一次基于BufferedImage的图片处理优化实践

1、引言

日常开发中接到这样的需求，上游系统请求获取一张A4单据用于仓库打印及展示，要求PNG图片格式，但是我们内部得到的单据格式为PDF，需要提取PDF文档的元素并生成一张PNG图片。目前已经有不少开源工具实现了这一功能，我们找了网上使用比较多的Apache PDFBox库来实现功能，如下

```Java
// Step 1
PDDocument document = PDDocument.load(content);
PDFRenderer pdfRenderer = new PDFRenderer(document);
// 获取第1页PDF文档
OutputStream os = new ByteArrayOutputStream()
// Step 2
// 为了保证图片的清晰，这里采用600DPI
BufferedImage image = pdfRenderer.renderImageWithDPI(0, 600);
// Step 3
ImageIO.write(image, "PNG", os);
```

实际测试时，明显感觉到卡顿，当一次请求的单据数目较多时尤其严重。

经统计，各步骤本机单次运行耗时如下：

pdf 初始化（Step 1）：2ms

文档提取及图片绘制（Step 2）：520ms

图片编码 (Step 3): 3823*ms*

我们发现，最后一句代码耗时接近4秒，拖累了整体性能。我们要如何优化这样一个问题呢？

2、BufferedImage介绍

在讨论优化问题之前，首先要搞清楚待优化的代码是做什么的。如上代码中，使用renderImageWithDPI方法，将文档元素绘制为BufferedImage对象。

The BufferedImage subclass describes an Image with an accessible buffer of image data. A BufferedImage is comprised of a ColorModel and a Raster of image data. The number and types of bands in the SampleModel of the Raster must match the number and types required by the ColorModel to represent its color and alpha components. All BufferedImage objects have an upper left corner coordinate of (0, 0). Any Raster used to construct a BufferedImage must therefore have minX=0 and minY=0.

This class relies on the data fetching and setting methods of Raster, and on the color characterization methods of ColorModel.

See Also: ColorModel, Raster, WritableRaster

```
public class BufferedImage extends java.awt.Image
                          implements WritableRenderedImage, Transparency
```

根据描述，BufferedImage用来描述一张图片，其内部保存了图片的颜色模型（ColorModel）及像素数据（Raster）。这里简单解释就是，内部的Raster实现类中，以某种数据结构（如Byte数组）表示图片的所有像素数据，而ColorModel实现类，则提供了将每个像素的数据，转换为对应RGB颜色的方式。

BufferedImage的构造函数中，可以传入图片类型来决定使用哪一种ColorModel和Raster。引言的示例中，PDFRender源码中默认生成的图片类型为 TYPE_INT_RGB，这种类型表示，每一个像素使用R、G、B三条数据表示，每条数据使用单字节（0~255）表示。

```Java
public BufferedImage(int width, int height, int imageType)
```

需要注意的是，BufferedImage并不表示某一张具体的位图，而是通过描述每个像素的数据，抽象地表达一张图片，因此，它可以在内存中通过操作像素数据，直接改变对应图片。而通过ImageIO.write方法，可以将BufferedImage编码为具体格式的图片数据流。此方法会根据formatName选择该文件格式的编码器，来对BufferedImage内部的像素数据进行编码。

```Java
public static boolean write(RenderedImage im, String formatName,
OutputStream output) throws IOException
```

以下代码为BufferedImage的简单应用

将一个GIF图片读取到BufferedImage中，在坐标（10，10）位置打出ABC三个字符，并重新编码成PNG图片

```Java
BufferedImage image = ImageIO.read(new File("exmaple.gif"));
image.getGraphics().drawString("ABC", 10, 10);
ImageIO.write(image, "PNG", new FileOutputStream("result.png"));
```

下面这段代码展示了另一类型的例子，它将图片中所有的红色像素点重置成黑色像素点

```Java
BufferedImage image = ImageIO.read(new File("example.gif"));
for(int i = 0 ; i < image.getWidth() ; i++) {
   for(int j = 0 ; j < image.getHeight() ; j++) {
       if(image.getRGB(i, j) == Color.RED.getRGB()) {
          image.setRGB(i, j, Color.BLACK.getRGB());
       }
   }
}
```

