

从输入URL到页面加载的过程？如何由一道题完善自己的前端知识体系！

2018-03-12

前言

见解有限，如有描述不当之处，请帮忙指出，如有错误，会及时修正。

为什么要梳理这篇文章？

最近恰好被问到这方面的问题，尝试整理后发现，这道题的覆盖面可以非常广，很适合作为一道承载知识体系的题目。

关于这道题目的吐槽暂且不提（这是一道被提到无数次的题，得到不少人的赞同，也被很多人反感），本文的目的是如何借助这道题梳理自己的前端知识体系！

窃认为，每一个前端人员，如果要往更高阶发展，必然会将自己的知识体系梳理一遍，没有牢固的知识体系，无法往更高处走！

展现形式：本文并不是将所有的知识点列一遍，而是偏向于分析+梳理

内容：在本文中只会梳理一些比较重要的前端向知识点，其它的可能会被省略

目标：本文的目标是梳理一个较为完整的前端向知识体系

本文是个人阶段性梳理知识体系的成果，然后加以修缮后发布成文章，因此并不确保适用于所有人员，但是，个人认为本文还是有一定参考价值的

另外，如有不同见解，可以一起讨论

———超长文预警，需要花费大量时间。———

本文适合有一定经验的前端人员，**新手请规避。**

本文内容超多，建议先了解主干，然后分成多批次阅读。

本文是**前端向**，以前端领域的知识为重点

大纲

- 对知识体系进行一次预评级
- 为什么说知识体系如此重要？
- 梳理主干流程
- 从浏览器接收url到开启网络请求线程
 - 多进程的浏览器
 - 多线程的浏览器内核
 - 解析URL
 - 网络请求都是单独的线程
 - 更多
- 开启网络线程到发出一个完整的http请求
 - DNS查询得到IP
 - tcp/ip请求
 - 五层因特网协议栈
- 从服务器接收到请求到对应后台接收到请求
 - 负载均衡

- 后台的处理
- 后台和前台的http交互
 - http报文结构
 - cookie以及优化
 - gzip压缩
 - 长连接与短连接
 - http 2.0
 - https
- 单独拎出来的缓存问题，http的缓存
 - 强缓存与弱缓存
 - 缓存头部简述
 - 头部的区别
- 解析页面流程
 - 流程简述
 - HTML解析，构建DOM
 - 生成CSS规则
 - 构建渲染树
 - 渲染
 - 简单层与复合层

- Chrome中的调试
- 资源外链的下载
- loaded和domcontentloaded
- CSS的可视化格式模型
 - 包含块（Containing Block）
 - 控制框（Controlling Box）
 - BFC（Block Formatting Context）
 - IFC（Inline Formatting Context）
 - 其它
- JS引擎解析过程
 - JS的解释阶段
 - JS的预处理阶段
 - JS的执行阶段
 - 回收机制
- 其它
- 总结

对知识体系进行一次预评级

看到这道题目，不借助搜索引擎，自己的心里是否有一个答案？

这里，以目前的经验（了解过一些处于不同阶段的相关前端人员的情况），大概有以下几种情况：（以下都是以点见面，实际上不同阶段人员

一般都会有其它的隐藏知识点的)

level1:

完全没什么概念的，支支吾吾的回答，一般就是这种水平（大致形象点描述）：

- 浏览器发起请求，服务端返回数据，然后前端解析成网页，执行脚本。。。

这类人员一般都是：

- 萌新（刚接触前端的，包括0-6个月都有可能会有这种回答）
- 沉淀人员（就是那种可能已经接触了前端几年，但是仍然处于初级阶段的那种。。。）

当然了，后者一般还会偶尔提下http、后台、浏览器渲染，js引擎等等关键字，但基本都是一详细的问就不知道了。。。

level2:

已经有初步概念，但是可能没有完整梳理过，导致无法形成一个完整的体系，或者是很多细节都不会展开，大概是这样子的：（可能符合若干条）

- 知道浏览器输入url后会有http请求这个概念
- 有后台这个概念，大致知道前后端的交互，知道前后端只要靠http报文通信
- 知道浏览器接收到数据后会进行解析，有一定概念，但是具体流程不熟悉（如render树构建流程，layout、paint，复合层与简单层，常用优化方案等不是很熟悉）
- 对于js引擎的解析流程有一定概念，但是细节不熟悉（如具体的形参，函数，变量提升，执行上下文以及VO、AO、作用域链，回收机

制等概念不是很熟悉)

- 如可能知道一些http规范初步概念，但是不熟悉（如http报文结构，常用头部，缓存机制，http2.0，https等特性，跨域与web安全等不是很熟悉)

到这里，看到这上面一大堆的概念后，心里应该也会有点底了。。。

实际上，大部分的前端人员可能都处于**level2**，但是，跳出这个阶段并不容易，一般需要积累，不断学习，才能水到渠成

这类人员一般都是：

- 工作1-3年左右的普通人员（占大多数，而且大多数人员工作3年左右并没有实质上的提升)
- 工作3年以上的老人（这部分人大多都业务十分娴熟，一个当好几个用，但是，基础比较薄弱，可能没有尝试写过框架、组件、脚手架等)

大部分的初中级都陷在这个阶段，如果要突破，不断学习，积累，自然能水到渠成，打通任督二脉

level3:

基本能到这一步的，不是高阶就是接近高阶，因为很多概念并不是靠背就能理解的，而要理解这么多，需形成体系，一般都需要积累，非一日之功。

一般包括什么样的回答呢？（这里就以自己的简略回答进行举例），一般这个阶段的人员都会符合若干条（不一定全部，当然可能还有些是这里遗漏的）：

- 首先略去那些键盘输入、和操作系统交互、以及屏幕显示原理、网卡等硬件交互之类的（前端向中，很多硬件原理暂时略去。。。）

- 对浏览器模型有整体概念，知道浏览器是多进程的，浏览器内核是多线程的，清楚进程与线程之间的区别，以及输入url后会开一个新的网络线程
- 对从开启网络线程到发出一个完整的http请求中间的过程有所了解（如dns查询，tcp/ip链接，五层因特网协议栈等等，以及一些优化方案，如dns-prefetch）
- 对从服务器接收到请求到对应后台接收到请求有一定了解（如负载均衡，安全拦截以及后台代码处理等）
- 对后台和前台的http交互熟悉（包括http报文结构，场景头部，cookie，跨域，web安全，http缓存，http2.0，https等）
- 对浏览器接收到http数据包后的解析流程熟悉（包括解析html，词法分析然后解析成dom树、解析css生成css规则树、合并成render树，然后layout、painting渲染、里面可能还包括复合图层的合成、GPU绘制、外链处理、加载顺序等）
- 对JS引擎解析过程熟悉（包括JS的解释，预处理，执行上下文，VO，作用域链，this，回收机制等）

可以看到，上述包括了一大堆的概念，仅仅是偏前端向，而且没有详细展开，就已经如此之多的概念了，所以，个人认为如果没有自己的见解，没有形成自己的知识体系，仅仅是看看，背背是没用的，过一段时间就会忘光了。

再说下一般这个阶段的都可能是什么样的人吧。（不一定准确，这里主要是靠少部分现实以及大部分推测得出）

- 工作2年以上的前端（基本上如果按正常进度的话，至少接触前端两年左右才会开始走向高阶，当然，现在很多都是上学时就开始学了的，还有部分是天赋异禀，不好预估。。。)
- 或者是已经十分熟悉其它某门语言，再转前端的人（基本上是很快就

可以将前端水准提升上去)

一般符合这个条件的都会有各种隐藏属性（如看过各大框架、组件的源码，写过自己的组件、框架、脚手架，做过大型项目，整理过若干精品博文等）

level4:

由于本人层次尚未达到，所以大致说下自己的见解吧。

一般这个层次，很多大佬都并不仅仅是某个技术栈了，而是成为了技术专家，技术leader之类的角色。所以仅仅是回答某个技术问题已经无法看出水准了，可能更多的要看架构，整体把控，大型工程构建能力等等

不过，对于某些执着于技术的大佬，大概会有一些回答吧：（猜的）

- 从键盘谈起到系统交互，从浏览器到CPU，从调度机制到系统内核，从数据请求到二进制、汇编，从GPU绘图到LCD显示，然后再分析系统底层的进程、内存等等

总之，从软件到硬件，到材料，到分子，原子，量子，薛定谔的猫，人类起源，宇宙大爆炸，平行宇宙？感觉都毫无违和感。。。

这点可以参考下本题的原始出处：

<http://fex.baidu.com/blog/2014/05/what-happen/>

为什么说知识体系如此重要？

为什么说知识体系如此重要呢？这里举几个例子

假设有被问到这样一道题目（随意想到的一个）：

- 如何理解 `getComputedStyle`

在尚未梳理知识体系前，大概会这样回答：

- 普通版本：`getComputedStyle`会获取当前元素所有最终使用的CSS属性值（最终计算后的结果），通过`window.getComputedStyle`等价于`document.defaultView.getComputedStyle`调用
- 详细版本：`window.getComputedStyle(elem, null).getPropertyValue("height")`可能的值为`100px`，而且，就算是CSS上写的是`inherit`，`getComputedStyle`也会把它最终计算出来的。不过注意，如果元素的背景色透明，那么`getComputedStyle`获取出来的就是透明的这个背景（因为透明本身也是有效的），而不会是父节点的背景。所以它不一定是最终显示的颜色。

就这个API来说，上述的回答已经比较全面了。

但是，其实它是可以继续延伸的。

譬如现在会这样回答：

- `getComputedStyle`会获取当前元素所有最终使用的CSS属性值，`window.`和`document.defaultView.`等价...
- `getComputedStyle`会引起回流，因为它需要获取祖先节点的一些信息进行计算（譬如宽高等），所以用的时候慎用，回流会引起性能问题。然后合适的话会将话题引导回流，重绘，浏览器渲染原理等等。当然也可以列举一些其它会引发回流的操作，如`offsetXXX`，`scrollXXX`，`clientXXX`，`currentStyle`等等

再举一个例子：

- `visibility: hidden`和`display: none`的区别

可以如下回答：

- 普通回答，一个隐藏，但占据位置，一个隐藏，不占据位置
- 进一步，`display`由于隐藏后不占据位置，所以造成了dom树的改

变，会引发回流，代价较大

- 再进一步，当一个页面某个元素经常需要切换`display`时如何优化，
一般会用复合层优化，或者要求低一点用`absolute`让其脱离普通文档流也行。然后将话题引到普通文档流，`absolute`文档流，复合图层的区别，
- 再进一步可以描述下浏览器渲染原理以及复合图层和普通图层的绘制区别（复合图层单独分配资源，独立绘制，性能提升，但是不能过多，还有隐式合成等等）

上面这些大概就是知识系统化后的回答，会更全面，容易由浅入深，而且一有机会就可以往更底层挖

前端向知识的重点

此部分的内容是站在个人视角分析的，并不是说就一定是正确答案

首先明确，计算机方面的知识是可以无穷无尽的挖的，而本文的重点是梳理前端向的重点知识

对于前端向（这里可能没有提到`node.js`之类的，更多的是指客户端前端），这里将知识点按重要程度划分成以下几大类：

- 核心知识，必须掌握的，也是最基础的，譬如浏览器模型，渲染原理，JS解析过程，JS运行机制等，作为骨架来承载知识体系
- 重点知识，往往每一块都是一个知识点，而且这些知识点都很重要，譬如http相关，web安全相关，跨域处理等
- 拓展知识，这一块可能更多的是了解，稍微实践过，但是认识上可能没有上面那么深刻，譬如五层因特网协议栈，hybrid模式，移动原生开发，后台相关等等（当然，在不同领域，可能有某些知识就上升到重点知识层次了，譬如hybrid开发时，懂原生开发是很重要的）

为什么要按上面这种方式划分？

这大概与个人的技术成长有关。

记得最开始学前端知识时，是一点一点的积累，一个知识点一个知识点的攻克。

就这样，虽然在很长一段时间内积累了不少的知识，但是，总是无法将它串联到一起。每次梳理时都是很分散的，无法保持思路连贯性。

直到后来，在将浏览器渲染原理、JS运行机制、JS引擎解析流程梳理一遍后，感觉就跟打通了任督二脉一样，有了一个整体的架构，以前的知识点都连贯起来了。

梳理出了一个知识体系，以后就算再学新的知识，也会尽量往这个体系上靠拢，环环相扣，更容易理解，也更不容易遗忘

梳理主干流程

回到这道题上，如何回答呢？先梳理一个骨架

知识体系中，最重要的是骨架，脉络。有了骨架后，才方便填充细节。所以，先梳理下主干流程：

1. 从浏览器接收url到开启网络请求线程（这一部分可以展开浏览器的机制以及进程与线程之间的关系）
2. 开启网络线程到发出一个完整的http请求（这一部分涉及到dns查询，tcp/ip请求，五层因特网模型）
3. 从服务器接收到请求到对应后台接收到请求（这一部分可能涉及到负载均衡，安全拦截以及后台处理）
4. 后台和前台的http交互（这一部分包括http头部、响应码、报文结构、cookie等知识，可以提前提前讲）
5. 单独拎出来的缓存问题，http的缓存（这部分包括http缓存头部，etag，cache-control等）
6. 浏览器接收到http数据包后的解析流程（解析html-词法分析然后解析成dom树、解析css生成cssom）
7. css的可视化格式模型（元素的渲染规则，如包含块，控制框，BFC，IFC等概念）
8. Js引擎解析过程（Js的解释阶段，预处理阶段，执行阶段生成执行上下文，vo，作用域链、回收）

