

1. 自定义控件其实很简单(1)

自定义 View，很多初学 Android 的童鞋听到这么一句话绝逼是一脸膜拜！因为在很多初学者眼里，能够自己去画一个 View 绝逼是一件很屌很 Cool 的事！但是，同样而言，自定义 View 对初学者来说却往往可望而不可及，可望是因为看了很多自定义 View 的源码好像并不难，有些自定义 View 甚至不足百行代码，不可及呢是因为即便看了很多文章很多类似的源码依然写不出一个霸气的 View 来。这时会有很多前辈告诉你多看看 View 类的源码，看看 View 类里是如何去处理这些绘制逻辑的，如果你去看了我只能说你是个很好学很有求知欲的孩纸，了解原理是好事，但是并非凡事都要去刨根问底的！如果你做 Android 开发必须要把 Android 全部源码弄懂，我只能呵呵了！你还不如去写一个系统实在对吧！同样的道理，写一个自定义 View 你非要去花巨量时间研究各类源码是不值得提倡的，当然哥没有否定追究原理的意义所在，只是对于一个普通的开发者你没有必要去深究一些不该值得你关心的东西，特别是一个具有良好面向对象思维的猿。举个生活中简单的例子，大家都用过吹风，吹风一般都会提供三个档位：关、冷风、热风对吧，你去买吹风人家只会告诉你这吹风三个档位分别是什么功能，我相信没有哪个傻逼买吹风的会把吹风拆开、电机写下来一个一个地跟你解说那是啥玩意吧！同样的，我们自定义 View 其实 Android 已经提供了大量类似吹风档位的方法，你只管在里面做你想做的事情就可，至于 Android 本身内部是如何实现的，你压根不用去管！用官方文档的原话来说就是：Just do you things! 初学者不懂如何去自定义 View 并非是不懂其原理，而是不懂这些类似“档位”的方法！

好了，扯了这么多废话！我们还是先步入正题，来看看究竟自定义 View 是如何实现的！在 Android 中自定义一个 View 类并定是直接继承 View 类或者 View 类的子类比如 TextView、Button 等等，这里呢我们也依葫芦画瓢直接继承 View 自定义一个 View 的子类 CustomView：

[java] view plain copy print?

```
1. public class CustomView extends View {  
2. }
```

在 View 类中没有提供无参的构造方法，这时我们的 IDE 会提示我们你得明确地声明一个和带有父类一样签名列表的构造方法：



这时我们点击“Add constructor CustomView(Context context)”，IDE 就会自动为我们生成一个带有 Context 类型签名的构造方法：

[java] view plaincopyprint?

```
1. public class CustomView extends View {  
2.     public CustomView(Context context) {  
3.         super(context);  
4.     }  
5. }
```

Context 是什么你不用管，只管记住它包含了许多各种不同的信息穿梭于 Android 中各组件、控件等等之间，说得不恰当点就是一个装满信息的信使，Android 需要它从里面获取需要的信息。

这样我们就定义了一个属于自己的自定义 View，我们尝试将它添加到 Activity：

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {  
2.     private LinearLayout llRoot;// 界面的根布局  
3.  
4.     @Override  
5.     protected void onCreate(Bundle savedInstanceState) {  
6.         super.onCreate(savedInstanceState);  
7.         setContentView(R.layout.activity_main);  
8.  
9.         llRoot = (LinearLayout) findViewById(R.id.main_root_ll);  
10.        llRoot.addView(new CustomView(this));  
11.    }  
12. }
```

运行后发现什么也没有，空的！因为我们的 CustomView 本来就什么都没有！但是添加到我们的界面后没有什么问题对吧！Perfect！那我们再直接在 xml 文档中引用它呢：

[html] view plaincopyprint?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
2.     android:id="@+id/main_root_ll"  
3.     android:layout_width="match_parent"  
4.     android:layout_height="match_parent"  
5.     android:orientation="vertical" >  
6.  
7.     <com.sigestudio.customviewdemo.views.CustomButton  
8.         android:layout_width="match_parent"
```

```
9.         android:layout_height="match_parent" />
10.
11. </LinearLayout>
```

这时我们还原 Activity 中的代码:

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
2.
3.     @Override
4.     protected void onCreate(Bundle savedInstanceState) {
5.         super.onCreate(savedInstanceState);
6.         setContentView(R.layout.activity_main);
7.     }
8. }
```

再次运行后发现 IDE 报错了:

```
java.lang.RuntimeException: Unable to start activity ComponentInfo{com.aigestudio.customviewdemo/com.aigestudio.customviewdemo.activities.MainActivity}: android.view.InflateException: Binary XML file line #7: Error inflating class com.sigestudio.customviewdemo.views.CustomView

Caused by: java.lang.NoSuchMethodException: <init> [class android.content.Context, interface android.util.AttributeSet]
```

大致意思是无法解析我们的 CustomView 类找不到方法, 为什么呢? 我们在 xml 文件引用我们的 CustomView 类时为其指定了两个 android 自带的两个属性: layout_width 和 layout_height, 当我们需要使用类似的属性(比如更多的什么 id 啊、padding 啊、margin 啊之类) 时必须在自定义 View 的构造方法中添加一个 AttributeSet 类型的签名来解析这些属性:

[java] view plaincopyprint?

```
1. public class CustomView extends View {
2.     public CustomView(Context context) {
3.         super(context);
4.     }
5.
6.     public CustomView(Context context, AttributeSet attrs) {
7.         super(context, attrs);
8.     }
9. }
```

再次运行发现一切又恢复了正常。现在我们来往我们的 View 里画点东西，毕竟自定义 View 总得有点什么才行对吧！Android 给我们提供了一个 `onDraw(Canvas canvas)` 方法来让我们绘制自己想要的东西：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     super.onDraw(canvas);
4. }
```

我们想要画些什么直接在这个方法里面画即可，在现实世界中，我们画画需要两样东西：笔（或者任何能涂画的东西）和纸（或者任何能被画的东西），同样地，Android 也给我们提供了这两样东西：`Paint` 和 `Canvas`，一个是画笔而另一个呢当然是画布啦~~，我们可以看到在 `onDraw` 方法中，画布 `Canvas` 作为签名被传递进来，也就是说这个画布是 Android 为我们准备好的，不需要你去管，当然你也可以自定义一张画布在上面绘制自己的东西并将其传递给父类，但是一般我们不建议这样去做！有人会问这画布是怎么来的？在这里我不想跟大家深究其原理，否则长篇大论也过于繁琐打击各位菜鸟哥的学习兴趣。但是我可以这样跟大家说，如果在一张大的画布（界面）上面有各种各样小的画布（界面中的各种控件），那么这些小的画布该如何确定其大小呢？自己去想哈哈！

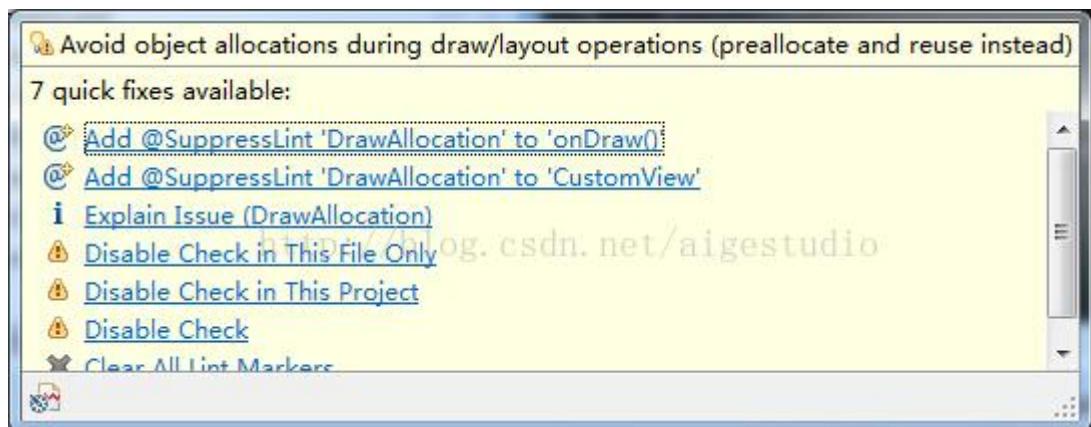
草！又跑题了！

画布有了，差一支画笔，简单！我们 `new` 一个呗！程序猿的好处就在万事万物都可以自己 `new`！女朋友也能自己 `new`，随便 `new`！！~~~：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     super.onDraw(canvas);
4.     Paint paint = new Paint();
5.     paint.setAntiAlias(true);
6. }
```

实例化了一个 `Paint` 对象后我们为其设置了抗锯齿（一种让图像边缘显得更圆滑光泽动感的碉堡算法）：`setAntiAlias(true)`，但是我们发现这是 IDE 又警告了！！！说什么“`Avoid object allocations during draw/layout operations (preallocate and reuse instead)`”：



Why? Why? 说白了就是不建议你在 `draw` 或者 `layout` 的过程中去实例化对象! 为啥? 因为 `draw` 或 `layout` 的过程有可能是一个频繁重复执行的过程, 我们知道 `new` 是需要分配内存空间的, 如果在一个频繁重复的过程中去大量地 `new` 对象内存爆不爆我不知道, 但是浪费内存那是肯定的! 所以 Android 不建议我们在这两个过程中去实例化对象。既然都这样说了我们就改改呗:

[java] view plaincopyprint?

```
1. public class CustomView extends View {
2.     private Paint mPaint;
3.
4.     public CustomView(Context context) {
5.         this(context, null);
6.     }
7.
8.     public CustomView(Context context, AttributeSet attrs) {
9.         super(context, attrs);
10.
11.        // 初始化画笔
12.        initPaint();
13.    }
14.
15. /**
16. * 初始化画笔
17. */
18. private void initPaint() {
19.     // 实例化画笔并打开抗锯齿
20.     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
21. }
22.
23. @Override
24. protected void onDraw(Canvas canvas) {
25.     super.onDraw(canvas);
```

```
26. }
```

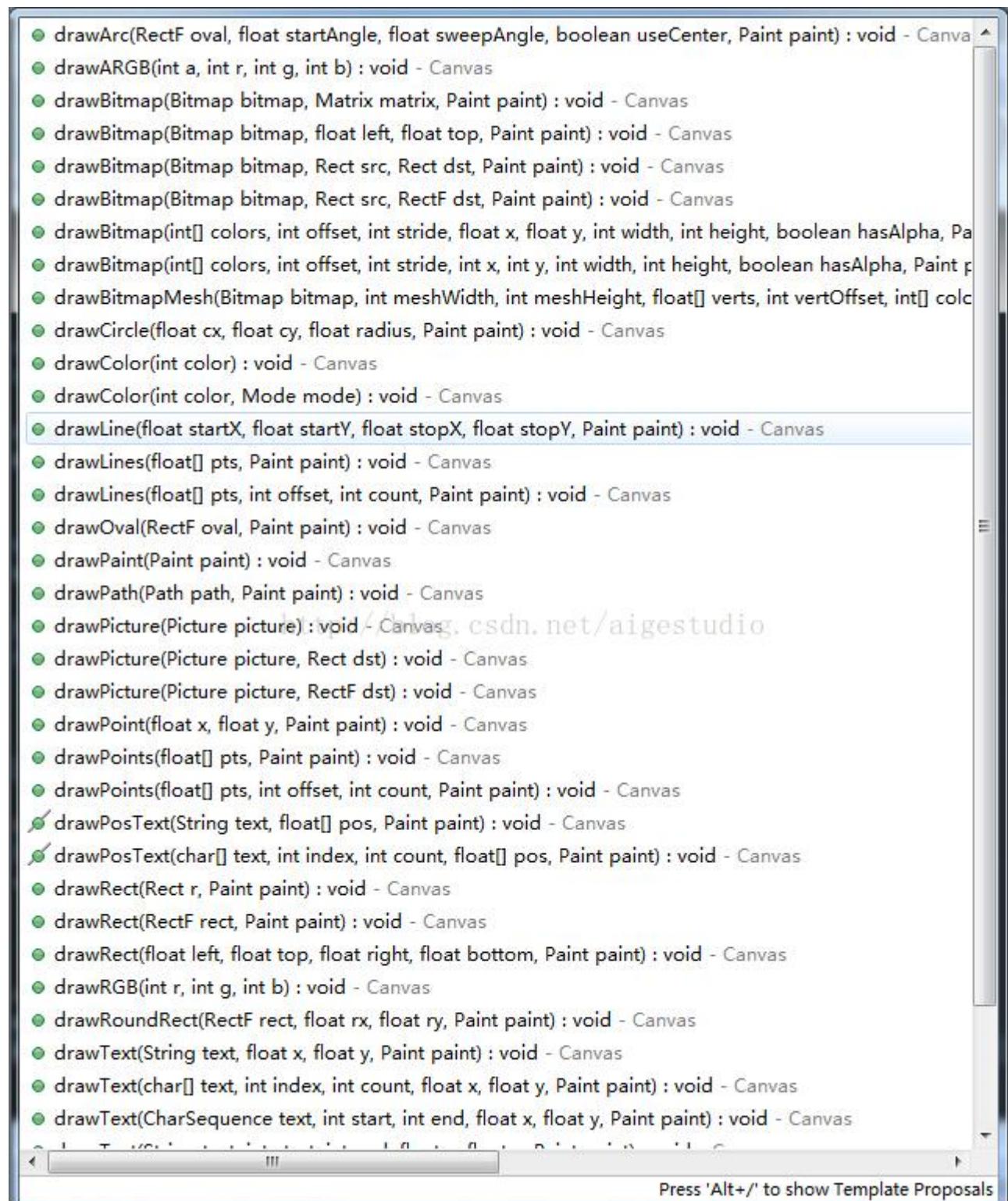
```
27. }
```

现实世界中，我们画画的画笔是多种多样的，有马克笔、铅笔、圆珠笔、毛笔、水彩笔、荧光笔等等等等……而这些笔的属性也各自不同，像铅笔按照炭颗粒的粗糙度可以分为 2B、3B、4B、5B、HB 当然还有 SB，而水彩笔也有各种不同的颜色，马克笔就更霸气了不说了！同样地在 Android 的画笔里，现实有的它也有，没有的它还有！我们可以用 Paint 的各种 **setter** 方法来设置各种不同的属性，比如 **setColor()** 设置画笔颜色，**setStrokeWidth()** 设置描边线条，**setStyle()** 设置画笔的样式：

```
● set(Paint src) : void - Paint
● setAlpha(int a) : void - Paint
● setAntiAlias(boolean aa) : void - Paint
● setARGB(int a, int r, int g, int b) : void - Paint
● setColor(int color) : void - Paint
● setColorFilter(ColorFilter filter) : ColorFilter - Paint
● setDither(boolean dither) : void - Paint
● setFakeBoldText(boolean fakeBoldText) : void - Paint
● setFilterBitmap(boolean filter) : void - Paint
● setFlags(int flags) : void - Paint
● setHinting(int mode) : void - Paint
● setLinearText(boolean linearText) : void - Paint
● setMaskFilter(MaskFilter maskfilter) : MaskFilter - Paint
● setPathEffect(PathEffect effect) : PathEffect - Paint
● setRasterizer(Rasterizer rasterizer) : Rasterizer - Paint
● setShader(Shader shader) : Shader - Paint
● setShadowLayer(float radius, float dx, float dy, int color) : void - Paint
● setStrikeThruText(boolean strikeThruText) : void - Paint
● setStrokeCap(Cap cap) : void - Paint
● setStrokeJoin(Join join) : void - Paint
● setStrokeMiter(float miter) : void - Paint
● setStrokeWidth(float width) : void - Paint
● setStyle(Style style) : void - Paint
● setSubpixelText(boolean subpixelText) : void - Paint
● setTextAlign(Align align) : void - Paint
● setTextLocale(Locale locale) : void - Paint
● setTextScaleX(float scaleX) : void - Paint
● setTextSize(float textSize) : void - Paint
● setTextSkewX(float skewX) : void - Paint
● setTypeface(Typeface typeface) : Typeface - Paint
● setUnderlineText(boolean underlineText) : void - Paint
● setXfermode(Xfermode xfermode) : Xfermode - Paint
```

Press 'Alt+/' to show Template Proposals

Paint 集成了所有“画”的属性，而 Canvas 则定义了所有要画的东西，我们可以通过 Canvas 下的各类 drawXXX 方法绘制各种不同的东西，比如绘制一个圆 drawCircle()，绘制一个圆弧 drawArc()，绘制一张位图 drawBitmap() 等等等：



既然初步了解了 `Paint` 和 `Canvas`, 我们不妨就尝试在我们的画布上绘制一点东西, 比如一个圆环? 我们先来设置好画笔的属性:

[java] view plain copy print?

```
1. /**
2. * 初始化画笔
```

```
3.  */
4. private void initPaint() {
5.     // 实例化画笔并打开抗锯齿
6.     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
7.
8.     /*
9.      * 设置画笔样式为描边，圆环嘛.....当然不能填充不然就么意思了
10.     *
11.     * 画笔样式分三种：
12.     * 1.Paint.Style.STROKE: 描边
13.     * 2.Paint.Style.FILL_AND_STROKE: 描边并填充
14.     * 3.Paint.Style.FILL: 填充
15.     */
16.     mPaint.setStyle(Paint.Style.STROKE);
17.
18.     // 设置画笔颜色为浅灰色
19.     mPaint.setColor(Color.LTGRAY);
20.
21.     /*
22.      * 设置描边的粗细，单位：像素 px
23.      * 注意：当 setStrokeWidth(0) 的时候描边宽度并不为 0 而是只占一个像素
24.      */
25.     mPaint.setStrokeWidth(10);
26. }
```

然后在我们的 `onDraw` 方法中绘制 Circle 即可：

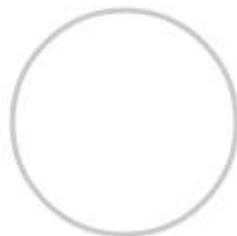
[java] view plain copy print?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     super.onDraw(canvas);
4.
5.     // 绘制圆环
6.     canvas.drawCircle(MeasureUtil.getScreenSize((Activity) mContext)[0] / 2,
7.                       MeasureUtil.getScreenSize((Activity) mContext)[1] / 2, 200, mPaint);
7. }
```

这里要注意哦！`drawCircle` 表示绘制的是圆形，但是在我们的画笔样式设置为描边后其绘制出来的是一个圆环！其中 `drawCircle` 的前两个参数表示圆心的 XY 坐标，这里我们用到了一个工具类获取屏幕尺寸以便将其圆心设置在屏幕中心位置，第三个参数是圆的半径，第四个参数则为我们的画笔！

这里有一点要注意：在 Android 中设置数字类型的参数时如果没有特别的说明，参数的单位一般都为 px 像素。

好了，我们来运行下我们的 Demo 看看结果：



一个灰常漂亮的圆环展现在我们眼前！怎么样是不是很爽，这算是我们写的第一个 View，当然这只是第一步，虽然只是一小步，但必定会是影响人类进步的一大步！……Fuck！不过一个简单地画一个圆恐怕难以满足各位的胃口对吧，那我们尝试让它动起来？比如让它的半径从小到大地不断变化，那怎么实现好呢？大家如果了解动画的原理就会知道，一个动画是由无数张连贯的图片构成的，这些图片之间快速地切换再加上我们眼睛的视觉暂留给我们造成了在“动”的假象。那么原理有了实现就很简单了，我们不断地改变圆环的半径并且重新去画并展示不就成了？同样地，在 Android 中提供了一个叫 `invalidate()` 的方法来让我们重绘我们的 View。现在我们重新构造一下我们的代码，添加一个 `int` 型的成员变量作为半径值的引用，再提供一个 `setter` 方法对外设置半径值，并在设置了该值后调用 `invalidate()` 方法重绘 View：

[java] view plaincopyprint?

```
1. public class CustomView extends View {  
2.     private Paint mPaint;// 画笔  
3.     private Context mContext;// 上下文环境引用  
4.  
5.     private int radiu;// 圆环半径
```

```
6.
7.     public CustomView(Context context) {
8.         this(context, null);
9.     }
10.
11.    public CustomView(Context context, AttributeSet attrs) {
12.        super(context, attrs);
13.        mContext = context;
14.
15.        // 初始化画笔
16.        initPaint();
17.    }
18.
19.    /**
20.     * 初始化画笔
21.     */
22.    private void initPaint() {
23.        // 实例化画笔并打开抗锯齿
24.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
25.
26.        /*
27.         * 设置画笔样式为描边，圆环嘛.....当然不能填充不然就么意思了
28.         *
29.         * 画笔样式分三种：
30.         * 1.Paint.Style.STROKE: 描边
31.         * 2.Paint.Style.FILL_AND_STROKE: 描边并填充
32.         * 3.Paint.Style.FILL: 填充
33.         */
34.        mPaint.setStyle(Paint.Style.STROKE);
35.
36.        // 设置画笔颜色为浅灰色
37.        mPaint.setColor(Color.LTGRAY);
38.
39.        /*
40.         * 设置描边的粗细，单位：像素 px
41.         * 注意：当 setStrokeWidth(0) 的时候描边宽度并不为 0 而是只占一个像素
42.         */
43.        mPaint.setStrokeWidth(10);
44.    }
45.
46.    @Override
47.    protected void onDraw(Canvas canvas) {
48.        super.onDraw(canvas);
49.
```

```
50.         // 绘制圆环
51.         canvas.drawCircle(MeasureUtil.getScreenSize((Activity) mContext)[0]
52.             / 2, MeasureUtil.getScreenSize((Activity) mContext)[1] / 2, radiu, mPaint);
53.
54.     }
55.     public synchronized void setRadius(int radiu) {
56.         this.radiu = radiu;
57.
58.         // 重绘
59.         invalidate();
60.     }
61. }
```

那么 OK，我们在 Activity 中开一个线程，通过 Handler 来定时间断地设置半径的值并刷新界面：

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
2.     private CustomView mCustomView;// 我们的自定义 View
3.
4.     private int radiu;// 半径值
5.
6.     @SuppressLint("HandlerLeak")
7.     private Handler mHandler = new Handler() {
8.         @Override
9.         public void handleMessage(Message msg) {
10.             // 设置自定义 View 的半径值
11.             mCustomView.setRadius(radiu);
12.         }
13.     };
14.
15.     @Override
16.     protected void onCreate(Bundle savedInstanceState) {
17.         super.onCreate(savedInstanceState);
18.         setContentView(R.layout.activity_main);
19.
20.         // 获取控件
21.         mCustomView = (CustomView) findViewById(R.id.main_cv);
22.
23.         /*
24.          * 开线程
25.          */
26.         new Thread(new Runnable() {
27.             @Override
```

```

28.         public void run() {
29.             /*
30.                 * 确保线程不断执行不断刷新界面
31.                 */
32.             while (true) {
33.                 try {
34.                     /*
35.                         * 如果半径小于 200 则自加否则大于 200 后重置半径值以实现往
复
36.                     */
37.                     if (radius <= 200) {
38.                         radius += 10;
39.
40.                         // 发消息给 Handler 处理
41.                         mHandler.obtainMessage().sendToTarget();
42.                     } else {
43.                         radius = 0;
44.                     }
45.
46.                     // 每执行一次暂停 40 毫秒
47.                     Thread.sleep(40);
48.                 } catch (InterruptedException e) {
49.                     e.printStackTrace();
50.                 }
51.             }
52.         }
53.     }).start();
54. }
55.
56. @Override
57. protected void onDestroy() {
58.     super.onDestroy();
59.     // 界面销毁后清除 Handler 的引用
60.     mHandler.removeCallbacksAndMessages(null);
61. }
62. }

```

运行后的效果我就不演示了，项目源码会共享。

但是有一个问题，这么一个类似进度条的效果我还要在 `Activity` 中处理一些逻辑多不科学！浪费代码啊！还要 `Handler` 来传递信息，Fuck！就不能在自定义 `View` 中一次性搞定吗？答案是肯定的，我们修改下 `CustomView` 的代码让其实现 `Runnable` 接口，这样就爽多了：

`[java] view plain copy print?`

```

1. public class CustomView extends View implements Runnable {

```

```
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.
5.     private int radiu;// 圆环半径
6.
7.     public CustomView(Context context) {
8.         this(context, null);
9.     }
10.
11.    public CustomView(Context context, AttributeSet attrs) {
12.        super(context, attrs);
13.        mContext = context;
14.
15.        // 初始化画笔
16.        initPaint();
17.    }
18.
19. /**
20. * 初始化画笔
21. */
22. private void initPaint() {
23.     // 实例化画笔并打开抗锯齿
24.     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
25.
26.     /*
27.      * 设置画笔样式为描边，圆环嘛.....当然不能填充不然就么意思了
28.      *
29.      * 画笔样式分三种：
30.      * 1.Paint.Style.STROKE: 描边
31.      * 2.Paint.Style.FILL_AND_STROKE: 描边并填充
32.      * 3.Paint.Style.FILL: 填充
33.      */
34.     mPaint.setStyle(Paint.Style.STROKE);
35.
36.     // 设置画笔颜色为浅灰色
37.     mPaint.setColor(Color.LTGRAY);
38.
39.     /*
40.      * 设置描边的粗细，单位：像素 px
41.      * 注意：当 setStrokeWidth(0) 的时候描边宽度并不为 0 而是只占一个像素
42.      */
43.     mPaint.setStrokeWidth(10);
44. }
45.
```

```
46.     @Override
47.     protected void onDraw(Canvas canvas) {
48.         super.onDraw(canvas);
49.
50.         // 绘制圆环
51.         canvas.drawCircle(MeasureUtil.getScreenSize((Activity) mContext)[0]
52.             / 2, MeasureUtil.getScreenSize((Activity) mContext)[1] / 2, radiu, mPaint);
53.
54.     }
55.     @Override
56.     public void run() {
57.         /*
58.          * 确保线程不断执行不断刷新界面
59.          */
60.         while (true) {
61.             try {
62.                 /*
63.                  * 如果半径小于 200 则自加否则大于 200 后重置半径值以实现往复
64.                  */
65.                 if (radiu <= 200) {
66.                     radiu += 10;
67.
68.                     // 刷新 View
69.                     invalidate();
70.                 } else {
71.                     radiu = 0;
72.                 }
73.
74.                 // 每执行一次暂停 40 毫秒
75.                 Thread.sleep(40);
76.             } catch (InterruptedException e) {
77.                 e.printStackTrace();
78.             }
79.         }
80.     }

```

而我们的 Activity 呢也能摆脱繁琐的代码逻辑：

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
2.     private CustomView mCustomView;// 我们的自定义 View
3.
4.     @Override
```

```
5.     protected void onCreate(Bundle savedInstanceState) {
6.         super.onCreate(savedInstanceState);
7.         setContentView(R.layout.activity_main);
8.
9.         // 获取控件
10.        mCustomView = (CustomView) findViewById(R.id.main_cv);
11.
12.        /*
13.         * 开线程
14.         */
15.        new Thread(mCustomView).start();
16.    }
17. }
```

运行一下看看呗！禽！！！报错了：

```
FATAL EXCEPTION: Thread-12867
Process: com.aigestudio.customviewdemo, PID: 2502
android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:6391)
at android.view.ViewRootImpl.invalidateChildInParent(ViewRootImpl.java:866)
at android.view.ViewGroup.invalidateChild(ViewGroup.java:4347)
at android.view.View.invalidate(View.java:10990)
at android.view.View.invalidate(View.java:10945)
at com.aigestudio.customviewdemo.views.CustomView.run(CustomView.java:84)
at java.lang.Thread.run(Thread.java:841)
```

Why!因为我们在非 UI 线程中更新了 UI!而在 Android 中非 UI 线程是不能直接更新 UI 的，怎么办？用 Handler? NO! Android 给我们提供了一个更便捷的方法：postInvalidate(); 用它替代我们原来的 invalidate()即可：

[java] view plaincopyprint?

```
1. @Override
2. public void run() {
3.     /*
4.      * 确保线程不断执行不断刷新界面
5.      */
6.     while (true) {
7.         try {
8.             /*
9.              * 如果半径小于 200 则自加否则大于 200 后重置半径值以实现往复
10.             */
11.            if (radius <= 200) {
```

```
12.             radius += 10;
13.
14.             // 刷新 View
15.             postInvalidate();
16.         } else {
17.             radius = 0;
18.         }
19.
20.         // 每执行一次暂停 40 毫秒
21.         Thread.sleep(40);
22.     } catch (InterruptedException e) {
23.         e.printStackTrace();
24.     }
25. }
26. }
```

运行效果不变。

下集精彩预告：Paint 为我们提供了大量的 **setter** 方法去设置画笔的属性，而 Canvas 呢也提供了大量的 **drawXXX** 方法去告诉我们能画些什么，那么小伙伴们知道这些方法是怎么用的又能带给我们怎样炫酷的效果呢？

2. 自定义控件其实很简单(2)

上一节我们粗略地讲了下如何去实现我们的 View 并概述了 View 形成动画的基本原理，这一节我们紧跟上一节的步伐来深挖如何去绘制更复杂的 View！

通过上一节的学习我们了解到什么是画布 Canvas 什么是画笔 Paint，并且学习了如何去设置画笔的属性如何在画布上画一个圆，然而，画笔的属性并非仅仅就设置个颜色、大小那么简单而画布呢肯定也不单单只是能画一个圆那么无趣，工欲善其事必先利其器，既然想画好图那必须学会画笔和画布的使用，那么今天我们就来看看 Android 的画笔跟我们现实中的画笔有什么不同呢？

如上节所说我们可以通过 Paint 中大量的 **setter** 方法来为画笔设置属性：

```
set(Paint src) : void - Paint
setAlpha(int a) : void - Paint
setAntiAlias(boolean aa) : void - Paint
setARGB(int a, int r, int g, int b) : void - Paint
setColor(int color) : void - Paint
setColorFilter(ColorFilter filter) : ColorFilter - Paint
setDither(boolean dither) : void - Paint
setFakeBoldText(boolean fakeBoldText) : void - Paint
setFilterBitmap(boolean filter) : void - Paint
setFlags(int flags) : void - Paint
setHinting(int mode) : void - Paint
setLinearText(boolean linearText) : void - Paint
setMaskFilter(MaskFilter maskfilter) : MaskFilter - Paint
setPathEffect(PathEffect effect) : PathEffect - Paint
setRasterizer(Rasterizer rasterizer) : Rasterizer - Paint
setShader(Shader shader) : Shader - Paint
setShadowLayer(float radius, float dx, float dy, int color) : void - Paint
setStrikeThruText(boolean strikeThruText) : void - Paint
setStrokeCap(Cap cap) : void - Paint
setStrokeJoin(Join join) : void - Paint
setStrokeMiter(float miter) : void - Paint
setStrokeWidth(float width) : void - Paint
setStyle(Style style) : void - Paint
setSubpixelText(boolean subpixelText) : void - Paint
setTextAlign(Align align) : void - Paint
setTextLocale(Locale locale) : void - Paint
setTextScaleX(float scaleX) : void - Paint
setTextSize(float textSize) : void - Paint
setTextSkewX(float skewX) : void - Paint
setTypeface(Typeface typeface) : Typeface - Paint
setUnderlineText(boolean underlineText) : void - Paint
setXfermode(Xfermode xfermode) : Xfermode - Paint
```

这些属性大多我们都可以见名知意，很好理解，即便如此，哥还是带大家过一遍逐个剖析其用法，其中会不定穿插各种绘图类比如 Canvas、Xfermode、ColorFilter 等等的用法。

set(Paint src)

顾名思义为当前画笔设置一个画笔，说白了就是把另一个画笔的属性设置 Copy 给我们的画笔，不累赘了

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

不扯了，别跟我说不懂

setAlpha(int a)

同上

setAntiAlias(boolean aa)

这个上一节我们用到了，打开抗锯齿，不过我要说明一点，抗锯齿是依赖于算法的，算法决定抗锯齿的效率，在我们绘制棱角分明的图像时，比如一个矩形、一张位图，我们不需要打开抗锯齿。

setColor(int color)

不扯

setColorFilter(ColorFilter filter)

设置颜色过滤，什么意思呢？就像拿个筛子把颜色“滤”一遍获取我们想要的色彩结果，感觉像是扯蛋白说一样是不是？没事我们慢慢说你一定会懂，这个方法需要我们传入一个 ColorFilter 参数同样也会返回一个 ColorFilter 实例，那么 ColorFilter 类是什么呢？追踪源码进去你会发现其里面很简单几乎没有：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)

```
1. public class ColorFilter {
2.     int native_instance;
3.
4.     /**
5.      * @hide
6.     */
7.     public int nativeColorFilter;
8.
9.     protected void finalize() throws Throwable {
10.         try {
11.             super.finalize();
12.         } finally {
13.             finalizer(native_instance, nativeColorFilter);
14.         }
15.     }
16.
17.     private static native void finalizer(int native_instance, int nativeColo
rFilter);
18. }
```

压根没有和图像处理相关的方法对吧，那么说明该类必定是个父类或者说其必有一定的子类去实现一些方法，查看 API 文档发现果然有三个子类：

```
public class
ColorFilter
extends Object

java.lang.Object
↳ android.graphics.ColorFilter

▶ Known Direct Subclasses
ColorMatrixColorFilter, LightingColorFilter, PorterDuffColorFilter
```

ColorMatrixColorFilter、LightingColorFilter 和 PorterDuffColorFilter，也就是说我们在 setColorFilter(ColorFilter filter) 的时候可以直接传入这三个子类对象作为参数，那么这三个子类又是什么东西呢？首先我们来看看

ColorMatrixColorFilter

中文直译为色彩矩阵颜色过滤器，要明白这玩意你得先了解什么是色彩矩阵。在 Android 中图片是以 RGBA 像素点的形式加载到内存中的，修改这些像素信息需要一个叫做 ColorMatrix 类的支持，其定义了一个 4×5 的 float[]类型的矩阵：

[java] view plain copy print?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{
2.         1, 0, 0, 0, 0,
3.         0, 1, 0, 0, 0,
4.         0, 0, 1, 0, 0,
5.         0, 0, 0, 1, 0,
6.     });
```

其中，第一行表示的 R（红色）的向量，第二行表示的 G（绿色）的向量，第三行表示的 B（蓝色）的向量，最后一行表示 A（透明度）的向量，这一顺序必须要正确不能混淆！这个矩阵不同的位置表示的 RGBA 值，其范围在 0.0F 至 2.0F 之间，1 为保持原图的 RGB 值。每一行的第五列数字表示偏移值，何为偏移值？顾名思义当我们想让颜色更倾向于红色的时候就增大 R 向量中的偏移值，想让颜色更倾向于蓝色的时候就增大 B 向量中的偏移值，这是最朴素的理解，但是事实上色彩偏移的概念是基于白平衡来理解的，什么是白平衡呢？说得简单点就是白色是什么颜色！如果大家是个单反爱好者或者会些 PS 就会很容易理解这个概念，在单反的设置参数中有个色彩偏移，其定义的就是白平衡的色彩偏移值，就是当你去拍一张照片的时候白色是什么颜色的，在正常情况下白色是 (255, 255, 255, 255) 但是现实世界中我们是无法找到这样的纯白物体的，所以在我们用单反拍照之前就会拿一个我们认为是白色的物体让相机记录这个物体的颜色作为白色，然后拍摄时整张照片的颜色都会依据这个定义的白色来偏移！而这个我们定义的“白色”(比如: 255, 253, 251, 247) 和纯白(255, 255, 255, 255) 之间的偏移值 (0, 2, 4, 8) 我们称之为白平衡的色彩偏移。如果你不理解不

要紧，自定义控件系列完结后紧接着就是设计色彩基础~~~~在这你就像我开头说的那样朴素地理解下就好。

那么说了这么多，这玩意到底有啥用呢？我们来做个 test！还是接着昨天那个圆环，不过我们今天把它改成绘制一个圆并且去掉线程动画的效果因为我们不需要：

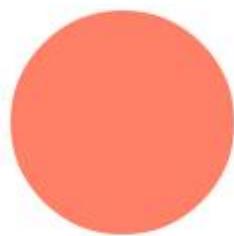
[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.
5.     public CustomView(Context context) {
6.         this(context, null);
7.     }
8.
9.     public CustomView(Context context, AttributeSet attrs) {
10.        super(context, attrs);
11.        mContext = context;
12.
13.        // 初始化画笔
14.        initPaint();
15.    }
16.
17. /**
18. * 初始化画笔
19. */
20. private void initPaint() {
21.     // 实例化画笔并打开抗锯齿
22.     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
23.
24.     /*
25.      * 设置画笔样式为描边，圆环嘛.....当然不能填充不然就么意思了
26.      *
27.      * 画笔样式分三种：
28.      * 1.Paint.Style.STROKE: 描边
29.      * 2.Paint.Style.FILL_AND_STROKE: 描边并填充
30.      * 3.Paint.Style.FILL: 填充
31.      */
32.     mPaint.setStyle(Paint.Style.FILL);
33.
34.     // 设置画笔颜色为自定义颜色
35.     mPaint.setColor(Color.argb(255, 255, 128, 103));
36.
37.     /*

```

```
38.         * 设置描边的粗细, 单位: 像素 px 注意: 当 setStrokeWidth(0) 的时候描边宽度并  
        不为 0 而是只占一个像素  
39.         */  
40.         mPaint.setStrokeWidth(10);  
41.     }  
42.  
43.     @Override  
44.     protected void onDraw(Canvas canvas) {  
45.         super.onDraw(canvas);  
46.  
47.         // 绘制圆形  
48.         canvas.drawCircle(MeasureUtil.getScreenSize((Activity) mContext)[0]  
        / 2, MeasureUtil.getScreenSize((Activity) mContext)[1] / 2, 200, mPaint);  
49.     }  
50. }
```

运行下是一个橙红色的圆~~是不是有点萝卜头国旗帜的感脚?



下面我们为 Paint 设置一个色彩矩阵:

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 生成色彩矩阵
2. ColorMatrix colorMatrix = new ColorMatrix(new float[]{ 
3.     1, 0, 0, 0, 0,
4.     0, 1, 0, 0, 0,
5.     0, 0, 1, 0, 0,
6.     0, 0, 0, 1, 0,
7. });
8. mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
```

再次运行发现没变化啊！！！草！！！是不是感觉被我坑了？如果你真的那么认为我只能说你压根就没认真看上面的文字，我说过什么？值为 1 时表示什么？表示不改变原色彩的值！！这时我们改变色彩矩阵：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 生成色彩矩阵
2. ColorMatrix colorMatrix = new ColorMatrix(new float[]{ 
3.     0.5F, 0, 0, 0, 0,
4.     0, 0.5F, 0, 0, 0,
5.     0, 0, 0.5F, 0, 0,
6.     0, 0, 0, 1, 0,
7. });
8. mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
```

再次运行：



是不是明显不一样了？颜色变深了便淳厚了！我们通过色彩矩阵与原色彩的计算得出的色彩就是这样的。那它们是如何计算的呢？其实说白了就是矩阵之间的运算乘积：

$$\text{ColorMatrix} = \begin{pmatrix} a & b & c & d & e \\ f & g & h & i & j \\ k & l & m & n & o \\ p & q & r & s & t \end{pmatrix} \quad \text{MyColor} = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

<http://blog.csdn.net/pigestudio>

$$\text{Result} = \text{ColorMatrix} * \text{MyColor} = \begin{pmatrix} R = a * R + b * G + c * B + d * A + e \\ G = f * R + g * G + h * B + i * A + j \\ B = k * R + l * G + m * B + n * A + o \\ A = p * R + q * G + r * B + s * A + t \end{pmatrix}$$

矩阵 **ColorMatrix** 的一行乘以矩阵 **MyColor** 的一列作为矩阵 **Result** 的一行，这里 **MyColor** 的 **RGBA** 值我们需要转换为 **[0, 1]**。那么我们依据此公式来计算下我们得到的 **RGBA** 值是否跟我们计算得出来的圆的 **RGBA** 值一样：

$$\text{ColorMatrix} = \begin{pmatrix} 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{MyColor} = \begin{pmatrix} 1 \\ 0.5 \\ 0.4 \\ 1 \end{pmatrix}$$

$$\text{Result} = \text{ColorMatrix} * \text{MyColor} = \begin{pmatrix} R = 0.5 * 1 + 0 * 0.5 + 0 * 0.4 + 0 * 1 + 0 = 0.5 \\ G = 0 * 1 + 0.5 * 0.5 + 0 * 0.4 + 0 * 1 + 0 = 0.25 \\ B = 0 * 1 + 0 * 0.5 + 0.5 * 0.4 + 0 * 1 + 0 = 0.2 \\ A = 0 * 1 + 0 * 0.5 + 0 * 0.4 + 1 * 1 + 0 = 1 \end{pmatrix}$$

我们计算得出最后的 RGBA 值应该为： 0.5, 0.25, 0.2, 1；有兴趣的童鞋可以去 PS 之类的绘图软件里试试看正不正确对不对~~~这里就不演示了！看完这里有朋友又会说了，这玩意有毛线用啊！改个颜色还这么复杂！劳资直接 setColor 多爽！！没错，你这样想是对的，因为毕竟我们只是一个颜色，可是如果是一张图片呢？？？？一张图片可有还几十万色彩呢！！！你麻痹你跟我说 setColor？那么我们换张图片来试试呗！看看是什么样的效果：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```

1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.     private Bitmap bitmap;// 位图
5.
6.     private int x,y;// 位图绘制时左上角的起点坐标
7.
8.     public CustomView(Context context) {
9.         this(context, null);
10.    }
11.
12.    public CustomView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.        mContext = context;
15.
16.        // 初始化画笔
17.        initPaint();
18.
19.        // 初始化资源
20.        initRes(context);
21.    }
22.
23.    /**
24.     * 初始化画笔
25.     */
26.    private void initPaint() {
27.        // 实例化画笔

```

```
28.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
29.     }
30.
31.     /**
32.      * 初始化资源
33.     */
34.     private void initRes(Context context) {
35.         // 获取位图
36.         bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.a);
37.
38.         /*
39.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
40.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
41.          * 屏幕坐标 y 轴向上偏移位图一半的高度
42.         */
43.         x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - bitmap.getWidth() / 2;
44.         y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - bitmap.getHeight() / 2;
45.     }
46.
47.     @Override
48.     protected void onDraw(Canvas canvas) {
49.         super.onDraw(canvas);
50.
51.         // 绘制位图
52.         canvas.drawBitmap(bitmap, x, y, mPaint);
53.     }
54. }
```

如代码所示我们清除了所有的画笔属性设置因为没必要，从资源获取一个 Bitmap 绘制在画布上：

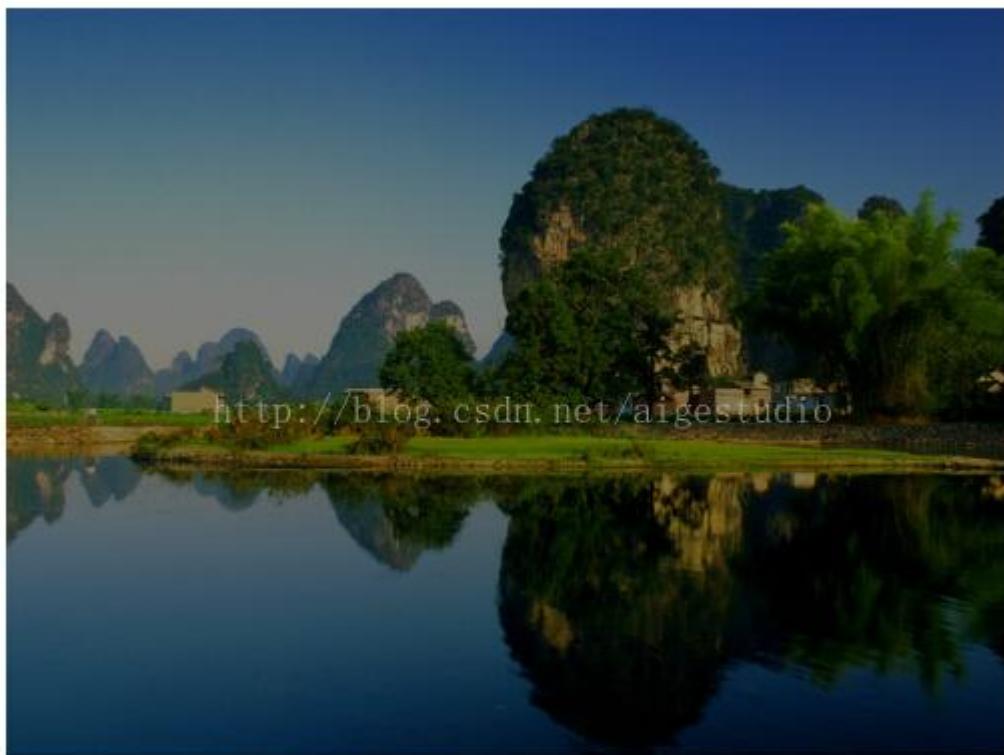


一张灰常漂亮的风景图，好！现在我们来为我们的画笔添加一个颜色过滤：

[java] view plaincopyprint?

```
1. // 生成色彩矩阵
2. ColorMatrix colorMatrix = new ColorMatrix(new float[]{ 
3.     0.5F, 0, 0, 0, 0,
4.     0, 0.5F, 0, 0, 0,
5.     0, 0, 0.5F, 0, 0,
6.     0, 0, 0, 1, 0,
7. });
8. mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
```

大家看到还是刚才那个色彩矩阵，运行下看看什么效果呢：



变暗了对吧！没意思，我们来点更刺激的，改下 ColorMatrix 矩阵：

[java] view plaincopyprint?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{
2.         0.33F, 0.59F, 0.11F, 0, 0,
3.         0.33F, 0.59F, 0.11F, 0, 0,
4.         0.33F, 0.59F, 0.11F, 0, 0,
5.         0, 0, 0, 1, 0,
6. });
```



噢！变灰了！还是没意思！继续改：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{
2.         -1, 0, 0, 1, 1,
3.         0, -1, 0, 1, 1,
4.         0, 0, -1, 1, 1,
5.         0, 0, 0, 1, 0,
6.     });
```



哟呵！！是不是有点类似 PS 里反相的效果？我们常看到的图片都是 RGB 的，颠覆一下思维，看看 BGR 的试试：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{
2.         0, 0, 1, 0, 0,
3.         0, 1, 0, 0, 0,
4.         1, 0, 0, 0, 0,
5.         0, 0, 0, 1, 0,
6.     });
```



这样红色的变成了蓝色而蓝色的就变成了红色，继续改：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{
2.         0.393F, 0.769F, 0.189F, 0, 0,
3.         0.349F, 0.686F, 0.168F, 0, 0,
4.         0.272F, 0.534F, 0.131F, 0, 0,
5.         0, 0, 0, 1, 0,
6.     });
```



是不是有点类似于老旧照片的感脚？继续：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{  
2.         1.5F, 1.5F, 1.5F, 0, -1,  
3.         1.5F, 1.5F, 1.5F, 0, -1,  
4.         1.5F, 1.5F, 1.5F, 0, -1,  
5.         0, 0, 0, 1, 0,  
6.     });
```



类似去色后高对比度的效果，继续：

[java] view plaincopyprint?

```
1. ColorMatrix colorMatrix = new ColorMatrix(new float[]{  
2.         1.438F, -0.122F, -0.016F, 0, -0.03F,  
3.         -0.062F, 1.378F, -0.016F, 0, 0.05F,  
4.         -0.062F, -0.122F, 1.483F, 0, -0.02F,  
5.         0, 0, 0, 1, 0,  
6.     });
```



饱和度对比度加强，好了不演示了……累死我了！截图粘贴上传！！

这些各种各样的图像效果在哪见过？PS?对的！还有各种拍照软件拍摄后的特效处理！大致原理都是这么来的！有人会问爱哥你傻逼么！这么多参数怎么玩！谁记得！而且TMD用参数调颜色？我映像中都是直接在各种绘图软件（比如PS）里拖进度条的！这怎么玩！淡定！如我所说很多时候你压根不需要了解太多原理，只需站在巨人的丁丁上即可，所以稍安勿躁！再下一个系列教程“设计色彩”中爱哥教你玩转色彩并且让设计和开发无缝结合！

ColorMatrixColorFilter 和 ColorMatrix 就是这么个东西，ColorMatrix 类里面也提供了一些实在的方法，比如 `setSaturation(float sat)` 设置饱和度，而且 ColorMatrix 每个方法都用了阵列的计算，如果大家感兴趣可以自己去深挖来看不过我是真心不推荐的~~~

下面我们来看看 ColorFilter 的另一个子类

LightingColorFilter

顾名思义光照颜色过滤，这肯定是跟光照是有关的了~~该类有且只有一个构造方法：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. LightingColorFilter (int mul, int add)
```

这个方法非常非常地简单！`mul` 全称是 `colorMultiply` 意为色彩倍增，而 `add` 全称是 `colorAdd` 意为色彩添加，这两个值都是 16 进制的色彩值 `0xAARRGGBB`。这个方法使用也是非常的简单。还是拿上面那张图片来说吧，比如我们想要去掉绿色：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.     private Bitmap bitmap;// 位图
5.
6.     private int x, y;// 位图绘制时左上角的起点坐标
7.
8.     public CustomView(Context context) {
9.         this(context, null);
10.    }
11.
12.    public CustomView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.        mContext = context;
15.
16.        // 初始化画笔
17.        initPaint();
18.
19.        // 初始化资源
20.        initRes(context);
21.    }
22.
23.    /**
24.     * 初始化画笔
25.     */
26.    private void initPaint() {
27.        // 实例化画笔
28.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
29.
30.        // 设置颜色过滤
31.        mPaint.setColorFilter(new LightingColorFilter(0xFFFF00FF, 0x00000000
32.));
32.    }
33.
34.    /**
35.     * 初始化资源
36.     */
37.    private void initRes(Context context) {
38.        // 获取位图
```

```

39.         bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.a);
40.
41.         /*
42.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
43.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
44.          * 屏幕坐标 y 轴向上偏移位图一半的高度
45.         */
46.         x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - bitmap.getWidth() / 2;
47.         y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - bitmap.getHeight() / 2;
48.     }
49.
50.    @Override
51.    protected void onDraw(Canvas canvas) {
52.        super.onDraw(canvas);
53.
54.        // 绘制位图
55.        canvas.drawBitmap(bitmap, x, y, mPaint);
56.    }
57. }

```

运行后你会发现绿色确实是没了但是原来偏绿的部分现在居然成了红色，为毛！敬请关注下一系列设计色彩文章！！哈哈哈！！当 LightingColorFilter(0xFFFFFFFF, 0x00000000)的时候原图是不会有任何改变的，如果我们想增加红色的值，那么

LightingColorFilter(0xFFFFFFFF, 0x00XX0000)就好，其中 XX 取值为 00 至 FF。那么这个方法有什么存在的意义呢？存在必定合理，这个方法存在一定是有它可用之处的，前些天有个盆友在群里问点击一个图片如何直接改变它的颜色而不是为他多准备另一张点击效果的图片，这种情况下该方法就派上用场了！如下图一个灰色的星星，我们点击后让它变成黄色



代码如下，注释很清楚我就不再多说了：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```

1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.     private Bitmap bitmap;// 位图
5.
6.     private int x, y;// 位图绘制时左上角的起点坐标

```

```
7.     private boolean isClick;// 用来标识控件是否被点击过
8.
9.     public CustomView(Context context) {
10.         this(context, null);
11.     }
12.
13.    public CustomView(Context context, AttributeSet attrs) {
14.        super(context, attrs);
15.        mContext = context;
16.
17.        // 初始化画笔
18.        initPaint();
19.
20.        // 初始化资源
21.        initRes(context);
22.
23.        setOnClickListener(new OnClickListener() {
24.            @Override
25.            public void onClick(View v) {
26.                /*
27.                 * 判断控件是否被点击过
28.                 */
29.                if (isClick) {
30.                    // 如果已经被点击了则点击时设置颜色过滤为还原本色
31.                    mPaint.setColorFilter(null);
32.                    isClick = false;
33.                } else {
34.                    // 如果未被点击则点击时设置颜色过滤后为黄色
35.                    mPaint.setColorFilter(new LightingColorFilter(0xFFFFFFFF
36. , 0X00FFFFFF));
37.                    isClick = true;
38.                }
39.                // 记得重绘
40.                invalidate();
41.            }
42.        });
43.    }
44.
45.    /**
46.     * 初始化画笔
47.     */
48.    private void initPaint() {
49.        // 实例化画笔
```

```
50.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
51.     }
52.
53.     /**
54.      * 初始化资源
55.     */
56.     private void initRes(Context context) {
57.         // 获取位图
58.         bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.a2);
59.
60.         /*
61.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
62.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
63.          * 屏幕坐标 y 轴向上偏移位图一半的高度
64.          */
65.         x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - bitmap.get
66.             etWidth() / 2;
66.         y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - bitmap.g
67.             etHeight() / 2;
68.     }
69.
70.     @Override
71.     protected void onDraw(Canvas canvas) {
72.         super.onDraw(canvas);
73.
74.         // 绘制位图
75.         canvas.drawBitmap(bitmap, x, y, mPaint);
76.     }
76. }
```

运行后点击星星即可变成黄色再点击变回灰色，当我们不想要颜色过滤的效果时，`setColorFilter(null)`并重绘视图即可！那么为什么要叫光照颜色过滤呢？原因很简单，因为它所呈现的效果就像有色光照在物体上染色一样~~~哎，不说这方法了，看下一个也是最后一个 `ColorFilter` 的子类。

PorterDuffColorFilter

`PorterDuffColorFilter` 跟 `LightingColorFilter` 一样，只有一个构造方法：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. PorterDuffColorFilter(int color, PorterDuff.Mode mode)
```

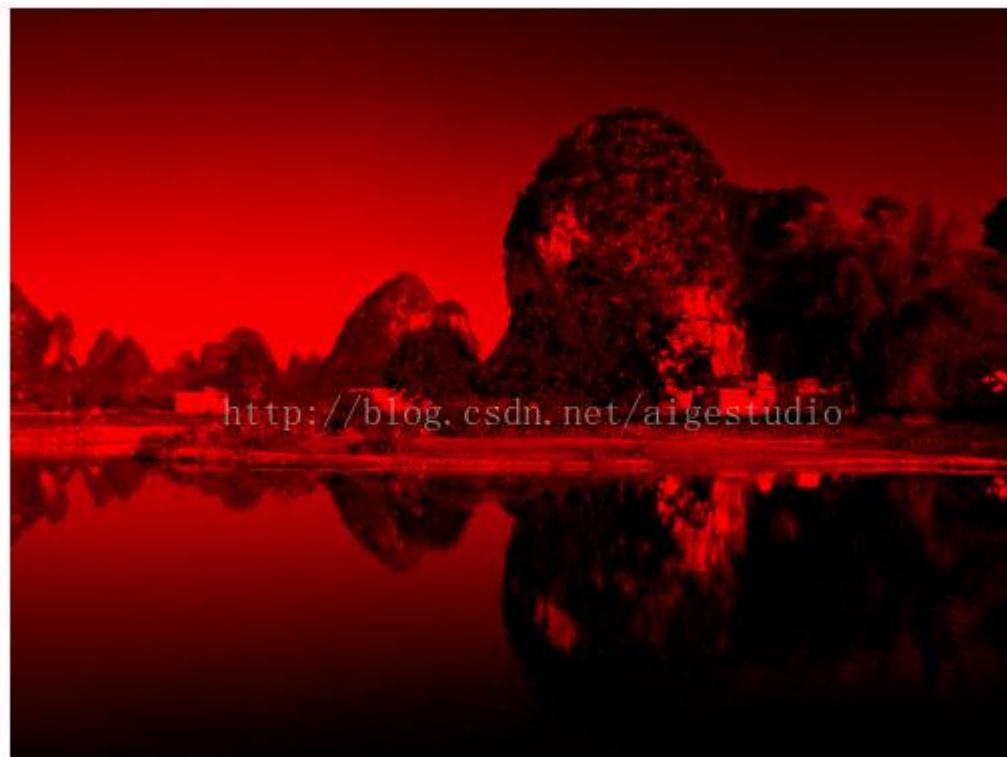
这个构造方法也接受两个值，一个是 16 进制表示的颜色值这个很好理解，而另一个是 PorterDuff 内部类 Mode 中的一个常量值，这个值表示混合模式。那么什么是混合模式呢？混合必定是有两种东西混才行，第一种就是我们设置的 color 值而第二种当然就是我们画布上的元素了！，比如这里我们把 Color 的值设为红色，而模式设为 PorterDuff.Mode.DARKEN 变暗：

[java] [view](#) [plain](#) [copy](#) [print](#)?

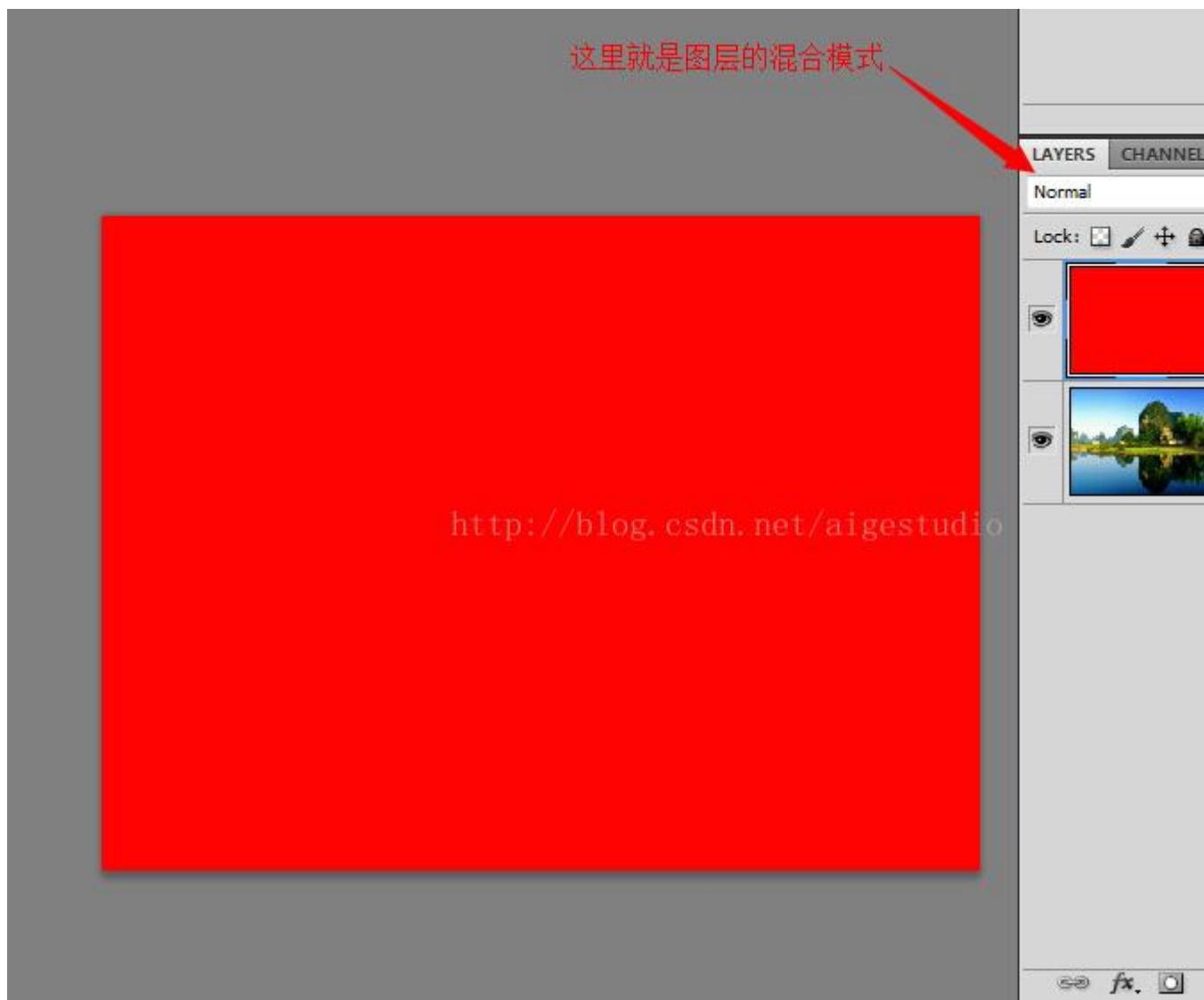
```
1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.     private Bitmap bitmap;// 位图
5.
6.     private int x, y;// 位图绘制时左上角的起点坐标
7.
8.     public CustomView(Context context) {
9.         this(context, null);
10.    }
11.
12.    public CustomView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.        mContext = context;
15.
16.        // 初始化画笔
17.        initPaint();
18.
19.        // 初始化资源
20.        initRes(context);
21.    }
22.
23.    /**
24.     * 初始化画笔
25.     */
26.    private void initPaint() {
27.        // 实例化画笔
28.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
29.
30.        // 设置颜色过滤
31.        mPaint.setColorFilter(new PorterDuffColorFilter(Color.RED, PorterDuf
f.Mode.DARKEN));
32.    }
33.
34.    /**
35.     * 初始化资源
36.     */
37.    private void initRes(Context context) {
```

```
38.         // 获取位图
39.         bitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.a);
40.
41.         /*
42.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
43.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
44.          * 屏幕坐标 y 轴向上偏移位图一半的高度
45.         */
46.         x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - bitmap.get
    etWidth() / 2;
47.         y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - bitmap.g
    etHeight() / 2;
48.     }
49.
50.     @Override
51.     protected void onDraw(Canvas canvas) {
52.         super.onDraw(canvas);
53.
54.         // 绘制位图
55.         canvas.drawBitmap(bitmap, x, y, mPaint);
56.     }
57. }
```

我们尝试在画布上 Draw 刚才的那张图片看看：



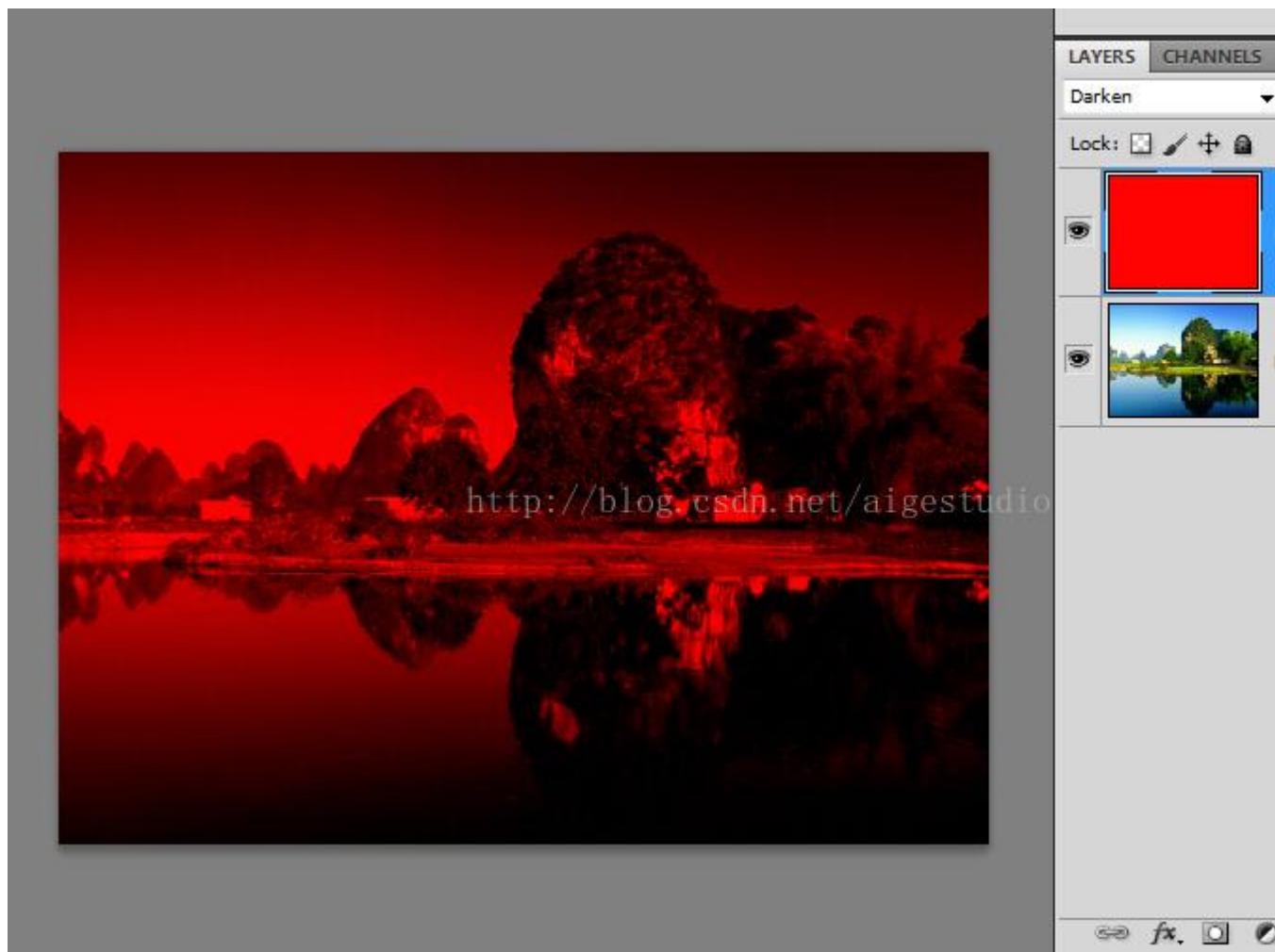
变暗了……也变红了……这就是 `PorterDuff.Mode.DARKEN` 模式给我们的效果，当然 `PorterDuff.Mode` 还有其他很多的混合模式，大家可以尝试，但是这里要注意一点，`PorterDuff.Mode` 中的模式不仅仅是应用于图像色彩混合，还应用于图形混合，比如 `PorterDuff.Mode.DST_OUT` 就表示裁剪混合图，如果我们在 `PorterDuffColorFilter` 中强行设置这些图形混合的模式将不会看到任何对应的效果，关于图形混合我们将在下面详解。为了提升大家的学习兴趣也算是扩展，我跟大家说说 PS 中的图层混合模式，在 PS 中图层的混合模式跟我们 `PorterDuff.Mode` 提供的其实是差不多的但是更霸气强大，同样我们还是使用上面那张图来说明：



如图所示，Layer 1 我们填充了一层红色而 Layer 0 是我们的图片，这时我们选择混合模式为 Darken（默认为 Normal）是不是连名字都跟 Android 的一样：

Normal
Dissolve
Darken
Multiply
Color Burn
Linear Burn
Darker Color
Lighten
Screen
Color Dodge
Linear Dodge (Add)
Lighter Color
Overlay
Soft Light
Hard Light
Vivid Light
Linear Light
Pin Light
Hard Mix
Difference
Exclusion
Subtract
Divide
Hue
Saturation
Color
Luminosity

效果也必须是一样的：



PS 的图层混合模式比 Android 更多更广泛，但两者同名混合模式所产生的效果是一样的，也基于同样的算法原理这里就不多说了。

下面我们来看另一个跟 setColorFilter 有几分相似的方法。

setXfermode(Xfermode xfermode)

Xfermode 国外有大神称之为过渡模式，这种翻译比较贴切但恐怕不易理解，大家也可以直接称之为图像混合模式，因为所谓的“过渡”其实就是图像混合的一种，这个方法跟我们上面讲到的 setColorFilter 蛮相似的，首先它与 set一样没有公开的实现的方法：

[java] view plain copy print?

```
1. public class Xfermode {  
2.  
3.     protected void finalize() throws Throwable {  
4.         try {  
5.             finalizer(native_instance);  
6.         } finally {  
7.             super.finalize();  
8.         }  
9.     }  
}
```

```
10.  
11.     private static native void finalizer(int native_instance);  
12.  
13.     int native_instance;  
14. }
```

同理可得其必然有一定的子类去实现一些方法供我们使用，查看 API 文档发现其果然有三个子类：AvoidXfermode, PixelXorXfermode 和 PorterDuffXfermode，这三个子类实现的功能要比 setColorFilter 的三个子类复杂得多，主要是涉及到图像处理的一些知识可能对大家来说会比较难以理解，不过我会尽量以通俗的方式阐述它们的作用，那好先来看看我们的第一个子类

AvoidXfermode

首先我要告诉大家的是这个 API 因为不支持硬件加速在 API 16 已经过时了（大家可以在 [HardwareAccel](#) 查看那些方法不支持硬件加速）……如果想在高于 API 16 的机子上测试这玩意，必须现在应用或手机设置中关闭硬件加速，在应用中我们可以通过在 AndroidManifest.xml 文件中设置 application 节点下的 android:hardwareAccelerated 属性为 false 来关闭硬件加速：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <application  
2.     android:allowBackup="true"  
3.     android:hardwareAccelerated="false"  
4.     android:icon="@drawable/ic_launcher"  
5.     android:label="@string/app_name"  
6.     android:theme="@android:style/Theme.Black.NoTitleBar.Fullscreen" >  
7.     <activity  
8.         android:name="com.aigestudio.customviewdemo.activities.MainActivity"  
  
9.         android:label="@string/app_name" >  
10.        <intent-filter>  
11.            <action android:name="android.intent.action.MAIN" />  
12.  
13.            <category android:name="android.intent.category.LAUNCHER" />  
14.        </intent-filter>  
15.    </activity>  
16. </application>
```

AvoidXfermode 只有一个含参的构造方法 AvoidXfermode(int opColor, int tolerance, AvoidXfermode.Mode mode)，其具体实现和 ColorFilter 一样都被封装在 C/C++内，它怎么实现我们不管我们只要知道这玩意怎么用就行对吧。AvoidXfermode 有三个参数，第一个

`opColor` 表示一个 16 进制的可以带透明通道的颜色值例如 `0x12345678`, 第二个参数 `tolerance` 表示容差值, 那么什么是容差呢? 你可以理解为一个可以标识“精确”或“模糊”的东西, 待会我们细讲, 最后一个参数表示 `AvoidXfermode` 的具体模式, 其可选值只有两个: `AvoidXfermode.Mode.AVOID` 或者 `AvoidXfermode.Mode.TARGET`, 两者的意思也非常简单, 我们先来看

AvoidXfermode.Mode.TARGET

在该模式下 Android 会判断画布上的颜色是否会有跟 `opColor` 不一样的颜色, 比如我 `opColor` 是红色, 那么在 `TARGET` 模式下就会去判断我们的画布上是否有存在红色的地方, 如果有, 则把该区域“染”上一层我们画笔定义的颜色, 否则不“染”色, 而 `tolerance` 容差值则表示画布上的像素和我们定义的红色之间的差别该是多少的时候才去“染”的, 比如当前画布有一个像素的色值是(200, 20, 13), 而我们的红色值为(255, 0, 0), 当 `tolerance` 容差值为 255 时, 即便(200, 20, 13)并不等于红色值也会被“染”色, 容差值越大“染”色范围越广反之则反, 空说无凭我们来看看具体的实现和效果:

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class CustomView extends View {
2.     private Paint mPaint;// 画笔
3.     private Context mContext;// 上下文环境引用
4.     private Bitmap bitmap;// 位图
5.     private AvoidXfermode avoidXfermode;// AV 模式
6.
7.     private int x, y, w, h;// 位图绘制时左上角的起点坐标
8.
9.     public CustomView(Context context) {
10.         this(context, null);
11.     }
12.
13.     public CustomView(Context context, AttributeSet attrs) {
14.         super(context, attrs);
15.         mContext = context;
16.
17.         // 初始化画笔
18.         initPaint();
19.
20.         // 初始化资源
21.         initRes(context);
22.     }
23.
24.     /**
```

```
25.     * 初始化画笔
26.     */
27.     private void initPaint() {
28.         // 实例化画笔
29.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
30.
31.         /*
32.             * 当画布中有跟 0xFFFFFFFF 色不一样的地方时候才“染”色
33.             */
34.         avoidXfermode = new AvoidXfermode(0xFFFFFFFF, 0, AvoidXfermode.Mode.
35.                                         TARGET);
36.
37.         /**
38.             * 初始化资源
39.             */
40.         private void initRes(Context context) {
41.             // 获取位图
42.             bitmap = BitmapFactory.decodeResource(context.getResources(), R.draw
43.                                         able.a);
44.
45.             /*
46.                 * 计算位图绘制时左上角的坐标使其位于屏幕中心
47.                 * 屏幕坐标 x 轴向左偏移位图一半的宽度
48.                 * 屏幕坐标 y 轴向上偏移位图一半的高度
49.                 */
50.             x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - bitmap.g
51.                 etWidth() / 2;
52.             y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - bitmap.g
53.                 etHeight() / 2;
54.             w = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 + bitmap.g
55.                 etWidth() / 2;
56.             h = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 + bitmap.g
57.                 etHeight() / 2;
58.
59.         }
60.
61.         @Override
62.         protected void onDraw(Canvas canvas) {
63.             super.onDraw(canvas);
64.
65.             // 先绘制位图
66.             canvas.drawBitmap(bitmap, x, y, mPaint);
67.
68.             // “染”什么色是由我们自己决定的
```

```
63.         mPaint.setARGB(255, 211, 53, 243);
64.
65.         // 设置 AV 模式
66.         mPaint.setXfermode(avoidXfermode);
67.
68.         // 画一个位图大小一样的矩形
69.         canvas.drawRect(x, y, w, h, mPaint);
70.     }
71. }
```

在高于 API 16 的测试机上会得到一个矩形的色块 (API 16+的都类似, 改 ROM 和关闭了硬件加速的除外) :



我们再用低于 API 16 (或高于 API 16 但关闭了硬件加速) 的测试机运行就会得到另一个不同的效果:



大家可以看到, 在我们的模式为 TARGET 容差值为 0 的时候此时只有当图片中像色颜色值为 0xFFFFFFFF 的地方才会被染色, 而其他地方不会有改变

AvoidXfermode(0xFFFFFFFF, 0, AvoidXfermode.Mode.TARGET);



而当容差值为 255 的时候只要是跟 0xFFFFFFFF 有点接近的地方都会被染色
而另外一种模式

AvoidXfermode.Mode.AVOID

则与 TARGET 恰恰相反，TARGET 是我们指定的颜色是否与画布的颜色一样，而 AVOID 是我们指定的颜色是否与画布不一样，其他的都与 TARGET 类似
AvoidXfermode(0xFFFFFFFF, 0, AvoidXfermode.Mode.AVOID):



当模式为 AVOID 容差值为 0 时，只有当图片中像素颜色值与 0xFFFFFFFF 完全不一样的地方才会被染色

AvoidXfermode(0xFFFFFFFF, 255, AvoidXfermode.Mode.AVOID):



当容差值为 255 时，只要与 0xFFFFFFFF 稍微有点不一样的地方就会被染色

那么这玩意究竟有什么用呢？比如说当我们只想在白色的区域画点东西或者想把白色区域的地方替换为另一张图片的时候就可以采取这种方式！

Xfermode 的第二个子类

PixelXorXfermode

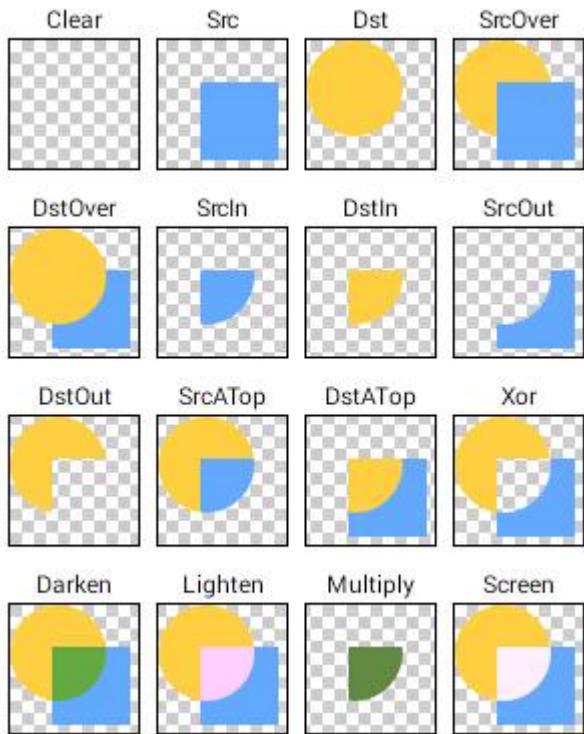
与 **AvoidXfermode** 一样也在 API 16 过时了，该类也提供了一个含参的构造方法

PixelXorXfermode(int opColor), 该类的计算实现很简单，从官方给出的计算公式来看就是：
 $op \wedge src \wedge dst$, 像素色值的按位异或运算，如果大家感兴趣，可以自己用一个纯色去尝试，并自己计算异或运算的值是否与得出的颜色值一样，这里我就不讲了，**Because it was deprecated and useless.**

Xfermode 的最后一个子类也是惟一一个没有过时且沿用至今的子类

PorterDuffXfermode

该类同样有且只有一个含参的构造方法 **PorterDuffXfermode(PorterDuff.Mode mode)**, 这个 **PorterDuff.Mode** 大家看后是否会有些面熟，它跟上面我们讲 **ColorFilter** 时候用到的 **PorterDuff.Mode** 是一样的！麻雀虽小五脏俱全，虽说构造方法的签名列表里只有一个 **PorterDuff.Mode** 的参数，但是它可以实现很多酷毙的图形效果！！而 **PorterDuffXfermode** 就是图形混合模式的意思，其概念最早来自于 SIGGRAPH 的 Tomas Proter 和 Tom Duff，混合图形的概念极大地推动了图形图像学的发展，延伸到计算机图形图像学像 Adobe 和 AutoDesk 公司著名的多款设计软件都可以说一定程度上受到影响，而我们 **PorterDuffXfermode** 的名字也来源于这俩人的人名组合 **PorterDuff**, 那 **PorterDuffXfermode** 能做些什么呢？我们先来看一张 API DEMO 里的图片：



这张图片从一定程度上形象地说明了图形混合的作用，两个图形一圆一方通过一定的计算产生不同的组合效果，在 API 中 Android 为我们提供了 18 种（比上图多了两种 ADD 和 OVERLAY）模式：

PorterDuff.Mode	ADD	Saturate($S + D$)
PorterDuff.Mode	CLEAR	[0, 0]
PorterDuff.Mode	DARKEN	[$Sa + Da - Sa * Da, Sc * (1 - Da) + Dc * (1 - Sa) + \min(Sc, Dc)$]
PorterDuff.Mode	DST	[Da, Dc]
PorterDuff.Mode	DST_ATOP	[$Sa, Sa * Dc + Sc * (1 - Da)$]
PorterDuff.Mode	DST_IN	[$Sa * Da, Sa * Dc$]
PorterDuff.Mode	DST_OUT	[$Da * (1 - Sa), Dc * (1 - Sa)$]
PorterDuff.Mode	DST_OVER	[$Sa + (1 - Sa) * Da, Rc = Dc + (1 - Da) * Sc$]
PorterDuff.Mode	LIGHTEN	[$Sa + Da - Sa * Da, Sc * (1 - Da) + Dc * (1 - Sa) + \max(Sc, Dc)$]
PorterDuff.Mode	MULTIPLY	[$Sa * Da, Sc * Dc$]
PorterDuff.Mode	OVERLAY	
PorterDuff.Mode	SCREEN	[$Sa + Da - Sa * Da, Sc + Dc - Sc * Dc$]
PorterDuff.Mode	SRC	[Sa, Sc]
PorterDuff.Mode	SRC_ATOP	[$Da, Sc * Da + (1 - Sa) * Dc$]
PorterDuff.Mode	SRC_IN	[$Sa * Da, Sc * Da$]
PorterDuff.Mode	SRC_OUT	[$Sa * (1 - Da), Sc * (1 - Da)$]
PorterDuff.Mode	SRC_OVER	[$Sa + (1 - Sa) * Da, Rc = Sc + (1 - Sa) * Dc$]
PorterDuff.Mode	XOR	[$Sa + Da - 2 * Sa * Da, Sc * (1 - Da) + (1 - Sa) * Dc$]

来定义不同的混合效果，这 18 种模式 Android 还为我们提供了它们的计算方式比如 LIGHTEN 的计算方式为 [$Sa + Da - Sa * Da, Sc * (1 - Da) + Dc * (1 - Sa) + \max(Sc, Dc)$]，其中 Sa 全称为 Source alpha 表示源图的 Alpha 通道； Sc 全称为 Source color 表示源图的颜色； Da 全称为 Destination alpha 表示目标图的 Alpha 通道； Sa 全称为 Destination color 表示目标图的颜色，细心的朋友会发现“[……]”里分为两部分，其中“,”前的部分为“ $Sa + Da - Sa * Da$ ”这一部分的值代表计算后的 Alpha 通道而“,”后的部分为“ $Sc * (1 - Da) + Dc * (1 - Sa) + \max(Sc, Dc)$ ”这一部分的值代表计算后的颜色值，图形混合后的图片依靠这个矢量来计算 ARGB 的值，如果大家感兴趣可以查看维基百科中对 Alpha 合成的解释：

http://en.wikipedia.org/wiki/Alpha_compositing。作为一个猿，我们不需要知道复杂的图形学计算但是一定要知道这些模式会为我们提供怎样的效果，当大家看到上面 API DEMO 给出的效果时一定会觉得 PorterDuffXfermode 其实就是简单的图形交并集计算，比如重叠的部分删掉或者叠加等等，事实上呢！PorterDuffXfermode 的计算绝非是根据于此！上面我们也说了 PorterDuffXfermode 的计算是要根据具体的 Alpha 值和 RGB 值的，既然如此，我们就来看一个比 API DEMO 稍微复杂的例子来更有力地说明 PorterDuffXfermode 是如何

工作而我们又能用它做些什么，在这个例子中我将用到两个带有 Alpha 通道的渐变图形

Bitmap:



我们将在不同的模式下混合这两个 Bitmap 来看看这两个渐变色的颜色值在不同的混合模式下究竟发生了什么？先看看我们的测试代码：

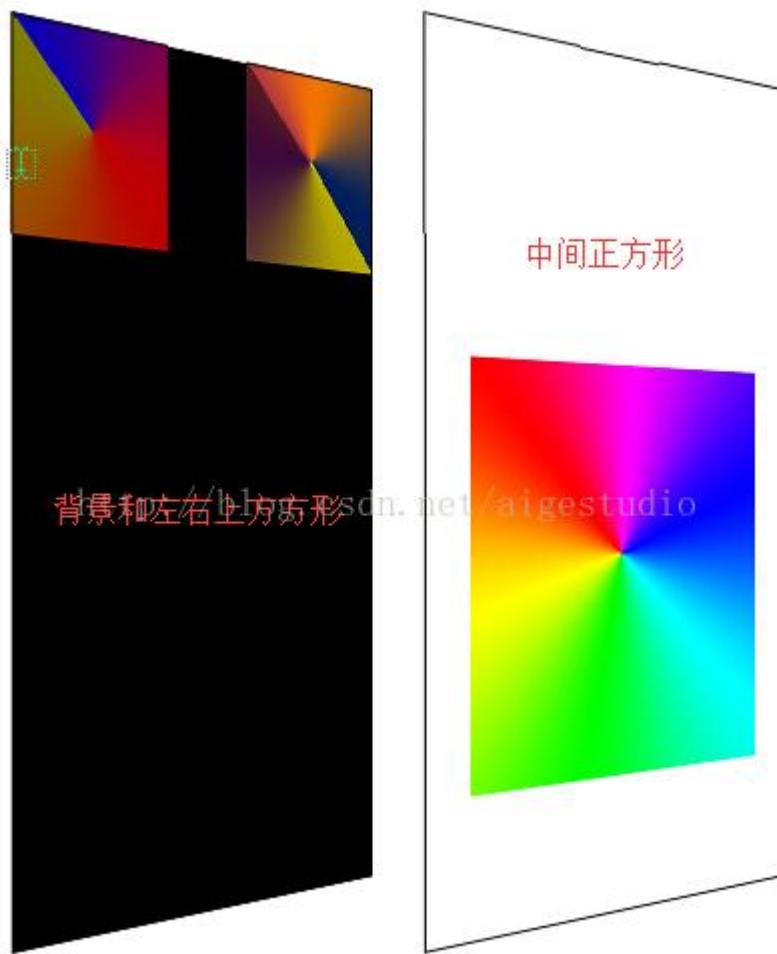
[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1.  @TargetApi(Build.VERSION_CODES.HONEYCOMB)
2.  public class PorterDuffView extends View {
3.      /*
4.       * PorterDuff 模式常量
5.       * 可以在此更改不同的模式测试
6.       */
7.      private static final PorterDuff.Mode MODE = PorterDuff.Mode.ADD;
8.
9.      private static final int RECT_SIZE_SMALL = 400;// 左右上方示例渐变正方形的
尺寸大小
10.     private static final int RECT_SIZE_BIG = 800;// 中间测试渐变正方形的尺寸大
小
11.
12.     private Paint mPaint;// 画笔
13.
14.     private PorterDuffBO porterDuffBO;// PorterDuffView 类的业务对象
15.     private PorterDuffXfermode porterDuffXfermode;// 图形混合模式
16.
17.     private int screenW, screenH;// 屏幕尺寸
18.     private int s_l, s_t;// 左上方正方形的原点坐标
19.     private int d_l, d_t;// 右上方正方形的原点坐标
20.     private int rectX, rectY;// 中间正方形的原点坐标
21.
22.     public PorterDuffView(Context context, AttributeSet attrs) {
23.         super(context, attrs);
24.
25.         // 实例化画笔并设置抗锯齿
26.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
27.     }
```

```
28.         // 实例化业务对象
29.         porterDuffBO = new PorterDuffBO();
30.
31.         // 实例化混合模式
32.         porterDuffXfermode = new PorterDuffXfermode(MODE);
33.
34.         // 计算坐标
35.         calu(context);
36.     }
37.
38. /**
39. * 计算坐标
40. *
41. * @param context
42. *          上下文环境引用
43. */
44. private void calu(Context context) {
45.     // 获取包含屏幕尺寸的数组
46.     int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
47.
48.     // 获取屏幕尺寸
49.     screenW = screenSize[0];
50.     screenH = screenSize[1];
51.
52.     // 计算左上方正方形原点坐标
53.     s_l = 0;
54.     s_t = 0;
55.
56.     // 计算右上方正方形原点坐标
57.     d_l = screenW - RECT_SIZE_SMALL;
58.     d_t = 0;
59.
60.     // 计算中间方正方形原点坐标
61.     rectX = screenW / 2 - RECT_SIZE_BIG / 2;
62.     rectY = RECT_SIZE_SMALL + (screenH - RECT_SIZE_SMALL) / 2 - RECT_SIZE_
63.             BIG / 2;
64. }
65. @Override
66. protected void onDraw(Canvas canvas) {
67.     super.onDraw(canvas);
68.     // 设置画布颜色为黑色以便我们更好地观察
69.     canvas.drawColor(Color.BLACK);
70.
```

```
71.         // 设置业务对象尺寸值计算生成左右上方的渐变方形
72.         porterDuffB0.setSize(RECT_SIZE_SMALL);
73.
74.         /*
75.          * 画出左右上方两个正方形
76.          * 其中左边的为 src 右边的为 dis
77.          */
78.         canvas.drawBitmap(porterDuffB0.initSrcBitmap(), s_l, s_t, mPaint);
79.         canvas.drawBitmap(porterDuffB0.initDisBitmap(), d_l, d_t, mPaint);
80.
81.         /*
82.          * 将绘制操作保存到新的图层（更官方的说法应该是离屏缓存）我们将在 1/3 中学习
83.          * 到 Canvas 的全部用法这里就先 follow me
84.          */
85.
86.         // 重新设置业务对象尺寸值计算生成中间的渐变方形
87.         porterDuffB0.setSize(RECT_SIZE_BIG);
88.
89.         // 先绘制 dis 目标图
90.         canvas.drawBitmap(porterDuffB0.initDisBitmap(), rectX, rectY, mPaint
91. );
92.
93.         // 设置混合模式
94.         mPaint.setXfermode(porterDuffXfermode);
95.
96.         // 再绘制 src 源图
97.         canvas.drawBitmap(porterDuffB0.initSrcBitmap(), rectX, rectY, mPaint
98. );
99.
100.
101.        // 还原画布
102.        canvas.restoreToCount(sc);
103.    }
104. }
```

代码中我们使用到了 View 的离屏缓冲，也通俗地称之为层，这个概念很简单，我们在绘图的时候新建一个“层”，所有的绘制操作都在该层上而不影响该层以外的图像，比如代码中我们在绘制了画布颜色和左右上方两个方形后就新建了一个图层来绘制中间的大正方形，这个方形和左右上方的方形是在两个不同的层上的：



注：图中所显示色彩效果与我们的代码不同，上图只为演示图层概念

当我们绘制完成后要通过 `restore` 将所有缓冲（层）中的绘制操作还原到画布以结束绘制，具体关于画布的知识在自定义控件其实很简单(4)，这里就不多说了，下面我们看具体各种模式的计算效果

PS： `Src` 为源图像，意为将要绘制的图像； `Dis` 为目标图像，意为我们将要把源图像绘制到的图像……是不是感脚很拗口 == ! Fuck……意会意会~~

PorterDuff.Mode.ADD

计算方式：`Saturate(S + D)`； Chinese：饱和相加



从计算方式和显示的结果我们可以看到，ADD 模式简单来说就是对图像饱和度进行相加，这个模式在应用中不常用，我唯一一次使用它是通过代码控制 RGB 通道的融合生成图片。

PorterDuff.Mode.CLEAR

计算方式: [0, 0]; Chinese: 清除
清除图像，很好理解不扯了。

PorterDuff.Mode.DARKEN

计算方式: [$Sa + Da - Sa \cdot Da, Sc \cdot (1 - Da) + Dc \cdot (1 - Sa) + \min(Sc, Dc)$]; Chinese: 变暗



这个模式计算方式目测很复杂，其实效果很好理解，两个图像混合，较深的颜色总是会覆盖较浅的颜色，如果两者深浅相同则混合，如图，黄色覆盖了红色而蓝色和青色因为是跟透明混合所以不变。细心的朋友会发现青色和黄色之间有一层类似橙色的过渡色，这就是混合的结果。在实际的测试中源图和目标图的 **DARKEN** 混合偶尔会有相反的结果比如红色覆盖了黄色，这源于 **Android** 对颜色值“深浅”的定义，我暂时没有在官方查到有关资料不知道是否与图形图像学一致。**DARKEN** 模式应用在图像色彩方面比较广泛我们可以利用其特性来获得不同的成像效果，这点与之前介绍的 **ColorFilter** 有点类似。

PorterDuff.Mode.DST

计算方式: [Da, Dc]; Chinese: 只绘制目标图像



如 Chinese 所说，很好理解。

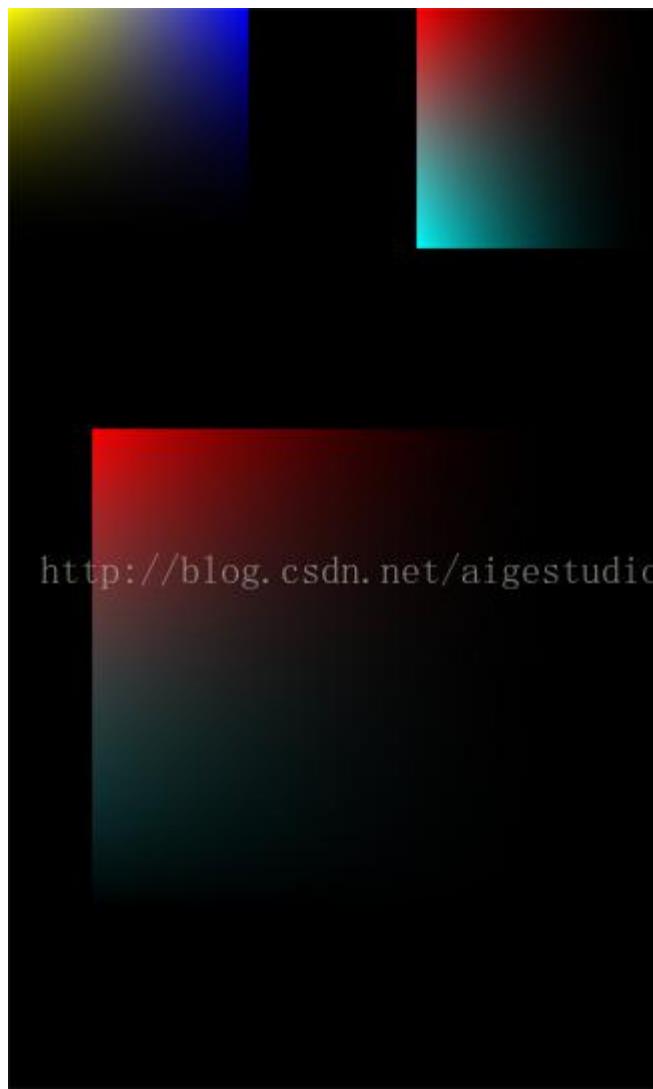
PorterDuff.Mode.DST_ATOP

计算方式: $[Sa, Sa * Dc + Sc * (1 - Da)]$; Chinese: 在源图像和目标图像相交的地方绘制目标图像而在不相交的地方绘制源图像

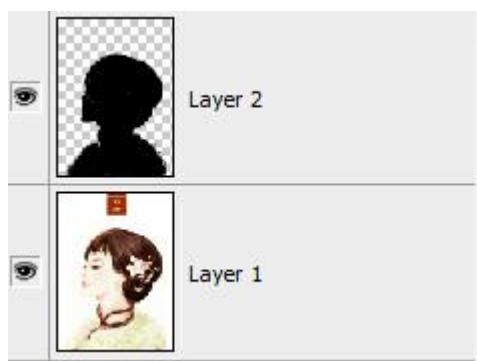


PorterDuff.Mode.DST_IN

计算方式: $[Sa * Da, Sa * Dc]$; Chinese: 只在源图像和目标图像相交的地方绘制目标图像

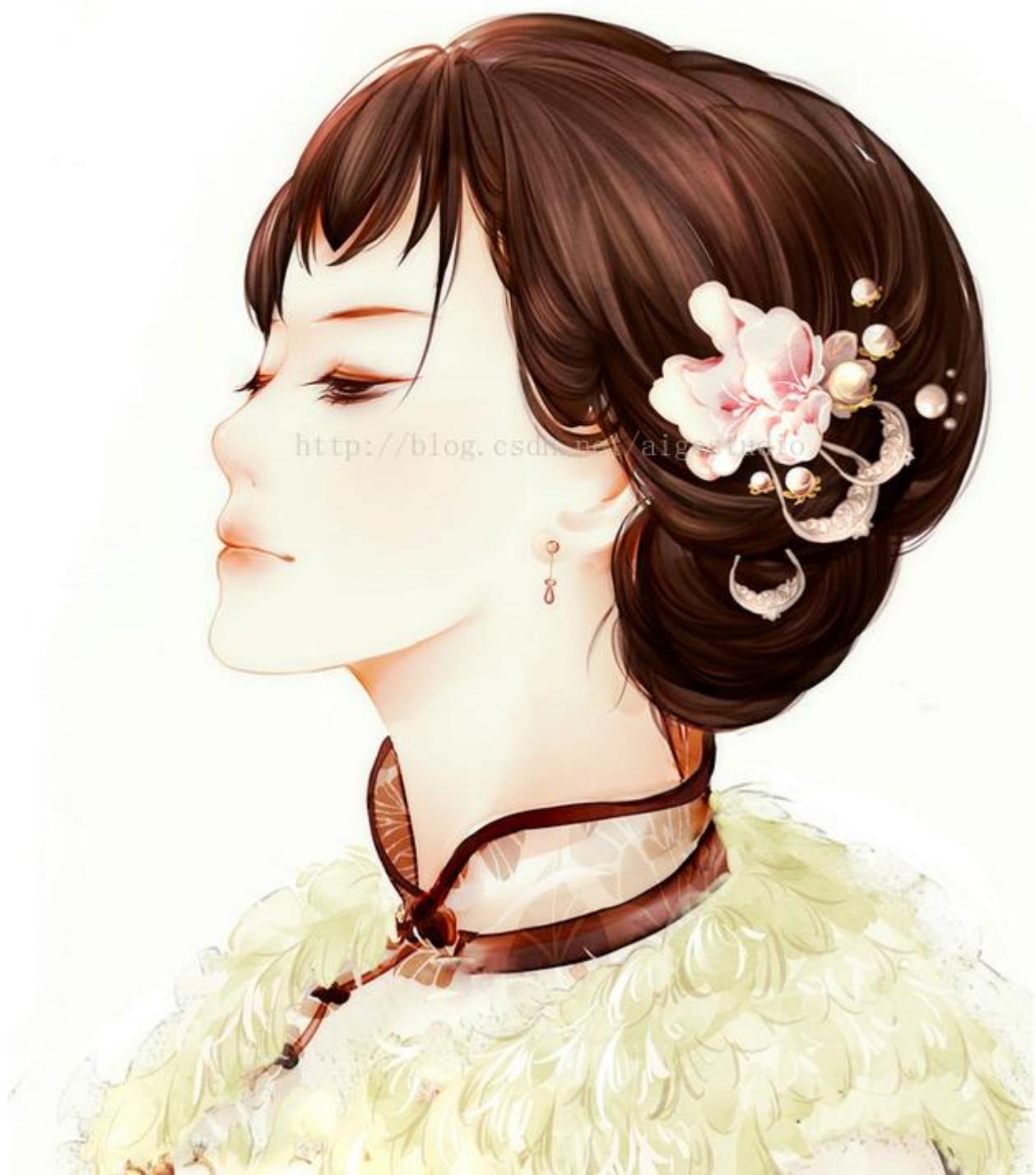


最常见的应用就是蒙板绘制，利用源图作为蒙板“抠出”目标图上的图像，这里我讲一个很简单的例子，如果家用过 PS 就很容易理解，我这里有两张图：



一张是一个很漂亮的手绘古典美女：

口
君



http://blog.csdn.net/aiguo_1980

而另一张是一张只有黑色和透明通道的遮罩图：



<http://blog.csdn.net/aigestudio>

我们把这张美女图画在我们的屏幕上：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class DisInView extends View {
2.     private Paint mPaint;// 画笔
3.     private Bitmap bitmapDis;// 位图
4.
5.     private int x, y;// 位图绘制时左上角的起点坐标
6.     private int screenW, screenH;// 屏幕尺寸
7.
8.     public DisInView(Context context) {
9.         this(context, null);
10.    }
11.
12.    public DisInView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.
15.        // 初始化画笔
16.        initPaint();
17.
18.        // 初始化资源
19.        initRes(context);
20.    }
21.
22.    /**
23.     * 初始化画笔
24.     */
25.    private void initPaint() {
26.        // 实例化画笔
27.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
28.    }
29.
30.    /**
31.     * 初始化资源
32.     */
33.    private void initRes(Context context) {
34.        // 获取位图
35.        bitmapDis = BitmapFactory.decodeResource(context.getResources(), R.d
rawable.a3);
36.
37.        // 获取包含屏幕尺寸的数组
38.        int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
39.
40.        // 获取屏幕尺寸
```

```
41.         screenW = screenSize[0];
42.         screenH = screenSize[1];
43.
44.         /*
45.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
46.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
47.          * 屏幕坐标 y 轴向上偏移位图一半的高度
48.          */
49.         x = screenW / 2 - bitmapDis.getWidth() / 2;
50.         y = screenH / 2 - bitmapDis.getHeight() / 2;
51.
52.     }
53.
54.     @Override
55.     protected void onDraw(Canvas canvas) {
56.         super.onDraw(canvas);
57.
58.         // 绘制美女图
59.         canvas.drawBitmap(bitmapDis, x, y, mPaint);
60.     }
61. }
```

运行后如下：



<http://blog.csdn.net/aigestudio>

美女脑袋上有个文字标识巨恶心而且因为图片画质问题美图周围还有一片淡黄色的不好看，那我们就通过刚才那个黑色的透明通道图把美女“抠”出来：

[java] view plaincopyprint?

```
1. public class DisInView extends View {  
2.     private Paint mPaint;// 画笔  
3.     private Bitmap bitmapDis, bitmapSrc;// 位图  
4.     private PorterDuffXfermode porterDuffXfermode;// 图形混合模式  
5.  
6.     private int x, y;// 位图绘制时左上角的起点坐标  
7.     private int screenW, screenH;// 屏幕尺寸  
8.  
9.     public DisInView(Context context) {  
10.         this(context, null);  
11.     }  
12. }
```

```
13.     public DisInView(Context context, AttributeSet attrs) {
14.         super(context, attrs);
15.
16.         // 实例化混合模式
17.         porterDuffXfermode = new PorterDuffXfermode(PorterDuff.Mode.DST_IN);
18.
19.         // 初始化画笔
20.         initPaint();
21.
22.         // 初始化资源
23.         initRes(context);
24.     }
25.
26. /**
27. * 初始化画笔
28. */
29. private void initPaint() {
30.     // 实例化画笔
31.     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
32. }
33.
34. /**
35. * 初始化资源
36. */
37. private void initRes(Context context) {
38.     // 获取位图
39.     bitmapDis = BitmapFactory.decodeResource(context.getResources(), R.drawable.a3);
40.     bitmapSrc = BitmapFactory.decodeResource(context.getResources(), R.drawable.a3_mask);
41.
42.     // 获取包含屏幕尺寸的数组
43.     int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
44.
45.     // 获取屏幕尺寸
46.     screenW = screenSize[0];
47.     screenH = screenSize[1];
48.
49.     /*
50.      * 计算位图绘制时左上角的坐标使其位于屏幕中心
51.      * 屏幕坐标 x 轴向左偏移位图一半的宽度
52.      * 屏幕坐标 y 轴向上偏移位图一半的高度
53.      */

```

```
54.         x = screenW / 2 - bitmapDis.getWidth() / 2;
55.         y = screenH / 2 - bitmapDis.getHeight() / 2;
56.
57.     }
58.
59.     @Override
60.     protected void onDraw(Canvas canvas) {
61.         super.onDraw(canvas);
62.         canvas.drawColor(Color.WHITE);
63.
64.         /*
65.          * 将绘制操作保存到新的图层（更官方的说法应该是离屏缓存）我们将在 1/3 中学习
66.          * 到 Canvas 的全部用法这里就先 follow me
67.         */
68.
69.         // 先绘制 dis 目标图
70.         canvas.drawBitmap(bitmapDis, x, y, mPaint);
71.
72.         // 设置混合模式
73.         mPaint.setXfermode(porterDuffXfermode);
74.
75.         // 再绘制 src 源图
76.         canvas.drawBitmap(bitmapSrc, x, y, mPaint);
77.
78.         // 还原混合模式
79.         mPaint.setXfermode(null);
80.
81.         // 还原画布
82.         canvas.restoreToCount(sc);
83.     }
84. }
```

看！只剩米女了~~~~:



<http://blog.csdn.net/aigestudio>

当然该混合模式的用法绝不止这么简单，这只是阐述了一个原理，更棒的用法就看你怎么用了~~~~

PorterDuff.Mode.DST_OUT

计算方式: $[Da * (1 - Sa), Dc * (1 - Sa)]$; Chinese: 只在源图像和目标图像不相交的地方绘制目标图像



上面那个例子呢我们把米女抠了出来，而这次我们将从一个色块中把米女的轮廓挖出来
~~~~啦啦啦：

[java] view plaincopyprint?

```
1. public class DisOutView extends View {  
2.     private Paint mPaint;// 画笔  
3.     private Bitmap bitmapSrc;// 位图  
4.     private PorterDuffXfermode porterDuffXfermode;// 图形混合模式  
5.  
6.     private int x, y;// 位图绘制时左上角的起点坐标  
7.     private int screenW, screenH;// 屏幕尺寸  
8.  
9.     public DisOutView(Context context) {  
10.         this(context, null);  
11.     }  
12.  
13.     public DisOutView(Context context, AttributeSet attrs) {  
14.         super(context, attrs);
```

```
15.          // 实例化混合模式
16.          porterDuffXfermode = new PorterDuffXfermode(PorterDuff.Mode.DST_OUT)
17.      ;
18.
19.          // 初始化画笔
20.          initPaint();
21.
22.          // 初始化资源
23.          initRes(context);
24.      }
25.
26.      /**
27.      * 初始化画笔
28.      */
29.      private void initPaint() {
30.          // 实例化画笔
31.          mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
32.      }
33.
34.      /**
35.      * 初始化资源
36.      */
37.      private void initRes(Context context) {
38.          // 获取位图
39.          bitmapSrc = BitmapFactory.decodeResource(context.getResources(), R.d
rawable.a3_mask);
40.
41.          // 获取包含屏幕尺寸的数组
42.          int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
43.
44.          // 获取屏幕尺寸
45.          screenW = screenSize[0];
46.          screenH = screenSize[1];
47.
48.          /*
49.          * 计算位图绘制时左上角的坐标使其位于屏幕中心
50.          * 屏幕坐标 x 轴向左偏移位图一半的宽度
51.          * 屏幕坐标 y 轴向上偏移位图一半的高度
52.          */
53.          x = screenW / 2 - bitmapSrc.getWidth() / 2;
54.          y = screenH / 2 - bitmapSrc.getHeight() / 2;
55.
56.      }
```

```
57.  
58.    @Override  
59.    protected void onDraw(Canvas canvas) {  
60.        super.onDraw(canvas);  
61.        canvas.drawColor(Color.WHITE);  
62.  
63.        /*  
64.           * 将绘制操作保存到新的图层（更官方的说法应该是离屏缓存）我们将在 1/3 中学习  
65.           到 Canvas 的全部用法这里就先 follow me  
66.           */  
66.        int sc = canvas.saveLayer(0, 0, screenW, screenH, null, Canvas.ALL_S  
       AVE_FLAG);  
67.  
68.        // 先绘制一层颜色  
69.        canvas.drawColor(0xFF8f66DA);  
70.  
71.        // 设置混合模式  
72.        mPaint.setXfermode(porterDuffXfermode);  
73.  
74.        // 再绘制 src 源图  
75.        canvas.drawBitmap(bitmapSrc, x, y, mPaint);  
76.  
77.        // 还原混合模式  
78.        mPaint.setXfermode(null);  
79.  
80.        // 还原画布  
81.        canvas.restoreToCount(sc);  
82.    }  
83. }
```

看看美女那动人的轮廓~~~~~么么哒:



<http://blog.csdn.net/aigestudio>

**PorterDuff.Mode.DST\_OVER**

计算方式:  $[Sa + (1 - Sa)*Da, Rc = Dc + (1 - Da)*Sc]$ ; Chinese: 在源图像的上方绘制目标图像



这个就不说啦，就是两个图片谁在上谁在下的意思

**PorterDuff.Mode.LIGHTEN**

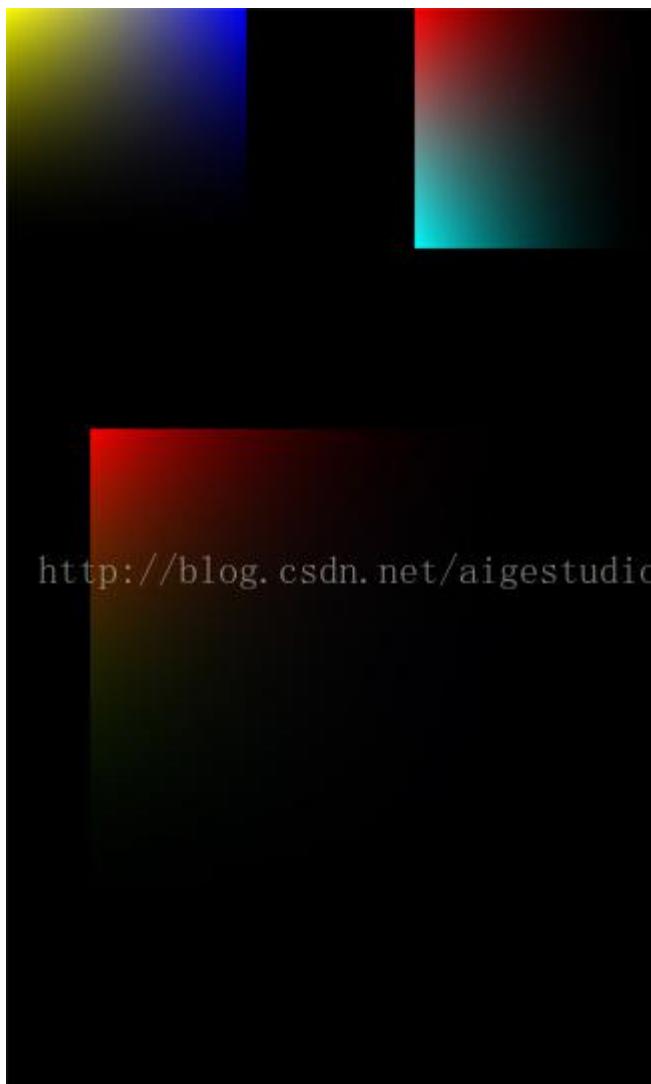
计算方式:  $[Sa + Da - Sa \cdot Da, Sc \cdot (1 - Da) + Dc \cdot (1 - Sa) + \max(Sc, Dc)]$ ; Chinese: 变亮



与 DARKEN 相反，不多说了

**PorterDuff.Mode.MULTIPLY**

计算方式:  $[Sa * Da, Sc * Dc]$ ; Chinese: 正片叠底



该模式通俗的计算方式很简单，源图像素颜色值乘以目标图像素颜色值除以 255 即得混合后图像像素的颜色值，该模式在设计领域应用广泛，因为其特性黑色与任何颜色混合都会得黑色，在手绘的上色、三维动画的 UV 贴图绘制都有应用，具体效果大家自己尝试我就不说了

#### **PorterDuff.Mode.OVERLAY**

计算方式：未给出； Chinese：叠加



<http://blog.csdn.net/aigestudio>

这个模式没有在官方的 API DEMO 中给出，谷歌也没有给出其计算方式，在实际效果中其对亮色和暗色不起作用，也就是说黑白色无效，它会将源色与目标色混合产生一种中间色，这种中间色生成的规律也很简单，如果源色比目标色暗，那么让目标色的颜色倍增否则颜色递减。

#### **PorterDuff.Mode.SCREEN**

计算方式:  $[Sa + Da - Sa * Da, Sc + Dc - Sc * Dc]$ ; Chinese: 滤色



<http://blog.csdn.net/aigestudio>

计算方式我不解释了，滤色产生的效果我认为是 Android 提供的几个色彩混合模式中最好的，它可以让图像焦躁幻化，有一种色调均和的感觉：

[java] view plain copy print?

