HTML基础

html语义化?

语义化是指使用恰当语义的html标签，让页面具有良好的结构与含义，比如p标签就代表段落，article代表正文内容等等。 语义化的好处主要有两点:

- 开发者友好:使用语义类标签增强了可读性，开发者也能够清晰地看出网页的结构，也更为便于团队的开发和维护
- 机器友好:带有语义的文字表现力丰富，更适合搜索引擎的爬虫爬取有效信息，语义类还可以支持读屏软件，根据文章可以自动生成目录 这对于简书、知乎这种富文本类的应用很重要，语义化对于其网站的内容传播有很大的帮助，但是对于功能性的web软 件重要性大打折扣，比如一个按钮、Skeleton这种组件根本没有对应的语义，也不需要什么SEO。

块级元素和行级元素?

行级元素:多个占一行，不能设置宽高 display:inline 例如：span a em i b

块级元素:自己占一行，可以设置宽高 display:block 例如：p h1-h6 ul li div

行级块元素： 多个占一行 可以设置宽高 display:inline-block

src和href的区别?

src是指向外部资源的位置，指向的内容会嵌入到文档中当前标签所在的位置，在请求src资源时会将其指向的资源 下载并应用到文档内，如js脚本，img图片和frame等元素。当浏览器解析到该元素时，会暂停其他资源的下载和处 理，知道将该资源加载、编译、执行完毕，所以一般js脚本会放在底部而不是头部。

href是指向网络资源所在位置(的超链接)，用来建立和当前元素或文档之间的连接，当浏览器识别到它他指向的 文件时，就会并行下载资源，不会停止对当前文档的处理。

doctype的作用是什么?

DOCTYPE是html5标准网页声明，且必须声明在HTML文档的第一行。来告知浏览

器的解析器用什么文档标准解析这个 文档，不同的渲染模式会影响到浏览器对于 CSS 代码甚至 JavaScript 脚本的解析

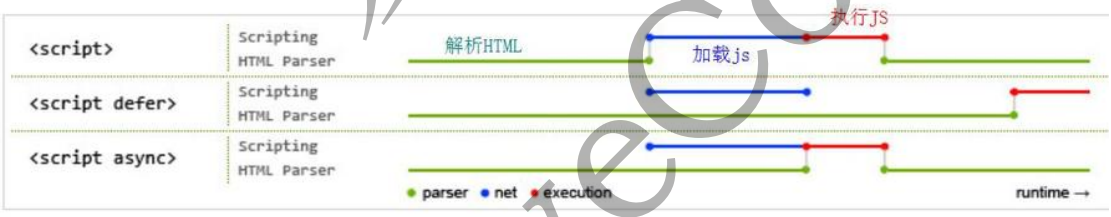
文档解析类型有:

- BackCompat:怪异模式，浏览器使用自己的怪异模式解析渲染页面。(如果没有声明DOCTYPE，默认就是这个 模式)
- CSS1Compat:标准模式，浏览器使用W3C的标准解析渲染页面。

script标签中defer和async的区别?

defer:浏览器指示脚本在文档被解析后执行，script被异步加载后并不会立刻执行，而是等待文档被解析完毕后执 行。

async:同样是异步加载脚本，区别是脚本加载完毕后立即执行，这导致async属性下的脚本是乱序的，对于script 有先后依赖关系的情况，并不适用。



蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的；绿色线代表 HTML 解析。

> 蓝色线代表网络读取，红色线代表执行时间，这俩都是针对脚本的;绿色线代表 HTML 解析

css基础

常见css选择器?

1. id选择器（#myid）
2. 类选择器（.myclassname）
3. 标签选择器（div,h1,p）

3. 标签选择器（div,h1,p）
4. 后代选择器（h1 p）
5. 相邻后代选择器（子）选择器（ul>li）
6. 兄弟选择器（li~a）
7. 相邻兄弟选择器（li+a）
8. 属性选择器（a[rel="external"]）
9. 伪类选择器（a:hover,li:nth-child）
10. 伪元素选择器（::before、::after）
11. 通配符选择器（*）

css选择器的权重值？

!important > style > id > class|伪类 >标签|伪元素

position定位的几种方式?

absolute 绝对定位

- 相对最近已定位的祖先元素
- 空间释放

relative 相对定位

- 相对于自己初始位置
- 空间不释放

fixed 固定定位

- 相对于视口
- 空间释放

static 静态 默认值

sticky 粘性定位 吸顶效果

display: none与visibility: hidden的区别

1. visibility具有继承性，给父元素设置visibility:hidden;子元素也会继承这个属性。但是如果重新给子元素设置visibility: visible,则子元素又会显示出来。这个和display: none有着质的区别

2. visibility: hidden不会影响计数器的计数，如图所示，visibility: hidden虽然让一个元素不见了，但是其计数器仍在运行。这和display: none完全不一样

谈谈你对BFC的理解?

BFC 块级格式化上下文 （Box、Formatting Context）BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。

- 触发BFC条件：
  1.html
  2.float属性不为none
  3.position为absolute或fixed
  4.display为inline-block, table-cell, table-caption, flex, inline-flex
  5.overflow不为visible

- 应用：
  1.阻止margin重叠
  2.清除内部浮动（由于在计算BFC高度时，自然也会检测浮动的子盒子高度。所以当子盒子有高度但是浮动的时候，通过激发父盒子的BFC功能，会产生清除浮动的效果。） 3.自适应两栏布局

- 规则

内部的**Box会在垂直方向，一个接一个地放置。**
Box垂直方向的距离由margin决定。属于同一个BFC的两个相邻Box的margin会发生重叠
每个元素的**margin box的左边，** 与包含块**border box的左边相接触(对于从左往右的格式化，否则相反)。即使存在浮动也是如此。**
BFC的区域不会与float box重叠。
BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。
计算**BFC的高度时，浮动元素也参与计算**

- 外边距合并问题

```
父元素 margin-top 10
子元素 margin-top 50
```

合并成两者较大的->50px
解决外边距合并：
    父元素overflow:hidden
    父元素加border
    子元素或父元素float
    子元素或父元素定位

> *清除浮动*

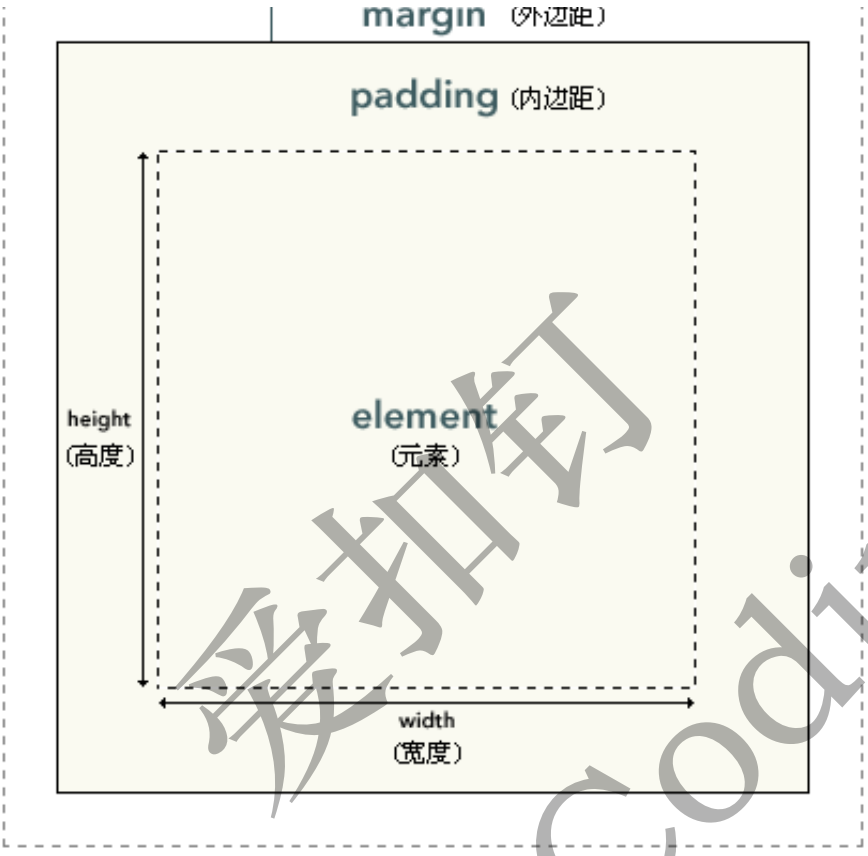
浮动怎么清除浮动?

