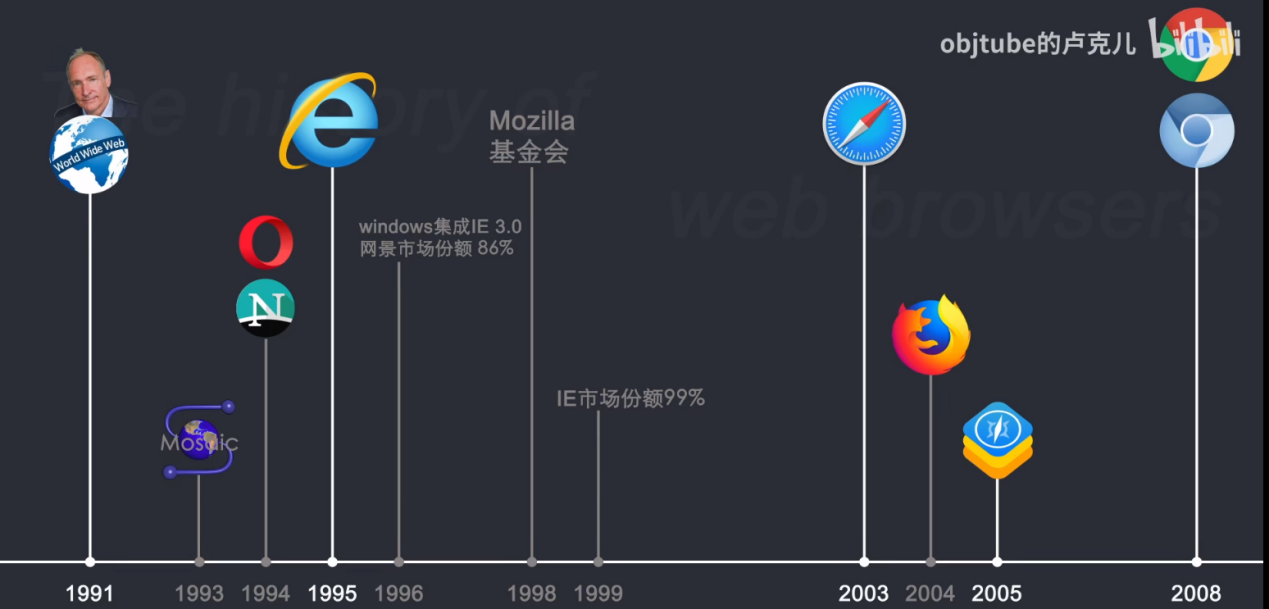
1. 浏览器发展



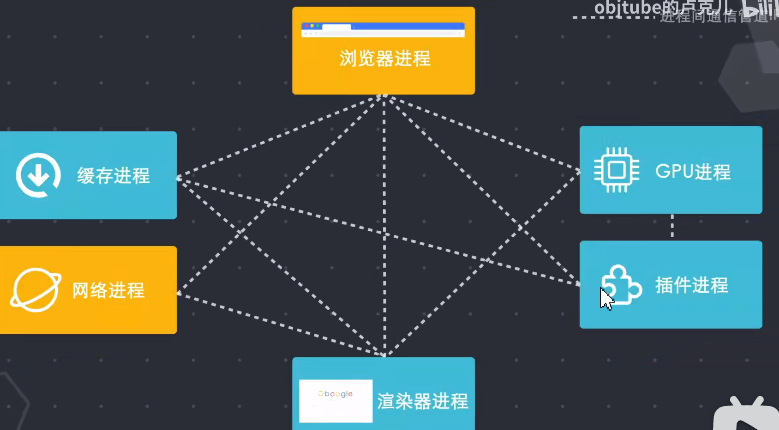


1. 浏览器结构
   1. 大致分



渲染引擎 -----> （通常所说的）内核

* 1. 按功能分



其中：

网络进程负责发起接收网络请求