9. 其它（可以拓展不同的知识模块，如跨域，web安全，hybrid模式等等内容）

梳理出主干骨架，然后就需要往骨架上填充细节内容

从浏览器接收url到开启网络请求线程

这一部分展开的内容是：浏览器进程/线程模型，JS的运行机制

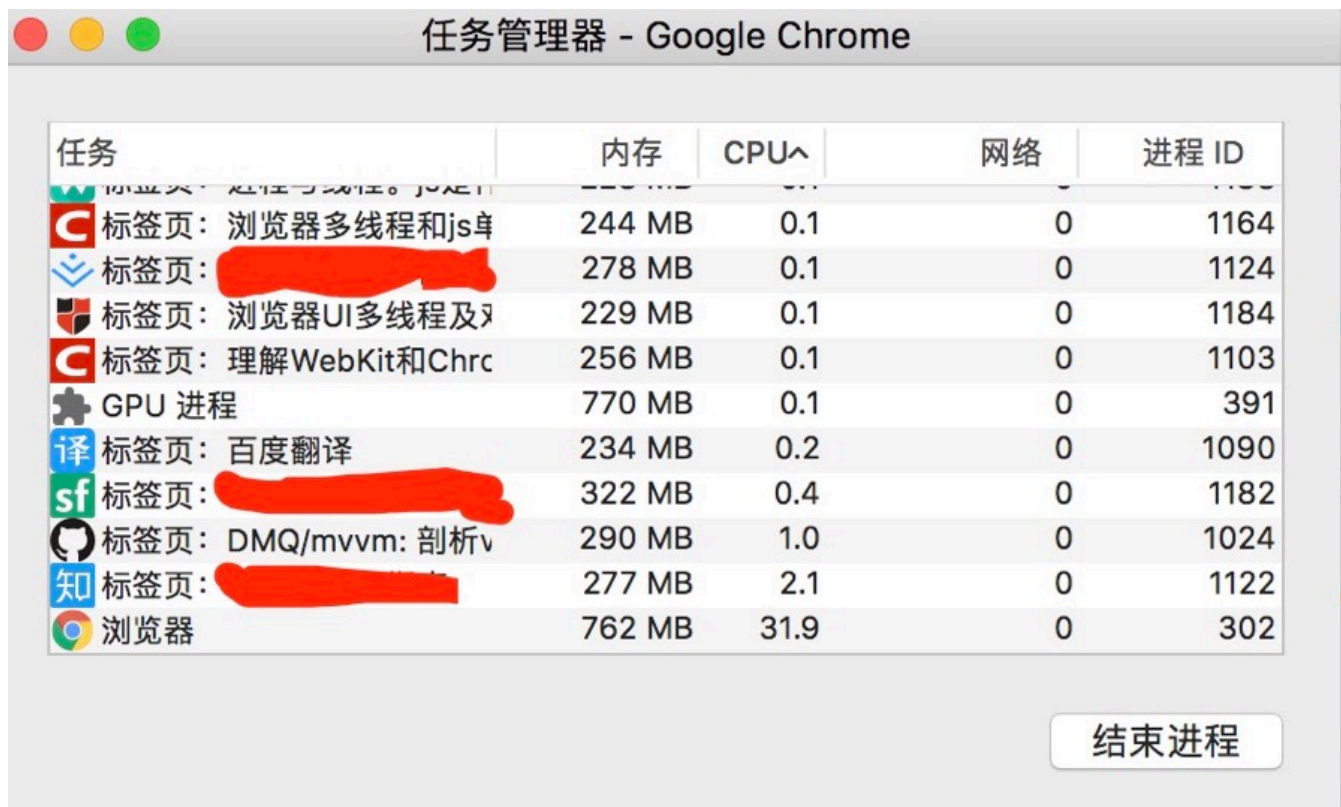
多进程的浏览器

浏览器是多进程的，有一个主控进程，以及每一个tab页面都会新开一个进程（某些情况下多个tab会合并进程）

进程可能包括主控进程，插件进程，GPU，tab页（浏览器内核）等等

- Browser进程：浏览器的主进程（负责协调、主控），只有一个
- 第三方插件进程：每种类型的插件对应一个进程，仅当使用该插件时才创建
- GPU进程：最多一个，用于3D绘制
- 浏览器渲染进程（内核）：默认每个Tab页面一个进程，互不影响，控制页面渲染，脚本执行，事件处理等（有时候会优化，如多个空白tab会合并成一个进程）

如下图：



任务管理器 - Google Chrome

任务	内存	CPU^	网络	进程 ID
标签页: 浏览器多线程和js单	244 MB	0.1	0	1164
标签页: [REDACTED]	278 MB	0.1	0	1124
标签页: 浏览器UI多线程及	229 MB	0.1	0	1184
标签页: 理解WebKit和Chrc	256 MB	0.1	0	1103
GPU 进程	770 MB	0.1	0	391
标签页: 百度翻译	234 MB	0.2	0	1090
标签页: [REDACTED]	322 MB	0.4	0	1182
标签页: DMQ/mvvm: 剖析v	290 MB	1.0	0	1024
标签页: [REDACTED]	277 MB	2.1	0	1122
浏览器	762 MB	31.9	0	302

结束进程

多线程的浏览器内核

每一个tab页面可以看作是浏览器内核进程，然后这个进程是多线程的，它有几大类子线程

- GUI线程
- JS引擎线程
- 事件触发线程
- 定时器线程
- 网络请求线程

浏览器内核



可以看到，里面的JS引擎是内核进程中的一个线程，这也是为什么常说JS引擎是单线程的

解析URL

输入URL后，会进行解析（URL的本质就是统一资源定位符）

URL一般包括几大部分：

- **protocol**，协议头，譬如有http，ftp等
- **host**，主机域名或IP地址

- **port**，端口号
- **path**，目录路径
- **query**，即查询参数
- **fragment**，即#后的hash值，一般用来定位到某个位置

网络请求都是单独的线程

每次网络请求时都需要开辟单独的线程进行，譬如如果URL解析到http协议，就会新建一个网络线程去处理资源下载

因此浏览器会根据解析出得协议，开辟一个网络线程，前往请求资源（这里，暂时理解为是浏览器内核开辟的，如有错误，后续修复）

更多

由于篇幅关系，这里就大概介绍一个主干流程，关于浏览器的进程机制，更多可以参考以前总结的一篇文章（因为内容实在过多，里面包括JS运行机制，进程线程的详解）

[从浏览器多进程到JS单线程，JS运行机制最全面的一次梳理](#)

开启网络线程到发出一个完整的http请求

这一部分主要包括：**dns**查询，**tcp/ip**请求构建，五层因特网协议栈等等

仍然是先梳理主干，有些详细的过程不展开（因为展开的话内容过多）

DNS查询得到IP

如果输入的是域名，需要进行dns解析成IP，大致流程：

- 如果浏览器有缓存，直接使用**浏览器缓存**，否则使用**本机缓存**，再没有的话就是用**host**

- 如果本地没有，就向dns域名服务器查询（当然，中间可能还会经过路由，也有缓存等），查询到对应的IP

注意，域名查询时有可能是经过了CDN调度器的（如果有cdn存储功能的话）

而且，需要知道dns解析是很耗时的，因此如果解析域名过多，会让首屏加载变得过慢，可以考虑dns-prefetch优化

这一块可以深入展开，具体请去网上搜索，这里就不占篇幅了（网上可以看到很详细的解答）

tcp/ip请求

http的本质就是tcp/ip请求

需要了解3次握手规则建立连接以及断开连接时的四次挥手

tcp将http长报文划分为短报文，通过三次握手与服务端建立连接，进行可靠传输

三次握手的步骤：（抽象派）

客户端：hello，你是server么？

服务端：hello，我是server，你是client么

客户端：yes，我是client

建立连接成功后，接下来就正式传输数据

然后，待到断开连接时，需要进行四次挥手（因为是全双工的，所以需要四次挥手）

四次挥手的步骤：（抽象派）

主动方：我已经关闭了向你那边的主动通道了，只能被动接收了

被动方：收到通道关闭的信息

被动方：那我也告诉你，我这边向你的主动通道也关闭了

主动方：最后收到数据，之后双方无法通信

tcp/ip的并发限制

浏览器对同一域名下并发的tcp连接是有限制的（2-10个不等）

而且在http1.0中往往一个资源下载就需要对应一个tcp/ip请求

所以针对这个瓶颈，又出现了很多的资源优化方案

get和post的区别

get和post虽然本质都是tcp/ip，但两者除了在http层面外，在tcp/ip层面也有区别。

get会产生一个tcp数据包，post两个

具体就是：

- get请求时，浏览器会把**headers**和**data**一起发送出去，服务器响应200（返回数据），
- post请求时，浏览器先发送**headers**，服务器响应100 continue，浏览器再发送**data**，服务器响应200（返回数据）。

再说一点，这里的区别是**specification**（规范）层面，而不是**implementation**（对规范的实现）

五层因特网协议栈

其实这个概念挺难记全的，记不全没关系，但是要有一个整体概念

其实就是一个概念：从客户端发出**http**请求到服务器接收，中间会经过一系列的流程。

简括就是：

从应用层的发送http请求，到传输层通过三次握手建立tcp/ip连接，再到网络层的ip寻址，再到数据链路层的封装成帧，最后到物理层的利用物理介质传输。

当然，服务端的接收就是反过来的步骤

五层因特网协议栈其实就是：

- 1.应用层(dns,http) DNS解析成IP并发送http请求
- 2.传输层(tcp,udp) 建立tcp连接（三次握手）
- 3.网络层(IP,ARP) IP寻址
- 4.数据链路层(PPP) 封装成帧
- 5.物理层(利用物理介质传输比特流) 物理传输（然后传输的时候通过双绞线，电磁波等各种介质）

当然，其实也有一个完整的OSI七层框架，与之相比，多了会话层、表示层。

OSI七层框架：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层

表示层：主要处理两个通信系统中交换信息的表示方式，包括数据格式交换，数据加密与解密，数据

会话层：它具体管理不同用户和进程之间的对话，如控制登陆和注销过程

从服务器接收到请求到对应后台接收到请求

服务端在接收到请求时，内部会进行很多的处理

这里由于不是专业的后端分析，所以只是简单的介绍下，不深入

负载均衡

对于大型的项目，由于并发访问量很大，所以往往一台服务器是吃不消的，所以一般会有若干台服务器组成一个集群，然后配合反向代理实现负载均衡

当然了，负载均衡不止这一种实现方式，这里不深入...

