Android内存优化大全

# 内存简介：

RAM（random access memory）随机存取存储器。说白了就是内存。

一般[**Java**](http://lib.csdn.net/base/javase)在内存分配时会涉及到以下区域：

**寄存器（Registers）：**速度最快的存储场所，因为寄存器位于处理器内部**，**我们在程序中无法控制

**栈（Stack）：**存放基本类型的数据和对象的引用，但对象本身不存放在栈中，而是存放在堆中

**堆（Heap）：**堆内存用来存放由new创建的对象和数组。在堆中分配的内存，由Java虚拟机的自动垃圾回收器（GC）来管理。

**静态域（static field）：**  静态存储区域就是指在固定的位置存放应用程序运行时一直存在的数据，Java在内存中专门划分了一个静态存储区域来管理一些特殊的数据变量如静态的数据变量

**常量池（constant pool）：**虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用到常量的一个有序集和，包括直接常量（string,integer和floating point常量）和对其他类型，字段和方法的符号引用。

**非RAM存储：**硬盘等永久存储空间

## **堆栈特点对比：**

由于篇幅原因，下面只简单的介绍一下堆栈的一些特性。

**栈**：当定义一个变量时，Java就在栈中为这个变量分配内存空间，当该变量退出该作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。

**堆**：当堆中的new产生数组和对象超出其作用域后，它们不会被释放，只有在没有引用变量指向它们的时候才变成垃圾，不能再被使用。即使这样，所占内存也不会立即释放，而是等待被垃圾回收器收走。这也是Java比较占内存的原因。

**栈**：存取速度比堆要快，仅次于寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。

**堆**：堆是一个运行时数据区，可以动态地分配内存大小，因此存取速度较慢。也正因为这个特点，堆的生存期不必事先告诉编译器，而且Java的垃圾收集器会自动收走这些不再使用的数据。

**栈**：栈中的数据可以共享， 它是由编译器完成的，有利于节省空间。

例如：需要定义两个变量int a = 3；int b = 3；

编译器先处理int a = 3；首先它会在栈中创建一个变量为a的引用，然后查找栈中是否有3这个值，如果没找到，就将3存放进来，然后将a指向3。接着处理int b = 3；在创建完b的引用变量后，因为在栈中已经有3这个值，便将b直接指向3。这样，就出现了a与b同时均指向3的情况。这时，如果再让a=4；那么编译器会重新搜索栈中是否有4值，如果没有，则将4存放进来，并让a指向4；如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。

**堆**：例如上面栈中a的修改并不会影响到b, 而在堆中一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

## **内存耗用名词解析：**

**VSS**- Virtual Set Size 虚拟耗用内存（包含共享库占用的内存）

**RSS**- Resident Set Size 实际使用物理内存（包含共享库占用的内存）

**PSS**- Proportional Set Size 实际使用的物理内存（比例分配共享库占用的内存）

**USS**- Unique Set Size 进程独自占用的物理内存（不包含共享库占用的内存）

一般来说内存占用大小有如下规律：VSS >= RSS >= PSS >= USS

# OOM：

**内存泄露可以引发很多的问题：**

1.程序卡顿，响应速度慢（内存占用高时JVM虚拟机会频繁触发GC）

2.莫名消失（当你的程序所占内存越大，它在后台的时候就越可能被干掉。反之内存占用越小，在后台存在的时间就越长）

3.直接崩溃（OutOfMemoryError）

**ANDROID内存面临的问题：**

1.有限的堆内存，原始只有16M

2.内存大小消耗等根据设备，[**操作系统**](http://lib.csdn.net/base/operatingsystem)等级，屏幕尺寸的不同而不同

3.程序不能直接控制

4.支持后台多任务处理（multitasking）

5.运行在虚拟机之上

# **5R：**

本文主要通过如下的5R方法来对ANDROID内存进行优化：

**1.Reckon（计算）**

首先需要知道你的app所消耗内存的情况，知己知彼才能百战不殆

**2.Reduce（减少）**

消耗更少的资源

**3.Reuse（重用）**

当第一次使用完以后，尽量给其他的使用

**5.Recycle（回收）**

返回资源给生产流

**4.Review（检查）**

回顾检查你的程序，看看设计或代码有什么不合理的地方。

## **Reckon （计算）：**

了解自己应用的内存使用情况是很有必要的。如果当内存使用过高的话就需要对其进行优化，因为更少的使用内存可以减少ANDROID系统终止我们的进程的几率，也可以提高多任务执行效率和体验效果。

下面从系统内存（system ram）和堆内存(heap)两个方面介绍一些查看和计算内存使用情况的方法：

### System Ram(系统内存)：

观察和计算系统内存使用情况，可以使用Android提供给我们的两个工具**procstats**，**meminfo**。他们一个侧重于后台的内存使用，另一个是运行时的内存使用。

**Process Stats:**

Android 4.4 KitKat 提出了一个新系统服务，叫做procstats。它将帮助你更好的理解你app在后台（background）时的内存使用情况。

Procstats可以去监视你app在一段时间的行为，包括在后台运行了多久，并在此段时间使用了多少内存。从而帮助你快速的找到应用中不效率和不规范的地方去避免影响其performs，尤其是在低内存的设备上运行时。

你可以通过adb shell命令去使用procstats（adb shell dumpsys procstats --hours 3），或者更方便的方式是运行Process Stats开发者工具（在4.4版本的手机中点击Settings > Developer options > Process Stats）

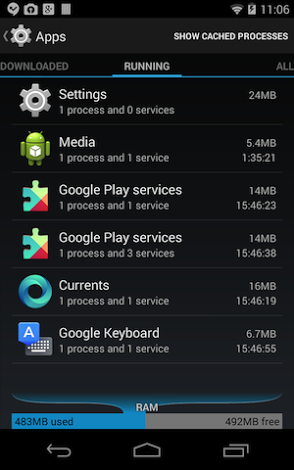


点击单个条目还可以查看详细信息

**meminfo：**

Android还提供了一个工具叫做meminfo。它是根据PSS标准 （Proportional Set Size——实际物理内存）计算每个进程的内存使用并且按照重要程度排序。

你可以通过命令行去执行它：（adb shell dumpsys meminfo）或者使用在设备上点击Settings > Apps > Running（与Procstats不用，它也可以在老版本上运行）



更多关于**Procstats**和**meninfo**的介绍可以参考我翻译的一篇文章：[Process Stats:了解你的APP如何使用内存](http://blog.csdn.net/a396901990/article/details/38390135)

### Heap（堆内存）：

在程序中可以使用如下的方法去查询内存使用情况

**ActivityManager#getMemoryClass()**

查询可用堆内存的限制

3.0(HoneyComb)以上的版本可以通过largeHeap=“true”来申请更多的堆内存（不过这算作“作弊”）

**ActivityManager#getMemoryInfo(ActivityManager.MemoryInfo)**

得到的MemoryInfo中可以查看如下Field的属性：

availMem:表示系统剩余内存

lowMemory：它是boolean值，表示系统是否处于低内存运行

hreshold：它表示当系统剩余内存低于好多时就看成低内存运行

**android.os.Debug#getMemoryInfo(Debug.MemoryInfo memoryInfo)**

得到的MemoryInfo中可以查看如下Field的属性：

dalvikPrivateDirty： The private dirty pages used by dalvik。

dalvikPss ：The proportional set size for dalvik.

