# 浏览器渲染

## 流程

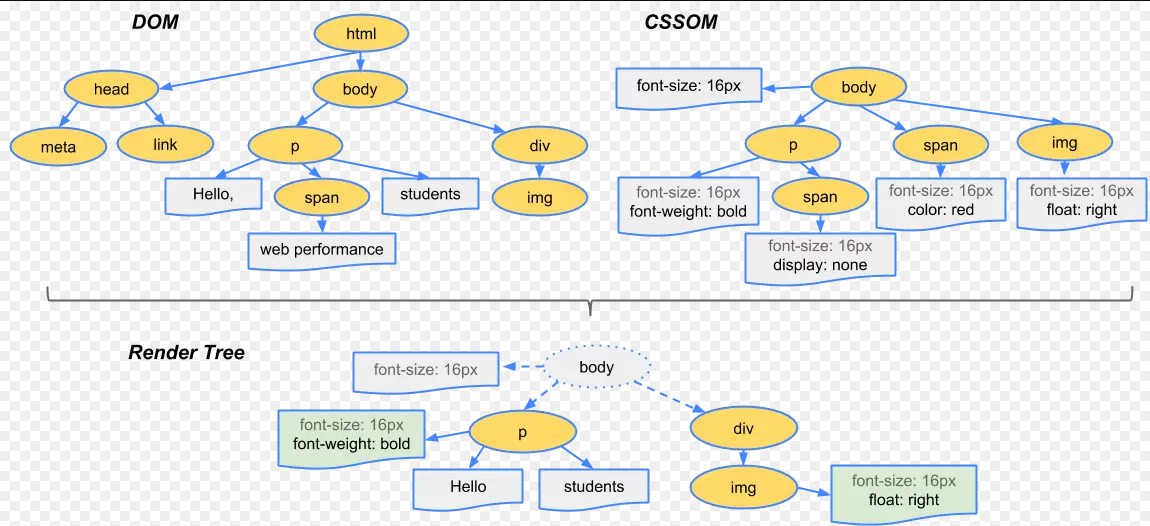
①解析 HTML Source 构建 DOM Tree

②解析CSS Style 构建 CSSOM Tree

③组合 DOM Tree 与 CSSOM Tree，去除不可见元素，构建Render Tree

④执行reflow，根据 Render Tree 计算每个可见的元素的布局

⑤执行repaint， 通过绘制流程，将每个像素渲染到屏幕上



Render Tree 只包含渲染网页所需要的节点（隐藏的元素不渲染）

reflow过程是布局计算元素的精确位置和大小

repaint 过程是将 Render Tree的每个像素渲染到屏幕上

## 绘制

绘制是填充像素的过程，像素最终合成到用户的屏幕上，它往往是管道中运行时间最长的任务，应当避免此任务

注：

1. 除tranform或 opacity之外，更改任何属性都会触发绘制
2. 绘制通常是像素管道中开销最大的部分，应避免
3. 通过层的提升和动画的编排 ，可以有效减少绘制区域
4. 可以使用Chrome DevTools绘制分析绘制的复杂性和开销，尽可能降低复杂性并减少开销。
5. 触发布局一定会触发绘制，但是触发绘制时，不一定现有布局触发。

## 提升移动或淡出的元素

绘制并非总是绘制到内存中的单个图像，在必要时浏览器可以绘制多个图像或合成器层。各个层可以在彼此的上面处理并合并，以创建最终图像。

创建新层的最佳方式是使用will-change CSS 属性（will-change: transform;）。

对于不支持的浏览器，可以滥用3D变形来强制创建新层（transform: translateZ(0);）

注意：

不要随意创建层，每层都需要内存和管理开销。

## 渲染优化

1.减少 JS 逐行修改元素样式：

let body = document.getElementsByTagName('body')[0];

body.className += ' class-name';

2.离线处理 DOM 操作：

①通过 documentFragment 集中处理临时操作；

②克隆节点进行操作，然后进行原节点替换；

③使用 display:none; 进行批量操作。

1. 减少样式的重新计算，即减少 offset、scroll、client\*、getComputedStyle、currentStyle 的使用，因为每次调用都会刷新操作缓冲区，执行 reflow & repaint。
2. 减少绘制区域

绘制一个大问题是，浏览器会将两个需要绘制的区域联合起来，这可能导致整个屏幕重绘（比如：页面顶部和底部都有绘制元素）

减少绘制区域往往是编排动画，使其不过多重叠（在不同的层中）。

1. 降低绘制的复杂性

一些绘制的开销更大（如涉及模糊的元素）

## 调试监控渲染

使用 Chrome DevTools中的rendering

# 浏览器线程

通常情况下，js与绘制、样式更新，用户操作公用的是一个线程。

浏览器中JavaScript和 UI公用一个线程，JavaScript的长时间执行会导致ui渲染和响应被中断。