



下载APP



12 | 如何使用滤镜函数实现美颜效果？

2020-07-20 月影

跟月影学可视化

[进入课程 >](#)**讲述：月影**

时长 19:28 大小 17.84M



你好，我是月影。

通过前面的课程，我们初步了解了浏览器的图形系统，也学会了使用基本的数学和几何方法来生成和处理图像，还能用简单的图形组合来构成复杂的图案。从这一节课开始，我们进入一个新的模块，开始学习像素处理。

在可视化领域里，我们常常需要处理大规模的数据，比如，需要呈现数万甚至数十万条信息在空间中的分布情况。如果我们用几何绘制的方式将这些信息一一绘制出来，性能可能就会很差。



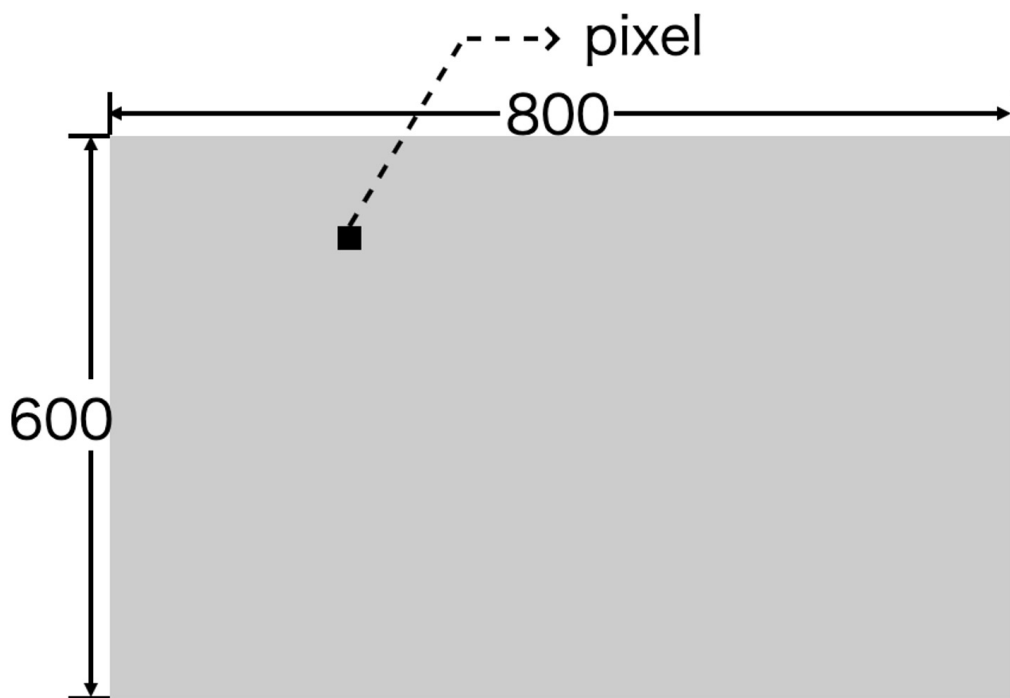
这时，我们就可以将这些数据简化为像素点进行处理。这种处理图像的新思路就叫做**像素化**。在可视化应用中，图片像素化处理是一个很重要手段，它能够在我们将原始数据信息

转换成图形后，进一步处理图形的细节，突出我们想要表达的信息，还能让视觉呈现更有冲击力。

因为像素化的内容比较复杂，能做的事情也非常多，所以我们会用五节课的时间来讨论。今天是第一节课，我们先来看看图片像素化的基本思路和方法，体会如何用像素化来处理照片，从而达到“美颜”的效果。

如何理解像素化？

首先，我们来理解两个基础的概念。第一个是像素化。所谓像素化，就是把一个图像看成是由一组像素点组合而成的。每个像素点负责描述图像上的一个点，并且带有这个点的基本绘图信息。那对于一张 800 像素宽、600 像素高的图片来说，整张图一共就有 48 万个像素点。



这么多的像素点是怎么存储的呢？Canvas2D 以 4 个通道来存放每个像素点的颜色信息，每个通道是 8 个比特位，也就是 0~255 的十进制数值，4 个通道对应 RGBA 颜色的四个值。后面，我们会用 RGBA 通道来分别代表它们。

知道了什么是像素化，那像素处理又是怎么回事呢？像素处理实际上就是我们为了达到特定的视觉效果，用程序来处理图像上每个像素点。像素处理的应用非常广泛，能实现的

效果也非常多。下面，我会列举几个常用的效果，希望你通过它们能理解像素处理的一般方法和思路。

应用一：实现灰度化图片

在可视化中，当我们要给用户强调某些信息的时候，一般会将我们不想强调的信息区域置成灰度状态。这样，用户就能够快速抓住我们想要表达的重点了。这个过程就是**灰度化图片**。简单来说就是将一张彩色图片变为灰白色图片。具体的实现思路是，我们先将该图片的每个像素点的 R、G、B 通道的值进行加权平均，然后将这个值作为每个像素点新的 R、G、B 通道值，具体公式如下：

$$\begin{cases} V = aR + bG + cB \\ R' = G' = B' = V \end{cases}$$

其中 R、G、B 是原图片中的 R、G、B 通道的色值，V 是加权平均色值，a、b、c 是加权系数，满足 $(a + b + c) = 1$ 。

好了，灰度化的原理你已经知道了，下面我们通过一个具体的例子来演示一下实际操作的过程。首先，我们写一段简单的 JavaScript 代码，通过这段代码把一张图片给加载并绘制到 Canvas 上，代码如下：

复制代码

```
1 <canvas id="paper" width="0" height="0"></canvas>
2 <script>
3   function loadImage(src) {
4     const img = new Image();
5     img.crossOrigin = 'anonymous';
6     return new Promise((resolve) => {
7       img.onload = () => {
8         resolve(img);
9       };
10      img.src = src;
11    });
12  }
13
14  const canvas = document.getElementById('paper');
15  const context = canvas.getContext('2d');
16
```

```
17 (async function () {  
18   // 异步加载图片  
19   const img = await loadImage('https://p2.ssl.qhimg.com/d/inn/4b7e384c55dc  
20  
21   const {width, height} = img;  
22   // 将图片绘制到 canvas  
23   canvas.width = width;  
24   canvas.height = height;  
25   context.drawImage(img, 0, 0);  
26 }());  
27
```

这段代码开始创建一个 image 元素，然后通过 image 元素的 src 属性异步加载图片，加载完成后，通过 canvas 的 2d 上下文对象的 drawImage 方法，将图片元素绘制到 canvas 上。这张图片如下图所示：



