第 1 章

绘图基础

迷茫，本就是青春该有的样子，但不要让未来的你讨厌现在的自己。

本章作为开篇第 1 章，主要讲解有关自定义控件系列的一些基础知识，是后续各章节的根基。

1.1 基本图形绘制

1.1.1 概述

我们平时画图需要两个工具：纸和笔。在 Android 中，Paint 类就是画笔，而 Canvas 类就是纸，在这里叫作画布。

所以，凡是跟画笔设置相关的，比如画笔大小、粗细、画笔颜色、透明度、字体的样式等，都在 Paint 类里设置；同样，凡是要画出成品的东西，比如圆形、矩形、文字等，都调用 Canvas 类里的函数生成。

下面通过一个自定义控件的例子来看一下如何生成自定义控件，以及 Paint 和 Canvas 类的用法。

（1）新建一个工程，然后写一个类派生自 View。

```
package com.harvic.PaintBasis;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.view.View;
public class BasisView extends View {
    public BasisView(Context context) {
        super(context);
    }
```

```
    public BasisView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public BasisView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

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

        //设置画笔的基本属性
        Paint paint=new Paint();
        paint.setColor(Color.RED);              //设置画笔颜色
        paint.setStyle(Paint.Style.STROKE);     //设置填充样式
        paint.setStrokeWidth(50);               //设置画笔宽度

        //画圆
        canvas.drawCircle(190, 200, 150, paint);
    }
}
```

代码很简单，首先，写一个类派生自 View。派生自 View 表示当前是一个自定义的控件，类似 Button、TextView 这些控件都是派生自 View 的。如果我们想像 LinearLayout、RelativeLayout 这样生成一个容器，则需要派生自 ViewGroup。有关 ViewGroup 的知识，我们会在后面的章节中讲述。

其次，重写 onDraw(Canvas canvas)函数。可以看到，在该函数中，入参是一个 Canvas 对象，也就是当前控件的画布，所以我们只要调用 Canvas 的绘图函数，效果就可以直接显示在控件上了。

在 onDraw(Canvas canvas)函数中，我们设置了画笔的基本属性。

```
Paint paint=new Paint();
paint.setColor(Color.RED);              //设置画笔颜色
paint.setStyle(Paint.Style.STROKE);     //设置填充样式
paint.setStrokeWidth(50);               //设置画笔宽度
```

在这里，我们将画笔设置成红色，填充样式为描边，并且将画笔的宽度设置为 50px（有关这些属性的具体含义，会在后面一一讲述）。

最后，我们利用 `canvas.drawCircle(190, 200, 150, paint);`语句画了一个圆。需要注意的是，画圆所用的画笔就是我们在这里指定的。

（2）使用自定义控件。

我们可以直接在主布局中使用自定义控件（main.xml）。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:orientation="vertical"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent">
```

```
    <com.harvic.PaintBasis.BasisView
          android:layout_width="match_parent"
          android:layout_height="match_parent"/>
</LinearLayout>
```

可以看到，在 XML 中使用自定义控件时，需要使用完整的包名加类包的方式来引入。注意：这里的布局方式使用的全屏方式。后面会讲到如何给自定义控件使用 wrap_content 属性，目前我们全屏显示控件即可。

效果如下图所示。

从这里可以看到，只需要先创建一个派生自 View 的类，再重新在 onDraw()函数中设置 Paint 并调用 Canvas 的一些绘图函数，就可以画出我们想要的图形。由此看来，自定义控件并不复杂。下面我们分别来看如何设置画笔，以及 Canvas 中常用的一些绘图函数。

1.1.2　画笔的基本设置

我们初步讲一下 1.1.1 节的示例中所涉及的几个函数。

1．setAntiAlias()

该函数的具体声明如下：

```
void setAntiAlias(boolean aa)
```

表示是否打开抗锯齿功能。抗锯齿是依赖算法的，一般在绘制不规则的图形时使用，比如圆形、文字等。在绘制棱角分明的图像时，比如一个矩形、一张位图，是不需要打开抗锯齿功能的。

下面绘制一个实体圆形，抗锯齿功能分别在打开和关闭的情况下效果如下图所示。

很明显，在打开抗锯齿功能的情况下，所绘图像可以产生平滑的边缘。

其中，打开抗锯齿功能的代码如下：

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

    Paint paint=new Paint();
    paint.setColor(Color.RED);            //设置画笔颜色
    paint.setStyle(Paint.Style.FILL);     //设置填充样式
```

```
        paint.setAntiAlias(true);              //打开抗锯齿功能
        paint.setStrokeWidth(50);              //设置画笔宽度
        //画圆
        canvas.drawCircle(190, 200, 150, paint);
    }
```

2．setColor()

该函数的作用是设置画笔颜色，完整的函数声明如下：

```
void setColor(int color)
```

我们知道，一个颜色值是由红、绿、蓝三色合成出来的，所以，参数 color 只能取 8 位的 0xAARRGGBB 样式颜色值。

其中：

- A 代表透明度（Alpha），取值范围是 0～255（对应十六进制的 0x00～0xFF），取值越小，透明度越高，图像也就越透明。当取 0 时，图像完全不可见。
- R 代表红色值（Red），取值范围是 0～255（对应十六进制的 0x00～0xFF），取值越小，红色越少。当取 0 时，表示红色完全不可见；当取 255 时，红色完全显示。
- G 代表绿色值（Green），取值范围是 0～255（对应十六进制的 0x00～0xFF），取值越小，绿色越少。当取 0 时，表示绿色完全不可见；当取 255 时，绿色完全显示。
- B 代表蓝色值（Blue），取值范围是 0～255（对应十六进制的 0x00～0xFF），取值越小，蓝色越少。当取 0 时，表示蓝色完全不可见；当取 255 时，蓝色完全显示。

比如 0xFFFF0000 就表示大红色。因为透明度是 255，表示完全不透明，红色取全量值 255，其他色值全取 0，表示颜色中只有红色；当然，如果我们不需要那么红，则可以适当减少红色值，比如 0xFF0F0000 就会显示弱红色。当表示黄色时，由于黄色是由红色和绿色合成的，所以 0xFFFFFF00 就表示纯黄色。当然，如果我们需要让黄色带有一部分透明度，以便显示出所画图像底层图像，则可以适当减少透明度值，比如 0xABFFFF00；当透明度减少到 0 时，任何颜色都是不可见的，也就是图像变成了全透明，比如 0x00FFFFFF，虽然有颜色值，但由于透明度是 0，所以整个颜色是不可见的。

其实，除手动组合颜色的方法以外，系统还提供了一个专门用来解析颜色的类——Color（有关 Color 类的使用，我们稍后将在本章中提及）。

下面绘制一大一小两个圆，并且将这两个圆叠加起来，上方的圆半透明，代码如下：

```
Paint paint=new Paint();
paint.setColor(0xFFFF0000);
paint.setStyle(Paint.Style.FILL);
paint.setStrokeWidth(50);
canvas.drawCircle(190, 200, 150, paint);

paint.setColor(0x7EFFFF00);
canvas.drawCircle(190, 200, 100, paint);
```

这里绘制了两个圆，第一个圆的颜色值是 0xFFFF0000，即不透明的红色，半径取 150px；第二个圆的颜色值是 0x7EFFFF00，即半透明的黄色，半径取 100px。效果如下图所示。

扫码看彩色图

3．setStyle()

完整的函数声明如下：

```
void setStyle(Style style)
```

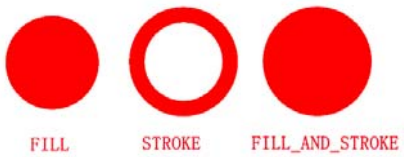
该函数用于设置填充样式，对于文字和几何图形都有效。style 的取值如下。

- Paint.Style.FILL：仅填充内部。
- Paint.Style.FILL_AND_STROKE：填充内部和描边。
- Paint.Style.STROKE：仅描边。

设置填充内部及描边的样式代码如下：

```
Paint paint=new Paint();
paint.setColor(0xFFFF0000);
paint.setStyle(Paint.Style.FILL_AND_STROKE);
paint.setStrokeWidth(50);
canvas.drawCircle(190, 200, 150, paint);
```

下面以绘制的一个圆形为例，看一下这三个类型的不同，效果如下图所示。



明显可见，FILL_AND_STROKE 是 FILL 和 STROKE 叠加在一起显示的结果，FILL_AND_STROKE 比 FILL 多了一个描边的宽度。

4．setStrokeWidth()

完整的函数声明如下：

```
void setStrokeWidth(float width)
```

用于设置描边宽度值，单位是 px。当画笔的 Style 样式是 STROKE 和 FILL_AND_STROKE 时有效。

有关该函数的使用，在上面的例子中已经多次涉及，这里就不再举例了。

1.1.3　Canvas使用基础

前面介绍了 Paint 的基本设置方法，下面再来讲讲有关 Canvas 绘图的知识。

1．画布背景设置

有三种方法可以实现画布背景设置。

```
void drawColor(int color)
void drawARGB(int a, int r, int g, int b)
void drawRGB(int r, int g, int b)
```

其中，drawColor()函数中参数 color 的取值必须是 8 位的 0xAARRGGBB 样式颜色值。

drawARGB()函数允许分别传入 A、R、G、B 分量，每个颜色值的取值范围都是 0～255（对应十六进制的 0x00～0xFF），内部会通过这些颜色分量构造出对应的颜色值。

drawRGB()函数只允许传入 R、G、B 分量，透明度 Alpha 的值取 255。

比如，将画布默认填充为紫色。

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawRGB(255,0,255);
}
```

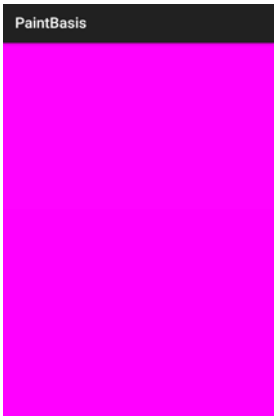
这里使用的是十进制的数值。当然，使用十六进制的数值会更直观。

```
drawRGB(0xFF,0x00,0xFF);
```

这个颜色值对应的另外两个函数的写法如下：

```
canvas.drawColor(0xFFFF00FF);
canvas.drawARGB(0xFF,0xFF,0,0xFF);
```

效果如下图所示。



2．画直线

```
void drawLine(float startX, float startY, float stopX, float stopY, Paint paint)
```

参数：

- startX：起始点 *X* 坐标。
- startY：起始点 *Y* 坐标。
- stopX：终点 *X* 坐标。
- stopY：终点 *Y* 坐标。

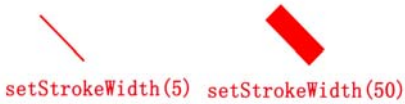
示例如下：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.FILL_AND_STROKE);
paint.setStrokeWidth(50);

canvas.drawLine(100, 100, 200, 200, paint);
```

当设置不同的 Style 类型时，效果如下图所示。

从效果图中可以明显看出，直线的粗细与画笔 Style 是没有关系的。

当设置不同的 StrokeWidth 时，效果如下图所示。



可见，直线的粗细是与 paint.setStrokeWidth 有直接关系的。所以，一般而言，paint.setStrokeWidth 在 Style 起作用时，用于设置描边宽度；在 Style 不起作用时，用于设置画笔宽度。

3．多条直线

构造函数一：

```
void drawLines(float[] pts, Paint paint)
```

参数：

pts：点的集合。从下面的代码中可以看到，这里不是形成连接线，而是每两个点形成一条直线，pts 的组织方式为｛x1,y1,x2,y2,x3,y3,…｝。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStrokeWidth(5);

float []pts={10,10,100,100,200,200,400,400};
canvas.drawLines(pts, paint);
```

上面有 4 个点，分别是(10,10)、(100,100)、(200,200)和(400,400)，两两连成一条直线。

效果如下图所示。

构造函数二：

```
void drawLines(float[] pts, int offset, int count, Paint paint)
```

相比上面的构造函数，这里多了两个参数。

- int offset：集合中跳过的数值个数。注意不是点的个数！一个点有两个数值。
- int count：参与绘制的数值个数，指 pts 数组中数值的个数，而不是点的个数，因为一个点有两个数值。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStrokeWidth(5);

float []pts={10,10,100,100,200,200,400,400};
canvas.drawLines(pts,2,4,paint);
```

表示从 pts 数组中索引为 2 的数字开始绘图，有 4 个数值参与绘图，也就是点(100,100)和(200,200)，所以效果图就是这两个点的连线。

效果如下图所示。

4. 点

```
void drawPoint(float x, float y, Paint paint)
```

参数：

- float x：点的 *X* 坐标。
- float y：点的 *Y* 坐标。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStrokeWidth(15);
canvas.drawPoint(100, 100, paint);
```

代码很简单，就是在(100,100)位置画一个点。同样，点的大小只与 paint.setStrokeWidth(width)有关，而与 paint.setStyle 无关。

效果如下图所示。

5. 多个点

```
void drawPoints(float[] pts, Paint paint)
void drawPoints(float[] pts, int offset, int count, Paint paint)
```

这几个参数的含义与多条直线中的参数含义相同。

- float[] pts：点的合集，与上面的直线一致，样式为｛x1,y1,x2,y2,x3,y3,…｝。
- int offset：集合中跳过的数值个数。注意不是点的个数！一个点有两个数值。

- int count：参与绘制的数值个数，指 pts 数组中数值的个数，而不是点的个数。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStrokeWidth(25);

float []pts={10,10,100,100,200,200,400,400};
canvas.drawPoints(pts, 2, 4, paint);
```

同样是上面的 4 个点：(10,10)、(100,100)、(200,200)和(400,400)，在 drawPoints()函数里跳过前两个数值，即第一个点的横、纵坐标，画出后面 4 个数值代表的点，即第二、三个点，第四个点没画。效果如下图所示。

6．矩形工具类 RectF、Rect 概述

这两个类都是矩形工具类，根据 4 个点构造出一个矩形结构。RectF 与 Rect 中的方法、成员变量完全一样，唯一不同的是：RectF 是用来保存 float 类型数值的矩形结构的；而 Rect 是用来保存 int 类型数值的矩形结构的。

我们先对比一下它们的构造函数。

RectF 的构造函数有如下 4 个，但最常用的还是第二个，即根据 4 个点构造出一个矩形。

```
RectF()
RectF(float left, float top, float right, float bottom)
RectF(RectF r)
RectF(Rect r)
```

Rect 的构造函数有如下 3 个。

```
Rect()
Rect(int left, int top, int right, int bottom)
Rect(Rect r)
```

可以看出，RectF 与 Rect 的构造函数基本相同，不同的只是 RectF 所保存的数值类型是 float 类型，而 Rect 所保存的数值类型是 int 类型。

一般而言，要构造一个矩形结构，可以通过以下两种方法来实现。

```
//方法一：直接构造
Rect rect = new Rect(10,10,100,100);
//方法二：间接构造
Rect rect = new Rect();
rect.set(10,10,100,100);
```

有关 Rect 和 RectF 的详细函数讲解参见 1.1.4 节。

7．矩形

在看完矩形的存储结构 RectF、Rect 以后，再来看看矩形的绘制方法。

```
void drawRect(float left, float top, float right, float bottom, Paint paint)
void drawRect(RectF rect, Paint paint)
```

```
void drawRect(Rect r, Paint paint)
```

第一个函数是直接传入矩形的 4 个点来绘制矩形的；第二、三个函数是根据传入 RectF 或者 Rect 的矩形变量来指定所绘制的矩形的。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.STROKE);
paint.setStrokeWidth(15);

//直接构造
canvas.drawRect(10, 10, 100, 100, paint);

//使用 RectF 构造
paint.setStyle(Paint.Style.FILL);
RectF rect = new RectF(210f, 10f, 300f, 100f);
canvas.drawRect(rect, paint);
```

这里绘制了两个同样大小的矩形。第一个直接使用 4 个点来绘制矩形，并且填充为描边类型；第二个通过 RectF 来绘制矩形，并且仅填充内容。

效果如下图所示。

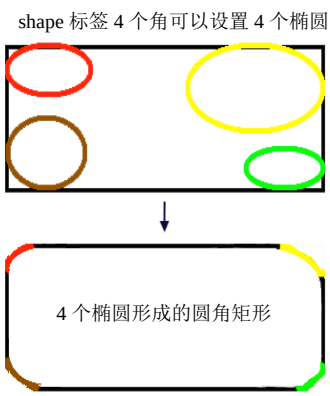
8. 圆角矩形

```
void drawRoundRect(RectF rect, float rx, float ry, Paint paint)
```

参数：

- RectF rect：要绘制的矩形。
- float rx：生成圆角的椭圆的 *X* 轴半径。
- float ry：生成圆角的椭圆的 *Y* 轴半径。

使用过标签的读者应该知道，Android 在生成矩形的圆角时，其实利用的是椭圆。shape 标签 4 个角都可以设置生成圆角的椭圆，它生成圆角矩形的原理如下图所示。



可见，圆角矩形的圆角其实是由椭圆的一角形成的。

与 shape 标签不同的是，drawRoundRect()函数不能针对每个角设置对应的椭圆，而只能统一设置 4 个角对应的椭圆。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStyle(Style.FILL);
paint.setStrokeWidth(15);

RectF rect = new RectF(100, 10, 300, 100);
canvas.drawRoundRect(rect, 20, 10, paint);
```

效果如下图所示。

9．圆形

```
void drawCircle(float cx, float cy, float radius, Paint paint)
```

参数：

- float cx：圆心点的 *X* 轴坐标。
- float cy：圆心点的 *Y* 轴坐标。
- float radius：圆的半径。

效果如下图所示。

10．椭圆

椭圆是根据矩形生成的，以矩形的长为椭圆的 *X* 轴，以矩形的宽为椭圆的 *Y* 轴。

```
void drawOval(RectF oval, Paint paint)
```

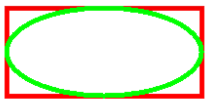
参数：

RectF oval：用来生成椭圆的矩形。

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.STROKE);
paint.setStrokeWidth(5);

RectF rect = new RectF(100, 10, 300, 100);
canvas.drawRect(rect, paint);

paint.setColor(Color.GREEN);//更改画笔颜色
canvas.drawOval(rect, paint);//根据同一个矩形画椭圆
```

针对同一个矩形，先把它的矩形区域画出来，然后再把根据这个矩形生成的椭圆画出来，

就可以很好地理解根据矩形所生成的椭圆与矩形的关系了，效果如下图所示。



11．弧

弧是椭圆的一部分，而椭圆是根据矩形来生成的，所以弧也是根据矩形来生成的。

```
    void drawArc(RectF oval, float startAngle, float sweepAngle, boolean useCenter,
Paint paint)
```

参数：

- RectF oval：生成椭圆的矩形。
- float startAngle：弧开始的角度，以 *X* 轴正方向为 0°。
- float sweepAngle：弧持续的角度。
- boolean useCenter：是否有弧的两边。为 true 时，表示带有两边；为 false 时，只有一条弧。

1）将画笔设为描边

```
    Paint paint=new Paint();
    paint.setColor(Color.RED);
    paint.setStyle(Paint.Style.STROKE);
    paint.setStrokeWidth(5);

    //带两边
    RectF rect1 = new RectF(10, 10, 100, 100);
    canvas.drawArc(rect1, 0, 90, true, paint);

    //不带两边
    RectF rect2 = new RectF(110, 10, 200, 100);
    canvas.drawArc(rect2, 0, 90, false, paint);
```

效果如下图所示。

左侧为带两边的弧，右侧为不带两边的弧。

2）将画笔设为填充

代码不变，只需要将 paint 的样式设置为 FILL 即可。

```
    paint.setStyle(Style.FILL);
```

效果如下图所示。

从效果图中可以看出，当画笔设为填充模式时，填充区域只限于圆弧的起始点和终点所形成的区域。当带有两边时，会将两边及圆弧内部全部填充；如果没有两边，则只填充圆弧部分。

1.1.4　Rect与RectF

在讲完 Paint、Canvas 的基本使用方法以后，再回来看看上面所涉及的 Rect、RectF 的常用函数。由于 Rect、RectF 所具有的函数是相同的，只是保存的数值类型不同，所以下面就以 Rect 为例来进行讲解。

1．是否包含点、矩形

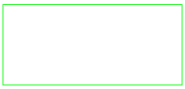
1）判断是否包含某个点

```
boolean contains(int x, int y)
```

该函数用于判断某个点是否在当前矩形中。如果在，则返回 true；如果不在，则返回 false。参数(x,y)就是当前要判断的点的坐标。

利用这个函数，可以定义一个很简单的控件：绘制一个矩形，当手指在这个矩形区域内时，矩形变为绿色；否则是红色的。

效果如下图所示。当手指在矩形区域内点击的时候，矩形边框是红色的；当手指在矩形区域外点击的时候，矩形边框是绿色的。



扫码看动画效果

下面讲解一下具体的实现方法。

（1）对相关变量进行初始化。

