Canvas 绘制效率不低

Canvas 没有 dom 操作，只是简单的 2D 绘制，所以效率不低，Chrome 浏览器下，每秒可绘制五万个基本图形元素（圆形，矩形或者线条），如果有阴影效果会慢很多，总的来说上万元素的绘制还是很轻松的。

适合简单应用

因为简单，做一些像素处理，2D 绘制，小游戏啥的还是很方便的，国际上有 javascript1k 作品大赛，用 1024 字节的 js 代码，实现丰富的效果，基本上都用到 canvas，所以在轻量小巧方面很有优势。

画布绘图的环境通过 translate(),scale(),rotate(), setTransform()和 transform()来改变，它们会对画布的变换矩阵产生影响

参数说明

函数	方法	描述
translate	dx,dx	转换的量的 X 和 Y 大小
scale	sx,sy	水平和垂直的缩放因子
rotate	angle	旋转的量，用弧度表示。正值表示顺时针方向旋转，负值表示逆时针方向旋转。
setTransform	a,b,c,d,e,f	水平缩放,水平倾斜(与旋转有关),垂直倾斜(与旋转有关，-水平倾斜),垂直缩放,水平移动,垂直移动

transform	a,b,c,d,e,f	水平缩放,水平倾斜,垂直倾斜,垂直缩放,水平移动,垂直移动

translate() 方法为画布的变换矩阵添加水平的和垂直的偏移。参数 dx 和 dy 添加给后续定义路径中的所有点。

scale() 方法为画布的当前变换矩阵添加一个缩放变换。缩放通过独立的水平和垂直缩放因子来完成。例如，传递一个值 2.0 和 0.5 将会导致绘图路径宽度变为原来的两倍，而高度变为原来的 1/2。指定一个负的 sx 值，会导致 X 坐标沿 Y 轴对折，而指定一个负的 sy 会导致 Y 坐标沿着 X 轴对折。

rotate() 方法通过指定一个角度，改变了画布坐标和 Web 浏览器中的 <Canvas> 元素的像素之间的映射，使得任意后续绘图在画布中都显示为旋转的。它并没有旋转 <Canvas> 元素本身。注意，这个角度是用弧度指定的。

setTransform() 会将当前的变换矩阵重置为单位矩阵，然后构建新的矩阵。

transform() 添加一个新的变换矩阵，再次绘制矩形，调用 transform() 时，它都会在前一个变换矩阵上构建。

具体使用范例

我们在画布上绘制一个矩形。

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="300" height="150" style="border:1px solid #d3d3d3;">
Your browser does not support the HTML5 canvas tag.
</canvas>
<script>
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");
ctx.fillStyle="yellow";
```

```
ctx.fillRect(0,0,250,100)
</script>
</body>
</html>
```

然后画出的图形如下图所示：绘制范围就是从坐标(0,0)开始画一个宽高分别为250,100 的矩形。

下面对这个矩形进行平移，缩放，旋转等操作。

1 平移 translate

如果往中间平移的话，我们可以改变它的坐标值例如ctx.fillRect(25,25,250,100)，就将矩形起始位置移动到了(25,25)。绘制出的效果如下图：

另外还有一种方法，我们可以通过 context 的 translate 方法移动画布，来达到同样的视觉效果。未变换前画布的原点在左上角，使用 translate 方法后相当于整个坐标系在平移。

注：在操作画布变换矩阵时，最好在变换前使用 save 方法保存记录画布当前状态，完成绘制后可以使用 restore 方法恢复变换前的矩阵状态。

```
ctx.save();
ctx.translate(25,25);
ctx.fillStyle="yellow";
ctx.fillRect(0,0,250,100)
ctx.restore();
```

2 缩放 scale

缩放矩形当然可以通过调整宽高的方式这里不再举例说明。

这里主要介绍通过 context 的 scale 方法缩放画布的方法，通过以下代码就可以将黄色矩形所缩小到原来的一半（如下图）。

```
ctx.save();
ctx.scale(0.5,0.5);
ctx.fillStyle="yellow";
ctx.fillRect(25,25,250,100);
ctx.restore();
```


3 旋转 rotate

旋转图中矩形，就需要把画布矩阵状态通过 rotate 方法变换下了。

rotate(Math.PI/6)就是将画布顺时针旋转 30 度，画出的矩形如下图。

```
ctx.save();
ctx.rotate(Math.PI/6);
ctx.fillStyle="yellow";
ctx.fillRect(25,25,250,100)
```

ctx.restore();

4 确定中心点 transform

通过图可以看出上面的缩放和旋转都是以画布左上角为中心进行的，如果我们需要以画布中心点为中心进行变换，需要在 rotate 和 scale 方法调用前去修正画布的坐标原点位置到画布占位的中心位置 translate(width/2,height/2)，然后进行缩放变换，之后再用 translate(-width/2,-height/2)修正坐标系。如下代码：