接下来，我们要获取每个像素点的 R、G、B 值。具体的操作就是通过 canvas 的 2d 上下文，获取图片剪裁区的数据 imgData。那什么是 imgData 呢？imgData 是我们在像素处理中经常用到的对象，它是一个 ImageData 对象，它有 3 个属性，分别是 width、height 和 data。其中 width 表示剪裁区的宽度属性，height 表示剪裁区的高度属性，data 用来存储图片的全部像素信息。

宽、高属性我就不用多说了，我来重点说说 data 是怎么保存图片全部像素信息的。首先，图片的全部像素信息会以**类型数组**（Uint8ClampedArray）的形式保存在 ImageData 对

象的 data 属性里，而类型数组的每 4 个元素组成一个像素的信息，这四个元素依次表示该像素的 RGBA 四通道的值，所以它的数据结构如下：

[复制代码](#)

```
1 data[0] // 第1行第1列的红色通道值
2 data[1] // 第1行第1列的绿色通道值
3 data[2] // 第1行第1列的蓝色通道值
4 data[3] // 第1行第1列的Alpha通道值
5 data[4] // 第1行第2列的红色通道值
6 data[5] // 第1行第2列的绿色通道值
7 ...
```

结合这个结构，我们可以得出 data 属性的类型数组的总长度：**width * height * 4**。这是因为图片一共是 **width * height** 个像素点，每个像素点有 4 个通道，所以总长度是像素点的 4 倍。

知道了数组长度以后，接着，我们就可以遍历 data 数组，读取每个像素的 RGBA 四通道的值，将原本的 RGBA 值都用一个加权平均值 **$0.2126 r + 0.7152 g + 0.0722 * b$** 替代了，代码如下：

[复制代码](#)

```
1 for(let i = 0; i < width * height * 4; i += 4) {
2   const r = data[i],
3     g = data[i + 1],
4     b = data[i + 2],
5     a = data[i + 3];
6   // 对RGB通道进行加权平均
7   const v = 0.2126 * r + 0.7152 * g + 0.0722 * b;
8   data[i] = v;
9   data[i + 1] = v;
10  data[i + 2] = v;
11  data[i + 3] = a;
12 }
```

这里你可能会觉得奇怪，我们为什么用 **0.2126**、**0.7152** 和 **0.0722** 这三个权重，而不是都用算术平均值 **1/3** 呢？这是因为，人的视觉对 R、G、B 三色通道的敏感度是不一样的，对绿色敏感度高，所以加权值高，对蓝色敏感度低，所以加权值低。

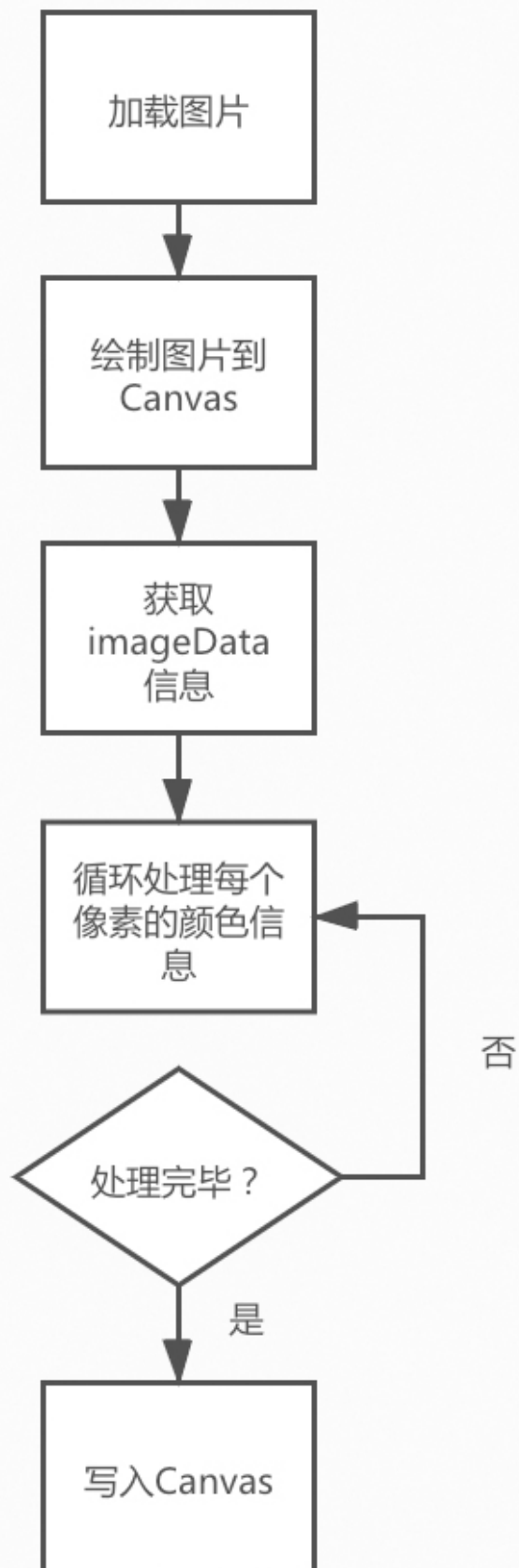
最后，我们将处理好的数据写回到 Canvas 中去。这样，我们就得到了一张经过灰度化后的图片。



灰度化后的图片

灰度化图片的过程非常简单，我们一起来总结一下。首先，我们加载一张图片将它绘制到 canvas，接着我们通过 `getImageData` 获取 `imageData` 信息，再通过 `imageData.data` 遍历图像上的所有像素点，对每个像素点的 RGBA 值进行加权平均处理，然后将处理好的信息回写到 canvas 中去。

我把灰度化图片的过程总结了一张流程图，你也可以参考它来理解。



重构代码以扩展其他效果

实际上，灰度化只是像素颜色处理中一个最简单的应用，除此以外，我们还可以对像素颜色做其他变换，比如增强或减弱某个通道的色值，改变颜色的亮度、对比度、饱和度、色

相等等。

那为了方便讲解，也为了更好地复用代码实现其他的功能，我先重构一下上面的代码，将我们最关注的**循环处理每个像素的颜色信息**这一步单独剥离出来，再把其他步骤都**分解并抽象成通用的模块**以便于实现其他效果的时候引入。

