

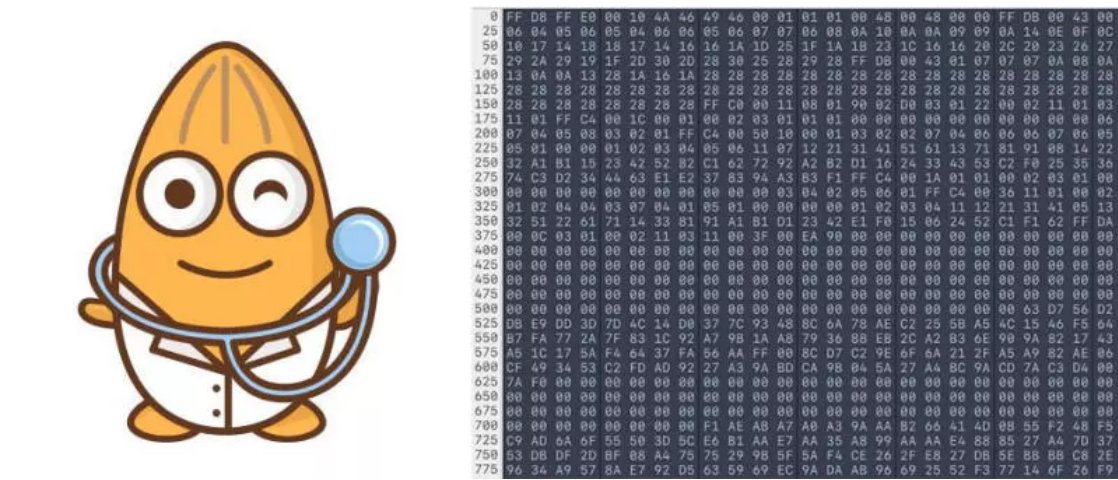
移动开发中我们经常和多媒体数据打交道，对这些数据的解析往往需要耗费大量资源，属于常见的性能瓶颈。

本文针对多媒体数据的一种——图片，介绍下图片的常见格式，它们如何在移动平台上被传输、存储和展示，以及优化图片显示性能的一种方法：强制子线程解码。

图片在计算机世界中怎样被存储和表示？

图片和其他所有资源一样，在内存中本质上都是0和1的二进制数据，计算机需要将这些原始内容渲染成人眼能观察的图片，反过来，也需要将图片以合适的形式保存在存储器或者在网络上传送。

下面是一张图片在硬盘中的原始十六进制表示：



这种将图片以某种规则进行二进制编码的方式，就是图片的格式。

常见的图片格式

图片的格式有很多种，除了我们熟知的 JPG、PNG、GIF，还有Webp，BMP，TIFF，CDR 等等几十种，用于不同的场景或平台。





这些格式可以分为两大类：有损压缩和无损压缩。

有损压缩：相较于颜色，人眼对光线亮度信息更为敏感，基于此，通过合并图片中的颜色信息，保留亮度信息，可以在尽量不影响图片观感的前提下减少存储体积。顾名思义，这样压缩后的图片将会永久损失一些细节。最典型的有损压缩格式是 jpg。









无损压缩：和有损压缩不同，无损压缩不会损失图片细节。它降低图片体积的方式是通过索引，对图片中不同的颜色特征建立索引表，减少了重复的颜色数据，从而达到压缩的效果。常见的无损压缩格式是 png，gif。

除了上述提到的格式，有必要再简单介绍下 **webp** 和 **bitmap**这两种格式：

Webp：jpg 作为主流的网络图片标准可以向上追溯到九十年代初期，已经十分古老

了。所以谷歌公司推出了Webp标准意图替代陈旧的jpg，以加快网络图片的加载速度，提高图片压缩质量。

webp 同时支持有损和无损两种压缩方式，压缩率也很高，无损压缩后的 webp 比 png 少了45%的体积，相同质量的 webp 和 jpg，前者也能节省一半的流量。同时 webp 还支持动图，可谓图片压缩格式的集大成者。

PNG	WebP-lossless	WebP-lossy (with alpha)
"Yellow Rose" 1		
		
PNG file size: 118.5 KB	WebP-lossless file size: 88.1 KB	WebP-lossy (with alpha) file size: 23.4 KB
"baby tux for my user page" 2		
		
