**性能优化**

由于Android移动设备的内存和cpu都有一定的限制，所以Android应用程序不能无限制的使用内存和cpu资源，过多的使用内存会导致内存溢出，即OOM。过多的使用CPU资源，一般指在主线程做大量的耗时任务，会导致程序卡顿甚至无法响应的情况，即ANR。

* **布局优化**
* **布局选择和设置**

布局优化就是尽量减少布局容器的层级嵌套，这样会减少Android绘制页的工作量。在容器的选择上，按布局需求如果页面布局能够使用一个容器来完成最优的选择是，FrameLayout->LinearLayout->RelativeLayout，这是因为RelativeLayout的绘制内容的功能比较复杂，需要花费更多的cpu时间。如果需要使用两个LinearLayout来完成那就不如使用RelativeLayout来完成，因为容器的嵌套会降低程序的性能。

更复杂的布局，单单使用一层使用LinearLayout或RelativeLayout都解决不了的布局，推荐使用约束布局ConstraintLayout来减少层级嵌套。ConstraintLayout使用约束的方式指定子控件之间的位置和关系。ConstraintLayout中的子控件即可以实现RelativeLayout的效果，又可以实现LinearLayout布局的效果。

背景颜色设置：关于背景颜色，就是尽可能少的去设置背景颜色，能复用背景颜色就复用背景颜色。因为相同的背景颜色的绘制是完全没有必要的，并且会浪费大量的GPU资源，增加绘制时间。在项目中可以在AndroidManifest文件的<application/>标签下的theme属性指定的主题中设置android:windowBackground属性来指定app中Activity界面的默认背景颜色，这样在Activity的布局文件的根标签下就不用再指定默认颜色了。

* **使用特定标签**

**<include>标签**

这个标签的作用是将一个指定的布局文件引入到当前布局当中。如下：



图1

<include>标签只支持layout\_\*的属性，并且会覆盖所引入的文件的根布局中layout\_\*属性（其它属性不受影响），前提是必须覆写layout\_width和layout\_height这两个属性，否则覆写其它的layout\_\*属性效果将不会生效。

**<merge>标签**

与<include>标签一起使用，作用是去掉引入文件的根布局容器的嵌套，从而减少布局的层级嵌套。RelativeLayout和LinearLayout容器下都可以引入使用<merge>标签的布局文件，并且<merge>标签的布局文件中的控件可以根据引入它的容器设置具体的属性，见下：

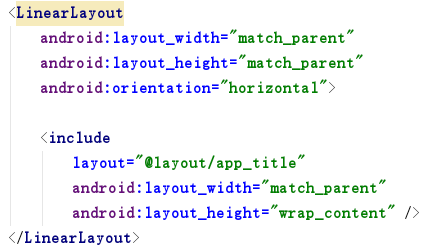


图2

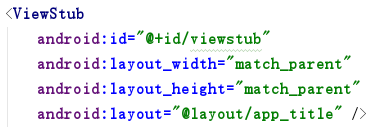
RelativeLayout引入：

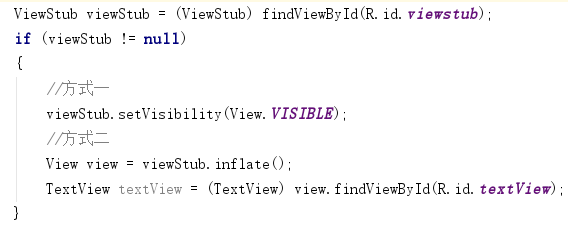
LinearLayout引入：

 1500885804(1)

**<ViewStub>标签**



<ViewStub>标签也是通过android:layout属性引入布局文件的，它的意义在于只有在使用的时候才会加载这个布局文件的内容。而View的隐藏和显示（GONE和VISIBLE）功能与它类似，但是性能远远不如<ViewStub>标签，因为它宽高都为0，它本身不参与任何布局解析，还有View的测量、布局、绘制过程，真正的做到了按需加载，提高了界面的初始化性能。



