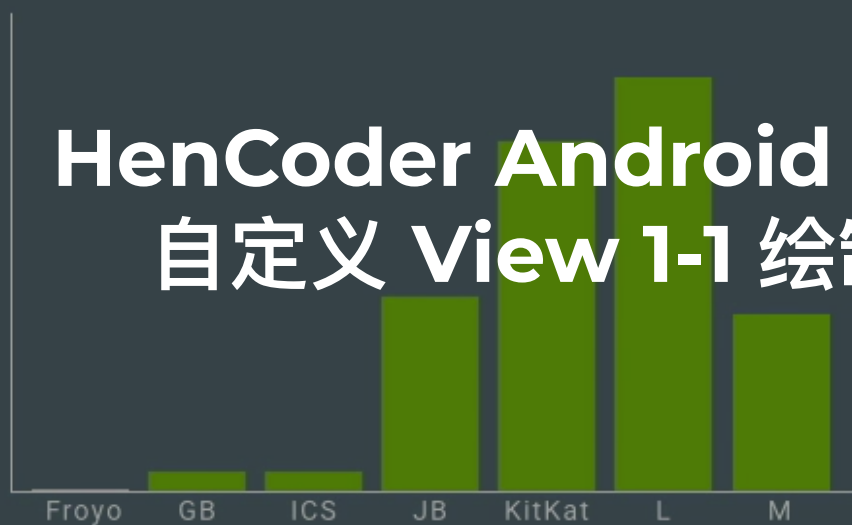


# HenCoder Android 开发进阶

## 自定义 View 1-1 绘制基础



Lollipop —

KitKat —



首先总结一下视频中的关键点：

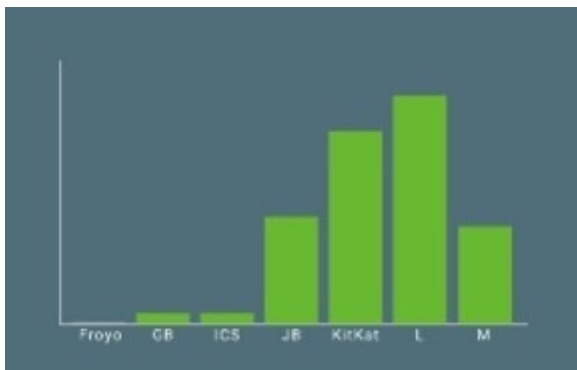
- 自定义绘制的方式是重写绘制方法，其中最常用的是 `onDraw()`
- 绘制的关键是 `Canvas` 的使用
  - `Canvas` 的绘制类方法：`drawXXX()`（关键参数：`Paint`）
  - `Canvas` 的辅助类方法：范围裁切和几何变换
- 可以使用不同的绘制方法来控制遮盖关系

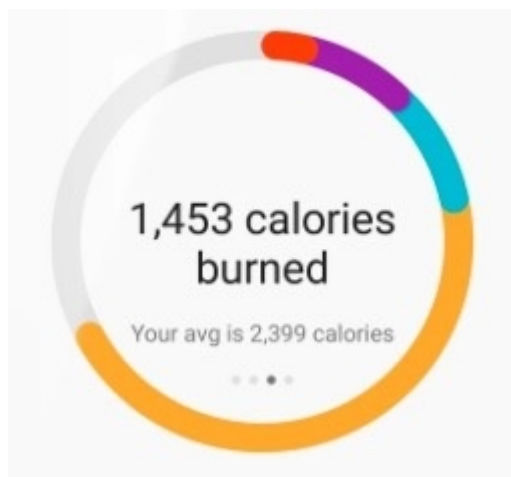
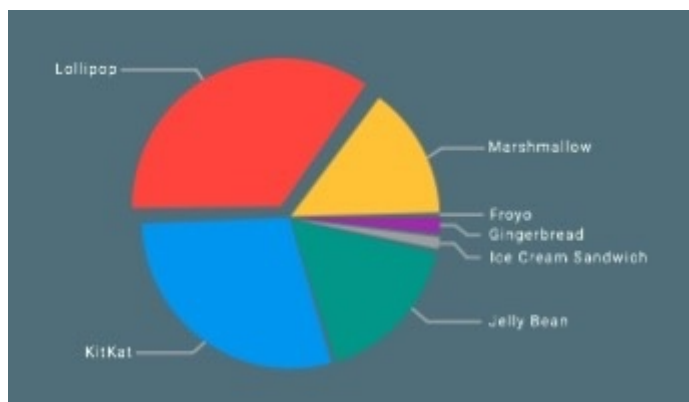
概念已经在视频里全部讲出来了，知识点并不多，但你可能也看出来了，我讲得并不细。这是因为知识点虽然不多，但细节还是很多的，仅仅靠一节分享不可能讲完。我按照顺序把这些知识分成了 4 个级别，拆成几节来讲，你按照这 4 个级别的顺序学习下来，就能够平滑地逐步进阶。

## 自定义绘制知识的四个级别

### 1. `Canvas` 的 `drawXXX()` 系列方法及 `Paint` 最常见的使用

`Canvas.drawXXX()` 是自定义绘制最基本的操作。掌握了这些方法，你才知道怎么绘制内容，例如怎么画圆、怎么画方、怎么画图像和文字。组合绘制这些内容，再配合上 `Paint` 的一些常见方法来对绘制内容的颜色和风格进行简单的配置，就能够应付大部分的绘制需求了。





今天这篇分享我要讲的就是这些内容。也就是说，你在看完这篇文章**并做完练习**之后，上面这几幅图你就会绘制出来了。从今以后，你也很少再需要假装一本正经地对设计师说「不行这个图技术上实现不了」，也不用心惊胆战得等待设计师的那句「那 iOS 怎么可以」了。



都 2017 了，你蒙我  
连台词都不带换的？

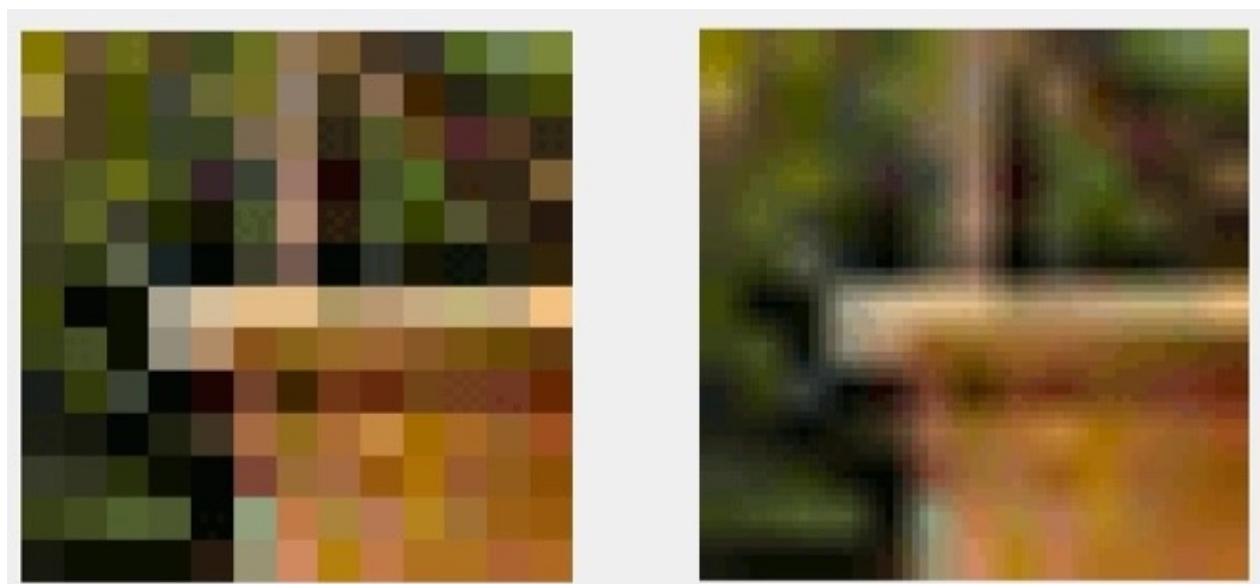
## 2. Paint 的完全攻略

Paint 可以做的事，不只是设置颜色，也不只是我在视频里讲的实心空心、线条粗细、有没有阴影，它可以做的风格设置真的是非常多、非常细。例如：

拐角要什么形状？



开不开双线性过滤？



加不加特效？



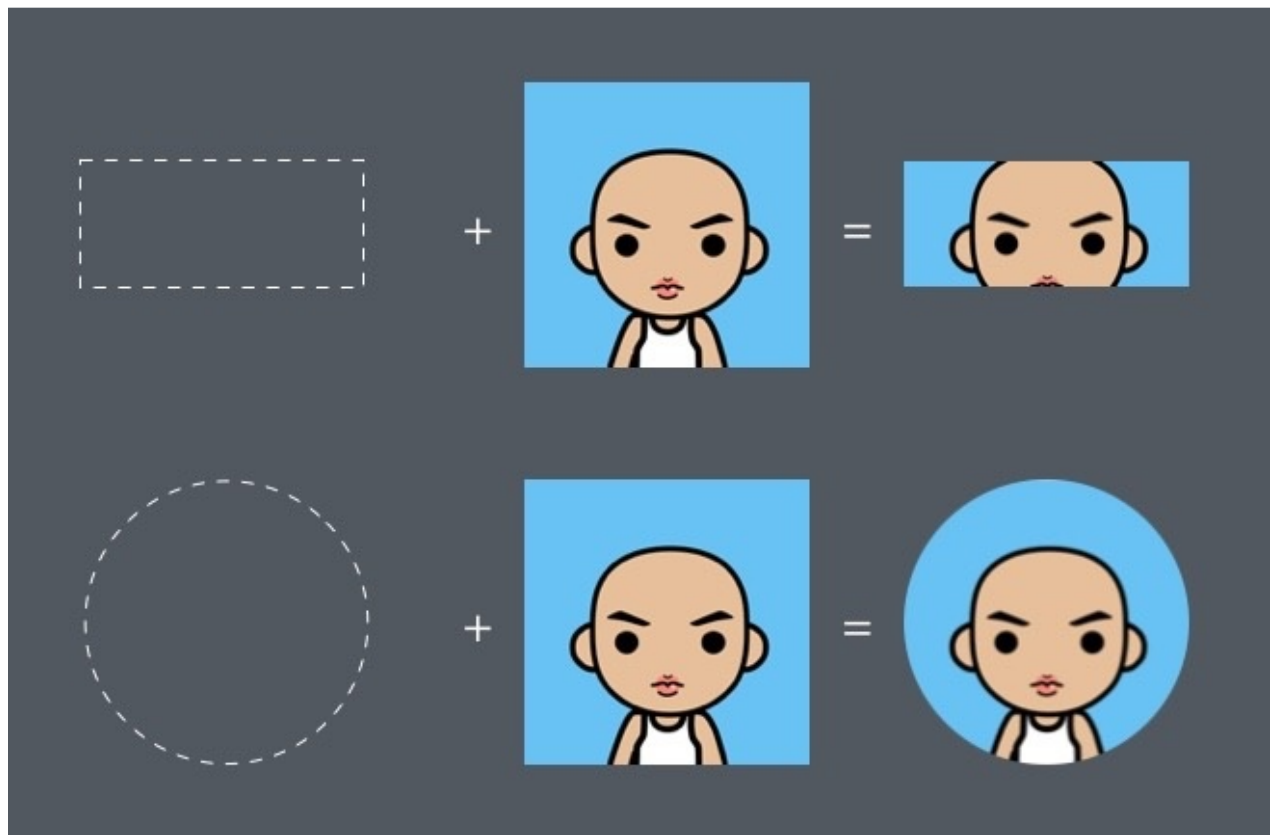
可以调节的非常多，我就不一一列举了。当你掌握到这个级别，就真的不会有什么东西会是 iOS 能做到但你做不到的了。就算设计师再设计出了很难做的东西，做不出来的也不再会是你们 Android 组了。



