Canvas API

目录

- 概述
- Canvas API: 绘制图形
  - 路径
  - 线型
  - 矩形
  - 弧线
  - 文本
  - 渐变色和图像填充
  - 阴影
- Canvas API: 图像处理
  - CanvasRenderingContext2D.drawImage()
  - 像素读写
  - CanvasRenderingContext2D.save()，CanvasRenderingContext2D.restore()
  - CanvasRenderingContext2D.canvas
  - 图像变换
- <canvas> 元素的方法
  - HTMLCanvasElement.toDataURL()
  - HTMLCanvasElement.toBlob()
- Canvas 使用实例
  - 动画效果
  - 像素处理
- 参考链接

概述

`<canvas>`元素用于生成图像。它本身就像一个画布，JavaScript 通过操作它的 API

，在上面生成图像。它的底层是一个个像素，基本上`<canvas>`是一个可以用 JavaScript 操作的位图（bitmap

）。

它与 SVG 图像的区别在于，`<canvas>` 是脚本调用各种方法生成图像，SVG 则是一个 XML 文件，通过各种子元素生成图像。

使用 Canvas API 之前，需要在网页里面新建一个 `<canvas>` 元素。

```
<canvas id="myCanvas" width="400" height="250">
  您的浏览器不支持 Canvas
</canvas>
```

如果浏览器不支持这个 API，就会显示 `<canvas>` 标签中间的文字：“您的浏览器不支持 Canvas”。

每个 `<canvas>` 元素都有一个对应的 `CanvasRenderingContext2D` 对象（上下文对象）。Canvas API 就定义在这个对象上面。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');
```

上面代码中，`<canvas>` 元素节点对象的 `getContext()` 方法，返回的就是 `CanvasRenderingContext2D` 对象。

注意，Canvas API 需要 `getContext` 方法指定参数 `2d`，表示该 `<canvas>` 节点生成 2D 的平面图像。如果参数是 `webgl`，就表示用于生成 3D 的立体图案，这部分属于 WebGL API。

按照用途，Canvas API 分成两大部分：绘制图形和图像处理。

Canvas API：绘制图形

Canvas 画布提供了一个作图的平面空间，该空间的每个点都有自己的坐标。原点 `(0, 0)` 位于图像左上角，`x` 轴的正向是原点向右，`y` 轴的正向是原点向下。

路径

以下方法和属性用来绘制路径。

- `CanvasRenderingContext2D.beginPath()`：开始绘制路径。
- `CanvasRenderingContext2D.closePath()`：结束路径，返回到当前路径的起始点，会从当前点到起始点绘制一条直线。如果图形已经封闭，或者只有一个点，那么此方法不会产生任何效果。
- `CanvasRenderingContext2D.moveTo()`：设置路径的起点，即将一个新路径的起始点移动到 `(x, y)` 坐标。
- `CanvasRenderingContext2D.lineTo()`：使用直线从当前点连接到 `(x, y)` 坐标。

- `CanvasRenderingContext2D.fill()`：在路径内部填充颜色（默认为黑色）。
- `CanvasRenderingContext2D.stroke()`：路径线条着色（默认为黑色）。
- `CanvasRenderingContext2D.fillStyle`：指定路径填充的颜色和样式（默认为黑色）。
- `CanvasRenderingContext2D.strokeStyle`：指定路径线条的颜色和样式（默认为黑色）。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 200);
ctx.lineTo(100, 200);
```

上面代码只是确定了路径的形状，画布上还看不出来，因为没有颜色。所以还需要着色。

```
ctx.fill()
// 或者
ctx.stroke()
```

上面代码中，这两个方法都可以使得路径可见。`fill()`在路径内部填充颜色，使之变成一个实心的图形；`stroke()`只对路径线条着色。

这两个方法默认都是使用黑色，可以使用`fillStyle`和`strokeStyle`属性指定其他颜色。

```
ctx.fillStyle = 'red';
ctx.fill();
// 或者
ctx.strokeStyle = 'red';
ctx.stroke();
```