通过上图中的两种方式来加载并显示需要的布局，当加载后ViewStub就会被它引用的布局替换掉。假如需要使用<ViewStub>标签下的控件，则可以使用第二种方式来加载ViewStub的对象。另外我们加载出来后可以通过setVisibility()方法来控制它的显示和隐藏了。注意，代码中对viewStub对象做了为空的判断，那是因为<ViewStub>标签的id只能被使用一次，一旦<ViewStub>标签所指定的布局被加载一次后，它的id也就失效了，再次使用findViewById()则会返回空值。

另外<ViewStub>标签不是容器，他继承自Veiw，在这个标签下不能包含其它控件。它引用的布局文件下也不能使用<merge>标签。

* **绘制优化（需要研究Android渲染机制）**

参考：<https://www.jianshu.com/p/0d00cb85fdf3>

**Android系统的渲染机制？**

Android系统每16ms会发送一次VSYNC信号，触发对UI进行渲染。为什么是16ms?官方给出建议是,View绘制的帧率保证在60fps是最佳的，即每秒绘制60帧，这就要求每帧的绘制时间不超过16ms（16ms=1000/60）。然而在这16ms中主要做的工作包括：cpu将UI对象转换成一系列多边形和纹理。然后cpu将这些处理的数据传递到GPU，GPU进行绘制组件的最基础的操作。其中在GPU的的工作时最耗时的。

**界面显示卡顿的原因？**

卡顿的原因主要是因为绘制一帧的时间超过了16ms。即由正常的每秒绘制60帧，变成了每秒绘制少于60帧，也就是我们常说的丢帧现象。当其中一帧绘制的时间比较长时，这时候给用户的体验就是卡顿的现象。而出现丢帧现象的原因就是界面的过度绘制。还有一方面是内存的使用，如果进程中无用的对象比较多导致内存吃紧会导致频繁的GC，这时候也会引起界面的卡顿。

**什么是过度绘制？**

过得绘制是指同某个像素在同一帧时间内被绘制多次。

**过度绘制检测**

下载“开发助手”工具对安装的app进行检测。或者在logcat打印日志中会有drop frames类似的警告。

**引起过度绘制的原因？**

1、布局文件层级嵌套比较多。

2、重复绘制父布局和子控件背景颜色。

3、自定义控件中在onDraw方法中不要做耗时任务，也不要创建新的布局对象。

onDraw方法中不要创建新的布局对象，因为当onDraw方法被频繁调用的时候，会在一瞬间产生大量的临时对象，这不仅占用大量的内存还会导致系统频繁的gc，降低程序的执行效率。

onDraw方法中尽量不要做耗时的任务，比如循环操作，这样会造成View的绘制过程很不流畅。。

* **响应速度优化**

响应速度优化的核心思想是避免在UI线程中做耗时操作，如果有耗时操作那么就将耗时操作放入线程中去执行。

如果主线程做的耗时任务太多，主线程睡眠、堵塞等等，Android规定，Activity如果在5秒钟之内无法响应屏幕触摸事件或者键盘输入事件就会出现ANR；如果BroadcastReceive在10秒内还未执行完操作也会出现ANR。

在实际开发中遇到ANR后，系统会在/data/anr目录下创建一个文件traces.txt，通过分析这个文件就可以定位出ANR的原因。

* **内存优化**

参考：<https://juejin.im/post/5b1b5e29f265da6e01174b84>

内存优化主要分为：静态使用内存优化和动态使用内存优化。

* **静态使用内存优化**

静态使用内存是指app整个生命周期中必须使用的那部分内存。我们可以从以下几个方面入手进行优化：

1. SharePreference文件禁止存储大量的数据，可以将一些大的数据存储到数据库或文件中。因为app启动后会加载出SP文件中所有的数据并储存在缓存中。
2. 因为app在运行过程中首页是一直存在的，所以对首页的优化也是必须的，例如降低首页图片显示占用内存、动态加载的图片的ImageView控件设置的背景图要及时释放、播放完的动画要及时回收、弹出的dialog实例最好也进行回收等等，总之尽可能少的在首页缓存不必要的实例。
3. 谨慎的使用单例模式。

