我们知道，在Android框架中提供了很多异步处理的工具类。然而，他们中大部分实现是通过提供单一的后台线程来处理任务队列的。如果我们需要更多的后台线程的时候该怎么办呢?

大家都知道Android的UI更新是在UI线程中进行的（也称之为主线程）。所以如果我们在UI线程中编写耗时任务都可能会阻塞UI线程更新UI。为了避免这种情况我们可以使用 AsyncTask, IntentService和Threads。但是，Android提供的AsyncTasks和IntentService都是利用单一的后台线程来处理异步任务的。那么，开发人员如何创建多个后台线程呢?

更新: Marco Kotz 指出结合使用ThreadPool Executor和AsyncTask，后台可以有多个线程（默认为5个）同时处理AsyncTask。

创建多线程常用的方法

在大多数使用场景下，我们没有必要产生多个后台线程，简单的创建AsyncTasks或者使用基于任务队列的IntentService就可以很好的满足我们对异步处理的需求。然而当我们真的需要多个后台线程的时候，我们常常会使用下面的代码简单的创建多个线程。

```
String[] urls = …        for (final String url : urls) {
    new Thread(new Runnable() {
        public void run() {
            // 调用API、下载数据或图片
        }
    }
    ).start();
}
```

该方法有几个问题。一方面，操作系统限制了同一域下连接数（限制为4）。这意味着，你的代码并没有真的按照你的意愿执行。新建的线程如果超过数量限制则需要等待旧线程执行完毕。 另外，每一个线程都被创建来执行一个任务，然后销毁。这些线程也没有被重用。

常用方法存在的问题

举个例子，如果你想开发一个连拍应用能在1秒钟连拍10张图片（或者更多）。应用该具备如下的子任务:

- 在一秒的时间内扑捉10张以byte[]形式储存的照片，并且不能够阻塞UI线程。
- 将byte[]储存的数据格式从YUV转换成RGB。
- 使用转换后的数据创建Bitmap。
- 变换Bitmap的方向。
- 生成缩略图大小的Bitmap。
- 将全尺寸的Bitmap以Jpeg压缩文件的格式写入磁盘中。
- 使用上传队列将图片保存到服务器中。

很明显，如果你将太多的子任务放在UI线程中，你的应用在性能上的表现将不会太好。在这种情况下，唯一的解决方案就是先将相机预览的数据缓存起来，当UI线程闲置的时候再来利用缓存的数据执行剩下的任务。

另外一个可选的解决方案是创建一个长时间在后台运行的HandlerThread，它能够接受相机预览的数据，并处理完剩下的全部任务。当然这种做法的性能会好些，但是如果用户想再连拍的话，将会面临较大的延迟，因为他需要等待HandlerThread处理完前一次连拍。

```
public class CameraHandlerThread extends HandlerThread
            implements Camera.PictureCallback, Camera.PreviewCallback {
```

```
    private static String TAG = "CameraHandlerThread";
    private static final int WHAT_PROCESS_IMAGE = 0;
    Handler mHandler = null;
    WeakReference<camerapreviewfragment> ref = null;
    private PictureUploadHandlerThread mPictureUploadThread;
    private Boolean mBurst = false;
    private int mCounter = 1;
    CameraHandlerThread(CameraPreviewFragment cameraPreview) {
        super(TAG);
        start();
        mHandler = new Handler(getLooper(), new Handler.Callback() {
            @Override
                        public Boolean handleMessage(Message msg) {
                if (msg.what == WHAT_PROCESS_IMAGE) {
                    // 业务逻辑
                }
                return true;
            }
        }
        );
        ref = new WeakReference<>(cameraPreview);
    }
    ...
            @Override
            public void onPreviewFrame(byte[] data, Camera camera) {
        if (mBurst) {
            CameraPreviewFragment f = ref.get();
            if (f != null) {
                mHandler.obtainMessage(WHAT_PROCESS_IMAGE, data).sendToTarget();
                try {
                    sleep(100);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
                if (f.isAdded()) {
                    f.readyForPicture();
                }
            }
            if (mCounter++ == 10) {
                mBurst = false;
                mCounter = 1;
            }
        }
    }
}
```

