## 浏览器的渲染流程

这个问题应该就是输入url到页面呈现问题的变种，只不过此时的侧重点是获取完数据之后进行的渲染流程。

第一步，HTML转换成DOM

第二步，CSS转换成浏览器可理解的styleSheets，然后计算DOM节点的样式

第三步，创建布居树，计算元素的布局信息

第四步，对布居树进行分层，构建分层树

第五步，为每个图层生产绘制列表，并将其提交到合成线程

第六步，合成线程将图层转化为图块，进而将图块转化成位图

第七步，合成线程发送绘制命令给浏览器

第八步，浏览器根据绘制命令生成页面，并显示到显示器上。

## 输入网址到实现的过程

1. DNS 解析:将域名解析成 IP 地址
2. TCP 连接：TCP 三次握手

为什么要三次握手?

谢希仁著《计算机网络》中讲“三次握手”的目的是“为了防止已失效的连接请求报文段 突然又传送到了服务端，因而产生错误”。

1. 发送 HTTP 请求
2. 服务器处理请求并返回 HTTP 报文
3. 浏览器解析渲染页面
4. 断开连接：TCP 四次挥手

## HTML 语义化的理解

用正确的标签做正确的事情。

html 语义化让页面的内容结构化，结构更清晰，便于对浏览器、搜索引擎解析；即使在没有样式 CSS 情况下也以一种文档格式显示，并且是容易阅读的;

搜索引擎的爬虫也依赖于 HTML 标记来确定上下文和各个关键字的权重，利于 SEO;

使阅读源代码的人对网站更容易将网站分块，便于阅读维护理解。

## document.write 和 innerHTML 的区别？

document.write 只能重绘整个页面  
innerHTML 可以重绘页面的一部分

## 常见兼容性问题？

* 浏览器默认的margin和padding不同。解决方案是加一个全局的\*{margin:0;padding:0;}来统一。
* Chrome 中文界面下默认会将小于 12px 的文本强制按照 12px 显示,
* 可通过加入 CSS 属性 -webkit-text-size-adjust: none; 解决.

## 图片懒加载的原理

图像是img标记，浏览器根据图像的src属性请求图像，因此惰性加载的关键是当图像不在可视区域时，不给`imgs‘赋值。

当页面滚动的事件被触发 -> 执行加载图片操作 -> 判断图片是否在可视区域内 -> 在，则动态将data-src的值赋予该图片

## SVG和CANVAS的区别？

SVG

表示以XML格式定义图像的可伸缩矢量图形。

* 依赖分辨率
* 不支持事件处理器
* 弱的文本渲染能力
* 能以.png或.jpg格式保存图像
* 最适合图像密集型的游戏,其中的许多对象会被频繁重绘

CANVAS

通过 JavaScript 来绘制 2D 图形。

* 不依赖分辨率
* 支持事件处理器
* 最适合带有大型渲染区域的应用程序(比如谷歌地图)
* 复杂度会减慢渲染速度(任何过度使用dom的应用都不快)
* 不适合游戏应用

## iframe的优缺点？

优点：

* 解决加载缓慢的第三方内容如图标和广告等的加载问题
* Security sandbox
* 并行加载脚本

缺点：

* iframe会阻塞主页面的Onload事件
* 即时内容为空，加载也需要时间
* 没有语意

## href 与 src？

* href (Hypertext Reference)指定网络资源的位置，从而在当前元素或者当前文档和由当前属性定义的需要的锚点或资源之间定义一个链接或者关系。（目的不是为了引用资源，而是为了建立联系，让当前标签能够链接到目标地址。）
* src source（缩写），指向外部资源的位置，指向的内容将会应用到文档中当前标签所在位置。
* href与src的区别

1. 请求资源类型不同：href 指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的联系。在请求 src 资源时会将其指向的资源下载并应用到文档中，比如 JavaScript 脚本，img 图片；
2. 作用结果不同：href 用于在当前文档和引用资源之间确立联系；src 用于替换当前内容；
3. 浏览器解析方式不同：当浏览器解析到src ，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等也如此，类似于将所指向资源应用到当前内容。这也是为什么建议把 js 脚本放在底部而不是头部的原因。

