Android 开发绕不过的坑：你的 Bitmap 究竟占多大内存？

原创2016-01-14霍丙乾腾讯 Bugly腾讯 Bugly

0、写在前面

本文涉及到屏幕密度的讨论，这里先要搞清楚 DisplayMetrics 的两个变量，摘录官方文档的解释：

- **density**：The logical density of the display. This is a scaling factor for the Density Independent Pixel unit, where one DIP is one pixel on an approximately 160 dpi screen (for example a 240x320, 1.5”x2” screen), providing the baseline of the system’s display. Thus on a 160dpi screen this density value will be 1; on a 120 dpi screen it would be .75; etc.

This value does not exactly follow the real screen size (as given by xdpi and ydpi, but rather is used to scale the size of the overall UI in steps based on gross changes in the display dpi. For example, a 240x320 screen will have a density of 1 even if its width is 1.8”, 1.3”, etc. However, if the screen resolution is increased to 320x480 but the screen size remained 1.5”x2” then the density would be increased (probably to 1.5).

- **densityDpi**：The screen density expressed as dots-per-inch.

简单来说，可以理解为 density 的数值是 1dp=density px；densityDpi 是屏幕每英寸对应多少个点（不是像素点），在 DisplayMetrics 当中，这两个的关系是线性的：

density	**1**	**1.5**	**2**	**3**	**3.5**	**4**
densityDpi	160	240	320	480	560	640

为了不引起混淆，本文所有提到的密度除非特别说明，都指的是 densityDpi，当然如果你愿意，也可以用 density 来说明问题。

另外，本文的依据主要来自 android 5.0 的源码，其他版本可能略有出入。文章难免疏漏，欢迎指正～

1、占了多大内存？

做移动客户端开发的朋友们肯定都因为图头疼过，说起来曾经还有过 leader 因为组里面一哥们在工程里面加了一张 jpg 的图发脾气的事儿，哈哈。

为什么头疼呢？吃内存呗，时不时还给你来个 OOM 冲冲喜，让你的每一天过得有滋有味（真是没救了）。那每次工程里面增加一张图片的时候，我们都需要关心这货究竟要占多大的坑，占多大呢？Android API 有个方便的方法，

```
public final int getByteCount() {
    // int result permits bitmaps up to 46,340 x 46,340
    return getRowBytes() * getHeight();
}
```

通过这个方法，我们就可以获取到一张 Bitmap 在运行时到底占用多大内存了。

举个例子

一张 **522x686** 的 **PNG** 图片，我把它放到 **drawable-xxhdpi** 目录下，在**三星 s6** 上加载，占用内存 2547360B，就可以用这个方法获取到。

2、给我一张图我告诉你占多大内存

每次都问 Bitmap 你到底多大啦。。感觉怪怪的，毕竟我们不能总是去问，而不去搞清楚它为嘛介么大吧。能不能给它算个命，算算它究竟多大呢？当然是可以的，很简单嘛，我们直接顺藤摸瓜，找出真凶，哦不，找出答案。

2.1 getByteCount

getByteCount 的源码我们刚刚已经认识了，当我们问 Bitmap 大小的时候，这孩子也是先拿到出生年月日，然后算出来的，那么问题来了，getHeight 就是图片的高度（单位：px），getRowBytes 是什么？

```
public final int getrowBytes() {
    if (mRecycled) {
            Log.w(TAG, "Called getRowBytes() on a recycle()'d bitmap! This is undefined behavior!");
    }
    return nativeRowBytes(mFinalizer.mNativeBitmap);
}
```

额，感觉太对了啊，要 JNI 了。由于在下 C++ 实在用得少，每次想起 JNI 都请想象脑门磕墙的场景，不过呢，毛爷爷说过，一切反动派都是纸老虎~与 **nativeRowBytes** 对应的函数如下：

Bitmap.cpp

```
static jint Bitmap_rowBytes(JNIEnv* env, jobject, jlong bitmapHandle) {
     SkBitmap* bitmap = reinterpret_cast<SkBitmap*>(bitmapHandle)
     return static_cast<jint>(bitmap->rowBytes());
}
```