```
1. public class ScreenView extends View {  
2.     private Paint mPaint;// 画笔  
3.     private Bitmap bitmapSrc;// 位图  
4.     private PorterDuffXfermode porterDuffXfermode;// 图形混合模式  
5.  
6.     private int x, y;// 位图绘制时左上角的起点坐标  
7.     private int screenW, screenH;// 屏幕尺寸  
8.  
9.     public ScreenView(Context context) {  
10.         this(context, null);  
11.     }  
12.  
13.     public ScreenView(Context context, AttributeSet attrs) {  
14.         super(context, attrs);
```

```
15.  
16.        // 实例化混合模式  
17.        porterDuffXfermode = new PorterDuffXfermode(PorterDuff.Mode.SCREEN);  
  
18.  
19.        // 初始化画笔  
20.        initPaint();  
21.  
22.        // 初始化资源  
23.        initRes(context);  
24.    }  
25.  
26.    /**  
27.     * 初始化画笔  
28.     */  
29.    private void initPaint() {  
30.        // 实例化画笔  
31.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
32.    }  
33.  
34.    /**  
35.     * 初始化资源  
36.     */  
37.    private void initRes(Context context) {  
38.        // 获取位图  
39.        bitmapSrc = BitmapFactory.decodeResource(context.getResources(), R.d  
rawable.a3);  
40.  
41.        // 获取包含屏幕尺寸的数组  
42.        int[] screenSize = MeasureUtil.getScreenSize((Activity) context);  
43.  
44.        // 获取屏幕尺寸  
45.        screenW = screenSize[0];  
46.        screenH = screenSize[1];  
47.  
48.        /*  
49.         * 计算位图绘制时左上角的坐标使其位于屏幕中心  
50.         * 屏幕坐标 x 轴向左偏移位图一半的宽度  
51.         * 屏幕坐标 y 轴向上偏移位图一半的高度  
52.         */  
53.        x = screenW / 2 - bitmapSrc.getWidth() / 2;  
54.        y = screenH / 2 - bitmapSrc.getHeight() / 2;  
55.  
56.    }
```

```
57.  
58.    @Override  
59.    protected void onDraw(Canvas canvas) {  
60.        super.onDraw(canvas);  
61.        canvas.drawColor(Color.WHITE);  
62.  
63.        /*  
64.         * 将绘制操作保存到新的图层（更官方的说法应该是离屏缓存）我们将在 1/3 中学习  
65.         * 到 Canvas 的全部用法这里就先 follow me  
66.         */  
66.        int sc = canvas.saveLayer(0, 0, screenW, screenH, null, Canvas.ALL_S  
67.                                     AVE_FLAG);  
67.  
68.        // 先绘制一层带透明度的颜色  
69.        canvas.drawColor(0xcc1c093e);  
70.  
71.        // 设置混合模式  
72.        mPaint.setXfermode(porterDuffXfermode);  
73.  
74.        // 再绘制 src 源图  
75.        canvas.drawBitmap(bitmapSrc, x, y, mPaint);  
76.  
77.        // 还原混合模式  
78.        mPaint.setXfermode(null);  
79.  
80.        // 还原画布  
81.        canvas.restoreToCount(sc);  
82.    }  
83. }
```

它比原图多一层蓝紫色调给人感觉更古典~~



#### PorterDuff.Mode.SRC

计算方式: [Sa, Sc]; Chinese: 显示源图

只绘制源图, SRC 类的模式跟 DIS 的其实差不多就不多说了, 大家多动手自己试试, 我已经写不动了

快.....

.....

#### PorterDuff.Mode.SRC\_ATOP

计算方式: [Da, Sc \* Da + (1 - Sa) \* Dc]; Chinese: 在源图像和目标图像相交的地方绘制源图像, 在不相交的地方绘制目标图像



#### PorterDuff.Mode.SRC\_IN

计算方式:  $[Sa * Da, Sc * Da]$ ; Chinese: 只在源图像和目标图像相交的地方绘制源图像



**PorterDuff.Mode.SRC\_OUT**

计算方式:  $[Sa * (1 - Da), Sc * (1 - Da)]$ ; Chinese: 只在源图像和目标图像不相交的地方绘制源图像



<http://blog.csdn.net/aigestudio>

#### PorterDuff.Mode.SRC\_OVER

计算方式:  $[Sa + (1 - Sa) * Da, Rc = Sc + (1 - Sa) * Dc]$ ; Chinese: 在目标图像的顶部绘制源图像



#### PorterDuff.Mode.XOR

计算方式:  $[Sa + Da - 2 * Sa * Da, Sc * (1 - Da) + (1 - Sa) * Dc]$ ; Chinese: 在源图像和目标图像重叠之外的任何地方绘制他们, 而在不重叠的地方不绘制任何内容



XOR 我们在将 PixelXorXfermode 的时候提到过，不了解的话上去看看~~~实在写不动了那么这些混合模式究竟有什么用？能够给我们带来什么好处呢？假设我们要画一个闹钟，如下：



注：该闹钟图标为已投入运行项目文件并已有商标，请大家不要以任何盈利手段为目的盗用，当然做练习是没问题的

构思一下怎么做，我们需要画一个圆圈做钟体，两个 Path（Path 为路径，我们将会在 1/3 详细学习到，这里就先 follow me）作为指针，问题是两个铃铛~~~~如果我们不会混合模式，

一定会想怎么计算坐标啊去绘制曲线然后闭合然后填充啊之类.....实际上有必要吗？这个闹铃不就是一个小圆再用一个大圆去遮罩吗：



问题是不是一下子就变简单了？如果等你去计算怎么画路径怎么闭合曲线填充颜色还有多屏幕的匹配.....哥已经死了又活过来又死了.....在学完 1/2 的 View 尺寸计算和布局后我会教大家如何做类似的 View 并匹配在所有的屏幕上~~~~这里就先缓一缓。大家一定要有这样的思维，当你想要去画一个 View 的时候一定要想想看这个 View 的图形是不是可以通过基本的几何图形混合来生成，如果可以，那么恭喜你，使用 PorterDuffXfermode 的混合模式你可以事半功倍！

PorterDuffXfermode 的另一个比较常见的应用就是橡皮擦效果，我们可以通过手指不断地触摸屏幕绘制 Path，再以 Path 作遮罩遮掉填充的色块显示下层的图像：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class EraserView extends View {  
2.     private static final int MIN_MOVE_DIS = 5;// 最小的移动距离：如果我们手指在  
    屏幕上的移动距离小于此值则不会绘制  
3.  
4.     private Bitmap fgBitmap, bgBitmap;// 前景橡皮擦的 Bitmap 和背景我们底图的  
    Bitmap  
5.     private Canvas mCanvas;// 绘制橡皮擦路径的画布  
6.     private Paint mPaint;// 橡皮擦路径画笔  
7.     private Path mPath;// 橡皮擦绘制路径  
8.  
9.     private int screenW, screenH;// 屏幕宽高  
10.    private float preX, preY;// 记录上一个触摸事件的位置坐标  
11.  
12.    public EraserView(Context context, AttributeSet set) {  
13.        super(context, set);  
14.    }
```

```
15.         // 计算参数
16.         cal(context);
17.
18.         // 初始化对象
19.         init(context);
20.     }
21.
22.     /**
23.      * 计算参数
24.      *
25.      * @param context
26.      *          上下文环境引用
27.      */
28.     private void cal(Context context) {
29.         // 获取屏幕尺寸数组
30.         int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
31.
32.         // 获取屏幕宽高
33.         screenW = screenSize[0];
34.         screenH = screenSize[1];
35.     }
36.
37.     /**
38.      * 初始化对象
39.      */
40.     private void init(Context context) {
41.         // 实例化路径对象
42.         mPath = new Path();
43.
44.         // 实例化画笔并开启其抗锯齿和抗抖动
45.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
46.
47.         // 设置画笔透明度为 0 是关键！我们要让绘制的路径是透明的，然后让该路径与前景
48.         // 的底色混合“抠”出绘制路径
49.         mPaint.setARGB(128, 255, 0, 0);
50.
51.         // 设置混合模式为 DST_IN
52.         mPaint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.DST_IN));
53.
54.         // 设置画笔风格为描边
55.         mPaint.setStyle(Paint.Style.STROKE);
56.
57.         // 设置路径结合处样式
```

```
57.         mPaint.setStrokeJoin(Paint.Join.ROUND);
58.
59.         // 设置笔触类型
60.         mPaint.setStrokeCap(Paint.Cap.ROUND);
61.
62.         // 设置描边宽度
63.         mPaint.setStrokeWidth(50);
64.
65.         // 生成前景图 Bitmap
66.         fgBitmap = Bitmap.createBitmap(screenW, screenH, Config.ARGB_8888);

67.
68.         // 将其注入画布
69.         mCanvas = new Canvas(fgBitmap);
70.
71.         // 绘制画布背景为中性灰
72.         mCanvas.drawColor(0xFF808080);
73.
74.         // 获取背景底图 Bitmap
75.         bgBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.a4);
76.
77.         // 缩放背景底图 Bitmap 至屏幕大小
78.         bgBitmap = Bitmap.createScaledBitmap(bgBitmap, screenW, screenH, true);
79.     }
80.
81.     @Override
82.     protected void onDraw(Canvas canvas) {
83.         // 绘制背景
84.         canvas.drawBitmap(bgBitmap, 0, 0, null);
85.
86.         // 绘制前景
87.         canvas.drawBitmap(fgBitmap, 0, 0, null);
88.
89.         /*
90.          * 这里要注意 canvas 和 mCanvas 是两个不同的画布对象
91.          * 当我们在屏幕上移动手指绘制路径时会把路径通过 mCanvas 绘制到 fgBitmap 上
92.          * 每当我们手指移动一次均会将路径 mPath 作为目标图像绘制到 mCanvas 上，而在
93.          * 上面我们先在 mCanvas 上绘制了中性灰色
94.          * 两者会因为 DST_IN 模式下的计算只显示中性灰，但是因为 mPath 的透明，计算生成
95.          * 的混合图像也会是透明的
96.          * 所以我们会得到“橡皮擦”的效果
97.         */
```

```
96.         mCanvas.drawPath(mPath, mPaint);
97.     }
98.
99.     /**
100.      * View 的事件将会在 7/12 详解
101.     */
102.    @Override
103.    public boolean onTouchEvent(MotionEvent event) {
104.        /*
105.         * 获取当前事件位置坐标
106.         */
107.        float x = event.getX();
108.        float y = event.getY();
109.
110.        switch (event.getAction()) {
111.            case MotionEvent.ACTION_DOWN:// 手指接触屏幕重置路径
112.                mPath.reset();
113.                mPath.moveTo(x, y);
114.                preX = x;
115.                preY = y;
116.                break;
117.            case MotionEvent.ACTION_MOVE:// 手指移动时连接路径
118.                float dx = Math.abs(x - preX);
119.                float dy = Math.abs(y - preY);
120.                if (dx >= MIN_MOVE_DIS || dy >= MIN_MOVE_DIS) {
121.                    mPath.quadTo(preX, preY, (x + preX) / 2, (y + preY) / 2);
122.                    preX = x;
123.                    preY = y;
124.                }
125.                break;
126.        }
127.
128.        // 重绘视图
129.        invalidate();
130.        return true;
131.    }
132. }
```

运行效果如下：



啊啊啊啊啊啊啊啊啊啊~~~~~PorterDuffXfermode 算是暂告一段落，大家一定要学会  
PorterDuffXfermode 各个混合模式的使用，这是 android 图形绘制的重点之一！  
再次强调：PorterDuffXfermode 是重点~~~~一定要学会如何灵活使用

### 3. 自定义控件其实很简单(3)

上一回关羽操刀怒砍秦桧子龙拼命相救，岂料刘备这狗贼要赖以张飞为祭品特殊召唤黑暗大法师消灭了场上所有逗逼，霎时间血流成河，鲜红的血液与冰冷的大地融合交汇在一起焕发出血液的煞气.....那么，问题来了，请问这是使用了哪种 PorterDuffXfermode ?  
在上一节的最后一个 Example 中我们做了一个橡皮擦的 View，但是这个 View 虽然在效果上没有什么问题，但是逻辑确实有问题的！你们发现了么？哥故意挖了个坑让你们往里面跳哦！！！

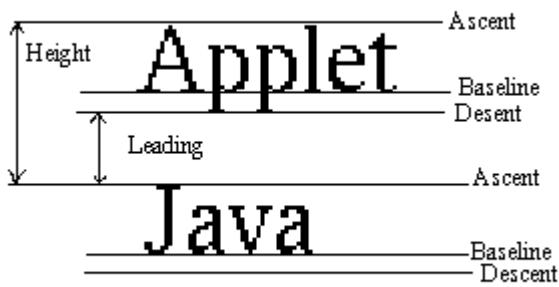
在对 `Xfermode` 和 `ColorFilter` 有了深情的了解后我们不能只爱上这俩二货，前方必定还有更多的好货色在等着我们开发……^\_~!今天我们继续向前看看 `Paint` 的其他一些“另类”的属性。笔对于我们来说第一印象一定是能写字对吧，而 `Android` 给我们的这支 `Paint` 当然也不例外，它也定义了大量关于“写字”的功能，这些方法总数接近 `Paint` 的一半！可见 `Android` 对 `Paint` 写字功能的重视，在讲 `Paint` 提供的“写字”方法前我先给大家说一个 `Android` 中和字体相关的很重要的类

### FontMetrics

`FontMetrics` 意为字体测量，这么一说大家是不是瞬间感受到了这玩意的重要性？那这东西有什么用呢？我们通过源码追踪进去可以看到 `FontMetrics` 其实是 `Paint` 的一个内部类，而它里面呢就定义了 `top,ascent,descent,bottom,leading` 五个成员变量其他什么也没有：

```
/*
 * Class that describes the various metrics for a font at a given text size.
 * Remember, Y values increase going down, so those values will be positive,
 * and values that measure distances going up will be negative. This class
 * is returned by getFontMetrics().
 */
public static class FontMetrics {
    /**
     * The maximum distance above the baseline for the tallest glyph in
     * the font at a given text size.
     */
    public float top;
    /**
     * The recommended distance above the baseline for singled spaced text.
     */
    public float ascent; //blog.csdn.net/aigestudio
    /**
     * The recommended distance below the baseline for singled spaced text.
     */
    public float descent;
    /**
     * The maximum distance below the baseline for the lowest glyph in
     * the font at a given text size.
     */
    public float bottom;
    /**
     * The recommended additional space to add between lines of text.
     */
    public float leading;
}
```

这五个成员变量除了 `top` 和 `bottom` 我们较熟悉外其余三个都很陌生是做什么用的呢？首先我给大家看张图：



这张图很简单但是也很扼要的说明了 top,ascent,descent,bottom,leading 这五个参数。首先我们要知道 **Baseline** 基线，在 Android 中，文字的绘制都是从 **Baseline** 处开始的，**Baseline** 往上至字符最高处的距离我们称之为 **ascent**（上坡度），**Baseline** 往下至字符最底处的距离我们称之为 **descent**（下坡度），而 **leading**（行间距）则表示上一行字符的 **descent** 到该行字符的 **ascent** 之间的距离，**top** 和 **bottom** 文档描述地很模糊，其实这里我们可以借鉴一下 **TextView** 对文本的绘制，**TextView** 在绘制文本的时候总会在文本的最外层留出一些内边距，为什么要这样做？因为 **TextView** 在绘制文本的时候考虑到了类似读音符号，可能大家很久没写过拼音了已经忘了什么叫读音符号了吧……下图中的 A 上面的符号就是一个拉丁文的类似读音符号的东西：



然而根据世界范围内已入案的使用语言中能够标注在字符上方或者下方的除了类似的符号肯定是数不胜数的……哥不是语言专家我母鸡啊……而 **top** 的意思其实就是除了 **Baseline** 到字符顶端的距离外还应该包含这些符号的高度，**bottom** 的意思也是一样，一般情况下我们极少使用到类似的符号所以往往会忽略掉这些符号的存在，但是 Android 依然会在绘制文本的时候在文本外层留出一定的边距，这就是为什么 **top** 和 **bottom** 总会比 **ascent** 和 **descent** 大一点的原因。而在 **TextView** 中我们可以通过 **xml** 设置其属性

`android:includeFontPadding="false"` 去掉一定的边距值但是不能完全去掉。下面我们在 **Canvas** 上绘制一段文本并尝试打印文本的 **top,ascent,descent,bottom** 和 **leading**:

[java] [view](#) [plain](#) [copy](#) [print](#)?

```

1. public class FontView extends View {
2.     private static final String TEXT = "ap 爱哥 ξτβ6ηω ㄭㄮ ㅓ ぬもト H@↓";
3.     private Paint mPaint;// 画笔

```

```
4.     private FontMetrics mFontMetrics;// 文本测量对象
5.
6.     public FontView(Context context) {
7.         this(context, null);
8.     }
9.
10.    public FontView(Context context, AttributeSet attrs) {
11.        super(context, attrs);
12.
13.        // 初始化画笔
14.        initPaint();
15.    }
16.
17.    /**
18.     * 初始化画笔
19.     */
20.    private void initPaint() {
21.        // 实例化画笔
22.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
23.        mPaint.setTextSize(50);
24.        mPaint.setColor(Color.BLACK);
25.
26.        mFontMetrics = mPaint.getFontMetrics();
27.
28.        Log.d("Aige", "ascent: " + mFontMetrics.ascent);
29.        Log.d("Aige", "top: " + mFontMetrics.top);
30.        Log.d("Aige", "leading: " + mFontMetrics.leading);
31.        Log.d("Aige", "descent: " + mFontMetrics.descent);
32.        Log.d("Aige", "bottom: " + mFontMetrics.bottom);
33.    }
34.
35.    @Override
36.    protected void onDraw(Canvas canvas) {
37.        super.onDraw(canvas);
38.        canvas.drawText(TEXT, 0, Math.abs(mFontMetrics.top), mPaint);
39.    }
40. }
```

logcat 输出如下：

```
Aige          ascent: -46.38672
Aige          top: -52.807617
Aige          leading: 0.0
Aige          descent: 12.207031
Aige          bottom: 13.549805
```

注：Baseline 上方的值为负，下方的值为正

如图我们得到了 top,ascent,descent,bottom 和 leading 的值，因为只有一行文本所以 leading 恒为 0，那么此时的显示效果是如何的呢？上面我们说到 Android 中文本的绘制是从 Baseline 开始的，在屏幕上的体现便是 Y 轴坐标，所以在

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. canvas.drawText(TEXT, 0, Math.abs(mFontMetrics.top), mPaint);
```

中我们将文本绘制的起点 Y 坐标向下移动 Math.abs(mFontMetrics.top) 个单位（注：mFontMetrics.top 是负数），相当于把文本的 Baseline 向下移动 Math.abs(mFontMetrics.top) 个单位，此时文本的顶部刚好会和屏幕顶部重合：



从代码中我们可以看到一个很特别的现象，在我们绘制文本之前我们便可以获取文本的 FontMetrics 属性值，也就是说我们 FontMetrics 的这些值跟我们要绘制什么文本是无关的，而仅与绘制文本 Paint 的 size 和 typeface 有关。我们来分别更改这两个值看看：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. mPaint.setTextSize(70);
```

```
Aige          ascent: -64.94141
Aige          top: -73.930664
Aige          leading: 0.0
Aige          descent: 17.089844
Aige          bottom: 18.969727
```

如图所示所有值都改变了，我们再为 Paint 设置一个 typeface：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. mPaint.setTypeface(Typeface.SERIF);
```

```
Aige          ascent: -74.819336
Aige          top: -73.34961
Aige          leading: 0.0
Aige          descent: 20.507813
Aige          bottom: 17.5
```

同样所有的值也改变了，那么我们知道这样的一个东西有什么用呢？如上所说文本的绘制是从 **Baseline** 开始，并且 **Baseline** 并非文本的分割线，当我们想让文本绘制的时候居中屏幕或其他的东西时就需要计算 **Baseline** 的 Y 轴坐标，比如我们让我们的文本居中画布：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1.  public class FontView extends View {
2.      private static final String TEXT = "ap 爱哥 ξτβ6ηω ㄭㄮ";
3.      private Paint textPaint, linePaint;// 文本的画笔和中心线的画笔
4.
5.      private int baseX, baseY;// Baseline 绘制的 XY 坐标
6.
7.      public FontView(Context context) {
8.          this(context, null);
9.      }
10.
11.     public FontView(Context context, AttributeSet attrs) {
12.         super(context, attrs);
13.
14.         // 初始化画笔
15.         initPaint();
16.     }
17.
18.     /**
19.      * 初始化画笔
20.      */
21.     private void initPaint() {
22.         // 实例化画笔
23.         textPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
24.         textPaint.setTextSize(70);
25.         textPaint.setColor(Color.BLACK);
26.
27.         linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
28.         linePaint.setStyle(Paint.Style.STROKE);
29.         linePaint.setStrokeWidth(1);
30.         linePaint.setColor(Color.RED);
31.     }
32.
33.     @Override
34.     protected void onDraw(Canvas canvas) {
```

```
35.         super.onDraw(canvas);
36.
37.         // 计算 Baseline 绘制的起点 X 轴坐标
38.         baseX = (int) (canvas.getWidth() / 2 - textPaint.measureText(TEXT) /
39.             2);
40.
41.         // 计算 Baseline 绘制的 Y 坐标
42.         baseY = (int) ((canvas.getHeight() / 2) - ((textPaint.descent() + te
43.             xtPaint.ascent()) / 2));
44.
45.         // 为了便于理解我们在画布中心处绘制一条中线
46.         canvas.drawLine(0, canvas.getHeight() / 2, canvas.getWidth(), canvas
47.             .getHeight() / 2, linePaint);
48.     }
```

效果如图：



**Baseline** 绘制的起点 x 坐标为画布宽度的一半（中点 x 坐标）减去文本宽度的一半（这里我们的画布大小与屏幕大小一样），这个很好理解，而 y 坐标为画布高度的一半（中点 y 坐标）减去 **ascent** 和 **descent** 绝对值之差的一半，这一点很多朋友可能不是很好理解，其实很简单，如果直接以画布的中心为 **Baseline**：

[java] [view plain](#) [copy](#) [print?](#)

```
1. baseY = canvas.getHeight() / 2;
```

那么画出来的效果必定是如下的样子



也就是说 **Baseline** 和屏幕中线重合，而这样子绘制出来的文本必定不在屏幕中心，因为 **ascent** 的距离大于 **descent** 的距离（大多数情况下我们没有考虑 **top** 和 **bottom**），那么我们就需要将 **Baseline** 往下移使绘制出来的文本能在中心



那么该下移多少呢？这是一个问题，很多童鞋的第一反应是下移 `ascent` 的一半高度，但是你要考虑到已经在中线下方的 `descent` 的高度，所以我们应该先在 `ascent` 的高度中减去 `descent` 的高度再除以二再让屏幕的中点 Y 坐标（也就是高度的一半）加上这个偏移值

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. baseY = (int) ((canvas.getHeight() / 2) + ((Math.abs(textPaint.ascent())-Math.abs(textPaint.descent())))) / 2);
```

这个公式跟我们上面代码中的是一样的，不信大家可以自己算算这里就不多说了。这里我们的需求是让文本绘制在某个区域的中心，实际情况中有很多不同的需求不如靠近某个区域离某个区域需要多少距离等等，熟练地去学会计算文本测绘中的各个值就显得很有必要了！

`Paint` 有一个唯一的子类 `TextPaint` 就是专门为文本绘制量身定做的“笔”，而这支笔就如 API 所描述的那样能够在绘制时为文本添加一些额外的信息，这些信息包括：

`baselineShift, bgColor, density, drawableState, linkColor`，这些属性都很简单大家顾名思义或者自己去尝试下即可这里就不多说了，那么这支笔有何用呢？最常用的用法是在绘制文本时能够实现换行绘制！在正常情况下 Android 绘制文本是不能识别换行符之类的标识符的，这时候如果我们想实现换行绘制就得另辟途径使用 `StaticLayout` 结合 `TextPaint` 实现换行，

StaticLayout 是 android.text.Layout 的一个子类，很明显它也是为文本处理量身定做的，其内部实现了文本绘制换行的处理，该类不是本系列重点我们不再多说直接 Look 一下它是如何实现换行的：

[java] view plaincopyprint?

```
1.  public class StaticLayoutView extends View {
2.      private static final String TEXT = "This is used by widgets to control text layout. You should not need to use this class directly unless you are implementing your own widget or custom display object, or would be tempted to call Canvas.drawText() directly.";
3.      private TextPaint mTextPaint;// 文本的画笔
4.      private StaticLayout mStaticLayout;// 文本布局
5.
6.      public StaticLayoutView(Context context) {
7.          this(context, null);
8.      }
9.
10.     public StaticLayoutView(Context context, AttributeSet attrs) {
11.         super(context, attrs);
12.
13.         // 初始化画笔
14.         initPaint();
15.     }
16.
17.     /**
18.      * 初始化画笔
19.      */
20.     private void initPaint() {
21.         // 实例化画笔
22.         mTextPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG);
23.         mTextPaint.setTextSize(50);
24.         mTextPaint.setColor(Color.BLACK);
25.     }
26.
27.     @Override
28.     protected void onDraw(Canvas canvas) {
29.         super.onDraw(canvas);
30.         mStaticLayout = new StaticLayout(TEXT, mTextPaint, canvas.getWidth(),
31.             Alignment.ALIGN_NORMAL, 1.0F, 0.0F, false);
32.         mStaticLayout.draw(canvas);
33.         canvas.restore();
34.     }
```

运行效果如下：

This is used by widgets to control text layout.  
You should not need to use this class directly  
unless you are implementing your own widget or  
custom display object, or would be tempted to  
call `Canvas.drawText()` directly.

<http://blog.csdn.net/aigestudio>

好了，对 `Paint` 绘制文本的一个简单了解就先到这，我们来看看 `Paint` 中到底提供了哪些实用的方法来绘制文本

#### **ascent()**

顾名思义就是返回上坡度的值，我们已经用过了

#### **descent()**

同上，不多说了

### **breakText (CharSequence text, int start, int end, boolean measureForwards, float maxWidth, float[] measuredWidth)**

这个方法让我们设置一个最大宽度在不超过这个宽度的范围内返回实际测量值否则停止测量，参数很多但是都很好理解，`text` 表示我们的字符串，`start` 表示从第几个字符串开始测量，`end` 表示从测量到第几个字符串为止，`measureForwards` 表示向前还是向后测量，`maxWidth` 表示一个给定的最大宽度在这个宽度内能测量出几个字符，`measuredWidth` 为一个可选项，可以为空，不为空时返回真实的测量值。同样的方法还有 `breakText (String text,`

boolean measureForwards, float maxWidth, float[] measuredWidth) 和 breakText (char[] text, int index, int count, float maxWidth, float[] measuredWidth)。这些方法在一些结合文本处理的应用里比较常用，比如文本阅读器的翻页效果，我们需要在翻页的时候动态折断或生成一行字符串，这就派上用场了~~~

#### **getFontMetrics (Paint.FontMetrics metrics)**

这个和我们之前用到的 getFontMetrics() 相比多了个参数，getFontMetrics() 返回的是 FontMetrics 对象而 getFontMetrics(Paint.FontMetrics metrics) 返回的是文本的行间距，如果 metrics 的值不为空则返回 FontMetrics 对象的值。

#### **getFontMetricsInt()**

该方法返回了一个 FontMetricsInt 对象，FontMetricsInt 和 FontMetrics 是一样的，只不过 FontMetricsInt 返回的是 int 而 FontMetrics 返回的是 float

#### **getFontMetricsInt(Paint.FontMetricsInt fmi)**

不扯了

#### **getFontSpacing()**

返回字符行间距

#### **setUnderlineText(boolean underlineText)**

设置下划线

#### **setTypeface(Typeface typeface)**

设置字体类型，上面我们也使用过，Android 中字体有四种样式：BOLD（加粗），BOLD\_ITALIC（加粗并倾斜），ITALIC（倾斜），NORMAL（正常）；而其为我们提供的字体有五种：DEFAULT，DEFAULT\_BOLD，MONOSPACE，SANS\_SERIF 和 SERIF，这些什么类型啊、字体啊之类的都很简单大家自己去试试就知道就不多说了。但是系统给我们的字体有限我们可不可以使用自己的字体呢？答案是肯定的！Typeface 这个类中给我们提供了多个方法去个性化我们的字体

#### **defaultFromStyle(int style)**

最简单的，简而言之就是把上面所说的四种 Style 封装成 Typeface

#### **create(String familyName, int style) 和 create(Typeface family, int style)**

两者大概意思都一样，比如

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. textPaint.setTypeface(Typeface.create("SERIF", Typeface.NORMAL));  
2. textPaint.setTypeface(Typeface.create(Typeface.SERIF, Typeface.NORMAL));
```

两者效果是一样的

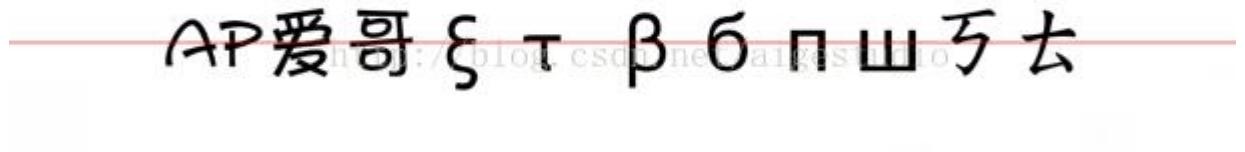
**createFromAsset(AssetManager mgr, String path)**、**createFromFile(String path)**和  
**createFromFile(File path)**

这三者也是一样的，它们都允许我们使用自己的字体比如我们从 `asset` 目录读取一个字体文件：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 获取字体并设置画笔字体
2. Typeface typeface = Typeface.createFromAsset(context.getAssets(), "kt.ttf");
3. textPaint.setTypeface(typeface);
```

我们将会得到如下效果：



这里我用了一个卡通的字体，而另外两个方法也类似的我就不讲了。

说到文本大家第一时间想到的应该是 `TextView`，其实在 `TextView` 里我们依然可以找到上面很多方法的影子，比如我们可以从 `TextView` 中获取到 `TextPaint`:

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. TextPaint paint = mTextView.getPaint();
```

当然也可以设置 `TextView` 的字体等等：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. Typeface typeface = Typeface.createFromAsset(getAssets(), "kt.ttf");
2. mTextView.setTypeface(typeface);
```

更多的雷同点还是留给大家去发掘，下面继续来看

**setTextSkewX(float skewX)**

这个方法可以设置文本在水平方向上的倾斜，效果类似下图：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 设置画笔文本倾斜  
2. textPaint.setTextSkewX(-0.25F);
```

ap爱哥xi β6пш5去

这个倾斜值没有具体的范围，但是官方推崇的值为-0.25 可以得到比较好的倾斜文本效果，  
值为负右倾为正左倾， 默认值为 0

#### **setTextSize (float textSize)**

不说了但是要注意该值必需大于零

#### **setTextScaleX (float scaleX)**

将文本沿 X 轴水平缩放， 默认值为 1， 当值大于 1 会沿 X 轴水平放大文本， 当值小于 1 会  
沿 X 轴水平缩放文本

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 设置画笔文本倾斜  
2. textPaint.setTextScaleX(0.5F);
```

ap爱哥xi β6пш5去

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 设置画笔文本倾斜  
2. textPaint.setTextScaleX(1.5F);
```

ap爱哥xi β6пш5去

大家注意哦！ `setTextScaleX` 不仅放大了文本宽度同时还拉伸了字符！这是亮点~~

#### **setTextLocale (Locale locale)**

设置地理位置，这个不讲，我们会在屏幕适配系列详解什么是 `Locale`，这里如果你要使用，直接传入 `Locale.getDefault()` 即可

### `setTextAlign (Paint.Align align)`

设置文本的对其方式，可供选的方式有三种：`CENTER`,`LEFT` 和 `RIGHT`，其实从这三者的名字上看我们就知道其意思，但是问题是这玩意怎么用的？好像没什么用啊……我们的文本大小是通过 `size` 和 `typeface` 确定的（其实还有其他的因素但这里影响不大忽略~~），一旦 `baseline` 确定，对不对齐好像不相干吧……但是，你要知道一点，文本的绘制是从 `baseline` 开始没错，但是是从哪边开始绘制的呢？左端还是右端呢？而这个 `Align` 就是为我们定义在 `baseline` 绘制文本究竟该从何处开始，上面我们在进行对文本的水平居中时是用 `Canvas` 宽度的一半减去文本宽度的一半：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      super.onDraw(canvas);
4.
5.      // 计算 Baseline 绘制的起点 X 轴坐标
6.      baseX = (int) (canvas.getWidth() / 2 - textPaint.measureText(TEXT) / 2);
7.
8.      // 计算 Baseline 绘制的 Y 坐标
9.      baseY = (int) ((canvas.getHeight() / 2) - ((textPaint.descent() + textPaint.ascent()) / 2));
10.
11.     canvas.drawText(TEXT, baseX, baseY, textPaint);
12.
13.     // 为了便于理解我们在画布中心处绘制一条中线
14.     canvas.drawLine(0, canvas.getHeight() / 2, canvas.getWidth(), canvas.getHeight() / 2, linePaint);
15. }
```

实际上我们大可不必这样计算，我们只需设置 `Paint` 的文本对齐方式为 `CENTER`, `drawText` 的时候起点 `x = canvas.getWidth() / 2` 即可：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1.  textPaint.setTextAlign(Align.CENTER);
2.  canvas.drawText(TEXT, canvas.getWidth() / 2, baseY, textPaint);
```

当我们将文本对齐方式设置为 `CENTER` 后就相当于告诉 `Android` 我们这个文本绘制的时候从文本的中点开始向两端绘制，如果设置为 `LEFT` 则从文本的左端开始往右绘制，如果为

RIGHT 则从文本的右端开始往左绘制：

爱哥ξτ βбпшㄅ去

ap爱哥ξτ βбпшㄅ去  
http://blog.ssdn.net/aigestudio

ap爱哥ξτ βбпшㄅ

### **setSubpixelText (boolean subpixelText)**

设置是否打开文本的亚像素显示，什么叫亚像素显示呢？你可以理解为对文本显示的一种优化技术，如果大家用的是 Win7+ 系统可以在控制面板中找到一个叫 ClearType 的设置，该设置可以让你的文本更好地显示在屏幕上就是基于亚像素显示技术。具体我们在设计色彩系列将会细说，这里就不扯了

### **setStrikeThruText (boolean strikeThruText)**

文本删除线，不扯

### **setLinearText (boolean linearText)**

设置是否打开线性文本标识，这玩意对大多数人来说都很奇怪不知道这玩意什么意思。想要明白这东西你要先知道文本在 Android 中是如何进行存储和计算的。在 Android 中文本的绘制需要使用一个 bitmap 作为单个字符的缓存，既然是缓存必定要使用一定的空间，我们可以通过 setLinearText (true) 告诉 Android 我们不需要这样的文本缓存。

### **setFakeBoldText (boolean fakeBoldText)**

设置文本仿粗体

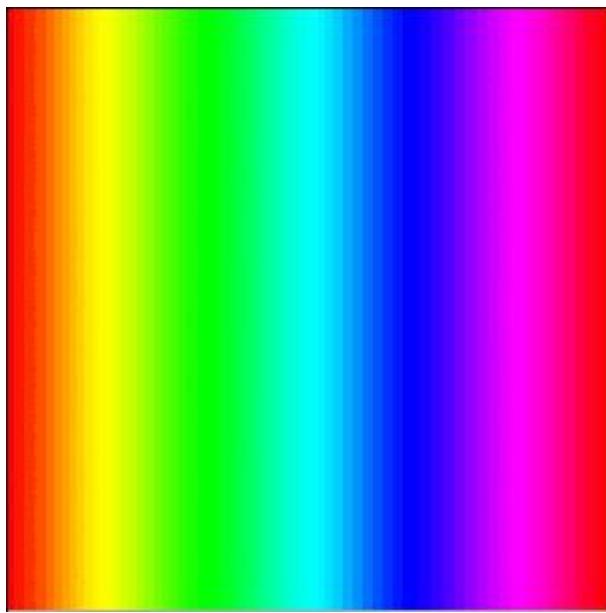
### **measureText (String text), measureText (CharSequence text, int start, int end), measureText (String text, int start, int end), measureText (char[] text, int index, int count)**

测量文本宽度，上面我们已经使用过了，这四个方法都是一样的只是参数稍有不同这里就不撤了！Paint 对文本的绘制方法就上面那些，API 21 中还新增了两个方法这里就先不讲了，大家可以看到虽然说这些方法很多很多但是效果都是显而易见的，很多方法大家一试就知道所以哥也没有做太多的测试之类什么什么的，这样讲东西是很累的，关于文本也没有什么有趣的 Demo 可以玩~~~~~so~~~~~Fuck.....

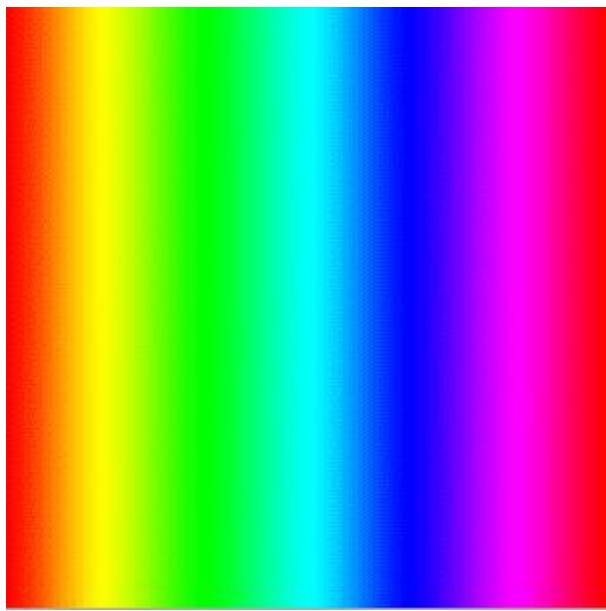
下面我们来看一个比较深奥的东西

### **setDither(boolean dither)**

这玩意用来设置我们在绘制图像时的抗抖动，也称为递色，那什么叫抗抖动呢？在 Android 中我确实不好拿出一个明显的例子，我就在 PS 里模拟说明一下



大家看到的这张七彩渐变图是一张 **RGB565** 模式下图片，即便图片不是很大我们依然可以很清晰地看到在两种颜色交接的地方有一些色块之类的东西感觉很不柔和，因为在 **RGB** 模式下只能显示  $2^{16}=65535$  种色彩，因此很多丰富的色彩变化无法呈现，而 **Android** 呢为我们提供了抗抖动这么一个方法，它会将相邻像素之间颜色值进行一种“中和”以呈现一个更细腻的过渡色：



放大来看，其在很多相邻像素之间插入了一个“中间值”：



抗抖动不是 Android 的专利，是图形图像领域的一种解决位图精度的技术。上面说了太多理论性的东西，估计大家都疲惫了，接下来我们来瞅瞅一个比较酷的东西 MaskFilter 遮罩过滤器！在 Paint 我们有个方法来设置这东西

### **setMaskFilter(MaskFilter maskfilter)**

MaskFilter 类中没有任何实现方法，而它有两个子类 BlurMaskFilter 和 EmbossMaskFilter，前者为模糊遮罩滤镜（比起称之为过滤器哥更喜欢称之为滤镜）而后者为浮雕遮罩滤镜，我们先来看第一个

#### **BlurMaskFilter**

Android 中的很多自带控件都有类似软阴影的效果，比如说 Button



它周围就有一圈很淡的阴影效果，这种效果看起来让控件更真实，那么是怎么做的呢？其实很简单，使用 BlurMaskFilter 就可以得到类似的效果

[java] view plain copy print?

```
1. public class MaskFilterView extends View {  
2.     private static final int RECT_SIZE = 800;  
3.     private Paint mPaint;// 画笔  
4.     private Context mContext;// 上下文环境引用
```

```
5.
6.     private int left, top, right, bottom;//
7.
8.     public MaskFilterView(Context context) {
9.         this(context, null);
10.    }
11.
12.    public MaskFilterView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.        mContext = context;
15.
16.        // 初始化画笔
17.        initPaint();
18.
19.        // 初始化资源
20.        initRes(context);
21.    }
22.
23.    /**
24.     * 初始化画笔
25.     */
26.    private void initPaint() {
27.        // 实例化画笔
28.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
29.        mPaint.setStyle(Paint.Style.FILL);
30.        mPaint.setColor(0xFF603811);
31.
32.        // 设置画笔遮罩滤镜
33.        mPaint.setMaskFilter(new BlurMaskFilter(20, BlurMaskFilter.Blur.SOLID));
34.    }
35.
36.    /**
37.     * 初始化资源
38.     */
39.    private void initRes(Context context) {
40.        /*
41.         * 计算位图绘制时左上角的坐标使其位于屏幕中心
42.         */
43.        left = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - RECT_SIZE / 2;
44.        top = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - RECT_SIZE / 2;
```

```
45.         right = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 + RECT
        _SIZE / 2;
46.         bottom = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 + REC
        T_SIZE / 2;
47.     }
48.
49.     @Override
50.     protected void onDraw(Canvas canvas) {
51.         super.onDraw(canvas);
52.         canvas.drawColor(Color.GRAY);
53.
54.         // 画一个矩形
55.         canvas.drawRect(left, top, right, bottom, mPaint);
56.     }
57. }
```

代码中我们在画布中央绘制了一个正方形，并设置了它的模糊滤镜，但是当你运行后发现并没有任何的效果：



为什么会这样呢？还记得上一节中我们讲的 `AvoidXfermode` 吗，在 API 16 的时候该类已经被标注为过时了，因为 `AvoidXfermode` 不支持硬件加速，如果在 API 16+ 上想获得正确的效果就必需关闭应用的硬件加速，当时我们是在 `AndroidManifest.xml` 文件中设置 `android:hardwareAccelerated` 为 `false` 来关闭的，具体有哪些绘制的方法不支持硬件加速可以参考下图

| Canvas                                              |   |    |
|-----------------------------------------------------|---|----|
| <code>drawBitmapMesh()</code> (colors array)        |   | 18 |
| <code>drawPicture()</code>                          | X |    |
| <code>drawPosText()</code>                          |   | 16 |
| <code>drawTextOnPath()</code>                       |   | 16 |
| <code>drawVertices()</code>                         | X |    |
| <code>setDrawFilter()</code>                        |   | 16 |
| <code>clipPath()</code>                             |   | 18 |
| <code>clipRegion()</code>                           |   | 18 |
| <code>clipRect(Region.Op.XOR)</code>                |   | 18 |
| <code>clipRect(Region.Op.Difference)</code>         |   | 18 |
| <code>clipRect(Region.Op.ReverseDifference)</code>  |   | 18 |
| <code>clipRect() with rotation/perspective</code>   |   | 18 |
| Paint                                               |   |    |
| <code>setAntiAlias() (for text)</code>              |   | 18 |
| <code>setAntiAlias() (for lines)</code>             |   | 16 |
| <code>setFilterBitmap()</code>                      |   | 17 |
| <code>setLinearText()</code>                        | X |    |
| <code>setMaskFilter()</code>                        | X |    |
| <code>setPathEffect() (for lines)</code>            | X |    |
| <code>setRasterizer()</code>                        | X |    |
| <code>setShadowLayer() (other than text)</code>     | X |    |
| <code>setStrokeCap() (for lines)</code>             |   | 18 |
| <code>setStrokeCap() (for points)</code>            |   | 19 |
| <code>setSubpixelText()</code>                      | X |    |
| Xfermode                                            |   |    |
| <code>AvoidXfermode</code>                          | X |    |
| <code>PixelXorXfermode</code>                       | X |    |
| <code>PorterDuff.Mode.DARKEN (framebuffer)</code>   | X |    |
| <code>PorterDuff.Mode.LIGHTEN (framebuffer)</code>  | X |    |
| <code>PorterDuff.Mode.OVERLAY (framebuffer)</code>  | X |    |
| Shader                                              |   |    |
| <code>ComposeShader inside ComposeShader</code>     | X |    |
| <code>Same type shaders inside ComposeShader</code> | X |    |
| <code>Local matrix on ComposeShader</code>          |   | 18 |

但是大家想过没如果在 `AndroidManifest.xml` 文件中关闭硬件加速那么我们整个应用都将不支持硬件加速，这显然是不科学的，如果可以只针对某个 `View` 关闭硬件加速那岂不是很好么？当然，Android 也给我们提供了这样的功能，我们可以在 `View` 中通过

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. setLayerType(LAYER_TYPE_SOFTWARE, null);
```

来关闭单个 View 的硬件加速功能

再次运行即可得到正确的效果：



<http://blog.csdn.net/aigestudio>

是不是很酷呢？BlurMaskFilter 只有一个含参的构造函数 BlurMaskFilter(float radius, BlurMaskFilter.Blur style)，其中 radius 很容易理解，值越大我们的阴影越扩散，比如在上面的例子中我将 radius 改为 50



<http://blog.csdn.net/aigestudio>

可以明显感到阴影的范围扩大了，这个很好理解。而第二个参数 `style` 表示的是模糊的类型，上面我们用到的是 `SOLID`，其效果就是在图像的 `Alpha` 边界外产生一层与 `Paint` 颜色一致的阴影效果而不影响图像本身，除了 `SOLID` 还有三种，`NORMAL, OUTER` 和 `INNER, NORMAL` 会将整个图像模糊掉：



<http://blog.csdn.net/aigestudio>

而 **OUTER** 会在 Alpha 边界外产生一层阴影且会将原本的图像变透明：



**INNER** 则会在图像内部产生模糊:



INNER 效果其实并不理想，实际应用中我们使用的也少，我们往往会使用混合模式和渐变和获得更完美的内阴影效果。如上所说 BlurMaskFilter 是根据 Alpha 通道的边界来计算模糊的，如果是一张图片(注：上面我们说过 Android 会把拷贝到资源目录的图片转为 RGB565，具体原因具体分析我会单独开一篇帖子说，这里就先假设所有提及的图片格式为 RGB565)你会发现没有任何效果，那么假使我们需要给图片加一个类似阴影的效果该如何做呢？其实很简单，我们可以尝试从 Bitmap 中获取其 Alpha 通道，并在绘制 Bitmap 前先以该 Alpha 通道绘制一个模糊效果不就行了？

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class BlurMaskFilterView extends View {  
2.     private Paint shadowPaint; // 画笔  
3.     private Context mContext; // 上下文环境引用  
4.     private Bitmap srcBitmap, shadowBitmap; // 位图和阴影位图  
5.  
6.     private int x, y; // 位图绘制时左上角的起点坐标  
7.  
8.     public BlurMaskFilterView(Context context) {  
9.         this(context, null);
```

```
10.    }
11.
12.    public BlurMaskFilterView(Context context, AttributeSet attrs) {
13.        super(context, attrs);
14.        mContext = context;
15.        // 记得设置模式为 SOFTWARE
16.        setLayerType(LAYER_TYPE_SOFTWARE, null);
17.
18.        // 初始化画笔
19.        initPaint();
20.
21.        // 初始化资源
22.        initRes(context);
23.    }
24.
25.    /**
26.     * 初始化画笔
27.     */
28.    private void initPaint() {
29.        // 实例化画笔
30.        shadowPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);

31.        shadowPaint.setColor(Color.DKGRAY);
32.        shadowPaint.setMaskFilter(new BlurMaskFilter(10, BlurMaskFilter.Blur
33.            .NORMAL));
34.    }
35.
36.    /**
37.     * 初始化资源
38.     */
39.    private void initRes(Context context) {
40.        // 获取位图
41.        srcBitmap = BitmapFactory.decodeResource(context.getResources(), R.d
42.            rawable.a);
43.
44.        // 获取位图的 Alpha 通道图
45.        shadowBitmap = srcBitmap.extractAlpha();
46.
47.        /*
48.         * 计算位图绘制时左上角的坐标使其位于屏幕中心
49.         */
50.        x = MeasureUtil.getScreenSize((Activity) mContext)[0] / 2 - srcBitma
51.        p.getWidth() / 2;
```

```
49.         y = MeasureUtil.getScreenSize((Activity) mContext)[1] / 2 - srcBitmap.getHeight() / 2;
50.     }
51.
52.     @Override
53.     protected void onDraw(Canvas canvas) {
54.         super.onDraw(canvas);
55.         // 先绘制阴影
56.         canvas.drawBitmap(shadowBitmap, x, y, shadowPaint);
57.
58.         // 再绘制位图
59.         canvas.drawBitmap(srcBitmap, x, y, null);
60.     }
61. }
```

如代码所示我们通过 `Bitmap` 的 `extractAlpha()` 方法从原图中分离出一个 Alpha 通道位图并在计算模糊滤镜的时候使用该位图生成模糊效果:



相对于 `BlurMaskFilter` 来说

### **EmbossMaskFilter**

的常用性比较低，倒不是说 `EmbossMaskFilter` 很没用，只是相对于 `EmbossMaskFilter` 实现的效果来说远不及 `BlurMaskFilter` 给人的感觉霸气，说了半天那么 `EmbossMaskFilter` 到底是做什么的呢？

我们先来看一张图：



<http://blog.csdn.net/aigestudio>

这么一个看着像巧克力的东西就是用 `EmbossMaskFilter` 实现了，正如其名，他可以实现一种类似浮雕的效果，说白了就是让你绘制的图像感觉像是从屏幕中“凸”起来更有立体感一样（在设计软件中类似的效果称之为斜面浮雕）。该类也只有一个含参的构造方法

`EmbossMaskFilter(float[] direction, float ambient, float specular, float blurRadius)`，这些参数理解起来要比 `BlurMaskFilter` 困难得多，如果你没有空间想象力的话，首先我们来看第一个 `direction` 指的是方向，什么方向呢？光照的方向！如果大家接触过三维设计就一定懂，没接触也没关系，我跟你说明白。假设一个没有任何光线的黑屋子里有一张桌子，桌子上有一个小球，这时我们打开桌子上的台灯，台灯照亮了小球，这时候小球的状态与下图类似：



PS: 略草, 凑合看

小球最接近光源的地方肯定是最亮的这个没有异议,在参数中 **specular** 就是跟高光有关的,其值是个双向值越小或越大高光越强中间值则是最弱的, 那么再看看什么是 **ambient** 呢? 同样我们看个球, 你会发现即便只有一盏灯光, 在球底部跟桌面相接的地方依然不会出现大片的“死黑”, 这是因为光线在传播的过程中碰到物体会产生反射! 这种反射按照物体介质的粗糙度可以分为漫反射和镜面反射, 而这里我们的小球之所以背面没有直接光照但仍能有一定的亮度就是因为大量的漫反射在空间传播让光线间接照射到小球背面, 这种区别于直接照明的二次照明我们称之为间接照明, 产生的光线叫做环境光 **ambient**, 参数中的该值就是用来设置环境光的, 在 **Android** 中环境光默认为白色, 其值越大, 阴影越浅, **blurRadius** 则是设置图像究竟“凸”出多大距离的很好理解, 最难理解的一个参数是 **direction**, 上面我们也说了是光照方向的意思, 该数组必须要有而且只能有三个值即 **float[x,y,z]**, 这三个值代表了一个空间坐标系, 我们的光照方向则由其定义, 那么它是怎么定义的呢? 首先 **x** 和 **y** 很好理解, 平面的两个维度嘛是吧, 上面我们使用的是 **[1,1]** 也就是个 45 度角, 而 **z** 轴表示光源是在屏幕后方还是屏幕前方, 上面我们是用的是 **1**, 正值表示光源往屏幕外偏移 **1** 个单位, 负值表示往屏幕里面偏移, 这么一说如果我把其值改为 **[1,1,-1]** 那么我们的巧克力朝着我们的一面应该就看不到了对吧, 试试看撒~~~这个效果我就不截图了, 因为一片漆黑.....但是你依然能够看到一点点灰度~就是因为我们的环境光 **ambient!** , 如果我们把值改为 **[1,1,2]** 往屏幕外偏移两个单位, 那么我们巧克力正面光照将更强:



<http://blog.csdn.net/aigestudio>

看吧都爆色了！这里要提醒一点[x,y,z]表示的是空间坐标，代表光源的位置，那么一旦这个位置确定，[ax,ay,az]则没有意义，也就是说同时扩大三个轴向值的倍数是没有意义的，最终效果还是跟[x,y,z]一样！懂了不？

额.....忘了给代码，大家可以自己去试试

[java] view plain copy print?

```
1. public class EmbossMaskFilterView extends View {  
2.     private static final int H_COUNT = 2, V_COUNT = 4; // 水平和垂直切割数  
3.     private Paint mPaint; // 画笔  
4.     private PointF[] mPointFs; // 存储各个巧克力坐上坐标的点  
5.  
6.     private int width, height; // 单个巧克力宽高  
7.     private float coorY; // 单个巧克力坐上 Y 轴坐标值  
8.  
9.     public EmbossMaskFilterView(Context context) {  
10.         this(context, null);  
11.     }  
12. }
```

```
13.     public EmbossMaskFilterView(Context context, AttributeSet attrs) {
14.         super(context, attrs);
15.         // 不使用硬件加速
16.         setLayerType(LAYER_TYPE_SOFTWARE, null);
17.
18.         // 初始化画笔
19.         initPaint();
20.
21.         // 计算参数
22.         cal(context);
23.     }
24.
25.     /**
26.      * 初始化画笔
27.      */
28.     private void initPaint() {
29.         // 实例化画笔
30.         mPaint = new Paint();
31.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
32.         mPaint.setStyle(Paint.Style.FILL);
33.         mPaint.setColor(0xFF603811);
34.
35.         // 设置画笔遮罩滤镜
36.         mPaint.setMaskFilter(new EmbossMaskFilter(new float[] { 1, 1, 1F },
37.             0.1F, 10F, 20F));
38.     }
39.
40.     /**
41.      * 计算参数
42.      */
43.     private void cal(Context context) {
44.         int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
45.
46.         width = screenSize[0] / H_COUNT;
47.         height = screenSize[1] / V_COUNT;
48.
49.         int count = V_COUNT * H_COUNT;
50.
51.         mPointFs = new PointF[count];
52.         for (int i = 0; i < count; i++) {
53.             if (i % 2 == 0) {
54.                 coorY = i * height / 2F;
55.                 mPointFs[i] = new PointF(0, coorY);
56.             } else {
```

```

56.             mPointFs[i] = new PointF(width, coorY);
57.         }
58.     }
59. }
60.
61. @Override
62. protected void onDraw(Canvas canvas) {
63.     super.onDraw(canvas);
64.     canvas.drawColor(Color.GRAY);
65.
66.     // 画矩形
67.     for (int i = 0; i < V_COUNT * H_COUNT; i++) {
68.         canvas.drawRect(mPointFs[i].x, mPointFs[i].y, mPointFs[i].x + width,
69.                         mPointFs[i].y + height, mPaint);
70.     }
71. }

```

上面我们说了 EmbossMaskFilter 的使用面并不是很大，因为虽说其参数稍复杂但是其实现原理是简单粗暴的，简而言之就是根据参数在图像周围绘制一个“色带”来模拟浮雕的效果，如果我们的图像很复杂 EmbossMaskFilter 很难会正确模拟，所以一般遇到这类图直接 call 美工 == 哈哈哈。

### **setRasterizer (Rasterizer rasterizer)**

设置光栅，光栅这东西涉及太多太多物理知识，不讲了一讲又是一大堆，而且该方法同样不支持 HW 在 API 21 中遗弃了~~~我们还是来看看对我们来说更好玩有趣的方法

### **setPathEffect(PathEffect effect)**

PathEffect 见文知意很明显就是路径效果的意思~~那这玩意肯定跟路径 Path 有关咯？那是必须的撒！PathEffect 跟上面的很多类一样没有具体的实现，但是其有六个子类：

public class

## PathEffect

extends Object

java.lang.Object

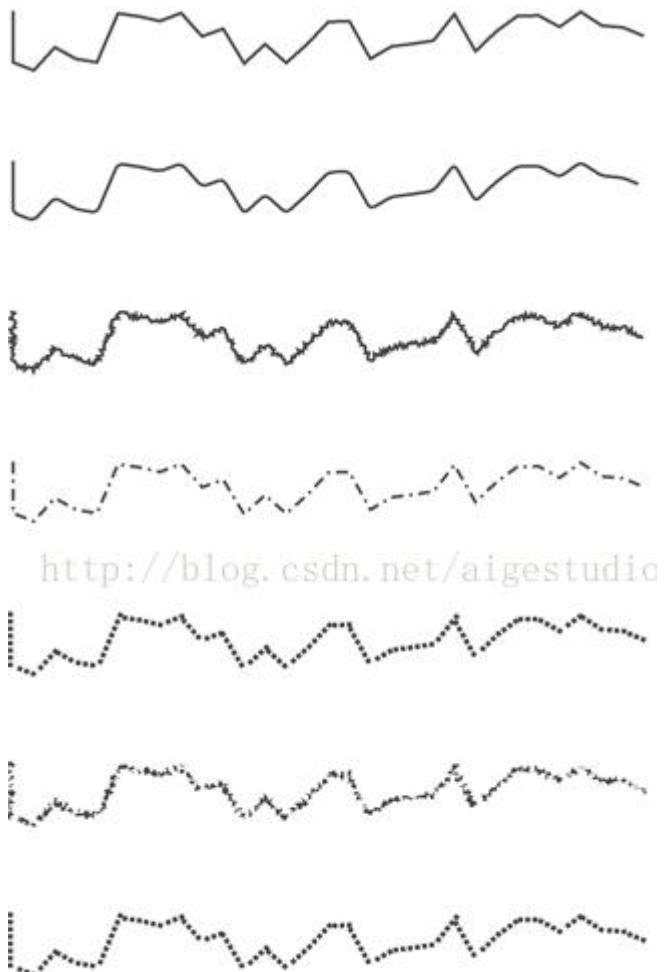
↳ android.graphics.PathEffect

<http://blog.csdn.net/aigestudio>

► Known Direct Subclasses

ComposePathEffect, CornerPathEffect, DashPathEffect, DiscretePathEffect, PathDashPathEffect, SumPathEffect

这六个子类分别可以实现不同的路径效果：



上图从上往下分别是没有 PathEffect、CornerPathEffect、DiscretePathEffect、DashPathEffect、PathDashPathEffect、ComposePathEffect、SumPathEffect 的效果，代码的实现也非常简单：

[java] view plain copy print?

```
1. public class PathEffectView extends View {  
2.     private float mPhase;// 偏移值  
3.     private Paint mPaint;// 画笔对象  
4.     private Path mPath;// 路径对象  
5.     private PathEffect[] mEffects;// 路径效果数组  
6.  
7.     public PathEffectView(Context context, AttributeSet attrs) {  
8.         super(context, attrs);  
9.  
10.        /*  
11.             * 实例化画笔并设置属性  
12.             */  
13.        mPaint = new Paint();
```

```
14.         mPaint.setStyle(Paint.Style.STROKE);
15.         mPaint.setStrokeWidth(5);
16.         mPaint.setColor(Color.DKGRAY);
17.
18.         // 实例化路径
19.         mPath = new Path();
20.
21.         // 定义路径的起点
22.         mPath.moveTo(0, 0);
23.
24.         // 定义路径的各个点
25.         for (int i = 0; i <= 30; i++) {
26.             mPath.lineTo(i * 35, (float) (Math.random() * 100));
27.         }
28.
29.         // 创建路径效果数组
30.         mEffects = new PathEffect[7];
31.     }
32.
33.     @Override
34.     protected void onDraw(Canvas canvas) {
35.         super.onDraw(canvas);
36.
37.         /*
38.          * 实例化各类特效
39.          */
40.         mEffects[0] = null;
41.         mEffects[1] = new CornerPathEffect(10);
42.         mEffects[2] = new DiscretePathEffect(3.0F, 5.0F);
43.         mEffects[3] = new DashPathEffect(new float[] { 20, 10, 5, 10 }, mPhase);
44.         Path path = new Path();
45.         path.addRect(0, 0, 8, 8, Path.Direction.CCW);
46.         mEffects[4] = new PathDashPathEffect(path, 12, mPhase, PathDashPathEffect.Style.ROTATE);
47.         mEffects[5] = new ComposePathEffect(mEffects[2], mEffects[4]);
48.         mEffects[6] = new SumPathEffect(mEffects[4], mEffects[3]);
49.
50.         /*
51.          * 绘制路径
52.          */
53.         for (int i = 0; i < mEffects.length; i++) {
54.             mPaint.setPathEffect(mEffects[i]);
55.             canvas.drawPath(mPath, mPaint);
```

```
56.  
57.          // 每绘制一条将画布向下平移 250 个像素  
58.          canvas.translate(0, 250);  
59.      }  
60.  
61.      // 刷新偏移值并重绘视图实现动画效果  
62.      mPhase += 1;  
63.      invalidate();  
64.  }  
65. }
```

当我们不设置路径效果的时候路径的默认效果就如上图第一条线那样直的转折生硬；而 **CornerPathEffect** 则可以将路径的转角变得圆滑如图第二条线的效果，这六种路径效果类都有且只有一个含参的构造方法，**CornerPathEffect** 的构造方法只接受一个参数 **radius**，意思就是转角处的圆滑程度，我们尝试更改一下上面的代码：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. mEffects[1] = new CornerPathEffect(50);
```



Look Pic 是不是更平滑了呢？**CornerPathEffect** 相对于其他的路径效果来说最简单了；**DiscretePathEffect** 离散路径效果相对来说则稍微复杂点，其会在路径上绘制很多“杂点”的突出来模拟一种类似生锈铁丝的效果如上图第三条线，其构造方法有两个参数，第一个呢指定这些突出的“杂点”的密度，值越小杂点越密集，第二个参数呢则是“杂点”突出的大小，值越大突出的距离越大反之反之，大家可以去自己去试下我就不演示了；**DashPathEffect** 的效果相对与上面两种路径效果来说要略显复杂，其虽说也是包含了两个参数，但是第一个参数是一个浮点型的数组，那这个数组有什么意义呢？其实是这样的，我们在定义该参数的时候只要浮点型数组中元素个数大于等于 2 即可，也就是说上面我们的代码可以写成这样的：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. mEffects[3] = new DashPathEffect(new float[] {20, 10}, mPhase);
```



从图中我们可以看到我们之前的那种线条变成了一长一短的间隔线条，而 `float[] {20, 10}` 的偶数参数 20（注意数组下标是从 0 开始哦）定义了我们第一条实线的长度，而奇数参数 10 则表示第一条虚线的长度，如果此时数组后面不再有数据则重复第一个数以此往复循环，比如我们 20,10 后没数了，那么整条线就成了[20,10,20,10,20,10,.....]这么一个状态，当然如果你无聊，也可以：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. mEffects[3] = new DashPathEffect(new float[] {20, 10, 50, 5, 100, 30, 10, 5},  
, mPhase);
```



而 `DashPathEffect` 的第二个参数我称之为偏移值，动态改变其值会让路径产生动画的效果，上面代码已给出大家可以自己去试试；`PathDashPathEffect` 和 `DashPathEffect` 是类似的，不同的是 `PathDashPathEffect` 可以让我们自己定义路径虚线的样式，比如我们将其换成一个个小圆组成的虚线：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. Path path = new Path();  
2. path.addCircle(0, 0, 3, Direction.CCW);  
3. mEffects[4] = new PathDashPathEffect(path, 12, mPhase, PathDashPathEffect.Style.ROTATE);
```

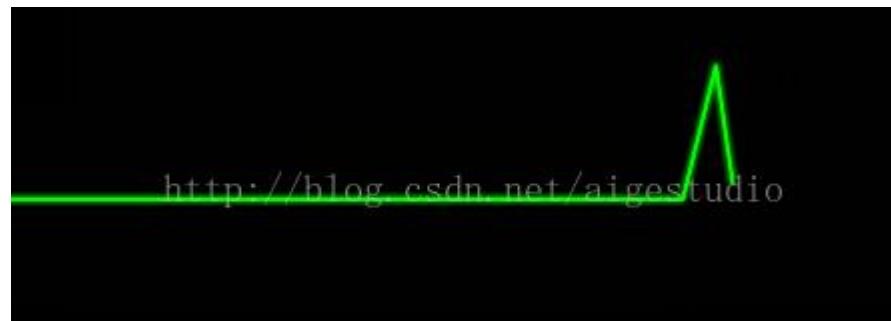


`ComposePathEffect` 和 `SumPathEffect` 都可以用来组合两种路径效果，唯一不同的是组合的方式，`ComposePathEffect(PathEffect outerpe, PathEffect innerpe)` 会先将路径变成 `innerpe` 的效果，再去复合 `outerpe` 的路径效果，即：`outerpe(innerpe(Path))`；而 `SumPathEffect(PathEffect first, PathEffect second)` 则会把两种路径效果加起来再作用于路径，具体区别大家去试试吧.....哥累了睡会~~~囧.....

记得在 1/12 中我们绘制了一个圆环并让其实现动画的效果，当时我们使用了线程来使其产生动画，但是我们是不是也可以像上面的例子一样直接在 `onDraw` 中 `invalidate()` 来产生动画呢？这个问题留给大家。

在 1/12 中我们还说过尽量不要在 `onDraw` 中使用 `new` 关键字来生成对象，但是上例的代码中我们却在频繁地使用，但是六个 `PathEffect` 的子类中除了构造方法什么都没有，我们该如何避免频繁地去 `new` 对象呢？这个问题也留给大家思考。

`Path` 应用的广泛性注定了 `PathEffect` 应用的广泛，所谓一人得道鸡犬升天就是这么个道理，只要是 `Path` 能存在的地方都可以考虑使用，下面我们来模拟一个类似心电图的路径小动画：



这种效果呢也是非常非常地简单，说白了就是无数条短小精悍的小“`Path`”连接成一条完整的心电路径：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. public class ECGView extends View {  
2.     private Paint mPaint;// 画笔  
3.     private Path mPath;// 路径对象  
4.  
5.     private int screenW, screenH;// 屏幕宽高  
6.     private float x, y;// 路径初始坐标  
7.     private float initScreenW;// 屏幕初始宽度  
8.     private float initX;// 初始 X 轴坐标  
9.     private float transX, moveX;// 画布移动的距离  
10.  
11.    private boolean isCanvasMove;// 画布是否需要平移  
12.  
13.    public ECGView(Context context, AttributeSet set) {  
14.        super(context, set);  
15.  
16.        /*  
17.         * 实例化画笔并设置属性  
18.         */  
19.    }  
20.    @Override  
21.    protected void onDraw(Canvas canvas) {  
22.        mPaint.setPathEffect(new DashPathEffect(new float[]{  
23.                10, 10}, 0));  
24.        mPaint.setStyle(Paint.Style.FILL);  
25.        mPaint.setColor(0xffff0000);  
26.        mPath.moveTo(initX, initScreenW);  
27.        mPath.lineTo(transX, moveX);  
28.        canvas.drawPath(mPath, mPaint);  
29.    }  
30.}
```

```
19.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
20.         mPaint.setColor(Color.GREEN);
21.         mPaint.setStrokeWidth(5);
22.         mPaint.setStrokeCap(Paint.Cap.ROUND);
23.         mPaint.setStrokeJoin(Paint.Join.ROUND);
24.         mPaint.setStyle(Paint.Style.STROKE);
25.         mPaint.setShadowLayer(7, 0, 0, Color.GREEN);
26.
27.         mPath = new Path();
28.         transX = 0;
29.         isCanvasMove = false;
30.     }
31.
32.     @Override
33.     public void onSizeChanged(int w, int h, int oldw, int oldh) {
34.         /*
35.          * 获取屏幕宽高
36.          */
37.         screenW = w;
38.         screenH = h;
39.
40.         /*
41.          * 设置起点坐标
42.          */
43.         x = 0;
44.         y = (screenH / 2) + (screenH / 4) + (screenH / 10);
45.
46.         // 屏幕初始宽度
47.         initScreenW = screenW;
48.
49.         // 初始 X 轴坐标
50.         initX = ((screenW / 2) + (screenW / 4));
51.
52.         moveX = (screenW / 24);
53.
54.         mPath.moveTo(x, y);
55.     }
56.
57.     @Override
58.     public void onDraw(Canvas canvas) {
59.         canvas.drawColor(Color.BLACK);
60.
61.         mPath.lineTo(x, y);
62.
```

```
63.         // 向左平移画布
64.         canvas.translate(-transX, 0);
65.
66.         // 计算坐标
67.         calCoors();
68.
69.         // 绘制路径
70.         canvas.drawPath(mPath, mPaint);
71.         invalidate();
72.     }
73.
74.     /**
75.      * 计算坐标
76.      */
77.     private void calCoors() {
78.         if (isCanvasMove == true) {
79.             transX += 4;
80.         }
81.
82.         if (x < initX) {
83.             x += 8;
84.         } else {
85.             if (x < initX + moveX) {
86.                 x += 2;
87.                 y -= 8;
88.             } else {
89.                 if (x < initX + (moveX * 2)) {
90.                     x += 2;
91.                     y += 14;
92.                 } else {
93.                     if (x < initX + (moveX * 3)) {
94.                         x += 2;
95.                         y -= 12;
96.                     } else {
97.                         if (x < initX + (moveX * 4)) {
98.                             x += 2;
99.                             y += 6;
100.                        } else {
101.                            if (x < initScreenW) {
102.                                x += 8;
103.                            } else {
104.                                isCanvasMove = true;
105.                                initX = initX + initScreenW;
106.                            }
107.                        }
108.                    }
109.                }
110.            }
111.        }
112.    }
113.
```

```
107.             }
108.         }
109.     }
110. }
111.
112.     }
113. }
114. }
```

我们在 `onSizeChanged(int w, int h, int oldw, int oldh)` 方法中获取屏幕的宽高，该方法的具体用法我们会在 7/12 学习 View 的测绘时具体说明，这里就先不说了

上面在设置 Paint 属性的时候我们使用到了一个

### **setStrokeCap(Paint.Cap cap)**

方法，该方法用来设置我们画笔的笔触风格，上面的例子中我使用的是 `ROUND`，表示是圆角的笔触，那么什么叫笔触呢，其实很简单，就像我们现实世界中的笔，如果你用圆珠笔在纸上戳一点，那么这个点一定是个圆，即便很小，它代表了笔的笔触形状，如果我们把一支铅笔笔尖削成方形的，那么画出来的线条会是一条弯曲的“矩形”，这就是笔触的意思。除了 `ROUND`，`Paint.Cap` 还提供了另外两种类型：`SQUARE` 和 `BUTT`，具体大家自己去 try~

### **setStrokeJoin(Paint.Join join)**

这个方法用于设置结合处的形态，就像上面的代码中我们虽说是花了一条心电线，但是这条线其实是由无数条小线拼接成的，拼接处的形状就由该方法指定。

上面的例子中我们还使用到了一个方法

### **setShadowLayer(float radius, float dx, float dy, int shadowColor)**

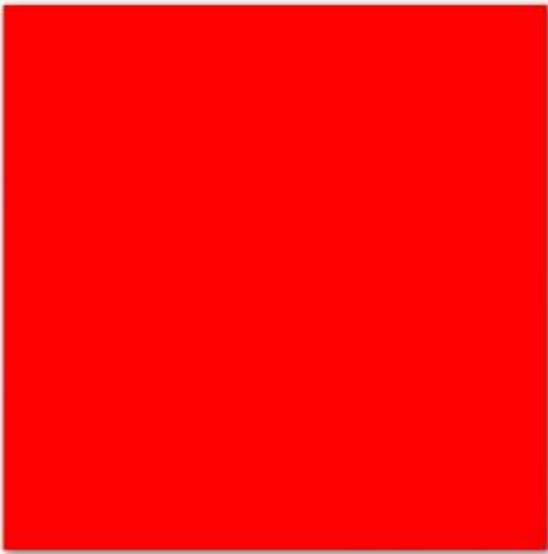
该方法为我们绘制的图形添加一个阴影层效果：

[java] [view](#) [plain](#) [copy](#) [print](#)

```
1. public class ShadowView extends View {
2.     private static final int RECT_SIZE = 800;// 方形大小
3.     private Paint mPaint;// 画笔
4.
5.     private int left, top, right, bottom;// 绘制时坐标
6.
7.     public ShadowView(Context context, AttributeSet attrs) {
8.         super(context, attrs);
9.         // setShadowLayer 不支持 HW
10.        setLayerType(LAYER_TYPE_SOFTWARE, null);
11.
12.        // 初始化画笔
13.        initPaint();
14.
15.        // 初始化资源
16.        initRes(context);
```

```
17.    }
18.
19.    /**
20.     * 初始化画笔
21.     */
22.    private void initPaint() {
23.        // 实例化画笔
24.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
25.        mPaint.setColor(Color.RED);
26.        mPaint.setStyle(Style.FILL);
27.        mPaint.setShadowLayer(10, 3, 3, Color.DKGRAY);
28.    }
29.
30.    /**
31.     * 初始化资源
32.     */
33.    private void initRes(Context context) {
34.        /*
35.         * 计算位图绘制时左上角的坐标使其位于屏幕中心
36.         */
37.        left = MeasureUtil.getScreenSize((Activity) context)[0] / 2 - RECT_SIZE / 2;
38.        top = MeasureUtil.getScreenSize((Activity) context)[1] / 2 - RECT_SIZE / 2;
39.        right = MeasureUtil.getScreenSize((Activity) context)[0] / 2 + RECT_SIZE / 2;
40.        bottom = MeasureUtil.getScreenSize((Activity) context)[1] / 2 + RECT_SIZE / 2;
41.    }
42.
43.    @Override
44.    protected void onDraw(Canvas canvas) {
45.        super.onDraw(canvas);
46.
47.        // 先绘制位图
48.        canvas.drawRect(left, top, right, bottom);
49.    }
50. }
```

radius 表示阴影的扩散半径，而 dx 和 dy 表示阴影平面上的偏移值，shadowColor 就不说了阴影颜色，最后提醒一点 setShadowLayer 同样不支持 HW 哦！



上面我们讲 **MaskFilter** 的时候曾用其子类 **BlurMaskFilter** 模拟过类似效果，跟 **BlurMaskFilter** 比起来这方法是不是更简捷呢？但是 **BlurMaskFilter** 能做的 **setShadowLayer** 却不一定能做到哦！

至此，**Paint** 下的几乎所有方法我们都已经学习了，正如我之前所说，工欲善其事必先利其器，自定义 **View** 很重要的一部分就是如何去画一个 **Perfect** 的图形，Android 给我们提供了绝大部分的方法和类来模拟现实中真正的画笔，我们只需要学会如何灵活运用即可，牛逼的人不需要复杂的技术即可实现复杂的效果~~这就是实力、才是真正大神~~~~So~共勉

**PS:** 这一节讲得有点戳~~Because something happen in my life maybe changed my life，还有最近忙着改的一个相机应用的开源项目，所以很多事 ==，讲得不是很好大家见谅，下一期补上更多精彩的内容

下集精彩预告：**Paint** 中的方法几乎都已经概述了一遍，但是有个方法我们还没说 **setShader(Shader shader)**，这个方法很重要吗？其实一般，但是其涉及到的一个东西对我们来说相当重要：**Matrix**，这个神秘的东西究竟是做什么用的呢？卖个关子先，下一节我们将会结束整个 **Paint** 的学习，是不是有点想跃跃一试的冲动？下期我将会给大家带来两个好玩的例子，这两个例子来自于群里朋友的提问，不多说了，上图：



## 4. 自定义控件其实很简单(4)

上一节结尾的时候我们说到，Paint 类中我们还有一个方法没讲

[java] view plaincopyprint?

1. `setShader(Shader shader)`

这个方法呢其实也没有什么特别的，那么为什么我们要把它单独分离出来讲那么异类呢？难道它贿赂了我吗？显然不是的，哥视金钱如粪土（我的要求很低，只需要一克反物质即可）！怎么可能做出如此下三滥的事情！之所以要把这货单独拿出来是为了引出 Android 在图形变换中非常重要的一个类！这个类是什么呢？我也先不说，咱还是先来看看 Shader:

```
public class
Shader
extends Object

java.lang.Object http://blog.csdn.net/aigestudio
↳ android.graphics.Shader

► Known Direct Subclasses
BitmapShader, ComposeShader, LinearGradient, RadialGradient, SweepGradient
```

Shader 类呢也是个灰常灰常简单的类，它有五个子类，像 PathEffect 一样每个子类都实现了一种 Shader，Shader 在三维软件中我们称之为着色器，其作用嘛就像它的名字一样是来

给图像着色的或者更通俗的说法是上色！这么说该懂了吧！再不懂去厕所哭去！这五个 Shader 里最异类的是 **BitmapShader**, 因为只有它是允许我们载入一张图片来给图像着色，那我们还是先来看看这个怪胎吧

### BitmapShader

只有一个含参的构造方法 **BitmapShader (Bitmap bitmap, Shader.TileMode tileX, Shader.TileMode tileY)** 而其他的四个兄弟姐妹呢都有两个！它只有一个蛋，又一魂谈！那好吧，我们来看看它是什么个效果，顺便呢也学习一下 **Shader** 的用法先，来看我们熟悉的代码：

[java] view plaincopyprint?

```
1. public class ShaderView extends View {
2.     private static final int RECT_SIZE = 400;// 矩形尺寸的一半
3.
4.     private Paint mPaint;// 画笔
5.
6.     private int left, top, right, bottom;// 矩形坐上右下坐标
7.
8.     public ShaderView(Context context, AttributeSet attrs) {
9.         super(context, attrs);
10.        // 获取屏幕尺寸数据
11.        int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
12.
13.        // 获取屏幕中点坐标
14.        int screenX = screenSize[0] / 2;
15.        int screenY = screenSize[1] / 2;
16.
17.        // 计算矩形左上右下坐标值
18.        left = screenX - RECT_SIZE;
19.        top = screenY - RECT_SIZE;
20.        right = screenX + RECT_SIZE;
21.        bottom = screenY + RECT_SIZE;
22.
23.        // 实例化画笔
24.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
25.
26.        // 获取位图
27.        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.a);
28.
29.        // 设置着色器
30.        mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.CLAMP));
31.    }
```

```
32.  
33.     @Override  
34.     protected void onDraw(Canvas canvas) {  
35.         // 绘制矩形  
36.         canvas.drawRect(left, top, right, bottom, mPaint);  
37.     }  
38. }
```

如果上面我们没有设置 **Shader**:

[java] view plaincopyprint?

```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.Tile  
Mode.CLAMP));
```

那么我们 **Draw** 出来的图像一定是一个位于屏幕正中黑色的正方形,但是我们设置了 **Shader** 后还是一样的吗? 看看效果:



我靠!这什么玩意!罪过罪过!真是看不懂!别急, **Shader.TileMode** 里有三种模式: **CLAMP**、**MIRROR** 和 **REPEATA**, 我们看看其他两种模式是什么效果呢:

[java] view plaincopyprint?

```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.MIRROR, Shader.Til  
eMode.MIRROR));
```



唉？这效果还能接受，我们还能看得出一点效果，说白了就是上下左右的镜像而已，那再看看 REPEATA：

[java] view plaincopyprint?

```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.REPEAT, Shader.TileMode.REPEAT));
```



这个就更简单了，明显的一个重复效果，而 REPEAT 也就是重复的意思，同理 MIRROR 也就是镜像的意思，这个很好理解吧。那第一个 CLAMP 模式究竟特么的是什么东西呢？看效果根本看不出来，我们不妨换个思维，BitmapShader (Bitmap bitmap, Shader.TileMode tileX, Shader.TileMode tileY)的第一个参数是位图这个很显然，而后两个参数则分别表示 XY 方向上的着色模式，既然可以分开设置，那么我们是不是可以这样设置一个 Shader 呢？

[java] view plaincopyprint?

```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.MIRROR));
```

也就是说我们在 X 轴方向上采取 CLAMP 模式而 Y 轴方向上采取 MIRROR 模式，那么这样肯定可行的撒：



大家可以看到图像分为两部分左边呢 Y 轴镜像了，而右边像是被拉伸了一样怪怪的！其实 CLAMP 的意思就是边缘拉伸的意思，比如上图中左边 Y 轴镜像了，而右边会紧挨着左边将图像边缘上的第一个像素沿 X 轴复制！产生一种被拉伸的效果！就像扯蛋，不过这里扯的不是蛋而是图像边缘的第一个像素，就是这么简单。但是！作为一个严谨的男人必须要又一个严谨的态度！这时我就会想，如果两种模式互换会怎样呢？比如：

[java] view plaincopyprint?

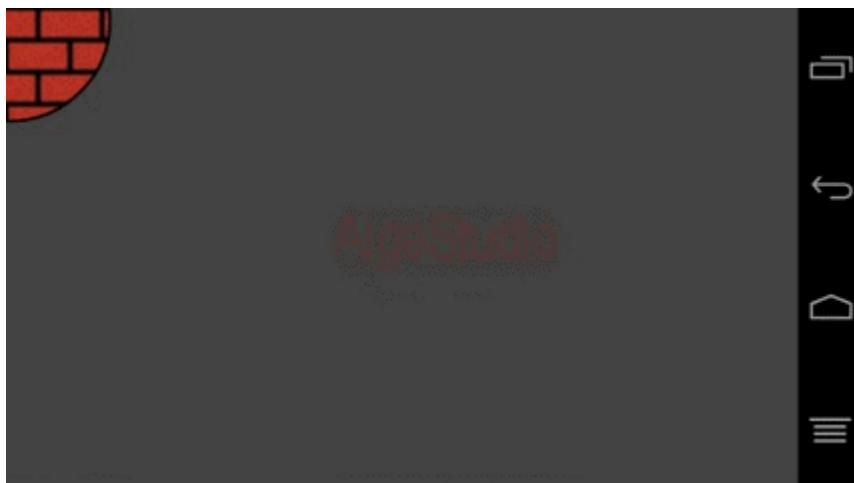
```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.MIRROR, Shader.TileMode.CLAMP));
```

这样来看，应该是 X 轴会镜像而 Y 轴会拉伸对吧，看看效果：



这.....好像跟我们想象中的不大一样唉.....是我们做错了吗？不是的，结合上一个例子大家有没有注意 `BitmapShader` 是先应用了 Y 轴的模式而 X 轴是后应用的！所以着色是先在 Y 轴拉伸了然后再沿着 X 轴重复对吧（阴笑 ing.....）？！

要善于发现生活规律！我曾经说过：存在必定合理，那么这么一个玩意有什么用处呢？给大家看一个非常有趣的东西：



这玩意是不是感觉跟以前那种小霸王某个游戏的开场动画很类似？我们就是利用 `BitmapShader` 来实现的，而且实现方法也异常简单：

[java] view plain copy print?

```
1. public class BrickView extends View {  
2.     private Paint mFillPaint, mStrokePaint; // 填充和描边的画笔  
3.     private BitmapShader mBitmapShader; // Bitmap 着色器  
4.  
5.     private float posX, posY; // 触摸点的 XY 坐标
```

```
6.  
7.     public BrickView(Context context, AttributeSet attrs) {  
8.         super(context, attrs);  
9.  
10.        // 初始化画笔  
11.        initPaint();  
12.    }  
13.  
14.    /**  
15.     * 初始化画笔  
16.     */  
17.    private void initPaint() {  
18.        /*  
19.         * 实例化描边画笔并设置参数  
20.         */  
21.        mStrokePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
22.  
23.        mStrokePaint.setColor(0xFF000000);  
24.        mStrokePaint.setStyle(Paint.Style.STROKE);  
25.        mStrokePaint.setStrokeWidth(5);  
26.  
27.        // 实例化填充画笔  
28.        mFillPaint = new Paint();  
29.  
30.        /*  
31.         * 生成 BitmapShader  
32.         */  
33.        Bitmap mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.brick);  
34.        mBitmapShader = new BitmapShader(mBitmap, Shader.TileMode.REPEAT, Shader.TileMode.REPEAT);  
35.        mFillPaint.setShader(mBitmapShader);  
36.    }  
37.  
38.    @Override  
39.    public boolean onTouchEvent(MotionEvent event) {  
40.        /*  
41.         * 手指移动时获取触摸点坐标并刷新视图  
42.         */  
43.        if (event.getAction() == MotionEvent.ACTION_MOVE) {  
44.            posX = event.getX();  
45.            posY = event.getY();  
46.            invalidate();
```

```
47.         }
48.         return true;
49.     }
50.
51.     @Override
52.     protected void onDraw(Canvas canvas) {
53.         // 设置画笔背景色
54.         canvas.drawColor(Color.DKGRAY);
55.
56.         /*
57.          * 绘制圆和描边
58.          */
59.         canvas.drawCircle(posX, posY, 300, mFillPaint);
60.         canvas.drawCircle(posX, posY, 300, mStrokePaint);
61.     }
62. }
```

我只是单纯地加载了一张板砖的贴图然后呢在 `BitmapShader` 中 XY 轴重复就这么简单 = =，是不是感脚被耍了一样好无聊~~~~是你不动脑而已.....囧！来看看另一个 `Shader` 叫做

### LinearGradient

线性渐变，顾名思义这锤子玩意就是来画渐变的，实际上 `Shader` 的五个子类中除了上面我们说的那个怪胎，还有个变形金刚 `ComposeShader` 外其余三个都是渐变只是效果不同而已，而这个 `LinearGradient` 线性渐变一说大家估计都懂，先来看张效果图：



是不是秒懂了！恩，说明你头脑简单，这个实现也很简单，具体代码跟上面的 `BitmapShader` 一样只是把 `BitmapShader` 换成了 `LinearGradient` 而已：

[java] view plain copy print?

```
1. mPaint.setShader(new LinearGradient(left, top, right, bottom, Color.RED, Color.YELLOW, Shader.TileMode.REPEAT));
```

上面我们提到过除了 `BitmapShader` 外其他子类都有两个构造方法，上面我们用到了

[java] view plaincopyprint?

```
1. LinearGradient(float x0, float y0, float x1, float y1, int color0, int color1, Shader.TileMode tile)
```

这是 `LinearGradient` 最简单的一个构造方法，参数虽多其实很好理解 `x0` 和 `y0` 表示渐变的起点坐标而 `x1` 和 `y1` 则表示渐变的终点坐标，这两点都是相对于屏幕坐标系而言的，而 `color0` 和 `color1` 则表示起点的颜色和终点的颜色，这些即便是 213 也能懂 - - .....`Shader.TileMode` 上面我们给的是 `REPEAT` 重复但是并没有任何效果，这时因为我们渐变的起点和终点都落在了图形的两端，整个渐变 `shader` 已经填充了图形所以不起作用，如果我们改改，把终点坐标变一下：

[java] view plaincopyprint?

```
1. mPaint.setShader(new LinearGradient(left, top, right - RECT_SIZE, bottom - RECT_SIZE, Color.RED, Color.YELLOW, Shader.TileMode.REPEAT));
```

此时我们渐变终点坐标落在了图形的终点上，根据我们的 `REPEAT` 模式，会呈现一个渐变重复的效果：



仅仅两种颜色的渐变根本无法满足我们身体的欲望，太单调乏味！我们是不是可以定义多种颜色渐变呢？答案是必须的，`LinearGradient` 的另一个构造方法

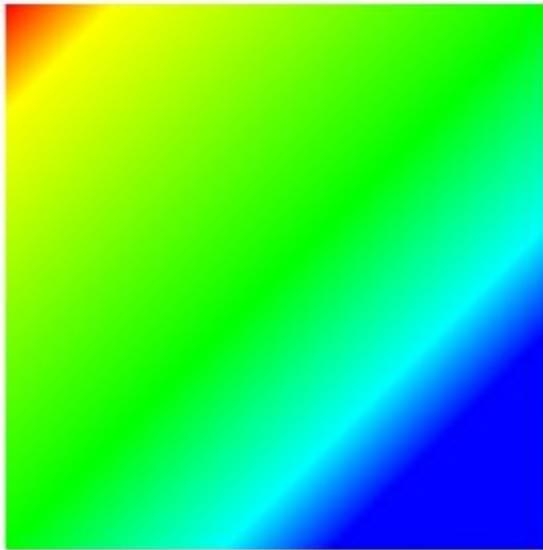
[java] view plaincopyprint?

```
1. LinearGradient(float x0, float y0, float x1, float y1, int[] colors, float[] positions, Shader.TileMode tile)
```

就为我们实现了这么一个功能：

[java] view plaincopyprint?

```
1. mPaint.setShader(new LinearGradient(left, top, right, bottom, new int[] { Color.RED, Color.YELLOW, Color.GREEN, Color.CYAN, Color.BLUE }, new float[] { 0, 0.1F, 0.5F, 0.7F, 0.8F }, Shader.TileMode.MIRROR));
```

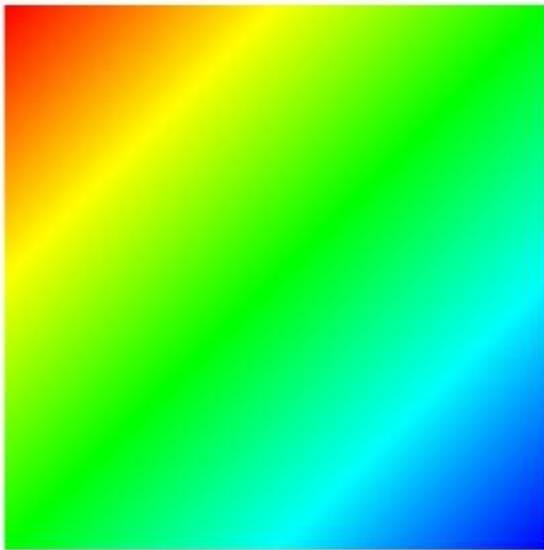


前面四个参数也是定义坐标的不扯了 **colors** 是一个 **int** 型数组，我们用来定义所有渐变的颜色，**positions** 表示的是渐变的相对区域，其取值只有 0 到 1，上面的代码中我们定义了一个 [0, 0.1F, 0.5F, 0.7F, 0.8F]，意思就是红色到黄色的渐变起点坐标在整个渐变区域(**left, top, right, bottom** 定义了渐变的区域) 的起点，而终点则在渐变区域长度 \* 10%的地方，而黄色到绿色呢则从渐变区域 10%开始到 50%的地方以此类推，**positions** 可以为空：

[java] view plaincopyprint?

```
1. mPaint.setShader(new LinearGradient(left, top, right, bottom, new int[] { Color.RED, Color.YELLOW, Color.GREEN, Color.CYAN, Color.BLUE }, null, Shader.TileMode.MIRROR));
```

为空时各种颜色的渐变将会均分整个渐变区域：



实际应用中线性渐变也是一个非常好玩的东西，比如哦我们呢可以通过它和混合模式一起制作我们常见的图片倒影效果：



效果并不复杂实现也非常简单：

[java] view plaincopyprint?

```
1. public class ReflectView extends View {  
2.     private Bitmap mSrcBitmap, mRefBitmap;// 位图
```

```
3.     private Paint mPaint;// 画笔
4.     private PorterDuffXfermode mXfermode;// 混合模式
5.
6.     private int x, y;// 位图起点坐标
7.
8.     public ReflectView(Context context, AttributeSet attrs) {
9.         super(context, attrs);
10.
11.        // 初始化资源
12.        initRes(context);
13.    }
14.
15.    /*
16.     * 初始化资源
17.     */
18.    private void initRes(Context context) {
19.        // 获取源图
20.        mSrcBitmap = BitmapFactory.decodeResource(getResources(), R.drawable
21.            .gril);
22.
23.        // 实例化一个矩阵对象
24.        Matrix matrix = new Matrix();
25.        matrix.setScale(1F, -1F);
26.
27.        // 生成倒影图
28.        mRefBitmap = Bitmap.createBitmap(mSrcBitmap, 0, 0, mSrcBitmap.getWidth(),
29.            mSrcBitmap.getHeight(), matrix, true);
30.
31.        int screenW = MeasureUtil.getScreenSize((Activity) context)[0];
32.        int screenH = MeasureUtil.getScreenSize((Activity) context)[1];
33.
34.        x = screenW / 2 - mSrcBitmap.getWidth() / 2;
35.        y = screenH / 2 - mSrcBitmap.getHeight() / 2;
36.
37.        // .....
38.        mPaint = new Paint();
39.        mPaint.setShader(new LinearGradient(x, y + mSrcBitmap.getHeight(), x
40.            , y + mSrcBitmap.getHeight() + mSrcBitmap.getHeight() / 4, 0xAA000000, Color
41.            .TRANSPARENT, Shader.TileMode.CLAMP));
42.    }
43.
```

```
43.     @Override
44.     protected void onDraw(Canvas canvas) {
45.         canvas.drawColor(Color.BLACK);
46.         canvas.drawBitmap(mSrcBitmap, x, y, null);
47.
48.         int sc = canvas.saveLayer(x, y + mSrcBitmap.getHeight(), x + mRefBitmap.getWidth(), y + mSrcBitmap.getHeight() * 2, null, Canvas.ALL_SAVE_FLAG);
49.
50.         canvas.drawBitmap(mRefBitmap, x, y + mSrcBitmap.getHeight(), null);
51.
52.         mPaint.setXfermode(mXfermode);
53.
54.         canvas.drawRect(x, y + mSrcBitmap.getHeight(), x + mRefBitmap.getWidth(), y + mSrcBitmap.getHeight() * 2, mPaint);
55.
56.         mPaint.setXfermode(null);
57.
58.         canvas.restoreToCount(sc);
59.     }
60. }
```

## SweepGradient

的意思是梯度渐变，也称之为扫描式渐变，因为其效果有点类似雷达的扫描效果，他也有两个构造方法：

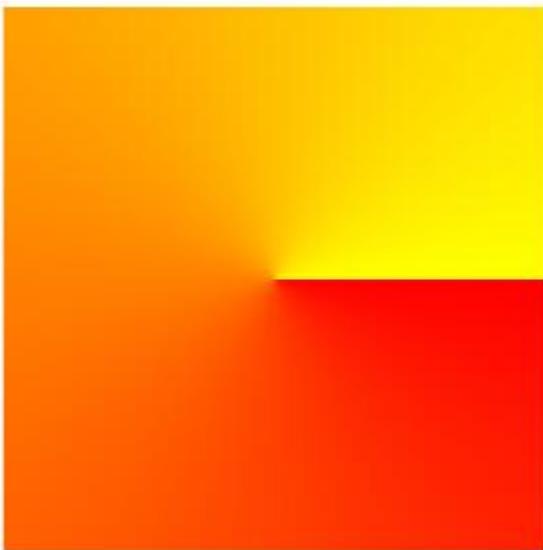
[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

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

其实都跟 `LinearGradient` 差不多的，简直没什么可说的，直接上效果跳过无聊的讲解：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. mPaint.setShader(new SweepGradient(screenX, screenY, Color.RED, Color.YELLOW
));
```



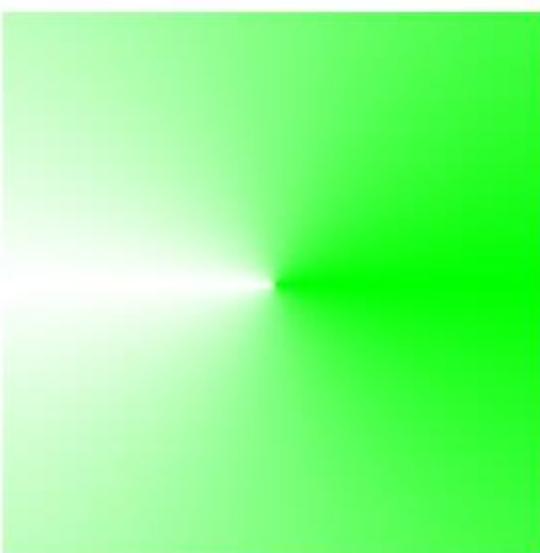
[java] view plaincopyprint?

```
1. SweepGradient(float cx, float cy, int[] colors, float[] positions)
```

类似，不重复浪费口水：

[java] view plaincopyprint?

```
1. mPaint.setShader(new SweepGradient(screenX, screenY, new int[] { Color.GREEN
    , Color.WHITE, Color.GREEN }, null));
```



### RadialGradient

径向渐变，径向渐变说的简单点就是个圆形中心向四周渐变的效果，他也一样有两个构造方法……艾玛！我都要吐了……

[java] view plaincopyprint?

```
1. RadialGradient (float centerX, float centerY, float radius, int centerColor,  
    int edgeColor, Shader.TileMode tileMode)
```

简单，一看就懂，例子劳资都懒得上了，中！

[java] view plaincopyprint?

```
1. RadialGradient (float centerX, float centerY, float radius, int centerColor,  
    int edgeColor, Shader.TileMode tileMode)
```

同上！还是说点有意思的，来个妹子养养眼：



是不是很漂亮？不过哥压根不放在眼里，哥女朋友比她更漂亮~~好，我们应用 1/6 里学到的知识给她校校色，因为这张图片的颜色实在太过鲜艳了不符合小清新的 LOMO 风格~~~~~

[java] view plaincopyprint?

```
1. public class DreamEffectView extends View {  
2.     private Paint mBitmapPaint;// 位图画笔  
3.     private Bitmap mBitmap;// 位图  
4.     private PorterDuffXfermode mXfermode;// 图形混合模式  
5.     private int x, y;// 位图起点坐标  
6.     private int screenW, screenH;// 屏幕宽高
```

```
7.
8.     public DreamEffectView(Context context, AttributeSet attrs) {
9.         super(context, attrs);
10.
11.        // 初始化资源
12.        initRes(context);
13.
14.        // 初始化画笔
15.        initPaint();
16.    }
17.
18.    /**
19.     * 初始化资源
20.     *
21.     * @param context
22.     *          丢你螺母
23.     */
24.    private void initRes(Context context) {
25.        // 获取位图
26.        mBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.gril);
27.
28.        // 实例化混合模式
29.        mXfermode = new PorterDuffXfermode(PorterDuff.Mode.SCREEN);
30.
31.        screenW = MeasureUtil.getScreenSize((Activity) context)[0];
32.        screenH = MeasureUtil.getScreenSize((Activity) context)[1];
33.
34.        x = screenW / 2 - mBitmap.getWidth() / 2;
35.        y = screenH / 2 - mBitmap.getHeight() / 2;
36.    }
37.
38.    /**
39.     * 初始化画笔
40.     */
41.    private void initPaint() {
42.        // 实例化画笔
43.        mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
44.
45.        // 去饱和、提亮、色相矫正
46.        mBitmapPaint.setColorFilter(new ColorMatrixColorFilter(new float[] {
47.            0.8587F, 0.2940F, -0.0927F, 0, 6.79F, 0.0821F, 0.9145F, 0.0634F, 0, 6.79F,
48.            0.2019F, 0.1097F, 0.7483F, 0, 6.79F, 0, 0, 0, 1, 0 }));
49.    }
```

```
48.  
49.    @Override  
50.    protected void onDraw(Canvas canvas) {  
51.        canvas.drawColor(Color.BLACK);  
52.  
53.        // 新建图层  
54.        int sc = canvas.saveLayer(x, y, x + mBitmap.getWidth(), y + mBitmap.  
getHeight(), null, Canvas.ALL_SAVE_FLAG);  
55.  
56.        // 绘制混合颜色  
57.        canvas.drawColor(0xcc1c093e);  
58.  
59.        // 设置混合模式  
60.        mBitmapPaint.setXfermode(mXfermode);  
61.  
62.        // 绘制位图  
63.        canvas.drawBitmap(mBitmap, x, y, mBitmapPaint);  
64.  
65.        // 还原混合模式  
66.        mBitmapPaint.setXfermode(null);  
67.  
68.        // 还原画布  
69.        canvas.restoreToCount(sc);  
70.    }  
71. }
```

这样感觉是不是好很多了呢？



类似的效果在一些相机特效中称之为梦幻，我最近在做的一个开源相机项目也有类似的效果。但是这样的效果好像还凑合，但是整张图片没重点，我们可以模拟单反相机的暗角效果，压暗图片周围的颜色亮度提亮中心，让整张图片的中心突出来！实现方式有很多种，比如 1/4 我们提到过的 `BlurMsakFilter` 向内模糊就可以得到一个类似的效果，但是 `BlurMsakFilter` 计算出来的像素太生硬毫无生气，这里我们就使用 `RadialGradient` 来模拟一下下效果：

[java] view plaincopyprint?

```
1. public class DreamEffectView extends View {  
2.     private Paint mBitmapPaint, mShaderPaint; // 位图画笔和 Shader 图形的画笔  
3.     private Bitmap mBitmap; // 位图  
4.     private PorterDuffXfermode mXfermode; // 图形混合模式  
5.     private int x, y; // 位图起点坐标  
6.     private int screenW, screenH; // 屏幕宽高  
7.  
8.     public DreamEffectView(Context context, AttributeSet attrs) {  
9.         super(context, attrs);  
10.
```

```
11.         // 初始化资源
12.         initRes(context);
13.
14.         // 初始化画笔
15.         initPaint();
16.     }
17.
18. /**
19. * 初始化资源
20. *
21. * @param context
22. *          丢你螺母
23. */
24. private void initRes(Context context) {
25.     // 获取位图
26.     mBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.gril);
27.
28.     // 实例化混合模式
29.     mXfermode = new PorterDuffXfermode(PorterDuff.Mode.SCREEN);
30.
31.     screenW = MeasureUtil.getScreenSize((Activity) context)[0];
32.     screenH = MeasureUtil.getScreenSize((Activity) context)[1];
33.
34.     x = screenW / 2 - mBitmap.getWidth() / 2;
35.     y = screenH / 2 - mBitmap.getHeight() / 2;
36. }
37.
38. /**
39. * 初始化画笔
40. */
41. private void initPaint() {
42.     // 实例化画笔
43.     mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
44.
45.     // 去饱和、提亮、色相矫正
46.     mBitmapPaint.setColorFilter(new ColorMatrixColorFilter(new float[] {
47.         0.8587F, 0.2940F, -0.0927F, 0, 6.79F, 0.0821F, 0.9145F, 0.0634F, 0, 6.79F,
48.         0.2019F, 0.1097F, 0.7483F, 0, 6.79F, 0, 0, 0, 1, 0 }));
49.
50.     // 实例化 Shader 图形的画笔
51.     mShaderPaint = new Paint();
```

```
51.         // 设置径向渐变，渐变中心当然是图片的中心也是屏幕中心，渐变半径我们直接拿图  
片的高度但是要稍微小一点  
52.         // 中心颜色为透明而边缘颜色为黑色  
53.         mShaderPaint.setShader(new RadialGradient(screenW / 2, screenH / 2,  
mBitmap.getHeight() * 7 / 8, Color.TRANSPARENT, Color.BLACK, Shader.TileMode  
.CLAMP));  
54.     }  
55.  
56.     @Override  
57.     protected void onDraw(Canvas canvas) {  
58.         canvas.drawColor(Color.BLACK);  
59.  
60.         // 新建图层  
61.         int sc = canvas.saveLayer(x, y, x + mBitmap.getWidth(), y + mBitmap.  
getHeight(), null, Canvas.ALL_SAVE_FLAG);  
62.  
63.         // 绘制混合颜色  
64.         canvas.drawColor(0xcc1c093e);  
65.  
66.         // 设置混合模式  
67.         mBitmapPaint.setXfermode(mXfermode);  
68.  
69.         // 绘制位图  
70.         canvas.drawBitmap(mBitmap, x, y, mBitmapPaint);  
71.  
72.         // 还原混合模式  
73.         mBitmapPaint.setXfermode(null);  
74.  
75.         // 还原画布  
76.         canvas.restoreToCount(sc);  
77.  
78.         // 绘制一个跟图片大小一样的矩形  
79.         canvas.drawRect(x, y, x + mBitmap.getWidth(), y + mBitmap.getHeight()  
, mShaderPaint);  
80.     }  
81. }
```

Look Look~~是不是效果更好了呢？四周的亮度被无情地压了下去，重点直接呈现在中心



所以.....大家一定要活用工具~~~~善于组合思考！说起组合，接下来 **Shader** 的最后一个子类简直就是组合他妈生的

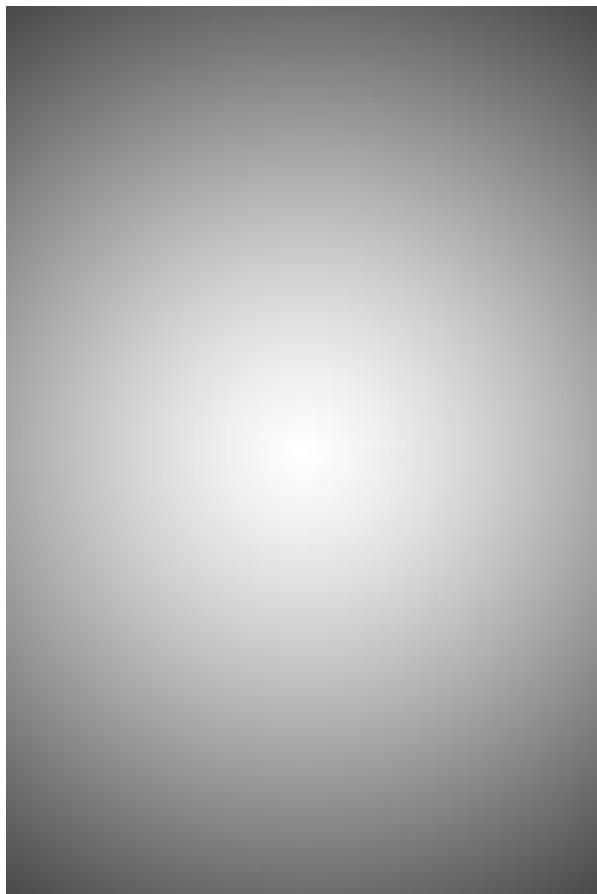
### ComposeShader

就是组合 **Shader** 的意思，顾名思义就是两个 **Shader** 组合在一起作为一个新 **Shader**.....老掉牙的剧情是吧！同样，这锤子玩意也有两个构造方法

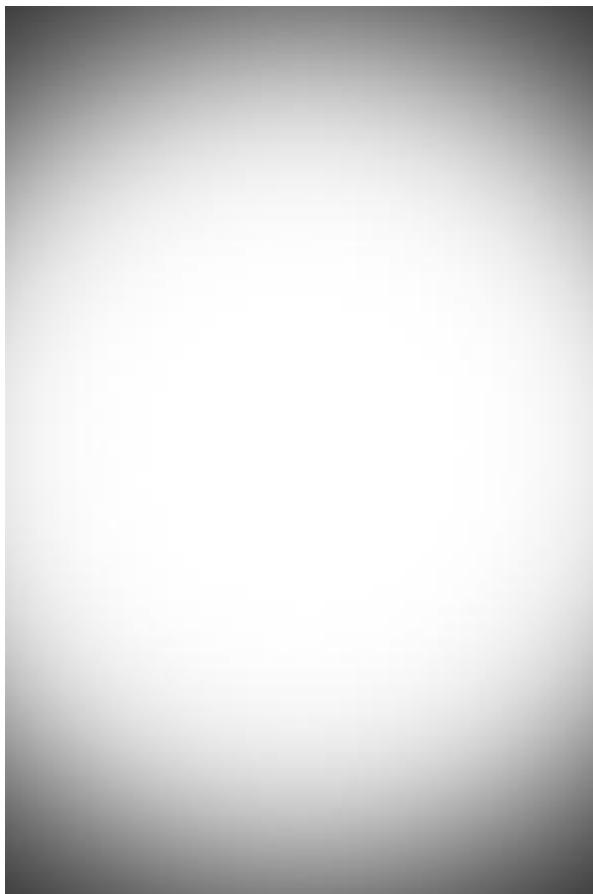
[java] view plain copy print?