上面代码将填充和线条的颜色指定为红色。

线型

以下的方法和属性控制线条的视觉特征。

- `CanvasRenderingContext2D.lineWidth`：指定线条的宽度，默认为 1.0。
- `CanvasRenderingContext2D.lineCap`：指定线条末端的样式，有三个可能的值：`butt`（默认值，末端为矩形）、`round`（末端为圆形）、`square`（末端为突出的矩形，矩形宽度不变，高度为线条宽度的一半）。
- `CanvasRenderingContext2D.lineJoin`：指定线段交点的样式，有三个可能的值：`round`（交点为扇形）、`bevel`（交点为三角形底边）、`miter`（默认值，交点为菱形）。

- `CanvasRenderingContext2D.miterLimit`：指定交点菱形的长度，默认为 10。该属性只在 `lineJoin` 属性的值等于 `miter` 时有效。
- `CanvasRenderingContext2D.getLineDash()`：返回一个数组，表示虚线里面线段和间距的长度。
- `CanvasRenderingContext2D.setLineDash()`：数组，用于指定虚线里面线段和间距的长度。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(100, 100);
ctx.lineTo(200, 200);
ctx.lineTo(100, 200);

ctx.lineWidth = 3;
ctx.lineCap = 'round';
ctx.lineJoin = 'round';
ctx.setLineDash([15, 5]);
ctx.stroke();
```

上面代码中，线条的宽度为 3，线条的末端和交点都改成圆角，并且设置为虚线。

矩形

以下方法用来绘制矩形。

- `CanvasRenderingContext2D.rect()`：绘制矩形路径。
- `CanvasRenderingContext2D.fillRect()`：填充一个矩形。
- `CanvasRenderingContext2D.strokeRect()`：绘制矩形边框。
- `CanvasRenderingContext2D.clearRect()`：指定矩形区域的像素都变成透明。

上面四个方法的格式都一样，都接受四个参数，分别是矩形左上角的横坐标和纵坐标、矩形的宽和高。

`CanvasRenderingContext2D.rect()` 方法用于绘制矩形路径。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.rect(10, 10, 100, 100);
ctx.fill();
```

上面代码绘制一个正方形，左上角坐标为 `(10, 10)`，宽和高都为 100。

`CanvasRenderingContext2D.fillRect()`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 100, 100);
```

`(10, 10)` 100

`CanvasRenderingContext2D.strokeRect()`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.strokeStyle = 'green';
ctx.strokeRect(10, 10, 100, 100);
```

`(10, 10)` 100

`CanvasRenderingContext2D.clearRect()`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.fillRect(10, 10, 100, 100);
ctx.clearRect(15, 15, 90, 90);
```

100 x 100 90 x 90 5

- `CanvasRenderingContext2D.arc()`
- `CanvasRenderingContext2D.arcTo()`

`CanvasRenderingContext2D.arc()`

```
// 格式
ctx.arc(x, y, radius, startAngle, endAngle, anticlockwise)

// 实例
ctx.arc(5, 5, 5, 0, 2 * Math.PI, true)
```

`arc()`方法的 `x`和 `y`参数是圆心坐标，`radius`是半径，`startAngle`和 `endAngle`则是扇形的起始角度和终止角度（以弧度表示），`anticlockwise`表示做图时应该逆时针画（`true`）还是顺时针画（`false`），这个参数用来控制扇形的方向（比如上半圆还是下半圆）。

下面是绘制实心圆形的例子。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.arc(60, 60, 50, 0, Math.PI * 2, true);
ctx.fill();
```

上面代码绘制了一个半径 50，起始角度为 0，终止角度为 2 * PI 的完整的圆。

绘制空心半圆的例子。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(50, 20);
ctx.arc(100, 20, 50, 0, Math.PI, false);
ctx.stroke();
```

`CanvasRenderingContext2D.arcTo()`

方法主要用来绘制圆弧，需要给出两个点的坐标，当前点与第一个点形成一条直线，第一个点与第二个点形成另一条直线，然后画出与这两根直线相切的弧线。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(0, 0);
ctx.arcTo(50, 50, 100, 0, 25);
ctx.lineTo(100, 0);
```

```
ctx.stroke();
```

