自定义View之Canvas

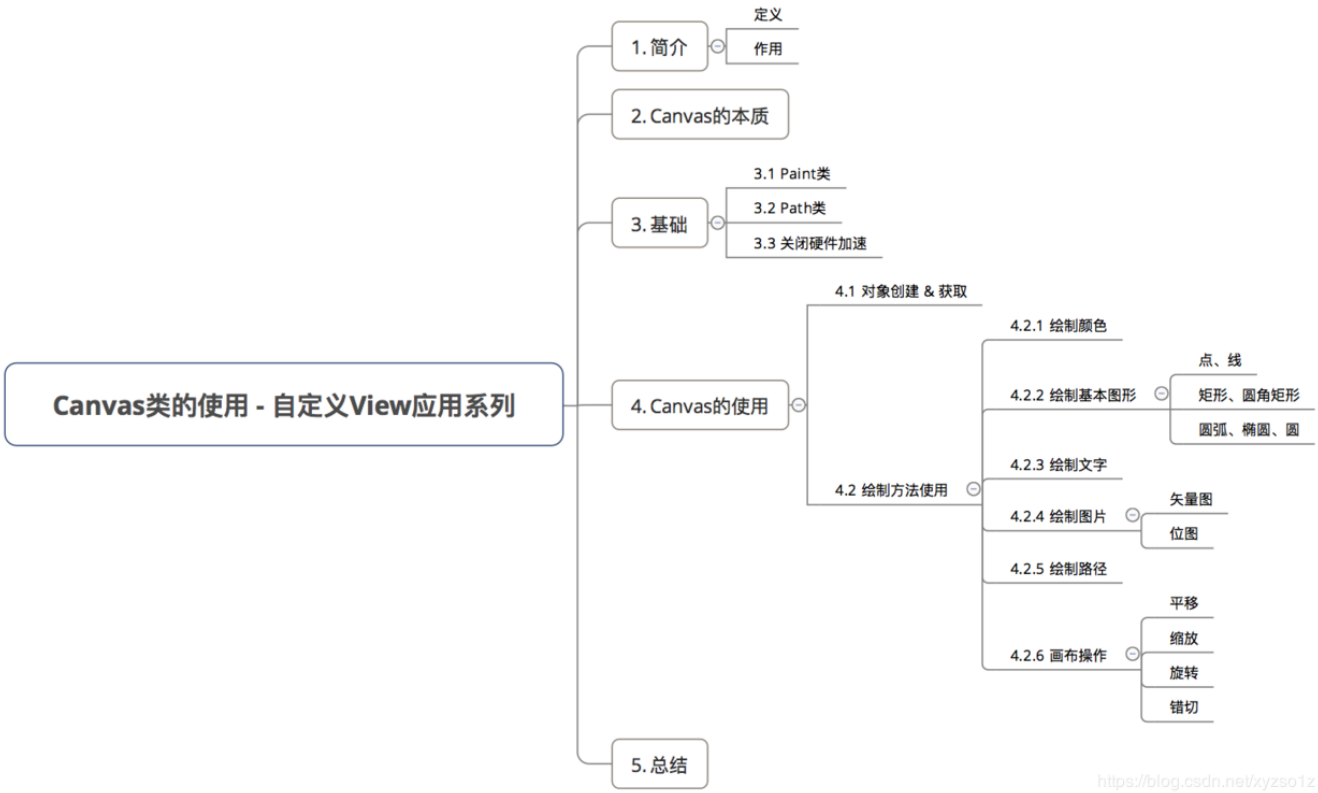
原创 xyzso1z 最后发布于2020-03-29 22:07:46 阅读数 25 收藏 编辑 展开

> 前言

查看Android总结专题

自定义View总结：

- View基础
- measure方法
- layout方法
- draw方法
- Path类
- Canvas类



1. 简介

- 定义：画布，是一种绘制时的规则
- 作用：规定绘制内容时的规则、内容

> 1. 绘制内容是根据画布的规定绘制在屏幕上的
> 2. 理解为： 画布只是绘制时的规则，但内容实际上是绘制在屏幕上

2. Canvas的本质

- 绘制内容时根据**画布(Canvas)**的规定绘制在屏幕上
- 画布(Canvas)只会绘制时的规则，但实际上是绘制在屏幕上的

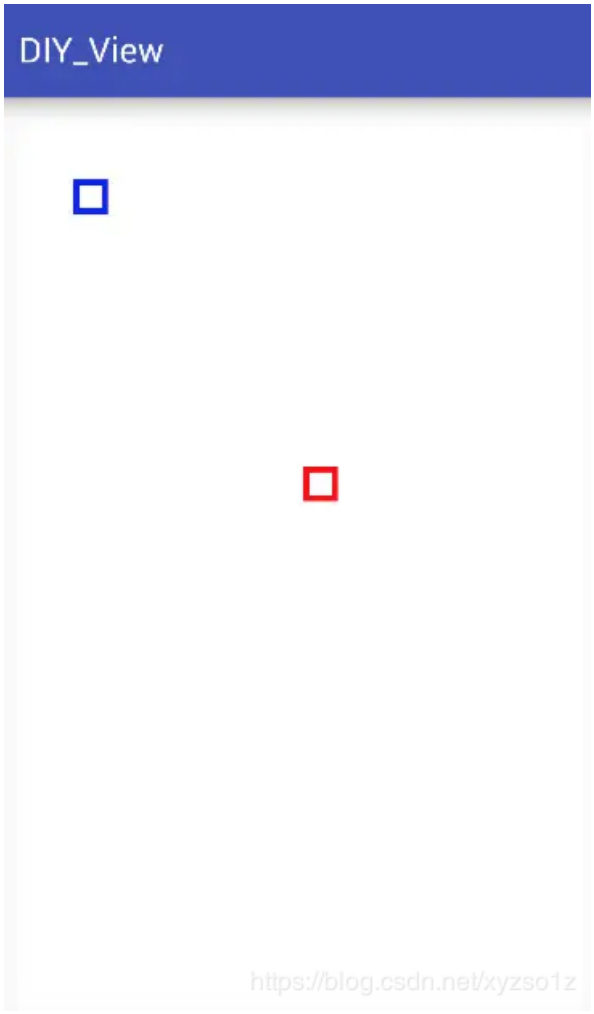
为了更好更好说明绘制内容的本质和Canvas，请看下面例子

2.1 实例

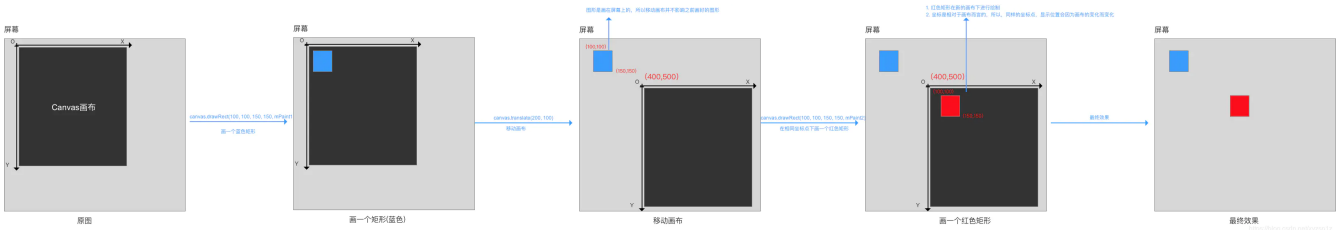
- 实例： 1. 先画一个矩形（蓝色）；2. 然后移动画布；3. 再画一个矩形（红色）
- 代码

```
// 画一个矩形(蓝色)
        canvas.drawRect(100, 100, 150, 150, mPaint1);

        // 将画布的原点移动到(400,500)
        canvas.translate(400,500);

        // 再画一个矩形(红色)
        canvas.drawRect(100, 100, 150, 150, mPaint2);
```

