# 1、LruCache原理

http://blog.csdn.net/qq\_25806863/article/details/77548468

http://www.voidcn.com/article/p-ornxekzd-bqr.html

LRU 是 Least Recently Used 最近最少使用算法；

就是当缓存空间满了的时候，将最近最少使用的数据从缓存空间中删除以增加可用的缓存空间来缓存新内容；

//第一个参数是初始容量

//第二个参数是填装因子，或叫加载因子，达到容量的75%时候，容量增加一半

//第三个参数是排序模式，true表示在访问的时候进行排序，否则只在插入的时候才排序。

this.map = new LinkedHashMap<K, V>(0, 0.75f, true);

LruCache 是通过 LinkedHashMap（双向循环链表） 构造方法的第三个参数的 accessOrder=true 实现了 LinkedHashMap

的数据排序基于访问顺序 （最近访问的数据会在链表尾部），在容量溢出的时候，将链表头部的数据移除。

从而，实现了 LRU 数据缓存机制

LinkedHashMap 的实现原理 http://wiki.jikexueyuan.com/project/java-collection/linkedhashmap.html

# 2、图片加载原理

https://www.jianshu.com/p/94e37c901107

Android系统加载图片是先要把图片转换为Bitmap对象然后再传到ImageView里显示

glide中文文档：https://muyangmin.github.io/glide-docs-cn/

glide缓存机制源码：http://blog.csdn.net/guolin\_blog/article/details/54895665

glide缓存分为：内存缓存、硬盘缓存

内存缓存：glide默认开启内存缓存，.skipMemoryCache(true)设置可以禁止缓存，glide内存缓存内部实现算法：LruCache

算法、弱引用的机制，共同完成缓存功能；

glide内部获取缓存的图片，会从两个方法中读取，分别是loadFromCache()和loadFromActiveResources()，

一个是LruCache算法，一个是弱引用；

LruCache算法获取到缓存图片后，会remove掉，但会在弱引用机制方法中存入到HashMap，也就是说弱引用的HashMap

用来缓存正在使用中的图片，保护这些图片不会被LruCache算法回收掉

总结：正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存；

硬盘缓存：默认缓存压缩过的图片， .diskCacheStrategy(DiskCacheStrategy.NONE)禁止

三种图片加载框架： http://blog.csdn.net/hong\_geek/article/details/49849339

# 3、模块化（组件化）

模块化：根据项目业务抽离，一个app分好多业务模块

组件化：根据功能点划分，如：支付、分享、下拉刷新库

方案：

模块间跳转包括携带参数：ARouter

模块间通讯：EventBus

单一模块调试：根目录gradle.properties文件定义变量，在模块的build.gradle文件判断时候需要继承调试，

然后单独配置调试的AndroidManifest.xml,用来启动单独调试的模块

原理：基于gradle方式编译来实现组件化

产生问题：组件中的switch-case语句，必须是常量，但组件中R.Java文件id不是常量，因此不能用

butterknife:需要把R.id.xx---->R2.id.xx;插件会自动帮我们生成R2文件，而R2文件里面的id是常量；

# 4、插件化、热修复

## 插件化

### 概念

插件化就是可以动态的扩展app的功能的一种技术，在app运行的时候加载插件，这样既可以减小apk的体积，又可以实现app功能的动态扩展。

### 需要考虑的问题

1. 插件于宿主app相互通信
2. 资源加载
3. 四大组件生命周期

### 原理

类加载机制和反射：比如资源加载，就是先通过dexclassloader加载apk（dex），在通过反射获取资源。

[参考](http://lruheng.com/2017/07/01/Android%E6%8F%92%E4%BB%B6%E5%8C%96%E5%85%A5%E9%97%A8%E6%8C%87%E5%8D%97/)

插件化：目前市面上做的兼容性比较好的平台有：滴滴（VirtualAPK）、360（DroidPlugin）

都支持四大组件以及android大部分特性，但VirtualAPK支持插件apk访问宿主apk

## 热修复

热修复：主要用于线上bug的紧急修复，兼容性较好的平台：阿里（Sophix）微信（Tinker）、饿了吗（Amigo）、美团（Robust）；代码修复主要有两大方案，一种是阿里系的底层替换方案，一种是腾讯系的类加载方案

两类优缺点：

阿里系：底层替换方案限制多，但是时效性最好，加载轻快，立即生效；

腾讯系：类加载方案时效性差，需要冷启动才能见效，但修复范围广，限制少；

底层替换原理：已经加载了的类中直接替换掉原有方法，是在原来类的基础上进行修改的。因而无法实现对与原有类进行方法和字段的增减，因为这样将破坏原有类的结构

类加载原理：是在app重新启动后让Classloader去加载新的类，

### Tinker

不支持新增四大组件，动态下发代码、So库以及资源，支持新增字段，分平台合成的想法，即在Dalvik平台合成全量Dex，在Art平台合成需要的小Dex

### Robust

美团平台，成功率高，不支持新增字段 install run 机制原理

### Dexposed/AndFix

阿里平台,最大挑战在于稳定性与兼容性，而且native异常排查难度更高。另一方面，由于无法增加变量与类等限制，Dexposed，由于对底层Dalvik结构过于依赖，最终无法继续兼容Android5.0以后ART虚拟机，AndFix兼容ART和Dalvik，且两者即时生效，Andfix只提供了代码层面的修复，对于资源和so的修复都还未能实现。

### Qzone

最大挑战在于性能，即Dalvik平台存在插桩导致的性能损耗，Art平台由于地址偏移问题导致补丁包可能过大的问题

### Aceso

蘑菇街平台，基于Instant Run Hot Swap，参考美团，实时性，支持方法的函数体修复、新增类，暂不支持静态代码块、构造函数的修复

### Amigo

饿了吗平台，支持四大组件

### Sophix

阿里平台，支持代码修复、资源修复、so修复方面，非侵入性，不支持四大组件，结合了底层替换和类加载两种方案，

[Sophix资料](https://zhuanlan.zhihu.com/p/27455593) [使用资料](https://www.jianshu.com/p/8ea4d653a53e)

## 增量更新

用于版本更新，宿主apk和新apk合并-->安装

# 5、Instant run机制

Android studio 2.0 Stable（稳定版）版本中集成了Install run即时编译技术

关于动态加载，实际上Instant run提供了两种动态加载的机制：

1. 修改java代码需要重启应用加载补丁dex，而在Application初始化时替换了Application，新建了一个自定义的ClassLoader去加载所有的dex文件。我们称为重启更新机制

2. 修改代码不需要重启，新建一个ClassLoader去加载修改部分。我们称为热更新机制

[参考资料](http://w4lle.com/2016/05/02/%E4%BB%8EInstant%20run%E8%B0%88Android%E6%9B%BF%E6%8D%A2Application%E5%92%8C%E5%8A%A8%E6%80%81%E5%8A%A0%E8%BD%BD%E6%9C%BA%E5%88%B6/)

# 6、进程间通讯（IPC）

Bundle

文件共享（对象持久化本地）

socket

Binder(AIDL、Messager、contentprovder)

## AIDL

Android接口定义语言(android Interface Definition Language),系统为我们快速生成跨进程通讯的工具。

AIDL支持数据类型：java基本数据类型、String和CharSequence、List、Map、实现Parcelabel接口的对象

# 7、JVM

包括：类加载器（ClassLoader）、执行引擎、运行时数据区（内存区）、本地方法接口

## 类加载器

：加载字节码文件（.class）到内存区（也就是运行时数据区）

## 执行引擎

：执行class文件中包含的字节码指令。

## 内存区

：又叫运行时数据区，包括：程序计数器、VM Stack、Native Method Stack、Heap(堆)、方法区等；

### 程序计数器

：用于当前线程执行的内存地址，JVM多线程执行，为了保证线程切换回来后，能恢复到原先的状态，就需要独立的pcr（程序计数器）,记录之前中断的地方

VM Stack（虚拟机栈）

:每创建一个线程，就会对应的创建一个Java栈，而这个Java栈又包含多个栈帧，用于存储局部变量表、操作栈、方法返回值，每一个方法从调用到结束，就对应一个栈帧在Java栈中入栈到出栈的过程

### Native Method Stack（本地方法栈）

:和Java栈相似，只不过是为JVM使用Native方法服务

### Heap（堆）

:存储实例或则对象的地方，GC的主要区域

### 方法区

：存储类结构信息，含常量池、静态常量、构造函数、运行时常量池

## 本地方法接口

：主要调用c/c++实例的本地方法以及返回结果

# 8、ThreadLocal原理

**ThreadLocal的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。**

**原理：**

**每个Thread维护一个ThreadLocalMap映射表，这个映射表的key是ThreadLocal实例本身，value是真正需要存储的Object**

[参考](https://www.zhihu.com/question/23089780)