等等，我们好像发现了什么，原来 Bitmap 本质上就是一个 SkBitmap。。而这个 SkBitmap 也是大有来头，不信你瞧：Skia。啥也别说了，赶紧瞅瞅 SkBitmap。

SkBitmap.h

```
/** Return the number of bytes between subsequent rows of the bitmap. */
size_t rowBytes() const { return fRowBytes; }
```

SkBitmap.cpp

```
size_t SkBitmap::ComputeRowBytes(Config c, int width) {
    return SkColorTypeMinRowBytes(SkBitmapConfigToColorType(c), width);
 }
SkImageInfo.h

static int SkColorTypeBytesPerPixel(SkColorType ct) {
   static const uint8_t gSize[] = {
    0,  // Unknown
    1,  // Alpha_8
    2,  // RGB_565
    2,  // ARGB_4444
    4,  // RGBA_8888
```

```
    4,  // BGRA_8888
    1,  // kIndex_8
  };
  SK_COMPILE_ASSERT(SK_ARRAY_COUNT(gSize) == (size_t)(kLastEnum_SkColorType + 1),
                    size_mismatch_with_SkColorType_enum);

  SkASSERT((size_t)ct < SK_ARRAY_COUNT(gSize));
  return gSize[ct];
}

static inline size_t SkColorTypeMinRowBytes(SkColorType ct, int width) {
    return width * SkColorTypeBytesPerPixel(ct);
}
```

好，跟踪到这里，我们发现 ARGB_8888（也就是我们最常用的 Bitmap 的格式）的一个像素占用 4byte，那么 rowBytes 实际上就是 4*width bytes。

那么结论出来了，一张 ARGB_8888 的 Bitmap 占用内存的计算公式

bitmapInRam = bitmapWidth*bitmapHeight *4 bytes

说到这儿你以为故事就结束了么？有本事你拿去试，算出来的和你获取到的总是会差个倍数，为啥呢？

还记得我们最开始给出的那个例子么？

一张 **522*686** 的 **PNG** 图片，我把它放到 **drawable-xxhdpi** 目录下，在**三星 s6** 上加载，占用内存 2547360B，就可以用这个方法获取到。

然而公式计算出来的可是 1432368B。。。

2.2 Density

知道我为什么在举例的时候那么费劲的说放到 xxx 目录下，还要说用 xxx 手机么？你以为 Bitmap 加载只跟宽高有关么？Naive。

还是先看代码，我们读取的是 drawable 目录下面的图片，用的是 decodeResource 方法，该方法本质上就两步：

- 读取原始资源，这个调用了 **Resource.openRawResource** 方法，这个方法调用完成之后会对 TypedValue 进行赋值，其中包含了原始资源的 density 等信息；
- 调用 **decodeResourceStream** 对原始资源进行解码和适配。这个过程实际上就是原始资源的 density 到屏幕 density 的一个映射。

原始资源的 density 其实取决于资源存放的目录（比如 xxhdpi 对应的是 480），而屏幕 density 的赋值，请看下面这段代码：

BitmapFactory.java

```
public static Bitmap decodeResourceStream(Resources res, TypedValue value,
    InputStream is, Rect pad, Options opts) {
```

```
//实际上，我们这里的 opts 是 null 的，所以在这里初始化。
if (opts == null) {
    opts = new Options();
}

if (opts.inDensity == 0 && value != null) {
    final int density = value.density;
    if (density == TypedValue.DENSITY_DEFAULT) {
        opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;
    } else if (density != TypedValue.DENSITY_NONE) {
        opts.inDensity = density; //这里 density 的值如果对应资源目录为 hdpi 的话，就是 240
    }
}

if (opts.inTargetDensity == 0 && res != null) {
//请注意，inTargetDensity 就是当前的显示密度，比如三星 s6 时就是 640
    opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
}

return decodeStream(is, pad, opts);
}
```

我们看到 opts 这个值被初始化，而它的构造居然如此简单：

```
public Options() {
    inDither = false;
    inScaled = true;
    inPremultiplied = true;
}
```

