**Java基础部分面试问题**

* **Java基础：强引用、弱引用、软引用、虚引用**

1. 强引用：强引用是使用最普遍的引用。如果一个对象具有强引用，那么垃圾回收器不会回收。当内存空间不足，Java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的问题。
2. 软引用：如果一个对象只具有软引用，则内存空间足够，垃圾回收器不会回收；如果内存空间不足，就会回收这些对象内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。（软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。）
3. 弱引用：弱引用与软引用的区别在于：只具有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。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。（如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。   ）
4. 虚引用：“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器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

* **HashMap原理**

在HashMap中有两个很重要的参数，容量(Capacity)和负载因子(Load factor)。简单的说，Capacity就是bucket的大小，Load factor就是bucket填满程度的最大比例。如果对迭代性能要求很高的话不要把capacity设置过大，也不要把load factor设置过小。当bucket中的entries的数目大于capacity\*load factor时就需要调整bucket的大小为当前的2倍。

**put函数大致的思路为：**

1. 对key的hashCode()做hash，然后再计算index（对key做null检查。如果key是null，会被存储到table[0]，因为null的hash值总是0）;
2. 如果没碰到直接放到bucket里；
3. 如果碰撞了，以链表的形式存在buckets后；
4. 如果碰到导致链表过长(大于等于TREEIFY\_THRESHOLD)，就把链表转换成红黑树；
5. 如果节点已经存在就替换old value(保证key的唯一性)
6. 如果bucket满了(超过load factor\*current capacity)，就要resize。

get函数的实现：

1. 对key进行hash计算在table的索引
2. 获取到table数组索引后，会迭代查找对应的entry。若为树，则在树中通过key.equals(k)查找，若为链表，则在链表中通过key.equals(k)查找。

* ArrayList与Linklist区别

1. ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。 （LinkedList是双向链表，有next也有previous）
2. 对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3. 对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。
4. 所以当插入的数据量很小时，两者区别不太大，当插入的数据量大时，大约在容量的1/10之前，LinkedList会优于ArrayList，在其后就劣与ArrayList，且越靠近后面越差。所以个人觉得，一般首选用ArrayList，由于LinkedList可以实现栈、队列以及双端队列等数据结构，所以当特定需要时候，使用LinkedList，当然咯，数据量小的时候，两者差不多，视具体情况去选择使用；当数据量大的时候，如果只需要在靠前的部分插入或删除数据，那也可以选用LinkedList，反之选择ArrayList反而效率更高。

* **Java容器（集合类）**

Java容器类类库的用途是“持有对象”，并将其划分为两个不同的概念

１）Collection：一个独立元素的序列，这些元素都服从一条或者多条规则。 List必须按照插入的顺序保存元素，而set不能有重复的元素。Queue按照排队规则来确定对象产生的顺序（通常与它们被插入的顺序相同）。

2）Map：一组成对的“键值对”对象，允许你使用键来查找值。

1. **List接口：**
2. **LinkedList类**   
   　　LinkedList实现了List接口，允许null元素。此外LinkedList提供额外的get，remove，insert方法在 LinkedList的首部或尾部。这些操作使LinkedList可被用作堆栈（stack），队列（queue）或双向队列（deque）。
3. **ArrayList类**   
   　　ArrayList实现了可变大小的数组。它允许所有元素，包括null。ArrayList没有同步。size，isEmpty，get，set方法运行时间为常数。但是add方法开销为分摊的常数，添加n个元素需要O(n)的时间。其他的方法运行时间为线性。每个ArrayList实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并 没有定义。当需要插入大量元素时，在插入前可以调用ensureCapacity方法来增加ArrayList的容量以提高插入效率。
4. **Vector类**   
   　　Vector非常类似ArrayList，但是Vector是同步的。由Vector创建的Iterator，虽然和ArrayList创建的 Iterator是同一接口，但是，因为Vector是同步的，当一个 Iterator被创建而且正在被使用，另一个线程改变了Vector的状态（例 如，添加或删除了一些元素），这时调用Iterator的方法时将抛出 ConcurrentModificationException，因此必须捕获该 异常。
5. **Stack 类**  
   　　Stack继承自Vector，实现一个后进先出的堆栈。Stack提供5个额外的方法使得Vector得以被当作堆栈使用。基本的push和pop方 法，还有 peek方法得到栈顶的元素，empty方法测试堆栈是否为空，search方法检测一个元素在堆栈中的位置。Stack刚创建后是空栈。
6. **Set接口**

Set是一种不包含重复的元素的Collection，即任意的两个元素e1和e2都有e1.equals(e2)=false，Set最多有一个null元素。 Set的构造函数有一个约束条件，传入的Collection参数不能包含重复的元素。 Set容器类主要有HashSet和TreeSet等。

**1）HashSet类**   
　 　Java.util.HashSet类实现了Java.util.Set接口。   
　　-> 它不允许出现重复元素；   
　　-> 不保证和政集合中元素的顺序   
　　->允许包含值为null的元素，但最多只能有一个null元素。

