

Dismiss

Join GitHub today

GitHub is home to over 20 million developers working together to host and review code, manage projects, and build software together.

Sign up

谈谈一些有趣的CSS题目（21）-- 提高 CSS 动画性能的正确姿势 | 盒子端 CSS 动画性能提升研究 #11

New issue

Open

chokcoco opened this issue on 27 Mar 2017 · 0 comments



chokcoco commented on 27 Mar 2017 • edited ▼

Owner

不同于传统的 PC Web 或者是移动 WEB，在客厅盒子端，接大屏显示器下，许多能流畅运行于 PC 端、移动端的 Web 动画，受限于硬件水平，在盒子端的表现的往往不尽如人意。

基于此，对于 Web 动画的性能问题，仅仅停留在**感觉已经优化的OK**之上，是不够的，想要在盒子端跑出高性能接近 60 FPS 的流畅动画，就必须刨根问底，深挖每一处可以提升的方法。

流畅动画的标准

理论上说，FPS 越高，动画会越流畅，目前大多数设备的屏幕刷新率为 60 次/秒，所以通常来讲 FPS 为 60frame/s 时动画效果最好，也就是每帧的消耗时间为 16.67ms。

直观感受，不同帧率的体验

- 帧率能够达到 50 ~ 60 FPS 的动画将会相当流畅，让人倍感舒适；
- 帧率在 30 ~ 50 FPS 之间的动画，因各人敏感程度不同，舒适度因人而异；
- 帧率在 30 FPS 以下的动画，让人感觉到明显的卡顿和不适感；
- 帧率波动很大的动画，亦会使人感觉到卡顿。

盒子端动画优化

在客厅盒子端，Web 动画未进行优化之前，一些复杂动画的帧率仅有 10 ~ 30 FPS，卡顿感非常明显，带来很不好的用户体验。

而进行优化之后，能将 10 ~ 30 FPS 的动画优化至 30 ~ 60 FPS，虽然不算优化到最完美，但是当前盒子硬件的条件下，已经算是非常大的进步。

盒子端 Web 动画性能比较

首先先给出在盒子端不同类型的Web 动画的性能比较。经过对比，在盒子端 CSS 动画的性能要优于 Javascript 动画，而在 CSS 动画里，使用 GPU 硬件加速的动画性能要优于不使用硬件加速的性能。

所以在盒子端，实现一个 Web 动画，优先级是：

GPU 硬件加速 CSS 动画 > 非硬件加速 CSS 动画 > Javascript 动画

动画性能上报分析

要有优化，就必须得有数据做为支撑。对比优化前后是否有提升。而对于动画而言，衡量一个动画的标准也就是 FPS 值。

所以现在的关键是如何计算出每个动画运行时的帧率，这里我使用的是 `requestAnimationFrame` 这个函数近似的得到动画运行时的帧率。

原理是，正常而言 `requestAnimationFrame` 这个方法在一秒内会执行 60 次，也就是不掉帧的情况下。假设动画在时间 A 开始执行，在时间 B 结束，耗时 x ms。而中间 `requestAnimationFrame` 一共执行了 n 次，则此段动画的帧率大致为： $n / (B - A)$ 。

Assignees

No one assigned

Labels

性能


Projects

None yet

Milestone

No milestone

1 participant



核心代码如下，能近似计算每秒页面帧率，以及我们额外记录一个 `allFrameCount`，用于记录 rAF 的执行次数，用于计算每次动画的帧率：

```
var rAF = function () {
  return (
    window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    function (callback) {
      window.setTimeout(callback, 1000 / 60);
    }
  );
}();

var frame = 0;
var allFrameCount = 0;
var lastTime = Date.now();
var lastFameTime = Date.now();

var loop = function () {
  var now = Date.now();
  var fs = (now - lastFameTime);
  var fps = Math.round(1000 / fs);

  lastFameTime = now;
  // 不置 0，在动画的开头及结尾记录此值的差值算出 FPS
  allFrameCount++;
  frame++;

  if (now > 1000 + lastTime) {
    var fps = Math.round((frame * 1000) / (now - lastTime));
    // console.log('fps', fps); 每秒 FPS
    frame = 0;
    lastTime = now;
  };

  rAF(loop);
}
```

