**聊聊Bitmap的那些事儿**

**1 一个Bitmap对象占多大内存空间**

1.1 解码格式

　　一般来讲，Bitmap的占用的内存空间，有下面的计算方法：

bitmap的内存空间 = 图片宽度 \* 图片高度 \* 单位像素所占字节数（Byte）

　　图片宽度和高度比较好理解，单位像素所占字节数是个什么玩意，难道每个像素的字节数会不一样吗？

　　确实会不一样，单位像素字节数与解析Bitmap时解码的格式有关，Android 系统支持四种像素解码格式，如下图所示：

| 像素格式 | 单位像素字节数 | 备注 |
| --- | --- | --- |
| ARGB\_8888 | 4 | ARGB四个通道，每个通道占8位，即每个像素点占32位，4个字节 |
| RGB\_565 | 2 | RGB三个通道，分别占5、6、5位，即每个像素点占16位，2个字节 |
| ARGB\_4444 | 2 | ARGB四个通道，每个通道占4位，即每个像素点占16位，2个字节 |
| ALPHA\_8 | 1 | 只有透明度通道，占8位，1个字节 |

　　ALPHA\_8只有透明度通道，只能用于一些特殊场景。

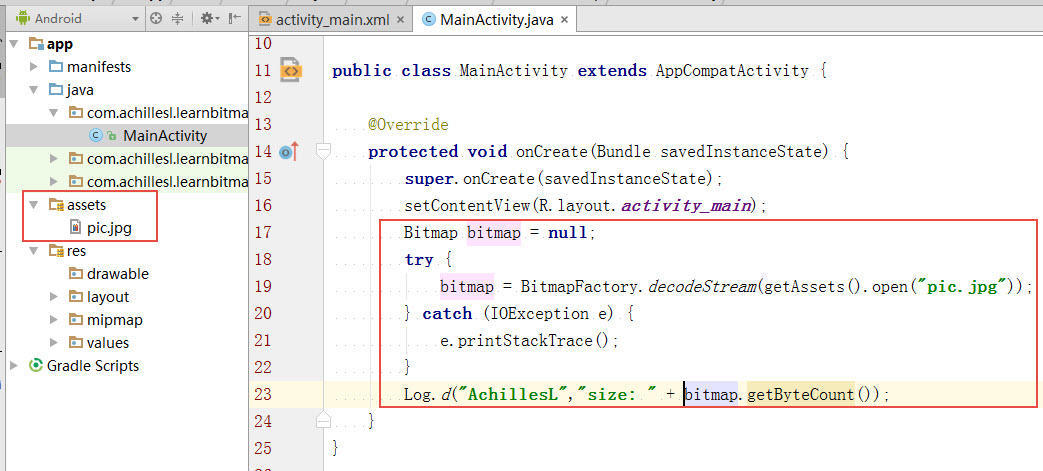
　　ARGB\_4444由于质量太差，已经不建议使用，而且在KITKAT（Android4.4）及其以上版本，使用ARGB\_4444参数会直接在代码中被忽略，用ARGB\_8888替代。

　　我们常用的只有RGB\_565和ARGB\_8888。

1.2 ARBG\_8888 PK RGB\_565

　　使用RGB\_565格式解码后，图片每像素点占2字节，而ARGB\_8888解码则每像素点占4字节。按道理，使用RGB\_565格式解码应该比RGB\_8888解码省一半内存。接下来，我们做个实验来验证一下：

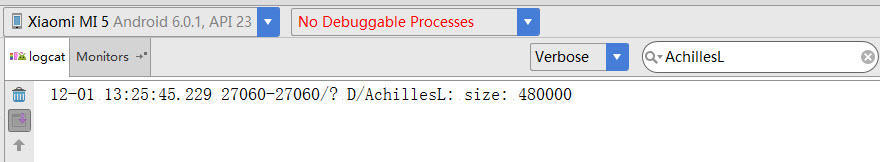
　　笔者手上有一张图，长400像素，宽300像素，jpg格式。我们把它放到 assets 目录中，然后编写代码，查看一下它的内存占用情况。



