JS性能优化 - 学员预习

一. 先看看有哪些性能指标

常规指标

FP：首次绘制时间

首次绘制包括了任何用户自定义的背景绘制，它是将第一个像素点绘制到屏幕的时刻，

对于应用页面，用户在视觉上首次出现不同于跳转之前的内容时间点，或者说是页面发生第一次绘制的时间点。

FCP：首次有内容绘制时间

指浏览器完成渲染 DOM 中第一个内容的时间点，可能是文本、图像、SVG或者其他任何元素，此时用户应该在视觉上有直观的感受。

FMP：首次有意义绘制时间

指页面关键元素渲染时间。这个概念并没有标准化定义，因为关键元素可以由开发者自行定义——究竟什么是“有意义”的内容，只有开发者或者产品经理自己了解。

首屏时间

对于所有网页应用，这是一个非常重要的指标。用大白话来说，就是进入页面之后，应用渲染完整个手机屏幕（未滚动之前）内容的时间。需要注意的是，业界对于这个指标其实同样并没有确切的定论，比如这个时间是否包含手机屏幕内图片的渲染完成时间。

TTI：用户可交互时间

顾名思义，也就是用户可以与应用进行交互的时间。一般来讲，我们认为是 domready 的时间，因为我们通常会在这时候绑定事件操作。如果页面中涉及交互的脚本没有下载完成，那么当然没有到达所谓的用户可交互时间。那么如何定义 domready 时间呢?

TTFB：网络请求耗时

TTFB是发出页面请求到接收到应答数据第一个字节所花费的毫秒数

DCL：DomContentLoaded

L：DOM onload

总下载时间

页面所有资源加载完成所需要的时间。一般可以统计 window.onload 时间，这样可以统计出同步加载的资源全部加载完的耗时。如果页面中存在较多异步渲染，也可以将异步渲染全部完成的时间作为总下载时间。

DOMContentLoaded 与 load 事件的区别

DOMContentLoaded 指的是文档中 DOM 内容加载完毕的时间，也就是说 HTML 结构已经完整。但是我们知道，很多页面包含图片、特殊字体、视频、音频等其他资源，这些资源由网络请求获取，DOM 内容加载完毕时，由于这些资源往往需要额外的网络请求，还没有请求或者渲染完成。而当页面上所有资源加载完成后，load 事件才会被触发。因此，在时间线上，load 事件往往会落后于 DOMContentLoaded 事件。

关于 DOMContentLoaded 和 domReady？

我们简单说一下，浏览器是从上到下，从左到右，一个个字符串读入，大致可以认为两个同名的开标签与闭标签就是一个DOM(有的是没有闭签)，这时就忽略掉它的两个标签间的内容。页面上有许多标签, 但标签会生成同样多的DOM，因为有的标签下只允许存在特定的子标签，比如tr下面一定是td,th, select下面一定是opgroup,option,而option下面，就算你写了，它都会忽略掉，option下面只存在文本，这就是我们需要自定义下拉框的缘故。

我们说过, 这顺序是从上到下, 有的元素很简单,会构建得很快,但标签存在src, href属性,它会引用外部资源,这就要区别对待了.比如说, script标签,它一定会等src指定的脚本文件加载下来,然后全部执行了里面的脚本,才会分析下一个标签.这种现象叫做堵塞.

堵塞是一种非常致命的现象,因为浏览器渲染引擎是单线程的,如果头部脚本过多过大会导致白屏,影响用户体验,因此雅虎的20军规就有一条提到 ,将所有script标签放到body之后.

此外, style标签与link标签,它们在加载样式文件时是不会堵塞,但它们一旦异步加载好,就立即开始渲染已经构建好的元素节点们, 这可能会引起reflow, 这也影响速度.

另一个影响DOM树构建的因此是iframe,它也会加载资源, 虽然不会堵塞DOM构建,但它由于是发出HTTP请求,而HTTP请求是有限,它会与父标签的其他需要加载外部资源的标签产生竞争。我们经常看到一些新闻网,上面会挂许多iframe广告, 这些页面一开始加载时就很卡,也是这缘故.

此外还有object元素, 用来加载flash

等等，这些东西都会影响到DOM树的构建过程.因此在这时候,当我们贸贸然,使用getElementById, getElementsByTagName获取元素,然后操作它们, 就会有很大机率碰到 元素为null的 异常. 这时, 目标元素还可以没有转换为DOM节点, 还只是一个普通的字符串呢!

很早期, 浏览器提供了一个window.onload方法,但这东西是等到所有标签变成DOM,并且外部资源,图片,背景音乐什么都加载好才触发, 时间上有点晚.

