canvas课程

第一季课堂笔记

# Canvas概述

推荐文档地址：火狐的mdn文档

https://developer.mozilla.org/zh-CN/docs/Web/API/Canvas\_API/Tutorial/Drawing\_shapes

长久以来，web上的动画都是Flash。比如动画广告、游戏等等，基本上都是Flash实现的。Flash是有缺点的，比如需要安装Adobe Flash Player，漏洞多，重量比较大。卡顿和不流畅等等。

HTML5提出了一个新的canvas标签，彻底颠覆了Flas 的主导地位。无论是广告、游戏都可以使用canvas实现了，Canvas是一个轻量级的画布，我们使用Canvas进行JavaScript的编程，不需要增加额外的插件，性能也很好，不卡顿，在手机中也很流畅

## Hello World

我们设置在页面设置一个canvas标签

<canvas width="500" height="400">

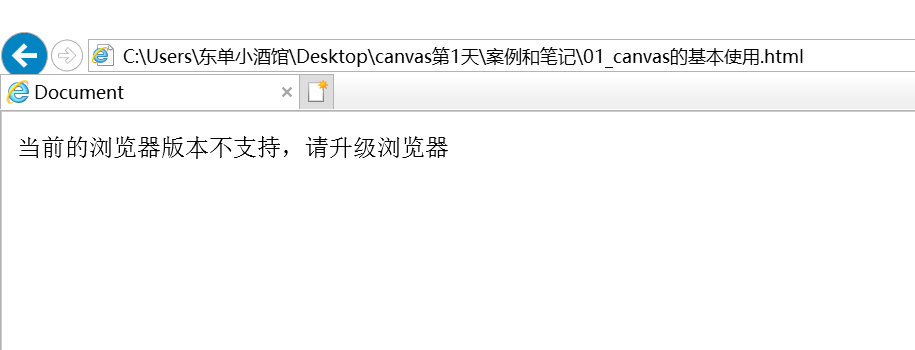
当前的浏览器版本不支持，请升级浏览器

</canvas>

canvas的标签属性只有两个，width和height。表示的是canvas画布的宽度和高度。

注意canvas的width和height不要用css的样式来设置，如果使用css的样式来设置，画布会失真，会变形。

标签对儿里面的文字是用来提示低版本浏览器（IE6/7/8）



上图是我们使用ie7版本浏览器打开的效果

高版本浏览器是看不到canvas标签内部的文字的

// 得到canvas的画布

var canvas = document.getElementById("mycanvas");

// 得到画布的上下文，上下文有两个，2d的上下文和3d的上下文

// 所有的图像绘制都是通过ctx属性或者 是方法进行设置的，和canvas标签没有关系了

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

// 设置颜色

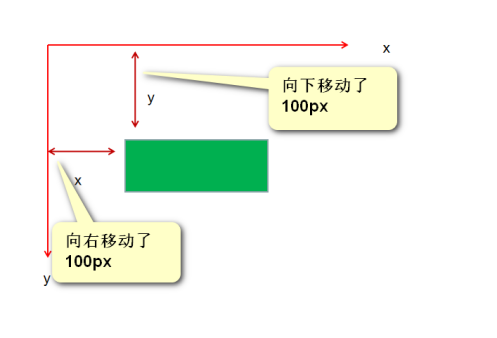
ctx.fillStyle = 'green';

// 绘制矩形

ctx.fillRect(100, 100, 200, 50);



通过上面的代码我们发现canvas本质上就是利用代码在浏览器的页面上进行画画，比如我们上面的代码fillRect就代表在页面中绘制矩形，一共四个属性前两个100，100，表示的是x和y的坐标，后两个是绘制的宽度和高度



## Canvas的像素化

我们使用canvas绘制了一个图形，一旦绘制成功了，canvas就像素化了他们。canvas没有能力，从画布上再次得到这个图形，也就是说我们没有能力去修改已经在画布上的内容。这个就是canvas比较轻量的原因，Flash重的原因之一就有它可以通过对应的api得到已经上“画布”的内容然后再次绘制

如果我们想要让这个canvas图形移动，必须按照清屏-更新-渲染的逻辑进行编程，总之就是重新再画一次

## canvas的动画思想

我们之前在面向对象的课程中讲过动画的思想就是**清屏-更新-渲染**，这个思想就可以为canvas的动画思想

canvas上画布的元素，就被像素化了，所以不能通过style.left方法进行修改，而是必须要重新绘制

// 得到画布

var canvas = document.getElementById("maycanvas");

// 获取上下文

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

// 设置颜色

ctx.fillStyle = "blue";

// 信号量

var left = 100;

// 动画过程

setInterval(function() {

// 清除画布，0,0,代表从什么位置开始清除，600,600代表清除的宽度和高度

ctx.clearRect(0, 0, 600, 600);

// 更新信号量

left++;

//重新绘制

ctx.fillRect(left, 100, 100, 100)

}, 30)

实际上，动画的生存就是相关静态画面连续播放了，这个就是动画的过程。我们把每一次绘制的静态画面叫做“**一帧**”，时间的间隔（定时器的间隔）就表示的是帧的间隔

## 面向对象思维实现canvas动画

因为canvas不能得到已经上屏的对象，所以我们要维持对象的状态。在canvas动画中，我们都使用面向对象来进行编程，因为我们可以使用面向对象的方式来维持canvas需要的属性和状态；

function Rect(x, y, w, h, color) {

// 维护状态

this.x = x;

this.y = y;

this.w = w;

this.h = h;

this.color = color;

