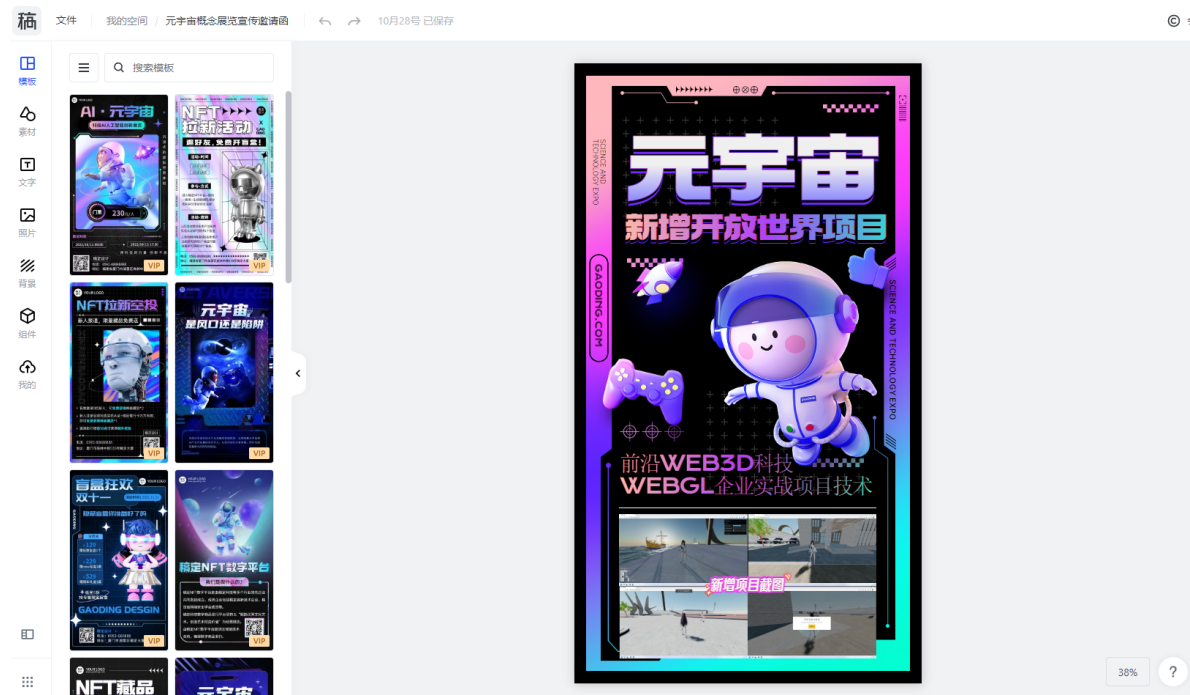


# 第五章 canvas路径和三角函数的故事

## 5.1 认识canvas

身为一个WEB开发人员，肯定都是想着能够开发出酷炫和激动人心的应用程序来。可以很多动画特效，例如黑客帝国的数字，彩色炫酷的例子动效。也可以实现各种图画面板，如实现类似于photoshop的web在线图像编辑。各种酷炫的表单等等。

### 案例




画布是H5中一个重要的概念，它面向开发人员提供了非常底层的绘图接口，使得绘制速度可以大幅提高，这对游戏等领域极为重要。



本次课程中将非常系统和全面的详解canvas的各种属性和项目中的应用。并且以实战的案例如刮刮卡、抽奖转盘、表单饼图、画板、粒子特效等案例帮助大家真正掌握canvas并能写出各种特效和动画。

## #canvas元素

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

canvas看起来和元素很相像，唯一的不同就是它并没有src和alt属性。实际上，canvas标签只有两个属性——width和height。这些都是可选的，并且同样利用DOM properties来设置。

当没有设置宽度和高度的时候，canvas会初始化宽度为300像素和高度为150像素。该元素可以使用CSS来定义大小，但在绘制时图像会伸缩以适应它的框架尺寸：如果CSS的尺寸与初始画布的比例不一致，它会出现扭曲。