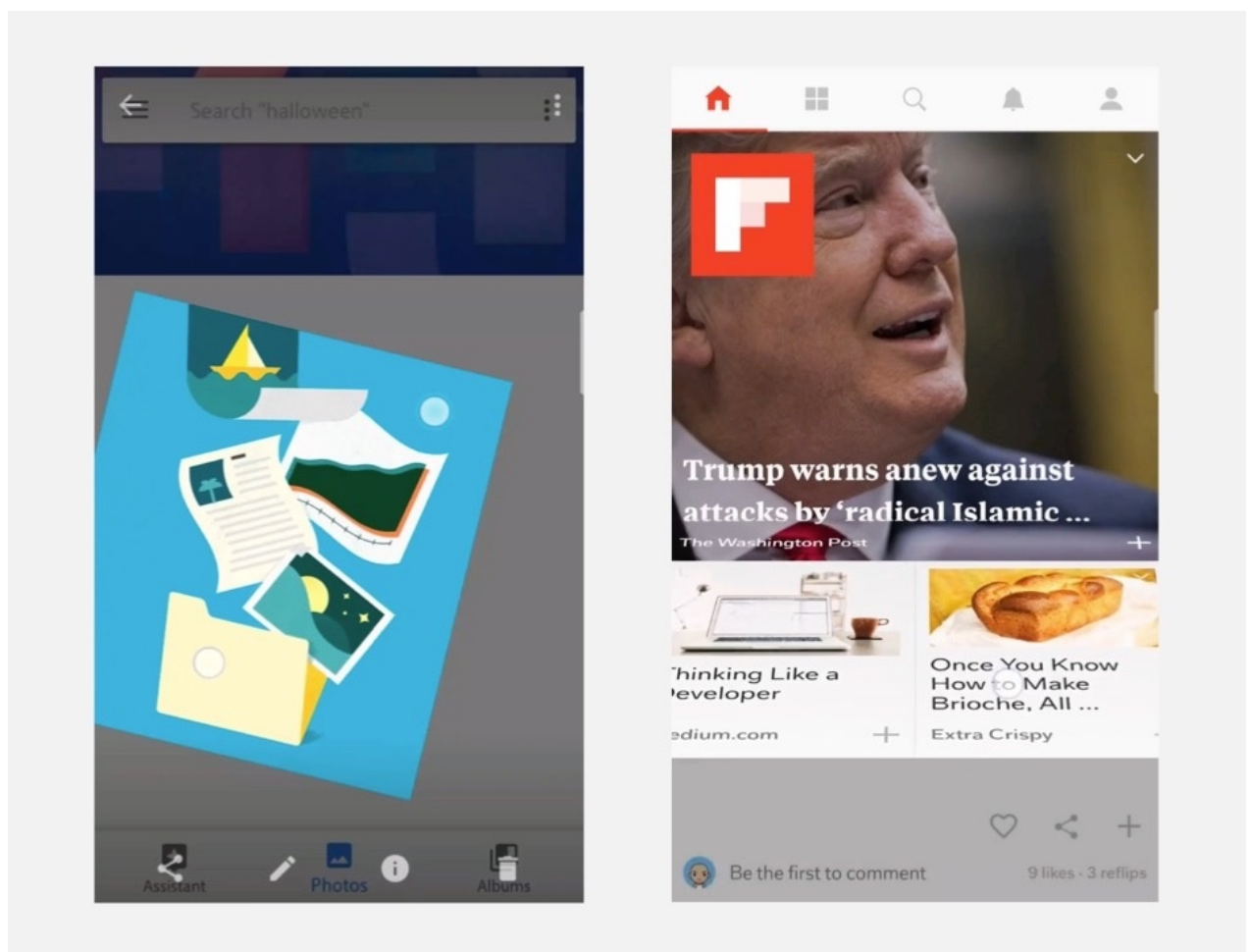
技术上实现不了？  
那 Android 组怎么可以？

3. Canvas 对绘制的辅助——范围裁切和几何变换。

范围裁切：



几何变换：



大多数时候，它们并不会被用到，但一旦用到，通常都是很炫酷的效果。范围裁切和几何变换都是用于辅助的，它们本身并不酷，让它们变酷的是设计师们的想象力与创造力。而你要做的，是把他们的想象力与创造力变成现实。

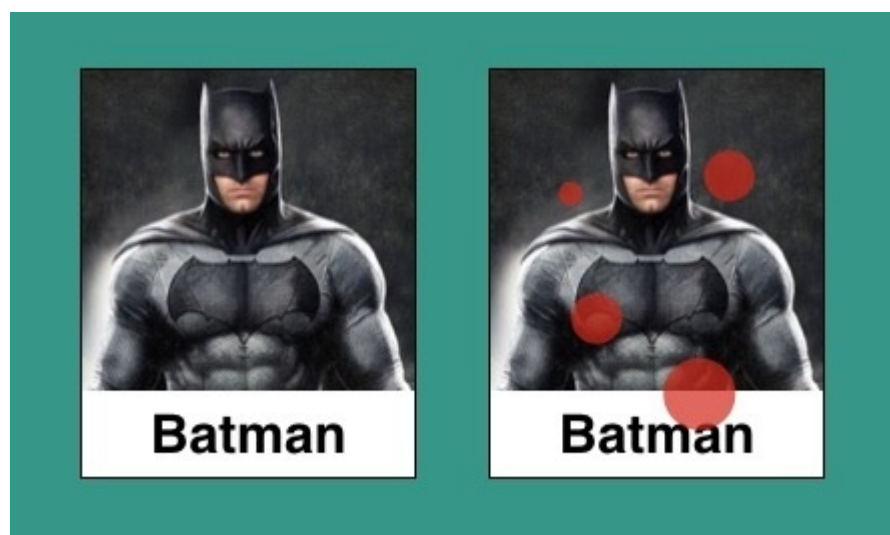


一不留神设计得太炫了  
能实现吗？  
不能实现我再改改



从今以后  
别问我能不能实现  
你就说你要不要

#### 4. 使用不同的绘制方法来控制绘制顺序



控制绘制顺序解决的并不是「做不到」的问题，而是性能问题。同样的一种效果，你不用绘制顺序的控制往往也能做到，但需要用多个 View 甚至是多层 View 才能拼凑出来，因此代价是 UI 的性能；而使用绘制顺序的控制的话，一个 View 就全部搞定了。

自定义绘制的知识，大概就分为上面这四个级别。在你把这四个级别依次掌握了之后，你就是一个自定义绘制的高手了。它们具体的细节，我将分成几篇来讲。今天这篇就是第一篇：Canvas.drawXXX() 系列方法及 Paint 最基本的使用。我要正式开始喽？

# 一切的开始：onDraw()

自定义绘制的上手非常容易：提前创建好 `Paint` 对象，重写 `onDraw()`，把绘制代码写在 `onDraw()` 里面，就是自定义绘制最基本的实现。大概就像这样：

```
Paint paint = new Paint();

@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    // 绘制一个圆
    canvas.drawCircle(300, 300, 200, paint);
}
```

就这么简单。所以关于 `onDraw()` 其实没什么好说的，一个很普通的方法重写，唯一需要注意的是别漏写了 `super.onDraw()`。

## Canvas.drawXXX() 和 Paint 基础

`drawXXX()` 系列方法和 `Paint` 的基础掌握了，就能够应付简单的绘制需求。它们主要包括：

1. `Canvas` 类下的所有 `draw-` 打头的方法，例如 `drawCircle()` `drawBitmap()`。
2. `Paint` 类的几个最常用的方法。具体是：
  - `Paint.setStyle(Style style)` 设置绘制模式
  - `Paint.setColor(int color)` 设置颜色
  - `Paint.setStrokeWidth(float width)` 设置线条宽度
  - `Paint.setTextSize(float textSize)` 设置文字大小
  - `Paint.setAntiAlias(boolean aa)` 设置抗锯齿开关

对于比较习惯于自学的人（我就是这样的人），你看到这里就已经可以去 Google 的官方文档里，打开 [Canvas](#) 和 [Paint](#) 的页面，把上面的这两类方法学习一下，然



后今天的内容就算结束了。当然，这篇文章也可以关掉了。



再见

F\*\*k Off

下面的内容就是展开讲解上面的这两类方法。

## Canvas.drawColor(@ColorInt int color) 颜色填充

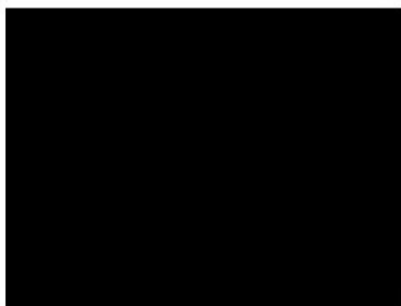
这是最基本的 `drawXXX()` 方法：在整个绘制区域统一涂上指定的颜色。

例如 `drawColor(Color.BLACK)` 会把整个区域染成纯黑色，覆盖掉原有内容；  
`drawColor(Color.parse("#88880000"))` 会在原有的绘制效果上加一层半透明的红色遮罩。

```
drawColor(Color.BLACK); // 纯黑
```



绘制前



绘制后

```
drawColor(Color.parse("#88880000")); // 半透明红色
```



绘制前



绘制后

类似的方法还有 `drawRGB(int r, int g, int b)` 和 `drawARGB(int a, int r, int g, int b)`，它们和 `drawColor(color)` 只是使用方式不同，作用都是一样的。