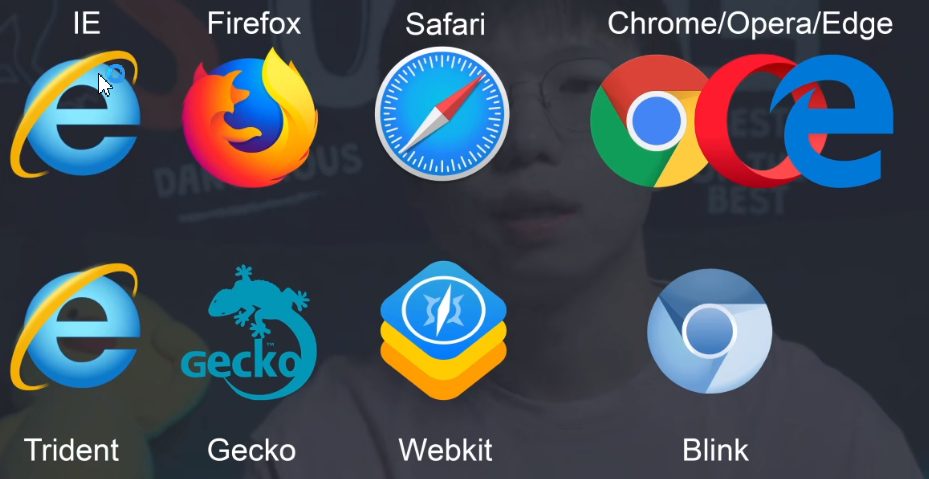
GPU进程负责整个浏览器界面的渲染

插件进程负责控制网站使用等所有插件，例如Flash

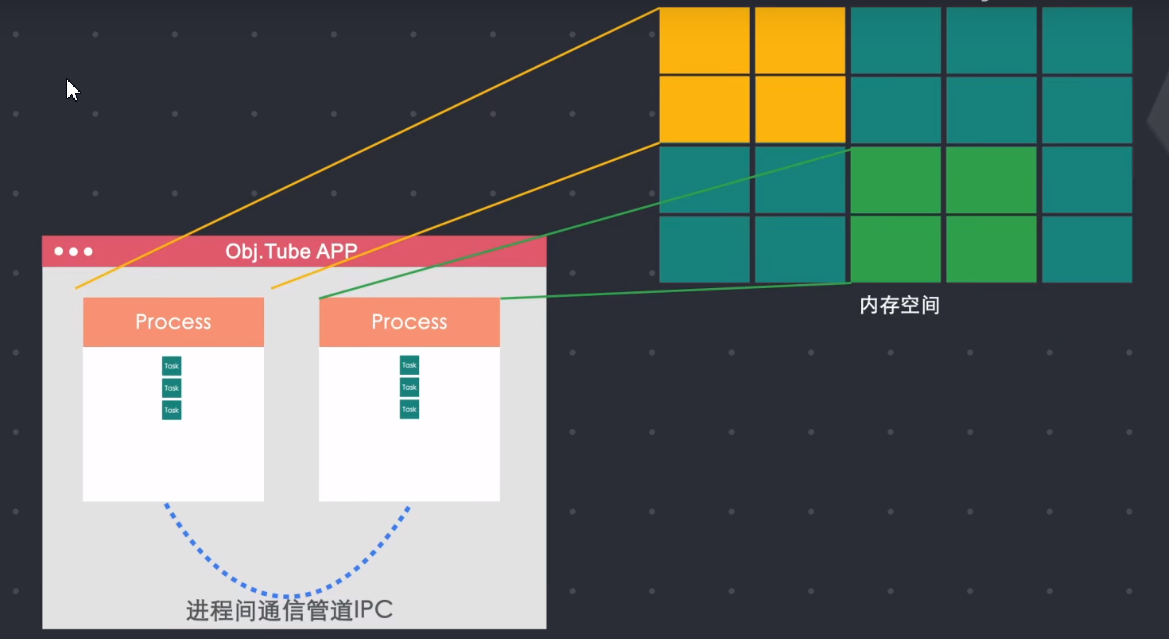
渲染器进程用来控制显示tab标签内的所有内容，浏览器在默认情况（这和你启动Chrome时选择的进程模型有关）下会为每个标签页都创建一个进程