上面代码中，`arcTo()`有 5 个参数，前两个参数是第一个点的坐标，第三个参数和第四个参数是第二个点的坐标，第五个参数是半径。然后，`(0, 0)`与`(50, 50)`形成一条直线，然后`(50, 50)`与`(100, 0)`形成第二条直线。弧线就是与这两根直线相切的部分。

文本

以下方法和属性用于绘制文本。

- `CanvasRenderingContext2D.fillText()`：在指定位置绘制实心字符。
- `CanvasRenderingContext2D.strokeText()`：在指定位置绘制空心字符。
- `CanvasRenderingContext2D.measureText()`：返回一个 TextMetrics 对象。
- `CanvasRenderingContext2D.font`：指定字型大小和字体，默认值为`10px sans-serif`。
- `CanvasRenderingContext2D.textAlign`：文本的对齐方式，默认值为`start`。
- `CanvasRenderingContext2D.direction`：文本的方向，默认值为`inherit`。
- `CanvasRenderingContext2D.textBaseline`：文本的垂直位置，默认值为`alphabetic`。

`fillText()`方法用来在指定位置绘制实心字符。

```
CanvasRenderingContext2D.fillText(text, x, y [, maxWidth])
```

该方法接受四个参数。

- `text`：所要填充的字符串。
- `x`：文字起点的横坐标，单位像素。
- `y`：文字起点的纵坐标，单位像素。
- `maxWidth`：文本的最大像素宽度。该参数可选，如果省略，则表示宽度没有限制。如果文本实际长度超过这个参数指定的值，那么浏览器将尝试用较小的字体填充。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.fillText('Hello world', 50, 50);
```

上面代码在`(50, 50)`位置写入字符串`Hello world`。

注意，`fillText()`方法不支持文本断行，所有文本一定出现在一行内。如果要生成多行文本，只有调用多次`fillText()`方法。

`strokeText()`方法用来添加空心字符，它的参数与`fillText()`一致。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.strokeText('Hello world', 50, 50);
```

上面这两种方法绘制的文本，默认都是 `10px` 大小、 `sans-serif` 字体， `font` 属性可以改变字体设置。该属性的值是一个字符串，使用 CSS 的 `font` 属性即可。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.font = 'Bold 20px Arial';
ctx.fillText('Hello world', 50, 50);
```

`textAlign` 属性用来指定文本的对齐方式。它可以取以下几个值。

- `left`：左对齐
- `right`：右对齐
- `center`：居中
- `start`：默认值，起点对齐（从左到右的文本为左对齐，从右到左的文本为右对齐）。
- `end`：结尾对齐（从左到右的文本为右对齐，从右到左的文本为左对齐）。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.font = 'Bold 20px Arial';
ctx.textAlign = 'center';
ctx.fillText('Hello world', 50, 50);
```

`direction` 属性指定文本的方向，默认值为 `inherit`，表示继承 `<canvas>` 或 `document` 的设置。其他值包括 `ltr`（从左到右）和 `rtl`（从右到左）。

`textBaseline` 属性指定文本的垂直位置，可以取以下值。

- `top`：上部对齐（字母的基线是整体上移）。
- `hanging`：悬挂对齐（字母的上沿在一根直线上），适用于印度文和藏文。
- `middle`：中部对齐（字母的中线在一根直线上）。
- `alphabetic`：默认值，表示字母位于字母表的正常位置（四线格的第三根线）。
- `ideographic`：下沿对齐（字母的下沿在一根直线上），使用于东亚文字。
- `bottom`：底部对齐（字母的基线下移）。对于英文字母，这个设置与 `ideographic` 没有差异。

`measureText()` 方法接受一个字符串作为参数，返回一个 TextMetrics 对象，可以从这个对象上面获取参数字符串的信息，目前主要是文本渲染后的宽度（ `width`）。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var text1 = ctx.measureText('Hello world');
text1.width // 49.46

ctx.font = 'Bold 20px Arial';
var text2 = ctx.measureText('Hello world');
text2.width // 107.78
```

上面代码中，`10px`大小的字符串`Hello world`，渲染后宽度为`49.46`。放大到`20px`以后，宽度为`107.78`。

渐变色和图像填充

以下方法用于设置渐变效果和图像填充效果。

- `CanvasRenderingContext2D.createLinearGradient()`：定义线性渐变样式。
- `CanvasRenderingContext2D.createRadialGradient()`：定义辐射渐变样式。
- `CanvasRenderingContext2D.createPattern()`：定义图像填充样式。

`createLinearGradient()`方法按照给定直线，生成线性渐变的样式。

