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：

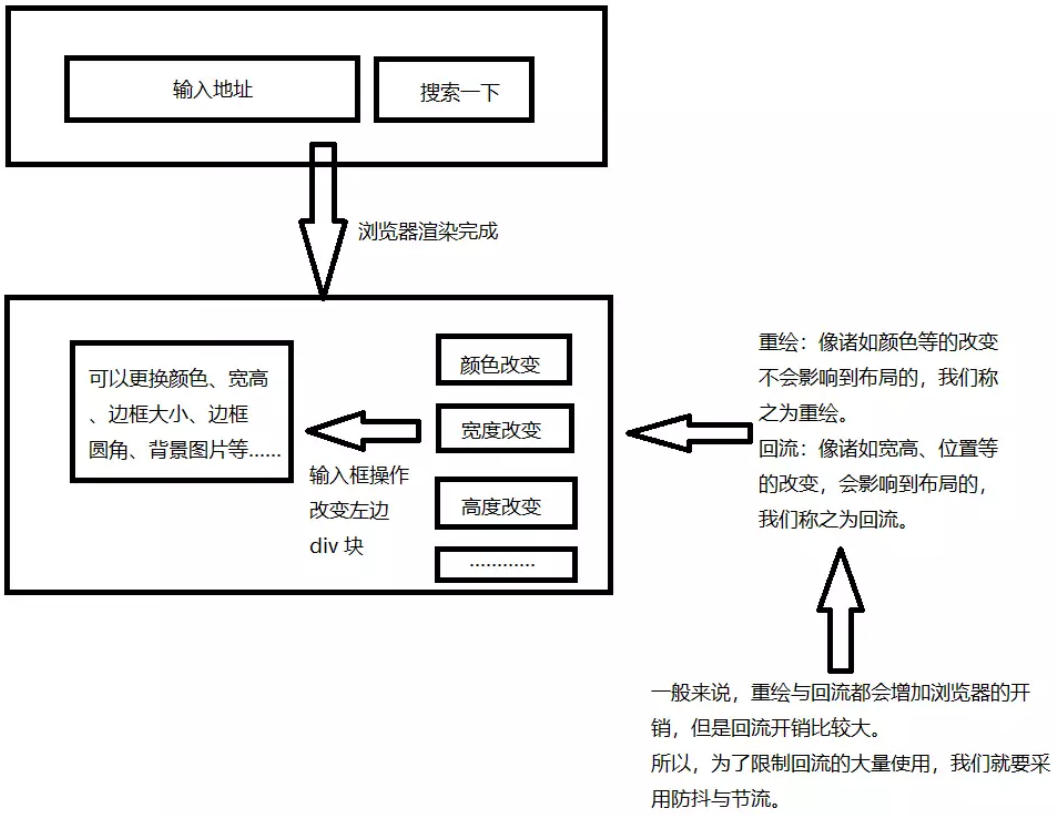
改变元素颜色 改变元素背景色 more ..... 回流(reflow)：又叫重排（layout）。当元素的尺寸、结构或者触发某些属性时，浏览器会重新渲染页面，称为回流。此时，浏览器需要重新经过计算，计算后还需要重新页面布局，因此是较重的操作。 常见的回流操作有：

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

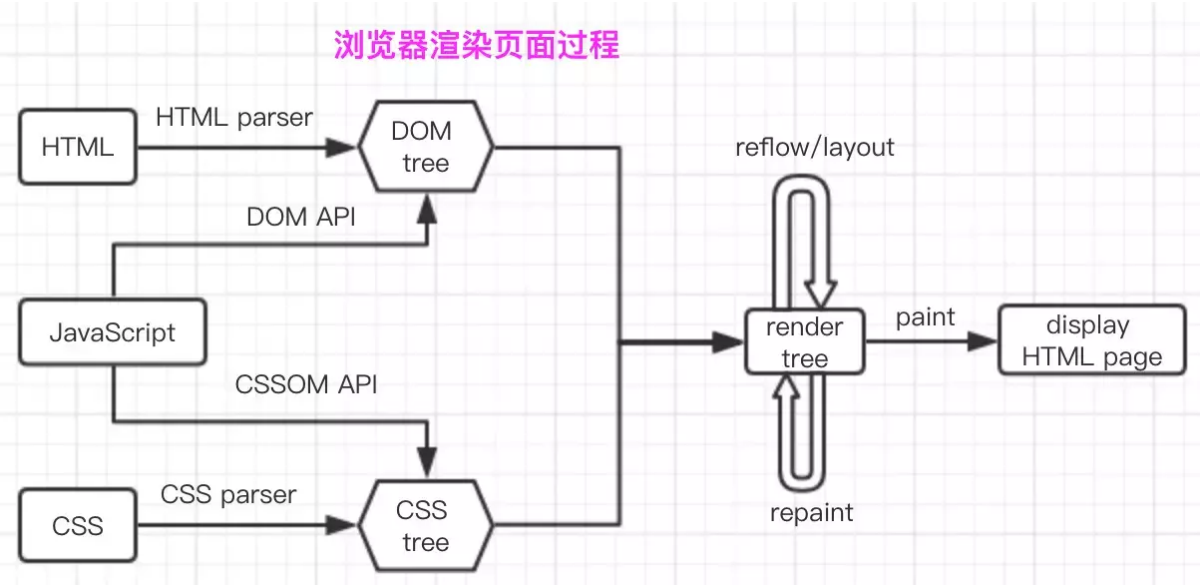
resolve=value(()=>{ if(state='pengding') { this.state='fullfilled' this.value=value } }) let reject=value(()=>{ if(state==='pengding') { this.state='rejected' this.reason=value } }) try() { fn(resolve,reject) } catch(e) { reject(e) }