简单的说：

用户发起的请求都指向**调度服务器**（**反向代理服务器，譬如安装了nginx控制负载均衡**），然后调度服务器根据实际的调度算法，分配不同的请求给对应集群中的服务器执行，然后调度器等待实际服务器的HTTP响应，并将它反馈给用户

后台的处理

一般后台都是部署到容器中的，所以一般为：

- 先是容器接受到请求（如tomcat容器）
- 然后对应容器中的后台程序接收到请求（如java程序）
- 然后就是后台会有自己的统一处理，处理完后响应响应结果

概括下：

- 一般有的后端是有统一的验证的，如安全拦截，跨域验证
- 如果这一步不符合规则，就直接返回了相应的http报文（如拒绝请求等）
- 然后当验证通过后，才会进入实际的后台代码，此时是程序接收到请求，然后执行（譬如查询数据库，大量计算等等）
- 等程序执行完毕后，就会返回一个http响应包（一般这一步也会经过多层封装）
- 然后就是将这个包从后端发送到前端，完成交互

后台和前台的http交互

前后端交互时，http报文作为信息的载体

所以http是一块很重要的内容，这一部分重点介绍它

http报文结构

报文一般包括了：通用头部，请求/响应头部，请求/响应体

通用头部

这也是开发人员见过的最多的信息，包括如下：

Request Url： 请求的web服务器地址

Request Method： 请求方式

(Get、POST、OPTIONS、PUT、HEAD、DELETE、CONNECT、TRACE)

Status Code： 请求的返回状态码，如200代表成功

Remote Address： 请求的远程服务器地址（会转为IP）

譬如，在跨域拒绝时，可能是method为options，状态码为404/405等
（当然，实际上可能的组合有很多）

其中，Method的话一般分为两批次：

HTTP1.0定义了三种请求方法：GET、POST 和 HEAD方法。

以及几种Additional Request Methods：PUT、DELETE、LINK、UNLINK

HTTP1.1定义了八种请求方法：GET、POST、HEAD、OPTIONS、PUT、DELETE、TRACE 和 CONNECT

HTTP 1.0定义参考：<https://tools.ietf.org/html/rfc1945>

HTTP 1.1定义参考：<https://tools.ietf.org/html/rfc2616>

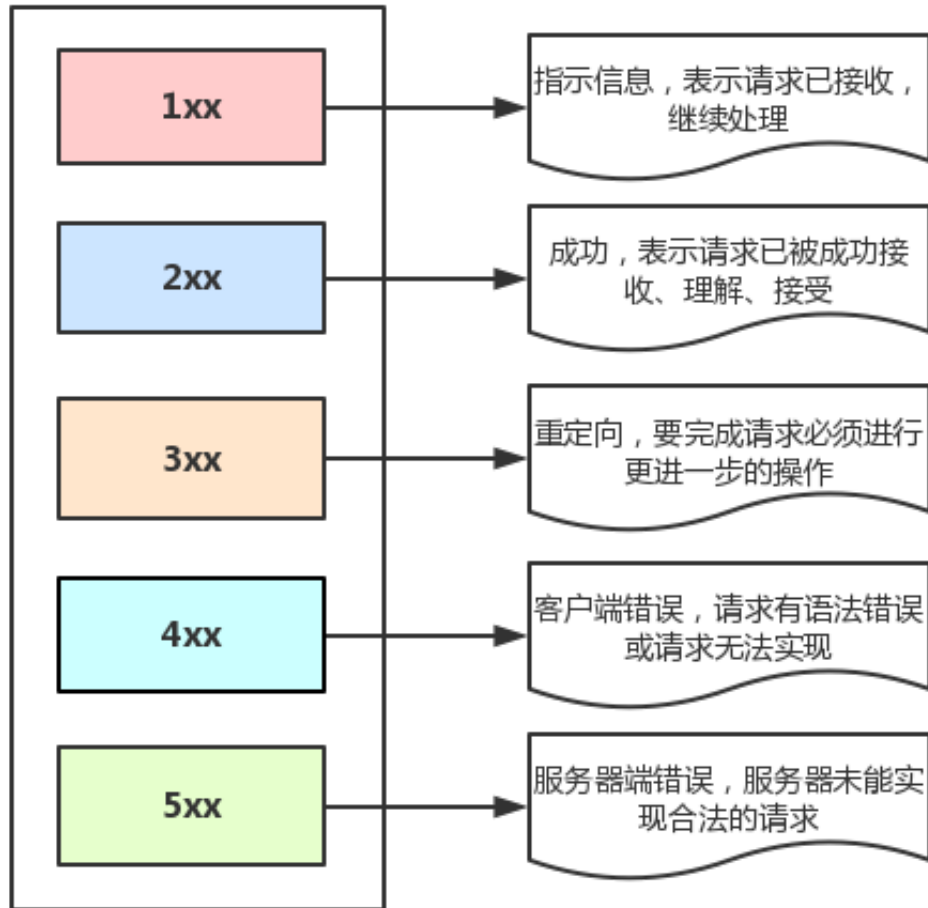
这里面最常用到的就是状态码，很多时候都是通过状态码来判断，如（列举几个最常见的）：

200——表明该请求被成功地完成，所请求的资源发送回客户端
304——自从上次请求后，请求的网页未修改过，请客户端使用本地缓存
400——客户端请求有错（譬如可以是安全模块拦截）
401——请求未经授权
403——禁止访问（譬如可以是未登录时禁止）
404——资源未找到
500——服务器内部错误
503——服务不可用
...

再列举下大致不同范围状态的意义

1xx——指示信息，表示请求已接收，继续处理
2xx——成功，表示请求已被成功接收、理解、接受
3xx——重定向，要完成请求必须进行更进一步的操作
4xx——客户端错误，请求有语法错误或请求无法实现
5xx——服务器端错误，服务器未能实现合法的请求

HTTP Status



总之，当请求出错时，状态码能帮助快速定位问题，完整版本的状态可以自行去互联网搜索

请求/响应头部

请求和响应头部也是分析时常用到的

常用的请求头部（部分）：

Accept：接收类型，表示浏览器支持的MIME类型（对标服务端返回的Content-Type）

Accept-Encoding：浏览器支持的压缩类型，如gzip等，超出类型不能接收

Content-Type: 客户端发送出去实体内容的类型
Cache-Control: 指定请求和响应遵循的缓存机制, 如no-cache
If-Modified-Since: 对应服务端的Last-Modified, 用来匹配看文件是否变动, 只能精确到1s;
Expires: 缓存控制, 在这个时间内不会请求, 直接使用缓存, http1.0, 而且是服务端时间
Max-age: 代表资源在本地缓存多少秒, 有效时间内不会请求, 而是使用缓存, http1.1中
If-None-Match: 对应服务端的ETag, 用来匹配文件内容是否改变 (非常精确), http1.1中
Cookie: 有cookie并且同域访问时会自动带上
Connection: 当浏览器与服务器通信时对于长连接如何处理, 如keep-alive
Host: 请求的服务器URL
Origin: 最初的请求是从哪里发起的 (只会精确到端口), Origin比Referer更尊重隐私
Referer: 该页面的来源URL (适用于所有类型的请求, 会精确到详细页面地址, csrf拦截常用到这)
User-Agent: 用户客户端的一些必要信息, 如UA头部等

常用的响应头部 (部分) :

Access-Control-Allow-Headers: 服务器端允许的请求Headers
Access-Control-Allow-Methods: 服务器端允许的请求方法
Access-Control-Allow-Origin: 服务器端允许的请求Origin头部 (譬如为*)
Content-Type: 服务端返回的实体内容的类型
Date: 数据从服务器发送的时间
Cache-Control: 告诉浏览器或其他客户, 什么环境可以安全的缓存文档
Last-Modified: 请求资源的最后修改时间
Expires: 应该在什么时候认为文档已经过期, 从而不再缓存它
Max-age: 客户端的本地资源应该缓存多少秒, 开启了Cache-Control后有效
ETag: 请求变量的实体标签的当前值
Set-Cookie: 设置和页面关联的cookie, 服务器通过这个头部把cookie传给客户端
Keep-Alive: 如果客户端有keep-alive, 服务端也会有响应 (如timeout=38)
Server: 服务器的一些相关信息

一般来说, 请求头部和响应头部是匹配分析的。

譬如, 请求头部的Accept要和响应头部的Content-Type匹配, 否则会报错

譬如, 跨域请求时, 请求头部的Origin要匹配响应头部的Access-Control-Allow-Origin, 否则会报跨域错误

譬如, 在使用缓存时, 请求头部的If-Modified-Since、If-None-Match分别和响应头部的Last-Modified、ETag对应

还有很多分析方法，这里不一一赘述

请求/响应实体

http请求时，除了头部，还有消息实体，一般来说

请求实体中会将一些需要的参数都放入（用于post请求）。

譬如实体中可以放参数的序列化形式（**a=1&b=2**这种），或者直接放表单对象（**Form Data**对象，上传时可以夹杂参数以及文件），等等

而一般响应实体中，就是放服务端需要传给客户端的内容

一般现在的接口请求时，实体中就是对于的信息的json格式，而像页面请求这种，里面就是直接放了一个html字符串，然后浏览器自己解析并渲染。

CRLF

CRLF（Carriage-Return Line-Feed），意思是回车换行，一般作为分隔符存在

请求头和实体消息之间有一个CRLF分隔，响应头部和响应实体之间用一个CRLF分隔

一般来说（分隔符类别）：

CRLF->Windows-style

LF->Unix Style

CR->Mac Style

如下图是对某请求的http报文结构的简要分析

✖ Headers	Preview	Response	Timing
▼ General Request URL: http://218.18.9.140:8100/EpintOATPFrameWebService/OAWebService/UserLogin_V6 请求的目标URL Request Method: POST 请求方式 Status Code: 200 OK 返回的状态码,200代表成功 Remote Address: 218.18.9.140:8100 请求的目标地址			
▼ Response Headers view source Access-Control-Allow-Headers: X-Requested-With,Content-Type,Accept 接口允许的头部，不能超出范围 Access-Control-Allow-Methods: Get,Post,Put,DELETE,OPTIONS 接口允许的请求类型,非简单请求必需带options Access-Control-Allow-Origin: * 接口允许的请求来源，*代表允许所有 Cache-Control: private 指定请求和响应遵循的缓存机制。private代表仅对当前用户有效，不被其他用户共享 Content-Encoding: gzip 返回文档的编码方法，gzip是一个公认的高效压缩方法 Content-Length: 224 内容长度，当浏览器使用持久http链接时才需要 Content-Type: text/html; charset=utf-8 返回内容的MIME类型 Date: Wed, 22 Mar 2017 09:19:58 GMT 原始服务器消息发出的时间			
Server: Microsoft-IIS/8.5 web服务器名字，一般由服务器自己设置 Vary: Accept-Encoding 告诉代理服务器/缓存/CDN，如何判断请求是否一样，值要么是*要么是header中的key名称组合(服务器判断的依据) X-AspNet-Version: 4.0.30319 .net的版本 X-AspNetMvc-Version: 4.0 X-Powered-By: ASP.NET ASP.NET引擎来处理请求和响应			
▼ Request Headers view source Accept: application/json 指定客户端能接收的内容MIME类型，如果指定json但返回不是，会报错 Accept-Encoding: gzip, deflate 客户端(浏览器)支持的压缩类型,如gzip等,超出类型不能接收 Accept-Language: zh-CN,zh;q=0.8 浏览器支持的语言类型，如zh-CN,zh;q=0.8，并且优先支持靠前的语言类型 Connection: keep-alive 当浏览器与服务器通信时对于长连接如何处理,如keep-alive代表保持连接 Content-Length: 177 Content-Type: text/html; charset=utf-8 发出去的内容的MIME类型 Host: 218.18.9.140:8100 指定请求服务器的域名以及端口号 Origin: http://192.168.114.35:8020 最初的请求是从哪里发起的(只用于POST请求),Origin比Referer更尊重隐私 Referer: http://192.168.114.35:8020/%E8%AF%B7%E6%B1%82%E7%A4%BA%E4%B E%8B/testAjax_test.html 该页面的来源URL(适用于所有类型的请求) User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/50.0.2657.3 Safari/537.36 用户客户端的一些必要信息，如UA头部等 X-Requested-With: XMLHttpRequest 自定义头部，这个头部是自定义加入的			
▼ Form Data view source view URL encoded {"ValidateData":"epointoa@aSPuDzFKrR3A747-hFTgYWav1co: @MTQ3Mjc3NzI 5Ng==","paras":{"LoginID":"admin","Password":"3D4F2BF07DC1BE38B20CD 6E46949A1071F9D0E3D","SoftVersion":"6.0.0"}} 发送给接口的数据，这里直接转成字符串，放在body里面了，一般不同接口要求的格式不一			

cookie以及优化

cookie是浏览器的一种本地存储方式，一般用来帮助客户端和服务端通信的，常用来进行身份校验，结合服务端的session使用。

场景如下（简述）：

在登陆页面，用户登陆了

此时，服务端会生成一个session，session中有对于用户的信息（如用户名、密码等）

然后会有一个sessionid（相当于是服务端的这个session对应的key）

然后服务端在登录页面中写入cookie，值就是：jsessionid=xxx

然后浏览器本地就有这个cookie了，以后访问同域名下的页面时，自动带上cookie，自动检验，在

上述就是cookie的常用场景简述（当然了，实际情况下得考虑更多因素）

一般来说，cookie是不允许存放敏感信息的（千万不要明文存储用户名、密码），因为非常不安全，如果一定要强行存储，首先，一定要在cookie中设置httponly（这样就无法通过js操作了），另外可以考虑rsa等非对称加密（因为实际上，浏览器本地也是容易被攻克的，并不安全）

