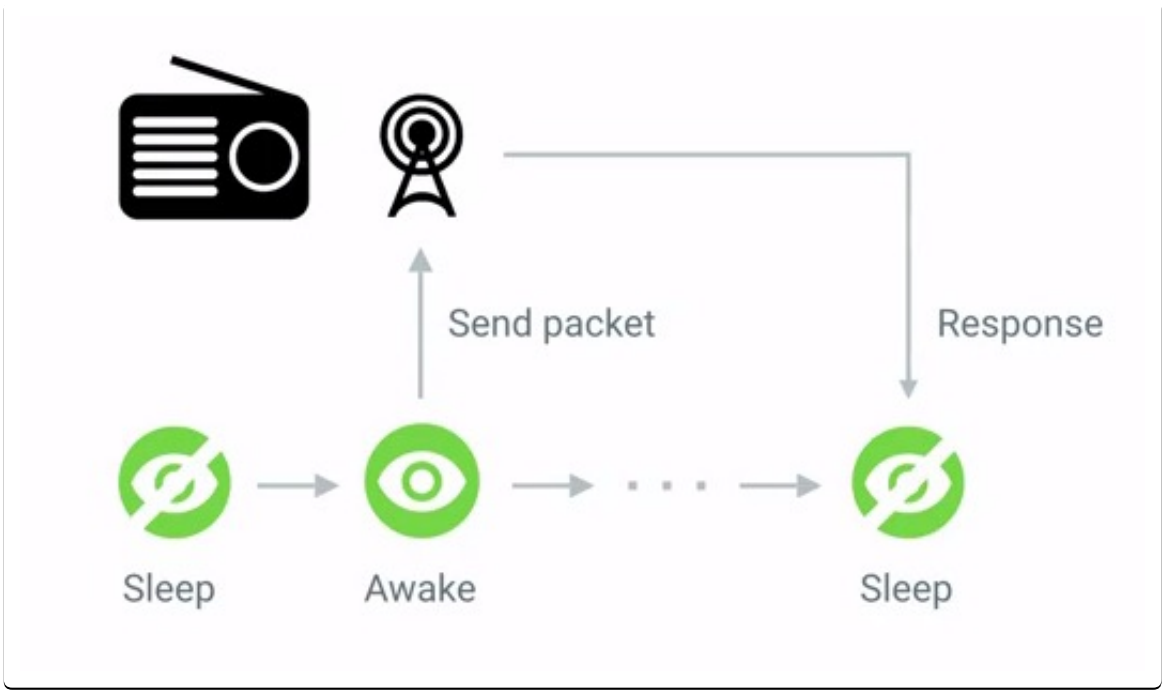
APR 29TH, 2015 | COMMENTS

Android性能优化典范 - 第2季

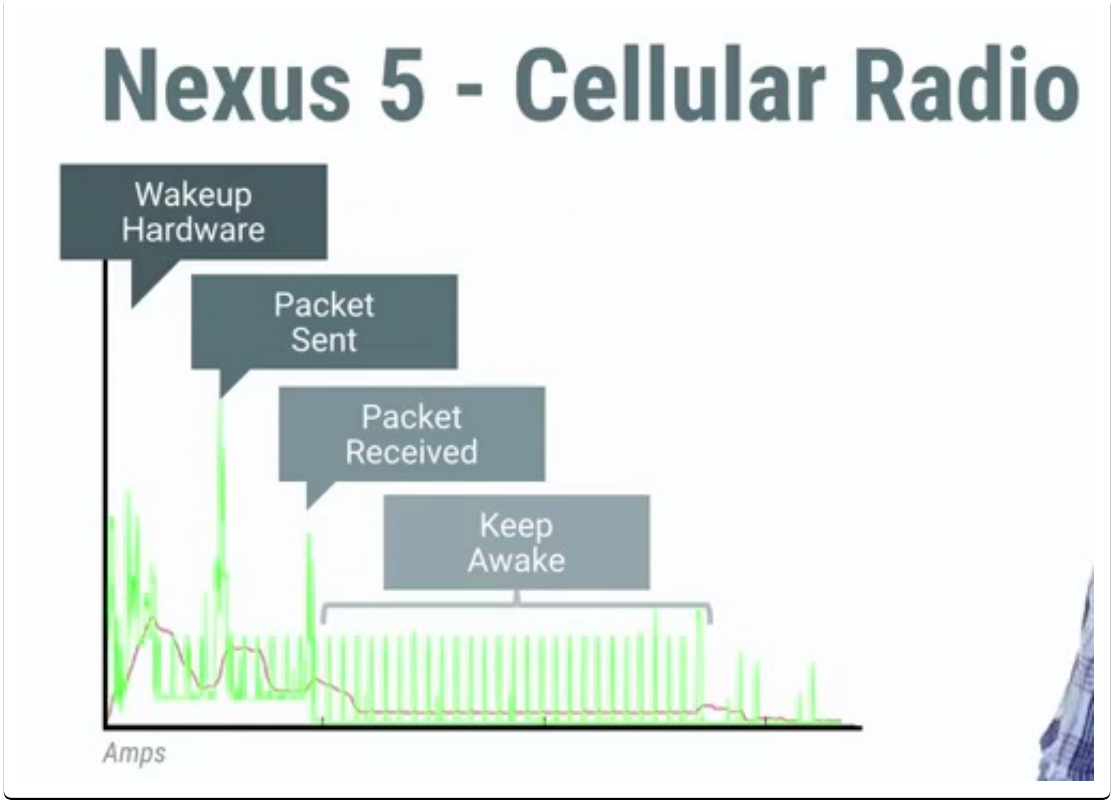
Google前几天刚发布了Android性能优化典范第2季的课程，一共20个短视频，包括的内容大致有：电量优化，网络优化，Wear上如何做优化，使用对象池来提高效率，LRU Cache，Bitmap的缩放，缓存，重用，PNG压缩，自定义View的性能，提升设置alpha之后View的渲染性能，以及Lint，StictMode等等工具的使用技巧。 下面是对这些课程的总结摘要，认知有限，理解偏差的地方请多多指教！

1)Battery Drain and Networking

对于手机程序，网络操作相对来说是比较耗电的行为。优化网络操作能够显著节约电量的消耗。在性能优化第1季里面有提到过，手机硬件的各个模块的耗电量是不一样的，其中移动蜂窝模块对电量消耗是比较大的，另外蜂窝模块在不同工作强度下，对电量的消耗也是有差异的。当程序想要执行某个网络请求之前，需要先唤醒设备，然后发送数据请求，之后等待返回数据，最后才慢慢进入休眠状态。这个流程如下图所示：



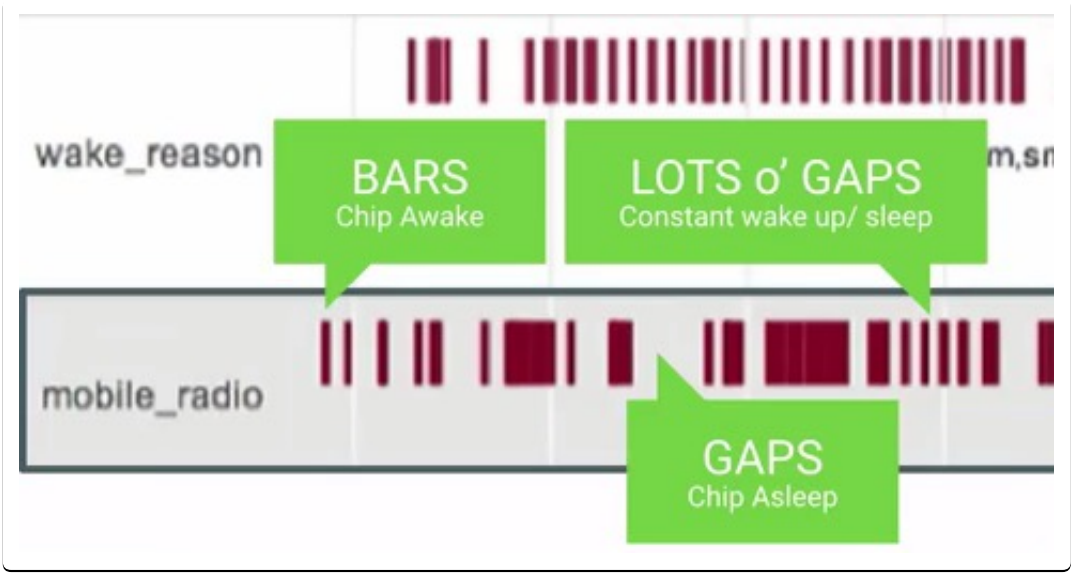
在上面那个流程中，蜂窝模块的电量消耗差异如下图所示：



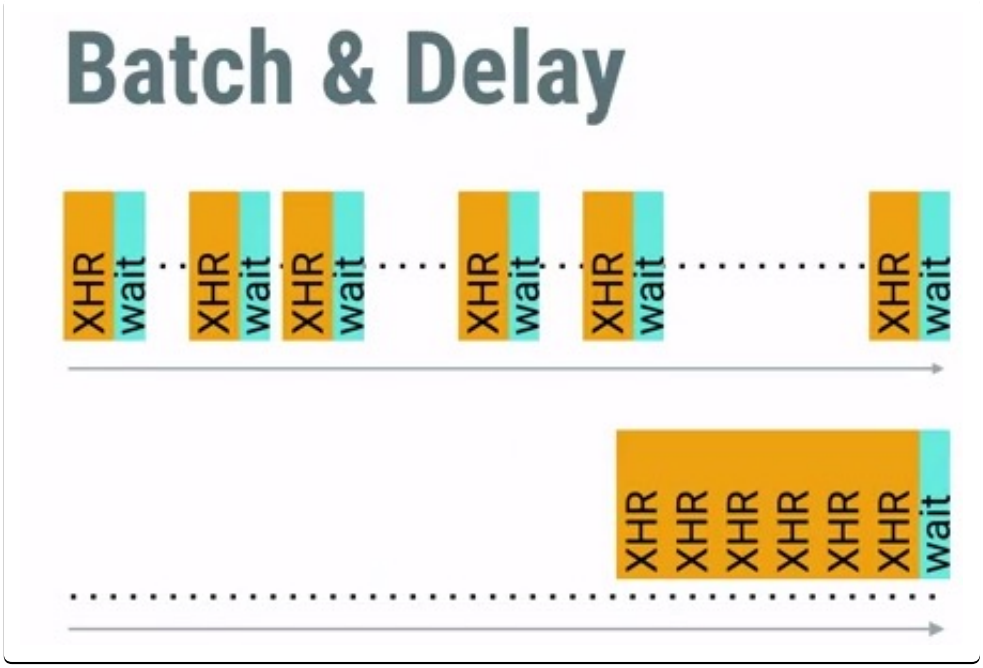
从图示中可以看到，激活瞬间，发送数据的瞬间，接收数据的瞬间都有很大的电量消耗，所以，我们应该从如何传递网络数据以及何时发起网络请求这两个方面来着手优化。

