Android让人头疼的OOM，造成OOM的原因之一就是图片，现在的手机像素越来越高，随便一张图片都是好几M，甚至几十M，这样的照片加载到app，可想而知，随便加载几张图片，手机内存就不够用了，自然而然就造成了OOM，所以，Android的图片压缩异常重要。这里，我推荐一款开源框架——Luban

效果与对比

这里就不放效果图了，我拷贝了鲁班github上面的介绍——Android图片压缩工具，仿微信朋友圈压缩策略，因为是逆向推算，效果还没法跟微信一模一样，但是已经很接近微信朋友圈压缩后的效果，具体看以下对比!

内容	原图	Luban	Wechat
截屏 720P	720*1280,390k	720*1280,87k	720*1280,56k
截屏 1080P	1080*1920,2.21M	1080*1920,104k	1080*1920,112k
拍照 13M(4:3)	3096*4128,3.12M	1548*2064,141k	1548*2064,147k
拍照 9.6M(16:9)	4128*2322,4.64M	1032*581,97k	1032*581,74k
滚动截屏	1080*6433,1.56M	1080*6433,351k	1080*6433,482k

从这里就能看出，效果还是非常不错的

依赖

```
implementation 'top.zibin:Luban:1.1.3'1
```

调用方式

异步调用

Luban内部采用IO线程进行图片压缩，外部调用只需设置好结果监听即可：

```
Luban.with(this)
        .load(photos)                                     // 传入要压缩的图片列表
        .ignoreBy(100)                                    // 忽略不压缩图片的大小
        .setTargetDir(getPath())                          // 设置压缩后文件存储位置
        .setCompressListener(new OnCompressListener() { //设置回调
          @Override
          public void onStart() {
            // TODO 压缩开始前调用，可以在方法内启动 loading UI
          }

          @Override
          public void onSuccess(File file) {
            // TODO 压缩成功后调用，返回压缩后的图片文件
          }
```

```
          @Override
          public void onError(Throwable e) {
            // TODO 当压缩过程出现问题时调用
          }
        }).launch();    //启动压缩12345678910111213141516171819120
```

同步调用

同步方法请尽量避免在主线程调用以免阻塞主线程，下面以rxJava调用为例

```
Flowable.just(photos)
    .observeOn(Schedulers.io())
    .map(new Function<List<String>, List<File>>() {
      @Override public List<File> apply(@NonNull List<String> list) throws
Exception {
        // 同步方法直接返回压缩后的文件
        return Luban.with(MainActivity.this).load(list).get();
      }
    })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe();12345678910
```

以上，均是它github上面说明都有的，我这里就是copy过来了而已。重点要说的是，他是怎么实现的，源码分析。

源码分析

第一步：*Luban.with()*

点击去看到源码为：

```
public static Builder with(Context context) {
    return new Builder(context);
  }123
```

这里是一个静态的with方法，返回值是Builder，一般对设计模式比较熟悉的人，看到这里就应该懂了，他这里使用的是建造者模式。什么是建造者模式呢？建造者模式和工厂模式很相似，比工厂模式多了一个控制类，其实说白了，就是在创建对象的时候，减少初始化数据的代码，怎么理解呢？我们接着往下看。我们点到Builder里面看到如下代码：

```
public static class Builder {
    private Context context;//上下文对象
    private String mTargetDir;//压缩后图片存放位置
    private List<String> mPaths;//多个文件的list
    private int mLeastCompressSize = 100;//忽略100kb以下的图片，不压缩
    private OnCompressListener mCompressListener;//回调方法

    Builder(Context context) {
      this.context = context;
      this.mPaths = new ArrayList<>();
    }

    private Luban build() {
      return new Luban(this);
```

```
    }
}
```

我们看到了是一个静态的内部类Builder，我们这里看到了有5个变量，上面我们说道了，为了减少初始化数据的代码，就拿这个举例子说明，我如果有4个地方调用这个鲁班压缩，其中这4个地方，mTargetDir，mLeastCompressSize这2个变量的值都是一样的，其他3个不一样，按照我们以往的写法都得一个一个的赋值，要写4遍，那如果使用建造者模式了，这里就只用写一遍赋值，这2个变量。其他3个不一样，就得写多遍。当然，这是我个人对于建造者模式的理解。

