



# Canvas 2D API 规范 1.0

(W3C Editor's Draft 21 October 2009)

翻译：[CodeEx.CN](http://codeex.cn) 2010/10/21

引用本文，请不要删掉翻译行，更多精彩，请访问：[www.codeex.cn](http://www.codeex.cn)

原文请参看：

<http://dev.w3.org/html5/canvas-api/canvas-2d-api.html>

摘要.....	2
1 介绍 .....	2
1.1 术语: .....	2
2 CANVAS接口元素定义 .....	3
2.1 GETCONTEXT() 方法 .....	3
2.2 TODATAURL() 方法 .....	3
3 二维绘图上下文 .....	4
3.1 CANVAS的状态 .....	7
3.2 转换 (TRANSFORMATIONS) .....	8
3.3 合成 (COMPOSITING) .....	10
3.4 颜色和风格 .....	12
3.5 线风格 .....	16
3.6 阴影 (SHADOWS) .....	18
3.7 简单形状 (矩形) .....	18
3.8 复杂形状 (路径-PATHS) .....	19
3.8.1 路径起始函数.....	19
3.8.2 绘制函数.....	19
3.8.3 辅助方法一点是否在路径里.....	22
3.8.4 MOVETO方法.....	22

3.8.5	LINETo方法.....	22
3.8.6	RECT方法.....	22
3.8.7	圆弧.....	23
3.8.8	最短圆弧.....	24
3.8.9	二次方、三次方贝塞尔曲线.....	24
3.9	文字 .....	25
3.10	绘制图片.....	26
3.11	像素级操作.....	27
3.11.1	CREATEIMAGEDATA方法.....	28
3.11.2	GETIMAGEDATA方法.....	28
3.11.3	PUTIMAGEDATA方法.....	28
3.11.4	演示例子.....	28
3.12	绘图模型【此段翻译不怎么样，可以参看原英文】.....	29
4	参考资料 .....	29

## 摘要

本规范定义了二维 Canvas（画布）的绘画 API，使用这些 API 使得可以在 Web 页面上进行立即模式的二维图形绘制。

## 1 介绍

本规范描述了立即模式的 API 和为了在光栅风格的绘图区域内绘制 2 维矢量图形所必须的方法。其主要应用是 HTML5 规范定义的 canvas 元素。

### 1.1 术语：

2D：二维，你们懂的

3D：三维，你们懂的

API：编程接口

Canvas interface element：实现了本规范定义的绘图方法和属性的元素，简言之，就是“canvas”元素

Drawing context：绘图上下文，一个左上角为(0, 0)的笛卡尔坐标平面，在本平面中往右则 x 坐标增加和往下方 y 坐标增加

Immediate-mode：立即模式，一种绘图格式，当绘制完成后，所有的绘图结构将从内存中立即丢弃，本 API 即为此种图形绘制格式

Retained-mode: 残留模式: 另一种绘图格式, 当绘制完成后, 所有的绘图结构仍在内存中残留, 例如 DOM、SVG 即是此种绘制格式

Raster: 光栅风格, 图形的一种风格, 其由多行断开的图片 (行) 组成, 每行都包含确定的像素个数

Vector: 矢量, 你们懂的

source-over operator : 我不懂, 你们自己看吧

## 2 Canvas接口元素定义

DOM 接口:

```
interface CanvasElement : Element {
    attribute unsigned long width;
    attribute unsigned long height;

    Object getContext(in DOMString contextId);
    DOMString toDataURL(optional in DOMString type, in any...
args);

};
```

这里 width 和 height 必须是非负值, 并且无论什么时候重新设置了 width 或 height 的值, 画布中任何已绘对象都将被清除, 如下所示的 JS 代码中, 仅仅最后一个矩形被绘制:

```
// canvas is a reference to a <canvas> element
var context = canvas.getContext('2d');
context.fillRect(0,0,50,50);
canvas.setAttribute('width', '300'); // clears the canvas
context.fillRect(0,100,50,50);
canvas.width = canvas.width; // clears the canvas
context.fillRect(100,0,50,50); // only this square remains
```

### 2.1 getContext()方法

为了在 canvas 上绘制, 你必须先得到一个画布上下文对象的引用, 用本方法即可完成这一操作, 格式如下:

**context = canvas.getContext(contextId)**

方法返回一个指定 contextId 的上下文对象, 如果指定的 id 不被支持, 则返回 null, 当前唯一被强制必须支持的是 “2d”, 也许在将来会有 “3d”, 注意, 指定的 id 是大小写敏感的。

### 2.2 toDataURL()方法

此函数, 返回一张使用 canvas 绘制的图片, 返回值符合 data:URL 格式, 格式如下:

**url = canvas.toDataURL([ type, ... ])**

规范规定, 在未指定返回图片类型时, 返回的图片格式必须为 PNG 格式,

如果 canvas 没有任何像素，则返回值为：“data:”，这是最短的 data:URL，在 text/plain 资源中表现为空字符串。type 的可以在 image/png，image/jpeg, image/svg+xml 等 MIME 类型中选择。如果是 image/jpeg，可以有第二个参数，如果第二个参数的值在 0-1 之间，则表示 JPEG 的质量等级，否则使用浏览器内置默认质量等级。

下面的代码可以从 ID 为 codeex 的 canvas 中取得绘制内容，并作为 DataURL 传递给 img 元素，并显示。

```
var canvas = document.getElementById('codeex');
var url = canvas.toDataURL();

//id为myimg的图片元素
myimg.src = url;
```