**2）TreeSet**   
　　TreeSet描述的是Set的一种变体——可以实现排序等功能的集合，它在讲对象元素添加到集合中时会自动按照某种比较规则将其插入到有序的对象序列中。

1. **Map集合接口**

Map没有继承Collection接口，Map提供key到value的映射。一个Map中不能包含相同的key，每个key只能映射一个 value。Map接口提供3种集合的视图，Map的内容可以被当作一组key集合，一组value集合，或者一组key-value映射。

**1、Hashtable类**   
　　Hashtable继承Map接口，实现一个key-value映射的哈希表。任何非空（non-null）的对象都可作为key或者value。\*\*添加数据使用put(key, value)，取出数据使用get(key)，这两个基本操作的时间开销为常数。\*\*Hashtable通过initial capacity和load factor两个参数调整性能。通常缺省的load factor 0.75较好地实现了时间和空间的均衡。增大load factor可以节省空间但相应的查找时间将增大，这会影响像get和put这样的操作。

　　由于作为key的对象将通过计算其散列函数来确定与之对应的value的位置，因此任何作为key的对象都必须实现hashCode和equals方法。hashCode和equals方法继承自根类Object，如果你用自定义的类当作key的话，要相当小心，按照散列函数的定义，如果两个对象相同，即obj1.equals(obj2)=true，则它们的hashCode必须相同，但如果两个对象不同，则它们的hashCode不一定不同，如果两个不同对象的hashCode相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的hashCode()方法，能加快哈希表的操作。

　　如果相同的对象有不同的hashCode，对哈希表的操作会出现意想不到的结果（期待的get方法返回null），要避免这种问题，只需要牢记一条：要同时复写equals方法和hashCode方法，而不要只写其中一个。

**２、HashMap类**   
　　HashMap和Hashtable类似，不同之处在于HashMap是非同步的，并且允许null，即null value和null key，但是将HashMap视为Collection时（values()方法可返回Collection），其迭代子操作时间开销和HashMap的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将HashMap的初始化容量设得过高，或者load factor过低。

－JDK1.0引入了第一个关联的集合类HashTable，它是线程安全的。 HashTable的所有方法都是同步的。

－JDK2.0引入了HashMap，它提供了一个不同步的基类和一个同步的包装器synchronizedMap。synchronizedMap被称为 有条件的线程安全类。

－JDK5.0util.concurrent包中引入对Map线程安全的实现ConcurrentHashMap，比起synchronizedMap， 它提供了更高的灵活性。同时进行的读和写操作都可以并发地

**３、WeakHashMap类**   
　　WeakHashMap是一种改进的HashMap，它对key实行“弱引用”，如果一个key不再被外部所引用，那么该key可以被GC回收。

* **Hashcode与equals比较**

一般在覆盖equals()方法的同时也要覆盖hashCode()方法，否则，就会违反Object.hashCode的通用约定，从而导致该类无法与所有基于散列值(hash)集合类(HashMap、HashSet和Hashtable)结合在一起正常运行。

**HashCode：**哈希码产生的依据：哈希码并不是完全唯一的，它是一种算法，让同一个类的对象按照自己不同的特征尽量的有不同的哈希码，但不表示不同的对象哈希码完全不同。也有相同的情况，看程序员如何写哈希码的算法。

**Equals:** 默认情况（没有覆盖equals方法）下equals方法都是调用Object类的equals方法，而Object的equals方法主要用于判断对象的内存地址引用是不是同一个地址（是不是同一个对象）;

**hashCode()方法的返回值和equals()方法的关系如下：**

x.equals(y)返回true，即两个对象根据equals()方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode()方法都必须产生同样的整数结果。如果x.equals(y)返回false，即两个对象根据equals()方法比较是不相等的，那么x和y的hashCode()方法的返回值有可能相等，也有可能不相等。反之，hashCode()方法的返回值不相等，一定能推出equals()方法的返回值也不相等，而hashCode()方法的返回值相等，equals()方法的返回值则可能相等，也可能不相等。

要是类中覆盖了equals方法，那么就要根据具体的代码来确定equals方法的作用了，覆盖后一般都是通过对象的内容是否相等来判断对象是否相等。

* **内部类与静态内部类区别**

静态内部类的作用：只是为了降低包的深度，方便类的使用，静态内部类适用于包含类当中，但又不依赖与外在的类，不用使用外在类的非静态属性和方法，只是为了方便管理类结构而定义。在创建静态内部类的时候，不需要外部类对象的引用。

非静态内部类有一个很大的优点：可以自由使用外部类的所有变量和方法

**内部类中的this：**内部类中的this与其他类一样是指的本身。创建内部类对象时，它会与创造它的外围对象有了某种联系，于是能访问外围类的所有成员，不需要任何特殊条件，可理解为内部类链接到外部类。用外部类创建内部类对象时，此内部类对象会秘密的捕获一个指向外部类的引用，于是，可以通过这个引用来访问外围类的成员。