我上面多粘贴了一个***build()***方法，为什么会多粘贴一个呢？就是为了更好的说明建造者模式，我们可以看到他这个方法，返回的是Luban对象，调用的是需要传Builder的构造方法，我们点进去看

```
private Luban(Builder builder) {
    this.mPaths = builder.mPaths;
    this.mTargetDir = builder.mTargetDir;
    this.mCompressListener = builder.mCompressListener;
    this.mLeastCompressSize = builder.mLeastCompressSize;
    mHandler = new Handler(Looper.getMainLooper(), this);
  }1234567
```

他这里就是赋值，他这个值就是Builder里面默认的，我们不论在哪里调用这个方法，都不用去一个一个赋值，因为，他已经处理好了。

第二步：load()

点击去看到源码为

```
public Builder load(File file) {
      this.mPaths.add(file.getAbsolutePath());
      return this;
    }

    public Builder load(String string) {
      this.mPaths.add(string);
      return this;
    }

    public Builder load(List<String> list) {
      this.mPaths.addAll(list);
      return this;
    }1234567891011121314
```

这里，我们会看到三个重载方法，一个传文件，他会获取到文件的绝对路径存进去，实际上还是存的字符串，中间那个存的是字符串，最后面那个传String类型的list，它调用的addAll方法，最后还是存的String在mPaths里面。我们点击mPaths，他就是一个String类型的list，在Builder的构造方法里面初始化的。他就是存放你的图片路径的集合

第三步：ignoreBy() 和 setTargetDir()

点击去看到源码为

```
    /**
     * do not compress when the origin image file size less than one value
     *
     * @param size
```

```
     *     the value of file size, unit KB, default 100K
     */
    public Builder ignoreBy(int size) {
      this.mLeastCompressSize = size;
      return this;
    }

    public Builder setTargetDir(String targetDir) {
      this.mTargetDir = targetDir;
      return this;
    }12345678910111213141516
```

这两个我为啥要放在一起讲呢？因为这两个没啥好说的，都是设置值，跟我们平时写的set方法的作用是一样的。没啥好说的

第四步：*setCompressListener(OnCompressListener listener)*

点击去看到源码为

```
public Builder setCompressListener(OnCompressListener listener) {
      this.mCompressListener = listener;
      return this;
    }1234
```

这个就是我们平时写自定义view的时候，要写回调方法，是一样的道理，他这里就是压缩方法的回调

第五步：*launch()*

点击去看到源码为

```
    /**
     * begin compress image with asynchronous
     */
    public void launch() {
      build().launch(context);
    }123456
```

这里，我们看到他先调用了build(),我们前面讲了，他这个方法就是赋值，然后调用了launch(context)方法，我们点进去看：

```
/**
   * start asynchronous compress thread
   */
  @UiThread private void launch(final Context context) {
    if (mPaths == null || mPaths.size() == 0 && mCompressListener != null) {
      mCompressListener.onError(new NullPointerException("image file cannot be
null"));
    }

    Iterator<String> iterator = mPaths.iterator();
    while (iterator.hasNext()) {
      final String path = iterator.next();
      if (Checker.isImage(path)) {
        AsyncTask.SERIAL_EXECUTOR.execute(new Runnable() {
```

```
          @Override public void run() {
            try {
              mHandler.sendMessage(mHandler.obtainMessage(MSG_COMPRESS_START));

              File result = Checker.isNeedCompress(mLeastCompressSize, path) ?
                  new Engine(path, getImageCacheFile(context,
Checker.checkSuffix(path))).compress() :
                  new File(path);

              mHandler.sendMessage(mHandler.obtainMessage(MSG_COMPRESS_SUCCESS,
result));
            } catch (IOException e) {
              mHandler.sendMessage(mHandler.obtainMessage(MSG_COMPRESS_ERROR,
e));
            }
          }
        });
      } else {
        Log.e(TAG, "can not read the path : " + path);
      }
      iterator.remove();
    }
  }123456789101112131415161718192021222324252627282930313233
```

