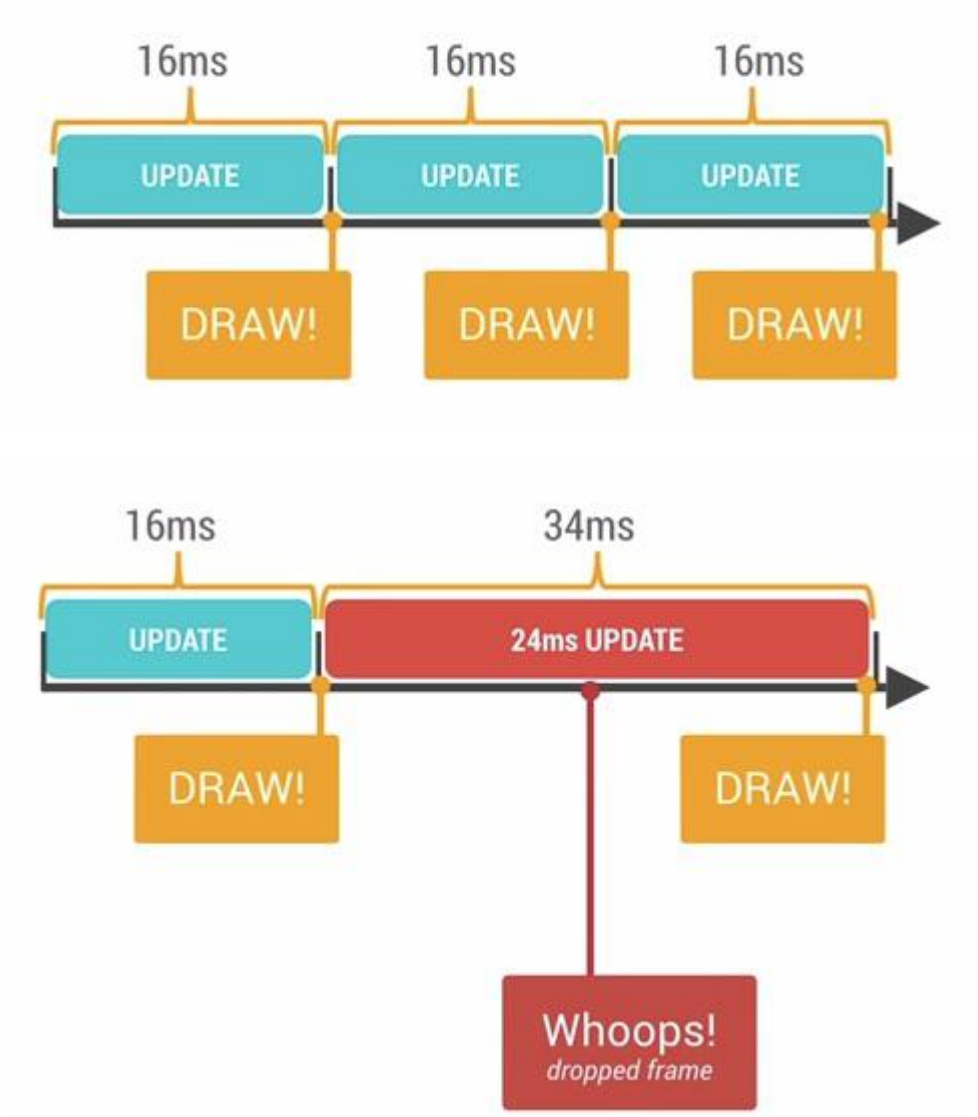
## 渲染性能

大多数用户感知到的卡顿等性能问题的最主要根源都是因为渲染性能。Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的 60fps，为了能够实现 60fps，这意味着程序的大多数操作都必须在 16ms 内完成。如果你的某个操作花费时间是 24ms，系统在得到 VSYNC 信号的时候就无法进行正常渲染，这样就发生了丢帧现象。那么用户在 32ms 内看到的会是同一帧画面。




有很多原因可以导致丢帧，也许是因为你的 layout 太过复杂，无法在 16ms 内完成渲染，有可能是因为你的 UI 上有层叠太多的绘制单元（过度绘制），还有可能是因为动画执行的次

数过多。这些都会导致 CPU 或者 GPU 负载过重。

为了理解 App 是如何进行渲染的，我们必须了解手机硬件是如何工作，那么就必须理解什么是 VSYNC。在讲解 VSYNC 之前，我们需要了解两个相关的概念：

1、Refresh Rate :代表了屏幕在一秒内刷新屏幕的次数 ,这取决于硬件的固定参数 ,例如 60Hz。

2、Frame Rate：代表了 GPU 在一秒内绘制操作的帧数，例如 30fps，60fps。

GPU 会获取图形数据进行渲染，然后硬件负责把渲染后的内容呈现到屏幕上，他们两者不停的进行协作。

通常来说，帧率超过刷新频率只是一种理想的状况，在超过 60fps 的情况下，GPU 所产生的帧数据会因为等待 VSYNC 的刷新信息而被 Hold 住，这样能够保持每次刷新都有实际的新的数据可以显示。但是我们遇到更多的情况是帧率小于刷新频率。如果发生帧率与刷新频率不一致的情况，就会容易出现 Tearing（断裂）的现象(画面上下两部分显示内容发生断裂，来自不同的两帧数据发生重叠)。

我们通常都会提到 60fps 与 16ms ,可是知道为何会是以程序是否达到 60fps 来作为 App 性能的衡量标准吗？这是因为人眼与大脑之间的协作无法感知超过 60fps 的画面更新。12fps 大概类似手动快速翻动书籍的帧率，这明显是可以感知到不够顺滑的。24fps 使得人眼感知的是连续线性的运动，这其实是归功于运动模糊的效果。24fps 是电影胶圈通常使用的帧率，因为这个帧率已经足够支撑大部分电影画面需要表达的内容，同时能够最大的减少费用支出。但是低于 30fps 是无法顺畅表现绚丽的画面内容的，此时就需要用到 60fps 来达到想要的效果，当然超过 60fps 是没有必要的。开发 app 的性能目标就是保持 60fps，这意味着每一帧你只有 16ms=1000/60 的时间来处理所有的任务。

## 内存性能

虽然 Android 有自动管理内存的机制，但是对内存的不恰当使用仍然容易引起严重的性能问题。在同一帧里面创建过多的对象是件需要特别引起注意的事情。

Android 系统里面有一个 Generational Heap Memory 的模型，系统会根据内存中不同的内存数据类型分别执行不同的 GC 操作。例如，最近刚分配的对象会放在 Young Generation 区域，这个区域的对象通常都是会快速被创建并且很快被销毁回收的，同时这个区域的 GC 操作速度也是比 Old Generation 区域的 GC 操作速度更快的。

除了速度差异之外，执行 GC 操作的时候，任何线程的任何操作都会需要暂停，等待 GC 操作完成之后，其他操作才能够继续运行（GC 操作在 2.3 以前是同步的，之后是并发）。

通常来说，单个的 GC 并不会占用太多时间，但是大量不停的 GC 操作则会显著占用帧间隔时间(16ms)。如果在帧间隔时间里面做了过多的 GC 操作，那么自然其他类似计算，渲染等操作的可用时间就变得少了。