# 9、ART和Dalvik区别

## AOT代替JIT

在Dalvik下，应用每次运行的时候，字节码都需要通过即时[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)(JIT, Just-in-time)转换为机器码，这会拖慢应用的运行效率，而在ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）

## garbage collection

dalvik的gc的过程:

1. 当gc被触发时候,其会去查找所有活动的对象,这个时候整个程序与虚拟机内部的所有线程就会挂起,这样目的是在较少的堆栈里找到所引用的对象.需要注意的是这个回收动作是和应用程序同时执行  
   2、gc对符合条件的对象进行标记  
   3、gc对标记的对象进行回收  
   4、恢复所有线程的执行现场继续运行

dalvik这么做的好处是,当pause了之后,gc势必是相当快速的.但是如果出现gc频繁并且内存吃紧势必会导致ui卡顿,掉帧.操作不流畅等

art改善了这种gc:

1. gc将会锁住java堆,扫描并进行标记  
   2、标记完毕释放掉java堆的锁,并且挂起所有线程  
   3、gc对标记的对象进行回收  
   4、恢复所有线程的执行现场继续运行  
   5、重复2-4直到结束

## Improved memory usage and reduce fragmentation

dalvik memory usage:  
可以对比一下两个虚拟机的内存分配的规则,首先是dalvik.他的内存管理特点是:内存碎片化严重,当然这也是Mark and Sweep算法带来的弊端, 每次gc后内存千疮百孔,本来连续分配的内存块变得碎片化严重,之后再分配进入的对象再进行内存寻址变得困难

Art memory usage:

在art中,它将java分了一块空间命名为Large-Object-Space,这块内存空间的引入用来专门存放large object.同时art又引入了moving collector的技术,即将不连续的物理内存块进行对齐.对齐了后内存碎片化就得到了很好的解决.Large-Object-Space的引入一是因为moving collector对大块内存的位移时间成本太高,而且提高内存的利用率

[参考资料](https://www.zhihu.com/question/29406156/answer/83413563)

# 10、类加载器（ClassLoader）

用来动态加载class文件到内存中用的；

## BootClassLoader

：系统启动的时候创建

## DexClassLoader

：可以加载jar/apk/dex，可以从SD卡中加载未安装的apk

## PathClassLoader

：应用启动时创建，只能加载系统中已经安装过的apk

## 双亲委托机制

：当一个ClassLoader实例加载某个类的时候，会先查询当前ClassLoader实例是否加载过此类，有就返回；

如果没有，查询parent是否已经加载过此类，如果已经加载过，就直接返回parent加载的类；

如果当前classLoader的基类都没有加载过，才有child执行类加载；

为什么要这样设计？

避免重复加载，保证唯一性，安全

Jvm怎么判断两个class相同？

1. 全类名（包括所在包名）
2. 同一个类加载器加载的实例

以上必须同时成立

[参考资料](https://zhuanlan.zhihu.com/p/20524252) https://blog.csdn.net/xiangzhihong8/article/details/52880327

# 11、如何计算app的启动时间

 adb shell am start -w packagename/activity

如果只关心某个应用自身启动耗时，参考TotalTime；如果关心系统启动应用耗时，参考WaitTime；如果关心应用有界面Activity启动耗时，参考ThisTime

[参考资料](http://androidperformance.com/2015/12/31/How-to-calculation-android-app-lunch-time.html)

# 12、注解(Annotation)

## 基本Annotation

@Override:强制子类必须覆盖父类方法  
@Deprecated:表示某个元素已经过时

@SuppressWarnings:抑制编译器警告

@SafeVarargs:(J1.7新增)“堆污染”警告，如：把一个不带参数的泛型的对象赋值给带泛型的变量时

## 元Anntation

@Retention:用于指定被修饰的Annotation可以保留多上时间，内部包含一个RetentionPolicy类型的value成员变量

RetentionPolicy.SOURCE:

Annotation只保留在源代码中，编译器直接丢弃这种Annotation

RetentionPolicy.RUNTIME:

Annotation保留在class文件中，程序运行时，JVM保留Annotation，程序可以通过反射获取Annotation信息

RetentionPolicy.CLASS:

Annotation记录在class文件中，当程序运行时，JVM不在保留Annotation

@Target:指定Annotation修饰程序那些单元

ElementType.ANNOTATION\_TYPE:修饰Annotation

ElementType.CONSTRUCTOR:修饰构造器

ElementType.FIELD:修饰成员变量

ElementType.LOCAL\_VARIABLE:修饰局部变量

ElementType.METHOD:修饰方法

ElementType.PACKAGE:修饰包

ElementType.PARAMETER:修饰类、接口、枚举

@Documented

修饰元素将被javadoc工具提取成文档

@Inherited

修饰的Annotation具有继承性

## 自定义Annotation

APT(Annotation Processing Tool):  
一种处理注解的工具，它可以检测源代码，找出Annotation，对其进行处理，APT还可以自动生成代码

Annotation 处理器

在编译时，提取并处理Java源文件中的Annotation,处理器需要实现javax.annotation.processing包下的Process接口，且要实现全部方法，但我们可以继承AbstractProcessor类来实现处理器类，

## 运行时注解

：效率比较低，因为底层利用java的反射技术，扫描注解修饰过的字段/方法/类

## 编译时注解

：编译时按照一定策略生成代码，避免编写重复代码，提高开发效率，且不影响性能

[参考资料](https://lizhaoxuan.github.io/2016/07/17/apt-wathapt/)

# 13、HashMap源码

[参考资料](https://tech.meituan.com/java-hashmap.html)

# 14、ButterKnife源码

[参考资料](https://segmentfault.com/a/1190000005994026)

# 15、Android性能优化

## UI优化

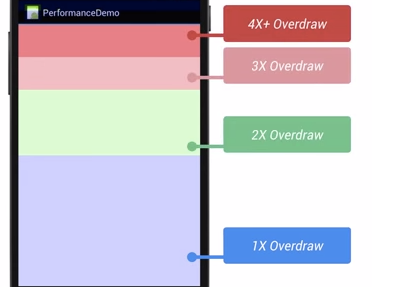
魅族性能优化工程师[个人博客](http://androidperformance.com/categories/Android/)

FPS（Frame Per Second）是图像领域中的定义，是指画面每秒传输帧数，也就是一秒钟内在屏幕上连续投射出多少张张静止画面，开发App的帧率性能目标就是保持在60fps,也就是大约16ms/帧，16ms要完成渲染，若果大于16ms/帧，则会出现卡顿现象。

UI性能差的直接表现——界面卡顿

1. UI线程做轻微的耗时操作，ps:数据库查询
2. 过度绘制（Overdraw）某个像素在同一帧的时间内被绘制了多次, 浪费大量的CPU以及GPU资源
3. 内存频繁gc，也会导致卡顿（Dalvik gc时，当前任务暂停，会出现轻微卡顿，貌似新版ART已经优化了，可以同时进行）
4. 布局过于复杂，无法在16ms内完成渲染
5. View频繁触发measure、layout导致重新渲染

### Show GPU Overdraw

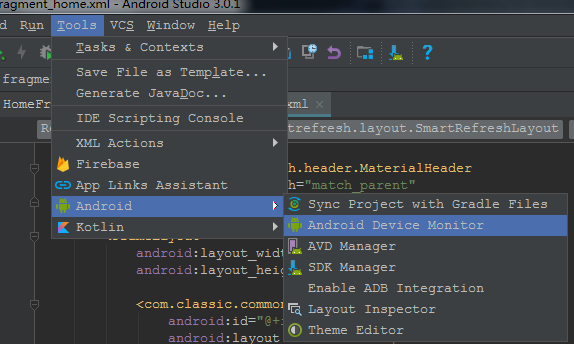


所以：优化布局层级、减少没必要的背景、暂时不显示的View设置为GONE而不是INVISIBLE、自定义View的onDraw方法设置canvas.clipRect()指定绘制区域或通过canvas.quickreject()减少绘制区域等

