301 和 302

- 301 永久性的重定向 永久性转移 从一个旧的网址调到一个新的网址 旧的网址销毁 这个资源不可访问 浏览器在拿到服务器返回的这个状态码 会自动调到一个新的URL地址，这个地址 从location首部获取 a地址变成b 搜索引擎抓取新的内容的同时 将旧的地址交换为重定向之后的地址
- 302 临时性的重定向 暂时性转移 从旧的网址跳到新的网址 旧的网址不变 旧地址a的资源还在 还可以访问 搜索引擎在抓取新的内容的同时 保存旧的网址
- 重定向： 地址a调到地址b 通过各种方法将各种网络请求重新定个方向转到其他位置
- 为什么要重定向 什么需要重定向 网站调整 网页新地址 网页拓展名改变 否则404
- 什么时候 进行 301 302 跳转 302 网页网站24-48小时 移动到一个新的位置 302 跳转 301 网站需要移除 需要到新的地址访问
- 尽量使用301跳转 网页劫持 URL劫持 就是一个地址a到地址b 地址a网页差 做了一个 302 重定向 到 地址b 网页好 搜索引擎排名靠前 所以a排名靠前 302 多个域名指向同一个网站 有封站的风险 301 告诉搜索引擎 地址弃用 转向一个新的地址 可以转移域名的权重 res(301,newUrl)

303

请求资源的路径改变 使用GET方法请求新的URL 类似302的功能 url location

400 bad request

请求报文存在语法错误 前后端接口数据

提交json json格式问题 400 bad request

405 method not allowed

请求方式错误 与后台规定的方式不符合 get post delete

401 用户未提供身份验证凭据 身份验证未通过

403 用户身份验证通过 资源访问权限受限

404 请求资源不存在 或者 不可用

410 请求1资源已转移 不可用

415 客户端返回格式不正确

422 客户端附件无法处理

429 客户端请求次数太多

500 客户端请求有效 服务端处理时发生意外 服务端遇到错误 无法完成请求

503 服务器无法处理请求 一般用于网站维护 目前无法使用

难点 304 307 504 502 501 等等

1xx:相关信息

2xx: 操作成功

3xx: 重定向

4xx: 客户端错误

5xx: 服务端错误

200 成功处理接收到请求 并返回相应网页结果

201 生成了新的资源，用户新建数据或者修改数据成功

202 服务器收到了请求 但未处理 用于异步操作

203

204 从服务器删除资源 资源不存在

Restful API

Restful 集中请求格式

- GET 从服务器取出资源 GET/articals/12?categories/12 200 ok
- POST 在服务器新建一个资源 201 created
- PUT 在服务器更新资源 200ok
- PATCH 在服务器部分更新部分资源 200 ok
- DELETE 从服务器删除资源 204 no content 用户删除成功

POST和PUT区别

- POST常用于提交数到服务端 新建资源时使用
- PUT用于修改更新已存在的资源使用
- GET 用于查询数据时使用 不推荐资源的更新以及新建 容易造成xxs或者CSRF攻击

GET和POST区别

1.url可见性： get，参数url可见 post，url参数不可见

**get把请求的数据放在url上，即HTTP协议头上，其格式为：以?分割URL和传输数据，参数之间以&相连；post把数据放在HTTP的包体内（requrest body）

2.传输数据的大小： get一般传输数据大小不超过2k-4k post请求传输数据的大小根据php.ini 配置文件设定，也可以无限大

**get提交的数据最大是2k（原则上url长度无限制，那么get提交的数据也没有限制咯？限制实际上取决于浏览器，浏览器通常都会限制url长度在2K个字节，即使(大多数)服务器最多处理64K大小的url，也没有卵用）；post理论上没有限制。实际上IIS4中最大量为80KB，IIS5中为100KB

3.数据传输上： get，通过拼接url进行传递参数 post，通过body体传输参数

**GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)

4.后退页面的反应： get请求页面后退时，不产生影响 post请求页面后退时，会重新提交请求

**GET在浏览器回退时是无影响的，POST会再次提交请求

5.缓存性： get请求是可以缓存的 post请求不可以缓存

**GET请求会被浏览器主动cache，而POST不会，除非手动设置

6.安全性： 都不安全，原则上post肯定要比get安全，毕竟传输参数时url不可见，但也挡不住部分人闲的没事在那抓包玩，浏览器还会缓存get请求的数据。安全性个人觉得是没多大区别的，防君子不防小人就是这个道理。对传递的参数进行加密，其实都一样

7.GET请求只能进行url编码，而POST支持多种编码方式

8.GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留

9.GET只接受ASCII字符的参数的数据类型，而POST没有限制

那么，post那么好为什么还用get？get效率高！

call apply bind

this.call(arr,...arguments) Function.proptype.mycall = function(context) { if (typeof this !== 'function') { throw TypeError('not a Function') } context=context || window context.fn=this let arg = [...arguments].slice(1) let result = context.fn(...arg) delete context.fn() return result

}

Function.prototype.myapply = function(context) { if (typeof this !== 'function') { throw TypeError('not a Function') } context = context || window context.fn = this if (arguments[1]) { return result=context.fn(...arguments[1]) } else { retun result = context.fn() } delete context.fn() return result }

手写bind

Function.prototype.mybind = function(context) { if (typeof this !== 'function') { throw TypeError('not a Function') } let _this = this let arg = [...arguments].slice(1) return function F() { if (this instanceof F) { return new _this =(...arg,...arguments) } else { return _this.apply(context,arg.concat(...arguments)) } } }

function instanceOf() { let leftValue = left.**proto** rightValue = right.prototype while(true) { if (leftValue === null) { return false } if (leftValue === rightValue) { return true } leftValue = leftValue.**proto** } }

原型链 原型

函数的原型链对象 默认指向函数本身 原型对象除了原型属性 还有继承 原型链指针__proto__ 该指针始终指向上一层的原型对象 Object.prototype.__prototype=null Object.**proto** === Object.prototype true

手写Promise

三个状态 class Promise { constructor() { this.state='pending' this.value=undefined this.reason=undefined let resolve=vale=>{ if(this.state==='pending') { this.state='fulfilled' this.value=value } } let reject=value=>{ if(this.state==='pending') { this.state='rejected' this.reason=value } } try{ fn(resolve,reject) } catch (e) { reject(e) } } then(onFulfilled,onRejected) { switch(this.state) { case 'fulfilled': onFulfilled() break; case 'rejected': onRejected() break; default: } } } 三个状态切换 pending fulfilled rejected 定义自执行函数 then

浅拷贝 和 深拷贝

浅拷贝 是对源对象的值 进行引用 当源对象 发生改变 拷贝对象也发生改变 浅拷贝是拷贝一层 高层级 拷贝引用 深拷贝 就是 源对象的值改变 不影响拷贝对象的值 多层级的拷贝 let copy1 = {x:1} let copy2 = Object.assign({},copy1) copy2.x=6

console.log(copy2.x) console.log(copy2)

JSON.parse(JSON.stringfiy)

function deepClone() { let copy = copy1 instanceof Array ? [] : {} for (let i in copy1) { if(copy1.hasOwnProperty(i)) { copy[i] = typeof copy1[i] === 'object' ? deepClone(copy1[i]) : copy1[i] } } return copy } let copy3 = deepClone(copy1) console.log(copy3)

6 { x: 1, y: 2, z: { x: 3 }, a: { x: 4 } }

类的创建和继承

数据结构与算法

- 栈 一种先进后出的有序集合 新插入的元素放在栈顶 旧的元素在栈底 新元素靠近栈顶 旧元素靠近栈底
- 队列 遵循先进先出的有序序列 新插入的元素放在队尾
- 链表 存储有序元素的集合 不同于数组 链表的元素不是有序的 不是连续防止的 是由该元素的结点和指向下一元素的指针或链接组成的
- 集合
- 字典
- 散列表
- 树
- 图

栈 一种 先进后出的有序集合

栈是一种遵从先进后出 (LIFO) 原则的有序集合；新添加的或待删除的元素都保存在栈的末尾，称作栈顶，另一端为栈底。在栈里，新元素都靠近栈顶，旧元素都接近栈底。 通俗来讲，一摞叠起来的书或盘子都可以看做一个栈，我们想要拿出最底下的书或盘子，一定要现将上面的移走才可以。

1. 定义一个栈 class stack { constructor() { this.items=[] } }
2. 入栈 this.items.push()
3. 出栈 this.items.push()
4. 栈的末位 peek this.items.length-1
5. 栈是否为空 get isEmpty
6. 栈的大小尺寸 get size
7. 清空栈的内存 clear() { this.items=[] }

8. 输出栈的所有元素 print() { console.log(this.items.toString()) }
9. 实例化 new Stack()

队列 先进先出的有序序列

与栈相反，队列是一种遵循先进先出 (FIFO / First In First Out) 原则的一组有序的项；队列在尾部添加新元素，并从头部移除元素。最新添加的元素必须排在队列的末尾。 在现实中，最常见的例子就是排队，吃饭排队、银行业务排队、公车的前门上后门下机制...，前面的人优先完成自己的事务，完成之后，下一个人才能继续。

1. 定义一个队列 class queue{ constructor(items) { this.items=items || [] } }
2. 入队 this.items.push()
3. 出队 this.items.shift()
4. 队的首位 front this.items[0]
5. 队是否为空 get isEmpty
6. 队的大小尺寸 get size
7. 清空队的内存 clear() { this.items=[] }
8. 输出队的所有元素 print() { console.log(this.items.toString()) }
9. 实例化 new Queue()

- 优先队列 循环队列

1. 实现一个优先队列，有两种选项：设置优先级，然后在正确的位置添加元素；或者用入列操作添加元素，然后按照优先级移除它们。
2. 为充分利用向量空间，克服"假溢出"现象的方法是：将向量空间想象为一个首尾相接的圆环，并称这种向量为循环向量。存储在其中的队列称为循环队列（Circular Queue）。这种循环队列可以以单链表、队列的方式来在实际编程应用中来实现。

两个栈 实现一个队列

两个队列 实现一个栈

链表

链表存储有序的元素集合，但不同于数组，链表中的元素在内存中并不是连续放置的。每个 元素由一个存储元素本身的节点和一个指向下一个元素的引用(也称指针或链接)组成。 数组可以查找任何元素 链表 必须从开头元素根据节点指针慢慢迭代查找 链表

- 定义节点 class Node { constructor(element) { this.head = element this.next=null } } class LinkedList { constructor() { this.head = null this.length=0 } append(element) { const node = new Node(element) let current = null if (this.head === null) { this.head = node } else { current = this.head while(current.next) { current=current.next } current.next=node } this.length++ } } const linkedList = new LinkedList() linkedList.append(1) linkedList.append(2) console.log(linkedList)

双向链表

双向链表和普通链表的区别在于，在链表中， 一个节点只有链向下一个节点的链接，而在双向链表中，链接是双向的:一个链向下一个元素， 另一个链向前一个元素，如下图所示:

双向链表提供了两种迭代列表的方法:从头到尾，或者反过来。我们也可以访问一个特定节 点的下一个或前一个元素。在单向链表中，如果迭代列表时错过了要找的元素，就需要回到列表 起点，重新开始迭代。这是双向链表的一个优点。

循环链表

树

树是一种非顺序数据结构，一种分层数据的抽象模型，它对于存储需要快速查找的数据非常有用。 一个树结构存在父子关系的节点 每个节点有一个或者零个父子节点和叶子节点

节点

- 根节点
- 内部节点：有子节点
- 外部节点： 没有子节点
- 子树：大大小小节点组成的树
- 深度：节点到根节点的数量
- 高度：深度的最大值
- 层级：节点层级来划分

二叉树

二叉树中的节点最多只能有两个子节点：一个是左侧子节点，另一个是右侧子节点。这些定 义有助于我们写出更高效的向/从树中插人、查找和删除节点的算法。二叉树在计算机科学中的 应用非常广泛。