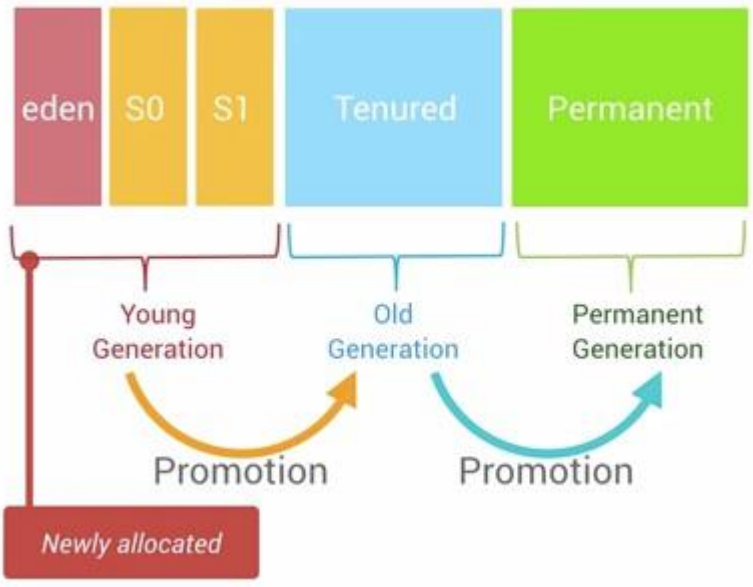
导致 GC 频繁执行有两个原因：

1、Memory Churn 内存抖动，内存抖动是因为大量的对象被创建又在短时间内马上被释放。
2、瞬间产生大量的对象会严重占用 Young Generation 的内存区域，当达到阀值，剩余空间不够的时候，也会触发 GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加 Heap 的压力，从而触发更多其他类型的 GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。

例如，你需要避免在 for 循环里面分配对象占用内存，需要尝试把对象的创建移到循环体

之外，自定义 View 中的 onDraw 方法也需要引起注意，每次屏幕发生绘制以及动画执行过程中，onDraw 方法都会被调用到，避免在 onDraw 方法里面执行复杂的操作，避免创建对象。对于那些无法避免需要创建对象的情况，我们可以考虑对象池模型，通过对象池来解决频繁创建与销毁的问题，但是这里需要注意结束使用之后，需要手动释放对象池中的对象。

原始 JVM 中的 GC 机制在 Android 中得到了很大程度上的优化。Android 里面是一个三级 Generation 的内存模型，最近分配的对象会存放在 Young Generation 区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到 Old Generation，最后到 Permanent Generation 区域。每一个级别的内存区域都有固定的大小，此后不断有新的对象被分配到此区域，当这些对象总的大小快达到这一级别内存区域的阀值时，会触发 GC 的操作，以便腾出空间来存放其他新的对象。




前面提到过每次 GC 发生的时候，所有的线程都是暂停状态的。GC 所占用的时间和它是哪一个 Generation 也有关系，Young Generation 的每次 GC 操作时间是最短的，Old Generation 其次，Permanent Generation 最长。执行时间的长短也和当前 Generation 中的对象数量有关，遍历查找 20000 个对象比起遍历 50 个对象自然是要慢很多的。

虽然 Java 有自动回收的机制，可是这不意味着 Java 中不存在内存泄漏的问题，而内存泄漏会很容易导致严重的性能问题。

内存泄漏指的是那些程序不再使用的对象无法被 GC 识别，这样就导致这个对象一直留在内存当中，占用了宝贵的内存空间。显然，这还使得每级 Generation 的内存区域可用空间变小，GC 就会更容易被触发，从而引起性能问题。

## 工具

为了寻找内存的性能问题，Android Studio 提供了工具来帮助开发者。

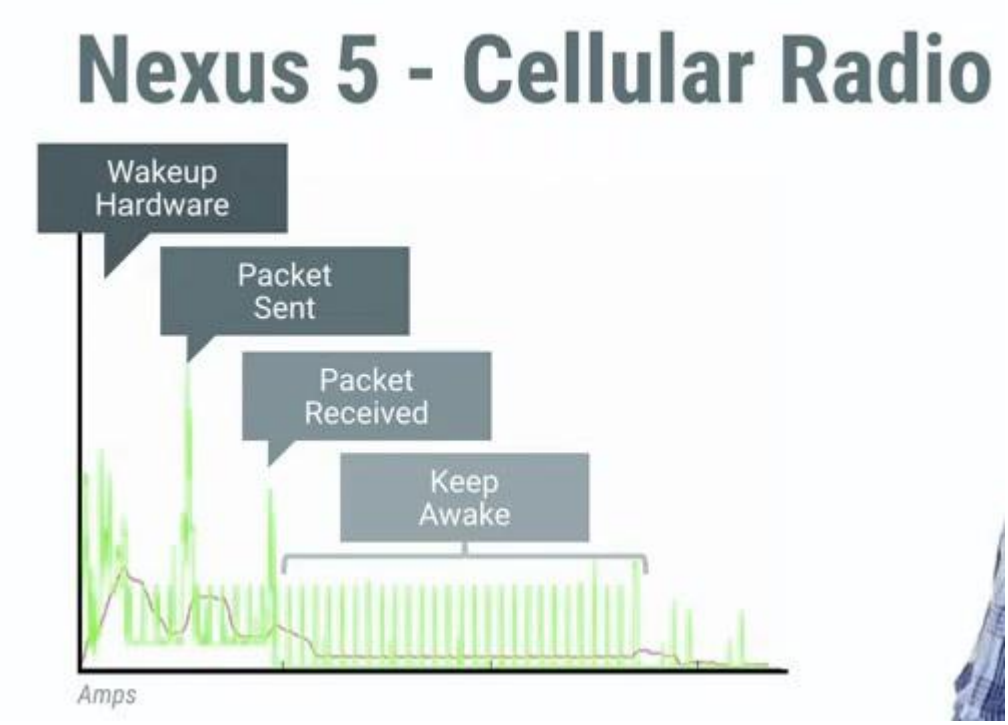
1、Memory Monitor：查看整个 app 所占用的内存，以及发生 GC 的时刻，短时间内发生大量的 GC 操作是一个危险的信号。

2、Allocation Tracker：使用此工具来追踪内存的分配，前面有提到过。

3、Heap Tool：查看当前内存快照，便于对比分析哪些对象有可能是泄漏了的，请参考前面的 Case。

## 电量优化

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。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，我们可以参考在 WiFi，4G，3G 等不同的网络下设计不同大小的预取数据量，也可以是按照图片数量或者操作时间来作为阀值。这需要我们需要根据特定的场景，不同的网络情况设计合适的方案。

## 压缩传输数据

关于压缩传输数据，我们可以学习以下的一些课程：

1、CompressorHead：这系列的课程会介绍压缩的基本概念以及一些常见的压缩算法知识。
2、Image Compression：介绍关于图片的压缩知识。
3、Gzip is not enough
4、Text Compression
5、FlatBuffers

执行延迟任务，通常有下面三种方式：

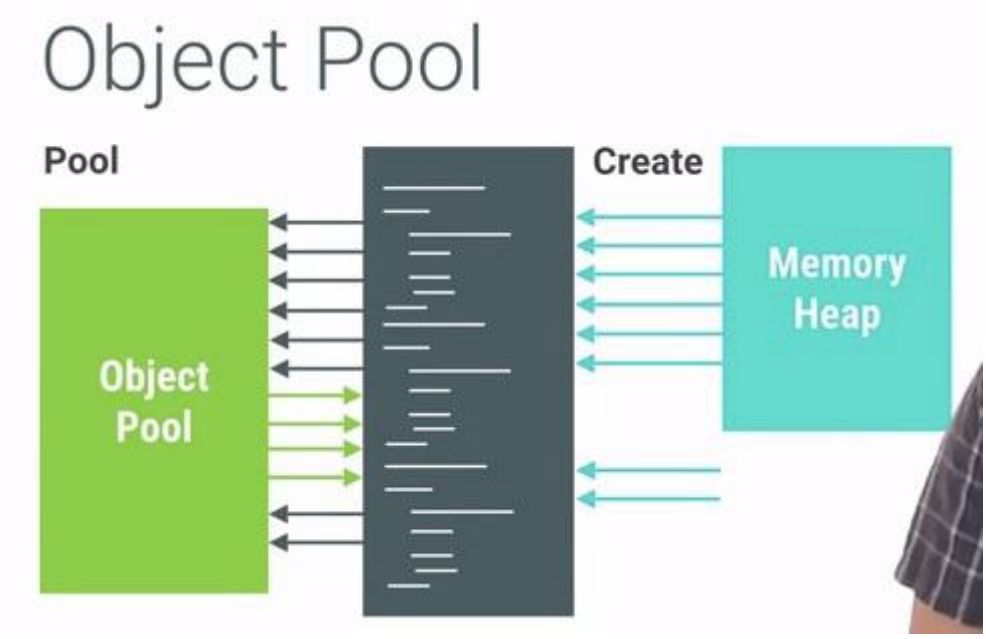
1、AlarmManager：使用 AlarmManager 设置定时任务，可以选择精确的间隔时间，也可以选择非精确时间作为参数。除非程序有很强烈的需要使用精确的定时唤醒，否者一定要避免使用他，我们应该尽量使用非精确的方式。