另外，由于在同域名的资源请求时，浏览器会默认带上本地的cookie，针对这种情况，在某些场景下是需要优化的。

譬如以下场景：

客户端在域名A下有cookie（这个可以是登陆时由服务端写入的）

然后在域名A下有一个页面，页面中有很多依赖的静态资源（都是域名A的，譬如有20个静态资源）

此时就有一个问题，页面加载，请求这些静态资源时，浏览器会默认带上cookie

也就是说，这20个静态资源的http请求，每一个都得带上cookie，而实际上静态资源并不需要cookie

此时就造成了较为严重的浪费，而且也降低了访问速度（因为内容更多了）

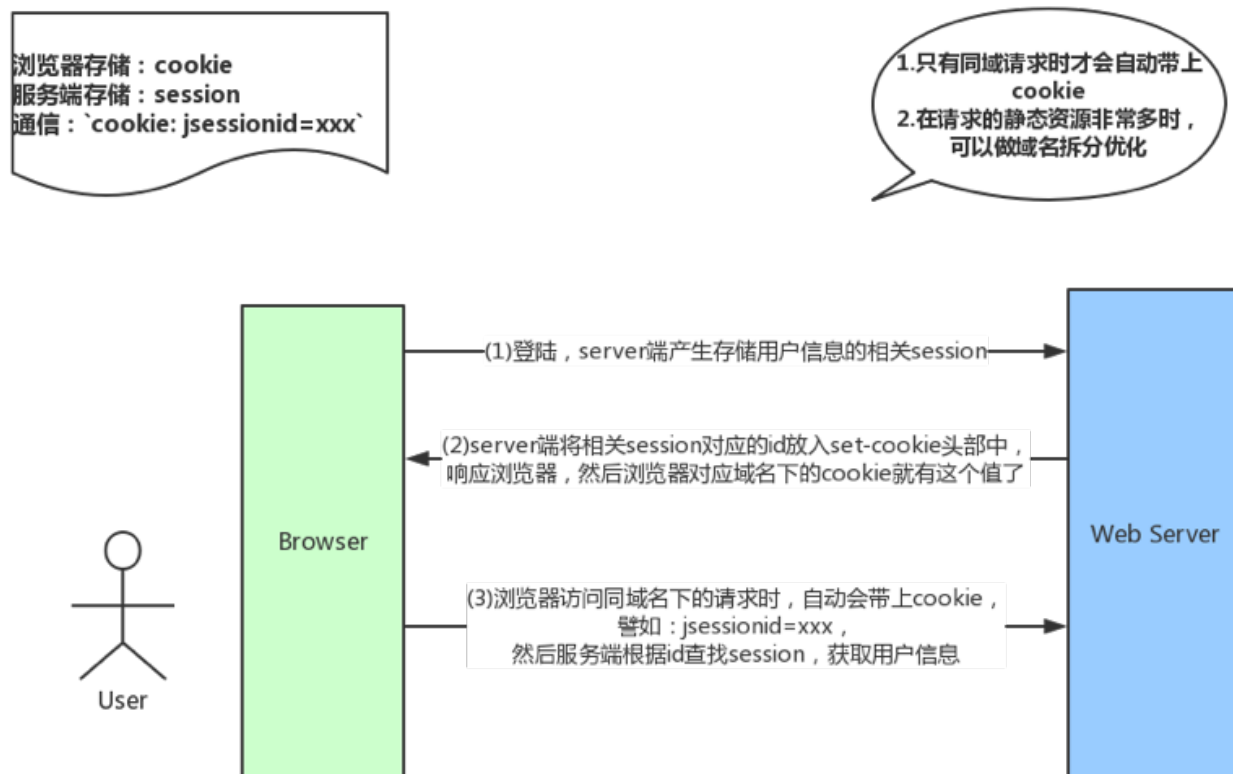
当然了，针对这种场景，是有优化方案的（多域名拆分）。具体做法就是：

- 将静态资源分组，分别放到不同的域名下（如`static.base.com`）
- 而`page.base.com`（页面所在域名）下请求时，是不会带上`static.base.com`域名的cookie的，所以就避免了浪费

说到了多域名拆分，这里再提一个问题，那就是：

- 在移动端，如果请求的域名数过多，会降低请求速度（因为域名整套解析流程是很耗费时间的，而且移动端一般带宽都比不上pc）
- 此时就需要用到一种优化方案：**dns-prefetch**（让浏览器空闲时提前解析dns域名，不过也请合理使用，勿滥用）

关于cookie的交互，可以看下图总结



gzip压缩

首先，明确**gzip**是一种压缩格式，需要浏览器支持才有效（不过一般现在浏览器都支持），而且gzip压缩效率很好（高达70%左右）

然后gzip一般是由**apache**、**tomcat**等web服务器开启

当然服务器除了gzip外，也还会有其它压缩格式（如deflate，没有gzip高效，且不流行）

所以一般只需要在服务器上开启了gzip压缩，然后之后的请求就都是基于gzip压缩格式的，非常方便。

长连接与短连接

首先看**tcp/ip**层面的定义：

- 长连接：一个tcp/ip连接上可以连续发送多个数据包，在tcp连接保持期间，如果没有数据包发送，需要双方发检测包以维持此连接，一般需要自己做在线维持（类似于心跳包）
- 短连接：通信双方有数据交互时，就建立一个tcp连接，数据发送完成后，则断开此tcp连接

然后在http层面：

- **http1.0**中，默认使用的是短连接，也就是说，浏览器没进行一次http操作，就建立一次连接，任务结束就中断连接，譬如每一个静态资源请求时都是一个单独的连接
- **http1.1**起，默认使用长连接，使用长连接会有这一行**Connection: keep-alive**，在长连接的情况下，当一个网页打开完成后，客户端和服务端之间用于传输http的tcp连接不会关闭，如果客户端再次访问这个服务器的页面，会继续使用这一条已经建立的连接

注意：**keep-alive**不会永远保持，它有一个持续时间，一般在服务器中配置（如**apache**），另外长连接需要客户端和服务端都支持时才有效

http 2.0

http2.0不是https，它相当于是http的下一代规范（譬如https的请求可以是http2.0规范的）

然后简述下http2.0与http1.1的显著不同点：

- http1.1中，每请求一个资源，都是需要开启一个tcp/ip连接的，所以对应的结果是，每一个资源对应一个tcp/ip请求，由于tcp/ip本身有并发数限制，所以当资源一多，速度就显著慢下来
- http2.0中，一个tcp/ip请求可以请求多个资源，也就是说，只要一次tcp/ip请求，就可以请求若干个资源，分割成更小的帧请求，速度明显提升。

所以，如果http2.0全面应用，很多http1.1中的优化方案就无需用到了（譬如打包成精灵图，静态资源多域名拆分等）

然后简述下http2.0的一些特性：

- 多路复用（即一个tcp/ip连接可以请求多个资源）
- 首部压缩（http头部压缩，减少体积）
- 二进制分帧（在应用层跟传送层之间增加了一个二进制分帧层，改进传输性能，实现低延迟和高吞吐量）
- 服务器端推送（服务端可以对客户端的一个请求发出多个响应，可以主动通知客户端）
- 请求优先级（如果流被赋予了优先级，它就会基于这个优先级来处理，由服务器决定需要多少资源来处理该请求。）

https

https就是安全版本的http，譬如一些支付等操作基本都是基于https的，

因为http请求的安全系数太低了。

简单来看，https与http的区别就是：在请求前，会建立ssl链接，确保接下来的通信都是加密的，无法被轻易截取分析

一般来说，如果要将网站升级成https，需要后端支持（后端需要申请证书等），然后https的开销也比http要大（因为需要额外建立安全链接以及加密等），所以一般来说http2.0配合https的体验更佳（因为http2.0更快了）

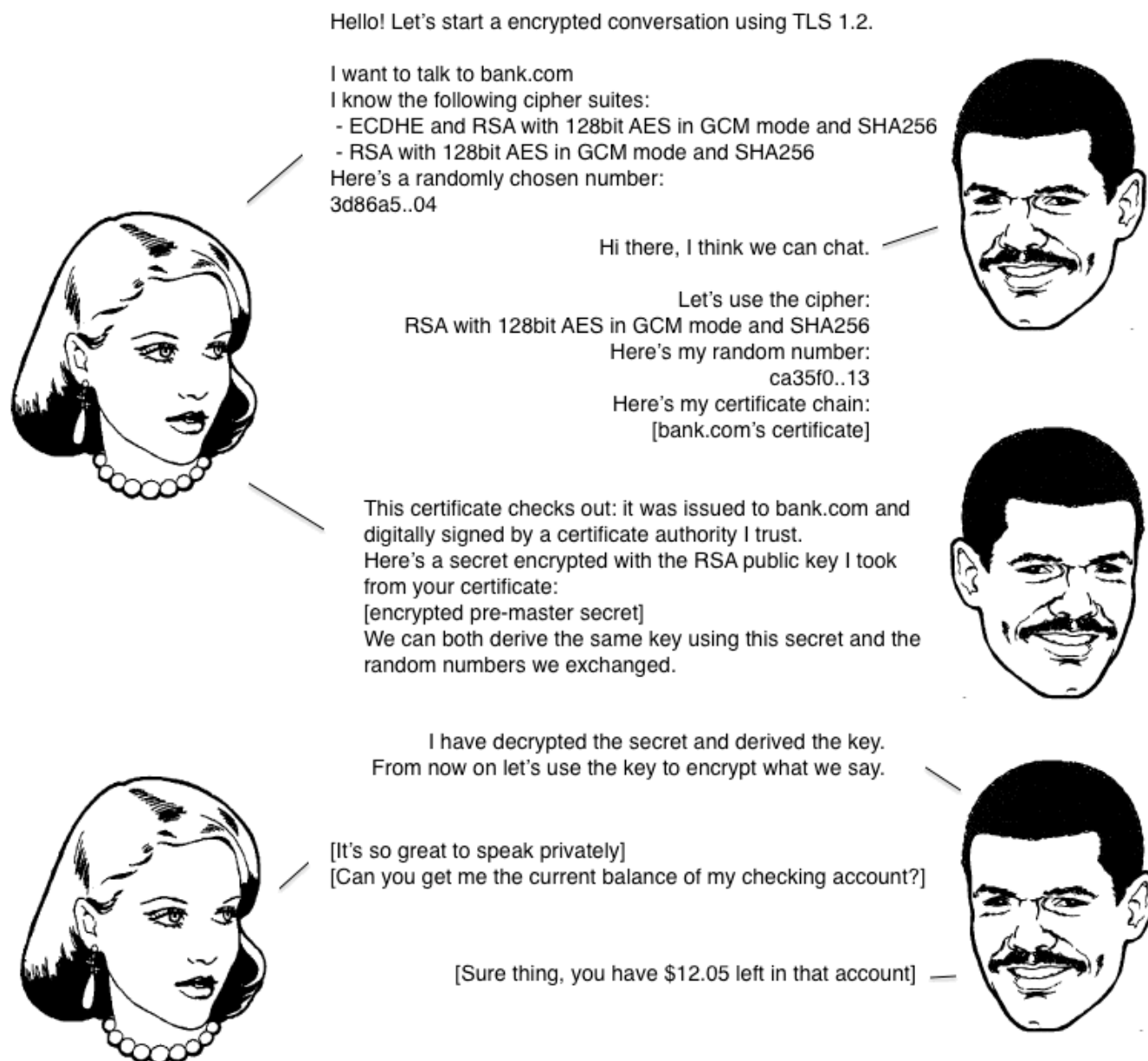
一般来说，主要关注的就是SSL/TLS的握手流程，如下（简述）：

1. 浏览器请求建立SSL链接，并向服务端发送一个随机数—**Client random**和客户端支持的加密方法
2. 服务端从中选出一组加密算法与**Hash**算法，回复一个随机数—**Server random**，并将自己的身份（证书里包含了网站地址，非对称加密的公钥，以及证书颁发机构等信息）
3. 浏览器收到服务端的证书后
 - 验证证书的合法性（颁发机构是否合法，证书中包含的网址是否和正在访问的一样），如果证
 - 用户接收证书后（不管信不信任），浏览会生产新的随机数—**Premaster secret**，然后证
 - 利用**Client random**、**Server random**和**Premaster secret**通过一定的算法生成**HTTP**
 - 使用约定好的**HASH**算法计算握手消息，并使用生成的`session key`对消息进行加密，最后
4. 服务端收到浏览器的回复
 - 利用已知的加解密方式与自己的私钥进行解密，获取`Premaster secret`
 - 和浏览器相同规则生成`session key`
 - 使用`session key`解密浏览器发来的握手消息，并验证**Hash**是否与浏览器发来的一致
 - 使用`session key`加密一段握手消息，发送给浏览器
5. 浏览器解密并计算握手消息的**HASH**，如果与服务端发来的**HASH**一致，此时握手过程结束，

之后所有的https通信数据将由之前浏览器生成的**session key**并利用对称

加密算法进行加密

这里放一张图（来源：[阮一峰-图解SSL/TLS协议](#)）



单独拎出来的缓存问题，http的缓存

前后端的http交互中，使用缓存能很大程度上的提升效率，而且基本上对性能有要求的前端项目都是必用缓存的

强缓存与弱缓存

缓存可以简单的划分成两种类型：**强缓存（200 from cache）**与**协商缓存（304）**

区别简述如下：

- **强缓存（200 from cache）**时，浏览器如果判断本地缓存未过期，就直接使用，无需发起http请求
- **协商缓存（304）**时，浏览器会向服务端发起http请求，然后服务端告诉浏览器文件未改变，让浏览器使用本地缓存