二叉搜索树（BST）是二叉树的一种，但是它只允许你在左侧节点存储（比父节点）小的值， 在右侧节点存储（比父节点）大（或者等于）的值。上图中就展现了一棵二叉搜索树。

注：不同于之前的链表和集合，在树中节点被称为"键"，而不是"项"。

二叉搜索树 左节点小于父节点 右节点大于父节点 class node { constructor(key) { this.key=key this.left=null this.right=null } } class BinarySearchTree { constructor() { this.root=null } insert(key) { const newNode = new Node(key) const insertNode=(newNode,node)=>{ if(newNode.key<node.key) { if(node.left===null) { node.left=newNode } else { insertNode(node.left,newNode) } } } } }

树的遍历

遍历一棵树是指访问树的每个节点并对它们进行某种操作的过程。但是我们应该怎么去做呢？应该从树的顶端还是底端开始呢？从左开始还是从右开始呢？

访问树的所有节点有三种方式：中序、先序、后序。

图

算法 排序算法

动态规划 贪心算法

前端

tcp三次握手 四次挥手

客户端c发起请求连接服务器端s确认 服务端也发起连接确认客户端确认

第一次握手： 客户端发送一个请求连接 服务端确认自己可以接收到客户端放的报文 第二次握手： 服务端s向客户端发送一个链接，确认客户端可以收到服务端发送的报文段 第三次握手： 服务端确认客服端收到了自己发送的报文段

- 四次挥手 第一次分手： 客户端向服务器端发送一个FIN报文段，Seq,Ack客户端进入FIN_WAIT状态，主机没有内容要发送给服务端了 第二次分手：服务端收到了客户端发送的FIN报文段，向主机客户端回一个ACK报文段，Ack+1 Seq+1 客户端进入FIN_WAIT2状态，服务器端同意客户端的关闭请求 第三次分手：服务器端向客户端发送FIN报文段，请求关闭连接，服务端进入LAST_ACK 第四次分手： 客户端收到服务端发送的报文段FIN，向服务端发送ACK报文段，客户端进入TIME_WAIT服务端收到ACK报文段，关闭连接，此时服务端等待2ms后没有收到回复，server正常关闭，主机1关闭连接

- 为什么要三次握手 为了防止已失效的连接请求报文段突然有传送给服务端，因而产生错误 “已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client不会向server的确认发出确认。server由于收不到确认，就知道client并没有要求建立连接。”

- 为什么要四次挥手 那四次分手又是为何呢？TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。如果要正确的理解四次分手的原理，就需要了解四次分手过程中的状态变化。

OSI中的层	功能	TCP/IP协议族
应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层	数据格式化，代码转换，数据加密	没有协议
会话层	解除或建立与别的接点的联系	没有协议
传输层	提供端对端的接口	TCP，UDP
网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层	以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802。IEEE802.2

- tcp udp 传输层
- http 应用层
- ip 网络层

懒加载的节流和防抖 代码实现 说明原理 和 使用场景

用户在搜索的时候，在不停敲字，如果每敲一个字我们就要调一次接口，接口调用太频繁，给卡住了。 用户在阅读文章的时候，我们需要监听用户滚动到了哪个标题，但是每滚动一下就监听，那样会太过频繁从而占内存，如果再加上其他的业务代码，就卡住了。 所以，这时候，我们就要用到 防抖与节流 了。 用户搜索每敲一个字调用一次接口 滚动监听

- 防抖：任务频繁触发的情况下，志愿任务出发的间隔超过指定间隔的时候，任务才会执行 有个输入框，输入之后会调用接口，获取联想词。但是，因为频繁调用接口不太好，所以我们在代码中使用防抖功能，只有在用户输入完毕的一段时间后，才会调用接口，出现联想词。 小伙伴们可以尝试看着上面的案例，先自己实现一遍这个场景的解决，如果感觉不行， 一般可以使用在用户输入停止一段时间过后再去获取数据，而不是每次输入都去获取，如下图：

防抖就是将一段时间连续多次触发转化未一次触发 用户输入框，输入停止一段时间后去解耦抓取数据 而不失输入几个字就抓取 太频繁了 原理：主要是判断是否到达等待时间，如果没到达的话就继续加入任务队列等待执行。使用方法： 判断任务触发间隔是否达到指定间隔 没有达到继续任务触发 function debounce(fn,wait) { var timeout; return function() { let context=this let args=arguments clearTimeout(timeout) let timeout=setTimeout(()=>{ timeou=null fn.apply(context,args) },wait)

```
}
```

} 防抖是 用户在输入框连续输入 转换成一次触发数据 判断是否到达等待时间

- 节流： 指定时间间隔内只会执行一次任务。 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。

降低事件出发频率 滚动条 不用每滚动一次 就监听 计算滚动

- 应用场景 懒加载要监听滚动条的位置 使用节流按一定时间频率去获取 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。 用户提交按钮 使用节流 只允许一定时间点击一次

function throttle(fn.wait) { var timeout; var prev=0; if(!timeout) { let context=this let args = arguments let timeout=setTimeout(()=>{ timeout=null fn.apply(context,args) },wait) } } 防抖和节流的目的都是为了减少不必要的计算，不浪费资源，只在适合的时候再进行触发计算。源码可以在 这个项目 里的fn模块看到，另外还有许多实用的功能

防抖和节流都是减少不必要的计算 在一定时间间隔进行触发计算

重绘与回流

在说浏览器渲染页面之前，我们需要先了解两个点，一个叫 浏览器解析 URL，另一个就是本章节将涉及的 重绘与回流：

重绘(repaint)：当元素样式的改变不影响布局时，浏览器将使用重绘对元素进行更新，此时由于只需要 UI 层面的重新像素绘制，因此损耗较少。 常见的重绘操作有：

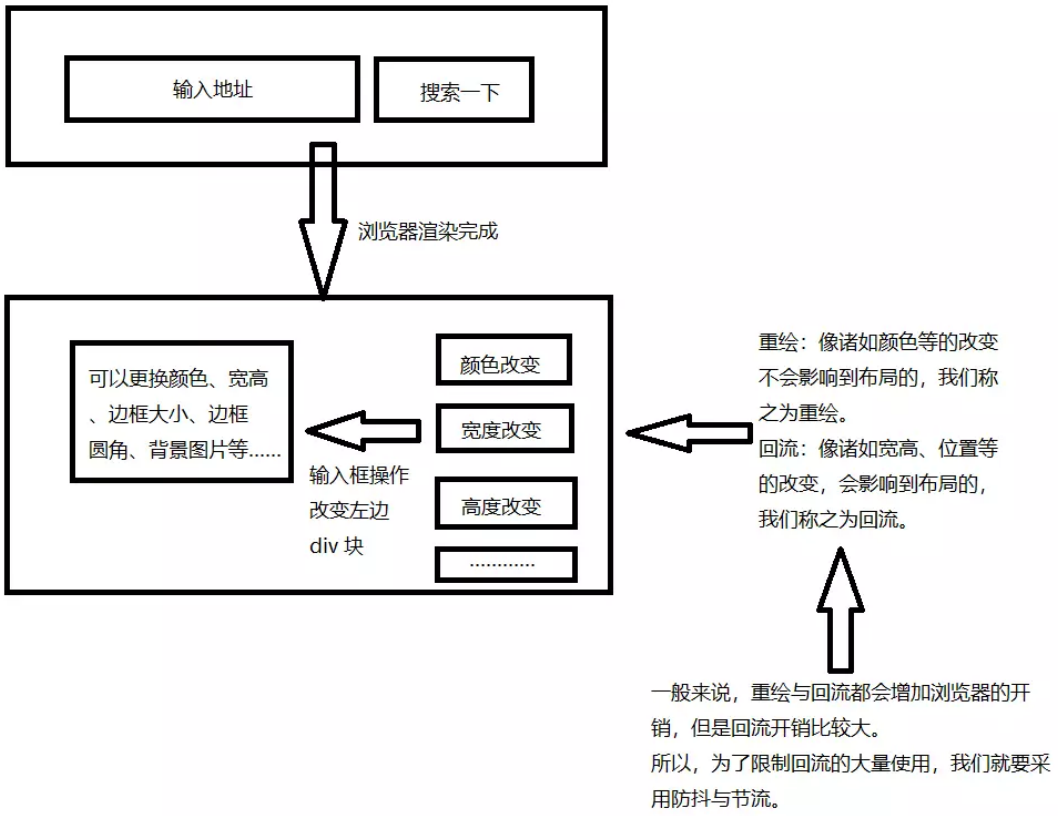
改变元素颜色 改变元素背景色 more ..... 回流(reflow)：又叫重排（layout）。当元素的尺寸、结构或者触发某些属性时，浏览器会重新渲染页面，称为回流。此时，浏览器需要重新经过计算，计算后还需要重新页面布局，因此是较重的操作。 常见的回流操作有：

页面初次渲染 浏览器窗口大小改变 元素尺寸/位置/内容发生改变 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（:hover……） more ..... 重点：回流必定会触发重绘，重绘不一定会触发回流。重绘的开销较小，回流的代价较高。 看到这里，小伙伴们可能有点懵逼，你刚刚还跟我讲着 防抖与节流 ，怎么一下子跳到 重绘与回流 了？

OK，卖个关子，先看下面场景：

界面上有个 div 框，用户可以在 input 框中输入 div 框的一些信息，例如宽、高等，输入完毕立即改变属性。但是，因为改变之后还要随时存储到数据库中，所以需要调用接口。如果不加限制..... 看到这里，小伙伴们可以将一些字眼结合起来了：为什么需要 节流，因为有些事情会造成浏览器的 回流，而 回流 会使浏览器开销增大，所以我们通过 节流 来防止这种增大浏览器开销的事情。

形象地用图来说明：



这样，我们就可以形象的将 防抖与节流 与 重绘与回流 结合起来记忆起来。

那么，在工作中我们要如何避免大量使用重绘与回流呢？：

避免频繁操作样式，可汇总后统一一次修改 尽量使用 class 进行样式修改，而不是直接操作样式 减少 DOM 的操作，可使用字符串一次性插入 OK，至此我们就讲完两个部分了，那么问题又来了：“浏览器渲染过程中，是不是也有重绘与回流？““从浏览器输入 URL 到渲染成功的过程中，究竟发生了什么？”

我们，继续深入探索.....

