

前端优化原理

浏览器运行机制 @王伟平



作为一名前端开发者，每天浏览器陪伴你度过的时光甚至比女朋友陪伴你的都要久，想想那每一个令人“不是那么期待”的早晨，每一个争分夺秒完成任务的黄昏，只有浏览器和编辑器一直是你忠实的伙伴。

今天就来一探究竟，走进这个我们与网络连接最紧密的中间地带

大纲：

- 浏览器的发展简史
- 浏览器的多进程架构
- 浏览器的渲染原理
- JS引擎的运行原理

一、浏览器的发展简史

```
[vite] connecting...  
[vite] connected.  
> navigator.userAgent  
< 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36'  
> |
```



- 1990年：蒂姆·伯纳斯-李（Tim Berners-Lee） WorldWideWeb，第一款Web浏览器
- 1993年：NCSA Mosaic 第一款可以显示图片的浏览器
- 1994年：Netscape Navigator 1.0 (win 3.1) 诞生，**95年** Brendan Eich **开发了** JavaScript
- 1995年：Microsoft 推出IE 1.0 （Internet Explorer Mozilla/1.22）
- 1997年：IE 4.0 vs Netscape 4.0， Internet Explorer 与 Windows 操作系统捆绑发行，此后四年内，IE 获得了 75% 的市场份额，（第一次浏览器大战）。
- 2003年：网景解散 为反垄断 开放Netscape源代码，非营利性质的 Mozilla 诞生
- 2003年：Safari 诞生 Webkit内核, 目前市场上第二大浏览器。
- 2004年：Mozilla Firefox 1.0 诞生，第二次浏览器大战开始。
- 2008年：Netscape最终灭绝，Google Chrome 诞生
- 2012年：Chrome 推出4年后取代 Internet Explorer 成为最受欢迎的浏览器。
- 2015年：Microsoft Edge发布，难掩颓势。截止2021年市场份额仅为5%

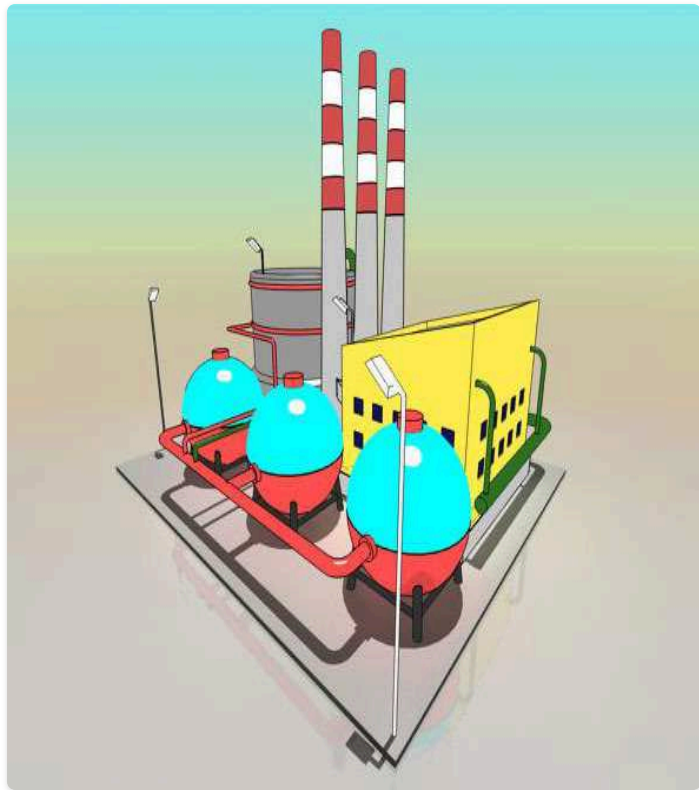
世界历史从不缺少史诗般的权力斗争，有征服世界的暴君，也有落败的勇士。Web 浏览器的历史也大抵如此。学术先驱们编写出引发信息革命的简易软件，并为浏览器的优势和互联网用户而战。

参考链接：

掘金：浏览器简史及其核心原理详解：47 张图带你走进浏览器的世界

Mozilla: Web 浏览器简史

二、浏览器的多进程架构



1. 区分进程和线程

- 系统给进程分配独立的内存空间
- 进程之间相互独立
- 一个进程由一个或多个线程组成
- 多个线程在进程中协助完成任务
- 同一进程下的各个线程之间共享程序的内存空间（包括代码段、数据集、堆等）