};

// 更新的方法

Rect.prototype.update = function() {

this.x++;

};

// 渲染

Rect.prototype.render = function() {

// 设置颜色

ctx.fillStyle = this.color;

// 渲染

ctx.fillRect(this.x, this.y, this.w, this.h);

};

我们创建了两个实例

// 实例化

var r1 = new Rect(100, 100, 50, 50, "purple");

var r2 = new Rect(50, 200, 80, 80, "blue");

动画过程在主定时器里面，每一帧都会调动实例的更新和渲染方法

setInterval(function() {

// 清屏

ctx.clearRect(0, 0, canvas.width, canvas.height);

// 更新方法

r1.update();

// 渲染方法

r1.render()

// 更新方法

r2.update();

// 渲染方法

r2.render()

}, 10)

# Canvas的绘制功能

## 2.1 绘制矩形

**填充**一个矩形

// 设置颜色

ctx.fillStyle = 'green';

// 填充矩形

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

参数含义：分别代表**填充**坐标x、**填充**坐标y、**填充**矩形的宽度、**填充**矩形的高度



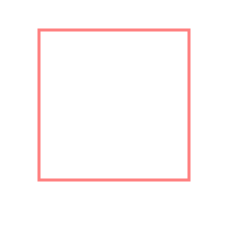
绘制矩形边框，和填充不同的是**绘制**使用的是strokeRect，和strokeStyle实现的

ctx.strokeStyle = "red";

// 绘制矩形

ctx.strokeRect(300, 100, 100, 100);

参数含义：分别代表**绘制**坐标x、**绘制**坐标y、**绘制**矩形的宽度、**绘制**矩形的高度



**清除**就是使用clearRect

ctx.clearRect(0, 0, 600, 600);

参数含义：分别代表**清除**坐标x、**清除**坐标y、**清除**矩形的宽度、**清除**矩形的高度

## 2.2 绘制路径

绘制路径作用是为了设置一个不规则的多边形状态

路径都是闭合的，使用路径进行绘制的时候需要既定的步骤

1. 需要设置路径的起点
2. 使用绘制命令画出路径
3. 封闭路径
4. 填充或者绘制已经封闭路径的形状

// 创建一个路径

ctx.beginPath();

// 移动绘制点

ctx.moveTo(100, 100);

// 描述行进路径

ctx.lineTo(200, 200);

ctx.lineTo(400, 180);

ctx.lineTo(380, 50);

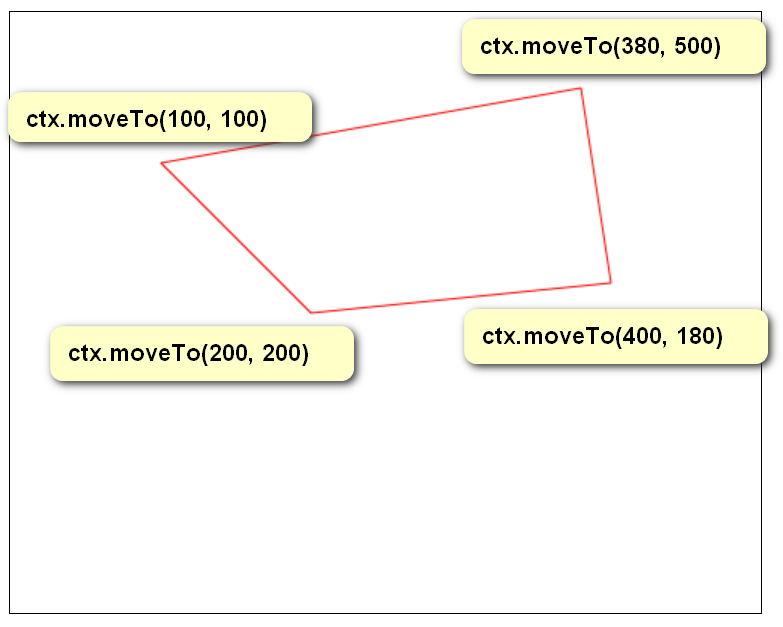
// 封闭路径

ctx.closePath();

// 绘制这个不规则的图形

ctx.strokeStyle = "red";

ctx.stroke();



总结我们要绘制一个图形，要安装顺序

开始路径

ctx.beginPath();

移动绘制点

ctx.moveTo(x,y);

描述绘制路径

ctx.lineTo(x,y);

多次描述绘制路径

ctx.lineTo(x,y);

闭合路径

ctx.closePath();

描边

ctx.stroke();

填充

ctx.fill();

此时我们发现之前我们在学习绘制矩形的时候使用的是fillRect和strokeRect，但是实际实施fill和stroke也是具有绘制填充功能的

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。

我们在绘制路径的时候可以选择不关闭路径（不设置closePath()），这个时候会实现自封闭现象（只针对fill功能）

|  |  |
| --- | --- |
| stroke的绘制 | fill的填充 |

## 2.3 圆弧

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

圆弧也是绘制路径的一种也需要beginPath和stroke

ctx.beginPath();

ctx.arc(200, 200, 100, 0, 1, false);

ctx.stroke();

