



下载APP



## 29 | 怎么给Canvas绘制加速？

2020-08-28 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 16:40 大小 15.27M



你好，我是月影。

上节课，我们从宏观上了解了各个图形系统在性能方面的优劣，以及影响性能的要素。实际上，想要解决性能问题，我们就必须要知道真正消耗性能的点，从而结合项目需求进行有针对性的处理，否则性能优化就是纸上谈兵、空中楼阁。

所以这节课，我们就深入讨论一下影响 Canvas 绘图性能的因素，一起来分析几个不同类型的 Canvas 项目，找到的性能瓶颈以及对应的解决办法，从而学会对大部分 Canvas 项目进行性能优化。



我们知道，Canvas 是指令式绘图系统，它有状态设置指令、绘图指令以及真正的绘图方法（fill 和 stroke）等各类 API。通常情况下利用 Canvas 绘图，我们要先调用状态设置指令

设置绘图状态，然后用绘图指令决定要绘制的图形，最后调用真正的 `fill()` 或 `stroke()` 方法将内容输出到画布上。

那结合上节课的实验我们知道，影响 Canvas 性能的两大因素分别是图形的数量和图形的大小。它们都会直接影响绘图指令，一个决定了绘图指令的多少，另一个决定了绘图指令的执行时间。通常来说，绘图指令越多、执行时间越长，渲染效率就越低，性能也就越差。

因此，我们想要对 Canvas 性能进行优化，最重要的就是优化渲染效率。常用的手段有 5 种，分别是优化 Canvas 指令、使用缓存、分层渲染、局部重绘和优化滤镜。此外，还有一种手段叫做**多线程渲染**，是用来优化非渲染的计算和交互方面导致的性能问题。

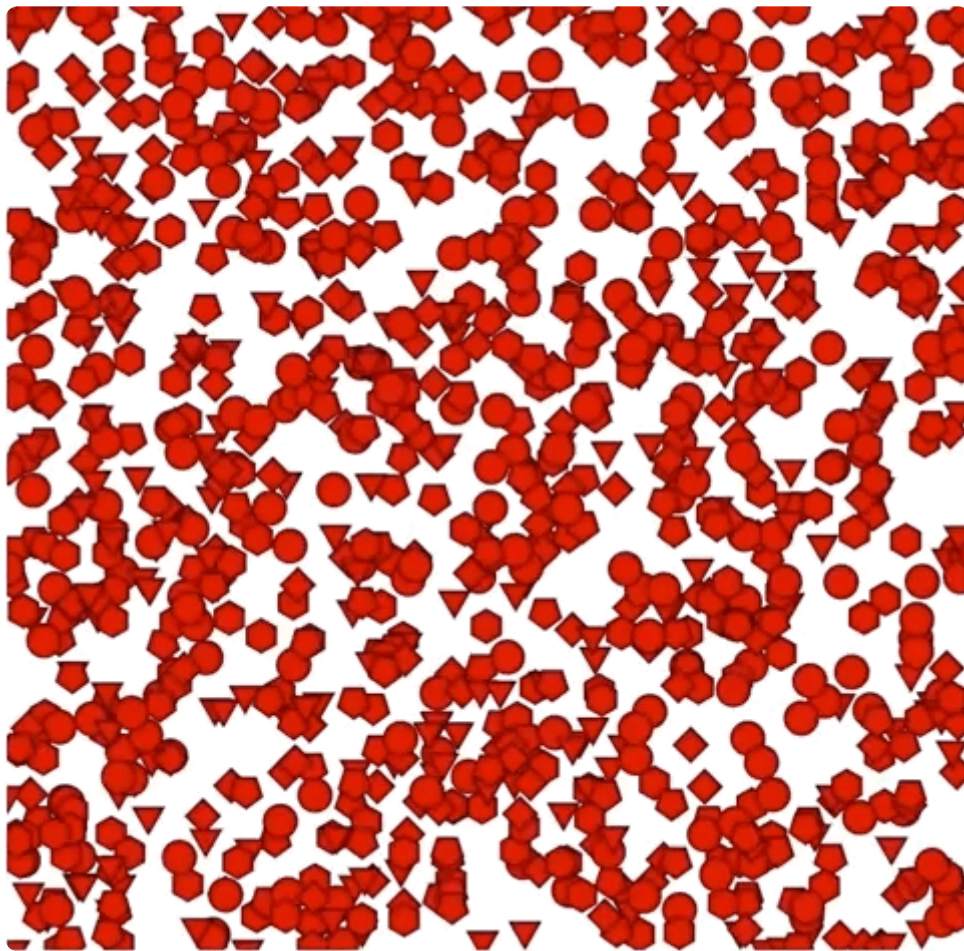
首先，我们来说说优化 Canvas 指令。

## 手段一：优化 Canvas 指令

刚刚我们说了，Canvas 执行的绘图指令越多，性能消耗就越大。那如果希望 Canvas 绘图达到更好的性能，我们要尽可能减少绘图指令的数量。这就是“优化 Canvas 指令”要做的事情。