* **动态使用内存优化**

动态使用内存主要从以下三方面入手即可：

**1、内存泄露的优化**

内存优化的宗旨是避免无用的对象占用内存空间。其实虚拟机的垃圾回收机制几乎已经可以帮我们回收所有的无用对象了，但是还有一种无用对象是虚拟机不能回收掉的，这就是内存泄露的对象，出现内存泄露的对象主要是我们的代码逻辑出现了问题。内存泄露的问题可以查看java-虚拟机垃圾回收这一篇。

导致内存泄露的问题有：

1、集合类泄漏  
 2、单例/静态变量造成的内存泄漏  
 3、匿名内部类/非静态内部类  
 4、资源未关闭造成的内存泄漏

等等。

**2、图片列表显示的优化**

**3、图片加载以及显示的优化**

见Bitmap篇，高效加载图片。

* **应用启动速度优化**

1. 为启动的Activity设置特定的应用主题，并设置theme主题的windowBackground属性值为一个背景图片，这样当启动应用的时候会立刻显示这张图片。
2. 不要在自定义Appliaction类以及首次启动的Activity的onCreate方法中做过多的初始化工作、耗时的工作。

* **Apk包体优化**

首先解压apk文件可以看出，apk中体积最大的几部分：

**classes.dex：**包含被编译的java源码文件。

**resources.arsc：**包含已编译的res/values文件夹下的所有二进制文件，以及res/目录下values

文件夹以外的资源文件的路径信息，例如layout，图片等。

**res：**包含res目录下values文件夹意外的资源。

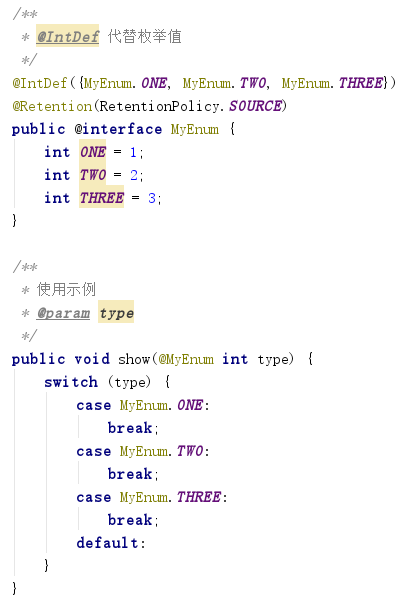
**assets：**存放资源文件，这些文件不会被编译成二进制文件。

* **calsses.dex优化**

首先，在反编译apk包后有可能会遇见多个calsses.dex文件，出现多个文件的原因是单个的classes.dex文件只能容纳64K的方法。超过这个限制，需要你在工程中启用multidexing，这样将会创建另一个calsses1.dex文件去存储剩下的方法。所以calsses.dex的文件数量时由你项目中方法数而定。

**1、避免使用枚举。**

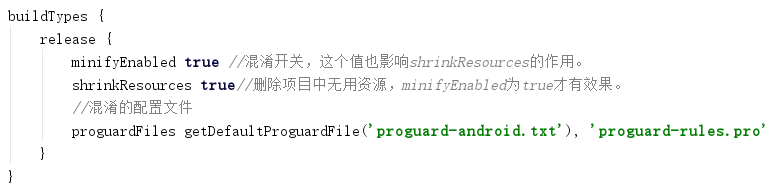
一个单独的枚举可以使classes.dex增加1.0到1.4KB的大小。可以使用@IntDef来代替枚举。如下：



**2、引用第三方库的优化**

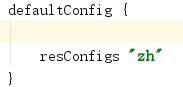
引用第三方库是避免不了的，优化引用的三方库主要是优化引用库的：代码、资源文件、引用库的国际化语言支持。

