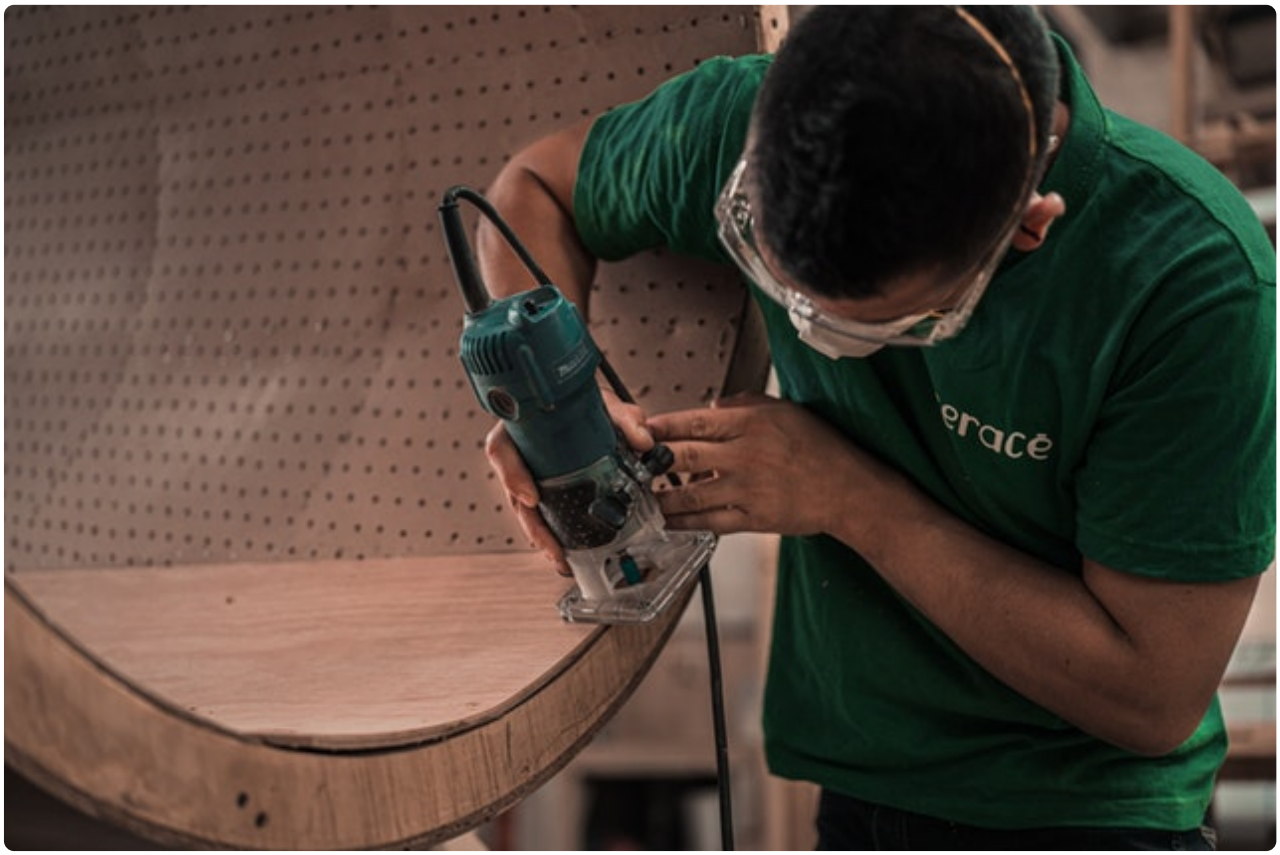
25 图解浏览器渲染引擎工作原理

更新时间：2020-05-26 14:40:39



> 老骥伏枥，志在千里； 烈士暮年，壮心不已。 ——曹操

本节我们介绍浏览器渲染引擎的运作机制。

> Tips: 这块知识非常容易出问答题/作为性能优化面试题的切入点，大家需要引起重视。

渲染引擎工作流解析

通过前面几节的学习，大家现在已经知道，浏览器的渲染引擎承载着把静态资源转换为可视化界面的任务。前面咱们已经明确了这样的转换关系：

中间这个“渲染引擎处理”目前对大家来说仍然是一个黑盒。我们把这个黑盒拆开，会看到它其实包含了以下几个具体流程：

整体来看，这五个过程分别完成了以下任务：

1. HTML解析

在这一步浏览器对HTML文档进行解析，并在解析 HTML 的过程中发出了页面渲染所需的各种外部资源请求。

2. CSS解析

浏览器将识别并加载所有的 CSS 样式信息。

3. 样式与结构合并

将样式信息和文档结构合并，最终生成页面 render 树（:after :before 这样的伪元素会在这个环节被构建到 DOM 树中）。