使用节流 防止增大浏览器开销的问题

css 垂直居中 不定宽高 和定宽高的2种实现方法

不定宽高的居中：

css

- { m0 p0 } .box1 { w100px h100px poa t0 l0 r0 b0 tranform:translateX(50%,50%) m0-a }

法2 .box1 { w100px h100px l50% t50% poa mt-50px ml-50px }

定宽高的垂直居中: .box1 { w100px h100px l0 t0 r0 b0 tranform:translateX(50%,50%) m0-a } img { w50px h50px pt25px pl25px por }

继承 原型链继承 构造继承 寄生继承 组合继承

- 原型链继承 function Aniaml() { this.name=name } function Cat() {} cat.prototype = new Aniaml cat.prototype.name='cat'
- 构造继承： function Cat() { Animal.call(this) this.name=name || 'wqs' }
- 实例继承 为父类实例添加新的特性 作为子类实力返回
- 拷贝继承 拷贝父类元素的属性和方法
- 组合继承 构造继承+原型继承组合
- 寄生组合继承 通过寄生方式，在构造继承上加一个Super() 没有实力和方法 让他的原型链指向父类的原型链 砍掉父类的实例属性 这样在调用两次父类的构造函数的时候，就不会初始化两次实例方法/属性如何判断是那种类型

ajax 手写ajax ajax后台对接数据如何实现 get post

- ajax原理
    - 相对于在用户和服务器之间加一个中间层 (ajax引擎) 是用户操作与服务器响应异步化
- 优点
    - 在不刷新整个页面的前提下与服务器通信维护数据 不会导致页面的重载 可以把前端服务器的任务转嫁到客户端处理，减轻服务器负担，节省宽带
- 劣势

    - 不支持back 对搜索引擎的支持比较弱；不容易调试

- 怎么解决

    - 通过location.hash 值来解决Ajax过程中导致的浏览器前进后退按键 解决以前被人遇到的重复加载问题
    - 主要比较前后hash值 看是否相等 判断是否触发ajax
      funtion getData(url) { var xhr = new XMLHttpRequest();//创建一个对象 一个异步调用的对象 xhr.open('get',url,true) //设置http请求，设置请求的方式，url以及验证身份 xhr.send()//发送一个http请求 xhr.onreadystatechange=function() { //设置一个http请求状态的函数 if(xhr.readyState==4 && xhr.status===200) { console.log(xhr.responseText) //获取异步调用返回的数据 } } } Promise(getData(url)).resolve(data=>data)

- AJAX状态码：

    - 0 未初始化 还没有调用send()方法
    - 1 载入 已经调用send方法 正在发送请求
    - 2 载入完成 send() 方法执行完毕
    - 3 交互 正在解析相应内容
    - 4 完成 相应内容解析完成 可以在客户端调用了

function getData(url) { var xhr = new XMLHttpRequest() //手写一个创建一个对象 异步调用的对象 xhr.open('get',url,true)// 设置http请求 设置请求的方式 url 以及验证身份 xht.send() //发送http请求 xhr.onreadystatechange= function() { //设置一个http请求状态函数 if(xhr.readyState==4 && xhr.status===200) { console.log(xhr.responseText) //获取异步调用返回的数据 } } }
状态码 0 send方法没有调用 1 send方法调用 2 send执行 3 解析 4 响应解析 客户端调用

es6 了解那几个 let var const

promise

- 异步回调(如何解决回调地狱)
  promise generator async/awit
  promise:

1. 是一个对象,用来传递异步操作的信息。代表着某个未来才会才会知道结果的时间，并未这个时间提供同意的api 供进异步处理

2. 有了这个对象，就可以让异步操作以同步的操作的流程表达出来，避免层层嵌套的回调地狱

3. promise代表一个一部状态，有三个状态 pending进行中 resolve完成 reject失败

4. 一旦状态改变 就不会再变 任何时候都可以得到结果。从进行中变为已完成或者失败
   promise.all() 里面状态都在改变 输出得到一个数组
   promise.race() 里面就志愿一个状态变为fulfilled或者 rejected 即输出
   promise.finally() 不管promise对象状态都如何 都会执行的操作 本质上还是一个then方法的特例

   class Promise { constructor(fn) { let state='pending' let value=undefined let reason = undefined let

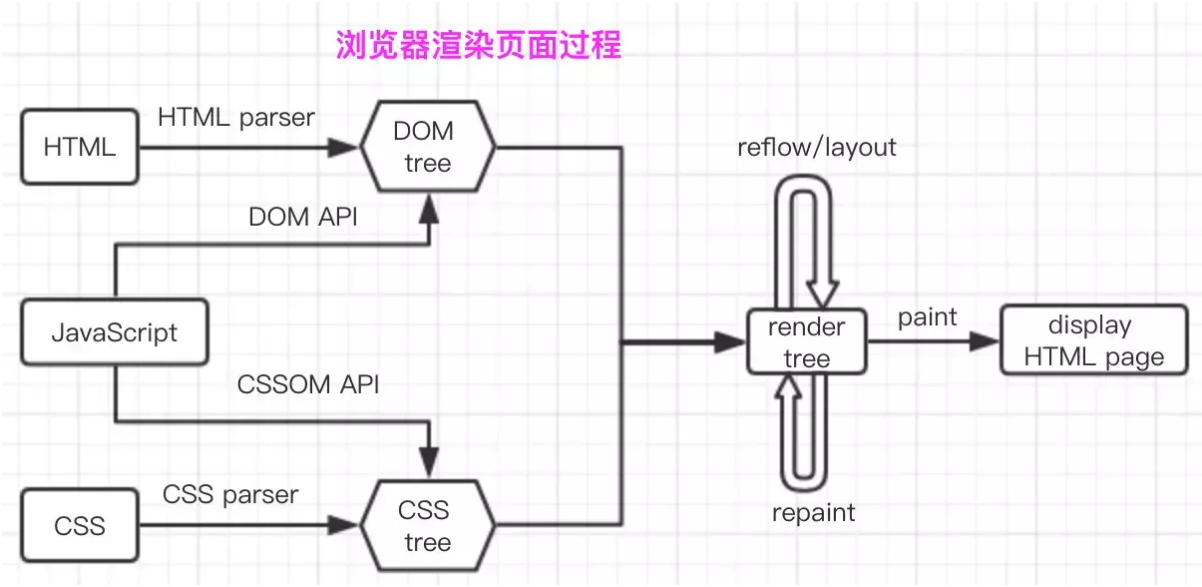
resolve=value(()=>{ if(state='pengding') { this.state='fullfilled' this.value=value } }) let reject=value(()=>{ if(state==='pengding') { this.state='rejected' this.reason=value } }) try() { fn(resolve,reject) } catch(e) { reject(e) }

} then(onFulfilled,onRejected) { switch(state) { case 'fulfilled': onFulfilled() break; case 'rejectd': onRejected() break; default: } } }

浏览器渲染过程 DOM渲染过程

DOM->CSSDOM->render->layout->print

- 流程： 解析html构建DOM树->构建render树->布局render树->绘制、
- 概念：
  - 构建DOM树 浏览器渲染引擎根据HTMLparse解析HTML文档，根据深度遍历将 HTML标签转换DOM树中的节点，生成内容树 DOM树
  - 构建渲染树 解析css样式文件 根据cssparse解析 生成 CSSDOM tree
  - 浏览器将js 通过DOM API 和 CSSDOM API 应用到布局 按要求呈现出结果
  - 布局渲染树 根据DOM CSSDOM 来构造 render树 从根结点递归调用 计算每一个元素的大小和所在位置 遍历渲染树 使用UI后端层来绘制render树
  - layout 重排 回流 当render树里的元素规模尺寸发生变化 render 树会重新构建页面 重新布局 重新计算元素节点所在位置 每个页面都要一次回流 重排一定影响重绘 重绘不一定影响重排
  - repaint 重绘 render树元素的字体背景改变 重绘
  - paint 遍历 render树 调用api 绘制每个节点



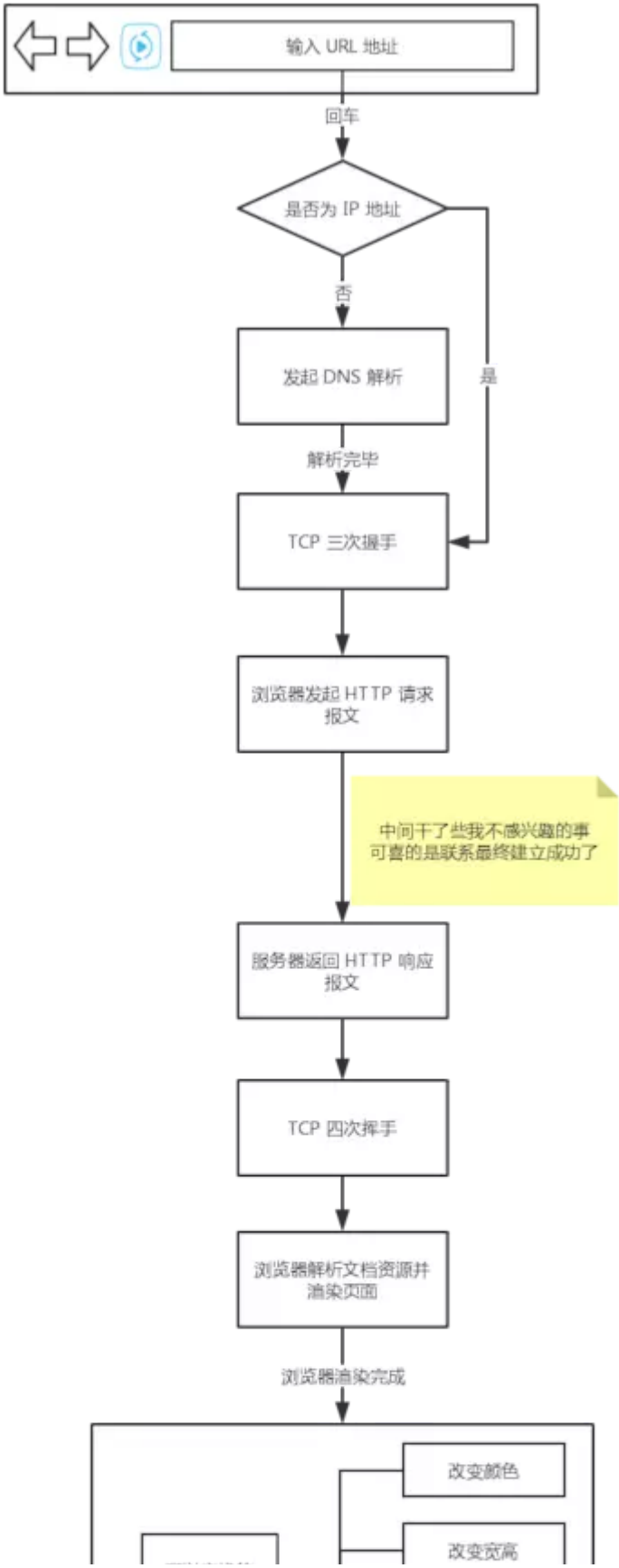
html -> DOM tree layout/reflow cssDOM ->CSStree -> render tree ---paint---> diplay HTML page js DOMAPI CSSDOM API repaint

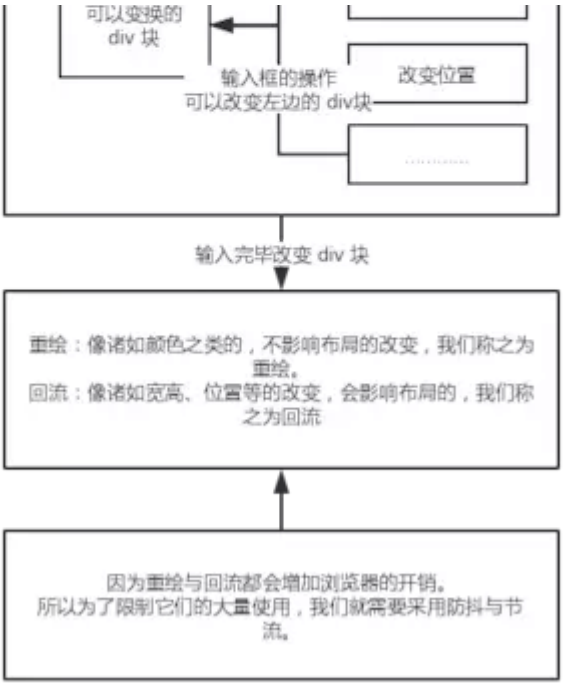
浏览器输入URL发生什么

浏览器解析 URL

1. 用户输入URL

2. 浏览器启动DNS 解析域名 获取域名对应的ip地址
3. 建立tcp三次握手
4. 客户端发送http请求报文段
5. 服务器端返回http响应报文段
6. 关闭四次挥手
7. 浏览器解析文档资源 渲染页面 DOM树渲染完成页面





em 和 rem 适配

em相对父元素 rem相对自身元素

前端性能优化

- 首屏加载 按需加载 懒加载
- 资源预加载
- 图片压缩 base64 内嵌图片
- 合理缓存DOM对象
- 不滥用web字体
- 使用事件代理 避免直接事件绑定
- viewport固定屏幕渲染 加速页面渲染内容
- touchstart代替click click300ms延迟
- tranfrom：tranlateZ(0) 开启硬件GUP加速

canvas

今日总结 防抖节流 三次握手 四次挥手 DOM树渲染 Promise 重绘回流 Ajax 等等

js闭包

- 闭包：能够读取其他函数内部变量的函数，定义在一个函数内部的函数，内部函数能够对外部函数变量的引用
- 闭包的用途
    - 读取其他函数内部的变量
    - 让变量的值始终保持在内存中
    - javascript闭包的应用 都有关键词 return

