谈谈 iOS 中图片的解压缩 - 雷纯锋的技术博客

对于大多数 iOS 应用来说，图片往往是最占用手机内存的资源之一，同时也是不可或缺的组成部分。将一张图片从磁盘中加载出来，并最终显示到屏幕上，中间其实经过了一系列复杂的处理过程，其中就包括了对图片的解压缩。

图片加载的工作流

概括来说，从磁盘中加载一张图片，并将它显示到屏幕上，中间的 主要工作流 如下：

- 假设我们使用 `+imageWithContentsOfFile:` 方法从磁盘中加载一张图片，这个时候的图片并没有解压缩；
- 然后将生成的 `UIImage` 赋值给 `UIImageView` ；
- 接着一个隐式的 `CATransaction` 捕获到了 `UIImageView` 图层树的变化；
- 在主线程的下一个 run loop 到来时，Core Animation 提交了这个隐式的 transaction ，这个过程可能会对图片进行 copy 操作，而受图片是否**字节对齐**等因素的影响，这个 copy 操作可能会涉及以下部分或全部步骤：
    - 分配内存缓冲区用于管理文件 IO 和解压缩操作；
    - 将文件数据从磁盘读到内存中；
    - 将压缩的图片数据解码成未压缩的位图形式，这是一个非常耗时的 CPU 操作；
    - 最后 Core Animation 使用未压缩的位图数据渲染 `UIImageView` 的图层。

在上面的步骤中，我们提到了图片的解压缩是一个非常耗时的 CPU 操作，并且它默认是在主线程中执行的。那么当需要加载的图片比较多时，就会对我们应用的响应性造成严重的影响，尤其是在快速滑动的列表上，这个问题会表现得更加突出。

为什么需要解压缩

既然图片的解压缩需要消耗大量的 CPU 时间，那么我们为什么还要对图片进行解压缩呢？是否可以不经过解压缩，而直接将图片显示到屏幕上呢？答案是否定的。要想弄明白这个问题，我们首先需要知道什么是 位图 ：

> A bitmap image (or sampled image) is an array of pixels (or samples). Each pixel represents a single point in the image. JPEG, TIFF, and PNG graphics files are examples of bitmap images.

其实，位图就是一个像素数组，数组中的每个像素就代表着图片中的一个点。我们在应用中经常用到的 JPEG 和 PNG 图片就是位图。下面，我们来看一个具体的例子，这是一张 PNG 图片，像素为 30×30 ，文件大小为 843B：

我们使用 下面的代码：

```
1        UIImage *image = [UIImage imageNamed:@"check_green"];
2        CFDataRef rawData = CGDataProviderCopyData(CGImageGetDataProvider(image.CGImage)
```

就可以获取到这个图片的原始像素数据，大小为 3600B：

```
1        00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
2        01020102 032c023c 0567048c 078d06bf 08a006d9 09b307f3 09b307f3 08a006d9 078d06b
3        0567048c 032c023c 01020102 00000000 00000000 00000000 00000000 00000000 0000000
4        00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
5        00000000 01060108 05570476 09ab07e9 09bb07ff 09bb07ff 09bb07ff 09bb07ff 09bb07f
6        09bb07ff 09bb07ff 09bb07ff 09bb07ff 09bb07ff 09ab07e9 05570476 01060108 0000000
7        00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
8        00000000 00000000 00000000 033d0353 08a607e2 09bb07ff 09bb07ff 09bb07ff 09bb07f
9        ...
10       09bb07ff 09bb07ff 09bb07ff 09bb07ff 08a607e2 033d0353 00000000 00000000 0000000
11       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
12       00000000 01060108 05570476 09ab07e9 09bb07ff 09bb07ff 09bb07ff 09bb07ff 09bb07f
13       09bb07ff 09bb07ff 09bb07ff 09bb07ff 09bb07ff 09ab07e9 05570476 01060108 0000000
14       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
15       00000000 00000000 00000000 00000000 00000000 00000000 01020102 032c023c 0567048
16       078d06bf 08a006d9 09b307f3 09b307f3 08a006d9 078d06bf 0567048c 032c023c 0102010
17       00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0000000
```

也就是说，这张文件大小为 843B 的 PNG 图片解压缩后的大小是 3600B，是原始文件大小的 4.27 倍。那么这个 3600B 是怎么得来的呢？与图片的文件大小或者像素有什么必然的联系吗？事实上，解压缩后的图片大小与原始文件大小之间没有任何关系，而只与图片的像素有关：

```
1        解压缩后的图片大小 = 图片的像素宽 30 * 图片的像素高 30 * 每个像素所占的字节数 4
```

```
1          解压缩后的图片大小    图片的像素宽 30    图片的像素高 30    每个像素所占的字节数 4
```

