

24 | 分层和合成机制：为什么CSS动画比JavaScript高效？

2019-09-28 李兵

浏览器工作原理与实践

[进入课程 >](#)



讲述：李兵

时长 11:39 大小 13.35M



在[上一篇文章](#)中我们分析了 CSS 和 JavaScript 是如何影响到 DOM 树生成的，今天我们继续沿着渲染流水线向下分析，来聊聊 DOM 树之后所发生的事情。

在前面[《05 | 渲染流程（上）：HTML、CSS 和 JavaScript 文件，是如何变成页面的？》](#)文章中，我们介绍过 DOM 树生成之后，还要经历布局、分层、绘制、合成、显示等阶段后才能显示出漂亮的页面。

本文我们主要讲解渲染引擎的分层和合成机制，因为分层和合成机制代表了浏览器最为先进的合成技术，Chrome 团队为了做到这一点，做了大量的优化工作。了解其工作原理，有助于拓宽你的视野，而且也有助于你更加深刻地理解 CSS 动画和 JavaScript 底层工作机制。

显示器是怎么显示图像的

每个显示器都有固定的刷新频率，通常是 60HZ，也就是每秒更新 60 张图片，更新的图片都来自于显卡中一个叫**前缓冲区**的地方，显示器所做的任务很简单，就是每秒固定读取 60 次前缓冲区中的图像，并将读取的图像显示到显示器上。

那么这里显卡做什么呢？

显卡的职责就是合成新的图像，并将图像保存到**后缓冲区**中，一旦显卡把合成的图像写到后缓冲区，系统就会让后缓冲区和前缓冲区互换，这样就能保证显示器能读取到最新显卡合成的图像。通常情况下，显卡的更新频率和显示器的刷新频率是一致的。但有时候，在一些复杂的场景中，显卡处理一张图片的速度会变慢，这样就会造成视觉上的卡顿。

帧 VS 帧率

了解了显示器是怎么显示图像的之后，下面我们再来明确下帧和帧率的概念，因为这是后续一切分析的基础。

当你通过滚动条滚动页面，或者通过手势缩放页面时，屏幕上就会产生动画的效果。之所以你能感觉到有动画的效果，是因为在滚动或者缩放操作时，渲染引擎会通过渲染流水线生成新的图片，并发送到显卡的后缓冲区。

大多数设备屏幕的更新频率是 60 次 / 秒，这也就意味着正常情况下要实现流畅的动画效果，渲染引擎需要每秒更新 60 张图片到显卡的后缓冲区。

我们把渲染流水线生成的每一副图片称为一帧，把渲染流水线每秒更新了多少帧称为帧率，比如滚动过程中 1 秒更新了 60 帧，那么帧率就是 60Hz（或者 60FPS）。

由于用户很容易观察到那些丢失的帧，如果在一次动画过程中，渲染引擎生成某些帧的时间过久，那么用户就会感受到卡顿，这会给用户造成非常不好的印象。

要解决卡顿问题，就要解决每帧生成时间过久的问题，为此 Chrome 对浏览器渲染方式做了大量的工作，其中最卓有成效的策略就是引入了分层和合成机制。分层和合成机制代表了当今最先进的渲染技术，所以接下来我们就来分析下什么是合成和渲染技术。

如何生成一帧图像

不过在开始之前，我们还需要聊一聊渲染引擎是如何生成一帧图像的。这需要回顾下我们前面[《06 | 渲染流程（下）：HTML、CSS 和 JavaScript 文件，是如何变成页面的？》](#)介绍的渲染流水线。关于其中任意一帧的生成方式，有**重排**、**重绘**和**合成**三种方式。

这三种方式的渲染路径是不同的，**通常渲染路径越长，生成图像花费的时间就越多**。比如**重排**，它需要重新根据 CSSOM 和 DOM 来计算布局树，这样生成一幅图片时，会让整个渲染流水线的每个阶段都执行一遍，如果布局复杂的话，就很难保证渲染的效率了。而**重绘**因为没有了重新布局的阶段，操作效率稍微高点，但是依然需要重新计算绘制信息，并触发绘制操作之后的一系列操作。