```
public class RectPointView extends View {
    private int mX,mY;
    private Paint mPaint;
    private Rect mrect;
    public RectPointView(Context context) {
        super(context);
        init();
    }

    public RectPointView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    public RectPointView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        init();
    }
```

```
    private void init(){
        mPaint = new Paint();
        mPaint.setStyle(Paint.Style.STROKE);
        mrect = new Rect(100,10,300,100);
    }
    ...
}
```

代码很简单，就是初始化 Paint，并指定一个矩形区域。

（2）拦截手指的触屏事件 onTouchEvent。

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    mX = (int)event.getX();
    mY = (int)event.getY();
    if (event.getAction() == MotionEvent.ACTION_DOWN){
        invalidate();
        return true;
    }else if (event.getAction() == MotionEvent.ACTION_UP){
        mX = -1;
        mY = -1;
    }
    postInvalidate();
    return super.onTouchEvent(event);
}
```

需要注意的是，因为我们需要判断当前手指是否在矩形区域内，以改变矩形的颜色，所以，我们必须首先获取到手指所在的位置，通过 event.getX()和 event.getY()函数就可以获取到当前手指在控件中的坐标。

然后，在手指下按时，我们需要让屏幕重绘，因为如果当前用户点击位置在矩形区域内，则需要将矩形变成红色。

```
if (event.getAction() == MotionEvent.ACTION_DOWN){
    invalidate();
    return true;
}
```

值得注意的是，在 MotionEvent.ACTION_DOWN 中返回 true，因为当 MotionEvent.ACTION_DOWN 消息到来时，系统会判断返回值，当返回 true 时，表示当前控件已经在拦截（消费）这个消息了，所以后续的 ACTION_MOVE、ACTION_UP 消息仍然继续传过来。如果返回 false（系统默认返回 false），就表示当前控件不需要这个消息，那么后续的 ACTION_MOVE、ACTION_UP 消息就不会再传到这个控件。

当用户手指弹起时，我们需要还原矩形的颜色（绿色），所以将 mX,mY 全部设置为负值。由于我们构造的矩形不包含坐标为负的点，所以也就还原了矩形的颜色。

最后，调用 postInvalidate()函数刷新控件屏幕，让控件重绘。

细心的读者会发现，在 ACTION_DOWN 消息到来时，我们调用了 invalidate()函数重绘控件。其实，postInvalidate()和 invalidate()函数都是用来重绘控件的，区别是 invalidate()函数一定要在主线程中执行，否则就会报错；而 postInvalidate()函数则没有那么多讲究，它可以在任

何线程中执行，而不必一定是主线程。因为在 postInvalidate()函数中就是利用 handler 给主线程发送刷新界面的消息来实现的，所以它可以在任何线程中执行而不会出错。而正因为它是通过发送消息来实现的，所以它的界面刷新速度可能没有直接调用 invalidate()函数那么快。因此，在确定当前线程是主线程的情况下，还是以 invalidate()函数为主。当我们不确定当前要刷新界面的位置所处的线程是不是主线程的时候，还是调用 postInvalidate()函数为好；这里笔者故意调用的是 postInvalidate()函数，因为 onTouchEvent()函数本来就是在主线程中的，所以使用 invalidate()函数更合适。

（3）绘图。

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

    if (mrect.contains(mX,mY)){
        mPaint.setColor(Color.RED);
    }else {
        mPaint.setColor(Color.GREEN);
    }
    canvas.drawRect(mrect,mPaint);
}
```

如果当前手指触点在矩形区域内，则将矩形画为红色；否则，画为绿色。

（4）使用控件。

在控件写好以后，就可以使用了，同样在布局中引入。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
            android:orientation="vertical"
            android:layout_width="match_parent"
            android:layout_height="match_parent">
    <com.harvic.PaintBasis.RectPointView
           android:layout_width="match_parent"
           android:layout_height="match_parent"/>
</LinearLayout>
```

到这里，这个自定义控件就结束了，我们很容易地实现了一个与手指交互的自定义控件。可见，自定义控件并没有想象中那么难。

2）判断是否包含某个矩形

```
Boolean contains(int left, int top, int right, int bottom)
boolean contains(Rect r)
```

根据矩形的 4 个点或者一个 Rect 矩形对象来判断这个矩形是否在当前的矩形区域内。

2．判断两个矩形是否相交

1）静态方法判断是否相交

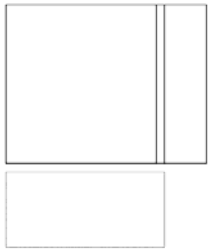
```
static boolean intersects(Rect a, Rect b)
```

这是 Rect 类的一个静态方法，用来判断参数中所传入的两个 Rect 矩形是否相交，如果相交则返回 true，否则返回 false。

下面用三种颜色画出三个矩形，然后判断矩形的相交情况。

```
Paint paint = new Paint();
paint.setStyle(Paint.Style.STROKE);

Rect rect_1 = new Rect(10,10,200,200);
Rect rect_2 = new Rect(190,10,250,200);
Rect rect_3 = new Rect(10,210,200,300);

//分别画出三个矩形
paint.setColor(Color.RED);
canvas.drawRect(rect_1,paint);
paint.setColor(Color.GREEN);
canvas.drawRect(rect_2,paint);
paint.setColor(Color.YELLOW);
canvas.drawRect(rect_3,paint);

//判断是否相交
Boolean interset1_2 = Rect.intersects(rect_1,rect_2);
Boolean interset1_3 = Rect.intersects(rect_1,rect_3);

Log.d("qijian","rect_1&rect_2:"+interset1_2+"  rect_1&rect_3:"+interset1_3);
```

效果如下图所示。



扫码看彩色图

很明显，rect_1 与 rect_2 是相交的，rect_1 与 rect_3 是不相交的。

日志如下：

```
1733-1733/com.harvic.PaintBasis D/qijian：  rect_1&rect_2:true  rect_1&rect_3:false
```

2）成员方法判断是否相交

还可以使用 Rect 类中自带的方法来判断当前 Rect 对象与其他矩形是否相交。

```
boolean intersects(int left, int top, int right, int bottom)
```

使用方法：

```
Rect rect_1 = new Rect(10,10,200,200);
Boolean interset1_2 = rect_1.intersects(190,10,250,200);
```

3）判断相交并返回结果

```
boolean intersect(int left, int top, int right, int bottom)
boolean intersect(Rect r)
```

这两个成员方法与 intersects()方法的区别是，不仅会返回是否相交的结果，而且会把相交

部分的矩形赋给当前 Rect 对象。如果两个矩形不相交，则当前 Rect 对象的值不变。

```
Rect rect_1 = new Rect(10, 10, 200, 200);
Boolean result_1 = rect_1.intersects(190, 10, 250, 200);
printResult(result_1,rect_1);

Boolean result_2 = rect_1.intersect(210, 10, 250, 200);
printResult(result_2, rect_1);

Boolean result_3 = rect_1.intersect(190, 10, 250, 200);
printResult(result_3,rect_1);
```

其中，printResult()函数的代码为：

```
private void printResult(Boolean result, Rect rect) {
    Log.d("qijian", rect.toShortString() + "  result:" + result);
}
```

在上面的示例中，分别使用 intersects()和 intersect()函数来判断是否与指定矩形相交，并将判断相交的对象 rect_1 的边角打印出来。

日志如下：

```
25427-25427/com.harvic.PaintBasis D/qijian：  [10,10][200,200]  result:true
25427-25427/com.harvic.PaintBasis D/qijian：  [10,10][200,200]  result:false
25427-25427/com.harvic.PaintBasis D/qijian：  [190,10][200,200]  result:true
```

很明显，intersects()函数只是判断是否相交，并不会改变原矩形 rect_1 的值。当 intersect()函数判断的两个矩形不相交时，也不会改变 rect_1 的值；只有当两个矩形相交时，intersect()函数才会把结果赋给 rect_1。

3．合并

1）合并两个矩形

合并两个矩形的意思就是将两个矩形合并成一个矩形，即无论这两个矩形是否相交，取两个矩形最小左上角点作为结果矩形的左上角点，取两个矩形最大右下角点作为结果矩形的右下角点。如果要合并的两个矩形有一方为空，则将有值的一方作为最终结果。

```
public void union(int left, int top, int right, int bottom)
public void union(Rect r)
```

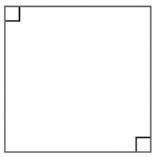
同样根据参数是矩形的 4 个点还是一个 Rect 对象分为两个构造函数。合并的结果将会被赋给当前的 rect 变量。

示例如下：

```
Paint paint = new Paint();
paint.setStyle(Paint.Style.STROKE);
Rect rect_1 = new Rect(10,10,20,20);
Rect rect_2 = new Rect(100,100,110,110);

//分别画出源矩形 rect_1、rect_2
paint.setColor(Color.RED);
canvas.drawRect(rect_1,paint);
paint.setColor(Color.GREEN);
canvas.drawRect(rect_2,paint);
```

```
//画出合并之后的结果 rect_1
paint.setColor(Color.YELLOW);
rect_1.union(rect_2);
canvas.drawRect(rect_1,paint);
```

上述代码构造了两个不相交的矩形，先分别画出它们各自所在的位置，然后通过 union()函数合并，并将合并结果画出来，效果如下图所示。



从结果图中可以看出，两个小矩形在合并以后，取两个矩形的最小左上角点作为结果矩形的左上角点，取两个矩形的最大右下角点作为结果矩形的右下角点。

2）合并矩形与某个点

```
public void union(int x, int y)
```

先判断当前矩形与目标合并点的关系，如果不相交，则根据目标点(*x*,*y*)的位置，将目标点设置为当前矩形的左上角点或者右下角点。如果当前矩形是一个空矩形，则最后的结果矩形为([0,0],[*x*,*y*])，即结果矩形的左上角点为[0,0]，右下角点为[*x*,*y*]。

```
Rect rect_1 = new Rect(10, 10, 20, 20);
rect_1.union(100,100);
printResult(rect_1);

rect_1 = new Rect();
rect_1.union(100,100);
printResult(rect_1);
```

其中，printResult()函数的代码为：

```
private void printResult( Rect rect) {
   Log.d("qijian", rect.toShortString());
}
```

在上述代码中，先将与指定矩形不相交的点与矩形合并，然后将矩形置空，再与同一个点相交，日志如下：

```
7028-7028/com.harvic.PaintBasis D/qijian：  [10,10][100,100]
7028-7028/com.harvic.PaintBasis D/qijian：  [0,0][100,100]
```

结果很容易理解，当与指定矩形合并时，根据当前点的位置，将该点设为矩形的右下角点；当点与空矩形相交时，结果为([0,0],[*x*,*y*])。

1.1.5 Color

前面提到，除手动组合颜色的方法以外，系统还提供了一个专门用来解析颜色的类：Color。Color 是 Android 中与颜色处理有关的类。

1．常量颜色

首先，它定义了很多常量的颜色值，我们可以直接使用。

```
int BLACK
int BLUE
int CYAN
int DKGRAY
int GRAY
int GREEN
int LTGRAY
int MAGENTA
int RED
int TRANSPARENT
int WHITE
int YELLOW
```

可以通过 Color.XXX 来直接使用这些颜色，比如红色，在代码中可以直接使用 Color.RED。在前面的代码中，我们也用到过 Color 的色彩常量，这里就不再赘述了。

2．构造颜色

1）带有透明度的颜色

```
static int argb(int alpha, int red, int green, int blue)
```

这个函数允许我们分别传入 A、R、G、B 4 个色彩分量，然后合并成一个色彩。其中，alpha、red、green、blue 4 个色彩分量的取值范围都是 0～255。

我们来看一下 argb()函数的具体实现源码，如下：

```
public static int argb(int alpha, int red, int green, int blue) {
    return (alpha << 24) | (red << 16) | (green << 8) | blue;
}
```

其中，<<是向左位移符号，表示将指定的二进制数字向左位移多少位。比如，二进制的 110 向左移一位之后的结果为 1100。

比如，alpha << 24 表示向左位移 24 位。我们知道，一个色彩值对应的取值范围是 0～255，所以每个色彩值对应二进制的 8 位，比如 alpha 的取值为 255，它的二进制表示就是 11111111（8 个 1）。

所以，alpha << 24 的结果为 11111111 00000000 00000000 00000000。

同样，如果我们构造的是一个白色，那么 A、R、G、B 各个分量的值都是 255，当它们对应位移之后的结果如下。

alpha << 24：11111111 00000000 00000000 00000000

red << 16：00000000 11111111 00000000 00000000

green << 8：00000000 00000000 11111111 00000000

blue：00000000 00000000 00000000 11111111

利用“|”（二进制的或运算）将各个二进制的值合并之后的结果就是 11111111 11111111 11111111 11111111，分别对应 A、R、G、B 分量。这就是各个分量最终合成对应颜色值的过

程。在读代码时，有时会看到直接利用(alpha << 24) | (red << 16) | (green << 8) | blue 来合成对应颜色值的情况，其实跟我们使用 Color.argb()函数来合成的结果是一样的。

2）不带透明度的颜色

```
static int rgb(int red, int green, int blue)
```

其实跟上面的构造函数是一样的，只是不允许指定 alpha 值，alpha 值取 255（完全不透明）。

3．提取颜色分量

我们不仅能通过 Color 类来合并颜色分量，而且能从一个颜色中提取出指定的颜色分量。

```
static int alpha(int color)
static int red(int color)
static int green(int color)
static int blue(int color)
```

我们能通过上面的 4 个函数提取出对应的 A、R、G、B 颜色分量。

比如：

```
int green = Color.green(0xFF000F00);
```

得到的结果 green 的值就是 0x0F，很简单，就不再赘述了。

本节主要讲解了有关 Paint 和 Canvas 的基本使用，并讲述了相关的 Rect 和 Color 类的用法。

但我们在示例中创建 Paint 对象和其他对象时都是在 onDraw()函数中实现的，其实这在现实代码中是不被允许的。因为当需要重绘时就会调用 onDraw()函数，所以在 onDraw()函数中创建的变量会一直被重复创建，这样会引起频繁的程序 GC（回收内存），进而引起程序卡顿。这里之所以这样做，是因为可以提高代码的可读性。大家一定要记住，在 onDraw()函数中不能创建变量！一般在自定义控件的构造函数中创建变量，即在初始化时一次性创建。

1.2 路径

1.2.1 概述

用过 Photoshop 的读者应该对路径比较熟悉，类似我们用画笔画画，画笔所画出来的一段不间断的曲线就是路径。

在 Android 中，Path 类就代表路径。

在 Canvas 中绘制路径的方法如下：

```
void drawPath(Path path, Paint paint)
```

1.2.2 直线路径

画一条直线路径，一般涉及下面三个函数。

```
void moveTo(float x1, float y1)
```

(x1,y1)是直线的起始点，即将直线路径的绘制点定在(x1,y1)位置。

```
void lineTo(float x2, float y2)
```

(x2,y2)是直线的终点，又是下一次绘制直线路径的起始点；lineTo()函数可以一直使用。

```
void close()
```

如果连续画了几条直线，但没有形成闭环，那么调用 close()函数会将路径首尾点连接起来，形成闭环。

示例：画一个三角形。

```
Paint paint=new Paint();
paint.setColor(Color.RED);                  //设置画笔颜色
paint.setStyle(Paint.Style.STROKE);         //填充样式改为描边
paint.setStrokeWidth(5);                    //设置画笔宽度

Path path = new Path();

path.moveTo(10, 10);          //设定起始点
path.lineTo(10, 100);         //第一条直线的终点，也是第二条直线的起始点
path.lineTo(300, 100);        //画第二条直线
path.close();                 //闭环

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

我们先沿逆时针方向画了两条直线，分别是从(10, 10)到(10, 100)和从(10, 100)到(300, 100)，然后利用 path.close()函数将路径闭合，路径的终点(300,100)就会自行向路径的起始点(10,10)画一条闭合线，所以最终我们看到的是一个路径闭合的三角形。

效果如下图所示。

1.2.3　弧线路径

```
void arcTo(RectF oval, float startAngle, float sweepAngle)
```

这是一个画弧线路径的方法，弧线是从椭圆上截取的一部分。

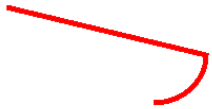
参数：

- RectF oval：生成椭圆的矩形。
- float startAngle：弧开始的角度，以 X 轴正方向为 0°。
- float sweepAngle：弧持续的角度。

示例：

```
Path path = new Path();
path.moveTo(10,10);
RectF rectF = new RectF(100,10,200,100);
path.arcTo(rectF,0,90);

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

效果如下图所示。



从效果图中发现一个问题：我们只画了一条弧，为什么弧最终还是会和起始点(10,10)连接起来？

因为在默认情况下路径都是连贯的，除非以下两种情况：

- 调用 addXXX 系列函数（参见 1.2.4 节），将直接添加固定形状的路径。
- 调用 moveTo()函数改变绘制起始位置。

如果我们不想连接怎么办？Path 类也提供了另外两个重载方法。

```
    void arcTo(float left, float top, float right, float bottom, float startAngle,
float sweepAngle, boolean forceMoveTo)
    void arcTo(RectF oval, float startAngle, float sweepAngle, boolean
forceMoveTo)
```

参数 boolean forceMoveTo 的含义是是否强制将弧的起始点作为绘制起始位置。

将上面的代码稍加改造：

```
    Path path = new Path();
    path.moveTo(10,10);
    RectF rectF = new RectF(100,10,200,100);
    path.arcTo(rectF,0,90,true);

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

效果如下图所示。

1.2.4 addXXX系列函数

前面我们讲过，路径一般都是连贯的，而 addXXX 系列函数可以让我们直接往 Path 中添加一些曲线，而不必考虑连贯性。注意“添加”这个词。

示例：

```
    Path path = new Path();
    path.moveTo(10,10);
    path.lineTo(100,50);
    RectF rectF = new RectF(100,100,150,150);
    path.addArc(rectF,0,90);
    canvas.drawPath(path, paint);
```

效果如下图所示。

虽然我们先绘制了从(10,10)到(100,50)的线段，但是在我们往路径中添加了一条弧线之后，弧线并没有与线段连接。除了 addArc()函数，Path 类还提供了一系列的 add 函数，下面我们就一一来讲解。

1．添加矩形路径

```
    void addRect(float left, float top, float right, float bottom, Path.Direction
dir)
    void addRect(RectF rect, Path.Direction dir)
```

这里 Path 类创建矩形路径的参数与 Canvas 绘制矩形的参数差不多，唯一不同的是增加了 Path.Direction 参数。

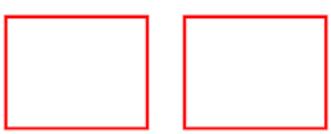
Path.Direction 参数有两个值。

- Path.Direction.CCW：是 counter-clockwise 的缩写，指创建逆时针方向的矩形路径。
- Path.Direction.CW：是 clockwise 的缩写，指创建顺时针方向的矩形路径。

示例：创建两条大小相同但方向不同的路径。

```
    //先创建两条大小相同的路径
    //第一条路径逆向生成
    Path CCWRectpath = new Path();
    RectF rect1 =  new RectF(50, 50, 240, 200);
    CCWRectpath.addRect(rect1, Direction.CCW);

    //第二条路径顺向生成
    Path CWRectpath = new Path();
    RectF rect2 =  new RectF(290, 50, 480, 200);
    CWRectpath.addRect(rect2, Direction.CW);

    //先画出这两条路径
    canvas.drawPath(CCWRectpath, paint);
    canvas.drawPath(CWRectpath, paint);
```

效果如下图所示。



从效果图中看不出顺时针生成和逆时针生成的任何区别，那么问题来了：为什么生成方向对路径大小没有影响呢？

答：无论是顺时针还是逆时针生成，仅仅是生成方向不同而已，路径的大小只与生成路径的矩形大小有关，与生成方向无关。

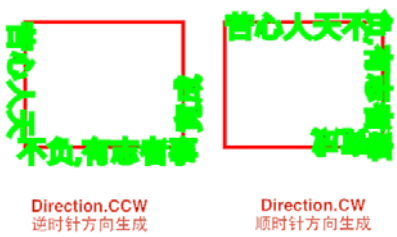
那么问题又来了：生成方向是用来做什么的呢？

答：生成方向的区别在于依据生成方向排版的文字。后面我们会讲到文字，文字是可以依据路径排版的，文字的行走方向依据的就是路径的生成方向。

示例：根据路径方向布局文字。