1.1)何时发起网络请求

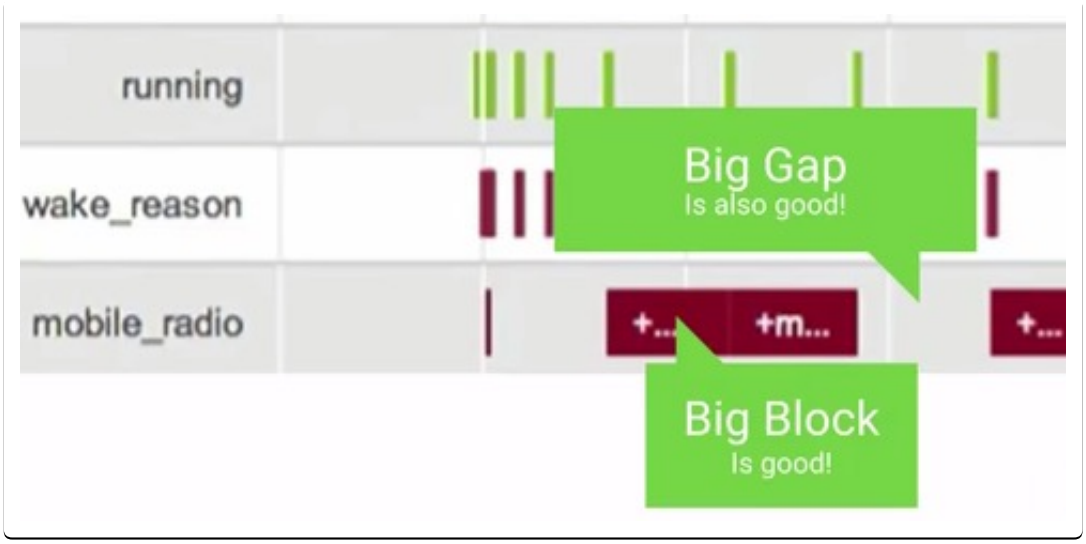
首先我们需要区分哪些网络请求是需要及时返回结果的，哪些是可以延迟执行的。例如，用户主动下拉刷新列表，这种行为需要立即触发网络请求，并等待数据返回。但是对于上传用户操作的数据，同步程序设置等等行为则属于可以延迟的行为。我们可以通过Battery Historian这个工具来查看关于移动蜂窝模块的电量消耗（关于这部分的细节，请点击Android性能优化之电量篇）。在Mobile Radio那一行会显示蜂窝模块的电量消耗情况，红色的部分代表模块正在工作，中间的间隔部分代表模块正在休眠状态，如果看到有一段区间，红色与间隔频繁的出现，那就说明这里有可以优化的行为。如下图所示：



对于上面可以优化的部分，我们可以有针对性的把请求行为捆绑起来，延迟到某个时刻统一发起请求。如下图所示：



经过上面的优化之后，我们再回头使用Battery Historian导出电量消耗图，可以看到唤醒状态与休眠状态是连续大块间隔的，这样的话，总体电量的消耗就会变得更少。



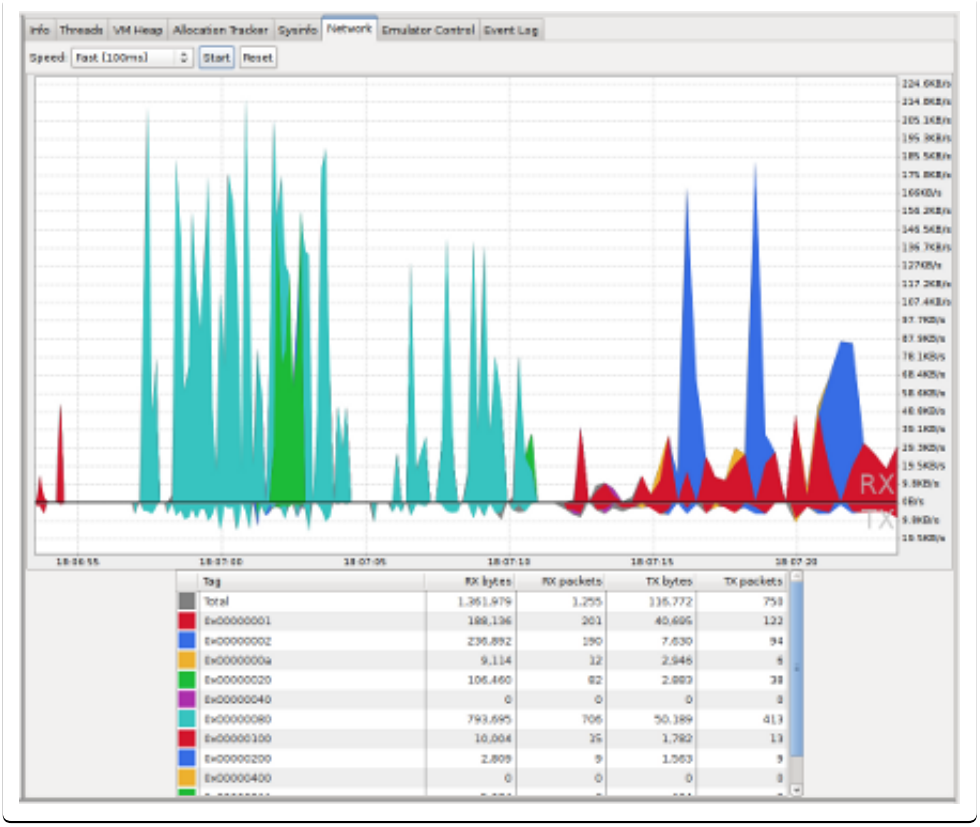
当然，我们甚至可以把请求的任务延迟到手机网络切换到WiFi，手机处于充电状态下再执行。在前面的描述过程中，我们会遇到的一个难题是如何把网络请求延迟，并批量进行执行。还好，Android提供了JobScheduler来帮助我们达成这个目标。

1.2)如何传递网络数据

关于这部分主要会涉及到Prefetch(预取)与Compressed(压缩)这两个技术。对于Prefetch的使用，我们需要预先判断用户在此次操作之后，后续零散的请求是否很有可能会马上被触发，可以把后面5分钟有可能会使用到的零散请求都一次集中执行完毕。对于Compressed的使用，在

上传与下载数据之前，使用CPU对数据进行压缩与解压，可以很大程度上减少网络传输的时间。

想要知道我们的应用程序中网络请求发生的时间，每次请求的数据量等等信息，可以通过Android Studio中的Networking Traffic Tool来查看详细的数据，如下图所示：



2)Wear & Sensors

在Android Wear上会大量的使用Sensors来实现某些特殊功能，如何在尽量节约电量的前提下利用好Sensor会是我们需要特别注意的问题。下面会介绍一些在Android Wear上的最佳实践典范。

尽量减少刷新请求，例如我们可以在不需要某些数据的时候尽快注销监听，减小刷新频率，对Sensor的数据做批量处理等等。那么如何做到这些优化呢？

- 首先我们需要尽量使用Android平台提供的既有运动数据，而不是自己去实现监听采集数据，因为大多数Android Watch自身记录Sensor数据的行为是有经过做电量优化的。
- 其次在Activity不需要监听某些Sensor数据的时候需要尽快释放监听注册。
- 还有我们需要尽量控制更新的频率，仅仅在需要刷新显示数据的时候才触发获取最新数据的操作。
- 另外我们可以针对Sensor的数据做批量处理，待数据累积一定次数或者某个程度的时候才更新到UI上。

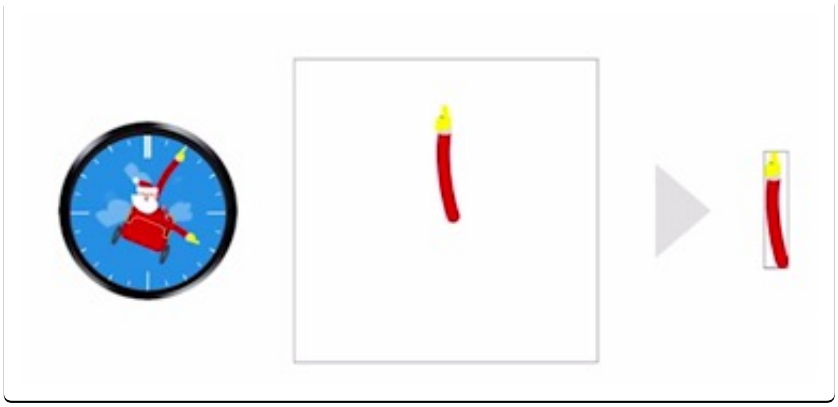
- 最后当Watch与Phone连接起来的时候，可以把某些复杂操作的事情交给Phone来执行，Watch只需要等待返回的结果。

更对关于Sensors的知识，可以点击这里

3)Smooth Android Wear Animation

Android Material Design风格的应用采用了大量的动画来进行UI切换，优化动画的性能不仅能够提升用户体验还可以减少电量的消耗，下面会介绍一些简单易行的方法。

在Android里面一个相对操作比较繁重的事情是对Bitmap进行旋转，缩放，裁剪等等。例如在一个圆形的钟表图上，我们把时钟的指针抠出来当做单独的图片进行旋转会比旋转一张完整的圆形图的所形成的帧率要高56%。



另外尽量减少每次重绘的元素可以极大的提升性能，假如某个钟表界面上有很多需要显示的复杂组件，我们可以把这些组件做拆分处理，例如把背景图片单独拎出来设置为一个独立的View，通过setLayerType()方法使得这个View强制用Hardware来进行渲染。至于界面上哪些元素需要做拆分，他们各自的更新频率是多少，需要有针对性的单独讨论。