相较于重排和重绘，**合成**操作的路径就显得非常短了，并不需要触发布局和绘制两个阶段，如果采用了 GPU，那么合成的效率会非常高。

所以，关于渲染引擎生成一帧图像的几种方式，按照效率我们推荐合成方式优先，若实在不能满足需求，那么就再退后一步使用重绘或者重排的方式。

本文我们的焦点在合成上，所以接下来我们就来深入分析下 Chrome 浏览器是怎么实现合成操作的。Chrome 中的合成技术，可以用三个词来概括总结：**分层**、**分块**和**合成**。

分层和合成

通常页面的组成是非常复杂的，有的页面里要实现一些复杂的动画效果，比如点击菜单时弹出菜单的动画特效，滚动鼠标滚轮时页面滚动的动画效果，当然还有一些炫酷的 3D 动画特效。如果没有采用分层机制，从布局树直接生成目标图片的话，那么每次页面有很小的变化时，都会触发重排或者重绘机制，这种“牵一发而动全身”的绘制策略会严重影响页面的渲染效率。

为了提升每帧的渲染效率，Chrome 引入了分层和合成的机制。那该怎么来理解分层和合成机制呢？

你可以把一张网页想象成是由很多个图片叠加在一起的，每个图片就对应一个图层，Chrome 合成器最终将这些图层合成了用于显示页面的图片。如果你熟悉 PhotoShop 的话，就能很好地理解这个过程了，PhotoShop 中一个项目是由很多图层构成的，每个图层都可以是一张单独图片，可以设置透明度、边框阴影，可以旋转或者设置图层的上下位置，将这些图层叠加在一起后，就能呈现出最终的图片了。

在这个过程中，将素材分解为多个图层的操作就称为**分层**，最后将这些图层合并到一起的操作就称为**合成**。所以，分层和合成通常是一起使用的。

考虑到一个页面被划分为两个层，当进行到下一帧的渲染时，上面的一帧可能需要实现某些变换，如平移、旋转、缩放、阴影或者 Alpha 渐变，这时候合成器只需要将两个层进行相应的变化操作就可以了，显卡处理这些操作驾轻就熟，所以这个合成过程时间非常短。

理解了为什么要引入合成和分层机制，下面我们再看看 Chrome 是怎么实现分层和合成机制的。

在 Chrome 的渲染流水线中，**分层体现在生成布局树之后**，渲染引擎会根据布局树的特点将其转换为层树（Layer Tree），层树是渲染流水线后续流程的基础结构。

层树中的每个节点都对应着一个图层，下一步的绘制阶段就依赖于层树中的节点。在[《06 | 渲染流程（下）：HTML、CSS 和 JavaScript 文件，是如何变成页面的？》](#)中我们介绍过，绘制阶段其实并不是真正地绘出图片，而是将绘制指令组合成一个列表，比如一个图层要设置的背景为黑色，并且还要在中间画一个圆形，那么绘制过程会生成 `| Paint BackGroundColor:Black | Paint Circle|` 这样的绘制指令列表，绘制过程就完成了。

有了绘制列表之后，就需要进入光栅化阶段了，光栅化就是按照绘制列表中的指令生成图片。每一个图层都对应一张图片，合成线程有了这些图片之后，会将这些图片合成为“一张”图片，并最终将生成的图片发送到后缓冲区。这就是一个大致的分层、合成流程。

需要重点关注的是，合成操作是在合成线程上完成的，这也就意味着在执行合成操作时，是不会影响到主线程执行的。这就是为什么经常主线程卡住了，但是 CSS 动画依然能执行的原因。

分块

如果说分层是从宏观上提升了渲染效率，那么分块则是从微观层面提升了渲染效率。

通常情况下，页面的内容都要比屏幕大得多，显示一个页面时，如果等待所有的图层都生成完毕，再进行合成的话，会产生一些不必要的开销，也会让合成图片的时间变得更久。

因此，合成线程会将每个图层分割为大小固定的图块，然后优先绘制靠近视口的图块，这样就可以大大加速页面的显示速度。不过有时候，即使只绘制那些优先级最高的图块，也要

耗费不少的时间，因为涉及到一个很关键的因素——**纹理上传**，这是因为从计算机内存上传到 GPU 内存的操作会比较慢。


为了解决这个问题，Chrome 又采取了一个策略：**在首次合成图块的时候使用一个低分辨率的图片**。比如可以是正常分辨率的一半，分辨率减少一半，纹理就减少了四分之三。在首次显示页面内容的时候，将这个低分辨率的图片显示出来，然后合成器继续绘制正常比例的网页内容，当正常比例的网页内容绘制完成后，再替换掉当前显示的低分辨率内容。这种方式尽管会让用户在开始时看到的是低分辨率的内容，但是也比用户在开始时什么都看不到要好。