```
//先创建两条大小相同的路径
//第一条路径逆向生成
Path CCWRectpath = new Path();
RectF rect1 =  new RectF(50, 50, 240, 200);
CCWRectpath.addRect(rect1, Path.Direction.CCW);

//第二条路径顺向生成
Path CWRectpath = new Path();
RectF rect2 =  new RectF(290, 50, 480, 200);
CWRectpath.addRect(rect2, Path.Direction.CW);

//先画出这两条路径
canvas.drawPath(CCWRectpath, paint);
canvas.drawPath(CWRectpath, paint);

//依据路径布局文字
String text="苦心人天不负,有志者事竟成";
paint.setColor(Color.GREEN);
paint.setTextSize(35);
canvas.drawTextOnPath(text, CCWRectpath, 0, 18, paint);//逆时针方向生成
canvas.drawTextOnPath(text, CWRectpath, 0, 18, paint);//顺时针方向生成
```

效果如下图所示。



这里我们只需要知道路径的生成方向与文字的关系即可，有关依据路径绘制文字的部分，将在 1.3 节中讲述。

2．添加圆角矩形路径

```
void addRoundRect(RectF rect, float[] radii, Path.Direction dir)
void addRoundRect(RectF rect, float rx, float ry, Path.Direction dir)
```

前面讲过，矩形的圆角都是利用椭圆生成的。在这两个构造函数中，RectF rect 是当前所构造路径的矩形；Path.Direction dir 依然是指路径的生成方向，当然只对依据路径布局的文字有用。

在第一个构造函数中，我们可以指定每个角的圆角大小。

float[] radii：必须传入 8 个数值，分 4 组，分别对应每个角所使用的椭圆的横轴半径和纵

轴半径，如｛x1,y1,x2,y2,x3,y3,x4,y4｝，其中，x1,y1 对应第一个角（左上角）的用来生成圆角的椭圆的横轴半径和纵轴半径，其他类推……

而在第二个构造函数中，只能构建统一的圆角大小。

- float rx：生成圆角的椭圆的横轴半径。
- float ry：生成圆角的椭圆的纵轴半径。

示例：

```
Path path = new Path();
RectF rect1 =  new RectF(50, 50, 240, 200);
path.addRoundRect(rect1, 10, 15 , Direction.CCW);

RectF rect2 =  new RectF(290, 50, 480, 200);
float radii[] ={10,15,20,25,30,35,40,45};
path.addRoundRect(rect2, radii, Direction.CCW);

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

从这段代码中可以看出，我们针对同一条路径连续两次调用 path.addRoundRect()函数添加两个椭圆矩形，直接在对应的区域画出矩形即可，而不必考虑连贯性。

效果如下图所示。

3．添加圆形路径

```
void addCircle(float x, float y, float radius, Path.Direction dir)
```

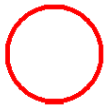
参数：

- float x：圆心 *X* 轴坐标。
- float y：圆心 *Y* 轴坐标。
- float radius：圆半径。

示例：

```
Path path = new Path();
path.addCircle(100, 100, 50, Direction.CCW);
canvas.drawPath(path, paint);
```

效果如下图所示。



4．添加椭圆路径

```
void addOval(RectF oval, Path.Direction dir)
```

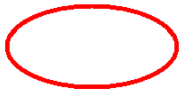
参数：

- RectF oval：生成椭圆的矩形。
- Path.Direction：生成方向，与矩形一样，分为顺时针与逆时针，意义完全相同，不再重复。

示例：

```
Path path = new Path();
RectF rect =  new RectF(10, 10, 200, 100);
path.addOval(rect, Path.Direction.CCW);
canvas.drawPath(path, paint);
```

效果如下图所示。



5．添加弧形路径

```
    void addArc(float left, float top, float right, float bottom, float startAngle,
float sweepAngle)
    void addArc(RectF oval, float startAngle, float sweepAngle)
```

参数：

- RectF oval：弧是椭圆的一部分，这个参数就是生成椭圆的矩形。
- float startAngle：弧开始的角度，以 X 轴正方向为 0°。
- float sweepAngel：弧持续的角度。

示例：

```
Path path = new Path();
RectF rect =  new RectF(10, 10, 100, 50);
path.addArc(rect, 0, 100);

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

效果如下图所示。

1.2.5 填充模式

Path 的填充模式与 Paint 的填充模式不同。Path 的填充模式是指填充 Path 的哪部分。Path.FillType 表示 Path 的填充模式，它有 4 个枚举值。

- FillType.WINDING：默认值，当两个图形相交时，取相交部分显示。
- FillType.EVEN_ODD：取 path 所在并不相交的区域。
- FillType.INVERSE_WINDING：取 path 的外部区域。
- FillType.INVERSE_EVEN_ODD：取 path 的外部和相交区域。

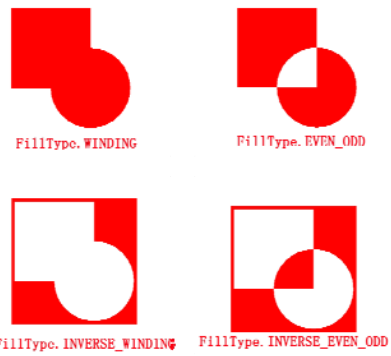
Inverse 就是取反的意思，所以 FillType.INVERSE_WINDING 就是取 FillType.WINDING 的相反部分；同理，FillType.INVERSE_EVEN_ODD 就是取 FillType.EVEN_ODD 的相反部分。

设置填充模式使用 setFillType(FillType filltype)函数。

示例：

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.FILL);

Path path = new Path();
path.addRect(100, 100, 300, 300, Path.Direction.CW);
path.addCircle(300, 300, 100, Path.Direction.CW);
path.setFillType(Path.FillType.WINDING);
canvas.drawPath(path,paint);
```

使用上面的代码，分别设置不同的填充类型，利用 Paint 填充路径，结果如下图所示。



所以，在利用画笔填充图形时，填充的肯定是图形内部，而 path.setFillType()函数就是用来界定哪里算 Path 内部的算法，进而让 Paint 填充这部分图像。

1.2.6　重置路径

1．概述

当我们需要重绘一条全新的路径时，Android 开发人员为了重复利用空间，允许我们重置路径对象。路径对象一旦被重置，其中保存的所有路径都将被清空，这样我们就不需要重新定义一个路径对象了。重新定义路径对象的问题在于老对象的回收和新对象的内存分配，当然这些过程都是会消耗手机性能的。

系统提供了两个重置路径的方法，分别是：

```
void reset()
void rewind()
```

这两个函数的共同点是都会清空内部所保存的所有路径，但二者也有区别。

- rewind()函数会清除 FillType 及所有的直线、曲线、点的数据等，但是会保留数据结构。这样可以实现快速重用，提高一定的性能。例如，重复绘制一类线段，它们的点的数量都相等，那么使用 rewind()函数可以保留装载点数据的数据结构，效率会更高。一

定要注意的是，只有在重复绘制相同的路径时，这些数据结构才是可以复用的。

- reset()函数类似于新建一个路径对象，它的所有数据空间都会被回收并重新分配，但不会清除 FillType。

从整体来讲，rewind()函数不会清除内存，但会清除 FillType；而 reset()函数则会清除内存，但不会清除 FillType。

有关是否清除内存，在代码中不易演示，我们来看一下这两个函数各自对于 FillType 的清除情况。

2．reset()与 FillType

```
Path path = new Path();
path.setFillType(Path.FillType.INVERSE_WINDING);
path.reset();
path.addCircle(100, 100, 50, Path.Direction.CW);
canvas.drawPath(path, paint);
```

效果如下图所示。

很明显，首先设置 Path 的填充类型为 FillType.INVERSE_WINDING，在调用 reset()函数以后，画了一个圆，但结果却是圆形以外的区域。很明显，调用 reset()函数并没有清除原有的 Path 填充类型。

3．rewind()与 FillType

我们还是使用上面的代码，把 reset()改成 rewind()。

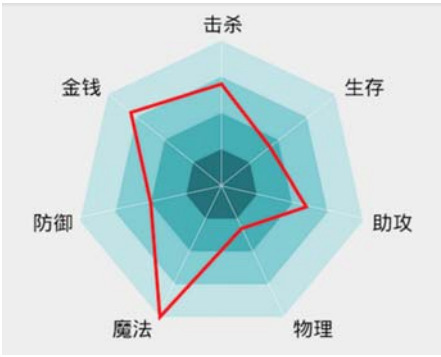
```
Path path = new Path();
path.setFillType(Path.FillType.INVERSE_WINDING);
path.rewind();
path.addCircle(100, 100, 50, Path.Direction.CW);
canvas.drawPath(path, paint);
```

效果如下图所示。

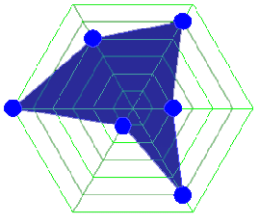
很明显，使用 rewind()函数清空了填充类型。

1.2.7　示例：蜘蛛网状图

大家在打游戏时，经常会看到如下图所示的技能分析。



想必大家一眼就可以看出中间的线条部分就是利用 Path 来实现的。本小节就利用 Path 来实现一个技能分析的简化版，效果如下图所示。



扫码看彩色图

从效果图中可以看出，我们要先画出一个网格，默认网格数和边角数都是 6。在代码中，为了简化逻辑，我们会将所有可变的内容，比如画笔颜色、网格数、边角数设为固定值。其实这些值都应该在初始化的时候通过对应的 set 函数设置到自定义控件内部，大家可以自行补充。

1．初始化

我们说过，不能在 onDraw()函数中创建变量，所以必然会有一个初始化函数，用于在创建控件的时候初始化画笔等参数。

```
public class SpiderView extends View {
    private Paint radarPaint,valuePaint;

    public SpiderView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }
    private void init(){
        radarPaint = new Paint();
        radarPaint.setStyle(Paint.Style.STROKE);
        radarPaint.setColor(Color.GREEN);

        valuePaint = new Paint();
        valuePaint.setColor(Color.BLUE);
        valuePaint.setStyle(Paint.Style.FILL);
    }
    ...
}
```

这里初始化了两个画笔，其中 radarPaint 是用来绘制网格的，所以类型设置为描边；而 valuePaint 是用来绘制结果图的，所以设置成蓝色画笔，样式为填充。

2．获得布局中心

在 onSizeChanged(int w, int h, int oldw, int oldh)函数中，根据 View 的长、宽，获取整个布局的中心坐标，因为整个雷达都是从这个中心坐标开始绘制的。

```
private float radius;    //网格最大半径
private int centerX;     //中心 X
private int centerY;     //中心 Y

@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    radius = Math.min(h, w)/2*0.9f;
    //中心坐标
    centerX = w/2;
    centerY = h/2;
    postInvalidate();
    super.onSizeChanged(w, h, oldw, oldh);
}
```

我们知道，在控件大小发生变化时，都会通过 onSizeChanged()函数通知我们当前控件的大小。所以，我们只需要重写 onSizeChanged()函数，即可得知当前控件的最新大小。

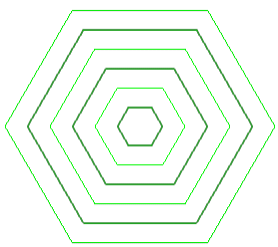
因为蜘蛛网的总大小占当前控件大小的 90%，所以，我们将蜘蛛网的半径设置为 Math.min(h, w)/2*0.9f。

然后依据绘图中心，分别绘制蜘蛛网格、画网格中线、画数据图，即可完成整个效果图的绘制。

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    //绘制蜘蛛网格
    drawPolygon(canvas);
    //画网格中线
    drawLines(canvas);
    //画数据图
    drawRegion(canvas);
}
```

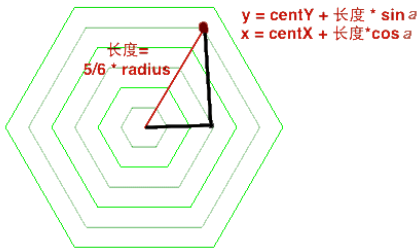
3．绘制蜘蛛网格

下面我们就要绘制蜘蛛网格了，效果如下图所示。



很显然，蜘蛛网格是利用 Path 的 moveTo()和 lineTo()函数一圈圈画出来的，我们需要计算出每个转折点的位置。

比如，计算下图中所标记点的 *x*,*y* 坐标。



很明显，标记点在半径的 5/6 位置，而标记点与中心点的连线与 *X* 轴的夹角为 *a*，所以由图可得：

```
x = centX + 5/6 * radius * sina;
y = centY + 5/6 * radius * cosa;
```

因为我们共画了 6 个角，所以每个角的度数应该是 360°/6 = 60°。

依据上面的原理，列出画蜘蛛网格的代码如下：

```
private void drawPolygon(Canvas canvas){
    Path path = new Path();
    float r = radius/(count);     //r 是蜘蛛丝之间的间距
    for(int i=1;i<=count;i++){    //中心点不用绘制
        float curR = r*i;         //当前半径
        path.reset();
        for(int j=0;j<count;j++){
            if(j==0){
                path.moveTo(centerX+curR,centerY);
            }else{
                //根据半径，计算出蜘蛛丝上每个点的坐标
                float x = (float) (centerX+curR*Math.cos(angle*j));
                float y = (float) (centerY+curR*Math.sin(angle*j));
                path.lineTo(x,y);
            }
        }
        path.close();//闭合路径
        canvas.drawPath(path, radarPaint);
    }
}
```

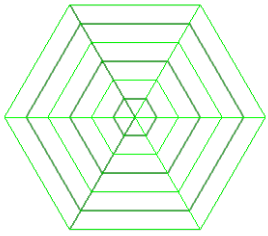
4．画网格中线

在画完蜘蛛网格以后，我们需要画从网格中心到末端的直线，代码如下：

```
private void drawLines(Canvas canvas){
    Path path = new Path();
    for(int i=0;i<count;i++){
        path.reset();
        path.moveTo(centerX, centerY);
```

```
            float x = (float) (centerX+radius*Math.cos(angle*i));
            float y = (float) (centerY+radius*Math.sin(angle*i));
            path.lineTo(x, y);
            canvas.drawPath(path, radarPaint);
        }
    }
```

效果如下图所示。



绘制原理与绘制蜘蛛网格是一样的，先找到各个末端点的坐标，然后画一条从中心点到末端点的连线即可。

5．画数据图

绘制数据区域其实也很简单，首先要确定每个数据点的位置。当然，网格线中的每一层网格都应该对应一个数值，在这里为了方便起见，将网格的最大值设为 6，即每一层数值是按 1,2,3,4,5,6 分布的。

```
    //数据
    private double[] data={2,5,1,6,4,5};
    //最大值
    private float maxValue=6;
```

这里的 data 数组就是我们要画出来的每个数据点。

数据图的绘制代码如下：

```
    private void drawRegion(Canvas canvas){
        Path path = new Path();
        valuePaint.setAlpha(127);
        for(int i=0;i<count;i++){
            double percent = data[i]/maxValue;
            float x = (float) (centerX+radius*Math.cos(angle*i)*percent);
            float y = (float) (centerY+radius*Math.sin(angle*i)*percent);
            if(i==0){
                path.moveTo(x, centerY);
            }else{
                path.lineTo(x,y);
            }
            //绘制小圆点
            canvas.drawCircle(x,y,10,valuePaint);
        }
        //绘制填充区域
        valuePaint.setStyle(Paint.Style.FILL_AND_STROKE);
        canvas.drawPath(path, valuePaint);
    }
```

其实逻辑很简单，就是先找到数组中所对应的每个点的坐标，然后在对应位置画一个圆，再将各个点利用 Path 连接起来，填充为蓝色即可。

1.3　文字

本节我们就来看看有关文字绘制的知识。

1.3.1　Paint设置

Paint 与文字相关的设置方法有如下几个。

```
//普通设置
paint.setStrokeWidth (5);    //设置画笔宽度
paint.setAntiAlias(true);    //指定是否使用抗锯齿功能。如果使用，则会使绘图速度变慢
paint.setStyle(Paint.Style.FILL);    //绘图样式，对于文字和几何图形都有效
paint.setTextAlign(Align.CENTER);    //设置文字对齐方式，取值为 Align.CENTER、
                                     //Align.LEFT 或 Align.RIGHT
paint.setTextSize(12);               //设置文字大小

//样式设置
paint.setFakeBoldText(true);         //设置是否为粗体文字
paint.setUnderlineText(true);        //设置下画线
paint.setTextSkewX((float) -0.25);   //设置字体水平倾斜度，普通斜体字设为-0.25
paint.setStrikeThruText(true);       //设置带有删除线效果

//其他设置
paint.setTextScaleX(2);              //只会将水平方向拉伸，高度不会变
```

我们逐个来看各个函数的用法。

1．填充样式的区别

前面说过，paint.setStyle()函数对文字和几何图形都有效。下面就来看看不同的填充样式对文字的影响

```
Paint paint=new Paint();
paint.setColor(Color.RED);   //设置画笔颜色

paint.setStrokeWidth (5);    //设置画笔宽度
paint.setAntiAlias(true);    //指定是否使用抗锯齿功能。如果使用，则会使绘图速度变慢
paint.setTextSize(30);       //设置文字大小

//绘图样式，设置为填充
paint.setStyle(Paint.Style.FILL);
canvas.drawText("床前明月光", 10,100, paint);
```

依据这段代码，在不同填充模式下的效果如下图所示。

2．setTextAlign()函数

```
public void setTextAlign(Align align)
```

用于设置所要绘制的字符串与起始点的相对位置。参数 Align align 的取值如下。

- Align.LEFT：居左绘制，即通过 drawText()函数指定的起始点在最左侧，文字从起始点位置开始绘制。
- Align.CENTER：居中绘制，即通过 drawText()函数指定的起始点在文字中间位置。
- Align.RIGHT：居右绘制，即通过 drawText()函数指定的起始点在文字右侧位置。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);

paint.setStrokeWidth (5);
paint.setAntiAlias(true);
paint.setTextSize(80);
paint.setTextAlign(Paint.Align.RIGHT);

canvas.drawText("床前明月光", 400,100, paint);
```

在上面的代码中，起始点是(400,100)，当取不同的 Align 值时，结果如下图所示。

从效果图中可以看出，当居左对齐（Align.LEFT）时，整个字符串都在起始点(400,100)的右侧，也就是说，通过 drawText()函数指定的起始点(400,100)是居左的。同样，当居右对齐（Align.RIGHT）时，起始点也是居右的，也就是说，所有文字都在起始点(400,100)的左侧。

3．设置字体样式

1）常规设置

与字体样式设置有关的几个函数如下。

```
void setFakeBoldText(boolean fakeBoldText)
```

设置是否粗体。当取 true 时，表示是粗体。

```
void setUnderlineText(boolean underlineText)
```

是否显示文字下画线。当取 true 时，显示下画线。

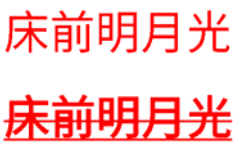
```
void setStrikeThruText(boolean strikeThruText)
```

是否显示中间删除线。当取 true 时，显示中间删除线。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(80);
canvas.drawText("床前明月光", 10,100, paint);

paint.setFakeBoldText(true);      //设置粗体文字
paint.setUnderlineText(true);     //设置下画线
paint.setStrikeThruText(true);    //设置带有删除线效果
canvas.drawText("床前明月光", 10,250, paint);
```

效果如下图所示。



我们先后写了两次，第一次没有任何的样式设置，第二次将所有的样式设置为 true，可以很明显地看出区别。

2）字体倾斜度设置

```
void setTextSkewX(float skewX)
```

该函数用于设置字体倾斜度。参数 float skewX 的默认值是 0，取负值时文字向右倾斜，取正值时文字向左倾斜，Word 文档中倾斜字体的倾斜度是−0.25f。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(80);
//正常样式
canvas.drawText("床前明月光", 10,100, paint);

//向右倾斜
paint.setTextSkewX(-0.25f);
canvas.drawText("床前明月光", 10,200, paint);

//向左倾斜
paint.setTextSkewX(0.25f);
canvas.drawText("床前明月光", 10,300, paint);
```

