HenCoder Android 开发进阶
自定义 View 1-2 Paint 详解

这期是 HenCoder 自定义绘制的第二期： `Paint` 。如果你没看过第一期，可以先去看一下第一期

简介

上一期我已经简单说过， `Canvas` 的 `drawXXX()` 方法配合 `Paint` 的几个常用方法可以实现最常见的绘制需求；而如果你只会基本的绘制， `Paint` 的完全功能的掌握，能让你更进一步，做出一些更加细致、炫酷的效果。把 `Paint` 掌握之后，你几乎不再会遇到「iOS 组可以实现，但你却实现不了」的绘制效果。

由于依然是讲绘制的，所以这期就没有介绍视频了。绘制的内容一共需要讲大概 5~6 期才能讲完，也就是说你要看 5~6 期才能成为自定义绘制的高手。相对于上期的内容，这期的内容更为专项、深度更深。对于没有深入研究过 `Paint` 的人，这期是一个对 `Paint` 的诠释；而对于尝试过研究 `Paint` 但仍然对其中一些 API 有疑惑的人，这期也可以帮你解惑。

另外，也正由于这期的内容是更为专项的，所以建议你在看的时候，不必像上期那样把所有东西都完全记住，而是只要把内容理解了就好。这期的内容，只要做到「知道有这么个东西」，在需要用到的时候能想起来这个功能能不能做、大致用什么做就好，至于具体的实现，到时候拐回来再翻一次就行了。

好，下面进入正题。

`Paint` 的 API 大致可以分为 4 类：

- 颜色
- 效果
- drawText() 相关
- 初始化

下面我就对这 4 类分别进行介绍：

1 颜色

`Canvas` 绘制的内容，有三层对颜色的处理：

> 这图大概看看就行，不用钻研明白再往下看，因为等这章讲完你就懂了。

1.1 基本颜色

像素的基本颜色，根据绘制内容的不同而有不同的控制方式： `Canvas` 的颜色填充类方法 `drawColor/RGB/ARGB()` 的颜色，是直接写在方法的参数里，通过参数来设置的（上期讲过了）； `drawBitmap()` 的颜色，是直接由 `Bitmap` 对象来提供的（上期也讲过了）；除此之外，是图形和文字的绘制，它们的颜色就需要使用 `paint` 参数来额外设置了（下面要讲的）。

Canvas 的方法	像素颜色的设置方式
drawColor/RGB/ARGB()	直接作为参数传入
drawBitmap()	与 `bitmap` 参数的像素颜色相同
图形和文字 (drawCircle() / drawPath() / drawText() ...)	在 `paint` 参数中设置

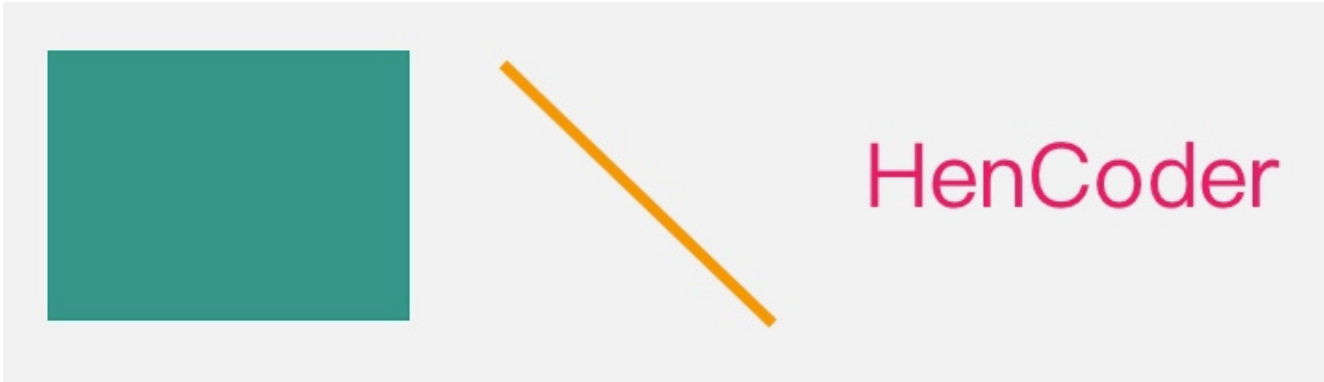
`Paint` 设置颜色的方法有两种：一种是直接用 `Paint.setColor/ARGB()` 来设置颜色，另一种是使用 `Shader` 来指定着色方案。

1.1.1 直接设置颜色

1.1.1.1 setColor(int color)

方法名和使用方法都非常简单直接，而且这个方法在上期已经介绍过了，不再多说。

```
paint.setColor(Color.parseColor("#009688"));
canvas.drawRect(30, 30, 230, 180, paint);

paint.setColor(Color.parseColor("#FF9800"));
canvas.drawLine(300, 30, 450, 180, paint);

paint.setColor(Color.parseColor("#E91E63"));
canvas.drawText("HenCoder", 500, 130, paint);
```



> `setColor()` 对应的 get 方法是 `getColor()`

1.1.1.2 setARGB(int a, int r, int g, int b)

其实和 `setColor(color)` 都是一样一样儿的，只是它的参数用的是更直接的三原色与透明度的值。实际运用中， `setColor()` 和 `setARGB()` 哪个方便和顺手用哪个吧。

```
paint.setARGB(100, 255, 0, 0);
canvas.drawRect(0, 0, 200, 200, paint);
paint.setARGB(100, 0, 0, 0);
canvas.drawLine(0, 0, 200, 200, paint);
```

1.1.2 setShader(Shader shader) 设置 Shader

除了直接设置颜色， `Paint` 还可以使用 `Shader` 。

Shader 这个英文单词很多人没有见过，它的中文叫做「着色器」，也是用于设置绘制颜色的。「着色器」不是 Android 独有的，它是图形领域里一个通用的概念，它和直接设置颜色的区别是，着色器设置的是一个颜色方案，或者说是一套着色规则。当设置了 `Shader` 之后， `Paint` 在绘制图形和文字时就不使用 `setColor/ARGB()` 设置的颜色了，而是使用 `Shader` 的方案中的颜色。

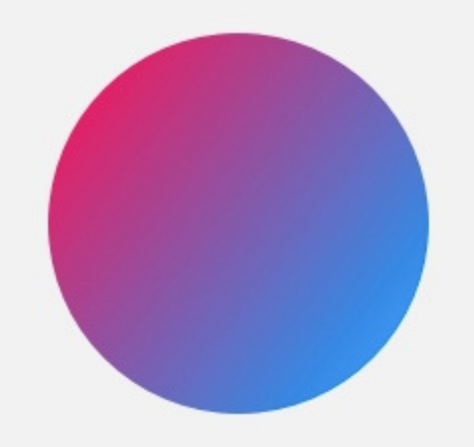
在 Android 的绘制里使用 `Shader` ，并不直接用 `Shader` 这个类，而是用它的几个子类。具体来讲有 `LinearGradient` `RadialGradient` `SweepGradient` `BitmapShader` `ComposeShader` 这么几个：

1.1.2.1 LinearGradient 线性渐变

设置两个点和两种颜色，以这两个点作为端点，使用两种颜色的渐变来绘制颜色。就像这样：

```
Shader shader = new LinearGradient(100, 100, 500, 500, Color.parseC
        Color.parseColor("#2196F3"), Shader.TileMode.CLAMP);
paint.setShader(shader);

...
```

```
canvas.drawCircle(300, 300, 200, paint);
```



> 设置了 Shader 之后，绘制出了渐变颜色的圆。（其他形状以及文字都可以这样设置颜色，我只是没给出图。）
>
> 注意：在设置了 Shader 的情况下， Paint.setColor/ARGB() 所设置的颜色就不再起作用。

构造方法：

```
LinearGradient(float x0, float y0, float x1, float y1, int color0, int c
```

。

参数：

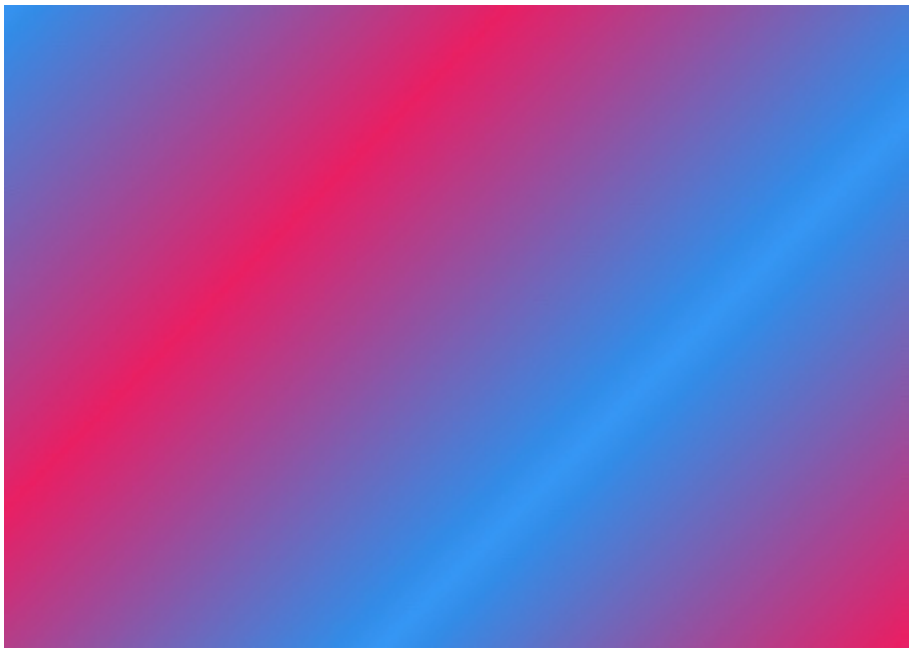
x0 y0 x1 y1：渐变的两个端点的位置

color0 color1 是端点的颜色

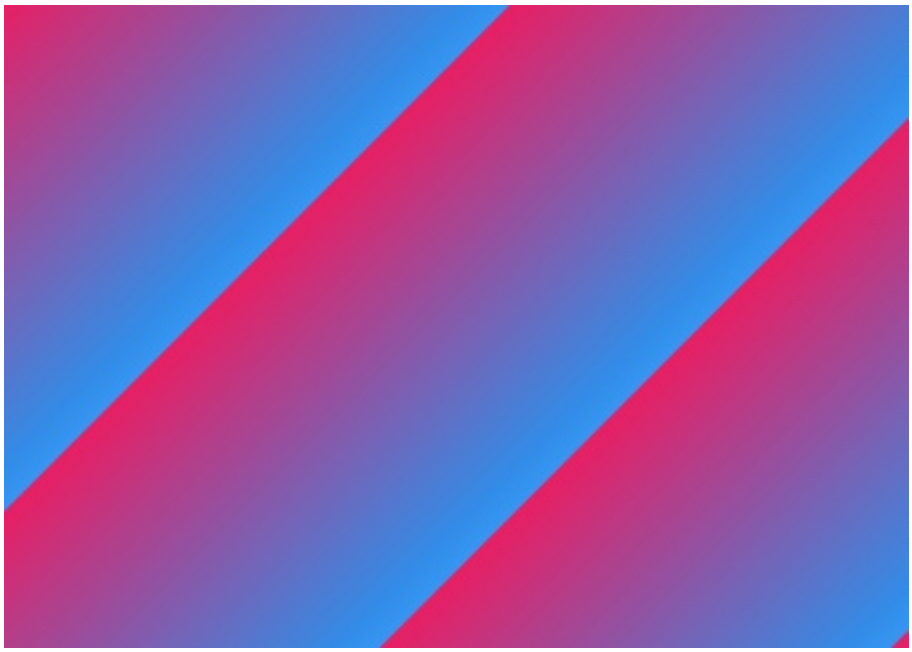
tile：端点范围之外的着色规则，类型是 TileMode。TileMode 一共有 3 个值可选： CLAMP，MIRROR 和 REPEAT。CLAMP （夹子模式？？？算了这个词我不会翻）会在端点之外延续端点处的颜色； MIRROR 是镜像模式； REPEAT 是重复模式。具体的看一下例子就明白。

CLAMP：

MIRROR:



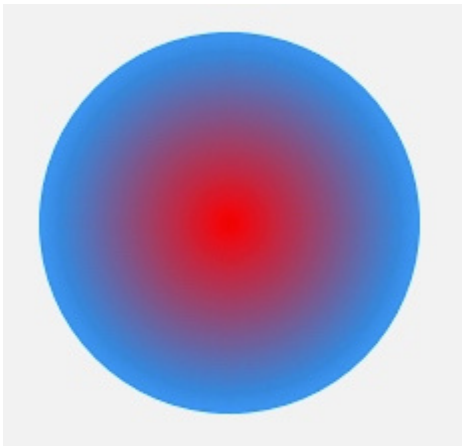
REPEAT:



1.1.2.2 RadialGradient 辐射渐变

辐射渐变很好理解，就是从中心向周围辐射状的渐变。大概像这样：

```
Shader shader = new RadialGradient(300, 300, 200, Color.parseColor(
        Color.parseColor("#2196F3"), Shader.TileMode.CLAMP);
paint.setShader(shader);

...

canvas.drawCircle(300, 300, 200, paint);
```



构造方法：

```
RadialGradient(float centerX, float centerY, float radius, int centerCol
```

参数:

centerX centerY： 辐射中心的坐标

radius： 辐射半径

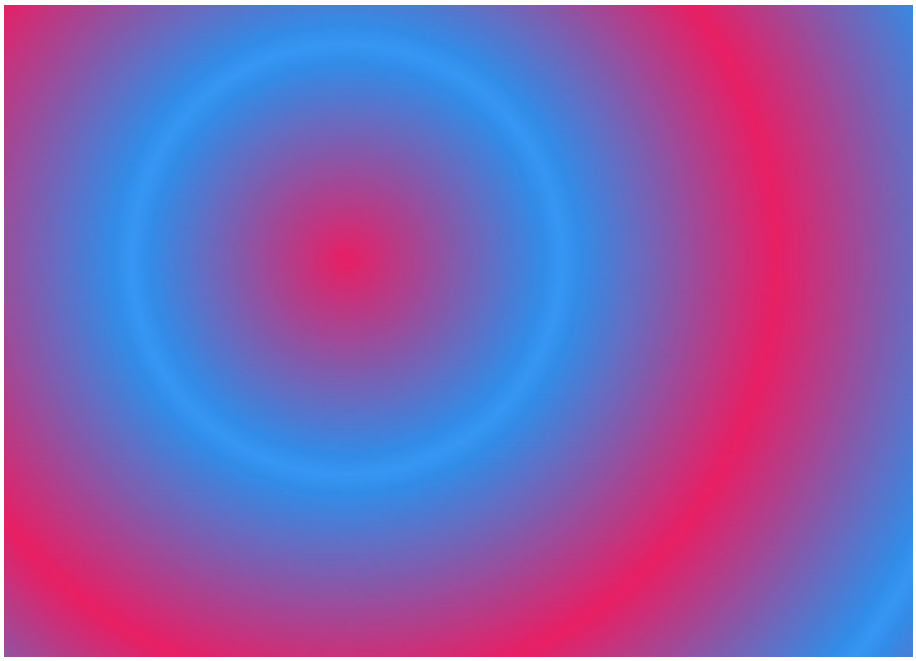
centerColor： 辐射中心的颜色

edgeColor： 辐射边缘的颜色

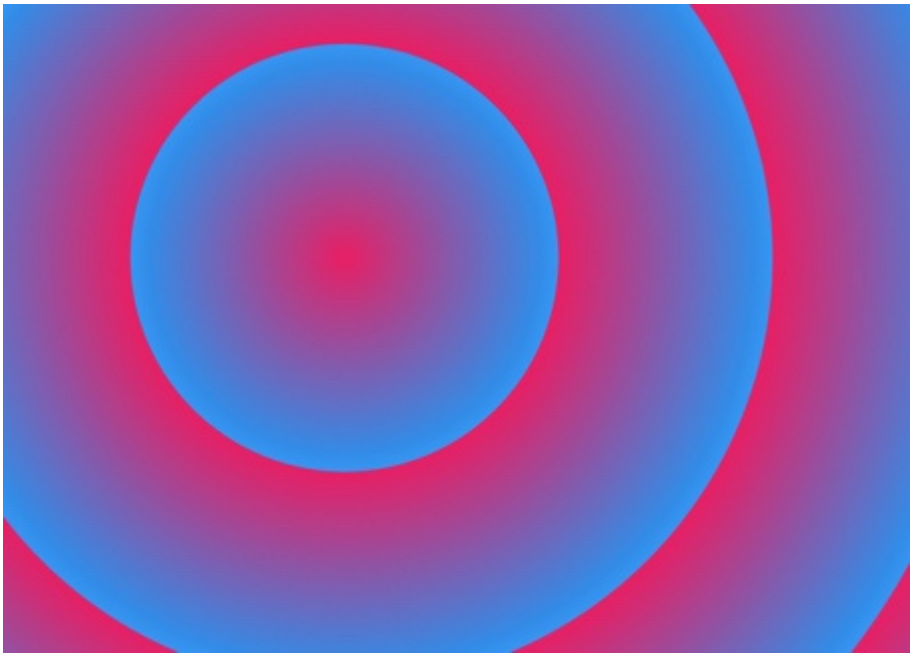
tileMode： 辐射范围之外的着色模式。

CLAMP：

MIRROR：



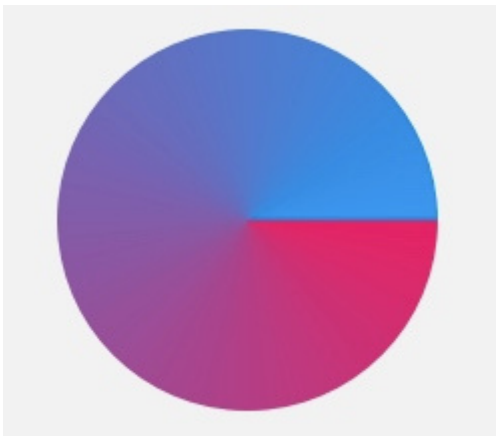
REPEAT：



1.1.2.3 SweepGradient 扫描渐变

又是一个渐变。「扫描渐变」这个翻译我也不知道精确不精确。大概是这样：

```
Shader shader = new SweepGradient(300, 300, Color.parseColor("#E91E
        Color.parseColor("#2196F3"));
paint.setShader(shader);

...

canvas.drawCircle(300, 300, 200, paint);
```



