|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类号 |  | |  | | | 密 级 | |  |
| U D C |  | |  | | | 编 号 | | 0003 |
|  | | | | | | | | |
| **读 书 笔 记** | | | | | | | | |
|  | | | | | | | | |
| Android应用性能优化最佳实践 | | | | | | | | |
|  | |  | |  |  | |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | | | | | | | | |
| By key.guan  2017.5~ | | | | | | | | |

目录

[1. 前言 5](#_Toc484470869)

[1.1. 为什么写这本书 5](#_Toc484470870)

[1.2. 本书面向的读者 5](#_Toc484470871)

[1.3. 本书特色 5](#_Toc484470872)

[1.4. 本书的主要内容 6](#_Toc484470873)

[1.5. 勘误和资源下载 7](#_Toc484470874)

[2. 绘制优化 8](#_Toc484470875)

[2.1. Android系统显示原理 8](#_Toc484470876)

[2.1.1. 绘制原理 9](#_Toc484470877)

[2.1.2. 刷新机制 12](#_Toc484470878)

[2.1.3. 卡顿的根本原因 16](#_Toc484470879)

[2.2. 性能分析工具 16](#_Toc484470880)

[2.2.1. 卡顿检测工具 17](#_Toc484470881)

[2.2.2. TraceView 20](#_Toc484470882)

[2.2.3. Systrace UI性能分析 23](#_Toc484470883)

[2.3. 布局优化 26](#_Toc484470884)

[2.3.1. 常用布局优化工具 27](#_Toc484470885)

[2.3.2. 布局优化方法 31](#_Toc484470886)

[2.4. 避免过度绘制 38](#_Toc484470887)

[2.4.1. 过度绘制检测工具 38](#_Toc484470888)

[2.4.2. 如何避免过度绘制 39](#_Toc484470889)

[2.5. 启动优化 41](#_Toc484470890)

[2.5.1. 应用启动流程 41](#_Toc484470891)

[2.5.2. 启动耗时监测 44](#_Toc484470892)

[2.5.3. 启动优化方案 46](#_Toc484470893)

[2.6. 合理的刷新机制 48](#_Toc484470894)

[2.6.1. 减少刷新次数 48](#_Toc484470895)

[2.6.2. 避免后台线程影响 48](#_Toc484470896)

[2.6.3. 缩小刷新区域 49](#_Toc484470897)

[2.7. 提升动画性能 49](#_Toc484470898)

[2.7.1. 帧动画 50](#_Toc484470899)

[2.7.2. 补间动画 50](#_Toc484470900)

[2.7.3. 属性动画 51](#_Toc484470901)

[2.7.4. 硬件加速 52](#_Toc484470902)

[2.8. 卡顿监控方案与实现 55](#_Toc484470903)

[2.8.1. 监控原理 55](#_Toc484470904)

[2.8.2. 代码实现 56](#_Toc484470905)

[2.9. 本章小结 59](#_Toc484470906)

[3. 内存优化 60](#_Toc484470907)

[3.1. Android内存管理机制 60](#_Toc484470908)

[3.1.1. Java对象生命周期 60](#_Toc484470909)

[3.1.2. 内存分配 62](#_Toc484470910)

[3.1.3. 内存回收机制 63](#_Toc484470911)

[3.1.4. GC类型 66](#_Toc484470912)

[3.2. 优化内存的意义 67](#_Toc484470913)

[3.3. 内存分析工具 70](#_Toc484470914)

[3.3.1. Memory Monitor 70](#_Toc484470915)

[3.3.2. Heap Viewer 72](#_Toc484470916)

[3.3.3. Allocation Tracker 74](#_Toc484470917)

[3.4. 避免内存泄漏 77](#_Toc484470918)

[3.4.1. 内存泄漏定义 77](#_Toc484470919)

[3.4.2. 使用MAT查找内存泄漏 78](#_Toc484470920)

[3.4.3. 常见内存泄漏场景 84](#_Toc484470921)

[3.4.4. 内存泄漏监控 86](#_Toc484470922)

[3.5. 优化内存空间 88](#_Toc484470923)

[3.5.1. 对象引用 89](#_Toc484470924)

[3.5.2. 减少不必要的内存开销 90](#_Toc484470925)

[3.5.3. 使用最优的数据类型 92](#_Toc484470926)

[3.5.4. 图片内存优化 97](#_Toc484470927)

[3.6. 图片管理模块设计与实现 101](#_Toc484470928)

[3.6.1. 实现异步加载功能 101](#_Toc484470929)

[3.6.2. 实现三重缓存 106](#_Toc484470930)

[3.6.3. 开源图片组件 110](#_Toc484470931)

[3.7. 本章小结 112](#_Toc484470932)

[4. 存储优化 113](#_Toc484470933)

[4.1. 存储方式 113](#_Toc484470934)

[4.1.1. SharedPreferences 113](#_Toc484470935)

[4.1.2. 文件存储 114](#_Toc484470936)

[4.1.3. SQLite（需要扩展） 114](#_Toc484470937)

[4.1.4. ContentProvider 115](#_Toc484470938)

[4.2. 序列化 115](#_Toc484470939)

[4.2.1. Serializable与Parcelable 115](#_Toc484470940)

[4.2.2. Gson实现JSON的序列化和反序列化 116](#_Toc484470941)

[4.2.3. Nano Proto Buffers 116](#_Toc484470942)

[4.2.4. FlatBuffers 116](#_Toc484470943)

[4.2.5. 小结 117](#_Toc484470944)

[4.3. SharedPreferences优化 117](#_Toc484470945)

[4.4. ？ 数据库使用及优化 118](#_Toc484470946)

[4.5. 本章小结 119](#_Toc484470947)

[5. 稳定性优化 120](#_Toc484470948)

[5.1. 提高代码质量 120](#_Toc484470949)

[5.1.1. 代码审查 120](#_Toc484470950)

[5.1.2. 代码静态扫描工具 123](#_Toc484470951)

[5.2. Crash监控 124](#_Toc484470952)

[5.2.1. Java层Crash监控 125](#_Toc484470953)

[5.2.2. ？Native层Crash监控：写NDK代码 128](#_Toc484470954)

[5.2.3. Crash上报机制 129](#_Toc484470955)

[5.3. ANR剖析 129](#_Toc484470956)

[5.3.1. ANR介绍 130](#_Toc484470957)

[5.3.2. ANR分析 130](#_Toc484470958)

[5.3.3. ANR监控 132](#_Toc484470959)

[5.4. 提高后台进程存活率 134](#_Toc484470960)

[5.4.1. 应用进程优先级 134](#_Toc484470961)

[5.4.2. 利用SyncAdapter提高进程优先级 135](#_Toc484470962)

[5.5. 本章小结 140](#_Toc484470963)

[6. 耗电优化 141](#_Toc484470964)

[6.1. 耗电检测工具 141](#_Toc484470965)

[6.1.1. Battery Historian使用步骤 141](#_Toc484470966)

[6.1.2. 报告参数解析 142](#_Toc484470967)

[6.2. 三大模块省电优化 143](#_Toc484470968)

[6.2.1. 显示 143](#_Toc484470969)

[6.2.2. 网络 144](#_Toc484470970)

[6.2.3. CPU 145](#_Toc484470971)

[6.3. 应用常用优化方案 148](#_Toc484470972)

[6.3.1. 计算优化 148](#_Toc484470973)

[6.3.2. 避免WakeLock使用不当 149](#_Toc484470974)

[6.3.3. 使用Job Scheduler 150](#_Toc484470975)

[6.4. Doze模式 154](#_Toc484470976)

[6.4.1. Doze模式介绍 154](#_Toc484470977)

[6.4.2. Doze模式应用策略 155](#_Toc484470978)

[6.4.3. 测试Doze模式应用工作状态 156](#_Toc484470979)

[6.5. 本章小结 156](#_Toc484470980)

[7. 安装包大小优化 157](#_Toc484470981)

