14 | 浏览器：一个浏览器是如何工作的？（阶段五）

winter 2019-02-19

00:00 11:33

讲述：winter 大小：10.59M

你好，我是 winter。我们的浏览器系列已经进行到最后一篇。

在之前的几篇文章中，我们已经经历了把 URL 变成字符流，把字符流变成词（token）流，把词（token）流构造成 DOM 树，把不含样式信息的 DOM 树应用 CSS 规则，变成包含样式信息的 DOM 树，并且根据样式信息，计算了每个元素的位置和大小。

那么，我们最后的步骤，就是根据这些样式信息和大小信息，为每个元素在内存中渲染它的图形，并且把它绘制到对应的位置。

渲染

首先我们来谈谈渲染这个词，渲染也是个外来词，它是英文词 render 的翻译，render 这个词在英文里面，有“导致” “变成”的意思，也有“粉刷墙壁”的意思。

在计算机图形学领域里，英文 render 这个词是一个简写，它是特指把模型变成位图的过程。我们把 render 翻译成“渲染”，是个非常有意思的翻译，中文里“渲染”这个词是一种绘画技法，是指沾清水把墨涂开的意思。

所以，render 翻译成“渲染”，我认为是非常高明的，对 render 这个过程，用国画的渲染手法来概括，是颇有神似的。

我们现在的一些框架，也会把“从数据变成 HTML 代码的过程”称为 render，其实我觉得这是非常具有误导性的，我个人是非常不喜欢这种命名方式，当然了，所谓“文无第一”，在自然语言的范围里，我们很难彻底否定这种用法的合理性。

不过，在本篇文章中，我们可以约定一下，本文中出现的“渲染”一词，统一指的是它在图形学的意义，也就是把模型变成位图的过程。

这里的位图就是在内存里建立一张二维表格，把一张图片的每个像素对应的颜色保存进去（位图信息也是 DOM 树中占据浏览器内存最多的信息，我们在做内存占用优化时，主要就是考虑这一部分）。

浏览器中渲染这个过程，就是把每一个元素对应的盒变成位图。这里的元素包括 HTML 元素和伪元素，一个元素可能对应多个盒（比如 inline 元素，可能会分成多行）。每一个盒对应着一张位图。

这个渲染过程是非常复杂的，但是总体来说，可以分成两个大类：图形和文字。

盒的背景、边框、SVG 元素、阴影等特性，都是需要绘制的图形类。这就像我们实现 HTTP 协议必须要基于 TCP 库一样，这一部分，我们需要一个底层库来支持。

一般的操作系统会提供一个底层库，比如在 Android 中，有大名鼎鼎的 Skia，而 Windows 平台则有 GDI，一般的浏览器会做一个兼容层来处理掉平台差异。

这些盒的特性如何绘制，每一个都有对应的标准规定，而每一个的实现都可以作为一个独立的课题来研究，当年圆角 + 虚线边框，可是难倒了各个浏览器的工程师。考虑到这些知识互相都比较独立，对前端工程师来说也不是特别重要的细节，我们这里就不详细探究了。

盒中的文字，也需要用底层库来支持，叫做字体库。字体库提供读取字体文件的基本能力，它能根据字符的码点抽取出字形。

字形分为像素字形和矢量字形两种。通常的字体，会在 6px 8px 等小尺寸提供像素字形，比较大的尺寸则提供矢量字形。矢量字形本身就需要经过渲染才能继续渲染到元素的位图上去。目前最常用的字体库是 Freetype，这是一个 C++ 编写的开源的字体库。

在最普遍的情况下，渲染过程生成的位图尺寸跟它在上一步排版时占据的尺寸相同。

但是理想和现实是有差距的，很多属性会影响渲染位图的大小，比如阴影，它可能非常巨大，或者渲染到非常遥远的位置，所以为了优化，浏览器实际的实现中会把阴影作为一个独立的盒来处理。

注意，我们这里讲的渲染过程，是不会把子元素绘制到渲染的位图上的，这样，当父子元素的相对位置发生变化时，可以保证渲染的结果能够最大程度被缓存，减少重新渲染。

合成

合成是英文术语 compositing 的翻译，这个过程实际上是一个性能考量，它并非实现浏览器的必要一环。

我们上一小节中讲到，渲染过程不会把子元素渲染到位图上面，合成的过程，就是为一些元素创建一个“合成后的位图”（我们把它称为合成层），把一部分子元素渲染到合成的位图上面。