4. 布局阶段

页面中所有元素的相对位置信息，大小等信息均在这一步得到计算。

5. 页面绘制

在这一步中浏览器会根据我们前面处理出来的结果，把每一个页面图层转换为像素，并对所有的媒体文件进行解码。

这五个步骤可以说是每一步都很关键，每一步都不白干，每一步都有一个阶段性产物作为收获。这些产物是我们理解渲染过程的重要抓手：

阶段性产物分析

这五个流程分别对应了以下五个产物：

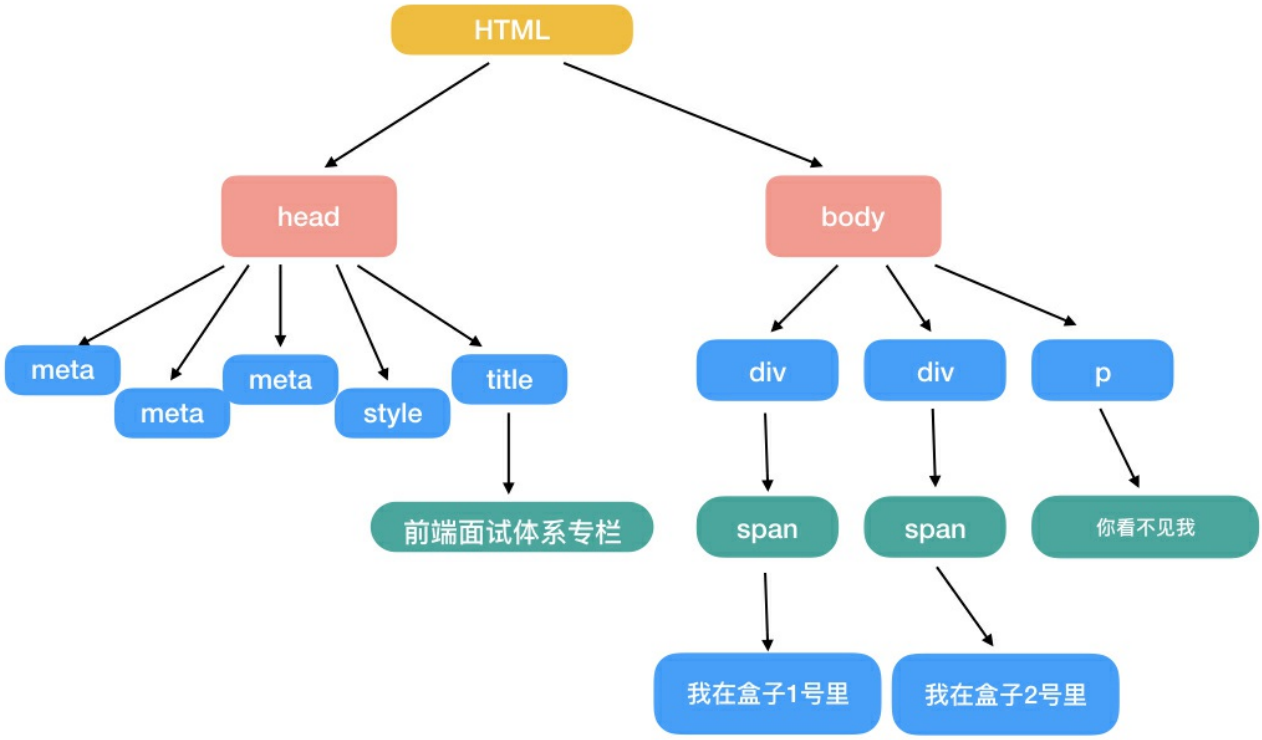
- DOM树
- CSSOM树
- 渲染树
- 盒模型
- 目标界面

认识产物的目的是为了能够更好地掌握创造产物的过程。产物是具体的，因此咱们此处的讲解也应该是具体的。下面我就带大家来观摩一个渲染引擎的工作实例：

HTML解析 —— DOM树

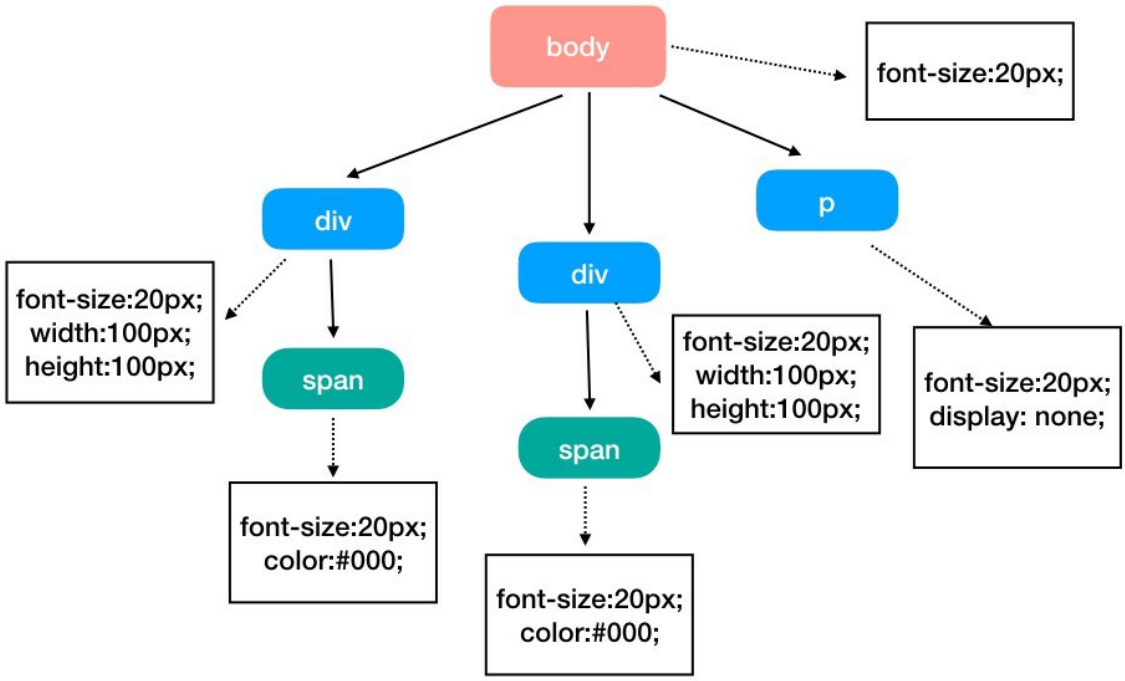
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>前端面试体系专栏</title>
  <style>
    body {
      font-size: 20px;