} 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



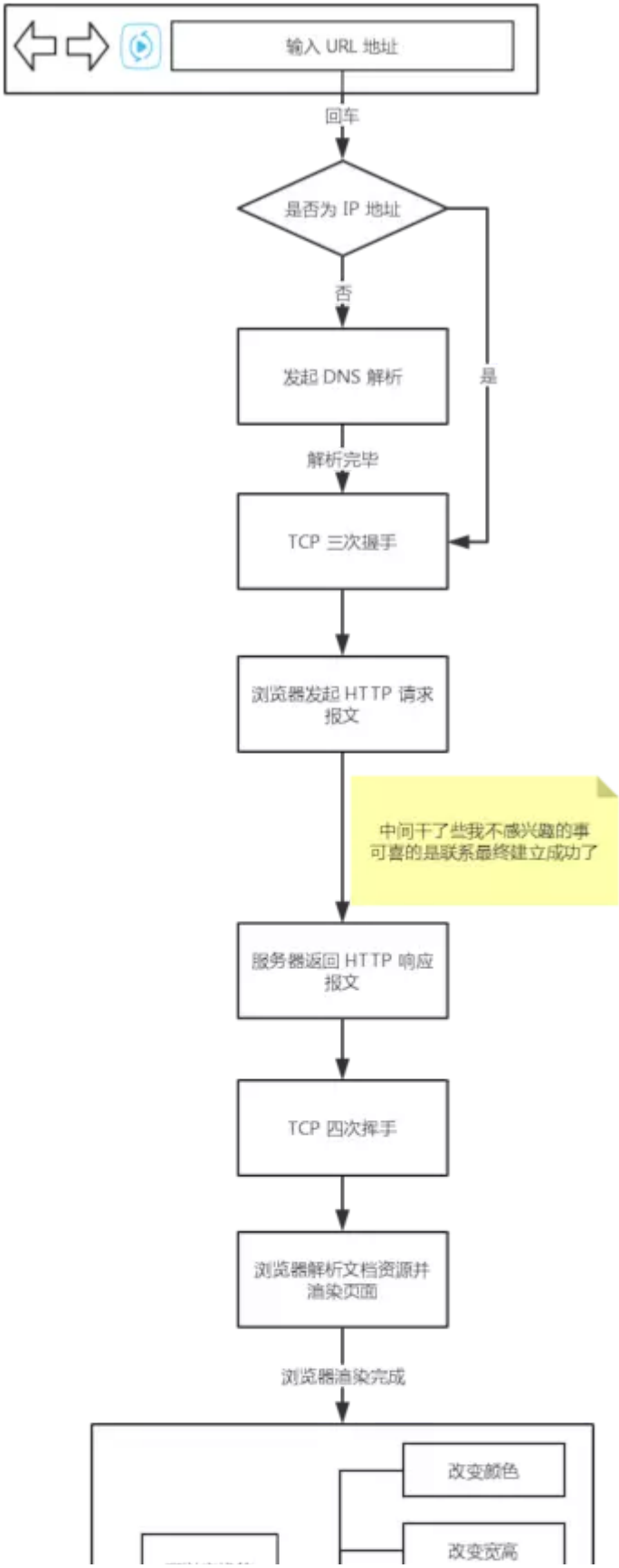
html -> DOM tree layout/reflow cssDOM ->CSStree -> render tree ---paint---> diplay HTML page js DOMAPI CSSDOM API repaint

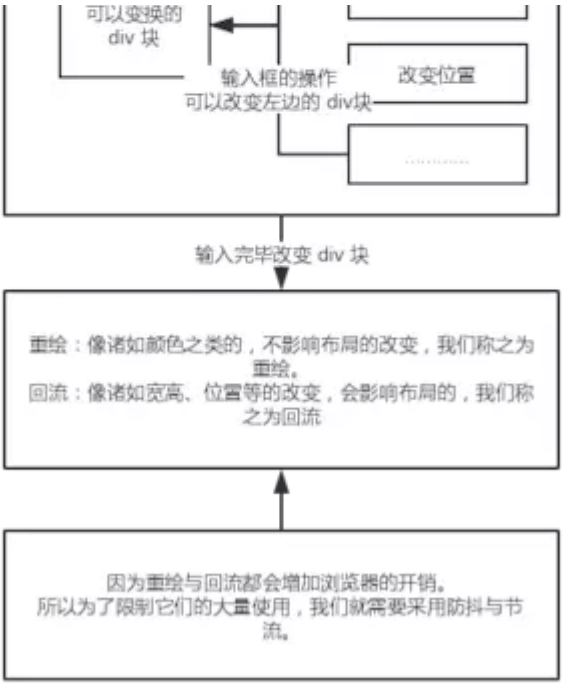
浏览器输入URL发生什么

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