如果你绘制出来的图像是扭曲的，尝试用width和height属性为canvas明确规定宽高，而不是使用CSS。

id属性并不是canvas元素所特有的，而是每一个 HTML 元素都默认具有的属性（比如 class 属性）。给每个标签都加上一个 id 属性是个好主意，因为这样你就能在我们的脚本中很容易的找到它。

canvas元素可以像任何一个普通的图像一样（有margin, border, background等等属性）被设计。然而，这些样式不会影响在 canvas 中的实际图像。我们将会在一个专门的章节里看到这是如何解决的。当开始时没有为 canvas 规定样式规则，其将会完全透明。

## #替换内容

canvas元素与img标签的不同之处在于，就像video, audio, 或者 picture元素一样，很容易定义一些替代内容。由于某些较老的浏览器（尤其是 IE9 之前的 IE 浏览器）或者文本浏览器不支持 HTML 元素"canvas"，在这些浏览器上你应该总是能展示替代内容。

这非常简单：我们只是在canvas标签中提供了替换内容。不支持canvas的浏览器将会忽略容器并在其中渲染后备内容。而支持canvas的浏览器将会忽略在容器中包含的内容，并且只是正常渲染 canvas。

举个例子，我们可以提供对 canvas 内容的文字描述或者是提供动态生成内容相对应的静态图片，如下所示：

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

<canvas id="clock" width="150" height="150">
  
</canvas>
```

## #结束标签不可省

与img元素不同，canvas 元素需要结束标签 (</canvas>)。如果结束标签不存在，则文档的其余部分会被认为是替代内容，将不会显示出来。

如果不需要替代内容，一个简单的 <canvas id="foo" ...></canvas> 在所有支持 canvas 的浏览器中都是完全兼容的。

## #渲染上下文 (The rendering context)

<canvas> 元素创造了一个固定大小的画布，它公开了一个或多个**渲染上下文**，其可以用来绘制和处理要展示的内容。我们将会将注意力放在 2D 渲染上下文中。其他种类的上下文也许提供了不同种类的渲染方式；比如，WebGL 使用了基于OpenGL ES 的 3D 上下文 ("webgl")。

canvas 起初是空白的。为了展示，首先脚本需要找到渲染上下文，然后在它的上面绘制。<canvas> 元素有一个叫做 getContext() 的方法，这个方法是用来获得渲染上下文和它的绘画功能。getContext()接受一个参数，即上下文的类型。对于 2D 图像而言，如本教程，你可以使用 CanvasRenderingContext2D。

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

代码的第一行通过使用 document.getElementById() 方法来为 <canvas> 元素得到 DOM 对象。一旦有了元素对象，你可以通过使用它的 getContext() 方法来访问绘画上下文。

## #检查支持性

替换内容是用于在不支持 `<canvas>` 标签的浏览器中展示的。通过简单的测试 `getContext()` 方法的存在，脚本可以检查编程支持性。上面的代码片段现在变成了这个样子：

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

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

## #一个模板骨架

这里的是一个最简单的模板，我们之后就可以把它作为之后的例子的起点。

```
<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>
```

上面的脚本中包含一个叫做 `draw()` 的函数，当页面加载结束的时候就会执行这个函数。通过使用在文档上加载事件来完成。只要页面加载结束，这个函数，或者像是这个的，同样可以使用 `window.setTimeout()`(en-US)，`window.setInterval()`(en-US)，或者其他任何事件处理程序来调用。

模板看起来会是这样。如这里所示，它最初是空白的。

## #一个简单例子

一开始，让我们来看个简单的例子，我们绘制了两个有趣的长方形，其中的一个有着 alpha 透明度。我们将在接下来的例子里深入探索一下这是如何工作的。

```
<html>
  <head>
    <script type="application/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);