从效果图中可以明显看出取不同倾斜度时的倾斜方向。

4．水平拉伸

```
void setTextScaleX(float scaleX)
```

该函数用于在水平方向拉伸文字。参数 float scaleX 表示拉伸倍数，当取值为 1 时，表示不拉伸。默认为不拉伸。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(80);
//正常样式
canvas.drawText("床前明月光", 10,100, paint);
//拉伸 2 倍
paint.setTextScaleX(2);
canvas.drawText("床前明月光", 10,200, paint);
```

效果如下图所示。

从效果图中可以看出，字体大小和字体间距是均匀拉伸的。在拉伸 2 倍以后，一个字所占的位置是原来两个字所占的位置。

1.3.2 Canvas绘制文本

1．普通绘制

一般的绘制方法有下面几种。

```
void drawText(String text, float x, float y, Paint paint)
```

该函数可以指定起始点来绘制文字，上面各个示例中使用的都是这个函数，其中参数(x,y)就是起始点坐标。

```
    void drawText(CharSequence text, int start, int end, float x, float y, Paint
paint)
    void drawText(String text, int start, int end, float x, float y, Paint paint)
```

这两个函数通过指定字符串中字符的起始和终止位置截取字符串的一部分绘制，它允许我们指定 CharSequence 或者 String 类型的字符串。其他参数的含义如下。

- start：表示起始绘制字符所在字符串中的索引。

- end：表示结束绘制字符所在字符串中的索引。
- x,y：起始点坐标。

```
void drawText(char[] text, int index, int count, float x, float y, Paint paint)
```

该函数同样截取字符串的一部分绘制，但它只接受由绘制 char 类型的数组所组成的字符串。其他参数的含义如下。

- int index：指定起始绘制字符的位置。
- int count：指定从起始绘制字符开始绘制几个字符。
- x,y：起始点坐标。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(80);

canvas.drawText("床前明月光",2,4, 10,100, paint);
```

效果如下图所示。

明月

很明显，这里使用的函数是 drawText(String text, int start, int end, float x, float y, Paint paint)。在代码中，我们指定起始字符的索引是 2，结束字符的索引是 4，也就是截取“明月”两个字绘制。

2．逐个指定文字位置

Canvas 允许我们指定每个要绘制的文字的具体位置。

```
void drawPosText(String text, float[] pos, Paint paint)
void drawPosText(char[] text, int index, int count, float[] pos, Paint paint)
```

很明显，第二个构造函数先将字符串截取一段，再指定所截取的文字的绘制位置。

参数：

- char[] text/String text：要绘制的字符串。
- int index：第一个要绘制的文字的索引。
- int count：要绘制的文字的个数，用来计算最后一个文字的位置，从第一个绘制的文字开始算起。
- float[] pos：要绘制的每个文字的具体位置，同样两两一组，如｛x1,y1,x2,y2,x3,y3,…｝。

示例：

```
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(80);

float []pos=new float[]{80,100,
       80,200,
       80,300,
```

```
        80,400};
    canvas.drawPosText("床前明月", pos, paint);
```

效果如下图所示。

3．沿路径绘制

```
    void drawTextOnPath (String text, Path path, float hOffset, float vOffset, Paint
paint)
    void drawTextOnPath (char[] text, int index, int count, Path path, float hOffset,
float vOffset, Paint paint)
```

这两个函数的区别就在于，第二个函数能实现截取一部分文字绘制。下面着重看一下 hOffset 和 vOffset 参数的含义。

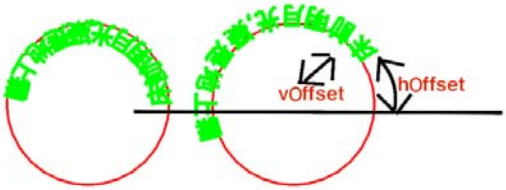
- float hOffset：与路径起始点的水平偏移量。
- float vOffset：与路径中心的垂直偏移量。

示例：

```
    Paint paint=new Paint();
    paint.setColor(Color.RED);
    paint.setStrokeWidth (5);
    paint.setTextSize(45);
    paint.setStyle(Paint.Style.STROKE);

    //先创建两条相同的圆形路径，并画出两条路径原形
    Path circlePath=new Path();
    circlePath.addCircle(220,300, 150, Path.Direction.CCW);//逆向绘制
    canvas.drawPath(circlePath, paint);  //绘制出路径原形

    Path circlePath2=new Path();
    circlePath2.addCircle(600,300, 150, Path.Direction.CCW);
    canvas.drawPath(circlePath2, paint); //绘制出路径原形

    //绘制原始文字与偏移文字
    String string="床前明月光,疑是地上霜";
    paint.setColor(Color.GREEN);
    //将 hOffset、vOffset 参数值全部设为 0，看原始状态是怎样的
    canvas.drawTextOnPath(string, circlePath, 0, 0, paint);
    //第二条路径，改变 hOffset、vOffset 参数值
    canvas.drawTextOnPath(string, circlePath2, 80, 30, paint);
```

在上面的代码中，我们先创建了两条相同的圆形路径，都包括路径创建方向（Path.Direction）和圆形大小；然后分别在 hOffset、vOffset 都为 0 时绘制出没有偏移时的初始状态；最后在另一条路径上绘制出 hOffset、vOffset 均不为 0 时的图像。

效果如下图所示。



从效果图中可以看出，hOffset 是与路径起始点的水平偏移量，而 yOffset 是路径中心的垂直偏移量。如果把圆拆开，拉成一条直线，那么 hOffset 显然是在 *X* 轴方向的偏移量，而 vOffset 是在 *Y* 轴方向的偏移量。

1.3.3　设置字体样式

在 Paint 中有一个函数是专门用来设置字体样式的。

```
Typeface setTypeface(Typeface typeface)
```

使用这个函数的前提是必须构造 Typeface 类的一个参数。

Typeface 是专门用来设置字体样式的类，通过 paint.setTypeface()函数来指定即将绘制的文字的字体样式。可以指定系统中的字体样式，也可以在自定义的样式文件中获取。在构建 Typeface 类时，可以指定所用样式的正常体、斜体、粗体等。如果在指定样式中没有相关文字的样式，就会用系统默认的样式来显示，一般默认是宋体。

1．使用系统中的字体样式

1）使用 Android 自带的字体样式

在 Typeface 类中保存着三种自带的字体样式：Typeface.SANS_SERIF、Typeface.MONOSPACE 和 Typeface.SERIF。我们可以直接使用这三种字体样式。

示例：

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setTypeface(Typeface.SERIF);
paint.setTextSize(50);

canvas.drawText("床前明月光",10,100,paint);
```

效果如下图所示。

由于这三种字体样式对中文的支持不是很好，所以，当遇到不支持的文字时，会使用系

统默认的样式来写。对中国用户而言，系统默认的字体一般是 DroidSansFallback，所以用这三种样式的文字写中文时是看不到差别的，我们一般不会使用这三种字体样式。值得一提的是，由于 Android 系统是可定制的，所以目前很多手机厂商所使用的字体已经不再是 Google 默认的，比如 MIUI V4 用的就是兰亭黑。

2）defaultFromStyle()函数

```
Typeface defaultFromStyle(int style)
```

该函数会根据字体样式获取对应的默认字体。参数 int style 的取值如下。

- Typeface.NORMAL：正常字体。
- Typeface.BOLD：粗体。
- Typeface.ITALIC：斜体。
- Typeface.BOLD_ITALIC：粗斜体。

如果系统默认的字体是宋体，那么，当指定 defaultFromStyle(Typeface.BOLD_ITALIC)时，获取的将是粗斜体的宋体样式。

示例：

```
Typeface typeface = Typeface.defaultFromStyle(Typeface.BOLD_ITALIC);
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setTypeface(typeface);
paint.setTextSize(50);

canvas.drawText("床前明月光",10,100,paint);
```

效果如下图所示。

3）create(String familyName, int style)函数

```
Typeface create(String familyName, int style)
```

该函数直接通过指定字体名来加载系统中自带的字体样式。如果字体样式不存在，则会用系统样式替代并返回。

示例：

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setTextSize(50);

String familyName = "宋体";
Typeface font = Typeface.create(familyName,Typeface.NORMAL);
paint.setTypeface(font);
canvas.drawText("床前明月光",10,100,paint);
```

效果如下图所示。

床前明月光

2．自定义字体样式

一般而言，我们不会指定系统自带的字体样式，因为除 Android 自带的三种字体样式以外，其他字体样式并不一定在每款手机上都有预装。所以，我们一般会选择加载自定义的字体文件来绘制文字，这样才不至于在每款手机上的表现不一样。

如果要自定义字体样式，就需要从外部字体文件中加载我们所需的字形，这时所使用的 Typeface 构造函数有如下三个：

```
Typeface createFromAsset(AssetManager mgr, String path)
Typeface createFromFile(String path)
Typeface createFromFile(File path)
```

很明显，既可以从 assets 文件夹中获取字体样式，也可以从指定的文件路径中获取字体样式。

下面讲一下从 assets 文件夹中获取字体样式的方法。

首先在 assets 文件夹下新建一个文件夹，命名为 fonts，然后将字体文件 jian_luobo.ttf 放入其中。

代码如下：

```
//自定义字体，迷你简萝卜
Paint paint=new Paint();
paint.setColor(Color.RED);
paint.setTextSize(60);

AssetManager mgr= mContext.getAssets();//得到 AssetManager
//根据路径得到 Typeface
Typeface typeface=Typeface.createFromAsset(mgr, "fonts/jian_luobo.ttf");
paint.setTypeface(typeface);
canvas.drawText("床前明月光,疑是地上霜",10,100, paint);
```

这里需要注意的是 Typeface.createFromAsset()函数的第二个参数，它传递的是 assets 文件夹下字体的完整路径，所以路径名和文件名必须完整。

效果如下图所示。

1.4　Region

Region 译为“区域”，顾名思义，区域是一块任意形状的封闭图形。

1.4.1 构造Region

1. 直接构造

```
public Region(Region region) //复制一个 Region 的范围
public Region(Rect r)        //创建一个矩形区域
public Region(int left, int top, int right, int bottom) //创建一个矩形区域
```

第一个构造函数通过其他 Region 来复制一个同样的 Region 变量。

第二、三个构造函数才是常用的，根据一个矩形或矩形的左上角点和右下角点构造出一个矩形区域。

示例：

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

    Paint paint = new Paint();
    paint.setStyle(Paint.Style.FILL);
    paint.setColor(Color.RED);

    Region region = new Region(new Rect(50,50,200,100));
    drawRegion(canvas,region,paint);
}
```

示例中构造出一个矩形的 Region 对象。由于 Canvas 中并没有用来画 Region 的方法，所以，如果我们想将 Region 画出来，就必须自己想办法。这里定义了一个 drawRegion()函数将整个 Region 画出来。drawRegion()函数的定义如下：

```
private void drawRegion(Canvas canvas,Region rgn,Paint paint)
{
    RegionIterator iter = new RegionIterator(rgn);
    Rect r = new Rect();

    while (iter.next(r)) {
        canvas.drawRect(r, paint);
    }
}
```

有关 drawRegion()函数的具体含义我们稍后讲解，在这里只需要知道我们自定义的 drawRegion()函数是可以将整个 Region 画出来的。

效果如下图所示。

从这里可以看出，Canvas 并没有提供针对 Region 的绘图方法，这就说明 Region 的本意并不是用来绘图的。对于上面构造的矩形填充，我们完全可以使用 Rect 来代替。

```
Paint paint = new Paint();
paint.setStyle(Paint.Style.FILL);
```

```
paint.setColor(Color.RED);

canvas.drawRect(new Rect(50,50,200,100),paint);
```

实现的效果与上面相同。

2．间接构造

间接构造主要是通过 public Region()的空构造函数与 set 系列函数相结合来实现的。

Region 的空构造函数：

```
public Region()
```

set 系列函数：

```
public void setEmpty()  //置空
public boolean set(Region region)
public boolean set(Rect r)
public boolean set(int left, int top, int right, int bottom)
public boolean setPath(Path path, Region clip)
```

注意：无论调用 set 系列函数的 Region 是不是有区域值，当调用 set 系列函数后，原来的区域值就会被替换成 set 系列函数里的区域值。

各函数的含义如下。

- setEmpty()：从某种意义上讲，置空也是一个构造函数，即将原来的一个区域变量变成空变量，再利用其他的 set 函数重新构造区域。
- set(Region region)：利用新的区域替换原来的区域。
- set(Rect r)：利用矩形所代表的区域替换原来的区域。
- set(int left, int top, int right, int bottom)：根据矩形的两个角点构造出矩形区域来替换原来的区域。
- setPath(Path path, Region clip)：根据路径的区域与某区域的交集构造出新的区域。

在这里主要讲解利用 setPath()函数构造不规则区域的方法，其他的几个函数使用难度都不大，就不再详细讲解了。

```
boolean setPath(Path path, Region clip)
```

参数：

- Path path：用来构造区域的路径。
- Region clip：与前面的 path 所构成的路径取交集，并将该交集设置为最终的区域。

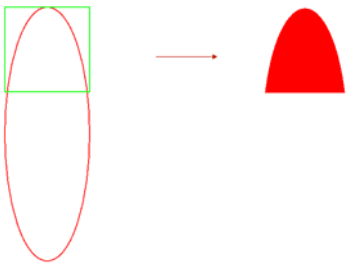
由于路径有很多种构造方法，而且可以轻易构造出非矩形的路径，因而摆脱了前面的构造函数只能构造矩形区域的限制。但这里有一个问题，即需要指定另一个区域来取交集。当然，如果想显示路径构造的区域，那么 Region clip 参数可以传入一个比 Path 范围大得多的区域，取完交集之后，当然就是 Path path 参数所对应的区域了。

示例：

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.FILL);
//构造一条椭圆路径
```

```
Path ovalPath = new Path();
RectF rect = new RectF(50, 50, 200, 500);
ovalPath.addOval(rect, Path.Direction.CCW);
//在 setPath()函数中传入一个比椭圆区域小的矩形区域，让其取交集
Region rgn = new Region();
rgn.setPath(ovalPath, new Region(50, 50, 200, 200));
//画出路径
drawRegion(canvas, rgn, paint);
```

效果如下图所示。



扫码看彩色图

左侧分别画出了所构造的椭圆和矩形，二者相交之后，所画出的 Region 对象是如右侧图像所示的椭圆上部分。

1.4.2 枚举区域——RegionIterator类

对于特定的区域，可以使用多个矩形来表示其大致形状。事实上，如果矩形足够小，一定数量的矩形就能够精确表示区域的形状。也就是说，一定数量的矩形所合成的形状也可以代表区域的形状。RegionIterator 类就实现了获取组成区域的矩形集的功能。其实 RegionIterator 类非常简单，它包含两个函数：一个构造函数和一个获取下一个矩形的函数。

（1）构造函数：根据区域构建对应的矩形集。

```
RegionIterator(Region region)
```

（2）获取下一个矩形，将结果保存在参数 Rect r 中。

```
boolean next(Rect r)
```

前面提到，由于在 Canvas 中没有直接绘制 Region 的函数，想要绘制一个区域，就只能通过 RegionIterator 类构造矩形集来逼近显示区域，所以 drawRegion()函数的具体实现如下：

```
private void drawRegion(Canvas canvas,Region rgn,Paint paint)
{
    RegionIterator iter = new RegionIterator(rgn);
    Rect r = new Rect();

    while (iter.next(r)) {
      canvas.drawRect(r, paint);
    }
}
```

首先根据区域构造一个矩形集，然后利用 next(Rect r)函数来逐个获取所有矩形并绘制出

来，最终得到的就是整个区域。如果我们想画一个椭圆区域，并且把画笔样式从 FILL 改为 STROKE，则效果更清楚。

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.STROKE);

//构造一条椭圆路径
Path ovalPath = new Path();
RectF rect = new RectF(50, 50, 200, 500);
ovalPath.addOval(rect, Path.Direction.CCW);

//构造椭圆区域
Region rgn = new Region();
rgn.setPath(ovalPath, new Region(50, 50, 200, 500));
drawRegion(canvas,rgn,paint);
```

在代码中，同样先构造了一条椭圆路径，然后在形成 Region 时传入一个与构造的椭圆区域相同大小的矩形，所以取交集之后的结果就是椭圆路径所对应的区域。效果如下图所示。

扫码看彩色图

从效果图中可以明显看出，在绘制 Region 对象时，其实就是先将其转换成矩形集，然后利用画笔将每个矩形画出来而已。

1.4.3　区域相交

前面说过，Region 不是用来绘图的，所以 Region 最重要的功能在区域的相交操作中。

1．union()函数

```
boolean union(Rect r)
```

该函数用于与指定矩形取并集，即将 Rect 所指定的矩形加入当前区域中。

示例：

```
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setStyle(Paint.Style.FILL);

Region region = new Region(10,10,200,100);
region.union(new Rect(10,10,50,300));
drawRegion(canvas,region,paint);
```

在上述代码中，先在横向、竖向分别画两个矩形区域，然后利用 union()函数将两个矩形区域合并。效果如下图所示。

2．区域操作

除通过 union()函数合并指定矩形以外，Region 还提供了如下几个更加灵活的操作函数。

系列方法一：

```
boolean op(Rect r, Op op)
boolean op(int left, int top, int right, int bottom, Op op)
boolean op(Region region, Op op)
```

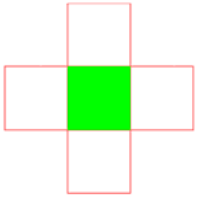
这些函数的含义是：用当前的 Region 对象与指定的一个 Rect 对象或者 Region 对象执行相交操作，并将结果赋给当前的 Region 对象。如果计算成功，则返回 true；否则返回 false。

其中最重要的是指定操作类型的 Op 参数。Op 参数值有如下 6 个。

```
public enum Op {
       DIFFERENCE(0),          //最终区域为 region1 与 region2 不同的区域
       INTERSECT(1),           // 最终区域为 region1 与 region2 相交的区域
       UNION(2),               //最终区域为 region1 与 region2 组合在一起的区域
       XOR(3),                 //最终区域为 region1 与 region2 相交之外的区域
       REVERSE_DIFFERENCE(4),//最终区域为 region2 与 region1 不同的区域
       REPLACE(5);             //最终区域为 region2 的区域
}
```

至于这 6 个参数值的具体含义，后面会给出具体的对比图，这里先举一个取交集的例子。

下图显示的是两个矩形相交的结果，横、竖两个矩形分别用描边画出来，相交区域用颜色填充。



绘图过程如下：

首先构造两个相交的矩形，并画出它们的轮廓。

```
//构造两个矩形
Rect rect1 = new Rect(100,100,400,200);
Rect rect2 = new Rect(200,0,300,300);

//构造一个画笔，画出矩形的轮廓
Paint paint = new Paint();
paint.setColor(Color.RED);
paint.setStyle(Style.STROKE);
```

```
paint.setStrokeWidth(2);

canvas.drawRect(rect1, paint);
canvas.drawRect(rect2, paint);
```

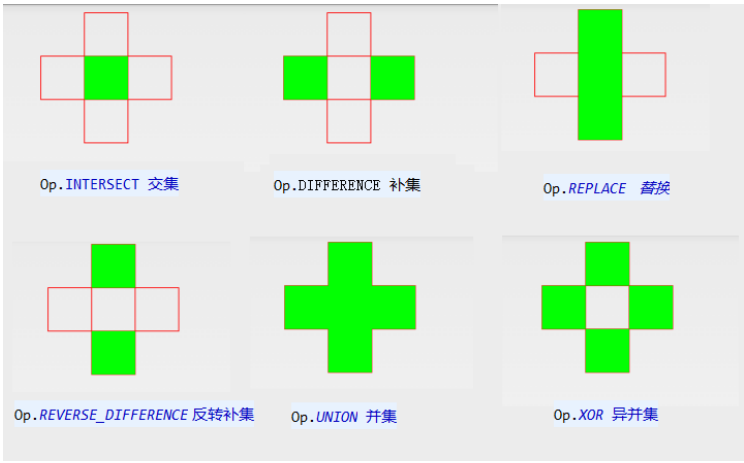
然后利用上面的两个 rect（rect1 和 rect2）来构造 Region，并在 rect1 的基础上取与 rect2 的交集。

```
//构造两个区域
Region region = new Region(rect1);
Region region2= new Region(rect2);

//取两个区域的交集
region.op(region2, Op.INTERSECT);
```

最后构造一个填充画笔，将所选区域用绿色填充。

```
Paint paint_fill = new Paint();
paint_fill.setColor(Color.GREEN);
paint_fill.setStyle(Style.FILL);
drawRegion(canvas, region, paint_fill);
```

其他参数的操作与此类似，其实只需要改动 region.op(region2, Op.INTERSECT);的 Op 参数值即可，这里就不一一列举了，给出操作后的对比图，如下图所示。



系列方法二：