那具体怎么做呢？我们看一个例子。

假设，我们要在一个 600 X 600 的画布上，实现一些位置随机的多边形，并且不断刷新这些图形的形状和位置，效果如下：



结合我们之前学过的知识，这个效果其实并不难实现，可以分为 4 步，分别是创建多边形的顶点，根据顶点绘制图形，生成随机多边形，执行绘制。

具体的实现代码如下：

复制代码

```
1 const canvas = document.querySelector('canvas');
2 const ctx = canvas.getContext('2d');
3
4 // 创建正多边形，返回顶点
5 function regularShape(x, y, r, edges = 3) {
6   const points = [];
7   const delta = 2 * Math.PI / edges;
8   for(let i = 0; i < edges; i++) {
9     const theta = i * delta;
10    points.push([x + r * Math.sin(theta), y + r * Math.cos(theta)]);
11  }
12  return points;
13 }
14
15 // 根据顶点绘制图形
16 function drawShape(context, points) {
17   context.fillStyle = 'red';
18   context.strokeStyle = 'black';
```

```
19 context.lineWidth = 2;
20 context.beginPath();
21 context.moveTo(...points[0]);
22 for(let i = 1; i < points.length; i++) {
23     context.lineTo(...points[i]);
24 }
25 context.closePath();
26 context.stroke();
27 context.fill();
28 }
29
30 // 多边形类型, 包括正三角形、正四边形、正五边形、正六边形和正100边形
31 const shapeTypes = [3, 4, 5, 6, 100];
32 const COUNT = 1000;
33
34 // 执行绘制
35 function draw() {
36     ctx.clearRect(0, 0, canvas.width, canvas.height);
37     for(let i = 0; i < COUNT; i++) {
38         const type = shapeTypes[Math.floor(Math.random() * shapeTypes.length)];
39         const points = regularShape(Math.random() * canvas.width,
40             Math.random() * canvas.height, 10, type);
41         drawShape(ctx, points);
42     }
43     requestAnimationFrame(draw);
44 }
45
46 draw();
47
```

这个效果实现起来虽然不难, 但性能却不是很好, 因为它在我的 Macbook Pro 电脑上只有不到 30fps 的帧率。那问题出在哪呢? 我们还是要回到代码中。

我们注意到 drawShape 函数里的 for 循环, 它是根据顶点来绘制图形的, 一个点对应一条绘图指令。而在我们绘制的随机图形里, 有 3、4、5、6 边形和 100 边形。对于一个 100 边形来说, 它的顶点数量非常多, 所以 Canvas 需要执行的绘图指令也会非常多, 那绘制很多个 100 边形自然会造成性能问题了。因此, 如何减少绘制 100 边形的绘图指令的数量, 才是我们要优化的重点。具体该怎么做呢?

我们知道, 对于半径为 10 的小图形来说, 正 100 边形已经完全是正圆形了, 所以我们可以用 arc 指令来替代 for 循环。



我们修改 `shapeTypes` 和 `draw` 函数，用 `-1` 代替正 `100` 边形，然后判断 `type` 是否大于 `0`，如果是就用之前的方式绘制正多边形，否则用 `arc` 指令来画圆。这么做了之后，整个效果的帧率就会从 `30fps` 提升到 `40fps`，效果还是比较明显的。

[复制代码](#)

```
1  const shapeTypes = [3, 4, 5, 6, -1];
2  const COUNT = 1000;
3  const TAU = Math.PI * 2;
4
5  function draw() {
6    ctx.clearRect(0, 0, canvas.width, canvas.height);
7    for(let i = 0; i < COUNT; i++) {
8      const type = shapeTypes[Math.floor(Math.random() * shapeTypes.length)];
9      const x = Math.random() * canvas.width;
10     const y = Math.random() * canvas.height;
11     if(type > 0) {
12       // 画正多边形
13       const points = regularShape(x, y, 10, type);
14       drawShape(ctx, points);
15     } else {
16       // 画圆
17       ctx.beginPath();
18       ctx.arc(x, y, 10, 0, TAU);
19       ctx.stroke();
20       ctx.fill();
21     }
22   }
23   requestAnimationFrame(draw);
24 }
```