```
ctx.createLinearGradient(x0, y0, x1, y1)
```

`ctx.createLinearGradient(x0, y0, x1, y1)`方法接受四个参数：`x0`和`y0`是起点的横坐标和纵坐标，`x1`和`y1`是终点的横坐标和纵坐标。通过不同的坐标值，可以生成从上至下、从左到右的渐变等等。

该方法的返回值是一个`CanvasGradient`对象，该对象只有一个`addColorStop()`方向，用来指定渐变点的颜色。`addColorStop()`方法接受两个参数，第一个参数是0到1之间的一个位置量，0表示起点，1表示终点，第二个参数是一个字符串，表示CSS颜色。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var gradient = ctx.createLinearGradient(0, 0, 200, 0);
gradient.addColorStop(0, 'green');
gradient.addColorStop(1, 'white');
ctx.fillStyle = gradient;
ctx.fillRect(10, 10, 200, 100);
```

上面代码中，定义了渐变样式`gradient`以后，将这个样式指定给`fillStyle`属性，然后`fillRect()`就会生成以这个样式填充的矩形区域。

`createRadialGradient()`方法定义一个辐射渐变，需要指定两个圆。

```
ctx.createRadialGradient(x0, y0, r0, x1, y1, r1)
```

`createRadialGradient()`方法接受六个参数，`x0`和`y0`是辐射起始的圆的圆心坐标，`r0`是起始圆的半径，`x1`和`y1`是辐射终止的圆的圆心坐标，`r1`是终止圆的半径。

该方法的返回值也是一个`CanvasGradient`对象。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var gradient = ctx.createRadialGradient(100, 100, 50, 100, 100, 100);
gradient.addColorStop(0, 'white');
gradient.addColorStop(1, 'green');
ctx.fillStyle = gradient;
ctx.fillRect(0, 0, 200, 200);
```

上面代码中，生成辐射样式以后，用这个样式填充一个矩形。

`createPattern()`方法定义一个图像填充样式，在指定方向上不断重复该图像，填充指定的区域。

```
ctx.createPattern(image, repetition)
```

该方法接受两个参数，第一个参数是图像数据，它可以是`<img>`元素，也可以是另一个`<canvas>`元素，或者一个表示图像的 Blob 对象。第二个参数是一个字符串，有四个可能的值，分别是`repeat`（双向重复）、`repeat-x`(水平重复)、`repeat-y`(垂直重复)、`no-repeat`(不重复)。如果第二个参数是空字符串或`null`，则等同于`null`。

该方法的返回值是一个`CanvasPattern`对象。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var img = new Image();
img.src = 'https://example.com/pattern.png';
img.onload = function( ) {
  var pattern = ctx.createPattern(img, 'repeat');
  ctx.fillStyle = pattern;
  ctx.fillRect(0, 0, 400, 400);
};
```

上面代码中，图像加载成功以后，使用 `createPattern()` 生成图像样式，然后使用这个样式填充指定区域。

阴影

以下属性用于设置阴影。

- `CanvasRenderingContext2D.shadowBlur`：阴影的模糊程度，默认为 `0`。
- `CanvasRenderingContext2D.shadowColor`：阴影的颜色，默认为 `black`。
- `CanvasRenderingContext2D.shadowOffsetX`：阴影的水平位移，默认为 `0`。
- `CanvasRenderingContext2D.shadowOffsetY`：阴影的垂直位移，默认为 `0`。

下面是一个例子。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.shadowOffsetX = 10;
ctx.shadowOffsetY = 10;
ctx.shadowBlur = 5;
ctx.shadowColor = 'rgba(0,0,0,0.5)';

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 100, 100);
```

Canvas API: 图像处理

CanvasRenderingContext2D.drawImage()

Canvas API 允许将图像文件写入画布，做法是读取图片后，使用 `drawImage()` 方法将这张图片放上画布。

`CanvasRenderingContext2D.drawImage()` 有三种使用格式。

