# 基础

## 坐标系

1. 屏幕坐标系

动设备定义屏幕左上角为坐标原点，向右为x轴增大方向，向下为y轴增大方向。

View坐标系：

1. View的坐标系统是相对于父控件而言的

getTop(); //获取子View左上角距父View顶部的距离

getLeft(); //获取子View左上角距父View左侧的距离

getBottom(); //获取子View右下角距父View顶部的距离

getRight(); //获取子View右下角距父View左侧的距离



1. MotionEvent中 getX 和 getRawX的区别

getX:触摸点相对于其所在组件坐标系的坐标

getRawX:触摸点相对于屏幕默认坐标系的坐标



## 颜色

### 安卓支持的颜色模式

|  |  |  |
| --- | --- | --- |
| 颜色模式 | 说明 |  |
| ARGB8888 | 四通道高精度(32位) | ARGB表示4通道，8888表示用4\*8位二进制表示 |
| ARGB4444 | 四通道低精度(16位) | ARGB表示4通道，8888表示用4\*4位二进制表示 |
| RGB565 | 屏幕默认模式(16位) | RGB表示3通道，8888表示用5+6+5位二进制表示 |
| Alpha8 | 仅有透明通道(8位) | 用8位二进制数表示 |

字母表示通道类型，数值表示该类型用多少位二进制来描述，常用的是ARGB8888和ARGB4444，而在所有的安卓设备屏幕上默认的模式都是RGB565。

A 从0x00到0xff表示从透明到不透明。

RGB 从0x00到0xff表示颜色从浅到深。

当RGB全取最小值(0或0x000000)时颜色为黑色，全取最大值(255或0xffffff)时颜色为白色

### 颜色的使用

1. 使用已经定义好的颜色

setBackgroundColor(Color.*BLUE*);

1. 使用argb方法

setBackgroundColor(Color.*argb*(255, 0, 255, 0));

1. 在xml定义颜色，在java或xml文件引用

setBackgroundColor(getResources().getColor(R.color.*testColor*));

setBackgroundColor(ContextCompat.*getColor*(this, R.color.*testColor*));

android:background="#ff0000" //创建并使用颜色

### 颜色混合模式

颜色一般都是四个通道(ARGB)的，其中(RGB)控制的是颜色,而A(Alpha)控制的是透明度。

因为我们的显示屏是没法透明的，因此最终显示在屏幕上的颜色里可以认为没有Alpha通道。Alpha通道主要在两个图像混合的时候生效。

默认情况下，当一个颜色绘制到Canvas上时的混合模式是这样计算的：

最终颜色 = 绘制的颜色 + (1 - 绘制颜色的透明度) × Canvas上的原有颜色。

注意：

1.这里一般把每个通道的取值从0(ox00)到255(0xff)映射到0到1的浮点数表示。

2.这里等式右边的“绘制的颜色”、“Canvas上的原有颜色” 都是经过预乘了自己的Alpha通道的值。如绘制颜色：0x88ffffff，那么参与运算时的每个颜色通道的值不是1.0，而是(1.0 \* 0.5333 = 0.5333)。 (其中0.5333 = 0x88/0xff)

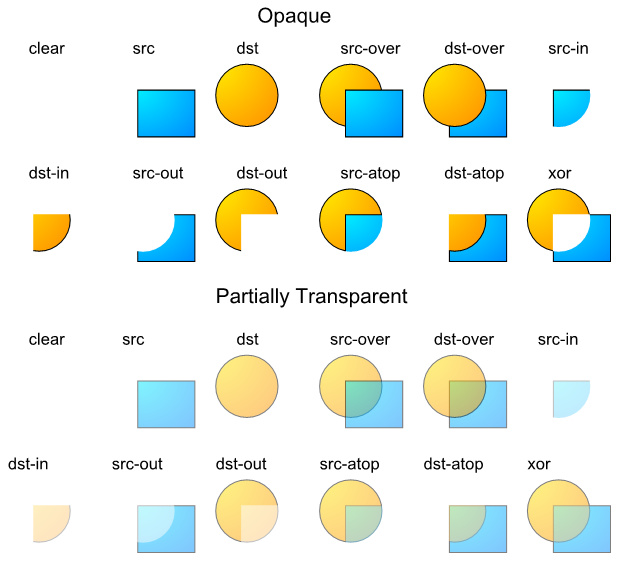
使用这种方式的混合，就会造成后绘制的内容以半透明的方式叠在上面的视觉效果。

其实还可以有不同的混合模式供我们选择，用Paint.setXfermode，指定不同的PorterDuff.Mode。（算法相关的东西不懂）

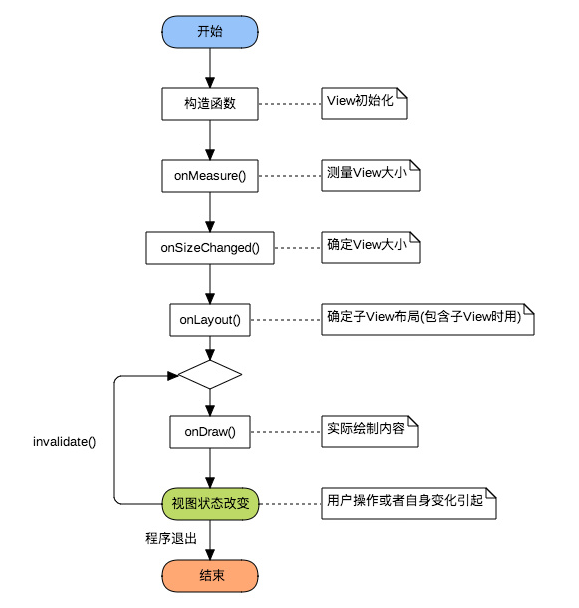
下表是各个PorterDuff模式的混合计算公式：（D指原本在Canvas上的内容dst，S指绘制输入的内容src，a指alpha通道，c指RGB各个通道）

|  |  |
| --- | --- |
| 混合模式 | 计算公式 |
| ADD | Saturate(S + D) |
| CLEAR | [0, 0] |
| DARKEN | [Sa + Da - SaDa, Sc(1 - Da) + Dc\*(1 - Sa) + min(Sc, Dc)] |
| DST | [Da, Dc] |
| DST\_ATOP | [Sa, Sa \* Dc + Sc \* (1 - Da)] |
| DST\_IN | [Sa \* Da, Sa \* Dc] |
| DST\_OUT | [Da \* (1 - Sa), Dc \* (1 - Sa)] |
| DST\_OVER | [Sa + (1 - Sa)Da, Rc = Dc + (1 - Da)Sc] |
| LIGHTEN | [Sa + Da - SaDa, Sc(1 - Da) + Dc\*(1 - Sa) + max(Sc, Dc)] |
| MULTIPLY | [Sa \* Da, Sc \* Dc] |
| SCREEN | [Sa + Da - Sa \* Da, Sc + Dc - Sc \* Dc] |
| SRC | [Sa, Sc] |
| SRC\_ATOP | [Da, Sc \* Da + (1 - Sa) \* Dc] |
| SRC\_IN | [Sa \* Da, Sc \* Da] |
| SRC\_OUT | [Sa \* (1 - Da), Sc \* (1 - Da)] |
| SRC\_OVER | [Sa + (1 - Sa)Da, Rc = Sc + (1 - Sa)Dc] |
| XOR | [Sa + Da - 2 \* Sa \* Da, Sc \* (1 - Da) + (1 - Sa) \* Dc] |

用示例图来查看使用不同模式时的混合效果如下（src表示输入的图，dst表示原Canvas上的内容）：



## 自定义View绘制流程函数调用链(简化版)：



## 自定义View的分类

1. 组合控件

组合控件，顾名思义就是将一些小的控件组合起来形成一个新的控件，这些小的控件多是系统自带的控件。

2. 自绘控件

自绘控件的内容都是自己绘制出来的，在View的onDraw方法中完成绘制。

3. 继承控件

就是继承已有的控件，创建新控件，保留继承的父控件的特性，并且还可以引入新特性。

## 自定义View的几个重要函数

### 构造函数

构造函数是View的入口，可以用于初始化一些的内容，和获取自定义属性。

View的构造函数有四种重载分别如下:

public void SloopView(Context context) {}

public void SloopView(Context context, AttributeSet attrs) {}  
public void SloopView(Context context, AttributeSet attrs, int defStyleAttr) {}  
public void SloopView(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {}

有四个参数的构造函数在API21的时候才添加上，暂不考虑。

有三个参数的构造函数中第三个参数是默认的Style，这里的默认的Style是指它在当前Application或Activity所用的Theme中的默认Style，且只有在明确调用的时候才会生效。即使你在View中使用了Style这个属性也不会调用三个参数的构造函数，所调用的依旧是两个参数的构造函数。

由于三个参数的构造函数第三个参数一般不用，暂不考虑。

排除了两个之后，只剩下一个参数和两个参数的构造函数，他们的详情如下：

//一般在直接New一个View的时候调用。

public void SloopView(Context context) {}  
//一般在layout文件中使用的时候会调用，关于它的所有属性(包括自定义属性)都会包含在attrs中传递进来。  
public void SloopView(Context context, AttributeSet attrs) {}

以下方法调用的是一个参数的构造函数：

//在Avtivity中

SloopView view new SloopView(this);

以下方法调用的是两个参数的构造函数：

//在layout文件中 - 格式为： 包名.View名

<com.sloop.study.SloopView  
 android:layout\_width"wrap\_content"  
 android:layout\_height"wrap\_content"/>

### 测量View大小(onMeasure)

Q: 为什么要测量View大小？

A: View的大小不仅由自身所决定，同时也会受到父控件的影响，为了我们的控件能更好的适应各种情况，一般会自己进行测量。

测量View大小使用的是onMeasure函数，我们可以从onMeasure的两个参数中取出宽高的相关数据：

@Override

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
 int widthsize MeasureSpec.*getSize*(widthMeasureSpec); //取出宽度的确切数值  
 int widthmode MeasureSpec.*getMode*(widthMeasureSpec); //取出宽度的测量模式  
  
 int heightsize MeasureSpec.*getSize*(heightMeasureSpec); //取出高度的确切数值  
 int heightmode MeasureSpec.*getMode*(heightMeasureSpec); //取出高度的测量模式  
}

从上面可以看出 onMeasure 函数中有 widthMeasureSpec 和 heightMeasureSpec 这两个 int 类型的参数， 毫无疑问他们是和宽高相关的， 但它们其实不是宽和高， 而是由宽、高和各自方向上对应的测量模式来合成的一个值。

测量模式一共有三种， 被定义在 Android 中的 View 类的一个内部类View.MeasureSpec中：

|  |  |  |
| --- | --- | --- |
| 模式 | 二进制数值 | 描述 |
| UNSPECIFIED | 00 | 默认值，父控件没有给子view任何限制，子View可以设置为任意大小。 |
| EXACTLY | 01 | 表示父控件已经确切的指定了子View的大小。 |
| AT\_MOST | 10 | 表示子View具体大小没有尺寸限制，但是存在上限，上限一般为父View大小。 |

在int类型的32位二进制位中，31-30这两位表示测量模式,29~0这三十位表示宽和高的实际值，实际上如下：

以数值1080(二进制为: 1111011000)为例(其中模式和实际数值是连在一起的，为了展示我将他们分开了)：

|  |  |  |
| --- | --- | --- |
| 模式名称 | 模式数值 | 实际数值 |
| UNSPECIFIED | 00 | 000000000000000000001111011000 |
| EXACTLY | 01 | 000000000000000000001111011000 |
| AT\_MOST | 10 | 000000000000000000001111011000 |

PS: 实际上关于上面的东西了解即可，在实际运用之中只需要记住有三种模式，用 MeasureSpec 的 getSize是获取数值， getMode是获取模式即可。

注意：

如果对View的宽高进行修改了，不要调用

super.onMeasure( widthMeasureSpec, heightMeasureSpec); 要调用

setMeasuredDimension( widthsize, heightsize); 这个函数。

### 确定View大小(onSizeChanged)

这个函数在视图大小发生改变时调用。

Q: 在测量完View并使用setMeasuredDimension函数之后View的大小基本上已经确定了，那么为什么还要再次确定View的大小呢？

A: 这是因为View的大小不仅由View本身控制，而且受父控件的影响，所以我们在确定View大小的时候最好使用系统提供的onSizeChanged回调函数。

onSizeChanged如下：

@Override

protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
 super.onSizeChanged(w, h, oldw, oldh);  
}

可以看出，它又四个参数，分别为 宽度，高度，上一次宽度，上一次高度。

这个函数比较简单，我们只需关注宽度(w), 高度(h)即可，这两个参数就是View最终的大小。

### 确定子View布局位置(onLayout)

确定布局的函数是onLayout，它用于确定子View的位置，在自定义ViewGroup中会用到，他调用的是子View的layout函数。

在自定义ViewGroup中，onLayout一般是循环取出子View，然后经过计算得出各个子View位置的坐标值，然后用以下函数设置子View位置。

child.layout(l, t, r, b);

四个参数分别为：

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 对应的函数 |
| l | View左侧距父View左侧的距离 | getLeft(); |
| t | View顶部距父View顶部的距离 | getTop(); |
| r | View右侧距父View左侧的距离 | getRight(); |
| b | View底部距父View顶部的距离 | getBottom(); |



### 5.绘制内容(onDraw)

onDraw是实际绘制的部分，也就是我们真正关心的部分，使用的是Canvas绘图。

@Override

protected void onDraw(Canvas canvas) {  
 super.onDraw(canvas);  
}

### 6.对外提供操作方法和监听回调

自定义完View之后，一般会对外暴露一些接口，用于控制View的状态等，或者监听View的变化.

# Canvas

Canvas操作方法速查表：

|  |  |
| --- | --- |
| 操作类型 | 相关API |
| 绘制颜色 | drawColor, drawRGB, drawARGB: 使用单一颜色填充整个画布 |
| 绘制基本形状 | drawPoint，drawPoints: 绘制点  drawLine，drawLines:绘制线  drawRect: 绘制矩形  drawRoundRect: 绘制圆角矩形  drawOval: 绘制椭圆  drawCircle: 绘制圆  drawArc: 绘制圆弧 |
| 绘制图片 | drawBitmap: 绘制位图  drawPicture: 绘制图片 |
| 绘制文本 | drawText: 绘制文字  drawPosText: 绘制文字时指定每个文字位置  drawTextOnPath: 根据路径绘制文字 |
| 绘制路径 | drawPath: 绘制路径，绘制贝塞尔曲线时也需要用到该函数 |
| 顶点操作 | 通过对顶点操作可以使图像形变  drawVertices: 直接对画布作用  drawBitmapMesh: 只对绘制的Bitmap作用 |
| 画布剪裁 | 设置画布的显示区域：  clipPath  clipRect |
| 画布快照 | save: 保存当前状态  restore: 回滚到上一次保存的状态  saveLayerXxx: 保存图层状态  restoreToCount: 回滚到指定状态  getSaveCount: 获取保存次数 |
| 画布变换 | translate: 位移  scale: 缩放  rotate:旋转  skew: 错切 |
| Matrix(矩阵) | 矩阵Matrix操作  getMatrix:  setMatrix:  concat: |

## Canvas简介

Canva称之为画布，能够在上面绘制各种东西，是安卓平台2D图形绘制的基础，非常强大。

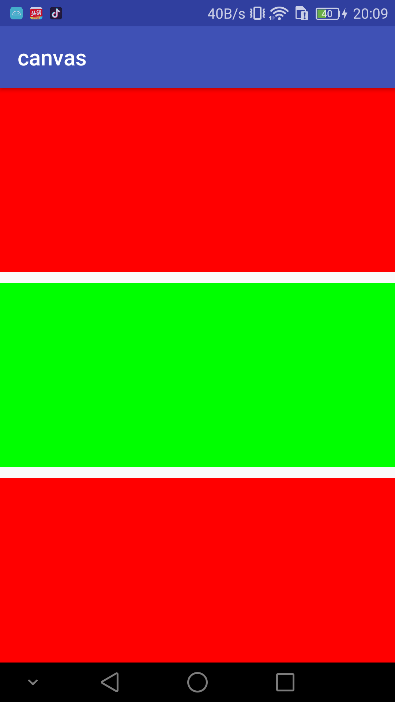
### 绘制颜色

绘制颜色是填充整个画布，常用于绘制底色。

canvas.drawColor(Color.*argb*(255, 255, 0, 0));

canvas.drawRGB(255, 0, 255);  
canvas.drawARGB(100, 0, 0, 255);

drawARGB第一个参数是透明度，取值0-255，注意控件不要设置background相关的属性，否则会造成叠加。



### 创建画笔

要想绘制内容，首先需要先创建一个画笔，画笔创建完成之后就可以在canvas中绘制各种内容了，创建画笔过程如下：

// 1.创建一个画笔

private Paint mPaint = new Paint();  
  