```
ctx.save();
ctx.translate(width/2,height/2);//将画布坐标系原点移至中心
ctx.scale(0.5,0.5);//如果是缩放，这里是缩放代码
ctx.translate(-width/2,-height/2);//修正画布坐标系
ctx.fillStyle="yellow";
ctx.fillRect(25,25,250,100)
ctx.restore();
```

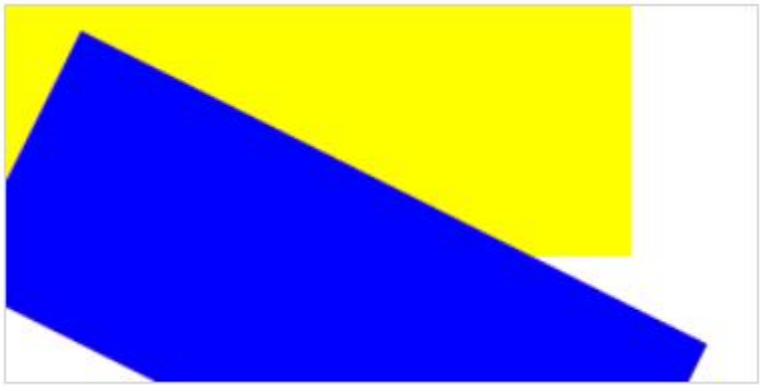
效果如下图：

5 设置矩阵 setTransform

方法 setTransform 的六个参数上面已经提到，其实可以理解为缩放、旋转、移动的复合方法。当您调用 setTransform() 时，它都会重置前一个变换矩阵然后构建新的矩阵，因此在下面的例子中，不会显示红色矩形，因为它在蓝色矩形下面。

```
var c=document.getElementById("myCanvas");
var ctx=c.getContext("2d");

ctx.fillStyle="yellow";
ctx.fillRect(0,0,250,100)
```

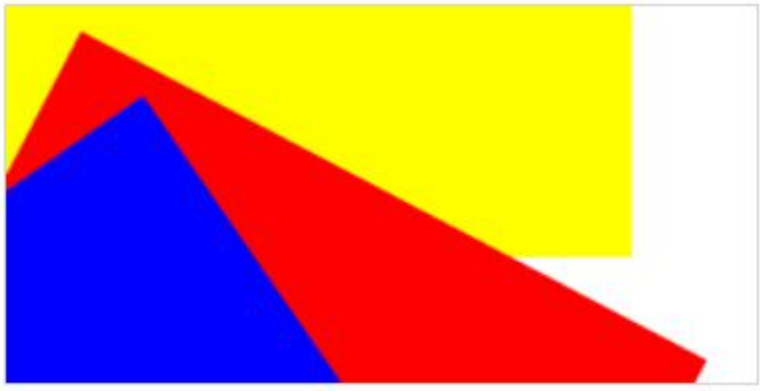
```
ctx.setTransform(1,Math.PI/6,-Math.PI/6,1,30,10);// 两个 1 代表画布进行缩放，
Math.PI/6 表示顺时针旋转 30 度，(30,10)表示平移
ctx.fillStyle="red";
ctx.fillRect(0,0,250,100);

ctx.setTransform(1,Math.PI/6,-Math.PI/6,1,30,10);
ctx.fillStyle="blue";
ctx.fillRect(0,0,250,100);
```



和 setTransform 类似的还有一个方法 transform()，它们都有六个参数。**但是不同的是每次调用 transform() 时，它都会在前一个变换矩阵上构建。**如下代码中先绘制的红色矩形和上图的位置是一致的，画完红色矩形后，再次调用 transform(1,Math.PI/6,-Math.PI/6,1,30,10)改变画布，这次变换是在当前画布状态上变换的所以画的蓝色矩形不会和红色的重叠。如下图看上去，红色和黄色的相对位置 与 蓝色和红色的相对位置 是一样的。