重构代码的实现思路是，先创建一个 **lib/utls.js** 文件，然后把加载图片的函数 **loadImage**，获取 imageData 对象的函数 **getImageData**，以及遍历 imageData 中的类型数组的函数 **traverse**，都添加到 **lib/utls.js** 文件中。代码如下：

复制代码

```
1 // lib/utls.js
2
3 // 异步加载图片
4 export function loadImage(src) {
5   const img = new Image();
6   img.crossOrigin = 'anonymous';
7   return new Promise((resolve) => {
8     img.onload = () => {
9       resolve(img);
10    };
11    img.src = src;
12  });
13 }
14
15 const imageDataContext = new WeakMap();
16 // 获得图片的 imageData 数据
17 export function getImageData(img, rect = [0, 0, img.width, img.height]) {
18   let context;
19   if(imageDataContext.has(img)) context = imageDataContext.get(img);
20   else {
21     const canvas = new OffscreenCanvas(img.width, img.height);
22     context = canvas.getContext('2d');
23     context.drawImage(img, 0, 0);
24     imageDataContext.set(img, context);
25   }
26   return context.getImageData(...rect);
27 }
28
29 // 循环遍历 imageData 数据
30 export function traverse(imageData, pass) {
31   const {width, height, data} = imageData;
32   for(let i = 0; i < width * height * 4; i += 4) {
33     const [r, g, b, a] = pass({
34       r: data[i] / 255,
35       g: data[i + 1] / 255,
```



```
36     b: data[i + 2] / 255,
37     a: data[i + 3] / 255,
38     index: i,
39     width,
40     height,
41     x: ((i / 4) % width) / width,
42     y: Math.floor(i / 4 / width) / height});
43     data.set([r, g, b, a].map(v => Math.round(v * 255)), i);
44 }
45 return imageData;
46
```

我们这样做了之后，像素处理的应用代码就可以得到简化。简化后的代码如下：

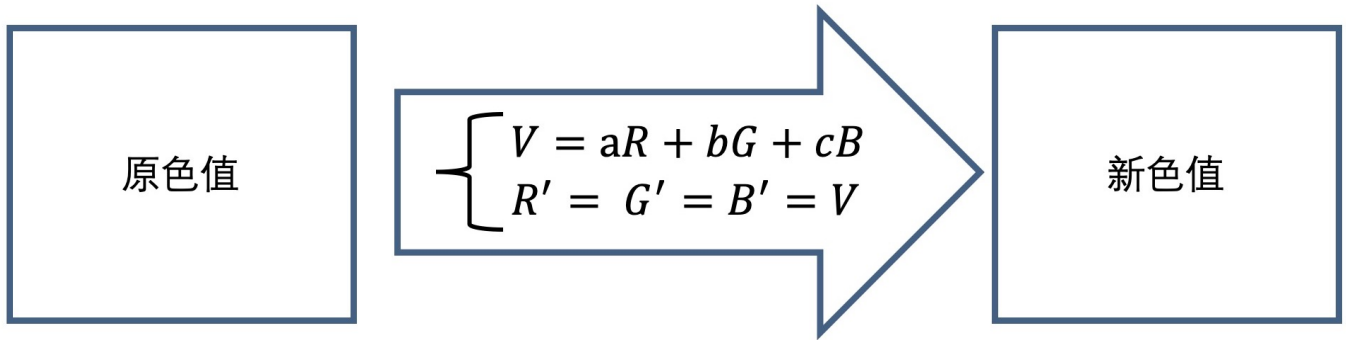
[复制代码](#)

```
1  import {loadImage, getImageData, traverse} from './lib/util.js';
2
3  const canvas = document.getElementById('paper');
4  const context = canvas.getContext('2d');
5
6  (async function () {
7    // 异步加载图片
8    const img = await loadImage('assets/girl1.jpg');
9    // 获取图片的 imageData 数据对象
10   const imageData = getImageData(img);
11   // 遍历 imageData 数据对象
12   traverse(imageData, ({r, g, b, a}) => { // 对每个像素进行灰度化处理
13     const v = 0.2126 * r + 0.7152 * g + 0.0722 * b;
14     return [v, v, v, a];
15   });
16   // 更新canvas内容
17   canvas.width = imageData.width;
18   canvas.height = imageData.height;
19   context.putImageData(imageData, 0, 0);
20 }());
```

这样做的好处是，traverse 函数会自动遍历图片的每个像素点，把获得的像素信息传给参数中的回调函数处理。这样，我们就只关注 traverse 函数里面的处理过程就可以了。

应用二：使用像素矩阵通用地改变像素颜色

在灰度化图片的例子中，我们用加权平均的计算公式来替换图片的 RGBA 的值。这本质上其实是利用线性方程组改变了图片中每一个像素的 RGB 通道的原色值，将每个通道的色值映射为一个新色值。



除了加权平均的计算公式以外，我们还可以用其他的线性方程组来实现各种不同的像素变换效果。比如说，要想实现改变图片的亮度，我们可以将 R、G、B 通道的值都乘以一个常量 p ，公式如下：

$$\begin{cases} R' = p \cdot R \\ G' = p \cdot G \\ B' = p \cdot B \end{cases}$$

这里的 p 是一个常量，如果它小于 1，那么 R、G、B 值就会变小，图片就会变暗，也就更接近于黑色了。相反，如果 p 大于 1，图片就会变亮，更接近白色。这样一来，我们用不同的公式就可以将像素的颜色处理成我们所期望的结果了。

但如果你想要实现不同的颜色变换，就必须使用不同的方程组，这会让我们使用起来非常麻烦。那你肯定想问了，有没有一种方式，可以更通用地实现更多的颜色变换效果呢？当然是有的，我们可以引入一个颜色矩阵，它能够处理几乎所有的颜色变换类滤镜。

我们创建一个 4×5 颜色矩阵，让它的第一行决定红色通道，第二行决定绿色通道，第三行决定蓝色通道，第四行决定 Alpha 通道。


$$M = \begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & o & m & n \\ o & p & q & r & s \end{bmatrix}$$

那如果要改变一个像素的颜色效果，我们只需要将该矩阵与像素的颜色向量相乘就可以了。

$$\begin{aligned} C' = M \cdot C &= \begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & o & m & n \\ o & p & q & r & s \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \\ A \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} aR + bG + cB + dA + e \\ fR + gG + hB + iA + j \\ kR + lG + oB + mA + n \\ oR + pG + qB + rA + s \end{bmatrix} = \begin{bmatrix} R' \\ G' \\ B' \\ A' \end{bmatrix} \end{aligned}$$

