SEP 12TH, 2015 | COMMENTS

Android内存优化之OOM

Android的内存优化是性能优化中很重要的一部分，而避免OOM又是内存优化中比较核心的一点，这是一篇关于内存优化中如何避免OOM的总结性概要文章，内容大多都是和OOM有关的实践总结概要。理解错误或是偏差的地方，还请多包涵指正，谢谢！

(一)Android的内存管理机制

Google在Android的官网上有这样一篇文章，初步介绍了Android是如何管理应用的进程与内存分配：http://developer.android.com/training/articles/memory.html。 Android系统的Dalvik虚拟机扮演了常规的内存垃圾自动回收的角色，Android系统没有为内存提供交换区，它使用paging与memory-mapping(mmapping)的机制来管理内存，下面简要概述一些Android系统中重要的内存管理基础概念。

1）共享内存

Android系统通过下面几种方式来实现共享内存：

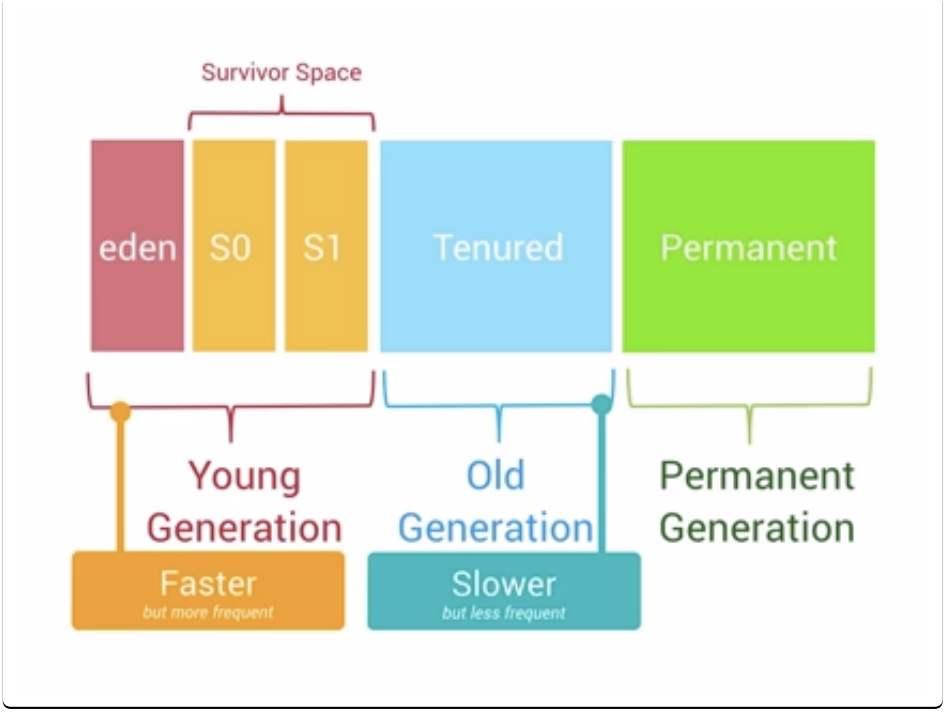
- Android应用的进程都是从一个叫做Zygote的进程fork出来的。Zygote进程在系统启动并且载入通用的framework的代码与资源之后开始启动。为了启动一个新的程序进程，系统会fork Zygote进程生成一个新的进程，然后在新的进程中加载并运行应用程序的代码。这使得大多数的RAM pages被用来分配给framework的代码，同时使得RAM资源能够在应用的所有进程之间进行共享。
- 大多数static的数据被mmapped到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享，而且使得它能够在需要的时候被paged out。常见的static数据包括Dalvik Code，app resources，so文件等。
- 大多数情况下，Android通过显式的分配共享内存区域(例如ashmem或者gralloc)来实现动态RAM区域能够在不同进程之间进行共享的机制。例如，Window Surface在App与Screen Compositor之间使用共享的内存，Cursor Buffers在Content Provider与Clients之间共享内存。

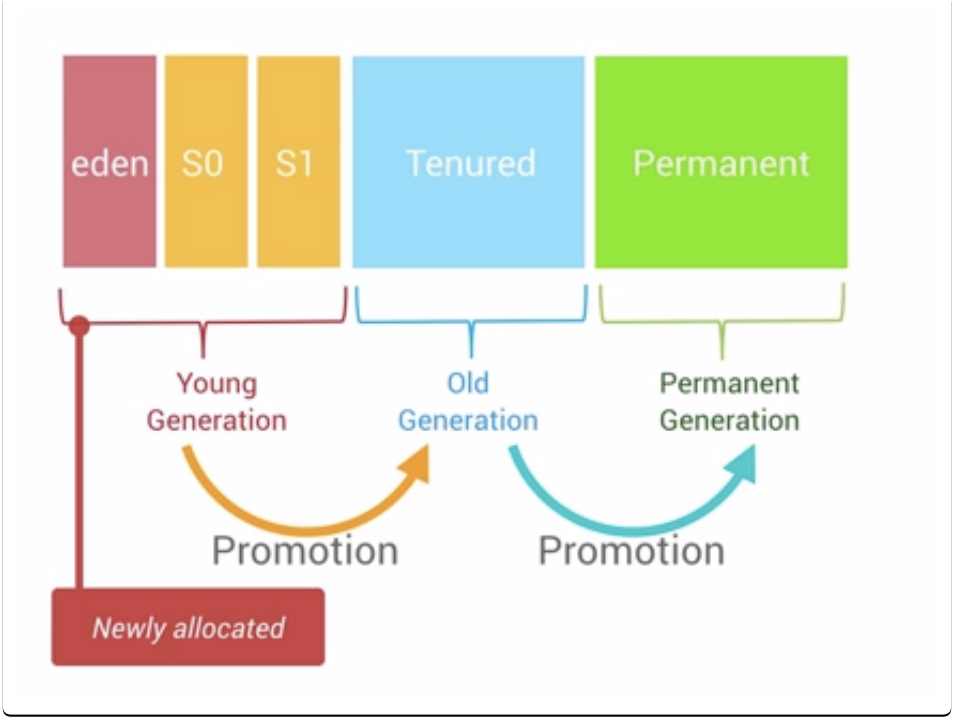
2）分配与回收内存

- 每一个进程的Dalvik heap都反映了使用内存的占用范围。这就是通常逻辑意义上提到的Dalvik Heap Size，它可以随着需要进行增长，但是增长行为会有一个系统为它设定的上限。
- 逻辑上讲的Heap Size和实际物理意义上使用的内存大小是不对等的，Proportional Set