如果我们想要取得图片的数据，可以通过BufferedImage内部的Raster对象获得。下面的示例，展示了采用了字节数组形式存储时，取得内部存储的字节数组的方式。注意，当需要查询到某一个像素的数据时，需要综合像素的x,y坐标及ColorModel模型中像素数据的存储方式来决定数组下标。

```Java
BufferedImage im = ImageIO.read(new File("exmaple.gif"));
DataBuffer dataBuffer = im.getRaster().getDataBuffer();
if(dataBuffer instanceof DataBufferByte) {
     DataBufferByte bufferByte = (DataBufferByte) dataBuffer;
     byte[] data = bufferByte.getData();
}
```

那么，现在我们可以通过看源码，了解引言的示例代码的作用。

根据源码可以了解到，PDFRender对象读取并识别PDF文档中的每条语句，利用BufferedImage中的Graphics2D重新画了一张图片，并编码成PNG格式。这里不详细说了。

3、PNG文件格式浅析

根据上一节的内容可知，把BufferedImage编码成PNG文件的过程，耗时接近2秒。我们需要简单了解下编码PNG文件的过程中，究竟在干什么。

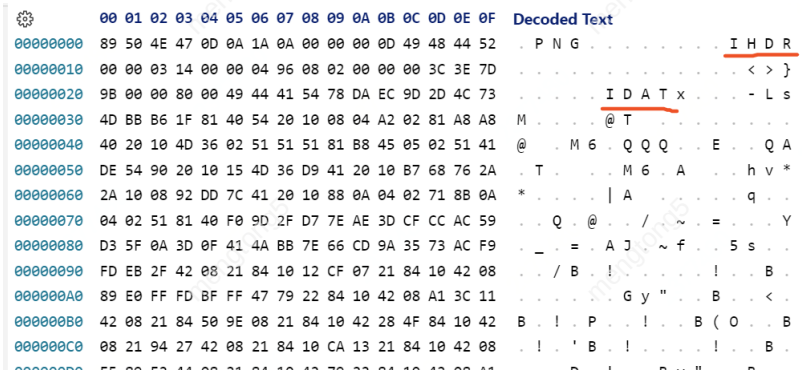
以下参考W3C上对PNG的描述 https://www.w3.org/TR/PNG/#11IHDR ，由于比较复杂，很多东西我也是一知半解，这里仅描述本次优化涉及到的主要内容。

PNG文件可以包含很多数据块，最主要且必须包含的，是IHDR,IDAT及IEND三个数据块

Table 5.3 — Chunk ordering rules

Critical chunks (shall appear in this order, except PLTE is optional)		
Chunk name	Multiple allowed	Ordering constraints
IHDR	No	Shall be first
PLTE	No	Before first IDAT
IDAT	Yes	Multiple IDAT chunks shall be consecutive
IEND	No	Shall be last

我们通过十六进制打开PNG文件，就可以看到具体的数据块分布



IEND

IEND为结束标志

IHDR

IHDR为文件头，其后紧跟的字节描述了PNG文件的一些基础属性，如宽、高各占4各字节，而Color type和Bit Depth分别表示颜色类型和位深。

The IHDR chunk shall be the first chunk in the PNG datastream. It contains:

Width	4 bytes
Height	4 bytes
Bit depth	1 byte
Colour type	1 byte
Compression method	1 byte
Filter method	1 byte
Interlace method	1 byte

