Android四大组件：Activity、service、

Launcher的源代码工程在packages/apps/Launcher2目录下，负责启动其它应用程序的源代码实现在src/com/android/launcher2/Launcher.java文件中

public void startActivity(Intent intent) 其实是默认启动使用了startActivityForResult(intent, -1);后面-1表示整个Activity结束时不用返回数据。这些控制是通过Intrumentation，定义在frameworks/base/core/java/android/app/Instrumentation.java文件中，它用来监控应用程序和系统的交互。

**整个应用程序的启动过程要执行很多步骤，但是整体来看，主要分为以下五个阶段：**

       一. Step1 - Step 11：Launcher通过Binder进程间通信机制通知ActivityManagerService，它要启动一个Activity；

       二. Step 12 - Step 16：ActivityManagerService通过Binder进程间通信机制通知Launcher进入Paused状态；

       三. Step 17 - Step 24：Launcher通过Binder进程间通信机制通知ActivityManagerService，它已经准备就绪进入Paused状态，于是ActivityManagerService就创建一个新的进程，用来启动一个ActivityThread实例，即将要启动的Activity就是在这个ActivityThread实例中运行；

       四. Step 25 - Step 27：ActivityThread通过Binder进程间通信机制将一个ApplicationThread类型的Binder对象传递给ActivityManagerService，以便以后ActivityManagerService能够通过这个Binder对象和它进行通信；

       五. Step 28 - Step 35：ActivityManagerService通过Binder进程间通信机制通知ActivityThread，现在一切准备就绪，它可以真正执行Activity的启动操作了。

**Activity的四种启动模式及其应用场景**

1. **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。

1. **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。

1. **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。

1. **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：

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。

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。

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。

**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.

**Android 进程间通信：aidl、localSocket、Broadcast、Messenger**

**android 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。

**利用onSaveInstanceState()方法保存Activity状态**

***内存泄漏，内存性能优化***

* **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。

* **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。

***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。

* **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。

* **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。

* **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。

* **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。

* **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。

* **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。

* **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。

**内存泄漏产生的原因在Android中大致分为以下几种：**

**1.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。

2.线程造成的内存泄漏

类似于上述例子中的情况，线程执行时间很长，及时Activity跳出还会执行，因为线程或者Runnable是Acticvity内部类，因此握有Activity的实例(因为创建内部类必须依靠外部类)，因此造成Activity无法释放。

AsyncTask 有线程池，问题更严重

解决办法：

1.合理安排线程执行的时间，控制线程在Activity结束前结束。

2.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

3.BitMap占用过多内存

bitmap的解析需要占用内存，但是内存只提供8M的空间给BitMap，如果图片过多，并且没有及时 recycle bitmap 那么就会造成内存溢出。

解决办法：

及时recycle 压缩图片之后加载图片

4.资源未被及时关闭造成的内存泄漏

比如一些Cursor 没有及时close 会保存有Activity的引用，导致内存泄漏

解决办法：

在onDestory方法中及时 close即可

5.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。

解决办法：

依旧使用 静态内部类+弱引用的方式 可解决

其中还有一些关于 集合对象没移除，注册的对象没反注册，代码压力的问题也可能产生内存泄漏，但是使用上述的几种解决办法一般都是可以解决的。

**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。

**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）。