### 3 二维绘图上下文

当使用一个 canvas 元素的 getContext(“2d”)方法时，返回的是 CanvasRenderingContext2D 对象，其内部表现为笛卡尔平面坐标，并且左上角坐标为(0,0)，在本平面中往右则 x 坐标增加和往下方 y 坐标增加。每一个 canvas 元素仅有一个上下文对象。其接口如下：

```
interface CanvasRenderingContext2D {

    // back-reference to the canvas
    readonly attribute HTMLCanvasElement canvas;

    // state
    void restore(); // pop state stack and restore state
    void save(); // push state on state stack

    // transformations (default transform is the identity matrix)
    void rotate(in float angle);
    void scale(in float x, in float y);
    void setTransform(in float m11, in float m12, in float m21, in float m22, in float dx, in float dy);
    void transform(in float m11, in float m12, in float m21, in float m22, in float dx, in float dy);
    void translate(in float x, in float y);

    // compositing
    attribute float globalAlpha; // (default 1.0)
    attribute DOMString globalCompositeOperation; // (default source-over)

    // colors and styles
    attribute any fillStyle; // (default black)
```

```
attribute any strokeStyle; // (default black)
CanvasGradient createLinearGradient(in float x0, in float y0, in
float x1, in float y1);
CanvasGradient createRadialGradient(in float x0, in float y0, in
float r0, in float x1, in float y1, in float r1);
CanvasPattern createPattern(in HTMLImageElement image, in
DOMString repetition);
CanvasPattern createPattern(in HTMLCanvasElement image, in
DOMString repetition);
CanvasPattern createPattern(in HTMLVideoElement image, in
DOMString repetition);

// line styles
attribute DOMString lineCap; // "butt", "round", "square"
(default "butt")
attribute DOMString lineJoin; // "miter", "round", "bevel"
(default "miter")
attribute float lineWidth; // (default 1)
attribute float miterLimit; // (default 10)

// shadows
attribute float shadowBlur; // (default 0)
attribute DOMString shadowColor; // (default transparent black)
attribute float shadowOffsetX; // (default 0)
attribute float shadowOffsetY; // (default 0)

// rects
void clearRect(in float x, in float y, in float w, in float h);
void fillRect(in float x, in float y, in float w, in float h);
void strokeRect(in float x, in float y, in float w, in float h);

// Complex shapes (paths) API
void arc(in float x, in float y, in float radius, in float
startAngle, in float endAngle, in boolean anticlockwise);
void arcTo(in float x1, in float y1, in float x2, in float y2,
in float radius);
void beginPath();
void bezierCurveTo(in float cp1x, in float cp1y, in float cp2x,
in float cp2y, in float x, in float y);
void clip();
void closePath();
void fill();
void lineTo(in float x, in float y);
void moveTo(in float x, in float y);
```

```
void quadraticCurveTo(in float cpx, in float cpy, in float x, in float y);
void rect(in float x, in float y, in float w, in float h);
void stroke();
boolean isPointInPath(in float x, in float y);

// text
attribute DOMString font; // (default 10px sans-serif)
attribute DOMString textAlign; // "start", "end", "left", "right", "center" (default: "start")
attribute DOMString textBaseline; // "top", "hanging", "middle", "alphabetic", "ideographic", "bottom" (default: "alphabetic")
void fillText(in DOMString text, in float x, in float y, optional in float maxWidth);
TextMetrics measureText(in DOMString text);
void strokeText(in DOMString text, in float x, in float y, optional in float maxWidth);

// drawing images
void drawImage(in HTMLImageElement image, in float dx, in float dy, optional in float dw, in float dh);
void drawImage(in HTMLImageElement image, in float sx, in float sy, in float sw, in float sh, in float dx, in float dy, in float dw, in float dh);
void drawImage(in HTMLCanvasElement image, in float dx, in float dy, optional in float dw, in float dh);
void drawImage(in HTMLCanvasElement image, in float sx, in float sy, in float sw, in float sh, in float dx, in float dy, in float dw, in float dh);
void drawImage(in HTMLVideoElement image, in float dx, in float dy, optional in float dw, in float dh);
void drawImage(in HTMLVideoElement image, in float sx, in float sy, in float sw, in float sh, in float dx, in float dy, in float dw, in float dh);

// pixel manipulation
ImageData createImageData(in float sw, in float sh);
ImageData createImageData(in ImageData imagedata);
ImageData getImageData(in float sx, in float sy, in float sw, in float sh);
void putImageData(in ImageData imagedata, in float dx, in float dy, optional in float dirtyX, in float dirtyY, in float dirtyWidth, in float dirtyHeight);
};
```

```
interface CanvasGradient {
    // opaque object
    void addColorStop(in float offset, in DOMString color);
};

interface CanvasPattern {
    // opaque object
};

interface TextMetrics {
    readonly attribute float width;
};

interface ImageData {
    readonly attribute CanvasPixelArray data;
    readonly attribute unsigned long height;
    readonly attribute unsigned long width;
};

interface CanvasPixelArray {
    readonly attribute unsigned long length;
    getter octet (in unsigned long index);
    setter void (in unsigned long index, in octet value);
};
```

### 3.1 canvas的状态

每个上下文都包含一个绘图状态的堆，绘图状态包含下列内容：

- ✧ 当前的 *transformation matrix*.
- ✧ 当前的 *clipping region*
- ✧ 当前的属性值: *fillStyle*, *font*, *globalAlpha*,  
*globalCompositeOperation*, *lineCap*, *lineJoin*,  
*lineWidth*, *miterLimit*, *shadowBlur*, *shadowColor*,  
*shadowOffsetX*, *shadowOffsetY*, *strokeStyle*, *textAlign*,  
*textBaseline*

注：当前 *path* 和当前 *bitmap* 不是绘图状态的一部分，当前 *path* 是持久存在的，仅能被 *beginPath()* 复位，当前 *bitmap* 是 *canvas* 的属性，而非绘图上下文。

**`context.restore()`** //弹出堆最上面保存的绘图状态

**context.save()** //在绘图状态堆上保存当前绘图状态

绘图状态可以看作当前画面应用的所有样式和变形的一个快照。而状态的应用则可以避免绘图代码的过度膨胀。

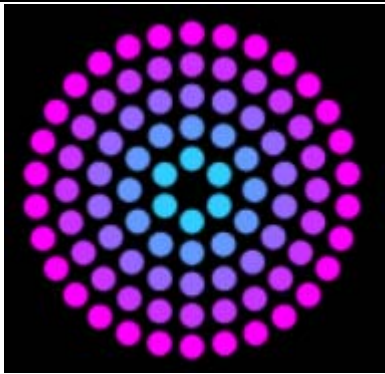
**3.2 转换(Transformations)**

当建立形状和路径时，转换矩阵被应用到其坐标上。转换的执行顺序是严格按顺序的(注：原文是反向，经试验应该是按调用顺序的)。

在做转换/变形之前先保存状态是一个良好的习惯。大多数情况下，调用 restore 方法比手动恢复原先的状态要简单得多。又，如果你是在一个循环中做位移但没有保存和恢复 canvas 的状态，很可能到最后会发现怎么有些东西不见了，那是因为它很可能已经超出 canvas 范围以外了。

**context.rotate(angle)** //按给定的弧度旋转,按顺时针旋转

rotate 方法旋转的中心始终是 canvas 的原点，如果要改变它，需要使用 translate 方法。



```
context.translate(75, 75);
for (var i=1; i<6; i++) {
    context.save();
    context.fillStyle =
'rgb('+(51*i)+' ,'+(255-51*i)+' ,255)';

    for (var j=0; j<i*6; j++) {
        context.rotate(Math.PI*2/(i*6));
        context.beginPath();

        context.arc(0, i*12.5, 5, 0, Math.PI*2, true)
        ;
        context.fill();
    }

    context.restore();
}
```

**context.scale(x, y)** //按给定的缩放倍率缩放，1.0,为不变

参数比 1.0 小表示缩小，否则表示放大。默认情况下，canvas 的 1 单位就是 1 个像素。举例说，如果我们设置缩放因子是 0.5，1 个单位就变成对应 0.5 个像素，这样绘制出来的形状就会是原先的一半。同理，设置为 2.0 时，1 个单位就对应变成了 2 像素，绘制的结果就是图形放大了 2 倍。