这样一来，灰度化图片的处理过程，就可以描述成如下的颜色矩阵：

```
1 function grayscale(p = 1) {
2   const r = 0.2126 * p;
3   const g = 0.7152 * p;
4   const b = 0.0722 * p;
5
6
7   return [
8     r + 1 - p, g, b, 0, 0,
9     r, g + 1 - p, b, 0, 0,
10    r, g, b + 1 - p, 0, 0,
```

 复制代码

```
11     0, 0, 0, 1, 0,  
12 ];  
13 }
```

注意，这里我们引入了一个参数 p ，它是一个 $0\sim 1$ 的值，表示灰度化的程度， 1 是完全灰度化， 0 是完全不灰度，也就是保持原始色彩。这样一来，我们通过调节 p 的值就可以改变图片灰度化的程度。因此这个灰度化矩阵，比前面直接用灰度化公式更加通用。

因为 p 的取值范围是 $0\sim 1$ ，所以 p 的取值可以分成三种情况。下面，我们一起来分析一下。

第一种， p 等于 0 ，这个时候， r 、 g 、 b 的值也都是 0 ，所以返回的矩阵就退化成单位矩阵，代码如下。这样一来，新色值和原色值就完全相同了。

```
1  [  
2      1, 0, 0, 0, 0,  
3      0, 1, 0, 0, 0,  
4      0, 0, 1, 0, 0,  
5      0, 0, 0, 1, 0  
6  ]
```

[复制代码](#)

第二种情况当 p 等于 1 的时候，这个矩阵就正好对应我们前面的灰度化公式。


```
1  [  
2      r, g, b, 0, 0,  
3      r, g, b, 0, 0,  
4      r, g, b, 0, 0,  
5      0, 0, 0, 1, 0,  
6  ]
```

[复制代码](#)

第三种取值情况，当 p 处于 $0\sim 1$ 之间的时候，颜色矩阵的值就在完全灰度的矩阵和单位矩阵之间线性变化。这样我们就实现了可调节的灰度化颜色矩阵。

但是，光有颜色矩阵还不行，要想实现不同的颜色变化，根据前面的公式，我们还得让旧的色值与颜色矩阵相乘，把新的色值计算出来。

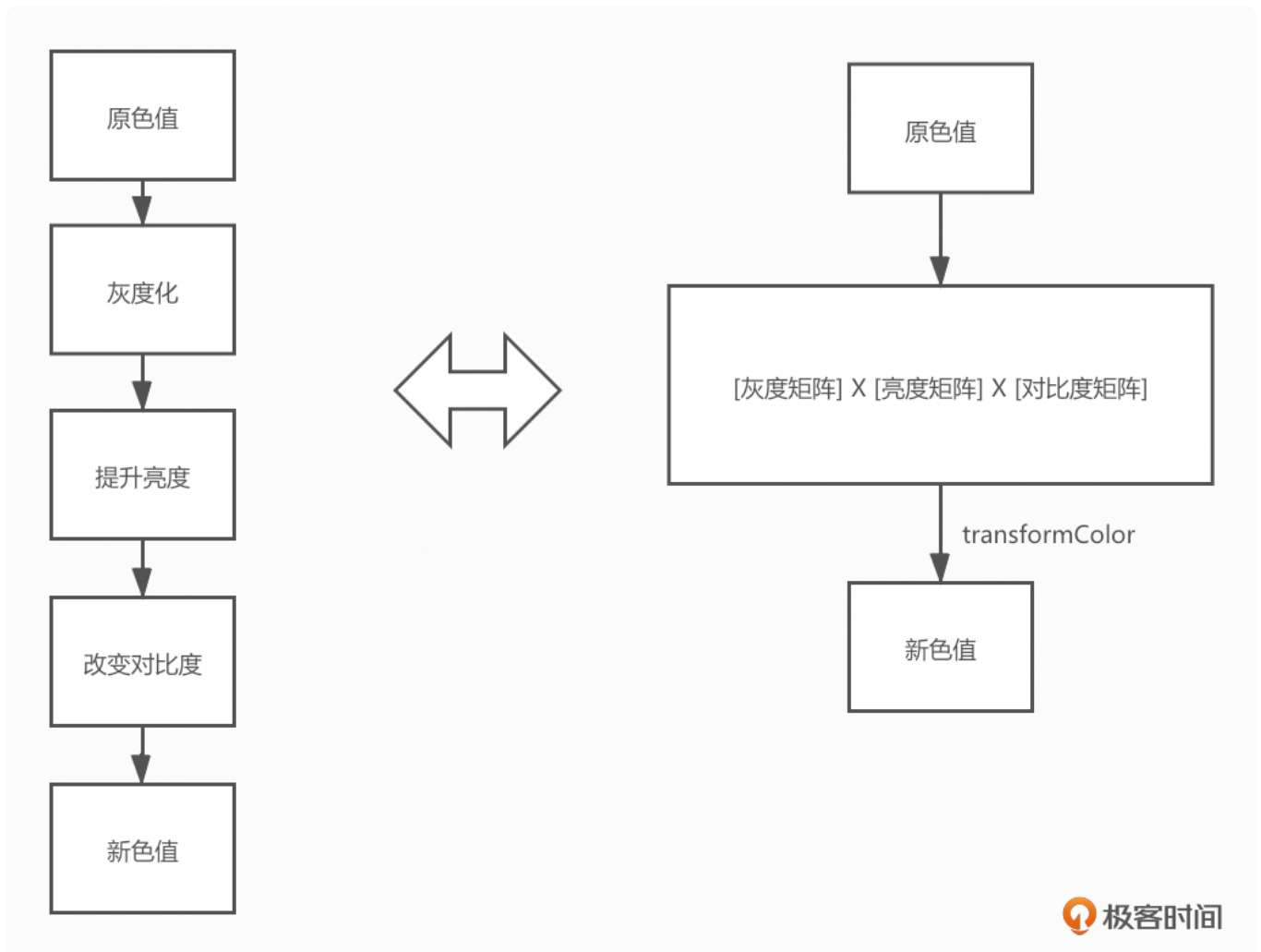
为了方便处理，我们可以增加处理颜色矩阵的模块。让它包含两个函数，一个是处理颜色矩阵的矩阵乘法运算 **multiply** 函数，另一个是将 RGBA 颜色通道组成的向量与颜色矩阵相乘，得到新色值的 **transformColor** 函数。