至于这个公式是怎么得来的，我们后面会有详细的说明，现在只需要知道即可。

至此，我们已经知道了什么是位图，并且直观地看到了它的原始像素数据，那么它与我们经常提到的图片的二进制数据有什么联系吗？是同一个东西吗？事实上，这二者是完全独立的两个东西，它们之间没有必然的联系。为了加深理解，我把这个图片拖进 Sublime Text 2 中，得到了这个图片的二进制数据，大小与原始文件大小一致，为 843B：

```
1          8950 4e47 0d0a 1a0a 0000 000d 4948 4452 0000 001e 0000 001e 0806 0000 003b 30ae
2          0000 0173 5247 4200 aece 1ce9 0000 0305 4944 4154 480d c557 4d68 1341 149e 3709
3          09c6 8a56 2385 9e14 f458 4fa2 d092 f4a6 28d8 2222 de04 3d09 a1d0 7a50 0954 8bad
4          4fde 3c89 482b 2ad6 8334 d183 e049 ef9e 4a41 48b0 42eb a549 6893 1ddf 9bcd b4d9
5          4dd8 a43a b0d9 9d79 3fdf bc79 3ff3 02ac 8591 1559 3e97 9b3e 5b05 fb32 6330 c098
6          183d 340a b886 8ff8 1e15 fced 587a e26b 16b2 b643 f2ff 057f 1263 fd9f fbbb 7ed7
7          1142 8c09 268e 04f1 2a1a 3058 0380 b9c3 91de a7ab 43ab 15b5 aebf 7d81 ad65 eb0a
8          8f4f 9f2e d4da 1c7e e249 64ca c3e5 d726 7eae 2fa2 7510 cb75 3d62 cc5e 0c0f 4a5a
9          ...
10         36ac b11e 7006 f71b 5386 a2b7 1e48 ad82 a26a 2880 95db 3f8b f525 b880 e0ed 7221
11         fa02 2cd4 1af7 1d0e 546a 98e5 d4ae 342a 337e 6b96 134f 1ba0 0c0b c83b a0f2 3593
12         6ca9 b541 cb4f 254e df58 d958 8955 a0fc 2638 658c 2660 f986 b5f1 f4dd 63f2 5aec
13         e3b6 b0a7 cdac ee55 145c c7dc 8f60 f53f e0a6 b436 e3c0 27b0 8ecf 5054 336a ccd0
14         2335 1f78 323d 6141 09c3 c1aa 5f8b 4e37 0899 e6b0 ed72 4046 759e d262 5247 9d01
15         a976 55fb c993 6ed5 7d10 8ff4 b162 fe6f cd1e ee4a d4bb c18e 594e 96ea 1da6 c762
16         bdff 7943 afc0 c91f bdd1 a327 28fc 29f7 d47a b337 f192 0cc9 36fa 5497 73f9 5827
17         1599 4eff 69fb 0b0d 1f7a 96cd 3eb0 7800 0000 0049 454e 44ae 4260 82
```

事实上，不管是 JPEG 还是 PNG 图片，都是一种压缩的位图图形格式。只不过 PNG 图片是无损压缩，并且支持 alpha 通道，而 JPEG 图片则是有损压缩，可以指定 0–100% 的压缩比。值得一提的是，在苹果的 SDK 中专门提供了两个函数用来生成 PNG 和 JPEG 图片：

```
1          // return image as PNG. May return nil if image has no CGImageRef or invalid bit
2          UIKIT_EXTERN NSData * __nullable UIImagePNGRepresentation(UIImage * __nonnull im
3
4          // return image as JPEG. May return nil if image has no CGImageRef or invalid bi
5          UIKIT_EXTERN NSData * __nullable UIImageJPEGRepresentation(UIImage * __nonnull i
```

因此，在将磁盘中的图片渲染到屏幕之前，必须先要得到图片的原始像素数据，才能执行后续

的绘制操作，这就是为什么需要对图片解压缩的原因。

强制解压缩的原理

既然图片的解压缩不可避免，而我们也不想让它在主线程执行，影响我们应用的响应性，那么是否有比较好的解决方案呢？答案是肯定的。

我们前面已经提到了，当未解压缩的图片将要渲染到屏幕时，系统会在主线程对图片进行解压缩，而如果图片已经解压缩了，系统就不会再对图片进行解压缩。因此，也就有了业内的解决方案，在子线程提前对图片进行强制解压缩。