对于协商缓存，使用**Ctrl + F5**强制刷新可以使得缓存无效

但是对于强缓存，在未过期时，必须更新资源路径才能发起新的请求（更改了路径相当于是另一个资源了，这也是前端工程化中常用到的技巧）

缓存头部简述

上述提到了强缓存和协商缓存，那它们是怎么区分的呢？

答案是通过不同的http头部控制

先看下这几个头部：

If-None-Match/E-tag、If-Modified-Since/Last-Modified、Cache-Control/Max-Age、

这些就是缓存中常用到的头部，这里不展开。仅列举下大致使用。

属于强缓存控制的：

(http1.1) **Cache-Control/Max-Age**
(http1.0) **Pragma/Expires**

注意：**Max-Age**不是一个头部，它是**Cache-Control**头部的值

属于协商缓存控制的：

```
(http1.1) If-None-Match/E-tag  
(http1.0) If-Modified-Since/Last-Modified
```

可以看到，上述有提到**http1.1**和**http1.0**，这些不同的头部是属于不同http时期的

再提一点，其实HTML页面中也有一个meta标签可以控制缓存方案-**Pragma**

```
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
```

不过，这种方案还是比较少用到，因为支持情况不佳，譬如缓存代理服务器肯定不支持，所以不推荐

头部的区别

首先明确，http的发展是从http1.0到http1.1

而在http1.1中，出了一些新内容，弥补了http1.0的不足。

http1.0中的缓存控制：

- **Pragma**：严格来说，它不属于专门的缓存控制头部，但是它设置**no-cache**时可以让本地强缓存失效（属于编译控制，来实现特定的指令，主要是因为兼容http1.0，所以以前又被大量应用）
- **Expires**：服务端配置的，属于强缓存，用来控制在规定的時間之前，浏览器不会发出请求，而是直接使用本地缓存，注意，Expires一般对应服务器端时间，如**Expires: Fri, 30 Oct 1998 14:19:41**
- **If-Modified-Since/Last-Modified**：这两个是成对出现的，属于协

商缓存的内容，其中浏览器的头部是**If-Modified-Since**，而服务端的是**Last-Modified**，它的作用是，在发起请求时，如果**If-Modified-Since**和**Last-Modified**匹配，那么代表服务器资源并未改变，因此服务端不会返回资源实体，而是只返回头部，通知浏览器可以使用本地缓存。**Last-Modified**，顾名思义，指的是文件最后的修改时间，而且只能精确到1s以内

http1.1中的缓存控制：

- **Cache-Control**：缓存控制头部，有no-cache、max-age等多种取值
- **Max-Age**：服务端配置的，用来控制强缓存，在规定的时间内，浏览器无需发出请求，直接使用本地缓存，注意，Max-Age是Cache-Control头部的值，不是独立的头部，譬如**Cache-Control: max-age=3600**，而且它值得是绝对时间，由浏览器自己计算
- **If-None-Match/E-tag**：这两个是成对出现的，属于协商缓存的内容，其中浏览器的头部是**If-None-Match**，而服务端的是**E-tag**，同样，发出请求后，如果**If-None-Match**和**E-tag**匹配，则代表内容未变，通知浏览器使用本地缓存，和Last-Modified不同，E-tag更精确，它是类似于指纹一样的东西，基于**FileEtag Inode Mtime Size**生成，也就是说，只要文件变，指纹就会变，而且没有1s精确度的限制。

Max-Age相比Expires？

Expires使用的是服务器端的时间

但是有时候会有这样一种情况-客户端时间和服务端不同步

那这样，可能就会出问题了，造成了浏览器本地的缓存无用或者一直无法过期

所以一般http1.1后不推荐使用**Expires**

而**Max-Age**使用的是客户端本地时间的计算，因此不会有这个问题

因此推荐使用**Max-Age**。

注意，如果同时启用了**Cache-Control**与**Expires**，**Cache-Control**优先级高。

E-tag相比Last-Modified?

Last-Modified:

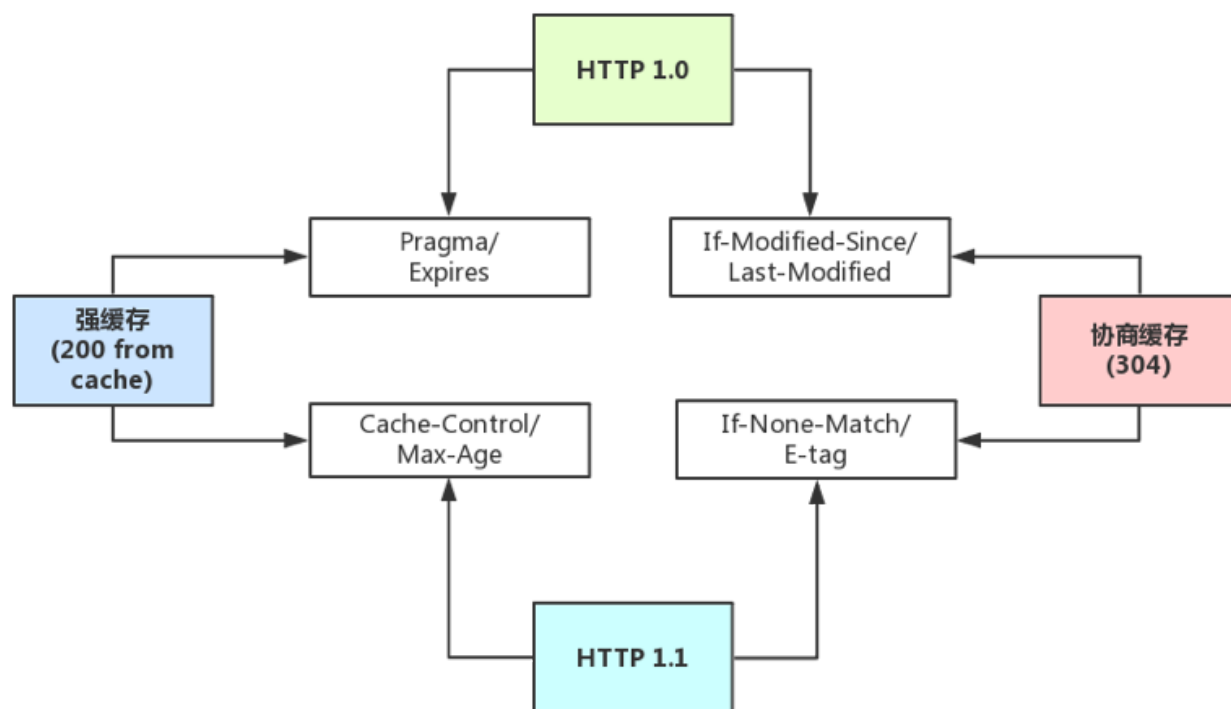
- 表明服务端的文件最后何时改变的
- 它有一个缺陷就是只能精确到1s,
- 然后还有一个问题就是有的服务端的文件会周期性的改变，导致缓存失效

而**E-tag**:

- 是一种指纹机制，代表文件相关指纹
- 只有文件变才会变，也只要文件变就会变，
- 也没有精确时间的限制，只要文件一变，立马E-tag就不一样了

如果同时带有**E-tag**和**Last-Modified**，服务端会优先检查**E-tag**

各大缓存头部的整体关系如下图



解析页面流程

前面有提到http交互，那么接下来就是浏览器获取到html，然后解析，渲染

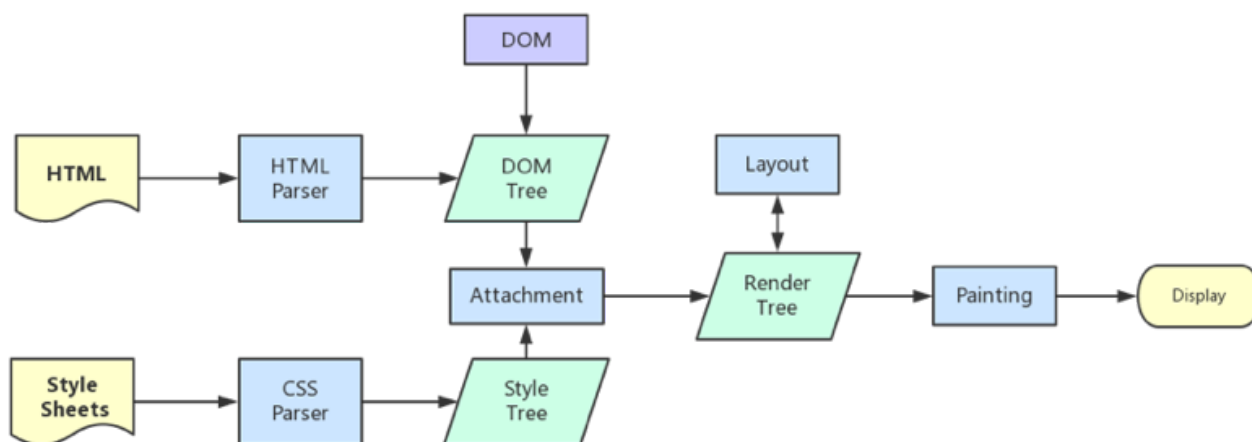
这部分很多都参考了网上资源，特别是图片，参考了来源中的文章

流程简述

浏览器内核拿到内容后，渲染步骤大致可以分为以下几步：

1. 解析HTML，构建DOM树
2. 解析CSS，生成CSS规则树
3. 合并DOM树和CSS规则，生成render树
4. 布局render树 (Layout/reflow)，负责各元素尺寸、位置的计算
5. 绘制render树 (paint)，绘制页面像素信息
6. 浏览器会将各层的信息发送给GPU，GPU会将各层合成 (composite)，显示在屏幕上

如下图：



HTML解析，构建DOM

整个渲染步骤中，HTML解析是第一步。

简单的理解，这一步的流程是这样的：浏览器解析HTML，构建DOM树。

但实际上，在分析整体构建时，却不能一笔带过，得稍微展开。

解析HTML到构建出DOM当然过程可以简述如下：

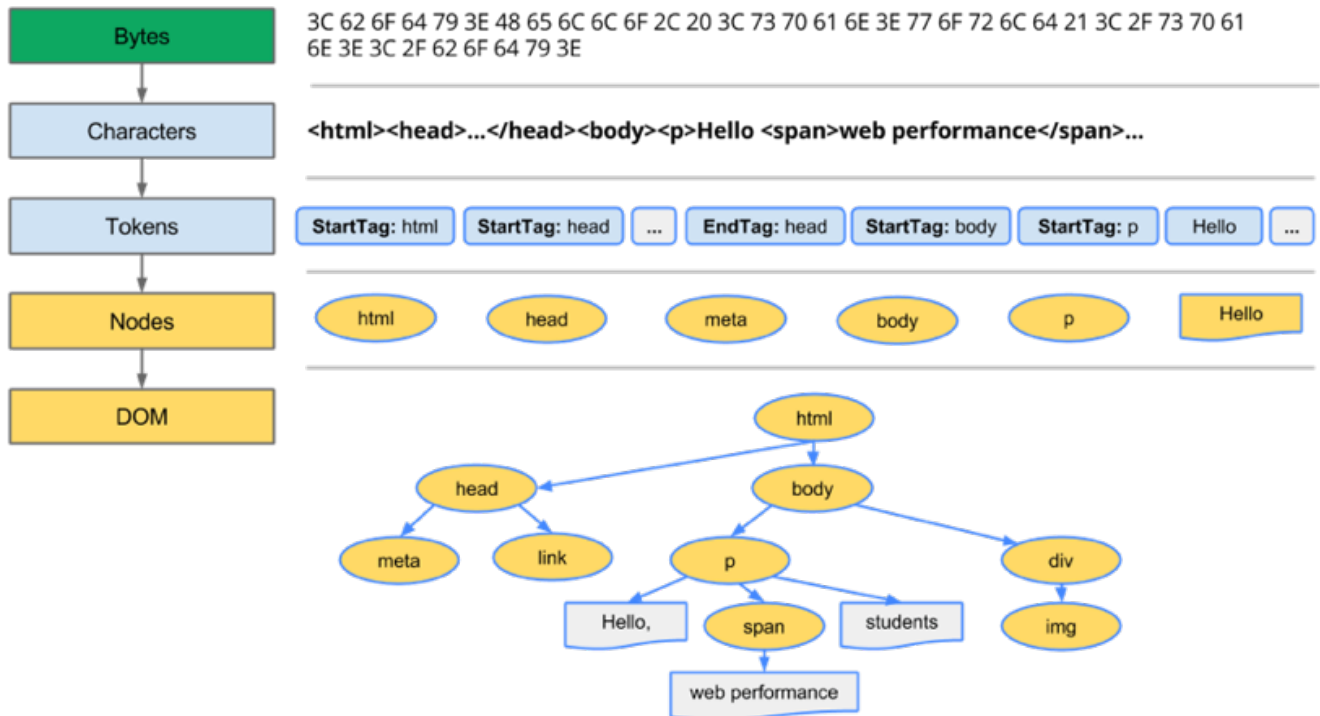
Bytes → characters → tokens → nodes → DOM