 复制代码

```
1 // lib/color-matrix.js
2 // 将 color 通过颜色矩阵映射成新的色值返回
3 export function transformColor(color, ...matrix) {
4   // 颜色向量与矩阵相乘
5   ... 省略的代码
6 }
7 // 将颜色矩阵相乘
8 export function multiply(a, b) {
9   // 颜色矩阵相乘
10  ...省略的代码
11 }
```

那你可能想问，为什么我们这里不仅提供了处理色值映射的 **transformColor**，还提供了一个矩阵乘法的 **multiply** 方法呢？

这是因为根据矩阵运算的性质，我们可以将多次颜色变换的过程，简化为将相应的颜色矩阵相乘，然后用最终的那个矩阵对颜色值进行映射。具体的过程你可以看看我给出的流程图。



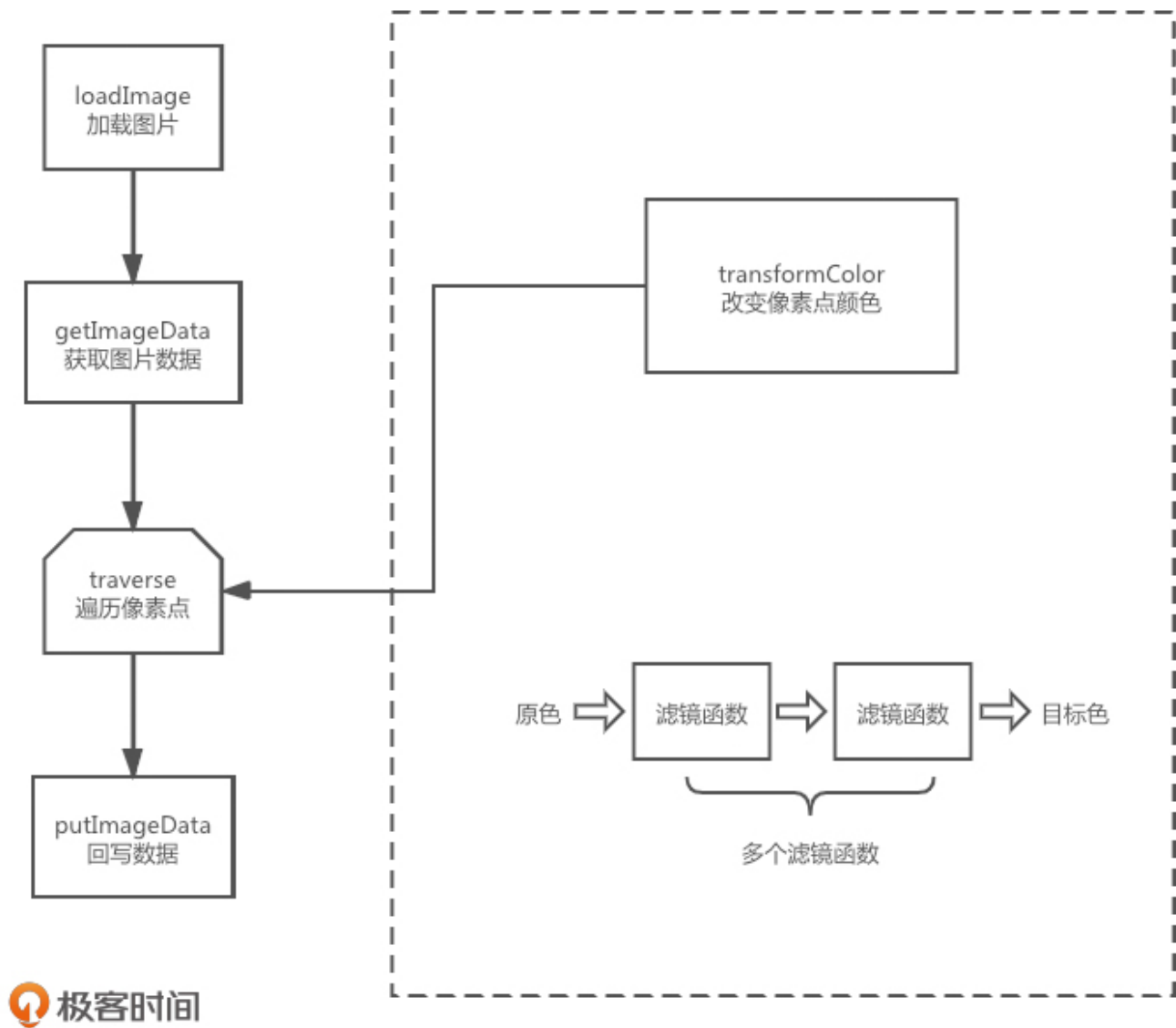
这样，灰度化图片的实现部分就可以写成如下代码：

```
1    ...省略代码...
2
3    traverse(imageData, ({r, g, b, a}) => {
4        return transformColor([r, g, b, a], grayscale(1));
5    });
6
7    ...省略代码...
8
```

复制代码

这里的 **grayscale 函数** 返回了实现灰度化的颜色矩阵，而要实现其他颜色变换效果，我们可以定义其他函数返回其他的颜色矩阵。这种返回颜色矩阵的函数，我们一般称为**颜色滤镜函数**。

抽象出了颜色滤镜函数之后，我们处理颜色代码的过程可以规范成如下图所示的过程：



我们还可以增加其他的颜色滤镜函数，比如：

复制代码

```

1 function channel({r = 1, g = 1, b = 1}) {
2   return [
3     r, 0, 0, 0, 0,
4     0, g, 0, 0, 0,
5     0, 0, b, 0, 0,
6     0, 0, 0, 1, 0,
7   ];
8 }

```

这个 **channel** 滤镜函数可以过滤或增强某个颜色通道。

复制代码

```

1 // 增强红色通道，减弱绿色通道

```

```
2 traverse(imageData, ({r, g, b, a}) => {  
3   return transformColor([r, g, b, a], channel({r: 1.5, g: 0.75}));  
4 })
```

举个例子，当我们调用 `channel({r: 1.5, g: 0.75})` 的时候，红色通道的值被映射为原来的 1.5 倍，绿色通道则被映射为 0.75 倍。这样得到的图片就显得比原来要红。



在处理图片时，我们还会用到有一些常用的颜色滤镜，比如，可以修改图片的亮度（Brightness）、饱和度（Saturate）、对比度（Contrast）、透明度（Opacity），还有对图片反色（Invert）和旋转色相（HueRotate）。这些滤镜函数的结构都是一样的，只是返回的矩阵不同而已。