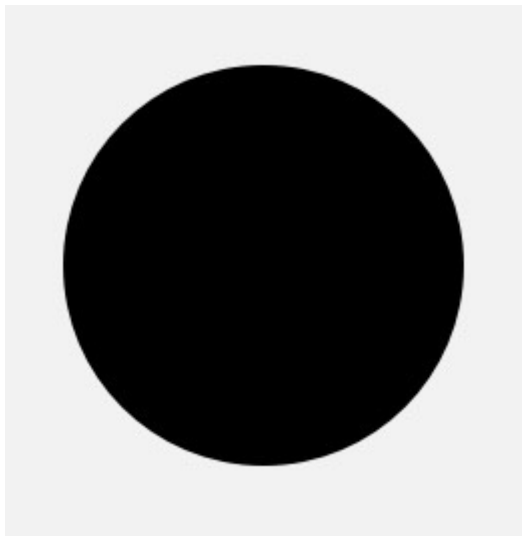
```
canvas.drawRGB(100, 200, 100);  
canvas.drawARGB(100, 100, 200, 100);
```

这类颜色填充方法一般用于在绘制之前设置底色，或者在绘制之后为界面设置半透明蒙版。

## `drawCircle(float centerX, float centerY, float radius, Paint paint)` 画圆

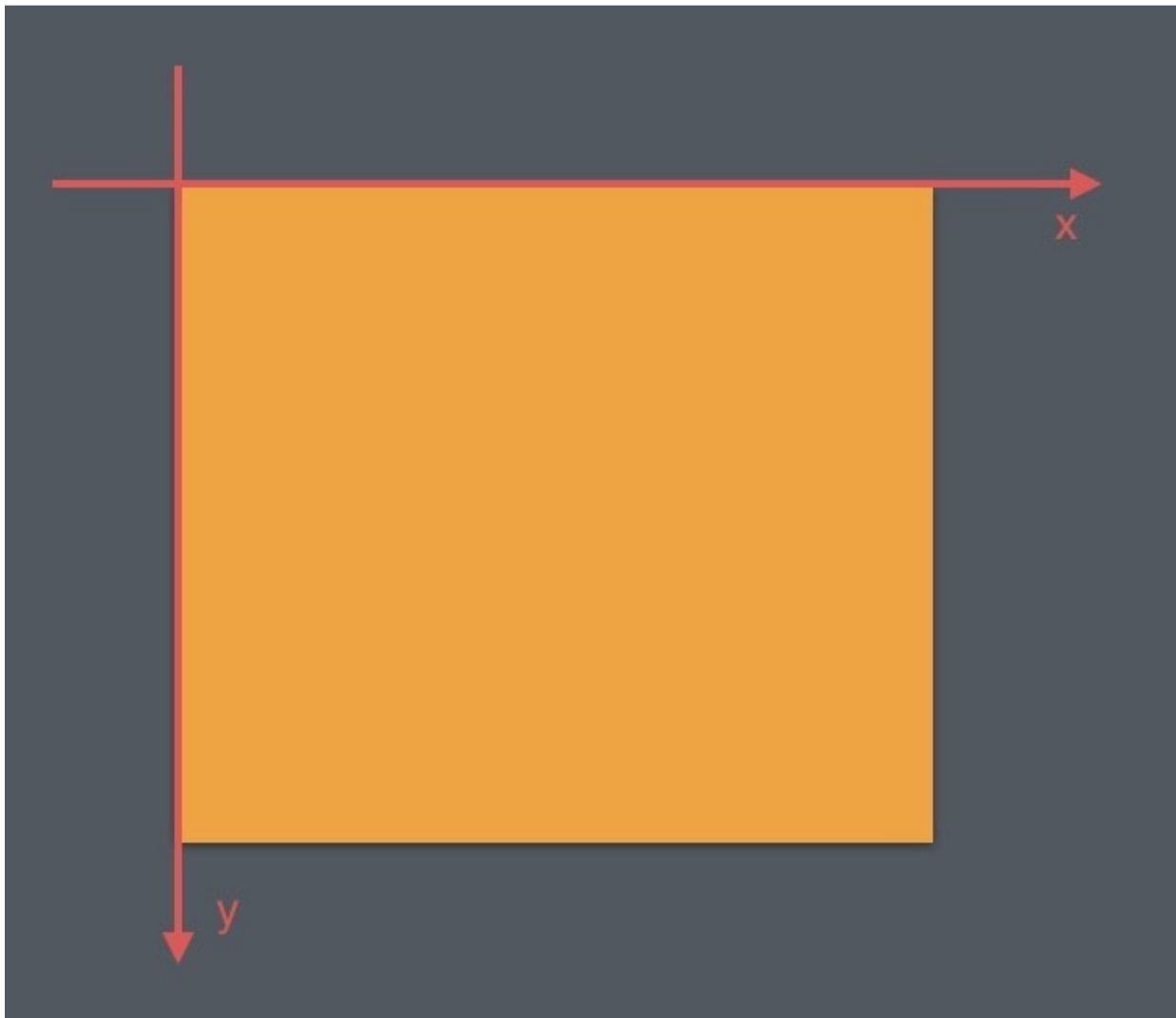
前两个参数 `centerX` `centerY` 是圆心的坐标，第三个参数 `radius` 是圆的半径，单位都是像素，它们共同构成了这个圆的基本信息（即用这几个信息可以构建出一个确定的圆）；第四个参数 `paint` 我在视频里面已经说过了，它提供基本信息之外的所有风格信息，例如颜色、线条粗细、阴影等。

```
canvas.drawRGB(100, 200, 100);  
canvas.drawARGB(100, 100, 200, 100);
```



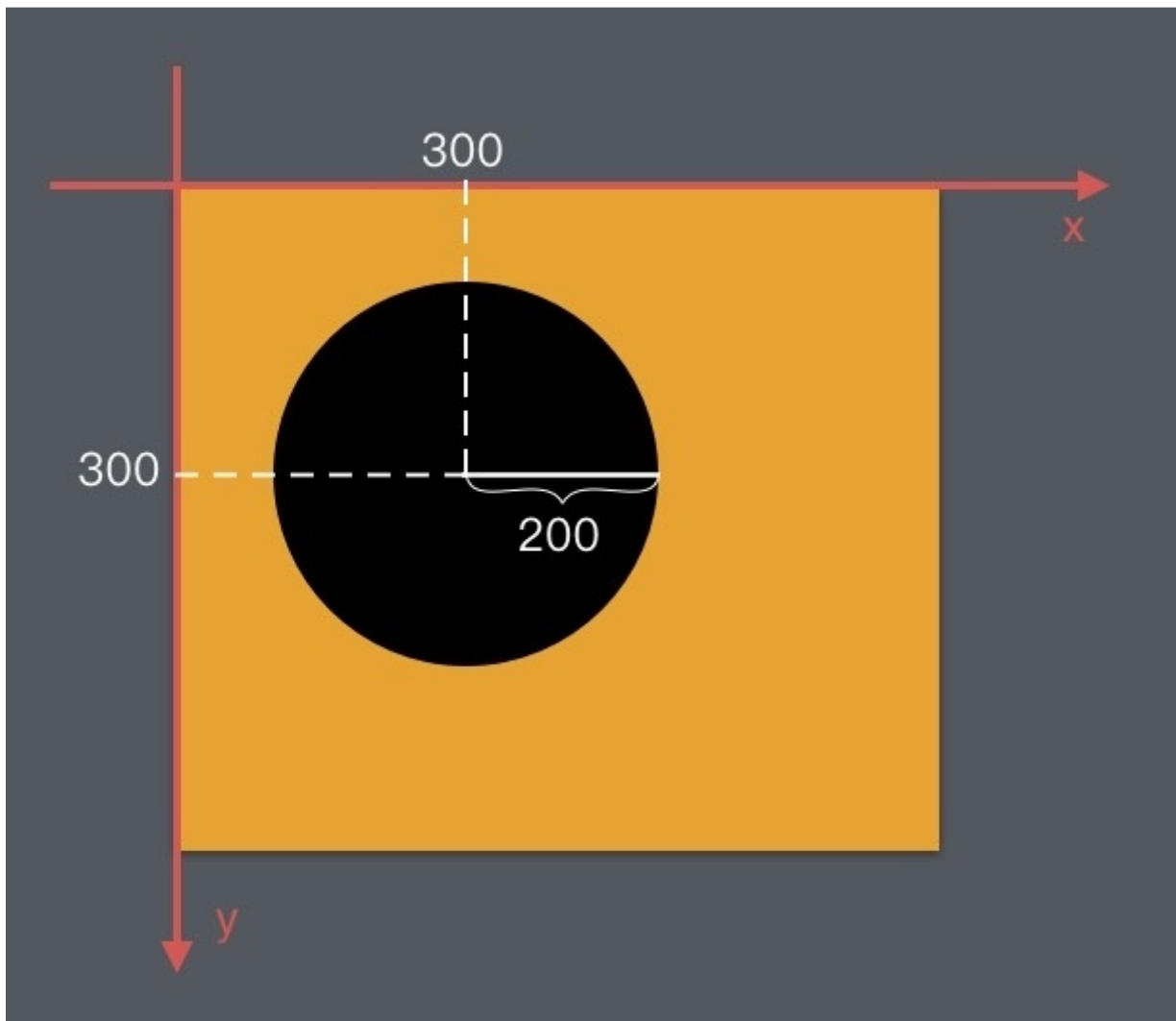
那位说：「你等会儿！先别往后讲，你刚才说圆心的坐标，我想问坐标系在哪儿呢？没坐标系你跟我聊什么坐标啊。」

我想说：问得好（强行插入剧情）。在 Android 里，每个 View 都有一个自己的坐标系，彼此之间是不影响的。这个坐标系的原点是 View 左上角的那个点；水平方向是 x 轴，右正左负；竖直方向是 y 轴，下正上负（注意，是下正上负，不是上正下负，和上学时候学的坐标系方向不一样）。也就是下面这个样子。



所以一个 View 的坐标  $(x, y)$  处，指的就是相对它的左上角那个点的水平方向  $x$  像素、竖直方向  $y$  像素的点。例如， $(300, 300)$  指的就是左上角的点向右 300、向下 300 的位置； $(100, -50)$  指的就是左上角的点向右 100、向上 50 的位置。

也就是说，`canvas.drawCircle(300, 300, 200, paint)` 这行代码绘制出的圆，在 View 中的位置和尺寸应该是这样的：



圆心坐标和半径，这些都是圆的基本信息，也是它的独有信息。什么叫独有信息？就是只有它有，别人没有的信息。你画圆有圆心坐标和半径，画方有吗？画椭圆有吗？这就叫独有信息。独有信息都是直接作为参数写进 `drawXXX()` 方法里的（比如 `drawCircle(centerX, centerY, radius, paint)` 的前三个参数）。

而除此之外，其他的都是公有信息。比如图形的颜色、空心实心这些，你不管是画圆还是画方都有可能用到的，这些信息则是统一放在 `paint` 参数里的。

### 插播一： `Paint.setColor(int color)`

例如，你要画一个红色的圆，并不是写成

`canvas.drawCircle(300, 300, 200, RED, paint)` 这样，而是像下面这样：

```
paint.setColor(Color.RED); // 设置为红色
canvas.drawCircle(300, 300, 200, paint);
```

