## 移动端相关

### 1. 移动端1px边框问题的解决方案 (华润集团)

1. 使用小数写px值 通过媒体查询判断设备的像素比 DRP 根据不同像素比写不同的边框大小

.border { border: 1px solid #ccc }  
@media screen and (-webkit-min-device-pixel-ratio: 2) {  
 .border { border: 0.5px solid #ccc }  
}  
@media screen and (-webkit-min-device-pixel-ratio: 3) {  
 .border { border: 0.333333px solid #ccc }  
}

1. 使用边框图片

这样的1张6X6的图片, 9宫格等分填充border-image, 这样元素的4个边框宽度都只有1px

@media screen and (-webkit-min-device-pixel-ratio: 2){   
 .border{   
 border: 1px solid transparent;  
 border-image: url(border.gif) 2 repeat;  
 }  
}

1. 使用CSS3 box-shadow

* .shadow {  
  -webkit-box-shadow:0 1px 1px -1px rgba(255, 0, 0, 0.5);  
  box-shadow:0 1px 1px -1px rgba(255, 0, 0, 0.5);  
  }

1. viewport结合rem (推荐使用)

//devicePixelRatio=2设置meta  
<meta name="viewport" content="initial-scale=0.5, maximum-scale=0.5, minimum-scale=0.5, user-scalable=no">  
  
//devicePixelRatio=3设置meta  
<meta name="viewport" content="initial-scale=0.3333333333333333, maximum-scale=0.3333333333333333, minimum-scale=0.3333333333333333, user-scalable=no">

1. 使用 :before , :after 与 transform（推荐使用）

//第一种方法  
//构建1个伪元素, 将它的长宽放大到2倍, 边框宽度设置为1px, 再以transform缩放到50%  
.radius-border{  
 position: relative;  
}  
@media screen and (-webkit-min-device-pixel-ratio: 2){  
 .radius-border:before{  
 content: "";  
 pointer-events: none; /\* 防止点击触发 \*/  
 box-sizing: border-box;  
 position: absolute;  
 width: 200%;  
 height: 200%;  
 left: 0;  
 top: 0;  
 border-radius: 8px;  
 border:1px solid #999;  
 -webkit-transform(scale(0.5));  
 -webkit-transform-origin: 0 0;  
 transform(scale(0.5));  
 transform-origin: 0 0;  
 }  
}  
  
@media screen and (-webkit-min-device-pixel-ratio: 3) {  
 .radius-border:before{  
 content: "";  
 pointer-events: none; /\* 防止点击触发 \*/  
 box-sizing: border-box;  
 position: absolute;  
 width: 200%;  
 height: 200%;  
 left: 0;  
 top: 0;  
 border-radius: 8px;  
 border:1px solid #999;  
 -webkit-transform(scale(0.3333));  
 -webkit-transform-origin: 0 0;  
 transform(scale(0.3333));  
 transform-origin: 0 0;  
 }  
}

### 2. 移动屏幕适配方案 (华润集团)

1. 使用rem+媒体查询实现
2. 使用rem+js实现
3. 使用rem+vw实现

rem是什么是一个相对单位参照根元素的字体大小 只要动态修改跟元素字体大小所有rem单位都会跟着改变 那么rem的根元素改变参照点是谁： 设计图 以设计图大小对应的的屏幕作为参照点 假设一个设计图大小对应的真实屏幕的根元素值 例如750设计图真实屏幕是375 根元素100px 计算其他屏幕就使用 其他屏幕大小/375 \* 100 求到其他屏幕的根元素大小

## CSS相关

### 3. css3中box-shadow存在的问题 (华润集团)

1. css3中box-shadow的溢出问题 子容器加了box-shadow在父容器中可能存在溢出 可以通过给父元素添加overflow:hidden解决
2. box-shadow阴影被覆盖问题
   1. 使用相对定位解决position:relative和z-index提高层数可以把阴影显示出来
   2. 开启gpu加速, 设置3d效果 { transform: translate3d(0, 0, 0); }

## JS相关

### 1. 数组转成字符串的方式 (智理软件)

1. 使用toString方法 转成字符串

例如 var arr = [1,2,3]

arr.toString() == '1,2,3';

1. 使用join()方法 转成字符串

例如 var arr = [1,2,3]

arr.join(',') == '1,2,3'; 还可以自由控制需要什么隔开例如 - , 如果不需要什么隔开就写空 arr.join('') == '123';

### 2. JS数组的排序方式(智理软件)

1. 使用数组的sort函数可以实现排序

但是默认sort是按照ASCII编码排序的 如果需要按照数值的从小到大 从大到小需要使用第二个参数 是一个回调函数 回调函数需要两个参数 用来说明你要按照什么方式排序

例如a,b  
如果 a - b < 0 表示 a 比 b小 那么真实的a代表的数字就会排在b前面  
如果 a - b == 0 那么表示2个值相等 就按顺序排序 如果 a - b > 0 表示 a比b大 那么真实a代表的数组就会排在b后面

function sortNumber(a, b) {  
   
 return a - b // 从小到大  
 // return b - a // 从大到小  
}  
  
var arr = [10, 5, 40, 25, 1000, 1]  
  
document.write(arr + "<br />")  
document.write(arr.sort(sortNumber));// 1 5 10 25 40 1000 从小到大

1. 自己使用算法实现排序 例如 冒泡排序

var arr = [10, 20, 1, 2];  
var t;  
for(var i=0;i<arr.length;i++){  
 for(j=i+1;j<arr.length;j++){  
 if(arr[i]>arr[j]){  
 t=arr[i];  
 arr[i]=arr[j];  
 arr[j]=t;  
 }  
 }  
}  
console.log(arr); //[1, 2, 10, 20]

1. 快速排序

// 快速排序  
var quickSort = function(arr) {  
    if(arr.length < 1) {//如果数组就是一项，那么可以直接返回  
        return arr;  
    }  
    var centerIndex = Math.floor(arr.length / 2);//获取数组中间的索引  
    var centerValue = arr[centerIndex];//获取数组中间项  
    var left = [], right = [];  
    for(var i = 0; i < arr.lenght; i++){  
        if(arr[i] < centerValue){  
            left.push(arr[i]);  
        }else{  
            right.push(arr[i]);  
        }  
    }  
    return quickSort(left).contanct([centerValue], quickSort(right));//递归调用  
}

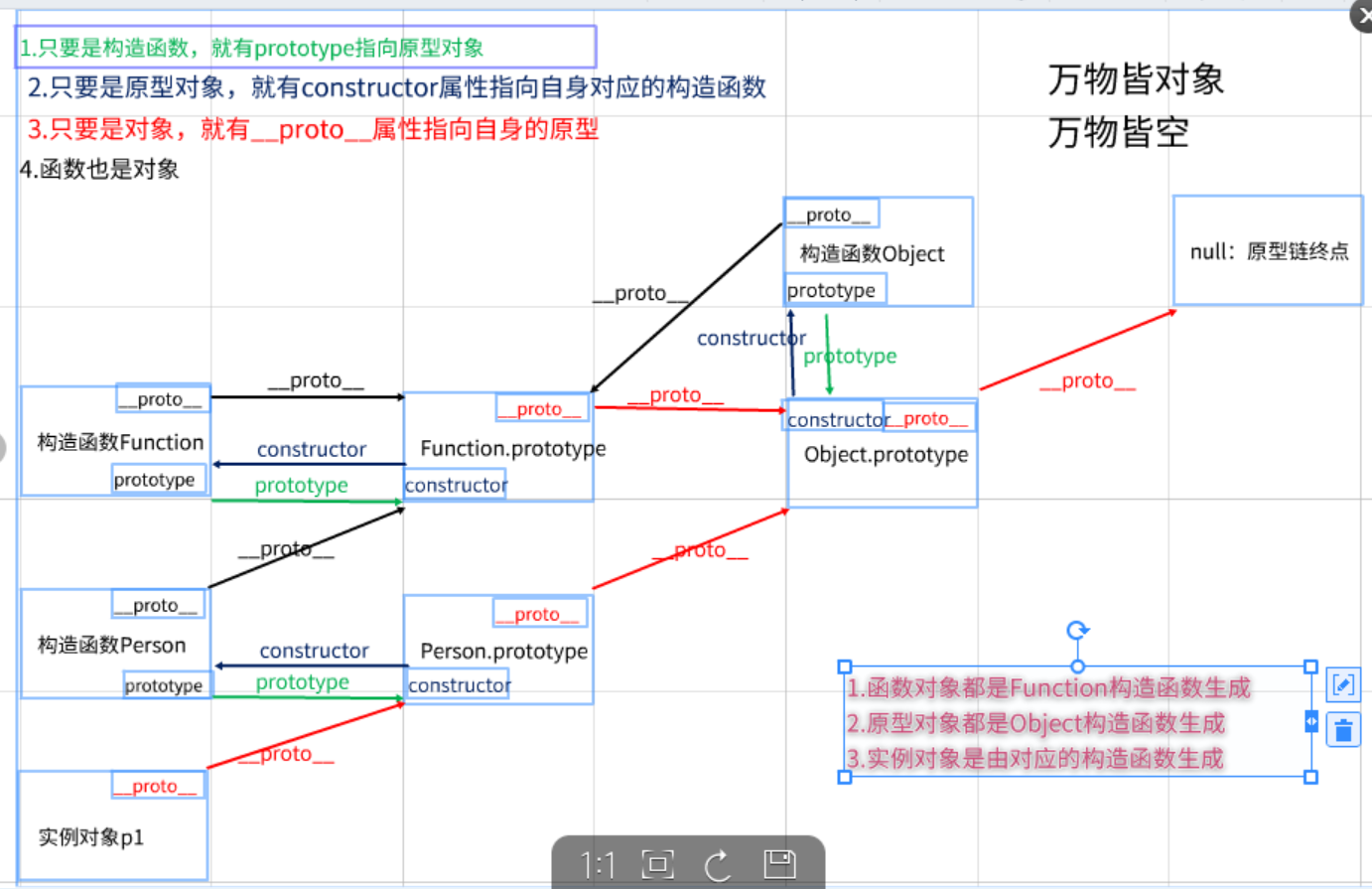
### 3. JSON转换(智理软件)

1. 将JS对象转成JSON字符串 var str = JSON.stringify(obj);
2. 将JSON字符串转成JS对象

var obj = JSON.parse(str);