而强制解压缩的原理就是对图片进行重新绘制，得到一张新的解压缩后的位图。其中，用到的最核心的函数是 `CGBitmapContextCreate` ：

```
/* Create a bitmap context. The context draws into a bitmap which is `width'
   pixels wide and `height' pixels high. The number of components for each
   pixel is specified by `space', which may also specify a destination color
   profile. The number of bits for each component of a pixel is specified by
   `bitsPerComponent'. The number of bytes per pixel is equal to
   `(bitsPerComponent * number of components + 7)/8'. Each row of the bitmap
   consists of `bytesPerRow' bytes, which must be at least `width * bytes
   per pixel' bytes; in addition, `bytesPerRow' must be an integer multiple
   of the number of bytes per pixel. `data', if non-NULL, points to a block
   of memory at least `bytesPerRow * height' bytes. If `data' is NULL, the
   data for context is allocated automatically and freed when the context is
   deallocated. `bitmapInfo' specifies whether the bitmap should contain an
   alpha channel and how it's to be generated, along with whether the
   components are floating-point or integer. */
CG_EXTERN CGContextRef __nullable CGBitmapContextCreate(void * __nullable data,
    size_t width, size_t height, size_t bitsPerComponent, size_t bytesPerRow,
    CGColorSpaceRef cg_nullable space, uint32_t bitmapInfo)
    CG_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);
```

顾名思义，这个函数用于创建一个位图上下文，用来绘制一张宽 `width` 像素，高 `height` 像素的位图。这个函数的注释比较长，参数也比较难理解，但是先别着急，我们先来了解下相关的知识，然后再回过头来理解这些参数，就会比较简单了。

Pixel Format

我们前面已经提到了，位图其实就是一个像素数组，而 像素格式 则是用来描述每个像素的

组成格式，它包括以下信息：

- Bits per component ： 一个像素中每个独立的颜色分量使用的 bit 数；
- Bits per pixel ： 一个像素使用的总 bit 数；
- Bytes per row ： 位图中的每一行使用的字节数。

有一点需要注意的是，对于位图来说，像素格式并不是随意组合的，目前只支持以下有限的 17 种特定组合 ：

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

从上图可知，对于 iOS 来说，只支持 8 种像素格式。其中颜色空间为 Null 的 1 种，Gray 的 2 种，RGB 的 5 种，CMYK 的 0 种。换句话说，iOS 并不支持 CMYK 的颜色空间。另外，在表格的第 2 列中，除了像素格式外，还指定了 bitmap information constant ， 我们在后面会详细介绍。

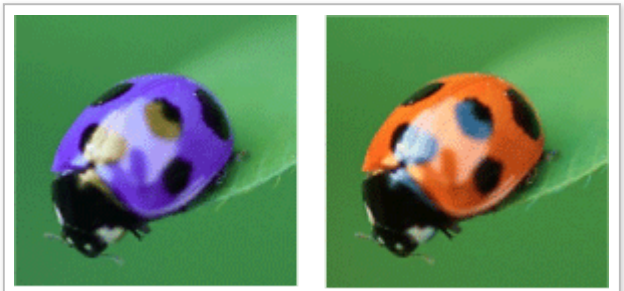
Color and Color Spaces

在上面我们提到了 颜色空间 ， 那么什么是颜色空间呢？它跟颜色有什么关系呢？在 Quartz

中，一个颜色是由一组值来表示的，比如 0, 0, 1 。而颜色空间则是用来说明如何解析这些值的，离开了颜色空间，它们将变得毫无意义。比如，下面的值都表示蓝色：

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

如果不知道颜色空间，那么我们根本无法知道这些值所代表的颜色。比如 0, 0, 1 在 RGB 下代表蓝色，而在 BGR 下则代表的是红色。在 RGB 和 BGR 两种颜色空间下，绿色是相同的，而红色和蓝色则相互对调了。因此，对于同一张图片，使用 RGB 和 BGR 两种颜色空间可能会得到两种不一样的效果：



是不是感觉非常有意思呢?

Color Spaces and Bitmap Layout

我们前面已经知道了，像素格式是用来描述每个像素的组成格式的，比如每个像素使用的总 bit 数。而要想确保 Quartz 能够正确地解析这些 bit 所代表的含义，我们还需要提供 位图的布局信息 `CGBitmapInfo` ：

```
typedef CF_OPTIONS(uint32_t, CGBitmapInfo) {
    kCGBitmapAlphaInfoMask = 0x1F,

    kCGBitmapFloatInfoMask = 0xF00,
```

```
    kCGBitmapFloatComponents = (1 << 8),

    kCGBitmapByteOrderMask     = kCGImageByteOrderMask,
    kCGBitmapByteOrderDefault  = (0 << 12),
    kCGBitmapByteOrder16Little = kCGImageByteOrder16Little,
    kCGBitmapByteOrder32Little = kCGImageByteOrder32Little,
    kCGBitmapByteOrder16Big    = kCGImageByteOrder16Big,
    kCGBitmapByteOrder32Big    = kCGImageByteOrder32Big
} CG_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_2_0);
```