## 跨域解决方法：

* jsonp方式
* 代理服务器的方式
* 服务端允许跨域访问(CORS)
* 取消浏览器的跨域限制

## 登录鉴权怎么处理？

* HTTP Auth Authentication
* Cookie + Session
* JWT
* OAuth

## 页面卡顿可以通过哪些方面排查原因？

    1、接口请求速度  
    2、是否有阻塞渲染的复杂逻辑  
    3、是否有较大量的dom渲染或较多消耗渲染的css如c3动画、阴影等新属性。没有必要的渲染是对性能的极大浪费。  
    4、是否有高频的重排重绘动作  
    5、是否是高频事件绑定导致。  
    6、内存泄漏（变量、对象、游离的dom）、内存占用

## 项目优化从哪些方面入手？

1、代码优化：工具类封装、公共组件封装、代码极简化、代码注释。  
            接口优化、加载优化、css封装/压缩、静态资源合理利用（精灵图等）  
            框架使用的优化。  
2、业务优化：产品设计，代码实现方案，交互设计、布局优化、懒加载/骨架屏等提升用户体验。  
3、技术升级：引进新的技术库，如vite打包、less/sass css预处理、ts等

## 通过哪些方式对项目性能监测？

1、通过network监测资源加载的情况，通过瀑布流进行分析  
       浅灰：查询中  
        深灰：停滞，代理转发，请求发送  
        橙色：初始连接  
        绿色：等待中  
        蓝色：内容下载  
    2、通过Performance进行性能排查，找出“慢”的原因，内存泄漏等  
    3、通过性能测试工具 Pingdom、Load Impact、WebPage Test、Octa Gate Site Timer、Free Speed Test、vue-devtools

## 什么是dom、bom、dom事件流。详细的介绍一下bom对象模型包含什么

dom：文档对象模型。  
bom：浏览器对象模型。   
    DOM的事件事件流  
        捕获阶段：事件对象通过目标的祖先从窗口传播到目标的父对象。这个阶段称为捕获阶段。  
        目标阶段：事件对象到达触发事件对象的目标。这个阶段也被称为目标阶段。如果事件类型没有事件冒泡，则事件对象将在完成此阶段后停止。  
        冒泡阶段：事件对象以相反的顺序通过目标依次向祖先进行传播，从目标的父对象开始，到窗口结束。这个阶段也称为冒泡阶段。  
    bom：screen / window / navigator / location / document / history

## 网页前端性能优化的方式有哪些？

1.压缩 css, js, 图片

2.减少 http 请求次数， 合并 css、js 、合并图片（雪碧图）

3.使用 CDN

4.减少 dom 元素数量

5.图片懒加载

6.静态资源另外用无 cookie 的域名

7.减少 dom 的访问（缓存 dom）

8.巧用事件委托

9.样式表置顶、脚本置低

## 线程与进程的区别？

一个程序至少有一个进程,一个进程至少有一个线程.

线程的划分尺度小于进程，使得多线程程序的并发性高。

另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

## 移动端性能优化？

尽量使用css3动画，开启硬件加速

适当使用touch时间代替click时间

避免使用css3渐变阴影效果

可以用transform: translateZ(0) 来开启硬件加速

不滥用float。float在渲染时计算量比较大，尽量减少使用

不滥用web字体。web字体需要下载，解析，重绘当前页面

合理使用requestAnimationFrame动画代替setTimeout

css中的属性（css3 transitions、css3 3D transforms、opacity、webGL、video）会触发GUP渲染，耗电

## axios和fetch区别对比

### axios

axios 是一个基于Promise 用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，它本身具有以下特征

* 从浏览器中创建 XMLHttpRequest
* 支持 Promise API
* 客户端支持防止CSRF
* 提供了一些并发请求的接口（重要，方便了很多的操作）
* 从 node.js 创建 http 请求
* 拦截请求和响应
* 转换请求和响应数据
* 取消请求
* 自动转换JSON数据

### fetch优势：

* 语法简洁，更加语义化
* 基于标准 Promise 实现，支持 async/await
* 同构方便，使用 isomorphic-fetch
* 更加底层，提供的API丰富（request, response）
* 脱离了XHR，是ES规范里新的实现方式
* fetch存在问题

fetch是一个低层次的API，你可以把它考虑成原生的XHR，所以使用起来并不是那么舒服，需要进行封装。

fetch只对网络请求报错，对400，500都当做成功的请求，服务器返回 400，500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject。

fetch默认不会带cookie，需要添加配置项： fetch(url, {credentials: ‘include’})

fetch不支持abort，不支持超时控制，使用setTimeout及Promise.reject的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费

fetch没有办法原生监测请求的进度，而XHR可以

## CSS 工程化的理解

CSS 工程化是为了解决以下问题：

1. 宏观设计：CSS 代码如何组织、如何拆分、模块结构怎样设计？
2. 编码优化：怎样写出更好的 CSS？
3. 构建：如何处理我的 CSS，才能让它的打包结果最优？
4. 可维护性：代码写完了，如何最小化它后续的变更成本？如何确保任何一个同事都能轻松接手？

