HTML5 资料

1 Canvas 教程

<canvas>是一个新的用于通过脚本（通常是 JavaScript）绘图的 HTML 元素。例如，他可以用于绘图、制作图片的组合或者简单的动画（当然并不那么简单）。It can for instance be used to draw graphs, make photo compositions or do simple (and not so simple) animations.

1.1 基本用法

Basic usage

<canvas>元素

Let's start this tutorial by looking at the <canvas> element itself.

让我们从<canvas>元素的定义开始吧。

<canvas id="tutorial" width="150" height="150"></canvas>

This looks a lot like the <img> element, the only difference is that it doesn't have the src and alt attributes. <canvas>看起来很像<img>，唯一不同就是它不含 src 和 alt 属性。The <canvas> element has only two attributes - width and height. These are both optional and can also be set using DOM properties or CSS rules.它只有两个属性，width 和 height，两个都是可选的，并且都可以 DOM 或者 CSS 来设置。When no width and height attributes are specified, the canvas will initially be 300 pixels wide and 150 pixels high.如果不指定 width 和 height，默认的是宽 300 像素，高 150 像素。The element can be sized arbitrarily by CSS, but during rendering the image is scaled to fit its layout size. (If your renderings seem distorted, try specifying your width and height attributes explicitly in the <canvas> attributes, and not with CSS.)虽然可以通过 CSS 来调整 canvas 的大小，但渲染图像会缩放来适应布局的（如果你发现渲染结果看上去变形了，不必一味依赖 CSS，可以尝试显式指定 canvas 的 width 和 height 属性值）。

The id attribute isn't specific to the <canvas> element but is one of default HTML attributes which can be applied to (almost) every HTML element (like class for instance). It's always a good idea to supply an id because this makes it much easier to identify it in our script.

id 属性不是<canvas>专享的，就像标准的 HTLM 标签一样，任何一个 HTML 元素都可以指定其 id 值。一般，为元素指定 id 是个不错的主意，这样使得在脚本中应用更加方便。

The <canvas> element can be styled just like any normal image (margin, border, background, etc). These rules however don't affect the actual drawing on the canvas. We'll see how this is done later in this tutorial. When no styling rules are applied to the canvas it will initially be fully transparent. <canvas>元素可以像普通图片一样指定其样式（边距，边框，背景等等）。然而这些样式并不会对 canvas 实际

生成的图像产生什么影响。下面我们会看到如何应用样式。如果不指定样式，canvas 默认是全透明的。

替用内容

Because the <canvas> element is still relatively new and isn't implemented in some browsers (such as Firefox 1.0 and Internet Explorer), we need a means of providing fallback content when a browser doesn't support the element.

因为 <canvas> 相对较新，有些浏览器并没实现，如 Firefox 1.0 和 Internet Explorer，所以我们需要为那些不支持 canvas 的浏览器提供替用显示内容。

Luckily this is very straightforward: we just provide alternative content inside the canvas element. Browsers who don't support it will ignore the element completely and render the fallback content, others will just render the canvas normally.

For instance we could provide a text description of the canvas content or provide a static image of the dynamically rendered content. This can look something like this:

我们只需要直接在 canvas 元素内插入替用内容即可。不支持 canvas 的浏览器会忽略 canvas 元素而直接渲染替用内容，而支持的浏览器则会正常地渲染 canvas。例如，我们可以把一些文字或图片填入 canvas 内，作为替用内容：

```
<canvas id="stockGraph" width="150" height="150">
  current stock price: $3.15 +0.15
</canvas>

<canvas id="clock" width="150" height="150">
  <img src="images/clock.png" width="150" height="150"/>
</canvas>
```

结束标签 </canvas> 是必须的

In the Apple Safari implementation, <canvas> is an element implemented in much the same way <img> is; it does not have an end tag. However, for <canvas> to have widespread use on the web, some facility for fallback content must be provided. Therefore, Mozilla's implementation requires an end tag (</canvas>).

在 Apple Safari 里，<canvas>的实现跟<img>很相似，它并不没有结束标签。然而，为了使 <canvas> 能在 web 的世界里广泛适用，需要给替用内容提供一个容身之所，因此，在 Mozilla 的实现里结束标签(</canvas>)是必须的。

If fallback content is not needed, a simple <canvas id="foo" ...></canvas> will be fully compatible with both Safari and Mozilla -- Safari will simply ignore the end tag.

如果没有替用内容，<canvas id="foo" ...></canvas> 对 Safari 和 Mozilla 是完全兼容的—— Safari 会简单地忽略结束标签。

If fallback content is desired, some CSS tricks must be employed to mask the fallback content from Safari (which should render just the canvas), and also to mask the CSS tricks themselves from IE (which should render the fallback content).

如果有替用内容，那么可以用一些 CSS 技巧来为并且仅为 Safari 隐藏替用内容，因为那些替用内容是需要在 IE 里显示但不需要在 Safari 里显示。

渲染上下文（Rendering Context）

<canvas> creates a fixed size drawing surface that exposes one or more rendering contexts, which are used to create and manipulate the content shown. We'll focus on the 2D rendering context, which is the only currently defined rendering context. In the future, other contexts may provide different types of rendering; for example, it is likely that a 3D context based on OpenGL ES will be added.

<canvas> 创建的固定尺寸的绘图画面开放了一个或多个渲染上下文（rendering context），我们可以通过它们来控制要显示的内容。我们专注于 2D 渲染上，这也是目前唯一的选择，可能在将来会添加基于 OpenGL ES 的 3D 上下文。

The <canvas> is initially blank, and to display something a script first needs to access the rendering context and draw on it. The canvas element has a DOM method called getContext, used to obtain the rendering context and its drawing functions. getContext() takes one parameter, the type of context.

<canvas> 初始化是空白的，要在上面用脚本画图首先需要其渲染上下文（rendering context），它可以通过 canvas 元素对象的 getContext 方法来获取，同时得到的还有一些画图用的函数。getContext() 接受一个用于描述其类型的值作为参数。

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

In the first line we retrieve the canvas DOM node using the getElementById method. We can then access the drawing context using the getContext method.

上面第一行通过 getElementById 方法取得 canvas 对象的 DOM 节点。然后通过其 getContext 方法取得其画图操作上下文。

检查浏览器的支持

The fallback content is displayed in browsers which do not support <canvas>; scripts can also check for support when they execute. This can easily be done by testing for the getContext method. Our code snippet from above becomes something like this:

除了在那些不支持 的浏览器上显示替用内容，还可以通过脚本的方式来检查浏览器是否支持 canvas 。方法很简单，判断 getContext 是否存在即可。

```
var canvas = document.getElementById('tutorial');
if (canvas.getContext){
  var ctx = canvas.getContext('2d');
  // drawing code here
} else {
  // canvas-unsupported code here
}
```

代码模板

Here is a minimalistic template, which we'll be using as a starting point for later examples. You can download this file to work with on your system.

我们会用下面这个最简化的代码模板来（后续的示例需要用到）作为开始，你可以 下载文件 到本地备用。

```
<html>
  <head>
    <title>Canvas tutorial</title>
    <script type="text/javascript">
      function draw(){
        var canvas = document.getElementById('tutorial');
        if (canvas.getContext){
          var ctx = canvas.getContext('2d');
        }
      }
    </script>
    <style type="text/css">
      canvas { border: 1px solid black; }
    </style>
  </head>
  <body onload="draw();">
    <canvas id="tutorial" width="150" height="150"></canvas>
  </body>
</html>
```

If you look at the script you'll see I've made a function called draw, which will get executed once the page finishes loading (via the onload attribute on the body tag). This function could also have been called from a setTimeout, setInterval, or any other event handler function just as long the page has been loaded first.

细心的你会发现我准备了一个名为 draw 的函数，它会在页面装载完毕之后执行一次（通过设置 body 标签的 onload 属性），它当然也可以在 setTimeout，setInterval，或者其他事件处理函数中被调用。

一个简单的例子

To start off, here's a simple example that draws two intersecting rectangles, one of which has alpha transparency. We'll explore how this works in more detail in later examples.

作为开始，来一个简单的吧——绘制两个交错的矩形，其中一个是有 alpha 透明效果。我们会在后面的示例中详细的让你了解它是如何运作的。

观看示例

```
<html>
 <head>
  <script type="application/x-javascript">
    function draw() {
      var canvas = document.getElementById("canvas");
      if (canvas.getContext) {
        var ctx = canvas.getContext("2d");

        ctx.fillStyle = "rgb(200,0,0)";
        ctx.fillRect (10, 10, 55, 50);

        ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
        ctx.fillRect (30, 30, 55, 50);
      }
    }
  </script>
 </head>
 <body onload="draw();">
   <canvas id="canvas" width="150" height="150"></canvas>
 </body>
</html>
```

1.2 绘图

Drawing shapes 绘制图形

网格 The grid

Before we can start drawing, we need to talk about the canvas grid or coordinate space. The HTML template on the previous page had a canvas element 150 pixels wide and 150 pixels high. I've drawn this image with the default grid overlayed. Normally 1 unit in the grid corresponds to 1 pixel on the canvas. The origin of this grid is positioned in the top left corner (coordinate (0,0)). All elements are placed relative to this origin. So the position of the top left corner of the blue square becomes x pixels from the left and y pixels from the top (coordinate (x,y)). Later in this tutorial we'll see how we can translate the origin to a different position, rotate the grid and even scale it. For now we'll stick to the default.

在真正开始之前，我们需要先探讨 canvas 的网格（grid）或者坐标空间（coordinate space）。在前一页的 HTML 模板里有一个 150 像素宽， 150 像素高的 canvas 对象。我在画面上叠加上默认网格，如右图。通常网格的 1 个单元对应 canvas 上的 1 个像素。网格的原点是定位在左上角（坐标(0,0)）。画面里的所有物体的位置都是相对这个原点。这样，左上角的蓝色方块的位置就是距左边 x 像素和距上边 Y 像素（坐标(x, y)）。后面的教程中我们将学会如何把移动原点，旋转以及缩放网格。不过现在我们会使用默认的状态。

绘制图形 Drawing shapes

Unlike SVG, canvas only supports one primitive shape - rectangles. All other shapes must be created by combining one or more paths. Luckily, we have a collection of path drawing functions which make it possible to compose very complex shapes.

不像 SVG，canvas 只支持一种基本形状——矩形，所以其它形状都是有一个或多个路径组合而成。还好，有一组路径绘制函数让我们可以绘制相当复杂的形状。

矩形 Rectangles

First let's look at the rectangle. There are three functions that draw rectangles on the canvas:

我们首先看看矩形吧，有三个函数用于绘制矩形的：

fillRect(x,y,width,height) : Draws a filled rectangle

strokeRect(x,y,width,height) : Draws a rectangular outline

clearRect(x,y,width,height) : Clears the specified area and makes it fully transparent

Each of these three functions takes the same parameters. x and y specify the position on the canvas (relative to the origin) of the top-left corner of the rectangle. width and height are pretty obvious. Let's see these functions in action.

它们都接受四个参数， x 和 y 指定矩形左上角(相对于原点)的位置，width 和 height 是矩形的宽和高。好，实战一下吧。

Below is the draw() function from the previous page, but now I've added the three functions above.

下面就是上页模板里的 draw() 函数，但添加了上面的三个函数。

绘制矩形的例子 Rectangular shape example

观看示例

```
function draw(){
  var canvas = document.getElementById('tutorial');
  if (canvas.getContext){
    var ctx = canvas.getContext('2d');

    ctx.fillRect(25,25,100,100);
    ctx.clearRect(45,45,60,60);
    ctx.strokeRect(50,50,50,50);
  }
}
```

The result should look something like the image on the right. The fillRect function draws a large black square 100x100 pixels. The clearRect function removes a 60x60 pixels square from the center and finally the strokeRect draws a rectangular outline 50x50 pixels inside the cleared square. In the following pages we'll see two alternative methods for the clearRect function and we'll also see how to change the color and stroke style of the rendered shapes.

出来的结果应该和右边的是一样的。fillRect 函数画了一个大的黑色矩形（100x100），clearRect 函数清空了中间 60x60 大小的方块，然后 strokeRect 函数又在清空了的空间内勾勒出一个 50x50 的矩形边框。在接下去的页面里，我们会看到和 clearRect 函数差不多另外两个方法，以及如何去改变图形的填充和边框颜色。

Unlike the path functions we'll see in the next section, all three rectangle functions draw immediately to the canvas.

与下一节的路径函数不一样，这三个函数的效果会立刻在 canvas 上反映出来。

绘制路径 Drawing paths

To make shapes using paths, we need a couple of extra steps.

不像画矩形那样的直截了当，绘制路径是需要一些额外的步骤的。

beginPath()

closePath()

stroke()

fill()

The first step to create a path is calling the beginPath method. Internally, paths are stored as a list of sub-paths (lines, arcs, etc) which together form a shape. Every time this method is called, the list is reset and we can start drawing new shapes.

第一步是用 beginPath 创建一个路径。在内存里，路径是以一组子路径（直线，弧线等）的形式储存的，它们共同构成一个图形。每次调用 beginPath，子路径组都会被重置，然后可以绘制新的图形。

The second step is calling the methods that actually specify the paths to be drawn. We'll see these shortly.

第二步就是实际绘制路径的部分，很快我们就会看到。

The third, and an optional step, would be to call the closePath method. This method tries to close the shape by drawing a straight line from the current point to the start. If the shape has already been closed or there's only one point in the list, this function does nothing.