1. 各个浏览器的内核



1. 线程进程IPC



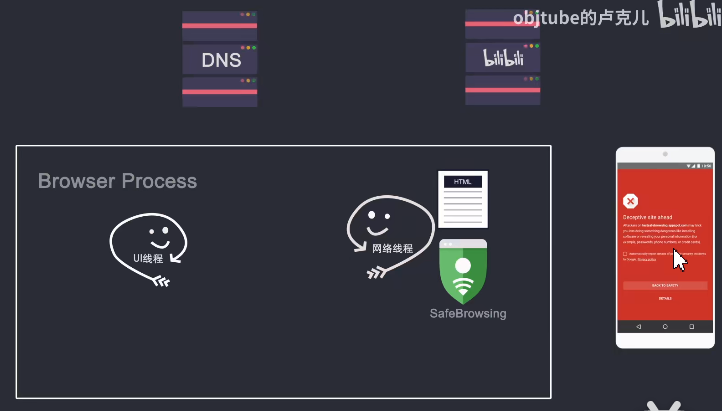
1. 搜索

浏览器进程的UI线程会捕捉你的输入内容。

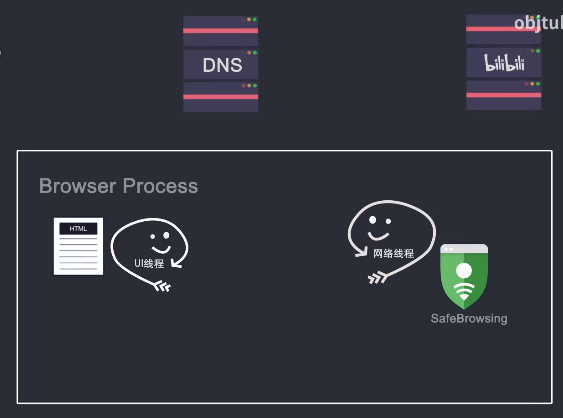
如果访问的是网址，则UI线程会启动一个网络线程来请求DNS进行域名分析，接着开始连接服务器获取数据。

如果你的输入不是网址，而是一串关键词，浏览器就知道你是要搜索，于是就会使用默认配置的搜索殷勤来查询

1. 网络线程获取到数据之后会发生什么样的事情（Chrome为例）
   1. 当网络线程获取到数据后会通过SafeBrowsing（谷歌内部的一套站点安全系统，通过检测该站点的数据来判断是否安全，比如通过查看该站点的IP是否在谷歌的黑名单内）来检查站点是否是恶意站点如果是，则会提示个警告页面，告诉你这个站点有安全问题，浏览器会阻止你的访问，当然你也可以强行继续访问。



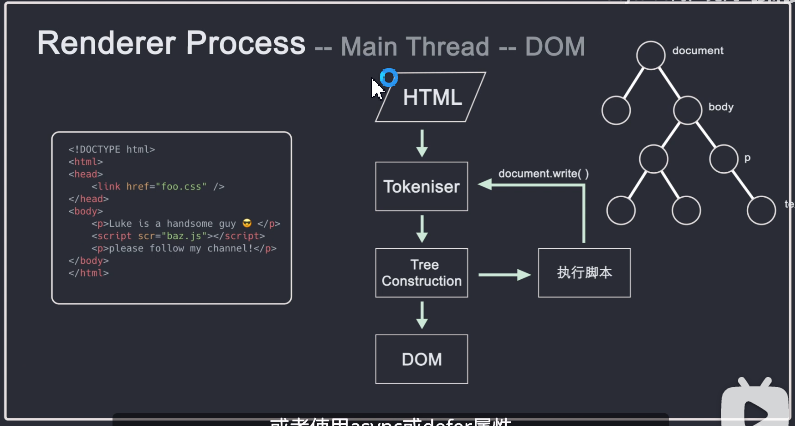
* 1. 当返回数据准备完毕，并且安全校验通过时，网络线程会通知UI线程我就要准备好了，该你了，然后UI线程会创建一个渲染器进程（Renderer Thread）来渲染页面,浏览器通过IPC管道将数据传递给渲染器进程，正式进入渲染流程



