**Bitmap加载和Cache**

参考：<http://www.runoob.com/w3cnote/android-tutorial-bitmap1.html>

* **Bitmap加载**
* **什么是Bitmap**

Bitmap在Android中指的是一张图片，可以是png格式，也可是jpg等其它常见的格式。

* **如何加载Bitmap**

BitmapFactory提供了四类方法：

1. BitmapFactory.decodeFile():从文件系统中加载。
2. BitmapFactory.decodeResourse():从资源中加载。
3. BitmapFactory.decodeStream():从输入流中加载。
4. BitmapFactory.decodeByteArray():从字节数组中加载。

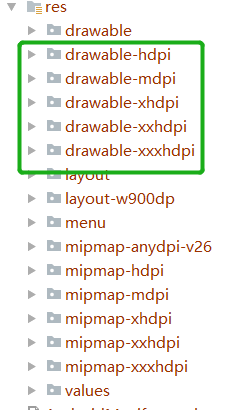
decodeFile()和decodeResourse()间接调用了decodeStream()方法。这四类方法最终是在Android底层的native方法实现的。这些方法的执行原理是一致的，读取目标文件转换成输入流，然后调用native方法解析流，解析完后调用Bitmap的构造方法，创建Bitmap对象并返回。

* **项目中图片**

参考：<https://www.jianshu.com/p/3f6f6e4f1c88>

https://juejin.im/post/5af84f4b51882542714fdaa9

项目中的图片一般res目录下，如下：



但是只有绿框内的文件夹下的图片被加载的时候可能会产生缩放。Android提供不同的drawable-\*\*\*文件夹来存放项目切图，项目运行在不同的屏幕密度的手机上会去对应的文件夹下加载切图，而且不同的屏幕密度一般也有对应的屏幕尺寸，所以，drawable-\*\*\*文件夹、屏幕密度、屏幕的分辨率，它们三者的关系如下：

drawable-mdpi（160dpi） 屏幕分辨率320\*480px

drawable-hdpi（240dpi） 屏幕分辨率480\*800px

drawable-xhdpi（320dpi） 屏幕分辨率720\*1280px

drawable-xxhdpi（480dpi） 屏幕分辨1080\*1920px

drawable-xxxhdpi（560dpi）

drawable-xxxxhdpi（640dpi）

其中屏幕密度的比例为1 **：**1.5 **：**2 **：**3 **：**3.5 **：**4。

而同一张图片分别放在这几个文件夹中的时候，在同一个设备上加载后的分辨率会有所缩放。缩放的比例公式：

**缩放倍数 = 设备像素密度 / 资源目录像素密度。**

如果图片放在了其他资源目录（包括drawable、mipmap-\*\*\*等），那么其他资源目录的分辨率默认为160，即：

**缩放倍数 = 设备像素密度 / 160。**

所以同一张图片放在不同的目录下，加载出来的分辨率是不一样的，同样加载出来所占内存也是不一样的，即，在drawable文件夹下加载的图片，所占内存公式如下：

**Bitmap内存占用 = 原始图片宽× 缩放倍数 × 原始图片高× 缩放倍数× 每个像素的字节大小。**

**可以简写为：**

**Bitmap内存占用 = 原始图片宽× 原始图片高× 缩放倍数的平方× 每个像素的字节大小。**

注意：Android drawable- \*\*\*文件夹选择图片逻辑是根据文件夹代表的密度由高到低选择的。

* **[android](http://lib.csdn.net/base/android" \o "Android知识库" \t "http://blog.csdn.net/tz_zs/article/details/_blank) 中像素的表现形式的三种模式**

三种表现像素模式如下：

**ARGB\_4444 : 2bytes 每个像素占据2 个字节；**A(Alpha)占4位的精度，R(Red)占4位的精度，G(Green)占4位的精度，B（Blue）占4位的精度，加起来一共是16位的精度，折合是2个字节，也就是一个像素占两个字节的内存，同时存储位图的透明度和颜色信息。**不过由于该精度的位图质量较差，官方不推荐使用**