**context.setTransform(m11, m12, m21, m22, dx, dy)** //



重设当前的转换到

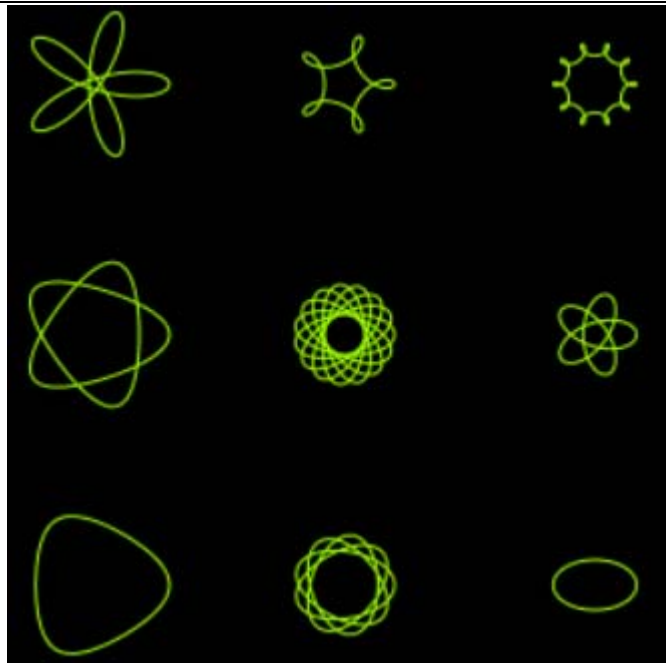
**context.transform(m11, m12, m21, m22, dx, dy)** //矩阵变换，结果等于当前的变形矩阵乘上

$$\begin{matrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{matrix}$$

后的结果

**context.translate(x, y)** //可以理解为偏移，向 x,y 方向偏移指定的量，其用来移动 Canvas 的原点到一个指定的值

下面是一个利用 translate 方法进行绘制螺旋图案的例子：



// 绘制螺旋图案的函数

```
function drawSpirograph(ctx,R,r,O){
  var x1 = R-O;
  var y1 = 0;
  var i = 1;
  ctx.beginPath();
  ctx.moveTo(x1,y1);
  do {
    if (i>20000) break;
    var x2 = (R+r)*Math.cos(i*Math.PI/72) -
(r+O)*Math.cos(((R+r)/r)*(i*Math.PI/72))
    var y2 = (R+r)*Math.sin(i*Math.PI/72) -
(r+O)*Math.sin(((R+r)/r)*(i*Math.PI/72))
```

```

    ctx.lineTo(x2,y2);
    x1 = x2;
    y1 = y2;
    i++;
  } while (x2 != R-O && y2 != 0 );
  ctx.stroke();
}

// 调用部分代码

context.fillRect(0,0,300,300);
for (var i=0;i<3;i++) {
  for (var j=0;j<3;j++) {
    context.save();
    context.strokeStyle = "#9CFF00";
    context.translate(50+j*100,50+i*100);

    drawSpirograph(context,20*(j+2)/(j+1),-8*(i+3)/(i+1),1
0);

    context.restore();
  }
}

```

### 3.3 合成 (Compositing)









在默认情况下，我们总是将一个图形画在另一个之上，但在特殊情况下，这样是不够的。比如说，它这样受制于图形的绘制顺序。不过，我们可以利用 `globalCompositeOperation` 属性来改变这些做法



**`context.globalAlpha [= value]`** //0-1.0 之间的数据，设定图像的透明度

**`context.globalCompositeOperation [= value]`** //设定重叠图像的覆盖方式，可以设定为(注，值大小写敏感)：

注意：下面所有图例中，B(蓝色方块)是先绘制的，即“已有的 canvas 内容”，A(红色圆形)是后面绘制，即“新图形”。

<b><i>source-over</i></b> <b>(default)</b>		A over B. 这是默认设置，新图形会覆盖在原有内容之上。
<b><i>destination-over</i></b>		B over A. 会在原有内容之下绘制新图形。

<b><i>source-atop</i></b>		<i>A atop B</i> . 新图形中与原有内容重叠的部分会被绘制, 并覆盖于原有内容之上。
<b><i>destination-atop</i></b>		<i>B atop A</i> . 原有内容中与新内容重叠的部分会被保留, 并会在原有内容之下绘制新图形
<b><i>source-in</i></b>		<i>A in B</i> . 新图形会仅仅出现与原有内容重叠的部分。其它区域都变成透明的。
<b><i>destination-in</i></b>		<i>B in A</i> . 原有内容中与新图形重叠的部分会被保留, 其它区域都变成透明的
<b><i>source-out</i></b>		<i>A out B</i> . 结果是只有新图形中与原有内容不重叠的部分会被绘制出来。
<b><i>destination-out</i></b>		<i>B out A</i> . 原有内容中与新图形不重叠的部分会被保留。
<b><i>lighter</i></b>		<i>A plus B</i> . 两图形中重叠部分作加色处理。
<b><i>darker</i></b>		两图形中重叠的部分作减色处理。 <i>* 标准中没有暂无此项</i>

<b><i>copy</i></b>		<i>A (B is ignored)</i> . 只有新图形会被保留，其它都被清除掉。
<b><i>xor</i></b>		<i>A xor B</i> . 重叠的部分会变成透明。
<b><i>vendorName-operationName</i></b>		Vendor-specific extensions to the list of composition operators should use this syntax

### 3.4 颜色和风格

***context.fillStyle [= value]*** //返回填充形状的当前风格，能被设置用来改变当前的填充风格，其值可以是CSS颜色字串，也可以是CanvasGradient或者CanvasPattern对象，非法的值将被忽略。

***context.strokeStyle [= value]*** //返回当前描绘形状的风格，能被设置，其值同上。

设置 Javascript 例子如下：

```
context.strokeStyle="#99cc33";
context.fillStyle='rgba(50,0,0,0.7)';
context.lineWidth=10;
context.fillRect(20,20,100,100);
context.strokeRect(20,20,100,100);
```

绘制的图形如下所示。



上面提到的还有CanvasGradient对象，规范规定有两类渐变类型，线性渐变和径向渐变。

***gradient.addColorStop(offset, color)*** //在给定偏移的地方增加一个渐变颜色点，偏移量取值范围为0-1.0之间，否则产生一个INDEX\_SIZE\_ERR的异常，color为DOM字符串，如果不能解析，则抛出一个SYNTAX\_ERR的异常

***gradient = context.createLinearGradient(x0, y0,***

**`x1, y1`** //建立一个线性渐变，如果任何一个参数不是有限值，则抛出一个 **NOT\_SUPPORTED\_ERR** 的异常。

**`gradient = context.createRadialGradient(x0, y0, r0, x1, y1, r1)`** //建立一个径向渐变，如果任何一个参数不是有限值，则抛出一个 **NOT\_SUPPORTED\_ERR** 的异常。假如 **`r0`** 或 **`r1`** 为负值，则抛出 **INDEX\_SIZE\_ERR** 的异常。

设置 Javascript 例子如下：

```
var gradient =
    context.createLinearGradient(0,2,420,2);
gradient.addColorStop(0,'rgba(200,0,0,0.8)');
gradient.addColorStop(0.5,'rgba(0,200,0,0.7)');
gradient.addColorStop(1,'rgba(200,0,200,0.9)');
context.strokeStyle="#99cc33";
context.fillStyle= gradient;//copyright codeex.cn
context.lineWidth=10;
context.fillRect(20,20,400,100);
context.strokeRect(20,20,400,100);
```

绘制的图形如下所示。



更改 `var gradient = context.createLinearGradient(0, 2, 420, 2);` 为 `var gradient = context.createLinearGradient(0, 2, 420, 200);` 则绘制图形如下：