### GPU呈现模式图工具

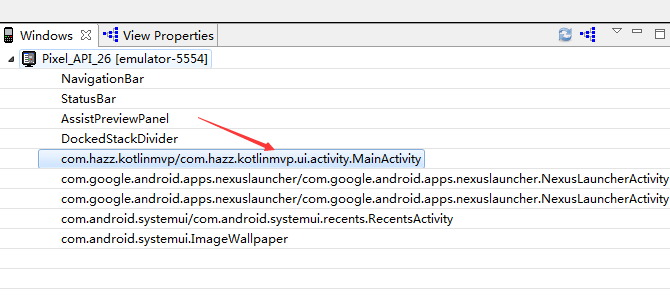
实时柱状图来显示每帧的渲染时间，柱状图越高表示渲染时间越长，每个柱状图偏上都有一根代表16ms基准的绿色横线，每一条竖着的柱状线都包含三部分（蓝色代表测量绘制Display List的时间，红色代表OpenGL渲染Display List所需要的时间，黄色代表CPU等待GPU处理的时间），只要我们每一帧的总时间低于基准线就不会发生UI卡顿问题

### Hierarchy Viewer

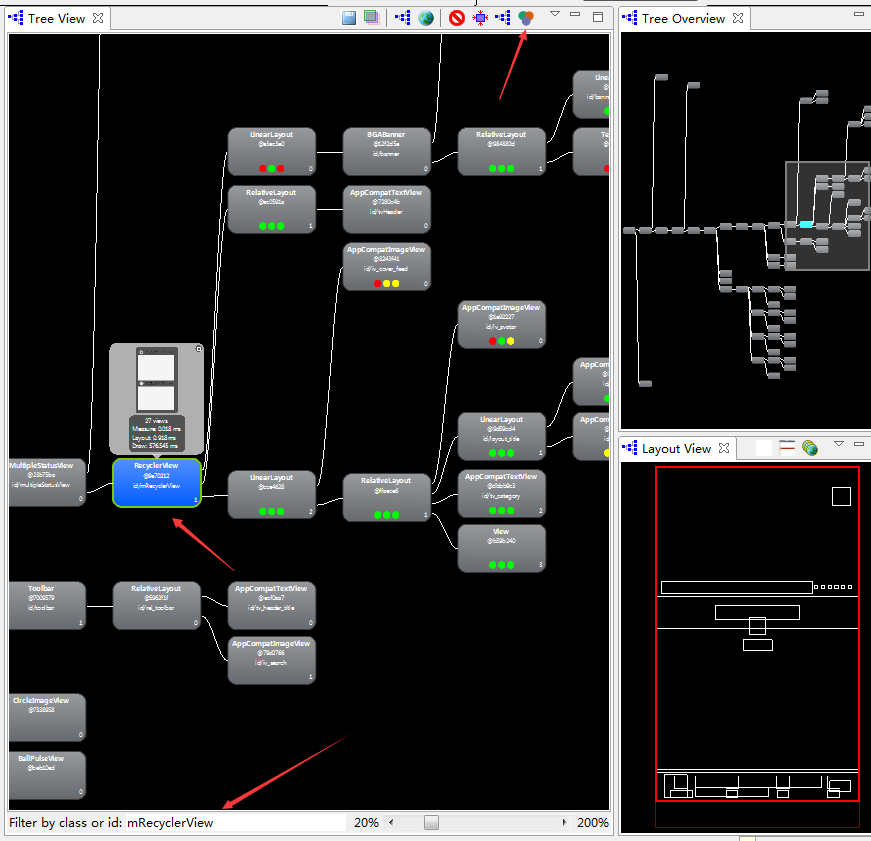


打开后，选择window🡪open perspective

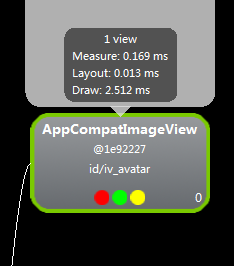
左侧找到打开后的app，找到对应的进程，需要测试那个界面直接双击，



然后看右侧视图：



最下方输入控件id，会自动定位，然后点击最上方红色箭头指向的小图标，如果点击的view分支较多，可能要耐心等待一会，



View参数表示有多少个子view包括自己

小圆点从左至右，分别表示：messure、layout、draw

下面的小圆点表示上面三个过程的程度：

绿色性能最佳，其次黄色，红色；红色就比较严重了

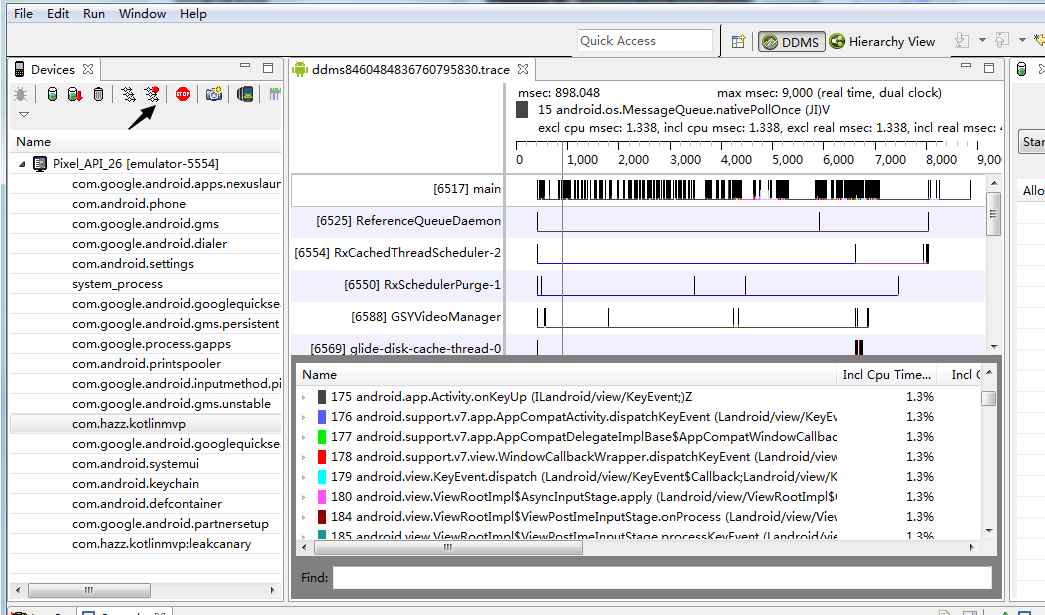
当然以上是调试的模拟器，如果是真机就不行了，如果是真机[参考](http://blog.csdn.net/swordgirl2011/article/details/52887925)

### Lint进行资源及冗余UI布局等优化

代码区点击右键->Analyze->Inspect Code–>界面选择你要检测的模块->点击确认开始检测

### Traceview

运行Traceview工具进行分析，他是一个分析器，记录了应用程序中每个函数的执行时间；我们可以打开DDMS然后选择一个进程，接着点击上面的“Start Method Profiling”按钮（红色小点变为黑色即开始运行），然后操作我们的卡顿UI（小范围测试，所以操作最好不要超过5s），完事再点一下刚才按的那个按钮，稍等片刻即可出现下图:

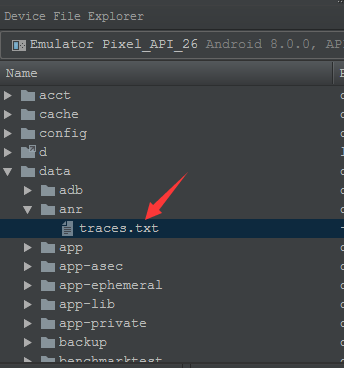


### ANR分析

是Android中AMS与WMS监测应用响应超时的表现

1、adb pull /data/anr/traces.txt ./

2、通过android studio导出:



### 优化点：

1、布局优化；尽量使用include、merge、ViewStub标签，尽量不存在冗余嵌套及过于复杂布局（譬如10层就会直接异常），尽量使用GONE替换INVISIBLE，使用weight后尽量将width和heigh设置为0dp减少运算，Item存在非常复杂的嵌套时考虑使用自定义Item View来取代，减少measure与layout次数等。

2、列表及Adapter优化；尽量复用getView方法中的相关View，不重复获取实例导致卡顿，列表尽量在滑动过程中不进行UI元素刷新等。

3、背景和图片等内存分配优化；尽量减少不必要的背景设置，图片尽量压缩处理显示，尽量避免频繁内存抖动等问题出现。

4、自定义View等绘图与布局优化；尽量避免在draw、measure、layout中做过于耗时及耗内存操作，尤其是draw方法中，尽量减少draw、measure、layout等执行次数。

1. 避免ANR，不要在UI线程中做耗时操作，遵守ANR规避守则，譬如多次数据库操作等。

## 内存优化

#### 内存泄漏

Leakcanary

[MAT](https://segmentfault.com/a/1190000006884310)

1、Context使用不当造成内存泄露；不要对一个Activity Context保持长生命周期的引用（譬如上面概念部分给出的示例）。尽量在一切可以使用应用ApplicationContext代替Context的地方进行替换（原理我前面有一篇关于Context的文章有解释）。

