本次分享准备较为仓促，望大家多多包涵，多多相互交流。

canvas

一. Canvas是啥

> - < canvas > 是一个可以使用脚本（通常是js）来绘图的HTML元素
> - < canvas > 最早由Apple引入WebKit，用于Mac OS X 的 Dashboard和 Safari
> - 如今，所有主流的浏览器都支持它（IE9+,更低版本需引入Explorer Canvas来支持）

1. 开始画图（渲染上下文）

< canvas > 元素创造了一个固定大小的画布，其上的渲染上下文，可以用来绘制和处理要展示的内容。
若要在canvas上绘图，首先得获取`CanvasRenderContext2D`2d渲染上下文。（此处指2d的，不谈webgl）

```
const canvas = document.getElementById('mycanvas');
const ctx = canvas.getContext('2d');
ctx.fillStyle = 'pink';
ctx.fillRect(10, 10, 300, 300);
```

示例

2. CanvasRenderContext2D的属性：

> 通过设置上下文的属性，可以指定绘图的样式。

所有属性如下：

属性	简介
canvas	canvas元素
fillStyle	用来**填充**路径的当前的*颜色、模式或渐变*
font	字体样式
globalAlpha	指定在画布上绘制的内容的不透明度
globalCompositeOperation	指定颜色如何与画布上已有的颜色组合（合成）
lineCap	指定线条的末端如何绘制
lineDashOffset	设置虚线偏移量
lineJoin	指定两条线条如何连接
lineWidth	指定画笔（绘制线条）操作的线条宽度
miterLimit	当 lineJoin 属性为 "miter" 的时候，这个属性指定了斜连接长度和线条宽度的最大比率
shadowBlur	模糊效果程度
shadowColor	阴影颜色
shadowOffsetX	阴影水平偏移距离
shadowOffsetY	阴影垂直偏移距离
strokeStyle	用于画笔（**绘制**）路径的颜色、模式和渐变
textAlign	文本的对齐方式
textBaseline	文字垂直方向的对齐方式

3. Canvas宽高

> - Canvas的宽高需要用属性值width，height来指定
> - 若未指定，则Canvas 的默认大小为300×150
> - 通过样式指定的宽高，只是canvas元素的显示大小，并不是绘图环境的大小

```
canvas {width: 1000px;height: 600px;}
<canvas id="mycanvas" width="1000" height="600"></canvas>
<canvas id="mycanvas1" width="500" height="300"></canvas>
<canvas id="mycanvas2"></canvas>
...
ctx.fillStyle = "red";
ctx.fillRect(10, 10, 100, 100);
```

宽高示例

为什么会样式设置了同样大小，显示却截然不同的情况呢？

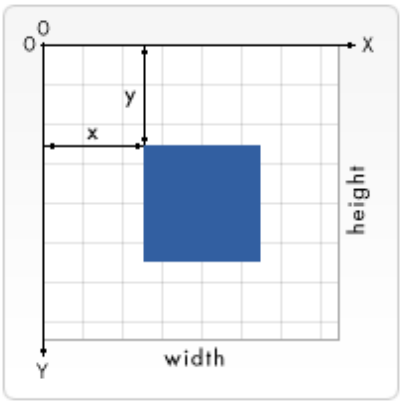
> - canvas元素本身有两套大小：一个是**元素本身**大小，一个是**绘图表面**（drawing surface）的大小
> - 如果通过width,height属性来设置，是同时修改了元素本身和绘图表面大小，
> - 如果canvas元素的大小不符合绘图表面大小时，则会对绘图表面进行缩放，使之符合元素本身大小，
> - 无特殊需求，建议直接使用canvas的width和height就好