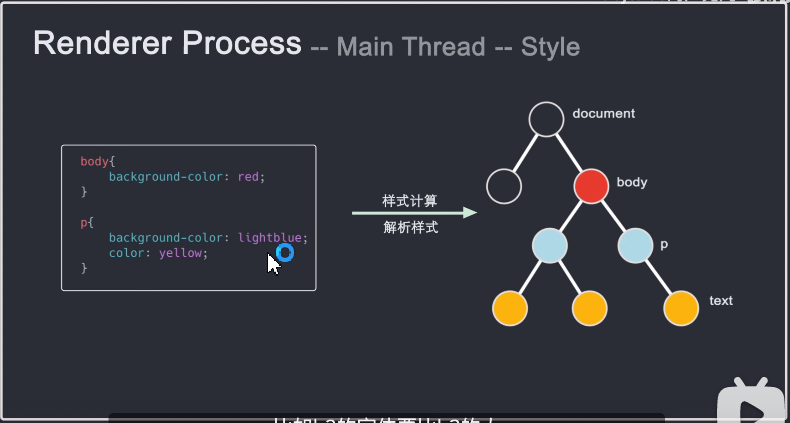
（<------ 渲染器进程）

* 1. 渲染器接收到的数据也就是html，渲染器进程的核心任务就是把html、css、js、image等资源渲染成用户可以交互的web页面，渲染器进程的主线程将html进行解析，构造DOM数据结构（DOM也就是文档对象模型，是浏览器对页面在其内部的表现形式，是web开发程序员可以通过JS与之交互的数据结构和API（接口））

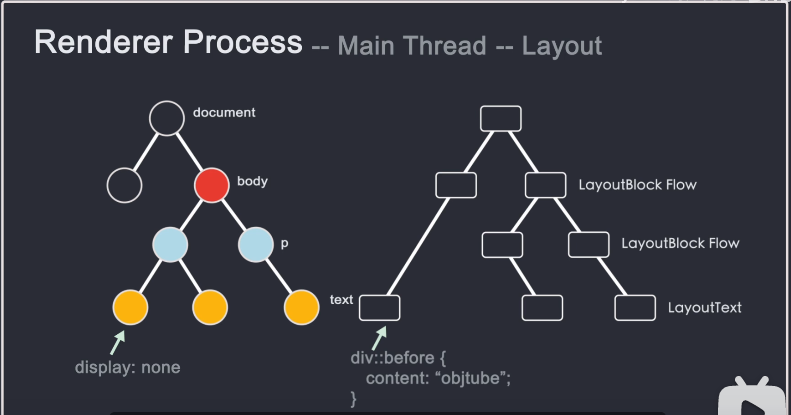
具体实现：html首先经过tokeniser标记化，通过词法分析将输入的html内容解析成多个标记，根据识别后的标记进行DOM树构造，在DOM树构造过程中会创建document对象，然后以document为根节点的DOM树不断进行修改，向其中添加各种元素，html代码中往往会引入一些额外的资源，比如说图片、CSS、JS脚本等，图片和CSS这些资源需要通过网络下载或从缓存中直接加载，这些资源不会阻塞html的解析，因为他们不会影响DOM的生成，但当HTML标解析过程中遇到script标签，就会停止html解析流程，转而去加载解析并且执行JS（因为浏览器并不知道JS执行是否会改变当前页面的HTML结构，如果JS代码里用了document.write方法来修改html，之前的HTML解析就没有任何意义了，这就是为什么不直接跳过JS的加载和执行这一过程，等html解析完之后再加载运行JS，这也就是为什么我们一直说要把script标签放在合适的位置，或者使用async或defer属性来异步加载执行JS）