如何使用Systrace等工具来查看某些View的渲染性能，在前面的章节里面有提到过，感兴趣的可以点击这里

对于大多数应用中的动画，我们会使用PropertyAnimation或者ViewAnimation来操作实现，Android系统会自动对这些Animation做一定的优化处理，在Android上面学习到的大多数性能优化的知识同样也适用于Android Wear。

想要获取更多关于Android Wear中动画效果的优化，请点击WatchFace这个范例。

4)Android Wear Data Batching

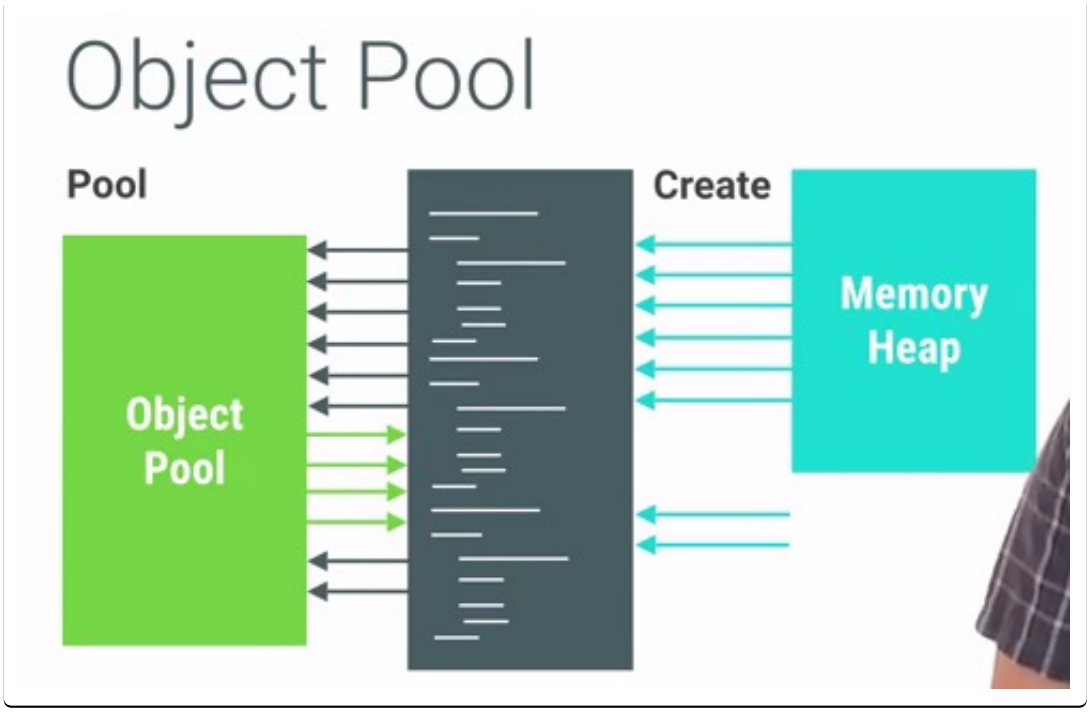
在Android Training里面有关于Wear上面如何利用Wearable API与Phone进行沟通协作的课程(详情请点击这里)。因为Phone的CPU与电量都比Wear要强大，另外Phone还可以直接接入网络，而Wear要接入网络则相对更加困难，所以我们在开发Wear应用的时候需要尽量做到把复杂的操作交给Phone来执行。例如我们可以让Phone来获取天气信息，然后把数据返回Wear进行显示。更进一步，在之前的性能优化课程里面我们有学习过如何使用JobScheduler来延迟批量处理任务，假设Phone收到来自Wear的其中一个任务是每隔5分钟检查一次天气情况，那么Phone使用JobScheduler执行检查天气任务之后，先判断这次返回的结果和之前是否有差异，仅仅当天气发生变化的时候，才有必要把结果通知到Wear，或者仅仅把变化的某一项数据通知给Wear，这样可以更大程度上减少Wear的电量消耗。

下面我们总结一下如何优化Wear的性能与电量：

- 仅仅在真正需要刷新界面的时候才发出请求
- 尽量把计算复杂操作的任务交给Phone来处理
- Phone仅仅在数据发生变化的时候才通知到Wear
- 把零碎的数据请求捆绑一起再进行操作

5)Object Pools

在程序里面经常会遇到的一个问题是短时间内创建大量的对象，导致内存紧张，从而触发GC导致性能问题。对于这个问题，我们可以使用对象池技术来解决它。通常对象池中的对象可能是bitmaps，views，paints等等。关于对象池的操作原理，不展开述说了，请看下面的图示：



使用对象池技术有很多好处，它可以避免内存抖动，提升性能，但是在使用的时候有一些内容是需要特别注意的。通常情况下，初始化的对象池里面都是空白的，当使用某个对象的时候先

去对象池查询是否存在，如果不存在则创建这个对象然后加入对象池，但是我们也可以在程序刚启动的时候就事先为对象池填充一些即将要使用到的数据，这样可以在需要使用到这些对象的时候提供更快的首次加载速度，这种行为就叫做**预分配**。使用对象池也有不好的一面，程序员需要手动管理这些对象的分配与释放，所以我们需要慎重地使用这项技术，避免发生对象的内存泄漏。为了确保所有的对象能够正确被释放，我们需要保证加入对象池的对象和其他外部对象没有互相引用的关系。

6)To Index or Iterate?

遍历容器是编程里面一个经常遇到的场景。在Java语言中，使用Iterate是一个比较常见的方法。可是在Android开发团队中，大家却尽量避免使用Iterator来执行遍历操作。下面我们看下在Android上可能用到的三种不同的遍历方法：

List (Iterator)

```
for(Iterator it = list.iterator(); it.hasNext();){
    Object obj = it.next();
    ...
}
```

List (For-Index)

```
int size = list.size();
for (int index = 0; index < size; index++) {
    Object object = list.get(index);
    ...
}
```

List (Simplified)

```
for (Object obj : list) {
    ...
}
```

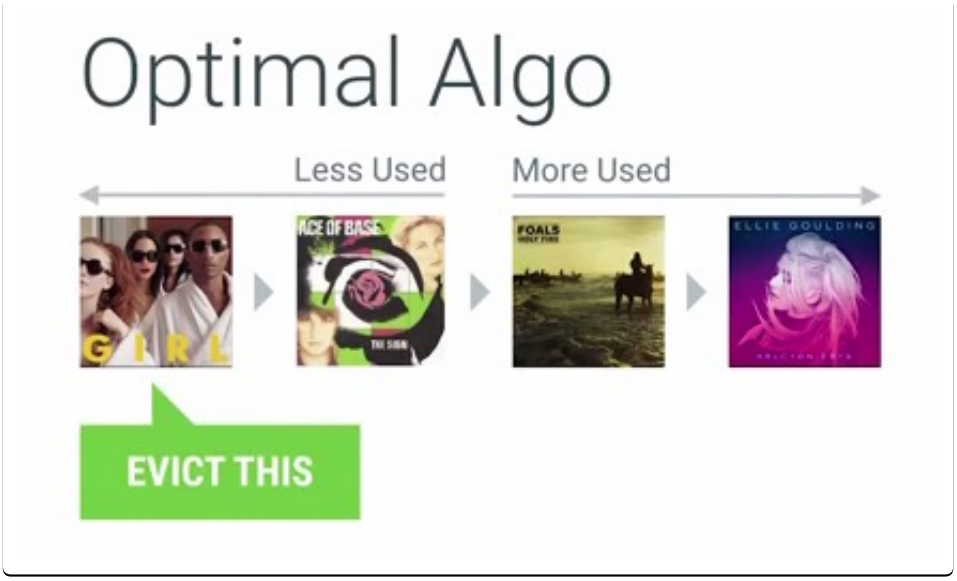
使用上面三种方式在同一台手机上，使用相同的数据集做测试，他们的表现性能如下所示：

Fcn	Time Taken(ms)
for index (ArrayList)	2603
for index (Vector)	4664
for simple (ArrayList)	5133
Iterator (ArrayList)	5142
Iterator (Vector)	11778
for simple (Vector)	11783

从上面可以看到for index的方式有更好的效率，但是因为不同平台编译器优化各有差异，我们最好还是针对实际的方法做一下简单的测量比较好，拿到数据之后，再选择效率最高的那个方式。

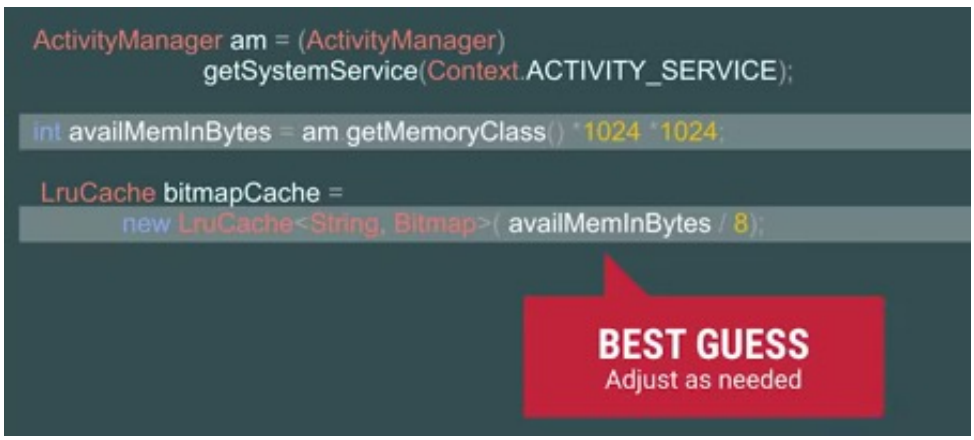
7)The Magic of LRU Cache

这小节我们要讨论的是缓存算法，在Android上面最常用的一个缓存算法是LRU(Least Recently Use)，关于LRU算法，不展开述说，用下面一张图演示下含义：

