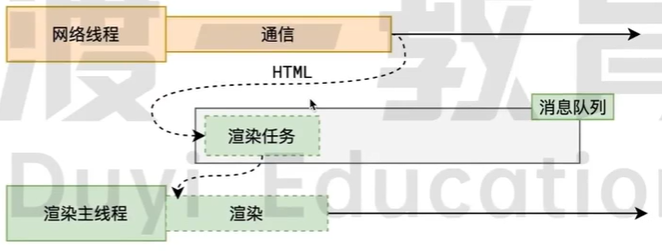
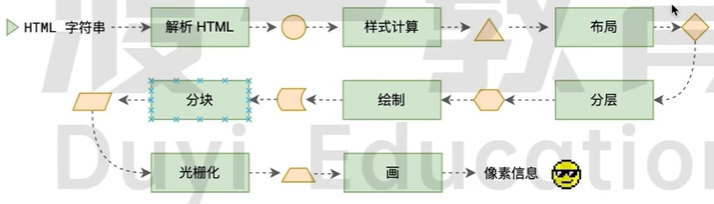
浏览器渲染原理

渲染的本质：将html/css/js字符串，渲染成页面的像素信息，也就是当前页的每个像素是什么颜色，并将像素信息告诉系统（cpu/gpu），使系统能够生成画面



渲染流水线：



每个步骤的输出，都是下一个步骤的输入

第一步：解析html（Parse HTML）

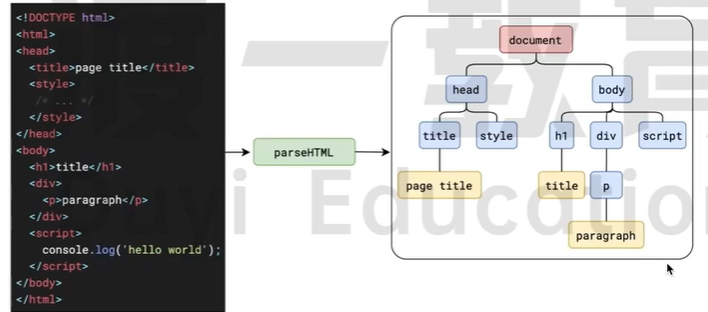
将html文件内的字符串（代码）解析成为两棵树：“DOM树” 和 “CSSOM树”

使浏览器可以直接操作“树”上的元素（对象），不用去和原始的字符串代码交互

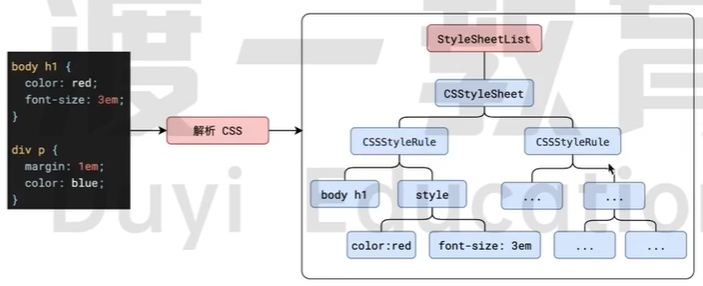
这一步是后续读取css和js的前置条件

Js可以操作“DOM树” 和 “CSSOM树”上的对象

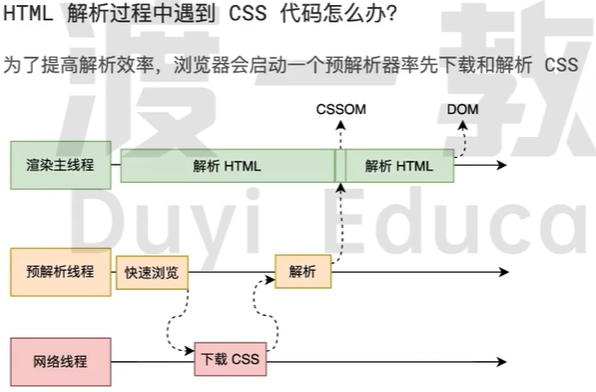
Dom（Document Object Model）树是一个“树型”结构的大对象，也就是说一个document的大对象，包含了多级的小对象