```
boolean op(Rect rect, Region region, Op op)
boolean op(Region region1, Region region2, Region.Op op)
```

这两个函数允许我们传入两个 Region 对象进行区域操作，并将操作结果赋给当前的 Region 对象。同样，当操作成功时，返回 true；否则返回 false。

函数用法如下：

```
Region region1 = new Region(100,100,400,200);
Region region2 = new Region(200,0,300,300);

Region region = new Region();
region.op(region1,region2, Region.Op.INTERSECT);
```

在这里，将 region1、region2 相交的结果赋给 Region 对象。

1.4.4 其他函数

下面的这些函数使用难度并不大，仅列出各函数的含义，不再一一举例。

1. 几个判断方法

```
public boolean isEmpty();
```

该函数用于判断该区域是否为空

```
public boolean isRect();
```

该函数用于判断该区域是否是一个矩阵。

```
public boolean isComplex();
```

该函数用于判断该区域是否是多个矩阵的组合。

2. getBound 系列函数

getBound 系列函数用于返回一个 Region 的边界。

```
public Rect getBounds()
public boolean getBounds(Rect r)
```

这两个函数用于返回能够包裹当前路径的最小矩形。

```
public Path getBoundaryPath()
public boolean getBoundaryPath(Path path)
```

这两个函数用于返回当前矩形所对应的 Path 对象。

3. 是否包含

Region 中仍存在一系列的判断是否包含某个点或某个矩形的函数。

```
public boolean contains(int x, int y);
```

该函数用于判断该区域是否包含某个点。

```
public boolean quickContains(Rect r)
public boolean quickContains(int left, int top, int right,int bottom)
```

这两个函数用于判断该区域是否包含某个矩形。

4. 是否相交

```
public boolean quickReject(Rect r)
public boolean quickReject(int left, int top, int right, int bottom);
```

这两个函数用于判断该区域是否没有和指定矩形相交。

```
public boolean quickReject(Region rgn);
```

该函数用于判断该区域是否没有和指定区域相交。

5. 平移变换

```
public void translate(int dx, int dy)
```

该函数用于将 Region 对象向 *X* 轴平移 dx 距离，向 *Y* 轴平移 dy 距离，并将结果赋给当前的 Region 对象。*X* 轴向右是正方向，*Y* 轴向下是正方向。

```
public void translate(int dx, int dy, Region dst);
```

该函数用于将 Region 对象向 *X* 轴平移 dx 距离，向 *Y* 轴平移 dy 距离。与上一个函数不同的是，该函数将结果赋给 dst 对象，而当前 Region 对象的值保持不变。

1.5　Canvas（画布）

在 1.1 节中讲过 Canvas 是用来作画的画布，并且讲述了利用 Canvas 的 drawXXX 系列函数来绘制各种图形的方法。除可以在 Canvas 上面绘图以外，还可以对画布进行变换及裁剪等操作。

1.5.1　Canvas变换

1．平移（Translate）

Canvas 中有一个函数 translate()是用来实现画布平移的。画布的原始状态是以左上角点为原点，向右是 *X* 轴正方向，向下是 *Y* 轴正方向，如下图所示。

由于画布的左上角点为坐标轴的原点(0,0)，所以当平移画布以后，坐标系也同样会被平移。被平移后的画布的左上角点是新的坐标原点。translate()函数的原型如下：

```
void translate(float dx, float dy)
```

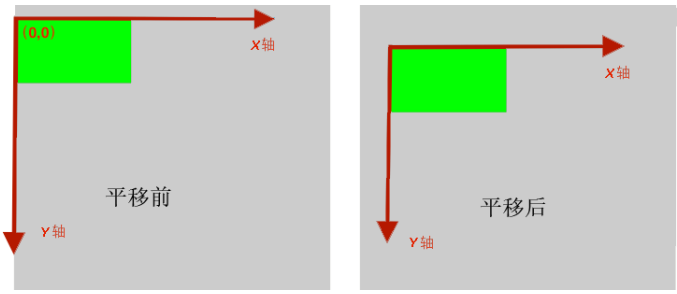
参数：

- float dx：水平方向平移的距离，正数为向正方向（向右）平移的量，负数为向负方向（向左）平移的量。
- float dy：垂直方向平移的距离，正数为向正方向（向下）平移的量，负数为向负方向（向上）平移的量。

示例：

```
protected void onDraw(Canvas canvas) {
    // TODO Auto-generated method stub
    super.onDraw(canvas);

    Paint paint = new Paint();
    paint.setColor(Color.GREEN);
    paint.setStyle(Style.FILL);

//  canvas.translate(100, 100);
    Rect rect1 = new Rect(0,0,400,220);
    canvas.drawRect(rect1, paint);
}
```

在上述代码中，先把 canvas.translate(100,100);注释掉，看原来矩形的位置；然后打开注释，看平移后的位置，对比如下图所示。



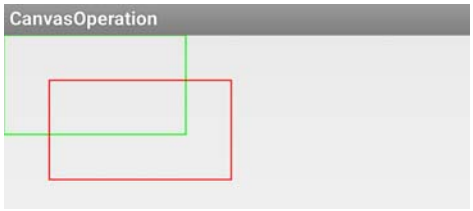
很明显，在平移后，同样绘制 Rect(0,0,400,220)，但是坐标系的起始点变成了平移后的原点。由此可见，在对 Canvas 进行变换时，坐标系的位置会随着 Canvas 左上角点的移动而移动。

2．屏幕显示与 Canvas 的关系

很多读者一直以为显示所绘图形的屏幕就是 Canvas，其实这是一种非常错误的理解，比如下面这段代码。在这段代码中，同一个矩形，在画布平移前后各画一次，结果会怎样？

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

    //构造两个画笔，一个红色，一个绿色
    Paint paint_green = generatePaint(Color.GREEN, Style.STROKE, 3);
    Paint paint_red   = generatePaint(Color.RED, Style.STROKE, 3);

    //构造一个矩形
    Rect rect1 = new Rect(0,0,400,220);

    //在平移画布前用绿色画笔画下边框
    canvas.drawRect(rect1, paint_green);

    //在平移画布后，再用红色画笔重新画下边框
    canvas.translate(100, 100);
    canvas.drawRect(rect1, paint_red);

}
private Paint generatePaint(int color,Paint.Style style,int width)  {
    Paint paint = new Paint();
    paint.setColor(color);
    paint.setStyle(style);
    paint.setStrokeWidth(width);
    return paint;
}
```

实际结果如下图所示。

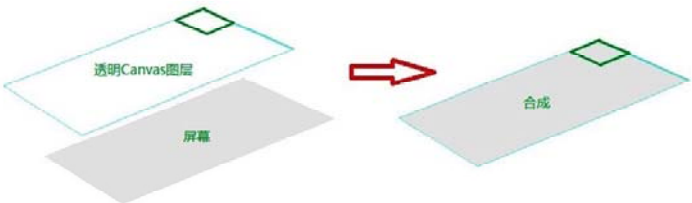


扫码看彩色图

为什么绿色框并没有移动？

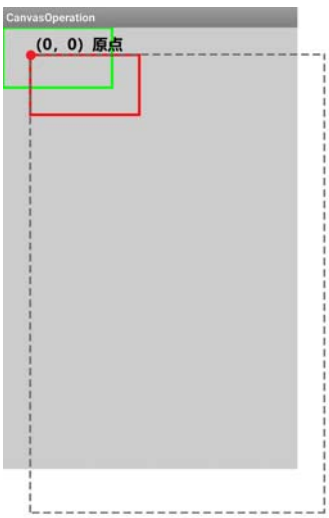
这是由于屏幕显示与 Canvas 根本不是一个概念！Canvas 是一个很虚幻的概念，相当于一个透明图层。每次在 Canvas 上画图时（调用 drawXXX 系列函数），都会先产生一个透明图层，然后在这个图层上画图，画完之后覆盖在屏幕上显示。所以，上述结果是经以下几个步骤形成的：

（1）在调用 canvas.drawRect(rect1, paint_green);时，产生一个 Canvas 透明图层，由于当时还没有对坐标系进行平移，所以坐标原点是(0,0)；在 Canvas 上画好之后，覆盖到屏幕上显示出来。过程如下图所示。



扫码看彩色图

（2）在调用 canvas.drawRect(rect1, paint_red);时，又会产生一个全新的 Canvas 透明图层，但此时画布坐标已经改变了，即分别向右和向下移动了 100 像素，所以此时的绘图方式如下图所示（合成视图，从上往下看的合成方式）。



上图展示了上层的 Canvas 图层与底部的屏幕的合成过程。由于 Canvas 已经平移了 100 像素，所以在画图时是以新原点来产生视图的，然后合成到屏幕上，这就是我们看到的最终结果。我们看到，当屏幕移动之后，有一部分超出了屏幕的范围，那超出范围的图像显不显示呢？当然不显示了！也就是说，在 Canvas 上虽然能画图，但超出了屏幕的范围，是不会显示的。当然，这里并没有超出显示范围。

下面对上述知识做一下总结：

（1）每次调用 drawXXX 系列函数来绘图时，都会产生一个全新的 Canvas 透明图层。

（2）如果在调用 drawXXX 系列函数前，调用平移、旋转等函数对 Canvas 进行了操作，那么这个操作是不可逆的。每次产生的画布的最新位置都是执行这些操作后的位置。

（3）在 Canvas 图层与屏幕合成时，超出屏幕范围的图像是不会显示出来的。

3．旋转（Rotate）

画布的旋转默认是围绕坐标原点来进行的。这里容易产生错觉，看起来是图片旋转了，其实我们旋转的是画布，以后在此画布上绘制的图形显示出来的时候看起来都是旋转的。rotate()函数有两个构造函数。

```
void rotate(float degrees)
void rotate(float degrees, float px, float py)
```

第一个构造函数直接输入旋转的度数，正数指顺时针旋转，负数指逆时针旋转，它的旋转中心点是原点(0,0)。

第二个构造函数除度数以外，还可以指定旋转的中心点坐标(px,py)。

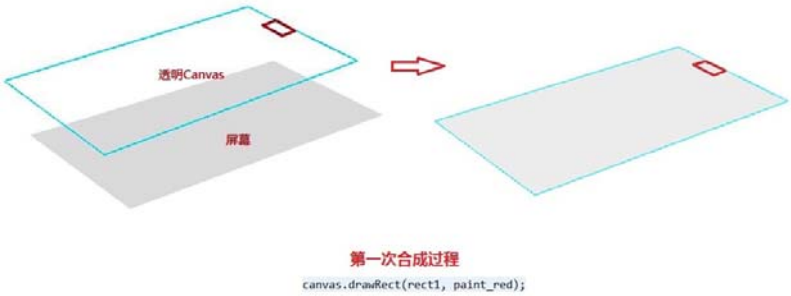
下面以第一个构造函数为例，旋转一个矩形。先画出未旋转前的图形，然后画出旋转后的图形。代码如下：

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


    Paint paint_green = generatePaint(Color.GREEN, Style.FILL, 5);
    Paint paint_red   = generatePaint(Color.RED, Style.STROKE, 5);

    Rect rect1 = new Rect(300,10,500,100);
    canvas.drawRect(rect1, paint_red);        //画出原轮廓

    canvas.rotate(30);                        //顺时针旋转画布
    canvas.drawRect(rect1, paint_green);      //画出旋转后的矩形
}
```

效果如下图所示。

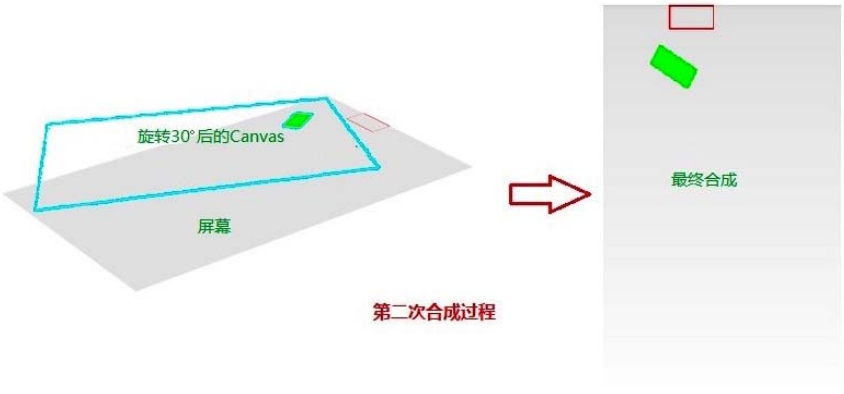
这个最终屏幕显示的构造过程如下：

下图显示的是第一次合成过程，此时仅仅调用 canvas.drawRect(rect1, paint_red); 画出原轮廓。



扫码看彩色图

下图显示的是第二次合成过程，先将 Canvas 沿正方向依原点旋转 30°，再与上面的屏幕合成，最后显示出复合效果。



扫码看彩色图

有关 Canvas 与屏幕的合成关系已经讲得足够详细了，后面几个操作 Canvas 的函数就不再一一讲述它们的合成过程了。

4．缩放（Scale）

该函数用于变更坐标轴密度，它有两个构造函数。

```
public void scale(float sx, float sy)
```

参数：

- float sx：水平方向伸缩的比例。假设原坐标轴的比例为 n，不变时为 1，变更后的 X 轴密度为 n×sx。所以，sx 是小数表示缩小，sx 是整数表示放大。
- float sy：垂直方向伸缩的比例。同样，sy 为小数表示缩小，sy 为整数表示放大。

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

这里多了两个参数 px,py，表示缩放中心位置。

下面以第一个构造函数为例，示例代码如下：

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

    Paint paint_green = generatePaint(Color.GREEN, Style.STROKE, 5);
    Paint paint_red   = generatePaint(Color.RED, Style.STROKE, 5);
```

```
        Rect rect1 = new Rect(10,10,200,100);
        canvas.drawRect(rect1, paint_green);

        canvas.scale(0.5f, 1);
        canvas.drawRect(rect1, paint_red);
    }
```

效果如下图所示。

扫码看彩色图

图中，绿框是原坐标轴密度图形，红框是 *X* 轴密度缩小到 0.5 倍之后显示的图形。

5．扭曲（Skew）

它的构造函数如下：

```
void skew(float sx, float sy)
```

参数：

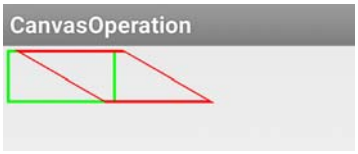
- float sx：将画布在 *X* 轴方向上倾斜相应的角度，sx 为倾斜角度的正切值。
- float sy：将画布在 *Y* 轴方向上倾斜相应的角度，sy 为倾斜角度的正切值。

注意：这里都是倾斜角度的正切值，比如，在 *X* 轴方向上倾斜 60°，tan60=1.732。

示例：

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

    Paint paint_green = generatePaint(Color.GREEN, Style.STROKE, 5);
    Paint paint_red   = generatePaint(Color.RED, Style.STROKE, 5);

    Rect rect1 = new Rect(10,10,200,100);

    canvas.drawRect(rect1, paint_green);
    canvas.skew(1.732f,0);//X 轴倾斜 60°，Y 轴不变
    canvas.drawRect(rect1, paint_red);
}
```

效果如下图所示。



6．裁剪画布（clip 系列函数）

裁剪画布是指利用 clip 系列函数，通过与 Rect、Path、Region 取交、并、差等集合运算来获得最新的画布形状。除调用 save()、restore()函数以外，这个操作是不可逆的，一旦 Canvas 被裁剪，就不能恢复。

注意：在使用 clip 系列函数时，需要禁用硬件加速功能，在 1.5.3 节中有具体用法。

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

有关硬件加速的知识，请参考第 5 章。

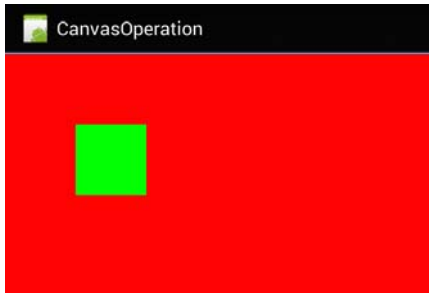
clip 系列函数如下：

```
boolean clipPath(Path path)
boolean clipPath(Path path, Region.Op op)
boolean clipRect(Rect rect, Region.Op op)
boolean clipRect(RectF rect, Region.Op op)
boolean clipRect(int left, int top, int right, int bottom)
boolean clipRect(float left, float top, float right, float bottom)
boolean clipRect(RectF rect)
boolean clipRect(float left, float top, float right, float bottom, Region.Op op)
boolean clipRect(Rect rect)
boolean clipRegion(Region region)
boolean clipRegion(Region region, Region.Op op)
```

以上就是根据 Rect、Path、Region 来获得最新画布形状的函数，使用难度都不大，就不再一一讲述了。下面以 clipRect()函数为例来讲解具体应用。

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

    canvas.drawColor(Color.RED);
    canvas.clipRect(new Rect(100, 100, 200, 200));
    canvas.drawColor(Color.GREEN);
}
```

效果如下图所示。



先把背景色涂成红色，显示在屏幕上；然后裁剪画布；最后将最新的画布涂成绿色。可见，绿色部分只有一小块，而不再是整个屏幕。

1.5.2　画布的保存与恢复

1．save()和 restore()函数

前面介绍的所有对画布的操作都是不可逆的，这会造成很多麻烦。比如，为了实现一些效果不得不对画布进行操作，但操作完了，画布状态也改变了，这会严重影响到后面的画图操作。如果能对画布的大小和状态（旋转角度、扭曲等）进行实时保存和恢复就好了。本小节就讲述与画布的保存和恢复相关的函数——save()和 restore()。这两个函数的原型如下：

```
int save()
void restore()
```

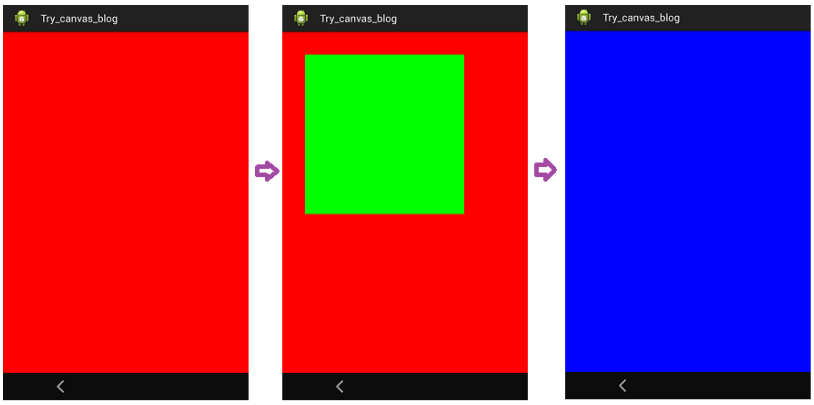
这两个函数没有任何参数，使用很简单。

- save()：每次调用 save()函数，都会先保存当前画布的状态，然后将其放入特定的栈中。
- restore()：每次调用 restore()函数，都会把栈中顶层的画布状态取出来，并按照这个状态恢复当前的画布，然后在这个画布上作画。

为了更清晰地显示这两个函数的作用，我们举一个例子，代码如下：

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

    canvas.drawColor(Color.RED);

    //保存当前画布大小，即整屏
    canvas.save();

    canvas.clipRect(new Rect(100, 100, 800, 800));
    canvas.drawColor(Color.GREEN);

    //恢复整屏画布
    canvas.restore();

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

在这个例子中，首先将整个画布填充为红色，在将画布状态保存之后，将画布裁剪并填充为绿色，然后将画布还原以后填充为蓝色。整个填充过程如下图所示。



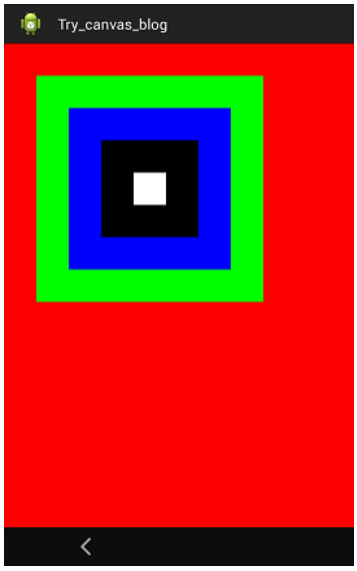
扫码看彩色图

下面通过一个多次利用 save()、restore()函数的例子来讲述有关保存画布状态的栈的概念。代码如下：

```
protected void onDraw(Canvas canvas) {
    // TODO Auto-generated method stub
    super.onDraw(canvas);
```

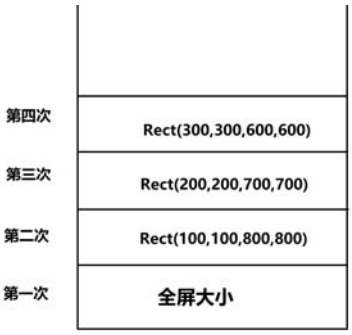
```
        canvas.drawColor(Color.RED);
        //保存的画布大小为全屏幕大小
        canvas.save();

        canvas.clipRect(new Rect(100, 100, 800, 800));
        canvas.drawColor(Color.GREEN);
        //保存的画布大小为 Rect(100, 100, 800, 800)
        canvas.save();

        canvas.clipRect(new Rect(200, 200, 700, 700));
        canvas.drawColor(Color.BLUE);
        //保存的画布大小为 Rect(200, 200, 700, 700)
        canvas.save();

        canvas.clipRect(new Rect(300, 300, 600, 600));
        canvas.drawColor(Color.BLACK);
        //保存的画布大小为 Rect(300, 300, 600, 600)
        canvas.save();

        canvas.clipRect(new Rect(400, 400, 500, 500));
        canvas.drawColor(Color.WHITE);
    }
```

效果如下图所示。



扫码看彩色图

在这段代码中，共调用了 4 次 save()函数。前面提到，每调用一次 save()函数就会将当前的画布状态保存到栈中，并将画布涂上一种颜色来显示当前画布的大小，所以这 4 次调用 save()函数所保存的栈的状态如下图所示。

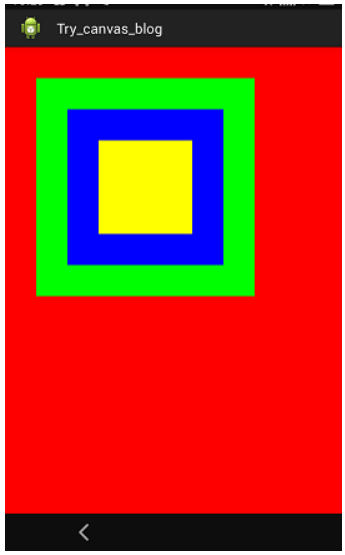


注意：在第四次调用 save()函数之后，还对画布进行了 canvas.clipRect(new Rect(400, 400, 500, 500));操作，并将当前画布填充为白色。

如果现在调用 restore()函数来还原画布，则会把栈顶的画布状态取出来，作为当前绘图的画布。代码如下：

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

    //各种 save 操作
    ...

