# 内存优化：

提到内存优化一块，最大问题就是OOM：

无处不在的OOM。

明明还有剩余内存，但是应用还是因为OOM而crash了。

崩溃率居高不下。

不断扩展的业务需求。

内存优化在应用上主要体现在两个方面，如下所示：

保证应用进程的稳定性。

减少应用进程不必要的内存使用。

# 布局优化：

# 选择 耗费性能较少的布局

* 性能耗费低的布局 = 功能简单 = FrameLayout、LinearLayout
* 性能耗费高的布局 = 功能复杂 = RelativeLayout

1. **减少布局的层级（嵌套）**

* 原理：布局层级少 ->> 绘制的工作量少 ->> 绘制速度快 ->> 性能提高
* 优化方式：使用布局标签<merge> & 合适选择布局类型

### 提高 布局 的复用性

### 减少初次测量 & 绘制时间

主要优化方案：使用 布局标签<ViewStub> & 尽可能少用布局属性 wrap\_content

但是这些方式都不是正途，减少应用进程不必要的内存使用才有我们优化内存的康庄大道。

1. 使用LeakCanary监测内存泄漏。
2. 图片是Android里的内存占用大户，关于图片的使用一方面是图片缓存，另一方面是图片压缩。
3. 避免频繁的创建小对象，避免在循环中创建临时对象，例如大量的字符串拼接。因为大量创建小对象，会造成内存频繁的分配和回收（内存抖动），形成大量内存碎片，因为内存碎片不连续，无法直接分配，所以可能会导致OOM。
4. 当App切换到后台以后，应该停止一些不是必须要运行的服务，我们一般可以使用 JobScheduler 来实现后台任务，如果必须要使用服务，则应该使用 IntentService ， IntentService 在处理完所有请求之后会自动停止，而不是会一直运行下去。
5. 使用优化的数据集合 SparseArray / SparseBooleanArray / LongSparseArray 代替常规的HashMap，HashMap为每个映射都单独创建一个对象，内存效率低下。
6. 注意控制代码量，减少无用代码，特别是对于一些庞大的第三方库，能少用则尽量善用，DEX文件加载到内存中也是需要占用不少内存的。
7. 界面不可及的时候释放一些资源，在页面里覆写onTrimMemory()方法，根据不同的情况进行不同的处理

# Apk的打包流程

Android的包文件APK分为两个部分：代码和资源，所以打包方面也分为资源打包和代码打包两个方面。

1. 通过AAPT工具进行资源文件（包括AndroidManifest.xml、布局文件、各种xml资源等）的打包，生成R.java文件。
2. 通过AIDL工具处理AIDL文件，生成相应的Java文件。
3. 通过Javac工具编译项目源码，生成Class文件。
4. 通过DX工具将所有的Class文件转换成DEX文件，该过程主要完成Java字节码转换成Dalvik字节码，压缩常量池以及清除冗余信息等工作。
5. 通过ApkBuilder工具将资源文件、DEX文件打包生成APK文件。
6. 利用KeyStore对生成的APK文件进行签名。
7. 如果是正式版的APK，还会利用ZipAlign工具进行对齐处理，对齐的过程就是将APK文件中所有的资源文件举例文件的起始距离都偏移4字节的整数倍，这样通过内存映射访问APK文件 的速度会更快。

# [FragmentTransaction的commit的异步操作](http://blog.csdn.net/Picasso_L/article/details/50994143)

FragmentTransaction是异步的，commit()仅是相当于把操作加入到FragmentManager的队列，然后FragmentManager会在某一个时刻来执行，并不是立即执行。所以，真正开始执行commit()时，如果Activity的生命周期发生了变化，比如走到了onPause，或者走到了onStop，或者onDestroy都走完了，那么就会报出IllegalStateException。

# RecyclerView

RecyclerView继承于ViewGroup，实现了ScrollingView与NestedScrollingChild接口，它是日常业务开发中使用频度非常高的一个组件，它被Android设计出来代替原来的ListView组件。RecyclerView的的 解耦与设计是十分精妙的，应用了适配器、观察者等多种模式。