如何利用分层技术优化代码

通过上面的介绍，相信你已经理解了渲染引擎是怎么将布局树转换为漂亮图片的，理解其中原理之后，你就可以利用分层和合成技术来优化代码了。

在写 Web 应用的时候，你可能经常需要对某个元素做几何形状变换、透明度变换或者一些缩放操作，如果使用 JavaScript 来写这些效果，会牵涉到整个渲染流水线，所以 JavaScript 的绘制效率会非常低下。

这时你可以使用 `will-change` 来告诉渲染引擎你会对该元素做一些特效变换，CSS 代码如下：

 复制代码

```
1 .box {  
2   will-change: transform, opacity;  
3 }
```

这段代码就是提前告诉渲染引擎 `box` 元素将要做几何变换和透明度变换操作，这时候渲染引擎会将该元素单独实现一帧，等这些变换发生时，渲染引擎会通过合成线程直接去处理变换，这些变换并没有涉及到主线程，这样就大大提升了渲染的效率。**这也是 CSS 动画比 JavaScript 动画高效的原因。**

所以，如果涉及到一些可以使用合成线程来处理 CSS 特效或者动画的情况，就尽量使用 `will-change` 来提前告诉渲染引擎，让它为该元素准备独立的层。但是凡事都有两面性，每当渲染引擎为一个元素准备一个独立层的时候，它占用的内存也会大大增加，因为从层树开

始，后续每个阶段都会多一个层结构，这些都需要额外的内存，所以你需要恰当地使用 will-change。

总结

好了，今天就介绍到这里，下面我来总结下今天的内容。


首先我们介绍了显示器显示图像的原理，以及帧和帧率的概念，然后基于帧和帧率我们又介绍渲染引擎是如何实现一帧图像的。通常渲染引擎生成一帧图像有三种方式：重排、重绘和合成。其中重排和重绘操作都是在渲染进程的主线程上执行的，比较耗时；而合成操作是在渲染进程的合成线程上执行的，执行速度快，且不占用主线程。

然后我们重点介绍了浏览器是怎么实现合成的，其技术细节主要可以使用三个词来概括：分层、分块和合成。

最后我们还讲解了 CSS 动画比 JavaScript 动画高效的原因，以及怎么使用 will-change 来优化动画或特效。

思考时间

观察下面代码，结合 Performance 面板、内存面板和分层面板，全面比较在 box 中使用 will-change 和不使用 will-change 的效率、性能和内存占用等情况。