　　图片的占用内存，我们可以用Bitmap的getByteCount方法得到，在运行程序前，我们先按照公式算出该图内存占用是多少。由于Android默认采用ARGB\_8888格式解码，即一个像素点占4个字节，因此：

　　该图占用内存 = 400 \* 300 \* 4 = 480000 Byte

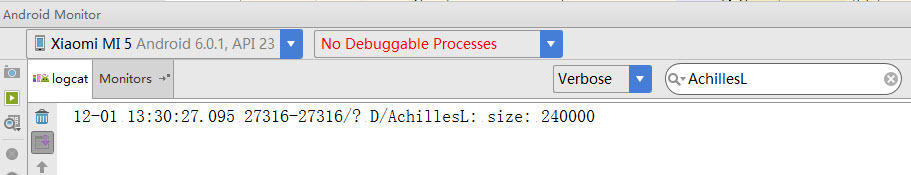
　　运行程序，发现结果和我们理论值是一模一样的：



　　第二步，我们验证一下采用RGB\_565格式进行解码，看图片内存占用如何。我们可以通过BitmapFactory.Options类的inPreferredConfig属性，指定图片解码格式，代码如下：



　　运行程序，发现果然比用ARGB\_8888解码少占用一半的内存。



　　但需要注意的是，使用RGB\_565解码，不一定总比ARGB\_8888解码占用内存更小，如果你对png格式的图片采用RGB\_565格式解码，得到的结果很可能是和使用ARGB\_8888的结果是一致。这是由于：

如果inPreferredConfig为null或者在解码时无法满足此参数指定的颜色模式，解码器会自动根据原始图片的特征以及当前设备的屏幕位深，选取合适的颜色模式来解码，例如，如果图片中包含透明度，那么对该图片解码时使用的配置就需要支持透明度，默认会使用ARGB\_8888来解码。

也就是说inPreferredConfig指定的配置并非是一个强制选项，而是建议的（preferred）选项，Android在实际解码时会参考此参数的配置，但如果此配置不满足，Android会重新选取一个合适的配置来对图片进行解码。