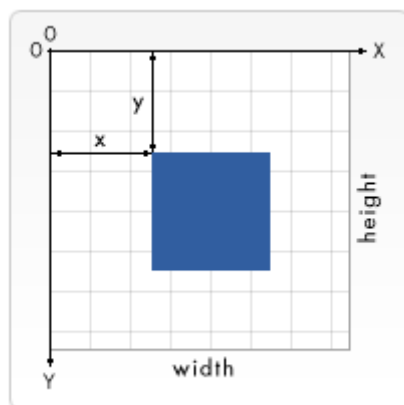
ctx.fillStyle = "rgba(0, 0, 200, 0.5)";
ctx.fillRect (30, 30, 55, 50);
}
}
</script>
</head>
<body onload="draw();">
  <canvas id="canvas" width="150" height="150"></canvas>
</body>
</html>
```



## 5.2 canvas绘制基本图形

既然我们已经设置了 canvas 环境，我们可以深入了解如何在 canvas 上绘制。到本文的最后，你将学会如何绘制矩形，三角形，直线，圆弧和曲线，变得熟悉这些基本的形状。绘制物体到 Canvas 前，需掌握路径，我们看看到底怎么做。

### #栅格



在我们开始画图之前，我们需要了解一下画布栅格（canvas grid）以及坐标空间。上一页中的 HTML 模板中有个宽 150px, 高 150px 的 canvas 元素。如右图所示，canvas 元素默认被网格所覆盖。通常来说网格中的一个单元相当于 canvas 元素中的一像素。栅格的起点为左上角（坐标为 (0,0) ）。所有元素的位置都相对于原点定位。所以图中蓝色方形左上角的坐标为距离左边（X 轴）x 像素，距离上边（Y 轴）y 像素（坐标为 (x,y) ）。在课程的最后我们会平移原点到不同的坐标上，旋转网格以及缩放。现在我们还是使用原来的设置。

### #绘制矩形

不同于 SVG，canvas 只支持两种形式的图形绘制：矩形和路径（由一系列点连成的线段）。所有其他类型的图形都是通过一条或者多条路径组合而成的。不过，我们拥有众多路径生成的方法让复杂图形的绘制成为了可能。

首先，我们回到矩形的绘制中。canvas 提供了三种方法绘制矩形：

`fillRect(x, y, width, height)`

绘制一个填充的矩形

`strokeRect(x, y, width, height)`

绘制一个矩形的边框

`clearRect(x, y, width, height)`

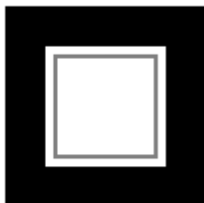
清除指定矩形区域，让清除部分完全透明。

上面提供的方法之中每一个都包含了相同的参数。x 与 y 指定了在 canvas 画布上所绘制的矩形的左上角（相对于原点）的坐标。width 和 height 设置矩形的尺寸。

下面的 draw() 函数是前一页中取得的，现在就来使用上面的三个函数。

## #矩形 (Rectangular) 例子