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。

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。

## 对象池（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。

## 遍历

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。

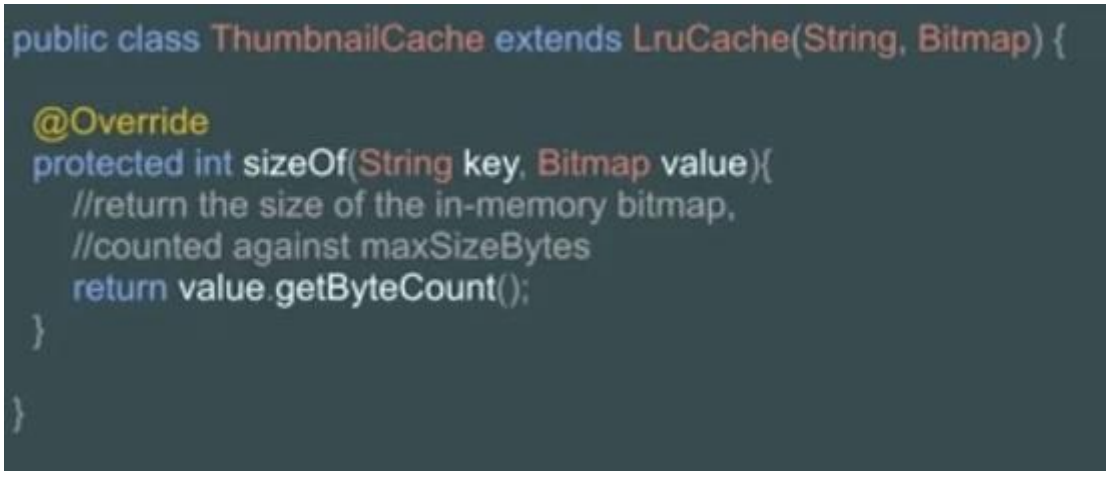
LRU(Least Recently Use)算法

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：



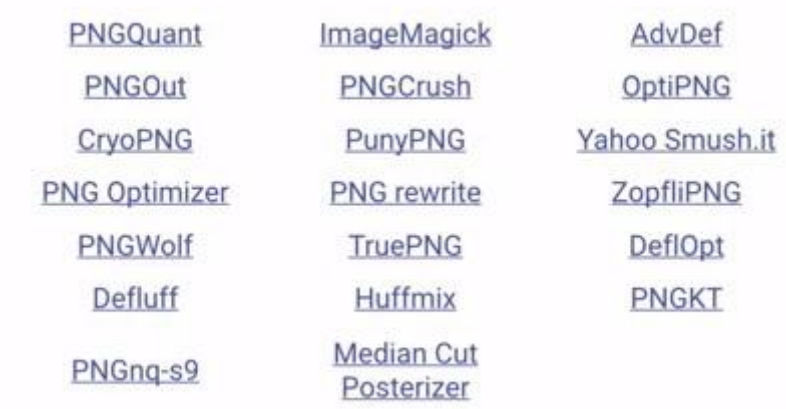
使用 LRU Cache 能够显著提升应用的性能，可是也需要注意 LRU Cache 中被淘汰对象的回收，否者会引起严重的内存泄露。

## 图片处理

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。

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：



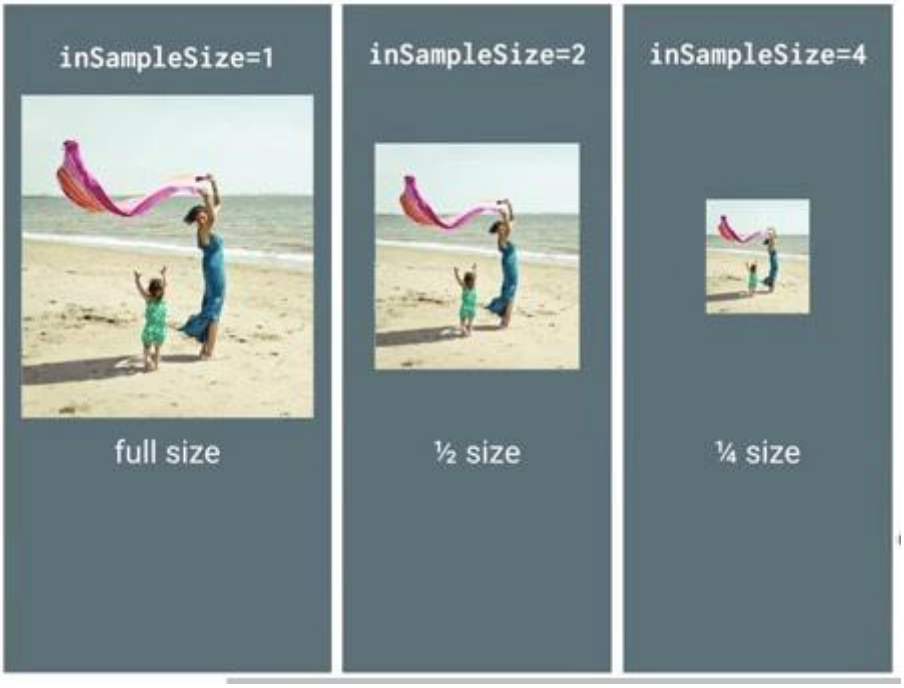
这里要介绍一种新的图片格式：Webp，它是由 Google 推出的一种既保留 png 格式的优点，又能够减少图片大小的一种新型图片格式。

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。

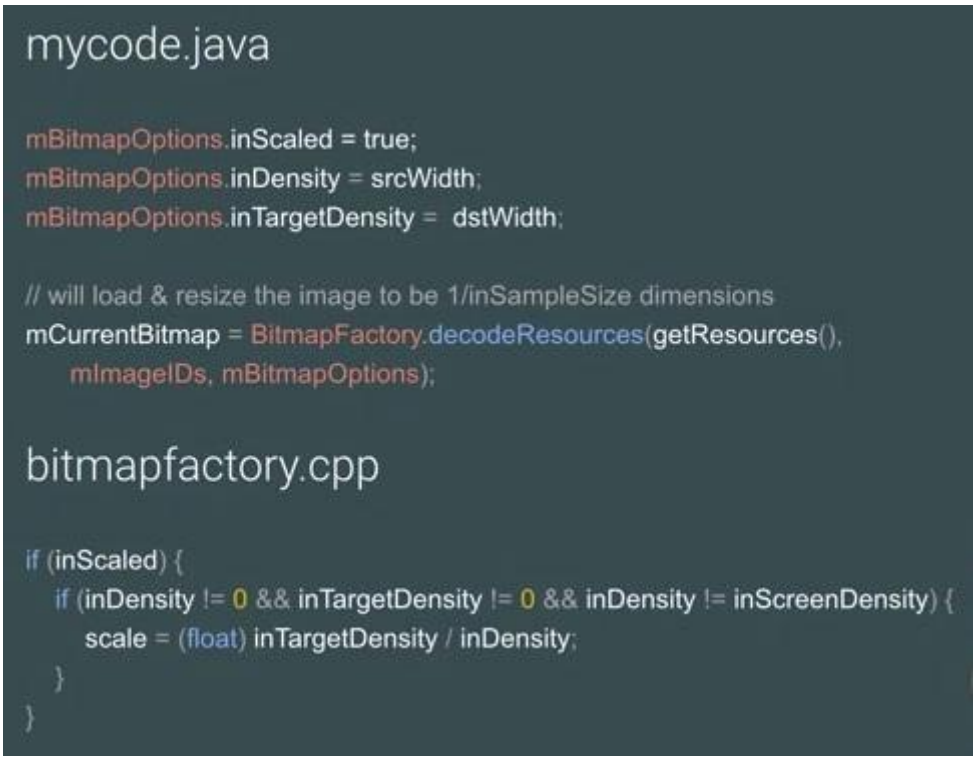

上面的方法能够快速的得到一张经过缩放的图片，可是这个方法能够执行的前提是，原图片需要事先加载到内存中，如果原图片过大，很可能导致 OOM。下面介绍其他几种缩放图片的方式。