幸好,浏览器提供了一个document.readyState属性,当它变成complete时,说明这时机到了

但这是一个属性,不是一个事件,需要使用不太精确的setInterval轮询

在标签浏览器, W3C终于绅士地提供了一个DOMContentLoaded事件把这件事解决了。

chrome 最新指标

LCP（Largest Contentful Paint）

衡量页面的加载体验，它表示视口内可见的最大内容元素的渲染时间。相比 FCP，这个指标可以更加真实地反映具体内容加载速度。比如，如果页面渲染前有一个 loading 动画，那么 FCP 可能会以 loading 动画出现的时间为准，而 LCP 定义了 loading 动画加载后，真实渲染出内容的时间。

FID（First Input Delay）

衡量可交互性，它表示用户和页面进行首次交互操作所花费的时间。它比 TTI（Time to Interact）更加提前，这个阶段虽然页面已经显示出部分内容，但并不能完全具备可交互性，对于用户的响应可能会有较大的延迟。

CLS（Cumulative Layout Shift）

衡量视觉稳定性，表示页面的整个生命周期中，发生的每个意外的样式移动的所有单独布局更改得分的总和。所以这个分数当然越小越好。

二. 如何获取这些指标

字段	含义

navigationStart	加载起始时间，如果没有前一个页面的unload,则与fetchStart值相等
redirectStart	重定向开始时间（如果发生了HTTP重定向，每次重定向都和当前文档同域的话，就返回开始重定向的fetchStart的值。其他情况，则返回0）
redirectEnd	重定向结束时间（如果发生了HTTP重定向，每次重定向都和当前文档同域的话，就返回最后一次重定向接受完数据的时间。其他情况则返回0）
fetchStart	fetchStart 浏览器发起资源请求时，如果有缓存，则返回读取缓存的开始时间
domainLookupStart	DNS域名开始查询的时间,如果有本地的缓存或keep-alive等，则返回fetchStart
domainLookupEnd	domainLookupEnd 查询DNS的结束时间。如果没有发起DNS请求，同上
connectStart	TCP开始建立连接的时间,如果有本地的缓存或keep-alive等,则与fetchStart值相等
secureConnectionStart	https 连接开始的时间,如果不是安全连接则为0
connectEnd	TCP完成握手的时间，如果有本地的缓存或keep-alive等，则与connectStart 值相等
requestStart	HTTP请求读取真实文档开始的时间,包括从本地缓存读取
requestEnd	HTTP请求读取真实文档结束的时间,包括从本地缓存读取
responseStart	返回浏览器从服务器收到（或从本地缓存读取）第一个字节时的Unix毫秒时间戳
responseEnd	返回浏览器从服务器收到（或从本地缓存读取，或从本地资源读取）最后一个字节时的Unix毫秒时间戳
unloadEventStart	前一个页面的unload的时间戳 如果没有则为0
unloadEventEnd	与unloadEventStart相对应，返回的是unload函数执行完成的时间戳
domLoading	这是当前网页DOM结构开始解析时的时间戳，是整个过程的起始时间戳，浏览器即将开始解析第一批收到的HTML文档字节，此时

	戳，浏览器即将开始解析第一批收到的 HTML 文档字节，此时document.readyState变成loading,并将抛出readyStateChange事件
domInteractive	返回当前网页DOM结构结束解析、开始加载内嵌资源时时间戳,document.readyState 变成interactive，并将抛出readyStateChange事件(注意只是DOM树解析完成,这时候并没有开始加载网页内的资源)
domContentLoadedEventStart	网页domContentLoaded事件发生的时间
domContentLoadedEventEnd	网页domContentLoaded事件脚本执行完毕的时间,domReady的时间
domComplete	DOM树解析完成,且资源也准备就绪的时间,document.readyState变成complete.并将抛出readystatechange事件
loadEventStart	load 事件发送给文档，也即load回调函数开始执行的时间
loadEventEnd	load回调函数执行完成的时间

相关参数计算

字段	描述	计算方式	意义
unload	前一个页面卸载耗时	unloadEventEnd — unloadEventStart	-
redirect	重定向耗时	redirectEnd — redirectStart	重定向的时间
appCache	缓存耗时	domainLookupStart — fetchStart	读取缓存的时间
dns	DNS 解析耗时	domainLookupEnd — domainLookupStart	可观察域名解析服务是否正常
tcp	TCP 连接耗时	connectEnd — connectStart	建立连接的耗时
ssl	SSL 安全连接耗时	connectEnd — secureConnectionSt	反映数据安全连接建立耗时