2、非静态内部类的静态实例容易造成内存泄漏；即一个类中如果你不能够控制它其中内部类的生命周期（譬如Activity中的一些特殊Handler等），则尽量使用静态类和弱引用来处理（譬如ViewRoot的实现）。

3、警惕线程未终止造成的内存泄露；譬如在Activity中关联了一个生命周期超过Activity的Thread，在退出Activity时切记结束线程。一个典型的例子就是HandlerThread的run方法是一个死循环，它不会自己结束，线程的生命周期超过了Activity生命周期，我们必须手动在Activity的销毁方法中中调运thread.getLooper().quit();才不会泄露。

4、对象的注册与反注册没有成对出现造成的内存泄露；譬如注册广播接收器、注册观察者（典型的譬如数据库的监听）等。

5、创建与关闭没有成对出现造成的泄露；譬如Cursor资源必须手动关闭，WebView必须手动销毁，流等对象必须手动关闭等。

6、不要在执行频率很高的方法或者循环中创建对象，可以使用HashTable等创建一组对象容器从容器中取那些对象，而不用每次new与释放。

7、避免代码设计模式的错误造成内存泄露。

#### 内存溢出

* 时刻记得不要加载过大的Bitmap对象；譬如对于类似图片加载我们要通过BitmapFactory.Options设置图片的一些采样比率和复用等，具体做法[点我参考官方文档](http://developer.android.com/intl/zh-cn/training/displaying-bitmaps/index.html)，不过过我们一般都用fresco或Glide开源库进行加载。
* 优化界面交互过程中频繁的内存使用；譬如在列表等操作中只加载可见区域的Bitmap、滑动时不加载、停止滑动后再开始加载。
* 有些地方避免使用强引用，替换为弱引用等操作。
* 避免各种内存泄露的存在导致OOM。
* 对批量加载等操作进行缓存设计，譬如列表图片显示，Adapter的convertView缓存等。
* 尽可能的复用资源；譬如系统本身有很多字符串、颜色、图片、动画、样式以及简单布局等资源可供我们直接使用，我们自己也要尽量复用style等资源达到节约内存。
* 对于有缓存等存在的应用尽量实现onLowMemory()和onTrimMemory()方法。
* 尽量使用线程池替代多线程操作，这样可以节约内存及CPU占用率。
* 尽量管理好自己的Service、Thread等后台的生命周期，不要浪费内存占用。
* 尽可能的不要使用依赖注入，中看不中用。
* 尽量在做一些大内存分配等可疑内存操作时进行try catch操作，避免不必要的应用闪退。
* 尽量的优化自己的代码，减少冗余，进行编译打包等优化对齐处理，避免类加载时浪费内存。

## 电量优化

[参考](https://tech.meituan.com/Dianping_Shortvideo_Battery_TestCase.html)：https://tech.meituan.com/Dianping\_Shortvideo\_Battery\_TestCase.html

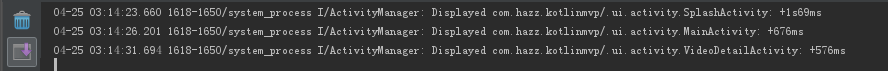
## App启动优化

冷启动：app第一次被打开，也就是该app进程第一次创建。

热启动：app从后台唤醒，或者说从其他app页面切换回来。

当用户点击桌面图标开始，系统会立即为这个APP创建独立的专属进程，然后显示启动窗口，直到APP在自己的进程里面完成了程序的创建以及主线程完成了Activity的初始化显示操作，再然后系统进程就会把启动窗口替换成APP的显示窗口。

首先我们可以通过log查看app启动的时间，当然你也可以看到每个activity的启动时间，因为每个activity的启动都有时间。



优化点：

Activity的初始化：优化布局、延迟加载数据（非立即需要的组件可以做延迟加载）

Application的初始化：oncreate()中不宜放太多的耗时操作、比如IO、网络、组件初始化操作。

用户视觉优化：App启动的时候系统启动窗口会出现短时间的白屏或则黑屏，我们可以用一张产品海报图片，window启动背景。

# 16、消息机制

# 17、Kotlin

## 空安全

一个方法返回值如果可以返回null或则一个变量初始化为空必须加“？”

Ps:

fun count(a:Int,b:Int):Int?{

return null

}

## 数据类型判断

Instanceof—>is

Ps：

Val obj:Any

if(obj is String){}

## 最顶层基类

Object🡪Any

## When 用法

When类似java的switch

Ps:

fun cases(obj: Any) {

when (obj) {

1 -> println("One")

"Hello" -> println("Greeting")

is Long -> println("Long")

!is String -> println("Not a string")

else -> println("Unknown")

}

}

## 循环遍历

循环集合、数组

for (arg in args)

println(arg)

// or

for (i in args.indices)

println(args[i])

范围循环

打印1、2、3、4、5

for (a in 1..5)

print("${a} ")

## 数据类

Ps:

data class User(val name: String, val id: Int)

fun getUser(): User {

return User("Alex", 1)

}

fun main(args: Array<String>) {

val user = getUser()

println("name = ${user.name}, id = ${user.id}")

// or

val (name, id) = getUser()

println("name = $name, id = $id")

// or

println("name = ${getUser().component1()}, id = ${getUser().component2()}")

}

## Map

Val map : mapOf(

"name" to "John Doe",

"age" to 25

)

fun main(args: Array<String>) {

val map = hashMapOf<String, Int>()

map.put("one", 1)

map.put("two", 2)

for ((key, value) in map) {

println("key = $key, value = $value")

}

}

## Delegate委托

## 扩展属性、方法

扩展属性、方法新建文件是File类型

fun Fragment.showToast(){

Toast.makeText(*activity*,".....fragment......",Toast.*LENGTH\_SHORT*).show()  
}

fun Activity.showToast(){  
 Toast.makeText(this,".....activty......",Toast.*LENGTH\_SHORT*).show()  
}

val <T> List<T>.*lastItem*:T  
get() = this.get(size-1)

页面跳转 ：newIntent<MainActivity>()

inline fun <reified T: Activity> Activity.newIntent() {  
 startActivity(Intent(this,T::class.*java*))  
}

## 伴生对象

相当于java的静态调用

关键词是companion object{ }

class ImageLoadUtils{  
 companion object{  
 fun display(context: Context, imageView: ImageView?, url: String) {  
 if (imageView == null) {  
 throw IllegalArgumentException("argument error")  
 }  
 Glide.with(context).load(url)  
 .diskCacheStrategy(DiskCacheStrategy.ALL)  
 .centerCrop()  
 .placeholder(R.drawable.*ic\_image\_loading*)  
 .error(R.drawable.*ic\_empty\_picture*)  
 .crossFade().into(imageView)  
 }  
 fun displayHigh(context: Context, imageView: ImageView?, url: String){  
 if (imageView == null) {  
 throw IllegalArgumentException("argument error")  
 }  
 Glide.with(context).load(url)  
 .asBitmap()  
 .format(DecodeFormat.PREFER\_ARGB\_8888)  
 .diskCacheStrategy(DiskCacheStrategy.ALL)  
 .centerCrop()  
 .placeholder(R.drawable.*ic\_image\_loading*)  
 .error(R.drawable.*ic\_empty\_picture*)  
 .into(imageView)  
 }  
 }  
  
}

## 单例

关键词是object

object KeyBoardUtils {  
  
 fun openKeyboard(context: Context, editText: EditText) {  
 val imm = context.getSystemService(Context.*INPUT\_METHOD\_SERVICE*) as InputMethodManager  
 imm.showSoftInput(editText, InputMethodManager.*RESULT\_SHOWN*)  
 imm.toggleSoftInput(InputMethodManager.*SHOW\_FORCED*, InputMethodManager.*HIDE\_IMPLICIT\_ONLY*)  
 }  
  
 fun closeKeyboard(context: Context, editText: EditText) {  
 val imm = context.getSystemService(Context.*INPUT\_METHOD\_SERVICE*) as InputMethodManager  
 imm.hideSoftInputFromWindow(editText.*windowToken*, 0)  
 }  
  
}

## 数据类序列化

配置：

Android{

androidExtensions {  
 experimental = true  
}

}

@Parcelize  
data class FindBean(var id: Int, var name: String?, var alias: Any?,  
 var description: String?, var bgPicture: String?,  
 var bgColor: String?, var headerImage: String?):Parcelable {

Any不能序列化，你的Java的Object可以序列化吗！不能！

}

## 常量

const val *SEARCH\_RELATED* = "http://baobab.kaiyanapp.com/api/v1/search?num=10&query=%E4%BD%A0&start=10"

## 内联函数（inline）

内联函数是C++的增强特性之一，用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联：编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段.