**外部类访问内部类：**内部类类似外部类的属性，因此访问内部类对象时总是需要一个创建好的外部对象。内部类对象通过‘外部类.this.xxx’的形式访问外部类的属性与方法。

**内部类向上转型：**内部类也可以和普通类一样拥有向上转型的特性。将内部类向上转型，尤其是接口，内部类就有了用武之地。如果内部类是private的，只可以被它的外部类访问，从而完全隐藏实现细节。

**方法内的类：**方法内创建的类（注意方法中也能定义类），不能加访问修饰符。另外，方法内部的类也不是在调用方法时才会创建的，它们一样也被事先编译了。

**静态内部类**

 定义静态内部类：在定义内部类的时候，可以在其前面加上一个权限修饰符static。此时这个内部类就变为了静态内部类。

通常称为嵌套类，当内部类是static时，意味着：

   [1]要创建嵌套类的对象，并不需要其外围类的对象；

   [2]不能从嵌套类的对象中访问非静态的外围类对象（不能够从静态内部类的对象中访问外部类的非静态成员）；

   嵌套类与普通的内部类还有一个区别：普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有static数据和static字段， 也不能包含嵌套类。但是在嵌套类里可以包含所有这些东西。也就是说，在非静态内部类中不可以声明静态成员，只有将某个内部类修饰为静态类，然后才能够在这 个类中定义静态的成员变量与成员方法。

   另外，在创建静态内部类时不需要将静态内部类的实例绑定在外部类的实例上。普通非静态内部类的 对象是依附在外部类对象之中的，要在一个外部类中定义一个静态的内部类，不需要利用关键字new来创建内部类的实例。静态类和方法只属于类本身，并不属于 该类的对象，更不属于其他外部类的对象。

**内部类标识符**

每个类会产生一个.class文件，文件名即为类名。同样，内部类也会产生这么一个.class文件，但是它的名称却不是内部类的类名，而是有着严格的限制：外围类的名字，加上$,再加上内部类名字。

**为何要用内部类？**

内部类一般只为其外部类使用；

内部类提供了某种进入外部类的窗户；

也是最吸引人的原因，每个内部类都能独立地继承一个接口，而无论外部类是否已经继承了某个接口。因此，内部类使多重继承的解决方案变得更加完整。

* **GC的几种算法**

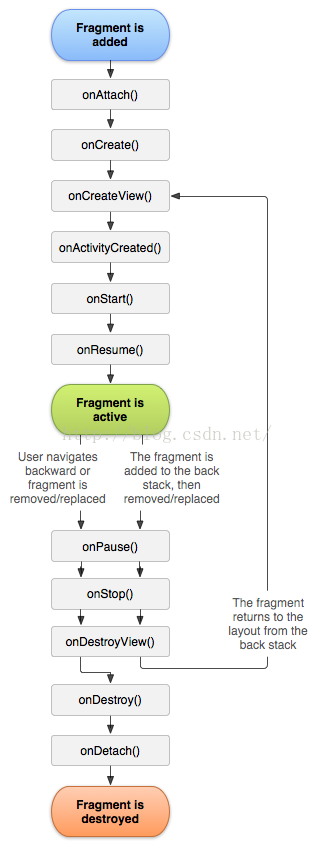
**Android部分面试**

**基础部分**

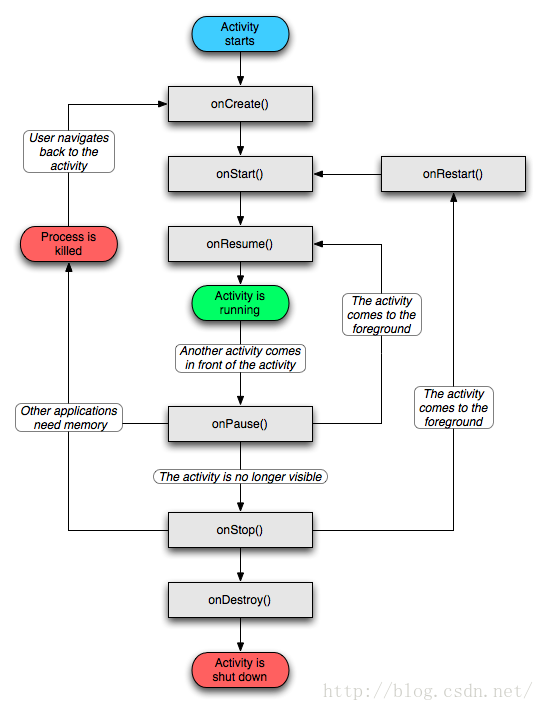
1. **Android四大组件：**Activity、service、broadcast、contentprovider。
2. **Android六大布局：**线性布局(LinearLayout)、框架布局(FrameLayout)、表格布局(TableLayout)、相对布局(RelativeLayout)、绝对布局(AbsoluteLayout)和网格布局(GridLayout) 。
3. **Android 进程间通信：**aidl、localSocket、Broadcast、Messenger。
4. **onSaveInstanceState()**：方法保存Activity状态。
5. **Android五大存储：**SharedPreferences、文件存储、SQLite[**数据库**](http://lib.csdn.net/base/14)方式、内容提供器（Content provider）和网络。
6. **assets目录：**是Android的一种特殊目录，用于放置APP所需的固定文件，且该文件被打包到APK中时，不会被编码到二进制文件。可以放一些资源文件，例如数据库文件，图片文件，语音文件等。主要放置多媒体等数据文件。
7. **Android五大应用开发框架是** ：Application Framework. 其系统架构由5部分组成，分别是： Linux Kernel、Android Runtime、Libraries、Application Framework、Applications。
8. **Activity的四种启动模式及其应用场景**
9. **Standard：**该模式是默认的启动模式，不用为配置android:launchMode属性即可，当然也可以指定值为standard。每次跳转系统都会在task中生成一个新的MainActivity实例， 并且放于栈结构顶部，当我们按下后退键时，才能看到原来的MainActivity实例。
10. **singleTop：** 如果发现有对应的Activity实例正位于栈顶，则重复利用，不再生成新的实例，这种启动模式通常适用于接受到消息厚显示的界面，例如QQ接受到消息后弹出Activity，如果一次来10条消息，总不能一次弹10个Activity。
11. **singleTask：** 如果发现有对应的Activity实例，则使此Activity实例之上的其他Activity实例统统出栈，使此Activity实例成为栈顶对象，显示到幕前。
12. **singleInstance：** 我们看到从MainActivity跳转到SecondActivity时，重新启用了一个新的栈结构，来放置SecondActivity实例，然后按下后退键，再次回到原始栈结构；图中下半部分显示的在SecondActivity中再次跳转到MainActivity，这个时候系统会在原始栈结构中生成一个MainActivity实例，然后回退两次，注意，并没有退出，而是回到了SecondActivity，为什么呢？是因为从SecondActivity跳转到MainActivity的时候，我们的起点变成了SecondActivity实例所在的栈结构，这样一来，我们需要“回归”到这个栈结构。