- 效果图



- 具体流程



- 总结
  绘制内容是根据画布的规定绘制在屏幕上的。

1. 内容实际上是绘制在屏幕上；
2. 画布，即Canvas，只是规定了绘制内容的规则；
3. 内容的位置有坐标决定，而坐标是相对画布而言的。

3.基础

3.1 Paint类

- 定义：画笔
- 作用：确定绘制内容的具体效果（如颜色、大小等等）
- 具体使用：

1. 创建一个画笔对象
2. 画笔设置，即设置绘制内容的具体效果
3. 初始化画笔（尽量选在在View的构造函数）
   具体使用如下：

```
// 步骤1：创建一个画笔
private Paint mPaint = new Paint();

// 步骤2：初始化画笔
// 根据需求设置画笔的各种属性，具体如下：

    private void initPaint() {

        // 设置最基本的属性
        // 设置画笔颜色
        // 可直接引入Color类，如Color.red等
        mPaint.setColor(int color);
        // 设置画笔模式
         mPaint.setStyle(Style style);
        // Style有3种类型：
        // 类型1：Paint.Style.FILLANDSTROKE（描边+填充）
        // 类型2：Paint.Style.FILL（只填充不描边）
        // 类型3：Paint.Style.STROKE（只描边不填充）
        // 具体差别请看下图：
        // 特别注意：前两种就相差一条边
        // 若边细是看不出分别的；边粗就相当于加粗

        //设置画笔的粗细
        mPaint.setStrokeWidth(float width)

        // 如设置画笔宽度为10
```

```
        // 如设置画笔宽度为10px
        mPaint.setStrokeWidth(10f);

        // 不常设置的属性
        // 得到画笔的颜色
        mPaint.getColor()
        // 设置Shader
        // 即着色器，定义了图形的着色、外观
        // 可以绘制出多彩的图形
        // 具体请参考文章：http://blog.csdn.net/iispring/article/details/50500106
        Paint.setShader(Shader shader)

        //设置画笔的a,r,p,g值
       mPaint.setARGB(int a, int r, int g, int b)
         //设置透明度
        mPaint.setAlpha(int a)
       //得到画笔的Alpha值
        mPaint.getAlpha()


        // 对字体进行设置（大小、颜色）
        //设置字体大小
          mPaint.setTextSize(float textSize)

        // 文字Style三种模式：
          mPaint.setStyle(Style style);
        // 类型1：Paint.Style.FILLANDSTROKE（描边+填充）
        // 类型2：Paint.Style.FILL（只填充不描边）
        // 类型3：Paint.Style.STROKE（只描边不填充）

      // 设置对齐方式
      setTextAlign（）
      // LEFT：左对齐
      // CENTER：居中对齐
      // RIGHT：右对齐

        //设置文本的下划线
          setUnderlineText(boolean underlineText)

        //设置文本的删除线
        setStrikeThruText(boolean strikeThruText)

         //设置文本粗体
        setFakeBoldText(boolean fakeBoldText)

           // 设置斜体
        Paint.setTextSkewX(-0.5f);


        // 设置文字阴影
        Paint.setShadowLayer(5,5,5,Color.YELLOW);
    }

// 步骤3：在构造函数中初始化
    public CarsonView(Context context, AttributeSet attrs) {
        super(context, attrs);
        initPaint();
    }
```

Style模式效果如下：

3.2 关闭硬件加速

- 在Android 4.0的设备上，在打开硬件加速的情况下，使用自定义View可能出现问题。
- 在测试前，**先关闭硬件加速**，具体如下：

```
// 在AndroidMenifest.xml的application节点添加
android:hardwareAccelerated="false"
```

4. Canvas的使用

4.1 对象创建和获取

Canvas对象获取方法有4种：

```
// 方法1
// 利用空构造方法直接创建对象
Canvas canvas = new Canvas();

// 方法2
// 通过传入装载画布Bitmap对象创建Canvas对象
// CBitmap上存储所有绘制在Canvas的信息
Canvas canvas = new Canvas(bitmap)

// 方法3
// 通过重写View.onDraw（）创建Canvas对象
// 在该方法里可以获得这个View对应的Canvas对象

   @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        //在这里获取Canvas对象
    }

// 方法4
// 在SurfaceView里画图时创建Canvas对象

        SurfaceView surfaceView = new SurfaceView(this);
        // 从SurfaceView的surfaceHolder里锁定获取Canvas
        SurfaceHolder surfaceHolder = surfaceView.getHolder();

```

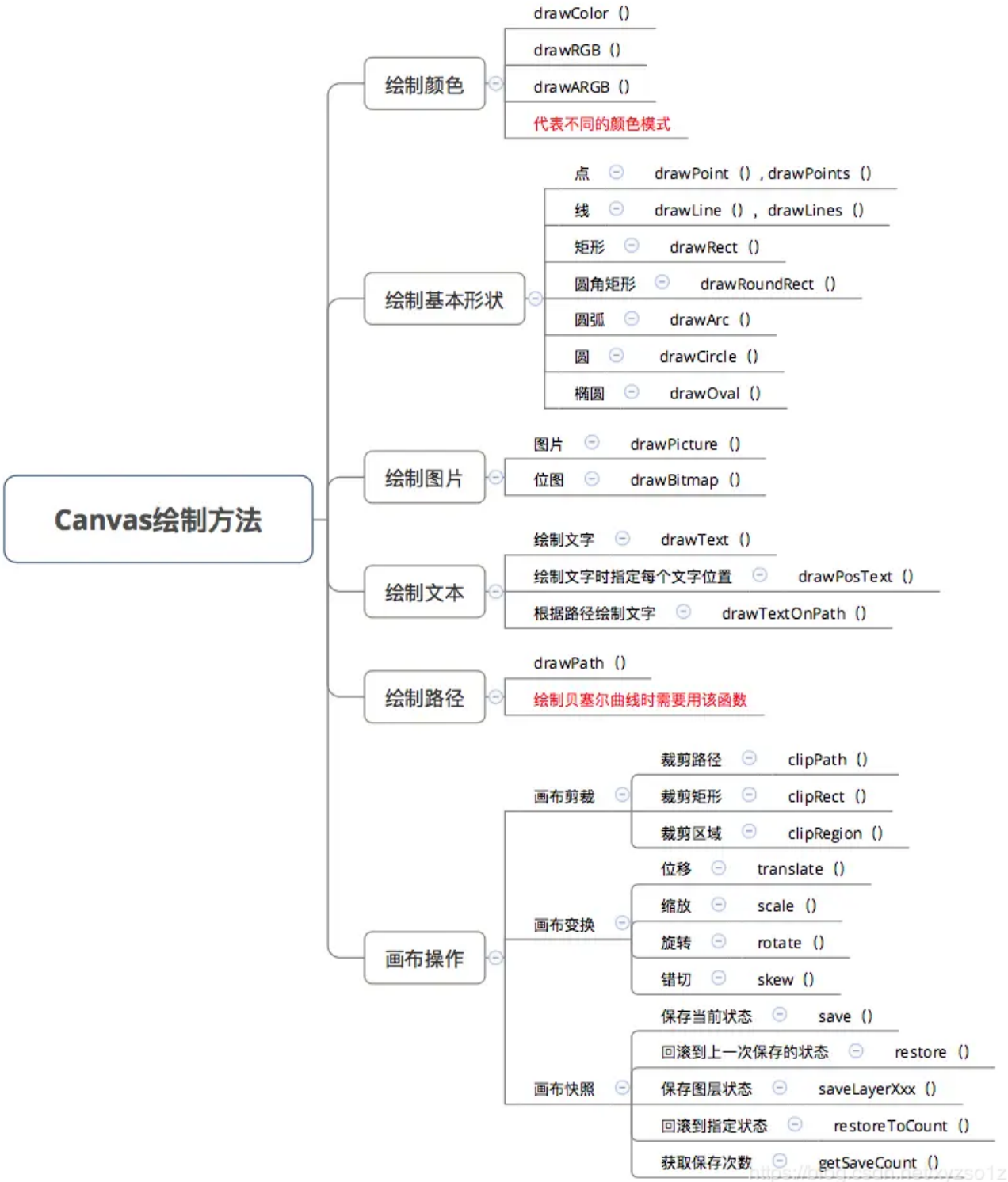
```
27          //获取Canvas
28          Canvas c = surfaceHolder.lockCanvas();
29
30          // ...（进行Canvas操作）
31          // Canvas操作结束之后解锁并执行Canvas
            surfaceHolder.unlockCanvasAndPost(c);
```