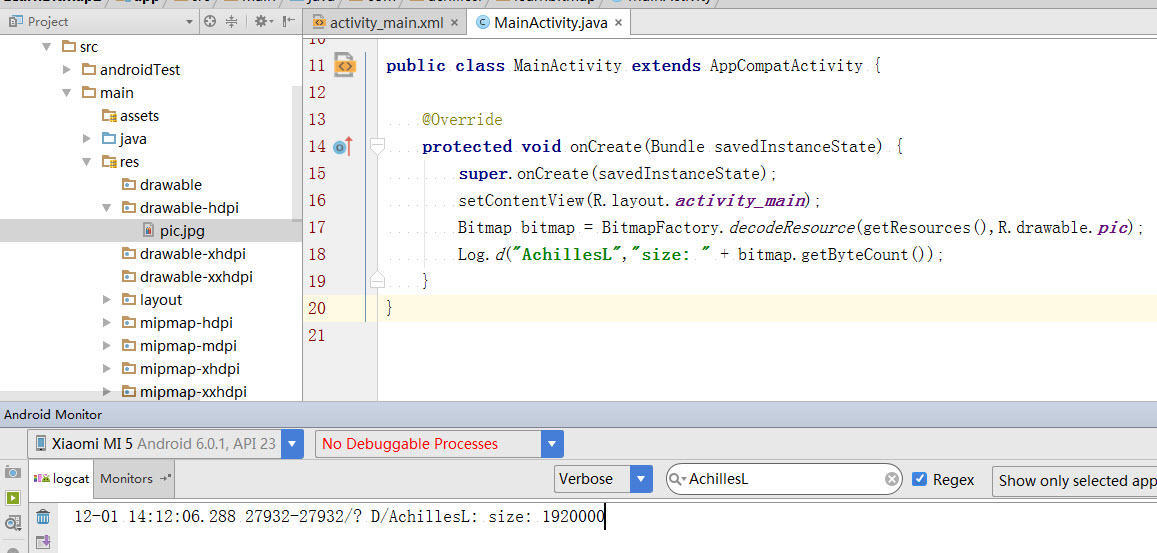
　　最后，我们有以下总结：

对于jpg格式图片，使用RGB\_565格式进行解码，图片占用的内存会更少

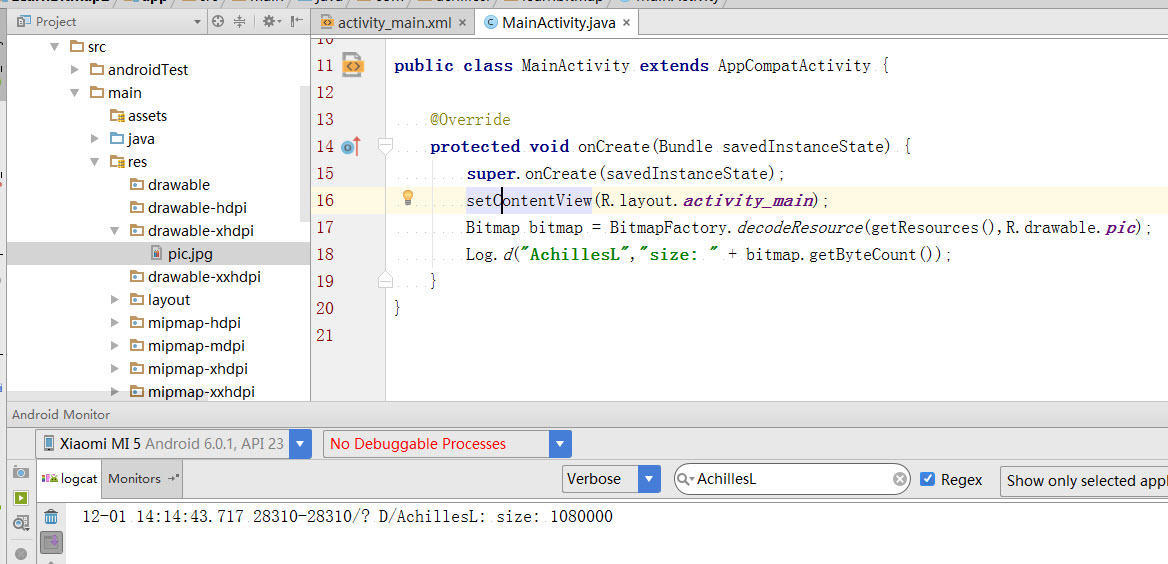
1.3 不同资源目录下，图片内存占用情况

　　细心的读者可能已经留意到，上述的实验过程，图片是存放在Android的assets 目录下，这是笔者为了避免图片存放在不同drawable目录下产生占用内存不同的影响。接下来，我们将图片放在不同的drawable目录并分别查看内存的占用情况。