**应用场景：**

1. singleTop： 适合接收通知启动的内容显示页面。例如，某个新闻客户端的新闻内容页面，如果收到10个新闻推送，每次都打开一个新闻内容页面是很烦人的。
2. singleTask：适合作为程序入口点。例如浏览器的主界面。不管从多少个应用启动浏览器，只会启动主界面一次，其余情况都会走onNewIntent，并且会清空主界面上面的其他页面。之前打开过的页面，打开之前的页面就ok，不再新建。
3. singleInstance：适合需要与程序分离开的页面。例如闹铃提醒，将闹铃提醒与闹铃设置分离。singleInstance不要用于中间页面，如果用于中间页面，跳转会有问题，比如：A -> B (singleInstance) -> C，完全退出后，在此启动，首先打开的是B。
4. Fragment生命周期：



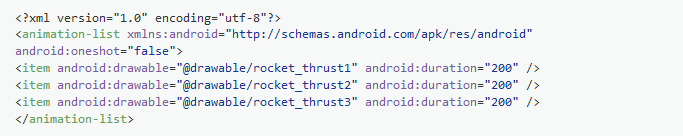
1. Activity生命周期：



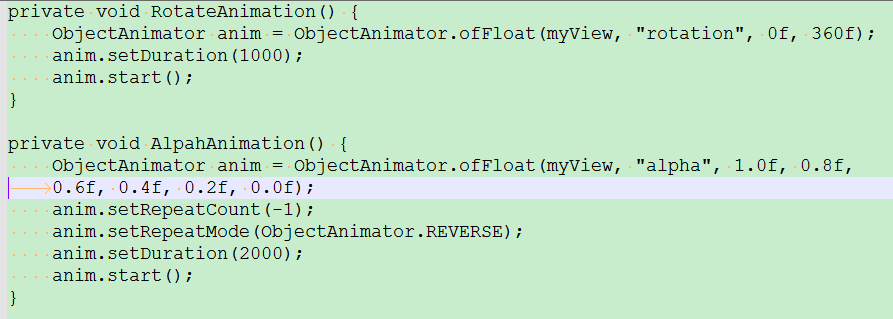
**Android动画**

**Android动画分类：**

1. **帧动画：**帧动画是最容易实现的一种动画，这种动画更多的依赖于完善的UI资源，他的原理就是将一张张单独的图片连贯的进行播放，从而在视觉上产生一种动画的效果；有点类似于某些软件制作gif动画的方式。



1. **补间动画：alpha（淡入淡出）、translate（位移）、scale（缩放大小）、rotate(旋转)**
2. **属性动画：属性动画，顾名思义它是对于对象属性的动画。因此，所有补间动画的内容，都可以通过属性动画实现。**



**帧动画与补间动画的缺点：**

1. **作用对象局限：View**

即补间动画 只能够作用在视图View上，即只可以对一个Button、TextView、甚至是LinearLayout、或者其它继承自View的组件进行动画操作，但无法对非View的对象进行动画操作

1. **没有改变View的属性，只是改变视觉效果**

补间动画只是改变了View的视觉效果，而不会真正去改变View的属性。如，将屏幕左上角的按钮 通过补间动画 移动到屏幕的右下角点击当前按钮位置（屏幕右下角）是没有效果的，因为实际上按钮还是停留在屏幕左上角，补间动画只是将这个按钮绘制到屏幕右下角，改变了视觉效果而已。

1. **动画效果单一:**

补间动画只能实现平移、旋转、缩放 & 透明度这些简单的动画需求一旦遇到相对复杂的动画效果，即超出了上述4种动画效果，那么补间动画则无法实现。即在功能 & 可扩展性有较大局限性