## ? !! ?: as?

?指定操作符可以为null，当你调用对象的属性的时候，必须加上？就是安全调用ps:

“name”?.length（非空判断）如果不加安全调用会报错，如果要输出null的结果可以这样：“name”?.length?:0

!! ps:name!!.length 只要name为空就会抛出空指针异常

As?:安全转换如果转化失败就会返回null

## 组件间跳转代码段

*newIntent*<MainActivity>("" *to* "","" *to* 1,"" *to* true)

inline fun <reified T : Activity> Activity.newIntent(vararg items : Pair<String, Any?>) {  
 val intent = Intent(this, T::class.*java*)  
 items.*forEach* **{** when (**it**.second) {  
 is String ->  
 intent.putExtra(**it**.first, **it**.second as String)  
 is Int ->  
 intent.putExtra(**it**.first, **it**.second as Int)  
 }  
  
 **}** startActivity(intent)  
}

# Kotlin实例

## 相机拍照

import kotlinx.android.synthetic.main.activity\_main3.\*  
  
class Main3Activity : AppCompatActivity() {  
 val CAMERA\_REQUEST\_CODE = 0  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main3*)  
 button.setOnClickListener **{** val callIntent = Intent(MediaStore.*ACTION\_IMAGE\_CAPTURE*)  
 if (callIntent.resolveActivity(*packageManager*) != null) {  
 startActivityForResult(callIntent, CAMERA\_REQUEST\_CODE)  
  
 }  
 **}** }  
  
  
 override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 super.onActivityResult(requestCode, resultCode, data)  
 when (requestCode) {  
 CAMERA\_REQUEST\_CODE -> {  
 if (resultCode == Activity.*RESULT\_OK* && data != null) {  
 imageView.setImageBitmap(data.*extras*.get("data") as Bitmap)  
 }  
 }  
 else -> {  
 Toast.makeText(this, "i don~t know!", Toast.*LENGTH\_SHORT*).show()  
 }  
 }  
 }  
}

## android 7.0 FileProvider

清单文件注册provider

<provider  
 android:name="android.support.v4.content.FileProvider"  
 android:authorities="${applicationId}.fileprovider"  
 android:exported="false"  
 android:grantUriPermissions="true">  
 <meta-data  
 android:name="android.support.FILE\_PROVIDER\_PATHS"  
 android:resource="@xml/filepath" />  
</provider>

Res下新建xml文件夹：

<?xml version="1.0" encoding="utf-8"?>  
<paths>  
<external-files-path  
 name="intent\_images"  
 path="Pictures"/>  
</paths>

Activity:

class Main3Activity : AppCompatActivity() {  
 val CAMERA\_REQUEST\_CODE = 0  
 lateinit var imageFilePath: String  
 override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_main3*)  
 button.setOnClickListener **{** try {  
 val imageFile = createImageFile()  
 val callIntent = Intent(MediaStore.*ACTION\_IMAGE\_CAPTURE*)  
 if (callIntent.resolveActivity(*packageManager*) != null) {  
 val authortites = *packageName* + ".fileprovider"  
 val imageUri = FileProvider.getUriForFile(this, authortites, imageFile)  
 callIntent.putExtra(MediaStore.*EXTRA\_OUTPUT*, imageUri)  
 startActivityForResult(callIntent, CAMERA\_REQUEST\_CODE)  
  
 }  
 } catch (e: Exception) {  
 Toast.makeText(this, "not find File", Toast.*LENGTH\_SHORT*).show()  
 }  
  
 **}** }  
  
  
 override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {  
 super.onActivityResult(requestCode, resultCode, data)  
 when (requestCode) {  
 CAMERA\_REQUEST\_CODE -> {  
 if (resultCode == Activity.*RESULT\_OK* && data != null) {  
 imageView.setImageBitmap(setScaleBitmap())  
 }  
 }  
 else -> {  
 Toast.makeText(this, "i don~t know!", Toast.*LENGTH\_SHORT*).show()  
 }  
 }  
 }  
  
 @Throws(IOException::class)  
 fun createImageFile(): File {  
 val timeStamp = SimpleDateFormat("yyyyMMdd\_HHmmss", Locale.*CHINA*).format(Date())  
 val imageFileName = "JPEG\_" + timeStamp  
 val storageDir = getExternalFilesDir(Environment.*DIRECTORY\_PICTURES*)  
 if (!storageDir.exists()) storageDir.mkdir()  
 val imageFile = *createTempFile*(imageFileName, ".jpg", storageDir)  
 imageFilePath = imageFile.*absolutePath* return imageFile  
 }  
  
 fun setScaleBitmap(): Bitmap {  
 val imageWidth = imageView.*width* val imageHeight = imageView.*height* val bmOptions = BitmapFactory.Options()  
 bmOptions.inJustDecodeBounds = true  
 BitmapFactory.decodeFile(imageFilePath, bmOptions)  
 val bitmapWidth = bmOptions.outWidth  
 val bitmapHeight = bmOptions.outHeight  
 val scaleFsctoe=Math.min(bitmapWidth/imageWidth,bitmapHeight/imageHeight)  
 bmOptions.inSampleSize=scaleFsctoe  
 bmOptions.inJustDecodeBounds = false  
 return BitmapFactory.decodeFile(imageFilePath)  
 }  
  
}

# 18、Butter Knife原理

[参考1](https://www.jianshu.com/p/bcddc376c0ef)

[参考2](https://segmentfault.com/a/1190000005994026#articleHeader5)

# 19、Mvc、 mvp、 mvvm、Android Architecture Components

## Mvc:

view:xml布局

model:实体、获取数据

controller:activity,处理数据、业务、操作UI

1. xml作为mvc的视图层功能太弱，大量处理view逻辑的操作都放在了activity
2. activity即充当view层又是controller层，这样就导致activity非常臃肿
3. 代码的解耦性差

## Mvp:

View：activity、fragment、xml

Model：实体、获取数据

Presenter:中间层，负责view和model的交互，view和model不能直接交互

优点：

1. mvp很好的解决的了mvc中activity充当2个角色的问题，通过抽象view层接口，让presenter层持有view接口，对接口进行操作，而不是直接操作view层中具体的activity,这样可以将视图层和业务逻辑分开
2. 具体流程：通过抽象的view层接口，让具体的view（activity）实现接口中的方法，且view持有presenter的引用，presenter作为中间者持有view接口的引用和model引用，当用户通过view层操作UI，会调用presenter层的逻辑，而persenter中持有model的引用，进而去调用model层具体的逻辑，到了model层，去执行真正的逻辑，逻辑执行完，通过接口回调的方式，回传给presenter，presenter回传给view层
3. Model和view通过presenter的隔离，降低了代码的耦合性

缺点：

1. presenter和view层通过接口交互，接口粒度不好控制，太小，就会存在大量接口，太大，解耦效果不好
2. UI的更新和数据变化，需要手动调用接口，缺乏自动性

## Mvvm:

# 20、RecyclerView和ListView 机制区别

Listview机制

1、RecycleBin机制:

RecycleBin是listview可以展示大量数据且不会oom的重要一个原因 ，RecycleBin存在于AbsListview中，RecycleBin当中使用mActiveViews这个数组来存储View，一旦被获取了之后就会从mActiveViews当中移除，再次获取同一位置时，会返回null,

mScrapViews和mCurrentScrap这两个List来存储废弃View

2、绘制流程

Onmeasure（）和普通的view绘制一样，ondraw（）不是listview负责的，而是子view，重点在于onlayout()方法，而且onlayout方法在AbsListview中，里面有个判断条件，如果listview的大小和位置发生改变，判断就是true，就会要求子布局强行重新绘制，接着layoutChildren（）这个方法就是子元素布局的，具体的实现是在listview中。

1. 第一次layout

第一次layout其实就是所有的子View都是调用LayoutInflater的inflate()方法加载出来的，所以第一次比较耗时，也就在我们重写适配器的getview的方法的时候第二个参数会判断一下，当前的view时候为空，第一次layout就相当于我们的view==null的时候，尽管我们的数据可能有大量数据，但是listview只会加载一屏的数据，其余的看不到，就不显示，这样能保证listview内容可以迅速显示出来。

1. 第二次layout

第一次layout的的子view会被填充到recyclerbin的数组中，所以这次直接取就好了

RecyclerView和ListView缓存机制类似，都是离屏的itemview被回收至内存，入屏的itemview优先从缓存中获取，只是实现的细节有差别。

1. 缓存层级不同

Recylerview比listview多两级缓存，支持多个离屏的itemview缓存，支持自定义缓存，支持所有的recyclerview公用一个recycleviewPool（缓存池）

[参考](https://zhuanlan.zhihu.com/p/23339185)

# 21、RxJava

## [中文文档](https://mcxiaoke.gitbooks.io/rxdocs/content/)

# 22、git命令

# 23、Java多态

事物在运行过程中存在不同的状态.