细心的读者会发现，此方法与 PHOTOSHOP 软件中的渐变工具类似。

注：如果 `X0=X1, Y0=Y1`，则绘制动作什么也不做，自己改改例子试试吧。

`createRadialGradient(x0, y0, r0, x1, y1, r1)` 方法有六个参数，前三个参数表示开始的圆，其圆点在 `(x0, y0)`，半径为 `r0`，后三个表示结束的圆，参数意义同上。其绘制过程如下：

1. 如果起始圆和结束圆重叠，则不绘制任何东西，并终止步骤；
2.  $x(w) = (x1-x0)w + x0$   
 $y(w) = (y1-y0)w + y0$   
 $r(w) = (r1-r0)w + r0$   
 在以  $(x(w), y(w))$  为圆点， $r(w)$  为半径的圆周上所有点的颜色均为  $Color(w)$ 。
3. 对于任意的  $w$  取值  $(-\infty \rightarrow +\infty)$ ，确保  $r(w) > 0$ ，总是可以知道画布中已知点的颜色，

简而言之，言而总之：这个效果就是建立一个圆锥体（手电筒效果）渲染效果，圆锥体的开始圆使用开始颜色偏移量为 0，圆锥体的结束圆使用颜色偏移量为 1.0，面积外的颜色均使用透明黑。

设置 Javascript 例子如下：

```
var gradient =
    context.createRadialGradient(100,100,20,300,300,80);
gradient.addColorStop(0,'rgba(200,0,0,0.8)');
gradient.addColorStop(1,'rgba(200,0,200,0.9)');
context.strokeStyle="#99cc33";
context.fillStyle= gradient;//'rgba(50,0,0,0.7)';
context.lineWidth=10;
context.fillRect(10,10,400,400);
context.strokeRect(10,10,400,400);
```

绘制的图形如下所示。



上面提到可以作为渲染风格还有图案对象:CanvasPattern, 其调用格式如下：

***pattern = context.createPattern(image, repetition)***

本方法用指定的图像和重复方向建立一个画布图案对象，image 参数可以为 img, canvas, video 元素中的任一个，如果不满足此条件，则抛出

TYPE\_MISMATCH\_ERR 异常，如果图片编码未知或没有图像数据，则抛出 INVALID\_STATE\_ERR 异常；第二个参数可以是下列值：

repeat	默认参数，如果为空，则为此参数，表示两个方向重复
repeat-x	仅水平重复
repeat-y	仅垂直重复
no-repeat	不重复

如果 image 参数是一个 HTMLImageElement 对象，但对象的 complete 属性是 false，则执行时抛出 INVALID\_STATE\_ERR 异常；

如果 image 参数是一个 HTMLVideoElement 对象，但其 readyState 属性是 HAVE\_NOTHING 或 HAVE\_METADATA，则执行时抛出 INVALID\_STATE\_ERR 异常；

如果 image 参数是一个 HTMLCanvasElement 对象，但其 width 属性或 height 属性是 0，则执行时抛出 INVALID\_STATE\_ERR 异常。

图案的绘制时从左上角开始的，根据不同的参数进行重复绘制。如果传递的图片是动画，则选取海报或第一帧作为其绘制图案源，如果使用 HTMLVideoElement 为对象，则当前播放位置帧被作为图案源。

设置 HTML 的核心代码如下：

```
</img>
<button onClick="drawCanvas()">绘制图形</button>
<button onClick="Show()">显示图形</button>
<div>
<canvas id="myCanvas" width="500" height="500" style="border:1px
dotted #000">your browser does not support the canvas tag
</canvas>
</div>
```

设置 Javascript 例子如下：

```
var imgSrc = document.getElementById('psrc')
var pattern = context.createPattern(imgSrc,'repeat');
context.strokeStyle="#99cc33";
context.fillStyle= pattern;//by codeex.cn
context.lineWidth=10;
context.fillRect(10,10,200,220);
context.strokeRect(10,10,200,220);
```

在 IE9 中的显示效果如图所示：



repeat 效果



Repeat-x 效果



Repeat-y 效果

3.5 线风格

操作线风格的方法有 4 个，格式如下：

**context.lineCap [= value]** //返回或设置线段的箭头样式，仅有三个选项：**butt**(默认值),**round**,**square**;其他值忽略

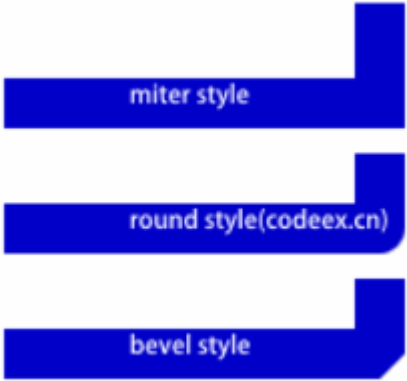
butt	每根线的头和尾都是长方形，也就是不做任何的处理，为默认值	 <p>三种风格的比较图，JS 代码如下</p>
round	每根线的头和尾都增加一个半圆形的箭头	
square	每根线的头和尾都增加一个长方形，长度为线宽一半，高度为线宽	

```
context.beginPath();
context.lineCap='butt';
context.moveTo(100,50);context.lineTo(250,50);
context.stroke();
context.beginPath();
context.lineCap='round';
context.moveTo(100,80); ;context.lineTo(250,80);
context.stroke();
context.beginPath();
context.lineCap='square';
context.moveTo(100,110);context.lineTo(250,110);
```



```
context.stroke();
```

**context.lineJoin [= value]** //返回或设置线段的连接方式，仅有三个选项：miter(默认值),round, bevel; 其他值忽略

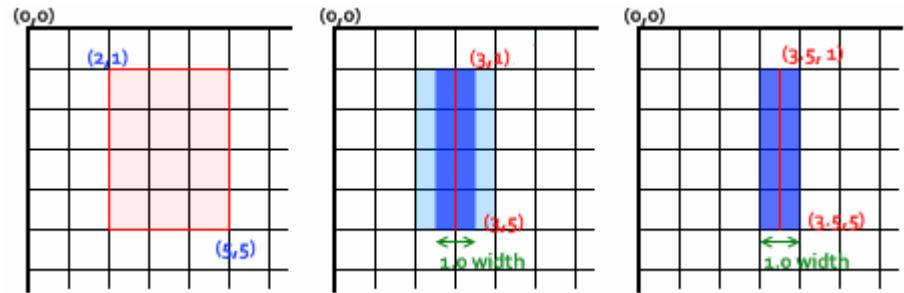
miter	线段在连接处外侧延伸直至交于一点，为默认值，外延效果受miterLimit 的值影响，当外延交点距离大于限制值时，则表现为bevel 风格	
round	连接处是一个圆角,圆的半径等于线宽	
bevel	连接处为斜角	

**context.lineWidth [= value]** //返回或设置线段的线宽，非大于 0 的值被忽略；默认值为 1.0;

**context.miterLimit [= value]** //返回或设置线段的连接处的斜率，非大于 0 的值被忽略；默认值为 10.0。本属性翻译不够准确，请参看英文部分

线宽是指给定路径的中心到两边的粗细。换句话说就是在路径的两边各绘制线宽的一半。因为画布的坐标并不和像素直接对应，当需要获得精确的水平或垂直线的时候要特别注意。

想要获得精确的线条，必须对线条是如何描绘出来的有所理解。见下图，用网格来代表 canvas 的坐标格，每一格对应屏幕上一个像素点。在第一个图中，填充了 (2, 1) 至 (5, 5) 的矩形，整个区域的边界刚好落在像素边缘上，这样就可以得到的矩形有着清晰的边缘。



