浏览器渲染机制

**浏览器中输入网址后到页面展现的过程**

1）用户输入网址

2）浏览器通过DNS获取网站的IP地址。

浏览器自身的缓存

本地的host缓存

向本地的dns服务器发起查询

向根dns服务器发起查询

向顶级域名dns服务器发起查询

向权威dns服务器发起查询

3）浏览器客户端发送http请求

4）传输层TCP传输报文。TCP协议通过“三次握手”等方法保证传输的安全可靠。(防止失效的连接请求报文段被服务端接收，从而产生错误。)

5）网络层IP协议查询MAC地址

　　IP协议的作用是把TCP分割好的各种数据包传送给接收方。而要保证确实能传到接收方还需要接收方的MAC地址，也就是物理地址。IP地址和MAC地址是一一对应的关系，一个网络设备的IP地址可以更换，但是MAC地址一般是固定不变的。ARP协议可以将IP地址解析成对应的MAC地址。当通信的双方不在同一个局域网时，需要多次中转才能到达最终的目标，在中转的过程中需要通过下一个中转站的MAC地址来搜索下一个中转目标。

6）数据到达数据链路层

　　在找到对方的MAC地址后，就将数据发送到数据链路层传输。这时，客户端发送请求的阶段结束

7）服务器接收数据

8）服务器响应请求

9）服务器返回响应文件

10）页面渲染： 解析HTML以构建DOM树 –> 构建渲染树 –> 布局渲染树 –> 绘制渲染树。

1. 关闭TCP连接（四次挥手）

**浏览器的渲染过程(最终版):**html文件解析过程

1. Browser进程下载html文件，随后将该任务通过**Browser的RendererHost接口**传递给Render进程

2. **Renderer进程的Renderer接口**收到消息后，交给自己的GUI线程开始解析html文件来构建出DOM 树

3. 当遇到外源css时，Browser进程下载该css文件并发送回来，GUI线程再解析该文件，在这同时，html的解析也同时进行，但不会渲染（还未形成渲染树）

4. 当遇到内部css时，html的解析和css的解析同时进行

5. 继续解析html文件，当遇到外源js时，Browser进程下载该js文件并发送回来，此时，js引擎线程解析并执行js，因为GUI线程和js引擎线程互斥，所以GUI线程被挂起，停止继续解析html。直到js引擎线程空闲，GUI线程继续解析html。

6. 遇到内部js也是同理

7. 解析完html文件，形成了完整的DOM树，也解析完了css，形成了完整的CSSOM树(样式层叠模型)，两者结合形成了render树

8. 根据render树来进行布局，若在布局的过程中发生了元素尺寸、位置、隐藏的变化或增加、删除元素时，则进行回流，修改

9. 根据render树进行绘制，若在布局的过程中元素的外观发生变换，则进行重绘

10. 将布局、绘制得到的各个简单图层的位图发送给Browser进程，由它来合并简单图层为复合图层，从而显示到页面上

11. 以上步骤就是html文件解析全过程，完成之后，如若当页面有元素的尺寸、大小、隐藏有变化时，重新布局计算回流，并修改页面中所有受影响的部分，如若当页面有元素的外观发生变化时，重绘

**浏览器是多进程的,具体包括:**

**GPU进程**

负责3D绘制，只有当该页面使用了硬件加速才会使用它来渲染页面。否则的话，不使用这个进程，而是用Browser进程来渲染（显示）页面

**Browser进程**

负责浏览器界面显示，与用户交互。如前进，后退等

负责各个页面的管理，创建和销毁其他进程

将浏览器渲染进程得到的内存中的位图，绘制到用户界面上

网络资源的管理，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

**浏览器渲染进程（浏览器内核）**

每个tab页面对应一个独立的renderer进程，内部有多个线程

负责脚本执行，位图绘制，事件触发，任务队列轮询等

**浏览器渲染进程是多线程的**

**GUI渲染线程**

用于解析html为DOM树，解析css为CSSOM树，布局layout，绘制paint

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: ( js引擎线程执行完才会执行GUI渲染线程 )

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，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**

也称JS内核,负责解析Javascript脚本，运行代码。

由一个主线程和多个web worker线程组成，由于web worker是附属于主线程，无法操作dom等，所以js还是单线程语言（在主线程运行js代码）

==>js引擎线程是多线程的,可以去了解web worker线程

JS引擎一直等待任务队列中任务的到来，然后加以处理，一个Tab页（进程）中只有一个JS线程在运行

同样注意，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，要放在body下面，否则就会导致页面渲染加载阻塞。