到这里，你会发现，我们讲的其实是个特例，那在实际工作中，我们是需要针对特例来优化的。我希望我讲完今天的内容你能够做到举一反三。

## 手段二：使用缓存

在上面的方法中，我们优化了绘图指令，让渲染性能有了比较明显的提升。不过，因为这个绘图任务的图形数量和状态都是有限的，我们还有更好的优化方法，那就是**使用缓存**。


因为 Canvas 的性能瓶颈主要在绘图指令方面，如果我们能将图形缓存下来，保存到离屏的 Canvas (offscreen Canvas) 中，然后在绘制的时候作为图像来渲染，那我们就可以将绘制顶点的绘图指令变成直接通过 `drawImage` 指令来绘制图像，而且也不需要 `fill()` 方法来填充图形，这样性能就会有大幅度的提升。

具体的做法，是我们先实现一个创建缓存的函数。代码如下：

 复制代码

```
1 function createCache() {
2   const ret = [];
3   for(let i = 0; i < shapeTypes.length; i++) {
4     // 创建离屏Canvas缓存图形
5     const cacheCanvas = new OffscreenCanvas(20, 20);
6     // 将图形绘制到离屏Canvas对象上
7     const type = shapeTypes[i];
8     const context = cacheCanvas.getContext('2d');
9     context.fillStyle = 'red';
10    context.strokeStyle = 'black';
11    if(type > 0) {
12      const points = regularShape(10, 10, 10, type);
13      drawShape(context, points);
14    } else {
15      context.beginPath();
16      context.arc(10, 10, 10, 0, TAU);
17      context.stroke();
18      context.fill();
19    }
20    ret.push(cacheCanvas);
21  }
22  // 将离屏Canvas数组（缓存对象）返回
23  return ret;
24 }
```

然后，我们一次性创建缓存，直接通过缓存来绘图。

 复制代码

```
1 const shapes = createCache();
2 const COUNT = 1000;
3
4 function draw() {
5   ctx.clearRect(0, 0, canvas.width, canvas.height);
6   for(let i = 0; i < COUNT; i++) {
7     const shape = shapes[Math.floor(Math.random() * shapeTypes.length)];
8     const x = Math.random() * canvas.width;
9     const y = Math.random() * canvas.height;
10    ctx.drawImage(shape, x, y);
11  }
12  requestAnimationFrame(draw);
13 }
```

这样，我们就通过缓存渲染，把原本数量非常多的绘图指令优化成了只有 `drawImage` 的一条指令，让渲染帧率达到了 60fps，从而大大提升了性能。

## 缓存的局限性

不过，虽然使用缓存能够显著降低 Canvas 的性能消耗，但是缓存的使用也有局限性。

首先，因为缓存是通过创建离屏 Canvas 对象实现的，如果我们要绘制的图形状态（指不同形状、颜色等）非常多的话，那将它们都缓存起来，就需要创建大量的离屏 Canvas 对象。这本身对内存消耗就非常大，有可能反而降低了性能。

其次，缓存适用于图形状态本身不变的图形元素，如固定的几何图形，它们每次刷新只需要更新它的 `transform`，这样的图形比较适合用缓存。如果是经常发生状态改变的图形元素，那么缓存就必须一直更新，缓存更新本身也是绘图过程。因此，这种情况下，采用缓存根本起不到减少绘图指令的作用，反而因为增加了一条 `drawImage` 指令产生了更大的开销。

第三，严格上来说，从缓存绘制和直接用绘图指令绘制还是有区别的，尤其是在 `fillText` 渲染文字或者我们绘制一个图形有较大缩放（`scale`）的时候。因为不使用缓存直接绘制的是矢量图，而通过缓存 `drawImage` 绘制出的则是位图，所以缓存绘制的图形，在清晰度上可能不是很好。

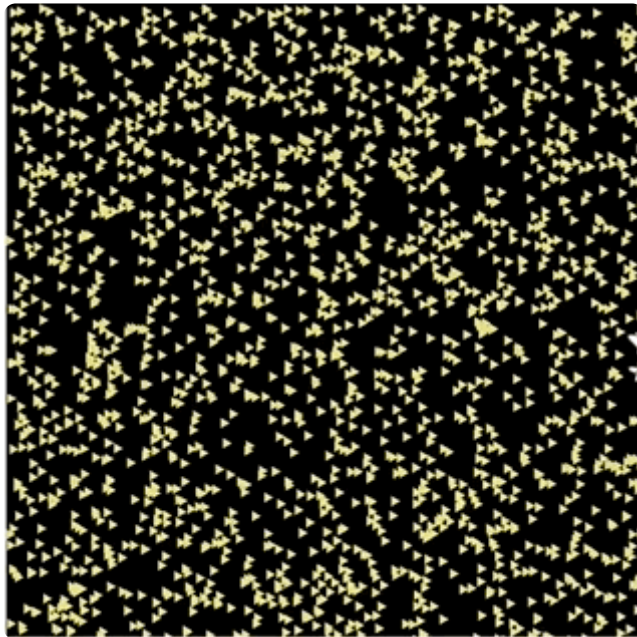
但是总体来说，缓存的应用还是非常多的，我们应该要掌握它的用法，学会在合适的时候运用缓存来提升 Canvas 的渲染性能。