### 4. 对原型链的理解(华润集团)

1. 什么是原型链 简单来说就是对象和构造函数之间连接的一个链条 简称原型链
2. 原型链的作用 可以通过原型链继承构造函数对象定义的一些公共的属性方法
3. 原型链的流程图

* 
* 原型链流程图

### 5. JS变量提升和自执行函数的面试题

(function(){  
 console.log(a);// undefined 虽然此时在函数内打印有a 因为var变量声明会提前所以不会报错 但是赋值不会提前所以a没有值是 undefined  
 var a = b = 1;   
})();   
console.log(b);//1 因为 var a = b = 1 其实b并没在里面var所以 b是全局变量并且赋值为了1所以是1  
console.log(a);// 因为 var a = b = 1; 这个var是在自执行函数里面的自执行函数是一个函数作用域 所以var a是函数内的局部变量所以外界拿不到 即是报错 a is not defined

### 6. JS this指向的面试题

var name = '王五';  
var obj = {  
 name: '张三',  
 getName:function(){  
 return this.name;  
 },  
 children:{  
 name:'李四',  
 getName:function(){  
 return this.name;  
 }  
 }  
}   
console.log(obj.getName());//张三  
console.log(obj.children.getName());//李四  
var getName = obj.children.getName;  
console.log(getName());//王五  
// 那么如果我想调用getName() 但是想打印张三怎么办 可以使用call函数改变调用当前函数的this指针  
console.log(getName.call(obj));//张三

### 7. 使用JS的类实现一个栈的类 并实现栈里面的常用API

// 定义一个栈的类 实现栈的push pop getMax getMin等函数   
// 并实现先进后出 后进先出 而且能够不断求出 当前栈里面的数据的最大值最小值  
class Stack {  
 constructor(){  
 this.dataStack = [];  
 this.maxStack = [];  
 this.minStack = [];  
  
 }  
 // 入栈  
 push(item){  
 if(this.minStack.length <= 0){  
 this.minStack.push(item)  
 }  
 else if(item <= this.getMin()){  
 this.minStack.push(item)  
 }  
  
 if(this.maxStack.length <= 0){  
 this.maxStack.push(item)  
 }  
 else if(item >= this.getMax()){  
 this.maxStack.push(item)  
 }  
 this.dataStack.push(item);   
 }  
 // 出栈  
 pop(){  
 if(this.dataStack.length <= 0){  
 return   
 }  
 let value = this.dataStack.pop();  
   
 if(value == this.getMin()){  
 this.minStack.pop()  
 }  
 if(value == this.getMax()){  
 this.maxStack.pop();  
 }  
 return value;  
 }  
 // 求最小值  
 getMin(){   
 if(this.minStack.length <=0){  
 return 0;  
 }  
 return this.minStack[this.minStack.length-1];  
 }  
 // 求最大值  
 getMax(){   
 if(this.maxStack.length <=0){  
 return 0;  
 }   
 return this.maxStack[this.maxStack.length-1];  
 }  
}   
// 创建栈的对象  
var stack = new Stack();  
// 调用入栈  
stack.push(3);  
stack.push(5);  
stack.push(10);  
stack.push(7);  
stack.push(6);  
// 调用求最大值 最小值的函数  
console.log(stack.dataStack);   
console.log(stack.getMax());  
console.log(stack.getMin());  
// 出栈  
stack.pop();  
stack.pop();  
// 出栈后的最大值最小值  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.minStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.minStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.dataStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.dataStack);  
console.log(stack.maxStack);

## 存储相关

### cookies sessionStorage和localstorage区别(华润集团)

相同点：都存储在客户端 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

1. 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。

## vue相关

### 1. react和vue有哪些不同，说说你对这两个框架的看法 (华润集团)

相同点

· 都支持服务器端渲染

· 都有Virtual DOM,组件化开发,通过props参数进行父子组件数据的传递,都实现webComponent规范

· 数据驱动视图

· 都有支持native的方案,React的React native,Vue的weex

不同点

· React严格上只针对MVC的view层,Vue则是MVVM模式

· virtual DOM不一样,vue会跟踪每一个组件的依赖关系,不需要重新渲染整个组件树.而对于React而言,每当应用的状态被改变时,全部组件都会重新渲染,所以react中会需要shouldComponentUpdate这个生命周期函数方法来进行控制

· 组件写法不一样, React推荐的做法是 JSX + inline style, 也就是把HTML和CSS全都写进JavaScript了,即'all in js'; Vue推荐的做法是webpack+vue-loader的单文件组件格式,即html,css,js写在同一个文件但是会区分开标签;

· 数据绑定: vue实现了数据的双向绑定,react数据流动是单向的

· state对象在react应用中不可变的,需要使用setState方法更新状态;在vue中,state对象不是必须的,数据由data属性在vue对象中管理

### 2. vue的双向绑定原理 (华润集团)

1. Vue.js 采用的是 数据劫持+发布/订阅模式 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（Wacther）, 触发相应的监听回调
2. 简单实现

* <!DOCTYPE html>  
  <html lang="en">  
  <head>  
   <title>双向绑定最最最初级demo</title>  
   <meta charset="UTF-8">  
  </head>  
  <body>  
   <div id="app">  
   <input type="text" id="txt">  
   <p id="show-txt"></p>  
   </div>  
  </body>  
  <script>  
   var obj={}  
   Object.defineProperty(obj,'txt',{  
   get:function(){  
   return obj  
   },  
   set:function(newValue){  
   document.getElementById('txt').value = newValue  
   document.getElementById('show-txt').innerHTML = newValue  
   }  
   })  
   document.addEventListener('keyup',function(e){  
   obj.txt = e.target.value  
   })  
  </script>  
  </html>

### 2. vue的生命周期 （华润集团 炫酷网络）

1. 什么是vue的声明周期 Vue 实例从创建到销毁的过程，就是生命周期。也就是从开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程，我们称这是 Vue 的生命周期。
2. 组件第一次加载会触发哪些声明周期函数 第一次页面加载时会触发 beforeCreate, created, beforeMount, mounted 这几个钩子
3. 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。
4. 声明周期各个函数的作用 生命周期钩子的一些使用方法： beforecreate : 可以在这加个loading事件，在加载实例时触发 created : 初始化完成时的事件写在这里，如在这结束loading事件，异步请求也适宜在这里调用 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 nextTick : 更新数据后立即操作dom
5. 当keep组件激活时会调用哪个生命周期的函数 activated 该钩子在服务器端渲染期间不被调用。
6. 当keep-alive组件停用时调用。 deactivated 该钩子在服务端渲染期间不被调用。
7. keep的调用时机 （炫酷网络）

activated调用时机：

第一次进入缓存路由/组件，在mounted后面，beforeRouteEnter守卫传给 next 的回调函数之前调用：

beforeMount=> 如果你是从别的路由/组件进来(组件销毁destroyed/或离开缓存deactivated)=>  
mounted=> activated 进入缓存组件 => 执行 beforeRouteEnter回调  
  
因为组件被缓存了，再次进入缓存路由/组件时，不会触发这些钩子：  
  
// beforeCreate created beforeMount mounted 都不会触发。  
所以之后的调用时机是：  
  
组件销毁destroyed/或离开缓存deactivated => activated 进入当前缓存组件   
=> 执行 beforeRouteEnter回调  
// 组件缓存或销毁，嵌套组件的销毁和缓存也在这里触发

1. 组件缓存和不缓存调用的生命周期

（1）不缓存： 进入的时候可以用created和mounted钩子，离开的时候用beforeDestory和destroyed钩子,beforeDestory可以访问this，destroyed不可以访问this。

（2）缓存了组件： 缓存了组件之后，再次进入组件不会触发beforeCreate、created 、beforeMount、 mounted，如果你想每次进入组件都做一些事情的话，你可以放在activated进入缓存组件的钩子中。 同理：离开缓存组件的时候，beforeDestroy和destroyed并不会触发，可以使用deactivated离开缓存组件的钩子来代替

1. 完整的触发顺序

* 触发钩子的完整顺序：  
    
  将路由导航、keep-alive、和组件生命周期钩子结合起来的，触发顺序，假设是从a组件离开，第一次进入b组件：  
    
  1. beforeRouteLeave:路由组件的组件离开路由前钩子，可取消路由离开。  
    
  2. beforeEach: 路由全局前置守卫，可用于登录验证、全局路由loading等。  
    
  3. beforeEnter: 路由独享守卫  
    
  4. beforeRouteEnter: 路由组件的组件进入路由前钩子。  
    
  5. beforeResolve:路由全局解析守卫  
    
  6. afterEach:路由全局后置钩子  
    
  7. beforeCreate:组件生命周期，不能访问this。  
    
  8. created:组件生命周期，可以访问this，不能访问dom。  
    
  9. beforeMount:组件生命周期  
    
  10. deactivated: 离开缓存组件a，或者触发a的beforeDestroy和destroyed组件销毁钩子。  
    
  11. mounted:访问/操作dom。  
    
  12. activated:进入缓存组件，进入a的嵌套子组件(如果有的话)。  
    
  13. 执行beforeRouteEnter回调函数next。

### 3. vue的computed属性 (华润集团)

1. computed用来监控自己定义的变量，该变量不在data里面声明，直接在computed里面定义，然后就可以在页面上进行双向数据绑定展示出结果或者用作其他处理
2. computed比较适合对多个变量或者对象进行处理后返回一个结果值，也就是数多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化，举例：购物车里面的商品列表和总金额之间的关系，只要商品列表里面的商品数量发生变化，或减少或增多或删除商品，总金额都应该发生变化。这里的这个总金额使用computed属性来进行计算是最好的选择
3. computed和watch的区别 watch主要用于监控vue实例的变化，它监控的变量当然必须在data里面声明才可以，它可以监控一个变量，也可以是一个对象

* data(){  
  　　　return {  
  　　　　　　　　　　example0:"",  
  　　　　　　　　　　example1:"",  
  　　　　　　　　　　example2:{  
  　　　　　　　　　　　　inner0:1, 　　　　　　　　　  
   　　　innner1:2 　　　　　　　　　  
   　}  
  　　　　　　}  
  　　　　},  
  watch:{  
  　example0(newVal,oldVal){//监控单个变量  
   ……  
   }，  
  example2(newVal,oldVal){//监控对象  
   ……  
   }，  
  }
* watch一般用于监控路由、input输入框的值特殊处理等等，它比较适合的场景是一个数据影响多个数据