构造方法：

```
SweepGradient(float cx, float cy, int color0, int color1)
```

参数：

cx cy ： 扫描的中心

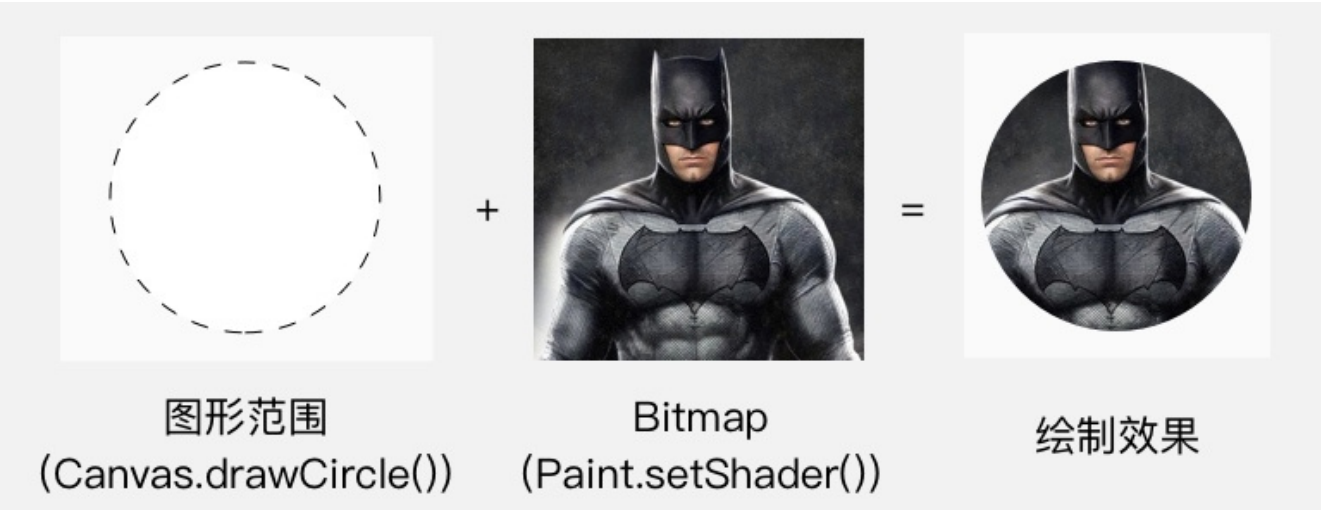
color0： 扫描的起始颜色

color1： 扫描的终止颜色

1.1.2.4 BitmapShader

用 Bitmap 来着色（终于不是渐变了）。其实也就是用 Bitmap 的像素来作为图形或文字的填充。大概像这样：

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.draw
Shader shader = new BitmapShader(bitmap, Shader.TileMode.CLAMP, Sha
paint.setShader(shader);

...

canvas.drawCircle(300, 300, 200, paint);
```



> 嗯，看着跟 Canvas.drawBitmap() 好像啊？事实上也是一样的效果。如果你想绘制圆形的 Bitmap，就别用 drawBitmap() 了，改用 drawCircle() + BitmapShader 就可以了（其他形状同理）。

构造方法:

```
BitmapShader(Bitmap bitmap, Shader.TileMode tileX, Shader.TileMode tileY
```

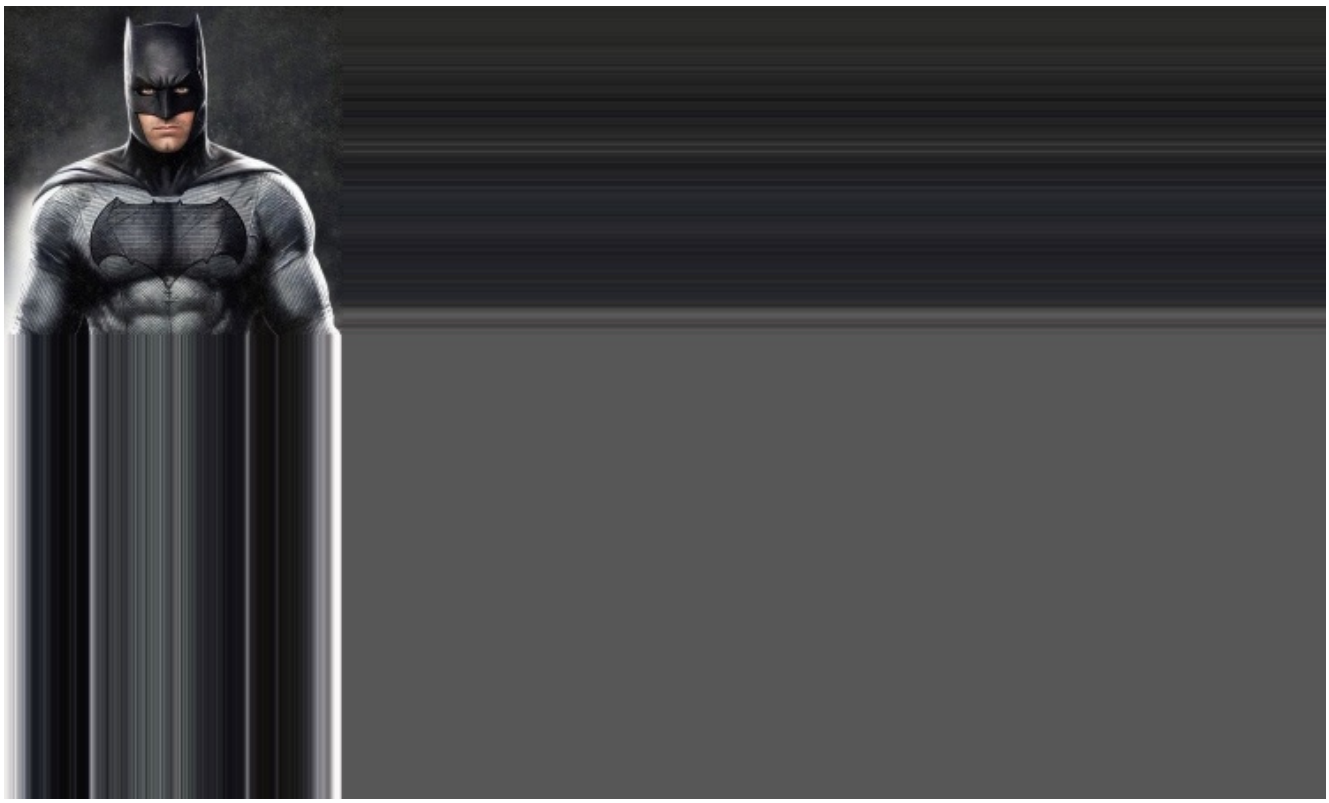
参数:

bitmap： 用来做模板的 Bitmap 对象

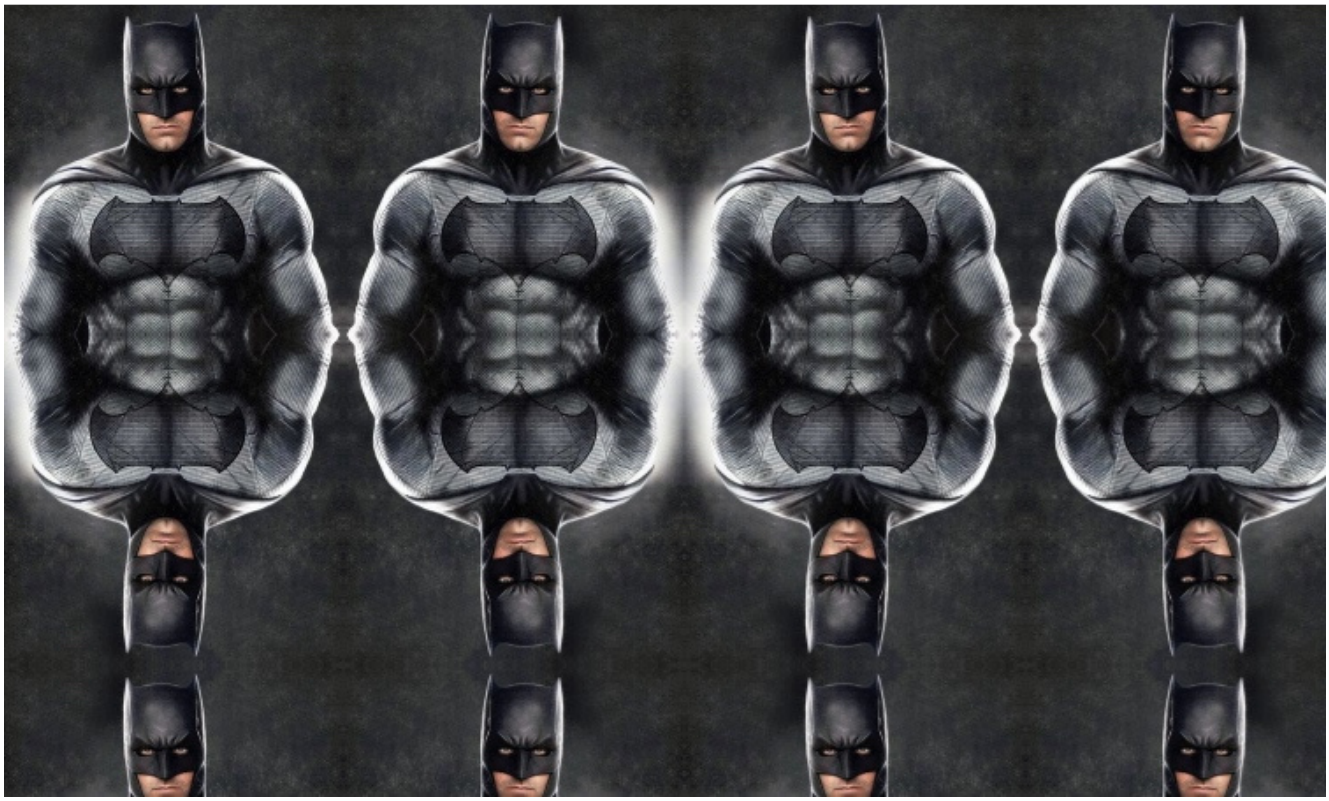
tileX： 横向的 TileMode

tileY： 纵向的 TileMode 。

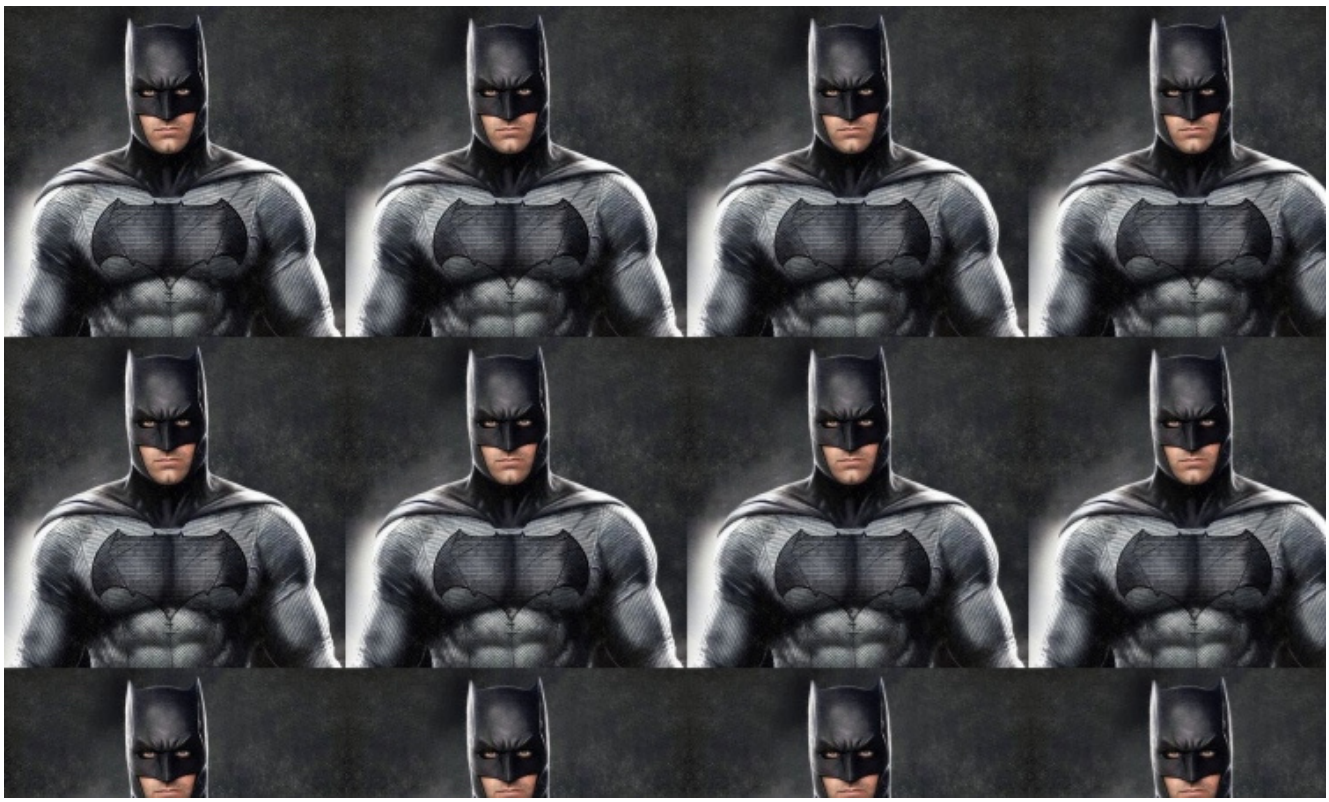
CLAMP：



MIRROR：



REPEAT：

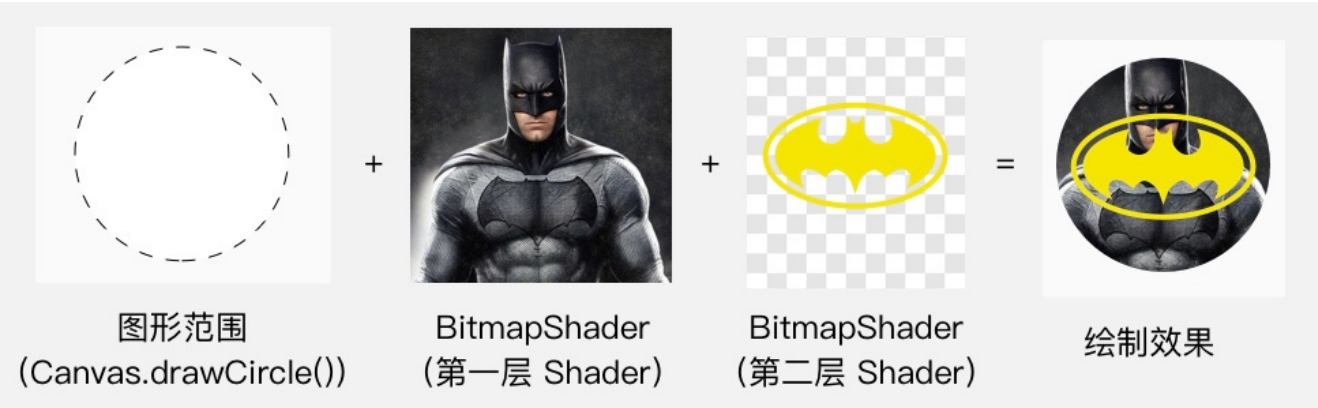


1.1.2.5 ComposeShader 混合着色器

所谓混合，就是把两个 Shader 一起使用。

```
// 第一个 Shader：头像的 Bitmap
Bitmap bitmap1 = BitmapFactory.decodeResource(getResources(), R.dra
Shader shader1 = new BitmapShader(bitmap1, Shader.TileMode.CLAMP, S

// 第二个 Shader：从上到下的线性渐变（由透明到黑色）
Bitmap bitmap2 = BitmapFactory.decodeResource(getResources(), R.dra
Shader shader2 = new BitmapShader(bitmap2, Shader.TileMode.CLAMP, S

// ComposeShader：结合两个 Shader
Shader shader = new ComposeShader(shader1, shader2, PorterDuff.Mode
paint.setShader(shader);

...