[7.1. 应用装包的构成 157](#_Toc484470982)

[7.2. 减少安装包大小的常用方案 159](#_Toc484470983)

[7.2.1. 代码混淆 160](#_Toc484470984)

[7.2.2. 资源优化 162](#_Toc484470985)

[7.2.3. 其他优化 164](#_Toc484470986)

[7.3. 本章小结 165](#_Toc484470987)

[结束语 165](#_Toc484470988)

[QA 166](#_Toc484470989)

[启发 166](#_Toc484470990)

[task 167](#_Toc484470991)

# 前言

## 为什么写这本书

一个好的应用，除了要有吸引人的功能和交互之外，在性能上也应该有高的要求，即使应用非常具有特色，或者功能和业务具有唯一性，在产品前期可能吸引了部分用户，但用户体验不好的话，也会给产品带来很差的口碑，如果有在体验上更好的竞品，用户也会很快转移。那么一个好的应用应该如何定义呢？主要有三方面：

* 业务/功能
* 符合逻辑的交互
* 优秀的性能

众所周知，Android系统作为以移动设备为主的一款操作系统，硬件配置有一定的限制，虽然配置现在越来越高级，但仍然无法和PC相比，在CPU和内存上的使用不合理或者耗费资源多时，就会碰到内存不足导致的稳定性问题、CPU消耗太多导致的卡顿问题等。例如，我们发布一款产品后会收到很多的反馈，这些反馈来自很多渠道，有用户反馈，有应用发布平台的反馈通道等。

面对这些问题时，大家想到的都是联系用户，然后看日志，特别是有关性能类问题的反馈，原因也非常难找，日志大多用处不大，为什么呢？因为性能问题大部分是非必现的问题，定位时很难复现，而又没有关键的日志，当然就无法找到原因了。这些问题非常影响用户的体验和功能的使用，所以解决这些问题是非常重要的。当前市场上讲解性能优化的书太少，即使有些书讲到，很多也是一笔带过，没有深入分析和寻找解决方案，所以有必要用一本书来从多个维度讲解在性能上我们面临了什么问题，如何解决这些问题，并在实际的项目中来优化我们的应用，以提高用户体验。

## 本书面向的读者

本书适合所有Android应用开发从业人员及在校学生，特别是有一定Android应用开发经验的开发人员，高级开发人员也可以通过本书了解更多的性能调优知识。

## 本书特色

本书为进阶类图书，对于一些基础技术和基础理论知识不会做过多的阐述，特别是入门类的知识点，大家可以从其他书籍获取相关的知识。书中以性能优化为核心，深入剖析性能优化具体涉及的技术背景与优化方案，同时提供典型案例，帮助读者更深入地掌握Android应用开发技术，理解Android的运行机制和原理，掌握Android性能优化的思想，让开发者快速成长，打造高质量的Android应用。

## 本书的主要内容

可以把用户能体验到的性能问题主要总结为4个类别：

* 流畅
* 稳定
* 省电
* 省流量

性能问题的主要原因是什么，原因有相同的，也有不同的，但归根结底，不外乎内存使用、代码效率、合适的策略逻辑、代码质量这一类问题。本书讲解内容的目标和方向如下图所示。



从上图可以看到，打造一个高质量的应用应该以4个方向为目标：快、稳、省、小。

* 快：使用时避免出现卡顿，响应速度快，减少用户的等待时间，满足用户预期。
* 稳：降低crash率和ANR率，不要在用户使用过程中崩溃和无响应。
* 省：节省流量和耗电，减小用户使用成本，避免使用时导致手机发烫。
* 小：安装包小可以降低用户的安装成本。

这4类问题需要从根源上解决，也就是要解决图中第二个框里的问题：卡顿、内存使用不合理、代码质量差、代码逻辑不优秀、安装包过大。这些问题也是在开发过程中碰到最多的问题，在实现业务需求的同时，也需要考虑到这些点，多花时间去思考，避免功能完成后再来做优化和修复Bug，这个时候带来的成本会增加。如果是维护之前的代码，就需要使用一系列工具来发现问题点。

性能优化不是更新一两个版本可以解决的，是持续性的需求，结合到实际中，在一个新产品/项目开始时，由于人力和上线时间的限制，可以把优先级放低，但有些点是在写代码时就要考虑的，这就体现出程序员的技术功底。

本书强调性能调优的核心思想和方向如下：

发现问题→分析问题原因及背景→寻找最优解决方案→解决问题。

本书一共7章，在简单介绍了Android Studio的使用指南后，分别从绘制（UI）、内存、存储、稳定性、耗电以及安装包6个方面进行优化，从系统上深入分析绘制和内存的原理，一步步深入了解导致性能问题的本质原因，同时讲述了多种性能优化工具的使用，通过分析典型案例，得到有效的优化方案，从而实现更高质量的应用。书中所讲述的内容均基于Android 6.0系统。

## 勘误和资源下载

本书代码的下载地址：<https://github.com/lyc7898/AndroidTech>。

Android Studio的使用不再这里展开了。

# 绘制优化

Android应用启动慢，使用时经常卡顿，是非常影响用户体验的，应该尽量避免出现。卡顿的场景有很多，按场景可以分成4类：UI绘制、应用启动、页面跳转、事件响应，如图2-1所示。在这四种场景下又有多个小分类，基本上覆盖了卡顿的各个场景。



图2-1　卡顿主要场景

这4种卡顿场景的根本原因又可以分成两大类。

界面绘制：主要原因是绘制的层级深、页面复杂、刷新不合理，由于这些原因导致卡顿的场景更多出现在UI和启动后的初始界面以及跳转到页面的绘制上。

数据处理：导致这种卡顿场景的原因是数据处理量太大，一般分为三种情况，一是数据处理在UI线程（这种应该避免），二是数据处理占用CPU高，导致主线程拿不到时间片，三是内存增加导致GC频繁，从而引起卡顿。

本章主要通过优化UI界面编程来减少卡顿，以场景为纬度，通过工具深入分析症结所在，找到导致问题的根本原因，利用涉及的相关技术背景，以及了解当前业内主流解决方案，然后结合实例来找到最终的优化方案，使应用流畅。

引起卡顿的原因有很多，但不管怎么样的原因和场景，最终都是通过设备屏幕上的显示来到达用户，归根到底就是显示有问题，所以，要解决卡顿，就要先了解Android系统的显示原理。

## Android系统显示原理

说到显示原理，相信大家从网上或其他书籍上看过相关的知识，但大部分人看得云里雾里，是因为整个显示系统很复杂吗？确实很复杂，但我们只需要了解整体流程，抓住关键知识，从应用角度上来讲，需要掌握的不多，如果自己有兴趣，可以阅读专门介绍系统框架的书籍，结合源码来分析，这里就不过多地介绍系统层的知识了。下面我们首先介绍在应用开发上需要涉及的知识点和整体流程。

Android的显示过程可以简单概括为：Android应用程序把经过测量、布局、绘制后的surface缓存数据，通过SurfaceFlinger把数据渲染到显示屏幕上，通过Android的刷新机制来刷新数据。也就是说应用层负责绘制，**系统层负责渲染**，**通过进程间通信把应用层需要绘制的数据传递到系统层服务，系统层服务通过刷新机制把数据更新到屏幕。**

通过阅读Android系统的源码可以了解显示的流程，Android的图形显示系统采用的是Client/Server架构。SurfaceFlinger（Server）由C++代码编写。Client端代码分为两部分，一部分是由Java提供给应用使用的API，另一部分则是由C++写成的底层具体实现。下面通过介绍绘制原理和刷新机制来学习整个显示过程。

### 绘制原理

绘制任务是由应用发起的，最终通过系统层绘制到硬件屏幕上，也就是说，应用进程绘制好后，通过跨进程通信机制把需要显示的数据传到系统层，由系统层中的SurfaceFlinger服务绘制到屏幕上。那么应用层和系统层中的流程是什么样的呢？接下来将进行具体介绍。