明日目标任务：

Event bus

Vue 数据监听MVVM Object.defineProperty()

proxy

ES6 set map

逆波兰表达式

setTimeout setInterval

第k大

webpack打包 插件

tarnsition animated.css

better-scroll

flex

手写发布订阅

网络协议

二叉树叶子节点

单链表

算法

大疆笔试

1. 什么情况下会引起死锁

- 什么是死锁 多个进程再运行过程中因争夺资源而造成的一种僵局 当进程处于这种僵持状态时，他们都无法再向前推进。
- 产生死锁的原因
    - 争夺资源 系统中的资源 可剥夺资源cpu 主存 不可剥夺资源 打印机 临时资源（信号 消息）
    - 进程间推进顺序非法 不能同时推进
- 死锁产生的必要条件
    - 互斥条件 一段时间内某资源仅为某一进程所占用 排他性控制
    - 请求和保持条件： 当进程因请求资源而阻塞时，对已获得的资源保持不放
    - 不剥夺条件： 进程以获得资源再未使用完之前 不能剥夺
    - 环路等待条件： 在发生死锁时 必然存在一个进程 --自愿的环形链（p0 p1 p2 p3 pn}
- 解决死锁的基本方法
    - 预防死锁
    - 获得性一次分配 破坏请求 一次性分配资源 不会有请求
    - 破坏请求保持条件
    - 破坏不可剥夺条件
    - 破坏环路等待条件 已确定顺序获得锁 超时放弃

2. Math.round() Math.ceil() Math.floor()

- Math.round() 四舍五入取整 Math.round(1.2) 1 Math.round(1.8) 2 Math.round(-1.2) -1 Math.round(-1.8) -2 Math.round(-7.5) -7 Math.round(-7.6) -8 四舍五入

- Math.ceil() 对于一个数向上取整 Math.ceil(x) Math.ceil(1.2) 2 Math.ceil(1.8) 2 Math.ceil(-1.2) -1 Math.ceil(-1.8) -1 往大的数取整

- Math.floor() 对于一个数向下取整 Math.floor(x) Math.floor(1.2) 1 Math.floor(1.8) 1 Math.floor(-1.2) -2 Math.floor(-1.8) -2

3. 闭包

- 闭包 在函数内部定义一个函数 内部函数对外部函数变量值的引用

    - 因为作用域链 外部不能访问内部的变量和方法 通过闭包 返回内部的方法和变量来给外部 闭包 垃圾回收机制 占用内存

- 怎么清楚消除闭包

    - 标记清除 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记（可以使用任何标记方式）。然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记。而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。最后，垃圾收集器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。
    - 引用计数 引用计数（reference counting）的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值赋给该变量时，则这个值的引用次数就是1。如果同一个值又被赋给另一个变量，则该值的引用次数加1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减1。当这个值的引用次数变成0 时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾收集器下次再运行时，它就会释放那些引用次数为零的值所占用的内存。

- 导致问题： 导致循环函数无法回收
- 解决： 将用完的函数或者变量置未null var a = {

  name :1,

  b:function(){ d=1 // this.name++ return this.name

  }

} console.log(a.d) console.log(a.b())

```
var c = a.b

console.log(c())
```

冒泡排序 插入排序 选择排序 快速排序 希尔排序 归并排序 二分查找 堆排序 桶排序 计数排序 基数排序

分析一个排序算法

复杂度是整个算法学习的精髓

- 时间复杂度： 一个算法执行所耗费的时间
- 空间复杂度： 执行完一个程序内存大小
- 执行效率 内存消耗 稳定性

冒泡排序

- 思想： 比较相邻两个元素，如果反序交换位置，直到没有反序记录为止 每次冒泡操作都会对相邻两个元素进行比较 看看是否满足大小关系 不满足则互相交换 重复n次
- 优点：排序算法基础 简单容易理解
- 缺点： 比较次数多 效率低 优化 const bubbleSort = arr => { let len = arr.length if(len<=1>) return ; for(let i=0;i<len-1;i++) { let hasChange=false for(let j=0;j<len-i+1;J++) { if(arr[j]>arr[j+1]) { const temp=arr[j+1] arr[j+1]=arr[j] arr[j]=temp hasChange=true

```
    }
}
if(!hasChange) break;
```

} return arr } console.log(bubbleSort([3,4,1,2])) 1,2,3,4 3 4 1 2 3 4 1 2 3 1 4 2 3 1 2 4 1 3 2 4 1 2 3 4 冒泡排序每一轮都是和后面的元素比较大小

- 空间复杂度 O(1)
- 时间复杂度 时间复杂度 平均O(n^2) 最好O(n) 最差O(n^2)

插入排序 直接插入排序 优化后的 折半排序 和 希尔排序

思想： 讲一个记录插入到已拍好的有序列表中，得到新的记录数增1的有序列表 扑克牌从小到大 或者 从大到小 构建有序序列 未排序数据 在已排序中从后向前扫描 找到相应位置并插入 const insertSort = arr => { let len = arr.length if(len<=1>) return; let preIndex, current for(let i=1;i<len;i++) { preIndex=i-1 current=arr[i] //当前元素 while(preIndex>0 && arr[preIndex]>current) { arr[preIndex+1]=array[preIndex] preIndex-- } if(preIndex+1 != i) { //避免同一个元素赋值 arr[preIndex] = current } } }

3 4 1 2

1 3 4 2 1 2 3 4

从排好序的直接插入 从第一个元素开始，该元素可以认为已经被排序； 取出下一个元素，在已经排序的元素序列中从后向前扫描； 如果该元素（已排序）大于新元素，将该元素移到下一位置； 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置； 将新元素插入到该位置后； 重复步骤 2 ~ 5。

- 空间复杂度 O(1)
- 时间复杂度 平均O(n^2) 最好O(n) 最差 O(n^2)

折半排序

取 0 ~ i-1 的中间点 (m = (i-1) >> 1)，array[i] 与 array[m] 进行比较，若 array[i] < array[m]，则说明待插入的元素 array[i] 应该处于数组的 0 ~ m 索引之间；反之，则说明它应该处于数组的 m ~ i-1 索引之间。 重复步骤 1，每次缩小一半的查找范围，直至找到插入的位置。 将数组中插入位置之后的元素全部后移一位。 在指定位置插入第 i 个元素。 x >> 1 是位运算中的右移运算，表示右移一位，等同于 x 除以 2 再取整，即 x >> 1 == Math.floor(x/2) 向下取整 O(log n^2) const binaryInsertSort = arr => { let len = arr.length if(len<1) return; let i,j,high,low,mid,current for(let i=1;i<len;i++) { low=0 high=i-1 current=arr[i] while(low<=high) { mid=(low+high)>>1 if(arr[i]>arr[mid]){ low = mid+1 } else { high=mid-1 } } for(j=i;j>low;j--) { arr[j]=arr[j-1] } arr[low]=current }

```
return arr
```

}

希尔排序 时间复杂度 O(n^1.3)

快速排序 时间复杂度 O(nlogn)

思想： 选一个中间值 左右两个 比它小的放左边 比它大的放右边 然后连接数组 const quickSort = arr => { let len = arr.length if(len<=1) { return arr } mid=Math.floor(len/2) //取基准点的值，splice(index,1) 则返回的是含有被删除的元素的数组。 val = arr.splice(mid, 1) midVal = val[0] let left = [] let right = [] for(let i=0;i<len;i++) { if(arr[i]<midVal) { left.push(arr[i]) }else { right.push(arr[i]) } } return quickSort(left).concat(midVal,quickSort(right)) } console.log(quickSort([5,4,3,2,1]))

选择排序

思想： 未排序 找到最大或者最小数据 放到已排序数组中

- 空间复杂度
- 时间复杂度 平均O(n^2) 最佳O(n^2) 最差O(n^2) 选择排序基本思想：是通过n-i次关键字间的比较，从n-i个记录中选出关键字最小的额记录，并和第i个记录交换位置。

过程：先在未排序序列中找到最小（大）元素，存放在已排序序列的起始位置；然后从剩余未排序元素中继续寻找最小（大）元素，然后放在已排序的末尾，直到所有元素排序完毕。 const selectSort = arr => { let len = arr.length let midIndex,temp for(let i=0; i<len-1;i++) { midIndex=i for(let j=i+1;j<len;j++) { if(arr[midIndex]>arr[j]) { midIndex=j } } temp=arr[i] arr[i]=arr[midIndex] arr[midIndex]=temp } return arr } // 测试 const array = [5, 4, 3, 2, 1]; console.log('原始array:', array); selectionSort(array); // 原始 array: [5, 4, 3, 2, 1] // array: [1, 4, 3, 2, 5] // array: [1, 2, 3, 4, 5] // array: [1, 2, 3, 4, 5] // array: [1, 2, 3, 4, 5]

归并排序

二分查找 就是折半查找

const bianrySearch = (arr,key) => { let len = arr.length let low=0 let high = len-1 while(low<=high) { mid = (low+high)>>1 if(key===arr[mid]) { return mid } else if(key>arr[mid]) { low = mid +1 } else if(key<arr[mid]) { high=mid-1 } else{ return -1 } } return -1 } console.log(binarySearch([3,4,1,2],2))

递归实现

头条

1. css的盒模型

- 标准盒模型 contentbox 左右border+左右padding+contentwidth 就是内容盒子 所设宽度=内容宽度
  - content-box 默认值 设置元素宽占100px 内容区会有100px的宽度 任何边框和内边距的宽度都会增加到最后绘制出来的元素宽度中

- ie盒模型 borderbox width=content+padding+border 任何内边距和边框都在设定的宽度和高度内绘制 当已经设置了width height那就是内容的宽度 border和padding不包含在里面 box-sizing 可以设置用那种盒模型 box-sizing:content-box || border-box 所设宽度=内容宽度+内边距+边框 设置的边框和内边距时包含在width和height中，内容的实际宽度就是width-（border+padding) 全局设置boder-box
- box-sizing:content-box;在宽度和高度之外绘制元素的内边距和边框。 box-sizing:border-box;通过从已设定的宽度和高度分别减去边框和内边距才能得到内容的宽度和高度. html { box-sizing:border-box; } ，：：before,*::after { box-sizing:inherit } 选择器*无法覆盖到伪元素 :befor, :after 分别设置 box-sizing：inherit 使元素尊重其父元素box-sizing规则

2. css选择器分类 css优先级

- css选择器
    - id选择器 id="a"
    - 类选择器 class="b c"
    - 标签选择器 body{} div.b.c{} ul{} li{}
    - 全局选择器 *{} 复杂:
    - 组合选择器 组合选择器 .head .head_logo 每个盒子的类名
    - 后代选择器 #head .nav ul li 从父元素到子孙集
    - 群组选择器 div, span, img {} 具有一样的标签
    - 继承选择器 继承父类
    - 伪类选择器 a标签 a:hover{}
    - 子选择器 div>p
    - css相邻兄弟选择器 h1+p ！ important>行内样式>id选择器>类选择器>标签选择器>通配符*>继承>浏览器默认属性 div { width:100px; height:100px; } #a { color:red; } div.b.c { color:green !important;

      ```
      }
      ```

      我

3. 居中的几种方式 口述
4. flex弹性布局 flex的属性
5. 块元素和行内元素
6. BFC
7. es6 的promise 原理 具体
8. es6的属性 了解哪些
9. this

下午自测： 表达

找出数组出现最多的元素 并标记下标 次数

数组去重 输出一个新的数组

二维数组的查找

替换空格

return str.replace(/\s/g,'%20')