dalvikSharedDirty ：The shared dirty pages used by dalvik.

nativePrivateDirty ：The private dirty pages used by the native heap.

nativePss ：The proportional set size for the native heap.

nativeSharedDirty ：The shared dirty pages used by the native heap.

otherPrivateDirty ：The private dirty pages used by everything else.

otherPss ：The proportional set size for everything else.

otherSharedDirty ：The shared dirty pages used by everything else.

dalvik：是指dalvik所使用的内存。

native：是被native堆使用的内存。应该指使用C\C++在堆上分配的内存。

other:是指除dalvik和native使用的内存。但是具体是指什么呢？至少包括在C\C++分配的非堆内存，比如分配在栈上的内存。

private:是指私有的。非共享的。

share:是指共享的内存。

PSS：实际使用的物理内存（比例分配共享库占用的内存）

 PrivateDirty：它是指非共享的，又不能换页出去（can not be paged to disk ）的内存的大小。比如Linux为了提高分配内存速度而缓冲的小对象，即使你的进程结束，该内存也不会释放掉，它只是又重新回到缓冲中而已。

SharedDirty:参照PrivateDirty我认为它应该是指共享的，又不能换页出去（can not be paged to disk ）的内存的大小。比如Linux为了提高分配内存速度而缓冲的小对象，即使所有共享它的进程结束，该内存也不会释放掉，它只是又重新回到缓冲中而已。

**android.os.Debug#getNativeHeapSize()**

返回的是当前进程navtive堆本身总的内存大小

**android.os.Debug#getNativeHeapAllocatedSize()**

返回的是当前进程navtive堆中已使用的内存大小

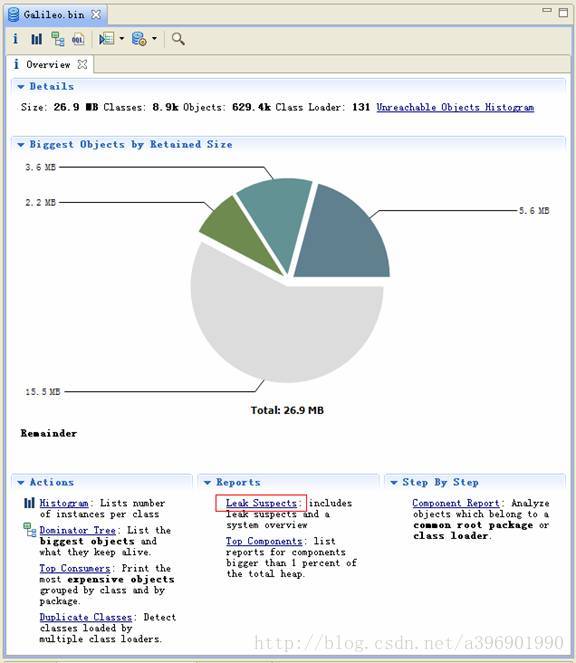
**android.os.Debug#getNativeHeapFreeSize()**

返回的是当前进程navtive堆中已经剩余的内存大小

**Memory Analysis Tool（MAT）：**

通常内存泄露分析被认为是一件很有难度的工作，一般由团队中的资深人士进行。不过，今天我们要介绍的 MAT（Eclipse Memory Analyzer）被认为是一个“傻瓜式“的堆转储文件分析工具，你只需要轻轻点击一下鼠标就可以生成一个专业的分析报告。

如下图：



关于详细的MAT使用我推荐下面这篇文章：[使用 Eclipse Memory Analyzer 进行堆转储文件分析](http://www.ibm.com/developerworks/cn/opensource/os-cn-ecl-ma/index.html)

## ****Reduce ：****

Reduce的意思就是减少，直接减少内存的使用是最有效的优化方式。

下面来看看有哪些方法可以减少内存使用：

### Bitmap：

Bitmap是内存消耗大户，绝大多数的OOM崩溃都是在操作Bitmap时产生的，下面来看看如何几个处理图片的方法：

**图片显示：**

我们需要根据需求去加载图片的大小。

例如在列表中仅用于预览时加载缩略图（thumbnails ）。

只有当用户点击具体条目想看详细信息的时候，这时另启动一个fragment／activity／对话框等等，去显示整个图片

**图片大小：**

直接使用ImageView显示bitmap会占用较多资源，特别是图片较大的时候，可能导致崩溃。   
使用**BitmapFactory.Options**设置inSampleSize, 这样做可以减少对系统资源的要求。   
属性值inSampleSize表示缩略图大小为原始图片大小的几分之一，即如果这个值为2，则取出的缩略图的宽和高都是原始图片的1/2，图片大小就为原始大小的1/4。

1. BitmapFactory.Options bitmapFactoryOptions = **new** BitmapFactory.Options();
2. bitmapFactoryOptions.inJustDecodeBounds = **true**;
3. bitmapFactoryOptions.inSampleSize = 2;
4. // 这里一定要将其设置回false，因为之前我们将其设置成了true
5. // 设置inJustDecodeBounds为true后，decodeFile并不分配空间，即，BitmapFactory解码出来的Bitmap为Null,但可计算出原始图片的长度和宽度
6. options.inJustDecodeBounds = **false**;
7. Bitmap bmp = BitmapFactory.decodeFile(sourceBitmap, options);

**图片像素：**

Android中图片有四种属性，分别是：  
**ALPHA\_8：**每个像素占用1byte内存   
**ARGB\_4444：**每个像素占用2byte内存   
**ARGB\_8888：**每个像素占用4byte内存 （默认）  
**RGB\_565：**每个像素占用2byte内存

Android默认的颜色模式为ARGB\_8888，这个颜色模式色彩最细腻，显示质量最高。但同样的，占用的内存也最大。 所以在对图片效果不是特别高的情况下使用RGB\_565（565没有透明度属性），如下：

1. publicstaticBitmapreadBitMap(Contextcontext, intresId) {
2. BitmapFactory.Optionsopt = newBitmapFactory.Options();
3. opt.inPreferredConfig = Bitmap.Config.RGB\_565;
4. opt.inPurgeable = **true**;
5. opt.inInputShareable = **true**;
6. //获取资源图片
7. InputStreamis = context.getResources().openRawResource(resId);
8. returnBitmapFactory.decodeStream(is, **null**, opt);
9. }

**图片回收：**

使用Bitmap过后，就需要及时的调用**Bitmap.recycle()**方法来释放Bitmap占用的内存空间，而不要等Android系统来进行释放。

下面是释放Bitmap的示例代码片段。

1. // 先判断是否已经回收
2. **if**(bitmap != **null** && !bitmap.isRecycled()){
3. // 回收并且置为null
4. bitmap.recycle();
5. bitmap = **null**;
6. }
7. System.gc();