PNG file size: 40.5 KB	WebP-lossless file size: 27.5 KB	WebP-lossy (with alpha) file size: 17.3 KB

webp 的缺点是浏览器和移动端支持还不是很完善，我们需要引入谷歌的 libwebp 框架，编解码也会消耗相对更多的资源。

bitmap：bitmap 又叫位图文件，它是一种非压缩的图片格式，所以体积非常大。所谓的非压缩，就是图片每个像素的原始信息在存储器中依次排列，一张典型的 1920*1080像素的 bitmap 图片，每个像素由 RGBA 四个字节表示颜色，那么它的体积就是 $1920 * 1080 * 4 = 1012.5\text{kb}$ 。

由于 bitmap 简单顺序存储图片的像素信息，它可以不经过解码就直接被渲染到 UI 上。实际上，其它格式的图片都需要先被首先解码为 bitmap，然后才能渲染到界面上。

如何判断图片的格式？

在一些场景中，我们需要手动去判断图片数据的格式，以进行不同的处理。一般来说，只要拿到原始的二进制数据，根据不同压缩格式的编码特征，就可以进行简单的分类了。以下是一些图片框架的常用实现，可以复制使用：

```
+ (XImageFormat)imageFormatForImageData:(nullable NSData *)data {
    if (!data) {
        return XImageFormatUndefined;
    }

    uint8_t c;
    [data getBytes:&c length:1];
    switch (c) {
        case 0xFF:
            return XImageFormatJPEG;
        case 0x89:
            return XImageFormatPNG;
        case 0x47:
            return XImageFormatGIF;
        case 0x49:
        case 0x4D:
            return XImageFormatTIFF;
        case 0x52:

            if (data.length < 12) {
                return XImageFormatUndefined;
            }

            NSString *testString = [[NSString alloc] initWithData:
                [data subdataWithRange:NSMakeRange(0, 12)]
                encoding:NSUTF8StringEncoding];
            if ([testString hasPrefix:@"RIFF"] && [testString
                hasSuffix:@"WEBP"]) {
                return XImageFormatWebP;
            }
    }
    return XImageFormatUndefined;
}
```

UIImageView 的性能瓶颈

如上文所说，大部分格式的图片，都需要被首先解码为bitmap，然后才能渲染到UI

上。

UIImageView 显示图片，也有类似的过程。实际上，一张图片从在文件系统中，到被显示到 UIImageView，会经历以下几个步骤：

1. 分配内存缓冲区和其它资源。
2. 从磁盘拷贝数据到内核缓冲区
3. 从内核缓冲区复制数据到用户空间
4. 生成UIImageView，把图像数据赋值给UIImageView
5. 将压缩的图片数据，解码为位图数据（bitmap），如果数据没有字节对齐，Core Animation会再拷贝一份数据，进行字节对齐。
6. CATransaction捕获到UIImageView layer树的变化，主线程RunLoop提交CATransaction，开始进行图像渲染
7. GPU处理位图数据，进行渲染。

由于 UIKit 的封装性，这些细节不会直接对开发者展示。实际上，当我们调用 `[UIImage imageNamed:@"xxx"]` 后，UIImage 中存储的是未解码的图片，而调用 `[UIImageView setImage:image]` 后，会在主线程进行图片的解码工作并且将图片显示到 UI 上，这时候，UIImage 中存储的是解码后的 bitmap 数据。

而图片的解压缩是一个非常消耗 CPU 资源的工作，如果我们有大量的图片需要展示到列表中，将会大大拖慢系统的响应速度，降低运行帧率。这就是 UIImageView 的一个性能瓶颈。

解决性能瓶颈：强制解码

如果 UIImage 中存储的是已经解码后的数据，速度就会快很多，所以优化的思路就是：在子线程中对图片原始数据进行强制解码，再将解码后的图片抛回主线程继续使用，从而提高主线程的响应速度。