canvas.drawCircle(300, 300, 300, paint);
```

> 注意：上面这段代码中我使用了两个 `BitmapShader` 来作为 `ComposeShader()` 的参数，而 `ComposeShader()` 在硬件加速下是不支持两个相同类型的 `Shader` 的，所以这里也需要关闭硬件加速才能看到效果。



构造方法：`ComposeShader(Shader shaderA, Shader shaderB, PorterDuff.Mode mode)`

参数：

`shaderA`, `shaderB`：两个相继使用的 `Shader`

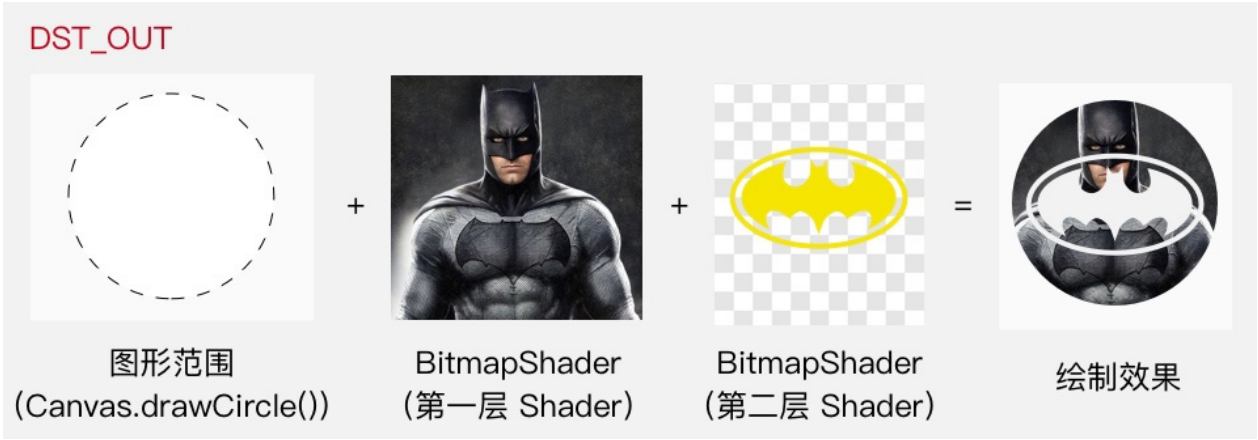
`mode`：两个 `Shader` 的叠加模式，即 `shaderA` 和 `shaderB` 应该怎样共同绘制。它的类型是 `PorterDuff.Mode` 。

PorterDuff.Mode

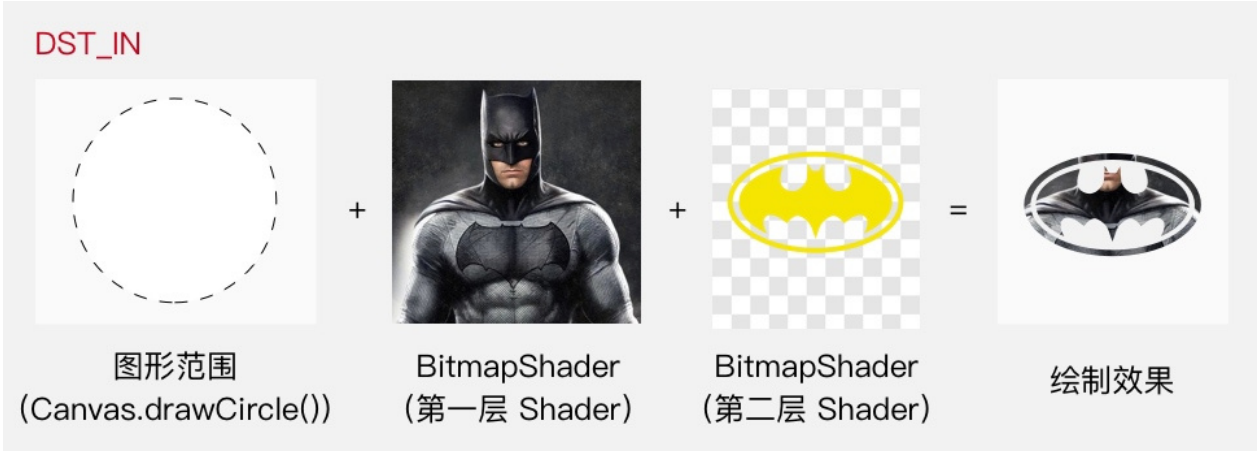
`PorterDuff.Mode` 是用来指定两个图像共同绘制时的颜色策略的。它是一个 enum，不同的 `Mode` 可以指定不同的策略。「颜色策略」的意思，就是说把源图像绘制到目标图像处时应该怎样确定二者结合后的颜色，而对于 `ComposeShader(shaderA, shaderB, mode)` 这个具体的方法，就是指应该怎样把 `shaderB` 绘制在 `shaderA` 上来得到一个结合后的 `Shader` 。

没有听说过 `PorterDuff.Mode` 的人，看到这里很可能依然会一头雾水：「什么怎么结合？就……两个图像一叠加，结合呗？还能怎么结合？」你还别说，还真的是有很多种策略来结合。

最符合直觉的结合策略，就是我在上面这个例子中使用的 `Mode：SRC_OVER` 。它的算法非常直观：就像上面图中的那样，把源图像直接铺在目标图像上。不过，除了这种，其实还有一些其他的结合方式。例如如果我把上面例子中的参数 `mode` 改为 `PorterDuff.Mode.DST_OUT` ，就会变成挖空效果：



而如果再把 `mode` 改为 `PorterDuff.Mode.DST_IN` ，就会变成蒙版抠图效果：



这下明白了吧?

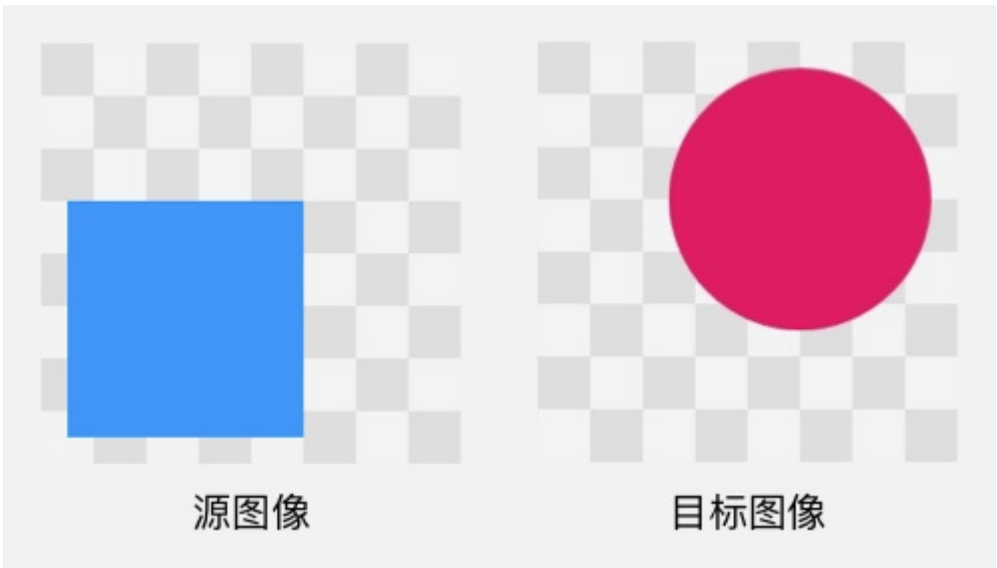
具体来说， `PorterDuff.Mode` 一共有 17 个，可以分为两类：

1. Alpha 合成 (Alpha Compositing)
2. 混合 (Blending)

第一类，Alpha 合成，其实就是 「PorterDuff」 这个词所指代的算法。「PorterDuff」 并不是一个具有实际意义的词组，而是两个人的名字（准确讲是姓）。这两个人当年共同发表了一篇论文，描述了 12 种将两个图像共同绘制的操作（即算法）。而这篇论文所论述的操作，都是关于 Alpha 通道（也就是我们通俗理解的「透明度」）的计算的，后来人们就把这类计算称为**Alpha 合成**（Alpha Compositing）。

看下效果吧。效果直接盗 Google 的官方文档了。

源图像和目标图像：

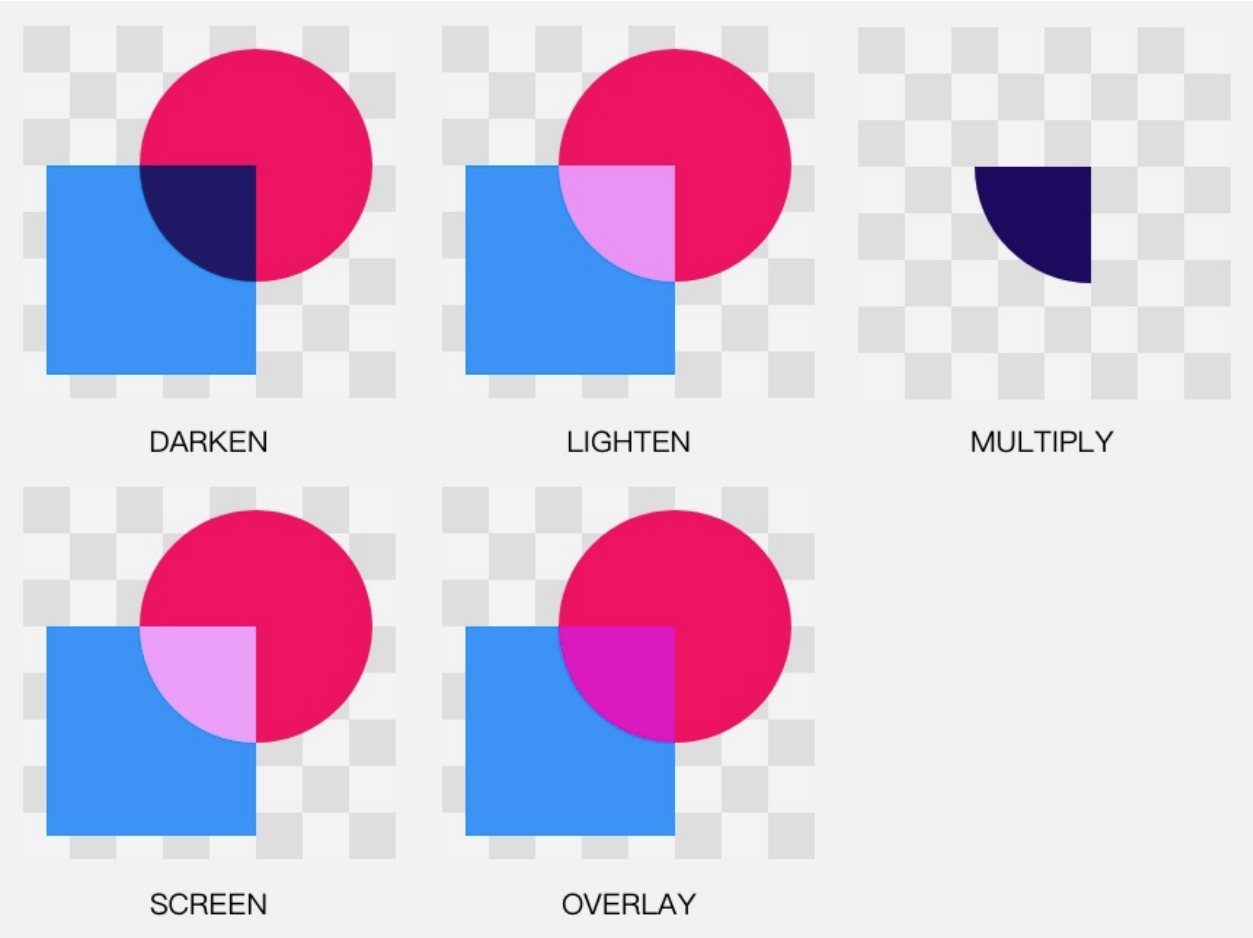


Alpha 合成：

第二类，混合，也就是 Photoshop 等制图软件里都有的那些混合模式（`multiply darken lighten` 之类的）。这一类操作的是颜色本身而不是 `Alpha` 通道，并不属于 `Alpha` 合成，所以和 Porter 与 Duff 这两个人也没什么

关系，不过为了使用的方便，它们同样也被 Google 加进了 `PorterDuff.Mode` 里。

效果依然盗 官方文档。



结论

从效果图可以看出，Alpha 合成类的效果都比较直观，基本上可以使用简单的口头表达来描述它们的算法（起码对于不透明的源图像和目标图像来说是可以的），例如 `SRC_OVER` 表示「二者都绘制，但要源图像放在目标图像的上面」，`DST_IN` 表示「只绘制目标图像，并且只绘制它和源图像重合的区域」。

而混合类的效果就相对抽象一些，只从效果图不太能看得出它们的着色算法，更看不出来它们有什么用。不过没关系，你如果拿着这些名词去问你司的设计师，他们八成都能给你说出来个 123。

所以对于这些 `Mode`，正确的做法是：**对于 Alpha 合成类的操作，掌握他们，并在实际开发中灵活运用；而对于混合类的，你只要把它们的名字记住就好**

> **了，这样当某一天设计师告诉你「我要做这种混合效果」的时候，你可以马上知道自己能不能做，怎么做。**
>
> 另外：`PorterDuff.Mode` 建议你动手用一下试试，对加深理解有帮助。

好了，这些就是几个 `Shader` 的具体介绍。

除了使用 `setColor/ARGB()` 和 `setShader()` 来设置基本颜色， `Paint` 还可以来设置 `ColorFilter`，来对颜色进行第二层处理。

1.2 setColorFilter(ColorFilter colorFilter)

`ColorFilter` 这个类，它的名字已经足够解释它的作用：为绘制设置颜色过滤。颜色过滤的意思，就是为绘制的内容设置一个统一的过滤策略，然后 `Canvas.drawXXX()` 方法会对每个像素都进行过滤后再绘制出来。举几个现实中比较常见的颜色过滤的例子：

- 有色光照射：

- 有色玻璃透视：

- 胶卷：

在 `Paint` 里设置 `ColorFilter` ，使用的是 `Paint.setColorFilter(ColorFilter filter)` 方法。 `ColorFilter` 并不直接使用，而是使用它的子类。它共有三个子类： `LightingColorFilter` `PorterDuffColorFilter` 和 `ColorMatrixColorFilter` 。

1.2.1 LightingColorFilter

这个 `LightingColorFilter` 是用来模拟简单的光照效果的。

`LightingColorFilter` 的构造方法是 `LightingColorFilter(int mul, int add)`，参数里的 `mul` 和 `add` 都是和颜色值格式相同的 int 值，其中 `mul` 用来和目标像素相乘，`add` 用来和目标像素相加：

```
R' = R * mul.R / 0xff + add.R
G' = G * mul.G / 0xff + add.G
B' = B * mul.B / 0xff + add.B
```

一个「保持原样」的「基本 `LightingColorFilter`」，`mul` 为 `0xffffff`，`add` 为 `0x000000`（也就是0），那么对于一个像素，它的计算过程就是：

```
R' = R * 0xff / 0xff + 0x0 = R // R' = R
G' = G * 0xff / 0xff + 0x0 = G // G' = G
B' = B * 0xff / 0xff + 0x0 = B // B' = B
```

基于这个「基本 `LightingColorFilter`」，你就可以修改一下做出其他的 filter。比如，如果你想去掉原像素中的红色，可以把它的 `mul` 改为 `0x00ffff`（红色部分为 0），那么它的计算过程就是：

```
R' = R * 0x0 / 0xff + 0x0 = 0 // 红色被移除
G' = G * 0xff / 0xff + 0x0 = G
B' = B * 0xff / 0xff + 0x0 = B
```

具体效果是这样的：

```
ColorFilter lightingColorFilter = new LightingColorFilter(0x00ffff,
paint.setColorFilter(lightingColorFilter);
```

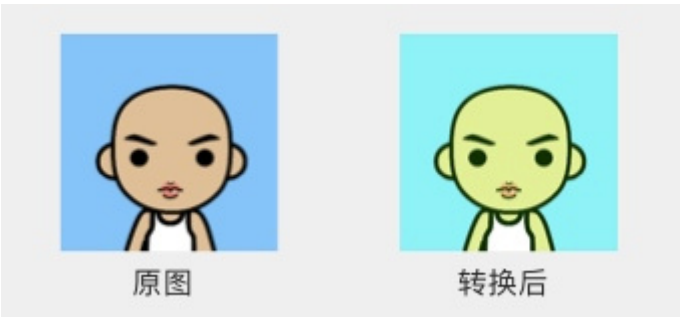
> 表情忽然变得阴郁了

或者，如果你想让它的绿色更亮一些，就可以把它的 `add` 改为 `0x003000` （绿色部分为 0x30 ），那么它的计算过程就是：

```
R' = R * 0xff / 0xff + 0x0 = R
G' = G * 0xff / 0xff + 0x30 = G + 0x30 // 绿色被加强
B' = B * 0xff / 0xff + 0x0 = B
```

效果是这样：

```
ColorFilter lightingColorFilter = new LightingColorFilter(0xffffff,
paint.setColorFilter(lightingColorFilter);
```



> 这样的表情才阳光

至于怎么修改参数来模拟你想要的某种具体光照效果，你就别问我了，还是跟你司设计师讨论吧，这个我不专业……

1.2.2 PorterDuffColorFilter

这个 PorterDuffColorFilter 的作用是使用一个指定的颜色和一种指定的 PorterDuff.Mode 来与绘制对象进行合成。它的构造方法是 PorterDuffColorFilter(int color, PorterDuff.Mode mode) 其中的 color 参数是指定的颜色， mode 参数是指定的 Mode 。同样也是 PorterDuff.Mode ，不过和 ComposeShader 不同的是， PorterDuffColorFilter 作为一个 ColorFilter，只能指定一种颜色作为源，而不是一个 Bitmap 。

PorterDuff.Mode 前面已经讲过了，而 PorterDuffColorFilter 本身的使用是非常简单的，所以不再展开讲。

1.2.3 ColorMatrixColorFilter

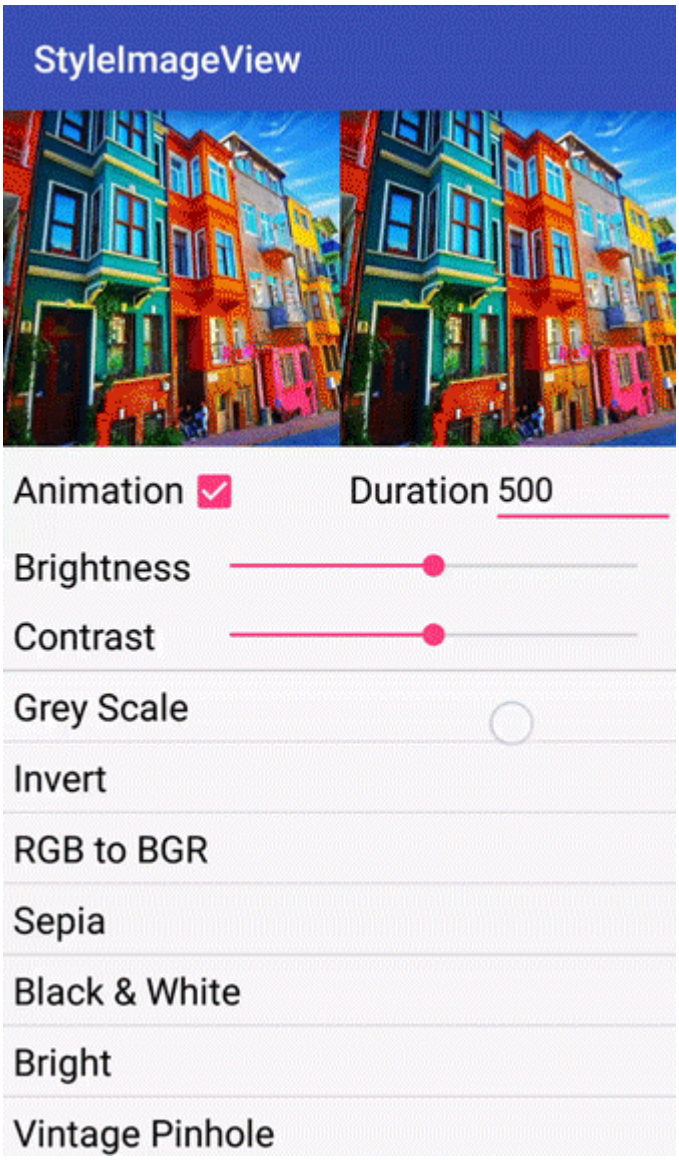
这个就厉害了。 ColorMatrixColorFilter 使用一个 ColorMatrix 来对颜色进行处理。 ColorMatrix 这个类，内部是一个 4x5 的矩阵：

```
[ a, b, c, d, e,
  f, g, h, i, j,
  k, l, m, n, o,
  p, q, r, s, t ]
```

通过计算， ColorMatrix 可以把要绘制的像素进行转换。对于颜色 [R, G, B, A] ，转换算法是这样的：