**ARGB\_8888 : 4bytes 每个像素占据4 个字节；**这个类型的跟ARGB\_4444的原理是一样的，只是A,R,G,B各占8个位的精度，所以一个像素占4个字节的内存。由于该类型的位图质量较好，官方特别推荐使用。

**RGB\_565 : 2bytes 每个像素占据2 个字节；**R占5位精度，G占6位精度，B占5位精度，一共是16位精度，折合两个字节。注意，这个类型存储的只是颜色信息，没有透明度信息。

例子：如果一个大小为500k，像素为480\*800的位图设置了此类型，如果使用使用默认的像素表现模式RGB\_8888加载该图片，加载后占用的内存空间是：480\*800\*4/(1024\*1024)=1.5M，也就是说一个500K的图片需要1.5M的内存空间才能100%的将所有的像素表现出来。这显然是很耗内存的。如何高效的加载图片，继续往下看。

* **如何高效的加载Bitmap**

所谓的高效的加载Bitmap就是，通过ImageView显示图片的时候，如果Bitmap的尺寸比要显示的ImageView要大，将加载Bitmap进行缩小后加载。这样就会降低内存占用从而在一定程度上避免OOM，提高了Bitmap加载时的性能。

1. **通过采样率压缩分辨率**

通过BitmapFactory.Options来缩放图片，主要用到它的inSampleSize参数，即采样率：

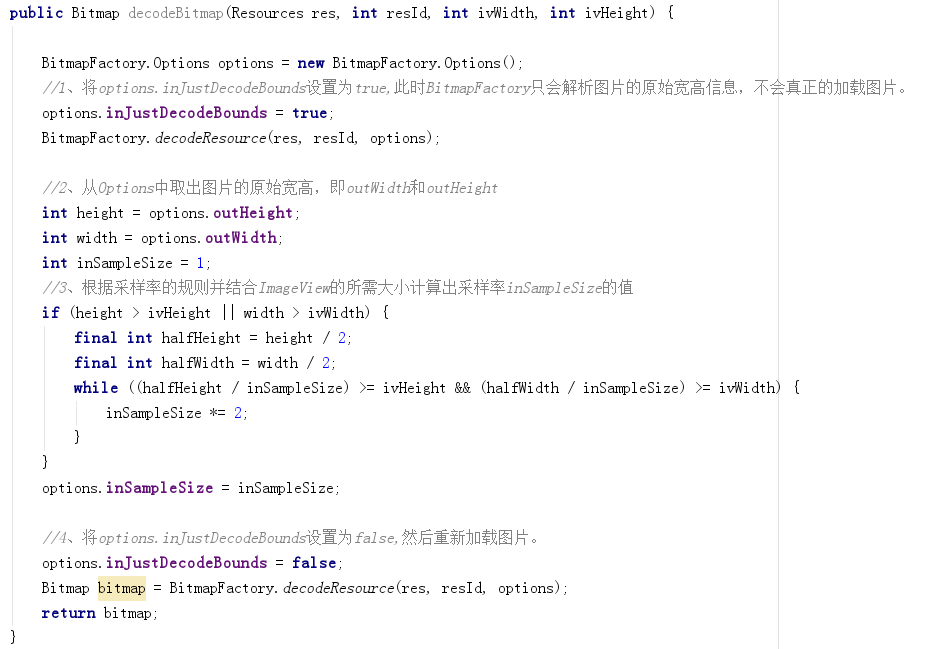
图片原始宽高 = 图片缩放宽高 \* 采样率

**通过设置采样率加载的图片，减少的是图像的像素点和尺寸（分辨率），图片的像素密度大小不变。**

例子：比如一张图片1024\*1024像素的图片，假定采用ARGB8888格式存储，那么他表现出来所有的像素需要占的内存是1024\*1024\*4，即4MB，如果加载的时候采样率inSampleSize设置为2，那么加载后占用的内存为512\*512\*4，即1MB。

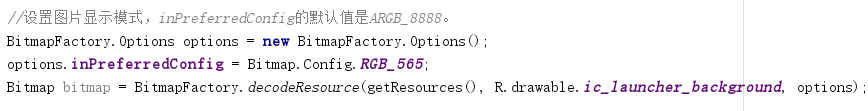
采样率：

1. 采样率必须是大于1的整数图片才会有缩小效果。假如采样率小于1，其作用相当于1，即无缩放效果。
2. 采样率同时作用于宽高，这将导致缩小后的图片所占内存的大小以采样率的2次方递减。比如inSampleSize为4，那么缩放比例就是1/16。
3. 官方文档建议inSampleSize的取值应该总为2的倍数，比如1、2、4、8、16等。如果不为2的倍数，那么系统会向下取整并选择一个最接近2的倍数来代替，比如1，系统会选择2来代替，但是经过验证发现，这个结论并非在所有的Android版本上都成立，因此只是当做一个开发建议即可。
4. 缩小后的图片的宽高最好不要比显示的ImageView的宽高小，否则会失真。比如，ImageView的大小是100\*100像素，而图片的原始大小是200\*200，那么采样率inSanpleSize只需要设置为2即可。假如图片为200\*300像素，这时候采样率还应该选择2，这样缩小后的图片大小为100\*150，仍然适合ImageView。如果采样率设置为3，那么缩小后的图片会小于ImageView所期望的大小，这样图片就会被拉伸从而导致模糊。