研究结论

所以，我们的目标就是在使用 GPU 硬件加速的基础之上，更深入的去优化 CSS 动画，先给出最后的一个优化步骤方案：

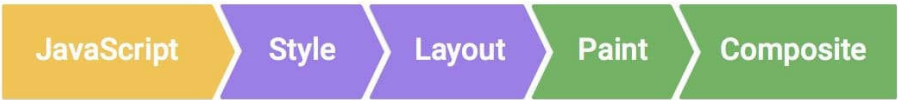
- 1. 精简 DOM，合理布局
- 2. 使用 transform 代替 left、top，减少使用耗性能样式
- 3. 控制频繁动画的层级关系
- 4. 考虑使用 will-change
- 5. 使用 dev-tool 时间线 timeline 观察，找出导致高耗时、掉帧的关键操作

下文会有每一步骤的具体分析解释。

Web 每一帧的渲染

要想达到 60 FPS，每帧的预算时间仅比 16 毫秒多一点 (1 秒/ 60 = 16.67 毫秒)。但实际上，浏览器有整理工作要做，因此您的所有工作需要尽量在 10 毫秒内完成。

而每一帧，如果有必要，我们能控制的部分，也是像素至屏幕管道中的关键步骤如下：



完整的像素管道 JS / CSS > 样式 > 布局 > 绘制 > 合成：

- 1. JavaScript。一般来说，我们会使用 JavaScript 来实现一些视觉变化的效果。比如用 jQuery 的 animate 函数做一个动画、对一个数据集进行排序或者往页面里添加一些 DOM 元素等。当然，除了 JavaScript，还有其他一些常用方法也可以实现视觉变化效果，比如：CSS Animations、Transitions 和 Web Animation API。
- 2. 样式计算。此过程是根据匹配选择器（例如 .headline 或 .nav > .nav_item）计算出哪些元素应用哪些 CSS 3. 规则的过程。从中知道规则之后，将应用规则并计算每个元素的最终样式。

3. 布局。在知道对一个元素应用哪些规则之后，浏览器即可开始计算它要占据的空间大小及其在屏幕的位置。网页的布局模式意味着一个元素可能影响其他元素，例如 元素的宽度一般会影响其子元素的宽度以及树中各处的节点，因此对于浏览器来说，布局过程是经常发生的。
4. 绘制。绘制是填充像素的过程。它涉及绘出文本、颜色、图像、边框和阴影，基本上包括元素的每个可视部分。绘制一般是在多个表面（ 通常称为层 ）上完成的。
5. 合成。由于页面的各部分可能被绘制到多层，由此它们需要按正确顺序绘制到屏幕上，以便正确渲染页面。对于与另一元素重叠的元素来说，这点特别重要，因为一个错误可能使一个元素错误地出现在另一个元素的上层。

当然，不一定每帧都总是会经过管道每个部分的处理。我们的目标就是，**每一帧的动画，对于上述的管道流程，能避免则避免，不能避免则最大限度优化。**

优化动画步骤

先给出一个步骤，调优一个动画，有一定的指导原则可以遵循，一步一步深入动画:

1.精简 DOM ，合理布局

这个没什么好说的，如果可以，精简 DOM 结构在任何时候都是对页面有帮助的。

2.使用 transform 代替 left、top，减少使用耗性能样式

现代浏览器在完成以下四种属性的动画时，消耗成本较低：

- position (位置)：transform: translate(npx, npx)
- scale (比例缩放)：transform: scale(n)
- rotation (旋转)：transform: rotate(ndeg)
- opacity (透明度)：opacity: 0...1

如果可以，尽量只使用上述四种属性去控制动画。

不同样式在消耗性能方面是不同的，改变一些属性的开销比改变其他属性要多，因此更可能使动画卡顿。

例如，与改变元素的文本颜色相比，改变元素的 box-shadow 将需要开销大很多的绘图操作。改变元素的 width 可能比改变其 transform 要多一些开销。如 box-shadow 属性，从渲染角度来讲十分耗性能，原因就是与其他样式相比，它们的绘制代码执行时间过长。

这就是说，如果一个耗性能严重的样式经常需要重绘，那么你就会遇到性能问题。其次你要知道，没有不变的事情，在今天性能很差的样式，可能明天就被优化，并且浏览器之间也存在差异。