譬如假设有这样一个HTML页面：（以下部分的内容出自参考来源，修改了下格式）

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
```

```
<div></div>
</body>
</html>
```

浏览器的处理如下：

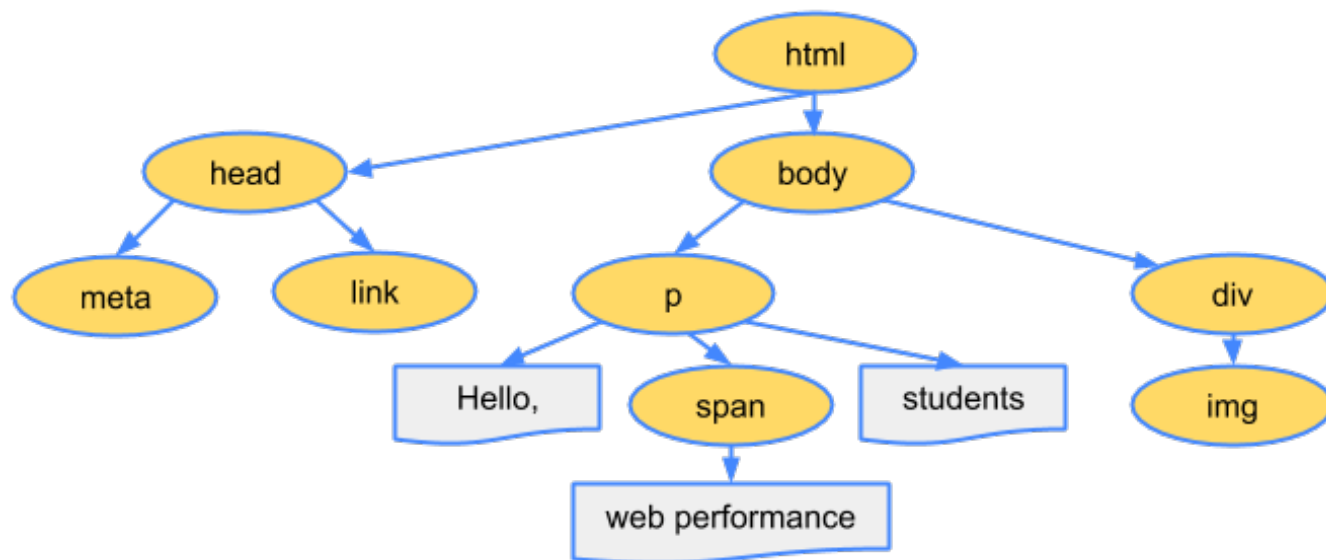


列举其中的一些重点过程：

1. **Conversion**转换：浏览器将获得的HTML内容（Bytes）基于他的编码转换为单个字符
2. **Tokenizing**分词：浏览器按照HTML规范标准将这些字符转换为不同的标记token。每个token
3. **Lexing**词法分析：分词的结果是得到一堆的token，此时把他们转换为对象，这些对象分别定义
4. **DOM**构建：因为HTML标记定义的就是不同标签之间的关系，这个关系就像是一个树形结构一样例如：`body`对象的父节点就是HTML对象，然后段略

对象的父节点就是body对象

最后的DOM树如下：



生成CSS规则

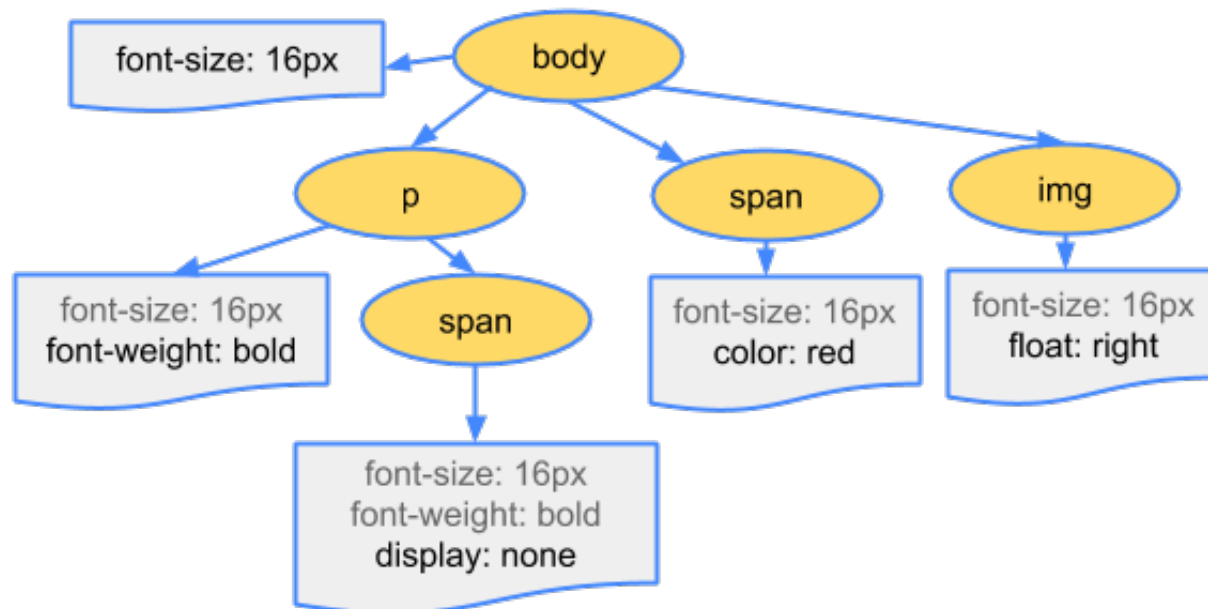
同理，CSS规则树的生成也是类似。简述为：

Bytes → characters → tokens → nodes → CSSOM

譬如style.css内容如下：

```
body { font-size: 16px }  
p { font-weight: bold }  
span { color: red }  
p span { display: none }  
img { float: right }
```

那么最终的CSSOM树就是：



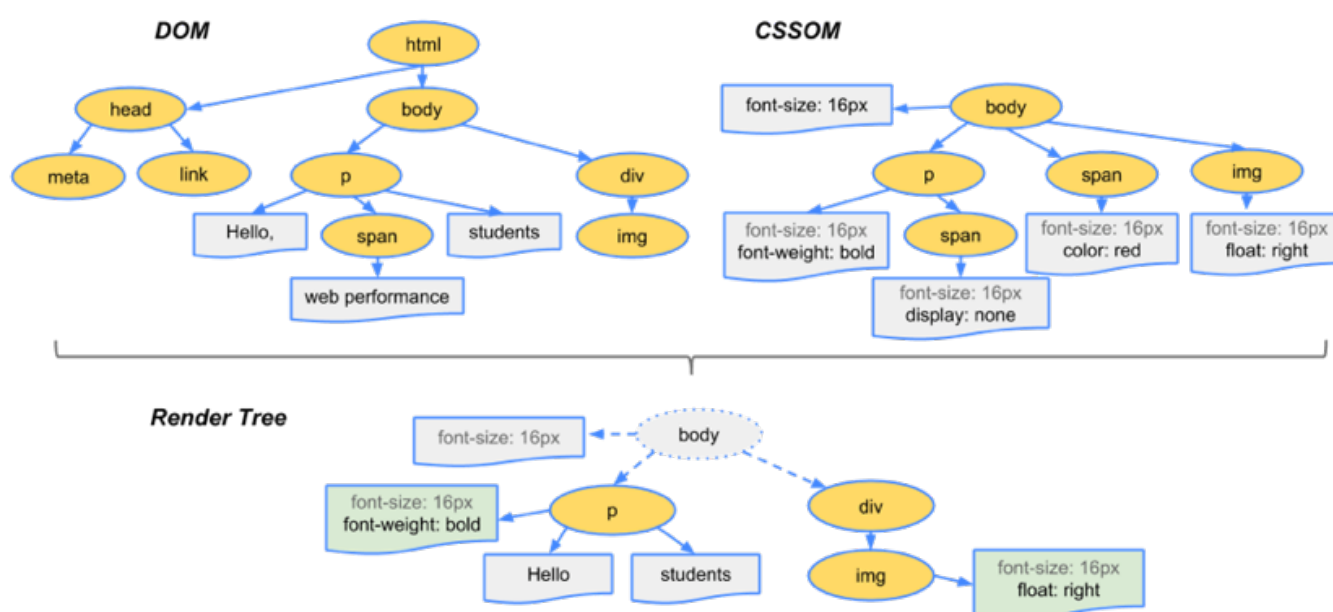
构建渲染树

当DOM树和CSSOM都有了后，就要开始构建渲染树了

一般来说，渲染树和DOM树相对应的，但不是严格意义上的一一对应

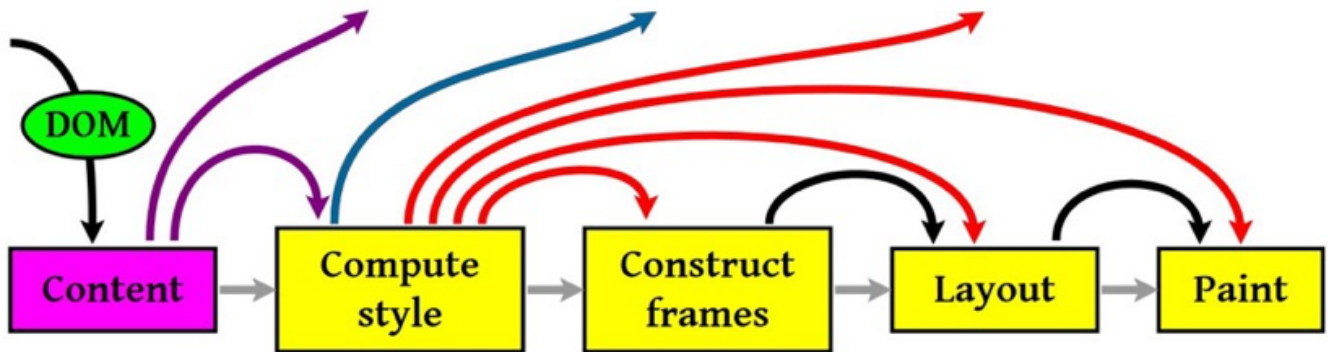
因为有一些不可见的DOM元素不会插入到渲染树中，如head这种不可见的标签或者**display: none**等

整体来说可以看图：



渲染

有了render树，接下来就是开始渲染，基本流程如下：



图中重要的四个步骤就是：

1. 计算css样式
2. 构建渲染树
3. 布局，主要定位坐标和大小，是否换行，各种`position overflow z-index`属性
4. 绘制，将图像绘制出来

然后，图中的线与箭头代表通过js动态修改了DOM或CSS，导致了重新布局（Layout）或渲染（Repaint）

这里Layout和Repaint的概念是有区别的：

- Layout，也称为Reflow，即回流。一般意味着元素的内容、结构、位置或尺寸发生了变化，需要重新计算样式和渲染树
- Repaint，即重绘。意味着元素发生的改变只是影响了元素的一些外观之类的时候（例如，背景色，边框颜色，文字颜色等），此时只需要应用新样式绘制这个元素就可以了

回流的成本开销要高于重绘，而且一个节点的回流往往会导致子节点以及同级节点的回流，所以优化方案中一般都包括，尽量避免回流。

什么会引起回流？

1. 页面渲染初始化

2. DOM结构改变，比如删除了某个节点

3. render树变化，比如减少了padding

4. 窗口resize

5. 最复杂的一种：获取某些属性，引发回流，

很多浏览器会对回流做优化，会等到数量足够时做一次批处理回流，

但是除了render树的直接变化，当获取一些属性时，浏览器为了获得正确的值也会触发回流，这样便

- (1) `offset(Top/Left/Width/Height)`
- (2) `scroll(Top/Left/Width/Height)`
- (3) `client(Top/Left/Width/Height)`
- (4) `width,height`
- (5) 调用了`getComputedStyle()`或者IE的`currentStyle`

回流一定伴随着重绘，重绘却可以单独出现

所以一般会有一些优化方案，如：

- 减少逐项更改样式，最好一次性更改style，或者将样式定义为class并一次性更新
- 避免循环操作dom，创建一个documentFragment或div，在它上面应用所有DOM操作，最后再把它添加到window.document
- 避免多次读取offset等属性。无法避免则将它们缓存到变量
- 将复杂的元素绝对定位或固定定位，使得它脱离文档流，否则回流代价会很高

注意：改变字体大小会引发回流

再来看一个示例：