## 手段三：分层渲染

前面两种手段是操作 Canvas 上所有元素来优化性能的，但有的时候，我们要绘制的元素很多，其中大部分元素状态是不变的，只有一小部分有变化。这个时候，我们又该如何进行优化呢？

我们知道，Canvas 是将上一次绘制的内容擦除，然后绘制新的内容来实现状态变化的。利用这一特点，我们就可以将变化的元素和不变的元素进行分层处理。也就是说，我们可以用两个 Canvas 叠在一起，将不变的元素绘制在一个 Canvas 中，变化的元素绘制在另一个 Canvas 中。

我们还是来看一个例子。



假设，我们要实现一个如上图的效果。这个效果的特点是，画面上有一个飞机在运动，运动的物体比较少，而其他静止不动的图形很多（如背景中的上千个三角形）。

在绘制的时候，我们如果将运动的物体和其他物体都绘制在同一个 Canvas 画布中，要改变飞机的运动状态，我们就要重新绘制所有的物体，这会非常浪费性能。因此，更好的做法是我们使用两层画布，一层 Canvas 作为背景，来绘制静态的图形，就是这个例子里的上千个小三角形，而另一层 Canvas 作为前景，用来绘制运动的物体就是运动的飞机。

这样的话，我们只需要一次绘制就能得到背景层 Canvas，并且不管飞机的状态怎么改变，我们都不需要重绘，而前景的飞机可以每一帧重绘，也就大大减少了图形绘制的数量，并且提升了性能。

下面我列出具体的代码，虽然很长但逻辑并不复杂，核心就是用两个 Canvas 元素来分别绘制，你可以看一下。

 复制代码

```
1 function drawRandomTriangle(path, context) {  
2   const {width, height} = context.canvas;  
3   context.save();  
4   context.translate(Math.random() * width, Math.random() * height);  
5   context.fill(path);  
6   context.restore();  
7 }
```



```
8 function drawBackground(context, count = 2000) {
9   context.fillStyle = '#ed7';
10  const d = 'M0,0L0,10L8.66, 5z';
11  const p = new Path2D(d);
12  for(let i = 0; i < count; i++) {
13    drawRandomTriangle(p, context);
14  }
15 }
16
17 function loadImage(src) {
18   const img = new Image();
19   img.crossOrigin = 'anonymous';
20   return new Promise((resolve) => {
21     img.onload = resolve(img);
22     img.src = src;
23   });
24 }
25
26 async function drawForeground(context) {
27   const img = await loadImage('http://p3.qhimg.com/t015b85b72445154fe0.png');
28   const {width, height} = context.canvas;
29   function update(t) {
30     context.clearRect(0, 0, width, height);
31     context.save();
32     context.translate(0, 0.5 * height);
33     const p = (t % 3000) / 3000;
34     const x = width * p;
35     const y = 0.1 * height * Math.sin(3 * Math.PI * p);
36     context.drawImage(img, x, y);
37     context.restore();
38     requestAnimationFrame(update);
39   }
40   update(0);
41 }
42
43 const bgcanvas = document.querySelector('#bg');
44 const fgcanvas = document.querySelector('#fg');
45 drawBackground(bgcanvas.getContext('2d'));
46 drawForeground(fgcanvas.getContext('2d'));
47
```

## 手段四：局部重绘

但是，我们用分层渲染解决性能问题的时候，所绘制的图形必须满足两个条件：一是有大量静态的图形元素不需要重新绘制，二是动态和静态图形元素绘制顺序是固定的，先绘制完静态元素再绘制动态元素。如果元素都有可能运动，或者动态元素和静态元素的绘制顺序是交错的，比如先绘制几个静态元素，再绘制几个动态元素，然后再绘制静态元素，这

样交替进行，那么分层渲染就不好实现了。这时候，我们还有另外一种优化手段，它叫做局部重绘。

**局部重绘**顾名思义，就是不需要清空 Canvas 的全局区域，而是根据运动的元素的范围来清空部分区域。在很大一部分可视化大屏项目中，我们不会让整个屏幕的所有元素都不断改变，而是只有一些固定的区域改变，所以我们直接刷新那部分区域，重绘区域中的元素就可以了。