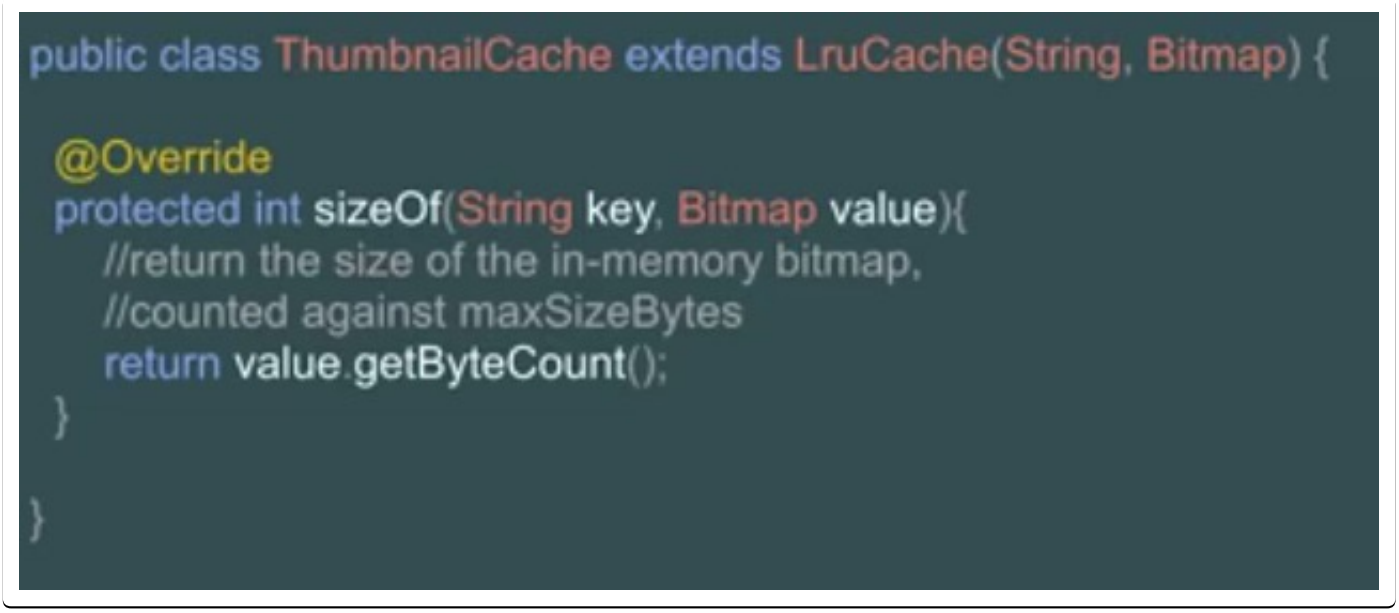


LRU Cache的基础构建用法如下：

为了给LRU Cache设置一个比较合理的缓存大小值，我们通常是用下面的方法来做界定的：



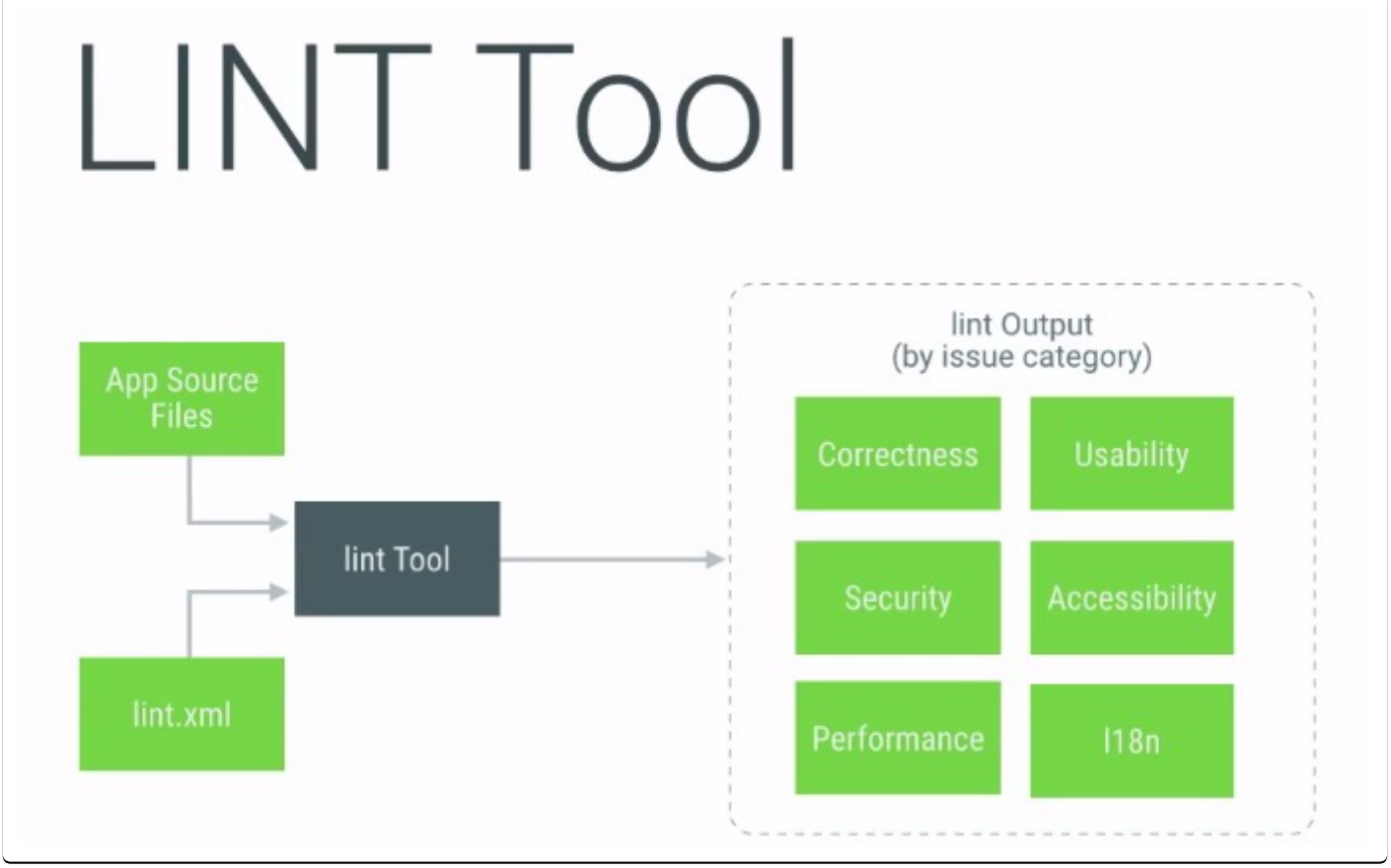
使用LRU Cache时为了能够让Cache知道每个加入的Item的具体大小，我们需要Override下面的方法：



使用LRU Cache能够显著提升应用的性能，可是也需要注意LRU Cache中被淘汰对象的回收，否者会引起严重的内存泄露。

8)Using LINT for Performance Tips

Lint是Android提供的一个静态扫描应用源码并找出其中的潜在问题的一个强大的工具。



例如，如果我们在onDraw方法里面执行了new对象的操作，Lint就会提示我们这里有性能问题，并提出对应的建议方案。Lint已经集成到Android Studio中了，我们可以手动去触发这个工具，点击工具栏的Analysis -> Inspect Code，触发之后，Lint会开始工作，并把结果输出到底部的工具栏，我们可以逐个查看原因并根据指示做相应的优化修改。

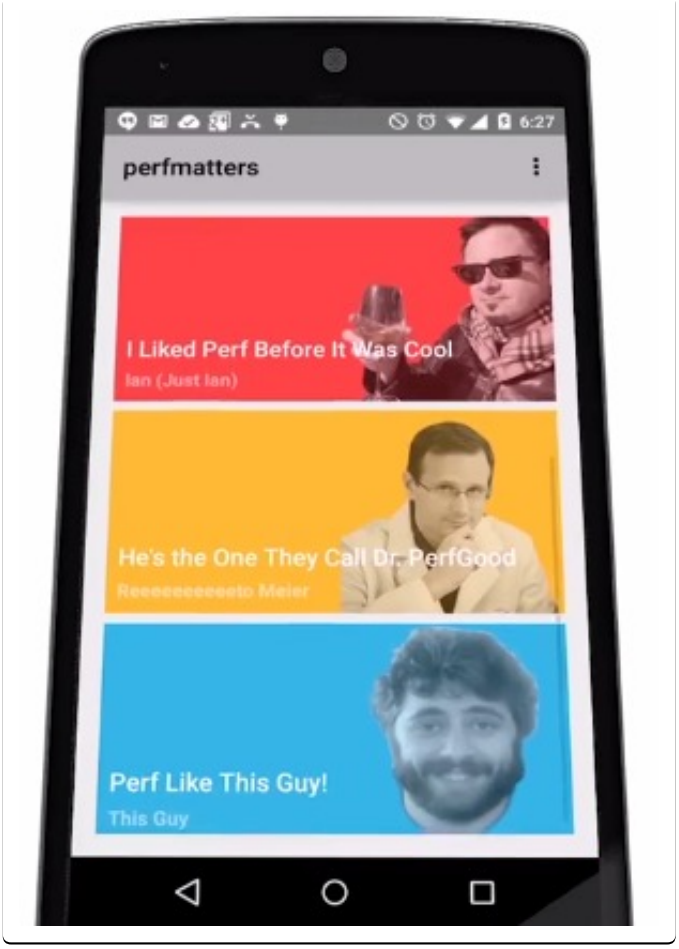
Lint的功能非常强大，他能够扫描各种问题。当然我们可以通过Android Studio设置找到Lint，对Lint做一些定制化扫描的设置，可以选择忽略掉那些不想Lint去扫描的选项，我们还可以针对部分扫描内容修改它的提示优先级。

