据(如果去修改的话，控制台中也会报错)，如果需要修改数据，可以传递给父组件，让父组件去修改

26.\*\*\* 说出至少4中Vue当中的指令和它的用法？

v-if(判断是否隐藏，用来判断元素是否创建)

v-show(元素的显示隐藏，类似CSS中的display的block和hidden)

v-for(把数据遍历出来)

v-bind(绑定属性)

v-model(实现双向绑定)

27.\*\*\* Vuex是什么？怎么使用？描述使用它登录功能的流程？

vue 框架中状态管理。在main.js引入 store 注入。新建一个目录store，...，export。

场景有:单页应用中，组件之间的状态。音乐播放，登录状态，加入购物车

28.\*\*\* 谈谈vue生命周期函数？vue的实例选项对象？谈谈动态数据绑定的原理是什么？如何切换路由。有哪些方法传值到下一个页面，区别是什么？

生命周期函数总共分为8个阶段，beforecreate,created, beforeMount,Mounted, beforeUpdate,Updeated, beforeDestory,destoryed。

创建前/后: 在 beforeCreate 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destory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。

29.\*\*\* Vue-loader解释一下

解析和转换 .vue 文件，提取出其中的逻辑代码 script、样式代码 style、以及 HTML 模板 template，再分别把它们交给对应的 Loader 去处理。

30.\*\* 用vue封装个组件，可以自定义里面有多少个input

31.\*\*\* 用过插槽吗？用的是具名插槽还是匿名插槽

都用过。插槽相当于预留了一个位置，可以将我们书写在组件内的内容放入，写一个插槽就会将组件内的内容替换一次，两次则替换两次。为了自定义插槽的位置我们可以给插槽取名，它会根据插槽名来插入内容，一一对应。

32.\*\*\* Vue-cli运行的命令行

npm run serve

33.\*\*\* Vue 路由守卫

vue-router 提供的导航守卫主要用来对路由的跳转进行监控，控制它的跳转或取消，路由守卫有全局的，单个路由独享的，或者组件级的。导航钩子有3个参数:

1、to:即将要进入的目标路由对象

2、from:当前导航即将要离开的路由对象

3、next:调用该方法后，才能进入下一个钩子函数(afterEach)

34.\*\* Vue中如何实现子组件内的css样式名在项目中绝对唯一性

在style标签加上scoped属性

35.**\*** Vue3中的双向数据绑定 proxy

proxy 相当于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，我们可以这样认为，proxy 是 object.defineProperty 的全方位加强版，它解决了之前 defineProperty 无法监听到数组变化的缺点

36.**\*** Vue 和 React中diff算法区别

节点对比:vue 当节点元素相同，但是 ClassName 不同，认为是不同类型的元素，删除重建，而react认为是同类型节点，只是修改节点属性

列表对比:vue 采用的是两端到中间比对的方式，而react采用的是从左到右依次对比的方式。总体上，vue的方式比较高效

37.\*\*\* 请你说一下 Vue 中 create 和 mount 的区别

create 为组件初始化阶段，在此阶段主要完成数据观测(data observe),属性和方法的运算,挂载阶段还没开始，此时还未生成真实的 DOM，也就无法获取和操作 DOM 元素。而 mount 主要完成从虚拟 DOM 到真实 DOM 的转换挂载，此时 html 已经渲染出来了，所以可以直接操作dom节点。

38. Object.defineProperty 有什么缺点

1、无法监听到数组下标的变化，导致通过数组下标添加元素或删除元素时，不能实时响应

2、只能劫持对象的属性，从而需要导致对每个对象，每个属性进行遍历，如果层级比较深，还需要深度遍历，使用递归。

39. axios 是什么？怎么使用？描述使用它登录功能的流程？

axios 是请求后台资源的模块。通过 cnpm i axios --save 来安装。使用axios时，在大多数情况下我们需要封装拦截器，在实现登录的过程中我们一般在请求拦截器中来加入 token，在响应请求器中通过判断后端返回的状态码来对返回的数据进行不同的处理

40.**\*** Vue 中父页面(组件)与子组件相互传参

vue 中父页面(组件)与子组件相互传参:

父页面(组件)给子组件传参:

1、在父页面(组件)上自定义属性，在子组件上使用 props 属性接受

2、props 属性写成一个数组 数组中是自定义属性名

3、props 中的值可以和 data 中的值一样使用

子组件给父页面(组件)传参:

1、在子组件中使用 this.$emit 调用在父组件(页面)中自定义的事件

2、自定义事件触发一个方法，方法的参数可以接受到值

3、this.$emit('事件名'，'传参')

41.\*\*\* 路由懒加载

把不同路由对应的组件分割不同的代码块，然后当路由被访问时才加载对应的组件就是路由的懒加载，可以加快项目的加载速度，提高效率。通过这种格式来导入组件 const foo = () => import('./foo.vue')

42. computed 和 watcher 的区别? wtach实现原理？watch有几种写法？

计算属性 computed:

1.支持缓存，只有当依赖数据发生改变，才会重新进行计算

2.不支持异步，当 computed 内有异步操作时无效，无法监听数据的变化

3.computed 属性值会默认走缓存，计算属性是基于它们的响应式依赖进行缓存的，也就是基于 data 中声明过或者父组件传递的 props 中的数据通过计算得到的值

4.如果一个属性是由其他属性计算而来的，这个属性依赖其他属性，是一个多对一或者一对一，一般用 computed

5.如果 computed 属性属性值是函数,那么默认会走 get 方法；函数的返回值就是属性的属性值；在 computed 中的属性都有一个 get 和一个 set 方法，当数据变化时，调用 set 方法

侦听属性watch:

1.不支持缓存，数据变化直接会触发相应的操作

2.watch支持异步

3.监听的函数接受两个参数，第一个参数是最新的值；第二个参数是输入之前的值

4.当一个属性发生变化时，需要执行对应的操作；一对多

5.监听数据必须是 data 中声明过或者父组件传递过来的 props 中的数据，当数据变化时，触发其他操作，函数有两个参数:

immediate:组件加载立即触发回调函数执行

deep:深度监听，为了发现对象内部值的变化，复杂类型的数据时使用，例如数组中的对象内容的改变，注意监听数组的变动不需要这么做。

注意:deep无法监听到数组的变动和对象的新增，参考vue数组变异，只有响应式的方式触发才会被监听到。

watch工作原理:

watch在一开始初始化的时候，会读取一遍监听的数据的值，此时那个数据就收集到 watch 的 watcher 了然后你给 watch 设置的 handler，watch 会放入 watcher 的更新函数中，当数据改变时，通知 watch 的 watcher 进行更新，于是你设置的 handler 就被调用了

44. 如果你是leader，做管理系统项目 Vue 和 React 怎么选择？

评估项目成员的水平，如果成员js基础较好，编码能力较强则选择 React，否则Vue

评估系统的大小，如果想构建生态系统，则选择React，如果要求快，简单和“能用就行”，则选择Vue

评估系统运行环境，如果你想要一个同时适用于web端和原生APP的框架，请选择React

45.\*\*\* Vuex的优缺点

优点: (1) 解决了非父子组件的消息传递(将数据存放在state中)

(2)减少了 AJAX 请求次数

缺点:如果您不打算开发大型单页应用，使用 vuex 可能是繁琐冗余的，并且 STATE 中的值会伴随着浏览器的刷新而初始化，无缓存,没有持久化存储的手段

46. Vue 和 React 的区别

1.Vue 使用的是 web 开发者更熟悉的模块与特性，Vue的API跟传统web开发者熟悉的模块契合度更高。React 的特色在于函数式编程的理念和丰富的技术选型，Vue更加注重web开发者的习惯。

2.Vue跟React的最大区别在于数据的 reactivity,就是数据的反应系统上。Vue提供反应式的数据，当数据改动时，界面就会自动更新，而React里面需要调用方法 SetState。

47.\*\*\* Vue路由传参，刷新后还有吗

通过 params 传参会出现参数丢失的情况，通过 query 的传参方式或者在路由匹配规则加入占位符即可以解决参数丢失的情况

48.\*\*\* Vue深层次嵌套传值方法

利用 $attrs 和 $listeners

49.\*\*\* Vue组件如何引入使用

1.定义组件并抛出

2.import引入，并在components里面定义

3.使用组件(注意首字母大写)

50. Vue路由实现的底层原理

51. 如何封装一个通用组件

通用组件的封装就是对可复用组件的解耦和样式复用，为了解耦一般数据都是通过父组件传递过来，在子组件中进行数据处理，对于一些较为复杂的数据可能还需要做数据验证，为了避免高耦合，逻辑最好放在父组件中，通过自定义事件将数据回传，子组件只是一个承载体，这样即降低耦合，又保证子组件中数据和逻辑不会混乱。

如果同一组件需要适应不同需求时，我们需要配合 slot 来使用，可以通过具名插槽灵活第解决了不同场景同一组件不同配置的问题。

52.\*\* Vue 生命周期通常使用那些

常用的生命周期有:

beforeCreate,created,beforeMount,mounted,beforeUpdate,updated,beforeDestory,destoryed

54.**\*** Vue 响应式原理

1、观察者 observer:首先通过观察者对 data 中的属性使用 object.defineproperty 劫持数据的 getter 和 setter，通知订阅者，触发他的 update 方法，对视图进行更新

2.Compile:用来解析模板指令，并替换模板数据，初始化视图，初始化相应的订阅器

3.订阅者Watcher:订阅者接到通知后，调用update方法更新对应的视图

4.订阅者Dep:订阅者可能有多个，因此需要订阅器Dep来专门接受这些订阅者，并统一管理，但在vue3中抛弃了object.defineproperty方法，

因此:

(1)Object.defineproperty 无法监测对象属性的添加和删除、数组索引和长度的变更，因此vue重写了数组的push/pop/shift/unshift/splice/sort/reverse方法

(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，这样很消耗性能

vue3 中实现数据双向绑定的原理是数据代理，使用 proxy 实现。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

55. Vue proxy的原理

主要通过 proxy 对对象进行绑定监听处理，通过 new map 对对象的属性操作进行处理，将要执行的函数匹配存到对应的 prop 上面，通过每次的访问触发 get 方法，通过修改触发 set 的方法，执行回调监听的函数，这样达到修改数据和视图的目标

56.\*\*\* Vue $forceUpdate的原理

1、作用:

迫使 vue 实例重新渲染。它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件

2、内部原理:

vue.prototype.$forceUpdate = function(){

const vm:Component = this

if(vm.\_watcher){

vm.\_watcher.updatw()

}

}

实例需要重新渲染是在依赖发生变化的时候会通知watcher，然后通知watcher来调用update方法

57.\*\*\* v-for key

\* key是为 Vue 中的 vnode 标记的唯一id，通过这个 key，我们的 diff 操作可以更准确、更快速

\* diff算法的过程中，先会进行新旧节点的首尾交叉对比，当无法匹配的时候会用新节点的key与旧节点进行对比，然后超出差异

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

diff过程可以概况为: oldCh 和 newCh 各有两个头尾的变量 Startldx 和Endldx,它们的2个变量相互比较，在比较的过程中，变量回往中间靠，一旦Startldx > Endldx 表明 oldCh 和 newCh 至少有一个已经遍历完了，就会结束比较，这四种比较方式就是首、尾、旧尾新头、旧头新尾

准确:如果不加 key ，那么vue会选择复用节点(Vue的就地更新策略)，导致之前的节点的状态被保留下来，会产生一系列的bug。快速:key的唯一性可以被Map数据结构充分利用，相比于遍历查找的时间复杂度O(n),Map的时间复杂度仅仅为O(1)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

建议使用id，不建议使用索引

62. axios 谁封装的，怎么封装的

*// 使用axios用于对数据的请求*

import axios from 'axios'

*// 创建axios实例*

const instance = axios.create({

baseURL:baseURL + version,

timeout:5000

})

*// 创建请求的拦截器*

instance.interceptors.request.use(confing => {

config.headers['Authorization'] = localStorage.

getItem('token')

return config

},error => {

return Promise.reject(error)

})

*// 创建响应的拦截器*

instance.interceptors.response.use(response => {

let res = null

*// 对相应的数据进行过滤*

if(response.status === 200){

if(response.data && response.data.err === 0){

res = response.data.data

}else if(response.data.err === -1){

return alert('token无效')

}

}else{

return alert('请求失败')

}

return res

},error =>{

return Promise.reject(error)

})

export default instance

66.\*\*\* 怎么修改 Vuex 中的状态？Vuex中有哪些方法

\* 通过 this.$store.state 属性的方法来访问状态

\* 通过 this.$store.commit 来修改状态

67.\*\*\* vue-router 路由传参的方式

1.query

*// 方法一*

<template>

<router-link :to="{

path:'blogDetail',

query:{id:item.id,views:item.views}

}">

</router-link>

</template>

*// 方法二*

this.$router.push({

path:'blogDetail',

query:{

id:item.id,

views:item.views

}

})

2.params

<template>

<router-link :to="{

name:'blogDetail',

params:{id:item.id,views:item.views}

}" tag="h2">

</template>

this.$router.push({

name:'blogDetail',

params:{

id:item.id,

views:item.views

}

})

68.\*\*\* hash和history 区别,怎么去解决history的回退问题

hash history

url显示 有#，很low 无#，好看

回车刷新 可以加载到hash值对应的页面 一般就是404掉了

支持版本 支持低版本浏览器和IE浏览器 HTML5新推出的API

69.\*\*\* 用过 beforeEach 吗？

每次通过 vue-router 进行页面跳转，都会触发 beforeEach 这个钩子函数,这个回调函数共有三个参数，to，from，next这三个参数，to表示我要跳转的目标路由对应的参数，from表示来自那个路由，就是操作路由跳转之前的，即将离开的路由对应的参数，next是一个回调函数，一定要调用next方法来 resolve 这个钩子函数

72.\*\*\* Vue中的单项数据流

单项数据流指只能从一个方向来修改状态

数据从父级组件传递给子组件，只能单向绑定

子组件内部不能直接修改从父级传递过来的数据

73.\*\*\* Vue组件中的Data为什么是函数，根组件却是对象呢？

如果 data 是一个函数的话，这样每复用一次组件，就会返回一份新的 data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份 data，就会造成了一个变了全都会变得结果。

所以说 vue 组件的 data 必须是函数。这都是因为js的特性带来的，跟 vue 本身设计无关。

74.\*\*\* 你做过那些Vue的性能优化？

1、首屏加载优化

2、路由懒加载

{

path:'/',

name:'home',

component:()=>import('@/views/home/index.vue'),

meta:{isShowHead:true}

}

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再解析。首先安装 webpack 提供的 compression-webpack-plugin 进行压缩，然后在 vue.config.js:

4.启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理将我们的压力分给其他服务器点。

5.代码层面优化

\* computed 和 watch 区分使用场景

computed:是计算属性，依赖其他属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其他数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算。

watch:类似于某些数据的监听回调，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作(访问一个API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

\* v-if 和 v-show 区分使用场景:v-if适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show则适用于需要非常频繁切换条件的场景。这里要说的优点在于减少页面中dom总数，我比较倾向于使用v-if，因为减少了dom数量。

\* v-for 遍历必须为item添加key，且避免同时使用v-if v-for遍历必须为item添加 key，循环调用子组件时添加key，key可以唯一标识一个循环个体，可以使用例如 item.id作为key避免同时使用v-if，v-for比v-if优先级高，如果每一次都需要遍历整个数组，将会影响速度。

\* Webpack对图片进行压缩

\* 避免内存泄漏

\* 减少 ES6 转为 ES5 的冗余代码

75.**\*** next Tick知道吗、实现的原理是什么？设计宏任务还是微任务？

微任务

原理:

nextTick 方法主要是使用了宏任务和微任务，定义了一个异步方法，多次调用nextTick会将方法存入队列中，通过这个异步方法清空队列。

作用:

nextTick用于下次Dom更新循环结束之后执行延迟回调。

78. 虚拟 dom 为什么会提高性能?