```
ctx.drawImage(image, dx, dy);
ctx.drawImage(image, dx, dy, dWidth, dHeight);
ctx.drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);
```

各个参数的含义如下。

- image：图像元素

- sx: 图像内部的横坐标，用于映射到画布的放置点上。
- sy: 图像内部的纵坐标，用于映射到画布的放置点上。
- sWidth
  : 图像在画布上的宽度，会产生缩放效果。如果未指定，则图像不会缩放，按照实际大小占据画布的宽度。
- sHeight
  : 图像在画布上的高度，会产生缩放效果。如果未指定，则图像不会缩放，按照实际大小占据画布的高度。
- dx: 画布内部的横坐标，用于放置图像的左上角
- dy: 画布内部的纵坐标，用于放置图像的右上角
- dWidth: 图像在画布内部的宽度，会产生缩放效果。
- dHeight: 图像在画布内部的高度，会产生缩放效果。

下面是最简单的使用场景，将图像放在画布上，两者左上角对齐。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var img = new Image();
img.src = 'image.png';
img.onload = function () {
  ctx.drawImage(img, 0, 0);
};
```

上面代码将一个 PNG
图像放入画布。这时，图像将是原始大小，如果画布小于图像，就会只显示出图像左上角，正好等于画布大小的那一块。

如果要显示完整的图片，可以用图像的宽和高，设置成画布的宽和高。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var image = new Image(60, 45);
image.onload = drawImageActualSize;
image.src = 'https://example.com/image.jpg';

function drawImageActualSize() {
  canvas.width = this.naturalWidth;
  canvas.height = this.naturalHeight;
  ctx.drawImage(this, 0, 0, this.naturalWidth, this.naturalHeight);
}
```

上面代码中，`<canvas>`元素的大小设置成图像的本来大小，就能保证完整展示图像。由于图像的本来大小，只有图像加载成功以后才能拿到，因此调整画布的大小，必须放在`image.onload`这个监听函数里面。

像素读写

以下三个方法与像素读写相关。

- `CanvasRenderingContext2D.getImageData()`：将画布读取成一个 ImageData 对象
- `CanvasRenderingContext2D.putImageData()`：将 ImageData 对象写入画布
- `CanvasRenderingContext2D.createImageData()`：生成 ImageData 对象

（1）getImageData()

`CanvasRenderingContext2D.getImageData()`方法用来读取`<canvas>`的内容，返回一个 ImageData 对象，包含了每个像素的信息。

```
ctx.getImageData(sx, sy, sw, sh)
```

`getImageData()`方法接受四个参数。`sx`和`sy`是读取区域的左上角坐标，`sw`和`sh`是读取区域的宽度和高度。如果想要读取整个`<canvas>`区域，可以写成下面这样。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);
```

`getImageData()`方法返回的是一个`ImageData`对象。该对象有三个属性。

- ImageData.data：一个一维数组。该数组的值，依次是每个像素的红、绿、蓝、alpha 通道值（每个值的范围是 0~ 255），因此该数组的长度等于`图像的像素宽度 x 图像的像素高度 x 4`。这个数组不仅可读，而且可写，因此通过操作这个数组，就可以达到操作图像的目的。
- ImageData.width：浮点数，表示 ImageData 的像素宽度。
- ImageData.height：浮点数，表示 ImageData 的像素高度。

（2）putImageData()

`CanvasRenderingContext2D.putImageData()`方法将`ImageData`对象的像素绘制在`<canvas>`画布上。该方法有两种使用格式。

```
ctx.putImageData(imagedata, dx, dy)
ctx.putImageData(imagedata, dx, dy, dirtyX, dirtyY, dirtyWidth, dirtyHeight)
```

该方法有如下参数。

- imagedata: 包含像素信息的 ImageData 对象。
- dx: `<canvas>` 元素内部的横坐标，用于放置 ImageData 图像的左上角。
- dy: `<canvas>` 元素内部的纵坐标，用于放置 ImageData 图像的左上角。
- dirtyX: ImageData 图像内部的横坐标，用于作为放置到 `<canvas>` 的矩形区域的左上角的横坐标，默认为 0。
- dirtyY: ImageData 图像内部的纵坐标，用于作为放置到 `<canvas>` 的矩形区域的左上角的纵坐标，默认为 0。
- dirtyWidth: 放置到 `<canvas>` 的矩形区域的宽度，默认为 ImageData 图像的宽度。
- dirtyHeight: 放置到 `<canvas>` 的矩形区域的高度，默认为 ImageData 图像的高度。