#### 应用层

先来看一个UI界面的典型构成框架，也可以是一个Activity的构成。如图2-2所示，有很多不同层次的基本元素——View，整体是一个树型结构，有不同的嵌套，存在着父子关系，子View在父View中，这些View都经过一个相同的流程最终显示到屏幕上，这也意味着要完整地显示所有数据，就要对其中的View都进行一次绘制工作，并且针对每个View的操作都是一个递归过程。

在Android的每个View绘制中有三个核心步骤（见图2-3），通过Measure和Layout来确定当前需要绘制的View所在的大小和位置，通过绘制（Draw）到surface，在Android系统中整体的绘图源码是在ViewRootImp类的performTraversals（）方法，通过这个方法可以看出Measure和Layout都是递归来获取View的大小和位置，并且以深度作为优先级。可以看出，层级越深，元素越多，耗时也就越长。



图2-2　页面构成框架



图2-3　View绘制流程

（1）Measure

用**深度优先**原则递归得到所有视图（View）的宽、高；获取当前View的正确宽度**childWidthMeasureSpec**和高度childHeightMeasureSpec之后，可以调用它的成员函数Measure来设置它的大小。如果当前正在测量的子视图child是一个视图容器，那么它又会重复执行操作，直到它的所有子孙视图的大小都测量完成为止。

（2）Layout

用深度优先原则递归得到所有视图（View）的位置；当一个子View在应用程序窗口左上角的位置确定之后，再结合它在前面测量过程中确定的宽度和高度，就可以**完全确定它在应用程序窗口中的布局。**

（3）Draw

目前Android支持了两种绘制方式：软件绘制（CPU）和硬件加速（GPU），其中硬件加速在Android 3.0开始已经全面支持，很明显，硬件加速在UI的显示和绘制的效率远远高于CPU绘制，但硬件加速并非如大家所想的那么完善，它也存在明显的缺点：

·耗电：GPU的功耗比CPU高。

·兼容问题：某些接口和函数不支持硬件加速。

·内存大：使用OpenGL的接口至少需要8MB内存。

所以是否使用硬件加速，需要考虑一些接口是否支持硬件加速，同时结合产品的形态和平台，比如TV版本就不需要考虑功耗的问题，而且TV屏幕大，使用硬件加速容易实现更好的显示效果。

#### 系统层

真正把需要显示的数据渲染到屏幕上，是通过系统级进程中的SurfaceFlinger服务来实现的，SurfaceFlinger的具体实现和工作原理因为和应用层关系不大，所以这里不做过多介绍，只需要了解它主要是做些什么工作。

* ·响应客户端事件，创建Layer与客户端的Surface建立连接。
* ·接收客户端数据及属性，修改Layer属性，如尺寸、颜色、透明度等。
* ·将创建的Layer内容刷新到屏幕上。
* ·维持Layer的序列，并对Layer最终输出做出裁剪计算。

既然是两个不同进程，那么肯定需要一个跨进程的通信机制来实现数据传输，在Android的显示系统，使用了Android的匿名共享内存：SharedClient，每一个应用和SurfaceFlinger之间都会创建一个SharedClient，如图2-4所示。从图2-4中可以看出，在每个SharedClient中，最多可以创建31个SharedBufferStack，每个Surface都对应一个SharedBufferStack，也就是一个window。

一个SharedClient对应一个Android应用程序，而一个Android应用程序可能包含多个窗口，即Surface。也就是说SharedClient包含的是SharedBufferStack的集合。因为最多可以创建31个SharedBufferStack，这也意味着一个Android应用程序最多可以包含31个窗口，同时每个SharedBufferStack中又包含了两个（低于4.1版本）或者三个（4.1及以上版本）缓冲区，即在后面的显示刷新机制中会提到的双缓冲和三重缓冲技术。



图2-4　Android显示框架

最后总结起来显示整体流程分为三个模块：应用层绘制到缓存区，SurfaceFlinger把缓存区数据渲染到屏幕，由于是两个不同的进程，所以使用Android的匿名共享内存SharedClient缓存需要显示的数据来达到目的。

SurfaceFlinger把缓存区数据渲染到屏幕（流程如图2-5所示），主要是驱动层的事情，这里不做太多解释。

从图2-5中可以看出，绘制过程首先是CPU准备数据，通过Driver层把数据交给CPU渲染，其中CPU主要负责Measure、Layout、Record、Execute的数据计算工作，GPU负责Rasterization（栅格化）、渲染。由于图形API不允许CPU直接与GPU通信，而是通过中间的一个图形驱动层（Graphics Driver）来连接这两部分。图形驱动维护了一个队列，CPU把display list添加到队列中，GPU从这个队列取出数据进行绘制，最终才在显示屏上显示出来。



图2-5　渲染数据流程图

知道了绘制的原理后，那么到底绘制一个单元多长时间才是合理的，首先需要了解一个名词：FPS。FPS（Frames Per Second）表示每秒传递的帧数。在理想情况下，**60 FPS就感觉不到卡**，这意味着每个绘制时长应该在**16ms**以内，如图2-6所示。



图2-6　理想状态下的绘制操作

但是Android系统很有可能无法及时完成那些复杂的界面渲染操作。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需的60FPS。即为了实现60FPS，就意味着程序的大多数绘制操作都必须在16ms内完成。

如果某个操作花费的时间是24ms，系统在得到VSYNC信号时就无法进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面。主要场景在执行动画或者滑动ListView时更容易感知到卡顿不流畅，是因为这里的操作相对复杂，容易发生丢帧的现象，从而感觉卡顿。有很多原因可以导致CPU或者GPU负载过重从而出现丢帧现象：可能是你的Layout太过复杂，无法在16ms内完成渲染；可能是UI上有层叠太多的绘制单元；还有可能是动画执行的次数过多。

最终的数据是刷新机制通过系统去刷新数据，刷新不及时也是引起卡顿的一个主要原因。接下来将详细介绍系统是怎么刷新的以及在什么情况下会导致卡顿发生。

### 刷新机制

Google发布Android操作系统后，Android OS系统一直在不断优化、更新。但直到Android 4.0版本发布，有关UI显示不流畅的问题仍未得到根本解决。在整个Android版本升级过程中，Android在显示系统方面做了不少优化和改进，比如支持硬件加速等技术，但本质原因似乎和硬件关系并不大，也没有得到太多改善。而与高端硬件配置的Android机器价格相近的iPhone，其UI的流畅性强却是有目共睹的。

从Android 4.1（Jelly Bean）开始，Android OS开发团队便力图在每个版本中解决一个重要问题。作为严重影响Android口碑问题之一的UI流畅性差的问题，首先在Android 4.1版本中得到了有效处理。其解决方法即在4.1版本推出的Project **Butter**。Project Butter对Android Display系统进行了重构，引入三个核心元素：**VSYNC、Triple Buffer和Choreographer**。其中，VSYNC是理解Project Buffer的核心。VSYNC是Vertical Synchronization（垂直同步）的缩写，是一种在PC上已经**很早就广泛使用的技术**，读者可简单地把它认为是一种定时中断。Choreographer起调度的作用，将绘制工作统一到VSYNC的某个时间点上，使应用的绘制工作有序。接下来，本文将围绕VSYNC来介绍Android Display系统的工作方式。

在讲解刷新机制之前，先介绍几个名词以及VSYNC和Choreographer主要功能及工作方式。

·双缓冲：显示内容的数据内存，为什么要用双缓冲，我们知道在Linux上通常使用Framebuffer来做显示输出，当用户进程更新Framebuffer中的数据后，显示驱动会把Framebuffer中每个像素点的值更新到屏幕，但这样会带来一个问题，如果上一帧的数据还没有显示完，Framebuffer中的数据又更新了，就会带来**残影的问题，给用户直观的感觉就会有闪烁感**，所以普遍采用了双缓冲技术。双缓冲意味着要使用两个缓冲区（在SharedBufferStack中），其中一个称为Front Buffer，另外一个称为Back Buffer。**UI总是先在Back Buffer中绘制**，后台绘制好，然后再和Front Buffer交换，渲染到显示设备中。即只有当另一个buffer的数据准备好后，通过io\_ctrl来通知显示设备切换Buffer。