所以我们就很容易的看到，Option.inScreenDensity 这个值没有被初始化，而实际上后面我们也会看到这个值根本不会用到；我们最应该关心的是什么呢？是 inDensity 和 inTargetDensity，这两个值与下面 cpp 文件里面的 density 和 targetDensity 相对应——重复一下，inDensity 就是原始资源的 density，inTargetDensity 就是屏幕的 density。

紧接着，用到了 nativeDecodeStream 方法，不重要的代码直接略过，直接给出最关键的 doDecode 函数的代码：

BitmapFactory.cpp

```
static jobject doDecode(JNIEnv* env, SkStreamRewindable* stream, jobject padding, jobject options) {

......
```

```
    if (env->GetBooleanField(options, gOptions_scaledFieldID)) {
        const int density = env->GetIntField(options, gOptions_densityFieldID);//对应 hdpi 的时候，是 240
        const int targetDensity = env->GetIntField(options, gOptions_targetDensityFieldID);//三星 s6 的为 640
        const int screenDensity = env->GetIntField(options, gOptions_screenDensityFieldID);
        if (density != 0 && targetDensity != 0 && density != screenDensity) {
            scale = (float) targetDensity / density;
        }
    }
}
```

```
const bool willScale = scale != 1.0f;
......
SkBitmap decodingBitmap;
if (!decoder->decode(stream, &decodingBitmap, prefColorType,decodeMode)) {
   return nullObjectReturn("decoder->decode returned false");
}
//这里这个 deodingBitmap 就是解码出来的 bitmap，大小是图片原始的大小
int scaledWidth = decodingBitmap.width();
int scaledHeight = decodingBitmap.height();
if (willScale && decodeMode != SkImageDecoder::kDecodeBounds_Mode) {
    scaledWidth = int(scaledWidth * scale + 0.5f);
    scaledHeight = int(scaledHeight * scale + 0.5f);
}
if (willScale) {
    const float sx = scaledWidth / float(decodingBitmap.width());
    const float sy = scaledHeight / float(decodingBitmap.height());

    // TODO: avoid copying when scaled size equals decodingBitmap size
    SkColorType colorType = colorTypeForScaledOutput(decodingBitmap.colorType());
    // FIXME: If the alphaType is kUnpremul and the image has alpha, the
    // colors may not be correct, since Skia does not yet support drawing
    // to/from unpremultiplied bitmaps.
    outputBitmap->setInfo(SkImageInfo::Make(scaledWidth, scaledHeight,
            colorType, decodingBitmap.alphaType()));
```

```
    if (!outputBitmap->allocPixels(outputAllocator, NULL)) {
        return nullObjectReturn("allocation failed for scaled bitmap");
    }

    // If outputBitmap's pixels are newly allocated by Java, there is no need
    // to erase to 0, since the pixels were initialized to 0.
    if (outputAllocator != &javaAllocator) {
        outputBitmap->eraseColor(0);
    }

    SkPaint paint;
    paint.setFilterLevel(SkPaint::kLow_FilterLevel);

    SkCanvas canvas(*outputBitmap);
    canvas.scale(sx, sy);
    canvas.drawBitmap(decodingBitmap, 0.0f, 0.0f, &paint);
}
......
}
```

注意到其中有个 density 和 targetDensity，前者是 decodingBitmap 的 density，这个值跟这张图片的放置的目录有关（比如 hdpi 是 240，xxhdpi 是 480），这部分代码我跟了一下，太长了，就不列出来了；targetDensity 实际上是我们加载图片的目标 density，这个值的来源我们已经在前面给出了，就是 DisplayMetrics 的 densityDpi，如果是三星 s6 那么这个数值就是 640。sx 和 sy 实际上是约等于 scale 的，因为 scaledWidth 和 scaledHeight 是由 width 和 height 乘以 scale 得到的。我们看到 Canvas 放大了 scale 倍，然后又把读到内存的这张 bitmap 画上去，相当于把这张 bitmap 放大了 scale 倍。

再来看我们的例子：

一张 **522*686** 的 **PNG** 图片，我把它放到 **drawable-xxhdpi** 目录下，在**三星 s6** 上加载，占用内存 2547360B，其中 density 对应 xxhdpi 为 480，targetDensity 对应三星 s6 的密度为 640：