真实 DOM 的操作，一般都会对某块元素的整体重新渲染，采用虚拟 DOM

的话，当数据变化的时候，只需要局部刷新变化的位置就好了

虚拟 DOM 相当于在 js 和真实 dom 中间加了一个缓存，利用虚拟 dom 的话，当数据变化的时候，只需要局部刷新变化的位置就好了

80.\*\*\* Vue中key的作用?不加会怎么样？

vue中列表循环需加:key="唯一标识"，唯一标识可以是 item 里面 id index等，因为vue组件高度复用增加key可以标识组件的唯一性，可以让diff算法操作更准确，更迅速，可以更高效的更新虚拟DOM。

81.\*\*\* Vue 的常用修饰符

一、v-model修饰符

1、lazy

输入框改变，这个数据就会改变，lazy这个修饰符会在光标离开 input 框才会更新数据；

2、trim

输入框过滤首尾的空格

3、number

先输入数字就会限制输入只能是数字，先字符串就相当于没有加 number，注意，不是输入框不能输入字符串，是这个数据是数字。

二、事件修饰符

4、.stop:

阻止事件冒泡，相当于调用了 event.stopPropagation( )方法

5、.prevent:

阻止默认事件行为，相当于调用了event.preventDefault( )方法，比如表单的提交、a标签的跳转就是默认事件

6、.self:

只有元素本身触发时才触发方法，就是只有点击元素本身才会触发。比如一个div里面有个按钮，div和按钮都有事件，我们点击按钮，div绑定的方法也会触发，如果div的 click 加上 self ，只有点击到 div 的时候才会触发，变相的算是阻止冒泡。

7、.once:

事件只能用一次，无论点击几次，执行一次之后都不会再执行

8、.capture:

事件的完整机制是捕获-目标-冒泡，事件触发是目标往外冒泡

9、.sync

对 prop 进行双向绑定

10、.keyCode:

监听按键的指令，具体可以查看 vue 的键码对应表

82.\*\*\* Vue 计算属性compouted和 watch 的区别.

methods(方法):

只要进行调用就会执行，不管依赖的值有没有改变。无缓存。

computed(计算属性):

监听其所有依赖的变化，如果有变化会执行，没有变化不执行。有缓存，不用每次重新算。不支持异步。

详解:在 vue 的模板内( {{ }} )是可以写一些简单的js表达式，很便利。但是如果在页面中使用大量或是复杂的表达式去处理数据，对页面的维护会有很大的影响。这个时候就需要用到 computed 计算属性来处理复杂的逻辑运算。

1.优点:

在数据未发生变化时，优先读取缓存。computed 计算属性只有在相关的数据发生变化时才会改变要计算的属性，当相关数据没有变化时，它会读取缓存。而不必想 methods 方法和 watch 方法是否每次都去执行函数。

2.setter 和 getter 方法:(注意在vue中书写时用set和get)

setter 方法在设置值时触发

getter 方法在获取值时触发

watch(帧听属性):

观察某一个变量，发生变化会执行。支持异步。Vue实例将会在实例化是调用$watch(),遍历watch对象的每一个属性

一个对象，键是需要观察的表达式，值是对应回调函数。值也可以是方法名，或者包含选项的对象。

小结:

1.主动调用的方法写在 methods 里，依据某些变量的更新进行某种操作用 computed 或者 watch。

2.computed 和 watch:如果要异步，只能用watch。如果是计算某个值推荐用 computed，比如购物车全选单选功能，购物车计算总价小计功能。

83.\*\*\* Vue 中 v-on 可以绑定多个方法吗?

<p v-on="{click:dbClick,mousemove:MouseClick}"></p>

84.\*\*\* Vue 中 template 的编译过程

87.\*\*\* Vue 中 router-link 和传统 a 链接的区别?

router-link组件支持用户在具有路由功能的应用中(点击)导航。通过 to 属性指定目标地址，默认渲染成带有正确链接的标签，可以通过配置 tag 属性生成别的标签。通过router-link进行跳转不会跳转到新的页面，也不会重新渲染，它会选择路由所指的组件进行渲染，避免了重复渲染的“无用功”

总结:router-link组件避免了不必要的重复渲染，它只更新变化的部分从而减少DOM性能的消耗

89.\*\*\* 说一下 Vue-router 守卫有哪些。如何实现路由懒加载？

\* router.beforeEach 全局前置守卫 进入路由之前

\* router.beforeResolve 全局解析守卫在 beforeRouteEnter 调用之后调用

\* router.afterEach 全局后置钩子 进入路由之后

实现路由懒加载:vue-router配置路由，使用webpack的require.ensure技术，也可以实现按需加载

90.\*\*\* Vue 异步加载技术

1:vue-router 配置路由,使用 vue 的异步组件技术，可以实现懒加载，此时一个组件会生成一个js文件。

2:component:resolve => require(['放入需要加载的路由地址'],resolve)

91.\*\*\* 说一下Vue中路由跳转和传值的方式

1、路由跳转

1.router-link组件 默认会被渲染成一个标签，进行跳转，在组件中可以通过绑定 to 属性来指定要跳转的链接；tag属性指本来的标签

<router-link :to="/foo" tag="h2">Foo</router-link>

2.$router.push()方法

2、路由传参

1.query

*// 方法一*

<template>

<router-link :to="{

path:'blogDetail',

query:{ id:item.id,views:item.views }

}"

tag="h2">

</router-link>

</template>

*// 方法二*

this.$router.push({

path:'blogDetail',

query:{

id:item.id,

views:item.views

}

})

2.params

<template>

<router-link :to="{

name:'blogDetail',

params:{id:item.id,views:item.views}

}"

tag="h2"

>

<router-link>

</template>

this.$router.push({

name:'blogDetail',

params:{

id:item.id,

views:item.views

}

})

93. Vuex 刷新页面数据会丢失吗？咋解决的？

1、问题描述:页面刷新的时候 vuex 里的数据会重新初始化，导致数据丢失。因为 vuex 里的数据是保存在运行内存中的，当页面刷新时，页面会重新加载 vue 实例，vuex 里面的数据就会被重新赋值。

2.解决思路:

方法一:将 vuex 中的数据直接保存到浏览器缓存中(sessionStorage,localStorage,cookie)

方法二:在页面刷新的时候再次请求远程数据，使之动态更新 vuex 数据

方法三:在父页面向后台请求远程数据，并且在页面刷新前将 vuex 的数据先保存至 sessionStorage (以防请求数据量过大页面加载时拿不到返回的数据)

94. 什么是 Vue.nextTick()?

1、$nextTick 是在下次 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 ，意思是等你 dom 加载完毕以后再去调用 nextTick() 里面的数据内容。

97.\*\*\* 什么是 v-model?它的原理是什么？

1、v-model 本质上是一个语法糖，可以看成是 value + input 方法的语法糖。可以通过 model 的 prop 属性和 event 事件来进行自定义

2.v-model 是 vue 的双向绑定的指令，能将页面上控件输入的值同步更新到相关绑定的 data 属性，也会在更新 data 绑定属性时候，更新页面上输入控件的值。

99.\*\*\* Vue 的组件通信

1、props 和 $emit

父组件向子组件传递数据是通过 props 传递的，子组件传递数据给父组件是通过 $emit 触发事件。

2、$attrs 和 $listeners

3、中央事件总线 bus

上面两种方式处理的都是父子组件之间的数据传递，而如果两个组件不是父子关系呢？这种情况下可以使用中央事件总线的方式。新建一个 Vue 事件 bus 对象，然后通过 bus.$emit 触发事件，bus.$on监听触发的事件。

4、provide 和 inject

父组件中通过 provide 来提供变量，然后在子组件中通过 inject 来注入变量。不论子组件有多深，只要调用了 inject 那么就可以注入 provide 中的数据。而不是局限于只能从当前父组件的 prop 属性来获取数据，只要在父组件的生命周期内，子组件都可以调用。

5、v-model

父组件通过 v-model 传递值给子组件时，会自动传递一个 value 的 prop 属性，在子组件中通过 this.$emit('input',val)自动修改 v-model 绑定的值

6、$parent 和 $children

7、boradcas 和 dispatch

8、vuex 处理组件之间的数据交互 如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候才有上面这一些方法可能不利于项目的维护，vuex的做法就是将这一些公共的数据抽离出来，然后其他组件就可以对这个公共数据进行读写操作，这样达到了解耦的目的。

102.**\*** Vue项目常见优化点

1、首屏加载优化

2、路由懒加载

{

path:'/',

name:'home',

component:() => import('./views/home/index.vue'),

meta:{ isShowHead:true }

}

3、开启服务器 Gzip

开启 Gzip 就是一种压缩技术，需要前端提供压缩包，然后在服务器开启压缩，文件在服务器压缩后传给浏览器，浏览器解压后进行再解析。

4、启动 CDN 加速

我们继续采用 cdn 的方式来引入一些第三方资源，就可以缓解我们服务器的压力，原理是将我们的压力分给其他服务器点。

5、代码层面优化

\* computed 和 watch 区分使用场景

computed : 是计算属性，依赖其他属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值。当我们需要进行数值计算，并且依赖于其他数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算。

watch:类似于某些数据的监听回调，每当监听的数据变化时都会执行回调进行后续操作；当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch 选项允许我们执行异步操作(访问一个API)，限制我们执行该操作的效率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

\* v-if 和 v-show 区分使用场景 v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。这里说的优点在于减少页面中 dom 总数，我比较倾向于使用 v-if，因为减少了 dom 数量。

\* v-for 遍历必须为 item 添加 key，且避免同时使用 v-if v-for ，遍历必须为 item 添加 key，循环调用子组件时添加key，key可以唯一标识一个循环个体，可以使用例如 item.id 作为 key 避免同时使用 v-if，v-for比v-if优先级高，如果每一次都需要遍历整个数组，将会影响速度。

\* Webpack对图片进行压缩

\* 避免内存泄漏

\* 减少 ES6 转为 ES5 的冗余代码

103.\*\*\* Vue 和 redux 有什么区别？他们的共同思想

Redux 和 Vuex 区别

\* Vuex改进了 Redux 中的 Action 和 Reducer 函数，以 mutations 变化函数取代 Reducer，无需 switch，只需在对应的 mutation 函数里改变 state 值就可以。

\* Vuex 由于 Vue 自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的 state 就可以

\* Vuex 数据流的顺序是:View 调用 store.commit 提交对应的请求到 Store 中对应的 mutation 函数--store改变(vue检测到数据变化自动渲染)

共同思想

\* 单一的数据源

\* 变化可以预测

\* 本质上:Redux 和 Vuex 都是对 MVVM 思想的服务，将数据从视图中抽离的一种方案

\* 形式上: Vuex 借鉴了 Redux，将 store 作为全局的数据中心，进行数据管理

# 面试汇总

## WebSocket

它的最大特点就是，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话，属于[服务器推送技术](https://en.wikipedia.org/wiki/Push_technology" \t "_blank)的一种。

其他特点包括：

（1）建立在 TCP 协议之上，服务器端的实现比较容易。

（2）与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

（3）数据格式比较轻量，性能开销小，通信高效。

（4）可以发送文本，也可以发送二进制数据。

（5）没有同源限制，客户端可以与任意服务器通信。

（6）协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

其他特点包括：

（1）建立在 TCP 协议之上，服务器端的实现比较容易。

（2）与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

（3）数据格式比较轻量，性能开销小，通信高效。

（4）可以发送文本，也可以发送二进制数据。

（5）没有同源限制，客户端可以与任意服务器通信。

（6）协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

delete和vue.delete

delete只是被删除的元素变成了 empty/undefined 其他的元素的键值还是不变。

Vue.delete 直接删除了数组 改变了数组的键值。

promise和ajax的区别

两个不同的东西啊，ajax用来获取后台的数据，promise是ES6引入的，用来充当异步操作与回调函数之间的中介。

Ajax

Ajax：即异步 JavaScript 和XML。Ajax是一种用于创建快速动态网页的技术。通过在后台与服务器进行少量数据交换，Ajax可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。而传统的网页(不使用 Ajax)如果需要更新内容，必需重载整个网页面。

客户端发送请求，请求交给xhr，xhr把请求提交给服务，服务器进行业务处理，服务器响应数据交给xhr对象，xhr对象接收数据，由javascript把数据写到页面上

要完整实现一个AJAX异步调用和局部刷新,通常需要以下几个步骤:

创建XMLHttpRequest对象,即创建一个异步调用对象.

创建一个新的HTTP请求,并指定该HTTP请求的方法、URL及验证信息.

设置响应HTTP请求状态变化的函数.

发送HTTP请求.

获取异步调用返回的数据.

使用JavaScript和DOM实现局部刷新.

[vue父子组件生命周期执行顺序](https://www.cnblogs.com/zmyxixihaha/p/10714217.html)

父beforeCreate->父created->父beforeMount->子beforeCreate->子created->子beforeMount->子mounted->父mounted

Ajax,jQuery ajax,axios和fetch介绍、区别以及优缺点

「Ajax」

全称 Asynchronous JavaScript and XML（异步的 JavaScript 和 XML）最早出现的发送后端请求技术，隶属于原始js中，核心使用 XMLHttpRequest对象，多个请求之间如果有先后关系的话，就会出现回调地狱。

「Jquery Ajax」

是 jQuery 底层 AJAX 实现。简单易用的高层实现见 $.get, $.post 等。 $.ajax() 返回其创建的 XMLHttpRequest 对象。大多数情况下你无需直接操作该函数，除非你需要操作不常用的选项，以获得更多的灵活性。jQuery ajax - ajax() 方法

「Axios」

axios不是原生JS的，需要进行安装，它不但可以在客户端使用，也可以在 nodejs端使用。 Axios也可以在请求和响应阶段进行拦截。同样也是基于 [Promise]对象的。 特性：从浏览器中创建 XMLHttpRequests、从 node.js 创建 http 请求、支持 Promise API、拦截请求和响应等。

「Fetch」

Fetch 提供了对 Request 和 Response （以及其他与网络请求有关的）对象的通用定义。使之今后可以被使用到更多地应用场景中：无论是 service workers、 Cache API、又或者是其他处理请求和响应的方式，甚至是任何一种需要你自己在程序中生成响应的方式。`Fetch`号称是AJAX的替代品，是在ES6出现的，使用了ES6中的[Promise]对象。Fetch是基于promise设计的。 Fetch的代码结构比起 ajax简单多了，参数有点像 jQuery ajax。 注意，一定记住fetch不是ajax的进一步封装，而是原生js。Fetch函数就是原生js。

[vue父子组件生命周期执行顺序](https://www.cnblogs.com/zmyxixihaha/p/10714217.html)

父子组件生命周期执行顺序：首先执行父组件的 beforeCreate created beforeMount 。在执行子组件的beforeCreate created beforeMount mounted

v-if和v-for的优先级

2.x 版本中在一个元素上同时使用 v-if 和 v-for 时，v-for 会优先作用。

3.x 版本中 v-if 总是优先于 v-for 生效

[为什么vue组件中的data不是一个对象而是一个函数](https://www.cnblogs.com/wuqilang/p/11245915.html)