·VSYNC：**从前面的双缓冲介绍中可以了解到，只有当另一个buffer准备好后，才能通知刷新，这就需要CPU以主动查询的方式来保证数据是否准备好，因为这种机制效率很低**，**所以引入了VSYNC**。VSYNC是Vertical Synchronization（垂直同步）的缩写，可以简单地把它认为是一种定时中断，一旦收到VSYNC中断，CPU就开始处理各帧数据。

Choreographer：收到VSYNC信号时，调用用户设置的回调函数。一共有以下三种类型的回调：

·CALLBACK\_INPUT：优先级最高，与输入事件有关。

·CALLBACK\_ANIMATION：第二优先级，与动画有关。

·CALLBACK\_TRAVERSAL：最低优先级，与UI控件绘制有关。

接下来通过时序图来分析刷新的过程，这些时序图是Google在2012 Google I/O讲解新的显示系统提供的，图2-7所示的时序图有三个元素：Display（显示设备），CPU-CPU准备数据，GPU-GPU准备数据。最下面的时间为显示时间，根据理想的60FPS，以16ms为一个显示周期。



图2-7　没有VSync信息的刷新

（1）没有VSync信号同步

我们以16ms为单位来进行分析：

1）从第一个16ms开始看，Display显示第0帧，CPU处理完第一帧后，GPU紧接其后处理继续第一帧。三者都在正常工作。

2）时间进入第二个16ms：因为在上一个16ms时间内，第1帧已经由CPU、GPU处理完毕。所以Display可以正常显示第1帧。显示没有问题，但在本16ms期间，CPU和GPU并未及时绘制第2帧数据（前面的空白区在忙别事情去了），而是在本周期快结束时，CPU/GPU才去处理第2帧数据。

3）时间进入第3个16ms，此时Display应该显示第2帧数据，但由于CPU和GPU还没有处理完第2帧数据，故Display只能继续显示第一帧的数据，结果使得第1帧多画了一次（对应时间段上标注了一个Jank），这就导致错过了显示第二帧。

通过上述分析可知，在第二个16ms时，发生Jank的关键问题在于，为何在第1个16ms段内，CPU/GPU没有及时处理第2帧数据？从第二个16ms开始有一段空白的时间，可以说明原因所在，**那就是CPU可能是在忙别的事情，不知道该到处理UI绘制的时间了**。可CPU一旦想起来要去处理第2帧数据，时间又错过了。为解决这个问题，4.1版本推出了Project Butter，核心目的就是解决刷新不同步的问题。

（2）有VSync信号同步

加入VSync后，从图2-8可以看到，一旦收到VSync中断，CPU就开始处理各帧的数据。大部分的Android显示设备刷新率是60Hz（图2-7的时间轴也是60ms），这也就意味着每一帧最多只能有1/60=16ms左右的准备时间。假如CPU/GPU的FPS高于这个值，显示效果将更好。但是，这时又出现了一个新问题：CPU和GPU处理数据的速度都能在16ms内完成，而且还有时间空余，但必须等到VSYNC信号到来后，才处理下一帧数据，因此CPU/GPU的FPS被拉低到与Display的FPS相同。

从图2-9采用双缓冲区的显示效果来看：在双缓冲下，CPU/GPU FPS大于刷新频率同时采用了双缓冲技术以及VSync，可以看到整个过程还是相当不错的，虽然CPU/GPU处理所用的时间时短时长，但总体来说都在16ms以内，因而不影响显示效果。A和B分别代表两个缓冲区，它们不断交换来正确显示画面。但如果CPU/GPU的FPS小于Display的FPS，情况又不同了，如图2-10所示。



图2-8　有VSync的绘制



图2-9　双缓冲下的时序图（CPU的第三个A->B）



图2-10　双缓冲下CPU/GPU FPS小于刷新频率时序图

从图2-10可以看到，当CPU/GPU的处理时间超过16ms时，第一个VSync就已经到来，但缓冲区B中的数据却还没有准备好，这样就只能继续显示之前A缓冲区中的内容。而后面B完成后，又因为还没有VSync信号，CPU/GPU这个时候只能等待下一个VSync的来临才开始处理下一帧数据。因此在整个过程中，有一大段时间被浪费。总结这段话就是：

1）在第二个16ms时间段内，Display本应显示B帧，但因为GPU还在处理B帧，导致A帧被重复显示。

2）同理，在第二个16ms时间段内，CPU无所事事，因为A Buffer由Display在使用。B Buffer由GPU使用。注意，一旦过了VSYNC时间点，CPU就不能被触发以及处理绘制工作了。

为什么CPU不能在第二个16ms处即VSync到来就开始工作呢？很明显，原因就是只有两个Buffer。如果有第三个Buffer存在，CPU就可以开始工作，而不至于空闲。于是在Andoird 4.1以后，引出了第三个缓冲区：Triple Buffer。Triple Buffer利用CPU/GPU的空闲等待时间提前准备好数据，并不一定会使用。

[注意　在大部分情况下，只使用到双缓存，只有在需要时，才会用三缓冲来增强，这时可以把输入的延迟降到最少，保持画面的流畅。

引入Triple Buffer后的刷新时序如图2-11所示。



图2-11　使用Triple Buffer时序图

在第二个16ms时间段，CPU使用C Buffer绘图。虽然还是会多显示一次A帧，但后续显示就比较顺畅了。是不是Buffer越多越好呢？回答是否定的。由图2-11可知，在第二个时间段内，CPU绘制的第C帧数据要到第四个16ms才能显示，这比双缓存情况多了16ms延迟。所以缓冲区不是越多越好，要做到平衡到最佳效果。

从以上的分析来看，Android系统在显示机制上解决了Android UI不流畅的问题，并且从Google I/O2012给出的视频来看，其效果也达到预期。但实际在应用开发过程中仍然存在卡顿的现象。因为VSync中断处理的线程优先级一定要最高，否则即使接收到VSync中断，不能及时处理，也是徒劳无功。

### 卡顿的根本原因

那卡顿的根本原因是什么呢，从Android系统的显示原理中可以看到，影响绘制的根本原因有以下两方面：

* ·绘制任务太重，绘制一帧内容耗时太长。
* ·主线程太忙了，导致VSync信号来时还没有准备好数据导致丢帧。

耗时太长，需要从UI布局和绘制上来具体分析，详见后面内容。这里我们主要讨论下第二个方面。我们知道所有的绘制工作都是由主线程，也就是UI线程来负责，主线程的关键职责是处理用户交互，在屏幕上绘制像素，并进行加载显示相关的数据。在Android应用开发中，特别需要避免任何阻碍主线程的事情，这样应用程序才能保持对用户操作的即时响应。

在实际的开发过程中，我们需要知道主线程应该做什么，总结起来主线程主要做以下几个方面的工作：

* ·UI生命周期控制
* ·系统事件处理
* ·消息处理
* ·界面布局
* ·界面绘制
* ·界面刷新

除了这些以外，尽量避免将其他处理放到主线程中，特别是复杂的数据计算和网络请求。

## 性能分析工具

从前一节可以看到，Android系统在4.1以后从框架上解决了由于系统问题导致的卡顿现象，但在实际的使用过程中，在用户的感受上，卡顿仍然是应用开发中主要面临的问题，而原因从上一节的分析中也知道本质是VSync信号到来时，不能及时处理绘制事件导致，本节先抛出以下两个问题：