522/480 * *640* * 686/480 **640* * 4 = 2546432B

2.3 精度

越来越有趣了是不是，你肯定会发现我们这么细致的计算还是跟获取到的数值

不！一！样！

为什么呢？由于结果已经非常接近，我们很自然地想到精度问题。来，再把上面这段代码中的一句拿出来看看：

```
outputBitmap->setInfo(SkImageInfo::Make(scaledWidth, scaledHeight,
            colorType, decodingBitmap.alphaType()));
```

我们看到最终输出的 outputBitmap 的大小是 scaledWidth*scaledHeight，我们把这两个变量计算的片段拿出来给大家一看就明白了：

```
if (willScale && decodeMode != SkImageDecoder::kDecodeBounds_Mode) {
    scaledWidth = int(scaledWidth * scale + 0.5f);
    scaledHeight = int(scaledHeight * scale + 0.5f);
}
```

在我们的例子中，

scaledWidth = int(522 * 640 / 480f + 0.5) = int(696.5) = 696

scaledHeight = int(686 * 640 / 480f + 0.5) = int(915.16666…) = 915

下面就是见证奇迹的时刻：

915 * *696* * 4 = 2547360

有木有很兴奋！有木有很激动！！

写到这里，突然想起《STL 源码剖析》一书的扉页，侯捷先生只写了一句话：

> “源码之前，了无秘密”。

2.4 小结

其实，通过前面的代码跟踪，我们就不难知道，Bitmap 在内存当中占用的大小其实取决于：

- 色彩格式，前面我们已经提到，如果是 ARGB8888 那么就是一个像素 4 个字节，如果是 RGB565 那就是 2 个字节
- 原始文件存放的资源目录（是 hdpi 还是 xxhdpi 可不能傻傻分不清楚哈）
- 目标屏幕的密度（所以同等条件下，红米在资源方面消耗的内存肯定是要小于三星 S6 的）

3、想办法减少 Bitmap 内存占用

3.1 Jpg 和 Png

说到这里，肯定会有人会说，我们用 jpg 吧，jpg 格式的图片不应该比 png 小么？

这确实是个好问题，因为同样一张图片，jpg 确实比 png 会多少小一些（甚至很多），原因很简单，jpg 是一种**有损**压缩的图片**存储格式**，而 png 则是 **无损**压缩的图片**存储格式**，显而易见，jpg 会比 png 小，代价也是显而易见的。

可是，这说的是文件存储范畴的事情，它们只存在于文件系统，而非内存或者显存。说得简单一点儿，我有一个极品飞车的免安装硬盘版的压缩包放在我的磁盘里面，这个游戏是不能玩的，我需要先解压，才能玩——jpg 也好，png 也好就是个压缩包的概念，而我们讨论的内存占用则是从使用角度来讨论的。

所以，jpg 格式的图片与 png 格式的图片在内存当中不应该有什么不同。

> 『啪！！！』
>
> 『谁这么缺德！！打人不打脸好么！』

肯定有人有意见，jpg 图片读到内存就是会小，还会给我拿出例子。当然，他说的不一定是错的。因为 jpg 的图片没有 alpha 通道！！所以读到内存的时候如果用 RGB565 的格式存到内存，这下大小只有 ARGB8888 的一半，能不小么。。。

不过，抛开 Android 这个平台不谈，从出图的角度来看的话，jpg 格式的图片大小也不一定比 png 的小，这要取决于图像信息的内容：

> JPG 不适用于所含颜色很少、具有大块颜色相近的区域或亮度差异十分明显的较简单的图片。对于需要高保真的较复杂的图像，PNG 虽然能无损压缩，但图片文件较大。

如果仅仅是为了 Bitmap 读到内存中的大小而考虑的话，jpg 也好 png 也好，没有什么实质的差别；二者的差别主要体现在：

- alpha 你是否真的需要？如果需要 alpha 通道，那么没有别的选择，用 png。
- 你的图色值丰富还是单调？就像刚才提到的，如果色值丰富，那么用 jpg，如果作为按钮的背景，请用 png。
- 对安装包大小的要求是否非常严格？如果你的 app 资源很少，安装包大小问题不是很凸显，看情况选择 jpg 或者 png（不过，我想现在对资源文件没有苛求的应用会很少吧。。）
- 目标用户的 cpu 是否强劲？jpg 的图像压缩算法比 png 耗时。这方面还是要酌情选择，前几年做了一段时间 Cocos2dx，由于资源非常多，项目组要求统一使用 png，可能就是出于这方面的考虑。

