前端性能优化的知识点零碎、不成体系，————————痛点

从输入url到页面加载完成简单概述(网络层)

DNS缓存 DNS prefetch

TCP: 长连接、预连接、接入

SPDY 协议

1.DNS解析

预解析<meta http-equiv="x-dns-prefetch-control" content="on">

<link rel="dns-prefetch" href="//example.com">

如果浏览器最近将一个域名解析为IP地址，所属的操作系统将会缓存，下一次DNS解析时间可以低至0-1ms。 如果结果不在系统本地缓存，则需要读取路由器的缓存，则解析时间的最小值大约为15ms。如果路由器缓存也不存在，则需要读取ISP（运营商）DNS缓存，一般像taobao.com、baidu.com这些常见的域名，读取ISP（运营商）DNS缓存需要的时间在80-120ms，如果是不常见的域名，平均需要200-300ms。一般的网站在运营商这里都能查询的到，所以普遍来说DNS Prefetch可以给一个域名的DNS解析过程带来15-300ms的提升

2.TCP连接 三次握手

预连接：<link rel="preconnect" href="//example.com">

3.HTTP请求

利用浏览器加载文件优先级 prefetch preload

好文[从Chrome源码看浏览器如何加载资源](https://zhuanlan.zhihu.com/p/30558018)

4.服务端处理请求，HTTP 响应返回

5.浏览器拿到相应数据，解析响应，走渲染流程，展示给用户

mind

图像裁切

Webview理解 浏览器回退挽留

又见性能优化！

经典题

懒加载

事件节流和防抖

可视化工具

Performance

pagespeed插件

LightHouse

chrome基于lighthouse的auditing面板

文档：https://developers.google.com/web/tools/lighthouse/?hl=zh-cn

chrome插件：https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk/related

命令行npm install -g lighthouse

lighthouse https://airhorner.com/

性能检测

网页性能管理详解:http://www.ruanyifeng.com/blog/2015/09/web-page-performance-in-depth.html

W3C性能API

```
const timing = window.performance.timing
// DNS查询耗时
timing.domainLookupEnd - timing.domainLookupStart
// TCP连接耗时
timing.connectEnd - timing.connectStart
// 内容加载耗时
timing.responseEnd - timing.requestStart

    // firstbyte：首包时间
    timing.responseStart — timing.domainLookupStart

    // fpt：First Paint Time, 首次渲染时间 / 白屏时间
    timing.responseEnd — timing.fetchStart

    // tti：Time to Interact，首次可交互时间
    timing.domInteractive — timing.fetchStart

    // ready：HTML 加载完成时间，即 DOM 就位的时间
    timing.domContentLoaded — timing.fetchStart

    // load：页面完全加载时间
    timing.loadEventStart — timing.fetchStart
```

未来

一.网络层

请求过程的优化

HTTP请求优化

构建工具性能优化

Gzip压缩原理

图片优化

减少网络请求

本地存储(缓存)

浏览器的缓存机制

离线缓存技术

二.渲染层

服务端渲染的探索和实践

浏览器的渲染机制解析

css性能优化

js性能方案

DOM优化

原理和基本思路

事件循环与异步机制

回流和重绘

首屏渲染提速：懒加载初探

浏览器端（渲染层）

资源加载优化

服务端渲染

浏览器缓存机制的利用

DOM数构建

网页排版和渲染过程

回流和重绘的考量

DOM操作的合理规避

按获取资源优先级

1. Server Worker Cache（https）

Service Worker 是一种独立于主线程之外的 Javascript 线程。它脱离于浏览器窗体

Service Worker 的生命周期包括 install、active、working 三个阶段。一旦 Service Worker 被 install，它将始终存在，只会在 active 与 working 之间切换，除非我们主动终止它。这是它可以用来实现离线存储的重要先决条件。

注意： 1，前提https 2.如果 Service Worker 没能命中缓存，一般情况会使用 fetch() 方法继续获取资源。这时候，浏览器就去 memory cache 或者 disk cache 进行下一次找缓存的工作了。注意：经过 Service Worker 的 fetch() 方法获取的资源，即便它并没有命中 Service Worker 缓存，甚至实际走了网络请求，也会标注为 from ServiceWorker