```
R′ = a*R + b*G + c*B + d*A + e;
G′ = f*R + g*G + h*B + i*A + j;
B′ = k*R + l*G + m*B + n*A + o;
A′ = p*R + q*G + r*B + s*A + t;
```

ColorMatrix 有一些自带的方法可以做简单的转换，例如可以使用 setSaturation(float sat) 来设置饱和度；另外你也可以自己去设置它的每一个元素来对转换效果做精细调整。具体怎样设置会有怎样的效果，我就不讲了（其实是我也不太会😅）。如果你有需求，可以试一下程大治同学做的这个库：StyleImageView



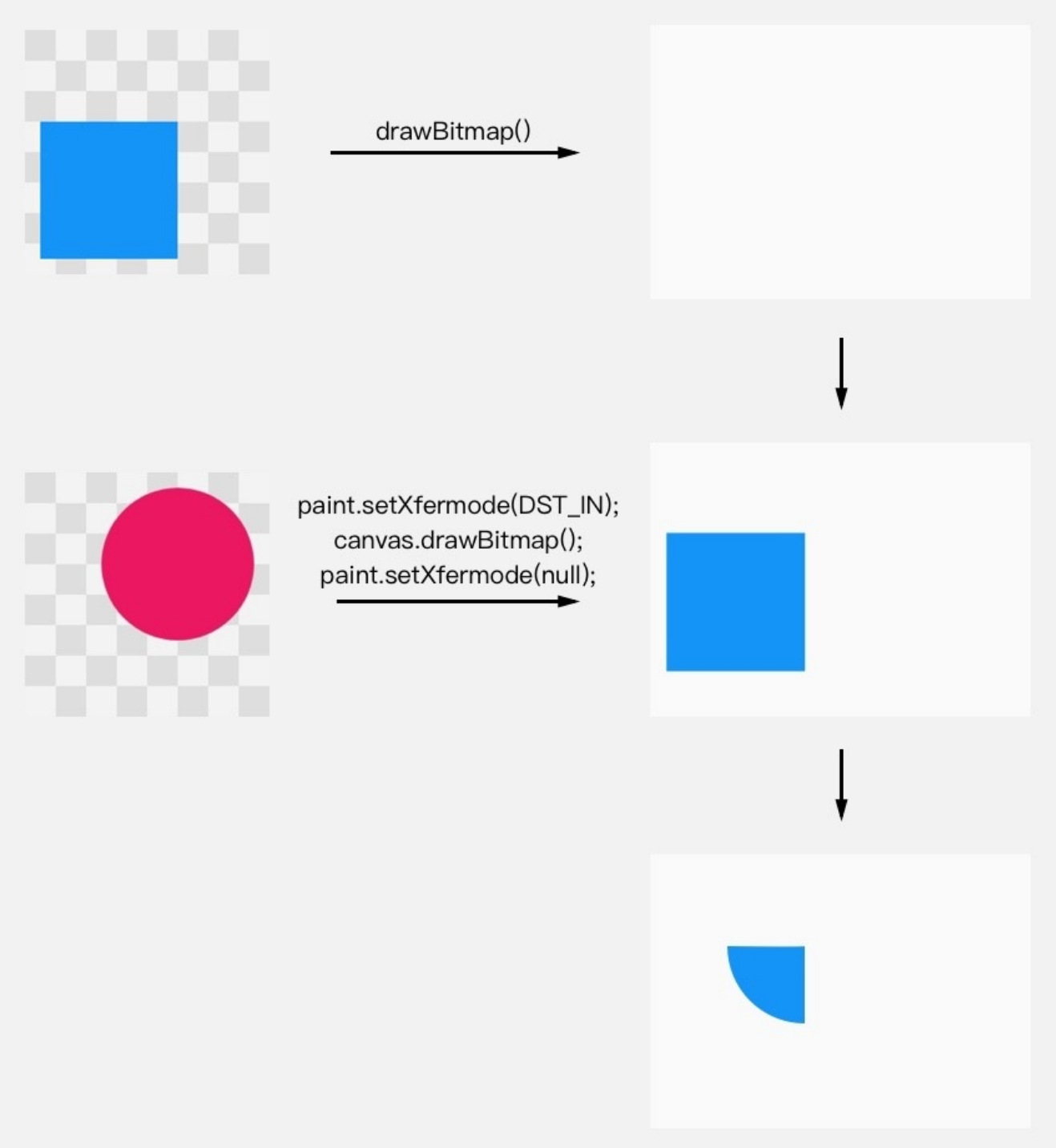
以上，就是 `Paint` 对颜色的第二层处理：通过 `setColorFilter(colorFilter)` 来加工颜色。

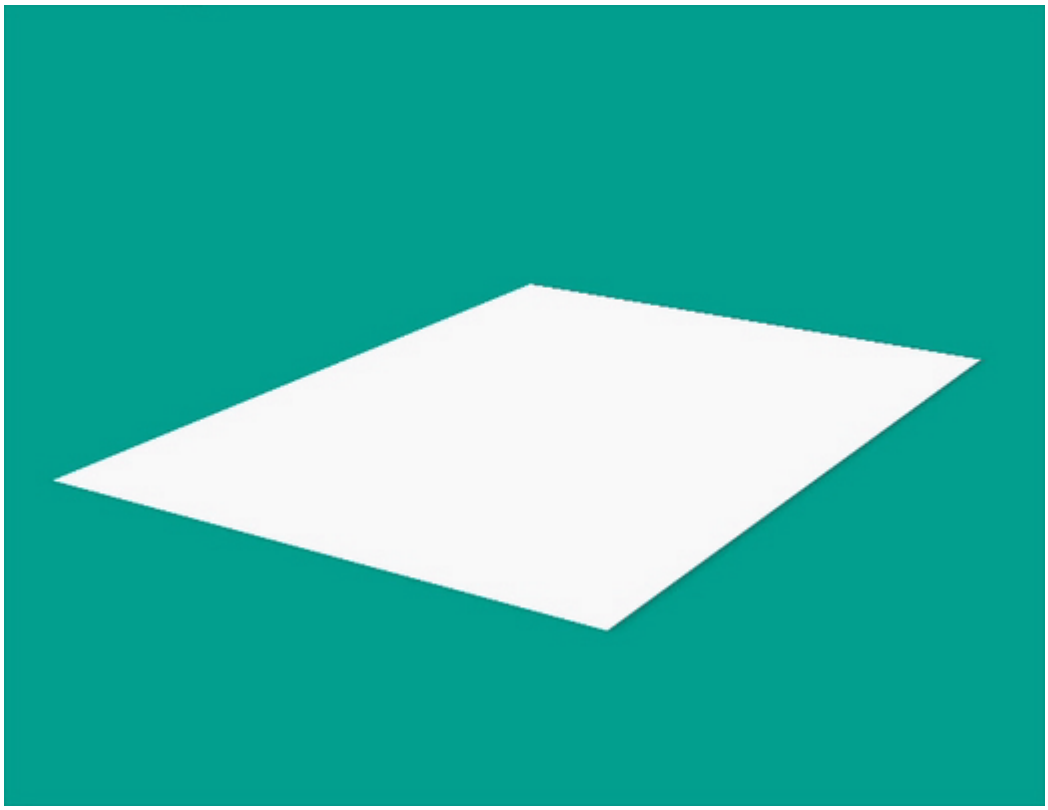
除了基本颜色的设置（ `setColor/ARGB()`, `setShader()` ）以及基于原始颜色的过滤（ `setColorFilter()` ）之外， `Paint` 最后一层处理颜色的方法是 `setXfermode(Xfermode xfermode)` ，它处理的是「当颜色遇上 View」的问题。

1.3 setXfermode(Xfermode xfermode)

"Xfermode" 其实就是 "Transfer mode"，用 "X" 来代替 "Trans" 是一些美国人喜欢用的简写方式。严谨地讲， `Xfermode` 指的是你要绘制的内容和 `Canvas` 的目标位置的内容应该怎样结合计算出最终的颜色。但通俗地说，其实就是要你以绘制的内容作为源图像，以 View 中已有的内容作为目标图像，选取一个 `PorterDuff.Mode` 作为绘制内容的颜色处理方案。就像这样：

```
Xfermode xfermode = new PorterDuffXfermode(PorterDuff.Mode.DST_IN);

...

canvas.drawBitmap(rectBitmap, 0, 0, paint); // 画方
paint.setXfermode(xfermode); // 设置 Xfermode
canvas.drawBitmap(circleBitmap, 0, 0, paint); // 画圆
paint.setXfermode(null); // 用完及时清除 Xfermode
```





又是 `PorterDuff.Mode` 。 `PorterDuff.Mode` 在 `Paint` 一共有三处 API ，它们的工作原理都一样，只是用途不同：

API	用途
ComposeShader	混合两个 Shader
PorterDuffColorFilter	增加一个单色的 ColorFilter
Xfermode	设置绘制内容和 View 中已有内容的混合计算方式

另外，从上面的示例代码可以看出，创建 `Xfermode` 的时候其实是创建的它的子类 `PorterDuffXfermode` 。而事实上， `Xfermode` 也只有这一个子类。所以在设置 `Xfermode` 的时候不用多想，直接用 `PorterDuffXfermode` 吧。

> 「只有一个子类？？？什么设计？」

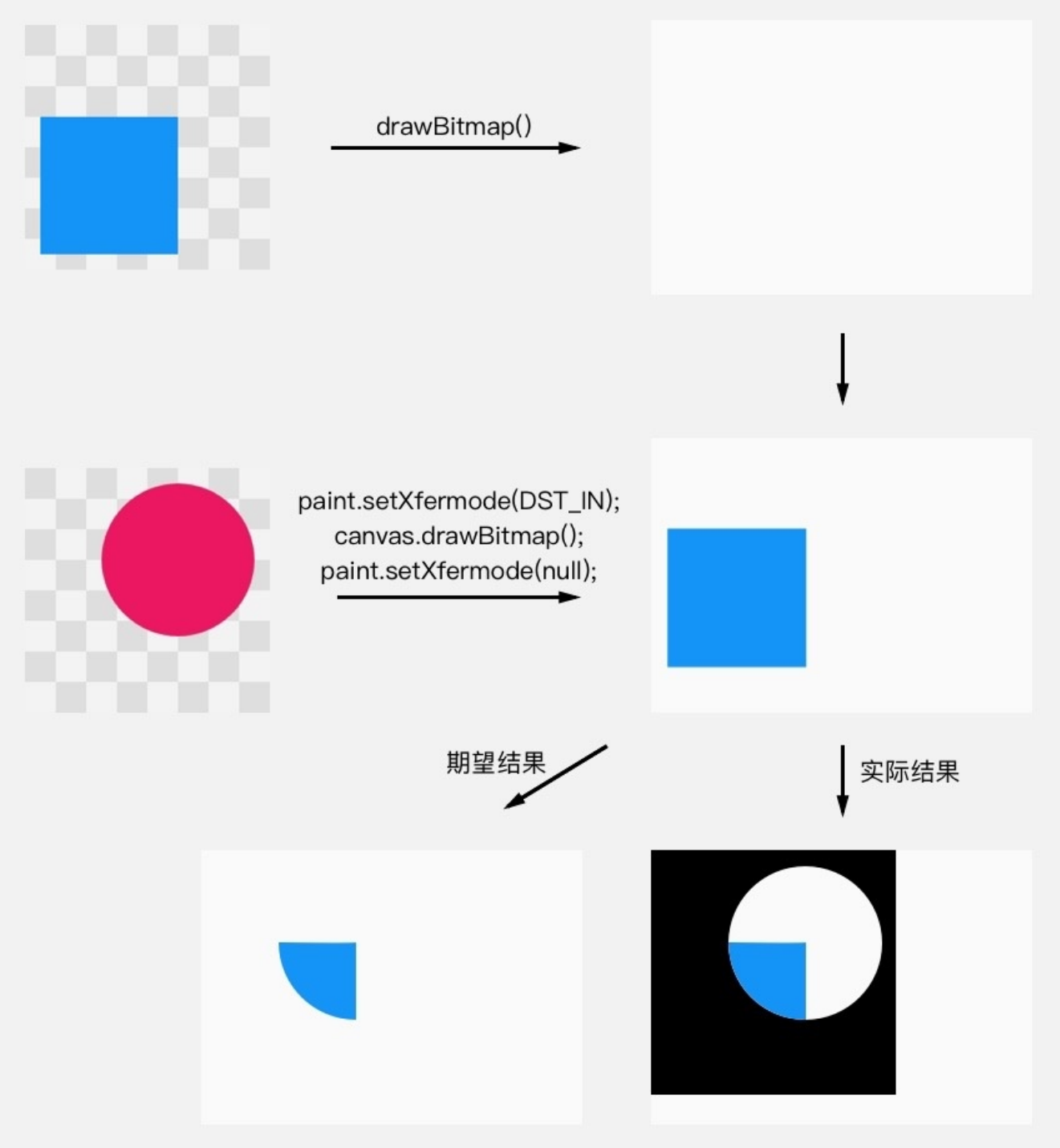
> 
>
>
> 其实在更早的 Android 版本中， `Xfermode` 还有别的子类，但别的子类现在已经 `deprecated` 了，如今只剩下了 `PorterDuffXfermode` 。所以目前它的使用看起来好像有点啰嗦，但其实是由于历史遗留问题。

Xfermode 注意事项

`Xfermode` 使用很简单，不过有两点需要注意：

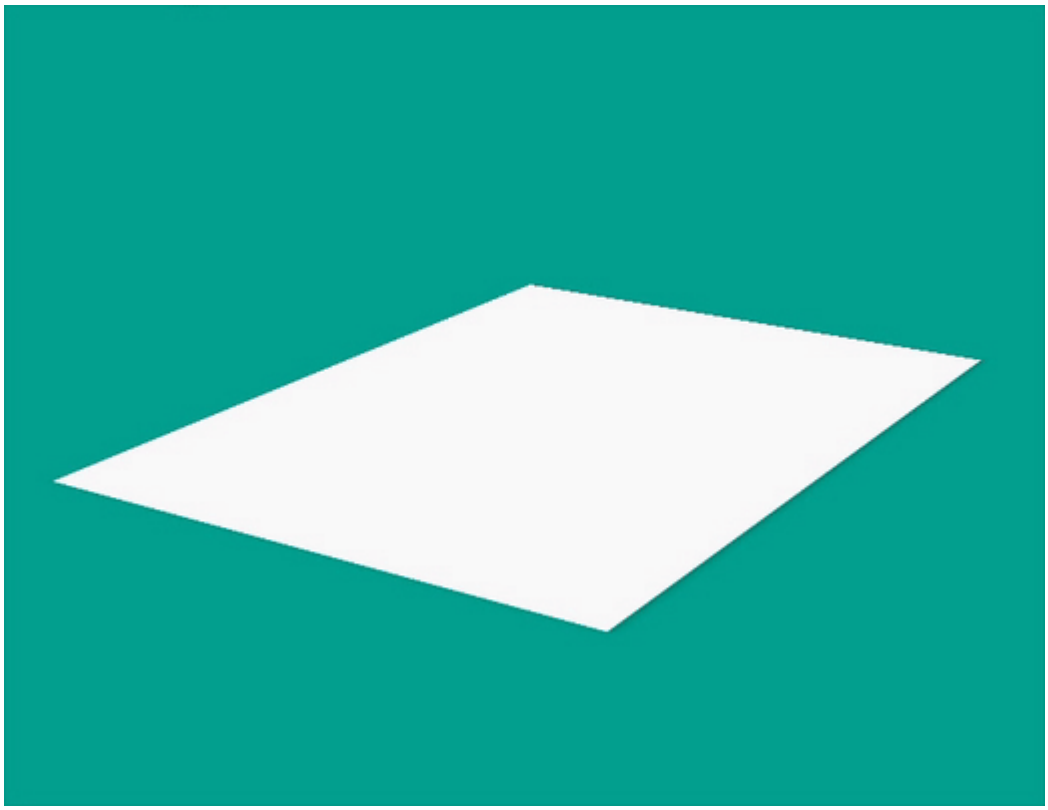
1. 使用离屏缓冲（Off-screen Buffer）

实质上，上面这段例子代码，如果直接执行的话是不会绘制出图中效果的，程序的绘制也不会像上面的动画那样执行，而是会像这样：



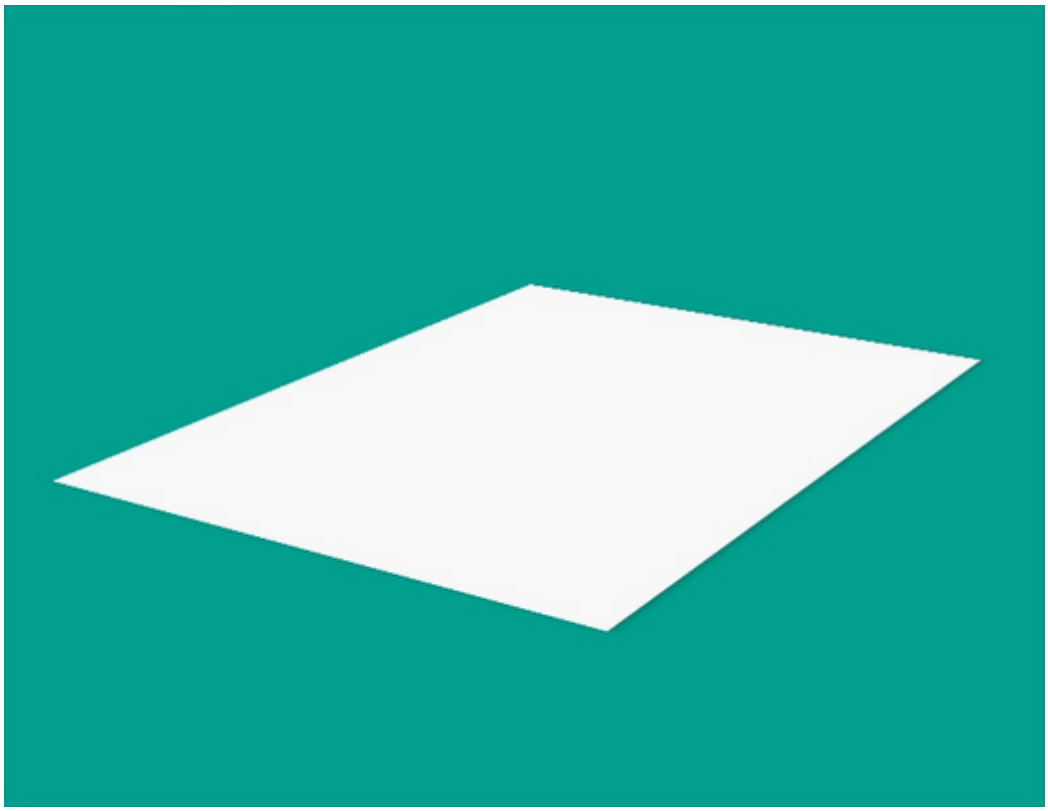
> 为什么会这样?

按照逻辑我们会认为，在第二步画圆的时候，跟它共同计算的是第一步绘制的方形。但实际上，却是整个 `View` 的显示区域都在画圆的时候参与计算，并且 `View` 自身的底色并不是默认的透明色，而且是遵循一种迷之逻辑，导致不仅绘制的是整个圆的范围，而且在范围之外都变成了黑色。就像这样：



> 这……那可如何是好?

要想使用 `setXfermode()` 正常绘制，必须使用离屏缓存 (Off-screen Buffer) 把内容绘制在额外的层上，再把绘制好的内容贴回 View 中。也就是这样：



通过使用离屏缓冲，把要绘制的内容单独绘制在缓冲层， `Xfermode` 的使用就不会出现奇怪的结果了。使用离屏缓冲有两种方式：

- `Canvas.saveLayer()`

  `saveLayer()` 可以做短时的离屏缓冲。使用方法很简单，在绘制代码的前后各加一行代码，在绘制之前保存，绘制之后恢复：

