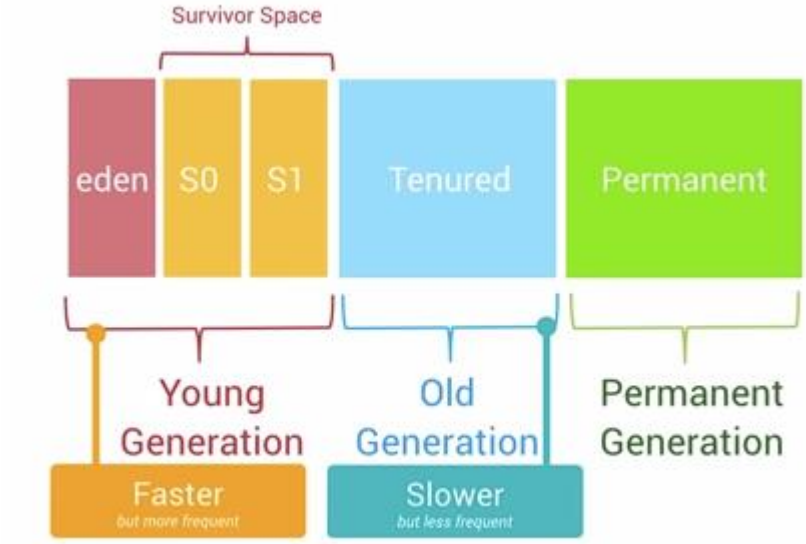
一　首先，我们先了解一下都有哪些性能问题

1、内存泄露

通俗来讲，内存泄露不仅仅会造成应用内存占用过大，还会导致应用卡顿，造成不好的用户体验，至于，为什么一个“小小的”内存泄露会造成应用卡顿，我不得不拿这幅图来说说话了。



没错，这就是 Android 开发童鞋需要了解的 Generational Heap Memory 模型，这里我们只关心当对象在 Young Generation 中存活了一段时间之后，如果没被干掉，那么会被移动到 Old Generation 中，同理，最后会移动到 Permanent Generation 中。那么用脚想一想就知道，如果内存泄露了，那么，抱歉，你那块内存随时间推移自然而然将进入 Permanent Generation 中，然鹅，内存不是白菜，想要多少就有多少，这里，因为沙盒机制的原因，分配给你应用的内存当然是有那么一个极限值的，你不能逾越（有人笑了，不是有 large heap 么，当然我也笑了，我并没有看到这货被宗师 android 玩家青睐过），好了，你那块造成泄露内存的对象占着茅坑不拉屎，剩下来可以供其他对象发挥的内存空间就少了；打个比方，舞台小了，演员要登台表演，没有多余空间，他就只能等待其他演员下来他才能表演啊，这等待的时间，是没法连续表演的，所以就卡了嘛。

2、频繁 GC

呵呵，频繁 GC 会造成卡顿，想必你经过上面的洗礼，已经知道了为什么，不错，当然也是

因为“舞台空间不足，新的演员上台表演需要先让表演完的下来”。那么造成这种现象的原因是什么呢？

a、内存泄露，好的，你懂了，不用讲了，这个必须有可能会造成。

b、大量对象短时间被创建，又在短时间内“需要”被释放，注意这里的需要，其实是不得不，为什么，同样是因为“舞台空间不够了”，举个例子，在 onDraw 中 new 对象，因为 onDraw 大约 16ms 会执行一次（wait，你能否确定一下，什么是大约 16ms，对不起，不能，掉帧了就不是，哪怕掉那么一点点）。脑补一下，每秒中创建大约 60 个对象，嗯，骚年，你以为 Young Generation 是白菜么，想拿多少就拿多少，对不起，这里是限量的，这里用完了，在来申请，我就得去回收一些回来，我回收总得耗时间吧，耗时间，好吧，onDraw 等着等着就错过了下一个 16ms 的执行了，如是，用户看起来就卡了。