Size(PSS)记录了应用程序自身占用以及和其他进程进行共享的内存。

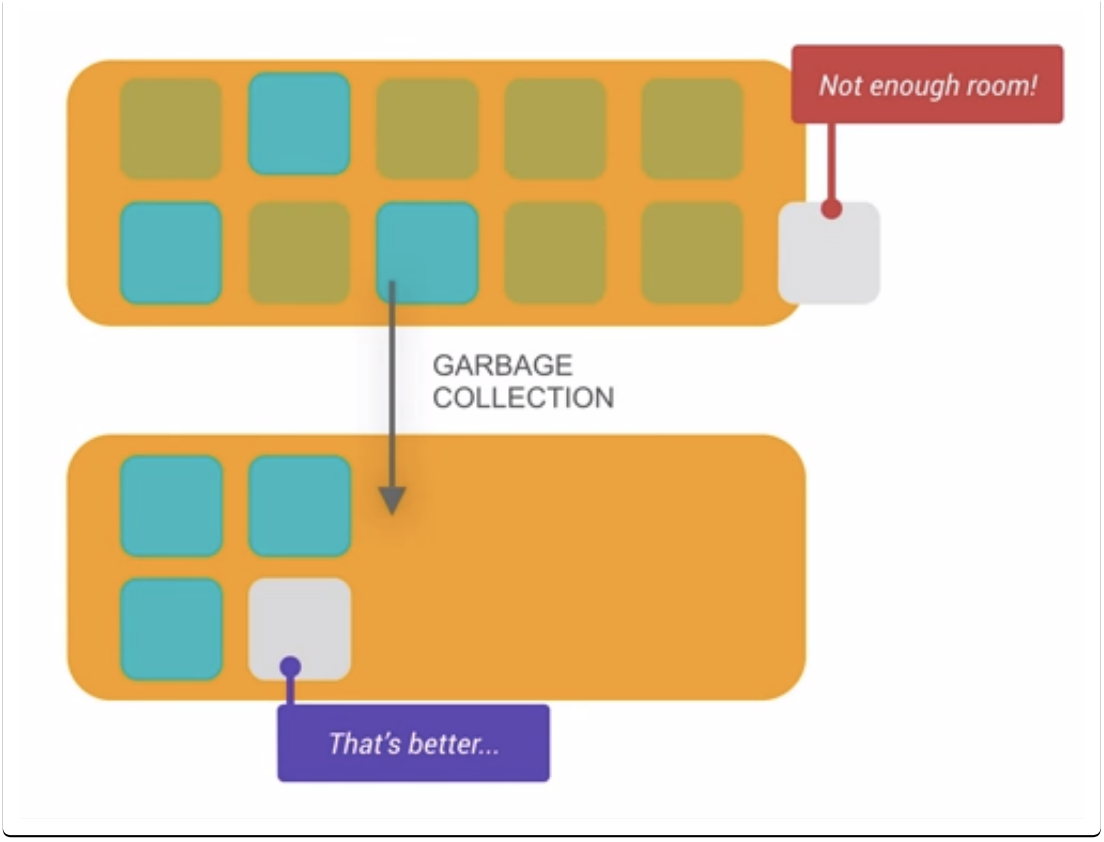
- Android系统并不会对Heap中空闲内存区域做碎片整理。系统仅仅会在新的内存分配之前判断Heap的尾端剩余空间是否足够，如果空间不够会触发gc操作，从而腾出更多空闲的内存空间。在Android的高级系统版本里面针对Heap空间有一个Generational Heap Memory的模型，最近分配的对象会存放在Young Generation区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。系统会根据内存中不同的内存数据类型分别执行不同的gc操作。例如，刚分配到Young Generation区域的对象通常更容易被销毁回收，同时在Young Generation区域的gc操作速度会比Old Generation区域的gc操作速度更快。如下图所示：





每一个Generation的内存区域都有固定的大小，随着新的对象陆续被分配到此区域，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。如下图所示：



通常情况下，GC发生的时候，所有的线程都是会被暂停的。执行GC所占用的时间和它发生在哪一个Generation也有关系，Young Generation中的每次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，遍历树结构查找20000个对象比起遍历50个对象自然是要慢很多的。

3）限制应用的内存

- 为了整个Android系统的内存控制需要，Android系统为每一个应用程序都设置了一个硬性的Dalvik Heap Size最大限制阈值，这个阈值在不同的设备上会因为RAM大小不同而各有差异。如果你的应用占用内存空间已经接近这个阈值，此时再尝试分配内存的话，很容易引起`OutOfMemoryError`的错误。
- `ActivityManager.getMemoryClass()`可以用来查询当前应用的Heap Size阈值，这个方法会返回一个整数，表明你的应用的Heap Size阈值是多少Mb(megabates)。

4）应用切换操作

- Android系统并不会在用户切换应用的时候做交换内存的操作。Android会把那些不包含Foreground组件的应用进程放到LRU Cache中。例如，当用户开始启动了一个应用，系统会为它创建了一个进程，但是当用户离开这个应用，此进程并不会立即被销毁，而是会被放到系统的Cache当中，如果用户后来再切换回到这个应用，此进程就能够被马上完整的恢

复，从而实现应用的快速切换。

- 如果你的应用中有一个被缓存的进程，这个进程会占用一定的内存空间，它会对系统的整体性能有影响。因此当系统开始进入Low Memory的状态时，它会由系统根据LRU的规则与应用的优先级，内存占用情况以及其他因素的影响综合评估之后决定是否被杀掉。
- 对于那些非foreground的进程，Android系统是如何判断Kill掉哪些进程的问题，请参考Processes and Threads。

(二)OOM（OutOfMemory）

前面我们提到过使用getMemoryClass()的方法可以得到Dalvik Heap的阈值。简要的获取某个应用的内存占用情况可以参考下面的示例（关于更多内存查看的知识，可以参考这篇官方教程：Investigating Your RAM Usage）

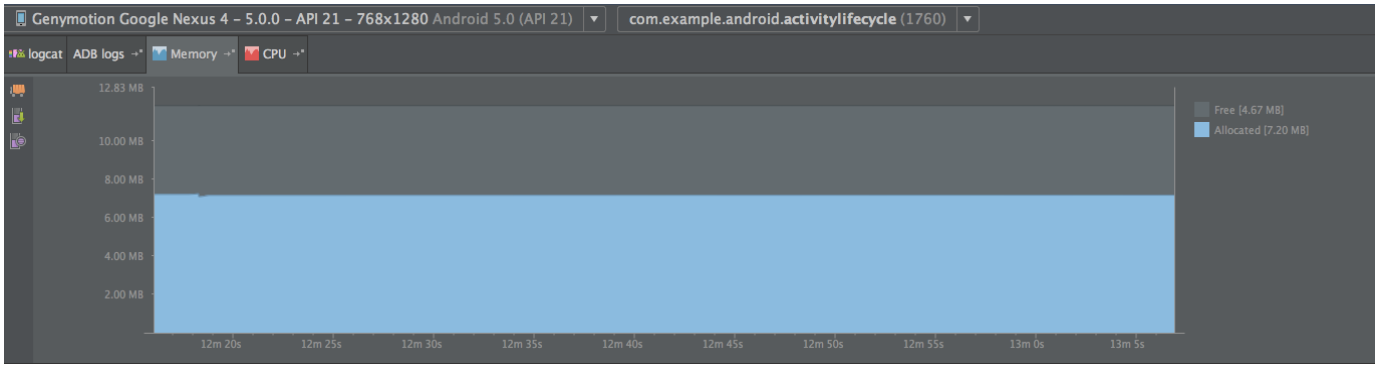
1）查看内存使用情况

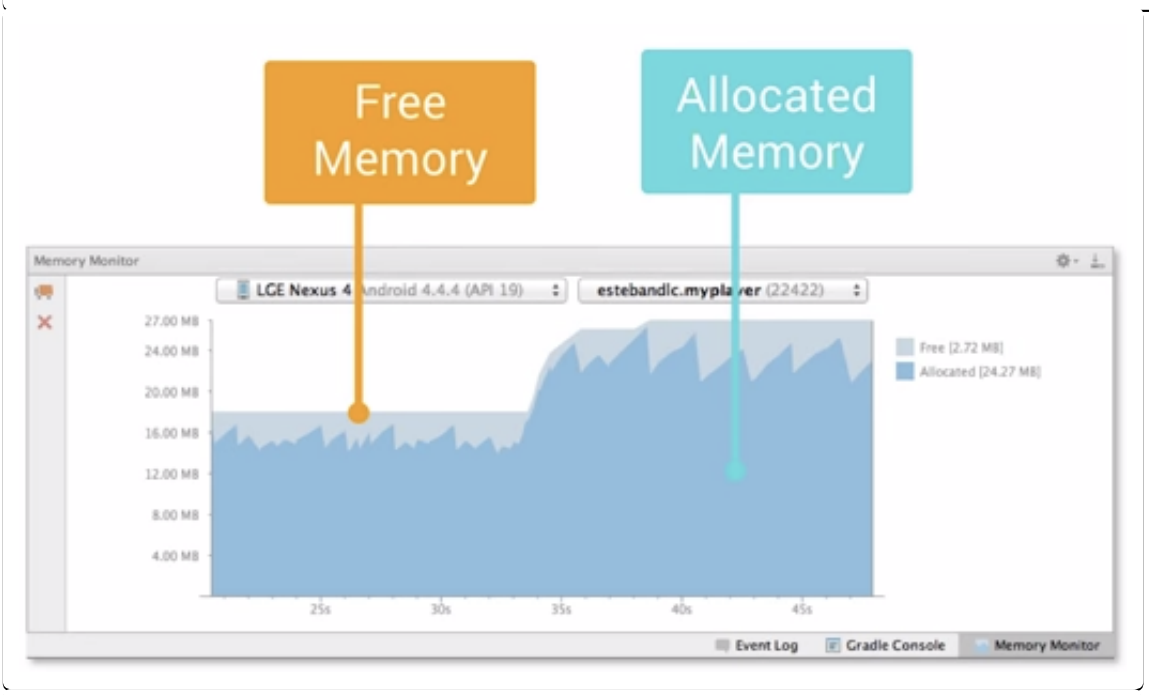
- 通过命令行查看内存详细占用情况：

```
→ ~ adb shell dumpsys meminfo -a com.example.android.activitylifecycle
Applications Memory Usage (kB):
Uptime: 757671 Realtime: 757671

** MEMINFO in pid 1760 [com.example.android.activitylifecycle] **
                   Pss      Pss   Shared  Private   Shared  Private  Swapped     Heap     Heap     Heap
                 Total    Clean    Dirty             Clean    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------  ------   ------   ------   ------   ------   ------   ------
  Native Heap     6432        0     1944    6344        0        0        0    16721    16721    15022
  Dalvik Heap     1923        0     8316    1576        0        0        0    12152     7443     4709
 Dalvik Other      448        0        4     448        0        0        0
        Stack      112        0        0     112        0        0        0
    Other dev        4        0       28       0        0        4        0
     .so mmap     1114      180     2588     148     6188      180        0
     .apk mmap     148        0        0       0     1064        0        0
     .ttf mmap      34        4        0       0       88        4        0
     .dex mmap     148      144        0       0        8      144        0
    code mmap     1320      204        0       0     9276      204        0
   image mmap     1953      648     2560     504     7824      648        0
   Other mmap       64        0        8       4      140       52        0
      Unknown       89        0      100      88        0        0        0
        TOTAL    13789     1180    15548    9224    24588     1236        0    28873    24164    19731
```