以下三个方向都是时下比较流行的、普适性非常好的 CSS 工程化实践：

* 预处理器：Less、 Sass 等；
* 重要的工程化插件： PostCss；
* Webpack loader 等 。

基于这三个方向，可以衍生出一些具有典型意义的子问题，这里我们逐个来看：

### （1）预处理器：为什么要用预处理器？它的出现是为了解决什么问题？

预处理器，其实就是 CSS 世界的“轮子”。预处理器支持我们写一种类似 CSS、但实际并不是 CSS 的语言，然后把它编译成 CSS 代码： 那为什么写 CSS 代码写得好好的，偏偏要转去写“类 CSS”呢？这就和本来用 JS 也可以实现所有功能，但最后却写 React 的 jsx 或者 Vue 的模板语法一样——为了爽！要想知道有了预处理器有多爽，首先要知道的是传统 CSS 有多不爽。随着前端业务复杂度的提高，前端工程中对 CSS 提出了以下的诉求：

1. 宏观设计上：我们希望能优化 CSS 文件的目录结构，对现有的 CSS 文件实现复用；
2. 编码优化上：我们希望能写出结构清晰、简明易懂的 CSS，需要它具有一目了然的嵌套层级关系，而不是无差别的一铺到底写法；我们希望它具有变量特征、计算能力、循环能力等等更强的可编程性，这样我们可以少写一些无用的代码；
3. 可维护性上：更强的可编程性意味着更优质的代码结构，实现复用意味着更简单的目录结构和更强的拓展能力，这两点如果能做到，自然会带来更强的可维护性。

这三点是传统 CSS 所做不到的，也正是预处理器所解决掉的问题。预处理器普遍会具备这样的特性：

* 嵌套代码的能力，通过嵌套来反映不同 css 属性之间的层级关系 ；
* 支持定义 css 变量；
* 提供计算函数；
* 允许对代码片段进行 extend 和 mixin；
* 支持循环语句的使用；
* 支持将 CSS 文件模块化，实现复用。

### （2）PostCss：PostCss 是如何工作的？我们在什么场景下会使用 PostCss？

它和预处理器的不同就在于，预处理器处理的是 类CSS，而 PostCss 处理的就是 CSS 本身。Babel 可以将高版本的 JS 代码转换为低版本的 JS 代码。PostCss 做的是类似的事情：它可以编译尚未被浏览器广泛支持的先进的 CSS 语法，还可以自动为一些需要额外兼容的语法增加前缀。更强的是，由于 PostCss 有着强大的插件机制，支持各种各样的扩展，极大地强化了 CSS 的能力。

PostCss 在业务中的使用场景非常多：

1. 提高 CSS 代码的可读性：PostCss 其实可以做类似预处理器能做的工作；
2. 当我们的 CSS 代码需要适配低版本浏览器时，PostCss 的 Autoprefixer 插件可以帮助我们自动增加浏览器前缀；
3. 允许我们编写面向未来的 CSS：PostCss 能够帮助我们编译 CSS next 代码；

### （3）Webpack 能处理 CSS 吗？如何实现？ Webpack 能处理 CSS 吗：

* Webpack 在裸奔的状态下，是不能处理 CSS 的，Webpack 本身是一个面向 JavaScript 且只能处理 JavaScript 代码的模块化打包工具；
* Webpack 在 loader 的辅助下，是可以处理 CSS 的。

如何用 Webpack 实现对 CSS 的处理：

Webpack 中操作 CSS 需要使用的两个关键的 loader：css-loader 和 style-loader

注意，答出“用什么”有时候可能还不够，面试官会怀疑你是不是在背答案，所以你还需要了解每个 loader 都做了什么事情：

* css-loader：导入 CSS 模块，对 CSS 代码进行编译处理；
* style-loader：创建style标签，把 CSS 内容写入标签。

在实际使用中，css-loader 的执行顺序一定要安排在 style-loader 的前面。因为只有完成了编译过程，才可以对 css 代码进行插入；若提前插入了未编译的代码，那么 webpack 是无法理解这坨东西的，它会无情报错。