**属性动画特点**

**作用对象：**任意 Java 对象,不再局限于 视图View对象；

**实现的动画效果：**可自定义各种动画效果,不再局限于4种基本变换：平移、旋转、缩放 & 透明度；

**特点：**

* 作用对象进行了扩展：不只是View对象，甚至没对象也可以
* 动画效果：不只是4种基本变换，还有其他动画效果
* 作用领域：API11后引入的

# 工作原理

* 在一定时间间隔内，通过不断对值进行改变，并不断将该值赋给对象的属性，从而实现该对象在该属性上的动画效果

**ValueAnimator类**

实现动画的原理： 通过不断控制 值 的变化，再不断 手动 赋给对象的属性，从而实现动画效果。

ValueAnimator类中有3个重要方法：ValueAnimator.ofInt（int values）; ValueAnimator.ofFloat（float values）;ValueAnimator.ofObject（int values）。

估值器（TypeEvaluator）介绍：设置动画如何从初始值过渡到结束值的逻辑。

插值器（Interpolator）决定值的变化模式（匀速、加速）

估值器（TypeEvaluator）决定值的具体变化数值

**android 自定义View**

* **View的测量模式**

**1、EXACTLY**

精确值模式，当我们的控件的layout\_width属性和layout\_height属性指定为具体数值时，如android:layout\_width="100dp",或者为match\_parent时系统使用的是EXACTLY模式。

**2、AT\_MOST**

最大值模式，当控件layout\_width属性和layout\_height属性为wrap\_content时，控件大小随控件子控件或内容变化，此时控件的尺寸只要不超过父控件允许的最大尺寸即可。

**3、UNSPECIFIED**

不指定大小测量模式。

**Android proguard规则以及用法**

**Android性能部分**

* **Android好的app所要达到的要求**

1. 好的应用定义包含三个方面：业务/功能、符合逻辑的交互、优秀的性能。
2. 可以把用户体验的性能问题主要总结为四个类别：流畅、稳定、省点省流量、安装包小。

* **OOM出现的常见原因以及解决办法**

**OOM产生的本质：**Dalivk VM主要管理的内存Java heap，由于手机设备限制一般一个应用使用的内存不能超过默认值32M(不同设备略有差异，通过ad shell | grep davlik.vm.heapgrowthlimit命令查看)，这也就是说，当在DVM上申请的堆内存大于默认值的时候，我们的应用就会抛出out of memory error。

**OOM产生的原因：**

1. **加载图片导致内存溢出；**
2. **大量的内存泄漏；**
3. **数据库的cursor没有关闭；**
4. **构造adapter没有使用缓存contentview；**
5. **调用registerReceiver()后未调用unregisterReceiver()；**
6. **未关闭InputStream/OutputStream；**
7. **Bitmap使用后未调用recycle()；**
8. **Context泄漏;**
9. **线程造成的内存泄漏（**合理安排线程执行的时间，控制线程在Activity结束前结束。
10. **将内部类改为静态内部类，**并使用弱引用WeakReference来保存Activity实例 因为弱引用 只要GC发现了 就会回收它 ，因此可尽快回收**）**
11. **Handler的使用造成的内存泄漏（**由于在Handler的使用中，handler会发送message对象到 MessageQueue中 然后 Looper会轮询MessageQueue 然后取出Message执行，但是如果一个Message长时间没被取出执行，那么由于 Message中有 Handler的引用，而 Handler 一般来说也是内部类对象，Message引用 Handler ，Handler引用 Activity 这样 使得 Activity无法回收。**）**
12. **static变量引起的内存泄漏**

因为static变量的生命周期是在类加载时开始 类卸载时结束，也就是说static变量是在程序进程死亡时才释放，如果在static变量中 引用了Activity 那么 这个Activity由于被引用，便会随static变量的生命周期一样，一直无法被释放，造成内存泄漏。

解决办法：在Activity被静态变量引用时，使用 getApplicationContext 因为Application生命周期从程序开始到结束，和static变量的一样。

**如何解决和避免OOM:**

1. 解决大图片导致内存溢出：使用软引用、弱引用。
2. 当内存不足的时候，就可以自动释放这些缓存的Bitmap对象；
3. 使用过的图并且不再使用，可以调用Bitmap.recycle()加速回收；
4. 考虑使用文件缓存，得到bitmap之前先利用BitmapFactory.Options的inSampleSize的值得到压缩图片；
5. 只加载部分图片。
6. **节制地使用Service**

当我们启动一个Service时，系统会倾向于将这个Service所依赖的进程进行保留，这样就会导致这个进程变得非常消耗内存。并且，系统可以在LRU cache当中缓存的进程数量也会减少，导致切换应用程序的时候耗费更多性能。严重的话，甚至有可能会导致崩溃，因为系统在内存非常吃紧的时候可能已无法维护所有正在运行的Service所依赖的进程了。

为了能够控制Service的生命周期，Android官方推荐的最佳解决方案就是使用IntentService，这种Service的最大特点就是当后台任务执行结束后会自动停止，从而极大程度上避免了Service内存泄漏的可能性。

1. **当界面不可见的释放内存**

当用户打开了另外一个程序，我们的程序界面已经不再可见的时候，我们应当将所有和界面相关的资源进行释放。重写onTrimMemory()方法，然后在这个方法中监听TRIM\_MEMORY\_UI\_HIDDEN这个级别，一旦触发了之后就说明用户已经离开了我们的程序，那么此时就可以进行资源释放操作了。