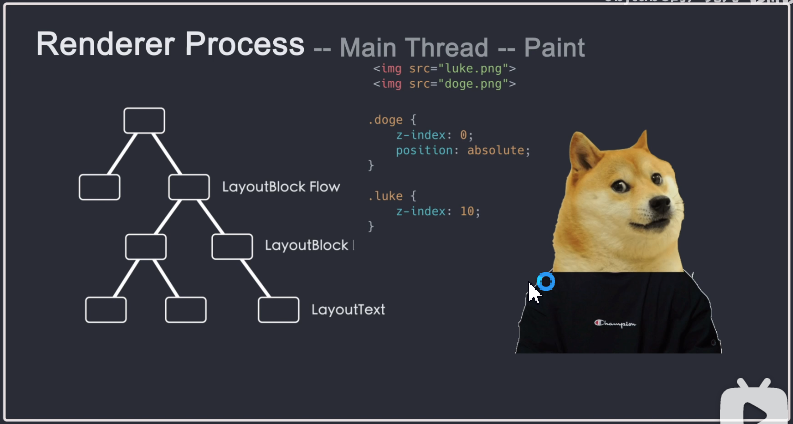
* 1. 在html解析完成后，我们就会获得一个DOM Tree（树），但我们还不知道DOM树上每个节点应该长什么样子，主线程需要解析CSS并确定每个DOM结点的计算样式（即使你没有提供自定义的CSS的样式，浏览器也会有自己默认的样式表，比如h2的字体要比h3的大）

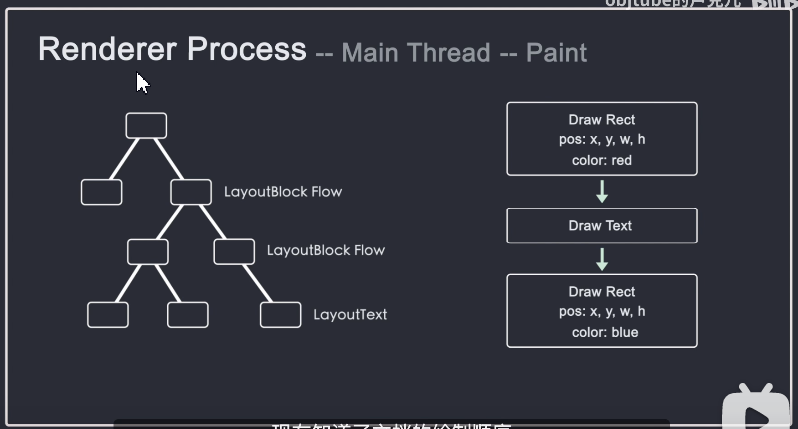


* 1. 在知道DOM结构和每个节点的样式后，我们接下来需要知道每个节点需要放在页面上的哪个位置，也就是节点的坐标以及该节点需要占用多大的区域，这个阶段被称为layout布局，主线程通过遍历dom和计算好的样式来生成Layout Tree，Layout tree上的每个节点都记录了x,y坐标和边框尺寸，这里需要注意的一点是DOM Tree和Layout Tree并不是一一对应的，设置了display:none的节点不会出现在Layout Tree上，而在before伪类中添加了content值的元素content里的内容会出现在Layout Tree上，不会出现在DOM树里，这是因为DOM是通过HTML解析获得，并不关心样式，而Layout Tree是根据DOM和计算好的样式来生成，Layout Tree是和最后展示在屏幕上的节点是对应的