前端面试必备技巧

一面/二面

- 准备要充分
- 知识系统化
- 沟通要简洁
- 内心要诚实
- 态度要谦虚
- 回答要灵活

页面布局 CSS盒模型 DOM事件 HTTP协议 面向对象 原型链 算法 通信 安全 算法

页面布局 三栏布局 两栏布局

1. 假设高度已知，写出三栏布局，其中左栏，右栏宽度各为300px - 中间自适应

- 多种方法
- 延伸 ：每个方案优缺点、不定高度 清楚高度 兼容性 浮动布局 float:left float:right center 绝对定位 left:l0 w300 centerl300 r300 right r0 w300 flex left center right center flex:1 外 display:flex; table-ceil table table-cell 网格布局grid display:grid gird-template-rows:300px grid-template-columns:300px auto 300px;

浮动定位：浮动 脱离文档流 不处理好 高度坍塌 有点：兼容性 绝对定位： 快捷 缺点：布局脱离文档流，子元素脱离文档流，导致方案有效性不好 可使用性很差 flex： table: grid：网格布局 栅格系统 960px设计模拟网格

flex 和 table有用

创建BFC 块级格式化上下文

总结

- 语义化要规范
- 页面布局理解深刻
- css基础知识扎实
- 思维灵活积极上进
- 代码书写规范

谈谈对css盒模型的认识

1. 基本概念： 标准模型+IE模型 margin padding border content
2. 标准模型和IE模型区别 计算高度宽度不同 宽和高计算方式的区别 标准盒模型宽度 content的width IE盒模型 包含 border padding
3. CSS如何设置这两种模型 box-sizing:content-box; 浏览器默认 box-sizing:border-box;
4. JS如何设置获取盒模型对应的宽和高

- dom.style.width/height 根据dom元素属性的style获取 dom节点内联样式的宽和高
- dom.currentStyle.width/height 通过浏览器渲染以后的即时运行的结果 ie浏览器支持
- window.getComputedStyle(dom).width/height 兼容性
- dom.getBoundingClientReact().width/height 计算一个元素的绝对位置，根据视窗口，左上角左顶点 left top width height拿到盒模型的宽和高

5. 实例 根据盒模型解释边距重叠 overflow:hidden
6. BFC 边距重叠解决方案 块级格式化上下文

- BFC基本概念：块级格式化上下文
- IFC 内联元素块级上下文
- BFC原理：BFC渲染规则
    - BFC 元素 垂直方向边距发生重叠
    - BFC 的区域不会与浮动元素的box发生重叠
    - BFC是一个独立的容器 外面元素不会影响里面元素
    - 计算BFC的高度，参与浮动元素的高度也会计算到里面
- 如何创建BFC
    - 根元素HTML 最大的BFC
    - float的值不为none
    - position:absolute/fixed
    - display:inline-block;table;table-caption
    - overflow:hidden/visible auto
- BFC使用场景 两栏布局的时候 阻止元素被浮动元素覆盖

BFC 块级化格式上下文 BFC是css布局的一个概念，独立的容器，里面的元素不会影响到外面的元素 如何创建 BFC float的值不为none overflow hidden auto visible display：inline-block table table-caption position：fixed /absolute BFC原理 BFC在垂直方向发生边距重叠 BFC 区域在边距重叠的元素不会与浮动元素区域 发生重叠 BFC 是一个独立的容器 容器里面的元素不会影响容器外面的元素 同时外面的元素也不会影响里面的元素 计算bfc高度的时候 浮动元素也会倍计算到

BFC发生的场景 清除浮动 两栏布局 阻止元素被浮动元素覆盖 自适应两栏布局 当左宽度固定 右高度撑开 设置 BFC

DOM事件类

1. 基本概念： DOM事件的级别 请用DOM2 DOM0 element.onclick=function(){} DOM1 没有设计和事件相关的 DOM2 element.addEventListener('click',function(){}, false)冒泡 捕获 DOM3 element.addEventListener('keyup',function(){},false)增加了dom 鼠标事件
2. 了解DOM事件模型是什么 事件冒泡 事件捕获

- 事件捕获 从当前元素从上往下

- 事件冒泡 从当前元素目标元素从下往上

3. DOM事件流 前端事件流 从页面接受事件的顺序 浏览器与用户交互的过程中 怎么传递 响应的

- 事件捕获
- 目标阶段 事件通过捕获到达目标元素
- 事件冒泡 从目标元素上传到window阶段

4. 描述DOM事件捕获的流程 window document html body ...标签 父级元素 子集元素 目标元素
5. Event对象的常见应用

- event.preventDefault() 阻止默认行为 function stopDefault() { if(e && e.preventDefault) { e.preventDefault() } else { //阻止函数企默认的动作的行为 window.event.returnValue = false; } }
- event.stopPropagation() 阻止冒泡 function stopBubble() { if(e && e.stopPropagation) { e.stopPropagation() //非ie浏览器 } else { window.event.cancleBubble=true //ie浏览器 } }
- event.stopImmediatePropagation() 按钮绑定 两个click事件 a,b 优先级 a点击 执行a
- event.currentTarget 当前绑定的事件
- event.target 事件委托 直接在父级元素上绑定 当前被点击的元素

6. 自定义事件 var eve = new Event('custome') ev.addEventListener('custome',function() { console.log('custome') })

ev.dispatchEvent(eve)

CustomEvent 参数

捕获流程 响应 window document html body 目标元素 自定义 自动触发 window.addeventListener document.addvent document.documentElement document.body ev.addEventListener setTimeout(function() { ev.addEventListener('test',function() { console.log('test captrue'); }) },1000)

HTTP协议类

1. HTTP协议的主要特点

- 无连接 连接一次就会断掉
- 无状态 客户端和服务端是两种状态 http负责客户端发起请求和服务端返回结果 断开后 再次链接是没办法区分客户的状态
- 简单快速 资源服务是固定的 url固定的
- 灵活 通过不用协议传输数据

2. HTTP报文的组成部分

- 请求报文 request 输入一个url地址 就是请求报文
    - 请求行 http方法 页面地址 http协议 http版本
    - 请求头 告诉服务端需要附加的地址 keyvalue
    - 空行 告诉服务端 下一个不再是请求投 当作请求体
    - 请求体 请求数据
- 响应报文 response
    - 状态行 http版本协议 状态码 状态消息三部分组成
    - 响应头 客户端要添加的消息
    - 空行 告诉客户端不是响应头

    - 响应体 服务端返回给客户端的文本

3. HTTP方法

- GET 获取资源
- POST 传输资源
- PUT 更新资源
- DELETE 删除资源
- HEAD 获取报文首部

4. POST/GET方法

5. GET在浏览器回退是无害的，post再次提交请求

6. GET产生的URL地址可以被收藏，POST不可以

7. GET请求会被浏览器主动缓存，而POST不会，除非手动设置

8. GET支持url编码，而POST支持多种编码方式

9. GET请求参数完整保留在浏览器历史纪录里，而POST的参数不会被保留在浏览器

10. GET请求在url传输的参数是有长度限制的2kb 拼接的url不要过长，而POST不会

11. 对参数的数据类型 GET只接受ASCII字符，而post没有限制

12. GET比POST更不安全，参数直接暴露在URL，所以不能用来传输敏感信息

13. GET参数通过URL传递，POST放在Request body中

14. HTTP状态码

- 1xx 指示信息 表示请求已接受，继续处理
- 2xx 成功 表示请求已成功被接受 200 206
    - 200 客户端请求成功
    - 206 客户端发送了一个待遇Range头的GET请求，服务器完成了它
- 3xx 重定向 要完成请求必须进行更进一步的操作
    - 301 所请求的页面已经转移至新的URL
    - 302 所有请求页面已经转移到临时的url
    - 304 缓存 服务器告诉客户端 浏览器已经缓存了 直接在浏览器中取 网页上次请求没有更新 节省宽带和开销
- 4xx 客户端错误 请求右语法错误 或者请求无法实现
    - 400 客户端请求有语法错误 服务器不理解
    - 401 请求未授权 用户没有权限，这个状态码必须和WWW-Authenticate报头域一起使用
    - 403 对访问请求页面 资源被禁止访问
    - 404 请求资源不存在 输入错误的网址
- 5xx 服务器错误 服务器未能实现合法的请求
    - 500 服务器产生不可预期的错误，无法完成请求
    - 503 服务器目前无法完成 服务器临时挂载或者停机维护

6. 什么是持久链接

- HTTP协议采用 请求-应答模式， 普通模式非keep-alive模式时，每次请求和应答 客户/服务器端都要建立一个新连接，完成后立即断开，完成后立即断开
- 当使用keep-alive模式，持久连接链接重用 keep-alive功能使客户端的连接持续有效，当服务器后继请求时，keep-alive避免重新建立连接 http1.1 http1.0
- http 超文本传输协议，是一种客户端请求服务端应答的一种标准tcp，是浏览器更加搞笑，网络传输减少
- https 是以安全为目标的http通道，安全版本的http，通过了ssl安全加密

http1.0 http1.1 http2.0的区别

- 支持长连接http1.0 需要告诉服务器建立一个keep-alive长连接 http1.1支持keep-alive
- 洁身宽带 http1.1 支持持发送一个header信息
- host域 设置虚拟站点 多个虚拟站点共享一个ip终端 http1.0没有host域
- http2.0是使用二进制文本传输数据 而http1 是文本格式 二进制在协议的解析和扩展更好
- 多路复用 一个连接个以处理多个请求
- 数据压缩
- 服务器推送 客户端发起请求 服务器端将一些重要的资源推送给客户端，避免客户端再次建立连接 发起请求 7。 什么是管线化 持久连接 请求1->响应1->请求2->响应2->请求3->响应3 请求一次响应一次 变化 请求1->请求2->请求3->响应1->响应2->响应3 将所有请求打包 一次性发送 然后一次性响应请求 管线化

原型链

1. 创建对象有几种方法

- 对象字面量 默认对象的原型链指向object var o1={name:'o1'}; var o11 = new Object({name:'o11'});
- 显示构造函数创建对象 var M = function() {this.name='o2'} var o2 = new M()
- 创建对象的方法函数 var P={name:'o3'}; var o3 = Object.create(P)

2. 原型、构造函数、实例、原型链

- 原型：**proto**
- 构造函数 凡是通过new操作一个函数 任何函数被new使用了 就是构造函数 可以使用new运算符生成new实例 构造函数也是函数 任何函数用new 变成构造函数
    - 函数都有prototype属性 prototype就是原型对象 原型对象有一个构造器 指向声明的函数 被哪个函数给引用了
- 实例： 只要是一个对象就是实例
- 原型链： 从实例对象 往上找 构造这个实例相关的对象 然后这个相关的对象 往上找 一直找到object object是整个原型链的顶端
  Object.prototype.**proto** = null M.__proto=function.prototype

3. instanceof 原理 实例对象 **proto** 引用的构造函数prototype 判断实例对象的属性 和构造函数 引用是否引用同一个对象 o3; M {name: "o3"} o3 instanceof M; true o3 instanceof Object; true o3.**proto**===M.prototype; true M.prototype.**proto**===Object; false M.prototype.**proto**===Object.prototype; true o3.**proto**.constructor===M true o3.**proto**.constructor===Object; false
4. new运算符

- new 新对象被创建 继承仔 foo.prototype 构造函数的原型对象
- 构造函数foo被执行，执行的时候 this上下文指定新实例 new foo = new foo() 不传递参数
- 构造函数返回一个新对象，这个对象取代整个new出来的结果，如果构造函数没有返回对象，那么new出来的结果为步骤1的

面向对象

1. 类与实例

- 类的声明 //类的声明 // 传统的构造函数 // ES6的class function Animal() { this.name = 'name' } class Animal2 { constructor() { this.name = name } }
- 生成实例 //简写实例化 console.log(new Animal(),new Animal2());

2. 类与继承

- 如何实现继承
- 继承的几种方式

通信

1. 什么是同源策略及限制 同源策略限制从一个源加载的文本或脚本如何来自另一个源的资源进行交互 协议+端口号+域名 要相同 限制

- cookie localstorage和 indexDB无法读取
- DOM无法获得
- Ajax请求不能发送

2. 前后端如何通信

- Ajax
- WebSocket
- CORS 支持跨域通信 和 同源

3. 如何创建Ajax

- XMLHttpRequest对象工作流程
- 兼容性处理
- 事件触发条件
- 事件的触发顺序 function getData(url) { let xhr = new XMLHttpRequest() xhr.open('get', url, true) xhr.send() xhr.onreadyStatechange = function() { if(xhr.readyState ===4 && xhr.status ===200) { console.log(xhr.responseText) } }