第三步是调用 closePath 方法，它会尝试用直线连接当前端点与起始端点来关闭路径，但如果图形已经关闭或者只有一个点，它会什么都不做。这一步不是必须的。

The final step will be calling the stroke and/or fill methods. Calling one of these will actually draw the shape to the canvas. stroke is used to draw an outlined shape, while fill is used to paint a solid shape.

最后一步是调用 stroke 或 fill 方法，这时，图形才是实际的绘制到 canvas 上去。stroke 是绘制图形的边框，fill 会用填充出一个实心图形。

Note: When calling the fill method any open shapes will be closed automatically and it isn't necessary to use the closePath method.

注意：当调用 fill 时，开放的路径会自动闭合，而无须调用 closePath 。

The code for a drawing simple shape (a triangle) would look something like this.

画一个简单图形（如三角形）的代码如下。

```
ctx.beginPath();
ctx.moveTo(75,50);

ctx.lineTo(100,75);

ctx.lineTo(100,25);

ctx.fill();
```

moveTo

One very useful function, which doesn't actually draw anything, but is part of the path list described above, is the moveTo function. You can probably best think of this as lifting a pen or pencil from one spot on a piece of paper and placing it on the next.

moveTo 是一个十分有用的方法，虽然并不能用它来画什么，但却是绘制路径的实用方法的一部分。你可以把它想象成是把笔提起，并从一个点移动到另一个点的过程。

moveTo(x, y)

The moveTo function takes two arguments - x and y, - which are the coordinates of the new starting point.

它接受 x 和 y （新的坐标位置）作为参数。

When the canvas is initialized or the beginPath method is called, the starting point is set to the coordinate (0,0). In most cases we would use the moveTo method to place the starting point somewhere else. We could also use the moveTo method to draw unconnected paths. Take a look at the smiley face on the right. I've marked the places where I used the moveTo method (the red lines).

当 canvas 初始化或者调用 beginPath 的时候，起始坐标设置就是原点(0,0)。大多数情况下，我们用 moveTo 方法将起始坐标移至其它地方，或者用于绘制不连续的路径。看看右边的笑脸，红线就是使用 moveTo 移动的轨迹。

To try this for yourself, you can use the code snippet below. Just paste it into the draw function we saw earlier.

试一试下面的代码，粘贴到之前用过的 draw 函数内在看看效果吧。

moveTo 的使用示例

观看示例

```
ctx.beginPath();
ctx.arc(75,75,50,0,Math.PI*2,true); // Outer circle
ctx.moveTo(110,75);
ctx.arc(75,75,35,0,Math.PI,false);   // Mouth (clockwise)
ctx.moveTo(65,65);
ctx.arc(60,65,5,0,Math.PI*2,true);  // Left eye
ctx.moveTo(95,65);
ctx.arc(90,65,5,0,Math.PI*2,true);  // Right eye
ctx.stroke();

//thegoneheart 完整例子

ctx.beginPath();
ctx.arc(75,75,50,0,Math.PI*2,true); // Outer circle
ctx.moveTo(110,75);
ctx.arc(75,75,35,0,Math.PI,false);   // Mouth (clockwise)
ctx.moveTo(65,65);
ctx.arc(60,65,5,0,Math.PI*2,true);  // Left eye
ctx.moveTo(95,65);
ctx.arc(90,65,5,0,Math.PI*2,true);  // Right eye
ctx.stroke();

ctx.beginPath();
ctx.moveTo(40,75);
ctx.lineTo(60,65);
ctx.lineTo(90,65);
ctx.moveTo(110,75);
ctx.lineTo(125,75);
ctx.stroke();
```

Note: remove the moveTo methods to see the connecting lines.

Note: For a description of the arc function and its parameters look below.

注意：你可以注释 moveTo 方法来观察那些连接起来的线。

注意：arc 方法的用法见下面。

绘制各种线条 Lines

For drawing straight lines we use the lineTo method.

我们用 lineTo 方法来画直线。

lineTo(x, y)

This method takes two arguments - x and y, - which are the coordinates of the line's end point. The starting point is dependent on previous drawn paths, where the end point of the previous path is the starting point for the following, etc. The starting point can also be changed by using the moveTo method.

lineTo 方法接受终点的坐标（x，y）作为参数。起始坐标取决于前一路径，前一路径的终点即当前路径的起点，起始坐标也可以通过 moveTo 方法来设置。

lineTo 的使用示例

In the example below two triangles are drawn, one filled and one outlined. (The result can be seen in the image on the right). First the beginPath method is called to begin a new shape path. We then use the moveTo method to move the starting point to the desired position. Below this two lines are drawn which make up two sides of the triangle.

示例（如右图）画的是两个三角形，一个实色填充，一个勾边。首先调用 beginPath 方法创建一个新路径，然后用 moveTo 方法将起始坐标移至想要的位置，然后画两条直线来构成三角形的两条边。

You'll notice the difference between the filled and stroked triangle. This is, as mentioned above, because shapes are automatically closed when a path is filled. If we would have done this for the stroked triangle only two lines would have been drawn, not a complete triangle.

可以注意到 fill 和 strok 绘三角形的区别，上面也提到过，使用 fill 路径会自动闭合，但使用 stroke 不会，如果不关闭路径，勾画出来的只有两边。

观看示例

```
// 填充三角形
ctx.beginPath();
ctx.moveTo(25,25);
ctx.lineTo(105,25);
ctx.lineTo(25,105);
ctx.fill();

// 勾边三角形
ctx.beginPath();
ctx.moveTo(125,125);
ctx.lineTo(125,45);
ctx.lineTo(45,125);
ctx.closePath();
ctx.stroke();
```

弧线 Arcs

For drawing arcs or circles we use the arc method. The specification also describes the arcTo method, which is supported by Safari but hasn't been implemented in the current Gecko browsers.

我们用 arc 方法来绘制弧线或圆。标准说明中还包含 arcTo 方法，当前 Safari 是支持的，但基于 Gecko 的浏览器还未实现。

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

This method takes five parameters: x and y are the coordinates of the circle's center. Radius is self explanatory. The startAngle and endAngle parameters define the start and end points of the arc in radians. The starting and closing angle are measured from the x axis. The anticlockwise parameter is a boolean value which when true draws the arc anticlockwise, otherwise in a clockwise direction.

方法接受五个参数：x，y 是圆心坐标，radius 是半径，startAngle 和 endAngle 分别是起末弧度（以 x 轴为基准），anticlockwise 为 true 表示逆时针，反之顺时针。

Warning: In the Firefox beta builds, the last parameter is clockwise. The final release will support the function as described above. All scripts that use this method in its current form will need to be updated once the final version is released.

警告：在 Firefox 的 beta 版本里，最后一个参数是 clockwise，而最终版本不是。因此如果是从 beta 升级至发行版需要做相应修改。

Note: Angles in the arc function are measured in radians, not degrees. To convert degrees to radians you can use the following JavaScript expression: var radians = (Math.PI/180)*degrees.

注意：arc 方法里用到的角度是以弧度为单位而不是度。度和弧度直接的转换可以用这个表达式：var radians = (Math.PI/180)*degrees;。

arc 的使用示例

The following example is a little more complex than the ones we've seen above. I've drawn 12 different arcs all with different angles and fills. If I would have written this example just like the smiley face above, firstly this would have become a very long list of statements and secondly, when drawing arcs, I would need to know every single starting point. For arcs of 90, 180 and 270 degrees, like the ones I used here, this wouldn't be to much of a problem, but for more complex ones this becomes way too difficult.

这个示例比之前见到过的要复杂一些，画了 12 个不同的弧形，有不同夹角和填充状态的。如果我用上面画笑脸的方式来画这些弧形，那会是一大段的代码，而且，画每一个弧形时我都需要知道其圆心位置。像我这里画 90，180 和 270 度的弧形看起来不是很麻烦，但是如果图形更复杂一些,则实现起来会越来越困难。

The two for loops are for looping through the rows and columns of arcs. For every arc I start a new path using beginPath. Below this I've written out all the parameters as variables, so it's easier to read what's going on. Normally this would be just one statement. The x and y coordinates should be clear enough. radius and startAngle are fixed. The endAngle starts of as 180 degrees (first column) and is increased with steps of 90 degrees to form a complete circle (last column). The statement for the clockwise parameter results in the first and third row being drawn as clockwise arcs and the second and fourth row as counterclockwise arcs. Finally, the if statement makes the top half stroked arcs and the bottom half filled arcs.

这里使用两个 for 循环来画多行多列的弧形。每一个弧形都用 beginPath 方法创建一个新路径。然后为了方便阅读和理解，我把所有参数都写成变量形式。显而易见，x 和 y 作为圆心坐标。 radius 和 startAngle 都是固定，endAngle 从 180 度半圆开始，以 90 度方式递增至圆。 anticlockwise 则取决于奇偶行数。最后，通过 if 语句判断使前两行表现为勾边，而后两行为填充效果。

观看示例

```
for (i=0;i<4;i++){
  for(j=0;j<3;j++){    //chinese_xu 原始代码
   ctx.beginPath();
   var x            = 25+j*50; // x coordinate
   var y            = 25+i*50; // y coordinate
   var radius       = 20; // Arc radius
   var startAngle   = 0; // Starting point on circle
   var endAngle     = Math.PI+(Math.PI*j)/2；  // End point on circle
   var anticlockwise  = i%2==0 ? false : true; // clockwise or anticlockwise

   ctx.arc(x,y,radius,startAngle,endAngle, anticlockwise);

   if (i>1){
     ctx.fill();
   } else {
     ctx.stroke();
   }
  }
}
//chinese_xu 原始代码并没有按照 1/4 圆递增来画。
//修改后输出 4 行 4 列，要把画布扩大到 200*200 观看
for (i=0;i<4;i++){
  for(j=0;j<4;j++){
   ctx.beginPath();
   var x            = 25+j*50; // x coordinate
   var y            = 25+i*50; // y coordinate
   var radius       = 20; // Arc radius
   var startAngle   = 0; // Starting point on circle
   var endAngle     = Math.PI*(2-j/2);  // End point on circle
   var anticlockwise  = i%2==0 ? false : true; // clockwise or anticlockwise

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

```
    if (i>1){
      ctx.fill();
    } else {
      ctx.stroke();
    }
  }
}
```

贝塞尔和二次方曲线 Bezier and quadratic curves

The next type of paths available are Bézier curves, available in the cubic and quadratic varieties. These are generally used to draw complex organic shapes.

接下来要介绍的路径是 贝塞尔曲线 ， 它可以是二次和三次方的形式，一般用于绘制复杂而有规律的形状。

```
quadraticCurveTo(cp1x, cp1y, x, y) // BROKEN in Firefox 1.5 (see work around below)
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

The difference between these can best be described using the image on the right. A quadratic Bézier curve has a start and an end point (blue dots) and just one control point (red dot) while a cubic Bézier curve uses two control points.

上面两行代码的区别见右图。它们都有一个起点一个终点（图中的蓝点），但二次方贝塞尔曲线只有一个（红色）控制点点）而三次方贝塞尔曲线有两个。

The x and y parameters in both these methods are the coordinates of the end point. cp1x and cp1y are the coordinates of the first control point, and cp2x and cp2y are the coordinates of the second control point.

参数 x 和 y 是终点坐标，cp1x 和 cp1y 是第一个控制点的坐标，cp2x 和 cp2y 是第二个的。

Using quadratic and cubic Bézier curves can be quite challenging, because unlike vector drawing software like Adobe Illustrator, we don't have direct visual feedback as to what we're doing. This makes it pretty hard to draw complex shapes. In the following example, we'll be drawing some simple organic shapes, but if you have the time and, most of all, the patience, much more complex shapes can be created.

使用二次方和三次方的贝塞尔曲线是相当有挑战的，因为不像在矢量绘图软件 Adobe Illustrator 里那样有即时的视觉反馈。因为用它来画复杂图形是比较麻烦的。但如果你有时间，并且最重要是有耐心，再复杂的图形都可以绘制出来的。下面我们来画一个简单而又规律的图形。

There's nothing very difficult in these examples. In both cases we see a succession of curves being drawn which finally result in a complete shape.

这些例子都比较简单。我们绘制的都是完整的图形。

quadraticCurveTo 的使用示例

查看示例

```
// Quadratic curves example
ctx.beginPath();
ctx.moveTo(75,25);
ctx.quadraticCurveTo(25,25,25,62.5);
ctx.quadraticCurveTo(25,100,50,100);
ctx.quadraticCurveTo(50,120,30,125);
ctx.quadraticCurveTo(60,120,65,100);
ctx.quadraticCurveTo(125,100,125,62.5);
ctx.quadraticCurveTo(125,25,75,25);
ctx.stroke();
```

It is possible to convert any quadratic Bézier curve to a cubic Bézier curve by correctly computing both cubic Bézier control points from the single quadratic Bézier control point, although the reverse is NOT true. An exact conversion of a cubic Bézier curve to a quadratic Bézier curve is only possible if the cubic term is zero, more commonly a subdivision method is used to approximate a cubic Bézier using multiple quadratic Bézier curves.

