一、前言

在 Android 中进行图片压缩是非常常见的开发场景，主要的压缩方法有两种：其一是质量压缩，其二是下采样压缩。

前者是在不改变图片尺寸的情况下，改变图片的存储体积，而后者则是降低图像尺寸，达到相同目的。

由于本文的篇幅问题，分为上下两篇发布。

二、Android 质量压缩逻辑

在Android中，对图片进行质量压缩，通常我们的实现方式如下所示：

```
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
//quality 为0～100，0表示最小体积，100表示最高质量，对应体积也是最大
bitmap.compress(Bitmap.CompressFormat.JPEG, 100, outputStream);
```

在上述代码中，我们选择的压缩格式是`CompressFormat.JPEG`，除此之外还有两个选择：

其一，`CompressFormat.PNG`， PNG 格式是无损的，它无法再进行质量压缩，quality 这个参数就没有作用了，会被忽略，所以最后图片保存成的文件大小不会有变化；
其二，`CompressFormat.WEBP`，这个格式是 google 推出的图片格式，它会比 JPEG 更加省空间，经过实测大概可以优化 30% 左右。

由于项目原因和兼容性选择了JPEG，因此接下来的分析也将是围绕 JPEG 展开。

将 PNG 图片转成 JPEG 格式之后不会降低这个图片的尺寸，但是会降低视觉质量，从而降低存储体积。同时，由于尺寸不变，所以将这个图片解码成相同色彩模式的 bitmap 之后，占用的内存大小和压缩前是一样的。

回到最初的代码示例，函数`compress`经过一连串的 java 层调用之后，最后来到了一个 native 函数，如下：

```
//Bitmap.cpp
static jboolean Bitmap_compress(JNIEnv* env, jobject clazz, jlong bitmapHandle,
                                jint format, jint quality,
                                jobject jstream, jbyteArray jstorage) {

    LocalScopedBitmap bitmap(bitmapHandle);
    SkImageEncoder::Type fm;

    switch (format) {
    case kJPEG_JavaEncodeFormat:
        fm = SkImageEncoder::kJPEG_Type;
        break;
    case kPNG_JavaEncodeFormat:
        fm = SkImageEncoder::kPNG_Type;
        break;
    case kWEBP_JavaEncodeFormat:
        fm = SkImageEncoder::kWEBP_Type;
        break;
    default:
        return JNI_FALSE;
    }
```

```
    if (!bitmap.valid()) {
        return JNI_FALSE;
    }

    bool success = false;

    std::unique_ptr<SkWStream> strm(CreateJavaOutputStreamAdaptor(env, jstream,
jstorage));
    if (!strm.get()) {
        return JNI_FALSE;
    }

    std::unique_ptr<SkImageEncoder> encoder(SkImageEncoder::Create(fm));
    if (encoder.get()) {
        SkBitmap skbitmap;
        bitmap->getSkBitmap(&skbitmap);
        success = encoder->encodeStream(strm.get(), skbitmap, quality);
    }
    return success ? JNI_TRUE : JNI_FALSE;
}
```

可以看到最后调用了函数 `encoder->encodeStream(....)` 编码保存本地。该函数是调用 skia 引擎来对图片进行编码压缩，对 skia 的介绍将在后文展开。

一段完整的示例代码如下：

```
// R.drawable.thumb 为 png 图片
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.thumb);
try {
    //保存压缩图片到本地
    File file = new File(Environment.getExternalStorageDirectory(), "aaa.jpg");
    if (!file.exists()) {
        file.createNewFile();
    }
    FileOutputStream fs = new FileOutputStream(file);
    bitmap.compress(Bitmap.CompressFormat.JPEG, 50, fs);
    Log.i(TAG, "onCreate: file.length " + file.length());
    fs.flush();
    fs.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
//查看压缩之后的 Bitmap 大小
ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.JPEG, 50, outputStream);
byte[] bytes = outputStream.toByteArray();
Bitmap compress = BitmapFactory.decodeByteArray(bytes, 0, bytes.length);
Log.i(TAG, "onCreate: bitmap.size = " + bitmap.getByteCount() + "   compress.size
= " + compress.getByteCount());
```

首先，我们来看看 quality 参数被设置为 50，质量压缩前后的图片对比，可以看到其尺寸大小并没有变化，但是视觉感受也可以明显地看到图片变的模糊了一些。

通过日志也可以看到，在质量压缩前后图片**转成 Bitmap 之后在内存中的大小也并没有变化**，这是在保持像素的前提下，改变图片的位深及透明度等：

```
//压缩之后图片占用的存储体积
compress.length = 7814
//在内存中压缩前后图片占用的大小
bitmap.size = 350000   compress.size = 350000
```

对比二者，保存前的图片存储体积是 106k，质量设为 50 并且保存为 JPEG 格式之后，图片存储大小就只有 8k 了，并且质量设的越低，保存成文件之后，文件的体积也就越小。