官方推荐方法4来创建并获取Canvas,因为SurfaceView里有一条线程是专门用于画图，所以方法4的画图性能最好，并使用于高刷新频率的图形。而方法3刷新频率低于方法4,但系统花销小，节省资源。

4.2 绘制方法是用

- 利用Canvas类可画出很多内容，如图形、文字、线条等；

- 对应方法如下：



下面逐个方法进行详细讲解：
Canva具体使用时是复写的onDraw()里

4.2.1绘制颜色

- 作用：将颜色填充整个画布，常用于绘制底色
- 具体使用

```
1
2
    // 传入一个Color类的常量参数来设置画布颜色
    // 绘制蓝色
```

```
    canvas.drawColor(Color.BLUE);
```

4.2.2 绘制基本图形

1. 绘制点

```
// 特别注意：需要用到画笔Paint
// 所以之前记得创建画笔
// 为了区分，这里使用了两个不同颜色的画笔

// 描绘一个点
// 在坐标(200,200)处
canvas.drawPoint(300, 300, mPaint1);

// 绘制一组点，坐标位置由float数组指定
// 此处画了3个点，位置分别是：（600,500）、（600,600）、（600,700）
canvas.drawPoints(new float[]{
                600,500,
                600,600,
                600,700
        },mPaint2);
```

2. 绘制直线（drawLine）

```
// 画一条直线
// 在坐标(100,200)，(700,200)之间绘制一条直线
  canvas.drawLine(100,200,700,200,mPaint1);

// 绘制一组线
// 在坐标(400,500)，(500,500)之间绘制直线1
// 在坐标(400,600)，(500,600)之间绘制直线2
        canvas.drawLines(new float[]{
                400,500,500,500,
                400,600,500,600
        },mPaint2);
    }
```

3. 绘制矩形(drawRect)

- 原理：矩形的对角线顶点确定一个矩形（一般采用左上角和右下角的两个点的坐标）
- 具体使用：

```
// 关于绘制矩形，Canvas提供了三种重载方法

        // 方法1：直接传入两个顶点的坐标
        // 两个顶点坐标分别是：（100,100），（800,400）
         canvas.drawRect(100,100,800,400,mPaint);

         // 方法2：将两个顶点坐标封装为RectRectF
         Rect rect = new Rect(100,100,800,400);
         canvas.drawRect(rect,mPaint);

         // 方法3：将两个顶点坐标封装为RectF

```

```
        RectF rectF = new RectF(100,100,800,400);
        canvas.drawRect(rectF,mPaint);

        // 特别注意：Rect类和RectF类的区别
        // 精度不同：Rect = int & RectF = float

        // 三种方法画出来的效果是一样的。
```

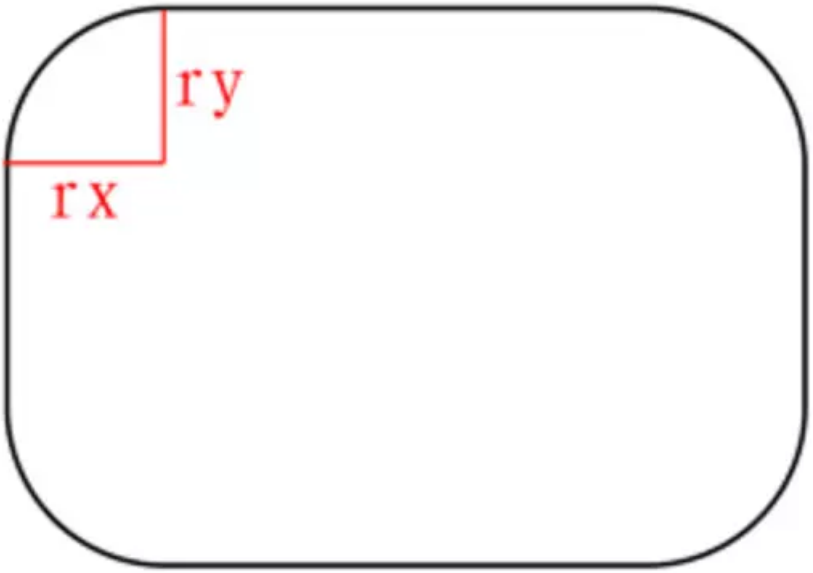
4. 绘制圆角矩形

- 原理：矩形的对角线顶点确定一个矩形（类似于绘制矩形）
- 具体使用：

```
        // 方法1：直接传入两个顶点的坐标
        // API21时才可使用
        // 第5、6个参数：rx、ry是圆角的参数，下面会详细描述
        canvas.drawRoundRect(100,100,800,400,30,30,mPaint);

        // 方法2：使用RectF类
        RectF rectF = new RectF(100,100,800,400);
        canvas.drawRoundRect(rectF,30,30,mPaint);
```

- 与矩形相比，圆角矩形多了两个参数rx和ry

- 圆角矩形的角是椭圆的圆弧



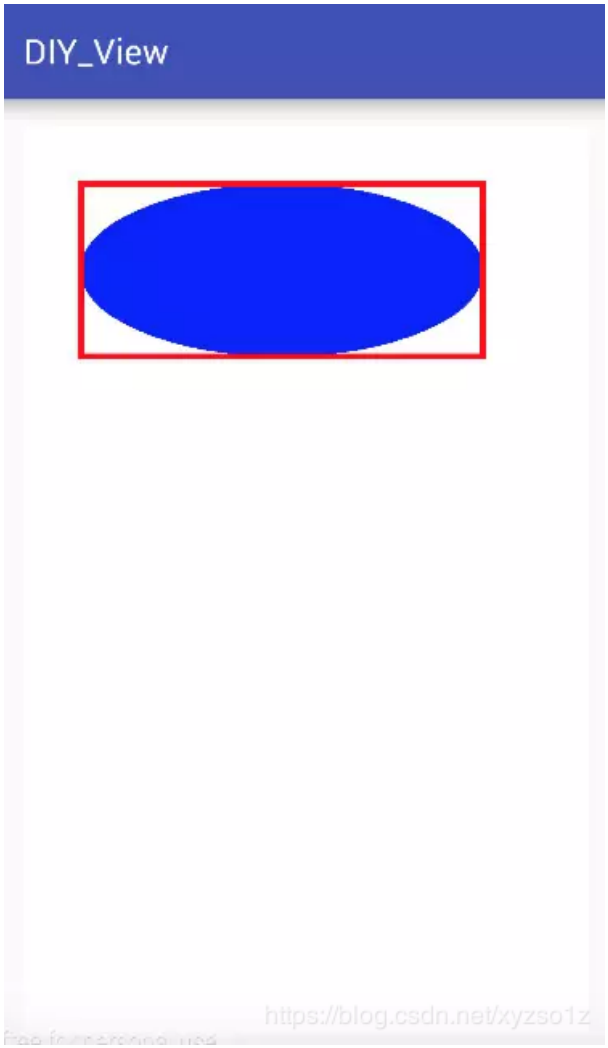
- 特别注意：当rx大于宽度一半，ry大于高度一半时，画出来的为椭圆

> 实际上，在rx为宽度的一半，ry为高度的一半时，刚好是一个椭圆；但由于当rx大于宽度一半，ry大于高度一半时，无法计算出圆弧，所以drawRoundRect对大于该数值的参数进行了修正，凡是大于一半的参数均按照一半来处理