----------我是神奇的分割线----------

二. 绘图

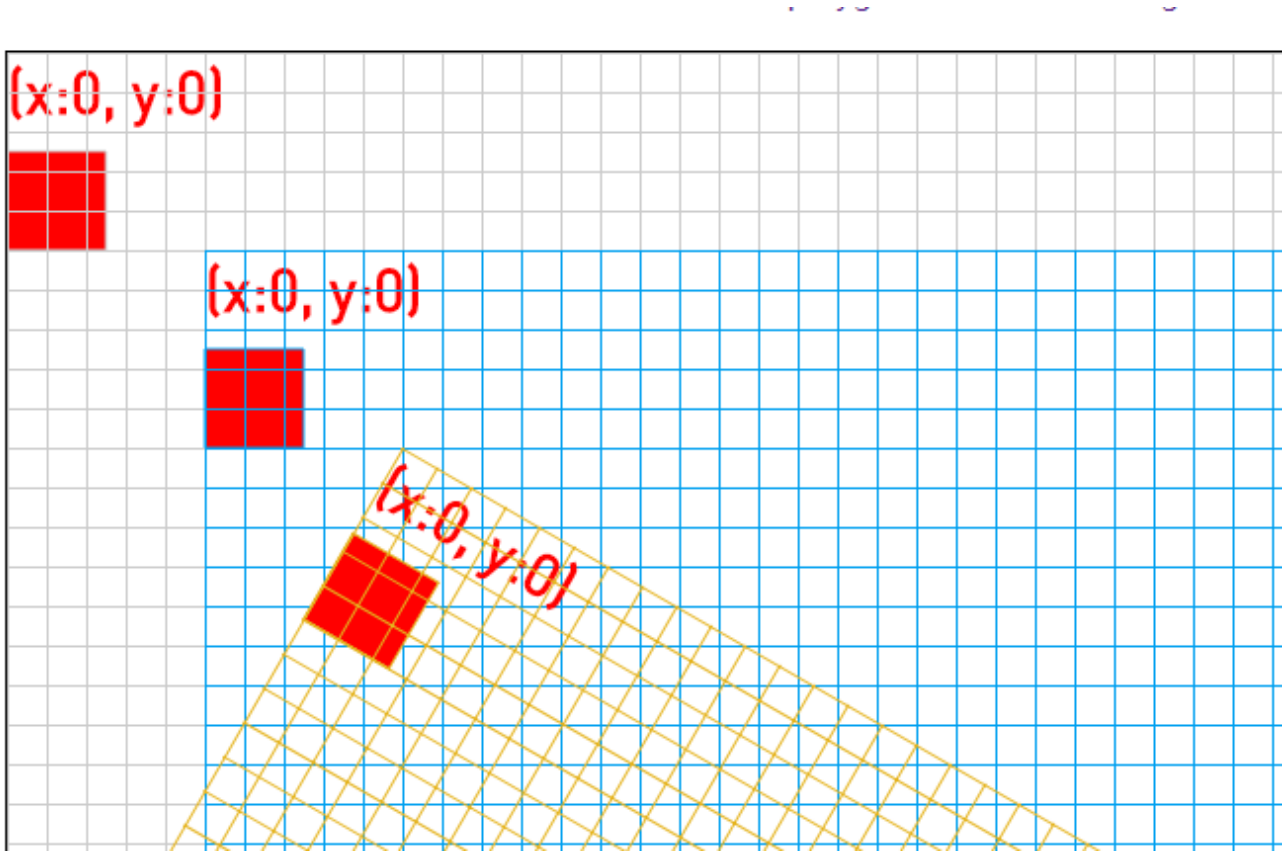
1. 坐标系

> - 画布左上角(0,0)默认原点
> - `x`坐标向右方增长，`y`坐标则向下方延伸



- 不过，Canvas的坐标系统并不是一尘不变的，原点是可改变的。
- 坐标变换：可以对Canvas坐标系统进行移动translate、旋转rotate和缩放等操作。
- 坐标变换之后绘制的图形x,y偏移量都以新原点为准， 旋转角度，缩放比，以新坐标系角度为准

例如同样在原点位置写上文字（x:0,y:0），在（0,50）位置绘制一个矩形，坐标变换前后的效果如下：



> - 这样， 当我们绘制多个图形时，既可以通过计算x，y偏移量控制图形的显示位置，也是通过变换坐标来做
> - 坐标变换是属于绘图状态存在Canvas绘图的堆栈中的，可以通过`save,restore`重置或更新坐标系

坐标示例

---

2. 矩形

> - canvas只支持一种原生的图形绘制：矩形。
> - 所有其他的图形的绘制都至少需要生成一条路径。

绘制矩形三种方法：

```
// 绘制一个填充的矩形
fillRect(x, y, width, height);
// 绘制一个矩形的边框
strokeRect(x, y, width, height);
// 清除指定矩形区域，让清除部分完全透明。
clearRect(x, y, width, height);
```

矩形示例

---

3. 路径

图形的基本元素是路径。路径是点的集合。
使用路径绘制图形一般步骤如下：

> - 1.`beginPath()`新建一条路径（有时需要创建路径起始点）
> - 2.使用`lineTo,arc,rect`等绘制路径
> - 3.`closePath`闭合路径（根据实际需求）

- 4.`stroke` `fill`描边或者填充（没有此步骤，图形不会显示）

路径绘制常见方法

```
// 直线路径
lineTo(x, y)

// 矩形路径
rect(x, y, width, height)

// 圆弧路径
arc(x, y, radius, startAngle, endAngle, anticlockwise)

// 椭圆路径(chrome37+)
ellipse(x, y, radiusX, radiusY, rotation, startAngle, endAngle, anticlockwise)

// 二次贝塞尔曲线
quadraticCurveTo(cp1x, cp1y, x, y)

// 三次贝塞尔曲线
bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)

// Path2D(chrome36+, addPath chrome68+)
new Path2D(path);
```

路径示例

3.1 线段（直线路径）

绘制线段一般步骤:

`moveTo(x,y)` 移动画笔到指定的坐标点(x,y)
`lineTo(x,y)` 使用直线连接当前端点和指定的坐标点(x,y)
`stroke()` 根据当前的画线样式，绘制当前或已经存在的路径

3.2 矩形路径

绘制矩形路径一般步骤:

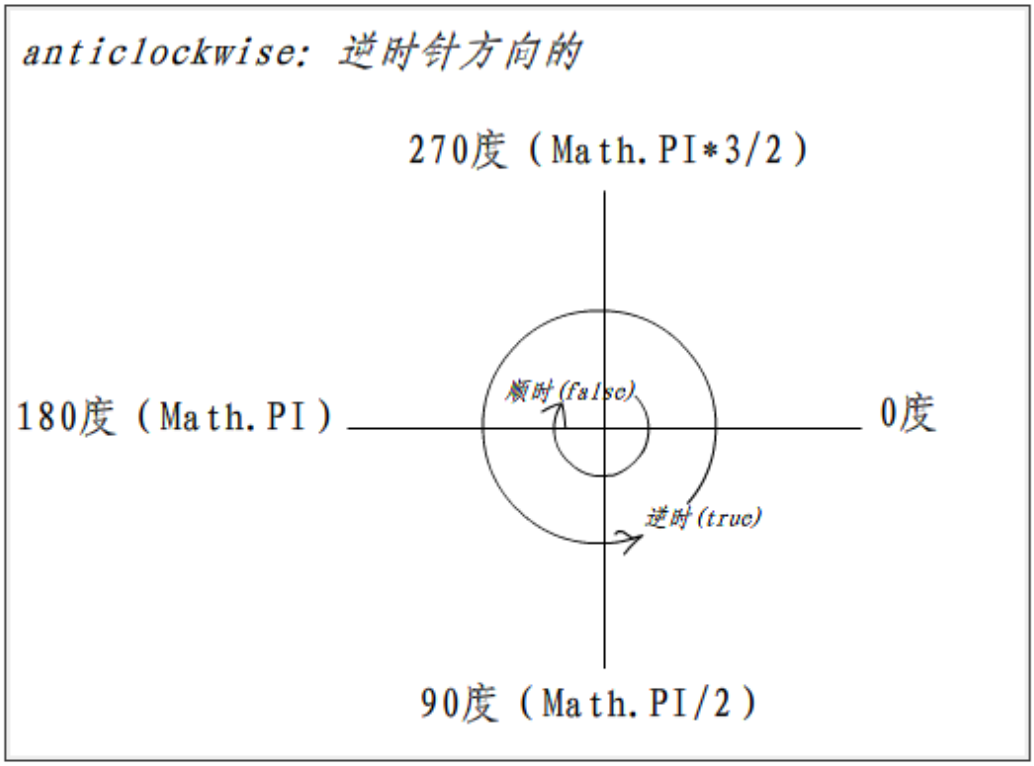
> `rect(x, y, width, height)` 矩形路径，坐标点(x,y)，width宽
> `stroke()`或`fill` 根据当前的样式，绘制或填充路径
> 也可使用前文提到的`strokeRect`或`fillRect`, 或者是通过`lineTo`拼接成矩形

3.3 圆弧路径

先看下绘制圆弧的api:

> - ctx.arc(x, y, radius, startAngle, endAngle, anticlockwise);
>
> `x, y` 圆弧中心， `radius` 圆弧半径, `startAngle` 起始点, `endAngle` 圆弧的终点, `anticlockwise` 默认为顺时针, true逆时针
> `CSS`中做旋转用到都是角度（ `deg` ），但是`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(deg)`。
> 这里弧度是以x轴正方向为基准、进行顺时针旋转的角度来计算

看下图示：



(图片来自大漠)

```
ctx.beginPath();
ctx.arc(200, 100, 100, 0, Math.PI / 2, false);
ctx.closePath();
ctx.stroke();
ctx.fill();
```

arc示例

3.4 贝塞尔曲线

canvas提供了两个绘制贝塞尔曲线api:

> - ctx.quadraticCurveTo(cpx, cpy, x, y);
>
> 二次贝塞尔曲线，(cpx, cpy)控制点 (x, y)终点
>
> - ctx.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y);
>
> 三次贝塞尔曲线，(cp1x, cp1y)控制点一, (cp2x, cp2y)控制点二， (x, y)终点

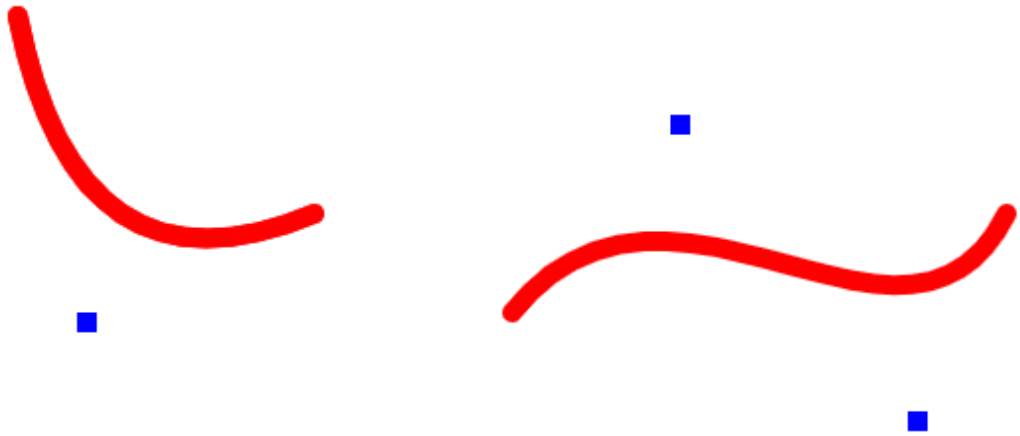
题外话：

> 贝塞尔曲线的数学基础是早在 1912 年就广为人知的伯恩斯坦多项式。最早用来辅助汽车车体的工业设计。
> CSS3的transition-timing-function属性，取值就可以设置为一个三次贝塞尔曲线方程`transition-timing-function: cubic-bezier(0.1, 0.7, 1.0, 0.1)`。

canvas绘图示例：

```
// 二次
ctx.moveTo(200, 100);
ctx.quadraticCurveTo(230, 250, 350, 200);
// 三次
ctx.moveTo(450, 250);
ctx.bezierCurveTo(530, 150, 650, 300, 700, 200);
```