通过计算，可以由二次曲线的单个控制点得出相应三次方曲线的两个控制点，因此二次方转三次方是可能的，但是反之不然。仅当三次方程中的三次项为零是才可能转换为二次的贝塞尔曲线。通常地可以用多条二次方曲线通过细分算法来近似模拟三次方贝塞尔曲线。

bezierCurveTo 的使用示例

查看示例

```
// Bezier curves example
ctx.beginPath();
ctx.moveTo(75,40);
ctx.bezierCurveTo(75,37,70,25,50,25);
ctx.bezierCurveTo(20,25,20,62.5,20,62.5);
ctx.bezierCurveTo(20,80,40,102,75,120);
ctx.bezierCurveTo(110,102,130,80,130,62.5);
ctx.bezierCurveTo(130,62.5,130,25,100,25);
ctx.bezierCurveTo(85,25,75,37,75,40);
ctx.fill();
```

Firefox 1.5 quadraticCurveTo() bug 的应对方案

There is a bug in the Firefox 1.5 implementation of quadatricCurveTo(). It does NOT draw a quadratic curve, as it is just calling the same cubic curve function bezierCurveTo() calls, and repeating the single quadratic control point (x,y) coordinate twice. For this reason quadraticCurveTo() will yield incorrect results. If you require the use of quadraticCurveTo() you must convert your quadratic Bézier curve to a cubic Bézier curve yourself, so you can use the working bezierCurveTo() method.

在 Firefox 1.5 里，quadatricCurveTo() 的实现是有 bug 的，它不是直接绘制二次方曲线，而是调用 bezierCurveTo()，其中两个控制点都是二次方曲线的那个单控制点。因此，它会绘制出不正确的曲线。如果必须使用到 quadraticCurveTo()，你需要自行去将二次方曲线转换成三次方的，这样就可以用 bezierCurveTo() 方法了。

```
var currentX, currentY;  // set to last x,y sent to lineTo/moveTo/bezierCurveTo or quadraticCurveToFixed()

function quadraticCurveToFixed( cpx, cpy, x, y ) {
  /*
   For the equations below the following variable name prefixes are used:
     qp0 is the quadratic curve starting point (you must keep this from your last point sent to moveTo(), lineTo(), or bezierCurveTo() ).
     qp1 is the quadatric curve control point (this is the cpx,cpy you would have sent to quadraticCurveTo() ).
     qp2 is the quadratic curve ending point (this is the x,y arguments you would have sent to quadraticCurveTo() ).
   We will convert these points to compute the two needed cubic control points (the starting/ending points are the same for both
   the quadratic and cubic curves.

   The equations for the two cubic control points are:
     cp0=qp0 and cp3=qp2
     cp1 = qp0 + 2/3 *(qp1-qp0)
     cp2 = cp1 + 1/3 *(qp2-qp0)

   In the code below, we must compute both the x and y terms for each point separately.

    cp1x = qp0x + 2.0/3.0*(qp1x - qp0x);
    cp1y = qp0y + 2.0/3.0*(qp1y - qp0y);
    cp2x = cp1x + (qp2x - qp0x)/3.0;
    cp2y = cp1y + (qp2y - qp0y)/3.0;

   We will now
     a) replace the qp0x and qp0y variables with currentX and currentY (which *you* must store for each
moveTo/lineTo/bezierCurveTo)
     b) replace the qp1x and qp1y variables with cpx and cpy (which we would have passed to
quadraticCurveTo)
     c) replace the qp2x and qp2y variables with x and y.
   which leaves us with:
  */
  var cp1x = currentX + 2.0/3.0*(cpx - currentX);
  var cp1y = currentY + 2.0/3.0*(cpy - currentY);
  var cp2x = cp1x + (x - currentX)/3.0;
  var cp2y = cp1y + (y - currentY)/3.0;

  // and now call cubic Bezier curve to function
  bezierCurveTo( cp1x, cp1y, cp2x, cp2y, x, y );

  currentX = x;
  currentY = y;
}
```

矩形路径 Rectangles

Besides the three methods we saw above which draw rectangular shapes directly to the canvas, we also have a method rect which adds a rectangular path to the path list.

除了上面提到的三个方法可以直接绘制矩形之外，我们还有一个 rect 方法是用于绘制矩形路径的。

rect(x, y, width, height)

This method takes four arguments. The x and y parameters define the coordinate of the top left corner of the new rectangular path. width and height define the width and the height of the rectangle.

它接受四个参数，x 和 y 是其左上角坐标，width 和 height 是其宽和高。

When this method is executed, the moveTo method is automatically called with the parameters (0,0) (i.e. it resets the starting point to its default location).

当它被调用时，moveTo 方法会自动被调用，参数为(0,0)，于是起始坐标又恢复成初始原点了。

综合 Making combinations

In all examples on this page I've only used one type of path function per shape. However there's absolutely no limitation to the amount or type of paths you can use to create a shape. So in this last example I've tried to combine all of the path functions to make a set of very famous game characters.

上面所用到的例子都只用到了一种类型的路径，当然 canvas 不会限制所使用的路径类型的多少。所以，我们来看一个路径大杂烩。

综合样例

I'm not going to run through this complete script, but the most important things to note are the function roundedRect and the use of the fillStyle property. It can be very usefull and time saving to define your own functions to draw more complex shapes. In this script it would have taken me twice as many lines of code as I have now.

We will look at the fillStyle property in greater depth later in this tutorial. Here I'm using it to change the fill color from the default black, to white, and back again.

在整个例子里，最值得注意的是 roundedRect 函数的使用和 fillStyle 属性的设置。自定义函数对于封装复杂图形的绘制是非常有用的。在这个例子里使用自定义函数就省掉了大约一半的代码。

在接下来的例子里会深入探讨 fillStyle 属性的使用。这里是用它来改变填充颜色，从默认的黑色，到白色，然后再回到黑色。

查看示例

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  roundedRect(ctx,12,12,150,150,15);
  roundedRect(ctx,19,19,150,150,9);
  roundedRect(ctx,53,53,49,33,10);
  roundedRect(ctx,53,119,49,16,6);
  roundedRect(ctx,135,53,49,33,10);
  roundedRect(ctx,135,119,25,49,10);

  ctx.beginPath();
  ctx.arc(37,37,13,Math.PI/7,-Math.PI/7,false); //chiensexu  本来是 true 呵呵，反了
  ctx.lineTo(31,37);
  ctx.fill();
  for(i=0;i<8;i++){
    ctx.fillRect(51+i*16,35,4,4);
  }
  for(i=0;i<6;i++){
    ctx.fillRect(115,51+i*16,4,4);
  }
  for(i=0;i<8;i++){
    ctx.fillRect(51+i*16,99,4,4);
  }
  ctx.beginPath();
  ctx.moveTo(83,116);
  ctx.lineTo(83,102);
  ctx.bezierCurveTo(83,94,89,88,97,88);
  ctx.bezierCurveTo(105,88,111,94,111,102);
  ctx.lineTo(111,116);
  ctx.lineTo(106.333,111.333);
  ctx.lineTo(101.666,116);
  ctx.lineTo(97,111.333);
  ctx.lineTo(92.333,116);
  ctx.lineTo(87.666,111.333);
  ctx.lineTo(83,116);
  ctx.fill();
  ctx.fillStyle = "white";
  ctx.beginPath();
  ctx.moveTo(91,96);
  ctx.bezierCurveTo(88,96,87,99,87,101);
  ctx.bezierCurveTo(87,103,88,106,91,106);
  ctx.bezierCurveTo(94,106,95,103,95,101);
  ctx.bezierCurveTo(95,99,94,96,91,96);
  ctx.moveTo(103,96);
  ctx.bezierCurveTo(100,96,99,99,99,101);
  ctx.bezierCurveTo(99,103,100,106,103,106);
  ctx.bezierCurveTo(106,106,107,103,107,101);
  ctx.bezierCurveTo(107,99,106,96,103,96);
  ctx.fill();
  ctx.fillStyle = "black";
  ctx.beginPath();
  ctx.arc(101,102,2,0,Math.PI*2,true);
  ctx.fill();
  ctx.beginPath();
  ctx.arc(89,102,2,0,Math.PI*2,true);
  ctx.fill();
}
function roundedRect(ctx,x,y,width,height,radius){
  ctx.beginPath();
  ctx.moveTo(x,y+radius);
  ctx.lineTo(x,y+height-radius);
 ctx.quadraticCurveTo(x,y+height,x+radius,y+height);
  ctx.lineTo(x+width-radius,y+height);
 ctx.quadraticCurveTo(x+width,y+height,x+width,y+height-radius);
  ctx.lineTo(x+width,y+radius);
  ctx.quadraticCurveTo(x+width,y,x+width-radius,y);
  ctx.lineTo(x+radius,y);
  ctx.quadraticCurveTo(x,y,x,y+radius);
  ctx.stroke();
}
```

1.3 使用图像

应用图像 Using images

One of the more fun features of the canvas is the abillity to use images. These can be used to do dynamic photo compositing or used as backdrops of graphs etc. It's currently also the only way to add text to them (The specification does not contain any functions to draw text). External images can be used in any format supported by Gecko (e.g. PNG, GIF or JPEG format). Other canvas elements on the same page can also be used as the source.

Canvas 相当有趣的一项功能就是可以引入图像，它可以用于图片合成或者制作背景等。而目前仅可以在图像中加入文字（标准说明中并没有包含绘制文字的功能）。只要是 Gecko 支持的图像（如 PNG，GIF，JPEG 等）都可以引入到 canvas 中，而且其它的 canvas 元素也可以作为图像的来源。

引入图像 Importing images

Importing images is basically a two step process:

引入图像只需要简单的两步：

Firstly we need a reference to a JavaScript Image object or other canvas element as a source. It isn't possible to use images by simply providing a URL/path to them.

Secondly we draw the image on the canvas using the drawImage function.

第一当然是来源图片，不是简单的 URL 路径，但可以是一个 JavaScript 的 Image 对象引用，又或者其它的 canvas 元素。

然后用 drawImage 方法将图像插入到 canvas 中。

Let's look at step one first. There are basically four options available:

先来看看第一步，基本上有四种可选方式：

引用页面内的图片 Using images which are on the same page

We can access all images on a page by using either the document.images collection, the document.getElementsByTagName method, or if we know the ID attribute of the image, the document.getElementById method.

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。

使用其它 canvas 元素 Using other canvas elements

Just as with normal images we access other canvas elements using either the document.getElementsByTagName method or the document.getElementById method. Make sure you've drawn something to the source canvas before using it in your target canvas.

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。

One of the more practical uses of this would be to use a second canvas element as a thumbnail view of the other larger canvas.

一个常用的应用就是为另一个大的 canvas 做缩略图。

由零开始创建图像 Creating an image from scratch

Another option is to create new Image objects in our script. The main disadvantage of this approach is that if we don't want our script to halt in the middle because it needs to wait for an image to load, we need some form of image preloading.

另外，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。

Basically to create a new image object we do this:

我们可以通过下面简单的方法来创建图片：

view plainprint?

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

When this script gets executed, the image starts loading. If loading isn't finished when a drawImage statement gets executed, the script halts until the image is finished loading. If you don't want this to happen, use an onload event handler:

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：

view plainprint?

```
var img = new Image();   // Create new Image object
img.onload = function(){
  // execute drawImage statements here
}
img.src = 'myImage.png'; // Set source path
```

If you're only using one external image this can be a good approach but once you need to track more than one we need to resort to something more cunning. It's beyond the scope of this tutorial to look at image preloading tactics but you can check out JavaScript Image Preloader for a complete solution.

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教程的范围，感兴趣的话可以参考 JavaScript Image Preloader。

通过 data: url 方式嵌入图像 Embedding an image via data: url

Another possible way to include images is via the data: url. Data urls allow you to completely define an image as a Base64 encoded string of characters directly in your code. One advantage of data urls is that the resulting image is available immediately without another round trip to the server. (Another advantage is that it is then possible to encapsulate in one file all of your CSS, Javascript, HTML, and images, making it more portable to other locations.) Some disadvantages of this method are that your image is not cached, and for larger images the encoded url can become quite long:

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：

view plainprint?

```
var img_src =
'data:image/gif;base64,R0lGODlhCwALAIAAAAAA3pn/ZiH5BAEAAAEALAAAAAALAAsAAAIUhA+hkcuO4lmN
Vindo7qyrIXiGBYAOw==';
```

drawImage

Once we have a reference to our source image object we can use the drawImage method to render it to the canvas. As we we'll see later the drawImage method is overloaded and has three different variants. In its most basic form it looks like this.

一旦获得了源图对象，我们就可以使用 drawImage 方法将它渲染到 canvas 里。drawImage 方法有三种形态，下面是最基础的一种。

drawImage(image, x, y)

Where image is a reference to our image or canvas object. x and y form the coordinate on the target canvas where our image should be placed.

其中 image 是 image 或者 canvas 对象，x 和 y 是其在目标 canvas 里的起始坐标。

drawImage 示例 1

In the following example I will be using an external image as the backdrop of a small line graph. Using backdrops can make your script considerably smaller because we don't need to draw an elaborate background. I'm only using one image here so I use the image object's onload event handler to execute the drawing statements. The drawImage method places the backdrop on the coordinate (0,0) which is the top left corner of the canvas.

下面一个例子我用一个外部图像作为一线性图的背景。用背景图我们就不需要绘制负责的背景，省下不少代码。这里只用到一个 image 对象，于是就在它的 onload 事件响应函数中触发绘制动作。drawImage 方法将背景图放置在 canvas 的左上角 (0,0) 处。

查看示例

view plainprint?