- 通过Android Studio的Memory Monitor查看内存中Dalvik Heap的实时变化





2）发生OOM的条件

关于Native Heap，Dalvik Heap，Pss等内存管理机制比较复杂，这里不展开描述。简单的说，通过不同的内存分配方式（malloc/mmap/JNIEnv/etc）对不同的对象（bitmap，etc）进行操作会因为Android系统版本的差异而产生不同的行为，对Native Heap与Dalvik Heap以及

OOM的判断条件都会有所影响。在2.x的系统上，我们常常可以看到Heap Size的total值明显超过了通过getMemoryClass()获取到的阈值而不会发生OOM的情况，那么针对2.x与4.x的Android系统，到底是如何判断会发生OOM呢？

- Android 2.x系统 GC LOG中的dalvik allocated + external allocated + 新分配的大小 >= getMemoryClass()值的时候就会发生OOM。 例如，假设有这么一段Dalvik输出的GC LOG：GC_FOR_MALLOC free 2K, 13% free 32586K/37455K, external 8989K/10356K, paused 20ms，那么32586+8989+(新分配23975)=65550>64M时，就会发生OOM。

- Android 4.x系统 Android 4.x的系统废除了external的计数器，类似bitmap的分配改到dalvik的java heap中申请，只要allocated + 新分配的内存 >= getMemoryClass()的时候就会发生OOM，如下图所示（虽然图示演示的是art运行环境，但是统计规则还是和dalvik保持一致）

```
I/art : Clamp target GC heap from 111MB to 96MB
I/art : Clamp target GC heap from 111MB to 96MB
I/art : Alloc partial concurrent mark sweep GC freed 4(128B) AllocSpace objects, 0(0B) LOS objects, 0% free, 95MB/96MB, paused 368u
I/art : WaitForGcToComplete blocked for 5.139ms for cause Background
I/art : Clamp target GC heap from 111MB to 96MB
I/art : Alloc concurrent mark sweep GC freed 7(12KB) AllocSpace objects, 0(0B) LOS objects, 0% free, 95MB/96MB, paused 261us total
I/art : WaitForGcToComplete blocked for 6.784ms for cause Background
I/art : Forcing collection of SoftReferences for 36KB allocation
I/art : Clamp target GC heap from 111MB to 96MB
I/art : Alloc concurrent mark sweep GC freed 12(392B) AllocSpace objects, 0(0B) LOS objects, 0% free, 95MB/96MB, paused 274us total
I/art : WaitForGcToComplete blocked for 6.255ms for cause Background
E/art : Throwing OutOfMemoryError "Failed to allocate a 36876 byte allocation with 26476 free bytes and 25KB until OOM"
```

(三)如何避免OOM总结

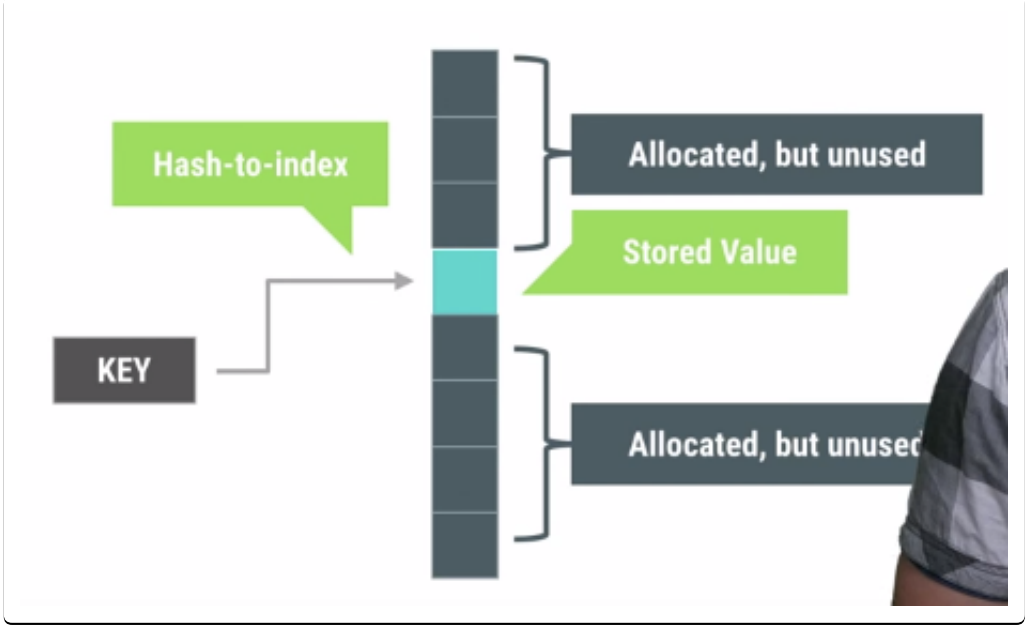
前面介绍了一些基础的内存管理机制以及OOM的基础知识，那么在实践操作当中，有哪些指导性的规则可以参考呢？归纳下来，可以从四个方面着手，首先是减小对象的内存占用，其次是内存对象的重复利用，然后是避免对象的内存泄露，最后是内存使用策略优化。

减小对象的内存占用

避免OOM的第一步就是要尽量减少新分配出来的对象占用内存的大小，尽量使用更加轻量的对象。

1）使用更加轻量的数据结构

例如，我们可以考虑使用ArrayMap/SparseArray而不是HashMap等传统数据结构，下图演示了HashMap的简要工作原理，相比起Android系统专门为移动操作系统编写的ArrayMap容器，在大多数情况下，都显示效率低下，更占内存。通常的HashMap的实现方式更加消耗内存，因为它需要一个额外的实例对象来记录Mapping操作。另外，SparseArray更加高效在于他们避免了对key与value的autobox自动装箱，并且避免了装箱后的解箱。



关于更多ArrayMap/SparseArray的讨论，请参考http://hukai.me/android-performance-patterns-season-3/的前三个段落

2）避免在Android里面使用Enum

Android官方培训课程提到过**“Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.”**，具体原理请参考http://hukai.me/android-performance-patterns-season-3/，所以请避免在Android里面使用到枚举。