蓝色是控制点



问题一：
那canvas是如何通过控制点来绘制出曲线的，或者如果不用这个，自己绘制曲线该如何操作呢：

这个是n阶贝塞尔曲线的方程：

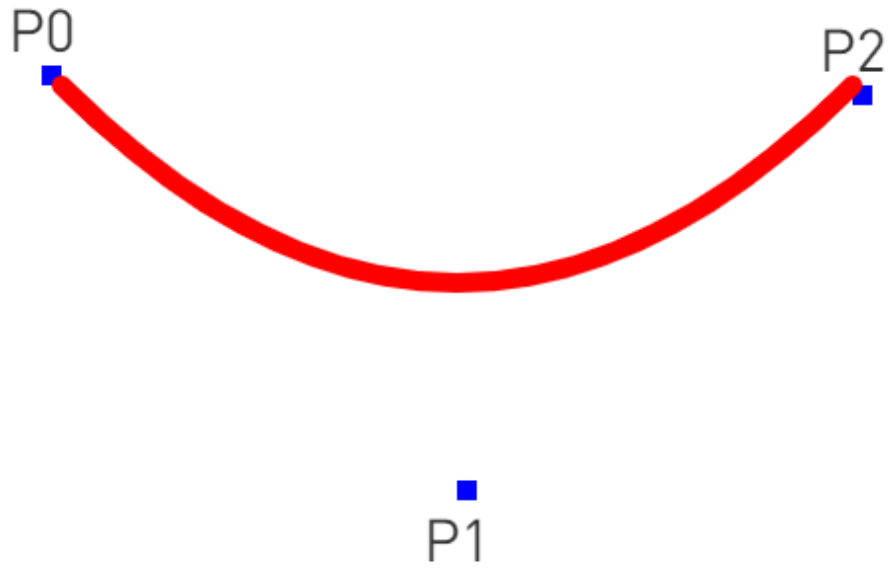
$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} \mathbf{P}_i (1-t)^{n-i} t^i = \binom{n}{0} \mathbf{P}_0 (1-t)^n t^0 + \binom{n}{1} \mathbf{P}_1 (1-t)^{n-1} t^1 + \cdots + \binom{n}{n-1} \mathbf{P}_{n-1} (1-t)^1 t^{n-1} + \binom{n}{n} \mathbf{P}_n (1-t)^0 t^n, \ t \in [0,1].$$

我们重点看二(三)阶方程：

$$\mathbf{B}(t) = (1-t)^2 \mathbf{P}_0 + 2t(1-t)\mathbf{P}_1 + t^2 \mathbf{P}_2 \ , \ t \in [0,1]$$

> B(t)是曲线上的点，t在0~1之间取值， P0起始点，P2终点，P1控制点
> t从0~1之间取值不断增大，B(t)不断取出曲线上的点，从P0移至P1

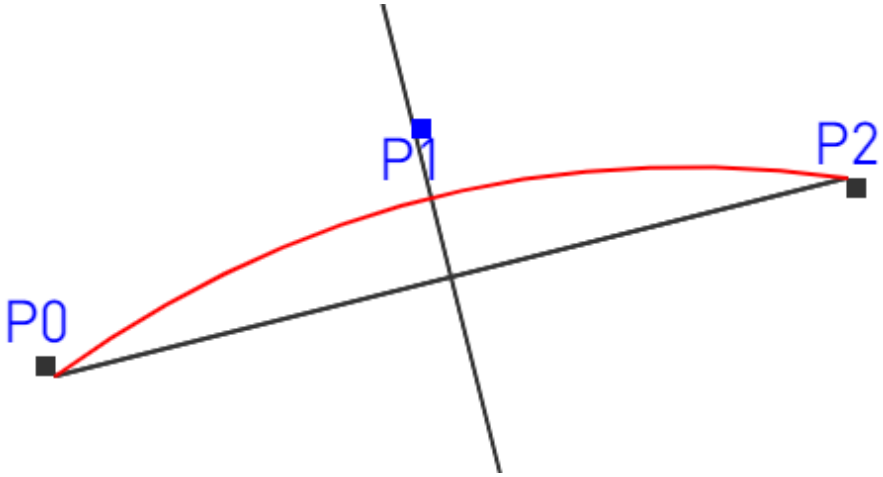
```
const bx = (1-t)*(1-t)*start.x + 2*t*(1-t)*control.x + t*t*end.x;
const by = (1-t)*(1-t)*start.y + 2*t*(1-t)*control.y + t*t*end.y;
```



问题二：
我咋知道控制点该怎么选，特别是起终点动态数据时(也就是说，我们使用时，往往只知道起点P0终点P1)：

> 这个根据曲线斜率，可视化需求可能选取的方式不一致，不过大致原理相似
> 可以在起点和终点的`垂直平分线`上选一点作为`控制点`，然后用一个参数来控制曲线的`弯曲程度`

```
// curveness 弯曲程度(0-1)
const cp = {
  x: ( start.x + end.x ) / 2 - ( start.y - end.y ) * curveness,
  y: ( start.y + end.y ) / 2 - ( end.x - start.x ) * curveness
};
```



题外话：

> 关于cp点的求解：
> 线段中点:
> const mid = [(start.x + end.x) / 2, (start.y + end.y) / 2];
> 根据起点和终点也可以得到一个向量v:
> const v = [end.x - start.x, end.y- start.y];
> 将这个向量顺时针旋转90度，得到一个垂直于它的向量v2:
> const v2 = [v.y, -v.x];
> 那么中间控制点的坐标为(向量v2乘curveness加上中间点坐标)
>
> const cp = {
> x: mid.x + v2.x * curveness,
> y: mid.y + v2.y * curveness
> } = {
> x:(start.x + end.x) / 2 - (start.y - end.y) * curveness,
> y:(start.y + end.y) / 2 - (end.x - start.x) * curveness
> {

3.5 Path2D

api:

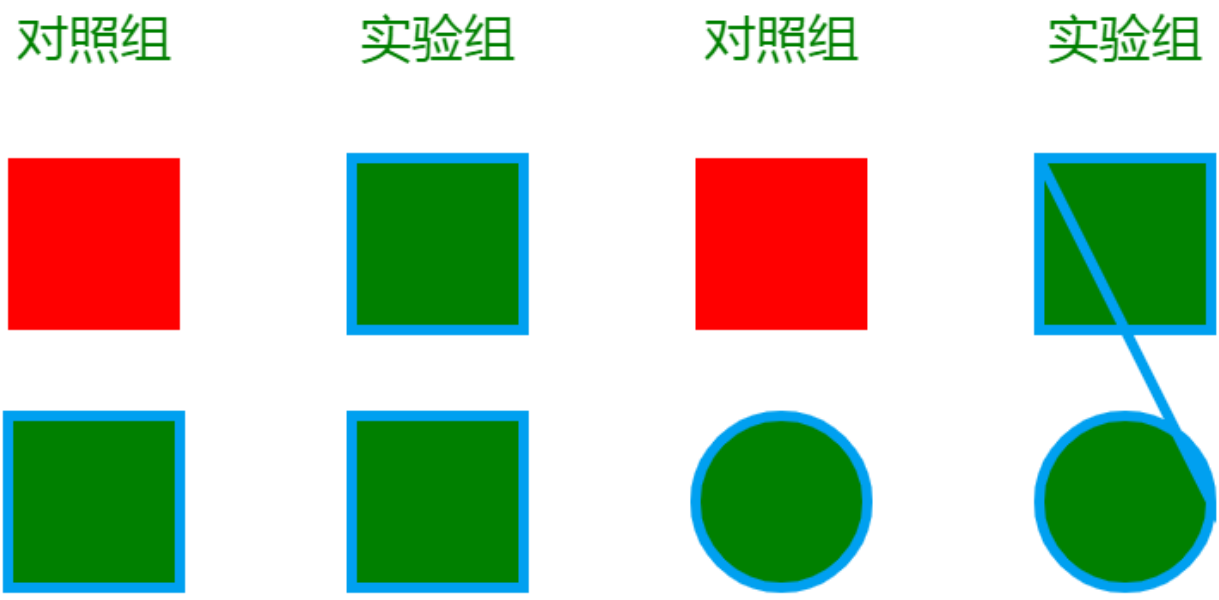
> - new Path2D();
> - new Path2D(path);
> - new Path2D(d);

```
// Path2D
const rectPath = new Path2D();
rectPath.rect(700, 100, 100, 100);
ctx.stroke(rectPath);

var arcPath = new Path2D();
arcPath.arc(750, 300, 50, 0, Math.PI * 2, true);
ctx.stroke(arcPath);

```

```
var arcPath1 = new Path2D();
arcPath1.moveTo(700, 380);
arcPath1.quadraticCurveTo(730, 420, 800, 380);
ctx.stroke(arcPath1);

// Path2D svg
const svgPath = new Path2D('M700 450 h 80 v 80 h 30 v 30 h -140 v -30 h 30 Z');
ctx.stroke(svgPath);
```

路径绘制有几个注意点：

- 1.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。
- 2.beginPath是通过清空子路径列表开始创建一个新路径。若在绘制新路径前未调用beginPath清空子路径，可能会发生一些意想不到的情况。
- 3.特别是arc，如果绘制新的圆弧路径前未清空子路径，则会在上一条路径的末尾处与新圆弧起点处绘制一条连接路径



beginPath意外示例

4. 填充，描边

要使路径显示出来，需使用`stroke`描边或者 `fill`填充, 在之前需指定对应的`fillStyle`或`strokeStyle`
fillStyle, strokeStyle 可以是`颜色、渐变或图案`

```
// Color接受符合CSS3规范的颜色值
ctx.fillStyle = "pink";
ctx.fillStyle = "#333";
ctx.fillStyle = "rgba(255, 165, 0, 0.1)";

// 线性渐变 createLinearGradient(x1, y1, x2, y2)
const linearGradient = ctx.createLinearGradient(200, 0, 200, 200);
linearGradient.addColorStop(0, '#f05a8c');
linearGradient.addColorStop(0.5, '#e6b41e');
linearGradient.addColorStop(1, '#00d796');
ctx.fillStyle = linearGradient;

// 径向渐变 createRadialGradient(x1, y1, r1, x2, y2, r2)
const radialGradient = ctx.createRadialGradient(30, 30, 20, 52, 50, 50);
radialGradient.addColorStop(0, '#e6b41e');
radialGradient.addColorStop(0.7, '#00B5E2');
radialGradient.addColorStop(1, 'rgba(0,201,255,0)');
ctx.fillStyle = radialGradient;

// 图案填充描边
const img = new Image();
img.src = 'https://stdl.qq.com/stdl/skin/10/assets/img/fei.png'
img.onload = function () {
    const ptrn = ctx.createPattern(img, 'repeat');
    ctx.fillStyle = ptrn;
    ctx.strokeStyle = ptrn;
}
```

颜色示例

---

5. 文本

绘制文本有两个方法

```
// 在指定的(x,y)位置填充指定的文本
fillText(text, x, y [, maxWidth])

// 在指定的(x,y)位置绘制指定的文本
strokeText(text, x, y [, maxWidth])

// demo
ctx.textAlign = 'center';
ctx.textBaseline = "middle";
ctx.font = 'bold 60px Microsoft YaHei';
ctx.shadowOffsetX = 5;
ctx.shadowOffsetY = 5;
ctx.shadowBlur = 10;
ctx.fillStyle = 'red';
// ctx.fillStyle = linearGradient;
// ctx.fillStyle = ctx.createPattern(img, 'repeat');
ctx.fillText('Alpha QQ浏览器', 100, 100);
```