浮动元素: 会脱离文档流 空间释放

解决浮动塌陷问题：

```
1. 父元素加高度
2. 父元素overflow:hidden  //触发BFC
3. 添加块级元素设置  style="clear:both"
4. clearfix
```

你对css盒模型的理解?



组成：margin + border + padding + element

标准盒模型 css width熟悉 = element宽度

```
怪异盒模型
    ie6 以及ie6以下的浏览器中 不写DOCTYPE
    css width属性 = border+padding+element宽度
盒模型的转化
    box-sizing:content-box(标准盒模型)|border-box(怪异盒模型)
```

CSS 中哪些属性可以继承?

```
（1）字体系列属性
font、font-family、font-weight、font-size、font-style、font-
```

```
variant、font-stretch、font-size-adjust


（2）文本系列属性
text-indent、text-align、text-shadow、line-height、word-
spacing、letter-spacing、
text-transform、direction、color

（3）表格布局属性
caption-side border-collapse empty-cells

（4）列表属性
list-style-type、list-style-image、list-style-position、list-style

（5）光标属性
cursor

（6）元素可见性
visibility
```

你对css sprites的理解，好处是什么?

雪碧图也叫CSS精灵， 是一CSS图像合成技术，开发人员往往将小图标合并在一起之后的图片称作雪碧图。 **好处**:

- 减少加载多张图片的 HTTP 请求数(一张雪碧图只需要一个请求)
- 提前加载资源

不足:

- Sprite维护成本较高，如果页面背景有少许改动，一般就要改这张合并的图片
- 加载速度优势在http2开启后荡然无存，HTTP2多路复用，多张图片也可以重复利用一个连接通道搞定

CSS3

css3有哪些新新特性?

css3新增选择器
border-raduis
background-size:cover|contain

弹性盒模型
transform :rotate translate scale skew
transition
animation
@media

transition 、animation定义动画怎么写？每个属性都代表什么?

transition:width 2s linear 2s
animation:name 2s 运动方式 延迟时间 执行次数 animation-fill-mode（none|forward） @keyframs name{

}

伪类和伪元素的区别?

伪类(pseudo-class) 是一个以冒号(:)作为前缀，被添加到一个选择器末尾的关键字，当你希望样式在特定状态下才被 呈现到指定的元素时，你可以往元素的选择器后面加上对应的伪类。
伪元素用于创建一些不在文档树中的元素，并为其添加样式。比如说，我们可以通过::before来在一个元素前增加一些 文本，并为这些文本添加样式。虽然用户可以看到这些文本，但是这些文本实际上不在文档树中。

区别： 其实上文已经表达清楚两者区别了，伪类是通过在元素选择器上加入伪类改变元素状态，而伪元素通过对元素的操作进 行对元素的改变。

flex有哪些属性?

父元素

```
display:flex;
方向: flex-direction: row | row-reverse | column | column-reverse
垂直对齐:  align-items: flex-start | flex-end | center | baseline |
```

```
垂直对齐：  align-items: flex-start | flex-end | center | baseline |
stretch
水平对齐：  align-content: flex-start | flex-end | center | space-
between | space-around | stretch
控制flex容器是单行或者多行：  flex-wrap: nowrap | wrap | wrap-reverse
```

子元素 flex:1

- <' flex-grow '>用来指定扩展比率
- <' flex-shrink '>：用来指定收缩比率
- <' flex-basis '>：用来指定伸缩基准值

order子元素出现的顺序 用整数值来定义排列顺序，数值小的排在前面。可以为负值。

布局问题

> 左侧固定右侧自适应
> 水平垂直都居中

浏览器重绘与重排的区别？？

重绘就是在一个元素的外观被改变，但没有改变布局（宽高）的情况下发生， 如改变visibility、outline、背景色等等。
重排就是DOM的变化影响到了元素的几何属性（宽和高）， 浏览器会重新计算元素的几何属性,如：改变窗口大小、改变文字大小、 内容的改变、浏览器窗口变化，style属性的改变等等。

单单改变元素的外观，肯定不会引起网页重新生成布局，但当浏览器完成重排之后，将会重新绘制受到此次重排影响的
部分重排和重绘代价是高昂的，它们会破坏用户体验，并且让UI展示非常迟缓，而相比之下重排的性能影响更大，在两者无法避免的情况下，一般我们宁可选择代价更小的重绘。

『重绘』不一定会出现『重排』，『重排』必然会出现『重绘』

如何触发重排和重绘?

任何改变用来构建渲染树的信息都会导致一次重排或重绘:

添加、删除、更新DOM节点

通过display: none隐藏一个DOM节点-触发重排和重绘
通过visibility: hidden隐藏一个DOM节点-只触发重绘，因为没有几何变化 移动或者给页面中的DOM节点添加动画
添加一个样式表，调整样式属性 用户行为，例如调整窗口大小，改变字号，或者滚动。

如何避免重绘或者重排?

- 集中改变样式 我们往往通过改变class的方式来集中改变样式
- 使用DocumentFragment 我们可以通过createDocumentFragment创建一个游离于DOM树之外的节点，然后在此节点上批量操作，最后插入DOM树中，因此只触发一次重排

浏览器渲染（扩展题）

https://github.com/CavsZhouyou/Front-End-Interview-Notebook/blob/master/Html/Html.md#19-async-和-defer-的作用是什么有什么区别浏览器解析过程

移动端

如何实现响应式布局?

百分比、rem、弹性盒模型、media媒体查询

rem和em、px区别

1. px：绝对单位，页面按精确像素展示。
2. em：相对单位，基准点为父节点字体的大小。
3. rem：相对单位，可理解为”root em”,相对根节点html的字体大小来计算。

（浏览器默认字体是16px）

rem的原理

rem 是指相对于根元素的字体大小的单位,rem布局按照设计稿的宽去设置一个合适

的rem ,配合js查询屏幕大小来改变html的font-size，从而达到适配各种屏幕的效果

CSS相关编程题

1. CSS实现水平垂直居中（多种方式）？
2. 实现一个三角形?
3. 画一条0.5px的线（https://juejin.cn/post/6844903582370643975）
4. 左侧固定右侧自适应（多种方式）

javascritp基础

js有哪几种数据类型？通过什么判断

基本数据类型 (存储在栈中)
number string boolean null undefined symbol
引用数据类型 （存在堆中）
object array typeof "123"

如何判断是否是还是对象?

```
Array.isArray(arr)
console.log(obj instanceof Array) // 判读arr是否是Array的实例化对象
console.log(obj.constructor == Object)  //通过构造函数
Object.prototype.toString.call(arr)
Array.isArray()
```