```
function draw() {
   var ctx = document.getElementById('canvas').getContext('2d');
   var img = new Image();
   img.onload = function(){
     ctx.drawImage(img,0,0);
     ctx.beginPath();
     ctx.moveTo(30,96);
     ctx.lineTo(70,66);
     ctx.lineTo(103,76);
     ctx.lineTo(170,15);
     ctx.stroke();
   }
   img.src = 'images/backdrop.png';
 }
```

缩放 Scaling

The second variant of the drawImage method adds two new parameters and it allows us to place scaled images on the canvas.

drawImage 方法的又一变种是增加了两个用于控制图像在 canvas 中缩放的参数。

drawImage(image, x, y, width, height)

Where width and height is the image's size on the target canvas. 当中 width 和 height 分别是图像在 canvas 中显示大小。

drawImage 示例 2

In this example I'm going to use an image as a wallpaper and repeat it several times on the canvas. This is done simply by looping and placing the scaled images at different positions. In the code below the first for loops through the rows the second for loop the columns. The image is scaled one third of its original size which is 50x38 pixels. We'll see how this could also have been achieved, by creating a custom pattern, later in this tutorial.

在这个例子里，我会用一张图片像背景一样在 canvas 中以重复平铺开来。实现起来也很简单，只需要循环铺开经过缩放的图片即可。见下面的代码，第一层 for 循环是做行重复，第二层是做列重复的。图像大小被缩放至原来的三分之一，50x38 px。这种方法可以用来很好的达到背景图案的效果，在下面的教程中会看到。

Note: Images can become blurry when scaling up or grainy if they're scaled down too much. Scaling is probably best not done if you've got some text in it which needs to remain legible.

注意：图像可能会因为大幅度的缩放而变得起杂点或者模糊。如果您的图像里面有文字，那么最好还是不要进行缩放，因为那样处理之后很可能图像里的文字就会变得无法辨认了。

查看示例

view plainprint?

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d');
    var img = new Image();
    img.onload = function(){
      for (i=0;i<4;i++){
        for (j=0;j<3;j++){
 ctx.drawImage(img,j*50,i*38,50,38);
        }
      }
    }
    img.src = 'images/rhino.jpg';
  }
```

切片 Slicing

The third and last variant of the drawImage method has eight new parameters. We can use this method to slice parts of a source image and draw them to the canvas.

drawImage 方法的第三个也是最后一个变种有 8 个新参数，用于控制做切片显示的。

drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)

The first parameter image, just as with the other variants, is either a reference to an image object or a reference to a different canvas element. For the other eight parametes it's best to look at the image on the right. The first four parameters define the location and size of the slice on the source image. The last four parameters define the position and size on the destination canvas.

第一个参数和其它的是相同的，都是一个图像或者另一个 canvas 的引用。其它 8 个参数最好是参照右边的图解，前 4 个是定义图像源的切片位置和大小，后 4 个则是定义切片的目标显示位置和大小。

Slicing can be a useful tool when you want to make compositions. You could have all elements in a single image file and use this method to composite a complete drawing. For instance, if you want to make a chart you could have a PNG image containing all the necessary text in a single file and depending on your data could change the scale of your chart without very much diffculty. Another advantage is that you don't need to load every image individually.

切片是个做图像合成的强大工具。假设有一张包含了所有元素的图像，那么你可以用这个方法来合成一个完整图像。例如，你想画一张图表，而手上有一个包含所有必需的文字的 PNG 文件，那么你可以很轻易的根据实际数据的需要来改变最终显示的图表。这方法的另一个好处就是你不需要单独装载每一个图像。

drawImage 示例 3

In this example I'm going to use the same rhino as we've seen above, but now I'm going to slice its head out and composite it into a picture frame. The image of the picture frame includes a dropshadow which has been saved as a 24-bit PNG image. Because 24-bit PNG images include a full 8-bit alpha channel, unlike GIF and 8-bit PNG images, I can place it onto any background and don't have to worry about a matte color.

在这个例子里面我用到上面已经用过的犀牛图像，不过这次我要给犀牛头做个切片特写，然后合成到一个相框里面去。相框带有阴影效果，是一个以 24-bit PNG 格式保存的图像。因为 24-bit PNG 图像带有一个完整的 8-bit alpha 通道，与 GIF 和 8-bit PNG 不同，我可以将它放成背景而不必担心底色的问题。

I took a different approach to the loading of the images than the example above. I just placed the images directly in my HTML document and used a CSS rule to hide them from view (display:none). I assigned both images an id attribute to make them easier to select. The script itself is very simple. I first draw the sliced and scaled image on the canvas (first drawImage statement), and then place the frame on top (second drawImage statement).

我用一个与上面用到的不同的方法来装载图像，直接将图像插入到 HTML 里面，然后通过 CSS 隐藏(display:none)它。两个图像我都赋了 id，方便后面使用。看下面的脚本，相当简单，首先对犀牛头做好切片(第一个 drawImage)放在 canvas 上，然后再上面套个相框(第二个 drawImage)。

查看示例

```
view plainprint?
function draw() {
  var canvas = document.getElementById('canvas');
  var ctx = canvas.getContext('2d');

  // Draw slice
 ctx.drawImage(document.getElementById('source'),
 33,71,104,124,21,20,87,104);

  // Draw frame
 ctx.drawImage(document.getElementById('frame'),0,0);
}
```

示例：画廊 Art gallery example

In the final example of this chapter I've made a little art gallery. The gallery consists of a table containing several images. When the page is loaded, for each image in the page a canvas element is inserted and a frame is drawn arround it.

我这一章最后的示例是弄一个小画廊。画廊由挂着几张画作的格子组成。当页面装载好之后，为每张画创建一个 canvas 元素并用加上画框然后插入到画廊中去。

In my case, all images have a fixed width and height, and so does the frame that's drawn around it. You could enhance the script so that it uses the image's width and height to make the frame fit perfectly around it.

在我这个例子里面，所有“画”都是固定宽高的，画框也是。你可以做些改进，通过脚本用画的宽高来准确控制围绕它的画框的大小。

The code below should be self-explanatory. We loop through the images array and add new canvas elements accordingly. Probably the only thing to note, for those not so familar with the DOM, is the use of the insertBefore method. insertBefore is a method of the parent node (a table cell) of the element (the image) before which we want to insert our new node (the canvas element).

下面的代码应该是蛮自我解释的了。就是遍历图像对象数组，依次创建新的 canvas 元素并添加进去。可能唯一需要注意的，对于那些并不熟悉 DOM 的朋友来说，是 insertBefore 方法的用法。insertBefore 是父节点（单元格）的方法，用于将新节点（canvas 元素）插入到我们想要插入的节点之前。

查看示例

view plainprint?

```
function draw() {

  // Loop through all images
  for (i=0;i<document.images.length;i++){

    // Don't add a canvas for the frame image
    if (document.images[i].getAttribute('id')!='frame'){

      // Create canvas element
      canvas = document.createElement('CANVAS');
      canvas.setAttribute('width',132);
 canvas.setAttribute('height',150);

      // Insert before the image
 document.images[i].parentNode.insertBefore(canvas,document.images[i]);

      ctx = canvas.getContext('2d');

      // Draw image to canvas
 ctx.drawImage(document.images[i],15,20);

      // Add frame
 ctx.drawImage(document.getElementById('frame'),0,0);
```

```
    }
  }
}
```

控制图像的缩放行为 Controlling image scaling behavior

Introduced in Gecko 1.9.2

(Firefox 3.6 / Thunderbird 3.1 / Fennec 1.0)

Gecko 1.9.2 introduced the mozImageSmoothingEnabled property to the canvas element; if this Boolean value is false, images won't be smoothed when scaled. This property is true by default.

Gecko 1.9.2 引入了 mozImageSmoothingEnabled 属性，值为 false 时，图像不会平滑地缩放。默认是 true 。

view plainprint?

```
cx.mozImageSmoothingEnabled = false;
```

1.4 应用风格和颜色

运用样式与颜色

在 绘制图形 的章节里，我只用到默认的线条和填充样式。而在这一章里，我们将会探讨 canvas 全部的可选项，来绘制出更加吸引人的内容。

色彩 Colors

Up until now we've only seen methods of the drawing context. If we want to apply colors to a shape, there are two important properties we can use: fillStyle and strokeStyle.

到目前为止，我们只看到过绘制内容的方法。如果我们想要给图形上色，有两个重要的属性可以做到：fillStyle 和 strokeStyle。

```
fillStyle = color
strokeStyle = color
```

strokeStyle is used for setting the shape outline color and fillStyle is for the fill color. color can be a string representing a CSS color value, a gradient object, or a pattern object. We'll look at gradient and pattern objects later. By default, the stroke and fill color are set to black (CSS color value #000000).

strokeStyle 是用于设置图形轮廓的颜色，而 fillStyle 用于设置填充颜色。color 可以是表示 CSS 颜色值的字符串，渐变对象或者图案对象。我们迟些再回头探讨渐变和图案对象。默认情况下，线条和填充颜色都是黑色(CSS 颜色值 #000000)。

The valid strings you can enter should, according to the specification, be CSS3 color values. Each of the following examples describe the same color.

您输入的应该是符合 CSS3 颜色值标准 的有效字符串。下面的例子都表示同一种颜色。

view plainprint?

```
// 这些 fillStyle 的值均为 '橙色'
ctx.fillStyle = "orange";
ctx.fillStyle = "#FFA500";
ctx.fillStyle = "rgb(255,165,0)";
ctx.fillStyle = "rgba(255,165,0,1)";
```

Note: Currently not all CSS 3 color values are supported in the Gecko engine. For instance the color values hsl(100%,25%,0) or rgb(0,100%,0) are not allowed. If you stick to the ones above, you won't run into any problems.

注意: 目前 Gecko 引擎并没有提供对所有的 CSS 3 颜色值的支持。例如，hsl(100%,25%,0) 或者 rgb(0,100%,0) 都不可用。但如果您遵循上面例子的规范，应该不会有问题。

Note: If you set the strokeStyle and/or fillStyle property, the new value becomes the default for all shapes being drawn from then on. For every shape you want in a different color, you will need to reassign the fillStyle or strokeStyle property.

注意: 一旦您设置了 strokeStyle 或者 fillStyle 的值，那么这个新值就会成为新绘制的图形的默认值。如果你要给每个图形上不同的颜色，你需要重新设置 fillStyle 或 strokeStyle 的值。

fillStyle 示例

In this example, I once again use two for loops to draw a grid of rectangles, each in a different color. The resulting image should look something like the image on the right. There is nothing too spectacular happening here. I use the two variables i and j to generate a unique RGB color for each square. I only modify the red and green values. The blue channel has a fixed value. By modifying the channels, you can generate all kinds of palettes. By increasing the steps, you can achieve something that looks like the color palettes Photoshop uses.

在本示例里，我会再度用两层 for 循环来绘制方格阵列，每个方格不同的颜色。结果如右图，但实现所用的代码却没那么绚丽。我用了两个变量 i 和 j 来为每一个方格产生唯一的 RGB 色彩值，其中仅修改红色和绿色通道的值，而保持蓝色通道的值不变。你可以通过修改这些颜色通道的值来产生各种各样的色板。通过增加渐变的频率，你还可以绘制出类似 Photoshop 里面的那样的调色板。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  for (var i=0;i<6;i++){
    for (var j=0;j<6;j++){
 ctx.fillStyle = 'rgb(' + Math.floor(255-42.5*i) + ',' +
  Math.floor(255-42.5*j) + ',0)';
 ctx.fillRect(j*25,i*25,25,25);
    }
  }
}
```

strokeStyle 示例

This example is similar to the one above but now using the strokeStyle property. Here I use the arc method to draw circles instead of squares.

这个示例与上面的有点类似，但这次用到的是 strokeStyle 属性，而且画的不是方格，而是用 arc 方法来画圆。

查看示例

view plainprint?

```
function draw() {
    var ctx = document.getElementById('canvas').getContext('2d');
    for (var i=0;i<6;i++){
     for (var j=0;j<6;j++){
ctx.strokeStyle = 'rgb(0,' + Math.floor(255-42.5*i) + ',' +
 Math.floor(255-42.5*j) + ')';
ctx.beginPath();
ctx.arc(12.5+j*25,12.5+i*25,10,0,Math.PI*2,true);
ctx.stroke();
     }
   }
 }
```

透明度 Transparency

Besides drawing opaque shapes to the canvas, we can also draw semi-transparent shapes. This is done by setting the globalAlpha property, or we could assign a semi-transparent color to the stroke and/or fill style.

除了可以绘制实色图形，我们还可以用 canvas 来绘制半透明的图形。通过设置 globalAlpha 属性或者使用一个半透明颜色作为轮廓或填充的样式。

globalAlpha = transparency value

This property applies a transparency value to all shapes drawn on the canvas. The valid range of values is from 0.0 (fully transparent) to 1.0 (fully opaque). By default, this property is set to 1.0 (fully opaque).

这个属性影响到 canvas 里所有图形的透明度，有效的值范围是 0.0 （完全透明）到 1.0（完全不透明），默认是 1.0。

The globalAlpha property can be useful if you want to draw a lot of shapes on the canvas with similar transparency. I think, however, that the next option is a little more practical.

globalAlpha 属性在需要绘制大量拥有相同透明度的图形时候相当高效。不过，我认为下面的方法可操作性更强一点。

Because the strokeStyle and fillStyle properties accept CSS 3 color values, we can use the following notation to assign a transparent color to them.

因为 strokeStyle 和 fillStyle 属性接受符合 CSS 3 规范的颜色值，那我们可以用下面的写法来设置具有透明度的颜色。

view plainprint?