Tips

进程是CPU资源**分配**的最小单位(能拥有资源和独立运行的最小单位)

线程是CPU**调度**的最小单位（线程是建立在进程的基础上的运行单位，一个进程可以有多个线程）

一般通用叫法的单线程和多线程，都是指的进程内的线程数量是单个还是多个

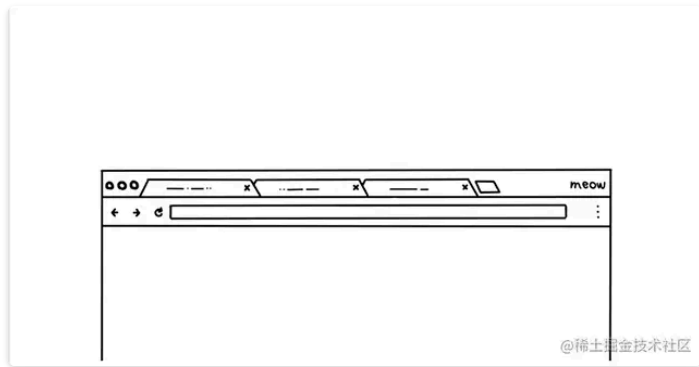
2. 浏览器包含哪些类型的进程？

- 主进程 (Browser 进程)
 - 负责浏览器的界面显示，与用户交互（前进后退按钮）等
 - 负责各个页面的管理（创建和销毁其他进程）
 - 网络资源的管理，下载等
- Renderer进程：浏览器内核（webkit、blink）渲染进程
 - 负责页面的渲染，JS执行，事件处理等
 - 每个tab页一个进程，互不影响
- GPU进程
 - 负责图形、3D绘制
- 插件进程
 - 仅当使用插件时才创建

2. 浏览器多进程的特点？

优点

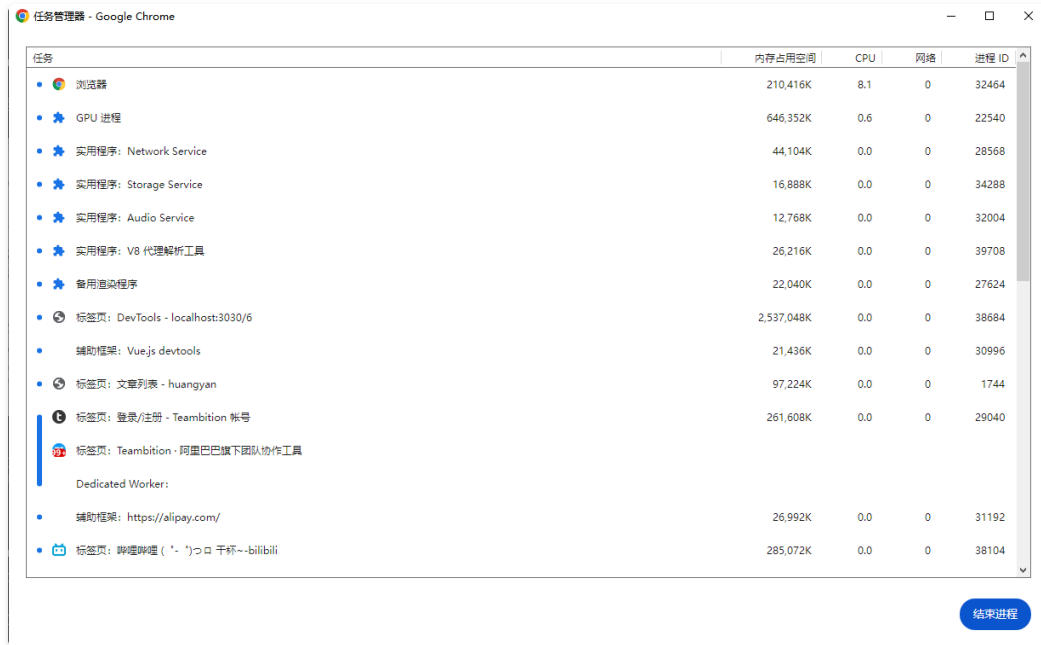
- 避免单个page crash（tab、插件等）影响整个浏览器
- 充分利用多核优势
- 更为安全，在系统层面界定了不同进程的权限