我们需要使用的工具是 Core Graphics 框架的 `CGBitmapContextCreate` 方法和相关的绘制函数。总体的步骤是：

A. 创建一个指定大小和格式的 bitmap context。 B. 将未解码图片写入到这个 context 中，这个过程包含了强制解码。 C. 从这个 context 中创建新的 UIImage 对象，返回。

下面是 SDWebImage 实现的核心代码：

```
// ...
```

```

// 1.
CGImageRef imageRef = image.CGImage;

// 2.
CGColorSpaceRef colorspaceRef = [UIImage
colorSpaceForImageRef:imageRef];

size_t width = CGImageGetWidth(imageRef);
size_t height = CGImageGetHeight(imageRef);

// 3.
size_t bytesPerRow = 4 * width;

// 4.
CGContextRef context = CGContextCreate(NULL,
                                         width,
                                         height,
                                         kBitsPerComponent,
                                         bytesPerRow,
                                         colorspaceRef,

kCGBitmapByteOrderDefault|kCGImageAlphaNoneSkipLast);
if (context == NULL) {
    return image;
}

// 5.
CGContextDrawImage(context, CGRectMake(0, 0, width, height),
imageRef);

// 6.
CGImageRef newImageRef = CGContextCreateImage(context);

// 7.
UIImage *newImage = [UIImage imageWithCGImage:newImageRef
                                         scale:image.scale
orientation:image.imageOrientation];

CGContextRelease(context);
CGImageRelease(newImageRef);

return newImage;

```

对上述代码的解析：

1、从 UIImage 对象中获取 CGImageRef 的引用。这两个结构是苹果在不同层级上

对图片的表示方式，UIImage 属于 UIKit，是 UI 层级图片的抽象，用于图片的展示；CGImageRef 是 QuartzCore 中的一个结构体指针，用C语言编写，用来创建像素位图，可以通过操作存储的像素位来编辑图片。这两种结构可以方便的互转：

```
// CGImageRef 转换成 UIImage
CGImageRef imageRef = CGBitmapContextCreateImage(context);
UIImage *image = [UIImage imageWithCGImage:imageRef];

// UIImage 转换成 CGImageRef
UIImage *image=[UIImage imageNamed:@"xxx"];
CGImageRef imageRef=loadImage.CGImage;
```

2、调用 UIImage 的 `+colorSpaceForImageRef:` 方法来获取原始图片的颜色空间参数。

什么叫颜色空间呢，就是对相同颜色数值的解释方式，比如说一个像素的数据是 (FF0000FF)，在 RGBA 颜色空间中，会被解释为红色，而在 BGRA 颜色空间中，则会被解释为蓝色。所以我们需要提取出这个参数，保证解码前后的图片颜色空间一致。



在不同的颜色空间下的同一张图片

CoreGraphic中支持的颜色空间类型：

Values	Color space	Components
240 degrees, 100%, 100%	HSB	Hue, saturation, brightness

0, 0, 1	RGB	Red, green, blue
1, 1, 0, 0	CMYK	Cyan, magenta, yellow, black
1, 0, 0	BGR	Blue, green, red

3、计算图片解码后每行需要的比特数，由两个参数相乘得到：每行的像素数 `width`，和存储一个像素需要的比特数4。

这里的4，其实是由每张图片的 像素格式 和 像素组合 来决定的，下表是苹果平台支持的像素组合方式。

CS	Pixel format and bitmap information constant	Availability
Null	8 bpp, 8 bpc, <code>kCGImageAlphaOnly</code>	Mac OS X, iOS
Gray	8 bpp, 8 bpc, <code>kCGImageAlphaNone</code>	Mac OS X, iOS
Gray	8 bpp, 8 bpc, <code>kCGImageAlphaOnly</code>	Mac OS X, iOS
Gray	16 bpp, 16 bpc, <code>kCGImageAlphaNone</code>	Mac OS X
Gray	32 bpp, 32 bpc, <code>kCGImageAlphaNone kCGBitmapFloatComponents</code>	Mac OS X
RGB	16 bpp, 5 bpc, <code>kCGImageAlphaNoneSkipFirst</code>	Mac OS X, iOS
RGB	32 bpp, 8 bpc, <code>kCGImageAlphaNoneSkipFirst</code>	Mac OS X, iOS
RGB	32 bpp, 8 bpc, <code>kCGImageAlphaNoneSkipLast</code>	Mac OS X, iOS
RGB	32 bpp, 8 bpc, <code>kCGImageAlphaPremultipliedFirst</code>	Mac OS X, iOS
RGB	32 bpp, 8 bpc, <code>kCGImageAlphaPremultipliedLast</code>	Mac OS X, iOS
RGB	64 bpp, 16 bpc, <code>kCGImageAlphaPremultipliedLast</code>	Mac OS X
RGB	64 bpp, 16 bpc, <code>kCGImageAlphaNoneSkipLast</code>	Mac OS X
RGB	128 bpp, 32 bpc, <code>kCGImageAlphaNoneSkipLast kCGBitmapFloatComponents</code>	Mac OS X
RGB	128 bpp, 32 bpc, <code>kCGImageAlphaPremultipliedLast kCGBitmapFloatComponents</code>	Mac OS X
CMYK	32 bpp, 8 bpc, <code>kCGImageAlphaNone</code>	Mac OS X
CMYK	64 bpp, 16 bpc, <code>kCGImageAlphaNone</code>	Mac OS X
CMYK	128 bpp, 32 bpc, <code>kCGImageAlphaNone kCGBitmapFloatComponents</code>	Mac OS X