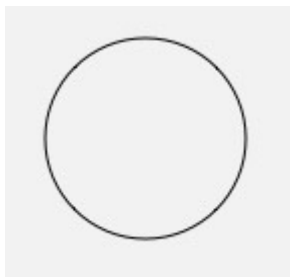


`Paint.setColor(int color)` 是 `Paint` 最常用的方法之一，用来设置绘制内容的颜色。你不止可以用它画红色的圆，也可以用它来画红色的矩形、红色的五角星、红色的文字。

### 插播二： `Paint.setStyle(Paint.Style style)`

而如果你想画的不是实心圆，而是空心圆（或者叫环形），也可以使用 `paint.setStyle(Paint.Style.STROKE)` 来把绘制模式改为画线模式。

```
paint.setStyle(Paint.Style.STROKE); // style 修改为画线模式
canvas.drawCircle(300, 300, 200, paint);
```



`setStyle(Style style)` 这个方法设置的是绘制的 `Style`。`Style` 具体来说有三种：`FILL`，`STROKE` 和 `FILL_AND_STROKE`。`FILL` 是填充模式，`STROKE` 是画线模式（即勾边模式），`FILL_AND_STROKE` 是两种模式一并使用：既画线又填充。它的默认值是 `FILL`，填充模式。

### 插播三： `Paint.setStrokeWidth(float width)`

在 `STROKE` 和 `FILL_AND_STROKE` 下，还可以使用 `paint.setStrokeWidth(float width)` 来设置线条的宽度：

```
paint.setStyle(Paint.Style.STROKE);  
paint.setStrokeWidth(20); // 线条宽度为 20 像素  
canvas.drawCircle(300, 300, 200, paint);
```



#### 插播四：抗锯齿

在绘制的时候，往往需要开启抗锯齿来让图形和文字的边缘更加平滑。开启抗锯齿很简单，只要在 `new Paint()` 的时候加上一个 `ANTI_ALIAS_FLAG` 参数就行：

```
Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

另外，你也可以使用 `Paint.setAntiAlias(boolean aa)` 来动态开关抗锯齿。

抗锯齿的效果如下：



可以看出，没有开启抗锯齿的时候，图形会有毛片现象，啊不，毛边现象。所以一定记得要打开抗锯齿哟！

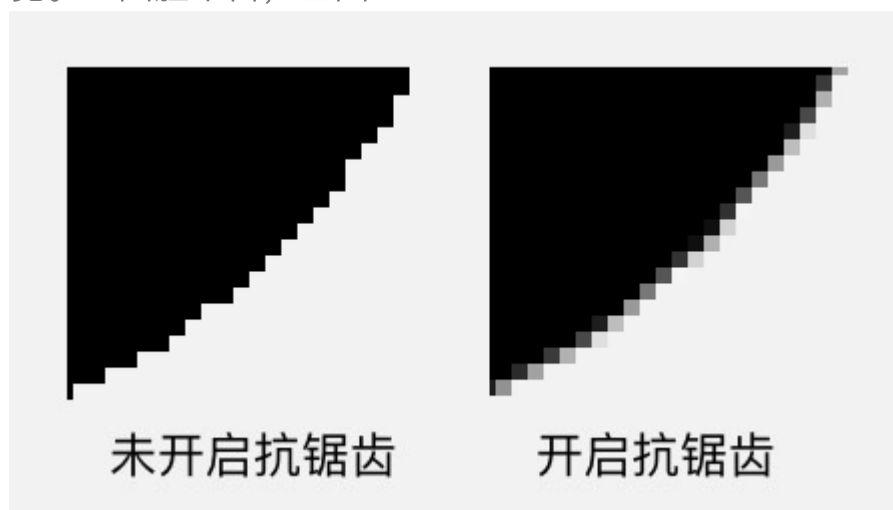
好奇的人可能会问：抗锯齿既然这么有用，为什么不默认开启，或者干脆把这个开关取消，自动让所有绘制都开启抗锯齿？

短答案：因为抗锯齿并不一定适合所有场景。

长答案：所谓的毛边或者锯齿，发生的原因并不是很多人所想象的「绘制太粗糙」「像素计算能力不足」；同样，抗锯齿的原理也并不是选择了更精细的算法来算出了更平滑的图形边缘。

实质上，锯齿现象的发生，只是由于图形分辨率过低，导致人眼察觉出了画面中的像素颗粒而已。换句话说，就算不开启抗锯齿，图形的边缘也已经是最完美的了，而并不是一个粗略计算的粗糙版本。

那么，为什么抗锯齿开启之后的图形边缘会更加平滑呢？因为抗锯齿的原理是：修改图形边缘处的像素颜色，从而让图形在肉眼看来具有更加平滑的感觉。一图胜千言，上图：



上面这个是把前面那两个圆放大后的局部效果。看到没有？未开启抗锯齿的圆，所有像素都是同样的黑色，而开启了抗锯齿的圆，边缘的颜色被略微改变了。这种改变可以让人眼有边缘平滑的感觉，但从某种角度讲，它也造成了图形的颜色失真。

所以，抗锯齿好不好？好，大多数情况下它都应该是开启的；但在极少数的某些时候，你还真的需要把它关闭。「某些时候」是什么时候？到你用到的时候自然就知道了。

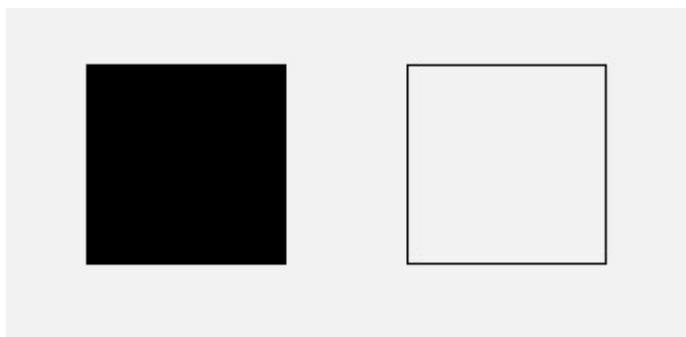
除了圆，Canvas 还可以绘制一些别的简单图形。它们的使用方法和 `drawCircle()` 大同小异，我就只对它们的 API 做简单的介绍，不再做详细的讲解。



## drawRect(float left, float top, float right, float bottom, Paint paint) 画矩形

left, top, right, bottom 是矩形四条边的坐标。

```
paint.setStyle(Style.FILL);  
canvas.drawRect(100, 100, 500, 500, paint);  
  
paint.setStyle(Style.STROKE);  
canvas.drawRect(700, 100, 1100, 500, paint);
```



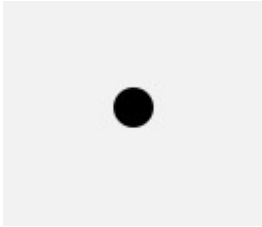
另外，它还有两个重载方法 `drawRect(RectF rect, Paint paint)` 和 `drawRect(Rect rect, Paint paint)`，让你可以直接填写 `RectF` 或 `Rect` 对象来绘制矩形。

## drawPoint(float x, float y, Paint paint) 画点

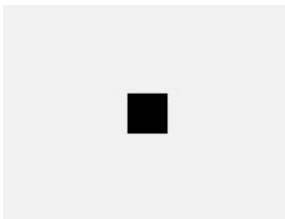
`x` 和 `y` 是点的坐标。点的大小可以通过 `paint.setStrokeWidth(width)` 来设置；点的形状可以通过 `paint.setStrokeCap(cap)` 来设置：`ROUND` 画出来是圆形的点，`SQUARE` 或 `BUTT` 画出来是方形的点。（点还有形状？是的，反正 Google 是这么说的，你要问问 Google 去，我也很懵逼。）

注：`Paint.setStrokeCap(cap)` 可以设置点的形状，但这个方法并不是专门用来设置点的形状的，而是一个设置线条端点形状的方法。端点有圆头（`ROUND`）、平头（`BUTT`）和方头（`SQUARE`）三种，具体会在下一节里面讲。

```
paint.setStrokeWidth(20);  
paint.setStrokeCap(Paint.Cap.ROUND);  
canvas.drawPoint(50, 50, paint);
```



```
paint.setStrokeWidth(20);  
paint.setStrokeCap(Paint.Cap.SQUARE);  
canvas.drawPoint(50, 50, paint);
```



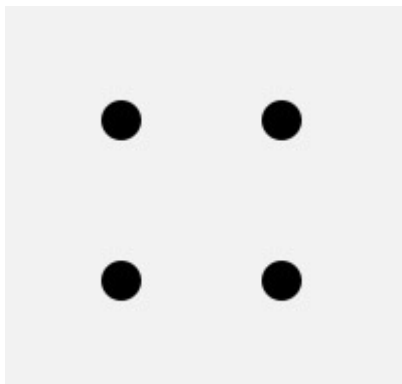
好像有点像 FILL 模式下的 `drawCircle()` 和 `drawRect()`？事实上确实是这样的，它们和 `drawPoint()` 的绘制效果没有区别。各位在使用的时候按个人习惯和实际场景来吧，哪个方便和顺手用哪个。

## **`drawPoints(float[] pts, int offset, int count, Paint paint) / drawPoints(float[] pts, Paint paint)` 画点（批量）**

同样是画点，它和 `drawPoint()` 的区别是可以画多个点。`pts` 这个数组是点的坐标，每两个成一对；`offset` 表示跳过数组的前几个数再开始记坐标；`count` 表示一共要绘制几个点。说这么多你可能越读越晕，你还是自己试试吧，这是个看着复杂用着简单的方法。

```
float[] points = {0, 0, 50, 50, 50, 100, 100, 50, 100, 100, 150, 50  
// 绘制四个点: (50, 50) (50, 100) (100, 50) (100, 100)}
```

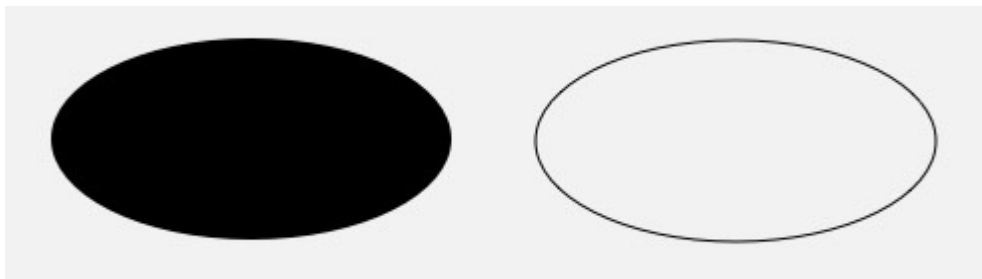
```
canvas.drawPoints(points, 2 /* 跳过两个数, 即前两个 0 */,  
                  8 /* 一共绘制 8 个数 (4 个点) */, paint);
```



## drawOval(float left, float top, float right, float bottom, Paint paint) 画椭圆

只能绘制横着的或者竖着的椭圆，不能绘制斜的（斜的倒是也可以，但不是直接使用 `drawOval()`，而是配合几何变换，后面会讲到）。`left`, `top`, `right`, `bottom` 是这个椭圆的左、上、右、下四个边界点的坐标。