大屏的动态区与静态区

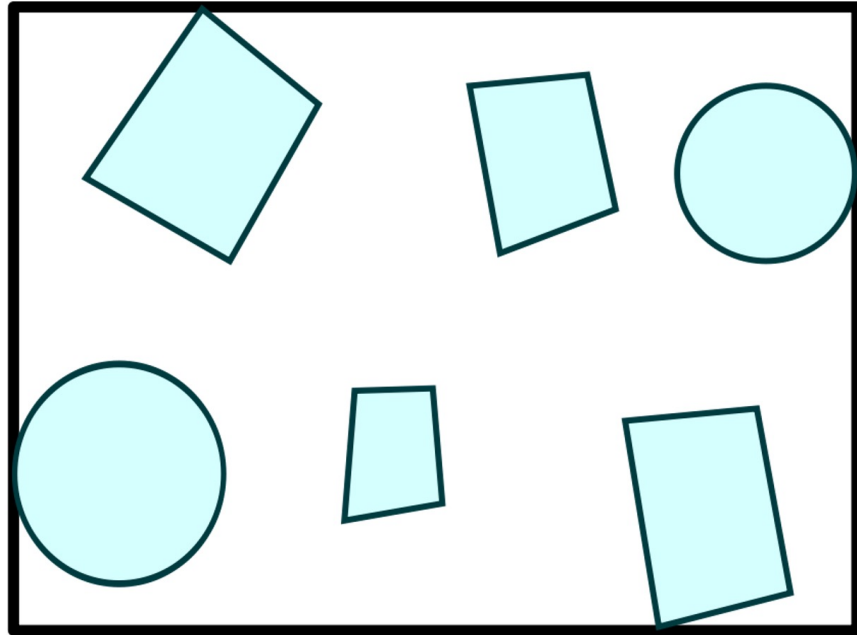
如上图所示，一个可视化大屏只有 2 块动态区域需要不断重绘，那我们用 Canvas 上下文的 **clearRect** 方法控制要刷新的动态区域，只对这些区域进行擦除然后重绘。

要注意的是，动态区重绘的时候，区域内的静态元素也需要跟着重绘。如果有静态元素跨越了动态和静态区域范围，那在重绘时，我们自然不希望破坏了静态区的图形。这时候，我们可以使用 Canvas 上下文的 **clip** 方法，它是一种特殊的绘图指令，可以设定一个绘图区，让图形的绘制限制在这个绘图区内部。这样的话，图形中超过 clip 范围的部分，浏览器就不会把它渲染到 Canvas 上。

这种固定区域的局部重绘使用起来不难，但有时候我们不知道具体的动态区域究竟多大。这个时候，我们可以使用动态计算要重绘区域的技术，它也被称为**脏区检测**。它的基本原理是根据动态元素的**包围盒**，动态算出需要重绘的范围。

那什么是包围盒呢？

我们知道，多边形由顶点构成，包围盒就是指能包含多边形所有顶点，并且与坐标轴平行的最小矩形。



多边形包围盒

在 Canvas 平面直角坐标系下，求包围盒并不复杂，只要分别找到所有顶点坐标中  $x$  的最大、最小值  $xmin$  和  $xmax$ ，以及  $y$  的最大、最小值  $ymin$  和  $ymax$ ，那么包围盒就是矩形  $[(xmin, ymin), (xmin, ymax), (xmax, ymax), (xmax, ymin)]$ 。

对所有的动态元素计算出包围盒，我们就能知道局部刷新的范围了。不过在实际操作的时候，我们经常会遇到各种复杂的细节问题需要解决。因为涉及的细节比较多，我没法全都讲到，所以，如果你遇到了问题，可以看看蚂蚁金服 AntV 团队的 [🔗Canvas 局部渲染优化总结](#) 这篇文章。