3、耗电问题

km 上有一个问题很尖锐，说是微视看小视频看一会手机就会发烫，所以，用户一直就很关注耗电问题，不过不好意思，我们的 app 至今还没有遇到过严重的耗电问题，虽然没有遇到比较严重的耗电问题，不代表就不需要去了解这样的问题的解决办法，我总结有：

a、没有什么特别重要的信息，比如，钱到账，电话来了，100 元实打实无门槛代金券方法，等等，请不要打扰用户，不要频繁唤醒用户，否则，结果只能是卸载，或者关闭一切通知。

b、适当的做本地缓存，避免频繁请求网络数据，这里，说起来容易，做起来并非三刀两斧就能搞定，要配合良好的缓存策略，区分哪些是一段时间不会变更的，哪些是绝对不能缓存的很重要。

c、对某些执行时间较长的同步操作在用户充电且有 wifi 的时候在做，除非用户强制同步..等等，就不扯太多，因为后面还有很多内容。

4、OOM 问题

呵呵，这个问题，想必经过前面 1、2 的洗礼，你应该已经明白这个什么原因导致的，你可以想想一下"舞台上将要上的一个演员是一个巨大胖子，即便不表演的演员都下来了，他还是挤不上去，怎么办，演砸了，还能怎么办，直接崩溃，散场！"造成这个问题的原因，可能有，（呵呵，保险起见，只能说可能，分析的时候可以从这里出发）

a、内存泄露了，想必你会心一笑。

b、大量不可见的对象占据内存，这个其实，很常见，只是大家可能一直不太关心罢了，比如，请求接口返回了列表有 100 项数据，每项数据比如有 100 个字段，其中你用户展示数据的只有 10 几个而已，但是，你解析的时候，剩下的 99 个不知不觉吃了你的内存，当，有个胖子要内存时，呵呵，嗝屁了。

c、还有一种很常见的场景是一个页面多图的场景，明明每个图只需要加载一个 100*100 的，你却使用原始尺寸（1080*1980）or 更大，而且你一下子还加载个几十张，扛得住么？所以了解一下 inSampleSize，或者，如果图片归你们上传管理，你可以借助万象优图，他为你做了剪切好不同尺寸的图片，这样省得你在客户端做图片缩放了。