} Promise(getData(url)).resolve(data=>data) 状态码 0 未初始化 还没有调用send方法 1 载入 已调用send方法 正发送请求 2 载入完成 已经发送请求 3 解析响应内容 4 完成 响应内容解析完成 客户端可以调用了 4. 跨域通信的几种方式

- JSONP 利用script标签的异步加载来实现 src不受同源策略限制的影响，可以请求第三方的资源
- 去创建一个script标签
- 利用script标签 src属性设置接口地址
- 接口参数必须自定义一个函数名，后台返回数据
- 通过定义函数名去接受返回的数据

- Hash
- postMessage H5新特性 window.postMessage() window.postMessage(data,url) h5 API event.origin event.source
- WebSocket
- CORS 服务器设置CORS Ajax变种 服务器设置Access-Control-Allow-Origin Http响应头 浏览器可以发起请求 fetch(url,{ method:'get' }).then(res).catch(err)
- document.domain 基础域名相同 子域名不同
- window.name

安全

1. CSRF

- CSRF 跨站请求伪造 Cross-site request forgery
- 攻击原理 通过伪装来自受信任用户的请求 引诱用户点击 网站接口有漏洞 用户已经登录
- 防御措施
    - 加Token验证
    - Referer验证 判断页面是不是站点的页面
    - 隐藏令牌 颖仓Token放在Http头部
    - 通过验证码

2. XSS

- XSS cross-site scripting 跨域脚本攻击
- 原理 往web页面注入恶意的html和js代码 向页面注入脚本 在哇昂站论坛放置一个看似安全的连接，窃取cookie的用户信息
- 防御措施
    - 尽量采用POST提交表单而不采用get方式
    - 避免cookie泄露用户的隐式 XSS 获取信息 不需要代码包 CSRF 需要

算法

1. 排序
2. 堆栈、 队列、链表
3. 递归
4. 波兰式和逆波兰式

高频题

复习1

1. 如何判断this指向 this的指向：谁调用它，this就指向谁

- 全局环境中的this

    - 浏览器环境下：在全局执行环境中，任何函数外部，this都指向全局对象window let a=1 //此时this指向是全局window a未定义 console.log(this.a);
    - node环境下 this是空对象 {}
- 是否是new绑定
    - 构造函数的值返回的不是function，object， 那this的指向这个新对象 function Person(name) { this.name=name } let person = new Person('zzh') console.log(person.name); console.log(this.Person);
    - 构造函数返回的是function或object this指向的是返回的对象 function Super(age) { this.age=age; let obj ={a:'2'}; return obj; } let s = new Super('21'); console.log(s.age); console.log(this.Super);
    - new 的原理 function new(fn) { let obj = {}; obj.**proto** = fn.prototype; let context = fn.call(obj); if(context && typeof === 'object' || typeof(context)=== 'function') { return context; } return obj; }
- 函数作为对象的方法调用时，this就会指向该对象
- 函数通过call apply bind绑定 显示绑定 this绑定的就是顶顶的这个对象
- 箭头函数 箭头函数没有自己的this 继承外层上下文绑定的this

2. js原始类型有哪几种 null是对象吗 原始数据类型和复杂数据类型有什么区别 原始类型 ： Boolean String Number Undefined Null Symbol BigInt 复杂数据类型： Object Null 不是一个对象 typeof null === object 错误的判断成了onject

- 基本数据类型和复杂数据类型的区别
    - 内存分配不同 基本数据类型 存储在栈中 复杂数据类型存储在堆中 栈中存储的变量 指向堆引用的内存
    - 访问机制不同 基本数据类型 按值进行访问 复杂数据类型 按找引用访问 根据地质去获取值
    - 复制变量时不同 a=b 基本数据类型 a=b 将b的副本复制给a 当a值发生改变的时候 b不好改变 复杂数据类型 a=b 就是说明 a\b都引用同一个内存地址 当a发生改变时 b也会发生改变 let b = { age:10 } let a=b a.age=20 console.log(b) age:20
    - 参数传递不同 函数传参都是按值传递 基本数据类型 拷贝的是值 复杂数据类型 拷贝的是引用地址

3. 说说你对HTML5语义化的理解 HTML5语义化：合理正确的使用语义化标签来创建页面结构，如header，footer，nav 可以直观地知道这个标签的作用 见名知意的效果，而不失滥用div

- 语义化的优点
    - 代码结构清晰 易于阅读，利于开发和维护
    - 方便其他设备如屏幕阅读器 根据语义化渲染网页
    - 有利于seo搜索引擎优化，搜索引擎爬虫会根据不同的标签来赋予权重

4. 如何让(a==1 && a==2 && a==3)的值为true a==1 && a==2 && a==3 要想为true的话 那么a==1就是复杂数据类型 object

- 根据Object.definProperty数据劫持 let i=1; Object.defineProperty(window,'a', { get:function() { return i++; } }) console.log(a==1 && b==2 && c==3); true
- 利用proxy let a = new Proxy({},{ i=1; get:function() { return ()=>this.i++; } }) console.log(a==1 && b==2 && c==3); true

5. 防抖debounce函数的作用是什么？ 有哪些应用场景，请实现一个防抖函数 防抖函数就是控制函数在一定时间内执行的次数，意味着n秒执行一次，N秒内再次被触发 重新计算延时的时间 应用场景：

- 搜索框查询
- 表单验证

- 按钮提交事件
- 浏览器缩放，resize事件等

function debounce(fn,wait) { var timeout; return function() { var context = this; var args= arguments; clearTimeout(timeout); timeout=setTimeout(()=>{ fn.apply(context,args); },wait) } } 6. 节流throttle函数的作用是什么？ 有哪些应用场景，请实现一个节流函数 规定一个单位时间；在这个单位时间内只能触发执行一次 若执行多次的话，只能有一次生效

- 按钮点击事件
- 拖拽事件
- onScroll
- 计算鼠标移动的距离 mousemove function throttle(fn,wait) { vat timeout=null; var prevous=0; return function() { var context=this; var args=arguments; if(!timeout) { timeout=setTimeout(()=>{ timeout=null; fn.apply(context,args); },wait); } } }

7. 说说你对JS执行上下文栈和作用域链的理解

JS执行上下文

执行上下文就是 js代码被解析和杯执行时所在的抽象环境，js任何代码都是在上下文中运行

- 执行上下文分为；
  - 全局执行上下文
  - 函数执行上下文
  - eval函数执行上下文
- 执行上下文创建
  - 创建变量 初始化函数的arguments
  - 创建作用域链
  - 确定this 绑定this
- 作用域 由当前环境和上一层环境一系列的变量和对象组成 当先执行环境里有权访问的变量和函数是有序的，那么作用域变量只能被向上访问 由当前环境和上一层环境的一系列变量和对象组成 内部一级级有序的向上访问变量和对象
- 作用域分类
  - 全局作用域
  - 函数作用域
  - 块级作用域
- 作用 在当前执行环境 变量和对象的 访问是有序的 只能向上访问 不能向下访问 最终对象是widow 改变作用域 可以用 with try中的catch
- JS执行上下文栈 js执行栈 执行栈 叫做调用栈 就具有后进先出的结构，用于存储代码执行期间创建的所有的执行上下文 function func3() { console.log('func3') } function func2() { func3() } function func1() { func2() } func1(); 全局上下文首先入栈 fun1 fun2 fun3 fun3执行完 fun2 执行完 fun1 执行完
- 作用域链 当前作用域链向上一级级有序的查找摸个变量，访问全局没有 就放弃 这就是作用域链

8. 什么是BFC？BFC的布局规则是什么？ 如何创建BFC

- BFC块级格式话上下文 是css的一个容器 独立的容器 外部的元素不好影响内部 内部也不会影响外部
- BFC的布局规则
  - BFC 盒子垂直排列

    - BFC 两个盒子的垂直距离由margin属性决定 ， 边距重叠 符合合并的margin合并后是使用大的margin
    - BFC 浮动的时候
    - BFC的区域不会和float box重叠
    - BFC是一个隔离独立的容器 容器里的子元素不会影响到外面的元素
    - 计算BFC的高度时 浮动元素耶参与计算
- BFC创建
    - display:inline-block display:table table-ceil table-capation
    - position:absolute
    - 浮动元素float不为none
    - 根元素 html时最大的BFC
    - overflow的值不为visiable
- BFC应用
    - 防止margin重叠 垂直方向上的盒子margin overflow:auto;
    - 清除内部浮动 外部容器 创建BFC display：inline-block
    - 自适应多栏布局 三栏布局 两栏布局 overflow:hidden;

9. let、const、var的区别是什么

- let const 是块级作用域 而var不是
- var允许重复声明 而let和const不允许
- let/const 定义的变量不会出现变量提升 而var会
- const设置的是一个常量 必须设置初始值 这个常量不可改变
- 顶级作用域中var声明的变量挂载window上
- let/const有暂时性死区的问题 变量在let和const未定义之前使用是会抛出错误的

10. 深拷贝和浅拷贝的区别是什么？ 如何实现一个深拷贝 深拷贝和浅拷贝是针对复杂数据类型来说的

- 深拷贝 复制变量值 深拷贝的对象和源对象是完全隔离 深拷贝是拷贝多级
- 浅拷贝 对源对象值的引用 值引用一级 源对象值发生改变 浅拷贝的值也会发生改变 引用 使用 for in Object.assign 扩展运算符... Array.prototype.slice() Array.prototype.concat()
- 浅拷贝 let obj = { name: 'Yvette', age: 18, hobbies: ['reading', 'photography'] } let obj2 = Object.assign({}, obj); let obj3 = {...obj};

obj.name = 'Jack'; obj.hobbies.push('coding'); console.log(obj);//{ name: 'Jack', age: 18,hobbies: ['reading', 'photography', 'coding'] } console.log(obj2);//{ name: 'Yvette', age: 18,hobbies: ['reading', 'photography', 'coding'] } console.log(obj3);//{ name: 'Yvette', age: 18,hobbies: ['reading', 'photography', 'coding'] }

浅拷贝 他只拷贝一层 就是第一层的属性 后面只是添加的 Object.assign({},obj) let obj3={...obj}; 当第一层是复杂数据类型的时候 let obj = { name: 'Yvette', age: 18, hobbies: ['reading', 'photography'] } let newObj={} for(let key in obj) { newObj[key]=obj[key] } console.log(newObj) obj.age=20; obj.hobbies.pop() console.log(newObj)

- 深拷贝 实现最简单的 JSON.parse(JSON.stringify(obj)）) let obj = { name: 'Yvette', age: 18, hobbies: ['reading', 'photography'] } let newObj = JSON.parse(JSON.stringify(obj)) obj.hobbies.push("coding") console.log(newObj) 互不影响 源对象和现在的对象
- JSON.parse(JSON.stringify(obj)) 对象的属性值是函数的时候无法获取 原型链上的属性无法获取 不能正确的处理Date类型 不能处理RegExp 会忽略Symbol 会忽略undefined function deepclone(obj) { if(obj instanceof RegExp) return new RegExp(obj) if(obj instanceof Date) return new Date(obj) if(obj== null && typeof obj !== 'object) { return obj; } let t = new obj.constructor(); for(let key in obj) {

if(obj.hasOwnProperty(key)) { t[key]=deepclone(obj[key]) } } } 循环递归引用 function deepClone(obj, hash=new WeakMap()) { if(obj instanceof RegExp) return new RegExp(obj); if(obj instanceof Date) return new Date(obj); if(obj==null || typeof obj !== 'object') { return obj; } if(hash.has(obj)) { return hash.get(obj); } let t = new obj.constructor(); hash.set(obj,t); for(let key in obj) { if(obj.hasOwnProperty(key)) { if(obj[key] && typeof obj[key]==='object') { t[key]=deepClone(obj[key],hash); }else { t[key]=obj[key]; } } } }