```
var c=document.getElementById("myCanvas");
```

```
var ctx=c.getContext("2d");


ctx.fillStyle="yellow";

ctx.fillRect(0,0,250,100)


ctx.transform(1,Math.PI/6,-Math.PI/6,1,30,10);

ctx.fillStyle="red";

ctx.fillRect(0,0,250,100);


ctx.transform(1,Math.PI/6,-Math.PI/6,1,30,10);

ctx.fillStyle="blue";

ctx.fillRect(0,0,250,100);
```



canvas 的 save 与 restore 方法的作用

save：用来保存 Canvas 的状态。save 之后，可以调用 Canvas 的平移、放缩、旋转、错切、裁剪等操作。
restore：用来恢复 Canvas 之前保存的状态。防止 save 后对 Canvas 执行的操作对后续的绘制有影响。

对 canvas 中特定元素的旋转平移等操作实际上是对整个画布进行了操作，所以如果不对 canvas 进行 save 以及 restore，那么每一次绘图都会在上一次的基础上进行操作，最后导致错位。比如说你相对于起始点每次 30 度递增旋转，30，60，90.如果不使用 save 以及 restore 就会变成 30, 90, 150，每一次在前一次基础上进行了旋转。

save 是入栈，restore 是出栈。

动画的基本步骤

在 web 应用中，实现动画的方式有很多种。

1. js 使用 setTimeout 实现
2. css 使用 transition 和 animation 实现
3. html5 使用 canvas 实现

   除此之外就是我们今天要介绍的 requestAnimationFrame，从字面意思翻译来看，

   就是请求动画帧。

setTimeout（setInterval）

动画原理

我们在浏览器中看到的图像是在以每秒 60 次的频率刷新的，由于刷新频率很高，因此你感觉不到它在刷新。而**动画本质就是要让人眼看到图像被刷新而引起变化的视觉效果，这个变化要以连贯的、平滑的方式进行过渡。** 那怎么样才能做到这种效果呢？

刷新频率为 60Hz 的屏幕每 16.7ms 刷新一次，我们在屏幕每次刷新前，将图像的位置向左移动一个像素，即 1px。这样一来，屏幕每次刷出来的图像位置都比前一个要差 1px，因此你会看到图像在移动；由于我们人眼的视觉停留效应，当前位置的图像停留在大脑的印象还没消失，紧接着图像又被移到了下一个位置，因此你才会看到图像在流畅的移动，这就是视觉效果上形成的动画。

setTimeout（setInterval）的缺点

理解了上面的概念以后，我们不难发现，setTimeout 其实就是通过设置一个间隔时间来不断的改变图像的位置，从而达到动画效果的。但我们会发现，利用 seTimeout 实现的动画在某些低端机上会出现卡顿、抖动的现象。 这种现象的产生有两个原因：

setTimeout 的执行时间并不是确定的。在 Javascript 中， setTimeout 任务被放进了异步队列中，只有当主线程上的任务执行完以后，才会去检查该队列里的任务是否需要开始执行，因此 setTimeout 的实际执行时间一般要比其设定的时间晚一些。

刷新频率受屏幕分辨率和屏幕尺寸的影响，因此不同设备的屏幕刷新频率可能会不同，而 setTimeout 只能设置一个固定的时间间隔，这个时间不一定和屏幕的刷新时间相同。

以上两种情况都会导致 setTimeout 的执行步调和屏幕的刷新步调不一致，从而引起丢帧现象。

所以，使用 window.setInterval()或者 window.setTimeout()制作的动画，有如下缺点：

1.他们都是通用的方法，并不是专为绘制动画而用

2.即使向其专递以毫秒为单位的参数值，它们也达不到毫秒级的准确性

3.没有对调用动画循环的机制优化

4.不考虑绘制动画的最佳时机，而只会一味地以某个大致的时间间隔调用动画循环。

requestAnimationFrame

你可以通过以下的步骤来画出一帧:

- **清空 canvas**
  除非接下来要画的内容会完全充满 canvas （例如背景图），否则你需要清空所有。最简单的做法就是用 clearRect 方法。
- **保存 canvas 状态 save()**
  如果你要改变一些会改变 canvas 状态的设置（样式，变形之类的），又要在每画一帧之时都是原始状态的话，你需要先保存一下。
- **绘制动画图形（animated shapes)**
  这一步才是重绘动画帧。
- **恢复 canvas 状态**
  如果已经保存了 canvas 的状态，可以先恢复它，然后重绘下一帧

window.requestAnimationFrame() 告诉浏览器——你希望执行一个动画，并且要求浏览器在下次重绘之前调用指定的回调函数更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