1）应用层做了什么会导致VSync事件不能及时处理？

2）卡顿能监控吗？

性能问题并不容易复现，也不好定位，光从几个场景不能完全覆盖所有的问题，因此在做性能优化时，最直接有效的方法，就是尽量复现存在性能问题的场景，并监控此过程中程序的执行流程，如果能够方便地分析程序中函数的调用关系和执行时间，自然也就很容易找出性能瓶颈。

分析问题和确认问题是否解决，都借助了相应的调试工具，比如查看Layout层次的次的Hierarchy View、Android系统上带的GPU Profile工具和静态代码检查工具Lint等。这些工具对性能优化都起到非常重要的作用。本节将介绍这些工具和另外两个性能优化非常重要的工具：**TraceView**和**Systrace**。这两个工具除了在UI上，对于在后面将要讲到的启动优化、动画优化等上都是很重要的工具，可以说大部分的性能分析都离不开这几个工具，接下来学习几个常用的与流畅度优化相关的工具的使用方法，在后面实际的优化方案中也会介绍其他辅助工具。

### 卡顿检测工具

要做性能优化，就非常有必要借助于一系列辅助工具，Android提供了多个开发辅助工具，在性能调优过程中非常重要，下面介绍几个常用的工具，在后面的具体优化过程中会多次使用到。

我们已经知道，从应用层绘制一个页面（View），主要有三个过程：CPU准备数据→GPU从数据缓存列表获取数据→Display设备绘制，这三个过程的耗时可以通过一个手机开发辅助工具查看：Profile GPU Rendering。Profile GPU Rendering是Android 4.1系统开始提供的一个开发辅助功能，在设置中打开开发者选项，如图2-12所示。

Profile GPU Rendering功能特点如下：

* + - ·它是一个图形监测工具，能实时反应当前绘制的耗时。
    - ·横轴表示时间，纵轴表示每一帧的耗时（单位为ms）。
    - 随着时间推移，从左到右的刷新呈现。
    - ·提供了一个标准的耗时，如果高于标准耗时，表示当前这一帧丢失。

注意：如果设置中没有开发者选项，可以通过设置页面中的“关于”选项，单击版本号七次即可打开开发者选项，在后面的章节中还会使用到开发者选项中的其他辅助工具。

打开Profile GPU Rendering后可以看到实时刷新的彩色图，如图2-13所示。每一根竖线表示一帧，由多个颜色组成，不同颜色的解释如下：



图2-12　打开Profile GPU rendering



图2-13　Profile GPU rendering

每一条柱状图都由4种颜色组成：红、黄、蓝、紫，这些线对应每一帧在不同阶段的实际耗时。

·蓝色代表测量绘制的时间，它代表需要多长时间去创建和更新DisplayList。在Android中，一个视图在进行渲染之前，它必须被转换成GPU熟悉的格式，简单来说就是几条绘图命令，蓝色就是记录了在屏幕上更新视图需要花费的时间，也可以理解为执行每一个View的onDraw方法，创建或者更新每一个View的Display List对象。在蓝色的线很高时，有可能是因为需要重新绘制，或者自定义视图的onDraw函数处理事情太多。

红色代表执行的时间，这部分是Android进行2D渲染Display List的时间，为了绘制到屏幕上，Android需要使用OpenGl ES的API接口来绘制Display List，这些API有效地将数据发送到GPU，最终在屏幕上显示出来。当红色的线非常高时，可能是由重新提交了视图而导致的。

橙色部分表示处理时间，或者是CPU告诉GPU渲染一帧的地方，这是一个阻塞调用，因为CPU会一直等待GPU发出接到命令的回复，如果柱状图很高，就意味着GPU太繁忙了。

紫色段表示将资源转移到渲染线程的时间，只有Android 4.0及以上版本才会提供。

任何时候超过绿线（警戒线，对应时长16ms），就有可能丢失一帧的内容，虽然对于大部分应用来说，丢失几帧确实感觉不出卡顿，但保持UI流畅的关键就在于让这些垂直的柱状条尽可能地保持在绿线下面。

GPU Profile工具能够很好地帮助你找到渲染相关的问题，但是要修复这些问题就不是那么简单了。需要结合另一个耗时工具和代码来具体分析，找到性能的瓶颈，并进行优化。在GPU Profile Render发现有问题的页面后，可以通过另外一个工具Hierarchy Viewer来查看页面的布局层次和每个View所花的时间，在后面的布局优化章节，将通过实例来讲解和学习使用方法。

综上，一般关注蓝色、红色和绿线就好。

[图片]技巧　在实际开发中，从图上虽然可以看到绘制的时间，但对不便于进行数据分析，比如进入某一个页面，柱形图虽然实时绘制出来，但不能更好地分析，这里可以通过：adb shell dumpsys gfxinfo com.\*\*.\*\*（包名）把具体的耗时输出到日志中来分析。

|  |
| --- |
| …  Total frames rendered: 1700  Janky frames: 555 (32.65%)  90th percentile: 26ms  95th percentile: 40ms  99th percentile: 81ms  Number Missed Vsync: 86  Number High input latency: 1  Number Slow UI thread: 201  Number Slow bitmap uploads: 18  Number Slow issue draw commands: 372  … |

### TraceView

TraceView是AndroidSDK自带的工具，用来分析函数调用过程，可以对Android的应用程序以及Framework层的代码进行性能分析。它是一个图形化的工具，最终会产生一个图表，用于对性能分析进行说明，可以分析到应用具体每一个方法的执行时间，使用可以非常直观简单，分析性能问题很方便。

#### 使用方法

在使用TraceVeiw分析问题之前需要得到一个\*.trace的文件，然后通过TraceView来分析trace文件的信息，trace文件的获取有两种方式：

1. 在DDMS中使用
2. 连接设备。
3. 打开应用。
4. 打开DDMS（若在Android Studio中则先打开Android Device Monitor）。
5. 单击Strart Method Profiling按钮，如图2-14所示。



1. 在应用中操作需要监控的点，比如进入一个Activity或者滑动一个列表，完成后单击Stop Method Profiling按钮，如图2-15所示。



6）结束会自动跳转到TraceView视图。

这种方法使用方便，但监控范围不够精确，如果需要精确监控某一个路径，就需要使用下一个方法：在代码中加入调试语句保存Trace文件。

1. **代码中加入调试语句保存trace文件**

有时在开发过程中不好复现的问题，需要在关键的路径上获取TraceView数据，在测试时复现此问题后直接拿到Trace文件查看对应的数据。这时可以在代码中使用TraceView工具并生成对应的trace文件。在android.os.Debug类中提供了相应的方法，过程很简单步骤如下：

1）在需要开始监控的地方调用startMethodTracing（）。

2）在需要结束监控的地方调用stopMethodTracing（）。

3）系统会在SD卡中创建<trace-name>.trace文件。

4）使用traceveiw打开该文件进行分析。

调用代码如下：



在应用程序中调用startMethodTracing（）时，系统会在指定的路径上创建一个名为<trace\_filename>.trace文件。这个文件包含了方法名跟踪数据，以及与线程和方法名的映射表。然后系统开始缓存应用产生的跟踪数据，直到应用程序调用stopMethodTracing（）结束，此时将其缓冲的数据写入输出文件中。如果系统达到最大缓存大小时，还没有调用stopMethodTracing（），系统会停止跟踪并发送一个通知。

在Android 4.4及更高版本中，可以通过基于采样的方法分析耗时情况，因为减少了分析Trace文件的次数，降低了IO读写，所以可以减少Trace工具在运行时对性能的影响，同时对分析结果也不会有很大的偏差。通过调用startMethodTracing（）方法，就可以指定具体的采样间隔，定期采集样本数据分析。

注意　在代码中使用此方法保存TraceView数据，不要忘记在应用中打开write to external storage权限（WRITE\_EXTERNAL\_STORAGE）。

#### TraceView视图说明

Traceview视图分两部分，上半部分为时间片面板（Timeline Panel），下半部分为分析面板（Profile Panel）。时间片面板如图2-16所示。