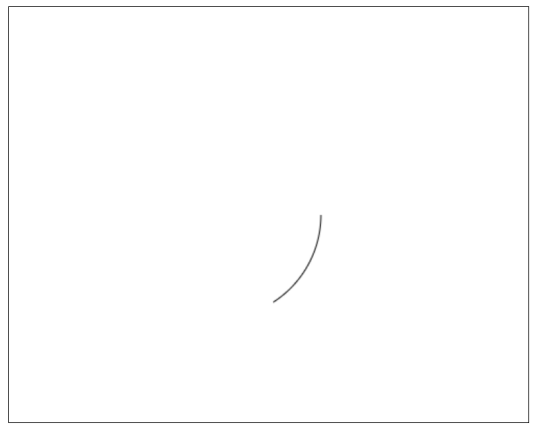
参数的含义：

200,200是起始坐标

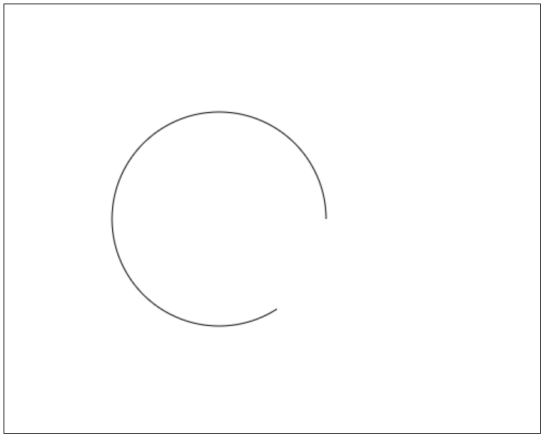
100代表的是圆心半径

0,1指的是开始和结束的位置，单位如果是数字，则代表一个圆弧的弧度（一个圆的弧度是Math.PI \* 2，约等于7个弧度），所以如果为两个参数的差为7，并且是顺时针方向，则表示绘制一个圆

最后一个参数false指的是绘制方向为顺时针

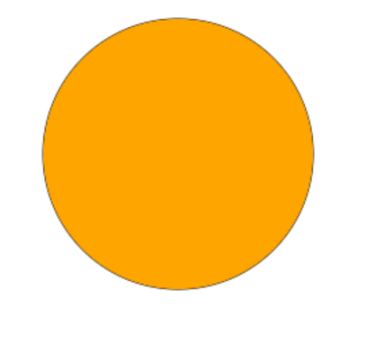


如果为 true的时候就代表，从0到1的位置逆时针方向绘制



如何设置一个圆？

ctx.arc(200, 200, 100, 0, 2 \* Math.PI, false);



## 2.4 透明度

透明度的值是0~1之间

ctx.globalAlpha = 0.2;

## 2.5 线型

我们可以利用lineWidth设置线的粗细，属性值必须是数字，默认是1.0。没有单位

ctx.beginPath();

ctx.moveTo(100, 100);

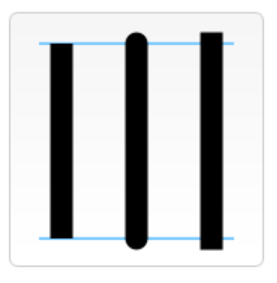
ctx.lineTo(200, 200);

ctx.lineTo(300, 100);

**ctx.lineWidth = 20;**



lineCap属性决定了每一条线段末端的属性，一共有3个属性值：分别是：butt, round and square。默认值是 butt



比如下面的代码

ctx.lineCap = "round";



lineJoin属性是决定了图形中两段连接处所显示的样子、一共有三种结果：round、bevel和miter，默认是miter

比如

ctx.lineJoin = "round"



我们也可以使用setLineDash方法来定义虚线的样式，内部接受一个数组（数组内部是虚线的交替状态）

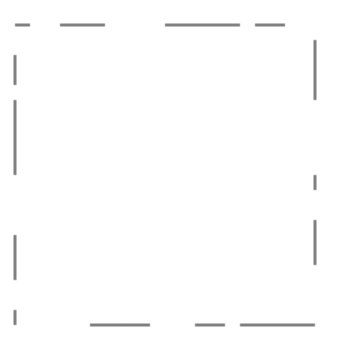
ctx.setLineDash([10, 20]);



数组的内部可以是为一组的状态，最少接受两个参数，也可以传多个

比如

ctx.setLineDash([10, 20, 30, 40, 50]);



还可以使用lineDashOffset来设置虚线的起始偏移量

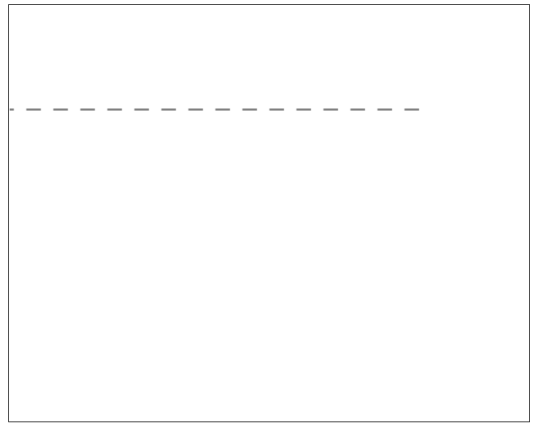
ctx.setLineDash([14, 12]);

ctx.lineDashOffset = 10;

ctx.moveTo(0, 100);

ctx.lineTo(400, 100);

ctx.stroke();