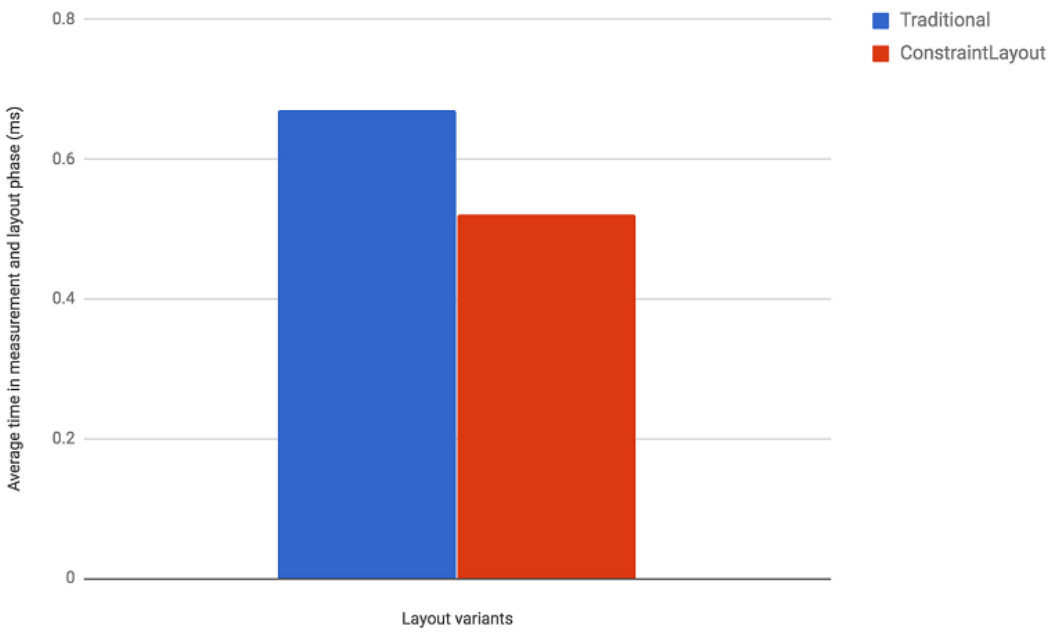
二　以上了解了一些性能问题，这里，简单的串一串导致这些性能问题的原因

1、人为在 ui 线程中做了轻微的耗时操作，导致 ui 线程卡顿

嗯，很多小伙伴不以为然，以为在 onCreate 中读一下 pref 算什么，解析下 json 数据算得了什么，可实际情况是并不是这样的，正确的做法是，将这些操作使用异步封装起来，小伙伴可以了解一下 rxjava，现在最新版本已经是 rxjava2 了，如果不清楚使用方式，可以 Google 一下。

2、layout 过于复杂，无法在 16ms 完成渲染

这个很多小伙伴深有体会了，这里简单的了解下，我们先简单的把渲染大概分为"layout","measure""draw"这么几个阶段，当然你不要以为实际情况也是如此，好，层级复杂，layout,measure 可能就用到了不该用的时间，自然而然，留给 draw 的时间就可能不够了，自然而然就悲剧了。那么以前给出的很多建议是，使用 RelativeLayout 替换 LinearLayout,说是可以减少布局层次，然鹅，现在请不要在建议别人使用 RelativeLayout，因为 ConstraintLayout 才是一个更高性能的消灭布局层级的神器。ConstraintLayout 基于 Cassowary 算法，而 Cassowary 算法的优势是在于解决线性方程时有极高的效率，事实证明，线性方程组是非常适合用于定义用户界面元素的参数。由于人们对图形的敏感度非常高，所以 UI 的渲染速度显得非常重要。因此在 2016 年，iOS 和 Android 都基于 Cassowary 算法来研发了属于自己的布局系统，这里是 ConstraintLayout 与传统布局 RelativeLayout，LinearLayout 实现时的性能对比，不过这里是老外的测试数据，原文可以参考这里。demo 中也提供了测试的方法，感兴趣的小伙伴可以尝试一下咯。



3、同一时间执行的动画过多，导致 CPU 或者 GPU 负载过重

这里主要是因为动画一般会频繁变更 view 的属性，导致 displayList 失效，而需要重新创建一个新的 displayList，如果动画过多，这个开销可想而知，如果你想了解得更加详细，推荐看这篇咯，知识点在第 5 节那里。

4、view 过度绘制的问题。

view 过度绘制的问题可以说是我们在写布局的时候遇到的一个最常见的问题之一，可以说写着写着一不留神就写出了一个过度绘制，通常发生在一个嵌套的 viewgroup 中，比如你给他设置了一个不必要的背景。这方面问题的排查不太难，我们可以通过手机设置里面的开发者选项，打开 Show GPU Overdraw 的选项，轻松发现这些问题，然后尽量往蓝色靠近。

5、gc 过多的问题，这里就不在赘述了，上面已经讲的非常直接了。

6、资源加载导致执行缓慢。

有些时候避免不要加载一些资源，这里有两种解决的办法，使用的场景也不相同。

a、预加载，即还没有来到路径之前，就提前加载好，诶，好像 x5 内核就是酱紫哦。

b、实在是要等到用到的时候加载，请给一个进度条，不要让用户干等着，也不知道什么时候结束而造成不好的用户体验。

7、工作线程优先级设置不对，导致和 ui 线程抢占 cpu 时间。

使用 Rxjava 的小伙伴要注意这点，设置任务的执行线程可能会对你的性能产生较大的影响，没有使用的小伙伴也不能太过大意。

8、静态变量。

嘿嘿，大家一定有过在 application 中设置静态变量的经历，遥想当年，为了越过 Intent 只能传递 1M 以下数据的坑，我在 application 中设置了一个静态变量，用于两个 activity“传递（共享）数据”，然而，一步小心，数据中，有着前一个 activity 的尾巴，因此泄露了。不光是这样的例子，随便举几个:

a、你用静态集合保存过数据吧?

b、某某单例的 Manger，比如管理 AudioManger 遇到过吧?