图2-16　时间片面板

X轴表示时间消耗，单位为毫秒（ms），Y轴表示各个线程，每个线程中的不同方法使用了不同的颜色来表示，颜色占用面积越宽，表示该方法占用CPU时间越长。

·时间片面板可以放大/缩小，也可以指定区域放到最大，方便查看具体的过程，一般优先选择放大耗时严重的区域。

分析面板（Profile Panel）如图2-17所示。



分析面板看起来并不复杂，但需要理解各列数据的意义，每一列表示的意义如表2-1所示。



表2-1　分析面板参数意义

使用TraceView查看耗时，主要关注**Calls+Recur Calls/Total**和**Cpu Time/Call**这两个值，也就是关注调用次数多和耗时久的方法，然后优化这些方法的逻辑和调用次数，减少耗时。

[图片]注意　RealTime与cputime区别为：因为RealTime包括了CPU的上下文切换、阻塞、GC等，所以RealTime方法的实际执行时间要比CPU Time稍微长一点。

### Systrace UI性能分析

在应用程序开发过程中，UI（用户界面）的流畅度是体验的核心，特别是在动画、跳转或者列表的滑动过程中，出现卡顿和无响应是非常影响用户体验的，要解决这些问题，首先要找到问题的原因，前面介绍的TraceView是分析性能的一款利器，下面再介绍一个分析应用程序UI性能的工具：Systrace。

Systrace是Android 4.1及以上版本提供的性能数据采样和分析工具。它可以帮助开发者收集Android关键子系统（如surfaceflinger、WindowManagerService等Framework部分关键模块、服务，View系统等）的运行信息，从而帮助开发者更直观地分析系统瓶颈，改进性能。Systrace的功能包括跟踪系统的I/O操作、内核工作队列、CPU负载等，在UI显示性能分析上提供很好的数据，特别是在动画播放不流畅、渲染卡等问题上。Systrace工具可以跟踪、收集、检查定时信息，可以很直观地查看CPU周期消耗的具体时间，显示每个线程和进程的跟踪信息，使用不同颜色来突出问题的严重性，并提供如何解决这些问题的建议。

[图片]注意　由于Systrace是以系统的角度返回一些信息，并不能定位到具体耗时的方法，要进一步获取CPU满负荷运行的原因，就需要使用前面介绍过的工具Traceview。

#### Systrace使用方法

Systrace的使用不复杂。但跟踪的设备必须是Android 4.1（API16）或更高版本。在4.3版本和4.3以前版本的使用上有些区别，后面会讲到。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　4.3以前系统版本的设备需要打开Settings>Developer options>Monitoring>Enable traces。

1. 在DDMS上使用

在Eclipse和Android Studio中都可以在DDMS直接使用Systrace，其他IDE也能支持，且流程都相同，下面以Android Studio为例说明其使用流程。

1）打开Android Device Monitor，连接手机并准备需要抓取的界面。

2）单击Systrace按钮进入抓取前的设置，选择需要跟踪的内容（见图2-18）：



图2-18　DDMS上打开Systrace

3）手机上开始操作需要跟踪的过程（如滑动列表）。

4）到了设定好的时间后，生成Trace文件。

5）使用Chrome打开文件即可分析。

**（2）使用命令行**

使用命令行方式更灵活，速度更快，并且配置好后再使用能快速得到结果，在Android 4.3及更高版本的设备上使用Systrace时，可以省略设置跟踪类别标签来获取默认值，或者可以手动列入指定标签。路径和命令如下：



[代码] python systrace.py --time=10 -o mynewtrace.html sched gfx view wm

其中参数设置对应的功能如表2-2所示。



表2-2　System参数命令

其中categories中的标签比较多，可以从[官方的文档上查询](http://developer.android.com/intl/zh-cn/tools/help/systrace.html)：

1. **应用中获取**

Systrace不会追踪应用的所有工作，所以在有需求的情况下，需要添加要追踪的代码部分。在Android 4.3及以上版本的代码中，可以通过Trace类来实现这个功能。它能够让你在任何时候跟踪应用的一举一动。在获取Trace的过程中，即Trace.beginSection（）与Trace.endSection（）之间的代码工作会一直被追踪。

在代码中加入Trace跟踪需要注意以下两点：

·在Trace被嵌套在另一个Trace中时，endSection（）方法只会结束离它最近的一个beginSection（String），即在一个Trace的过程中是无法中断其他Trace的。所以要保证endSection（）与beginSection（String）调用次数匹配。

·Trace的begin与end必须在同一线程中执行。

下面这部分代码为使用Trace的例子，在整个方法中含有两个Trace块，可以根据需求定义更多的块，但都要成对出现，如果有开始块但没有结束块，会严重影响应用的性能。

#### 分析Systrace报告

通过前面方法获取到的trace.html文件，需要使用Chrome打开，有一些常用的快捷键，定义如表2-3所示。



目前Systrace产生的trace文件只能使用Chrome打开，使用Chrome打开文件后如图2-19所示。



图2-19　Systrace Viewer

从图2-19中可以看到完整的数据，其中和UI绘制关系最密切的是Alerts和Frame两个数据，接下来重点介绍Alerts和Frame。

（1）Alerts

从图2-19可以看到，Alerts一栏标记了性能有问题的点，单击该点可以查看详细信息，在右边侧边栏还有一个Alerts框，单击可以查看每个类型的Alerts的数量，单击某一个Alert可以看到问题的详细描述。

（2）Frame

每个应用都有一行专门显示frame，每一帧就显示为一个绿色的圆圈。当显示为黄色或者红色时，它的渲染时间超过了16.6ms（即达不到60fps的水准）。使用W键放大，看看这一帧的渲染过程中系统到底做了什么，同时它会将任何它认为性能有问题的东西都高亮警告，并提示要怎么优化。如图2-19所示，在Frame栏有一个F帧（第二帧）黄色告警，从下面的问题详细描述可以看出，警告的主要原因是ListView的回收和重新绑定花费太多时间。在Systrace中也会提供一些对应链接，提供更多解释。

如果想知道UI线程怎么会花费这么多时间的话，就需要使用2.2.2节讲到的TraceView，来分析具体是哪些函数在消耗时间。

## 布局优化

布局是否合理主要影响的是页面测量时间的多少，我们知道一个页面的显示测量和绘制过程都是通过递归来完成的，多叉树遍历的时间与树的高度h相关，其时间复杂度为O（h），如果层级太深，每增加一层则会增加更多的页面显示时间。

任何时候View中的绘制内容发生变化时，都需要重新创建DisplayList、渲染DisplayList，更新到屏幕上等一系列操作。这个流程的表现性能取决于View的复杂程度、View的状态变化以及渲染管道的执行性能。例如，假设某个Button的大小需要增大到目前的两倍，在增大Button大小之前，需要通过父View重新计算并摆放其他子View的位置。修改View的大小会触发整个HierarcyView的重新计算大小的操作。如果是修改View的位置，则会触发HierarchView重新计算其他View的位置。如果布局很复杂，就很容易导致严重的性能问题。

在优化前首先讲解两个布局优化的常用工具。

### 常用布局优化工具

#### Hierarchy Viewer

Hierarchy Viewer是Android SDK自带的一款可视化调试工具，用来检查Layout嵌套及绘制时间，以可视化的布局角度直观获取Layout布局设计和各种属性信息，开发者在调试和布局UI界面时可以很方便地使用，提高用户的开发效率。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意

* ·出于安全考虑，Hierarchy Viewer只能连接**Android开发版手机**或模拟器。
* ·在应用程序DEBUG模式中，无法启动Hierarchy Viewer。

接下来一步步介绍如何使用Hierarchy Viewer。

**Step1：构造页面**

先构造一个简单的页面LayoutPerActivity，该页面如图2-20所示，然后启动应用，进入这个页面。



图2-20　页面显示

**Step2：打开Hierarchy View**

Eclipse和Android Studio都带有Hierarchy View工具，下面介绍在这两款IDE上打开Hierarchy View的方法。

·Android Studio

在Android Studio上可以直接在快捷工具栏打开Android Device Monitor，如图2-21所示。Android Device Monitor上就有Hierarchy View视图，可直接查看。选择Tools->Android->Android Device Monitor菜单，直接打开Hierarchy View。