```
paint.setStyle(Style.FILL);  
canvas.drawOval(50, 50, 350, 200, paint);  
  
paint.setStyle(Style.STROKE);  
canvas.drawOval(400, 50, 700, 200, paint);
```

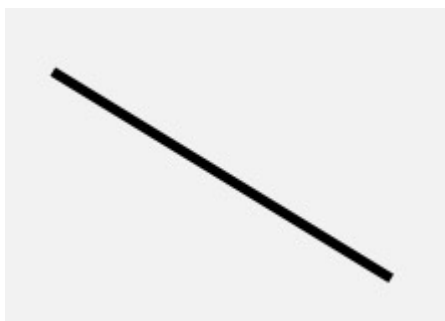


另外，它还有一个重载方法 `drawOval(RectF rect, Paint paint)`，让你可以直接填写 `RectF` 来绘制椭圆。

## drawLine(float startX, float startY, float stopX, float stopY, Paint paint) 画线

startX, startY, stopX, stopY 分别是线的起点和终点坐标。

```
canvas.drawLine(200, 200, 800, 500, paint);
```

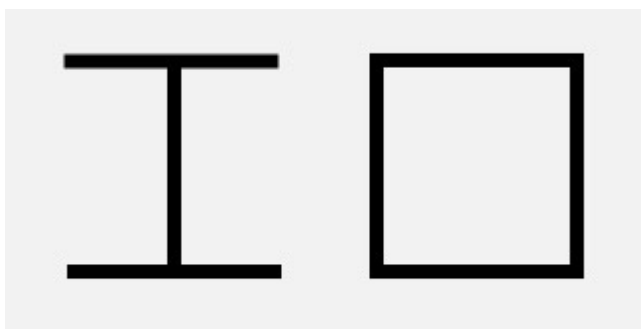


由于直线不是封闭图形，所以 `setStyle(style)` 对直线没有影响。

## **drawLines(float[] pts, int offset, int count, Paint paint) / drawLines(float[] pts, Paint paint) 画线（批量）**

`drawLines()` 是 `drawLine()` 的复数版。

```
float[] points = {20, 20, 120, 20, 70, 20, 70, 120, 20, 120, 120, 120, 20, 20};
canvas.drawLines(points, paint);
```

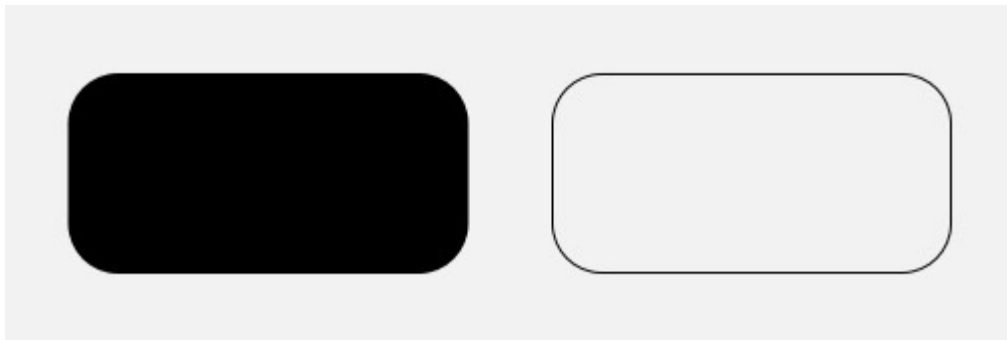


咦，不小心打出两个汉字。——是汉字吧？

## **drawRoundRect(float left, float top, float right, float bottom, float rx, float ry, Paint paint) 画圆角矩形**

left, top, right, bottom 是四条边的坐标, rx 和 ry 是圆角的横向半径和纵向半径。

```
canvas.drawRoundRect(100, 100, 500, 300, 50, 50, paint);
```



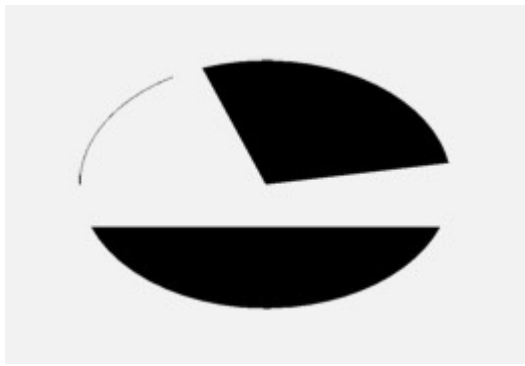
另外, 它还有一个重载方法

`drawRoundRect(RectF rect, float rx, float ry, Paint paint)`, 让你可以直接填写 `RectF` 来绘制圆角矩形。

## **`drawArc(float left, float top, float right, float bottom, float startAngle, float sweepAngle, boolean useCenter, Paint paint)` 绘制弧形或扇形**

`drawArc()` 是使用一个椭圆来描述弧形的。`left, top, right, bottom` 描述的是这个弧形所在的椭圆; `startAngle` 是弧形的起始角度 (x 轴的正向, 即正右的方向, 是 0 度的位置; 顺时针为正角度, 逆时针为负角度), `sweepAngle` 是弧形划过的角度; `useCenter` 表示是否连接到圆心, 如果不连接到圆心, 就是弧形, 如果连接到圆心, 就是扇形。

```
paint.setStyle(Paint.Style.FILL); // 填充模式
canvas.drawArc(200, 100, 800, 500, -110, 100, true, paint); // 绘制扇形
canvas.drawArc(200, 100, 800, 500, 20, 140, false, paint); // 绘制弧形
paint.setStyle(Paint.Style.STROKE); // 画线模式
canvas.drawArc(200, 100, 800, 500, 180, 60, false, paint); // 绘制不
```



到此为止，以上就是 Canvas 所有的简单图形的绘制。除了简单图形的绘制，Canvas 还可以使用 `drawPath(Path path)` 来绘制自定义图形。

## `drawPath(Path path, Paint paint)` 画自定义图形

这个方法有点复杂，需要展开说一下。

前面的这些方法，都是绘制某个给定的图形，而 `drawPath()` 可以绘制自定义图形。当你要绘制的图形比较特殊，使用前面的那些方法做不到的时候，就可以使用 `drawPath()` 来绘制。



`drawPath(path)` 这个方法是通过描述路径的方式来绘制图形的，它的 `path` 参数就是用来描述图形路径的对象。`path` 的类型是 `Path`，使用方法大概像下面这样：

```
public class PathView extends View {  
  
    Paint paint = new Paint();  
    Path path = new Path(); // 初始化 Path 对象
```

```

.....

{
    // 使用 path 对图形进行描述（这段描述代码不必看懂）
    path.addArc(200, 200, 400, 400, -225, 225);
    path.arcTo(400, 200, 600, 400, -180, 225, false);
    path.lineTo(400, 542);
}

@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    canvas.drawPath(path, paint); // 绘制出 path 描述的图形（心形），
}
}

```



Path 可以描述直线、二次曲线、三次曲线、圆、椭圆、弧形、矩形、圆角矩形。把这些图形结合起来，就可以描述出很多复杂的图形。下面我就说一下具体的怎么把这些图形描述出来。

Path 有两类方法，一类是直接描述路径的，另一类是辅助的设置或计算。

## Path 方法第一类：直接描述路径。

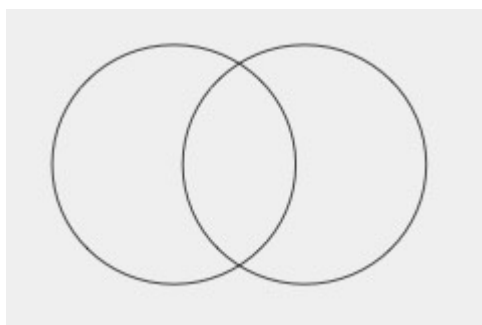
这一类方法还可以细分为两组：添加子图形和画线（直线或曲线）

**第一组：** `addXxx()` —— 添加子图形

## addCircle(float x, float y, float radius, Direction dir) 添加圆

`x`, `y`, `radius` 这三个参数是圆的基本信息，最后一个参数 `dir` 是画圆的路径的方向。

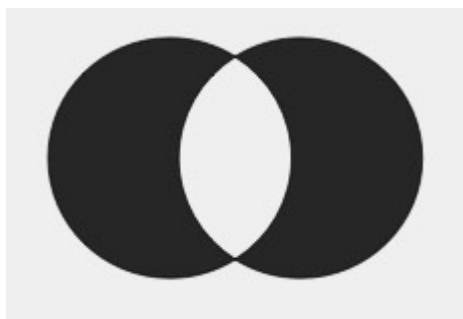
路径方向有两种：顺时针 (cw clockwise) 和逆时针 (ccw counter-clockwise)。对于普通情况，这个参数填 cw 还是填 ccw 没有影响。它只是在需要填充图形 (Paint.Style 为 FILL 或 FILL\_AND\_STROKE)，并且图形出现自相交时，用于判断填充范围的。比如下面这个图形：



是应该填充成这样呢：



还是应该填充成这样呢：





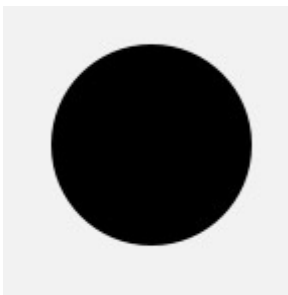


还有这种操作？

想用哪种方式来填充，都可以由你来决定。具体怎么做，下面在讲 `Path.setFillType()` 的时候我会详细介绍，而在这里你可以先忽略 `dir` 这个参数。

在用 `addCircle()` 为 `Path` 中新增一个圆之后，调用 `canvas.drawPath(path, paint)`，就能画一个圆出来。就像这样：

```
path.addCircle(300, 300, 200, Path.Direction.CW);  
  
...  
  