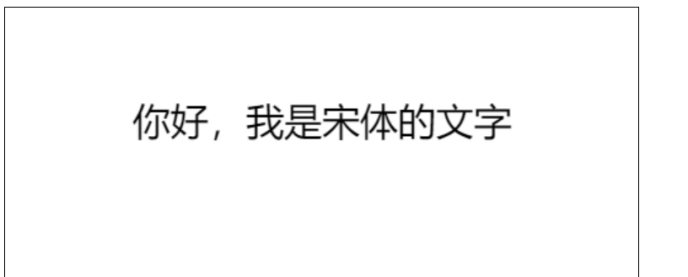


## 2.6 文本

我们可以在画布上绘制文字内容

ctx.font = "30px 微软雅黑";

ctx.fillText("你好，我是宋体的文字", 100, 100)



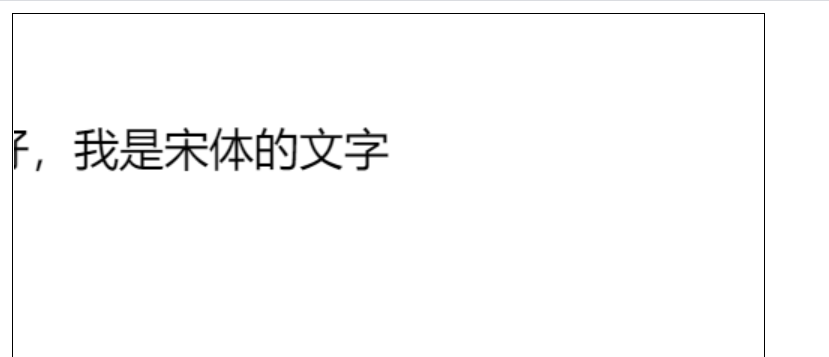
再比如设置文字的对齐方式：start (默认), end, left, right 或 center

需要注意的是文本的对齐方式的属性。注意，该对齐是基于CanvasRenderingContext2D.fillText方法的x的值

ctx.font = "30px 微软雅黑";

ctx.textAlign = "center"

ctx.fillText("你好，我是宋体的文字", 100, 100)



## 2.7 渐变

提供两种渐变方式，一种是线型渐变，一种是径向渐变

线型渐变

方法需要指定四个参数，分别表示渐变线段的开始和结束点

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

* x0起点的 x 轴坐标。
* y0起点的 y 轴坐标。
* x1终点的 x 轴坐标。
* y1终点的 y 轴坐标。

var linear = ctx.createLinearGradient(0, 0, 200, 200);

linear.addColorStop(0, "red");

linear.addColorStop(0.3, "blue");

linear.addColorStop(0.5, "yellow");

linear.addColorStop(1, "purple");

ctx.fillStyle = linear;

ctx.fillRect(10, 10, 200, 100);



addColorStop内部接受两个参数，第一个参数是当前渐变的位置（0-1之间的小数），第二个参数是颜色

径向渐变

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

* x0开始圆形的 x 轴坐标。
* y0开始圆形的 y 轴坐标。
* r0开始圆形的半径。
* x1结束圆形的 x 轴坐标。
* y1结束圆形的 y 轴坐标。
* r1结束圆形的半径。

var linear = ctx.createRadialGradient(95, 25, 25, 102, 20, 40);

linear.addColorStop(0, "red");

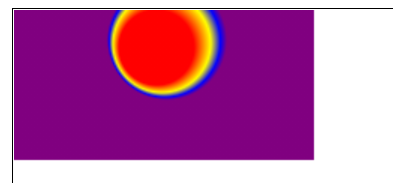
linear.addColorStop(0.5, "yellow");

linear.addColorStop(0.8, "blue");

linear.addColorStop(1, "purple");

ctx.fillStyle = linear;

ctx.fillRect(0, 0, 200, 100);



## 2.8 阴影

我们可以在画布中设置阴影的状态

ctx.shadowOffsetX = 10;

ctx.shadowOffsetY = 10;

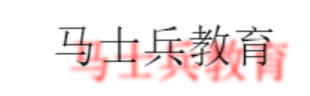
ctx.shadowBlur = 3;

ctx.shadowColor = "red";

ctx.font = "30px 宋体";

ctx.fillText("马士兵教育", 100, 100);

* shadowOffsetX阴影左右偏离的距离
* shadowOffsetY阴影上下偏离的距离
* shadowBlur模糊状态
* shadowColor阴影颜色



# 使用图片

canvas中使用drawImage来绘制图片，主要是把外部的图片导入进来绘制到画布上。

来看一个图片的简单渲染过程

// 第一步是创建一个image元素

var image = new Image();

