浏览器的主要进程和职责



主进程：负责浏览器界面的显示与交互。各个页面的管理,创建和销毁其他进程。网络的资源管理、下载等。

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。

GUI进程：最多只有一个，用于3D绘制等

渲染进程：称为浏览器渲染进程活浏览器内核，内部是多线程的，主要负责页面渲染，脚本执行，事件处理等

渲染的主要线程：GUI渲染线程、js引擎线程，事件触发线程，定时器触发线程，异步http请求线程

GUI渲染线程：

负责渲染浏览器界面，解析html，css，构建DOM树和RenderObject树，布局和绘制等

当界面需要重绘（Repaint）或由于某种操作引发回流(reflow)时,该线程就会执行。

注意,GUI 渲染线程与 JS 引擎线程是互斥的,当 JS 引擎执行时 GUI 线程会被挂起（相当于被冻结了）,GUI 更新会被保存在一个队列中等到 JS 引擎空闲时立即被执行。

JS引擎线程

Javascript 引擎,也称为 JS 内核,负责处理 Javascript 脚本程序。（例如 V8 引擎）

JS 引擎线程负责解析 Javascript 脚本,运行代码。

JS 引擎一直等待着任务队列中任务的到来,然后加以处理,一个 Tab 页（renderer 进程）中无论什么时候都只有一个 JS 线程在运行 JS 程序。

注意,GUI 渲染线程与 JS 引擎线程是互斥的,所以如果 JS 执行的时间过长,这样就会造成页面的渲染不连贯,导致页面渲染加载阻塞。

事件触发线程

归属于浏览器而不是 JS 引擎,用来控制事件循环（可以理解,JS 引擎自己都忙不过来,需要浏览器另开线程协助）

当 JS 引擎执行代码块如 setTimeOut 时（也可来自浏览器内核的其他线程,如鼠标点击、AJAX 异步请求等）,会将对应任务添加到事件线程中

当对应的事件符合触发条件被触发时,该线程会把事件添加到待处理队列的队尾,等待 JS 引擎的处理

·注意,由于 JS 的单线程关系,所以这些待处理队列中的事件都得排队等待 JS 引擎处理（当 JS 引擎空闲时才会去执行）

定时器触发线程

传说中的 setInterval 与 setTimeout 所在线程

浏览器定时计数器并不是由 JavaScript 引擎计数的,（因为 JavaScript 引擎是单线程的, 如果处于阻塞线程状态就会影响记计时的准确）

因此通过单独线程来计时并触发定时（计时完毕后,添加到事件队列中,等待 JS 引擎空闲后执行）

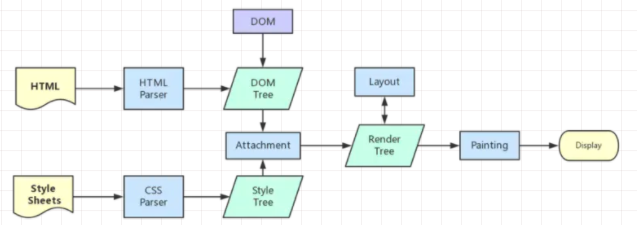
注意,W3C 在 HTML 标准中规定,规定要求 setTimeout 中低于 4ms 的时间间隔算为 4ms。

异步http请求线程

在 XMLHttpRequest 在连接后是通过浏览器新开一个线程请求。

将检测到状态变更时,如果设置有回调函数,异步线程就产生状态变更事件,将这个回调再放入事件队列中。再由 JavaScript 引擎执行。

浏览器渲染流程



1.解析 HTML 文件,构建 DOM 树,同时浏览器主进程负责下载 CSS 文件

2.CSS 文件下载完成,解析 CSS 文件成树形的数据结构,然后结合 DOM 树合并成 RenderObject 树

3.布局 RenderObject 树 （Layout/reflow）,负责 RenderObject 树中的元素的尺寸,位置等计算

4.绘制 RenderObject 树 （paint）,绘制页面的像素信息

5.浏览器主进程将默认的图层和复合图层交给 GPU 进程,GPU 进程再将各个图层合成（composite）,最后显示出页面

1、为什么JavaScript是单线程的？

这是因为 Javascript 这门脚本语言诞生的使命所致!JavaScript 为处理页面中用户的交互,以及操作 DOM 树、CSS 样式树来给用户呈现一份动态而丰富的交互体验和服务器逻辑的交互处理。

如果 JavaScript 是多线程的方式来操作这些 UI DOM,则可能出现 UI 操作的冲突。

如果 Javascript 是多线程的话,在多线程的交互下,处于 UI 中的 DOM 节点就可能成为一个临界资源,

假设存在两个线程同时操作一个 DOM,一个负责修改一个负责删除,那么这个时候就需要浏览器来裁决如何生效哪个线程的执行结果。

当然我们可以通过锁来解决上面的问题。但为了避免因为引入了锁而带来更大的复杂性,Javascript 在最初就选择了单线程执行。

2、为什么 JS 阻塞页面加载 ?