经过上面四个步骤加载出来的图片就是缩小后的图片，这样加载图片就能节省很多内存的浪费了。但是注意，步骤二中获取的图片宽高与图片的位置以及程序运行的设备有关，比如同一张图片放在不同的Drawable目录下，当程序运行在不同密度的设备上，这可能导致步骤二获取的图片信息不同，这和Android的资源加载机制有关。BitmapFactory加载图片的四种方式中只有decodeStream(从输入流中加载)与上面步骤比有点特殊。

**2、设置图片的像素显示模式**



系统默认加载图片的显示模式是RGB\_8888，这种模式下每个像素占4个字节，如果改为RGB\_565则加载出来所占内存你会减少一半。

**3、质量压缩**



质量压缩改变的是图片的位深（即色深）及透明度，所以这种方式压缩不会无限制的压缩，达到一个值后就不会变小了。

质量压缩不会影响图片的像素数，也不会影响图片的分辨率，所以不管压缩到多小，也不会影响加载这张图片时所占用内存大小。如上图代码中，传递进来imgBitmap是加载的原图的Bitmap，它与最后经过压缩后bitmap对象的所占内存大小是一样的，只不过再将这个压缩后的bitmap对象转成成图片后，这个图片的大小会比原来的图片要小。质量压缩这种方式适用于向服务器传递图片时使用。

注意图片存在形式：

1、文件（存储在硬盘上）。size = file.length()。

2、二进制流（存储在内存中）。size = 加载到内存中流的byte数。

3、Bitmap形势存在。 size = bitmap.getByteCount()

例如：一个文件形式的图片大小是100k，那么以流的方式加载到内存中，也是占用100k的内存。但是当把图片加载到内存中以Bitmap形式存在时占用的内存就不止100k了。质量压缩只会影响文件形式、以及流的形式的大小。

**4、合理放置项目中图片**

通过上面对res目录下图片加载的分析，Android加载res目录下图片会有缩放，缩放的比例具体看上面，既然有缩放，那么图片加载后的分辨率也会有变化，进而加载出来的bitmap占用内存也会有变化。当然，缩放系数越小bitmap占用内存就越小。根据上面的公式我们需要可以得出：

1. 我们将图片放在drawable-\*\*\*代表的像素密度越大的目录下，加载图片的缩放系数就越小。这样理论上，加载后的图片的分辨率是降下来了，但是低分辨率图片在显示上会出现失真的现象，而高分辨率图片会占用更多的内存，所以这方面的优化，看具体情况。
2. 尽量不要把图片放在drawable-\*\*\*意外的目录下，因为其他的目录默认代表的像素密度是160，现在的市面上的手机像素密度一般都很高，将图片放在其他目录来加载后，分辨率会很大。

**5、图片使用完要回收**

主要是在退出Activity中调用：

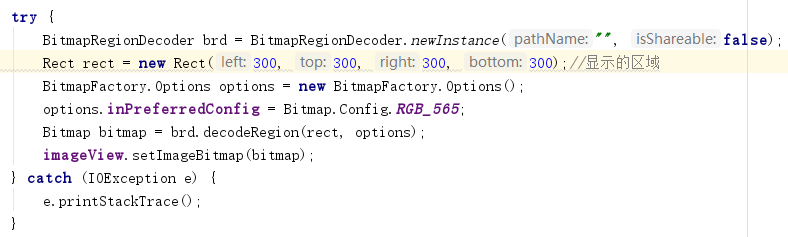
1533541525(1)