成立多态前提:

1.要有继承关系

2.子类要重写父类的方法

3.父类引用指向子类

成员变量，多态调用会执行父类

成员方法，多态调用会执行子类

静态方法，多态会执行父类

缺点：多态后不能使用自己特有的属性和方法，除非向下转型

# 24、Binder机制

[小米工程师gityuan](https://www.zhihu.com/question/39440766?utm_source=wechat_search&utm_medium=organic)

[参考](https://blog.csdn.net/qian520ao/article/details/78089877)

## 为什么Android要采用Binder作为IPC机制？

因为Android是基于Linux的，先说下Linux系统IPC方式：

1、管道：

2、消息队列：

3、共享内存：

4、套接字：

5、信号量：

6、信号：

为什么选用Binder？

1. 性能：Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要两次，共享内存无需拷贝
2. 稳定性：Bidner是基于C/S架构的，架构清晰明朗，Server端和Client相对独立，稳定性好，而内存共享实现方式复杂，需要考虑并发同步问题
3. 安全性：传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份；传统Linux IPC无任何保护措施，完全由上层协议来确保。Android为每个安装好的应用程序分配了自己的UID，故进程的UID是鉴别进程身份的重要标志，Bidner是基于C/S架构的，Android系统中对外只暴露Client端，Client端将任务发送给Server端，Server端会根据权限控制策略，判断UID/PID是否满足访问权限

25、Retrofit源码

[参考](https://blog.piasy.com/2016/06/25/Understand-Retrofit/)

[参考](https://segmentfault.com/a/1190000006767113)

# 26、反编译、混淆

## 反编译

分为：代码反编译、资源反编译

### 反编译代码：

Apk解压后取出**classes.dex文件，用工具dex2jar** 转化为jar包：d2j-dex2jar classes.dex，接着用**jd-gui查看class代码**  
  
资源反编译：

包括一些资源文件，比如清单文件、布局文件、xml文件等，方便以前，你是打不开的，即使打开也是编译过得，看不懂，所以我们可以借助apktool.bat和apktool.jar来来反编译资源： apktool d Demo.apk

### 重新打包：

还是借用apktool.bat和apktool.jar工具，执行命令行打包：apktool b 解压后包名 -o 新的.apk

当然现在是不能直接安装的，需要你找一个签名文件，重新签名，所以现在这个重新打包的apk就是盗版的，哈哈哈，重新签名：jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore 签名文件名 -storepass 签名密码 待签名的APK文件名 签名的别名。

其中jarsigner命令文件是存放在jdk的bin目录下的，需要将bin目录配置在系统的环境变量当中才可以在任何位置执行此命令

## 混淆代码

代码被反编译后，我们就可以随意修改代码，然后重新打包，但是签名是唯一的，即使你反编译了apk，重新打包没有签名文件，也是盗版，这个不重要，重要的是，我们的代码核心技术会泄漏出去，所以我们在发布之前一定要混淆代码，避免公司的技术泄漏。

## 混淆后代码无法反编译吗？

混淆代码不是让代码无法反编译，而是将代码中的类、方法、变量名等重命名，重命名后就是一些无意义的字符，使破解者难以理解程序的逻辑，从而提升程序的安全性。

# 27、Https

## Http缺点：

1. 明文通讯，容易被窃听
2. 不验证通信方身份，中间通信的过程可以伪装
3. 无法保证报文的完整性，中途可能发生篡改

HTTPS=HTTP+SSL/TLS

TLS 就是SSL的升级版

## TLS处于TCP/IP模型的那一层呢？

HTTP

TLS

TCP

处于传输层和应用层：

## 握手过程：

Client Server

协议版本

第一次生成随机数

支持的加密方法

压缩方法

确实加密协议是否支持

第二次生成随机数

确认加密方法，如：RSA

RSA公钥放入CA证书

校验证书：

如证书有问题发出警告。

如通过则：

生成随机数用公钥加密

这个随机数就是对称

用私钥解密公钥里面的随机数，也就是一会通讯用的对称加密密钥

## 客户端怎么校验证书呢？

数字签名生成过程：待签名内容->Hash->数字摘要->CA私钥加密->数字签名

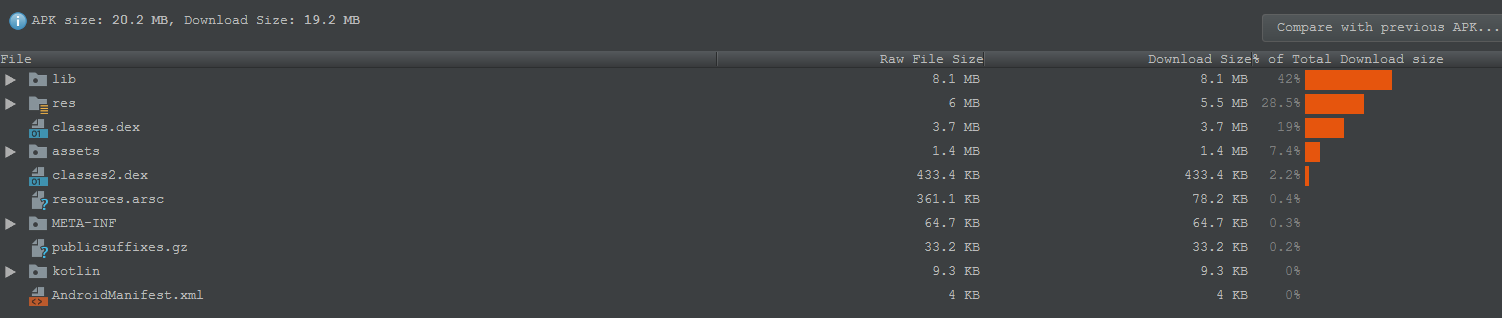
校验：

1. 数字签名-> CA公钥解密->数字摘要1
2. 待签名内容->Hash->数字摘要2
3. 对比数字摘要1和数字摘要2是否相等

# 28、减小APk体积

其实apk瘦身无非就是代码和资源两大块。

了解apk结构



Lib:so文件、jar文件

Res:一些资源文件，如音频、图片、xml文件（布局、颜色、动画等）

Classes.dex:字节码文件打包的.dex包，里面包括自己code和引用的库，dex格式文件有可能是多个，因为每个dex文件最多64k个方法，超出需要分包。

Assets:图片、音频

META-INF: 包含CERT.SF和 CERT.RSA签名文件以及MANIFEST.MF 清单文件

Kotlin:因为我用到kotlin的引用库，所以出现了kotlin

AndroidManifest.xml:app的清单文件，这个很熟悉了。

瘦身方法：

1. 在build.gradle文件中开启了minifEnabled与shrinkResources的属性，去除无用的代码和资源、开启混淆（代码中的方法名、类名、字段名经混淆后会大大缩短）。
2. 在符合条件的情况下，使用Vertor Drawable替代传统的PNG/JPEG图片，Vertor Drawable一份xml文件可以解决多套传统的PNG/JPEG的图片的适配问题。
3. 尽量复用已经存在的问题，比如需要一个向下、一个向右的箭头，你可以选择一个做一些旋转处理
4. 减少第三方库的引用，有什么功能最好参照别人的库，自己实现。（举个例子：下拉刷新库，他可能下拉刷新有很多样式，而你的产品只需要一种，所以，你参考着实现自己的就行了，如果全部引入，编译的时候，lib中的代码全部打包到你的apk~）
5. 避免枚举，一个枚举可以为您的应用程序的classes.dex文件添加大约1.0到1.4 KB的大小 。这些添加可以快速累积到复杂系统或共享库。如果可能，请考虑使用@IntDef注释，这种类型转换保留了枚举的所有类型安全优势。
6. 图片压缩工具
7. 定位Android 3.2（API级别13）或更高级别时 ，您也可以使用WebP文件格式来制作图像，而不是使用PNG或JPEG文件。WebP格式提供有损压缩（如JPEG）以及透明度（如PNG），但可以提供比JPEG或PNG更好的压缩。

Android 4.0 (API level 14) 支持有损压缩的WebP格式，Android 4.3 (API level 18) 开始支持无损透明WebP图像。