提到RecyclerView，就不得不说一下早期使用的AdapterView（ListView），这里做下它们的简单对比：

* Item复用方面：RecyclerView内置了RecyclerViewPool、多级缓存、ViewHolder，而AdapterView需要手动添加ViewHolder且复用功能也没RecyclerView完善。
* 样式丰富方面：RecyclerView通过支持水平、垂直和表格列表及其他更复杂形式，而AdapterView只支持具体某一种。
* 效果增强方面：RecyclerView内置了ItemDecoration和ItemAnimator，可以自定义绘制itemView之间的一些特殊UI或item项数据变化时的动画效果，而用AdapterView实现时采取的做法是 将这些特殊UI作为itemView的一部分，设置可见不可见决定是否展现，且数据变化时的动画效果没有提供，实现较为繁琐。
* 代码内聚方面：RecyclerView将功能密切相关的类写成内部类（例如：ViewHolder，Adapter），而AdapterView则没有。

# Handler

在安卓开发中：

* 为了保证Android的UI操作是线程安全的，Android规定了只允许UI线程修改Activity里的UI组件；
* 但在实际开发中，必然会用到多个线程并发操作UI组件，这又将导致UI操作的线程不安全

所以问题在于：如何同时满足：

* 保证线程安全
* 使多个线程并发操作UI组件

Handler消息传递机制就是这个问题的。

## Looper

每一个线程只有一个Looper，每个线程在初始化Looper之后，然后Looper会维护好该线程的消息队列，用来存放Handler发送的Message，并处理消息队列出队的Message。它的特点是它跟它的线程是绑定的，处理消息也是在Looper所在的线程去处理，所以当我们在主线程创建Handler时，它就会跟主线程唯一的Looper绑定，从而我们使用Handler在子线程发消息时，最终也是在主线程处理，达到了异步的效果。

那么就会有人问，为什么我们使用Handler的时候从来都不需要创建Looper呢？这是因为在主线程中，ActivityThread默认会把Looper初始化好，prepare以后，当前线程就会变成一个Looper线程。

## MessageQueue

MessageQueue是一个消息队列，用来存放Handler发送的消息。每个线程最多只有一个MessageQueue。MessageQueue通常都是由Looper来管理，而主线程创建时，会创建一个默认的Looper对象，而Looper对象的创建，将自动创建一个MessageQueue。其他非主线程，不会自动创建Looper。

## Message

消息对象，就是MessageQueue里面存放的对象，一个MessageQueu可以包括多个Message。当我们需要发送一个Message时，我们一般不建议使用new Message()的形式来创建，更推荐使用Message.obtain()来获取Message实例，因为在Message类里面定义了一个消息池，当消息池里存在未使用的消息时，便返回，如果没有未使用的消息，则通过new的方式创建返回，所以使用Message.obtain()的方式来获取实例可以大大减少当有大量Message对象而产生的垃圾回收问题。

# ****推送****

### 轮询

* 原理  
  基于Pull方式
* 具体描述  
  应用程序隔固定时间**主动**与服务器进行连接并查询是否有新的消息
* 优点  
  实时性好
* 缺点
  1. 成本大，需要自己实现与服务器之间的通信，例如消息排队等；
  2. 到达率不确定，考虑轮询的频率：太低可能导致消息的延迟；太高，更费客户端的资源（CPU资源、网络流量、系统电量）和服务器资源（网络带宽）

应用开发框架

Picasso原理

消息通讯机制

多线程，tcp udp 协议

设计模式

# Picasso

* 使用简单、方便（一行代码的事情）
* 由于同样是出品自Square的，Square 公司的其他开源库如 Retrofit 或者 OkHttp和Picasso搭配使用兼容性会更好些，占用体积也会少些

所以，如果项目已经使用了 Square 公司的其他开源库（如 Retrofit 或者 OkHttp），在满足需求的前提下建议使用Picasso

# 插件化

下面是一些插件化的优势：

* 在一个大的项目里面，为了明确的分工，往往不同的团队负责不同的插件APP，这样分工更加明确。
* 各个模块封装成不同的插件APK，不同模块可以单独编译，提高了开发效率。
* 解决了上述的方法数超过限制的问题。
* 可以通过上线新的插件来解决线上的BUG，达到“热修复”的效果。
* 减小了宿主APK的体积。

# 什么情况下会导致内存泄漏

* 程序代码的问题，长期保持某些资源，如Context、Cursor、IO流的引用，资源得不到释放造成内存泄露。
* 对象内存过大问题  
  保存了多个耗用内存过大的对象（如Bitmap、XML文件），造成内存超出限制。
* static关键字的使用问题  
  static是Java中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（Context的情况最多），这时就要谨慎对待了。  
  针对static的解决方案：

1. 应该尽量避免static成员变量引用资源耗费过多的实例，比如Context。
2. Context尽量使用ApplicationContext，因为Application的Context的生命周期比较长，引用它不会出现内存泄露的问题。
3. 使用WeakReference代替强引用。比如可以使用WeakReference<Context> mContextRef;