// 用src设置图片的地址

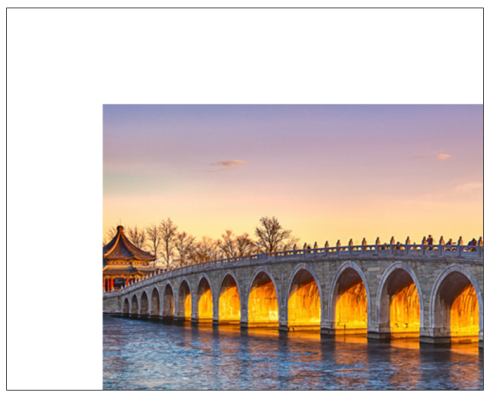
image.src = "images/2.jpg";

// 必须要在onLoad之后再进行绘制图片，否则不会渲染

image.onload = function() {

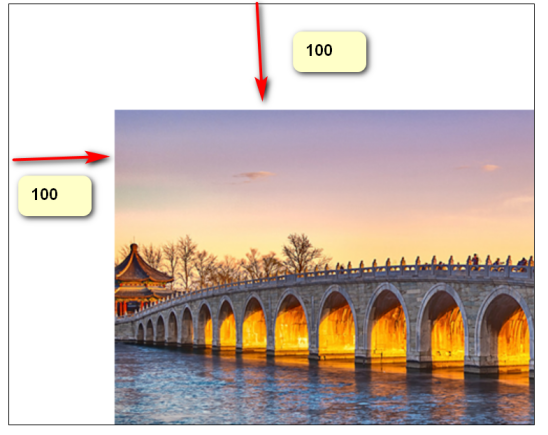
ctx.drawImage(image, 100, 100)

}



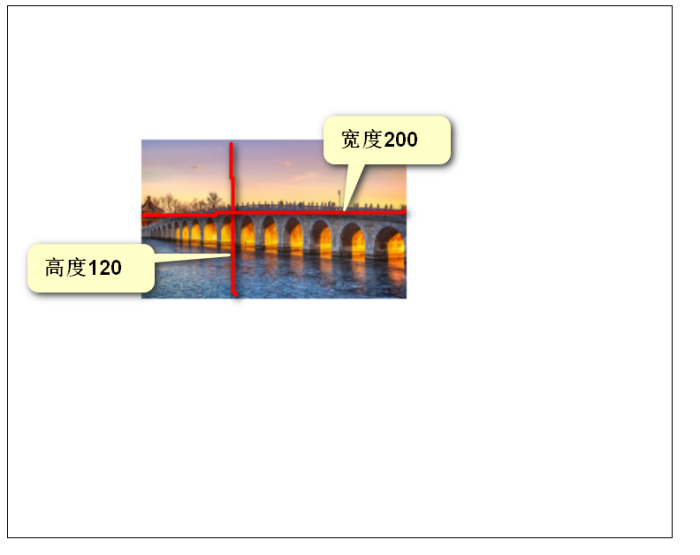
如果我们设置的参数一共是2个（不包含第一个image参数），表示的 是图片的加载位置

ctx.drawImage(image, 100, 100)



如果有4个参数，分别表示位置和宽高

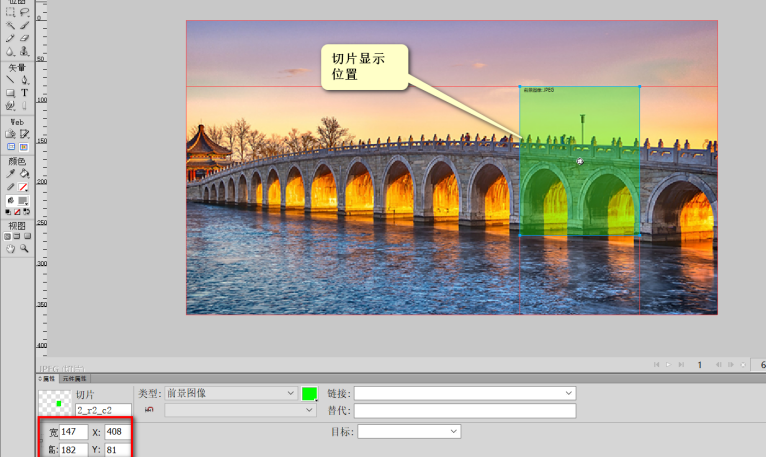
ctx.drawImage(image, 100, 100, 200, 120)



如果有8个参数，代表的是切片在图片上切下一块内容之后在画布中进行渲染

ctx.drawImage(image, 408, 81, 147, 182, 200, 200, 147, 182)

前4个参数指的是你在图片中设置切片的宽度和高度，以及切片位置



后4个参数指的是切片在画布上的位置和切片宽度高度



# 资源管理器



我们在开发游戏的时候，有一些静态资源是需要请求回来的，否则如果直接开始，某些静态资源没有，会报错，或者空白，比如我们的游戏背景图，如果没有请求回来就直接开始，页面会有空白现象

资源管理器就是当游戏需要资源全部加装完毕的时候，再开始游戏

我们现在主要是图片的资源，所以我们要在canvas渲染过程中进行对图片的资源加载

## 4.1 获取对象中属性的长度

有下面一个JSON（对象），此时我们想获取当前这个JSON属性数量

this.R = {

"gugong": "./images/0.jpg",

"niaochao": "./images/1.jpg",

"yiheyuan": "./images/2.jpg",

"tiantan": "./images/3.jpg",

"guojiadajuyuan": "./images/4.jpg"

};

此时如果我们使用this.R.length是得不到的，因为当前的this.R.length指的是获取R对象的length属性，而不是获取当前对象的属性个数，会返回undefined