```
int saved = canvas.saveLayer(null, null, Canvas.ALL_SAVE_FLAG);


canvas.drawBitmap(rectBitmap, 0, 0, paint); // 画方
paint.setXfermode(xfermode); // 设置 Xfermode
canvas.drawBitmap(circleBitmap, 0, 0, paint); // 画圆
paint.setXfermode(null); // 用完及时清除 Xfermode


canvas.restoreToCount(saved);
```

- `View.setLayerType()`

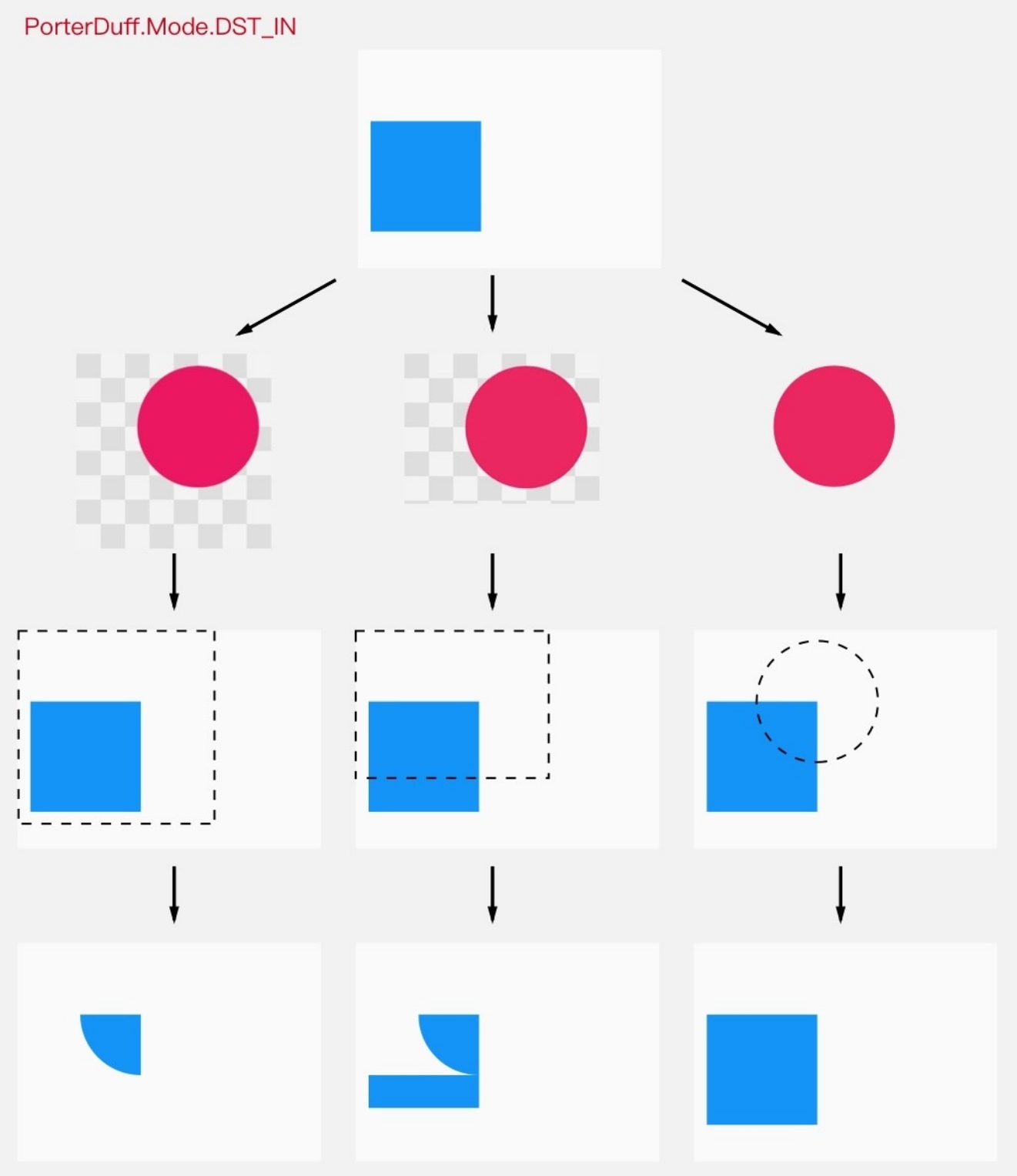
  `View.setLayerType()` 是直接把整个 `View` 都绘制在离屏缓冲中。`setLayerType(LAYER_TYPE_HARDWARE)` 是使用 GPU 来缓冲，`setLayerType(LAYER_TYPE_SOFTWARE)` 是直接直接用一个 `Bitmap` 来缓冲。

> 关于 `Canvas.saveLayer()` 和 `View.setLayerType()` ，这里就不细讲它们的意义和原理了，后面也许我会专门用一期来讲它们。

如果没有特殊需求，可以选用第一种方法 `Canvas.saveLayer()` 来设置离屏缓冲，以此来获得更高的性能。更多关于离屏缓冲的信息，可以看官方文档中对于硬件加速的介绍。

2. 控制好透明区域

使用 Xfermode 来绘制的内容，除了注意使用离屏缓冲，还应该注意控制它的透明区域不要太小，要让它足够覆盖到要和它结合绘制的内容，否则得到的结果很可能不是你想要的。我用图片来具体说明一下：



> 如图所示，由于透明区域过小而覆盖不到的地方，将不会受到 Xfermode 的影响。

好，到此为止，前面讲的就是 `Paint` 的第一类 API——关于颜色的三层设置：直接设置颜色的 API 用来给图形和文字设置颜色； `setColorFilter()` 用来基于颜色进行过滤处理； `setXfermode()` 用来处理源图像和 `View` 已有内容的关系。

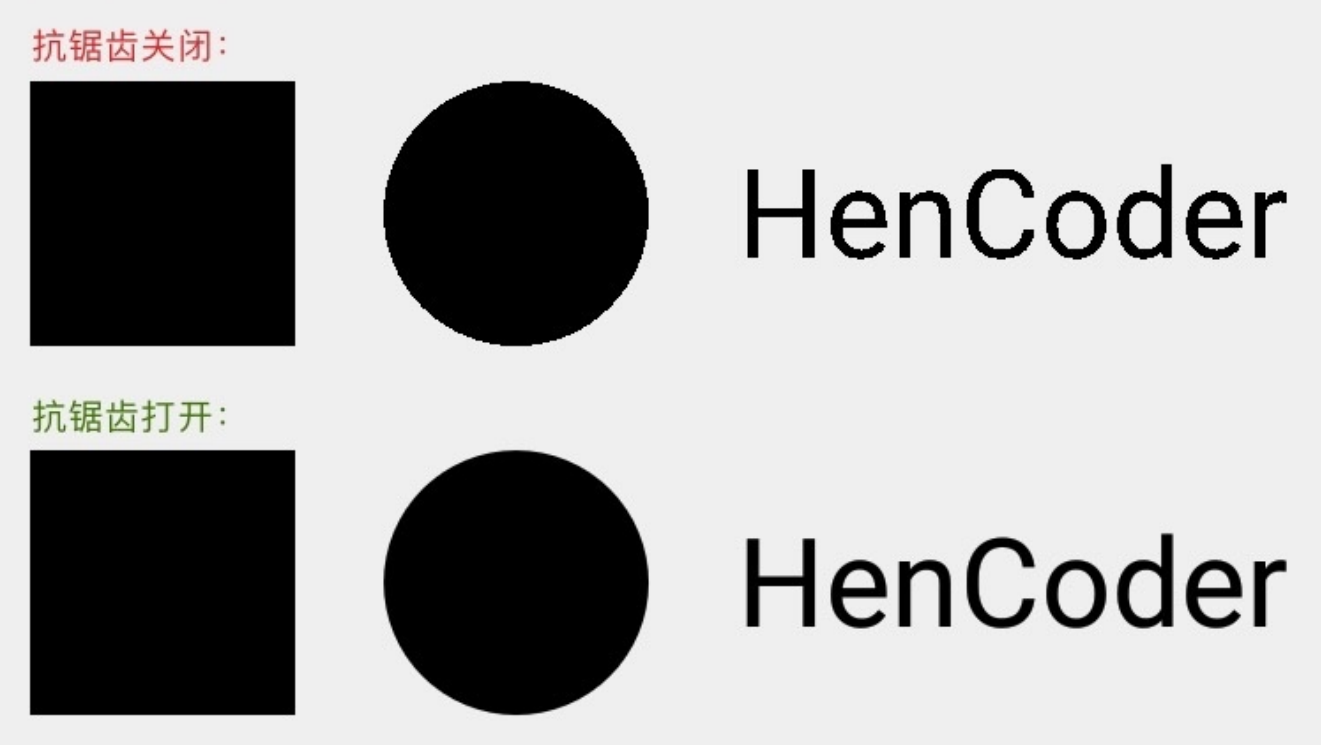
再贴一次本章开始处的图作为回顾：

2 效果

效果类的 API ，指的就是抗锯齿、填充/轮廓、线条宽度等等这些。

2.1 setAntiAlias (boolean aa) 设置抗锯齿

抗锯齿在上一节已经讲过了，话不多说，直接上图：



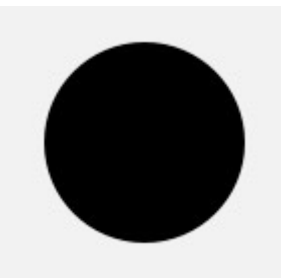
抗锯齿默认是关闭的，如果需要抗锯齿，需要显式地打开。另外，除了 `setAntiAlias(aa)` 方法，打开抗锯齿还有一个更方便的方式：构造方法。创建 `Paint` 对象的时候，构造方法的参数里加一个 `ANTI_ALIAS_FLAG` 的 flag，就可以在初始化的时候就开启抗锯齿。

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

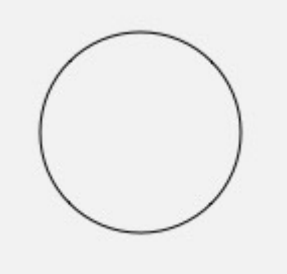
2.2 setStyle(Paint.Style style)

`setStyle(style)` 也在上一节讲过了，用来设置图形是线条风格还是填充风格的（也可以二者并用）：

```
paint.setStyle(Paint.Style.FILL); // FILL 模式，填充
canvas.drawCircle(300, 300, 200, paint);
```



```
paint.setStyle(Paint.Style.STROKE); // STROKE 模式，画线
canvas.drawCircle(300, 300, 200, paint);
```



```
paint.setStyle(Paint.Style.FILL_AND_STROKE); // FILL_AND_STROKE 模式
canvas.drawCircle(300, 300, 200, paint);
```

FILL 模式是默认模式，所以如果之前没有设置过其他的 Style，可以不用 setStyle(Paint.Style.FILL) 这句。

2.3 线条形状

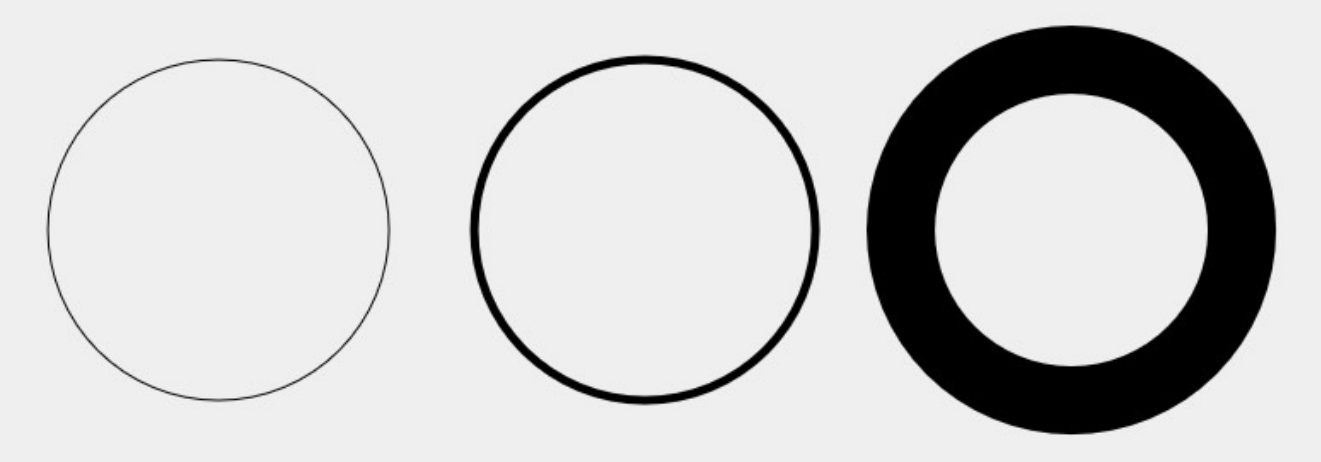
设置线条形状的一共有 4 个方法：setStrokeWidth(float width)，setStrokeCap(Paint.Cap cap)，setStrokeJoin(Paint.Join join)，setStrokeMiter(float miter) 。

2.3.1 setStrokeWidth(float width)

设置线条宽度。单位为像素，默认值是 0。

```
paint.setStyle(Paint.Style.STROKE);
paint.setStrokeWidth(1);
canvas.drawCircle(150, 125, 100, paint);
```