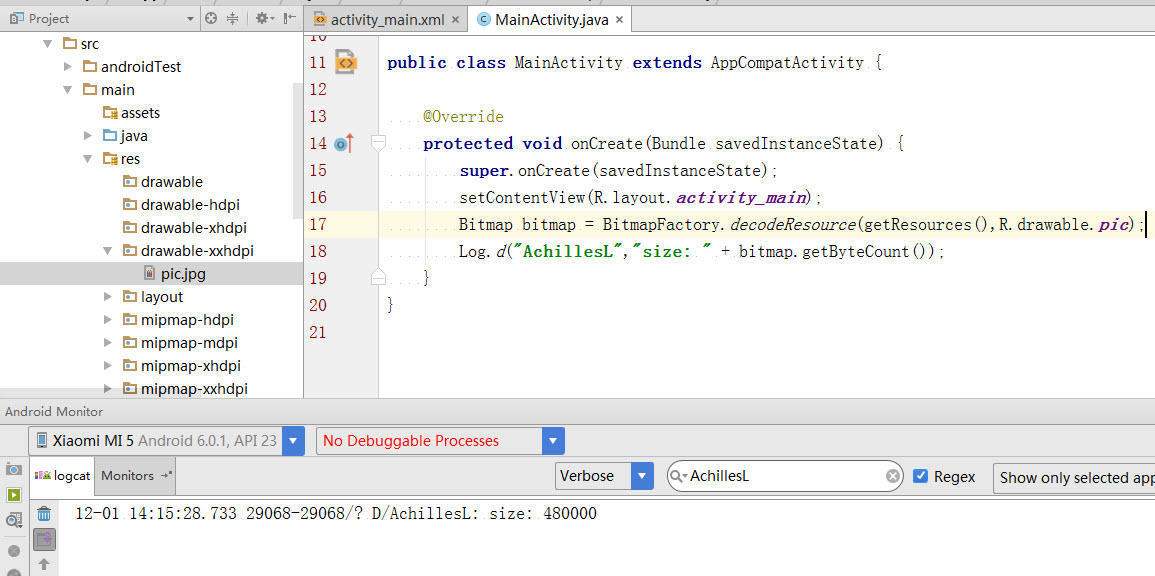
　　图片放在drawable-hdpi目录时，内存占用1920000字节：



　　图片放在drawable-xhdpi目录时，内存占用1080000字节：



　　图片放在drawable-xxhdpi目录时，内存占用480000字节：



　　我们从实验结果发现，除了放在 drawable-xxhdpi目录下，否则图片内存占用都要比上述计算的理论值要大很多，这是为什么呢？

　　首先我们知道，Android中drawable系列目录的存在，如：drawable-ldpi、drawable-mdpi、drawable-hdpi、drawable-xhdpi、drawable-xxhdpi等，是为了兼容适配不同手机屏幕的屏幕密度。

　　屏幕密度与Drawable系列目录的对应关系如下:

| 目录 | 屏幕密度 |
| --- | --- |
| drawable-ldpi | 120dpi |
| drawable-mdpi | 160dpi |
| drawable-hdpi | 240dpi |
| drawable-xhdpi | 320dpi |
| drawable-xxhdpi | 480dpi |

　　当我们使用decodeResource方法读取drawable目录下面的图片时，会根据手机的屏幕密度，到对应的文件夹中查找该图片。若该图片存在于其他目录下，系统先对该图片进行缩放处理，再显示。

　　也就是说，资源图片需要根据手机屏幕密度的值来存放到不同的drawable目录中，图片的显示才是正常的。

　　对于我们的例子，由于笔者实验的机器是小米5，该手机的屏幕密度是480dpi，因此图片需要放在drawable-xxhpid目录下。如果图片放在其他目录如hdpi时，系统会认为该图是使用在hdpi密度的屏幕上，而实际的屏幕密度却比hdpi更高，此时系统将会对图片进行放大处理，这样避免了不同屏幕密度影响图片显示效果。由于存在系统对图片进行缩放操作，也就解释了为什么实际的内存占用比理论值要大。

　　当图片存放在drawable目录时，占用的内存空间有以下计算公式：

scale = 设备的屏幕密度 / drawable目录设定的屏幕密度

图片占用内存 = int（图片长度 \* scale + 0.5） \* int（图片宽度 \* scale + 0.5） \* 单位像素字节数

　　因此，我们的例子中，图片放入hdpi文件夹时所占内存为：

　　scale = 480 / 240 = 2;  
　　内存 = int(400 \* 2 + 0.5) \* int(300 \* 2 + 0.5) \* 4 = 1920000

　　图片放入xhdpi文件夹时所占内存：  
　　scale = 480 / 320 = 1.5;  
　　内存 = int(400 \* 1.5 + 0.5) \* int(300 \* 1.5 + 0.5) \* 4 = 1080000

　　这样，十分完美地解释了上述的实验结果。另外，关于图片在内存中占用空间，笔者也推荐大家阅读这篇由[腾讯Bugly](http://bugly.qq.com/bbs/forum.php?mod=viewthread&tid=498" \t "_blank)出品的文章：[Android坑档案：你的Bitmap究竟占多大内存？](https://zhuanlan.zhihu.com/p/20732309?refer=bennyhuo" \t "_blank)