缺点

- 内存消耗比较大，不同进程之间的内存不共享，不同进程的内存需要包含浏览器内核的多份副本

Chrome浏览器的多进程架构



任务	内存占用空间	CPU	网络	进程 ID
浏览器	210,416K	8.1	0	32464
GPU 进程	646,352K	0.6	0	22540
实用程序: Network Service	44,104K	0.0	0	28568
实用程序: Storage Service	16,888K	0.0	0	34288
实用程序: Audio Service	12,768K	0.0	0	32004
实用程序: V8 代理解析工具	26,216K	0.0	0	39708
备用渲染程序	22,040K	0.0	0	27624
标签页: DevTools - localhost:3030/6	2,537,048K	0.0	0	38684
辅助框架: Vue.js devtools	21,436K	0.0	0	30996
标签页: 文章列表 - huangyan	97,224K	0.0	0	1744
标签页: 登录/注册 - Teambition 帐号	261,608K	0.0	0	29040
标签页: Teambition - 阿里巴巴旗下团队协作工具				
Dedicated Worker:				
辅助框架: https://alipay.com/	26,992K	0.0	0	31192
标签页: 哔哩哔哩 ('~'~') 干杯~ - bilibili	285,072K	0.0	0	38104

结束进程

为了节省内存，Chrome限制了最多的进程数，最大进程数量由设备内存和CPU能力决定，当达到这一限制时，Chrome会将共用之前同一个站点的渲染进程。

渲染进程（浏览器内核，主要）

- GUI渲染线程（主线程、工作线程、排版线程、光栅线程、合成器线程等）
 - 负责渲染页面、布局和绘制
 - 页面需要回流或重绘时，该线程就会执行
- JS引擎线程
 - 负责解析和执行 JavaScript 脚本程序
 - 只有一个 JavaScript 引擎线程（单线程）
- 事件触发线程
 - 用来控制事件循环（鼠标点击等）
 - 当事件满足触发条件时，将事件放入到 JS 引擎的执行队列中
- 定时触发器线程
 - 用来控制定时器，如 `setTimeout`、`setInterval` 所在的线程
 - 定时器任务不是由 JS 引擎计时的，而是由定时器触发线程来计时的
 - 当定时器计时完毕后，通知事件触发线程
- 异步 http 请求线程
 - 浏览器有一个单独的线程来处理 Ajax 请求
 - 当请求完成时，若有回调函数，通知事件触发线程

思考两个问题：

1. 为什么 Javascript 是单线程的？
2. 为什么 GUI 渲染线程与 JS 引擎线程互斥？

1. 创建 JS 这门语言的时候，多进程多线程的架构不流行，硬件的支持不好，而且多线程操作时需要加锁，编码的复杂性会增加，且如果多个JavaScript操作同个DOM元素时，浏览器无法判断渲染结果是否符合预期。

2. JS 是可以操作 DOM 的，如果同时修改元素属性并同时渲染界面（即 JS 线程和 UI 线程同时运行），那么渲染线程前后获得的元素就可能不一致了。为了防止渲染出现不可预期的结果，浏览器设定 GUI 渲染线程和 JS 引擎线程为互斥关系，当 JS 引擎线程执行时GUI渲染线程会被挂起，GUI 更新则被保存在一个队列中等待 JS 引擎线程空闲时立即被执行。