三　既然遇到问题分析也有了，那么接下来，自然而然是如何使用各种刀棒棍剑来解决这些问题了

1、GPU 过度绘制，定位过度绘制区域

这里直接在开发者选项，打开 Show GPU Overdraw，就可以看到效果，轻松发现哪块需要优化，那么具体如何去优化

a、减少布局层级，上面有提到过，使用 ConstraintLayout 替换传统的布局方式。如果你对 ConstraintLayout 不了解, 没有关系, 这篇文章教你 15 分钟了解如何使用 ConstraintLayout。

b、检查是否有多余的背景色设置，我们通常会犯一些低级错误--对被覆盖的父 view 设置背景，多数情况下这些背景是没有必要的。

2、主线程耗时操作排查。

a、开启 strictmode,这样一来，主线程的耗时操作都将以告警的形式呈现到 logcat 当中。

b、直接对怀疑的对象加@DebugLog，查看方法执行耗时。DebugLog 注解需要引入插件 hugo，这个是 Android 之神 JakeWharton 的早期作品，对于监控函数执行时间非常方便，直接在函数上加入注解就可以实现，但是有一个缺点，就是 JakeWharton 发布的最后一个版本没有支持 release 版本用空方法替代监控代码，因此，我这里发布了一个到公司的 maven 仓库，引用的方式和官网类似，只不过，地址是：'com.tencent.tip:hugo-plugin:2.0.0-SNAPSHOT'。

3、对于 measure，layout 耗时过多的问题

一般这类问题是优于布局过于复杂的原因导致，现在因为有 ConstraintLayout，所以，强烈建议使用 ConstraintLayout 减少布局层级，问题一般得以解决，如果发现还存在性能问题，可以使用 traceView 观察方法耗时，来定位下具体原因。

4、leakcany

这个是内存泄露监测的银弹，大家应该都使用过，需要提醒一下的是，要注意

```
dependencies {
debugImplementation 'com.squareup.leakcanary:leakcanary-android:1.5.4'
releaseImplementation 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.4'
}
```

引入方式，releaseImplementation 保证在发布包中移除监控代码，否则，他自生不停的 catch 内存快照，本身也影响性能。

5、onDraw 里面写代码需要注意

onDraw 优于大概每 16ms 都会被执行一次，因此本身就相当于一个 forloop，如果你在里面 new 对象的话，不知不觉中就满足了短时间内大量对象创建并释放，于是频繁 GC 就发生了，嗯，内存抖动，于是，卡了。因此，正确的做法是将对象放在外面 new 出来。

6、json 反序列化问题

json 反序列化是指将 json 字符串转变为对象，这里如果数据量比较多，特别是有相当多的 string 的时候，解析起来不仅耗时，而且还很吃内存。解决的方式是：

a、精简字段，与后台协商，相关接口剔除不必要的字段。保证最小可用原则。

b、使用流解析，之前我考虑过 json 解析优化，在 Stack Overflow 上搜索到这个。于是了解到 Gson.fromJson 是可以这样玩的，可以提升 25%的解析效率。

```
public List<Message> readJsonStream(InputStream in) throws IOException {
        JsonReader reader = new JsonReader(new InputStreamReader(in, "UTF-8"));
        List<Message> messages = new ArrayList<Message>();
        reader.beginArray();
        while (reader.hasNext()) {
            Message message = gson.fromJson(reader, Message.class);
            messages.add(message);
        }
        reader.endArray();
        reader.close();
        return messages;
}
```