3）减小Bitmap对象的内存占用

Bitmap是一个极容易消耗内存的大胖子，减小创建出来的Bitmap的内存占用是很重要的，通常来说有下面2个措施：

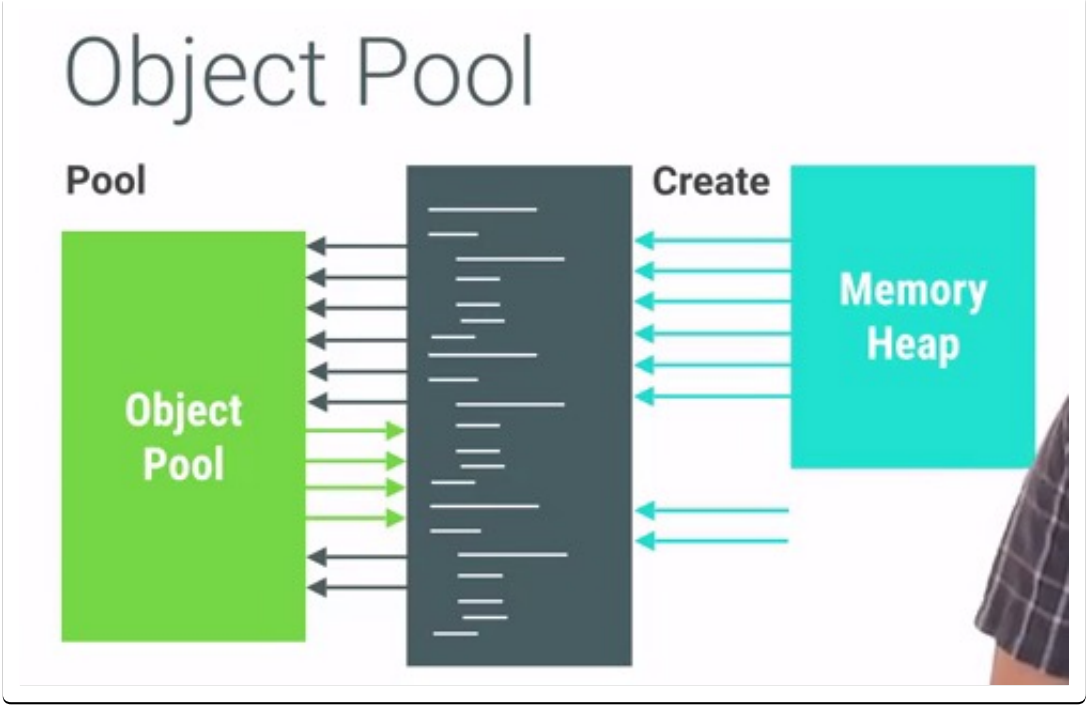
- inSampleSize：缩放比例，在把图片载入内存之前，我们需要先计算出一个合适的缩放比例，避免不必要的大图载入。
- decode format：解码格式，选择ARGB_8888/RBG_565/ARGB_4444/ALPHA_8，存在很大差异。

4）使用更小的图片

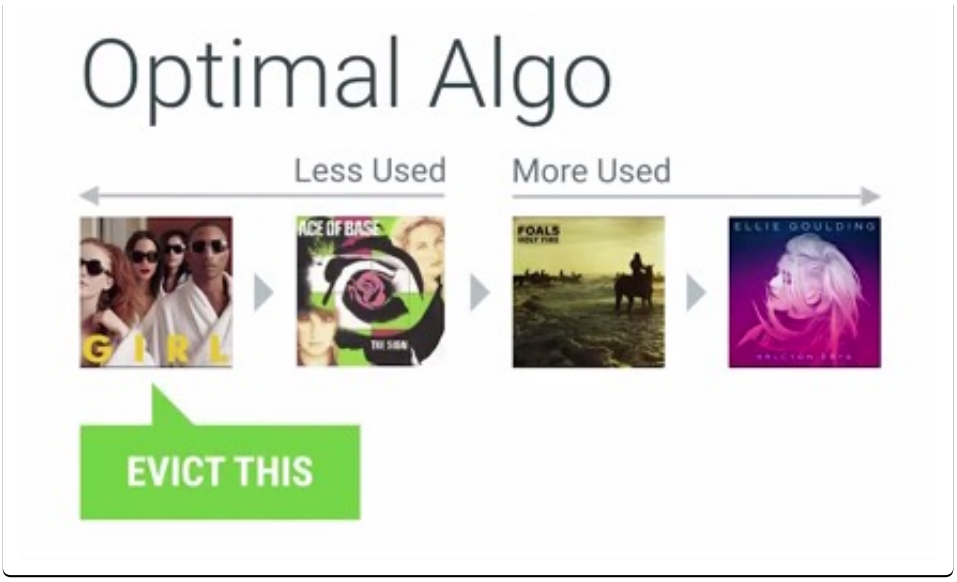
在设计给到资源图片的时候，我们需要特别留意这张图片是否存在可以压缩的空间，是否可以使用一张更小的图片。尽量使用更小的图片不仅仅可以减少内存的使用，还可以避免出现大量的InflationException。假设有一张很大的图片被XML文件直接引用，很有可能在初始化视图的时候就会因为内存不足而发生InflationException，这个问题的根本原因其实是发生了OOM。

内存对象的重复利用

大多数对象的复用，最终实施的方案都是利用对象池技术，要么是在编写代码的时候显式的在程序里面去创建对象池，然后处理好复用的实现逻辑，要么就是利用系统框架既有的某些复用特性达到减少对象的重复创建，从而减少内存的分配与回收。



在Android上面最常用的一个缓存算法是LRU(Least Recently Use)，简要操作原理如下图所示：

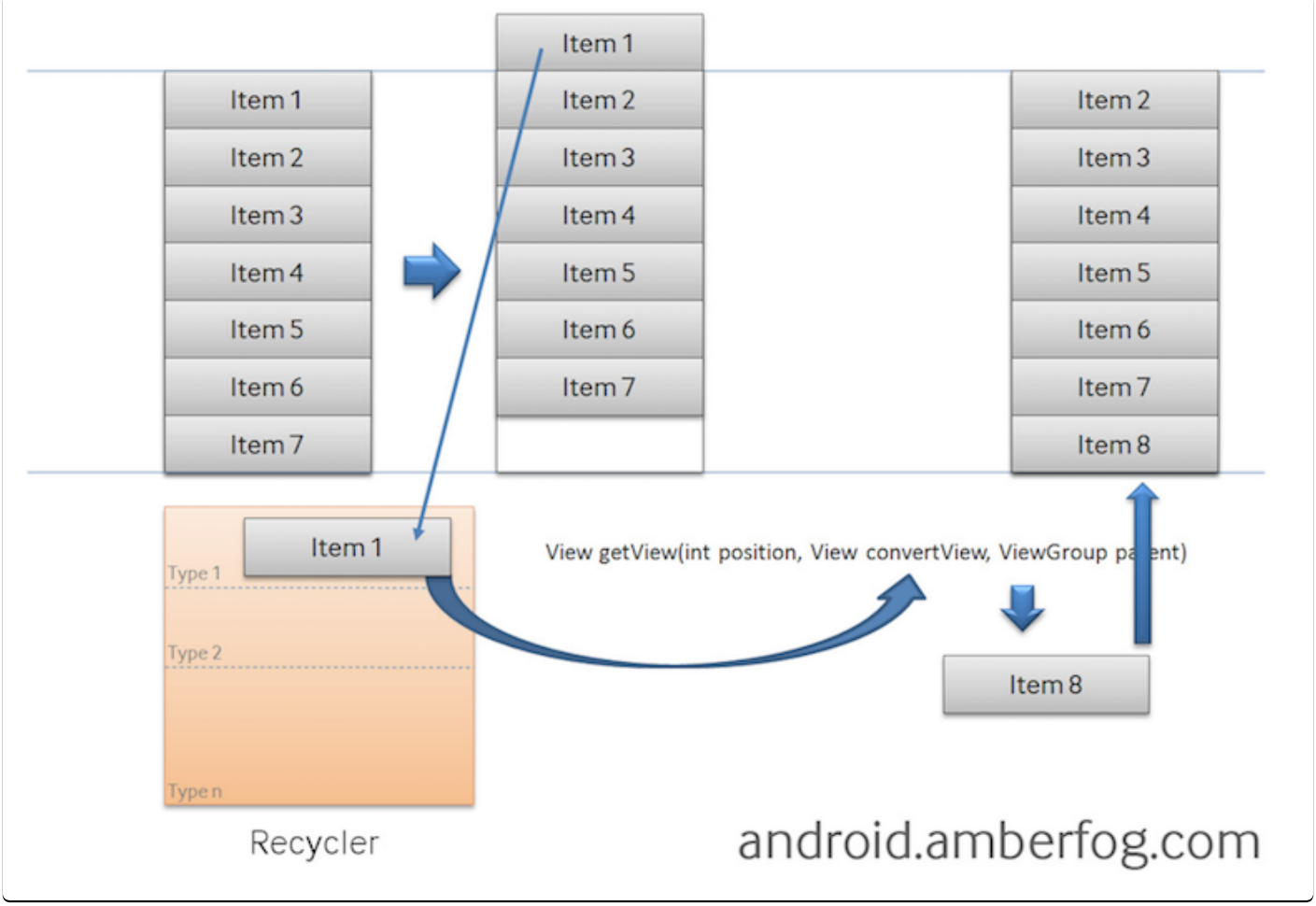


1）复用系统自带的资源

Android系统本身内置了很多的资源，例如字符串/颜色/图片/动画/样式以及简单布局等等，这些资源都可以在应用程序中直接引用。这样做不仅仅可以减少应用程序的自身负重，减小APK的大小，另外还可以一定程度上减少内存的开销，复用性更好。但是也有必要留意Android系