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);
```

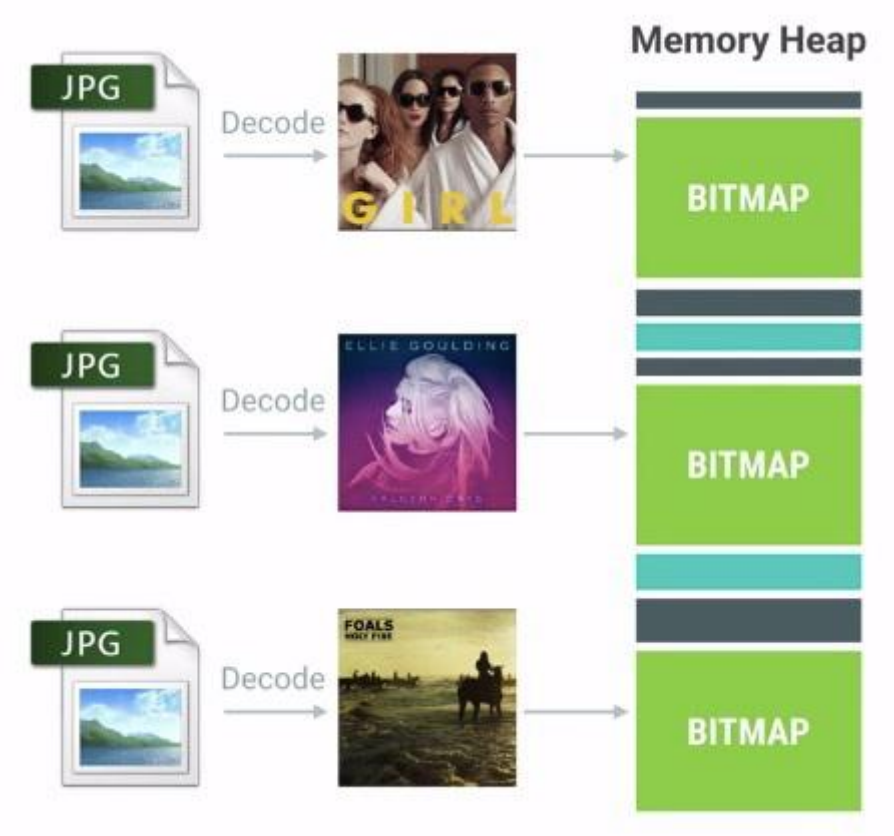



另外，我们还可以使用 inScaled，inDensity，inTargetDensity 的属性来对解码图片做处理，源码如下图所示：

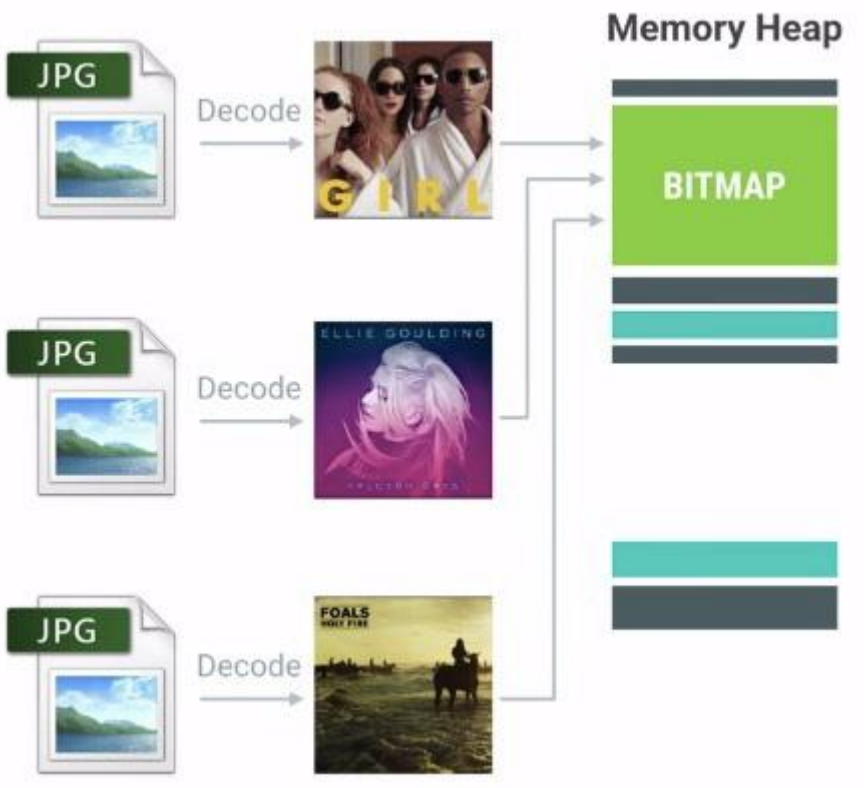



还有一个经常使用到的技巧是 inJustDecodeBounds，使用这个属性去尝试解码图片，可以事先获取到图片的大小而不至于占用什么内存。如下图所示：


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：

1、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。

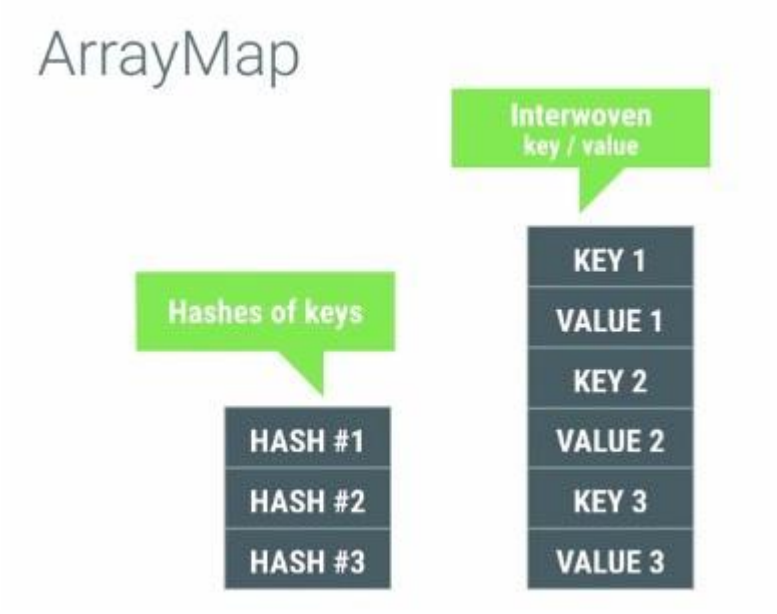
2、新申请的 bitmap 与旧的 bitmap 必须有相同的解码格式，例如大家都是 8888 的，如果前面的 bitmap 是 8888，那么就不能支持 4444 与 565 格式的 bitmap 了。

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：

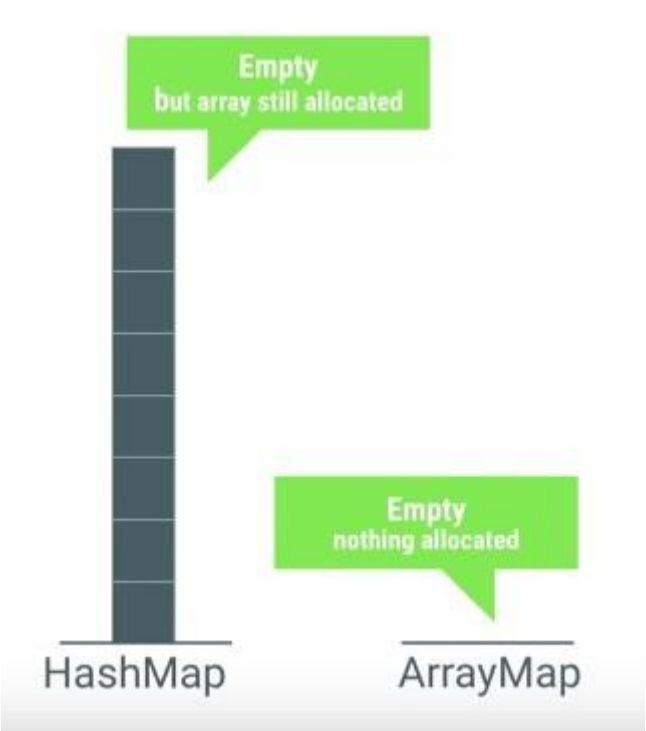

Google 介绍了一个开源的加载 bitmap 的库：Glide，这里面包含了各种对 bitmap 的优化技巧。

ArrayMap

为了解决 HashMap 更占内存的弊端，Android 提供了内存效率更高的 ArrayMap。它内部使用两个数组进行工作，其中一个数组记录 key hash 过后的顺序列表，另外一个数组按 key 的顺序记录 Key-Value 值，如下图所示：




当你想获取某个 value 的时候，ArrayMap 会计算输入 key 转换过后的 hash 值，然后对 hash 数组使用二分查找法寻找到对应的 index，然后我们可以通过这个 index 在另外一个数组中直接访问到需要的键值对。如果在第二个数组键值对中的 key 和前面输入的查询 key 不一致，那么就认为是发生了碰撞冲突。为了解决这个问题，我们会以该 key 为中心点，分别上下展开，逐个去对比查找，直到找到匹配的值。ArrayMap 的插入与删除的效率是不够高的，但是如果数组的列表只是在一百这个数量级上，则完全不用担心这些插入与删除的效率问题。HashMap 与 ArrayMap 之间的内存占用效率对比图如下：




与 HashMap 相比，ArrayMap 在循环遍历的时候也更加简单高效，如下图所示：

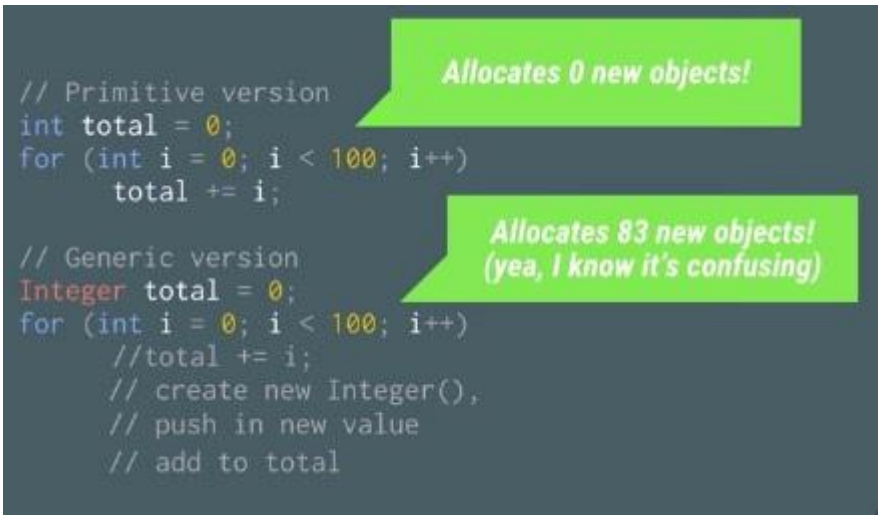
```
// ArrayMap
for(int i = 0; i < map.size(); i++){
    Object keyObj = map.keyAt(i)
    Object valObj = map.valueAt(i)
     ...
}