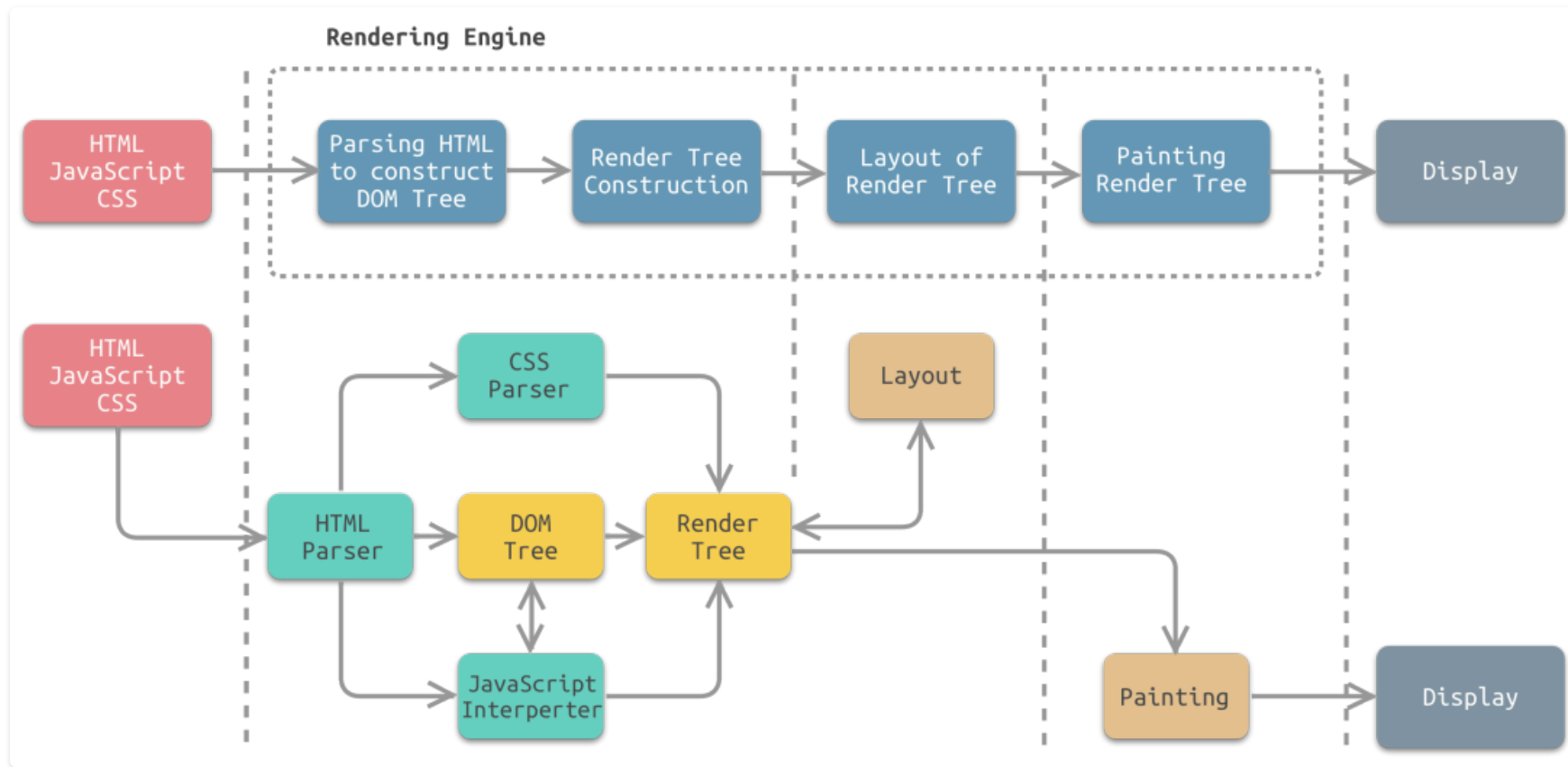
三、浏览器渲染流程

1. 解析和构建 DOM Tree (DOM 树到屏幕图形的转化原理, 其本质就是**树结构**到**层结构**的演化)
2. 解析 CSS 构建 CSSOM Tree , 合成 Render Tree , 同一z轴空间的渲染对象都归并到同一渲染层中 (第一个层模型)
3. 布局 (Layout) 计算每个节点在屏幕中的位置, 生成布局树

三、浏览器渲染流程

4. 合成层：图形层 Graphics Layer（第二个层模型），满足下列条件把渲染层提升为一个合成层
 - 3D transform
 - 对子元素使用了 will-change: transform 属性的元素
 - 使用加速视频解码的 <video> 元素，拥有3D (WebGL) 上下文或加速的2D上下文的 <canvas> 元素
 - 对 opacity、transform、filter 属性使用了 animation 或 transition 的元素
5. 绘制：合成层的绘制，生成每一个合成层的绘制记录
6. 栅格化：将合成层分为图块，每个图块转化为合成帧（位图），作为纹理通过GPU进程存储在GPU的显存中。
7. 合成与显示：合成帧随后会通过 IPC 协议将消息传递给浏览器主进程。浏览器收到消息后，会将页面内容绘制到内存中。最后再将内存中的内容显示在屏幕上。