- 绘制可以指定文本的样式
- 1.font (使用和 CSS font 属性相同的语法. 默认的字体是 `10px sans-serif`)
- 2.textAlign (文本对齐选项,可选的值包括：start, end, left, right or center. 默认值是 start)
- 3.textBaseline (基线对齐选项,top, hanging, middle, alphabetic, ideographic, bottom。默认值是 alphabetic)
- 4.direction (文本方向,ltr, rtl, inherit。默认值是 inherit)

- 文本的填充描边样式可以是颜色、渐变或图案,文本可以设置阴影

文本示例

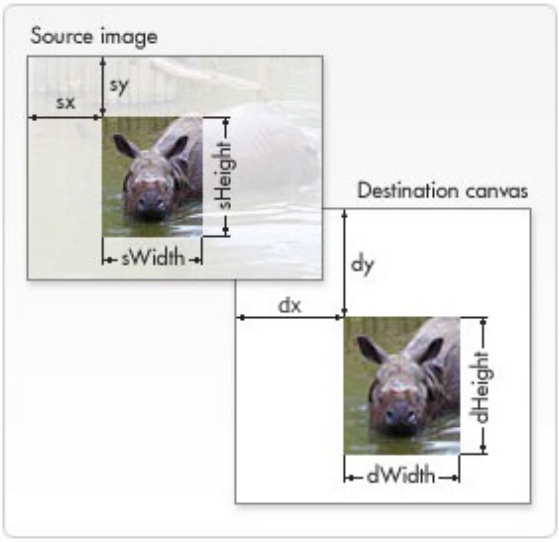
---

6. 图像、视频

- 浏览器支持的任意格式的外部图片都可以使用
- 视频或其他canvas元素生成的图片也可以作为图片源
- 使用drawImage引入图像到canvas：
- 1.获取图片源

（HTMLImageElement，HTMLVideoElement，HTMLCanvasElement，ImageBitmap）

- 2.使用drawImage()函数将图片绘制到画布上



(图来自mdn)

```
// drawImage有三种传参方式
ctx.drawImage(image, dx, dy);
ctx.drawImage(image, dx, dy, dWidth, dHeight);
ctx.drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight);

// image可以是图片，视频或canvas
// image是HTMLImageElement
const img = new Image();
img.onload = function() {
    ctx.drawImage(img, 0, 0, 300, 450, 10, 100, 300, 450);
};
img.src = '../images/feifei.png';

// image是HTMLCanvasElement
const canvas1 = document.getElementById('mycanvas1');
...
ctx.drawImage(canvas1, 0, 0, 300, 450, 10, 100, 300, 450);

// image是HTMLVideoElement
const video = document.getElementById('video');
ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
```

图片处理还可以通过createPattern绘制图案的方式，或者是像素处理putImageData来做。

7. 像素处理

```
// 返回一个ImageData对象，用来描述canvas区域隐含的像素数据
ctx.getImageData(sx, sy, sw, sh)

// 将数据从已有的 ImageData 对象绘制到位图
ctx.putImageData(imagedata, dx, dy)
ctx.putImageData(imagedata, dx, dy, dirtyX, dirtyY, dirtyWidth, dirtyHeigh
t);
```

> - imagedata.data是Uint8ClampedArray类型（8位无符号整型固定数组，值固定在0-255区间的8位无符号整型组成的数组），数组长度是4*width*height，因为数组依次表示每个点的RGBA的值
> - data[0] -> point[0].r, data[1]->point[0].g, data[2] -> point[0].b, data[3] -> point[0].a ……
> - imagedata.width 图像宽度
> - imagedata.height 图像高度
> -
> - 通过像素处理可以实现取色，简单的滤镜效果等

像素示例

3. 特性

1. 事件处理

同别的html元素一样， < cavnas > 上也有一些常见的事件处理：
1. 鼠标事件（mousemove, click...）
2. 触摸事件（touchstart...）
3. 拖拽事件（drag, drop...）

区别之处在于：由于canvas其上图形是并没有dom和对象的，所以canvas上绘制的图形是没有这

些事件的

```
canvas.addEventListener('click', e => {
...
});

canvas.addEventListener('drop', e => {
...
});

// 由于canvas是无法获取焦点的，所以没有键盘事件
canvas.addEventListener('keypress', e => {
...
});
```

事件示例

---

2. save & restore（状态保存和恢复）

官方文档：

> `CanvasRenderingContext2D.save()` 是 Canvas 2D API 通过将 `当前状态` 放入栈中，保存 canvas 全部状态的方法。
> `CanvasRenderingContext2D.restore()` 是 Canvas 2D API 通过在绘图状态栈中弹出顶端的状态，将 canvas 恢复到最近的保存状态的方法。如果没有保存状态，此方法不做任何改变。

- 简单说，`save` 和 `restore` 用于保存及恢复当前Canvas绘图环境的所有属性。其中save()可以保存当前状态，而restore()可以还原之前保存的状态。
- save()方法会将Canvas的状态推到Canvas绘图的堆栈中，然后通过restore()方法从Canvas绘图的堆栈中取出save()保存的状态。
- 这里可能有个误区，会认为每一步 `save()` 之后 `resotre()` 就等同于 `ctrl+z`，其实并不是的，因为save保存的是绘图的状态，并不包括canvas上绘制的图形。

哪些状态可以保存在栈中？

- CanvasRenderContext2D的除canvas外的所有属性（如fillStyle,strokerStyle等）属性
- 当前的变换矩阵
- 当前的剪切区域
- 当前的虚线列表