开启 GPU 硬件加速

归根结底，上述四种属性的动画消耗较低的原因是会开启了 GPU 硬件加速。动画元素生成了自己的图形层（GraphicsLayer）。

通常而言，开启 GPU 加速的方法我们可以使用

- will-change: transform

这会使声明了该样式属性的元素生成一个图形层，告诉浏览器接下来该元素将会进行 transform 变换，让浏览器提前做好准备。

使用 will-change 并不一定会有性能的提升，因为即使浏览器预料到会有这些更改，依然会为这些属性运行布局和绘制流程，所以提前告诉浏览器，也并不会有太多性能上的提升。这样做的好处是，创建新的图层代价很高，而等到需要时匆忙地创建，不如一开始直接创建好。

对于 Safari 及一些旧版本浏览器，它们不能识别 will-change，则需要使用某种 translate 3D 进行 hack，通常会使用

- transform: translateZ(0)

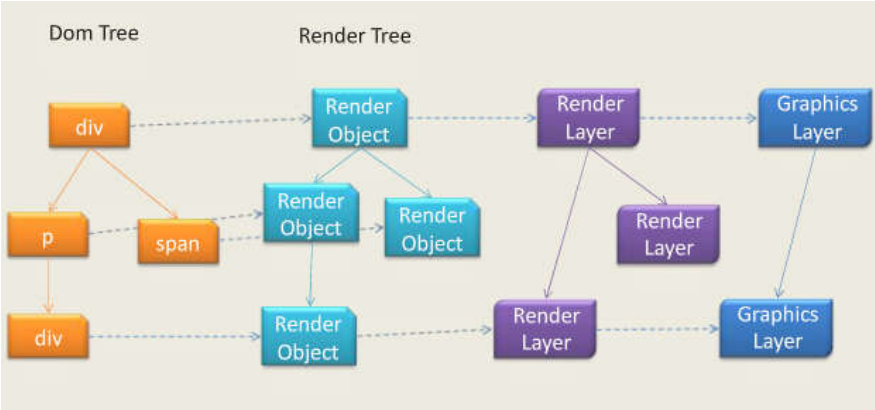
所以，正常而言，在生产环境下，我们可能需要使用如下代码，开启硬件加速：

```
{
  will-change: transform;
  transform: translateZ(0);
}
```

3.控制频繁动画的层级关系

动画层级的控制的意思是**尽量让需要进行 CSS 动画的元素的 z-index 保持在页面最上方，避免浏览器创建不必要的图形层（GraphicsLayer）**，能够很好的提升渲染性能。

OK，这里又提到了图形层（GraphicsLayer），这是一个浏览器渲染原理相关的知识（WebKit/blink内核下）。它能对动画进行加速，但同时也存在相应的加速坑！



简单来说，浏览器为了提升动画的性能，为了在动画的每一帧的过程中不必每次都重新绘制整个页面。在特定方式下可以触发生成一个合成层，合成层拥有单独的 GraphicsLayer。

需要进行动画的元素包含在这个合成层之下，这样动画的每一帧只需要去重新绘制这个 Graphics Layer 即可，从而达到提升动画性能的目的。

那么一个元素什么时候会触发创建一个 Graphics Layer 层？从目前来说，满足以下任意情况便会创建层：

- 硬件加速的 iframe 元素（比如 iframe 嵌入的页面中有合成层）
- 硬件加速的插件，比如 flash 等等
- 使用加速视频解码的 <video> 元素
- 3D 或者 硬件加速的 2D Canvas 元素
- 3D 或透视变换 (perspective、transform) 的 CSS 属性
- 对自己的 opacity 做 CSS 动画或使用一个动画变换的元素
- 拥有加速 CSS 过滤器的元素
- 元素有一个包含复合层的后代节点(换句话说，就是一个元素拥有一个子元素，该子元素在自己的层里)
- 元素有一个 z-index 较低且包含一个复合层的兄弟元素

本小点中说到的动画层级的控制，原因就在于上面生成层的最后一条：

元素有一个 z-index 较低且包含一个复合层的兄弟元素。

这里是存在坑的地方，首先我们要明确两点：

1. 我们希望我们的动画得到 GPU 硬件加速，所以我们会利用类似 transform: translateZ() 这样的方式生成一个 Graphics Layer 层。
2. Graphics Layer 虽好，但不是越多越好，每一帧的渲染内核都会去遍历计算当前所有的 Graphics Layer，并计算他们下一帧的重绘区域，所以过量的 Graphics Layer 计算也会给渲染造成性能影响。