**捕获异常：**

经过上面这些优化后还会存在报OOM的风险，所以下面需要一道最后的关卡——捕获OOM异常：

1. Bitmap bitmap = **null**;
2. **try** {
3. // 实例化Bitmap
4. bitmap = BitmapFactory.decodeFile(path);
5. } **catch** (OutOfMemoryError e) {
6. // 捕获OutOfMemoryError，避免直接崩溃
7. }
8. **if** (bitmap == **null**) {
9. // 如果实例化失败 返回默认的Bitmap对象
10. **return** defaultBitmapMap;
11. }

### 修改对象引用类型：

**引用类型：**

引用分为四种级别，这四种级别由高到低依次为：强引用>软引用>弱引用>虚引用。

**强引用（strong reference）**  
如：Object object=new Object（），object就是一个强引用了。当内存空间不足，[**Java**](http://lib.csdn.net/base/javase)虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

**软引用（SoftReference）**  
只有内存不够时才回收,常用于缓存；当内存达到一个阀值，GC就会去回收它；

**弱引用（WeakReference）**

弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

**虚引用（PhantomReference）**

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

**软引用和弱引用的应用实例：**

注意：对于SoftReference(软引用)或者WeakReference(弱引用)的Bitmap缓存方案，现在已经不推荐使用了。自Android2.3版本(API Level 9)开始，垃圾回收器更着重于对软/弱引用的回收，所以下面的内容可以选择忽略。

在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

下面以使用软引用为例来详细说明（弱引用的使用方式与软引用是类似的）：

假设我们的应用会用到大量的默认图片，而且这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生OutOfMemory异常。这时，我们可以考虑使用软引用技术来避免这个问题发生。

首先定义一个HashMap，保存软引用对象。

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **private** Map<String, SoftReference<Bitmap>> imageCache = **new** HashMap<String, SoftReference<Bitmap>>();

再来定义一个方法，保存Bitmap的软引用到HashMap。

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **public** **void** addBitmapToCache(String path) {
2. // 强引用的Bitmap对象
3. Bitmap bitmap = BitmapFactory.decodeFile(path);
4. // 软引用的Bitmap对象
5. SoftReference<Bitmap> softBitmap = **new** SoftReference<Bitmap>(bitmap);
6. // 添加该对象到Map中使其缓存
7. imageCache.put(path, softBitmap);
8. }

获取的时候，可以通过SoftReference的get()方法得到Bitmap对象。

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **public** Bitmap getBitmapByPath(String path) {
2. // 从缓存中取软引用的Bitmap对象
3. SoftReference<Bitmap> softBitmap = imageCache.get(path);
4. // 判断是否存在软引用
5. **if** (softBitmap == **null**) {
6. **return** **null**;
7. }
8. // 取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空
9. Bitmap bitmap = softBitmap.get();
10. **return** bitmap;
11. }

使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从而避免内存达到上限，避免Crash发生。

需要注意的是，在垃圾回收器对这个Java对象回收前，SoftReference类所提供的get方法会返回Java对象的强引用，一旦垃圾线程回收该Java对象之后，get方法将返回null。所以在获取软引用对象的代码中，一定要判断是否为null，以免出现NullPointerException异常导致应用崩溃。

**到底什么时候使用软引用，什么时候使用弱引用呢？**

个人认为，如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

还有就是可以根据对象是否经常使用来判断。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

另外，和弱引用功能类似的是WeakHashMap。WeakHashMap对于一个给定的键，其映射的存在并不阻止垃圾回收器对该键的回收，回收以后，其条目从映射中有效地移除。WeakHashMap使用ReferenceQueue实现的这种机制。

### 其他小tips：

#### 对常量使用static final修饰符

让我们来看看这两段在类前面的声明：

**static**int intVal = 42;  
**static**String strVal = "Hello, world!";  
编译器会生成一个叫做clinit的初始化类的方法，当类第一次被使用的时候这个方法会被执行。方法会将42赋给intVal，然后把一个指向类中常量表 的引用赋给strVal。当以后要用到这些值的时候，会在成员变量表中查找到他们。 下面我们做些改进，使用“final”关键字：

**static final**int intVal = 42;  
**static final**String strVal = "Hello, world!";

现在，类不再需要clinit方法，因为在成员变量初始化的时候，会将常量直接保存到类文件中。用到intVal的代码被直接替换成42，而使用strVal的会指向一个字符串常量，而不是使用成员变量。

将一个方法或类声明为final不会带来性能的提升，但是会帮助编译器优化代码。举例说，如果编译器知道一个getter方法不会被重载，那么编译器会对其采用内联调用。

你也可以将本地变量声明为final，同样，这也不会带来性能的提升。使用“final”只能使本地变量看起来更清晰些（但是也有些时候这是必须的，比如在使用匿名内部类的时候）。

#### 静态方法代替虚拟方法

如果不需要访问某对象的字段，将方法设置为静态，调用会加速15%到20%。这也是一种好的做法，因为你可以从方法声明中看出调用该方法不需要更新此对象的状态。

#### 减少不必要的全局变量

尽量避免static成员变量引用资源耗费过多的实例,比如Context

因为Context的引用超过它本身的生命周期，会导致Context泄漏。所以尽量使用Application这种Context类型。 你可以通过调用Context.getApplicationContext()或 Activity.getApplication()轻松得到Application对象。

#### 避免创建不必要的对象

最常见的例子就是当你要频繁操作一个字符串时，使用StringBuffer代替String。

对于所有所有基本类型的组合：int数组比Integer数组好，这也概括了一个基本事实，两个平行的int数组比 (int,int)对象数组性能要好很多。

总体来说，就是避免创建短命的临时对象。减少对象的创建就能减少垃圾收集，进而减少对用户体验的影响。

#### 避免内部Getters/Setters

在Android中，虚方法调用的代价比直接字段访问高昂许多。通常根据面向对象语言的实践，在公共接口中使用Getters和Setters是有道理的，但在一个字段经常被访问的类中宜采用直接访问。

#### 避免使用浮点数

通常的经验是，在Android设备中，浮点数会比整型慢两倍。

#### 使用实体类比接口好

假设你有一个HashMap对象，你可以将它声明为HashMap或者Map：

Map map1 = new HashMap();

HashMap map2 = new HashMap();

哪个更好呢？

按照传统的观点Map会更好些，因为这样你可以改变他的具体实现类，只要这个类继承自Map接口。传统的观点对于传统的程序是正确的，但是它并不适合[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)系统。调用一个接口的引用会比调用实体类的引用多花费一倍的时间。如果HashMap完全适合你的程序，那么使用Map就没有什么价值。如果有些地方你不能确定，先避免使用Map，剩下的交给IDE提供的重构功能好了。(当然公共API是一个例外：一个好的API常常会牺牲一些性能）

#### 避免使用枚举

枚举变量非常方便，但不幸的是它会牺牲执行的速度和并大幅增加文件体积。

使用枚举变量可以让你的API更出色，并能提供编译时的检查。所以在通常的时候你毫无疑问应该为公共API选择枚举变量。但是当性能方面有所限制的时候，你就应该避免这种做法了。