5. 绘制椭圆（drawOval）

- 原理：矩形的对角线顶点确定矩形，根据传入矩形的长宽作为长轴和短轴画椭圆
- 具体使用

```
        // 方法1：使用RectF类
        RectF rectF = new RectF(100,100,800,400);
        canvas.drawOval(rectF,mPaint);

        // 方法2：直接传入与矩形相关的参数
        canvas.drawOval(100,100,800,400,mPaint);

        // 为了方便表示，画一个和椭圆一样参数的矩形
         canvas.drawRect(100,100,800,400,mPaint);
```



6. 绘制圆（drawCircle）

- 原理：圆心坐标+半径决定圆
- 具体使用：

```
// 参数说明：
// 1、2：圆心坐标
// 3：半径
// 4：画笔

// 绘制一个圆心坐标在(500,500)，半径为400 的圆。
    canvas.drawCircle(500,500,400,mPaint);
```

7. 绘制圆弧

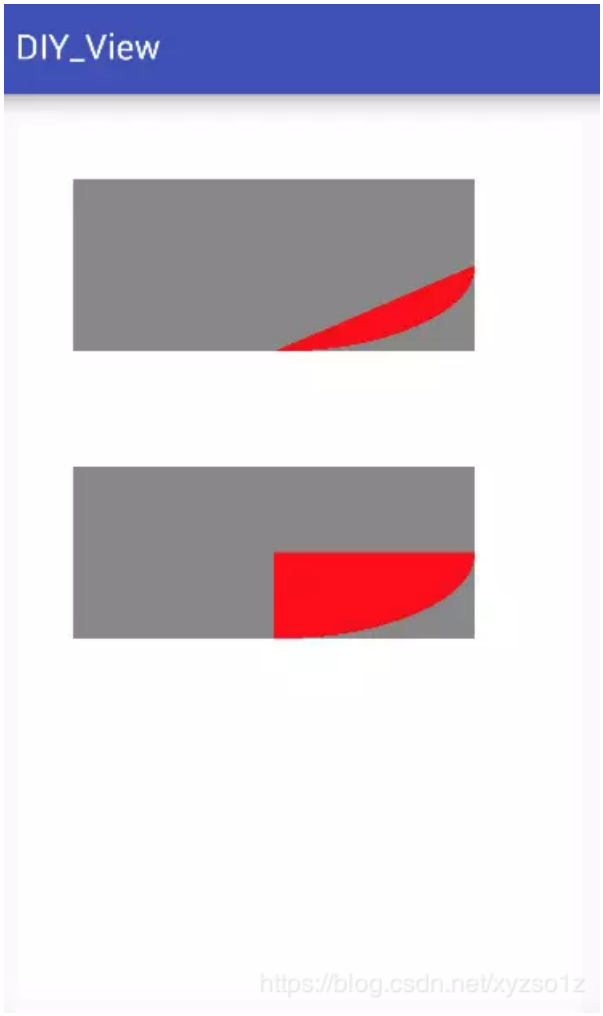
- 原理：通过圆弧角度的起始位置和扫过地角度确定圆弧
- 具体使用

```
// 绘制圆弧共有两个方法
// 相比于绘制椭圆，绘制圆弧多了三个参数：
startAngle  // 确定角度的起始位置
sweepAngle // 确定扫过的角度
useCenter   // 是否使用中心（下面会详细说明）

```

```
// 方法1
public void drawArc(@NonNull RectF oval, float startAngle, float sweepAngle, boolean useCenter, @NonNu

// 方法2
public void drawArc(float left, float top, float right, float bottom, float startAngle,
            float sweepAngle, boolean useCenter, @NonNull Paint paint) {}
```

为了理解第三个参数：`userCenter`，看一下示例：

```
// 以下示例：绘制两个起始角度为0度、扫过90度的圆弧
// 两者的唯一区别就是是否使用了中心点

    // 绘制圆弧1(无使用中心)
        RectF rectF = new RectF(100, 100, 800,400);
        // 绘制背景矩形
        canvas.drawRect(rectF, mPaint1);
        // 绘制圆弧
        canvas.drawArc(rectF, 0, 90, false, mPaint2);

   // 绘制圆弧2(使用中心)
        RectF rectF2 = new RectF(100,600,800,900);
        // 绘制背景矩形
        canvas.drawRect(rectF2, mPaint1);
        // 绘制圆弧
        canvas.drawArc(rectF2,0,90,true,mPaint2);
```



可以得出：

- 不使用中心点：圆弧的形状=（起、止点连线+圆弧）构成的面积
- 使用中心点：圆弧面积=（起点、圆心连线+止点、圆心连线+圆弧）构成的面积

4.2.3 绘制文字

绘制文字分为三种应用场景

- 情景1： 指定文本开始的位置

1. 即指定文本基线位置
2. 基线x默认在字符串左侧，基线y默认在字符串下方

- 情景2：指定每个文字的位置
- 情景3：指定路径，并根据路径绘制文字

情况1：指定文本开始的位置

```
// 参数text：要绘制的文本
// 参数x，y：指定文本开始的位置（坐标）

// 参数paint：设置的画笔属性
    public void drawText (String text, float x, float y, Paint paint)


```

```
// 实例
canvas.drawText("abcdefg",300,400,mPaint1);



// 仅绘制文本的一部分
// 参数start，end：指定绘制文本的位置
// 位置以下标标识，由0开始
    public void drawText (String text, int start, int end, float x, float y, Paint paint)
    public void drawText (CharSequence text, int start, int end, float x, float y, Paint paint)

// 对于字符数组char[]
// 截取文本使用起始位置(index)和长度(count)
    public void drawText (char[] text, int index, int count, float x, float y, Paint paint)

// 实例：绘制从位置1-3的文本
canvas.drawText("abcdefg",1,4,300,400,mPaint1);

        // 字符数组情况
        // 字符数组(要绘制的内容)
        char[] chars = "abcdefg".toCharArray();

        // 参数为（字符数组 起始坐标 截取长度 基线x 基线y 画笔)
        canvas.drawText(chars,1,3,200,500,textPaint);
        // 效果同上
```

情况2：分别指定文本的位置

```
// 参数text：绘制的文本
// 参数pos：数组类型，存放每个字符的位置（坐标）
// 注意：必须指定所有字符位置
 public void drawPosText (String text, float[] pos, Paint paint)

// 对于字符数组char[],可以截取部分文本进行绘制
// 截取文本使用起始位置(index)和长度(count)
    public void drawPosText (char[] text, int index, int count, float[] pos, Paint paint)

// 特别注意：
// 1. 在字符数量较多时，使用会导致卡顿
// 2. 不支持emoji等特殊字符，不支持字形组合与分解

  // 实例
  canvas.drawPosText("abcde", new float[]{
                100, 100,    // 第一个字符位置
                200, 200,    // 第二个字符位置
                300, 300,    // ...
                400, 400,
                500, 500
        }, mPaint1);



// 数组情况（绘制部分文本）
       char[] chars = "abcdefg".toCharArray();

        canvas.drawPosText(chars, 1, 3, new float[]{
```

```
                300, 300,    // 指定的第一个字符位置
                400, 400,    // 指定的第二个字符位置
                500, 500,    // 指定的第三个字符位置

        }, mPaint1);
```

情况3：指定路径，并根据路径绘制文字

```
// 在路径(540,750,640,450,840,600)写上"在Path上写的字:Carson_Ho"字样
        // 1.创建路径对象
        Path path = new Path();
        // 2. 设置路径轨迹
        path.cubicTo(540, 750, 640, 450, 840, 600);
         // 3. 画路径
        canvas.drawPath(path,mPaint2);
        // 4. 画出在路径上的字
        canvas.drawTextOnPath("画出在路径上的字", path, 50, 0, mPaint2);
```

4.2.4 绘制图片

绘制图片分为：绘制矢量图（drawPicture）和绘制位图（drawBitmap）

绘制矢量图(drawPicture)