它们的滤镜函数如下，你结合上节课我们学过的颜色理论很容易就可以理解了，我就不详细讲了。

```
1 // 改变亮度, p = 0 全暗, p > 0 且 p < 1 调暗, p = 1 原色, p > 1 调亮  
2 function brightness(p) {  
3   return [  
4     p, 0, 0, 0, 0,  
5     0, p, 0, 0, 0,  
6     0, 0, p, 0, 0,  
7     0, 0, 0, 1, 0,  
8   ];  
}
```

[复制代码](#)

```
9  }
10
11 // 饱和度,与grayscale正好相反
12 // p = 0 完全灰度化, p = 1 原色, p > 1 增强饱和度
13 function saturate(p) {
14     const r = 0.2126 * (1 - p);
15     const g = 0.7152 * (1 - p);
16     const b = 0.0722 * (1 - p);
17     return [
18         r + p, g, b, 0, 0,
19         r, g + p, b, 0, 0,
20         r, g, b + p, 0, 0,
21         0, 0, 0, 1, 0,
22     ];
23 }
24
25 // 对比度, p = 1 原色, p < 1 减弱对比度, p > 1 增强对比度
26 function contrast(p) {
27     const d = 0.5 * (1 - p);
28     return [
29         p, 0, 0, 0, d,
30         0, p, 0, 0, d,
31         0, 0, p, 0, d,
32         0, 0, 0, 1, 0,
33     ];
34 }
35
36 // 透明度, p = 0 全透明, p = 1 原色
37 function opacity(p) {
38     return [
39         1, 0, 0, 0, 0,
40         0, 1, 0, 0, 0,
41         0, 0, 1, 0, 0,
42         0, 0, 0, p, 0,
43     ];
44 }
45
46 // 反色, p = 0 原色, p = 1 完全反色
47 function invert(p) {
48     const d = 1 - 2 * p;
49     return [
50         d, 0, 0, 0, p,
51         0, d, 0, 0, p,
52         0, 0, d, 0, p,
53         0, 0, 0, 1, 0,
54     ];
55 }
56
57 // 色相旋转,将色调沿极坐标转过deg角度
58 function hueRotate(deg) {
59     const rotation = deg / 180 * Math.PI;
60     const cos = Math.cos(rotation),
```



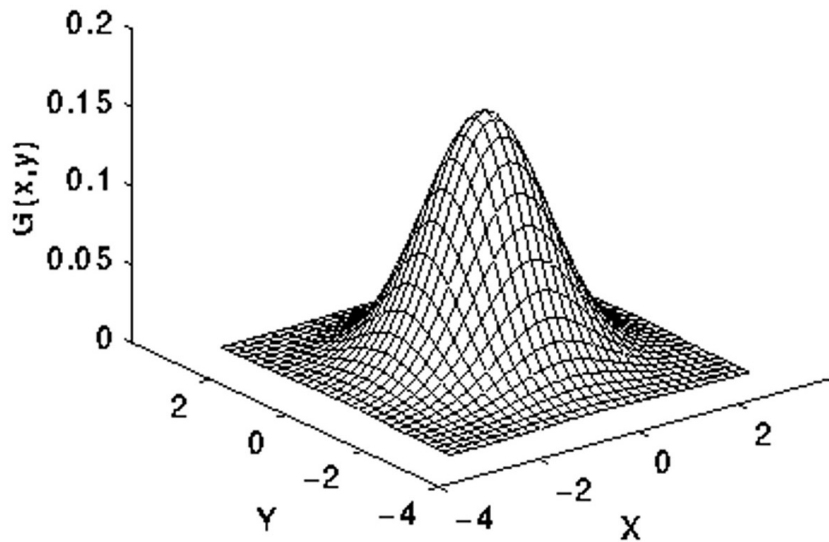
```
7     );  
8     ``
```



应用三：使用高斯模糊对照片美颜

你肯定发现了，刚才我们讲的颜色滤镜都比较简单。没错，其实它们都是一些简单滤镜。那在实际的可视化项目中，我们通常会使用颜色滤镜来增强视觉呈现的细节，而用一种相对复杂的滤镜来模糊背景，从而突出我们要呈现给用户的内容。这个复杂滤镜就叫做高斯模糊（ Gaussian Blur ）。

高斯模糊的原理与颜色滤镜不同，高斯模糊不是单纯根据颜色矩阵计算当前像素点的颜色值，而是会按照高斯分布的权重，对当前像素点及其周围像素点的颜色按照高斯分布的权重加权平均。这样做，我们就能让图片各像素色值与周围色值的差异减小，从而达到平滑，或者说是模糊的效果。所以，高斯模糊是一个非常重要的**平滑效果滤镜**（ Blur Filters ）。