它主要提供了三个方面的布局信息：

- alpha 的信息；
- 颜色分量是否为浮点数；
- 像素格式的字节顺序。

其中，alpha 的信息由枚举值 `CGImageAlphaInfo` 来表示：

```
typedef CF_ENUM(uint32_t, CGImageAlphaInfo) {
    kCGImageAlphaNone,               /* For example, RGB. */
    kCGImageAlphaPremultipliedLast,  /* For example, premultiplied RGBA */
    kCGImageAlphaPremultipliedFirst, /* For example, premultiplied ARGB */
    kCGImageAlphaLast,               /* For example, non-premultiplied RGBA */
    kCGImageAlphaFirst,              /* For example, non-premultiplied ARGB */
    kCGImageAlphaNoneSkipLast,       /* For example, RBGX. */
    kCGImageAlphaNoneSkipFirst,      /* For example, XRGB. */
    kCGImageAlphaOnly                /* No color data, alpha data only */
};
```

上面的注释其实已经比较清楚了，它同样也提供了三个方面的 alpha 信息：

- 是否包含 alpha ；
- 如果包含 alpha ，那么 alpha 信息所处的位置，在像素的 最低有效位 ，比如 RGBA ，还是 最高有效位 ，比如 ARGB ；
- 如果包含 alpha ，那么每个颜色分量是否已经乘以 alpha 的值，这种做法可以加速图片的渲染时间，因为它避免了渲染时的额外乘法运算。比如，对于 RGB 颜色空间，用已经乘以 alpha 的数据来渲染图片，每个像素都可以避免 3 次乘法运算，红色乘以 alpha ，绿色乘以 alpha 和蓝色乘以 alpha 。

那么我们在解压缩图片的时候应该使用哪个值呢？根据 Which CGImageAlphaInfo

should we use 和官方文档中对 `UIGraphicsBeginImageContextWithOptions` 函数的讨论：

> You use this function to configure the drawing environment for rendering into a bitmap. The format for the bitmap is a ARGB 32–bit integer pixel format using host–byte order. If the opaque parameter is YES, the alpha channel is ignored and the bitmap is treated as fully opaque (kCGImageAlphaNoneSkipFirst | kCGBitmapByteOrder32Host). Otherwise, each pixel uses a premultipled ARGB format (kCGImageAlphaPremultipliedFirst | kCGBitmapByteOrder32Host).

我们可以知道，当图片不包含 alpha 的时候使用 `kCGImageAlphaNoneSkipFirst` ，否则使用 `kCGImageAlphaPremultipliedFirst` 。另外，这里也提到了字节顺序应该使用 32 位的主机字节顺序 `kCGBitmapByteOrder32Host` ，而这个值具体是什么，我们后面再讨论。

至于颜色分量是否为浮点数，这个就比较简单了，直接逻辑或 `kCGBitmapFloatComponents` 就可以了。更详细的内容就不展开了，因为我们一般用不上这个值。

接下来，我们来简单地了解下像素格式的 字节顺序 ，它是由枚举值 `CGImageByteOrderInfo` 来表示的：

```
typedef CF_ENUM(uint32_t, CGImageByteOrderInfo) {
    kCGImageByteOrderMask     = 0x7000,
    kCGImageByteOrder16Little = (1 << 12),
    kCGImageByteOrder32Little = (2 << 12),
    kCGImageByteOrder16Big    = (3 << 12),
    kCGImageByteOrder32Big    = (4 << 12)
} CG_AVAILABLE_STARTING(__MAC_10_12, __IPHONE_10_0);
```