## 手段五：优化滤镜

实际上，分层渲染和局部重绘解决的都是图形重绘的问题。那除了重绘，影响渲染效率的还有 Canvas 滤镜。

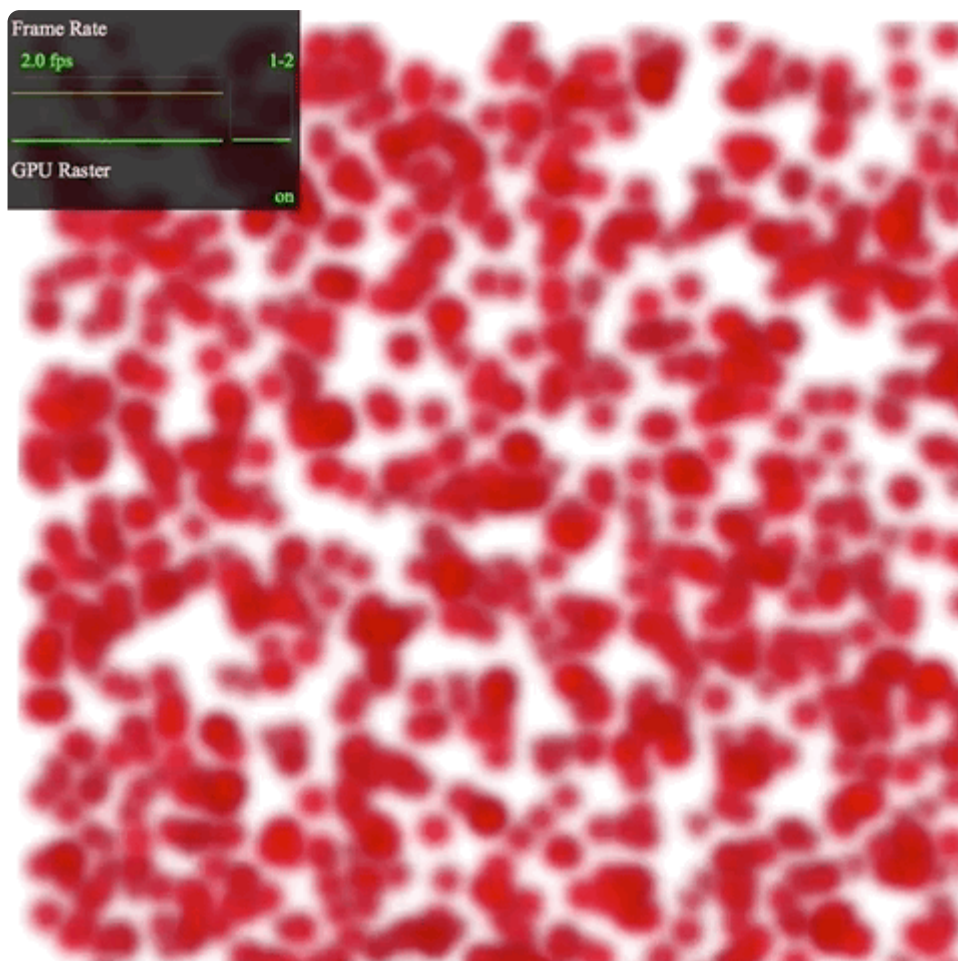
我们知道，滤镜是一种对图形像素进行处理的方法，Canvas 支持许多常用的滤镜。不过 Canvas 渲染滤镜的性能开销比较大。到底有多大呢？我们还是用前面绘制随机图形的例子来体验一下。

这次我们用缓存优化版本的代码，这一版代码的性能最高。在绘制前，我们给 Canvas 设置一个 blur 滤镜。代码如下：

```
1 ctx.filter = 'blur(5px)';
```

[复制代码](#)

这样呢，我们让 Canvas 绘制出来的图形有了模糊的效果。但是这么设置了之后，你会发现原本 60fps 的帧率直接掉到 2fps，画面看上去一顿一顿的，卡得惨不忍睹。这就是因为滤镜对渲染性能的开销实在太大了。



那这种情况下，其实我们也有优化手段。针对这个场，我们实际上是对 Canvas 应用一个全局的 blur 滤镜，把绘制的所有元素都变得模糊，所以，我们完全没必要对每个元素应用滤镜，而是可以采用类似后期处理通道的做法，先将图形以不使用滤镜的方式绘制到一个