1. `ComposeShader (Shader shaderA, Shader shaderB, Xfermode mode)`
2. `ComposeShader (Shader shaderA, Shader shaderB, PorterDuff.Mode mode)`

两个都差不多的，只不过一个指定了只能用 **PorterDuff** 的混合模式而另一个只要是 **Xfermode** 下的混合模式都没问题！上面我们为米女图片加暗角的时候只是单纯地使用了一下径向渐变，但是其实实际获得的效果并不好，因为我们应用的径向渐变是个圆形的，但是我们的图片实际上是个竖向矩形的，直接往上面“盖”一个径向渐变实际效果简而言之应该是这样的：



可见渐变的坡度太平缓了，不符合我们的 **Style**，能不能改一下让它拉伸下变成一个竖向的椭圆形呢？比如下图这样的：



我们来试试看：

[java] view plaincopyprint?

```
1. public class DreamEffectView extends View {  
2.     private Paint mBitmapPaint, mShaderPaint;// 位图画笔和 Shader 图形的画笔  
3.     private Bitmap mBitmap, darkCornerBitmap;// 源图的 Bitmap 和我们自己画的暗角  
    Bitmap  
4.     private PorterDuffXfermode mXfermode;// 图形混合模式  
5.     private int x, y;// 位图起点坐标  
6.     private int screenW, screenH;// 屏幕宽高  
7.  
8.     public DreamEffectView(Context context, AttributeSet attrs) {  
9.         super(context, attrs);  
10.  
11.        // 初始化资源  
12.        initRes(context);  
13.  
14.        // 初始化画笔  
15.        initPaint();  
16.    }  
17.  
18.    /**
```

```
19.     * 初始化资源
20.     *
21.     * @param context
22.     *          丢你螺母
23.     */
24.    private void initRes(Context context) {
25.        // 获取位图
26.        mBitmap = BitmapFactory.decodeResource(context.getResources(), R.drawable.gril);
27.
28.        // 实例化混合模式
29.        mXfermode = new PorterDuffXfermode(PorterDuff.Mode.SCREEN);
30.
31.        screenW = MeasureUtil.getScreenSize((Activity) context)[0];
32.        screenH = MeasureUtil.getScreenSize((Activity) context)[1];
33.
34.        x = screenW / 2 - mBitmap.getWidth() / 2;
35.        y = screenH / 2 - mBitmap.getHeight() / 2;
36.    }
37.
38.    /**
39.     * 初始化画笔
40.     */
41.    private void initPaint() {
42.        // 实例化画笔
43.        mBitmapPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
44.
45.        // 去饱和、提亮、色相矫正
46.        mBitmapPaint.setColorFilter(new ColorMatrixColorFilter(new float[] {
47.            0.8587F, 0.2940F, -0.0927F, 0, 6.79F, 0.0821F, 0.9145F, 0.0634F, 0, 6.79F,
48.            0.2019F, 0.1097F, 0.7483F, 0, 6.79F, 0, 0, 0, 1, 0 })));
49.
50.        // 实例化 Shader 图形的画笔
51.        mShaderPaint = new Paint();
52.
53.        // 根据我们源图的大小生成暗角 Bitmap
54.        darkCornerBitmap = Bitmap.createBitmap(mBitmap.getWidth(), mBitmap.getHeight(),
55.            Bitmap.Config.ARGB_8888);
56.
57.        // 将该暗角 Bitmap 注入 Canvas
58.        Canvas canvas = new Canvas(darkCornerBitmap);
```

```
59.         // 实例化径向渐变
60.         RadialGradient radialGradient = new RadialGradient(canvas.getWidth()
61.             / 2F, canvas.getHeight() / 2F, radiu, new int[] { 0, 0, 0xAA000000 }, new float[]
62.             { 0F, 0.7F, 1.0F }, Shader.TileMode.CLAMP);
63.         // 实例化一个矩阵
64.         Matrix matrix = new Matrix();
65.         // 设置矩阵的缩放
66.         matrix.setScale(canvas.getWidth() / (radiu * 2F), 1.0F);
67.         // 设置矩阵的预平移
68.         matrix.preTranslate(((radiu * 2F) - canvas.getWidth()) / 2F, 0);
69.         // 将该矩阵注入径向渐变
70.         radialGradient.setLocalMatrix(matrix);
71.         // 设置画笔 Shader
72.         mShaderPaint.setShader(radialGradient);
73.         // 绘制矩形
74.         canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(), mShader
75.             Paint);
76.     }
77.     // 新建图层
78.     int sc = canvas.saveLayer(x, y, x + mBitmap.getWidth(), y + mBitmap.
79.         getHeight(), null, Canvas.ALL_SAVE_FLAG);
80.     // 绘制混合颜色
81.     canvas.drawColor(0xcc1c093e);
82.     // 设置混合模式
83.     protected void onDraw(Canvas canvas) {
84.         canvas.drawColor(Color.BLACK);
85.         // 新建图层
86.         int sc = canvas.saveLayer(x, y, x + mBitmap.getWidth(), y + mBitmap.
87.             getHeight(), null, Canvas.ALL_SAVE_FLAG);
88.         // 绘制混合颜色
89.         canvas.drawColor(0xcc1c093e);
90.         // 设置混合模式
91.         mBitmapPaint.setXfermode(mXfermode);
92.         // 绘制位图
93.         canvas.drawBitmap(mBitmap, x, y, mBitmapPaint);
94.         // 还原混合模式
95.         canvas.restore();
96.     }
97. }
```

```
99.         mBitmapPaint.setXfermode(null);
100.
101.        // 还原画布
102.        canvas.restoreToCount(sc);
103.
104.        // 绘制我们画好的径向渐变图
105.        canvas.drawBitmap(darkCornerBitmap, x, y, null);
106.    }
107. }
```

运行一下可以看到如下结果：



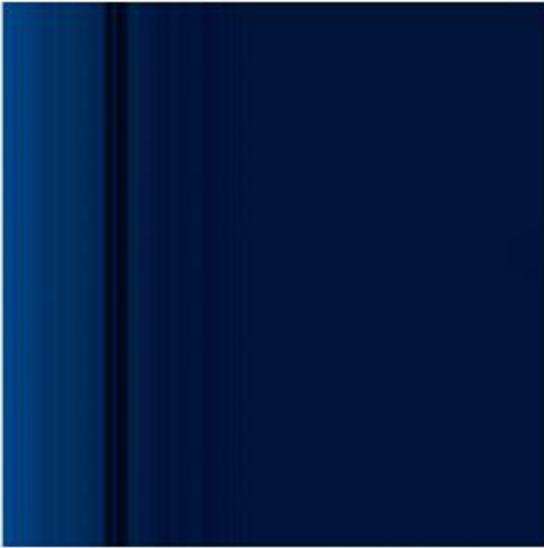
是不是感觉暗角比上面我们做的那个更 **Perfect**? 图片中心大部分区域不受任何干扰只把四角边缘处的亮度压了下去,当然这个效果对哥来说也不是很满意,不过先将就凑合着 == 。上例中我们使用到了两个东西,一个是在独立的 **Canvas** 中绘制自己的 **Bitmap**,在哥下一节我们就会详细讲到这里就先不扯了,而另一个则是我们说的重点: **Matrix**,其实在上面我们做倒影的时候已经简单地使用过了 **Matrix**,那么 **Matrix** 究竟是什么呢? 其中文直译为矩阵,而我更愿意称之为矩阵变换或者图形变换,它和我们 1/6 中学到的图形的混合可谓是双簧,同样地重要!

在本文的开头哥给大家挖了一个坑，在讲 **BitmapShader** 的时候我们在

[java] view plaincopyprint?

```
1. mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.CLAMP));
```

的模式下得到了如下这样狗吃屎的效果：



不知道大家有没有质疑过为什么？如果没有，情有可原！但是在我们使用另外两种模式 **REPEAT** 和 **MIRROR** 的时候难道大家就没想过为什么我们的位图会像那样重复或者镜像吗？是必须那样吗？那个例子中我们使用的是一个边长为 800 置于屏幕中心的矩形，我们何不尝试将其铺满屏幕看看？

[java] view plaincopyprint?

```
1. public class ShaderView extends View {
2.     private static final int RECT_SIZE = 400; // 矩形尺寸的一半
3.
4.     private Paint mPaint; // 画笔
5.
6.     private int left, top, right, bottom; // 矩形坐上右下坐标
7.     private int screenX, screenY;
8.
9.     public ShaderView(Context context, AttributeSet attrs) {
10.         super(context, attrs);
11.
12.         // 获取屏幕尺寸数据
13.         int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
14.     }
}
```

```
15.         // 获取屏幕中点坐标
16.         screenSize = screenSize[0] / 2;
17.         screenY = screenSize[1] / 2;
18.
19.         // 计算矩形左上右下坐标值
20.         left = screenX - RECT_SIZE;
21.         top = screenY - RECT_SIZE;
22.         right = screenX + RECT_SIZE;
23.         bottom = screenY + RECT_SIZE;
24.
25.         // 实例化画笔
26.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
27.
28.         // 获取位图
29.         Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.a);
30.
31.         // 设置着色器
32.         mPaint.setShader(new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.CLAMP));
33.         // mPaint.setShader(new LinearGradient(left, top, right - RECT_SIZE,
34.             bottom - RECT_SIZE, Color.RED, Color.YELLOW, Shader.TileMode.MIRROR));
34.         // mPaint.setShader(new LinearGradient(left, top, right, bottom, new
35.             int[] { Color.RED, Color.YELLOW, Color.GREEN, Color.CYAN, Color.BLUE }, null,
36.             Shader.TileMode.MIRROR));
35.         // mPaint.setShader(new SweepGradient(screenX, screenY, Color.RED, Color.YELLOW));
36.         // mPaint.setShader(new SweepGradient(screenX, screenY, new int[] {
37.             Color.GREEN, Color.WHITE, Color.GREEN }, null));
37.     }
38.
39.     @Override
40.     protected void onDraw(Canvas canvas) {
41.         // 绘制矩形
42.         // canvas.drawRect(left, top, right, bottom, mPaint);
43.         canvas.drawRect(0, 0, screenX * 2, screenY * 2, mPaint);
44.     }
45. }
```

看看效果：



是不是忽然明白了什么？好了，你可以从坑里爬出来了…………这样看是不是懂了？  
`BitmapShader` 是从画布的左上方开始着色，回到我们刚才的问题，这个着色方式必须是这样的么？显然不是！在 `Shader` 类中有一对 `setter` 和 `getter` 方法：`setLocalMatrix(Matrix localM)` 和 `getLocalMatrix(Matrix localM)` 我们可以利用它们来设置或获取 `Shader` 的变换矩阵，比如上面的例子我还是绘制成了一个边长为 800 的矩形：

[java] view plaincopyprint?

```
1. public class ShaderView extends View {  
2.     private static final int RECT_SIZE = 400; // 矩形尺寸的一半  
3.  
4.     private Paint mPaint; // 画笔  
5.  
6.     private int left, top, right, bottom; // 矩形坐上右下坐标  
7.     private int screenX, screenY;  
8.  
9.     public ShaderView(Context context, AttributeSet attrs) {
```

```
10.        super(context, attrs);
11.
12.        // 获取屏幕尺寸数据
13.        int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
14.
15.        // 获取屏幕中点坐标
16.        screenX = screenSize[0] / 2;
17.        screenY = screenSize[1] / 2;
18.
19.        // 计算矩形左上右下坐标值
20.        left = screenX - RECT_SIZE;
21.        top = screenY - RECT_SIZE;
22.        right = screenX + RECT_SIZE;
23.        bottom = screenY + RECT_SIZE;
24.
25.        // 实例化画笔
26.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
27.
28.        // 获取位图
29.        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.a);
30.
31.        // 实例化一个 Shader
32.        BitmapShader bitmapShader = new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.CLAMP);
33.
34.        // 实例一个矩阵对象
35.        Matrix matrix = new Matrix();
36.
37.        // 设置矩阵变换
38.        matrix.setTranslate(left, top);
39.
40.        // 设置 Shader 的变换矩阵
41.        bitmapShader.setLocalMatrix(matrix);
42.
43.        // 设置着色器
44.        mPaint.setShader(bitmapShader);
45.        // mPaint.setShader(new LinearGradient(left, top, right - RECT_SIZE, bottom - RECT_SIZE, Color.RED, Color.YELLOW, Shader.TileMode.MIRROR));
46.        // mPaint.setShader(new LinearGradient(left, top, right, bottom, new int[] { Color.RED, Color.YELLOW, Color.GREEN, Color.CYAN, Color.BLUE }, null, Shader.TileMode.MIRROR));
47.        // mPaint.setShader(new SweepGradient(screenX, screenY, Color.RED, Color.YELLOW));
```

```

48.         // mPaint.setShader(new SweepGradient(screenX, screenY, new int[] {
        Color.GREEN, Color.WHITE, Color.GREEN }, null));
49.     }
50.
51.     @Override
52.     protected void onDraw(Canvas canvas) {
53.         // 绘制矩形
54.         canvas.drawRect(left, top, right, bottom, mPaint);
55.         // canvas.drawRect(0, 0, screenX * 2, screenY * 2, mPaint);
56.     }
57. }

```

不一样的是我在给画笔设置着色器前为我们的着色器设置了一个变换矩阵，让我们的 **Shader** 依据自身的坐标→平移 **left** 个单位↓平移 **top** 个单位，也就是说原本 **shader** 的原点应该是画布（注意不是屏幕！这里只是刚好画布与屏幕重合了而已！切记！）的左上方[0,0]的位置，通过变换移至了[left,top]的位置，如果没问题，**Shader** 此时应该是刚好是从我们矩形的左上方开始着色：



Is anyone has question yet? 是不是感觉有点儿懂 **Matrix** 了? Really?

其实说了半天我们依然没进入到 **Matrix** 的本质，在 1/6 中我们学习了一个和它比较类似的玩意叫做 **ColorMatrix** 大家还记得否？我在讲 **ColorMatrix** 的时候说过其是一个 4x5 的颜色矩阵，而同样，我们的 **Matrix** 也是一个矩阵，只不过不是 4\*5 而是 3\*3 的位置坐标矩阵：

$$\text{Matrix : } \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix}$$

变换变换，既然说到变换那么必定涉及最基本的旋转啊、缩放啊、平移之类，而在 **Matrix** 中除了该三者还多了一种：错切，什么叫错切呢？所谓错切数学中也称之为剪切变换，原理呢就是将图形上所有点的 X/Y 坐标保持不变而按比例平移 Y/X 坐标，并且平移的大小和某

点到 X/Y 轴的垂直距离成正比，变换前后图形的面积不变。其实对于 Matrix 可以这样说：图形的变换实质上就是图形上点的变换，而我们的 Matrix 的计算也正是基于此，比如点 P(x0,y0) 经过上面的矩阵变换后会去到 P(x,y) 的位置：

$$\text{Matrix} : \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix}$$

$$P_0 = \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

$$P = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} A & B & C \\ D & E & F \\ G & H & I \end{pmatrix} \times \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix} = \begin{pmatrix} x = A * x_0 + B * y_0 + C \\ y = D * x_0 + E * y_0 + F \\ 1 = G * x_0 + H * y_0 + I \end{pmatrix}$$

注：除了平移外，缩放、旋转、错切、透视都是需要一个中心点作为参照的，如果没有平移，默认为图形的[0,0]点，平移只需要指定移动的距离即可，平移操作会改变中心点的位置！非常重要！记牢了！

**PS：**唉……说实话矩阵的计算原理真心不想写……不过算是个小总结吧，爱看看不爱看直接跳过即可……

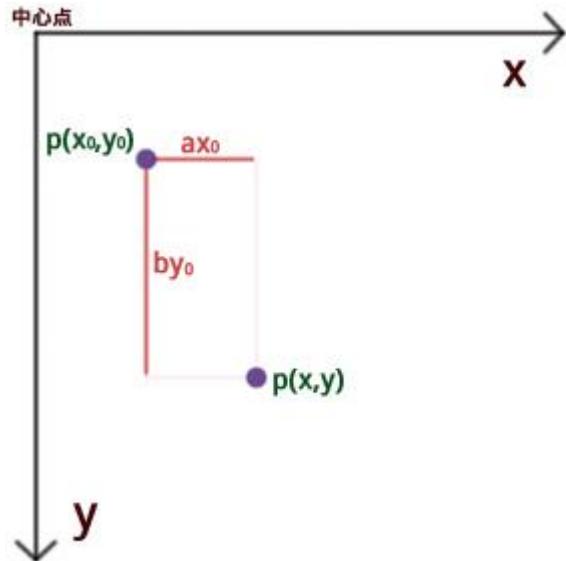
有一点需要注意的是，矩阵的乘法运算是不符合交换律的，因此矩阵  $B * A$  和  $A * B$  是两个截然不同的结果，前者表示  $A$  右乘  $B$ ，是列变换；后者表示  $A$  左乘  $B$ ，是行变换。如果有心的童鞋会发现 Matrix 类中会有三大类方法：`setXXX`、`preXXX` 和 `postXXX`，而 `preXXX` 和 `postXXX` 就是分别表示矩阵的左右乘，也有前后乘的说法，对于不懂矩阵的人来说都一样 = = ……但是要注意一点！！！大家在理解 Matrix 的时候要把它想象成一个容器，什么容器呢？存放我们变换信息的容器，Matrix 的所有方法都是针对其自身的！！！当我们把所有的变换操作做完后再“一次性地”把它注入我们想要的地方，比如上面我们为 `shader` 注入了一个 Matrix。还有一点要注意，一定要注意：`ColorMatrix` 和 `Matrix` 在应用给其他对象时都是左乘的，而其自身内部是可以左右乘的！千万别搞混了！UnderStand？一定要理解这一点，不然我只能哭晕在厕所了压根没法讲下去你也不会听的懂……

上图的公式中，GHI 都表示的是透视参数，一般情况下我们不会去处理，三维的透视我更乐意使用 Camare，所以很多时候 G 和 H 的值都为 0 而 I 的值恒为 1，至于为什么如果有时间待会会说。

所有的 Matrix 变换中最好理解的其实是缩放变换，因为缩放的本质其实就是图形上所有的点 X/Y 轴沿着中心点放大一定的倍数，比如：

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

这么一个矩阵变换实质就是  $x = x_0 * a$ 、 $y = y_0 * b$ , 难度系数: 0

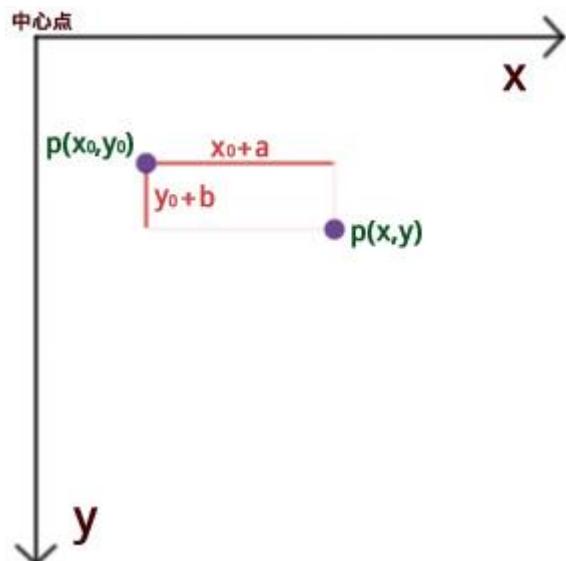


X/Y 轴分别放大  $a\bslash b$  倍

相对来说平移稍难但是也好理解:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

同理  $x = x_0 + a$ 、 $y = y_0 + b$ , 难度系数: 0



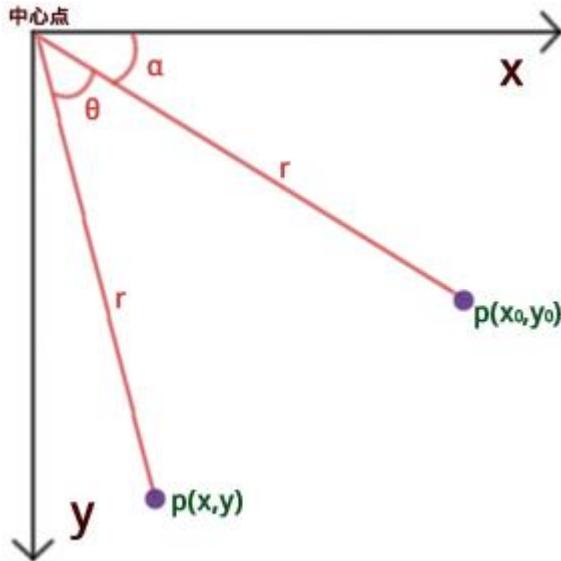
X/Y 轴分别平移! % # % ..... % ..... # % %

旋转就很复杂了.....分为两种：一种是直接绕默认中点[0,0]旋转，另一种是指定中点，也就是将中点[0,0]平移后在旋转：

直接绕[0,0]顺时针转：

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

唉、这个先看图吧：



根据三角函数的关系我们可以得出  $p(x,y)$  的坐标：

$$x = r * \cos(\theta + \alpha) = r * \cos \alpha * \cos \theta - r * \sin \alpha * \sin \theta.$$

$$y = r * \sin(\theta + \alpha) = r * \sin \alpha * \cos \theta + r * \cos \alpha * \sin \theta.$$

同样根据三角函数的关系我们也可以得出  $p(x_0,y_0)$  的坐标：

$$x_0 = r * \cos \alpha$$

$$y_0 = r * \sin \alpha$$

上述两公式结合我们则可以得出简化后的  $p(x,y)$  的坐标：

$$x = x_0 * \cos \theta - y_0 * \sin \theta$$

$$y = y_0 * \cos \theta + x_0 * \sin \theta$$

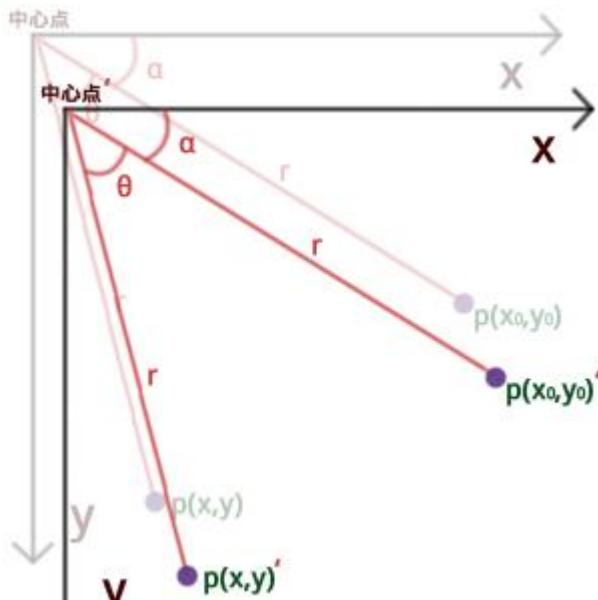
这是什么公式呢？是不是就是上面矩阵的乘积呢？囧.....

绕点  $p(a,b)$  顺时针转：

其实绕某个点旋转没有想象中的那么复杂，相对于绕中心点来说就多了两步：先将坐标原点移到我们的  $p(a,b)$  处然后执行旋转最后再把坐标原点移回去：

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix}$$

对了……开头忘说了……矩阵的乘法是从右边开始的，额，其实也只有上面这算式才有多个矩阵相乘 == 因，也就是说最右边的两个会先乘，大家看看最右边的两个乘积是什么……是不是就是我们把原点移动到  $P(a,b)$  后  $[x_0,y_0]$  的新坐标啊？然后继续往左乘，旋转一定得角度这跟上面  $[0,0]$  旋转是一样的，最后往左乘把坐标还原



不扯了，你听得懂甚好！不懂没关系！真的没关系！哥不骗你！哥从不骗妹子……

哎……错切我也先不说了，大家听了这么多槽点是不是头都大了？麻痹的做个变换还这么麻烦劳资 TM 还不如不做！是的，这样做变换真心太麻烦！要是开发 Android 这么麻烦的话特么谁还玩？所以这些复杂繁琐的扑街玩意 Android 早就为我们封装好了……压根就不需要我们去管那么多！上面我们曾提到的 setXXX、preXXX 和 postXXX 方法就是 Android 为我们封装好的针对不同运算的“档位”，那么怎么用呢？非常非常简单，你压根可以现在忘记上面我们说到的各种计算原理，比如，我这里还是拿 BitmapShader 来说吧，我们想要移动 Shader 的位置：

[java] view plain copy print?

```

1. public class MatrixView extends View {
2.     private static final int RECT_SIZE = 400;// 矩形尺寸的一半
3.
4.     private Paint mPaint;// 画笔
5.
6.     private int left, top, right, bottom;// 矩形坐上右下坐标
7.     private int screenX, screenY;
8.
9.     public MatrixView(Context context, AttributeSet attrs) {

```

```
10.        super(context, attrs);
11.
12.        // 获取屏幕尺寸数据
13.        int[] screenSize = MeasureUtil.getScreenSize((Activity) context);
14.
15.        // 获取屏幕中点坐标
16.        screenX = screenSize[0] / 2;
17.        screenY = screenSize[1] / 2;
18.
19.        // 计算矩形左上右下坐标值
20.        left = screenX - RECT_SIZE;
21.        top = screenY - RECT_SIZE;
22.        right = screenX + RECT_SIZE;
23.        bottom = screenY + RECT_SIZE;
24.
25.        // 实例化画笔
26.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
27.
28.        // 获取位图
29.        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.a);
30.
31.        // 实例化一个 Shader
32.        BitmapShader bitmapShader = new BitmapShader(bitmap, Shader.TileMode.CLAMP, Shader.TileMode.CLAMP);
33.
34.        // 实例一个矩阵对象
35.        Matrix matrix = new Matrix();
36.
37.        // 设置矩阵变换
38.        matrix.setTranslate(500, 500);
39.
40.        // 设置 Shader 的变换矩阵
41.        bitmapShader.setLocalMatrix(matrix);
42.
43.        // 设置着色器
44.        mPaint.setShader(bitmapShader);
45.    }
46.
47.    @Override
48.    protected void onDraw(Canvas canvas) {
49.        // 绘制矩形
50.        // canvas.drawRect(left, top, right, bottom, mPaint);
51.        canvas.drawRect(0, 0, screenX * 2, screenY * 2, mPaint);
```

```
52.      }
53. }
```

这段代码其实跟前面的有点类似，不纠结，我们只是简单地在 Matrix 中做了个平移：



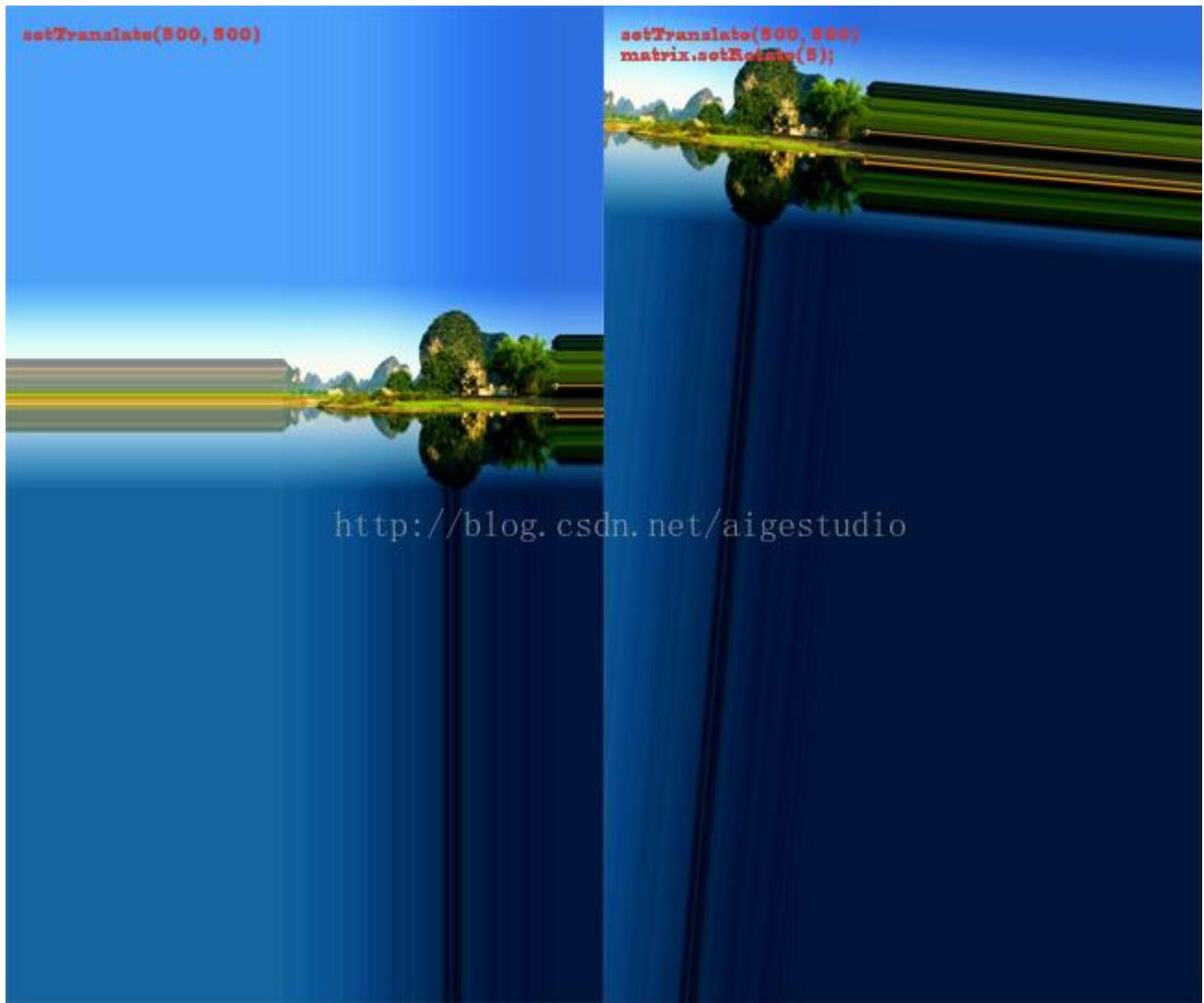
<http://blog.csdn.net/aigestudio>

效果也很简单，那我们再来个旋转 5 度？

[java] view plaincopyprint?

```
1. // 设置矩阵变换
2. matrix.setTranslate(500, 500);
3. matrix.setRotate(5);
```

完事后看看效果尼玛怎么是介个样子？说好的平移呢？被狗吃了？



Why? 其实是这样的，在我们 new 了一个 Matrix 对象后，这个 Matrix 对象中已经就为我们封装了一组原始数据：

[java] view plaincopyprint?

```
1. float[] {  
2.     1, 0, 0  
3.     0, 1, 0  
4.     0, 0, 1  
5. }
```

而我们的 setXXX 方法执行的操作是把原本 Matrix 对象中的数据重置，重新设置新的数据，比如：

[java] view plaincopyprint?

```
1. matrix.setTranslate(500, 500);
```

后数据即变为：

[java] view plaincopyprint?

```
1. float[]{
2.     1, 0, 500
3.     0, 1, 500
4.     0, 0, 1
5. }
```

而如果再旋转了呢？比如我们上面的：

[java] view plaincopyprint?

```
1. matrix.setTranslate(500, 500);
2. matrix.setRotate(5);
```

那旋转的数据就会直接覆盖掉我们平移的数据：

[java] view plaincopyprint?

```
1. float[]{
2.     cos, sin, 0
3.     sin, cos, 0
4.     0, 0, 1
5. }
```

具体参数值我也就不计算了，从这里大家也可以看出 Android 给我们封装的方法是多么的体贴到位~~~你只需要 `setRotate` 个角度即可压根就不需要你关心如何去算的对吧？我们来看另外的两个方法 `preXXX` 和 `postXXX`，这里我把 `setRotate` 换成 `preRotate`：

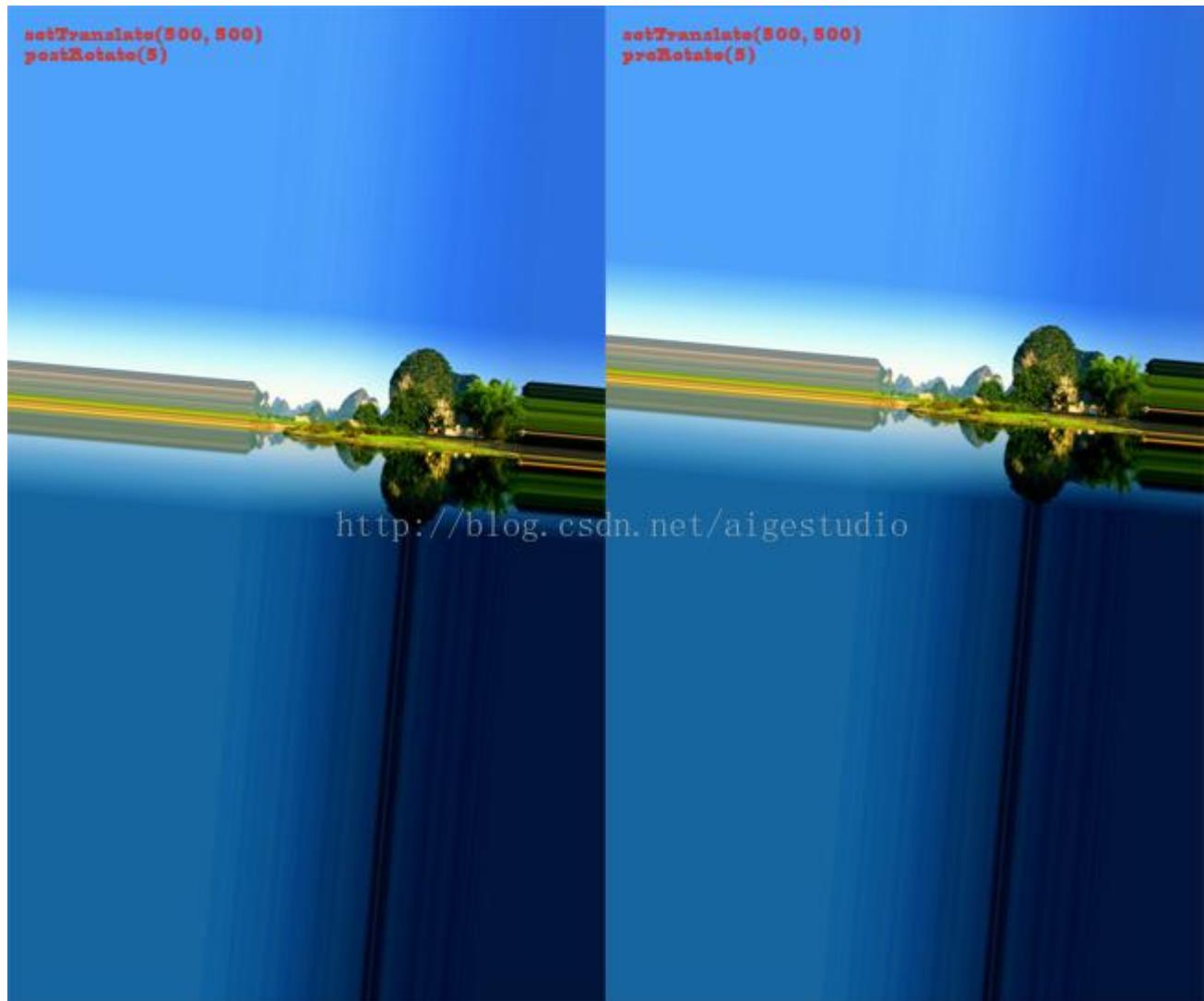
[java] view plaincopyprint?

```
1. matrix.setTranslate(500, 500);
2. matrix.preRotate(5);
```

和

[java] view plaincopyprint?

```
1. matrix.setTranslate(500, 500);
2. matrix.postRotate(5);
```



好像没啥区别啊？看不出有什么特别的对吧？其实呢这是一个谁先谁后的问题，`preXXX` 和 `postXXX` 我们之前说过一个是前乘一个是后乘，那么具体表现是什么样的呢？，非常简单，比如我有如下代码

[java] view plaincopyprint?

```
1. matrix.preScale(0.5f, 1);
2. matrix.setScale(1, 0.6f);
3. matrix.postScale(0.7f, 1);
4. matrix.preTranslate(15, 0);
```

那么 `Matrix` 的计算过程即为： `translate (15, 0) -> scale (1, 0.6f) -> scale (0.7f, 1)`，我们说过 `set` 会重置数据，所以最开始的

[java] view plaincopyprint?

```
1. matrix.preScale(0.5f, 1);
```

也就 GG 了

同样地，对于类似的变换：

[java] view plaincopyprint?

```
1. matrix.preScale(0.5f, 1);
2. matrix.preTranslate(10, 0);
3. matrix.postScale(0.7f, 1);
4. matrix.postTranslate(15, 0);
```

其计算过程为：translate (10, 0) -> scale (0.5f, 1) -> scale (0.7f, 1) -> translate (15, 0)，是不是很简单呢？你一定不傻逼对吧！

那么对于上图的结果真的是一样的吗？这里我教给大家一个方法自己去验证，**Matrix** 有一个 **getValues** 方法可以获取当前 **Matrix** 的变换浮点数组，也就是我们之前说的矩阵：

[java] view plaincopyprint?

```
1. /*
2.  * 新建一个 9 个单位长度的浮点数组
3.  * 因为我们的 Matrix 矩阵是 9 个单位长的对吧
4. */
5. float[] fs = new float[9];
6.
7. // 将从 matrix 中获取到的浮点数组装载进我们的 fs 里
8. matrix.getValues(fs);
9. Log.d("Aige", Arrays.toString(fs)); // 输出看看呗！
```

大家觉得好奇的都可以去验证，这三类方法我就不多说了，**Matrix** 中还有其他很多实用的方法，以后我们用到的时候在讲，因为 **Matrix** 太常用了~~~~

现在，大家回过头去再看看我给妹子图加暗角的那段代码，里面关于 **Matrix** 的操作能大致看懂了么：

[java] view plaincopyprint?

```
1. // 计算径向渐变半径
2. float radiu = canvas.getHeight() * (2F / 3F);
3.
4. // 实例化径向渐变
5. RadialGradient radialGradient = new RadialGradient(canvas.getWidth() / 2F, c
   anvas.getHeight() / 2F, radiu, new int[] { 0, 0, 0xAA000000 }, new float[] {
   0F, 0.7F, 1.0F }, Shader.TileMode.CLAMP);
6.
7. // 实例化一个矩阵
8. Matrix matrix = new Matrix();
9.
10. // 设置矩阵的缩放
11. matrix.setScale(canvas.getWidth() / (radiu * 2F), 1.0F);
```

```
12.  
13. // 设置矩阵的预平移  
14. matrix.preTranslate((radius * 2F) - canvas.getWidth() / 2F, 0);  
15.  
16. // 将该矩阵注入径向渐变  
17. radialGradient.setLocalMatrix(matrix);  
18.  
19. // 设置画笔 Shader  
20. mShaderPaint.setShader(radialGradient);
```

是不是灰常滴简单呢？这里其实你只要注意我们除了平移所有变换操作都是基于一个原点的即可！找对原点你就成功一大半了！

好了，对 `Matrix` 的一个简单介绍就到这里，正如我所说，`Matrix` 的应用是相当广泛的，不仅仅是在我们的 `Shader`，我们的 `canvas` 也有 `setMatrix(matrix)` 方法来设置矩阵变换，更常见的是在 `ImageView` 中对 `ImageView` 进行变换，当我们手指在屏幕上划过一定的距离后根据这段距离来平移我们的控件，根据两根手指之间拉伸的距离和相对于上一次旋转的角度来缩放旋转我们的图片：



[java] view plaincopyprint?

```
1. public class MatrixImageView extends ImageView {  
2.     private static final int MODE_NONE = 0x00123; // 默认的触摸模式  
3.     private static final int MODE_DRAG = 0x00321; // 拖拽模式  
4.     private static final int MODE_ZOOM = 0x00132; // 缩放 or 旋转模式  
5.  
6.     private int mode; // 当前的触摸模式  
7.  
8.     private float preMove = 1F; // 上一次手指移动的距离  
9.     private float saveRotate = 0F; // 保存了的角度值  
10.    private float rotate = 0F; // 旋转的角度  
11.  
12.    private float[] preEventCoor; // 上一次各触摸点的坐标集合  
13.  
14.    private PointF start, mid; // 起点、中点对象  
15.    private Matrix currentMatrix, savedMatrix; // 当前和保存了的 Matrix 对象
```

```
16.     private Context mContext;// Fuck.....
17.
18.     public MatrixImageView(Context context, AttributeSet attrs) {
19.         super(context, attrs);
20.         this.mContext = context;
21.
22.         // 初始化
23.         init();
24.     }
25.
26.     /**
27.      * 初始化
28.      */
29.     private void init() {
30.         /*
31.          * 实例化对象
32.          */
33.         currentMatrix = new Matrix();
34.         savedMatrix = new Matrix();
35.         start = new PointF();
36.         mid = new PointF();
37.
38.         // 模式初始化
39.         mode = MODE_NONE;
40.
41.         /*
42.          * 设置图片资源
43.          */
44.         Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.mylove);
45.         bitmap = Bitmap.createScaledBitmap(bitmap, MeasureUtil.getScreenSize((Activity) mContext)[0], MeasureUtil.getScreenSize((Activity) mContext)[1], true);
46.         setImageBitmap(bitmap);
47.     }
48.
49.     @Override
50.     public boolean onTouchEvent(MotionEvent event) {
51.         switch (event.getAction() & MotionEvent.ACTION_MASK) {
52.             case MotionEvent.ACTION_DOWN:// 单点接触屏幕时
53.                 savedMatrix.set(currentMatrix);
54.                 start.set(event.getX(), event.getY());
55.                 mode = MODE_DRAG;
56.                 preEventCoor = null;
```

```
57.         break;
58.     case MotionEvent.ACTION_POINTER_DOWN:// 第二个点接触屏幕时
59.         preMove = calSpacing(event);
60.         if (preMove > 10F) {
61.             savedMatrix.set(currentMatrix);
62.             calMidPoint(mid, event);
63.             mode = MODE_ZOOM;
64.         }
65.         preEventCoor = new float[4];
66.         preEventCoor[0] = event.getX(0);
67.         preEventCoor[1] = event.getX(1);
68.         preEventCoor[2] = event.getY(0);
69.         preEventCoor[3] = event.getY(1);
70.         saveRotate = calRotation(event);
71.         break;
72.     case MotionEvent.ACTION_UP:// 单点离开屏幕时
73.     case MotionEvent.ACTION_POINTER_UP:// 第二个点离开屏幕时
74.         mode = MODE_NONE;
75.         preEventCoor = null;
76.         break;
77.     case MotionEvent.ACTION_MOVE:// 触摸点移动时
78.         /*
79.          * 单点触控拖拽平移
80.          */
81.         if (mode == MODE_DRAG) {
82.             currentMatrix.set(savedMatrix);
83.             float dx = event.getX() - start.x;
84.             float dy = event.getY() - start.y;
85.             currentMatrix.postTranslate(dx, dy);
86.         }
87.         /*
88.          * 两点触控拖放旋转
89.          */
90.         else if (mode == MODE_ZOOM && event.getPointerCount() == 2) {
91.             float currentMove = calSpacing(event);
92.             currentMatrix.set(savedMatrix);
93.             /*
94.              * 指尖移动距离大于 10F 缩放
95.              */
96.             if (currentMove > 10F) {
97.                 float scale = currentMove / preMove;
98.                 currentMatrix.postScale(scale, scale, mid.x, mid.y);
99.             }
100.            /*

```

```
101.         * 保持两点时旋转
102.         */
103.         if (preEventCoor != null) {
104.             rotate = calRotation(event);
105.             float r = rotate - saveRotate;
106.             currentMatrix.postRotate(r, getMeasuredWidth() / 2, get
107.             MeasuredHeight() / 2);
108.         }
109.         break;
110.     }
111.
112.     setImageMatrix(currentMatrix);
113.     return true;
114. }
115.
116. /**
117. * 计算两个触摸点间的距离
118. */
119. private float calSpacing(MotionEvent event) {
120.     float x = event.getX(0) - event.getX(1);
121.     float y = event.getY(0) - event.getY(1);
122.     return (float) Math.sqrt(x * x + y * y);
123. }
124.
125. /**
126. * 计算两个触摸点的中点坐标
127. */
128. private void calMidPoint(PointF point, MotionEvent event) {
129.     float x = event.getX(0) + event.getX(1);
130.     float y = event.getY(0) + event.getY(1);
131.     point.set(x / 2, y / 2);
132. }
133.
134. /**
135. * 计算旋转角度
136. *
137. * @param 事件对象
138. * @return 角度值
139. */
140. private float calRotation(MotionEvent event) {
141.     double deltaX = (event.getX(0) - event.getX(1));
142.     double deltaY = (event.getY(0) - event.getY(1));
143.     double radius = Math.atan2(deltaY, deltaX);
```

```
144.         return (float) Math.toDegrees(radius);
145.     }
146. }
```

记得在 xml 中设置我们 MatrixImageview 的 scaleType="matrix":

[html] view plaincopyprint?

```
1. <com.aigestudio.customviewdemo.views.MatrixImageview
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:scaleType="matrix" />
```

虽然我们通过 **Matrix** 简单地实现了对 **ImageView** 的变换操作，但是有一些小 **BUG**，比如我们两指缩放/旋转图片后抬起一只手指，此时应该立即切换回平移模式对吧？但是我们上述的代码中却是终止了各种操作，事件机制虽然我们还没讲，但是我们也在这几节中用到了不少，这个简单的问题你能解决么？

我在之前讲到大家可以使用 **Matrix** 的 **getValues(float[])** 方法去验证自己不确定的东西，同时呢，我们也可以使用 **Matrix** 的 **setValues(float[])** 方法来直接给 **Matrix** 设置一个矩阵数组，like:

[java] view plaincopyprint?

```
1. setValues(new float[]{
2.     1, 0, 57
3.     0, 1, 78
4.     0, 0, 1
5.});
```

效果跟

[java] view plaincopyprint?

```
1. matrix.setTranslate(57, 78);
```

是一样的。上面我们说到 **Matrix** 矩阵最后的 3 个数是用来设置透视变换的，为什么最后一个值恒为 1？因为其表示的是在 Z 轴向的透视缩放，这三个值都可以被设置，前两个值跟右手坐标系的 XY 轴有关，大家可以尝试去发现它们之间的规律，我就不多说了。这里多扯一点，大家一定要学会如何透过现象看本质，即便看到的本质不一定就是实质，但是离实质已经不远了，不要一上来就去追求什么底层源码啊、逻辑什么的，就像上面的矩阵变换一样，矩阵的 9 个数作用其实很多人都说不清，与其听别人胡扯还不如自己动手试试你说是吧，不然苦逼的只是你自己。

在实际应用中我们极少会使用到 **Matrix** 的尾三数做透视变换，更多的是使用 **Camare** 摄像机，比如我们使用 **Camare** 让 **ListView** 看起来像倒下去一样：（这里只做了解，已超出我们本系列的范畴）

[html] view plaincopyprint?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:background="#FFFFFF"
5.     android:gravity="center"
6.     android:orientation="vertical" >
7.
8.     <com.aigestudio.customviewdemo.views.AnimListView
9.         android:id="@+id/main_alv"
10.        android:layout_width="match_parent"
11.        android:layout_height="match_parent"
12.        android:scrollbars="none" />
13.
14. </LinearLayout>
```

自定义 ListView 重写 onDraw:

[java] view plaincopyprint?

```
1. public class AnimListView extends ListView {
2.     private Camera mCamera;
3.     private Matrix mMMatrix;
4.
5.     public AnimListView(Context context, AttributeSet attrs) {
6.         super(context, attrs);
7.         mCamera = new Camera();
8.         mMMatrix = new Matrix();
9.     }
10.
11.    @Override
12.    protected void onDraw(Canvas canvas) {
13.        mCamera.save();
14.        mCamera.rotate(30, 0, 0);
15.        mCamera.getMatrix(mMMatrix);
16.        mMMatrix.preTranslate(-getWidth() / 2, -getHeight() / 2);
17.        mMMatrix.postTranslate(getWidth() / 2, getHeight() / 2);
18.        canvas.concat(mMMatrix);
19.        super.onDraw(canvas);
20.        mCamera.restore();
21.    }
22. }
```

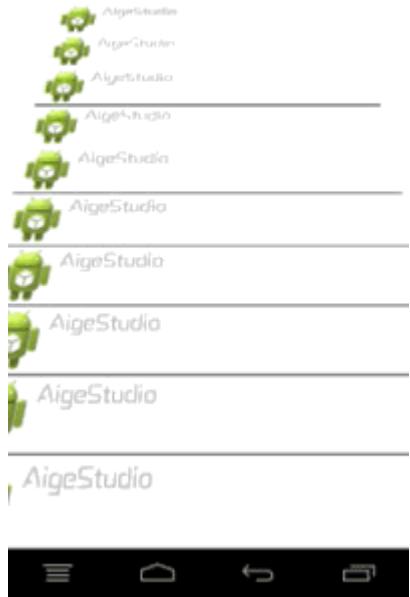
在 MainActivity 中设置数据:

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
```

```
2.
3.     @Override
4.     public void onCreate(Bundle savedInstanceState) {
5.         super.onCreate(savedInstanceState);
6.         setContentView(R.layout.activity_main);
7.
8.         AnimListView animListView = (AnimListView) findViewById(R.id.main_al
v);
9.         animListView.setAdapter(new BaseAdapter() {
10.
11.             @Override
12.                 public View getView(int position, View convertView, ViewGroup pa
rent) {
13.                     convertView = LayoutInflater.from(getApplicationContext()).i
nflate(R.layout.item, null);
14.                     return convertView;
15.                 }
16.
17.             @Override
18.                 public long getItemId(int position) {
19.                     return 0;
20.                 }
21.
22.             @Override
23.                 public Object getItem(int position) {
24.                     return null;
25.                 }
26.
27.             @Override
28.                 public int getCount() {
29.                     return 100;
30.                 }
31.             });
32.         }
33. }
```

效果 like below:

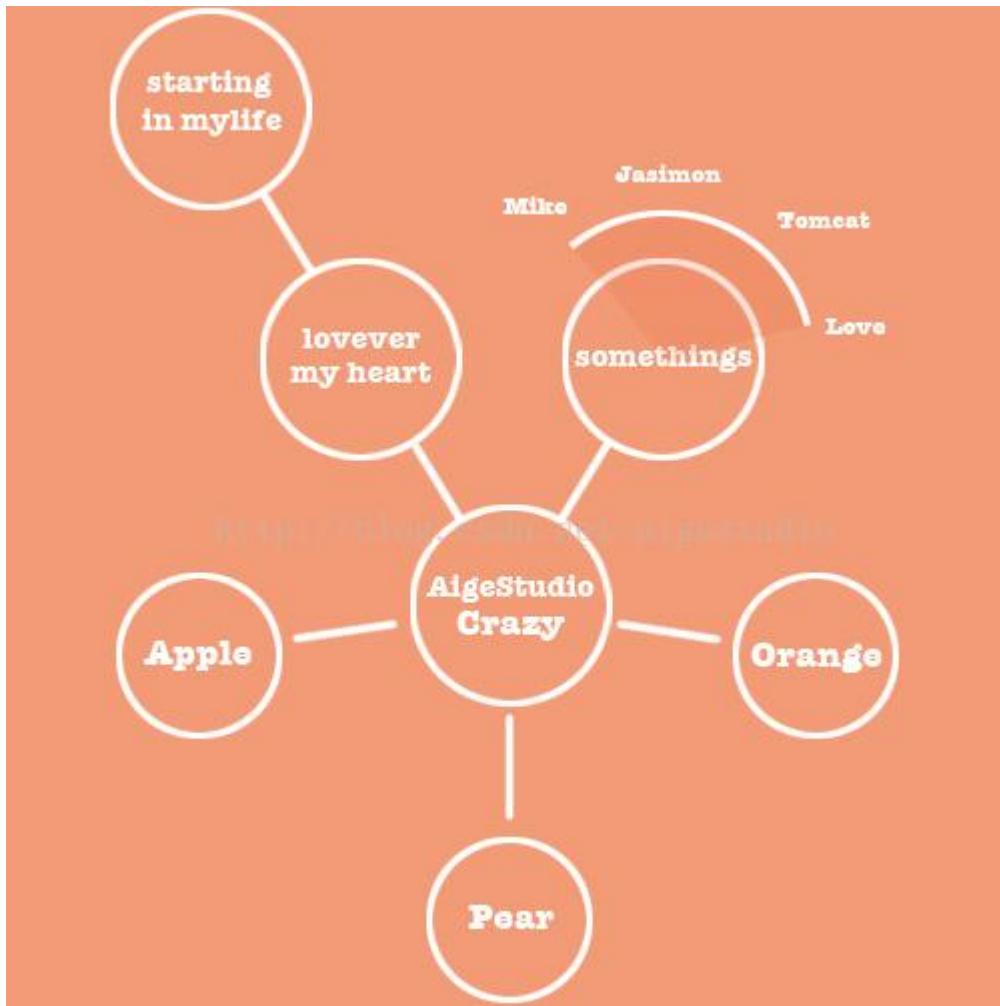


好了，Paint 所有的方法已经 Over 了~~~大家是不是觉得终于解放的赶脚呢？别急……还有个 Canvas……光学会了如何用笔而不知道画什么有何用呢？不过别急，Canvas 我们下一节再细讲，这一节我们算是对 Paint 来个总结，注意不是了断哦！了断可不行，上一节末尾我留了几张图说要在这一节给大家说怎么在 View 中画，这一节呢我们来实现它，注意哦，不要指望在这里你会得到一个完美的控件拿去用……我们还没学怎么去测绘 View 也还没有讲到 ViewGroup，So~~~~在这我们只是单纯地先画。

几个图中最难的大概就是那个各种圈圈的了：

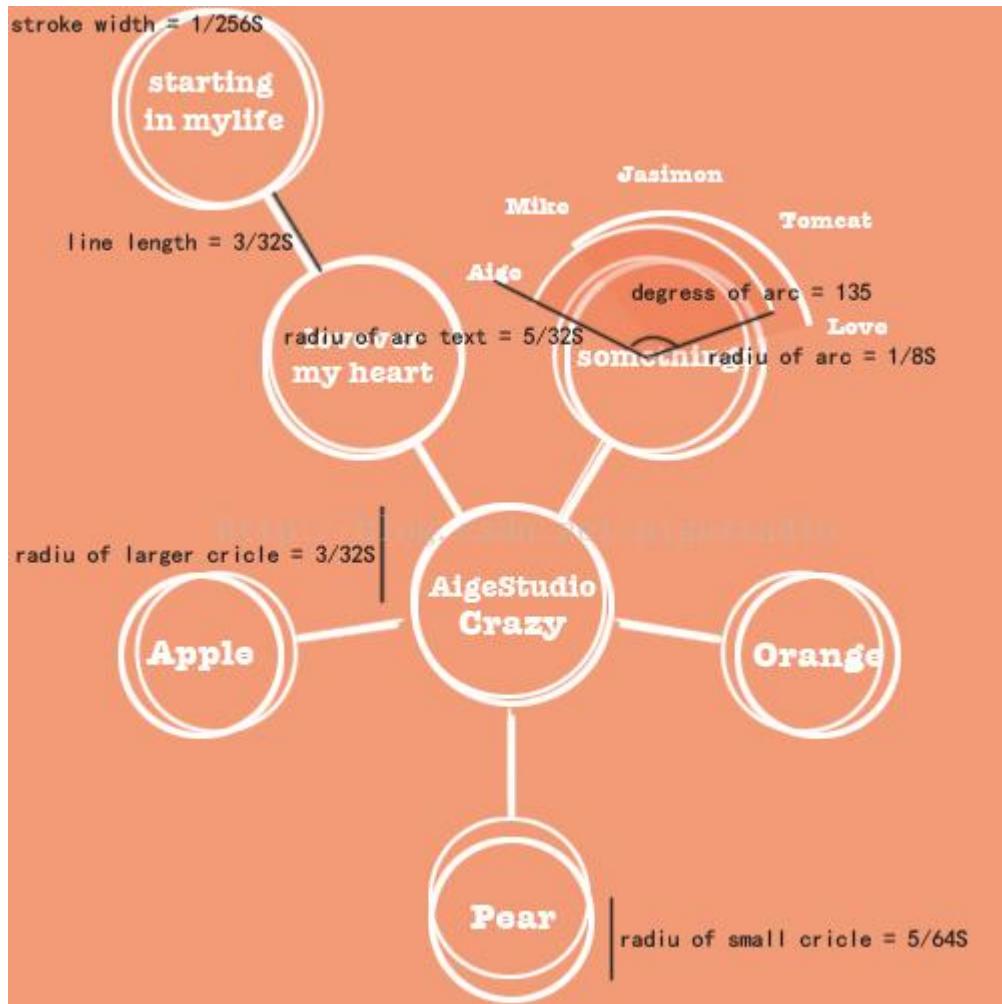


其他的三个图表千篇一律……会画上面那玩意几个图标简直就是小 case，这个圈圈图也是我在群里搜刮的，看不出来是吧，没事，我临摹了一个差不多的：

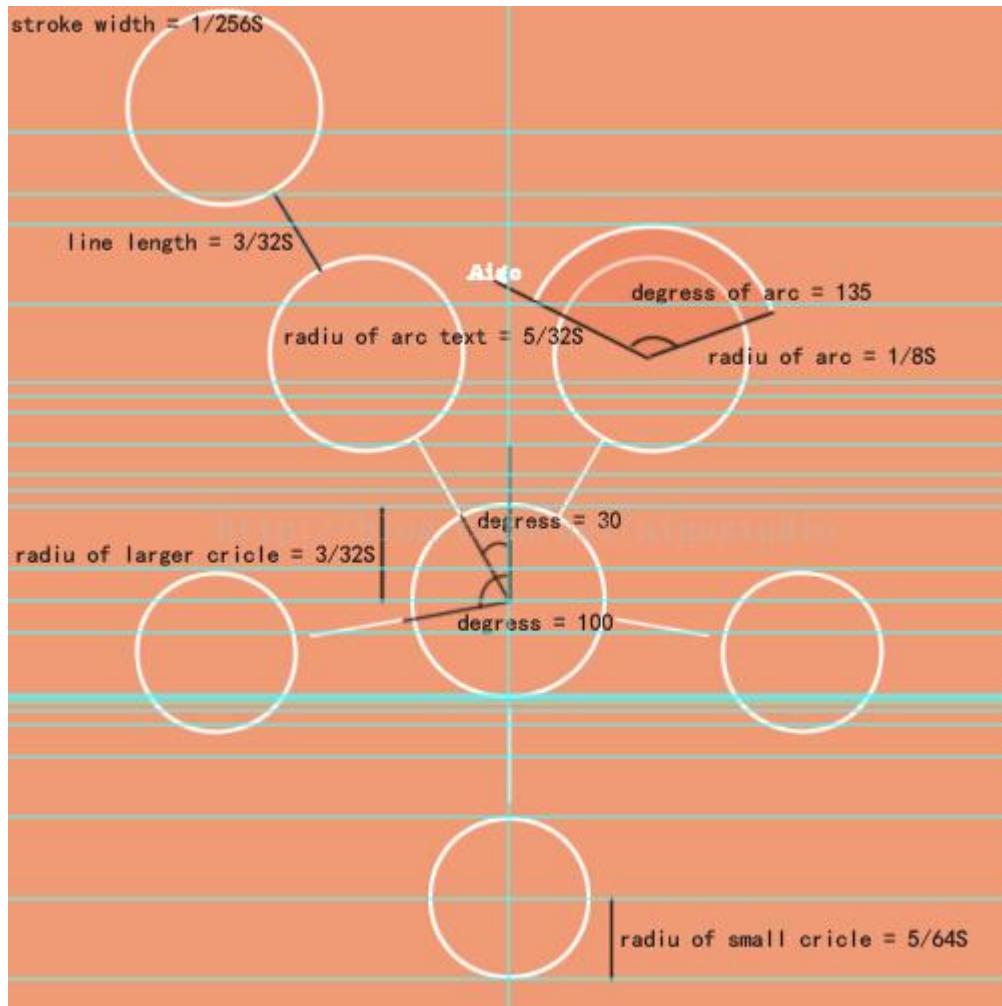


这样看总清晰了吧……大家在自定义一个 View 的时候不要老想着自己是个 Coder，你老是这样想越做不出棒的 View，你要把自己看成一个 Designer，这个 View 将会是由你创造的！而不是你敲出了它……本来在这一系列之后我有单独的番外篇叫如何去设计一个控件，今天在做这个圈圈图的时候突然有这么一个想法还是干脆穿插进来说算了，直接粗暴！

既然我们是一个设计者，那么我们必然要有这么一张图纸去概述我们的 View，因为一般情况下大家都知道美工跟开发者之间是有代沟的，假如，我是说假如美工花了一个很屌的界面，但是你如果直接按着他给你的设计图照着做你会发现做不出一模一样的来……这时你就该调整自己多去与美工沟通在不大改设计图的前提下把难度降到你能力范围内。同样的，我们这里也一样。假如这个圈圈图是 JB 美工给你的效果图，叫你照着做，我们不可能真的这样照着做，因为他给的东西对我们来说无规律可循，而对于开发者来说，有规律的东西往往是最简单，这时我们就要“设计和代码相结合”：

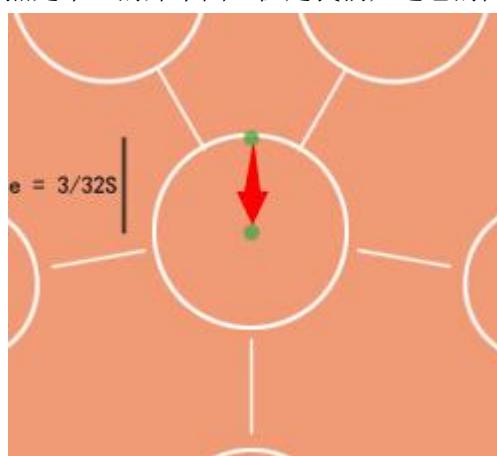


大家看到我在刚才那张临摹图上做了一些改动，从这张草图上来看改动其实并不大，只是把一些位置啊、大小啊什么的做了一些调整，要记住一点，我们的控件要做到在任何屏幕设备上都能完美使用！而不是像布局、资源图之类的还要做好几套，那就是扯蛋！所以我们这的所有尺寸都是以控件的边长  $S$  作为参考依据的：



注：因为我们还没讲如何测绘控件，所以我们在自定义 View 的时候强制控件的长宽一致以简化不必要的口水

还是老套路，先分析一下：控件中心往下是最中心的圆，而其他的六个圆都直接或间接地与其相连，上面的三个是大圆而下面的三个是相对较小的圆，大圆之间的线段是紧挨着的而中心大圆和下面三个小圆之间的线段是有一定距离的，右上方的大圆上方还有一段实体描边弧，弧上有文字，而每个圆内都可以设置文本，大概就是酱紫，那么我们从哪作为插入点呢？当然是中心的那个圆，但是我们知道它的圆心是要往控件中心向下偏移一个半径的：



代码如何实现呢？不用我说你也应该知道：

[java] view plain copy print?

```
1. public class MultiCricleView extends View {  
2.     private static final float STROKE_WIDTH = 1F / 256F, // 描边宽度占比  
3.             LINE_LENGTH = 3F / 32F, // 线段长度占比  
4.             CRICLE_LARGER_RADIUS = 3F / 32F, // 大圆半径  
5.             CRICLE_SMALL_RADIUS = 5F / 64F, // 小圆半径  
6.             ARC_RADIUS = 1F / 8F, // 弧半径  
7.             ARC_TEXT_RADIUS = 5F / 32F; // 弧围绕文字半径  
8.  
9.     private Paint strokePaint; // 描边画笔  
10.  
11.    private int size; // 控件边长  
12.  
13.    private float strokeWidth; // 描边宽度  
14.    private float ccX, ccY; // 中心圆圆心坐标  
15.    private float largeCricleRadius; // 大圆半径  
16.  
17.    public MultiCricleView(Context context, AttributeSet attrs) {  
18.        super(context, attrs);  
19.  
20.        // 初始化画笔  
21.        initPaint(context);  
22.    }  
23.  
24.    /**  
25.     * 初始化画笔  
26.     *  
27.     * @param context  
28.     *          Fuck  
29.     */  
30.    private void initPaint(Context context) {  
31.        /*  
32.         * 初始化描边画笔  
33.         */  
34.        strokePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
35.  
36.        strokePaint.setStyle(Paint.Style.STROKE);  
37.        strokePaint.setColor(Color.WHITE);  
38.        strokePaint.setStrokeCap(Paint.Cap.ROUND);  
39.  
40.    }  
41.    @Override
```

```
41.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
42.         // 强制长宽一致
43.         super.onMeasure(widthMeasureSpec, widthMeasureSpec);
44.     }
45.
46.     @Override
47.     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
48.         // 获取控件边长
49.         size = w;
50.
51.         // 参数计算
52.         calculation();
53.     }
54.
55.     /*
56.      * 参数计算
57.      */
58.     private void calculation() {
59.         // 计算描边宽度
60.         strokeWidth = STROKE_WIDTH * size;
61.
62.         // 计算大圆半径
63.         largeCricleRadius = size * CRICLE_LARGER_RADIUS;
64.
65.         // 计算中心圆圆心坐标
66.         ccX = size / 2;
67.         ccY = size / 2 + size * CRICLE_LARGER_RADIUS;
68.
69.         // 设置参数
70.         setPara();
71.     }
72.
73.     /**
74.      * 设置参数
75.      */
76.     private void setPara() {
77.         // 设置描边宽度
78.         strokePaint.setStrokeWidth(strokeWidth);
79.     }
80.
81.     @Override
82.     protected void onDraw(Canvas canvas) {
83.         // 绘制背景
```

```
84.         canvas.drawColor(0xFFFF29B76);
85.
86.         // 绘制中心圆
87.         canvas.drawCircle(ccX, ccY, largeCricleRadiu, strokePaint);
88.     }
89. }
```

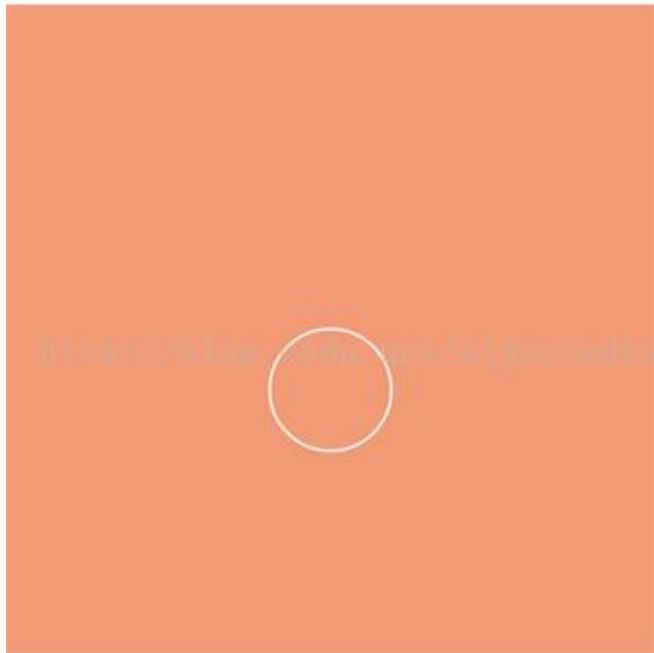
大家可以看到我在 View 重写了这了一个方法:

[java] view plaincopyprint?

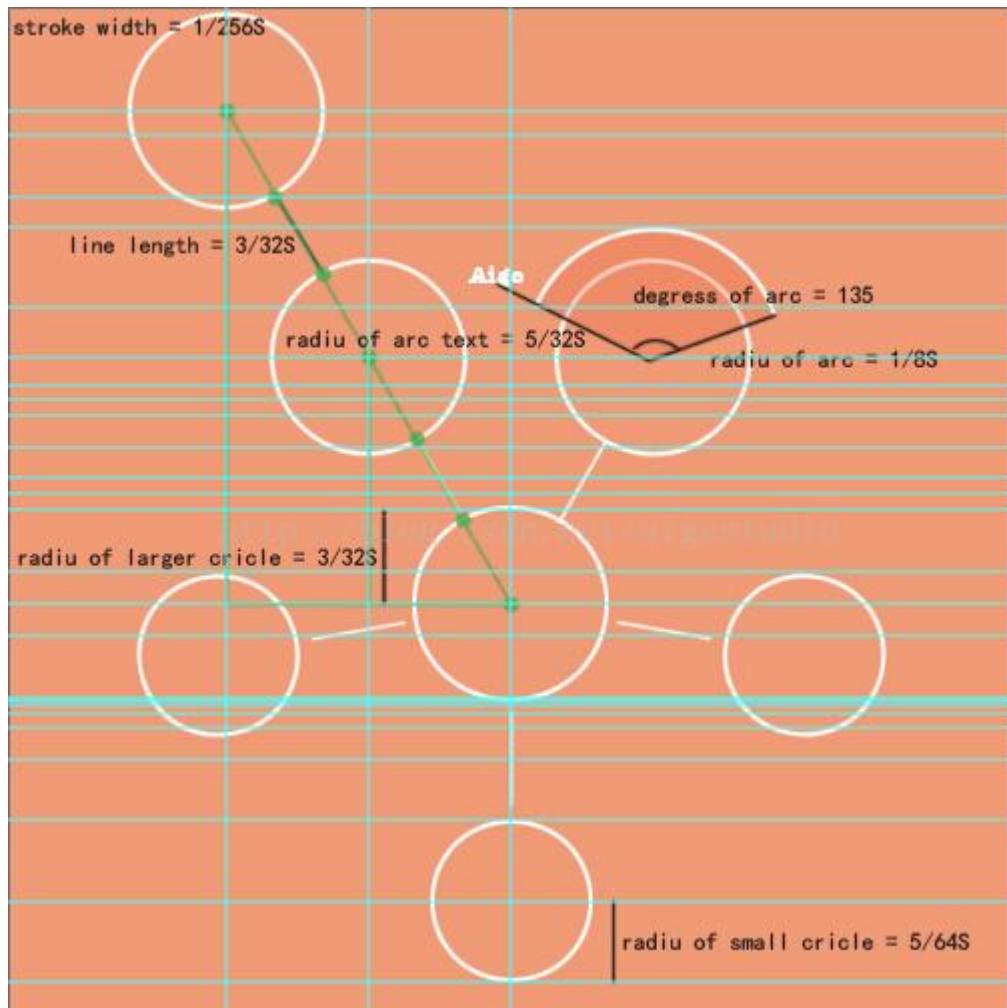
```
1. @Override
2. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.     // 强制长宽一致
4.     super.onMeasure(widthMeasureSpec, widthMeasureSpec);
5. }
```

这个方法就是用来测量控件宽高的，而其从爹那获取的两个参数 widthMeasureSpec 和 heightMeasureSpec 分别封装了 View 的 Size 和 Mode，我们会在 1/2 讲 View 的绘制流程，这里只需跟着我的脚步在这光滑的地板上摩擦摩擦即可 == !

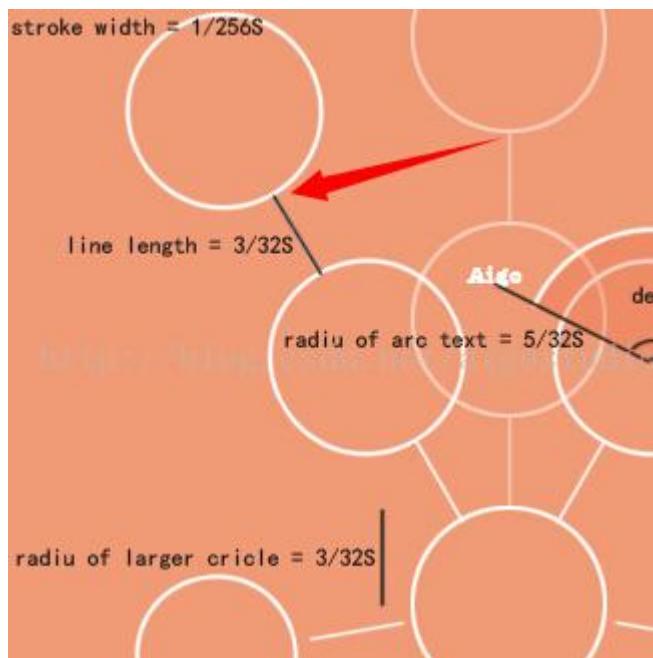
onSizeChanged 这个方法我们前面也提过就不多扯了，效果如下：



然后下一步该如何做呢？画哪个呢？再从左上开始吧……，好我们计算坐标：



没错对吧，绿色的点就是我们要计算的坐标。但是这样去算太 TM 复杂了！毫无违和感！我们细心观察，左上边那一节不就是等同于：



这样的一个变换吗？之前我们曾多次使用到画布的图层，如果我们能这样画图形再以中心圆的圆心坐标为旋转点向右旋转画布岂不是可以很简单：

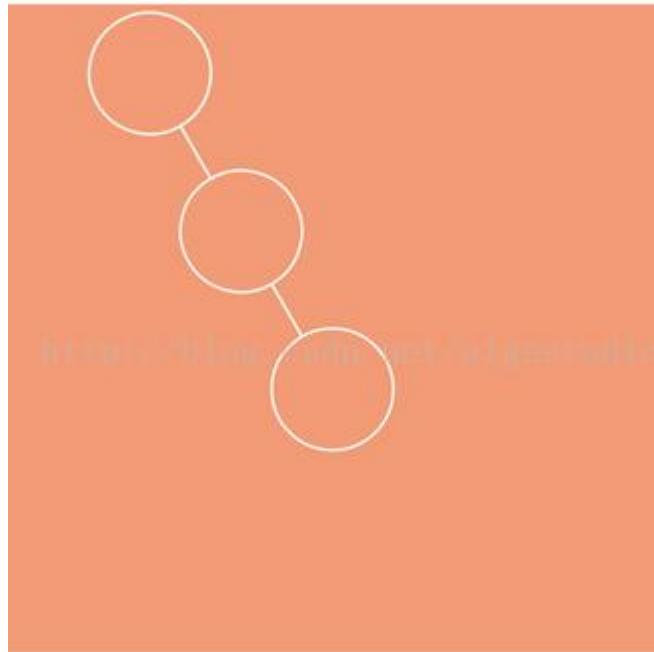
[java] view plaincopyprint?

```
1. public class MultiCricleView extends View {  
2.     private static final float STROKE_WIDTH = 1F / 256F, // 描边宽度占比  
3.             LINE_LENGTH = 3F / 32F, // 线段长度占比  
4.             CRICLE_LARGER_RADIUS = 3F / 32F, // 大圆半径  
5.             CRICLE_SMALL_RADIUS = 5F / 64F, // 小圆半径  
6.             ARC_RADIUS = 1F / 8F, // 弧半径  
7.             ARC_TEXT_RADIUS = 5F / 32F; // 弧围绕文字半径  
8.  
9.     private Paint strokePaint; // 描边画笔  
10.  
11.    private int size; // 控件边长  
12.  
13.    private float strokeWidth; // 描边宽度  
14.    private float ccX, ccY; // 中心圆圆心坐标  
15.    private float largeCricleRadius; // 大圆半径  
16.    private float lineLength; // 线段长度  
17.  
18.    public MultiCricleView(Context context, AttributeSet attrs) {  
19.        super(context, attrs);  
20.  
21.        // 初始化画笔  
22.        initPaint(context);  
23.    }  
24.  
25.    /**  
26.     * 初始化画笔  
27.     *  
28.     * @param context  
29.     *          Fuck  
30.     */  
31.    private void initPaint(Context context) {  
32.        /*  
33.         * 初始化描边画笔  
34.         */  
35.        strokePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
36.  
37.        strokePaint.setStyle(Paint.Style.STROKE);  
38.        strokePaint.setColor(Color.WHITE);  
39.        strokePaint.setStrokeCap(Paint.Cap.ROUND);  
39.    }
```

```
40.  
41.    @Override  
42.    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
  
43.        // 强制长宽一致  
44.        super.onMeasure(widthMeasureSpec, widthMeasureSpec);  
45.    }  
46.  
47.    @Override  
48.    protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
49.        // 获取控件边长  
50.        size = w;  
51.  
52.        // 参数计算  
53.        calculation();  
54.    }  
55.  
56.    /*  
57.     * 参数计算  
58.     */  
59.    private void calculation() {  
60.        // 计算描边宽度  
61.        strokeWidth = STROKE_WIDTH * size;  
62.  
63.        // 计算大圆半径  
64.        largeCricleRadius = size * CRICLE_LARGER_RADIUS;  
65.  
66.        // 计算线段长度  
67.        lineLength = size * LINE_LENGTH;  
68.  
69.        // 计算中心圆圆心坐标  
70.        ccX = size / 2;  
71.        ccY = size / 2 + size * CRICLE_LARGER_RADIUS;  
72.  
73.        // 设置参数  
74.        setPara();  
75.    }  
76.  
77.    /**
78.     * 设置参数
79.     */  
80.    private void setPara() {
81.        // 设置描边宽度
82.        strokePaint.setStrokeWidth(strokeWidth);
```

```
83.    }
84.
85.    @Override
86.    protected void onDraw(Canvas canvas) {
87.        // 绘制背景
88.        canvas.drawColor(0xFFFF29B76);
89.
90.        // 绘制中心圆
91.        canvas.drawCircle(ccX, ccY, largeCricleRadius, strokePaint);
92.
93.        // 绘制左上方图形
94.        drawTopLeft(canvas);
95.    }
96.
97. /**
98. * 绘制左上方图形
99. *
100. * @param canvas
101. */
102. private void drawTopLeft(Canvas canvas) {
103.     // 锁定画布
104.     canvas.save();
105.
106.     // 平移和旋转画布
107.     canvas.translate(ccX, ccY);
108.     canvas.rotate(-30);
109.
110.     // 依次画: 线-圈-线-圈
111.     canvas.drawLine(0, -largeCricleRadius, 0, -lineLength * 2, strokePaint);
112.     canvas.drawCircle(0, -lineLength * 3, largeCricleRadius, strokePaint);
113.     canvas.drawLine(0, -largeCricleRadius * 4, 0, -lineLength * 5, strokePaint);
114.     canvas.drawCircle(0, -lineLength * 6, largeCricleRadius, strokePaint);
115.
116.     // 释放画布
117.     canvas.restore();
118. }
119. }
```

like below:

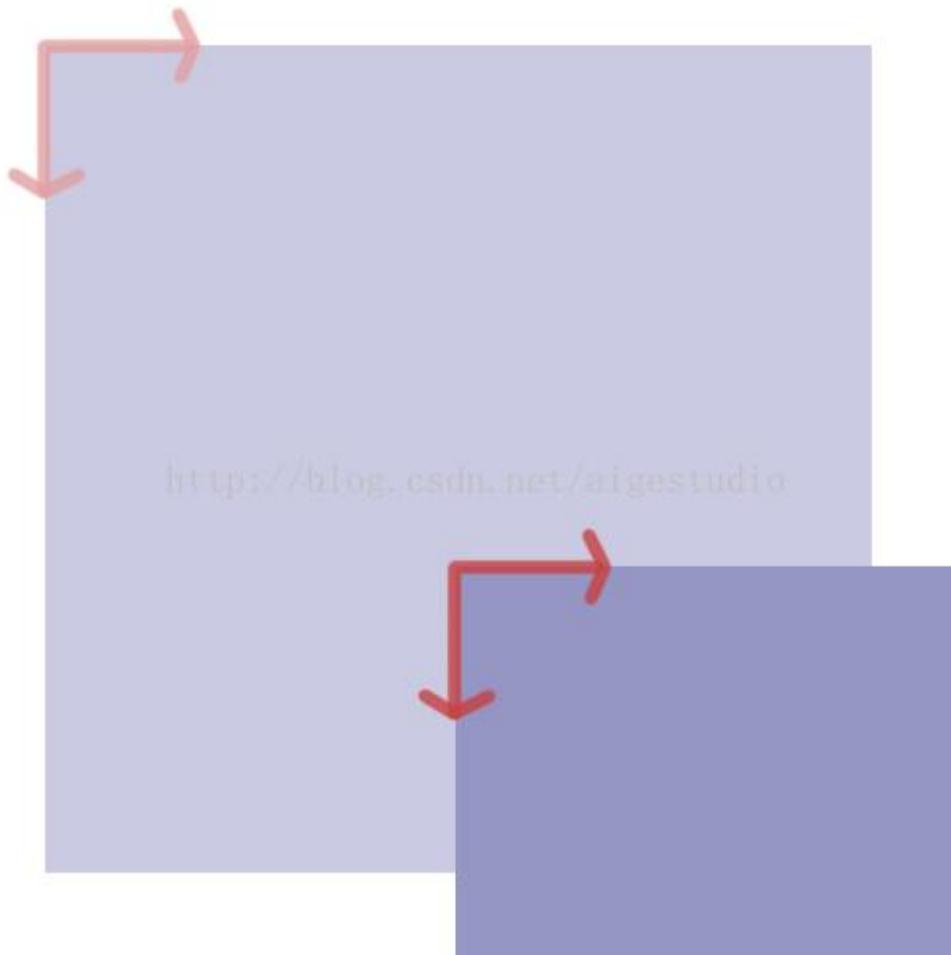


保存和释放画布就不说了，用过 **N** 次了。

[java] view plaincopyprint?

```
1. canvas.translate(ccX, ccY);
```

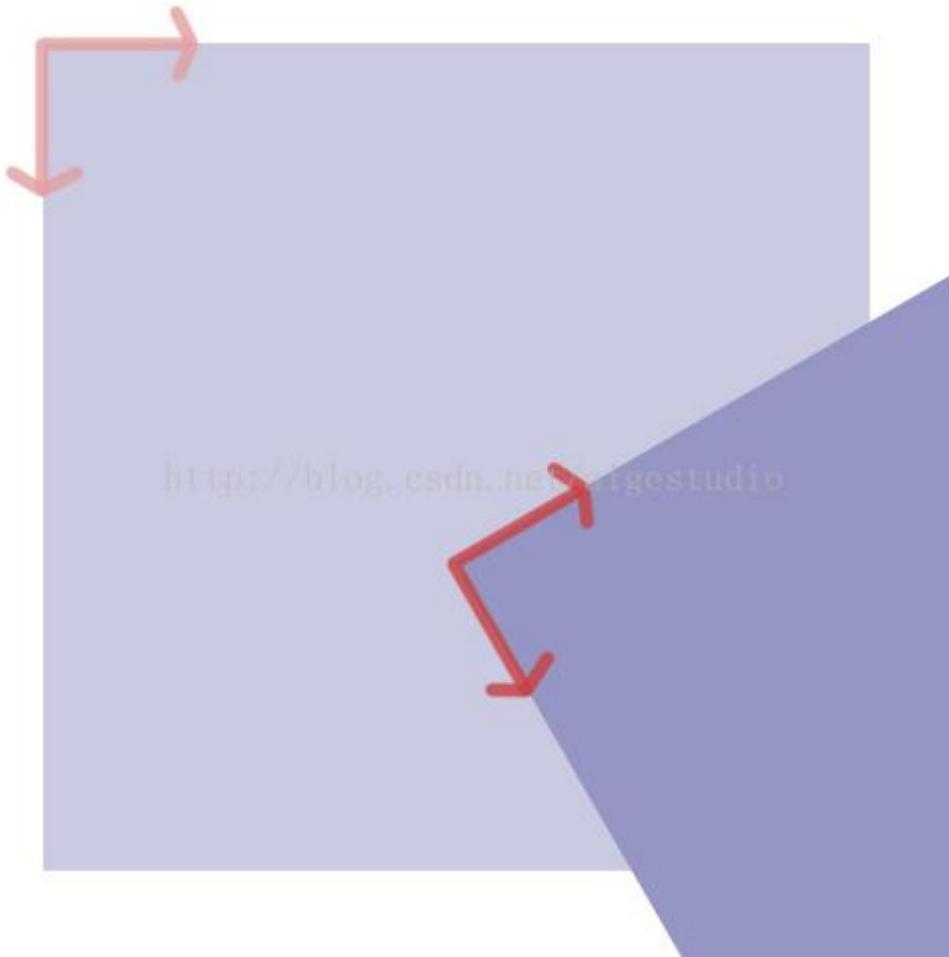
我们将画布平移了  $ccx$  和  $ccy$  个单位其实就是在让画布的左上端原点远我们的中心圆圆心重合：



[java] view plaincopyprint?

```
1. canvas.rotate(-30);
```

向左旋转画布 30 度:



这里有一点非常非常地重要！画布的平移旋转同样也会影响画布的自身坐标！如上图，我们看到画布的坐标也跟着旋转了 30 度！我们正是利用了这一点巧妙地在画图而避免繁杂的坐标计算！！

同理我们可以绘制出其他三个小圆：

[java] view plaincopyprint?

```
1. public class MultiCricleView extends View {  
2.     private static final float STROKE_WIDTH = 1F / 256F, // 描边宽度占比  
3.             SPACE = 1F / 64F, // 大圆小圆线段两端间隔占比  
4.             LINE_LENGTH = 3F / 32F, // 线段长度占比  
5.             CRICLE_LARGER_RADIUS = 3F / 32F, // 大圆半径  
6.             CRICLE_SMALL_RADIUS = 5F / 64F, // 小圆半径  
7.             ARC_RADIUS = 1F / 8F, // 弧半径  
8.             ARC_TEXT_RADIUS = 5F / 32F; // 弧围绕文字半径  
9.  
10.    private Paint strokePaint; // 描边画笔  
11.
```

```
12.     private int size;// 控件边长
13.
14.     private float strokeWidth;// 描边宽度
15.     private float ccX, ccY;// 中心圆圆心坐标
16.     private float largeCricleRadius, smallCricleRadius;// 大圆半径和小圆半径
17.     private float lineLength;// 线段长度
18.     private float space;// 大圆小圆线段两端间隔
19.
20.     private enum Type {
21.         LARGER, SMALL
22.     }
23.
24.     public MultiCricleView(Context context, AttributeSet attrs) {
25.         super(context, attrs);
26.
27.         // 初始化画笔
28.         initPaint(context);
29.     }
30.
31. /**
32. * 初始化画笔
33. *
34. * @param context
35. *          Fuck
36. */
37. private void initPaint(Context context) {
38.     /*
39.      * 初始化描边画笔
40.      */
41.     strokePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);

42.     strokePaint.setStyle(Paint.Style.STROKE);
43.     strokePaint.setColor(Color.WHITE);
44.     strokePaint.setStrokeCap(Paint.Cap.ROUND);
45. }
46.
47. @Override
48. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

49.     // 强制长宽一致
50.     super.onMeasure(widthMeasureSpec, widthMeasureSpec);
51. }
52.
53. @Override
```

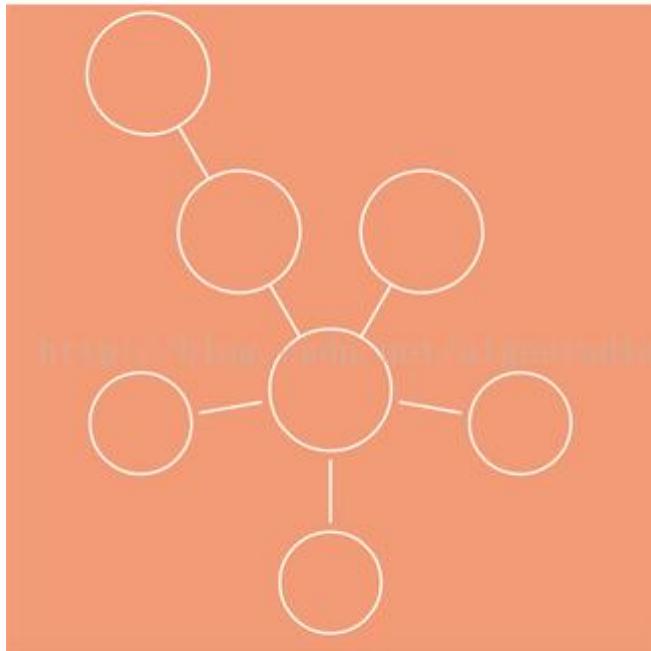
```
54.     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
55.         // 获取控件边长
56.         size = w;
57.
58.         // 参数计算
59.         calculation();
60.     }
61.
62.     /*
63.      * 参数计算
64.      */
65.     private void calculation() {
66.         // 计算描边宽度
67.         strokeWidth = STROKE_WIDTH * size;
68.
69.         // 计算大圆半径
70.         largeCricleRadius = size * CRICLE_LARGER_RADIUS;
71.
72.         // 计算小圆半径
73.         smallCricleRadius = size * CRICLE_SMALL_RADIUS;
74.
75.         // 计算线段长度
76.         lineLength = size * LINE_LENGTH;
77.
78.         // 计算大圆小圆线段两端间隔
79.         space = size * SPACE;
80.
81.         // 计算中心圆圆心坐标
82.         ccX = size / 2;
83.         ccY = size / 2 + size * CRICLE_LARGER_RADIUS;
84.
85.         // 设置参数
86.         setPara();
87.     }
88.
89.     /**
90.      * 设置参数
91.      */
92.     private void setPara() {
93.         // 设置描边宽度
94.         strokePaint.setStrokeWidth(strokeWidth);
95.     }
96.
97.     @Override
```

```
98.     protected void onDraw(Canvas canvas) {
99.         // 绘制背景
100.        canvas.drawColor(0xFFFF29B7);
101.
102.        // 绘制中心圆
103.        canvas.drawCircle(ccX, ccY, largeCricleRadius, strokePaint);
104.
105.        // 绘制左上方图形
106.        drawTopLeft(canvas);
107.
108.        // 绘制右上方图形
109.        drawTopRight(canvas);
110.
111.        // 绘制左下方图形
112.        drawBottomLeft(canvas);
113.
114.        // 绘制下方图形
115.        drawBottom(canvas);
116.
117.        // 绘制右下方图形
118.        drawBottomRight(canvas);
119.    }
120.
121.    /**
122.     * 绘制左上方图形
123.     *
124.     * @param canvas
125.     */
126.    private void drawTopLeft(Canvas canvas) {
127.        // 锁定画布
128.        canvas.save();
129.
130.        // 平移和旋转画布
131.        canvas.translate(ccX, ccY);
132.        canvas.rotate(-30);
133.
134.        // 依次画: 线-圈-线-圈
135.        canvas.drawLine(0, -largeCricleRadius, 0, -lineLength * 2, strokePaint);
136.        canvas.drawCircle(0, -lineLength * 3, largeCricleRadius, strokePaint);
137.        canvas.drawLine(0, -largeCricleRadius * 4, 0, -lineLength * 5, strokePaint);
```

```
138.         canvas.drawCircle(0, -lineLength * 6, largeCricleRadiu, strokePaint
    );
139.
140.         // 释放画布
141.         canvas.restore();
142.     }
143.
144.     /**
145.      * 绘制右上方图形
146.      *
147.      * @param canvas
148.      */
149.     private void drawTopRight(Canvas canvas) {
150.         // 锁定画布
151.         canvas.save();
152.
153.         // 平移和旋转画布
154.         canvas.translate(ccX, ccY);
155.         canvas.rotate(30);
156.
157.         // 依次画: 线-圈
158.         canvas.drawLine(0, -largeCricleRadiu, 0, -lineLength * 2, strokePai
nt);
159.         canvas.drawCircle(0, -lineLength * 3, largeCricleRadiu, strokePaint
    );
160.
161.         // 释放画布
162.         canvas.restore();
163.     }
164.
165.     private void drawBottomLeft(Canvas canvas) {
166.         // 锁定画布
167.         canvas.save();
168.
169.         // 平移和旋转画布
170.         canvas.translate(ccX, ccY);
171.         canvas.rotate(-100);
172.
173.         // 依次画: (间隔)线(间隔)-圈
174.         canvas.drawLine(0, -largeCricleRadiu - space, 0, -lineLength * 2 -
space, strokePaint);
175.         canvas.drawCircle(0, -lineLength * 2 - smallCricleRadiu - space * 2
, smallCricleRadiu, strokePaint);
176.
```

```
177.         // 释放画布
178.         canvas.restore();
179.     }
180.
181.     private void drawBottom(Canvas canvas) {
182.         // 锁定画布
183.         canvas.save();
184.
185.         // 平移和旋转画布
186.         canvas.translate(ccX, ccY);
187.         canvas.rotate(180);
188.
189.         // 依次画: (间隔)线(间隔)-圈
190.         canvas.drawLine(0, -largeCricleRadius - space, 0, -lineLength * 2 -
space, strokePaint);
191.         canvas.drawCircle(0, -lineLength * 2 - smallCricleRadius - space * 2
, smallCricleRadius, strokePaint);
192.
193.         // 释放画布
194.         canvas.restore();
195.     }
196.
197.     private void drawBottomRight(Canvas canvas) {
198.         // 锁定画布
199.         canvas.save();
200.
201.         // 平移和旋转画布
202.         canvas.translate(ccX, ccY);
203.         canvas.rotate(100);
204.
205.         // 依次画: (间隔)线(间隔)-圈
206.         canvas.drawLine(0, -largeCricleRadius - space, 0, -lineLength * 2 -
space, strokePaint);
207.         canvas.drawCircle(0, -lineLength * 2 - smallCricleRadius - space * 2
, smallCricleRadius, strokePaint);
208.
209.         // 释放画布
210.         canvas.restore();
211.     }
212. }
```

大家可以看到，每一次绘制我都锁定了画布并平移旋转以调整画布的原点坐标极大程度地方便我们计算。效果如下：



上面的代码会有巨量的重复代码，正如上面我说，这个图形我们是画死的，在没讲完 View 的测绘和 ViewGroup 之前我们不会做任何一个完整的控件，So~~~~同时也是为了方便大家容易理解这玩意是怎么画的，我也不对重复的方法做进一步封装了，不过在做项目的时候切忌大量的重复代码。

我们再给这些圈圈里加些文字，文字的中点很明显就是这些圈圈的圆心对吧，1/4 中我说过如何把文字画到中心的？

[java] view plain copy print?

```
1. public class MultiCricleView extends View {  
2.     private static final float STROKE_WIDTH = 1F / 256F, // 描边宽度占比  
3.             SPACE = 1F / 64F, // 大圆小圆线段两端间隔占比  
4.             LINE_LENGTH = 3F / 32F, // 线段长度占比  
5.             CRICLE_LARGER_RADIUS = 3F / 32F, // 大圆半径  
6.             CRICLE_SMALLER_RADIUS = 5F / 64F, // 小圆半径  
7.             ARC_RADIUS = 1F / 8F, // 弧半径  
8.             ARC_TEXT_RADIUS = 5F / 32F; // 弧围绕文字半径  
9.
```

```
10.     private Paint strokePaint, textPaint;// 描边画笔和文字画笔
11.
12.     private int size;// 控件边长
13.
14.     private float strokeWidth;// 描边宽度
15.     private float ccX, ccY;// 中心圆圆心坐标
16.     private float largeCricleRadius, smallCricleRadius;// 大圆半径和小圆半径
17.     private float lineLength;// 线段长度
18.     private float space;// 大圆小圆线段两端间隔
19.     private float textOffsetY;// 文本的Y轴偏移值
20.
21.     public MultiCricleView(Context context, AttributeSet attrs) {
22.         super(context, attrs);
23.
24.         // 初始化画笔
25.         initPaint(context);
26.     }
27.
28. /**
29. * 初始化画笔
30. *
31. * @param context
32. *      Fuck
33. */
34. private void initPaint(Context context) {
35.     /*
36.      * 初始化描边画笔
37.      */
38.     strokePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);

39.     strokePaint.setStyle(Paint.Style.STROKE);
40.     strokePaint.setColor(Color.WHITE);
41.     strokePaint.setStrokeCap(Paint.Cap.ROUND);
42.
43.     /*
44.      * 初始化文字画笔
45.      */
46.     textPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG
| Paint.SUBPIXEL_TEXT_FLAG);
47.     textPaint.setColor(Color.WHITE);
48.     textPaint.setTextSize(30);
49.     textPaint.setTextAlign(Paint.Align.CENTER);
50.
51.     textOffsetY = (textPaint.descent() + textPaint.ascent()) / 2;
```

```
52.     }
53.
54.     @Override
55.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
56.
57.         // 强制长宽一致
58.         super.onMeasure(widthMeasureSpec, widthMeasureSpec);
59.     }
60.
61.     @Override
62.     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
63.         // 获取控件边长
64.         size = w;
65.
66.         // 参数计算
67.         calculation();
68.     }
69.
70.     /*
71.      * 参数计算
72.      */
73.     private void calculation() {
74.         // 计算描边宽度
75.         strokeWidth = STROKE_WIDTH * size;
76.
77.         // 计算大圆半径
78.         largeCricleRadius = size * CRICLE_LARGER_RADIUS;
79.
80.         // 计算小圆半径
81.         smallCricleRadius = size * CRICLE_SMALL_RADIUS;
82.
83.         // 计算线段长度
84.         lineLength = size * LINE_LENGTH;
85.
86.         // 计算大圆小圆线段两端间隔
87.         space = size * SPACE;
88.
89.         // 计算中心圆圆心坐标
90.         ccX = size / 2;
91.         ccY = size / 2 + size * CRICLE_LARGER_RADIUS;
92.
93.         // 设置参数
94.         setPara();
95.     }
```

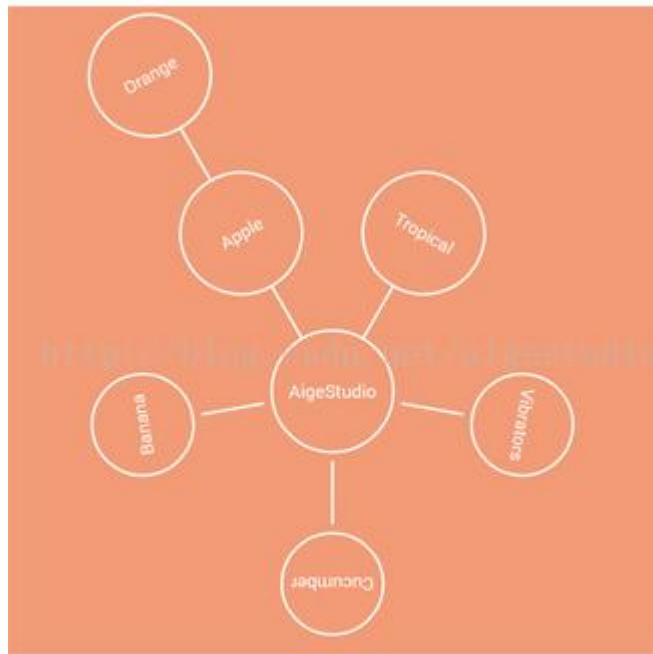
```
95.  
96.     /**  
97.      * 设置参数  
98.      */  
99.     private void setPara() {  
100.         // 设置描边宽度  
101.         strokePaint.setStrokeWidth(strokeWidth);  
102.     }  
103.  
104.     @Override  
105.     protected void onDraw(Canvas canvas) {  
106.         // 绘制背景  
107.         canvas.drawColor(0xFFFF29B7);  
108.  
109.         // 绘制中心圆  
110.         canvas.drawCircle(ccX, ccY, largeCricleRadius, strokePaint);  
111.         canvas.drawText("AigeStudio", ccX, ccY - textOffsetY, textPaint);  
112.  
113.         // 绘制左上方图形  
114.         drawTopLeft(canvas);  
115.  
116.         // 绘制右上方图形  
117.         drawTopRight(canvas);  
118.  
119.         // 绘制左下方图形  
120.         drawBottomLeft(canvas);  
121.  
122.         // 绘制下方图形  
123.         drawBottom(canvas);  
124.  
125.         // 绘制右下方图形  
126.         drawBottomRight(canvas);  
127.     }  
128.  
129.     /**  
130.      * 绘制左上方图形  
131.      *  
132.      * @param canvas  
133.      */  
134.     private void drawTopLeft(Canvas canvas) {  
135.         // 锁定画布  
136.         canvas.save();  
137.  
138.         // 平移和旋转画布
```

```
139.         canvas.translate(ccX, ccY);
140.         canvas.rotate(-30);
141.
142.         // 依次画: 线-圈-线-圈
143.         canvas.drawLine(0, -largeCricleRadius, 0, -lineLength * 2, strokePaint);
144.         canvas.drawCircle(0, -lineLength * 3, largeCricleRadius, strokePaint);
145.         canvas.drawText("Apple", 0, -lineLength * 3 - textOffsetY, textPaint);
146.
147.         canvas.drawLine(0, -largeCricleRadius * 4, 0, -lineLength * 5, strokePaint);
148.         canvas.drawCircle(0, -lineLength * 6, largeCricleRadius, strokePaint);
149.         canvas.drawText("Orange", 0, -lineLength * 6 - textOffsetY, textPaint);
150.
151.         // 释放画布
152.         canvas.restore();
153.     }
154.
155.     /**
156.      * 绘制右上方图形
157.      *
158.      * @param canvas
159.      */
160.     private void drawTopRight(Canvas canvas) {
161.         float cricleY = -lineLength * 3;
162.
163.         // 锁定画布
164.         canvas.save();
165.
166.         // 平移和旋转画布
167.         canvas.translate(ccX, ccY);
168.         canvas.rotate(30);
169.
170.         // 依次画: 线-圈
171.         canvas.drawLine(0, -largeCricleRadius, 0, -lineLength * 2, strokePaint);
172.         canvas.drawCircle(0, cricleY, largeCricleRadius, strokePaint);
173.         canvas.drawText("Tropical", 0, cricleY - textOffsetY, textPaint);
174.
175.         // 释放画布
```

```
176.         canvas.restore();
177.     }
178.
179.     private void drawBottomLeft(Canvas canvas) {
180.         float lineYS = -largeCricleRadius - space, lineYE = -lineLength * 2
181.             - space, cricleY = -lineLength * 2 - smallCricleRadius - space * 2;
182.
183.         // 锁定画布
184.         canvas.save();
185.
186.         // 平移和旋转画布
187.         canvas.translate(ccX, ccY);
188.         canvas.rotate(-100);
189.
190.         // 依次画: (间隔)线(间隔)-圈
191.         canvas.drawLine(0, lineYS, 0, lineYE, strokePaint);
192.         canvas.drawCircle(0, cricleY, smallCricleRadius, strokePaint);
193.         canvas.drawText("Banana", 0, cricleY - textOffsetY, textPaint);
194.
195.         // 释放画布
196.         canvas.restore();
197.
198.     private void drawBottom(Canvas canvas) {
199.         float lineYS = -largeCricleRadius - space, lineYE = -lineLength * 2
200.             - space, cricleY = -lineLength * 2 - smallCricleRadius - space * 2;
201.
202.         // 锁定画布
203.         canvas.save();
204.
205.         // 平移和旋转画布
206.         canvas.translate(ccX, ccY);
207.         canvas.rotate(180);
208.
209.         // 依次画: (间隔)线(间隔)-圈
210.         canvas.drawLine(0, lineYS, 0, lineYE, strokePaint);
211.         canvas.drawCircle(0, cricleY, smallCricleRadius, strokePaint);
212.         canvas.drawText("Cucumber", 0, cricleY - textOffsetY, textPaint);
213.
214.         // 释放画布
215.         canvas.restore();
216.
217.     private void drawBottomRight(Canvas canvas) {
```

```
218.         float lineYS = -largeCricleRadius - space, lineYE = -lineLength * 2
219.         - space, cricleY = -lineLength * 2 - smallCricleRadius - space * 2;
220.
221.         // 锁定画布
222.         canvas.save();
223.
224.         // 平移和旋转画布
225.         canvas.translate(ccX, ccY);
226.         canvas.rotate(100);
227.
228.         // 依次画: (间隔)线(间隔)-圈
229.         canvas.drawLine(0, lineYS, 0, lineYE, strokePaint);
230.         canvas.drawCircle(0, cricleY, smallCricleRadius, strokePaint);
231.         canvas.drawText("Vibrators", 0, cricleY - textOffsetY, textPaint);
232.
233.         // 释放画布
234.     }
235. }
```

That's so easy right?