可以看下以下各部分绘制的矩形是什么颜色的：

```
1.
2.  ctx.save();
3.  ctx.translate(100, 100);
4.
5.
6.  ctx.fillStyle = 'red';
7.  ctx.fillRect(10, 0, 200, 200);
8.  ctx.save();
9.
10. ctx.fillStyle = 'green';
11. ctx.fillRect(20, 50, 200, 200);
12. ctx.save();
13.
14. ctx.fillStyle = 'yellow';
15. ctx.fillRect(30, 100, 200, 200);
16. ctx.save();
17.
18. ctx.fillStyle = 'blue';
19. ctx.fillRect(40, 150, 200, 200);
20. ctx.save();
21.
22. ctx.restore();
23. ctx.fillRect(310, 0, 200, 200); // ?
24.
25. ctx.restore();
26. ctx.fillRect(320, 50, 200, 200); // ?
27.
28. ctx.restore();
29. ctx.fillRect(330, 100, 200, 200); // ?
30.
31. ctx.restore();
32. ctx.fillRect(340, 150, 200, 200); // ?
33.
34. ctx.restore();
35. ctx.fillRect(350, 200, 200, 200); // ?
```

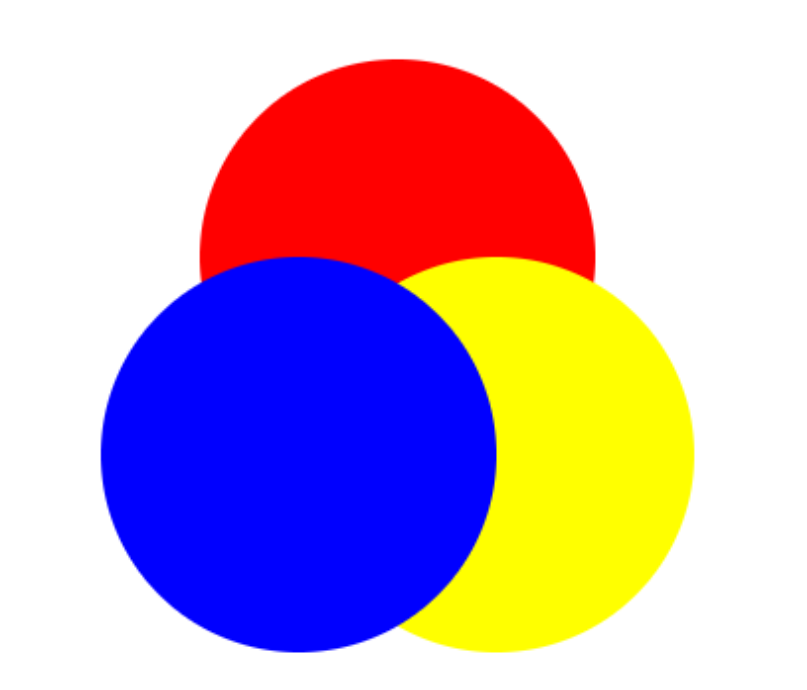
save_restore示例

3. globalCompositeOperation(图像合成)

官方文档：

> `globalCompositeOperation` 属性设置要在绘制新形状时应用的合成操作的类型
> 语法：ctx.globalCompositeOperation = type; 默认设置是source-over

说人话：

> - 这个属性用户控制画布上对象的透明度和分层效果，当一个图形绘制在另一个图形上时，浏览器如何处理两个图形之间相互遮盖产生的效果。见下图：
>
> 

`type`有12个值，标志了12中遮盖方式
- `source-over` 默认值，在现有画布上下文之上绘制新图形。
- `source-in` 新图形只在新图形和目标画布重叠的地方绘制。其他的都是透明的。
- overlay...
- screen.. 更多参见MDN

canvas的这个属性，跟css3的一个属性很类似：`混合模式 <blend-mode>` 参见MDN

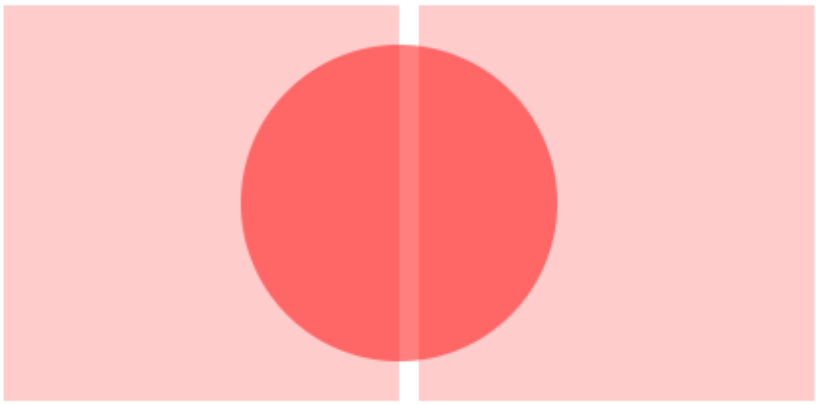
4. clip剪裁

官方文档：

> `CanvasRenderingContext2D.clip()`是 Canvas 2D API 将当前创建的路径设置为当前剪切路径的方法。
> 语法
> void ctx.clip();
> void ctx.clip(fillRule);
> void ctx.clip(path, fillRule);

再讲人话：

> - 剪切区域是Canvas之中由路径所定义的一块区域，浏览器会将所有的绘图操作都限制在本区域内执行。
> - 用clip()设置剪切区域，一旦设置好剪辑区域，那么你在Canvas之中绘制的所有内容都将局限在该区域内。这也意味着在剪辑区域以外进行绘制是没有任何效果的。
> - 如下图，设置红色区域为剪切区域，那么绘制的所有内容都在红色区域内，实际设置后的效果如右图