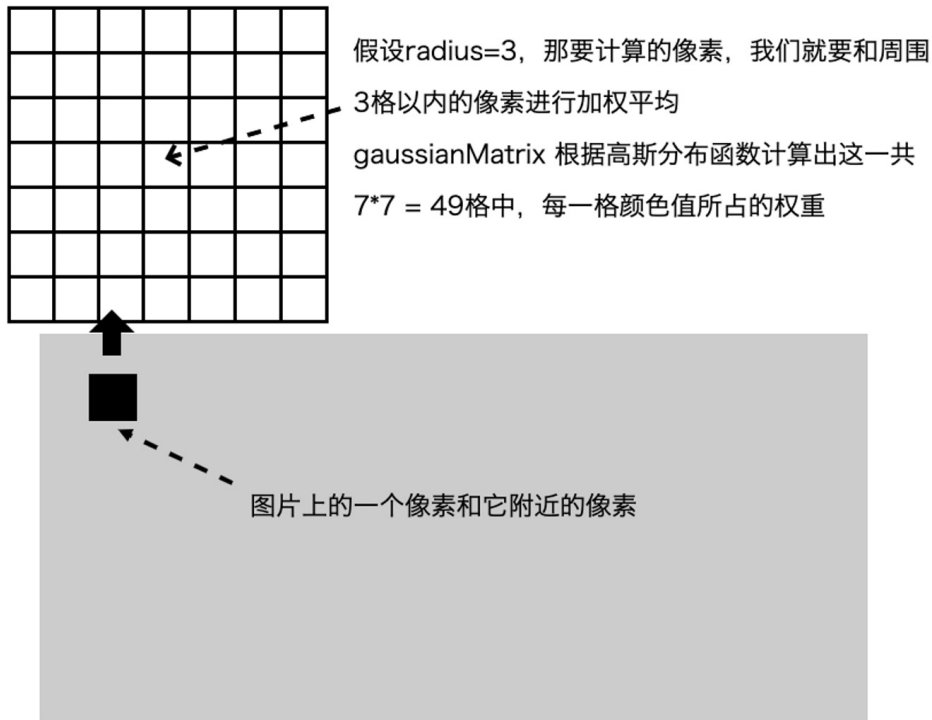


高斯模糊的算法分两步，第一步是生成高斯分布矩阵，这个矩阵的作用是按照高斯函数提供平滑过程中参与计算的像素点的加权平均权重。代码如下：

[复制代码](#)

```
1 function gaussianMatrix(radius, sigma = radius / 3) {
2   const a = 1 / (Math.sqrt(2 * Math.PI) * sigma);
3   const b = -1 / (2 * sigma ** 2);
4   let sum = 0;
5   const matrix = [];
6   for(let x = -radius; x <= radius; x++) {
7     const g = a * Math.exp(b * x ** 2);
8     matrix.push(g);
9     sum += g;
10  }
11
12  for(let i = 0, len = matrix.length; i < len; i++) {
13    matrix[i] /= sum;
14  }
15  return {matrix, sum};
16 }
17
```

那高斯分布的原理是什么呢？它其实就是正态分布，简单来说就是将当前像素点的颜色值设置为附近像素点颜色值的加权平均，而距离当前像素越近的点的权重越高，权重分布满足正态分布。



因为高斯分布涉及比较专业的数学知识，所以要展开细讲会非常复杂，而且我们在实际工作中只要理解原理并且会使用相关公式就够了。因此，你要记住我这里给出的二维高斯函数公式。

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}$$

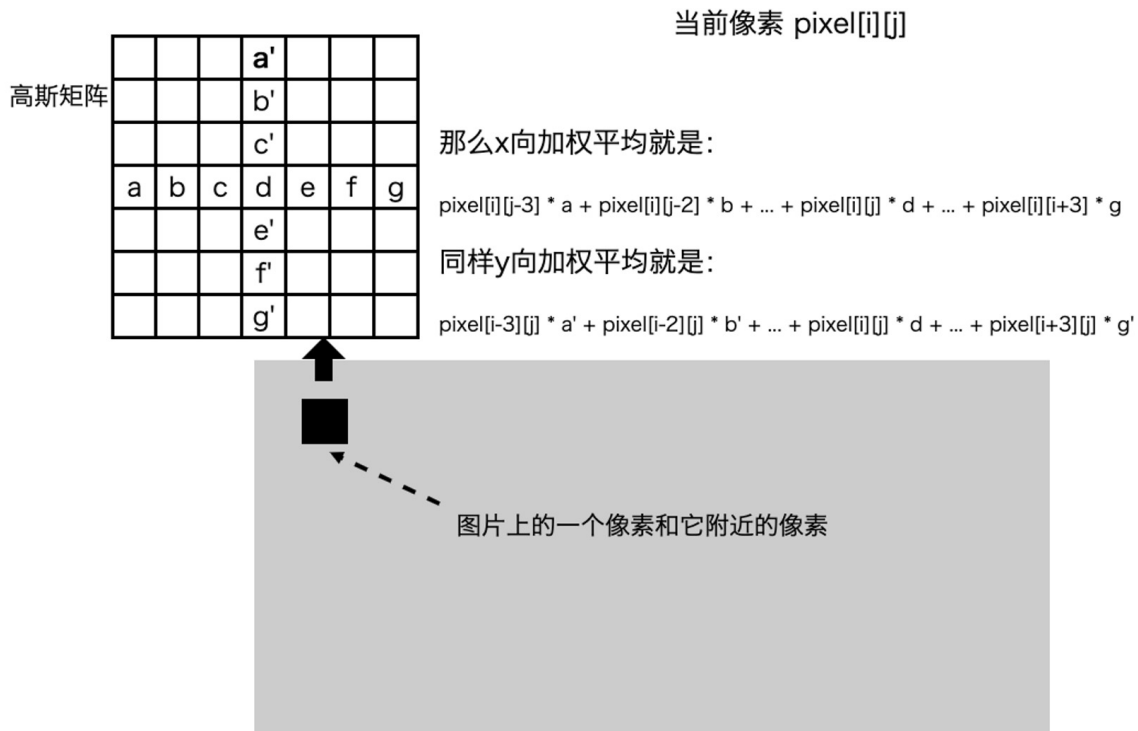
二维高斯函数

这个公式其实就是上面代码中的计算式：

```
1 const a = 1 / (Math.sqrt(2 * Math.PI) * sigma);
2 const b = -1 / (2 * sigma ** 2);
3
4 const g = a * Math.exp(b * x ** 2);
```

复制代码

第二步，对图片在 x 轴、y 轴两个方向上分别进行高斯运算。也就是沿着图片的宽、高方向对当前像素和它附近的像素，应用上面得出的权重矩阵中的值进行加权平均。



实现代码如下：

复制代码

```

1 export function gaussianBlur(pixels, width, height, radius = 3, sigma = radius
2   const {matrix, sum} = gaussianMatrix(radius, sigma);
3   // x 方向一维高斯运算
4   for(let y = 0; y < height; y++) {
5     for(let x = 0; x < width; x++) {
6       let r = 0,
7       g = 0,
8       b = 0;
9
10      for(let j = -radius; j <= radius; j++) {
11        const k = x + j;
12        if(k >= 0 && k < width) {
13          const i = (y * width + k) * 4;
14          r += pixels[i] * matrix[j + radius];
15          g += pixels[i + 1] * matrix[j + radius];
16          b += pixels[i + 2] * matrix[j + radius];
17        }
18      }
19      const i = (y * width + x) * 4;
20      // 除以 sum 是为了消除处于边缘的像素，高斯运算不足的问题
21      pixels[i] = r / sum;
22      pixels[i + 1] = g / sum;
23      pixels[i + 2] = b / sum;
24    }
25  }
26
27  // y 方向一维高斯运算