实现深克隆一个对象

```
var obj3 = {name:'zs',age:18,father:{name:'zzz'}};
function cloneArray(o){
    var result = {};
```

```
    var result = {};
    for(key in o){
        if(typeof o[key] == "object"){ //引用数据类型
            result[key] = cloneArray(o[key]);
        }else{
            result[key] = o[key];
        }
    }
    return result;
}
var obj4 =  cloneArray(obj3);
obj4.father.name = 'zxx';
console.log(obj3,obj4);
```

函数引用传值 (基本数据类型传递值，引用数据类型传地址)

```
var a = 10
function fn(x){
    x = 20;
    console.log(x);  //20
}
fn(a)
console.log(a);  //10


var arr = [1,2,3];
function fn2(x){
    x[0] = 5
    console.log(x)  //[5,2,3]
}
fn2(arr)
console.log(arr) //[5,2,3]
```

js类

```
 // 类 具有相同方法和属性的集合
// 属性写在构造函数中，方法写在原型对象下
function Person(name,age){
```

```
function Person(name,age){
    this.name = name;
    this.age = age;
}
// 原型对象
Person.prototype.eat = function(){
    console.log('吃')
}
var p1 = new Person('zs',18);
console.log(p1.name)
p1.eat();
```

原型对象

构造函数有个prototype属性，他指向一个原型对象，(原型对象有个constructor属性,他指向构造函数)这个原型对象下的属性和方法可以被(该构造函数的)实例对象所共享，

js实现继承

假设有一个ClassA和ClassB，ClassB想继承ClassA 首先要在A的构造函数里定义属性，在ClassA的原型里定义方法：

```
function ClassA() {
    this.color = sColor;  //属性写在构造函数中
}
//方法 写在原型对象中
ClassA.prototype.sayColor = function () {
    alert(this.color);
};
```

然后在ClassB的构造函数中使用ClassA.call(this)来继承ClassA中的属性：

```
function ClassB() {
    ClassA.call(this);
}
```

再用ClassB.prototype等于ClassA的一个实例对象来继承ClassA中的方法：

```
```

```
ClassB.prototype = new ClassA();   //缺点就是这个构造函数有问题，需要指回去

ClassB.prototype.constructor = ClassB;
```

原型链

当从一个对象那里调取属性或方法时，如果该对象自身不存在这样的属性或方法，就会去自己关联的prototype对象那里寻找，如果prototype没有，就会去prototype关联的前辈prototype那里寻找，如果再没有则继续查找Prototype.Prototype引用的对象，依次类推，直到Prototype.....Prototype为undefined（Object的Prototype就是undefined）从而形成了所谓的“原型链”。

__proto__

每个对象都有 __proto__ 属性，此属性指向该对象的构造函数的原型。

this指向有哪些方式?


```
1. 定时器
setInterval(()=>{
    console.log(this) //window
```

```
    console.log(this) //window
})
2. 事件
var oDiv = document.getElementById('div1');

oDiv.onclick = function(){
    console.log(this);  //oDiv
}
3. 对象方法中
var obj = {
    name:'zs',
    eat:function(){
        console.log(this)  //obj
    }
}
obj.eat();
4. 构造函数中
function Person(name){
    this.name = name
}
var p1 = new Person('zs'); //this实例化对象 =>p1
var p2 = new Person('lisi')//this实例化对象 =>p2
```

怎么改变this指向?

```
function fn(x,y){
    console.log(this,x,y)
}
fn()//window
// this指向 参数1,参数2
fn.call(obj,1,2);
// this指向 []参数
fn.apply(obj,[1,2])
// // this指向 参数1,参数2
fn.bind(obj,1,2)();
```

> call apply 和bind区别：

1. call apply都会调用fn方法，而bind只会修改this指向，不会调用fn方法
2. call apply的传参方式不一样

DOM的事件模型是什么?

DOM之事件模型分脚本模型、内联模型(同类一个、后者覆盖)、动态绑定(同类多

个)

```
<body>
<!--行内绑定:脚本模型-->
<button onclick="javascrpt:alert('Hello')">Hello1</button> <!--内联
模型-->
<button onclick="showHello()">Hello2</button> <!--动态绑定-->
<button id="btn3">Hello3</button>
</body>
<script>
/*DOM0:同一个元素，同类事件只能添加一个，如果添加多个，
* 后面添加的会覆盖之前添加的*/
function showHello() {
    alert("Hello");
}
var btn3 = document.getElementById("btn3");
btn3.onclick = function () {
    alert("Hello");
}
/*DOM2:可以给同一个元素添加多个同类事件*/

btn3.addEventListener("click",function () { alert("hello1");
});
btn3.addEventListener("click",function () { alert("hello2");
})
if (btn3.attachEvent){
    /*IE*/
    btn3.attachEvent("onclick",function () {
        alert("IE Hello1");
    })
}else {
    /*W3C*/
    btn3.addEventListener("click",function () {
        alert("W3C Hello");
    })
}
</script>
```


事件冒泡

事件冒泡(event bubbling)，即事件开始时由最具体的元素(文档中嵌套层次最深的那个节点)接收，然后逐级向上传播到 较为不具体的节点

如果单击了页面中的div元素，那么这个click事件沿DOM树向上传播，在每一级节点上都会发生，按照如下顺序传播:

```
<div id="div1">
    <div id="div2">
        <div id="div3"></div>
    </div>
</div>

<script>
    var oDiv1 = document.getElementById('div1');
    var oDiv2 = document.getElementById('div2');
    var oDiv3 = document.getElementById('div3');

    oDiv1.addEventListener('click',function(){
        console.log('div1 捕获')
    },true)
    oDiv2.addEventListener('click',function(){
        console.log('div2 捕获')
    },true)
    oDiv3.addEventListener('click',function(){
        console.log('div3 捕获')
    },true)

    oDiv1.addEventListener('click',function(){
        console.log('div1 冒泡')
    },false)
    oDiv2.addEventListener('click',function(){
        console.log('div2 冒泡')
    },false)
    oDiv3.addEventListener('click',function(){
        console.log('div3 冒泡')
    },false)
</script>
```

> 执行顺序： div1捕获 div2捕获 div3捕获 div3冒泡 div2冒泡 div1冒泡

事件流

事件流又称为事件传播，DOM2级事件规定的事件流包括三个阶段:事件捕获阶段(capture phase)、处于目标阶段 (target phase)和事件冒泡阶段(bubbling phase)。

事件委托

事件委托就是利用事件冒泡，通过给父元素绑定事件，当点击子元素的时候通过事件冒泡会触发父元素的点击事件，在父元素的点击事件中，通过判断事件源执行时间回调函数。

```
<button id="btn">click</button>
<ul>
    <li>111</li>
    <li>222</li>
    <li>333</li>
</ul>

var obtn = document.getElementById('btn');
var oUl = document.getElementsByTagName('ul')[0];
oUl.onclick = function(e){
    e = e || window.event;
    // 判断当你点击的是li执行输出 ->获取事件源
    if(e.target.tagName == "LI"){
        console.log(e.target.innerHTML)
    }
}
obtn.onclick = function(){
    var oLi = document.createElement('li');
    oLi.innerHTML = Math.random();
    oUl.appendChild(oLi)
}
```

解决问题：

1. 后生成元素的事件绑定问题
2. 节省内存占用，减少事件注册

局限性:

1. focus、blur 之类的事件本身没有事件冒泡机制，所以无法委托

事件中常用的属性方法

```
e.target    //事件源
e.keyCode   //键盘码
e.clientX   //鼠标距离文档水平坐标
```

```
e.clientX   //鼠标距离文档水平坐标
e.clientY   //鼠标距离文档垂直坐标

// 阻止事件冒泡
e.stopPropagation()//阻止事件冒泡
e.cancelBubble = true  //ie

// 阻止默认行为
e.preventDefault(); //标准
e.returnValue = false; //ie
```