canvas.drawPath(path, paint);
```



好像在哪见过

可以看出, `path.AddCircle(x, y, radius, dir) + canvas.drawPath(path, paint)` 这种写法, 和直接使用 `canvas.drawCircle(x, y, radius, paint)` 的效果是一样的, 区别只是它的写法更复杂。所以如果只画一个圆, 没必要用 `Path`, 直接用 `drawCircle()` 就行了。`drawPath()` 一般是在绘制组合图形时才会用到的。

其他的 `Path.add-()` 方法和这类似, 例如:

**`addOval(float left, float top, float right, float bottom, Direction dir) / addOval(RectF oval, Direction dir)` 添加椭圆**

**`addRect(float left, float top, float right, float bottom, Direction dir) / addRect(RectF rect, Direction dir)` 添加矩形**

**`addRoundRect(RectF rect, float rx, float ry, Direction dir) / addRoundRect(float left, float top, float right, float bottom, float rx, float ry, Direction dir) / addRoundRect(RectF rect, float[] radii, Direction dir) / addRoundRect(float left, float top, float right, float bottom, float[] radii, Direction dir)` 添加圆角矩形**

**`addPath(Path path)` 添加另一个 `Path`**

上面这几个方法和 `addCircle()` 的使用都差不多, 不再做过多介绍。

## 第二组: `xxxTo()` ——画线 (直线或曲线)

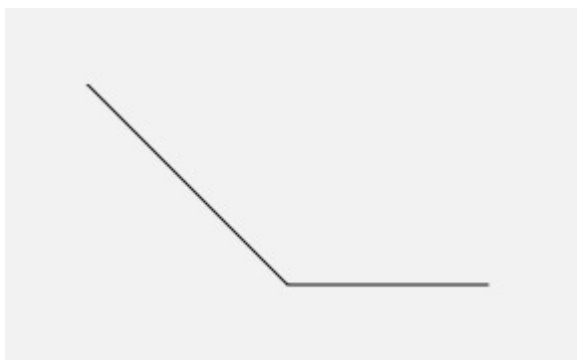
这一组和第一组 `addXxx()` 方法的区别在于, 第一组是添加的完整封闭图形 (除了 `addPath()`), 而这一组添加的只是一条线。

### **`lineTo(float x, float y) / rLineTo(float x, float y)` 画直线**

从当前位置向目标位置画一条直线, `x` 和 `y` 是目标位置的坐标。这两个方法的区别是, `lineTo(x, y)` 的参数是绝对坐标, 而 `rLineTo(x, y)` 的参数是相对当前位置的相对坐标 (前缀 `r` 指的就是 `relatively` 「相对地」)。

**当前位置:** 所谓当前位置, 即最后一次调用画 `Path` 的方法的终点位置。初始值为原点 (0, 0)。

```
paint.setStyle(Style.STROKE);  
path.lineTo(100, 100); // 由当前位置 (0, 0) 向 (100, 100) 画一条直线  
path.rLineTo(100, 0); // 由当前位置 (100, 100) 向正右方 100 像素的位置画
```



**quadTo(float x1, float y1, float x2, float y2) / rQuadTo(float dx1, float dy1, float dx2, float dy2) 画二次贝塞尔曲线**

这条二次贝塞尔曲线的起点就是当前位置，而参数中的  $x_1$ ,  $y_1$  和  $x_2$ ,  $y_2$  则分别是控制点和终点的坐标。和 `rLineTo(x, y)` 同

理，`rQuadTo(dx1, dy1, dx2, dy2)` 的参数也是相对坐标

**贝塞尔曲线：**贝塞尔曲线是几何上的一种曲线。它通过起点、控制点和终点来描述一条曲线，主要用于计算机图形学。概念总是说着容易听着难，总之使用它可以绘制很多圆润又好看的图形，但要把它熟练掌握、灵活使用却是不容易的。不过还好的是，一般情况下，贝塞尔曲线并没有什么用处，只在少数场景下绘制一些特殊图形的时候才会用到，所以如果你还没掌握自定义绘制，可以先把贝塞尔曲线放一放，稍后再学也完全没问题。至于怎么学，贝塞尔曲线的知识网上一搜一大把，我这里就不讲了。

**cubicTo(float x1, float y1, float x2, float y2, float x3, float y3) / rCubicTo(float x1, float y1, float x2, float y2, float x3, float y3) 画三次贝塞尔曲线**

和上面这个 `quadTo()` `rQuadTo()` 的二次贝塞尔曲线同理，`cubicTo()` 和 `rCubicTo()` 是三次贝塞尔曲线，不再解释。

**moveTo(float x, float y) / rMoveTo(float x, float y) 移动到目标位置**

不论是直线还是贝塞尔曲线，都是以当前位置作为起点，而不能指定起点。但你可以通过 `moveTo(x, y)` 或 `rMoveTo()` 来改变当前位置，从而间接地设置这些方法的起点。

```
paint.setStyle(Paint.Style.FILL); // 填充模式
canvas.drawArc(200, 100, 800, 500, -110, 100, true, paint); // 绘制扇形
canvas.drawArc(200, 100, 800, 500, 20, 140, false, paint); // 绘制弧线
paint.setStyle(Paint.Style.STROKE); // 画线模式
canvas.drawArc(200, 100, 800, 500, 180, 60, false, paint); // 绘制不
```



`moveTo(x, y)` 虽然不添加图形，但它会设置图形的起点，所以它是非常重要的一个辅助方法。

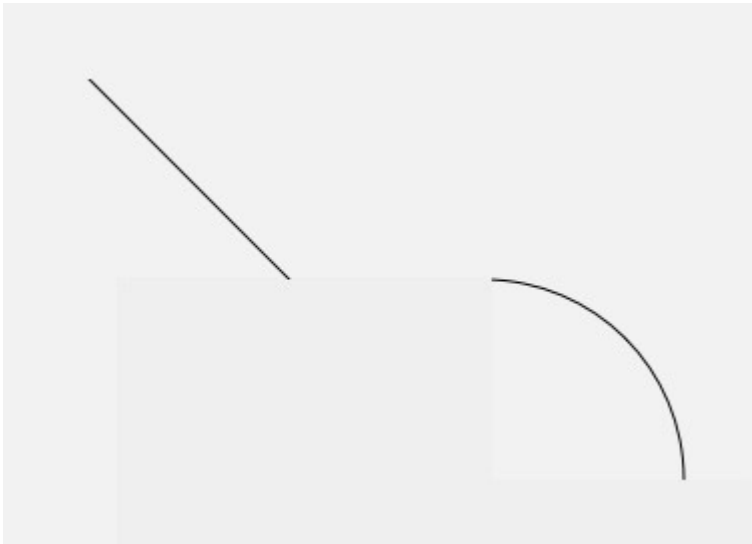
另外，第二组还有两个特殊的方法：`arcTo()` 和 `addArc()`。它们也是用来画线的，但并不使用当前位置作为弧线的起点。

**`arcTo(RectF oval, float startAngle, float sweepAngle, boolean forceMoveTo)` / `arcTo(float left, float top, float right, float bottom, float startAngle, float sweepAngle, boolean forceMoveTo)` / `arcTo(RectF oval, float startAngle, float sweepAngle)` 画弧形**

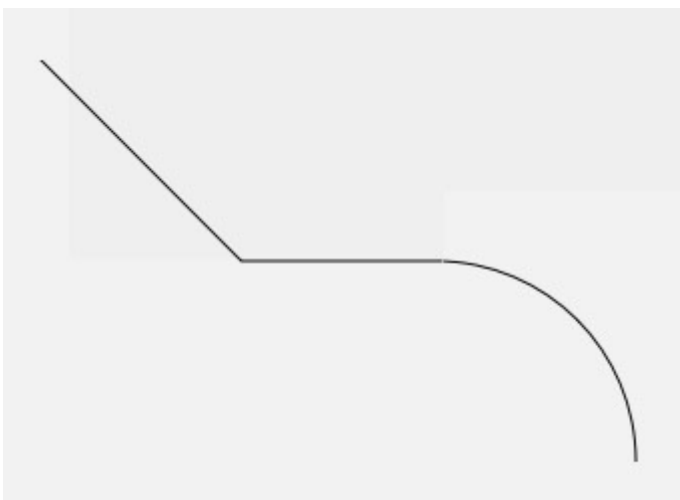
这个方法和 `Canvas.drawArc()` 比起来，少了一个参数 `useCenter`，而多了一个参数 `forceMoveTo`。

少了 `useCenter`，是因为 `arcTo()` 只用来画弧形而不画扇形，所以不再需要 `useCenter` 参数；而多出来的这个 `forceMoveTo` 参数的意思是，绘制是要「抬一下笔移动过去」，还是「直接拖着笔过去」，区别在于是否留下移动的痕迹。

```
paint.setStyle(Style.STROKE);  
path.lineTo(100, 100);  
path.arcTo(100, 100, 300, 300, -90, 90, true); // 强制移动到弧形起点 (300, 300)
```



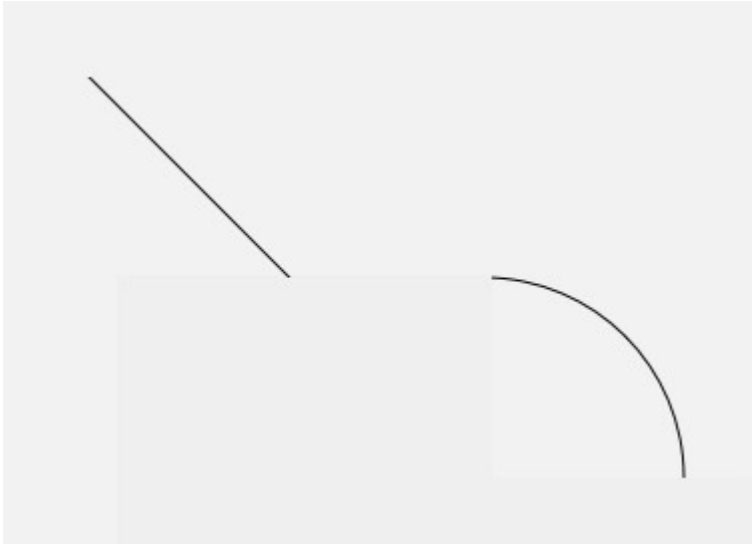
```
path.addCircle(300, 300, 200, Path.Direction.CW);  
  
...  
  
canvas.drawPath(path, paint);
```



**addArc(float left, float top, float right, float bottom, float startAngle, float sweepAngle) / addArc(RectF oval, float startAngle, float sweepAngle)**

又是一个弧形的方法。一个叫 `arcTo`，一个叫 `addArc()`，都是弧形，区别在哪里？其实很简单：`addArc()` 只是一个直接使用了 `forceMoveTo = true` 的简化版 `arcTo()`。

```
paint.setStyle(Style.STROKE);  
path.lineTo(100, 100);  
path.addArc(100, 100, 300, 300, -90, 90);
```



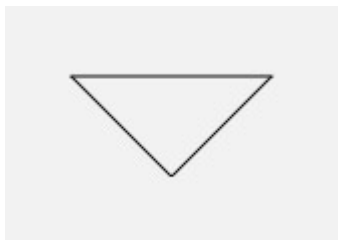
## `close()` 封闭当前子图形

它的作用是把当前的子图形封闭，即由当前位置向当前子图形的起点绘制一条直线。

```
paint.setStyle(Style.STROKE);  
path.moveTo(100, 100);  
path.lineTo(200, 100);  
path.lineTo(150, 150);  
// 子图形未封闭
```