它主要提供了两个方面的字节顺序信息：

- 小端模式 还是 大端模式 ；
- 数据以 16 位还是 32 位为单位。

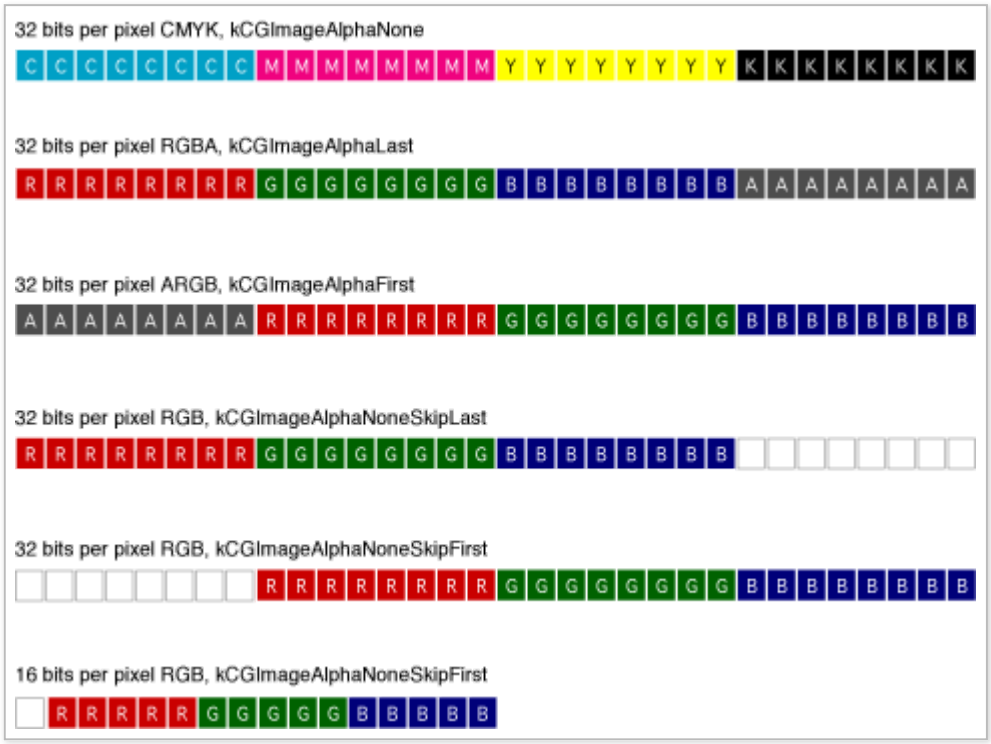
对于 iPhone 来说，采用的是小端模式，但是为了保证应用的向后兼容性，我们可以使用系统提供的宏，来避免 Hardcoding ：

```
#ifdef __BIG_ENDIAN
```

```
#if __BIG_ENDIAN__
    #define kCGBitmapByteOrder16Host kCGBitmapByteOrder16Big
    #define kCGBitmapByteOrder32Host kCGBitmapByteOrder32Big
#else /* Little endian. */
    #define kCGBitmapByteOrder16Host kCGBitmapByteOrder16Little
    #define kCGBitmapByteOrder32Host kCGBitmapByteOrder32Little
#endif
```

根据前面的讨论，我们知道字节顺序的值应该使用的是 32 位的主机字节顺序 `kCGBitmapByteOrder32Host` ，这样的话不管当前设备采用的是小端模式还是大端模式，字节顺序始终与其保持一致。

下面，我们来看一张图，它非常形象地展示了在使用 16 或 32 位像素格式的 CMYK 和 RGB 颜色空间下，一个像素是如何被表示的：



我们从图中可以看出，在 32 位像素格式下，每个颜色分量使用 8 位；而在 16 位像素格式下，每个颜色分量则使用 5 位。

好了，了解完这些相关知识后，我们再回过头来看看 `CGBitmapContextCreate` 函数中每个参数所代表的具体含义：

- `data` ：如果不为 `NULL` ，那么它应该指向一块大小至少为 `bytesPerRow * height` 字节的内存；如果 为 `NULL` ，那么系统就会为我们自动分配和释放所需的内存，所以一般指定 `NULL` 即可；

- `width` 和 `height` ：位图的宽度和高度，分别赋值为图片的像素宽度和像素高度即可；
- `bitsPerComponent` ：像素的每个颜色分量使用的 bit 数，在 RGB 颜色空间下指定 8 即可；
- `bytesPerRow` ：位图的每一行使用的字节数，大小至少为 `width * bytes per pixel` 字节。有意思的是，当我们指定 0 时，系统不仅会为我们自动计算，而且还会进行 cache line alignment 的优化，更多信息可以查看 what is byte alignment (cache line alignment) for Core Animation? Why it matters? 和 Why is my image's Bytes per Row more than its Bytes per Pixel times its Width? ，亲测可用；
- `space` ：就是我们前面提到的颜色空间，一般使用 RGB 即可；
- `bitmapInfo` ：就是我们前面提到的位图的布局信息。