事件循环(event loop)

首先先了解一下任务队列

js是单线程的

任务队列：

- 同步任务： 同步任务指的是在主线程上排队执行的任务，只有前一个任务执行完毕，才能执行后一个任务。
- 异步任务： 异步指的是不进入主线程，而进入“任务队列”的任务，只有“任务队列”通知主线程，某个异步任务可以执行了，该任务才会进入主线程执行。

事件循环：

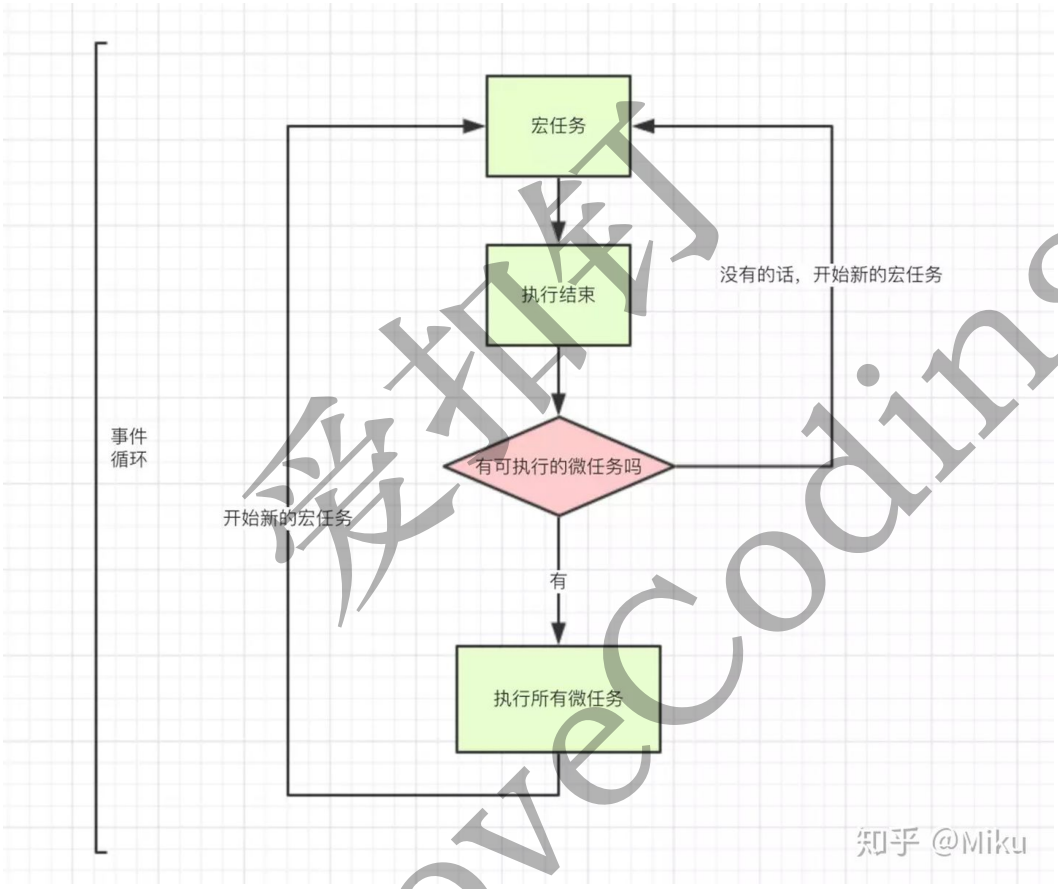
1. 所有同步任务都在主线程上执行，形成一个执行栈(execution context stack)
2. 主线程之外，还存在一个“任务队列”，只要异步任务有了运行结果，就在“任务队列”之中放置一个事件
3. 一旦“执行栈”中的所有同步任务执行完毕，系统就会读取“任务队列”，看看里面有哪些事件。那些对应的异步任务，于是结束等待，进入执行栈，开始执行
4. 主线程不断重复第3步

宏任务和微任务：

- 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)。主要场景有：主代码块script、setTimeout、setInterval等

- 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。主要场景有：Promise的then回调、process.nextTick等。

> 执行顺序优先级：SYNC => MICRO => MACRO




```
console.log('1');

setTimeout(function() { //宏任务
```

```
setTimeout(function() { //宏任务
    console.log('2');
    process.nextTick(function() { //2-微任务
        console.log('3');
    });
    new Promise(function(resolve) {
        console.log('4');
        resolve();
    }).then(function() {  //2-微任务
        console.log('5')
    });
})
process.nextTick(function() { //微任务
    console.log('6');
})
// 1 6 2 4 3 5
```

js的作用域

- 函数作用域
- 全局作用域

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。

闭包

什么是闭包：闭包的一大特性就是内部函数总是可以访问其所在的外部函数中声明的参数和变量

- 函数嵌套函数
- 内部函数引用外部函数的局部变量

闭包的作用：

- 实现函数外部访问私有变量
- 避免全局变量的污染 实现封装
- 希望一个变量长期驻扎在内存中

闭包的缺点：容易引发内存泄漏

```
function f1(a){
```

```
function fn1(val){
    var a = 20;
    return function(){
        a++;
        console.log(a,val);
    }
}
var f = fn1();
f();
f();
f();
```

```
<ul>
    <li>1111</li>
    <li>222</li>
    <li>3333</li>
</ul>

var aLi = document.getElementsByTagName('li');
for(var i=0; i<aLi.length; i++){
    (function(i){
        // i是外部函数中的参数
        aLi[i].onclick = function(){
            console.log(i);
        }
    })(i);
}
```

什么是函数节流

一个函数执行一次后，只有大于设定的执行周期后才执行第二次，有个需要频繁触发的函数出于优化性能角度，在规定时间内，只让函数触发第一次生效，后面不生效。

```
function throttle(fn,delay){
    var startTime = 0;
    return function(){
```

```
        return function(){
            var nowTime = Date.now();
            if(nowTime - startTime > delay){
                fn.call(this);
                startTime = nowTime;
            }
        }
    }
    document.onmousemove = throttle(function(){
        console.log(Date.now())
        console.log(this);
    },1000);
```

什么是函数防抖

函数去抖是一个需要频繁触发的函数，在规定时间内只让最后一次生效，前面的不生效比如 搜索

```
<button id="btn">click</button>

function debounce(fn,delay) {
    var timer = null;
    return function(){
        clearTimeout(timer);

        timer = setTimeout(() => {
            fn.apply(this);
        },delay);
    }
}

var oBtn = document.getElementById('btn');
oBtn.onclick = debounce(function(){
    console.log(Date.now());
    console.log(this);
},300);
```

ajax原理

AJAX 是一种在无需重新加载整个网页的情况下，能够更新部分网页的技术。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。

0. 创建 XMLHttpRequest 对象

```
var xmlhttp;
if (window.XMLHttpRequest)
  {// code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
  }
else
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
```

2. AJAX - 向服务器发送请求

```
xmlhttp.open("GET","test1.txt",true);
xmlhttp.send();
```

```
// post  请求方式  url  是否异步
xmlhttp.open("POST","ajax_test.asp",true);
xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");
xmlhttp.send("fname=Bill&lname=Gates");
```