计算属性可用于快速计算视图（View）中显示的属性。这些计算将被缓存，并且只在需要时更新

### 4. vuex (所有公司)

1. 什么时候需要使用vuex : 当项目业务复杂 页面之间很多关联数据需要交互的时候
2. 本地存储和vuex的区别
   1. vuex是状态管理，是为了解决跨组件之间数据共享问题的，一个组件的数据变化会映射到使用这个数据的其他组件当中。如果刷新页面，之前存储的vuex数据全部都会被初始化掉
   2. localStorage是H5提供的一个更简单的数据存储方式，之前是用cookie存放数据，但是cookie的数据量太小，所以就用localStorage，它可以有5M的限制，不受刷新页面的控制，长久保存
   3. 所以，在用vue进行项目开发的时候，什么时候用到vuex呢？

* 当应用遇到多个组件共享状态时候，即：多个视图依赖于同一个状态，不同视图的行为需要变更同一状态。
* vuex的官网也说了，对于页面之间的传参对于多层嵌套组件将会很繁琐，而且对于兄弟组件之间的状态传递无能为力。所以就将这些组件的共享状态抽取出来，以一个全局单例模式管理，即vuex
* 1.【响应式】vuex的状态存储是响应式的，当Vue组件从store中读取状态的时候，若store中的状态发生变化，那么相应的组件也会得到高效更新。  
    
  2.【不能直接改变store】不能直接改变store的变化，改变store中状态的唯一途径是commit mutation。方便于跟踪每一个状态的变化
  1. 但是localstorage改变了不会自动动态更新视图需要手动获取更新

### 5. vuex中常见的属性 (所有公司)

1. vuex有哪几种属性 有五种，分别是 State、 Getter、Mutation 、Action、 Module
2. vuex的State特性是？
   1. Vuex就是一个仓库，仓库里面放了很多对象。其中state就是数据源存放地，对应于与一般Vue对象里面的data
   2. state里面存放的数据是响应式的，Vue组件从store中读取数据，若是store中的数据发生改变，依赖这个数据的组件也会发生更新
   3. 它通过mapState把全局的 state 和 getters 映射到当前组件的 computed 计算属性中
3. vuex的Getter特性是？
   1. getters 可以对State进行计算操作，它就是Store的计算属性
   2. 虽然在组件内也可以做计算属性，但是getters 可以在多组件之间复用
   3. 如果一个状态只在一个组件内使用，是可以不用getters
4. vuex的Mutation特性是？
   1. Action 类似于 mutation，不同在于：
   2. Action 提交的是 mutation，而不是直接变更状态。
   3. Action 可以包含任意异步操作
5. Vue.js中ajax请求代码应该写在组件的methods中还是vuex的actions中？
   1. 如果请求来的数据是不是要被其他组件公用，仅仅在请求的组件内使用，就不需要放入vuex 的state里。
   2. 如果被其他地方复用，这个很大几率上是需要的，如果需要，请将请求放入action里，方便复用，并包装成promise返回，在调用处用async await处理返回的数据。如果不要复用这个请求，那么直接写在vue文件里很方便。

## ES6相关 (所有公司)

### 1.es6熟悉吗，说几个es6的新增的一些东西

1. 新增声明命令let和const
2. 模板字符串（Template String）
3. 函数的扩展(默认参数和箭头函数)
4. 对象扩展
5. import和export
6. Promise
7. 解构赋值
8. 展开运算符(...运算符)

### 2. let count的详解

1. let 和 const 都是块级作用域。以{}代码块作为作用域范围 只能在代码块里面使用。
2. 不存在变量提升，只能先声明再使用，否则会报错。在代码块内，在声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。
3. 在同一个代码块内，不允许重复声明。
4. const声明的是一个只读常量，在声明时就需要赋值。（如果 const 的是一个对象，对象所包含的值是可以被修改的。抽象一点儿说，就是对象所指向的地址不能改变，而变量成员是可以修改的。）

### 3. 模板字符串详解

用一对反引号(`)标识，它可以当作普通字符串使用，也可以用来定义多行字符串，也可以在字符串中嵌入变量，js表达式或函数，变量、js表达式或函数需要写在${ }中

var str = `abc  
def  
gh`;  
console.log(str);  
let name = "小明";  
function a() {  
 return "ming";  
}  
console.log(`我的名字叫做${name}，年龄${17+2}岁，性别${'男'}，游戏ID：${a()}`);

### 4. 函数的扩展详解

1. 可以给函数参数设置默认值

function A(a,b=1){  
 console.log(a+b);  
}  
A(1); //2  
A(2,3); //5

1. 箭头函数
   1. 可以省略function关键字
   2. 可以不写函数名
   3. 如果只有一个表达式可以不需要写{} 和 返回值

//省略写法  
var people = name => 'hello' + name;  
   
var getFullName = (firstName, lastName) => {  
 var fullName = firstName + lastName;  
 return fullName;  
}

### 5. 对象的扩展详解

1. 属性的简写。ES6 允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值
2. 方法简写

var o = {  
 method() {  
 return "Hello!";  
 }  
};  
   
// 等同于  
var o = {  
 method: function() {  
 return "Hello!";  
 }  
};

1. Object.keys()方法，获取对象的所有属性名或方法名（不包括原形的内容），返回一个数组

var obj={name: "john", age: "21", getName: function () { alert(this.name)}};  
console.log(Object.keys(obj)); // ["name", "age", "getName"]  
console.log(Object.keys(obj).length); //3  
   
console.log(Object.keys(["aa", "bb", "cc"])); //["0", "1", "2"]  
console.log(Object.keys("abcdef")); //["0", "1", "2", "3", "4", "5"]

1. Object.assign ()，assign方法将多个原对象的属性和方法都合并到了目标对象上面。可以接收多个参数，第一个参数是目标对象，后面的都是源对象

var target = {}; //目标对象  
var source1 = {name : 'ming', age: '19'}; //源对象1  
var source2 = {sex : '女'}; //源对象2  
var source3 = {sex : '男'}; //源对象3，和source2中的对象有同名属性sex  
Object.assign(target,source1,source2,source3);  
   
console.log(target); //{name : 'ming', age: '19', sex: '男'}

### 6. import和export模块化引包和导出详解

1. export用于对外输出本模块（一个文件可以理解为一个模块）变量的接口。
2. import用于在一个模块中加载另一个含有export接口的模块。
3. import和export命令只能在模块的顶部，不能在代码块之中

//导入部分  
//全部导入  
import Person from './example'  
   
//将整个模块所有导出内容当做单一对象，用as起别名  
import \* as example from "./example.js"  
console.log(example.name)  
console.log(example.getName())  
   
//导入部分  
import { name } from './example'  
   
//导出部分  
// 导出默认  
export default App  
   
// 部分导出  
export class User extend Component {};

### 7. Promise对象异步对象

1. Promise的概念和作用： Promise是异步编程的一种解决方案，将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
2. Promise的状态： 它有三种状态，分别是pending进行中、resolved已完成、rejected已失败
3. Promise语法： Promise 构造函数包含一个参数和一个带有 resolve（解析）和 reject（拒绝）两个参数的回调。在回调中执行一些操作（例如异步），如果一切都正常，则调用 resolve，否则调用 reject。对于已经实例化过的 promise 对象可以调用 promise.then() 方法，传递 resolve 和 reject 方法作为回调。then()方法接收两个参数：onResolve和onReject，分别代表当前 promise 对象在成功或失败时

语法：

var promise = new Promise((resolve, reject) => {  
 var success = true;  
 if (success) {  
 resolve('成功');  
 } else {  
 reject('失败');  
 }  
}).then(  
 (data) => { console.log(data)},  
 (data) => { console.log(data)}  
)

执行过程：

setTimeout(function() {  
 console.log(0);  
}, 0);  
var promise = new Promise((resolve, reject) => {  
 console.log(1);  
 setTimeout(function () {  
 var success = true;  
 if (success) {  
 resolve('成功');  
 } else {  
 reject('失败');  
 }  
 },2000);  
}).then(  
 (data) => { console.log(data)},  
 (data) => { console.log(data)}  
);  
console.log(promise); //<pending> 进行中  
setTimeout(function () {  
 console.log(promise); //<resolved> 已完成  
},2500);  
console.log(2);  
   
//1  
//Promise {<pending>}  
//2  
//0  
//成功  
//Promise {<resolved>: undefined}

### 8. 解构赋值详解

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）

1. 数组的解构赋值 数组中的值会自动被解析到对应接收该值的变量中，数组的解构赋值要一一对应 如果有对应不上的就是undefined

* var [name, pwd, sex]=["小周", "123456", "男"];  
  console.log(name) //小周  
  console.log(pwd)//123456  
  console.log(sex)//男

1. 对象的解构赋值 对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值

* var obj={name:"小周", pwd:"123456", sex:"男"}  
  var {name, pwd, sex}=obj;  
  console.log(name) //小周  
  console.log(pwd)//123456  
  console.log(sex)//男  
    
  //如果想要变量名和属性名不同，要写成这样  
  let { foo: foz, bar: baz } = { foo: "aaa", bar: "bbb" };  
  console.log(foz) // "aaa"  
  console.log(foo) // error: foo is not defined

### 9. 展开运算符(...运算符)

1. 将字符串转成数组

* var str="abcd";  
  console.log([...str]) // ["a", "b", "c", "d"]

1. 将集合转成数组

* var sets=new Set([1,2,3,4,5])  
  console.log([...sets]) // [1, 2, 3, 4, 5]

1. 两个数组的合并

* var a1=[1,2,3];  
  var a2=[4,5,6];  
  console.log([...a1,...a2]); //[1, 2, 3, 4, 5, 6]

1. 在函数中，用来代替arguments参数 ```js rest参数  …变量名称

rest 参数是一个数组 ，它的后面不能再有参数，不然会报错

function func(...args){ console.log(args);//[1, 2, 3, 4] } func(1, 2, 3, 4);

function f(x, ...y) { console.log(x); console.log(y); } f('a', 'b', 'c'); //a 和 ["b","c"] f('a') //a 和 [] f() //undefined ```