正确答案是使用Object.keys()来获取当前的属性key列表，然后通过这个列表获取长度，策略就是曲线救国

Object.keys(this.R)



此时会返回一个数组，内部是存放对象中所有的属性key集合，我们可以根据这个得出我们的属性属性，因为key的数量就是当前JOSN或者对象的属性数量

var allAmount = Object.keys(this.R).length

返回的就是数量

## 4.2 资源管理器的封装

// 游戏类

function Game() {

this.dom = document.querySelector("canvas");

this.ctx = this.dom.getContext("2d");

// 添加属性，保存需要的图片地址

this.R = {

"gugong": "./images/0.jpg",

"niaochao": "./images/1.jpg",

"yiheyuan": "./images/2.jpg",

"tiantan": "./images/3.jpg",

"guojiadajuyuan": "./images/4.jpg"

};

// 获取资源图片的总数

var allAmount = Object.keys(this.R).length;

// 计数器，记录的是加载完毕的数量

var count = 0;

// 遍历R对象获取每一个路径地址

for (k in this.R) {

// 备份每一张图片的地址

var src = this.R[k];

// 创建一个图片

this.R[k] = new Image();

// 赋值src图片地址

this.R[k].src = src;

// 判断图片是否加载完成，如果完成了，计数，如果加载完毕的数量和总数量相同了，则说明资源加载完毕，开始游戏

// 备份上下文

var self = this;

this.R[k].onload = function() {

// 增加计数器

count++;

// 清屏

self.ctx.clearRect(0, 0, 600, 400);

self.ctx.font = "16px Arial";

// 设置资源加载文案

self.ctx.fillText("图片已经加载:" + count + "/" + allAmount, 10, 50)

// 判断图片是否加载完毕，如果加载完毕了，开始游戏

if (count == allAmount) {

self.start();

}

}

}

};

Game.prototype.start = function() {

var self = this;

this.ctx.drawImage(this.R["gugong"], 200, 200, 100, 80)

}



# 变形

canvas是可以进行变形的，但是变形的不是元素，而是ctx，ctx就是整个画布的渲染区域，整个画布在变形，我们需要在画布变形前，进行保存和恢复

**save()保存画布的所有状态**

**restore()恢复 canvas 状态的**

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。

可能保存的状态有：  
当前应用的变形（即移动，旋转和缩放，见下）

以及下面这些属性：strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, lineDashOffset, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation, font, textAlign, textBaseline, direction, imageSmoothingEnabled

## 5.1 translate

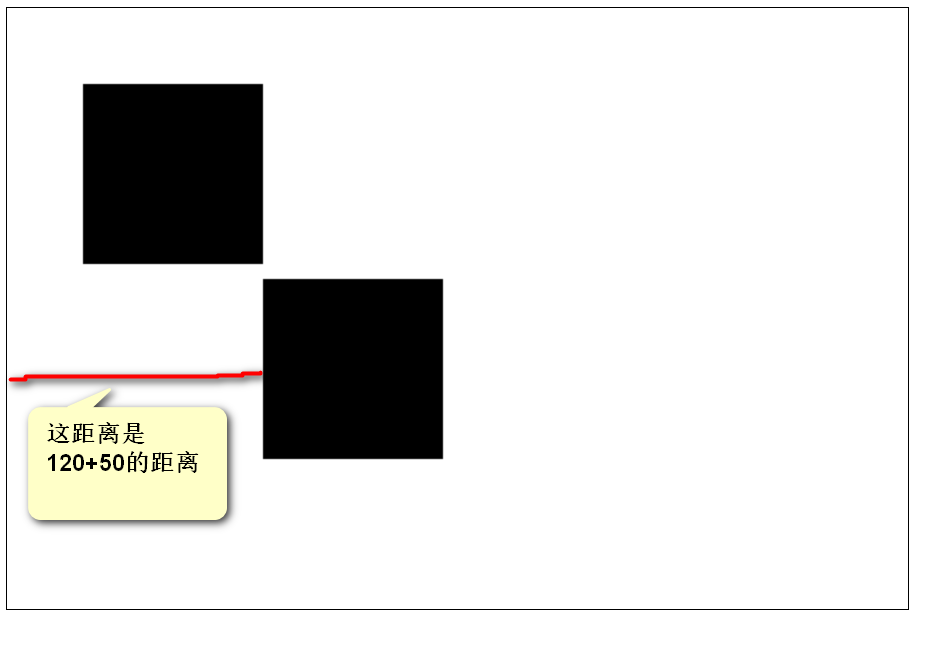
translate和CSS3的translate属性一样也是发挥着空间平移作用

ctx.translate(50, 50);

ctx.fillRect(0, 0, 120, 120);

// 渲染位置是没有存档之前的位置

ctx.fillRect(120, 220, 120, 120)



此时我们发现上图的第二次的渲染位置和预期不一致，上图的第二次渲染预期x的位置是120，但是由于第一次进行了变形操作，整体画布移动了50，所以位置变成170

我们知道了变形实际上都是将整个画布进行的变形，所以如果一旦我们的变形操作多了，画布将变得不可控

所以如果我们使用到变形，一定记住下面的规律：变形之前要先备份，将世界和平的状态进行备份，然后再变形，变形完毕后一定要恢复到世界和平的样子，不要影响下一次的操作

下面的代码是正确的操作

// 保存

ctx.save();

ctx.translate(50, 50);