稍微有点难的是右上方的那个半回扇形，扇形的中心应该是与右上方的圆心重合的对吧，那我们在画右上方图形的时候一起画不就是了？

[java] view plaincopyprint?

```
1. /**
2.  * 绘制右上方图形
3. *
4. * @param canvas
5. */
6. private void drawTopRight(Canvas canvas) {
7.     float cricleY = -lineLength * 3;
8.
9.     // 锁定画布
10.    canvas.save();
11.
12.    // 平移和旋转画布
```

```
13.     canvas.translate(ccX, ccY);
14.     canvas.rotate(30);
15.
16.     // 依次画: 线-圈
17.     canvas.drawLine(0, -largeCricleRadiu, 0, -lineLength * 2, strokePaint);

18.     canvas.drawCircle(0, cricleY, largeCricleRadiu, strokePaint);
19.     canvas.drawText("Tropical", 0, cricleY - textOffsetY, textPaint);
20.
21.     // 画弧形
22.     drawTopRightArc(canvas, cricleY);
23.
24.     // 释放画布
25.     canvas.restore();
26. }
27.
28. /**
29. * 绘制右上角画弧形
30. *
31. * @param canvas
32. * @param cricleY
33. */
34. private void drawTopRightArc(Canvas canvas, float cricleY) {
35.     canvas.save();
36.
37.     canvas.translate(0, cricleY);
38.     canvas.rotate(-30);
39.
40.     float arcRadiu = size * ARC_RADIUS;
41.
42.     RectF oval = new RectF(-arcRadiu, -arcRadiu, arcRadiu, arcRadiu);
43.
44.     arcPaint.setStyle(Paint.Style.FILL);
45.     arcPaint.setColor(0x55EC6941);
46.     canvas.drawArc(oval, -22.5F, -135, true, arcPaint);
47.
48.     arcPaint.setStyle(Paint.Style.STROKE);
49.     arcPaint.setColor(Color.WHITE);
50.     canvas.drawArc(oval, -22.5F, -135, false, arcPaint);
51.
52.     canvas.restore();
53. }
```

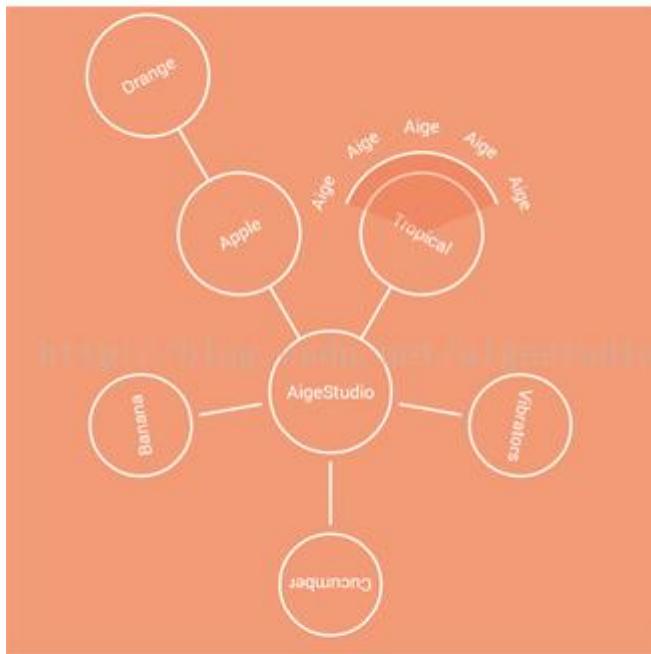
代码逻辑不复杂，大家很容易能看懂……注释什么的我就不写了……扇形上的文本如果大家理解了画布的变换很容易实现，首先在上面我们绘制扇形的时候已经将画布原点与右上圆心重合了对吧，这时我们再把画布向左旋转  $\text{扇形弧度}/2$  个度数是不是就可以让画布的坐标与扇形的右边重合了呢？那第一个文字的坐标就是[0,负的文字弧形半径]，第二个文字坐标只需转画布过  $\text{扇形弧度}/4$  个弧度以此类推可以画出五个文字：

[java] view plaincopyprint?

```
1. /**
2.  * 绘制右上角画弧形
3. *
4. * @param canvas
5. * @param cricleY
6. */
7. private void drawTopRightArc(Canvas canvas, float cricleY) {
8.     canvas.save();
9.
10.    canvas.translate(0, cricleY);
11.    canvas.rotate(-30);
12.
13.    float arcRadius = size * ARC_RADIUS;
14.    RectF oval = new RectF(-arcRadius, -arcRadius, arcRadius, arcRadius);
15.    arcPaint.setStyle(Paint.Style.FILL);
16.    arcPaint.setColor(0x55EC6941);
17.    canvas.drawArc(oval, -22.5F, -135, true, arcPaint);
18.    arcPaint.setStyle(Paint.Style.STROKE);
19.    arcPaint.setColor(Color.WHITE);
20.    canvas.drawArc(oval, -22.5F, -135, false, arcPaint);
21.
22.    float arcTextRadius = size * ARC_TEXT_RADIUS;
23.
24.    canvas.save();
25.    // 把画布旋转到扇形左端的方向
26.    canvas.rotate(-135F / 2F);
27.
28.    /*
29.     * 每隔 33.75 度角画一次文本
30.     */
31.    for (float i = 0; i < 5 * 33.75F; i += 33.75F) {
32.        canvas.save();
33.        canvas.rotate(i);
34.
35.        canvas.drawText("Aige", 0, -arcTextRadius, textPaint);
36.
37.        canvas.restore();
38.    }
```

```
39.  
40.    canvas.restore();  
41.  
42.    canvas.restore();  
43. }
```

对吧？看看效果：



好了，正如我所说，这只是一个单纯地画，而且此类奇葩的玩意难以真正做成一个控件去复用除非真的是公事公办，但是，我们依然可以尝试把它做成一个独立的控件，这在我们学写了如何测绘 View 和 ViewGroup 之后对你来说一定是小 case。

上面我们的最终效果有一点是不对的，大家发现文字 TMD 居然都旋转了 --，那有木有方法让文字保持水平呢？其实答案我已经告诉你。自己去发掘吧！

## 5. 自定义控件其实很简单(5)

在(4)中我们结束了全部的 `Paint` 方法学习还略带地说了下 `Matrix` 的简单用法，这两节呢，我们将甩掉第二个陌生又熟悉的情妇：`Canvas`。`Canvas` 从我们该系列教程的第一节起就喋喋不休个没完没了，几乎每个 `View` 都扯到了它，就像我之前说的那样，自定义控件的关键一步就是如何去绘制控件，绘制说白了就是画，既然要画那么笔和纸是必须的，`Canvas` 就是 `Android` 给我们的纸，弥足轻重，它决定了我们能画什么：

|      |                                                                                                                                                                                                                                                                                                                                                                                         |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void | <code>drawARGB(int a, int r, int g, int b)</code><br>Fill the entire canvas' bitmap (restricted to the current clip) with the specified ARGB color.                                                                                                                                                                                                                                     |
| void | <code>drawArc(RectF oval, float startAngle, float sweepAngle, boolean useCenter, Paint paint)</code><br>Draw the specified arc, which will be scaled to fit inside the specified oval.                                                                                                                                                                                                  |
| void | <code>drawArc(float left, float top, float right, float bottom, float startAngle, float sweepAngle, boolean useCenter, Paint paint)</code><br>Draw the specified arc, which will be scaled to fit inside the specified oval.                                                                                                                                                            |
| void | <code>drawBitmap(int[] colors, int offset, int stride, float x, float y, int width, int height, boolean hasAlpha, boolean filter, boolean reuseCache)</code><br><i>This method was deprecated in API level 21. Usage with a hardware accelerated canvas requires a copy of the bitmap, and allows the application to more explicitly control the lifetime and copies of pixel data.</i> |
| void | <code>drawBitmap(Bitmap bitmap, Matrix matrix, Paint paint)</code><br>Draw the bitmap using the specified matrix.                                                                                                                                                                                                                                                                       |
| void | <code>drawBitmap(int[] colors, int offset, int stride, int x, int y, int width, int height, boolean hasAlpha, boolean filter, boolean reuseCache)</code><br><i>This method was deprecated in API level 21. Usage with a hardware accelerated canvas requires a copy of the bitmap, and allows the application to more explicitly control the lifetime and copies of pixel data.</i>     |
| void | <code>drawBitmap(Bitmap bitmap, Rect src, RectF dst, Paint paint)</code><br>Draw the specified bitmap, scaling/translating automatically to fill the destination rectangle.                                                                                                                                                                                                             |
| void | <code>drawBitmap(Bitmap bitmap, float left, float top, Paint paint)</code><br>Draw the specified bitmap, with its top/left corner at (x,y), using the specified paint, translating it if necessary.                                                                                                                                                                                     |
| void | <code>drawBitmap(Bitmap bitmap, Rect src, Rect dst, Paint paint)</code><br>Draw the specified bitmap, scaling/translating automatically to fill the destination rectangle.                                                                                                                                                                                                              |
| void | <code>drawBitmapMesh(Bitmap bitmap, int meshWidth, int meshHeight, float[] verts, int vertOffset, int[] texs, int texOffset, Paint paint)</code><br>Draw the bitmap through the mesh, where mesh vertices are evenly distributed across the rectangle.                                                                                                                                  |
| void | <code>drawCircle(float cx, float cy, float radius, Paint paint)</code><br>Draw the specified circle using the specified paint.                                                                                                                                                                                                                                                          |
| void | <code>drawColor(int color)</code><br>Fill the entire canvas' bitmap (restricted to the current clip) with the specified color, using the specified paint.                                                                                                                                                                                                                               |
| void | <code>drawColor(int color, PorterDuff.Mode mode)</code><br>Fill the entire canvas' bitmap (restricted to the current clip) with the specified color and mode.                                                                                                                                                                                                                           |
| void | <code>drawLine(float startX, float startY, float stopX, float stopY, Paint paint)</code><br>Draw a line segment with the specified start and stop x,y coordinates, using the specified paint.                                                                                                                                                                                           |
| void | <code>drawLines(float[] pts, Paint paint)</code>                                                                                                                                                                                                                                                                                                                                        |
| void | <code>drawLines(float[] pts, int offset, int count, Paint paint)</code><br>Draw a series of lines.                                                                                                                                                                                                                                                                                      |
| void | <code>drawOval(float left, float top, float right, float bottom, Paint paint)</code><br>Draw the specified oval using the specified paint.                                                                                                                                                                                                                                              |
| void | <code>drawOval(RectF oval, Paint paint)</code><br>Draw the specified oval using the specified paint.                                                                                                                                                                                                                                                                                    |
| void | <code>drawPaint(Paint paint)</code><br>Fill the entire canvas' bitmap (restricted to the current clip) with the specified paint.                                                                                                                                                                                                                                                        |
| void | <code>drawPath(Path path, Paint paint)</code><br>Draw the specified path using the specified paint.                                                                                                                                                                                                                                                                                     |

上面所罗列出来的各种 `drawXXX` 方法就是 `Canvas` 中定义好的能画什么的方法(`drawPaint`除外)，除了各种基本型比如矩形圆形椭圆直曲线外 `Canvas` 也能直接让我们绘制各种图片以及颜色等等，但是 `Canvas` 真正屌的我觉得不是它能画些什么，而是对画布的各种活用，上一节最后的一个例子大家已经粗略见识了变换 `Canvas` 配合 `save` 和 `restore` 方法给我们绘制图形带来的极大便利，事实上 `Canvas` 的活用远不止此，在讲 `Canvas` 之前，我想先给大家说说 `Canvas` 中非常屌毛而且很有个性的一个方法：

[java] view plaincopyprint?

```
1. drawBitmapMesh(Bitmap bitmap, int meshWidth, int meshHeight, float[] verts,  
    int vertOffset, int[] colors, int colorOffset, Paint paint)
```

`drawBitmapMesh` 是个很屌毛的方法，为什么这样说呢？因为它可以对 `Bitmap` 做几乎任何改变，是的，你没听错，是任何，几乎无所不能，这个屌毛方法我曾一度怀疑谷歌那些逗比为何将它屈尊在 `Canvas` 下，因为它对 `Bitmap` 的处理实在在强大了。上一节我们在讲到 `Matrix` 的时候说过 `Matrix` 可以对我们的图像做多种变换，实际上 `drawBitmapMesh` 也可以，只不过需要一点计算，比如我们可以使用 `drawBitmapMesh` 来模拟错切 `skew` 的效果：



实现过程也非常非常简单：

[java] view plaincopyprint?

```
1. public class BitmapMeshView extends View {  
2.     private static final int WIDTH = 19;// 横向分割成的网格数量  
3.     private static final int HEIGHT = 19;// 纵向分割成的网格数量  
4.     private static final int COUNT = (WIDTH + 1) * (HEIGHT + 1); // 横纵向网格  
    交织产生的点数量  
5.  
6.     private Bitmap mBitmap; // 位图资源  
7.  
8.     private float[] verts; // 交点的坐标数组  
9.
```

```
10.     public BitmapMeshView(Context context, AttributeSet attrs) {
11.         super(context, attrs);
12.
13.         // 获取位图资源
14.         mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.gr
15.             il);
16.
17.         // 实例化数组
18.         verts = new float[COUNT * 2];
19.
20.         /*
21.          * 生成各个交点坐标
22.          */
23.         int index = 0;
24.         float multiple = mBitmap.getWidth();
25.         for (int y = 0; y <= HEIGHT; y++) {
26.             float fy = mBitmap.getHeight() * y / HEIGHT;
27.             for (int x = 0; x <= WIDTH; x++) {
28.                 float fx = mBitmap.getWidth() * x / WIDTH + ((HEIGHT - y) *
29.                     1.0F / HEIGHT * multiple);
30.                 setXY(fx, fy, index);
31.                 index += 1;
32.             }
33.         }
34.         /**
35.          * 将计算后的交点坐标存入数组
36.          *
37.          * @param fx
38.          *          x 坐标
39.          * @param fy
40.          *          y 坐标
41.          * @param index
42.          *          标识值
43.          */
44.         private void setXY(float fx, float fy, int index) {
45.             verts[index * 2 + 0] = fx;
46.             verts[index * 2 + 1] = fy;
47.         }
48.
49.         @Override
50.         protected void onDraw(Canvas canvas) {
51.             // 绘制网格位图
```

```
52.         canvas.drawBitmapMesh(mBitmap, WIDTH, HEIGHT, verts, 0, null, 0, null  
1);  
53.     }  
54. }
```

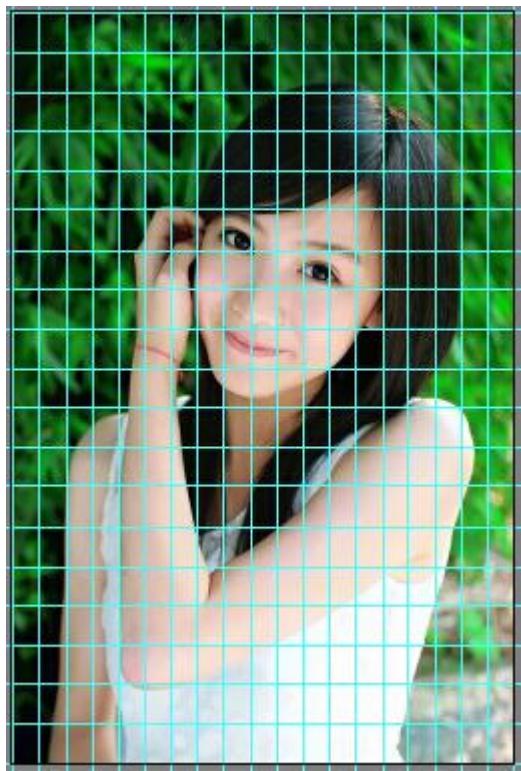
其他的我就不说了，关键代码就一段：

[java] view plaincopyprint?

```
1. /*  
2.  * 生成各个交点坐标  
3. */  
4. int index = 0;  
5. float multiple = mBitmap.getWidth();  
6. for (int y = 0; y <= HEIGHT; y++) {  
7.     float fy = mBitmap.getHeight() * y / HEIGHT;  
8.     for (int x = 0; x <= WIDTH; x++) {  
9.         float fx = mBitmap.getWidth() * x / WIDTH + ((HEIGHT - y) * 1.0F / H  
EIGHT * multiple);  
10.        setXY(fx, fy, index);  
11.        index += 1;  
12.    }  
13. }
```

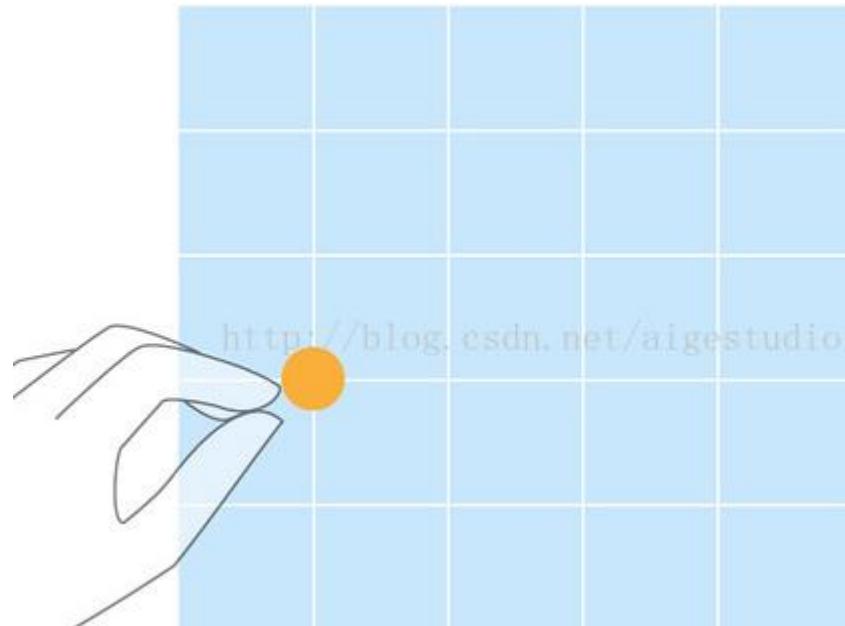
这段代码生成了 200 个点的坐标数据全部存入 `verts` 数组，`verts` 数组中，偶数位表示 x 轴坐标，奇数位表示 y 轴坐标，最终 `verts` 数组中的元素构成为：

[x,y,x,y,x,y,x,y,x,y,x,y,x,y,x,y,x,y.....]共  $200 \times 2 = 400$  个元素，为什么是 400 个？如果你不是蠢 13 的话一定能计算过来。那么现在我们一定很好奇，`drawBitmapMesh` 到底是个什么个意思呢？，其实 `drawBitmapMesh` 的原理灰常简单，它按照 `meshWidth` 和 `meshHeight` 这两个参数的值将我们的图片划分成一定数量的网格，比如上面我们传入的 `meshWidth` 和 `meshHeight` 均为 19，意思就是把整个图片横向分成 19 份：

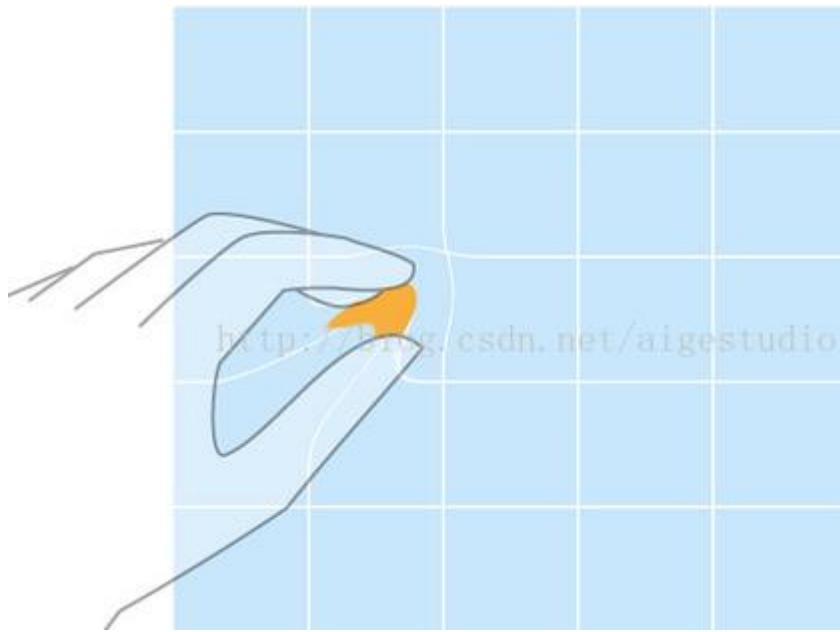


横向 19 个网格那么意味着横向分别有 20 条分割线对吧，这 20 条分割线交织又构成了  $20 * 20$  个交织点

每个点又有 x、y 两个坐标……而 `drawBitmapMesh` 的 `verts` 参数就是存储这些坐标值的，不过是图像变化后的坐标值，什么意思？说起来有点抽象，借用国外大神的两幅图来理解：



如上图，黄色的点是使用 `mesh` 分割图像后分割线的交点之一，而 `drawBitmapMesh` 的原理就是通过移动这些点来改变图像：

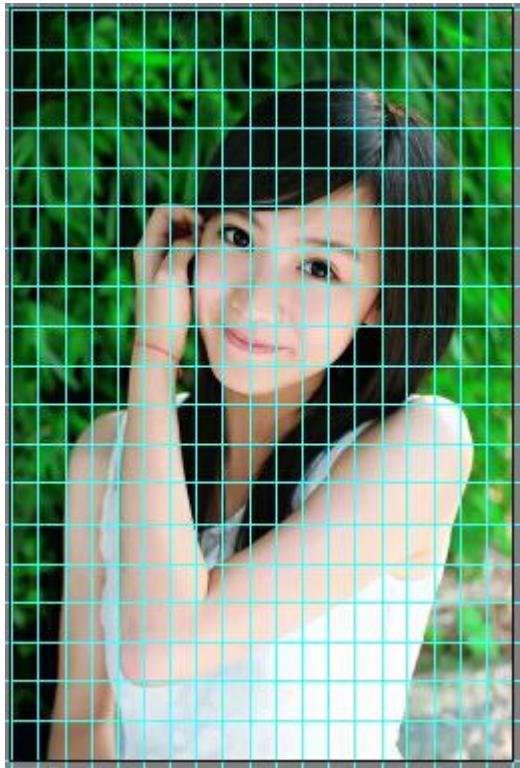


如上图，移动黄色的点后，图像被扭曲改变，你能想象在一幅刚画好的油画上有手指尖一抹的感觉么？油画未干，手指抹过的地方必将被抹得一塌糊涂，`drawBitmapMesh` 的原理就与之类似，只不过我们不常只改变一点，而是改变大量的点来达到效果，而参数 `verts` 则存储了改变后的坐标，`drawBitmapMesh` 依据这些坐标来改变图像，如果上面的代码中我们不将每行的 `x` 轴坐标进行平移而是单纯地计算了一下均分后的各点坐标：

[java] view plaincopyprint?

```
1. /*
2.  * 生成各个交点坐标
3. */
4. int index = 0;
5. //     float multiple = mBitmap.getWidth();
6. for (int y = 0; y <= HEIGHT; y++) {
7.     float fy = mBitmap.getHeight() * y / HEIGHT;
8.     for (int x = 0; x <= WIDTH; x++) {
9.         float fx = mBitmap.getWidth() * x / WIDTH;
10. //             float fx = mBitmap.getWidth() * x / WIDTH + ((HEIGHT - y) *
11. //                 1.0F / HEIGHT * multiple);
12.         setXY(fx, fy, index);
13.         index += 1;
14. }
```

你会发现图像没有任何改变，为什么呢？因为上面我们说过，`verts` 表示了图像变化后各点的坐标，而点坐标的变化是参照最原始均分后的坐标点，也就是图：



中的各个交织点，在此基础上形成变化，比如我们最开始的错切效果，原理很简单，我们这里把图像分成了横竖 20 条分割线（实际上错切变换只需要四个顶点即可，这里我只作点稍复杂的演示），我们只需将第一行的点 x 轴向上移动一定距离，而第二行的点移动的距离则比第一行点稍短，依次类推即可，每行点移动的距离我们通过

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. (HEIGHT - y) * 1.0F / HEIGHT * multiple
```

来计算，最终形成错切的效果

`drawBitmapMesh` 不能存储计算后点的值，每次调用 `drawBitmapMesh` 方法改变图像都是以基准点坐标为参考的，也就是说，不管你执行 `drawBitmapMesh` 方法几次，只要参数没改变，效果不累加。

`drawBitmapMesh` 可以做出很多很多的效果，比如类似放大镜的：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 生成各个交点坐标
3. */
4. int index = 0;
5. float multipleY = mBitmap.getHeight() / HEIGHT;
6. float multipleX = mBitmap.getWidth() / WIDTH;
7. for (int y = 0; y <= HEIGHT; y++) {
8.     float fy = multipleY * y;
```

```
9.     for (int x = 0; x <= WIDTH; x++) {
10.         float fx = multipleX * x;
11.
12.         setXY(fx, fy, index);
13.
14.         if (5 == y) {
15.             if (8 == x) {
16.                 setXY(fx - multipleX, fy - multipleY, index);
17.             }
18.             if (9 == x) {
19.                 setXY(fx + multipleX, fy - multipleY, index);
20.             }
21.         }
22.         if (6 == y) {
23.             if (8 == x) {
24.                 setXY(fx - multipleX, fy + multipleY, index);
25.             }
26.             if (9 == x) {
27.                 setXY(fx + multipleX, fy + multipleY, index);
28.             }
29.         }
30.
31.         index += 1;
32.     }
33. }
```

这时我们将图片眼睛附近的四个点外移到临近的四个点上，图像该区域就会被像放大一样：



太恶心了……我们借助另外一个例子来更好地理解 `drawBitmapMesh`，这个例子与 API DEMO 类似，我只是参考了国外大神的效果给他加上了一些标志点和位移线段来更好地展示 `drawBitmapMesh` 做了什么：

[java] view plaincopyprint?

```
1.  public class BitmapMeshView2 extends View {  
2.      private static final int WIDTH = 9, HEIGHT = 9;// 分割数  
3.      private static final int COUNT = (WIDTH + 1) * (HEIGHT + 1); // 交点数  
4.  
5.      private Bitmap mBitmap; // 位图对象  
6.  
7.      private float[] matrixOriganal = new float[COUNT * 2]; // 基准点坐标数组  
8.      private float[] matrixMoved = new float[COUNT * 2]; // 变换后点坐标数组  
9.  
10.     private float clickX, clickY; // 触摸屏幕时手指的 xy 坐标  
11.  
12.     private Paint origPaint, movePaint, linePaint; // 基准点、变换点和线段的绘制  
     Paint  
13.  
14.     public BitmapMeshView2(Context context, AttributeSet set) {  
15.         super(context, set);  
16.         setFocusable(true);  
17.  
18.         // 实例画笔并设置颜色  
19.         origPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
20.         origPaint.setColor(0x660000FF);  
21.         movePaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
22.         movePaint.setColor(0x99FF0000);  
23.         linePaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
24.         linePaint.setColor(0xFFFFFB00);  
25.  
26.         // 获取位图资源  
27.         mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.bt  
     );  
28.  
29.         // 初始化坐标数组  
30.         int index = 0;  
31.         for (int y = 0; y <= HEIGHT; y++) {  
32.             float fy = mBitmap.getHeight() * y / HEIGHT;  
33.  
34.             for (int x = 0; x <= WIDTH; x++) {  
35.                 float fx = mBitmap.getWidth() * x / WIDTH;  
36.                 setXY(matrixMoved, index, fx, fy);  
37.                 setXY(matrixOriganal, index, fx, fy);  
38.                 index += 1;  
39.             }  
40.         }  
41.     }  
42. }
```

```
42.  
43.     /**  
44.      * 设置坐标数组  
45.      *  
46.      * @param array  
47.      *          坐标数组  
48.      * @param index  
49.      *          标识值  
50.      * @param x  
51.      *          x 坐标  
52.      * @param y  
53.      *          y 坐标  
54.      */  
55.     private void setXY(float[] array, int index, float x, float y) {  
56.         array[index * 2 + 0] = x;  
57.         array[index * 2 + 1] = y;  
58.     }  
59.  
60.     @Override  
61.     protected void onDraw(Canvas canvas) {  
62.  
63.         // 绘制网格位图  
64.         canvas.drawBitmapMesh(mBitmap, WIDTH, HEIGHT, matrixMoved, 0, null,  
65.             0, null);  
66.         // 绘制参考元素  
67.         drawGuide(canvas);  
68.     }  
69.  
70.     /**  
71.      * 绘制参考元素  
72.      *  
73.      * @param canvas  
74.      *          画布  
75.      */  
76.     private void drawGuide(Canvas canvas) {  
77.         for (int i = 0; i < COUNT * 2; i += 2) {  
78.             float x = matrixOriganal[i + 0];  
79.             float y = matrixOriganal[i + 1];  
80.             canvas.drawCircle(x, y, 4, origPaint);  
81.  
82.             float x1 = matrixOriganal[i + 0];  
83.             float y1 = matrixOriganal[i + 1];  
84.             float x2 = matrixMoved[i + 0];
```

```
85.         float y2 = matrixMoved[i + 1];
86.         canvas.drawLine(x1, y1, x2, y2, origPaint);
87.     }
88.
89.     for (int i = 0; i < COUNT * 2; i += 2) {
90.         float x = matrixMoved[i + 0];
91.         float y = matrixMoved[i + 1];
92.         canvas.drawCircle(x, y, 4, movePaint);
93.     }
94.
95.     canvas.drawCircle(clickX, clickY, 6, linePaint);
96. }
97.
98. /**
99. * 计算变换数组坐标
100. */
101. private void smudge() {
102.     for (int i = 0; i < COUNT * 2; i += 2) {
103.
104.         float xOriginal = matrixOriganal[i + 0];
105.         float yOriginal = matrixOriganal[i + 1];
106.
107.         float dist_click_to_origin_x = clickX - xOriginal;
108.         float dist_click_to_origin_y = clickY - yOriginal;
109.
110.         float kv_kat = dist_click_to_origin_x * dist_click_to_origin_x
+ dist_click_to_origin_y * dist_click_to_origin_y;
111.
112.         float pull = (float) (1000000 / kv_kat / Math.sqrt(kv_kat));
113.
114.         if (pull >= 1) {
115.             matrixMoved[i + 0] = clickX;
116.             matrixMoved[i + 1] = clickY;
117.         } else {
118.             matrixMoved[i + 0] = xOriginal + dist_click_to_origin_x * p
ull;
119.             matrixMoved[i + 1] = yOriginal + dist_click_to_origin_y * p
ull;
120.         }
121.     }
122. }
123.
124. @Override
125. public boolean onTouchEvent(MotionEvent event) {
```

```
126.         clickX = event.getX();
127.         clickY = event.getY();
128.         smudge();
129.         invalidate();
130.         return true;
131.     }
132. }
```

运行后的效果如下：



大波妹子图上我们绘制了很多蓝色和红色的点，默认状态下，蓝色和红色的点是重合在一起的，两者间通过一线段连接，当我们手指在图片上移动时，会出现一个黄色的点，黄色的点代表我们当前的触摸点，而红色的点代表变换后的坐标点，蓝色的点代表基准坐标点：



可以看到越靠近触摸点的红点越向触摸点坍塌，红点表示当前变换后的点坐标，蓝点表示基准点的坐标，所有的变化都是参照蓝点进行的，这个例子可以很容易地理解

drawBitmapMesh:



大波妹子揉啊揉~~~~揉啊揉~~~~

drawBitmapMesh 参数中有个 `vertOffset`，该参数是 `verts` 数组的偏移值，意为从第一个元素开始才对位图就行变化，这些大家自己去尝试下吧，还有 `colors` 和 `colorOffset`，类似。

drawBitmapMesh 说实话真心很屌，但是计算复杂确是个鸡肋，这么屌的一个方法被埋没其实是由原因可循的，高不成低不就，如上所示，有些变换我们可以使用 `Matrix` 等其他方法简单实现，但是 drawBitmapMesh 就要通过一些列计算，太复杂。那真要做复杂的图形效

果呢，考虑到效率我们又会首选 OpenGL……这真是一个悲伤的故事……无论怎样，请记住这位烈士一样的方法…………总有用处的

好了，真的要开始搞 Canvas，开始搞了哦~~谁先上？

要学懂 Canvas 就要知道 Canvas 的本质是什么，那有益友就会说了，麻痹你不是扯过无数次 Canvas 是画布么，难道又不是了？是，Canvas 是画布，但是我们真的是在 Canvas 上画东西么？在前几节的一些例子中我们曾这样使用过 Canvas：

[java] view plaincopyprint?

```
1. Bitmap bitmap = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_8888);
2. Canvas canvas = new Canvas(bitmap);
3. canvas.drawColor(Color.RED);
```

也就是说将 Bitmap 注入到 Canvas 中，尔后 Canvas 所有的操作都会在这个 Bitmap 上进行，如果，此时我们的界面中有一个 ImageView，那么我们可以直接将绘制后的 Bitmap 显示出来：

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
2.     private ImageView ivMain;
3.
4.     @Override
5.     public void onCreate(Bundle savedInstanceState) {
6.         super.onCreate(savedInstanceState);
7.         setContentView(R.layout.activity_main);
8.
9.         ivMain = (ImageView) findViewById(R.id.main_iv);
10.
11.        Bitmap bitmap = Bitmap.createBitmap(100, 100, Bitmap.Config.ARGB_888
12. );
12.        Canvas canvas = new Canvas(bitmap);
13.        canvas.drawColor(Color.RED);
14.
15.        ivMain.setImageBitmap(bitmap);
16.    }
17. }
```

运行效果如图所示：



我们只是简单地填充了一块红色色块，色块的大小由 `bitmap` 决定，更确切地说，这个 `Canvas` 的大小是由 `bitmap` 决定的，类似的方法我们在前几节的例子中也不少用到，这里就不多说了。除了我们自己去 `new` 一个 `Canvas` 外，我们更常获得 `Canvas` 对象的地方是在 `View` 的：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     super.onDraw(canvas);
4. }
```

在这里通过 `onDraw` 方法的参数传递我们可以获取一个 `Canvas` 对象，好奇的同学一定很想知道这个 `Canvas` 对象是如何来的，跟我们自己 `new` 的有何区别。事实上两者区别不大，最终都是 `new` 过来的，只是 `onDraw` 方法传过来的 `Canvas` 对象拥有一些绘制的上下文关联。这一过程涉及到太多的源码，这里我只简单地提一下。在 `framework` 中，`Activity` 被创建时（更准确地说是在 `addView` 的时候）会同时创建一个叫做 `ViewRootImpl` 的对象，`ViewRootImpl` 是个很碉堡的类，它负责很多 `GUI` 的东西，包括我们常见的窗口显示、用户的输入输出等等，同时，它也负责 `Window` 跟 `WMS` 通信（`Window` 你可以想象是一个容器，里面包含着我们的一个 `Activity`，而 `AMS` 呢全称为 `Activity Manager Service`，顾名思义很好理解它的作用），当 `ViewRootImpl` 跟 `WMS` 建立通信注册了 `Window` 后就会发出第一次渲染 `View Hierarchy` 的请求，涉及到的方法均在 `ViewRootImpl` 下：`setView`、`requestLayout`、`scheduleTraversals` 等，大家有兴趣可以自己去搜罗看看，在 `performTraversals` 方法中 `ViewRootImpl` 就会去创建 `Surface`，而此后的渲染则可以通过 `Surface` 的 `lockCanvas` 方法获取 `Surface` 的 `Canvas` 来进行，然后遍历 `View Hierarchy` 把需要绘制的 `View` 通过 `Canvas` (`View.onDraw(Canvas canvas)`) 绘制到 `Surface` 上，绘制完成后解锁 (`Surface.unlockCanvasAndPost`) 让 `SurfaceFlinger` 将 `Surface` 绘制到屏幕上。我们 `onDraw(Canvas canvas)` 方法中传入的 `Canvas` 对象大致就是这么来的，说起简单，其实中间还有大量的过程被我省略了…………还是不扯为好，扯了讲通宵都讲不完。

上面我们概述了下 `onDraw` 参数列表中的 `Canvas` 对象是怎么来的，那么 `Canvas` 的实质是什么呢？我们通过追踪 `Canvas` 的两个构造方法可以发现两者的实现过程：

无参构造方法：

[java] view plaincopyprint?

```
1. /**
2. * Construct an empty raster canvas. Use setBitmap() to specify a bitmap to
```

```
3.     * draw into. The initial target density is {@link Bitmap#DENSITY_NONE};
4.     * this will typically be replaced when a target bitmap is set for the
5.     * canvas.
6. */
7. public Canvas() {
8.     if (!isHardwareAccelerated()) {
9.         // 0 means no native bitmap
10.        mNativeCanvas = initRaster(0);
11.        mFinalizer = new CanvasFinalizer(mNativeCanvas);
12.    } else {
13.        mFinalizer = null;
14.    }
15.}
```

含 Bitmap 对象作为参数的构造方法:

[java] view plaincopyprint?

```
1. /**
2.  * Construct a canvas with the specified bitmap to draw into. The bitmap
3.  * must be mutable.
4. *
5.  * <p>The initial target density of the canvas is the same as the given
6.  * bitmap's density.
7. *
8.  * @param bitmap Specifies a mutable bitmap for the canvas to draw into.
9. */
10. public Canvas(Bitmap bitmap) {
11.     if (!bitmap.isMutable()) {
12.         throw new IllegalStateException("Immutable bitmap passed to Canvas c
onstructor");
13.     }
14.     throwIfCannotDraw(bitmap);
15.     mNativeCanvas = initRaster(bitmap.ni());
16.     mFinalizer = new CanvasFinalizer(mNativeCanvas);
17.     mBitmap = bitmap;
18.     mDensity = bitmap.mDensity;
19. }
```

大家看到这两个构造方法我都把它的注释给 COPY 出来了，目的就是想告诉大家，虽然说无参的构造方法并没有传入 Bitmap 对象，但是 Android 依然建议（苛刻地说是要求）我们使用 Canvas 的 setBitmap()方法去为 Canvas 指定一个 Bitmap 对象！为什么 Canvas 非要一样 Bitmap 对象呢？原因很简单，Canvas 需要一个 Bitmap 对象来保存像素。Canvas 有大量的代码被封装并通过 jni 调用，事实上 Android 涉及图形图像处理的大量方法都是通过

jni 调用的，比如上面两个构造方法都调用了一个 `initRaster` 方法，这个方法的实现灰常简单：

[java] view plaincopyprint?

```
1. static SkCanvas* initRaster(JNIEnv* env, jobject, SkBitmap* bitmap) {
2.     if (bitmap) {
3.         return new SkCanvas(*bitmap);
4.     } else {
5.         // Create an empty bitmap device to prevent callers from crashing
6.         // if they attempt to draw into this canvas.
7.         SkBitmap emptyBitmap;
8.         return new SkCanvas(emptyBitmap);
9.     }
10. }
```

可以看到 `bitmap` 又被封装成了一个 `SkCanvas` 对象。上面我们曾说过，`onDraw` 中传来的 `Cnavas` 对象来自于 `ViewRootImpl` 的 `Surface`，当调用 `Surface.lockCanvas` 时会从图像缓存队列中取出一个可用缓存，把当前 `Posted Buffer` 的内容 `COPY` 到新缓存中然后加锁该缓存区域并设置为 `Locked Buffer`。此时会根据新缓存的内存地址构建一个 `SkBitmap` 并将该 `SkBitmap` 设置到 `SkCanvas` 中并返回与之对应 `Canvas`。而当调用 `Surface.unlockCanvasAndPost` 时则会清空 `SkCanvas` 并将 `SkBitmap` 设置为空，此时 `Locked Buffer` 将会被解锁并重新扔回图像缓存队列中，同时将 `Poated Buffer` 设置为 `Locked Buffer`，旧的 `Posted Buffer` 就可以被下次取出来使用，设置 `Locked Buffer` 为空，当 SF 下次进行 `screen composite` 的时候就会把当前 `Poated Buffer` 绘制到屏幕上，这算是 `Canvas` 到屏幕绘制的一个小过程，当然事实比我说的复杂得多，这又是我的一个删减版本而已，懂得就听，不懂的权当废话不用管，我们不会涉及到这么深，像什么 `HardwareCanvas`、`GL` 之类的太过深入没必要去学，这里只阐述一个小原理而已。

对我们普通开发者来说，要记住的的是，一个 `Canvas` 需要一个 `Bitmap` 来保存像素信息，你说不要行不行？当然可以，画得东西没法保存而已，既然没法保存那我画来还有何意义呢？isn't it？

`Canvas` 所提供的各种方法根据功能来看大致可以分为几类，第一是以 `drawXXX` 为主的绘制方法，第二是以 `clipXXX` 为主的裁剪方法，第三是以 `scale`、`skew`、`translate` 和 `rotate` 组成的 `Canvas` 变换方法，最后一类则是以 `saveXXX` 和 `restoreXXX` 构成的画布锁定和还原，还有一些渣渣方法就不归类了。

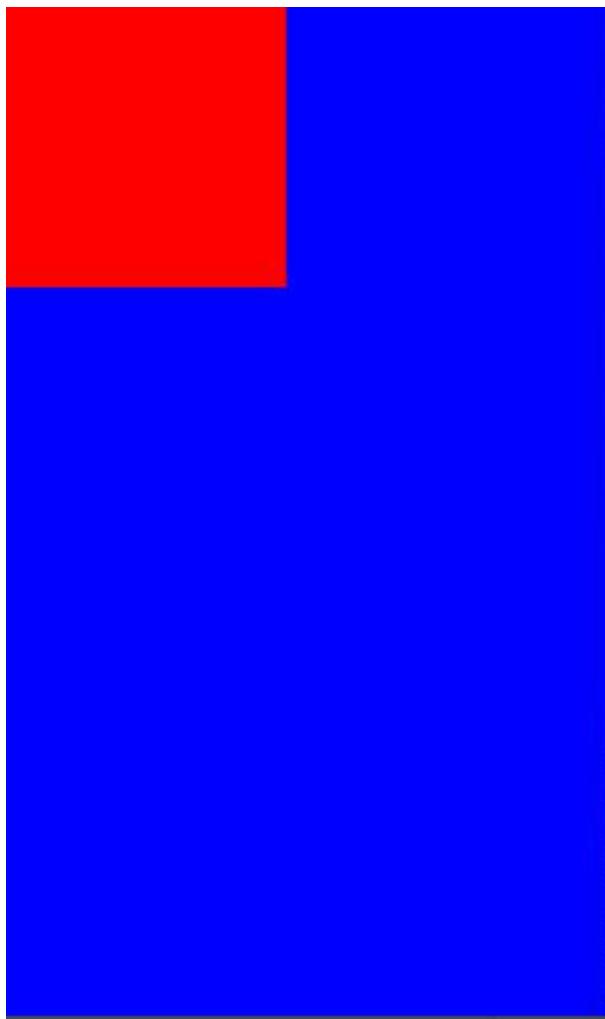
绘制图形、变换锁定还原画布我们都在前面的一些 `code` 中使用过，那么什么叫裁剪画布呢？我们来看一段 `code`：

[java] view plaincopyprint?

```
1. public class CanvasView extends View {
2.     public CanvasView(Context context, AttributeSet attrs) {
3.         super(context, attrs);
```

```
4.      }
5.
6.      @Override
7.      protected void onDraw(Canvas canvas) {
8.          canvas.drawColor(Color.BLUE);
9.          canvas.clipRect(0, 0, 500, 500);
10.         canvas.drawColor(Color.RED);
11.     }
12. }
```

这段代码非常简单，我们在 `onDraw` 中将整个画布绘制为蓝色，然后我们在当前画布上从 [0,0] 为左端点开始裁剪出一块 500x500 大小的矩形，再次将画布绘制为红色，你会发现只有被裁剪的区域才能被绘制为红色：



是不是有点懂裁剪的意思了？不懂？没事，我们再画一个圆加深理解：

[java] view plaincopyprint?

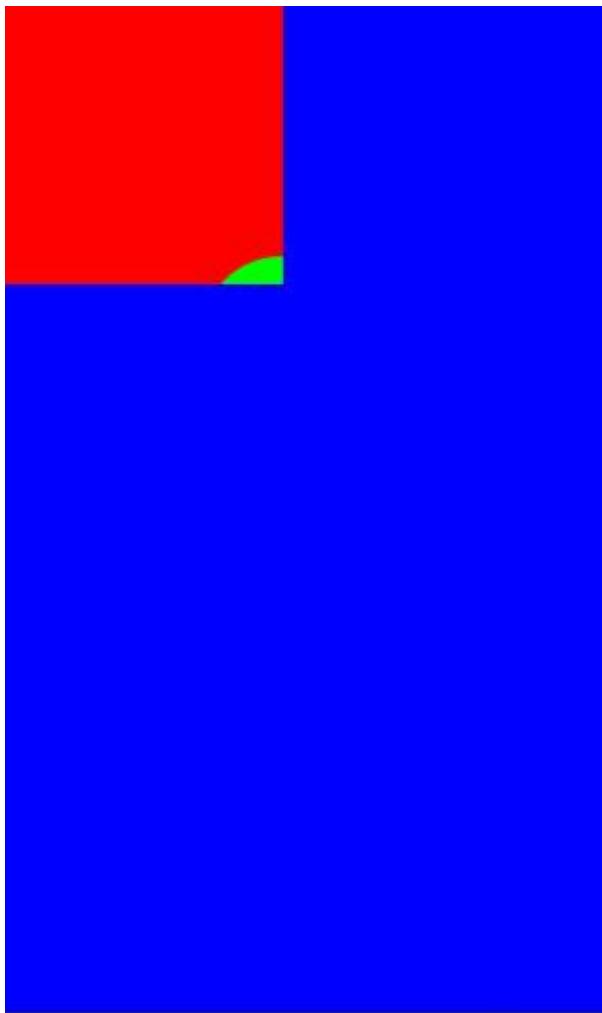
```
1. public class CanvasView extends View {
2.     private Paint mPaint;
```

```
3.  
4.     public CanvasView(Context context, AttributeSet attrs) {  
5.         super(context, attrs);  
6.  
7.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);  
8.         mPaint.setStyle(Paint.Style.FILL);  
9.         mPaint.setColor(Color.GREEN);  
10.    }  
11.  
12.    @Override  
13.    protected void onDraw(Canvas canvas) {  
14.        canvas.drawColor(Color.BLUE);  
15.        canvas.clipRect(0, 0, 500, 500);  
16.        canvas.drawColor(Color.RED);  
17.        canvas.drawCircle(500, 600, 100, mPaint);  
18.    }  
19. }
```

如代码所示，我们在以[500,600]为圆心绘制一个半径为 100px 的绿色圆，按道理来说，这个圆应该刚好与红色区域下方相切对吧，但是事实上呢我们见不到任何效果，为什么？因为如上所说，当前画布被“裁剪”了，只有 500x500 也就是上图中红色区域的大小了，如果我们所绘制的东西在该区域外部，即便绘制了你也看不到，这时我们稍增大圆的半径：

[java] view plaincopyprint?

```
1.    canvas.drawCircle(500, 600, 150, mPaint);
```



终于看到我们的圆“露”出来了~~现在你能稍微明白裁剪的作用了么？上面的代码中我们使用到了 Canvas 的

[java] view plaincopyprint?

```
1. clipRect(int left, int top, int right, int bottom)
```

方法，与之类似的还有

[java] view plaincopyprint?

```
1. clipRect(float left, float top, float right, float bottom)
```

方法，一个 int 一个 float，不扯了。除此之外还有两个与之对应的方法

[java] view plaincopyprint?

```
1. clipRect(Rect rect)  
2. clipRect(RectF rect)
```

, Rect 和 RectF 是类似的, 只不过 RectF 中涉及计算的时候数值类型均为 float 型, 两者均表示一块规则矩形, 何以见得呢? 我们以 Rect 为例来 Test 一下:

[java] view plaincopyprint?

```
1. public class CanvasView extends View {  
2.     private Rect mRect;  
3.  
4.     public CanvasView(Context context, AttributeSet attrs) {  
5.         super(context, attrs);  
6.         mRect = new Rect(0, 0, 500, 500);  
7.     }  
8.  
9.     @Override  
10.    protected void onDraw(Canvas canvas) {  
11.        canvas.drawColor(Color.BLUE);  
12.  
13.        canvas.clipRect(mRect);  
14.  
15.        canvas.drawColor(Color.RED);  
16.    }  
17. }
```

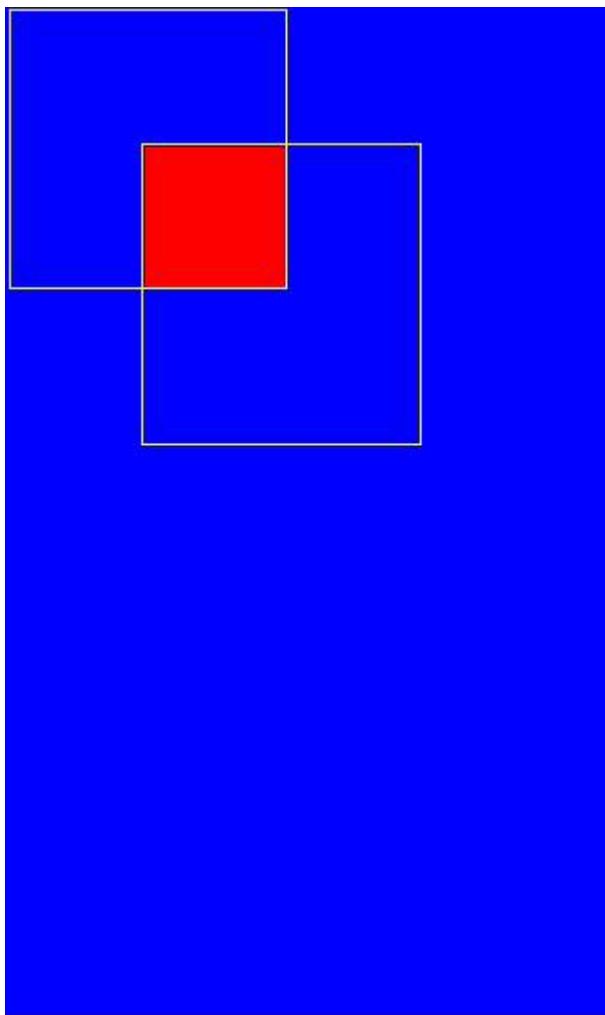
如代码所示这样我们得到的结果跟上面的结果并无二致, 蓝色的底, 500x500 大小的红色矩形, 但是 Rect 的意义远不止于此, 鉴于 Rect 类并不复杂, 我就讲两个其比较重要的方法, 我们稍微更改下我们的代码:

[java] view plaincopyprint?

```
1. public class CanvasView extends View {  
2.     private Rect mRect;  
3.  
4.     public CanvasView(Context context, AttributeSet attrs) {  
5.         super(context, attrs);  
6.         mRect = new Rect(0, 0, 500, 500);  
7.  
8.         mRect.intersect(250, 250, 750, 750);  
9.     }  
10.  
11.    @Override  
12.    protected void onDraw(Canvas canvas) {  
13.        canvas.drawColor(Color.BLUE);  
14.  
15.        canvas.clipRect(mRect);  
16.    }
```

```
17.         canvas.drawColor(Color.RED);
18.     }
19. }
```

大家看到我在实例化了一个 `Rect` 后调用了 `intersect` 方法，这个方法的作用是什么？来看看效果先：



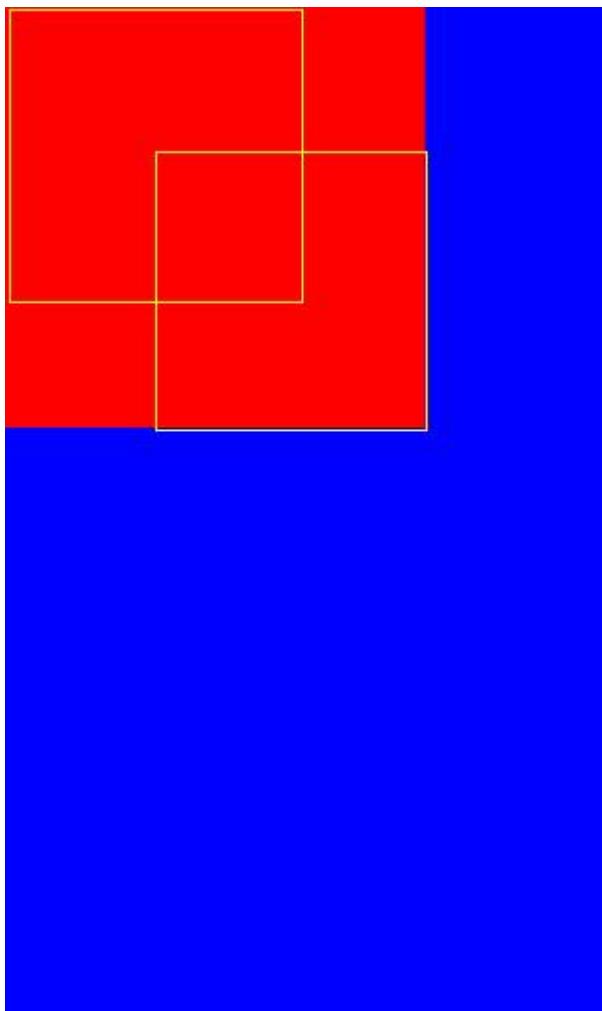
PS：黄色线框为后期加上的辅助线非程序生成

可以看到原先的红色区域变小了，这是怎么回事呢？其实 `intersect` 的作用跟我们之前学到的图形混合模式有点类似，它会取两个区域的相交区域作为最终区域，上面我们的第一个区域是在实例化 `Rect` 时确定的 `(0, 0, 500, 500)`，第二个区域是调用 `intersect` 方法时指定的 `(250, 250, 750, 750)`，这两个区域对应上图的两个黄色线框，两者相交的地方则为最终的红色区域，而 `intersect` 方法的计算方式是相当有趣的，它不是单纯地计算相交而是去计算相交区域最近的左上端点和最近的右下端点，不知道大家是否明白这个意思，我们来看 `Rect` 中的另一个 `union` 方法你就会懂，`union` 方法与 `intersect` 相反，取的是相交区域最远的左上端点作为新区域的左上端点，而取最远的右下端点作为新区域的右下端点，比如：

[java] view plain copy print?

```
1. mRect.union(250, 250, 750, 750);
```

运行后我们会看到如下结果：



是不是觉得不是我们想象中的那样单纯地两个区域相加？没事，好好体会，后面还有类似的。

类似的方法 `Rect` 和 `RectF` 都有很多，效果都是显而易见的就不多说了，有兴趣大家可以自己去 `try`。

说到这里会有很多童鞋会问，裁剪只是个矩形区域，如果我想要更多不规则的裁剪区域怎么办呢？别担心，Android 必然也考虑到这样的情况，其提供了一个

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. clipPath(Path path)
```

方法给我们以 `Path` 的方式创建更多不规则的裁剪区域，在 1/4 讲 `PathEffect` 的时候我们曾对 `Path` 有所接触，但是依旧不了解

`Path` 是 android 中用来封装几何学路径的一个类，因为 `Path` 在图形绘制上占的比重还是相当大的，这里我们先来学习一下这个 `Path`，来看看其一些具体的用法：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public class PathView extends View {
2.     private Path mPath;// 路径对象
3.     private Paint mPaint;// 画笔对象
4.
5.     public PathView(Context context, AttributeSet attrs) {
6.         super(context, attrs);
7.
8.         /*
9.          * 实例化画笔并设置属性
10.         */
11.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
12.        mPaint.setStyle(Paint.Style.STROKE);
13.        mPaint.setColor(Color.CYAN);
14.
15.        // 实例化路径
16.        mPath = new Path();
17.
18.        // 连接路径到点[100,100]
19.        mPath.lineTo(100, 100);
20.    }
21.
22.    @Override
23.    protected void onDraw(Canvas canvas) {
24.        // 绘制路径
25.        canvas.drawPath(mPath, mPaint);
26.    }
27. }
```

这里我们用到了 **Path** 的一个方法

[java] view plaincopyprint?

```
1. lineTo(float x, float y)
```

该方法很简单咯，顾名思义将路径连接至某个坐标点，事实也是如此：



注意，当我们没有移动 Path 的点时，其默认的起点为画布的[0,0]点，当然我们可以通过

[java] view plaincopyprint?

```
1. moveTo(float x, float y)
```

方法来改变这个起始点的位置：

[java] view plaincopyprint?

```
1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[300,300]
5. mPath.moveTo(300, 300);
6.
7. // 连接路径到点[100,100]
8. mPath.lineTo(100, 100);
```

效果如下：



当然我们可以考虑多次调用 `lineTo` 方法来绘制更复杂的图形：

[java] view plaincopyprint?

```
1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[300,300]
```

```
5. mPath.moveTo(100, 100);
6.
7. // 连接路径到点
8. mPath.lineTo(300, 100);
9. mPath.lineTo(400, 200);
10. mPath.lineTo(200, 200);
```

一个没有封闭的类似平行四边形的线条：



如果此时我们想闭合该曲线让它变成一个形状该怎么做呢？聪明的你一定想到

[java] view plaincopyprint?

```
1. mPath.lineTo(100, 100)
```

然而 Path 给我提供了更便捷的方法

[java] view plaincopyprint?

```
1. close()
```

去闭合曲线：

[java] view plaincopyprint?

```
1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[300,300]
5. mPath.moveTo(100, 100);
6.
7. // 连接路径到点
8. mPath.lineTo(300, 100);
9. mPath.lineTo(400, 200);
```

```
10. mPath.lineTo(200, 200);
11.
12. // 闭合曲线
13. mPath.close();
```



那么有些朋友会问 **Path** 就只能光绘制这些单调的线段么？肯定不是！**Path** 在绘制的方法中提供了许多 **XXXTo** 的方法来帮助我们绘制各类直线、曲线，例如，方法

[java] view plaincopyprint?

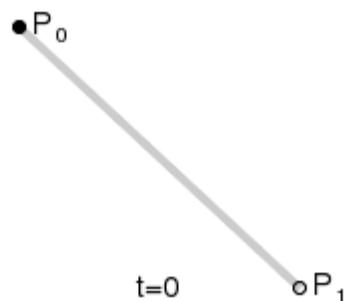
```
1. quadTo(float x1, float y1, float x2, float y2)
```

可以让我们绘制二阶贝塞尔曲线，什么叫贝塞尔曲线？其实很简单，使用三个或多个点来确定的一条曲线，贝塞尔曲线在图形图像学中有相当重要的地位，**Path** 中也提供了一些方法来给我们模拟低阶贝塞尔曲线。

贝塞尔曲线的定义也比较简单，你只需要一个起点、一个终点和至少零个控制点则可定义一个贝塞尔曲线，当控制点为零时，只有起点和终点，此时的曲线说白了就是一条线段，我们称之为

**PS：**以下图片和公式均来自维基百科和互联网

一阶贝塞尔曲线



其公式可概括为：

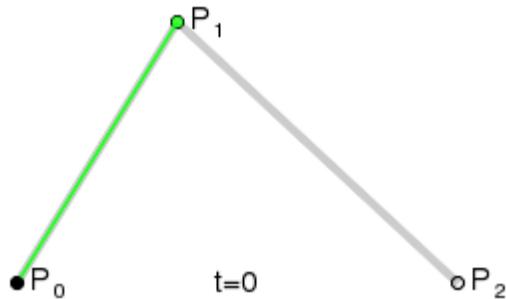
$$B(t) = (1-t)p_0 + tP_1 \quad , t \in [0,1]$$

其中  $B(t)$  为时间为  $t$  时点的坐标,  $P_0$  为起点、 $P_n$  为终点

贝塞尔曲线于 1962 年由法国数学家 Pierre Bézier 第一次研究使用并给出了详细的计算公式,

So 该曲线也是由其名字命名。Path 中给出的 quadTo 方法属于

二阶贝塞尔曲线



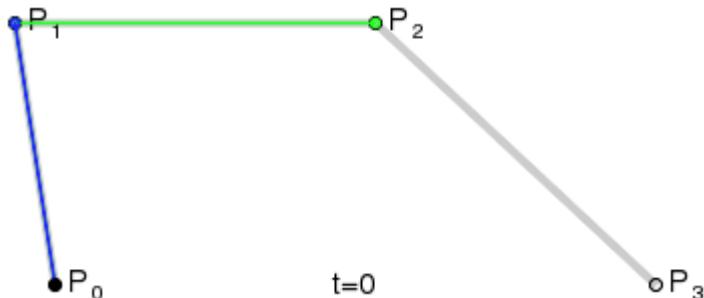
二阶贝塞尔曲线的一个明显特征是其拥有一个控制点, 大家可以这样想想贝塞尔曲线, 在一根两端固定橡皮筋上有一块磁铁, 现在我们拿另一块磁铁去吸引橡皮筋上的磁铁, 因为引力, 橡皮筋会随着我们手上磁铁的移动而改变形状, 又因为橡皮筋的张力让束缚在橡皮筋上的磁铁不会轻易吸附到我们手上的磁铁, 这时橡皮筋的状态就可以看成是一条贝塞尔曲线, 而我们手中的磁铁就是一个控制点, 通过这个控制点我们“拉扯”橡皮筋的曲度。

二阶贝塞尔曲线的公式为:

$$B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2, \quad t \in [0, 1]$$

同样的, Path 中也提供了三阶贝塞尔曲线的方法 cubicTo, 按照上面我们的推论, 三阶应该是有两个控制点才对吧

三阶贝塞尔曲线



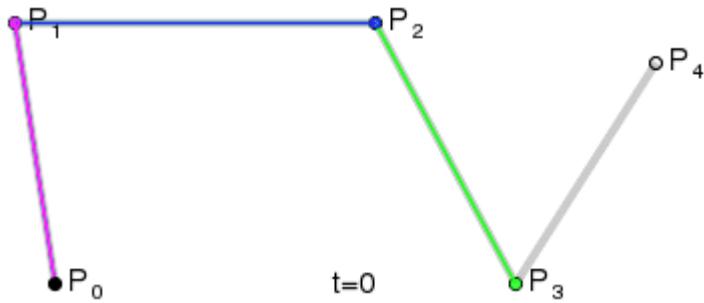
公式:

$$B(t) = P_0(1-t)^3 + 3P_1t(1-t)^2 + 3P_2t^2(1-t) + P_3t^3, \quad t \in [0, 1]$$

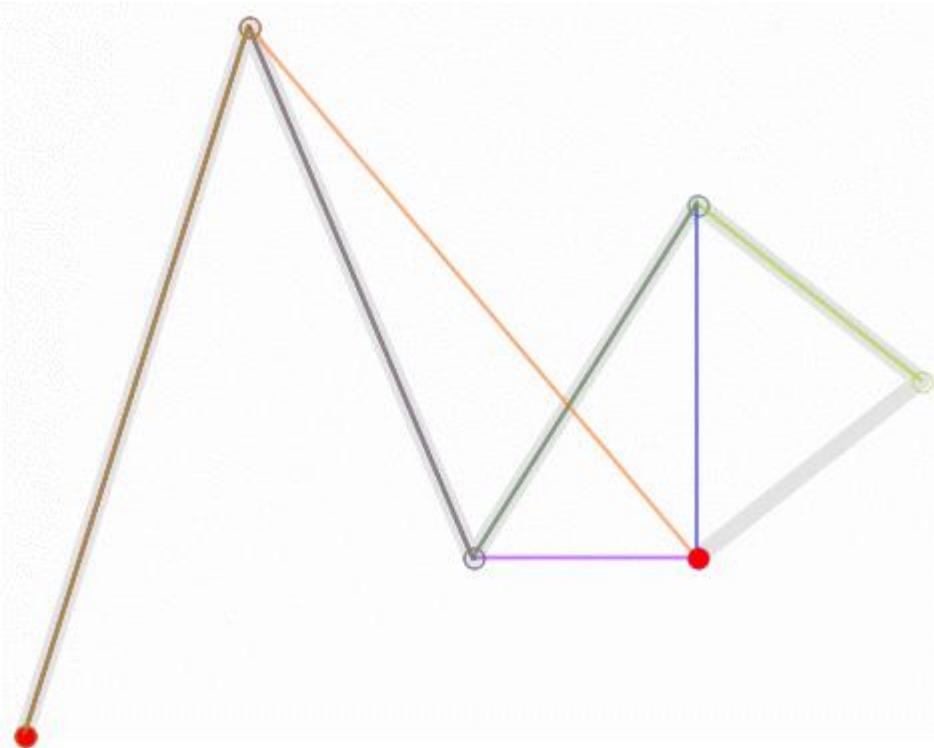
高阶贝塞尔曲线在 Path 中没有对应的方法, 对我们来说三阶也足够了, 不过大家可以了解下, 难得我在墙外找到如此动感的贝塞尔曲线高清无码动图

高阶贝塞尔曲线

四阶:



五阶：



贝塞尔曲线通用公式：

$$P_i^k = \begin{cases} P_i & k = 0 \\ (1-t)P_i^{k-1} + tP_{i+1}^{k-1} & k = 1, 2, \dots, n, i = 0, 1, \dots, n-k \end{cases}$$

回到我们 Path 的 quadTo 方法，我们可以使用它来绘制一条曲线：

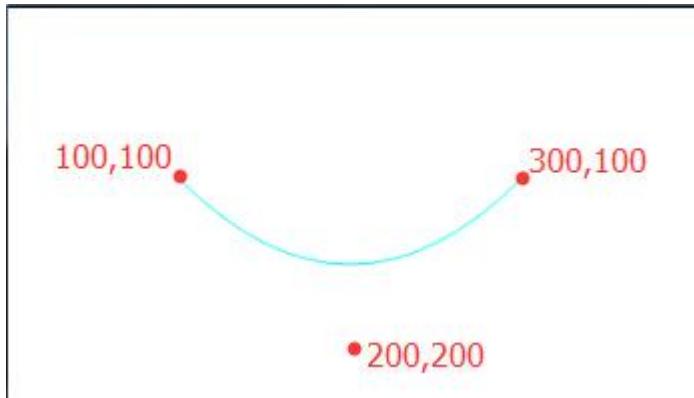
[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```

1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[100,100]
5. mPath.moveTo(100, 100);
6.
7. // 连接路径到点
8. mPath.quadTo(200, 200, 300, 100);

```

看图说话：



其中 `quadTo` 的前两个参数为控制点的坐标，后两个参数为终点坐标，至于起点嘛……这么二的问题就别问了……是不是很简单？如果你这么认为那就太小看贝塞尔曲线了。在我们对 `Path` 有一定的了解后会使用 `Path` 和裁剪做个有趣的东西，接着看 `Path` 的三阶贝赛尔曲线：

[java] view plaincopyprint?

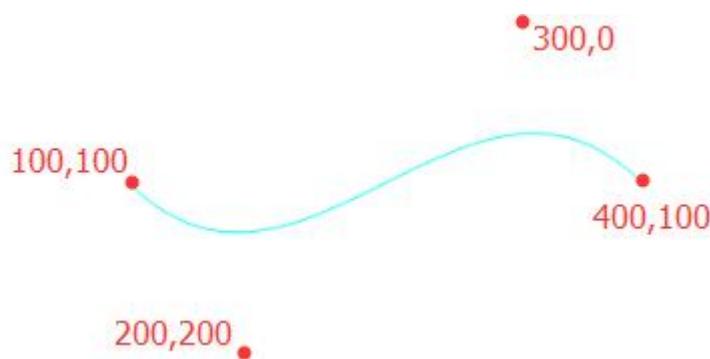
```
1. cubicTo(float x1, float y1, float x2, float y2, float x3, float y3)
```

与 `quadTo` 类似，前四个参数表示两个控制点，最后两个参数表示终点：

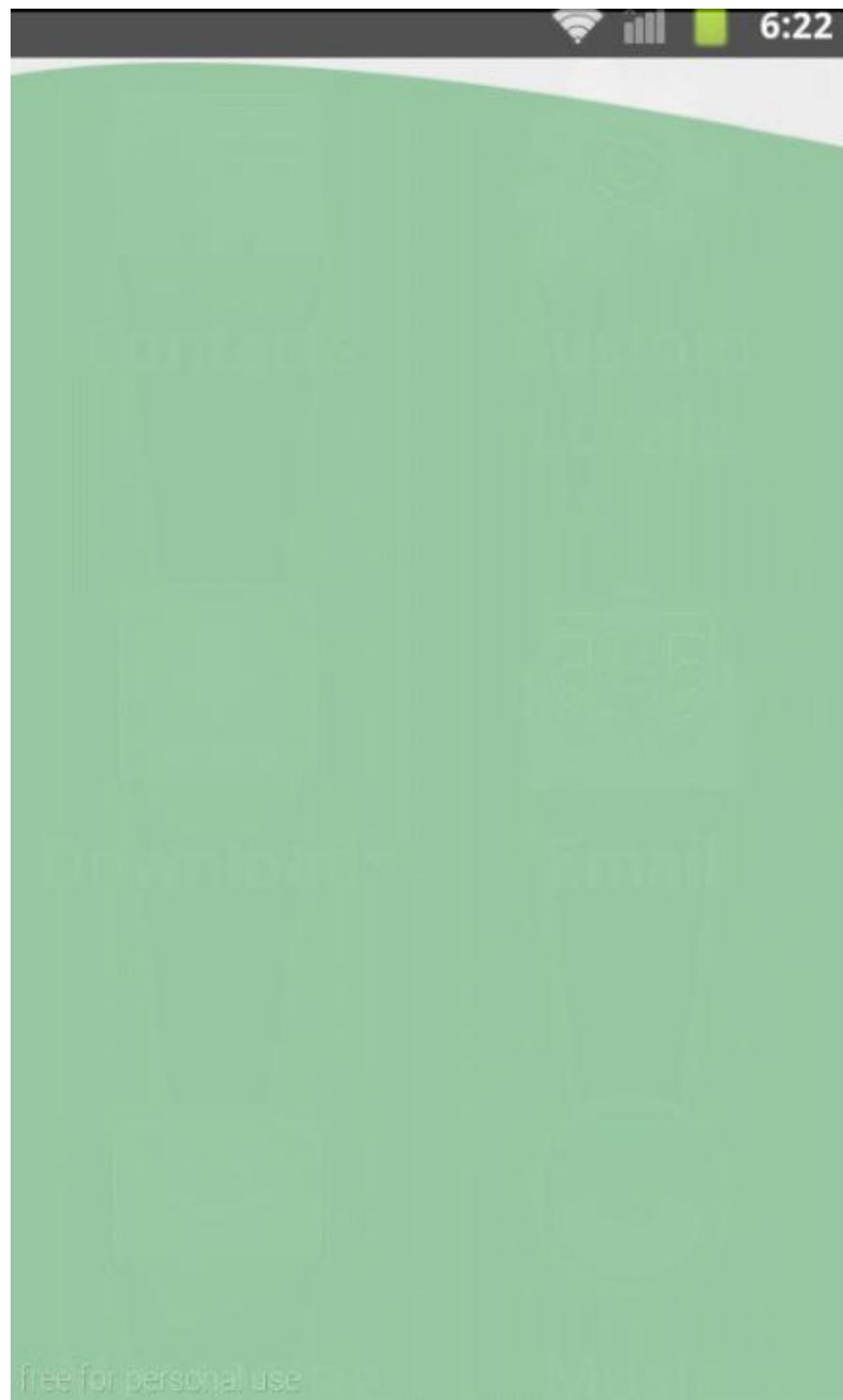
[java] view plaincopyprint?

```
1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[100,100]
5. mPath.moveTo(100, 100);
6.
7. // 连接路径到点
8. mPath.cubicTo(200, 200, 300, 0, 400, 100);
```

很好理解：



贝塞尔曲线是图形图像学中相当重要的一个概念，活用它可以得到很多很有意思的效果，比如，我在界面中简单模拟一下杯子中水消匿的效果：



当然你也可以反过来让模拟往杯子里倒水的效果~实现过程非常简单，说白了就是不断移动二阶曲线的控制点同时不断更改顶部各点的 Y 坐标，然后不断重绘：

[java] view plaincopyprint?

```
1. public class WaveView extends View {  
2.     private Path mPath; // 路径对象
```

```
3.     private Paint mPaint;// 画笔对象
4.
5.     private int vWidth, vHeight;// 控件宽高
6.     private float ctrX, ctrY;// 控制点的 xy 坐标
7.     private float waveY;// 整个 Wave 顶部两端点的 Y 坐标,该坐标与控制点的 Y 坐标增减
   帧一致
8.
9.     private boolean isInc;// 判断控制点是该右移还是左移
10.
11.    public WaveView(Context context, AttributeSet attrs) {
12.        super(context, attrs);
13.
14.        // 实例化画笔并设置参数
15.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
16.        mPaint.setColor(0xFFA2D6AE);
17.
18.        // 实例化路径对象
19.        mPath = new Path();
20.    }
21.
22.    @Override
23.    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
24.        // 获取控件宽高
25.        vWidth = w;
26.        vHeight = h;
27.
28.        // 计算控制点 Y 坐标
29.        waveY = 1 / 8F * vHeight;
30.
31.        // 计算端点 Y 坐标
32.        ctrY = -1 / 16F * vHeight;
33.    }
34.
35.    @Override
36.    protected void onDraw(Canvas canvas) {
37.        /*
38.         * 设置 Path 起点
39.         * 注意我将 Path 的起点设置在了控件的外部看不到的区域
40.         * 如果我们将起点设置在控件左端 x=0 的位置会使得贝塞尔曲线变得生硬
41.         * 至于为什么刚才我已经说了
42.         * 所以我们稍微让起点往“外”走点
43.         */
44.        mPath.moveTo(-1 / 4F * vWidth, waveY);
45.    }
}
```

```
46.         /*
47.          * 以二阶曲线的方式通过控制点连接位于控件右边的终点
48.          * 终点的位置也是在控件外部
49.          * 我们只需不断让 ctrX 的大小变化即可实现“浪”的效果
50.         */
51.         mPath.quadTo(ctrX, ctrY, vWidth + 1 / 4F * vWidth, waveY);
52.
53.         // 围绕控件闭合曲线
54.         mPath.lineTo(vWidth + 1 / 4F * vWidth, vHeight);
55.         mPath.lineTo(-1 / 4F * vWidth, vHeight);
56.         mPath.close();
57.
58.         canvas.drawPath(mPath, mPaint);
59.
60.         /*
61.          * 当控制点的 x 坐标大于或等于终点 x 坐标时更改标识值
62.          */
63.         if (ctrX >= vWidth + 1 / 4F * vWidth) {
64.             isInc = false;
65.         }
66.         /*
67.          * 当控制点的 x 坐标小于或等于起点 x 坐标时更改标识值
68.          */
69.         else if (ctrX <= -1 / 4F * vWidth) {
70.             isInc = true;
71.         }
72.
73.         // 根据标识值判断当前的控制点 x 坐标是该加还是减
74.         ctrX = isInc ? ctrX + 20 : ctrX - 20;
75.
76.         /*
77.          * 让“水”不断减少
78.          */
79.         if (ctrY <= vHeight) {
80.             ctrY += 2;
81.             waveY += 2;
82.         }
83.
84.         mPath.reset();
85.
86.         // 重绘
87.         invalidate();
88.     }
89. }
```

除了上面的几个 XXXTo 外，Path 还提供了一个

[java] view plaincopyprint?

```
1. arcTo (RectF oval, float startAngle, float sweepAngle)
```

方法用来生成弧线，其实说白了就是从圆或椭圆上截取一部分而已 ==

[java] view plaincopyprint?

```
1. // 实例化路径
2. mPath = new Path();
3.
4. // 移动点至[100,100]
5. mPath.moveTo(100, 100);
6.
7. // 连接路径到点
8. RectF oval = new RectF(100, 100, 200, 200);
9. mPath.arcTo(oval, 0, 90);
```

效果如下图：



这里要注意哦，使用 Path 生成的路径必定都是连贯的，虽然我们使用 arcTo 绘制的是一段弧但其最终都会与我们的起始点[100,100]连接起来，如果你不想连怎么办？简单，强制让 arcTo 绘制的起点作为 Path 的起点不就是了？Path 也提供了另一个重载方法：

[java] view plaincopyprint?

```
1. arcTo (RectF oval, float startAngle, float sweepAngle, boolean forceMoveTo)
```

该方法只是多了一个布尔值，值为 true 时将会把弧的起点作为 Path 的起点：

[java] view plaincopyprint?

```
1. mPath.arcTo(oval, 0, 90, true);
```

like below:



Path 中除了上面介绍的几个 XXXTo 方法外还有一套 rXXXTo 方法:

[java] view plaincopyprint?

```
1. rCubicTo(float x1, float y1, float x2, float y2, float x3, float y3)
2. rLineTo(float dx, float dy)
3. rMoveTo(float dx, float dy)
4. rQuadTo(float dx1, float dy1, float dx2, float dy2)
```

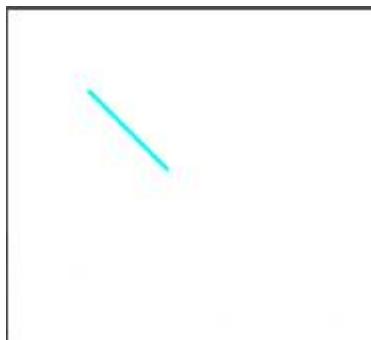
这一系列 rXXXTo 方法其实跟上面的那些 XXXTo 差不多的，唯一的不同是 rXXXTo 方法的参考坐标是相对的而 XXXTo 方法的参考坐标始终是参照画布原点坐标，什么意思呢？举个简单的例子：

[java] view plaincopyprint?

```
1. public class PathView extends View {
2.     private Path mPath; // 路径对象
3.     private Paint mPaint; // 画笔对象
4.
5.     public PathView(Context context, AttributeSet attrs) {
6.         super(context, attrs);
7.
8.         /*
9.          * 实例化画笔并设置属性
10.         */
11.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
```

```
12.         mPaint.setStyle(Paint.Style.STROKE);
13.         mPaint.setColor(Color.CYAN);
14.         mPaint.setStrokeWidth(5);
15.
16.         // 实例化路径
17.         mPath = new Path();
18.
19.         // 移动点至[100,100]
20.         mPath.moveTo(100, 100);
21.
22.         // 连接路径到点
23.         mPath.lineTo(200, 200);
24.     }
25.
26.     @Override
27.     protected void onDraw(Canvas canvas) {
28.         // 绘制路径
29.         canvas.drawPath(mPath, mPaint);
30.     }
31. }
```

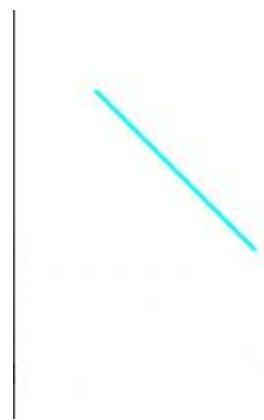
上述代码我们从点[100,100]开始连接点[200,200]构成了一条线段：



这个点[200,200]是相对于画布圆点坐标[0,0]而言的，这点大家应该好理解，如果我们换成  
[java] view plaincopyprint?

```
1. mPath.rLineTo(200, 200);
```

那么它的意思就是将会以[100,100]作为原点坐标，连接以其为原点坐标的坐标点[200,200]，如果换算成一画布原点的话，实际上现在的[200,200]就是[300,300]了：



懂了么？而这个前缀 `r` 也就是 `relative`（相对）的简写，`so easy` 是么！头脑简单！

`XXXTo` 方法可以连接 `Path` 中的曲线而 `Path` 提供的另一系列 `addXXX` 方法则可以让我们直接往 `Path` 中添加一些曲线，比如

[java] view plaincopyprint?

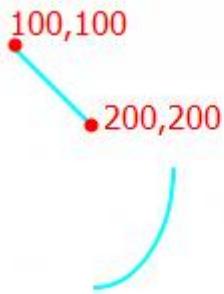
```
1. addArc(RectF oval, float startAngle, float sweepAngle)
```

方法允许我们将一段弧形添加至 `Path`，注意这里我用到了“添加”这个词汇，也就是说，通过 `addXXX` 方法添加到 `Path` 中的曲线是不会和上一次的曲线进行连接的：

[java] view plaincopyprint?

```
1. public class PathView extends View {  
2.     private Path mPath; // 路径对象  
3.     private Paint mPaint; // 路径画笔对象  
4.  
5.     public PathView(Context context, AttributeSet attrs) {  
6.         super(context, attrs);  
7.  
8.         /*  
9.             * 实例化画笔并设置属性  
10.            */  
11.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
12.         mPaint.setStyle(Paint.Style.STROKE);  
13.         mPaint.setColor(Color.CYAN);  
14.         mPaint.setStrokeWidth(5);  
15.  
16.         // 实例化路径  
17.         mPath = new Path();  
18.  
19.         // 移动点至[100,100]  
20.         mPath.moveTo(100, 100);  
21.  
22.         // 连接路径到点
```

```
23.         mPath.lineTo(200, 200);
24.
25.         // 添加一条弧线到 Path 中
26.         RectF oval = new RectF(100, 100, 300, 400);
27.         mPath.addArc(oval, 0, 90);
28.     }
29.
30.    @Override
31.    protected void onDraw(Canvas canvas) {
32.        // 绘制路径
33.        canvas.drawPath(mPath, mPaint);
34.    }
35.}
```



如图和代码所示，虽然我们先绘制了由[100,100]到[200,200]的线段，但是在我们往 Path 中添加了一条弧线后该弧线并没与线段连接。除了 addArc，Path 还提供了一系列的 add 方法

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. addCircle(float x, float y, float radius, Path.Direction dir)
2. addOval(float left, float top, float right, float bottom, Path.Direction dir
   )
3. addRect(float left, float top, float right, float bottom, Path.Direction dir
   )
4. addRoundRect(float left, float top, float right, float bottom, float rx, float
   ry, Path.Direction dir)
```

这些方法和 addArc 有很明显的区别，就是多了一个 Path.Direction 参数，其他呢都大同小异，除此之外不知道大家还发现没有，addArc 是往 Path 中添加一段弧，说白了就是一条开放的曲线，而上述几种方法都是一个具体的图形，或者说是一条闭合的曲线，Path.Direction 的意思就是标识这些闭合曲线的闭合方向。那什么叫闭合方向呢？光说大家一定会蒙，有学习激情的童鞋看到后肯定会马上敲代码试验一下两者的区别，可是不管你如何改，单独地在

一条闭合曲线上你是看不出所谓闭合方向区别的，这时我们可以借助 **Canvas** 的另一个方法来简单地说明一下

[java] view plaincopyprint?

```
1. drawTextOnPath(String text, Path path, float hOffset, float vOffset, Paint p  
aint)
```

这个方法呢很简单沿着 **Path** 绘制一段文字，参数也是一看就该懂得了不多说。

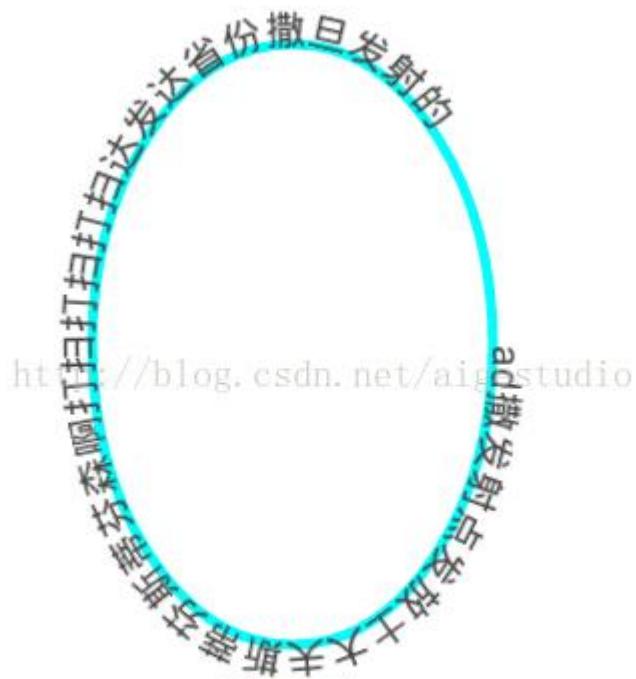
**Path.Direction** 只有两个常量值 **CCW** 和 **CW** 分别表示逆时针方向闭合和顺时针方向闭合，我们来看一段代码

[java] view plaincopyprint?

```
1. public class PathView extends View {  
2.     private Path mPath; // 路径对象  
3.     private Paint mPaint; // 路径画笔对象  
4.     private TextPaint mTextPaint; // 文本画笔对象  
5.  
6.     public PathView(Context context, AttributeSet attrs) {  
7.         super(context, attrs);  
8.  
9.         /*  
10.            * 实例化画笔并设置属性  
11.            */  
12.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
13.         mPaint.setStyle(Paint.Style.STROKE);  
14.         mPaint.setColor(Color.CYAN);  
15.         mPaint.setStrokeWidth(5);  
16.  
17.         mTextPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG  
| Paint.LINEAR_TEXT_FLAG);  
18.         mTextPaint.setColor(Color.DKGRAY);  
19.         mTextPaint.setTextSize(20);  
20.  
21.         // 实例化路径  
22.         mPath = new Path();  
23.  
24.         // 添加一条弧线到 Path 中  
25.         RectF oval = new RectF(100, 100, 300, 400);  
26.         mPath.addOval(oval, Path.Direction.CW);  
27.     }  
28.  
29.     @Override  
30.     protected void onDraw(Canvas canvas) {
```

```
31.         // 绘制路径
32.         canvas.drawPath(mPath, mPaint);
33.
34.         // 绘制路径上的文字
35.         canvas.drawTextOnPath("ad 撒发射点发放士大夫斯蒂芬斯蒂芬森啊打扫打扫打扫  
发达省份撒旦发射的", mPath, 0, 0, mTextPaint);
36.     }
37. }
```

我们往 Path 中添加了一条闭合方向为 CW 椭圆形的闭合曲线并将其绘制在 Canvas 上，同时呢我们沿着该曲线绘制了一段文本，效果如下：



<http://blog.csdn.net/aigstudio>

如果我们把闭合方向改为 CCW 那么会发生什么呢？

[java] view plaincopyprint?

```
1. mPath.addOval(oval, Path.Direction.CCW);
```



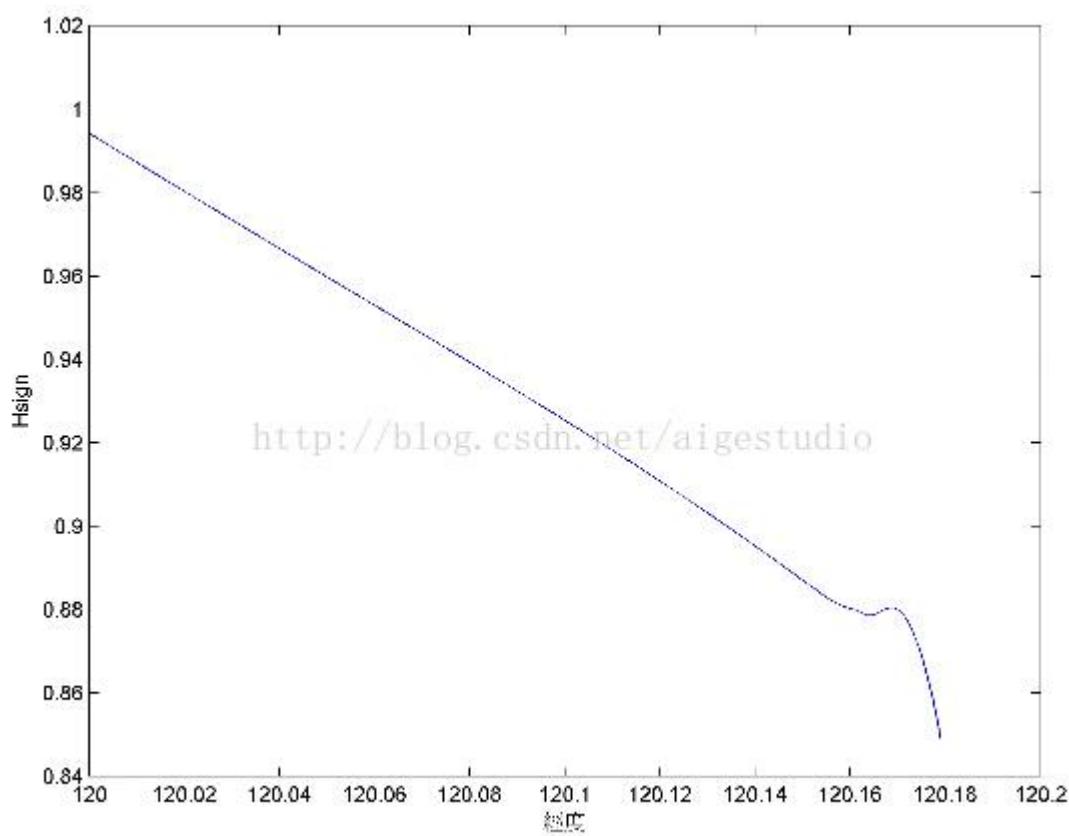
沿着 **Path** 的文字全都在闭合曲线的“内部”了，**Path.Direction** 闭合方向大概就是这么个意思。对于我们平时开发来说，掌握 **Path** 的以上一些方法已经是足够了，当然 **Path** 的方法还有很多，但是因为平时开发涉及的少，我也就不累赘了，毕竟用得少或者根本不会用到的东西说了也是浪费口水，对吧。

**Path** 用的也相当广泛，在之前的章节中我们也讲过一个 **PathEffect** 类，两者结合可以得到很多很酷的效果。在众多的用途中，使用 **Path** 做折线图算是最最最常见的了，仅仅使用以上我们讲到的一些 **Path** 的方法可以完成很多的折线图效果。

在上一节最后的一个例子中我们绘制了一个自定义的圈圈 **View**，当时我跟大家说过在你想去自定义一个控件的时候一定要把自己看作一个 **designer** 而不是 **coder**，你要用设计的眼光去看待一个控件，那么我们在做一个折线图的控件之前就应该要分析一个折线图应该是怎样的，下面我 **google** 一些简单折线图的例子：



这种比较简单

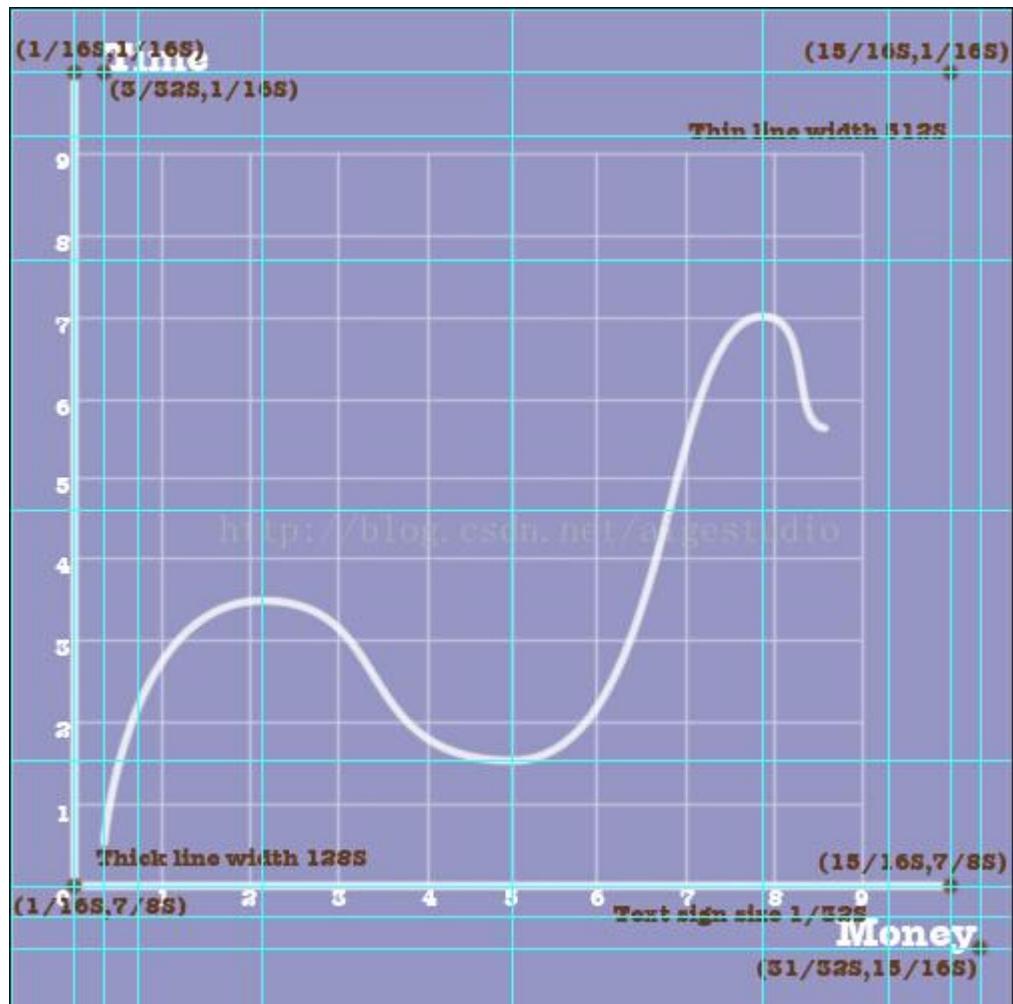


这种呢有文字标注稍难

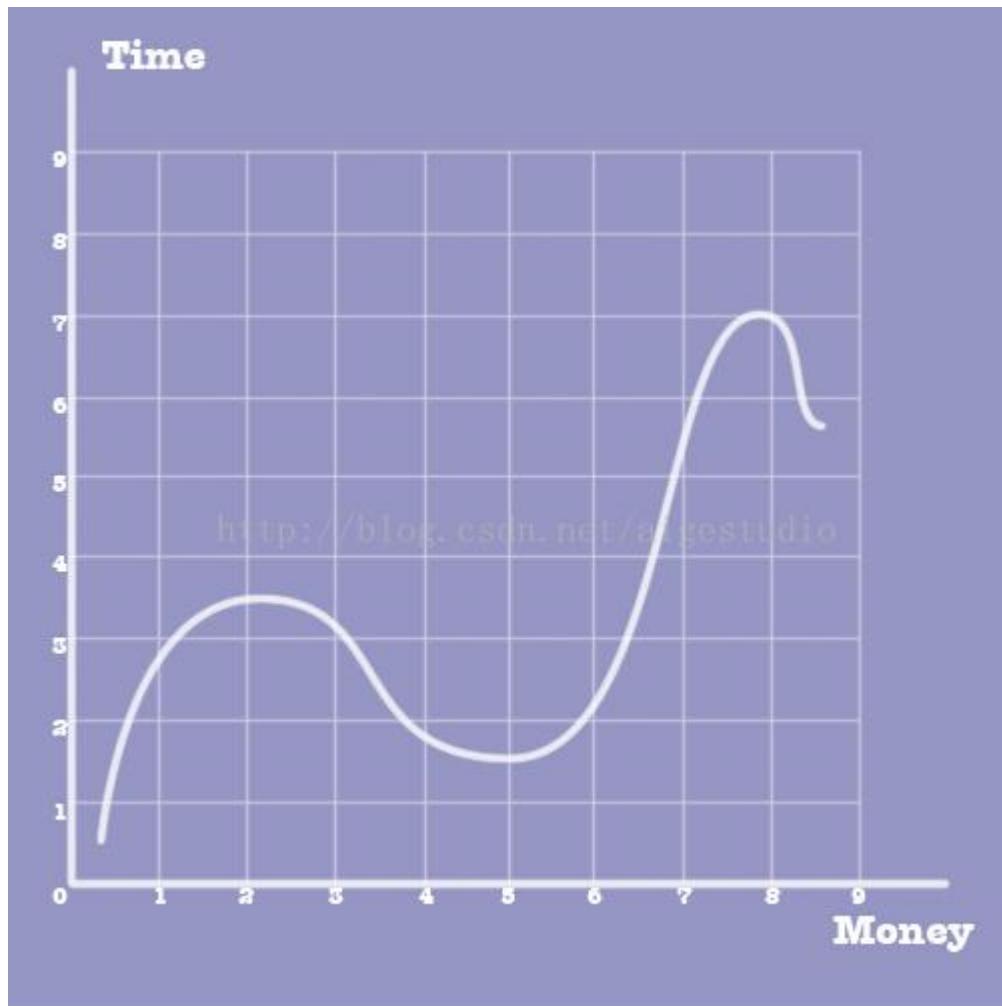


这种就复杂了点

不管是哪种折线图，我们都可以发现其必有一个横坐标和一个纵坐标且其上都有刻度，一般情况下来说横纵坐标上的刻度数量是一样的。对于平面折线图来说，分析到上面一点就差不多了，而我们要做的折线图控件我在 PS 里简单地做了一个 design:



设计地很简单，当中有一些辅助参数什么的，实际上整个控件就几个元素：



如上图所示，两个带刻度的轴和一个网格还有两个轴文字标识和一条曲线，**very simple!** 图好像很简单~~但是真要 **code** 起来就不是件容易的事了，首先我们要考虑到不同的数据、其次是屏幕的适配，说到适配，上一节我们曾讲过，因为屏幕的多元化，我们必定不能写死一个参数，**so~**我们在上一节画圈圈的时候是使用控件的边长来作为所有数值的基准参考，这次也一样。因为折线图的形状是跟外部数据相关的，所以在设计的时候我们必定要考虑对外公布一个设置数据的方法：

[java] view plaincopyprint?

```
1. /**
2.  * 设置数据
3. *
4.  * @param pointFs
5.  *          点集合
6. */
7. public synchronized void setData(List<PointF> pointFs, String signX, String
signY) {
8.     /*
9.      * 数据为空直接 GG

```

```
10.     */
11.     if (null == pointFs || pointFs.size() == 0)
12.         throw new IllegalArgumentException("No data to display !");
13.
14.     /*
15.      * 控制数据长度不超过 10 个
16.      * 对于折线图来说数据太多就没必要用折线图表示了而是使用散点图
17.      */
18.     if (pointFs.size() > 10)
19.         throw new IllegalArgumentException("The data is too long to display
   !");
20.
21.     // 设置数据并重绘视图
22.     this.pointFs = pointFs;
23.     this.signX = signX;
24.     this.signY = signY;
25.     invalidate();
26. }
```

折线图是表示的数据一般不会太多，如果太多，在有限的先是空间内必定显示鸡肋……当然股票那种巨幅大盘之类的另说。所以在上面的数据设置中我强制将数据长度控制在 10 个以内。

PS：该方法在设计上不太符合设计原则，这里就当大家都不会设计模式设计原则了 == Fuck.....

上面我们说过会以控件的边长作为基准参考计算各种数值，因为我们还没学习如何测量控件，这里还是和上一节一样强制将控件的宽高设置一致（强制竖屏）：

[java] view plaincopyprint?

```
1. @Override
2. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.     // 在我们没学习测量控件之前强制宽高一致
4.     super.onMeasure(widthMeasureSpec, widthMeasureSpec);
5. }
```

而控件尺寸我们在 `onSizeChanged` 方法中获取，这个方法是官方比较推崇的获取控件尺寸的方法，如果你不需要更精确的测量的话，同时我们也就将就在该方法内计算各类数值了：

[java] view plaincopyprint?

```
1. @Override
2. protected void onSizeChanged(int w, int h, int oldw, int oldh) {
3.     // 获取控件尺寸
4.     viewSize = w;
```

```

5.      // 计算纵轴标识文本坐标
6.      textY_X = viewSize * TIME_X;
7.      textY_Y = viewSize * TIME_Y;
8.
9.
10.     // 计算横轴标识文本坐标
11.     textX_X = viewSize * MONEY_X;
12.     textX_Y = viewSize * MONEY_Y;
13.
14.     // 计算 xy 轴标识文本大小
15.     textSignSzie = viewSize * TEXT_SIGN;
16.
17.     // 计算网格左上右下两点坐标
18.     left = viewSize * LEFT;
19.     top = viewSize * TOP;
20.     right = viewSize * RIGHT;
21.     bottom = viewSize * BOTTOM;
22.
23.     // 计算粗线宽度
24.     thickLineWidth = viewSize * THICK_LINE_WIDTH;
25.
26.     // 计算细线宽度
27.     thinLineWidth = viewSize * THIN_LINE_WIDTH;
28. }
```

其中的常量值均为比例值，根据控件中元素占比计算实际的像素大小，绘制逻辑稍微有点复杂，但是并不难，这里我就直接上全部代码了：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```

1.  public class PolylineView extends View {
2.      private static final float LEFT = 1 / 16F, TOP = 1 / 16F, RIGHT = 15 / 1
   6F, BOTTOM = 7 / 8F;// 网格区域相对位置
3.      private static final float TIME_X = 3 / 32F, TIME_Y = 1 / 16F, MONEY_X =
   31 / 32F, MONEY_Y = 15 / 16F;// 文字坐标相对位置
4.      private static final float TEXT_SIGN = 1 / 32F;// 文字相对大小
5.      private static final float THICK_LINE_WIDTH = 1 / 128F, THIN_LINE_WIDTH
   = 1 / 512F;// 粗线和细线相对大小
6.
7.      private TextPaint mTextPaint;// 文字画笔
8.      private Paint linePaint, pointPaint;// 线条画笔和点画笔
9.      private Path mPath;// 路径对象
10.     private Bitmap mBitmap;// 绘制曲线的 Bitmap 对象
11.     private Canvas mCanvas;// 装载 mBitmap 的 Canvas 对象
12.
```

```
13.     private List<PointF> pointFs;// 数据列表
14.     private float[] rulerX, rulerY;// xy 轴向刻度
15.
16.     private String signX, signY;// 设置 X 和 Y 坐标分别表示什么的文字
17.     private float textY_X, textY_Y, textX_X, textX_Y;// 文字坐标
18.     private float textSignSzie;// xy 坐标标识文本字体大小
19.     private float thickLineWidth, thinLineWidth;// 粗线和细线宽度
20.     private float left, top, right, bottom;// 网格区域左上右下两点坐标
21.     private int viewSize;// 控件尺寸
22.     private float maxX, maxY;// 横纵轴向最大刻度
23.     private float spaceX, spaceY;// 刻度间隔
24.
25.     public PolylineView(Context context, AttributeSet attrs) {
26.         super(context, attrs);
27.
28.         // 实例化文本画笔并设置参数
29.         mTextPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG
| Paint.LINEAR_TEXT_FLAG);
30.         mTextPaint.setColor(Color.WHITE);
31.
32.         // 实例化线条画笔并设置参数
33.         linePaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
34.         linePaint.setStyle(Paint.Style.STROKE);
35.         linePaint.setColor(Color.WHITE);
36.
37.         // 实例化点画笔并设置参数
38.         pointPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
39.         pointPaint.setStyle(Paint.Style.FILL);
40.         pointPaint.setColor(Color.WHITE);
41.
42.         // 实例化 Path 对象
43.         mPath = new Path();
44.
45.         // 实例化 Canvas 对象
46.         mCanvas = new Canvas();
47.
48.         // 初始化数据
49.         initData();
50.     }
51.
52.     /**
53.      * 初始化数据支撑
54.      * View 初始化时可以考虑给予一个模拟数据
55.      * 当然我们可以通过 setData 方法设置自己的数据
```

```
56.     */
57.     private void initData() {
58.         Random random = new Random();
59.         pointFs = new ArrayList<PointF>();
60.         for (int i = 0; i < 20; i++) {
61.             PointF pointF = new PointF();
62.
63.             pointF.x = (float) (random.nextInt(100) * i);
64.             pointF.y = (float) (random.nextInt(100) * i);
65.
66.             pointFs.add(pointF);
67.         }
68.     }
69.
70.     @Override
71.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
72.
73.         // 在我们没学习测量控件之前强制宽高一致
74.         super.onMeasure(widthMeasureSpec, widthMeasureSpec);
75.     }
76.
77.     @Override
78.     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
79.         // 获取控件尺寸
80.         viewSize = w;
81.
82.         // 计算纵轴标识文本坐标
83.         textY_X = viewSize * TIME_X;
84.         textY_Y = viewSize * TIME_Y;
85.
86.         // 计算横轴标识文本坐标
87.         textX_X = viewSize * MONEY_X;
88.         textX_Y = viewSize * MONEY_Y;
89.
90.         // 计算xy轴标识文本大小
91.         textSignSzie = viewSize * TEXT_SIGN;
92.
93.         // 计算网格左上右下两点坐标
94.         left = viewSize * LEFT;
95.         top = viewSize * TOP;
96.         right = viewSize * RIGHT;
97.         bottom = viewSize * BOTTOM;
98.
99.         // 计算粗线宽度
```

```
99.         thickLineWidth = viewSize * THICK_LINE_WIDTH;
100.
101.        // 计算细线宽度
102.        thinLineWidth = viewSize * THIN_LINE_WIDTH;
103.    }
104.
105.    @Override
106.    protected void onDraw(Canvas canvas) {
107.        // 填充背景
108.        canvas.drawColor(0xFF9596C4);
109.
110.        // 绘制标识元素
111.        drawSign(canvas);
112.
113.        // 绘制网格
114.        drawGrid(canvas);
115.
116.        // 绘制曲线
117.        drawPolyline(canvas);
118.    }
119.
120.    /**
121.     * 绘制曲线
122.     * 这里我使用一个新的 Bitmap 对象结合新的 Canvas 对象来绘制曲线
123.     * 当然你可以直接在原来的 canvas (onDraw 传来的那个) 中直接绘制如果你还没被坐标
124.     * 搞晕的话.....
125.     *
126.     * @param canvas
127.     *          画布
128.     */
129.    private void drawPolyline(Canvas canvas) {
130.        // 生成一个 Bitmap 对象大小和我们的网格大小一致
131.        mBitmap = Bitmap.createBitmap((int) (viewSize * (RIGHT - LEFT) - spaceX), (int) (viewSize * (BOTTOM - TOP) - spaceY), Bitmap.Config.ARGB_8888);
132.
133.        // 将 Bitmap 注入 Canvas
134.        mCanvas.setBitmap(mBitmap);
135.
136.        // 为画布填充一个半透明的红色
137.        mCanvas.drawRGB(75, 255, 0, 0);
138.
139.        // 重置曲线
```

```
140.         mPath.reset();
141.
142.         /*
143.          * 生成 Path 和绘制 Point
144.          */
145.         for (int i = 0; i < pointFs.size(); i++) {
146.             // 计算 x 坐标
147.             float x = mCanvas.getWidth() / maxX * pointFs.get(i).x;
148.
149.             // 计算 y 坐标
150.             float y = mCanvas.getHeight() / maxY * pointFs.get(i).y;
151.             y = mCanvas.getHeight() - y;
152.
153.             // 绘制小点点
154.             mCanvas.drawCircle(x, y, thickLineWidth, pointPaint);
155.
156.             /*
157.              * 如果是第一个点则将其设置为 Path 的起点
158.              */
159.             if (i == 0) {
160.                 mPath.moveTo(x, y);
161.             }
162.
163.             // 连接各点
164.             mPath.lineTo(x, y);
165.         }
166.
167.         // 设置 PathEffect
168.         // linePaint.setPathEffect(new CornerPathEffect(200));
169.
170.         // 重置线条宽度
171.         linePaint.setStrokeWidth(thickLineWidth);
172.
173.         // 将 Path 绘制到我们自定的 Canvas 上
174.         mCanvas.drawPath(mPath, linePaint);
175.
176.         // 将 mBitmap 绘制到原来的 canvas
177.         canvas.drawBitmap(mBitmap, left, top + spaceY, null);
178.     }
179.
180.     /**
181.      * 绘制网格
182.      *
183.      * @param canvas
```

```
184.     *          画布
185.     */
186.     private void drawGrid(Canvas canvas) {
187.         // 锁定画布
188.         canvas.save();
189.
190.         // 设置线条画笔宽度
191.         linePaint.setStrokeWidth(thickLineWidth);
192.
193.         // 计算 xy 轴 Path
194.         mPath.moveTo(left, top);
195.         mPath.lineTo(left, bottom);
196.         mPath.lineTo(right, bottom);
197.
198.         // 绘制 xy 轴
199.         canvas.drawPath(mPath, linePaint);
200.
201.         // 绘制线条
202.         drawLines(canvas);
203.
204.         // 释放画布
205.         canvas.restore();
206.     }
207.
208.     /**
209.      * 绘制网格
210.      *
211.      * @param canvas
212.      *          画布
213.      */
214.     private void drawLines(Canvas canvas) {
215.         // 计算刻度文字尺寸
216.         float textRulerSize = textSignSzie / 2F;
217.
218.         // 重置文字画笔文字尺寸
219.         mTextPaint.setTextSize(textRulerSize);
220.
221.         // 重置线条画笔描边宽度
222.         linePaint.setStrokeWidth(thinLineWidth);
223.
224.         // 获取数据长度
225.         int count = pointFs.size();
226.
227.         // 计算除数的值为数据长度减一
```

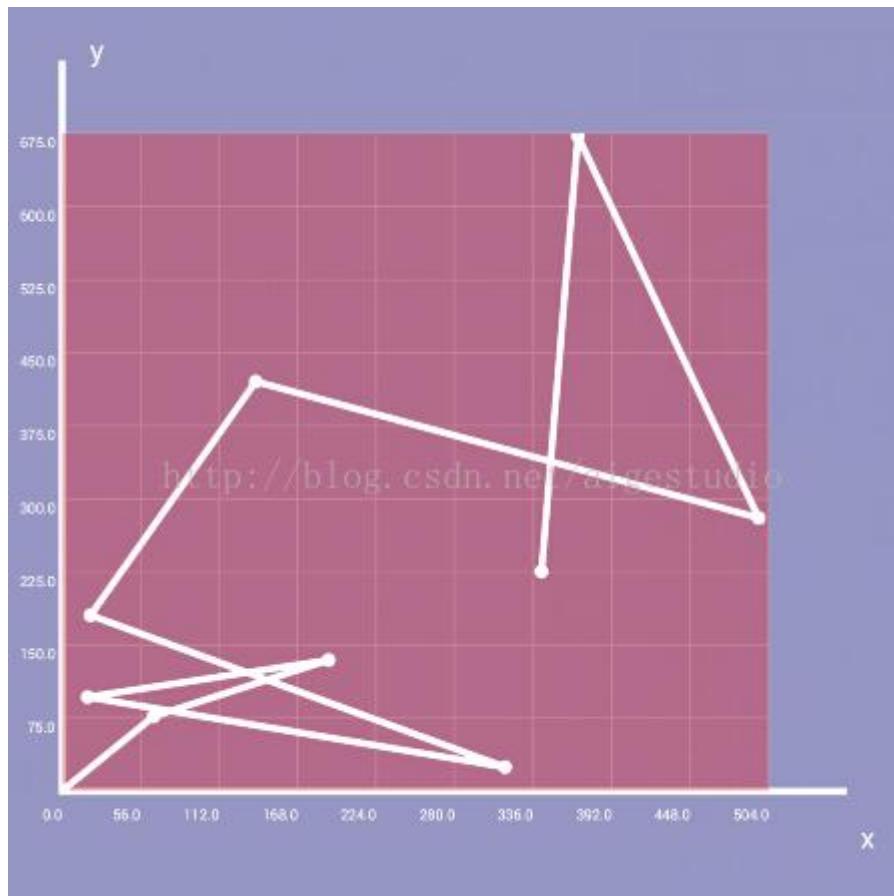
```
228.         int divisor = count - 1;
229.
230.         // 计算横轴数据最大值
231.         maxX = 0;
232.         for (int i = 0; i < count; i++) {
233.             if (maxX < pointFs.get(i).x) {
234.                 maxX = pointFs.get(i).x;
235.             }
236.         }
237.
238.         // 计算横轴最近的能被 count 整除的值
239.         int remainderX = ((int) maxX) % divisor;
240.         maxX = remainderX == 0 ? ((int) maxX) : divisor - remainderX + ((int)
241.             ) * maxX);
242.
243.         // 计算纵轴数据最大值
244.         maxY = 0;
245.         for (int i = 0; i < count; i++) {
246.             if (maxY < pointFs.get(i).y) {
247.                 maxY = pointFs.get(i).y;
248.             }
249.
250.         // 计算纵轴最近的能被 count 整除的值
251.         int remainderY = ((int) maxY) % divisor;
252.         maxY = remainderY == 0 ? ((int) maxY) : divisor - remainderY + ((int)
253.             ) * maxY);
254.
255.         // 生成横轴刻度值
256.         rulerX = new float[count];
257.         for (int i = 0; i < count; i++) {
258.             rulerX[i] = maxX / divisor * i;
259.
260.         // 生成纵轴刻度值
261.         rulerY = new float[count];
262.         for (int i = 0; i < count; i++) {
263.             rulerY[i] = maxY / divisor * i;
264.         }
265.
266.         // 计算横纵坐标刻度间隔
267.         spaceY = viewSize * (BOTTOM - TOP) / count;
268.         spaceX = viewSize * (RIGHT - LEFT) / count;
269.
```

```
270.         // 锁定画布并设置画布透明度为 75%
271.         int sc = canvas.saveLayerAlpha(0, 0, canvas.getWidth(), canvas.getHeight(), 75, Canvas.ALL_SAVE_FLAG);
272.
273.         // 绘制横纵线段
274.         for (float y = viewSize * BOTTOM - spaceY; y > viewSize * TOP; y -= spaceY) {
275.             for (float x = viewSize * LEFT; x < viewSize * RIGHT; x += spaceX) {
276.                 /*
277.                  * 绘制纵向线段
278.                  */
279.                 if (y == viewSize * TOP + spaceY) {
280.                     canvas.drawLine(x, y, x, y + spaceY * (count - 1), linePaint);
281.                 }
282.
283.                 /*
284.                  * 绘制横向线段
285.                  */
286.                 if (x == viewSize * RIGHT - spaceX) {
287.                     canvas.drawLine(x, y, x - spaceX * (count - 1), y, linePaint);
288.                 }
289.             }
290.         }
291.
292.         // 还原画布
293.         canvas.restoreToCount(sc);
294.
295.         // 绘制横纵轴向刻度值
296.         int index_x = 0, index_y = 1;
297.         for (float y = viewSize * BOTTOM - spaceY; y > viewSize * TOP; y -= spaceY) {
298.             for (float x = viewSize * LEFT; x < viewSize * RIGHT; x += spaceX) {
299.                 /*
300.                  * 绘制横轴刻度数值
301.                  */
302.                 if (y == viewSize * BOTTOM - spaceY) {
303.                     canvas.drawText(String.valueOf(rulerX[index_x]), x, y + textSignSzie + spaceY, mTextPaint);
304.                 }
305.             }
```

```
306.         /*
307.          * 绘制纵轴刻度数值
308.          */
309.         if (x == viewSize * LEFT) {
310.             canvas.drawText(String.valueOf(rulerY[index_y]), x - thickLineWidth, y + textRulerSize, mTextPaint);
311.         }
312.
313.         index_x++;
314.     }
315.     index_y++;
316. }
317. }
318.
319. /**
320.  * 绘制标识元素
321. *
322. * @param canvas
323. *          画布
324. */
325. private void drawSign(Canvas canvas) {
326.     // 锁定画布
327.     canvas.save();
328.
329.     // 设置文本画笔文字尺寸
330.     mTextPaint.setTextSize(textSignSzie);
331.
332.     // 绘制纵轴标识文字
333.     mTextPaint.setTextAlign(Paint.Align.LEFT);
334.     canvas.drawText(null == signY ? "y" : signY, textY_X, textY_Y, mTextPaint);
335.
336.     // 绘制横轴标识文字
337.     mTextPaint.setTextAlign(Paint.Align.RIGHT);
338.     canvas.drawText(null == signX ? "x" : signX, textX_X, textX_Y, mTextPaint);
339.
340.     // 释放画布
341.     canvas.restore();
342. }
343.
344. /**
345.  * 设置数据
346. *
```

```
347.     * @param pointFs
348.     *
349.     */
350.     public synchronized void setData(List<PointF> pointFs, String signX, St
ring signY) {
351.         /*
352.          * 数据为空直接 GG
353.          */
354.         if (null == pointFs || pointFs.size() == 0)
355.             throw new IllegalArgumentException("No data to display !");
356.
357.         /*
358.          * 控制数据长度不超过 10 个
359.          * 对于折线图来说数据太多就没必要用折线图表示了而是使用散点图
360.          */
361.         if (pointFs.size() > 10)
362.             throw new IllegalArgumentException("The data is too long to dis
play !");
363.
364.         // 设置数据并重绘视图
365.         this.pointFs = pointFs;
366.         this.signX = signX;
367.         this.signY = signY;
368.         invalidate();
369.     }
370. }
```

代码纯天然，我连封装都没有做什么 == So、你可以从代码中直接看到哥的思路~~如果没有设置数据，我这里给了一个初始化的随机数据，话说……随机生成的数据画出来的曲线挺带感的：



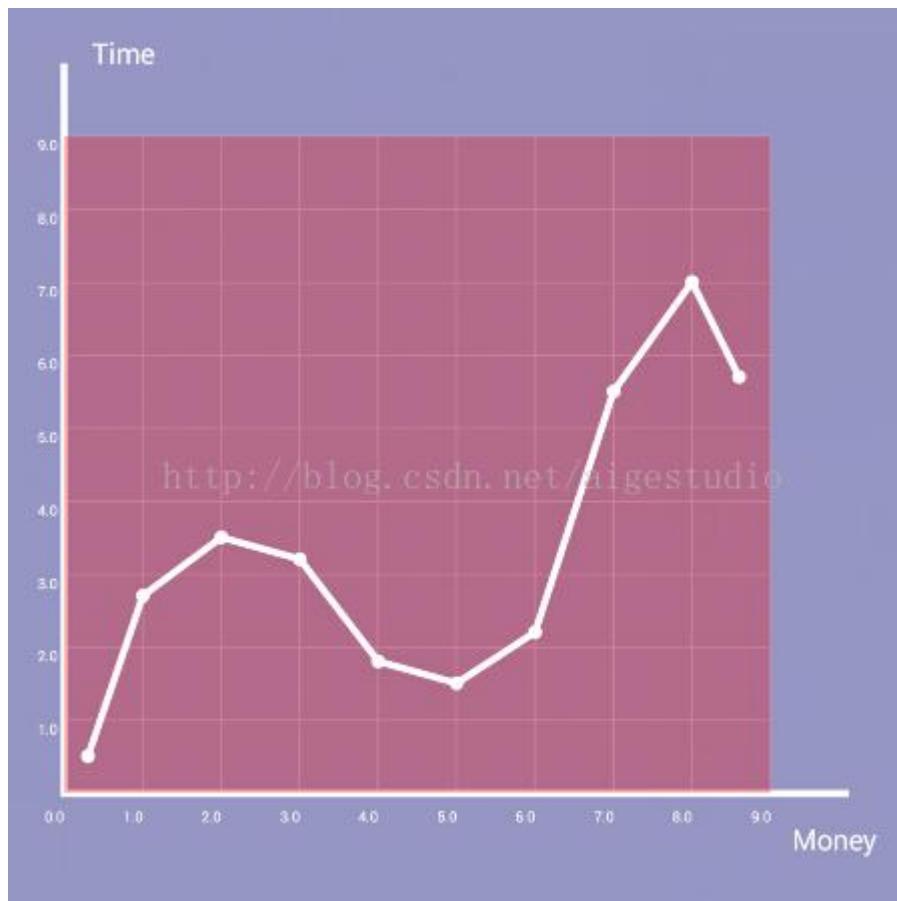
如果你想得到我们设计图的那种曲线，就需要自己去做特定的数据，实际应用中曲线的数据也肯定是特性的，比如天气-时间曲线图之类，这里的数据我们就直接在 `MainActivity` 中做：

[java] view plaincopyprint?