- 作用：绘制矢量图的内容，即绘制存储在矢量图里某个时刻Canva绘制内容的操作。

> 矢量图(Picture)的作用：存储某个时刻Canvas绘制内容得操作

- 应用场景：绘制之前绘制过的内容

> 1. 相比于再次调用各种绘图API，使用Picture能节省操作和时间
> 2. 如果不手动调用，录制的内容不会显示在屏幕上，只是存储起来

特别注意：使用绘制矢量图前请关闭硬件加速，以免引起不必要的问题！

- 具体使用

```
// 获取宽度
Picture.getWidth ();

// 获取高度
Picture.getHeight ()

// 开始录制
// 即将Canvas中所有的绘制内容存储到Picture中
// 返回一个Canvas
Picture.beginRecording（int width, int height）

// 结束录制
Picture.endRecording ()
```

```
// 将Picture里的内容绘制到Canvas中
Picture.draw (Canvas canvas)

// 还有两种方法可以将Picture里的内容绘制到Canvas中
// 方法2：Canvas.drawPicture ( )
// 方法3：将Picture包装成为PictureDrawable，使用PictureDrawable的draw方法绘制。

// 下面会详细介绍
```

- 一般使用的具体步骤

```
// 步骤1：创建Picture对象
Picture mPicture = new Picture();

// 步骤2：开始录制
mPicture.beginRecording（int width, int height）;

// 步骤3：绘制内容 or 操作Canvas
canvas.drawCircle(500,500,400,mPaint);
...（一系列操作）

// 步骤4：结束录制
mPicture.endRecording ();

步骤5：某个时刻将存储在Picture的绘制内容绘制出来
mPicture.draw (Canvas canvas);
```

- 实例介绍
  将坐标系移动到(450，650)；绘制一个圆，将上述Canvas操作录制下来，并在某个时刻出来。

步骤1:创建Picture对象

```
Picture mPicture = new Picture();
```

步骤2:开始录制

```
Canvas recordingCanvas = mPicture.beginRecording(500, 500);

// 注：要创建Canvas对象来接收beginRecording()返回的Canvas对象
```

步骤3:绘制内容/操作Canvas

```
        // 位移
        // 将坐标系的原点移动到(450,650)
        recordingCanvas.translate(450,650);

        // 记得先创建一个画笔
        Paint paint = new Paint();
        paint.setColor(Color.BLUE);
        paint.setStyle(Paint.Style.FILL);

        // 绘制一个圆

```

```
        // 圆心为(0,0),半径为100
        recordingCanvas.drawCircle(0,0,100,paint);
```

步骤4:结束录制

```
mPicture.endRecording();
```

步骤5:将存储在Picture的绘制内容绘制出来
有三种方法：

```
Picture.draw (Canvas canvas)

Canvas.drawPicture ()

PictureDrawable.draw ()
```

主要区别如下：

主要区别/类型	是否对Canvas状态(clip、Matrix)有影响	对绘制结
Picture.draw()	有	
Canvas.drawPicture()	无	
PictureDrawable.draw()	无	

绘制位图(drawBitmap)

- 作用：将已有的图片转换为位图（Bitmap），最后再绘制到Canvas上

> 位图，即平时我们使用的图片资源

- 获取BIgmap对象的方式

要绘制Bitmap，就要先获取Bitmap对象，具体获取方式如下：

获取方式	具体形式
通过Bitmap创建	• 复制一个已有的Bitmap • 创建一个空白的Bitmap
通过BitmapDrawable获取	从资源文件、内存卡、网络等地方获取一张图片并转换为Bitmap（内容不可变）
通过BitmapFactory获取	

通过BitmapFactory获取Bitmap(从不同位置获取)

```
// 共3个位置：资源文件、内存卡、网络

// 位置1：资源文件(drawable/mipmap/raw)
        Bitmap bitmap = BitmapFactory.decodeResource(mContext.getResources(),R.raw.bitmap);

// 位置2：资源文件(assets)
        Bitmap bitmap=null;
        try {
            InputStream is = mContext.getAssets().open("bitmap.png");
            bitmap = BitmapFactory.decodeStream(is);
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

// 位置3：内存卡文件
    Bitmap bitmap = BitmapFactory.decodeFile("/sdcard/bitmap.png");

// 位置4：网络文件:
// 省略了获取网络输入流的代码
        Bitmap bitmap = BitmapFactory.decodeStream(is);
        is.close();
```

绘制Bitmap

```
// 方法1
// 注：图片左上角位置默认为坐标原点。
    public void drawBitmap (Bitmap bitmap, Matrix matrix, Paint paint)

 // 方法2
 // 参数 left、top指定了图片左上角的坐标(距离坐标原点的距离)：
    public void drawBitmap (Bitmap bitmap, float left, float top, Paint paint)

// 方法3
// 参数（src，dst） = 两个矩形区域
// Rect src：指定需要绘制图片的区域（即要绘制图片的哪一部分）
// Rect dst 或RectF dst：指定图片在屏幕上显示(绘制)的区域
    public void drawBitmap (Bitmap bitmap, Rect src, Rect dst, Paint paint)

// 方法4
    public void drawBitmap (Bitmap bitmap, Rect src, RectF dst, Paint paint)
```

方法3、4应用场景：实现动态效果

> 动态效果 = 逐渐绘制图形部分

4.2.5 绘制路径

```
// 通过传入具体路径Path对象 & 画笔
canvas.drawPath(mPath, mPaint)
```

4.2.6 画布操作

- 作用：改变画布的性质

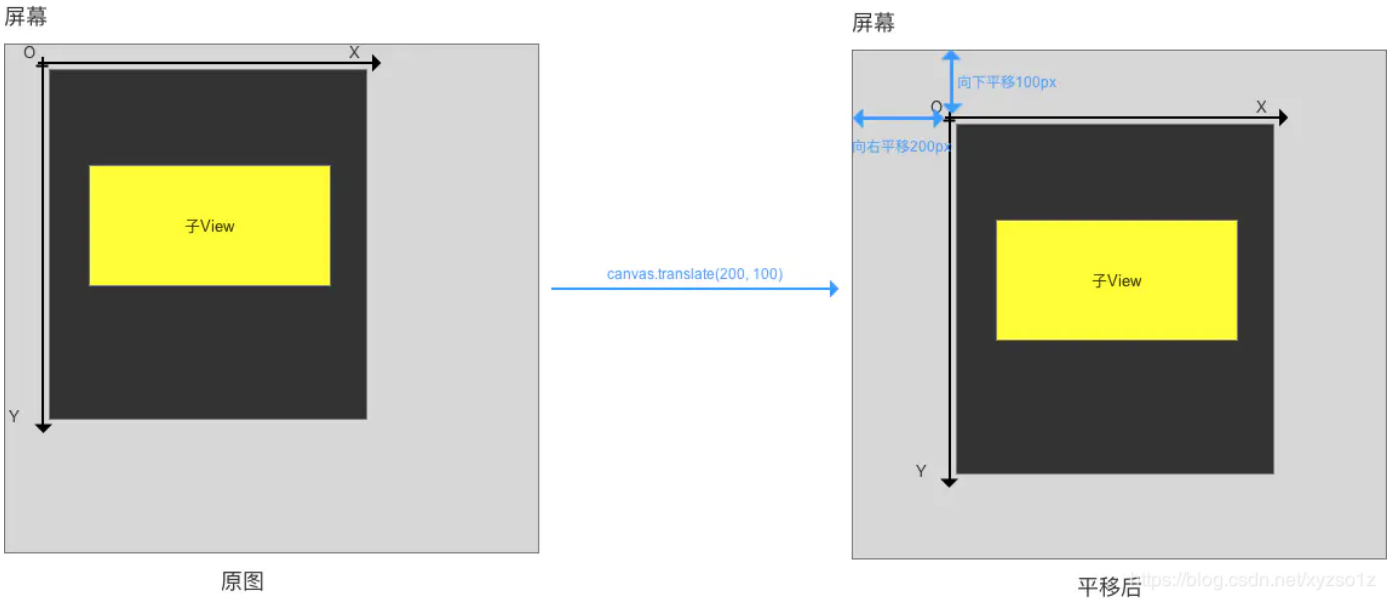
> 改变后，任何的后续操作都会受到影响

1. **画布变换**

平移（translate）

- 作用：移动画布（实际上是移动坐标系，如下图）
- 具体使用

```
// 将画布原点向右移200px，向下移100px
canvas.translate(200, 100)
// 注：位移是基于当前位置移动，而不是每次都是基于屏幕左上角的(0,0)点移动
```