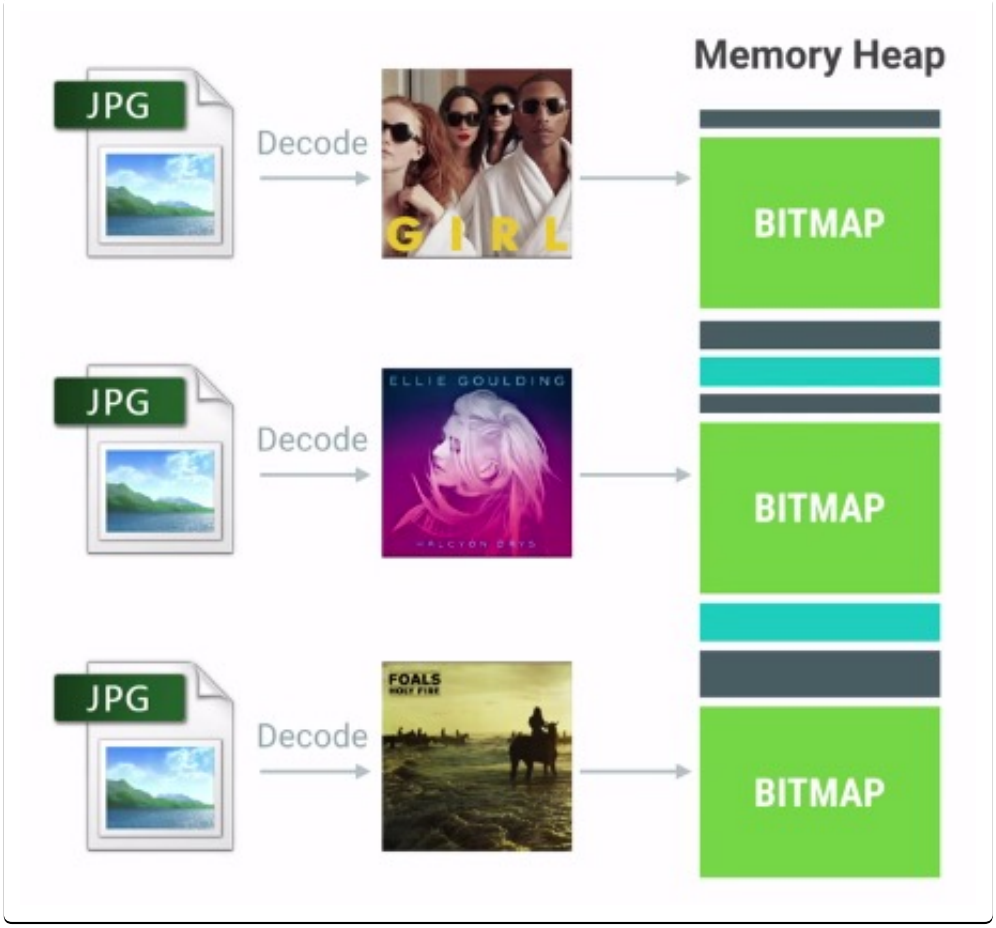
统的版本差异性，对那些不同系统版本上表现存在很大差异，不符合需求的情况，还是需要应用程序自身内置进去。

2）注意在ListView/GridView等出现大量重复子组件的视图里面对ConvertView的复用

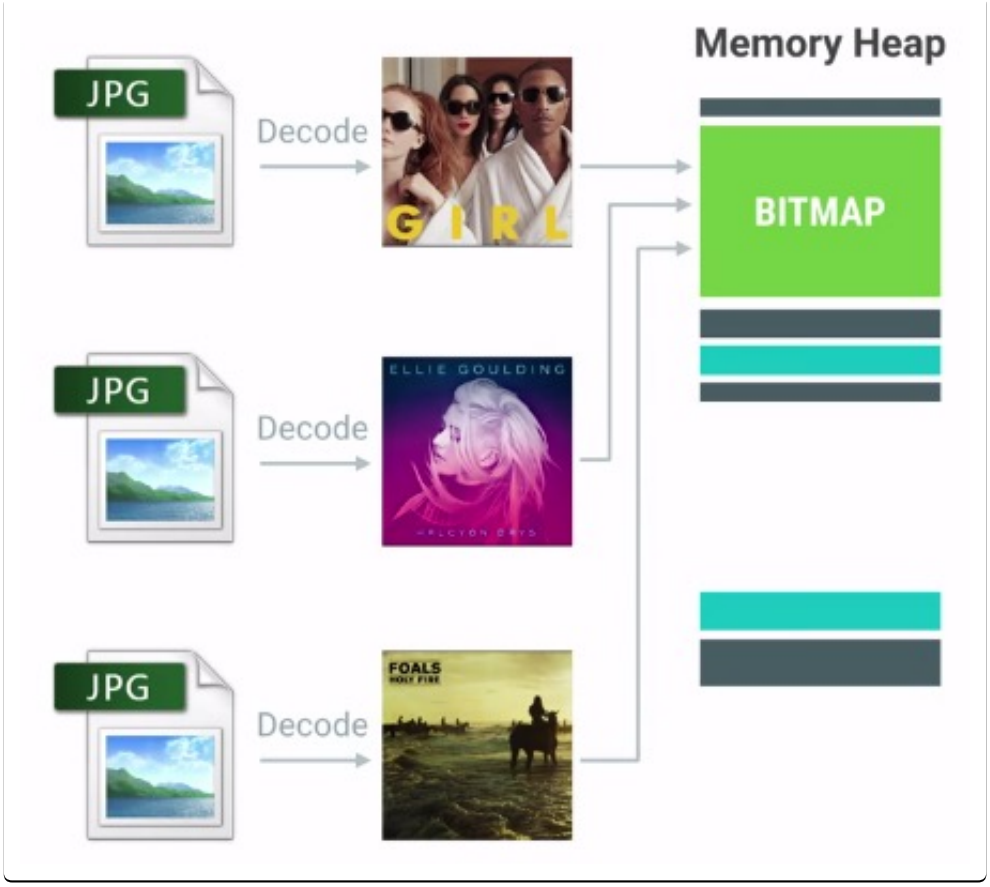


3）Bitmap对象的复用

- 在ListView与GridView等显示大量图片的控件里面需要使用LRU的机制来缓存处理好的Bitmap。



- 利用inBitmap的高级特性提高Android系统在Bitmap分配与释放执行效率上的提升(3.0以及4.4以后存在一些使用限制上的差异)。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。



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：

另外提一点：在2.x的系统上，尽管bitmap是分配在native层，但是还是无法避免被计算到OOM的引用计数器里面。这里提示一下，不少应用会通过反射BitmapFactory.Options里面的inNativeAlloc来达到扩大使用内存的目的，但是如果大家都这么做，对系统整体会造成一定的负面影响，建议谨慎采纳。