如果 data 是一个函数的话，这样每复用一次组件，就会返回一份新的 data，类似于给每个组件实例创建一个私有的数据空间，让各个组件实例维护各自的数据。而单纯的写成对象形式，就使得所有组件实例共用了一份 data，就会造成了一个变了全都会变得结果。

所以说 vue 组件的 data 必须是函数。这都是因为js的特性带来的，跟 vue 本身设计无关。

外派贝壳找房(中科铭天)

1、js 的作用域和作用域链了解吗,说一下

作用域:

在 JavaScript 中，作用域分为 全局作用域 和 局部作用域

全局作用域:代码在程序的任何地方都能被访问

局部作用域:在固定的代码片段才能被访问

作用域有上下级关系，上下级关系的确定就看函数是在那一个作用域下创建的。作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突

作用域链:

一般情况下，变量取值到 创建 这个变量 的函数的作用域中去取值

但是如果在当前作用域中没有查到值，就会向上级作用域去查，直到查到全局作用域，这么一个查找过程形成的链条就叫做作用域链

2、 原型和原型链的理解

原型:

每个实例对象都有一个\_proto\_属性，当我们使用构造函数去创建实例时，实例的\_proto\_属性就会指向对应构造函数的原型对象

原型链:

当视图访问一个 JavaScript 实例的属性时，首先搜索这个实例本身，当没有发现实例对应的方法时，它转而去搜索实例的原型对象，如果原型对象中也搜索不到，它就去搜索原型对象的原型对象，这个搜索的轨迹叫做原型链

3、闭包的理解，用作用域链解释一下闭包

闭包:能够访问其他函数内部变量的函数

闭包之所以可以读取外部函数的内部变量，即使外部函数已经返回，是因为他把外部函数的活动对象加到了自己的作用域链中

4、react 事件处理中修改 this 指向的方法

(1) 通过 bind 来指明当前方法中的 this 指向当前组件

(2) 在构造函数 constructor 中改变 this 指向0

(3) 使用箭头函数改变 this 指向

5、说一下防抖是怎么做的

在事件被触发 n 秒后再执行回调，如果该事件在这 n 秒内被重复触发，则重新计时

思路:通过闭包维护一个变量，此变量代表是否已经开始计时，如果已经开始计时则清空之前的计时器，重新开始计时

6、说一下 ES6 的新特性

(1) 新增的变量声明:const 和 let

(2) 模板字符串

(3) 箭头函数

(4) 函数的参数默认值

(5) Spread / Rest 操作符

(6) 二级制和八进制字面量

(7) 对象和数组解构

(8) 对象超类

(9) for...of 和 for...in

(10) ES6中的类

7、说一下 promise 怎么用的

Promise 简单来理解就是一个容器，里面存放着某个未来才会结束的事件结果。Promise 是一个对象，从它可以获取异步操作的消息，Promise 提供了统一的API，各种异步操作都可以用同样的方法进行处理。

Promise 的两个特点:

1.对象的状态不受外界影响。有三种状态:padding(进行中)，fulfilled(成功)，rejected(失败)。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态

2.一旦状态改变，就不会再变，任何时候都可以得到这个结果。三个状态只有从padding到fulfilled或者从padding到rejected。状态只有从padding改变到fulfilled或者refected，两种改变。

Promise的基本用法:

(1)Promise的语法糖 then

(2)Promise的语法糖 catch

(3)Promise的 finally 方法

(4)Promise的 all 方法

(5)Promise的 race 方法

8、说一下 var 、let 、const 的区别

(1)是否存在变量声明提升

var 声明的变量存在变量提升，全局作用域下是 window 属性

let 和 const 声明的变量没有变量提升

(2)能否重复声明

var 能重复声明

let 和 const 不能重复声明

(3)变量值能否修改

var 和 let 声明的值可以修改

const 声明的值是常量，不能修改。如果声明的是引用类型，可以修改其属性

(4)是否必须设置初始值

var 和 let 声明时可以不用设置初始值

const 声明时必须设置初始值，不能使用 null 占位

(5)是否存在块级作用域

var 没有块级作用域，let 和 const 有块级作用域

(6)是否存在暂时性死区

var 不存在暂时性死区，let 和 const 存在暂时性死区

9、如果 const 定义的是对象可以重新赋值吗？

可以

因为对象是引用类型的，const 定义的对象 q 中保存的是指向对象 q 的指针，这里的“不变”指的是指向对象的指针不变，而修改对象中的属性并不会让指向对象的指针方式变化，所以用 const 定义对象，对象的属性是可以改变的

10、说一下 react 的生命周期

11、说一下 react 的合成事件

12、This.setstate 重复调用会执行几次

两次

13、说一下 Flex 布局

使用 弹性布局 可以有效的分配一个容器的空间

即使我们的容器元素尺寸改变

它的内部元素也可以调整它的尺寸来适应空间

14、定位的属性属性值说一下

static:静态定位

relative:相对定位

absolute:绝对定位

fixed:固定定位

sticky:粘性定位

15、测试环境是什么阶段进行的

项目测试阶段

bug修改阶段

16、手机端网站怎么优化

1、 SEO(搜索引擎优化)

2、 首屏加载优化

(1)对文件和图片进行压缩

(2)图片懒加载

(3)采取 SDN 方式，对页面进行缓存

万里目少儿

1、用过画布吗 css3 动画了解多少

2、说一下什么是块级元素和行内元素 有什么区别

行内元素:(1)与其他行内元素并排

(2)不能设置宽高，默认的宽度就是文字的宽度

块级元素:(1)独占一行，不能与其他任何元素并列

(2)能接受宽高，如果不设置宽度，那么宽度将默认变为父级的100%

3、Img 是行内元素吗

是行内元素

4、你用过那些图片格式 问谷歌那个格式

BMP、tif、gif、jpeg、png

webp

5、Css选择器

1、标签选择器 2、id选择器 3、类选择器 4、并集选择器 5、符合选择器

6、通配选择器 7、后代元素选择器 8、子类选择器 9、伪类选择器 10、属性选择器 11、兄弟元素选择器

6、说几个伪类选择器

(1)结构性伪类选择器

first-child / last-child / only-child / nth:child(n)

nth-last-child(n)

(2)目标伪类选择器

target

(3)语言伪类选择器

lang

(4)否定伪类选择器

not

(5)动态伪类选择器

link 超链接并未被访问过

visisted 超链接已被访问过

active 元素被激活

hover 鼠标停留在元素上

focus 元素获取焦点

7、CSS居中方式

水平居中:(1)内联元素水平居中:text-align:center 实现在块级元素内部的内联元素水平居中

(2)块级元素水平居中:margin-left 和 margin-right 设成auto

(3)display:flex 弹性布局。使用 justify-content 用于设置弹性盒子元素在主轴方向上的对齐方式

8、盒模型

W3C标准盒模型 和 IE怪异盒子模型(box-sizing:boder-box)

W3C标准盒模型:总宽度 = width（content的宽度）+ margin(左右) + padding(左右) + border(左右)

IE怪异盒子模型:总宽度 = width(content + border + padding) + margin(左右)

9、Flex布局

弹性布局，为盒装模型提供最大的灵活性

6个属性：flex-direction / flex-wrap / flex-flow / justify-content / align-items / align-cintent

10、HTML5新增的内容有哪些

新增语义化标签 、 新增表单类型 、 表单元素 、 表单属性 、 表单事件 、 多媒体标签

11、HTML5新增的语义化标签有哪些

语义化标签优点:

(1) 提升可访问性

(2) seo

(3) 结构清晰，利于维护

Header页面头部 、 main页面主要内容 、 footer页面底部 、 nav导航栏

aside侧边栏 、 article加载页面一块独立内容

12、清除浮动的方式有哪些？请说出各自的优点

高度塌陷:当所有的子元素浮动的时候，且父元素没有设置高度，这时候父元素就会产生高度塌陷

清除浮动方式 1 :给父元素单独定义高度

优点:快速简单，代码少 缺点:无法进行响应式布局

清除浮动方式 2 :父级定义 overflow:hidden;zoom:1

优点:简单快速，代码少 缺点:超出部分被隐藏，布局时要注意

清除浮动方式 3 :在浮动元素后面加一个空标签，clear:both;

height:0;overflow:hidden

优点:简单快速，代码少，兼容性较高

缺点:增加空标签，不利于页面优化

清除浮动方式 4 :万能清除法

.father:after{

content:"随便写";

clear:both;

display:block;

height:0;

overflow:hidden;

visibility:hidden;

}

优点:写法固定，兼容性高 缺点:代码多

13、定位的属性

position ：relative absolute fixed static sticky(css3新增定位属性)

Relative 相对定位 不脱离文档流 相对于自身定位

Absolute 绝对定位 脱离文档流 相对于父级定位

Fixed 固定定位 脱离文档流 相对于浏览器窗口定位

Static 默认值 元素出现正常的流中

14、如何让 chrome 浏览器显示小于 12px 的文字

本来添加谷歌私有属性 -webkit-text-size-adjust:none 现在-webkit-transform:scale()

15、网页中有大量图片加载很慢，你有什么办法进行优化？

(1) 图片懒加载。在图片未可视区域加一个滚动条事件，判断图片位置与浏览器顶端和页面的距离，如果前者小于后者，优先加载

(2) 使用图片预加载技术，将当前展示图片的前一张和后一张优先下载

(3) 使用 csssprite 或者 svgsprite

16、Margin 和 padding 在什么场合下使用

Margin 外边距 ：自身边框到另一个边框之间的距离

Padding 内边距 ：自身边框到自身内容之间的距离

当需要在 border 外侧添加空白是用 margin，当需要在 border 内侧添加空白是用 padding

# 兄弟组件之间的传值

**方法1：借助父组件，让两个兄弟组件可以联动，子组件A将值传递给父组件，父组件再将值传递给子组件B**

**第一步：**子组件A传值给父组件,**通过$emit将e.target.innerText传递给父组件**

**第二步：父组件接收子组件A的传值，并将值保存到自己的data中，并取名letter（letter：字母）**

第三步：父组件将letter传递给子组件B, 子组件B通过props接收该值

**方法2：bus总线传值**

第一步：在main.js中创建bus总线

第二步：在子组件A中，通过bus总线抛出信息和值。**this.bus就是Vue实例，$emit也是上面的方法**

第三步：在子组件B中，在created或mounted等生命周期函数上，监听那个事件和获取那个值。

注意：bus总线，是联动的，当A组件触动bus总线的时候，B组件中也会响应式的触发。

B组件中的bus，需要在函数中，比如：created函数、mounted函数、甚至是data函数。

总之，监听bus总线的事件，必须要函数中，但是该函数执行的时候，跟bus总线无关。

路由组件中：两个组件不在一个页面，bus总线即使改变了值，当点击另一个路由路径的时候，data函数又将值初始化了，

所以说，bus总线作用的两个组件应该在同一页面中。当然，可以用一个仓库来存储这个改变的值，比如：vuex、webStorage、数据库等等。

# [js的简单数据类型和复杂数据类型](https://www.cnblogs.com/echol/p/12483766.html)

简单数据类型又被叫做基本数据类型或者值类型，存储变量时存储的是值本身，

string,number,boolean,undefined,null;

特殊情况：简单数据类型null的返回值为一个空的对象object;

复杂数据类型

通过new关键字创建的对象（系统对象，自定义对象），如Object,Array,Date;

**堆和栈**

栈（操作系统）：由操作系统自动分配释放存放函数的参数值，局部变量的值等，其操作方式类似于数据结构中的栈，**简单数据类型存放到栈里面·**栈里直接开辟一个空间用来存放存入的值

堆（操作系统）：存储复杂类型（对象），一般由程序员分派释放，若程序员不释放，由垃圾回收机构回收，**复杂数据类型存放到堆里面**，首先在栈里面存放地址，十六进制表示，然后这个地址指向堆里面的数据。

注意：JavaScript里面并没有堆栈的概念，通过堆栈，可以帮助更好的理解代码的执行方式

简单数据类型传参

函数的形参也可以看作是一个变量，当我们把一个值类型变量作为参数传递给函数的形参时，其实是把变量在栈空间的值复制了一份给形参，那么在方法内部对形参做修改，都不会影响外部变量

[复制代码](javascript:void(0);)

function fn(a){

a++;

console.log(a);

}

var x=10;

fn(x);

console.log(x);

//输出的值为11，10

[复制代码](javascript:void(0);)

复杂数据类型传参

函数的形参也可以看作是一个变量，当我们把一个引用类型变量传递给函数的形参时，其实是把变量在栈空间保留的堆地址复制给了形参，形参和实参其实保留的是同一个堆地址，所以操作的是同一个对象。

## 高阶组件

高阶组件是参数为组件，返回值为新组件的**函数**（将组件转换为另一个组件，纯函数，无副作用）

## hooks

Hooks 是React在16.8版本中出的一个新功能，本质是一种函数，可以实现组件逻辑复用，让我们在函数式组件中使用类组件中的状态、生命周期等功能，hooks的名字都是以use开头。

​

react：

  1. useState——创建状态

      接收一个参数作为初始值；返回一个数组，第一个值为状态，第二个值为改变状态的函数

​

  2、useEffect——副作用（数据获取、dom操作影响页面——在渲染结束之后执行

      (1)第一个参数为函数，第二个参数为依赖列表，只有依赖更新时才会执行函数；返回一个函数，当页面刷新的时候先执行返回函数再执行参数函数

      (2)如果不接受第二个参数，那么在第一次渲染完成之后和每次更新渲染页面的时候，都会调用useEffect的回调函数

  3、useRef

      返回一个可变的ref对象，此索引在整个生命周期中保持不变。可以用来获取元素或组件的实例，用来做输入框的聚焦或者动画的触发。

  4、useMemo——优化函数组件中的功能函数——在渲染期间执行

      （1）接收一个函数作为参数，同样接收第二个参数作为依赖列表，返回值可以是任何，函数、对象等都可以

      （2）这种优化有助于避免在每次渲染时都进行高开销的计算，仅会在某个依赖项改变时才重新计算

  5、useContext——获取上下文注入的值

      (1)接受一个context 对象，并返回该对象<MyContext.Provider> 元素的 value值；

              const value = useContext(MyContext)；

  6、useLayoutEffect——有DOM操作的副作用——在DOM更新之后执行

      和useEffet类似，但是执行时机不同，useLayoutEffect在DOM更新之后执行，useEffect在render渲染结束后执行，也就是说useLayoutEffect比useEffect先执行，这是因为DOM更新之后，渲染才结束或者渲染还会结束

  7、useCallback——与useMemo类似

      useMemo与useCallback相同，接收一个函数作为参数，也同样接收第二个参数作为依赖列表；

      useCallback是对传过来的回调函数优化，返回的是一个函数

      可以缓存函数，虽然useMemo也能实现，但是callback写法直观。缓存函数的时候，闭包变量也会被缓存，所以新手还是不推荐使用，容易出bug。每次render时定义新的函数就可以，成本很小。缓存函数的好处是，保证这个函数的引用不变。比如说有个组件A，是Pure Component，或者被React.memo包裹，而它需要通过props传递一个函数。<A func={func} />。如果每次render，没有用useCallback缓存函数，func都会是一个新的对象，那么Pure Component或者React.memo就没用了。这个时候useCallback作用就体现出来了，能在多次渲染期间保持引用不变（只要dependencies不变）。

​

​

react-router:

      被route包裹的组件，可以直接使用props进行路由相关操作，但是没有被route包裹的组件只能用withRouter高阶组件修饰或者使用hooks进行操作

      1、useHistory——跳转路由

      2、useLocation——得到url对象

      3、useParams——得到url上的参数

​

react-redux:

      1、useSelector——共享状态——从redux的store中提取数据

      2、useDispatch——共享状态——返回redux的store中对dispatch的引用

Hooks的优势

​

优势：解决 class 的 this 和 生命周期两大痛点

更好地拆分逻辑代码，每个 Hook 就做一件事

更好地复用代码，不会跟 HOC 一样出现嵌套地狱

区别：¬代码可读性更强，原本同一块功能的代码逻辑被拆分在了不同的生命周期函数中，容易使开发者不利于维护和迭代，通过 React Hooks 可以将功能代码聚合，方便阅读维护

¬组件树层级变浅，在原本的代码中，我们经常使用 HOC/render/Props 等方式来复用组件的状态，增强功能等，无疑增加了组件树层数及渲染，而在 React Hooks 中，这些功能都可以通过强大的自定义的 Hooks 来实现

常用hook函数：

useState：保存组件状态，功能类似于类组件中的this.state状态管理；

let [count, setCount] = useState(值/回调函数)

当值为回调函数时，只会在初始化执行，不会重复计算。

useEffect：模拟函数组件的生命周期；此函数的操作时异步的，useEffect(()=>{ },) 可以调用多次；参数一是一个回调函数，参数二可选，参数二如果为空数据 [ ]则模拟componentDidMount，如果没有写则模拟componentDidMount componentDidUpdate，如果写了值则表示此值进行componentDidMount componentDidUpdate，此函数也可以返回一个匿名函数，此匿名数据就是模拟componentWillUnMount

​

## 受控组件和非受控组件

受控组件

在表单元素中，state是唯一数据源，渲染表单的React组件控制着用户输入过程中表单发生的操作。被React以这种方式控制取值的表单输入元素叫做受控组件

​

非受控组件

表单数据由DOM节点来处理，而不是用state来管理数据，一般可以使用ref来从DOM节点中获取表单数据

​

区别

​

受控组件和非受控组件是表单中的组件，高阶组件相当于对某个组件注入一些属性方法

高阶组件是解决代码复用性问题产生的技术

受控组件必须要有一个value，结合onChange来控制这个value，取值为event.target.value/event.target.checked

非受控组件相当于操作DOM，一般有个defaultValue，通过onBlur触发响应方法

## hash和history

1、hash模式

这里的hash就是指url尾巴后的#号以及后面的字符。这里的#和css里的#是一个意思。hash也称作锚点，本身是用来做页面定位的，他可以使对应的id元素显示在可视区域内。

由于hash值变化不会导致浏览器向服务器发出请求，而且hash改变会触发hashchange事件，浏览器的进后退也能对其进行控制，所以人们在html5的history出现前，基本都是使用hash来实现前端路由的。他的特点在于：hash虽然出现url中，但不会被包含在HTTP请求中，对后端完全没有影响，因此改变hash不会重新加载页面。hash 本来是拿来做页面定位的，如果拿来做路由的话，原来的锚点功能就不能用了。其次，hash的而传参是基于url的，如果要传递复杂的数据，会有体积的限制

​

使用到api：

​

 window.location.hash='qq'//设置 url 的 hash，会在当前url后加上'#qq'

​

var hash = window.location.hash //'#qq'

​

window.addEventListener('hashchange',function(){

 //监听hash变化，点击浏览器的前进后退会触发

})

2、history模式

history模式不仅可以在url里放参数，还可以将数据存放在一个特定的对象中。

history———利用了HTML5 History Interface 中新增的pushState（）和replaceState（）方法。（需要特定浏览器的支持）history不能运用与IE8一下

相关API：

​

window.history.pushState(state,title,url)

//state：需要保存的数据，这个数据在触发popstate事件时，可以在event.state里获取

//title：标题，基本没用，一般传null

//url：设定新的历史纪录的url。新的url与当前url的origin必须是一样的，否则会抛出错误。url可以时绝对路径，也可以是相对路径。

//如 当前url是 https://www.baidu.com/a/,执行history.pushState(null, null, './qq/')，则变成 https://www.baidu.com/a/qq/，

//执行history.pushState(null, null, '/qq/')，则变成 https://www.baidu.com/qq/

​

window.history.replaceState(state,title,url)

//与pushState 基本相同，但她是修改当前历史纪录，而 pushState 是创建新的历史纪录

​

window.addEventListener("pospstate",function(){

//监听浏览器前进后退事件，pushState与replaceState方法不会触发

})

window.history.back()//后退

window.history.forward()//前进

window.history.go(1)//前进一部，-2回退两不，window.history.lengthk可以查看当前历史堆栈中页面的数量

​

这两个方法应用于浏览器的历史纪录站，在当前已有的back、forward、go 的基础之上，他们提供了对历史纪录进行修改的功能，只是当他们执行修改使，虽然改变了当前的url，但你的浏览器不会立即像后端发送请求。

## hook不能再在条件判断中

 1）只能在React函数式组件或自定义Hook中使用Hook。

 2）不要在循环，条件或嵌套函数中调用Hook，必须始终在React函数的顶层使用Hook。这是因为React需要利用调用顺序来正确更新相应的状态，以及调用相应的钩子函数。一旦在循环或条件分支语句中调用Hook，就容易导致调用顺序的不一致性，从而产生难以预料到的后果。

## redux流程

### 三大规则

单一数据源

​

整个应用的state被存储在一棵object tree中，并且整个 object tree 只存在于唯一一个 store 中

State是只读的

​

唯一改变state的方法就是触发 action，action是一个描述已发生事件的普通对象

这样确保视图和网络请求不能直接修改state

使用纯函数来执行修改

​

为了描述action如何改变state tree，你需要编写reducers

### redux原理

​

redux 是一个应用数据流框架，主要是解决了组件间状态共享的问题，原理是集中式管理，主要有三个核心方法，action，store，reducer。

工作流程是 view 调用 store 的 dispatch 接收 action 传入 store，reducer 进行 state 操作，view 通过 store 提供的 getState 获取最新的数据；

​

Redux是将整个应用状态存储到store,里面保存着一个状态树store tree,组件可以派发(dispatch)行为(action)给store,而不是直接通知其他组件，组件内部通过订阅store中的状态state来刷新自己的视图。

​

1）store就是保存数据的地方，Redux提供createStore这个函数，用来生成Store；

​

2）state就是store里面存储的数据，store里面可以拥有多个state，Redux规定一个state对应一个View,只要state相同，view就是一样的，反过来也是一样的，可以通过store.getState( )获取

​

3）state的改变会导致View的变化，但是在redux中不能直接操作state也就是说不能使用this.setState来操作，用户只能接触到View。在Redux中提供了一个对象来告诉Store需要改变state。Action是一个对象其中type属性是必须的，

​

4）store.dispatch( )是view发出Action的唯一办法，store.dispatch接收一个Action作为参数，将它发送给store通知store来改变state。

​

5）Store收到Action以后，必须给出一个新的state，这样view才会发生变化。这种state的计算过程就叫做Reducer。

Reducer是一个纯函数，他接收Action和当前state作为参数，返回一个新的state

## 性能优化

1、使用纯组件；

  2、使用 React.memo 进行组件记忆（React.memo 是一个高阶组件），对于相同的输入，不重复执行；

  3、如果是类组件，使用 shouldComponentUpdate（这是在重新渲染组件之前触发的其中一个生命周期事件）生命周期事件，可以利用此事件来决定何时需要重新渲染组件；

  4、路由懒加载；

  5、使用 React Fragments 避免额外标记；

  6、不要使用内联函数定义（如果我们使用内联函数，则每次调用“render”函数时都会创建一个新的函数实例）；

  7、避免在Willxxx系列的生命周期中进行异步请求，操作dom等；

  8、如果是类组件，事件函数在Constructor中绑定bind改变this指向；

  9、避免使用内联样式属性；

  10、优化 React 中的条件渲染；

  11、不要在 render 方法中导出数据；

  12、列表渲染的时候加key；

  13、在函数组件中使用useCallback和useMemo来进行组件优化，依赖没有变化的话，不重复执行；

  14、类组件中使用immutable对象；

## 虚拟DOM

虚拟dom是用js模拟一颗dom树,放在浏览器内存中，相当于在js和真实dom中加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。

  优点：

      （1）虚拟DOM具有批处理和高效的Diff算法,最终表现在DOM上的修改只是变更的部分，可以保证非常高效的渲染,优化性能；

      （2）虚拟DOM不会立马进行排版与重绘操作，对虚拟DOM进行频繁修改，最后一次性比较并修改真实DOM中需要改的部分；

      （3）虚拟 DOM 有效降低大面积真实 DOM 的重绘与排版，因为最终与真实 DOM 比较差异，可以只渲染局部；

      缺点：

      （1）首次渲染大量DOM时，由于多了一层虚拟DOM的计算，会比innerHTML插入慢；

​

  React组件的渲染过程：

      （1）使用JSX编写React组件后所有的JSX代码会通过Babel转化为 React.createElement执行;

      （2）createElement函数对 key和 ref等特殊的 props进行处理，并获取 defaultProps对默认 props进行赋值，并且对传入的子节点进行处理，最终构造成一个 ReactElement对象（所谓的虚拟 DOM）。

      （3）ReactDOM.render将生成好的虚拟 DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实 DOM。

​

  虚拟DOM的组成——ReactElementelement对象结构：

      （1）type：元素的类型，可以是原生html类型（字符串），或者自定义组件（函数或class）

      （2）key：组件的唯一标识，用于Diff算法，下面会详细介绍

      （3）ref：用于访问原生dom节点

      （4）props：传入组件的props，chidren是props中的一个属性，它存储了当前组件的孩子节点，可以是数组（多个孩子节点）或对象（只有一个孩子节点）

      （5）owner：当前正在构建的Component所属的Component

      （6）self：（非生产环境）指定当前位于哪个组件实例

      （7）\_source：（非生产环境）指定调试代码来自的文件(fileName)和代码行数(lineNumber)

## 监听props的变化

setState改变会触发的生命周期钩子

componentWillReceiveProps  父组件发生render的时候子组件就会调用

shouldComponentUpdate    判断是否需要重新渲染组件

componentWillUpdate       组件即将重新渲染

Render     组件渲染

componentDidUpdate       组件重新渲染完成

props改变会触发的生命周期

componentWillReveiceProps

shouldComponentUpdate

componentWillUpdate

render

componentDidUpdate

首次渲染会执行的生命周期：

componentWillMount（即将进入dom），Render，componentdidmount（进入dom），

1.组件第一次渲染时不会执行componentWillReceiveProps；

2.当props发生变化的时时候会执行componentWillReceiveProps；

​

# JavaScript

## for...in和for...of的区别

1.for...in 循环：只能获得对象的键名，不能获得键值

for...of 循环：允许遍历获得键值

2.for…in循环主要是为遍历对象而设计的，不适用于遍历数组。

3.对于普通的对象，for…in循环可以遍历键名，for…of循环会报错，必须部 署了 Iterator 接口后才能使用。

4.for…of循环可用于字符串、DOM NodeList 对象、arguments对象。

## Promise&async&await

/\*---------Promise概念：---------\*/

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大，简单地说，Promise好比容器，里面存放着一些未来才会执行完毕（异步）的事件的结果，而这些结果一旦生成是无法改变的

​

/\*---------async await概念：---------\*/

async await也是异步编程的一种解决方案，他遵循的是Generator 函数的语法糖，他拥有内置执行器，不需要额外的调用直接会自动执行并输出结果，它返回的是一个Promise对象。

​

两者的区别：

Promise的出现解决了传统callback函数导致的“地域回调”问题，但它的语法导致了它向纵向发展行成了一个回调链，遇到复杂的业务场景，这样的语法显然也是不美观的。而async await代码看起来会简洁些，使得异步代码看起来像同步代码，await的本质是可以提供等同于”同步效果“的等待异步返回能力的语法糖，只有这一句代码执行完，才会执行下一句。

​

async await与Promise一样，是非阻塞的。

​

async await是基于Promise实现的，可以说是改良版的Promise，它不能用于普通的回调函数

​

​

Promise补充

Promise是ES6新增的一个内置构造函数，在ES6之前只能通过回调函数的形式来实现在异步的中间或者末尾做一些事情的封装，只有封装异步代码的时候才会用到promise；promise是为了解决异步以及回调地狱{回调函数(回调函数就是一个被作为参数传递的函数)嵌套太多导致代码维护成本增加，代码可读性降低的情况}的问题，

Promise的三种状态：

等待中（pending）

完成了 （resolved）

拒绝了（rejected）

两种切换状态：pending到resolved、pending到rejected

Promise构造函数会接收两个函数resolved和rejected，resolved是一个函数，直接调用会把本次的状态改为成功，rejected也是一个函数直接调用会讲本次promise的状态改为失败，成功后执行.then里面的函数，失败后执行.catch里面的函数；每次return出来的都是一个新的promise。

优点：Promise 实现了链式调用，也就是说每次调用 then 之后返回的都是一个 Promise，并且是一个全新的 Promise，原因也是因为状态不可变。

缺点：无法取消 Promise，错误需要通过回调函数捕获。

在构造 Promise 的时候，构造函数内部的代码是立即执行的；Promise 实现了链式调用，也就是说每次调用 then 之后返回的都是一个 Promise，并且是一个全新的 Promise，原因也是因为状态不可变。

Promise.all用来执行多个 promise ，接受一个 promise 数组作为参数并返回一个新的 promise。结果数组中元素的顺序与其在源 promise 中的顺序相同。即使第一个 promise 花费了最长的时间才 resolve，但它仍是结果数组中的第一个。只有所有的promise都成功新的 promise 才会 resolve；如果其中一个 promise 被 reject，Promise.all 就会立即被 reject，完全忽略列表中其他的 promise。

length：打印出来为1

name：打印出来为Promise

all：将多个promise的实例包裹成一个实例，数组形势传入，遇到失败就直接调用失败，后续的操作不在关注。成功直接调用成功函数 ，返回一个数组。返回的数组和传入的promise的实例一样

race：将多个promise得实例包裹为一个实例，数组形势传入，返回实例中结果最快的哪一个 ， 不管失败还是成功的调用。all和race参考资料

resolve：和new Promise(resolve)生成一样的结果，都是生成一个实例对象。生成一个状态码和一个需要返回的数据

reject：和new Promise(resolve,reject)只调用reject生成一样的实例。同上

finally：不管结果如何都会执行的函数

promise过程：

（1）生成Promise实例

（2）执行一系列同步操作

（3）使用resolve函数将异步操作的结果传递出去, reject函数传递异步操作的错误

（4）用then方法分别指定Resolve状态和Reject状态的函数,then方法返回一个新的Promise实例,因此可以采用链式写法

​

## 事件循环

Event Loop(事件循环)中，每一次循环称为 tick, 每一次tick的任务如下：

* 执行栈选择最先进入队列的宏任务(通常是script整体代码)，如果有则执行
* 检查是否存在 Microtask，如果存在则不停的执行，直至清空 microtask 队列
* 更新render(每一次事件循环，浏览器都可能会去更新渲染)
* 重复以上步骤

## this

this 指向：

  1. 全局上下文（函数外）

      无论是否为严格模式，均指向全局对象。注意:严格模式下全局对象为undifined

  2. 函数上下文（函数内）

      默认的，指向函数的调用对象，且是最直接的调用对象：

  简单调用，指向全局对象注意:严格模式下全局对象为undifined，某些浏览器未实现此标准也可能会是window

​

改变this指向的方式：

  1. 第一种： new关键字改变this指向

​

  //构造函数版this

  function Fn(){

      this.user = "李某";