原图　　平移后

缩放（scale）

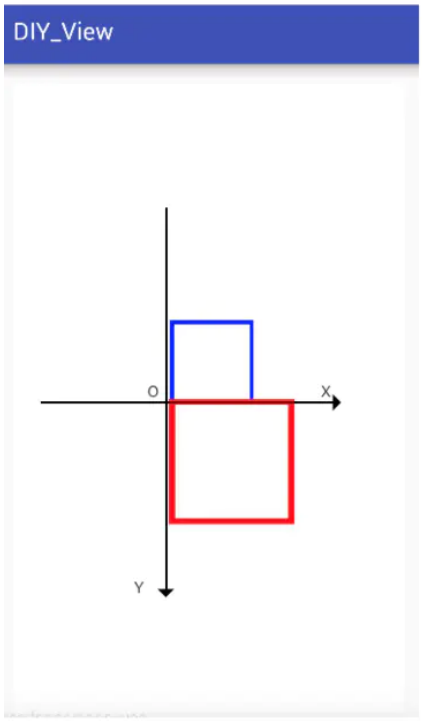
- 作用：放大/缩小 画布的倍数
- 具体使用：

```
// 共有两个方法
// 方法1
// 以(px,py)为中心，在x方向缩放sx倍，在y方向缩放sy倍
// 缩放中心默认为（0,0）
public final void scale(float sx, float sy)

// 方法2
// 比方法1多了两个参数（px,py），用于控制缩放中心位置
// 缩放中心为（px,py）
 public final void scale (float sx, float sy, float px, float py)
```

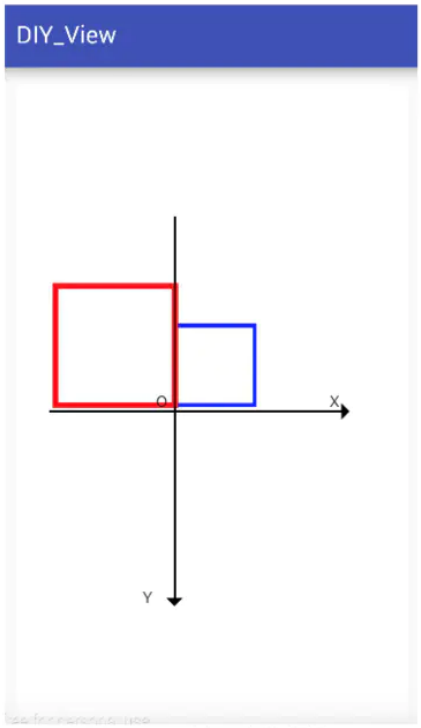
当缩放倍数为负数时，会先进行缩放，然后根据不同情况进行图形翻转
假设缩放倍数为（a,b），旋转中心为(px,py)

旋转方式	a>0	a<0
b>0	\	以px为轴翻转
b<0	以py为轴翻转	以旋转中心翻转

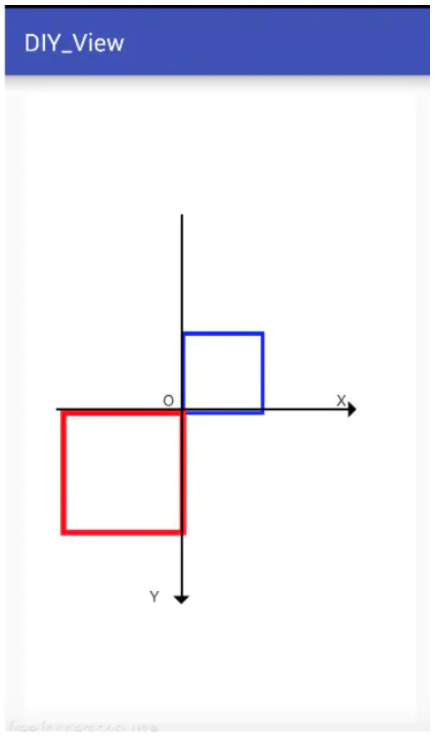
具体如下图：（缩放倍数为1.5，旋转中心为（0，0）为例）



a>0，b<0：以py为轴翻转



a<0，b>0：以px为轴翻转



a<0，b<0：以旋转中心翻转

旋转(rotate)

- 注意：角度增加方向为顺时针
- 具体使用：

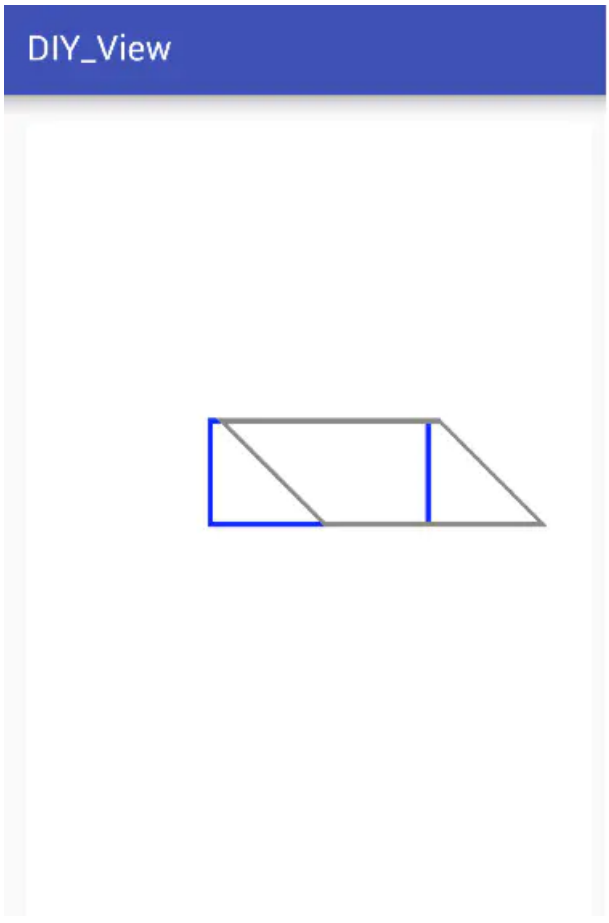
```
// 方法1
// 以原点(0,0)为中心旋转 degrees 度
public final void rotate(float degrees)
  // 以原点(0,0)为中心旋转 90 度
canvas.rotate(90);

// 方法2
// 以(px,py)点为中心旋转degrees度
public final void rotate(float degrees, float px, float py)
// 以(30,50)为中心旋转 90 度
canvas.rotate(90,30,50);
```