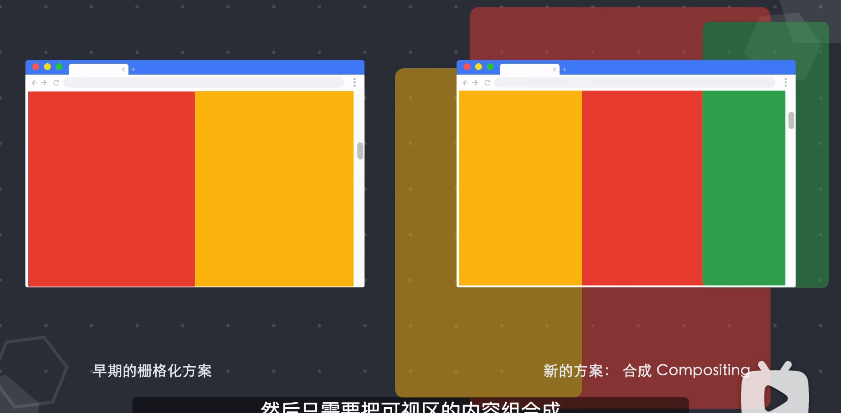


* 1. 现在还需要知道以什么样的顺序绘制(paint)这个节点，举例来说：z-index属性会影响节点绘制的层级关系，如果按照dom的层级结构来绘制页面，则会导致错误的渲染，比如像下面图片中的代码，神烦狗是不应该盖住头的，正确层级应该是头压在狗头上，所以为了保证在屏幕上展示正确的层级，主线程遍历Layout tree创建一个绘制记录表(Paint Record)，该表记录了绘制的顺序，这个阶段被成为绘制（paint）

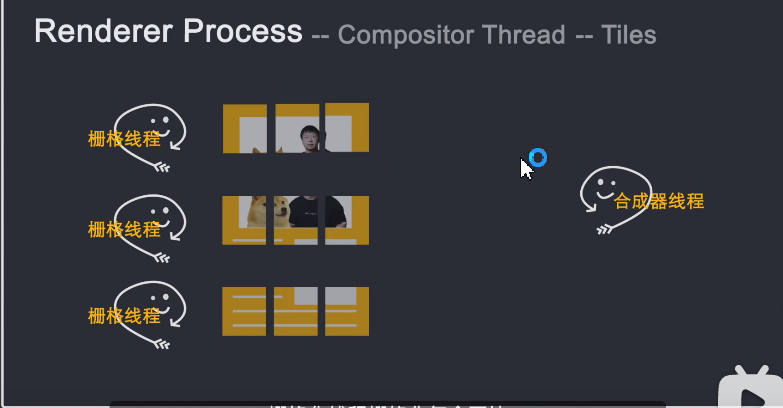




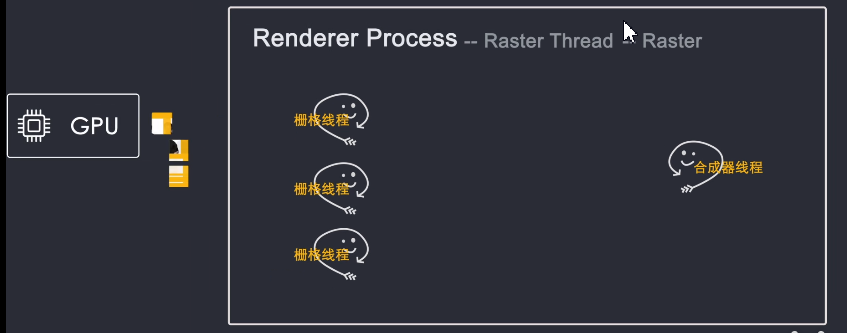
* 1. 现在知道了文档的绘制顺序，终于该把这些信息转化成像素点显示在屏幕上的时候了，这种行为被称为栅格化（Rastering），Chrome最早使用了一种很简单的方式，只栅格化用户可视区域(Viewport)的内容，当用户滚动页面时，再栅格化更多的内容来填充缺失的部分，这种方式带来的问题显而易见，会导致展示延迟，随着不断的优化升级，现在的Chrome使用了一种更为复杂的栅格化流程叫做合成(Composting)，合成是一种将页面的各个部分分成多个图层，分别对其进行栅格化，并在合成器(Compositor Thread)中单独进行合成页面的技术，简单来说就是页面所有的元素按照某种规则进行分图层，并把图层都栅格化好了，然后只需要把可视区的内容和成一帧展示给用户即可