```
function draw() {  
  var canvas = document.getElementById('canvas');  
  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);  
  }  
}
```



`fillRect()`函数绘制了一个边长为 100px 的黑色正方形。`clearRect()`函数从正方形的中心开始擦除了一个 6060px 的正方形，接着`strokeRect()`在清除区域内生成一个 5050 的正方形边框。

接下来我们能够看到 `clearRect()` 的两个可选方法，然后我们会知道如何改变渲染图形的填充颜色及描边颜色。

不同于下一节所要介绍的路径函数（path function），以上的三个函数绘制之后会马上显现在 canvas 上，即时生效。

## #矩形

直接在画布上绘制矩形的三个额外方法，正如我们开始所见的绘制矩形，同样，也有 `rect()` 方法，将一个矩形路径增加到当前路径上。

`rect(x, y, width, height)`

绘制一个左上角坐标为 (x,y)，宽高为 width 以及 height 的矩形。



当该方法执行的时候，moveTo() 方法自动设置坐标参数 (0,0) 。也就是说，当前笔触自动重置回默认坐标。

## #绘制路径

图形的基本元素是路径。路径是通过不同颜色和宽度的线段或曲线相连形成的不同形状的点的集合。一个路径，甚至一个子路径，都是闭合的。使用路径绘制图形需要一些额外的步骤。

1. 首先，你需要创建路径起始点。
2. 然后你使用画图命令去画出路径。
3. 之后你把路径封闭。
4. 一旦路径生成，你就能通过描边或填充路径区域来渲染图形。

以下是所要用到的函数：

beginPath()

新建一条路径，生成之后，图形绘制命令被指向到路径上生成路径。

closePath()

闭合路径之后图形绘制命令又重新指向到上下文中。

stroke()

通过线条来绘制图形轮廓。

fill()

通过填充路径的内容区域生成实心的图形。

生成路径的第一步叫做 beginPath()。本质上，路径是由很多子路径构成，这些子路径都是在一个列表中，所有的子路径（线、弧形、等等）构成图形。而每次这个方法调用之后，列表清空重置，然后我们就可以重新绘制新的图形。

**备注：**当前路径为空，即调用 beginPath() 之后，或者 canvas 刚建的时候，第一条路径构造命令通常被视为是 moveTo ()，无论实际上是什么。出于这个原因，你几乎总是要在设置路径之后专门指定你的起始位置。

第二步就是调用函数指定绘制路径，本文稍后我们就能看到了。

第三，就是闭合路径 closePath(),不是必需的。这个方法会通过绘制一条从当前点到开始点的直线来闭合图形。如果图形是已经闭合了的，即当前点为开始点，该函数什么也不做。

**备注：**当你调用 fill() 函数时，所有没有闭合的形状都会自动闭合，所以你不需要调用 closePath() 函数。但是调用 stroke() 时不会自动闭合。

## #绘制一个三角形

例如，绘制三角形的代码如下：

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

    ctx.beginPath();
    ctx.moveTo(75, 50);
    ctx.lineTo(100, 75);
    ctx.lineTo(100, 25);
    ctx.fill();
  }
}
```

## #移动笔触

一个非常有用的函数，而这个函数实际上并不能画出任何东西，也是上面所描述的路径列表的一部分，这个函数就是moveTo()。或者你可以想象一下在纸上作业，一支钢笔或者铅笔的笔尖从一个点到另一个点的移动过程。

moveTo(x, y)

将笔触移动到指定的坐标 x 以及 y 上。

当 canvas 初始化或者beginPath()调用后，你通常会使用moveTo()函数设置起点。我们也能够使用moveTo()绘制一些不连续的路径。看一下下面的笑脸例子。我将用到moveTo()方法（红线处）的地方标记了。

你可以尝试一下，使用下边的代码片。只需要将其复制到之前的draw()函数即可。

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

    ctx.beginPath();
    ctx.arc(75, 75, 50, 0, Math.PI * 2, true); // 绘制
    ctx.moveTo(110, 75);
    ctx.arc(75, 75, 35, 0, Math.PI, false); // 口（顺时针）
    ctx.moveTo(65, 65);
    ctx.arc(60, 65, 5, 0, Math.PI * 2, true); // 左眼
    ctx.moveTo(95, 65);
    ctx.arc(90, 65, 5, 0, Math.PI * 2, true); // 右眼
    ctx.stroke();
  }
}
```





## #线

绘制直线，需要用到的方法lineTo()。

lineTo(x, y)

绘制一条从当前位置到指定 x 以及 y 位置的直线。

该方法有两个参数：x 以及 y，代表坐标系中直线结束的点。开始点和之前的绘制路径有关，之前路径的结束点就是接下来的开始点，等等。。。开始点也可以通过moveTo()函数改变。

下面的例子绘制两个三角形，一个是填充的，另一个是描边的。

```
function draw() {  
  var canvas = document.getElementById('canvas');  
  if (canvas.getContext){  
    var ctx = canvas.getContext('2d');  
  