看起来所有的任务都被后台的单一线程处理完毕了，我们性能提升主要得益于后台线程长期运行并不会被销毁和重建。然而，我们后台的单一线程却要和其他优先等级更高的任务共享，而且这些任务只能够顺序执行。

我们也可以创建第二个HandlerThread来处理我们的图像，然后创建第三个HandlerThread来将照片写入磁盘，最后再创建第四个HandlerThread来将照片上传到服务器中。我们能够加快拍照的速度，但是，这些线程相互之间还是遵循顺序执行的规则，并不是真的并发。因为每张照片是顺序处理的，而且处理每一张照片需要一定的时间，导致用户在点击拍照按钮到显示全部缩略图的时候仍然能够明显的感觉到延迟。

使用ThreadPool并发处理任务

我们可以根据需求创建多个线程，但是创建过多的线程会消耗CPU周期影响性能，并且线程的创建和销毁也需要时间成本。所以我们不想创建多余的线程，但是又想能够充分的利用设备的硬件资源。这个时候我们可以使用ThreadPool。

通过创建ThreadPool对象的单例来在你的应用中使用ThreadPool。

```
public class BitmapThreadPool {
    private static BitmapThreadPool mInstance;
    private ThreadPoolExecutor mThreadPoolExec;
    private static int MAX_POOL_SIZE;
    private static final int KEEP_ALIVE = 10;
    BlockingQueue<runnable> workQueue = new LinkedBlockingQueue<>();
    public static synchronized void post(Runnable runnable{
        if (mInstance == null) {
            mInstance = new BitmapThreadPool();
        }
        mInstance.mThreadPoolExec.execute(runnable);
    }
    private BitmapThreadPool() {
        int coreNum = Runtime.getRuntime().availableProcessors();
        MAX_POOL_SIZE = coreNum * 2;
        mThreadPoolExec = new ThreadPoolExecutor(
                            coreNum,
MAX_POOL_SIZE,                                          KEEP_ALIVE,
                        TimeUnit.SECONDS,
                            workQueue);
    }
    public static void finish() {
        mInstance.mThreadPoolExec.shutdown();
    }
}
```

然后，在上面的代码中，简单的修改Handler的回调函数为：

```
mHandler = new Handler(getLooper(), new Handler.Callback() {
    @Override
          public Boolean handleMessage(Message msg) {
        if (msg.what == WHAT_PROCESS_IMAGE) {
            BitmapThreadPool.post(new Runnable() {
                @Override
                                    public void run() {
                    // 做你想做的任何事情
                }
            }
            );
        }
        return true;
    }
}
);
```

优化已经完成！通过下面的视频，我们观察到加载缩略图的速度提升是非常明显的。

这种做法的优点是我们可以定义线程池的大小并且指定空余线程保持活动的时间。我们也可以创建多个ThreadPools来处理多个任务或者使用单个ThreadPool来处理多个任务。但是在使用完后记得清理资源。

我们甚至可以为每一个功能创建一个独立的ThreadPool。譬如说在这个例子中我们可以创建三个ThreadPool,第一个ThreadPool负责数据转换成Bitmap，第二个ThreadPool负责写数据到磁盘中去，第三个ThreadPool上传Bitmap到服务器中去。这样做的话，如果我们的ThreadPool最大拥有4条线程，那么我们就能够同时的转换，写入，上传四张相片。用户将看到4张缩略图是同时显示而不是一个个的显示出来的。

使用ThreadPool前: 如果可以，从顶部观察计数器的变化来得知当底部缩略图从开始显示到全部显示完成所耗费的时间。在程序中除了adapter中的notifyDataSetChanged()方法外，我已经将大部分的操作从主线程中剥离，所以计数器的运行是很流畅的。

使用ThreadPool后: 通过顶部的计数器，我们发现使用了ThreadPool后，照片的缩略图加载速度明显变快。