如果你想要绘制一条从 (3, 1) 到 (3, 5)，宽度是 1.0 的线条，你会得到像第二幅图一样的结果。实际填充区域（深蓝色部分）仅仅延伸至路径两旁各一半像素。而这半个像素又会以近似的方式进行渲染，这意味着那些像素只是部分着色，结果就是以实际笔触颜色一半色调的颜色来填充整个区域

(浅蓝和深蓝的部分)。

要解决这个问题，你必须对路径施以更加精确的控制。已知粗 1.0 的线条会在路径两边各延伸半像素，那么像第三幅图那样绘制从 (3.5, 1) 到 (3.5, 5) 的线条，其边缘正好落在像素边界，填充出来就是准确的宽为 1.0 的线条。

对于那些宽度为偶数的线条，每一边的像素数都是整数，那么你想要其路径是落在像素点之间（如那从 (3, 1) 到 (3, 5)）而不是在像素点的中间。如果不是的话，端点上同样会出现半渲染的像素点。

### 3.6 阴影(Shadows)

有关阴影的四个全局属性将影响所有的绘画操作。有关定义如下：

**`context.shadowBlur [= value]`** //返回或设置阴影模糊等级，非大于等于 0 的值被忽略；

**`context.shadowColor [= value]`** //返回或设置阴影颜色

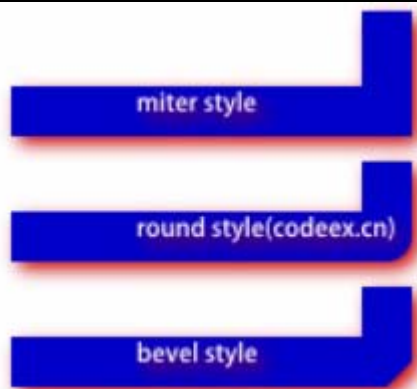
**`context.shadowOffsetX [= value]`**

**`context.shadowOffsetY [= value]`** //返回或设置阴影的偏移量

注意：上面的值均不受坐标转换的影响，可以看做是绝对值。

在上面的例子中增加下列语句，可以得到设置阴影的图像：

```
context.shadowBlur=7;
context.shadowColor='rgb(200,0,0)';
context.shadowOffsetX = 3;
context.shadowOffsetY=3;
```



### 3.7 简单形状（矩形）

形状的绘制不影响当前路径(path)，形状是剪切区域的主题，也是阴影(Shadow)效果，全局透明(alpha)，全局组合(composition)操作等的主题。其由下面三个方法来进行简单的操作：

**`context.clearRect(x, y, w, h)`** //在给定的矩形内清除所有的像素为透明黑(transparent black)

**`context.fillRect(x, y, w, h)`** //用当前的填充风格填充给定的区域

**`context.strokeRect(x, y, w, h)`** //使用当前给定的线风格，绘制一个盒子区域，影响其绘制风格的有：**`strokeStyle`**, **`lineWidth`**, **`lineJoin`**, **`miterLimit`** (可能)。

### 3.8 复杂形状(路径-paths)

绘图上下文总有一个当前路径，并且是仅此一个，它不是绘图状态的一部分。

一个路径有 0 个或多个子路径列表。每个子路径包含一个或多个点列表（这些点组成直的或弯曲的线段），和一个标识子路径是否闭合的标志。少于两个点的子路径在绘图时被忽略。操作这些形状的方法稍微多些，如下所示：

默认情况下，图形上下文的路径有 0 个子路径。

#### 3.8.1 路径起始函数

调用格式：

**`context.beginPath()`** //清空子路径

**`context.closePath()`** //闭合路径

方法概述：

`beginPath` 方法重设绘图上下文的子路径列表，并清空所有的子路径。

`closePath` 方法在绘图上下文如果没有子路径时，什么也不做；否则，它先把最后一个子路径标示为闭合，然后建立一个包含最后子路径的第一个点的子路径，并加入到绘图上下文。有点拗口，其一般可以看作，假如最后一个子路径，我们命名为 `spN`，假设 `spN` 有多个点，则用直线连接 `spN` 的最后一个点和第一个点，然后关闭此路径和 `moveTo` 到第一个点。

#### 3.8.2 绘制函数

调用格式：

**`context.stroke()`**

**`context.fill()`**

**`context.clip()`**

方法概述：

`stroke`方法使用`lineWidth`, `lineCap`, `lineJoin`, 以及`strokeStyle`对所有的子路径进行填充。

`fill`方法使用`fillStyle`方式填充子路径，未闭合的子路径在填充式按照

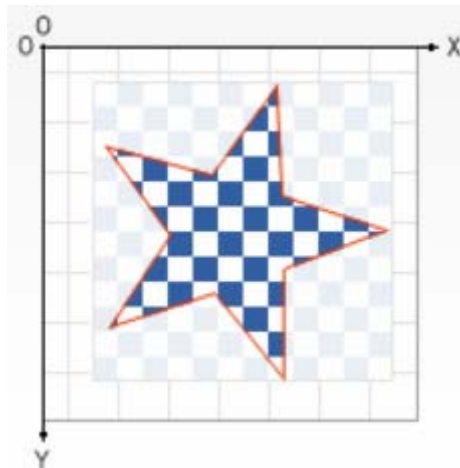
闭合方式填充，但并不影响实际的子路径集合。

`clip` 方法使用计算所有的子路径而建立新的剪切区域，未闭合的子路径在填充式按照闭合方式填充，但并不影响实际的子路径集合，新的剪切区域将替换当前的剪切区域。

对于剪切，也许需要一个小节才能说明白，这里我摘抄来自 mozilla 的教程来阐述剪切的应用。

### 3.8.3 剪切(clip)

裁切路径和普通的 canvas 图形差不多，不同的是它的作用是遮罩，用来隐藏没有遮罩的部分。如下图所示。红边五角星就是裁切路径，所有在路径以外的部分都不会在 canvas 上绘制出来。



如果和上面介绍的 `globalCompositeOperation` 属性作一比较，它可以实现与 `source-in` 和 `source-atop` 差不多的效果。最重要的区别是裁切路径不会在 canvas 上绘制东西，而且它永远不受新图形的影响。这些特性使得它在特定区域里绘制图形时相当好用。

我们用 `clip` 方法来创建一个新的裁切路径。默认情况下，canvas 有一个与它自身一样大的裁切路径（也就是没有裁切效果）。

下面展示一个随机星星的例子，并且我们会用 `clip` 来限制星星出现的区域。



首先，我画了一个与 canvas 一样大小的黑色方形作为背景，然后移动原点至中心点。然后用 `clip` 方法创建一个弧形的裁切路径。裁切路径也属于 canvas 状态的一部分，可以被保存起来。如果我们在创建新裁切路径时想保留原来的裁切路径，我们需要做的就是保存一下 canvas 的状态。