    //将栈顶的画布状态取出来，作为当前画布，并填充为黄色
    canvas.restore();
    canvas.drawColor(Color.YELLOW);
}
```

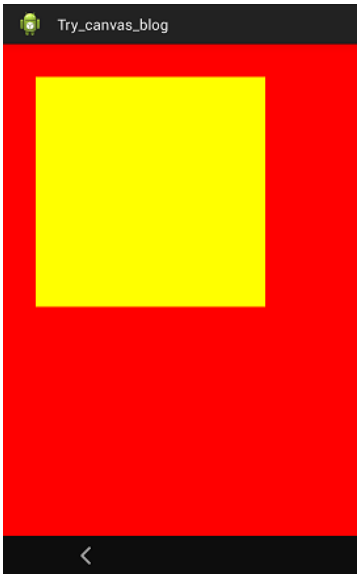
在示例代码中，我们省略了各种 save 操作的代码，在执行了 4 次保存画布操作以后，第一次调用 restore()函数将栈顶的画布状态取出来，并填充为黄色。效果如下图所示。



扫码看彩色图

如果连续三次调用 restore()函数会怎样呢？连续三次调用 restore()函数，会连续三次出栈，然后把第三次出栈的画布状态当作当前画布，也就是 Rect(100, 100, 800, 800)。代码如下：

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

    //各种 save 操作
    ...

    //连续三次出栈，将最后一次出栈的画布状态作为当前画布，并填充为黄色
    canvas.restore();
    canvas.restore();
    canvas.restore();
    canvas.drawColor(Color.YELLOW);
}
```



扫码看彩色图

2．restoreToCount(int saveCount)函数

上面讲述了 save()和 restore()函数的用法，我们可以多次调用 save()函数，但每次调用 restore()函数，只会将顶层的画布状态出栈，有时可能只需要用到特定的画布，这就需要多次出栈。为了解决这个问题，Google 提供了另一个出栈函数 restoreToCount(int saveCount)。

首先来看 save()函数的完整声明。

```
public int save();
```

在利用 save()函数保存画布时，会有一个 int 类型的返回值。该返回值是当前所保存的画布所在栈的索引。

然后来看 restoreToCount()函数的完整声明。

```
public void restoreToCount(int saveCount);
```

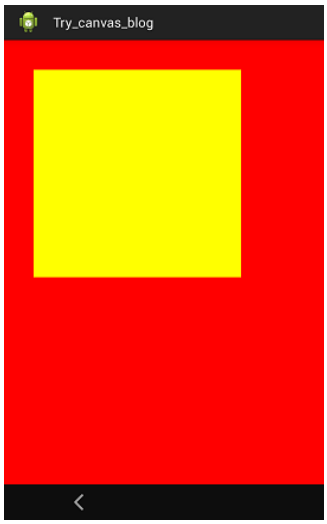
而 restoreToCount()函数的用法就是一直出栈，直到指定索引的画布出栈为止，即将指定索引的画布作为当前画布。

对连续三次出栈的代码加以改造，调用 restoreToCount()函数来直接拿到原本需要连续三次出栈的画布。代码如下：

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

    canvas.drawColor(Color.RED);
    //保存的画布大小为全屏幕大小
    int c1 = canvas.save();

    canvas.clipRect(new Rect(100, 100, 800, 800));
    canvas.drawColor(Color.GREEN);
    //保存的画布大小为 Rect(100, 100, 800, 800)
    int c2 = canvas.save();

    canvas.clipRect(new Rect(200, 200, 700, 700));
    canvas.drawColor(Color.BLUE);
    //保存的画布大小为 Rect(200, 200, 700, 700)
    int c3 = canvas.save();

    canvas.clipRect(new Rect(300, 300, 600, 600));
    canvas.drawColor(Color.BLACK);
    //保存的画布大小为 Rect(300, 300, 600, 600)
    int c4 = canvas.save();

    canvas.clipRect(new Rect(400, 400, 500, 500));
    canvas.drawColor(Color.WHITE);

    //连续三次出栈，将最后一次出栈的画布状态作为当前画布，并填充为黄色
    canvas.restoreToCount(c2);
    canvas.drawColor(Color.YELLOW);
}
```

在每次调用 save()函数时，都将当前保存画布所在栈的索引保存起来；在恢复时，直接利用 c2 的索引，连续三次出栈，直到取出 c2 所保存的画布为止。

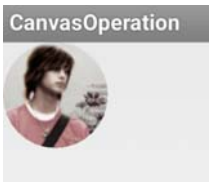
效果如下图所示，与前面的例子相同。



其实，保存画布还有很多种方法，我们在第 6 章中将会详细讲述。

1.5.3　示例一：圆形头像

裁剪画布的一个重要用法就是根据一张正方形的图片产生一个圆形头像，效果如下图所示。



首先进行初始化。

```
public class CustomCircleView extends View {
    private Bitmap mBmp;
    private Paint mPaint;
    private Path mPath;

    public CustomCircleView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    private void init(){
        setLayerType(LAYER_TYPE_SOFTWARE,null);
        mBmp = BitmapFactory.decodeResource(getResources(),R.drawable.avator);
        mPaint = new Paint();
        mPath = new Path();
        int width = mBmp.getWidth();
        int height = mBmp.getHeight();
        mPath.addCircle(width/2,height/2,width/2, Path.Direction.CCW);
    }

    ...
}
```

前面说过，在使用 clip 系列函数时，要禁用硬件加速功能。

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

如果不禁用硬件加速功能，则将不会产生任何效果。

然后利用 BitmapFactory.decodeResource()函数从本地 res 文件夹中提取一个 Bitmap（位图）文件。

接着根据位图文件的大小，构造一条与图像大小相同的圆形路径。

在绘图时，先将画布裁剪成圆形，再将位图画上去。

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

    canvas.save();
    canvas.clipPath(mPath);
    canvas.drawBitmap(mBmp,0,0,mPaint);
```

```
        canvas.restore();
    }
```

需要注意的是，在对画布进行变换或者裁剪操作以后，需要利用 save()和 restore()函数将它们复原。当然，这里在裁剪画布以后，并没有进行其他操作，所以不添加复原代码也不会有任何影响。

最后，直接在 Activity 的布局 XML 中引入即可。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent>

    <com.harvic.myapp.CustomCircleView
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

注意：目前，所有的自定义控件在被引入布局中时，layout_width 和 layout_height 属性的值都是 match_parent，在第 9 章中将详细讲解如何自测量控件大小以使用 wrap_content 属性。

1.5.4 示例二：裁剪动画

本小节再举一个裁剪 Region 的例子。在前面讲解 Region 的时候多次提到，Region 并不是用来画图的，它的主要作用就是裁剪画布。本小节将实现一个切割出现图片的动画。

动画截图如下图所示。

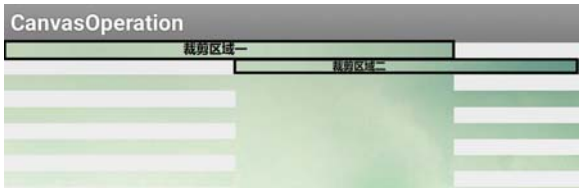
扫码看动画完整过程

1. 原理

其实，这个动画的原理很简单，就是每次将裁剪区域变大，在裁剪区域内的图像就会显

示出来，而裁剪区域之外的图像不会显示。而关键问题在于如何计算裁剪区域。

再来看一下动画截图，如下图所示。



这里很容易给大家造成错觉，以为黑色条是裁剪区域。其实不然，裁剪画布，在裁剪画布内的区域都是显示出来的，所以显示出来的区域才是裁剪区域。从图示中可以看出，有两个裁剪区域。

裁剪区域一：从左向右，逐渐变大。假设宽度是 clipWidth，高度是 CLIP_HEIGHT，那么裁剪区域一所对应的 Rect 对象如下：

```
Rect(0, 0, clipWidth, CLIP_HEIGHT);
```

裁剪区域二：从右向左，同样逐渐变大，它的宽度、高度都与裁剪区域一相同。但它是从右向左变化的，假设图片的宽度是 width，那么裁剪区域二所对应的 Rect 对象如下：

```
Rect(width - clipWidth, CLIP_HEIGHT, width, 2* CLIP_HEIGHT)
```

2．示例代码

在了解了动画原理以后，来看一下具体的代码实现。

首先进行初始化。

```
public class ClipRgnView extends View {
    private Bitmap mBitmap;
    private int clipWidth = 0;
    private int width;
    private int heigth;
    private static final int CLIP_HEIGHT = 30;
    private Region mRgn;

    public ClipRgnView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init();
    }

    private void init(){
        setLayerType(LAYER_TYPE_SOFTWARE,null);
        mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.
scenery);
        width = mBitmap.getWidth();
        heigth = mBitmap.getHeight();
        mRgn = new Region();
    }

    ...
}
```

前面说过，不要在 onDraw()函数中创建变量，所以，在 onDraw()函数中用到的有关函数都需要提前创建并初始化。

然后在 onDraw()函数中计算裁剪区域并绘图。

```
    @Override
    protected void onDraw(Canvas canvas) {
        mRgn.setEmpty();
        int i = 0;
        while (i * CLIP_HEIGHT <= heigth) {//计算 clip 的区域
            if (i % 2 == 0) {
                mRgn.union(new Rect(0, i * CLIP_HEIGHT, clipWidth, (i + 1) * CLIP_
HEIGHT));
            } else {
                mRgn.union(new Rect(width - clipWidth, i * CLIP_HEIGHT, width,
(i + 1) * CLIP_HEIGHT));
            }
            i++;
        }

        canvas.clipRegion(mRgn);
        canvas.drawBitmap(mBitmap, 0, 0, new Paint());
        if (clipWidth > width){
            return;
        }
        clipWidth += 5;

        invalidate();
    }
```

我们现在还没有学习有关 Animation（动画）的知识，所以暂时通过调用 invalidate()函数的方式来重复触发 onDraw()函数，然后在 onDraw()函数中计算需要裁剪的画布。

在上述代码中，首先，由于 mRgn 对象是每次都复用的，所以，在每次计算裁剪区域前，都需要调用 mRgn.setEmpty()函数将区域置空。

其次，根据计算裁剪区域的原理循环计算图片中每条间隔的裁剪区域并添加到 mRgn 对象中。

```
    while (i * CLIP_HEIGHT <= heigth) {//计算 clip 的区域
        if (i % 2 == 0) {
            mRgn.union(new Rect(0, i * CLIP_HEIGHT, clipWidth, (i + 1) *
CLIP_HEIGHT));
        } else {
            mRgn.union(new Rect(width - clipWidth, i * CLIP_HEIGHT, width, (i + 1) *
CLIP_HEIGHT));
        }
        i++;
    }
```

最后，将图片绘制在裁剪过的画布上，并渐变增大裁剪区域。

```
    canvas.clipRegion(mRgn);
    canvas.drawBitmap(mBitmap, 0, 0, new Paint());
    clipWidth += 5;
```

需要注意的是，当裁剪区域超过图像大小时，表示当前图像已经完全被绘制出来，可以暂停当前的绘制，以免浪费 CPU 资源。

```
if (clipWidth > width){
    return;
}
```

至此，有关 Canvas 的基础知识就介绍完了，针对 Canvas 的操作在以后的绘图中会经常用到，读者务必掌握。

1.6　控件的使用方法

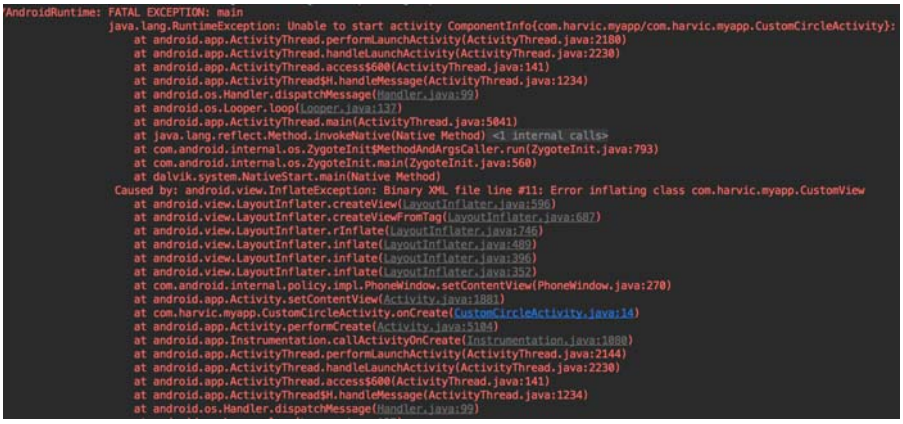
1.6.1　控件概述

在自定义一个派生自 View 或 ViewGroup 类的控件时，必须实现一个构造函数。有三个构造函数供我们选择。

比如，自定义一个 CustomView 控件，我们可以选择实现它的三个构造函数。

```
public class CustomView extends View {
    public CustomView(Context context) {
        super(context);
    }

    public CustomView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public CustomView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }
}
```

其实每种构造函数都是在特定的使用情景下所必须实现的，否则将会报 inflate 错误，如下图所示。



一个控件可以通过引入控件名所对应的 XML 的方式在布局中直接使用，也可以在代码中动态添加。下面分别来看一下通过 XML 引入控件与动态添加控件所使用的构造函数的区别。

1.6.2 通过XML引入控件

首先自定义一个控件，并在控件中画一个矩形。

```
public class CustomView extends View {
    private String TAG = "customView";
    public CustomView(Context context) {
        super(context);
        Log.d(TAG,"context");
    }

    public CustomView(Context context, AttributeSet attrs) {
        super(context, attrs);
        Log.d(TAG,"attrs");
    }

    public CustomView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        Log.d(TAG,"defStyle");
    }

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

        Paint paint = new Paint();
        paint.setColor(Color.RED);

        canvas.drawRect(0,0,200,100,paint);
    }
}
```

在每个构造函数中都贴上了不同的标签，然后将控件添加到布局代码中。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <com.harvic.myapp.CustomView
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</LinearLayout>
```

运行效果如下图所示。

日志如下：

```
11-06 03:00:34.008 21876-21876/com.harvic.myapp D/customView: attrs
```

从日志中可以看出，通过 XML 引入控件，所调用的构造函数如下：

```
public CustomView(Context context, AttributeSet attrs) {
    super(context, attrs);
}
```

如果要通过 XML 引入控件，就必须实现这个构造函数；否则会报错。

1.6.3　动态添加控件

1．概述

有下面这样一个布局，动态生成一个 CustomView 控件，并添加到 LinearLayout 中。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="自定义控件四部曲"/>
</LinearLayout>
```

这里需要注意的是，我们给要添加控件的根节点添加了 ID，可以方便地在代码中利用 findViewById 来找到它对应的实例，以便将控件添加为它的子节点。

```
public class CustomViewActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.custom_view_activity);

        LinearLayout rootView = (LinearLayout)findViewById(R.id.root);
        CustomView customView = new CustomView(this);
        LinearLayout.LayoutParams layoutParams = new
LinearLayout.LayoutParams (LinearLayout.LayoutParams.MATCH_PARENT,
LinearLayout.LayoutParams.MATCH_PARENT);
        rootView.addView(customView,layoutParams);
    }
}
```

在添加子控件时，需要先找到要添加子控件的节点的实例，这里先找到 LinearLayout 的实例，然后通过 rootView.addView(customView,layoutParams)函数将生成的 CustomView 控件的实例添加到 LinearLayout 中。效果如下图所示。

这里用到了两个新的概念：LayoutParams 和 addView，下面一一讲解。

2．LayoutParams

在 XML 中添加控件时，layout_width 和 layout_height 是必须设置的属性，它们的取值分别有 fill_parent、match_parent、wrap_content 和具体值。很明显，它们的意思是告诉父控件当前控件的布局样式。

同样，LayoutParams 的作用就是设置控件的宽和高，对应的是 XML 中的 layout_width 和 layout_height 属性。

LayoutParams 有三个构造函数。

```
public LayoutParams(int width, int height)
public LayoutParams(Context c, AttributeSet attrs)
public LayoutParams(LayoutParams source)
```

第一个构造函数用于指定具体的宽和高，取值有 LayoutParams.MATCH_PARENT、LayoutParams.FILL_PARENT 和具体值。比如，指定 LayoutParams(LayoutParams.MATCH_PARENT,500);就是指 layout_width 设置为 match_parent，layout_height 设置为 500px。

第二、三个构造函数都不常用。第二个构造函数用于从 AttributeSet 中提取出 layout_width、layout_height 等各属性的值。有关 AttributeSet 的知识，将在第 9 章自定义控件属性时具体讲述。第三个构造函数更简单，直接从一个现成的 LayoutParams 中复制一份。

前面在添加 LayoutParams 时使用的代码是 new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout.LayoutParams.MATCH_PARENT);，而这里用的是 LinearLayout.LayoutParams;。

其实，LinearLayout、FrameLayout、RelativeLayout 都有各自的 LayoutParams，它们都派生自 ViewGroup.LayoutParams 类。那么问题来了：不就是指定宽和高吗，为什么每个容器类控件又各自实现了一遍 LayoutParams 呢？

虽然我们在自定义控件时常用的方法是 public LayoutParams(int width, int height)，即只指定宽和高，但 LayoutParams 的作用却不止于此，public LayoutParams(Context c, AttributeSet attrs)函数可以从 XML 中提取出各种属性所对应的值。因为各个容器所对应的布局属性是不一样的，比如 ReleativeLayout 就有特有的 layout_alignBottom、layout_alignParentTop 等属性，这些属性所对应值的提取都是在 public LayoutParams(Context c, AttributeSet attrs)函数中进行的。

既然每个容器类控件都会实现一套 LayoutParams 类，那么，我们在动态添加布局时，如

何选择使用哪一套 LayoutParams 类呢？

前面讲过，在 XML 中添加一个控件时，layout_width 和 layout_height 是必须设置的属性。它们的意思是告诉父控件当前控件的布局样式，所以，父控件是什么就使用它对应的 LayoutParams。

比如，我们需要将自定义的 CustomView 添加到 LinearLayout 中，所以使用的是 new LinearLayout.LayoutParams(LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout.LayoutParams.MATCH_PARENT);。