```
1. public class MainActivity extends Activity {
2.     private PolylineView mPolylineView;
3.
4.     @Override
5.     public void onCreate(Bundle savedInstanceState) {
6.         super.onCreate(savedInstanceState);
7.         setContentView(R.layout.activity_main);
8.
9.         mPolylineView = (PolylineView) findViewById(R.id.main_pv);
10.
11.        List<PointF> pointFs = new ArrayList<PointF>();
12.        pointFs.add(new PointF(0.3F, 0.5F));
13.        pointFs.add(new PointF(1F, 2.7F));
14.        pointFs.add(new PointF(2F, 3.5F));
15.        pointFs.add(new PointF(3F, 3.2F));
16.        pointFs.add(new PointF(4F, 1.8F));
17.        pointFs.add(new PointF(5F, 1.5F));
18.        pointFs.add(new PointF(6F, 2.2F));
```

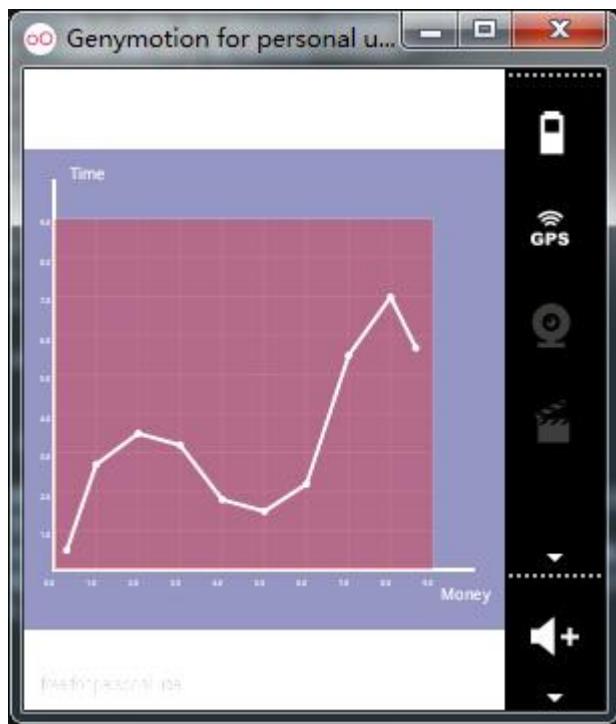
```
19.         pointFs.add(new PointF(7F, 5.5F));
20.         pointFs.add(new PointF(8F, 7F));
21.         pointFs.add(new PointF(8.6F, 5.7F));
22.
23.         mPolylineView.setData(pointFs, "Money", "Time");
24.     }
25. }
```

xml 里面的代码就不给了，运行效果如下：

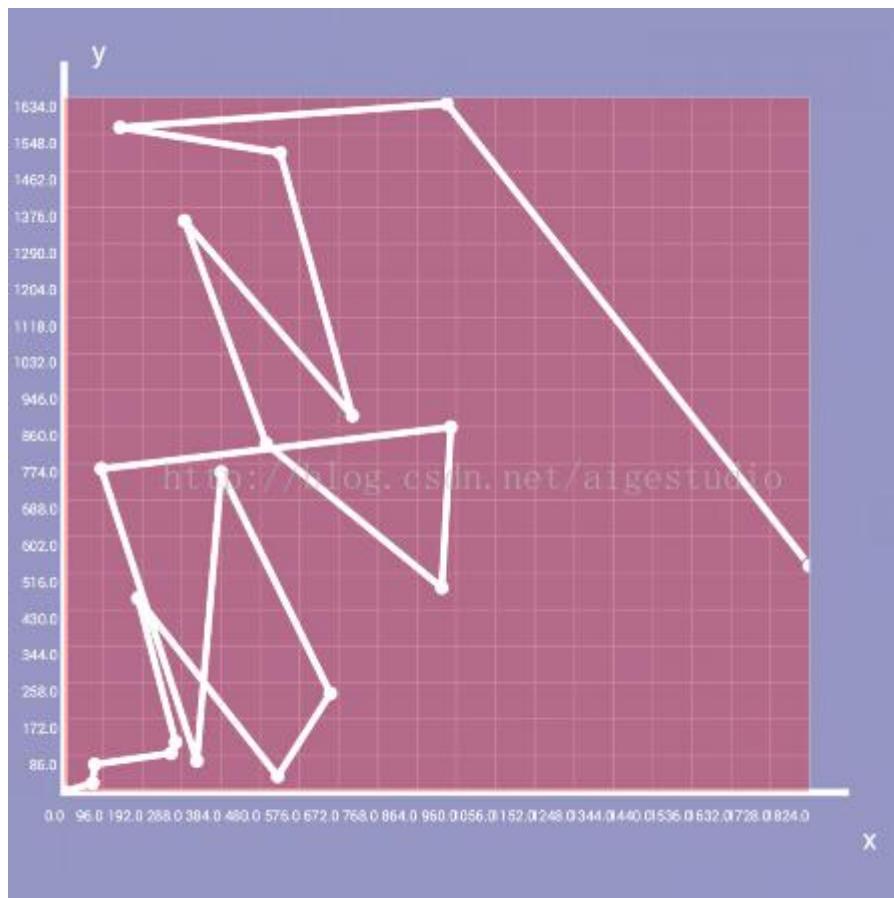


大家发现得出的曲线很生硬，在 1/4 中我们曾讲过 **PathEffect**，可以应用到这里，如果大家还不知道 **PathEffect**.....可以去看我前面的文章。

自定义控件很重要的一个地方就是屏幕的适配，我们以控件的边长作为基准参考可以避免很多的大小不一问题，上面的图我都是在 mx3 上截取的，mx3 分辨率高达 1800\*1080，我们可以换个手机测试下，以下是模拟器 240\*300 分辨率上的样子：



可以看到虽然刻度有点看不清了，但是整个控件的比例大小保持得很好。但是，如我所说，控件都是不完美的，如果能有完美的控件那就不需要我们自定义了，这个折线图控件也一样，首先它只能满足特定的数据，而且风格就是这样，如果我们把数据增多，比如 20 条数据：



可以看到轴上的刻度已经很紧凑了.....这时我们可以考虑控制刻度的位数或使用科学记数法等等，但是.....最有效的办法还是控制数据长度.....哟西！

简单地介绍了 **Path** 之后回到我们的 **Canvas** 中，关于裁剪的方法

[java] view plaincopyprint?

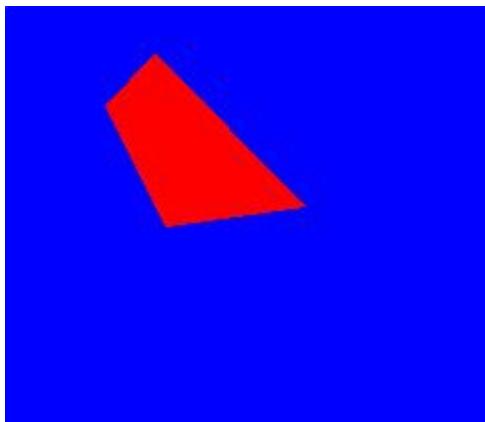
```
1. clipPath(Path path)
```

是不是变得透彻起来呢？

我们可以利用该方法从 **Canvas** 中“挖”取一块不规则的画布：

[java] view plaincopyprint?

```
1. public class CanvasView extends View {
2.     private Path mPath;
3.
4.     public CanvasView(Context context, AttributeSet attrs) {
5.         super(context, attrs);
6.
7.         mPath = new Path();
8.         mPath.moveTo(50, 50);
9.         mPath.lineTo(75, 23);
10.        mPath.lineTo(150, 100);
11.        mPath.lineTo(80, 110);
12.        mPath.close();
13.    }
14.
15.    @Override
16.    protected void onDraw(Canvas canvas) {
17.        canvas.drawColor(Color.BLUE);
18.
19.        canvas.clipPath(mPath);
20.
21.        canvas.drawColor(Color.RED);
22.    }
23. }
```



回顾 `Canvas` 中有关裁剪的方法，你会发现有一大堆带有 `Region.Op` 参数的重载方法：

[java] view plaincopyprint?

```
1. clipPath(Path path, Region.Op op)
2. clipRect(Rect rect, Region.Op op)
3. clipRect(RectF rect, Region.Op op)
4. clipRect(float left, float top, float right, float bottom, Region.Op op)
5. clipRegion(Region region, Region.Op op)
```

要明白这些方法的 `Region.Op` 参数那么首先要了解 `Region` 为何物。`Region` 的意思是“区域”，在 Android 里呢它同样表示的是一块封闭的区域，`Region` 中的方法都非常的简单，我们重点来瞧瞧 `Region.Op`，`Op` 是 `Region` 的一个枚举类，里面呢有六个枚举常量：

| Enum Values            |                    |
|------------------------|--------------------|
| <code>Region.Op</code> | DIFFERENCE         |
| <code>Region.Op</code> | INTERSECT          |
| <code>Region.Op</code> | REPLACE            |
| <code>Region.Op</code> | REVERSE_DIFFERENCE |
| <code>Region.Op</code> | UNION              |
| <code>Region.Op</code> | XOR                |

那么 `Region.Op` 究竟有什么用呢？其实它就是个组合模式，在 1/6 中我们曾学过一个叫图形混合模式的，而在本节开头我们也曾讲过 `Rect` 也有类似的组合方法，`Region.Op` 灰常简单，如果你看过 1/6 的图形混合模式的话。这里我就给出一段测试代码，大家可以尝试去改变不同的组合模式看看效果

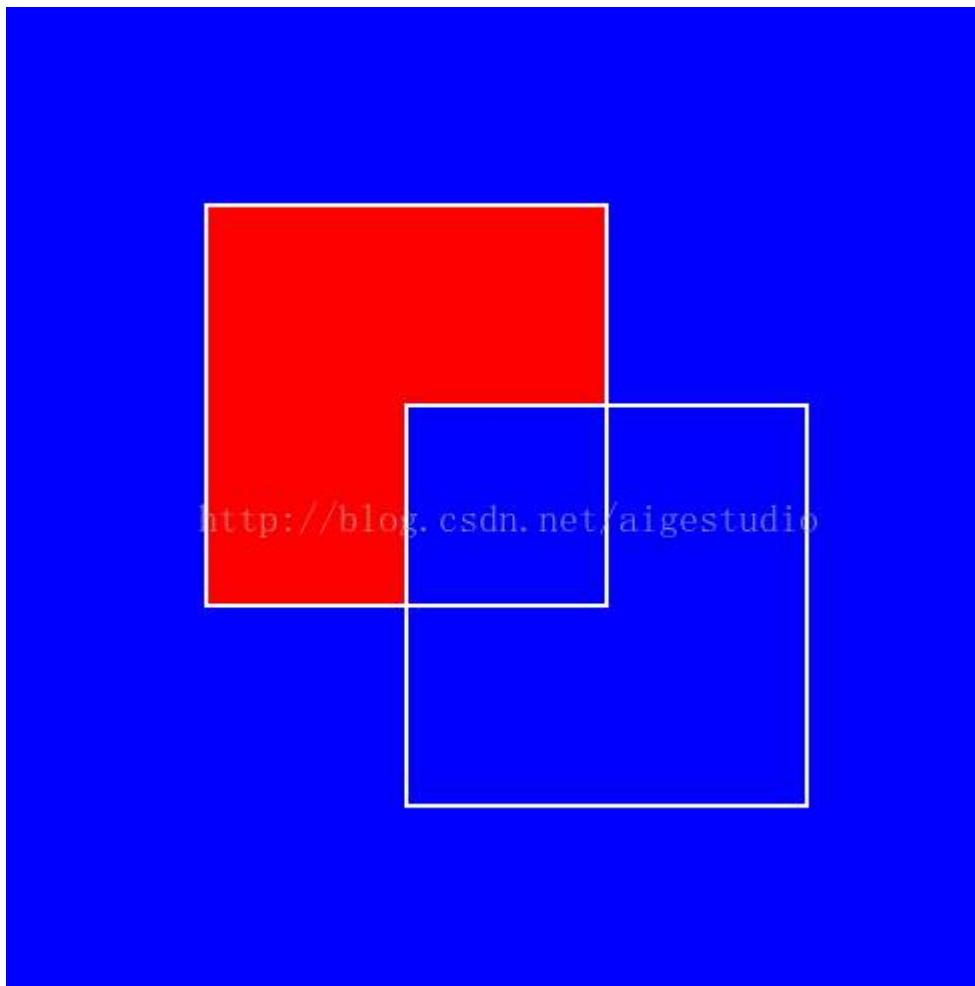
[java] view plaincopyprint?

```
1. public class CanvasView extends View {
2.     private Region mRegionA, mRegionB;// 区域 A 和区域 B 对象
```

```
3.     private Paint mPaint;// 绘制边框的 Paint
4.
5.     public CanvasView(Context context, AttributeSet attrs) {
6.         super(context, attrs);
7.
8.         // 实例化画笔并设置属性
9.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
10.        mPaint.setStyle(Paint.Style.STROKE);
11.        mPaint.setColor(Color.WHITE);
12.        mPaint.setStrokeWidth(2);
13.
14.        // 实例化区域 A 和区域 B
15.        mRegionA = new Region(100, 100, 300, 300);
16.        mRegionB = new Region(200, 200, 400, 400);
17.    }
18.
19.    @Override
20.    protected void onDraw(Canvas canvas) {
21.        // 填充颜色
22.        canvas.drawColor(Color.BLUE);
23.
24.        canvas.save();
25.
26.        // 裁剪区域 A
27.        canvas.clipRegion(mRegionA);
28.
29.        // 再通过组合方式裁剪区域 B
30.        canvas.clipRegion(mRegionB, Region.Op.DIFFERENCE);
31.
32.        // 填充颜色
33.        canvas.drawColor(Color.RED);
34.
35.        canvas.restore();
36.
37.        // 绘制框框帮助我们观察
38.        canvas.drawRect(100, 100, 300, 300, mPaint);
39.        canvas.drawRect(200, 200, 400, 400, mPaint);
40.    }
41. }
```

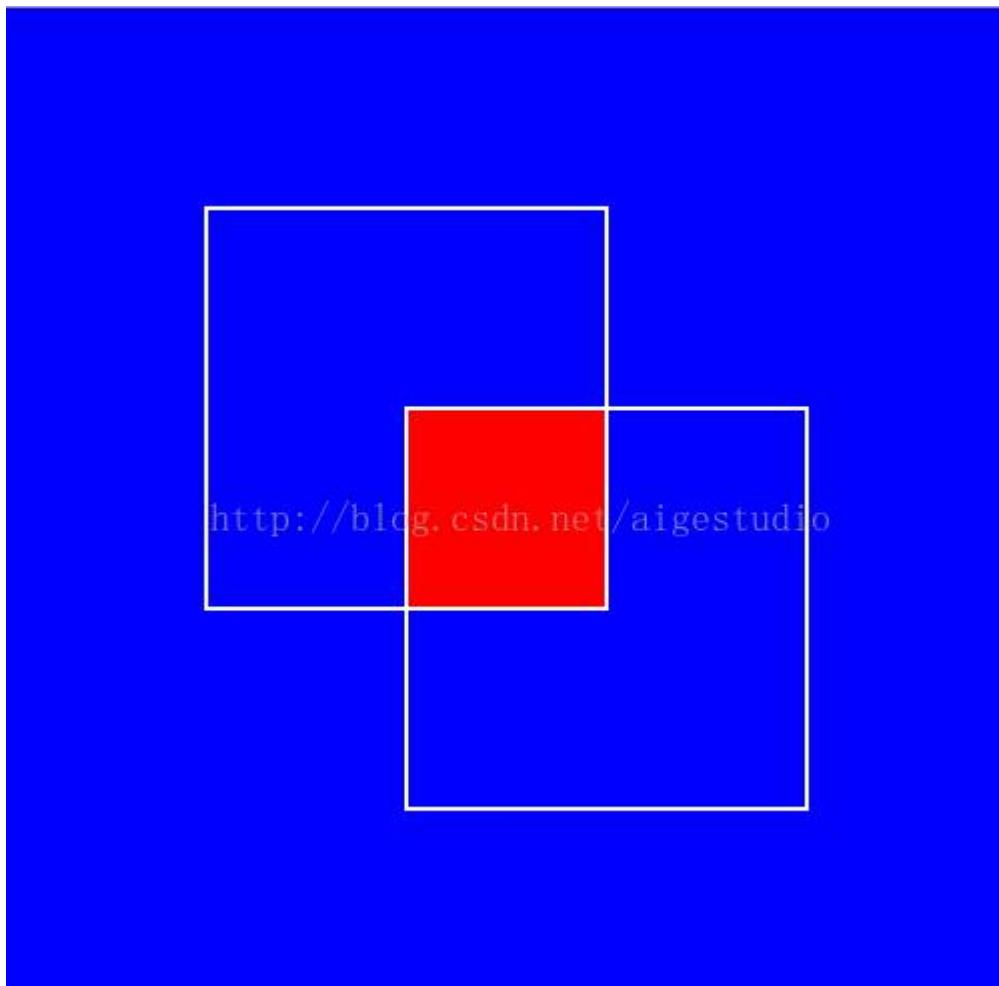
以下是各种组合模式的效果

DIFFERENCE



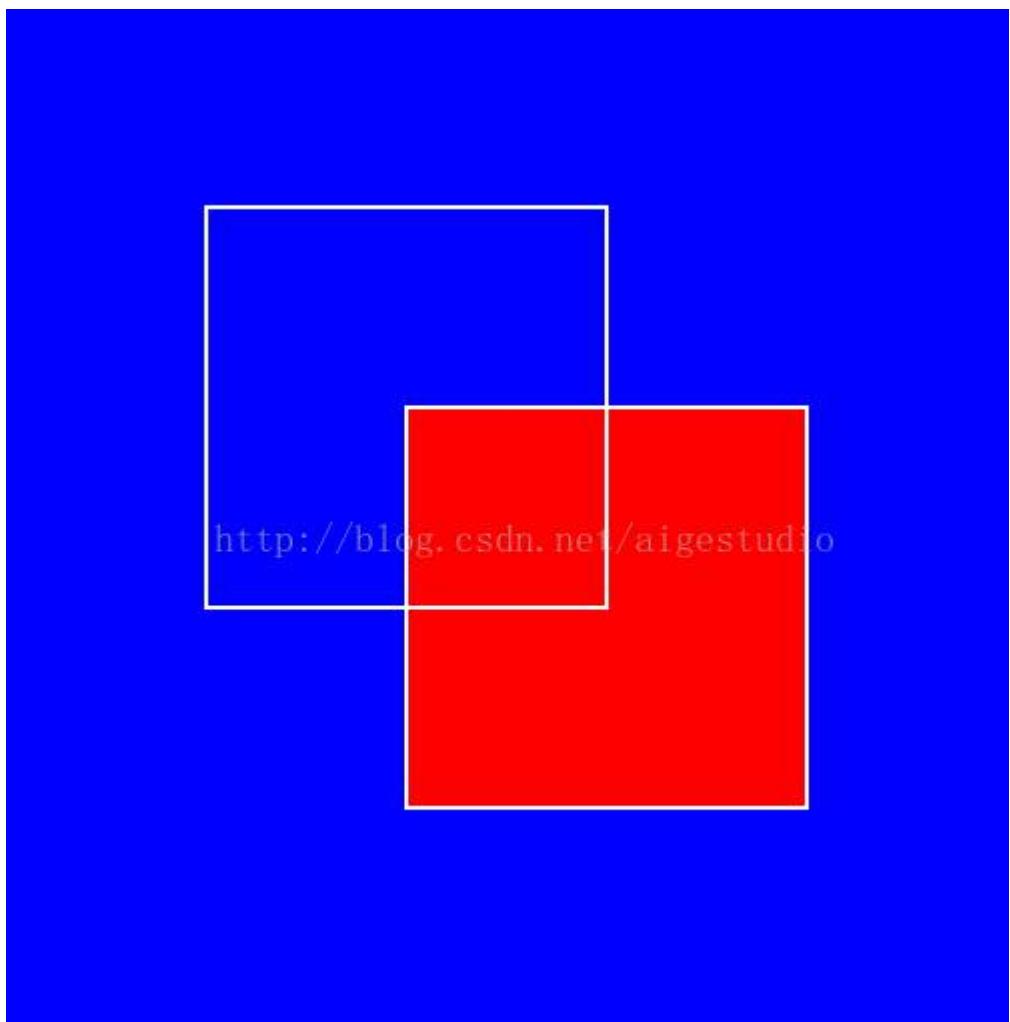
最终区域为第一个区域与第二个区域不同的区域。

**INTERSECT**



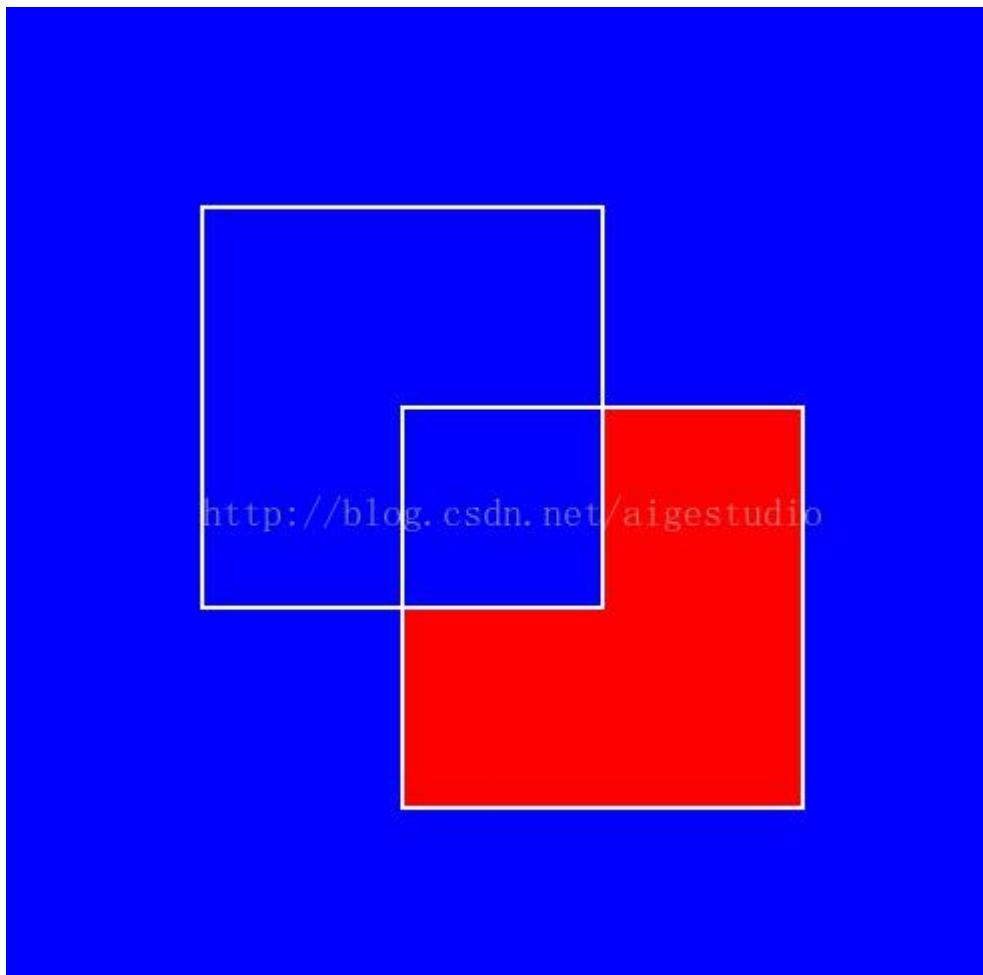
最终区域为第一个区域与第二个区域相交的区域。

REPLACE



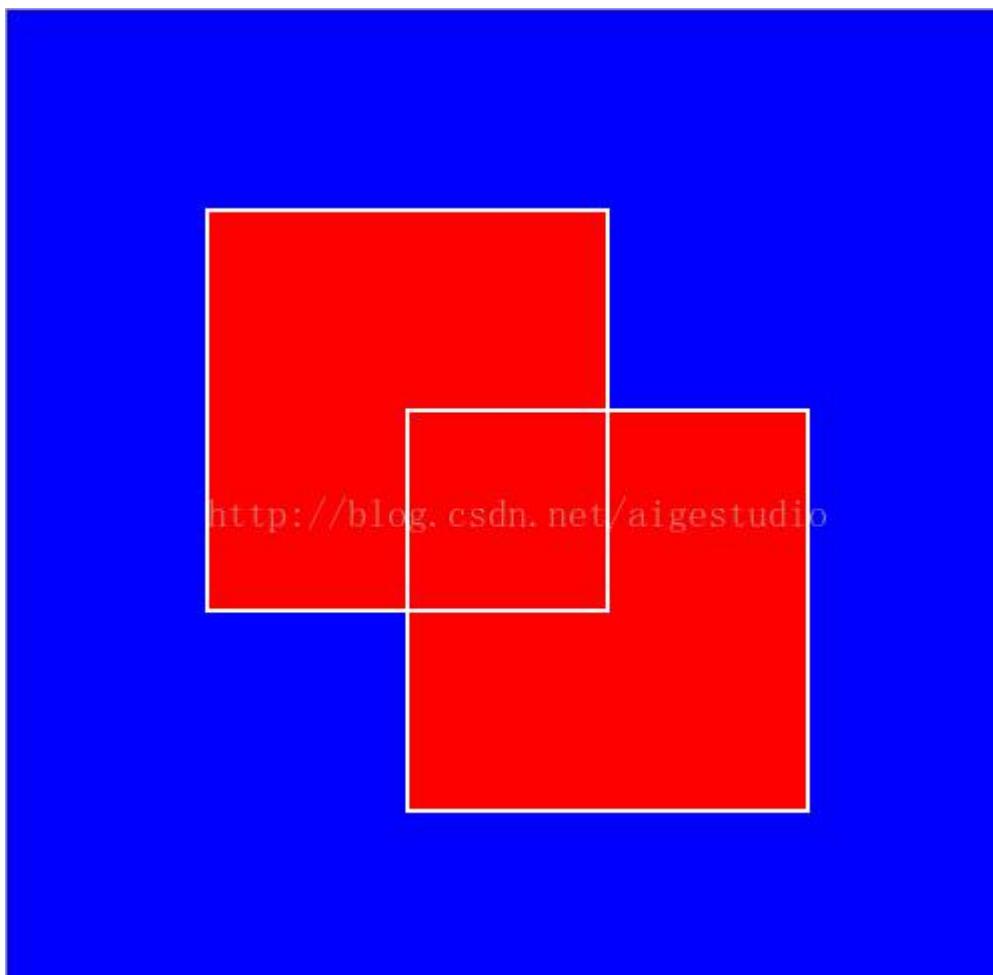
最终区域为第二个区域。

REVERSE\_DIFFERENCE



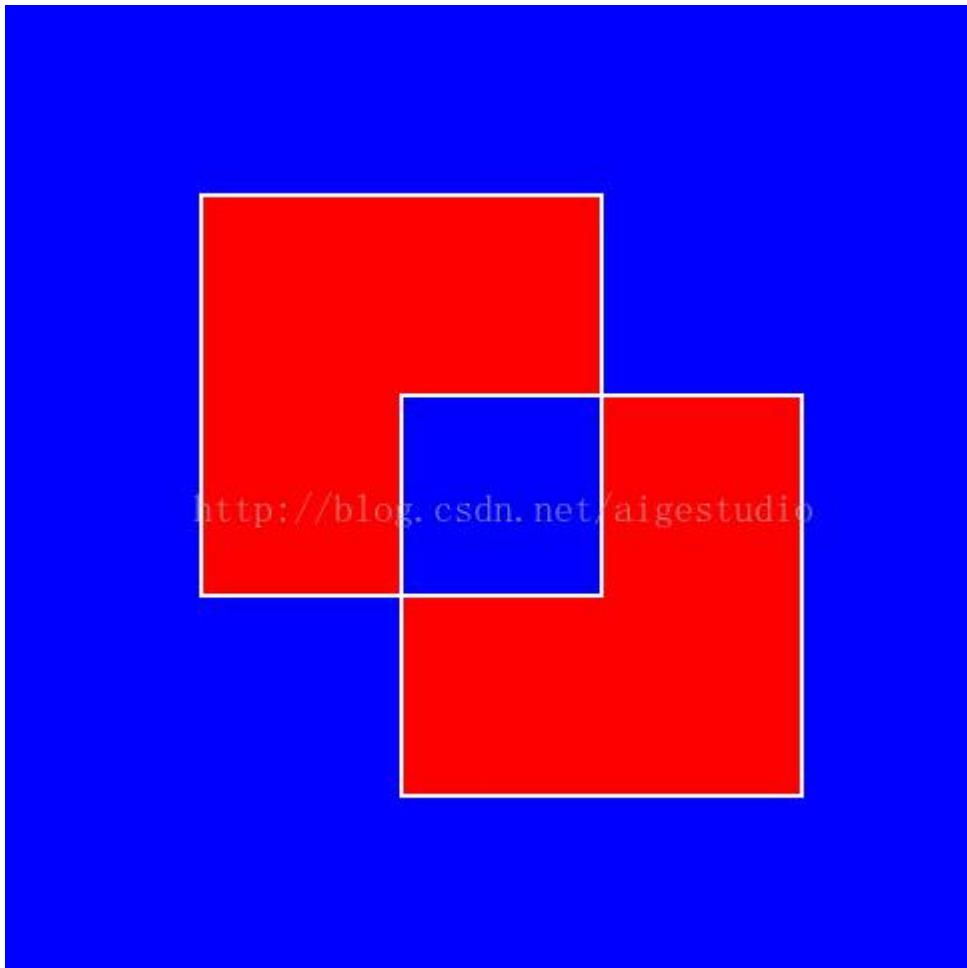
最终区域为第二个区域与第一个区域不同的区域。

UNION



最终区域为第一个区域加第二个区域。

XOR



最终区域为第一个区域加第二个区域并减去两者相交的区域。

Region.Op 就是这样，它和我们之前讲到的图形混合模式几乎一模一样换汤不换药……我在做示例的时候仅仅是使用了一个 Region，实际上 Rect、Cricle、Ovel 等封闭的曲线都可以使用 Region.Op，介于篇幅，而且也不难以理解就不多说了。

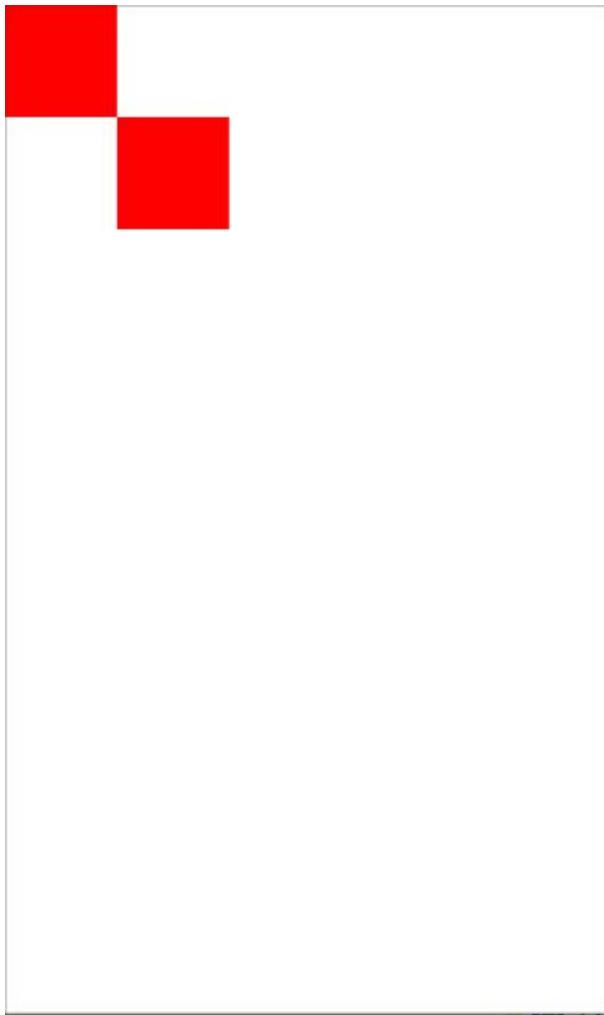
有些童鞋会问那么 Region 和 Rect 有什么区别呢？首先最重要的一点，Region 表示的是一个区域，而 Rect 表示的是一个矩形，这是最根本的区别之一，其次，Region 有个很特别的地方是它不受 Canvas 的变换影响，Canvas 的 local 不会直接影响到 Region 自身，什么意思呢？我们来看一个 simple 你就会明白：

[java] view plaincopyprint?

```
1. public class CanvasView extends View {  
2.     private Region mRegion;// 区域对象  
3.     private Rect mRect;// 矩形对象  
4.     private Paint mPaint;// 绘制边框的 Paint  
5.  
6.     public CanvasView(Context context, AttributeSet attrs) {  
7.         super(context, attrs);  
8.     }
```

```
9.         // 实例化画笔并设置属性
10.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
11.        mPaint.setStyle(Paint.Style.STROKE);
12.        mPaint.setColor(Color.DKGRAY);
13.        mPaint.setStrokeWidth(2);
14.
15.        // 实例化矩形对象
16.        mRect = new Rect(0, 0, 200, 200);
17.
18.        // 实例化区域对象
19.        mRegion = new Region(200, 200, 400, 400);
20.    }
21.
22.    @Override
23.    protected void onDraw(Canvas canvas) {
24.        canvas.save();
25.
26.        // 裁剪矩形
27.        canvas.clipRect(mRect);
28.        canvas.drawColor(Color.RED);
29.
30.        canvas.restore();
31.
32.        canvas.save();
33.
34.        // 裁剪区域
35.        canvas.clipRegion(mRegion);
36.        canvas.drawColor(Color.RED);
37.
38.        canvas.restore();
39.
40.        // 为画布绘制一个边框便于观察
41.        canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(), mPaint)
42.    }
43. }
```

大家看到，我在[0, 0, 200, 200]和[200, 200, 400, 400]的位置分别绘制了 Rect 和 Region，它们两个所占大小是一样的：



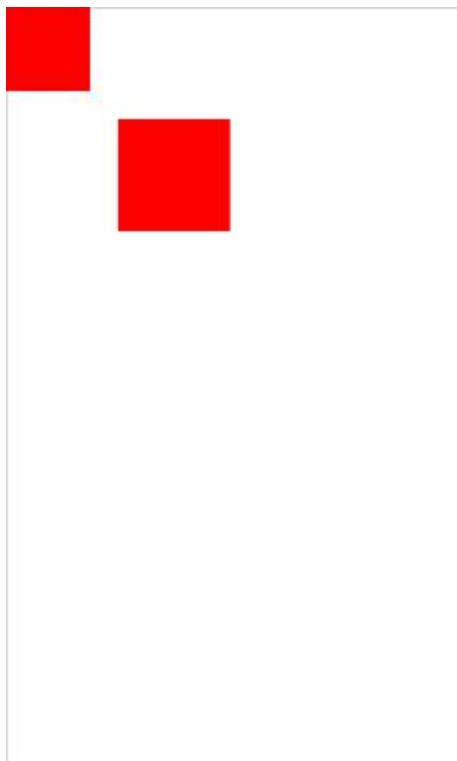
画布因为和屏幕一样大，so~我们看不出描边的效果，这时，我们将 Canvas 缩放至 75%大小，看看会发生什么：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      // 缩放画布
4.      canvas.scale(0.75F, 0.75F);
5.
6.      canvas.save();
7.
8.      // 裁剪矩形
9.      canvas.clipRect(mRect);
10.     canvas.drawColor(Color.RED);
11.
12.     canvas.restore();
13.
14.     canvas.save();
```

```
16.     // 裁剪区域
17.     canvas.clipRegion(mRegion);
18.     canvas.drawColor(Color.RED);
19.
20.     canvas.restore();
21.
22.     // 为画布绘制一个边框便于观察
23.     canvas.drawRect(0, 0, canvas.getWidth(), canvas.getHeight(), mPaint);
24. }
```

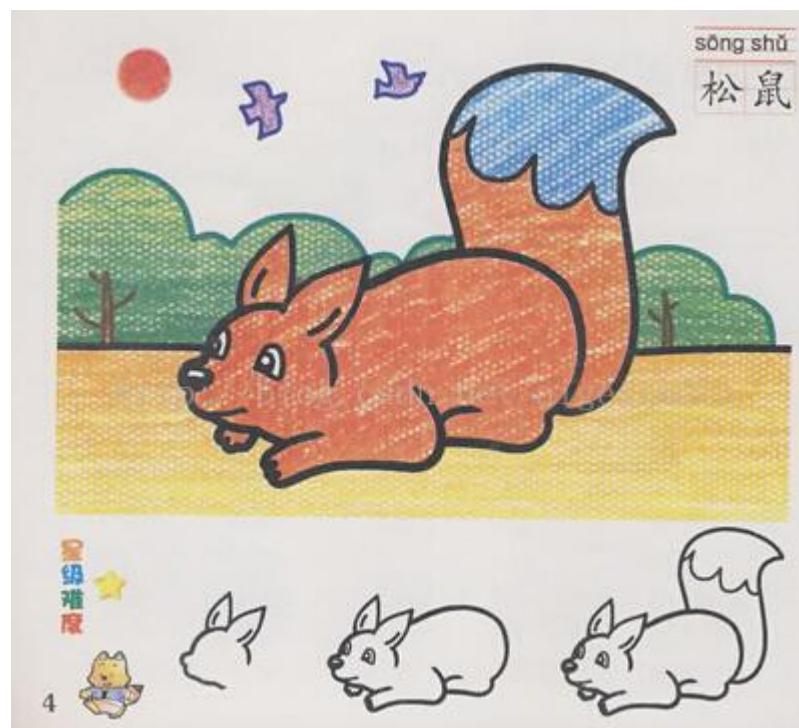
这时我们会看到, Rect 随着 Canvas 的缩放一起缩放了,但是 Region 依旧泰山不动地淡定:



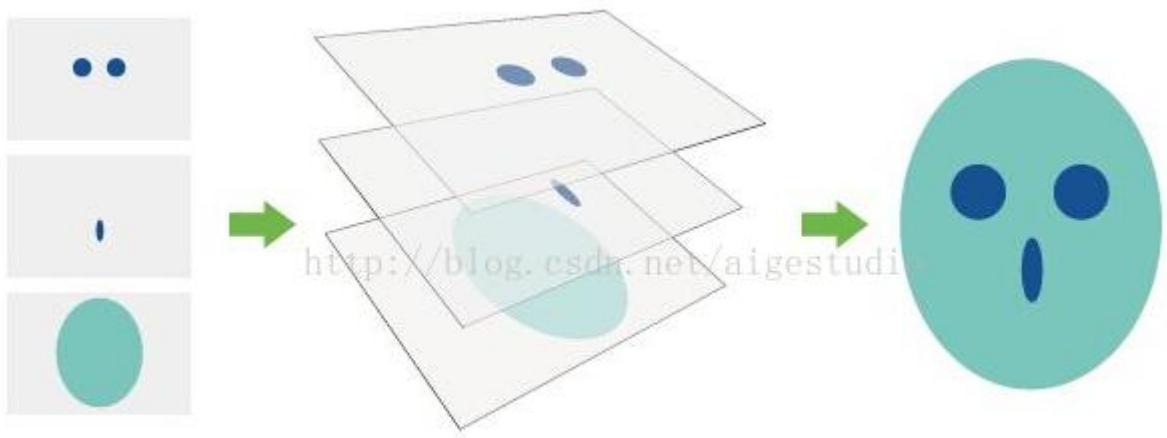
呼呼呼.....关于 Canvas 的一部分内容就先介绍到此, Canvas 的内容比我想象的还要多啊啊啊啊啊啊啊啊啊！！！！！主要是 Canvas 涉及不少的擦边球类一写根本停不下来, 妈蛋！！！！

## 6. 自定义控件其实很简单 (6)

年关将至事情巨多，最近因为安排蓄谋已久旅行事宜很久没更我们的系列教程，约莫着有一个月了，这事情多起来啊闲都闲不下来~~那么我们闲话少说，来看看这一节我们的重点，上一节因为之前从未涉及 **Canvas** 的 **clipXXX** 方法所以我们优先对其做了一定的介绍并顺带将 **Path** 类的方法做了一个小结，如我之前所说 **Canvas** 方法可以分为几类，**clipXXX** 算一类，各种 **drawXXX** 又是一类，还有一类则是对 **Canvas** 的各种变换操作，这一节我们将来具体看看关于 **Canvas** 变换操作的一些具体内容，在讲解之前呢我们先来了解一个关于“层”的设计理念，为什么说它是一个设计理念呢？因为在很多很多的地方，当然不止是开发，还有设计等领域你都能见到它的踪影，那么何为图层呢？大家小时候一定都画过画，比如下面这种：



一个松鼠、几棵树、两个鸟、一个日，这几个简单的图画其实就包含了最简单“层”的概念，由图我们可以知道松鼠一定是在树和地面的前面，而树和地面的关系呢则比较模糊，可以是树在前也可以是地面向前，日肯定是在最底层的，两只鸟我们按照一般逻辑可以推测在日的上一层也就是倒数第二层，那么从底层到顶层我们就有这样的一个层次关系：日-鸟-树/地面-地面/树-松鼠，这么一说大家觉得好像也是，但是目测没毛用啊……意义何在，别急，想像一下，这时候如果你不想要松鼠而是想放一只猫在前面……或者你想把松鼠放在树的后面“藏”起来……这时你就蛋疼了，不停地拿橡皮擦擦啊擦草啊草，一不小心还得把其他的擦掉一块，这时候你就会想可以不可以有这么一个功能能让不同的元素通过一定的次序单独地画在一张大小一致“纸”上直到画完最后一个元素后把这些所有“纸”上的元素都整合起来构成一幅完整的图画呢？这样一个功能的存在能大大提高我们绘图的效率还能实现更多的绘图功能，基于这样的一个假想，“层”的概念应运而生：



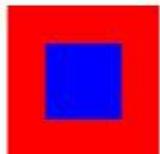
如上图所示，位于最底层的是一个圆，第二层是一个蓝色的椭圆，最顶层的是两个蓝色的圆，三个层中不同的元素最终构成右边的图像，这就是图层最直观也是最简单的体现。在Android中我们可以使用Canvas的saveXXX和restoreXXX方法来模拟图层的类似效果：

[java] view plaincopyprint?

```
1. public class LayerView extends View {  
2.     private Paint mPaint;// 画笔对象  
3.  
4.     private int mViewWidth, mViewHeight;// 控件宽高  
5.  
6.     public LayerView(Context context, AttributeSet attrs) {  
7.         super(context, attrs);  
8.  
9.         // 实例化画笔对象并设置其标识值  
10.        mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);  
11.    }  
12.  
13.    @Override  
14.    protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
15.        /*  
16.         * 获取控件宽高  
17.         */  
18.        mViewWidth = w;  
19.        mViewHeight = h;  
20.    }  
21.  
22.    @Override  
23.    protected void onDraw(Canvas canvas) {  
24.        /*  
25.         * 绘制一个红色矩形  
26.         */  
27.    }  
28.}
```

```
26.         */
27.         mPaint.setColor(Color.RED);
28.         canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mView
   Width / 2F + 200, mViewHeight / 2F + 200, mPaint);
29.
30.         /*
31.          * 保存画布并绘制一个蓝色的矩形
32.          */
33.         canvas.save();
34.         mPaint.setColor(Color.BLUE);
35.         canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mView
   Width / 2F + 100, mViewHeight / 2F + 100, mPaint);
36.         canvas.restore();
37.     }
38. }
```

如代码所示，我们先在 `onDraw` 方法中绘制一个红色的大矩形再保存画布绘制了一个蓝色的小矩形：



此时我们尝试旋转一下我们的画布：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     // 旋转画布
4.     canvas.rotate(30);
5.
```

```
6.      /*
7.       * 绘制一个红色矩形
8.       */
9.      mPaint.setColor(Color.RED);
10.     canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
11.
12.      /*
13.       * 保存画布并绘制一个蓝色的矩形
14.       */
15.      canvas.save();
16.      mPaint.setColor(Color.BLUE);
17.      canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
18.      canvas.restore();
19. }
```

如代码所示顺时针旋转 30 度，这里要注意，我们在对 Canvas（实际上大多数 Android 中的其他与坐标有关的）进行坐标操作的时候，默认情况下是以控件的左上角为原点坐标的，效果如下：



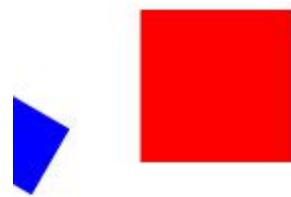
可以看到两个矩形都一起飞了，可是我们只想让蓝色的飞而红色的不动怎么办呢？很简单，我们只在保存的图层里操作即可：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
```

```
3.      /*
4.       * 绘制一个红色矩形
5.       */
6.      mPaint.setColor(Color.RED);
7.      canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8.
9.      /*
10.     * 保存画布并绘制一个蓝色的矩形
11.     */
12.     canvas.save();
13.     mPaint.setColor(Color.BLUE);
14.
15.     // 旋转画布
16.     canvas.rotate(30);
17.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
18.     canvas.restore();
19. }
```

可以看到，我们只针对蓝色的矩形进行了旋转：



至此结合上一节对 **Canvas** 的一些原理阐述我们该对它有个全新的认识，之前我们一直称其为画布，其实更准确地说 **Canvas** 是一个容器，如果把 **Canvas** 理解成画板，那么我们的“层”就像张张夹在画板上的透明的纸，而这些纸对应到 **Android** 则是一个个封装在 **Canvas** 中的 **Bitmap**。

除了 `save()` 方法 `Canvas` 还给我们提供了一系列的 `saveLayerXXX` 方法给我们保存画布，与 `save()` 方法不同的是，`saveLayerXXX` 方法会将所有的操作存到一个新的 `Bitmap` 中而不影响当前 `Canvas` 的 `Bitmap`，而 `save()` 方法则是在当前的 `Bitmap` 中进行操作，并且只能针对 `Bitmap` 的形变和裁剪进行操作，`saveLayerXXX` 方法则无所不能，当然两者还有很多的不同，我们稍作讲解。虽然 `save` 和 `saveLayerXXX` 方法有着很大的区别但是在一般应用上两者能实现的功能是差不多，上面的代码我们也可以改成这样：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      /*
4.       * 绘制一个红色矩形
5.       */
6.      mPaint.setColor(Color.RED);
7.      canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8.
9.      /*
10.       * 保存画布并绘制一个蓝色的矩形
11.       */
12.      canvas.saveLayer(0, 0, mViewWidth, mViewHeight, null, Canvas.ALL_SAVE_FLAG);
13.      mPaint.setColor(Color.BLUE);
14.
15.      // 旋转画布
16.      canvas.rotate(30);
17.      canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
18.      canvas.restore();
19. }
```

当然实现的效果也是一样的就不多说了。`saveLayer` 可以让我们自行设定需要保存的区域，比如我们可以只保存和蓝色方块一样的区域：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      /*
4.       * 绘制一个红色矩形
5.       */
6.      mPaint.setColor(Color.RED);
```

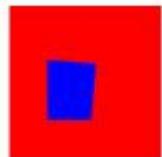
```
7.     canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8.
9.     /*
10.      * 保存画布并绘制一个蓝色的矩形
11.      */
12.     canvas.saveLayer(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, null, Canvas.ALL_SAVE_FLAG);
13.     mPaint.setColor(Color.BLUE);
14.
15.     // 旋转画布
16.     canvas.rotate(30);
17.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
18.     canvas.restore();
19. }
```

这时候如果你运行就会发现蓝色的方块已经不见了，因为我们图层的大小就这么点，超出的部分就不能被显示了，这时我们改小画布旋转：

[java] view plaincopyprint?

```
1. canvas.rotate(5);
```

你就可以看到旋转后的蓝色方块的一角：



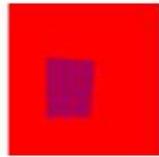
是不是有点类似于 `clipRect` 的效果呢？那么很多朋友会好奇为什么会有这样一种保存一小块画布区域的功能呢？其实原因很简单，上面我们说了 `saveLayerXXX` 方法会将操作保存到一个新的 `Bitmap` 中，而这个 `Bitmap` 的大小取决于我们传入的参数大小，`Bitmap` 是个相当危险的对象，很多朋友在操作 `Bitmap` 时不太理解其原理经常导致 OOM，在 `saveLayer` 时我们会依据传入的参数获取一个相同大小的 `Bitmap`，虽然这个 `Bitmap` 是空的但是其会占用一定的内存空间，我们希望尽可能小地保存该保存的区域，而 `saveLayer` 则提供了这样的功能，顺带提一下，`onDraw` 方法传入的 `Canvas` 对象的 `Bitmap` 在 Android 没引入 HW 之前理论上是无限大的，实际上其依然是根据你的图像来不断计算的，而在引入 HW 之后，该 `Bitmap` 受到限制，具体多大大家可以尝试画一个超长的 `path` 运行下你就可以在 `Logcat` 中看到 warning。

好了，闲话不扯，接着说，除了 `saveLayer`，`Canvas` 还提供了一个 `saveLayerAlpha` 方法，顾名思义，该方法可以在我们保存画布时设置画布的透明度：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      /*
4.       * 绘制一个红色矩形
5.       */
6.      mPaint.setColor(Color.RED);
7.      canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8.
9.      /*
10.       * 保存画布并绘制一个蓝色的矩形
11.       */
12.      canvas.saveLayerAlpha(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, 0x55, Canvas.ALL_SAVE_FLAG);
13.      mPaint.setColor(Color.BLUE);
14.
15.      // 旋转画布
16.      canvas.rotate(5);
17.      canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
18.      canvas.restore();
19. }
```

我们将 `saveLayer` 替换成 `saveLayerAlpha` 并设置透明值为 `0x55`，运行可得如下效果：



可见蓝色的方块被半透明了。such easy! 如果大家留心，会发现 `save()` 也有个重载方法 `save(int saveFlags)`，而 `saveLayer` 和 `saveLayerAlpha` 你也会发现又一个类似的参数，那么这个参数是干嘛用的呢？在 `Canvas` 中有六个常量值：

| Constants |                                         |                                                                  |
|-----------|-----------------------------------------|------------------------------------------------------------------|
| int       | <code>ALL_SAVE_FLAG</code>              | restore everything when <code>restore()</code> is called         |
| int       | <code>CLIP_SAVE_FLAG</code>             | restore the current clip when <code>restore()</code> is called   |
| int       | <code>CLIP_TO_LAYER_SAVE_FLAG</code>    | clip against the layer's bounds                                  |
| int       | <code>FULL_COLOR_LAYER_SAVE_FLAG</code> | the layer needs to 8-bits per color component                    |
| int       | <code>HAS_ALPHA_LAYER_SAVE_FLAG</code>  | the layer needs to per-pixel alpha                               |
| int       | <code>MATRIX_SAVE_FLAG</code>           | restore the current matrix when <code>restore()</code> is called |

这六个常量值分别标识了我们在调用 `restore` 方法后还原什么，六个标识位除了 `CLIP_SAVE_FLAG`、`MATRIX_SAVE_FLAG` 和 `ALL_SAVE_FLAG` 是 `save` 和 `saveLayerXXX` 方法都通用外其余三个只能使 `saveLayerXXX` 方法有效，`ALL_SAVE_FLAG` 很简单也是我们新手级常用的标识保存所有，`CLIP_SAVE_FLAG` 和 `MATRIX_SAVE_FLAG` 也很好理解，一个是裁剪的标识位一个是变换的标识位，`CLIP_TO_LAYER_SAVE_FLAG`、`FULL_COLOR_LAYER_SAVE_FLAG` 和 `HAS_ALPHA_LAYER_SAVE_FLAG` 只对 `saveLayer` 和 `saveLayerAlpha` 有效，`CLIP_TO_LAYER_SAVE_FLAG` 表示对当前图层执行裁剪操作需要对齐图层边界，`FULL_COLOR_LAYER_SAVE_FLAG` 表示当前图层的色彩模式至少需要是 8 位色，而 `HAS_ALPHA_LAYER_SAVE_FLAG` 表示在当前图层中将需要使用逐像素 Alpha 混合模式，关于色彩深度和 Alpha 混合大家可以参考维基百科，这里就不多说，这些标识位，特别是 `layer` 的标识位，大大超出了本系列的范畴，我就不多说了，

平时使用大家可以直接 ALL\_SAVE\_FLAG，有机会将单独开一篇剖析 Android 对色彩的处理。

所有的 save、saveLayer 和 saveLayerAlpha 方法都有一个 int 型的返回值，该返回值作为一个标识给与了一个你当前保存操作的唯一 ID 编号，我们可以利用 restoreToCount(int saveCount)方法来指定在还原的时候还原哪一个保存操作：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      /*
4.       * 绘制一个红色矩形
5.       */
6.      mPaint.setColor(Color.RED);
7.      canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8.
9.      /*
10.       * 保存并裁剪画布填充绿色
11.       */
12.      int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
13.      canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200);
14.      canvas.drawColor(Color.GREEN);
15.
16.      /*
17.       * 保存画布并旋转后绘制一个蓝色的矩形
18.       */
19.      int saveID2 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
20.
21.      // 旋转画布
22.      canvas.rotate(5);
23.      mPaint.setColor(Color.BLUE);
24.      canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
25.
26.      canvas.restoreToCount(saveID1);
27. }
```

如上代码所示，我们第一次保存画布并获取其返回值：

[java] view plaincopyprint?

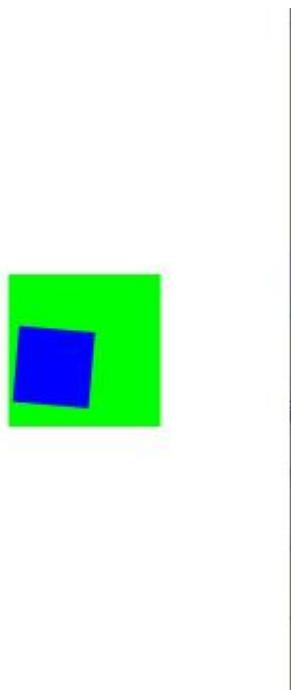
```
1.  int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
```

然后对画布进行裁剪并填色，第二次保存画布并获取其返回值：

[java] view plaincopyprint?

```
1. int saveID2 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
```

然后绘制一个蓝色的矩形，最后我们只还原了 saveID1 的画布状态，运行一下你会发现好像效果没什么不同啊：



然后我们试试

[java] view plaincopyprint?

```
1. canvas.restoreToCount(saveID2);
```

发现效果还是一样…………很多童鞋就困惑了，是哪不对么？没有，其实都是对的，你觉得奇怪是你还不理解 save 和 restore，这里我在 restore 之后再绘制一个矩形：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     /*
4.      * 绘制一个红色矩形
5.     */
6.     mPaint.setColor(Color.RED);
7.     canvas.drawRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
8. }
```

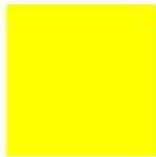
```
9.      /*
10.         * 保存并裁剪画布填充绿色
11.         */
12.     int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
13.     canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth
14.                     / 2F + 200, mViewHeight / 2F + 200);
15.     canvas.drawColor(Color.GREEN);
16.     /*
17.         * 保存画布并旋转后绘制一个蓝色的矩形
18.         */
19.     int saveID2 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
20.
21.     // 旋转画布
22.     canvas.rotate(5);
23.     mPaint.setColor(Color.BLUE);
24.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth
25.                     / 2F + 100, mViewHeight / 2F + 100, mPaint);
26.
27.     canvas.restoreToCount(saveID2);
28.
29.     mPaint.setColor(Color.YELLOW);
30.     canvas.drawRect(mViewWidth / 2F - 400, mViewHeight / 2F - 400, mViewWidth
31.                     / 2F + 400, mViewHeight / 2F + 400, mPaint);
```

可以看到我在

[java] view plaincopyprint?

```
1. canvas.restoreToCount(saveID2);
```

之后又绘制了一个黄色的矩形：



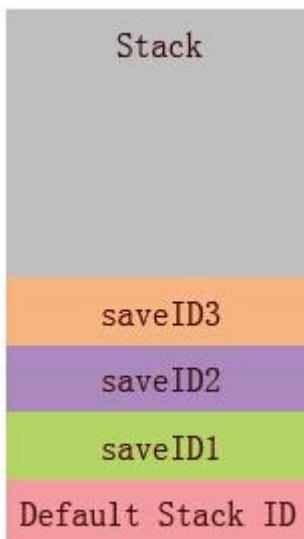
可是不管你如何调大这个矩形，你会发现它就那么大点……也就是说，这个黄色的矩形其实是被 `clip` 掉了，进一步说，我们绘制黄色矩形的这个操作其实说白了就是在 `saveID1` 的状态下进行的。前面我们曾说过 `save` 和 `saveLayerXXX` 方法有着本质的区别，`saveLayerXXX` 方法会将所有操作在一个新的 `Bitmap` 中进行，而 `save` 则是依靠 `stack` 栈来进行，假设我们有如下代码：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      /*
4.       * 保存并裁剪画布填充绿色
5.       */
6.      int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
7.      canvas.clipRect(mViewWidth / 2F - 300, mViewHeight / 2F - 300, mViewWidth / 2F + 300, mViewHeight / 2F + 300);
8.      canvas.drawColor(Color.YELLOW);
9.
10.     /*
11.      * 保存并裁剪画布填充绿色
12.      */
13.     int saveID2 = canvas.save(Canvas.CLIP_SAVE_FLAG);
14.     canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200);
15.     canvas.drawColor(Color.GREEN);
16.
17.     */
```

```
18.     * 保存画布并旋转后绘制一个蓝色的矩形
19.     */
20.     int saveID3 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
21.     canvas.rotate(5);
22.     mPaint.setColor(Color.BLUE);
23.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
24. }
```

此时，在 Canvas 内部会有这样的一个 Stack 栈：



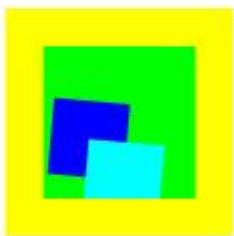
Canvas 会默认保存一个底层的空间给我们绘制一些东西，当我们没有调用 `save` 方法时所有的绘图操作都在这个 `Default Stack ID` 中进行，每当我们调用一次 `save` 就会往 `Stack` 中存入一个 ID，将其后所有的操作都在这个 ID 所指向的空间进行直到我们调用 `restore` 方法还原操作，上面代码我们 `save` 了三次且没有 `restore`，stack 的结构就如上图所示，此时如果我们继续绘制东西，比如：

[java] view plaincopyprint?

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     /*
4.      * 保存并裁剪画布填充绿色
5.      */
6.     int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
7.     canvas.clipRect(mViewWidth / 2F - 300, mViewHeight / 2F - 300, mViewWidth / 2F + 300, mViewHeight / 2F + 300);
8.     canvas.drawColor(Color.YELLOW);
9.
10.    /*
```

```
11.     * 保存并裁剪画布填充绿色
12.     */
13.     int saveID2 = canvas.save(Canvas.CLIP_SAVE_FLAG);
14.     canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth / 2F + 200, mViewHeight / 2F + 200);
15.     canvas.drawColor(Color.GREEN);
16.
17.     /*
18.     * 保存画布并旋转后绘制一个蓝色的矩形
19.     */
20.     int saveID3 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
21.
22.     // 旋转画布
23.     canvas.rotate(5);
24.     mPaint.setColor(Color.BLUE);
25.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth / 2F + 100, mViewHeight / 2F + 100, mPaint);
26.
27.     mPaint.setColor(Color.CYAN);
28.     canvas.drawRect(mViewWidth / 2F, mViewHeight / 2F, mViewWidth / 2F + 200, mViewHeight / 2F + 200, mPaint);
29. }
```

我们在 `saveID3` 之后又画了一个青色的矩形，只要我不是傻子明眼都能看出这段代码是在 `saveID3` 所标识的空间中绘制的，因此其必然会受到 `saveID3` 的约束旋转：



除此之外，大家还可以很明显的看到，这个矩形除了被旋转，还被 `clip` 了~也就是说 `saveID1`、`saveID2` 也同时对其产生了影响，此时我们再次尝试在 `saveID2` 绘制完我们想要的东西后将其还原：

[java] view plaincopyprint?

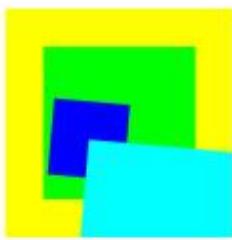
```
1. /*
2.  * 保存并裁剪画布填充绿色
3. */
4. int saveID2 = canvas.save(Canvas.CLIP_SAVE_FLAG);
5. canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth /
   2F + 200, mViewHeight / 2F + 200);
6. canvas.drawColor(Color.GREEN);
7. canvas.restore();
```

同时将青色的矩形变大一点：

[java] view plaincopyprint?

```
1. canvas.drawRect(mViewWidth / 2F, mViewHeight / 2F, mViewWidth / 2F + 400, mV
   iewHeight / 2F + 400, mPaint);
```

这时我们得到什么样的效果呢：

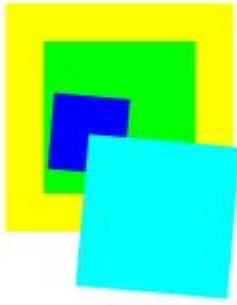


其实猜都猜得到，`saveID2` 已经不再对下面的 `saveID3` 起作用了，也就是说当我们调用 `canvas.restore()` 后标志着上一个 `save` 操作的结束或者说回滚了。同理，我们再把 `saveID1` 也 `restore`：

[java] view plaincopyprint?

```
1. /*
2.  * 保存并裁剪画布填充绿色
3. */
4. int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
5. canvas.clipRect(mViewWidth / 2F - 300, mViewHeight / 2F - 300, mViewWidth /
   2F + 300, mViewHeight / 2F + 300);
6. canvas.drawColor(Color.YELLOW);
7. canvas.restore();
```

这时 saveID3 将彻底不再受前面操作的影响:

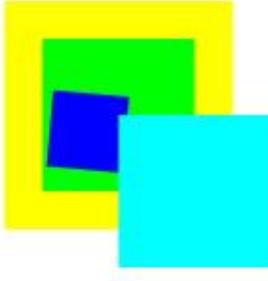


如果我们在绘制青色的矩形之前将 saveID3 也还原:

[java] view plaincopyprint?

```
1. /*
2.  * 保存画布并旋转后绘制一个蓝色的矩形
3. */
4. int saveID3 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
5. canvas.rotate(5);
6. mPaint.setColor(Color.BLUE);
7. canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth /
   2F + 100, mViewHeight / 2F + 100, mPaint);
8. canvas.restore();
```

那么这个青色的矩形将会被绘制在 Default Stack ID 上而不受其他 save 状态的影响：



上面我们提到的 `restoreToCount(int saveCount)` 方法接受一个标识值，我们可以根据这个标识值来还原特定的栈空间，效果类似就不多说了。每当我们调用 `restore` 还原 `Canvas`，对应的 `save` 栈空间就会从 `Stack` 中弹出去，`Canvas` 提供了 `getSaveCount()` 方法来为我们提供查询当前栈中有多少 `save` 的空间：

[java] view plaincopyprint?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      System.out.println(canvas.getSaveCount());
4.      /*
5.       * 保存并裁剪画布填充绿色
6.       */
7.      int saveID1 = canvas.save(Canvas.CLIP_SAVE_FLAG);
8.      System.out.println(canvas.getSaveCount());
9.      canvas.clipRect(mViewWidth / 2F - 300, mViewHeight / 2F - 300, mViewWidth
   h / 2F + 300, mViewHeight / 2F + 300);
10.     canvas.drawColor(Color.YELLOW);
11.     canvas.restore();
12.
13.     /*
14.      * 保存并裁剪画布填充绿色
15.      */
16.     int saveID2 = canvas.save(Canvas.CLIP_SAVE_FLAG);
17.     System.out.println(canvas.getSaveCount());
```

```

18.     canvas.clipRect(mViewWidth / 2F - 200, mViewHeight / 2F - 200, mViewWidth
   h / 2F + 200, mViewHeight / 2F + 200);
19.     canvas.drawColor(Color.GREEN);
20.     canvas.restore();
21.
22.     /*
23.      * 保存画布并旋转后绘制一个蓝色的矩形
24.      */
25.     int saveID3 = canvas.save(Canvas.MATRIX_SAVE_FLAG);
26.     System.out.println(canvas.getSaveCount());
27.
28.     // 旋转画布
29.     canvas.rotate(5);
30.     mPaint.setColor(Color.BLUE);
31.     canvas.drawRect(mViewWidth / 2F - 100, mViewHeight / 2F - 100, mViewWidth
   h / 2F + 100, mViewHeight / 2F + 100, mPaint);
32.     canvas.restore();
33.
34.     System.out.println(canvas.getSaveCount());
35.     mPaint.setColor(Color.CYAN);
36.     canvas.drawRect(mViewWidth / 2F, mViewHeight / 2F, mViewWidth / 2F + 400
   , mViewHeight / 2F + 400, mPaint);
37. }

```

运行后你会看到 Logcat 的如下输出：

|                           |            |   |
|---------------------------|------------|---|
| com.aigestudio.customv... | System.out | 1 |
| com.aigestudio.customv... | System.out | 2 |
| com.aigestudio.customv... | System.out | 2 |
| com.aigestudio.customv... | System.out | 2 |
| com.aigestudio.customv... | System.out | 1 |

OK，对层的了解到此为止，接下来我们主要来看看 **Canvas** 中的变换操作，说起变换，无非就几种：平移、旋转、缩放和错切，而我们的 **Canvas** 也继承了变换的精髓，同样提供了这几种相应的方法，前面的很多章节我们都用到了，像 **translate(float dx, float dy)** 方法平移画布用了无数次，这里再次强调，**translate** 方法会改变画布的原点坐标，原点坐标对变换的影响弥足轻重，前面也多次强调了！**scale(float sx, float sy)** 缩放也很好理解，但是它有一个重载方法 **scale(float sx, float sy, float px, float py)**，后两个参数用于指定缩放的中心点，前两个参数用于指定横纵向的缩放比率值在 0-1 之间为缩小：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```

1. public class LayerView extends View {
2.     private Bitmap mBitmap;// 位图对象
3.

```

```
4.     private int mViewWidth, mViewHeight;// 控件宽高
5.
6.     public LayerView(Context context, AttributeSet attrs) {
7.         super(context, attrs);
8.
9.         // 从资源中获取位图对象
10.        mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.z)
11.        ;
12.    }
13.
14.    @Override
15.    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
16.        /*
17.         * 获取控件宽高
18.         */
19.        mViewWidth = w;
20.        mViewHeight = h;
21.
22.        // 缩放位图与控件一致
23.        mBitmap = Bitmap.createScaledBitmap(mBitmap, mViewWidth, mViewHeight
24.        , true);
25.    }
26.
27.    @Override
28.    protected void onDraw(Canvas canvas) {
29.        canvas.save(Canvas.MATRIX_SAVE_FLAG);
30.        canvas.scale(1.0F, 1.0F);
31.        canvas.drawBitmap(mBitmap, 0, 0, null);
32.        canvas.restore();
33.    }
34.}
```

当缩放比率为 1 时表示不缩放：



我们改变下缩放比率：

[java] view plaincopyprint?

```
1. canvas.scale(0.8F, 0.35F);
```

此时画面效果如下：



可以看到缩放中心在左上角，我们可以使用 `scale` 的重载方法更改缩放中心：

[java] view plaincopyprint?

```
1. canvas.scale(0.8F, 0.35F, mViewWidth, 0);
```

效果如下，很好理解：



rotate(float degrees)和重载方法 rotate(float degrees, float px, float py)类似前面也用过不少就不多说了，没接触过的只有 skew(float sx, float sy)错切方法，关于错切的概念前面我们都有讲过很多，其实知道原理，方法再怎么变都不难：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      canvas.save(Canvas.MATRIX_SAVE_FLAG);
4.      canvas.skew(0.5F, 0F);
5.      canvas.drawBitmap(mBitmap, 0, 0, null);
6.      canvas.restore();
7. }
```

两个参数与 scale 类似表示横纵向的错切比率，上面代码的效果如下：



在之前的章节中我们曾讲过一个类似的用来专门操作变换的玩意 `Matrix`, 之前我也说过我们会在很多地方用到这畜生, `Canvas` 也提供了对应的方法来便于我们设置 `Matrix` 直接变换 `Canvas`:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1.  @Override
2.  protected void onDraw(Canvas canvas) {
3.      canvas.save(Canvas.MATRIX_SAVE_FLAG);
4.      Matrix matrix = new Matrix();
5.      matrix.setScale(0.8F, 0.35F);
6.      matrix.postTranslate(100, 100);
7.      canvas.setMatrix(matrix);
8.      canvas.drawBitmap(mBitmap, 0, 0, null);
9.      canvas.restore();
10. }
```

运行效果如下:



好了，关于 **Canvas** 的保存还原和变换的简单操作就介绍到这吧，剩些的一些 **draw** 方法都很好理解简单，难的我前面已经陆续穿插讲了，作为自定义控件的一部分，绘制我们用了六节的篇幅去介绍，内容多主要是 **Android** 给我们提供了很完善的接口方法以至于你在上层开发的时候压根不用去管什么源码实现，接下来的章节我们会开始进入另一个重点：控件的测量，不过在此之前我想给大家结合前面学到的一些知识来做一个关于翻页效果的小例子。

## 7. 自定义控件其实很简单 (7)

在《自定义控件其实很简单》系列的前半部分中我们用了整整六节近两万字两百多张配图讲了 **Android** 图形的绘制，虽然篇幅很巨大但仍然只是图形绘制的冰山一角，旨在领大家入门，至于修行成果就看各位的了……那么这个些列主要是通过前面学习到的一些方法来尝试完成一个翻页的效果。

对于我个人来说，我是不太建议大家在没自己去尝试前看本文的，因为你看了别人的思路就会有个惯性思维朝着别人的思路去靠，实际上如果你自己尝试去分析去做的话不见得做不出

来，甚至可能方法更简捷效率更高。分析这个效果的时候我还找妹子要了个新的笔记本翻了两三个小时，后来又在 PS 里模拟了一下然后通过绘图计算最终才有了一个简短的思路。

翻页虽然几乎每个人都试过，除非你没看过书……没碰过本子……甚至没碰过纸……一个看似简单的动作其实隐藏了巨量的信息，如果我们要把翻页的过程模拟得真实，涉及面会相当广，这里我假定我们所做的翻页效果类似于课本翻页，从右下角掀开纸张翻至左边，而我们的控件就假定为课本的右边部分而左边部分呢在我们控件左侧外看不到，那么控件的左端即可看成我们课本的装订线，这样的假定我们可以简化问题，如之前我所说，控件必定都是不完美的，如果存在完美的控件就不需要我们 Custom 了~那么这个控件实现的是一个怎样的效果呢？效果很简单，往控件中传入多张图片，我们以翻页的形式依次展示这些图片。整个翻页的原理都是想通的，虽然这个效果我模拟得很简单，但是你完全可以照我的思路定义 ViewGroup 或者 ValueAnimation 等等……

为了进一步简化问题，我们将整个翻页效果的实现分为四部分，第一部分为翻页的尝试实现，第二部分则是折线翻页的实现，第三部分我们尝试引入曲线翻页，第四部分则为一些后续效果的处理以及效率的优化，如果有必要还会增加一些章节继续完善效果。这样我们的流程就很清晰了，这一节我们首先来尝试实现翻页，如果大家能拿个本子或者书来跟着我的思路走就更好了，本来是打算拍些 Photo 作为展示的，但是我发现现实的翻页不好控制，算了，一些理论上的东西只好靠各位自行动手+脑补了。

首先，我们要进行一些约定，上面说到我们模拟的翻页效果是以控件左侧为纸张装订线使其能够实现从右下角翻开的效果，这是约定之一，其次，我们规定忽略掀起部分相对于本页的弧线使之始终与本页平行，再次规定视点与光源方向均位于纸张正上方并且光源为单光源聚光灯光锥刚好罩住纸张，这些约定不理解不要紧，在涉及到的时候我会具体说明。

我们知道 View 并非像 ViewGroup 那样是个容器可以容纳其他的控件，那么要将一些图片依次“放入”View 中并依次呈现该如何办呢？通过前面对 Canvas 的学习，我们可以尝试使用 Canvas 的“图层”功能来实现这一效果，将 Bitmap 依次至于不同的“图层”再通过 Canvas 的 clipXXX 方法裁剪呈现不同 Bitmap，理论上是可行的，那实际如何呢？我们来试试，新建一个 View 的子类 PageCurlView：

[java] view plaincopyprint?

```
1. public class PageTurnView extends View {  
2.     private List<Bitmap> mBitmaps;// 位图数据列表  
3.  
4.     public PageTurnView(Context context, AttributeSet attrs) {  
5.         super(context, attrs);  
6.     }  
7. }
```

同样，`PageCurlView` 是与数据有关的，我们对外提供一个方法来为 `PageCurlView` 设置数据：

[java] view plaincopyprint?

```
1. /**
2.  * 设置位图数据
3. *
4.  * @param mBitmaps
5.  *          位图数据列表
6. */
7. public synchronized void setBitmaps(List<Bitmap> mBitmaps) {
8.     /*
9.      * 如果数据为空则抛出异常
10.     */
11.    if (null == mBitmaps || mBitmaps.size() == 0)
12.        throw new IllegalArgumentException("no bitmap to display");
13.
14.    /*
15.     * 如果数据长度小于 2 则 GG 思密达
16.     */
17.    if (mBitmaps.size() < 2)
18.        throw new IllegalArgumentException("fuck you and fuck to use imageview");
19.
20.    this.mBitmaps = mBitmaps;
21.    invalidate();
22. }
```

这里要注意，如果图片小于两张，那就没必要去做翻页效果了，当然你也可以将其绘制出来然后在用户实行“翻页”的时候提示“已是最后一页”也可以，这里我就直接不允许图片张数小于 2 张了。

在《自定义控件其实很简单(5)》中我们自定义了一个折线视图，在该例中我们为 `PolylineView` 设置了一个初始化数据，即当用户没有设置数据时默认显示了一组随机值数据。那在这里呢我不再做初始化数据而是当绘制时如果数据为空那么我们就显示一组文本信息提示用户设置数据：

[java] view plaincopyprint?

```
1. /**
2.  * 默认显示
3. *
4.  * @param canvas
5.  *          Canvas 对象
```

```
6.  */
7. private void defaultDisplay(Canvas canvas) {
8.     // 绘制底色
9.     canvas.drawColor(Color.WHITE);
10.
11.    // 绘制标题文本
12.    mTextPaint.setTextSize(mTextSizeLarger);
13.    mTextPaint.setTextColor(Color.RED);
14.    canvas.drawText("FBI WARNING", mViewWidth / 2, mViewHeight / 4, mTextPaint);
15.
16.    // 绘制提示文本
17.    mTextPaint.setTextSize(mTextSizeNormal);
18.    mTextPaint.setTextColor(Color.BLACK);
19.    canvas.drawText("Please set data use setBitmaps method", mViewWidth / 2,
20.        mViewHeight / 3, mTextPaint);
```

如果没有设置数据，那么 PageCurlView 的默认显示效果如下：

## FBI WARNING

Please set data use setBitmaps method

如果有数据，那么我们在绘制这些位图之前要对其大小进行调整，这里我就直接将位图的大小矫正与控件一致，当然实际应用当中你可以根据比例来缩放图片使其保持宽高比，这里我就直接舍弃宽高比了：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. /**
2.  * 初始化位图数据
3.  * 缩放位图尺寸与屏幕匹配
4. */
5. private void initBitmaps() {
6.     List<Bitmap> temp = new ArrayList<Bitmap>();
7.     for (int i = 0; i < mBitmaps.size(); i++) {
8.         Bitmap bitmap = Bitmap.createScaledBitmap(mBitmaps.get(i), mViewWidth,
9.             mViewHeight, true);
10.        temp.add(bitmap);
11.    }
12.    mBitmaps = temp;
13. }
```

那么数据有了，我们尝试将其绘制出来看看：

[java] view plaincopyprint?

```
1. /**
2.  * 绘制位图
3. *
4.  * @param canvas
5.  *          Canvas 对象
6. */
7. private void drawBtimaps(Canvas canvas) {
8.     for (int i = 0; i < mBitmaps.size(); i++) {
9.         canvas.save();
10.
11.        canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);
12.
13.        canvas.restore();
14.    }
15. }
```

如上代码所示，每一次绘制位图我们都锁定还原 `Canvas` 使每一个 `Bitmap` 在绘制时都独立开来，方便我们操作：



非常壮观的建筑物~虽然我们是把 `Bitmap` 绘制出来了，但是细心的朋友会发现，绘制顺序是颠倒的，位于列表末端的 `Bitmap` 被绘制在了最顶层，很简单，我们在 `initBitmaps` 的时候掉个头不就是了么：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. private void initBitmaps() {  
2.     List<Bitmap> temp = new ArrayList<Bitmap>();  
3.     for (int i = mBitmaps.size() - 1; i >= 0; i--) {  
4.         Bitmap bitmap = Bitmap.createScaledBitmap(mBitmaps.get(i), mViewWidth,  
5.             mViewHeight, true);  
6.         temp.add(bitmap);  
7.     }  
8.     mBitmaps = temp;  
}
```

这时候运行就会显示第一张图片了：



古典欧式建筑~~

很多细心的朋友可能会有这样的疑问问什么不在 `drawBitmaps` 里面翻转顺序呢？原因有二，其一是既然是初始化数据那么我们希望在 `initBitmaps` 之后拿到的数据是直接能用的而不是在 `draw` 的时候还要执行没必要计算影响效率，其二是我们会在 `drawBitmaps` 执行一些计算，需要的一些参数包括循环的一些参数，如果我们的循环还要自行计算必定会增加逻辑的复杂度。

图片是画出来了，但是要如何去“遮住”上一张同时显示下一张图片呢？我们可以利用《自定义控件其实很简单(5)》中讲到的 `clipXXX` 裁剪方法去做，通过控制 `clipRect` 的 `right` 坐标来显示图片，好我们修改一下 `drawBitmaps` 方法加入 `clip`：

[java] view plaincopyprint?

```
1. /**
2.  * 绘制位图
3. *
4.  * @param canvas
5.  *          Canvas 对象
6. */
```

```
7. private void drawBtimaps(Canvas canvas) {  
8.     for (int i = 0; i < mBitmaps.size(); i++) {  
9.         canvas.save();  
10.        canvas.clipRect(0, 0, mClipX, mViewHeight);  
11.        canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);  
12.    }  
13.    canvas.restore();  
14. }  
15.  
16. }
```

mClipX 为裁剪区域右端的坐标值，我们在 `onSizeChanged` 中为其赋值使其等于控件宽度：

[java] view plaincopyprint?

```
1. @Override  
2. protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
3.     // 省去一些代码.....  
4.  
5.     // 初始化裁剪右端点坐标  
6.     mClipX = mViewWidth;  
7. }
```

并且重写 `View` 的 `onTouchEvent` 方法获取事件，将当前触摸点的 X 坐标赋予 `mClipX` 并重绘视图：

[java] view plaincopyprint?

```
1. @Override  
2. public boolean onTouchEvent(MotionEvent event) {  
3.     // 获取触摸点的 x 坐标  
4.     mClipX = event.getX();  
5.  
6.     invalidate();  
7.     return true;  
8. }
```

此时运行效果如下：



大家可以看到虽然我们可以通过手指的滑动来 clip 裁剪图片，但目测并没有达到我们理想的效果，clip 裁剪了所有的图片而我们其实只想裁剪第一张并使第二张显示出来.....OK，那我们再改下 drawBtimaps 方法：

[java] view plaincopyprint?

```
1. /**
2.  * 绘制位图
3. *
4. * @param canvas
5. *          Canvas 对象
6. */
7. private void drawBtimaps(Canvas canvas) {
8.     for (int i = 0; i < mBitmaps.size(); i++) {
9.         canvas.save();
10.
11.        /*
12.         * 仅裁剪位于最顶层的画布区域
13.         */
14.        if (i == mBitmaps.size() - 1) {
15.            canvas.clipRect(0, 0, mClipX, mViewHeight);
16.        }
17.        canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);
18. }
```

```
19.         canvas.restore();
20.     }
21. }
```

我们只针对位于最顶层的画布区域进行裁剪而其他的则保持不变,这样我们就可以得到一个“翻”的效果:



现在想想每次我们去滑动都要从至右滑到至左对吧,可是我们的手指是有宽度的,想精确地一次性从至右滑到至左太麻烦,我们可以在左右两端设定一个区域,当当前触摸点在该区域时让我们的图片自动滑至或者说吸附到至左或至右:

[java] view plaincopyprint?

```
1. @Override
2. public boolean onTouchEvent(MotionEvent event) {
3.     switch (event.getAction() & MotionEvent.ACTION_MASK) {
4.     default:
5.         // 获取触摸点的 x 坐标
6.         mClipX = event.getX();
7.
8.         invalidate();
9.         break;
10.    case MotionEvent.ACTION_UP:// 触点抬起时
11.        // 判断是否需要自动滑动
```

```
12.         judgeSlideAuto();
13.         break;
14.     }
15.     return true;
16. }
```

那么这个事件我们在手指抬起时触发，手指抬起后判断当前点的位置：

[java] view plaincopyprint?

```
1. /**
2.  * 判断是否需要自动滑动
3.  * 根据参数的当前值判断绘制
4. */
5. private void judgeSlideAuto() {
6.     /*
7.      * 如果裁剪的右端点坐标在控件左端五分之一的区域内，那么我们直接让其自动滑到控件
8.      * 左端
9.      */
10.    if (mClipX < mViewWidth * 1 / 5F) {
11.        while (mClipX > 0) {
12.            mClipX--;
13.            invalidate();
14.        }
15.        /*
16.         * 如果裁剪的右端点坐标在控件右端五分之一的区域内，那么我们直接让其自动滑到控件
17.         * 右端
18.         */
19.        if (mClipX > mViewWidth * 4 / 5F) {
20.            while (mClipX < mViewWidth) {
21.                mClipX++;
22.                invalidate();
23.            }
24.        }
}
```

如图所示，当我们的触摸点在距控件左端  $1/5$  的区域内时抬起手指后图片自动吸附到了左端，同样当我们的触摸点在距控件右端  $4/5-5/5$  的区域内时抬起手指后图片自动吸附到了右端：



OK，好像没什么问题是么？哈哈哈哈啊哈哈哈哈哈哈哈哈哈如果你真要这么认为你就上当了，大家可以认真地看看是不是真没问题，这里其实我给大家挖了个坑，看似没啥问题，其实涉及到一个很重要的信息，下一节我们会讲。这里要注意，因为我们会不断地触发触摸事件，也就是说 `onTouchEvent` 会不断地被调用，而在 `onTouchEvent` 中我们会不断重复地去计算 `mViewWidth * 1 / 5F` 和 `mViewWidth * 4 / 5F` 的值，这对我们控件的效率来说是相当不利的，我们考虑将其封装成成员变量并在 `onSizeChanged` 中赋予初始值：

[java] view plaincopyprint?

```
1. @Override
2. protected void onSizeChanged(int w, int h, int oldw, int oldh) {
3.     // 省去一些代码.....
4.
5.     // 计算控件左侧和右侧自动吸附的区域
6.     autoAreaLeft = mViewWidth * 1 / 5F;
7.     autoAreaRight = mViewWidth * 4 / 5F;
8. }
```

再在 `judgeSlideAuto` 中调用：

[java] view plaincopyprint?

```
1. private void judgeSlideAuto() {
```

```
2.      /*
3.       * 如果裁剪的右端点坐标在控件左端五分之一的区域内，那么我们直接让其自动滑到控件
4.       * 左端
5.       */
6.      if (mClipX < autoAreaLeft) {
7.          while (mClipX > 0) {
8.              mClipX--;
9.              invalidate();
10.         }
11.     /*
12.      * 如果裁剪的右端点坐标在控件右端五分之一的区域内，那么我们直接让其自动滑到控件
13.      * 右端
14.      */
15.      if (mClipX > autoAreaRight) {
16.          while (mClipX < mViewWidth) {
17.              mClipX++;
18.              invalidate();
19.         }
20.     }
```

对象和参数的复用一定要运用得炉火纯青，特别是在像 `onDraw`、`onTouchEvent` 这类有可能被不断重复调用的方法中，尽量避免不必要的计算（特别是浮点值的计算）和对象生成，这样对提高 `View` 运行效率有着举足轻重的意义！

到目前为止我们成功翻起了第一张图片，但是如何能够连续不断地翻起剩下的图片呢？先别急，在这之前我们先来看一个浪费资源影响效率的东西。我在 `setBitmaps` 的时候传了五张图片进来，也就是说 `mBitmaps` 数据列表的 `size` 长度为 5，而在 `drawBitmaps` 中呢我们直接通过循环将五张图片依次绘制在了 `Canvas` 中：

[java] view plain copy print?

```
1.  for (int i = 0; i < mBitmaps.size(); i++) {
2.      canvas.save();
3.
4.      // 省略一些代码.....
5.
6.      canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);
7.
8.      canvas.restore();
9. }
```

现在想想看我们是否有这样的必要呢？这里是五张图片，如果是 10 张、100 张、1000 张、10000 张……呢？这样直接一次性地全部 `draw` 绝逼要崩~而事实上我们也没有必要去一次

性绘制这么多图片，因为我们每次最多只会显示两张：上一张翻和下一张显示，也就是说我们仅需显示当前最顶层的两张图片即可~这样在有大量图片的时候可以大大提高我们绘图的效率，然后通过一些参数的判断来不断地在滑动过程中绘制往后的两张图片直至最后一张图片绘制完成，so~我们更改一下 `drawBtimaps`:

[java] view plaincopyprint?

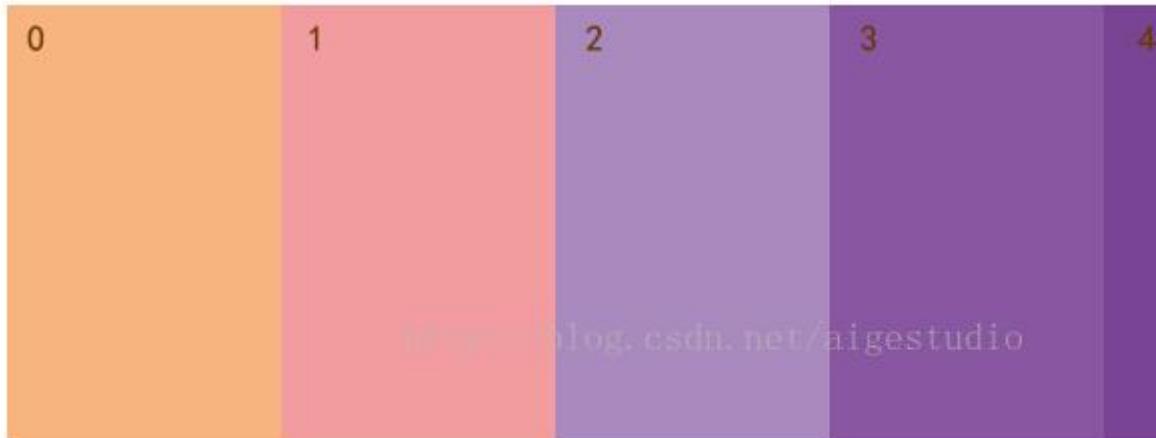
```
1. /**
2.  * 绘制位图
3. *
4.  * @param canvas
5.  *          Canvas 对象
6. */
7. private void drawBtimaps(Canvas canvas) {
8.     // 绘制位图前重置 isLastPage 为 false
9.     isLastPage = false;
10.
11.    // 限制 pageIndex 的值范围
12.    pageIndex = pageIndex < 0 ? 0 : pageIndex;
13.    pageIndex = pageIndex > mBitmaps.size() ? mBitmaps.size() : pageIndex;
14.
15.    // 计算数据起始位置
16.    int start = mBitmaps.size() - 2 - pageIndex;
17.    int end = mBitmaps.size() - pageIndex;
18.
19.    /*
20.     * 如果数据起点位置小于 0 则表示当前已经到了最后一张图片
21.     */
22.    if (start < 0) {
23.        // 此时设置 isLastPage 为 true
24.        isLastPage = true;
25.
26.        // 并显示提示信息
27.        showToast("This is fucking lastest page");
28.
29.        // 强制重置起始位置
30.        start = 0;
31.        end = 1;
32.    }
33.
34.    for (int i = start; i < end; i++) {
35.        canvas.save();
36.
37.        /*
```

```

38.         * 仅裁剪位于最顶层的画布区域
39.         * 如果到了末页则不在执行裁剪
40.         */
41.         if (!isLastPage && i == end - 1) {
42.             canvas.clipRect(0, 0, mClipX, mViewHeight);
43.         }
44.         canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);
45.
46.         canvas.restore();
47.     }
48. }

```

我们增加了一个 int 类型的成员变量 `pageIndex`, 用来作为计算读取数据列表的参考值。在这里我们约定控件左侧小于 `autoAreaLeft` 的区域为“回滚区域”, 什么意思呢? 如果我们的手指触摸点在该区域, 那么我们就认为用户的操作为“返回上一页”(当然你也可以去计算滑动起始点之差的正负来判断用户的行为, 事件不是本系列重点就不讲了)。`pageIndex` 的作用可以用简单用下图表示:



```

int pageIndex = 0;
int start = mBitmaps.size() - 2 - pageIndex;
int end = mBitmaps.size() - pageIndex;

```

| pageIndex                                                 | 0 | 1 | 2 | 3 |
|-----------------------------------------------------------|---|---|---|---|
| <code>int start = mBitmaps.size() - 2 - pageIndex;</code> | 3 | 2 | 1 | 0 |
| <code>int end = mBitmaps.size() - pageIndex;</code>       | 5 | 4 | 3 | 2 |

五种颜色代表五张图片, 左上角的序号表示其在列表中的下标位置, 当 `pageIndex` 为 0 时 `start` 为 3 而 `end` 为 5, 那么意味着列表最后的图片会被绘制在最上层, 接着绘制倒数第二张, 如果 `pageIndex` 为 1 时 `start` 为 2 而 `end` 为 4, 那么意味着列表倒数第二张的图片会被绘制在最上层, 接着绘制倒数第三张……以此类推, 那么我们该如何控制 `pageIndex` 的值呢? 由上可知, `pageIndex++` 表示显示下一页而 `pageIndex--` 则表示上一页, 那么故事就很简单了, 当我们的 `mClipX` 值为 0 时意味着图片已被裁剪完了, 那么这时候我们就可以使 `pageIndex++`, 而当用户的手指触碰回滚区域的时候则让 `pageIndex--` 显示回上一页, 既然需要判断事件, 那我们只好修改 `onTouchEvent` 啦:

[java] view plain copy print?

```
1.  @Override
2.  public boolean onTouchEvent(MotionEvent event) {
3.      // 每次触发 TouchEvent 重置 isNextPage 为 true
4.      isNextPage = true;
5.
6.      /*
7.       * 判断当前事件类型
8.       */
9.      switch (event.getAction() & MotionEvent.ACTION_MASK) {
10.         case MotionEvent.ACTION_DOWN:// 触摸屏幕时
11.             // 获取当前事件点 x 坐标
12.             mCurPointX = event.getX();
13.
14.             /*
15.              * 如果事件点位于回滚区域
16.              */
17.             if (mCurPointX < mAutoAreaLeft) {
18.                 // 那就不翻下一页了而是上一页
19.                 isNextPage = false;
20.                 pageIndex--;
21.                 mClipX = mCurPointX;
22.                 invalidate();
23.             }
24.             break;
25.         case MotionEvent.ACTION_MOVE:// 滑动时
26.             float SlideDis = mCurPointX - event.getX();
27.             if (Math.abs(SlideDis) > mMoveValid) {
28.                 // 获取触摸点的 x 坐标
29.                 mClipX = event.getX();
30.
31.                 invalidate();
32.             }
33.             break;
34.         case MotionEvent.ACTION_UP:// 触点抬起时
35.             // 判断是否需要自动滑动
36.             judgeSlideAuto();
37.
38.             /*
39.              * 如果当前页不是最后一页
40.              * 如果是需要翻下一页
41.              * 并且上一页已被 clip 掉
42.              */
43.             if (!isLastPage && isNextPage && mClipX <= 0) {
44.                 pageIndex++;

```

```
45.         mClipX = mViewWidth;
46.         invalidate();
47.     }
48.     break;
49. }
50. return true;
51. }
```

在 ACTION\_MOVE 事件中我们重新定义了事件的执行标准，如果 MOVE 的距离小于  $mMoveValid = mViewWidth * 1 / 100F$  即控件宽度的百分之一则无效，上面代码的注释和上面的分析过程一致就不多扯了，具体效果如下：



以下是这一部分 PageTurnView 的全部代码，下一节我们将尝试引入折线，将单纯的由右至左切换变成一个折页翻转的效果

[java] view plaincopyprint?