    }
    div {
      width: 100px;
      height: 100px;
    }
    span {
      color: #000;
    }
    p {
      display: none
    }
  </style>
</head>
<body>
  <div class="box1">
    <span>我在盒子1号里</span>
  </div>
  <div class="box2">
    <span>我在盒子2号里</span>
  </div>
  <p>你看不见我</p>
</body>
</html>
```

楼上我们给出的是一段非常简单的 HTML。相信对大家来说，阅读 HTML 不是什么难事。不过对浏览器来说，这可不是什么美差——浏览器不能够直接理解 HTML，它首先会把它转化成自己能理解的 DOM 树：



CSS 解析 —— CSSOM 树

一样的道理，浏览器也是没法直接理解 CSS 代码的，需要把它处理成自己能理解的 CSSOM 树——注意，虽然我们编写 CSS 代码时不像 HTML 代码一样表现出明显的嵌套关系，**但是 CSSOM 也是具有树结构的**。这是因为在样式计算的过程中，浏览器总是从适用于该节点的最通用规则开始（例如 div 节点是 body 元素的子项，则应用所有 body 样式），一层一层递归细化出具体的样式。这个由通用到具体的细化关系，我们也可以用树结构来描述：



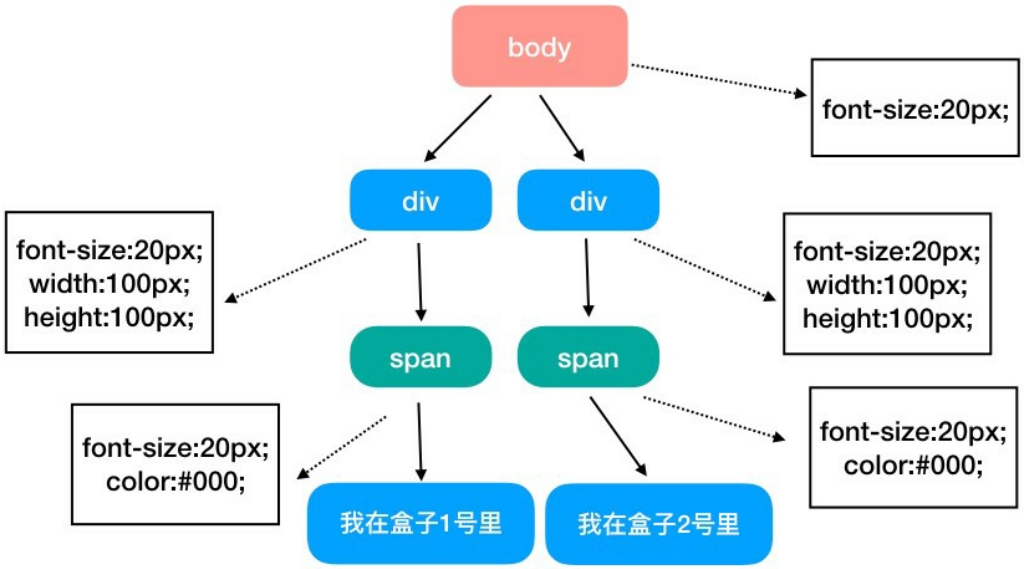
DOM树与CSSOM树“合体”——渲染树

当 DOM 树和 CSSOM 树都解析完毕后，它们就会被结合在一起，构建出 Render Tree（渲染树）。

值得注意的是，渲染树的特点是它**只包含渲染网页所需的节点。**所以在构建渲染树的过程中，除了“结合”之外，浏览器还做了一些关键的小动作，这些小动作可能作为考点出现：

- step1： 从 DOM 树的根节点开始遍历，筛选出所有可见的节点；
- step2：仅针对可见节点，为其匹配 CSSOM 中的 CSS 规则；
- step3：发射可见节点（连同其内容和计算的样式）。

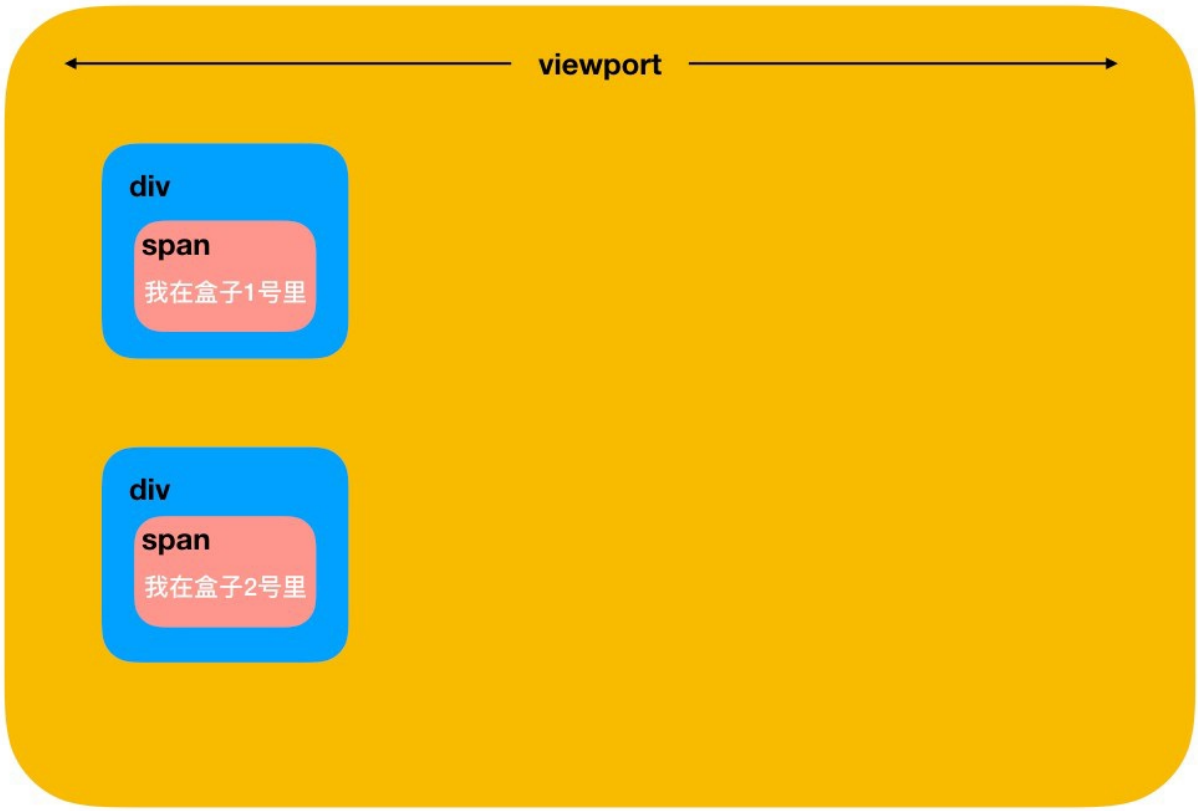
经过这么一顿操作之后，渲染树就到手了：