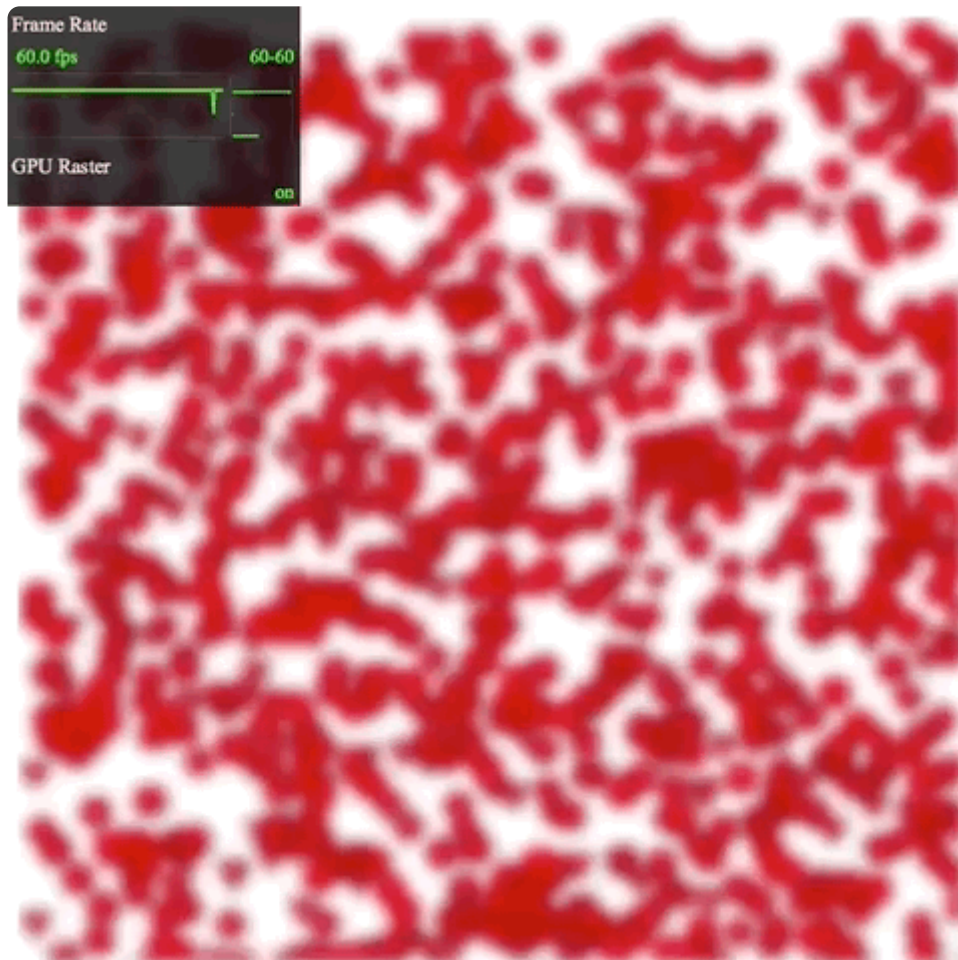
离屏的 Canvas 上，然后直接将这个离屏 Canvas 以图片方式绘制到要显示的画布上，在这次绘制的时候采用滤镜。这样，我们就把大量滤镜绘制的过程缩减为对一张图片使用一次滤镜了。大大减少了处理滤镜的次数之后，效果立竿见影，帧率立即回到了 60fps。

那么具体实现的代码和效果我也列出来，你可以看一下。

[复制代码](#)

```
1 ctx.filter = 'blur(5px)';
2
3 // 创建离屏的 Canvas
4 const ofc = new OffscreenCanvas(canvas.width, canvas.height);
5 const octx = ofc.getContext('2d');
6 function draw() {
7   ctx.clearRect(0, 0, canvas.width, canvas.height);
8   octx.clearRect(0, 0, canvas.width, canvas.height);
9   // 将图形不应用滤镜，绘制到离屏Canvas上
10  for(let i = 0; i < COUNT; i++) {
11    const shape = shapes[Math.floor(Math.random() * shapeTypes.length)];
12    const x = Math.random() * canvas.width;
13    const y = Math.random() * canvas.height;
14    octx.drawImage(shape, x, y);
15  }
16  // 再将离屏Canvas图像绘制到画布上，这一次绘制采用了滤镜
17  ctx.drawImage(ofc, 0, 0);
18  requestAnimationFrame(draw);
19 }
20
21 draw();
```





当然这种优化滤镜的方式，只有当我们要对画布上绘制的所有图形，都采用同一种滤镜的时候才有效。不过，如果有部分图形采用相同的滤镜，而且它们是连续绘制的，我们也可以采用类似的办法，把这部分图形绘制到离屏 Canvas 上，之后再将图像应用滤镜并绘制回画布。这样也能够减少滤镜的处理次数，明显提升性能。总之，想要达到比较好的性能，我们要记住一个原则，尽量合并图形应用相同滤镜的过程。

## 手段六：多线程渲染

到这里，我们说完了几种提升渲染性能的常见手段。不过，影响用户体验的不仅仅是渲染性能，有时候，我们还要对绘制的内容进行交互，而如果渲染过程消耗了大量的时间，它也可能阻塞其他的操作，比如对事件的响应。

遇到这种问题的时候，以前我们会比较头疼，甚至不得不降低渲染性能，以减少 CPU 资源占用，从而让交互行为不被阻塞。不过现在，浏览器支持的 Canvas 可以在 WebWorker 中以单独的线程来渲染，这样就可以避免对主线程的阻塞，也不会影响用户交互行为了。

那么具体怎么才能在 Worker 中绘制呢？其实也很简单。我们在浏览器主线程中创建 Worker，然后将 Canvas 对象通过 `transferControlToOffscreen` 转成离屏 Canvas 对象