		art	
ttfb	Time to First Byte(TTFB)网络请求耗时	responseStart — requestStart	TTFB是发出页面请求到接收到应答数据第一个字节所花费的毫秒数
response	响应数据传输耗时	responseEnd — responseStart	观察网络是否正常
dom	DOM解析耗时	domInteractive — responseEnd	观察DOM结构是否合理，是否有JS阻塞页面解析
dcl	DOMContentLoaded 事件耗时	domContentLoadedEventEnd — domContentLoadedEventStart	当 HTML 文档被完全加载和解析完成之后，DOMContentLoaded事件被触发，无需等待样式表、图像和子框架的完成加载
resources	资源加载耗时	domComplete — domContentLoadedEventEnd	可观察文档流是否过大
domReady	DOM阶段渲染耗时	domContentLoadedEventEnd — fetchStart	DOM树和页面资源加载完成时间，会触发`domContentLoaded`事件
首次渲染耗时	首次渲染耗时	responseEnd–fetchStart	加载文档到看到第一帧非空图像的时间，也叫白屏时间
首次可交互时间	首次可交互时间	domInteractive–fetchStart	DOM树解析完成时间，此时document.readyState为interactive
首包时间耗时	首包时间	responseStart–	DNS解析到响应返回

		domainLookupStart	给浏览器第一个字节的时间
页面完全加载时间	页面完全加载时间	loadEventStart – fetchStart	–
onLoad	onLoad事件耗时	loadEventEnd – loadEventStart	