1. **当内存紧张时释放内存**

除了刚才讲的TRIM\_MEMORY\_UI\_HIDDEN这个回调，onTrimMemory()方法还有很多种其它类型的回调，可以在手机内存降低的时候及时通知我们。我们应该根据回调中传入的级别来去决定如何释放应用程序的资源：

**TRIM\_MEMORY\_RUNNING\_MODERATE**：表示应用程序正常运行，并且不会被杀掉。但是目前手机的内存已经有点低了，系统可能会开始根据LRU缓存规则来去杀死进程了。

**TRIM\_MEMORY\_RUNNING\_LOW：**表示应用程序正常运行，并且不会被杀掉。但是目前手机的内存已经非常低了，我们应该去释放掉一些不必要的资源以提升系统的性能，同时这也会直接影响到我们应用程序的性能。

**TRIM\_MEMORY\_RUNNING\_CRITICAL**：表示应用程序仍然正常运行，但是系统已经根据LRU缓存规则杀掉了大部分缓存的进程了。这个时候我们应当尽可能地去释放任何不必要的资源，不然的话系统可能会继续杀掉所有缓存中的进程，并且开始杀掉一些本来应当保持运行的进程，比如说后台运行的服务。

以上是当我们的应用程序正在运行时的回调，那么如果我们的程序目前是被缓存的，则会收到以下几种类型的回调：

**TRIM\_MEMORY\_BACKGROUND**：表示手机目前内存已经很低了，系统准备开始根据LRU缓存来清理进程。这个时候我们的程序在LRU缓存列表的最近位置，是不太可能被清理掉的，但这时去释放掉一些比较容易恢复的资源能够让手机的内存变得比较充足，从而让我们的程序更长时间地保留在缓存当中，这样当用户返回我们的程序时会感觉非常顺畅，而不是经历了一次重新启动的过程。

**TRIM\_MEMORY\_MODERATE**：表示手机目前内存已经很低了，并且我们的程序处于LRU缓存列表的中间位置，如果手机内存还得不到进一步释放的话，那么我们的程序就有被系统杀掉的风险了。

**TRIM\_MEMORY\_COMPLETE：**表示手机目前内存已经很低了，并且我们的程序处于LRU缓存列表的最边缘位置，系统会最优先考虑杀掉我们的应用程序，在这个时候应当尽可能地把一切可以释放的东西都进行释放。

1. **避免在Bitmap上浪费内存**

当我们读取一个Bitmap图片的时候，有一点一定要注意，就是千万不要去加载不需要的分辨率。在一个很小的ImageView上显示一张高分辨率的图片不会带来任何视觉上的好处，但却会占用我们相当多宝贵的内存。需要仅记的一点是，将一张图片解析成一个Bitmap对象时所占用的内存并不是这个图片在硬盘中的大小，可能一张图片只有100k你觉得它并不大，但是读取到内存当中是按照像素点来算的，比如这张图片是1500\*1000像素，使用的ARGB\_8888颜色类型，那么每个像素点就会占用4个字节，总内存就是1500\*1000\*4字节，也就是5.7M，这个数据看起来就比较恐怖了。可以进行图片压缩显示。

1. **使用优化过的数据集合**

Android API当中提供了一些优化过后的数据集合工具类，如SparseArray，SparseBooleanArray，以及LongSparseArray等，使用这些API可以让我们的程序更加高效。传统Java API中提供的HashMap工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间。

1. **知晓内存的开支情况**

我们还应当清楚我们所使用语言的内存开支和消耗情况，并且在整个软件的设计和开发当中都应该将这些信息考虑在内。可能有一些看起来无关痛痒的写法，结果却会导致很大一部分的内存开支，例如：

使用枚举通常会比使用静态常量要消耗两倍以上的内存，在Android开发当中我们应当尽可能地不使用枚举。

任何一个Java类，包括内部类、匿名类，都要占用大概500字节的内存空间。

任何一个类的实例要消耗12-16字节的内存开支，因此频繁创建实例也是会一定程序上影响内存的。

在使用HashMap时，即使你只设置了一个基本数据类型的键，比如说int，但是也会按照对象的大小来分配内存，大概是32字节，而不是4字节。因此最好的办法就是像上面所说的一样，使用优化过的数据集合。

1. **谨慎使用抽象编程**

许多程序员都喜欢各种使用抽象来编程，认为这是一种很好的编程习惯。当然，这一点不可否认，因为的抽象的编程方法更加面向对象，而且在代码的维护和可扩展性方面都会有所提高。但是，在Android上使用抽象会带来额外的内存开支，因为抽象的编程方法需要编写额外的代码，虽然这些代码根本执行不到，但是却也要映射到内存当中，不仅占用了更多的内存，在执行效率方面也会有所降低。当然这里我并不是提倡大家完全不使用抽象编程，而是谨慎使用抽象编程，不要认为这是一种很酷的编程方式而去肆意使用它，只在你认为有必要的情况下才去使用。