## 2020-4-14更新

### 8. JS笔试题对象赋值属性考察

var fun = function(){  
  
 }  
 fun.prototype = {  
 info:{  
 name:'张三',  
 age:20  
 }  
 }  
 var a = new fun();  
 var b = new fun();  
   
 a.info.name = '李四';  
 b.info.name = '王五';  
 console.log(a.info.name) // 王五  
 console.log(b.info.name) // 王五   
 // 为何当前两个对象的info.name都输出王五 因为构造函数fun 并没有给new出来的每个实例对象赋予单独的info 对象这个属性   
 // 因为原型上的对象属性继承给实例后其实是把对象的内存地址和a.info b.info关联起来 其实a.info b.info都指的是同一个info对象  
 // 那么a.info.name 和 b.info.name都找的是fun原型属性上的info   
 // 所以修改的是同一个值 取出来也是同一个值 故都是王五  
  
 var fun = function () {  
  
 }  
 fun.prototype = {  
 name: '张三',  
 age: 20  
 }  
 var a = new fun();  
 var b = new fun();  
 a.name = '李四';  
 b.name = '王五';  
 console.log(a.name) // 李四  
 console.log(b.name) // 王五  
 // 为何此时打印的a 和 b的name又分别是 李四 和 王五   
 // 因为name是普通属性 构造函数创建实例后 原型上的普通属性会继承到实例对象上会形成2个不同的变量 所以a.name 和 b.name 一开始虽然值一样但是他们代表的变量不一样  
 // 所以赋值改变不同后 a.name 和 b.name的属性是改变后的属性  
  
 var fun = function () {  
 this.info = {  
 name: '张三',  
 age: 20  
 }  
 }  
  
 var a = new fun();  
 var b = new fun();  
 a.info.name = '李四';  
 b.info.name = '王五';  
 console.log(a.info.name)// 李四  
 console.log(b.info.name)// 王五  
 // 为何此时打印的又是李四 王五 因为 fun构造函数里面的 this当前实例对象 就是分别是 a 和 b 是两个不一样的实例对象   
 // 是分别给实例对象添加info对象属性 那么此时 a.info 和 b.info 是分别属于对象自己的 a.info.name 和 b.info.name也是不同的2个属性  
 // 所以赋值取值后也是分别的

### 9. JS面试题 严格模式和非严格模式的考察

(function(){  
   
 var a = b = 10;  
 })();  
   
 console.log(b) // 10  
 // 为什么是b是10 呢 因为 上面的表达式虽然是写在一个只执行函数里面(闭包函数里面) 理论上 a和b在外面是无法访问到的  
 // 但是由于表达式是 var a = b = 10; 其实只声明了a 并没有var声明b 所以a是函数内的局部变量   
 // 但是b由于没有被声明 在非严格模式下 其实是一个全局变量 所以外部也能访问到打印就是10  
 (function(){  
 'use strict'  
 var a = b = 10;  
   
 })();  
  
 console.log(b) // 报错 b is not defined  
 // 为什么是此时都是未定义呢 因为 上面的表达式虽然是写在一个只执行函数里面(闭包函数里面) 理论上 a和b在外面是无法访问到的  
 // 但是由于表达式是 var a = b = 10; 其实只声明了a 并没有var声明b 所以a是函数内的局部变量   
 // 但是b由于没有被声明 但是当前模式是严格模式 严格模式下不允许使用未声明的变量 所以b是不存在的 无论在哪里打印都会报错不存在

### 10. 函数声明和函数表达式的区别

(function(){  
 console.log(typeof foo) // function 为何此时foo打印的是function 因为 函数声明会提前虽然在后面定义的函数foo其实声明会在最前面  
 console.log(typeof bar) // undefined 为何此时打印bar是undefined 因为bar是 函数表达式 相当于把函数通过表达式赋值给变量bar 表达式赋值不会提前所以此时还未定义就是undefined  
 var foo = 'hello',  
 bar = function(){  
 return 'world';  
 }  
 function foo(){  
 return 'hello'  
 }  
 })()

### 11. JS中什么是伪数组 有哪些常见的伪数组 如何转成普通数组

伪数组: 就是拥有数组的可以存储多个值的特性并拥有长度length属性 但是不拥有普通数组的内置方法 例如push pop slice等方法 哪些是常见的伪数组 DOM元素结构 jq获取的元素 这些都是伪数组

伪数组转化成普通数组 需要用到数组原型中的方法 slice

//需要使用到call方法，因为arr伪数组并没有该方法，需要使用call来借调 Array对象的slice方法 console.log(Array.prototype.slice.call(arr));

或者使用ES6的 console.log(Array.from(arr)) 函数也可以转化成普通数组

### 12. 函数返回值和 函数里面的this指向问题

```js var x = 20; var a = { x:15, fun:function(){ var x = 30; return function(){ return this.x; }

}  
}  
console.log(a.fun()) // ƒ (){return this.x;} a.fun的调用后的返回值是一个函数 所以输出就是当前返回的这个函数  
console.log((a.fun())()) // 20 a.fun() 调用后返回的函数再被表达式()包裹了起来 然后再调用 就是把当前返回的函数重新调用 但是当前返回的函数并没有赋值给谁那么调用者this指向window window的x是20  
console.log(a.fun()()) // 20 a.fun() 调用后返回的函数 然后再调用 就是把当前返回的函数重新调用 但是当前返回的函数并没有赋值给谁那么调用者this指向window window的x是20  
console.log(a.fun()() == (a.fun())()) // true 因为两种调用方法返回的结果都是20 所以为true  
console.log(a.fun().call(this)); // 20 将返回的函数使用call然后再传入this 此时的this还是window 所以函数里面的this也是window 所以还是外面的20  
console.log(a.fun().call(a)); // 15 将返回的函数call然后传入a 此时的函数里面的this指向的 当前对象a a有一个x为15 所以是a的15

```

### 13. 定义一个JS数组去重的方法

```js //1: 思路：定义一个新数组，并存放原数组的第一个元素，然后将元素组一一和新数组的元素对比，若不同则存放在新数组中。

function unique(arr) {  
 let newArr = [arr[0]];  
 for (let i = 1; i < arr.length; i++) {  
 let repeat = false;  
 for (let j = 0; j < newArr.length; j++) {  
 if (arr[i] === newArr[j]) {  
 repeat = true;  
 break;  
 }else{  
   
 }  
 }  
 if (!repeat) {  
 newArr.push(arr[i]);  
 }  
 }  
 return newArr;  
 }  
  
 console.log(unique([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //2: 思路：先将原数组排序，在与相邻的进行比较，如果不同则存入新数组。  
  
 function unique2(arr) {  
 var formArr = arr.sort()  
 var newArr=[formArr[0]]  
 for (let i = 1; i < formArr.length; i++) {  
 if (formArr[i]!==formArr[i-1]) {  
 newArr.push(formArr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique2([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 4,5, 6, 7]  
  /3: 利用对象属性存在的特性，如果没有该属性则存入新数组。  
  
 function unique3(arr) {  
 var obj={}  
 var newArr=[]  
 for (let i = 0; i < arr.length; i++) {  
 if (!obj[arr[i]]) {  
 obj[arr[i]] = 1  
 newArr.push(arr[i])  
 }   
 }  
 return newArr  
 }  
 console.log(unique2([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //4: 利用数组的indexOf下标属性来查询。  
  
 function unique4(arr) {  
 var newArr = []  
 for (var i = 0; i < arr.length; i++) {  
 if (newArr.indexOf(arr[i])===-1) {  
 newArr.push(arr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique4([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //5: 利用数组原型对象上的includes方法。  
  
 function unique5(arr) {  
 var newArr = []  
 for (var i = 0; i < arr.length; i++) {  
 if (!newArr.includes(arr[i])) {  
 newArr.push(arr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique5([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //6: 利用数组原型对象上的 filter 和 includes方法。  
  
 function unique6(arr) {  
 var newArr = []  
 newArr = arr.filter(function (item) {  
 return newArr.includes(item) ? '' : newArr.push(item)  
 })  
 return newArr  
 }  
 console.log(unique6([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //7: 利用数组原型对象上的 forEach 和 includes方法。  
  
 function unique7(arr) {  
 var newArr = []  
 array.forEach(item => {  
 return newArr.includes(item) ? '' : newArr.push(item)  
 });  
 return newArr  
 }  
 console.log(unique7([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //8: 利用数组原型对象上的 splice 方法。  
  
 function unique8(arr) {  
 var 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;  
 }  
 console.log(unique8([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 //9: 利用数组原型对象上的 lastIndexOf 方法。  
  
 function unique9(arr) {  
 var res = [];  
 for (var i = 0; i < arr.length; i++) {  
 res.lastIndexOf(arr[i]) !== -1 ? '' : res.push(arr[i]);  
 }  
 return res;  
 }  
 console.log(unique9([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //10: 利用 ES6的set 方法。  
  
 function unique10(arr) {  
 //Set数据结构，它类似于数组，其成员的值都是唯一的  
 return Array.from(new Set(arr)); // 利用Array.from将Set结构转换成数组  
 }  
 console.log(unique10([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));

```

### 14. 定义一个长度为10 的数组并赋值为0-9

```js // 最简单最笨的方法 但是这样写的话 就显得的初级水平 如果定义长度为100 赋值为0-99就会很麻烦了 考点其实是数组的方法 var arr = [0,1,2,3,4,5,6,7,8,9]

// 一、使用Array.apply  
let arr= Array.apply(null, { length: 10 }).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
//原理：Array.apply的第二个参数是类数组调用Array.apply(null, { length: 10 })等于生成了长度为10内容都为undefinded的数组  
  
// 二、使用Array.from  
let arr= Array.from({length:10}).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
// 三、使用Array.prototype.fill  
let arr= new Array(10).fill(1).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
//说明一下：new Array(10).fill(1)生成一个长度为10的空数组并用1填充