 复制代码

```
1 <html>
2 <head>
3   <title> 观察 will-change</title>
4   <style>
5     .box {
6       will-change: transform, opacity;
7       display: block;
8       float: left;
9       width: 40px;
10      height: 40px;
11      margin: 15px;
12      padding: 10px;
13      border: 1px solid rgb(136, 136, 136);
14      background: rgb(187, 177, 37);
15      border-radius: 30px;
16      transition: border-radius 1s ease-out;
17    }
18
19
20    body {
```

```
21         font-family: Arial;
22     }
23
24
25
26     </style>
27 </head>
28
29
30 <body>
31     <div id="controls">
32         <button id="start">start</button>
33         <button id="stop">stop</button>
34     </div>
35     <div>
36         <div class="box"> 旋转盒子 </div>
37         <div class="box"> 旋转盒子 </div>
38         <div class="box"> 旋转盒子 </div>
39         <div class="box"> 旋转盒子 </div>
40         <div class="box"> 旋转盒子 </div>
41         <div class="box"> 旋转盒子 </div>
42         <div class="box"> 旋转盒子 </div>
43         <div class="box"> 旋转盒子 </div>
44         <div class="box"> 旋转盒子 </div>
45         <div class="box"> 旋转盒子 </div>
46         <div class="box"> 旋转盒子 </div>
47         <div class="box"> 旋转盒子 </div>
48         <div class="box"> 旋转盒子 </div>
49         <div class="box"> 旋转盒子 </div>
50         <div class="box"> 旋转盒子 </div>
51         <div class="box"> 旋转盒子 </div>
52         <div class="box"> 旋转盒子 </div>
53         <div class="box"> 旋转盒子 </div>
54         <div class="box"> 旋转盒子 </div>
55         <div class="box"> 旋转盒子 </div>
56         <div class="box"> 旋转盒子 </div>
57         <div class="box"> 旋转盒子 </div>
58         <div class="box"> 旋转盒子 </div>
59         <div class="box"> 旋转盒子 </div>
60         <div class="box"> 旋转盒子 </div>
61         <div class="box"> 旋转盒子 </div>
62         <div class="box"> 旋转盒子 </div>
63         <div class="box"> 旋转盒子 </div>
64         <div class="box"> 旋转盒子 </div>
65         <div class="box"> 旋转盒子 </div>
66         <div class="box"> 旋转盒子 </div>
67         <div class="box"> 旋转盒子 </div>
68         <div class="box"> 旋转盒子 </div>
69         <div class="box"> 旋转盒子 </div>
70         <div class="box"> 旋转盒子 </div>
71         <div class="box"> 旋转盒子 </div>
72         <div class="box"> 旋转盒子 </div>
```

```
73     <div class="box"> 旋转盒子 </div>
74     <div class="box"> 旋转盒子 </div>
75     <div class="box"> 旋转盒子 </div>
76     <div class="box"> 旋转盒子 </div>
77     <div class="box"> 旋转盒子 </div>
78     <div class="box"> 旋转盒子 </div>
79     <div class="box"> 旋转盒子 </div>
80     <div class="box"> 旋转盒子 </div>
81         <div class="box"> 旋转盒子 </div>
82         <div class="box"> 旋转盒子 </div>
83         <div class="box"> 旋转盒子 </div>
84         <div class="box"> 旋转盒子 </div>
85         <div class="box"> 旋转盒子 </div>
86         <div class="box"> 旋转盒子 </div>
87         <div class="box"> 旋转盒子 </div>
88         <div class="box"> 旋转盒子 </div>
89         <div class="box"> 旋转盒子 </div>
90         <div class="box"> 旋转盒子 </div>
91         <div class="box"> 旋转盒子 </div>
92         <div class="box"> 旋转盒子 </div>
93         <div class="box"> 旋转盒子 </div>
94         <div class="box"> 旋转盒子 </div>
95         <div class="box"> 旋转盒子 </div>
96 </div>
97 <script>
98
99     let boxes = document.querySelectorAll('.box');
100     let boxes1 = document.querySelectorAll('.box1');
101     let start = document.getElementById('start');