```
var s = document.body.style;

s.padding = "2px"; // 回流+重绘
s.border = "1px solid red"; // 再一次 回流+重绘
s.color = "blue"; // 再一次重绘
s.backgroundColor = "#ccc"; // 再一次 重绘
s.fontSize = "14px"; // 再一次 回流+重绘
// 添加node, 再一次 回流+重绘
document.body.appendChild(document.createTextNode('abc!'));
```

简单层与复合层

上述中的渲染中止步于绘制，但实际上绘制这一步也没有这么简单，它可以结合复合层和简单层的概念来讲。

这里不展开，进简单介绍下：

- 可以认为默认只有一个复合图层，所有的DOM节点都是在这个复合图层下的
- 如果开启了硬件加速功能，可以将某个节点变成复合图层
- 复合图层之间的绘制互不干扰，由GPU直接控制
- 而简单图层中，就算是absolute等布局，变化时不影响整体的回流，但是由于在同一个图层中，仍然是会影响绘制的，因此做动画时性能仍然很低。而复合层是独立的，所以一般做动画推荐使用硬件加速

更多参考：

[普通图层和复合图层](#)

Chrome中的调试

Chrome的开发者工具中，Performance中可以看到详细的渲染过程：



资源外链的下载

上面介绍了html解析，渲染流程。但实际上，在解析html时，会遇到一些资源连接，此时就需要进行单独处理了

简单起见，这里将遇到的静态资源分为一下几大类（未列举所有）：

- CSS样式资源
- JS脚本资源

- `img`图片类资源

遇到外链时的处理

当遇到上述的外链时，会单独开启一个下载线程去下载资源（http1.1中是每一个资源的下载都要开启一个http请求，对应一个tcp/ip链接）

遇到CSS样式资源

CSS资源的处理有几个特点：

- CSS下载时异步，不会阻塞浏览器构建DOM树
- 但是会阻塞渲染，也就是在构建render时，会等到css下载解析完毕后才进行（这点与浏览器优化有关，防止css规则不断改变，避免了重复的构建）
- 有例外，`media query`声明的CSS是不会阻塞渲染的

遇到JS脚本资源

JS脚本资源的处理有几个特点：

- 阻塞浏览器的解析，也就是说发现一个外链脚本时，需等待脚本下载完成并执行后才会继续解析HTML
- 浏览器的优化，一般现代浏览器有优化，在脚本阻塞时，也会继续下载其它资源（当然有并发上限），但是虽然脚本可以并行下载，解析过程仍然是阻塞的，也就是说必须这个脚本执行完毕后会接下来的解析，并行下载只是一种优化而已
- `defer`与`async`，普通的脚本是会阻塞浏览器解析的，但是可以加上`defer`或`async`属性，这样脚本就变成异步了，可以等到解析完毕后再执行

注意，`defer`和`async`是有区别的：**`defer`**是延迟执行，而**`async`**是异步执

行。

简单的说（不展开）：

- **async**是异步执行，异步下载完毕后就会执行，不确保执行顺序，一定在**onload**前，但不确定在**DOMContentLoaded**事件的前或后
- **defer**是延迟执行，在浏览器看起来的效果像是将脚本放在了**body**后面一样（虽然按规范应该是在**DOMContentLoaded**事件前，但实际上不同浏览器的优化效果不一样，也有可能在他后面）

遇到img图片类资源

遇到图片等资源时，直接就是异步下载，不会阻塞解析，下载完毕后直接用图片替换原有src的地方

loaded和domcontentloaded

简单的对比：

- **DOMContentLoaded** 事件触发时，仅当DOM加载完成，不包括样式表，图片(譬如如果有**async**加载的脚本就不一定完成)
- **load** 事件触发时，页面上所有的DOM，样式表，脚本，图片都已经加载完成了

CSS的可视化格式模型

这一部分内容很多参考《精通**CSS**-高级Web标准解决方案》以及参考来源

前面提到了整体的渲染概念，但实际上文档树中的元素是按什么渲染规则渲染的，是可以进一步展开的，此部分内容即：**CSS的可视化格式模型**

先了解：

- CSS中规定每一个元素都有自己的盒子模型（相当于规定了这个元素如何显示）
- 然后可视化格式模型则是把这些盒子按照规则摆放到页面上，也就是如何布局
- 换句话说，盒子模型规定了怎么在页面里摆放盒子，盒子的相互作用等等

说到底：**CSS的可视化格式模型就是规定了浏览器在页面中如何处理文档树**

关键字：

包含块 (Containing Block)

控制框 (Controlling Box)

BFC (Block Formatting Context)

IFC (Inline Formatting Context)

定位体系

浮动

...

另外，CSS有三种定位机制：普通流，浮动，绝对定位，如无特别提及，下文中都是针对普通流中的

包含块 (Containing Block)

一个元素的box的定位和尺寸，会与某一矩形框有关，这个框就称之为包含块。

元素会为它的子孙元素创建包含块，但是，并不是说元素的包含块就是它的父元素，元素的包含块与它的祖先元素的样式等有关系

譬如：

- 根元素是最顶端的元素，它没有父节点，它的包含块就是初始包含块

- **static**和**relative**的包含块由它最近的块级、单元格或者行内块祖先元素的内容框（**content**）创建
- **fixed**的包含块是当前可视窗口
- **absolute**的包含块由它最近的**position** 属性为**absolute**、**relative**或者**fixed**的祖先元素创建
 - 如果其祖先元素是行内元素，则包含块取决于其祖先元素的**direction**特性
 - 如果祖先元素不是行内元素，那么包含块的区域应该是祖先元素的内边距边界

控制框（**Controlling Box**）

块级元素和块框以及行内元素和行框的相关概念

块框：

- 块级元素会生成一个块框（**Block Box**），块框会占据一整行，用来包含子box和生成的内容
- 块框同时也是一个块包含框（**Containing Box**），里面要么只包含块框，要么只包含行内框（不能混杂），如果块框内部有块级元素也有行内元素，那么行内元素会被匿名块框包围

关于匿名块框的生成，示例：

```
<DIV>
Some text
<P>More text
</DIV>
```

div生成了一个块框，包含了另一个块框**p**以及文本内容**Some text**，此时

Some text文本会被强制加到一个匿名的块框里面，被**div**生成的块框包含（其实这个就是**IFC**中提到的行框，包含这些行内框的这一行匿名块形成的框，行框和行内框不同）

换句话说：

如果一个块框在其中包含另外一个块框，那么我们强迫它只能包含块框，因此其它文本内容生成出来的都是匿名块框（而不是匿名行内框）

行内框：

- 一个行内元素生成一个行内框
- 行内元素能排在一行，允许左右有其它元素

关于匿名行内框的生成，示例：

```
<P>Some <EM>emphasized</EM> text</P>
```

P元素生成一个块框，其中有几个行内框（如**EM**），以及文本**Some**，**text**，此时会专门为这些文本生成匿名行内框

display属性的影响

display的几个属性也可以影响不同框的生成：

- **block**，元素生成一个块框
- **inline**，元素产生一个或多个的行内框
- **inline-block**，元素产生一个行内级块框，行内块框的内部会被当作块块来格式化，而此元素本身会被当作行内级框来格式化（这也是为什么会产生**BFC**）
- **none**，不生成框，不再格式化结构中，当然了，另一个**visibility**：

hidden则会产生一个不可见的框

总结：

- 如果一个框里，有一个块级元素，那么这个框里的内容都会被当作块框来进行格式化，因为只要出现了块级元素，就会将里面的内容分块几块，每一块独占一行（出现行内可以用匿名块框解决）
- 如果一个框里，没有任何块级元素，那么这个框里的内容会被当成行内框来格式化，因为里面的内容是按照顺序成行的排列

BFC（Block Formatting Context）

FC（格式上下文）？

FC即格式上下文，它定义框内部的元素渲染规则，比较抽象，譬如

FC像是一个大箱子，里面装有很多元素

箱子可以隔开里面的元素和外面的元素（所以外部并不会影响**FC**内部的渲染）

内部的规则可以是：如何定位，宽高计算，**margin**折叠等等

不同类型的框参与的FC类型不同，譬如块级框对应BFC，行内框对应IFC

注意，并不是说所有的框都会产生**FC**，而是符合特定条件才会产生，只有产生了对应的**FC**后才会应用对应渲染规则

BFC规则：

在块格式化上下文中

每一个元素左外边与包含块的左边相接触（对于从右到左的格式化，右外边接触右边）

即使存在浮动也是如此（所以浮动元素正常会直接贴近它的包含块的左边，与普通元素重合）

除非这个元素也创建了一个新的**BFC**

总结几点BFC特点：

1. 内部**box**在垂直方向，一个接一个的放置
2. **box**的垂直方向由**margin**决定，属于同一个BFC的两个**box**间的**margin**会重叠
3. BFC区域不会与**float box**重叠（可用于排版）
4. BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此
5. 计算BFC的高度时，浮动元素也参与计算（不会浮动坍塌）

如何触发BFC？

1. 根元素
2. **float**属性不为**none**
3. **position**为**absolute**或**fixed**
4. **display**为**inline-block**, **flex**, **inline-flex**, **table**, **table-cell**, **table-caption**
5. **overflow**不为**visible**

这里提下，**display: table**，它本身不产生BFC，但是它会产生匿名框（包含**display: table-cell**的框），而这个匿名框产生BFC

更多请自行网上搜索

IFC（Inline Formatting Context）

IFC即行内框产生的格式上下文

IFC规则

在行内格式化上下文中

框一个接一个地水平排列，起点是包含块的顶部。

水平方向上的 `margin`，`border` 和 `padding` 在框之间得到保留

框在垂直方向上可以以不同的方式对齐：它们的顶部或底部对齐，或根据其中文字的基线对齐

行框

包含那些框的长方形区域，会形成一行，叫做行框

行框的宽度由它的包含块和其中的浮动元素决定，高度的确定由行高度计算规则决定

行框的规则：

如果几个行内框在水平方向无法放入一个行框内，它们可以分配在两个或多个垂直堆叠的行框中（即：

行框在堆叠时没有垂直方向上的分割且永不重叠

行框的高度总是足够容纳所包含的所有框。不过，它可能高于它包含的最高的框（例如，框对齐会引

行框的左边接触到其包含块的左边，右边接触到其包含块的右边。

结合补充下IFC规则：

浮动元素可能会处于包含块边缘和行框边缘之间

尽管在相同的行内格式化上下文中的行框通常拥有相同的宽度（包含块的宽度），它们可能会因浮动

同一行内格式化上下文中的行框通常高度不一样（如，一行包含了一个高的图形，而其它行只包含文

当一行中行内框宽度的总和小于包含它们的行框的宽，它们在水平方向上的对齐，取决于 `text-align`

空的行内框应该被忽略

即不包含文本，保留空白符，`margin/padding/border`非0的行内元素，以及其他常规流中的内容(比如，图片，`inline blocks` 和 `inline tables`)，并且不是以换行结束的行框，必须被当作零高度行框对待

总结：

- 行内元素总是会应用IFC渲染规则
- 行内元素会应用IFC规则渲染，譬如`text-align`可以用来居中等
- 块框内部，对于文本这类的匿名元素，会产生匿名行框包围，而行框内部就应用IFC渲染规则
- 行内框内部，对于那些行内元素，一样应用IFC渲染规则
- 另外，`inline-block`，会在元素外层产生IFC（所以这个元素是可以通过`text-align`水平居中的），当然，它内部则按照BFC规则渲染

相比BFC规则来说，IFC可能更加抽象（因为没有那么条理清晰的规则和触发条件）

但总的来说，它就是行内元素自身如何显示以及在框内如何摆放的渲染规则，这样描述应该更容易理解

其它

当然还有有一些其它内容：

- 譬如常规流，浮动，绝对定位等区别
- 譬如浮动元素不包含在常规流中
- 譬如相对定位，绝对定位，`Fixed`定位等区别
- 譬如`z-index`的分层显示机制等

这里不一一展开，更多请参考：

<http://bbs.csdn.net/topics/340204423>

JS引擎解析过程

前面有提到遇到JS脚本时，会等到它的执行，实际上是需要引擎解析的，这里展开描述（介绍主干流程）

JS的解释阶段

首先得明确：**JS**是解释型语言，所以它无需提前编译，而是由解释器实时运行

引擎对JS的处理过程可以简述如下：

1. 读取代码，进行词法分析（**Lexical analysis**），然后将代码分解成词元（**token**）
2. 对词元进行语法分析（**parsing**），然后将代码整理成语法树（**syntax tree**）
3. 使用翻译器（**translator**），将代码转为字节码（**bytecode**）
4. 使用字节码解释器（**bytecode interpreter**），将字节码转为机器码

最终计算机执行的就是机器码。

为了提高运行速度，现代浏览器一般采用即时编译（**JIT-Just In Time compiler**）

即字节码只在运行时编译，用到哪一行就编译哪一行，并且把编译结果缓存（**inline cache**）

这样整个程序的运行速度能得到显著提升。

而且，不同浏览器策略可能还不同，有的浏览器就省略了字节码的翻译步骤，直接转为机器码（如chrome的v8）

总结起来可以认为是：**核心的JIT编译器将源码编译成机器码运行**

JS的预处理阶段

上述将的是解释器的整体过程，这里再提下在正式执行JS前，还会有一个预处理阶段（譬如变量提升，分号补全等）

预处理阶段会做一些事情，确保JS可以正确执行，这里仅提部分：

分号补全

JS执行是需要分号的，但为什么以下语句却可以正常运行呢？