  }

  var a = new Fn();

  console.log(a.user); //李某

/\*----------------------------------------\*/

  2. 第二种： call()

  // 把b添加到第一个参数的环境中，简单来说，this就会指向那个对象

  var a = {

      user:"李某",

      fn:function(){

          console.log(this.user); //李某

      }

  }

  var b = a.fn;

  b.call(a); //若不用call，则b()执行后this指的是Window对象

/\*----------------------------------------\*/

  3. 第三种：apply()

  // apply方法和call方法有些相似，它也可以改变this的指向，也可以有多个参数，但是不同的是，第二个参数必须是一个数组

  var a = {

      user:"李某",

      fn:function(){

          console.log(this.user); //李某

      }

  }

  var b = a.fn;

  b.apply(a);

/\*----------------------------------------\*/

  4. 第四种：bind()

  // bind方法返回的是一个修改过后的函数,

  // bind也可以有多个参数，并且参数可以执行的时候再次添加，但是要注意的是，参数是按照形参的顺序进行的。

  var a = {

      user:"李某",

      fn:function(){

          console.log(this.user); //李某

      }

  }

  var b = a.fn;

  var c = b.bind(a);

  c();

## 闭包

闭包的作用：

  1. 在外部访问函数内部的变量

  2. 让函数内的局部变量可以一直保存下去

  3. 模块化私有属性和公共属性

​

闭包的原理：

  全局变量生存周期是永久，局部变量生存周期随着函数的调用介绍而销毁。

  闭包就是 在函数中定义且成为该函数内部返回的函数的自由变量 的变量，该变量不会随着外部函数调用结束而销毁。

  （注：不光是变量，函数内声明的函数也可以形成闭包）

  当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这时就产生了闭包。

​

​

简述闭包的问题以及优化

​

  闭包的缺点：占用内层空间 大量使用闭包会造成 栈溢出

  由于闭包会一直占用内存空间，直到页面销毁，我们可以主动将已使用的闭包销毁：

  将闭包函数赋值为null 可以销毁闭包

## 原型链

原型链:

  因为每个对象和原型都有原型，对象的原型指向原型对象，

  而父的原型又指向父的父，这种原型层层连接起来的就构成了原型链。

​

什么是原型：

  Javascript规定，每一个函数都有一个prototype对象属性，指向另一个对象（原型链上面的）。

  prototype(对象属性)的所有属性和方法，都会被构造函数的实例继承。这意味着，我们可以把那些不变(公用)的属性和方法，直接定义在prototype对象属性上。

​

prototype就是调用构造函数所创建的那个实例对象的原型（proto）。

​

prototype可以让所有对象实例共享它所包含的属性和方法。也就是说，不必在构造函数中定义对象信息，而是可以直接将这些信息添加到原型中。

​

为什么要用原型：使用原型对象解决浪费内存

## 作用域链

在 JavaScript 中有两种作用域类型：

​

  1. 局部作用域:只能在函数内部访问它们

  2. 全局作用域:网页的所有脚本和函数都能够访问它

  JavaScript 拥有函数作用域：每个函数创建一个新的作用域。

​

作用域决定了这些变量的可访问性（可见性）。

​

函数内部定义的变量从函数外部是不可访问的（不可见的）。

​

作用域链：

  当查找变量的时候，会先从当前上下文的变量对象中查找，

  如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。

这样由多个执行上下文的变量对象构成的链表就叫做作用域链

## ES6

1. 可以使用let和const定义一个变量，都是块级作用域

2. 模板字符串（倒引号）

3. 解构赋值

4. 对象简写法

5. for...of循环，但是他不能循环对象

6. 展开运算符（...）他可以展开数组和对象的多个元素

7. 剩余参数(可变参数)

8. ES6箭头函数

9. 参数默认值

10.类和继承

11.模块化规范

12.用class定义一个类

13.新增了this的方法call()，apply()

14.新增了数组的方法forEach()，map()，filter()，reduce()

## 宏任务和微任务

**1、\***\*微任务和宏任务\*\*

1. 宏任务：当前栈中执行的代码是宏任务
2. 微任务：当前宏任务执行完，在下一个宏任务开始之前需要执行的任务，就是微任务，又被称为回调事件。
3. 运行：从宏任务开始，先执行一次宏任务，再询问一次微任务，微任务队列有任务就清空微任务队列，循环往复，直到无宏任务和微任务时事件轮询结束。轮流询问宏任务队列和微任务队列。
4. **微任务**：promise nextTick
5. **宏任务：原生的js就是宏任务，如**setTimeout setInterval
6. JS执行顺序：先执行宏任务列，微任务队列

## 防抖节流应用场景

（都是为了降低频率）

可以将一些事件降低触发频率。比如懒加载时要监听计算滚动条的位置，但不必每次滑动都触发，可以降低计算的频率，而不必去浪费资源；另外还有做商品预览图的放大镜效果时，不必每次鼠标移动都计算位置。

### 防抖

防抖，顾名思义，防止抖动，以免把一次事件误认为多次，敲键盘就是一个每天都会接触到的防抖操作。

想要了解一个概念，必先了解概念所应用的场景。在 JS 这个世界中，有哪些防抖的场景呢

1. 登录、发短信等按钮避免用户点击太快，以致于发送了多次请求，需要防抖
2. 调整浏览器窗口大小时，resize 次数过于频繁，造成计算过多，此时需要一次到位，就用到了防抖
3. 文本编辑器实时保存，当无任何更改操作一秒后进行保存

### 节流

节流，顾名思义，控制水的流量。控制事件发生的频率，如控制为1s发生一次，甚至1分钟发生一次。与服务端(server)及网关(gateway)控制的限流 (Rate Limit) 类似。

1. scroll 事件，每隔一秒计算一次位置信息等
2. 浏览器播放事件，每个一秒计算一次进度信息等
3. input 框实时搜索并发送请求展示下拉列表，没隔一秒发送一次请求 (也可做防抖)

### 函数防抖的应用场景

连续的事件，只需触发一次回调的场景有：

搜索框搜索输入。只需用户最后一次输入完，再发送请求 手机号、邮箱验证输入检测(change、input、blur、keyup等事件触发，每次键入都会触发) 窗口大小Resize。只需窗口调整完成后，计算窗口大小。防止重复渲染。 鼠标的mousemove、mouseover 导航条上，用户不停的在导航区域滑动相当于

### 函数节流的应用场景

间隔一段时间执行一次回调的场景有：

滚动加载，加载更多或滚到底部监听，window.onscroll和滑到底部自动加载更多 谷歌搜索框，搜索联想功能 高频点击提交，表单重复提交

* 防抖：防止抖动，单位时间内事件触发会被重置，避免事件被误伤触发多次。代码实现重在清零 clearTimeout
* 节流：控制流量，单位时间内事件只能触发一次，如果服务器端的限流即 Rate Limit。代码实现重在开锁关锁 timer=timeout; timer=null

## 垃圾回收机制和内存管理

垃圾回收方式

  ① 标记清除

      工作原理：是当变量进入环境时，将这个变量标记为“进入环境”。当变量离开环境时，则将其标记为“离开环境”。标记“离开环境”的就回收内存。

  ② 引用计数

      工作原理：跟踪记录每个值被引用的次数。一旦没有引用，内存就直接释放了。

​

内存管理

  什么时候触发垃圾回收？

      垃圾回收器周期性运行，如果分配的内存非常多，那么回收工作也会很艰巨，确定垃圾回收时间间隔就变成了一个值得思考的问题。

      1、合理的GC方案：(1)、遍历所有可访问的对象; (2)、回收已不可访问的对象。

      2、GC缺陷：       (1)、停止响应其他操作；

      3、GC优化策略： (1)、分代回收（Generation GC）;(2)、增量GC

什么是垃圾？通俗的讲，我们不会再使用到的东西就是垃圾

垃圾回收，顾名思义就是释放垃圾占用的空间，防止内存泄露。垃圾回收机制：一个变量不在被使用时，会被回收；两个互相引用的对象，若不在被第三方使用，

# HTML+CSS

## 回流和重绘

什么是回流

当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候，这时候是一定会发生回流的，因为要构建render tree。在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程成为重绘。

​

什么是重绘

当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为重绘。

​

区别：

他们的区别很大：

回流必将引起重绘，而重绘不一定会引起回流。比如：只有颜色改变的时候就只会发生重绘而不会引起回流

当页面布局和几何属性改变时就需要回流

比如：添加或者删除可见的DOM元素，元素位置改变，元素尺寸改变——边距、填充、边框、宽度和高度，内容改变

​

触发重绘

触发回流一定会触发重绘

​

可以把页面理解为一个黑板，黑板上有一朵画好的小花。现在我们要把这朵从左边移到了右边，那我们要先确定好右边的具体位置，画好形状（回流），再画上它原有的颜色（重绘）

​

除此之外还有一些其他引起重绘行为：

​

颜色的修改

​

文本方向的修改

​

阴影的修改

​

​

触发回流

回流触发时机

回流这一阶段主要是计算节点的位置和几何信息，那么当页面布局和几何信息发生变化的时候，就需要回流，如下面情况：

​

添加或删除可见的DOM元素

​

元素的位置发生变化

​

元素的尺寸发生变化（包括外边距、内边框、边框大小、高度和宽度等）

​

内容发生变化，比如文本变化或图片被另一个不同尺寸的图片所替代

​

页面一开始渲染的时候（这避免不了）

​

浏览器的窗口尺寸变化（因为回流是根据视口的大小来计算元素的位置和大小的）

## H5新特性

一、语义标签

​

  二、增强型表单

​

  三、视频和音频

​

  四、Canvas绘图

​

  五、SVG绘图

​

  六、地理定位

​

  七、拖放API

​

  八、WebWorker

​

  九、WebStorage

​

  十、WebSocket

## 判断数据类型

1. typeof
2. instanceof
3. 通过Object下的toString.call()方法来判断
4. Object.prototype.toString.call();
5. console.log(toString.call(123)); //[object Number]
6. console.log(toString.call('123')); //[object String]
7. console.log(toString.call(undefined)); //[object Undefined]
8. console.log(toString.call(true)); //[object Boolean]
9. console.log(toString.call({})); //[object Object]
10. console.log(toString.call([])); //[object Array]
11. console.log(toString.call(function(){})); //[object Function]
12. 根据对象的contructor判断
13. ​
14. console.log('数据类型判断' -  constructor);
15. console.log(arr.constructor === Array); //true
16. console.log(date.constructor === Date); //true
17. console.log(fn.constructor === Function); //true
18. ​
19. 作者：8d2855a6c5d0
20. 链接：https://www.jianshu.com/p/967d6db70437
21. 来源：简书
22. 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
23. jq中判断数据类型的方法
24. jQuery提供了一系列工具方法，用来判断数据类型，以弥补JavaScript原生的typeof运算符的不足。以下方法对参数进行判断，返回一个布尔值。
25. jQuery.isArray();是否为数组
26. jQuery.isEmptyObject();是否为空对象 （不含可枚举属性）。
27. jQuery.isFunction():是否为函数
28. jQuery.isNumberic():是否为数字
29. jQuery.isPlainObject():是否为使用“{}”或“new Object”生成对象，而不是浏览器原生提供的对象。
30. jQuery.isWindow(): 是否为window对象；
31. jQuery.isXMLDoc(): 判断一个DOM节点是否处于XML文档中。
32. ​
33. 作者：8d2855a6c5d0
34. 链接：https://www.jianshu.com/p/967d6db70437
35. 来源：简书
36. 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# BFC？

BFC（块级格式化上下文），一个创建了新的 BFC 的盒子是独立布局的，盒子内元素的布局不会影响盒子外面的元素。在同一个 BFC 中的两个相邻的盒子在垂直方向发生 margin 重叠的问题 ；

BFC 是指浏览器中创建了一个独立的渲染区域，该区域内所有元素的布局不会影响到区域外元素的布局，这个渲染区域只对块级元素起作用

BFC触发条件以及规则

触发条件：

1.float的值不为none

2.overflow的值不为visible

3.display的值为table-cell、tabble-caption和inline-block之一

4.position的值不为static或则releative中的任何一个

规则：

1.浮动的元素会被父级计算高度（父级触发了BFC）

2.非浮动元素不会覆盖浮动元素位置（非浮动元素触发了BFC）

3.margin不会传递给父级（父级触发了BFC），两个相邻元素上下margin会重叠（给其中一个元素增加一个父级，然后让他的父级触发BFC）

清除浮动；

非浮动元素盖住浮动元素，可依靠触发BFC来解决

margin合并情况，在其中一个元素父级触发BFC，将不会合并margin

Bfc的优点

清除元素之间的影响

清除内部浮动元素对父级元素的影响

创建自适应布局

​

# 网路请求

## axios

axios是通过Promise实现对ajax技术的一种封装，就像jquery对ajax的封装一样，

简单来说就是ajax技术实现了局部数据的刷新，axios实现了对ajax的封装，axios有的ajax都有，ajax有的axios不一定有，总结一句话就是axios是ajax，ajax不止axios

### 拦截器

....

## Ajax

### 流程

一，创建一个异步调用对象

二，创建一个新的http请求

三，设置响应http请求状态变化的函数

四，发送http请求

五，获取异步调用返回的数据

六，使用js和dom实现局部刷新

​

​

ajax请求:

Asynchronous Javascript And XML

异步的 JavaScript 和 XML 的数据交互

​

通过 html文件 使用 JavaScript 技术 实现 数据交互

​

天生 就是一个 异步请求

​

本质就是 通过 JavaScript技术 设定 http协议 参数

执行完成 http协议

​

ajax 有 固定的执行步骤 必须严格遵守

​

必须要通过服务器运行才能正常执行ajax请求

​

ajax的步骤过程

1   创建 ajax 对象

同时 ajax 对象 设定 属性属性值 完成 http协议配置设定

​

readyState: 0

ajax请求状态码 结果是 0 1 2 3 4 5个数值

分别表示 ajax 执行的5个状态

0 表示创建了 ajax 对象 但是 什么都没有设定呢

1 表示设定了open() 也就是 设定了 请求方式 和 url地址

2 表示执行了send() 也就是 发送了请求

3 表示 有 响应报文 响应结束

4 从 响应报文 中 剥离 响应体内容成功 获取到响应体内容

​

response: ""

responseText: ""

从 响应报文中 剥离的 响应体内容

也就是 后端程序 输出的内容

​

responseURL:

响应体来源地址路径

​

status: 0

http状态码

statusText: ""

http状态描述

​

onload:

请求结束执行的函数程序

请求成功/失败都会执行

open:

设定请求方式 和 请求路径

send:

发送请求

​

2,  设定请求方式和请求URL地址

ajax对象.open()

参数1:  请求方式 get / post

参数2:  请求url地址

如果是get方式如果需要携带参数 在 url地址后 拼接参数

url地址?键名=键值&键名=键值...