```
paint.setStrokeWidth(5);
canvas.drawCircle(400, 125, 100, paint);
paint.setStrokeWidth(40);
canvas.drawCircle(650, 125, 100, paint);
```



> 线条宽度 0 和 1 的区别
>
> 默认情况下，线条宽度为 0，但你会发现，这个时候它依然能够画出线，线条的宽度为 1 像素。那么它和线条宽度为 1 有什么区别呢?
>
> 其实这个和后面要讲的一个「几何变换」有关：你可以为 Canvas 设置 Matrix 来实现几何变换（如放大、缩小、平移、旋转），在几何变换之后 Canvas 绘制的内容就会发生相应变化，包括线条也会加粗，例如 2 像素宽度的线条在 Canvas 放大 2 倍后会被以 4 像素宽度来绘制。而当线条宽度被设置为 0 时，它的宽度就被固定为 1 像素，就算 Canvas 通过几何变换被放大，它也依然会被以 1 像素宽度来绘制。Google 在文档中把线条宽度为 0 时称作「hairline mode（发际线模式）」。

2.3.2 setStrokeCap(Paint.Cap cap)

设置线头的形状。线头形状有三种： BUTT 平头、 ROUND 圆头、 SQUARE 方头。默认为 BUTT 。

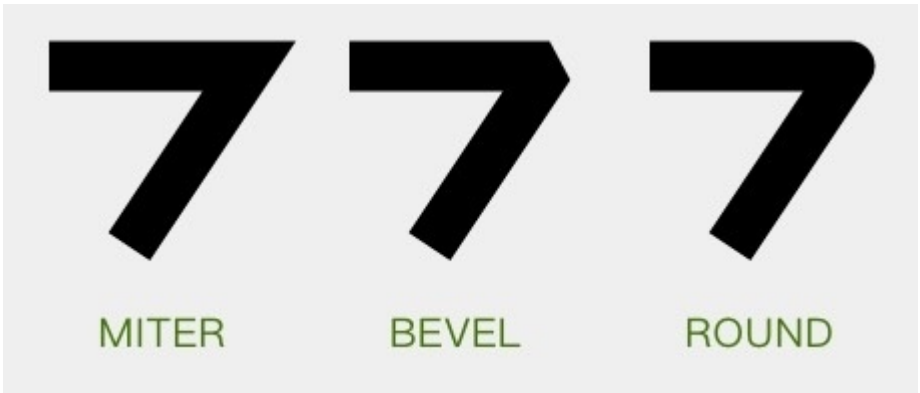
> 放出「平头」「圆头」「方头」这种翻译我始终有点纠结：既觉得自己翻译得简洁清晰尽显机智，同时又担心用词会不会有点太过通俗，让人觉得我不够高贵冷艳?

当线条的宽度是 1 像素时，这三种线头的表现是完全一致的，全是 1 个像素的点；而当线条变粗的时候，它们就会表现出不同的样子：

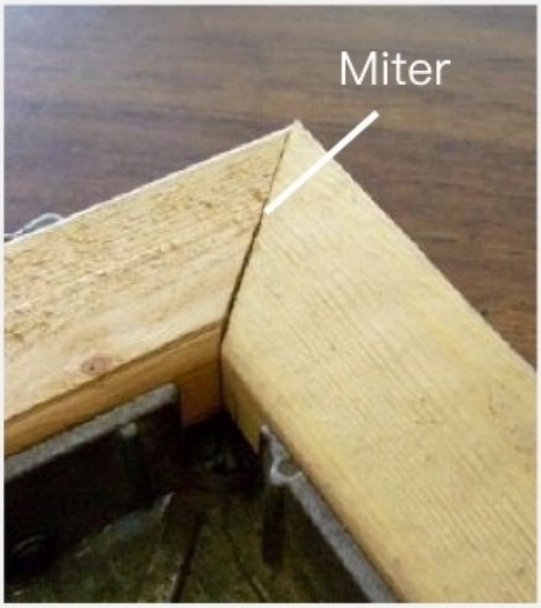
虚线是额外加的，虚线左边是线的实际长度，虚线右边是线头。有了虚线作为辅助，可以清楚地看出 `BUTT` 和 `SQUARE` 的区别。

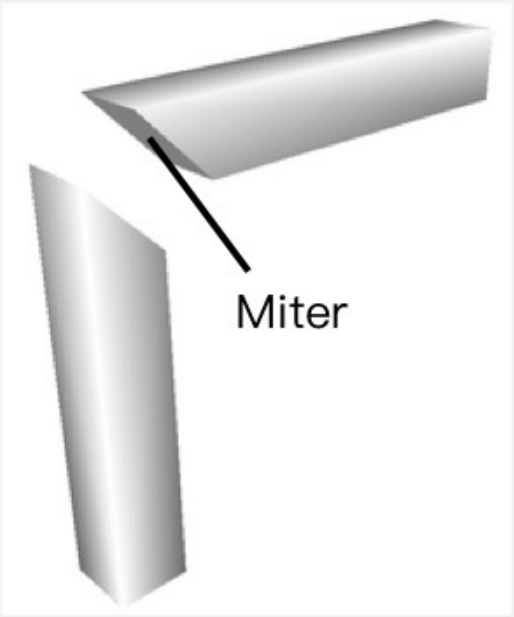
2.3.3 setStrokeJoin(Paint.Join join)

设置拐角的形状。有三个值可以选择：`MITER` 尖角、 `BEVEL` 平角和 `ROUND` 圆角。默认为 `MITER`。

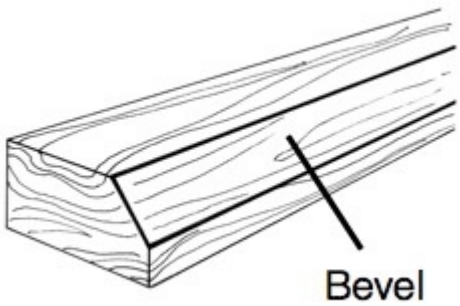


> 辅助理解：
>
> MITER 在现实中其实就是这玩意：





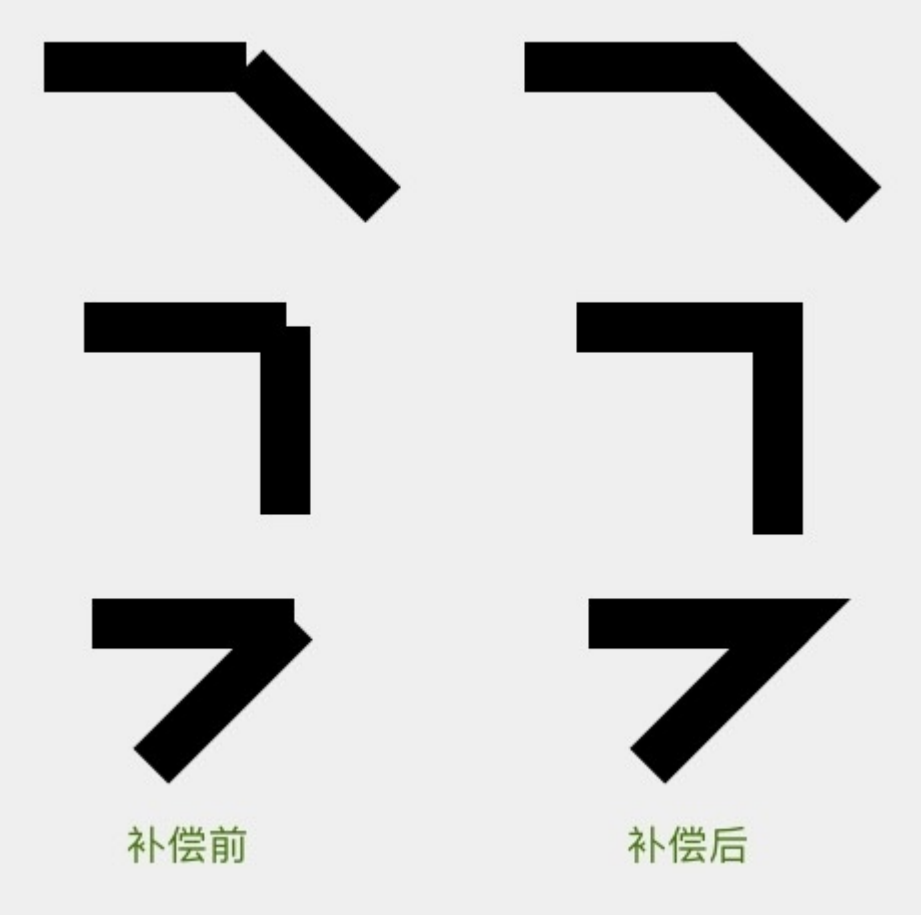
而 BEVEL 是这玩意：



2.3.4 setStrokeMiter(float miter)

这个方法是对于 `setStrokeJoin()` 的一个补充，它用于设置 `MITER` 型拐角的延长线的最大值。所谓「延长线的最大值」，是这么一回事：

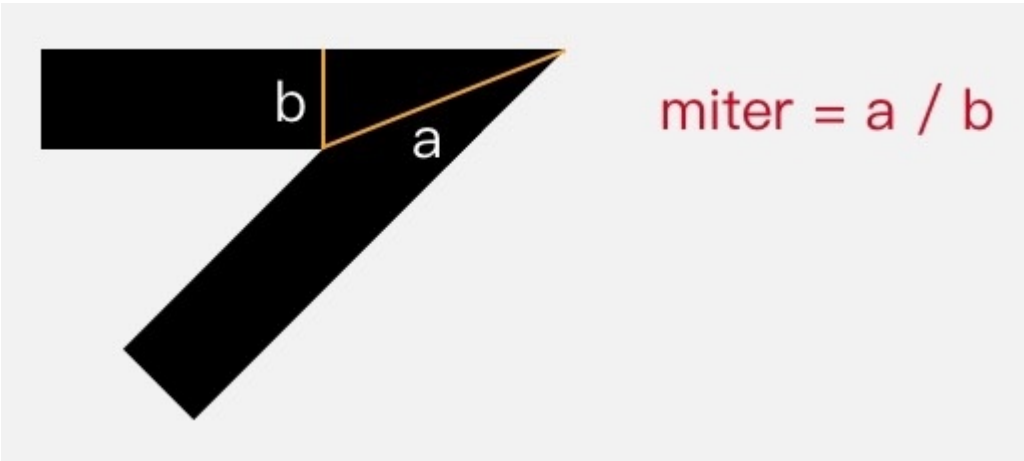
当线条拐角为 `MITER` 时，拐角处的外缘需要使用延长线来补偿：



而这种补偿方案会有一个问题：如果拐角的角度太小，就有可能由于出现连接点过长的情况。比如这样：

所以为了避免意料之外的过长的尖角出现， MITER 型连接点有一个额外的规则：当尖角过长时，自动改用 BEVEL 的方式来渲染连接点。例如上图的这个尖角，在默认情况下是不会出现的，而是会由于延长线过长而被转为 BEVEL 型连接点：

至于多尖的角属于过于尖，尖到需要转为使用 `BEVEL` 来绘制，则是由一个属性控制的，而这个属性就是 `setStrokeMiter(miter)` 方法中的 `miter` 参数。`miter` 参数是对于转角长度的限制，具体来讲，是指尖角的外缘端点和内部拐角的距离与线条宽度的比。也就是下面这两个长度的比：



用几何知识很容易得出这个比值的计算公式：如果拐角的大小为 θ ，那么这个比值就等于 1 / sin (θ / 2) 。

这个 miter limit 的默认值是 4，对应的是一个大约 29° 的锐角：

> 默认情况下，大于这个角的尖角会被保留，而小于这个夹角的就会被「削成平头」

> 所以，这个方法虽然名叫 `setStrokeMiter(miter)` ，但它其实设置的是「线条在 `Join` 类型为 `MITER` 时对于 `MITER` 的长度限制」。它的这个名字虽然短，但却存在一定的迷惑性，如果叫 `setStrokeJoinMiterLimit(limit)` 就更准确了。 Google 的工程师没有这么给它命名，大概也是不想伤害大家的手指吧，毕竟程序员何苦为难程序员。
>
> 
>

以上就是 4 个关于线条形状的方法： `setStrokeWidth(width)` `setStrokeCap(cap)` `setStrokeJoint(join)` 和 `setStrokeMiter(miter)` 。

2.4 色彩优化

`Paint` 的色彩优化有两个方法： `setDither(boolean dither)` 和 `setFilterBitmap(boolean filter)` 。它们的作用都是让画面颜色变得更加「顺眼」，但原理和使用场景是不同的。

2.4.1 setDither(boolean dither)

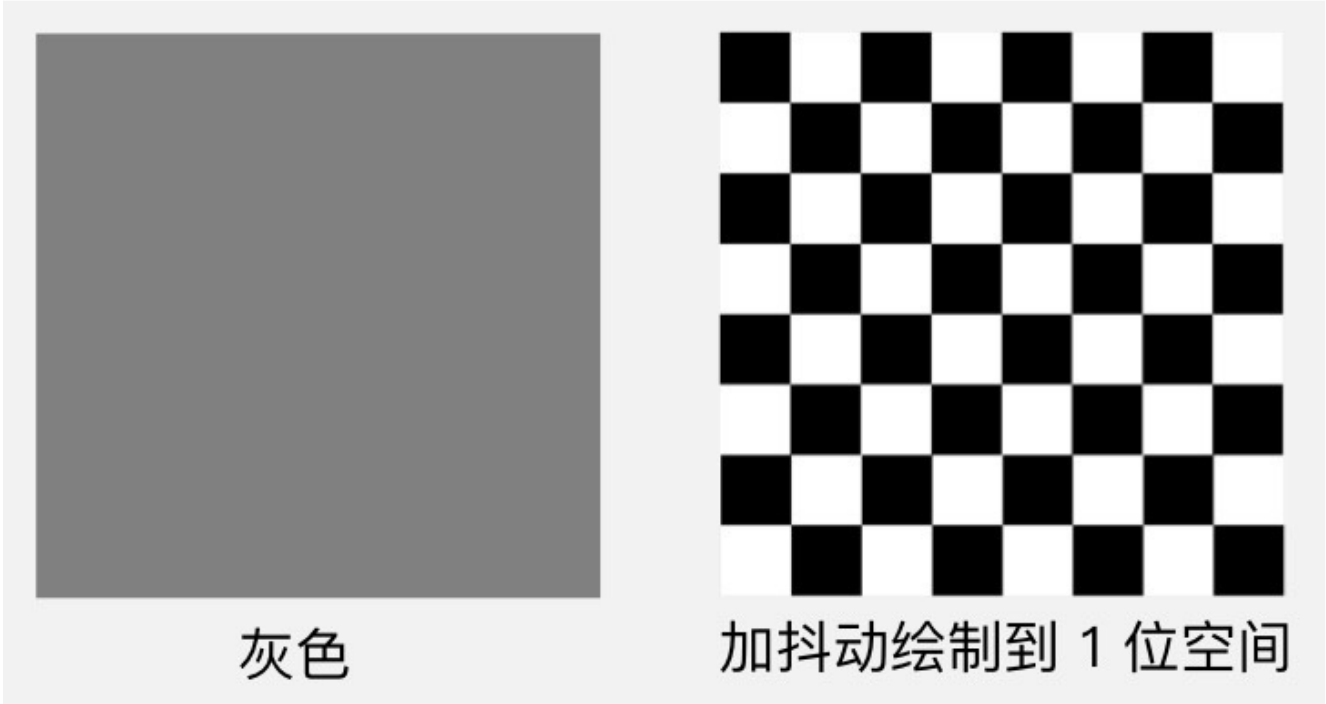
设置图像的抖动。

在介绍抖动之前，先来看一个猥琐男：

注意毛利小五郎脸上的红晕，它们并不是使用一片淡红色涂抹出来的，而是画了三道深色的红线。这三道深色红线放在脸上，给人的视觉效果就成了「淡淡的红晕」。

抖动的原理和这个类似。所谓抖动（注意，它就叫抖动，不是防抖动，也不是去抖动，有些人在翻译的时候自作主张地加了一个「防」字或者「去」字，这是不对的），是指把图像从较高色彩深度（即可用的颜色数）向较低色彩深度的区域绘制时，在图像中有意地插入噪点，通过有规律地扰乱图像来让图像对于肉眼更加真实的做法。

比如向 1 位色彩深度的区域中绘制灰色，由于 1 位深度只包含黑和白两种颜色，在默认情况下，即不加抖动的时候，只能选择向上或向下选择最接近灰色的白色或黑色来绘制，那么显示出来也只能是一片白或者一片黑。而加了抖动后，就可以绘制出让肉眼识别为灰色的效果了：



瞧，像上面这样，用黑白相间的方式来绘制，就可以骗过肉眼，让肉眼辨别为灰色了。

嗯？你说你看不出灰色，只看出黑白相间？没关系，那是因为像素颗粒太大，我把像素颗粒缩小，看到完整效果你就会发现变灰了：

这下变灰了吧?

什么，还没有变灰？那一定是你看图的姿势不对了。

不过，抖动可不只可以用在纯色的绘制。在实际的应用场景中，抖动更多的作用是在图像降低色彩深度绘制时，避免出现大片的色带与色块。效果盗一下维基百科的

图：

看着很牛逼对吧？确实很牛逼，而且在 Android 里使用起来也很简单，一行代码就搞定：

```
paint.setDither(true);
```

只要加这么一行代码，之后的绘制就是加抖动的了。

不过对于现在（2017年）而言， `setDither(dither)` 已经没有当年那么实用了，因为现在的 Android 版本的绘制，默认的色彩深度已经是 32 位的 `ARGB_8888` ，效果已经足够清晰了。只有当你向自建的 `Bitmap` 中绘制，并且选择 16 位色的 `ARGB_4444` 或者 `RGB_565` 的时候，开启它才会有比较明显的效果。

2.4.2 setFilterBitmap(boolean filter)

设置是否使用双线性过滤来绘制 `Bitmap` 。

图像在放大绘制的时候，默认使用的是最近邻插值过滤，这种算法简单，但会出现马赛克现象；而如果开启了双线性过滤，就可以让结果图像显得更加平滑。效果依然盗维基百科的图：

牛逼吧？而且它的使用同样也很简单：

```
paint.setFilterBitmap(true);
```

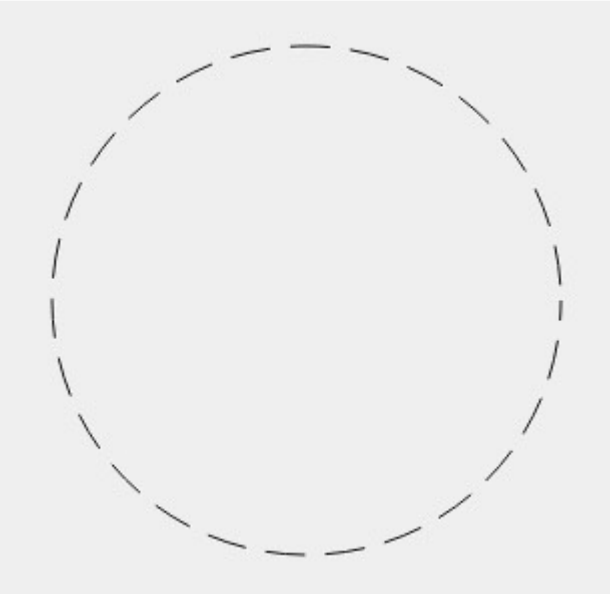
加上这一行，在放大绘制 `Bitmap` 的时候就会使用双线性过滤了。

以上就是 `Paint` 的两个色彩优化的方法： `setDither(dither)` ，设置抖动来优化色彩深度降低时的绘制效果； `setFilterBitmap(filterBitmap)` ，设置双线性过滤来优化 `Bitmap` 放大绘制的效果。

2.5 setPathEffect(PathEffect effect)

使用 `PathEffect` 来给图形的轮廓设置效果。对 `Canvas` 所有的图形绘制有效，也就是 `drawLine()` `drawCircle()` `drawPath()` 这些方法。大概像这样：

```
PathEffect pathEffect = new DashPathEffect(new float[]{10, 5}, 10);
paint.setPathEffect(pathEffect);

...

canvas.drawCircle(300, 300, 200, paint);
```

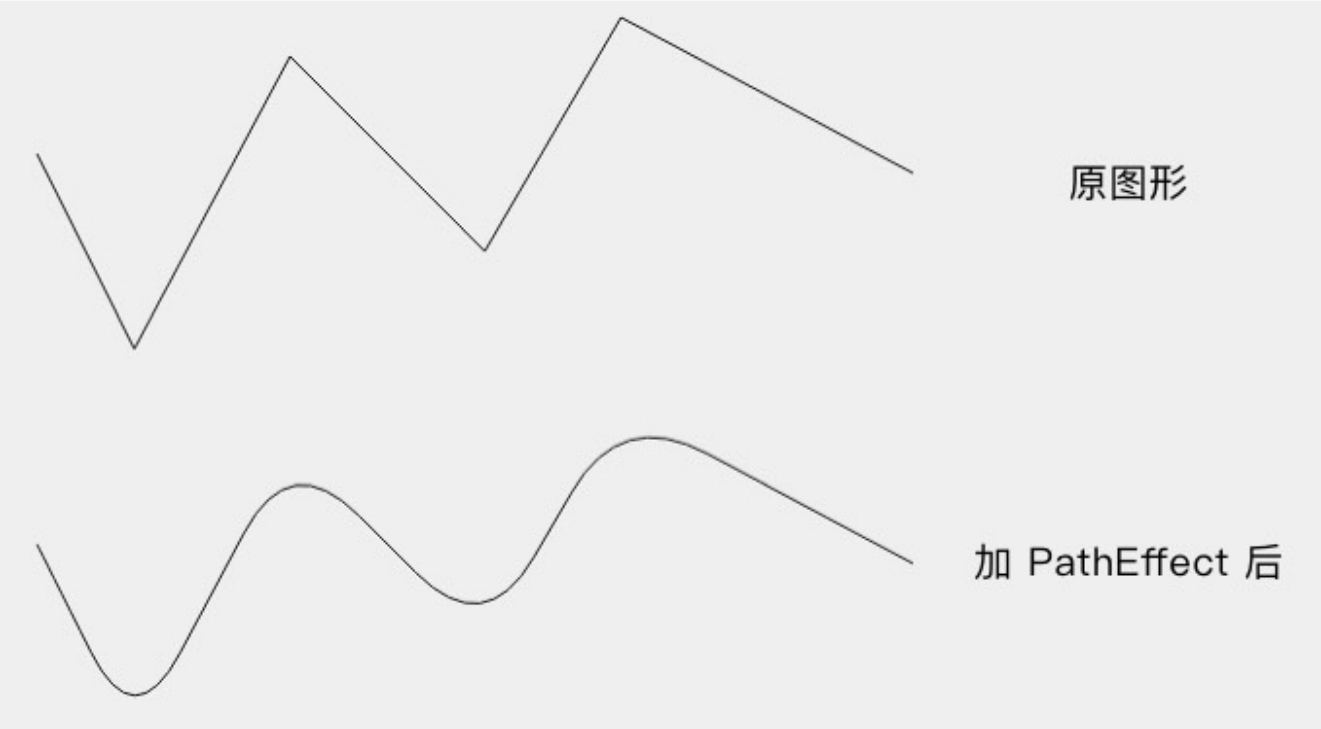


下面就具体说一下 Android 中的 6 种 `PathEffect`。`PathEffect` 分为两类，单一效果的 `CornerPathEffect` `DiscretePathEffect` `DashPathEffect` `PathDashPathEffect`，和组合效果的 `SumPathEffect` `ComposePathEffect`。

2.5.1 CornerPathEffect

把所有拐角变成圆角。

```
PathEffect pathEffect = new CornerPathEffect(20);
paint.setPathEffect(pathEffect);

...