嗯，跑题了，我们其实想说的是怎么减少内存占用的。。这一小节只是想说，休想通过这个方法来减少内存占用。。。XD

3.2 使用 inSampleSize

有些朋友一看到这个肯定就笑了。采样嘛，我以前是学信号处理的，一看到 Sample 就抽抽。。哈哈开个玩笑，这个采样其实就跟统计学里面的采样是一样的，在保证最终效果满足要求的前提下减少样本规模，方便后续的数据采集和处理。

这个方法主要用在图片资源本身较大，或者适当地采样并不会影响视觉效果的条件下，这时候我们输出地目标可能相对较小，对图片分辨率、大小要求不是非常的严格。

举个例子

我们现在有个需求，要求将一张图片进行模糊，然后作为 ImageView 的 src 呈现给用户，而我们的原始图片大小为 1080*1920，如果我们直接拿来模糊的话，一方面模糊的过程费时费力，另一方面生成的图片又占用内存，实际上在模糊运算过程中可能会存在输入和输出并存的情况，此时内存将会有一个短暂的峰值。

这时候你一定会想到三个字母在你的脑海里挥之不去，它们就是『OOM』。

既然图片最终是要被模糊的，也看不太情况，还不如直接用一张采样后的图片，如果采样率为 2，那么读出来的图片只有原始图片的 1/4 大小，真是何乐而不为呢？？

```
BitmapFactory.Options options = new Options();
options.inSampleSize = 2;
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), resId, options);
```

3.3 使用矩阵

用到 Bitmap 的地方，总会见到 Matrix。这时候你会想到什么？

> 『基友』
>
> 『是在下输了。。』

其实想想，Bitmap 的像素点阵，还不就是个矩阵，真是你中有我，我中有你的交情啊。那么什么时候用矩阵呢？

大图小用用采样，小图大用用矩阵。

还是用前面模糊图片的例子，我们不是采样了么？内存是小了，可是图的尺寸也小了啊，我要用 Canvas 绘制这张图可怎么办？当然是用矩阵了：

方式一：

```
Matrix matrix = new Matrix();
matrix.preScale(2, 2, 0f, 0f);
//如果使用直接替换矩阵的话，在 Nexus6 5.1.1 上必须关闭硬件加速
canvas.concat(matrix);
canvas.drawBitmap(bitmap, 0,0, paint);
```

需要注意的是，在使用搭载 5.1.1 原生系统的 Nexus6 进行测试时发现，如果使用 Canvas 的 setMatrix 方法，可能会导致与矩阵相关的元素的绘制存在问题，本例当中如果使用 setMatrix 方法，bitmap 将不会出现在屏幕上。因此请尽量使用 canvas 的 scale、rotate 这样的方法，或者使用 concat 方法。

方式二：

```
Matrix matrix = new Matrix();
matrix.preScale(2, 2, 0, 0);
canvas.drawBitmap(bitmap, matrix, paint);
```

这样，绘制出来的图就是放大以后的效果了，不过占用的内存却仍然是我们采样出来的大小。

如果我要把图片放到 ImageView 当中呢？一样可以，请看：

```
Matrix matrix = new Matrix();
matrix.postScale(2, 2, 0, 0);
imageView.setImageMatrix(matrix);
imageView.setScaleType(ScaleType.MATRIX);
imageView.setImageBitmap(bitmap);
```