​

3,  判断 如果是 post 请求 要 设定 请求头 get方式不用设定

​

4,  发送请求

ajax对象.send()

​

5,  接收结果

ajax对象.onload = function(){执行程序}

ajax执行结束 就会触发的程序

ajax对象.readyState

ajax状态码为 4

​

并且ajax对象.status

http状态码 是 200 - 299

​

两个条件都满足 证明 请求结束并且请求成功

### 原理

Ajax的工作原理相当于在用户和服务器之间加了—个中间层(AJAX引擎),使用户操作与服务器响应异步化。并不是所有的用户请求都提交给服务器,像—些数据验证和数据处理等都交给 Ajax引擎自己来做, 只有确定需要从服务器读取新数据时再由Ajax引擎代为向服务器提交请求。

## get和post 的区别

GET

  => 语义偏向于获取

  => 携带参数的位置 在地址后面直接拼接

  => 相对不安全, 明文发送

  => 发送的数据量比较小, 一般在 2KB 以下

  => 发送数据必须是 ASCII 码, 叫做 url 编码格式

GET请求返回的内容可以被浏览器缓存起来

  + POST

  => 语义偏向于提交

  => 携带参数的位置是在请求体中

  => 相对安全, 暗文发送

  => 发送的数据量比较大, 理论上来说无限大, 但是服务器可以限制

浏览器不会缓存POST请求返回的内容

get一般用于信息获取对所发信息的数量也有限制

post一般用于修改服务器上的资源，对所发信息的数量没有限制

网站上的普通的a标签，默认的都是GET请求，请求的是一些静态文件，比如HTML，CSS，JS，图片之类的。就是，一般的静态文件都是GET。另外，自己发送请求的话，看看需不需要与数据库交互，就是发送的请求与数据库有没有关系，无关就用GET，有关的话，比如你发送的请求里面有用户信息，密码之类的，一定要用POST。因为，GET请求可以直接看到，它在链接里面，而POST安全一些

​

# 手写代码

### 数组去重

let arr1 = [5, 5, 9, 9, 6, 3, 5, 4, 8, 6, 4, 5];

function unique(arr) {

 return Array.from(new Set(arr));

}

### 数组扁平化

function myFlat(arr){

   let res = [];

   for(let i=0; i<arr.length; i++){

       if(arr[i] instanceof Array){

           res = res.concat(myFlat(arr[i]));

      }else {

           res.push(arr[i]);

      }

  }

   return res;

}

​

let arr = [1,[2,3,[4,5]]];

console.log(myFlat(arr))

* 使用Typescript写一段对props限制的函数组件(interface)

1、JS基本数据类型有哪些

字符串String 数值Number 布尔boolean null undefined 对象

数组

2、Ajax 如何使用

一个完整的 AJAX 请求包括五个步骤:

1.创建 XMLHTTPRequest 对象

2.使用 open 方法创建 http 请求，并设置请求地址xhr.open(get/post , url , async , true(异步) , false(同步)) 经常使用前三个参数

3.设置发送的数据，用 send 发送请求

4.注册事件(给 ajax 设置事件)

5.获取响应并更新页面

1、bfc 的理解

BFC:块级格式上下文，它是指一个独立的块级渲染区域，只有 Block-level BOX 参与，该区域拥有一套渲染规则来约束块级盒子的布局，且与区域外部无关

BFC的特性及应用:

1、同一个 BFC 下外边距会发生折叠，如果想要避免外边距的重叠，可以将其放在不同的 BFC 的容器中

2、BFC 可以包含浮动的元素(清除浮动)

3、BFC 可以阻止元素被浮动元素覆盖

触发 BFC :

1、根元素

2、float 属性不为 none

3、position为absolute或fixed

4、overflow不为visible

5、display为inline-block,table-cell,table-caption,

flex,inline-flex

2、同源与跨域

什么是同源策略？

限制从一个源加载的文档或脚本如何与来自另一个源的资源进行交互

一个源指的是主机名，协议和端口号的组合，必须相同

跨域通信的方式：JSONP Hash CORS

JSONP原理:

基本原理:利用 script 标签的异步加载特性实现

给服务端传一个回调函数，服务器返回一个传递过去的回调函数名称的JS代码

3、原型与闭包相关对象

原型就是一个普通的对象，原型能存储我们的方法，构造函数创建出来的实例对象能够引用原型中的方法

用 Object.getPrototypeOf()方法来获取对象的原型

闭包是什么？

专业说法:闭包就是能够访问其他函数内部的函数

闭包的特性:

1、函数嵌套函数

2、函数内部可以引用外部的参数和变量

3、参数和变量不会被垃圾回收机制回收

4、说一下原型和原型链

原型 ：每一个 javascript 对象在创建的时候就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型“继承”属性

原型链 ：当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型和原型，一直找到最顶层为止

5、ES6的新特性

1、新增 const 与 let 变量

使用 var 会导致执行 function 函数之前，所有变量都会被提升，提升到函数作用域顶部

let 与 const 声明的变量解决了这种问题，因为他们是块级作用域，在代码块中使用 let 或 const 声明变量，该变量会陷入暂时性死区直到该变量的声明被处理

6、说一下 var let const 的区别

1、是否存在变量提升

var 声明的变量存在变量提升，全局作用域下是 window 属性，let 和 const 声明的变量没有变量提升，在变量声明之前存在暂时性死区

2、能否重复声明

var 能重复声明 ，let 和 const 不能重复声明

3、变量值能否修改

var 和 let 声明的值可以修改。const 声明的值是常量，不能修改。如果申明的是引用类型，可以修改其属性

4、是否必须设置初始值

var 和 let 声明时可以不用设置初始值。const 声明时必须设置初始值，不能使用 null 占位

5、是否存在块级作用域

块级作用域:申明的变量只在该块级作用域有效

var 没有块级作用域，let 和 const 有块级作用域

6、是否存在暂时性死区

暂时性死区:只要块级作用域有 let 或者 const 命令，他们所申明的变量就绑定这个区域，不受外部影响

var 不存在暂时性死区，let 和 const 存在暂时性死区

7、如果 const 定义的是一对象可以重新赋值吗？

const 作为常量是不可以重新赋值的。

const 定义的对象属性可以改变

8、说一下深浅拷贝

浅拷贝:

浅拷贝即只复制对象的引用，所以副本最终也是指向父对象在堆内存中的对象，无论是副本还是父对象修改这个对象，副本或者父对象都会因此发生同样的改变

深拷贝:

深拷贝则是直接复制父对象在堆内存中的对象，最终在堆内存中生成一个独立的，与父对象无关的新对象。深拷贝的对象虽然与父对象无关，但是却与父对象一致。当深拷贝完成之后，如果对父对象进行了改变，不会影响到深拷贝的副本，反之亦然

9、assgin 那个是进行浅拷贝还是深拷贝

Object.assgin()拷贝的是属性值。假如源对象的属性值是一个对象的引用，那么它也只指向那个引用。也就是说，如果对象的属性值为简单类型，通过Object.assgin()得到的新对象为深拷贝；如果属性值为对象或其他引用类型，那对于这个对象而言其实是浅拷贝

10、说一下 promise

Promise是异步编程的一种解决方案，比传统的回调函数和事件更合理和强大。所谓Promise，简单来说就是一个容器，里面保存着某个未来才会结束的事情。从语法上说，Promise是一个对象，从他可以获取异步操作的消息

特点:(1)对象的状态不受外界的影响。Promise对象代表一个异步操作，有三种状态:pending(进行中)、fulfilled(已成功)和rejected(已失败)。只有异步操作的结果可以决定当前是哪一种状态，任何其他操作都无法改变这个状态

(2)一旦状态改变，就不会再变。Promise对象的状态改变只有两种可能:从 pending 变为 fulfilled 和从 pending 变为 rejected。

1、vue的底层原理

Vue 是采用数据劫持配合发布者-订阅者模式，通过 Object.define

Property 来劫持各个属性的 getter 和 setter。在数据发生变化的时候，发布消息给依赖收集器，去通知观察者，走出对应的回调函数去更新视图

2、vue3.0的新特性

1、代码结构更清晰

2、压缩包体积更小

3、Object.defineProperty 替换为es6的proxy。将原本对对象属性的操作变为整个对象的操作，可优化性增强

4、启用 TypeScript 语法，大大的简化了代码

3、vue 中 compute 和 watch 的区别

computed 是通过几个数据的变化，来影响一个数据，而 watch，则是一个数据的变化，去影响多个数据

4、vue 里 data 为什么是函数而不是对象

如果写成对象，当组件被复用时，那么多个组件实例就会共用同一个 data 对象，这样改变其中一个实例的 data，就会改变所有的

因此写成函数返回值的形式，使得每个组件实例都会拥有自己的私有数据空间，不会造成混乱

5、vue 里数组改变为什么有的会渲染，有的不会渲染

对用数组的变异方法进行数组操作时可以达到视图的重新渲染，因为 vue 对这些方法进行了包裹

vue 不能检测一下数组的变动:

(1) 利用索引直接设置一个数组项时

(2) 修改数组的长度时

因为这样的操作 vue 根本没法做一个 Object.defineProperty 处理

6、update 里去改变 data 值会有问题吗？

由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用 update 钩子。应避免在此期间更改状态，否则可能导致死循环

7、加载数据在 created 好还是 mounted 好，两者区别？

在 created 比较好，如果在 mounted 钩子函数中请求数据可能导致页面闪烁问题

两者区别:

created:在模块渲染成html前调用，即通常初始化某些属性值，然后在渲染成视图

mounted:在模块渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作

8、vue 通信的几种方式

(1) props 和 $emit

(2) $parent 和 $children

(3) provide/inject

(4) vuex(状态管理器)

(5) localstorage 和 sessionStorage

9、vue 单向数据流

父级 prop 的更新会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解

10、vue 虚拟 dom 的作用

减少了 dom 操作，减小了页面渲染过程的次数，优化了加载速度

1、vue 的双向绑定?怎么实现的？

vue 数据双向绑定是通过数据劫持结合发布者-订阅者模式的方式来实现的。

实现数据的双向绑定:

1、实现一个监听器 Observer，用来劫持并监听所有属性，如果有变动的，就通知订阅者

2、实现一个订阅者 Watcher，可以收到属性的变化通知并执行相应的函数，从而更新视图

3、实现一个解析器 Compile，可以扫描和解析每个节点的相应指令，并根据初始化模块数据以及初始化响应的订阅器

v-bind input.value 事件 v-on input 事件

2、vue 的 vue-router 怎么实现的？

vue-router 就是作为'窗口'的存在，来渲染需要展示的组件

vue-router 是通过 vue.use 的方法被注入进 vue 实例中，在使用的时候我们需要全局用到 vue-router 和 router-link 组件，以及 this.$router/$route 这样的实例对象

主要有 hash 模式 和 history 模式

3、Webpack 的核心原理

(1) 一切皆模块

js、css、image和html文件都可以视作模块。因此，你可以require。我们可以将事物分割成更小的易于管理的片段，从而达到重复利用的目的

(2) 按需加载

传统的模块打包工具(module bundlers)最终将所有的模块编译生成一个庞大的bundle.js文件。但是在真实的app里边。“bundle.js”文件可能有10M到15M之大可能会导致应用一直处于加载中状态。因此 Webpack 使用许多特性来分割代码然后生成多个“bundle”文件，而且异步加载部分代码以实现按需加载

4、vue.use

在用 Vue 使用别人的组件时，会用到 Vue.use()。例如:Vue.use

(VueRouter)、Vue.use(MintUI)。但是用 axios 时，就不需要用到 Vue.use(axios)，就能直接使用。因为 axios 没有 install

安装 Vue.js 插件。如果插件是一个对象，必须提供 install 方法。如果插件是一个函数，它会被作为 install 方法。install 方法调用时，会将 Vue 作为参数传入

5、bind call apply的区别

相同之处：

改变函数体内 this 的指向。

不同之处：

call、apply的区别：接受参数的方式不一样。

bind：不立即执行。而apply、call 立即执行。

区别：1.apply和call会使当前函数立即执行，bind会返回一个函数，后续需要时再调用

2、call是apply的语法糖，只有传的参数不同，call中要传参数列表，apply只可以直接数组

3、bind是为函数绑定一个this上下文

规则：fn.apply(上下文环境 ，执行所需数组)

#### call

* 功能: 调用函数,改变函数中的this指向
* 参数:
* 第一个参数 : 设置this的指向,如果指定了 null 或者 undefined 则内部 this 指向 window
* 其他参数 : 对应函数的参数
* call的返回值就是函数的返回值

#### apply

* 调用函数，改变函数中的this
* 第一个参数 : 设置函数内部this的指向

第二个参数 : 是数组

* call的返回值就是函数的返回值

bind

bind() 函数会创建一个新函数（称为绑定函数），新函数与被调函数（绑定函数的目标函数）具有相同的函数体（在 ECMAScript 5 规范中内置的call属性）。

当目标函数被调用时 this 值绑定到 bind() 的第一个参数，该参数不能被重写。绑定函数被调用时，bind() 也接受预设的参数提供给原函数。

一个绑定函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。提供的 this 值被忽略，同时调用时的参数被提供给模拟函数。

# export与export default的区别以及需要注意事项

1.export与export default均可用于导出常量、函数、文件、模块等

2.在一个文件或模块中，export、import可以有多个，export default仅有一个

3.通过export方式导出，在导入时要加{ }，export default则不需要

(1) 输出单个值，使用export default

(2) 输出多个值，使用export

(3) export default与普通的export不要同时使用

出现问题：当使用export default {a, b, c, d} 容易造成嵌套多层；

结果：{a: {a, b, c, d}, b:{a, b, c, d}, c:{a, b, c, d}, d:{a, b, c, d}} //error

# 1、vue和react的区别

**Vue 使用的是 web 开发者更熟悉的模板与特性**，Vue的API跟传统web开发者熟悉的模板契合度更高，比如Vue的单文件组件是以模板+JavaScript+CSS的组合模式呈现，它跟web现有的HTML、JavaScript、CSS能够更好地配合。**React 的特色在于函数式编程的理念和丰富的技术选型**。Vue 比起 React 更容易被前端工程师接受，这是一个直观的感受；React 则更容易吸引在 FP 上持续走下去的开发者。

从**使用习惯和思维模式**上考虑，对于一个没有任何Vue和React基础的web开发者来说， Vue会更友好，更符合他的思维模式。React对于拥有函数式编程背景的开发者以及一些并不是以web为主要开发平台的开发人员而言，React更容易接受。这并不意味着他们不能接受Vue，Vue和React之间的差异对他们来说就没有web开发者那么明显。可以说，**Vue更加注重web开发者的习惯**。

**实现上，Vue跟React的最大区别在于数据的reactivity，就是反应式系统上。**Vue提供反应式的数据，当数据改动时，界面就会自动更新，而React里面需要调用方法SetState。我把两者分别称为**Push-based**和**Pull-based**。所谓Push-based就是说，改动数据之后，数据本身会把这个改动推送出去，告知渲染系统自动进行渲染。在React里面，它是一个Pull的形式，用户要给系统一个明确的信号说明现在需要重新渲染了，这个系统才会重新渲染。两者并没有绝对的优劣之分，更多的也是思维模式和开发习惯的不同。

两者不是完全互斥的，比如说在React里面，你也可以用一些第三方的库像MobX实现Push-based的系统，同时你也可以在Vue2.0里面，通过一些手段，比如把数据freeze起来，让数据不再具有反应式特点，或者通过手动调用组件更新的方法来做一个pull-based系统。所以两者并没有一个绝对的界限，只是默认的倾向性不同而已。

# [前端性能优化方案](https://www.cnblogs.com/angle-xiu/p/11347024.html)

　　1.在js中尽量减少闭包的使用(闭包不会释放栈内存)