三、Android Skia 图像引擎

在上文中，提到的Skia是Android 的重要组成部分。

Skia 是一个 Google 自己维护的 c++ 实现的图像引擎，实现了各种图像处理功能，并且广泛地应用于谷歌自己和其它公司的产品中（如：Chrome、Firefox、 Android等），基于它可以很方便为操作系统、浏览器等开发图像处理功能。

Skia 在 Android 中提供了基本的画图和简单的编解码功能，可以挂接其他的第三方编码解码库或者硬件编解码库，例如 libpng 和 libjpeg，libgif 等等。因此，这个函数调用 `bitmap.compress(Bitmap.CompressFormat.JPEG...)`，实际会调用 libjpeg.so 动态库进行编码压缩。

最终 Android 编码保存图片的逻辑是 Java 层函数→Native 函数→Skia函数→对应第三库函数（例如 libjpeg）。所以 skia 就像一个胶水层，用来链接各种第三方编解码库，不过 Android 也会对这些库做一些修改，比如修改内存管理的方式等等。

Android 在之前从某种程度来说使用的算是 libjpeg 的功能阉割版，压缩图片默认使用的是 standard huffman，而不是 optimized huffman，也就是说使用的是默认的哈夫曼表，并没有根据实际图片去计算相对应的哈夫曼表，Google 在初期考虑到手机的性能瓶颈，计算图片权重这个阶段非常占用 CPU 资源的同时也非常耗时，因为此时需要计算图片所有像素 argb 的权重，这也是 Android 的图片压缩率对比 iOS 来说差了一些的原因之一。

四、图像压缩与 Huffman 算法

这里简单介绍一下哈夫曼算法，哈夫曼算法是在多媒体处理里常用的算法之一。比如一个文件中可能会出现五个值 a,b,c,d,e，它们用二进制表达是：

a. 1010
b. 1011
c. 1100
d. 1101
e. 1110

我们可以看到，最前面的一位数字是 1，其实是浪费掉了，在定长算法下最优的表达式为：

a. 010
b. 011
c. 100
d. 101
e. 110

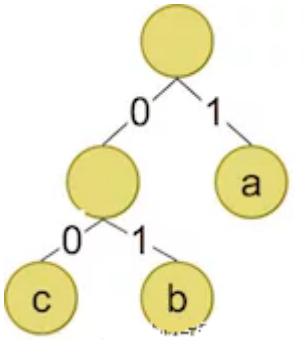
这样我们就能做到节省一位的损耗，那哈夫曼算法比起定长算法改进的地方在哪里呢？在哈夫曼算法中我们可以给信息赋予权重，即为信息加权，假设 a 占据了 60%，b 占据了 20%， c 占据了 20%，d,e 都是 0%：

a:010（60%）
b:011（20%）
c:100（20%）
d:101（0%）
e:110（0%）

在这种情况下，我们可以使用哈夫曼树算法再次优化为：

a:1
b:01
c:00

所以思路当然就是出现频率高的字母使用短码，对出现频率低的使用长码，不出现的直接就去掉，最后 abcde 的哈夫曼编码就对应：1 01 00
通过权重对应生成的的哈夫曼表为：



定长编码下的abcde：010 011 100 101 110，使用哈夫曼树加权后的编码则为 1 01 00，这就是哈夫曼算法的整体思路（关于算法的详细介绍可以去查阅相关资料）。

所以这个算法一个很重要的思路是必须知道每一个元素出现的权重，如果我们能够知道每一个元素的权重，那么就能够根据权重动态生成一个最优的哈夫曼表。

但是怎么去获取每一个元素，对于图片就是每一个像素中 argb 的权重呢，只能去循环整个图片的像素信息，这无疑是非常消耗性能的，所以早期 android 就使用了默认的哈夫曼表进行图片压缩。

五、libjpeg 与 optimize_coding

libjpeg 在压缩图像时，有一个参数叫 `optimize_coding`，关于这个参数，libjpeg.doc 有如下解释：

> TRUE causes the compressor to compute optimal Huffman coding tables for the image. This requires an extra pass over the data and therefore costs a good deal of space and time. The default is FALSE, which tells the compressor to use the supplied or default Huffman tables. In most cases optimal tables save only a few percent of file size compared to the default tables. Note that when this is TRUE, you need not supply Huffman tables at all, and any you do supply will be overwritten.