记住这两点之后，回到上面我们说的坑。

假设我们有一个轮播图，有一个 ul 列表，结构如下：

```
<div class="container">
  <div class="swiper">轮播图</div>
  <ul class="list">
    <li>列表li</li>
    <li>列表li</li>
    <li>列表li</li>
    <li>列表li</li>
  </ul>
</div>
```

假设给他们定义如下 CSS：

```
.swiper {
  position: static;
  animation: 10s move infinite;
}

.list {
  position: relative;
```

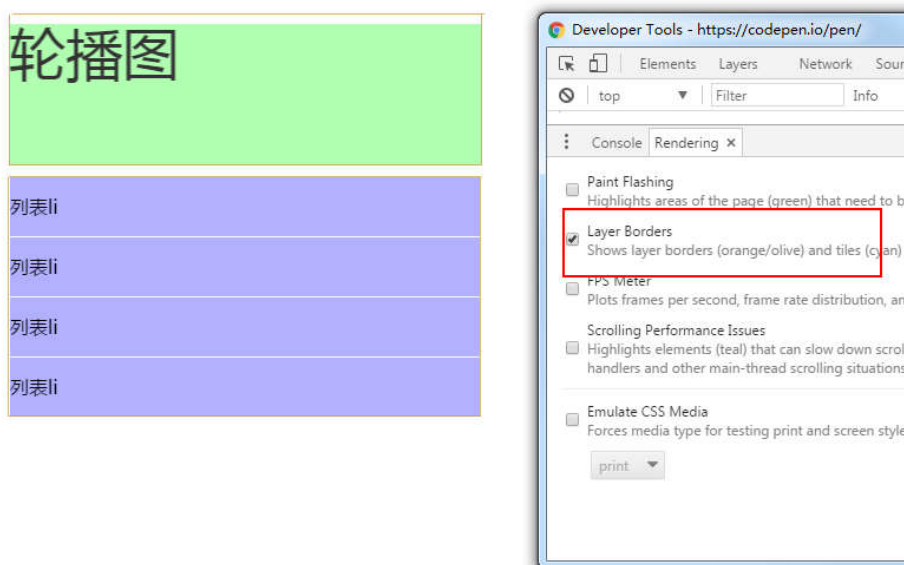
```

}

@keyframes move {
  100% {
    transform: translate3d(10px, 0, 0);
  }
}

```

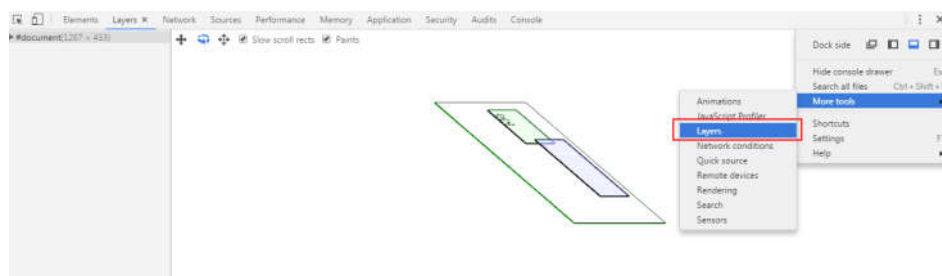
由于给 `.swiper` 添加了 `translate3d(10px, 0, 0)` 动画，所以它会生成一个 Graphics Layer，如下图所示，用开发者工具可以打开层的展示，图形外的黄色边框即代表生成了一个独立的复合层，拥有独立的 Graphics Layer。



但是！在上面的图中，我们并没有给下面的 `list` 也添加任何能触发生成 Graphics Layer 的属性，但是它也同样也有黄色的边框，生成了一个独立的复合层。

原因在于上面那条元素有一个 `z-index` 较低且包含一个复合层的兄弟元素。我们并不希望 `list` 元素也生成 Graphics Layer，但是由于 CSS 层级定义原因，下面的 `list` 的层级高于上面的 `swiper`，所以它被动的也生成了一个 Graphics Layer。