建议把与内存有关的选项中的严重程度标记为红色的Error，对于Layout的性能问题标记为黄色Warning。

9)Hidden Cost of Transparency

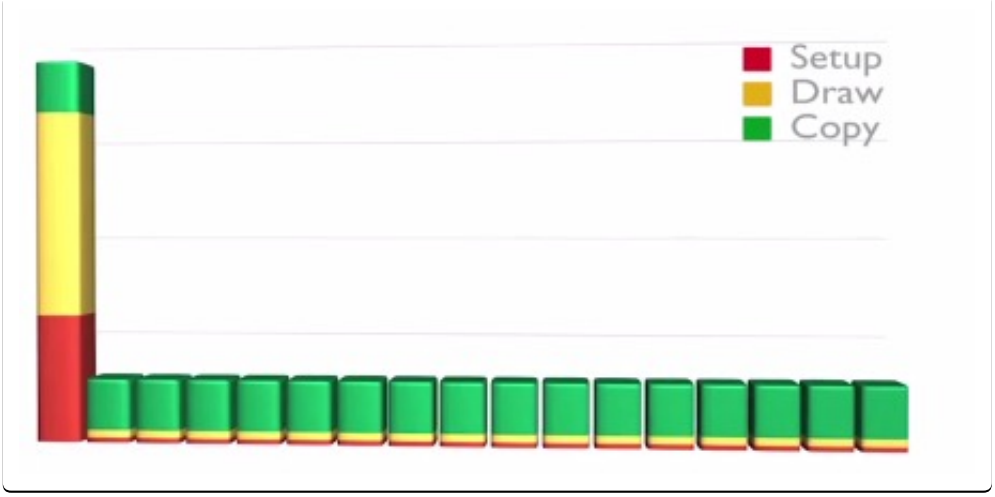
这小节会介绍如何减少透明区域对性能的影响。通常来说，对于不透明的View，显示它只需要渲染一次即可，可是如果这个View设置了alpha值，会至少需要渲染两次。原因是包含alpha的view需要事先知道混合View的下一层元素是什么，然后再结合上层的View进行Blend混色处理。

在某些情况下，一个包含alpha的View有可能会触发改View在HierarchyView上的父View都被额外重绘一次。下面我们看一个例子，下图演示的ListView中的图片与二级标题都有设置透明度。



大多数情况下，屏幕上的元素都是由后向前进行渲染的。在上面的图示中，会先渲染背景图(蓝，绿，红)，然后渲染人物头像图。如果后渲染的元素有设置alpha值，那么这个元素就会和屏幕上已经渲染好的元素做blend处理。很多时候，我们会给整个View设置alpha的来达到fading的动画效果，如果我们图示中的ListView做alpha逐渐减小的处理，我们可以看到ListView上的TextView等等组件会逐渐融合到背景色上。但是在这个过程中，我们无法观察到它其实已经触发了额外的绘制任务，我们的目标是让整个View逐渐透明，可是期间ListView在不停的做Blending的操作，这样会导致不少性能问题。

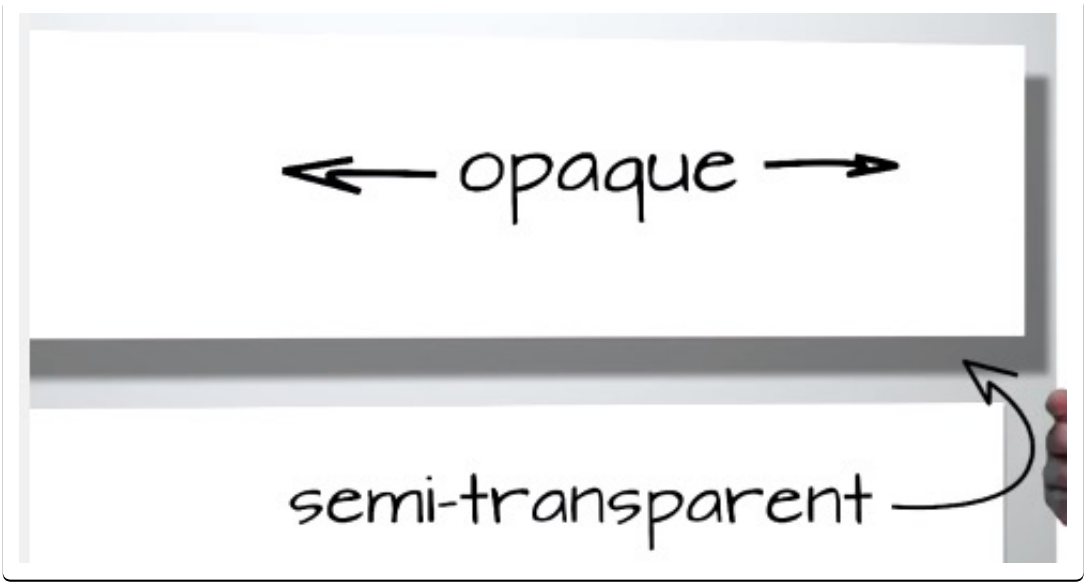
如何渲染才能够得到我们想要的效果呢？我们可以先按照通常的方式把View上的元素按照从后到前的方式绘制出来，但是不直接显示到屏幕上，而是使用GPU预处理之后，再又GPU渲染到屏幕上，GPU可以对界面上的原始数据直接做旋转，设置透明度等等操作。使用GPU进行渲染，虽然第一次操作相比起直接绘制到屏幕上更加耗时，可是一旦原始纹理数据生成之后，接下去的操作就比较省时省力。



如何才能够让GPU来渲染某个View呢？我们可以通过setLayerType的方法来指定View应该如何进行渲染，从SDK 16开始，我们还可以使用ViewPropertyAnimator.alpha().withLayer()来指定。如下图所示：

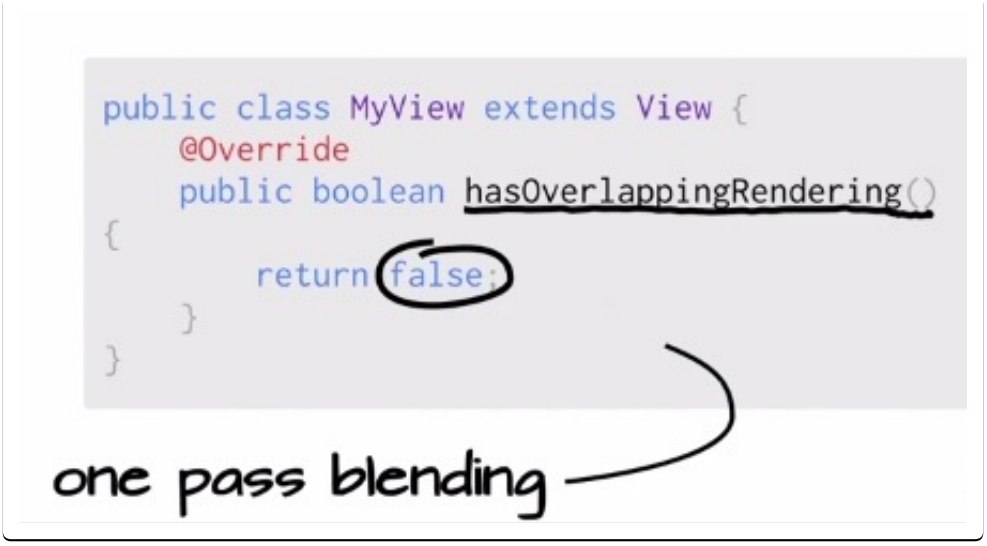
```
setLayerType(View.LAYER_TYPE_HARDWARE, null);

ViewPropertyAnimator.alpha(0.0f).withLayer();

setLayerType(View.LAYER_TYPE_NONE, null);
```

另外一个例子是包含阴影区域的View，这种类型的View并不会出现我们前面提到的问题，因为他们并不存在层叠的关系。

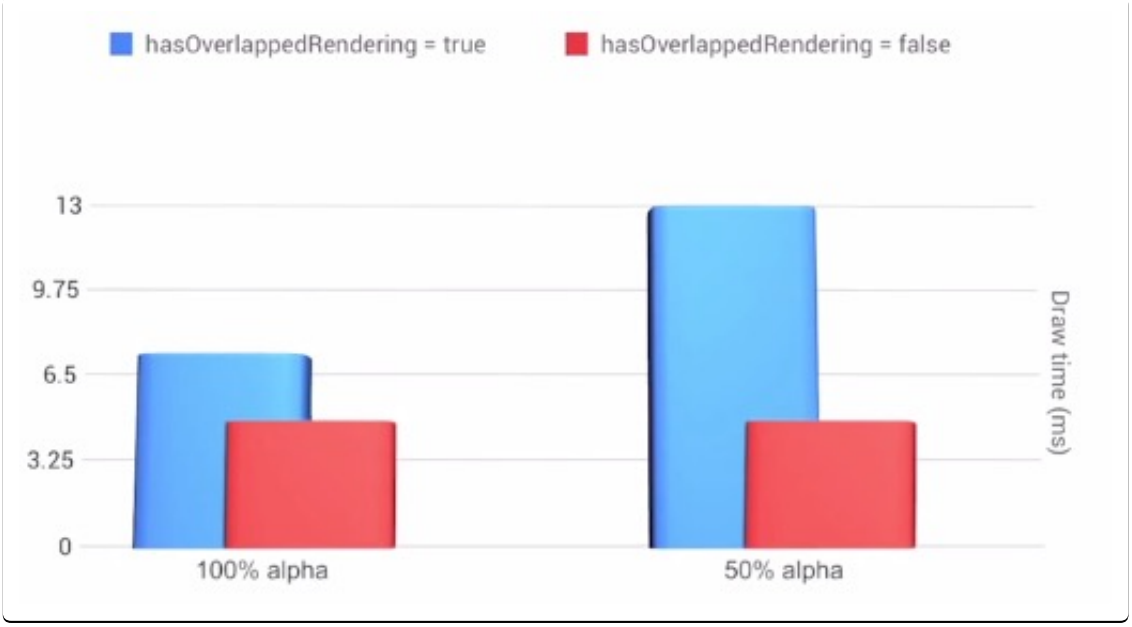


为了能够让渲染器知道这种情况，避免为这种View占用额外的GPU内存空间，我们可以做下

面的设置。



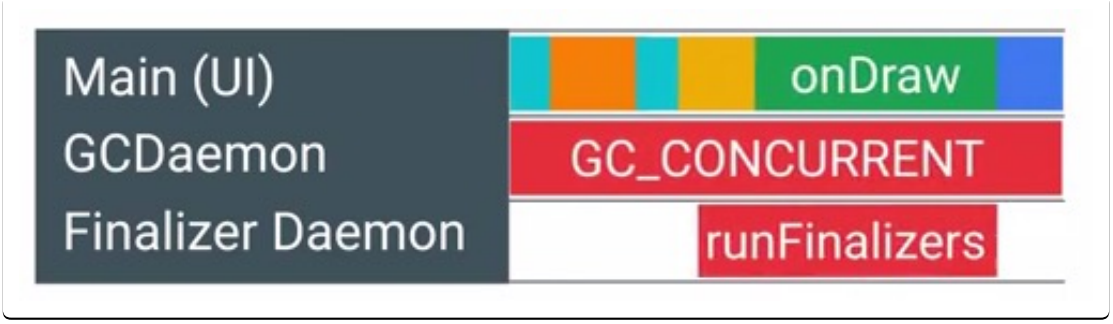
通过上面的设置以后，性能可以得到显著的提升，如下图所示：



10)Avoiding Allocations in onDraw()