由上可知，如果设置 `optimize_coding` 为 TRUE，将会使得压缩图像过程中，会先基于图像数据计算哈弗曼表。由于这个计算会显著消耗空间和时间，默认值被设置为 FALSE。

那么 `optimize_coding` 参数的影响究竟会有多大呢？查阅一些博客资料介绍，使用相同的原始图片，分别设置 optimize_coding=TRUE 和 FALSE 进行压缩，发现 FALSE 时的图片大小大约是 TRUE 时的 5-10 倍。换言之就是相同文件体积的图片，不使用哈夫曼编码图片质量会比使用哈夫曼低 5-10 倍。

关于这个差异我们再去查阅其他资料，发现有两篇讨论非常热烈：Investigate using “optimize_coding” when encoding to JPEG，About libjpeg optimize_coding，甚至Skia 的官方人员也参与了讨论，他据此测试了两组数据：

> sample image 1 (RGB gradients):
> default (80): 2.5x slower, 34% smaller
> quality 0: 1.7x slower, 52% smaller
> quality 20: 2.1x slower, 55% smaller
> quality 40: 2.3x slower, 37% smaller
> quality 60: 2.5x slower, 36% smaller
> quality 100: 3.9x slower, 22% smaller
>
> sample image 2 (photo):
> default (80): 2x slower, 8% smaller
> quality 0: 1.5x slower, 49% smaller
> quality 20: 1.7x slower, 22% smaller
> quality 40: 1.9x slower, 15% smaller
> quality 60: 1.9x slower, 11% smaller
> quality 100: 2x slower, 9% smaller

可以看到效果并不是 5-10 倍的体积差距，最多也就在 2 倍而已，有国人也测试了一下，结果一致：JPEG Optimized Huffman。

尽管如此，社区里对此的疑虑并没有彻底打消，最终，官方人员修改了这个默认的实现：skia / skia.git / 0a35620a16b368356888d15771392fb00cbb777d。在 `SkImageDecoder_libjpeg.cpp` 文件中给 optimize_code 赋值了一个默认值 TRUE。

六、Android 与 optimize_coding

那么在 Android 中有没有使用哈夫曼变长编码呢？查阅了 7.0 源码，如下：

```
/* Use Huffman coding, not arithmetic coding, by default */
cinfo->arith_code = FALSE;
```

可以看到注释里面很清楚，默认是哈夫曼变长编码，而不是算数编码。同时去查阅 14 年时的 Android 4.4 源码，发现依旧如此。

对于`optimize_coding`，早期的 Android 考虑到性能瓶颈，将其设置为 FALSE。但是，现在 Android 手机性能比以前好很多，所以目前性能往往不是瓶颈，时间和压缩质量反而成为更重要的指标了。为此，Google 在 Android 7.0 版本左右，也做了相应修改，如 7.0 和 6.0 源码所示：

七、Android JPEG VS. iOS JPEG

经过上面的介绍大家应该了解了为什么 Android 的 JPEG 图片压缩率会比 iOS 小一些，那么还有另一个问题就是为什么同一张 PNG 图片设置成同样的压缩质量压缩成 JPEG 之后，Android 输出的图像质量会比 iOS 差一些呢，经过相关资料的查找，发现造成这个结果有两方面的因素。

第一个因素是 JPEG 编码过程中有一个步骤是颜色空间 RGB -> YUV 的转换，之前的 Android 版本同样考虑到性能问题，skia 引擎写了一个函数替代了原来 libjpeg 的转换函数，好处是提高了编码速度，坏处就是牺牲了每一个像素的精度。

第二个因素是离散余弦变换有三种方式，Skia 引擎选择了 JDCT_IFAST，JDCT_IFAST 是最快的变换方式，当然也是精度最差的一种。

上面两种因素第一个会造成色调偏差，第二个会造成色块的出现，所以如果需要提高压缩之后的图像质量，可以考虑从这两方面入手。

八、总结

首先，从 Android 7.0 版本开始，`optimize_code` 标示已经设置为了 TRUE，也就是默认使用图像生成哈夫曼表，而不是使用默认哈夫曼表。而至于这个标志所产生的体积差距也没有 5-10 倍那么大，大约可以在原图的基础上缩小 10%～50% 的体积，经过修改前后不同 Android 版本实测，数据吻合。

其次，如何提高 Android 的压缩率，这里需要提到两个库，一个是 mozilla/mozjpeg，另一个是 libjpeg-turbo，前者是一个来自 Mozilla 实验室的 JPEG 图像编码器项目，目标是在不降低图像质量且兼容主流的解码器的情况下，提供产品级的 JPEG 格式编码器来提高压缩率以减小 JPEG 文件的大小，后者相当于是一个 libjpeg 的增强版，前者也是基于后者，在后者的基础上进行了一些优化。

所以想要提升图片压缩率的可以从这两个库着手，网上资料也不少，后续有机会可以测试一下这两个库，然后给大家分享一下。

最后，编码方式除了哈夫曼之外，还有定长的算术编码，这个算法的详细介绍大家可以网上查阅一下。对比哈夫曼编码和算术编码，网上相关资料显示算术编码在压缩 jpeg 方面可以比哈夫曼编码体积小 5%～12%，所以需要提升图片压缩率的同样也可以尝试从切换成算术编码这方面入手。