// 2.初始化画笔  
private void initPaint() {  
 mPaint.setColor(Color.*BLACK*); //设置画笔颜色  
 mPaint.setStyle(Paint.Style.*FILL*); //设置画笔模式为填充  
 mPaint.setStrokeWidth(10f); //设置画笔宽度为10px  
}  
// 3.在构造函数中初始化  
public TestView(Context context, @Nullable AttributeSet attrs) {  
 super(context, attrs);  
 initPaint();  
}

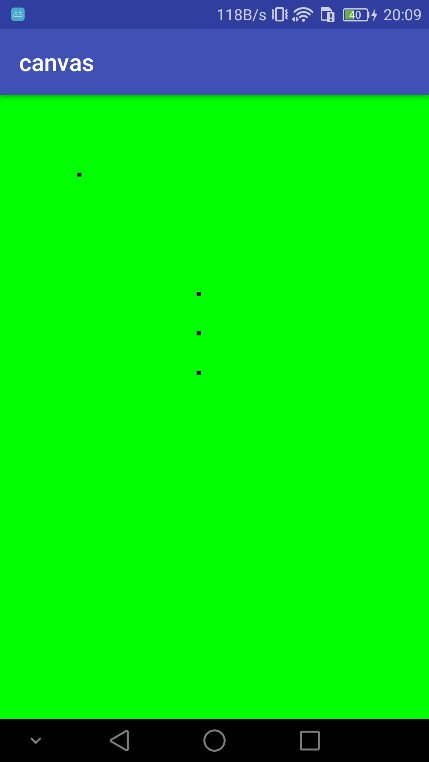
### 绘制点

可以绘制一个点，也可以绘制一组点

canvas.drawPoint(200, 200, mPaint); //在坐标(200,200)位置绘制一个点

canvas.drawPoints(new float[]{ //绘制一组点，坐标位置由float数组指定  
 500,500,  
 500,600,  
 500,700  
},mPaint);

坐标原点默认在左上角，水平向右为x轴增大方向，竖直向下为y轴增大方向。

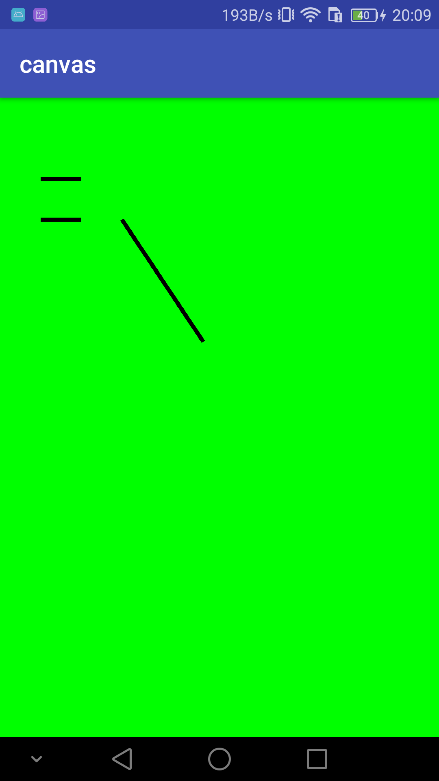


### 绘制直线

绘制直线需要两个点，初始点和结束点，同样绘制直线也可以绘制一条或者绘制一组：

canvas.drawLine(300,300,500,600,mPaint); // 在坐标(300,300)(500,600)之间绘制一条线

canvas.drawLines(new float[]{ // 绘制一组线 每四数字(两个点的坐标)确定一条线  
 100,200,200,200,  
 100,300,200,300  
},mPaint);

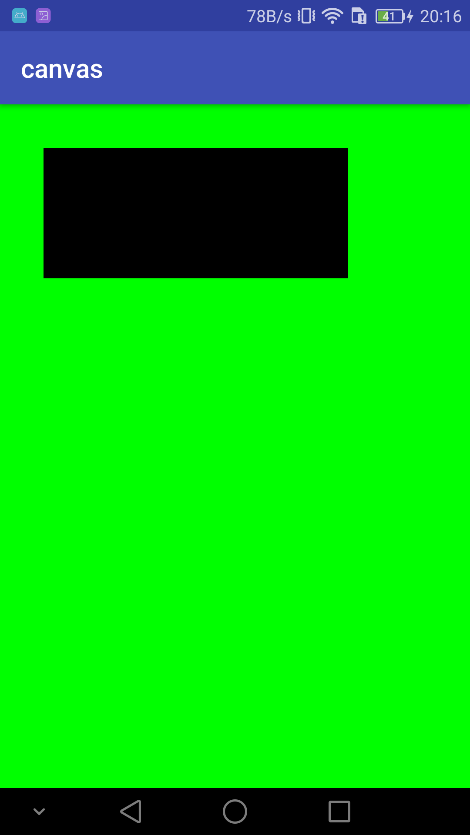


### 绘制矩形

确定确定一个矩形最少需要四个数据，就是对角线的两个点的坐标值，这里一般采用左上角和右下角的两个点的坐标。Canvas提供了三种重载方法，第一种就是提供四个数值(矩形左上角和右下角两个点的坐标)来确定一个矩形进行绘制。 其余两种是先将矩形封装为Rect或RectF(实际上仍然是用两个坐标点来确定的矩形),然后传递给Canvas绘制，如下：

// 第一种

canvas.drawRect(100,100,800,400,mPaint);  
  
// 第二种  
Rect rect = new Rect(100,100,800,400);  
canvas.drawRect(rect,mPaint);  
  
// 第三种  
RectF rectF = new RectF(100,100,800,400);  
canvas.drawRect(rectF,mPaint);



### 绘制圆角矩形

绘制圆角矩形也提供了两种重载方式：

// 第一种

RectF rectF = new RectF(100,100,800,400);  
canvas.drawRoundRect(rectF,30,30,mPaint);  
  
// 第二种  
canvas.drawRoundRect(100,100,800,400,30,30,mPaint);

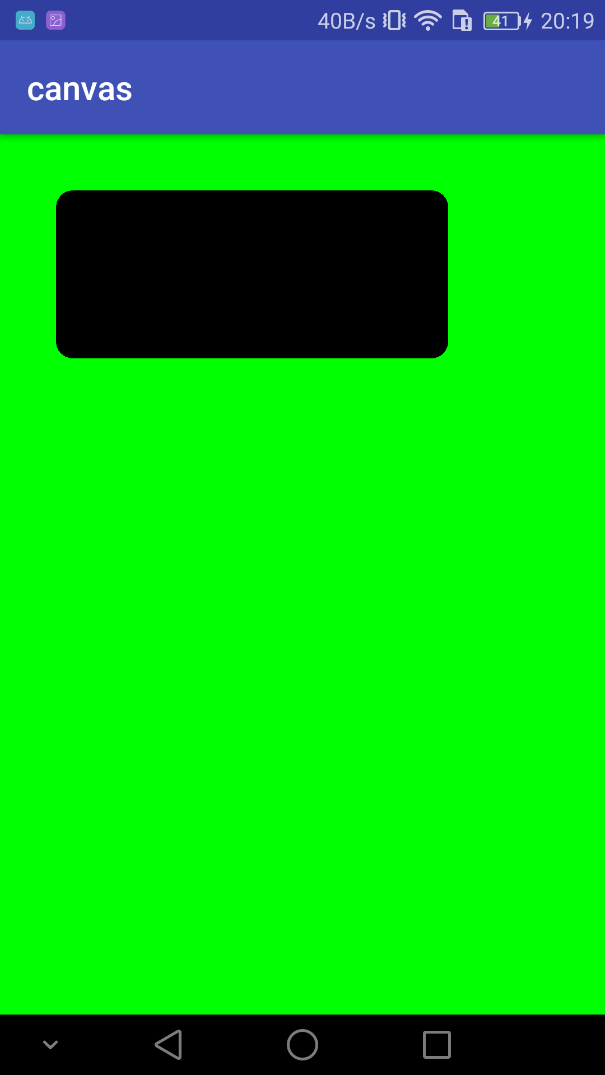
上面两种方法绘制效果也是一样的，但鉴于第二种方法在API21的时候才添加上，所以我们一般使用的都是第一种

public void drawRoundRect(@NonNull RectF rect, float rx, float ry, @NonNull Paint

paint){}

上面的rx和ry对应的是椭圆圆弧的半径。

当rx为宽度的一半，ry为高度的一半时，刚好是一个椭圆，而当rx大于宽度的一半，ry大于高度的一半时，实际上是无法计算出圆弧的，所以drawRoundRect对大于该数值的参数进行了限制(修正)，凡是大于一半的参数均按照一半来处理。



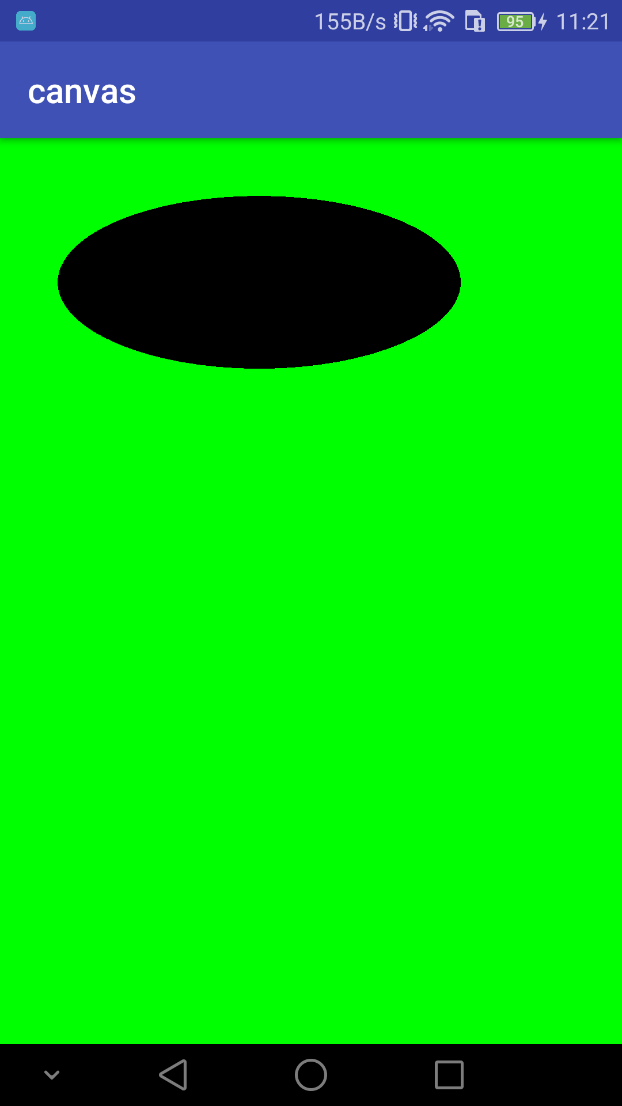
### 绘制椭圆

绘制椭圆只需要提供椭圆外切矩形的2个对角线坐标即可。

// 第一种

RectF rectF = new RectF(100,100,800,400);  
canvas.drawOval(rectF,mPaint);  
  
// 第二种  
canvas.drawOval(100,100,800,400,mPaint);

2种效果一样，第二种是api21才提供的方法，一般使用第一种。

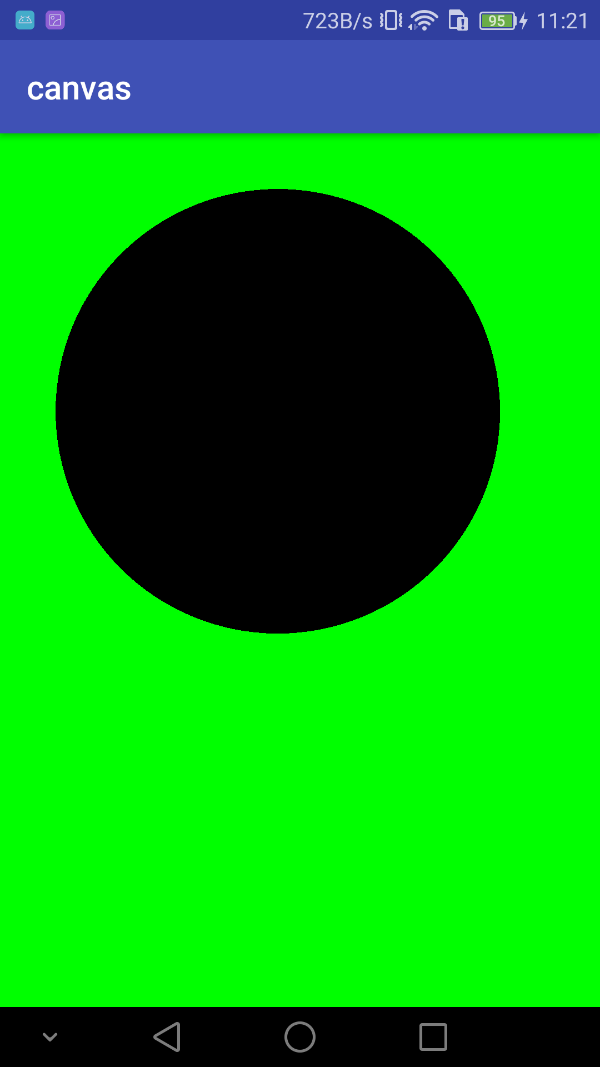


绘制椭圆实际上就是绘制一个矩形的内切图形，如果传递进来的是一个长宽相等的矩形(即正方形)，那么绘制出来的实际上就是一个圆。

### 绘制圆

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

绘制圆形有四个参数，前两个是圆心坐标，第三个是半径，最后一个是画笔。



### 绘制圆弧

绘制圆弧的方法

// 第一种

public void drawArc(@NonNull RectF oval, float startAngle, float sweepAngle, boolean useCenter, @NonNull Paint paint){}  
  
// 第二种  
public void drawArc(float left, float top, float right, float bottom, float startAngle,  
 float sweepAngle, boolean useCenter, @NonNull Paint paint) {}

相比于绘制椭圆，绘制圆弧还多了三个参数

startAngle // 开始角度

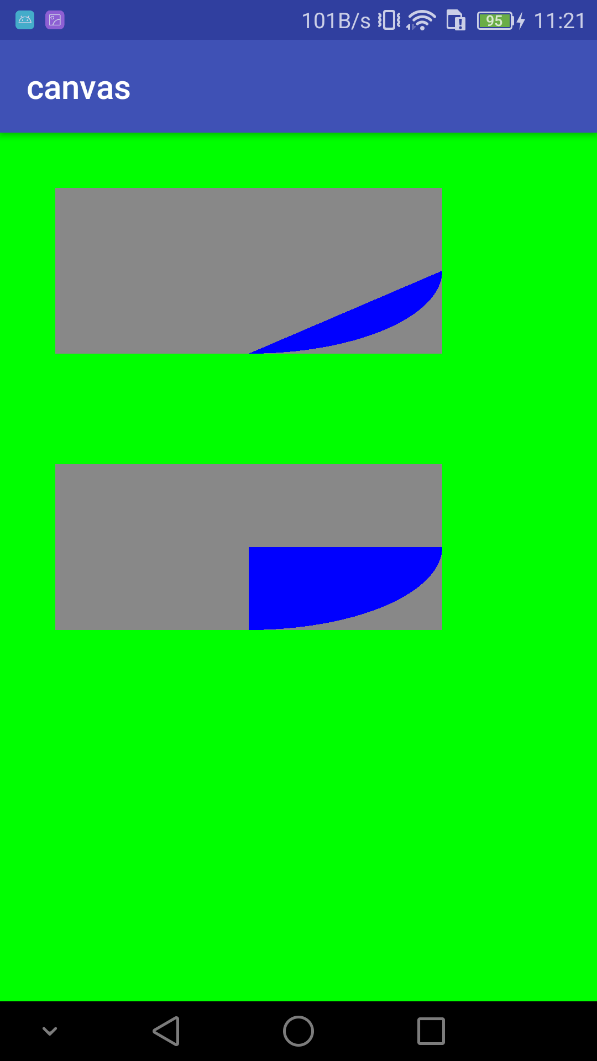
sweepAngle // 扫过角度,正数表示顺时针方向

useCenter // 是否使用中心, 使用了中心点之后绘制出来类似于一个扇形，而不使用中心点则是圆弧起始点和结束点之间的连线加上圆弧围成的图形

测试代码如下：

RectF rectF = new RectF(100,100,800,400);