我们都知道应该避免在onDraw()方法里面执行导致内存分配的操作，下面讲解下为何需要这样做。

首先onDraw()方法是执行在UI线程的，在UI线程尽量避免做任何可能影响到性能的操作。虽然分配内存的操作并不需要花费太多系统资源，但是这并不意味着是免费无代价的。设备有一定的刷新频率，导致View的onDraw方法会被频繁的调用，如果onDraw方法效率低下，在频繁刷新累积的效应下，效率低的问题会被扩大，然后会对性能有严重的影响。



如果在onDraw里面执行内存分配的操作，会容易导致内存抖动，GC频繁被触发，虽然GC后来被改进为执行在另外一个后台线程(GC操作在2.3以前是同步的，之后是并发)，可是频繁的GC的操作还是会影响到CPU，影响到电量的消耗。

那么简单解决频繁分配内存的方法就是把分配操作移动到onDraw()方法外面，通常情况下，我们会把onDraw()里面new Paint的操作移动到外面，如下面所示：

```
static Paint mPaint = new Paint();

static {
  mPaint.setColor(0xA4C739);
}
```

11)Tool: Strict Mode

UI线程被阻塞超过5秒，就会出现ANR，这太糟糕了。防止程序出现ANR是很重要的事情，那么如何找出程序里面潜在的坑，预防ANR呢？很多大部分情况下执行很快的方法，但是他们有可能存在巨大的隐患，这些隐患的爆发就很容易导致ANR。

Android提供了一个叫做Strict Mode的工具，我们可以通过手机设置里面的开发者选项，打开Strict Mode选项，如果程序存在潜在的隐患，屏幕就会闪现红色。我们也可以通过StrictMode API在代码层面做细化的跟踪，可以设置StrictMode监听那些潜在问题，出现问题时如何提醒开发者，可以对屏幕闪红色，也可以输出错误日志。下面是官方的代码示例：

```
1  public void onCreate() {
2       if (DEVELOPER_MODE) {
3           StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
4                   .detectDiskReads()
5                   .detectDiskWrites()
6                   .detectNetwork()   // or .detectAll() for all detectable prob
7                   .penaltyLog()
8                   .build());
9           StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
```

```
                .detectLeakedSqlLiteObjects()
                .detectLeakedClosableObjects()
                .penaltyLog()
                .penaltyDeath()
                .build());
        }
        super.onCreate();
    }
```

12)Custom Views and Performance

Android系统有提供超过70多种标准的View，例如TextView，ImageView，Button等等。在某些时候，这些标准的View无法满足我们的需要，那么就需要我们自己来实现一个View，这节会介绍如何优化自定义View的性能。

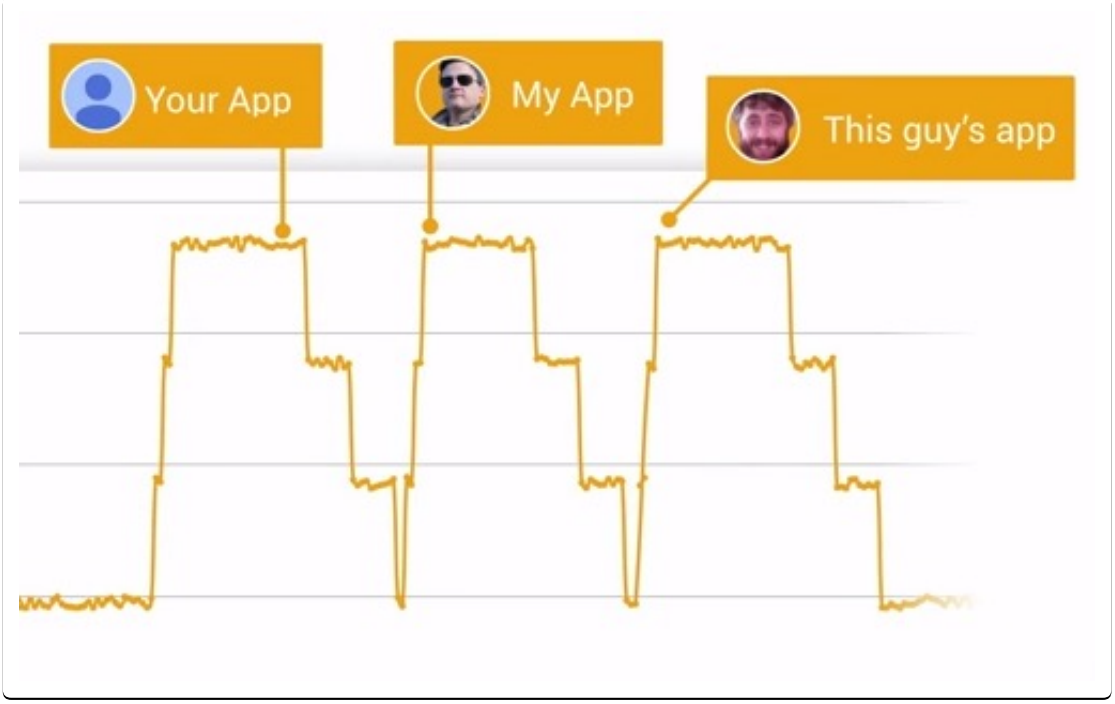
通常来说，针对自定义View，我们可能犯下面三个错误：

- **Useless calls to onDraw()**：我们知道调用View.invalidate()会触发View的重绘，有两个原则需要遵守，第1个是仅仅在View的内容发生改变的时候才去触发invalidate方法，第2个是尽量使用ClipRect等方法来提高绘制的性能。
- **Useless pixels**：减少绘制时不必要的绘制元素，对于那些不可见的元素，我们需要尽量避免重绘。
- **Wasted CPU cycles**：对于不在屏幕上的元素，可以使用Canvas.quickReject把他们给剔除，避免浪费CPU资源。另外尽量使用GPU来进行UI的渲染，这样能够极大的提高程序的整体表现性能。

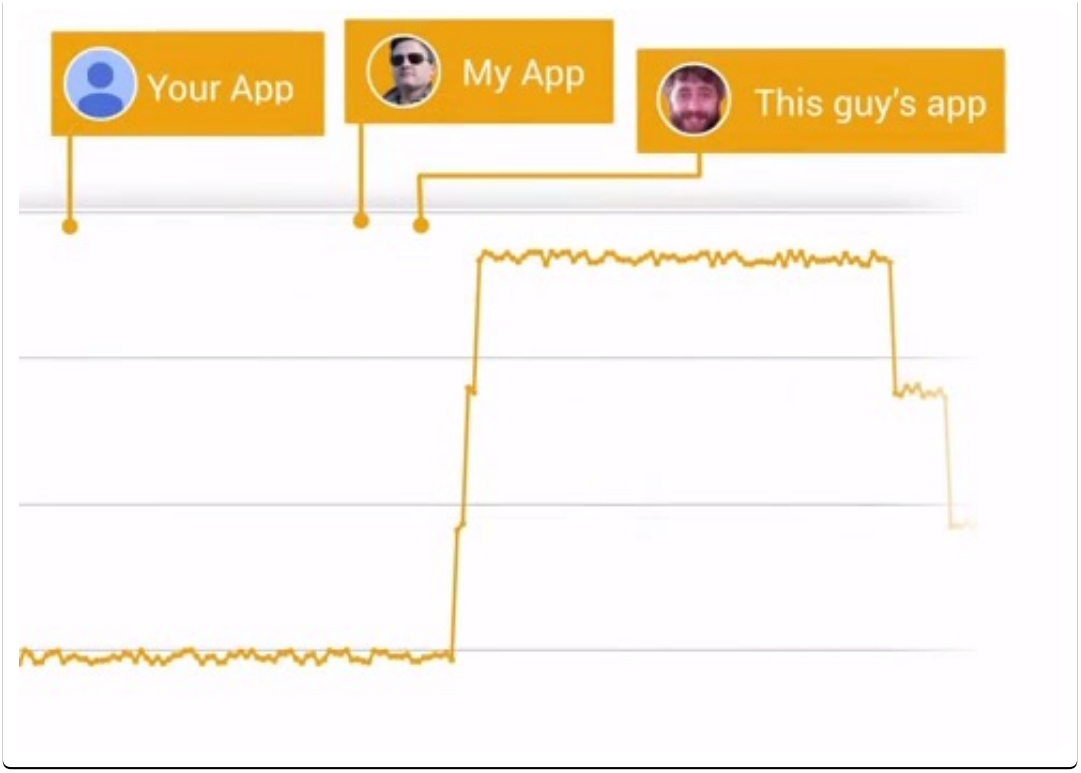
最后请时刻牢记，尽量提高View的绘制性能，这样才能保证界面的刷新帧率尽量的高。更多关于这部分的内容，可以看这里

13)Batching Background Work Until Later

优化性能时大多数时候讨论的都是如何减少不必要的操作，但是选择何时去执行某些操作同样也很重要。在第1季以及上一期的性能优化之电量篇里面，我们有提到过移动蜂窝模块的电量消耗模型。为了避免我们的应用程序过多的频繁消耗电量，我们需要学习如何把后台任务打包批量，并选择一个合适的时机进行触发执行。下图是每个应用程序各自执行后台任务导致的电量消耗示意图：



因为像上面那样做会导致浪费很多电量，我们需要做的是把部分应用的任务延迟处理，等到一定时机，这些任务一并进行处理。结果如下面的示意图：



执行延迟任务，通常有下面三种方式：

1)AlarmManager

使用AlarmManager设置定时任务，可以选择精确的间隔时间，也可以选择非精确时间作为参数。除非程序有很强烈的需要使用精确的定时唤醒，否者一定要避免使用他，我们应该尽量使用非精确的方式。