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

前面演示了很多 ArrayMap 的优点，但并不是所有情况下都适合使用 ArrayMap，我们应该在满足下面 2 个条件的时候才考虑使用 ArrayMap：

1、对象个数的数量级最好是千以内；

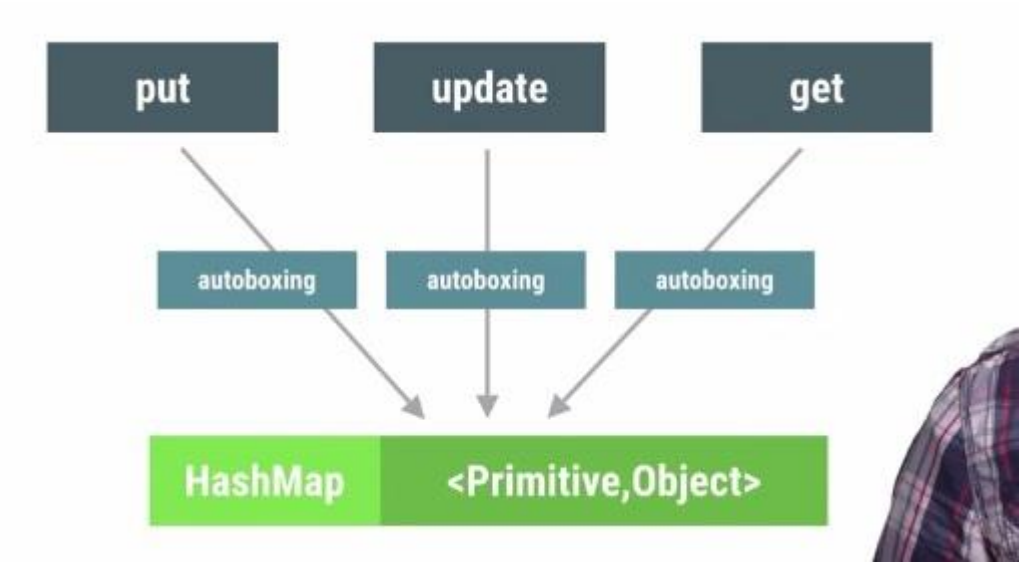
2、数据组织形式包含 Map 结构。

有时候性能问题也可能是因为那些不起眼的小细节引起的 ,例如在代码中不经意的“自动装箱”。我们知道基础数据类型的大小 :boolean(8 bits), int(32 bits), float(32 bits) ,long(64 bits)，为了能够让这些基础数据类型在大多数 Java 容器中运作，会需要做一个 autoboxing 的操作，转换成 Boolean，Integer，Float 等对象，如下演示了循环操作的时候是否发生 autoboxing 行为的差异：





Autoboxing 的行为还经常发生在类似 HashMap 这样的容器里面 ,对 HashMap 的增删改查操作都会发生了大量的 autoboxing 的行为。




为了避免 HashMap 的 autoboxing 行为，Android 系统提供了 SparseBoolMap，SparseIntMap，SparseLongMap，LongSparseMap 等容器。

## 枚举

使用 enum 之后的 dex 大小是 4188 bytes，相比起 2556 增加了 1632 bytes，增长量是使用 static int 的 13 倍。不仅仅如此，使用 enum，运行时还会产生额外的内存占用，Android 官方强烈建议不要在 Android 程序里面使用到 enum。

Android 系统的一大特色是多任务，用户可以随意在不同的 app 之间进行快速切换。为了确保你的应用在这种复杂的多任务环境中正常运行，我们需要了解下面的知识。

为了让 background 的应用能够迅速的切换到 forground，每一个 background 的应用都会占用一定的内存。Android 系统会根据当前的系统内存使用情况，决定回收部分 background 的应用内存。如果 background 的应用从暂停状态直接被恢复到 forground，