裁切路径创建之后所有出现在它里面的东西才会画出来。在画线性渐变时这个就更加明显了。然后在随机位置绘制 50 大小不一（经过缩放）的颗，

当然也只有在裁切路径里面的星星才会绘制出来。

代码如下：

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.fillRect(0, 0, 150, 150);
  ctx.translate(75, 75);

  // Create a circular clipping path
  ctx.beginPath();
  ctx.arc(0, 0, 60, 0, Math.PI*2, true);
  ctx.clip();

  // draw background
  var lingrad = ctx.createLinearGradient(0, -75, 0, 75);
  lingrad.addColorStop(0, '#232256');
  lingrad.addColorStop(1, '#143778');

  ctx.fillStyle = lingrad;
  ctx.fillRect(-75, -75, 150, 150);

  // draw stars
  for (var j=1; j<50; j++) {
    ctx.save();
    ctx.fillStyle = 'fff';
    ctx.translate(75-Math.floor(Math.random()*150),
      75-Math.floor(Math.random()*150));
    drawStar(ctx, Math.floor(Math.random()*4)+2);
    ctx.restore();
  }
}

function drawStar(ctx, r) {
  ctx.save();
  ctx.beginPath();
  ctx.moveTo(r, 0);
  for (var i=0; i<9; i++) {
    ctx.rotate(Math.PI/5);
    if(i%2 == 0) {
      ctx.lineTo((r/0.525731)*0.200811, 0);
    } else {
      ctx.lineTo(r, 0);
    }
  }
  ctx.closePath();
```

```
ctx.fill();  
ctx.restore();  
}
```

#### 3.8.4 辅助方法一点是否在路径里

调用格式:

***context . isPointInPath(x, y)***

方法概述:

给定的坐标 (x, y) 是否在当前路径中, 坐标 (x, y) 为绘图坐标系坐标, 并不受转换的影响。

#### 3.8.5 moveTo方法

调用格式:

***context . moveTo(x, y)***

方法概述:

建立新的子路径, 并制定其第一个点为 (x, y)。

#### 3.8.6 lineTo方法

调用格式:

***context . lineTo(x, y)***

方法概述:

如果绘图上下文没有子路径, 则其等同于 moveTo(x, y), 否则, 其建立一条在子路径最后一个点到给定点的直线, 并增加 (x, y) 到子路径中。

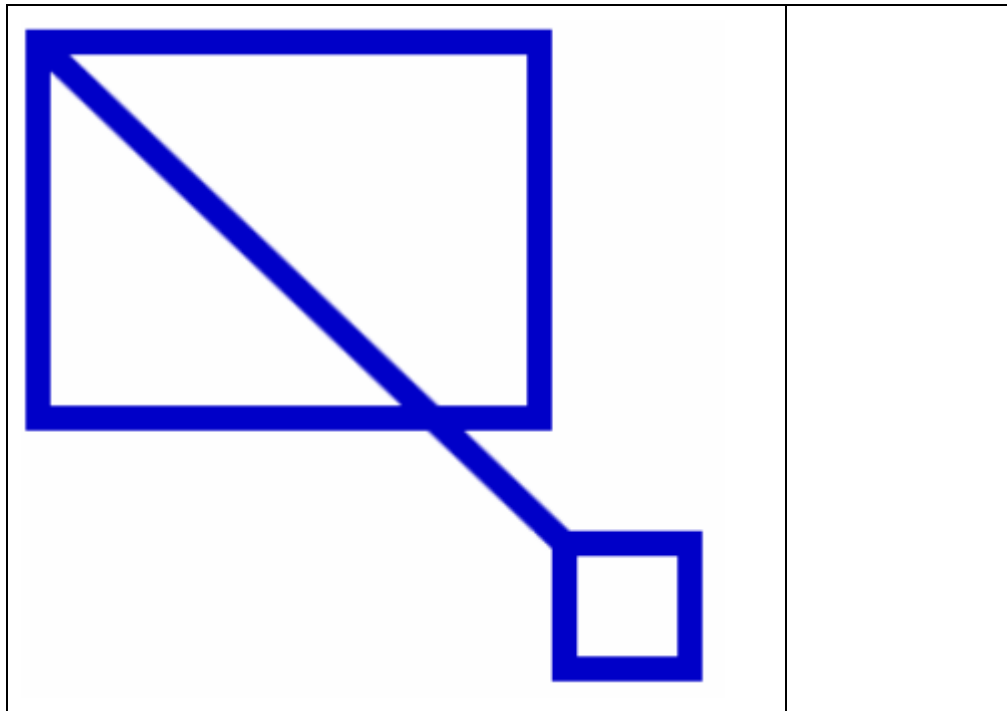
#### 3.8.7 rect方法

调用格式:

***context . rect(x, y, w, h)***

方法概述:

本方法建立二个新的子路径, 第一个子路径包含四个点: (x, y), (x+w, y), (x+w, y+h), (x, y+h), 四个点的连接方式为直线, 该子路径被标示为闭合路径; 最后再增加一个子路径, 其仅有一个点 (x, y)。

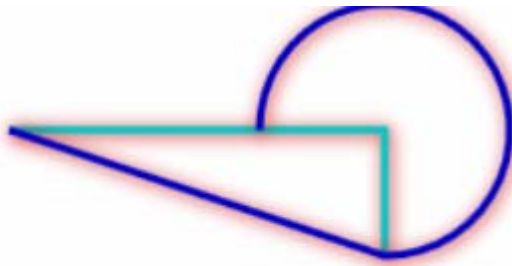


### 3.8.8 圆弧

方法调用格式:

```
context . arc(x, y, radius, startAngle, endAngle,  
anticlockwise)
```

**方法概述：** 本方法先增加一条直线到子路径列表，然后增加一个圆弧子路径到子路径列表。直线子路径是由上一个点到圆弧子路径的起始点，而圆弧则为按照给定的开始角度、结束角度和半径描述的按照给定的方向[布尔类型，anticlockwise-逆时针（true）]圆弧上；假如半径为负值，抛出INDEX SIZE ERR的异常；



蓝线表示的即是此函数绘制的结果，最左边的点为上一个路径的最后一个点

JS 代码:

```
context.beginPath();
context.moveTo(100, 50);
context.arc(250, 50, 50, 1.5708, 3.14, true);
context.stroke();
```

注释掉 `moveTo` 语句, 则仅仅绘制圆弧:



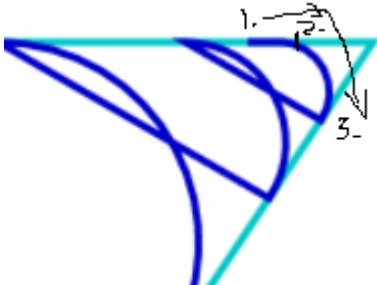
### 3.8.9 最短圆弧

方法调用格式:

**`context.arcTo(x1, y1, x2, y2, radius)`**

方法概述: 本方法绘制出子路径最后一个点 (x0, y0) 和 (x1, y1) 以及 (x1, y1) 和 (x2, y2) 构成的两条直线间半径为 radius 的最短弧线, 并用直线连接 (x0, y0); 假如半径为负值, 抛出 INDEX\_SIZE\_ERR 的异常;

如图所示, 绘制曲线由 1, 开始绘制。



JS 代码如下:

```
context.beginPath();
context.moveTo(150, 50);
context.arcTo(200, 50, 100, 200, 20);
context.arcTo(200, 50, 100, 200, 40);
context.arcTo(200, 50, 100, 200, 80);
context.arcTo(200, 50, 100, 200, 120);
context.arcTo(200, 50, 100, 200, 160);
context.stroke();
```

### 3.8.10 二次方、三次方贝塞尔曲线

贝塞尔曲线的一般概念: 在数学的数值分析领域中, 贝赛尔曲线 (Bezier curve) 是电脑图形学中相当重要的参数曲线。更高维度的贝赛尔曲线被称作贝塞尔曲面。对于 n 阶贝塞尔曲线可如下推断, 给定  $P_0$ 、 $P_1$ 、 $P_2 \cdots P_n$ , 其贝赛尔曲线即为

$$B(t) = \sum_{i=0}^n \binom{n}{i} P_i (1-t)^{n-i} t^i = P_0 (1-t)^n + \binom{n}{1} P_1 (1-t)^{n-1} t + \dots + P_n t^n, \quad t \in [0, 1]$$

用平常话说, n 阶的贝赛尔曲线就是双 n-1 阶贝赛尔曲线之间的插值。

由公式可以得出二次方贝塞尔曲线公式如下:

$$B(t) = P_0 (1-t)^2 + 2tP_1(1-t) + P_2 t^2, \quad t \in [0, 1]$$



TrueType 字型就运用了以贝塞尔样条组成的二次方贝塞尔曲线。

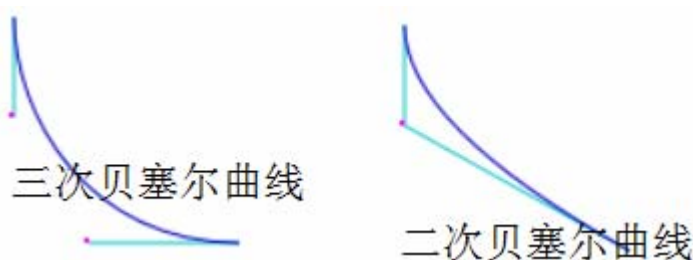
方法调用格式：

***bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)***

***quadraticCurveTo(cpx, cpy, x, y)***

方法概述：上面分别是三次贝塞尔曲线和二次贝塞尔曲线的调用格式。其主要区别在于控制曲线的控制点式不一样的。其起始点均为子路径的最后一个点，结束点均为(x, y)；在最后均要把(x, y)点加入到子路径中。

其绘制图形的例子如下，三次贝塞尔曲线有两个红点作为曲线平滑的控制点，而二次贝塞尔曲线仅有一个控制点。



JS 代码如下：

```
context.moveTo(10, 10);
context.quadraticCurveTo(10, 50, 100, 100);
context.stroke();
```

### 3.9 文字

绘图上下文提供了得到和设置文字样式的接口方法，这里将一一介绍给大家。

获得和设置文字设置： context.font[=value]，可以参考 CSS 中对 font 风格的设置。

IE9 对中文的支持好像没有添加，我么有试出来。

获取或设置文字对齐方式： context.textAlign[=value]，取值如下：

start	默认值，与 canvas 风格中的 direction 定义有关
end	与 canvas 风格中的 direction 定义有关
left	左
right	右
center	居中

获得和设置文字对齐基线： context.textBaseline[=value]，value 的取值如下：

alphabetic	默认值，alphabetic 基线
top	文字矩形的顶部
bottom	文字矩形的底部
middle	文字矩形的中间
hanging	Hanging 基线
ideographic	Ideographic 基线

绘制文字的方法：`context.fillText(text, x, y[, maxWidth])`  
`context.strokeText(text, x, y[, maxWidth])`

前一个方法为绘制填充的文字，后一个方法为对文字进行描边，不填充内部区域。

按照当前字体对给定的文字进行测量：

`metrics = context.measureText(text)`，该方法返回一个 `TextMetrics` 对象，可以调用对象的 `width` 属性得到文字的宽度。

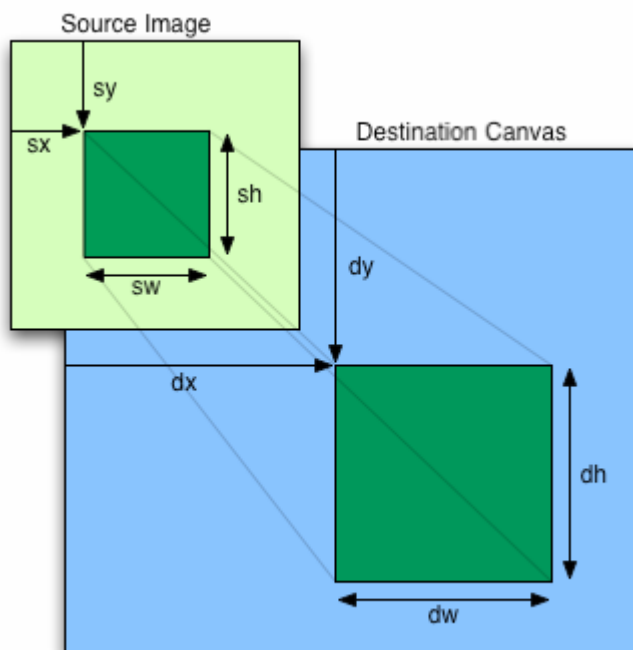
### 3.10 绘制图片

要在绘图上下文中绘制图片，可以使用 `drawImage` 方法。该方法有三种不同的参数：

- ✧ `drawImage(image, dx, dy)`
- ✧ `drawImage(image, dx, dy, dw, dh)`
- ✧ `drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)`

其中的 `image` 参数可以是 `HTMLImageElement`、`HTMLCanvasElement` 和 `HTMLVideoElement` 中的任一个对象。

绘制参数的含义可以参看下图：



异常：如果第一个参数不是指定的元素类型，抛出一个 `TYPE_MISMATCH_ERR` 异常，如果图片不能进行解码，则抛出 `INVALID_STATE_ERR` 异常，如果第二个参数不是允许的值，则抛出 `SYNTAX_ERR` 异常。

参数默认值：如果没有指定 `dw` 和 `dh`，则默认等于 `sw` 和 `sh`，如果 `sx`, `sy`, `sw`, `sh` 均没有提供，则默认为 `sx, sy=0, 0`；`sw` 和 `sh` 为图片的像素宽高。

下面给出图片的几种调用方式：

#### 1. 引用页面内图片：

我们可以通过 `document.images` 集合、`document.getElementsByTagName` 方法又或者 `document.getElementById` 方法来获取页面内的图片（如果已知图片元素的 ID）。

## 2. 引用 canvas 元素

和引用页面内的图片类似地，用 `document.getElementsByTagName` 或 `document.getElementById` 方法来获取其它 `canvas` 元素。但你引入的应该是已经准备好的 `canvas`。一个常用的应用就是为另一个大的 `canvas` 做缩略图。

### 3. 创建图像

我们可以用脚本创建一个新的 Image 对象，但这种方法的主要缺点是如果不希望脚本因为等待图片装置而暂停，还得需要突破预装载。