（1）Colour type颜色类型分为以下几种：

Greyscale为灰度图，每个像素用单一的灰度值来描述颜色，灰度值由0（白）到255（黑）逐步加深。

Truecolor即为一般的RGB三通道图片，R、G、B每一个通道允许用8或16个比特来表示。

Indexed-color为索引色，需要配合调色板PLTE数据块使用，这里不多做介绍。

后面两种Greyscale with alpha, truecolor with alpha，顾名思义，即灰度和RGB图像增加透明度通道

Table 11.1 — Allowed combinations of colour type and bit depth

PNG image type	Colour type	Allowed bit depths	Interpretation
Greyscale	0	1, 2, 4, 8, 16	Each pixel is a greyscale sample
Truecolour	2	8, 16	Each pixel is an R,G,B triple
Indexed-colour	3	1, 2, 4, 8	Each pixel is a palette index; a PLTE chunk shall appear.
Greyscale with alpha	4	8, 16	Each pixel is a greyscale sample followed by an alpha sample.
Truecolour with alpha	6	8, 16	Each pixel is an R,G,B triple followed by an alpha sample.

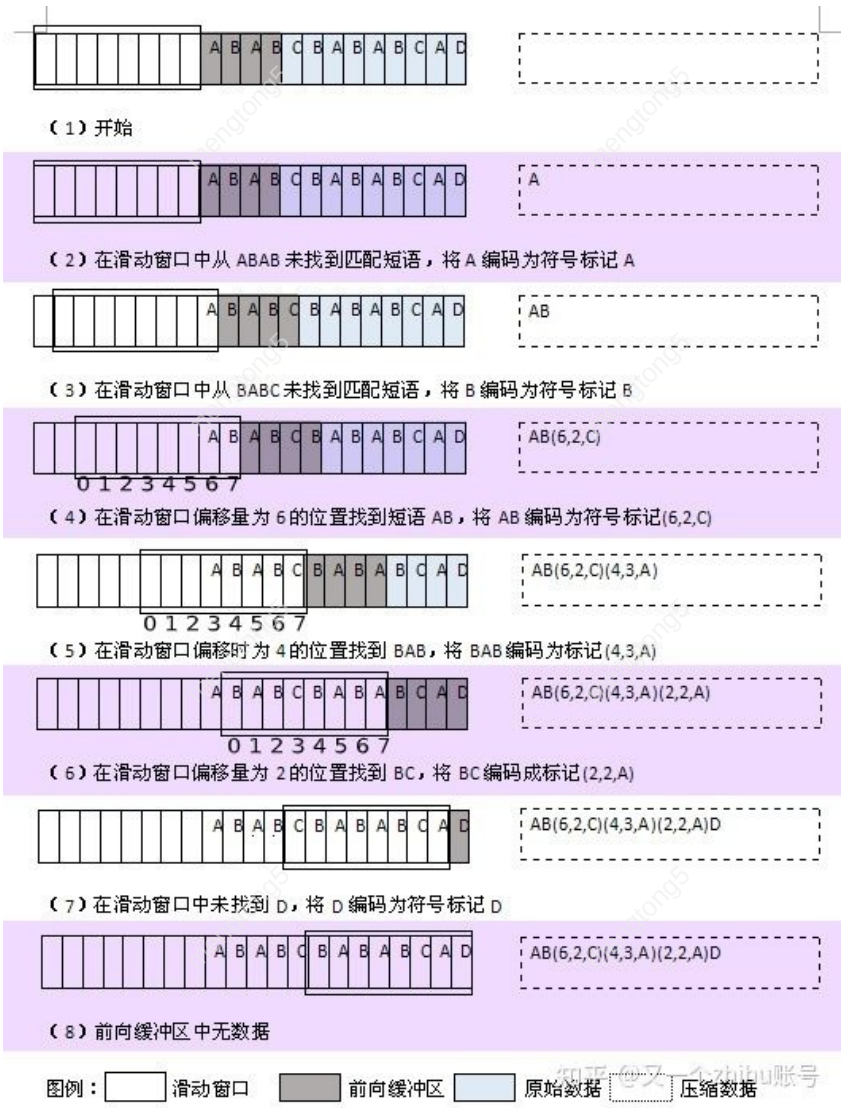
（2）Bit Depth（位深度），即每个通道使用多少比特来表示。

比如在一张Colour type=Greyscale中，一个像素由1~255的灰度值来表示，那么这张图片就是单通道8位深。

根据上表，我们知道位深度于颜色类型是有相关性的。比如Greyscale灰度图只能支持1，2，4，8，16位深。

（3）Compression Method压缩算法

后面的Compression Method为数据压缩算法，固定为zlib LZ77算法。该算法通过编码一定范围内的重复数据来压缩整体数据，有兴趣的同学可以了解一下，这里不多做介绍了。找了一张网上的解说图，通过此图可以大致了解此压缩具体在做些什么。



