从URL到页面加载完成，发生了什么

1. DNS 解析 --- 域名解析成ip
2. TCP 连接 --- 根据ip与服务器建立起连接
3. 发送HTTP请求
4. 服务器处理请求，HTTP响应返回
5. 浏览器解析响应内容，渲染到页面

\* 1~2前端做的有限。主要优化在3~5方面

1. 网络层面的性能优化
2. 减少单次请求所花时间---http请求优化
   1. 构建工具性能调优

·webpack主要性能瓶颈及优化方案

一、构建过程太花时间

(一) loader

配置include或exclude避免不必要的转义

为loader增加cacheDirectory=true参数

1. 第三方库

DLLPlugin

1. 将loader单线程改成多线程---happypack
2. 打包的代码块体积过大
3. 通过webpack-bundle-analyzer分析块的依赖关系和块大小
4. 拆分资源
5. 删除冗余代码--Tree-shaking

模块级别：基于import/export语法的模块。webpack会根据依赖图把没有实际用到的模块去除

代码级别：UglifyJsPlugin，压缩过程中对console\注释等进行自动化删除



这段手动引入 UglifyJsPlugin 的代码其实是 webpack3 的用法，webpack4 现在已经默认使用 uglifyjs-webpack-plugin 对代码做压缩了——在 webpack4 中，我们是通过配置 optimization.minimize 与 optimization.minimizer 来自定义压缩相关的操作的。

1. 按需加载(代码分离)---以路由代码分离为例

采用ES的import().then 和webpack的require.ensure(dependencies, callback, chunk)来进行分离。

import()内部都会调用Promise。要用polyfill来或者plugin-syntax-dynamic-plugin处理兼容性

二者本质上在特定的时间触发相应的回调。

bundle-loader基于require.ensure()，实现代码分离

react-loadable基于import实现代码分离。同时可以实现预加载，懒加载等强大功能

推荐使用import()，require.ensure属于传统的回调方式，有些问题

* 1. Gzip压缩原理

代码压缩不是构建工具的专利

HTTP压缩：请求头header里面加上 accept-encoding: gzip实现。服务器在没有发送前进行压缩。gzip压缩内核是deflafe，原理是找出文件中重复的代码，替换他们，从而使整个文件变小。服务器压缩GZIP，浏览器解压GZIP（content-encoding）时间与中间传输时间来决定是否使用。

webpack的GZIP压缩: js，css这些，减少服务器gzip时的压力。

GZIP对于图片，视频等资源根据实际情况而定，有肯能压缩了之后变大。

* 1. 图片优化---质量与性能之间的平衡点(多体现在电商项目)

1. 像素用二进制表示，像素中二进制位数越多，颜色越丰富2^n中颜色
2. 各种格式比较及应用场景
3. jpeg、jpg --- 有损，体积小，不透明 banner
4. png-8、png-24---无损，体积大，透明 简单图像logo
5. svg --- 文本，体积小，无限放大 矢量图标
6. base64 ---- 文本，依赖编码(url-loader)，减少服务器请求次数

小图标(2k,编码后是原来的4/3)，雪碧图，更新评率低(减少编译)

1. webP --- 全能选手 兼容性差 根据浏览器做url的字符串切割
2. iconfont ---
3. css雪碧图---base64，webpack可以自动生成雪碧图
4. 减少网络请求
5. 缓存策略
   1. 浏览器缓存机制

Memory Cache from memory cache 内存 Base64图片/体积小的js,css文件

from disk cache 磁盘

Service Worker Cache from serviceworker

HTTP Cache 2kb

Push Cache(HTTP2)

\*缓存在内存或磁盘具有随机性

\*浏览器只有在 Memory Cache、HTTP Cache 和 Service Worker Cache 均未命中的情况下才会去询问 Push Cache。

强缓存：不与服务器发生通信, cache-control:max-age=秒 > expires, 命中status code:200

协商缓存：与服务器发生通信, Etag > Last-modified 命中 status code:304

强缓存要点:

\*expires 过期时间戳，依赖本地时间

private 默认值 只允许浏览器缓存

no-cache 都可以出现在request或response中 不允许任何未经目标服务器验证 的缓存来响应请求。

no-store 都可以出现在request或response中 不允许任何缓存策略，保存响应。

\*允许代理服务器缓存：cache-control:public/cache-control: s-maxage=秒

协商缓存要点：

\*Etag 是根据内容编码的标识字符串，Last-modified 是时间戳

Last-modified原理:第二次的request header:If-modified-since=上一次reponse-header:Last-modified。弊端：编辑了但内容没变，if…since只能感知秒级别的，在毫秒完成更改是感知不到的。

* 1. 离线缓存技术

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，因此无法直接访问 DOM。可以实现离线缓存，消息推送，网络代理等。

\*  Server Worker必须以 https 协议为前提。

* 1. 缓存决策指南

是否重复利用资源---> 是否每次服务器验证---> 是否允许代理服务器缓存

1. 本地存储

a) Cookie

以键值对的形式存在 --- application

cookie 4KB ---- 多余会被裁剪

cookie 是紧跟域名的----response header 中 set-cookie 可以看出来

同一域名下的**所有**请求都会携带cookie --- 网络浪费

客户端设置：documen.cookie = “usename= fcheng;path=/”

服务端设置：Set-Cookie: usename= fcheng;path=/

b) Web Storage (IE8及以上)

生命周期 作用域

Local Storage 持久化本地储存，用不过期，除非主动删除。同源可访问

Session Storage 临时性本地储存，会话级别，页面关闭即被删除。除了同源还需同一个窗口

Web Storage特性：①容量大，根据浏览器不同，在5-10M之间。

②仅位于浏览器端，不与服务器端通信

③只能存字符串，需要解析

Web Storage核心API: localstorage.setItem(‘’,’’)，localstorage.getItem(‘’) localstorage.removeItem(‘’)，localstorage.clear ()

使用场景：内容稳定的资源，base64格式的图片，不经常跟新的js,css等（Local）

会话级别相关的资源（session）

1. IndexedDB

\*运行在浏览器上的非关系型数据库

\*字符串，二进制

\*核心API:

let request = window.indexedDB.open --- 创建数据库

request.onsuccess = funcition(){db = envent.target.result} --- 成功获取实例

objectStore ---表

PWA --- webapp的框架

1. CDN (content delivery newwork)

网络内容分发。一组分布在不同地区的服务器。提高响应速度。

两个核心：缓存和回源

缓存：把资源copy到cdn服务器的过程

回源：发现缓存数据过期了，向根服务器拉去资源的过程。

根服务器----业务服务器

CND服务器 ----存放静态资源服务器，不需要业务服务器计算而来的资源，js,css,图片等

CND应用：业务服务器与CND服务器用不同的域，避免cookie带来的不必要的网络开销。

1. 渲染层面，浏览器端的性能优化----对浏览器底层原理、运行机制的理解
2. 服务器端渲染（SSR）
3. 什么是服务器端渲染 --- 页面上呈现的内容在，HTML源文件中找得到。由服务器吧需要的组件或页面渲染成HTML字符串。
4. 解决了什么问题 --- ①网站SEO问提。②首屏加载速度过慢的问题
5. 应用实例：
   1. react

import {renderToString} from ‘react-dom/server’

const tDom = renderToString(<tDom/>)

\* renderToString 是将虚拟dom转换成真实dom的关键方法。

b) vue

**const** renderer = require('vue-server-renderer').createRenderer()

renderer.renderToString(vue实例, (err, html)=>{})