#### for循环

访问成员变量比访问本地变量慢得多，如下面一段代码：

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **for**(**int** i =0; i < **this**.mCount; i++)  {}

永远不要在for的第二个条件中调用任何方法，如下面一段代码：

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **for**(**int** i =0; i < **this**.getCount(); i++) {}

对上面两个例子最好改为：

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **int** count = **this**.mCount; / **int** count = **this**.getCount();
2. **for**(**int** i =0; i < count; i++)  {}

在java1.5中引入的for-each语法。编译器会将对数组的引用和数组的长度保存到本地变量中，这对访问数组元素非常好。 但是编译器还会在每次循环中产生一个额外的对本地变量的存储操作（如下面例子中的变量a），这样会比普通循环多出4个字节，速度要稍微慢一些：

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. **for** (Foo a : mArray) {
2. sum += a.mSplat;
3. }

#### 了解并使用类库

选择Library中的代码而非自己重写，除了通常的那些原因外，考虑到系统空闲时会用汇编代码调用来替代library方法，这可能比JIT中生成的等价的最好的Java代码还要好。

当你在处理字串的时候，不要吝惜使用**String.indexOf()**，**String.lastIndexOf()**等特殊实现的方法。这些方法都是使用C/C++实现的，比起Java循环快10到100倍。

**System.arraycopy**方法在有JIT的Nexus One上，自行编码的循环快9倍。

android.text.format包下的**Formatter**类，提供了IP地址转换、文件大小转换等方法；DateFormat类，提供了各种时间转换，都是非常高效的方法。

**TextUtils**类，对于字符串处理Android为我们提供了一个简单实用的TextUtils类，如果处理比较简单的内容不用去思考正则表达式不妨试试这个在android.text.TextUtils的类

高性能**MemoryFile**类，很多人抱怨Android处理底层I/O性能不是很理想，如果不想使用NDK则可以通过MemoryFile类实现高性能的文件读写操作。MemoryFile适用于哪些地方呢？对于I/O需要频繁操作的，主要是和外部存储相关的I/O操作，MemoryFile通过将 NAND或SD卡上的文件，分段映射到内存中进行修改处理，这样就用高速的RAM代替了ROM或SD卡，性能自然提高不少，对于Android手机而言同时还减少了电量消耗。该类实现的功能不是很多，直接从Object上继承，通过JNI的方式直接在C底层执行。

#### 使用优化过的数据容器

如SparseArray,SparseBooleanArray,LongSparseArray，代替HashMap

**前提：**Key为Integer类型

**原因：**

HashMap 是内存低效的，因为每一个mapping都需要单独的entry（如下图）。

每个元素多占用8byte内存（多了next和hash两个成员变量）。AutoBox【int转Integer，导致产生另一个对象】也会额外加4byte。Entry对象本身至少16byte。

SparseArray可以避免AutoBox，查找方法为二分查找，效率比HashMap低一些，但百量级以内性能差距不大。

#### 使用混淆器移除不必要的代码

       ProGuard工具通过移除无用代码，使用语意模糊来保留类，字段和方法来压缩，优化和混淆代码。可以使你的代码更加完整，更少的RAM 映射页。

#### 对象不用时最好显式置为Null

对象不用时最好显式置为Null可以减少GC开销。

#### 频繁修改时使用 StringBuffer(Thread-Safe)或StringBuilder(Thread-Unsafe)

使用String修改字符串时，若修改后字符串在字符串常量区不存在，便会新生成一个String对象。

#### 在私有内部内中，考虑用包访问权限替代私有访问权限

[复制代码](javascript:void(0);)

public class Foo {

public class Inner {

public void stuff() {

Foo.this.doStuff(Foo.this.mValue);

}

}

private int mValue;

public void run() {

Inner in = new Inner();

mValue = 27;

in.stuff();

}

private void doStuff(int value) {

System.out.println("value:"+value);

}

}

[复制代码](javascript:void(0);)

　　需要注意的关键是：我们定义的一个私有内部类（FooInner），直接访问外部类中的一个私有方法和私有变量。这是合法的，代码也会打印出预期的“Valueis27”。但问题是，虚拟机认为从FooInner），直接访问外部类中的一个私有方法和私有变量。这是合法的，代码也会打印出预期的“Valueis27”。但问题是，虚拟机认为从FooInner中直接访问Foo的私有成员是非法的，因为他们是两个不同的类，尽管Java语言允许内部类访问外部类的私有成员，但是通过编译器生成几个综合方法来桥接这些间隙的。

/\*package\*/static int Foo.access$100(Foo foo) {

return foo.mValue;

}

/\*package\*/static void Foo.access%200(Foo foo,int value) {

foo.duStuff(value);

}

　　内部类会在外部类中任何需要访问mValue字段或调用doStuff方法的地方调用这些静态方法。这意味着这些代码将直接存取成员变量表现为通过存取器方法访问。之前提到过存取器访问如何比直接访问慢，这例子说明，某些语言约会定导致不可见的性能问题。如果你在高性能的Hotspot中使用这些代码，可以通过声明被内部类访问的字段和成员为包访问权限，而非私有。但这也意味着这些字段会被其他处于同一个包中的类访问，因此在公共API中不宜采用。

#### 将与内部类一同使用的变量声明在包范围内

请看下面的类定义：

[复制代码](javascript:void(0);)

public class Foo {

private class Inner {

void stuff() {

Foo.this.doStuff(Foo.this.mValue);

}

}

private int mValue;

public void run() {

Inner in = new Inner();

mValue = 27;

in.stuff();

}

private void doStuff(int value) {

System.out.println("Value is " + value);

}

}

[复制代码](javascript:void(0);)

这其中的关键是，我们定义了一个内部类(FooInner)，它需要访问外部类的私有域变量和函数。这是合法的，并且会打印出我们希望的结果Valueis27。问题是在技术上来讲（在幕后）FooInner)，它需要访问外部类的私有域变量和函数。这是合法的，并且会打印出我们希望的结果Valueis27。问题是在技术上来讲（在幕后）FooInner是一个完全独立的类，它要直接访问Foo的私有成员是非法的。要跨越这个鸿沟，编译器需要生成一组方法：

/\*package\*/ static int Foo.access$100(Foo foo) {

return foo.mValue;

}

/\*package\*/ static void Foo.access$200(Foo foo, int value) {

foo.doStuff(value);

}

　　内部类在每次访问mValueg和gdoStuffg方法时，都会调用这些静态方法。就是说，上面的代码说明了一个问题，你是在通过接口方法访问这些成员变量和函数而不是直接调用它们。在前面我们已经说过，使用接口方法（getter、setter）比直接访问速度要慢。所以这个例子就是在特定语法下面产生的一个“隐性的”性能障碍。通过将内部类访问的变量和函数声明由私有范围改为包范围，我们可以避免这个问题。这样做可以让代码运行更快，并且避免产生额外的静态方法。（遗憾的是，这些域和方法可以被同一个包内的其他类直接访问，这与经典的OO原则相违背。因此当你设计公共API的时候应该谨慎使用这条优化原则）。