示例

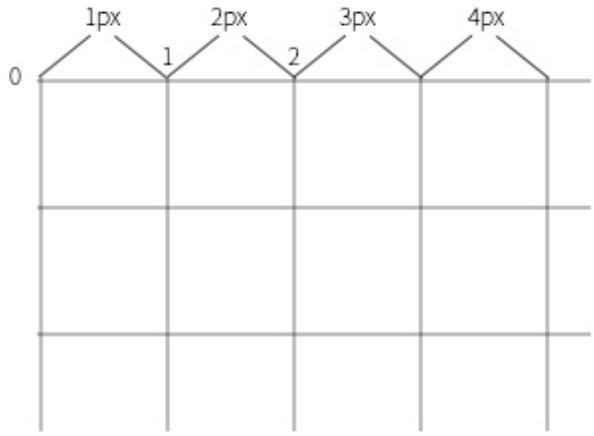
```
ctx.save();
ctx.arc(250, 150, 50, 0, Math.PI*2, false);
ctx.clip();
ctx.fillStyle = 'green';
ctx.fillRect(250, 100, 100, 100);
ctx.restore();
```

clip示例

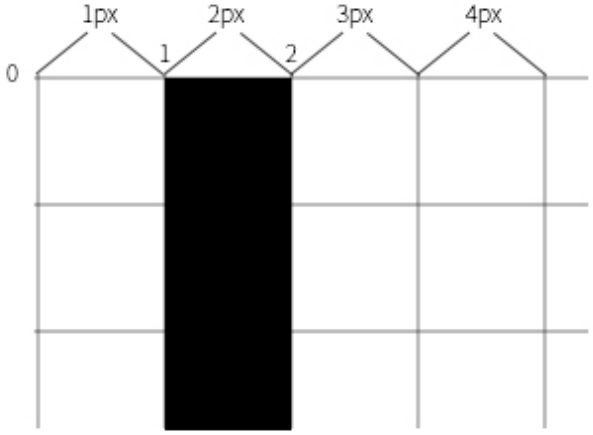
> - 这个跟之前设置globalCompositeOperation为`source-in`效果类似
> - 在save，restore里说过，剪切区域是作为绘图状态保存在栈里的，所以可以通过`save,restore`来取消剪切区域

5.1px线条模糊问题

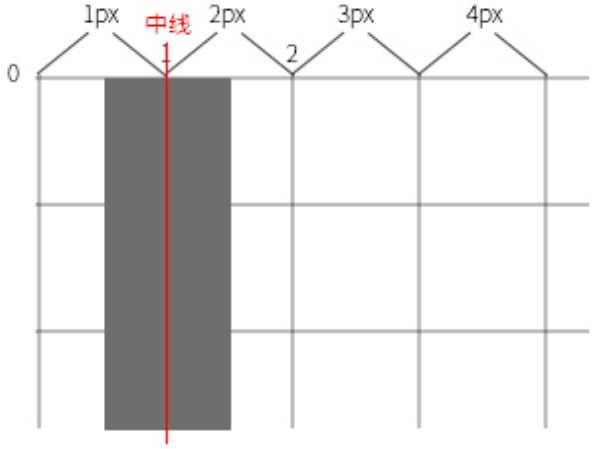
canvas画布像素模拟：



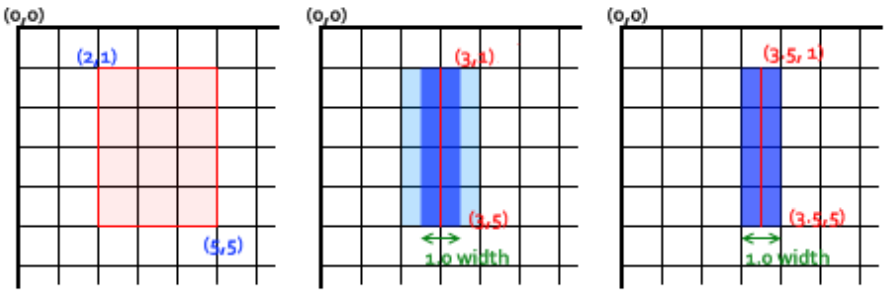
绘制一个宽度为1px的线条, 我们预想的结果，是在(1, y)和(2, y)绘制线条：



实际结果，是在(0.5,y)和(1.5,y)之间绘制线条：



也就是说，是以x=1为中线向两端延伸。



> - canvas绘制线条时，宽度会沿着线条宽度的中线向两端延伸，路径两旁各一半像素
> - 计算机并不会只绘制0-1和1-2的半个像素，而将绘制0-1和1-2的整个像素，那么加起来就是2px。
> - 那么为何会模糊呢，这是因为0-0.5和1.5-2之间的内容会被更浅颜色的内容填充，所以看到模糊效果

MDN

三. 动画

四. 其他

canvas vs svg

> - SVG: 可缩放矢量图形（Scalable Vector Graphics）
> - SVG 基于 XML，意味着 SVG DOM 中的每个元素都是可用的。可以为某个元素附加 JavaScript 事件处理器。
> - 在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变化，那么浏览器能够自动重现图形。

项目	canvas	SVG
图形格式	位图	矢量图形

项目	canvas	SVG
dom结构	一个HTML元素	绘制的各个图形都是单独的一个元素，是DOM的一部分
开发方式	js绘制	XML 描述
事件	图像不支持事件处理（canvas自身支持）	图像支持事件处理
模式	立即模式，不记住历史绘制对象列表	保留模式
场景		234