Eclipse(命令行方法):可以从Android SDK工具包中，通过命令行的方式启动，在Android SDK下的tools目录sdk\tools下，在命令行方式下运行hierachyviewer。

使用AS打开后的整体窗口如图2-22所示。



图2-22　Hierarchy View整体窗口

可以看到4个窗口，各个窗口的功能如下：

·Windows：显示当前设备信息，以及当前设备的所有页面列表。

·View Properties：当前选中View的属性。

·TreeView：把Activity中所有控件（View）的层次结构从左到右显示出来，其中最右边部分是最底层的控件（View）。

·Tree Overview：全局概览，以缩略图的方式显示整个应用中各控件的层次关系，并且框出TreeView窗口中显示部分在全局中的位置，如果一个界面中的控件和层级比较多，可以通过鼠标移动这个显示区域移动。

·Layout View：整体Layout布局图，以手机屏幕上真实位置呈现出来，在TreeView中选中某一个控件时，会在Layout View用红色的框标注。

Step3：使用Hierarchy Viewer查看层级和耗时

·查看层级图：在Windows窗口页，选择需要查看的组件，双击或单击Load View Hierarchy按钮即可打开。双击后在Tree View界面中，从左到右把所有控件层级图显示出来，这样就可以看到整体界面的层级深度。

·查看某个View的耗时：在快捷键工具栏中单击Obtain layout times for tree rooted at selected node按钮，如图2-23所示。



图2-23　查看单个View的耗时

这时可以看到Tree View中的页面增加了属性，单击一个控件，可以看到这个View的耗时情况。

根据图2-24从上往下看，1 view表示这个控件是这个树下的最后一个控件，即表示是它本身，下面的时间表示Measure、Layout以及Draw三个阶段的耗时。最后一个框有不同色的三个指示灯，分别对应当前控件在测量、布局以及画视图三个阶段，颜色表示这个控件占用的时间百分比，如果是绿色的，表示该控件在该阶段比其他50%的控件的速度要快，黄色表示比其他50%的控件的速度要慢，红色表示该控件在该阶段的处理速度是最慢的，就需要注意了



图2-24　单个View的耗时图

到这里我们就知道如何使用Hierarchy View来分析一个页面的层级和耗时，并且使用这个工具，用户可以很方便地查看和调试应用中的UI界面，分析其性能，建议开发者在开发阶段也使用这款工具。但一个应用的界面非常多，如果一个个这样分析的话效率非常低，所以再介绍另外一个工具Lint，用于检查所有页面的层级，并把深度高于N（自定义）的界面输出，然后通过Hierarchy View工具来仔细分析。

#### 布局层级检查

Android Lint是Android SDK Tools 16（ADT 16）之后引入的代码检查工具，通过代码静态检查，可以发现潜在的代码问题，并给出优化建议。Android-Lint检查工具使用的方式有以下两种：

·命令行使用脚本执行。

·在IDE中使用视图化工具。

Lint的检查结果分为6类，如图2-25所示。



图2-25　Android Lint示意图

* Correctness（正确性）
* Security（安全性）
* ·Performance（性能）
* ·Usability（可用性）
* ·Accessibility（可达性）
* ·国际化

问题的严重程度（severity）从高到低依次是：

* ·Fatal
* ·Error
* ·Warning
* ·Information
* ·Ignore

扫描规则和缺陷级别可以在File→Settings→Inspections→Android Lint中配置，Lint的功能非常强大，强烈建议开发者深入学习使用方法，这里只讲解如何使用Android Lint来发现XML布局检查。使用Lint扫描前，先配置需要检查的项目，只需要检查Layout层级深度。先进入File→Settings→Inspections→Android Lint。

如图2-26所示，这里的配置只扫描Layout的层级和View的个数，如下所示：

·TooDeepLayout：表示布局太深，默认**层级超过10**层会提示该问题，可以自定义环境变量**ANDROID\_LINT\_MAX\_DEPTH**来修改。布局深度增加会导致内存消耗也随之增加，因此布局尽可能浅而宽。

·TooManyViews：表示控件太多，默认**超过80个控件**会提示该问题。

在Android Studio中启动Lint，从菜单栏选择Analyze→Inspect Code，进去后可以指定扫描的范围，可以是整个工程，也可以是一个Module或单独的文件。启动扫描，扫描结果如图2-27所示。



图2-26　设置Lint规则



图2-27　扫描结果

可以很清楚地显示哪个layout有问题，进而打开对应文件并修改。介绍完两个工具，接下来我们从多个方面来优化UI，让应用使用更流畅。

### 布局优化方法

在Android应用开发中，常用的布局方式主要有LinearLayout、RelativeLayout、FrameLayout等，通过这些布局可以实现各种各样的界面。我们需要知道如何高效地使用这些布局方式来组织UI控件，布局的好坏影响到绘制的时间，本节将通过减少Layout层级，减少测量、绘制时间，提高复用性三个方面来优化布局，优化的目的就是减少层级，让布局扁平化，以提高绘制的时间，提高布局的复用性节省开发和维护成本。

#### 减少层级

层级越少，测试和绘制的时间就越短，通常减少层级有以下两个常用方案：

* 合理使用RelativeLayout和LinearLayout。
* 合理使用Merge。

（1）RelativeLayout与LinearLayout

使用LinearLayout布局的具体方法，使用RelativeLayout进行优化，达到相同的布局效果，并且RelativeLayout允许子元素指定它们相对于其他元素或父元素的位置，有最大自由度的布局属性，而且布局层次最浅，占用内存最少。Xml代码不再展开，优化后用Hierarchy View检查结果

但**ReativeLayout**也存在**性能低**的问题，原因是RelativeLayout会对子View做两次测量，在RelativeLayout中子View的排列方式是基于彼此的依赖关系，因为这个依赖关系可能和布局中View的顺序并不相同，在确定每个子View的位置时，需要先给所有子View做一次排序。如果在RelativeLayout中允许子View横向和纵向互相依赖，就需要横向、纵向分别进行一次排序测量。但如果在LinearLayout中有weight属性，也需要进行两次测量，因为没有更多的依赖关系，所以仍然会比RelativeLayout的效率高，在布局上RelativeLayout不如LinearLayout快。

但是如果布局本身层次太深，还是推荐用RelativeLayout减少布局本身层次，相较于测量两次，虽然会增加一些计算时间，但在体验上影响不会特别大，如果优化掉两层仅仅是增加一次测量，还是非常值得的，布局层次深会增加内存消甚至引起栈溢出等问题，即使耗点时间，也不能让应用不可用。

根据以上分析，可以总结出以下几点布局原则：**扁平化设计**

* ·尽量使用RelativeLayout和LinearLayout。
* ·在布局层级相同的情况下，使用LinearLayout。
* ·用LinearLayout有时会使嵌套层级变多，应该使用RelativeLayout，使界面尽量扁平化。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　由于Android的碎片化程度很高，市面上的屏幕尺寸也是各式各样，使用RelativeLayout能使构建的布局适应性更强，构建出来的UI布局对多屏幕的适配效果更好，通过指定UI控件间的相对位置，使不同屏幕上布局的表现基本保持一致。当然，也不是所有情况下都得使用相对布局，根据具体情况选择和搭配使用其他布局方式来实现最优布局

（2）Merge的使用

从名字上就可以看出，Merge就是合并的意思。使用它可以有效优化某些符合条件的多余的层级。使用Merge的场合主要有以下两处：

* ·在自定义View中使用，父元素尽量是FrameLayout或者LinearLayout。
* ·在Activity中整体布局，根元素需要是FrameLayout。