下面是将 ImageData 对象绘制到 `<canvas>` 的例子。

```
ctx.putImageData(imageData, 0, 0);
```

（3）createImageData()

`CanvasRenderingContext2D.createImageData()` 方法用于生成一个空的 `ImageData` 对象，所有像素都是透明的黑色（即每个值都是 `0`）。该方法有两种使用格式。

```
ctx.createImageData(width, height)
ctx.createImageData(imagedata)
```

`createImageData()` 方法的参数如下。

- width: ImageData 对象的宽度，单位为像素。
- height: ImageData 对象的高度，单位为像素。
- imagedata: 一个现有的 ImageData 对象，返回值将是这个对象的拷贝。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var imageData = ctx.createImageData(100, 100);
```

上面代码中，`imageData` 是一个 100 x 100 的像素区域，其中每个像素都是透明的黑色。

CanvasRenderingContext2D.save(), CanvasRenderingContext2D.restore()

`CanvasRenderingContext2D.save()`

方法用于将画布的当前样式保存到堆栈，相当于在内存之中产生一个样式快照。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.save();
```

上面代码中，`save()`会为画布的默认样式产生一个快照。

`CanvasRenderingContext2D.restore()`
方法将画布的样式恢复到上一个保存的快照，如果没有已保存的快照，则不产生任何效果。

上下文环境，restore方法用于恢复到上一次保存的上下文环境。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.save();

ctx.fillStyle = 'green';
ctx.restore();

ctx.fillRect(10, 10, 100, 100);
```

上面代码画一个矩形。矩形的填充色本来设为绿色，但是`restore()`
方法撤销了这个设置，将样式恢复上一次保存的状态（即默认样式），所以实际的填充色是黑色（默认颜色）
。

CanvasRenderingContext2D.canvas

`CanvasRenderingContext2D.canvas`属性指向当前`CanvasRenderingContext2D`对象所在的`<canvas>`
元素。该属性只读。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.canvas === canvas // true
```

图像变换

以下方法用于图像变换。

- `CanvasRenderingContext2D.rotate()`: 图像旋转

- `CanvasRenderingContext2D.scale()`
- `CanvasRenderingContext2D.translate()`
- `CanvasRenderingContext2D.transform()`
- `CanvasRenderingContext2D.setTransform()`

1. rotate()

`CanvasRenderingContext2D.rotate()`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.rotate(45 * Math.PI / 180);
ctx.fillRect(70, 0, 100, 30);
```

45 `rotate()` `fillRect()`

`translate()`

2. scale()

`CanvasRenderingContext2D.scale()` `x` `y` `0.5` 50% `10`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.scale(10, 3);
ctx.fillRect(10, 10, 10, 10);
```

10 x 10 100 x 30

1 -1 `ctx.scale(-1, 1)` `ctx.scale(1, -1)`

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.scale(1, -2);
ctx.font = "16px serif";
ctx.fillText('Hello world!', 20, -20);
```

[illegible] 2[illegible] `Hello World!`[illegible]

[illegible]

[illegible] 3[illegible] translate()

`CanvasRenderingContext2D.translate()`[illegible] x [illegible] y [illegible]

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.translate(50, 50);
ctx.fillRect(0, 0, 100, 100);
```

[illegible] 4[illegible] transform()

`CanvasRenderingContext2D.transform()`
[illegible]

[illegible]

```
ctx.transform(a, b, c, d, e, f);
/*
a:[illegible](1[illegible])
b:[illegible](0[illegible])
c:[illegible](0[illegible])
d:[illegible](1[illegible])
e:[illegible](0[illegible])
f:[illegible](0[illegible])
*/
```

[illegible]