事件触发线程

当对应事件触发（不论是WebAPIs完成事件触发，还是页面交互事件触发）时，该线程会将事件对应的回调函数放入callback queue（任务队列）中，等待js引擎线程的处理

**定时触发器线程**

setInterval与setTimeout在此线程中计时完毕后，把回调函数放入事件队列中

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，当setTimeout的定时的时间小于4ms，一律按4ms来算

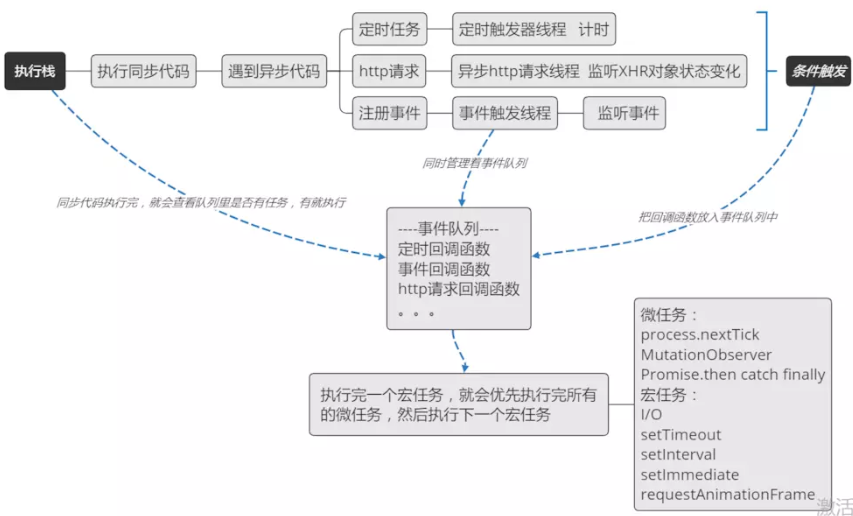
**任务队列轮询线程**

用于轮询监听任务队列，以知道任务队列是否为空

**异步http请求线程**

每有一个http请求就开一个该线程

当检测到状态变更的话，就会产生一个状态变更事件，如果该状态变更事件对应有回调函数的话，则放入任务队列中



**css的堵塞情况**

首先，是在Browser进程中下载css文件，当下载完成后，发送给GUI线程。

其次，是在GUI线程中解析html及css，不过这两者是并行的。

由于css的下载和解析不会影响DOM树，所以不会堵塞html文件的解析，但会堵塞页面渲染。这样的设计是非常合理的，如果css文件的下载和解析不会堵塞页面渲染，那么在页面渲染的途中或结束后发现元素样式有变化，则又需要回流和重绘。

**js的堵塞情况**

明确的是，js文件的下载和解析执行都会堵塞html文件的解析及页面渲染。

(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，要放在body下面，否则就会导致页面渲染加载阻塞。)

因为js脚本可能会改变DOM结构，若是其不堵塞html文件的解析及页面渲染的话，那么当js脚本改变DOM结构或元素样式时，会引发回流和重绘，会造成不必要的性能浪费，不如等待js执行完，在进行html解析和页面渲染。

如果你不想js堵塞的话，则使用async属性，这样就可以异步加载js文件，加载完成后立即执行。

**css和js文件应当放在html哪个位置**

css：一般放在head中，因为css的解析不影响html的解析，所以越早引入越早同时解析。

js：

①当需要在DOM树完成之前用js进行初始化操作的话，在head中使用js。

②如果是需要在DOM树形成之后，即要操作DOM，则在body元素的末尾。不过也可以使用load事件。

③如果js的内容比较小，则推荐使用内部js而不是引用js，这样可以减少http请求。

**repaint(重绘)**

repiant或者redraw遍历所有的节点检测各节点的可见性、颜色、轮廓等可见的样式属性，然后根据检测的结果更新页面的响应部分。

当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为重绘

**reflow(回流)**

reflow指的是计算页面布局。某个节点reflow时会重新计算节点的尺寸和位置，而且还有可能触发其子节点、祖先节点和页面上的其他节点reflow。在这之后再触发一次repaint。

当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为回流,每个页面至少需要一次回流，就是在页面第一次加载的时候。

**导致reflow的操作:**

页面初始化渲染

窗口的尺寸变化

元素的尺寸、位置变化

**隐藏的话分情况：display:none会引起回流和重绘，visibility:hidden会引起重绘**

DOM结构发生变化，如删除节点

字体大小修改及内容更新也会导致回流

获取某些属性，引发回流:

①很多浏览器会对回流进行优化，一定时间段后或数量达到阕值时，做一次批处理回流。

②当获取一些属性时，浏览器为了返回正确的值也会触发回流，导致浏览器优化无效,如：

offset（top/bottom/left/right）

client (top/bottom/left/right)

scroll (top/bottom/left/right)

getComputedStyle()

width,height

**它们会大大影响web性能，如何减少reflow(回流)、repaint(重绘)**

①不要通过父级来改变子元素样式，最好直接改变子元素样式，改变子元素样式尽可能不要影响父元素和兄弟元素的大小和尺寸

②避免不必要的复杂的 CSS 选择器，尤其是后代选择器，因为为了匹配选择器将耗费更多的 CPU。

③将复杂的元素绝对定位或固定定位，使他脱离文档流，否则回流代价很高

④避免循环操作DOM，而是新建一个节点，在他上面应用所有DOM操作，然后再将他接入到DOM中

⑤尽量不要过多的频繁的去增加，修改，删除元素，因为这可能会频繁的导致页面reflow，可以先把该dom节点抽离到内存中进行复杂的操作然后再display到页面上。

⑥频繁读取如下值offsetTop, offsetLeft, offsetWidth, offsetHeight，scrollTop/Left/Width/Height，clientTop/Left/Width/Height，浏览器会发生reflow，建议只读取一次然后赋值给变量并且将他们合并到一起操作，可以减少回流的次数。

**宏任务和微任务**

宏任务（macrotask）：

- 主代码块和setTimeout，setInterval(任务队列中的回调函数就是宏任务)。

- 为了使js内部宏任务和DOM任务能够有序的执行，每次执行完宏任务后，会在下一个宏任务执行之前，对页面重新进行渲染。（宏任务 -> 渲染 -> 宏任务）

微任务（microtask）：

- 微任务: Promise

- 在宏任务执行过程中，执行到微任务时，将微任务放入微任务队列中。

- 在宏任务执行完后，在重新渲染之前执行。

- 当一个宏任务执行完后，他会将产生的所有微任务执行完。

**导致页面无法立即响应的原因**

执行栈中还有任务未执行完

或者是js引擎线程被GUI线程堵塞。

**事件循环机制**

事件循环机制的核心就是观察者模式。

而跟观察者模式相对应的是轮询=>

映射在现实世界中即为：B不停到A的房间观察房间里是否有人，从而知道A是否回来。

发展=>B在自己房间做自己的事情，不再不停地到A的房间看他是否回来，而是当A回到自己房间时，打电话通知B他回来了，B再去房间找A玩。

**事件循环机制执行的流程:**

1. js程序进入线程，函数入栈，当遇到同步代码的时候就顺序执行，遇到异步代码时，把异步任务抛给WebAPIs执行，然后继续执行接下来的同步代码，直到栈为空。
2. 在步骤1进行的同时，WebAPIs执行异步任务(处理诸如DOM事件，http请求，定时器等异步任务)，当执行完一个异步任务就将其对应的回调函数放入任务队列（Callback Queue）中等待。
3. 当执行栈为空时，从Callback Queue中取出队列头放入执行栈中，回到第一步。

**js事件循环机制优缺点及与多线程的比较**

通过事件循环机制，我们就可以实现代码的异步，从而不会堵塞线程。

====>

js在IO上有着卓越的表现，因为IO操作不再会堵塞住线程。

可以做到高并发。稍微解释一下为什么能够高并发——当同时有多个任务要执行，js将他一一排列起来，然后按顺序执行，这样cpu就不会因为同时要处理的工作太多而负载过大。

**js事件循环机制的坏处:**

①不适合cpu密集型。也解释一下——如一段代码需要非常大量的计算量，以至于他长时间地占着线程，这就堵塞了，后继的同步代码及异步代码都无法执行。不过，html5推出了web worker，可以有效地解决这一缺陷，在本章不表，后面我会专门写一篇文章来讲他。

②只能使用一个线程，无法充分利用计算机的多核cpu。

③可靠性低，一旦一个环节崩溃则整个程序全部崩溃。

**总结一句话：**

 事件循环机制的核心是事件触发线程，由于执行栈产生异步任务，异步任务完成后事件触发线程将其回调函数传入到任务队列中，当执行栈为空，任务队列将队列头的回调函数入执行栈，从而新的一轮循环开始。这就是称为循环的原因。