7、viewStub&merge 的使用。

这里 merge 和 viewStub 想必是大家非常了解的两个布局组件了，对于只有在某些条件下才展示出来的组件，建议使用 viewStub 包裹起来，同样的道理，include 某布局如果其根布局和引入他的父布局一致，建议使用 merge 包裹起来，如果你担心 preview 效果问题，这里完全没有必要，因为你可以
tools:showIn=""属性，这样就可以正常展示 preview 了。

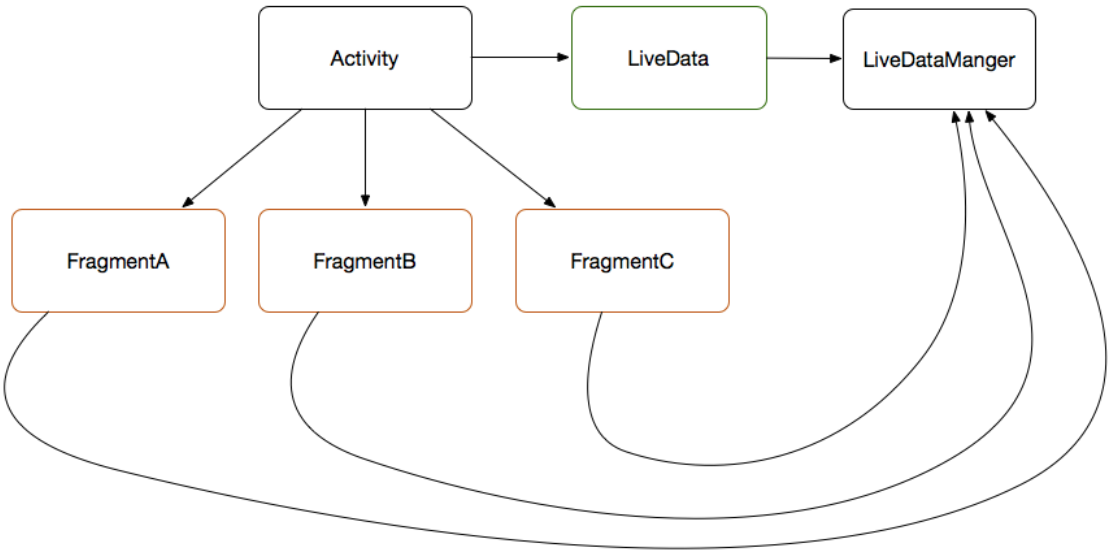
8、加载优化

这里并没有过多的技术点在里面，无非就是将耗时的操作封装到异步中去了，但是，有一点不得不提的是，要注意多进程的问题，如果你的应用是多进程，你应该认识到你的 application 的 oncreate 方法会被执行多次，你一定不希望资源加载多次吧，于是你只在主进程加载，如是有些坑就出现了，有可能其他进程需要那某份资源，然后他这个进程缺没有加载相应的资源，然后就嗝屁了。

9、刷新优化。

这点在我之前的文章中有提到过，这里举两个例子吧。
a、对于列表的中的 item 的操作，比如对 item 点赞，此时不应该让整个列表刷新，而是应该只刷新这个 item，相比对于熟练使用 recyclerView 的你，应该明白如何操作了,不懂请看这里，你将会明白什么叫做 recyclerView 的局部刷新。