3. 服务器响应

```
xmlhttp.onreadystatechange=function()
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {

document.getElementById("myDiv").innerHTML=xmlhttp.responseText;
    }
  }
```

onreadystatechange 存储函数（或函数名），每当 readyState 属性改变时，就会调用该函数。

readyState 存有 XMLHttpRequest 的状态。从 0 到 4 发生变化。

```
0: 请求未初始化
1: 服务器连接已建立
2: 请求已接收
```

```
2: 请求已接收
3: 请求处理中
4: 请求已完成，且响应已就绪
```

status

```
200: "OK"

404: 未找到页面
```

谈一谈HTTP状态码

```
2XX 成功
200 OK，表示从客户端发来的请求在服务器端被正确处理
201 Created 请求已经被实现，而且有一个新的资源已经依据请求的需要而建立
202 Accepted 请求已接受，但是还没执行，不保证完成请求
204 No content，表示请求成功，但响应报文不含实体的主体部分
206 Partial Content，进行范围请求
3XX 重定向
301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
302 found，临时性重定向，表示资源临时被分配了新的 URL
303 see other，表示资源存在着另一个 URL，应使用 GET 方法丁香获取资源
304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
307 temporary redirect，临时重定向，和302含义相同
4XX 客户端错误
400 bad request，请求报文存在语法错误
401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
403 forbidden，表示对请求资源的访问被服务器拒绝
404 not found，表示在服务器上没有找到请求的资源
408 Request timeout，客户端请求超时
409 Confict，请求的资源可能引起冲突
5XX 服务器错误
500 internal sever error，表示服务器端在执行请求时发生了错误
501 Not Implemented 请求超出服务器能力范围，例如服务器不支持当前请求所需要的
某个功能，或者请求是服务 器不支持的某个方法
503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求
505 http version not supported 服务器不支持，或者拒绝支持在请求中使用的
HTTP 版本
```

什么是跨域?

何谓同源: URL由协议、域名、端口和路径组成，如果两个URL的协议、域名和端口

相同，则表示他们同源。有一个不同就是**跨域**

同源策略:
同源策略限制了从同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。这是一个用于隔离潜在恶意文件的 重要安全机制。

解决跨域的方式?

链接: https://pan.baidu.com/s/1mHTuk3FDpJYPpHavSaLBfQ 密码: acw1

编程题

1. js 中的深浅拷贝实现?
2. 手写 call、apply 及 bind 函数

HTML5

H5有哪些新特性?

1. 新的语义化标签，比如header、footer、nav、article等section、video 等
2. 新增表单控件: input type="email/number/range..."
3. 新的选择器：querySelector和querySelectorAll
4. JSON的方法： JSON.parse() 字符串转化成对象 JSON.stringify()json对象转成字符串
5. 历史管理：history.pushState()和window.onpopstate事件
6. 本地存储：localStorage和sessionStorage（可能会问和cookie）
7. Canvas画布：

```
var cvs = document.getElementById('canvas1');
var cxt = cvs.getContext('2d');
```

sessionStorage 和 localStorage cookie区别？

共同点：都是保存在浏览器端，且同源的。 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。而sessionStorage和localStorage不会自动把数据发给服务器，仅在本地保存。 cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下。

2. 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。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。
3. 生命周期不同，sessionstorage:关闭浏览器就删除， localstorage:一直有，除非手动删除，Cookie：可以设置过期时间 sessionStorage：仅在当前浏览器窗口关闭前有效，自然也就不可能持久保持；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。
4. 作用域不同， sessionStorage不在不同的浏览器窗口中共享，即使是同一个页面；localStorage在所有同源窗口中都是共享的；

ES6

你了解ES6哪些新特性?

Let（定义变量） const（定义常量,特性和let一样）
arrow(箭头函数) 、class 类 、map set 、数组、对象解构、promise、模板字符串、 数组对象字符串新加了一些函数

> set可以用来去重 类似于数组 map类似于对象 属性可以是任意类型

let和var const区别?

let没有变量提升
let不能重复命名
Let只在块级作用域有效（作用域外面不好使，块级作用域就是一个{}就是一个作用域 ， for{}也是一个块作用域）
let有临时失效区，暂时性死区（在let的作用域里面，只要和用let定义的变量同名的，都不起作用，哪怕是全局的，哪是在这个变量的前面定义 ，前面用的也不行）

什么是变量提升?

通过var 定义变量、会在当前作用域的顶端、提升只是将变量声明提升到变量所在

的变量范围的顶端

箭头函数

Arrow(箭头函数)的this指向问题?

- 普通的this: this总是代表它的直接调用者,没找到直接调用者,则this指的是window
  (匿名函数,定时器中的函数,由于没有默认的宿主对象,所以默认this指向window)
  父作用域的this
- 箭头函数的this 默认指向在定义它时,它所处的对象,而不是执行时的对象没有

箭头函数和普通函数区别?

- 箭头函数的this 默认指向在定义它时，它所处的对象 父作用域
- 箭头函数 不能new
- 箭头函数不能用arguments,可以用rest(...)代替

promise

就可以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。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。

Promise规范如下：

- 一个promise可能有三种状态：pending（进行中）、fulfilled（已成功）和rejected
- 一个promise的状态只可能从“等待”转到“成功”态或者“失败”态，不能逆向转

- 一个promise的状态只可能从“等待”转到“成功”态或者“失败”态，不能逆向转换，同时“成功”态和“失败”态不能相互转换
- promise必须实现then方法（可以说，then就是promise的核心），而且then必须返回一个promise，同一个promise的then可以调用多次，并且回调的执行顺序跟它们被定义时的顺序一致
- then方法接受两个参数，第一个参数是成功时的回调，在promise由“等待”态转换到“完成”态时调用，另一个是失败时的回调，在promise由“等待”态转换到“拒绝”态时调用。同时，then可以接受另一个promise传入，也接受一个“类then”的对象或方法
- 解决 函数的回调嵌套 问题

Promise 也有一些缺点

- 首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。
- 其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。
- 第三，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）。

promise下面有这些方法

- Promise.all([p1,p2,p3]) 都执行完 再执行，
- Promise.race() 最快的那个执行完了就执行

async await

async函数(源自ES7)

概念： 真正意义上去解决异步回调的问题，同步流程表达异步操作
本质： Generator的语法糖
语法：

```
async function foo(){
    await 异步操作;
    await 异步操作;
}
```

特点：

1. 不需要像Generator去调用next方法，遇到await等待，当前的异步操作完成就

往下执行

2. 返回的总是Promise对象，可以用then方法进行下一步操作
3. async取代Generator函数的星号*，await取代Generator的yield
4. 语意上更为明确，使用简单

```
async function first(){
    return new Promise(function(resolve){
        setTimeout(() => {
            resolve()
        }, 2000);
    })
}
async function asyncPrint(){
    console.log('开始');
    await first();
    console.log('第一个请求完成');
}
asyncPrint();
```

Vue

vue.js的特点

简洁：页面由HTML模板+Json数据+Vue实例组成
数据驱动：自动计算属性和追踪依赖的模板表达式
组件化：用可复用、解耦的组件来构造页面
轻量：代码量小，不依赖其他库
快速：精确有效批量DOM更新
模板友好：可通过npm，bower等多种方式安装，很容易融入

你对MVVM的理解?