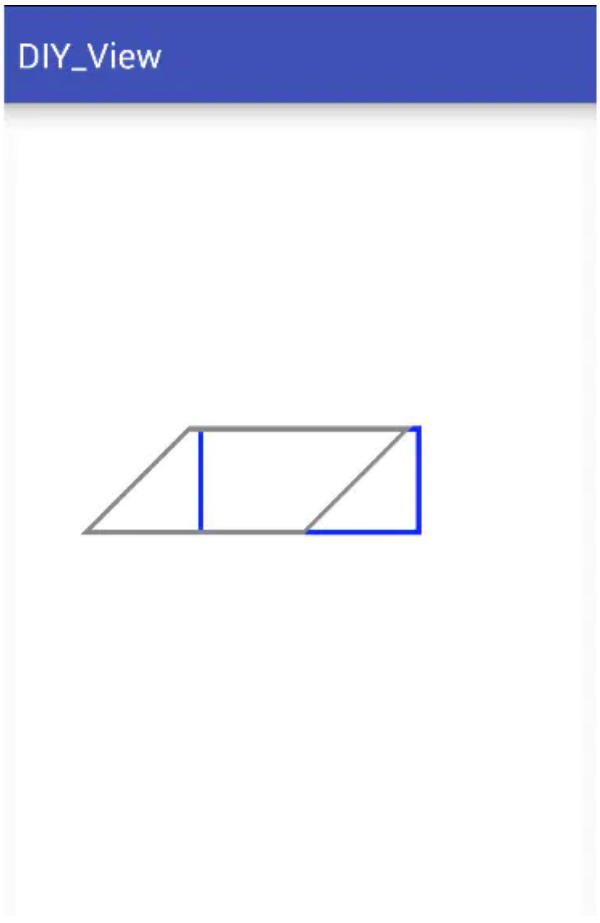
错切(skew)

- 作用：将画布在x方向倾斜a角度，在y方向倾斜b角度

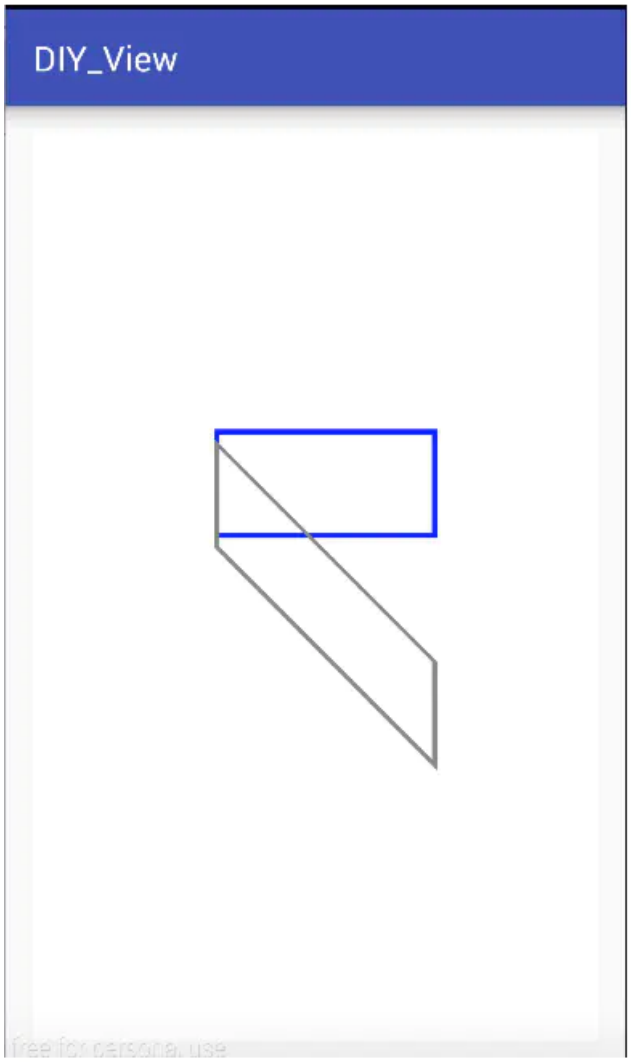
- 具体使用：

```
// 参数 sx = tan a ，sx>0时表示向X正方向倾斜（即向左）
// 参数 sy = tan b ，sy>0时表示向Y正方向倾斜（即向下）
public void skew(float sx, float sy)


// 实例
   // 为了方便观察,我将坐标系移到屏幕中央
        canvas.translate(300, 500);
        // 初始矩形
        canvas.drawRect(20, 20, 400, 200, mPaint2);

        // 向X正方向倾斜45度
        canvas.skew(1f, 0);
        canvas.drawRect(20, 20, 400, 200, mPaint1);

        //向X负方向倾斜45度
        canvas.skew(-1f, 0);
        canvas.drawRect(20, 20, 400, 200, mPaint1);

        // 向Y正方向倾斜45度
        canvas.skew(0, 1f);
        canvas.drawRect(20, 20, 400, 200, mPaint1);

       // 向Y负方向倾斜45度
        canvas.skew(0, -1f);
        canvas.drawRect(20, 20, 400, 200, mPaint1);
```

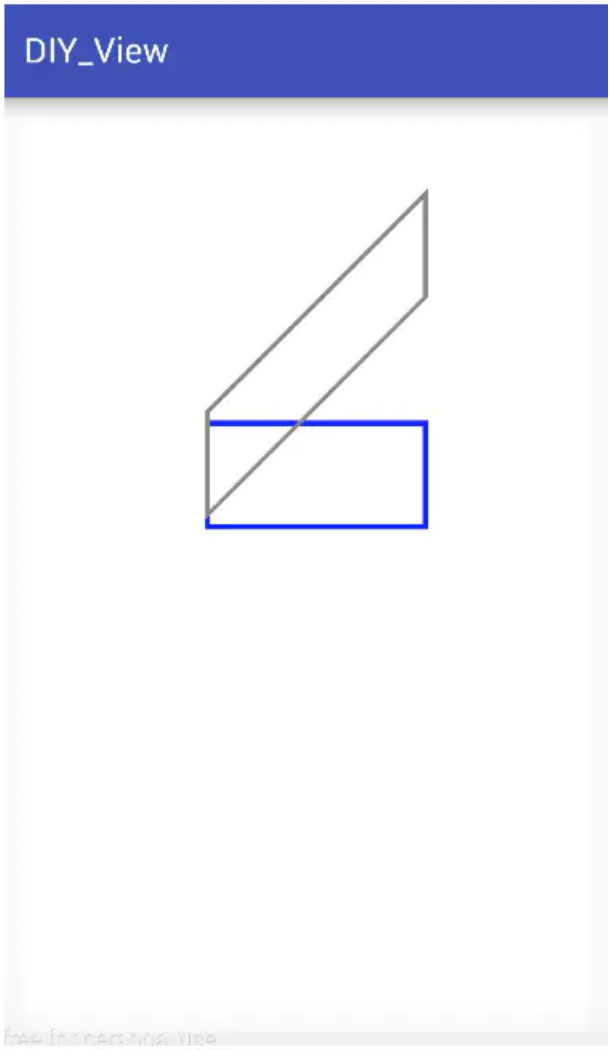




canvas.skew(1f, 0);



canvas.skew(-1f, 0);



canvas.skew(0, 1f);

canvas.skew(0, -1f);

2. **画布裁剪**

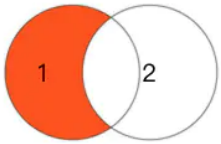
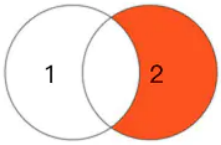
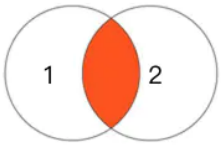
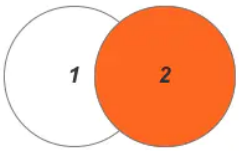
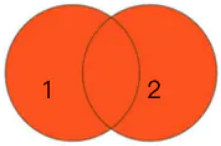
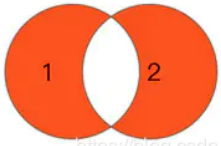
特别注意：其余的区域只是不能编辑，但是并没有消失。

- 具体使用

```
裁剪共分为：裁剪路径、裁剪矩形、裁剪区域

// 裁剪路径
// 方法1
public boolean clipPath(@NonNull Path path)
// 方法2
public boolean clipPath(@NonNull Path path, @NonNull Region.Op op)


// 裁剪矩形
```

```
// 方法1
public boolean clipRect(int left, int top, int right, int bottom)
// 方法2
public boolean clipRect(float left, float top, float right, float bottom)
// 方法3
public boolean clipRect(float left, float top, float right, float bottom,
            @NonNull Region.Op op)

// 裁剪区域
// 方法1
public boolean clipRegion(@NonNull Region region)
// 方法2
public boolean clipRegion(@NonNull Region region, @NonNull Region.Op op)
```