2. Memory Cache

特点：又快又短

相应最快的，当渲染进程结束后就会消失(关闭tab)

Base64 格式的图片几乎永远可以，体积不大的js,css也可以，较大的js css，由于内存资源有限，会被放到磁盘中

memory cache 机制保证了一个页面中如果有两个相同的请求 (例如两个 src 相同的 <img>，两个 href 相同的 <link>)都实际只会被请求最多一次，避免浪费

不过在匹配缓存时，除了匹配完全相同的 URL 之外，还会比对他们的类型，CORS 中的域名规则等。因此一个作为脚本 (script) 类型被缓存的资源是不能用在图片 (image) 类型的请求中的，即便他们 src 相等

在从 memory cache 获取缓存内容时，浏览器会忽视例如 max-age=0, no-cache 等头部配置。例如页面上存在几个相同 src 的图片，即便它们可能被设置为不缓存，但依然会从 memory cache 中读取。这是因为 memory cache 只是短期使用，大部分情况生命周期只有一次浏览而已。而 max-age=0 在语义上普遍被解读为“不要在下次浏览时使用”，所以和 memory cache 并不冲突。

3. HTTP Cache (disk Cache)

强缓存(利用Expires 和 Cache-Control)

实现：

expires

服务器返回相应时，在response header写入expries过期时间(再次请求如果本地时间小于 expires 设定的过期时间，那么就直接去缓存中取这个资源)

expires: Wed, 11 Sep 2019 xx:xx:xx GMT

缺点：“本地时间”的依赖

HTTP1.1新增Cache-Control

Max-age是时间长度而非时间戳(Cache-Control 是 expires的完全替代方案。我们继续使用 expires 的唯一目的就是向下兼容)

cache-control: max-age=31536000

cache-control: max-age=3600, s-maxage=31536000（s-maxage 用于表示 cache 服务器上（比如 cache CDN）的缓存的有效时间的，并只对 public 缓存有效。s-maxage 仅在代理服务器中生效，客户端中我们只考虑max-age。）

publish与private(默认)

Public：可以被浏览器缓存，也可以被代理服务器缓存

private:只能被浏览器缓存

no-store与no-cache

No-cache绕过浏览器，每一次发起请求都不会再去询问浏览器的缓存情况，而是直接向服务端去确认该资源是否过期（走协商缓存）

no-store不会走浏览器缓存(memory cache)，连服务端的缓存确认也绕开了，只允许你直接向服务端发送请求、并下载完整的响应。（F5强刷发送的请求头带有no-store,不适用浏览器缓存）

协商缓存

特征：浏览器与服务器合作下的缓存策略，浏览器需要向服务器询问缓存的相关信息，进而判断是重新发送请求，还是从本地

State Code: 304 Not Modified

两者同时存在以Etag为准

实现：

Last-Modified

1.Last-Modified是个时间戳，会在首次请求时在response header返回

Last-Modified: Fri, 27 Oct 2017 06:35:57 GMT

2.随后每次请求都会带上If-Modified-Since的时间戳，这个值是上次返回给他的last-modified

If-Modified-Since: Fri, 27 Oct 2017 06:35:57 GMT

3.服务器收到时间戳后会对比该时间戳和资源在服务器上的最后修改时间是否一致。

不一致：就返回一个完整的相应内容，并且在response header上添加新的last-modified

一致：返回304， response header上不会添加last-modified

缺点(服务器并没有正确感知文件的变化。)：

1.我们编辑了文件，但文件的内容没有改变。服务端并不清楚我们是否真正改变了文件，它仍然通过最后编辑时间进行判断。因此这个资源在再次被请求时，会被当做新资源

2.If-Modified-Since 只能检查到以秒为最小计量单位的时间差，假如修改文件速度太快，该重新请求时，没有请求

Etag(作为last-modified的补充 HTTP1.1)

Etag是服务器为每个资源生成的唯一的表示字符串，基于文件内容编码的能精确感知文件变化

过程和last-modified类似

1.首次请求在respinse header加etag标识符字符串

ETag: W/"2a3b-1602480f459"

2.再次请求会带上一个值相同步的名为：If-Modified-Match

If-None-Match: W/"2a3b-1602480f459"