    // 填充三角形  
    ctx.beginPath();  
    ctx.moveTo(25, 25);  
    ctx.lineTo(105, 25);  
    ctx.lineTo(25, 105);  
    ctx.fill();  
  
    // 描边三角形  
    ctx.beginPath();  
    ctx.moveTo(125, 125);  
    ctx.lineTo(125, 45);  
    ctx.lineTo(45, 125);  
    ctx.closePath();  
    ctx.stroke();  
  }  
}
```

这里从调用beginPath()函数准备绘制一个新的形状路径开始。然后使用moveTo()函数移动到目标位置上。然后下面，两条线段绘制后构成三角形的两条边。



## #圆弧

绘制圆弧或者圆，我们使用arc()方法。当然可以使用arcTo()，不过这个的实现并不是那么的可靠，所以我们这里不作介绍。

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

画一个以 (x,y) 为圆心的以 radius 为半径的圆弧（圆），从 startAngle 开始到 endAngle 结束，按照 anticlockwise 给定的方向（默认为顺时针）来生成。

arcTo(x1, y1, x2, y2, radius)

根据给定的控制点和半径画一段圆弧，再以直线连接两个控制点。

这里详细介绍一下 arc 方法，该方法有六个参数：x,y为绘制圆弧所在圆上的圆心坐标。radius为半径。startAngle以及endAngle参数用弧度定义了开始以及结束的弧度。这些都是以 x 轴为基准。参数 anticlockwise为一个布尔值。为 true 时，是逆时针方向，否则顺时针方向。

**备注：** arc() 函数中表示角的单位是弧度，不是角度。角度与弧度的 js 表达式：

**弧度=(Math.PI/180)\*角度。**

下面的例子比上面的要复杂一下，下面绘制了 12 个不同的角度以及填充的圆弧。

下面两个for循环，生成圆弧的行列 (x,y) 坐标。每一段圆弧的开始都调用beginPath()。代码中，每个圆弧的参数都是可变的，实际编程中，我们并不需要这样做。

x,y 坐标是可变的。半径 (radius) 和开始角度 (startAngle) 都是固定的。结束角度 (endAngle) 在第一列开始时是 180 度 (半圆) 然后每列增加 90 度。最后一列形成一个完整的圆。

clockwise语句作用于第一、三行是顺时针的圆弧，anticlockwise作用于二、四行为逆时针圆弧。if语句让一、二行描边圆弧，下面两行填充路径。

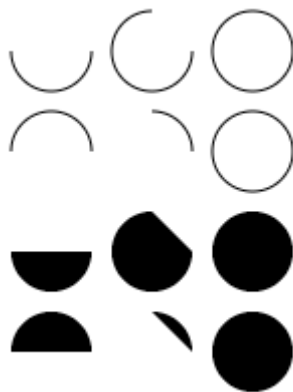
**备注：** 这个示例所需的画布大小略大于本页面的其他例子：150 x 200 像素。

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

        for(var i = 0; i < 4; i++){
            for(var j = 0; j < 3; j++){
                ctx.beginPath();
                var x = 25 + j * 50; // x 坐标值
                var y = 25 + i * 50; // y 坐标值
                var radius = 20; // 圆弧半径
                var startAngle = 0; // 开始点
                var endAngle = Math.PI + (Math.PI * j) / 2; // 结束点
                var anticlockwise = i % 2 == 0 ? false : true; // 顺时针或逆时针

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

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



## #使用arcTo方法

这是一段绘制圆弧的简单的代码片段。基础点是蓝色的，两个控制点是红色的。

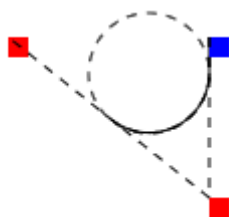
```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
```

```
ctx.setLineDash([])
ctx.beginPath();
ctx.moveTo(150, 20);
ctx.arcTo(150,100,50,20,30);
ctx.stroke();
```