2)SyncAdapter

我们可以使用SyncAdapter为应用添加设置账户，这样在手机设置的账户列表里面可以找到我们的应用。这种方式功能更多，但是实现起来比较复杂。我们可以从这里看到官方的培训课程：http://developer.android.com/training/sync-adapters/index.html

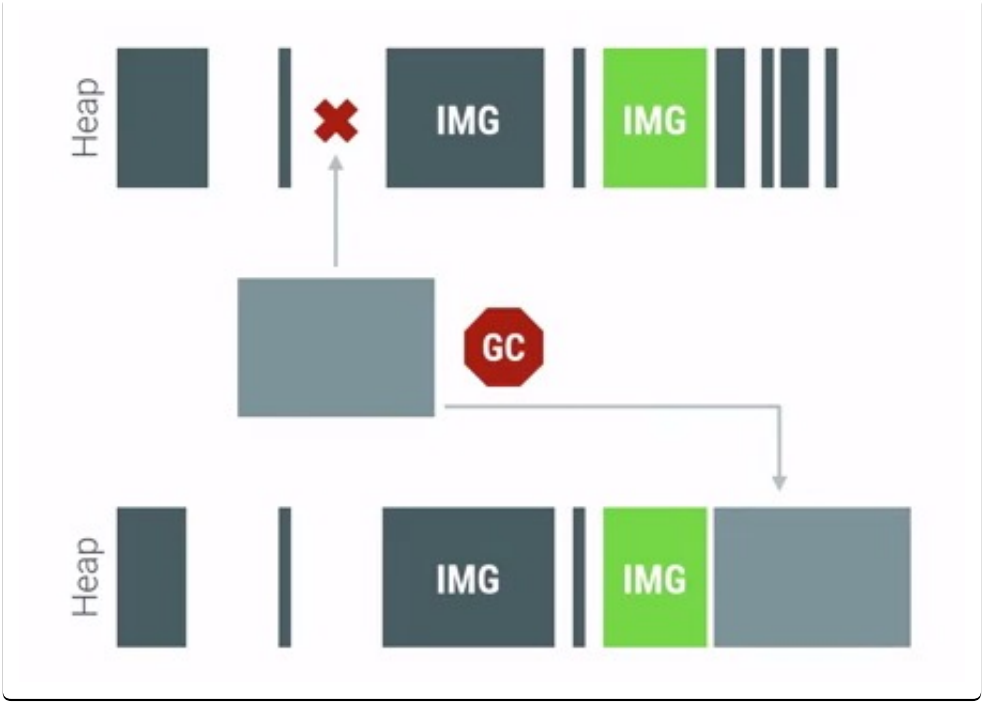
3)JobSchedulor

这是最简单高效的方法，我们可以设置任务延迟的间隔，执行条件，还可以增加重试机制。

14)Smaller Pixel Formats

常见的png,jpeg,webp等格式的图片在设置到UI上之前需要经过解码的过程，而解压时可以选择不同的解码率，不同的解码率对内存的占用是有很大差别的。在不影响到画质的前提下尽量减少内存的占用，这能够显著提升应用程序的性能。

Android的Heap空间是不会自动做兼容压缩的，意思就是如果Heap空间中的图片被收回之后，这块区域并不会和其他已经回收过的区域做重新排序合并处理，那么当一个更大的图片需要放到heap之前，很可能找不到那么大的连续空闲区域，那么就会触发GC，使得heap腾出一块足以放下这张图片的空闲区域，如果无法腾出，就会发生OOM。如下图所示：



所以为了避免加载一张超大的图片，需要尽量减少这张图片所占用的内存大小，Android为图片提供了4种解码格式，他们分别占用的内存大小如下图所示：

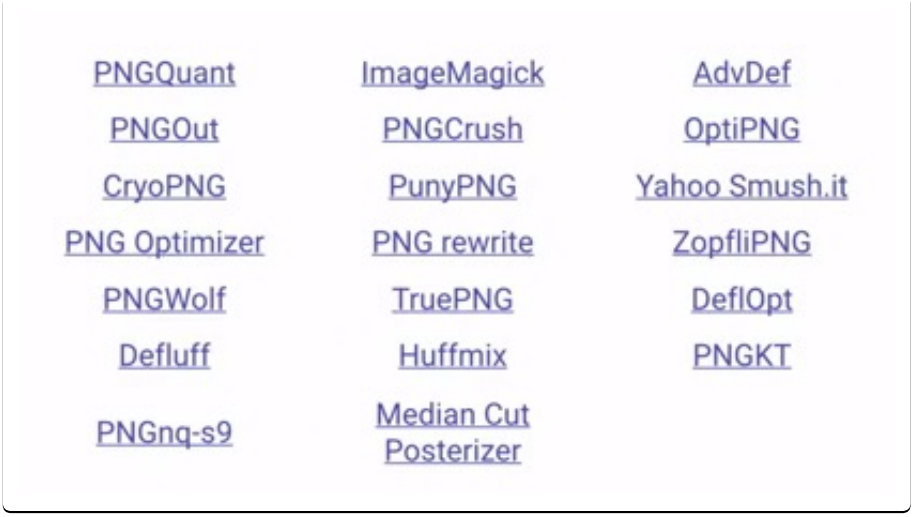
Format	Bits Per Pixel
ARGB_8888	32
RGB_565	16
ARGB_4444	16
ALPHA_8	8

随着解码占用内存大小的降低，清晰度也会有损失。我们需要针对不同的应用场景做不同的处理，大图和小图可以采用不同的解码率。在Android里面可以通过下面的代码来设置解码率：

```
mBitmapOptions = new BitmapFactory.Options();

mBitmapOptions.inPreferredConfig =
        Bitmap.Config.RGB_565;

BitmapFactory.decodeResource(getResources(),
        R.drawable.firstBitmap,
        mBitmapOptions);
```

15)Smaller PNG Files

尽量减少PNG图片的大小是Android里面很重要的一条规范。相比起JPEG，PNG能够提供更加清晰无损的图片，但是PNG格式的图片会更大，占用更多的磁盘空间。到底是使用PNG还是JPEG，需要设计师仔细衡量，对于那些使用JPEG就可以达到视觉效果的，可以考虑采用JPEG即可。我们可以通过Google搜索到很多关于PNG压缩的工具，如下图所示：



这里要介绍一种新的图片格式：Webp，它是由Google推出的一种既保留png格式的优点，又能够减少图片大小的一种新型图片格式。关于Webp的更多细节，请点击这里

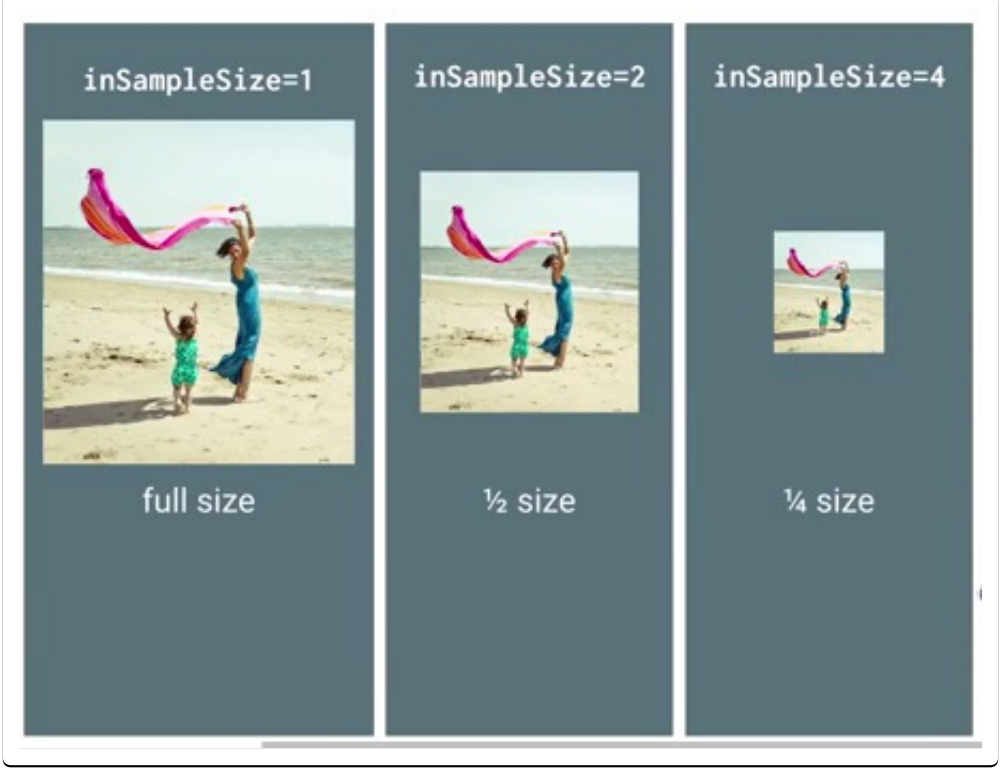
16)Pre-scaling Bitmaps

对bitmap做缩放，这也是Android里面最遇到的问题。对bitmap做缩放的意义很明显，提示显示性能，避免分配不必要的内存。Android提供了现成的bitmap缩放的API，叫做createScaledBitmap()，使用这个方法可以获取到一张经过缩放的图片。

上面的方法能够快速的得到一张经过缩放的图片，可是这个方法能够执行的前提是，原图片需要事先加载到内存中，如果原图片过大，很可能导致OOM。下面介绍其他几种缩放图片的方式。