能够获得较快的恢复体验，如果 background 应用是从 Kill 的状态进行恢复，就会显得稍微有点慢。

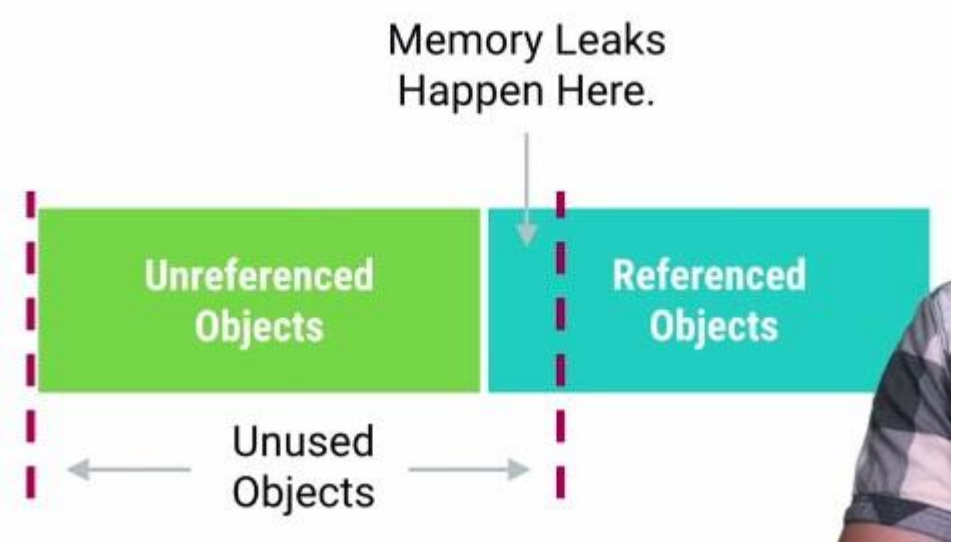
Android 系统提供了一些回调来通知应用的内存使用情况，通常来说，当所有的 background 应用都被 kill 掉的时候，forground 应用会收到 onLowMemory()的回调。在这种情况下，需要尽快释放当前应用的非必须内存资源，从而确保系统能够稳定继续运行。Android 系统还提供了 onTrimMemory()的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调，同时在这个回调里面会传递以下的参数，代表不同的内存使用情况，下图介绍了各种不同的回调参数：

| | |
|---|---|
| TRIM_MEMORY_RUNNING_MODERATE | this is your first warning |
| TRIM_MEMORY_RUNNING_MODERATE | This is like the yellow light. It is your second warning to begin to trim resources to improve performance. |
| TRIM_MEMORY_RUNNING_CRITICAL | This is the red light. If you keep on executing without clearing up memory resource, the system is going to begin killing background processes to get more memory for you. Unfortunately that will lower the performance of your application. |
| TRIM_MEMORY_UI_HIDDEN | Your application was just moved off the screen, so this is a good time to release large UI resources. Now your application is on the list of cached applications. If there are memory problems, your process may be killed |
| TRIM_MEMORY_BACKGROUND | Being a background app - release as much as you can so that your app can resume faster than a pure restart |
| TRIM_MEMORY_BACKGROUND | You are a background app, but near the end of the list |
| TRIM_MEMORY_MODERATE | You are a background app, but in the middle |
| TRIM_MEMORY_COMPLETE | You are a background app, but about to be killed |

另外 onTrimMemory()的回调可以发生在 Application，Activity，Fragment，Service，Content Provider。从 Android 4.4 开始，ActivityManager 提供了 isLowRamDevice()的 API，通常指的是 Heap Size 低于 512M 或者屏幕大小<=800*480 的设备。

## 内存泄漏

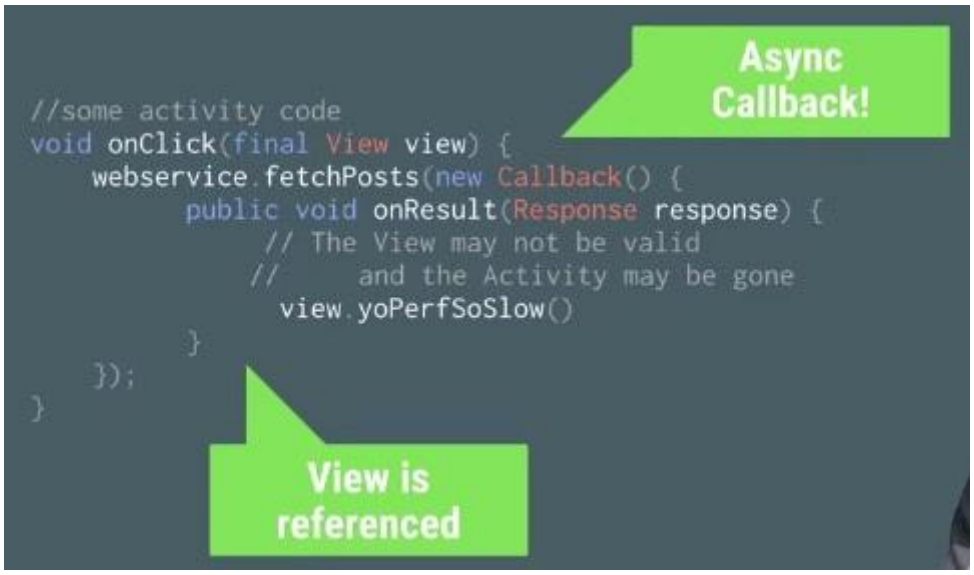
内存泄漏的概念，下面一张图演示下：




通常来说，View 会保持 Activity 的引用，Activity 同时还和其他内部对象也有可能保持引用关系。当屏幕发生旋转的时候，activity 很容易发生泄漏，这样的话，里面的 view 也会发生泄漏。Activity 以及 view 的泄漏是非常严重的，为了避免出现泄漏，请特别留意以下的规则：

### 6.1) 避免使用异步回调

异步回调被执行的时间不确定，很有可能发生在 activity 已经被销毁之后，这不仅仅很容易引起 crash，还很容易发生内存泄露。




### 6.2) 避免使用 Static 对象

因为 static 的生命周期过长，使用不当很可能导致 leak，在 Android 中应该尽量避免使用 static 对象。


### 6.3) 避免把 View 添加到没有清除机制的容器里面

假如把 view 添加到 WeekHashMap，如果没有执行清除操作，很可能会导致泄漏。

## 网络缓存

Android 系统上关于网络请求的 Http Response Cache 是默认关闭的，这样会导致每次即使请求的数据内容是一样的也会需要重复被调用执行，效率低下。我们可以通过下面的代码示例开启 HttpResponseCache。

```
protected void onCreate(Bundle savedInstanceState) {
    ...

    try {
        File httpCacheDir = new File(context.getCacheDir(), "http");
        long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
        HttpResponseCache.install(httpCacheDir, httpCacheSize);
    } catch (IOException e) {
        Log.i(TAG, "HTTP response cache installation failed:" + e);
    }
}

protected void onStop() {
    ...

    HttpResponseCache cache = HttpResponseCache.getInstalled();
    if (cache != null) {
        cache.flush();
    }
}}
```

开启 Http Response Cache 之后，Http 操作相关的返回数据就会缓存到文件系统上，不仅仅是主程序自己编写的网络请求相关的数据会被缓存，另外引入的 library 库中的网络相关的请求数据也会被缓存到这个 Cache 中。

实现自定义的 http 缓存，需要解决两个问题：第一个是实现一个 DiskCacheManager，另外一个是制定 Cache 的缓存策略。关于 DiskCacheManager，我们可以扩展 Android 系统提供的 DiskLruCache 来实现。而 Cache 的缓存策略，相对来说复杂一些，我们可能需要把部分 JSON 数据设计成不能缓存的，另外一些 JSON 数据设计成可以缓存几天的，把缩略图设计成缓存一两天的等等，为不同的数据类型根据他们的使用特点制定不同的缓存策略。想要比较好的实现这两件事情，如果全部自己从头开始写会比较繁琐复杂，所幸的是，有不少著名的开源框架帮助我们快速的解决了那些问题。我们可以使用 Volly、okHTTP、Picasso 来实现网络缓存。

## 去除无用的代码

Android 为我们提供了 Proguard 的工具来帮助应用程序对代码进行瘦身，优化，混淆的处理。它会帮助移除那些没有使用到的代码，还可以对类名，方法名进行混淆处理以避免程序被反编译。举个例子，Google I/O 2015 这个应用使用了大量的 library，没有经过 Proguard 处理之前编译出来的包是 8.4Mb 大小，经过处理之后的包仅仅是 4.1Mb 大小。