```
ctx.fillStyle = 'blue';
// base point
ctx.fillRect(150, 20, 10, 10);
```

```
ctx.fillStyle = 'red';
// control point one
ctx.fillRect(150, 100, 10, 10);
// control point two
ctx.fillRect(50, 20, 10, 10);
//
```

```
ctx.setLineDash([5,5])
ctx.moveTo(150, 20);
ctx.lineTo(150,100);
ctx.lineTo(50, 20);
ctx.stroke();
ctx.beginPath();
ctx.arc(120,38,30,0,2*Math.PI);
ctx.stroke();
```

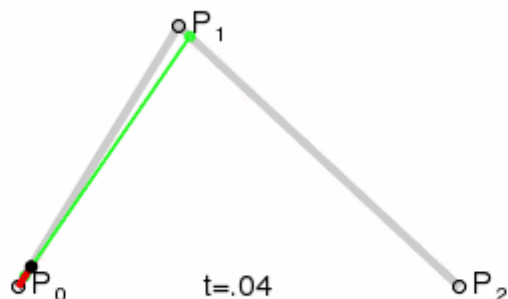


## #二次贝塞尔曲线及三次贝塞尔曲线

下一个十分有用的路径类型就是贝塞尔曲线。二次及三次贝塞尔曲线都十分有用，一般用来绘制复杂有规律的图形。

`quadraticCurveTo(cp1x, cp1y, x, y)`

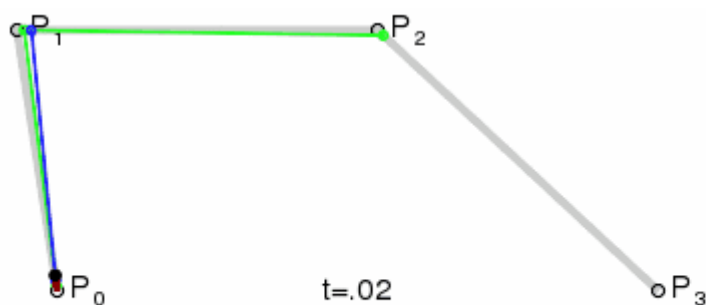
绘制二次贝塞尔曲线，cp1x,cp1y 为一个控制点，x,y 为结束点。



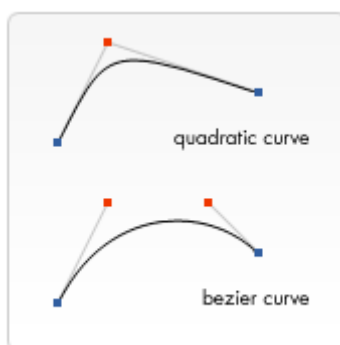
$$B(t) = (1 - t)^2 P_0 + 2t(1 - t)P_1 + t^2 P_2, t \in [0, 1]$$

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

绘制三次贝塞尔曲线，cp1x,cp1y为控制点一，cp2x,cp2y为控制点二，x,y为结束点。



$$B(t) = P_0(1 - t)^3 + 3P_1t(1 - t)^2 + 3P_2t^2(1 - t) + P_3t^3, t \in [0, 1]$$



右边的图能够很好的描述两者的关系，二次贝塞尔曲线有一个开始点（蓝色）、一个结束点（蓝色）以及一个控制点（红色），而三次贝塞尔曲线有两个控制点。

参数 x、y 在这两个方法中都是结束点坐标。cp1x,cp1y为坐标中的第一个控制点，cp2x,cp2y为坐标中的第二个控制点。

使用二次以及三次贝塞尔曲线是有一定的难度的，因为不同于像 Adobe Illustrators 这样的矢量软件，我们所绘制的曲线没有给我们提供直接的视觉反馈。这让绘制复杂的图形变得十分困难。在下面的例子中，我们会绘制一些简单有规律的图形，如果你有时间以及更多的耐心，很多复杂的图形你也可以绘制出来。

下面的这些例子没有多少困难。这两个例子中我们会连续绘制贝塞尔曲线，最后形成复杂的图形。

## #二次贝塞尔曲线

这个例子使用多个贝塞尔曲线来渲染对话气泡。