使用 Chrome，我们也可以观察到这种层级关系，可以看到 `.list` 的层级高于 `.swiper`：



所以，下面我们修改一下 CSS，改成：

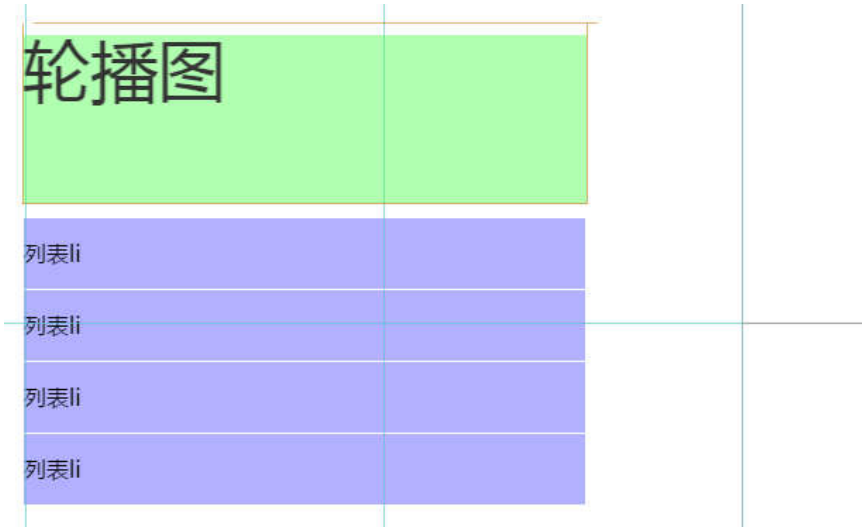
```

.swiper {
  position: relative;
  z-index: 100;
}

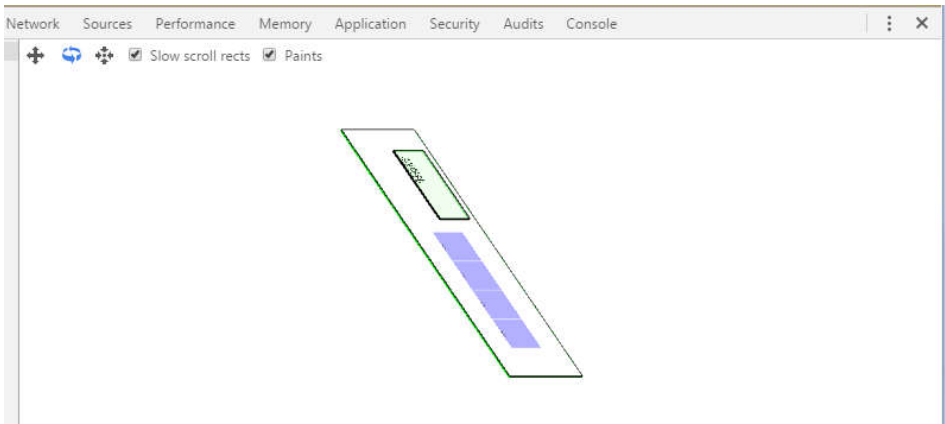
.list {
  position: relative;
}

```

这里，我们明确使得 `.swiper` 的层级高于 `.list`，再打开开发者工具观察一下：



可以看到，这一次，`.list` 元素已经没有了黄色外边框，说明此时没有生成 Graphics Layer 。再看看层级图：



此时，层级关系才是我们希望看到的，`.list` 元素没有触发生成 Graphics Layer 。而我们希望需要硬件加速的 `.swiper` 保持在最上方，每次动画过程中只会独立重绘这部分区域。

总结

这个坑最早见于张云龙发布的这篇文章[CSS3硬件加速也有坑](#)，这里还要总结补充的是：

- GPU 硬件加速也会有坑，当我们希望使用利用类似 `transform: translate3d()` 这样的方式开启 GPU 硬件加速，一定要注意元素层级的关系，尽量保持让需要进行 CSS 动画的元素的 `z-index` 保持在页面最上方。
- Graphics Layer 不是越多越好，每一帧的渲染内核都会去遍历计算当前所有的 Graphics Layer ，并计算他们下一帧的重绘区域，所以过量的 Graphics Layer 计算也会给渲染造成性能影响。
- 可以使用 Chrome ，用上面介绍的两个工具对自己的页面生成的 Graphics Layer 和元素层级进行观察然后进行相应修改。
- 上面观察页面层级的 chrome 工具非常吃内存？好像还是一个处于实验室的功能，分析稍微大一点的页面容易直接卡死，所以要多学会使用第一种观察黄色边框的方式查看页面生成的 Graphics Layer 这种方式。