```
// Assigning transparent colors to stroke and fill style
ctx.strokeStyle = "rgba(255,0,0,0.5)";
ctx.fillStyle = "rgba(255,0,0,0.5)";
```

The rgba() function is similar to the rgb() function but it has one extra parameter. The last parameter sets the transparency value of this particular color. The valid range is again between 0.0 (fully transparent) and 1.0 (fully opaque).

rgba() 方法与 rgb() 方法类似，就多了一个用于设置色彩透明度的参数。它的有效范围是从 0.0（完全透明）到 1.0（完全不透明）。

globalAlpha 示例

In this example I've drawn a background of four different colored squares. On top of these, I've draw a set of semi-transparent circles. The globalAlpha property is set at 0.2 which will be used for all shapes from that point on. Every step in the for loop draws a set of circles with an increasing radius. The final result is a radial gradient. By overlaying ever more circles on top of each other, we effectively reduce the transparency of the circles that have already been drawn. By increasing the step count and in effect drawing more circles, the background would completely disappear from the center of the image.

在这个例子里，我用四色格作为背景，设置 globalAlpha 为 0.2 后，在上面画一系列半径递增的半透明圆。最终结果是一个径向渐变效果。圆叠加得越更多，原先所画的圆的透明度会越低。通过增加循环次数，画更多的圆，背景图的中心部分会完全消失。

注意:

这个例子在 Firefox 1.5 beta 1 里是行不通的。你需要 nightly branch build 或者等待新版本发布来实践这个效果。

这个例子在 Safari 下可能由于颜色值无效而达不到效果。例子里是 '#09F' 是不符合规范要求的，不过 Firefox 是认识这种格式的。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  // draw background
 ctx.fillStyle = '#FD0';
 ctx.fillRect(0,0,75,75);
 ctx.fillStyle = '#6C0';
 ctx.fillRect(75,0,75,75);
 ctx.fillStyle = '#09F';
 ctx.fillRect(0,75,75,75);
 ctx.fillStyle = '#F30';
 ctx.fillRect(75,75,150,150);
 ctx.fillStyle = '#FFF';

  // set transparency value
 ctx.globalAlpha = 0.2;

  // Draw semi transparent circles
  for (var i=0;i<7;i++){
 ctx.beginPath();
 ctx.arc(75,75,10+10*i,0,Math.PI*2,true);
 ctx.fill();
  }
}
```

rgba()示例

In this second example I've done something similar to the one above, but instead of drawing circles on top of each other, I've drawn small rectangles with increasing opacity. Using rgba() gives you a little more control and flexibility because we can set the fill and stroke style individually.

第二个例子和上面那个类似，不过不是画圆，而是画矩形。这里还可以看出，rgba() 可以分别设置轮廓和填充样式，因而具有更好的可操作性和使用弹性。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  // Draw background
 ctx.fillStyle = 'rgb(255,221,0)';
 ctx.fillRect(0,0,150,37.5);
 ctx.fillStyle = 'rgb(102,204,0)';
 ctx.fillRect(0,37.5,150,37.5);
 ctx.fillStyle = 'rgb(0,153,255)';
 ctx.fillRect(0,75,150,37.5);
 ctx.fillStyle = 'rgb(255,51,0)';
 ctx.fillRect(0,112.5,150,37.5);

  // Draw semi transparent rectangles
  for (var i=0;i<10;i++){
 ctx.fillStyle = 'rgba(255,255,255,'+(i+1)/10+')';
    for (var j=0;j<4;j++){
 ctx.fillRect(5+i*14,5+j*37.5,14,27.5)
    }
  }
}
```

线型 Line styles

There are several properties which allow us to style lines.

可以通过一系列属性来设置线的样式。

lineWidth = value

lineCap = type

lineJoin = type

miterLimit = value

I could describe these in detail, but it will probably be made clearer by just looking at the examples below.

我会详细介绍这些属性，不过通过以下的样例可能会更加容易理解。

lineWidth 属性的例子

This property sets the current line thickness. Values must be positive numbers. By default this value is set to 1.0 units.

这个属性设置当前绘线的粗细。属性值必须为正数。默认值是 1.0。

The line width is the thickness of the stroke centered on the given path. In other words, the area that's drawn extends to half the line width on either side of the path. Because canvas coordinates do not directly reference pixels, special care must be taken to obtain crisp horizontal and vertical lines.

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。

In the example below, 10 straight lines are drawn with increasing line widths. The line on the far left is 1.0 units wide. However, the leftmost and all other odd-integer-width thickness lines do not appear crisp, because of the path's positioning.

在下面的例子中，用递增的宽度绘制了 10 条直线。最左边的线宽 1.0 单位。并且，最左边的以及所有宽度为奇数的线并不能精确呈现，这就是因为路径的定位问题。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  for (var i = 0; i < 10; i++){
 ctx.lineWidth = 1+i;
 ctx.beginPath();
 ctx.moveTo(5+i*14,5);
 ctx.lineTo(5+i*14,140);
 ctx.stroke();
  }
}
```

Obtaining crisp lines requires understanding how paths are stroked. In the images below, the grid represents the canvas coordinate grid. The squares between gridlines are actual on-screen pixels. In the first grid image below, a rectangle from (2,1) to (5,5) is filled. The entire area between them (light red) falls on pixel boundaries, so the resulting filled rectangle will have crisp edges.

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。

If you consider a path from (3,1) to (3,5) with a line thickness of 1.0, you end up with the situation in the second image. The actual area to be filled (dark blue) only extends halfway into the pixels on either side of the path. An approximation of this has to be rendered, which means that those pixels being only partially shaded, and results in the entire area (the light blue and dark blue) being filled in with a color only half as dark as the actual stroke color. This is what happens with the 1.0 width line in the previous example code.

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）。这就是上例中为何宽度为 1.0 的线并不准确的原因。

To fix this, you have to be very precise in your path creation. Knowing that a 1.0 width line will extend half a unit to either side of the path, creating the path from (3.5,1) to (3.5,5) results in the situation in the third image -- the 1.0 line width ends up completely and precisely filling a single pixel vertical line.

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。

For even-width lines, each half ends up being an integer amount of pixels, so you want a path that is between pixels (that is, (3,1) to (3,5)), instead of down the middle of pixels. Also, be aware that in our vertical line example, the Y position still referenced an integer gridline position -- if it hadn't, we would see pixels with half coverage at the endpoints.

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。同样，注意到那个例子的垂直线条，其 Y 坐标刚好落在网格线上，如果不是的话，端点上同样会出现半渲染的像素点。

While slightly painful when initially working with scalable 2D graphics, paying attention to the pixel grid and the position of paths ensures that your drawings will look correct regardless of scaling or any other transformations involved. A 1.0-width vertical line drawn at the correct position will become a crisp 2-pixel line when scaled up by 2, and will appear at the correct position.

虽然开始处理可缩放的 2D 图形时会有点小痛苦，但是及早注意到像素网格与路径位置之间的关系，可以确保图形在经过缩放或者其它任何变形后都可以保持看上去蛮好：线宽为 1.0 的垂线在放大 2 倍后，会变成清晰的线宽为 2.0，并且出现在它应该出现的位置上。

lineCap 属性的例子

The lineCap property determines how the end points of every line are drawn. There are three possible values for this property and those are: butt, round and square. By default this property is set to butt.

属性 lineCap 的指决定了线段端点显示的样子。它可以为下面的三种的其中之一：butt，round 和 square。默认是 butt。

In this example, I've drawn three lines, each with a different value for the lineCap property. I also added two guides to see the exact differences between the three. Each of these lines starts and ends exactly on these guides.

这个例子里面，我绘制了三条直线，分别赋予不同的 lineCap 值。还有两条辅助线，为了可以看得更清楚它们之间的区别，三条线的起点终点都落在辅助线上。

The line on the left uses the default butt option. You'll notice that it's drawn completely flush with the guides. The second is set to use the round option. This adds a semicircle to the end that has a radius half the width of the line. The line on the right uses the square option. This adds a box with an equal width and half the height of the line thickness.

最左边的线用了默认的 butt 。可以注意到它是与辅助线齐平的。中间的是 round 的效果，端点处加上了半径为一半线宽的半圆。右边的是 square 的效果，端点出加上了等宽且高度为一半线宽的方块。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
```

```
 var lineCap = ['butt','round','square'];

 // Draw guides
ctx.strokeStyle = '#09f';
ctx.beginPath();
ctx.moveTo(10,10);
ctx.lineTo(140,10);
ctx.moveTo(10,140);
ctx.lineTo(140,140);
ctx.stroke();

 // Draw lines
ctx.strokeStyle = 'black';
 for (var i=0;i<lineCap.length;i++){
ctx.lineWidth = 15;
ctx.lineCap = lineCap[i];
ctx.beginPath();
ctx.moveTo(25+i*50,10);
ctx.lineTo(25+i*50,140);
ctx.stroke();
 }
}
```

lineJoin 属性的例子

The lineJoin property determines how two connecting lines in a shape are joined together. There are three possible values for this property: round, bevel and miter. By default this property is set to miter.

lineJoin 的属性值决定了图形中两线段连接处所显示的样子。它可以是这三种之一：round, bevel 和 miter。默认是 miter。

Again I've drawn three different paths, each with a different lineJoin property setting. The top path uses the round option. This setting rounds off the corners of a shape. The radius for these rounded corners is equal to the line width. The second line uses the bevel option and the line at the bottom uses the miter option. When set to miter, lines are joined by extending the outside edges to connect at a single point. This setting is effected by the miterLimit property which is explained below.

这里我同样用三条折线来做例子，分别设置不同的 lineJoin 值。最上面一条是 round 的效果，边角处被磨圆了，圆的半径等于线宽。中间和最下面一条分别是 bevel 和 miter 的效果。当值是 miter 的时候，线段会在连接处外侧延伸直至交于一点，延伸效果受到下面将要介绍的 miterLimit 属性的制约。

查看示例

```
view plainprint?
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  var lineJoin = ['round','bevel','miter'];
 ctx.lineWidth = 10;
  for (var i=0;i<lineJoin.length;i++){
 ctx.lineJoin = lineJoin[i];
 ctx.beginPath();
 ctx.moveTo(-5,5+i*40);
 ctx.lineTo(35,45+i*40);
 ctx.lineTo(75,5+i*40);
 ctx.lineTo(115,45+i*40);
 ctx.lineTo(155,5+i*40);
 ctx.stroke();
  }
}
```

miterLimit 属性的演示例子

As you've seen in the previous example, when joining two lines with the miter option, the outside edges of the two joining lines are extended up to the point where they meet. For lines which are at large angles with each other, this point is not far from the inside connection point. However, when the angles between each line decreases, the distance (miter length) between these points increases exponentially.

就如上一个例子所见的应用 miter 的效果，线段的外侧边缘会延伸交汇于一点上。线段直接夹角比较大的，交点不会太远，但当夹角减少时，交点距离会呈指数级增大。

The miterLimit property determines how far the outside connection point can be placed from the inside connection point. If two lines exceed this value, a bevel join will be drawn.

miterLimit 属性就是用来设定外延交点与连接点的最大距离，如果交点距离大于此值，连接效果会变成了 bevel。

I've made a little demo in which you can set miterLimit dynamically and see how this effects the shapes on the canvas. The blue lines show where the start and endpoints for each of the lines in the zig-zag pattern are.

我做了一个演示页面，你可以动手改变 miterLimit 的值，观察其影响效果。蓝色辅助线显示锯齿折线段的起点与终点所在的位置。

去看看演示

渐变 Gradients

Just like any normal drawing program, we can fill and stroke shapes using linear and radial gradients. We create a canvasGradient object by using one of the following methods. We use this object to assign it to the fillStyle or strokeStyle properties.

就好像一般的绘图软件一样，我们可以用线性或者径向的渐变来填充或描边。我们用下面的方法新建一个 canvasGradient 对象，并且赋给图形的 fillStyle 或 strokeStyle 属性。

createLinearGradient(x1,y1,x2,y2)

createRadialGradient(x1,y1,r1,x2,y2,r2)

The createLinearGradient method takes four arguments representing the starting point (x1,y1) and end point (x2,y2) of the gradient.

The createRadialGradient method takes six arguments. The first three arguments define a circle with coordinates (x1,y1) and radius r1 and the second a circle with coordinates (x2,y2) and radius r2.

createLinearGradient 方法接受 4 个参数，表示渐变的起点 (x1,y1) 与终点 (x2,y2)。

createRadialGradient 方法接受 6 个参数，前三个定义一个以 (x1,y1) 为原点，半径为 r1 的圆，后三个参数则定义另一个以 (x2,y2) 为原点，半径为 r2 的圆。

view plainprint?

```
var lineargradient = ctx.createLinearGradient(0,0,150,150);
```

var radialgradient = ctx.createRadialGradient(75,75,0,75,75,100);

Once we've created a canvasGradient object we can assign colors to it by using the addColorStop method.

创建出 canvasGradient 对象后，我们就可以用 addColorStop 方法给它上色了。

addColorStop(position, color)

This method takes two arguments. The position must be a number between 0.0 and 1.0 and defines the relative position of the color in the gradient. Setting this to 0.5 for instance would place the color precisely in the middle of the gradient. The color argument must be a string representing a CSS color (ie #FFF, rgba(0,0,0,1),etc).

addColorStop 方法接受 2 个参数，position 参数必须是一个 0.0 与 1.0 之间的数值，表示渐变中颜色所在的相对位置。例如，0.5 表示颜色会出现在正中间。color 参数必须是一个有效的 CSS 颜色值（如 #FFF， rgba(0,0,0,1)，等等）。

You can add as many color stops to a gradient as you need. Below is a very simple linear gradient from white to black.

你可以根据需要添加任意多个色标（color stops）。下面是最简单的线性黑白渐变的例子。

view plainprint?