## AJAX

题目描述:利用 XMLHttpRequest 手写 AJAX 实现

实现代码如下:

const getJSON = function (url) {

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

const xhr = new XMLHttpRequest();

xhr.open("GET", url, false);

xhr.setRequestHeader("Content-Type", "application/json");

xhr.onreadystatechange = function () {

if (xhr.readyState !== 4) return;

if (xhr.status === 200 || xhr.status === 304) {

resolve(xhr.responseText);

} else {

reject(new Error(xhr.responseText));

}

};

xhr.send();

});

};

## ajax过程？

(1)创建XMLHttpRequest对象,也就是创建一个异步调用对象.

(2)创建一个新的HTTP请求,并指定该HTTP请求的方法、URL及验证信息.

(3)设置响应HTTP请求状态变化的函数.

(4)发送HTTP请求.

(5)获取异步调用返回的数据.

(6)使用JavaScript和DOM实现局部刷新.

## 浏览器的缓存

### 1.缓存机制

浏览器发送请求前，根据请求头的expires和cache-control判断是否命中（包括是否过期）强缓存策略，如果命中，直接从缓存获取资源，并不会发送请求。如果没有命中，则进入下一步。 2. 没有命中强缓存规则，浏览器会发送请求，根据请求头的last-modified和etag判断是否命中协商缓存，如果命中，直接从缓存获取资源。如果没有命中，则进入下一步。 3. 如果前两步都没有命中，则直接从服务端获取资源。

### 2.强缓存

强缓存：不会向服务器发送请求，直接从缓存中读取资源。

强制缓存就是向浏览器缓存查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存结果的过程，强制缓存的情况主要有三种(暂不分析协商缓存过程)，如下：

* 第一次请求，不存在缓存结果和缓存标识，直接向服务器发送请求
* 存在缓存标识和缓存结果，但是已经失效，强制缓存失败，则使用协商缓存（暂不分析）
* 存在该缓存结果和缓存标识，且该结果尚未失效，强制缓存生效，直接返回该结果

那么强制缓存的缓存规则是什么？ 当浏览器向服务器发起请求时，服务器会将缓存规则放入HTTP响应报文的HTTP头中和请求结果一起返回给浏览器，控制强制缓存的字段分别是Expires和Cache-Control，其中Cache-Control优先级比Expires高。

### 3.协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

* 协商缓存生效，返回304和Not Modified
* 协商缓存失效，返回200和请求结果

## cookie、sessionStorage、localStorage

### cookie

* 会话性：默认不设置过期时间，保存在内存中；关闭浏览器、用户手动清除 则cookie失效
* 持久性：设置了过期时间，保存在硬盘中；过期时间到、用户手动清除 则cookie失效
* 同一浏览器下的同一个域名下共享(同一浏览器的不同网页窗口共享)
* 单个4KB
* 每次请求都会携带在http请求头中，存储过多会cookie会造成性能的浪费
* 可用于登录识别用户

### session

* 关闭浏览器的网页窗口、用户手动清除 则失效
* 同一浏览器的不同网页窗口不共享，只在当前网页窗口有效
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### localStorage

* 长期有效，不管是关闭网页窗口还是浏览器都不会失效，只能手动清除
* 单个5MB
* 仅在客户端（即浏览器）中保存，不参与和服务器的通信

### 总结

| **类型** | **时效性** | **共享性** | **大小** | **参与** |
| --- | --- | --- | --- | --- |
| cookie | 设置过期时间，时间到则失效；未设置过期时间，浏览器关闭失效 | 同域名下共享 | 单个4KB，一般可存储50个，Opera为30个 | 参与，携带在http请求头中 |
| sessionStorage | 网页窗口关闭则失效 | 只在当前网页窗口中可访问 | 单个5MB | 不参与 |
| localStorage | 永久存储，除非手动清除 | 在特定页面协议内访问 | 单个5MB | 不参与 |

从**数据存储位置、生命周期、存储大小、写入方式、数据共享、发送请求时是否携带、应用场景**这几个方面来回答。

1.存储位置：Cookie、SessionStorage、 LocalStorage都是浏览器的本地存储。 它们的共同点：都是存储在浏览器本地的，它们的区别：cookie是由服务器端写入的，而SessionStorage、 LocalStorage都是由前端写入的，