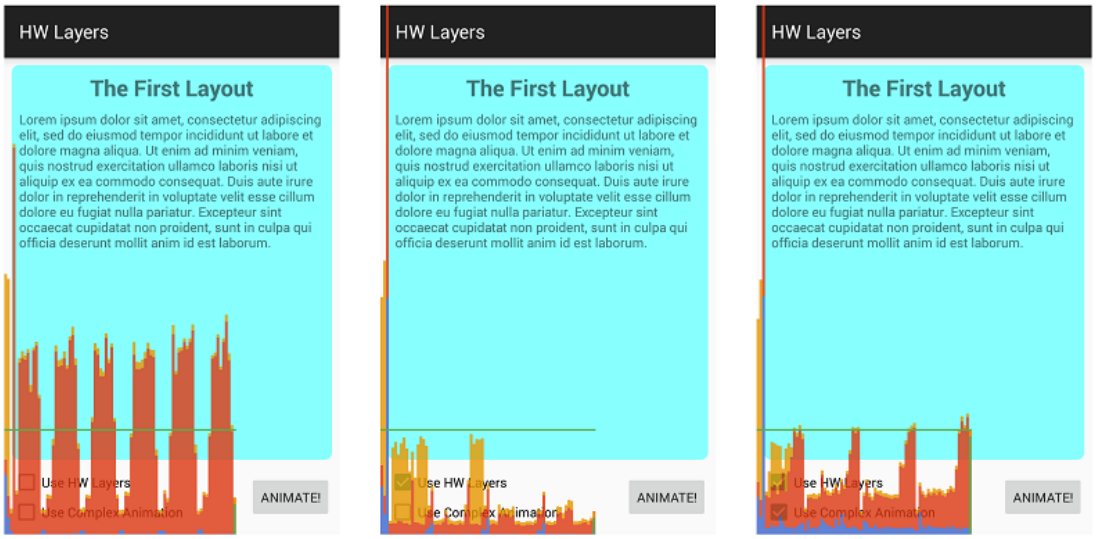
b、对于较为复杂的页面，个人建议不要写在一个 activity 中，建议使用几个 fragment 进行组装，这样一来，module 的变更可以只刷新某一个具体的 fragment，而不用整个页面都走刷新逻辑。但是问题来了，fragment 之间如何共享数据呢？好，看我怎么操作。



Activity 将数据这部分抽象成一个 LiveData，交个 LiveDataManger 数据进行管理，然后各个 Fragment 通过 Activity 的这个 context 从 LiveDataManger 中拿到 LiveData,进行操作，通知 activity 数据变更等等。哈哈，你没有看错，这个确实和 Google 的那个 LiveData 有点像，当然，如果你想使用 Google 的那个，也自然没问题，只不过，这个是简化版的。项目的引入 'com.tencent.tip:simple_live_data:1.0.1-SNAPSHOT'

10、动画优化

这里主要是想说使用硬件加速来做优化，不过要注意，动画做完之后，关闭硬件加速，因为开启硬件加速本身就是一种消耗。下面有一幅图，第二幅对比第一幅是说开启硬件加速和没开启的时候做动画的效果对比，可以看到开启后的渲染速度明显快不少，开启硬件加速就一定万事大吉么？第三幅图实际上就说明，如果你的这个 view 不断的失效的话，也会出现性能问题，第三图中可以看到蓝色的部曲线图有了一定的起色，这说明，displaylist 不断的失效并重现创建，如果你想了解的更加详细，可以查看这里



// Set the layer type to hardware

```
myView.setLayerType(View.LAYER_TYPE_HARDWARE, null);

// Setup the animation
ObjectAnimator animator = ObjectAnimator.ofFloat(myView,View.TRANSLATION_X, 150);

// Add a listener that does cleanup
animator.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationEnd(Animator animation) {
                myView.setLayerType(View.LAYER_TYPE_NONE, null);
   }
});
```