4. 使用 will-change 可以在元素属性真正发生变化之前提前做好对应准备

```
// 示例
.example{
  will-change: transform;
}
```

上面已经提到过 `will-change` 了。

`will-change` 为 web 开发者提供了一种告知浏览器该元素会有哪些变化的方法，这样浏览器可以在元素属性真正发生变化之前提前做好对应的优化准备工作。这种优化可以将一部分复杂的计算工作提前准备好，使页面的反应更为快速灵敏。

值得注意的是，用好这个属性并不是很容易：

- 在一些低端盒子上，will-change 会导致很多小问题，譬如会使图片模糊，有的时候很容易适得其反，所以使用的时候还需要多加测试。
- 不要将 will-change 应用到太多元素上：浏览器已经尽力尝试去优化一切可以优化的东西了。有一些更强大的优化，如果与 will-change 结合在一起的话，有可能会消耗很多机器资源，如果过度使用的话，可能导致页面响应缓慢或者消耗非常多的资源。
- 有节制地使用：通常，当元素恢复到初始状态时，浏览器会丢弃掉之前做的优化工作。但是如果直接在样式表中显式声明了 will-change 属性，则表示目标元素可能会经常变化，浏览器会将优化工作保存得比之前更久。所以最佳实践是当元素变化之前和之后通过脚本来切换 will-change 的值。
- 不要过早应用 will-change 优化：如果你的页面在性能方面没什么问题，则不要添加 will-change 属性来榨取一丁点的速度。will-change 的设计初衷是作为最后的优化手段，用来尝试解决现有的性能问题。它不应该被用来预防性能问题。过度使用 will-change 会导致生成大量图层，进而导致大量的内存占用，并会导致更复杂的渲染过程，因为浏览器会试图准备可能存在的变化过程，这会导致更严重的性能问题。
- 给它足够的工作时间：这个属性是用来让页面开发者告知浏览器哪些属性可能会变化的。然后浏览器可以选择在变化发生前提前去做一些优化工作。所以给浏览器一点时间去真正做这些优化工作是非常重要的。使用时需要尝试去找到一些方法提前一定时间获知元素可能发生的变化，然后为它加上 will-change 属性。

5. 使用 dev-tool 时间线 timeline 观察，找出导致高耗时、掉帧的关键操作

- 1) 对比屏幕快照，观察每一帧包含的内容及具体的操作
- 2) 找到掉帧的那一帧，分析该帧内不同步骤的耗时占比，进行有针对性的优化
- 3) 观察是否存在内存泄漏

对于 timeline 的使用用法，这里有个非常好的教程，通俗易懂，可以看看：

[浏览器渲染优化 Udacity 课程](#)

对于盒子端 CSS 动画的性能，很多方面仍处于探索中，本文的优化方案研究同样适用于 PC Web 及移动 Web，文章难免有错误及疏漏，欢迎不吝赐教。



2

chokcoco changed the title from 21、 to 21、谈谈一些有趣的CSS题目（21） -- 提高 CSS 动画性能的正确姿势 on 27 Mar 2017

chokcoco changed the title from 21、谈谈一些有趣的CSS题目（21） -- 提高 CSS 动画性能的正确姿势 to 谈谈一些有趣的CSS题目（21） -- 提高 CSS 动画性能的正确姿势 on 27 Mar 2017

chokcoco referenced this issue on 13 Aug 2017
CSS新特性contain，控制页面的重绘与重排 #23

Open

chokcoco changed the title from 谈谈一些有趣的CSS题目（21） -- 提高 CSS 动画性能的正确姿势 to 谈谈一些有趣的CSS题目（21） -- 提高 CSS 动画性能的正确姿势 | 盒子端 CSS 动画性能提升研究 on 23 Sep 2017

chokcoco added the `性能` label on 26 Sep 2017