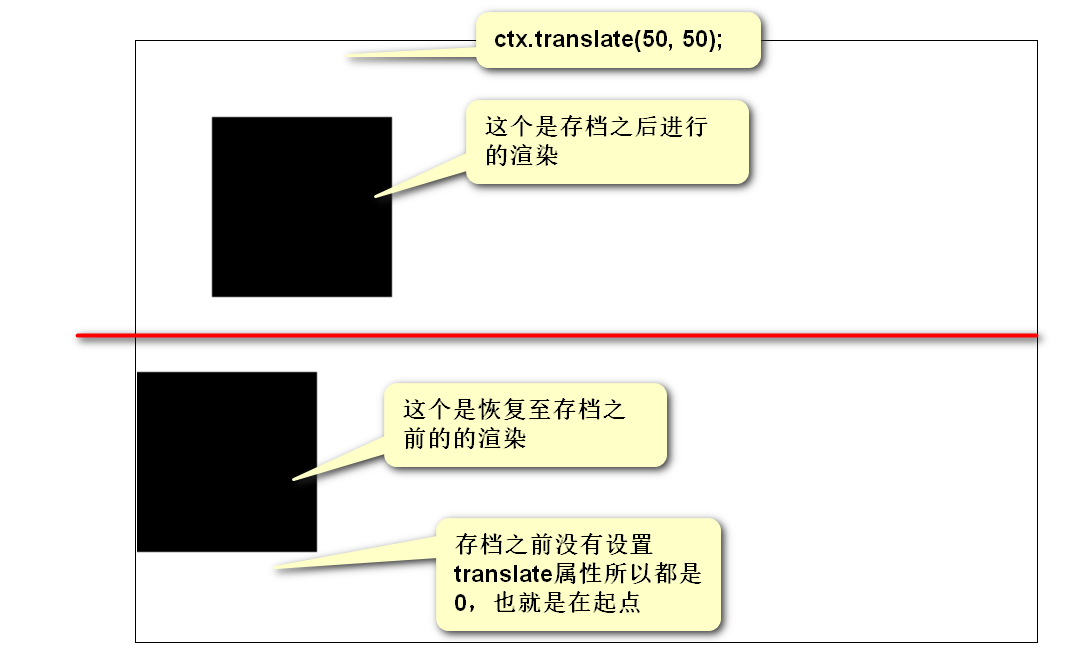
ctx.fillRect(0, 0, 120, 120);

// 恢复

ctx.restore();

// 渲染位置是没有存档之前的位置

ctx.fillRect(0, 220, 120, 120)



## 5.2 rotate

ctx.save();

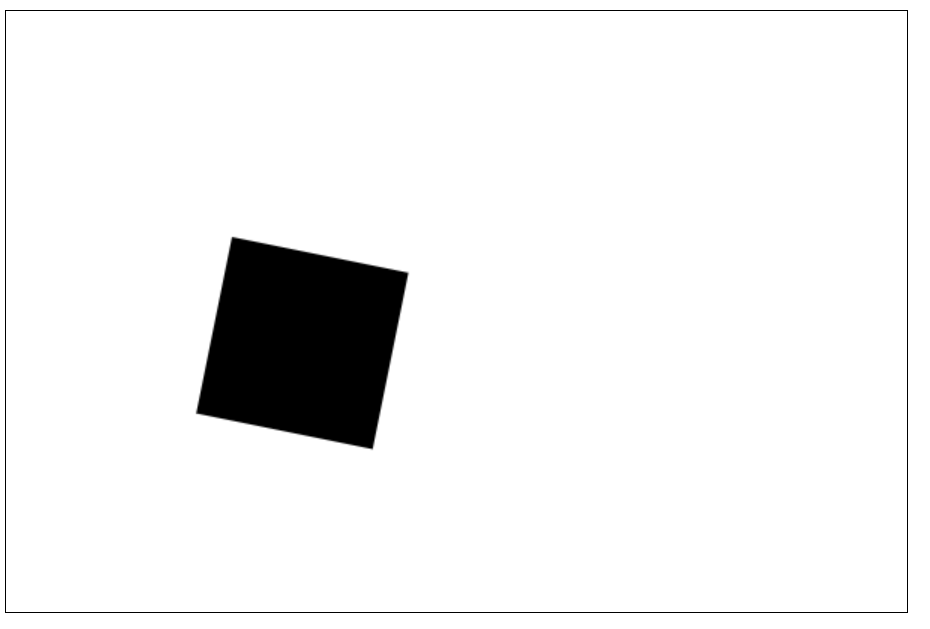
ctx.translate(150, 150);

ctx.rotate(deg)

ctx.fillRect(0, 0, 120, 120);

// 恢复

ctx.restore();



## 5.3 scale

和我们CSS3中的scale不同的是，canvas的scale是两个参数，分别代表了x，y。都必须是正数。1.0为正常比例，大于1代表放大，小于1代表缩小

// 保存

ctx.save();

ctx.translate(150, 150);

ctx.scale(0.5, 0.5)