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.transform(2, 0, 0, 1, 50, 50);
ctx.fillRect(0, 0, 100, 100);
```

[illegible] 100 x 100 [illegible] 200 x 100 [illegible] `(0, 0)`[illegible] `(50, 50)`[illegible]

注意，多个 `transform()` 方法具有叠加效果。

（5） setTransform()

`CanvasRenderingContext2D.setTransform()`

方法取消前面的图形变换，将画布恢复到该方法指定的状态。该方法的参数与 `transform()` 方法完全一致。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

ctx.translate(50, 50);
ctx.fillRect(0, 0, 100, 100);

ctx.setTransform(1, 0, 0, 1, 0, 0);
ctx.fillRect(0, 0, 100, 100);
```

上面代码中，第一个 `fillRect()` 方法绘制的矩形，左上角从 `(0, 0)` 平移到 `(50, 50)`。`setTransform()` 方法取消了这个变换（已绘制的图形不受影响），将画布恢复到默认状态（变换矩形 `1, 0, 0, 1, 0, 0`），所以第二个矩形的左上角回到 `(0, 0)`。

`<canvas>` 元素的方法

除了 `CanvasRenderingContext2D` 对象提供的方法，`<canvas>` 元素本身也有自己的方法。

HTMLCanvasElement.toDataURL()

`<canvas>` 元素的 `toDataURL()` 方法，可以将 Canvas 数据转为 Data URI 格式的图像。

```
canvas.toDataURL(type, quality)
```

`toDataURL()` 方法接受两个参数。

- type: 字符串，表示图像的格式。默认为 `image/png`，另一个可用的值是 `image/jpeg`，Chrome 浏览器还可以使用 `image/webp`。
- quality: 浮点数，0到 1之间，表示 JPEG 和 WebP 图像的质量系数，默认值为 0.92。

该方法的返回值是一个 Data URI 格式的字符串。

```
function convertCanvasToImage(canvas) {
  var image = new Image();
  image.src = canvas.toDataURL('image/png');
```

```
  return image;
}
```

上面的代码将 `<canvas>` 元素，转化成 PNG Data URI。

```
var fullQuality = canvas.toDataURL('image/jpeg', 0.9);
var mediumQuality = canvas.toDataURL('image/jpeg', 0.6);
var lowQuality = canvas.toDataURL('image/jpeg', 0.3);
```

上面代码将 `<canvas>` 元素转成高画质、中画质、低画质三种 JPEG 图像。

HTMLCanvasElement.toBlob()

`HTMLCanvasElement.toBlob()` 方法用于将 `<canvas>` 图像转成一个 Blob 对象，默认类型是 `image/png` 。它的使用格式如下。

```
// 格式
canvas.toBlob(callback, mimeType, quality)

// 示例
canvas.toBlob(function (blob) {...}, 'image/jpeg', 0.95)
```

`toBlob()` 方法可以接受三个参数。

- callback: 回调函数。它接受生成的 Blob 对象作为参数。
- mimeType: 字符串，图像的 MIMEType 类型，默认是 `image/png` 。
- quality: 浮点数，0到 1之间，表示图像的质量，只对 `image/jpeg` 和 `image/webp` 类型的图像有效。

注意，该方法没有返回值。

下面的例子将 `<canvas>` 图像复制成 `<img>` 图像。

```
var canvas = document.getElementById('myCanvas');

function blobToImg(blob) {
  var newImg = document.createElement('img');
  var url = URL.createObjectURL(blob);

  newImg.onload = function () {
    // 使用完毕，释放 URL 对象
    URL.revokeObjectURL(url);
```

```
  };

  newImg.src = url;
  document.body.appendChild(newImg);
}

canvas.toBlob(blobToImg);
```

Canvas 使用实例

动画效果

通过改变坐标，很容易在画布 Canvas 元素上产生动画效果。

```
var canvas = document.getElementById('myCanvas');
var ctx = canvas.getContext('2d');

var posX = 20;
var posY = 100;

setInterval(function () {
  ctx.fillStyle = 'black';
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  posX += 1;
  posY += 0.25;

  ctx.beginPath();
  ctx.fillStyle = 'white';

  ctx.arc(posX, posY, 10, 0, Math.PI * 2, true);
  ctx.closePath();
  ctx.fill();
}, 30);
```