到这里，你已经掌握了强制解压缩图片需要用到的最核心的函数，点个赞。

开源库的实现

接下来，我们来看看在三个比较流行的开源库 YYKit 、SDWebImage 和 FLAnimatedImage 中，对图片的强制解压缩是如何实现的。

首先，我们来看看 YYKit 中的相关代码，用于解压缩图片的函数 `YYCGImageCreateDecodedCopy` 存在于 YYImageCoder 类中，核心代码如下：

```
CGImageRef YYCGImageCreateDecodedCopy(CGImageRef imageRef, BOOL decodeForDispla
    ...

    if (decodeForDisplay) { // decode with redraw (may lose some precision)
        CGImageAlphaInfo alphaInfo = CGImageGetAlphaInfo(imageRef) & kCGBitmapA

        BOOL hasAlpha = NO;
        if (alphaInfo == kCGImageAlphaPremultipliedLast ||
            alphaInfo == kCGImageAlphaPremultipliedFirst ||
            alphaInfo == kCGImageAlphaLast ||
            alphaInfo == kCGImageAlphaFirst) {
            hasAlpha = YES;
        }

        // BGRA8888 (premultiplied) or BGRX8888
        // same as UIGraphicsBeginImageContext() and -[UIView drawRect:]
        CGBitmapInfo bitmapInfo = kCGBitmapByteOrder32Host;
        bitmapInfo |= hasAlpha ? kCGImageAlphaPremultipliedFirst : kCGImageAlph

```

```
            CGContextRef context = CGBitmapContextCreate(NULL, width, height, 8, 0,
            if (!context) return NULL;

            CGContextDrawImage(context, CGRectMake(0, 0, width, height), imageRef);
            CGImageRef newImage = CGBitmapContextCreateImage(context);
            CFRelease(context);

            return newImage;
        } else {
            ...
        }
    }
```

它接受一个原始的位图参数 `imageRef` ，最终返回一个新的解压缩后的位图 `newImage` ，中间主要经过了以下三个步骤：

- 使用 `CGBitmapContextCreate` 函数创建一个位图上下文；
- 使用 `CGContextDrawImage` 函数将原始位图绘制到上下文中；
- 使用 `CGBitmapContextCreateImage` 函数创建一张新的解压缩后的位图。

事实上，SDWebImage 和 FLAnimatedImage 中对图片的解压缩过程与上述完全一致，只是传递给 `CGBitmapContextCreate` 函数的部分参数存在细微的差别，如下表所示：

参数		YYKit	SDWebImage	FLAnimatedImage
data		NULL	NULL	NULL
width		width	width	width
height		height	height	height
bitsPerComponent		8	8	8
bytesPerRow		0	4 * width	4 * width
space		RGB	RGB	RGB
bitmapInfo	CGImageAlphaInfo	kCGImageAlphaPremultipliedFirst kCGImageAlphaNoneSkipFirst	kCGImageAlphaNoneSkipLast	kCGImageAlphaPremultipliedLast kCGImageAlphaPremultipliedFirst kCGImageAlphaNoneSkipLast kCGImageAlphaNoneSkipFirst
	CGImageByteOrderInfo	kCGBitmapByteOrder32Host	kCGBitmapByteOrderDefault	kCGBitmapByteOrderDefault

在上表中，用浅绿色背景标记的参数即为我们在前面的分析中所推荐的参数，用这些参数解压缩后的图片渲染的速度会更快。因此，从理论上说 YYKit 中的解压缩算法是三者之中最优的。

性能对比

口说无凭，因此我编写了一个小的测试程序，来简单地对比一下这三个开源库的解压缩性能，源码可以在 GitHub 上找到。

采用的测试样例分别为 5 张 PNG 图片和 5 张 JPEG 图片，像素依次为 128x96 、256x192 、

512x384 、1024x768 和 2048x1536 ， 它们其实都长一个样：

首先，我们来了解下测试的原理，我们可以将从磁盘加载一张图片到最终渲染到屏幕上的过程划分为三个阶段：

- 初始化阶段：从磁盘初始化图片，生成一个未解压缩的 `UIImage` 对象；
- 解压缩阶段：分别使用 YYKit 、SDWebImage 和 FLAnimatedImage 对第 1 步中得到的 `UIImage` 对象进行解压缩，得到一个新的解压缩后的 `UIImage` 对象；
- 绘制阶段：将第 2 步中得到的 `UIImage` 对象绘制到屏幕上。