**2 解决大图加载OOM的几种方式**

　　大图加载避免OOM，算是老话题了，网上的资料很多，在此笔者只介绍几种方法。OOM名为内存溢出，在避免内存溢出前，我们先搞清楚一个问题：为什么会发生内存溢出？

2.1 largeHeap模式

　　这要从Android的内存管理说起，Android系统的手机在系统底层指定了堆内存的上限值，我们的程序在申请内存空间时，为了确保能够成功申请到内存空间，应该保证当前已分配的内存加上当前需要分配的内存值的总大小不能超过当前堆的最大内存值（手机厂商会根据手机的配置情况来对其进行调整）。

　　如果解析后图片的大小加上目前已分配的堆内存大小超过堆内存最大值，系统先会执行一遍gc操作，若gc后仍然超过堆内存最大值，这时候将会抛出异常，这是就是发生了我们常说的OOM。

　　查看手机的堆内存大小，我们有两种方式：

　　通过代码查看：

ActivityManager activityManager = (ActivityManager) getSystemService(ACTIVITY\_SERVICE);

Log.d("AchillesL","size: " + activityManager.getMemoryClass());

　　OR，通过adb指令查看：

adb shell getprop|grep heapgrowthlimit

　　笔者的测试机是小米5，查到堆内存最大值为256M。

　　可不可以提高堆内存最大值来避免OOM呢？答案是可以的，我们只需要在AndroidManifest.xml中的Application节点中声明android:largeHeap="true"标识，即可以分配到更大的堆内存空间。

　　查看能分配到最大的堆内存空间，我们有两种方式：

　　通过代码查看：

ActivityManager activityManager = (ActivityManager) getSystemService(ACTIVITY\_SERVICE);

Log.d("AchillesL","size: " + activityManager.getLargeMemoryClass());

　　OR，通过adb指令查看：

adb shell getprop|grep dalvik.vm.heapsize

　　需要注意的是，采用设置largeHeap的方式，提高了分配到堆内存的上限，但这仅仅延迟了OOM的方式时期，治标不治本。关键还需要我们从本身的代码入手，从根本上解决问题。

　　如果我们在机顶盒上开发播放器应用，显示的图片都较大，而且非常消耗内存，一般在应用开发完成并且测试通过发布时，最后再加上android:largeHeap="true"标记。

2.2 采用合适的图片解码格式

　　很多情况下，网络下载的图片都使用jpg格式。对于jpg图片，使用RGB\_565格式进行解码比ARGB\_8888解码会节省更多内存。

　　设置图片的解码格式，可以通过BitmapFactory.Options类的inPreferredConfig属性来指定。代码如下所示：

BitmapFactory.Options options = new BitmapFactory.Options();

options.inPreferredConfig = Bitmap.Config.RGB\_565;

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

　　总结：

如果不需要alpha通道，特别是资源本身为jpg格式的情况下，用RGB\_565格式解码更加节省内存。

2.3 设置采样率

　　设置采样率，这种方式我们再熟悉不过了。解析图片时，通过设置采样率减少加载后图片的尺寸，达到减少图片内存占用的目的。该方法一般分为三步：

设置BitmapFactory.Options类的inJustDecodeBounds属性为true，表示加载图片时不分配图片的内存空间，仅仅计算出原始图片的宽度和高度，即options.width和options.height。

得到原始图片的宽度、高度，以及图片需要展示的宽度、高度，计算出合适的采样率inSampleSize。

设置BitmapFactory.Options类的inJustDecodeBounds属性为false，并设置图片采样率inSampleSize，然后加载图片。

其中，inSampleSize的值必须大于1。若inSampleSize的值为2，表示目标图片的宽、高都为原图的 1 / 2，此时内存占用只有原图的 1 / 4。

　　代码如下：

BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true;

BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

//目标图片宽度为200像素

int scale = options.outWidth / 200;

scale = scale > 1 ? scale: 1;

options.inJustDecodeBounds = false;

options.inSampleSize = scale;

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

Log.d("AchillesL","width: " + bitmap.getWidth() + " height: " + bitmap.getHeight());