```
var lineargradient = ctx.createLinearGradient(0,0,150,150);
lineargradient.addColorStop(0,'white');
lineargradient.addColorStop(1,'black');
```

createLinearGradient 的例子

In this example, I've created two different gradients. In the first, I create the background gradient. As you can see, I've assigned two colors at the same position. You do this to make very sharp color transitions - in this case from white to green. Normally, it doesn't matter in what order you define the color stops, but in this special case, it does significantly. If you keep the assignments in the order you want them to appear, this won't be a problem.

本例中，我弄了两种不同的渐变。第一种是背景色渐变，你会发现，我给同一位置设置了两种颜色，你也可以用这来实现突变的效果，就像这里从白色到绿色的突变。一般情况下，色标的定义是无所谓顺序的，但是色标位置重复时，顺序就变得非常重要了。所以，保持色标定义顺序和它理想的顺序一致，结果应该没什么大问题。

In the second gradient, I didn't assign the starting color (at position 0.0) since it wasn't strictly necessary. Assigning the black color at position 0.5 automatically makes the gradient, from the start to this stop, black.

第二种渐变，我并不是从 0.0 位置开始定义色标，因为那并不是那么严格的。在 0.5 处设一黑色色标，渐变会默认认为从起点到色标之间都是黑色。

As you can see here, both the strokeStyle and fillStyle properties can accept a canvasGradient object as valid input.

你会发现，strokeStyle 和 fillStyle 属性都可以接受 canvasGradient 对象。

查看示例

```
view plainprint?
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

  // Create gradients
  var lingrad = ctx.createLinearGradient(0,0,0,150);
 lingrad.addColorStop(0, '#00ABEB');
 lingrad.addColorStop(0.5, '#fff');
 //lingrad.addColorStop(0.5, '#26C000');
 //lingrad.addColorStop(1, '#fff');

  var lingrad2 = ctx.createLinearGradient(0,50,0,95);
 lingrad2.addColorStop(0.5, '#000');
 lingrad2.addColorStop(1, 'rgba(0,0,0,0)');

  // assign gradients to fill and stroke styles
 ctx.fillStyle = lingrad;
 ctx.strokeStyle = lingrad2;
```

```
  // draw shapes
 ctx.fillRect(10,10,130,130);
 ctx.strokeRect(50,50,50,50);

}
```

createRadialGradient 的例子

In this example, I've defined four different radial gradients. Because we have control over the start and closing points of the gradient, we can achieve more complex effects than we would normally have in the 'classic' radial gradients we see in, for instance, Photoshop. (i.e. a gradient with a single center point where the gradient expands outward in a circular shape.)

这个例子，我定义了 4 个不同的径向渐变。由于可以控制渐变的起始与结束点，所以我们可以实现一些比（如在 Photoshop 中所见的）经典的径向渐变更为复杂的效果。（经典的径向渐变是只有一个中心点，简单地由中心点向外围的圆形扩张）

In this case, I've offset the starting point slightly from the end point to achieve a spherical 3D effect. It's best to try to avoid letting the inside and outside circles overlap because this results in strange effects which are hard to predict.

这里，我让起点稍微偏离终点，这样可以达到一种球状 3D 效果。但最好不要让里圆与外圆部分交叠，那样会产生什么效果就真是不得而知了。

The last color stop in each of the four gradients uses a fully transparent color. If you want to have a nice transition from this to the previous color stop, both colors should be equal. This isn't very obvious from the code because I've used two different CSS color methods, but in the first gradient #019F62 = rgba(1,159,98,1)

4 个径向渐变效果的最后一个色标都是透明色。如果想要两色标直接的过渡柔和一些，只要两个颜色值一致就可以了。代码里面看不出来，是因为我用了两种不同的颜色表示方法，但其实是相同的，#019F62 = rgba(1,159,98,1)。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
```

```
 // Create gradients
 var radgrad = ctx.createRadialGradient(45,45,10,52,50,30);
radgrad.addColorStop(0, '#A7D30C');
radgrad.addColorStop(0.9, '#019F62');
radgrad.addColorStop(1, 'rgba(1,159,98,0)');

 var radgrad2 = ctx.createRadialGradient(105,105,20,112,120,50);
radgrad2.addColorStop(0, '#FF5F98');
radgrad2.addColorStop(0.75, '#FF0188');
radgrad2.addColorStop(1, 'rgba(255,1,136,0)');

 var radgrad3 = ctx.createRadialGradient(95,15,15,102,20,40);
radgrad3.addColorStop(0, '#00C9FF');
radgrad3.addColorStop(0.8, '#00B5E2');
radgrad3.addColorStop(1, 'rgba(0,201,255,0)');

 var radgrad4 = ctx.createRadialGradient(0,150,50,0,140,90);
radgrad4.addColorStop(0, '#F4F201');
radgrad4.addColorStop(0.8, '#E4C700');
radgrad4.addColorStop(1, 'rgba(228,199,0,0)');

 // draw shapes
ctx.fillStyle = radgrad4;
ctx.fillRect(0,0,150,150);
ctx.fillStyle = radgrad3;
ctx.fillRect(0,0,150,150);
ctx.fillStyle = radgrad2;
ctx.fillRect(0,0,150,150);
ctx.fillStyle = radgrad;
ctx.fillRect(0,0,150,150);
}
```

图案 Patterns

In one of the examples on the previous page, I used a series of loops to create a pattern of images. There is, however, a much simpler method: the createPattern method.

上一节的一个例子里面，我用了循环来实现图案的效果。其实，有一个更加简单的方法：createPattern。

createPattern(image,type)

This method takes two arguments. Image is either a reference to an Image object or a different canvas element. Type must be a string containing one of the following values: repeat, repeat-x, repeat-y and no-repeat.

该方法接受两个参数。Image 可以是一个 Image 对象的引用，或者另一个 canvas 对象。Type 必须是下面的字符串值之一：repeat，repeat-x，repeat-y 和 no-repeat。

注意: 用 canvas 对象作为 Image 参数在 Firefox 1.5 (Gecko 1.8) 中是无效的。

We use this method to create a Pattern object which is very similar to the gradient methods we've seen above. Once we've created a pattern, we can assign it to the fillStyle or strokeStyle properties.

图案的应用跟渐变很类似的，创建出一个 pattern 之后，赋给 fillStyle 或 strokeStyle 属性即可。

view plainprint?

```
var img = new Image();
img.src = 'someimage.png';
var ptrn = ctx.createPattern(img,'repeat');
```

Note: Unlike the drawImage method, you must make sure the image you use is loaded before calling this method or the pattern may be drawn incorrectly.

注意：与 drawImage 有点不同，你需要确认 image 对象已经装载完毕，否则图案可能效果不对的。

Note: Firefox currently only supports the repeat property. If you assign anything else, you won't see any changes.

注意：Firefox 目前只支持属性值 repeat 。如果赋其它值，什么效果都没有的。

createPattern 的例子

In this last example, I created a pattern which I assigned to the fillStyle property. The only thing worth noting is the use of the Image object onload handler. This is to make sure the image is loaded before it is assigned to the pattern.

这最后的例子，我创建一个图案然后赋给了 fillStyle 属性。值得一提的是，使用 Image 对象的 onload handler 来确保设置图案之前图像已经装载完毕。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

  // create new image object to use as pattern
  var img = new Image();
  img.src = 'images/wallpaper.png';
  img.onload = function(){

    // create pattern
    var ptrn = ctx.createPattern(img,'repeat');
 ctx.fillStyle = ptrn;
 ctx.fillRect(0,0,150,150);

  }
}
```

阴影 Shadows

Firefox 3.5 note

Firefox 3.5 中支持阴影效果.

Firefox 3.5 (Gecko 1.9.1) introduced support for shadows in canvases. Using shadows involves just four properties:

Firefox 3.5 (Gecko 1.9.1) 在 canvas 中加入了对阴影的支持，就 4 个属性。

shadowOffsetX = float

shadowOffsetY = float

shadowBlur = float

shadowColor = color

shadowOffsetX and shadowOffsetY indicate how far the shadow should extend from the object in the X and Y directions; these values aren't affected by the current transformation matrix. Use negative values to cause the shadow to extend up or to the left, and positive values to cause the shadow to extend down or to the right. These are both 0 by default.

shadowOffsetX 和 shadowOffsetY 用来设定阴影在 X 和 Y 轴的延伸距离，它们是不受变换矩阵所影响的。负值表示阴影会往上或左延伸，正值则表示会往下或右延伸，他们默认都是 0。

shadowBlur indicates the size of the blurring effect; this value doesn't correspond to a number of pixels and is not affected by the current transformation matrix. The default value is 0.

shadowBlur 用于设定阴影的模糊程度，其数值并不跟像素数量挂钩，也不受变换矩阵的影响，默认为 0。

shadowColor is a standard CSS color value indicating the color of the shadow effect; by default, it is fully-transparent black.

shadowColor 用于设定阴影效果的延伸，值可以是标准的 CSS 颜色值，默认是全透明的黑色。

文字阴影的例子

This example draws a text string with a shadowing effect.

这个例子绘制了带阴影效果的文字。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

 ctx.shadowOffsetX = 2;
 ctx.shadowOffsetY = 2;
 ctx.shadowBlur = 2;
 ctx.shadowColor = "rgba(0, 0, 0, 0.5)";

  ctx.font = "20px Times New Roman";
 ctx.fillStyle = "Black";
 ctx.fillText("Sample String", 5, 30);
}
```

1.5 变型

状态的保存和恢复 Saving and restoring state

Before we look at the transformation methods, I'll introduce two other methods which are indispensable once you start generating ever more complex drawings.

在了解变形之前，我先介绍一下两个你一旦开始绘制复杂图形就必不可少的方法。

save()

restore()

The canvas save and restore methods are used to save and retrieve the canvas state. The canvas drawing state is basically a snapshot of all the styles and transformations that have been applied. Both methods take no parameters.

save 和 restore 方法是用来保存和恢复 canvas 状态的，都没有参数。Canvas 的状态就是当前画面应用的所有样式和变形的一个快照。

Canvas states are stored on a stack. Every time the save method is called, the current drawing state is pushed onto the stack. A drawing state consists of

Canvas 状态是以堆(stack)的方式保存的，每一次调用 save 方法，当前的状态就会被推入堆中保存起来。这种状态包括：

The transformations that have been applied (i.e. translate, rotate and scale - see below).

当前应用的变形（即移动，旋转和缩放，见下）

The values of strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation properties.

strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation 的值

The current clipping path, which we'll see in the next section.

当前的裁切路径（clipping path），会在下一节介绍

You can call the save method as many times as you like.

你可以调用任意多次 save 方法。

Every time the restore method is called, the last saved state is returned from the stack and all saved settings are restored.

每一次调用 restore 方法，上一个保存的状态就从堆中弹出，所有设定都恢复。

save 和 restore 的应用例子

This example tries to illustrate how the stack of drawing states functions by drawing a set of consecutive rectangles.

我们尝试用这个连续矩形的例子来描述 canvas 的状态堆是如何工作的。

The first step is to draw a large rectangle with the default settings. Next we save this state and make changes to the fill color. We then draw the second and smaller blue rectangle and save the state. Again we change some drawing settings and draw the third semi-transparent white rectangle.

第一步是用默认设置画一个大四方形，然后保存一下状态。改变填充颜色画第二个小一点的蓝色四方形，然后再保存一下状态。再次改变填充颜色绘制更小一点的半透明的白色四方形。

So far this is pretty similar to what we've done in previous sections. However once we call the first restore statement, the top drawing state is removed from the stack, and settings are restored. If we hadn't saved the state using save, we would need to change the fill color and transparency manually in order to return to the previous state. This would be easy for two properties, but if we have more than that, our code would become very long, very fast.

到目前为止所做的动作和前面章节的都很类似。不过一旦我们调用 restore，状态堆中最后的状态会弹出，并恢复所有设置。如果不是之前用 save 保存了状态，那么我们就需要手动改变设置来回到前一个状态，这个对于两三个属性的时候还是适用的，一旦多了，我们的代码将会猛涨。

When the second restore statement is called, the original state (the one we set up before the first call to save) is restored and the last rectangle is once again drawn in black.