布局盒子模型

经过咱们不断地构建、修修剪剪，和“树”之间的故事，算是告一段落了。接下来咱们以手里这棵渲染树作为依据，进入布局阶段了。

到现在为止，我们已经掌握了需要渲染的所有节点之间的结构关系及其样式信息。但是我们还不知道它们在渲染时，到底应该出现在浏览器视口的哪个位置上、占据多大的空间——计算这些信息，就是布局阶段要做的事情。浏览器对渲染树进行遍历，将元素间嵌套关系以盒子模型的形式写入文档流：



盒模型在布局过程中会计算出元素确切的大小和定位。计算完毕后，相应的信息被写回渲染树上，就形成了“布局渲染树”。同时，每一个元素盒子也都携带着自身的样式信息，作为后续绘制的依据。

目标界面

大家注意，走到现在这一步，我们浏览器的视窗内实际上还是啥也没展示出来的状态。咱们上述的渲染树也好，盒模型也好，它们都乖乖地躺在内存世界里，悄无声息。

布局阶段结束后，浏览器终于拿到了它绘制页面所需要的所有信息。此时它会将渲染树上的每一个节点转换成我们肉眼可见的像素，最终将页面呈现在我们面前，这个过程就是“绘制”。绘制完成后，目标界面也就在你眼前了：

```
我在盒子1
号里


我在盒子2
号里
```

}