**性能优化点**

1. 绘制 : 布局重叠 ， 多个layout背景重复设置，移除不需要背景的layout的bg
2. 电量：网络请求相对耗电 ， 非即时性请求可以延迟打包操作，切换到wifi或者充电状态下执行 JobScheduler ,GCMNetworkManager

a)传输方面： 预加载：某次操作之后，用户可能会马上请求的几次数据集中到一次执行完毕

b)压缩 ：上传和加载数据之前，对数据压缩和解压

3、短时间内创建大量对象，导致内存紧张的问题，使用对象池技术 ，例如bitmaps，views，paints

4、遍历操作 尽量采用Index的遍历 for(int index = 0 ; index < size ; index ++){...}; 这种遍历效率更高，

下面两种效率比上面效率低，下第一个效率最低

for (Iterator it = list.iterator() ; it.hasNext();){

Object obj = it.next();

}

for(Object obj : list){

...

}

5、使用Android Studio Lint工具查看源码并找出潜在问题

6、避免在onDraw方法里进行内存分配，。设备有一定的刷新频率，导致View的onDraw方法会被频繁的调用， 例如new Paint()可以移动到ondraw()外面

7、使用Android 提供的Strict Mode找出程序潜在的隐患，出现问题可以对屏幕闪红色，也可以输出错误日志，下面是官方示例代码

public void onCreate() {

if (DEVELOPER\_MODE) {

StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()

.detectDiskReads()

.detectDiskWrites()

.detectNetwork() // or .detectAll() for all detectable problems

.penaltyLog()

.build());

StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()

.detectLeakedSqlLiteObjects()

.detectLeakedClosableObjects()

.penaltyLog()

.penaltyDeath()

.build());

}

super.onCreate();

}

8、自定义View的优化

a)仅仅在View的内容发生改变的时候才去触发invalidate方法

b)尽量使用ClipRect等方法来提高绘制的性能。

c)减少绘制时不必要的绘制元素，对于那些不可见的元素，我们需要尽量避免重绘。

d)不在屏幕上的元素，可以使用Canvas.quickReject把他们给剔除，避免浪费CPU资源。另外尽量使用GPU来进行UI的渲染，这样能够极大的提高程序的整体表现性能。

9、例如第二点，将一定的后台任务打包处理，在一定的时机一起处理，可以通过AlarmManager ， SyncAdapter ，JobSchedulor实现

10、图片：

a)解码：随着解码占用内存大小的降低，清晰度也会有损失。我们需要针对不同的应用场景做不同的处理，大图和小图可以采用不同的解码率。

b)尽量减少PNG图片的大小是Android里面很重要的一条规范，比起JPEG，PNG能够提供更加清晰无损的图片，但是PNG格式的图片会更大，占用更多的磁盘空间，对于那些使用JPEG就可以达到视觉效果的，可以考虑采用JPEG即可

c)新图片格式:Webp，它是由Google推出的一种既保留png格式的优点

11、bitmap的优化，，Glide，这里面包含了各种对bitmap的优化技巧。

12、容器的使用： HaspMap好用但是很占内存，Android提供了内存效率更高的ArrayMap ， Android系统还提供了SparseBoolMap，SparseIntMap，SparseLongMap，LongSparseMap等容器。

ArrayMap<Object , Object> arrayMap = new ArrayMap<>();

arrayMap.put(); arrayMap.get();

for(int i = 0 ; i < arrayMap.size() ; i ++){

Object key = arrayMap.keyAt(i);

Object val = arrayMap.valueAt(i);

}

适用：对象个数的数量级最好是千以内 , 数据组织形式包含Map结构

13、Android官方强烈建议不要在Android程序里面使用到enum。使用enum编译成dex后会使dex增大，使用enum还会产生额外的运行内存 ，可以使用普通static常量相关作为判断值

14、避免使用异步回调 ， 避免使用Static对象 ？？？ 避免把View添加到没有清除机制的容器里面

15、网络数据的缓存

a)Android系统上关于网络请求的Http Response Cache是默认关闭的...

b)我们可以使用Volly，okHTTP，Picasso来实现网络缓存

16、避免客户端对服务器的轮询操作，这样会浪费很多的电量与带宽流量。解决这个问题，我们可以使用Google Cloud Message来对更新的数据进行推送。集成推送之后再说

17、使用Service: Service是默认运行在UI线程的，这意味着Service可能会影响到系统的流畅度。

a)避免错误的使用Service，例如我们不应该使用Service来监听某些事件的变化，不应该搞一个Service在后台对服务器不断的进行轮询(应该使用Google Cloud Messaging)

b)如果已经事先知道Service里面的任务应该执行在后台线程(非默认的主线程)的时候，我们应该使用IntentService或者结合HanderThread，AsycnTask Loader实现的Service。

c)注意正确的关闭Service。

18、Android为我们提供了Proguard的工具来帮助应用程序对代码进行瘦身，优化，混淆的处理 ， 但是我们需要手动的把这些需要保留的类名与方法名添加到Proguard的配置文件中

19、使用Gradle来帮助我们分析代码，分析引用的资源，对于那些没有被引用到的资源，会在编译阶段被排除在APK安装包之外

a)要实现这个功能，对我们来说仅仅只需要在build.gradle文件中配置shrinkResource为true就好了

b)Gradle目前无法对values，drawable等根据运行时来决定使用的资源进行优化，对于这些资源，需要我们自己来确保资源不会有冗余。 我们可以通过tools:keep或者是tools:discard标签来实现对特定资源的保留与废弃

20、序列化，传统的Serializable接口效率不高，消耗更多的内存， GSON库可以处理这个序列化的问题，不仅仅执行速度更快，内存的使用效率也更高 ， 三个候选方案：Protocal Buffers ， Nano-Proto-Buffers ， FlatBuffers；

21、优化数据结构

22、 UI界面数据缓存，先呈现旧的数据，同时加载，不至于在加载过程中没有数据

23、多线程问题：增加并发的线程数会导致内存消耗的增加 ， 多线程并发访问同一块内存区域有可能带来很多问题，例如读写的权限争夺问题，ABA问题等等。为了解决这些问题，我们会需要引入锁的概念。不要在任何非UI线程里面去持有UI对象的引用。

24、谨慎使用AsyncTask