如果我们用错了，则会怎样？

如果我们要将控件添加到 LinearLayout 布局中，却使用了 RelatvieLayout.LayoutParams，比如：

```
    LinearLayout rootView = (LinearLayout)findViewById(R.id.root);
    CustomView customView = new CustomView(this);
    RelativeLayout.LayoutParams layoutParams = new RelativeLayout.LayoutParams
(RelativeLayout.LayoutParams.MATCH_PARENT,RelativeLayout.LayoutParams.MATCH_
PARENT);
    rootView.addView(customView,layoutParams);
```

在 addView()函数中，会把 RelativeLayout.LayoutParams 转换成 LinearLayout.LayoutParams。如果能够转换，则不会报错；如果不能转换，则会报转换错误。是否能够转换，就看它们是否具有相同的属性。

当然，这里是不会报转换错误的，因为 LinearLayout.LayoutParams 和 RelativeLayout.LayoutParams 都具有 layout_width 和 layout_height 属性。

RelativeLayout 要比 LinearLayout 特殊，因为它不仅要设置 layout_width 和 layout_height 属性，还需要设置相对属性，比如 layout_alignBottom、layout_alignParentTop 等。

在代码中动态设置这些属性的方法是通过 RelativeLayout.LayoutParams 的 addRule()函数。该函数的声明如下：

```
    public void addRule(int verb, int anchor)
```

我们对上面示例的布局进行改造，将根节点设置为 RelativeLayout。

```
    <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/root"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical">

        <TextView
            android:id="@+id/text"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="自定义控件四部曲" />
    </RelativeLayout>
```

在代码中，我们将 CustomView 实例添加到 TextView 控件的右侧。这里注意一下，我们给 TextView 控件设置了 ID 为“@+id/text”。

```
    RelativeLayout rootView = (RelativeLayout) findViewById(R.id.root);
    CustomView customView = new CustomView(this);
    RelativeLayout.LayoutParams layoutParams = new RelativeLayout.LayoutParams
(RelativeLayout.LayoutParams.MATCH_PARENT, RelativeLayout.LayoutParams.MATCH_
PARENT);
    layoutParams.addRule(RelativeLayout.RIGHT_OF,R.id.text);
    rootView.addView(customView, layoutParams);
```

可见，addRule()函数的第一个参数是指 RelativeLayout 的布局属性，第二个参数是指相对于哪个控件 ID 来布局，这里对应的 XML 属性是 android:layout_toRightOf="@id/text"。除 RelativeLayout.RIGHT_OF 以外，其他属性都是以 RelativeLayout 静态变量的形式提供的，比如 RelativeLayout.ALIGN_PARENT_LEFT、RelativeLayout.ABOVE。添加其他属性依然通过 addRule()函数，用法一样，这里就不再赘述了。

下面再来看一下如何添加其他布局参数。

1）设置 margin

示例：

```
    LinearLayout.LayoutParams lp = new LinearLayout.LayoutParams(LinearLayout.
LayoutParams.WRAP_CONTENT, LinearLayout.LayoutParams.WRAP_CONTENT);
    lp.setMargins(10, 20, 30, 40);
    imageView.setLayoutParams(lp);
```

2）设置 layout_weight（方法一）

对于 LinearLayout 而言，它具有一个非常特殊的属性 layout_weight；对于 LinearLayout.LayoutParams 而言，它具有一个额外的构造函数，用于设置 layout_weight 属性，该函数的声明如下：

```
    public LayoutParams(int width, int height, float weight)
```

示例：

```
    TextView tv_like = new TextView(this);
    LinearLayout.LayoutParams LP_LIKE_MW = new LinearLayout.LayoutParams
(LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout.LayoutParams.WRAP_
CONTENT, 1.0f);
    tv_like.setText("赞(8)");
    tv_like.setTextSize(16);
    layout_sub_Lin.addView(tv_like, LP_LIKE_MW);
```

3）设置 layout_weight（方法二）

设置 layout_weight 属性还有另一种方法，如下：

```
    LinearLayout rootView = (LinearLayout) findViewById(R.id.root);
    CustomView customView = new CustomView(this);
    LinearLayout.LayoutParams layoutParams = new LinearLayout.LayoutParams
(LinearLayout.LayoutParams.MATCH_PARENT, LinearLayout.LayoutParams.MATCH_
PARENT);
    layoutParams.weight = 1.0f;
    rootView.addView(customView, layoutParams);
```

即直接通过 layoutParams.weight = 1.0f;赋值。

4）设置 layout_gravity

当我们给动态控件添加 layout_gravity 属性时，使用的方法如下：

```
    LinearLayout rootView = (LinearLayout) findViewById(R.id.root);
    LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(LayoutParams.
WRAP_CONTENT, LayoutParams.FILL_PARENT);
    params.gravity = Gravity.TOP;

    Button button = new Button(this);
    rootView.addView(button, layoutParams);
```

同样使用 LinearLayout.LayoutParams 的参数直接赋值。在 LinearLayout.LayoutParams 中有两个变量：gravity 和 weight。

值得注意的是，params.gravity 的取值有 Gravity.TOP、Gravity.BOTTOM、Gravity.LEFT、Gravity.RIGHT、Gravity.CENTER_VERTICAL、Gravity.CENTER_HORIZONTAL 等，这些属性值可以通过“|”（或）运算符合并。比如，垂直居底并且水平居中的写法是：Gravity.BOTTOM | Gravity.CENTER_HORIZONTAL。

5）设置 android:gravity

对于容器类和派生自 TextView 类的控件而言，都会具有 android:gravity 属性。如果我们不设置 android:gravity 属性，则默认居左上显示。设置 android:gravity 属性的示例代码如下：

```
    RelativeLayout rootView = (RelativeLayout) findViewById(R.id.root);
    RelativeLayout.LayoutParams layoutParams = new
RelativeLayout.LayoutParams(RelativeLayout.LayoutParams.WRAP_CONTENT,200);
    layoutParams.addRule(RelativeLayout.RIGHT_OF, R.id.text);

    Button button = new Button(this);
    button.setGravity(Gravity.TOP);
    button.setText("btn");
    rootView.addView(button, layoutParams);
    rootView.setGravity(Gravity.TOP | Gravity.CENTER_HORIZONTAL);
```

在上面的代码中，我们分别给动态生成的 Button 和根布局 RelativeLayout 都设置了 button.setGravity(Gravity.TOP);。对于 Button 而言，其中的文字将居顶显示；而对于 RelativeLayout 而言，其中的控件将垂直居顶且水平居中显示。

效果如下图所示。

从效果图中可以看出，对于 RelativeLayout 而言，其中的所有控件都垂直居顶且水平居中显示。对于 Button 而言，由于只设置了其中的文字居顶显示，所以可以明显看出，文字的布局在垂直方向是居顶的；但由于没有设置水平方向，所以默认居左显示。

3．addView

我们动态添加控件都是通过 addView 来实现的，addView 是 ViewGroup 类中的一个函数，它有 5 个构造函数。

```
public void addView(View child)
```

在节点末尾添加一个 View 控件，布局使用默认布局，即

```
layout_width=wrap_content, layout_height=wrap_content;
public void addView(View child, int index)
```

在指定位置添加一个 View 控件，index 的取值有−1、0 和正数。当取值为−1 时，表示在末尾添加一个 View 控件，此时的效果就与 addView(View child)相同；当取值为 0 时，表示在容器顶端添加一个 View 控件；当取值为正数时，表示在对应的索引位置插入一个 View 控件。

```
public void addView(View child, LayoutParams params)
```

这个构造函数在上面的例子中一直在使用，它允许我们自定义布局参数。

```
public void addView(View child, int index, LayoutParams params)
```

这个函数不仅允许我们自定义布局参数，也允许我们指定添加控件的位置。index 的取值同样是−1、0 和正数，其含义与 addView(View child, int index)中的 index 参数相同。

```
public void addView(View child, int width, int height)
```

我们在指定 LayoutParams 参数时一般只指定宽和高，这个函数简化了我们的需求。当我们只需要指定宽和高时，就可以直接使用这个函数。它的内部会利用我们传入的 width 和 height 属性来构造出一个 LayoutParams 对象。

第11章 Matrix与坐标变换

哪有什么一蹴而就，不过是在点滴积累。

11.1 矩阵运算

11.1.1 矩阵的加法与减法

1．运算规则

设矩阵 $\boldsymbol{A}=\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$，$\boldsymbol{B}=\begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}$，则 $\boldsymbol{A}\pm\boldsymbol{B}=\begin{pmatrix} a_{11}\pm b_{11} & a_{12}\pm b_{12} & \cdots & a_{1n}\pm b_{1n} \\ a_{21}\pm b_{21} & a_{22}\pm b_{22} & \cdots & a_{2n}\pm b_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1}\pm b_{m1} & a_{m2}\pm b_{m2} & \cdots & a_{mn}\pm b_{mn} \end{pmatrix}$。简言之，两个矩阵相加减，即它们相同位置的元素相加减。

注意：只有对于两个行数、列数分别相等的矩阵（同型矩阵），加减法运算才有意义，即加减法运算是可行的。

2．运算性质

满足交换律和结合律。

交换律：$\boldsymbol{A}+\boldsymbol{B}=\boldsymbol{B}+\boldsymbol{A}$。

结合律：$(\boldsymbol{A}+\boldsymbol{B})+\boldsymbol{C}=\boldsymbol{A}+(\boldsymbol{B}+\boldsymbol{C})$。

11.1.2 矩阵与数的乘法

1．运算规则

数 λ 乘以矩阵 $\boldsymbol{A}$，就是将数 λ 乘以矩阵 $\boldsymbol{A}$ 中的每一个元素，记为 $\lambda\boldsymbol{A}$ 或 $\boldsymbol{A}\lambda$。

特别地，称$-\boldsymbol{A}$ 为 $\boldsymbol{A}=(a_{ij})_{m\times s}$ 的负矩阵。

2．运算性质

满足结合律和分配律。

结合律：$(\lambda\mu)\boldsymbol{A}=\lambda(\mu\boldsymbol{A})$；$(\lambda+\mu)\boldsymbol{A}=\lambda\boldsymbol{A}+\mu\boldsymbol{A}$。

分配律：$\lambda(\boldsymbol{A}+\boldsymbol{B})=\lambda\boldsymbol{A}+\lambda\boldsymbol{B}$。

例：已知两个矩阵 $\boldsymbol{A}=\begin{bmatrix}3 & -1 & 2\\ 1 & 5 & 7\\ 2 & 4 & 5\end{bmatrix}$，$\boldsymbol{B}=\begin{bmatrix}7 & 5 & -2\\ 5 & 1 & 9\\ 4 & 2 & 1\end{bmatrix}$，满足矩阵方程 $\boldsymbol{A}+2\boldsymbol{X}=\boldsymbol{B}$，求未知矩阵 $\boldsymbol{X}$。

解：由已知条件可知 $\boldsymbol{X}=\frac{1}{2}(\boldsymbol{B}-\boldsymbol{A})=\frac{1}{2}\left(\begin{bmatrix}7 & 5 & -2\\ 5 & 1 & 9\\ 4 & 2 & 1\end{bmatrix}-\begin{bmatrix}3 & -1 & 2\\ 1 & 5 & 7\\ 2 & 4 & 5\end{bmatrix}\right)$

$$=\frac{1}{2}\begin{bmatrix}4 & 6 & -4\\ 4 & -4 & 2\\ 2 & -2 & -4\end{bmatrix}=\begin{bmatrix}2 & 3 & -2\\ 2 & -2 & 1\\ 1 & -1 & -2\end{bmatrix}$$

11.1.3 矩阵与矩阵的乘法

1．运算规则

设 $\boldsymbol{A}=(a_{ij})_{m\times s}$，$\boldsymbol{B}=(b_{ij})_{s\times n}$，则 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的乘积是这样一个矩阵：（1）行数与（左矩阵）$\boldsymbol{A}$ 相同，列数与（右矩阵）$\boldsymbol{B}$ 相同；（2）$\boldsymbol{C}$ 的第 i 行第 j 列的元素 c_{ij} 由 $\boldsymbol{A}$ 的第 i 行元素与 $\boldsymbol{B}$ 的第 j 列元素对应相乘，再取乘积之和。

定义：设 $\boldsymbol{A}$ 为 $m\times p$ 的矩阵，$\boldsymbol{B}$ 为 $p\times n$ 的矩阵，那么称 $m\times n$ 的矩阵 $\boldsymbol{C}$ 为矩阵 $\boldsymbol{A}$ 与 $\boldsymbol{B}$ 的乘积，记作 $\boldsymbol{C}=\boldsymbol{AB}$，其中矩阵 $\boldsymbol{C}$ 中的第 i 行和第 j 列元素可以表示为

$$(\boldsymbol{AB})_{ij}=\sum_{k=1}^{p}a_{ik}b_{kj}=a_{i1}b_{1j}+a_{i2}b_{2j}+\cdots+a_{ip}b_{pj}$$

如下所示：

$$\boldsymbol{C}=\boldsymbol{AB}=\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{pmatrix}\begin{pmatrix}1 & 4\\ 2 & 5\\ 3 & 6\end{pmatrix}=\begin{pmatrix}1\times1+2\times2+3\times3 & 1\times4+2\times5+3\times6\\ 4\times1+5\times2+6\times3 & 4\times4+5\times5+6\times6\end{pmatrix}=\begin{pmatrix}14 & 32\\ 32 & 77\end{pmatrix}$$

矩阵乘法其实并不难，它的意思就是将第一个矩阵 ***A*** 的第一行与第二个矩阵 ***B*** 的第一列的数字分别相乘，得到的结果相加，最终的值作为结果矩阵的第(1,1)位置的值（第一行第一列）。

同样，***A*** 矩阵的第一行与 ***B*** 矩阵的第二列的数字分别相乘，然后相加，最终的值作为结果矩阵第(1,2)位置的值（第一行第二列）。

再如，***A*** 矩阵的第二行与 ***B*** 矩阵的第一列的数字分别相乘，然后相加，最终的值作为结果矩阵的第(2,1)位置的值（第二行第一列）。

这里主要说明两个问题：

- ***A*** 矩阵的列数必须与 ***B*** 矩阵的行数相同，才能相乘。因为我们需要把 ***A*** 矩阵一行中的各个数字与 ***B*** 矩阵一列中的各个数字分别相乘，所以 ***A*** 矩阵的列数与 ***B*** 矩阵的行数必须相同。
- 矩阵 ***A*** 乘以矩阵 ***B*** 和矩阵 ***B*** 乘以矩阵 ***A*** 的结果必然是不一样的。

$$\boldsymbol{C}=\boldsymbol{AB}=\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}\begin{pmatrix}1&4\\2&5\\3&6\end{pmatrix}=\begin{pmatrix}1\times1+2\times2+3\times3&1\times4+2\times5+3\times6\\4\times1+5\times2+6\times3&4\times4+5\times5+6\times6\end{pmatrix}=\begin{pmatrix}14&32\\32&77\end{pmatrix}$$

$$\boldsymbol{D}=\boldsymbol{BA}=\begin{pmatrix}1&4\\2&5\\3&6\end{pmatrix}\begin{pmatrix}1&2&3\\4&5&6\end{pmatrix}=\begin{pmatrix}1\times1+4\times4&1\times2+4\times5&1\times3+4\times6\\2\times1+5\times4&2\times2+5\times5&2\times3+5\times6\\3\times1+6\times4&3\times2+6\times5&3\times3+6\times6\end{pmatrix}=\begin{pmatrix}17&22&27\\22&29&36\\27&36&45\end{pmatrix}$$

2．运算性质（假设运算都是可行的）

（1）结合律：(***AB***)***C***=***A***(***BC***)。

（2）分配律：***A***(***B***±***C***)=***AB***±***AC***（左分配律）；(***B***±***C***)***A***=***BA***±***CA***（右分配律）。

（3）(*λ****A***)***B***=*λ*(***AB***)=***A***(*λ****B***)。

11.2　ColorMatrix色彩变换

11.2.1　色彩变换矩阵

对于色彩的存储，Bitmap 类使用一个 32 位的数值来保存，红、绿、蓝及透明度各占 8 位，每个色彩分量的取值范围是 0～255。透明度为 0 表示完全透明，为 255 时色彩完全可见。

1．色彩信息的矩阵表示

由于一个色彩信息包含 R、G、B、Alpha 信息，所以，我们必然要使用一个四阶色彩变换矩阵来修改色彩的每一个分量值。

$$\begin{bmatrix} \text{Red} & 0 & 0 & 0 \\ 0 & \text{Green} & 0 & 0 \\ 0 & 0 & \text{Blue} & 0 \\ 0 & 0 & 0 & \text{Alpha} \end{bmatrix}$$

注意：对于色彩变换矩阵，这里的色彩顺序是 R、G、B、A，而不是 A、R、G、B。

如果想将色彩(0,255,0,255)更改为半透明，则可以使用下面的矩阵运算来表示：

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.5 \end{pmatrix} \times \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} = \begin{pmatrix} R \\ G \\ B \\ 0.5A \end{pmatrix}$$

上面使用四阶矩阵完全可以改变图片的 RGBA 值，但考虑一种情况：如果我们只想在原有的 R 色上增加一些分量呢？

这时，我们就得再多加一阶来表示平移变换。所以，一个既包含线性变换又包含平移变换的组合变换称为仿射变换。使用四阶色彩变换矩阵来修改色彩，只能对色彩的每个分量值进行乘（除）运算。如果要对这些分量值进行加减法运算（平移变换），则只能通过五阶矩阵来完成。

考虑下面这个变换：

（1）红色分量值更改为原来的 2 倍。

（2）绿色分量值增加 100。

这个变换使用四阶矩阵的乘法无法实现。所以，应该在四阶色彩变换矩阵上增加一个“哑元坐标”，来实现所列的矩阵运算。

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 100 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 25 \\ 100 \\ 100 \\ 255 \\ 1 \end{pmatrix} = \begin{pmatrix} 50 \\ 200 \\ 100 \\ 255 \end{pmatrix}$$

在这个矩阵中，分量值用的是 100。

2．Android 中的色彩变换矩阵

在 Anroid 中，色彩变换矩阵的表示形式也是五阶的。所以，在默认情况下，色彩变换矩阵的形式如下：

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Android 中的色彩变换矩阵是用 ColorMatrix 类来表示的。使用 ColorMatrix 类的方法如下：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
        1, 0, 0, 0, 0,
        0, 1, 0, 0, 0,
        0, 0, 1, 0, 0,
        0, 0, 0, 0.5, 0,
});
mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
```

有关 setColorFilter()函数的其他用法，将在本节末尾详细讲解。

3．示例：彩色图片的蓝色通道输出

下面以为 Bitmap 应用 ColorMatrix 类为例，代码如下：

```
public class MyView extends View {
    private Paint mPaint = new Paint();
    private Bitmap bitmap;// 位图

    public MyView(Context context, AttributeSet attrs) {
        super(context, attrs);

        mPaint.setAntiAlias(true);
        // 获取位图
        bitmap = BitmapFactory.decodeResource(context.getResources(),
R.drawable.dog);
    }

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

        // 绘制原始位图
        canvas.drawBitmap(bitmap, null, new Rect(0, 0, 500, 500 * bitmap.
getHeight() / bitmap.getWidth()), mPaint);

        canvas.translate(510, 0);
        // 生成色彩变换矩阵
        ColorMatrix colorMatrix = new ColorMatrix(new float[]{
                0, 0, 0, 0, 0,
                0, 0, 0, 0, 0,
                0, 0, 1, 0, 0,
                0, 0, 0, 1, 0,
        });
        mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
        canvas.drawBitmap(bitmap, null, new Rect(0, 0, 500, 500 * bitmap.
getHeight() / bitmap.getWidth()), mPaint);
    }
}
```

这里分两次绘制了一个 Bitmap，先绘制了一个原始图像，然后利用 ColorMatrix 类生成了一个仅包含蓝色的图像。用过 Photoshop 的读者应该很清楚，这跟 Photoshop 中蓝色通道的效果是一致的。效果如下图所示。

扫码看彩色图

注意：不要在 onDraw()函数里新建 Paint 对象，否则会造成内存回收，严重影响性能。

11.2.2 色彩的几种运算方式

在简单理解了 ColorMatrixs 类的使用方式后，下面来详细看看色彩的几种运算方式。

1．色彩的平移运算

1）增加色彩饱和度

色彩的平移运算实际上就是色彩的加法运算，其实就是在色彩变换矩阵的最后一列加上某个值，这样可以增加特定色彩的饱和度。

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{pmatrix}$$

比如，同样是上面的图片，我们给它应用下面的色彩值。

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       1, 0, 0, 0, 0,
       0, 1, 0, 0, 50,
       0, 0, 1, 0, 0,
       0, 0, 0, 1, 0,
});
```