```

### 15. 封装一个JS函数满足以下要求

```js var num1 = 1; var num2 = 2; var num3 = 3; var num4 = 4; // computed 函数 计算函数 但是本身不计算 返回一个计算函数 再调用的时候在计算 把当前计算函数的参数和调用计算返回值的参数累加起来计算 function computed(num1, num2) { return function (num) {  
return num+num1+num2; } } // add 函数+函数 返回+当前数字 function add(num) { return +num; } // mutiply函数 -函数 返回-当前数字 function mutiply(num) { return -num; } console.log(computed(mutiply(num2), add(num3))(num1)); // 结果等于 num1-num2+num3 == 2

var num1 = 1;  
 var num2 = 2;  
 var num3 = 3;  
 var num4 = 4;  
  
 function computed(num1, num2,num3) {  
 return function (num) {   
   
 return num+num1+num2+num3;  
 }  
 }  
  
 function add(num) {  
 return +num;  
 }  
  
 function mutiply(num) {  
 return -num;  
 }  
 // console.log(computed(mutiply(num2), add(num3))(num1)); // num1-num2+num3 == 2  
 console.log(computed(add(num2),mutiply(num3), add(num4))(num1)); // num1+num2-num3+num4 == 4  
  
  
 // 前面的方法如果有很多个数需要计算方法就要写很多次无法实现复用 下面是改造版本  
 var num1 = 1;  
 var num2 = 2;  
 var num3 = 3;  
 var num4 = 4;  
 // ...args表示把所有单个项参数装在一个数组args里面  
 function computed(...args) {  
 return function (num) {   
 console.log(args)  
 // 获取所有参数数组args 遍历 然后累加到num上   
 args.map((item) => {  
 num+=item  
 })  
 // 最后return num  
 return num  
 }  
 }  
  
 function add(num) {  
 return +num;  
 }  
  
 function mutiply(num) {  
 return -num;  
 }  
 console.log(computed(add(num2),mutiply(num3), add(num4))(num1)); // num1+num2-num3+num4 == 4

```

### 16. 使用观察者模式实现一个事件监听器 包含 监听事件on 删除监听off 触发监听emit 满足以下执行结果

class EventEmitter {  
 constructor() {  
 this.\_events = {}  
 }  
  
 on(event, callback) { //监听event事件，触发时调用callback函数  
 let callbacks = this.\_events[event] || []  
 callbacks.push(callback)  
 this.\_events[event] = callbacks  
 return this  
 }  
 off(event, callback) { //停止监听event事件  
 let callbacks = this.\_events[event]  
 this.\_events[event] = callbacks && callbacks.filter(fn => fn !== callback)  
 return this  
 }  
 emit(...args) { //触发事件，并把参数传给事件的处理函数  
 const event = args[0]  
 const params = [].slice.call(args, 1)  
 const callbacks = this.\_events[event]  
 callbacks.forEach(fn => fn.apply(this.params))  
 return this  
 }  
 once(event, callback) { //为事件注册单次监听器  
 let wrapFanc = (...args) => {  
 callback.apply(this.args)  
 this.off(event, wrapFanc)  
 }  
 this.on(event, wrapFanc)  
 return this  
 }  
  
}  
  
function query(val) {  
 console.log("query");  
}  
  
function query2(val) {  
 console.log("quer2");  
}  
  
  
var emitter = new EventEmitter();  
console.log(emitter)  
emitter.on("query", query);  
emitter.on("query", query2);  
emitter.emit("query"); // query query2  
emitter.off("query", query);  
emitter.emit("query"); // query2

### 4. 完成左中右布局 左边固定200px 中间自适应右边固定100px 中间部分优先渲染 并解释原理

.box{  
 padding-left: 200px;  
 padding-right: 100px;  
 height: 100px;  
 position: relative;  
 box-sizing:border-box;  
}  
.middle{  
 height: 100px;  
}  
.left{  
 width: 200px;  
 height: 100px;  
 position: absolute;  
 left: 0;  
 top: 0;  
}  
.right{  
 width: 100px;  
 height: 100px;  
 position: absolute;  
 right: 0;  
 top: 0;  
}

<div class="box">  
 <div class="middle"></div>   
 <div class="left"></div>  
 <div class="right"></div>  
</div>

1. 中间部分优先渲染其实就是指把中间的盒子放到最前面写 html的元素是从上往下执行 所以写在越前面越优先渲染
2. 布局 思路就是把左右2遍固定 宽高 分别定位到左右2边 然后父元素相对定位 并且给父元素设置左右的padding 把中间的content内容区域往里面挤 如果父元素的宽度要固定 最好加上一个box-sizing:border-box;防止padding 撑大盒子范围

### 5. 使用CSS画一个扇形 并说出实现原理

.sector1 {  
 width: 100px;  
 height: 100px;  
 border-radius: 100px 0 0;  
 background: #ff0000;   
 /\* 原理是利用 边框圆角的属性设置盒子的某一个角为圆角 然后其他角设置为0 然后给整个盒子加背景色就只显示出一个扇形角 \*/  
}  
  
.sector2 {  
 width: 0;  
 height: 0;  
 border-radius: 100px;  
 border-width: 100px;  
 border-style: solid;  
 border-color: red transparent transparent transparent;  
 /\* 原理是利用边框颜色可以分别设置的属性 设置其中一条边框有颜色其他边框没有颜色 在把整个盒子变为圆角盒子没有宽高只显示出边框部分 突出扇形 \*/  
}

## 2021-05-17更新

### 1.ts中的数据类型

ts必须指定数据类型（给人理解将数据类型分成3种） 1.js有的类型 boolean类型、number类型、string类型、array类型、undefined、null 2.ts多出的类型 tuple类型（元组类型）、enum类型（枚举类型）、any类型（任意类型） 3.特别的类型 void类型（没有任何类型）表示定义方法没有返回值 never类型：是其他类型（包括null和undefined）的子类型，代表从不会出现的值 这意味着声明never变量只能被never类型所赋值

// 第一种定义array类型方法  
 var arr1:number[] = [1,2,3]  
 // 第二种定义array类型方法  
 var arr2:Array<number> = [11,22,33]   
  
 // 定义元组类型的方法  
 let arr3:[number,string] = [111,'111']  
  
 // 定义enum枚举类型方法（在程序中用自然语言和计算机状态联系起来，方便理解）  
 enum Flag {success=1,error=2}  
 let s:Flag = Flag.success  
 console.log(s)  
  
 enum Color {red,blue,orange}  
 let num:Color = Color.red  
 console.log(num)  
  
 // 定义any任意类型方法  
 var num1:any = 123  
 num1 = true  
 var obox:any = document.getElementById('box')  
 obox.style.color = 'red'  
  
 // undefined类型  
 var num2:number | undefined  
 console.log(num2)  
  
 // void类型,函数没有返回值  
 function run():void{  
 console.log('run')  
 }  
  
 function run1():number{  
 return 123  
 }  
  
 // never类型定义方法  
 var a:undefined  
 a = undefined  
  
 var b:null  
 b = null  
  
 // var c:never  
 // c = (()=>{  
 // throw new Error('错误')  
 // })()

### 2. Ts函数

1. ts函数定义  
 // es5函数声明  
 function run3(){  
 return 'run'  
 }  
 // es5匿名函数  
 var run4 = function(){  
 return 'run'  
 }  
  
 // ts函数声明  
 function run5():string{  
 return 'run'  
 }  
 // ts匿名函数  
 var run6 = function():number{  
 return 123  
 }  
  
 // ts中定义方法传参  
 function getInfo(name:string,age:number):string{  
 return 'info'+`$(name)---$(age)`  
 }  
 var getInfo1 = function(name:string,age:number):string{  
 return 'info'+`$(name)---$(age)`  
 }  
 // 没有返回值的方法  
 function getInfo2():void{  
 console.log(123)  
 }  
  
  
2.方法可选参数，在参数后面加？变为可选参数，可选参数必须配置到参数的最后面  
  
 function getInfo3(name:string,age?:number):string{  
 if(age){  
 return 'info'+`$(name)---$(age)`  
 }else{  
 return 'info'+`$(name)---年龄保密`  
 }  
 }  
  
3.默认参数 es6和ts都可以设置默认参数  
  
function getInfo4(name:string,age:number=20):string{  
 if(age){  
 return 'info'+`$(name)---$(age)`  
 }else{  
 return 'info'+`$(name)---年龄保密`  
 }  
 }  
 getInfo4('张三',30)  
  
  
4.剩余参数  
  
function sum(a:number,b:number,c:number):number{  
 return a+b+c  
 }  
 sum(1,2,3)  
 // 三点运算符接收传过来的值  
 function sum1(...result:number[]):number{  
 var sum = 0  
 for(var i=0;i<result.length;i++){  
 sum += result[i]  
 }  
 return sum  
 }  
 sum1(1,2,3)  
  
 function sum2(a:number,...result:number[]):number{  
 var sum = 0  
 for(var i=0;i<result.length;i++){  
 sum += result[i]  
 }  
 return a + sum  
 }  
 sum1(1,2,3)  
  
  
5.函数重载  
  
// java重载是指两个或两个以上同名函数，但是函数参数不同，这时候会出现函数重载的情况  
 // ts重载是指通过一个函数提供多个函数定义来试下多种功能的目的  
 function getInfo5(name:string):string  
 function getInfo5(age:number):number  
 function getInfo5(str:any):any{  
 if(typeof str == 'string'){  
 return str  
 }else{  
 return str  
 }  
 }  
 alert(getInfo5(123))  
 // 方法重载可以和函数选择传参一起用  
  
6.箭头函数  
  
setTimeout(function(){  
 alert('run')  
 },1000)  
 setTimeout(()=>{  
 alert('run')  
 },1000)

### 3. Ts中的类

//1.  
 function Person1(name,age){  
 this.name='zhangsan'  
 this.age=20  
 this.run = function(){  
 alert('yundong')  
 }  
 }  
 Person.prototype.sex = '男'  
 Person.prototype.work = function(){  
 alert('work')  
 }  
 var p = new Person1('zhangsan')  
 //2.  
 class Person3{  
 name:string //属性，前面省略了public关键词  
 constructor(name:string){ //构造函数 实例化类的时候触发的函数  
 this.name = name  
 }  
 getName():string{  
 return this.name  
 }  
 setName(name:string):void{  
 this.name=name  
 }  
 }  
 var p1 = new Person3('zhangsan')  
   
 // ts实现继承  
 class Person4{  
 name:string  
 constructor(name:string){  
 this.name=name  
 }  
 run():string{  
 return `$(this.name)在运动`  
 }   
 }  
 // var p2=new Person4('wangwu')  
 // alert(p2.run())  
 class Web4 extends Person4{  
 constructor(name:string){  
 super(name)  
 }  
 }  
 var w = new Web4()  
 alert(w.run())  
  
  
 // 3.类里面的修饰符，ts三种：public（公类、子类、类外面） protected（类外面不能访问） private（子类、类外面不能访问）  
  
 // 4.静态属性 静态方法  
 class Person{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 run(){  
 alert('这是实例方法')  
 }  
 static print(){  
 alert('这是静态方法')  
 // 静态方法没办法直接调用类里面的属性  
 }  
 }  
 var p = new Person('zhangsan')  
 p.run()  
 Person.print()