3.4 合理选择 Bitmap 的像素格式

其实前面我们已经多次提到这个问题。ARGB8888 格式的图片，每像素占用 4 Byte，而 RGB565 则是 2 Byte。我们先看下有多少种格式可选：

格式	描述
ALPHA_8	只有一个 alpha 通道
ARGB_4444	这个从 API 13 开始不建议使用，因为质量太差
ARGB_8888	ARGB 四个通道，每个通道 8bit
RGB_565	每个像素占 2Byte，其中红色占 5bit，绿色占 6bit，蓝色占 5bit

这几个当中，

ALPHA8 没必要用，因为我们随便用个颜色就可以搞定的。

ARGB4444 虽然占用内存只有 **ARGB8888** 的一半，不过已经被官方嫌弃，失宠了。。『又要占省内存，又要看着爽，臣妾做不到啊 T T』。

ARGB8888 是最常用的，大家应该最熟悉了。

RGB565 看到这个，我就看到了资源优化配置无处不在，这个绿色。。（不行了，突然好邪恶 XD），其实如果不需要 alpha 通道，特别是资源本身为 jpg 格式的情况下，用这个格式比较理想。

3.5 高能：索引位图(Indexed Bitmap)

索引位图，每个像素只占 1 Byte，不仅支持 RGB，还支持 alpha，而且看上去效果还不错！等等，请收起你的口水，Android 官方并不支持这个。是的，你没看错，官方并不支持。

```
public enum Config {
    // these native values must match up with the enum in SkBitmap.h

    ALPHA_8     (2),
    RGB_565     (4),
    ARGB_4444   (5),
    ARGB_8888   (6);

    final int nativeInt;
}
```

不过，Skia 引擎是支持的，不信你再看：

```
enum Config {
   kNo_Config,   //!< bitmap has not been configured
    kA8_Config,   //!< 8-bits per pixel, with only alpha specified (0 is transparent, 0xFF is opaque)

   //看这里看这里！！↓↓↓↓↓
    kIndex8_Config, //!< 8-bits per pixel, using SkColorTable to specify the colors
    kRGB_565_Config, //!< 16-bits per pixel, (see SkColorPriv.h for packing)
    kARGB_4444_Config, //!< 16-bits per pixel, (see SkColorPriv.h for packing)
    kARGB_8888_Config, //!< 32-bits per pixel, (see SkColorPriv.h for packing)
    kRLE_Index8_Config,

    kConfigCount
};
```

其实 Java 层的枚举变量的 nativeInt 对应的就是 Skia 库当中枚举的索引值，所以，如果我们能够拿到这个索引是不是就可以了？对不起，拿不到。

不行了，废话这么多，肯定要挨板砖了T T。

不过呢，在 png 的解码库里面有这么一段代码：

```
bool SkPNGImageDecoder::getBitmapColorType(png_structp png_ptr, png_infop info_ptr,
```

```
                                    SkColorType* colorTypep,
                                    bool* hasAlphap,
                                    SkPMColor* SK_RESTRICT theTranspColorp) {
png_uint_32 origWidth, origHeight;
int bitDepth, colorType;
png_get_IHDR(png_ptr, info_ptr, &origWidth, &origHeight, &bitDepth,
             &colorType, int_p_NULL, int_p_NULL, int_p_NULL);

#ifdef PNG_sBIT_SUPPORTED
  // check for sBIT chunk data, in case we should disable dithering because
  // our data is not truely 8bits per component
  png_color_8p sig_bit;
  if (this->getDitherImage() && png_get_sBIT(png_ptr, info_ptr, &sig_bit)) {
#if 0
    SkDebugf("----- sBIT %d %d %d %d\n", sig_bit->red, sig_bit->green,
             sig_bit->blue, sig_bit->alpha);
#endif
    // 0 seems to indicate no information available
    if (pos_le(sig_bit->red, SK_R16_BITS) &&
        pos_le(sig_bit->green, SK_G16_BITS) &&
        pos_le(sig_bit->blue, SK_B16_BITS)) {
        this->setDitherImage(false);
    }
}
#endif

if (colorType == PNG_COLOR_TYPE_PALETTE) {
    bool paletteHasAlpha = hasTransparencyInPalette(png_ptr, info_ptr);
    *colorTypep = this->getPrefColorType(kIndex_SrcDepth, paletteHasAlpha);
    // now see if we can upscale to their requested colortype
    //这段代码，如果返回 false，那么 colorType 就被置为索引了，那么我们看看如何返回 false
    if (!canUpscalePaletteToConfig(*colorTypep, paletteHasAlpha)) {
        *colorTypep = kIndex_8_SkColorType;
    }
```

```
} else {
......
}
return true;
}
```