LZ77算法可以设置一个压缩级别参数，参数范围为0 ~ 9，其中0为不压缩；1为最快速度，但压缩率较低；9为压缩率最高，但速度会相对较慢。

（4）Filter Method过滤方法

过滤方法即压缩前的预处理，主要目的是对于一些颜色变化比较“陡”的图片，通过一些数据的变换增加像素数据的重复度，从而增加压缩率。

试想一个场景，一张图片每一个像素点都是前一个像素颜色的递增，那么这张图片每一个像素点都是不同的数值，按照上面的压缩方法，它将无法被压缩。而如果我们对它进行预处理，以第一个像素为基准，后面每一个像素点均变换为当前像素与前一个像素的差值，那么这个变换是可逆的，并且会人为创造出大量的重复数据便于压缩。

具体这些过滤方法为什么可以增加重复数据，由于不涉及此次优化，我也没有做深入了解。

后面可以看到，因为我们业务场景本身的原因，并不需要预处理。

IDAT

IDAT数据块为真正的图片像素数据，这部分数据是经过过滤（Filter）及压缩（Compresson）的，这些方法都有比较成熟的实现，我们也不考虑在这里做任何优化了，因此不多做介绍。

4、优化方案

经过上述内容，针对引言中的问题，我们确定了2个优化方向

1、业务上，无论怎样的单据，都是要仓库打印的，基本都是黑白图片。PNG的颜色类型使用Truecolor是冗余的，根据上图中IHDR文件头表格内容可知，PNG图片是支持灰度（Greyscale）同时位深为1的，即每个像素点由1比特来表示（0代表白点，1代表黑点）。这样可以减少PNG文件的体积，以及压缩生成IDAT块的时间。

2、调整zlib压缩算法的级别为1，牺牲压缩率来提高速度

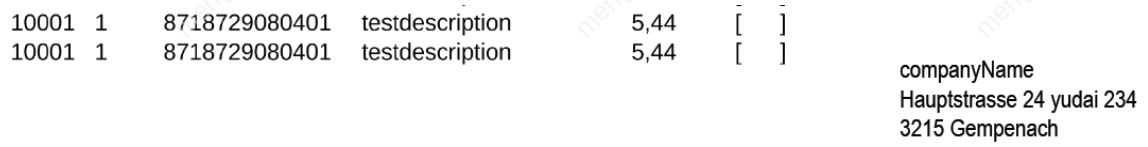
经过查看源码，当BufferedImage的imageType=TYPE_BYTE_BINARY（二进制）时，JDK中的PNG编码器会使用灰度的color type及1位深，而我们发现PDFRender类是有参数可控的，当传入BINARY时，绘制的BufferedImage的类型即为TYPE_BYTE_BINARY。

```Java
BufferedImage image = pdfRenderer.renderImageWithDPI(0, 304,
ImageType.BINARY);
```

