内存使用策略优化

### 1）谨慎使用large heap

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，如图17所示。

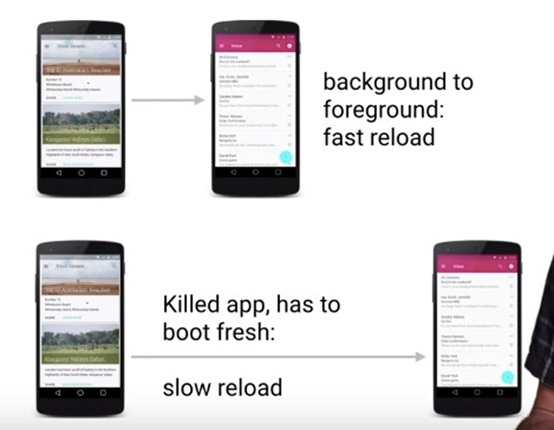


图17  从Kill状态进行恢复体验更慢

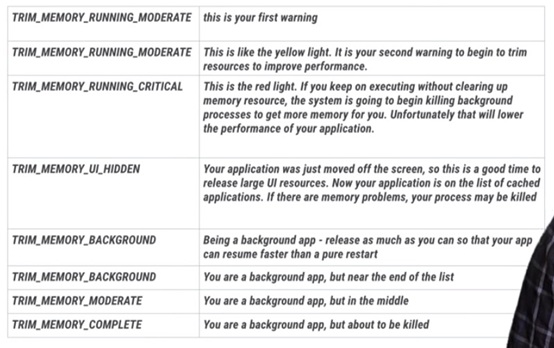
* **onLowMemory()：**Android系统提供了一些回调来通知当前应用的内存使用情况，通常来说，当所有的background应用都被kill掉的时候，forground应用会收到onLowMemory()的回调。在这种情况下，需要尽快释放当前应用的非必须的内存资源，从而确保系统能够继续稳定运行。
* **onTrimMemory(int)：**Android系统从4.0开始还提供了onTrimMemory()的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调，同时在这个回调里面会传递以下的参数，代表不同的内存使用情况，收到onTrimMemory()回调的时候，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。
* **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。



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100100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（如图19所示）。



图19

### 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](http://developer.android.com/reference/android/app/IntentService.html)，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](http://developer.android.com/training/run-background-service/index.html)。

### 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](https://android.googlesource.com/platform/external/protobuf/+/master/java/README.txt" \t "_blank)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。图20是使用RoboGuice前后的对比图：



图20  使用RoboGuice前后对比图

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](http://developer.android.com/tools/help/proguard.html)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：

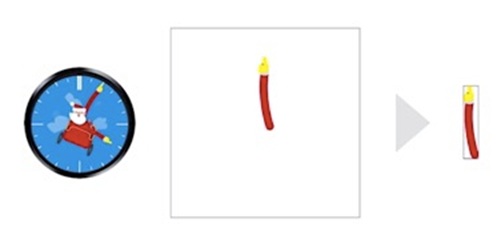


图21

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。