2.生命周期：cookie的生命周期是由服务器端在写入的时候就设置好的，LocalStorage是写入就一直存在，除非手动清除，SessionStorage是页面关闭的时候就会自动清除。

3.存储大小：cookie的存储空间比较小大概4KB，SessionStorage、LocalStorage存储空间比较大，大概5M。

4.数据共享：Cookie、SessionStorage、LocalStorage数据共享都遵循同源原则，SessionStorage还限制必须是同一个页面。

5.是否携带：在前端给后端发送请求的时候会自动携带Cookie中的数据，但是SessionStorage、LocalStorage不会

总结：由于它们的以上区别，所以它们的应用场景也不同，Cookie一般用于存储登录验证信息SessionID或者token，LocalStorage常用于存储不易变动的数据，减轻服务器的压力，SessionStorage可以用来检测用户是否是刷新进入页面，如音乐播放器恢复播放进度条的功能。

## deffer和async的区别

<script src="demo\_defer.js" defer></script>

如果defer设置了该属性，则指定脚本与解析页面并行下载，并在页面解析完成后执行。

注意：该属性仅适用于外部脚本（仅当存在defer该属性时才应使用）。 src

注意：可以通过多种方式执行外部脚本：

如果async存在：脚本与解析页面并行下载，并在可用时立即执行（在解析完成之前）

如果 defer存在（而不是async）：脚本与解析页面并行下载，并在页面完成解析后执行

如果两者都不存在async或 defer不存在：脚本被立即下载并执行，阻塞解析直到脚本完成

## 重绘和回流有什么区别

什么是回流，什么是重绘，有什么区别？

html 加载时发生了什么：

在页面加载时，浏览器把获取到的HTML代码解析成1个DOM树，DOM树里包含了所有HTML标签，包括display:none隐藏，还有用JS动态添加的元素等。

浏览器把所有样式(用户定义的CSS和用户代理)解析成样式结构体

DOM Tree 和样式结构体组合后构建render tree, render tree类似于DOM tree，但区别很大，因为render tree能识别样式，render tree中每个NODE都有自己的style，而且render tree不包含隐藏的节点(比如display:none的节点，还有head节点)，因为这些节点不会用于呈现，而且不会影响呈现的，所以就不会包含到 render tree中。我自己简单的理解就是DOM Tree和我们写的CSS结合在一起之后，渲染出了render tree。

什么是回流：

当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流(reflow)。每个页面至少需要一次回流，就是在页面第一次加载的时候，这时候是一定会发生回流的，因为要构建render tree。在回流的时候，浏览器会使渲染树中受到影响的部分失效，并重新构造这部分渲染树，完成回流后，浏览器会重新绘制受影响的部分到屏幕中，该过程成为重绘。

什么是重绘：

当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为重绘。

区别：

他们的区别很大：

回流必将引起重绘，而重绘不一定会引起回流。比如：只有颜色改变的时候就只会发生重绘而不会引起回流

当页面布局和几何属性改变时就需要回流

比如：添加或者删除可见的DOM元素，元素位置改变，元素尺寸改变——边距、填充、边框、宽度和高度，内容改变

扩展：

浏览器的帮忙

所以我们能得知回流比重绘的代价要更高，回流的花销跟render tree有多少节点需要重新构建有关系

因为这些机制的存在，所以浏览器会帮助我们优化这些操作，浏览器会维护1个队列，把所有会引起回流、重绘的操作放入这个队列，等队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会flush队列，进行一个批处理。这样就会让多次的回流、重绘变成一次回流重绘。

自己的优化

靠浏览器不如靠自己，我们可以改变一些写法减少回流和重绘

比如改变样式的时候，不去改变他们每个的样式，而是直接改变className 就要用到cssText 但是要注意有一个问题，会把原有的cssText清掉，比如原来的style中有’display:none;’，那么执行完上面的JS后，display就被删掉了。

为了解决这个问题，可以采用cssText累加的方法，但是IE不支持累加，前面添一个分号可以解决。

还有添加节点的时候比如要添加一个div里面有三个子元素p，如果添加div再在里面添加三次p，这样就触发很多次回流和重绘，我们可以用cloneNode(true or false) 来避免，一次把要添加的都克隆好再appened就好了，还有其他很多的方法就不依依说了