这个方法就是最后，执行压缩的方法，前面都是初始化，我们可以看到，他这个方法是在主线程调用的，所以，我们不用考虑切换线程的问题，直接可以操作UI变化。我一步一步的讲：

1. 首先，他这个是用的迭代器，循环遍历，遍历一个就移除一个
2. 然后就是通过handler发消息调用
3. 具体压缩代码。最重要的就是第三点，我把第三点，提到下面讲

接着上面的第三点，具体压缩

```
File result = Checker.isNeedCompress(mLeastCompressSize, path) ?
                  new Engine(path, getImageCacheFile(context,
Checker.checkSuffix(path))).compress() :
                  new File(path);123
```

首先，他整体是一个三目运算符，我们点isNeedCompress()方法看一下

```
static boolean isNeedCompress(int leastCompressSize, String path) {
    if (leastCompressSize > 0) {
      File source = new File(path);
      if (!source.exists()) {
        return false;
      }

      if (source.length() <= (leastCompressSize << 10)) {
        return false;
      }
    }
    return true;
  }12345678910111213
```

这个方法就是用来判断，你给定路径的图片大小和你规定的忽略文件大小比较，他这里先做了你给定的最小值判断，要大于0，不大于0就返回ture。然后做了文件是否存在的判断，如果文件不存在，就返回fals。最后，给定文件大小是不是小于等于最小值左移10位的值，小于就返回false。

然后，如果返回的是true，就去压缩，如果，返回的是false，就直接返回file文件。压缩的方法点进去：

```
Engine(String srcImg, File tagImg) throws IOException {
    if (Checker.isJPG(srcImg)) {
      this.srcExif = new ExifInterface(srcImg);
    }
    this.tagImg = tagImg;
    this.srcImg = srcImg;

    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    options.inSampleSize = 1;

    BitmapFactory.decodeFile(srcImg, options);
    this.srcWidth = options.outWidth;
    this.srcHeight = options.outHeight;
  }123456789101112131415
```

这就又要说道另一个类了Engine类，它的类注释就是：用于操作，开始压缩，管理活动，缓存资源的类。他这里传原文件，也就是你需要压缩的图片，还有一个就是目标文件，也就是你压缩之后，要保存的文件。

我们先看第二个参数是什么怎么传的，有的人看不懂

```
/**
   * Returns a mFile with a cache audio name in the private cache directory.
   *
   * @param context
   *     A context.
   */
  private File getImageCacheFile(Context context, String suffix) {
    if (TextUtils.isEmpty(mTargetDir)) {
      mTargetDir = getImageCacheDir(context).getAbsolutePath();
    }

    String cacheBuilder = mTargetDir + "/" +
        System.currentTimeMillis() +
        (int) (Math.random() * 1000) +
        (TextUtils.isEmpty(suffix) ? ".jpg" : suffix);

    return new File(cacheBuilder);
  }
```

他这里就是新建一个文件，设置路径，设置名称，然后返回文件

再掉回去看Engine的构造方法，我们这里获取到了源文件和目标文件，我们只用把压缩后的流存到目标文件就行了。我之前写过一篇关于图片压缩的博客。它这里的option就是设置压缩的参数，不懂的可以看一下我之前的博客，或者用google百度一下就知道了。具体压缩就是用的bitmap的工厂类，调用的decodeFile方法。没错就是这一句 ***BitmapFactory.decodeFile(srcImg, options);***

最后，辣么一切都准备就绪了，怎么样开始压缩呢？***compress()***

```
File compress() throws IOException {
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inSampleSize = computeSize();

    Bitmap tagBitmap = BitmapFactory.decodeFile(srcImg, options);
    ByteArrayOutputStream stream = new ByteArrayOutputStream();

    tagBitmap = rotatingImage(tagBitmap);
    tagBitmap.compress(Bitmap.CompressFormat.JPEG, 60, stream);
    tagBitmap.recycle();

    FileOutputStream fos = new FileOutputStream(tagImg);
    fos.write(stream.toByteArray());
    fos.flush();
    fos.close();
    stream.close();

    return tagImg;
  }12345678910111213141516171819
```