表中的bpp，表示每个像素需要多少位；bpc表示颜色的每个分量，需要多少位。具体的解释方式，可以看下面这张图：



32 bits per pixel RGBA, kCGImageAlphaLast



32 bits per pixel ARGB, kCGImageAlphaFirst



32 bits per pixel RGB, kCGImageAlphaNoneSkipLast



32 bits per pixel RGB, kCGImageAlphaNoneSkipFirst



16 bits per pixel RGB, kCGImageAlphaNoneSkipFirst



我们解码后的图片，默认采用 kCGImageAlphaNoneSkipLast RGB 的像素组合，没有 alpha 通道，每个像素32位4个字节，前三个字节代表红绿蓝三个通道，最后一个字节废弃不被解释。

4、最关键的函数：调用 `CGBitmapContextCreate()` 方法，生成一个空白的图片绘制上下文，我们传入了上述的一些参数，指定了图片的大小、颜色空间、像素排列等等属性。

5、调用 `CGContextDrawImage()` 方法，将未解码的 `imageRef` 指针内容，写入到我们创建的上下文中，这个步骤，完成了隐式的解码工作。

6、从 context 上下文中创建一个新的 `imageRef`，这是解码后的图片了。

7、从 `imageRef` 生成供UI层使用的 `UIImage` 对象，同时指定图片的 `scale` 和 `orientation` 两个参数。

`scale` 指的是图片被渲染时需要被压缩的倍数，为什么会存在这个参数呢，因为苹果为了节省安装包体积，允许开发者为同一张图片上传不同分辨率的版本，也就是我们熟悉的@2x，@3x后缀图片。不同屏幕素质的设备，会获取到对应的资源。为了绘制图片时统一，这些图片会被set自己的scale属性，比如@2x图片，`scale` 值就是2，虽然和1x图片的绘制宽高一样，但是实际的长是 `width * scale`。

`orientation` 很好理解，就是图片的旋转属性，告诉设备，以哪个方向作为图片的默认方向来渲染。

通过以上的步骤，我们成功在子线程中对图片进行了强制转码，回调给主线程使用，从而大大提高了图片的渲染效率。这也是现在主流 App 和大量三方库的最佳实践。

总结

- 图片在计算机世界中按照不同的封装格式进行压缩，以便存储和传输。
- 手机会在主线程中将压缩的图片解压为可以进行渲染的位图格式，这个过程会消耗大量资源，影响App性能。
- 我们使用 Core Graphics 的绘制方法，强制在子线程中先对 UIImage 进行转码工作，减少主线程的负担，从而提升App的响应速度。

和 UIImageView 类似，UIKit 隐藏了很多技术细节，降低开发者的学习门槛，但另一方面，却也限制了对一些底层技术的探究。文中提到的强制解码方法，其实也是 `CGBitmapContextCreate` 方法的一个『副作用』，属于比较hack方式，这也是iOS平台的一个局限：苹果过于封闭了。

用户对软件性能（帧率、响应速度、闪退率等等）其实非常敏感，作为开发者，必须不断探究性能瓶颈背后的原理，并且尝试解决，移动端开发的性能优化永无止境