11. 什么事XSS攻击，XSS攻击可以分为哪几类，我们如何防范XSS攻击 XSS 跨站脚本攻击 脚本注入攻击 代码注入攻击 在网站恶意注入乱的html标签和js代码 当被攻击者登录网站就会执行这些恶意代码 脚本读取cookie session tokens 在评论区放一个看似安全的连接 防御措施： 采用post提交表单 而不采用get提交表单 采用cookies验证

- XSS分类
    - 反射性xss
    - DOM型XSS
    - 存储行XSS

12. 如何隐藏页面的某个元素

- 完全隐藏
    - display:none 不占据空间
    - hidden属性 不占据空间
- 视觉上的隐藏
    - 通过position设置为absolute和fixed 设置top left 将其移出可视区域 可视区域不占位 position:absolute; left:-99999px;
    - 利用transform
        - 缩放 transform：scale(0)
        - 移动 tranlateX,translateY 占据空间 transform:translateX(-99999px)
        - 旋转 transform:rotateY(90deg)
    - 设置大小为0 宽高为0 字体大小为0 width:0; hieght:0; font-size:0; overflow:hidden;//超出则隐藏
    - 设置透明度为0 opacity:0;
    - visibility属性 visibility：hidden；
    - 层级覆盖 z-index position:relative; z-index:-999;
    - clip-path 裁剪 clip-path:polygon(0 0,00,00,00);
    - 语意上的覆盖 读屏软件不可读

通过js移除某个元素

总结: display:none; hidden overflow:hidden; opacity:0; tranform:scale(0); position:absolute; left:-999999; 层级覆盖 position:relative z-index:-999 visibility:hidden; clip-path：plogn（00，00，00，00）； aria-hidden=true js移除元素

13. 浏览器事件代理机制的原理是什么 前段时间刘：指的是从页面接受事件的顺序，分为三个阶段，事件捕获阶段，目标阶段，冒泡阶段 aadEventListener true的话事件捕获 false事件冒泡

- 事件代理机制 事件委托 原理： 不在目标元素事件上设置见她挺，而在父元素上设置监听，父元素监听子元素上的事件 通过事件冒泡 父元素可以监听到子元素上时间的触发 最经典的就是ul li 在ul上设置监听通过事件冒泡 ul上监听到li时间上的触发 来做出响应
- 好处
    - 事件频繁触发 页面性能不好 可以增加提高页面性能

    - 事件代理 子元素动态增加 适合动态元素的绑定
    - 允许一个事件注册多个监听
- addEventListener 接受三个参数 addEventListener('处理的时间名',{},true/false);

14. setTimeout倒计时为什么会出现误差 setInterval setTimeout 只能保证延时或者间隔不小于设定的事件，实际上是把回调添加到宏任务队列中，如果线程任务没有完成，必须等待

- JS的运行机制
    - 所有同步任务都在主线程上完成对
    - 住县城上还存在任务队列
    - 同步任务执行完毕，执行任务队列的代码
    - 主线程不断同步 setTimeout(()=>{callback},1000) 表示1s后将callback事件放到宏任务队列中，当1s时间到达时，如果主线程的任务还在执行，那么必须等待，另外callback执行也需要时间，所有setTimeout是有时间间隔的 只能保证延时不小于设置的时间
- 如何减少误差
    - 减少执行多次的setTimeout，如倒计时功能

15. 什么是闭包？闭包的作用是什么 闭包 就是有权访问另一个函数作用域的变量和值的函数

- 闭包：内部函数有权访问外部函数定义的变量和值，创建闭包 就是在一个函数里创建另一个函数

- 创建一个闭包 function foo() { var a=2; return function fn() { console.log(a) } } let func = foo() func();//输出2

- 闭包的作用

    - 阻止被回收 能够访问函数定义时所在词法作用域
    - 私有化变量 function person () { let age=20; return { getAge:function() { return age; } } } let obj = person() console.log(obj.getAge()); //20
    - 模拟块级作用域 var a=[] for(var i=0;i<10;i++) { a[i]=(function(j) { return function() { console,log(j); } })(i) } a6
    - 创建模块 function person() { let name='yu' let age=20 function getName() { console.log(name); } function getAge() { console.log(age); } return { getName; getAge; } } let info = Person(); info.getName();//yu

- 闭包条件 函数嵌套 子函数访问父函数

- 闭包的缺点 闭包会导致大量很多大量的常驻内存 使用不当会导致内存泄漏

16. 实现Promise.all方法 promise它是一个用来传递异步操作的信息对象 异步处理 promise.all()里面状态都在改变 那就回输出 返回的是一个数组 let p= Promise.all([p1,p2,p3])

- 当p1,p2,p3的状态都变成fullfilled时候 才会编程fullfileed resolve返回一个成功的数组
- 当p1,p2,p3有一个rejected 那p就是rejected 返回的是一个失败的回调 error

promise.all 的特点

- promise.all返回的是一个实例
    - 如果传入的是一个空的可迭代对象 那么Promise.all同步返回
    - 如果参数没有promise promise异步返回一个已完成的promise

    - 其他 返回一个pending的状态 代码promise.all() promise.race() promise.finally()

17. 异步加载js脚本的方法有哪些

-

defer 要等到整个页面渲染完成之后才会渲染 async 一旦下载完成 渲染就好终端 执行这个脚本再渲染 多个 defer 按顺序 async不一定按照顺序

- 动态创建script标签 设置src let script = createElement("script") script.src="#" document.body.append(script); 设置src不会下载 添加到文档中才会开始下载
- 异步加载XHR js let xhr = XTMLHttprequest(); xhr.open('get',url,true); xhr.send() xhr.onreadyStatechange=function() { if(xhr.status==200 && xhr.reaystate==4) { rhr.responseText; cs } }

18. 请实现一个flattenDeep函数，把嵌套的数组扁平化
19. 可迭代对象有什么特点
20. 实现promise.race方法
21. JSONP原理及简单实现
22. 实现一个数组驱动的方法
23. 清除浮动的方法有哪些
24. 编写一个通用的科利华函数 currying
25. 原型链继承的基本思路是什么？有什么优缺点
26. 借用构造函数和组合继承基本思路是什么？有什么优缺点
27. 原型式继承的基本思路，有什么优缺点
28. 寄生式继承的基本思路
29. 寄生组合式继承
30. ES6继承
31. 实现一个JSON.stringify
32. 实现一个JSON.parse
33. 实现一个观察者模式
34. 使用CSS让一个元素水平垂直剧中的方式
35. ES6模块和CommonJs模块有哪些差异

复习2

1. Vue的双向绑定说一下
2. 数组遍历的方法有哪些，es6有哪些，map和forEach的区别 好处有有点
3. js的 == 和 === 的区别
4. jq用过的方法
5. http协议了解哪些 说说常见的状态码
6. 协商缓存和强缓存说一下 对应的http的header头部是什么，如何使用
7. 前端跨域说一下， 同源和非同源说一下，jsop如何使用，cors？
8. webSocket如何使用，做聊天室的时候node服务端如何写
9. 懒加载和预加载在什么场景使用，用法是啥
10. ES6的 async和await的用法
11. ES6中const改变一个对象的属性是可疑的，是对是错，为什么
12. http缓存了解多少

13. 数组和链表的区别
14. jq的ajax方法 实现过程
15. flex布局
16. position：absolute 会脱离文档流码 设置margin会如何
17. css取消点击时间，哪个属性
18. z-index
19. Vue 版本 3.0 原理 生命周期
20. 两栏布局
21. 随机生成给定长度的字符串
22. ES6新增的方法
23. 提取url中search部分参数 https://www.aliexpress.com?a=1&b=2
24. 讲一下DOM和时间相关内容
25. 介绍一下存储
26. 介绍node node.js的流
27. 快排quickSort 时间复杂度 空间复杂度 最好 平均 最差

复习3

1. 导航菜单 hover变色
2. link标签的作用
3. Doctype的作用 meta作用
4. 移动端性能优化
5. lazyload如何实现
6. 点透问题
7. 前端安全 web安全
8. js原生模板引擎
9. repaint和reflow的区别
10. requirejs如何避免循环依赖
11. 盒模型
12. 实现布局 左边一张图片，右边一段文字 不是环绕
13. window.onload 和$(document).ready()的区别
14. form表单当前页面无刷新提交 target和iframe
15. margin坍塌?水平方向会不会
16. flex属性
17. 伪类和伪元素的区别
18. vue如何实现付子组件通信 以及费付子组件通信
19. 实现 bind call apply
20. 二分查找
21. n长的数组放入n+1个数 不能重复，找出哪个确实的数
22. 评价现在的前端
23. 用原生js实现复选框选择以及全选非全选功能
24. amd和cmd区别
25. vue特点 双向数据绑定是如何实现的 Object.defineProperty
26. 页面加载过程
27. 浏览器如何实现图片缓存

复习4

1. 介绍React特点
2. react对于项目选型的考虑 为啥用react (同Vue)
3. react声明周期
4. React的props和state区别
5. React如何实现父子组件的传值
6. React如何实现任意两个组件之间的传值
7. 介绍Redux MVC
8. Redux的action如果是一个异步操作 如何解决 redux-thunk
9. 一个整形数组，求任意三个元素为0的集合 不能重复 [-1,0,1,2] [-1,1,0] [0,0,0] [-1,-1,2]
10. 介绍箭头函数 和普通函数的区别
11. flex布局 align-items和align-content区别
12. 加入一个class里有箭头函数与function 在调用 this有什么区别
13. 递归有什么缺点
14. typeof判断结果 和instanceof区别
15. 怎么理解原型链 跟es6的class有什么区别 应用场景
16. Ajax过程 状态码
17. react和vue的异同
18. 虚拟dom和diff
19. tostring call isArray
20. 页面即在优化
21. 异步编程 generator async/await promise
22. dom获取元素的方式
23. rem em区别
24. 设计模式
25. 青蛙跳台阶

复习4

1. http和tcp的区别 及关系
2. 一个网址从输入到页面展示 经历了哪些流程
3. 网页渲染过程
4. js和java c++继承有哪些区别
5. 三次握手 四次挥手
6. nginx负载均衡
7. 变量提升
8. Vue声明钩子函数
9. Vue公共模块
10. vue如何获取路由的hash值
11. Vue为什么能过获取Vuex的值
12. 如何计算一个组件的渲染时间
13. http请求头
14. viewport的作用
15. 移动端适应

16. 前缀表达式
17. rem如何计算px
18. scroll滚动页面卡顿如何解决 scroll计算
19. event loop
20. keep-alive
21. cookie session localStorage sessionStorage
22. ul节点和最后一个节点交换 添加
23. http是如何连接的
24. git的相关命令
25. meta标签内的width设置

复习5

1. 前端存储方式
2. json对象删除一个键对
3. HTML5模板
4. restful的风格
5. 手写质因数
6. visibility:hidden 和display:none的区别
7. input中readonly和enabled的区别
8. 设计模式 手写单利模式 手写观察者模式的实现
9. event loop如何将时间推送到异步队列中的
10. js为什么不是多线程的 如果是多线程的会产生什么后果
11. prototype constructor proto
12. css选择器和优先级
13. 正则表达式的新特征 []内部的特殊字符需要转义码
14. 引入css的方式
15. http 出现304如何解决 出现场景
16. js引起渲染阻塞的解决方法
17. 手写tool-tip 封装完成的插件
18. script文件的请求和优化
19. HTML的新特征
20. 实现的动画方式
21. Vue的axios封装
22. xss和csrf
23. loder和plugin webpack配置
24. autocomplete表单自动完成
25. css预处理 stylus less sass 区别
26. box-sizing
27. 二叉树遍历