看到这句话，我想你一定会问问题，到底是为哪些元素创建合成后的位图，把哪些子元素渲染到合成的位图上面呢？

这就是我们要讲的合成的策略。我们前面讲了，合成是一个性能考量，那么合成的目标就是提高性能，根据这个目标，我们建立的原则就是最大限度减少绘制次数原则。

我们举一个极端的例子。如果我们把所有元素都进行合成，比如我们为根元素 html 创建一个合成后的位图，把所有子元素都进行合成，那么会发生什么呢？

那就是，一旦我们用 JavaScript 或者别的什么方式，改变了任何一个 CSS 属性，这份合成后的位图就失效了，我们需要重新绘制所有的元素。

那么如果我们所有的元素都不合成，会怎样呢？结果就是，相当于每次我们都必须要重新绘制所有的元素，这也不是对性能友好的选择。

那么好的合成策略是什么呢，好的合成策略是“猜测”可能变化的元素，把它排除到合成之外。

我们来举个例子：

复制代码

```
<div id="a">
    <div id="b">...</div>
    <div id="c" style="transform:translate(0,0)"></div>
</div>
```

假设我们的合成策略能够把 a、b 两个 div 合成，而不把 c 合成，那么，当我执行以下代码时：

复制代码

```
document.getElementById("c").style.transform = "translate(100px, 0)";
```

我们绘制的时候，就可以只需要绘制 a 和 b 合成好的位图和 c，从而减少了绘制次数。这里需要注意的是，在实际场景中，我们的 b 可能有很多复杂的子元素，所以当合成命中时，性能提升收益非常之高。

目前，主流浏览器一般根据 position、transform 等属性来决定合成策略，来“猜测”这些元素未来可能发生变化。

但是，这样的猜测准确性有限，所以新的 CSS 标准中，规定了 will-change 属性，可以由业务代码来提示浏览器的合成策略，灵活运用这样的特性，可以大大提升合成策略的效果。

绘制

绘制是把“位图最终绘制到屏幕上，变成肉眼可见的图像”的过程，不过，一般来说，浏览器并不需要用代码来处理这个过程，浏览器只需要把最终要显示的位图交给操作系统即可。

一般最终显式的位图位于显存中，也有一些情况下，浏览器只需要把内存中的一张位图提交给操作系统或者显式驱动就可以了，这取决于浏览器运行的环境。不过无论如何，我们把任何位图合成到这个“最终位图”的操作称为绘制。

这个过程听上去非常简单，这是因为在前面两个小节中，我们已经得到了每个元素的位图，并且对它们部分进行了合成，那么绘制过程，实际上就是按照 z-index 把它们依次绘制到屏幕上。

然而如果在实际中这样做, 会带来极其糟糕的性能。

有一个一度非常流行于前端群体的说法，讲做 CSS 性能优化，应该尽量避免"重排"和"重绘"，前者讲的是我们上一课的排版行为，后者模糊地指向了我们本课程三小节讲的三个步骤，而实际上，这个说法大体不能算错，却不够准确。

因为，实际上，“绘制”发生的频率比我们想象中要高得多。我们考虑一个情况：鼠标划过浏览器显示区域。这个过程中，鼠标的每次移动，都造成了重新绘制，如果我们不重新绘制，就会产生大量的鼠标残影。

这个时候，限制绘制的面积就很重要了。如果鼠标某次位置恰巧遮盖了某个较小的元素，我们完全可以重新绘制这个元素来完成我们的目标，当然，简单想想就知道，这种事情不可能总是发生的。

计算机图形学中，我们使用的方案就是“脏矩形”算法，也就是把屏幕均匀地分成若干矩形区域。

当鼠标移动、元素移动或者其它导致需要重绘的场景发生时，我们只重新绘制它所影响到的几个矩形区域就够了。比矩形区域更小的影响最多只会涉及 4 个矩形，大型元素则覆盖多个矩形。

设置合适的矩形区域大小，可以很好地控制绘制时的消耗。设置过大的矩形会造成绘制面积增大，而设置过小的矩形则会造成计算复杂。

我们重新绘制脏矩形区域时，把所有与矩形区域有交集的合成层（位图）的交集部分绘制即可。

总结

在这一节课程中，我们讲解了浏览器中的位图操作部分，这包括了渲染、合成和绘制三个部分。渲染过程把元素变成位图，合成把一部分位图变成合成层，最终的绘制过程把合成层显示到屏幕上。

当绘制完成时，就完成了浏览器的最终任务，把一个 URL 最后变成了一个可以看的网页图像。当然了，我们对每一个部分的讲解，都省略了大量的细节，比如我们今天讲到的绘制，就有意地无视了滚动区域。

尽管如此，对浏览器工作原理的感性认识，仍然可以帮助我们理解很多前端技术的设计和应用技巧，浏览器的工作原理和性能部分非常强相关，我们在实践部分的性能优化部分，会再次跟你做一些探讨。

实际上，如果你认真阅读浏览器系列的课程，是可以用 JavaScript 实现一个玩具浏览器的，我非常希望学习课程的同学中能有人这样做，一旦你做到了，收益会非常大。这就是我今天留给你的课外作业，你可以尝试一下。