```
paint.setStyle(Style.STROKE);
path.moveTo(100, 100);
path.lineTo(200, 100);
path.lineTo(150, 150);
path.close(); // 使用 close() 封闭子图形。等价于 path.lineTo(100, 100)
```

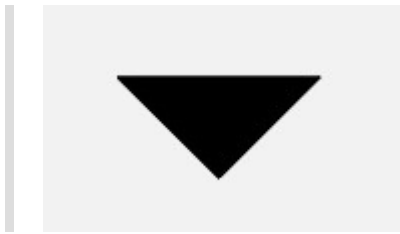


`close()` 和 `lineTo(起点坐标)` 是完全等价的。

「子图形」：官方文档里叫做 `contour` 。但由于在这个场景下我找不到这个词合适的中文翻译（直译的话叫做「轮廓」），所以我换了个便于中国人理解的词：「子图形」。前面说到，第一组方法是「添加子图形」，所谓「子图形」，指的就是一次不间断的连线。一个 `Path` 可以包含多个子图形。当使用第一组方法，即 `addCircle()` `addRect()` 等方法的时候，每一次方法调用都是新增了一个独立的子图形；而如果使用第二组方法，即 `lineTo()` `arcTo()` 等方法的时候，则是每一次断线（即每一次「抬笔」），都标志着一个子图形的结束，以及一个新的子图形的开始。

另外，不是所有的子图形都需要使用 `close()` 来封闭。当需要填充图形时（即 `Paint.Style` 为 `FILL` 或 `FILL_AND_STROKE`），`Path` 会自动封闭子图形。

```
java
paint.setStyle(Style.FILL);
path.moveTo(100, 100);
path.lineTo(200, 100);
path.lineTo(150, 150);
// 这里只绘制了两条边，但由于 style 是 FILL，所以绘制时会自动封口
```



以上就是 `Path` 的第一类方法：直接描述路径的。

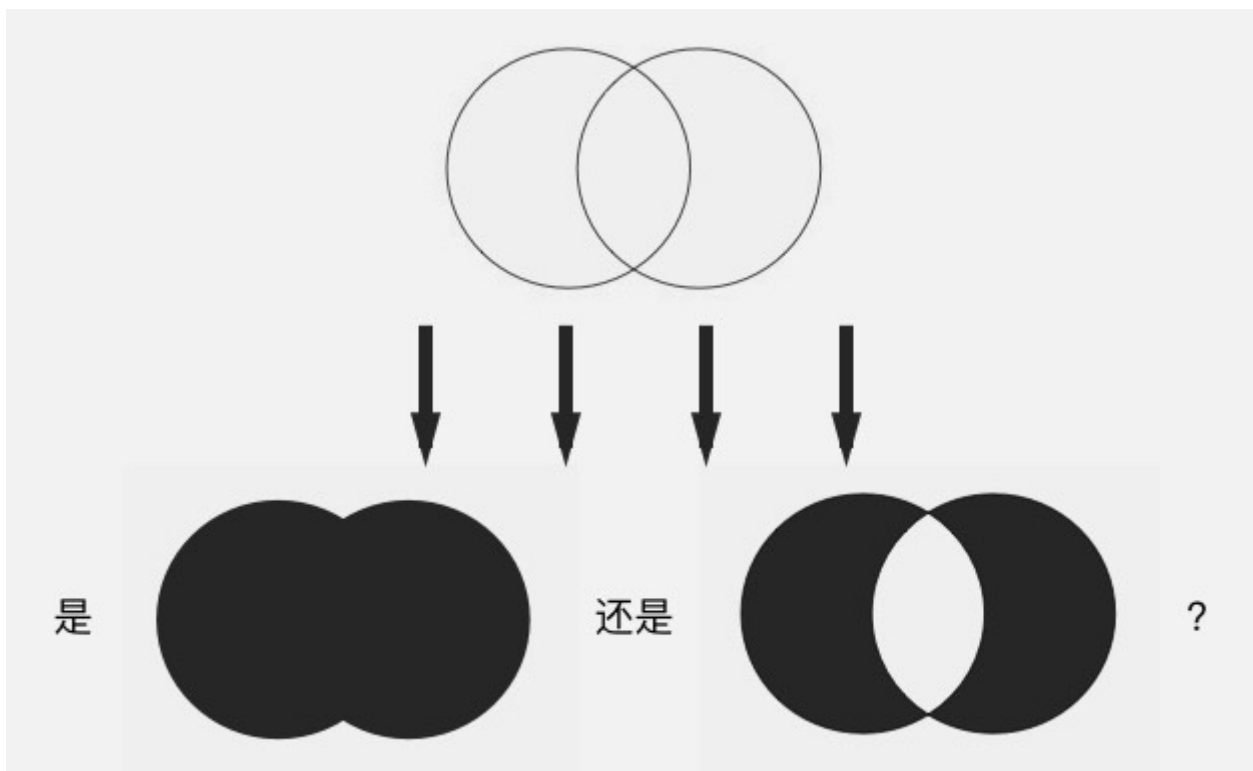
## Path 方法第二类：辅助的设置或计算

这类方法的使用场景比较少，我在这里就不多讲了，只讲其中一个方法：

`setFillType(FillType fillType)`。

### `Path.setFillType(Path.FillType ft)` 设置填充方式

前面在说 `dir` 参数的时候提到，`Path.setFillType(fillType)` 是用来设置图形自相交时的填充算法的：



方法中填入不同的 `FillType` 值，就会有不同的填充效果。`FillType` 的取值有四个：

- `EVEN_ODD`

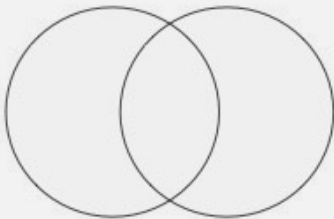


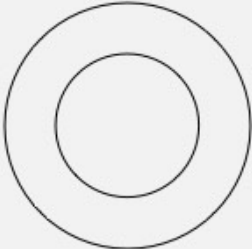


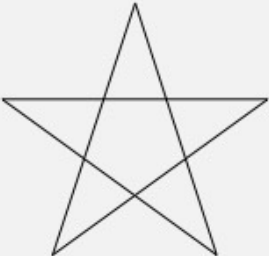




- WINDING （默认值）
- INVERSE\_EVEN\_ODD
- INVERSE\_WINDING

其中后面的两个带有 `INVERSE_` 前缀的，只是前两个的反色版本，所以只要把前两个，即 `EVEN_ODD` 和 `WINDING`，搞明白就可以了。

`EVEN_ODD` 和 `WINDING` 的原理有点复杂，直接讲出来的话信息量太大，所以我先给一个简单粗暴版的总结，你感受一下：`WINDING` 是「全填充」，而 `EVEN_ODD` 是「交叉填充」：

WINDING 和 EVEN\_ODD 效果展示（简单粗暴版）

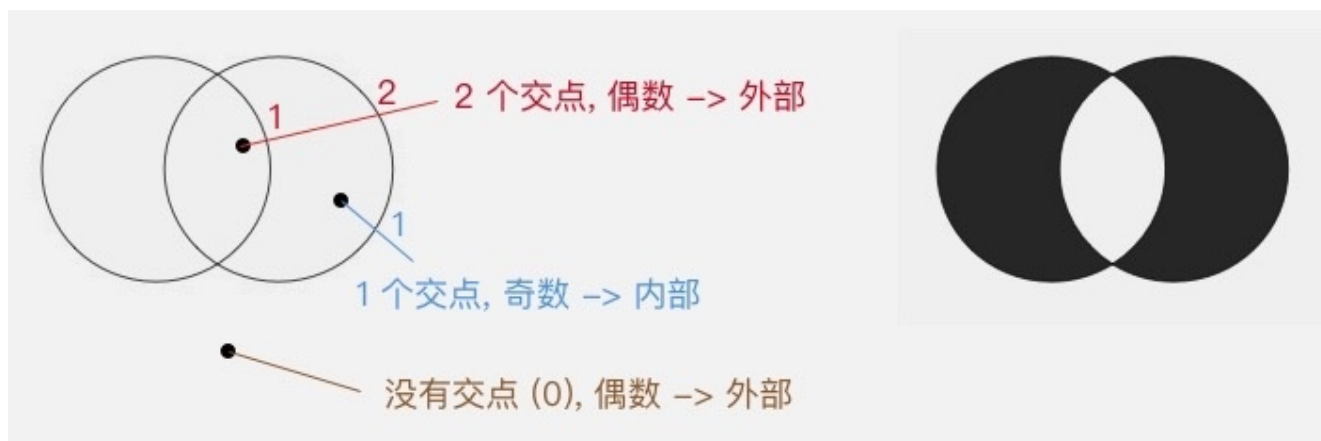
原图形	WINDING	EVEN_ODD
		
		
		

之所以叫「简单粗暴版」，是因为这些只是通常情形下的效果；而如果要准确了解它们在所有情况下的效果，就得先知道它们的原理，即它们的具体算法。

# EVEN\_ODD 和 WINDING 的原理

## EVEN\_ODD

即 even-odd rule（奇偶原则）：对于平面中的任意一点，向任意方向射出一条射线，这条射线和图形相交的次数（相交才算，相切不算哦）如果是奇数，则这个点被认为在图形内部，是要被涂色的区域；如果是偶数，则这个点被认为在图形外部，是不被涂色的区域。还以左右相交的双圆为例：

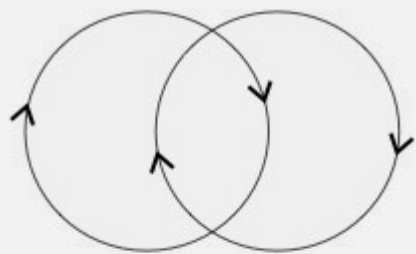


射线的方向无所谓，同一个点射向任何方向的射线，结果都是一样的，不信你可以试试。

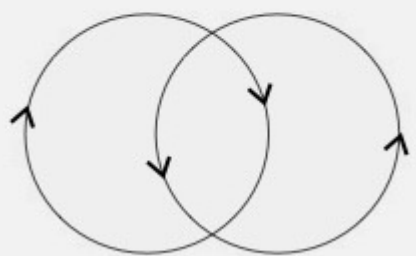
从上图可以看出，射线每穿过图形中的一条线，内外状态就发生一次切换，这就是为什么 EVEN\_ODD 是一个「交叉填充」的模式。

## WINDING

即 non-zero winding rule（非零环绕数原则）：首先，它需要你图形中的所有线条都是有绘制方向的：

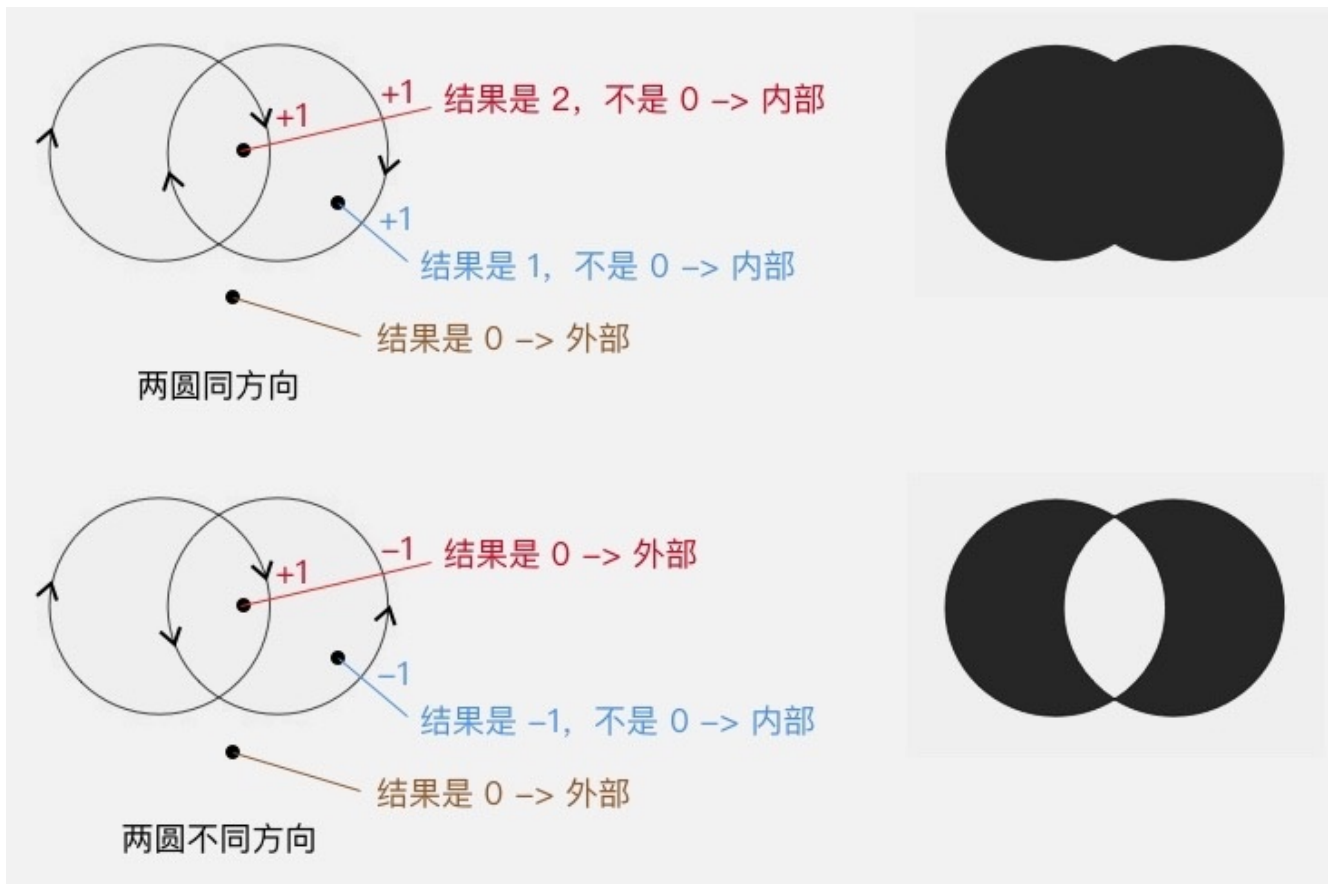


两圆同方向



两圆不同方向

然后，同样是从平面中的点向任意方向射出一条射线，但计算规则不一样：以 0 为初始值，对于射线和图形的所有交点，遇到每个顺时针的交点（图形从射线的左边向右穿过）把结果加 1，遇到每个逆时针的交点（图形从射线的右边向左穿过）把结果减 1，最终把所有的交点都算上，得到的结果如果不是 0，则认为这个点在图形内部，是要被涂色的区域；如果是 0，则认为这个点在图形外部，是不被涂色的区域。



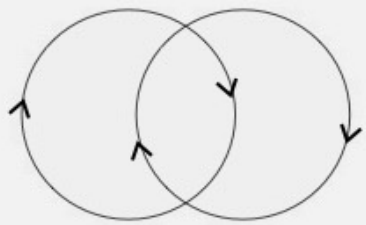
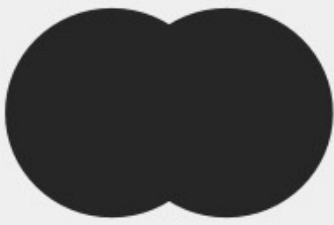
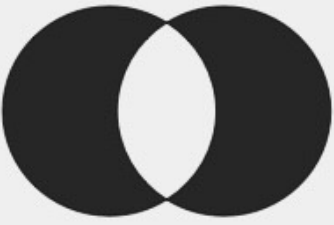
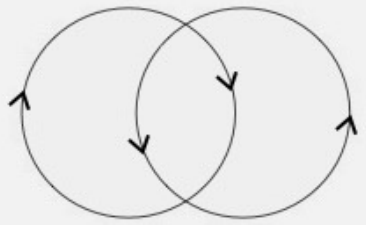
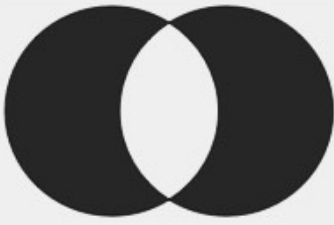
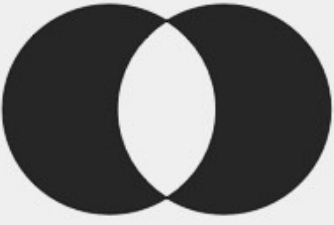
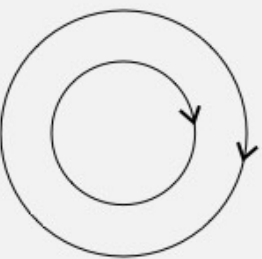


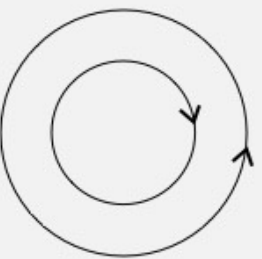


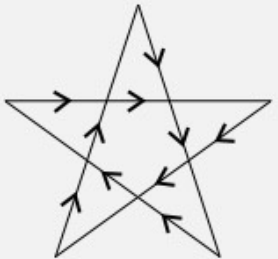


和 `EVEN_ODD` 相同，射线的方向并不影响结果。

所以，我前面的那个「简单粗暴」的总结，对于 `WINDING` 来说并不完全正确：如果你所有的图形都用相同的方向来绘制，那么 `WINDING` 确实是一个「全填充」的规则；但如果使用不同的方向来绘制图形，结果就不一样了。

图形的方向：对于添加子图形类方法（如 `Path.addCircle()` `Path.addRect()`）的方向，由方法的 `dir` 参数来控制，这个在前面已经讲过了；而对于画线类的方法（如 `Path.lineTo()` `Path.arcTo()`）就更简单了，线的方向就是图形的方向。

所以，完整版的 `EVEN_ODD` 和 `WINDING` 的效果应该是这样的：

# WINDING 和 EVEN\_ODD 效果展示

原图形	WINDING	EVEN_ODD
		
		
		
		
		

而 `INVERSE_EVEN_ODD` 和 `INVERSE_WINDING`，只是把这两种效果进行反转而已，你懂了 `EVEN_ODD` 和 `WINDING`，自然也就懂 `INVERSE_EVEN_ODD` 和 `INVERSE_WINDING` 了，我就不讲了。

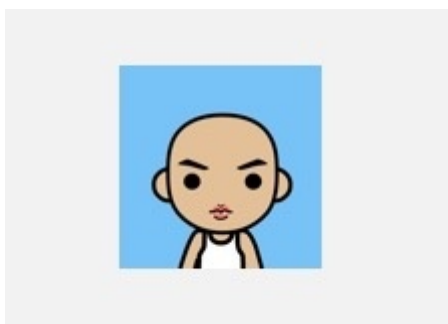
好，花了好长的篇幅来讲 `drawPath(path)` 和 `Path`，终于讲完了。同时，`Canvas` 对图形的绘制也就讲完了。图形简单时，使用 `drawCircle()` `drawRect()` 等方法来直接绘制；图形复杂时，使用 `drawPath()` 来绘制自定义图形。

除此之外，`Canvas` 还可以绘制 `Bitmap` 和文字。

## `drawBitmap(Bitmap bitmap, float left, float top, Paint paint)` 画 `Bitmap`

绘制 `Bitmap` 对象，也就是把这个 `Bitmap` 中的像素内容贴过来。其中 `left` 和 `top` 是要把 `bitmap` 绘制到的位置坐标。它的使用非常简单。