1. **尽量避免使用依赖注入框架**

现在有很多人都喜欢在Android工程当中使用依赖注入框架，比如说像Guice或者RoboGuice等，因为它们可以简化一些复杂的编码操作，看上去确实十分诱人，我们甚至可以将findViewById()这一类的繁琐操作全部省去了。但是这些框架为了要搜寻代码中的注解，通常都需要经历较长的初始化过程，并且还可能将一些你用不到的对象也一并加载到内存当中。这些用不到的对象会一直占用着内存空间，可能要过很久之后才会得到释放，相较之下，也许多敲几行看似繁琐的代码才是更好的选择。

1. **使用ProGuard简化代码**

ProGuard相信大家都不会陌生，很多人都会使用这个工具来混淆代码，但是除了混淆之外，它还具有压缩和优化代码的功能。ProGuard会对我们的代码进行检索，删除一些无用的代码，并且会对类、字段、方法等进行重命名，重命名之后的类、字段和方法名都会比原来简短很多，这样的话也就对内存的占用变得更少了。

1. **使用多个进程**

这个技巧其实并不是非常建议使用，但它确实是一种可以帮助我们节省和管理内存的高级技巧。如果你要使用它的话一定要谨慎使用，因为绝大多数的应用程序都不应该在多个进程当中运行的，一旦使用不当，它甚至会增加额外的内存而不是帮我们节省内存。这个技巧比较适用于那些需要在后台去完成一项独立的任务，和前台的功能是可以完全区分开的场景。

这里举一个比较适合去使用多进程技巧的场景，比如说我们正在做一个音乐播放器软件，其中播放音乐的功能应该是一个独立的功能，它不需要和UI方面有任何关系，即使软件已经关闭了也应该可以正常播放音乐。如果此时我们只使用一个进程，那么即使用户关闭了软件，已经完全由Service来控制音乐播放了，系统仍然会将许多UI方面的内存进行保留。在这种场景下就非常适合使用两个进程，一个用于UI展示，另一个则用于在后台持续地播放音乐。

想要实现多进程的功能也非常简单，只需要在AndroidManifest文件的应用程序组件中声明一个android:process属性就可以了，比如说我们希望播放音乐的Service可以运行在一个单独的进程当中

**首先第一部分GC\_Reason，这个是触发这次GC操作的原因，一般情况下一共有以下几种触发GC操作的原因：**

**GC\_CONCURRENT:** 当我们应用程序的堆内存快要满的时候，系统会自动触发GC操作来释放内存。

**GC\_FOR\_MALLOC:**  当我们的应用程序需要分配更多内存，可是现有内存已经不足的时候，系统会进行GC操作来释放内存。

**GC\_HPROF\_DUMP\_HEAP:**  当生成HPROF文件的时候，系统会进行GC操作，关于HPROF文件我们下面会讲到。

**GC\_EXPLICIT:** 这种情况就是我们刚才提到过的，主动通知系统去进行GC操作，比如调用System.gc()方法来通知系统。或者在DDMS中，通过工具按钮也是可以显式地告诉系统进行GC操作的。

1. **避免创建不必要的对象**

下面来看一些我们可以避免创建对象的场景：

* 如果我们有一个需要拼接的字符串，那么可以优先考虑使用StringBuffer或者StringBuilder来进行拼接，而不是加号连接符，因为使用加号连接符会创建多余的对象，拼接的字符串越长，加号连接符的性能越低。
* 在没有特殊原因的情况下，尽量使用基本数据类来代替封装数据类型，int比Integer要更加高效，其它数据类型也是一样。
* 当一个方法的返回值是String的时候，通常可以去判断一下这个String的作用是什么，如果我们明确地知道调用方会将这个返回的String再进行拼接操作的话，可以考虑返回一个StringBuffer对象来代替，因为这样可以将一个对象的引用进行返回，而返回String的话就是创建了一个短生命周期的临时对象。
* 正如前面所说，基本数据类型要优于对象数据类型，类似地，基本数据类型的数组也要优于对象数据类型的数组。另外，两个平行的数组要比一个封装好的对象数组更加高效，举个例子，Foo[]和Bar[]这样的两个数组，使用起来要比Custom(Foo,Bar)[]这样的一个数组高效得多。

1. **静态优于抽象**

如果你并不需要访问一个对象中的某些字段，只是想调用它的某个方法来去完成一项通用的功能，那么可以将这个方法设置成静态方法，这会让调用的速度提升15%-20%，同时也不用为了调用这个方法而去专门创建对象了，这样还满足了上面的一条原则。另外这也是一种好的编程习惯，因为我们可以放心地调用静态方法，而不用担心调用这个方法后是否会改变对象的状态（静态方法内无法访问非静态字段）。

* **内存检测工具**

1. [LeakCannary](http://www.liaohuqiu.net/cn/posts/leak-canary/)在dependencies中加入以下三行代码：

* debugCompile 'com.squareup.leakcanary:leakcanary-android:1.5.1'
* releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.1'
* testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.1'

初始化：

继承自Application，然后在ExampleApplication中加入以下代码：

public class ExampleApplication extends Application {

@Override public void onCreate() {

super.onCreate();

if (LeakCanary.isInAnalyzerProcess(this)) {

// This process is dedicated to LeakCanary for heap analysis.

// You should not init your app in this process.

return;

}

LeakCanary.install(this);

// Normal app init code...