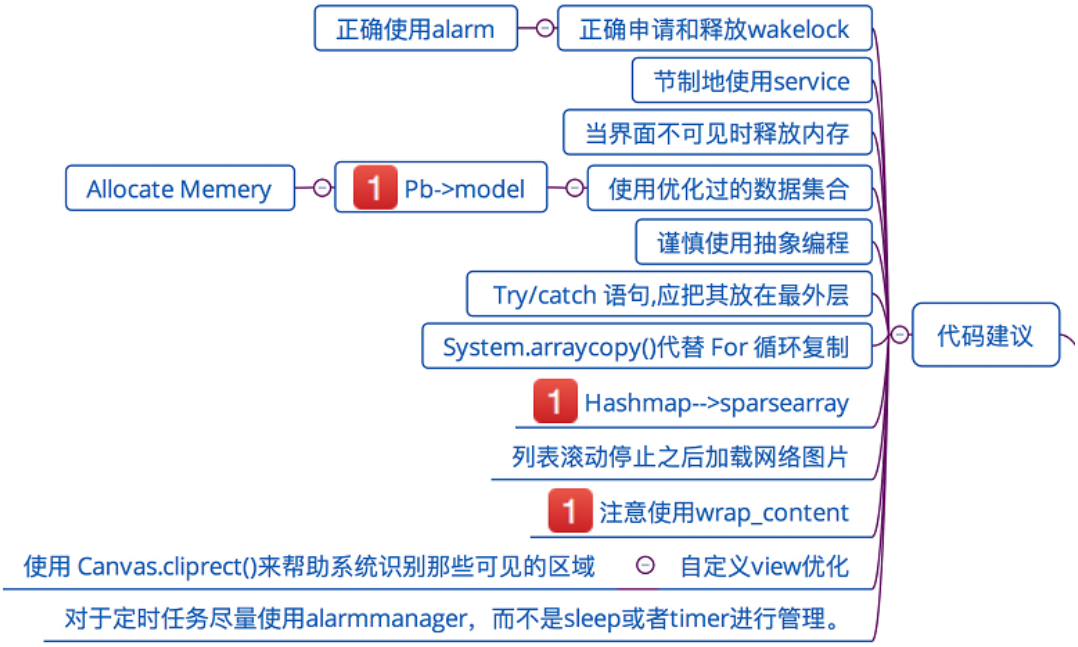
11 耗电优化

这里仅仅只是建议;

a、在定位精度要求不高的情况下，使用 wifi 或移动网络进行定位，没有必要开启 GPS 定位。

b、先验证网络的可用性，在发送网络请求，比如，当用户处于 2G 状态下，而此时的操作是查看一张大图，下载下来可能都 200 多 K 甚至更大，我们没必要去发送这个请求，让用户一直等待那个菊花吧。

四　接下来的一些内容就比较轻松了，是关于一些代码的建议



这里不一一细讲了，仅仅挑标记的部分说下。

pb->model 这里的优化就不在赘述，前面有讲如何优化。

然后建议使用 SparseArray 代替 HashMap,这里是 Google 建议的，因为 SparseArray 比 HashMap 更省内存，在某些条件下性能更好，主要是因为它避免了对 key 的自动装箱比如（int 转为 Integer 类型），它内部则是通过两个数组来进行数据存储的，一个存储 key，另外一个存储 value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间。

不到不得已，不要使用 wrap_content,,推荐使用 match_parent,或者固定尺寸，配合 gravity="center"，哈哈，你应该懂了的。

那么为什么说这样会比较好。

因为 在测量过程中，match_parent 和固定宽高度对应 EXACTLY ,而 wrap_content 对应 AT_MOST,这两者对比 AT_MOST 耗时较多。

五　总结

这是以上关于我在工作中遇到的性能问题的及处理的一些总结，性能优化设计的方方面面实

在是太多太多，本文不可能将全部的性能问题全部总结的清清楚楚，或许还多多少少存在一些纰漏之处，有不对的地方欢迎指出补充。

参考资料

http://developers.googleblog.cn/2017/09/constraintlayout.html

http://hukai.me/android-performance-patterns

https://juejin.im/entry/59396e01fe88c2006afc3862

https://github.com/JakeWharton/hugo

https://stackoverflow.com/questions/15509544/optimizing-gson-deserialization

https://medium.com/livefront/recyclerview-trick-selectively-bind-viewholders-with-payloads-4b28e3d2cce8

https://github.com/hehonghui/android-tech-frontier/blob/master/issue-30/%E9%80%9A%E8%BF%87%E7%A1%AC%E4%BB%B6%E5%B1%82%E6%8F%90%E9%AB%98Android%E5%8A%A8%E7%94%BB%E7%9A%84%E6%80%A7%E8%83%BD.md