### 4. Ts中的多态

// 父类定义一个方法不去实现，让继承它的子类去实现，每一个子类有不同的表现多态属于继承  
class Animal{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 eat(){  
 console.log('吃的方法')  
 }  
}  
class Dog extends Animal{  
 constructor(name:string){  
 super(name)  
 }  
 eat(){  
 return this.name+'吃肉'  
 }  
}  
class Cat extends Animal{  
 constructor(name:string){  
 super(name)  
 }  
 eat(){  
 return this.name+'吃粮食'  
 }  
}

### 6. Ts中的抽象类

// 01.抽象类是提供其他类继承的基类，不能直接被实例化  
// 02.用abstract关键字定义抽象类和抽象方法，抽象类中的抽象方法不包含具体实现并且必须在派生类中实现  
// 03.abstract抽象方法只能在抽象类中  
// 04.抽象类和抽象方法用来定义标准：例如，要求Animal类的子类必须有eat方法  
  
abstract class Animal{  
 name:string  
 constructor(name:string){  
 this.name = name  
 }  
 abstract eat():any  
}  
class Dog extends Animal{  
 constructor(name:any){  
 super(name)  
 }  
 // 抽象类的子类必须实现抽象类里面的方法  
 eat(){  
 console.log(this.name+'吃肉')  
 }  
}  
var d = new Dog('xiaogou')  
d.eat()

### 7. Ts中的接口

接口的作用：在面向对象编程中，接口是一种规范的定义，它定义行为和动作的规范。 在程序设计里面，接口起到一定的限制和规范作用。接口定义某一些类所遵守的规范，接口不关心这些类的内部状态数据，也不关心类里面方法的实现细节 它只规定这批类中必须提供某些方法，提供的这些方法就可以满足某些需求。 ts的接口同时增加更灵活的接口类型，包括属性，函数，可索引和类等。

1.属性类接口  
  
 // 对json的约束  
 // function printLabel(label:string):void{  
 // console.log('printLable')  
 // }  
 // printLabel('hahah')  
 // 自定义方法传入参数对json进行约束  
 function printLable(labelInfo:{lable:string}):void{  
 console.log('printLabel')  
 }  
 printLable({lable:'string'})  
  
2.定义接口对参数进行约束  
  
interface FullName{  
 firstName:string;  
 secondName:string  
 }  
 function printName(name:FullName){  
 console.log(name.firstName+'---'+name.secondName)  
 }  
 var obj = {  
 firstName:'zhang',  
 secondName:'san'  
 }  
 printName(obj)  
  
3.接口：可选属性  
  
interface FullName{  
 firstName:string;  
 secondName?:string  
 // 添加？号之后 secondName可传可不传  
 }  
 function getName(name:FullName){  
 console.log(name.firstName+'---'+name.secondName)  
 }  
 var obj = {  
 firstName:'zhang',  
 secondName:'san'  
 }  
 getName(obj)  
  
ajax接口实践  
interface Config{  
 type:string  
 url:string  
 data?:string  
 dataType:string  
 }  
 function ajax(config:Config){  
 var xhr = new XMLHttpRequest()  
 xhr.open(config.type,config.url,true)  
 xhr.send(config.data)  
 xhr.onreadystatechange=function(){  
 if(xhr.readyState==4&&xhr.status==200){  
 console.log('success')  
 }  
 }  
 }  
 ajax({  
 type:'get'  
 url:'www://http:baidu.com'  
 dataType:'json'  
 })

### 8. TS中的泛型

// 1泛型的定义  
// 泛型：在软件工程中，我们不仅要创建一致的定义良好的api，同时也要考虑可重用性。组件不仅能够支持当前的数据类型，还能支持未来的数据类型  
// 在C#和Java这种语言中，可使用泛型来创建可重用的组件，一个组件支持多种类型的数据  
// 2泛型函数  
// T表示泛型，具体什么类型调用这个方法的时候决定的  
function getData<T>(value:T):T{  
 return value  
}  
getData<number>(123)  
// 3泛型类  
// 比如有个最小堆算法，需要同时支持返回数字和字符串两种类型  
// class Minclass{  
// public list:number[]=[]  
// add(num){  
// this.list.push(num)  
// }  
// min():number{  
// var minNum=this.list[0]  
// for(var i=0;i<this.list.length;i++){  
// if(minNum>this.list[i]){  
// minNum = this.list[i]  
// }  
// }  
// return minNum  
// }  
// }  
// var m = new Minclass()  
// m.add(2)  
  
class Minclass<T>{  
 public list:T[]=[]  
 add(value:T):void{  
 this.list.push(value)  
 }  
 min():T{  
 var minNum=this.list[0]  
 for(var i=0;i<this.list.length;i++){  
 if(minNum>this.list[i]){  
 minNum = this.list[i]  
 }  
 }  
 return minNum  
 }  
}  
var m = new Minclass()  
m.add(2)  
// 4泛型接口  
// 函数类型接口  
// interface Configfn{  
// (value1:string,value2:string):string;  
// }  
// var setData:Configfn=function(value1:string,value2:string):string{  
// return value1+value2  
// }  
// 泛型接口  
interface Configfn{  
 <T>(value:T):T;  
}  
var setData:Configfn=function<T>(value:T):T{  
 return value  
}

### 1. React类组件的生命周期函数

组件的生命周期可分成三个状态：  
  
Mounting：已插入真实 DOM  
Updating：正在被重新渲染  
Unmounting：已移出真实 DOM  
生命周期的方法有：  
  
componentWillMount 在渲染前调用,在客户端也在服务端。  
  
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)。  
  
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。  
  
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。  
可以在你确认不需要更新组件时使用。  
  
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。  
  
componentDidUpdate 在组件完成更新后立即调用。在初始化时不会被调用。  
  
componentWillUnmount在组件从 DOM 中移除之前立刻被调用。

### 2. React中的Context(上下文) 是什么 怎么使用

1. React.Context上下文就是解决React父子组件之间公共数据传输的一个APi 减少组件之间公共数据的重复传参 和 多级嵌套传参的问题 但是又不想使用Redux等太复杂的库时使用Context上下文用来管理全局状态

import React from 'react'  
  
// 创建 Context 填入默认值（任何一个 js 变量）  
const ThemeContext = React.createContext('light')  
  
// 底层组件 - 函数是组件  
function ThemeLink (props) {  
 // const theme = this.context // 会报错。函数式组件没有实例，即没有 this  
  
 // 函数式组件可以使用 Consumer  
 return <ThemeContext.Consumer>  
 { value => <p>link's theme is {value}</p> }  
 </ThemeContext.Consumer>  
}  
  
// 底层组件 - class 组件  
class ThemedButton extends React.Component {  
 // 指定 contextType 读取当前的 theme context。  
 // static contextType = ThemeContext // 也可以用 ThemedButton.contextType = ThemeContext  
 render() {  
 const theme = this.context // React 会往上找到最近的 theme Provider，然后使用它的值。  
 return <div>  
 <p>button's theme is {theme}</p>  
 </div>  
 }  
}  
ThemedButton.contextType = ThemeContext // 指定 contextType 读取当前的 theme context。  
  
// 中间的组件再也不必指明往下传递 theme 了。  
function Toolbar(props) {  
 return (  
 <div>  
 <ThemedButton />  
 <ThemeLink />  
 </div>  
 )  
}  
  
class App extends React.Component {  
 constructor(props) {  
 super(props);  
 this.state = {  
 theme: 'light'  
 }  
 }  
 render() {  
 return <ThemeContext.Provider value={this.state.theme}>  
 <Toolbar />  
 <hr/>  
 <button onClick={this.changeTheme}>change theme</button>  
 </ThemeContext.Provider>  
 }  
 changeTheme = () => {  
 this.setState({  
 theme: this.state.theme === 'light' ? 'dark' : 'light'  
 })  
 }  
}  
  
export default App

### 3. React Hooks常用APi

1. useState

useState 的使用非常简单，我们从 React 中拿到 useState 后，只需要在使用的地方直接调用 useState 函数就可以， useState 会返回一个数组，第一个值是我们的 state， 第二个值是一个函数，用来修改该 state 的，那么这里为什么叫 count 和 setCount？一定要叫这个吗，这里使用了 es6 的解构赋值，所以你可以给它起任何名字：如updateCount, doCount、any thing，当然，为了编码规范，所以建议统一使用一种命名规范，尤其是第二个值

function App () {  
 const [ count, setCount ] = useState(0)  
 return (  
 <div>  
 点击次数: { count }   
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 </div>  
 )  
}

我们在使用 useState 的第二个参数时，我们想要获取上一轮该 state 的值的话，只需要在 useState 返回的第二个参数，也就是我们上面的例子中的 setCount 使用时，传入一个参数，该函数的参数就是上一轮的 state 的值

setCount((count => count + 1)

1. useEffect

Effect Hook 可以让你在函数组件中执行副作用操作，这里提到副作用，什么是副作用呢，就是除了状态相关的逻辑，比如网络请求，监听事件，查找 dom

function App () {  
 const [ count, setCount ] = useState(0)  
   
 useEffect(() => {  
 document.title = count  
 })  
   
 return (  
 <div>  
 页面名称: { count }   
 <button onClick={() => { setCount(count + 1 )}}>点我</button>  
 </div>  
 )  
}

useEffect如何模拟生命周期组件加载组件销毁

useEffect(() => {  
 // 相当于 componentDidMount  
 console.log('add resize event')  
 window.addEventListener('resize', onChange, false)  
   
 return () => {  
 // 相当于 componentWillUnmount  
 window.removeEventListener('resize', onChange, false)  
 }  
 }, [])  
   
 useEffect(() => {  
 // 相当于 componentDidUpdate  
 document.title = count  
 })

1. useContext ```jsx

// 创建一个 context const Context = createContext(0)

// 组件一, Consumer 写法 class Item1 extends Component { render () { return ( <Context.Consumer> { (count) => (

{count}

) } </Context.Consumer> ) } } // 组件二, contextType 写法 class Item2 extends Component { static contextType = Context render () { const count = this.context return (

{count}

)

} } // 组件一, useContext 写法 function Item3 () { const count = useContext(Context); return (