* 线程导致内存溢出  
  线程产生内存泄露的主要原因在于线程生命周期的不可控。  
  针对这种线程导致的内存泄露问题的解决方案：  
  λ 将线程的内部类，改为静态内部类（因为非静态内部类拥有外部类对象的强引用，而静态类则不拥有）。  
  λ 在线程内部采用弱引用保存Context引用。
* 查询数据库没有关闭cursor  
  程序中经常会进行查询数据库的操作，但是经常会有使用完毕Cursor后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会出现内存问题，这样就会给以后的测试和问题排查带来困难和风险。
* 构造Adapter没有复用convertview  
  在使用ListView的时候通常会使用Adapter，那么我们应该尽可能的使用ConvertView。  
  为什么要复用convertView?  
  当convertView为空时，用setTag()方法为每个View绑定一个存放控件的ViewHolder对象。当convertView不为空，重复利用已经创建的view的时候，使用getTag()方法获取绑定的ViewHolder对象，这样就避免了findViewById对控件的层层查询，而是快速定位到控件。
* Bitmap不再使用时没有调用recycle()释放内存

有时我们会手工的操作Bitmap对象，如果一个Bitmap对象比较占内存，当它不再被使用的时候，可以调用Bitmap.recycle()方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。

# ****插件化和热修复的区别****

* 插件化：插件化是体现在功能拆分方面的，它将某个功能独立提取出来，独立开发，独立测试，再插入到主应用中。依次来较少主应用的规模。
* 热修复：热修复是体现在bug修复方面的，它实现的是不需要重新发版和重新安装，就可以去修复已知的bug。

ArrayList和LinkedList的大致区别:  
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。   
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。

# MVC框架是如何实现的？都充当什么角色？

1. View接受用户的交互请求；

2. View将请求转交给Controller；

3. Controller（用户做的动作比如：update数据，删除指定名字的学生等等）操作Model进行数据更新（根据用户指示，执行底层的数据动作等等）；

4. 数据更新之后，Model通知View数据变化；

5. View显示更新之后的数据；

M层适合做一些业务逻辑处理，比如数据库存取操作、网络操作、复杂的算法等耗时操作；

V层显示数据部分，XML布局可以视为是V层，显示Model层的数据结果；

C层适合使用Activity担当，Android中Activity用于处理用户交互问题（发起业务请求），读取用户输入（等待业务处理结果），响应用户点击等等事件。

# AIDL的作用

AIDL (Android Interface Definition Language) 是一种IDL 语言，用于生成可以在Android设备上两个进程之间进行进程间通信(interprocess communication, IPC)的代码。

如果在一个进程中（例如Activity）要调用另一个进程中（例如Service）对象的操作，就可以使用AIDL生成可序列化的参数。

WHY？

a. 某些情况下远端的服务更适合运算或者更适合执行耗时操作，这时候我们会使用aidl请求远程服务；   
b. android对单个应用的内存限制，当有需求需要突破这个限制的时候我们需要另启进程扩大内存。

# [Android 中三种启用线程的方法](http://www.cnblogs.com/propheterLiu/p/6082666.html)

### 1首先第一种启用方法是通过继承Thread类，并改写run方法来实现一个线程

new MyThread().start();

### 2第二种启用方式创建一个Runnable对象

new Thread(new MyRunnable()).start();

### 3第三种启用方式通过Handler启动线程

Thread 是一个类，rannable是一个接口

# 数据结构

Android中一般使用的数据结构有java中的基础数据结构Set, List, Map。

## 1.List结构的集合类:ArrayList、LinkedList、Vector、Stack；

Arraylist是线性顺序数据结构，而LinkedList是线性链表数据结构。也就是说，Arraylist集合中的元素内存地址是一个挨着一个，当知道第一个元素的内存地址，便可通过计算，得到其余元素内存地址；而LinkedList是链表数据结构，元素内存地址可以连续排列也可以不连续，一般通过前一个元素查找下一个元素，因此对于随机访问get和set，ArrayList优于LinkedList。但对于新增和删除操作add和remove，LinedList比较占优势，因为向ArrayList中插入或者删除数据时，所插入数据位置后的所有数据都要移动，而LinkedList不需要移动数据。