```
console.log('a')
console.log('b')
```

原因就是JS解释器有一个[Semicolon Insertion](#)规则，它会按照一定规则，在适当的位置补充分号

譬如列举几条自动加分号的规则：

- 当有换行符（包括含有换行符的多行注释），并且下一个**token**没法跟前面的语法匹配时，会自动补分号。
- 当有}时，如果缺少分号，会补分号。
- 程序源代码结束时，如果缺少分号，会补分号。

于是，上述的代码就变成了

```
console.log('a');
console.log('b');
```

所以可以正常运行

当然了，这里有一个经典的例子：

```
function b() {  
    return  
    {  
        a: 'a'  
    };  
}
```

由于分号补全机制，所以它变成了：

```
function b() {  
    return;  
    {  
        a: 'a'  
    };  
}
```

所以运行后是`undefined`

变量提升

一般包括函数提升和变量提升

譬如：

```
a = 1;  
b();  
function b() {  
    console.log('b');  
}  
var a;
```

经过变量提升后，就变成：

```
function b() {
```



```
    console.log('b');  
  }  
  var a;  
  a = 1;  
  b();
```

这里没有展开，其实展开也可以牵涉到很多内容的

譬如可以提下变量声明，函数声明，形参，实参的优先级顺序，以及es6中let有关的临时死区等

JS的执行阶段

此阶段的内容中的图片来源：[深入理解JavaScript系列（10）：JavaScript核心（晋级高手必读篇）](#)

解释器解释完语法规则后，就开始执行，然后整个执行流程中大致包含以下概念：

- 执行上下文，执行堆栈概念（如全局上下文，当前活动上下文）
- VO（变量对象）和AO（活动对象）
- 作用域链
- this机制等

这些概念如果深入讲解的话内容过多，因此这里仅提及部分特性

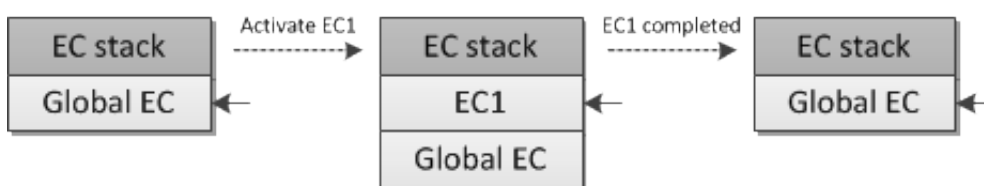
执行上下文简单解释

- JS有执行上下文）
- 浏览器首次载入脚本，它将创建全局执行上下文，并压入执行栈栈顶（不可被弹出）
- 然后每进入其它作用域就创建对应的执行上下文并把它压入执行栈的

顶部

- 一旦对应的上下文执行完毕，就从栈顶弹出，并将上下文控制权交给当前的栈。
- 这样依次执行（最终都会回到全局执行上下文）

譬如，如果程序执行完毕，被弹出执行栈，然后有没有被引用（没有形成闭包），那么这个函数中用到的内存就会被垃圾处理器自动回收



然后执行上下文与VO，作用域链，this的关系是：

每一个执行上下文，都有三个重要属性：

- 变量对象(**variable object**, **VO**)
- 作用域链(**scope chain**)
- **this**

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

VO与AO

VO是执行上下文的属性（抽象概念），但是只有全局上下文的变量对象允许通过VO的属性名称来间接访问（因为在全局上下文里，全局对象本身就是变量对象）

AO (**activation object**)，当函数被调用者激活，AO就被创建了

可以理解为：

- 在函数上下文中： **VO === AO**
- 在全局上下文中： **VO === this === global**

总的来说，VO中会存放一些变量信息（如声明的变量，函数，**arguments** 参数等等）

作用域链

它是执行上下文中的一个属性，原理和原型链很相似，作用很重要。

譬如流程简述：

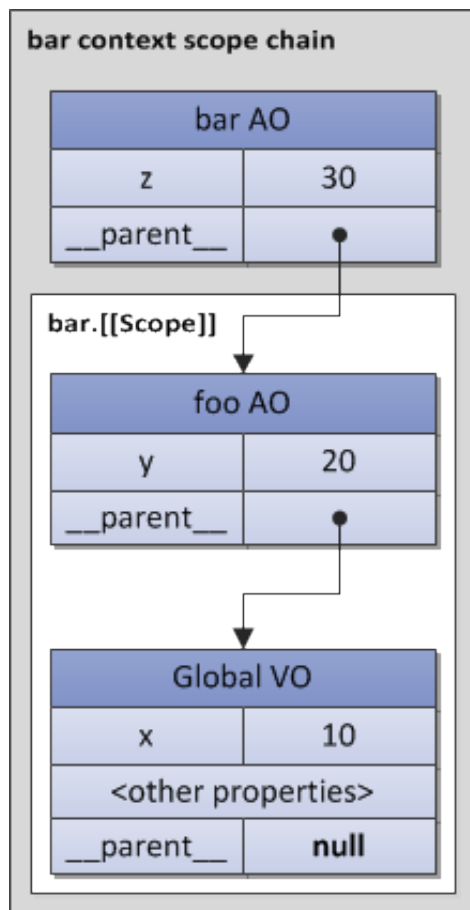
在函数上下文中，查找一个变量**foo**

如果函数的vo中找到了，就直接使用

否则去它的父级作用域链中（**__parent__**）找

如果父级中没找到，继续往上找

直到全局上下文中也没找到就报错



this指针

这也是JS的核心知识之一，由于内容过多，这里就不展开，仅提及部分

注意：**this**是执行上下文环境的一个属性，而不是某个变量对象的属性

因此：

- this是有一个类似搜寻变量的过程
- 当代码中使用了this，这个 this的值就直接从执行的上下文中获取了，而不会从作用域链中搜寻
- this的值只取决中进入上下文时的情况

所以经典的例子：

```
var baz = 200;  
var bar = {
```

```
    baz: 100,
    foo: function() {
        console.log(this.baz);
    }
};
var foo = bar.foo;

// 进入环境: global
foo(); // 200, 严格模式中会报错, Cannot read property 'baz' of undefined

// 进入环境: global bar
bar.foo(); // 100
```

就要明白了上面this的介绍，上述例子很好理解

更多参考：

[深入理解JavaScript系列（13）：This? Yes,this!](#)

回收机制

JS有垃圾处理器，所以无需手动回收内存，而是由垃圾处理器自动处理。

一般来说，垃圾处理器有自己的回收策略。

譬如对于那些执行完毕的函数，如果没有外部引用（被引用的话会形成闭包），则会回收。（当然一般会把回收动作切割到不同的时间段执行，防止影响性能）

常用的两种垃圾回收规则是：

- 标记清除
- 引用计数

Javascript引擎基础GC方案是（**simple GC**）：**mark and sweep**（标记清除），简单解释如下：

1. 遍历所有可访问的对象。
2. 回收已不可访问的对象。

譬如：（出自javascript高程）

当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。

从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。

而当变量离开环境时，则将其标记为“离开环境”。

垃圾回收器在运行的时候会给存储在内存中的所有变量都加上标记（当然，可以使用任何标记方式）。

然后，它会去掉环境中的变量以及被环境中的变量引用的变量的标记（闭包，也就是说在环境中的以及相关引用的变量会被去除标记）。

而在此之后再被加上标记的变量将被视为准备删除的变量，原因是环境中的变量已经无法访问到这些变量了。

最后，垃圾回收器完成内存清除工作，销毁那些带标记的值并回收它们所占用的内存空间。

关于引用计数，简单点理解：

跟踪记录每个值被引用的次数，当一个值被引用时，次数+1，减持时-1，下次垃圾回收器会回收次数为0的值的内存（当然了，容易出循环引用的bug）

GC的缺陷

和其他语言一样，javascript的GC策略也无法避免一个问题：**GC时，停止响应其他操作**

这是为了安全考虑。

而Javascript的GC在100ms甚至以上

对一般的应用还好，但对于JS游戏，动画对连贯性要求比较高的应用，就麻烦了。

这就是引擎需要优化的点：避免GC造成的长时间停止响应。

GC优化策略

这里介绍常用到的：分代回收（Generation GC）

目的是通过区分“临时”与“持久”对象：

- 多回收“临时对象”区（**young generation**）
- 少回收“持久对象”区（**tenured generation**）
- 减少每次需遍历的对象，从而减少每次GC的耗时。

像node v8引擎就是采用的分代回收（和java一样，作者是java虚拟机作者。）

更多可以参考：

[V8 内存浅析](#)

其它

可以提到跨域

譬如发出网络请求时，会用AJAX，如果接口跨域，就会遇到跨域问题

可以参考：

[ajax跨域，这应该是最全的解决方案了](#)

可以提到web安全

譬如浏览器在解析HTML时，有**xSSAuditor**，可以延伸到web安全相关领域

可以参考：

[AJAX请求真的不安全么？谈谈Web安全与AJAX的关系。](#)

更多

如可以提到**viewport**概念，讲讲物理像素，逻辑像素，CSS像素等概念

如熟悉Hybrid开发的话可以提及一下Hybrid相关内容以及优化

...

总结

上述这么多内容，目的是：**梳理出自己的知识体系**

本文由于是前端向，所以知识梳理时有重点，很多其它的知识点都简述或略去了，重点介绍的模块总结：

- 浏览器的进程/线程模型、JS运行机制（这一块的详细介绍链接到了另一篇文章）
- http规范（包括报文结构，头部，优化，http2.0，https等）
- http缓存（单独列出来，因为它很重要）
- 页面解析流程（HTML解析，构建DOM，生成CSS规则，构建渲染树，渲染流程，复合层的合成，外链的处理等）
- JS引擎解析过程（包括解释阶段，预处理阶段，执行阶段，包括执行上下文、VO、作用域链、this、回收机制等）

- 跨域相关，web安全单独链接到了具体文章，其它如CSS盒模型，viewport等仅是提及概念

关于本文的价值？

本文是个人阶段性梳理知识体系的成果，然后加以修缮后发布成文章，因此并不确保适用于所有人员

但是，个人认为本文还是有一定参考价值的

写在最后的话

还是那句话：知识要形成体系

梳理出知识体系后，有了一个骨架，知识点不易遗忘，而且学习新知识时也会更加迅速，更重要的是容易举一反三，可以由一个普通的问题，深挖拓展到底层原理

前端知识是无穷无尽的，本文也仅仅是简单梳理出一个承载知识体系的骨架而已，更多的内容仍然需要不断学习，积累

另外，本文结合[从浏览器多进程到JS单线程，JS运行机制最全面的一次梳理](#)这篇文章，更佳噢！

附录

博客

初次发布2018.03.12于我个人博客上面

<http://www.dailichun.com/2018/03/12/whenyouenteraurl.html>

参考资料

- <https://segmentfault.com/a/1190000012925872>
- <https://www.html5rocks.com/zh/tutorials/internals/howbrowserswor>

[k/](#)

- <https://coolshell.cn/articles/9666.html>
- <http://igoro.com/archive/what-really-happens-when-you-navigate-to-a-url/>
- <http://blog.csdn.net/dojiangv/article/details/51794535>
- <http://bbs.csdn.net/topics/340204423>
- <https://segmentfault.com/a/1190000004246731>
- <http://www.bubuko.com/infodetail-1379568.html>
- <http://fex.baidu.com/blog/2014/05/what-happen/>
- <http://www.cnblogs.com/winter-cn/archive/2013/05/21/3091127.html>
- <https://fanerge.github.io/%E6%B5%8F%E8%A7%88%E5%99%A8%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86-webkit%E5%86%85%E6%A0%B8%E7%A0%94%E7%A9%B6.html>
- <http://www.cnblogs.com/TomXu/archive/2012/01/12/2308594.html>
- <https://segmentfault.com/q/1010000000489803>