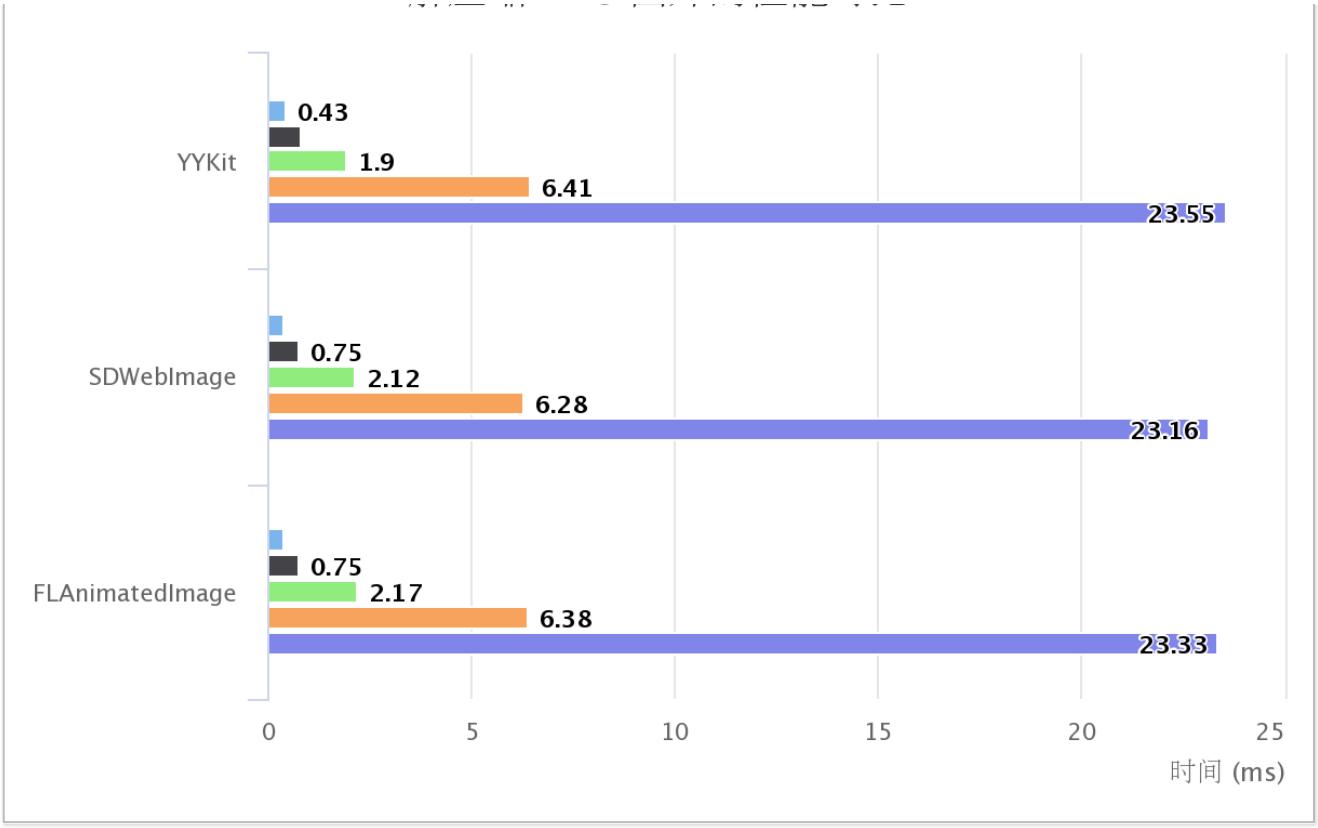
这里我们以绘制阶段的耗时为依据来评测解压缩的性能，解压缩的算法越优秀，那么得到的图片就越符合系统渲染时的需求，绘制的时间也就越短。为了让测试的结果更准确，我们对每张图片都解压缩 10 次，然后取平均值。说明，本次使用的测试设备是 iPhone 5s 。

首先，我们来看看解压缩 PNG 图片的测试结果：

图片	YYKit	SDWebImage	FLAnimatedImage
128x96.png	0.43	0.38	0.37
256x192.png	0.82	0.75	0.75
512x384.png	1.90	2.12	2.17
1024x768.png	6.41	6.28	6.38
2048x1536.png	23.55	23.16	23.33