102     let stop = document.getElementById('stop');
103     let stop_flag = false
104
105
106     start.addEventListener('click', function () {
107         stop_flag = false
108         requestAnimationFrame(render);
109     })
110
111
112     stop.addEventListener('click', function () {
113         stop_flag = true
114     })
115
116
117     let rotate_ = 0
118     let opacity_ = 0
119     function render() {
120         if(stop_flag)
121             return 0
122         rotate_ = rotate_ + 6
123         if( opacity_ > 1)
124             opacity_ = 0
```



```
125         opacity_ = opacity_ + 0.01
126         let command = 'rotate('+rotate_ + 'deg)';
127         for (let index = 0; index < boxes.length; index++) {
128             boxes[index].style.transform = command
129             boxes[index].style.opacity = opacity_
130         }
131         requestAnimationFrame(render);
132     }
133
134
135
136     </script>
137 </body>
138
139
140 </html>
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



浏览器工作原理与实践

>>> 透过浏览器看懂前端本质

李兵

前盛大创新院高级研究员



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 23 | 渲染流水线：CSS如何影响首次加载时的白屏时间？

精选留言 (10)

写留言



宇宙全栈

2019-09-28

请问老师：既然css动画会跳过重绘阶段，则意味着合成阶段的绘制列表不会变化。但是最终得到的相邻两帧的位图是不一样的。那么在合成阶段，相同的绘制列表是如何绘制出不同的位图的？难道绘制列表是有状态的？还是绘制列表一次能绘制出多张位图？

展开 ∨

作者回复：

记住一点，能直接在合成线程中完成的任务都不会改变图层的内容，如文字信息的改变，布局的改变，颜色的改变，统统不会涉及，涉及到这些内容的变化就要牵涉到重排或者重绘了。

能直接在合成线程中实现的是整个图层的几何变换，透明度变换，阴影等，这些变换都不会影响到图层的内容。

比如滚动页面的时候，整个页面内容没有变化，这时候做的其实是对图层做上下移动，这种操作直接在合成线程里面就可以完成了。

再比如文章题目列子中的旋转操作，如果样式里面使用了will-change，那么这些box元素都会生成单独的一层，那么在旋转操作时，只要在合成线程将这些box图层整体旋转到设置的角度，再拿旋转后的box图层和背景图层合成一张新图片，这个图片就是最终输出的一帧，整个过程都是在合成线程中实现的。



1

4



早起不吃虫

2019-09-28

这篇文章信息量巨大，需要很多的知识储备，老师能不能提供一些课外阅读帮助理解呢，谢谢

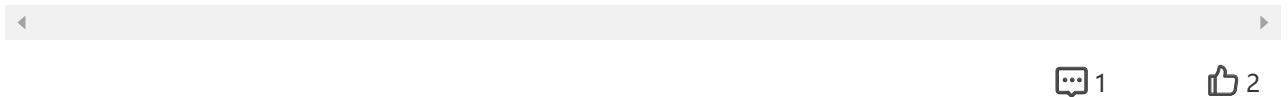
作者回复：这块资料比较少，都是通过chromium源码还有blinkon上一些视频总结的。

blinkon: <https://www.youtube.com/channel/UCIfQb9u7ALnOE4ZmexRecDg>

Chromium源码: <https://chromium.googlesource.com/chromium/src>

https://chromium.googlesource.com/chromium/src/+/_master/docs/README.md

不过源码看起来会比较吃力，里面充斥着大量的回调，梳理起来也是非常不轻松的



晓小东

2019-09-30

单独进程打开两个Tab, 一个tab设置set-will-change, 一个tab no-will-change, 打开浏览器任务管理器查看页面内存情况set-will-change:29M; no-will-change:21M; 有个疑问老师， 如果在一个Tab进行切换时当从no-will-change 到 set-will-change 再到no-will-change， 刷新发现内存变化21M -> 29M -> 29M降不下了 chrome canary版本

展开 ∨



伪装

2019-09-29

will-change有很多的局限性 而且浏览器兼容不是很好 在移动端 cpu开销很大



Angus

2019-09-29

题设的问题答案会不会很牵强？因为使用will-change渲染引擎会通过合成线程去处理元素的变化，所以CSS动画比JavaScript高效？不是应该从CSS动画的原理实现层面去解释吗，will-change只是让CSS动画更高效的一个API，就像JavaScript中的requestAnimationFrame也只是一个优化方案而已。

展开 ∨



Sobine

2019-09-29

老师请教一个问题，spa页面有外链到别人家的网站，新开页面报错如下，error 404—bad request .From RFC 2068 Hypertext Transfer protocol—HTTP/1.1:

展开 ∨



Snow同學

2019-09-29

文中说：我们介绍过 DOM 树生成之后，还要经历布局、分层、绘制、合成，显示。

1.那如何用代码检测页面第一次打开时，元素的合成和显示阶段的完？

2.还有页面显示后，利用ajax请求会内容，在某个节点插入一段html，如何用代码检测新插入的html的合成和显示阶段完成时间？

展开 ▾



易儿易

2019-09-28

大道至简！

展开 ▾



空间

2019-09-28

请教两个问题: 1，我经常使用css动画的方法是用js触发，比如加个css class，或者直接操作element style。这样是否会导致文中这样的css优化效果失效？ 2，能否比较css动画，c anvase 2D动画和webgl动画的性能？比如在插值动画和逐帧动画不同场景下。

展开 ▾



宇宙全栈

2019-09-28

文中这段话中的“帧”应该改为“层”：

这段代码就是提前告诉渲染引擎 box 元素将要做几何变换和透明度变换操作，这时候渲染引擎会将该元素单独实现一帧，等这些变换发生时，渲染引擎会通过合成线程直接去处理变换，这些变换并没有涉及到主线程，这样就大大提升了渲染的效率。

展开 ▾

作者回复: 嗯。多谢指正