灵魂操作：①把虚拟dom转化成真实dom字符串 ②将字符串放进模板。实际上就是在node上跑了一遍。

\*服务器稀少而宝贵。SSR虽好，但流量上来，非常吃硬件资源。

\*所以在实际中对于SEO和首屏加载优先考虑其他维度的方案。没招的情况下，再去搞SSR。

1. 浏览器端渲染
   1. 浏览器运行机制
2. 渲染引擎和js引擎
3. 渲染引擎 == 内核==chrome(blink) 、FF(gecko)、IE(Trident)、wekit(safari)

渲染功能模块包括：

HTML解释器（DOM树）---parseHTML

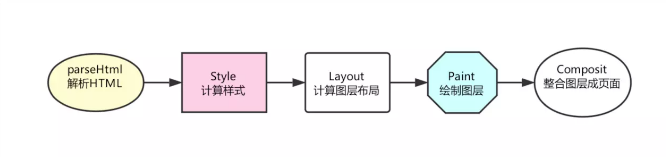
CSS解释器（样式规则）----style

图层布局计算模块（位置和大小）---Layout

视图绘制模块（将像素渲染到屏幕上）---Paint

JS引擎：执行编译js代码

1. 初次渲染过程解析



parseHTML --- 加载解析逻辑，发出资源请求

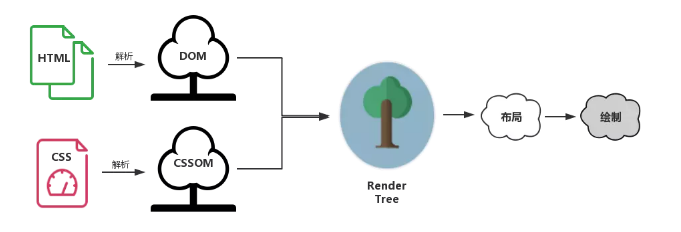
style --- 加载所有css样式信息，结合dom生成render树

layout --- 元素位置，大小计算

paint --- 页面图层转化成像素

整合图层，得到页面---合并各个图层，CPU将数据输出给GPU,最终绘制到屏幕

1. 几颗重要的“树”



CSSOM的解析过程和DOM是并行的

布局渲染树：从根节点递归调用

* 1. css性能方案

css 选择符是从右到左进行匹配的 ---- #dd li{}：浏览器遍历页面上的所有li元素，并且每次缺当前li元素的父元素id

1. 避免使用通配符，只对需要用到的元素做选择
2. 关注可以通过继承实现的属性，避免重复定义后带来的重复匹配。
3. 少用标签选择器，用类选择器代替。
4. 减少嵌套（深度最多三层）。后代选择器的开销是最多的，尽可能使用类来关联元素
   1. css与js加载顺序优化
5. css是阻塞渲染的资源。需要尽快,早的下载到客户端，以便缩短首次渲染时间。

--- 浏览器在构建CSSOM的时候，不会渲染任何已处理的内容。

1. JS引擎是独立渲染引擎而存在的。当HTML解析器遇到script标签时，浏览器会暂停渲染过程，去解析JS代码。

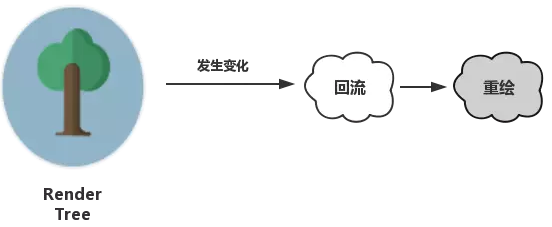
---------------------

JS的三种加载方式

1. 正常模式 --- JS加载和执行是同步的，JS会阻塞浏览器
2. async模式 – JS加载是异步的，JS不会阻塞浏览器。与DOM关联不大用
3. defer模式 – JS加载是异步的，执行是被推迟的。与DOM关联大时用
4. DOM优化
   1. 原理与基本思路