　　另外，最新的官方文档中指出，采样率inSampleSize 的值，应当总是2的指数，比如1，2，4，8，16等，若传递给系统的值不是2的指数，那么系统会向下取整并选择一个最接近2的指数来代替。因此，在大部分情况下设置图片的采样率，只能达到节省内存的目的，并不能精确地控制图片的尺寸。

**3 图片的缩放、旋转、倾斜等处理**

　　本节主要总结Matrix对图片进行处理的方法。Matrix操作，总共分为translate（平移），scale（缩放），rotate（旋转）和skew（倾斜）四种。

3.1 缩放

　　上一节我们提到，设置图片的采样率，可以节省内存，但不能精确控制图片尺寸。如果我们需要图片在节省内存的同时，再使用具体的数值来展示，则需要我们另外处理。

　　使用Matrix类的postScale方法可以很轻松地完成图片的缩放，代码如下：

//获取原始图片

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic);

Log.d("AchillesL","origin width: " + bitmap.getWidth() + " origin height: " + bitmap.getHeight());

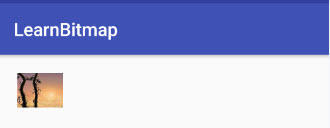
Matrix matrix = new Matrix();//将图片宽度缩放到150像素float scale = (float) (150.0 / bitmap.getWidth());//宽高等比例缩放

matrix.postScale(scale,scale);

Bitmap resultBitmap = Bitmap.createBitmap(bitmap,0,0,bitmap.getWidth(),bitmap.getHeight(),matrix,true);

Log.d("AchillesL","width: " + resultBitmap.getWidth() + " height: " + resultBitmap.getHeight());

　　效果图：



　　一般来讲，我们需要将一张大图以具体的小尺寸显示时，一般先设置采样率，进行“粗略”缩小达到目标尺寸，再利用Matrix 进行精确的尺寸控制。

　　另外，除了Matrix，也可以使用Bitmap类的createScaledBitmap方法进行缩放处理。

3.2 旋转

　　借助Matrix的postRotate方法可以使图片旋转一定角度。

　　代码如下：

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic);

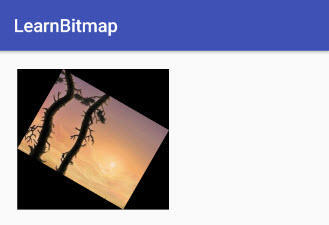
Matrix matrix = new Matrix();

matrix.postRotate(30);

Bitmap resultBitmap = Bitmap.createBitmap(bitmap,0,0,bitmap.getWidth(),bitmap.getHeight(),matrix,true);

imageView.setImageBitmap(resultBitmap);

　　效果图：



3.3 倾斜

　　借助Matrix的setSkew方法可以使图片产生倾斜效果。

　　代码如下：

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic);

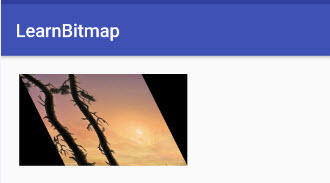
Matrix matrix = new Matrix();

matrix.setSkew((float) 0.5,0);

Bitmap resultBitmap = Bitmap.createBitmap(bitmap,0,0,bitmap.getWidth(),bitmap.getHeight(),matrix,true);

imageView.setImageBitmap(resultBitmap);

　　效果图：



3.4 preXXX、postXXX、setXXX等几个方法的区别

　　我们注意到，Matrix的四种基本操作：translate（平移），scale（缩放），rotate（旋转）和skew（倾斜）都提供了pre、post、set等三个方法。那它们有什么区别呢？

　　set是直接设置Matrix的值，每一次set，整个Matrix的数组都会重置。例如：

Matrix matrix = new Matrix();

matrix.setSkew((float) 0.5,0);

matrix.setRotate(30);

　　我们只能看到旋转的效果，看不到倾斜效果。

　　post是后乘，当前的矩阵乘以参数给出的矩阵。可以连续多次使用post，来完成所需的整个变换。例如：

Matrix matrix = new Matrix();

matrix.postRotate(30);

matrix.postTranslate(100,100);