// 绘制背景矩形  
mPaint.setColor(Color.*GRAY*);  
canvas.drawRect(rectF,mPaint);  
  
// 绘制圆弧  
mPaint.setColor(Color.*BLUE*);  
canvas.drawArc(rectF,0,180,false,mPaint);  
  
//-------------------------------------  
RectF rectF2 = new RectF(100,600,800,900);  
// 绘制背景矩形  
mPaint.setColor(Color.*GRAY*);  
canvas.drawRect(rectF2,mPaint);  
  
// 绘制圆弧  
mPaint.setColor(Color.*BLUE*);  
canvas.drawArc(rectF2,0,180,true,mPaint);



## Paint

绘制的基本形状由Canvas确定，但绘制出来的颜色,具体效果则由Paint确定

画笔有三种模式，由paint的setStyle方法进行设置，这三种模式如下：

STROKE //描边

FILL //填充  
FILL\_AND\_STROKE //描边加填充

设置方式：

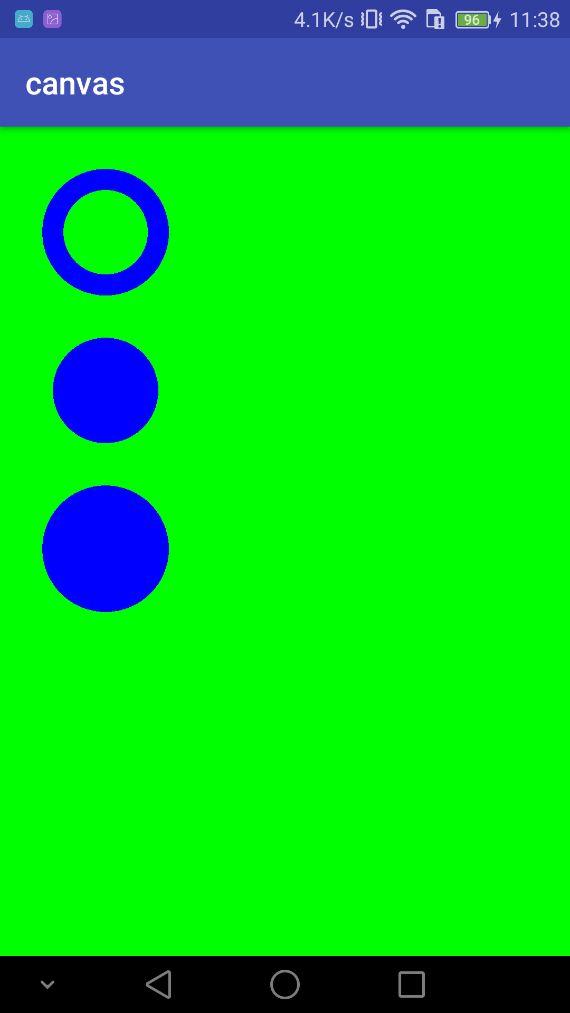
mPaint.setStyle(Paint.Style.*FILL*); //设置画笔模式为填充

测试代码：

Paint mPaint = new Paint();

mPaint.setColor(Color.*BLUE*);  
mPaint.setStrokeWidth(40); //为了实验效果明显，特地设置描边宽度非常大

// 描边  
mPaint.setStyle(Paint.Style.*STROKE*);  
canvas.drawCircle(200,200,100,mPaint);  
  
// 填充  
mPaint.setStyle(Paint.Style.*FILL*);  
canvas.drawCircle(200,500,100,mPaint);  
  
// 描边加填充  
mPaint.setStyle(Paint.Style.*FILL\_AND\_STROKE*);  
canvas.drawCircle(200, 800, 100, mPaint);



## 画布操作

合理的使用画布操作可以帮助你用更容易理解的方式创作你想要的效果，这也是画布操作存在的原因。 所有的画布操作都只影响后续的绘制，对之前已经绘制过的内容没有影响。

### 位移(translate)

translate是坐标系的移动，可以为图形绘制选择一个合适的坐标系。位移是基于当前位置移动，而不是每次基于屏幕左上角的(0,0)点移动，多次位置的效果是可以叠加的，代码如下：

// 在坐标原点绘制一个黑色圆形

mPaint.setColor(Color.*BLACK*);  
canvas.translate(200,000);  
canvas.drawCircle(0,0,100,mPaint);  
  
// 在坐标原点绘制一个蓝色圆形  
mPaint.setColor(Color.*BLUE*);  
canvas.translate(200,200);  
canvas.drawCircle(0,0,100,mPaint);

### 缩放(scale)

缩放提供了两个方法：

public void scale (float sx, float sy)

public final void scale (float sx, float sy, float px, float py)

这两个方法中前两个参数是相同的分别为x轴和y轴的缩放比例。而第二种方法比前一种多了两个参数，用来控制缩放中心位置的。

缩放比例(sx,sy)取值范围详解：

|  |  |
| --- | --- |
| 取值范围(n) | 说明 |
| [-∞, -1) | 先根据缩放中心放大n倍，再根据中心轴进行翻转 |
| -1 | 根据缩放中心轴进行翻转 |
| (-1, 0) | 先根据缩放中心缩小到n，再根据中心轴进行翻转 |
| 0 | 不会显示，若sx为0，则宽度为0，不会显示，sy同理 |
| (0, 1) | 根据缩放中心缩小到n |
| 1 | 没有变化 |
| (1, +∞) | 根据缩放中心放大n倍 |

测试代码如下：

// 将坐标系原点移动到画布正中心

canvas.translate(getWidth() / 2, getHeight() / 2);  
RectF rect = new RectF(0,-400,400,0); // 矩形区域  
mPaint.setColor(Color.*BLACK*); // 绘制黑色矩形  
canvas.drawRect(rect,mPaint);  
  
// canvas.scale(0.5f, 0f); // 画布缩放  
canvas.scale(0.5f,0.5f,200,0); // 画布缩放 <-- 缩放中心向右偏移了200个单位  
mPaint.setColor(Color.*BLUE*); // 绘制蓝色矩形  
canvas.drawRect(rect,mPaint);

和位移(translate)一样，缩放也是可以叠加的。

canvas.scale(0.5f,0.5f);

canvas.scale(0.5f,0.1f);

调用两次缩放则 x轴实际缩放为0.5x0.5=0.25 y轴实际缩放为0.5x0.1=0.05

#### 旋转(rotate)

旋转提供了两种方法：

和缩放一样，第二种方法多出来的两个参数依旧是控制旋转中心点的。

public void rotate (float degrees)

public final void rotate (float degrees, float px, float py)

默认的旋转中心依旧是坐标原点：

// 将坐标系原点移动到画布正中心

canvas.translate(getWidth() / 2, getHeight() / 2);  
RectF rect = new RectF(0,-400,400,0); // 矩形区域  
mPaint.setColor(Color.*BLACK*); // 绘制黑色矩形  
canvas.drawRect(rect,mPaint);  
  
canvas.rotate(180); // 旋转180度 <-- 默认旋转中心为原点  
mPaint.setColor(Color.*BLUE*); // 绘制蓝色矩形  
canvas.drawRect(rect,mPaint);

旋转也是可叠加的，叠加的效果等于多次旋转角度的和

#### 错切(skew)

skew这里翻译为错切，错切是特殊类型的线性变换。

错切只提供了一种方法：

public void skew (float sx, float sy)

参数含义：

float sx:将画布在x方向上倾斜相应的角度，sx倾斜角度的tan值，

float sy:将画布在y轴方向上倾斜相应的角度，sy为倾斜角度的tan值.

变换后:

X = x + sx \* y

Y = sy \* x + y

测试代码如下：

// 将坐标系原点移动到画布正中心

canvas.translate(getWidth() / 2, getHeight() / 2);  
RectF rect = new RectF(0,0,200,200); // 矩形区域  
mPaint.setColor(Color.*BLACK*); // 绘制黑色矩形  
canvas.drawRect(rect,mPaint);  
  
canvas.skew(1, -1); // 水平错切 <- 45度  
mPaint.setColor(Color.*BLUE*); // 绘制蓝色矩形  
canvas.drawRect(rect,mPaint);

错切也是可叠加的，不过调用次序不同绘制结果也会不同。

#### 快照(save)和回滚(restore)

画布的操作是不可逆的，而且很多画布操作会影响后续的步骤，例如第一个例子，两个圆形都是在坐标原点绘制的，而因为坐标系的移动绘制出来的实际位置不同。所以会对画布的一些状态进行保存和回滚。

与之相关的API:

|  |  |
| --- | --- |
| 相关API | 简介 |
| save | 把当前的画布的状态进行保存，然后放入特定的栈中 |
| saveLayerXxx | 新建一个图层，并放入特定的栈中 |
| restore | 把栈中最顶层的画布状态取出来，并按照这个状态恢复当前的画布 |
| restoreToCount | 弹出指定位置及其以上所有的状态，并按照指定位置的状态进行恢复 |
| getSaveCount | 获取栈中内容的数量(即保存次数) |

画布和图层

画布是由多个图层构成的,之前的绘制操作和画布操作都是在默认图层上进行的。可以把这些图层看做是一层一层的玻璃板，你在每层的玻璃板上绘制内容，然后把这些玻璃板叠在一起看就是最终效果。

SaveFlags

|  |  |
| --- | --- |
| 名称 | 简介 |
| ALL\_SAVE\_FLAG | 默认，保存全部状态 |
| CLIP\_SAVE\_FLAG | 保存剪辑区 |
| CLIP\_TO\_LAYER\_SAVE\_FLAG | 剪裁区作为图层保存 |
| FULL\_COLOR\_LAYER\_SAVE\_FLAG | 保存图层的全部色彩通道 |
| HAS\_ALPHA\_LAYER\_SAVE\_FLAG | 保存图层的alpha(不透明度)通道 |
| MATRIX\_SAVE\_FLAG | 保存Matrix信息( translate, rotate, scale, skew) |

save

save 有两种方法

// 保存全部状态

public int save ()  
// 根据saveFlags参数保存一部分状态  
public int save (int saveFlags)

可以看到第二种方法比第一种多了一个saveFlags参数，使用这个参数可以只保存一部分状态，更加灵活，这个saveFlags参数具体可参考上面表格中的内容。

每调用一次save方法，都会在栈顶添加一条状态信息，以上面状态栈图片为例，再调用一次save则会在第5次上面载添加一条状态。

saveLayerXxx

saveLayerXxx有比较多的方法：

// 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)

saveLayerXxx方法会让你花费更多的时间去渲染图像(图层多了相互之间叠加会导致计算量成倍增长)，如果可能，尽量避免使用。使用saveLayerXxx方法，也会将图层状态也放入状态栈中，同样使用restore方法进行恢复。

restore

状态回滚，就是从栈顶取出一个状态然后根据内容进行恢复。

restoreToCount

弹出指定位置以及以上所有状态，并根据指定位置状态进行恢复。

getSaveCount

获取保存的次数，即状态栈中保存状态的数量。该函数的最小返回值为1，即使弹出了所有的状态，返回值依旧为1，代表默认状态。

常用格式

save(); //保存状态

... //具体操作  
restore(); //回滚到之前的状态

### 图片文字

绘制有两种方法，drawPicture(矢量图) 和 drawBitmap(位图)。

#### drawPicture

使用Picture前请关闭硬件加速，以免引起不必要的问题！

在AndroidMenifest文件中application节点下添上

android:hardwareAccelerated=”false”以关闭整个应用的硬件加速。

Picture和上文中的录像功能是类似的，只不过Picture录的是Canvas中绘制的内容。我们把Canvas绘制点，线，矩形等诸多操作用Picture录制下来，下次需要的时候拿来就能用，使用Picture相比于再次调用绘图API，开销是比较小的，也就是说对于重复的操作可以更加省时省力。可以把Picture看作是一个录制Canvas操作的录像机。

Picture的相关方法：

|  |  |
| --- | --- |
| 相关方法 | 简介 |
| public int getWidth () | 获取宽度 |
| public int getHeight () | 获取高度 |
| public Canvas beginRecording (int width, int height) | 开始录制 (返回一个Canvas，在Canvas中所有的绘制都会存储在Picture中) |
| public void endRecording () | 结束录制 |
| public void draw (Canvas canvas) | 将Picture中内容绘制到Canvas中 |
| public static Picture createFromStream (InputStream stream) | (已废弃)通过输入流创建一个Picture |
| public void writeToStream (OutputStream stream) | (已废弃)将Picture中内容写出到输出流中 |

beginRecording 和 endRecording 是成对使用的，一个开始录制，一个是结束录制，两者之间的操作将会存储在Picture中。录制的内容是不会直接显示在屏幕上的，只是存储起来了,需要显示的时候调用draw方法。

例子：

// 1.创建Picture

private Picture mPicture = new Picture();  
  
// 2.录制内容方法  
private void recording() {  
 // 开始录制 (接收返回值Canvas)  
 Canvas canvas = mPicture.beginRecording(500, 500);  
 // 创建一个画笔  
 Paint paint = new Paint();  
 paint.setColor(Color.*BLUE*);  
 paint.setStyle(Paint.Style.*FILL*);  
 // 在Canvas中具体操作  
 // 位移  
 canvas.translate(250,250);  
 // 绘制一个圆  
 canvas.drawCircle(0,0,100,paint);  
 mPicture.endRecording();  
}  
// 3.在构造函数中录制

public TestView(Context context, @Nullable AttributeSet attrs) {  
 super(context, attrs);  
 recording(); // 调用录制  
}

// 4.绘制picture

mPicture.draw(canvas);

将Picture中的内容绘制出来可以有以下几种方法:

1. 使用Picture提供的draw方法绘制，可能会影响Canvas状态，一般不会使用。
2. 使用Canvas提供的drawPicture方法绘制，不影响Canvas，控制输出结果灵活。
3. 将Picture包装成为PictureDrawable，使用PictureDrawable的draw方法绘制，不影响Canvas，控制输出结果灵活。

以上3中方式的使用方法：

1. 使用Picture提供的draw方法绘制

// 将Picture中的内容绘制在Canvas上

mPicture.draw(canvas);

1. 使用Canvas提供的drawPicture方法绘制

drawPicture有三种方法：

public void drawPicture (Picture picture)

public void drawPicture (Picture picture, Rect dst)  
public void drawPicture (Picture picture, RectF dst)

Canvas的drawPicture不会影响Canvas状态

例子：

canvas.drawPicture(mPicture,new RectF(0, 0, mPicture.getWidth(), 200));

1. 将Picture包装成为PictureDrawable，使用PictureDrawable的draw方法绘制。

// 包装成为Drawable

PictureDrawable drawable = new PictureDrawable(mPicture);  
// 设置绘制区域 -- 注意此处所绘制的实际内容不会缩放  
drawable.setBounds(0,0,250,mPicture.getHeight());  
// 绘制  
drawable.draw(canvas);

setBounds是设置在画布上的绘制区域，并非根据该区域进行缩放，也不是剪裁Picture，每次都从Picture的左上角开始绘制。

#### drawBitmap

获取Bitmap方式:

|  |  |
| --- | --- |
| 获取方式 | 备注 |
| 通过Bitmap创建 | 复制一个已有的Bitmap(新Bitmap状态和原有的一致) 或者 创建一个空白的Bitmap(内容可改变) |
| 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内容不可变的Bitmap |
| 通过BitmapFactory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内容不可变的Bitmap |

BitmapFactory从不同位置获取Bitmap的方式:

1. 资源文件(drawable/mipmap/raw)

Bitmap bitmap = BitmapFactory.*decodeResource*(mContext.getResources(),R.raw.bitmap);

1. 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();  
}

1. 内存卡文件

Bitmap bitmap = BitmapFactory.*decodeFile*("/sdcard/bitmap.png");

1. 网络文件

// 此处省略了获取网络输入流的代码

Bitmap bitmap = BitmapFactory.*decodeStream*(is);  
is.close();

1. 绘制Bitmap

drawBitmap的常用方法：

// 第一种

public void drawBitmap (Bitmap bitmap, Matrix matrix, Paint paint)  
  
// 第二种  
public void drawBitmap (Bitmap bitmap, float left, float top, Paint paint)  
  
// 第三种  
public void drawBitmap (Bitmap bitmap, Rect src, Rect dst, Paint paint)  
public void drawBitmap (Bitmap bitmap, Rect src, RectF dst, Paint paint)

第一种方法中后两个参数(matrix, paint)是在绘制的时候对图片进行一些改变，图片左上角位置默认为坐标原点，如果只是需要将图片内容绘制出来只需要如下操作就可以了：

canvas.drawBitmap(bitmap,new Matrix(),new Paint());