```
1. public class PageTurnView extends View {
2.     private static final float TEXT_SIZE_NORMAL = 1 / 40F, TEXT_SIZE_LARGER
   = 1 / 20F;// 标准文字尺寸和大号文字尺寸的占比
3.
4.     private TextPaint mTextPaint;// 文本画笔
5.     private Context mContext;// 上下文环境引用
6.
```

```
7.     private List<Bitmap> mBitmaps;// 位图数据列表
8.
9.     private int pageIndex;// 当前显示 mBitmaps 数据的下标
10.    private int mViewWidth, mViewHeight;// 控件宽高
11.
12.    private float mTextSizeNormal, mTextSizeLarger;// 标准文字尺寸和大号文字尺
寸
13.    private float mClipX;// 裁剪右端点坐标
14.    private float mAutoAreaLeft, mAutoAreaRight;// 控件左侧和右侧自动吸附的区
域
15.    private float mCurPointX;// 指尖触碰屏幕时点 X 的坐标值
16.    private float mMoveValid;// 移动事件的有效距离
17.
18.    private boolean isNextPage, isLastPage;// 是否该显示下一页、是否最后一页的标
识值
19.
20.    public PageTurnView(Context context, AttributeSet attrs) {
21.        super(context, attrs);
22.        mContext = context;
23.
24.        /*
25.         * 实例化文本画笔并设置参数
26.         */
27.        mTextPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG
| Paint.LINEAR_TEXT_FLAG);
28.        mTextPaint.setTextAlign(Paint.Align.CENTER);
29.    }
30.
31.    @Override
32.    public boolean onTouchEvent(MotionEvent event) {
33.        // 每次触发 TouchEvent 重置 isNextPage 为 true
34.        isNextPage = true;
35.
36.        /*
37.         * 判断当前事件类型
38.         */
39.        switch (event.getAction() & MotionEvent.ACTION_MASK) {
40.            case MotionEvent.ACTION_DOWN:// 触摸屏幕时
41.                // 获取当前事件点 x 坐标
42.                mCurPointX = event.getX();
43.
44.                /*
45.                 * 如果事件点位于回滚区域
46.                 */
```

```
47.         if (mCurPointX < mAutoAreaLeft) {
48.             // 那就不翻下一页了而是上一页
49.             isNextPage = false;
50.             pageIndex--;
51.             mClipX = mCurPointX;
52.             invalidate();
53.         }
54.         break;
55.     case MotionEvent.ACTION_MOVE:// 滑动时
56.         float SlideDis = mCurPointX - event.getX();
57.         if (Math.abs(SlideDis) > mMoveValid) {
58.             // 获取触摸点的x坐标
59.             mClipX = event.getX();
60.
61.             invalidate();
62.         }
63.         break;
64.     case MotionEvent.ACTION_UP:// 触点抬起时
65.         // 判断是否需要自动滑动
66.         judgeSlideAuto();
67.
68.         /*
69.          * 如果当前页不是最后一页
70.          * 如果是需要翻下一页
71.          * 并且上一页已被 clip 掉
72.          */
73.         if (!isLastPage && isNextPage && mClipX <= 0) {
74.             pageIndex++;
75.             mClipX = mViewWidth;
76.             invalidate();
77.         }
78.         break;
79.     }
80.     return true;
81. }
82.
83. /**
84.  * 判断是否需要自动滑动
85.  * 根据参数的当前值判断自动滑动
86.  */
87. private void judgeSlideAuto() {
88.     /*
89.      * 如果裁剪的右端点坐标在控件左端十分之一的区域内，那么我们直接让其自动滑到
控件左端
```

```
90.         */
91.         if (mClipX < mAutoAreaLeft) {
92.             while (mClipX > 0) {
93.                 mClipX--;
94.                 invalidate();
95.             }
96.         }
97.         /*
98.          * 如果裁剪的右端点坐标在控件右端十分之一的区域内，那么我们直接让其自动滑到
控件右端
99.         */
100.        if (mClipX > mAutoAreaRight) {
101.            while (mClipX < mViewWidth) {
102.                mClipX++;
103.                invalidate();
104.            }
105.        }
106.    }
107.
108.    @Override
109.    protected void onSizeChanged(int w, int h, int oldw, int oldh) {
110.        // 获取控件宽高
111.        mViewWidth = w;
112.        mViewHeight = h;
113.
114.        // 初始化位图数据
115.        initBitmaps();
116.
117.        // 计算文字尺寸
118.        mTextSizeNormal = TEXT_SIZE_NORMAL * mViewHeight;
119.        mTextSizeLarger = TEXT_SIZE_LARGER * mViewHeight;
120.
121.        // 初始化裁剪右端点坐标
122.        mClipX = mViewWidth;
123.
124.        // 计算控件左侧和右侧自动吸附的区域
125.        mAutoAreaLeft = mViewWidth * 1 / 5F;
126.        mAutoAreaRight = mViewWidth * 4 / 5F;
127.
128.        // 计算一度的有效距离
129.        mMoveValid = mViewWidth * 1 / 100F;
130.    }
131.
132.    /**

```

```
133.     * 初始化位图数据
134.     * 缩放位图尺寸与屏幕匹配
135.     */
136.     private void initBitmaps() {
137.         List<Bitmap> temp = new ArrayList<Bitmap>();
138.         for (int i = mBitmaps.size() - 1; i >= 0; i--) {
139.             Bitmap bitmap = Bitmap.createScaledBitmap(mBitmaps.get(i), mViewWidth, mViewHeight, true);
140.             temp.add(bitmap);
141.         }
142.         mBitmaps = temp;
143.     }
144.
145.     @Override
146.     protected void onDraw(Canvas canvas) {
147.         /*
148.          * 如果数据为空则显示默认提示文本
149.          */
150.         if (null == mBitmaps || mBitmaps.size() == 0) {
151.             defaultDisplay(canvas);
152.             return;
153.         }
154.
155.         // 绘制位图
156.         drawBitmaps(canvas);
157.     }
158.
159. /**
160. * 默认显示
161. *
162. * @param canvas
163. *          Canvas 对象
164. */
165. private void defaultDisplay(Canvas canvas) {
166.     // 绘制底色
167.     canvas.drawColor(Color.WHITE);
168.
169.     // 绘制标题文本
170.     mTextPaint.setTextSize(mTextSizeLarger);
171.     mTextPaint.setColor(Color.RED);
172.     canvas.drawText("FBI WARNING", mViewWidth / 2, mViewHeight / 4, mTextPaint);
173.
174.     // 绘制提示文本
```

```
175.         mTextPaint.setTextSize(mTextSizeNormal);
176.         mTextPaint.setColor(Color.BLACK);
177.         canvas.drawText("Please set data use setBitmaps method", mViewWidth
178.             / 2, mViewHeight / 3, mTextPaint);
179.     }
180. 
181.     /**
182.      * 绘制位图
183.      *
184.      * @param canvas
185.      *          Canvas 对象
186.      */
187.     private void drawBitmaps(Canvas canvas) {
188.         // 绘制位图前重置 isLastPage 为 false
189.         isLastPage = false;
190. 
191.         // 限制 pageIndex 的值范围
192.         pageIndex = pageIndex < 0 ? 0 : pageIndex;
193.         pageIndex = pageIndex > mBitmaps.size() ? mBitmaps.size() : pageIndex;
194. 
195.         // 计算数据起始位置
196.         int start = mBitmaps.size() - 2 - pageIndex;
197.         int end = mBitmaps.size() - pageIndex;
198. 
199.         /*
200.          * 如果数据起点位置小于 0 则表示当前已经到了最后一张图片
201.          */
202.         if (start < 0) {
203.             // 此时设置 isLastPage 为 true
204.             isLastPage = true;
205. 
206.             // 并显示提示信息
207.             showToast("This is fucking lastest page");
208. 
209.             // 强制重置起始位置
210.             start = 0;
211.             end = 1;
212.         }
213. 
214.         for (int i = start; i < end; i++) {
215.             canvas.save();
216.             /*
```

```
217.             * 仅裁剪位于最顶层的画布区域
218.             * 如果到了末页则不在执行裁剪
219.             */
220.             if (!isLastPage && i == end - 1) {
221.                 canvas.clipRect(0, 0, mClipX, mViewHeight);
222.             }
223.             canvas.drawBitmap(mBitmaps.get(i), 0, 0, null);
224.
225.             canvas.restore();
226.         }
227.     }
228.
229. /**
230. * 设置位图数据
231. *
232. * @param bitmaps
233. *          位图数据列表
234. */
235. public synchronized void setBitmaps(List<Bitmap> bitmaps) {
236.     /*
237.      * 如果数据为空则抛出异常
238.      */
239.     if (null == bitmaps || bitmaps.size() == 0)
240.         throw new IllegalArgumentException("no bitmap to display");
241.
242.     /*
243.      * 如果数据长度小于 2 则 GG 思密达
244.      */
245.     if (bitmaps.size() < 2)
246.         throw new IllegalArgumentException("fuck you and fuck to use im
ageview");
247.
248.     mBitmaps = bitmaps;
249.     invalidate();
250. }
251.
252. /**
253. * Toast 显示
254. *
255. * @param msg
256. *          Toast 显示文本
257. */
258. private void showToast(Object msg) {
```

```
259.         Toast.makeText(mContext, msg.toString(), Toast.LENGTH_SHORT).show()
260.     }
261. }
```

该部分源码下载：[传送门](#)

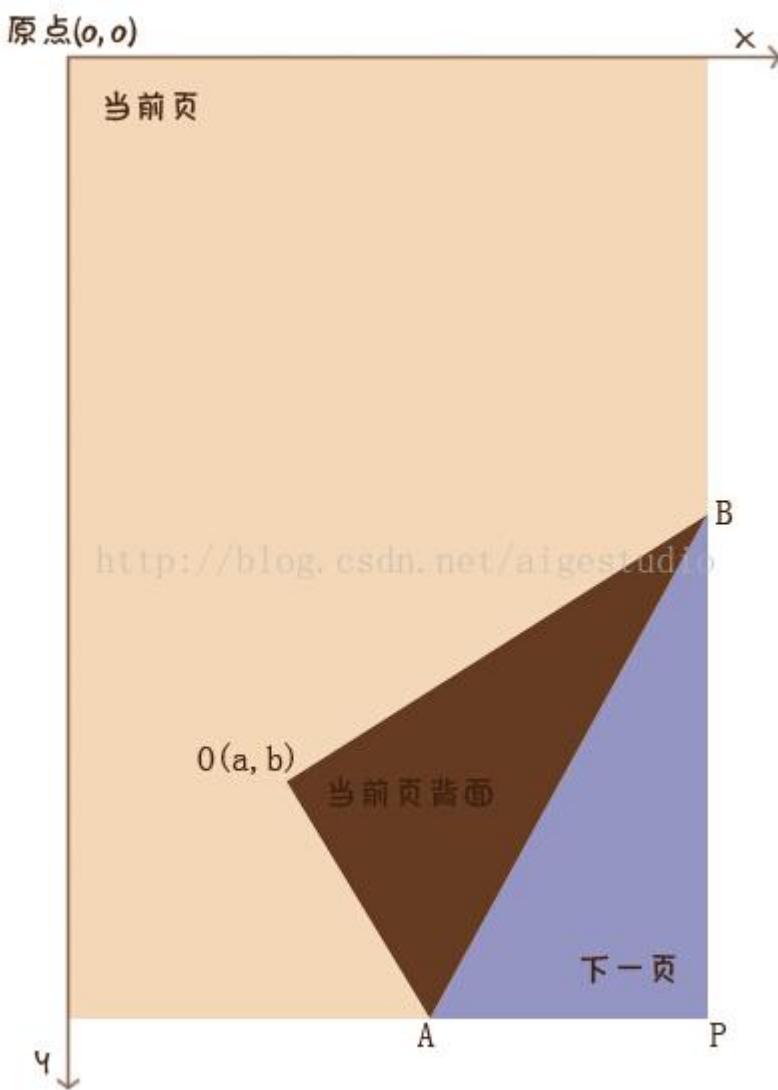
## 8. 自定义控件其实很简单(8)

**PS:** 写得太嗨忘了说明一点，下面文章中提到的“长边”（也就是代码部分中出现的 `sizeLong`）指的是折叠区域直角三角形中与控件右边相连的边，而“短边”（也就是代码部分中出现的 `sizeShort`）则指的是折叠区域直角三角形中与控件底边相连的边。两者术语并非指的是较长的边和较短的边，这点要注意。其命名来源于 My 参考图.....囧.....

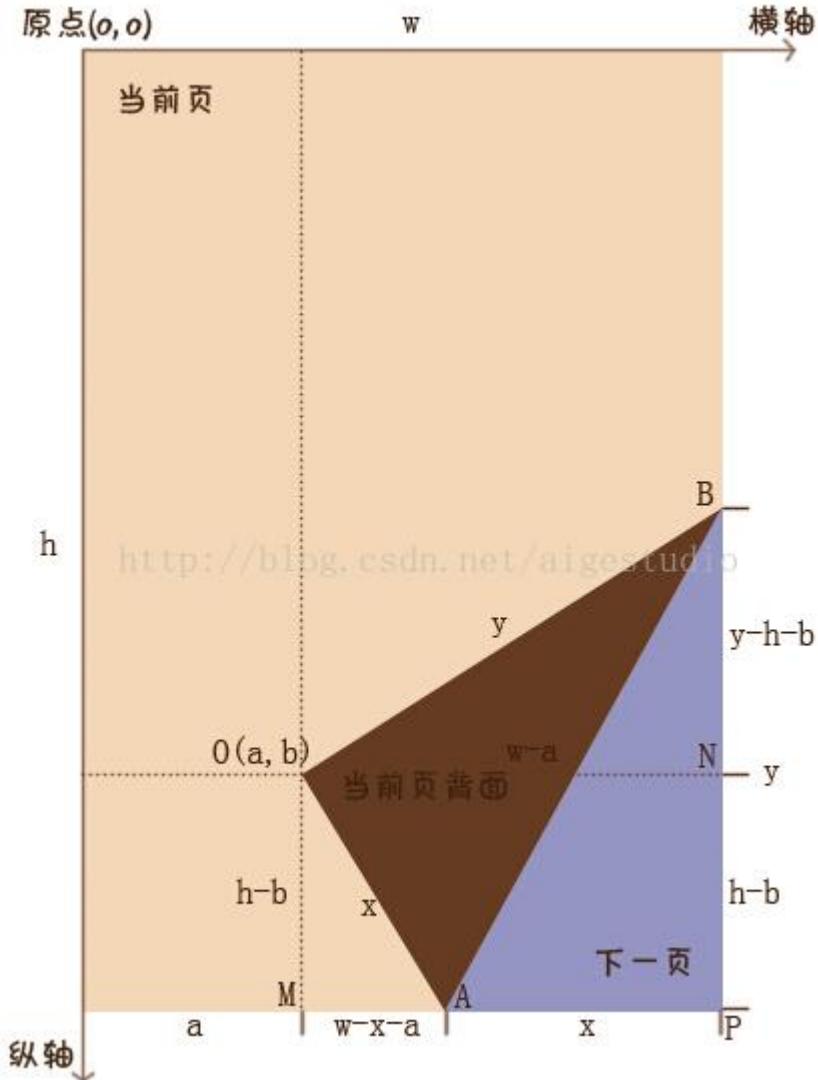
上一节中我们讲了翻页的原理实现，说白了就是 `Canvas` 中 `clip` 方法的使用，而现实生活中的翻页必然不是像我们上节 `demo` 那样左右切换的，我们总是会在看书翻页的时候掀起纸张的一角拉向书的另一侧实现翻页，翻页的过程对纸张来说是一个曲度和形状改变的过程，这一节我们先不讲曲度的实现，我们先假设翻页的过程是一个折页的过程，类似下图：



先以折页的方式对翻页过程进行一个细致的分析，然后再在下一节将折线变为曲线。折页的实现可分为两种方式，一种是纯计算，我们利用已知的条件根据各类公式定理计算出未知的值，第二种呢则是通过图形的组合巧妙地去获取图形的交并集来实现，第二种方式需要很好的空间想象力这里就先不说了，而第一种纯计算的方式呢又可以分为使用高等数学和解三角形两种方法，前者对于数学不好的童鞋来说不易理解，这里我们选择后者使用解三角形来计算，首先我们先来搞个简单的辅助图：



图很简单，一看就懂，大家可以拿个本子或者书尝试折页，不管你如何折，折叠区域  $AOB$  和下一页显示的区域  $APB$  必定是完全相等的对吧，那么我们就可以得到一个惊人的事实：角  $AOB$  恒为直角，这时我们来添加一些辅助线便于理解：



我们设折叠后的三角形  $AOB$  的短边长度为  $x$  而长边长度为  $y$ ，由图可以得出以下运算：

设： $\downarrow$

$$K = w - a, L = h - b; \downarrow$$

则有： $\downarrow$

$$x^2 = (K - x)^2 + L^2$$

解得： $\downarrow$

$$x = \frac{L^2 + K^2}{2K} \downarrow$$

我们可以使用相同的方法去解得  $y$  的值，这里我使用的是等面积法，由图可知梯形 MOBP 的面积是三角形 MOA、AOB、APB 面积之和：

因为：

$$xy + \frac{(K-x)xL}{2} = \frac{(L+y)xK}{2}$$

则有：

$$y = \frac{xL}{2x-K}$$

将  $x$  代入解得：

$$y = \frac{L^2 + K^2}{2L}$$

这样我们可以根据任意一点得出两边边长，我们来代码中实践一下看看是不是这样的呢？为了便于理解，这里我重新使用了一个新的 FoldView：

```
[java] view plaincopyprint?
```

```
1. public class FoldView extends View {  
2.     public FoldView(Context context, AttributeSet attrs) {  
3.         super(context, attrs);  
4.     }  
5. }
```

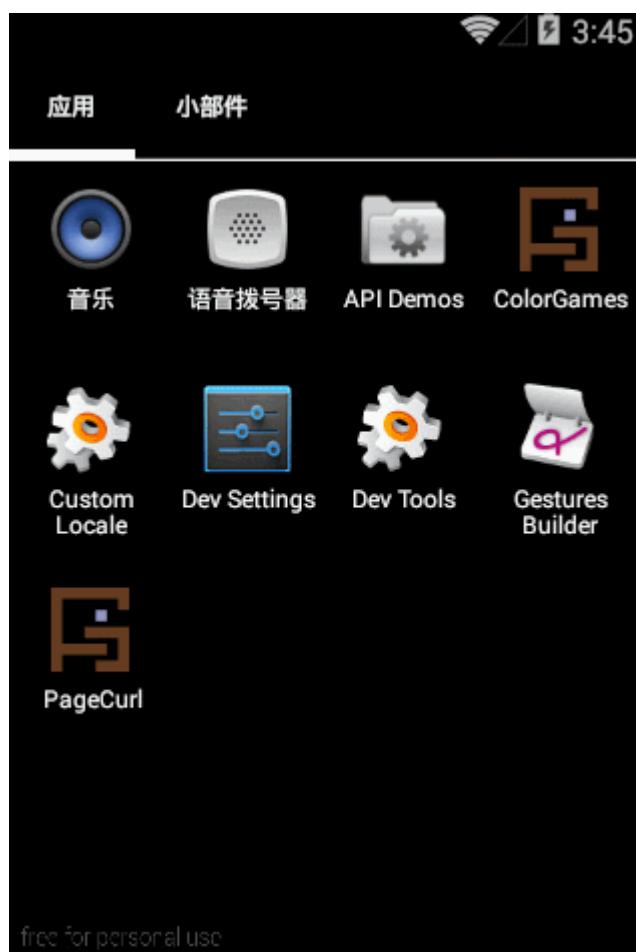
那么尝试根据我们以上分析的原理来绘制这么一个折页的效果，获取事件点、获取控件宽高就不说了，我们重点来看看 onDraw 中的计算：

```
[java] view plaincopyprint?
```

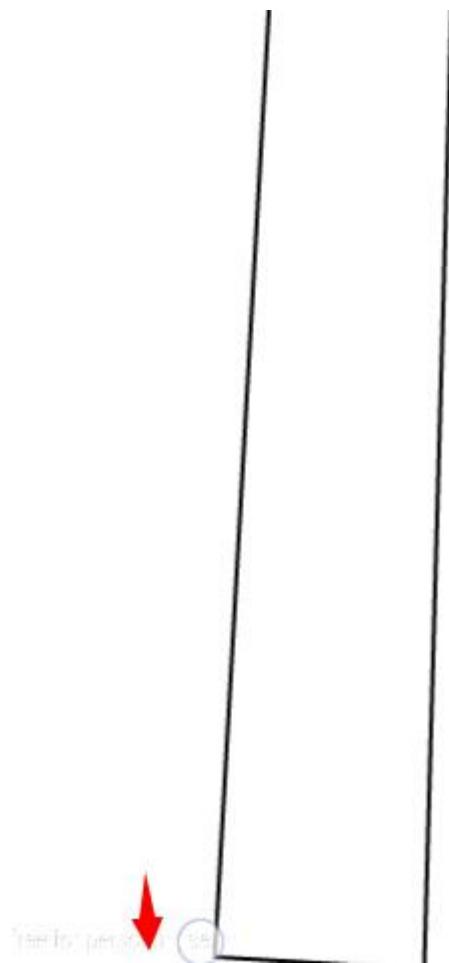
```
1. @Override  
2. protected void onDraw(Canvas canvas) {  
3.     // 重绘时重置路径  
4.     mPath.reset();  
5.  
6.     // 绘制底色  
7.     canvas.drawColor(Color.WHITE);  
8.  
9.     /*  
10.      * 如果坐标点在右下角则不执行绘制  
11.      */  
12.     if (pointX == 0 && pointY == 0) {  
13.         return;
```

```
14.    }
15.
16.    /*
17.     * 额，这个该怎么注释好呢.....根据图来
18.     */
19.    float mK = mViewWidth - pointX;
20.    float mL = mViewHeight - pointY;
21.
22.    // 需要重复使用的参数存值避免重复计算
23.    float temp = (float) (Math.pow(mL, 2) + Math.pow(mK, 2));
24.
25.    /*
26.     * 计算短边长边长度
27.     */
28.    float sizeShort = temp / (2F * mK);
29.    float sizeLong = temp / (2F * mL);
30.
31.    /*
32.     * 生成路径
33.     */
34.    mPath.moveTo(pointX, pointY);
35.    mPath.lineTo(mViewWidth, mViewHeight - sizeLong);
36.    mPath.lineTo(mViewWidth - sizeShort, mViewHeight);
37.    mPath.close();
38.
39.    // 绘制路径
40.    canvas.drawPath(mPath, mPaint);
41. }
```

每次绘制的时候我们需要重置 Path 不然上一次的 Path 就会跟这一次叠加在一起，效果如下：



效果是大致出来了，但是我们发现有一处不对的地方，当我们非常靠左或非常靠下地折叠时：

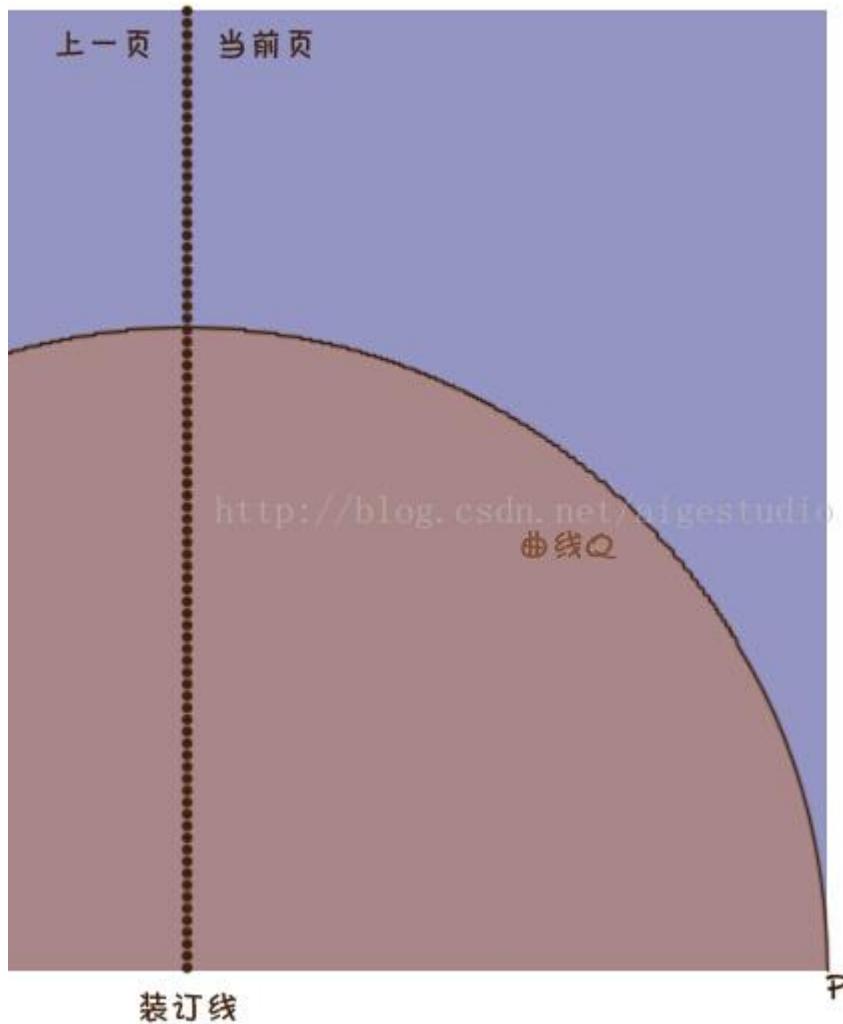


如果再往下折



如果再往左折

此时我们的 Path 就会消失掉，其实这跟我们现实中的折页是一样的，折页的过程是有限制的，如下图：



右下角点 P 因为受装订线的制约，其半径最大只能为纸张的宽度，如果我们始终以该宽度为半径折页，那么点 P 的轨迹就可以形成曲线 Q，图中半透明红色区域为一个半圆形，也就是说，我们的点 P 只能在该范围内才应当有效对吧，那么该如何做限制呢？很简单，我们只需在计算长短边长之前判断触摸点是否在该区域即可：

[java] view plain copy print?

```
1. /**
2.  * 计算短边的有效区域
3. */
4. private void computeShortSizeRegion() {
5.     // 短边圆形路径对象
6.     Path pathShortSize = new Path();
7.
8.     // 用来装载 Path 边界值的 RectF 对象
```

```
9.     RectF rectShortSize = new RectF();
10.
11.    // 添加圆形到 Path
12.    pathShortSize.addCircle(0, mViewHeight, mViewWidth, Path.Direction.CCW);
13.
14.    // 计算边界
15.    pathShortSize.computeBounds(rectShortSize, true);
16.
17.    // 将 Path 转化为 Region
18.    mRegionShortSize.setPath(pathShortSize, new Region((int) rectShortSize.left,
19.                                                       (int) rectShortSize.top, (int) rectShortSize.right, (int) rectShortSize.bottom));
20. }
```

同样计算有效区域这个过程是在 `onSizeChanged` 中进行，我们说过尽量不要在一些重复调用的方法内执行没必要的计算，在 `onDraw` 里我们只需在绘制钱判断下当前触摸点是否在该区域内，如果不在，那么我们通过坐标 x 轴重新计算坐标 y 轴：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 判断触摸点是否在短边的有效区域内
3. */
4. if (!mRegionShortSize.contains((int) mPointX, (int) mPointY)) {
5.     // 如果不在则通过 x 坐标强行重算 y 坐标
6.     mPointY = (float) (Math.sqrt((Math.pow(mViewWidth, 2) - Math.pow(mPointX,
7.         2))) - mViewHeight);
8.     // 精度附加值避免精度损失
9.     mPointY = Math.abs(mPointY) + mValueAdded;
10. }
```

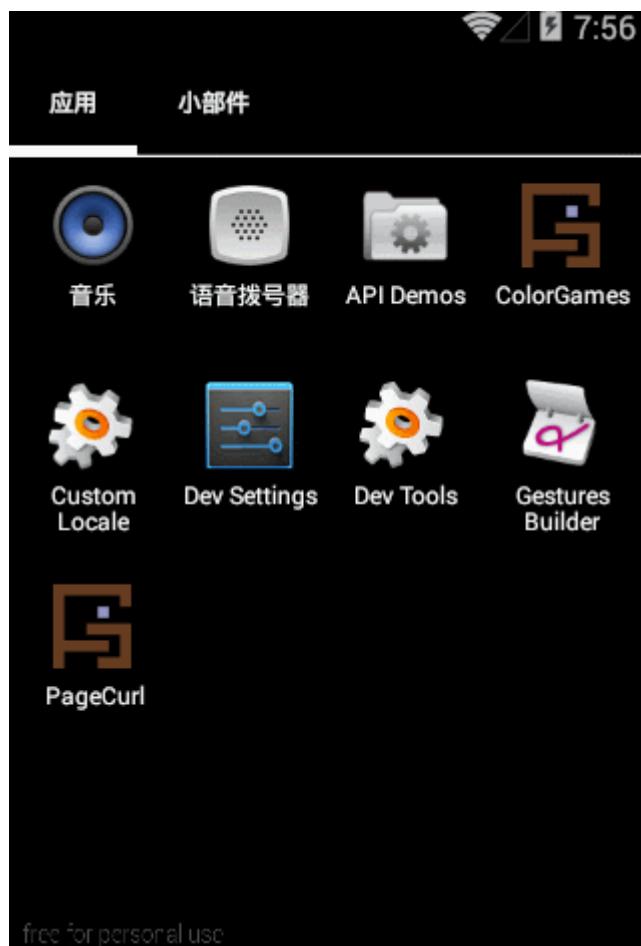
那么如何来计算 y 坐标呢？很简单，我们只需根据圆的方程求解即可，因为 P 的轨迹是个圆~在得到新的 y 坐标 `mPointY` 后我们还应该为其加上一点点的精度值 `mValueAdded` 来挽回因浮点计算而损失的精度。而对于过分往下折出现的问题我们使用限制下折最大值的方法来避免：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

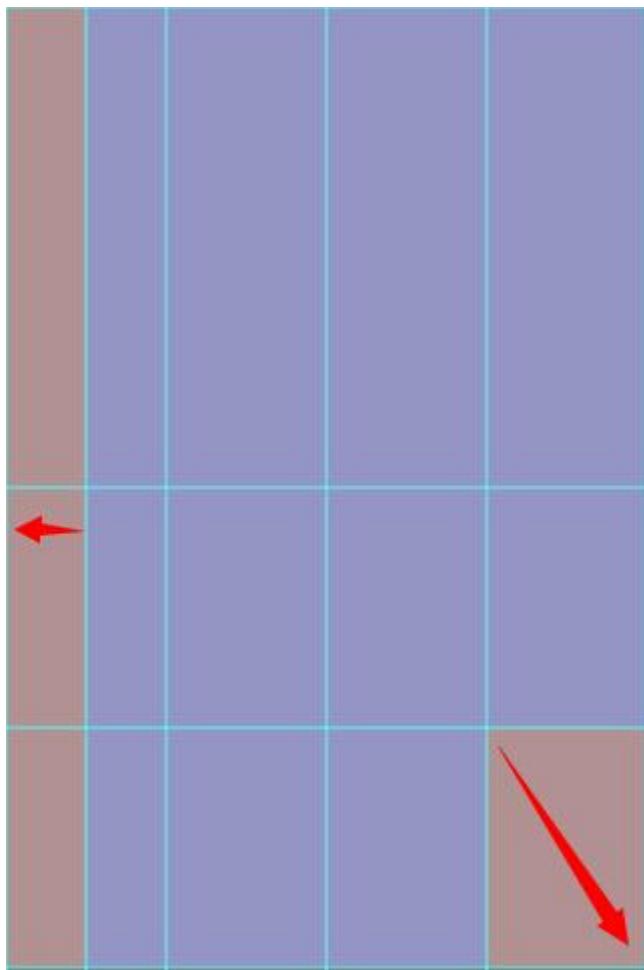
```
1. /*
2.  * 缓冲区域判断
3. */
4. float area = mViewHeight - mBuffArea;
```

```
5. if (mPointY >= area) {  
6.     mPointY = area;  
7. }
```

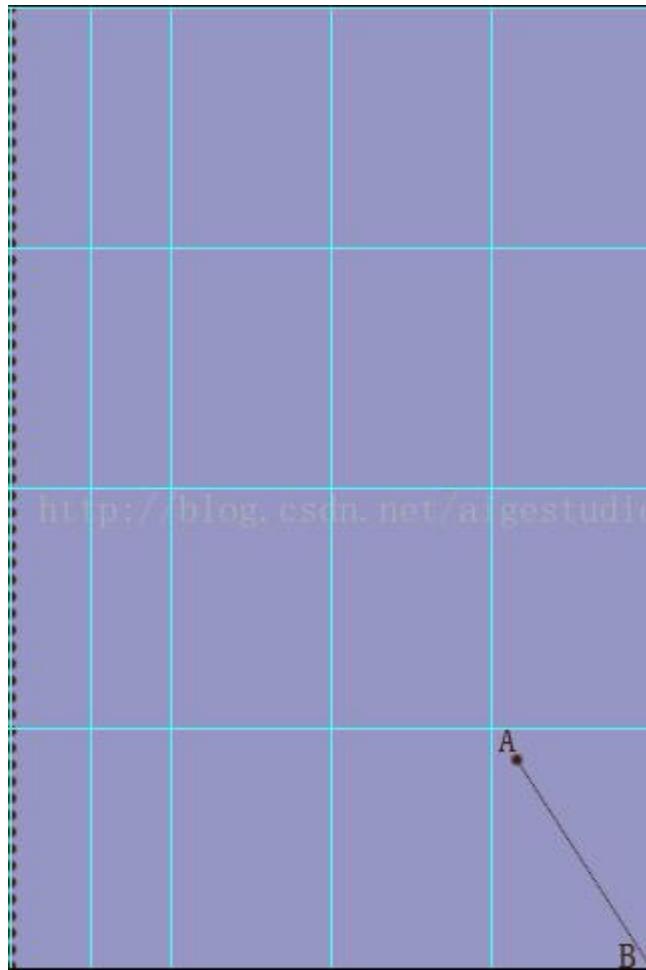
在控件下方接近底部的地方我们设定一个缓冲区域，触摸点永远不能到达该区域，因为没有必要也没有意义，再往下就要划出控件了，别浪费多余的计算，运行效果大致如下：



大致的效果出来了，我们还需要做一些补充工作，当触摸点在右下角某个区域时如果我们抬起手指，那么就让“纸张”自动滑下去，同理当触摸点在左边某个区域时我们让“纸张”自动翻过去，这里我们约定这两个区域分别是控件右下角宽高四分之一的区域和控件左侧八分之一的区域（当然你可以约定你自己的控件行为，这里我就哪简单往哪走了~）：



那么在上一节中我们也有类似的效果，这里我们依葫芦画瓢，当手指抬起时判断当前事件点是否位于右下角自滑区域内，如果在那么以当前事件点为坐标点 A 右下角为坐标点 B 根据两点式我们可以获得一条直线方程：



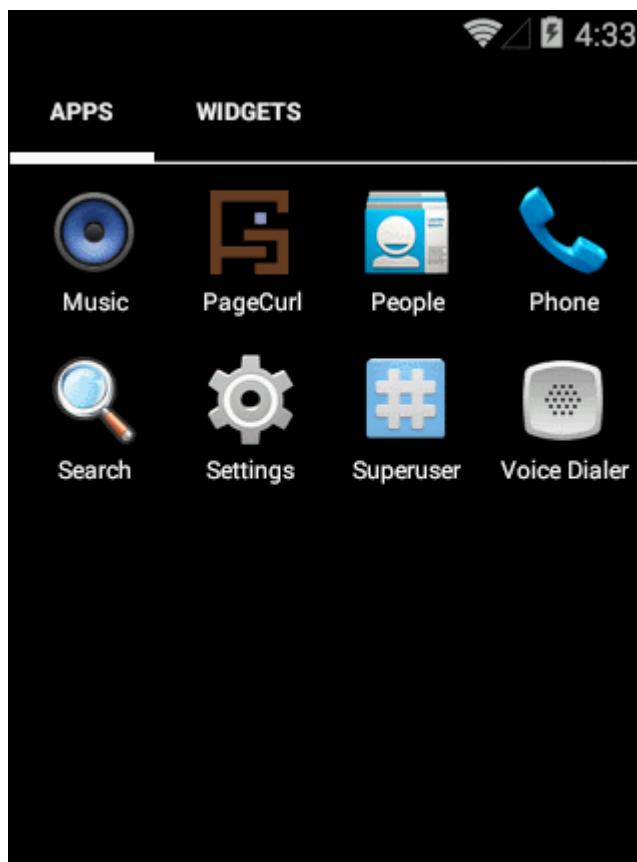
此后根据不断自加递增的 x 坐标不断计算对应的 y 坐标直至点滑至右下角为止,既然涉及到事件, So 我们在 `onTouchEvent` 处理:

[java] view plain copy print?

```
1. case MotionEvent.ACTION_UP:// 手指抬起时候
2.     /*
3.      * 获取当前事件点
4.      */
5.     float x = event.getX();
6.     float y = event.getY();
7.
8.     /*
9.      * 如果当前事件点位于右下自滑区域
10.     */
11.    if (x > mAutoAreaRight && y > mAutoAreaBottom) {
12.        // 获取当前点为直线方程坐标之一
13.        float startX = x, startY = y;
14.
15.        /*
16.         * 当 x 坐标小于控件宽度时
```

```
17.         */
18.     while (x < mViewWidth) {
19.         // 不断让x自加
20.         x++;
21.
22.         // 重置当前点的值
23.         mPointX = x;
24.         mPointY = startY + ((x - startX) * (mViewHeight - startY)) / (mViewWidth - startX);
25.
26.         // 重绘视图
27.         invalidate();
28.     }
29. }
30. break;
```

OK，我们来看看效果：



大家看到当手指弹起时如果触摸点在右下角的自滑区域内的话就会自动“滑动”到右下角去，可是大家细心的话会发现效果好像不太对啊！怎么一下子就到右下角了？说好的“滑动”呢？好像毫无滑动效果啊！！！为什么会这样？其实如果你细心就会发现上一节我们在讲图片左右两侧自滑的时候也是一样的效果！根本就没有什么滑动！为什么？难道在我们的 **while**

循环中没有执行 `invalidate` 吗？大家可以尝试在 `View` 中重写 `invalidate()` 方法 `Log` 一些信息看看 `invalidate()` 是否没有执行。这里鉴于篇幅我就直接简单地说一下了，具体的我们会在《自定义控件其实很简单》系列文章讲到 `View` 绘制流程的时候详细阐述。这里我先可以告诉大家的是 `invalidate()` 方法即便你调用了也不会马上执行，`invalidate()` 的作用更准确地说是将我们的 `View` 标记为无效，当 `View` 被标记为无效后 `Android` 就会尝试去调用 `onDraw()` 对其重绘，如果大家曾翻阅过 API 文档就会看到在 `invalidate()` 方法中 Google 给出了这么一句话：

```
public void invalidate ()
```

Invalidate the whole view. If the view is visible, `onDraw(android.graphics.Canvas)` will be called at some point.

我们知道 UI 的刷新需要在 `UI Thread` 也就是主线程中进行，这里会涉及到一个叫做 `message` 和 `message queue` 的东西，`message` 你可以见文知意地称其为消息而 `message queue` 则为消息队列，我们将一个 `message` 压入 `message queue` 后 `UI Thread` 会处理它，而我们刷新 UI 也需要有 `message` 作为载体去告诉 `UI Thread` 该需要更新 UI 了哦，而当我们在 `UI Thread` 中去做一个 `loop` 不断地往 `message queue` 中压入消息时，我们的 `UI Thread` 是不会去处理这些 `message` 的，直到 `loop` 结束为止，这就是为什么我们在 `while` 中不断调用 `invalidate()` 的时候你只会看到最后的结果而不会得到中间过程的变化。这里我只阐述了一个很浅显能懂的原因，更深入的原因涉及到 `View` 中各种标识位的运算如上所说篇幅过长就不多说了。那么知道了原因该如何去处理呢？`message` 和 `message queue` 如果大家对 `Handler` 有一定的了解一定不陌生，没错，这里我们也将使用 `Handler` 来实现我们的滑动，首先，在我们的 `View` 中创建一个内部类，该内部类是 `Handler` 的一个子类，我们将使用它来更新 `View` 实现滑动效果：

[java] view plain copy print?

```
1. /**
2.  * 处理滑动的 Handler
3. */
4. @SuppressLint("HandlerLeak")
5. private class SlideHandler extends Handler {
6.     @Override
7.     public void handleMessage(Message msg) {
8.         // 循环调用滑动计算
9.         FoldView.this.slide();
10.
11.        // 重绘视图
12.        FoldView.this.invalidate();
13.    }
14.}
```

```
15.     /**
16.      * 延迟向 Handler 发送消息实现时间间隔
17.      *
18.      * @param delayMillis
19.      *          间隔时间
20.      */
21.     public void sleep(long delayMillis) {
22.         this.removeMessages(0);
23.         sendMessageDelayed(obtainMessage(0), delayMillis);
24.     }
25. }
```

我们额外提供一个 `slide()` 方法来对参数值进行更新：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 计算滑动参数变化
3. */
4. private void slide() {
5.     /*
6.      * 如果 x 坐标恒小于控件宽度
7.      */
8.     if (isSlide && mPointX < mViewWidth) {
9.         // 则让 x 坐标自加
10.        mPointX++;
11.
12.        // 并根据 x 坐标的值重新计算 y 坐标的值
13.        mPointY = mStart_BR_Y + ((mPointX - mStart_BR_X) * (mViewHeight - mStart_BR_Y)) / (mViewWidth - mStart_BR_X);
14.
15.        // 让 SlideHandler 处理重绘
16.        mSlideHandler.sleep(1);
17.    }
18. }
```

而在 `onTouchEvent` 中我们则不在处理参数的计算和重绘，仅需简单调用 `slide()` 方法即可：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. case MotionEvent.ACTION_UP:// 手指抬起时候
2.     /*
3.      * 获取当前事件点
4.      */
```

```
5.     float x = event.getX();
6.     float y = event.getY();
7.
8.     /*
9.      * 如果当前事件点位于右下自滑区域
10.     */
11.    if (x > mAutoAreaRight && y > mAutoAreaBottom) {
12.        // 获取并设置直线方程的起点
13.        mStart_BR_X = x;
14.        mStart_BR_Y = y;
15.
16.        // OK 要开始滑动了哦~
17.        isSlide = true;
18.
19.        // 滑动
20.        slide();
21.    }
22.    break;
```

注: `mStart_BR_X` 和 `mStart_BR_Y` 为直线方程的一点, 这里我单独使用两个引用存值便于大家理解, 如果各位数学基础好完全可以将其并入到 `slide()` 方法中一并计算并省去这两个引用的声明。

这里我们定义了一个 `boolean` 类型的 `isSlide` 标识值, 目的是方便控制动画, 我们对外提供一个 `slideStop` 方法便于其他组件对动画的控制 (当然你可以提供更多方法来控制 `slide`, 这里就不多说了), 例如当 `Activity` 的 `onDestroy` 被调用时让动画停止:

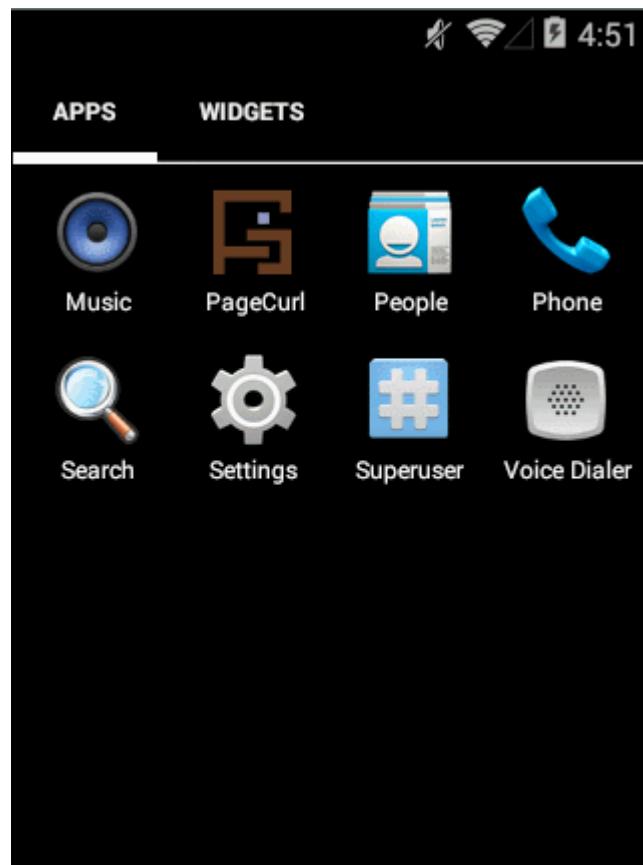
[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)

```
1. /**
2.  * 为 isSlide 提供对外的停止方法便于必要时释放滑动动画
3. */
4. public void slideStop() {
5.     isSlide = false;
6. }
```

在这一过程中, 我们在事件手指抬起时判断点的所在, 如果在滑动区域我们则触发 `slide()` 方法的执行, 在 `slide()` 方法中我们重新计算坐标值并调用 `SlideHandler` 的 `sleep(long delayMillis)` 方法, `sleep(long delayMillis)` 的处理逻辑也很简单, 根据 `delayMillis` 延时向 `SlideHandler` 发送 `obtainMessage`, 在 `SlideHandler` 的 `handleMessage` 方法中再次调用 `slide()` 方法重新计算参数值并刷新界面。看到这里你可能会问为什么 `slide()` 和 `sleep()` 没有死循环而? `Don't worry!` 我们来细致分析一下我们到底干了什么, 首先我们创建了一个 `Handler` 的子类 `SlideHandler` 与当前 `Thread` 绑定在一起由此我们才可以直接给 `Thread` 发送并处理 `message`。因为 `Handler` 对 `message` 的处理都是异步的, 所以在我们自定的

SlideHandler 中 sleep()方法也是个异步方法，所以 slide()和 sleep()之间的相互调用才没有构成死循环。

好了，分析归分析，我们还是要看实际效果的对吧，执行一下看看：



是不是有“滑动”的效果了？之前我们在处理 MOVE 的时候在 onDraw 中定义了一个底部的缓冲区：

[java] view plaincopyprint?

```
1. float area = mViewHeight - mBuffArea;
2. if (mPointY >= area) {
3.     mPointY = area;
4. }
```

而在自滑的时候我们是不需要去判断它的，So~我们改改：

[java] view plaincopyprint?

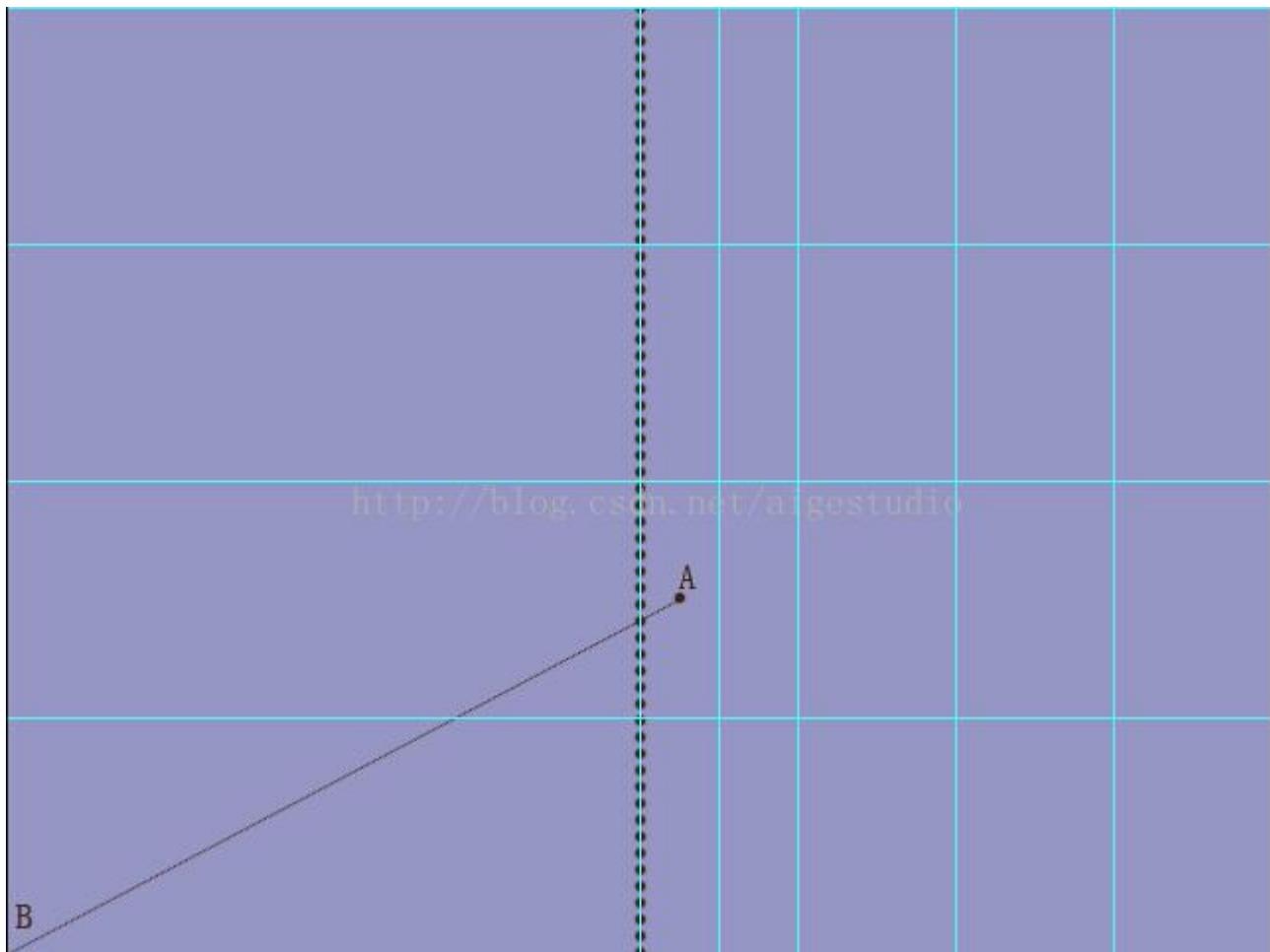
```
1. /*
2.  * 缓冲区域判断
3. */
4. float area = mViewHeight - mBuffArea;
5. if (!isSlide && mPointY >= area) {
```

```
6.     mPointY = area;  
7. }
```

只有当没有产生滑动动画时才去判断缓冲区~

至此，从《自定义控件其实很简单》系列开始我们已经学会三种对 View 进行刷新的方式：第一种是在刚开始讲《自定义控件其实很简单(1)》让 View 作为 Runnable 的实现类，在 run 方法中更新，另一种是我们后来用的比较多的直接在 onDraw 方法中 invalidate()，最后一种呢则是上面我们讲的 Handler 来处理绘制逻辑，这三种方法虽说本质一样但是实现方式各不相同且应用场景也不尽相同~第一种更倾向于多种状态进行同时重绘，第二种局限性很大虽说常见但能实现的功能很弱，第三种可以应用到绝大多数的重绘情景且不受不同状态的影响自由度更大。

好了，我们继续修改下代码让左侧也实现自滑的功能：



如图所示，当我们的触摸点 x 坐标落于控件左侧 1/8 处弹起手指时，我们让该点与“上一页”的左下角相连构成一条直线，让点沿着该直线不断下滑直至“上一页”的左下角，鉴于我们要分开对左下和右下滑动进行处理，这里我定义一个枚举内部类：

```
[java] view plain copy print?
```

```
1.  /**
2.   * 枚举类定义滑动方向
3.   */
4.  private enum Slide {
5.      LEFT_BOTTOM, RIGHT_BOTTOM
6.  }
```

对应地我们就需要一个成员变量来存值咯：

```
[java] view plaincopyprint?
```

```
1.  private Slide mSlide;// 定义当前滑动是往左下滑还是右下滑
```

重新整理 onTouchEvent 处理逻辑：

```
[java] view plaincopyprint?
```

```
1.  case MotionEvent.ACTION_UP:// 手指抬起时候
2.      /*
3.       * 获取当前事件点
4.       */
5.      float x = event.getX();
6.      float y = event.getY();
7.
8.      /*
9.       * 如果当前事件点位于右下自滑区域
10.      */
11.     if (x > mAutoAreaRight && y > mAutoAreaBottom) {
12.         // 当前为往右下滑
13.         mSlide = Slide.RIGHT_BOTTOM;
14.
15.         // 摩擦吧骚年！
16.         justSlide(x, y);
17.     }
18.
19.     /*
20.      * 如果当前事件点位于左侧自滑区域
21.      */
22.     if (x < mAutoAreaLeft) {
23.         // 当前为往左下滑
24.         mSlide = Slide.LEFT_BOTTOM;
25.
26.         // 摩擦吧骚年！
27.         justSlide(x, y);
```

```
28.    }
29.    break;
```

我们将一些相同的方法封装在 justSlide 中：

```
[java] view plaincopyprint?
```

```
1. /**
2. * 在这光滑的地板上~
3. *
4. * @param x
5. *          当前触摸点 x
6. * @param y
7. *          当前触摸点 y
8. */
9. private void justSlide(float x, float y) {
10.    // 获取并设置直线方程的起点
11.    mStart_X = x;
12.    mStart_Y = y;
13.
14.    // OK 要开始滑动了哦~
15.    isSlide = true;
16.
17.    // 滑动
18.    slide();
19. }
```

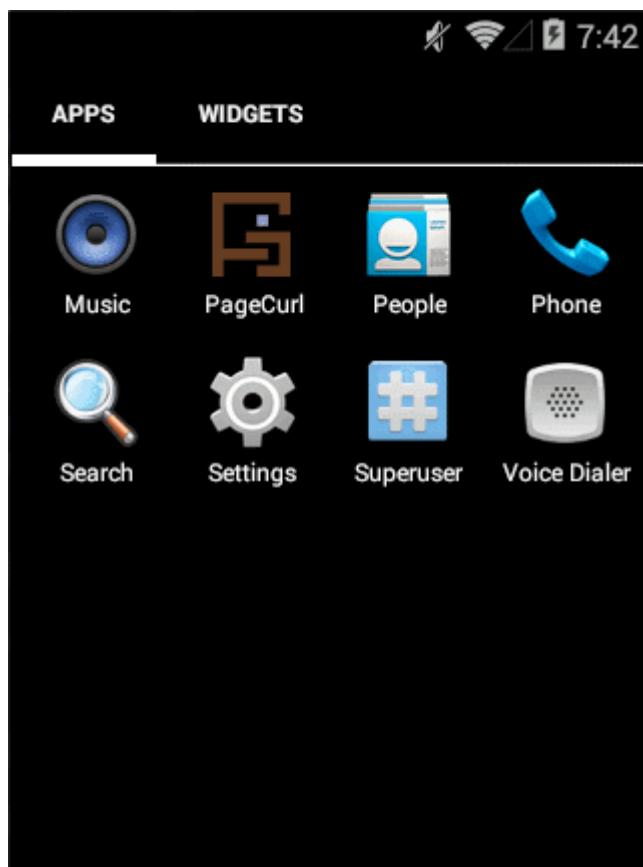
slide()中的处理则会根据滑动方向来计算参数值：

```
[java] view plaincopyprint?
```

```
1. /**
2. * 计算滑动参数变化
3. */
4. private void slide() {
5.    /*
6.     * 如果滑动标识值为 false 则返回
7.     */
8.    if (!isSlide) {
9.        return;
10.    }
11.
12.    /*
13.     * 如果当前滑动标识为向右下滑动 x 坐标恒小于控件宽度
```

```
14.     */
15.     if (mSlide == Slide.RIGHT_BOTTOM && mPointX < mViewWidth) {
16.         // 则让 x 坐标自加
17.         mPointX += 10;
18.
19.         // 并根据 x 坐标的值重新计算 y 坐标的值
20.         mPointY = mStart_Y + ((mPointX - mStart_X) * (mViewHeight - mStart_Y
21. )) / (mViewWidth - mStart_X);
22.
23.         // 让 SlideHandler 处理重绘
24.         mSlideHandler.sleep(25);
25.     }
26.
27.     /*
28.      * 如果当前滑动标识为向左下滑动 x 坐标恒大于控件宽度的负值
29.      */
30.     if (mSlide == Slide.LEFT_BOTTOM && mPointX > -mViewWidth) {
31.         // 则让 x 坐标自减
32.         mPointX -= 20;
33.
34.         // 并根据 x 坐标的值重新计算 y 坐标的值
35.         mPointY = mStart_Y + ((mPointX - mStart_X) * (mViewHeight - mStart_Y
36. )) / (-mViewWidth - mStart_X);
37.
38.         // 让 SlideHandler 处理重绘
39.         mSlideHandler.sleep(25);
40.     }
41. }
```

看看效果：



好像一切都很完美，但是我们发现当 Path 绘制到快结束时卡了两下，同时我们的 LogCat 也出现了如下警告：

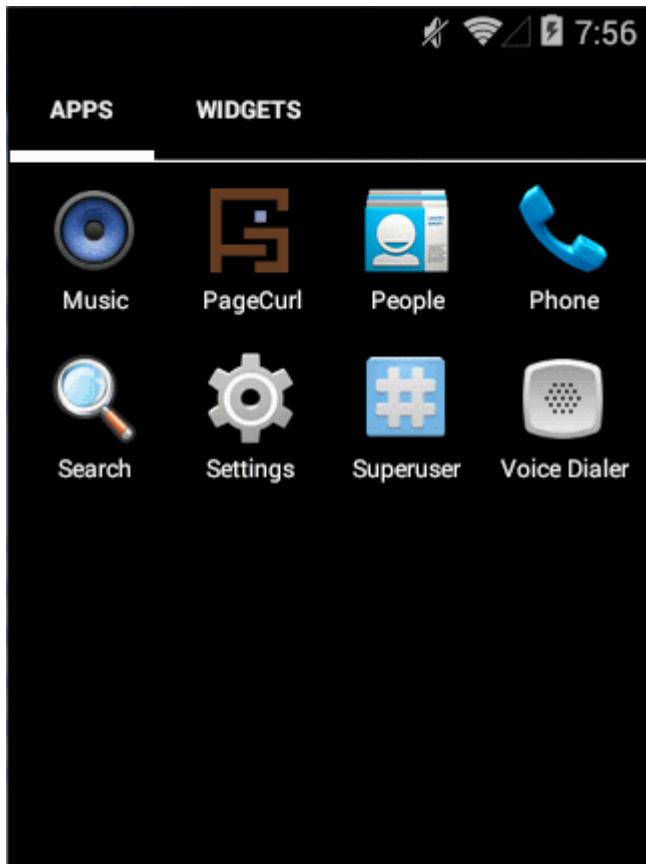
```
Shape too large to be rendered into a texture (643x136217, max=16384x16384)
```

说是我们的 Path 太大了已经超出了 texture 的渲染范围，什么是 texture 这要涉及到 GL 等 Android 底层对图形绘制的过程，我们不需要理解，但是要知道的是，从 API 11 开始 Android 开始支持 HW 硬件加速绘制视图，硬件加速对 texture 是有限制的，这个限制值因机而异，如上面我们在警告信息的最后看到的 `max = 16384 x 16384`，如何解决呢？最简单的方法当然是直接关闭硬件加速咯，关于关闭硬件加速的两种方法在《自定义控件其实很简单》中我们已经说过：

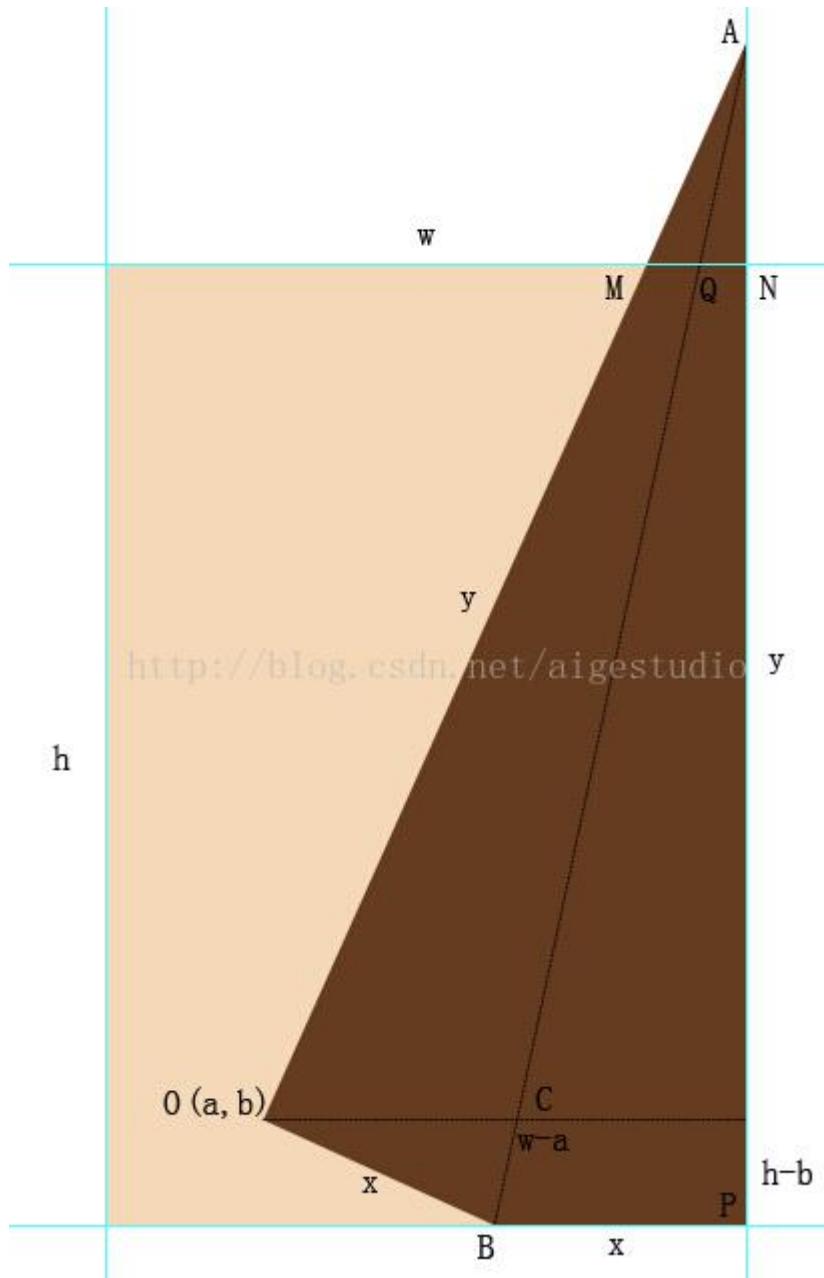
```
[java] view plaincopyprint?
```

```
1. setLayerType(LAYER_TYPE_SOFTWARE, null);
```

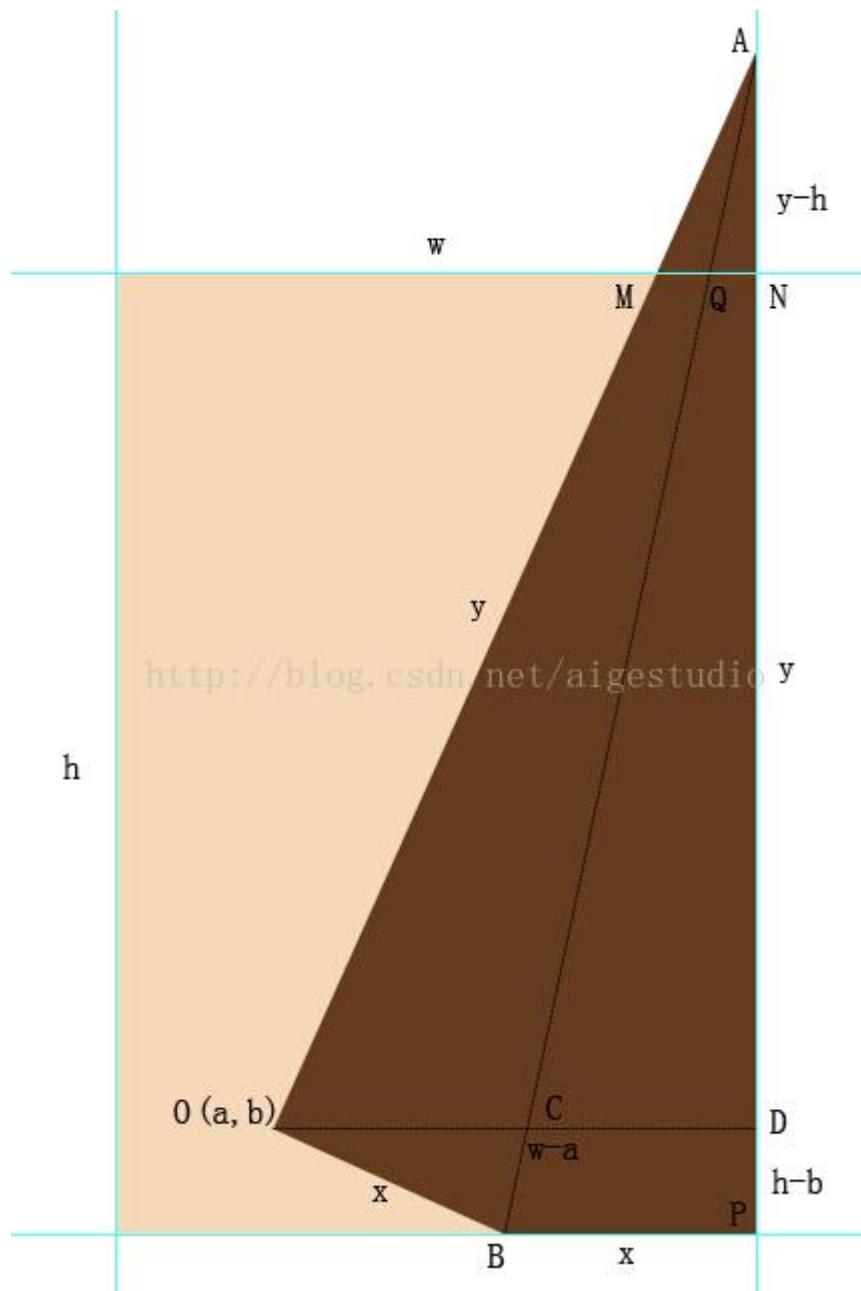
这样我们再次运行：



唉~Good，很顺畅也没出现警告了对吧，但是 Path 的大小依然是没有改变的，依然是灰常灰常大，超出控件上方的部分依旧被绘制而且很大很大，其实这部分绘制是完全没必要的，而且为此我们还关闭了 HW 更可怕的是还将 APP 的最低支持版本升到了 11.....我们可不可以通过其他方式来避免呢？答案是肯定的，方法很多，但是最简明扼要的还是 Calculation~~~~~我们尝试去判断折叠后长边的的长度，如果长边的长度大于控件的高度则我们折叠的分部就不是一个三角形而是一个四边形了：



如上图中的四边形  $OBQM$ , 那么如何来生成这个四边形的区域呢? 我们曾约定在手指  $MOVE$  的过程中控件下方有一个“缓冲区域”, 也就是说我们的触摸点  $Y$  坐标永远不可能在  $MOVE$  的过程中与控件底部重合, 这个约定给我们在计算四边形区域的时候带来一个好处: 如上图所示  $OA$  边总会与  $PN$  的延长线有交点, 那样计算就非常简单了, 我们过点  $O$  作一条垂直于  $PN$  边的垂线与  $PN$  边相交于点  $D$ :



那么我么就会有：

$$\therefore \frac{AN}{AD} = \frac{MN}{OD}$$

$$\therefore MN = \frac{AN}{AD} \times OD$$

$$\therefore \frac{AN}{AP} = \frac{QN}{BP}$$

$$\therefore QN = \frac{AN}{AD} \times BP$$

由此很容易得出

[java] view plain copy print?

```
1. MN = largerTrianShortSize = an / (sizeLong - (mViewHeight - mPointY)) * (mVi  
ewWidth - mPointX);  
2. QN = smallTrianShortSize = an / sizeLong * sizeShort;
```

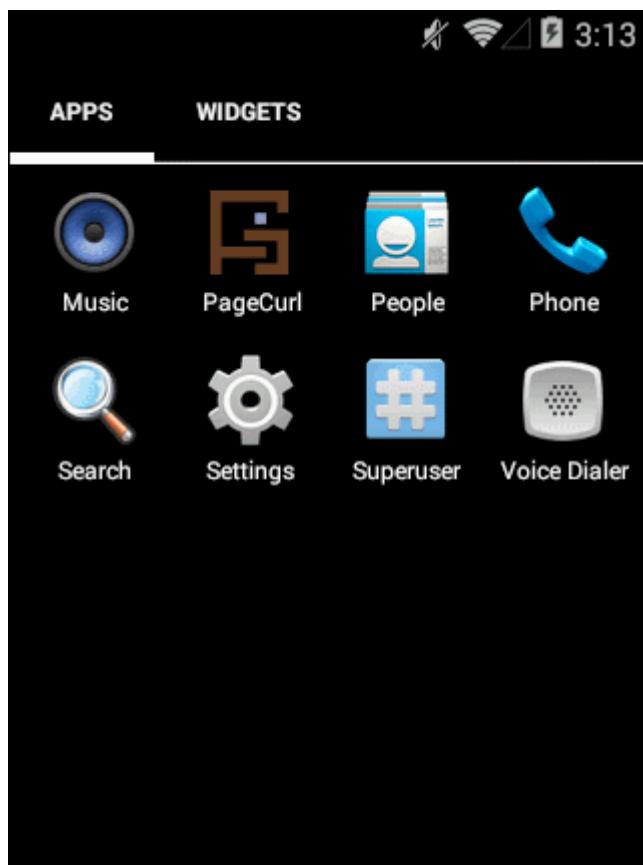
在 `onDraw` 中我们在计算出 `sizeLong` 和 `sizeShort` 后加一步判断：

[java] view plain copy print?

```
1. /*  
2.  * 计算短边长边长度  
3. */  
4. float sizeShort = temp / (2F * mL);  
5. float sizeLong = temp / (2F * mL);  
6.  
7. // 移动路径起点至触摸点  
8. mPath.moveTo(mPointX, mPointY);  
9.  
10. if (sizeLong > mViewHeight) {  
11.     // 计算.....额.....按图来 AN 边~  
12.     float an = sizeLong - mViewHeight;  
13.  
14.     // 三角形 AMN 的 MN 边  
15.     float largerTrianShortSize = an / (sizeLong - (mViewHeight - mPointY)) *  
16.         (mViewWidth - mPointX);  
17.     // 三角形 AQN 的 QN 边  
18.     float smallTrianShortSize = an / sizeLong * sizeShort;  
19.  
20.     /*  
21.      * 生成四边形路径  
22.      */  
23.     mPath.lineTo(mViewWidth - largerTrianShortSize, 0);  
24.     mPath.lineTo(mViewWidth - smallTrianShortSize, 0);  
25.     mPath.lineTo(mViewWidth - sizeShort, mViewHeight);  
26.     mPath.close();  
27. } else {  
28.     /*  
29.      * 生成三角形路径  
30.      */  
31.     mPath.lineTo(mViewWidth, mViewHeight - sizeLong);  
32.     mPath.lineTo(mViewWidth - sizeShort, mViewHeight);
```

```
33.     mPath.close();  
34. }  
35.  
36. // 绘制路径  
37. canvas.drawPath(mPath, mPaint);
```

来看看具体效果：



挺不错的感觉，好，继续！

折线是 OK 了，剩下的问题是如何将我们的图片显示，也就是上一节的内容融合进来呢？在此之前，我们要知道的是折叠区域、当前页和下一页这三部分显示的是不同的内容，如文章开头的示图：



那么如何显示不同的图片就必须要先将这三部分表达出来对吧，我们之前曾学过 **Region** 区域对象，在这里就可以派上用场。首先，我们可以考虑将整个控件分为三个区域：显示当前页的区域、显示折叠的区域（也就是当前页的背面）、显示下一页的区域。与上图类似，我们可以利用图层的功能使用三个图层来模拟。第一步我们声明一个 **Region** 类型的引用来定义当前页的区域：

```
[java] view plaincopyprint?
```

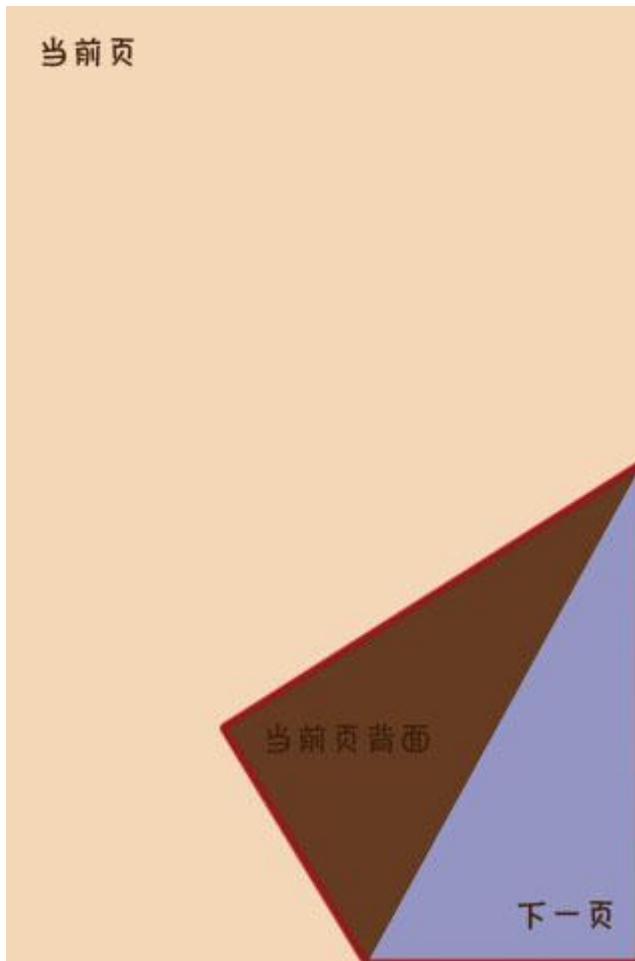
```
1. private Region mRegionCurrent; // 当前页区域，其实就是控件的大小
```

因为当前页的区域其实就是控件大小（置于最底层无所谓了~），我们直接就可以在 **onSizeChanged** 中完成对象的生成：

```
[java] view plaincopyprint?
```

```
1. // 计算当前页区域  
2. mRegionCurrent.set(0, 0, mViewWidth, mViewHeight);
```

尔后我们需要计算折叠区域和下一页的路径：



如图红色线条所示路径，我们需要这部分区域来计算下一页的区域，即：下一页区域=线条部分区域-折叠区域对吧，同样我们声明一个 Path 类型的引用来定义该部分 Path:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. private Path mPathFoldAndNext; // 一个包含折叠和下一页区域的 Path
```

在 `onDraw` 中在计算折叠区域的同时计算该部分区域:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 计算短边长边长度
3. */
4. float sizeShort = temp / (2F * mK);
5. float sizeLong = temp / (2F * mL);
6.
7. // 移动路径起点至触摸点
8. mPath.moveTo(mPointX, mPointY);
9. mPathFoldAndNext.moveTo(mPointX, mPointY);
10.
```

```
11. if (sizeLong > mViewHeight) {  
12.     // 计算.....额.....按图来 AN 边~  
13.     float an = sizeLong - mViewHeight;  
14.  
15.     // 三角形 AMN 的 MN 边  
16.     float largerTrianShortSize = an / (sizeLong - (mViewHeight - mPointY)) *  
        (mViewWidth - mPointX);  
17.  
18.     // 三角形 AQN 的 QN 边  
19.     float smallTrianShortSize = an / sizeLong * sizeShort;  
20.  
21.     /*  
22.      * 计算参数  
23.      */  
24.     float topX1 = mViewWidth - largerTrianShortSize;  
25.     float topX2 = mViewWidth - smallTrianShortSize;  
26.     float btmX2 = mViewWidth - sizeShort;  
27.  
28.     /*  
29.      * 生成四边形路径  
30.      */  
31.     mPath.lineTo(topX1, 0);  
32.     mPath.lineTo(topX2, 0);  
33.     mPath.lineTo(btmX2, mViewHeight);  
34.     mPath.close();  
35.  
36.     /*  
37.      * 生成包含折叠和下一页的路径  
38.      */  
39.     mPathFoldAndNext.lineTo(topX1, 0);  
40.     mPathFoldAndNext.lineTo(mViewWidth, 0);  
41.     mPathFoldAndNext.lineTo(mViewWidth, mViewHeight);  
42.     mPathFoldAndNext.lineTo(btmX2, mViewHeight);  
43.     mPathFoldAndNext.close();  
44. } else {  
45.     /*  
46.      * 计算参数  
47.      */  
48.     float leftY = mViewHeight - sizeLong;  
49.     float btmX = mViewWidth - sizeShort;  
50.  
51.     /*  
52.      * 生成三角形路径  
53.      */
```

```
54.     mPath.lineTo(mViewWidth, leftY);
55.     mPath.lineTo(btmX, mViewHeight);
56.     mPath.close();
57.
58.     /*
59.      * 生成包含折叠和下一页的路径
60.      */
61.     mPathFoldAndNext.lineTo(mViewWidth, leftY);
62.     mPathFoldAndNext.lineTo(mViewWidth, mViewHeight);
63.     mPathFoldAndNext.lineTo(btmX, mViewHeight);
64.     mPathFoldAndNext.close();
65. }
```

将路径转换为 **Region** 我们独立一个方法来调用:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2.  * 通过路径计算区域
3. *
4.  * @param path
5.  *          路径对象
6.  * @return 路径的 Region
7. */
8. private Region computeRegion(Path path) {
9.     Region region = new Region();
10.    RectF f = new RectF();
11.    path.computeBounds(f, true);
12.    region.setPath(path, new Region((int) f.left, (int) f.top, (int) f.right
13.        , (int) f.bottom));
14.    return region;
15. }
```

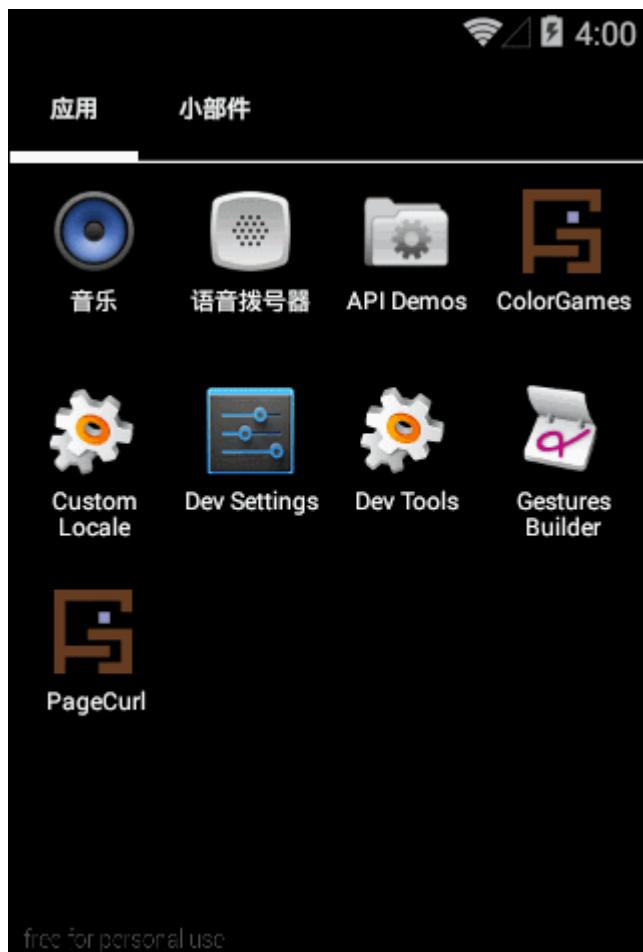
之后我们就可以计算并绘制这三部分区域:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 定义区域
3. */
4. Region regionFold = null;
5. Region regionNext = null;
6.
7. */
```

```
8.     * 通过路径生成区域
9.     */
10.    regionFold = computeRegion(mPath);
11.    regionNext = computeRegion(mPathFoldAndNext);
12.
13. /*
14.     * 计算当前页的区域
15. */
16.    canvas.save();
17.    canvas.clipRegion(mRegionCurrent);
18.    canvas.clipRegion(regionNext, Region.Op.DIFFERENCE);
19.    canvas.drawColor(0xFFFF4D8B7);
20.    canvas.restore();
21.
22. /*
23.     * 计算折叠页的区域
24. */
25.    canvas.save();
26.    canvas.clipRegion(regionFold);
27.    canvas.drawColor(0xFF663C21);
28.    canvas.restore();
29.
30. /*
31.     * 计算下一页的区域
32. */
33.    canvas.save();
34.    canvas.clipRegion(regionNext);
35.    canvas.clipRegion(regionFold, Region.Op.DIFFERENCE);
36.    canvas.drawColor(0xFF9596C4);
37.    canvas.restore();
```

我们看看效果是否与我们期待的一致：



差不多对吧，只是有点小问题，我们在没触摸之前显示的是空白这很好解决，下面我们结合上一节的内容把图片的效果也整合进来，该部分全部的代码如下，我就直接贴出来了：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. package com.aigestudio.pagecurl.views;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. import android.annotation.SuppressLint;
7. import android.content.Context;
8. import android.graphics.Bitmap;
9. import android.graphics.Canvas;
10. import android.graphics.Color;
11. import android.graphics.Paint;
12. import android.graphics.Path;
13. import android.graphics.RectF;
14. import android.graphics.Region;
15. import android.os.Handler;
16. import android.os.Message;
17. import android.text.TextPaint;
```

```
18. import android.util.AttributeSet;
19. import android.view.MotionEvent;
20. import android.view.View;
21. import android.widget.Toast;
22.
23. /**
24.  * 折叠 View
25. *
26. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
27. * @version 1.0.0
28. * @since 2014/12/27
29. */
30. public class FoldView extends View {
31.     private static final float VALUE_ADDED = 1 / 500F;// 精度附加值占比
32.     private static final float BUFF_AREA = 1 / 50F;// 底部缓冲区域占比
33.     private static final float AUTO_AREA_BUTTON_RIGHT = 3 / 4F, AUTO_AREA_BU
    TTOM_LEFT = 1 / 8F;// 右下角和左侧自滑区域占比
34.     private static final float AUTO_SLIDE_BL_V = 1 / 25F, AUTO_SLIDE_BR_V =
    1 / 100F;// 滑动速度占比
35.     private static final float TEXT_SIZE_NORMAL = 1 / 40F, TEXT_SIZE_LARGER
    = 1 / 20F;// 标准文字尺寸和大号文字尺寸的占比
36.
37.     private List<Bitmap> mBitmaps;// 位图数据列表
38.
39.     private SlideHandler mSlideHandler;// 滑动处理 Handler
40.     private Paint mPaint;// 画笔
41.     private TextPaint mTextPaint;// 文本画笔
42.     private Context mContext;// 上下文环境引用
43.
44.     private Path mPath;// 折叠路径
45.     private Path mPathFoldAndNext;// 一个包含折叠和下一页区域的 Path
46.
47.     private Region mRegionShortSize;// 短边的有效区域
48.     private Region mRegionCurrent;// 当前页区域，其实就是控件的大小
49.
50.     private int mViewWidth, mViewHeight;// 控件宽高
51.     private int mPageIndex;// 当前显示 mBitmaps 数据的下标
52.
53.     private float mPointX, mPointY;// 手指触摸点的坐标
54.     private float mValueAdded;// 精度附减值
55.     private float mBuffArea;// 底部缓冲区域
56.     private float mAutoAreaBottom, mAutoAreaRight, mAutoAreaLeft;// 右下角和
    左侧自滑区域
57.     private float mStart_X, mStart_Y;// 直线起点坐标
```

```
58.     private float mAutoSlideV_BL, mAutoSlideV_BR;// 滑动速度
59.     private float mTextSizeNormal, mTextSizeLarger;// 标准文字尺寸和大号文字尺
寸
60.     private float mDegrees;// 当前Y边长与Y轴的夹角
61.
62.     private boolean isSlide, isLastPage, isNextPage;// 是否执行滑动、是否已到最
后一页、是否可显示下一页的标识值
63.
64.     private Slide mSlide;// 定义当前滑动是往左下滑还是右下滑
65.
66.     /**
67.      * 枚举类定义滑动方向
68.      */
69.     private enum Slide {
70.         LEFT_BOTTOM, RIGHT_BOTTOM
71.     }
72.
73.     private Ratio mRatio;// 定义当前折叠边长
74.
75.     /**
76.      * 枚举类定义长边短边
77.      */
78.     private enum Ratio {
79.         LONG, SHORT
80.     }
81.
82.     public FoldView(Context context, AttributeSet attrs) {
83.         super(context, attrs);
84.         mContext = context;
85.
86.         /*
87.          * 实例化文本画笔并设置参数
88.          */
89.         mTextPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG
| Paint.LINEAR_TEXT_FLAG);
90.         mTextPaint.setTextAlign(Paint.Align.CENTER);
91.
92.         /*
93.          * 实例化画笔对象并设置参数
94.          */
95.         mPaint = new Paint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG);
96.         mPaint.setStyle(Paint.Style.STROKE);
97.         mPaint.setStrokeWidth(2);
98.
```

```
99.         /*
100.             * 实例化路径对象
101.             */
102.             mPath = new Path();
103.             mPathFoldAndNext = new Path();
104.
105.             /*
106.                 * 实例化区域对象
107.                 */
108.             mRegionShortSize = new Region();
109.             mRegionCurrent = new Region();
110.
111.             // 实例化滑动 Handler 处理器
112.             mSlideHandler = new SlideHandler();
113.         }
114.
115.     @Override
116.     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
117.         /*
118.             * 获取控件宽高
119.             */
120.         mViewWidth = w;
121.         mViewHeight = h;
122.
123.         // 初始化位图数据
124.         if (null != mBitmaps) {
125.             initBitmaps();
126.         }
127.
128.         // 计算文字尺寸
129.         mTextSizeNormal = TEXT_SIZE_NORMAL * mViewHeight;
130.         mTextSizeLarger = TEXT_SIZE_LARGER * mViewHeight;
131.
132.         // 计算精度附加值
133.         mValueAdded = mViewHeight * VALUE_ADDED;
134.
135.         // 计算底部缓冲区域
136.         mBuffArea = mViewHeight * BUFF_AREA;
137.
138.         /*
139.             * 计算自滑位置
140.             */
141.         mAutoAreaButton = mViewHeight * AUTO_AREA_BUTTON_RIGHT;
142.         mAutoAreaRight = mViewWidth * AUTO_AREA_BUTTON_RIGHT;
```

```
143.         mAutoAreaLeft = mViewWidth * AUTO_AREA_BUTTON_LEFT;
144.
145.         // 计算短边的有效区域
146.         computeShortSizeRegion();
147.
148.         /*
149.             * 计算滑动速度
150.         */
151.         mAutoSlideV_BL = mViewWidth * AUTO_SLIDE_BL_V;
152.         mAutoSlideV_BR = mViewWidth * AUTO_SLIDE_BR_V;
153.
154.         // 计算当前页区域
155.         mRegionCurrent.set(0, 0, mViewWidth, mViewHeight);
156.     }
157.
158. /**
159. * 初始化位图数据
160. * 缩放位图尺寸与屏幕匹配
161. */
162. private void initBitmaps() {
163.     List<Bitmap> temp = new ArrayList<Bitmap>();
164.     for (int i = mBitmaps.size() - 1; i >= 0; i--) {
165.         Bitmap bitmap = Bitmap.createScaledBitmap(mBitmaps.get(i), mViewWidth, mViewHeight, true);
166.         temp.add(bitmap);
167.     }
168.     mBitmaps = temp;
169. }
170.
171. /**
172. * 计算短边的有效区域
173. */
174. private void computeShortSizeRegion() {
175.     // 短边圆形路径对象
176.     Path pathShortSize = new Path();
177.
178.     // 用来装载 Path 边界值的 RectF 对象
179.     RectF rectShortSize = new RectF();
180.
181.     // 添加圆形到 Path
182.     pathShortSize.addCircle(0, mViewHeight, mViewWidth, Path.Direction.CCW);
183.
184.     // 计算边界
```

```
185.         pathShortSize.computeBounds(rectShortSize, true);
186.
187.         // 将 Path 转化为 Region
188.         mRegionShortSize.setPath(pathShortSize, new Region((int) rectShortS
189.             ize.left, (int) rectShortSize.top, (int) rectShortSize.right, (int) rectShortS
190.                 tSize.bottom));
191.     }
192.
193.     @Override
194.     protected void onDraw(Canvas canvas) {
195.         /*
196.          * 如果数据为空则显示默认提示文本
197.          */
198.         if (null == mBitmaps || mBitmaps.size() == 0) {
199.             defaultDisplay(canvas);
200.             return;
201.         }
202.
203.         // 重绘时重置路径
204.         mPath.reset();
205.         mPathFoldAndNext.reset();
206.
207.         /*
208.          * 如果坐标点在原点（即还没发生触碰时）则绘制第一页
209.          */
210.         if (mPointX == 0 && mPointY == 0) {
211.             canvas.drawBitmap(mBitmaps.get(mBitmaps.size() - 1), 0, 0, null
212. );
213.             return;
214.         }
215.
216.         /*
217.          * 判断触摸点是否在短边的有效区域内
218.          */
219.         if (!mRegionShortSize.contains((int) mPointX, (int) mPointY)) {
220.             // 如果不在则通过 x 坐标强行重算 y 坐标
221.             mPointY = (float) (Math.sqrt((Math.pow(mViewWidth, 2) - Math.po
222.                 w(mPointX, 2))) - mViewHeight);
223.
224.             // 精度附加值避免精度损失
225.             mPointY = Math.abs(mPointY) + mValueAdded;
```

```
225.         }
226.
227.         /*
228.          * 缓冲区域判断
229.          */
230.         float area = mViewHeight - mBuffArea;
231.         if (!isSlide && mPointY >= area) {
232.             mPointY = area;
233.         }
234.
235.         /*
236.          * 额，这个该怎么注释好呢.....根据图来
237.          */
238.         float mK = mViewWidth - mPointX;
239.         float mL = mViewHeight - mPointY;
240.
241.         // 需要重复使用的参数存值避免重复计算
242.         float temp = (float) (Math.pow(mL, 2) + Math.pow(mK, 2));
243.
244.         /*
245.          * 计算短边长边长度
246.          */
247.         float sizeShort = temp / (2F * mK);
248.         float sizeLong = temp / (2F * mL);
249.
250.         /*
251.          * 根据长短边边长计算旋转角度并确定 mRatio 的值
252.          */
253.         if (sizeShort < sizeLong) {
254.             mRatio = Ratio.SHORT;
255.             float sin = (mK - sizeShort) / sizeShort;
256.             mDegrees = (float) (Math.asin(sin) / Math.PI * 180);
257.         } else {
258.             mRatio = Ratio.LONG;
259.             float cos = mK / sizeLong;
260.             mDegrees = (float) (Math.acos(cos) / Math.PI * 180);
261.         }
262.
263.         // 移动路径起点至触摸点
264.         mPath.moveTo(mPointX, mPointY);
265.         mPathFoldAndNext.moveTo(mPointX, mPointY);
266.
267.         if (sizeLong > mViewHeight) {
268.             // 计算.....额.....按图来 AN 边~
```

```
269.         float an = sizeLong - mViewHeight;
270.
271.         // 三角形 AMN 的 MN 边
272.         float largerTrianShortSize = an / (sizeLong - (mViewHeight - mP
273. ointY)) * (mViewWidth - mPointX);
274.
275.         // 三角形 AQN 的 QN 边
276.         float smallTrianShortSize = an / sizeLong * sizeShort;
277.
278.         /*
279.          * 计算参数
280.          */
281.         float topX1 = mViewWidth - largerTrianShortSize;
282.         float topX2 = mViewWidth - smallTrianShortSize;
283.         float btmX2 = mViewWidth - sizeShort;
284.
285.         /*
286.          * 生成四边形路径
287.          */
288.         mPath.lineTo(topX1, 0);
289.         mPath.lineTo(topX2, 0);
290.         mPath.lineTo(btmX2, mViewHeight);
291.         mPath.close();
292.
293.         /*
294.          * 生成包含折叠和下一页的路径
295.          */
296.         mPathFoldAndNext.lineTo(topX1, 0);
297.         mPathFoldAndNext.lineTo(mViewWidth, 0);
298.         mPathFoldAndNext.lineTo(mViewWidth, mViewHeight);
299.         mPathFoldAndNext.lineTo(btmX2, mViewHeight);
300.         mPathFoldAndNext.close();
301.     } else {
302.         /*
303.          * 计算参数
304.          */
305.         float leftY = mViewHeight - sizeLong;
306.         float btmX = mViewWidth - sizeShort;
307.
308.         /*
309.          * 生成三角形路径
310.          */
311.         mPath.lineTo(mViewWidth, leftY);
312.         mPath.lineTo(btmX, mViewHeight);
```

```
312.         mPath.close();
313.
314.         /*
315.          * 生成包含折叠和下一页的路径
316.          */
317.         mPathFoldAndNext.lineTo(mViewWidth, leftY);
318.         mPathFoldAndNext.lineTo(mViewWidth, mViewHeight);
319.         mPathFoldAndNext.lineTo(btmX, mViewHeight);
320.         mPathFoldAndNext.close();
321.     }
322.
323.     drawBitmaps(canvas);
324. }
325.
326. /**
327.  * 绘制位图数据
328. *
329. * @param canvas
330. *          画布对象
331. */
332. private void drawBitmaps(Canvas canvas) {
333.     // 绘制位图前重置 isLastPage 为 false
334.     isLastPage = false;
335.
336.     // 限制 pageIndex 的值范围
337.     mPageIndex = mPageIndex < 0 ? 0 : mPageIndex;
338.     mPageIndex = mPageIndex > mBitmaps.size() ? mBitmaps.size() : mPage
Index;
339.
340.     // 计算数据起始位置
341.     int start = mBitmaps.size() - 2 - mPageIndex;
342.     int end = mBitmaps.size() - mPageIndex;
343.
344.     /*
345.      * 如果数据起点位置小于 0 则表示当前已经到了最后一张图片
346.      */
347.     if (start < 0) {
348.         // 此时设置 isLastPage 为 true
349.         isLastPage = true;
350.
351.         // 并显示提示信息
352.         showToast("This is fucking lastest page");
353.
354.         // 强制重置起始位置
```

```
355.         start = 0;
356.         end = 1;
357.     }
358.
359.     /*
360.      * 定义区域
361.      */
362.     Region regionFold = null;
363.     Region regionNext = null;
364.
365.     /*
366.      * 通过路径生成区域
367.      */
368.     regionFold = computeRegion(mPath);
369.     regionNext = computeRegion(mPathFoldAndNext);
370.
371.     /*
372.      * 计算当前页的区域
373.      */
374.     canvas.save();
375.     canvas.clipRegion(mRegionCurrent);
376.     canvas.clipRegion(regionNext, Region.Op.DIFFERENCE);
377.     canvas.drawBitmap(mBitmaps.get(end - 1), 0, 0, null);
378.     canvas.restore();
379.
380.     /*
381.      * 计算折叠页的区域
382.      */
383.     canvas.save();
384.     canvas.clipRegion(regionFold);
385.
386.     canvas.translate(mPointX, mPointY);
387.
388.     /*
389.      * 根据长短边标识计算折叠区域图像
390.      */
391.     if (mRatio == Ratio.SHORT) {
392.         canvas.rotate(90 - mDegrees);
393.         canvas.translate(0, -mViewHeight);
394.         canvas.scale(-1, 1);
395.         canvas.translate(-mViewWidth, 0);
396.     } else {
397.         canvas.rotate(-(90 - mDegrees));
398.         canvas.translate(-mViewWidth, 0);
```

```
399.         canvas.scale(1, -1);
400.         canvas.translate(0, -mViewHeight);
401.     }
402.
403.     canvas.drawBitmap(mBitmaps.get(end - 1), 0, 0, null);
404.     canvas.restore();
405.
406.     /*
407.      * 计算下一页的区域
408.      */
409.     canvas.save();
410.     canvas.clipRegion(regionNext);
411.     canvas.clipRegion(regionFold, Region.Op.DIFFERENCE);
412.     canvas.drawBitmap(mBitmaps.get(start), 0, 0, null);
413.     canvas.restore();
414. }
415.
416. /**
417.  * 默认显示
418. *
419. * @param canvas
420. *          Canvas 对象
421. */
422. private void defaultDisplay(Canvas canvas) {
423.     // 绘制底色
424.     canvas.drawColor(Color.WHITE);
425.
426.     // 绘制标题文本
427.     mTextPaint.setTextSize(mTextSizeLarger);
428.     mTextPaint.setColor(Color.RED);
429.     canvas.drawText("FBI WARNING", mViewWidth / 2, mViewHeight / 4, mTe
xtPaint);
430.
431.     // 绘制提示文本
432.     mTextPaint.setTextSize(mTextSizeNormal);
433.     mTextPaint.setColor(Color.BLACK);
434.     canvas.drawText("Please set data use setBitmaps method", mViewWidth
/ 2, mViewHeight / 3, mTextPaint);
435. }
436.
437. /**
438.  * 通过路径计算区域
439. *
440. * @param path
```

```
441.     *          路径对象
442.     * @return 路径的 Region
443.     */
444.     private Region computeRegion(Path path) {
445.         Region region = new Region();
446.         RectF f = new RectF();
447.         path.computeBounds(f, true);
448.         region.setPath(path, new Region((int) f.left, (int) f.top, (int) f.
449.             right, (int) f.bottom));
450.         return region;
451.     }
452.     /**
453.      * 计算滑动参数变化
454.      */
455.     private void slide() {
456.         /*
457.          * 如果滑动标识值为 false 则返回
458.          */
459.         if (!isSlide) {
460.             return;
461.         }
462.         /*
463.          * 如果当前页不是最后一页
464.          * 如果是需要翻下一页
465.          * 并且上一页已被做掉
466.          */
467.         if (!isLastPage && isNextPage && (mPointX - mAutoSlideV_BL <= -mVie
468.             wWidth)) {
469.             mPointX = -mViewWidth;
470.             mPointY = mViewHeight;
471.             mPageIndex++;
472.             invalidate();
473.         }
474.         /*
475.          * 如果当前滑动标识为向右下滑动 x 坐标恒小于控件宽度
476.          */
477.         else if (mSlide == Slide.RIGHT_BOTTOM && mPointX < mViewWidth) {
478.             // 则让 x 坐标自加
479.             mPointX += mAutoSlideV_BR;
480.
481.             // 并根据 x 坐标的值重新计算 y 坐标的值
482.         }
```

```
483.             mPointY = mStart_Y + ((mPointX - mStart_X) * (mViewHeight - mSt
   art_Y)) / (mViewWidth - mStart_X);
484.
485.             // 让 SlideHandler 处理重绘
486.             mSlideHandler.sleep(25);
487.         }
488.
489.         /*
490.          * 如果当前滑动标识为向左下滑动 x 坐标恒大于控件宽度的负值
491.          */
492.         else if (mSlide == Slide.LEFT_BOTTOM && mPointX > -mViewWidth) {
493.             // 则让 x 坐标自减
494.             mPointX -= mAutoSlideV_BL;
495.
496.             // 并根据 x 坐标的值重新计算 y 坐标的值
497.             mPointY = mStart_Y + ((mPointX - mStart_X) * (mViewHeight - mSt
   art_Y)) / (-mViewWidth - mStart_X);
498.
499.             // 让 SlideHandler 处理重绘
500.             mSlideHandler.sleep(25);
501.         }
502.     }
503.
504. /**
505.  * 为 isSlide 提供对外的停止方法便于必要时释放滑动动画
506. */
507. public void slideStop() {
508.     isSlide = false;
509. }
510.
511. /**
512.  * 提供对外的方法获取 View 内 Handler
513. *
514.  * @return mSlideHandler
515. */
516. public SlideHandler getSlideHandler() {
517.     return mSlideHandler;
518. }
519.
520. @Override
521. public boolean onTouchEvent(MotionEvent event) {
522.     isNextPage = true;
523.
524.     /*
```

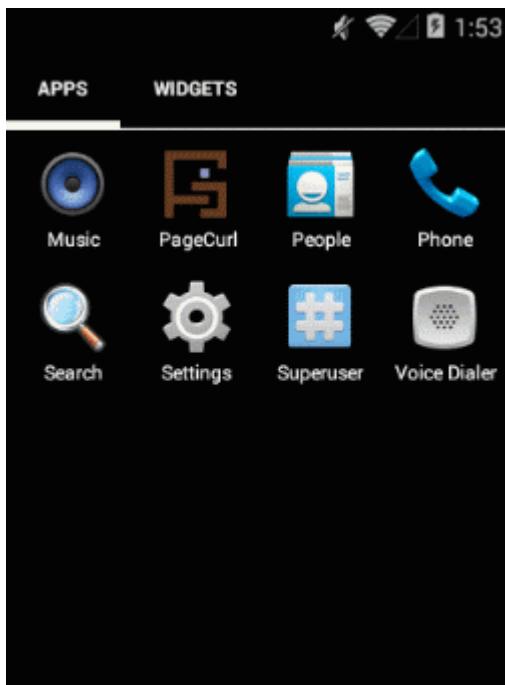
```
525.         * 获取当前事件点
526.         */
527.         float x = event.getX();
528.         float y = event.getY();
529.
530.         switch (event.getAction() & MotionEvent.ACTION_MASK) {
531.             case MotionEvent.ACTION_UP:// 手指抬起时候
532.                 if (isNextPage) {
533.                     /*
534.                      * 如果当前事件点位于右下自滑区域
535.                      */
536.                     if (x > mAutoAreaRight && y > mAutoAreaBottom) {
537.                         // 当前为往右下滑
538.                         mSlide = Slide.RIGHT_BOTTOM;
539.
540.                         // 摩擦吧骚年!
541.                         justSlide(x, y);
542.                     }
543.
544.                     /*
545.                      * 如果当前事件点位于左侧自滑区域
546.                      */
547.                     if (x < mAutoAreaLeft) {
548.                         // 当前为往左下滑
549.                         mSlide = Slide.LEFT_BOTTOM;
550.
551.                         // 摩擦吧骚年!
552.                         justSlide(x, y);
553.                     }
554.                 }
555.                 break;
556.             case MotionEvent.ACTION_DOWN:
557.                 isSlide = false;
558.                 /*
559.                  * 如果事件点位于回滚区域
560.                  */
561.                 if (x < mAutoAreaLeft) {
562.                     // 那就不翻下一页了而是上一页
563.                     isNextPage = false;
564.                     mPageIndex--;
565.                     mPointX = x;
566.                     mPointY = y;
567.                     invalidate();
568.                 }
569.             }
570.         }
571.     }
572. }
```

```
569.         downAndMove(event);
570.         break;
571.     case MotionEvent.ACTION_MOVE:
572.         downAndMove(event);
573.         break;
574.     }
575.     return true;
576. }
577.
578. /**
579. * 处理 DOWN 和 MOVE 事件
580. *
581. * @param event
582. *          事件对象
583. */
584. private void downAndMove(MotionEvent event) {
585.     if (!isLastPage) {
586.         mPointX = event.getX();
587.         mPointY = event.getY();
588.
589.         invalidate();
590.     }
591. }
592.
593. /**
594. * 在这光滑的地板上~
595. *
596. * @param x
597. *          当前触摸点 x
598. * @param y
599. *          当前触摸点 y
600. */
601. private void justSlide(float x, float y) {
602.     // 获取并设置直线方程的起点
603.     mStart_X = x;
604.     mStart_Y = y;
605.
606.     // OK 要开始滑动了哦~
607.     isSlide = true;
608.
609.     // 滑动
610.     slide();
611. }
612.
```

```
613.     /**
614.      * 设置位图数据
615.      *
616.      * @param bitmaps
617.      *          位图数据列表
618.      */
619.     public synchronized void setBitmaps(List<Bitmap> bitmaps) {
620.         /*
621.          * 如果数据为空则抛出异常
622.          */
623.         if (null == bitmaps || bitmaps.size() == 0)
624.             throw new IllegalArgumentException("no bitmap to display");
625.
626.         /*
627.          * 如果数据长度小于 2 则 GG 思密达
628.          */
629.         if (bitmaps.size() < 2)
630.             throw new IllegalArgumentException("fuck you and fuck to use im
ageview");
631.
632.         mBitmaps = bitmaps;
633.         invalidate();
634.     }
635.
636.     /**
637.      * Toast 显示
638.      *
639.      * @param msg
640.      *          Toast 显示文本
641.      */
642.     private void showToast(Object msg) {
643.         Toast.makeText(mContext, msg.toString(), Toast.LENGTH_SHORT).show()
644. ;
644.     }
645.
646.     /**
647.      * 处理滑动的 Handler
648.      */
649.     @SuppressLint("HandlerLeak")
650.     public class SlideHandler extends Handler {
651.         @Override
652.         public void handleMessage(Message msg) {
653.             // 循环调用滑动计算
654.             FoldView.this.slide();
```

```
655.  
656.        // 重绘视图  
657.        FoldView.this.invalidate();  
658.    }  
659.  
660.    /**  
661.     * 延迟向 Handler 发送消息实现时间间隔  
662.     *  
663.     * @param delayMillis  
664.     *          间隔时间  
665.     */  
666.    public void sleep(long delayMillis) {  
667.        this.removeMessages(0);  
668.        sendMessageDelayed(obtainMessage(0), delayMillis);  
669.    }  
670.}  
671.}
```

以上代码就是本节的全部内容，运行的效果如下：



代码部分除了折叠区域图像的生成外都是直接从上一节 COPY 过来，而折叠区域图像的生成为了便于大家的理解我分为了两种情况计算（如果你的图形学屏可以将其并入一个计算方式）即当长边大于短边和短边大于长边两种情况（注意文章开头我们的长短边声明），而相等的情况我们并入其中一种一并计算即可。具体的计算过程很简单，首先我们必定要移动 Canvas 使其原点与我们的触摸点对应：

```
[java] view plaincopyprint?
```

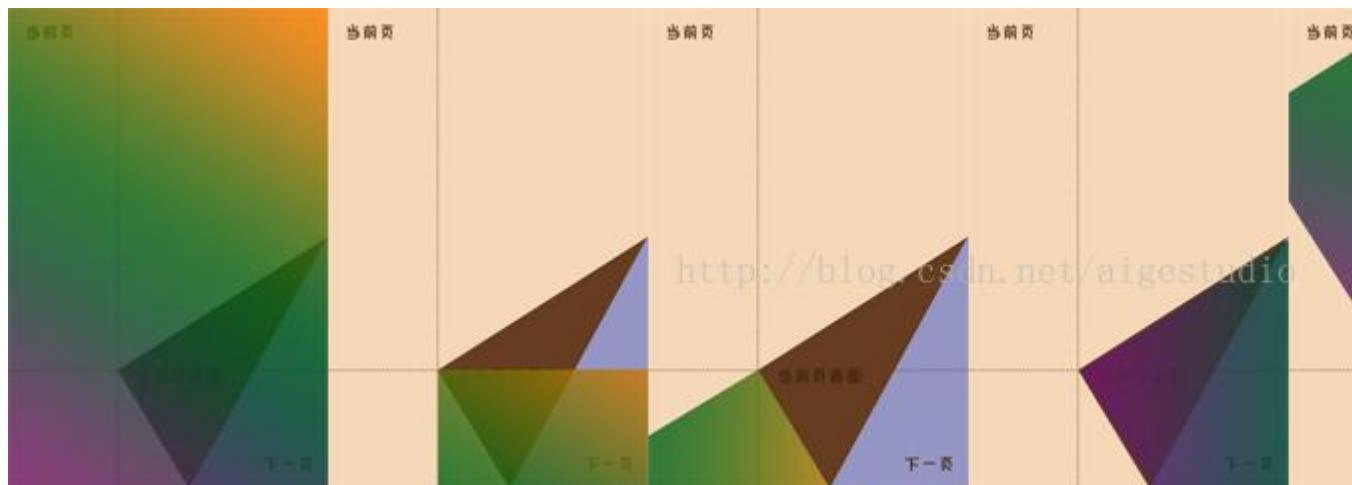
```
1. canvas.translate(mPointX, mPointY); //386 行
```

然后就是分情况了，第一种情况短边小于长边：

```
[java] view plaincopyprint?
```

```
1. //392-395 行  
2. canvas.rotate(90 - mDegrees);  
3. canvas.translate(0, -mViewHeight);  
4. canvas.scale(-1, 1);  
5. canvas.translate(-mViewWidth, 0);
```

这个过程图解的话就是：



而另一种情况短边大于长边：

```
[java] view plaincopyprint?
```

```
1. //397-400 行  
2. canvas.rotate(-(90 - mDegrees));  
3. canvas.translate(-mViewWidth, 0);  
4. canvas.scale(1, -1);  
5. canvas.translate(0, -mViewHeight);
```

该过程其实与上面类似、就不在画图了，图解很累的.....

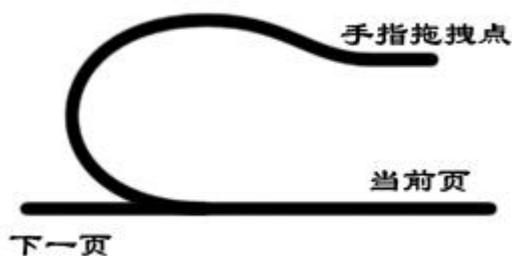
其他的跟上一节我们所讲的没有太大的出入。

OK，这一节到此为止，下一节我们将去尝试曲线的生成，如何将折线变为曲线并简单地实现扭曲的效果，thx all 谢谢大家、、

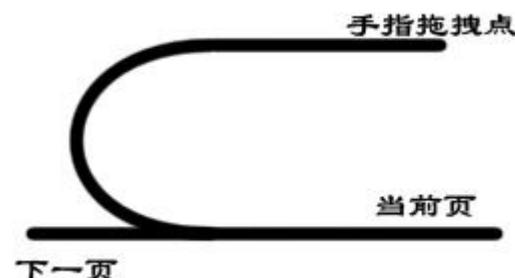
## 9. 自定义控件其实很简单(9)

上一节我们通过引入折线实现了页面的折叠翻转效果，有了前面两节的基础呢其实曲线的实现可以变得非常简单，为什么这么说呢？因为曲线无非就是在折线的基础上对 Path 加入了曲线的实现，进而只是影响了我们的 Region 区域，而其他的什么事件啊、滑动计算啊之类的几乎都是不变的对吧，说白了就是对现有的折线 View 进行 update 改造，虽然是改造，但是我们该如何下手呢？首先我们来看看现实中翻页的效果应该是怎样的呢？如果大家身边有书或本子甚至一张纸也行，尝试以不同的方式去翻动它，你会发现除了我们前面两节曾提到过的一些限制外，还有一些 special 的现象：

一、翻起来的区域从侧面来看是一个有弧度的区域，如图所示侧面图：



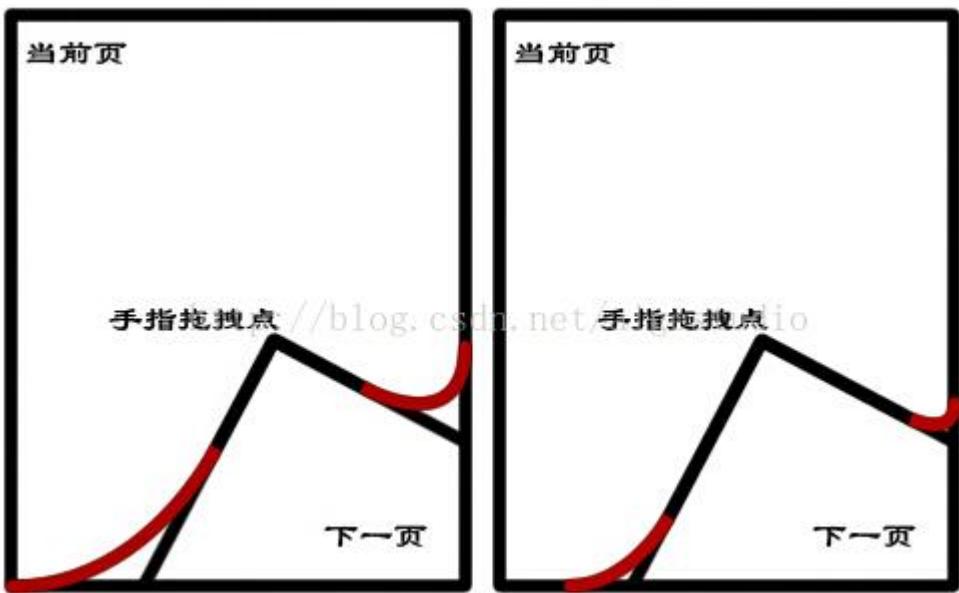
而我们将按照第一节中的约定忽略这部分弧度的表现，因为从正俯视的角度我们压根看不到弧度的效果，So~我们强制让其与页面平行：



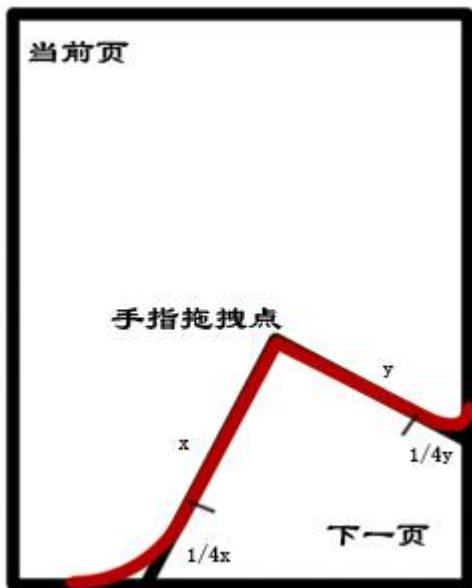
二、根据拖拽点距离页面高度的不同，我们可以得到不同的卷曲度：



而其在我们正俯视点的表现则是曲线的弧度不同：

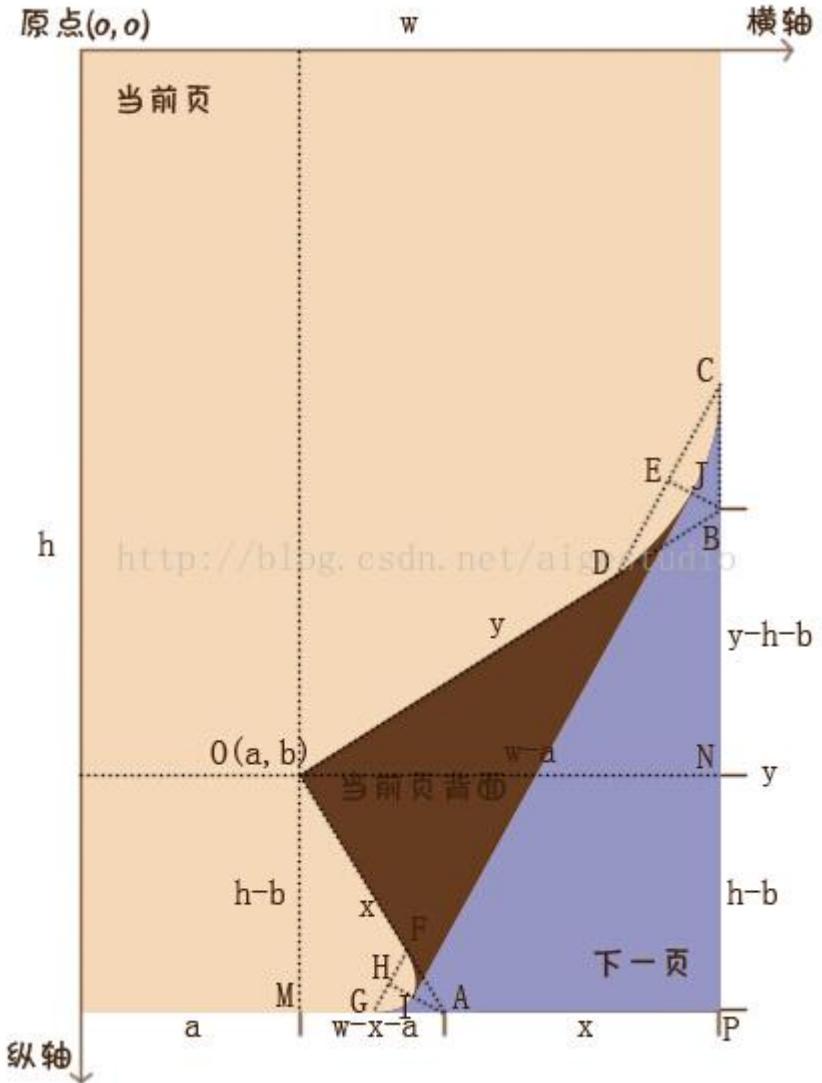


同样的，我们按照第一节的约定，为了简化问题，我们将拖拽点距离页面的高度视为一个定值使在我们正俯视点表现的曲线起点从距离控件交点  $1/4$  处开始：

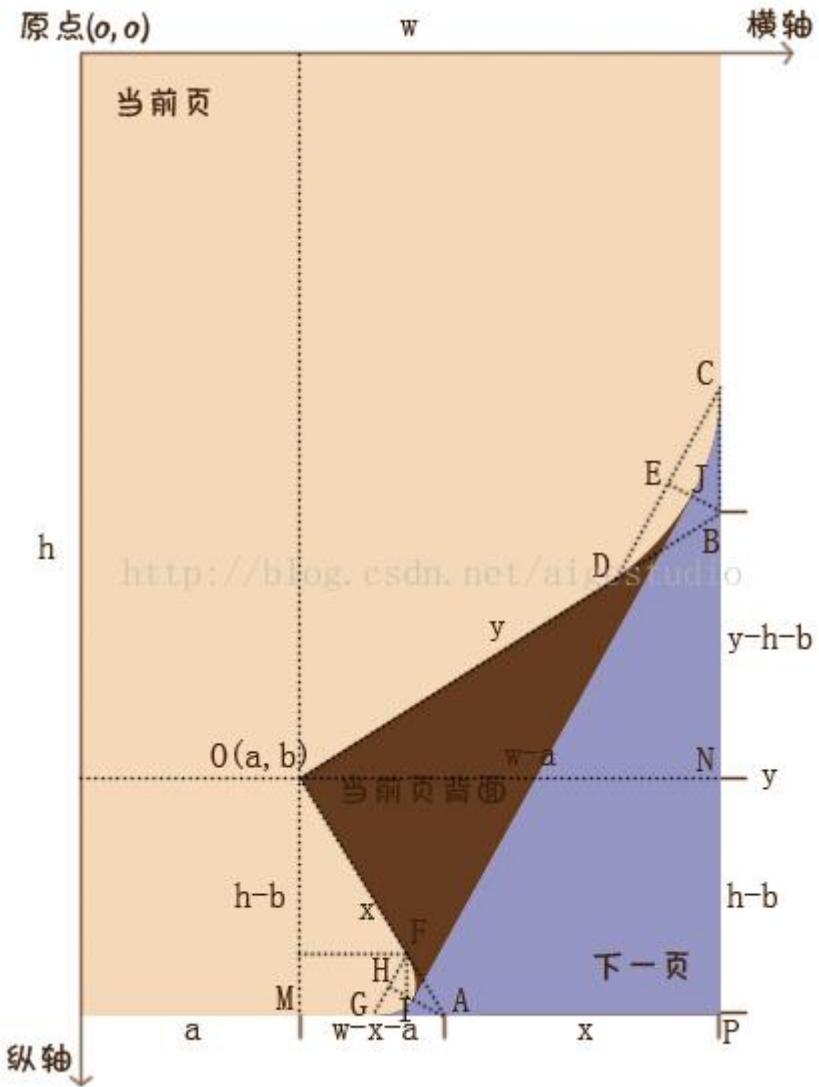


三、如上一节末所说，在弯曲的区域图像也会有相似的扭曲效果

OK，大致的一个分析就是这样，我们根据分析结果可以得出下面的一个分析图：



由上图配合我们上面的分析我们可知:  $DB = 1/4OB$ ,  $FA = 1/4OA$ , 而点 F 和点 D 分别为两条曲线 (如无特别声明, 我们所说的曲线均为贝赛尔曲线, 下同) 的起点 (当然你也可以说是终点无所谓), 这时, 我们以点 A、B 为曲线的控制点并以其为端点分别沿着 x 轴和 y 轴方向作线段 AG、BC, 另  $AG = AF$ 、 $BC = BD$ , 并令点 G、C 分别为曲线的终点, 这样, 我们的这两条二阶贝塞尔曲线就非常非常的特殊, 例如上图中的曲线 DC, 它是由起始点 D、C 和控制点 B 构成, 而  $BD = BC$ , 也就是说三角形 BDC 是的等腰三角形, 进一步地说就是曲线 DC 的两条控制杆力臂相等, 进一步地我们可以推断出曲线 DC 的顶点 J 必定在直线 DC 的中垂线上, 更进一步地我们可以根据《自定义控件其实很简单(5)》所说的二阶贝塞尔曲线公式得出当且仅当  $t = 0.5$  时曲线的端点刚好会在顶点 J 上, 由此我们可以非常非常简单地得到曲线的顶点坐标。好了, YY 归 YY 我们还是要回归到具体的操作中来, 首先, 我们要计算出点 G、F、D、C 的坐标值, 这四点坐标也相当 easy, 就拿 F 点坐标来说, 我们过点 F 分别作 OM、AM 的垂线:



因为  $FA = 1/4OA$ , 那么我们可以得到 F 点的 x 坐标  $Fx = a + 3/4MA$ , y 坐标  $Fy = b + 3/4OM$ , 而 G 点的 x 坐标  $Gx = a + MA - 1/4x$ ; 其他两点 D、C 就不多扯了, 那么在代码中如何体现呢? 首先, 为了便于观察效果, 我们先注释掉图片的绘制:

[java] view plain copy print?