由于 JavaScript 是可操纵 DOM 的,如果在修改这些元素属性同时渲染界面（即 JavaScript 线程和 UI 线程同时运行）,那么渲染线程前后获得的元素数据就可能不一致了。

因此为了防止渲染出现不可预期的结果,浏览器设置 GUI 渲染线程与 JavaScript 引擎为互斥的关系。

当 JavaScript 引擎执行时 GUI 线程会被挂起,GUI 更新会被保存在一个队列中等到引擎线程空闲时立即被执行。

从上面我们可以推理出,由于 GUI 渲染线程与 JavaScript 执行线程是互斥的关系,

当浏览器在执行 JavaScript 程序的时候,GUI 渲染线程会被保存在一个队列中,直到 JS 程序执行完成,才会接着执行。

因此如果 JS 执行的时间过长,这样就会造成页面的渲染不连贯,导致页面渲染加载阻塞的感觉

3、css加载会造成阻塞吗？

由上面浏览器渲染流程我们可以看出：

DOM解析和CSS解析是两个并行的进程，所以css加载不会阻塞DOM的解析

然后，有与RenderTree是以来以DOMTree和cssTree的，所以他必须等到cssDOmtree构建完成，也就是css资源加载完成活css资源加载失败后才开始渲染，因此，css加载会阻塞DOM的渲染

由于JavaScript是可操纵DOM和css样式的，如果在修改这些元素属性同时渲染界面（即JavaScript线程和UI线程同事运行），那么渲染线程前后获得的元素数据就可能不一致了，因此为了防止渲染出现不可预期的结果，浏览器设置 GUI 渲染线程与 JavaScript 引擎为互斥的关系。

因此,样式表会在后面的 js 执行前先加载执行完毕,所以css 会阻塞后面 js 的执行。

4、DOMContentLoaded 与 load 的区别 ?

当 DOMContentLoaded 事件触发时,仅当 DOM 解析完成后,不包括样式表,图片。我们前面提到 CSS 加载会阻塞 Dom 的渲染和后面 js 的执行,js 会阻塞 Dom 解析,所以我们可以得到结论:  
当文档中没有脚本时,浏览器解析完文档便能触发 DOMContentLoaded 事件。如果文档中包含脚本,则脚本会阻塞文档的解析,而脚本需要等 CSSOM 构建完成才能执行。在任何情况下,DOMContentLoaded 的触发不需要等待图片等其他资源加载完成。

当 onload 事件触发时,页面上所有的 DOM,样式表,脚本,图片等资源已经加载完毕。

DOMContentLoaded -> load。

5. 什么是 CRP,即关键渲染路径(Critical Rendering Path)? 如何优化 ?

关键渲染路径是浏览器将 HTML CSS JavaScript 转换为在屏幕上呈现的像素内容所经历的一系列步骤。也就是我们上面说的浏览器渲染流程。

为尽快完成首次渲染,我们需要最大限度减小以下三种可变因素:

关键资源的数量: 可能阻止网页首次渲染的资源。

关键路径长度: 获取所有关键资源所需的往返次数或总时间。

关键字节: 实现网页首次渲染所需的总字节数,等同于所有关键资源传送文件大小的总和。

1. 优化 DOM

删除不必要的代码和注释包括空格,尽量做到最小化文件。

可以利用 GZIP 压缩文件。

结合 HTTP 缓存文件。

2. 优化 CSSOM

缩小、压缩以及缓存同样重要,对于 CSSOM 我们前面重点提过了它会阻止页面呈现,因此我们可以从这方面考虑去优化。

减少关键 CSS 元素数量

当我们声明样式表时,请密切关注媒体查询的类型,它们极大地影响了 CRP 的性能 。

3. 优化 JavaScript

当浏览器遇到 script 标记时,会阻止解析器继续操作,直到 CSSOM 构建完毕,JavaScript 才会运行并继续完成 DOM 构建过程。

async: 当我们在 script 标记添加 async 属性以后,浏览器遇到这个 script 标记时会继续解析 DOM,同时脚本也不会被 CSSOM 阻止,即不会阻止 CRP。

defer: 与 async 的区别在于,脚本需要等到文档解析后（ DOMContentLoaded 事件前）执行,而 async 允许脚本在文档解析时位于后台运行（两者下载的过程不会阻塞 DOM,但执行会）。

当我们的脚本不会修改 DOM 或 CSSOM 时,推荐使用 async 。

预加载 —— preload & prefetch 。

DNS 预解析 —— dns-prefetch 。

总结

分析并用 关键资源数 关键字节数 关键路径长度 来描述我们的 CRP 。

最小化关键资源数: 消除它们（内联）、推迟它们的下载（defer）或者使它们异步解析（async）等等 。

优化关键字节数（缩小、压缩）来减少下载时间 。

优化加载剩余关键资源的顺序: 让关键资源（CSS）尽早下载以减少 CRP 长度 。

[前端性能优化之关键路径渲染优化](https://github.com/fi3ework/blog/issues/16" \t "/Users/JRJ/Documents\\x/_blank)

6. defer 和 async 的区别 ?