{ count }

) }

function App () { const [ count, setCount ] = useState(0) return (

点击次数: { count }   
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 <Context.Provider value={count}>  
 <Item1></Item1>  
 <Item2></Item2>  
 <Item3></Item3>  
 </Context.Provider>  
</div>  
)

} ```

1. useMemo

useMemo 是什么呢，它跟 memo 有关系吗 ，说白了 memo 就是函数组件的 PureComponent，用来做性能优化的手段，useMemo 也是，useMemo 在我的印象中和 Vue 的 computed 计算属性类似，都是根据依赖的值计算出结果，当依赖的值未发生改变的时候，不触发状态改变

function App () {  
 const [ count, setCount ] = useState(0)  
 const add = useMemo(() => {  
 return count + 1  
 }, [count])  
 return (  
 <div>  
 点击次数: { count }  
 <br/>  
 次数加一: { add }  
 <button onClick={() => { setCount(count + 1)}}>点我</button>  
 </div>  
 )  
}

上面的例子中，useMemo 也支持传入第二个参数，用法和 useEffect 类似

不传数组，每次更新都会重新计算 空数组，只会计算一次 依赖对应的值，当对应的值发生变化时，才会重新计算(可以依赖另外一个 useMemo 返回的值) 需要注意的是，useMemo 会在渲染的时候执行，而不是渲染之后执行，这一点和 useEffect 有区别，所以 useMemo 不建议有 副作用相关的逻辑

同时，useMemo 可以作为性能优化的手段，但不要把它当成语义上的保证，将来，React 可能会选择“遗忘”以前的一些 memoized 值，并在下次渲染时重新计算它们

1. useCallback

useCallback 是什么呢，可以说是 useMemo 的语法糖，能用 useCallback 实现的，都可以使用 useMemo, 在 react 中我们经常面临一个子组件渲染优化的问题， 尤其是在向子组件传递函数props时，每次 render 都会创建新函数，导致子组件不必要的渲染，浪费性能，这个时候，就是 useCallback 的用武之地了，useCallback 可以保证，无论 render 多少次，我们的函数都是同一个函数，减小不断创建的开销

const onClick = useMemo(() => {  
 return () => {  
 console.log('button click')  
 }  
}, [])  
   
const onClick = useCallback(() => {  
 console.log('button click')  
}, [])

useCallback 的第二个参数和useMemo一样，没有区别

1. useRef

useRef 有什么作用呢，其实很简单，总共有两种用法

获取子组件的实例(只有类组件可用) 在函数组件中的一个全局变量，不会因为重复 render 重复申明， 类似于类组件的 this.xxx

// 使用 ref 子组件必须是类组件  
class Children extends PureComponent {  
 render () {  
 const { count } = this.props  
 return (  
 <div>{ count }</div>  
 )  
 }  
}  
   
function App () {  
 const [ count, setCount ] = useState(0)  
 const childrenRef = useRef(null)  
 // const   
 const onClick = useMemo(() => {  
 return () => {  
 console.log('button click')  
 console.log(childrenRef.current)  
 setCount((count) => count + 1)  
 }  
 }, [])  
 return (  
 <div>  
 点击次数: { count }  
 <Children ref={childrenRef} count={count}></Children>  
 <button onClick={onClick}>点我</button>  
 </div>  
 )  
}

使用useRef来保存state的值

function App () {  
 const [ count, setCount ] = useState(0)  
 const timer = useRef(null)  
 let timer2   
   
 useEffect(() => {  
 let id = setInterval(() => {  
 setCount(count => count + 1)  
 }, 500)  
   
 timer.current = id  
 timer2 = id  
 return () => {  
 clearInterval(timer.current)  
 }  
 }, [])  
   
 const onClickRef = useCallback(() => {  
 clearInterval(timer.current)  
 }, [])  
   
 const onClick = useCallback(() => {  
 clearInterval(timer2)  
 }, [])  
   
 return (  
 <div>  
 点击次数: { count }  
 <button onClick={onClick}>普通</button>  
 <button onClick={onClickRef}>useRef</button>  
 </div>  
 )  
}

当我们们使用普通的按钮去暂停定时器时发现定时器无法清除，因为 App 组件每次 render，都会重新申明一次 timer2, 定时器的 id 在第二次 render 时，就丢失了，所以无法清除定时器，针对这种情况，就需要使用到 useRef，来为我们保留定时器 id，类似于 this.xxx，这就是 useRef 的另外一种用法

1. useReducer

useReducer 是什么呢，它其实就是类似 redux 中的功能，相较于 useState，它更适合一些逻辑较复杂且包含多个子值，或者下一个 state 依赖于之前的 state 等等的特定场景， useReducer 总共有三个参数

1. 第一个参数是 一个 reducer，就是一个函数类似 (state, action) => newState 的函数，传入 上一个 state 和本次的 action
2. 第二个参数是初始 state，也就是默认值，是比较简单的方法
3. 第三个参数是惰性初始化，这么做可以将用于计算 state 的逻辑提取到 reducer 外部，这也为将来对重置 state 的 action 做处理提供了便利

function reducer(state, action) {  
 switch (action.type) {  
 case 'increment':  
 return {count: state.count + 1};  
 case 'decrement':  
 return {count: state.count - 1};  
 default:  
 throw new Error();  
 }  
}  
   
function App() {  
 const [state, dispatch] = useReducer(reducer, {  
 count: 0  
 });  
 return (  
 <>  
 点击次数: {state.count}  
 <button onClick={() => dispatch({type: 'increment'})}>+</button>  
 <button onClick={() => dispatch({type: 'decrement'})}>-</button>  
 </>  
 );  
}

### 17. js中的与解析

1. js 中的声明  
声明就是 变量的声明和函数的声明, 其目的是让 js 解释引擎知道有什么东西.  
声明时不参与运算的, 是不参与执行的, 是在预解析阶段就完成的.  
变量的声明  
  
// 变量的声明就是 var 变量名.  
var num = 123;  
// 这是一个语法糖,可以理解成  
var num;  
num = 123;  
函数的声明  
  
function 函数名 () { ... }  
// 在一个独立于任何语句( 表达式, if 结构, while 结构 等 )的独立结构中, 或函数中出现的代码, 为函数声明.  
2. js 预解析代码如何执行  
js 的代码执行要经理两个步骤, 首先是预解析. 预解析会通读所有代码. 如果发现错误则停下, 如果遇到声明则记录.  
在声明的时候, 如果是变量名的声明, 解析器内部就会记录下这个变量. 如果使用遍历则检查是否有记录.  
  
在声明的时候, 如果是函数声明, 则 解析器会先记录函数的名字( 相当于变量声明 ), 然后将函数的名字与函数体联系在一起.  
  
在预解析中, 如果出现重复声明, 则第一次声明起作用, 其后所有的同名的声明无效.  
  
// 例如:  
 var num = 1;  
 var num = 2;  
// 等价  
 var num = 1;  
 num = 2;  
声明结束后代码就会再从第一句话开始一句一句的执行.

### 18. js事件循环机制

在事件循环中，每进行一次循环操作称为 tick，每一次 tick 的任务处理模型是比较复杂的，但关键步骤如下：

在此次 tick 中选择最先进入队列的任务(异步队列)，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 Microtasks Queue

更新 render

主线程重复执行上述步骤在上诉tick的基础上需要了解几点：

JS分为同步任务和异步任务

同步任务都在主线程上执行，形成一个执行栈

主线程之外，事件触发线程管理着一个任务队列，只要异步任务有了运行结果，就在任务队列之中放置一个事件。

一旦执行栈中的所有同步任务执行完毕（此时JS引擎空闲），系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行

### 19. 宏任务与微任务

1. 宏任务

macrotask，可以理解是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）。

浏览器为了能够使得JS内部macrotask与DOM任务能够有序的执行，会在一个macrotask执行结束后，在下一个macrotask 执行开始前，对页面进行重新渲染，流程如下：

macrotask->渲染->macrotask->...

宏任务包括：script(整体代码)、 setTimeout、 setInterval I/O、 UI交互事件 、postMessage、 MessageChannel、 setImmediate(Node.js 环境)

### 20. 微任务

microtask,可以理解是在当前 task 执行结束后立即执行的任务。也就是说，在当前task任务后，下一个task之前，在渲染之前。

所以它的响应速度相比setTimeout（setTimeout是task）会更快，因为无需等渲染。也就是说，在某一个macrotask执行完后，就会将在它执行期间产生的所有microtask都执行完毕（在渲染前）。

微任务包括： Promise.then  、Object.observe、MutaionObserver、process.nextTick(Node.js 环境)

console.log('begin')  
  
        setTimeout(() => {    
  
//宏任务  
  
            console.log('setTimeout')  
  
        }, 0);  
  
  
  
        Promise.resolve().then(() => {  
  
//微任务  
  
            console.log('promise');  
  
        });  
  
        console.log('end');  
  
执行结果是 begin  ——>  end  ——> promise ——> setTimeout

let con = "<h2>页面渲染</h2>";  
  
        document.write(con);  
  
        console.log(1);  
  
        Promise.resolve().then(()=>{  
  
            console.log('promise  2 ');  
  
            alert('promise then')  
  
        })  
  
        setTimeout(()=>{  
  
            console.log('setTimeout 3');  
  
            alert('setTimeout')  
  
        },0)  
  
        console.log(4)  
  
执行顺序：  1  ——>  4  ——> promise 2 ——> 页面渲染 ——>  setTimeout 3  
  
由此我们可知 微任务 > DOM渲染 > 宏任务

### 浏览器的回流与重绘 (Reflow & Repaint)

1. 回流重绘的概念

浏览器使用流式布局模型 (Flow Based Layout)。 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Render Tree。 有了RenderTree，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。 由于浏览器使用流式布局，对Render Tree的计算通常只需要遍历一次就可以完成，但table及其内部元素除外，他们可能需要多次计算，通常要花3倍于同等元素的时间，这也是为什么要避免使用table布局的原因之一。

一句话：回流必将引起重绘，重绘不一定会引起回流。

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

1. 回流 的常见场景

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 激活CSS伪类（例如：:hover） 查询某些属性或调用某些方法

一些常用且会导致回流的属性和方法：

clientWidth、clientHeight、clientTop、clientLeft offsetWidth、offsetHeight、offsetTop、offsetLeft scrollWidth、scrollHeight、scrollTop、scrollLeft scrollIntoView()、scrollIntoViewIfNeeded() getComputedStyle() getBoundingClientRect() scrollTo()