```
var img = new Image(); // Create new Image
img.src = 'myImage.png'; // Set source path
```

当脚本执行后，图片开始装载。若调用 `drawImage` 时，图片没装载完，脚本会等待直至装载完毕。如果不希望这样，可以使用 `onload` 事件：

```
var img = new Image();           // Create new Image
img.onload = function() {
    // 执行 drawImage 语句
}
```

```
img.src = 'myImage.png'; // Set source path
```

如果你只用到一张图片的话，这已经够了。但一旦需要不止一张图片，那就需要更加复杂的处理方法，但图片预装载策略超出本规范的范围。

#### 4. 通过 data: url 方式嵌入图像

我们还可以通过 `data: url` 方式来引用图像。Data urls 允许用一串 Base64 编码的字符串的方式来定义一个图片。其优点就是图片内容即时可用，无须再到服务器兜一圈。（还有一个优点是，可以将 CSS, JavaScript, HTML 和 图片全部封装在一起，迁移起来十分方便。）缺点就是图像没法缓存，图片大的话内嵌的 url 数据会相当的长，例如：

[illegible]

有兴趣的朋友可以使用<img src=' 上述变量值'>的方法显示出上面的图片。

### 3.11 像素级操作

2D Context API 提供了三个方法用于像素级操作: `createImageData`, `getImageData`, 和 `putImageData`。

ImageData 对象保存了图像像素值。每个对象有三个属性：width, height 和 data。data 属性类型为 CanvasPixelArray，用于储存 width\*height\*4 个像素值。每一个像素有 RGB 值和透明度 alpha 值(其值为 0 至 255，包括 alpha 在内)。像素的顺序从左至右，从上到下，按行存储。

#### 3.11.1 createImageData 方法

```
imagedata = context . createImageData(sw, sh)
```

```
imagedata = context . createImageData(imagedata)
```

方法概述：createImageData 方法根据给定的 CSS 像素宽高或指定的 imagedata 具有的宽高建立一个 ImageData 对象，该对象为透明黑。该方法具体实例化一个新的空 ImageData 对象。

#### 3.11.2 getImageData 方法

```
imagedata = context . getImageData(sx,sy,sw, sh)
```

方法概述：getImageData 方法根据给定的绘图画布矩形面积 (sx, sy, sw, sh)，生成画布上该矩形面积的图形内容，并综合为 ImageData 对象返回。画布外的像素作为透明黑返回。

#### 3.11.3 putImageData 方法

```
imagedata = context . putImageData(imagedata, dx,  
dy[, dirtyX, dirtyY, dirtyWidth, dirtyHeight])
```

方法概述：在绘图画布上绘制给定的 ImageData 对象。假如脏矩形被提供，则只有在脏矩形上面的像素被绘制。本方法对全局透明、阴影和全局组合属性均忽略。

异常：假如第一个参数不是 ImageData 对象，抛出 TYPE\_MISMATCH\_ERR 异常，假如任一数字参数是无穷或非数字，则抛出 NOT\_SUPPORTED\_ERR 错误

#### 3.11.4 演示例子

下面展示了对一张图片进行反色、透明的一个例子，从例子中可以看出，有了像素级的控制能力，我们可以很轻易的对原有图片进行各种图像滤镜操作。图示如下：



JS 代码：[当你在 word 中选择上面的图片，会发现反色滤镜常用在选择

操作里]

```
var imgSrc = document.getElementById('codeex.cn')
context.drawImage(imgSrc, 10, 10);
var imgd = context.getImageData(10, 10, 100, 122);
var pix = imgd.data;
//反色处理
for(var i=0, n=pix.length; i<n; i+=4)
{
    pix[i] = 255 - pix[i]; //红
    pix[i+1] = 255 - pix[i+1]; //绿
    pix[i+2] = 255 - pix[i+2]; //蓝
    pix[i+3] = pix[i+3]; //alpha
}
context.putImageData(imgd, 130, 10);
imgd = context.getImageData(10, 10, 100, 122);
pix = imgd.data;
//透明处理 透明度0.6
for(var i=0, n=pix.length; i<n; i+=4)
{
    pix[i] = pix[i]; //红
    pix[i+1] = pix[i+1]; //绿
    pix[i+2] = pix[i+2]; //蓝
    pix[i+3] = pix[i+3]*0.6; //alpha
}
context.putImageData(imgd, 260, 10);
```

### 3.12 绘图模型【此段翻译不怎么样，可以参看原英文】

在本文描述的画布中绘图，浏览器一般按照下面的顺序进行绘制：

1. 准备形状或图片，此时图片假设为 A，形状必须被所有属性描述的形状，且经过坐标转换；
2. 当绘制阴影时，准备图片 A，并绘制阴影，形成图片 B；
3. 当绘制阴影时，为 B 的每个像素乘上 alpha 值；
4. 当绘制阴影时，则根据组合参数对 B 和本画布剪贴区域内的图片进行组合；
5. 在图片 A 上每个像素乘上 alpha 值；
6. 在图片 A 上根据组合参数对 A 和本画布剪贴区域内的图片进行组合

4, 5, 6 章节未被翻译。

## 4 参考资料

[BEZIER] *Courbes à poles*, P. de Casteljaeu. INPI, 1959.

[CSS] Cascading Style Sheets Level 2 Revision 1, B. Bos, T. Çelik,

I. Hickson, H. Lie. W3C, April 2009.

[**CSSCOLOR**] CSS Color Module Level 3, T. Çelik, C. Lilley, L. Baron. W3C, August 2008.

[**CSSFONTS**] CSS Fonts Module Level 3, J. Daggett. W3C, June 2009.

[**CSSOM**] Cascading Style Sheets Object Model (CSSOM), A. van Kesteren. W3C, December 2007.

[**ECMA262**] ECMAScript Language Specification. ECMA, December 1999.

[**GRAPHICS**] (Non-normative) *Computer Graphics: Principles and Practice in C*, Second Edition, J. Foley, A. van Dam, S. Feiner, J. Hughes. Addison-Wesley, July 1995. ISBN 0-201-84840-6.

[**HTML5**] HTML 5, I. Hickson, D. Hyatt, eds. World Wide Web Consortium, 23 April 2009, work in progress.

The latest edition of HTML 5 is available at <http://www.w3.org/TR/html5/>.

[**IEEE754**] IEEE Standard for Floating-Point Arithmetic (IEEE 754). IEEE, August 2008. ISBN 978-0-7381-5753-5.

[**PNG**] Portable Network Graphics (PNG) Specification, D. Duce. W3C, November 2003.

[**PORTERDUFF**] Compositing Digital Images, T. Porter, T. Duff. In *Computer graphics*, volume 18, number 3, pp. 253-259. ACM Press, July 1984.

[**RFC2119**] Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997.

Available at <http://tools.ietf.org/html/rfc2119>. =====

[**RFC2119**] Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997.

Available at <http://tools.ietf.org/html/rfc2119>.

[**HTML5**] HTML 5, I. Hickson, D. Hyatt, eds. World Wide Web Consortium, 23 April 2009, work in progress.

This edition of HTML 5 is <http://www.w3.org/TR/2009/WD-html5-20090423/>.

The latest edition of HTML 5 is available at <http://www.w3.org/TR/html5/>. >>>>>>> 1.1 =====