代码工程方法获取

性能面板，火箭图

lighthouse

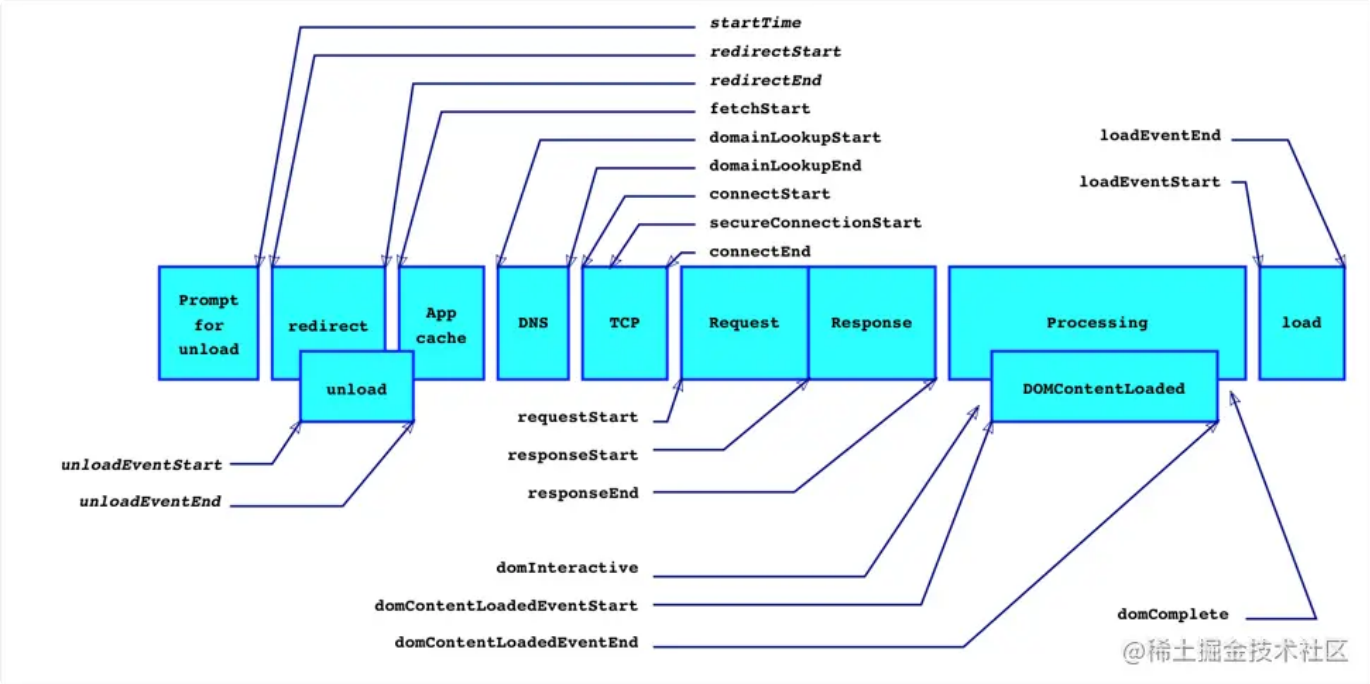
window.performance

web-vitals

https://web.dev/vitals/

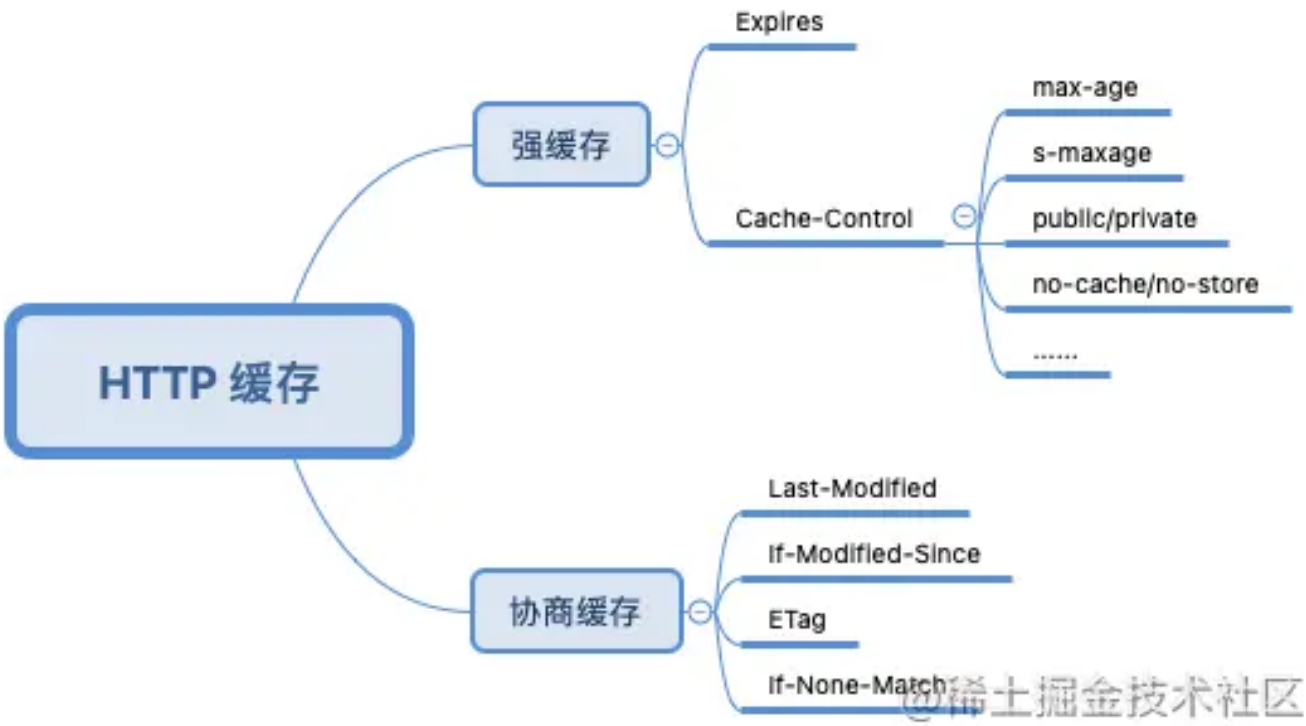
三. 从什么维度来剖析性能?

维度1：I/O(network) 维度



App cache 阶段

- 合理利用缓存；



缓存中有哪些细节需要注意?

1. 直接在浏览器端输入的 http://xxx.xxx.com/index.html ，该文件是不会被缓存的；
2. webpack 中的hash 指纹，需要合理的利用，去让“该缓存的内容被缓存”；
3. 304 是协商缓存；
4. 协商缓存中的 Modified 是根据时间判断，这一秒的时间，可能引发很多问题；
5. 在 CDN 下，hash 缓存是否能够有比较好的缓存效果?
6. 没有了强缓存的必要字段值，浏览器还会走强缓存吗？答案是肯定的。（启发式缓存）

DNS 阶段

这里就涉及到一些计算机网络的知识，不详细赘述。

TCP 阶段

- http1.1 ?
- http 2 ?

RES、REQ 阶段

1. 首先是包的总体积，如何缩小到极致?

- uglify；
- runtime；
- tree shaking；
- 图片格式，应该如何进行分解？webp?

2. 首屏加载的内容，如何进行分解?

- code splitting；

3. 如何和TCP请求数量之间 tradeoff?

- Chrome ，同源下最多6个并发；

（以上规则同样适用于接口的请求）

Processing 阶段

- 如何最大效率地满足加载?

浏览器是如何加载这几种文件的?

- 一般顺序是什么样子?
- async, defer 标签的区别是什么?

维度2：渲染维度

如何有效避免频繁操作DOM?

- 利用 fragment；

如何有效利用回流与重绘的特点?