3服务器对比

一致：返回304

不一致：请求完整的内容

缺点

Etag 的生成过程需要服务器额外付出开销，会影响服务端的性能

决策建议：(根据Chrome官方图)

当我们的资源内容不可复用时，直接为 Cache-Control 设置 no-store，拒绝一切形式的缓存；否则考虑是否每次都需要向服务器进行缓存有效确认，如果需要，那么设 Cache-Control 的值为 no-cache；否则考虑该资源是否可以被代理服务器缓存，根据其结果决定是设置为 private 还是 public；然后考虑该资源的过期时间，设置对应的 max-age 和 s-maxage 值；最后，配置协商缓存需要用到的 Etag、Last-Modified 等参数

频繁变动的资源：Cache-Control: no-cache

每次请求服务器，然后配合 ETag 或者 Last-Modified 来验证资源是否有效

不常变动的资源：

Cache-Control: max-age=31536000

这样浏览器之后请求相同的 URL 会命中强制缓存。而为了解决更新的问题，就需要在文件名(或者路径)中添加 hash， 版本号等动态字符，之后更改动态字符，从而达到更改引用 URL 的目的，让之前的强制缓存失效 (其实并未立即失效，只是不再使用了而已)。

4. Push Cache（HTTP2）

1.缓存的最后一个策略

2.。存在于会话阶段，当session终止时，缓存释放

4. 只能使用一次

3.不同页面只要页献了一个http2链接，就可以共享一个push cache

https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/

5. 网络请求

浏览器缓存

按获取资源优先级

总结

1. 调用 Service Worker 的 fetch 事件响应

2. 查看 memory cache

3. 查看 disk cache。这里又细分：

– 如果有强制缓存且未失效，则使用强制缓存，不请求服务器。这时的状态码全部是 200

– 如果有强制缓存已失效，使用对比缓存，比较后确定 304 还是 200

4. 发送网络请求，等待网络响应

5. 把响应内容存入 disk cache (如果 HTTP 头信息配置可以存的话)

6. 把响应内容 的引用 存入 memory cache (无视 HTTP 头信息的配置)

7. 把响应内容存入 Service Worker 的 Cache Storage (如果 Service Worker 的脚本调用了 cache.put())

本地存储

1.cookie

解决http无状态问题，以键值对存在

性能弊端

1.不够大 最大只能有 4KB

2.过大cookie会带来性能浪费

cookie默认被设置为主机名，也可以手动设置domain

同一域名下所有请求都会存在cookie

Web Storage(不会与服务器发生通信)

localstorage（持久化本地存储）

持久化，永远不会过期

5M内存，but超过2.5M某些机型会卡顿

localstorage扩容

<iframe src="http://xxxx.com.cn/storage.html"></iframe> //嵌入另一个域的一个空页面

5M的限制是针对同域的，可以利用iframe跨域扩容

localStorage（postmessage），不过这种就是异步的了

应用场景：存储一些内容稳定的资源

base64格式的图片

不常变动的静态文件css js等

ls-cache-files库

sessuibstorage(会话级别存储)

tab关闭则会消失

应用场景：微博的 Session Storage 就主要是存储你本次会话的浏览足迹

PWA: service worker

workbox-webpack-plugin

Workbox :Service worker 是一个在后台运行的脚本，主要用于浏览器和服务器之间的网络交互，从而实现一些新功能。而 Workbox 则是 Google 基于这项功能开发的一个库，使一些常用的功能做到了开箱即用。

Web sql

兼容问题

webSQL关系型

Index DB

indexedDB非关系型 是webSQL的继承者 更加好用

它不仅可以存储字符串，还可以存储二进制数据

没有存储上限的（一般来说不会小于 250M）

创建多个数据库，一个数据库中创建多张表，一张表中存储多条数据

方法略繁琐

应用解决方案：localforage

localStorage API 的异步存储来改进你的 Web 应用程序的离线体验。它能存储多种类型的数据，而不仅仅是字符串。

支持回调和promise

针对浏览器做降级处理本地存储不支持 IndexedDB 或 WebSQL，则使用 localStorage，你也可以自己设置驱动的优先级