当浏览器碰到 script 脚本的时候 :

1. <script src="script.js">

没有 defer 或 async,浏览器会立即加载并执行指定的脚本,“立即”指的是在渲染该 script 标签之下的文档元素之前,也就是说不等待后续载入的文档元素,读到就加载并执行。

2. <script async src="script.js">

有 async,加载和渲染后续文档元素的过程将和 script.js 的加载与执行并行进行（异步）。

3. <script defer src="myscript.js">

有 defer,加载后续文档元素的过程将和 script.js 的加载并行进行（异步）,但是 script.js 的执行要在所有元素解析完成之后,DOMContentLoaded 事件触发之前完成。

从实用角度来说,首先把所有脚本都丢到 </body> 之前是最佳实践,因为对于旧浏览器来说这是唯一的优化选择,此法可保证非脚本的其他一切元素能够以最快的速度得到加载和解析。

defer 和 async 在网络读取（下载）这块儿是一样的,都是异步的（相较于 HTML 解析）

它俩的差别在于脚本下载完之后何时执行,显然 defer 是最接近我们对于应用脚本加载和执行的要求的

关于 defer,此图未尽之处在于它是按照加载顺序执行脚本的,这一点要善加利用

async 则是一个乱序执行的主,反正对它来说脚本的加载和执行是紧紧挨着的,所以不管你声明的顺序如何,只要它加载完了就会立刻执行

仔细想想,async 对于应用脚本的用处不大,因为它完全不考虑依赖（哪怕是最低级的顺序执行）,不过它对于那些可以不依赖任何脚本或不被任何脚本依赖的脚本来说却是非常合适的

7. 谈谈浏览器的回流与重绘

回流必将引起重绘,重绘不一定会引起回流。

回流(Reflow)

当 Render Tree 中部分或全部元素的尺寸、结构、或某些属性发生改变时,浏览器重新渲染部分或全部文档的过程称为回流。

会导致回流的操作：

页面首次渲染

浏览器窗口大小发生改变

元素尺寸或位置发生改变元素内容变化（文字数量或图片大小等等）

元素字体大小变化

添加或者删除可见的 DOM 元素

激活 CSS 伪类（例如：:hover）

查询某些属性或调用某些方法

一些常用且会导致回流的属性和方法:

clientWidth、clientHeight、clientTop、clientLeft、offsetWidth、offsetHeight、offsetTop、offsetLeft、scrollWidth、scrollHeight、scrollTop、scrollLeft、scrollIntoView()、scrollIntoViewIfNeeded()、getComputedStyle()、getBoundingClientRect()、scrollTo()

重绘(Repaint)

当页面中元素样式的改变并不影响它在文档流中的位置时（例如：color、background-color、visibility 等）,浏览器会将新样式赋予给元素并重新绘制它,这个过程称为重绘。

性能影响

回流比重绘的代价要更高。有时即使仅仅回流一个单一的元素,它的父元素以及任何跟随它的元素也会产生回流。现代浏览器会对频繁的回流或重绘操作进行优化：浏览器会维护一个队列,把所有引起回流和重绘的操作放入队列中,如果队列中的任务数量或者时间间隔达到一个阈值的,浏览器就会将队列清空,进行一次批处理,这样可以把多次回流和重绘变成一次。当你访问以下属性或方法时,浏览器会立刻清空队列:

clientWidth、clientHeight、clientTop、clientLeft、offsetWidth、offsetHeight、offsetTop、offsetLeft、scrollWidth、scrollHeight、scrollTop、scrollLeft、width、height、getComputedStyle()、getBoundingClientRect()

因为队列中可能会有影响到这些属性或方法返回值的操作,即使你希望获取的信息与队列中操作引发的改变无关,浏览器也会强行清空队列,确保你拿到的值是最精确的。

如何避免

CSS

避免使用 table 布局。

尽可能在 DOM 树的最末端改变 class。

避免设置多层内联样式。

将动画效果应用到 position 属性为 absolute 或 fixed 的元素上。

避免使用 CSS 表达式（例如：calc()）。

Javascript

避免频繁操作样式,最好一次性重写 style 属性,或者将样式列表定义为 class 并一次性更改 class 属性。

避免频繁操作 DOM,创建一个 documentFragment,在它上面应用所有 DOM 操作,最后再把它添加到文档中。

也可以先为元素设置 display: none,操作结束后再把它显示出来。因为在 display 属性为 none 的元素上进行的 DOM 操作不会引发回流和重绘。

避免频繁读取会引发回流/重绘的属性,如果确实需要多次使用,就用一个变量缓存起来。

对具有复杂动画的元素使用绝对定位,使它脱离文档流,否则会引起父元素及后续元素频繁回流。

8. 什么是渲染层合并 (Composite) ?