canvas.drawPath(path, paint);
```



它的构造方法 `CornerPathEffect(float radius)` 的参数 `radius` 是圆角的半径。

2.5.2 DiscretePathEffect

把线条进行随机的偏离，让轮廓变得乱七八糟。乱七八糟的方式和程度由参数决定。

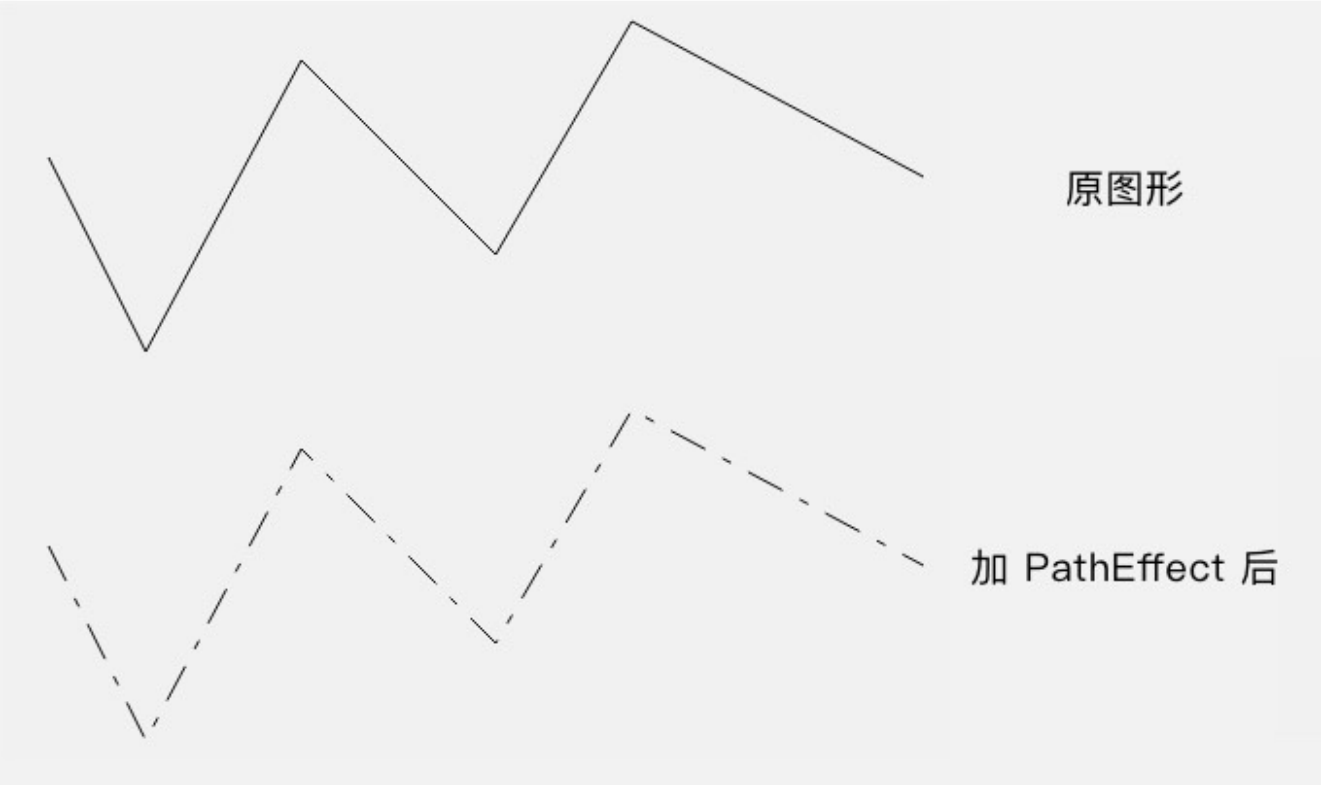
```
PathEffect pathEffect = new DiscretePathEffect(20, 5);
paint.setPathEffect(pathEffect);

...

canvas.drawPath(path, paint);
```

`DiscretePathEffect` 具体的做法是，把绘制改为使用定长的线段来拼接，并且在拼接的时候对路径进行随机偏离。它的构造方法 `DiscretePathEffect(float segmentLength, float deviation)` 的两个参数中， `segmentLength` 是用来拼接的每个线段的长度， `deviation` 是偏离量。这两个值设置得不一样，显示效果也会不一样，具体的你自己多试几次就明白了，这里不再贴更多的图。

2.5.3 DashPathEffect

使用虚线来绘制线条。

```
PathEffect pathEffect = new DashPathEffect(new float[]{20, 10, 5, 1
paint.setPathEffect(pathEffect);

...

canvas.drawPath(path, paint);
```

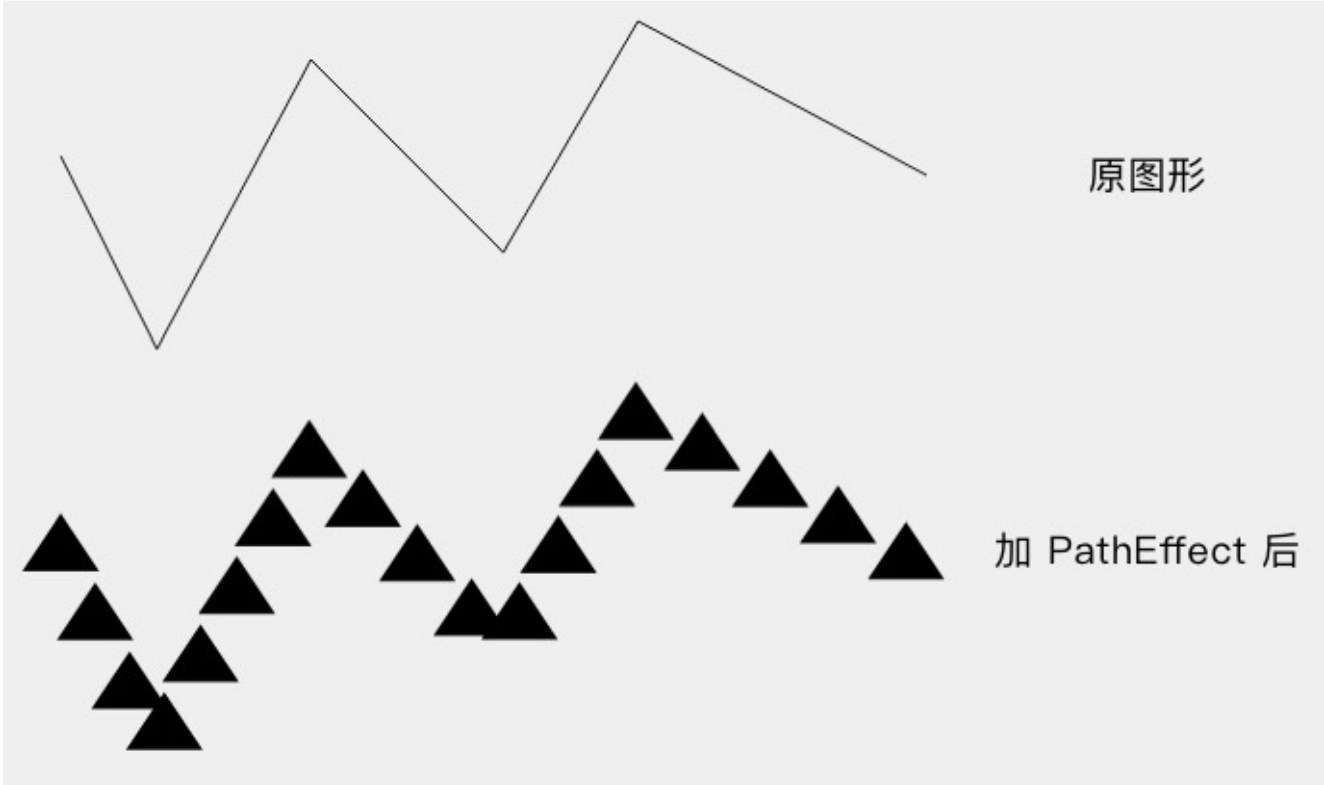


它的构造方法 `DashPathEffect(float[] intervals, float phase)` 中， 第一个参数 `intervals` 是一个数组，它指定了虚线的格式：数组中元素必须为偶数（最少是 2 个），按照「画线长度、空白长度、画线长度、空白长度」……的顺序排列，例如上面代码中的 `20, 5, 10, 5` 就表示虚线是按照「画 20 像素、空 5 像素、画 10 像素、空 5 像素」的模式来绘制；第二个参数 `phase` 是虚线的偏移量。

2.5.4 PathDashPathEffect

这个方法比 `DashPathEffect` 多一个前缀 `Path` ，所以顾名思义，它是使用一个 `Path` 来绘制「虚线」。具体看图吧：

```
Path dashPath = ...; // 使用一个三角形来做 dash
PathEffect pathEffect = new PathDashPathEffect(dashPath, 40, 0,
        PathDashPathEffectStyle.TRANSLATE);
paint.setPathEffect(pathEffect);

...

canvas.drawPath(path, paint);
```

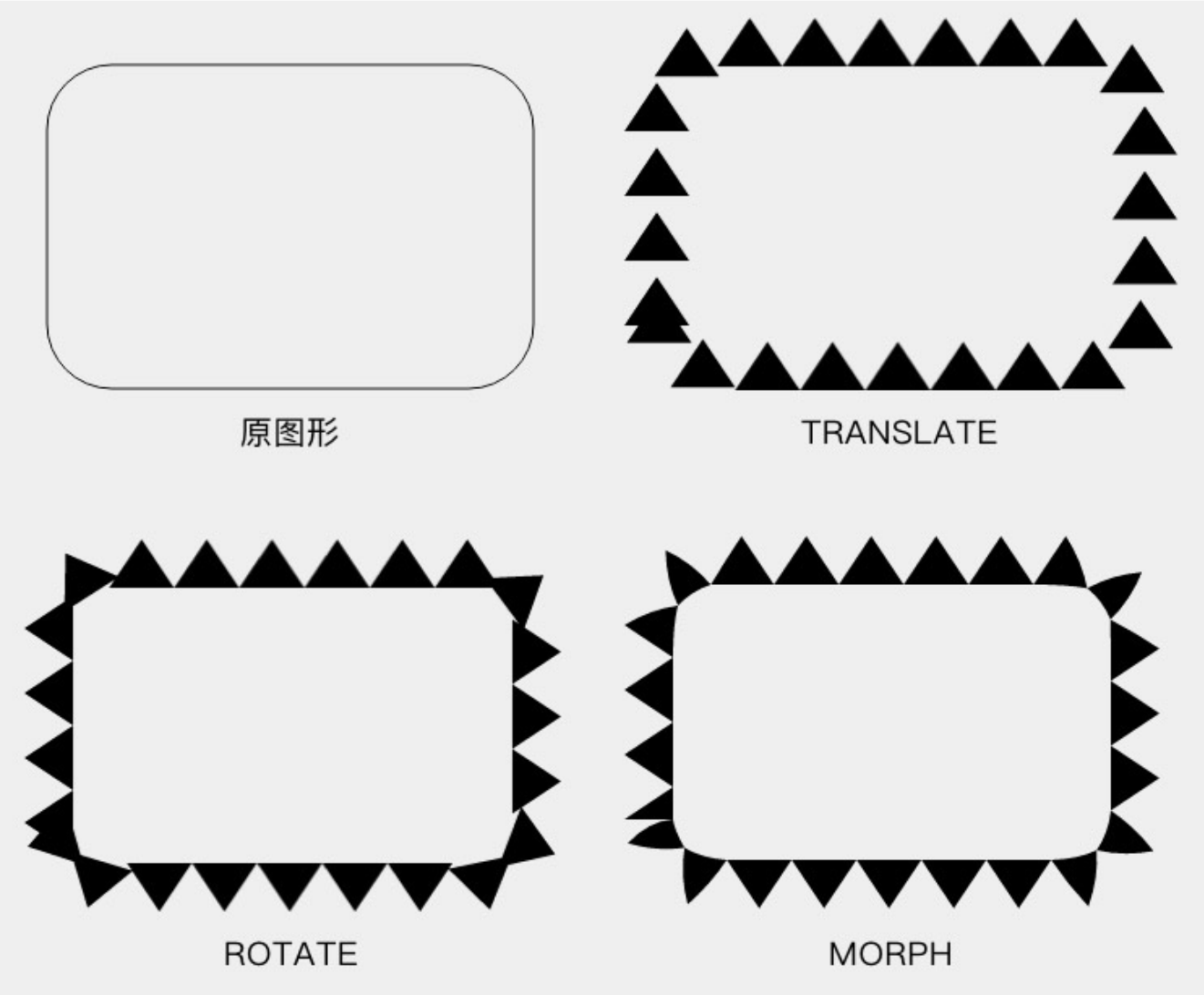


它的构造方法

`PathDashPathEffect(Path shape, float advance, float phase, PathDashPathE`

中， `shape` 参数是用来绘制的 `Path` ； `advance` 是两个相邻的 `shape` 段之间的间隔，不过注意，这个间隔是两个 `shape` 段的起点的间隔，而不是前一个的终点和后一个的起点的距离； `phase` 和 `DashPathEffect` 中一样，是虚线的偏移；最后一个参数 `style` ，是用来指定拐弯改变的时候 `shape` 的转换方式。 `style` 的类型为 `PathDashPathEffect.Style` ，是一个 `enum` ，具体有三个值：

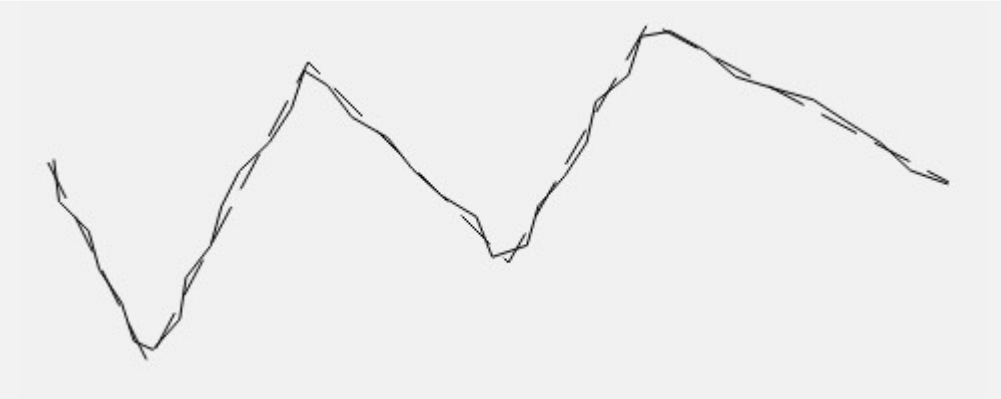
- `TRANSLATE` ： 位移
- `ROTATE` ： 旋转
- `MORPH` ： 变体



2.5.5 SumPathEffect

这是一个组合效果类的 `PathEffect` 。它的行为特别简单，就是分别按照两种 `PathEffect` 分别对目标进行绘制。

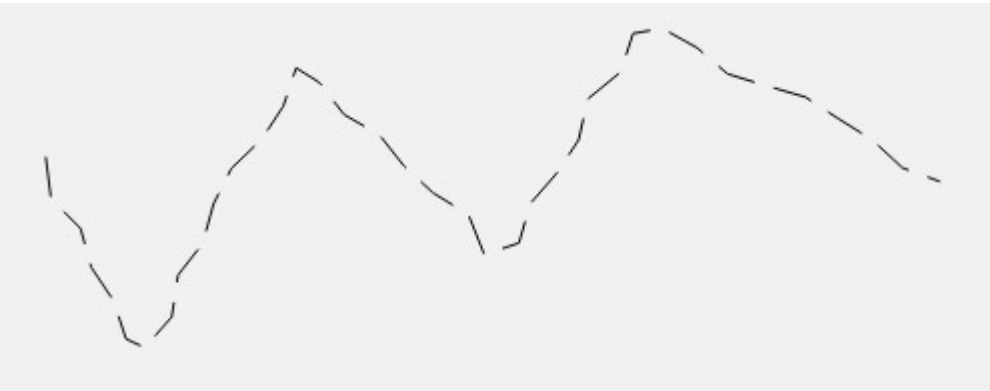
```
PathEffect dashEffect = new DashPathEffect(new float[]{20, 10}, 0);
PathEffect discreteEffect = new DiscretePathEffect(20, 5);
pathEffect = new SumPathEffect(dashEffect, discreteEffect);

...

canvas.drawPath(path, paint);
```



2.5.6 ComposePathEffect

这也是一个组合效果类的 `PathEffect` 。不过它是先对目标 `Path` 使用一个 `PathEffect`，然后再对这个改变后的 `Path` 使用另一个 `PathEffect` 。

```
PathEffect dashEffect = new DashPathEffect(new float[]{20, 10}, 0);
PathEffect discreteEffect = new DiscretePathEffect(20, 5);
pathEffect = new ComposePathEffect(dashEffect, discreteEffect);

...

canvas.drawPath(path, paint);
```



它的构造方法

`ComposePathEffect(PathEffect outerpe, PathEffect innerpe)` 中的两个 `PathEffect` 参数， `innerpe` 是先应用的， `outerpe` 是后应用的。所以上面的代码就是「先偏离，再变虚线」。而如果把两个参数调换，就成了「先变虚线，再偏离」。至于具体的视觉效果……我就不贴图了，你自己试试看吧！

上面这些就是 Paint 中的 6 种 PathEffect。它们有的是有独立效果的，有的是用来组合不同的 PathEffect 的，功能各不一样。

> 注意： PathEffect 在有些情况下不支持硬件加速，需要关闭硬件加速才能正常使用：
>
> 1. Canvas.drawLine() 和 Canvas.drawLines() 方法画直线时， setPathEffect() 是不支持硬件加速的；
> 2. PathDashPathEffect 对硬件加速的支持也有问题，所以当使用 PathDashPathEffect 的时候，最好也把硬件加速关了。

剩下的两个效果类方法： setShadowLayer() 和 setMaskFilter()，它们和前面的效果类方法有点不一样：它们设置的是「附加效果」，也就是基于在绘制内容的额外效果。

2.6 setShadowLayer(float radius, float dx, float dy, int shadowColor)

在之后的绘制内容下面加一层阴影。

```
paint.setShadowLayer(10, 0, 0, Color.RED);

...