第二种方法就是在绘制时指定了图片左上角的坐标(距离坐标原点的距离)：

canvas.drawBitmap(*bitmap*,200,500,new Paint());

第三种方法src表示需要绘制图片的区域，dst表示绘制在屏幕的区域，如果src和dst大小不等，会根据dst的大小进行拉伸缩放。

// 将画布坐标系移动到画布中央

canvas.translate(mWidth/2,mHeight/2);  
  
// 指定图片绘制区域(左上角的四分之一)  
Rect src = new Rect(0,0,*bitmap*.getWidth()/2,*bitmap*.getHeight()/2);  
  
// 指定图片在屏幕上显示的区域  
Rect dst = new Rect(0,0,200,400);  
  
// 绘制图片  
canvas.drawBitmap(*bitmap*,src,dst,null);

#### 绘制文字

绘制文字方法分为3类：

// 第一类

public void drawText (String text, float x, float y, Paint paint)  
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)  
public void drawText (char[] text, int index, int count, float x, float y, Paint paint)  
  
// 第二类  
public void drawPosText (String text, float[] pos, Paint paint)  
public void drawPosText (char[] text, int index, int count, float[] pos, Paint paint)  
  
// 第三类  
public void drawTextOnPath (String text, Path path, float hOffset, float vOffset, Paint paint)  
public void drawTextOnPath (char[] text, int index, int count, Path path, float hOffset, float vOffset, Paint paint)

第一类只能指定文本基线位置(基线x默认在字符串左侧，基线y默认在字符串下方)。

第二类可以分别指定每个文字的位置。

第三类是指定一个路径，根据路径绘制文字。

绘制文字是需要画笔的，文字的大小,颜色,字体,对齐方式都是由画笔控制的。

Paint文本相关常用方法表

|  |  |  |
| --- | --- | --- |
| 标题 | 相关方法 | 备注 |
| 色彩 | setColor setARGB setAlpha | 设置颜色，透明度 |
| 大小 | setTextSize | 设置文本字体大小 |
| 字体 | setTypeface | 设置或清除字体样式 |
| 样式 | setStyle | 填充(FILL),描边(STROKE),填充加描边(FILL\_AND\_STROKE) |
| 对齐 | setTextAlign | 左对齐(LEFT),居中对齐(CENTER),右对齐(RIGHT) |
| 测量 | measureText | 测量文本大小(注意，请在设置完文本各项参数后调用) |

绘制文字首先需要创建画笔：

Paint textPaint = new Paint(); // 创建画笔

textPaint.setColor(Color.*BLACK*); // 设置颜色  
textPaint.setStyle(Paint.Style.*FILL*); // 设置样式  
textPaint.setTextSize(50); // 设置字体大小

1. 第一类(drawText)

第一类可以指定文本开始的位置，可以截取文本中部分内容进行绘制。

其中x，y两个参数是指定文本绘制两个基线,示例：

// 文本(要绘制的内容)

String str = "ABCDEFG";  
// 参数分别为 (文本 基线x 基线y 画笔)  
canvas.drawText(str,200,500,textPaint);

也可以解决字符串的一部分进行绘制，使用start和end指定的区间是前闭后开的，即包含start指定的下标，而不包含end指定的下标。

对于字符数组char[]我们截取字符串使用起始位置(index)和长度(count)来确定。

1. 第二类(drawPosText)

这类方法给每个字符都指定一个位置：

String str = "SLOOP";

canvas.drawPosText(str,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  
},textPaint);

不推荐使用这类方法，原因如下：

1. 必须指定所有字符位置，否则直接crash掉，反人类设计
2. 性能不佳，在大量使用的时候可能导致卡顿
3. 不支持emoji等特殊字符，不支持字形组合与分解

## Path

使用Path不仅能够绘制简单图形，也可以绘制这些比较复杂的图形。另外，根据路径绘制文本和剪裁画布都会用到Path。Path封装了由直线和曲线(二次，三次贝塞尔曲线)构成的几何路径。你能用Canvas中的drawPath来把这条路径画出来(同样支持Paint的不同绘制模式)，也可以用于剪裁画布和根据路径绘制文字。我们有时会用Path来描述一个图像的轮廓，所以也会称为轮廓线(轮廓线仅是Path的一种使用方法，两者并不等价)

Path常用方法:

|  |  |  |
| --- | --- | --- |
| 作用 | 相关方法 | 备注 |
| 移动起点 | moveTo | 移动下一次操作的起点位置 |
| 设置终点 | setLastPoint | 重置当前path中最后一个点位置，如果在绘制之前调用，效果和moveTo相同 |
| 连接直线 | lineTo | 添加上一个点到当前点之间的直线到Path |
| 闭合路径 | close | 连接第一个点连接到最后一个点，形成一个闭合区域 |
| 添加内容 | addRect, addRoundRect, addOval, addCircle, addPath, addArc, arcTo | 添加(矩形， 圆角矩形， 椭圆， 圆， 路径， 圆弧) 到当前Path (注意addArc和arcTo的区别) |
| 是否为空 | isEmpty | 判断Path是否为空 |
| 是否为矩形 | isRect | 判断path是否是一个矩形 |
| 替换路径 | set | 用新的路径替换到当前路径所有内容 |
| 偏移路径 | offset | 对当前路径之前的操作进行偏移(不会影响之后的操作) |
| 贝塞尔曲线 | quadTo, cubicTo | 分别为二次和三次贝塞尔曲线的方法 |
| rXxx方法 | rMoveTo, rLineTo, rQuadTo, rCubicTo | 不带r的方法是基于原点的坐标系(偏移量)， rXxx方法是基于当前点坐标系(偏移量) |
| 填充模式 | setFillType, getFillType, isInverseFillType, toggleInverseFillType | 设置,获取,判断和切换填充模式 |
| 提示方法 | incReserve | 提示Path还有多少个点等待加入(这个方法貌似会让Path优化存储结构) |
| 布尔操作(API19) | op | 对两个Path进行布尔运算(即取交集、并集等操作) |
| 计算边界 | computeBounds | 计算Path的边界 |
| 重置路径 | reset, rewind | 清除Path中的内容reset不保留内部数据结构，但会保留FillType.rewind会保留内部的数据结构，但不保留FillType |
| 矩阵操作 | transform | 矩阵变换 |

### 基础使用

使用path之前，需要创建画笔：

Paint mPaint = new Paint(); // 创建画笔

mPaint.setColor(Color.*BLACK*); // 画笔颜色 - 黑色  
mPaint.setStyle(Paint.Style.*STROKE*); // 填充模式 - 描边  
mPaint.setStrokeWidth(10); // 边框宽度 - 10

#### lineTo

原型如下：

public void lineTo (float x, float y)

该方法实现到某个坐标点的连线，如果之前没有设置原点，默认是坐标原点。

测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心(宽高数据在

onSizeChanged中获取)  
Path path = new Path(); // 创建Path  
path.lineTo(200, 200); // lineTo  
path.lineTo(200,0);  
canvas.drawPath(path, mPaint); // 绘制Path

#### moveTo

moveTo的作用是移动到下一次操作起点的位置，该方法并不会影响之前的操作，对后续的操作产生作用，只改变下一次操作的起点。方法原型如下：

// moveTo

public void moveTo (float x, float y)

测试代码如下：

canvas.translate(getWidth() / 2, getWidth() / 2); // 移动坐标系到屏幕中心

Path path = new Path(); // 创建Path  
path.lineTo(200, 200); // lineTo  
path.moveTo(200,100); // moveTo  
path.lineTo(200,0); // lineTo  
canvas.drawPath(path, mPaint); // 绘制Path

#### setLastPoint

setLastPoint是设置之前操作的最后一个点位置，对前一次操作的结果产生效果，及时前面操作已经绘制完成，对后续的操作也会产生作用：

方法原型如下：

// setLastPoint

public void setLastPoint (float dx, float dy)

测试代码：

canvas.translate(getWidth() / 2, getWidth() / 2); // 移动坐标系到屏幕中心

Path path = new Path(); // 创建Path  
path.lineTo(100, 200); // lineTo  
path.lineTo(200, 200); // lineTo  
path.setLastPoint(200,100); // setLastPoint  
path.lineTo(200,0); // lineTo  
canvas.drawPath(path, mPaint); // 绘制Path

#### close

close方法用于连接当前最后一个点和最初的一个点(如果两个点不重合的话)，最终形成一个封闭的图形。方法原型如下：

public void close ()

测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心

Path path = new Path(); // 创建Path  
path.lineTo(200, 200); // lineTo  
path.lineTo(200,0); // lineTo  
path.close(); // close  
canvas.drawPath(path, mPaint); // 绘制Path

close的作用是封闭路径，与连接当前最后一个点和第一个点并不等价。如果连接了最后一个点和第一个点仍然无法形成封闭图形，则close什么 也不做。

#### 基本形状

相关方法原型：

// 圆形

public void addCircle (float x, float y, float radius, Path.Direction dir)  
// 椭圆  
public void addOval (RectF oval, Path.Direction dir)  
// 矩形  
public void addRect (float left, float top, float right, float bottom, Path.Direction dir)  
public void addRect (RectF rect, Path.Direction dir)  
// 圆角矩形  
public void addRoundRect (RectF rect, float[] radii, Path.Direction dir)  
public void addRoundRect (RectF rect, float rx, float ry, Path.Direction dir)

这些方法最后一个参数都有一个Path.Direction的参数，代表顺时针和逆时针的方向，Direction里面只有2个每局变量，CW代表顺时针，CCW代表逆时针。这个参数的作用是：

1. 在添加图形时确定闭合顺序(各个点的记录顺序)
2. 对图形的渲染结果有影响(是判断图形渲染的重要条件)

测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心

Path path = new Path();  
path.addRect(200, 200, -200, -200, Path.Direction.*CW*);  
path.setLastPoint(-100, 100);  
canvas.drawPath(path, mPaint);

上面方法按照顺时针方向得到4个点，最后一个点就是(200, -200)，改变了最后一个点的位置，但是最后一个点在一些机器上只能显示靠矩形内部的点，矩形外部的点显示不出来。上面addRect的参数交换下顺序，绘制的结果也会不一样，比如改为（-200，-200,200,200）。

#### Path

方法原型：

public void addPath (Path src)

public void addPath (Path src, float dx, float dy)  
public void addPath (Path src, Matrix matrix)

这些方法的功能就是将两个Path合并成为一个。

第三个方法是将src添加到当前path之前先使用Matrix进行变换。

第二个方法比第一个方法多出来的两个参数是将src进行了位移之后再添加进当前path中。

测试代码：

canvas.translate(getWidth()/ 2, getHeight() / 2); // 移动坐标系到屏幕中心

canvas.scale(1,-1); // <-- 注意 翻转y坐标轴  
Path path = new Path();  
Path src = new Path();  
path.addRect(-200,-200,200,200, Path.Direction.*CW*);  
src.addCircle(0,0,200, Path.Direction.*CW*);  
path.addPath(src,200,200);  
mPaint.setColor(Color.*BLACK*); // 绘制合并后的路径  
canvas.drawPath(path,mPaint);

#### addArc与arcTo

方法原型：

// addArc

public void addArc (RectF oval, float startAngle, float sweepAngle)  
// arcTo  
public void arcTo (RectF oval, float startAngle, float sweepAngle)  
public void arcTo (RectF oval, float startAngle, float sweepAngle, boolean forceMoveTo)

addArc和arcTo都是添加一个圆弧到path，addArc是直接添加一个圆弧到path，arcTo是添加一个圆弧到path中，如果圆弧的起点和上次最后一个坐标点不相同，就连接两个点。

各参数作用如下：

oval:圆弧的外切矩形。

startAngle:开始角度

sweepAngle:扫过角度(-360 <= sweepAngle <360), sweepAngle取值范围是 [-360, 360)，不包括360，当 >= 360 或者 < -360 时将不会绘制任何内容， 对于360，你可以用一个接近的值替代，例如: 359.99。

forceMoveTo:是否强制使用MoveTo，true：将最后一个点移动到圆弧起点，即不连接最后一个点与圆弧起点，false：不移动，而是连接最后一个点与圆弧起点

addArc测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心

canvas.scale(1,-1); // <-- 注意 翻转y坐标轴  
Path path = new Path();  
path.lineTo(100,100);  
RectF oval = new RectF(0,0,300,300);  
path.addArc(oval,0,270);  
// path.arcTo(oval,0,270,true); // <-- 和上面一句作用等价  
canvas.drawPath(path,mPaint);

arcTo测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心

canvas.scale(1,-1); // <-- 注意 翻转y坐标轴  
Path path = new Path();  
path.lineTo(100,100);  
RectF oval = new RectF(0,0,300,300);  
path.arcTo(oval,0,-270);  
// path.arcTo(oval,0,270,false); // <-- 和上面一句作用等价  
canvas.drawPath(path,mPaint);

#### isEmpty

方法原型：

public boolean isEmpty ()

判断path中是否包含内容。

测试代码：

Path path = new Path();

Log.*e*("1",path.isEmpty()+"");  
path.lineTo(100,100);  
Log.*e*("2",path.isEmpty()+"");

#### isRect

方法原型：

public boolean isRect (RectF rect)

判断path是否是一个矩形，如果是一个矩形的话，会将矩形的信息存放进参数rect中。

测试代码：

Path path = new Path();

path.lineTo(0,400);  
path.lineTo(400,400);  
path.lineTo(400,0);  
path.lineTo(0,0);  
  
RectF rect = new RectF();  
boolean b = path.isRect(rect);  
Log.*e*("Rect","isRect:"+b+"| left:"+rect.left+"| top:"+rect.top+"| right:"+rect.right+"| bottom:"+rect.bottom);

#### set

方法原型：

public void set (Path src)

将新的path赋值到现有path。

测试代码：

canvas.translate(getWidth() / 2, getHeight() / 2); // 移动坐标系到屏幕中心

canvas.scale(1,-1); // <-- 注意 翻转y坐标轴  
Path path = new Path(); // path添加一个矩形  
path.addRect(-200,-200,200,200, Path.Direction.*CW*);  
Path src = new Path(); // src添加一个圆  
src.addCircle(0,0,100, Path.Direction.*CW*);  
path.set(src); // 大致相当于 path = src;  
canvas.drawPath(path,mPaint);

#### offset

方法原型：

public void offset (float dx, float dy)

public void offset (float dx, float dy, Path dst)

offest就是对path进行一段平移，它和Canvas中的translate作用很像，但Canvas作用于整个画布，而path的offset只作用于当前path。

第二个方法的dst使用来存储平移后的path的，不会影响当前的path，如果dst为null，效果和第一种方法相同。

### 贝塞尔曲线

贝塞尔曲线是用一系列点来控制曲线状态的，可以将这些点简单分为两类：

数据点：确定曲线的起始和结束位置

控制点：确定曲线的弯曲程度

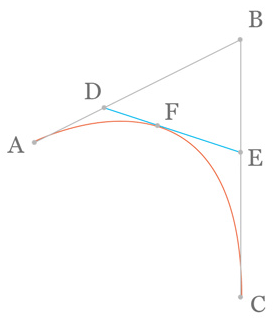
#### 一阶曲线

一阶曲线是没有控制点的，仅有两个数据点(A 和 B)，最终效果一个线段。

一阶曲线其实就是前面讲解过的lineTo。

#### 二阶曲线

二阶曲线由两个数据点(A 和 C)，一个控制点(B)来描述曲线状态，大致如下：



连接AB BC，并在AB上取点D，BC上取点E，使其满足条件：

AD/AB=BE/BC

连接DE，取点F，使得:

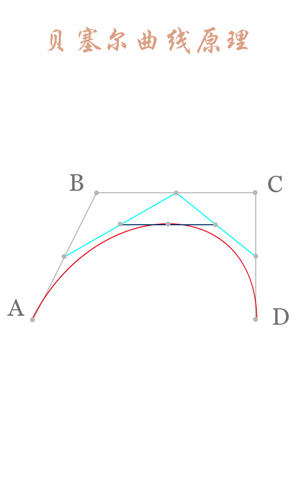
AD/AB=BE/BC=DF/DE

这样获取到的点F就是贝塞尔曲线上的一个点

二阶曲线对应的方法是quadTo

#### 三阶贝塞尔曲线

三阶曲线由两个数据点(A 和 D)，两个控制点(B 和 C)来描述曲线状态，如下：



三阶曲线对应的方法是cubicTo。

### Path方法详解

#### rXxx方法

Xxx和rXxx方法的区别：

Xxx方法是指绝对坐标，例如LineTo，不需要加上当前点偏移

rXxx方法是相对坐标

举个例子:

Path path = new Path();

path.moveTo(100,100);

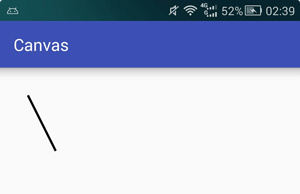
path.lineTo(100,200);  
canvas.drawPath(path,mDeafultPaint);



上面代码先移动点到坐标(100，100)处，之后再连接 点(100，100) 到 (100，200) 之间点直线,非常简单，画出来就是一条竖直的线，那接下来看下一个例子：

Path path = new Path();

path.moveTo(100,100);  
path.rLineTo(100,200);  
canvas.drawPath(path,mDeafultPaint);



上面例子将 lineTo 换成了 rLineTo 可以看到在屏幕上原本是竖直的线变成了倾斜的线。这是因为最终我们连接的是 (100,100) 和 (200, 300) 之间的线段。

#### 填充模式

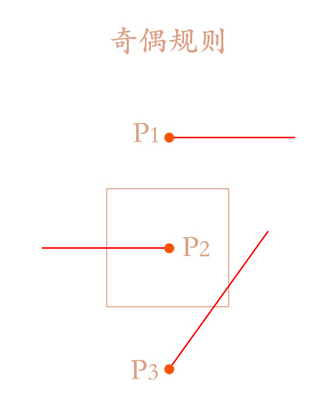
Paint有三种样式，“描边” “填充” 以及 “描边加填充”，这里所了解到就是在Paint设置为后两种样式时不同的填充模式对图形渲染效果的影响。

要给一个图形内部填充颜色，首先需要分清哪一部分是外部，哪一部分是内部，机器判断图形内外，一般有以下两种方法：

|  |  |  |
| --- | --- | --- |
| 方法 | 判定条件 | 解释 |
| 奇偶规则 | 奇数表示在图形内，偶数表示在图形外 | 从任意位置p作一条射线， 若与该射线相交的图形边的数目为奇数，则p是图形内部点，否则是外部点。 |
| 非零环绕数规则 | 若环绕数为0表示在图形外，非零表示在图形内 | 首先使图形的边变为矢量。将环绕数初始化为零。再从任意位置p作一条射线。当从p点沿射线方向移动时，对在每个方向上穿过射线的边计数，每当图形的边从右到左穿过射线时，环绕数加1，从左到右时，环绕数减1。处理完图形的所有相关边之后，若环绕数为非零，则p为内部点，否则，p是外部点。 |

奇偶规则(Even-Odd Rule)

用一个简单示例来说明:



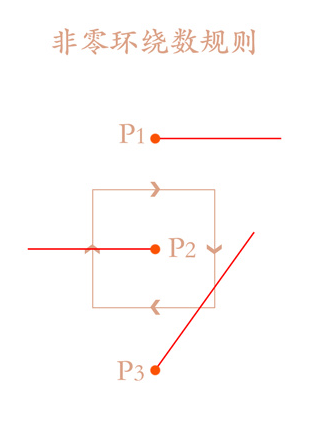
上图中有一个四边形，我们选取了三个点来判断这些点是否在图形内部。

* P1: 从P1发出一条射线，发现图形与该射线相交边数为0，偶数，故P1点在图形外部。
* P2: 从P2发出一条射线，发现图形与该射线相交边数为1，奇数，故P2点在图形内部。
* P3: 从P3发出一条射线，发现图形与该射线相交边数为2，偶数，故P3点在图形外部。

非零环绕数规则(Non-Zero Winding Number Rule)

在给Path中添加图形时需要指定图形的添加方式，是用顺时针还是逆时针，另外我们不论是使用lineTo，quadTo，cubicTo还是其他连接线的方法，都是从一个点连接到另一个点，换言之，Path中任何线段都是有方向性的，这也是使用非零环绕数规则的基础。

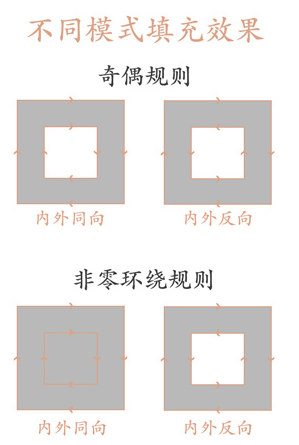
用一个简单的例子来说明非零环绕数规则的用法:



* P1: 从P1点发出一条射线，沿射线防线移动，并没有与边相交点部分，环绕数为0，故P1在图形外边。
* P2: 从P2点发出一条射线，沿射线方向移动，与图形点左侧边相交，该边从左到右穿过穿过射线，环绕数－1，最终环绕数为－1，故P2在图形内部。
* P3: 从P3点发出一条射线，沿射线方向移动，在第一个交点处，底边从右到左穿过射线，环绕数＋1，在第二个交点处，右侧边从左到右穿过射线，环绕数－1，最终环绕数为0，故P3在图形外部。

通常，这两种方法的判断结果是相同的，但也存在两种方法判断结果不同的情况，如下面这种情况：

用非零环绕规则判断，内外同向和内外反向结果肯定不一样。



#### Android中的填充模式

Android中的填充模式有四种，是封装在Path中的一个枚举。

|  |  |
| --- | --- |
| 模式 | 简介 |
| EVEN\_ODD | 奇偶规则 |
| INVERSE\_EVEN\_ODD | 反奇偶规则 |
| WINDING | 非零环绕数规则 |
| INVERSE\_WINDING | 反非零环绕数规则 |

我们可以看到上面有四种模式，分成两对，例如 “奇偶规则” 与 “反奇偶规则” 是一对，它们之间有什么关系呢？

Inverse 和含义是“相反，对立”，说明反奇偶规则刚好与奇偶规则相反，例如对于一个矩形而言，使用奇偶规则会填充矩形内部，而使用反奇偶规则会填充矩形外部，这个会在后面示例中代码展示两者对区别。

#### Android与填充模式相关的方法

|  |  |
| --- | --- |
| 方法 | 作用 |
| setFillType | 设置填充规则 |
| getFillType | 获取当前填充规则 |
| isInverseFillType | 判断是否是反向(INVERSE)规则 |
| toggleInverseFillType | 切换填充规则(即原有规则与反向规则之间相互切换) |

示例演示：

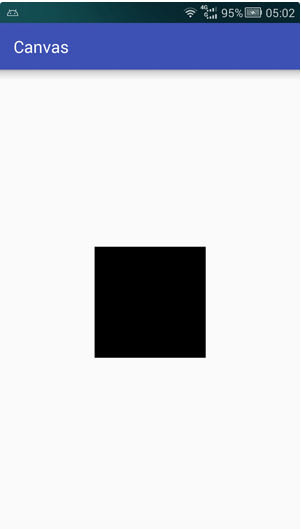
奇偶规则与反奇偶规则

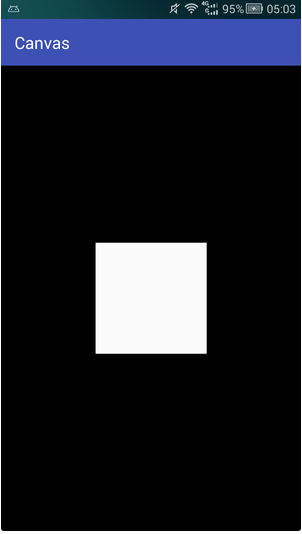
mDeafultPaint.setStyle(Paint.Style.*FILL*); // 设置画布模式为填充

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 移动画布(坐标系)  
Path path = new Path(); // 创建Path  
//path.setFillType(Path.FillType.EVEN\_ODD); // 设置Path填充模式为 奇偶规则  
path.setFillType(Path.FillType.*INVERSE\_EVEN\_ODD*); // 反奇偶规则  
path.addRect(-200,-200,200,200, Path.Direction.*CW*); // 给Path中添加一个矩形

下面两张图片分别是在奇偶规则于反奇偶规则的情况下绘制的结果，可以看出其填充的区域刚好相反：

PS: 白色为背景色，黑色为填充色



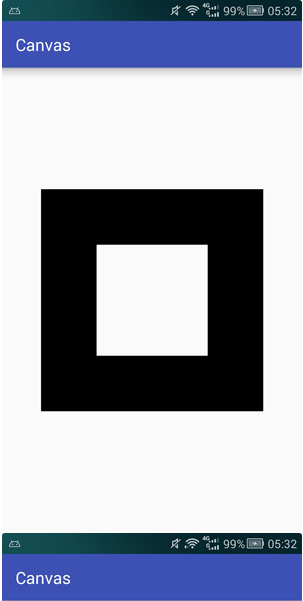


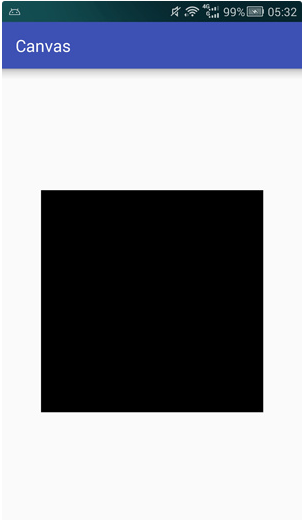
图形边的方向对非零奇偶环绕数规则填充结果的影响

我们之前讨论过给Path添加图形时顺时针与逆时针的作用，除了上次讲述的方便记录外，就是本文所涉及的另外一个重要作用了: “作为非零环绕数规则的判断依据。”

mDeafultPaint.setStyle(Paint.Style.*FILL*); // 设置画笔模式为填充

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 移动画布(坐系)  
 Path path = new Path(); // 创建Path  
// 添加小正方形 (通过这两行代码来控制小正方形边的方向,从而演示不同的效果)  
// path.addRect(-200, -200, 200, 200, Path.Direction.CW);  
path.addRect(-200, -200, 200, 200, Path.Direction.*CCW*);  
// 添加大正方形  
path.addRect(-400, -400, 400, 400, Path.Direction.*CCW*);  
path.setFillType(Path.FillType.*WINDING*); // 设置Path填充模式为非零环绕规则  
canvas.drawPath(path, mDeafultPaint); // 绘制Path





#### 布尔操作(API19)

布尔操作是两个Path之间的运算，主要作用是用一些简单的图形通过一些规则合成一些相对比较复杂，或难以直接得到的图形。

如太极中的阴阳鱼，如果用贝塞尔曲线制作的话，可能需要六段贝塞尔曲线才行，而在这里我们可以用四个Path通过布尔运算得到，而且会相对来说更容易理解一点。

canvas.translate(mViewWidth / 2, mViewHeight / 2);

Path path1 = new Path();  
Path path2 = new Path();  
Path path3 = new Path();  
Path path4 = new Path();  
path1.addCircle(0, 0, 200, Path.Direction.*CW*);  
path2.addRect(0, -200, 200, 200, Path.Direction.*CW*);  
path3.addCircle(0, -100, 100, Path.Direction.*CW*);  
path4.addCircle(0, 100, 100, Path.Direction.*CCW*);  
path1.op(path2, Path.Op.*DIFFERENCE*);  
path1.op(path3, Path.Op.*UNION*);  
path1.op(path4, Path.Op.*DIFFERENCE*);  
canvas.drawPath(path1, mDeafultPaint);



Path的布尔运算有五种逻辑，如下:

|  |  |  |
| --- | --- | --- |
| 逻辑名称 | 类比 | 说明 |
| DIFFERENCE | 差集 | Path1中减去Path2后剩下的部分 |
| REVERSE\_DIFFERENCE | 差集 | Path2中减去Path1后剩下的部分 |
| INTERSECT | 交集 | Path1与Path2相交的部分 |
| UNION | 并集 | 包含全部Path1和Path2 |
| XOR | 异或 | 包含Path1与Path2但不包括两者相交的部分 |

5种布尔运算逻辑示意图如下：

|  |  |
| --- | --- |
| 类比 | 示意图 |
| 差集 | C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\DHMFM1P$H_T5DL_@8OA$S98.png |
| 差集 | C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\9P96303RMQAO3F64STQ(Q~7.png |
| 交集 | C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\{@0D727937C{WQ_O$C[N_)R.png |
| 并集 | C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\MQMBMJI[ADR1E$JO4_~47GY.png |
| 异或 | C:\Users\Administrator\AppData\Roaming\Tencent\Users\282330332\QQ\WinTemp\RichOle\XX[BIWUQTJ~OWUT94F})U%A.png |

#### 布尔运算方法

在Path中的布尔运算有两个方法

boolean op (Path path, Path.Op op)

boolean op (Path path1, Path path2, Path.Op op)

两个方法中的返回值用于判断布尔运算是否成功，它们使用方法如下:

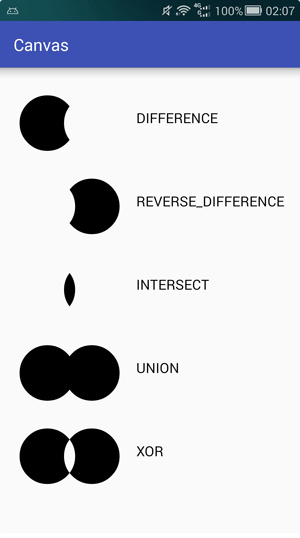
// 对 path1 和 path2 执行布尔运算，运算方式由第二个参数指定，运算结果存入到path1中。

path1.op(path2, Path.Op.*DIFFERENCE*);  
  
// 对 path1 和 path2 执行布尔运算，运算方式由第三个参数指定，运算结果存入到path3中。  
path3.op(path1, path2, Path.Op.*DIFFERENCE*)

布尔运算示例:

int x = 80;

int r = 100;  
  
canvas.translate(250,0);  
  
Path path1 = new Path();  
Path path2 = new Path();  
Path pathOpResult = new Path();  
  
path1.addCircle(-x, 0, r, Path.Direction.*CW*);  
path2.addCircle(x, 0, r, Path.Direction.*CW*);  
  
pathOpResult.op(path1,path2, Path.Op.*DIFFERENCE*);  
canvas.translate(0, 200);  
canvas.drawText("DIFFERENCE", 240,0,mDeafultPaint);  
canvas.drawPath(pathOpResult,mDeafultPaint);  
  
pathOpResult.op(path1,path2, Path.Op.*REVERSE\_DIFFERENCE*);  
canvas.translate(0, 300);  
canvas.drawText("REVERSE\_DIFFERENCE", 240,0,mDeafultPaint);  
canvas.drawPath(pathOpResult,mDeafultPaint);  
  
pathOpResult.op(path1,path2, Path.Op.*INTERSECT*);  
canvas.translate(0, 300);  
canvas.drawText("INTERSECT", 240,0,mDeafultPaint);  
canvas.drawPath(pathOpResult,mDeafultPaint);  
  
pathOpResult.op(path1,path2, Path.Op.*UNION*);  
canvas.translate(0, 300);  
canvas.drawText("UNION", 240,0,mDeafultPaint);  
canvas.drawPath(pathOpResult,mDeafultPaint);  
  
pathOpResult.op(path1,path2, Path.Op.*XOR*);  
canvas.translate(0, 300);  
canvas.drawText("XOR", 240,0,mDeafultPaint);  
canvas.drawPath(pathOpResult,mDeafultPaint);



#### 计算边界

这个方法主要作用是计算Path所占用的空间以及所在位置,方法如下：

void computeBounds (RectF bounds, boolean exact)

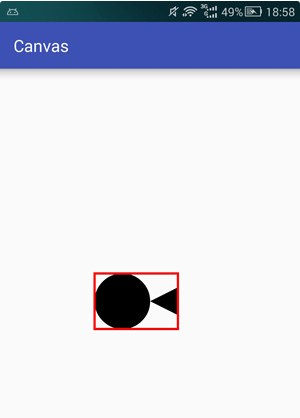
它有两个参数：

|  |  |
| --- | --- |
| 参数 | 作用 |
| bounds | 测量结果会放入这个矩形 |
| exact | 是否精确测量，目前这一个参数作用已经废弃，一般写true即可。 |

简单示例

// 移动canvas,mViewWidth与mViewHeight在 onSizeChanged 方法中获得

canvas.translate(mViewWidth/2,mViewHeight/2);  
RectF rect1 = new RectF(); // 存放测量结果的矩形  
  
Path path = new Path(); // 创建Path并添加一些内容  
path.lineTo(100,-50);  
path.lineTo(100,50);  
path.close();  
path.addCircle(-100,0,100, Path.Direction.*CW*);  
  
path.computeBounds(rect1,true); // 测量Path  
  
canvas.drawPath(path,mDeafultPaint); // 绘制Path  
  
mDeafultPaint.setStyle(Paint.Style.*STROKE*);  
mDeafultPaint.setColor(Color.*RED*);  
canvas.drawRect(rect1,mDeafultPaint); // 绘制边界