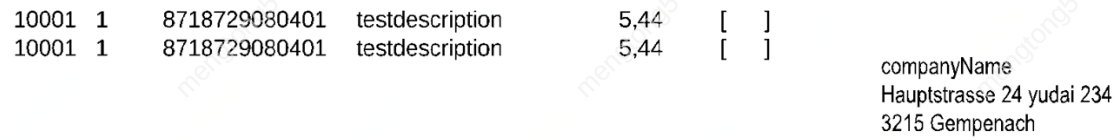
使用此方法后，ImageIO.write编码过程耗时减少到150ms左右。

但是这样改后，我们发现生成的PNG图像，与原PDF文档在观感上相比，有一些发“虚”，如下图

PDF截图



PNG截图



由于TYPE_BYTE_BINARY类型的BufferedImage每个像素只由0，1来表示黑白，很容易想到，这个现象的原因是出在判断“多灰才算黑”上。

我们来看一下源码中，BINARY类型BufferedImage的ColorModel，是如何判断黑白的。

BINARY类型的BufferedImage使用的实现类为IndexColorModel，确定颜色的代码段如下，最终由pix变量决定颜色的索引号。

```Java
int minDist = 256;
int d;
// 计算像素的灰度值
int gray = (int) (red*77 + green*150 + blue*29 + 128)/256;
// 在BINARY类型下，map_size = 2
for (int i = 0; i < map_size; i++) {
    // rgb数组为调色板，每个数组元素表示一个在图片中可能出现的颜色
    // 在BINARY类型下，rgb只有0x00,0xFE两个元素
    if (this.rgb[i] == 0x0) {
        // For allgrayopaque colormaps, entries are 0
        // iff they are an invalid color and should be
        // ignored during color searches.
        continue;
    }
    // 分别计算黑&白与当前灰度值的差值
    d = (this.rgb[i] & 0xff) - gray;
    if (d < 0) d = -d;
    // 选择差值较小的一边
    if (d < minDist) {
        pix = i;
        if (d == 0) {
            break;
        }
        minDist = d;
    }
}
```

由以上代码，在JDK的实现中，通过像素的灰度值更靠近0和255的哪一个，来确定当前像素是黑是白。

这种实现方式对于通用功能来说是合适的，却不适合我们的业务场景，因为我们生成的图片都是单据，大部分需要仓库等场景现场打印，需要优先保证内容的准确性，即不能因为图片上某一处灰得有点“浅”，就不显示它。

对于当前业务场景，我们认为简单地设置一个固定的阈值，来区分灰度值是一个适合的方式。

所以，为解决这个问题，我们设计了2种思路

1、继承实现自己的ColorModel，通过阈值来指定调色板索引号，所有要编码成PNG的BufferedImage都使用自己实现的ColorModel。

2、不使用JDK默认的PNG编码器，使用其他开源实现,在编码阶段通过判断BufferedImage像素灰度值是否超过阈值，来决定编入PNG文件的像素数据是黑是白。

从合理性上看，我认为1方案从程序结构角度是更合理的，但是实际应用中，却选择了方案2，理由如下

（1）BufferedImage通常不是自己生成的，我们往往控制不了其他开源工具操作生成的BufferedImage使用哪种ColorModel，比如我们的项目里PDF Box，IcePdf，Apache poi等开源包都会提供生成BufferedImage的方法，针对每个开源工具都要重新更改源代码，生成使用自己实现的ColorModel的BufferedImage，太过于繁琐了，不具有通用性。

（2）JDK提供的PNG编码器不能设置压缩级别

5、实际优化过程

我们通过网上搜到了开源Java实现的PNG编码器pngencoder作为此业务场景下的编码器。

```
Plain

<groupId>com.pngencoder</groupId>
<artifactId>pngencoder</artifactId>
<version>0.14.0-SNAPSHOT</version>
```

但是我们发现一个问题，开源实现的PNG编码器在编码BufferedImage时，为了方便整字节进行操作，基本都是只能支持8或16比特的位深的PNG，无法支持我们需要的1比特的位深. 经过分析，这一点可以通过自己开发简单的代码实现来补充，因为无论使用几位深，最终PNG编码都是针对像素数据整理过后，对整字节的数据进行后续的过滤及压缩来生成IDAT数据，因此，我们只需要实现对原BufferedImage像素数据的提取并转换为1比特位深度这一步骤。