**代码、资源文件：**需要则build.gradle文件中进行配置



首先，shrinkResources会删除项目中或者lib中无用资源（如图片），但是不会删除value文件夹下无用资源。然后ProGuard规则的proguardFiles属性来移除lib中无用的类，其中proguard-rules.pro文件中配置了混淆规则，开启混淆后，默认情况下也会压缩项目中的无用代码。但是当三方引用库引用了其他的三方引用库，这种设置不能对引用库的引用库起到作用。

**引用库的国际化语言：**有时我们引用的三方库会支持很多不同的语言，但是我们只用到中文，我们可以通过以下方式解决：



* **资源文件优化（主要是图片的优化）**

1. 对于简单的图片，使用Drawable Xml文件代替。因为xml文件基本上在1kb左右。
2. 相似的图片复用，这里是指通过ImageView控件的tint（译为着色）、backgroundTint、backgroundTintMode等属性来改变图片的内容的颜色来复用图片。在5.0系统一下需要使用ColorFilter来调整。
3. 使用三方工具对png或者jpg格式的图片进行压缩，在减小图片体积的同时保持着感知的图片的质量。
4. 使用Android Studio将jpg、png、Gif格式的图片直接转换为webP格式。webP格式的图片同时拥有jpg和png格式图片的优点，体积小，提供透明度。

* **Bitmap优化**

**参考：<https://juejin.im/post/5af84f4b51882542714fdaa9>**

**<https://www.jianshu.com/p/3f6f6e4f1c88>**

图片占用的内存包括两部分，加载之前和加载之后。对于加载之前的图片可以进行压缩可以减少apk的大小（具体见apk包体优化）。这里只讨论加载Bitmap的优化。

我们可以从项目的资源目录、手机磁盘中、网络三个地方加载Bitmap，想要高效的加载图片可以从以下几个点入手：

1. 压缩加载图片的尺寸（包括从资源目录中加载）。
2. 设置像素的显示模式。
3. 压缩加载图片的质量。
4. 使用libjpeg.so库进行压缩。

* **线程优化**

线程优化主要是使用线程池，避免程序中创建大量的Thread。线程池通过内部的核心线程，可以避免因创建大量的线程的创建和销毁而造成的性能开销。线程池通过控制最大的线程并发数，避免大量的线程并发互抢系统资源而造成的堵塞现象。

* **其他优化**
* **避免创建多余的对象**

创建一个对象就意味着垃圾回收器需要回收一个对象，创建和回收是非常消耗时间的。所以我们需要遵守的基本原则：尽量少的创建临时对象，创建的临时对象越多GC的操作也就越多，这就意味着程序的性能和体验就越差。

例子：

1. 使用StringBuffer或者SringBuilder代替字符串拼接。
2. 使用基本数据类型代替封装数据类型，比如int比Integer要更高效。

* **常量使用static final修饰**

对于值不变的静态变量，我们可以添加上final修饰符让它变成一个静态常量。这样会提高编译器的编译效率。这种优化类型只对基本数据类型和String类型的常量有效。

* **使用系统封装好的API**

使用系统封装好的API，当系统提供的API不足以完成我们的需求时再去自己写。比如数组拷贝的功能，使用系统提供的System.arraycopy()方法要比我们自己写循环的效率高。

使用Android系统提供好的数据结构，SparseArray，SparseBooleanArray，以及LongSparseArray代替HashMap工具类等。

* **使用IntentService**

当使用Service时要注意只有在需要运行任务的时候再去启动一个Service，并且当任务执行完了之后也要及时的停止Service。当我们启动一个Service时，系统会尽量的保留这个Service所在的进程，这样就导致这个进程十分的消耗内存。为了更好的管理Service的生命周期，Android官方推荐使用IntentService，他不仅能够异步的执行不同的任务还能在执行完任务后及时的停止Service。

* **不要过多的使用枚举**

使用枚举占用的内存要比整形大。

* **尽量使用静态内部类**

这样可以避免潜在的由于内部类而导致的内存泄露。