这里面就是常规的压缩，存储的逻辑了，最最重要的压缩算法呢？就是这里的***computeSize()***方法

```
private int computeSize() {
    srcWidth = srcWidth % 2 == 1 ? srcWidth + 1 : srcWidth;
    srcHeight = srcHeight % 2 == 1 ? srcHeight + 1 : srcHeight;

    int longSide = Math.max(srcWidth, srcHeight);
    int shortSide = Math.min(srcWidth, srcHeight);

    float scale = ((float) shortSide / longSide);
    if (scale <= 1 && scale > 0.5625) {
      if (longSide < 1664) {
        return 1;
      } else if (longSide >= 1664 && longSide < 4990) {
        return 2;
      } else if (longSide > 4990 && longSide < 10240) {
        return 4;
      } else {
        return longSide / 1280 == 0 ? 1 : longSide / 1280;
      }
    } else if (scale <= 0.5625 && scale > 0.5) {
      return longSide / 1280 == 0 ? 1 : longSide / 1280;
    } else {
      return (int) Math.ceil(longSide / (1280.0 / scale));
    }
  }12345678910111213141516171819202122232425
private Bitmap rotatingImage(Bitmap bitmap) {
    if (srcExif == null) return bitmap;

    Matrix matrix = new Matrix();
    int angle = 0;
    int orientation = srcExif.getAttributeInt(ExifInterface.TAG_ORIENTATION,
ExifInterface.ORIENTATION_NORMAL);
    switch (orientation) {
      case ExifInterface.ORIENTATION_ROTATE_90:
        angle = 90;
```

```
        break;
      case ExifInterface.ORIENTATION_ROTATE_180:
        angle = 180;
        break;
      case ExifInterface.ORIENTATION_ROTATE_270:
        angle = 270;
        break;
    }

    matrix.postRotate(angle);

    return Bitmap.createBitmap(bitmap, 0, 0, bitmap.getWidth(),
bitmap.getHeight(), matrix, true);
  }12345678910111213141516171819202122
```

你以为我会一步一步给你讲Luban算法逻辑吗？那是不可能的，我特么都不会，怎么给你讲。我直接把他github上算法逻辑的介绍拷贝过来了：

1. 判断图片比例值，是否处于以下区间内；
    - [1, 0.5625) 即图片处于 [1:1 ~ 9:16) 比例范围内
    - [0.5625, 0.5) 即图片处于 [9:16 ~ 1:2) 比例范围内
    - [0.5, 0) 即图片处于 [1:2 ~ 1:∞) 比例范围内
2. 判断图片最长边是否过边界值；
    - [1, 0.5625) 边界值为：1664 * n（n=1），4990 * n（n=2），1280 * pow(2, n-1)（n≥3）
    - [0.5625, 0.5) 边界值为：1280 * pow(2, n-1)（n≥1）
    - [0.5, 0) 边界值为：1280 * pow(2, n-1)（n≥1）
3. 计算压缩图片实际边长值，以第2步计算结果为准，超过某个边界值则：width / pow(2, n-1)，height/pow(2, n-1)
4. 计算压缩图片的实际文件大小，以第2、3步结果为准，图片比例越大则文件越大。
   size = (newW * newH) / (width * height) * m；
    - [1, 0.5625) 则 width & height 对应 1664，4990，1280 * n（n≥3），m 对应 150，300，300；
    - [0.5625, 0.5) 则 width = 1440，height = 2560, m = 200；
    - [0.5, 0) 则 width = 1280，height = 1280 / scale，m = 500；注：scale为比例值
5. 判断第4步的size是否过小
    - [1, 0.5625) 则最小 size 对应 60，60，100
    - [0.5625, 0.5) 则最小 size 都为 100
    - [0.5, 0) 则最小 size 都为 100
6. 将前面求到的值压缩图片 width, height, size 传入压缩流程，压缩图片直到满足以上数值