1. 重绘 (Repaint) 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），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。
2. 性能影响 回流比重绘的代价要更高。 有时即使仅仅回流一个单一的元素，它的父元素以及任何跟随它的元素也会产生回流。 现代浏览器会对频繁的回流或重绘操作进行优化： 浏览器会维护一个队列，把所有引起回流和重绘的操作放入队列中，如果队列中的任务数量或者时间间隔达到一个阈值的，浏览器就会将队列清空，进行一次批处理，这样可以把多次回流和重绘变成一次。 当你访问以下属性或方法时，浏览器会立刻清空队列：

clientWidth、clientHeight、clientTop、clientLeft offsetWidth、offsetHeight、offsetTop、offsetLeft scrollWidth、scrollHeight、scrollTop、scrollLeft width、height getComputedStyle() getBoundingClientRect()

因为队列中可能会有影响到这些属性或方法返回值的操作，即使你希望获取的信息与队列中操作引发的改变无关，浏览器也会强行清空队列，确保你拿到的值是最精确的。

1. 减少回流重绘的优化方案

CSS

避免使用table布局。 尽可能在DOM树的最末端改变class。 避免设置多层内联样式。 将动画效果应用到position属性为absolute或fixed的元素上。 避免使用CSS表达式（例如：calc()）。

JavaScript

避免频繁操作样式，最好一次性重写style属性，或者将样式列表定义为class并一次性更改class属性。 避免频繁操作DOM，创建一个documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中。 也可以先为元素设置display: none，操作结束后再把它显示出来。因为在display属性为none的元素上进行的DOM操作不会引发回流和重绘。 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。 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。

### 10. 反射 Reflection

1. 为什么需要反射 ?很多强类型语言长期以来都有其反射(Reflection)API（如 Python 或 C#），而 JavaScript 作为一种动态语言，则几乎用不着反射。在 ES6 特性里引入的少量扩展之处中，允许开发者用Proxy访问此前的一些语言内部行为就算得上一项。你可能会反驳，尽管在规范和社区中没有明确那么称呼过，但 JS 在 ES5 中已经有反射特性了。诸如 Array.isArray, Object.getOwnPropertyDescriptor, 甚至 Object.keys 这些，在其他语言中都是典型的被列为反射的方法。而内置的 Reflect 对象则更进了一步，将这些方法归纳在一起。这很有用，是吧？为什么要用超类 Object的静态反射方法(如getOwnPropertyDescriptor或 create)呢？毕竟Object表示一个基本原型更合适，而不是成为反射方法的仓库。用一个专有接口暴露更多反射方法更有意义。
2. Reflect的语法 返回值 vs 通过 Object 反射

// 和 Object 中等价的 Reflect 反射方法同时也提供了更有意义的返回值。比如，Reflect.defineProperty方法返回一个布尔值，表示属性是否被成功定义；而对应的Object.defineProperty则返回其首个参数中接收到的对象 -- 这并不是很有用。举例来说，以下代码演示了Object.defineProperty如何工作  
  
try {  
 Object.defineProperty(target, 'foo', { value: 'bar' })  
 // yay!  
} catch (e) {  
 // oops.  
}  
// 相反，用Reflect.defineProperty就会感觉自然得多：  
var yay = Reflect.defineProperty(target, 'foo', { value: 'bar' })  
if (yay) {  
 // yay!  
} else {  
 // oops.  
}  
// 这种方法免去了使用try/catch 代码块，并使得代码更易维护。

### 6. CSS选择器优先级

!important，加在样式属性值后，权重值为 10000  
  
  
内联样式，如：style=””，权重值为1000  
  
  
ID选择器，如：#content，权重值为100  
  
  
类，伪类和属性选择器，如： class="content"、:hover 权重值为10  
  
  
标签选择器和伪元素选择器，如：div、p、:before 权重值为1  
  
  
通用选择器（\*）、子选择器（>）、相邻选择器（+）、同胞选择器（~）、权重值为0  
  
继承（Inherited） 权重值最低级 只要有人任意其他的选择器都会覆盖(细节)

### 7. flex布局的常用api

采用 Flex 布局的元素，称为 Flex容器（flex container），简称"容器"。它的所有子元素自动成为容器成员，称为 Flex项目（flex item），简称“项目”。 1.父元素属性 | **属性名** | **属性值** | **备注** | display | flex | 定义了一个flex容器，它的直接子元素会接受这个flex环境 | | flex-direction | row, row-reverse, column, column-reverse | 决定主轴的方向  
| flex-wrap | nowrap, wrap, wrap-reverse | 如果一条轴线排不下，如何换行  
| flex-flow | [flex-direction] , [flex-wrap] | 是 flex-direction属性和 flex-wrap属性的简写形式， 默认值为 row nowrap | | justify-content | flex-start, flex-end, center, space-between, space-around | 设置或检索弹性盒子元素在主轴（横轴）方向上的对齐方式 | | align-items | flex-start, flex-end, center, baseline, stretch | 设置或检索弹性盒子元素在侧轴（纵轴）方向上的对齐方式 | | 2.子元素属性 | |  
| 属性名 | 属性值 | 备注  
| order | [int] | 默认情况下flex order会按照书写顺训呈现， 可以通过order属性改变， 数值小的在前面，还可以是负数。 | | flex-grow | [number] | 设置或检索弹性盒的扩展比率, 根据弹性盒子元素所设置的扩展因子作为比率来分配剩余空间 | | flex-shrink | [number] | 设置或检索弹性盒的收缩比率, 根据弹性盒子元素所设置的收缩因子作为比率来收缩空间 | | flex-basis | [length], auto | 设置或检索弹性盒伸缩基准值  
| align-self | auto,flex-start,flex-end,center,baseline,stretch | 设置或检索弹性盒子元素在侧轴（纵轴）方向上的对齐方式， 可以覆盖父容器align-items的设置 |

### 8. CSS中实现0.5px边框 解决安卓兼容性问题

/\*标准1px边框\*/  
 .b1{  
 height: 40px;  
 border: 1px solid #ff0000;  
 }  
 /\*1.可以利用利用渐变样式=>实现.5px\*/  
 .a1{  
 height: 1px;  
 margin-top: 20px;  
 background-image: linear-gradient(0deg, #f00 50%, transparent 50%);  
 }  
 /\*2.可以利用缩放-发虚=>实现.5px\*/  
 .a2{  
 height: 1px;  
 margin-top: 20px;  
 background-color: #f00;  
 -webkit-transform: scaleY(.5);  
 transform:scaleY(.5);  
 }  
 /\*3.四条边框都需要的样式\*/  
 .scale-half {  
 margin-top: 20px;  
 height: 100px;  
 border:1px solid #f00;  
 -webkit-transform-origin: 0 0;  
 transform-origin: 0 0;  
 -webkit-transform: scale(.5, .5);  
 transform: scale(.5, .5);  
 }  
 /\*4.给伪元素添加设置边框\*/  
 .border3{  
 position: relative;  
 }  
 .border3:before{  
 content: '';  
 position: absolute;  
 width: 200%;  
 height: 200%;  
 border: 1px solid blue;  
 -webkit-transform-origin: 0 0;  
 -moz-transform-origin: 0 0;  
 -ms-transform-origin: 0 0;  
 -o-transform-origin: 0 0;  
 transform-origin: 0 0;  
 -webkit-transform: scale(.5, .5);  
 -ms-transform: scale(.5, .5);  
 -o-transform: scale(.5, .5);  
 transform: scale(.5, .5);  
 -webkit-box-sizing: border-box;  
 -moz-box-sizing: border-box;  
 box-sizing: border-box;  
 }

### localStorage如何设置过期时间 如何封装自定义localStorage

1. 创建Storage类

* 定义 对应的get set remove clear api
* 通过set函数添加过期时间参数来实现过期时间的记录
* 设置存储时存储当前值和过期时间
* get取值的时候先验证当前值是否存在 以及时间是否大于过期时间 如果存在且不大于过期时间既可返回对应的值否则返回空

class Storage {  
 constructor() {  
 this.storageName = 'expiredStorage'  
 }  
   
   
 /\*\*  
 \* 设置缓存  
 \* @param {string} name 缓存名称  
 \* @param {any} value 缓存的值  
 \* @param {any} expires 缓存过期时间(秒)   
 \*\*/  
 set(name, value, expires) {  
 const storages ={}  
 storages[name] = {  
 value,  
 expires: storages[name]  
 ? storages[name].expires  
 : expires === undefined  
 ? +new Date() + 31536000000 //默认365天 这个值可以自己修改  
 : expires \* 1000 + +new Date(),  
 };  
 localStorage.setItem(this.storageName, JSON.stringify(storages))  
 }  
   
 /\*\*  
 \* 获取缓存  
 \* @param {string} name 缓存名称  
 \*\*/  
 get(name) {  
 const storages = JSON.parse(localStorage.getItem(this.storageName))  
 try {  
 if (!storages[name]) {  
 // 不存在  
 return null;  
 }  
 console.log('log=====', storages[name].expires - new Date());  
 if (+new Date() > storages[name].expires) {  
 // 存在但过期  
 this.remove(name);  
 return null;  
 }  
 return storages[name].value;  
 } catch (error) {  
 console.log('[ControlStorage] the error message: get field failed\n', error);  
 }  
 }  
   
 /\*\*  
 \* 移除对应缓存  
 \* @param {string} name 缓存名称  
 \*\*/  
 remove(name) {  
 const storages = JSON.parse(localStorage.getItem(this.storageName))  
 try {  
 delete storages[name];  
 if (JSON.stringify(storages) === '{}') {  
 // 缓存字段为空对象时，删除该字段  
 this.clear();  
 return;  
 }  
 this.baseStorage.setItem(storages);  
 } catch (error) {  
 console.log('[ControlStorage] the error message: remove field failed\n', error);  
 }  
 }  
 /\*\*  
 \* 清除所有带过期时间的缓存  
 \*\*/  
 clear() {  
 localStorage.removeItem(this.storageName)  
 }  
}  
  
export default new Storage();

1. 实际调用 引入对应的Storage类 调用里面的方法传递对应参数

import storage from './js/storage.js'  
  
...  
setStorage() {  
 // 5秒过期  
 storage.set('name', 'ghkmmm', 5)  
},  
getStorage() {  
 console.log(storage.get('name'))  
},  
removeStorage() {  
 storage.remove('name')  
},