三、浏览器渲染流程



注意隐式合成

一个或多个非合成元素应出现在堆叠顺序上的合成元素之上，被提升到合成层

1. 两个 absolute 定位的 div 在屏幕上交叠了，根据 z-index 的关系，其中一个层级较高的 div 会“盖在”另外一个的上边。
2. 这个时候，如果处于下方的 div 被加上了CSS属性： `transform: translateZ(0)`，就会被浏览器提升到合成层。提升后的合成层位于Document上方，假如没有隐式合成，原本应该处于上方的div就依然跟Document共用一个Graphics Layer，层级反而降了，就出现了元素交叠关系错乱的问题。
3. 所以为了纠正错误的交叠顺序，浏览器必须让原本应该“盖在”它上面的渲染层也同时提升为合成层。

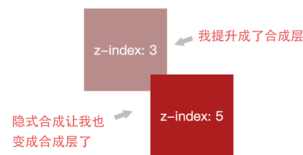


WecTeam



我提升成了合成层

WecTeam



隐式合成让我也
变成合成层了

WecTeam

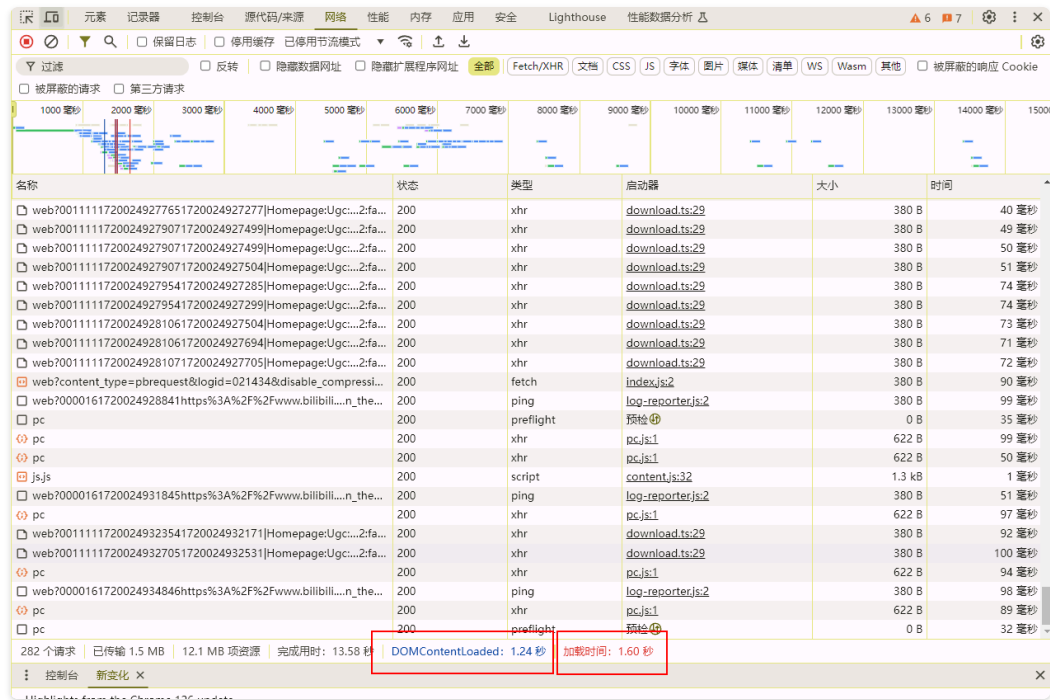
注意隐式合成

结论：使用3D硬件加速提升动画性能时，最好给元素增加一个 `z-index` 属性，人为干扰复合层的排序，可以有效减少Chrome创建不必要的复合层，提升渲染性能，移动端优化尤为明显。

Q1: DOMContentLoaded 和 Load 事件的区别?