　　　　A:循环进行事件绑定时，尽可能使用自定义属性，而不用创建闭包来存储信息。

　　　　B:在最外层形成一个闭包，把一些后期需要的公共信息进行存储，而不是每一个方法都创建一个闭包（例如单例模式）。

　　　　C：尽可能手动释放掉不需要的内存。

　　　   。。。

　　2.进行js和css文件的合并，减少http请求的次数，尽可能将文件进行压缩，减少请求资源的大小。

　　　　A：webpack这种自动化构建工具，可以帮我们实现代码的合并和压缩（工程化开发）

　　　　B:在移动端开发过程中，如果代码不是很多，直接将css和js写html中。

　　3.尽量使用字体图标和SVG图标，来代替传统的PNG等格式的图片（字体图标等是矢量图）

　　4.减少对DOM的操作（主要是减少DOM的重绘和回流（重排））

　　　　A：关于重排的分离读写（浏览器会将连续dom操作一起缓存起来一起操作）

　　　　B:使用文档碎片或者字符串做数据绑定（DOM的动态创建）

　　5.js避免“嵌套循环”（会额外增加很多次循环次数）和“死循环”（浏览器会死机）

　　6.采用图片“懒加载”，加快第一次加载的速度，实际并没有减少请求次数

　　　　步骤：开始加载页面是，所有的真实图片都不去发送请求，而是给一张占位的背景图，当页面加载完后，并且图片出现在可是区域再去做图片加载。

　　7.利用浏览器和服务器端的缓存技术（304缓存），把一些不经常变更的资源进行缓存，例如js和css文件。目的就是减少请求大小。

　　8.尽可能使用事件委托来处理绑定的操作，减少DOM的频繁操作。

# 前端常用的缓存技术

CDN缓存，即内容分发网络。CDN是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率

**优点：**

1. 减少带宽的占用，减轻网络拥堵，提高访问速度
2. 减轻本地服务器负担

DNS（Domin Name System）是域名和IP相互映射的分布式数据库，可将域名解析成供计算机识别的IP地址

**DNS缓存机制：** 当首次访问一个域名，通过DNS解析获取其IP，并将域名及对应的IP缓存下来，当下次访问该域名时，则不再进行DNS查询，直接使用缓存下来的IP

**缓存时间：** 不同的浏览器DNS缓存时间也不同，IE默认的DNS缓存时间为30min，Firfox、Chrome默认的DNS缓存时间为1min

若域名对应的ip已更改，则需要清除DNS缓存（ipconfig/flushdns）

## HTTP缓存

当客户端第一次完成数据请求后，浏览器会缓存本次请求的数据，当下次执行相同请求，则直接在缓存数据库中返回；但HTTP缓存有多种规则，根据是否向服务器发送请求分为 **“强制缓存”** 和 **“对比缓存”** 两种

### 强制缓存

当客户端发送请求时，若浏览器中有该请求的缓存数据，则直接返回缓存数据；

若浏览器中未有本次请求的缓存数据或缓存数据过期，则本次请求会抵达服务器去请求数据；

### 对比缓存

浏览器完成一次完整请求后，服务器返回数据和缓存规则，客户端收到并缓存在缓存数据库中；

对比缓存顾名思义就是每次请求，客户端都会拿缓存标识去服务器判断该缓存是否可用，若可用，服务器返回304并通知客户端从缓存中读取，若不可用则服务器返回200和最新的数据和缓存规则

### 前端如何优化大量数据处理操作

从数据上处理：分页分表，比如前端可以把数据分页展示，后端也分段吐数据

从渲染上解决：

2.1 异步渲染，比如进入页面先不渲染，然后加载好页面再渲染。

2.2 局部渲染：只渲染目前可见区域的数据，再渲染次屏数据。

2.3 还有性能瓶颈，可以考虑web worker 做压缩和解码，也可以考虑离屏canvas做预渲染。

3. 减少网络耗时：压缩数据，gzip等

作者：counterxing

链接：https://www.zhihu.com/question/378973828/answer/1085538980

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

# 项目实战

1、项目中的挑战

1.（使用缓存处理）

在做Vue开发移动端APP时，有个页面比较常见，左边是对所有菜谱品类的展示，右边是对对应菜谱的展示，一开始在开发的时候没注意，就直接在mounted()前面使用async,里面使用await调了两边接口后，又通过watch监听了品类索引变化，调了一遍接口。后面发现每次切换品类时，页面都有一闪而过的感觉，发现每次都调了一遍接口，这对性能消耗挺大。所以就尝试的走了Vuex做缓存处理。当时解决这个缓存问题用了一些巧妙的方法，首先缓存的数据，采用的是对象形式，不是数组。这样写起来更快，因为用品类的下标直接做缓存数据的key值，非常好写。不过后面又有点坑，就是我监听的是引用数据类型，不管是vue还是react，引用数据类型发生变化时，页面可能不会更新。后面又在mutation中对vuex中的数据更新时，做了一下深复制，就做好了缓存处理。因为我是根据判断这个对象中有没有数据去调接口的，如果存在，就不调接口。现在做了缓存没错，那如果以后后台的数据发生了变化的话，那我这里也不调接口了，后台数据就没有在页面上实现更新，所以在跳出这个页面的时候还要清下缓存。清缓存要在生命周期结束的时候清，如果碰到了动态组件把APP页面下的 Tap 栏包住的时候， Keep-alive。就不能用 destroyed 生命周期清除，要用 deactivated 生命周期去清除。这样才有始有终，完成缓存处理。

2.（解决关键词高亮问题---->原理和字符串敏感词替换一样，但是用在关键词高亮上算是一个技术亮点吧）

用户配置一堆关键词，在页面上将这些关键词高亮，也许你会觉得这有什么难度？用正则匹配一下出来高亮不就行了吗？但是，一开始，用户的词不多，我确实使用的是遍历，时间复杂度为n2。后来用户会配置100w量级的词，使用遍历就会使页面卡死崩溃。解决的方法就是：优化性能，高亮分三步，生成字典树，遍历页面文字，取出文字进行匹配。使用字典树代替遍历，整个页面100w量级的词绘制可以实现在1秒以内。

3.（封装自定义组件）

在一个小程序项目需求中，要求页面头部tab栏切换的同时，对应tab栏品类的页面也要展示出来。你可能会觉得这个需求用个taroUI组件库中的tabs标签页组件不就可以完成吗？但是这个需求想满足用户的沉浸式体验，切换页面时需要有独特丝滑的专场特效。所以当时面临的问题就是，使用的UI库的组件默认样式生硬，满足不了需求。我当时是对小程序原生swiper组件进行了二次封装，主要实现了几点自定义需求：头部Tab栏品类样式使用flex动态布局，实现品类数量可变；使用 slot 插槽来动态渲染 Tab区块中的内容，配合原生swiper组件使用定义插槽；小程序原生组件<swiper>是有默认高度的，必须手动设置其高度，这里使用wx.getSystemInfo来动态获取屏幕尺寸。自己封装组件，踩了不少坑，但从中我学习到了：使用小程序的原生组件，并修改其默认的样式；学会使用 slot 插槽，实现组件内容的差异化；学会了使用小程序原生 api 获取手机信息，用 js 改变组件样式等等

4.（遇到了难用的轮子）

在写XXX小程序项目中需要实现音频播放的功能，官方已经推荐使用新的API了，然而这个小程序官方文档文档写得并不好，很多时候我们会遇到一些需求，或者改变而这些需求文档里又写得非常模糊，这就比较头疼了。小程序官方文档更新十分频繁，坑非常多。所以很多时候，我就要去不断地从文档的字里行间猜测，并结合源码一步一步地去跟踪，去尝试解决这个音频播放问题。但有时候确实超出了我的能力范围，那么我就会把我的问题提炼成一个小demo，到知乎、segmentfault思否去问，或者提问一些同样用这个轮子的作者，最终找到了新API的使用规范。

2、项目研发流程中作为前端开发一般扮演的啥角色？

前端开发一般扮演着一个“团队核心废物”的角色。为什么这么说，首先在项目的研发中，UI小姐姐总感觉前端开发的页面满足不了她们的设计理念，说你没品味；后端小哥哥觉得前端开发就像只会写写样式的js交互工程师，像极了一个破美工的；测试小哥哥拿着测试报告说：这锅谁他娘的来背一下。还记得当年那个请你根据手机壳的颜色，来实现APP启动的颜色的产品经理嘛？这些对前端开发的误会难道不能折射出前端是团队里最应该学会沟通的人嘛？

界面有问题需要和UI沟通,数据有问题需要和后台沟通,功能有问题需要和产品沟通,测试的时候给你提bug你还需要和测试沟通……毕竟前端是最接近用户的人,用户对一个网站,软件最直观的感受是反映到前端；交互体验更是前端项目的核心点。

和UI的沟通,在工作中我们不应该是被动的实现UI的设计,而是应该合理化的提出自己的想法,不然日后返工浪费的是双方的时间。比如通用组件的设计，每次页面的提示弹窗设计，再比如你需要做一个图表,用到了echarts,你完全可以让UI基于echarts去设计样式,而不是让她在那里自由发挥，因为你永远不知道设计师的脑子里装了多少创意,这样节省的是两个人的时间,不会出现他做好样式而你实现不了的尴尬。

和后端联调接口前，先要对业务需求了解透彻，需要哪些数据，有时候明明后台来处理某个事件很简单,后台非要你来做,这就需要我们对一个需求,一个任务的要有清晰认识了,如果对任务含糊不清,自己都没搞明白,你只能受后台摆布了.最后可能也会因为任务没有完成而备受责难了。有理有据,后台开发人员是不会说什么的,否则,后台会很不耐烦的,甚至骂你的可能都有,本身做后台比较难,尤其在查询数据,取数据,封装数据方面都比较难处理。

面对产品经理的需求，前端应该深刻理解需求，毕竟工作性质影响了一个人的思维逻辑，前端能站在一个产品经理的角度去思考每一个需求，便显得尤其重要。不放过每一个细节也很重要。产品经理在设计一个产品的时候，都是从大方向去想问题的，大方向没有错就行了，细节脱离不了大方向。这是他们想的。但是对于程序来说，却万万不能。因为一个细节的逻辑往往决定了整个大方向。

举个例子：有一个需求，用户的作品需要提交审核，经过审核才可以让所有人看到。当产品经理交这个需求给你的时候，你能察觉到什么问题了吗？这里面有几个细节：1.用户提交审核后，用户可以不可以再编辑作品；2.作品是否会多次审核；3.需不需要记录审核历史；4.用户作品是否需要有版本的控制，如要产生版本，版本又是如何产生的；5.审核通过后，用户可以不可以再修改作品，若不可以，那么是不是其他人就看不见用户作品等等。

3、现在有的项目中觉得哪些项目可以继续优化，为啥没有优化？

之前用vue做了一个动态官网项目，后期客户要求seo，百度上之前搜索不到官网地址，后来在项目的入口文件index.html页面加上了，固定的meta标签，加上name名为keywords、description的meta标签。以上做了个简单的seo优化，这个项目有几个官网，但是其中只有一个官网要求seo，也就是在百度能够搜索到，当时为了应急，就写死了，但是，其它的网站也就会受到干扰了，也就是对于一个项目对应几个官网，写死的meta标签做seo是不科学的。

如果这个项目要解决seo优化，可以用服务端渲染（ssr），如果项目刚开始就考虑到seo，采用服务端渲染，那么就用服务端渲染就得了。但是一般来讲，项目做到后期才会考虑到seo的问题，这时再去搞服务端渲染，相当于重头写项目，非常耗费人力物力。

所以先只考虑在首页加入 meta 标签提供一些元数据，使用简单、具有表意性的 title 以及使用 h5 提供的具有语义化的标签（不要一堆 div），生成对 search engine 友好的 sitemap，使用合理的 html 结构（比如按标题、内容、页脚这样的顺序、或者将重要的内容放在 html 前，其他放在后）

4、讲一下最近的这个项目中都负责什么

1. 这个项目的列表展示与详情页是我负责的。我对于后端传递的数据进行了怎样的处理，在异步请求中选择了一定的异步分割处理数据，拆分一次性阻塞主线程的时间，可以减少用户的等待，页面滚动时选择节流，减少无效的axios请求等等，对自己模块所负责的内容，进行梳理。

2. 这个项目的登录，注册，模块是我负责的。我对不同角色的鉴权是怎么实现的。。。

3. 这个项目的表单提交页是我负责的。。。。用了什么UI组件。。。

5、 谈谈对前后端分离的理解。

在我看来，前后端分离给开发工作带来了很多好处，在以前，没有这个概念的时候，一种经典的设计模式MVC模式。前端开发很不盛行，几乎很多后端程序员就兼顾了前端开发工作，前后端耦合性极强，在jsp时代，前端写好的页面最后要和后端实现交互，需要程序员手动的更改代码，这是很大的一个工作量，这种事情交给前端还是后端？谁都不愿意做吧！

前后端分离过后，通过预先定义好接口规范，前端后端独立部署独立开发，后端只需要提供接口供调用即可，解耦效果是真的强，这对工作效率的提升是巨大的，对于后期的维护只需要前后端单独完成即可！

**6、eventloop**

JavaScript从诞生就是单线程。但是单线程就导致有很多任务需要排队，只有一个任务执行完才能执行后一个任务。如果某个执行时间太长，就容易造成阻塞；为了解决这一问题，JavaScript引入了事件循环机制

# Javascript单线程任务被分为同步任务和异步任务。

同步任务：立即执行的任务，在主线程上排队执行，前一个任务执行完毕，才能执行后一个任务；

异步任务：异步执行的任务，不进入主线程， 而是在异步任务有了结果后，将注册的回调函数放入任务队列中等待主线程空闲的时候读取执行。

每次单个宏任务执行完毕后， 检查微任务队列是否为空， 如果不为空，会按照先入先出的规则全部执行完微任务后， 清空微任务队列， 然后再执行下一个宏任务，如此循环

如何区分宏任务与微任务呢？

宏任务：macrotask，又称为task,  可以理解为每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）。

微任务：microtask, 又称为job, 可以理解是在当前 task 执行结束后立即执行的任务。包括：Promise.then/cath /finally回调（平时常见的）、 MutationObserver回调（html5新特性）

## 自己实现promise的大体思路

1. 我们要明确我们需要一个异步的操作方法,满足异步回调。所以选择加入setTimeout 作为实现的基础， 让函数实现延迟触发。
2. 保持一个原则，控制 promise 改变状态的只有 promise 构造函数里的 reslove 、 reject 函数。
3. 链式调用的原理， 类似jQuery，它会在调用方法后， return this. 从而形成链式调用。所以我们采用在调用then(fn)、 catch(fn) 后 会返回一个新的 promise 对象， 然而 这个 promise 对象 受到 它的上级promise 对象的状态结果 和 fn 运行结果的控制。

Promise的实现

* 第一步，首先考虑Promise是一个类，在这个类中有一个属性标识它当前所处的状态(初始化状态为pending)，并且这个类接收一个函数作为参数(该函数接收resolve和reject两个函数作为参数，但这两个参数都不是必传项)，并且该函数会立即执行，
* 因为在callbacks中存储的回调都是只能在Promise接收到的函数的其他事务处理完毕后才进行处理，所以这里对callbacks里的回调进行调用时候需要使用异步调用，可以使用setTimeout(因为setTimeout是宏任务，所以可能产生一些其他问题，但是在这里可以使用它来暂时模拟)，在node中可以使用process.nextTick
* 也就是说succeed是在resolve之后才执行的，然后当我再写一个then的时候，其中的succeed将会在第一个then的succeed执行完毕后执行，这时候我们就需要一个数组对then传入的succeed函数进行存储，而调用then实际上就是将这些succeed函数放入这个数组中，失败时候的fail函数同理放入到callbacks中:

# 小程序和H5有什么区别?

1.运行环境

微信小程序的运行环境是一个不完整的浏览器和基于浏览器内核重构的内置解析器。通过定制开发语言标准，可以大大提高微信小程序的流畅性。

H5的运营环境包括浏览器，包括webview。

2.开发成本

小程序开发不需要考虑浏览器兼容性和漏洞。微信为需要代码填充和低开发成本的小程序提供统一的开发工具和开发标准。

H5开发需要从开发工具、前端框架、模块管理、任务管理工具、用户界面库、界面调用工具、浏览器兼容性等方面综合考虑。与小程序相比，H5的开发成本更高。

3.系统权限

微信小程序可以通过微信获取系统权限，如网络通讯、数据缓存、用户信息等。

H5获得的系统权限较少，所以H5的大多数应用程序逻辑简单，功能单一。

4.用户体验

小程序的功能类似于APP，用户可以像普通应用程序一样操作小程序。此外，由于小程序是相对便携的，它们可以进入下一个页面而不等待。

H5本质上是一个网页，当它打开时需要被加载，这个加载过程对用户来说是不敏感的，不管它的长度如何。

# TypeScript

## 1：什么是Typescript？

Typescript是强类型的Javascript超集，支持ES6语法，支持面向对象编程的概念，如类、接口、继承、泛型等。Typescript并不直接在浏览器上运行，需要编译器编译成纯Javascript来运行。

## 2：说说Typescripy和Javascript的区别？



## 3：说说Typescript的优缺点？

**优点：**

1：快速简单，易于学习。

2：编译时提供错误检查， 在代码运行前就会进行错误提示。

3：支持所有的JS库。

4：支持ES6，提供了ES6所有优点和更高的生产力。

5：使用继承提供可重用性。

6：有助于代码结构。

7：通过定义模块来定义命名空间。

**缺点：**

1：需要长时间的来编译代码。

2：在使用第三方库时，需要有三方库的定义文件，并不是所有三方库都提供了定义文件，提供的定义文件是否准确也值得商榷。

## 4：Typescript有哪些基础类型？

1：number

2：string

3：boolean

4：Symbol

5：Array

6：Tuple(元组)

7：enum(枚举)

8：object

9：never

表示那些永不存在的值类型。如总是抛出异常或者根本不会有返回值的函数的返回值类型。

10：void

与any相反表示没有任何类型。函数没有返回值时用void。

11：null和undefined

它们是所有类型的子类型。当你指定structNullChecks时，它们只能赋值给void或者它们自己本身。

12：any

## 5：如何编译Typescript?

tsc xxx.ts

## 6：如何将多个ts文件合并成一个js文件？

tsc --outfile compact.js file1.ts file2.ts file3.js

## 7：如何自动编译ts文件并实时修改？

tsc --watch file.ts

## 8：什么是TS接口？说说它有哪些特性？

TS的核心原则之一就是对值所具有的结构进行类型检查。

它有时被称为“鸭式辩型法”或“结构性子类型化”。

其作用就是为这些类型进行命名，或为你的代码或者三方代码定义契约。

**特点：**

1：定义对象、数组、函数、类等。

2：接口可以相互继承

3：接口可以继承类

4：可选属性与额外检查

## 9：如何理解Typescript中的类？并说说它有什么特性？

Typescript是一种面向对象的Javascript语言，和其他任何面向对象编程的强语言一样，类是描述某一组对象共有属性状态或行为的实体。它就是构建具体对象实例的模板和蓝图。

**特性：**

1：继承

2：多态

3：抽象

4：封装

5：实例

## 10：Typescript支持哪些面向对象术语？

1：类

2：继承

3：多态

4：抽象

5：泛化

6：接口封装

7：实例化

等等...

## 11：如何调用基类中的构造函数？

super()

## 12：如何实现类的继承？

extends

## 13：Typescript中的模块是什么？

Typescript1.5后为了与ES6术语保持一致，内部模块都称为命名空间，外部模块简称模块。

模块在自身的作用域里执行，并不是全局作用域。这就意味着模块类的类、函数、对象等对外都是不可见的。除非你通过export导出，import导入。

模块通过使用模块加载器导入另一个模块。在运行时，模块加载器负责在执行模块之前定位和执行模块的所有依赖项。JavaScript中最常用的模块加载器是用于Node.js的CommonJS模块加载器和用于Web应用程序的require.js模块加载器。

**特别说明：**

为了支持CommonJS和AMD语法中的exports，TS提供了export = 语法，引入方式为import xxx = require("xxx")；

## 14：解释下Typescript的装饰器是什么？

装饰器是一种特殊类型的声明，它能被附加在类、方法、属性、访问符、参数上。

装饰器使用@expression这种方式，expression求值后必须为一个函数，它在运行时调用，被装饰器声明的信息作为参数传入。

## 15：什么是Mixins?

一种通过重用组件构建类的方法。

不通过类的直接继承来实现，而是将基类作为接口来实现。对于基类实例化部分在子类中实现，基类中原型的部分在子类中进行声明占位，然后通过一个Minxin函数将基类上的原型属性拷贝到子类上。

## 16：TSD是什么？

TSD是Typescript的包管理工具，我们都知道在.ts文件中引入第三方库时，第三库是需要.d.ts声明文件的，否则三方库在.ts中是无法识别报错的。

TSD就是帮我查找对应的三方库TS声明文件并下载安装。

使用过程如下：

1：npm install tsd -g

2: tsd init

3: tsd query xxx三方库 --action install

4：在使用的.ts文件中通过reference指向该声明文件

/// <reference path="typings/jquery/jquery.d.ts" />

## 17：Declare关键字是干嘛用的？

我们在.ts中使用的第三方库时没有.d.ts声明文件的时候，我们可以通过declare来写申明文件。

可以声明该模块，甚至可以直接声明一个值为any的同名的变量，然后我们就可以在代码中直接使用该三方库了。

## 18：如何让.ts文件自动生成对应的.d.ts声明文件？

tsc --declaration test.ts

## 19：tsconfig.json文件有什么作用？

该文件存在于Typescript项目的根目录里，其作用是指定相关选项告诉ts编译器如何编译ts文件。

## 20：说说什么Typescript中的泛型？作用是什么？

泛型代表的是泛指某一类型，更像是一个类型变量。由尖括号包裹<T>。

主要作用是创建逻辑可复用的组件。

泛型可以作用在函数、类、接口上。

函数：

function greet<T>(name: T) {}

类：

class createObj<T> {

    name: T

}

接口：

interface IF<T> {

    name: T

}

泛型还可以被约束，这样就是任意类型了。

interface TIF {

    length: number

}

function test<T extends TIF>(params: T) {

    console.log("=========>>>", params.length);

}

泛型约束之类型参数

function getPropoty<T, K extends keyof T>(obj: T, key: K) {

    return obj[key];

}

## 21：说说接口和类型别名type的区别？

他们很相似，type可以作用于原始值，联合类型，元组以及其它任何你需要手写的类型。

区别一：它并不会真的创建一个新的名字，当你在编译器上将鼠标悬停在定义为该类型别名定义的变量上时返回的是该类型别名引用的对象。相反，接口会创建一个新名字 ，当你把鼠标悬停在该接口定义的变量上时返回的是该接口名。

区别二：类型别名不能extends和implements

区别三：对于元组，联合类型我们一般使用类型别名type。

## 22：什么是Typescript映射文件？

.map源映射文件

它是编译后的.js与源文件之间的映射文件。调试器使用该文件，使我们可以直接调试Typescript文件而不是编译后的JS文件。

## 23：什么是类型断言？

类型断言对运行没有什么影响，仅供编译器使用。

向编译器提供我们所希望的分析代码的提示。

表示断言的两种方式：

1：<类型>变量

2：变量 as 类型 （在tsx中只能使用这种方式）

## 24：什么是Rest参数？

在不使用arguments对象的情况允许我们的函数传递可变数量的参数的另一种实现方式。

表示方式是...params。

**rest参数的规则是：**

1：一个函数只能有一个rest参数。

2：它只能出现在参数列表的最后一个。

3：该参数必须是数组类型。

## 25：什么是枚举？

枚举可以使我们定义一些带名字的常量，用于清晰的表达意图和创建一组有区别的用例。

枚举主要分为两类。一类是基于数字的，有自增长和反向映射的特性。一类是基于字符串的。

当然还有混合了这两种基础类型的枚举，我们叫做异构枚举。

## 26：说说TS的模块解析策略，什么是相对导入？什么是非相对导入？

可以通过moduleResolution属性来设置解析模式。

## 27：什么是声明合并？

声明合并是编译器将2个或多个同名声明合并为一个，合并后的声明拥有被合并声明的所有特性。

目前除了类不能与其他类和变量合并外，其他声明都是可以相互合并的。

# CSS3常用语法

1、CSS3属性选择器；

2、CSS3结构伪类选择器；

3、CSS3伪类选择器；

E:[not](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "not)(s)　　E:[root](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "root)　　E:[target](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "target)

　　E:[first-child](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "first-child)　　E:[last-child](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "last-child)　　E:[only-child](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "only-child)　　E:[nth-child](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "nth-child)(n)　　E:[nth-last-child](https://www.cnblogs.com/yanggeng/p/11188285.html" \l "nth-last-child)(n)

:: before

::after

4、CSS3 2D转换；

5、CSS3动画；

6、CSS3 3D转换；

7、浏览器私有前缀。

# 手写算法

1、手写深拷贝

*// 乞丐版*

function deepCopy(obj){

return obj.JSON.Parse(JSON.Stringify(obj))

}

*// 面试够用版*

function deepCopy(obj){

if(typeof obj)

if(typeof obj == 'object'){ *// 判断是否为复杂数据源类型*

var result = obj.constructor == Array?[]:{} *// 数组还是对象*

for(let i in obj){

result[i] = typeof obj[i] == 'object'?deepCopy(obj[i]):obj[i]

}

}else{

*// 简单数据类型*

var result = obj;

}

result result;

}

2、手写浅拷贝

function clone(target){

if(target === null){

return target

}

*// 克隆 数组 和 对象*

let cloneTarget = Array.isArray(target)?[]:{}

for(const key in target){

if(target.hasOwnProperty(key)){

*// 判断是否是本身的属性*

cloneTarget[key] = target[key]

}

}

return cloneTarget

}

3、手写数组去重

*// 1.ES6的Set*

function unique(arr){

return Array.from(new Set(arr))

}

*// 2.双层 for 循环*

function unique(arr){

let result = []

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

*// 第一层遍历*

let flag = false *// 进行标记*

for(let k = i + 1,k < arr.length;k++){

if(arr[i] === arr[k]){

*// 如果后面有重复的就跳过*

flag = true

k = arr.length

}

}

if(!flag)

result.push(arr[i]) *// 没有重复的，添加到数组中*

}

return result

}

*// 3.利用数组的 indexOf 方法*

function unique(arr){

let result = []

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

if(result.indexOf(arr[i]) === -1)

result.push(arr[i])

}

return result

}

*// 4.利用 filter 方法*

function unique(arr){

return arr.filter((ele,index) => (arr.indexOf(ele) === index)) *// 过滤*

}

*// 5.利用 map 方法*

function unique(arr){

return arr.map((ele,index) => {

if(arr.indexOf(ele) === index)

return ele

})

}

*// 6.排序后进行数组*

function unique(arr){

arr.sort()

let result = []

arr.forEach((ele,index) => {

if(ele !== arr[index + 1])

return.push(ele)

})

return result

}

4、手写防抖

function debounce(fn,delay){

let timer = null

return () => {

clearTimeout(timer)

timer = setTimeout(() => (fn()),delay)

}

}

let a = debounce(() => (console.log('防抖处理')),500)

function 点击事件(){

a()

}

5、手写节流

function throttle(fn,delay = 500){

let lastTime,time

return function(){

let context = this;

let args = [].slice.call(arguments);

time = Date.now()

if(!lastTime || time - lastTime > delay){

fn.apply(context)

lastTime = time

}

}

}

function fn(){

console.log('节流')

}

let a = throttle(fn,1000)

function 点击事件(){

a()

}

6、手写 bind

Function.prototype.myBind = function(context){

if(typeof this !== 'function'){

throw new TypeError('Error')

}

*// 返回一个绑定 this 的函数，这里我们需要保存 this*

const \_this = this

const args = [...arguments].slice(1)

*// 返回一个函数*

return function F(){

*// 因为返回一个函数，我们可以 new F() 需要判断能当做构造函数吗*

if(this instanceof F){ *// 实例是 F 这个构造函数造出来的*

return new \_this(...args,...arguments)

}

return \_this.apply(context,args.concat(...arguments))

}

}

7、手写 call

Function.prototype.myApply = function(context){

context = context || window

context.fn = this // 给传入的上下文对象添加一个 fn 方法，这个 fn 方法即为 myApply 调用者

}

1、冒泡排序

function bubbleSort(arr){

*// 冒泡每次处理最大/最小值，i 代表每次最大值的位置*

for(let i = arr.length - 1;i > 0;i--){

for(let j = 0;j < i;j++){

if(arr[j] > arr[j+1]){

*// 交换两个变换值*

[arr[j],arr[j+1]] = [arr[j+1],arr[j]]

}

}

}

}

2、选择排序

function selectSort(arr){

*// 排出 lenth - 1 个最小值即可*

for(let i = 0,len\_i = arr.length - 1;i < len\_i;i++){

let index = i

for(let j = i + 1,len\_j = arr.length;j < len\_j;j++){

if(arr[index] > arr[j]){

index = j

}

}

*// 判断最小索引是否改变*

if(index !== i){

[arr[i],arr[index]] = [arr[index],arr[i]]

}

}

}

3、插入排序

function insertSort(arr){

*// 将要插入的索引*

for(let i = 1,len = arr.length;i < len;i++){

let tmp = arr[i]

*// 插入到已经排好序的序列中*

for(let j = i - 1;j >= 0 && arr[j] > tmp;j--){

arr[j+1] = arr[j]

}

arr[j+1] = tmp

}

}

4、合并两个有序数组

function mergeArr(arr1,arr2){

*// sort 自定义排序规则*

return (arr1.concat(arr2)).sort((a,b) => a - b)

}

# 自我介绍

面试官你好，我叫刘源，来自湖南娄底。2018年山西大学本科毕业，我上一家离职的公司是 北京阳光第一车网科技有限公司 公司旗下有一个自己的项目第一车网，我主要负责的是第一车网的 pc端、webapp还有微信小程序的前端界面的设计开发，日常维护以及后期的更新迭代。我熟练掌握并且可以运用当前主流的 vue 框架，能够使用webpack、axios、vuex、vue-router等vue全家桶进行组件化、模块化开发、搭建项目。熟悉闭包，熟悉面向对象编程，熟练使用 AJAX 请求后台数据,熟练使用Element-ui、vant、echarts等常用组件库和类库，熟悉微信小程序开发流程，对微信小程序有一定的开发经验。熟练使用gulp、webpack模块化管理工具，熟练使用git代码管理工具，对nodejs有一定的了解，有良好的前后端交互能力

面试官你好，我叫刘源，来自湖南娄底。山西大学本科毕业，我上一家离职的公司是 北京阳光第一车网科技有限公司 公司旗下有一个自己的项目第一车网，我主要负责的是第一车网的 pc端、webapp还有微信小程序的前端界面的设计开发，日常维护以及后期的更新迭代。我熟练掌握并且可以运用当前主流的 react 框架，能够使用webpack、axios、redux、react-router等react全家桶进行组件化、模块化开发、搭建项目。熟悉闭包，熟悉面向对象编程，熟练使用 AJAX 请求后台数据,熟练使用Ant Design、React-Bootstrap、echarts等常用组件库和类库，熟悉微信小程序开发流程，对微信小程序有一定的开发经验。熟练使用webpack模块化管理工具，熟练使用git代码管理工具，有良好的前后端交互能力