发送给 Worker 线程去处理。

 复制代码

```
1 const canvas = document.querySelector('canvas');
2
3 const worker = new Worker('./random_shapes_worker.js');
4 const ofc = canvas.transferControlToOffscreen();
5 worker.postMessage({
6   canvas: ofc,
7   type: 'init',
8 }, [ofc]);
```

这样，从使用上来说，无论在 Worker 线程中还是在主线程中操作都没有太大的区别，还能不阻塞浏览器主线程的任何操作。

我这里列出一部分核心代码，完整的代码我放在 GitHub 仓库里，你可以试着运行一下，看看效果。

 复制代码

```
1 function draw(ctx, shapes) {
2   const canvas = ctx.canvas;
3   ctx.clearRect(0, 0, canvas.width, canvas.height);
4   for(let i = 0; i < COUNT; i++) {
5     const shape = shapes[Math.floor(Math.random() * shapeTypes.length)];
6     const x = Math.random() * canvas.width;
7     const y = Math.random() * canvas.height;
8     ctx.drawImage(shape, x, y);
9   }
10  requestAnimationFrame(draw.bind(null, ctx, shapes));
11 }
12
13 self.addEventListener('message', (evt) => {
14   if(evt.data.type === 'init') {
15     const canvas = evt.data.canvas;
16     if(canvas) {
17       const ctx = canvas.getContext('2d');
18       const shapes = createCache();
19       draw(ctx, shapes);
20     }
21   }
22 });
```

## 要点总结

这节课我们讲了 Canvas 性能优化的 6 种手段，其中前 5 种是针对渲染效率进行优化，分别是优化 Canvas 指令、使用缓存、分层渲染、局部重绘，以及针对滤镜的优化。最后一种是通过多线程来优化计算的性能，让计算过程能够并行执行不会阻塞浏览器的 UI。下面，我再带你一起梳理一下性能优化的原则。

首先，我们在绘制图形时，用越简单的绘图指令来绘制，渲染的效率就越高。所以，我们要想办法减少 Canvas 绘图指令的数量，比如，用 arc 指令画圆来代替绘制边数很多的正多边形。

然后，当我们大批量绘制有限的几种形状的图形时，可以采用缓存将图形一次绘制后保存在离屏的 Canvas 中，下一次绘制的时候，我们直接绘制缓存的图片来取代原始的绘图指令，也能大大提升性能。

可如果我们绘制的元素中只有一部分元素发生改变，我们就可以采用分层渲染，将变化的元素绘制在一个图层，剩下的元素绘制在另一个图层。这样每次只需要重新绘制变化元素所在的图层，大大减少绘制的图形数，从而显著提升了性能。

还有一种情况是，如果 Canvas 只有部分区域发生变化，那我们只需要刷新局部区域，不需要刷新整个 Canvas，这样能显著降低消耗、提升性能。

还要注意的，一些 Canvas 滤镜渲染起来非常耗费性能，所以我们可以对滤镜进行合并，让多个元素只应用一次滤镜，从而减少滤镜对性能的消耗。

最后，除了优化渲染性能外，我们还可以通过 WebWork 以多线程的手段优化计算性能，以达到渲染不阻塞 UI 操作的目的。

## 小试牛刀

学会了使用多种优化手段之后，我们来尝试实现一个粒子效果吧！

具体效果是，我们要让小三角形以不同的角度和速度，由画布中心点向四周运动，同时小三角形自身也以随机的角速度旋转。

你可以尝试用两种方式来实现这个效果，分别是使用性能优化和不使用性能优化。在这两种情况下，你的电脑最多能支持同时绘制多少个小三角形？

我们今天学的这 6 种性能优化手段，对你的工作是不是很有帮助？那不妨就把这节课分享出去吧！我们下节课再见！

---

## 源码

[🔗](#) 课程中详细示例代码 GitHub 仓库

## 推荐阅读

[1] [🔗](#) AntV Canvas 局部渲染总结

[2] [🔗](#) Speed up Your Canvas Operations with a Web Worker, WebWorker 和 OffscreenCanvas 使用参考文档

提建议

## 更多课程推荐

# 程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 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 28 | Canvas、SVG与WebGL在性能上的优势与劣势

下一篇 30 | 怎么给WebGL绘制加速?

## 精选留言 (1)

写留言



Mingzhang

2020-08-28

在屏幕上显示主 Canvas，利用 `transferControlToOffscreen` 将绘制交由 `offScreen canvas` 来绘制，而主 Canvas 负责监测鼠标在其上的 `move` 事件，然后将 `event` 的坐标 `postMessage` 给 Web Worker。请问月影这种方法可行吗？

展开 ∨

💬 1

👍 2