```

1. /*
2.  * 如果坐标点在原点（即还没发生触碰时）则绘制第一页
3. */
4. if (mPointX == 0 && mPointY == 0) {
5.     // canvas.drawBitmap(mBitmaps.get(mBitmaps.size() - 1), 0, 0, null);
6.     return;
7. }
8.
9. // 省略大量代码
10.
11. //drawBitmaps(canvas);

```

并绘制线条：

```
[java] view plaincopyprint?
```

```
1. canvas.drawPath(mPath, mPaint);
```

在上一节中我们在生成 **Path** 时将情况分为了两种：

```
[java] view plaincopyprint?
```

```
1. if (sizeLong > mViewHeight) {  
2.     //.....  
3. } else {  
4.     //.....  
5. }
```

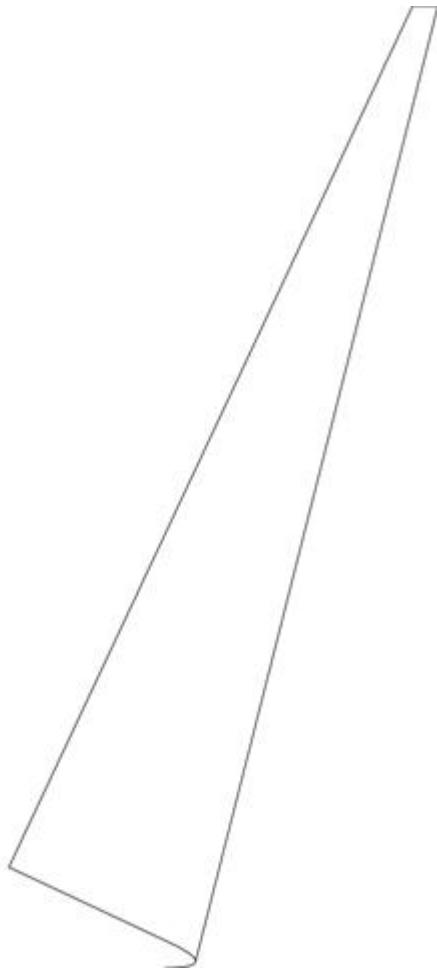
同样，我们也分开处理两种情况，那么针对 `sizeLong > mViewHeight` 的时候此时控件顶部的曲线效果已经是看不到了，我们只需考虑底部的曲线效果：

```
[java] view plaincopyprint?
```

```
1. // 计算曲线起点  
2. float startXBtm = btmX2 - CURVATURE * sizeShort;  
3. float startYBtm = mViewHeight;  
4.  
5. // 计算曲线终点  
6. float endXBtm = mPointX + (1 - CURVATURE) * (tempAM);  
7. float endYBtm = mPointY + (1 - CURVATURE) * mL;  
8.  
9. // 计算曲线控制点  
10. float controlXBtm = btmX2;  
11. float controlYBtm = mViewHeight;  
12.  
13. // 计算曲线顶点  
14. float bezierPeakXBtm = 0.25F * startXBtm + 0.5F * controlXBtm + 0.25F * endXBtm;  
15. float bezierPeakYBtm = 0.25F * startYBtm + 0.5F * controlYBtm + 0.25F * endYBtm;  
16.  
17. /*  
18. * 生成带曲线的四边形路径  
19. */  
20. mPath.moveTo(startXBtm, startYBtm);  
21. mPath.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
```

```
22. mPath.lineTo(mPointX, mPointY);
23. mPath.lineTo(topX1, 0);
24. mPath.lineTo(topX2, 0);
25. mPath.lineTo(bezierPeakXBottom, bezierPeakYBottom);
```

该部分的实际效果如下：



PS：为了便于大家对参数的理解，我对每一个点的坐标都重新给予了一个引用其命名也浅显易懂，实际过程可以省略这一步简化代码

而当 `sizeLong <= mViewHeight` 时这时候不但底部有曲线效果，右侧也有：

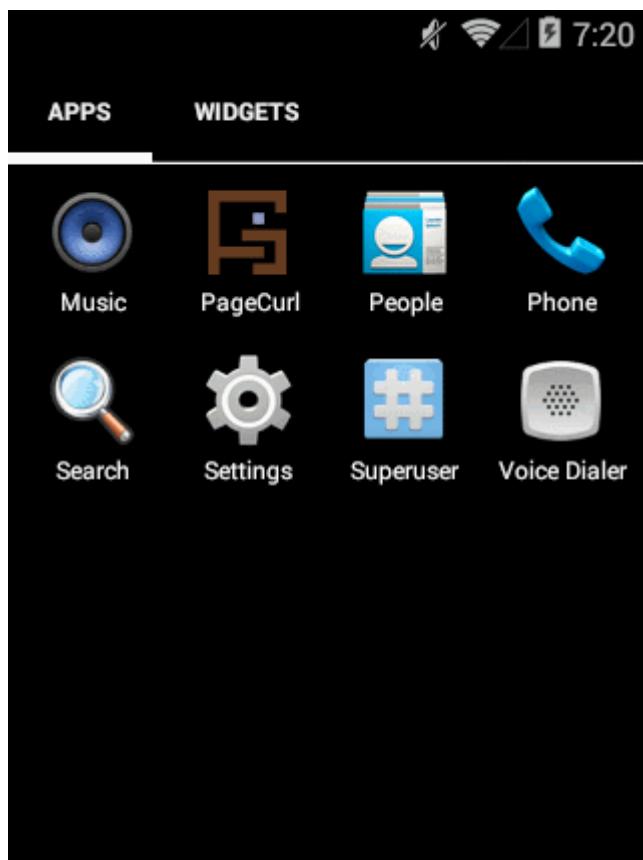
[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 计算参数
3. */
4. float leftY = mViewHeight - sizeLong;
5. float btmX = mViewWidth - sizeShort;
6.
7. // 计算曲线起点
```

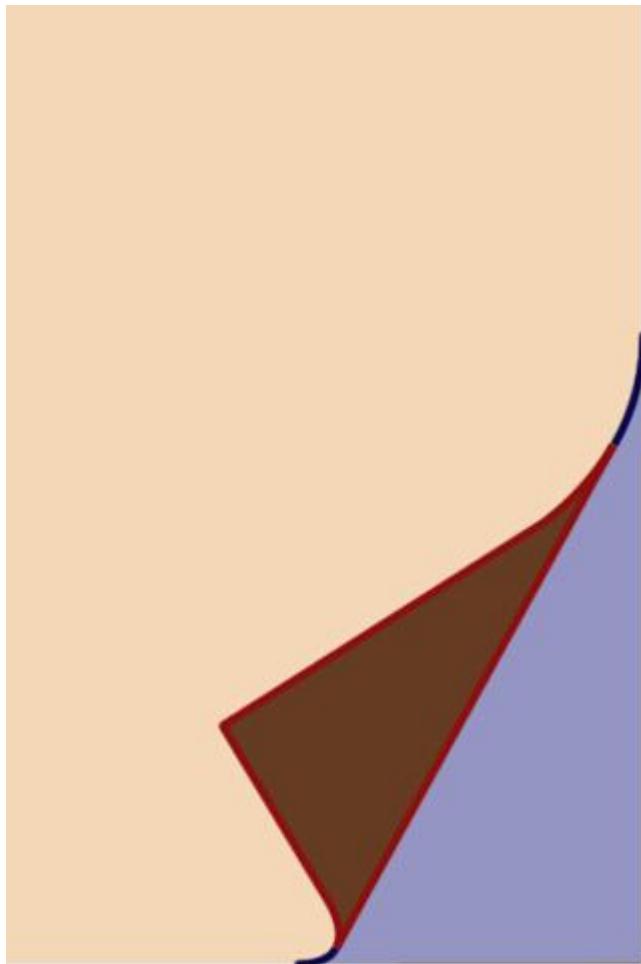
```
8. float startXBtm = btmX - CURVATURE * sizeShort;
9. float startYBtm = mViewHeight;
10. float startXLeft = mViewWidth;
11. float startYLeft = leftY - CURVATURE * sizeLong;
12.
13. /*
14.  * 限制左侧曲线起点
15. */
16. if (startYLeft <= 0) {
17.     startYLeft = 0;
18. }
19.
20. /*
21.  * 限制右侧曲线起点
22. */
23. if (startXBtm <= 0) {
24.     startXBtm = 0;
25. }
26.
27. // 计算曲线终点
28. float endXBtm = mPointX + (1 - CURVATURE) * (tempAM);
29. float endYBtm = mPointY + (1 - CURVATURE) * mL;
30. float endXLeft = mPointX + (1 - CURVATURE) * mK;
31. float endYLeft = mPointY - (1 - CURVATURE) * (sizeLong - mL);
32.
33. // 计算曲线控制点
34. float controlXBtm = btmX;
35. float controlYBtm = mViewHeight;
36. float controlXLeft = mViewWidth;
37. float controlYLeft = leftY;
38.
39. // 计算曲线顶点
40. float bezierPeakXBtm = 0.25F * startXBtm + 0.5F * controlXBtm + 0.25F * endX
   Btm;
41. float bezierPeakYBtm = 0.25F * startYBtm + 0.5F * controlYBtm + 0.25F * endY
   Btm;
42. float bezierPeakXLeft = 0.25F * startXLeft + 0.5F * controlXLeft + 0.25F * e
   ndXLeft;
43. float bezierPeakYLeft = 0.25F * startYLeft + 0.5F * controlYLeft + 0.25F * e
   ndYLeft;
44.
45. /*
46.  * 生成带曲线的三角形路径
47. */
```

```
48. mPath.moveTo(startXBtm, startYBtm);
49. mPath.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
50. mPath.lineTo(mPointX, mPointY);
51. mPath.lineTo(endXLeft, endYLeft);
52. mPath.quadTo(controlXLeft, controlYLeft, startXLeft, startYLeft);
```

效果如下：



Path 有了，我们就该考虑如何将其转换为 Region，在这个过程中呢又一个问题，曲线路径不像上一节的直线路径我们可以轻易获得其范围区域，因为我们的折叠区域其实应该是这样的：



如图所示红色路径区域，这部分区域则是我们折叠的区域，而事实上我们为了计算方便将整条二阶贝赛尔曲线都绘制了出来，也就是说我们的 **Path** 除了红色线条部分还包含了蓝色线条部分对吧，那么问题来了，如何将这两部分“做掉”呢？其实方法很多，我们可以在计算的时候就只生成半条曲线，这是方法一我们利用纯计算的方式，记得我在该系列文章开头曾说过翻页效果的实现可以有两种方式，一种是纯计算而另一种则是利用图形的组合思想，如何组合呢？这里对于区域的计算我们就不用纯计算的方式了，我们尝试用图形组合来试试。首先我们将 **Path** 转为 **Region** 看看是什么样的：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. Region region = computeRegion(mPath);
2. canvas.clipRegion(region);
3. canvas.drawColor(Color.RED);
4. // canvas.drawPath(mPath, mPaint);
```

效果如下：



可以看到我们没有封闭的 Path 形成的 Region 效果，事实呢跟我们需要的区域差距有点大，首先上下两个月半圆是多余的，其次目测少了一块对吧：



如上图蓝色的那块，那么我们该如何把这块“补”回来呢？利用图形组合的思想，我们设法为该 **Region** 补一块矩形：



然后差集掉两个月半圆不就成了？这部分代码改动较大，我先贴代码再说吧：

```
[java] view plaincopyprint?
1. if (sizeLong > mViewHeight) {
2.     // 计算.....额.....按图来 AN 边~
3.     float an = sizeLong - mViewHeight;
4.
5.     // 三角形 AMN 的 MN 边
6.     float largerTrianShortSize = an / (sizeLong - (mViewHeight - mPointY)) *
(mViewWidth - mPointX);
7.
8.     // 三角形 AQN 的 QN 边
9.     float smallTrianShortSize = an / sizeLong * sizeShort;
10.
11.    /*
12.     * 计算参数
13.     */
14.    float topX1 = mViewWidth - largerTrianShortSize;
15.    float topX2 = mViewWidth - smallTrianShortSize;
```

```
16.     float btmX2 = mViewWidth - sizeShort;
17.
18.     // 计算曲线起点
19.     float startXBtm = btmX2 - CURVATURE * sizeShort;
20.     float startYBtm = mViewHeight;
21.
22.     // 计算曲线终点
23.     float endXBtm = mPointX + (1 - CURVATURE) * (tempAM);
24.     float endYBtm = mPointY + (1 - CURVATURE) * mL;
25.
26.     // 计算曲线控制点
27.     float controlXBtm = btmX2;
28.     float controlYBtm = mViewHeight;
29.
30.     // 计算曲线顶点
31.     float bezierPeakXBtm = 0.25F * startXBtm + 0.5F * controlXBtm + 0.25F *
   endXBtm;
32.     float bezierPeakYBtm = 0.25F * startYBtm + 0.5F * controlYBtm + 0.25F *
   endYBtm;
33.
34.     /*
35.      * 生成带曲线的四边形路径
36.      */
37.     mPath.moveTo(startXBtm, startYBtm);
38.     mPath.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
39.     mPath.lineTo(mPointX, mPointY);
40.     mPath.lineTo(topX1, 0);
41.     mPath.lineTo(topX2, 0);
42.
43.     /*
44.      * 替补区域 Path
45.      */
46.     mPathTrap.moveTo(startXBtm, startYBtm);
47.     mPathTrap.lineTo(topX2, 0);
48.     mPathTrap.lineTo(bezierPeakXBtm, bezierPeakYBtm);
49.     mPathTrap.close();
50.
51.     /*
52.      * 底部月半圆 Path
53.      */
54.     mPathSemicircleBtm.moveTo(startXBtm, startYBtm);
55.     mPathSemicircleBtm.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
56.     mPathSemicircleBtm.close();
57.
```

```
58.     /*
59.      * 生成包含折叠和下一页的路径
60.      */
61.     //暂时没用省略掉
62.
63.     // 计算月半圆区域
64.     mRegionSemicircle = computeRegion(mPathSemicircleBtm);
65. } else {
66.     /*
67.      * 计算参数
68.      */
69.     float leftY = mViewHeight - sizeLong;
70.     float btmX = mViewWidth - sizeShort;
71.
72.     // 计算曲线起点
73.     float startXBtm = btmX - CURVATURE * sizeShort;
74.     float startYBtm = mViewHeight;
75.     float startXLeft = mViewWidth;
76.     float startYLeft = leftY - CURVATURE * sizeLong;
77.
78.     // 计算曲线终点
79.     float endXBtm = mPointX + (1 - CURVATURE) * (tempAM);
80.     float endYBtm = mPointY + (1 - CURVATURE) * mL;
81.     float endXLeft = mPointX + (1 - CURVATURE) * mK;
82.     float endYLeft = mPointY - (1 - CURVATURE) * (sizeLong - mL);
83.
84.     // 计算曲线控制点
85.     float controlXBtm = btmX;
86.     float controlYBtm = mViewHeight;
87.     float controlXLeft = mViewWidth;
88.     float controlYLeft = leftY;
89.
90.     // 计算曲线顶点
91.     float bezierPeakXBtm = 0.25F * startXBtm + 0.5F * controlXBtm + 0.25F *
endXBtm;
92.     float bezierPeakYBtm = 0.25F * startYBtm + 0.5F * controlYBtm + 0.25F *
endYBtm;
93.     float bezierPeakXLeft = 0.25F * startXLeft + 0.5F * controlXLeft + 0.25F *
endXLeft;
94.     float bezierPeakYLeft = 0.25F * startYLeft + 0.5F * controlYLeft + 0.25F *
endYLeft;
95.
96.     /*
97.      * 限制右侧曲线起点
```

```
98.      */
99.      if (startYLeft <= 0) {
100.          startYLeft = 0;
101.      }
102.
103.      /*
104.         * 限制底部左侧曲线起点
105.         */
106.      if (startXBtm <= 0) {
107.          startXBtm = 0;
108.      }
109.
110.      /*
111.         * 根据底部左侧限制点重新计算贝塞尔曲线顶点坐标
112.         */
113.      float partOfShortLength = CURVATURE * sizeShort;
114.      if (btmX >= -mValueAdded && btmX <= partOfShortLength - mValueAdded) {

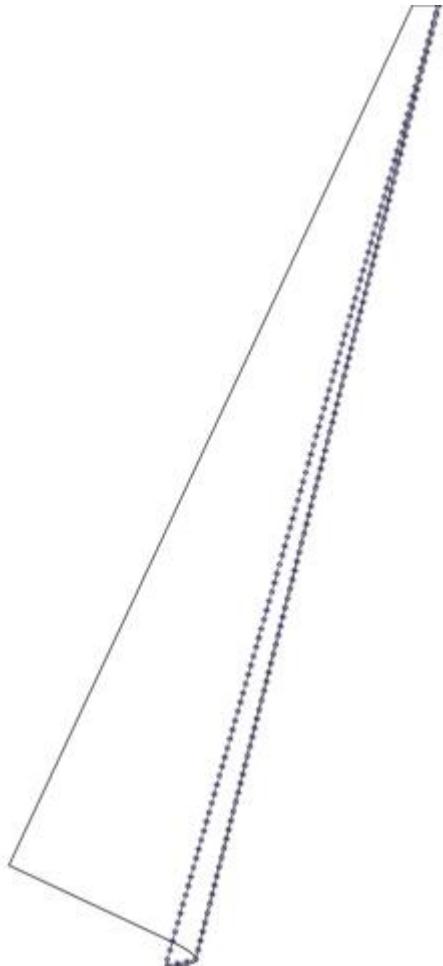
115.          float f = btmX / partOfShortLength;
116.          float t = 0.5F * f;
117.
118.          float bezierPeakTemp = 1 - t;
119.          float bezierPeakTemp1 = bezierPeakTemp * bezierPeakTemp;
120.          float bezierPeakTemp2 = 2 * t * bezierPeakTemp;
121.          float bezierPeakTemp3 = t * t;
122.
123.          bezierPeakXBtm = bezierPeakTemp1 * startXBtm + bezierPeakTemp2 * controlXBtm + bezierPeakTemp3 * endXBtm;
124.          bezierPeakYBtm = bezierPeakTemp1 * startYBtm + bezierPeakTemp2 * controlYBtm + bezierPeakTemp3 * endYBtm;
125.      }
126.
127.      /*
128.         * 根据右侧限制点重新计算贝塞尔曲线顶点坐标
129.         */
130.      float partOfLongLength = CURVATURE * sizeLong;
131.      if (leftY >= -mValueAdded && leftY <= partOfLongLength - mValueAdded) {

132.          float f = leftY / partOfLongLength;
133.          float t = 0.5F * f;
134.
135.          float bezierPeakTemp = 1 - t;
136.          float bezierPeakTemp1 = bezierPeakTemp * bezierPeakTemp;
137.          float bezierPeakTemp2 = 2 * t * bezierPeakTemp;
```

```
138.         float bezierPeakTemp3 = t * t;
139.
140.         bezierPeakXLeft = bezierPeakTemp1 * startXLeft + bezierPeakTemp2 *
controlXLeft + bezierPeakTemp3 * endXLeft;
141.         bezierPeakYLeft = bezierPeakTemp1 * startYLeft + bezierPeakTemp2 *
controlYLeft + bezierPeakTemp3 * endYLeft;
142.     }
143.
144.     /*
145.      * 替补区域 Path
146.     */
147.     mPathTrap.moveTo(startXBtm, startYBtm);
148.     mPathTrap.lineTo(startXLeft, startYLeft);
149.     mPathTrap.lineTo(bezierPeakXLeft, bezierPeakYLeft);
150.     mPathTrap.lineTo(bezierPeakXBtm, bezierPeakYBtm);
151.     mPathTrap.close();
152.
153.     /*
154.      * 生成带曲线的三角形路径
155.     */
156.     mPath.moveTo(startXBtm, startYBtm);
157.     mPath.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
158.     mPath.lineTo(mPointX, mPointY);
159.     mPath.lineTo(endXLeft, endYLeft);
160.     mPath.quadTo(controlXLeft, controlYLeft, startXLeft, startYLeft);
161.
162.     /*
163.      * 生成底部月半圆的 Path
164.     */
165.     mPathSemicircleBtm.moveTo(startXBtm, startYBtm);
166.     mPathSemicircleBtm.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
167.
168.     mPathSemicircleBtm.close();
169.
170.     /*
171.      * 生成右侧月半圆的 Path
172.     */
173.     mPathSemicircleLeft.moveTo(endXLeft, endYLeft);
174.     mPathSemicircleLeft.quadTo(controlXLeft, controlYLeft, startXLeft, startYLeft);
175.
176.     mPathSemicircleLeft.close();
177.
178.     /*
179.      * 生成包含折叠和下一页的路径
180.     */
```

```
178.     */
179.     //暂时没用省略掉
180.
181.     /*
182.      * 计算底部和右侧两月半圆区域
183.      */
184.     Region regionSemicircleBtm = computeRegion(mPathSemicircleBtm);
185.     Region regionSemicircleLeft = computeRegion(mPathSemicircleLeft);
186.
187.     // 合并两月半圆区域
188.     mRegionSemicircle.op(regionSemicircleBtm, regionSemicircleLeft, Region.
Op.UNION);
189. }
190.
191. // 根据 Path 生成的折叠区域
192. Region regioFlod = computeRegion(mPath);
193.
194. // 替补区域
195. Region regionTrap = computeRegion(mPathTrap);
196.
197. // 令折叠区域与替补区域相加
198. regioFlod.op(regionTrap, Region.Op.UNION);
199.
200. // 从相加后的区域中剔除掉月半圆的区域获得最终折叠区域
201. regioFlod.op(mRegionSemicircle, Region.Op.DIFFERENCE);
202.
203. /*
204.  * 根据裁剪区域填充画布
205. */
206. canvas.clipRegion(regioFlod);
207. canvas.drawColor(Color.RED);
```

200 行的代码我们就做了一件事就是正确计算 Path，同样我们还是按照之前的分了两种情况来计算，第一种情况 `sizeLong > mViewHeight` 时，我们先计算替补的这块区域：

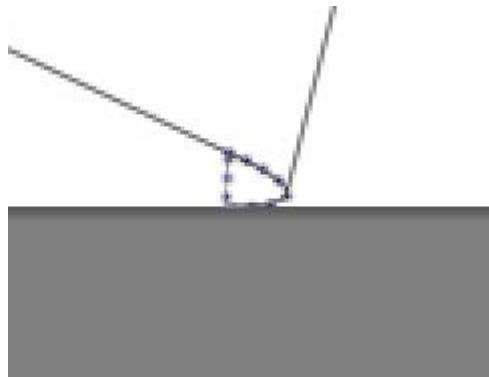


如上代码 46-49 行

[java] view plaincopyprint?

```
1. /*
2.  * 替补区域 Path
3. */
4. mPathTrap.moveTo(startXBtm, startYBtm);
5. mPathTrap.lineTo(topX2, 0);
6. mPathTrap.lineTo(bezierPeakXBtm, bezierPeakYBtm);
7. mPathTrap.close();
```

然后计算底部的月半圆 Path:



对应代码 54-56 行

[java] [view](#) [plain](#) [copy](#) [print](#)?

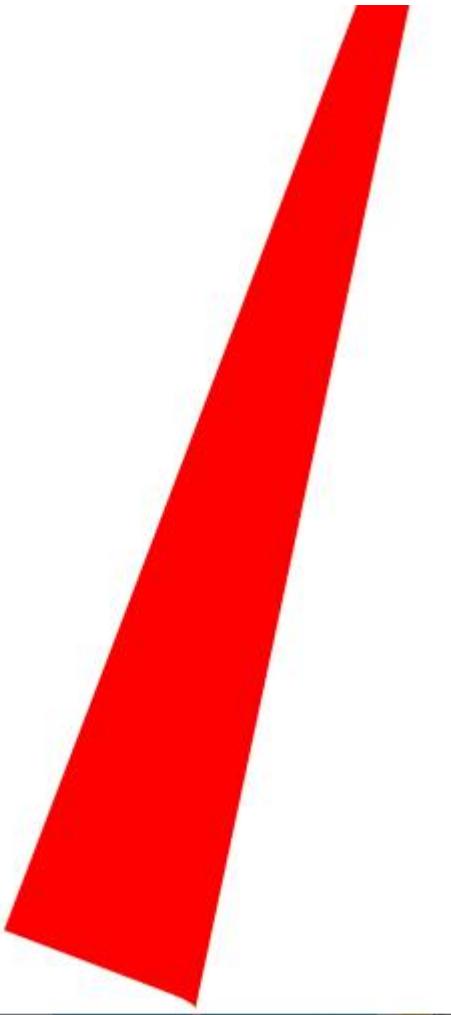
```
1. /*
2.  * 底部月半圆 Path
3. */
4. mPathSemicircleBtm.moveTo(startXBtm, startYBtm);
5. mPathSemicircleBtm.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);
6. mPathSemicircleBtm.close();
```

将当前折叠区域和替补区域相加再减去月半圆 Path 区域我们就可以得到正确的折叠区域，  
对应代码 64 行和 192-201 行：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 计算月半圆区域
2. mRegionSemicircle = computeRegion(mPathSemicircleBtm);
3.
4. // .....中间省略巨量代码.....
5.
6. // 根据 Path 生成的折叠区域
7. Region regioFlod = computeRegion(mPath);
8.
9. // 替补区域
10. Region regionTrap = computeRegion(mPathTrap);
11.
12. // 令折叠区域与替补区域相加
13. regioFlod.op(regionTrap, Region.Op.UNION);
14.
15. // 从相加后的区域中剔除掉月半圆的区域获得最终折叠区域
16. regioFlod.op(mRegionSemicircle, Region.Op.DIFFERENCE);
```

该情况下我们的折叠区域是酱紫的：



两种情况则稍微复杂些，除了要计算底部，我们还要计算右侧的月半圆 Path 区域，代码 165-174：

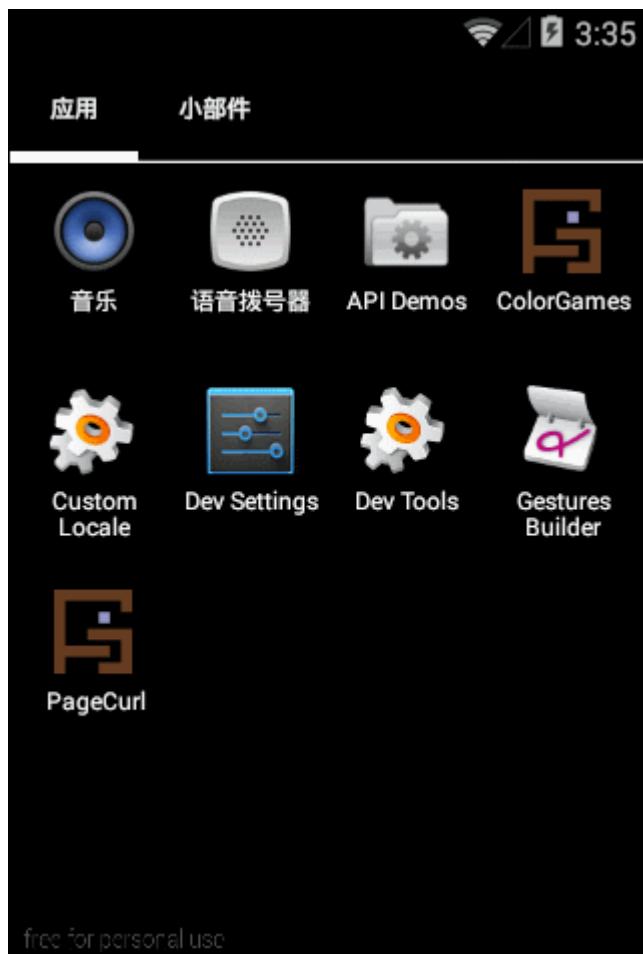
```
[java] view plain copy print?  
1. /*  
2.  * 生成底部月半圆的 Path  
3. */  
4. mPathSemicircleBtm.moveTo(startXBtm, startYBtm);  
5. mPathSemicircleBtm.quadTo(controlXBtm, controlYBtm, endXBtm, endYBtm);  
6. mPathSemicircleBtm.close();  
7.  
8. /*  
9.  * 生成右侧月半圆的 Path  
10. */  
11. mPathSemicircleLeft.moveTo(endXLeft, endYLeft);  
12. mPathSemicircleLeft.quadTo(controlXLeft, controlYLeft, startXLeft, startYLeft);  
13. mPathSemicircleLeft.close();  
14. 替补区域的计算, 147-151:
```

```
15. /*
16. * 替补区域 Path
17. */
18. mPathTrap.moveTo(startXBtm, startYBtm);
19. mPathTrap.lineTo(startXLeft, startYLeft);
20. mPathTrap.lineTo(bezierPeakXLeft, bezierPeakYLeft);
21. mPathTrap.lineTo(bezierPeakXBtm, bezierPeakYBtm);
22. mPathTrap.close();
23. 区域的转换, 184-188:
24. /*
25. * 计算底部和右侧两月半圆区域
26. */
27. Region regionSemicircleBtm = computeRegion(mPathSemicircleBtm);
28. Region regionSemicircleLeft = computeRegion(mPathSemicircleLeft);
29.
30. // 合并两月半圆区域
31. mRegionSemicircle.op(regionSemicircleBtm, regionSemicircleLeft, Region.Op.UNION);
```

最终的计算跟上面第一种情况一样，效果如下：



结合两种情况，我们可以得到下面的效果：

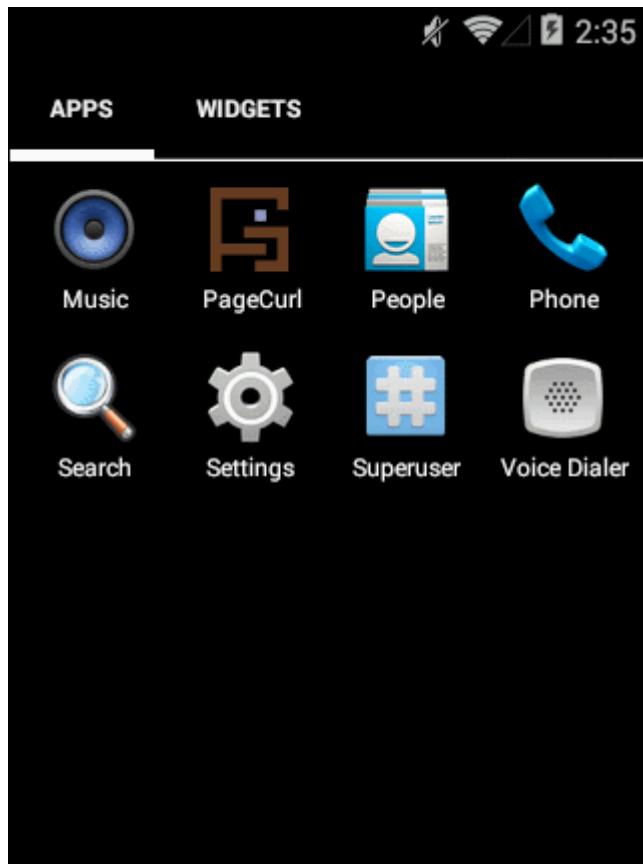


然后，我们需要计算“下一页”的区域，同样，根据上一节我们的讲解，我们先获取折叠区域和下一页区域之和再减去折叠区域就可以得到下一页的区域：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. mRegionNext = computeRegion(mPathFoldAndNext);  
2. mRegionNext.op(mRegionFold, Region.Op.DIFFERENCE);
```

绘制效果如下：



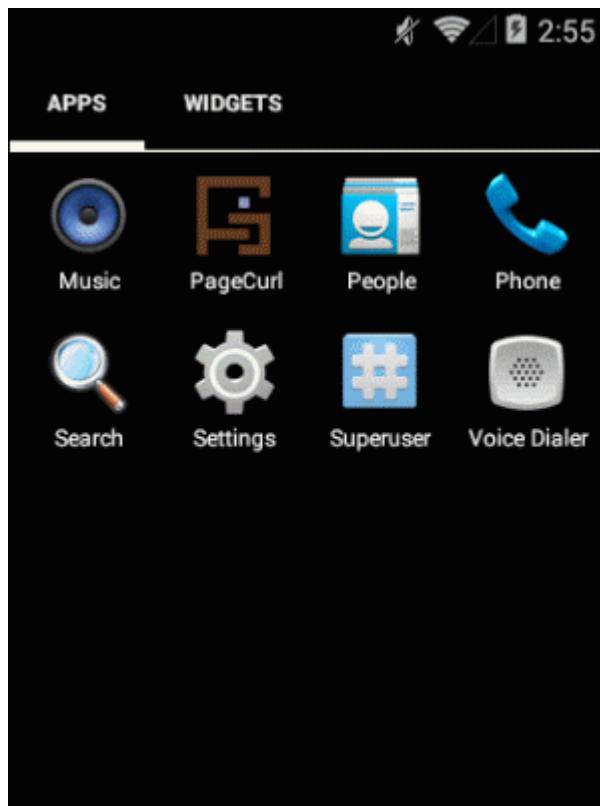
最后，我们结合上两节，注入数据：

```
[java] view plaincopyprint?  
1.  /**
2.   * 绘制位图数据
3.   *
4.   * @param canvas
5.   *          画布对象
6.   */
7. private void drawBitmaps(Canvas canvas) {
8.     // 绘制位图前重置 isLastPage 为 false
9.     isLastPage = false;
10.
11.    // 限制 pageIndex 的值范围
12.    mPageIndex = mPageIndex < 0 ? 0 : mPageIndex;
13.    mPageIndex = mPageIndex > mBitmaps.size() ? mBitmaps.size() : mPageIndex
14.    ;
15.    // 计算数据起始位置
16.    int start = mBitmaps.size() - 2 - mPageIndex;
17.    int end = mBitmaps.size() - mPageIndex;
18.
19.    /*
```

```
20.     * 如果数据起点位置小于 0 则表示当前已经到了最后一张图片
21.     */
22.     if (start < 0) {
23.         // 此时设置 isLastPage 为 true
24.         isLastPage = true;
25.
26.         // 并显示提示信息
27.         showToast("This is fucking lastest page");
28.
29.         // 强制重置起始位置
30.         start = 0;
31.         end = 1;
32.     }
33.
34.     /*
35.      * 计算当前页的区域
36.      */
37.     canvas.save();
38.     canvas.clipRegion(mRegionCurrent);
39.     canvas.drawBitmap(mBitmaps.get(end - 1), 0, 0, null);
40.     canvas.restore();
41.
42.     /*
43.      * 计算折叠页的区域
44.      */
45.     canvas.save();
46.     canvas.clipRegion(mRegionFold);
47.
48.     canvas.translate(mPointX, mPointY);
49.
50.     /*
51.      * 根据长短边标识计算折叠区域图像
52.      */
53.     if (mRatio == Ratio.SHORT) {
54.         canvas.rotate(90 - mDegrees);
55.         canvas.translate(0, -mViewHeight);
56.         canvas.scale(-1, 1);
57.         canvas.translate(-mViewWidth, 0);
58.     } else {
59.         canvas.rotate(-(90 - mDegrees));
60.         canvas.translate(-mViewWidth, 0);
61.         canvas.scale(1, -1);
62.         canvas.translate(0, -mViewHeight);
63.     }
```

```
64.  
65.     canvas.drawBitmap(mBitmaps.get(end - 1), 0, 0, null);  
66.     canvas.restore();  
67.  
68.     /*  
69.      * 计算下一页的区域  
70.      */  
71.     canvas.save();  
72.     canvas.clipRegion(mRegionNext);  
73.     canvas.drawBitmap(mBitmaps.get(start), 0, 0, null);  
74.     canvas.restore();  
75. }
```

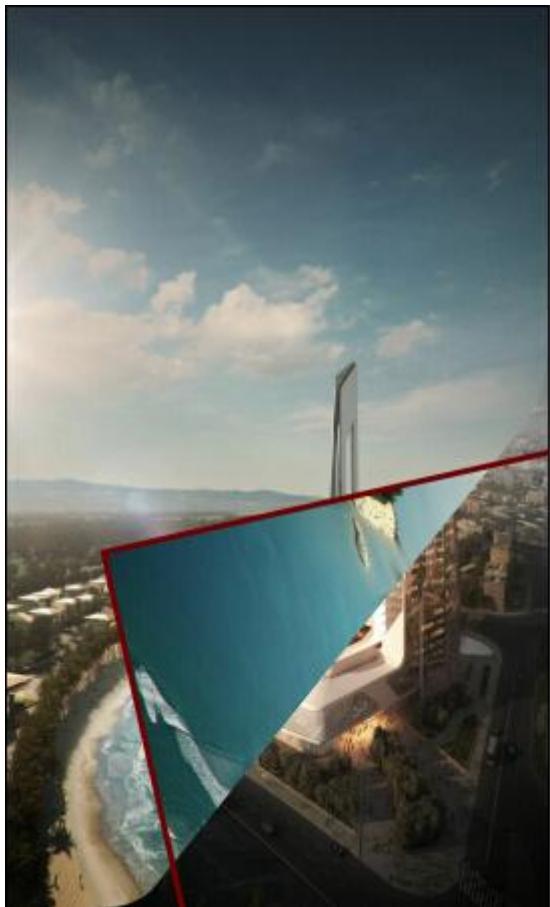
最终效果如下：



该部分的代码就不贴出了，大部分跟上一节相同，因为过两天要去旅游时间略紧这节略讲得粗糙，不过也没什么太大的改动，如果大家有不懂的地方可以留言或群里@哥，下一节我们将尝试实现翻页时图像扭曲的效果。

## 10. 自定义控件其实很简单(10)

上一节我们实现了翻页的曲线效果，但是效果有点小瑕疵不知道大家发现没有：



如图，我们发现折叠区域怪怪的，并没有实现我们之前的“弯曲”效果，为什么呢？是计算错了么？其实不是的，我们之前测试的时候使用的将 `canvas` 填色，但是这里我们用到的是一张位图，虽然我们的 `Path` 是曲线、`Region` 有曲线区域，但是我们的 `Bitmap` 是个规规矩矩的矩形啊，怎么弯曲~怎么办呢？说起扭曲，我们首先想到的是 `drawBitmapMesh` 方法，它是我们现在了解的也是唯一的一个能对图像进行扭曲的 API，而使用 `drawBitmapMesh` 方法呢我们也可以有多种思路，最简单的就是最大化恒定细分值，将图像分割成一定的网格区域，然后判断离曲线起点和顶点最近的细分线获取该区域内的细分线交点按指定方向百分比递减移动起点和顶点的距离值即可，这种方法简单粗暴，但扭曲不是很精确，正确地说精确度取决于细分，细分也大越精确当然也越耗性能，而第二种方法呢是根据曲线的起点和顶点动态生成细分值，我们可以确保在起点和顶点处都有一条细分线，这样就可以很准确地计算扭曲范围，但是我们就需要动态地去不断计算细分值相当麻烦，用哪种呢？这里鉴于时间关系还是尝试用第一种去做，首先定义宽高的细分值：

```
[java] view plaincopyprint?
```

```
1. private static final int SUB_WIDTH = 19, SUB_HEIGHT = 19; // 细分值横竖各 19 个  
网格
```

19 个网格将控件分割为 20x20 的网格细分线条区域，尔后我们就不需要使用 `drawBitmap` 绘制折叠区域了而是改用 `drawBitmapMesh`。之前在讲 1/6 的时候有盆友多次小窗过我离屏

缓冲是个什么意思需要注意什么，这里我权当演示，在绘制扭曲图像的时候使用一个单独的 Bitmap 并将其装载进一个额外的 Canvas 中：

[java] view plaincopyprint?

```
1. private Canvas mCanvasFoldCache;// 执行绘制离屏缓冲的 Canvas  
2. private Bitmap mBitmapFoldCache;// 存储绘制离屏缓冲数据的 Bitmap
```

在构造方法中我们实例化 mCanvasFoldCache：

[java] view plaincopyprint?

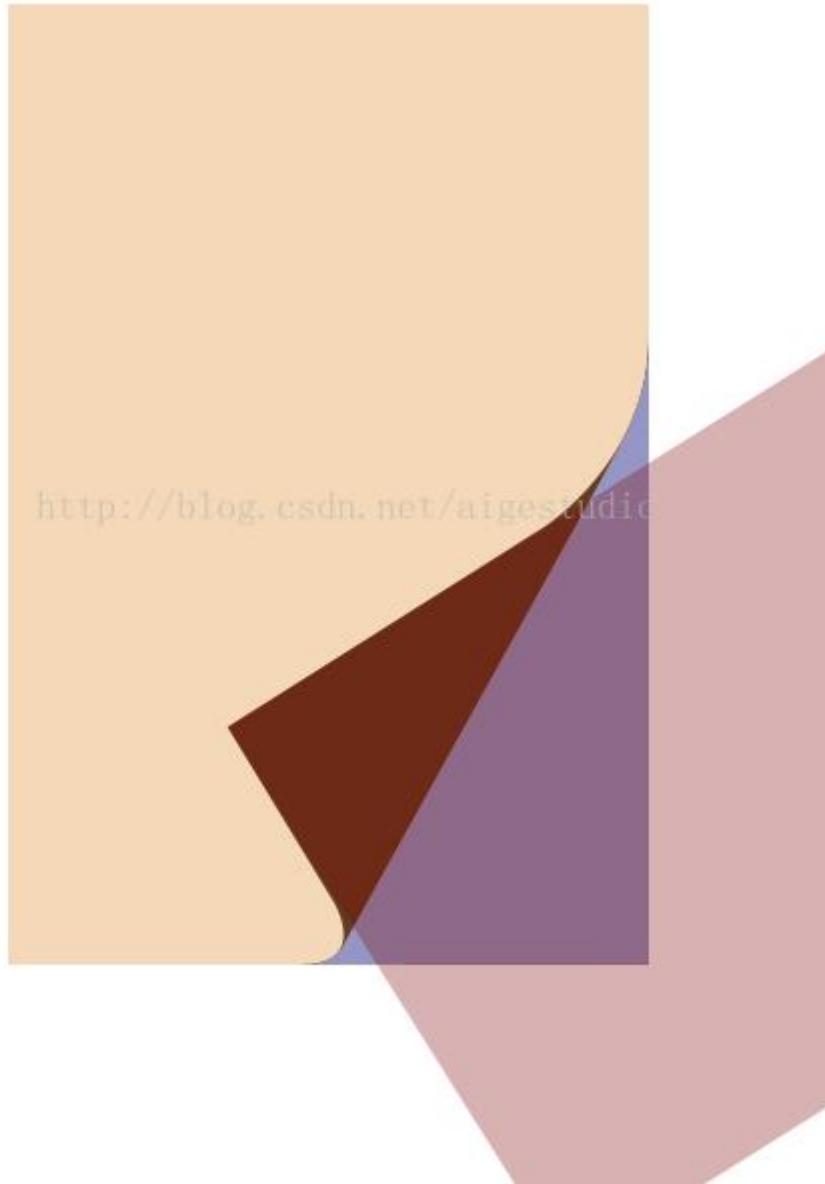
```
1. /*  
2.  * 实例化 Canvas  
3. */  
4. mCanvasFoldCache = new Canvas();
```

在 onSizeChanged 中我们生成 mBitmapFoldCache：

[java] view plaincopyprint?

```
1. /*  
2.  * 生成缓冲位图并注入 Canvas  
3. */  
4. mBitmapFoldCache = Bitmap.createBitmap(mViewWidth + 100, mViewHeight + 100,  
    Bitmap.Config.ARGB_8888);  
5. mCanvasFoldCache.setBitmap(mBitmapFoldCache);
```

这里+100 的目的是让 Bitmap 有多余的空间绘制扭曲的那部分图像，我们之前说过 Canvas 的大小实际取决于内部装载的 Bitmap，如果这里我们不+100，那么 mBitmapFoldCache 的大小就刚好和我们的控件一样大，但是我们实现扭曲的那一部分图像是超出该范围外的：



如上图透明红色的范围是我们 `mBitmapFoldCache` 的大小，但是底部和右侧的扭曲没有被包含进来，为了弥补这部分的损失我将 `mBitmapFoldCache` 的宽高各+100，当然你也可以计算出具体的值，这里只做演示。

而在绘制时，我们先将所有的数据绘制到 `mBitmapFoldCache` 上再将该 `Bitmap` 绘制到我们的 `canvas` 中：

```
[java] view plaincopyprint?
```

```
1. mCanvasFoldCache.drawBitmapMesh(mBitmaps.get(end - 1), SUB_WIDTH, SUB_HEIGHT  
, mVerts, 0, null, 0, null);
```

```
2. canvas.drawBitmap(mBitmapFoldCache, 0, 0, null);
```

这里要注意的是，我们的 `mBitmapFoldCache` 在 `onSizeChanged` 方法中生成，每次我们绘制的时候都不再重新生成，也就是说，每次绘制其实都是叠加在上一次的绘制数据上，那么这就会给我们带来一个问题，虽然显示结果有可能不会出错但是每次绘制都要不断计算前面的像素次数一多必定会大大影响性能，这时候我们考虑在绘制每一次结果前清空掉 `mBitmapFoldCache` 中的内容：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. mCanvasFoldCache.drawColor(Color.TRANSPARENT, PorterDuff.Mode.CLEAR);
2. mCanvasFoldCache.drawBitmapMesh(mBitmaps.get(end - 1), SUB_WIDTH, SUB_HEIGHT,
   , mVerts, 0, null, 0, null);
3. canvas.drawBitmap(mBitmapFoldCache, 0, 0, null);
```

题外话到此为止，实际上我们不需要缓冲绘制，直接使用 `drawBitmapMesh` 即可：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. canvas.drawBitmapMesh(mBitmaps.get(end - 1), SUB_WIDTH, SUB_HEIGHT, mVerts,
   0, null, 0, null);
```

而重点则是我们的这些扭曲点怎么生成，在构造方法中我们实例化坐标数组：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 实例化数组并初始化默认数组数据
2. mVerts = new float[(SUB_WIDTH + 1) * (SUB_HEIGHT + 1) * 2];
```

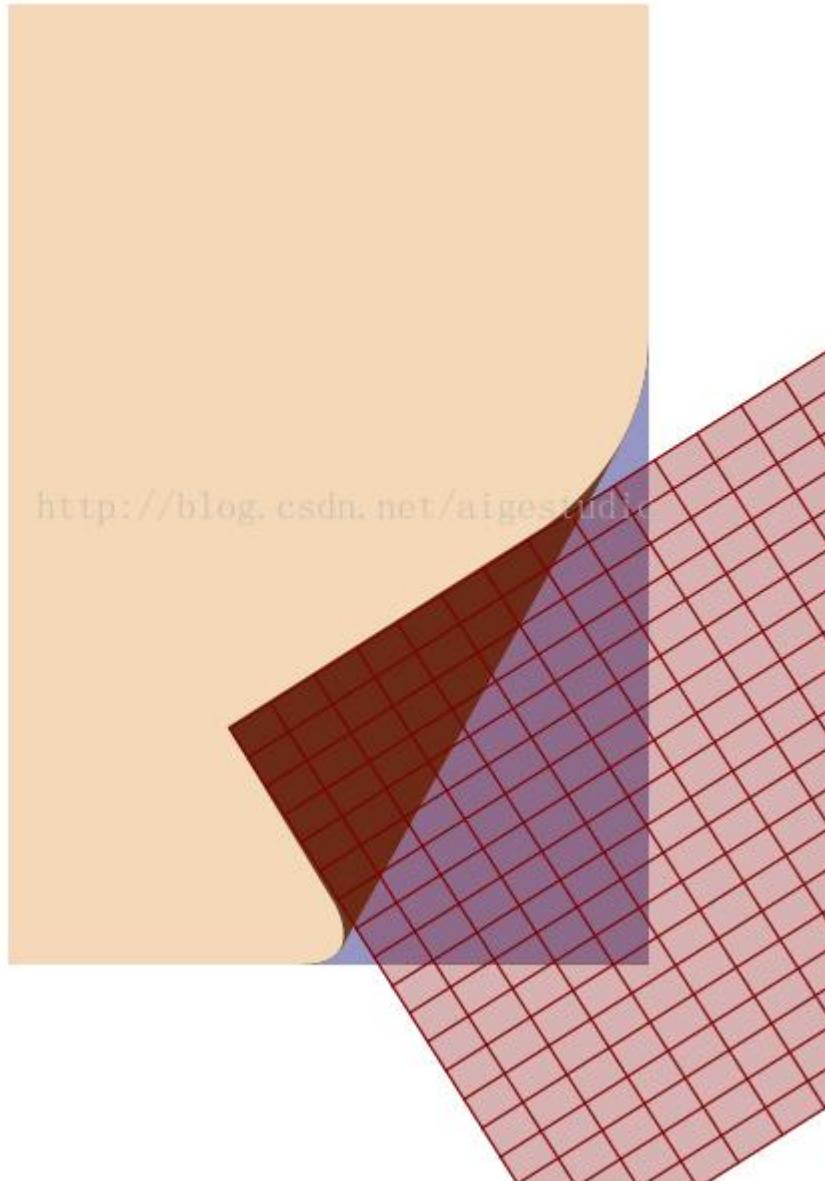
在计算了曲线各个点坐标之后我们生成扭曲坐标：

[java] [view](#) [plain](#) [copy](#) [print](#)?

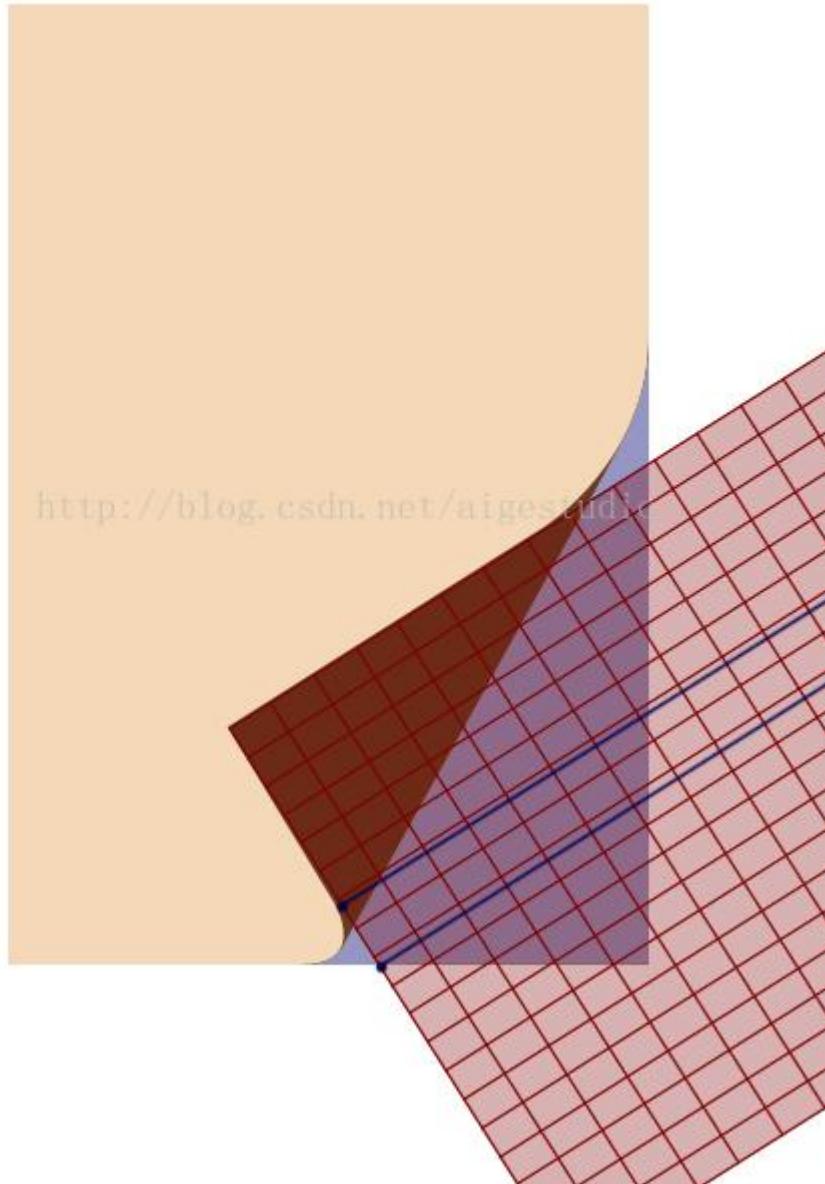
```
1. if (sizeLong > mViewHeight) {
2.     // 省略大量代码.....
3. } else {
4.     // 省略巨量代码.....
5.     /*
6.      * 生成折叠区域的扭曲坐标
7.      */
8.     int index = 0;
9.     for (int y = 0; y <= SUB_HEIGHT; y++) {
10.         float fy = mViewHeight * y / SUB_HEIGHT;
```

```
11.         for (int x = 0; x <= SUB_WIDTH; x++) {  
12.             float fx = mViewWidth * x / SUB_WIDTH;  
13.  
14.             mVerts[index * 2 + 0] = fx;  
15.             mVerts[index * 2 + 1] = fy;  
16.  
17.             index += 1;  
18.         }  
19.     }  
20. }
```

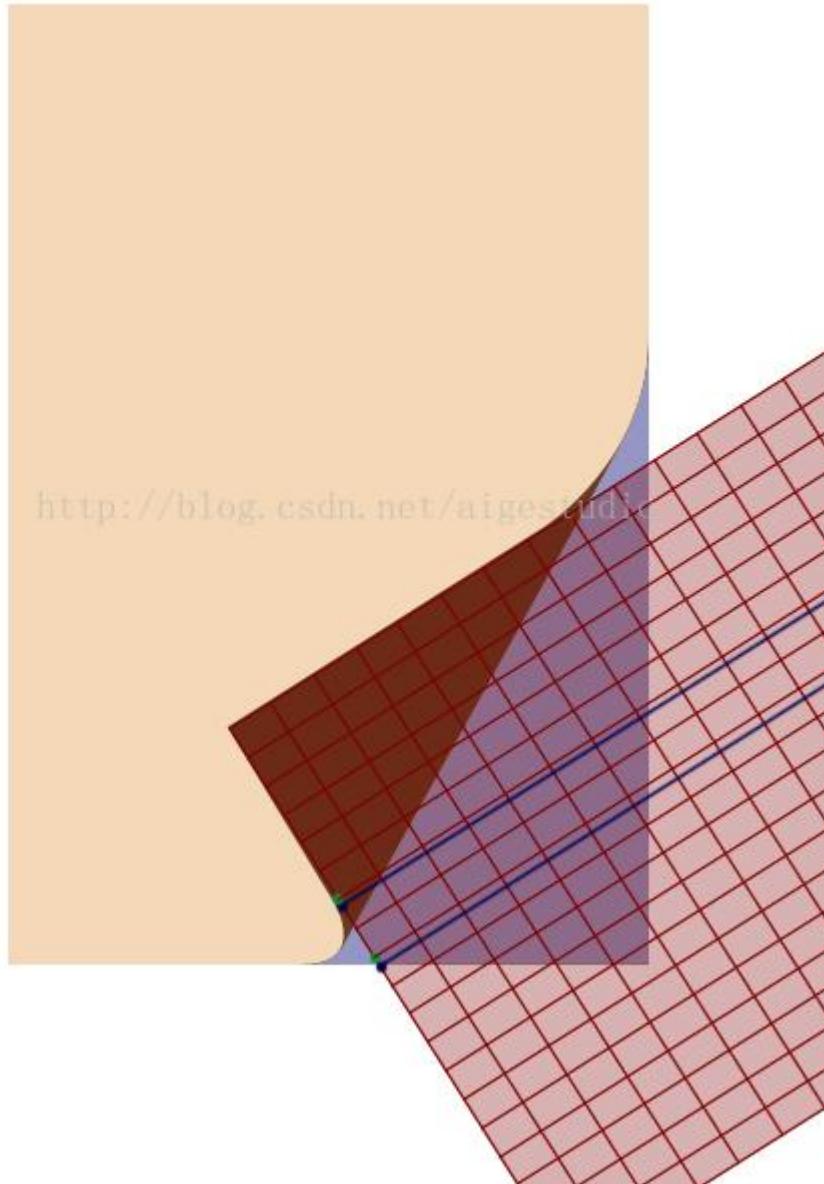
虽然上面我们生成了坐标数组，但是并没有扭曲图像，在进行下一步操作前我们先来分析一下如何进行扭曲呢，当我们在折叠区域以 `drawBitmapMesh` 的方式绘制 Bitmap 时这时候的图像实质上是被网格分割了的：



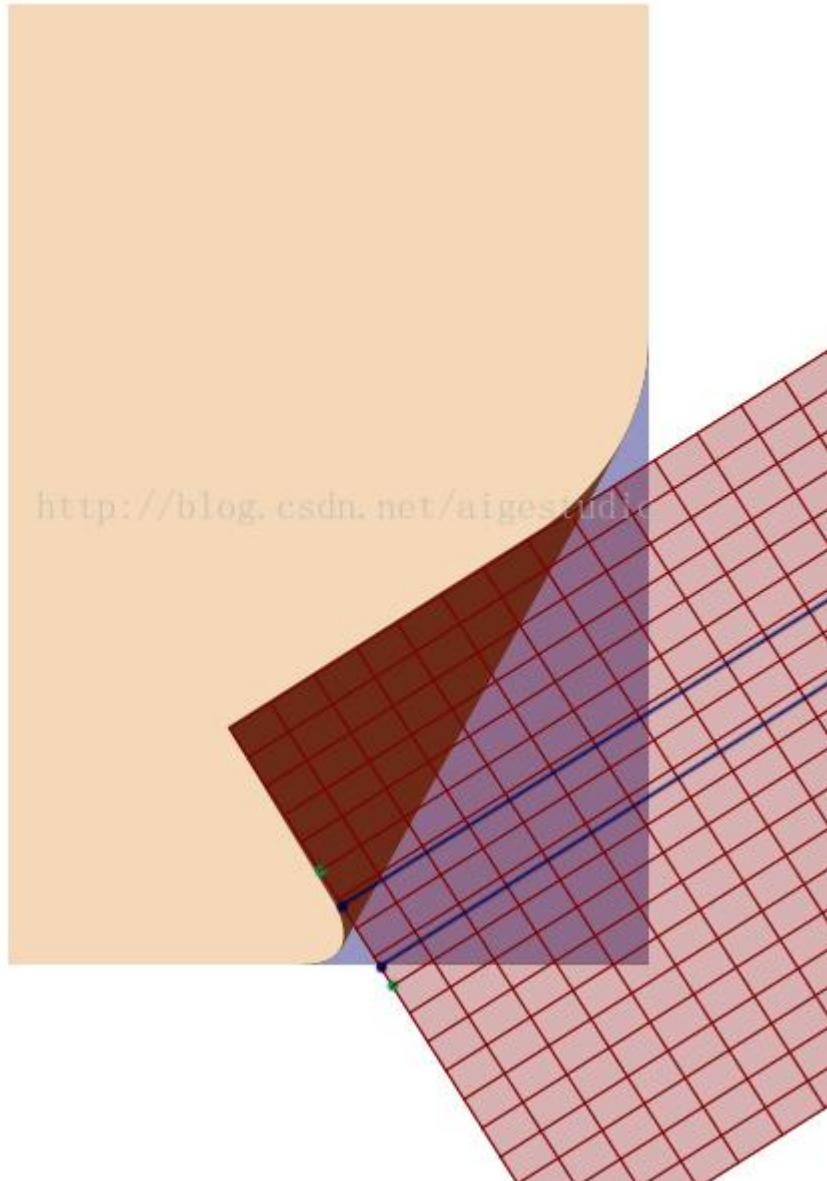
我们的方法其实很简单，只需要把从短边长度减短边长度乘以  $1/4$  的位置开始到短边长度位置的点按递增向下拽即可对吧：



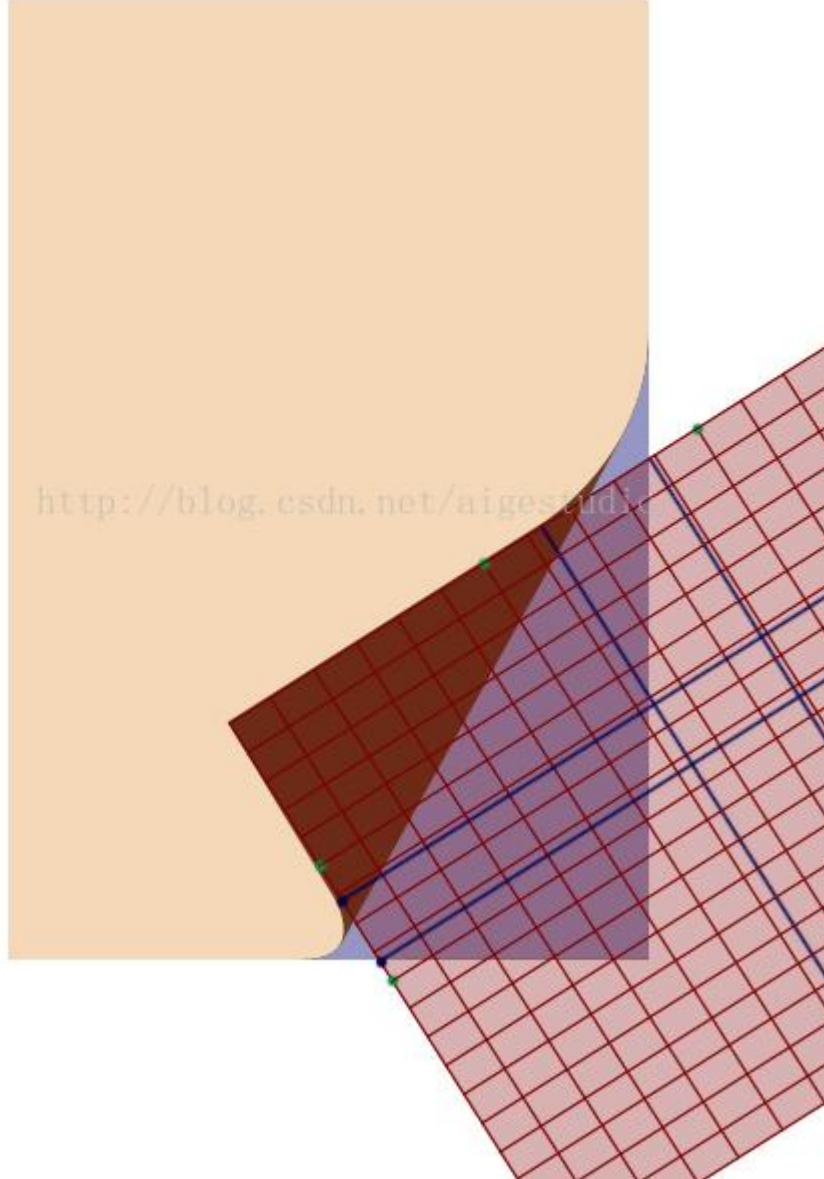
如上图所示的两个蓝点分别代表短边长度减短边长度乘以  $1/4$  的位置和短边长度位置，因为我们的网格是不变的，但是位置在不断改变，我们应当获取离当前位置最近的网格点，比如上图中的两个蓝点此时我们应该获取到网格中的对应位置是：



如图中绿色的蓝点，考虑到更好的容差值，我们令起点往后挪一个点而终点往前挪一个点，最终我们的取舍点如下：



同样，我们右侧的也一样：

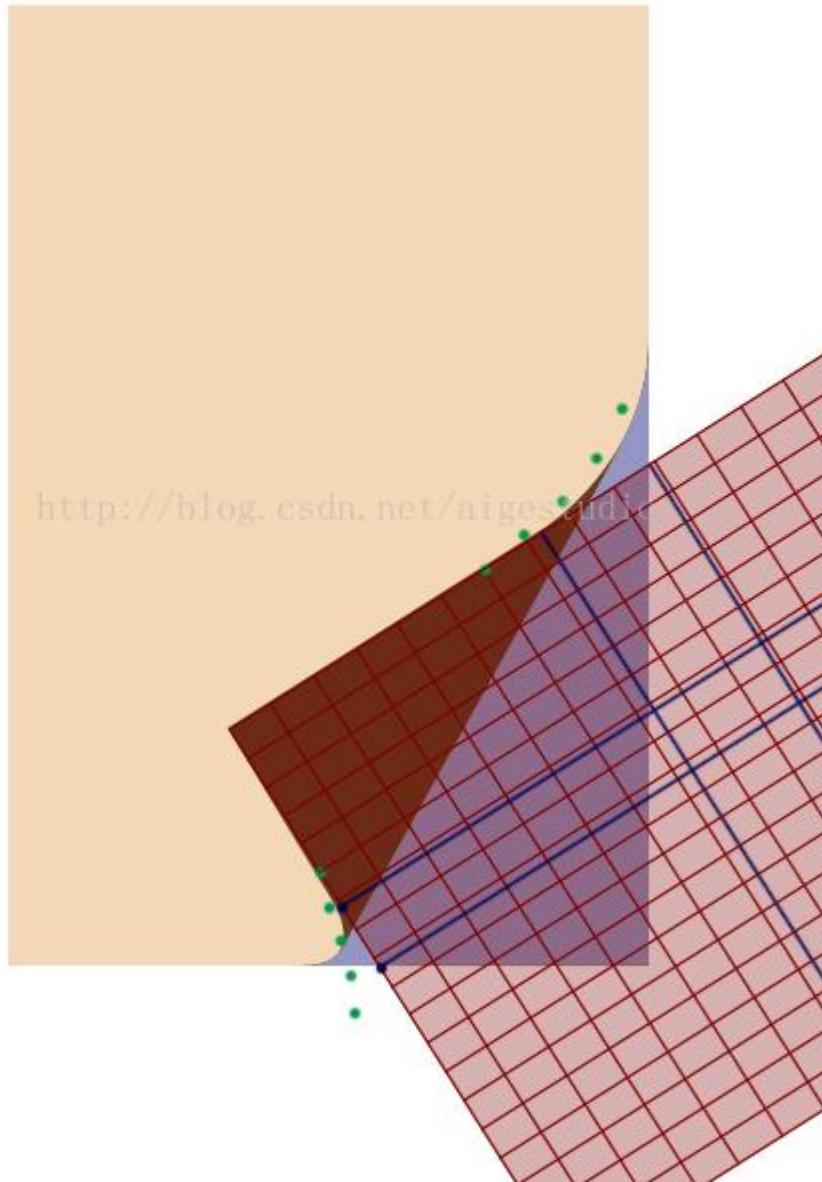


那在代码中的实现也很简单：

```
[java] view plaincopyprint?  
1. // 计算底部扭曲的起始细分下标  
2. mSubWidthStart = Math.round((btmX / mSubMinWidth)) - 1;  
3. mSubWidthEnd = Math.round(((btmX + CURVATURE * sizeShort) / mSubMinWidth)) +  
    1;  
4.  
5. // 计算右侧扭曲的起始细分下标  
6. mSubHeightStart = (int) (leftY / mSubMinHeight) - 1;
```

```
7. mSubHeightEnd = (int) (leftY + CURVATURE * sizeLong / mSubMinHeight) + 1;
```

我们只需要将 `mSubWidthStart` 到 `mSubWidthEnd` 之间的点往下“拽”，`mSubHeightStart` 到 `mSubHeightEnd` 的点往右“拽”即可实现初步的“扭曲”效果对吧，但是这个拽是有讲究的，首先，拽的距离是倍增的，如图：



每一个点的偏移值相对于上一个点来说是倍增的，倍增多少呢？是基于最大的偏移值来说的，这里为了简化一定的问题，我就不去计算了，而是给定一个固定的起始值和倍增率：

```
[java] view plaincopyprint?
```

```
1. // 长边偏移
2. float offsetLong = CURVATURE / 2F * sizeLong;
3.
4. // 长边偏移倍增
5. float mulOffsetLong = 1.0F;
6.
7. // 短边偏移
8. float offsetShort = CURVATURE / 2F * sizeShort;
9.
10. // 短边偏移倍增
11. float mulOffsetShort = 1.0F;
```

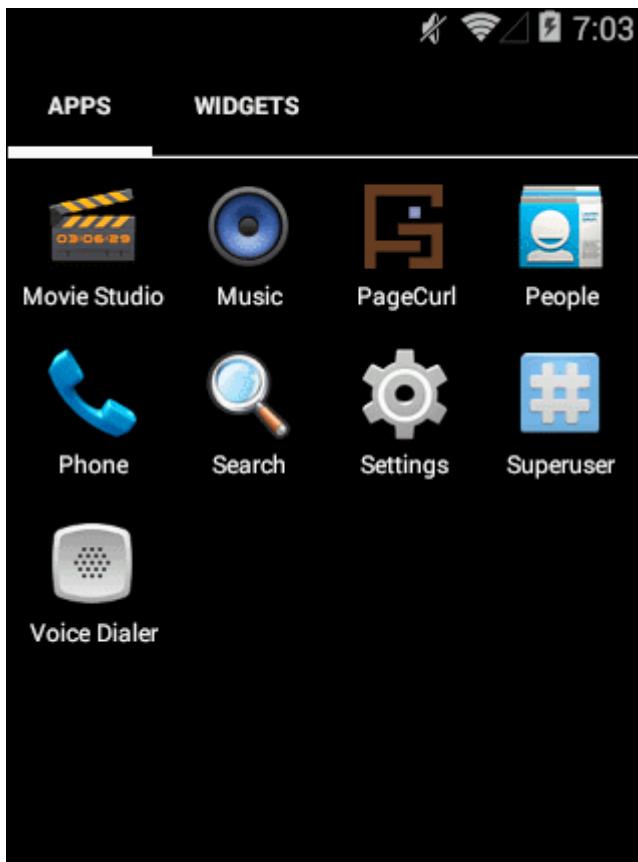
这时候我们可以考虑开始计算扭曲坐标：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 计算底部扭曲的起始细分下标
2. mSubWidthStart = Math.round((btmX / mSubMinWidth)) - 1;
3. mSubWidthEnd = Math.round(((btmX + CURVATURE * sizeShort) / mSubMinWidth)) +
   1;
4.
5. // 计算右侧扭曲的起始细分下标
6. mSubHeightStart = (int) (leftY / mSubMinHeight) - 1;
7. mSubHeightEnd = (int) (leftY + CURVATURE * sizeLong / mSubMinHeight) + 1;
8.
9. /*
10. * 生成折叠区域的扭曲坐标
11. */
12. int index = 0;
13.
14. // 长边偏移
15. float offsetLong = CURVATURE / 2F * sizeLong;
16.
17. // 长边偏移倍增
18. float mulOffsetLong = 1.0F;
19.
20. // 短边偏移
21. float offsetShort = CURVATURE / 2F * sizeShort;
22.
23. // 短边偏移倍增
24. float mulOffsetShort = 1.0F;
25. for (int y = 0; y <= SUB_HEIGHT; y++) {
26.     float fy = mViewHeight * y / SUB_HEIGHT;
27.     for (int x = 0; x <= SUB_WIDTH; x++) {
```

```
28.         float fx = mViewWidth * x / SUB_WIDTH;
29.
30.         /*
31.          * 右侧扭曲
32.          */
33.         if (x == SUB_WIDTH) {
34.             if (y >= mSubHeightStart && y <= mSubHeightEnd) {
35.                 fx = mViewWidth * x / SUB_WIDTH + offsetLong * mulOffsetLong
36. ;
37.                 mulOffsetLong = mulOffsetLong / 1.5F;
38.             }
39.         }
40.
41.         /*
42.          * 底部扭曲
43.          */
44.         if (y == SUB_HEIGHT) {
45.             if (x >= mSubWidthStart && x <= mSubWidthEnd) {
46.                 fy = mViewHeight * y / SUB_HEIGHT + offsetShort * mulOffsetS
hort;
47.                 mulOffsetShort = mulOffsetShort / 1.5F;
48.             }
49.         }
50.         mVerts[index * 2 + 0] = fx;
51.         mVerts[index * 2 + 1] = fy;
52.
53.         index += 1;
54.     }
55. }
```

效果如下：



上面的图因为上传大小的限制我压缩过可能大家看不清楚，如果大家 DL 我想项目运行可以看到在我们翻动的过程中扭曲的部分会有一点小跳动，原因很简单，我们的扭曲只针对了底部最后的一行点  $y == \text{SUB\_HEIGHT}$  和右侧最右的一列点  $x == \text{SUB\_WIDTH}$ ，而事实上扭曲是个拉扯联动的效果，扭曲不仅仅会影响最后一行/列同时也会影倒数第二、三、四行等，只不过这个影响效力是递减的，这部分就留给大家自己去做了，原理我讲的很清楚了。这一节到此为止，下一节我们将完善最终效果结束本例所有的 Study~

## 11. 自定义控件其实很简单(11)

要在数量上统计中国菜的品种，在地域上毫无争议地划分菜系，在今天，是一件几乎不可能



完成的事……Cut…… ……抱歉……忘吃药了，再来一遍。如果非要对自定义控件的流程进行一个简单的划分，我会尝试将其分为三大部分：控件的绘制、控件的测量和控件的交互行为。前面我们用了六节的篇幅和一个翻页的例子来对控件的绘制有了一个全新的认识但是我们所做出的所有例子都是不完美的，为什么这么说呢，还是先来看个 sample：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/12
5. *
6. */
7. public class ImgView extends View {
8.     private Bitmap mBitmap;// 位图对象
9.
10.    public ImgView(Context context, AttributeSet attrs) {
11.        super(context, attrs);
12.    }
13.
14.    @Override
15.    protected void onDraw(Canvas canvas) {
16.        // 绘制位图
17.        canvas.drawBitmap(mBitmap, 0, 0, null);
18.    }
19.
20. /**
21. * 设置位图
22. *
23. * @param bitmap
24. *          位图对象
25. */
26. public void setBitmap(Bitmap bitmap) {
27.     this.mBitmap = bitmap;
28. }
29. }
```

这个例子呢非常简单，我们用它来模拟类似 `ImageView` 的效果显示一张图片，在 `MainActivity` 中我们获取该控件并为其设置 `Bitmap`:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. /**
2. * 主界面
3. *
4. * @author Aige {@link http://blog.csdn.net/aigestudio}
5. * @since 2014/11/17
6. */
7. public class MainActivity extends Activity {
8.     private ImgView mImgView;
9. }
```

```
10.     @Override
11.     public void onCreate(Bundle savedInstanceState) {
12.         super.onCreate(savedInstanceState);
13.         setContentView(R.layout.activity_main);
14.
15.         mImgView = (ImageView) findViewById(R.id.main_pv);
16.         Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.lovestory);
17.         mImgView.setImageBitmap(bitmap);
18.     }
19. }
```

此时运行效果如下：



很简单对吧，可是上面的代码其实是有问题的，至于什么问题？我们待会再说，就看你通过前面我们的学习能不能发现了。这一节我们重点是控件的测量，大家不知道注意没有，这个系列文章的命名我用了“控件”而非“View”其实目的就是说明我们的控件不仅包括 View 也应该包含 ViewGroup，当然你也可以以官方的方式将其分为控件和布局，不过我更喜欢 View 和 ViewGroup，好了废话不说，我们先来看看 View 的测量方式，上面的代码中 MainActivity 对应的布局文件如下：

[html] view plain copy print?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:background="#FFFFFF"
```

```
5.     android:orientation="vertical" >
6.
7.     <com.aigestudio.customviewdemo.views.ImgView
8.         android:id="@+id/main_pv"
9.         android:layout_width="match_parent"
10.        android:layout_height="match_parent" />
11. </LinearLayout>
```

既然我们的自定义 View 也算一个控件那么我们也可以像平时做布局那样往我们的 LinearLayout 中添加各种各样的其他控件对吧：

[html] view plaincopyprint?

```
1.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      android:layout_width="match_parent"
3.      android:layout_height="match_parent"
4.      android:background="#FFFFFF"
5.      android:orientation="vertical" >
6.
7.      <com.aigestudio.customviewdemo.views.ImgView
8.          android:id="@+id/main_pv"
9.          android:layout_width="match_parent"
10.         android:layout_height="match_parent" />
11.
12.      <Button
13.          android:layout_width="wrap_content"
14.          android:layout_height="wrap_content"
15.          android:text="AigeStudio" />
16.
17.      <TextView
18.          android:layout_width="wrap_content"
19.          android:layout_height="wrap_content"
20.          android:text="AigeStudio" />
21.
22. </LinearLayout>
```

但是运行后你却发现我们的 Button 和 TextView 却没有显示在屏幕上，这时你可能会说那当然咯，因为我们的 ImgView layout\_width 和 layout\_height 均为 match\_parent，可是即便你将其改成 wrap\_content：

[html] view plaincopyprint?

```
1.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      android:layout_width="match_parent"
```

```
3.     android:layout_height="match_parent"
4.     android:background="#FFFFFF"
5.     android:orientation="vertical" >
6.
7.     <com.aigestudio.customviewdemo.views.ImgView
8.         android:id="@+id/main_pv"
9.         android:layout_width="wrap_content"
10.        android:layout_height="wrap_content" />
11.
12.        <!-- .....省略一些代码-->
13. </LinearLayout>
```

结果也一样，这时你肯定很困惑，不解的主要原因是没有搞懂 View 的测量机制，在前面的几节中我们或多或少有提到控件的测量，也曾经说过 Android 提供给我们能够操纵控件测量的方法是 `onMeasure`:

[java] view plain copy print?

```
1. @Override
2. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.     super.onMeasure(widthMeasureSpec, heightMeasureSpec);
4. }
```

默认情况下 `onMeasure` 方法中只是简单地将签名列表中的两个 `int` 型参数回传给父类的 `onMeasure` 方法，然后由父类的方法去计算出最终的测量值。但是，这里有个问题非常重要，就是 `onMeasure` 签名列表中的这两个参数是从何而来，这里可以告诉大家的是，这两个参数是由 `view` 的父容器，代码中也就是我们的 `LinearLayout` 传递进来的，很多初学 Android 的朋友会将位于 `xml` 布局文件顶端的控件称之为根布局，比如这里我们的 `LinearLayout`，而事实上在 Android 的 GUI 框架中，这个 `LinearLayout` 还称不上根布局，我们知道一个 `Activity` 可以对应一个 `View`（也可以是 `ViewGroup`），很多情况下我们会通过 `Activity` 的 `setContentView` 方法去设置我们的 `View`:

[java] view plain copy print?

```
1. @Override
2. public void onCreate(Bundle savedInstanceState) {
3.     super.onCreate(savedInstanceState);
4.     setContentView(R.layout.activity_main);
5. }
```

`setContentView` 在 `Activity` 内的实现也非常简单，就是调用 `getWindow` 方法获取一个 `Window` 类型的对象并调用其 `setContentView` 方法:

```
[java] view plaincopyprint?
```

```
1. public void setContentView(int layoutResID) {  
2.     getWindow().setContentView(layoutResID);  
3.     initActionBar();  
4. }
```

而这个 Window 对象

```
[java] view plaincopyprint?
```

```
1. public Window getWindow() {  
2.     return mWindow;  
3. }
```

其本质也就是一个 PhoneWindow，在 Activity 的 attach 方法中通过 makeNewWindow 生成：

```
[java] view plaincopyprint?
```

```
1. final void attach(Context context, ActivityThread aThread,  
2. // 此处省去一些代码.....  
3.  
4.     mWindow = PolicyManager.makeNewWindow(this);  
5.     mWindow.setCallback(this);  
6.     mWindow.getLayoutInflater().setPrivateFactory(this);  
7.     if (info.softInputMode != WindowManager.LayoutParams.SOFT_INPUT_STATE_UNSPECIFIED) {  
8.         mWindow.setSoftInputMode(info.softInputMode);  
9.     }  
10.    if (info.uiOptions != 0) {  
11.        mWindow.setUiOptions(info.uiOptions);  
12.    }  
13.    // 此处省去巨量代码.....  
14. }  
15. }
```

在 PolicyManager 中通过反射的方式获取 com.android.internal.policy.impl.Policy 的一个实例：

```
[java] view plaincopyprint?
```

```
1. public final class PolicyManager {  
2.     private static final String POLICY_IMPL_CLASS_NAME =
```

```
3.         "com.android.internal.policy.impl.Policy";
4.
5.     private static final IPolicy sPolicy;
6.
7.     static {
8.         try {
9.             Class policyClass = Class.forName(POLICY_IMPL_CLASS_NAME);
10.            sPolicy = (IPolicy)policyClass.newInstance();
11.        } catch (ClassNotFoundException ex) {
12.            throw new RuntimeException(
13.                POLICY_IMPL_CLASS_NAME + " could not be loaded", ex);
14.        } catch (InstantiationException ex) {
15.            throw new RuntimeException(
16.                POLICY_IMPL_CLASS_NAME + " could not be instantiated", e
17.            x);
18.        }
19.    }
20. }
21.
22.
23. // 省去构造方法.....
24.
25. public static Window makeNewWindow(Context context) {
26.     return sPolicy.makeNewWindow(context);
27. }
28.
29. // 省去无关代码.....
30. }
```

并通过其内部的 makeNewWindow 实现返回一个 PhoneWindow 对象：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public Window makeNewWindow(Context context) {
2.     return new PhoneWindow(context);
3. }
```

PhoneWindow 是 Window 的一个子类，其对 Window 中定义的大量抽象方法作了具体的实现，比如我们的 setContentView 方法在 Window 中仅做了一个抽象方法定义：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public abstract class Window {  
2.     // 省去不可估量的代码.....  
3.  
4.     public abstract void setContentView(int layoutResID);  
5.  
6.     public abstract void setContentView(View view);  
7.  
8.     public abstract void setContentView(View view, ViewGroup.LayoutParams pa  
rams);  
9.  
10.    // 省去数以亿计的代码.....  
11. }
```

其在 PhoneWindow 中都有具体的实现：

[java] view plain copy print?

```
1. public class PhoneWindow extends Window implements MenuBuilder.Callback {  
2.     // 省去草泥马个代码.....  
3.  
4.     @Override  
5.     public void setContentView(int layoutResID) {  
6.         if (mContentParent == null) {  
7.             installDecor();  
8.         } else {  
9.             mContentParent.removeAllViews();  
10.        }  
11.        mLayoutInflater.inflate(layoutResID, mContentParent);  
12.        final Callback cb = getCallback();  
13.        if (cb != null && !isDestroyed()) {  
14.            cb.onContentChanged();  
15.        }  
16.    }  
17.  
18.    @Override  
19.    public void setContentView(View view) {  
20.        setContentView(view, new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_  
PARENT));  
21.    }  
22.  
23.    @Override  
24.    public void setContentView(View view, ViewGroup.LayoutParams params) {  
25.        if (mContentParent == null) {  
26.            installDecor();  
27.        }  
28.        mContentParent.addView(view, params);  
29.    }  
30.    @Override  
31.    public void setFeatureDrawableResource(int featureId, int resourceId) {  
32.        if (mContentParent == null) {  
33.            installDecor();  
34.        }  
35.        mContentParent.setFeatureDrawableResource(featureId, resourceId);  
36.    }  
37.    @Override  
38.    public void setFeatureDrawableResource(int featureId, Drawable resource) {  
39.        if (mContentParent == null) {  
40.            installDecor();  
41.        }  
42.        mContentParent.setFeatureDrawableResource(featureId, resource);  
43.    }  
44.    @Override  
45.    public void setFeatureDrawableResource(int featureId, int[] resource) {  
46.        if (mContentParent == null) {  
47.            installDecor();  
48.        }  
49.        mContentParent.setFeatureDrawableResource(featureId, resource);  
50.    }  
51.    @Override  
52.    public void setFeatureDrawableResource(int featureId, Resource.Drawable resource) {  
53.        if (mContentParent == null) {  
54.            installDecor();  
55.        }  
56.        mContentParent.setFeatureDrawableResource(featureId, resource);  
57.    }  
58.    @Override  
59.    public void setFeatureDrawableResource(int featureId, int id) {  
60.        if (mContentParent == null) {  
61.            installDecor();  
62.        }  
63.        mContentParent.setFeatureDrawableResource(featureId, id);  
64.    }  
65.    @Override  
66.    public void setFeatureDrawableResource(int featureId, int id, int[] resource) {  
67.        if (mContentParent == null) {  
68.            installDecor();  
69.        }  
70.        mContentParent.setFeatureDrawableResource(featureId, id, resource);  
71.    }  
72.    @Override  
73.    public void setFeatureDrawableResource(int featureId, Resource.Drawable id, Resource.Drawable resource) {  
74.        if (mContentParent == null) {  
75.            installDecor();  
76.        }  
77.        mContentParent.setFeatureDrawableResource(featureId, id, resource);  
78.    }  
79.    @Override  
80.    public void setFeatureDrawableResource(int featureId, int id, int[] resource, int[] resource2) {  
81.        if (mContentParent == null) {  
82.            installDecor();  
83.        }  
84.        mContentParent.setFeatureDrawableResource(featureId, id, resource, resource2);  
85.    }  
86.    @Override  
87.    public void setFeatureDrawableResource(int featureId, Resource.Drawable id, Resource.Drawable resource, Resource.Drawable resource2) {  
88.        if (mContentParent == null) {  
89.            installDecor();  
90.        }  
91.        mContentParent.setFeatureDrawableResource(featureId, id, resource, resource2);  
92.    }  
93.    @Override  
94.    public void setFeatureDrawableResource(int featureId, int id, int[] resource, int[] resource2, int[] resource3) {  
95.        if (mContentParent == null) {  
96.            installDecor();  
97.        }  
98.        mContentParent.setFeatureDrawableResource(featureId, id, resource, resource2, resource3);  
99.    }  
100.   @Override  
101.   public void setFeatureDrawableResource(int featureId, Resource.Drawable id, Resource.Drawable resource, Resource.Drawable resource2, Resource.Drawable resource3) {  
102.       if (mContentParent == null) {  
103.           installDecor();  
104.       }  
105.       mContentParent.setFeatureDrawableResource(featureId, id, resource, resource2, resource3);  
106.   }  
107. }
```

```
27.         } else {
28.             mContentParent.removeAllViews();
29.         }
30.         mContentParent.addView(view, params);
31.         final Callback cb = getCallback();
32.         if (cb != null && !isDestroyed()) {
33.             cb.onContentChanged();
34.         }
35.     }
36.
37.     // 省去法克就个代码.....
38. }
```

当然如果你要是使用了 TV 的 SDK 那么这里就不是 PhoneWindow 而是 TVWindow 了，至于是不是呢？留给大家去验证。到这里我们都还没完，在 PhoneWindow 的 setContentView 方法中先会去判断 mContentParent 这个引用是否为空，如果为空则表示我们是第一次生成那么调用 installDecor 方法去生成一些具体的对象否则清空该 mContentParent 下的所有子元素（注意 mContentParent 是一个 ViewGroup）并通过 LayoutInflater 将 xml 布局转换为 View Tree 添加至 mContentParent 中（这里根据 setContentView(int layoutResID)方法分析，其他重载方法类似），installDecor 方法做的事相对多但不复杂，首先是对 DecorView 类型的 mDecor 成员变量赋值继而将其注入 generateLayout 方法生成我们的 mContentParent:

[java] view plain copy print?

```
1. private void installDecor() {
2.     if (mDecor == null) {
3.         mDecor = generateDecor();
4.         // 省省省.....
5.     }
6.
7.     if (mContentParent == null) {
8.         mContentParent = generateLayout(mDecor);
9.
10.        // 省省省.....
11.    }
12.
13.    // 省省省.....
14. }
```

generateLayout 方法中做的事就多了，简直可以跟 performTraversals 拼，这里不贴代码了简单分析一下，generateLayout 方法中主要根据当前我们的 Style 类型为当前 Window 选择不同的布局文件，看到这里，想必大家也该意识到，这才是我们的“根布局”，其会指定一个用来存放我们自定义布局文件（也就是我们口头上常说的根布局比如我们例子中的

`LinearLayout`)的 `ViewGroup`, 一般情况下这个 `ViewGroup` 的重任由 `FrameLayout` 来承担, 这也是为什么我们在获取我们 `xml` 布局文件中的顶层布局时调用其 `getParent()`方法会返回 `FrameLayout` 对象的原因, 其 `id` 为 `android:id="@+id/content"`:

[java] view plain copy print?

```
1. protected ViewGroup generateLayout(DecorView decor) {  
2.     // 省去巨量代码.....  
3.  
4.     ViewGroup contentParent = (ViewGroup) findViewById(ID_ANDROID_CONTENT);  
5.  
6.     // 省去一些代码.....  
7. }
```

在这个 `Window` 布局文件被确定后, `mDecor` 则会将该布局所生成的对应 `View` 添加进来并获取 `id` 为 `content` 的 `View` 将其赋给 `mContentParent`, 至此 `mContentParent` 和 `mDecor` 均已生成, 而我们 `xml` 布局文件中的布局则会被添加至 `mContentParent`。对应关系类似下图:

## Activity

**PhoneWindow(Window)**

**DecorView(FrameLayout)**

**OurContent(which view you want)**

**AigeStudio**

说了大半天才理清这个小关系，但是我们还没说到重点.....就是 widthMeasureSpec 和 heightMeasureSpec 究竟是从哪来的.....如果我们不做上面的一个分析，很多童鞋压根无从下手，有了上面一个分析，我们知道我们界面的真正根视图应该是 DecorView，那么我们的 widthMeasureSpec 和 heightMeasureSpec 应该从这里或者更上一层 PhoneWindow 传递进来对吧，但是 DecorView 是 FrameLayout 的一个实例，在 FrameLayout 的 onMeasure 中我们确实有对子元素的测量，但是问题是 FrameLayout:onMeasure 方法中的 widthMeasureSpec 和 heightMeasureSpec 又是从何而来呢？追溯上去我们又回到了 View.....不了解 Android GUI 框架的童鞋迈出的第一步就被无情地煽了回去。其实在 Android 中我们可以在很多方面看到类似 MVC 架构的影子，比如最最常见的就是我们的 xml 界面布局

——Activity 等组件——model 数据之间的关系，而在整个 GUI 的框架中，我们也可以对其做出类似的规划，View 在设计过程中就注定了其只会对显示数据进行处理比如我们的测量布局和绘制还有动画等等，而承担 Controller 控制器重任的是谁呢？在 Android 中这一功能由 ViewRootImpl 承担，我们在前面提到过这个类，其负责的东西很多，比如我们窗口的显示、用户的输入输出当然还有关于处理我们绘制流程的方法：

```
[java] view plaincopyprint?
```

```
1. private void performTraversals() {  
2.     // .....啦啦啦啦.....  
3. }
```

performTraversals 方法是处理绘制流程的一个开始，内部逻辑相当相当多&复杂，虽然没有 View 类复杂.....但是让我选的话我宁愿看整个 View 类也不愿看 performTraversals 方法那邪恶的逻辑.....囧，在该方法中我们可以看到如下的一段逻辑（具体各类变量的赋值就不贴了实在太多）：

```
[java] view plaincopyprint?
```

```
1. private void performTraversals() {  
2.     // .....省略宇宙尘埃数量那么多的代码.....  
3.  
4.     if (!mStopped) {  
5.         // .....省略一些代码  
6.  
7.         int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);  
8.         int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);  
9.  
10.        // .....省省省  
11.  
12.        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);  
13.    }  
14.  
15.    // .....省略人体细胞数量那么多的代码.....  
16. }
```

可以看到在 performTraversals 方法中通过 getRootMeasureSpec 获取原始的测量规格并将其作为参数传递给 performMeasure 方法处理，这里我们重点来看 getRootMeasureSpec 方法是如何确定测量规格的，首先我们要知道 mWidth, lp.width 和 mHeight, lp.height 这两组参数的意义，其中 lp.width 和 lp.height 均为 MATCH\_PARENT，其在 mWindowAttributes ( WindowManager.LayoutParams 类型) 将值赋予给 lp 时就已被确定，mWidth 和 mHeight 表示当前窗口的大小，其值由 performTraversals 中一系列逻辑计算确定，这里跳过，而在 getRootMeasureSpec 中作了如下判断：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. private static int getRootMeasureSpec(int windowSize, int rootDimension) {
2.     int measureSpec;
3.     switch (rootDimension) {
4.
5.         case ViewGroup.LayoutParams.MATCH_PARENT:
6.             // Window 不能调整其大小, 强制使根视图大小与 Window 一致
7.             measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.EX
8.                 ACTLY);
9.             break;
10.        case ViewGroup.LayoutParams.WRAP_CONTENT:
11.            // Window 可以调整其大小, 为根视图设置一个最大值
12.            measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.AT
13.                 _MOST);
14.            break;
15.        default:
16.            // Window 想要一个确定的尺寸, 强制将根视图的尺寸作为其尺寸
17.            measureSpec = MeasureSpec.makeMeasureSpec(rootDimension, MeasureSpec
18.                 .EXACTLY);
19.            break;
20.    }
21.    return measureSpec;
22. }
```

也就是说不管如何，我们的根视图大小必定都是全屏的.....

至此，我们算是真正接触到根视图的测量规格，尔后这个规格会被由上至下传递下去，并由当前 `view` 与其父容器共同作用决定最终的测量大小，在 `View` 与 `ViewGroup` 递归调用实现测量的过程中有几个重要的方法，对于 `View` 而言则是 `measure` 方法：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
2.     // 省略部分代码.....
3.
4.     /*
5.      * 判断当前 mPrivateFlags 是否带有 PFLAG_FORCE_LAYOUT 强制布局标记
6.      * 判断当前 widthMeasureSpec 和 heightMeasureSpec 是否发生了改变
7.      */
8.     if ((mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT ||
9.         widthMeasureSpec != mOldWidthMeasureSpec ||
10.         heightMeasureSpec != mOldHeightMeasureSpec) {
```

```
11.
12.        // 如果发生了改变表示需要重新进行测量此时清除掉 mPrivateFlags 中已测量的标识位 PFLAG_MEASURED_DIMENSION_SET
13.        mPrivateFlags &= ~PFLAG_MEASURED_DIMENSION_SET;
14.
15.        resolveRtlPropertiesIfNeeded();
16.
17.        int cacheIndex = (mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT ? -1 :
18.                mMeasureCache.indexOfKey(key);
19.        if (cacheIndex < 0 || sIgnoreMeasureCache) {
20.            // 测量 View 的尺寸
21.            onMeasure(widthMeasureSpec, heightMeasureSpec);
22.            mPrivateFlags3 &= ~PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
23.        } else {
24.            long value = mMeasureCache.valueAt(cacheIndex);
25.
26.            setMeasuredDimension((int) (value >> 32), (int) value);
27.            mPrivateFlags3 |= PFLAG3_MEASURE_NEEDED_BEFORE_LAYOUT;
28.        }
29.
30.        /*
31.         * 如果 mPrivateFlags 里没有表示已测量的标识位
32.         * PFLAG_MEASURED_DIMENSION_SET 则会抛出异常
33.         */
34.        if ((mPrivateFlags & PFLAG_MEASURED_DIMENSION_SET) != PFLAG_MEASURED_DIMENSION_SET) {
35.            throw new IllegalStateException("onMeasure() did not set the"
36.                    + " measured dimension by calling"
37.                    + " setMeasuredDimension()");
38.
39.        // 如果已测量 View 那么就可以往 mPrivateFlags 添加标识位
40.        // PFLAG_LAYOUT_REQUIRED 表示可以进行布局了
41.        mPrivateFlags |= PFLAG_LAYOUT_REQUIRED;
42.
43.        // 最后存储测量完成的测量规格
44.        mOldWidthMeasureSpec = widthMeasureSpec;
45.        mOldHeightMeasureSpec = heightMeasureSpec;
46.
47.        mMeasureCache.put(key, ((long) mMeasuredWidth) << 32 |
48.                            (long) mMeasuredHeight & 0xffffffffL); // suppress sign extensio
```

```
49. }
```

可以看到，View 对控件的测量是在 `onMeasure` 方法中进行的，也就是文章开头我们在自定义 View 中重写的 `onMeasure` 方法，但是我们并没有对其做任何的处理，也就是说保持了其在父类 View 中的默认实现，其默认实现也很简单：

```
[java] view plaincopyprint?
```

```
1. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
2.     setMeasuredDimension(getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec),
3.                           getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec));
4. }
```

其直接调用了 `setMeasuredDimension` 方法为其设置了两个计算后的测量值：

```
[java] view plaincopyprint?
```

```
1. protected final void setMeasuredDimension(int measuredWidth, int measuredHeight) {
2.     // 省去部分代码.....
3.
4.     // 设置测量后的宽高
5.     mMeasuredWidth = measuredWidth;
6.     mMeasuredHeight = measuredHeight;
7.
8.     // 重新将已测量标识位存入 mPrivateFlags 标识测量的完成
9.     mPrivateFlags |= PFLAG_MEASURED_DIMENSION_SET;
10. }
```

回到 `onMeasure` 方法，我们来看看这两个测量值具体是怎么获得的，其实非常简单，首先来看 `getSuggestedMinimumWidth` 方法：

```
[java] view plaincopyprint?
```

```
1. protected int getSuggestedMinimumWidth() {
2.     return (mBackground == null) ? mMinWidth : max(mMinWidth, mBackground.getMinimumWidth());
3. }
```

如果背景为空那么我们直接返回 `mMinWidth` 最小宽度否则就在 `mMinWidth` 和背景最小宽度之间取一个最大值，`getSuggestedMinimumHeight` 类同，`mMinWidth` 和 `mMinHeight` 我没记错的话应该都是 100px，而 `getDefaultSize` 方法呢也很简单：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public static int getDefaultSize(int size, int measureSpec) {
2.     // 将我们获得的最小值赋给 result
3.     int result = size;
4.
5.     // 从 measureSpec 中解算出测量规格的模式和尺寸
6.     int specMode = MeasureSpec.getMode(measureSpec);
7.     int specSize = MeasureSpec.getSize(measureSpec);
8.
9.     /*
10.      * 根据测量规格模式确定最终的测量尺寸
11.      */
12.     switch (specMode) {
13.         case MeasureSpec.UNSPECIFIED:
14.             result = size;
15.             break;
16.         case MeasureSpec.AT_MOST:
17.         case MeasureSpec.EXACTLY:
18.             result = specSize;
19.             break;
20.     }
21.     return result;
22. }
```

注意上述代码中当模式为 AT\_MOST 和 EXACTLY 时均会返回解算出的测量尺寸，还记得上面我们说的 PhoneWindow、DecorView 么从它们那里获取到的测量规格层层传递到我们的自定义 View 中，这就是为什么我们的 View 在默认情况下不管是 match\_parent 还是 wrap\_content 都能占满父容器的剩余空间（这里面还有父布局 LinearLayout 的作用就先略过了了解即可）。上述 onMeasure 的过程则是 View 默认的处理过程，如果我们不喜欢 Android 帮我们处理那么我们可以自己重写 onMeasure 实现自己的测量逻辑：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. @Override
2. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.     // 设置测量尺寸
4.     setMeasuredDimension(250, 250);
5. }
```