原理：

1. JS操作DOM属于“跨界操作”，开销大 ---- JS引擎与渲染引擎是独立的
2. DOM修改引发样式的更迭 ---- 回流（重排）与重绘



回流：DOM修改引发几何尺寸变化时，浏览器会重新计算元素位置和大小的过程叫做回流（重排）

重绘：DOM修改引发样式变化而并未影响几何变化时，浏览器会重新绘制新的样式的过程叫重绘。

1. 重绘不一定导致回流，但回流一定会导致重绘。

\*\*重回，回流都是在“该元素位于页面上的”前提下才会发生的。

------------------------------------------------------------------------------------------------------------------

优化：

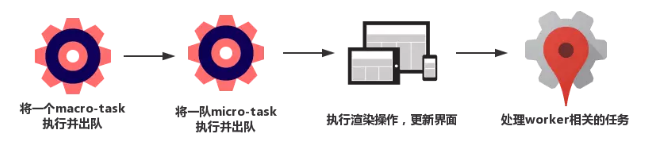
1. 减少DOM操作 --- 缓存变量
2. 减少不必要的DOM操作，而导致的过度渲染（循环）。让JS去给DOM解压（domFragment）
   1. 事件循环（Event Loop）与异步更新
3. JS引擎
   1. JS为什么是单线程？为什么需要异步？异步如何实现

i、不是单线程就会存在同时编辑同一个dom的情况，会产生矛盾。

ii、js自上而下执行，有代码阻塞情况而用户体验差

iii、通过Event Loop实现异步。

* 1. Event Loop
     1. 主线程空闲时循环执行宏队列和微队列任务的过程。
     2. macro: XHR回调\setTimeout\setInterval\setImmediate\script\键盘等IO操作\UI渲染等 --- 宿主发起的任务
     3. micro: process.nexTick\Promise\MutationObserver等 ---- JS引擎发起的任务
     4. 宏任务和微任务关系：每个macro宏任务会维护一个micro微任务表。
     5. 每一次循环都是下面一个过程



\*宏队列是一个个，微队列是一队队，worker有就执行，没就不执行。

\*不是每轮循环都执行渲染，更新界面。浏览器通常以60帧的速率刷新页面即每帧16.7ms。如果让用户觉得顺畅，单个macroTask即他相关的microTask最好都能在16.7ms内完成。

* 1. 异步任务中实现DOM修改时，建议包装成micro任务，尽可能靠近渲染时机。
  2. setTimeout(…, 3)的解释：3秒后，且主线程空闲，同时满足才会执行该函数
  3. 参考https://segmentfault.com/a/1190000018951319?utm\_source=tag-newest#articleHeader5

1. 生产实践：异步更新策略
   1. Vue和React 的数据更新接口，更新并不会立即生效，而是推入异步队列，批量触发。这就是异步更新。
   2. 异步更新可以避免过度渲染。
   3. vue状态更新：nextTick

异步任务默认情况下都是promise包装起来的。callbacks为任务组。

宏派发：setTimeout(flushCallbacks, 0)

微派发：promise().then(flushCallbacks)

* 1. react状态更新：setState
  2. 回流与重绘

1. 引起回流的操作
   1. 最“贵”的操作：改变 DOM 元素的几何属性 ---父子兄弟节点都会重新计算
   2. “价格适中”的操作：改变 DOM 树的结构 ---不会影响前面已经遍历过的元素
   3. 最容易被忽略的操作：获取一些特定属性的值---即时计算
2. 优化方法
   1. JS变量的形式缓存起来
   2. 避免逐条改变样式，使用类名代替
   3. 将DOM“离线”操作---display：’none’
   4. flush队列。--- 浏览器批处理优化，但并不是所有浏览器都可以。
3. 首屏渲染提速：懒加载（大厂面试题）

思路：div占位，出现在可视区域瞬间，div元素被填充。

实现：当前可视区高度document.documentElement.clientHeight，元素距离可视区顶部的高度 getBoundingClientRect()返回元素大小及相对视口位置或者

1. 性能监测
2. 可视化工具
   1. Performance
   2. Lighthouse
3. w3c性能api