```
window.requestAnimationFrame(callback);
```

简单来说，requestAnimationFrame 方法用于通知浏览器重采样动画。

当 requestAnimationFrame(callback)被调用时不会执行 callback，而是会将元组<handle,callback>插入到动画帧请求回调函数列表末尾（其中元组的 callback 就是传入 requestAnimationFrame 的回调函数），并且返回 handle 值，该值为浏览器定义的、大于 0 的整数，唯一标识了该回调函数在列表中位置。

每个回调函数都有一个布尔标识 cancelled，该标识初始值为 false，并且对外不可见。

浏览器在执行“采样所有动画”的任务时会遍历动画帧请求回调函数列表，判断每个元组的 callback 的 cancelled，如果为 false，则执行 callback。

cancelAnimationFrame 方法用于取消先前安排的一个动画帧更新的请求。

当调用 cancelAnimationFrame(handle)时，浏览器会设置该 handle 指向的回调函数的 cancelled 为 true。

无论该回调函数是否在动画帧请求回调函数列表中，它的 cancelled 都会被设置为 true。

如果该 handle 没有指向任何回调函数，则调用 cancelAnimationFrame 不会发生任何事情。

当页面可见并且动画帧请求回调函数列表不为空时，浏览器会定期地加入一个“采样所有动画”的任务到 UI 线程的队列中。

这个 API 的调用很简单，如下所示：

```
var progress = 0;//回调函数
function render() {
    progress += 1; //修改图像的位置
    if (progress < 100) {
    //在动画没有结束前，递归渲染
    window.requestAnimationFrame(render);
    }
```

```
}
//第一帧渲染
window.requestAnimationFrame(render);
```

cancelAnimationFrame 的使用方法也很简单

```
function a(time) {
        console.log("animation");
        id = window.requestAnimationFrame(a);
    }
    a();
window.cancelAnimationFrame(id);
```

与 setTimeout 相比，requestAnimationFrame 最大的优势是**由系统来决定回调函数的执行时机。**具体一点讲，如果屏幕刷新率是 60Hz,那么回调函数就每 16.7ms 被执行一次，如果刷新率是 75Hz，那么这个时间间隔就变成了 1000/75=13.3ms，换句话说就是，requestAnimationFrame 的步伐跟着系统的刷新步伐走。**它能保证回调函数在屏幕每一次的刷新间隔中只被执行一次**，这样就不会引起丢帧现象，也不会导致动画出现卡顿的问题。

requestAnimationFrame 还有以下两个优势

- **CPU 节能**：使用 setTimeout 实现的动画，当页面被隐藏或最小化时，setTimeout 仍然在后台执行动画任务，由于此时页面处于不可见或不可用状态，刷新动画是没有意义的，完全是浪费 CPU 资源。而 requestAnimationFrame 则完全不同，当页面处理未

激活的状态下，该页面的屏幕刷新任务也会被系统暂停，因此跟着系统步伐走的 requestAnimationFrame 也会停止渲染，当页面被激活时，动画就从上次停留的地方继续执行，有效节省了 CPU 开销。

- **函数节流**：在高频率事件(resize,scroll 等)中，为了防止在一个刷新间隔内发生多次函数执行，使用 requestAnimationFrame 可保证每个刷新间隔内，函数只被执行一次，这样既能保证流畅性，也能更好的节省函数执行的开销。一个刷新间隔内函数执行多次时没有意义的，因为显示器每 16.7ms 刷新一次，多次绘制并不会在屏幕上体现出来。

浏览器兼容性

由于 requestAnimationFrame 目前还存在兼容性问题，而且不同的浏览器还需要带不同的前缀。因此需要通过优雅降级的方式对 requestAnimationFrame 进行封装，优先使用高级特性，然后再根据不同浏览器的情况进行回退，直止只能使用 setTimeout 的情况。

所以，如果想要简单的兼容，可以这样子：

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

```
})();
```

简单的应用

现在分别使用 setInterval、setTimeout 和 requestAnimationFrame 这三个方法制作一个简单的进制度效果

setInterval

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    clearInterval(timer);
    myDiv.style.width = '0';
    timer = setInterval(function(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
            myDiv.innerHTML =       parseInt(myDiv.style.width)/5 + '%';
        }else{
            clearInterval(timer);
        }
```

```
    },16);
}
</script>
```

setTimeout

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    clearTimeout(timer);
    myDiv.style.width = '0';
    timer = setTimeout(function fn(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
            myDiv.innerHTML =       parseInt(myDiv.style.width)/5 + '%';
            timer = setTimeout(fn,16);
        }else{
            clearTimeout(timer);
        }
    },16);
```

```
}
</script>
```

requestAnimationFrame

```
<div id="myDiv" style="width: 0;height: 20px;line-height: 20px;">0%</div>
<button id="btn">run</button>
<script>
var timer;
btn.onclick = function(){
    myDiv.style.width = '0';
    cancelAnimationFrame(timer);
    timer = requestAnimationFrame(function fn(){
        if(parseInt(myDiv.style.width) < 500){
            myDiv.style.width = parseInt(myDiv.style.width) + 5 + 'px';
            myDiv.innerHTML =       parseInt(myDiv.style.width)/5 + '%';
            timer = requestAnimationFrame(fn);
        }else{
            cancelAnimationFrame(timer);
        }
    });
}
```

</script>