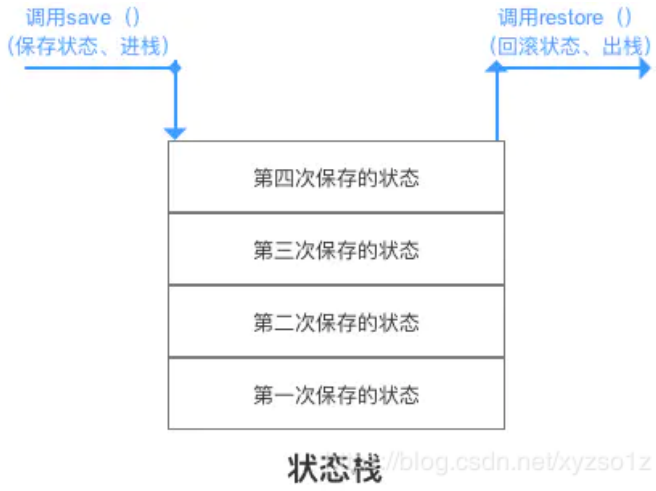
这里特别说明一下参数Region.Op op
作用：在剪下多个区域下来的情况，当这些区域有重叠的时候，这个参数决定重叠部分该如何处理，多次裁剪之后究竟获得了哪个区域，有以下几种参数：

参数	说明	示意图
DIFFERENCE	第一次裁剪不同于第二次裁剪的区域	
REVERSEDIFFERENCE	第二次裁剪不同于第一次裁剪的区域	
INTERSECT	（交集）第二次裁剪与第一次裁剪的重叠区域	
REPLACE	第二次裁剪的区域	
UNION	（并集）两次裁剪区域的和	
XOR	（异或）两次裁剪区域之和并减去重叠的部分	

3. **画布快照**
这里先理清几个概念

- 画布状态：当前画布经过的一系列操作

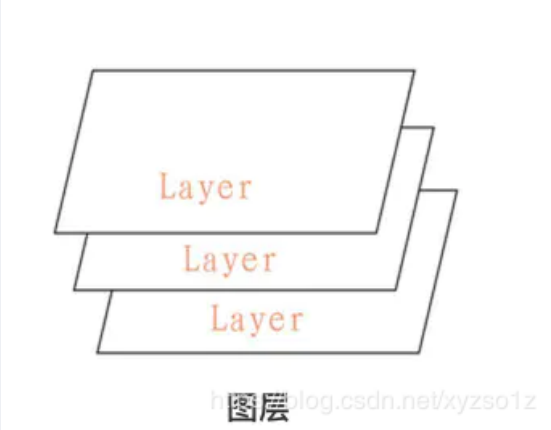
- 状态栈：存放画布状态和图层的栈（后进先出）



状态栈

- 画布的构成：有多个图层构成，如下图：

1. 在画布上操作=在图层上操作
2. 如无设置，绘制操作和画布操作是默认图层上进行
3. 在通常情况下，使用默认图层就可满足需求；若需要绘制复杂的内容(如地图)使用更多的图层
4. 最终显示的结果=所有图层叠在一起的效果



图层

保存当前画布状态(save)

- 作用：保存画布状态(即保存画布的一系列操作)
- 应用场景：画布的操作是不可逆的，而且会影响后续的步骤，假如需要回到之前画布进行下一次操作，就需要对画布的状态进行保存和回滚
- 具体使用：

```
1
2  // 方法1:
3    // 保存全部状态
4    public int save ()
5
6  // 方法2：
7    // 根据saveFlags参数保存一部分状态
8
```

```
    // 使用该参数可以只保存一部分状态，更加灵活
    public int save (int saveFlags)

// saveFlags参数说明：
// 1.ALL_SAVE_FLAG（默认）：保存全部状态
// 2. CLIP_SAVE_FLAG：保存剪辑区
// 3. CLIP_TO_LAYER_SAVE_FLAG：剪裁区作为图层保存
// 4. FULL_COLOR_LAYER_SAVE_FLAG：保存图层的全部色彩通道
// 5. HAS_ALPHA_LAYER_SAVE_FLAG：保存图层的alpha(不透明度)通道
// 6. MATRIX_SAVE_FLAG：保存Matrix信息(translate, rotate, scale, skew)

// 每调用一次save（）,都会在栈顶添加一条状态信息（入栈）
```

保存某个图层状态（saveLayer）

- 作用：新建一个图层，并放入特定的栈中
- 具体使用

> 使用起来非常复杂，因为图层之间叠加会导致计算量成倍增长，应尽量避免使用

```
// 无图层alpha(不透明度)通道
public int saveLayer (RectF bounds, Paint paint)
public int saveLayer (RectF bounds, Paint paint, int saveFlags)
public int saveLayer (float left, float top, float right, float bottom, Paint paint)
public int saveLayer (float left, float top, float right, float bottom, Paint paint, int saveFlags)

// 有图层alpha(不透明度)通道
public int saveLayerAlpha (RectF bounds, int alpha)
public int saveLayerAlpha (RectF bounds, int alpha, int saveFlags)
public int saveLayerAlpha (float left, float top, float right, float bottom, int alpha)
public int saveLayerAlpha (float left, float top, float right, float bottom, int alpha, int saveFlags)
```

回滚上一次保存的状态(restore)

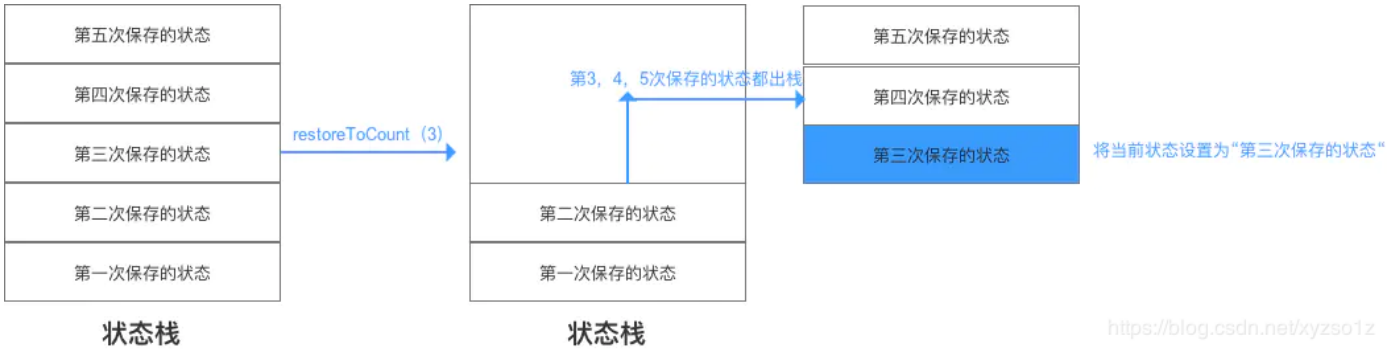
- 作用：恢复上一次保存的画布状态
- 具体使用

```
// 采取状态栈的形式。即从栈顶取出一个状态进行恢复。
canvas.restore();
```

回滚指定保存的状态（restoreToCount）

- 作用：恢复指定状态。将指定位置以及以上所有状态出栈
- 具体使用：

```
 canvas.restoreToCount(3) ;
// 弹出 3、4、5的状态，并恢复第3次保存的画布状态
```



获取保存的次数（getSaveCount）

- 作用：获取保存过图层的次数
- 具体使用：

```
canvas.getSaveCount();
// 以上面栈为例，则返回5
// 注：即使弹出所有的状态，返回值依旧为1，代表默认状态。（返回值最小为1）
```

xyzso1z
原创文章 80　获赞 40　访问量 2万+