因此，我们的需求就是，针对一个BufferedImage，每个像素的灰度值通过与阈值比较大小，映射为一个bit数组，并将bit数组转换为byte数组。

下面是我们借助这个开源工具内部实现的部分代码：

```
Java

/**
* 在开源工具原有代码基础上，判断1bit位深时，使用另外的像素数据收集方法
*/
case TYPE_BYTE_GRAY:
    if(bitDepth == 1) {
        // 针对灰度图像，当位深为1的时候走自己实现的数据获取方法
        // RGB图像也可用类似方式
         getByteOneBitGrey(bufferedImage, yStart, width, heightToStream,
consumer);
    } else {
        // 原代码
         getByteGray(bufferedImage, yStart, width, heightToStream,
consumer);
    }
    break;
```

自定义1bit位深取数据方法

Java

```
/**
* 生成使用1bit位深，Greyscale的PNG的像素数据
* 当IHDR中bit Depth为1时，使用这个方法来生成IDAT的原始数据
* @param image 图片BufferedImage
* @param yStart 从图片哪一行开始扫描
* @param width 图像宽度
* @param heightToStream 待处理的高度
* @param consumer 原始数据块后续处理函数
*/
static void getByteOneBitGrey(BufferedImage image, int yStart, int
width, int heightToStream, AbstractPNGLineConsumer consumer)
        throws IOException {
    // 字节数组的长度
    int rowByteSize = (int) Math.ceil(width / 8.0);
    byte[] currLine = new byte[rowByteSize + 1];
    // BufferedImage Raster像素数据
    byte[] rawBytes = ((DataBufferByte)
image.getRaster().getDataBuffer()).getData();
    int currLineIndex, bitIndex;
    byte currValue = 0;
    for(int y = 0 ; y < heightToStream ; y++) {
        int start = (yStart + y) * width;
        currLineIndex = 0;
        bitIndex = 0;
        // 这里有一个坑，PNG数据每行要以一个额外的0x00开头
        currLine[currLineIndex++] = 0;
        for (int i = 0; i < width; i++) {
           // 查到当前像素的灰度值，150为手动设置的阈值，小于150则认为是白色
            byte bitVal = (byte) ((rawBytes[start + i] & 0xFF) < 150 ?
0 : 1);
           // 把每个像素的bit合并到一个byte中
            currValue |= bitVal << (7 - bitIndex++);
            // 当取了8个bit时，将一个完整的byte放入待处理数据
            if (bitIndex == 8) {
                currLine[currLineIndex++] = currValue;
                currValue = 0;
                bitIndex = 0;
            }
        }
       // 如果剩余的bit不够8个，最后一个byte剩余位为0
        if (bitIndex != 0) {
            currLine[currLineIndex++] = currValue;
        }
```

```
            // 调用开源工具的方法对数据做后续处理
            consumer.consume(currLine, null);
        }
    }
```

最终用修改后的开源PNG编码器代替ImageIO.write方法，这里使用压缩级别为1

```
Java

byte[] result = new PngEncoder()
    .withBufferedImage(image)
    .withMultiThreadedCompressionEnabled(false)
    // 配置压缩级别为1
    .withCompressionLevel(2)
    // 设置位深度为1bit
    .withBitDepth(1)
    .toBytes();
```

最终经过优化后测试，和最开始测试时相比，PNG编码步骤上，无论在耗时还是文件大小上都有很大改善

单据类型	PNG编码平均耗时	PNG文件大小 压缩级别:1,9	PNG编码平均耗时 （优化后） 压缩级别:1,9	PNG文件大小 （优化后） 压缩级别:1,9
面单 10*15 203DPI	253ms	52KB	压缩级别1：4ms 压缩级别9：10ms	压缩级别1：10KB 压缩级别9：7KB
退货单 A4 600DPI	3750ms	2732KB	压缩级别1：95ms 压缩级别9： 142ms	压缩级别1： 333KB 压缩级别9： 244KB

5、小结

通过对问题的优化，对以PNG为例的位图文件结构，和Java中对图片的基本操作有了渐进式的理解；同时也意识到，日常工作中，通过对业务本身的理解，清楚知道业务的边界在那里，加上对技术基础知识的深入理解，才能更细致地针对性做出优化。