4）避免在onDraw方法里面执行对象的创建

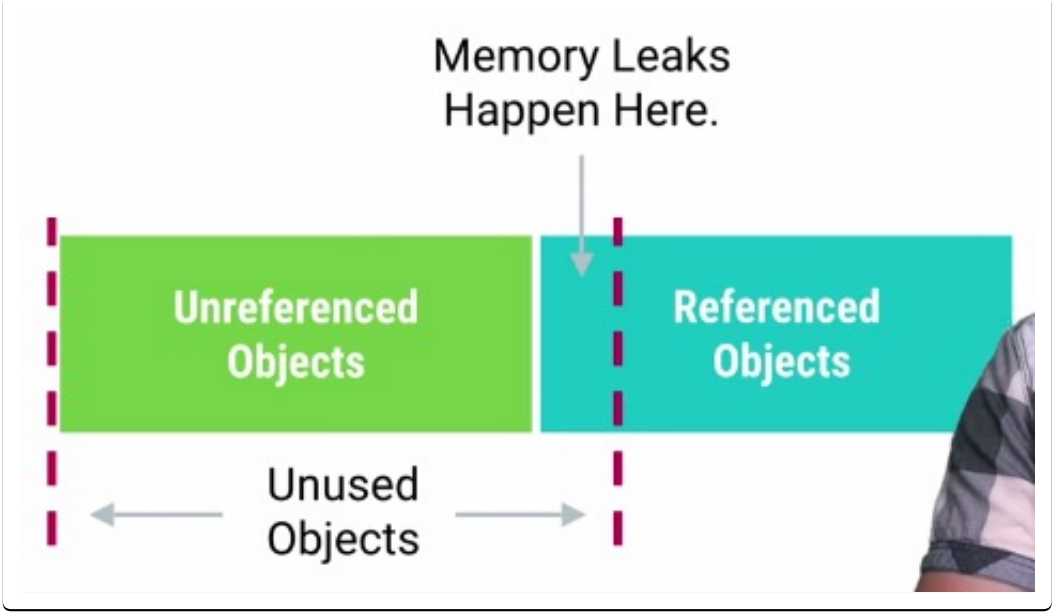
类似onDraw等频繁调用的方法，一定需要注意避免在这里做创建对象的操作，因为他会迅速增加内存的使用，而且很容易引起频繁的gc，甚至是内存抖动。

5）StringBuilder

在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用StringBuilder来替代频繁的“+”。

避免对象的内存泄露

内存对象的泄漏，会导致一些不再使用的对象无法及时释放，这样一方面占用了宝贵的内存空间，很容易导致后续需要分配内存的时候，空闲空间不足而出现OOM。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，容易出现内存抖动，从而引起性能问题。



最新的LeakCanary开源控件，可以很好的帮助我们发现内存泄露的情况，更多关于LeakCanary的介绍，请看这里https://github.com/square/leakcanary(中文使用说明http://www.liaohuqiu.net/cn/posts/leak-canary-read-me/)。另外也可以使用传统的MAT工具查找内存泄露，请参考这里http://android-developers.blogspot.pt/2011/03/memory-analysis-for-android.html（便捷的中文资料http://androidperformance.com/2015/04/11/AndroidMemory-Usage-Of-MAT/）

1）注意Activity的泄漏

通常来说，Activity的泄漏是内存泄漏里面最严重的问题，它占用的内存多，影响面广，我们需要特别注意以下两种情况导致的Activity泄漏：

- 内部类引用导致Activity的泄漏

最典型的场景是Handler导致的Activity泄漏，如果Handler中有延迟的任务或者是等待执行的任务队列过长，都有可能因为Handler继续执行而导致Activity发生泄漏。此时的引用关系链是Looper -> MessageQueue -> Message -> Handler -> Activity。为了解决这个问题，可以在UI退出之前，执行remove Handler消息队列中的消息与runnable对象。或者是使用Static + WeakReference的方式来达到断开Handler与Activity之间存在引用关系的目的。

- Activity Context被传递到其他实例中，这可能导致自身被引用而发生泄漏。

内部类引起的泄漏不仅仅会发生在Activity上，其他任何内部类出现的地方，都需要特别留意！我们可以考虑尽量使用static类型的内部类，同时使用WeakReference的机制来避免因为互相引用而出现的泄露。

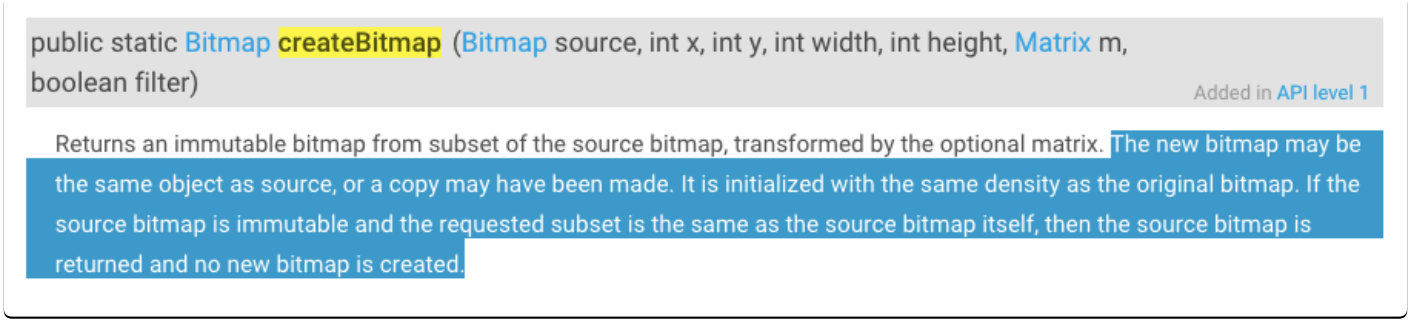
2）考虑使用Application Context而不是Activity Context

对于大部分非必须使用Activity Context的情况（Dialog的Context就必须是Activity Context），我们都可以考虑使用Application Context而不是Activity的Context，这样可以避免不经意的Activity泄露。

3）注意临时Bitmap对象的及时回收

虽然在大多数情况下，我们会对Bitmap增加缓存机制，但是在某些时候，部分Bitmap是需要及时回收的。例如临时创建的某个相对比较大的bitmap对象，在经过变换得到新的bitmap对象之后，应该尽快回收原始的bitmap，这样能够更快释放原始bitmap所占用的空间。

需要特别留意的是Bitmap类里面提供的createBitmap()方法：



这个函数返回的bitmap有可能和source bitmap是同一个，在回收的时候，需要特别检查source bitmap与return bitmap的引用是否相同，只有在不等的情况下，才能够执行source bitmap的recycle方法。

4）注意监听器的注销

在Android程序里面存在很多需要register与unregister的监听器，我们需要确保在合适的时候及时unregister那些监听器。自己手动add的listener，需要记得及时remove这个listener。

5）注意缓存容器中的对象泄漏

有时候，我们为了提高对象的复用性把某些对象放到缓存容器中，可是如果这些对象没有及时从容器中清除，也是有可能导致内存泄漏的。例如，针对2.3的系统，如果把drawable添加到缓存容器，因为drawable与View的强应用，很容易导致activity发生泄漏。而从4.0开始，就不存在这个问题。解决这个问题，需要对2.3系统上的缓存drawable做特殊封装，处理引用解绑的问题，避免泄漏的情况。

6）注意WebView的泄漏

Android中的WebView存在很大的兼容性问题，不仅仅是Android系统版本的不同对WebView产生很大的差异，另外不同的厂商出货的ROM里面WebView也存在着很大的差异。更严重的是标准的WebView存在内存泄露的问题，看这里WebView causes memory leak - leaks the

parent Activity。所以通常根治这个问题的办法是为WebView开启另外一个进程，通过AIDL与主进程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，从而达到内存的完整释放。

7）注意Cursor对象是否及时关闭

在程序中我们经常会进行查询数据库的操作，但时常会存在不小心使用Cursor之后没有及时关闭的情况。这些Cursor的泄露，反复多次出现的话会对内存管理产生很大的负面影响，我们需要谨记对Cursor对象的及时关闭。

内存使用策略优化

1）谨慎使用large heap

正如前面提到的，Android设备根据硬件与软件的设置差异而存在不同大小的内存空间，他们为应用程序设置了不同大小的Heap限制阈值。你可以通过调用`getMemoryClass()`来获取应用的可用Heap大小。在一些特殊的情景下，你可以通过在`manifest`的`application`标签下添加`largeHeap=true`的属性来为应用声明一个更大的heap空间。然后，你可以通过`getLargeMemoryClass()`来获取到这个更大的heap size阈值。然而，声明得到更大Heap阈值的本意是为了一小部分会消耗大量RAM的应用(例如一个大图片的编辑应用)。不要轻易的因为你需要使用更多的内存而去请求一个大的Heap Size。只有当你清楚的知道哪里会使用大量的内存并且知道为什么这些内存必须被保留时才去使用large heap。因此请谨慎使用large heap属性。使用额外的内存空间会影响系统整体的用户体验，并且会使得每次gc的运行时间更长。在任务切换时，系统的性能会大打折扣。另外, large heap并不一定能够获取到更大的heap。在某些有严格限制的机器上，large heap的大小和通常的heap size是一样的。因此即使你申请了large heap，你还是应该通过执行`getMemoryClass()`来检查实际获取到的heap大小。