# 29、apk打包流程

## 打包的大致步骤：

1、通过aapt打包res资源文件，生成R.java、resources.arsc和res文件夹

2、如果有aidl文件，生成对应的java文件

3、通过Java Compiler编译R.java、Java源文件，生成.class文件

4、通过dex命令，将.class和第三方中.classs处理成classes.dex

5、通过apkbuilder工具将resources.arsc和res文件、assets文件和classes.dex、清单文件，一起打包生成apk

6、通过Jarsigner工具，对上面的apk进行debug或release签名

7、通过zipalign工具，将签名后的apk进行对齐处理

对齐处理：对齐的主要过程是将APK包中所有的资源文件距离文件起始偏移为4字节整数倍，这样通过内存映射访问apk文件时的速度会更快。

对齐的作用就是减少运行时内存的使用

[参考：http://shinelw.com/2016/04/27/android-make-apk/](http://shinelw.com/2016/04/27/android-make-apk/)

## 打包流程图：



# 30、http协议

## 概念

基于tcp/ip的一种应用层协议，全称超文本传输协议，用于浏览器和服务器之间的数据通信，特点：简单、快速、无状态（http本质是无状态的，使用cookie创建有状态会话），

## 资源

URI：统一资源标识符，用于标识http请求标识的资源

URL：统一资源定位符，一种特殊的URI

URN：统一资源命名，一种特殊的URI

URI组成：如<http://www.aspxfans.com:8080/news/index.asp?boardID=5&ID=24618&page=1#name>

协议：http

主机：www.aspxfans.com

端口：8080（默认http 80,https 443）

路径：/news/index.asp

参数：boardID=5&ID=24618&page=1

片段标识符：#name

## http连接

连接不属于http的范畴，是靠传输层来控制的，最常用的传输层协议就是：TCP、UDP，但TCP是可靠地，而UDP不是，因此http依赖TCP协议

http1.0为每一个请求\响应都打开一次连接（一次连接需要多次往返消息，如三次握手）导致速度慢，为了减轻这些缺点，http1.1引入持久连接，底层的TCP连接可以通过Connection: keep-alive头部来被部分控制，http2.0一个连接复用消息

http/2和http1.1基本不同

1、http2是二进制协议，不再可读。

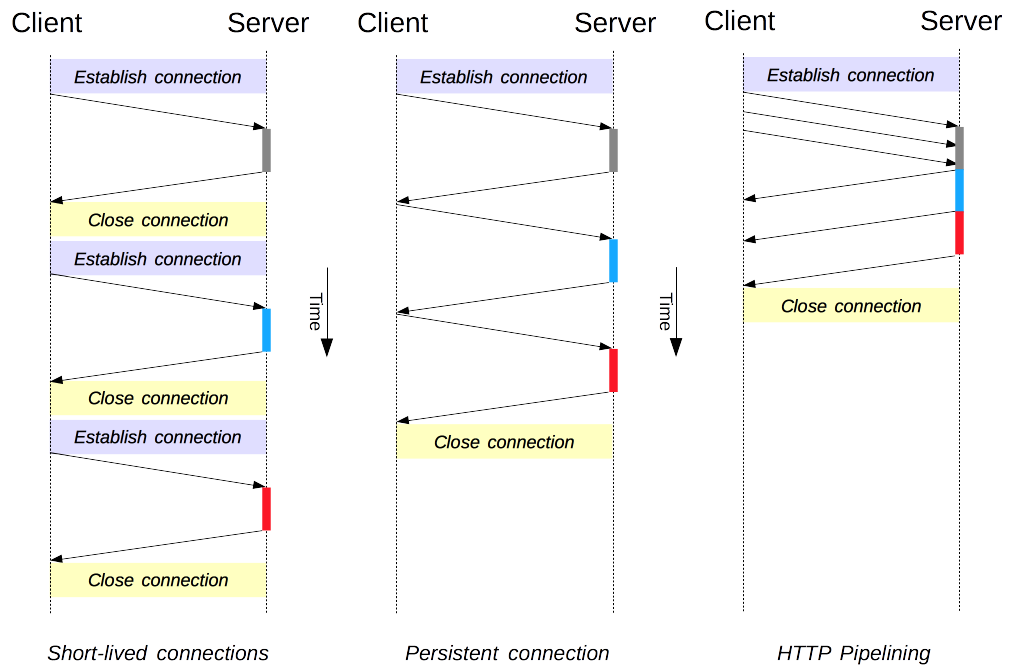
2、http2是一个复用协议，并行的请求在同一个连接中处理。

3、压缩header,header请求中常常是相似的，可以移除重复和传输重复的数据。

4、服务器在客户端缓存中填充数据、

5、HTTP/2 引入了一个额外的步骤：它将 HTTP/1.x 消息分成帧并嵌入到流 (stream) 中。数据帧和报头帧分离，这将允许报头压缩。将多个流组合，这是一个被称为 多路复用 (multiplexing) 的过程，它允许更有效的底层 TCP 连接

## 不同类型连接图例



短连接：每发起一个请求时都会创建一个新的连接，并在收到应答时立即关闭。

持久连接：保持连接去完成多次连续的请求，减少了不断重新打开连接的时间

http流水线：多个连续的请求甚至都不用等待立即返回就可以被发送，这样就减少了耗费在网络延迟上的时间

参考：<http://www.ruanyifeng.com/blog/2016/08/http.html>

## http/2对比http1.1的改进

1. 多路复用 (Multiplexing)：多路复用允许同时通过单一的 HTTP/2 连接发起多重的请求-响应消息
2. **二进制分帧（Binary Framing）:** **在 应用层(HTTP/2)和传输层(TCP or UDP)之间增加一个二进制分帧层**
3. **首部压缩（Header Compression）:** HTTP/1.1并不支持 HTTP 首部压缩，为此 SPDY 和 HTTP/2 应运而生，SPDY用DEFLATE算法，HTTP/2 HPACK算法

4、**服务端推送（Server Push）：在 HTTP/2 中，服务器可以对客户端的一个请求发送多个响应**

## SPDY

Google研发的协议，为了降低http1.1的延迟和提高安全性。2016年让路于http/2，已经停止维护。

参考：<https://www.zhihu.com/question/34074946>

Java反射

package com.zjzrjt.lib;  
  
import java.lang.reflect.Constructor;  
import java.lang.reflect.Field;  
import java.lang.reflect.InvocationTargetException;  
  
public class MyClass {*/\*\*  
 \* 为了看清楚Java反射部分代码，所有异常我都最后抛出来给虚拟机处理！  
 \** ***@param*** *args  
 \** ***@throws*** *ClassNotFoundException  
 \** ***@throws*** *InstantiationException  
 \** ***@throws*** *IllegalAccessException  
 \** ***@throws*** *InvocationTargetException  
 \** ***@throws*** *IllegalArgumentException  
 \** ***@throws*** *NoSuchFieldException  
 \** ***@throws*** *SecurityException  
 \** ***@throws*** *NoSuchMethodException  
 \*/*public static void main(String[] args) throws ClassNotFoundException, InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException, SecurityException, NoSuchFieldException, NoSuchMethodException {  
 // *TODO Auto-generated method stub* //Demo1. 通过Java反射机制得到类的包名和类名  
 *Demo1*();  
 System.*out*.println("===============================================");  
  
 //Demo2. 验证所有的类都是Class类的实例对象  
 *Demo2*();  
 System.*out*.println("===============================================");  
  
 //Demo3. 通过Java反射机制，用Class 创建类对象[这也就是反射存在的意义所在]，无参构造  
 *Demo3*();  
 System.*out*.println("===============================================");  
  
 //Demo4: 通过Java反射机制得到一个类的构造函数，并实现构造带参实例对象  
 *Demo4*();  
 System.*out*.println("===============================================");  
  
 //Demo5: 通过Java反射机制操作成员变量, set 和 get  
 *Demo5*();  
 System.*out*.println("===============================================");  
  
 //Demo6: 通过Java反射机制得到类的一些属性： 继承的接口，父类，函数信息，成员信息，类型等  
 *Demo6*();  
 System.*out*.println("===============================================");  
  
 //Demo7: 通过Java反射机制调用类中方法  
 *Demo7*();  
 System.*out*.println("===============================================");  
  
 //Demo8: 通过Java反射机制获得类加载器  
 *Demo8*();  
 System.*out*.println("===============================================");  
  
}  
  
 */\*\*  
 \* Demo1: 通过Java反射机制得到类的包名和类名  
 \*/* public static void Demo1()  
 {  
 Person person = new Person();  
 System.*out*.println("Demo1: 包名: " + person.getClass().getPackage().getName() + "，"  
 + "完整类名: " + person.getClass().getName());  
 }  
  