最简单的粗暴的就是直接将两个值作为参数传入 setMeasuredDimension 方法，效果如下：



AigeStudio

AigeStudio

# AigeStudio

当然这样不好,用 Android 官方的话来说就是太过“专政”,因为它完全摒弃了父容器的意愿,完全由自己决定了大小,如果大家逛 blog 看技术文章或者听别人讨论常常会听到别人这么说 view 的最终测量尺寸是由 view 本身何其父容器共同决定的,至于如何共同决定我们呆会再说,这里我们先看看如何能在一定程度上顺应爹的“意愿”呢?从 View 默认的测量模式中我们可以看到它频繁使用了一个叫做 MeasureSpec 的类,而在 ViewRootImpl 中呢也有大量用到该类,该类的具体说明大家可以围观我早期的一篇文章:

<http://blog.csdn.net/aigestudio/article/details/38636531>,里面有对 MeasureSpec 类的详细说明,这里我就简单概述下 MeasureSpec 类中的三个 Mode 常量值的意义,其中 UNSPECIFIED 表示未指定,爹不会对儿子作任何的束缚,儿子想要多大都可以; EXACTLY 表示完全的,意为儿子多大爹心里有数,爹早已算好了; AT\_MOST 表示至多,爹已经为儿

子设置好了一个最大限制，儿子你不能比这个值大，不能再多了！父容器所谓的“意图”其实就由上述三个常量值表现，既然如此我们就该对这三个 Mode 常量做一个判断才行，不然怎么知道爹的意图呢：

[java] view plain copy print?

```
1.  @Override
2.  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.      // 声明一个临时变量来存储计算出的测量值
4.      int resultWidth = 0;
5.
6.      // 获取宽度测量规格中的 mode
7.      int modeWidth = MeasureSpec.getMode(widthMeasureSpec);
8.
9.      // 获取宽度测量规格中的 size
10.     int sizeWidth = MeasureSpec.getSize(widthMeasureSpec);
11.
12.     /*
13.      * 如果爹心里有数
14.      */
15.     if (modeWidth == MeasureSpec.EXACTLY) {
16.         // 那么儿子也不要让爹难做就取爹给的大小吧
17.         resultWidth = sizeWidth;
18.     }
19.     /*
20.      * 如果爹心里没数
21.      */
22.     else {
23.         // 那么儿子可要自己看看自己需要多大了
24.         resultWidth = mBitmap.getWidth();
25.
26.         /*
27.          * 如果爹给儿子的是一个限制值
28.          */
29.         if (modeWidth == MeasureSpec.AT_MOST) {
30.             // 那么儿子自己的需求就要跟爹的限制比比看谁小要谁
31.             resultWidth = Math.min(resultWidth, sizeWidth);
32.         }
33.     }
34.
35.     int resultHeight = 0;
36.     int modeHeight = MeasureSpec.getMode(heightMeasureSpec);
37.     int sizeHeight = MeasureSpec.getSize(heightMeasureSpec);
38.
```

```
39.     if (modeHeight == MeasureSpec.EXACTLY) {
40.         resultHeight = sizeHeight;
41.     } else {
42.         resultHeight = mBitmap.getHeight();
43.         if (modeHeight == MeasureSpec.AT_MOST) {
44.             resultHeight = Math.min(resultHeight, sizeHeight);
45.         }
46.     }
47.
48.     // 设置测量尺寸
49.     setMeasuredDimension(resultWidth, resultHeight);
50. }
```

如上代码所示我们从父容器传来的 `MeasureSpec` 中分离出了 `mode` 和 `size`, `size` 只是一个期望值我们需要根据 `mode` 来计算最终的 `size`, 如果父容器对子元素没有一个确切的大小那么我们就需要尝试去计算子元素也就是我们的自定义 `View` 的大小, 而这部分大小更多的是由我们也就是开发者去根据实际情况计算的, 这里我们模拟的是一个显示图片的控件, 那么控件的实际大小就应该跟我们的图片一致, 但是虽然我们可以做出一定的决定也要考虑父容器的限制值, 当 `mode` 为 `AT_MOST` 时 `size` 则是父容器给予我们的一个最大值, 我们控件的大小就不应该超过这个值。下面是运行效果:



AigeStudio

AigeStudio

如我所说，控件的实际大小需要根据我们的实际需求去计算，这里我更改一下 xml 为我们的 ImageView 加一个内边距值：

[html] view plaincopyprint?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:background="#FFFFFF"
5.     android:orientation="vertical" >
6.
7.     <com.aigestudio.customviewdemo.views.ImageView
8.         android:id="@+id/main_pv"
9.         android:layout_width="wrap_content"
```

```
10.        android:layout_height="wrap_content"
11.        android:padding="20dp" />
12.
13.    <Button
14.        android:layout_width="wrap_content"
15.        android:layout_height="wrap_content"
16.        android:text="AigeStudio" />
17.
18.    <TextView
19.        android:layout_width="wrap_content"
20.        android:layout_height="wrap_content"
21.        android:text="AigeStudio" />
22.
23. </LinearLayout>
```

这时你会发现蛋疼了……毫无内边距的效果，而在这种情况下我们则需在计算控件尺寸时考虑内边距的大小：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. resultWidth = mBitmap.getWidth() + getPaddingLeft() + getPaddingRight();
2. resultHeight = mBitmap.getHeight() + getPaddingTop() + getPaddingBottom();
```

这时我们就有了内边距的效果对吧：



唉、等等，好像不对啊，上边距和左边距为什么没有了？原因很简单，因为我们在绘制时并没有考虑到 **Padding** 的影响，下面我们更改一下绘制逻辑：

```
[java] view plain copy print?
```

```
1. @Override
2. protected void onDraw(Canvas canvas) {
3.     // 绘制位图
4.     canvas.drawBitmap(mBitmap, getPaddingLeft(), getPaddingTop(), null);
5. }
```

这时我们的内边距就完美了：



AigeStudio

AigeStudio

很多朋友问那 Margin 外边距呢？？淡定，外边距轮不到 view 来算，Andorid 将其封装在 LayoutParams 内交由父容器统一处理。很多时候我们的控件往往不只是简单的图片那么乏味，比如类似图标的效果：



一个图标常常除了一张图片外底部还有一个 title，这时我们的测量逻辑就应该做出相应的改变了，这里我用一个新的 IconView 做:

```
[java] view plaincopyprint?
```

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/13
5. *
6. */
7. public class IconView extends View {
8.     private Bitmap mBitmap;// 位图
9.     private TextPaint mPaint;// 绘制文本的画笔
```

```
10.     private String mStr;// 绘制的文本
11.
12.     private float mTextSize;// 画笔的文本尺寸
13.
14.     /**
15.      * 宽高枚举类
16.      *
17.      * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
18.      *
19.      */
20.     private enum Ratio {
21.         WIDTH, HEIGHT
22.     }
23.
24.     public IconView(Context context, AttributeSet attrs) {
25.         super(context, attrs);
26.
27.         // 计算参数
28.         calArgs(context);
29.
30.         // 初始化
31.         init();
32.     }
33.
34.     /**
35.      * 参数计算
36.      *
37.      * @param context
38.      *          上下文环境引用
39.      */
40.     private void calArgs(Context context) {
41.         // 获取屏幕宽
42.         int sreenW = MeasureUtil.getScreenSize((Activity) context)[0];
43.
44.         // 计算文本尺寸
45.         mTextSize = sreenW * 1 / 10F;
46.     }
47.
48.     /**
49.      * 初始化
50.      */
51.     private void init() {
52.         /*
53.             * 获取 Bitmap
```

```
54.         */
55.         if (null == mBitmap) {
56.             mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.logo);
57.         }
58.
59.         /*
60.          * 为 mStr 赋值
61.          */
62.         if (null == mStr || mStr.trim().length() == 0) {
63.             mStr = "AigeStudio";
64.         }
65.
66.         /*
67.          * 初始化画笔并设置参数
68.          */
69.         mPaint = new TextPaint(Paint.ANTI_ALIAS_FLAG | Paint.DITHER_FLAG | Paint.LINEAR_TEXT_FLAG);
70.         mPaint.setColor(Color.LTGRAY);
71.         mPaint.setTextSize(mTextSize);
72.         mPaint.setTextAlign(Paint.Align.CENTER);
73.         mPaint.setTypeface(Typeface.DEFAULT_BOLD);
74.     }
75.
76.     @Override
77.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
78.
79.         // 设置测量后的尺寸
80.         setMeasuredDimension(getMeasureSize(widthMeasureSpec, Ratio.WIDTH),
81.             getMeasureSize(heightMeasureSpec, Ratio.HEIGHT));
82.     }
83.     /**
84.      * 获取测量后的尺寸
85.      *
86.      * @param measureSpec
87.      *          测量规格
88.      * @param ratio
89.      *          宽高标识
90.      */
91.     private int getMeasureSize(int measureSpec, Ratio ratio) {
92.         // 声明临时变量保存测量值
93.         int result = 0;
```

```
94.  
95.        /*  
96.         * 获取 mode 和 size  
97.         */  
98.        int mode = MeasureSpec.getMode(measureSpec);  
99.        int size = MeasureSpec.getSize(measureSpec);  
100.  
101.       /*  
102.        * 判断 mode 的具体值  
103.        */  
104.       switch (mode) {  
105.           case MeasureSpec.EXACTLY:// EXACTLY 时直接赋值  
106.               result = size;  
107.               break;  
108.           default:// 默认情况下将 UNSPECIFIED 和 AT_MOST 一并处理  
109.               if (ratio == Ratio.WIDTH) {  
110.                   float textWidth = mPaint.measureText(mStr);  
111.                   result = ((int) (textWidth >= mBitmap.getWidth() ? textWidth :  
112.                               mBitmap.getWidth())) + getPaddingLeft() + getPaddingRight();  
113.               } else if (ratio == Ratio.HEIGHT) {  
114.                   result = ((int) ((mPaint.descent() - mPaint.ascent()) * 2 +  
115.                               mBitmap.getHeight())) + getPaddingTop() + getPaddingBottom();  
116.               }  
117.               /*  
118.                * AT_MOST 时判断 size 和 result 的大小取小值  
119.                */  
120.               if (mode == MeasureSpec.AT_MOST) {  
121.                   result = Math.min(result, size);  
122.               }  
123.               break;  
124.       }  
125.       return result;  
126.  
127.   @Override  
128.   protected void onDraw(Canvas canvas) {  
129.       /*  
130.        * 绘制  
131.        * 参数就不做单独处理了因为只会 Draw 一次不会频繁调用  
132.        */  
133.       canvas.drawBitmap(mBitmap, getWidth() / 2 - mBitmap.getWidth() / 2,  
134.                           getHeight() / 2 - mBitmap.getHeight() / 2, null);
```

```
134.         canvas.drawText(mStr, getWidth() / 2, mBitmap.getHeight() + getHeig
    ht() / 2 - mBitmap.getHeight() / 2 - mPaint.ascent(), mPaint);
135.     }
136. }
```

在 xml 文件中对其引用并加入一些系统自带的控件：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:background="#FFFFFF"
5.     android:orientation="vertical" >
6.
7.     <com.aigestudio.customviewdemo.views.IconView
8.         android:id="@+id/main_pv"
9.         android:layout_width="wrap_content"
10.        android:layout_height="wrap_content"
11.        android:padding="50dp" />
12.
13.     <Button
14.         android:layout_width="wrap_content"
15.         android:layout_height="wrap_content"
16.         android:text="AigeStudio" />
17.
18.     <TextView
19.         android:layout_width="wrap_content"
20.         android:layout_height="wrap_content"
21.         android:text="AigeStudio" />
22.
23. </LinearLayout>
```

效果如下：



好了就先这样吧，上面我们曾说过 `View` 的测量大小是由 `View` 和其父容器共同决定的，但是上述源码的分析中我们其实并没有体现，因为它们都在 `ViewGroup` 中，这里我们就要涉及 `ViewGroup` 中与测量相关的另外几个方法：`measureChildren`、`measureChild` 和 `measureChildWithMargins` 还有 `getChildMeasureSpec`，见名知意这几个方法都跟 `ViewGroup` 测量子元素有关，其中 `measureChildWithMargins` 和 `measureChildren` 类似只是加入了对 `Margins` 外边距的处理，`ViewGroup` 提供对子元素测量的方法从 `measureChildren` 开始：

`[java] view plain copy print?`

```
1. protected void measureChildren(int widthMeasureSpec, int heightMeasureSpec)  
{
```

```
2.     final int size = mChildrenCount;
3.     final View[] children = mChildren;
4.     for (int i = 0; i < size; ++i) {
5.         final View child = children[i];
6.         if ((child.mViewFlags & VISIBILITY_MASK) != GONE) {
7.             measureChild(child, widthMeasureSpec, heightMeasureSpec);
8.         }
9.     }
10. }
```

`measureChildren` 的逻辑很简单，通过父容器传入的 `widthMeasureSpec` 和 `heightMeasureSpec` 遍历子元素并调用 `measureChild` 方法去测量每一个子元素的宽高：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. protected void measureChild(View child, int parentWidthMeasureSpec,
2.                               int parentHeightMeasureSpec) {
3.     // 获取子元素的布局参数
4.     final LayoutParams lp = child.getLayoutParams();
5.
6.     /*
7.      * 将父容器的测量规格已经上下和左右的边距还有子元素本身的布局参数传入
8.      getChildMeasureSpec 方法计算最终测量规格
9.     */
10.    final int childWidthMeasureSpec = getChildMeasureSpec(parentWidthMeasureSpec,
11.                                                       mPaddingLeft + mPaddingRight, lp.width);
12.    final int childHeightMeasureSpec = getChildMeasureSpec(parentHeightMeasureSpec,
13.                                                       mPaddingTop + mPaddingBottom, lp.height);
14.    // 调用子元素的 measure 传入计算好的测量规格
15.    child.measure(childWidthMeasureSpec, childHeightMeasureSpec);
16. }
```

这里我们主要就是看看 `getChildMeasureSpec` 方法是如何确定最终测量规格的：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public static int getChildMeasureSpec(int spec, int padding, int childDimension) {
2.     // 获取父容器的测量模式和尺寸大小
3.     int specMode = MeasureSpec.getMode(spec);
4.     int specSize = MeasureSpec.getSize(spec);
```

```
5.      // 这个尺寸应该减去内边距的值
6.      int size = Math.max(0, specSize - padding);
7.
8.
9.      // 声明临时变量存值
10.     int resultSize = 0;
11.     int resultMode = 0;
12.
13.     /*
14.      * 根据模式判断
15.      */
16.     switch (specMode) {
17.         case MeasureSpec.EXACTLY: // 父容器尺寸大小是一个确定的值
18.             /*
19.              * 根据子元素的布局参数判断
20.              */
21.             if (childDimension >= 0) { //如果 childDimension 是一个具体的值
22.                 // 那么就将该值作为结果
23.                 resultSize = childDimension;
24.
25.                 // 而这个值也是被确定的
26.                 resultMode = MeasureSpec.EXACTLY;
27.             } else if (childDimension == LayoutParams.MATCH_PARENT) { //如果子元
素的布局参数为 MATCH_PARENT
28.                 // 那么就将父容器的大小作为结果
29.                 resultSize = size;
30.
31.                 // 因为父容器的大小是被确定的所以子元素大小也是可以被确定的
32.                 resultMode = MeasureSpec.EXACTLY;
33.             } else if (childDimension == LayoutParams.WRAP_CONTENT) { //如果子元
素的布局参数为 WRAP_CONTENT
34.                 // 那么就将父容器的大小作为结果
35.                 resultSize = size;
36.
37.                 // 但是子元素的大小包裹了其内容后不能超过父容器
38.                 resultMode = MeasureSpec.AT_MOST;
39.             }
40.             break;
41.
42.         case MeasureSpec.AT_MOST: // 父容器尺寸大小拥有一个限制值
43.             /*
44.              * 根据子元素的布局参数判断
45.              */
46.             if (childDimension >= 0) { //如果 childDimension 是一个具体的值
```

```
47.          // 那么就将该值作为结果
48.          resultSize = childDimension;
49.
50.          // 而这个值也是被确定的
51.          resultMode = MeasureSpec.EXACTLY;
52.      } else if (childDimension == LayoutParams.MATCH_PARENT) { //如果子元
素的布局参数为 MATCH_PARENT
53.          // 那么就将父容器的大小作为结果
54.          resultSize = size;
55.
56.          // 因为父容器的大小是受到限制值的限制所以子元素的大小也应该受到父容器
的限制
57.          resultMode = MeasureSpec.AT_MOST;
58.      } else if (childDimension == LayoutParams.WRAP_CONTENT) { //如果子元
素的布局参数为 WRAP_CONTENT
59.          // 那么就将父容器的大小作为结果
60.          resultSize = size;
61.
62.          // 但是子元素的大小包裹了其内容后不能超过父容器
63.          resultMode = MeasureSpec.AT_MOST;
64.      }
65.      break;
66.
67.  case MeasureSpec.UNSPECIFIED: // 父容器尺寸大小未受限制
68.      /*
69.       * 根据子元素的布局参数判断
70.       */
71.      if (childDimension >= 0) { //如果 childDimension 是一个具体的值
72.          // 那么就将该值作为结果
73.          resultSize = childDimension;
74.
75.          // 而这个值也是被确定的
76.          resultMode = MeasureSpec.EXACTLY;
77.      } else if (childDimension == LayoutParams.MATCH_PARENT) { //如果子元
素的布局参数为 MATCH_PARENT
78.          // 因为父容器的大小不受限制而对子元素来说也可以是任意大小所以不指定也
不限制子元素的大小
79.          resultSize = 0;
80.          resultMode = MeasureSpec.UNSPECIFIED;
81.      } else if (childDimension == LayoutParams.WRAP_CONTENT) { //如果子元
素的布局参数为 WRAP_CONTENT
82.          // 因为父容器的大小不受限制而对子元素来说也可以是任意大小所以不指定也
不限制子元素的大小
83.          resultSize = 0;
```

```
84.         resultMode = MeasureSpec.UNSPECIFIED;
85.     }
86.     break;
87. }
88.
89. // 返回封装后的测量规格
90. return MeasureSpec.makeMeasureSpec(resultSize, resultMode);
91. }
```

至此我们可以看到一个 View 的大小由其父容器的测量规格 MeasureSpec 和 View 本身的布局参数 LayoutParams 共同决定，但是即便如此，最终封装的测量规格也是一个期望值，究竟有多大还是我们调用 setMeasuredDimension 方法设置的。上面的代码中有些朋友看了可能会有疑问为什么 childDimension  $\geq 0$  就表示一个确切值呢？原因很简单，因为在 LayoutParams 中 MATCH\_PARENT 和 WRAP\_CONTENT 均为负数、哈哈！！正是基于这点，Android 巧妙地将实际值和相对的布局参数分离开来。那么我们该如何对 ViewGroup 进行测量呢？这里为了说明问题，我们自定义一个 ViewGroup：

[java] view plain copy print?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/15
5. *
6. */
7. public class CustomLayout extends ViewGroup {
8.
9.     public CustomLayout(Context context, AttributeSet attrs) {
10.         super(context, attrs);
11.     }
12.
13.     @Override
14.     protected void onLayout(boolean changed, int l, int t, int r, int b) {
15.
16.     }
17.
18. }
```

ViewGroup 中的 onLayout 方法是一个抽象方法，这意味着你在继承时必须实现，onLayout 的目的是为了确定子元素在父容器中的位置，那么这个步骤理应该由父容器来决定而不是子元素，因此，我们可以猜到 View 中的 onLayout 方法应该是一个空实现：

[java] view plain copy print?

```
1. public class View implements Drawable.Callback, KeyEvent.Callback,
2.         AccessibilityEventSource {
3.     // 省去无数代码.....
4.
5.     /**
6.      * Called from layout when this view should
7.      * assign a size and position to each of its children.
8.      *
9.      * Derived classes with children should override
10.     * this method and call layout on each of
11.     * their children.
12.     * @param changed This is a new size or position for this view
13.     * @param left Left position, relative to parent
14.     * @param top Top position, relative to parent
15.     * @param right Right position, relative to parent
16.     * @param bottom Bottom position, relative to parent
17.     */
18.     protected void onLayout(boolean changed, int left, int top, int right, i
nt bottom) {
19. }
20.
21.     // 省去无数代码.....
22. }
```

与 View 不同的是，ViewGroup 表示一个容器，其内可以包含多个元素，既可以是一个布局也可以是一个普通的控件，那么在对 ViewGroup 测量时我们也应该对这些子元素进行测量：

[java] view plaincopyprint?

```
1. /**
2.  *
3.  * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4.  * @since 2015/1/15
5.  *
6. */
7. public class CustomLayout extends ViewGroup {
8.
9.     public CustomLayout(Context context, AttributeSet attrs) {
10.         super(context, attrs);
11.     }
12.
13.     @Override
14.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
```

```
15.         super.onMeasure(widthMeasureSpec, heightMeasureSpec);
16.
17.         /*
18.             * 如果有子元素
19.             */
20.         if (getChildCount() > 0) {
21.             // 那么对子元素进行测量
22.             measureChildren(widthMeasureSpec, heightMeasureSpec);
23.         }
24.     }
25.
26.     @Override
27.     protected void onLayout(boolean changed, int l, int t, int r, int b) {
28.
29.     }
30.
31. }
```

然后我们在 xml 布局文件中替换原来的 LinearLayout 使用我们自定义的布局：

[html] [view](#) [plain](#) [copy](#) [print](#)

```
1.  <com.aigestudio.customviewdemo.views.CustomLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      android:layout_width="match_parent"
3.      android:layout_height="match_parent"
4.      android:background="#FFFFFF"
5.      android:orientation="vertical" >
6.
7.      <com.aigestudio.customviewdemo.views.IconView
8.          android:id="@+id/main_pv"
9.          android:layout_width="wrap_content"
10.         android:layout_height="wrap_content"
11.         android:padding="50dp" />
12.
13.      <Button
14.          android:layout_width="wrap_content"
15.          android:layout_height="wrap_content"
16.          android:text="AigeStudio" />
17.
18.      <TextView
19.          android:layout_width="wrap_content"
20.          android:layout_height="wrap_content"
21.          android:text="AigeStudio" />
```

22.

23. </com.aigestudio.customviewdemo.views.CustomLayout>

运行后你会发现没有任何东西显示，为什么呢？如上所说我们需要父容器告诉子元素它的出现位置，而这个过程由 `onLayout` 方法去实现，但是此时我们的 `onLayout` 方法什么都没有，子元素自然也不知道自己该往哪搁，自然就什么都没有咯……知道了原因我们就来实现 `onLayout` 的逻辑：

[java] view plain copy print?

```
1. @Override
2. protected void onLayout(boolean changed, int l, int t, int r, int b) {
3.
4.     /*
5.      * 如果有子元素
6.      */
7.     if (getChildCount() > 0) {
8.         // 那么遍历子元素并对其进行定位布局
9.         for (int i = 0; i < getChildCount(); i++) {
10.             View child = getChildAt(i);
11.             child.layout(0, 0, getMeasuredWidth(), getMeasuredHeight());
12.         }
13.     }
14. }
```

逻辑很简单，如果有子元素那么我们遍历这些子元素并调用其 `layout` 方法告诉它们自己该在的位置，这里我们就直接让所有的子元素都从父容器的 [0, 0] 点开始到 [`getMeasuredWidth()`, `getMeasuredHeight()`] 父容器的测量宽高结束，这么一来，所有的子元素应该都是填充了父容器的对吧：



看到屏幕上的巨大 Button 我不禁吸了一口屁！这样的布局太蛋疼，全被 Button 一个玩完了还搞毛，可不可以像 LinearLayout 那样挨个显示呢？答案是肯定的！我们来修改下 `onLayout` 的逻辑：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1.  @Override
2.  protected void onLayout(boolean changed, int l, int t, int r, int b) {
3.
4.      /*
5.       * 如果有子元素
6.       */
7.      if (getChildCount() > 0) {
```

```
8.         // 声明一个临时变量存储高度倍增值
9.         int muntilHeight = 0;
10.
11.        // 那么遍历子元素并对其进行定位布局
12.        for (int i = 0; i < getChildCount(); i++) {
13.            // 获取一个子元素
14.            View child = getChildAt(i);
15.
16.            // 通知子元素进行布局
17.            child.layout(0, muntilHeight, child.getMeasuredWidth(), child.get
18.            MeasuredHeight() + muntilHeight);
19.            // 改变高度倍增值
20.            muntilHeight += child.getMeasuredHeight();
21.        }
22.    }
23. }
```

可以看到我们通过一个 `muntilHeight` 来存储高度倍增值，每一次子元素布局完后将当前 `muntilHeight` 与当前子元素的高度相加并在下一个子元素布局时在高度上加上 `muntilHeight`，效果如下：



AigeStudio **AigeStudio**

AigeStudio

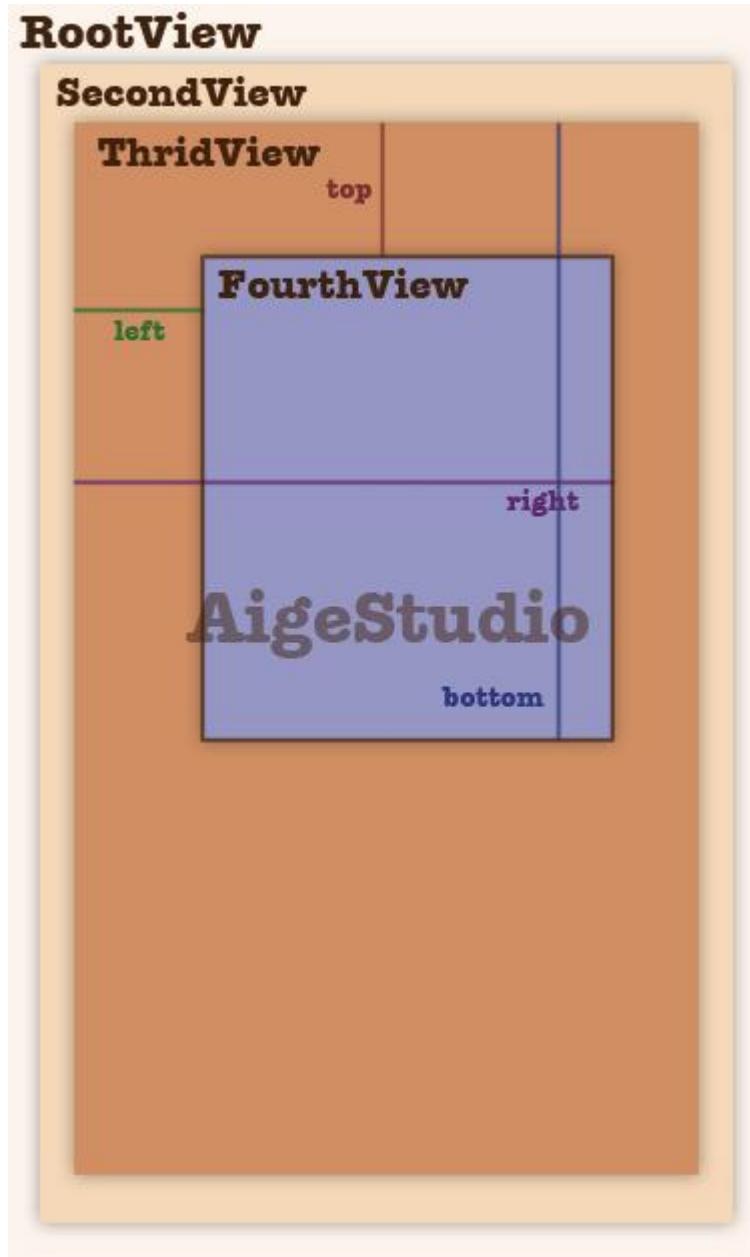
是不是和上面 `LinearLayout` 效果有点一样了？当然 `LinearLayout` 的布局逻辑远比我们的复杂得多，我们呢也只是对其进行一个简单的模拟而已。大家注意到 `ViewGroup` 的 `onLayout` 方法的签名列表中有五个参数，其中 `boolean changed` 表示是否与上一次位置不同，其具体值在 `View` 的 `layout` 方法中通过 `setFrame` 等方法确定：

[java] view plaincopyprint?

```
1. public void layout(int l, int t, int r, int b) {  
2.     // 省略一些代码.....  
3.  
4.     boolean changed = isLayoutModeOptical(mParent) ?  
5.             setOpticalFrame(l, t, r, b) : setFrame(l, t, r, b);  
6.
```

```
7.      // 省略大量代码.....  
8. }
```

而剩下的四个参数则表示当前 View 与父容器的相对距离，如下图：



好了，说到这里想必大家对 **ViewGroup** 的测量也有一定的了解了，但是这必定不是测量过程全部，如我上面所说，测量的具体过程因控件而异，上面我们曾因为给我们的自定义 View 加了内边距后修改了绘制的逻辑，因为我们需要在绘制时考虑内边距的影响，而我们的自定义 **ViewGroup** 呢？是不是也一样呢？这里我给其加入 60dp 的内边距：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <com.aigestudio.customviewdemo.views.CustomLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:padding="60dp"
5.     android:background="#FFFFFF"
6.     android:orientation="vertical" >
7.
8.     <com.aigestudio.customviewdemo.views.IconView
9.         android:id="@+id/main_pv"
10.        android:layout_width="wrap_content"
11.        android:layout_height="wrap_content"
12.        android:padding="50dp" />
13.
14.     <Button
15.         android:layout_width="wrap_content"
16.         android:layout_height="wrap_content"
17.         android:text="AigeStudio" />
18.
19.     <TextView
20.         android:layout_width="wrap_content"
21.         android:layout_height="wrap_content"
22.         android:text="AigeStudio" />
23.
24. </com.aigestudio.customviewdemo.views.CustomLayout>
```

运行后效果如下：



内边距把我们的子元素给“吃”掉了，那么也就是说我们在对子元素进行定位时应该进一步考虑到父容器内边距的影响对吧，OK，我们重理 `onLayout` 的逻辑：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. @Override
2. protected void onLayout(boolean changed, int l, int t, int r, int b) {
3.     // 获取父容器内边距
4.     int parentPaddingLeft = getPaddingLeft();
5.     int parentPaddingTop = getPaddingTop();
6.
7.     /*
8.      * 如果有子元素
9.      */

```

```
10.     if (getChildCount() > 0) {
11.         // 声明一个临时变量存储高度倍增值
12.         int mUtilHeight = 0;
13.
14.         // 那么遍历子元素并对其进行定位布局
15.         for (int i = 0; i < getChildCount(); i++) {
16.             // 获取一个子元素
17.             View child = getChildAt(i);
18.
19.             // 通知子元素进行布局
20.             // 此时考虑父容器内边距的影响
21.             child.layout(parentPaddingLeft, mUtilHeight + parentPaddingTop,
22.                         child.getMeasuredWidth() + parentPaddingLeft, child.getMeasuredHeight() + mu-
23.                         tilHeight + parentPaddingTop);
24.
25.             // 改变高度倍增值
26.             mUtilHeight += child.getMeasuredHeight();
27.         }
28.     }
29. }
```

此时的效果如下：



既然内边距如此，那么 Margins 外边距呢？我们来看看，在 xml 布局文件中为我们的 CustomLayout 加一个 margins：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <com.aigestudio.customviewdemo.views.CustomLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:layout_margin="30dp"
5.     android:padding="20dp"
6.     android:background="#FF597210"
7.     android:orientation="vertical" >
8.
```

```
9.      <com.aigestudio.customviewdemo.views.IconView
10.         android:id="@+id/main_pv"
11.         android:layout_width="wrap_content"
12.         android:layout_height="wrap_content" />
13.
14.      <Button
15.         android:layout_width="wrap_content"
16.         android:layout_height="wrap_content"
17.         android:text="AigeStudio" />
18.
19.      <TextView
20.         android:layout_width="wrap_content"
21.         android:layout_height="wrap_content"
22.         android:text="AigeStudio" />
23.
24. </com.aigestudio.customviewdemo.views.CustomLayout>
```

效果如下：



OK，目测没什么问题，可是当我们为子元素设置外边距时，问题就来了……不管你怎么设都不会有任何效果，原因很简单，我们上面也说了，**Margins** 是由父容器来处理，而我们的**CustomLayout** 中并没有对其做任何的处理，那么我们应该怎么做呢？首先要知道**Margins** 封装在**LayoutParams** 中，如果我们想实现自己对它的处理那么我们必然也有必要实现自己布局的**LayoutParams**:

[\[java\]](#) [view plain](#) [copy](#) [print?](#)

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/15
5. *
```

```
6.  */
7. public class CustomLayout extends ViewGroup {
8.     // 省略部分代码.....
9.
10.    /**
11.     *
12.     * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
13.     *
14.     */
15.    public static class CustomLayoutParams extends MarginLayoutParams {
16.
17.        public CustomLayoutParams(MarginLayoutParams source) {
18.            super(source);
19.        }
20.
21.        public CustomLayoutParams(android.view.ViewGroup.LayoutParams source
22. ) {
22.            super(source);
23.        }
24.
25.        public CustomLayoutParams(Context c, AttributeSet attrs) {
26.            super(c, attrs);
27.        }
28.
29.        public CustomLayoutParams(int width, int height) {
30.            super(width, height);
31.        }
32.    }
33. }
```

我们在我们的 `CustomLayout` 中生成了一个静态内部类 `CustomLayoutParams`，保持其默认的构造方法即可，这里我们什么也没做，当然你可以定义自己的一些属性或逻辑处理，因控件而异这里不多说了，后面慢慢会用到。然后在我们的 `CustomLayout` 中重写所有与 `LayoutParams` 相关的方法，返回我们自己的 `CustomLayoutParams`:

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/15
5. *
6. */
7. public class CustomLayout extends ViewGroup {
```

```
8.     // 省略部分代码.....
9.
10.    /**
11.     * 生成默认的布局参数
12.     */
13.    @Override
14.    protected CustomLayoutParams generateDefaultLayoutParams() {
15.        return new CustomLayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, V
iewGroup.LayoutParams.MATCH_PARENT);
16.    }
17.
18.    /**
19.     * 生成布局参数
20.     * 将布局参数包装成我们的
21.     */
22.    @Override
23.    protected android.view.ViewGroup.LayoutParams generateLayoutParams(android
.view.ViewGroup.LayoutParams p) {
24.        return new CustomLayoutParams(p);
25.    }
26.
27.    /**
28.     * 生成布局参数
29.     * 从属性配置中生成我们的布局参数
30.     */
31.    @Override
32.    public android.view.ViewGroup.LayoutParams generateLayoutParams(Attribute
eSet attrs) {
33.        return new CustomLayoutParams(getContext(), attrs);
34.    }
35.
36.    /**
37.     * 检查当前布局参数是否是我们定义的类型这在 code 声明布局参数时常常用到
38.     */
39.    @Override
40.    protected boolean checkLayoutParams(android.view.ViewGroup.LayoutParams
p) {
41.        return p instanceof CustomLayoutParams;
42.    }
43.
44.    // 省略部分代码.....
45. }
```

最后更改我们的测量逻辑：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1.  @Override
2.  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
3.      // 声明临时变量存储父容器的期望值
4.      int parentDesireWidth = 0;
5.      int parentDesireHeight = 0;
6.
7.      /*
8.       * 如果有子元素
9.       */
10.     if (getChildCount() > 0) {
11.         // 那么遍历子元素并对其进行测量
12.         for (int i = 0; i < getChildCount(); i++) {
13.
14.             // 获取子元素
15.             View child = getChildAt(i);
16.
17.             // 获取子元素的布局参数
18.             CustomLayoutParams clp = (CustomLayoutParams) child.getLayoutPar-
19.             ams();
20.
21.             // 测量子元素并考虑外边距
22.             measureChildWithMargins(child, widthMeasureSpec, 0, heightMeasur-
23.             eSpec, 0);
24.
25.             // 计算父容器的期望值
26.             parentDesireWidth += child.getMeasuredWidth() + clp.leftMargin +
27.               clp.rightMargin;
28.             parentDesireHeight += child.getMeasuredHeight() + clp.topMargin +
29.               clp.bottomMargin;
30.
31.             }
32.
33.             // 考虑父容器的内边距
34.             parentDesireWidth += getPaddingLeft() + getPaddingRight();
35.             parentDesireHeight += getPaddingTop() + getPaddingBottom();
36.
37.             // 尝试比较建议最小值和期望值的大小并取大值
38.             parentDesireWidth = Math.max(parentDesireWidth, getSuggestedMinimumW-
39.             idth());
40.             parentDesireHeight = Math.max(parentDesireHeight, getSuggestedMinimu-
41.             mHeight());
42.
43.     }
44. }
```

```
36.  
37.     // 设置最终测量值 0  
38.     setMeasuredDimension(resolveSize(parentDesireWidth, widthMeasureSpec), r  
      esolveSize(parentDesireHeight, heightMeasureSpec));  
39. }  
40.  
41. @Override  
42. protected void onLayout(boolean changed, int l, int t, int r, int b) {  
43.     // 获取父容器内边距  
44.     int parentPaddingLeft = getPaddingLeft();  
45.     int parentPaddingTop = getPaddingTop();  
46.  
47.     /*  
48.      * 如果有子元素  
49.      */  
50.     if (getChildCount() > 0) {  
51.         // 声明一个临时变量存储高度倍增值  
52.         int mUtilHeight = 0;  
53.  
54.         // 那么遍历子元素并对其进行定位布局  
55.         for (int i = 0; i < getChildCount(); i++) {  
56.             // 获取一个子元素  
57.             View child = getChildAt(i);  
58.  
59.             CustomLayoutParams clp = (CustomLayoutParams) child.getLayoutPar  
      ams();  
60.  
61.             // 通知子元素进行布局  
62.             // 此时考虑父容器内边距和子元素外边距的影响  
63.             child.layout(parentPaddingLeft + clp.leftMargin, mUtilHeight + p  
      arentPaddingTop + clp.topMargin, child.getMeasuredWidth() + parentPaddingLe  
      ft + clp.leftMargin, child.getMeasuredHeight() + mUtilHeight + parentPaddingT  
      op + clp.topMargin);  
64.  
65.             // 改变高度倍增值  
66.             mUtilHeight += child.getMeasuredHeight() + clp.topMargin + clp.b  
      ottomMargin;  
67.         }  
68.     }  
69. }
```

布局文件如下：

[[html](#)] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <com.aigestudio.customviewdemo.views.CustomLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:background="#FF597210"
5.     android:orientation="vertical" >
6.
7.     <com.aigestudio.customviewdemo.views.IconView
8.         android:id="@+id/main_pv"
9.         android:layout_width="wrap_content"
10.        android:layout_height="wrap_content"
11.        android:layout_marginBottom="10dp"
12.        android:layout_marginLeft="20dp"
13.        android:layout_marginRight="30dp"
14.        android:layout_marginTop="5dp" />
15.
16.    <Button
17.        android:layout_width="wrap_content"
18.        android:layout_height="wrap_content"
19.        android:layout_marginBottom="16dp"
20.        android:layout_marginLeft="2dp"
21.        android:layout_marginRight="8dp"
22.        android:layout_marginTop="4dp"
23.        android:text="AigeStudio" />
24.
25.    <TextView
26.        android:layout_width="wrap_content"
27.        android:layout_height="wrap_content"
28.        android:layout_marginBottom="28dp"
29.        android:layout_marginLeft="7dp"
30.        android:layout_marginRight="19dp"
31.        android:layout_marginTop="14dp"
32.        android:background="#FF166792"
33.        android:text="AigeStudio" />
34.
35. </com.aigestudio.customviewdemo.views.CustomLayout>
```

运行效果如下：



~~~~~好了好了、不讲了，View 的基本测量过程大致就是这样，如我所说测量并不是定式的过程，总会因控件而已，我们在自定义控件时要准确地测量，一定要准确，测量的结果会直接影响后面的布局定位、绘制甚至交互，所以马虎不得，你也可以看到 Android 给我们提供的 LinearLayout、FrameLayout 等布局都有极其严谨的测量逻辑，为的就是确保测量结果的准确。

本篇幅虽长，但是我们其实就讲了三点：

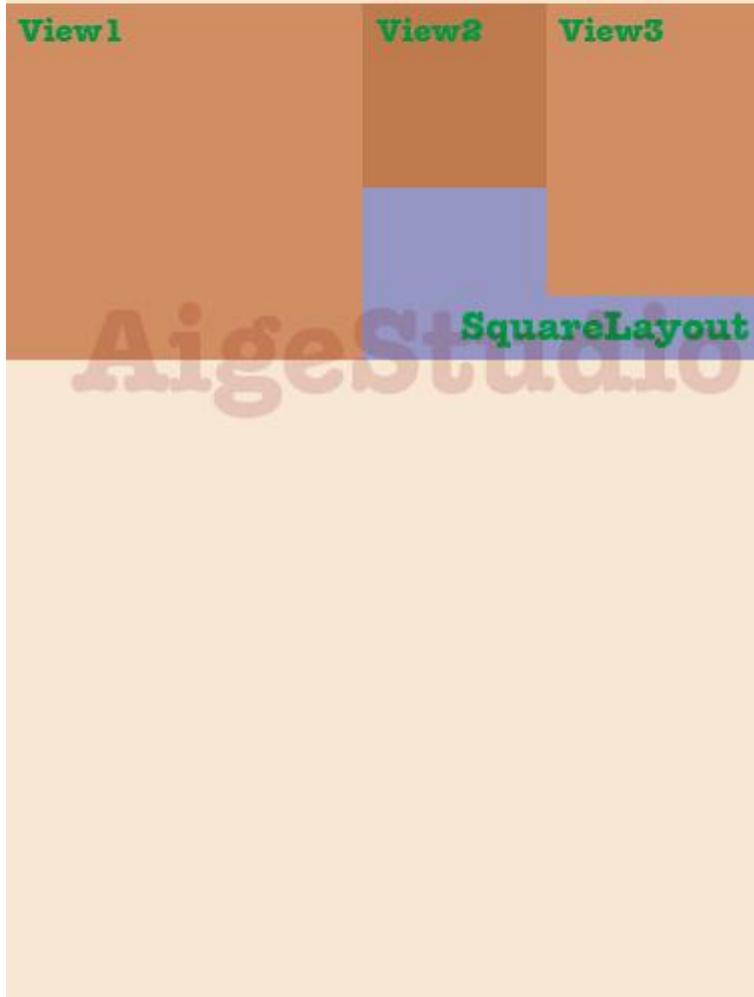
1. 一个界面窗口的元素构成
2. framework 对 View 测量的控制处理
3. View 和 ViewGroup 的简单测量

好了、不说了、实在说不动了.....到此为止&￥.....#￥.....%#￥%#￥%#%￥哦！对了，文章开头我给各位设了一个问题，不知道大家发现没有，本来说这节顺带讲了，看着篇幅太长下节再说吧.....

12. 自定义控件其实很简单(12)

又要开始鸡冻人心的一刻了有木有！有木有鸡冻！ = =通过上一节的讲解呢我们大致对 Android 测量控件有个初步的了解，而此后呢也有不少盆友 Q 小窗我问了不少问题，不过其实这些问题大多都不是问题，至于到底是不是问题呢，还要等我研究下究竟可不可以把这些问题归为问题.....稍等、我吃个药先。大多数盆友的反应是在对控件测量的具体方法还不是很了解，不过不要着急，上一节的内容就当饭前甜点，接下来我们会用一个例子来说明这一切不是问题的问题，这个例子中的控件呢我称其为 **SquareLayout**，意为方形布局（注：该例子仅作演示，实际应用意义并不大），我们将置于该布局下的所有子元素都强制变为一个正方形~~说起简单，但是如我上一节所说控件的设计要尽可能考虑到所有的可能性才能趋于完美~~但是没有绝对的完美.....在 5/12 时我曾说过不要将自己当作一个 **coder** 而要把自己看成一个 **designer**，控件是我们设计出来的而不是敲出来的，在我们 **code** 之前就该对这个控件有一个较为 **perfect** 的 **design**，考虑到控件的属性设计、行为设计、交互设计等等，这里呢我也对我们的 **SquareLayout** 做了一个简单的设计：

ViewGroup



非常简单，如上我们所说，`SquareLayout` 内部的子元素都会以正方形的形状显示，我们可以给其定义一个 `orientation` 属性来表示子元素排列方式，如上是 `orientation` 为横向时的排列方式，而下面则是纵向的排列方式：



指定了排列方式后我们的子元素就会以此为基准排列下去,但是如果子元素超出了父容器的区域怎么办呢?这时我们可以指定两种处理方式:一、不管,任由子元素被父容器的边距裁剪;二、强制被裁剪的子元素舍弃其原有布局重新布局。暂时我们先默认第一种吧。接着看,如果我们的子元素只能是横着一排或竖着一排着实单调,我们可以考虑定义两个属性控制其最大排列个数,比如纵向排列时,我们可以指定一个 `max_row` 属性,当排列的行数超过该值时自动换列:

ViewGroup

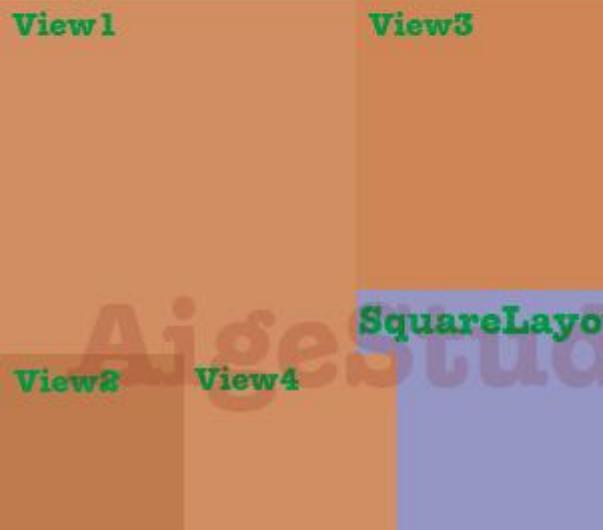
max_row = 2



当然我们也可以在子元素横向排列时为其指定 `max_column` 属性，当横向排列列数超过该值时自动换行：

ViewGroup

max_column = 2



仔细想想，当 `max_column` 为 1 时，我们的排列方式其实就是纵向的而当 `max_row` 为 1 时就是横向的，那么我们的 `orientation` 属性岂不是成了摆设？会不会呢？留给各位去想。好吧、暂时就先定义这俩属性，光这俩已经够折腾的了，来来来创建我们的布局：

[java] view plaincopyprint?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
```

```
8.     private int mMaxRow;// 最大行数
9.     private int mMaxColumn;// 最大列数
10.
11.    private int mOrientation;// 排列方向
12.
13.    public SquareLayout(Context context, AttributeSet attrs) {
14.        super(context, attrs);
15.    }
16.
17.    @Override
18.    protected void onLayout(boolean changed, int l, int t, int r, int b) {
19.    }
20. }
```

上一节我们曾说过，要让我们父容器下子元素的 `margins` 外边距能够被正确计算，我们必须重写父容器的三个相关方法并返回一个 `MarginLayoutParams` 的子类：

[java] view plain copy print?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     // 省去各种蛋疼的成员变量.....
9.
10.    // 省去构造方法.....
11.
12.    // 省去 onLayout 方法.....
13.
14.    @Override
15.    protected LayoutParams generateDefaultLayoutParams() {
16.        return new MarginLayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, V
iewGroup.LayoutParams.MATCH_PARENT);
17.    }
18.
19.    @Override
20.    protected android.view.ViewGroup.LayoutParams generateLayoutParams(android
.view.ViewGroup.LayoutParams p) {
21.        return new MarginLayoutParams(p);
22.    }
23. }
```

```
24.     @Override
25.     public android.view.ViewGroup.LayoutParams generateLayoutParams(Attribut
eSet attrs) {
26.         return new MarginLayoutParams(getContext(), attrs);
27.     }
28. }
```

这里我直接返回一个 `MarginLayoutParams` 的实例对象，因为我不需要在 `LayoutParams` 处理自己的逻辑，单纯地计算 `margins` 就没必要去实现一个自定义的 `MarginLayoutParams` 子类了，除此之外，你还可以重写 `checkLayoutParams` 方法去验证当前所使用的 `LayoutParams` 对象是否 `MarginLayoutParams` 的一个实例：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     // 省去各种蛋疼的成员变量.....
9.
10.    // 省去构造方法.....
11.
12.    // 省去 onLayout 方法.....
13.
14.    // 省去三个屁毛方法.....
15.
16.    @Override
17.    protected boolean checkLayoutParams(android.view.ViewGroup.LayoutParams
p) {
18.        return p instanceof MarginLayoutParams;
19.    }
20. }
```

然后呢我们就要开始对控件进行测量了，首先重写 `onMeasure` 方法是肯定的，那么我们就先在 `onMeasure` 中先把测量的逻辑处理了先，不过我们还是按部就班一步一步来，先把排列方式搞定：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
```

```
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     private static final int ORIENTATION_HORIZONTAL = 0, ORIENTATION_VERTICAL
9.     L = 1;// 排列方向的常量标识值
10.    private static final int DEFAULT_MAX_ROW = Integer.MAX_VALUE, DEFAULT_MA
11.    X_COLUMN = Integer.MAX_VALUE;// 最大行列默认值
12.    private int mMaxRow = DEFAULT_MAX_ROW;// 最大行数
13.    private int mMaxColumn = DEFAULT_MAX_COLUMN;// 最大列数
14.    private int mOrientation = ORIENTATION_HORIZONTAL;// 排列方向默认横向
15.    // 省去构造方法.....
16.    @SuppressLint("NewApi")
17.    @Override
18.    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
19.
20.        /*
21.         * 声明临时变量存储父容器的期望值
22.         * 该值应该等于父容器的内边距加上所有子元素的测量宽高和外边距
23.         */
24.        int parentDesireWidth = 0;
25.        int parentDesireHeight = 0;
26.
27.        // 声明临时变量存储子元素的测量状态
28.        int childMeasureState = 0;
29.
30.        /*
31.         * 如果父容器内有子元素
32.         */
33.        if (getChildCount() > 0) {
34.            /*
35.             * 那么就遍历子元素
36.             */
37.            for (int i = 0; i < getChildCount(); i++) {
38.                // 获取对应遍历下标的子元素
39.                View child = getChildAt(i);
40.
41.                /*
42.                 * 如果该子元素没有以“不占用空间”的方式隐藏则表示其需要被测量计算
43.                 */
44.            }
45.        }
46.    }
47.}
```

```
44.         */
45.         if (child.getVisibility() != View.GONE) {
46.
47.             // 测量子元素并考量其外边距
48.             measureChildWithMargins(child, widthMeasureSpec, 0, heightMeasureSpec, 0);
49.
50.             // 比较子元素测量宽高并比较取其较大值
51.             int childMeasureSize = Math.max(child.getMeasuredWidth(),
52.                 child.getMeasuredHeight());
53.
54.             // 重新封装子元素测量规格
55.             int childMeasureSpec = MeasureSpec.makeMeasureSpec(child
56.                 .MeasureSpecSize, MeasureSpec.EXACTLY);
57.
58.             // 重新测量子元素
59.             child.measure(childMeasureSpec, childMeasureSpec);
60.
61.             /*
62.             * 考量外边距计算子元素实际宽高
63.             */
64.
65.             int childActualWidth = child.getMeasuredWidth() + mlp.leftMargin +
66.                 mlp.rightMargin;
67.             int childActualHeight = child.getMeasuredHeight() + mlp.
68.                 topMargin + mlp.bottomMargin;
69.
70.             /*
71.             * 如果为横向排列
72.             */
73.
74.             if (mOrientation == ORIENTATION_HORIZONTAL) {
75.                 // 累加子元素的实际宽度
76.                 parentDesireWidth += childActualWidth;
77.
78.                 // 获取子元素中高度最大值
79.                 parentDesireHeight = Math.max(parentDesireHeight, child
80.                     .ActualHeight);
81.             }
82.
83.             /*
84.             * 如果为纵向排列
85.             */
86.             parentDesireWidth = Math.max(parentDesireWidth, child
87.                 .ActualWidth);
88.
89.             /*
90.             * 根据子元素的宽高和父元素的宽高来决定父元素的宽高
91.             */
92.             if (parentDesireWidth > parentWidth) {
93.                 parentDesireWidth = parentWidth;
94.             }
95.             if (parentDesireHeight > parentHeight) {
96.                 parentDesireHeight = parentHeight;
97.             }
98.         }
99.     }
100.
```

```
81.          */
82.          else if (mOrientation == ORIENTATION_VERTICAL) {
83.              // 累加子元素的实际高度
84.              parentDesireHeight += childActualHeight;
85.
86.              // 获取子元素中宽度最大值
87.              parentDesireWidth = Math.max(parentDesireWidth, childActualWidth);
88.          }
89.
90.          // 合并子元素的测量状态
91.          childMeasureState = combineMeasuredStates(childMeasureState,
92.              child.getMeasuredState());
93.      }
94.
95.      /*
96.       * 考量父容器内边距将其累加到期望值
97.       */
98.      parentDesireWidth += getPaddingLeft() + getPaddingRight();
99.      parentDesireHeight += getPaddingTop() + getPaddingBottom();
100.
101.     /*
102.      * 尝试比较父容器期望值与 Android 建议的最小值大小并取较大值
103.      */
104.     parentDesireWidth = Math.max(parentDesireWidth, getSuggestedMinimumWidth());
105.     parentDesireHeight = Math.max(parentDesireHeight, getSuggestedMinimumHeight());
106. }
107.
108. // 确定父容器的测量宽高
109. setMeasuredDimension(resolveSizeAndState(parentDesireWidth, widthMeasureSpec,
110.                             childMeasureState),
111.                         resolveSizeAndState(parentDesireHeight, heightMeasureSpec,
112.                             childMeasureState << MEASURED_HEIGHT_STATE_SHIFT));
113. }
114.
115. // 省去 onLayout 方法.....
116. }
```

上面代码注释很清楚，具体的我就不扯了，小窗我的童鞋有一部分问过我上一节中我在确定

测量尺寸时候使用的 `resolveSize` 方法作用（以下代码源自上一节的 `CustomLayout`）：

```
[java] view plaincopyprint?  
1.  /**
2.   *
3.   * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4.   * @since 2015/1/15
5.   *
6.   */
7. public class CustomLayout extends ViewGroup {
8.     // 省去 N 多代码
9.
10.    @Override
11.    protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
12.
13.        // 省省省.....
14.        // 设置最终测量值
15.        setMeasuredDimension(resolveSize(parentDesireWidth, widthMeasureSpec),
16.            resolveSize(parentDesireHeight, heightMeasureSpec));
17.    }
18.    // 省去 N+1 多代码
19. }
```

那么这个 `resolveSize` 方法其实是 `View` 提供给我们解算尺寸大小的一个工具方法，其具体实现在 API 11 后交由另一个方法 `resolveSizeAndState` 也就是我们这一节例子所用到的去处理：

```
[java] view plaincopyprint?  
1. public static int resolveSize(int size, int measureSpec) {
2.     return resolveSizeAndState(size, measureSpec, 0) & MEASURED_SIZE_MASK;
3. }
```

而这个 `resolveSizeAndState` 方法具体实现其实跟我们上一节开头解算 `Bitmap` 尺寸的逻辑类似：

```
[java] view plaincopyprint?  
1. public static int resolveSizeAndState(int size, int measureSpec, int childMe
asuredState) {
2.     int result = size;
```

```
3.     int specMode = MeasureSpec.getMode(measureSpec);
4.     int specSize = MeasureSpec.getSize(measureSpec);
5.     switch (specMode) {
6.         case MeasureSpec.UNSPECIFIED:
7.             result = size;
8.             break;
9.         case MeasureSpec.AT_MOST:
10.            if (specSize < size) {
11.                result = specSize | MEASURED_STATE_TOO_SMALL;
12.            } else {
13.                result = size;
14.            }
15.            break;
16.        case MeasureSpec.EXACTLY:
17.            result = specSize;
18.            break;
19.    }
20.    return result | (childMeasuredState&MEASURED_STATE_MASK);
21. }
```

是不是很类似呢？与我们不同的是这个方法多了一个 `childMeasuredState` 参数，而上面例子我们在具体测量时也引入了一个 `childMeasureState` 临时变量的计算，那么这个值的作用是什么呢？有何意义呢？说到这里不得不提 API 11 后引入的几个标识位：

public static final int MEASURED_HEIGHT_STATE_SHIFT

Bit shift of `MEASURED_STATE_MASK` to get to the height bits for functions that combine both width and height childState argument of `resolveSizeAndState(int, int, int)`.

Constant Value: 16 (0x00000010)

public static final int MEASURED_SIZE_MASK

Bits of `getMeasuredWidthAndState()` and `getMeasuredWidthAndState()` that provide the actual measured size.

Constant Value: 16777215 (0x00fffff)

public static final int MEASURED_STATE_MASK

Bits of `getMeasuredWidthAndState()` and `getMeasuredWidthAndState()` that provide the additional state bits.

Constant Value: -16777216 (0xff000000)

public static final int MEASURED_STATE_TOO_SMALL

Bit of `getMeasuredWidthAndState()` and `getMeasuredWidthAndState()` that indicates the measured size is small.

Constant Value: 16777216 (0x01000000)

这些标识位上面的代码中我们都有用到，而官方文档对其作用的说明也是模棱两可，源码里的运用也不明朗，比如说我们看其它几个与其相关的几个方法：

[java] [view plain](#) [copy](#) [print?](#)

```
1. public final int getMeasuredWidth() {
2.     return mMeasuredWidth & MEASURED_SIZE_MASK;
3. }
4. public final int getMeasuredHeight() {
5.     return mMeasuredHeight & MEASURED_SIZE_MASK;
6. }
7. public final int getMeasuredState() {
8.     return (mMeasuredWidth&MEASURED_STATE_MASK)
9.         | ((mMeasuredHeight>>MEASURED_HEIGHT_STATE_SHIFT)
10.           & (MEASURED_STATE_MASK>>MEASURED_HEIGHT_STATE_SHIFT));
11. }
```

这里大家注意 `getMeasuredWidth` 和 `getMeasuredHeight` 这两个我们用来获取控件测量宽高的方法，在其之中对其做了一个按位与的运算，然后才把这个测量值返回给我们，也就是说这个 `mMeasuredWidth` 和 `mMeasuredHeight` 里面应该还封装了些什么对吧，那么我们来看其赋值，其赋值是在 `setMeasuredDimension` 方法下进行的：

[java] view plain copy print?

```
1. protected final void setMeasuredDimension(int measuredWidth, int measuredHei  
ght) {  
2.     // 省去无关代码.....  
3.     mMeasuredWidth = measuredWidth;  
4.     mMeasuredHeight = measuredHeight;  
5.  
6.     // 省去一行代码.....  
7. }
```

也就是说当我们给控件设置最终测量尺寸时这个值就直接被赋予给了 `mMeasuredWidth` 和 `mMeasuredHeight` 这两个成员变量.....看到这里很多朋友蛋疼了，那有啥区别和意义呢？我们尝试来翻翻系统自带控件关于它的处理，其中 `TextView` 没有涉及到这个参数的应用，而 `ImageView` 里则有：

[java] view plain copy print?

```
1. @Override  
2. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
3.     // 省去海量代码.....  
4.  
5.     widthSize = resolveSizeAndState(w, widthMeasureSpec, 0);  
6.     heightSize = resolveSizeAndState(h, heightMeasureSpec, 0);  
7.  
8.     // 省去一点代码.....  
9. }
```

在 `ImageView` 的 `onMeasure` 方法中使用 `resolveSizeAndState` 再次对计算得出的宽高进行解算，而这里的第三个参数直接传的 0，也就是不作任何处理~~~~~蛋疼！真蛋疼，以前寡人也曾纠结过一段时间，后来在 `stackoverflow` 在找到两个比较靠谱的答案：

Edit: Sorry for misunderstanding your question. I've taken a look at the concept of measured state for a view, and I can only find a single documented state: `MEASURED_STATE_TOO_SMALL`. This leads me to believe that it's use may be very limited and it's existence is primarily for the purpose of future functionality or to be made use of by custom Views/ViewGroups.

The documentation for `MEASURED_STATE_TOO_SMALL` states the following:

Bit of `getMeasuredWidthAndState()` and `getMeasuredWidthAndState()` that indicates the measured size is smaller than the space the view would like to have.

This leads me to believe that the bit will be set whenever a fixed dp/px value is given to the View which is larger than the parent's width and height and therefore the View's measured width and height will have been scaled down.

大概意思就是当控件的测量尺寸比其父容器大时将会设置
MEASURED_STATE_TOO_SMALL 这个二进制值，而另一个 stackoverflow 的回答就更官方了：

The childMeasuredState is the value returned by View.getMeasuredState(). A layout will aggregate its children measured states using view.combineMeasuredStates(). Here is an example:

```
int childState = 0;

for (int i = 0; i < count; i++) {
    final View child = getChildAt(i);
    if (child.getVisibility() != GONE) {
        measureTheChild(child);
        childState = combineMeasuredStates(childState, child.getMeasuredState());
    }
}
```

In most case however you can simply pass 0 instead. The child state is currently used only to tell whether a View was measured at a smaller size than it would like to have. This information is in turn used to resize dialogs if needed. In your particular case you shouldn't worry about it.

[share](#) [edit](#)

answered Nov 30 '12 at 19:07



Romain Guy

62.3k ● 8 ● 148 ● 163

注意右下角的用户名和头像，你就知道为什么这个回答有权威性了，鄙人是他脑残粉。来我们好好翻一下 Romain 这段话的意思：“childMeasuredState 这个值呢由 View.getMeasuredState() 这个方法返回，一个布局（或者按我的说法父容器）通过 View.combineMeasuredStates() 这个方法来统计其子元素的测量状态。在大多数情况下你可以简单地只传递 0 作为参数值，而子元素状态值目前的作用只是用来告诉父容器在对其进行测量得出的测量值比它自身想要的尺寸要小，如果有必要的话一个对话框将会根据这个原因来重新校正它的尺寸。”So、可以看出，测量状态对谷歌官方而言也还算个测试性的功能，具体鄙人也没有找到很好的例证，如果大家谁找到了其具体的使用方法可以分享一下，这里我们还是就按照谷歌官方的建议依葫芦画瓢。好了这个问题就先到这里为止，我们继续看，在测量子元素尺寸时我分了两种情况：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. /*
2. * 如果为横向排列
3. */
4. if (mOrientation == ORIENTATION_HORIZONTAL) {
5.     // 累加子元素的实际宽度
6.     parentDesireWidth += childActualWidth;
7.
8.     // 获取子元素中高度最大值
```

```
9.     parentDesireHeight = Math.max(parentDesireHeight, childActualHeight);
10. }
11.
12. /*
13. * 如果为竖向排列
14. */
15. else if (mOrientation == ORIENTATION_VERTICAL) {
16.     // 累加子元素的实际高度
17.     parentDesireHeight += childActualHeight;
18.
19.     // 获取子元素中宽度最大值
20.     parentDesireWidth = Math.max(parentDesireWidth, childActualWidth);
21. }
```

如果为横/竖向排列，那么我们应该统计各个子元素的宽/高，而高/宽呢则不需要统计，我们取其最高/最宽的那个子元素的值即可，注意在上一节的处理中我们并没有这样去做哦！不知道大家发现没~~~好了，`onMeasure` 方法的逻辑就是这样，如果你觉得好长，那么恭喜你，这只是我们的第一步，尔后还有几个参数的处理~~~~~这时候你如果运行会发现什么都没有，因为 `onMeasure` 方法的作用仅仅是测量的一步，按照官方的说法，Android 对 Viewgroup 的测量由两方面构成：一是对父容器和子元素大小尺寸的测量主要体现在 `onMeasure` 方法，二是对父容器的子元素在其区域内的定位主要体现在 `onLayout` 方法。也就是说，即便我们完成了测量但没告诉儿子们该出现在哪的话也不会有任何显示效果，OK，现在我们来看看 `onLayout` 方法的逻辑处理：

[java] view plain copy print?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     private static final int ORIENTATION_HORIZONTAL = 0, ORIENTATION_VERTICAL
L = 1; // 排列方向的常量标识值
9.     private static final int DEFAULT_MAX_ROW = Integer.MAX_VALUE, DEFAULT_MA
X_COLUMN = Integer.MAX_VALUE; // 最大行列默认值
10.
11.    private int mMaxRow = DEFAULT_MAX_ROW; // 最大行数
12.    private int mMaxColumn = DEFAULT_MAX_COLUMN; // 最大列数
13.
14.    private int mOrientation = ORIENTATION_HORIZONTAL; // 排列方向默认横向
15.
16.    // 省去构造方法.....
```

```
17.  
18.    // 省去上面已经给过的 onMeasure 方法.....  
19.  
20.    @Override  
21.    protected void onLayout(boolean changed, int l, int t, int r, int b) {  
22.        /*  
23.         * 如果父容器下有子元素  
24.         */  
25.        if (getChildCount() > 0) {  
26.            // 声明临时变量存储宽高倍增值  
27.            int multi = 0;  
28.  
29.            /*  
30.             * 遍历子元素  
31.             */  
32.            for (int i = 0; i < getChildCount(); i++) {  
33.                // 获取对应遍历下标的子元素  
34.                View child = getChildAt(i);  
35.  
36.                /*  
37.                 * 如果该子元素没有以“不占用空间”的方式隐藏则表示其需要被测量计算  
38.                 */  
39.                if (child.getVisibility() != View.GONE) {  
40.                    // 获取子元素布局参数  
41.                    MarginLayoutParams mlp = (MarginLayoutParams) child.getL  
ayoutParams();  
42.  
43.                    // 获取控件尺寸  
44.                    int childActualSize = child.getMeasuredWidth();// child.  
getMeasuredHeight()  
45.  
46.                    /*  
47.                     * 如果为横向排列  
48.                     */  
49.                    if (mOrientation == ORIENTATION_HORIZONTAL) {  
50.                        // 确定子元素左上、右下坐标  
51.                        child.layout(getPaddingLeft() + mlp.leftMargin + mul  
ti, getPaddingTop() + mlp.topMargin, childActualSize + getPaddingLeft()  
+ mlp.leftMargin + multi, childActualSize +  
getPaddingTop() + mlp.topMargin);  
52.  
53.                        // 累加倍增值  
54.                        multi += childActualSize + mlp.leftMargin + mlp.righ  
tMargin;
```

```
56.             }
57.
58.             /*
59.              * 如果为竖向排列
60.              */
61.             else if (mOrientation == ORIENTATION_VERTICAL) {
62.                 // 确定子元素左上、右下坐标
63.                 child.layout(getPaddingLeft() + mlp.leftMargin, getP
   addingTop() + mlp.topMargin + multi, childActualSize + getPaddingLeft()
64.                                         + mlp.leftMargin, childActualSize + getPaddi
   ngTop() + mlp.topMargin + multi);
65.
66.                 // 累加倍增值
67.                 multi += childActualSize + mlp.topMargin + mlp.botto
   mMargin;
68.             }
69.         }
70.     }
71. }
72. }
73.
74. // 省去四个眉毛方法.....
75. }
```

比起对 `onMeasure` 方法的逻辑处理，`onLayout` 方法相对简单，主要是在对子元素 `layout` 的地方需要我们一点计算思维，也不是很复杂，哥相信你能懂，毕竟注释如此清楚，来我们尝试用一下我们的布局：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:layout_gravity="center"
5.     android:background="#ffffffff" >
6.
7.     <com.aigestudio.customviewdemo.views.SquareLayout
8.         android:layout_width="match_parent"
9.         android:layout_height="wrap_content"
10.        android:paddingLeft="5dp"
11.        android:paddingTop="12dp"
12.        android:layout_margin="5dp"
13.        android:paddingRight="7dp"
14.        android:paddingBottom="20dp"
```

```
15.        android:layout_gravity="center"
16.        android:background="#679135" >
17.
18.        <Button
19.            android:layout_width="wrap_content"
20.            android:layout_height="wrap_content"
21.            android:background="#125793"
22.            android:text="tomorrow"
23.            android:textSize="24sp"
24.            android:textStyle="bold"
25.            android:typeface="serif" />
26.
27.        <Button
28.            android:layout_width="50dp"
29.            android:layout_height="100dp"
30.            android:layout_marginBottom="5dp"
31.            android:layout_marginLeft="10dp"
32.            android:layout_marginRight="20dp"
33.            android:layout_marginTop="30dp"
34.            android:background="#495287"
35.            android:text="AigeStudio" />
36.
37.        <ImageView
38.            android:layout_width="wrap_content"
39.            android:layout_height="wrap_content"
40.            android:layout_marginBottom="50dp"
41.            android:layout_marginLeft="5dp"
42.            android:layout_marginRight="20dp"
43.            android:layout_marginTop="15dp"
44.            android:background="#976234"
45.            android:scaleType="centerCrop"
46.            android:src="@drawable/lovestory_little" />
47.
48.        <Button
49.            android:layout_width="wrap_content"
50.            android:layout_height="wrap_content"
51.            android:background="#594342"
52.            android:text="AigeStudio" />
53.
54.        <Button
55.            android:layout_width="wrap_content"
56.            android:layout_height="wrap_content"
57.            android:background="#961315"
58.            android:text="AigeStudio" />
```

```
59.      </com.aigestudio.customviewdemo.views.SquareLayout>
60.
61. </LinearLayout>
```



下面是运行后显示的效果：



将排列方式改为纵向排列：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. private int mOrientation = ORIENTATION_VERTICAL; // 排列方向默认横向
```

再来瞅瞅 ADT 的显示效果：



在运行看看：



看样子目测还是很完美，不过这只是我们伟大的第一步而已！如我多次强调，控件的测量一定要尽可能地考虑到所有因素，这样你的控件才能立于 N 次不倒的暴力测试中，现在开始我们的第二步，`max_row` 和 `max_column` 属性的计算：

[java] view plain copy print?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     private static final int ORIENTATION_HORIZONTAL = 0, ORIENTATION_VERTICAL
9.     L = 1; // 排列方向的常量标识值
10.    private static final int DEFAULT_MAX_ROW = Integer.MAX_VALUE, DEFAULT_MA
11.    X_COLUMN = Integer.MAX_VALUE; // 最大行列默认值
12.
```

```
11.     private int mMaxRow = DEFAULT_MAX_ROW;// 最大行数
12.     private int mMaxColumn = DEFAULT_MAX_COLUMN;// 最大列数
13.
14.     private int mOrientation = ORIENTATION_HORIZONTAL;// 排列方向默认横向
15.
16.     public SquareLayout(Context context, AttributeSet attrs) {
17.         super(context, attrs);
18.
19.         // 初始化最大行列数
20.         mMaxRow = mMaxColumn = 2;
21.     }
22.
23.     // 省去 onMeasure 方法.....
24.
25.     // 省去 onLayout 方法.....
26.
27.     // 省去四个眉毛方法.....
28. }
```

首先呢在构造方法内初始化我们的最大行列数，不然我们可不可能造出 Integer.MAX_VALUE 这么多的子元素~~~~~

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. // 初始化最大行列数
2. mMaxRow = mMaxColumn = 2;
```

我们的 SquareLayout 中有 5 个子元素，那么这里就暂定我们的最大行列均为 2 好了，首先来看看 onMeasure 方法的逻辑处理，变动较大我先贴代码好了：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     private static final int ORIENTATION_HORIZONTAL = 0, ORIENTATION_VERTICAL
L = 1;// 排列方向的常量标识值
9.     private static final int DEFAULT_MAX_ROW = Integer.MAX_VALUE, DEFAULT_MA
X_COLUMN = Integer.MAX_VALUE;// 最大行列默认值
10.
```

```
11.     private int mMaxRow = DEFAULT_MAX_ROW;// 最大行数
12.     private int mMaxColumn = DEFAULT_MAX_COLUMN;// 最大列数
13.
14.     private int mOrientation = ORIENTATION_HORIZONTAL;// 排列方向默认横向
15.
16.     // 省去构造方法.....
17.
18.     @SuppressLint("NewApi")
19.     @Override
20.     protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
21.
22.         /*
23.          * 声明临时变量存储父容器的期望值
24.          * 该值应该等于父容器的内边距加上所有子元素的测量宽高和外边距
25.         */
26.         int parentDesireWidth = 0;
27.         int parentDesireHeight = 0;
28.
29.         // 声明临时变量存储子元素的测量状态
30.         int childMeasureState = 0;
31.
32.         /*
33.          * 如果父容器内有子元素
34.          */
35.         if (getChildCount() > 0) {
36.             // 声明两个一维数组存储子元素宽高数据
37.             int[] childWidths = new int[getChildCount()];
38.             int[] childHeights = new int[getChildCount()];
39.
40.             /*
41.              * 那么就遍历子元素
42.              */
43.             for (int i = 0; i < getChildCount(); i++) {
44.                 // 获取对应遍历下标的子元素
45.                 View child = getChildAt(i);
46.
47.                 /*
48.                  * 如果该子元素没有以“不占用空间”的方式隐藏则表示其需要被测量计算
49.                  */
50.
51.                 // 测量子元素并考量其外边距
52.                 measureChildWithMargins(child, widthMeasureSpec, 0, heightMeasureSpec, 0);
```

```
53.                         // 比较子元素测量宽高并比较取其较大值
54.                         int childMeasureSize = Math.max(child.getMeasuredWidth()
55. , child.getMeasuredHeight());
56.
57.                         // 重新封装子元素测量规格
58.                         int childMeasureSpec = MeasureSpec.makeMeasureSpec(child
59. MeasureSize, MeasureSpec.EXACTLY);
60.
61.                         // 重新测量子元素
62.                         child.measure(childMeasureSpec, childMeasureSpec);
63.
64.                         // 获取子元素布局参数
65.                         MarginLayoutParams mlp = (MarginLayoutParams) child.getL
66. ayoutParams();
67.
68.                         /*
69.                         * 考量外边距计算子元素实际宽高并将数据存入数组
70.                         */
71.                         childWidths[i] = child.getMeasuredWidth() + mlp.leftMarg
72. in + mlp.rightMargin;
73.                         childHeights[i] = child.getMeasuredHeight() + mlp.topMar
74. gin + mlp.bottomMargin;
75.
76.                         // 合并子元素的测量状态
77.                         childMeasureState = combineMeasuredStates(childMeasureSt
78. ate, child.getMeasuredState()));
79.
80.                         }
81.                     }
82.
83.                     // 声明临时变量存储行/列宽高
84.                     int indexMultiWidth = 0, indexMultiHeight = 0;
85.
86.                     /*
87.                     * 如果为横向排列
88.                     */
89.                     if (mOrientation == ORIENTATION_HORIZONTAL) {
90.                         /*
91.                         * 如果子元素数量大于限定值则进行折行计算
92.                         */
93.                         if (getChildCount() > mMaxColumn) {
94.
95.                             // 计算产生的行数
96.                             int row = getChildCount() / mMaxColumn;
```

```
91.          // 计算余数
92.          int remainder = getChildCount() % mMaxColumn;
93.
94.
95.          // 声明临时变量存储子元素宽高数组下标值
96.          int index = 0;
97.
98.          /*
99.           * 遍历数组计算父容器期望宽高值
100.          */
101.         for (int x = 0; x < row; x++) {
102.             for (int y = 0; y < mMaxColumn; y++) {
103.                 // 单行宽度累加
104.                 indexMultiWidth += childWidths[index];
105.
106.                 // 单行高度取最大值
107.                 indexMultiHeight = Math.max(indexMultiHeight, c
108.                     hildHeights[index++]);
109.             }
110.             // 每一行遍历完后将该行宽度与上一行宽度比较取最大值
111.             parentDesireWidth = Math.max(parentDesireWidth, ind
112.                 exMultiWidth);
113.
114.             // 每一行遍历完后累加各行高度
115.             parentDesireHeight += indexMultiHeight;
116.
117.             // 重置参数
118.             indexMultiWidth = indexMultiHeight = 0;
119.         }
120.
121.         /*
122.          * 如果有余数表示有子元素未能占据一行
123.          */
124.         if (remainder != 0) {
125.             /*
126.              * 遍历剩下的这些子元素将其宽高计算到父容器期望值
127.              */
128.             for (int i = getChildCount() - remainder; i < getCh
129.                 ildCount(); i++) {
130.                 indexMultiWidth += childWidths[i];
131.                 indexMultiHeight = Math.max(indexMultiHeight, c
132.                     hildHeights[i]);
133.             }
134.         }
135.     }
136.
```

```
130.                         parentDesireWidth = Math.max(parentDesireWidth, indexMultiWidth);
131.                         parentDesireHeight += indexMultiHeight;
132.                         indexMultiWidth = indexMultiHeight = 0;
133.                     }
134.                 }
135.
136.             /*
137.             * 如果子元素数量还没有限制值大那么直接计算即可不须折行
138.             */
139.         else {
140.             for (int i = 0; i < getChildCount(); i++) {
141.                 // 累加子元素的实际高度
142.                 parentDesireHeight += childHeights[i];
143.
144.                 // 获取子元素中宽度最大值
145.                 parentDesireWidth = Math.max(parentDesireWidth, childWidths[i]);
146.             }
147.         }
148.     }
149.
150.     /*
151.     * 如果为竖向排列
152.     */
153.     else if (mOrientation == ORIENTATION_VERTICAL) {
154.         if (getChildCount() > mMaxRow) {
155.             int column = getChildCount() / mMaxRow;
156.             int remainder = getChildCount() % mMaxRow;
157.             int index = 0;
158.
159.             for (int x = 0; x < column; x++) {
160.                 for (int y = 0; y < mMaxRow; y++) {
161.                     indexMultiHeight += childHeights[index];
162.                     indexMultiWidth = Math.max(indexMultiWidth, childWidths[index++]);
163.                 }
164.                 parentDesireHeight = Math.max(parentDesireHeight, indexMultiHeight);
165.                 parentDesireWidth += indexMultiWidth;
166.                 indexMultiWidth = indexMultiHeight = 0;
167.             }
168.
169.             if (remainder != 0) {
```

```
170.                     for (int i = getChildCount() - remainder; i < getCh
    ildCount(); i++) {
171.                         indexMultiHeight += childHeights[i];
172.                         indexMultiWidth = Math.max(indexMultiHeight, ch
    ildWidths[i]);
173.                     }
174.                     parentDesireHeight = Math.max(parentDesireHeight, i
    ndexMultiHeight);
175.                     parentDesireWidth += indexMultiWidth;
176.                     indexMultiWidth = indexMultiHeight = 0;
177.                 }
178.             } else {
179.                 for (int i = 0; i < getChildCount(); i++) {
180.                     // 累加子元素的实际宽度
181.                     parentDesireWidth += childWidths[i];
182.
183.                     // 获取子元素中高度最大值
184.                     parentDesireHeight = Math.max(parentDesireHeight, c
    hildHeights[i]);
185.                 }
186.             }
187.         }
188.
189.         /*
190.          * 考量父容器内边距将其累加到期望值
191.          */
192.         parentDesireWidth += getPaddingLeft() + getPaddingRight();
193.         parentDesireHeight += getPaddingTop() + getPaddingBottom();
194.
195.         /*
196.          * 尝试比较父容器期望值与 Android 建议的最小值大小并取较大值
197.          */
198.         parentDesireWidth = Math.max(parentDesireWidth, getSuggestedMin
    imumWidth());
199.         parentDesireHeight = Math.max(parentDesireHeight, getSuggestedM
    inimumHeight());
200.     }
201.
202.     // 确定父容器的测量宽高
203.     setMeasuredDimension(resolveSizeAndState(parentDesireWidth, widthMe
    asureSpec, childMeasureState),
204.                           resolveSizeAndState(parentDesireHeight, heightMeasureSpec,
    childMeasureState << MEASURED_HEIGHT_STATE_SHIFT));
205. }
```

```
206.  
207.     // 省去 onLayout 方法.....  
208.  
209.     // 省去四个眉毛方法.....  
210. }
```

逻辑计算变动较大，首先在遍历子元素时我没有直接对横纵向排列进行计算而是先用两个数组将子元素的宽高存储起来：

[java] [view](#) [plain](#) [copy](#) [print?](#)

```
1. @SuppressLint("NewApi")  
2. @Override  
3. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  
4.     // 省去几行代码.....  
5.  
6.     /*  
7.      * 如果父容器内有子元素  
8.      */  
9.     if (getChildCount() > 0) {  
10.        // 声明两个一维数组存储子元素宽高数据  
11.        int[] childWidths = new int[getChildCount()];  
12.        int[] childHeights = new int[getChildCount()];  
13.  
14.        /*  
15.         * 那么就遍历子元素  
16.         */  
17.        for (int i = 0; i < getChildCount(); i++) {  
18.            // 省省省.....  
19.  
20.            /*  
21.             * 如果该子元素没有以“不占用空间”的方式隐藏则表示其需要被测量计算  
22.             */  
23.            if (child.getVisibility() != View.GONE) {  
24.  
25.                // 省去 N 行代码.....  
26.  
27.                /*  
28.                 * 考量外边距计算子元素实际宽高并将数据存入数组  
29.                 */  
30.                childWidths[i] = child.getMeasuredWidth() + mlp.leftMargin +  
31.                               mlp.rightMargin;  
31.                childHeights[i] = child.getMeasuredHeight() + mlp.topMargin  
+ mlp.bottomMargin;
```

```
32.                     // 省去一行代码.....
33.                 }
34.             }
35.         }
36.
37.         // 声明临时变量存储行/列宽高
38.         int indexMultiWidth = 0, indexMultiHeight = 0;
39.
40.         // 省去无数行代码.....
41.     }
42.
43.     // 省去一行代码.....
44. }
```

然后上面还声明两个临时变量 `indexMultiWidth` 和 `indexMultiHeight` 用来分别存储单行/列的宽高并将该行计算后的结果累加到父容器的期望值，这里我们就看看横向排列的逻辑：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 如果为横向排列
3. */
4. if (mOrientation == ORIENTATION_HORIZONTAL) {
5.     /*
6.      * 如果子元素数量大于限定值则进行折行计算
7.      */
8.     if (getChildCount() > mMaxColumn) {
9.
10.        // 计算产生的行数
11.        int row = getChildCount() / mMaxColumn;
12.
13.        // 计算余数
14.        int remainder = getChildCount() % mMaxColumn;
15.
16.        // 声明临时变量存储子元素宽高数组下标值
17.        int index = 0;
18.
19.        /*
20.         * 遍历数组计算父容器期望宽高值
21.         */
22.        for (int x = 0; x < row; x++) {
23.            for (int y = 0; y < mMaxColumn; y++) {
24.                // 单行宽度累加
25.                indexMultiWidth += childWidths[index];
```

```
26.          // 单行高度取最大值
27.          indexMultiHeight = Math.max(indexMultiHeight, childHeights[i
   index++]);
28.      }
29.      // 每一行遍历完后将该行宽度与上一行宽度比较取最大值
30.      parentDesireWidth = Math.max(parentDesireWidth, indexMultiWidth)
   ;
31.      parentDesireHeight += indexMultiHeight;
32.      // 每一行遍历完后累加各行高度
33.      parentDesireHeight += indexMultiHeight;
34.      // 重置参数
35.      indexMultiWidth = indexMultiHeight = 0;
36.  }
37.  /*
38.  * 如果有余数表示有子元素未能占据一行
39.  */
40.  if (remainder != 0) {
41.      /*
42.      * 遍历剩下的这些子元素将其宽高计算到父容器期望值
43.      */
44.      for (int i = getChildCount() - remainder; i < getChildCount(); i
   ++
   ) {
45.          indexMultiWidth += childWidths[i];
46.          indexMultiHeight = Math.max(indexMultiHeight, childHeights[i
   ]);
47.      }
48.      parentDesireWidth = Math.max(parentDesireWidth, indexMultiWidth)
   ;
49.      parentDesireHeight += indexMultiHeight;
50.      indexMultiWidth = indexMultiHeight = 0;
51.  }
52.  parentDesireHeight += indexMultiHeight;
53.  indexMultiWidth = indexMultiHeight = 0;
54. }
55. /*
56. * 如果子元素数量还没有限制值大那么直接计算即可不须折行
57. */
58. else {
59.     for (int i = 0; i < getChildCount(); i++) {
60.         // 累加子元素的实际高度
61.         parentDesireHeight += childHeights[i];
62.     }
63. }
64.
```

```
65.         // 获取子元素中宽度最大值
66.         parentDesireWidth = Math.max(parentDesireWidth, childWidths[i]);
67.     }
68. }
69. }
```

计算我分了两种情况，子元素数量如果小于我们的限定值，例如我们布局下只有 2 个子元素，而我们的限定值为 3，这时候就没必要计算折行，而另一种情况则是子元素数量大于我们的限定值，例如我们的布局下有 7 个子元素而我们的限定值为 3，这时当我们横向排列到第三个子元素后就得折行了，在新的一行开始排列，在这种情况下，我们先计算了能被整除的子元素数：例如 $7/3$ 为 2 余 1，也就意味着我们此时能排满的只有两行，而多出来的那一行只有一个子元素，分别计算两种情况累加结果就 OK 了。纵向排列类似不说了，这里我逻辑比较臃肿，但是可以让大家很好理解，如果你 Math 好可以简化很多逻辑，不说了，既然 `onMeasure` 方法改动了，那么我们的 `onLayout` 方法也得跟上时代的步伐才行：

[java] view plain copy print?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     private static final int ORIENTATION_HORIZONTAL = 0, ORIENTATION_VERTICAL
L = 1; // 排列方向的常量标识值
9.     private static final int DEFAULT_MAX_ROW = Integer.MAX_VALUE, DEFAULT_MA
X_COLUMN = Integer.MAX_VALUE; // 最大行列默认值
10.
11.    private int mMaxRow = DEFAULT_MAX_ROW; // 最大行数
12.    private int mMaxColumn = DEFAULT_MAX_COLUMN; // 最大列数
13.
14.    private int mOrientation = ORIENTATION_HORIZONTAL; // 排列方向默认横向
15.
16.    // 省去构造方法.....
17.
18.    // 省去上面已经给过的 onMeasure 方法.....
19.
20.    @Override
21.    protected void onLayout(boolean changed, int l, int t, int r, int b) {
22.        /*
23.         * 如果父容器下有子元素
24.         */
```



```
67.                                     child.layout(getPaddingLeft() + mlp.leftMargin
    in + indexMultiWidth, getPaddingTop() + mlp.topMargin + indexMultiHeight,
68.                                         childActualSize + getPaddingLeft() +
    mlp.leftMargin + indexMultiWidth, childActualSize + getPaddingTop()
69.                                         + mlp.topMargin + indexMulti
    Height);
70.                                     indexMultiWidth += childActualSize + mlp.lef
    tMargin + mlp.rightMargin;
71.                                     tempHeight = Math.max(tempHeight, childActua
    lSize) + mlp.topMargin + mlp.bottomMargin;
72.
73.                                     /*
74. * 如果下一次遍历到的子元素下标值大于限定值
75. */
76.         if (i + 1 >= mMaxColumn * indexMulti) {
77.             // 那么累加高度到高度倍增值
78.             indexMultiHeight += tempHeight;
79.
80.             // 重置宽度倍增值
81.             indexMultiWidth = 0;
82.
83.             // 增加指数倍增值
84.             indexMulti++;
85.         }
86.     }
87. } else {
88.     // 确定子元素左上、右下坐标
89.     child.layout(getPaddingLeft() + mlp.leftMargin +
    multi, getPaddingTop() + mlp.topMargin, childActualSize
90.                 + getPaddingLeft() + mlp.leftMargin + mu
    lti, childActualSize + getPaddingTop() + mlp.topMargin);
91.
92.     // 累加倍增值
93.     multi += childActualSize + mlp.leftMargin + mlp.
    rightMargin;
94. }
95. }
96.
97. /*
98. * 如果为竖向排列
99. */
100. else if (mOrientation == ORIENTATION_VERTICAL) {
101.     if (getChildCount() > mMaxRow) {
102.         if (i < mMaxRow * indexMulti) {
```

```

103.                     child.layout(getPaddingLeft() + mlp.leftMargin
104.                         + indexMultiWidth, getPaddingTop() + mlp.topMargin + indexMultiHeight,
105.                                         childActualSize + getPaddingLeft()
106.                                         + mlp.leftMargin + indexMultiWidth, childActualSize + getPaddingTop()
107.                                         + mlp.topMargin + indexMultiHeight);
108.                     indexMultiHeight += childActualSize + mlp.topMargin
109.                     + mlp.bottomMargin;
110.                     tempWidth = Math.max(tempWidth, childActualSize)
111.                     + mlp.leftMargin + mlp.rightMargin;
112.                     if (i + 1 >= mMaxRow * indexMulti) {
113.                         indexMultiWidth += tempWidth;
114.                         indexMultiHeight = 0;
115.                         indexMulti++;
116.                     }
117.                 }
118.             }
119.             // 确定子元素左上、右下坐标
120.             child.layout(getPaddingLeft() + mlp.leftMargin,
121.                         getPaddingTop() + mlp.topMargin + multi, childActualSize
122.                                         + getPaddingLeft() + mlp.leftMargin, ch
123.                                         idActualSize + getPaddingTop() + mlp.topMargin + multi);
124.         }
125.     }
126. }
127.
128. // 省去四个眉毛方法.....
129. }

```

`onLayout` 方法就不具体说了，其实现要比 `onMeasure` 方法简单，我们稍微更改下布局文件尽可能地测试多种情况：

[[html](#)] [view](#) [plain](#) [copy](#) [print](#)?

```

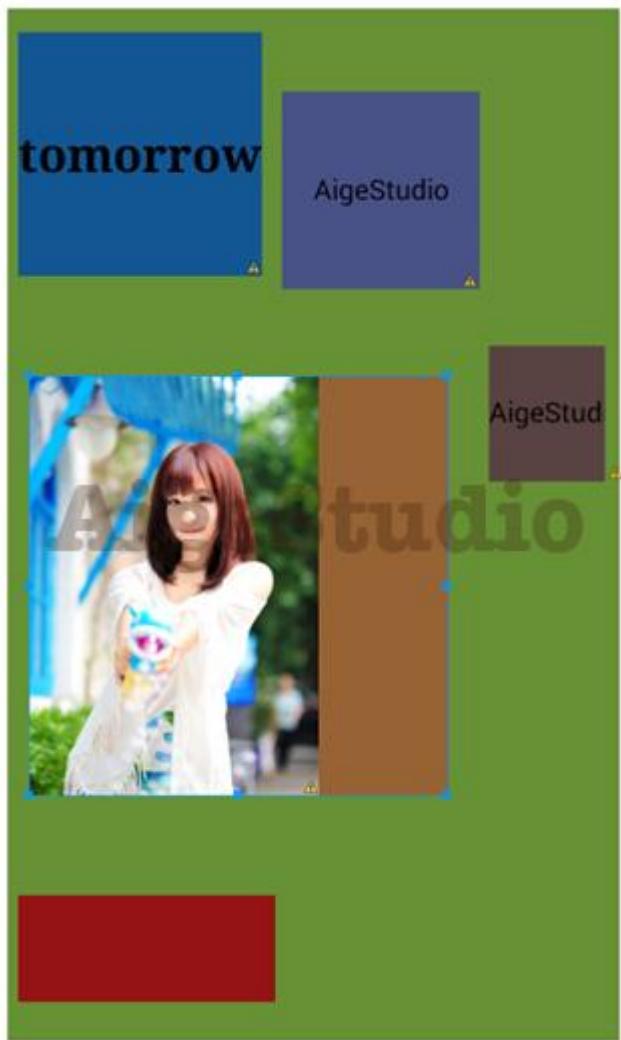
1.  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      android:layout_width="match_parent"
3.      android:layout_height="match_parent"

```

```
4.      android:layout_gravity="center"
5.      android:background="#ffffff" >
6.
7.      <com.aigestudio.customviewdemo.views.SquareLayout
8.          android:layout_width="wrap_content"
9.          android:layout_height="wrap_content"
10.         android:layout_gravity="center"
11.         android:layout_margin="5dp"
12.         android:background="#679135"
13.         android:paddingBottom="20dp"
14.         android:paddingLeft="5dp"
15.         android:paddingRight="7dp"
16.         android:paddingTop="12dp" >
17.
18.         <Button
19.             android:layout_width="wrap_content"
20.             android:layout_height="wrap_content"
21.             android:background="#125793"
22.             android:text="tomorrow"
23.             android:textSize="24sp"
24.             android:textStyle="bold"
25.             android:typeface="serif" />
26.
27.         <Button
28.             android:layout_width="50dp"
29.             android:layout_height="100dp"
30.             android:layout_marginBottom="5dp"
31.             android:layout_marginLeft="10dp"
32.             android:layout_marginRight="20dp"
33.             android:layout_marginTop="30dp"
34.             android:background="#495287"
35.             android:text="AigeStudio" />
36.
37.         <LinearLayout
38.             android:layout_width="wrap_content"
39.             android:layout_height="wrap_content"
40.             android:layout_marginBottom="50dp"
41.             android:layout_marginLeft="5dp"
42.             android:layout_marginRight="20dp"
43.             android:layout_marginTop="15dp"
44.             android:background="#976234" >
45.
46.         <ImageView
47.             android:layout_width="wrap_content"
```

```
48.         android:layout_height="wrap_content"
49.         android:scaleType="centerCrop"
50.         android:src="@drawable/lovestory_little" />
51.     </LinearLayout>
52.
53.     <Button
54.         android:layout_width="wrap_content"
55.         android:layout_height="wrap_content"
56.         android:background="#594342"
57.         android:text="AigeStudio" />
58.
59.     <Button
60.         android:layout_width="wrap_content"
61.         android:layout_height="wrap_content"
62.         android:background="#961315"
63.         android:text="Welcome AigeStudio" />
64.     </com.aigestudio.customviewdemo.views.SquareLayout>
65.
66. </LinearLayout>
```

下面看看 ADT 中的直接显示效果:



运行后的显示效果：



换成纵向排列看看：



运行后的效果：



尝试更改下纵向排列时的限制值：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print?](#)

```
1. // 初始化最大行列数  
2. mMaxRow = 2;  
3. mMaxColumn =3;
```

直接看运行效果：



表示暂时木有发现什么大问题，OK，这两个属性值的实现就到这里，虽然只有两个属性值`= =`TMD实在是菊紧啊，可想而知`LinearLayout`等布局这么多属性控制是有多蛋疼了么，不过如我文章开头所说，我们的这个自定义布局实用意义不大，主要还是给大家演示了解下自定义布局是多么蛋疼、啊不……是由多么复杂，像系统自带的那些布局控件都是经过N多`update`版本才有今天，即便如此，依然还有很多`BUG`，不过大多不会影响实际使用我们也可以很好地解决，所以，再次强调、控件的测量是一个极为严谨缜密的过程，稍有不慎你的控件便到处都会是说不出的`BUG~~~~~`上一节我们为了能让我们的自定义布局能对外边距进行计算，我们定义了一个内部类`LayoutParams`继承于`MarginLayoutParams`但是其中什么也没做，而这一节呢我们没有定义这么一个内部类而是直接返回`MarginLayoutParams`的实例，我们之所以能从布局参数中获取到外边距的属性值，比如：

```
[java] view plain copy print?
```

```
1. // 获取子元素布局参数  
2. MarginLayoutParams mlp = (MarginLayoutParams) child.getLayoutParams();
```

然后各种

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. mlp.leftMargin  
2. mlp.topMargin  
3. mlp.rightMargin  
4. mlp.bottomMargin
```

是因为在 `MarginLayoutParams` 中已经为我们定义好了这些参数，具体代码就不贴了，如果我们定义了自己的布局，我们也可以去定义自己的布局参数，比如我们在其中定义子元素在布局中的对其方式：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**  
2. *  
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}  
4. * @since 2015/1/23  
5. *  
6. */  
7. public class SquareLayout extends ViewGroup {  
8.     // 省去无数代码.....  
9.  
10.    public static class LayoutParams extends MarginLayoutParams {  
11.        public int mGravity;// 对齐方式  
12.  
13.        public LayoutParams(MarginLayoutParams source) {  
14.            super(source);  
15.        }  
16.  
17.        public LayoutParams(android.view.ViewGroup.LayoutParams source) {  
18.            super(source);  
19.        }  
20.  
21.        public LayoutParams(Context c, AttributeSet attrs) {  
22.            super(c, attrs);  
23.        }  
24.  
25.        public LayoutParams(int width, int height) {  
26.            super(width, height);  
27.        }  
28.    }  
29.}
```

然后呢我们就要修改那四个屁毛方法返回我们自己定义的 LayoutParams:

```
[java] view plaincopyprint?  
1.  /**
2.   *
3.   * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4.   * @since 2015/1/23
5.   *
6.   */
7. public class SquareLayout extends ViewGroup {
8.     // 省去无数代码.....
9.
10.    @Override
11.    protected LayoutParams generateDefaultLayoutParams() {
12.        return new LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.MATCH_PARENT);
13.    }
14.
15.    @Override
16.    protected android.view.ViewGroup.LayoutParams generateLayoutParams(android.view.ViewGroup.LayoutParams p) {
17.        return new LayoutParams(p);
18.    }
19.
20.    @Override
21.    public android.view.ViewGroup.LayoutParams generateLayoutParams(AttributeSet attrs) {
22.        return new LayoutParams(getContext(), attrs);
23.    }
24.
25.    @Override
26.    protected boolean checkLayoutParams(android.view.ViewGroup.LayoutParams p) {
27.        return p instanceof LayoutParams;
28.    }
29.
30.    // 省去 LayoutParams 的定义.....
31. }
```

然后你就可以通过其获取这个对其方式的值:

```
[java] view plaincopyprint?
```

```
1. // 获取子元素布局参数
2. LayoutParams mlp = (LayoutParams) child.getLayoutParams();
3. if (mlp.mGravity == xxxxxxx) {
4.     .....
5. }
```

用法跟 margin 类似，那么我们如何为该变量赋值呢？方法多种多样，可以写死可以直接调用赋值，这里我们来看另外的一种方式：通过 xml 在布局文件中直接设置其属性值，我们在使用 xml 进行布局时经常会使用这样的方式指定属性值：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. android:layout_width="wrap_content"
2. android:layout_height="wrap_content"
3. android:background="#125793"
4. android:text="tomorrow"
5. android:textSize="24sp"
6. android:textStyle="bold"
7. android:typeface="serif"
```

使用起来灰常方便，而这里我们也可以自定义属于自己的 xml 属性，方法非常非常简单，首先需要我们在 declare-styleable 标签下定义我们的各类属性：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <!-- http://blog.csdn.net/aigestudio -->
2. <declare-styleable name="SquareLayout">
3.     <attr name="my_gravity" format="enum">
4.         <enum name="left" value="0" />
5.         <enum name="right" value="1" />
6.         <enum name="center" value="2" />
7.         <enum name="top" value="3" />
8.         <enum name="bottom" value="4" />
9.     </attr>
10. </declare-styleable>
```

一般情况下，declare-styleable 的定义存放在 values/attr.xml 文件中，属性定义好了我们就该在布局中使用这些属性，使用方法也很简单，比如我们在 SquareLayout 的 Button 中应用 my_gravity 属性：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. <Button
```

```
2.     xmlns:aigestudio="http://schemas.android.com/apk/res/com.aigestudio.cust  
omviewdemo"  
3.     aigestudio:my_gravity="left" />
```

在使用自定义属性前声明我们包内的命名空间即可，你可以直接写在布局文件的根布局下，命名空间的声明有两种写法，上面是其一，其格式如下：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. xmlns:你想要的名字="http://schemas.android.com/apk/res/完整包名"
```

第二种方式如果你是用的是 Studio，IDE 则会提示你使用该方式：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. xmlns:你想要的名字="http://schemas.android.com/apk/res-auto"
```

都可以，最后就是从 xml 中获取这些属性了，我们可以直接简单地通过带有 AttributeSet 对象的构造方法来获取：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**  
2. *  
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}  
4. * @since 2015/1/23  
5. *  
6. */  
7. public class SquareLayout extends ViewGroup {  
8.     // 省去无数代码.....  
9.  
10.    public static class LayoutParams extends MarginLayoutParams {  
11.        public int mGravity;// 对齐方式  
12.  
13.        public LayoutParams(MarginLayoutParams source) {  
14.            super(source);  
15.        }  
16.  
17.        public LayoutParams(android.view.ViewGroup.LayoutParams source) {  
18.            super(source);  
19.        }  
20.  
21.        public LayoutParams(Context c, AttributeSet attrs) {  
22.            super(c, attrs);
```

```
23.  
24.        /*  
25.         * 获取 xml 对应属性  
26.         */  
27.        TypedArray a = c.obtainStyledAttributes(attrs, R.styleable.SquareLayout);  
28.        mGravity = a.getInt(R.styleable.SquareLayout_my_gravity, 0);  
29.    }  
30.  
31.    public LayoutParams(int width, int height) {  
32.        super(width, height);  
33.    }  
34.}  
35.}
```

通过 Context 的 obtainStyledAttributes 方法注入 AttributeSet 对象和我们资源文件中定义的 declare-styleable 属性获取一个 TypedArray 对象, 我们通过这个 TypedArray 对象各种相应的方法来获取参数值, 本来呢我之前写了很长的篇幅来给大家介绍这其中的过程, 后来发现实在太繁琐太多干脆删了重写旨在教会大家如何用即可。Android 支持如下十种不同类型的属性定义:

[html] view plain copy print?

```
1. <!-- http://blog.csdn.net/aigestudio -->  
2. <declare-styleable name="AttrView">  
3.     <!-- 引用资源 -->  
4.     <attr name="image" format="reference" />  
5.     <!-- 颜色 -->  
6.     <attr name="text_color" format="color" />  
7.     <!-- 布尔值 -->  
8.     <attr name="text_display" format="boolean" />  
9.     <!-- 尺寸大小 -->  
10.    <attr name="temp1" format="dimension" />  
11.    <!-- 浮点值 -->  
12.    <attr name="temp2" format="float" />  
13.    <!-- 整型值 -->  
14.    <attr name="temp3" format="integer" />  
15.    <!-- 字符串 -->  
16.    <attr name="text" format="string" />  
17.    <!-- 百分比 -->  
18.    <attr name="alpha" format="fraction" />  
19.    <!-- 枚举 -->  
20.    <attr name="text_align" format="integer">  
21.        <enum name="left" value="0" />
```

```
22.          <enum name="right" value="1" />
23.          <enum name="center" value="2" />
24.      </attr>
25.      <!-- 位运算 -->
26.      <attr name="text_optimize" format="integer">
27.          <flag name="anti" value="0x001" />
28.          <flag name="dither" value="0x002" />
29.          <flag name="linear" value="0x004" />
30.      </attr>
31. </declare-styleable>
```

`name` 都是我乱取的不要在意，主要看后面的 `format`，这些类型都很好理解，它们在 `TypedArray` 中都有各种对应或重载的方法，比如获取 `color` 的 `getColor` 方法，上面我们获取 `int` 的 `getInt` 等等，这里对大家来说比较新颖的是 `fraction` 百分比这个类型，其在 `TypedArray` 的对应方法如下：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. getFraction(int index, int base, int pbase, float defValue)
```

第一个参数很好理解表示我们定义的属性资源 ID，最后一个参数呢也和前面的 `getInt` 类似，主要是这第二、三个参数，其作用是分开来的，当我们在 `xml` 中使用百分比属性时有两种写法，一种是标准的 `10%` 而另一种是带 `p` 的 `10%p`：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. aigestudio:alpha="10%"
2. aigestudio:alpha="10%p"
```

当属性值为 `10%` 的时候 `base` 参数起作用，我们此时获取的参数值就等于(`10% * base`)，而 `pbase` 参数则无效，同理当属性值为 `10%p` 时参数值就等于(`10% * pbase`)而 `base` 无效，Just it。还有两个比较类似的类型：枚举和位运算，这两个类型也很好理解，枚举嘛就是从众多的选项中选一个，而位运算则可以选多个并通过“|”组合各种结果：

[html] [view](#) [plain](#) [copy](#) [print](#)?

```
1. aigestudio:text_optimize="anti|dither"
```

这种写法相信大家也很常见，比如 `layout_gravity` 属性就可以以类似的方式多选，这种方式有一个好处就是我们不用在属性声明中定义太多的值，上面的 `text_optimize` 属性只有三个对应值，但是在 `code` 中我们可以以位运算的方式组合这三个参数值：

[java] [view](#) [plain](#) [copy](#) [print](#)?

```
1. /*
2.  * 画笔优化的标识位们
3. */
4. private static final int OPTIMIZE_ANTI = 0x001, OPTIMIZE_DITHER = 0x002, OPT
    IMIZE_LINEAR = 0x004, OPTIMIZE_ANTI_DITHER = 0x003, OPTIMIZE_ANTI_LINEAR = 0
    x005, OPTIMIZE_DITHER_LINEAR = 0x006, OPTIMIZE_ALL = 0x007;
```

通过三个参数值的位运算我们实质上就得到了 7 种不同的结果，Just it。xml 属性值的定义不难不多用几次就会就不多说了，上面呢我们通过自定义的属性 `mGravity` 来尝试定义子元素相对于父容器的对其方式，而事实上 Android 提供给我们一个简便的方法去计算这玩意，Android 定义了 `Gravity` 类来实现我们对对其方式的计算，其中定义了大量的常量值定义不同对其方式，比如什么左对齐、右对齐、水平居中乱七八糟的等等，也提供了多个方法来实现计算，使用方式呢也不难，比如上面的布局参数我们换成如下方式：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. /**
2. *
3. * @author AigeStudio {@link http://blog.csdn.net/aigestudio}
4. * @since 2015/1/23
5. *
6. */
7. public class SquareLayout extends ViewGroup {
8.     // 省去无数代码.....
9.
10.    public static class LayoutParams extends MarginLayoutParams {
11.        public int mGravity = Gravity.LEFT | Gravity.RIGHT;// 对齐方式
12.
13.        // 省去没变的代码.....
14.    }
15.}
```

而在我们的 xml 属性定义中则可以直接使用 `android:layout_gravity` 这样的 name 而无需定义类型值：

[\[html\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. <declare-styleable name="SquareLayout">
2.     <attr name="android:layout_gravity" />
3. </declare-styleable>
```

这样则表示我们的属性使用的 Android 自带的标签，之后我们只需根据布局文件中 `layout_gravity` 属性的值调用 `Gravity` 类下的方法去计算对齐方式则可，`Gravity` 类下的方法很好用，为什么这么说呢？因为其可以说是无关布局的，拿最简单的一个来说：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. public static void apply(int gravity, int w, int h, Rect container, Rect out  
Rect)
```

第一个参数表示我们的对其方式值，第二三个参数呢则表示我们要对齐的元素，这里通俗地说就是我们父容器下的子元素，而 `container` 参数表示的则是我们父容器的矩形区域，最后一个参数是接收计算后子元素位置区域的矩形对象，随便 `new` 个传进去就行，可见 `apply` 方式是根据矩形区域来计算对其方式的，所以说非常好用，我们只需在 `onLayout` 方法中确定出父容器的矩形区域就可以轻松地计算出子元素根据对其方式出现在父容器中的矩形区域，这一个过程留给大家自行尝试，我就不多说了，TMD 说的太多，又忘了上一节的那个问题了、禽!!!! 好吧，下一节再说那个问题，哦！对了，还有一个擦边球的东西忘了讲，在 Android 很多的布局控件中都会重写这么一个方法：

[\[java\]](#) [view](#) [plain](#) [copy](#) [print](#)?

```
1. @Override  
2. public boolean shouldDelayChildPressedState() {  
3.     return false;  
4. }
```

并且都会一致地返回 `false`，其作用是告诉 framework 我们当前的布局不是一个滚动的布局，我们这里的自定义布局控件也重写了该方法~~~好了，不讲了，这节就到此为止，接下来就是下一节，接踵而至 in two days~~~~~thx