canUpscalePaletteToConfig 函数如果返回 false，那么 **colorType** 就被置为 **kIndex_8_SkColorType** 了。

```
static bool canUpscalePaletteToConfig(SkColorType dstColorType, bool srcHasAlpha) {
  switch (dstColorType) {
    case kN32_SkColorType:
    case kARGB_4444_SkColorType:
        return true;
    case kRGB_565_SkColorType:
        // only return true if the src is opaque (since 565 is opaque)
        return !srcHasAlpha;
    default:
        return false;
  }
}
```

如果传入的 **dstColorType** 是 **kRGB_565_SkColorType**，同时图片还有 alpha 通道，那么返回 false~~咳咳，那么问题来了，这个 **dstColorType** 是哪儿来的？？就是我们在 decode 的时候，传入的 **Options** 的 **inPreferredConfig**。

下面是实验时间~

准备：在 assets 目录当中放了一个叫 index.png 的文件，大小 192*192，这个文件是通过 PhotoShop 编辑之后生成的索引格式的图片。

代码：

```
try {
   Options options = new Options();
   options.inPreferredConfig = Config.RGB_565;
Bitmap bitmap = BitmapFactory.decodeStream(getResources().getAssets().open("index.png"), null, options);
   Log.d(TAG, "bitmap.getConfig() = " + bitmap.getConfig());
   Log.d(TAG, "scaled bitmap.getByteCount() = " + bitmap.getByteCount());
   imageView.setImageBitmap(bitmap);
} catch (IOException e) {
    e.printStackTrace();
```

```
}
```

程序运行在 Nexus6 上，由于从 assets 中读取不涉及前面讨论到的 scale 的问题，所以这张图片读到内存以后的大小理论值（ARGB8888）：
192 * *192* *4=147456

好，运行我们的代码，看输出的 Config 和 ByteCount：

```
D/MainActivity: bitmap.getConfig() = null
```

```
D/MainActivity: scaled bitmap.getByteCount() = 36864
```

先说大小为什么只有 36864，我们知道如果前面的讨论是没有问题的话，那么这次解码出来的 Bitmap 应该是索引格式，那么占用的内存只有 **ARGB 8888** 的 1/4 是意料之中的；再说 Config 为什么为 null。。额。。黑户。。官方说：

> public final Bitmap.Config getConfig ()
>
> Added in API level 1
>
> If the bitmap's internal config is in one of the public formats, return that config, otherwise return null.

再说一遍，黑户。。XD。

看来这个法子还真行啊，占用内存一下小很多。不过由于官方并未做出支持，因此这个方法有诸多限制，比如不能在 xml 中直接配置，，生成的 Bitmap 不能用于构建 Canvas 等等。

3.6 不要辜负。。。『哦，不要姑父！』

其实我们一直在抱怨资源大，有时候有些场景其实不需要图片也能完成的。比如在开发中我们会经常遇到 Loading，这些 Loading 通常就是几帧图片，图片也比较简单，只需要黑白灰加 alpha 就齐了。

> 『排期太紧了，这些给我出一系列图吧』
>
> 『好，不过每张图都是 300*30 0 的 png 哈，总共 5 张，为了适配不同的分辨率，需要出 xxhdpi 和 xxxhdpi 的两套图。。』

Orz。。。

如果是这样，你还是自定义一个 View，覆写 onDraw 自己画一下好了。。。

4、结语

写了这么多，我们来稍稍理一理，本文主要讨论了如何运行时获取 Bitmap 占用内存的大小，如果事先根据 Bitmap 的格式、读取方式等算出其占用内存的大小，后面又整理了一些常见的 Bitmap 使用建议。突然好像说，是时候研究一下 Skia 引擎了。

怎么办，看来扔了好几年的 C++还是要捡回来么。。噗。。。

阅读原文