使用 Proguard 相当的简单，只需要在 build.gradle 文件中配置 minifEnable 为 true 即可

```
android {
    ...
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                                 'proguard-rules.pro'
        }
    }
}
```

## 去除无用资源文件

减少 APK 安装包的大小也是 Android 程序优化中很重要的一个方面，我们不应该给用户下载到一个臃肿的安装包。假设这样一个场景，我们引入了 Google Play Service 的 library，是想要使用里面的 Maps 的功能，但是里面的登入等等其他功能是不需要的，可是这些功能相关的代码与图片资源，布局资源如果也被引入我们的项目，这样就会导致我们的程序安装包臃肿。

所幸的是，我们可以使用 Gradle 来帮助我们分析代码，分析引用的资源，对于那些没有被引用到的资源，会在编译阶段被排除在 APK 安装包之外，要实现这个功能，对我们来说仅仅只需要在 build.gradle 文件中配置 shrinkResource 为 true 就好了，如下图所示：

```
android {
    ...

    buildTypes {
        release {
            minifyEnabled true
            shrinkResources true
            proguardFiles
getDefaultProguardFile('proguard-android.txt'),
                          'proguard-rules.pro'
        }
    }
}
```

Gradle 目前无法对 values，drawable 等根据运行时来决定使用的资源进行优化，对于这些资源，需要我们自己来确保资源不会有冗余。

## 数据序列化

数据的序列化是程序代码里面必不可少的组成部分，当我们讨论到数据序列化的性能的时候，需要了解有哪些候选的方案，他们各自的优缺点是什么。通常情况下，我们会把那些需要序列化的类实现 Serializable 接口(如下图所示)，但是这种传统的做法效率不高，实施的过程会消耗更多的内存。

```
class Rectangle implements
Serializable {
    int width, height;

    public void set_values (int,int);
    public int area (void);
}
```


但是我们如果使用 GSON 库来处理这个序列化的问题，不仅仅执行速度更快，内存的使用效率也更高。Android 的 XML 布局文件会在编译的阶段被转换成更加复杂的格式，具备更加高效的执行性能与更高的内存使用效率。

下面介绍三个数据序列化的候选方案：

- **Protocal Buffers**：强大，灵活，但是对内存的消耗会比较大，并不是移动终端上的最佳选择。
- **Nano-Proto-Buffers**：基于 Protocal，为移动终端做了特殊的优化，代码执行效率更高，内存使用效率更佳。
- **FlatBuffers**：这个开源库最开始是由 Google 研发的，专注于提供更优秀的性能。

## 多线程

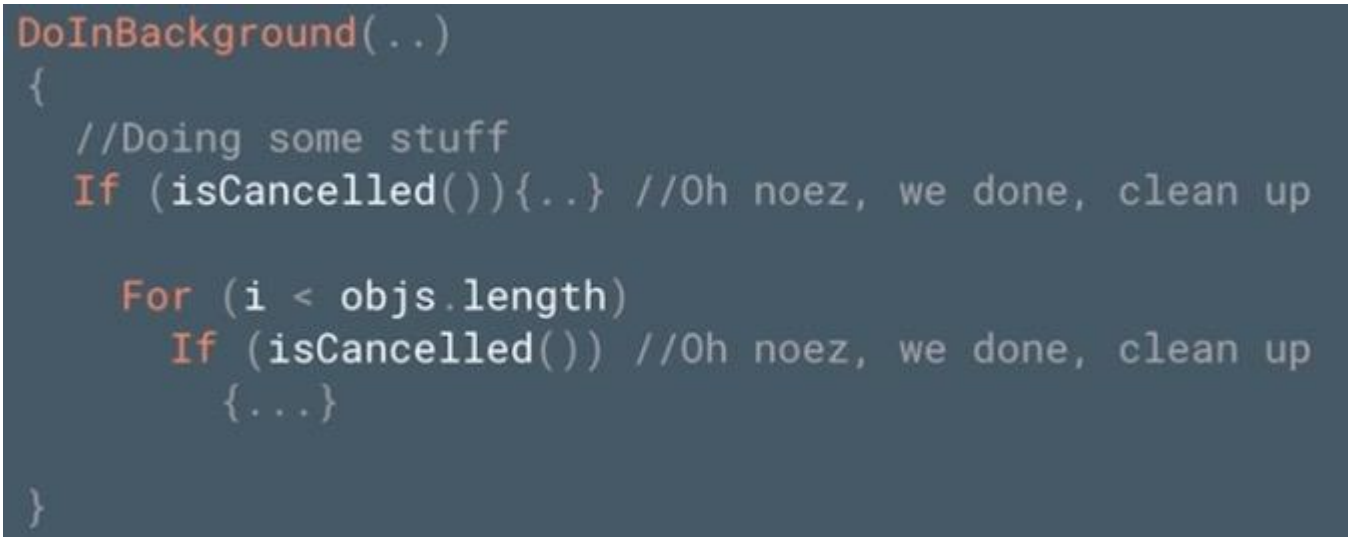
Android 系统为我们提供了若干组工具类来帮助解决多线程问题。

- **AsyncTask**: 为 UI 线程与工作线程之间进行快速的切换提供一种简单便捷的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的使用场景。
- **HandlerThread**: 为某些回调方法或者等待某些任务的执行设置一个专属的线程，并提供线程任务的调度机制。
- **ThreadPool**: 把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。
- **IntentService**: 适合于执行由 UI 触发的后台 Service 任务，并可以把后台任务执行的情况通过一定的机制反馈给 UI。

使用 AsyncTask 需要注意的问题有哪些呢？请关注以下几点：

1、首先，默认情况下，所有的 AsyncTask 任务都是被线性调度执行的，他们处在同一个任务队列当中，按顺序逐个执行。假设你按照顺序启动 20 个 AsyncTask，一旦其中的某个 AsyncTask 执行时间过长，队列中的其他剩余 AsyncTask 都处于阻塞状态，必须等到该任务执行完毕之后才能够有机会执行下一个任务。为了解决线性队列等待的问题，我们可以使用 *AsyncTask.executeOnExecutor()*强制指定 AsyncTask 使用线程池并发调度任务。

2、其次，如何才能够真正的取消一个 AsyncTask 的执行呢？我们知道 AsyncTaks 有提供 *cancel()*的方法，但是这个方法实际上做了什么事情呢？线程本身并不具备中止正在执行的代码的能力，为了能够让一个线程更早的被销毁，我们需要在 *doInBackground()*的代码中不断的添加程序是否被中止的判断逻辑，如下图所示：




一旦任务被成功中止，AsyncTask 就不会继续调用 *onPostExecute()*，而是通过调用 *onCancelled()*的回调方法反馈任务执行取消的结果。我们可以根据任务回调到哪个方法（是 onPostExecute 还是 onCancelled )来决定是对 UI 进行正常的更新还是把对应的任务所占用的内存进行销毁等。

3、最后，使用 AsyncTask 很容易导致内存泄漏，一旦把 AsyncTask 写成 Activity 的内部类的形式就很容易因为 AsyncTask 生命周期的不确定而导致 Activity 发生泄漏。

综上所述，AsyncTask 虽然提供了一种简单便捷的异步机制，但是我们还是很有必要特别关注到他的缺点，避免出现因为使用错误而导致的严重系统性能问题。

**使用 IntentService 需要特别留意以下几点：**

1、首先，因为 IntentService 内置的是 HandlerThread 作为异步线程，所以每一个交给 IntentService 的任务都将以队列的方式逐个被执行到，一旦队列中有某个任务执行时间过长，那么就会导致后续的任务都会被延迟处理。

2、2、其次，通常使用到 IntentService 的时候，我们会结合使用 BroadcastReceiver 把工作线程的任务执行结果返回给主 UI 线程。使用广播容易引起性能问题，我们可以使用 LocalBroadcastManager 来发送只在程序内部传递的广播，从而提升广播的性能。我们也可以使用 *runOnUiThread()*快速回调到主 UI 线程。