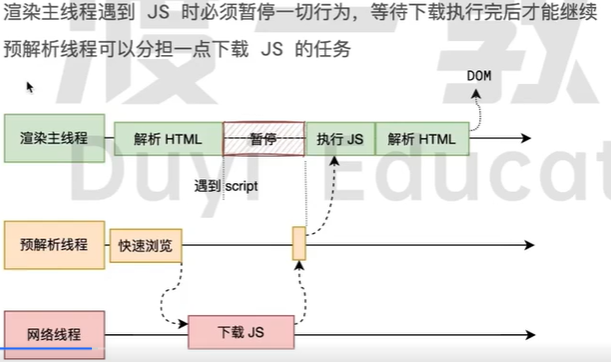
Cssom（CSS Object Model）树同理，包含了所有css样式，包括浏览器默认样式



当html解析过程中遇到css代码怎么办



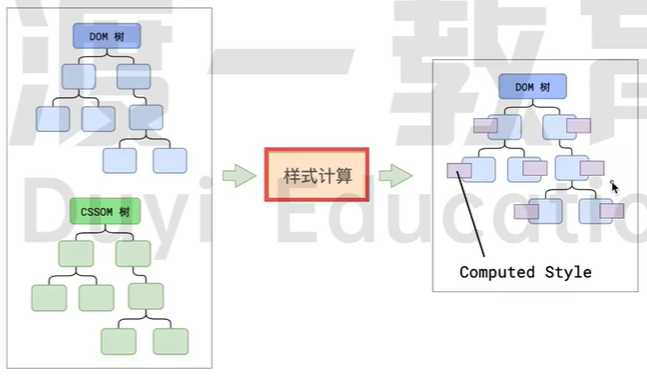
当html解析过程中遇到js代码怎么办



注意！！为什么需要暂停一切行为等待js？因为js可能会更改dom树，所以dom树的生成必须暂停，这也是js为什么会阻塞浏览器运行的根本原因，而遇到css代码则不用暂停，因为css并不会改变dom树

第二步：样式计算（Recalculate Style）

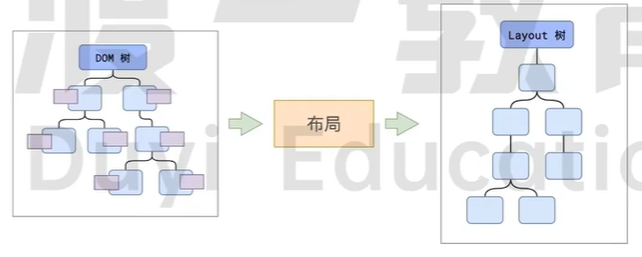
将css样式计算出来（颜色，字体），并挂载到相应的dom元素上



主线程会遍历上一步得到的dom树，依次为树上的每个节点计算出最终样式（computed style），在这一过程中，很多预设值会变成绝对值，比如red会变成rgb（255，0，0），相对单位会变成绝对单位，比如em会变成px，完成这一步之后会得到一颗带有样式的dom树