DOMContentLoaded : 仅当DOM加载完成, 不包括样式表、图片等外部资源加载完成;

Load : 当所有资源加载完成, 包括样式表、图片等外部资源加载完成;



Q2: CSS加载是否会阻塞页面加载?

```
<!--@format-->

<!DOCTYPE html>
<html lang="en">
  <head>
    <!--<meta charset="UTF-8" /-->
    <!--<meta name="viewport" content="width=device-width, initial-scale=1.0" /-->
    <!--<title>Document</title>
    <!--<style>
    <!--<h1 {
    <!--< color: red !important;
    <!--< }
    <!--< /style>
    <!--< script>
    <!--< function h() {
    <!--< console.log(document.querySelectorAll('h1'));
    <!--< }
    <!--< setTimeout(() => {
    <!--< h();
    <!--< }, 0);
    <!--< /script>
    <!--<!-- Latest compiled and minified CSS -->
    <!--< link
    <!--< rel="stylesheet"
    <!--< href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css"
    <!--< integrity="sha384-BVYi1SIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
    <!--< crossorigin="anonymous"
    <!--< />
    <!--< /head>
    <!--< body>
    <!--< <h1>这是红色</h1>
    <!--< /body>
  <!--< /html>
```

实验1: 弱网下

现象: CSS没加载出来之前, 页面开始白屏, 直到CSS加载完成之后, 红色字体才显示出来, 但是控制台有输出

Q2: CSS加载是否会阻塞页面加载?

实验2: 弱网下

现象: 位于CSS加载语句前的JS代码先执行了, 但位于CSS加载语句后的代码却迟迟没有执行, 直到CSS加载完成后, 它才开始执行

```
1 <!--@format-->
2
3 <!DOCTYPE html>
4 <html lang="en">
5 <head>
6 <<< <meta charset="UTF-8" />
7 <<< <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8 <<< <title>Document</title>
9 <<< <style>
10 <<<< h1 {
11 <<<<< color: red !important;
12 <<<< }
13 <<< </style>
14 <<< <script>
15 <<<< console.log('css加载前', new Date());
16 <<< </script>
17 <<<<!-- Latest compiled and minified CSS -->
18 <<< <link
19 <<<< rel="stylesheet"
20 <<<< href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css"
21 <<<< integrity="sha384-BVYi6Fv01PwKg6PXffTUGA50F850pE58K50tU5PB6NF4QAQw43KAyqh4p/8"
22 <<<< crossorigin="anonymous"
23 <<< />
24 << </head>
25 << <body>
26 <<< <h1>这是红色</h1>
27 <<< <script>
28 <<<< console.log('css加载后', new Date());
29 <<< </script>
30 << </body>
31 </html>
32
```

Q2: CSS加载是否会阻塞页面加载?

结论:

1. CSS加载不会阻塞DOM树的解析 (因此JS中能获取dom元素, 虽然页面还未渲染)
2. 会阻塞Render树的合成 (浏览器渲染到页面时需等待CSS加载完毕)
3. 会阻塞后面的JS执行 (JS执行时可能会操作DOM元素, 如果CSS未加载完毕, DOM元素可能未加载完毕)

Q3: script 元素的 defer 和 async 的区别?

绿色线代表HTML解析过程

灰色线代表解析被阻塞

蓝色线代表网络读取

红色线代表执行时间

1. 网络读取都是异步的，不会阻塞HTML解析。
2. 下载后的执行时机不一样。
3. defer是顺序的。

Legend

- HTML parsing
- HTML parsing paused
- Script download
- Script execution

<script>

Let's start by defining what **<script>** without any attributes does. The HTML file will be parsed until the script file is hit, at that point parsing will stop and a request will be made to fetch the file (if it's external). The script will then be executed before parsing is resumed.



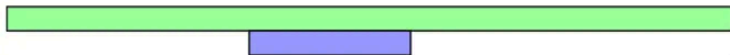
<script async>

async downloads the file during HTML parsing and will pause the HTML parser to execute it when it has finished downloading.



<script defer>

defer downloads the file during HTML parsing and will only execute it after the parser has completed. **defer** scripts are also guaranteed to execute in the order that they appear in the document.



JS的运行原理

The image shows the JavaScript logo, which consists of the letters 'JS' in a bold, black, sans-serif font. The letters are centered within a solid yellow square. The 'J' and 'S' are connected, with the 'S' having a thick, rounded bottom curve.