}

}

1. MAT

**开源框架问题**

* Volley：特别适合数据量小，通信频繁的网络操作。

首先从与框架同名的Volley.java开始分析,文件位于toolbox。该类主要用于建立一个新的请求队列RequestQueue，方法有两个分别是：

newRequestQueue(Context context, HttpStack stack)

newRequestQueue(Context context)

1.传入Context时，调用newRequestQueue(context, null)，然后根据DEFAULT\_CACHE\_DIR定义缓存文件名。

2.然后根据系统版本号，sdk大于9时使用HurlStack( HttpUrlConnection )，否则使用HttpClientStack( HttpClient ),然后将Stack传入Network，Network可以调用Stack处理请求，然后将结果转换成NetworkResponse

3.然后将定义的Network和缓存目录传入RequestQueue(Cache cache, Network network)去新建请求队列。

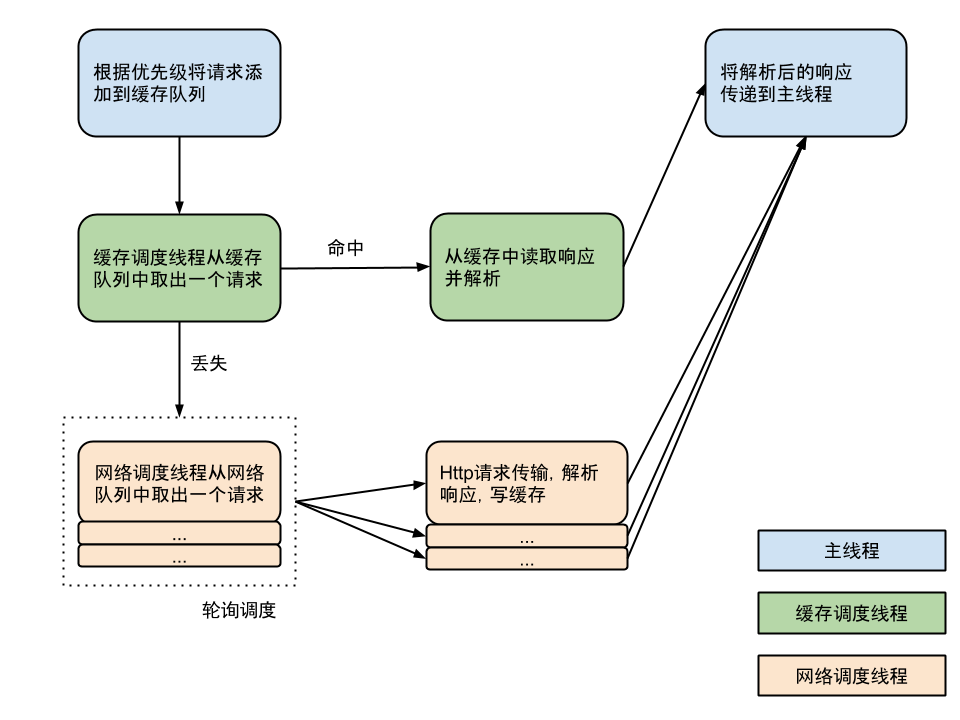
4.最后开启请求队列，执行start()方法，该方法会产生根据线程池大小产生对应数量的网络调度线程NetworkDispatcher和一个缓存调度线程CacheDispatcher(此处是用于异步处理请求的)

5.每次调用只需要将自定义的Request通过add()方法传入，请求将会被添加到mCurrentRequests，该Set用于存储当前正在处理的请求，接着根据请求的shouldCache()方法判断是否需要进行缓存，如果不需要则直接放入网络队列，否则根据Url将请求放入缓存队列。

6.缓存调度器CacheDispatcher会从缓存队列中取出请求，然后根据CacheKey获取缓存，如果获取失败则放入网络队列中去处理，否则进行验证有效期等，最后将处理结果交由ResponseDelivery，他是返回结果分发接口，用于分发返回的结果

7.网络调度器NetworkDispatcher会从网络队列中取出请求，然后处理完成交由ResponseDelivery分发处理结果。

最后附上Volley的请求流程图





**android内存缓存类：LruCache。Android本地缓存：DiskLruCache**

**Dalvik虚拟机与java虚拟机的区别**

1.java虚拟机运行的是Java字节码，Dalvik虚拟机运行的是Dalvik字节码；传统的Java程序经过编译，生成Java字节码保存在class文件中，java虚拟机通过解码class文件中的内容来运行程序。而Dalvik虚拟机运行的是Dalvik字节码，所有的Dalvik字节码由Java字节码转换而来，并被打包到一个DEX(Dalvik Executable)可执行文件中Dalvik虚拟机通过解释Dex文件来执行这些字节码。

2.Dalvik可执行文件体积更小。SDK中有一个叫dx的工具负责将java字节码转换为Dalvik字节码。

3.java虚拟机与Dalvik虚拟机架构不同。java虚拟机基于栈架构。程序在运行时虚拟机需要频繁的从栈上读取或写入数据。这过程需要更多的指令分派与内存访问次数，会耗费不少CPU时间，对于像手机设备资源有限的设备来说，这是相当大的一笔开销。Dalvik虚拟机基于寄存器架构，数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式快的多.