## Reuse:

Reuse重用，减少内存消耗的重要手段之一。

核心思路就是将已经存在的内存资源重新使用而避免去创建新的，最典型的使用就是**缓存（Cache）**和**池（Pool）**。

### Bitmap缓存：

Bitmap缓存分为两种：

一种是内存缓存，一种是硬盘缓存。

* **内存缓存（LruCache）：**

以牺牲宝贵的应用内存为代价，内存缓存提供了快速的Bitmap访问方式。系统提供的LruCache类是非常适合用作缓存Bitmap任务的，它将最近被引用到的对象存储在一个强引用的LinkedHashMap中，并且在缓存超过了指定大小之后将最近不常使用的对象释放掉。

注意：以前有一个非常流行的内存缓存实现是SoftReference(软引用)或者WeakReference(弱引用)的Bitmap缓存方案，然而现在已经不推荐使用了。自Android2.3版本(API Level 9)开始，垃圾回收器更着重于对软/弱引用的回收，这使得上述的方案相当无效。

* **硬盘缓存（DiskLruCache）：**

一个内存缓存对加速访问最近浏览过的Bitmap非常有帮助，但是你不能局限于内存中的可用图片。GridView这样有着更大的数据集的组件可以很轻易消耗掉内存缓存。你的应用有可能在执行其他任务(如打电话)的时候被打断，并且在后台的任务有可能被杀死或者缓存被释放。一旦用户重新聚焦(resume)到你的应用，你得再次处理每一张图片。

在这种情况下，硬盘缓存可以用来存储Bitmap并在图片被内存缓存释放后减小图片加载的时间(次数)。当然，从硬盘加载图片比内存要慢，并且应该在后台线程进行，因为硬盘读取的时间是不可预知的。

注意：如果访问图片的次数非常频繁，那么ContentProvider可能更适合用来存储缓存图片，例如Image Gallery这样的应用程序。

更多关于内存缓存和硬盘缓存的内容请看Google官方教程<https://developer.android.com/develop/index.html>

### 图片缓存的开源项目：

对于图片的缓存现在都倾向于使用开源项目，这里我列出几个我搜到的：

**1. Android-Universal-Image-Loader 图片缓存**

目前使用最广泛的图片缓存，支持主流图片缓存的绝大多数特性。  
项目地址：https://github.com/nostra13/Android-Universal-Image-Loader

**2. picasso square开源的图片缓存**  
项目地址：https://github.com/square/picasso  
特点：(1)可以自动检测adapter的重用并取消之前的下载  
(2)图片变换  
(3)可以加载本地资源  
(4)可以设置占位资源  
(5)支持debug模式

**3. ImageCache 图片缓存，包含内存和Sdcard缓存**  
项目地址：https://github.com/Trinea/AndroidCommon  
特点：

(1)支持预取新图片，支持等待队列  
(2)包含二级缓存，可自定义文件名保存规则  
(3)可选择多种缓存[**算法**](http://lib.csdn.net/base/datastructure)(FIFO、LIFO、LRU、MRU、LFU、MFU等13种)或自定义缓存算法  
(4)可方便的保存及初始化恢复数据  
(5)支持不同类型网络处理  
(6)可根据系统配置初始化缓存等

**4. Android 网络通信框架Volley**

项目地址：https://android.googlesource.com/platform/frameworks/volley

我们在程序中需要和网络通信的时候，大体使用的东西莫过于AsyncTaskLoader，HttpURLConnection，AsyncTask，HTTPClient（Apache）等，在2013年的Google I/O发布了Volley。Volley是Android平台上的网络通信库，能使网络通信更快，更简单，更健壮。

特点：

(1)JSON，图像等的异步下载；

(2)网络请求的排序（scheduling）

(3)网络请求的优先级处理

(4)缓存

(5)多级别取消请求

(6)和Activity和生命周期的联动（Activity结束时同时取消所有网络请求）

### Adapter适配器

在Android中Adapter使用十分广泛，特别是在list中。所以adapter是数据的**“集散地”** ，所以对其进行内存优化是很有必要的。

下面算是一个标准的使用模版：

主要使用convertView和ViewHolder来进行缓存处理

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. @Override
2. **public** View getView(**int** position, View convertView, ViewGroup parent) {
3. ViewHolder vHolder = **null**;
4. //如果convertView对象为空则创建新对象，不为空则复用
5. **if** (convertView == **null**) {
6. convertView = inflater.inflate(..., **null**);
7. // 创建 ViewHodler 对象
8. vHolder = **new** ViewHolder();
9. vHolder.img= (ImageView) convertView.findViewById(...);
10. vHolder.tv= (TextView) convertView.findViewById(...);
11. // 将ViewHodler保存到Tag中(Tag可以接收Object类型对象，所以任何东西都可以保存在其中)
12. convertView.setTag(vHolder);
13. } **else** {
14. //当convertView不为空时，通过getTag()得到View
15. vHolder = (ViewHolder) convertView.getTag();
16. }
17. // 给对象赋值，修改显示的值
18. vHolder.img.setImageBitmap(...);
19. vHolder.tv.setText(...);
20. **return** convertView;
21. }
22. //将显示的View 包装成类
23. **static** **class** ViewHolder {
24. TextView tv;
25. ImageView img;
26. }

### 池（PooL）

**对象池：**

对象池使用的基本思路是：将用过的对象保存起来，等下一次需要这种对象的时候，再拿出来重复使用，从而在一定程度上减少频繁创建对象所造成的开销。 并非所有对象都适合拿来池化――因为维护对象池也要造成一定开销。对生成时开销不大的对象进行池化，反而可能会出现“维护对象池的开销”大于“生成新对象的开销”，从而使性能降低的情况。但是对于生成时开销可观的对象，池化技术就是提高性能的有效策略了。

**线程池：**

线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

比如：一个应用要和网络打交道，有很多步骤需要访问网络，为了不阻塞主线程，每个步骤都创建个线程，在线程中和网络交互，用线程池就变的简单，线程池是对线程的一种封装，让线程用起来更加简便，只需要创一个线程池，把这些步骤像任务一样放进线程池，在程序销毁时只要调用线程池的销毁函数即可。

java提供了**ExecutorService**和**Executors**类，我们可以应用它去建立线程池。

通常可以建立如下4种：

**[java]** [view plaincopy](http://blog.csdn.net/a396901990/article/details/38707007)

1. /\*\* 每次只执行一个任务的线程池 \*/
2. ExecutorService singleTaskExecutor =  Executors.newSingleThreadExecutor();
4. /\*\* 每次执行限定个数个任务的线程池 \*/
5. ExecutorService limitedTaskExecutor = Executors.newFixedThreadPool(3);
7. /\*\* 所有任务都一次性开始的线程池 \*/
8. ExecutorService allTaskExecutor = Executors.newCachedThreadPool();
10. /\*\* 创建一个可在指定时间里执行任务的线程池，亦可重复执行 \*/
11. ExecutorService scheduledTaskExecutor = Executors.newScheduledThreadPool(3);