**6、加载高清原图**

我们面对加载一张超长图片，或者几兆（MB）起步图片使用最容易出现OOM，使用上面的几种方案显然乏力，这时候可以使用以下方案：

# 使用三方subsampling-scale-image-view控件。它还提供双击放大，手指滑动缩放，放大后拖拽浏览等功能。

1. 使用BitmapRegionDecoder类来显示图片的某一个区域。一张好几兆的巨图它的分辨率肯定也会很大，超过屏幕的分辨率很正常，我们肯定无法按照原尺寸进行加载，所以我们加载这张巨图的某一个区域，当然，显示巨图的某一个区域是没有意义的，所以我们可以借助这个类来自定义个一个可拖拽的控件，来浏览这张巨图，大体思路：重写控件的onTouchEvent方法，根据移动手势更新巨图的显示区域的参数，当更新完显示区域参数后，调用invalidate()方法，最后在onDraw里面获取需要显示的bitmap，并画到画布上。



* **Android中的缓存策略**
* **缓存策略**

缓存，当程序第一次从网络上加载图片后，将其缓存到磁盘，这样下载加载图片就不用从网络上获取了。有时候为了提高用户体验，往往还会把图片在内存中在缓存一份，这样当展示图片的时候，首先会去内存中查找，如果内存中没有，那么就去磁盘中查找，磁盘中再没有，就去网络中加载。这就是所谓的三级缓存。这样不仅为用户节省了流量而且还提高了用户体验。

* **缓存算法**

缓存涉及到的操作有三种：添加，获取和删除。因为不管是内存还是磁盘容量都有大小限制，所以我们需要指定一个缓存的大小，当缓存满了的时候我们就去删除一下旧的缓存，然后添加一下新的缓存。

缓存算法主要讲的是对于删除旧的缓存的策略。不同的策略对应着不同的算法。

* **LRU算法**

比较常见的算法是LRU(Least Recently Used)，它的核心思想是，会优先删除近期最少使用的缓存。采用LRU算法的缓存有两种：

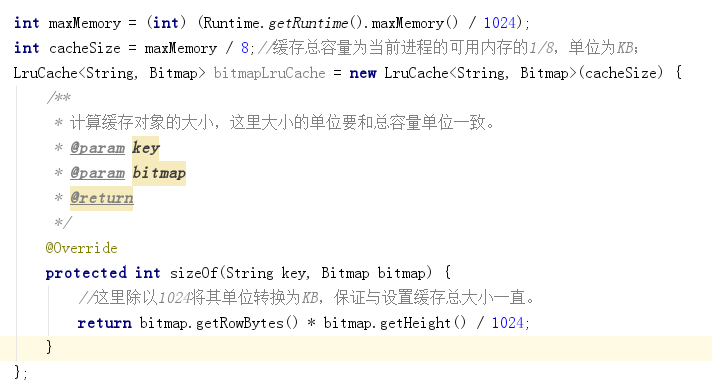
LruCash：用于内存缓存。

DiskLruCash：用于磁盘缓存。

* **LruCache**

LruCache是Android 3.1所提供的一个缓存类。通过suppre-v4兼容包可以兼容到3.1之前的Android版本。为了达到兼容性建议采用support-v4兼容包中提供的LruCache。

LruCache是一个泛型类，它内部采用一个LinkedHashMap以强引用的方式存储外界的缓存对象。通过put和get方法完成缓存的获取和存储，当缓存满的的时候LruCache会移除较早使用的缓存对象，然后再添加新的缓存对象。**另外LruCache是线程安全的**：



LruCache的使用很简单，只要提供缓存的总大小并重写sizeOf方法即可。另外在一些特殊的情况下，还需重写LruCache的entryRemoved方法，LruCache移除旧的缓存时会调用entryRemoved方法，因此可以在此方法中做一些资源回收的工作（如果有需要）。

* **DiskLruCache**

DiskLruCache通过将缓存对象写入文件系统从而实现缓存的效果。DiskLruCache不是Android SDK的一部分，但是得到了Android官方的推荐，所以要想使用必须从[android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/libcore/io/DiskLruCache.java](https://android.googlesource.com/platform/libcore/+/jb-mr2-release/luni/src/main/java/libcore/io/DiskLruCache.java" \t "http://blog.csdn.net/guolin_blog/article/details/_blank)下载下源码来加入到项目中。

**DiskLruCache创建**

DiskLruCache并不是new出来的，它提供了open方法创建自身。



这里解释下第一个参数，如果选择上面的两个路径做磁盘缓存路径，当卸载应用的时候缓存会被删除。如果选择其他的路径，则不会被删除。

第二个参数是应用版本号，当版本号发生改变的时候，不管选择的是不是默认的路径缓存路径下的缓存都会被清空。

**DiskLruCache的缓存添加**

缓存的添加是通过Editor完成的。