3、最后，包含正在运行的 IntentService 的程序相比起纯粹的后台程序更不容易被系统杀死，该程序的优先级是介于前台程序与纯后台程序之间的。

## 启动闪屏

对于启动闪屏，正确的使用方法是自定义一张图片，把这张图片通过设置主题的方式显示为启动闪屏，代码执行到主页面的 onCreate 的时候设置为程序正常的主题。

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android"
android:opacity="opaque">
  <!-- The background color, preferably the same as your normal theme -
  <item android:drawable="@android:color/white"/>
  <!-- Your product logo - 144dp color version of your app icon -->
  <item>
    <bitmap
      android:src="@drawable/product_logo_144dp"
      android:gravity="center"/>
  </item>
</layer-list>



<activity ... android:theme="@style/AppTheme.Launcher" />
```

```
public class MyMainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // Make sure this is before calling super.onCreate
    setTheme(R.style.Theme_MyApp);
    super.onCreate(savedInstanceState);
    // ...
  }
}
```

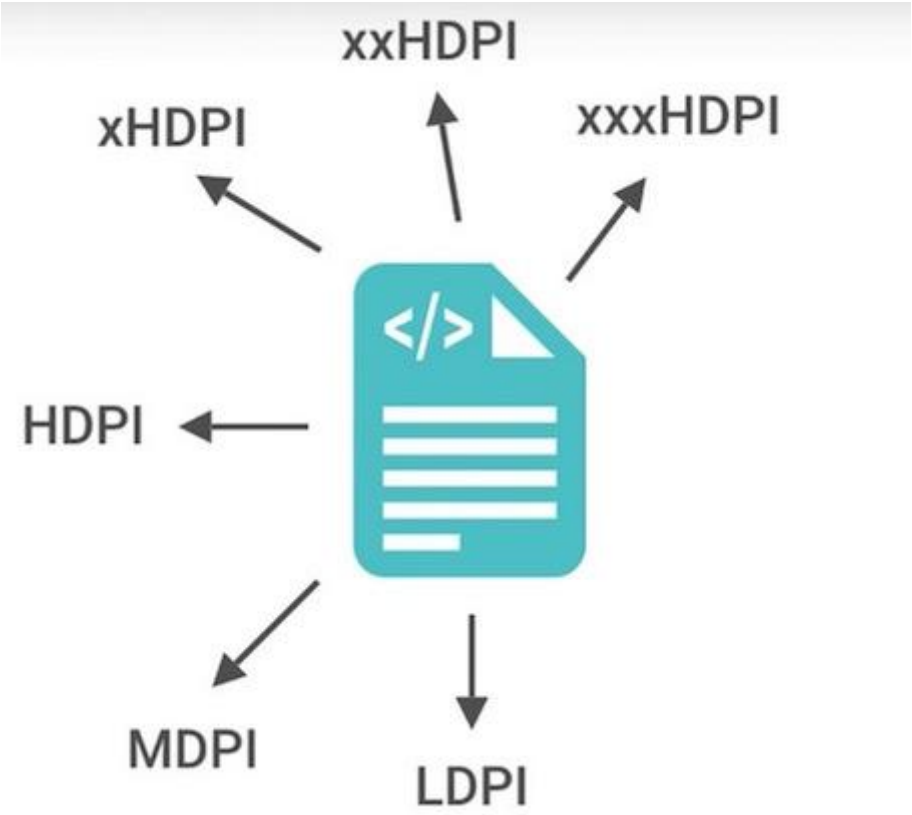
## 减少程序图片资源的大小

1. 确保在 build.gradle 文件中开启了 *minifEnabled* 与 *shrinkResources* 的属性，这两个属性可以帮助移除那些在程序中使用不到的代码与资源，帮助减少 App 的安装包大小。

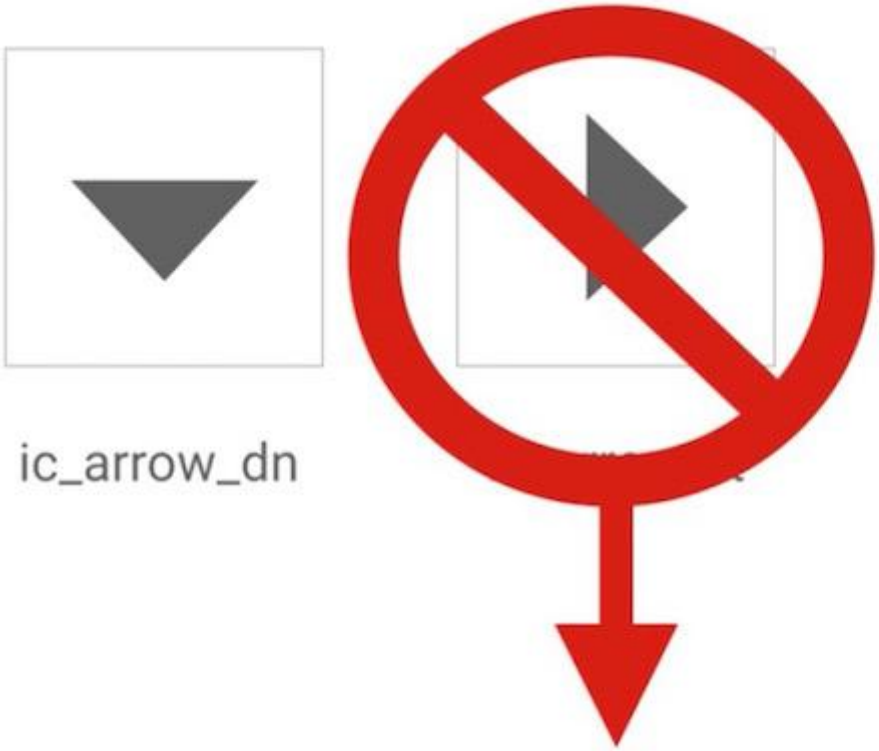
```
buildTypes {
  release {
    minifyEnabled true
    shrinkResources true
    proguardFiles getDefaultProguardFile('proguard-android.t
                  'proguard-rules.pro'
  }
}
```

2. 有选择性的提供对应分辨率的图片资源，系统会自动匹配最合适分辨率的图片并执行拉伸或者压缩的处理。

3. 在符合条件的情况下，使用 Vertor Drawable 替代传统的 PNG/JPEG 图片，能够极大地减少图片资源的大小。传统模式下，针对不同 dpi 的手机都需要提供一套 PNG/JPEG 的图片，而如果使用 Vector Drawable 的话，只需要一个 XML 文件即可。




4. 尽量复用已经存在的资源图片，使用代码的方式对已有的资源进行复用，如下图所示：




```xml
<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/
android"
    android:drawable="@drawable/ic_arrow_expand"
    android:fromDegrees="0"
    android:toDegrees="270" />
```

以上几点虽然看起来都微不足道，但是真正执行之后，能够显著减少安装包的资源图片大小。

## 减少程序的代码量

1、开启 MinifEnabled、Proguard。打开这些编译属性之后，程序在打包的时候就不会把没有引用到的代码编译进来，以此达到减少安装包大小的目的。

2、注意因为编译行为额外产生的方法数，例如类似 Enum、Protocal Buffer 可能导致方法数与类的个数增加。

3、部分引入到工程中的 jar 类库可能并不是专门针对移动端 App 而设计的，它们最开始可能是运用在 PC 或者 Server 上的。使用这些类库不仅仅额外增加了包的大小，还增加了编译时间。单纯依靠 Proguard 可能无法完全移除那些使用不到的方法，最佳的方式是使用一些更加轻量化，专门为 Android App 设计的 jar 类库。

## 安装包的拆分

设想一下，一个 low dpi，API<14 的用户手机下载安装的 APK 里面却包含了大量 xxhdpi 的资源文件，对于这个用户来说，这个 APK 是存在很大的资源浪费的。幸好 Android 平台为我们提供了拆分 APK 的方法，它能够根据 API Level、屏幕大小以及 GPU 版本的不同进行拆分，使得对应平台的用户下载到最合适自己手机的安装包。（国内不可行）