#### 重置路径

重置Path有两个方法，分别是reset和rewind，两者区别主要有一下两点：

|  |  |  |
| --- | --- | --- |
| 方法 | 是否保留FillType设置 | 是否保留原有数据结构 |
| reset | 是 | 否 |
| rewind | 否 | 是 |

这个两个方法应该何时选择呢？

选择权重: FillType > 数据结构

因为“FillType”影响的是显示效果，而“数据结构”影响的是重建速度。

### PathMeasure

PathMeasure是一个用来测量Path的类，主要有以下方法:

构造方法

|  |  |
| --- | --- |
| 方法名 | 释义 |
| PathMeasure() | 创建一个空的PathMeasure |
| PathMeasure(Path path,  boolean forceClosed) | 创建 PathMeasure 并关联一个指定的Path(Path需要已经创建完成)。 |

公共方法

|  |  |  |
| --- | --- | --- |
| 返回值 | 方法名 | 释义 |
| void | setPath(Path path, boolean forceClosed) | 关联一个Path |
| boolean | isClosed() | 是否闭合 |
| float | getLength() | 获取Path的长度 |
| boolean | nextContour() | 跳转到下一个轮廓 |
| boolean | getSegment(float startD, float stopD, Path dst, boolean startWithMoveTo) | 截取片段 |
| boolean | getPosTan(float distance, float[] pos, float[] tan) | 获取指定长度的位置坐标及该点切线值 |
| boolean | getMatrix(float distance, Matrix matrix, int flags) | 获取指定长度的位置坐标及该点Matrix |

#### 构造函数

无参构造函数：

PathMeasure ()

用这个构造函数可创建一个空的 PathMeasure，但是使用之前需要先调用 setPath 方法来与 Path 进行关联。被关联的 Path 必须是已经创建好的，如果关联之后 Path 内容进行了更改，则需要使用 setPath 方法重新关联。

有参构造函数：

PathMeasure (Path *path*, boolean forceClosed)

用这个构造函数是创建一个 PathMeasure 并关联一个 Path， 其实和创建一个空的 PathMeasure 后调用 setPath 进行关联效果是一样的，同样，被关联的 Path 也必须是已经创建好的，如果关联之后 Path 内容进行了更改，则需要使用 setPath 方法重新关联。

该方法有两个参数，第一个参数自然就是被关联的 Path 了，第二个参数是用来确保 Path 闭合，如果设置为 true， 则不论之前Path是否闭合，都会自动闭合该 Path(如果Path可以闭合的话)。

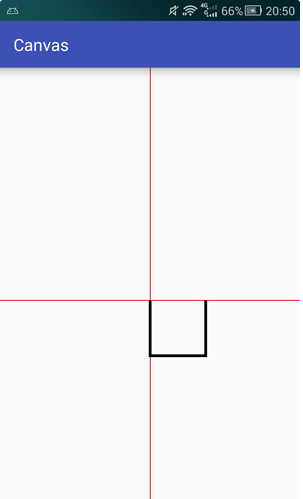
在这里有两点需要明确:

1. 不论 forceClosed 设置为何种状态(true 或者 false)， 都不会影响原有Path的状态，即 Path 与 PathMeasure 关联之后，之前的的 Path 不会有任何改变。
2. forceClosed 的设置状态可能会影响测量结果，如果 Path 未闭合但在与 PathMeasure 关联的时候设置 forceClosed 为 true 时，测量结果可能会比 Path 实际长度稍长一点，获取到到是该 Path 闭合时的状态。

例子：

canvas.translate(mViewWidth/2,mViewHeight/2);

Path path = new Path();  
  
path.lineTo(0,200);  
path.lineTo(200,200);  
path.lineTo(200,0);  
  
PathMeasure measure1 = new PathMeasure(path,false);  
PathMeasure measure2 = new PathMeasure(path,true);  
  
Log.*e*("TAG", "forceClosed=false---->"+measure1.getLength());  
Log.*e*("TAG", "forceClosed=true----->"+measure2.getLength());  
  
canvas.drawPath(path,mDeafultPaint);



由上面例子可以得到结论：

1. Path 与 PathMeasure进行关联并不会影响 Path 状态。
2. forceClosed 为 false 测量的是当前 Path 状态的长度， forceClosed 为 true，则不论Path是否闭合测量的都是 Path 的闭合长度。

#### setPath、 isClosed 和 getLength

setPath 是 PathMeasure 与 Path 关联的重要方法，效果和构造函数中两个参数的作用是一样的。

isClosed 用于判断 Path 是否闭合，但是如果你在关联 Path 的时候设置 forceClosed 为 true 的话，这个方法的返回值则一定为true。

getLength 用于获取 Path 的总长度，在之前的测试中已经用过了。

#### getSegment

getSegment 用于获取Path的一个片段，方法如下：

boolean getSegment (float startD, float stopD, Path dst, boolean startWithMoveTo)

方法各个参数释义：

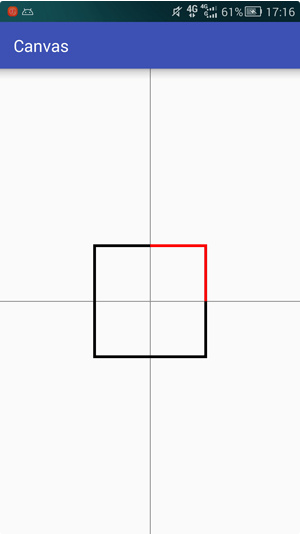
|  |  |  |
| --- | --- | --- |
| 参数 | 作用 | 备注 |
| 返回值(boolean) | 判断截取是否成功 | true 表示截取成功，结果存入dst中，false 截取失败，不会改变dst中内容 |
| startD | 开始截取位置距离 Path 起点的长度 | 取值范围: 0 <= startD < stopD <= Path总长度 |
| stopD | 结束截取位置距离 Path 起点的长度 | 取值范围: 0 <= startD < stopD <= Path总长度 |
| dst | 截取的 Path 将会添加到 dst 中 | 注意: 是添加，而不是替换 |
| startWithMoveTo | 起始点是否使用 moveTo | 用于保证截取的 Path 第一个点位置不变 |

* 如果 startD、stopD 的数值不在取值范围 [0, getLength] 内，或者 startD == stopD 则返回值为 false，不会改变 dst 内容。
* 如果在安卓4.4或者之前的版本，在默认开启硬件加速的情况下，更改 dst 的内容后可能绘制会出现问题，请关闭硬件加速或者给 dst 添加一个单个操作，例如: dst.rLineTo(0, 0)

我们先看看这个方法如何使用：

我们创建了一个 Path， 并在其中添加了一个矩形，现在我们想截取矩形中的一部分，就是下图中红色的部分。

矩形边长400dp，起始点在左上角，顺时针

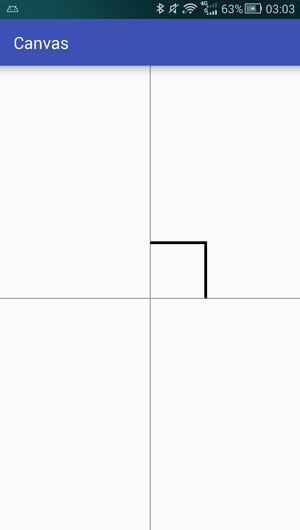


代码：

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 平移坐标系

Path path = new Path(); // 创建Path并添加了一个矩形  
path.addRect(-200, -200, 200, 200, Path.Direction.CW);  
Path dst = new Path(); // 创建用于存储截取后内容的 Path  
PathMeasure measure = new PathMeasure(path, false); // 将 Path 与 PathMeasure 关联  
// 截取一部分存入dst中，并使用 moveTo 保持截取得到的 Path 第一个点的位置不变  
measure.getSegment(200, 600, dst, true);   
canvas.drawPath(dst, mDeafultPaint); // 绘制 dst

结果如下：

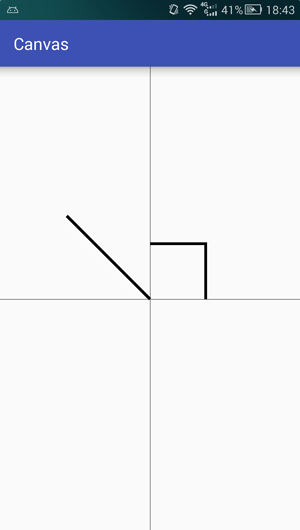


当 dst 中有内容时

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 平移坐标系

Path path = new Path(); // 创建Path并添加了一个矩形  
path.addRect(-200, -200, 200, 200, Path.Direction.*CW*);  
Path dst = new Path(); // 创建用于存储截取后内容的 Path  
dst.lineTo(-300, -300); // <--- 在 dst 中添加一条线段  
PathMeasure measure = new PathMeasure(path, false); // 将 Path 与 PathMeasure 关联  
measure.getSegment(200, 600, dst, true); // 截取一部分 并使用 moveTo 保持截取得到的 Path 第一个点的位置不变  
canvas.drawPath(dst, mDeafultPaint); // 绘制 Path

结果如下:



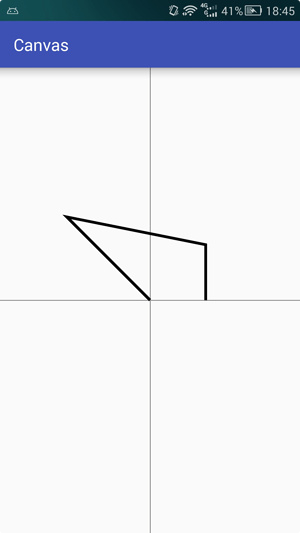
从上面的示例可以看到 dst 中的线段保留了下来，可以得到结论：被截取的 Path 片段会添加到 dst 中，而不是替换 dst 中到内容。

前面两个例子中 startWithMoveTo 均为 true， 如果设置为false会怎样呢?

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 平移坐标系

Path path = new Path(); // 创建Path并添加了一个矩形  
path.addRect(-200, -200, 200, 200, Path.Direction.*CW*);  
Path dst = new Path(); // 创建用于存储截取后内容的 Path  
dst.lineTo(-300, -300); // 在 dst 中添加一条线段  
PathMeasure measure = new PathMeasure(path, false);// 将 Path 与 PathMeasure 关联  
measure.getSegment(200, 600, dst, false); // <--- 截取一部分 不使用 startMoveTo, 保持 dst 的连续性  
canvas.drawPath(dst, mDeafultPaint); // 绘制 Path

结果如下：



从该示例我们又可以得到一条结论：如果 startWithMoveTo 为 true, 则被截取出来到Path片段保持原状，如果 startWithMoveTo 为 false，则会将截取出来的 Path 片段的起始点移动到 dst 的最后一个点，以保证 dst 的连续性。

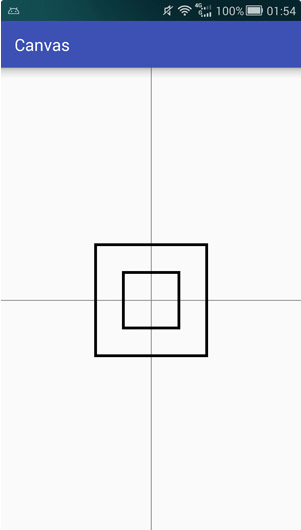
从而我们可以用以下规则来判断 startWithMoveTo 的取值：

|  |  |
| --- | --- |
| 取值 | 主要功用 |
| true | 保证截取得到的 Path 片段不会发生形变 |
| false | 保证存储截取片段的 Path(dst) 的连续性 |

#### nextContour

我们知道 Path 可以由多条曲线构成，但不论是 getLength , getgetSegment 或者是其它方法，都只会在其中第一条线段上运行，而这个 nextContour 就是用于跳转到下一条曲线到方法，如果跳转成功，则返回 true， 如果跳转失败，则返回 false。

如下，我们创建了一个 Path 并使其中包含了两个闭合的曲线，内部的边长是200，外面的边长是400，现在我们使用 PathMeasure 分别测量两条曲线的总长度。



代码：

canvas.translate(mViewWidth / 2, mViewHeight / 2); // 平移坐标系

Path path = new Path();  
path.addRect(-100, -100, 100, 100, Path.Direction.*CW*); // 添加小矩形  
path.addRect(-200, -200, 200, 200, Path.Direction.*CW*); // 添加大矩形  
canvas.drawPath(path,mDeafultPaint); // 绘制 Path  
PathMeasure measure = new PathMeasure(path, false); // 将Path与PathMeasure关联  
float len1 = measure.getLength(); // 获得第一条路径的长度  
measure.nextContour(); // 跳转到下一条路径  
float len2 = measure.getLength(); // 获得第二条路径的长度  
Log.i("LEN","len1="+len1); // 输出两条路径的长度  
Log.i("LEN","len2="+len2);

log输出结果:

com.gcssloop.canvas I/LEN: len1=800.0c

om.gcssloop.canvas I/LEN: len2=1600.0

通过测试，我们可以得到以下内容：

1.曲线的顺序与 Path 中添加的顺序有关。

2.getLength 获取到到是当前一条曲线分长度，而不是整个 Path 的长度。

3.getLength 等方法是针对当前的曲线。

#### getPosTan

这个方法是用于得到路径上某一长度的位置以及该位置的正切值：

boolean getPosTan (float distance, float[] pos, float[] tan)

方法各个参数释义：

|  |  |  |
| --- | --- | --- |
| 参数 | 作用 | 备注 |
| 返回值(boolean) | 判断获取是否成功 | true表示成功，数据会存入 pos 和 tan 中，false 表示失败，pos 和 tan 不会改变 |
| distance | 距离 Path 起点的长度 | 取值范围: 0 <= distance <= getLength |
| pos | 该点的坐标值 | 当前点在画布上的位置，有两个数值，分别为x，y坐标。 |
| tan | 该点的正切值 | 当前点在曲线上的方向，使用 Math.atan2(tan[1], tan[0]) 获取到正切角的弧度值。 |

# 自定义属性

Android自定义view的属性可以在xml中配置，需要经过以下4个步骤：

1. 通过<declare-styleable>为自定义View添加属性
2. 在xml中为相应的属性声明属性值
3. 在运行时（一般为构造函数）获取属性值
4. 将获取到的属性值应用到View

## 为自定义view添加属性

在res/values/attrs.xml中添加自定义属性：

<?xml version="1.0" encoding="utf-8"?>

<resources>  
 <declare-styleable name="Customize">  
 <attr name="attr\_one" format="string" />  
 <attr name="attr\_two" format="string" />  
 <attr name="attr\_three" format="string" />  
 <attr name="attr\_four" format="string" />  
 </declare-styleable>  
 <attr name="CustomizeStyle" format="reference" />  
</resources>

上述定义在<declare-styleable>和没有定义在<declare-styleable>中是有区别的：

1. 如果attr没有定义在<declare-styleable>中，系统只会在R.attr中生成attribute
2. 如果定义在<declare-styleable>中，系统还会在R.styleable中生成R.styleablename的数组。
3. 将attr声明在<declare-styleable>中，系统会帮助我们生成R.styleable这些信息，如果没有声明，也可以手动写代码，但是在属性获取的时候会有警告。

## 在xml中为相应的属性声明属性值

在xml为属性赋值有以下方式：

1. 直接在layout中使用属性
2. 设置style并在style中设置属性
3. Application和Activity可以指定theme，可以在theme中指定在当前Application或Activity中属性的默认值

在layout中使用属性和设置style并在style中设置属性值的方式如下：

<RelativeLayout xmlns:android=<http://schemas.android.com/apk/res/android>

xmlns:ad="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 tools:context=".MainActivity" >  
  
 <com.cold.customattr.CustomView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 ad:attr\_one="attr one in xml"  
 style="@style/ThroughStyle"  
 android:text="hello\_world" />  
  
</RelativeLayout>

上述xmlns:ad=<http://schemas.android.com/apk/res-auto>声明了命名空间，这样在下面的控件可以直接使用自定义属性。

Style引用的是在style.xml中的ThroughStyle

<style name="ThroughStyle">

<item name="attr\_one">attr one from style</item>  
 <item name="attr\_two">attr two from style</item>  
</style>

theme中指定在当前Application或Activity中属性的默认值的方式如下：

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">

<item name="attr\_one">attr one from theme</item>  
 <item name="attr\_two">attr two from theme</item>  
 <item name="attr\_three">attr three from theme</item>  
 <item name="CustomizeStyle">@style/CustomizeStyleInTheme</item>  
</style>  
  