更多关于线程池的内容我推荐这篇文章：<http://www.xuanyusong.com/archives/2439>

**注意：**

要根据情况适度使用缓存，因为内存有限。

能保存路径地址的就不要存放图片数据，不经常使用的尽量不要缓存，不用时就清空

## Recycle（回收）:

Recycle（回收），回收可以说是在内存使用中最重要的部分。因为内存空间有限，无论你如何优化，如何节省内存总有用完的时候。而回收的意义就在于去清理和释放那些已经闲置，废弃不再使用的内存资源和内存空间。

因为在[**Java**](http://lib.csdn.net/base/javase)中有垃圾回收（GC）机制，所以我们平时都不会太关注它，下面就来简单的介绍一下回收机制：

### 垃圾回收（GC）：

**Java垃圾回收器：**

在C，C++或其他程序设计语言中，资源或内存都必须由程序员自行声明产生和回收，否则其中的资源将消耗，造成资源的浪费甚至崩溃。但手工回收内存往往是一项复杂而艰巨的工作。

于是，Java技术提供了一个系统级的线程，即垃圾收集器线程（Garbage Collection Thread），来跟踪每一块分配出去的内存空间，当Java 虚拟机（[**Java**](http://lib.csdn.net/base/java)Virtual Machine）处于空闲循环时，垃圾收集器线程会自动检查每一快分配出去的内存空间，然后自动回收每一快可以回收的无用的内存块。

**作用：**

**1.**清除不用的对象来释放内存：

采用一种动态存储管理技术，它自动地释放不再被程序引用的对象，按照特定的垃圾收集[**算法**](http://lib.csdn.net/base/datastructure)来实现资源自动回收的功能。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用。

**2.**消除堆内存空间的碎片：

由于创建对象和垃圾收集器释放丢弃对象所占的内存空间，内存会出现碎片。碎片是分配给对象的内存块之间的空闲内存洞。碎片整理将所占用的堆内存移到堆的一端，JVM将整理出的内存分配给新的对象。

**垃圾回收器优点：**

**1.**减轻编程的负担，提高效率：

使程序员从手工回收内存空间的繁重工作中解脱了出来，因为在没有垃圾收集机制的时候，可能要花许多时间来解决一个难懂的存储器问题。在用Java语言编程的时候，靠垃圾收集机制可大大缩短时间。

**2.**它保护程序的完整性：

因此垃圾收集是Java语言安全性策略的一个重要部份。

**垃圾回收器缺点：**

**1.**占用资源时间：

Java虚拟机必须追踪运行程序中有用的对象, 而且最终释放没用的对象。这一个过程需要花费处理器的时间。

**2.**不可预知:

垃圾收集器线程虽然是作为低优先级的线程运行，但在系统可用内存量过低的时候，它可能会突发地执行来挽救内存资源。当然其执行与否也是不可预知的。

**3.**不确定性：

不能保证一个无用的对象一定会被垃圾收集器收集，也不能保证垃圾收集器在一段Java语言代码中一定会执行。

同样也没有办法预知在一组均符合垃圾收集器收集标准的对象中，哪一个会被首先收集。

**4.**不可操作

垃圾收集器不可以被强制执行，但程序员可以通过调用System. gc方法来建议执行垃圾收集器。

**垃圾回收算法：**

**1.**引用计数（Reference Counting）   
比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

**2.**标记-清除（Mark-Sweep）   
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。

**3.**复制（Copying）   
此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。次算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不过出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。

**4.**标记-整理（Mark-Compact）   
此算法结合了 “标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象 “压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

**5.**增量收集（Incremental Collecting）   
实施垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

**6.**分代（Generational Collecting）   
基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

**finalize（）:**

每一个对象都有一个finalize方法，这个方法是从Object类继承来的。

当垃圾回收确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

Java 技术允许使用finalize方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。一旦垃圾回收器准备好释放对象占用的空间，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。  
简单的说finalize方法是在垃圾收集器删除对象之前对这个对象调用的

**System.gc（）：**

我们可以调用System.gc方法，建议虚拟机进行垃圾回收工作（注意，是建议，但虚拟机会不会这样干，我们也无法预知！）

下面来看一个例子来了解**finalize（）**和**System.gc（）**的使用：

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. **public** **class** TestGC {
2. **public** TestGC() {}
4. //当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
5. **protected** **void** finalize() {
6. System.out.println("我已经被垃圾回收器回收了...");
7. }
9. **public** **static** **void** main(String [] args) {
10. TestGC gc = **new** TestGC();
11. gc = **null**;
12. // 建议虚拟机进行垃圾回收工作
13. System.gc();
14. }
15. }

如上面的例子所示，大家可以猜猜重写的finalize方法会不会执行？

答案是：**不一定**！

因为无论是设置gc的引用为null还是调用System.gc()方法都只是**"建议"**垃圾回收器进行垃圾回收，但是最终所有权还在垃圾回收器手中，它会不会进行回收我们无法预知！  
  
**垃圾回收面试题：**  
最后通过网上找到的3道面试题来结束垃圾回收的内容。

**面试题一：**

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. 1．fobj = **new** Object ( ) ;
2. 2．fobj. Method ( ) ;
3. 3．fobj = **new** Object ( ) ;
4. 4．fobj. Method ( ) ;

**问：**这段代码中，第几行的fobj 符合垃圾收集器的收集标准？   
**答：**第3行。因为第3行的fobj被赋了新值，产生了一个新的对象，即换了一块新的内存空间，也相当于为第1行中的fobj赋了null值。这种类型的题是最简单的。   
  
**面试题二：**

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. 1．Object sobj = **new** Object ( ) ;
2. 2．Object sobj = **null** ;
3. 3．Object sobj = **new** Object ( ) ;
4. 4．sobj = **new** Object ( ) ;

**问：**这段代码中，第几行的内存空间符合垃圾收集器的收集标准？   
**答：**第2行和第4行。因为第2行为sobj赋值为null，所以在此第1行的sobj符合垃圾收集器的收集标准。而第4行相当于为sobj赋值为null，所以在此第3行的sobj也符合垃圾收集器的收集标准。   
  
如果有一个对象的句柄a，且你把a作为某个构造器的参数，即 new Constructor ( a )的时候，即使你给a赋值为null，a也不符合垃圾收集器的收集标准。直到由上面构造器构造的新对象被赋空值时，a才可以被垃圾收集器收集。   
  
**面试题三：**

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. 1．Object aobj = **new** Object ( ) ;
2. 2．Object bobj = **new** Object ( ) ;
3. 3．Object cobj = **new** Object ( ) ;
4. 4．aobj = bobj;
5. 5．aobj = cobj;
6. 6．cobj = **null**;
7. 7．aobj = **null**;