　　表示把图片先旋转30度，再平移到坐标(100,100)的地方。

　　pre是前乘，参数给出的矩阵乘以当前的矩阵。所以操作是在当前矩阵的最前面发生的。例如：

Matrix matrix = new Matrix();

matrix.setTranslate(100,100);

matrix.preRotate(30)

　　表示把图片先旋转30度，再平移到坐标(100,100)的地方。

4 图片的切割处理

　　使用Bitmap时，对图片进行切割是经常遇到的事情，本节将总结将Bitmap切割为矩形、圆角、圆形等三种情况。

4.1 矩形

　　Bitmap的矩形切割非常简单，直接使用Bitmap类的createBitmap方法即可，代码如下：

//获取原始图片

BitmapFactory.Options options = new BitmapFactory.Options();

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

/\*从原图以0,0位坐标原点，切割一个宽100，高100的图片\*/

Bitmap resultBitmap = Bitmap.createBitmap(bitmap,0,0,100,100);

Log.d("AchillesL","width: " + resultBitmap.getWidth() + " height: " + resultBitmap.getHeight());

　　通过这个API，实现一个拼图游戏也是非常简单的事情，如下图所示。（如何实现拼图游戏，笔者将会在后续的文章中讲述）



4.2 圆角

　　实现图片的圆角并没有现成的API，需要借助Xfermode和PorterDuffXfermode来处理，我们通过把圆角矩形套在原Bitmap上取交集得到圆角Bitmap。

　　代码如下所示：

//获取原始图片

BitmapFactory.Options options = new BitmapFactory.Options();

Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

// 准备画笔

Paint paint = new Paint();

paint.setAntiAlias(true);

// 准备裁剪的矩阵

Rect rect = new Rect(0, 0, bitmap.getWidth(), bitmap.getHeight());

RectF rectF = new RectF(rect);

Bitmap roundBitmap = Bitmap.createBitmap(bitmap.getWidth(), bitmap.getHeight(), Bitmap

.Config.ARGB\_8888);

Canvas canvas = new Canvas(roundBitmap);

// 圆角矩阵，圆角半径为20像素

canvas.drawRoundRect(rectF, 20 , 20, paint);

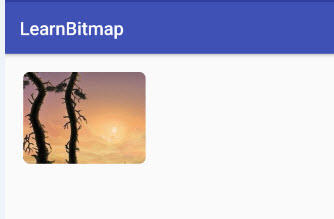
// 关键代码，关于Xfermode和SRC\_IN请自行查阅

paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC\_IN));

canvas.drawBitmap(bitmap, rect, rect, paint);

imageView.setImageBitmap(roundBitmap);

　　效果图：



4.3 圆形

　　圆形切割和上面的圆角切割原理相同，区别是将圆形套在图片上并取交集。

　　代码如下所示：

//获取原始图片BitmapFactory.Options options = new BitmapFactory.Options();Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic,options);

int min = bitmap.getWidth() > bitmap.getHeight() ?

bitmap.getHeight() : bitmap.getWidth();Paint paint = new Paint();

paint.setAntiAlias(true);Bitmap circleBitmap = Bitmap.createBitmap(min, min,

Bitmap.Config.ARGB\_8888);Canvas canvas = new Canvas(circleBitmap);// 圆形

canvas.drawCircle(min / 2, min / 2, min / 2, paint);

paint.setXfermode(new PorterDuffXfermode(PorterDuff.Mode.SRC\_IN));// 居中显示

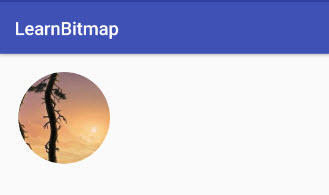
int left = - (bitmap.getWidth() - min) / 2;

int top = - (bitmap.getHeight() - min) / 2;

canvas.drawBitmap(bitmap, left, top, paint);

imageView.setImageBitmap(circleBitmap);