相应的柱状图如下：

解压缩 PNG 图片的性能对比



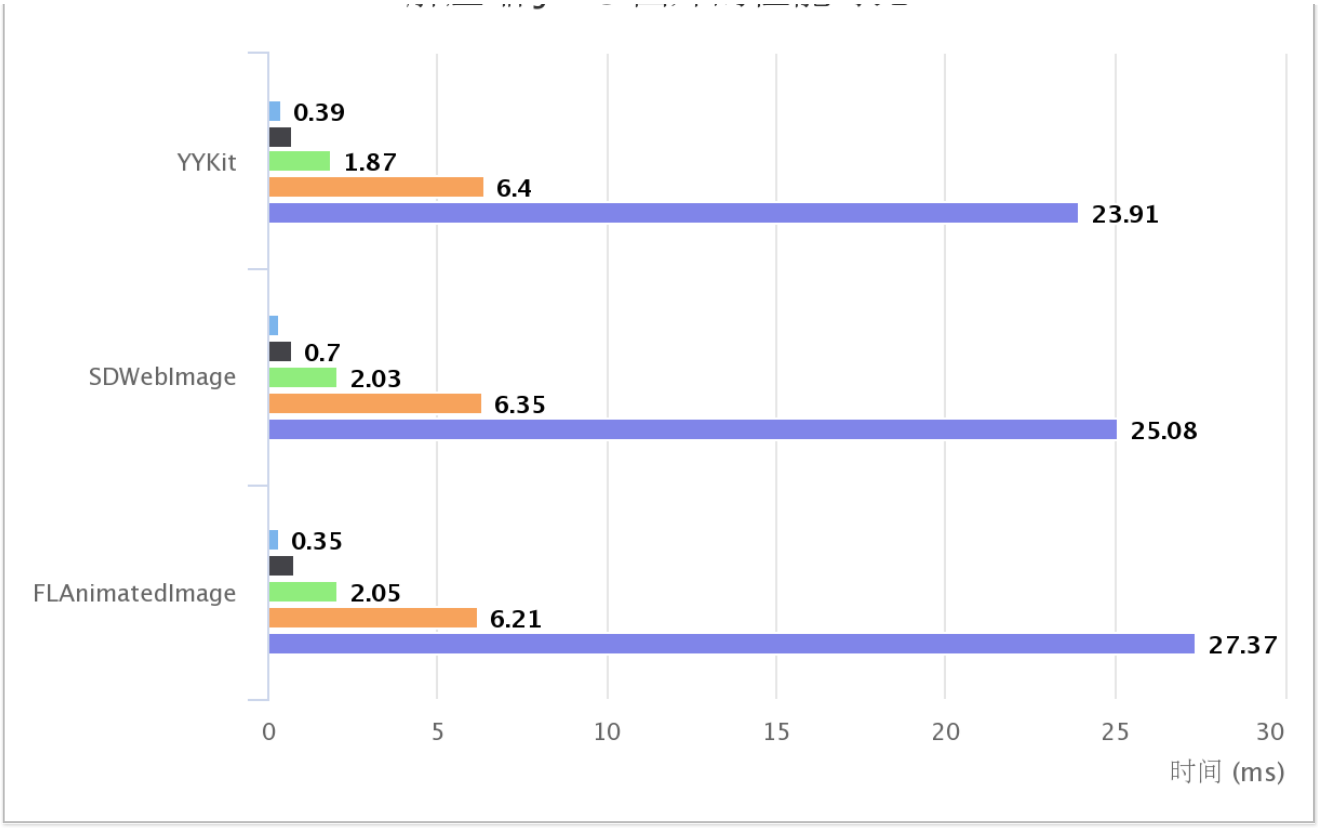
从上图可以看出，就我们采用的测试样例来说，解压缩 PNG 图片的性能 SDWebImage 最好，FLAnimatedImage 次之，YYKit 最差。这与我们前面的理论结果有一定的差距，可能是测试样例太少，也可能这就是真实结果。另外，需要说明的是，我们这里使用的 PNG 图片都是不带 alpha 值，因为 SDWebImage 不支持解压缩带 alpha 值的 PNG 图片。

接着，我们再来看看解压缩 JPEG 图片的测试结果：

图片	YYKit	SDWebImage	FLAnimatedImage
128x96.jpg	0.39	0.34	0.35
256x192.jpg	0.68	0.7	0.74
512x384.jpg	1.87	2.03	2.05
1024x768.jpg	6.4	6.35	6.21
2048x1536.jpg	23.91	25.08	27.37

相应的柱状图如下：

解压缩 JPEG 图片的性能对比



这次 YYKit 终于翻盘了，解压缩 JPEG 图片的性能最好，SDWebImage 和 FLAnimatedImage 并列第二。

总结

其实，要理解 iOS 中图片的解压缩并不难，重点是要理解位图的概念。而图片解压缩的过程其实就是将图片的二进制数据转换成像素数据的过程。了解这些知识，将有助于我们更好地处理图片，管理好它们所占用的内存。

参考链接

https://www.cocoanetics.com/2011/10/avoiding-image-decompression-sickness/

https://developer.apple.com/library/content/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html

https://github.com/path/FastImageCache

http://stackoverflow.com/questions/23790837/what-is-byte-

alignment-cache-line-alignment-for-core-animation-why-it-matters

全文完