复习6

1. 小程序的运行机制
2. 手写cookie 设置一条过期

3. 网页兼容模式和正常模式
4. dns解析
5. 算法 寻找文本出现字多的字符串
6. mysql引擎及区别
7. 红黑树 B树
8. nodejs
9. http长连接
10. hybird app
11. 写一个组合模式
12. 算法写个判断是不是回文数的函数
13. 项目的前后端鉴权如何做的token token失效
14. 客户端开发和web开发有哪些区别 跟jquery
15. 为什么操作DOM性能不好 虚拟DOm 为什么
16. eventloop 说一下宏任务和微任务的区别
17. post和get区别 什么是幂等
18. http1 和 2的区别
19. 雪碧图
20. 判断数组是否连续的问题 [0 10] [8 10] [10 30] 返回true [0 10] [25 20] [8 20] 返回false
21. script标签为什么要放在后面 css引入为什么放在header里
22. background-image 和 img区别 哪个先加载 bg
23. meta标签的作用 两种属性
24. 块级元素和行内元素
25. position的参数
26. 页面整体div如何实现等比缩放

复习7

1. margin重叠是什么原因 如何解决 BFC
2. 作用域 如何查找值
3. 小程序与vue的区别
4. ES6新增特性
5. map set结构
6. 画一个三角形 均分原理
7. 画一个扇形
8. 动画实现 animation与requestAnimationFrame
9. 与请求Options fetch
10. web安全
11. 物理像素 逻辑像素 1px 2px
12. TCP与UDP区别
13. 二叉树遍历 先序遍历 中序遍历 后续遍历
14. 事件委托机制
15. keys values entries方法
16. http ssl协议 公钥密钥
17. vuerouter 原理
18. 模块化
19. ES6和node模块化本质区别

20. canvas 和 svg
21. 手写一个路由
22. flex布局和流式布局
23. 圣杯布局和双飞翼布局
24. 32位有符号整数反转
25. 不含重复字符的最长紫川
26. vuex
27. 什么叫组件 组件化是什么意思
28. JSON反序列化一个对象
29. ES6的reflect

复习8

1. ES6模块化机制到底是什么样的 每个模块import了多个模块 怎么样使得包更小
2. 怎么看用户是否连接到了服务器
3. 桶排序 原理 时空复杂度 使用场景
4. cookie怎么实现的 和session的区别
5. 手写 写一个alert函数 弹出n次对话框 每次弹出间隔m秒
6. 计算机网络5层协议 和 7层协议的理解
7. CDN原理 怎么负载均衡
8. 使用reduce实现一个map函数
9. Vue中nextTick实现原理 和 node process.nextTick区别
10. 不同子域如何共享js脚本
11. 按需加载

复习9

1. 实现倒计时功能 秒杀
2. es6 箭头函数的指向 和一般函数this指向的区别
3. 同源策略
4. retina屏幕的了解
5. flexible 移动端的布局
6. 轮播图优化
7. css布局的属性有哪些
8. boder的div 里面有图片 图片和下面的border有空隙 baseline
9. 函数调用的方式 区别
10. 异步编程
11. 回调地狱
12. linux linux命令
13. vue指令 v-if v-show
14. DOM树 两个节点 找出节点公共的父节点
15. 反转二叉树 算法
16. Vue和Reaact路由实现方式
17. mongodb
18. 手写代码 判断是否是浏览器环境

19. node 全局对象
20. tap是怎么回事 click

复习10

1. 动态规划是什么
2. 进程和线程的区别
3. restful api
4. typescript
5. css的 渐进增强 优雅降级
6. display有哪些
7. node线程 崩溃 如何知道
8. 堆排序 100万个大数据 找出前10大个数据
9. 画出tcp三次握手
10. websocket和 http 的区别 websocket
11. html css js分别实现一个动画效果
12. 复杂度 优化 非递归写法
13. 闭包内存泄露
14. http 缓存 跨域 请求头
15. webpack提取公共模块 作用 原理
16. 前端模块化工程化
17. 定位 定位基准
18. 堆 栈
19. 不通过http传输数据的方式
20. mvvm和mvc区别
21. 热加载原理 热更新
22. mvvm框架
23. document.domain
24. 了解贪心算法 分治吗 说说例子
25. 大文件续传
26. 长轮询机制
27. jwt机制 如何实现 刷新token时如何恢复之前终端的请求
28. eventbus 有时i为什么失效

复习11

1. 如何白屏优化
2. 数组添加元素 删除元素
3. http请求方法
4. delete filter
5. css实现直角梯形
6. 手写下拉刷新
7. vue版本
8. vue数据劫持
9. vue声明周期 组件传值 vuex

10. http https 什么事对称加密 什么非对称
11. 1-100 缺少一个值怎么找 却两个值怎么找
12. 前端路由的权限实现
13. 后端怎么做鉴权的
14. 每一次访问 KOA时怎么实现的
15. KOA中间件
16. CSRF 原因
17. KOA特性 特点 了解KOA怎么用
18. VUE组件封装
19. VUE里的computed、watch、data各适用的范围
20. VUE源码
21. 虚拟DOM Vnode的了解 什么阳的形式
22. ajax跨域有哪些方案

复习12

1. Vue源码 数据驱动层面的实现
2. 获取页面所有图片的src css部分获取 dom方法
3. element表格组件的实现
4. 前端监控
5. commonjs 和 es6 module 区别 webpack对模块怎么处理
6. webpack HMR 原理
7. 打包优化 怎么变快
8. egg.js 多进程通信机制和架构
9. 设计模式 随机洗牌算法
10. webkit
11. redis
12. vue的template时什么 模板引擎原理
13. vue和ng 和 react对比
14. npm查看已有包的命令
15. splice和slice区别
16. koa原理 koa-compose实现中间件
17. weakset wekmap set map
18. proxy 和 defineProperty 对比 双向绑定
19. 服务器渲染
20. promise并发
21. 注册登录 加密机制和原理
22. 正确的括号算法
23. 大数相加
24. 300ms 移动端延时 产生原因和解决方案

复习13

1. 进程通信方式
2. TCP/UDP 画三次握手/四次挥手

3. 七层网络结构/详细介绍每一层 每层之间是怎么交互的
4. 面向对象的概念 三个特点 分别说明什么意思
5. https原理
6. 虚拟dom，diff算法
7. vue的优点
8. es6用了什么 promise和Ajax的区别
9. webpack 了结果其他glup
10. http请求的过程 过程中的优化
11. 路由实现方式
12. 深拷贝 浅拷贝
13. es6新增
14. express中间件
15. express原理
16. 模块化规范 AMD CMD CommonJS
17. 跨域
18. 性能优化
19. web安全
20. vue-router 底层
21. express和koa
22. 域名划分
23. 网关切换
24. 子网掩码
25. https加密原理
26. css position
27. css 动画
28. vue声明周期钩子函数
29. vue的data为什么是return{}
30. 闭包
31. get和post请求的区别
32. options 使用

复习14

1. Charles Fildder原理
2. webpack多入口怎么配置
3. git命令
4. for in 遍历特点 枚举属性
5. 迭代器
6. 正则 前置断言
7. 数组字符串常用方法
8. es6
9. vuex的store
10. React DIff算法和传统diff算法有什么区别
11. css选择器的权重
12. 虚拟dom和传统dom区别
13. Node服务器

14. 本地web存储

复习15

1. koa源码说一说
2. koa源码怎么实现的
3. generator去模拟async
4. koa中ctx的request来源是什么 node request的来源是啥
5. node的cluster机制大概说一下
6. 事务ACID
7. 事务的隔离级别
8. http缓存的各种知识
9. ssl
10. react fiber
11. 进程和线程的区别
12. 怎么提高缓存的命中率
13. vue源码
14. js的运行机制 eventloop
15. vue的生命周期 created和mounted
16. vue router生命周期
17. computed和watch区别
18. 网页优化
19. css预处理框架
20. JSONP原理
21. Proxy原理 源码怎么实现的
22. 怎么样在事件捕获中触发事件
23. 前端路由有了解吗
24. 平衡二叉树

复习16

1. display的属性
2. position的属性 作用 使用
3. flex的属性
4. css引入样式有那几种方法 不考虑预处理和正常的三种方法 有没有其他的方法
5. css单位 每一种的使用
6. css选择器的优先级 有击中方法可以超过id选择器
7. 浏览器机制 DOM事件流
8. eventloop 宏任务和微任务
9. ajax前后端通信 除此之外 通信方式
10. 浏览器存储以及各自的区别 如何使用cookie和session机制
11. es6新特性
12. 箭头函数
13. 浏览器兼容
14. vue框架 使用路由做前端拦截的具体实现是什么

15. vue-router实现的击中方式
16. vuex有那些作用和使用场景 实现原理是什么
17. axios的使用 原理讲一下
18. 异步的解决方案 原理简单讲一下 async和await的使用
19. diff 脏检测 DOM比对
20. url输入到页面展示的过程
21. 前端性能优化
22. 浏览器的安全问题 如何解决

复习17

1. addEventListener是用来干嘛的
2. css没下载完会影响dom树吗
3. 1-N之间的素数 O(n)
4. 乱序输入一个数组 要求输出每个数字的概率相同
5. stopPropagation 包括捕获吗
6. 移动端适配方法
7. 再原型链上添加一个方法 让他每隔2秒就打印一次name的属性
8. 实现一个类似百度的搜索狂 然后点击后 下面出现一个div 我要怎么让我点到旁边的时候他消失
9. 实现一个九宫格布局 不知宽高

Vue专题

1. MVVM 顾名思义，MVVM就是model、view、viewModel model是模板层，view是视图层，view Mode1是桥梁，负责沟通view和model层 view层负责viewmodel数据和状态的展示，viewmodel底层设置数据绑定的监听，当viewmodel层数据发生裱花的时候，view试图自动更新，view层数据改变，viewmodel层自动同步
2. object.defineProperty 数据双向绑定的原理 数据劫持+发布者订阅者 实现一个数据监听其observer，利用object.defineProperty()重写get和set方法，然后值更新在set中，通知订阅者数据发生改变 编译的时候compile，深度遍历dom，让每个元素的节点的html模板元素发生改变都告诉订阅者 实现watch，用于连接observer和compile，使订阅者可以接收到数据状态的改变 手写object.defineProperty 遍历对象 对其属性进行劫持 const data= { name: '' } function say() { if(name===='ym') { console.log('ym') }else if(name='zly') { console.log('zly') }esle { console.log('lyf') } } Object.keys(data).forEach(function(keys) { Object.defineProperty(data,key, { emurable:true, configurable:true, get:function() { console.log('get') }, set:function() { console.log(`${newVal}`) say(newVal) } }) }) data.name='zzh' 核心就是通过数据劫持+发布者/订阅者模式 用object.defineProperty()去监听get/set方法 监听数据的操作，从而触发数据同步
3. eventBus 专车 就是 发布者和订阅者都引用专车，这个vue实例来完成订阅发布，emit发布 on订阅 eventBus解决了兄弟组件之间传值的问题，避免了了通过父组件转而传值的方法 将数据流单独抽离出来击中状态管理
4. 生命周期 new Vue vue实例化 init event 内部事件初始化 beforeCreate Obsever Data 监听数据的变化 created compile 编译模板 把data里边的数据和模型生成html模板 beforeMount 还没有生成HTML元素 mounted 已经挂载完成 html模板渲染完成 beforeUpdate vrtual dom updated beforeDestroy destroyed
5. 前端模块化 一个复杂的程序封装了多个块 ，每个块里面的内容是思密达，暴露出对外的接口，这就是前端模块化，前端模块化，可以降低代码耦合，提高服用，减少重复代码，使页面结构清晰，增加页面性能的提升。

Next.js

React专题

React 新特性

Nuxt.js

node专题

Koa

express

restful

typescript

ES6

webpack

glup

SSR

ts-axios

mysql mongodb

前端测试课 Jest TDD/BDD

源码解析

React源码深度解析

Vue源码解析

Nginx反向代理机制

Javascript设计模式系统讲解

Redis