MVVM 模式，顾名思义即 Model-View-ViewModel 模式。
MVVM:将“数据模型数据双向绑定”的思想作为核心，因此在View和Model之间没有联系、通过ViewModel进行交互、而且Model和ViewModel之间的交互是双向的、

因此视图的数据的变化会同时修改数据源，而数据源数据的变化也会立即反应到View上。

MVC是比较直观的架构模式，用户操作 ->View（负责接收用户的输入操作）->Controller（业务逻辑处理）->Model（数据持久化）->View（将结果反馈给View）。

Vue.js常用指令

- v-if指令 条件判断指令
- v-show指令 条件渲染指令，与v-if不同的是，无论v-show的值为true或false，元素都会存在于HTML代码中；而只有当v-if的值为true，元素才会存在于HTML代码中。v-show指令只是设置了元素CSS的style值
- v-else指令 可配合v-if使用，v-else指令必须紧邻v-if，否则该命令无法正常工作。v-else绑定的元素能否渲染在HTML中，取决于前面使用的是v-if。若前面使用的是v-if，且v-if值为true，则v-else元素不会渲染；
- v-for指令 循环指令，基于一个数组渲染一个列表，与JavaScript遍历类似
- v-bind指令 给DOM绑定元素属性 v-bind指令可以缩写为:冒号
- v-on指令 用于监听DOM事件 v-on指令可以缩写为@符号
- v-html指令
- v-text指令
- v-model指令

Vue生命周期 和 钩子函数

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、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。

Lifecycle hooks

生命周期钩子应该算 vue 这次升级中 broken changes 最多的一部分了，对照 1.0 的文档和 release note，作了下面这张表

vue 1.0+	vue 2.0	Description
init	beforeCreate	组件实例刚被创建，组件属性计算之前，如 data 属性等
created	created	组件实例创建完成，属性已绑定，但 DOM 还未生成，`$el` 属性还不存在
beforeCompile	beforeMount	模板编译/挂载之前
compiled	mounted	模板编译/挂载之后
ready	mounted	模板编译/挂载之后（不保证组件已在 document 中）
-	beforeUpdate	组件更新之前
-	updated	组件更新之后
-	activated	for `keep-alive`，组件被激活时调用
-	deactivated	for `keep-alive`，组件被移除时调用
attached	-	不用了还说啥哪...
detached	-	那就不说了吧...
beforeDestory	beforeDestory	组件销毁前调用
destoryed	destoryed	组件销毁后调用

异步请求适合在哪个生命周期调用?

官方实例的异步请求是在mounted生命周期中调用的，而实际上也可以在created生命周期中调用。

Vue组件如何通信?

- props/$emit+v-on: 通过props将数据自上而下传递，而通过$emit和v-on来向上传递信息。
- vuex: 是全局数据管理库，可以通过vuex管理全局的数据流
- EventBus: 通过EventBus进行信息的发布与订阅

> 详细可以参考这篇文章vue中8种组件通信方式,不过太偏门的通信方式根本不会用到,单做知识点了解即可

computed和watch有什么区别?

computed:

1. 是计算属性,也就是计算值,它更多用于计算值的场景
2. 具有缓存性,computed的值在getter执行后是会缓存的，只有在它依赖的属性值改变之后，下一次获取computed的值时才会重新调用对应的getter来计算

改变之后，下一次获取computed的值时才会重新调用对应的getter来计算
3. computed 适用于计算比较消耗性能的计算场景

watch:

1. 更多的是「观察」的作用,类似于某些数据的监听回调,用于观察 props 行回调进行后续操作
2. 无缓存性，页面重新渲染时值不变化也会执行

小结:

1. 当我们要进行数值计算,而且依赖于其他数据，那么把这个数据设计为computed
2. 如果你需要在某个数据变化时做一些事情，使用watch来观察这个数据变化

Vue中的key到底有什么用?

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.

- 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).

Vue 组件 data 为什么必须是函数?

组件复用时所有组件实例都会共享 data，如果 data 是对象的话，就会造成一个组件修改 data 以后会影响到其他所有组件，所以需要将 data 写成函数，每次用到就调用一次函数获得新的数据

组件中 data 什么时候可以使用对象?

当我们使用 new Vue() 的方式的时候，无论我们将 data 设置为对象还是函数都是可以的，因为 new Vue() 的方式是生成一个根组件，该组件不会复用，也就不存在共享 data 的情况了

扩展

Vue是如何实现双向绑定的?

利用 Object.defineProperty 劫持对象的访问器,在属性值发生变化时我们可以获取变化,然后根据变化进行后续响应,在 vue3.0中通过Proxy代理对象进行类似的操作。

```
// 这是将要被劫持的对象
const data = {
    name: '',
};
function say(name) {
    if (name === '古天乐') {
        console.log('给大家推荐一款超好玩的游戏');
    } else if (name === '渣渣辉') {
        console.log('戏我演过很多,可游戏我只玩贪玩懒月');
    } else {
        console.log('来做我的兄弟'); }
    }
// 遍历对象,对其属性值进行劫持 Object.keys(data).forEach(function(key) {
Object.defineProperty(data, key, {
    enumerable: true,
    configurable: true,
    get: function() {
        console.log('get');
    },
    set: function(newVal) {
        // 当属性值发生变化时我们可以进行额外操作 console.log(`大家好,我系
${newVal}`); say(newVal);
    },
    });
});
data.name = '渣渣辉';
//大家好,我系渣渣辉
//戏我演过很多,可游戏我只玩贪玩懒月
```

> 详细实现见Proxy比defineproperty优劣对比?

虚拟dom 、diff算法

> 详见 https://github.com/CavsZhouyou/Front-End-Interview-

> Notebook/blob/master/JavaScript/JavaScript.md#109-vue-双向数据绑定原理

其他方向扩展

前端安全

1. 什么是 XSS 攻击？如何防范 XSS 攻击？
2. 什么是 CSRF 攻击？如何防范 CSRF 攻击？

> 详细见 https://github.com/CavsZhouyou/Front-End-Interview-Notebook/blob/master/JavaScript/JavaScript.md#102-什么是-xss-攻击如何防范-xss-攻击

计算机网络 （见计算机网络相关面试题.pdf）

> 链接: https://pan.baidu.com/s/14ZSn32Ri1QZBDmAXMebkgQ 密码: fg4f

常用工具知识总结

GIT

1. git 与 svn 的区别在哪里？

```
git 和 svn 最大的区别在于 git 是分布式的，而 svn 是集中式的。因此我们不能再离线的情况下使用 svn。如果服务器
出现问题，我们就没有办法使用 svn 来提交我们的代码。
svn 中的分支是整个版本库的复制的一份完整目录，而 git 的分支是指针指向某次提交，因此 git 的分支创建更加开销更小
并且分支上的变化不会影响到其他人。svn 的分支变化会影响到所有的人。
svn 的指令相对于 git 来说要简单一些，比 git 更容易上手。
```

详细资料可以参考： 《常见工作流比较》 《对比 Git 与 SVN，这篇讲的很易懂》 《GIT 与 SVN 世纪大战》 《Git 学习小记之分支原理》

2. 经常使用的 git 命令？

```
git init                          // 新建 git 代码库
git add                           // 添加指定文件到暂存区

git rm                            // 删除工作区文件，并且将这次删除放入暂存区
git commit -m [message]           // 提交暂存区到仓库区
git branch                        // 列出所有分支
git checkout -b [branch]          // 新建一个分支，并切换到该分支
git status                        // 显示有变更的文件
```

详细资料可以参考： 《常用 Git 命令清单》

3. git pull 和 git fetch 的区别

```
git fetch 只是将远程仓库的变化下载下来，并没有和本地分支合并。
git pull 会将远程仓库的变化下载下来，并和当前分支合并。
```