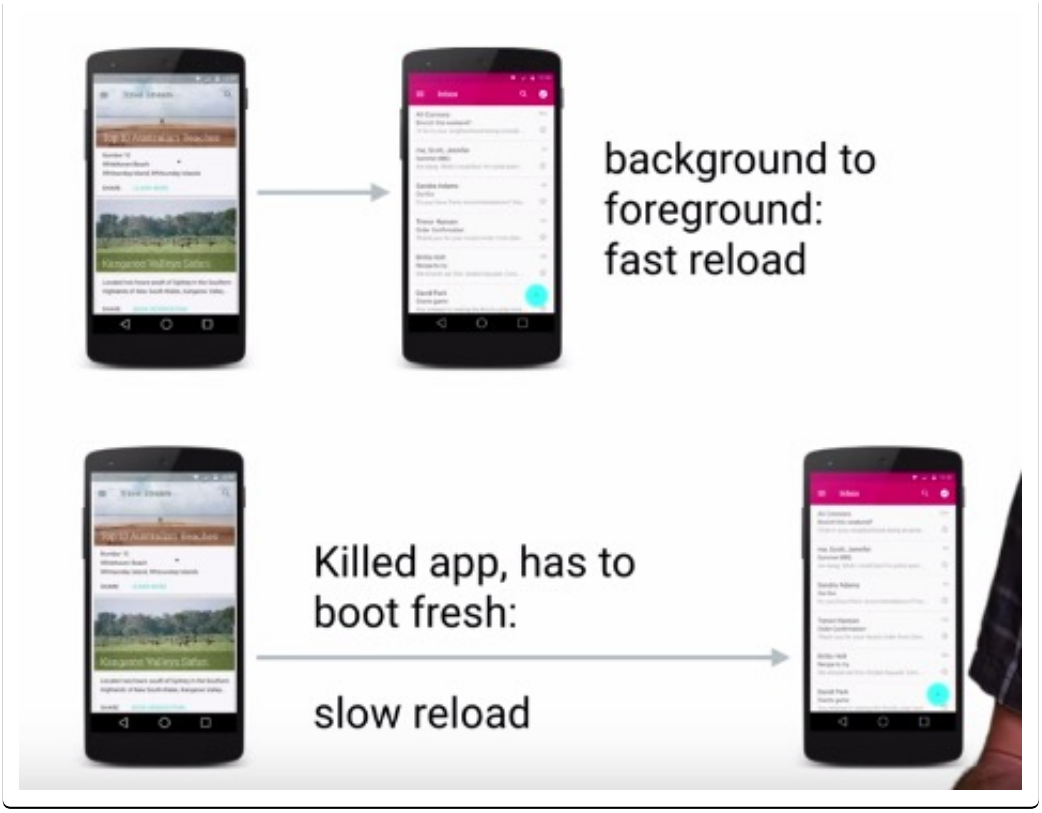
2）综合考虑设备内存阈值与其他因素设计合适的缓存大小

例如，在设计ListView或者GridView的Bitmap LRU缓存的时候，需要考虑的点有：

- 应用程序剩下了多少可用的内存空间?
- 有多少图片会被一次呈现到屏幕上？有多少图片需要事先缓存好以便快速滑动时能够立即显示到屏幕?
- 设备的屏幕大小与密度是多少? 一个xhdpi的设备会比hdpi需要一个更大的Cache来hold住同样数量的图片。
- 不同的页面针对Bitmap的设计的尺寸与配置是什么，大概会花费多少内存?
- 页面图片被访问的频率？是否存在其中的一部分比其他的图片具有更高的访问频繁？如果是，也许你想要保存那些最常访问的到内存中，或者为不同组别的位图(按访问频率分组)设置多个LruCache容器。

3）onLowMemory()与onTrimMemory()

Android用户可以随意在不同的应用之间进行快速切换。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的内存使用情况，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，相比之下就显得稍微有点慢。



- **onLowMemory()**：Android系统提供了一些回调来通知当前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的内存资源，从而确保系统能够继续稳定运行。
- **onTrimMemory(int)**：Android系统从4.0开始还提供了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，收到onTrimMemory()回调的时候，需要根据传递的参数类型进行判断，合理的选择释放自身的一些内存占用，一方面可以提高系统的整体运行流畅度，另外也可以避免自己被系统判断为优先需要杀掉的应用。下图介绍了各种不同的回调参数：

- **TRIM_MEMORY_UI_HIDDEN**：你的应用程序的所有UI界面被隐藏了，即用户点击了Home键或者Back键退出应用，导致应用的UI界面完全不可见。这个时候应该释放一些不可见的时候非必须的资源

当程序正在前台运行的时候，可能会接收到从onTrimMemory()中返回的下面的值之一：

- **TRIM_MEMORY_RUNNING_MODERATE**：你的应用正在运行并且不会被列为可杀死的。但是设备此时正运行于低内存状态下，系统开始触发杀死LRU Cache中的Process的机制。
- **TRIM_MEMORY_RUNNING_LOW**：你的应用正在运行且没有被列为可杀死的。但是设备正运行于更低内存的状态下，你应该释放不用的资源用来提升系统性能。
- **TRIM_MEMORY_RUNNING_CRITICAL**：你的应用仍在运行，但是系统已经把LRU Cache中的大多数进程都已经杀死，因此你应该立即释放所有非必须的资源。如果系统不能回收到足够的RAM数量，系统将会清除所有的LRU缓存中的进程，并且开始杀死那些之前被认为不应该杀死的进程，例如那个包含了一个运行态Service的进程。

当应用进程退到后台正在被Cached的时候，可能会接收到从onTrimMemory()中返回的下面的值之一：

- **TRIM_MEMORY_BACKGROUND:** 系统正运行于低内存状态并且你的进程正处于LRU缓存名单中最不容易杀掉的位置。尽管你的应用进程并不是处于被杀掉的高危险状态，系统可能已经开始杀掉LRU缓存中的其他进程了。你应该释放那些容易恢复的资源，以便于你的进程可以保留下来，这样当用户回退到你的应用的时候才能够迅速恢复。
- **TRIM_MEMORY_MODERATE:** 系统正运行于低内存状态并且你的进程已经已经接近LRU名单的中部位置。如果系统开始变得更加内存紧张，你的进程是有可能被杀死的。
- **TRIM_MEMORY_COMPLETE:** 系统正运行于低内存的状态并且你的进程正处于LRU名单中最容易被杀掉的位置。你应该释放任何不影响你的应用恢复状态的资源。

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*

- 因为onTrimMemory()的回调是在API 14才被加进来的，对于老的版本，你可以使用onLowMemory)回调来进行兼容。onLowMemory相当与TRIM_MEMORY_COMPLETE。
- 请注意：当系统开始清除LRU缓存中的进程时，虽然它首先按照LRU的顺序来执行操作，但是它同样会考虑进程的内存使用量以及其他因素。占用越少的进程越容易被留下来。

4）资源文件需要选择合适的文件夹进行存放

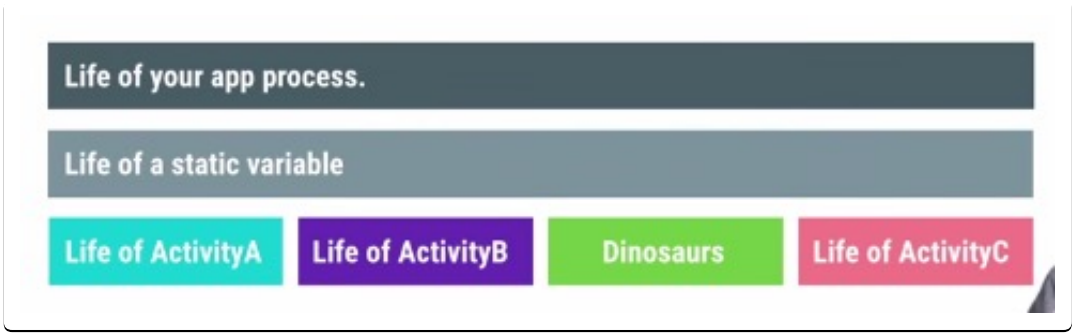
我们知道`hdpi/xhdpi/xxhdpi`等等不同dpi的文件夹下的图片在不同的设备上会经过scale的处理。例如我们只在hdpi的目录下放置了一张100*100的图片，那么根据换算关系，*`xxhdpi`*的手机去引用那张图片就会被拉伸到*200200。需要注意到在这种情况下，内存占用是会显著提高的。对于不希望被拉伸的图片，需要放到assets或者nodpi的目录下。