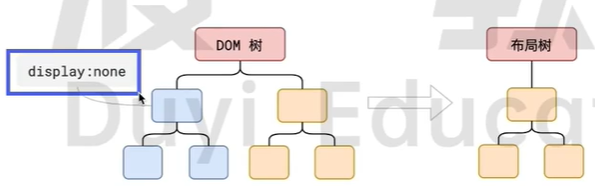
第三步：布局（Layout）

会依次遍历每个节点，计算出各个元素的大小（宽高）以及位置（相对于包含块的位置）

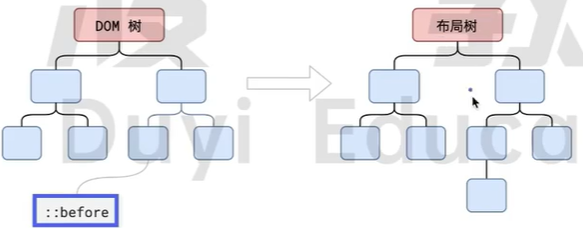


注意！！Dom树和Layout树不一定是一一对应的：

Layout树记录每个元素的几何信息，但有些隐藏块没有几何信息（display：none），没有几何信息的元素就不会生成在Layout树上

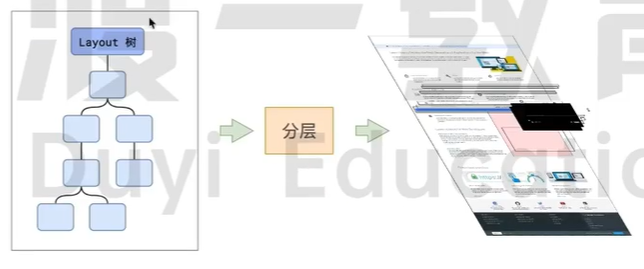


相反，如果存在一个伪元素，有几何信息，则会生成在Layout树上，但因为它是伪元素，反而不会在dom树上生成



第四步：分层（Layer）

将页面划分为多个层次，其中一个层次有改动，并不需要整个页面重新渲染，也不会影响到其他层次的元素，节省内存空间，提高效率



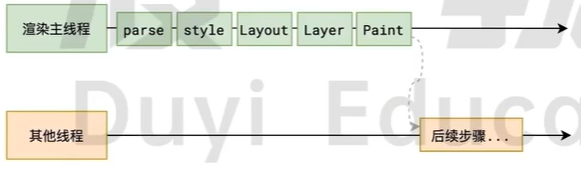
第五步：绘制（Paint）

这里的绘制不是生成像素点，是为每一层生成如何绘制的指令（类似于canvas）

比如：在10，30的位置生成一个矩形，宽高200\*300，背景为红色...

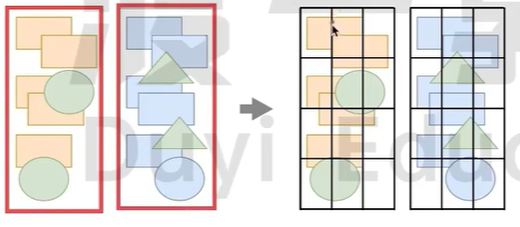


到此，主线程的工作结束，剩余步骤交给其他线程

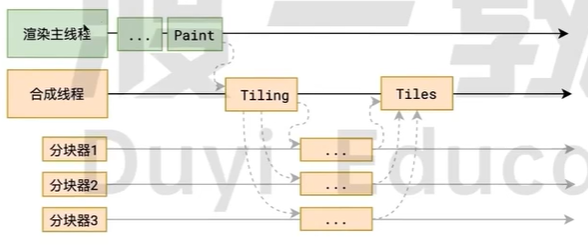


第六步：分块（Tiling）

会将每一层分为多个小的区域，以便光栅化的时候考虑优先级



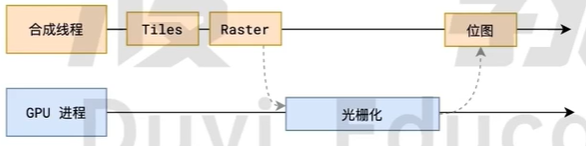
分块工作是由多个线程同时进行的：



第七步：光栅化（Raster）

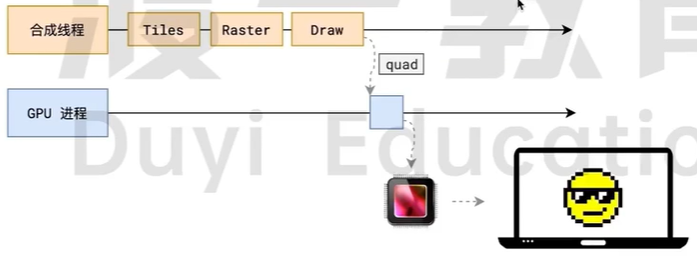
将每个块变成位图（内部记录所有像素点的位置，颜色等信息），优先处理靠近视口的块

GPU来做光栅化的工作



第八步：画（Draw）

合成线程计算出每个位图在屏幕上的位置，交给GPU进行最终呈现



完整流程：