《详解 git pull 和 git fetch 的区别》

4. git rebase 和 git merge 的区别

```
git merge 和 git rebase 都是用于分支合并，关键在 commit 记录的处理上不同。
git merge 会新建一个新的 commit 对象，然后两个分支以前的 commit 记录都指向这个新 commit 记录。这种方法会
保留之前每个分支的 commit 历史。
git rebase 会先找到两个分支的第一个共同的 commit 祖先记录，然后将提取当前分支这之后的所有 commit 记录，然后
将这个 commit 记录添加到目标分支的最新提交后面。经过这个合并后，两个分支合并后的 commit 记录就变为了线性的记
录了。
```

《git rebase 和 git merge 的区别》 《git merge 与 git rebase 的区别》

常用算法和数据结构总结

排序

冒泡排序

冒泡排序的基本思想是，对相邻的元素进行两两比较，顺序相反则进行交换，这样，每一趟会将最小或最大的元素“浮”到顶端， 最终达到完全有序。

代码实现：

```
function bubbleSort(arr) {
    if (!Array.isArray(arr) || arr.length <= 1) return;
    let lastIndex = arr.length - 1;
    while (lastIndex > 0) { // 当最后一个交换的元素为第一个时，说明后面全部排序完毕
        let flag = true, k = lastIndex;
        for (let j = 0; j < k; j++) {
            if (arr[j] > arr[j + 1]) {
                flag = false;
                lastIndex = j; // 设置最后一次交换元素的位置
                [arr[j], arr[j+1]] = [arr[j+1], arr[j]];
            }
        }
        if (flag) break;
    }
}
```

冒泡排序有两种优化方式。

一种是外层循环的优化，我们可以记录当前循环中是否发生了交换，如果没有发生交换，则说明该序列已经为有序序列了。 因此我们不需要再执行之后的外层循环，此时可以直接结束。

一种是内层循环的优化，我们可以记录当前循环中最后一次元素交换的位置，该位置以后的序列都是已排好的序列，因此下 一轮循环中无需再去比较。

优化后的冒泡排序，当排序序列为已排序序列时，为最好的时间复杂度为 O(n)。

冒泡排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 O(1) ，是稳定排序。

详细资料可以参考： 《图解排序算法(一)》 《常见排序算法 - 鸡尾酒排序 》 《前端笔试&面试爬坑系列---算法》 《前端面试之道》

选择排序

选择排序的基本思想为每一趟从待排序的数据元素中选择最小（或最大）的一个元素作为首元素，直到所有元素排完为止。

在算法实现时，每一趟确定最小元素的时候会通过不断地比较交换来使得首位置为当前最小，交换是个比较耗时的操作。其实 我们很容易发现，在还未完全确定当前最小元素之前，这些交换都是无意义的。我们可以通过设置一个变量 min，每一次比较 仅存储较小元素的数组下标，当轮循环结束之后，那这个变量存储的就是当前最小元素的下标，此时再执行交换操作即可。

代码实现：

```
function selectSort(array) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;
  for (let i = 0; i < length - 1; i++) {
    let minIndex = i; // 设置当前循环最小元素索引
    for (let j = i + 1; j < length; j++) {
      // 如果当前元素比最小元素索引，则更新最小元素索引
      if (array[minIndex] > array[j]) {
        minIndex = j;
      }
    }
    // 交换最小元素到当前位置
    // [array[i], array[minIndex]] = [array[minIndex], array[i]];
    swap(array, i, minIndex);
  }
  return array;
}
// 交换数组中两个元素的位置
function swap(array, left, right) {
  var temp = array[left];
  array[left] = array[right];
  array[right] = temp;
}
```

选择排序不管初始序列是否有序，时间复杂度都为 $O(n^2)$。

选择排序的平均时间复杂度为 $O(n^2)$ ， 最坏时间复杂度为 $O(n^2)$ ， 空间复杂度为 $O(1)$ ， 不是稳定排序。

详细资料可以参考： 《图解排序算法(一)》

插入排序

直接插入排序基本思想是每一步将一个待排序的记录，插入到前面已经排好序的有序序列中去，直到插完所有元素为止。

插入排序核心--扑克牌思想： 就想着自己在打扑克牌，接起来一张，放哪里无所谓，再接起来一张，比第一张小，放左边，继续接，可能是中间数，就插在中间....依次

代码实现：

```
function insertSort(array) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;
  // 循环从 1 开始，0 位置为默认的已排序的序列
  for (let i = 1; i < length; i++) {
    let temp = array[i]; // 保存当前需要排序的元素
    let j = i;
    // 在当前已排序序列中比较，如果比需要排序的元素大，就依次往后移动位置
    while (j -1 >= 0 && array[j - 1] > temp) {
      array[j] = array[j - 1];
      j--;
    }
    // 将找到的位置插入元素
    array[j] = temp;
  }
  return array;
}
```

当排序序列为已排序序列时，为最好的时间复杂度 O(n)。

插入排序的平均时间复杂度为 $O(n^2)$ ，最坏时间复杂度为 $O(n^2)$ ，空间复杂度为 O(1) ，是稳定排序。

详细资料可以参考： 《图解排序算法(一)》

希尔排序

希尔排序的基本思想是把数组按下标的一定增量分组，对每组使用直接插入排序算

法排序；随着增量逐渐减少，每组包含的元 素越来越多，当增量减至1时，整个数组恰被分成一组，算法便终止。

代码实现：

```
function hillSort(array) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;
  // 第一层确定增量的大小，每次增量的大小减半
  for (let gap = parseInt(length >> 1); gap >= 1; gap =
parseInt(gap >> 1)) {
    // 对每个分组使用插入排序，相当于将插入排序的1换成了 n
    for (let i = gap; i < length; i++) {
      let temp = array[i];
      let j = i;
      while (j - gap >= 0 && array[j - gap] > temp) {
        array[j] = array[j - gap];
        j -= gap;
      }
      array[j] = temp;
    }
  }
  return array;
}
```

希尔排序是利用了插入排序对于已排序序列排序效果最好的特点，在一开始序列为无序序列时，将序列分为多个小的分组进行 基数排序，由于排序基数小，每次基数排序的效果较好，然后在逐步增大增量，将分组的大小增大，由于每一次都是基于上一 次排序后的结果，所以每一次都可以看做是一个基本排序的序列，所以能够最大化插入排序的优点。

简单来说就是，由于开始时每组只有很少整数，所以排序很快。之后每组含有的整数越来越多，但是由于这些数也越来越有序， 所以排序速度也很快。

希尔排序的时间复杂度根据选择的增量序列不同而不同，但总的来说时间复杂度是小于 O(n^2) 的。

插入排序是一个稳定排序，但是在希尔排序中，由于相同的元素可能在不同的分组中，所以可能会造成相同元素位置的变化， 所以希尔排序是一个不稳定的排序。

希尔排序的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(n^s)，空间复杂度为 O(1)，不是稳定排序。

详细资料可以参考：《图解排序算法(二)之希尔排序》《数据结构基础 希尔排序之 算法复杂度浅析》

归并排序

归并排序是利用归并的思想实现的排序方法，该算法采用经典的分治策略。递归的将数组两两分开直到只包含一个元素，然后 将数组排序合并，最终合并为排序好的数组。

代码实现：


```
function mergeSort(array) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于0，直接返回，不需要排序
```

```
  // 如果不是数组或者数组长度小于等于0，直接返回，不需要排序
  if (!Array.isArray(array) || length === 0) return;
  if (length === 1) {
    return array;
  }
  let mid = parseInt(length >> 1), // 找到中间索引值
    left = array.slice(0, mid), // 截取左半部分
    right = array.slice(mid, length); // 截取右半部分
  return merge(mergeSort(left), mergeSort(right)); // 递归分解后，进行
排序合并
}
function merge(leftArray, rightArray) {
  let result = [],
    leftLength = leftArray.length,
    rightLength = rightArray.length,
    il = 0,
    ir = 0;
  // 左右两个数组的元素依次比较，将较小的元素加入结果数组中，直到其中一个数组的
元素全部加入完则停止
  while (il < leftLength && ir < rightLength) {
    if (leftArray[il] < rightArray[ir]) {
      result.push(leftArray[il++]);
    } else {
      result.push(rightArray[ir++]);
    }
  }
  // 如果是左边数组还有剩余，则把剩余的元素全部加入到结果数组中。
  while (il < leftLength) {
    result.push(leftArray[il++]);
  }
  // 如果是右边数组还有剩余，则把剩余的元素全部加入到结果数组中。
  while (ir < rightLength) {
    result.push(rightArray[ir++]);
  }
  return result;
}
```