5）Try catch某些大内存分配的操作

在某些情况下，我们需要事先评估那些可能发生OOM的代码，对于这些可能发生OOM的代码，加入catch机制，可以考虑在catch里面尝试一次降级的内存分配操作。例如decode bitmap的时候，catch到OOM，可以尝试把采样比例再增加一倍之后，再次尝试decode。

6）谨慎使用static对象

因为static的生命周期过长，和应用的进程保持一致，使用不当很可能导致对象泄漏，在Android中应该谨慎使用static对象。



7）特别留意单例对象中不合理的持有

虽然单例模式简单实用，提供了很多便利性，但是因为单例的生命周期和应用保持一致，使用不合理很容易出现持有对象的泄漏。

8）珍惜Services资源

如果你的应用需要在后台使用service，除非它被触发并执行一个任务，否则其他时候Service都应该是停止状态。另外需要注意当这个service完成任务之后因为停止service失败而引起的内存泄漏。 当你启动一个Service，系统会倾向为了保留这个Service而一直保留Service所在的进程。这使得进程的运行代价很高，因为系统没有办法把Service所占用的RAM空间腾出来让给其他组件，另外Service还不能被Paged out。这减少了系统能够存放到LRU缓存当中的进程

数量，它会影响应用之间的切换效率，甚至会导致系统内存使用不稳定，从而无法继续保持住所有目前正在运行的service。 建议使用IntentService，它会在处理完交代给它的任务之后尽快结束自己。更多信息，请阅读Running in a Background Service。

9）优化布局层次，减少内存消耗

越扁平化的视图布局，占用的内存就越少，效率越高。我们需要尽量保证布局足够扁平化，当使用系统提供的View无法实现足够扁平的时候考虑使用自定义View来达到目的。

10）谨慎使用“抽象”编程

很多时候，开发者会使用抽象类作为”好的编程实践”，因为抽象能够提升代码的灵活性与可维护性。然而，抽象会导致一个显著的额外内存开销：他们需要同等量的代码用于可执行，那些代码会被mapping到内存中，因此如果你的抽象没有显著的提升效率，应该尽量避免他们。

11）使用nano protobufs序列化数据

Protocol buffers是由Google为序列化结构数据而设计的，一种语言无关，平台无关，具有良好的扩展性。类似XML，却比XML更加轻量，快速，简单。如果你需要为你的数据实现序列化与协议化，建议使用nano protobufs。关于更多细节，请参考protobuf readme的”Nano version”章节。

12）谨慎使用依赖注入框架

使用类似Guice或者RoboGuice等框架注入代码，在某种程度上可以简化你的代码。下面是使用RoboGuice前后的对比图：

```
class AndroidWay extends Activity {
    TextView name;
    ImageView thumbnail;
    LocationManager loc;
    Drawable icon;
    String myName;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        name      = (TextView) findViewById(R.id.name);
        thumbnail = (ImageView) findViewById(R.id.thumbnail);
        loc       = (LocationManager) getSystemService(Activity.LOCATION_SERVIC
        icon      = getResources().getDrawable(R.drawable.icon);
        myName    = getString(R.string.app_name);
        name.setText( "Hello, " + myName );
    }
}
```

```
@ContentView(R.layout.main)
class RoboWay extends RoboActivity {
    @InjectView(R.id.name)              TextView name;
    @InjectView(R.id.thumbnail)         ImageView thumbnail;
    @InjectResource(R.drawable.icon)    Drawable icon;
    @InjectResource(R.string.app_name) String myName;
    @Inject                             LocationManager loc;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        name.setText( "Hello, " + myName );
    }
}
```

使用RoboGuice之后，代码是简化了不少。然而，那些注入框架会通过扫描你的代码执行许多初始化的操作，这会导致你的代码需要大量的内存空间来mapping代码，而且mapped pages会长时间的被保留在内存中。除非真的很有必要，建议谨慎使用这种技术。

13）谨慎使用多进程

使用多进程可以把应用中的部分组件运行在单独的进程当中，这样可以扩大应用的内存占用范围，但是这个技术必须谨慎使用，绝大多数应用都不应该贸然使用多进程，一方面是因为使用多进程会使得代码逻辑更加复杂，另外如果使用不当，它可能反而会导致显著增加内存。当你的应用需要运行一个常驻后台的任务，而且这个任务并不轻量，可以考虑使用这个技术。

一个典型的例子是创建一个可以长时间后台播放的Music Player。如果整个应用都运行在一个进程中，当后台播放的时候，前台的那些UI资源也没有办法得到释放。类似这样的应用可以切分成2个进程：一个用来操作UI，另外一个给后台的Service。

14）使用ProGuard来剔除不需要的代码

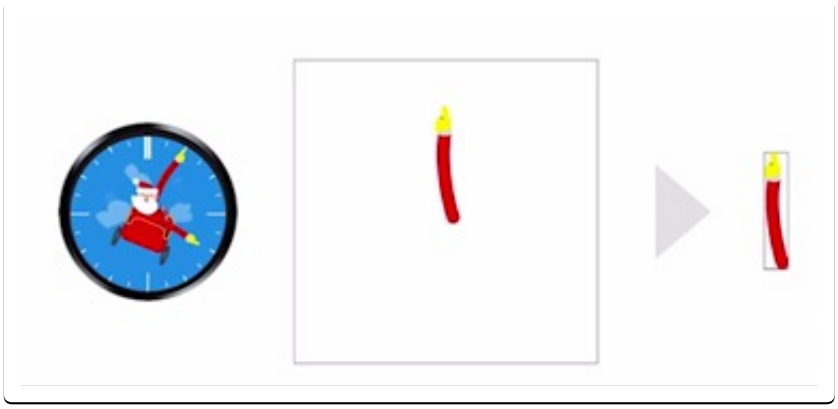
ProGuard能够通过移除不需要的代码，重命名类，域与方法等等对代码进行压缩，优化与混淆。使用ProGuard可以使得你的代码更加紧凑，这样能够减少mapping代码所需要的内存空间。

15）谨慎使用第三方libraries

很多开源的library代码都不是为移动网络环境而编写的，如果运用在移动设备上，并不一定适合。即使是针对Android而设计的library，也需要特别谨慎，特别是在你不知道引入的library具体做了什么事情的时候。例如，其中一个library使用的是nano protobufs, 而另外一个使用的是micro protobufs。这样一来，在你的应用里面就有2种protobuf的实现方式。这样类似的冲突还可能发生在输出日志，加载图片，缓存等等模块里面。另外不要为了1个或者2个功能而导入整个library，如果没有一个合适的库与你的需求相吻合，你应该考虑自己去实现，而不是导入一个大而全的解决方案。

16）考虑不同的实现方式来优化内存占用

在某些情况下，设计的某个方案能够快速实现需求，但是这个方案却可能在内存占用上表现的效率不够好。例如：



对于上面这样一个时钟表盘的实现，最简单的就是使用很多张包含指针的表盘图片，使用帧动画实现指针的旋转。但是如果把指针扣出来，单独进行旋转绘制，显然比载入N多张图片占用的内存要少很多。当然这样做，代码复杂度上会有所增加，这里就需要在优化内存占用与实现简易度之间进行权衡了。

写在最后：

- 设计风格很大程度上会影响到程序的内存与性能，相对来说，如果大量使用类似Material Design的风格，不仅安装包可以变小，还可以减少内存的占用，渲染性能与加载性能都会有一定的提升。
- 内存优化并不就是说程序占用的内存越少就越好，如果因为想要保持更低的内存占用，而频繁触发执行gc操作，在某种程度上反而会导致应用性能整体有所下降，这里需要综合考虑做一定的权衡。
- Android的内存优化涉及的知识面还有很多：内存管理的细节，垃圾回收的工作原理，如何查找内存泄漏等等都可以展开讲很多。OOM是内存优化当中比较突出的一点，尽量减少OOM的概率对内存优化有着很大的意义。