**2. Map结构集合类：HashMap、HashTable、LinkedHashMap和TreeMap。**

Map中每个项都是成对的。映射中存储的每个对象都有一个相关的关键字（Key）对象，关键字决定了对象在映射中的存储位置，检索对象时必须提供相应的关键字，就像在字典中查单词一样。关键字应该是唯一的。

**3.Set集合中的对象不按特定的方式排序，并且没有重复对象**。

Set接口主要实现了两个实现类：HashSet和TreeSet 。HashSet类按照哈希算法来存取集合中的对象，存取速度比较快。TreeSet：TreeSet类实现了SortedSet接口，能够对集合中的对象进行排序。

# Android中图片的三级缓存策略

通常情况下，Android应用程序中图片的缓存策略采用“内存-本地-网络”三级缓存策略，首先应用程序访问网络拉取图片，分别将加载的图片保存在本地SD卡中和内存中，当程序再一次需要加载图片的时候，先判断内存中是否有缓存，有则直接从内存中拉取，否则查看本地SD卡中是否有缓存，SD卡中如果存在缓存，则图片从SD卡中拉取，否则从网络加载图片。依据这三级缓存机制，可以让我们的应用程序在加载图片的时候做到游刃有余，有效的避免内存溢出。

**图片的内存优化：**

**1) 要及时回收Bitmap的内存**

Bitmap类有一个方法recycle()，从方法名可以看出意思是回收。这里就有疑问了，Android系统有自己的垃圾回收机制，可以不定期的回收掉不使用的内存空间，当然也包括Bitmap的空间。那为什么还需要这个方法呢？

Bitmap类的构造方法都是私有的，所以开发者不能直接new出一个Bitmap对象，只能通过BitmapFactory类的各种静态方法来实例化一个Bitmap。仔细查看BitmapFactory的源代码可以看到，生成Bitmap对象最终都是通过JNI调用方式实现的。所以，加载Bitmap到内存里以后，是包含两部分内存区域的。简单的说，一部分是Java部分的，一部分是C部分的。这个Bitmap对象是由Java部分分配的，不用的时候系统就会自动回收了，但是那个对应的C可用的内存区域，虚拟机是不能直接回收的，这个只能调用底层的功能释放。所以需要调用recycle()方法来释放C部分的内存。从Bitmap类的源代码也可以看到，recycle()方法里也的确是调用了JNI方法了的。

**2) 捕获异常**

因为Bitmap是吃内存大户，为了避免应用在分配Bitmap内存的时候出现OutOfMemory异常以后Crash掉，需要特别注意实例化Bitmap部分的代码。通常，在实例化Bitmap的代码中，一定要对OutOfMemory异常进行捕获。

**3) 缓存通用的Bitmap对象**

有时候，可能需要在一个Activity里多次用到同一张图片。比如一个Activity会展示一些用户的头像列表，而如果用户没有设置头像的话，则会显示一个默认头像，而这个头像是位于应用程序本身的资源文件中的。

如果有类似上面的场景，就可以对同一Bitmap进行缓存。如果不进行缓存，尽管看到的是同一张图片文件，但是使用BitmapFactory类的方法来实例化出来的Bitmap，是不同的Bitmap对象。缓存可以避免新建多个Bitmap对象，避免内存的浪费。

**4) 压缩图片**

使用BitmapFactory.Options设置inJustDecodeBounds为true后，再使用decodeFile()等方法，并不会真正的分配空间，即解码出来的Bitmap为null，但是可计算出原始图片的宽度和高度，即options.outWidth和options.outHeight。通过这两个值，就可以知道图片是否过大了。

# Activity四大启动模式

## standard-默认模式

每次启动一个Activity都会重写创建一个新的实例

## singleTop-栈顶复用模式

 这个模式下，如果新的activity已经位于栈顶，那么这个Activity不会被重写创建，同时它的onNewIntent方法会被调用，通过此方法的参数我们可以去除当前请求的信息。如果栈顶不存在该Activity的实例，则情况与standard模式相同。

## singleTask-栈内复用模式

在这个模式下，如果栈中存在这个Activity的实例就会复用这个Activity，不管它是否位于栈顶，复用时，会将它上面的Activity全部出栈，并且会回调该实例的onNewIntent方法。

## singleInstance-全局唯一模式

该模式具备singleTask模式的所有特性外，与它的区别就是，这种模式下的Activity会单独占用一个Task栈，具有全局唯一性，即整个系统中就这么一个实例，由于栈内复用的特性，后续的请求均不会创建新的Activity实例，除非这个特殊的任务栈被销毁了。