上面代码会产生一个小圆点，每隔 30毫秒就向右下方移动的效果。 `setInterval()` 函数的一开始，之所以要将画布重新渲染黑色底色，是为了抹去上一步的小圆点。

[illegible]

```
var vx = 10;
var vy = -10;
var gravity = 1;

setInterval(function () {
  posX += vx;
  posY += vy;
  vy += gravity;
  // ...
});
```

[illegible] `x` [illegible] `y`
[illegible]

[illegible]

[illegible] `getImageData()` [illegible] `putImageData()`
[illegible]

[illegible]

```
if (canvas.width > 0 && canvas.height > 0) {
  var imageData = context.getImageData(0, 0, canvas.width, canvas.height);
  filter(imageData);
  context.putImageData(imageData, 0, 0);
}
```

[illegible] `filter` [illegible] `filter` [illegible]

[illegible] 1 [illegible]

[illegible] grayscale [illegible]

```
grayscale = function (pixels) {
  var d = pixels.data;
  for (var i = 0; i < d.length; i += 4) {
    var r = d[i];
    var g = d[i + 1];
    var b = d[i + 2];
```

```
    d[i] = d[i + 1] = d[i + 2] = (r + g + b) / 3;
  }
  return pixels;
};
```

上面代码中，`d[i]`是红色值，`d[i+1]`是绿色值，`d[i+2]`是蓝色值，`d[i+3]`是 alpha 通道值。转成灰度的算法，就是将红、绿、蓝三个值相加后除以 3，再将结果写回数组。

（2）复古效果

复古效果（sepia）是将红、绿、蓝三种值，分别取这三个值的某种加权平均值，使得图像有一种古旧的效果。

```
sepia = function (pixels) {
    var d = pixels.data;
    for (var i = 0; i < d.length; i += 4) {
      var r = d[i];
      var g = d[i + 1];
      var b = d[i + 2];
      d[i]     = (r * 0.393) + (g * 0.769) + (b * 0.189); // red
      d[i + 1] = (r * 0.349) + (g * 0.686) + (b * 0.168); // green
      d[i + 2] = (r * 0.272) + (g * 0.534) + (b * 0.131); // blue
    }
    return pixels;
};
```

（3）红色蒙版效果

红色蒙版指的是，让图像呈现一种偏红的效果。算法是将红色通道设为红、绿、蓝三个值的平均值，而将绿色通道和蓝色通道都设为 0。

```
var red = function (pixels) {
  var d = pixels.data;
  for (var i = 0; i < d.length; i += 4) {
    var r = d[i];
    var g = d[i + 1];
    var b = d[i + 2];
    d[i] = (r + g + b)/3;         // 红色通道取平均值
    d[i + 1] = d[i + 2] = 0; // 绿色通道和蓝色通道都设为0
  }
  return pixels;
};
```

（4）亮度效果

亮度效果（brightness）是指让图像变得更亮或更暗。算法将红色通道、绿色通道、蓝色通道，同时加上一个正值或负值。

```
var brightness = function (pixels, delta) {
  var d = pixels.data;
  for (var i = 0; i < d.length; i += 4) {
    d[i] += delta;     // red
    d[i + 1] += delta; // green
    d[i + 2] += delta; // blue
  }
  return pixels;
};
```

（5）反转效果

反转效果（invert）是指图片呈现一种色彩颠倒的效果。算法为红、绿、蓝通道都取各自的相反值（`255 - 原值`）。

```
invert = function (pixels) {
  var d = pixels.data;
  for (var i = 0; i < d.length; i += 4) {
    d[i] = 255 - d[i];
    d[i + 1] = 255 - d[i + 1];
    d[i + 2] = 255 - d[i + 2];
  }
  return pixels;
};
```

参考链接

- David Walsh, JavaScript Canvas Image Conversion
- Matt West, Getting Started With The Canvas API
- John Robinson, How You Can Do Cool Image Effects Using HTML5 Canvas
- Ivaylo Gerchev, HTML5 Canvas Tutorial: An Introduction
- Donovan Hutchinson, Particles in canvas

版本号 #2

由 壳壳虫 创建于 2017-12-11 13:23:24 UTC

□ □□□ □□□ 2021-04-08 16:02:18 UTC