<style name="CustomizeStyleInTheme">  
 <item name="attr\_one">attr one from theme reference</item>  
 <item name="attr\_two">attr two from theme reference</item>  
 <item name="attr\_three">attr three from theme reference</item>  
</style>

## 在运行时获取属性值

在styles.xml中定义一个DefaultCustomStyle的style

<style name="DefaultCustomizeStyle">

<item name="attr\_one">attr one from defalut style res</item>  
 <item name="attr\_two">attr two from defalut style res</item>  
 <item name="attr\_three">attr three from defalut style res</item>  
</style>

CustomView的代码如下：

public class CustomView extends View {

private static final String *TAG* = CustomView.class.getSimpleName();  
  
 public CustomView(Context context) {  
 super(context);  
 }  
  
 public CustomView(Context context, AttributeSet attrs) {  
 this(context, attrs, R.attr.*CustomizeStyle*);  
 }  
  
 public CustomView(Context context, AttributeSet attrs, int defStyle) {  
 super(context, attrs, defStyle);  
  
 TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.*Customize*, defStyle, R.style.*DefaultCustomizeStyle*);  
 String one = a.getString(R.styleable.*Customize\_attr\_one*);  
 String two = a.getString(R.styleable.*Customize\_attr\_two*);  
 String three = a.getString(R.styleable.*Customize\_attr\_three*);  
 String four = a.getString(R.styleable.*Customize\_attr\_four*);  
 Log.*i*(*TAG*, "===========================> one:" + one);  
 Log.*i*(*TAG*, "===========================> two:" + two);  
 Log.*i*(*TAG*, "===========================> three:" + three);  
 Log.*i*(*TAG*, "===========================> four:" + four);  
 a.recycle();  
 }  
}

## View的第三个构造函数的第三个参数defStyle

如果在Code中实例化一个View会调用第一个构造函数，如果在xml中定义会调用第二个构造函数，而第三个函数系统是不调用的，要由View（我们自定义的或系统预定义的View）显式调用，比如在这里我们在第二个构造函数中调用了第三个构造函数，并将R.attr.CustomizeStyle传给了第三个参数。

　　第三个参数的意义就如同它的名字所说的，是默认的Style，这里的默认的Style是指它在当前Application或Activity所用的Theme中的默认Style。

## obtainStyledAtributes

我们要获取的属性值都是通过这个函数返回的TypedArray获得的，以下是函数原型：

public TypedArray obtainStyledAttributes (AttributeSet set, int[] attrs, int defStyleAttr, int defStyleRes)

set：属性值的集合

attrs：我们要获取的属性的资源ID的一个数组，如同ContextProvider中请求数据库时的Projection数组，就是从一堆属性中我们希望查询什么属性的值

defStyleAttr：这个是当前Theme中的一个attribute，是指向style的一个引用，当在layout xml中和style中都没有为View指定属性时，会从Theme中这个attribute指向的Style中查找相应的属性值，这就是defStyle的意思，如果没有指定属性值，就用这个值，所以是默认值，但这个attribute要在Theme中指定，且是指向一个Style的引用，如果这个参数传入0表示不向Theme中搜索默认值

defStyleRes：这个也是指向一个Style的资源ID，但是仅在defStyleAttr为0或defStyleAttr不为0但Theme中没有为defStyleAttr属性赋值时起作用

## 属性使用优先方法

android为属性赋值方式：

1. 直接在xml为属性赋值
2. 在xml中使用style，在style中为属性赋值
3. 在Application或Activity的Theme中为属性赋值
4. 在Application或Activity的Theme中定义style，在style中为属性赋值，并把该style作为view中第三个构造函数的第三个参数(defStyleAttr)
5. 定义一个style，在obtainStyledAttributes第四个参数中使用(defStyleRes)

不同属性赋值优先级

1. 直接在XML中定义 > style定义 > 由defStyleAttr定义的值 > defStyleRes指定的默认值 > 直接在Theme中指定的值
2. defStyleAttr为0或Theme中没有定义defStyle时defStyleRes才起作用。如果defStyleAttr不为0且在当前Theme中可以找到这个attribute的定义时，defStyleRes不起作用，即使在defStyleRes定义了该属性，但defStyleAttr里没有定义，也不会获取该属性值。

# View常用方法

## measure相关

View的绘制过程一半包括3个步骤：

1. 测量：onMeasure，测量View的大小
2. 布局：onLayout，决定View在ViewGroup的位置
3. 绘制：onDraw，把view绘制显示出来

## Android view measure过程

1. view的测量从measure开始，由view所在的ViewGroup调用
2. 在measure内部调用了view的onMeasure方法
3. onMeasure的参数是MeasureSpec类型的，包含了宽高模式和尺寸
4. onMeasure完成了测量过程，内部会调用getSuggestedMinimumWidth和getDefaultSize方法，最后调用setMeasuredDimension保存测量结果
5. setMeasuredDimension内部会调用setMeasuredDimensionRaw储存测量的宽度和高度，并把view的状态位设置为已测量状态
6. 测量完成，ViewGroup可以同过下列方法获取测量的值

1. getMeasuredWidth和getMeasuredHeight方法

2. getMeasuredWidthAndState和getMeasuredHeightAndState方法

3. getMeasuredState方法

## MeasureSpec

MeasureSpec的源码如下：

*/\*\**

*\* MeasureSpec封装了父布局传递给子布局的布局要求，每个MeasureSpec代表了一组宽度和高度的要求   
 \* MeasureSpec由size和mode组成。   
 \* 三种Mode：   
 \* 1.UNSPECIFIED   
 \* 父不没有对子施加任何约束，子可以是任意大小（也就是未指定）   
 \* (UNSPECIFIED在源码中的处理和EXACTLY一样。当View的宽高值设置为0的时候或者没有设置宽高时，模式为UNSPECIFIED   
 \* 2.EXACTLY   
 \* 父决定子的确切大小，子被限定在给定的边界里，忽略本身想要的大小。   
 \* (当设置width或height为match\_parent时，模式为EXACTLY，因为子view会占据剩余容器的空间，所以它大小是确定的)   
 \* 3.AT\_MOST   
 \* 子最大可以达到的指定大小   
 \* (当设置为wrap\_content时，模式为AT\_MOST, 表示子view的大小最多是多少，这样子view会根据这个上限来设置自己的尺寸)   
 \*  
 \* MeasureSpecs使用了二进制去减少对象的分配。   
 \*/*public class MeasureSpec {  
 // 进位大小为2的30次方(int的大小为32位，所以进位30位就是要使用int的最高位和倒数第二位也就是32和31位做标志位)   
 private static final int *MODE\_SHIFT* = 30;  
  
 // 运算遮罩，0x3为16进制，10进制为3，二进制为11。3向左进位30，就是11 00000000000(11后跟30个0)   
 // (遮罩的作用是用1标注需要的值，0标注不要的值。因为1与任何数做与运算都得任何数，0与任何数做与运算都得0）   
 private static final int *MODE\_MASK* = 0x3 << *MODE\_SHIFT*;  
  
 // 0向左进位30，就是00 00000000000(00后跟30个0)   
 public static final int *UNSPECIFIED* = 0 << *MODE\_SHIFT*;  
 // 1向左进位30，就是01 00000000000(01后跟30个0)   
 public static final int *EXACTLY* = 1 << *MODE\_SHIFT*;  
 // 2向左进位30，就是10 00000000000(10后跟30个0)   
 public static final int *AT\_MOST* = 2 << *MODE\_SHIFT*;  
  
 */\*\*  
 \* 根据提供的size和mode得到一个详细的测量结果   
 \*/* // measureSpec = size + mode； (注意：二进制的加法，不是十进制的加法！)   
 // 这里设计的目的就是使用一个32位的二进制数，32和31位代表了mode的值，后30位代表size的值   
 // 例如size=100(4)，mode=AT\_MOST，则measureSpec=100+10000...00=10000..00100   
 public static int makeMeasureSpec(int size, int mode) {  
 return size + mode;  
 }  
  
 */\*\*  
 \* 通过详细测量结果获得mode   
 \*/* // mode = measureSpec & MODE\_MASK;   
 // MODE\_MASK = 11 00000000000(11后跟30个0)，原理是用MODE\_MASK后30位的0替换掉measureSpec后30位中的1,再保留32和31位的mode值。   
 // 例如10 00..00100 & 11 00..00(11后跟30个0) = 10 00..00(AT\_MOST)，这样就得到了mode的值   
 public static int getMode(int measureSpec) {  
 return (measureSpec & *MODE\_MASK*);  
 }  
  
 */\*\*  
 \* 通过详细测量结果获得size   
 \*/* // size = measureSpec & ~MODE\_MASK;   
 // 原理同上，不过这次是将MODE\_MASK取反，也就是变成了00 111111(00后跟30个1)，将32,31替换成0也就是去掉mode，保留后30位的size   
 public static int getSize(int measureSpec) {  
 return (measureSpec & ~*MODE\_MASK*);  
 }  
  
 */\*\*  
 \* 重写的toString方法，打印mode和size的信息，这里省略   
 \*/* public static String toString(int measureSpec) {  
 return null;  
 }  
}

MeasureSpec其实就是尺寸和模式通过各种位运算计算出的一个整型值，它提供了三种模式，还有三个方法（合成约束、分离模式、分离尺寸）。这三种模式对应的意思和布局文件中的参数值的对应关系如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 约束 | 布局参数 | 值 | 说明 |
| UNSPECIFIED  (未指定) |  | 0 | 父控件没有对子控件施加任何约束，子控件可以得到任意想要的大小（使用较少）。 |
| EXACTLY(完全) | match\_parent/具体宽高值 | 1073741824 | 父控件给子控件决定了确切大小，子控件将被限定在给定的边界里而忽略它本身大小。特别说明如果是填充父窗体，说明父控件已经明确知道子控件想要多大的尺寸了（就是剩余的空间都要了） |
| AT\_MOST(至多) | wrap-content | -2147483648 | 子控件至多达到指定大小的值。包裹内容就是父窗体并不知道子控件到底需要多大尺寸（具体值），需要子控件自己测量之后再让父控件给他一个尽可能大的尺寸以便让内容全部显示但不能超过包裹内容的大小 |

UNSPECIFIED一般由类似RecyclerView这样的控件传入。

## View measure方法

Measure的原型是：

public final void measure(int widthMeasureSpec, int heightMeasureSpec)

因为添加了final限制子类没有办法改变measure逻辑，该方法由父类调用，传递的参数代表父类对子类一些宽高尺寸的限制，measure做的工作主要有：

1. 判断当前的MeasureSpec参数与上一次测量的MeasureSpec是否相同，如果相同则不会重新测量
2. 判断此次的测量参数能否在缓存mMeasureCache里面找到,如果可以找到不会重新测量。
3. 如果没有找到缓存或者忽略缓存标志sIgnoreMeasureCache为false，这时会调用onMeasure进行测量。
4. 测量完成后调用setMeasuredDimension保存测量结果，并保存状态值mPrivateFlags中，否则measure里会抛出异常。
5. 最后分别保存本次的MeasureSpec和缓存测量结果到mMeasureCache中。

## onMeasure方法

当确定需要重新测量时，measure会调用onMeasure方法进行测量，View中提供了一个onMeasure的默认实现。

protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

//onMeasure调用了setMeasuredDimension方法，  
 //setMeasuredDimension又需要调用getDefaultSize方法，  
 //getDefaultSize又需要调用getSuggestedMinimumWidth和getSuggestedMinimumHeight方法  
 setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),  
 getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));  
}

在onMeasure中，分别调用了setMeasuredDimension，getDefaultSize和getSuggestedMinimumWidth方法。

## getSuggestedMinimumWidth方法

getSuggestedMinimumWidth方法用于返回View推荐的最小宽度，其代码如下所示：

protected int getSuggestedMinimumWidth() {

//如果没有给View设置背景，那么就返回View本身的最小宽度mMinWidth  
 //如果给View设置了背景，那么就取View本身最小宽度mMinWidth和背景的最小宽度的最大值  
 return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());  
}

如果没有给View设置背景，那么就返回View本身的最小宽度mMinWidth

如果给View设置了背景，那么就取View本身最小宽度mMinWidth和背景的最小宽度的最大值

第一种情况是，mMinWidth是在View的构造函数中被赋值的，View通过读取XML中定义的minWidth的值来设置View的最小宽度mMinWidth.

第二种情况是调用View的setMinimumWidth方法给View的最小宽度mMinWidth赋值.

getSuggestedMinimumHeight与getSuggestedMinimumWidth的逻辑一致。

## getDefaultSize方法

Android会将View想要的尺寸以及其父控件对其尺寸限制信息measureSpec传递给getDefaultSize方法，该方法要根据这些综合信息计算最终的量算的尺寸。

public static int getDefaultSize(int size, int measureSpec) {

//size表示的是View想要的尺寸信息，比如最小宽度或最小高度  
 int result = size;  
 //从measureSpec中解析出specMode信息  
 int specMode = MeasureSpec.getMode(measureSpec);  
 //从measureSpec中解析出specSize信息，不要将specSize与上面的size变量搞混  
 int specSize = MeasureSpec.getSize(measureSpec);  
  
 switch (specMode) {  
 //如果mode是UNSPECIFIED，表示View的父ViewGroup没有给View在尺寸上设置限制条件  
 case MeasureSpec.UNSPECIFIED:  
 //此处当mode是UNSPECIFIED时，View就直接用自己想要的尺寸size作为量算的结果  
 result = size;  
 break;  
 //如果mode是UNSPECIFIED，那么表示View最大可以取其父ViewGroup给其指定的尺寸  
 //如果mode是EXACTLY，那么表示View必须使用其父ViewGroup指定的尺寸  
 case MeasureSpec.AT\_MOST:  
 case MeasureSpec.EXACTLY:  
 //此处mode是UNSPECIFIED或EXACTLY时，View就用其父ViewGroup指定的尺寸作为量算的结果  
 result = specSize;  
 break;  
 }  
 return result;  
}

首先根据measuredSpec解析出对应的specMode和specSize

当mode是UNSPECIFIED时，View就直接用自己想要的尺寸size作为量算的结果

当mode是AT\_MOST或EXACTLY时，View就用其父ViewGroup指定的尺寸作为量算的结果.最终，View会根据measuredSpec限制条件，得到最终的量算的尺寸。

## setMeasuredDimension方法

setMeasuredDimension的源码如下：

protected final void setMeasuredDimension(int measuredWidth, int measuredHeight)

boolean optical = isLayoutModeOptical(this);  
 if (optical != isLayoutModeOptical(mParent)) {  
 //layoutMode是LAYOUT\_MODE\_OPTICAL\_BOUNDS的特殊情况，我们不考虑  
 Insets insets = getOpticalInsets();  
 int opticalWidth = insets.left + insets.right;  
 int opticalHeight = insets.top + insets.bottom;  
  
 measuredWidth += optical ? opticalWidth : -opticalWidth;  
 measuredHeight += optical ? opticalHeight : -opticalHeight;  
 }  
 //最终调用setMeasuredDimensionRaw方法，将量算结果传入进去  
 setMeasuredDimensionRaw(measuredWidth, measuredHeight);  
}

该方法会在开始判断layoutMode是不是LAYOUT\_MODE\_OPTICAL\_BOUNDS的特殊情况,最后将量算的结果传递给方法setMeasuredDimensionRaw。

## setMeasuredDimensionRaw方法

setMeasuredDimensionRaw接收两个参数，分别是已经量算完成的宽度和高度。

private void setMeasuredDimensionRaw(int measuredWidth, int measuredHeight) {

//将量算完成的宽度measuredWidth保存到View的成员变量mMeasuredWidth中  
 mMeasuredWidth = measuredWidth;  
 //将量算完成的高度measuredHeight保存到View的成员变量mMeasuredHeight中  
 mMeasuredHeight = measuredHeight;  
 //最后将View的状态位mPrivateFlags设置为已量算状态  
 mPrivateFlags |= PFLAG\_MEASURED\_DIMENSION\_SET;  
}

在该方法中做了三件事：

将量算完成的宽度measuredWidth保存到View的成员变量mMeasuredWidth中

将量算完成的高度measuredHeight保存到View的成员变量mMeasuredHeight中

最后将View的状态位mPrivateFlags设置为已量算状态

## 量算完成的尺寸的state

View的量算过程就完成了，但是View的父ViewGroup可以通过下面三组方法得到测量结果，分别是：

1. getMeasuredWidth和getMeasuredHeight方法

2. getMeasuredWidthAndState和getMeasuredHeightAndState方法

3. getMeasuredState方法

Android为让其View的父控件获取更多的信息，就在mMeasuredWidth上下了很大功夫，虽然是一个Int值，但是想让它存储更多信息，具体来说就是把mMeasuredWidth分成两部分：