在绿色值上添加增量 50，即增大绿色的饱和度。效果如下图所示。

扫码看彩色图

同样，左侧是原图，右侧是增大绿色饱和度后的效果。大家要特别注意的是，由于图片

是由一个个像素组成的，所以用每个像素所对应的色彩数组来乘以色彩变换矩阵，结果就是变换后的当前点的颜色值。在应用 ColorMatrix 类后，图片中每个像素的绿色值都增加了 50，从小狗的脸上也可以看出来。

2）色彩反转/反相功能

色彩平移除增加指定色彩的饱和度以外，另一个应用就是色彩反转，也就是 Photoshop 中的反相功能。色彩反转就是求出每个色彩的补值来作为目标图像的对应颜色值。示例代码如下：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       -1,0,0,0,255,
       0,-1,0,0,255,
       0,0,-1,0,255,
       0,0,0,1,0
});
```

效果如下图所示。

扫码看彩色图

2．色彩的缩放运算

1）调节亮度

色彩的缩放运算其实就是色彩的乘法运算。将色彩变换矩阵对角线上分别代表 R、G、B、A 的几个值分别乘以指定的值，就是所谓的缩放运算，如下图所示。

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{pmatrix}$$

我们可以针对某个颜色值进行放大/缩小运算。但是当对 R、G、B、A 同时进行放大/缩小运算时，就是对亮度进行调节。

看下面将亮度增大 1.2 倍的代码：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       1.2f, 0, 0, 0, 0,
       0, 1.2f, 0, 0, 50,
       0, 0, 1.2f, 0, 0,
       0, 0, 0, 1.2f, 0,
});
```

效果如下图所示。

扫码看彩色图

2）通道输出

由于在色彩变换矩阵中对角线上的数的取值范围为 0～1，所以，当取 0 时，这个色彩就完全不显示；当 R、G 都取 0，而独有 B 取 1 时，就只显示蓝色，所形成的图像也就是我们通常所说的蓝色通道。看一下几个通道输出的效果图，如下图所示。

扫码看彩色图

红色通道矩阵：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
      1, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 1, 0,
});
```

绿色通道矩阵：

```
ColorMatrix colorMatrix2 = new ColorMatrix(new float[]{
       0, 0, 0, 0, 0,
       0, 1, 0, 0, 0,
```

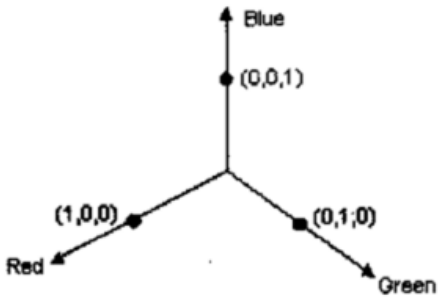
```
        0, 0, 0, 0, 0,
        0, 0, 0, 1, 0,
});
```

蓝色通道矩阵：

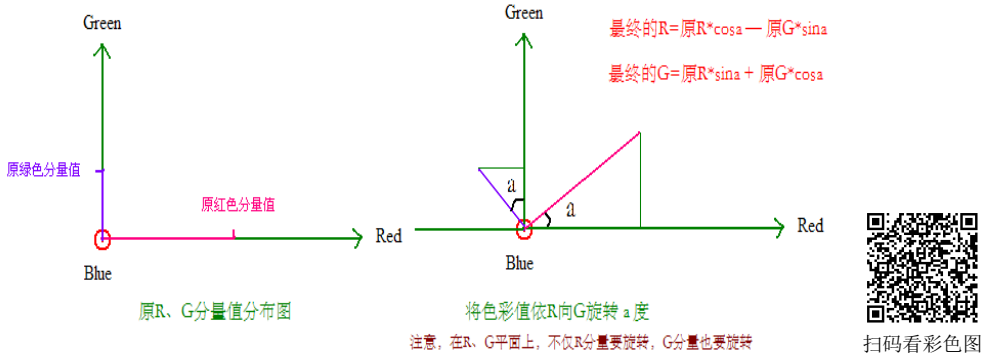
```
ColorMatrix colorMatrix3 = new ColorMatrix(new float[]{
        0, 0, 0, 0, 0,
        0, 0, 0, 0, 0,
        0, 0, 1, 0, 0,
        0, 0, 0, 1, 0,
});
```

3．色彩的旋转运算

RGB 色是如何旋转的呢？首先用 R、G、B 三色建立立体坐标系，如下图所示。



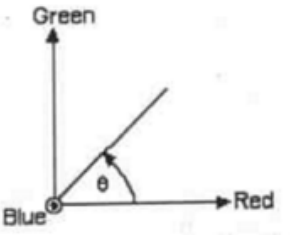
所以，我们可以把一个色彩值看成三维空间里的一个点，色彩值的三个分量可以看成该点的坐标（三维坐标）。我们先不考虑在三个维度综合情况下是怎么旋转的，来看看将某个轴作为 *Z* 轴，在另外两个轴形成的平面上旋转的情况。下图分析了将蓝色轴作为 *Z* 轴，仅在红—绿平面上旋转 *a* 度的情况。



扫码看彩色图

可以看到，在旋转后，原 R 在 R 轴上的分量变为原 Rxcos*a*，原 G 在 R 轴上也有了分量，但分量落在了负轴上，所以要减去这部分分量，最终的结果是最终的 R=原 R×cos*a*−原 G×sin*a*。下面来看一下几种旋转计算及结果矩阵（注意：这几张图只标记了原 *X* 轴色彩分量的旋转，没有把 *Y* 轴色彩分量的旋转标记出来）。

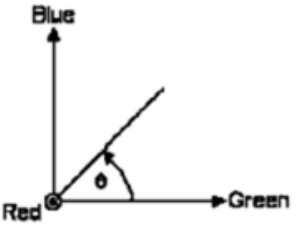
（1）绕蓝色轴旋转 θ 度。



对应的色彩变换矩阵如下：

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

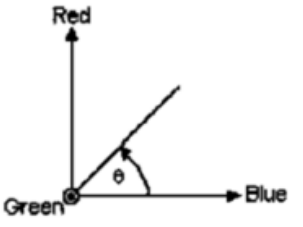
（2）绕红色轴旋转 θ 度。



对应的色彩变换矩阵如下：

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 & 0 \\ 0 & -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

（3）绕绿色轴旋转 θ 度。



对应的色彩变换矩阵如下：

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

当围绕红色轴进行色彩旋转时，由于当前红色轴的色彩是不变的，而仅利用三角函数来动态变更绿色和蓝色的颜色值，这种改变就叫作色相调节。当围绕红色轴旋转时，是对图片进行红色色相的调节；当围绕蓝色轴旋转时，是对图片进行蓝色色相的调节；当围绕绿色轴旋转时，是对图片进行绿色色相的调节。

下面我们会再次讲到 ColorMatrix 的色彩旋转函数，这里先理解原理，代码和效果后面会给出。

4．色彩的投射运算

我们再回过头来看看色彩变换矩阵运算的公式，如下：

$$\begin{Bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \end{Bmatrix} \times \begin{Bmatrix} R \\ G \\ B \\ A \\ 1 \end{Bmatrix} = \begin{Bmatrix} a_{11}R+a_{12}G+a_{13}B+a_{14}A+a_{15} \\ a_{21}R+a_{22}G+a_{23}B+a_{24}A+a_{25} \\ a_{31}R+a_{32}G+a_{33}B+a_{34}A+a_{35} \\ a_{41}R+a_{42}G+a_{43}B+a_{44}A+a_{45} \end{Bmatrix}$$

在上式中，把红色运算单独标记出来，在运算中，它们就是利用 G、B、A 的颜色值的分量来增加红色值的。

来看具体的运算：

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 100 \\ 200 \\ 125 \\ 200 \\ 1 \end{pmatrix} = \begin{pmatrix} 100 \\ 220 \\ 125 \\ 200 \end{pmatrix}$$

注意：最终结果的 220=0.2×100+1×200，可见绿色分量在原有绿色分量的基础上增加了红色分量值的 0.2 倍。利用其他色彩分量的倍数来更改自己色彩分量的值，这种运算就叫作投射运算。

在对下图中阴影部分的值进行修改时，所使用的增加值来自其他色彩分量的信息。

$$\begin{bmatrix} m[0][0] & m[0][1] & m[0][2] & m[0][3] & m[0][4] \\ m[1][0] & m[1][1] & m[1][2] & m[1][3] & m[1][4] \\ m[2][0] & m[2][1] & m[2][2] & m[2][3] & m[2][4] \\ m[3][0] & m[3][1] & m[3][2] & m[3][3] & m[3][4] \end{bmatrix}$$

应用一：黑白图片

色彩投射的一个最简单的应用就是将彩色图片变为黑白图片。代码如下：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       0.213f, 0.715f, 0.072f, 0, 0,
       0.213f, 0.715f, 0.072f, 0, 0,
       0.213f, 0.715f, 0.072f, 0, 0,
       0,      0,    0, 1, 0,
});
```

效果如下图所示。

扫码看彩色图

首先了解一下去色原理：只要把 R、G、B 三通道的色彩信息设置成一样，即 R＝G＝B，图像就变成了灰色。并且，为了保证图像亮度不变，同一个通道中的 R+G+B=1，如 0.213+0.715+0.072＝1。

下面谈一下 0.213、0.715、0.072 这三个数字的由来。

按理说应该把 R、G、B 平分，都是 0.3333333。三个数字应该是根据色彩光波频率及色彩心理学计算出来的。

在作用于人眼的光线中，彩色光要明显强于无色光。如果对一张图像按 RGB 平分理论给图像去色，人眼就会明显感觉到图像变暗了（当然可能有心理上的原因，也有光波的科学依据）。另外，在彩色图像中能够识别的一些细节也可能会丢失。

所以 Google 最终给出的颜色值就是上面的三个数字：0.213、0.715、0.072。我们在给图像去色时保留了大量的 G 通道信息，使得图像不至于变暗或者绿色信息不至于丢失。

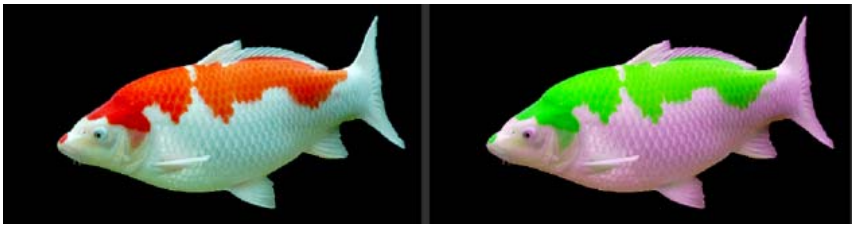
应用二：色彩反色

利用色彩变换矩阵将两个颜色反转，这种操作就叫作色彩反色。

比如，将红色和绿色反色（红绿反色），代码如下：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       0,1,0,0,0,
       1,0,0,0,0,
       0,0,1,0,0,
       0,0,0,1,0
});
```

效果如下图所示。



扫码看彩色图

左侧为原图，右侧为红绿反色以后的效果图。

从色彩变换矩阵中可以看出，红绿反色的关键在于，第一行用绿色来代替红色，第二行用红色来代替绿色。类似的可以有红蓝反色、绿蓝反色等，对应矩阵难度不大，就不再细讲了。

应用三：照片变旧

投射运算的另一个应用是照片变旧，其对应的矩阵如下：

```
ColorMatrix colorMatrix = new ColorMatrix(new float[]{
       1/2f,1/2f,1/2f,0,0,
       1/3f,1/3f,1/3f,0,0,
       1/4f,1/4f,1/4f,0,0,
       0,0,0,1,0
});
```

效果如下图所示。

扫码看彩色图

11.2.3　ColorMatrix函数

上一小节讲述了利用色彩变换矩阵来进行的一些运算，但这些都是需要特定的色彩设计基础的。在 Android 中，ColorMatrix 自带一些函数，用来帮助我们完成调整饱和度、色彩旋转等操作。

1．构造函数

ColorMatrix 共有三个构造函数，如下：

```
ColorMatrix()
ColorMatrix(float[] src)
ColorMatrix(ColorMatrix src)
```

在这三个构造函数中，我们已经使用过第二个构造函数；至于第三个构造函数，就是利用另一个 ColorMatrix 实例来复制一个一样的 ColorMatrix 对象。

2．设置和重置函数

第一个构造函数 ColorMatrix()需要与其他函数共用才行。

```
public void set(ColorMatrix src)
public void set(float[] src)
public void reset()
```

上面的函数是设置和重置函数，重置后，对应的数组如下：

```
/**
* Set this colormatrix to identity:
* [ 1 0 0 0 0   - red vector
*   0 1 0 0 0   - green vector
*   0 0 1 0 0   - blue vector
*   0 0 0 1 0 ] - alpha vector
*/
```

3．setSaturation()函数——设置饱和度

我们可以通过色彩的平移运算单独增强 R、G、B 其中一个分量的饱和度，但当我们需要整体增强色彩饱和度时，需要如何来做呢？ColorMatrix 提供了一个函数来整体增强色彩饱和度，如下：

```
//整体增强色彩饱和度，即同时增强 R、G、B 的色彩饱和度
public void setSaturation(float sat)
```

其中，参数 float sat 表示把当前色彩饱和度放大的倍数。当取值为 0 时，表示完全无色彩，即灰度图像（黑白图像）；当取值为 1 时，表示色彩不变动；当取值大于 1 时，显示色彩过度饱和。

举个例子：滑块默认在 1 倍的位置，向左到底是 0，向右到底是 20（饱和度放大 20 倍）。

扫码看动态效果图

布局非常简单，上面是一张图片，下面是一个 SeekBar，这里就不再列出具体的布局代码，核心处理代码如下：

```
public class MyActivity extends Activity {
   private SeekBar mSeekBar;
```

```
        private ImageView mImageView;
        private Bitmap mOriginBmp,mTempBmp;
        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);

            mImageView = (ImageView) findViewById(R.id.img);
            mSeekBar = (SeekBar)findViewById(R.id.seekbar);
            mOriginBmp = BitmapFactory.decodeResource(getResources(), R.drawable.
dog);
            mTempBmp = Bitmap.createBitmap(mOriginBmp.getWidth(), mOriginBmp.
getHeight(),
                    Bitmap.Config.ARGB_8888);


            mSeekBar.setMax(20);
            mSeekBar.setProgress(1);

            mSeekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChange
Listener() {
                @Override
                public void onProgressChanged(SeekBar seekBar, int progress,
boolean fromUser) {

                    Bitmap bitmap = handleColorMatrixBmp(progress);
                    mImageView.setImageBitmap(bitmap);
                }

                @Override
                public void onStartTrackingTouch(SeekBar seekBar) {

                }

                @Override
                public void onStopTrackingTouch(SeekBar seekBar) {

                }
            });

        }
     ...
    }
```

这里首先将原图加载到 mOriginBmp 中，并且创建一张和原图相同大小的空白图像 mTempBmp，在 SeekBar 滚动时，根据当前进度值生成当前调整饱和度后的新图像，之后重新设置给 ImageView。

其中，根据饱和度生成新图像的 handleColorMatrixBmp()函数的定义如下：

```
    private Bitmap handleColorMatrixBmp(int progress){
        // 创建一个相同尺寸的可变的位图区，用于绘制调色后的图片
        Canvas canvas = new Canvas(mTempBmp); // 得到画笔对象
        Paint paint = new Paint();
```

```
        ColorMatrix mSaturationMatrix = new ColorMatrix();
        mSaturationMatrix.setSaturation(progress);
         // 设置色彩变换效果
        paint.setColorFilter(new ColorMatrixColorFilter(mSaturationMatrix));
         // 将色彩变换后的图片输出到新创建的位图区
        canvas.drawBitmap(mOriginBmp, 0, 0, paint);
        // 返回新的位图，即调色处理后的图片
        return mTempBmp;
    }
```

mTempBmp 是新生成的一张跟原始 Bitmap 同样大小的空白图片，在设置 Paint 的 ColorMatrix 之后，利用 canvas.drawBitmap(mOriginBmp, 0, 0, paint);在原始图片的基础上应用 Paint 把生成的图像画在 Canvas 上。drawBitmap()函数的第一个参数表示的是源图像。

4．setScale()函数——色彩缩放

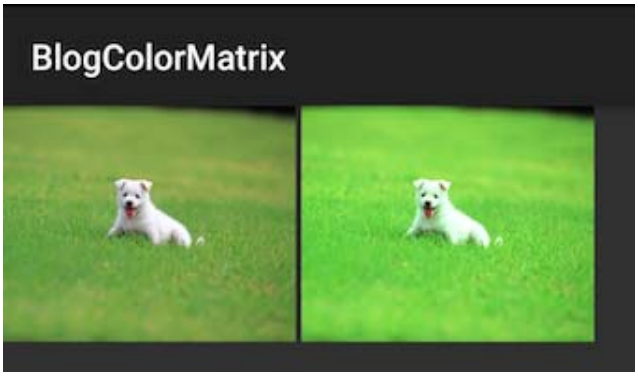
同样，对于色彩的缩放运算，ColorMatrix 也为我们封装了一个函数。

```
    public void setScale(float rScale, float gScale, float bScale,float aScale)
```

这个函数共有 4 个参数，分别对应 R、G、B、A 颜色值的缩放倍数。比如，在小狗图片中，绿色占大部分，所以我们仅将绿色放大 1.3 倍。

```
    canvas.drawBitmap(bitmap, null, new Rect(0, 0, 500, 500 * bitmap.getHeight() /
bitmap.getWidth()), mPaint);

    canvas.save();
    canvas.translate(510, 0);
    // 生成色彩变换矩阵
    ColorMatrix colorMatrix = new ColorMatrix();
    colorMatrix.setScale(1,1.3f,1,1);
    mPaint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));

    canvas.drawBitmap(bitmap, null, new Rect(0, 0, 500, 500 * bitmap.getHeight() /
bitmap.getWidth()), mPaint);
```

效果如下图所示。



扫码看彩色图

在仅将绿色放大 1.3 倍后，整张图片看起来更鲜艳了。

5．setRotate()函数——色彩旋转

上面在讲解色彩旋转运算时，列出了在色彩旋转时的效果和原理。由于涉及正、余弦函

数的计算，而且这些公式推导起来具有一定的难度，所以 Android 已经封装好了色彩旋转的函数。

```
/**
 * 将旋转围绕某一个颜色轴进行
 * axis=0 围绕红色轴旋转
 * axis=1 围绕绿色轴旋转
 * axis=2 围绕蓝色轴旋转
 */
public void setRotate(int axis, float degrees);
```

这里有两个参数。

- int axis：表示围绕哪个轴旋转，取值为 0、1、2。当取值为 0 时，表示围绕红色轴旋转；当取值为 1 时，表示围绕绿色轴旋转；当取值为 2 时，表示围绕蓝色轴旋转。
- float degrees：表示旋转的度数。

同样利用上面色彩旋转的图像和滑动条的框架，来看一下当围绕某一个颜色轴旋转时色相变化的效果。

扫码看动态效果图

这里的代码与调节饱和度的代码只有两点不同。

（1）设置 SeekBar 的范围。

```
mSeekBar.setMax(360);
mSeekBar.setProgress(180);
```

（2）在 handleColorRotateBmp()函数中处理当前 progress。

```
ColorMatrix colorMatrix = new ColorMatrix();
colorMatrix.setRotate(0,progress-180);
paint.setColorFilter(new ColorMatrixColorFilter(colorMatrix));
```

将当前 progress 位置减去 180，即中间位置的数字。所以，中间位置的色彩旋转度数为 0，整个旋转度数的范围是-180°～180°；360°正好是正/余弦函数的一个最小正周期。

上面的效果针对的是红色色相，同理，可以得到围绕绿色轴旋转的效果图。

扫码看动态效果图

也可以得到围绕蓝色轴旋转的效果图。

扫码看动态效果图

11.2.4 ColorMatrix相乘

矩阵相乘涉及三个函数。

```
public void setConcat(ColorMatrix matA, ColorMatrix matB)
```

这个函数接收两个 ColorMatrix 矩阵 matA 和 matB，乘法规则为 matA×matB，然后将结果作为当前 ColorMatrix 的值。

```
public void preConcat(ColorMatrix prematrix)
```

假设当前矩阵为 ***A***，那么 preConcat()函数的含义就是将当前的矩阵 ***A*** 乘以 prematrix。

```
public void postConcat(ColorMatrix postmatrix)
```

postConcat()函数的含义就是 postmatrix 乘以当前矩阵 ***A***。

由于这部分内容基本用不到，有关这几个函数的具体应用这里就不再详细讲述，有兴趣的读者可以参考http://blog.csdn.net/harvic880925/article/details/51187277。