归并排序将整个排序序列看成一个二叉树进行分解，首先将树分解到每一个子节点，树的每一层都是一个归并排序的过程，每 一层归并的时间复杂度为 O(n)，因为整个树的高度为 lgn，所以归并排序的时间复杂度不管在什么情况下都为 O(nlogn)。

归并排序的空间复杂度取决于递归的深度和用于归并时的临时数组，所以递归的深度为 logn，临时数组的大小为 n，所以归 并排序的空间复杂度为 O(n)。

归并排序的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(nlogn)，空间复杂度为 O(n)，是稳定排序。

详细资料可以参考：《图解排序算法(四)之归并排序》《归并排序的空间复杂度？》

快速排序

快速排序的基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据 都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

代码实现：


```
function quickSort(array, start, end) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
```

```
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1 || start >= end) return;

  let index = partition(array, start, end); // 将数组划分为两部分，并返回右部分的第一个元素的索引值
  quickSort(array, start, index - 1); // 递归排序左半部分
  quickSort(array, index + 1, end); // 递归排序右半部分
}
function partition(array, start, end) {
  let pivot = array[start]; // 取第一个值为枢纽值，获取枢纽值的大小
  // 当 start 等于 end 指针时结束循环
  while (start < end) {
    // 当 end 指针指向的值大等于枢纽值时，end 指针向前移动
    while (array[end] >= pivot && start < end) {
      end--;
    }
    // 将比枢纽值小的值交换到 start 位置
    array[start] = array[end];
    // 移动 start 值，当 start 指针指向的值小于枢纽值时，start 指针向后移动
    while (array[start] < pivot && start < end) {
      start++;
    }
    // 将比枢纽值大的值交换到 end 位置，进入下一次循环
    array[end] = array[start];
  }
  // 将枢纽值交换到中间点
  array[start] = pivot;
  // 返回中间索引值
  return start;
}
```

这一种方法是填空法，首先将第一个位置的数作为枢纽值，然后 end 指针向前移动，当遇到比枢纽值小的值或者 end 值 等于 start 值的时候停止，然后将这个值填入 start 的位置，然后 start 指针向后移动，当遇到比枢纽值大的值或者 start 值等于 end 值的时候停止，然后将这个值填入 end 的位置。反复循环这个过程，直到 start 的值等于 end 的 值为止。将一开始保留的枢纽值填入这个位置，此时枢纽值左边的值都比枢纽值小，枢纽值右边的值都比枢纽值大。然后在递 归左右两边的的序列。

当每次换分的结果为含 ⌊n/2⌋和 ⌈n/2⌉−1 个元素时，最好情况发生，此时递归的次数为 logn，然后每次划分的时间复杂 度为 O(n)，所以最优的时间复杂度为 O(nlogn)。一般来说只要每次换分都是常比例的划分，时间复杂度都为 O(nlogn)。

当每次换分的结果为 n-1 和 0 个元素时，最坏情况发生。划分操作的时间复杂度为 O(n)，递归的次数为 n-1，所以最坏 的时间复杂度为 $O(n^2)$。所以当排序序列有序的时候、快速排序有可能被转换为冒泡排序。

快速排序的空间复杂度取决于递归的深度，所以最好的时候为 O(logn)，最坏的时候为 O(n)。

快速排序的平均时间复杂度为 O(nlogn)，最坏时间复杂度为 O(n^2)，空间复杂度为 O(logn)，不是稳定排序。

详细资料可以参考：《图解排序算法(五)之快速排序——三数取中法》《关于快速排序的四种写法》《快速排序的时间和空间复杂度》《快速排序最好，最坏，平均复杂度分析》《快速排序算法的递归深度》

堆排序

堆排序的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行 交换，此时末尾就为最大值。然后将剩余 n-1 个元素重新构造成一个堆，这样会得到 n 个元素的次小值。如此反复执行， 便能得到一个有序序列了。

```
function heapSort(array) {
  let length = array.length;
  // 如果不是数组或者数组长度小于等于1，直接返回，不需要排序
  if (!Array.isArray(array) || length <= 1) return;
  buildMaxHeap(array); // 将传入的数组建立为大顶堆
  // 每次循环，将最大的元素与末尾元素交换，然后剩下的元素重新构建为大顶堆
  for (let i = length - 1; i > 0; i--) {
    swap(array, 0, i);
    adjustMaxHeap(array, 0, i); // 将剩下的元素重新构建为大顶堆
  }
  return array;
}
function adjustMaxHeap(array, index, heapSize) {
  let iMax,
    iLeft,
    iRight;
  while (true) {
    iMax = index; // 保存最大值的索引
    iLeft = 2 * index + 1; // 获取左子元素的索引
    iRight = 2 * index + 2; // 获取右子元素的索引
    // 如果左子元素存在，且左子元素大于最大值，则更新最大值索引
    if (iLeft < heapSize && array[iMax] < array[iLeft]) {
      iMax = iLeft;
    }
    // 如果右子元素存在，且右子元素大于最大值，则更新最大值索引
    if (iRight < heapSize && array[iMax] < array[iRight]) {
```

```
      iMax = iRight;
    }
    // 如果最大元素被更新了，则交换位置，使父节点大于它的子节点，同时将索引值跟新为被替换的值，继续检查它的子树
    if (iMax !== index) {
      swap(array, index, iMax);
      index = iMax;
    } else {
      // 如果未被更新，说明该子树满足大顶堆的要求，退出循环
      break;
    }
  }
}
// 构建大顶堆
function buildMaxHeap(array) {
  let length = array.length,
    iParent = parseInt(length >> 1) - 1; // 获取最后一个非叶子点的元素
  for (let i = iParent; i >= 0; i--) {
    adjustMaxHeap(array, i, length); // 循环调整每一个子树，使其满足大顶堆的要求
  }
}
// 交换数组中两个元素的位置
function swap(array, i, j) {
  let temp = array[i];
  array[i] = array[j];
  array[j] = temp;
}
```

建立堆的时间复杂度为 O(n)，排序循环的次数为 n-1，每次调整堆的时间复杂度为 O(logn)，因此堆排序的时间复杂度在 不管什么情况下都是 O(nlogn)。

堆排序的平均时间复杂度为 O(nlogn) ， 最坏时间复杂度为 O(nlogn) ， 空间复杂度为 O(1) ， 不是稳定排序。

详细资料可以参考： 《图解排序算法(三)之堆排序》 《常见排序算法 - 堆排序 (Heap Sort)》 《堆排序中建堆过程时间复杂度O(n)怎么来的？》 《排序算法之 堆排序 及其时间复杂度和空间复杂度》 《最小堆 构建、插入、删除的过程图解》

项目

1. 项目背景
2. 项目功能
3. 技术栈
4. 项目中遇到的问题（整理）*****

问题1
解决

问题2
解决

银行