```

```
28    ...省略代码
29
30    return pixels;
31 }
```

在实现了高斯模糊函数之后，我们就可以对前面例子中的图片进行高斯模糊处理了，代码如下：

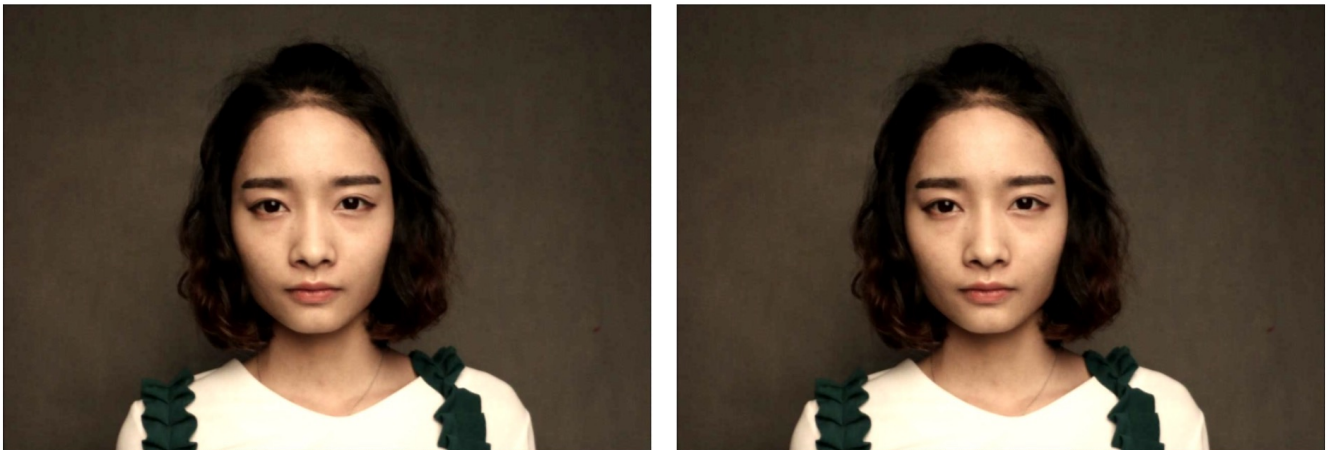
[复制代码](#)

```
1    ...省略代码...
2
3    // 获取图片的 imageData 数据对象
4    const imageData = getImageData(img);
5    // 对imageData应用高斯模糊
6    gaussianBlur(data, width, height);
7
8    ...省略代码...
```

由于高斯模糊不是处理单一像素，而是处理一个范围内的所有像素，因此我们不能采用前面的 `traverse` 遍历函数的方法，而是整体对图片所有像素应用高斯模糊函数。我们最终得到的效果如下图：



你会发现，图片整体都变得“模糊”了。那这种模糊效果就完全可以用来给图片“磨皮”。处理后的图片对比效果如下：




在这里，我们不仅给图片加了高斯模糊，还用灰度化、增强了饱和度、增强了对比度和亮度。这样图片上的人的皮肤就会显得更白皙。我建议你，可以试着自己动手来实现看看。

像素化与 CSS 滤镜


那在上面这几个例子中，我们都是自己通过 Canvas 的 `getImageData` 方法拿到像素数据，然后遍历读取或修改像素信息。实际上，如果只是按照某些特定规则改变一个图像上的所有像素，浏览器提供了更简便的方法：CSS 滤镜。

前面讲过的几种图片效果，我们都可以通过 CSS 滤镜实现。比如灰度化图片可以直接使用 `img` 元素来，代码如下：

 复制代码

```
1 <img src="https://p2.ssl.qhimg.com/d/inn/4b7e384c55dc/girl1.jpg" style="filter
2
```

同样美颜效果我们也可以用 CSS 滤镜来实现：

 复制代码

```
1 <img src="https://p0.ssl.qhimg.com/t01161037b5fe87f236.jpg"
2 style="filter:blur(1.5px) grayscale(0.5) saturate(1.2) contrast(1.1) brightnes
3
```

除此以外，比较新的浏览器上还实现了原生的 Canvas 滤镜，与 CSS 滤镜相对应。[🔗 CSS 滤镜](#)和[🔗 Canvas 滤镜](#)都能实现非常丰富的滤镜效果，在处理视觉呈现上很有用。尽管 CSS 滤镜和 Canvas 滤镜都很好用，但是在实现效果上都有局限性，它们一般只能实现比较固定的视觉效果。这对于可视化来说，这并不够用。

这个时候像素处理的优势就体现出来了，用像素处理图片更灵活，因为它可以实现滤镜功能，还可以实现更加丰富的效果，包括一些非常炫酷的视觉效果，这正是可视化领域所需要的。

要点总结

这一节课我们学习了图片像素处理的基本原理。你要掌握的核心概念有滤镜函数、高斯模糊滤镜以及内置滤镜。

在像素处理的过程中，我们可以利用 Canvas 的 `getImageData` API 来获取图片的像素数据，然后遍历图片的每个像素点，最后用线性方程或者矩阵变换来改变图片的像素颜色。我们可以定义函数来生成矩阵变换，这些生成矩阵变换的函数就是滤镜函数。

像 `grayscale` 一类的函数，就是比较简单的滤镜函数了。除此以外，我们还学习了一类复杂滤镜，其中最基础的一种是高斯模糊滤镜，我们可以用它的平滑效果来给照片“美颜”。实际上，高斯模糊滤镜经常会用来做背景模糊，以突出主题内容。

而且，我们还探讨了像素化的优势，虽然浏览器提供了内置的 Canvas 滤镜和 CSS 滤镜可以实现大部分滤镜函数的功能，我们可以直接使用它们。但是，内置滤镜实现不了更加复杂的视觉效果，而像素化可以。不过，我也会在后面的课程中，继续和你深入讨论一些复杂的滤镜，用它们来生成复杂效果。

小试牛刀

你能利用鼠标事件和今天学过的内容，做出一个图片局部“放大器”的效果吗？具体的效果就是，在鼠标移动在图片上时，将图片以鼠标坐标为圆心，指定半径内的内容局部放大。

源码

[🔗 GitHub 仓库](#)

推荐阅读

[《高斯模糊的算法》](#)

欢迎留言和我分享你的练习过程，如果有收获，欢迎你把这节课分享给你的朋友。

提建议

跟月影学可视化

系统掌握图形学与可视化核心原理

月影

奇虎 360 奇舞团团长

可视化 UI 框架 SpriteJS 核心开发者



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 11 | 图案生成：如何生成重复图案、分形图案以及随机效果？

下一篇 13 | 如何给简单的图案添加纹理和复杂滤镜？

精选留言 (1)

写留言



张旭

2020-07-21

老师，颜色矩阵为什么是4*5，不是4*4呢？

展开

作者回复: 实际上4*5是一个简化了最后一行的5*5的矩阵，因为4*4矩阵的仿射变换用5*5齐次矩阵来表示。这个和为什么css的transform-matix是2*3道理一样

1