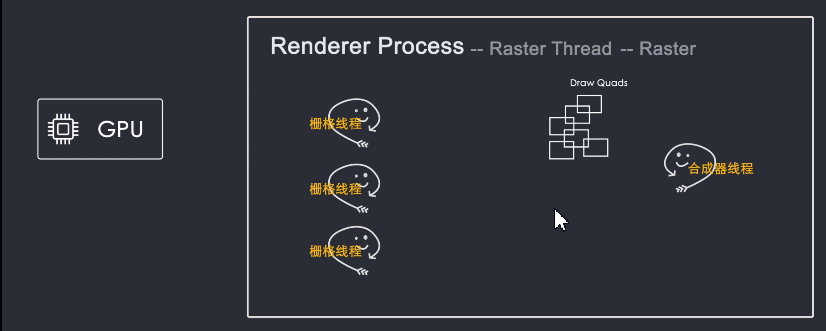
主线程遍历Layout Tree生成Layer（图层）Tree，当Layer Tree生成完毕和绘制顺序确定后，主线程将这些信息传递给合成器线程，合成器线程将每个图层栅格化（由于一层可能像页面的整个长度一样大，因此合成器线程将他们切分为许多图块（tiles）然后将每个图块发送给栅格化线程（Raster Thread））



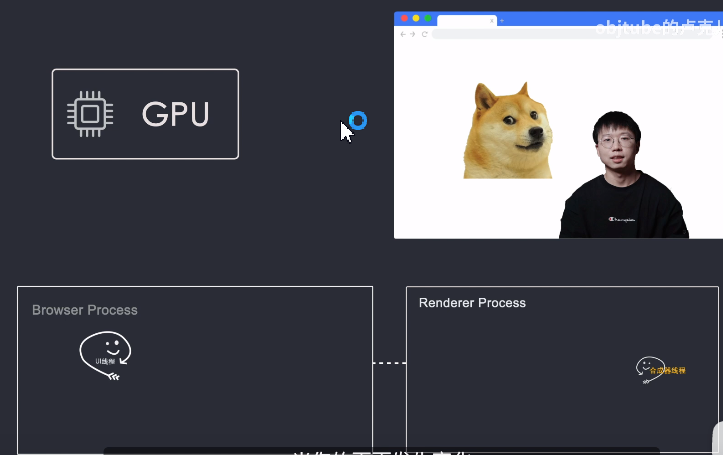
栅格化线程栅格化每个图块并将它们存储在GPU内存中



当图块栅格化完成后，合成器线程将收集称为”draw quads”的图块信息，这些信息记录了图块在内存中的位置和在哪个位置绘制图块的信息



根据这些信息合成器线程生成了一个合成器帧（Compositor Frame），然后这个合成器Frame(帧)通过IPC传递给浏览器进程，接着浏览器进程将合成器帧传送到GPU，然后GPU渲染展示到屏幕上



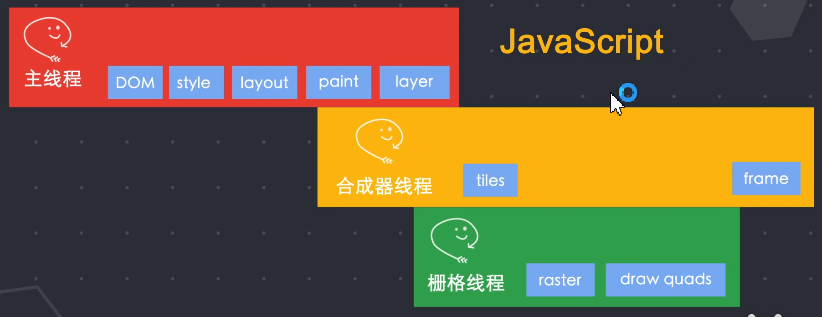
当你的页面发生变化，比如你滚动了当前页面，则会生成一个新的合成器帧，新的帧再传给GPU，然后再次渲染到屏幕上

1. 总结

浏览器进程中的网络线程请求获取到html数据后，通过IPC将数据传给渲染器进程的主线程，主线程将html解析构造DOM树，然后进行样式计算，根据DOM树和生成好的样式生成Layout Tree，通过遍历Layout Tree生成绘制顺序表，接着遍历了Layout Tree生成了Layer Tree，然后主线程将Layer Tree和绘制信息一起传给合成器线程合成器线程按规则进行分图层，并把图层分为更小的图块(tiles)传给栅格化线程进行栅格化，栅格化完成后合成器线程会获得栅格线程传过来的”draw quads”图块信息，根据这些信息合成器上合成了一个合成器帧，然后将该合成器帧通过IPC传回给浏览器进程，浏览器进程再传到GPU进行渲染，最后就展示到你的屏幕了