# ANR

在Android里，应用程序的响应是由ActivityManager和WindowManager服务系统服务监视的，当检测到下面三种情况的任何一种时，Android就会针对特定的应用程序显示ANR对话框。

* Activity的UI在5秒内没有响应输入事件（例如，按键按下，屏幕触摸）–主要类型KeyDispatchTimeout
* BroadcastReceiver在10秒内没有执行完毕BroadcastTimeout
* Service在特定时间内（20秒内）无法处理完成–小概率类型ServiceTimeout

### 如何避免ANR

* 避免在主线程进行复杂耗时的操作，特别是文件读取或者数据库操作；
* 避免频繁实时更新UI；
* BroadCastReceiver 要进行复杂操作的的时候，可以在onReceive()方法中启动一个Service来处理；
* 避免在IntentReceiver里启动一个Activity，因为它会创建一个新的画面，并从当前用户正在运行的程序上抢夺焦点。如果你的应用程序在响应Intent广 播时需要向用户展示什么，你应该使用Notification Manager来实现。
* 在设计及代码编写阶段避免出现出现同步/死锁或者错误处理不恰当等情况。

# ListView的四种优化方式

* convertView的复用
* ViewHolder的使用
* 使用分段加载
* 使用分页加载

**图片错位**

每当有新的元素进入界面时就会回调getView()方法，而在getView()方法中会开启异步请求从网络上获取图片，注意网络操作都是比较耗时的，也就是说当我们快速滑动ListView的时候就很有可能出现这样一种情况，某一个位置上的元素进入屏幕后开始从网络上请求图片，但是还没等图片下载完成，它就又被移出了屏幕。这种情况下会产生什么样的现象呢？根据ListView的工作原理，被移出屏幕的控件将会很快被新进入屏幕的元素重新利用起来，而如果在这个时候刚好前面发起的图片请求有了响应，就会将刚才位置上的图片显示到当前位置上，因为虽然它们位置不同，但都是共用的同一个ImageView实例，这样就出现了图片乱序的情况。

但是还没完，新进入屏幕的元素它也会发起一条网络请求来获取当前位置的图片，等到图片下载完的时候会设置到同样的ImageView上面，因此就会出现先显示一张图片，然后又变成了另外一张图片的情况，那么刚才我们看到的图片会自动变来变去的情况也就得到了解释。

# 设计模式

## 单例模式

### 确保某个类有且只有一个对象的场景，例如创建一个对象需要消耗的资源过多，如要访问 IO 和数据库等资源。

#### 优点

* 由于单例模式在内存中只有一个实例，减少了内存开支，特别是一个对象需要频繁地创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显。
* 由于单例模式只生成一个实例，所以减少了系统的性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后用永久驻留内存的方式来解决；
* 单例模式可以避免对资源的多重占用，例如一个写文件动作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。
* 单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

#### 缺点

* 单例模式一般没有接口，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。

**Builder模式**

### 模式的使用场景

1. 相同的方法，不同的执行顺序，产生不同的事件结果时；
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时；
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适；

常见的builder模式有创建dialog的过程。

#### 优点

* 良好的封装性， 使用建造者模式可以使客户端不必知道产品内部组成的细节；
* 建造者独立，容易扩展；
* 在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到。

#### 缺点

* 会产生多余的Builder对象以及Director对象，消耗内存；
* 对象的构建过程暴露。

**策略模式**

### 模式的使用场景

* 针对同一类型问题的多种处理方式，仅仅是具体行为有差别时。
* 需要安全的封装多种同一类型的操作时。
* 出现同一抽象多个子类，而又需要使用if-else 或者 switch-case来选择时。

日常的Android开发中经常会用到动画

优点：

* 结构清晰明了、使用简单直观。
* 耦合度相对而言较低，扩展方便。
* 操作封装也更为彻底，数据更为安全。

缺点：

* 随着策略的增加，子类也会变得繁多。

**模板方法模式**

### 模式的使用场景

1. 多个子类有公有的方法，并且逻辑基本相同时。
2. 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。
3. 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为。

在Android中，使用了模板方法且为我们熟知的一个典型类就是AsyncTask了

#### 优点

* 封装不变部分，扩展可变部分
* 提取公共部分代码，便于维护

#### 缺点

* 模板方法会带来代码阅读的难度，会让心觉得难以理解

### 观察者模式

定义对象间一种一对多的依赖关系，使得当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

Android中的应用，从最简单的开始，Button的点击事件

另外广播机制，本质也是观察者模式

开源框架EventBus也是基于观察者模式的

### 适配器模式

Adapter