- CLS 指标；
- 哪些参数的 get 和 set 会引起回流和重绘?

什么是回流与重绘?

回流

回流又名重排，指几何属性需改变的渲染。但感觉回流这个词较高大上，后续统称回流吧。

可理解成，将整个网页填白，对内容重新渲染一次。只不过以人眼的感官速度去看浏览器回流是不会有任何变化的，若你拥有闪电侠的感官速度去看浏览器回流(实质是将时间调慢)，就会发现每次回流都会将页面清空，再从左上角第一个像素点从左到右从上到下这样一点一点渲染，直至右下角最后一个像素点。每次回流都会呈现该过程，只是感受不到而已。

渲染树的节点发生改变，影响了该节点的几何属性，导致该节点位置发生变化，此时就会触发浏览器回流并重新生成渲染树。回流意味着节点的几何属性改变，需重新计算并生成渲染树，导致渲染树的全部或部分发生变化。

重绘

重绘指更改外观属性而不影响几何属性的渲染。相比回流，重绘在两者中会温和一些，后续谈到的CSS性能优化就会基于该特点展开。

渲染树的节点发生改变，但不影响该节点的几何属性。由此可见，回流对浏览器性能的消耗是高于重绘的，而且回流一定会伴随重绘，重绘却不一定伴随回流。

为何回流一定会伴随重绘呢？整个节点的位置都变了，肯定要重新渲染它的外观属性啊！

属性分类

以下对一些常用的几何属性和外观属性分类，其实同种分类的属性都有一些共同点，各位同学可自行感受。推荐一个查询属性渲染状态的网站CssTriggers，可查看每个属性在渲染过程中发生了什么影响了什么。

- 几何属性：包括布局、尺寸等可用数学几何衡量的属性
  - 布局：display、float、position、list、table、flex、columns、grid
  - 尺寸：margin、padding、border、width、height
- 外观属性：包括界面、文字等可用状态向量描述的属性
  - 界面：appearance、outline、background、mask、box-shadow、box-reflect、filter、opacity、clip
  - 文字：text、font、word

https://csstriggers.com/

哪些情况会回流或重绘?

- 改变窗口大小
- 修改盒模型
- 增删样式
- 重构布局
- 重设尺寸
- 改变字体
- 改动文字

如何合理使用 GPU 加速?

- `transform`

四. 一些比较好的文章

节流和防抖相关

- Debouncing and Throttling Explained Through Examples
- 谈谈 JS 中的函数节流
- JavaScript 函数节流和函数防抖之间的区别
- 高性能滚动 scroll 及页面渲染优化
- 从 lodash 源码学习节流与防抖
- 理解并优化函数节流 Throttle

浏览器引擎渲染性能相关

- Inside look at modern web browser
- How Browsers Work: Behind the scenes of modern web browsers
- How browsers work
- How browser rendering works—behind the scenes
- What Every Frontend Developer Should Know About Webpage Rendering
- 前端文摘：深入解析浏览器的幕后工作原理
- 从 Chrome 源码看浏览器如何加载资源

- 浏览器内核渲染：重建引擎
- 体现工匠精神的 Resource Hints
- 浏览器页面渲染机制，你真的弄懂了吗
- 前端不止：Web 性能优化 — 关键渲染路径以及优化策略
- 浏览器前端优化
- 浅析前端页面渲染机制
- 浅析渲染引擎与前端优化
- 渲染性能
- Repaint 、Reflow 的基本认识和优化 (2)

动画性能相关

- Timing control for script-based animations
- Gain Motion Superpowers with requestAnimationFrame
- CSS Animation 性能优化
- GSAP的动画快于 jQuery 吗？为何?
- Javascript 高性能动画与页面渲染
- 也许你不知道，JS animation 比 CSS 更快！
- 渐进式动画解决方案
- 你应该知道的 requestIdleCallback
- 无线性能优化：Composite
- 优化动画卡顿：卡顿原因分析及优化方案
- 一篇文章说清浏览器解析和 CSS（GPU）动画优化

实战案例相关

- Building the Google Photos Web UI
- A Netflix Web Performance Case Study
- The Cost Of JavaScript In 2018
- How we reduced our initial JS/CSS size by 67%
- Front-End Performance Checklist 2019
- 网站性能优化实战——从 12.67s 到 1.06s 的故事
- 前端黑科技：美团网页首帧优化实践

- Web 字体图标–自动化方案
- JS 加载慢？谷歌大神带你飞!
- 前端性能优化（三） 移动端浏览器前端优化策略
- CSS @font–face 性能优化
- 移动 Web 性能优化从入门到进阶
- 记一次惊心动魄的前端性能优化之旅