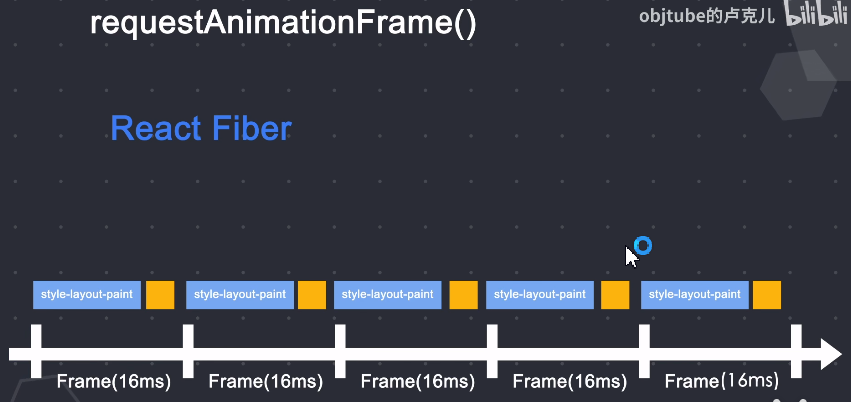
1. 优化

当我们改变一个元素的尺寸位置属性时，会重新进行样式计算(Computed Style)，布局(Layout)绘制(Paint)以及后面的所有流程，这种行为我们成为重排，当我们改变某个元素的颜色属性时，不会重新触发布局，但还是会除法样式计算和绘制，这个就是重绘，我们可以发现重排和重绘都会占用主线程，那还有另外一个东西也是运行在主线上，那就是JS，既然他们都是在主线程运行，都会出现抢占执行时间的问题，如果你写了一个不断导致重排重绘的动画，浏览器则需要在每一帧都运行样式计算布局和绘制的操作。



当我们知道页面以每秒60帧的刷新率时（也就是每帧16ms）才不会让用户感觉到页面卡顿，如果你在运行动画时还有大量的JS任务需要执行，因为布局、绘制和JS执行都是在主线程运行的，当在一帧的时间内布局和绘制结束后，如果还有剩余时间，JS就会拿到主线程的使用权，如果JS执行时间过长就会导致在下一帧开始时JS没有及时归还主线程，导致下一帧动画没有按时渲染，就会出现页面动画的卡顿有什么优化的手段吗？

有！第一种就是可以通过requestAnimationFrame()这个API来帮助我们解决这个问题，这个方法会在每一帧被调用，通过API的回调，然后我们可以把JS运行任务分成一些更小的任务块（分到每一帧），在每一帧时间用完前暂停JS执行，归还主线程，这样的话在下一帧开始时主线程就可以按时执行布局和绘制，React最新的渲染引擎React Fiber就是用到了API来做了很多优化



还有第二个优化方法，通过刚才的流程图我们知道栅格化的整个流程是不占用主线程的，只在合成器线程额和栅格线程中运行这就意味着它无需和JS抢夺主线程，我们刚才提到如果反复进行重绘和重排可能会导致掉帧这是因为有可能JS执行阻塞了主线程，而CSS中有个动画属性叫transform，通过该属性实现的动画不会经过布局和绘制而是直接运行在合成器线程和栅格化线程，所以不会受到主线程中JS执行的影响，更重要的是通过transform实现的动画由于不需要经过布局绘制样式计算等操作所以节省了很多运算时间（方便实现负责的动画），我们常常会使用哪些手段来实现动画效果呢？位置变化、宽高变化（旋转、3D等）这些都是可以使用transform来代替的

附相关资料：

Script标签中的async和defer属性：https://www.cnblogs.com/jiasm/p/7683930.html