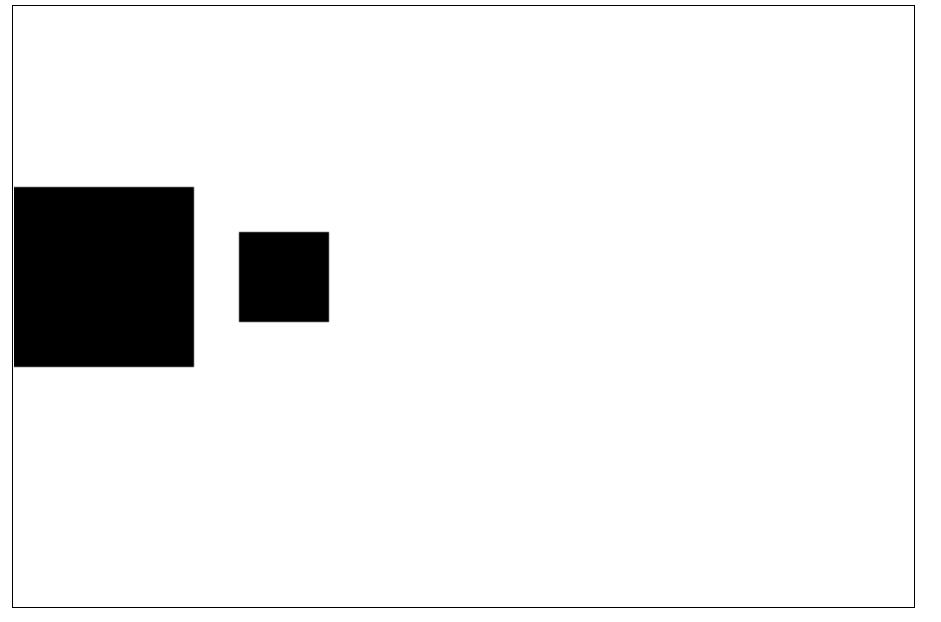
ctx.fillRect(0, 0, 120, 120);

// 恢复

ctx.restore();

// 渲染位置是没有存档之前的位置

ctx.fillRect(0, 120, 120, 120)

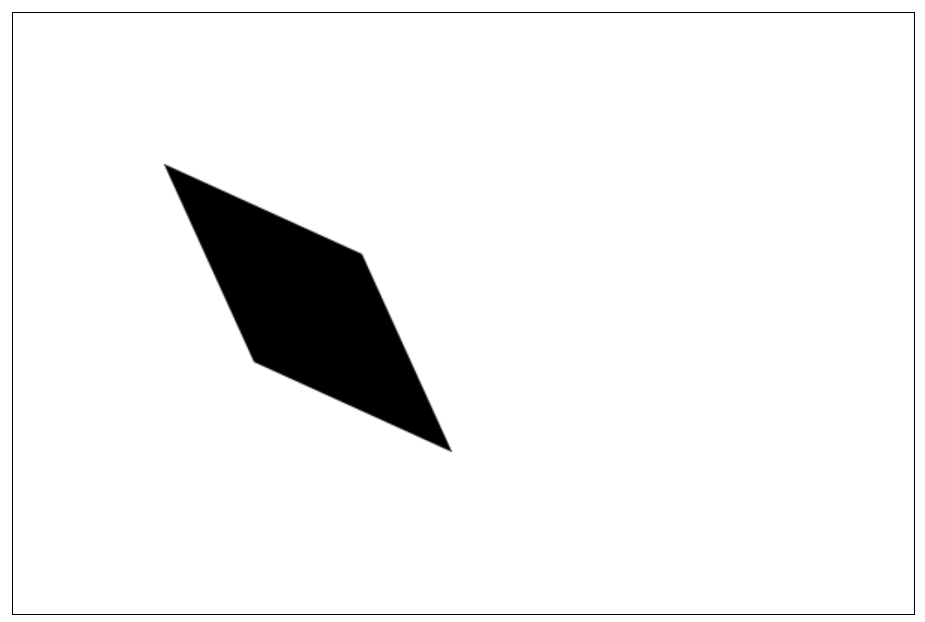
s

我们可以使用transform进行属性的translate、rotate、scale的综合写法

transform(a, b, c, d, e, f)

* a (m11)水平方向的缩放
* b(m12)竖直方向的倾斜偏移
* c(m21)水平方向的倾斜偏移
* d(m22)竖直方向的缩放
* e(dx)水平方向的移动
* f(dy)竖直方向的移动

ctx.transform(0.5, 1.1, 1.1, 0.5, 100, 100)



# 合成

# 合成其实就是我们常见的蒙板状态，本质就是如何进行压盖，如何进行显示

我们可以通过globalCompositeOperation来设置压盖顺序

比如我们此时画了一个方和一个圆，第一次画的是方，第二次画的是圆，所以会出现圆压盖方的现象

ctx.fillStyle = "skyblue";

ctx.fillRect(100, 100, 100, 100)

ctx.fillStyle = "deeppink";

ctx.beginPath();

ctx.arc(200, 200, 60, 0, 7, false);

ctx.fill()



比如此时我们想让粉色在下面

ctx.globalCompositeOperation = "destination-over"



|  |  |  |
| --- | --- | --- |
| 默认source-over | "destination-over" | source-in |
| source-out | source-atop | destination-in |
| destination-out | destination-atop | lighter |