**问：**这段代码中，第几行的内存空间符合垃圾收集器的收集标准？   
**答：**第4，7行。注意这类题型是认证考试中可能遇到的最难题型了。   
行1-3：分别创建了Object类的三个对象：aobj，bobj，cobj  
行4：此时对象aobj的句柄指向bobj，原来aojb指向的对象已经没有任何引用或变量指向，这时，就符合回收标准。  
行5：此时对象aobj的句柄指向cobj，所以该行的执行不能使aobj符合垃圾收集器的收集标准。   
行6：此时仍没有任何一个对象符合垃圾收集器的收集标准。   
行7：对象cobj符合了垃圾收集器的收集标准，因为cobj的句柄指向单一的地址空间。在第6行的时候，cobj已经被赋值为null，但由cobj同时还指向了aobj（第5行），所以此时cobj并不符合垃圾收集器的收集标准。而在第7行，aobj所指向的地址空间也被赋予了空值null，这就说明了，由cobj所指向的地址空间已经被完全地赋予了空值。所以此时cobj最终符合了垃圾收集器的收集标准。 但对于aobj和bobj，仍然无法判断其是否符合收集标准。   
  
总之，在Java语言中，判断一块内存空间是否符合垃圾收集器收集的标准只有两个：   
**1．**给对象赋予了空值null，以下再没有调用过。   
**2．**给对象赋予了新值，既重新分配了内存空间。   
  
最后再次提醒一下，一块内存空间符合了垃圾收集器的收集标准，并不意味着这块内存空间就一定会被垃圾收集器收集。

### 资源的回收：

刚才讲了一堆理论的东西，下面来点实际能用上的，资源的回收：

* **Thread（线程）回收：**

线程中涉及的任何东西GC都不能回收（Anything reachable by a thread cannot be GC'd ），所以线程很容易造成内存泄露。

如下面代码所示：

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. Thread t = **new** Thread() {
2. **public** **void** run() {
3. **while** (**true**) {
4. **try** {
5. Thread.sleep(1000);
6. System.out.println("thread is running...");
7. } **catch** (InterruptedException e) {
9. }
10. }
11. }
12. };
13. t.start();
14. t = **null**;
15. System.gc();

如上在线程t中每间隔一秒输出一段话，然后将线程设置为null并且调用System.gc方法。

最后的结果是线程并不会被回收，它会一直的运行下去。

因为运行中的线程是称之为垃圾回收根（GC Roots）对象的一种，不会被垃圾回收。当垃圾回收器判断一个对象是否可达，总是使用垃圾回收根对象作为参考点。

* **Cursor（游标）回收：**

Cursor是Android查询数据后得到的一个管理数据集合的类，在使用结束以后。应该保证Cursor占用的内存被及时的释放掉，而不是等待GC来处理。并且Android明显是倾向于编程者手动的将Cursor close掉，因为在源代码中我们发现，如果等到垃圾回收器来回收时，会给用户以错误提示。

所以我们使用Cursor的方式一般如下：

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. Cursor cursor = **null**;
2. **try** {
3. cursor = mContext.getContentResolver().query(uri,**null**, **null**,**null**,**null**);
4. **if**(cursor != **null**) {
5. cursor.moveToFirst();
6. //do something
7. }
8. } **catch** (Exception e) {
9. e.printStackTrace();
10. } **finally** {
11. **if** (cursor != **null**) {
12. cursor.close();
13. }
14. }

有一种情况下，我们不能直接将Cursor关闭掉，这就是在CursorAdapter中应用的情况，但是注意，CursorAdapter在Acivity结束时并没有自动的将Cursor关闭掉，因此，你需要在onDestroy函数中，手动关闭。

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. @Override
2. **protected** **void** onDestroy() {
3. **if** (mAdapter != **null** && mAdapter.getCurosr() != **null**) {
4. mAdapter.getCursor().close();
5. }
6. **super**.onDestroy();
7. }

* **Receiver（接收器）回收**

调用registerReceiver()后未调用unregisterReceiver().

当我们Activity中使用了registerReceiver()方法注册了BroadcastReceiver，一定要在Activity的生命周期内调用unregisterReceiver()方法取消注册   
也就是说registerReceiver()和unregisterReceiver()方法一定要成对出现，通常我们可以重写Activity的onDestory()方法：

**[java]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. @Override
2. **protected** **void** onDestroy() {
3. **this**.unregisterReceiver(receiver);
4. **super**.onDestroy();
5. }

* **Stream/File（流/文件）回收：**

主要针对各种流，文件资源等等如：

InputStream/OutputStream，SQLiteOpenHelper，SQLiteDatabase，Cursor，文件，I/O，Bitmap图片等操作等都应该记得显示关闭。  
和之前介绍的Cursor道理类似，就不多说了。

## Review:

Review（回顾，检查），大家都知道Code Review的重要性。而这里我说的Review和Code Review差不多，主要目的就是检查代码中存在的不合理和可以改进的地方，当然这个Review需要大家自己来做啦。