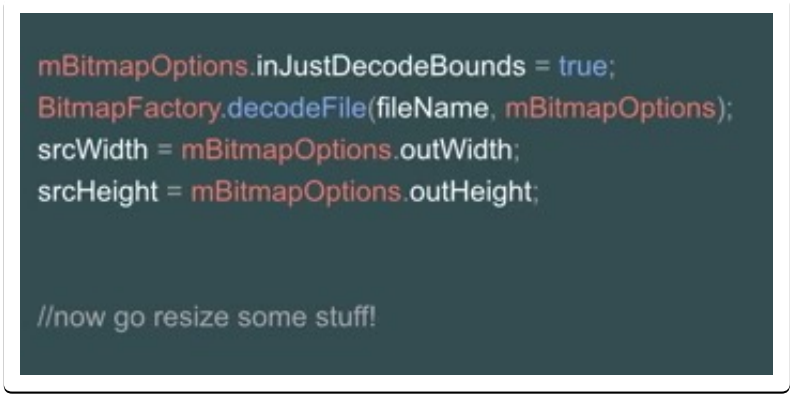
inSampleSize能够等比的缩放显示图片，同时还避免了需要先把原图加载进内存的缺点。我们会使用类似像下面一样的方法来缩放bitmap：

```
mBitmapOptions.inSampleSize = 4

// will load & resize the image to be 1/inSampleSize dimensions
mCurrentBitmap = BitmapFactory.decodeFile(fileName, mBitmapOptions);
```



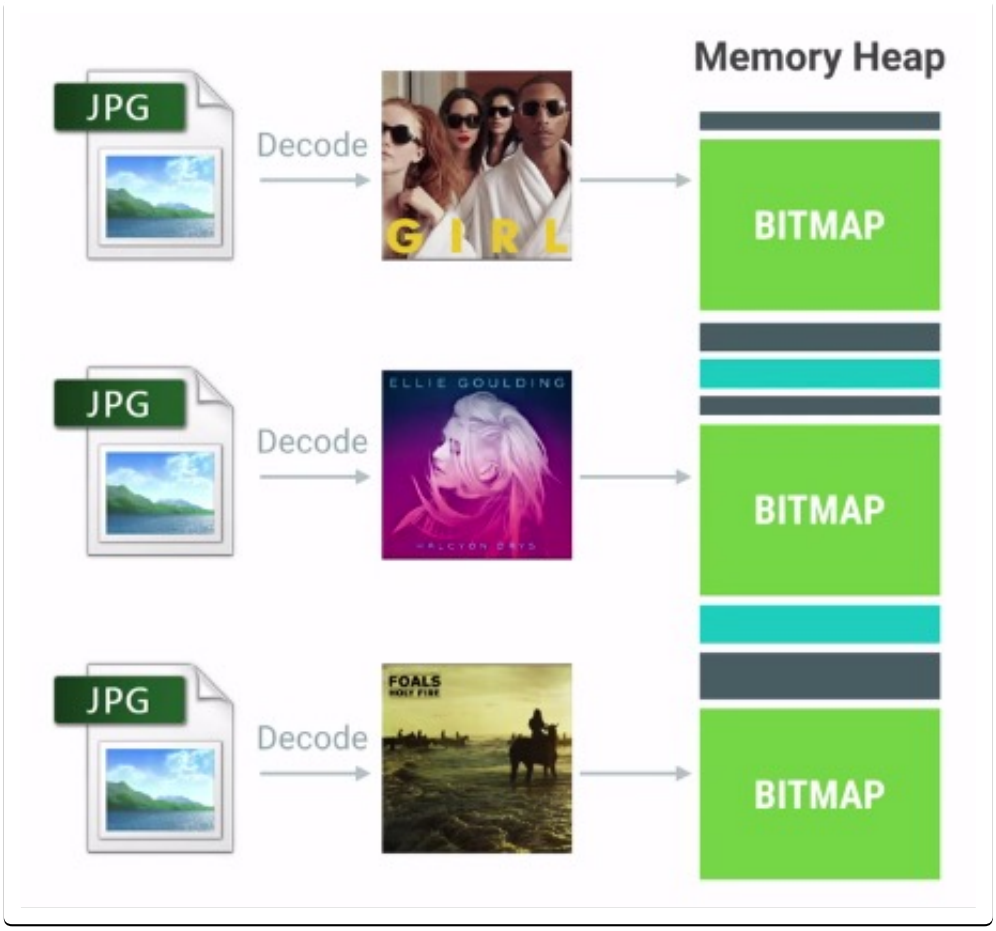
另外，我们还可以使用inScaled，inDensity，inTargetDensity的属性来对解码图片做处理，源码如下图所示：

还有一个经常使用到的技巧是inJustDecodeBounds，使用这个属性去尝试解码图片，可以事先获取到图片的大小而不至于占用什么内存。如下图所示：

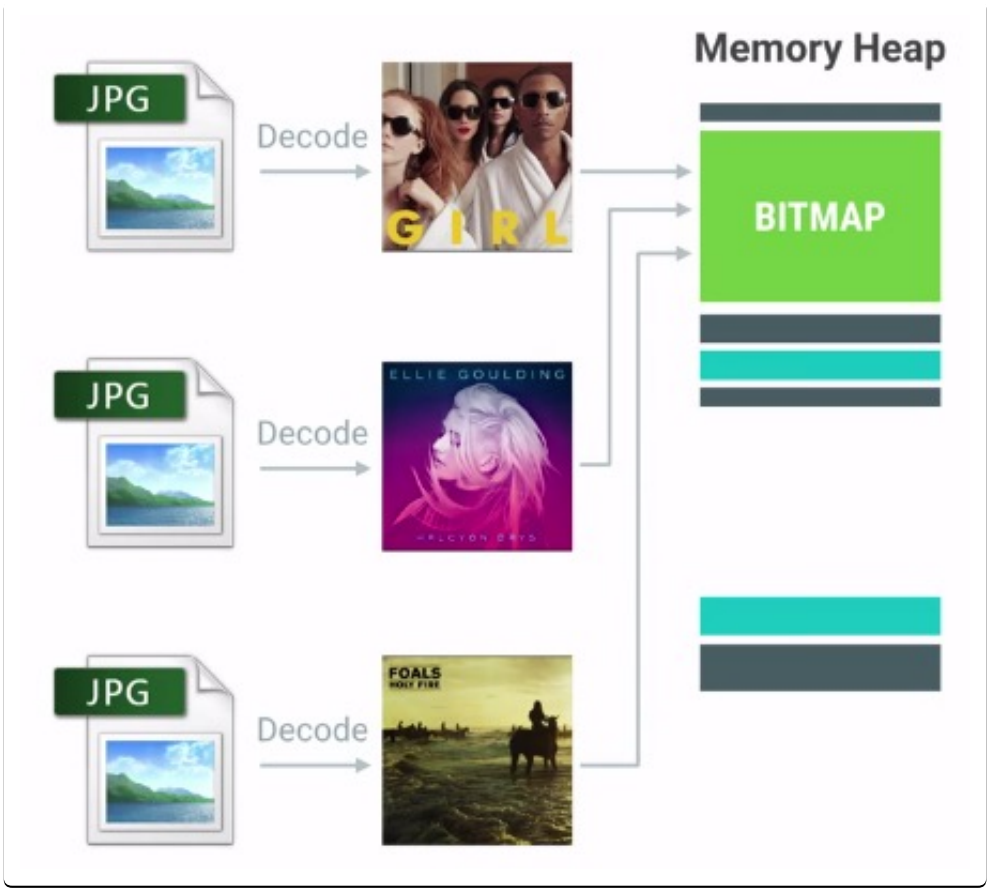


17)Re-using Bitmaps

我们知道bitmap会占用大量的内存空间，这节会讲解什么是inBitmap属性，如何利用这个属性来提升bitmap的循环效率。前面我们介绍过使用对象池的技术来解决对象频繁创建再回收的效率问题，使用这种方法，bitmap占用的内存空间会差不多是恒定的数值，每次新创建出来的bitmap都会需要占用一块单独的内存区域，如下图所示：



为了解决上图所示的效率问题，Android在解码图片的时候引进了**inBitmap**属性，使用这个属性可以得到下图所示的效果：



使用inBitmap属性可以告知Bitmap解码器去尝试使用已经存在的内存区域，新解码的bitmap会尝试去使用之前那张bitmap在heap中所占据的pixel data内存区域，而不是去问内存重新申请一块区域来存放bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小。下面是如何使用inBitmap的代码示例：

```
mBitmapOptions.inBitmap = mCurrentBitmap

// will reuse the mCurrentBitmap
// (or not, depending on if inBitmap is set)
mCurrentBitmap = BitmapFactory.decodeFile(fileName,
mBitmapOptions);
```

使用inBitmap需要注意几个限制条件：

- 在SDK 11 -> 18之间，重用的bitmap大小必须是一致的，例如给inBitmap赋值的图片大小为100-100，那么新申请的bitmap必须也为100-100才能够被重用。从SDK 19开始，新申请的bitmap大小必须小于或者等于已经赋值过的bitmap大小。
- 新申请的bitmap与旧的bitmap必须有相同的解码格式，例如大家都是8888的，如果前面的bitmap是8888，那么就不能支持4444与565格式的bitmap了。

我们可以创建一个包含多种典型可重用bitmap的对象池，这样后续的bitmap创建都能够找到合适的“模板”去进行重用。如下图所示：

Google介绍了一个开源的加载bitmap的库：Glide，这里面包含了各种对bitmap的优化技巧。

18)The Performance Lifecycle

大多数开发者在没有发现严重性能问题之前是不会特别花精力去关注性能优化的，通常大家关注的都是功能是否实现。当性能问题真的出现的时候，请不要慌乱。我们通常采用下面三个步骤来解决性能问题。

Gather：收集数据

我们可以通过Android SDK里面提供的诸多工具来收集CPU，GPU，内存，电量等等性能数据，

Insight：分析数据

通过上面的步骤，我们获取到了大量的数据，下一步就是分析这些数据。工具帮我们生成了很多可读性强的表格，我们需要事先了解如何查看表格的数据，每一项代表的含义，这样才能够快速定位问题。如果分析数据之后还是没有找到问题，那么就只能不停的重新收集数据，再进行分析，如此循环。

Action：解决问题

定位到问题之后，我们需要采取行动来解决问题。解决问题之前一定要先有个计划，评估这个解决方案是否可行，是否能够及时的解决问题。

19)Tools not Rules

虽然前面介绍了很多调试的方法，处理技巧，规范建议等等，可是这并不意味着所有的情况都适用，我们还是需要根据当时的情景做特定灵活的处理。

20)Memory Profiling 101

围绕Android生态系统，不仅仅有Phone，还有Wear，TV，Auto等等。对这些不同形态的程序进行性能优化，都离不开内存调试这个步骤。这节中介绍的内容大部分和Android性能优化典范与Android性能优化之内存篇重合，不展开了。

首发于CSDN：Android性能优化典范（二）