JS

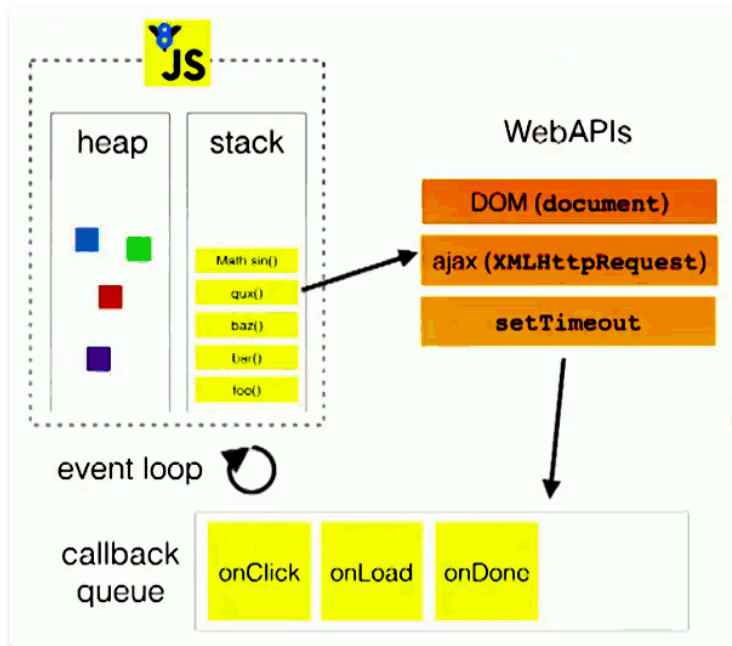
1. JS是单线程的

单线程的优点：

1. 简单、安全
2. 无锁问题
3. 节省内存

2. JS是非阻塞的

实现机制：事件循环机制（Event Loop）



同步任务和异步任务

- (1) 所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。
- (2) 主线程之外，还存在一个"任务队列"（task queue）。只要异步任务有了运行结果，就在"任务队列"之中放置一个事件。
- (3) 一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。
- (4) 主线程不断重复上面的第三步。

JavaScript 运行机制详解：再谈Event Loop —— 阮一峰

同步任务和异步任务

1. ajax 进入EventLoop并注册回调函数 success
2. 执行栈执行 console.log('start')
3. ajax事件完成，回调函数进入Callback Queue
4. 主线程空闲时从Callback Queue读取回调函数
success 执行

想下 setTimeout(fn,0) 的执行过程？主线程执行完同步任务后，会立即执行 setTimeout 的回调函数吗？答案是不会，setTimeout 的回调函数会进入任务队列，等待主线程空闲时才会执行。

```
1 <!--@format-->
2
3 <!DOCTYPE html>
4 <html lang="en">
5   <head>
6     <meta charset="UTF-8" />
7     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
8     <title>Document</title>
9     <style>
10    </style>
11    <!-- Latest compiled and minified CSS -->
12    <script src="https://cdn.bootcdn.net/ajax/libs/jquery/3.6.1/jquery.min.js"></script>
13  </head>
14  <body>
15    <script>
16      let data = []
17      $.ajax({
18        url: 'http://192.168.96.104:9700//swp/device/bedDevice/queryBaseDeviceInfo',
19        data,
20        method: 'get',
21        success: function (res) {
22          console.log('发送成功! ')
23        },
24      })
25      console.log('代码执行结束');
26    </script>
27  </body>
28 </html>
29
```

宏任务与微任务

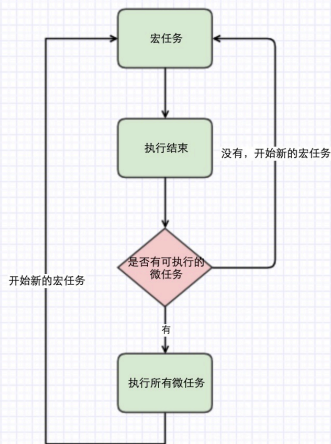
除了同步任务异步任务之分，任务还可以细分为宏任务与微任务：

macro-task（宏任务）：包括整体脚本代码 `script`、`setTimeout`、`setInterval`

micro-task（微任务）：`Promise`、`process.nextTick`、`MutationObserver` 等

宏任务与微任务

进入整体代码（宏任务）后，开始第一次循环。接着执行所有的微任务，当微任务中发起了新的微任务，会继续执行微任务，直到微任务队列为空，浏览器端在微任务执行完成之后会进行一次UI的渲染。接着执行宏任务队列中的下一个任务，然后再次执行所有的微任务，如此循环。



谢谢