canvas.drawText(text, 80, 300, paint);
```

效果就是上面这样。方法的参数里， radius 是阴影的模糊范围； dx dy 是阴影的偏移量； shadowColor 是阴影的颜色。

如果要清除阴影层，使用 clearShadowLayer()。

注意：

- 在硬件加速开启的情况下， `setShadowLayer()` 只支持文字的绘制，文字之外的绘制必须关闭硬件加速才能正常绘制阴影。

- 如果 `shadowColor` 是半透明的，阴影的透明度就使用 `shadowColor` 自己的透明度；而如果 `shadowColor` 是不透明的，阴影的透明度就使用 `paint` 的透明度。

2.7 setMaskFilter(MaskFilter maskfilter)

为之后的绘制设置 `MaskFilter` 。上一个方法 `setShadowLayer()` 是设置的在绘制层下方的附加效果；而这个 `MaskFilter` 和它相反，设置的是在绘制层上方的附加效果。

> 到现在已经有两个 `setXxxFilter(filter)` 了。前面有一个 `setColorFilter(filter)` ，是对每个像素的颜色进行过滤；而这里的 `setMaskFilter(filter)` 则是基于整个画面来进行过滤。

`MaskFilter` 有两种： `BlurMaskFilter` 和 `EmbossMaskFilter` 。

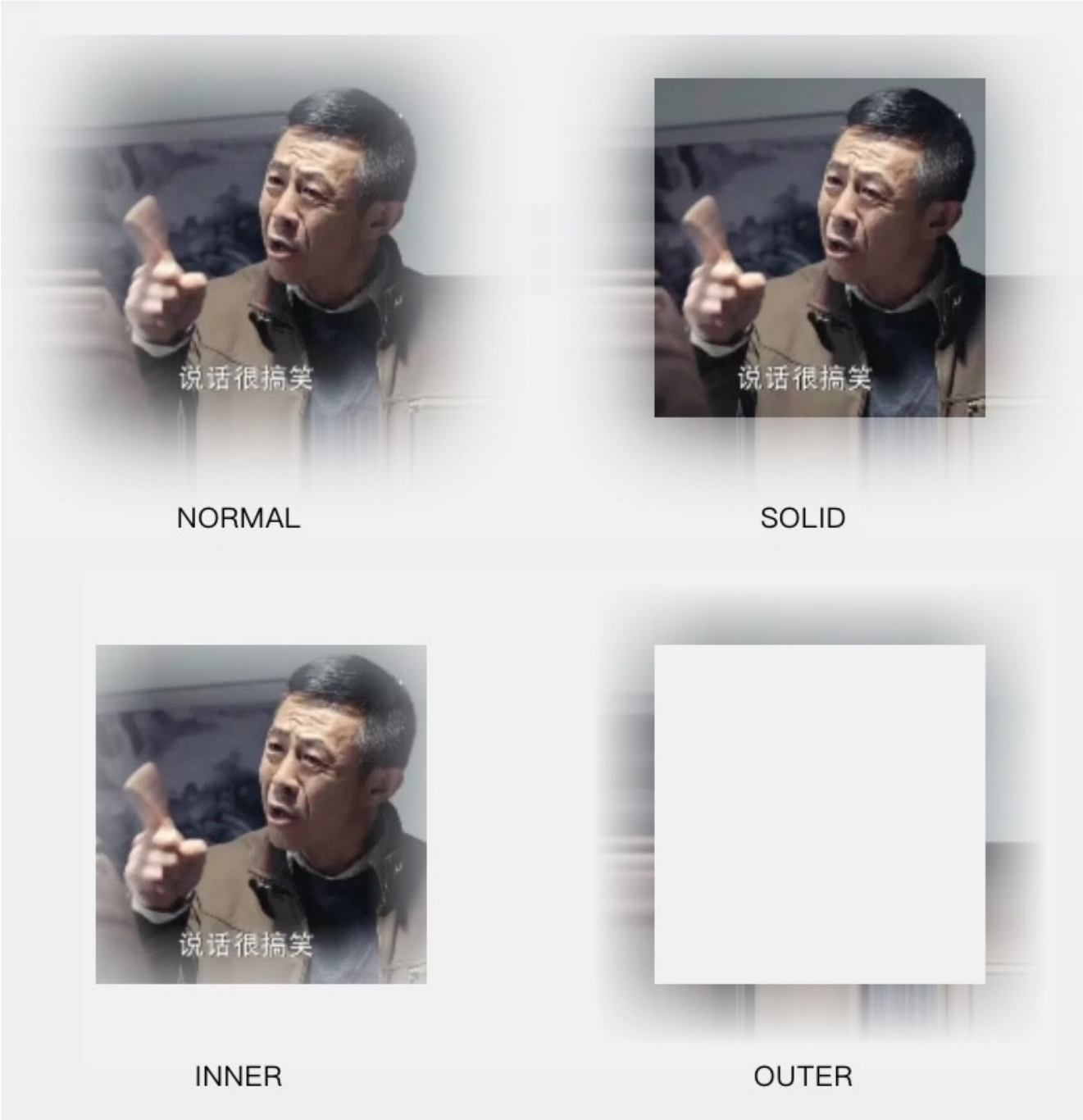
2.7.1 BlurMaskFilter

模糊效果的 `MaskFilter` 。

```
paint.setMaskFilter(new BlurMaskFilter(50, BlurMaskFilter.Blur.NORM

...

canvas.drawBitmap(bitmap, 100, 100, paint);
```

它的构造方法 `BlurMaskFilter(float radius, BlurMaskFilter.Blur style)` 中， `radius` 参数是模糊的范围， `style` 是模糊的类型。一共有四种：

- `NORMAL`：内外都模糊绘制
- `SOLID`：内部正常绘制，外部模糊
- `INNER`：内部模糊，外部不绘制
- `OUTER`：内部不绘制，外部模糊（什么鬼？）



2.7.2 EmbossMaskFilter

浮雕效果的 `MaskFilter` 。

```
paint.setMaskFilter(new EmbossMaskFilter(new float[]{0, 1, 1}, 0.2f

...

canvas.drawBitmap(bitmap, 100, 100, paint);
```

原图　　设置 EmbossMaskFilter 后

它的构造方法

`EmbossMaskFilter(float[] direction, float ambient, float specular, float`

的参数里， `direction` 是一个 3 个元素的数组，指定了光源的方向； `ambient` 是环境光的强度，数值范围是 0 到 1； `specular` 是炫光的系数； `blurRadius` 是应用光线的范围。

不过由于我没有在项目中使用过 `EmbossMaskFilter`，对它的每个参数具体调节方式并不熟，你有兴趣的话自己研究一下吧。

2.8 获取绘制的 Path

这是效果类的最后一组方法，也是效果类唯一的一组 `get` 方法。

这组方法做的事是，根据 `paint` 的设置，计算出绘制 `Path` 或文字时的**实际 Path**。

这里你可能会冒出两个问题：

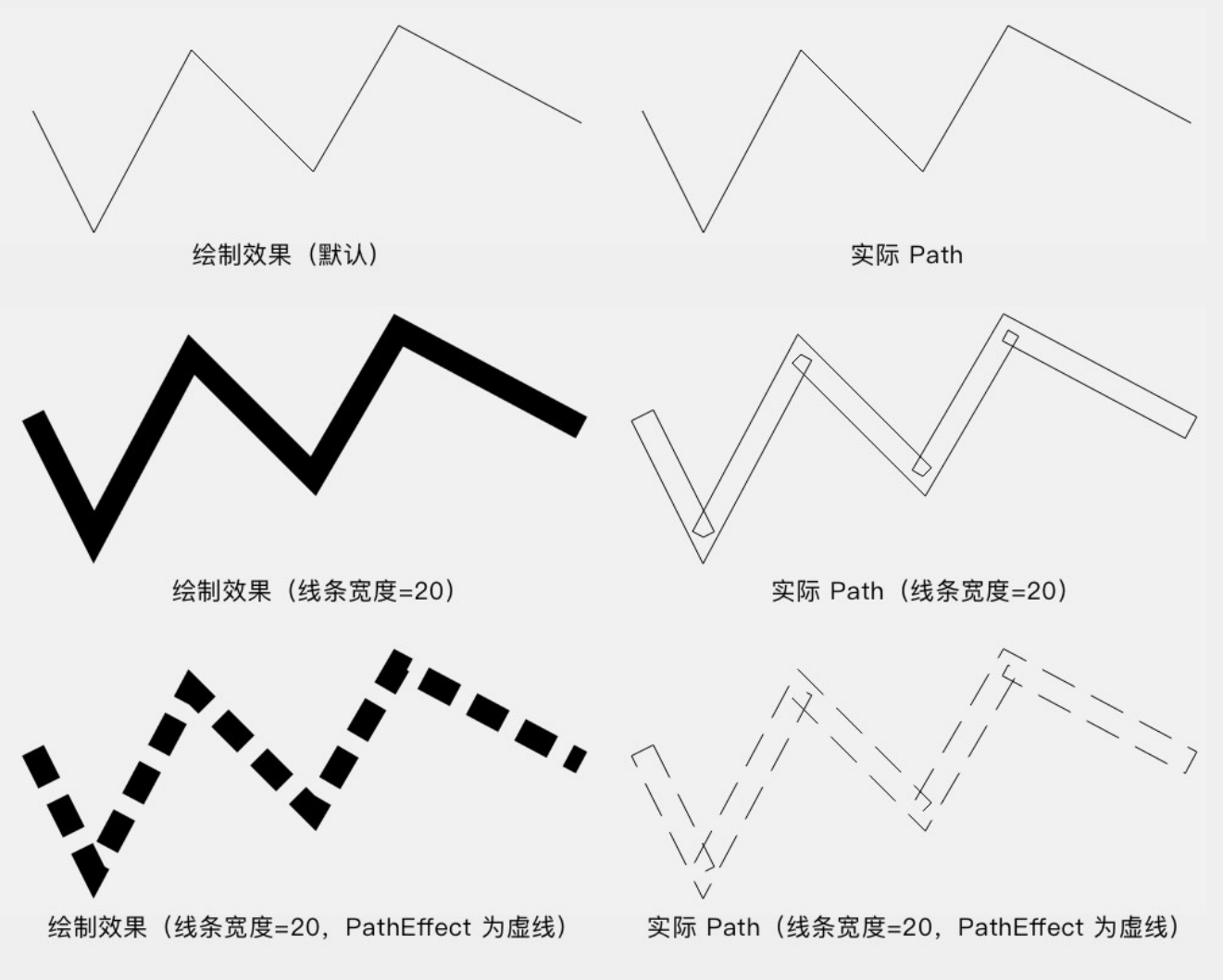
1. 什么叫「实际 `Path`」？ `Path` 就是 `Path`，这加上个「实际」是什么意思？
2. 文字的 `Path` ？文字还有 `Path`？

这两个问题（咦好像有四个问号）的答案就在后面的内容里。

2.8.1 getFillPath(Path src, Path dst)

首先解答第一个问题：「实际 Path」。所谓实际 Path，指的就是 drawPath() 的绘制内容的轮廓，要算上线条宽度和设置的 PathEffect。

默认情况下（线条宽度为 0、没有 PathEffect），原 Path 和实际 Path 是一样的；而在线条宽度不为 0（并且模式为 STROKE 模式或 FLL_AND_STROKE），或者设置了 PathEffect 的时候，实际 Path 就和原 Path 不一样了：



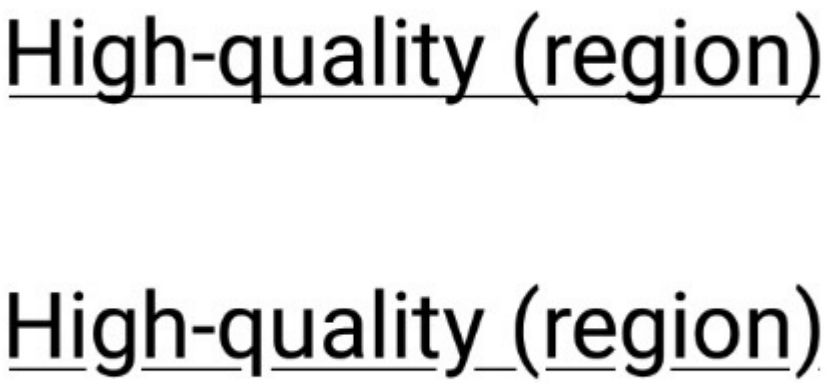
> 看明白了吗？

通过 getFillPath(src, dst) 方法就能获取这个实际 Path。方法的参数里，src 是原 Path，而 dst 就是实际 Path 的保存位置。
getFillPath(src, dst) 会计算出实际 Path，然后把结果保存在 dst 里。

2.8.2 getTextPath(String text, int start, int end, float x, float y, Path path) / getTextPath(char[] text, int index, int count, float x, float y, Path path)

这里就回答第二个问题：「文字的 Path」。文字的绘制，虽然是使用 Canvas.drawText() 方法，但其实在下层，文字信息全是被转化成图形，对图形进行绘制的。 getTextPath() 方法，获取的就是目标文字所对应的 Path 。这个就是所谓「文字的 Path」。

这两个方法， getFillPath() 和 getTextPath() ，就是获取绘制的 Path 的方法。之所以把它们归类到「效果」类方法，是因为它们主要是用于图形和文字的装饰效果的位置计算，比如自定义的下划线效果。



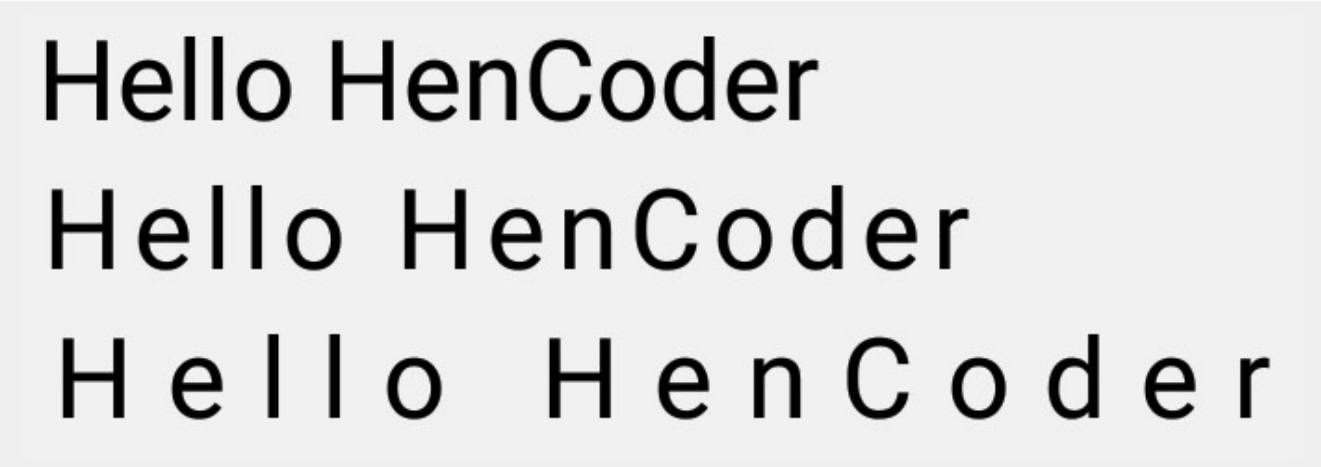
到此为止， Paint 的第二类方法——效果类，就也介绍完了。

3 drawText() 相关

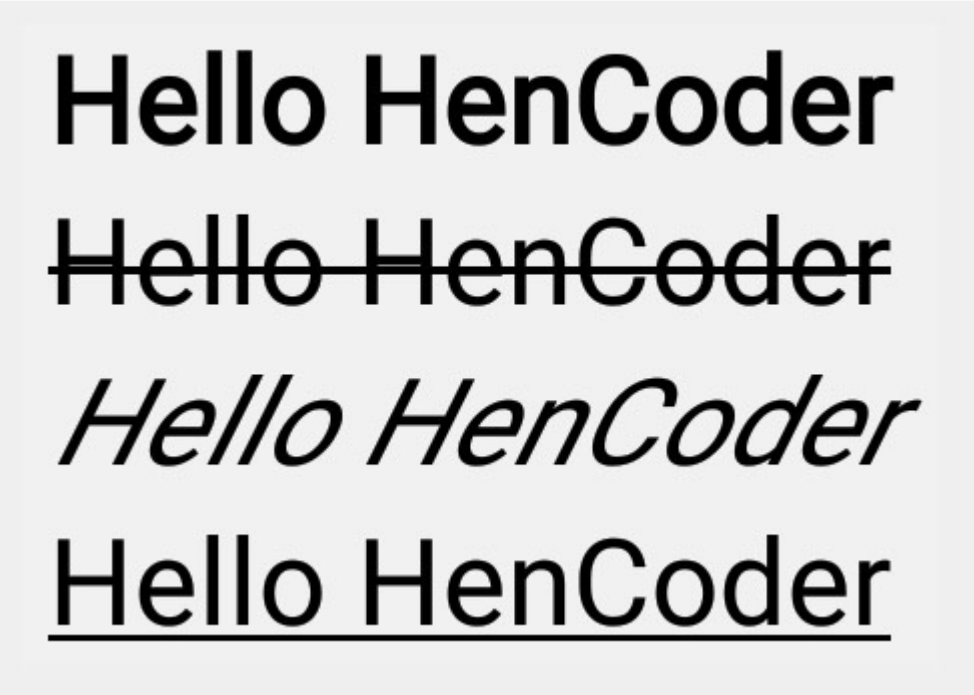
Paint 有些设置是文字绘制相关的，即和 drawText() 相关的。

比如设置文字大小：

比如设置文字间隔：



比如设置各种文字效果：



除此之外，`Paint` 还有很多与文字绘制相关的设置或计算的方法，非常详细。不过由于太详细了，相关方法太多了（`Paint` 超过一半的方法都是 `drawText()` 相关的，算不算多？），如果放在这里讲它们的话，内容会显得有点过量。所以这一节我就不讲它们了，把它们放在下一节里单独讲。

4 初始化类

这一类方法很简单，它们是用来初始化 `Paint` 对象，或者是批量设置 `Paint` 的多个属性的方法。

4.1 reset()

重置 `Paint` 的所有属性为默认值。相当于重新 `new` 一个，不过性能当然高一些啦。

4.2 set(Paint src)

把 `src` 的所有属性全部复制过来。相当于调用 `src` 所有的 `get` 方法，然后调用这个 `Paint` 的对应的 `set` 方法来设置它们。

4.3 setFlags(int flags)

批量设置 flags。相当于依次调用它们的 `set` 方法。例如：

```
paint.setFlags(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
```

这行代码，和下面这两行是等价的：

```
paint.setAntiAlias(true);
paint.setDither(true);
```

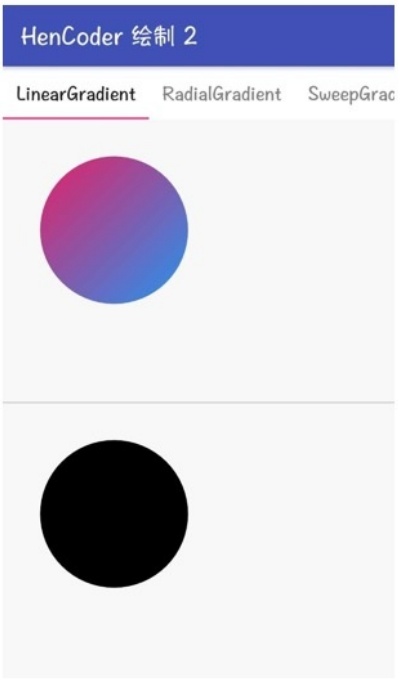
`setFlags(flags)` 对应的 `get` 方法是 `int getFlags()`。

好了，这些就是 `Paint` 的四类方法：**颜色类**、**效果类**、**文字绘制相关**以及**初始化类**。其中颜色类、效果类和初始化类都已经在这节里面讲过了，剩下的一类——文字绘制类，下一节单独讲。

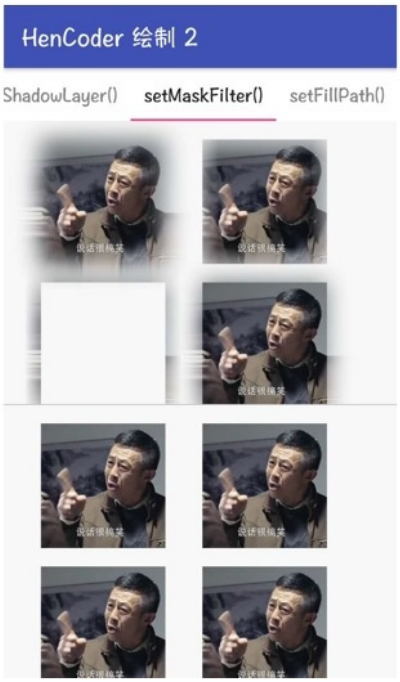
最后再强调一遍：**这期的内容没必要全部背会，只要看懂、理解，记住有这么个东西就行了。以后在用到的时候，再拐回来翻一翻就行了。**

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：
HenCoderPracticeDraw2


HenCoder 绘制 2
LinearGradient
RadialGradient
SweepGrac


HenCoder 绘制 2
trixColorFilter
setXfermode()
setStrokeCap


HenCoder 绘制 2
ShadowLayer()
setMaskFilter()
setFillPath()