```
function draw() {  
  var canvas = document.getElementById('canvas');  
  if (canvas.getContext) {  
    var ctx = canvas.getContext('2d');  
  
    // 二次贝塞尔曲线  
    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();  
  }  
}
```



## #三次贝塞尔曲线

这个例子使用贝塞尔曲线绘制心形。

```
function draw() {  
  var canvas = document.getElementById('canvas');  
  if (canvas.getContext){  
    var ctx = canvas.getContext('2d');  
  
    //三次贝塞尔曲线  
    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();  
  }  
}
```



## #

## #组合使用

目前为止，每一个例子中的每个图形都只用到一种类型的路径。然而，绘制一个图形并没有限制使用数量以及类型。所以在最后的一个例子里，让我们组合使用所有的路径函数来重现一款著名的游戏。

```
function draw() {  
  var canvas = document.getElementById('canvas');  
  if (canvas.getContext){  
    var ctx = 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);  
    ctx.lineTo(31, 37);  
    ctx.fill();  
  
    for(var 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();
}
}

// 封装的一个用于绘制圆角矩形的函数。

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();
}

```

我们不会很详细地讲解上面的代码，因为事实上这很容易理解。重点是绘制上下文中使用到了 `fillStyle` 属性，以及封装函数（例子中的 `roundedRect()`）。使用封装函数对于减少代码量以及复杂度十分有用。

在稍后的课程里，我们会讨论 `fillStyle` 样式的更多细节。这章节中，我们对 `fillStyle` 样式所做的仅是改变填充颜色，由默认的黑色到白色，然后又又是黑色。

## #Path2D 对象

正如我们在前面例子中看到的，你可以使用一系列的路径和绘画命令来把对象“画”在画布上。为了简化代码和提高性能，Path2D对象已可以在较新版本的浏览器中使用，用来缓存或记录绘画命令，这样你能快速地回顾路径。

怎样产生一个 Path2D 对象呢？

Path2D()

Path2D()会返回一个新初始化的 Path2D 对象（可能将某一个路径作为变量——创建一个它的副本，或者将一个包含 SVG path 数据的字符串作为变量）。

所有的路径方法比如moveTo, rect, arc或quadraticCurveTo等，如我们前面见过的，都可以在 Path2D 中使用。

Path2D API 添加了 addPath作为将path结合起来的方法。当你想要从几个元素中来创建对象时，这将会很实用。比如：

Path2D.addPath(path [, transform])

添加了一条路径到当前路径（可能添加了一个变换矩阵）。

## #Path2D 示例

在这个例子中，我们创造了一个矩形和一个圆。它们都被存为 Path2D 对象，后面再派上用场。随着新的 Path2D API 产生，几种方法也相应地被更新来使用 Path2D 对象而不是当前路径。在这里，带路径参数的stroke和fill可以把对象画在画布上。

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

    var rectangle = new Path2D();
    rectangle.rect(10, 10, 50, 50);

    var circle = new Path2D();
    circle.moveTo(125, 35);
    circle.arc(100, 35, 25, 0, 2 * Math.PI);

    ctx.stroke(rectangle);
    ctx.fill(circle);
  }
}
```



## #使用 SVG paths

新的 Path2D API 有另一个强大的特点，就是使用 SVG path data 来初始化 canvas 上的路径。这将使你获取路径时可以以 SVG 或 canvas 的方式来重用它们。

这条路径将先移动到点 (M10 10) 然后再水平移动 80 个单位(h 80)，然后下移 80 个单位 (v 80)，接着左移 80 个单位 (h -80)，再回到起点处 (z)。

```
var p = new Path2D("M10 10 h 80 v 80 h -80 z");
```