　　效果图：



　　采用Canvas与PorterDuffXfermode的方式，我们可以绘制任何形状的图片。

5 颜色矩阵ColorMatrix

　　在Android中，对图像进行颜色方面的处理，如黑白老照片、泛黄旧照片、高对比度、低饱和度等效果，都可以通过使用颜色矩阵ColorMatrix来实现。

　　灰阶效果：

Bitmap originBitmap = BitmapFactory.decodeResource(getResources(),R.drawable.pic);

Bitmap grayBitmap = Bitmap.createBitmap(originBitmap.getWidth(),

originBitmap.getHeight(), Bitmap.Config.RGB\_565);

Canvas canvas = new Canvas(grayBitmap);

Paint paint = new Paint();

ColorMatrix colorMatrix = new ColorMatrix();// 设置饱和度为0，实现了灰阶效果

colorMatrix.setSaturation(0);

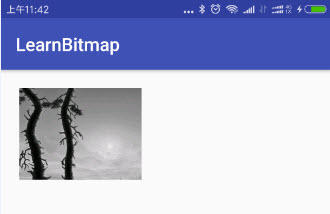
ColorMatrixColorFilter colorMatrixColorFilter = new ColorMatrixColorFilter(colorMatrix);

paint.setColorFilter(colorMatrixColorFilter);

canvas.drawBitmap(originBitmap, 0, 0, paint);

imageView.setImageBitmap(grayBitmap);

　　效果图：



6 使用BitmapRegionDecoder加载高清巨图

　　前面我们讲到使用采样率来避免大图加载OOM，但开发中还可能会遇到一种场景，就是加载长图或世界地图。这种图片体积大，而且不能通过设置采样率来加载显示，否则将会丢失图片细节。

　　一般加载高清巨图，我们使用BitmapRegionDecoder来处理。通过该类，我们可以加载局部的图片。

　　BitmapRegionDecoder的用法很简单：

public static BitmapRegionDecoder newInstance(byte[] data, int offset,

int length, boolean isShareable) throws IOException {

}public static BitmapRegionDecoder newInstance(

FileDescriptor fd, boolean isShareable) throws IOException {

}public static BitmapRegionDecoder newInstance(InputStream is,

boolean isShareable) throws IOException {

}public static BitmapRegionDecoder newInstance(String pathName,

boolean isShareable) throws IOException {

}

　　我们以加载世界地图为例，通过BitmapRegionDecoder类加载世界地图的左上角部分并显示，代码如下：

BitmapRegionDecoder bitmapRegionDecoder = null;try {

bitmapRegionDecoder = BitmapRegionDecoder.newInstance(getAssets().open("world.jpg"),

true);

} catch (IOException e) {

e.printStackTrace();

}

int screenWidth = getResources().getDisplayMetrics().widthPixels;int screenHeight = getResources().getDisplayMetrics().heightPixels;/\*以手机屏幕宽高生成一个矩形区域\*/

Rect rect = new Rect(0,0,screenWidth,screenHeight);

BitmapFactory.Options options = new BitmapFactory.Options();/\*设置RGB\_565格式\*/

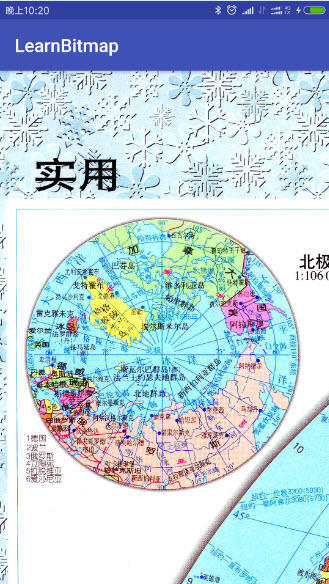
options.inPreferredConfig = Bitmap.Config.RGB\_565;

/\*加载部分图片\*/

Bitmap bitmap = bitmapRegionDecoder.decodeRegion(rect,options);

imageView.setImageBitmap(bitmap);

　　效果图：



　　另外，我们可以配合滑动手势，动态加载不同区域的图片，实现一个可以浏览大图的APP，笔者在后续的文章将会讲到。

7 结束语

　　本文总结了关于Bitmap的知识点，如果发现错漏之处，请留言回复提出。