当第二次调用 restore 时，已经恢复到最初的状态，因此最后是再一次绘制出一个黑色的四方形。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

  ctx.fillRect(0,0,150,150);   // Draw a rectangle with default settings
  ctx.save();                  // Save the default state

  ctx.fillStyle = '#09F'       // Make changes to the settings
  ctx.fillRect(15,15,120,120); // Draw a rectangle with new settings

  ctx.save();                  // Save the current state
  ctx.fillStyle = '#FFF'       // Make changes to the settings
  ctx.globalAlpha = 0.5;
  ctx.fillRect(30,30,90,90);   // Draw a rectangle with new settings

  ctx.restore();               // Restore previous state
  ctx.fillRect(45,45,60,60);   // Draw a rectangle with restored settings

  ctx.restore();               // Restore original state
  ctx.fillRect(60,60,30,30);   // Draw a rectangle with restored settings
}
```

移动 Translating

The first of the transformation methods we'll look at is translate. This method is used to move the canvas and its origin to a different point in the grid.

来变形，我们先介绍 translate 方法，它用来移动 canvas 和它的原点到一个不同的位置。

translate(x, y)

This method takes two arguments. x is the amount the canvas is moved to the left or right, and y is the amount it's moved up or down (illustrated by the image on the right).

translate 方法接受两个参数。x 是左右偏移量，y 是上下偏移量，如右图所示。

It's a good idea to save the canvas state before doing any transformations. In most cases, it is just easier to call the restore method than having to do a reverse translation to return to the original state. Also if you're translating inside a loop and don't save and restore the canvas state, you might end up missing part of your drawing, because it was drawn outside the canvas edge.

在做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。

translate 的例子

This example demonstrates some of the benefits of translating the canvas origin. I've made a function drawSpirograph that draws spirograph patterns. These are drawn around the origin. If I didn't use the translate function, I would only see a quarter of the pattern on the canvas. The translate method also gives me the freedom to place it anywhere on the canvas without having to manually adjust coordinates in the spirograph function. This makes it a little easier to understand and use.

这个例子显示了一些移动 canvas 原点的好处。我创建了一个 drawSpirograph 方法用来绘制螺旋（spirograph）图案，那是围绕原点绘制出来的。如果不使用 translate 方法，那么只能看见其中的四分之一。translate 同时让我可以任意放置这些图案，而不需要在 spirograph 方法中手工调整坐标值，既好理解也方便使用。

In the draw function I call the drawSpirograph nine times using two for loops. In each loop the canvas is translated, the spirograph is drawn, and the canvas is returned back to its original state.

我在 draw 方法中调用 drawSpirograph 方法 9 次，用了 2 层循环。每一次循环，先移动 canvas，画螺旋图案，然后恢复早原始状态。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.fillRect(0,0,300,300);
  for (var i=0;i<3;i++) {
    for (var j=0;j<3;j++) {
      ctx.save();
      ctx.strokeStyle = "#9CFF00";
      ctx.translate(50+j*100,50+i*100);
      drawSpirograph(ctx,20*(j+2)/(j+1),-8*(i+3)/(i+1),10);
      ctx.restore();
    }
  }
}
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();
}
```

旋转 Rotating

The second transformation method is rotate. We use it to rotate the canvas around the current origin.

第二个介绍 rotate 方法，它用于以原点为中心旋转 canvas。

rotate(angle)

This method only takes one parameter and that's the angle the canvas is rotated by. This is a clockwise rotation measured in radians (illustrated in the image on the right).

这个方法只接受一个参数：旋转的角度(angle)，它是顺时针方向的，以弧度为单位的值。

The rotation center point is always the canvas origin. To change the center point, we will need to move the canvas by using the translate method.

旋转的中心点始终是 canvas 的原点，如果要改变它，我们需要用到 translate 方法。

rotate 的例子

In the example, you can see on the right, I used the rotate method to draw shapes in a circular pattern. You could also have calculated the individual x and y coordinates (x = r*Math.cos(a); y = r*Math.sin(a)). In this case it doesn't really matter which method you choose, because we're drawing circles. Calculating the coordinates results in only rotating the center positions of the circles and not the circles themselves, while using rotate results in both, but of course circles look the same no matter how far they are rotated about their centers.

在这个例子里，见右图，我用 rotate 方法来画圆并构成圆形图案。当然你也可以分别计算出 x 和 y 坐标(x = r*Math.cos(a); y = r*Math.sin(a))。这里无论用什么方法都无所谓的，因为我们画的是圆。计算坐标的结果只是旋转圆心位置，而不是圆本身。即使用 rotate 旋转两者，那些圆看上去还是一样的，不管它们绕中心旋转有多远。

Again we have two loops. The first determines the number of rings, and the second determines the number of dots drawn in each ring. Before drawing each ring, I save the canvas state, so I can easily retrieve it. For each dot that is drawn, I rotate the canvas coordinate space by an angle that is determined by the number of dots in the ring. The innermost circle has six dots, so in each step, I rotate over an angle of 360/6 = 60 degrees. With each additional ring, the number of dots is doubled, and the angle in turn is halved.

这里我们又用到了两层循环。第一层循环决定环的数量，第二层循环决定每环有多少个点。每环开始之前，我都保存一下 canvas 的状态，这样恢复起来方便。每次画圆点，我都以一定夹角来旋转 canvas，而这个夹角则是由环上的圆点数目的决定的。最里层的环有 6 个圆点，这样，每次旋转的

夹角就是 360/6 = 60 度。往外每一环的圆点数目是里面一环的 2 倍，那么每次旋转的夹角随之减半。

查看示例

```
view plainprint?
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.translate(75,75);

  for (var i=1;i<6;i++){ // Loop through rings (from inside to out)
    ctx.save();
    ctx.fillStyle = 'rgb('+(51*i)+','+(255-51*i)+',255)';

    for (var j=0;j<i*6;j++){ // draw individual dots
      ctx.rotate(Math.PI*2/(i*6));
      ctx.beginPath();
      ctx.arc(0,i*12.5,5,0,Math.PI*2,true);
      ctx.fill();
    }

    ctx.restore();
  }
}
```

缩放 Scaling

The next transformation method is scaling. We use it to increase or decrease the units in our canvas grid. This can be used to draw scaled down or enlarged shapes and bitmaps.

接着是缩放。我们用它来增减图形在 canvas 中的像素数目，对形状，位图进行缩小或者放大。

scale(x, y)

This method takes two parameters. x is the scale factor in the horizontal direction and y is the scale factor in the vertical direction. Both parameters must be positive numbers. Values smaller than 1.0 reduce the unit size and values larger than 1.0 increase the unit size. Setting the scaling factor to precisely 1.0 doesn't affect the unit size.

scale 方法接受两个参数。x,y 分别是横轴和纵轴的缩放因子，它们都必须是正值。值比 1.0 小表示缩小，比 1.0 大则表示放大，值为 1.0 时什么效果都没有。

By default one unit on the canvas is exactly one pixel. If we apply, for instance, a scaling factor of 0.5, the resulting unit would become 0.5 pixels and so shapes would be drawn at half size. In a similar way setting the scaling factor to 2.0 would increase the unit size and one unit now becomes two pixels. This results in shapes being drawn twice as large.

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。

scale 的例子

In this last example I've used the spirograph function from one of the previous examples to draw nine shapes with different scaling factors. The top left shape has been drawn with no scaling applied. The yellow shapes to the right both have a uniform scaling factor (the same value for x and y parameters). If you look at the code below you'll see that I've used the scale method twice with equal parameter values for the second and third spirograph. Because I didn't restore the canvas state, the third shape is drawn with a scaling factor of 0.75 × 0.75 = 0.5625.

这最后的例子里，我再次启用前面曾经用过的 spirograph 方法，来画 9 个图形，分别赋予不同的缩放因子。左上角的图形是未经缩放的。黄色图案从左到右应用了统一的缩放因子（x 和 y 参数值是一致的）。看下面的代码，你可以发现，我在画第二第三个图案时 scale 了两次，中间没有 restore canvas 的状态，因此第三个图案的缩放因子其实是 0.75 × 0.75 = 0.5625。

The second row of blue shapes have a non-uniform scaling applied in a vertical direction. Each of the shapes has the x scaling factor set to 1.0 which means no scaling. The y scaling factor is set to 0.75. This results in the three shapes being squashed down. The original circular shape has now become an ellipse. If you look closely you'll see that the line width has also been reduced in the vertical direction.

第二行蓝色图案堆垂直方向应用了不统一的缩放因子，每个图形 x 方向上的缩放因子都是 1.0，意味着不缩放，而 y 方向缩放因子是 0.75，得出来的结果是，图案被依次压扁了。原来的圆形图案变成了椭圆，如果细心观察，还可以发现在垂直方向上的线宽也减少了。

The third row of green shapes is similar to the one above but now I've applied a scaling in the horizontal direction.

第三行的绿色图案与第二行类似，只是缩放限定在横轴方向上了。

查看示例

view plainprint?

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.strokeStyle = "#fc0";
  ctx.lineWidth = 1.5;
  ctx.fillRect(0,0,300,300);

  // Uniform scaling
  ctx.save()
  ctx.translate(50,50);
  drawSpirograph(ctx,22,6,5);  // no scaling

  ctx.translate(100,0);
  ctx.scale(0.75,0.75);
  drawSpirograph(ctx,22,6,5);

  ctx.translate(133.333,0);
  ctx.scale(0.75,0.75);
  drawSpirograph(ctx,22,6,5);
  ctx.restore();

  // Non-uniform scaling (y direction)
  ctx.strokeStyle = "#0cf";
  ctx.save()
  ctx.translate(50,150);
  ctx.scale(1,0.75);
  drawSpirograph(ctx,22,6,5);

  ctx.translate(100,0);
  ctx.scale(1,0.75);
  drawSpirograph(ctx,22,6,5);

  ctx.translate(100,0);
  ctx.scale(1,0.75);
  drawSpirograph(ctx,22,6,5);
  ctx.restore();

  // Non-uniform scaling (x direction)
  ctx.strokeStyle = "#cf0";
  ctx.save()
  ctx.translate(50,250);
  ctx.scale(0.75,1);
  drawSpirograph(ctx,22,6,5);

  ctx.translate(133.333,0);
  ctx.scale(0.75,1);
  drawSpirograph(ctx,22,6,5);

  ctx.translate(177.777,0);
  ctx.scale(0.75,1);
  drawSpirograph(ctx,22,6,5);
  ctx.restore();

}
```
变形 Transforms

The final transformation methods allow modifications directly to the transformation matrix.

最后一个方法是允许直接对变形矩阵作修改。

transform(m11, m12, m21, m22, dx, dy)

This method must multiply the current transformation matrix with the matrix described by:

这个方法必须将当前的变形矩阵乘上下面的矩阵：

m11   m21   dx

m12   m22   dy

0 0 1

If any of the arguments are Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

如果任意一个参数是无限大，变形矩阵也必须被标记为无限大，否则会抛出异常。

setTransform(m11, m12, m21, m22, dx, dy)

This method must reset the current transform to the identity matrix, and then invoke the transform method with the same arguments. If any of the arguments are Infinity the transformation matrix must be marked as infinite instead of the method throwing an exception.

这个方法必须重置当前的变形矩阵为单位矩阵，然后以相同的参数调用 transform 方法。如果任意一个参数是无限大，那么变形矩阵也必须被标记为无限大，否则会抛出异常。

transform / setTransform 的例子

view plainprint?

```
function draw() {
  var canvas = document.getElementById("canvas");
  var ctx = canvas.getContext("2d");

  var sin = Math.sin(Math.PI/6);
  var cos = Math.cos(Math.PI/6);
  ctx.translate(200, 200);
  var c = 0;
  for (var i=0; i <= 12; i++) {
    c = Math.floor(255 / 12 * i);
    ctx.fillStyle = "rgb(" + c + "," + c + "," + c + ")";
    ctx.fillRect(0, 0, 100, 10);
    ctx.transform(cos, sin, -sin, cos, 0, 0);
  }