渲染层合并,对于页面中 DOM 元素的绘制(Paint)是在多个层上进行的。

在每个层上完成绘制过程之后,浏览器会将绘制的位图发送给 GPU 绘制到屏幕上,将所有层按照合理的顺序合并成一个图层,然后在屏幕上呈现。

对于有位置重叠的元素的页面,这个过程尤其重要,因为一旦图层的合并顺序出错,将会导致元素显示异常。

RenderLayers 渲染层,这是负责对应 DOM 子树。

GraphicsLayers 图形层,这是负责对应 RenderLayers 子树。

RenderObjects 保持了树结构,一个 RenderObjects 知道如何绘制一个 node 的内容, 他通过向一个绘图上下文（GraphicsContext）发出必要的绘制调用来绘制 nodes。

每个 GraphicsLayer 都有一个 GraphicsContext,GraphicsContext 会负责输出该层的位图,位图是存储在共享内存中,作为纹理上传到 GPU 中,最后由 GPU 将多个位图进行合成,然后 draw 到屏幕上,此时,我们的页面也就展现到了屏幕上。

GraphicsContext 绘图上下文的责任就是向屏幕进行像素绘制(这个过程是先把像素级的数据写入位图中,然后再显示到显示器),在 chrome 里,绘图上下文是包裹了的 Skia（chrome 自己的 2d 图形绘制库）

某些特殊的渲染层会被认为是合成层（Compositing Layers）,合成层拥有单独的 GraphicsLayer,而其他不是合成层的渲染层,则和其第一个拥有 GraphicsLayer 父层公用一个。

合成层的优点

一旦 renderLayer 提升为了合成层就会有自己的绘图上下文,并且会开启硬件加速,有利于性能提升。

合成层的位图,会交由 GPU 合成,比 CPU 处理要快 (提升到合成层后合成层的位图会交 GPU 处理,但请注意,仅仅只是合成的处理（把绘图上下文的位图输出进行组合）需要用到 GPU,生成合成层的位图处理（绘图上下文的工作）是需要 CPU。)

当需要 repaint 时,只需要 repaint 本身,不会影响到其他的层 (当需要 repaint 的时候可以只 repaint 本身,不影响其他层,但是 paint 之前还有 style, layout,那就意味着即使合成层只是 repaint 了自己,但 style 和 layout 本身就很占用时间。)

对于 transform 和 opacity 效果,不会触发 layout 和 paint (仅仅是 transform 和 opacity 不会引发 layout 和 paint,其他的属性不确定。)

一般一个元素开启硬件加速后会变成合成层,可以独立于普通文档流中,改动后可以避免整个页面重绘,提升性能。

注意不能滥用 GPU 加速,一定要分析其实际性能表现。因为 GPU 加速创建渲染层是有代价的,每创建一个新的渲染层,就意味着新的内存分配和更复杂的层的管理。并且在移动端 GPU 和 CPU 的带宽有限制,创建的渲染层过多时,合成也会消耗跟多的时间,随之而来的就是耗电更多,内存占用更多。过多的渲染层来带的开销而对页面渲染性能产生的影响,甚至远远超过了它在性能改善上带来的好处。

这里就不细说了,有兴趣的童鞋推荐以下三篇文章 ~

[Accelerated Rendering in Chrome](https://www.html5rocks.com/zh/tutorials/speed/layers/" \t "/Users/JRJ/Documents\\x/_blank)

[CSS GPU Animation: Doing It Right](https://www.smashingmagazine.com/2016/12/gpu-animation-doing-it-right/" \t "/Users/JRJ/Documents\\x/_blank)

[无线性能优化：Composite](https://fed.taobao.org/blog/taofed/do71ct/performance-composite/?spm=taofed.blogs.blog-list.10.67bd5ac8fHy0LS" \t "/Users/JRJ/Documents\\x/_blank)

参考

[史上最全！图解浏览器的工作原理](https://www.infoq.cn/article/CS9-WZQlNR5h05HHDo1b" \t "/Users/JRJ/Documents\\x/_blank)

[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](https://juejin.im/post/5a6547d0f265da3e283a1df7" \t "/Users/JRJ/Documents\\x/_blank)