 */\*\*  
 \* Demo2: 验证所有的类都是Class类的实例对象  
 \** ***@throws*** *ClassNotFoundException  
 \*/* public static void Demo2() throws ClassNotFoundException  
 {  
 //定义两个类型都未知的Class , 设置初值为null, 看看如何给它们赋值成Person类  
 Class<?> class1 = null;  
 Class<?> class2 = null;  
  
 //写法1, 可能抛出 ClassNotFoundException [多用这个写法]  
 class1 = Class.*forName*("cn.lee.demo.Person");  
 System.*out*.println("Demo2:(写法1) 包名: " + class1.getPackage().getName() + "，"  
 + "完整类名: " + class1.getName());  
  
 //写法2  
 class2 = Person.class;  
 System.*out*.println("Demo2:(写法2) 包名: " + class2.getPackage().getName() + "，"  
 + "完整类名: " + class2.getName());  
 }  
  
 */\*\*  
 \* Demo3: 通过Java反射机制，用Class 创建类对象[这也就是反射存在的意义所在]  
 \** ***@throws*** *ClassNotFoundException  
 \** ***@throws*** *IllegalAccessException  
 \** ***@throws*** *InstantiationException  
 \*/* public static void Demo3() throws ClassNotFoundException, InstantiationException, IllegalAccessException  
 {  
 Class<?> class1 = null;  
 class1 = Class.*forName*("cn.lee.demo.Person");  
 //由于这里不能带参数，所以你要实例化的这个类Person，一定要有无参构造函数哈～  
 Person person = (Person) class1.newInstance();  
 person.setAge(20);  
 person.setName("LeeFeng");  
 System.*out*.println("Demo3: " + person.getName() + " : " + person.getAge());  
 }  
  
 */\*\*  
 \* Demo4: 通过Java反射机制得到一个类的构造函数，并实现创建带参实例对象  
 \** ***@throws*** *ClassNotFoundException  
 \** ***@throws*** *InvocationTargetException  
 \** ***@throws*** *IllegalAccessException  
 \** ***@throws*** *InstantiationException  
 \** ***@throws*** *IllegalArgumentException  
 \*/* public static void Demo4() throws ClassNotFoundException, IllegalArgumentException, InstantiationException, IllegalAccessException, InvocationTargetException  
 {  
 Class<?> class1 = null;  
 Person person1 = null;  
 Person person2 = null;  
  
 class1 = Class.*forName*("cn.lee.demo.Person");  
 //得到一系列构造函数集合  
 Constructor<?>[] constructors = class1.getConstructors();  
  
 person1 = (Person) constructors[0].newInstance();  
 person1.setAge(30);  
 person1.setName("leeFeng");  
  
 person2 = (Person) constructors[1].newInstance(20,"leeFeng");  
  
 System.*out*.println("Demo4: " + person1.getName() + " : " + person1.getAge()  
 + " , " + person2.getName() + " : " + person2.getAge()  
 );  
  
 }  
  
 */\*\*  
 \* Demo5: 通过Java反射机制操作成员变量, set 和 get  
 \*  
 \** ***@throws*** *IllegalAccessException  
 \** ***@throws*** *IllegalArgumentException  
 \** ***@throws*** *NoSuchFieldException  
 \** ***@throws*** *SecurityException  
 \** ***@throws*** *InstantiationException  
 \** ***@throws*** *ClassNotFoundException  
 \*/* public static void Demo5() throws IllegalArgumentException, IllegalAccessException, SecurityException, NoSuchFieldException, InstantiationException, ClassNotFoundException  
 {  
 Class<?> class1 = null;  
 class1 = Class.*forName*("cn.lee.demo.Person");  
 Object obj = class1.newInstance();  
  
 Field personNameField = class1.getDeclaredField("name");  
 personNameField.setAccessible(true);  
 personNameField.set(obj, "胖虎先森");  
  
  
 System.*out*.println("Demo5: 修改属性之后得到属性变量的值：" + personNameField.get(obj));  
  
 }  
  
  
 */\*\*  
 \* Demo6: 通过Java反射机制得到类的一些属性： 继承的接口，父类，函数信息，成员信息，类型等  
 \** ***@throws*** *ClassNotFoundException  
 \*/* public static void Demo6() throws ClassNotFoundException  
 {  
 Class<?> class1 = null;  
 class1 = Class.*forName*("cn.lee.demo.SuperMan");  
  
 //取得父类名称  
 Class<?> superClass = class1.getSuperclass();  
 System.*out*.println("Demo6: SuperMan类的父类名: " + superClass.getName());  
  
 System.*out*.println("===============================================");  
  
  
 Field[] fields = class1.getDeclaredFields();  
 for (int i = 0; i < fields.length; i++) {  
 System.*out*.println("类中的成员: " + fields[i]);  
 }  
 System.*out*.println("===============================================");  
  
  
 //取得类方法  
 Method[] methods = class1.getDeclaredMethods();  
 for (int i = 0; i < methods.length; i++) {  
 System.*out*.println("Demo6,取得SuperMan类的方法：");  
 System.*out*.println("函数名：" + methods[i].getName());  
 System.*out*.println("函数返回类型：" + methods[i].getReturnType());  
 System.*out*.println("函数访问修饰符：" + Modifier.toString(methods[i].getModifiers()));  
 System.*out*.println("函数代码写法： " + methods[i]);  
 }  
  
 System.*out*.println("===============================================");  
  
 //取得类实现的接口,因为接口类也属于Class,所以得到接口中的方法也是一样的方法得到哈  
 Class<?> interfaces[] = class1.getInterfaces();  
 for (int i = 0; i < interfaces.length; i++) {  
 System.*out*.println("实现的接口类名: " + interfaces[i].getName() );  
 }  
  
 }  
  
 */\*\*  
 \* Demo7: 通过Java反射机制调用类方法  
 \** ***@throws*** *ClassNotFoundException  
 \** ***@throws*** *NoSuchMethodException  
 \** ***@throws*** *SecurityException  
 \** ***@throws*** *InvocationTargetException  
 \** ***@throws*** *IllegalAccessException  
 \** ***@throws*** *IllegalArgumentException  
 \** ***@throws*** *InstantiationException  
 \*/* public static void Demo7() throws ClassNotFoundException, SecurityException, NoSuchMethodException, IllegalArgumentException, IllegalAccessException, InvocationTargetException, InstantiationException  
 {  
 Class<?> class1 = null;  
 class1 = Class.*forName*("cn.lee.demo.SuperMan");  
  
 System.*out*.println("Demo7: \n调用无参方法fly()：");  
 Method method = class1.getMethod("fly");  
 method.invoke(class1.newInstance());  
  
 System.*out*.println("调用有参方法walk(int m)：");  
 method = class1.getMethod("walk",int.class);  
 method.invoke(class1.newInstance(),100);  
 }  
  
 */\*\*  
 \* Demo8: 通过Java反射机制得到类加载器信息  
 \*  
 \* 在java中有三种类类加载器。[这段资料网上截取]  
 1）Bootstrap ClassLoader 此加载器采用c++编写，一般开发中很少见。  
 2）Extension ClassLoader 用来进行扩展类的加载，一般对应的是jre\lib\ext目录中的类  
 3）AppClassLoader 加载classpath指定的类，是最常用的加载器。同时也是java中默认的加载器。  
 \*  
 \** ***@throws*** *ClassNotFoundException  
 \*/* public static void Demo8() throws ClassNotFoundException  
 {  
 Class<?> class1 = null;  
 class1 = Class.*forName*("cn.lee.demo.SuperMan");  
 String nameString = class1.getClassLoader().getClass().getName();  
  
 System.*out*.println("Demo8: 类加载器类名: " + nameString);  
 }  
  
  
  
}  
*/\*\*  
 \*  
 \** ***@author*** *xiaoyaomeng  
 \*  
 \*/*class Person{  
 private int age;  
 private String name;  
 public Person(){  
  
 }  
 public Person(int age, String name){  
 this.age = age;  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
}  
  
class SuperMan extends Person implements ActionInterface  
{  
 private boolean BlueBriefs;  
  
 public void fly()  
 {  
 System.*out*.println("超人会飞耶～～");  
 }  
  
 public boolean isBlueBriefs() {  
 return BlueBriefs;  
 }  
 public void setBlueBriefs(boolean blueBriefs) {  
 BlueBriefs = blueBriefs;  
 }  
  
 @Override  
 public void walk(int m) {  
 // *TODO Auto-generated method stub* System.*out*.println("超人会走耶～～走了" + m + "米就走不动了！");  
 }  
}  
interface ActionInterface{  
 public void walk(int m);  
}  
  
}