  ctx.setTransform(-1, 0, 0, 1, 200, 200);
  ctx.fillStyle = "rgba(255, 128, 255, 0.5)";
  ctx.fillRect(0, 50, 100, 100);
}
```

1.6 组合

组合 Compositing

In all of our previous examples, shapes were always drawn one on top of the other. This is more than adequate for most situations. This, for instance, limits in what order composite shapes are built up. We can however change this behaviour by setting the globalCompositeOperation property.

之前的例子里面，我们总是将一个图形画在另一个之上，大多数情况下，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。

globalCompositeOperation

We can not only draw new shapes behind existing shapes but we can also use it to mask off certain areas, clear sections from the canvas (not limited to rectangles like the clearRect method does) and more.

我们不仅可以在已有图形后面再画新图形，还可以用来遮盖，清除（比 clearRect 方法强劲得多）某些区域。

globalCompositeOperation = type

type is a string representing any one of twelve compositing operations. Each of the available types is described below.

type 是下面 12 种字符串值之一：

Note: In all of the examples below the blue square is drawn first and referred to as 'existing canvas content'. The red circle is drawn second and referred to as 'new shape'.

注意：下面所有例子中，蓝色方块是先绘制的，即“已有的 canvas 内容”，红色圆形是后面绘制，即“新图形”。

source-over (default)

This is the default setting and draws new shapes on top of the existing canvas content.

这是默认设置，新图形会覆盖在原有内容之上。

destination-over

New shapes are drawn behind the existing canvas content.

会在原有内容之下绘制新图形。

source-in

The new shape is drawn only where both the new shape and the destination canvas overlap. Everything else is made transparent

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。

destination-in

The existing canvas content is kept where both the new shape and existing canvas content overlap. Everything else is made transparent.

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。

source-out

The new shape is drawn where it doesn't overlap the existing canvas content.

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。

destination-out

The existing content is kept where it doesn't overlap the new shape.

原有内容中与新图形不重叠的部分会被保留。

source-atop

The new shape is only drawn where it overlaps the existing canvas content.

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。

destination-atop

The existing canvas is only kept where it overlaps the new shape. The new shape is drawn behind the canvas content.

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

lighter

Where both shapes overlap the color is determined by adding color values.

两图形中重叠部分作加色处理。

darker

Where both shapes overlap the color is determined by subtracting color values.

两图形中重叠的部分作减色处理。

xor

Shapes are made transparent where both overlap and drawn normal everywhere else.

重叠的部分会变成透明。

copy

Only draws the new shape and removes everything else.

只有新图形会被保留，其它都被清除掉。

Note: Currently the copy and darker settings don't do anything in the Gecko 1.8 based browsers (Firefox 1.5 betas, etc).

注意：copy 和 darker 属性值在 Gecko 1.8 型的浏览器（Firefox 1.5 betas，等等）上暂时还无效。

查看所有示例

裁切路径 Clipping paths

A clipping path is like a normal canvas shape but it acts as a mask to hide unwanted parts of shapes. This is visualized in the image on the right. The red star shape is our clipping path. Everything that falls outside of this path won't get drawn on the canvas.

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。如右图所示。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。

If we compare clipping paths to the globalCompositeOperation property we've seen above; settings that achieve more or less the same effect are source-in and source-atop. The most important differences between the two are that clipping paths are never actually drawn to the canvas and the clipping path is never affected by adding new shapes. This makes clipping paths ideal for drawing multiple shapes in a restricted area.

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。

In the chapter about Drawing shapes I only mentioned the stroke and fill methods, but there's a third method we can use with paths, called clip.

在 绘制图形 一章中，我只介绍了 stroke 和 fill 方法，这里介绍第三个方法 clip。

clip()

We use the clip method to create a new clipping path. By default the canvas element has a clipping path that's the exact same size as the canvas itself (i.e. no clipping occurs).

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）。

clip 的例子

In this example I'll be using a circular clipping path to restrict the drawing of a set of random stars to a particular region.

这个例子，我会用一个圆形的裁切路径来限制随机星星的绘制区域。

In the first few lines of code I draw a black rectangle the size of the canvas as a backdrop and translate the origin to the center. Below this I create the circular clipping path by drawing an arc. By calling the clip method the clipping path is created. Clipping paths are also part of the canvas save state. If we wanted to keep the original clipping path we could have saved the canvas state before creating the new one.

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。

Everything that's drawn after creating the clipping path will only appear inside that path. You can see this clearly in the linear gradient that's drawn next. After this a set of 50 randomly positioned and scaled stars is drawn (I'm using a custom function for this). Again the stars only appear inside the defined clipping path.

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。

查看示例

view plainprint?

```
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();
}
```

1.7 基本动画

基本的动画

Since we're using script to control canvas elements it's also very easy to make (interactive) animations. Unfortunately the canvas element was never designed to be used in this way (unlike Flash) so there are limitations.

由于我们是用脚本去操控 canvas 对象，这样要实现一些交互动画也是相当容易的。只不过，canvas 从来都不是专门为动画而设计的（不像 Flash），难免会有些限制。

Probably the biggest limitation is that once a shape gets drawn it stays that way. If we need to move it we have to redraw it and everything that was drawn before it. It takes a lot of time to redraw complex frames and the performance depends highly on the speed of the computer it's running on.

可能最大的限制就是图像一旦绘制出来，它就是一直保持那样了。如果需要移动它，我们不得不对所有东西（包括之前的）进行重绘。重绘是相当费时的，而且性能很依赖于电脑的速度。

基本动画的步骤 Basic animation steps

画一帧，你需要以下一些步骤：

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

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。

操控动画 Controlling an animation

Shapes are drawn to the canvas by using the canvas methods directly or calling custom functions. In normal circumstances we only see these results appear on the canvas when the script finishes execution. For instance it isn't possible to do an animation from within a for loop.

在 canvas 上绘制内容是用 canvas 提供的或者自定义的方法，而通常，我们仅仅在脚本执行结束后才能看见结果，比如说，在 for 循环里面做完成动画是不太可能的。

We need a way to execute our drawing functions over a period of time. There are two ways to control an animation like this. First there's the setInterval and setTimeout functions which can be used to call a specific function over a set period of time.

我们需要一些可以定时的执行重绘的方法。有两种方法可以实现这样的动画操控。首先可以通过 setInterval 和 setTimeout 方法来控制在设定的时间点上执行重绘。

view plainprint?

```
setInterval(animateShape,500);
setTimeout(animateShape,500);
```

If you don't want any user interaction it's best to use the setInterval function which repeatedly executes the supplied code. In the example above the animateShape function is executed every 500 miliseconds (half a second). The setTimeout function only executes once after the set amount of time.

如果你不需要任何交互操作，用 setInterval 方法定时执行重绘是最适合的了。在上面的例子（leegorous 不见了？）里 animateShape 方法每半秒执行一次。setTimeout 方法只会在预设时间点上执行操作。

The second method we can use to control an animation is user input. If we wanted to make a game we could use keyboard or mouse events to control the animation. By setting eventListeners we catch any user interaction and execute our animation functions.

第二个方法，我们可以利用用户输入来实现操控。如果需要做一个游戏，我们可以通过监听用户交互过程中触发的事件（如 keyboard，mouse）来控制动画效果。

In the examples below I'm using the first method to control the animation. At the bottom of this page are some links to examples which use the second.

下面的例子，我还使用第一种方式实现动画效果。页面底部有些链接，那些是应用第二种方法的例子。

动画例子 1

In this example I'm going to animate a mini simulation of our solar system.

这个例子里面，我会让一个小型的太阳系模拟系统动起来。

```
view plainprint?
var sun = new Image();
var moon = new Image();
var earth = new Image();
function init(){
  sun.src = 'images/sun.png';
  moon.src = 'images/moon.png';
  earth.src = 'images/earth.png';
  setInterval(draw,100);
}

function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');

  ctx.globalCompositeOperation = 'destination-over';
  ctx.clearRect(0,0,300,300); // clear canvas

  ctx.fillStyle = 'rgba(0,0,0,0.4)';
  ctx.strokeStyle = 'rgba(0,153,255,0.4)';
  ctx.save();
  ctx.translate(150,150);

  // Earth
  var time = new Date();
  ctx.rotate( ((2*Math.PI)/60)*time.getSeconds() + ((2*Math.PI)/60000)*time.getMilliseconds() );
  ctx.translate(105,0);
  ctx.fillRect(0,-12,50,24); // Shadow
  ctx.drawImage(earth,-12,-12);

  // Moon
  ctx.save();
  ctx.rotate( ((2*Math.PI)/6)*time.getSeconds() + ((2*Math.PI)/6000)*time.getMilliseconds() );
  ctx.translate(0,28.5);
  ctx.drawImage(moon,-3.5,-3.5);
  ctx.restore();

  ctx.restore();

  ctx.beginPath();
  ctx.arc(150,150,105,0,Math.PI*2,false); // Earth orbit
  ctx.stroke();

  ctx.drawImage(sun,0,0,300,300);
}
```

Canvas sun.png

source image sun

Canvas earth.png

source image earth

Canvas moon.png

source image moon

动画例子 2

view plainprint?

```
function init(){
  clock();
  setInterval(clock,1000);
```

```
}
function clock(){
  var now = new Date();
  var ctx = document.getElementById('canvas').getContext('2d');
  ctx.save();
  ctx.clearRect(0,0,150,150);
  ctx.translate(75,75);
  ctx.scale(0.4,0.4);
  ctx.rotate(-Math.PI/2);
  ctx.strokeStyle = "black";
  ctx.fillStyle = "white";
  ctx.lineWidth = 8;
  ctx.lineCap = "round";

  // Hour marks
  ctx.save();
  for (var i=0;i<12;i++){
    ctx.beginPath();
    ctx.rotate(Math.PI/6);
    ctx.moveTo(100,0);
    ctx.lineTo(120,0);
    ctx.stroke();
  }
  ctx.restore();

  // Minute marks
  ctx.save();
  ctx.lineWidth = 5;
  for (i=0;i<60;i++){
    if (i%5!=0) {
      ctx.beginPath();
      ctx.moveTo(117,0);
      ctx.lineTo(120,0);
      ctx.stroke();
    }
    ctx.rotate(Math.PI/30);
  }
  ctx.restore();

  var sec = now.getSeconds();
  var min = now.getMinutes();
  var hr  = now.getHours();
  hr = hr>=12 ? hr-12 : hr;

  ctx.fillStyle = "black";

  // write Hours
  ctx.save();
  ctx.rotate( hr*(Math.PI/6) + (Math.PI/360)*min + (Math.PI/21600)*sec )
  ctx.lineWidth = 14;
  ctx.beginPath();
  ctx.moveTo(-20,0);
  ctx.lineTo(80,0);
  ctx.stroke();
  ctx.restore();

  // write Minutes
  ctx.save();
  ctx.rotate( (Math.PI/30)*min + (Math.PI/1800)*sec )
  ctx.lineWidth = 10;
  ctx.beginPath();
  ctx.moveTo(-28,0);
  ctx.lineTo(112,0);
  ctx.stroke();
  ctx.restore();
```

```
  // Write seconds
  ctx.save();
  ctx.rotate(sec * Math.PI/30);
  ctx.strokeStyle = "#D40000";
  ctx.fillStyle = "#D40000";
  ctx.lineWidth = 6;
  ctx.beginPath();
  ctx.moveTo(-30,0);
  ctx.lineTo(83,0);
  ctx.stroke();
  ctx.beginPath();
  ctx.arc(0,0,10,0,Math.PI*2,true);
  ctx.fill();
  ctx.beginPath();
  ctx.arc(95,0,10,0,Math.PI*2,true);
  ctx.stroke();
  ctx.fillStyle = "#555";
  ctx.arc(0,0,3,0,Math.PI*2,true);
  ctx.fill();
  ctx.restore();

  ctx.beginPath();
  ctx.lineWidth = 14;
  ctx.strokeStyle = '#325FA2';
  ctx.arc(0,0,142,0,Math.PI*2,true);
  ctx.stroke();

  ctx.restore();
}
```

动画例子 3

This is the code for a left-to-right looping panoramic image scroller. Make sure the image is larger than the canvas.

这是一个从左到右滚动的全景动画的代码。请注意图片的大小需要比 canvas 大。

本例中用的是这张图片：

http://commons.wikimedia.org/wiki/File:Capitan_Meadows,_Yosemite_National_Park.jpg

```
view plainprint?
var img = new Image();

//User Variables
img.src = 'Capitan_Meadows,_Yosemite_National_Park.jpg';
var CanvasXSize = 800;
var CanvasYSize = 200;
var speed = 30; //lower is faster
var scale = 1.05;
var y = -4.5; //vertical offset
//End User Variables

var dx = 0.75;
var imgW = img.width*scale;
var imgH = img.height*scale;
var x = 0;
if (imgW > CanvasXSize) { x = CanvasXSize-imgW; } // image larger than canvas
var clearX;
var clearY;
if (imgW > CanvasXSize) { clearX = imgW; } // image larger than canvas
else { clearX = CanvasXSize; }
if (imgH > CanvasYSize) { clearY = imgH; } // image larger than canvas
else { clearY = CanvasYSize; }
var ctx;

function init() {
   //Get Canvas Element
   ctx = document.getElementById('canvas').getContext('2d');
```

```
    //Set Refresh Rate
    return setInterval(draw, speed);
}

function draw() {
    //Clear Canvas
    ctx.clearRect(0,0,clearX,clearY);
    //If image is <= Canvas Size
    if (imgW <= CanvasXSize) {
        //reset, start from beginning
        if (x > (CanvasXSize)) { x = 0; }
        //draw aditional image
        if (x > (CanvasXSize-imgW)) { ctx.drawImage(img,x-CanvasXSize+1,y,imgW,imgH); }
    }
    //If image is > Canvas Size
    else {
        //reset, start from beginning
        if (x > (CanvasXSize)) { x = CanvasXSize-imgW; }
        //draw aditional image
        if (x > (CanvasXSize-imgW)) { ctx.drawImage(img,x-imgW+1,y,imgW,imgH); }
    }
    //draw image
    ctx.drawImage(img,x,y,imgW,imgH);
    //amount to move
    x += dx;
}
```

html code. Canvas width and height should match the CanvasXSize, CanvasYSize.

view plainprint?

```
<body onload="init();">
<canvas id="canvas" width="800" height="200"></canvas>
```

其它例子

粒子喷泉 (Particle Fountain)

你可以用鼠标来控制物理粒子的产生。

Canvascape

一个 3D （第一人称射击）冒险游戏。

http://www.gradius-js.com

一个经典的 2D 太空射击游戏。

A Basic RayCaster

怎样使用键盘操控动画效果的好例子。

canvas adventure

同样是一个键盘操控动画的好例子。

An interactive Blob

有趣的泡泡。

Flying through a starfield

在星星，圆圈，方块间滑翔。

iGrapher

股票数据图表的例子。

1.8 参考资料：

1.8.1 https://developer.mozilla.org/cn/Canvas_tutorial

See document: Canvas_tutorial

Canvas 教程 Canvas tutorial - MDC

See document: Canvas_tutorial

Basic usage - MDC

See document: Basic_usage

Drawing shapes 绘制图形 - MDC

See document: Drawing_shapes

应用图像 Using images - MDC

See document: Using_images

运用样式与颜色 - MDC

See document: Applying_styles_and_colors

变形 Transformations - MDC

See document: Transformations

组合 Compositing - MDC

See document: Compositing

基本的动画 - MDC

See document: Basic_animations