### Code Review（代码检查）：

Code Review主要检查代码中存在的一些不合理或可以改进优化的地方，大家可以参考之前写的Reduce，Reuse和Recycle都是侧重讲解这方面的。

### UI Review（视图检查）：

Android对于视图中控件的布局渲染等会消耗很多的资源和内存，所以这部分也是我们需要注意的。

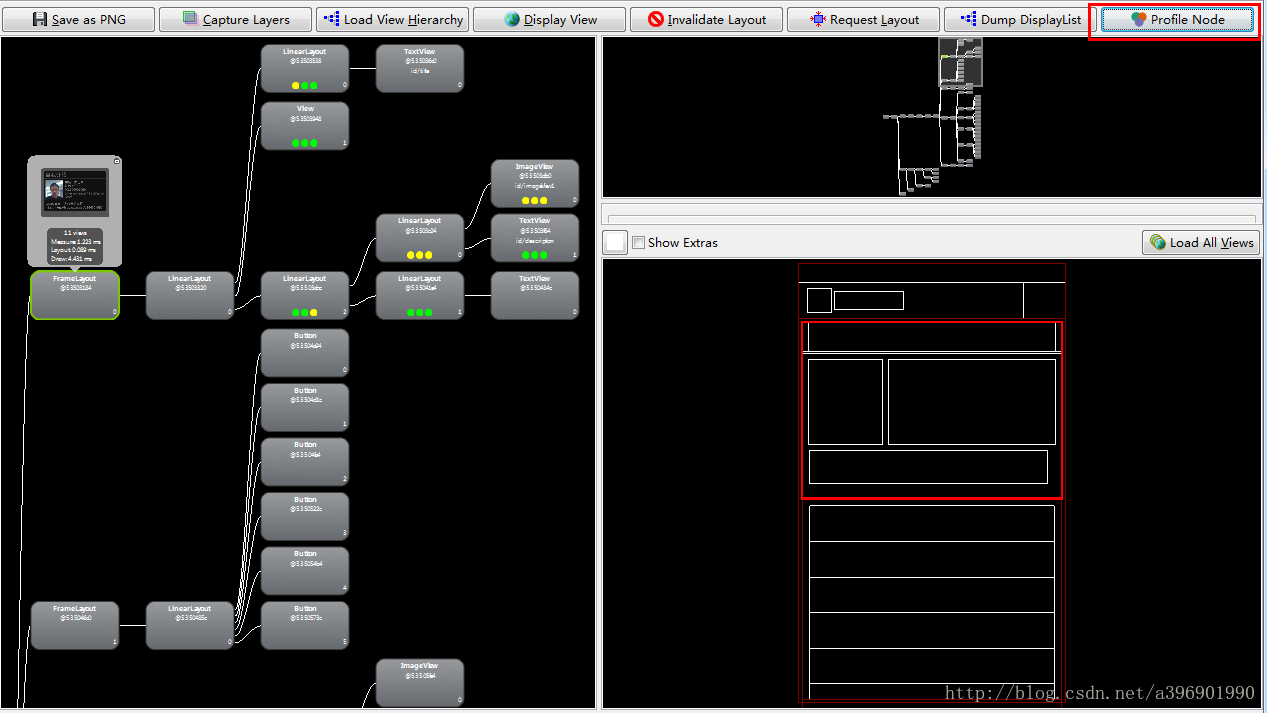
* **减少视图层级：**

减少视图层级可以有效的减少内存消耗，因为视图是一个树形结构，每次刷新和渲染都会遍历一次。

**hierarchyviewer：**

想要减少视图层级首先就需要知道视图层级，所以下面介绍一个SDK中自带的一个非常好用的工具hierarchyviewer。

你可以在下面的地址找到它：**your sdk path\sdk\tools**



如上图大家可以看到，hierarchyviewer可以非常清楚的看到当前视图的层级结构，并且可以查看视图的执行效率（视图上的小圆点，绿色表示流畅，黄色和红色次之），所以我们可以很方便的查看哪些view可能会影响我们的性能从而去进一步优化它。

hierarchyviewer还提供另外一种列表式的查看方式，可以查看详细的屏幕画面，具体到像素级别的问题都可以通过它发现。

**ViewStub标签**

此标签可以使UI在特殊情况下，直观效果类似于设置View的不可见性，但是其更大的意义在于被这个标签所包裹的Views在默认状态下不会占用任何内存空间。

**include标签**

可以通过这个标签直接加载外部的xml到当前结构中，是复用UI资源的常用标签。

**merge标签**

它在优化UI结构时起到很重要的作用。目的是通过删减多余或者额外的层级，从而优化整个Android Layout的结构。

（注意：灵活运用以上3个标签可以有效减少视图层级，具体使用大家可以上网搜搜）

**布局用Java代码比写在XML中快**

一般情况下对于Android程序布局往往使用XML文件来编写，这样可以提高开发效率，但是考虑到代码的安全性以及执行效率，可以通过Java代码执行创建，虽然Android编译过的XML是二进制的，但是加载XML解析器的效率对于资源占用还是比较大的，Java处理效率比XML快得多，但是对于一个复杂界面的编写，可能需要一些套嵌考虑，如果你思维灵活的话，使用Java代码来布局你的Android应用程序是一个更好的方法。

* **重用系统资源：**

**1. 利用系统定义的id**

比如我们有一个定义ListView的xml文件，一般的，我们会写类似下面的代码片段。

**[html]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. **<ListView**
2. android:id="@+id/mylist"
3. android:layout\_width="fill\_parent"
4. android:layout\_height="fill\_parent"**/>**

这里我们定义了一个ListView，定义它的id是"@+id/mylist"。实际上，如果没有特别的需求，就可以利用系统定义的id，类似下面的样子。

**[html]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. **<ListView**
2. android:id="@android:id/list"
3. android:layout\_width="fill\_parent"
4. android:layout\_height="fill\_parent"**/>**

在xml文件中引用系统的id，只需要加上“@android:”前缀即可。如果是在Java代码中使用系统资源，和使用自己的资源基本上是一样的。不同的是，需要使用android.R类来使用系统的资源，而不是使用应用程序指定的R类。这里如果要获取ListView可以使用android.R.id.list来获取。

**2. 利用系统的图片资源**

这样做的好处，一个是美工不需要重复的做一份已有的图片了，可以节约不少工时；另一个是能保证我们的应用程序的风格与系统一致。

**3. 利用系统的字符串资源**

如果使用系统的字符串，默认就已经支持多语言环境了。如上述代码，直接使用了@android:string/yes和@android:string/no，在简体中文环境下会显示“确定”和“取消”，在英文环境下会显示“OK”和“Cancel”。

**4. 利用系统的Style**

 假设布局文件中有一个TextView，用来显示窗口的标题，使用中等大小字体。可以使用下面的代码片段来定义TextView的Style。

**[html]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. **<TextView**
2. android:id="@+id/title"
3. android:layout\_width="wrap\_content"
4. android:layout\_height="wrap\_content"
5. android:textAppearance="?android:attr/textAppearanceMedium" **/>**

其中android:textAppearance="?android:attr/textAppearanceMedium"就是使用系统的style。需要注意的是，使用系统的style，需要在想要使用的资源前面加“?android:”作为前缀，而不是“@android:”。

**5. 利用系统的颜色定义**

除了上述的各种系统资源以外，还可以使用系统定义好的颜色。在项目中最常用的，就是透明色的使用。

**[html]** [view plain](http://blog.csdn.net/a396901990/article/details/38904543) [copy](http://blog.csdn.net/a396901990/article/details/38904543)

[print?](http://blog.csdn.net/a396901990/article/details/38904543)

1. android:background ="@android:color/transparent"

除了上面介绍的以外还有很多其他Android系统本身自带的资源，它们在应用中都可以直接使用。具体的，可以进入android-sdk的相应文件夹中去查看。例如：可以进入$android-sdk$\platforms\android-8\data\res，里面的系统资源就一览无余了。

开发者需要花一些时间去熟悉这些资源，特别是图片资源和各种Style资源，这样在开发过程中，能重用的尽量重用，而且有时候使用系统提供的效果可能会更好。

* 其他小tips：

**1.**分辨率适配-ldpi,-mdpi, -hdpi配置不同精度资源，系统会根据设备自适应，包括drawable, layout,style等不同资源。

**2.**尽量使用dp(density independent pixel)开发，不用px(pixel)。

**3.**多用wrap\_content, match\_parent

**4.**永远不要使用AbsoluteLayout

**5.**使用9patch（通过~/tools/draw9patch.bat启动应用程序），png格式

**6.**将Acitivity中的Window的背景图设置为空。getWindow().setBackgroundDrawable(null);android的默认背景是不是为空。

**7.**View中设置缓存属性.setDrawingCache为true。

### Desgin Review（设计检查）：

Desgin Review主要侧重检查一下程序的设计是否合理，包括框架的设计，界面的设计，逻辑的设计(其实这些东西开发之前就应该想好了)。

* **框架设计：**

是否定义了自己的Activity和fragment等常用控件的基类去避免进行重复的工作

是否有完善的异常处理机制，即使真的出现OOM也不会直接崩溃导致直接退出程序

* **界面设计：**

**1.**在视图中加载你所需要的，而不是你所拥有。因为用户不可能同时看到所有东西。最典型的例子就是ListView中的滑动加载。

**2.**如果数据特别大，此时应该暗示用户去点击加载，而不是直接加载。

**3.**合理运用分屏，转屏等，它是个双刃剑，因为它即可以使程序更加美观功能更加完善，但也相应增加了资源开销。

* **逻辑设计：**

避免子类直接去控制父类中内容，可以使用监听等方式去解决