```
paint.setStyle(Style.STROKE);
path.moveTo(100, 100);
path.lineTo(200, 100);
path.lineTo(150, 150);
// 子图形未封闭
```



它的重载方法：

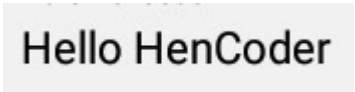
```
drawBitmap(Bitmap bitmap, Rect src, RectF dst, Paint paint) /
drawBitmap(Bitmap bitmap, Rect src, Rect dst, Paint paint) /
drawBitmap(Bitmap bitmap, Matrix matrix, Paint paint)
```

`drawBitmap` 还有一个兄弟方法 `drawBitmapMesh()`，可以绘制具有网格拉伸效果的 `Bitmap`。`drawBitmapMesh()` 的使用场景较少，所以不讲了，如果有兴趣你可以自己研究一下。

## **`drawText(String text, float x, float y, Paint paint)` 绘制文字**

界面里所有的显示内容，都是绘制出来的，包括文字。`drawText()` 这个方法就是用来绘制文字的。参数 `text` 是用来绘制的字符串，`x` 和 `y` 是绘制的起点坐标。

```
canvas.drawText(text, 200, 100, paint);
```



Hello HenCoder

### **插播五： `Paint.setTextSize(float textSize)`**

通过 `Paint.setTextSize(textSize)`，可以设置文字的大小。

```
paint.setTextSize(18);
canvas.drawText(text, 100, 25, paint);
paint.setTextSize(36);
canvas.drawText(text, 100, 70, paint);
paint.setTextSize(60);
canvas.drawText(text, 100, 145, paint);
paint.setTextSize(84);
canvas.drawText(text, 100, 240, paint);
```

Hello HenCoder

Hello HenCoder

Hello HenCoder

Hello HenCoder

设置文字的位置和尺寸，这些只是绘制文字最基本的操作。文字的绘制具有极高的定制性，不过由于它的定制性实在太高了，所以我会后面专门用一期来讲文字的绘制。这一期就不多讲了。

嗯.....就这样吧。绘制部分第一节，Canvas 的 drawXXX() 系列方法和 Paint 的基本使用，就到这里。

## 练习项目

为了避免转头就忘，强烈建议你趁热打铁，做一下这个练习项目：

[HenCoderPracticeDraw1](#)