1. 高位的第一个字节为第一部分，用于标记量算完的尺寸是不是达到了View想要的宽度，我们称该信息为量算的state信息。
2. 低位的三个字节为第二部分，用于存储实际的量算到的宽度。

这个有点类似于measureSpec的道理，但是二者又有不同：

measureSpec是将限制条件mode从ViewGroup传递给其子View。

mMeasuredWidth、mMeasuredHeight是将带有量算结果的state标志位信息从View传递给其父ViewGroup。

其中对于mMeasuredWidth、mMeasuredHeight的处理是在resolveSizeAndState方法中

## resolveSizeAndState方法

resolveSizeAndState方法源码如下:

public static int resolveSizeAndState(int size,int measureSpec,int childMeasuredState){

final int specMode = MeasureSpec.*getMode*(measureSpec);  
 final int specSize = MeasureSpec.*getSize*(measureSpec);  
 final int result;  
 switch (specMode) {  
 case MeasureSpec.*AT\_MOST*:  
 if (specSize < size) {  
 //当specMode为AT\_MOST，并且父控件指定的尺寸specSize小于View自己想要的尺寸时，  
 //我们就会用掩码MEASURED\_STATE\_TOO\_SMALL向量算结果加入尺寸太小的标记  
 //这样其父ViewGroup就可以通过该标记其给子View的尺寸太小了，  
 //然后可能分配更大一点的尺寸给子View  
 result = specSize | *MEASURED\_STATE\_TOO\_SMALL*;  
 } else {  
 result = size;  
 }  
 break;  
 case MeasureSpec.*EXACTLY*:  
 result = specSize;  
 break;  
 case MeasureSpec.*UNSPECIFIED*:  
 default:  
 result = size;  
 }  
 return result | (childMeasuredState & *MEASURED\_STATE\_MASK*);  
}

当specMode为AT\_MOST，并且父控件指定的尺寸specSize小于View自己想要的尺寸时，我们就会用掩码MEASURED\_STATE\_TOO\_SMALL向量算结果加入尺寸太小的标记，这样其父ViewGroup就可以通过该标记其给子View的尺寸太小了，然后可能分配更大一点的尺寸给子View。

getDefaultSize方法只是onMeasure方法中获取最终尺寸的默认实现，其返回的信息比resolveSizeAndState要少，那么什么时候才会调用resolveSizeAndState方法呢？ 主要有两种情况：

Android中的许多layout类都调用了resolveSizeAndState方法，比如LinearLayout在量算过程中会调用resolveSizeAndState方法而非getDefaultSize方法。

我们自己在实现自定义的View或ViewGroup时，我们可以重写onMeasure方法，并在该方法内调用resolveSizeAndState方法。

## getMeasuredWidth和getMeasuredHeight方法

该组方法只返回量算结果中的的尺寸信息，去掉了高位字节的state信息，以getMeasuredWidth方法为例，其源码如下：

public final int getMeasuredWidth() {

//MEASURED\_SIZE\_MASK的值为0x00ffffff，用mMeasuredWidth与掩码MEASURED\_SIZE\_MASK进行按位与运算，  
 //可以将返回值中的高位字节的8个bit位全置为0，从而去掉了高位字节的state信息  
 return mMeasuredWidth & *MEASURED\_SIZE\_MASK*;  
}

## getMeasuredWidthAndState和getMeasuredHeightAndState方法

该组方法返回的量算结果中同时包含尺寸和state信息（如果state存在的话），以getMeasuredWidthAndState方法为例，其源码如下所示：

public final int getMeasuredWidthAndState() {

//该方法直接返回成员变量mMeasuredWidth，因为mMeasuredWidth本身已经包含了尺寸以及可能的state信息  
 return mMeasuredWidth;  
}

## getMeasuredState方法

该方法返回的Int值中同时包含宽度量算的state以及高度量算的state，不包含任何的尺寸信息，其源码如下所示：

public final int getMeasuredState() {

//将宽度量算的state存储在Int值的第一个字节中，即高位首字节  
 //将高度量算的state存储在Int值的第三个字节中  
 return (mMeasuredWidth&*MEASURED\_STATE\_MASK*)  
 | ((mMeasuredHeight>>*MEASURED\_HEIGHT\_STATE\_SHIFT*)  
 & (*MEASURED\_STATE\_MASK*>>*MEASURED\_HEIGHT\_STATE\_SHIFT*));  
}

将得到的宽度的state与高度的state进行按位或操作，这样就将宽度和高度的state都保存在一个Int值中：第一个字节存储宽度的state，第三个字节存储高度的state。

## ViewGroup onMeasure方法

要自定义ViewGroup就必须重写onMeasure方法，在这里测量子控件的尺寸，ViewGroup中提供了三个关于测量子控件的方法：

*/\*\**

*\*遍历ViewGroup中所有的子控件，调用measuireChild测量宽高  
 \*/*protected void measureChildren (int widthMeasureSpec, int heightMeasureSpec) {  
 final int size = mChildrenCount;  
 final View[] children = mChildren;  
 for (int i = 0; i < size; ++i) {  
 final View child = children[i];  
 if ((child.mViewFlags & VISIBILITY\_MASK) != *GONE*) {  
 //测量某一个子控件宽高  
 measureChild(child, widthMeasureSpec, heightMeasureSpec);  
 }  
 }  
}  
  
*/\*\*  
 \* 测量某一个child的宽高  
 \*/*protected void measureChild (View child, int parentWidthMeasureSpec,  
 int parentHeightMeasureSpec) {  
 final LayoutParams lp = child.getLayoutParams();  
 //获取子控件的宽高约束规则  
 final int childWidthMeasureSpec = *getChildMeasureSpec*(parentWidthMeasureSpec,  
 mPaddingLeft + mPaddingRight, lp. width);  
 final int childHeightMeasureSpec = *getChildMeasureSpec*(parentHeightMeasureSpec,  
 mPaddingTop + mPaddingBottom, lp. height);  
  
 child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
}  
  
*/\*\*  
 \* 测量某一个child的宽高，考虑margin值  
 \*/*protected void measureChildWithMargins (View child,  
 int parentWidthMeasureSpec, int widthUsed,  
 int parentHeightMeasureSpec, int heightUsed) {  
 final MarginLayoutParams lp = (MarginLayoutParams) child.getLayoutParams();  
 //获取子控件的宽高约束规则  
 final int childWidthMeasureSpec = *getChildMeasureSpec*(parentWidthMeasureSpec,  
 mPaddingLeft + mPaddingRight + lp. leftMargin + lp.rightMargin  
 + widthUsed, lp. width);  
 final int childHeightMeasureSpec = *getChildMeasureSpec*(parentHeightMeasureSpec,  
 mPaddingTop + mPaddingBottom + lp. topMargin + lp.bottomMargin  
 + heightUsed, lp. height);  
 //测量子控件  
 child.measure(childWidthMeasureSpec, childHeightMeasureSpec);  
}

measureChildren 就是遍历所有子控件挨个测量，最终测量子控件的方法就是measureChild 和measureChildWithMargins，measureChildWithMargins跟measureChild的区别就是父控件支不支持margin属性。

## getChildMeasureSpec方法

measureChildWithMargins跟measureChild 都调用了这个方法，作用就是通过父控件的宽高约束规则和父控件加在子控件上的宽高布局参数生成一个子控件的约束。View的onMeasure方法需要两个参数（父控件对View的宽高约束），这个宽高约束就是通过这个方法生成的。子控件的宽高约束规则是父控件调用getChildMeasureSpec方法生成的。



getChildMeasureSpec方法的结果能够用一个表来概括：

|  |  |  |  |
| --- | --- | --- | --- |
| 父控件的约束规则 | 子控件的宽高属性 | 子控件的约束规则 | 说明 |
| EXACTLY（父控件是填充父窗体，或者具体size值） | 具体的size（20dip）/MATCH\_PARENT | EXACTLY | 子控件如果是具体值，约束尺寸就是这个值，模式为确定的；子控件为填充父窗体，约束尺寸是父控件剩余大小，模式为确定的。 |
|  | WRAP-CONTENT | AT\_MOST | 子控件如果是包裹内容，约束尺寸值为父控件剩余大小 ，模式为至多 |
| AT\_MOST（父控件是包裹内容） | 具体的size（20dip） | EXACTLY | 子控件如果是具体值，约束尺寸就是这个值，模式为确定的； |
|  | MATCH\_PARENT/WRAP\_CONTENT | AT\_MOST | 子控件为填充父窗体或者包裹内容 ，约束尺寸是父控件剩余大小 ，模式为至多 |
| UNSPECIFIED（父控件未指定） | 具体的size（20dip） | EXACTLY | 子控件如果是具体值，约束尺寸就是这个值，模式为确定的； |
|  | MATCH\_PARENT/WRAP\_CONTENT | UNSPECIFIED | 子控件为填充父窗体或者包裹内容 ，约束尺寸0，模式为未指定 |

# layout相关

android控件的Layout从View的layout方法开始，该方法添加了final修饰，子类不允许修改其结构，在该方法内部会调用onLayout方法。

## View的layout方法

*/\*\**

*\* 给View和其所有子View分配大小和位置   
 \*  
 \* 这是布局的第二个阶段（第一个阶段是测量）。在这个阶段中，每个父视图需要去调用layout去为他所有的子视图确定位置   
 \* 派生的子类不应该重写layout方法，应该重写onLayout方法，在onlayout方法中应该去调用每一个view的layout   
 \*/*public void layout(int l, int t, int r, int b) {  
 // 将当前视图的左上右下记录为old值(参数中传入的为新的l，t,r,b值)   
 int oldL = mLeft;  
 int oldT = mTop;  
 int oldB = mBottom;  
 int oldR = mRight;  
  
 // setFrame方法的作用就是将新传入的ltrb属性赋值给View，然后判断当前View大小和位置是否发生了变化并返回   
 boolean changed = setFrame(l, t, r, b);  
  
 if (changed || (mPrivateFlags & PFLAG\_LAYOUT\_REQUIRED) == PFLAG\_LAYOUT\_REQUIRED) {  
 // 调用onLayout回调方法，具体实现由重写了onLayout方法的ViewGroup的子类去实现（后面详细说明）   
 onLayout(changed, l, t, r, b);  
 mPrivateFlags &= ~PFLAG\_LAYOUT\_REQUIRED;  
  
 // 调用所有重写了onLayoutChange监听的方法，通知View大小和位置发生了改变   
 ListenerInfo li = mListenerInfo;  
 if (li != null && li.mOnLayoutChangeListeners != null) {  
 ArrayList<OnLayoutChangeListener> listenersCopy =  
 (ArrayList<OnLayoutChangeListener>)li.mOnLayoutChangeListeners.clone();  
 int numListeners = listenersCopy.size();  
 for (int i = 0; i < numListeners; ++i) {  
 listenersCopy.get(i).onLayoutChange(this, l, t, r, b, oldL, oldT, oldR, oldB);  
 }  
 }  
 }  
 mPrivateFlags &= ~PFLAG\_FORCE\_LAYOUT;  
}

如果视图的大小和位置发生变化后，会调用我们前面分析过的onLayout方法。对于onLayout方法的最终实现全部依靠我们在自定义ViewGroup类中重写的onLayout去实现。

## onLayout方法

*/\*\**

*\* 当这个view和其子view被分配一个大小和位置时，被layout调用。   
 \** ***@param*** *changed 当前View的大小和位置改变了   
 \** ***@param*** *left 左部位置（相对于父视图）   
 \** ***@param*** *top 顶部位置（相对于父视图）   
 \** ***@param*** *right 右部位置（相对于父视图）   
 \** ***@param*** *bottom 底部位置（相对于父视图）   
 \*/*protected void onLayout(boolean changed, int left, int top, int right, int bottom) {}

onLayout被定义为抽象方法，所以在继承ViewGroup时必须要重写该方法（onMeasure不需要）。另外这个方法也被override标注，所以也是重写的方法，他重写的是其父类view中的onLayout方法,View的onLayout方法不是虚方法。

重写onLayout的目的就是对当前视图和其所有子View设置它们在父视图中具体位置（确定这个位置就依靠mLeft，mTop，mRight，mBottom这四个值）,mLeft，mTop，mRight，mBottom这四个值表示的是子view相对于父view的位置。

下面列出父view的各个值是如何计算和相关函数：

mLeft，mTop，mRight，mBottom：

view.getLeft()——mLeft：子View左边界到父view左边界的距离

public final int getLeft() {

return mLeft;  
}

view.getTop()——mTop：子View上边界到父view上边界的距离

view.getRight()——mRight：子View右边界到父view左边界的距离

view.getBottom()——mBottom：子View下边距到父View上边界的距离

## 相关函数

视图宽度 view.getWidth();子View的右边界 - 子view的左边界。

public final int getWidth() {

return mRight - mLeft;  
}

视图高度 view.getHeight() ;子View的下边界 - 子view的上边界。

public final int getHeight() {

return mBottom - mTop;  
}

测量宽高：

view.getMeasuredWidth();measure过程中返回的mMeasuredWidth

public final int getMeasuredWidth() {

return mMeasuredWidth & *MEASURED\_SIZE\_MASK*;  
}

view.getMeasuredHeight();measure过程中返回的mMeasuredHeight

public final int getMeasuredHeight() {

return mMeasuredHeight & *MEASURED\_SIZE\_MASK*;  
}

## getWidth和getMeasuredWidth的区别

getWidth/getHeight和getMeasuredWidth/getMeasuredHeight的区别：

getWidth,和getLeft等这些函数都是View相对于其父View的位置。而getMeasuredWidth,getMeasuredHeight是测量后该View的实际值。

实际上在当屏幕可以包裹内容的时候，他们的值是相等的，只有当view超出屏幕后，才能看出他们的区别：

getMeasuredHeight()是实际View的大小，与屏幕无关，而getHeight的大小此时则是屏幕的大小。当超出屏幕后，getMeasuredHeight()等于getHeight()加上屏幕之外没有显示的大小

# draw相关

## draw方法

ViewRoot中的代码会继续执行并创建出一个Canvas对象，然后调用View的draw()方法来执行具体的绘制工作,draw()方法内部的绘制过程总共可以分为六步:

/\*

\* Draw traversal performs several drawing steps which must be executed  
 \* in the appropriate order:  
 \*  
 \* 1. Draw the background  
 \* 2. If necessary, save the canvas' layers to prepare for fading  
 \* 3. Draw view's content  
 \* 4. Draw children  
 \* 5. If necessary, draw the fading edges and restore layers  
 \* 6. Draw decorations (scrollbars for instance)  
 \*/

翻译过来就是：

1. 绘制背景
2. 如果需要，为准备渐变保存画布层
3. 绘制内容(onDraw)
4. 绘制子View(dispatchDraw)
5. 如果需要，绘制渐变并且恢复画布层
6. 绘制其他装饰，例如滚动条

## onDraw方法

onDraw是个空方法，因为每个视图的内容部分肯定都是各不相同的，这部分的功能交给子类来去实现。

## dispatchDraw方法

View中的dispatchDraw()方法是一个空方法，因为这个方法是绘制子View，所以交给了ViewGroup，ViewGroup的dispatchDraw()方法中就会有具体的绘制代码。

# onSizeChanged方法

该方法在view的大小发生改变时由系统回调，该方法的调用流程是：

layout()->setFrame()->sizeChange()->onSizeChanged()

也就是发生在layout过程，如果view的实际位置或大小发生改变了就调用该方法。

# 自定义 View的其他常用方法

onFinishInflate() 当View中所有的子控件 均被映射成xml后触发

onMeasure(int, int) 确定所有子元素的大小

onLayout(boolean, int, int, int, int) 当View分配所有的子元素的大小和位置时触发

onDraw(Canvas) view渲染内容的细节

onKeyDown(int, KeyEvent) 有按键按下后触发

onKeyUp(int, KeyEvent) 有按键按下后弹起时触发

onTrackballEvent(MotionEvent) 轨迹球事件

onTouchEvent(MotionEvent) 触屏事件

onFocusChanged(boolean, int, Rect) 当View获取 或失去焦点时触发

onWindowFocusChanged(boolean) 当窗口包含的view获取或失去焦点时触发

onAttachedToWindow() 当view被附着到一个窗口时触发

onDetachedFromWindow() 当view离开附着的窗口时触发，Android123提示该方法和 onAttachedToWindow() 是相反的。

　onWindowVisibilityChanged(int) 当窗口中包含的可见的view发生变化时触发

# 目录1

## 目录2

### 目录3

#### 目录4

##### 目录5