我们仍以前面的布局为例，在页面增加一个自定义控件TopBar，故在代码清单2-2布局的基础上增加如下代码：

|  |
| --- |
| <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:orientation="vertical">  <k.opt.ui.TopBar  android:id="@+id/lay\_out\_topbar"  android:layout\_width="fill\_parent"  android:background="@color/black\_overlay"  android:layout\_height="80dp"/>  <TextView  android:id="@+id/layout\_per\_txt\_1"  android:layout\_width="100dp"  android:layout\_height="100dp"  android:layout\_below="@id/lay\_out\_topbar"  android:background="@color/yellow"  android:gravity="center"  android:text="TITLE"  android:textColor="#FFFF0000"  android:textSize="20sp" />  <TextView  android:id="@+id/layout\_per\_txt\_2"  android:layout\_width="match\_parent"  android:layout\_height="50dp"  android:layout\_below="@id/lay\_out\_topbar"  android:layout\_toEndOf="@id/layout\_per\_txt\_1"  android:layout\_toRightOf="@id/layout\_per\_txt\_1"  android:background="@color/blue"  android:text="Des"  android:textColor="#FFFF0000"  android:textSize="20sp" />  <ImageView  android:id="@+id/imageView\_2"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_below="@id/layout\_per\_txt\_2"  android:layout\_gravity="center"  android:layout\_toEndOf="@id/layout\_per\_txt\_1"  android:layout\_toRightOf="@id/layout\_per\_txt\_1"  android:background="@mipmap/ic\_launcher" />  </RelativeLayout> |

其中TopBar的XML布局如下：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:orientation="horizontal" android:layout\_width="match\_parent"  android:background="@color/black"  android:layout\_height="@dimen/topbar\_height">  <ImageView  android:layout\_gravity="center"  android:id="@+id/backImg"  android:layout\_width="60dp"  android:layout\_height="44dp"  android:background="@drawable/img\_top\_back"  android:focusable="true" />  <TextView  android:layout\_gravity="center"  android:id="@+id/titleTextView"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:ellipsize="end"  android:layout\_centerVertical="true"  android:gravity="center"  android:singleLine="true"  android:textColor="#FFFFFFFF"  android:text="标题"  android:textSize="18sp" />  </LinearLayout> |

显示结果如图2-31所示。这种布局在一些列表的Item中非常常见，而且列表中Item本身的层级比较深，因此优化显得更有意义。



我们使用HierarchyView查看增加TopBar后的布局层级，如图2-32所示。可以看到，就是这么简单的一个布局，却把层级增加了两级，从图2-32中很明显地看出TopBar后一层的LinearLayout是多余的，这时可以使用Merge把这一层消除。

使用Merge来优化布局，使用Merge标签替换LinearLayout后，原来的LinearLayout属性也没有用了，修改后的代码如代码清单2-3。

这样就把多余的LinearLayout消除了，原理是在Android布局的源码中，如果是Merge标签，那么直接将其中的子元素添加到Merge标签Parent中，这样就保证了不会引入额外的层级。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　如果Merge代替的布局元素为LinearLayout，在自定义布局代码中将LinearLayout的属性添加到引用上，如垂直或水平布局、背景色等。

但Merge不是所有地方都可以任意使用，有以下几点要求：

* ·Merge只能用在布局XML文件的根元素。
* ·使用merge来加载一个布局时，必须指定一个ViewGroup作为其父元素，并且要设置加载的**attachToRoot**参数为true（参照inflate（int，ViewGroup，boolean））。
* ·不能在ViewStub中使用Merge标签。原因就是ViewStub的inflate方法中根本没有attachToRoot的设置。

这一节讲了如何减少层级，那么在Android系统中，多少层才是合理的呢？当然是越少越好，但从Lint检查的配置上看，超过10层才会报警，实际上在开发时，随着产品设计的丰富和多样性，很容易超过10层，根据实际开发过程中超过15层就要重视并准备做优化，20层就必须修改了。在实在没有办法优化的情况下，需要把复杂的层级用自绘控件来实现，自绘控件中的图层层级再多，在布局上也只是一层，但这样也会带来过度绘制的问题，在后一章节中会重点介绍这个问题的优化方案。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg意　在Activiy的总布局中使用Merge，但又想设置整体的属性（布局方式或背景色），可以不使用setContentView方法加载Layout，而使用（id/content）将FrameLayout取出来，在代码中手动加载布局，但如果层级压力不大（小于10级），则没有必要，因为这样代码的维护性较差。

#### 提高显示速度

我们在开发的过程中会碰到这样的场景或者显示逻辑：某个布局当中的子布局非常多，但并不是所有元素都同时显示出来，而是二选一或者N选一，打开这个界面根据不同的场景和属性显示不同的Layout。例如：一个页面对不同的用户（未登录、普通用户、会员）来说，显示的布局不同。或者，有些用户喜欢对不同的元素使用INVISIBLE或者GONE隐藏，通过设计元素的visable属性来控制，这样虽然达到了隐藏的目的，但效率非常低，原因是即使将元素隐藏，它们仍在布局中，仍会测试和解析这些布局。Android提供了ViewStub控件来解决这个场景。

ViewStub是一个轻量级的View，它是一个看不见的，并且不占布局位置，占用资源非常小的视图对象。可以为ViewStub指定一个布局，加载布局时，只有ViewStub会被初始化，然后当ViewStub被设置为可见时，或是调用了ViewStub.inflate（）时，ViewStub所指向的布局会被加载和实例化，然后ViewStub的布局属性都会传给它指向的布局。这样，就可以使用ViewStub来设置是否显示某个布局。

代码清单2-4是两个ViewStub通过不同的初始化来加载两个不同的布局，以满足用户的需求。

代码清单2-4　使用ViewStub

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:background="@color/white"  android:orientation="vertical">  <include  android:id="@+id/topBar"  layout="@layout/common\_top\_bar"  android:layout\_width="match\_parent"  android:layout\_height="@dimen/topbar\_height" />  <ViewStub  android:id="@+id/viewstub\_text"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginLeft="5dip"  android:layout\_marginRight="5dip"  android:layout\_marginTop="25dip"  android:layout="@layout/viewstub\_text\_layout1"/>  <ViewStub  android:id="@+id/viewstub\_image"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_marginLeft="5dip"  android:layout\_marginRight="5dip"  android:layout\_marginTop="25dip"  android:layout="@layout/layout\_bitmap\_show"/>  </LinearLayout> |

在调用时，根据需求切换不同的Layout，这样可以提高页面初始化的速度，使用代码如下：

|  |
| --- |
| protected View createView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {  mContext = getHostActivity();  View view = inflater.inflate(R.layout.fm\_xml\_show, container, false);  if (changeView) {  ViewStub stub = (ViewStub) view.findViewById(R.id.viewstub\_text);  stub.inflate();  changeView = false;  } else {  ViewStub stub = (ViewStub) view.findViewById(R.id.viewstub\_image);  stub.inflate();  changeView = true;  }  return view;  } |

ViewStub显示有两种方式，上面代码使用的是inflate方法，也可以直接使用ViewStub.setVisibiltity（View.Visible）方法。

使用ViewStub时需要注意以下几点：

·ViewStub只能加载一次，之后ViewStub对象会被置为空。换句话说，某个被ViewStub指定的布局被加载后，就不能再通过ViewStub来控制它了。所以它不适用于需要按需显示隐藏的情况。

·ViewStub只能用来加载一个布局文件，而不是某个具体的View，当然也可以把View写在某个布局文件中。如果想操作一个具体的View，还是使用visibility属性。

·VIewStub中不能嵌套Merge标签。

不过这些限制都无伤大雅，我们还是能够用ViewStub来做很多事情，ViewStub的主要使用场景如下：

* ·在程序运行期间，某个布局在加载后，就不会有变化，除非销毁该页面再重新加载。
* ·想要控制显示与隐藏的是一个布局文件，而非某个View。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　因为ViewStub只能Inflate一次，之后会被置空，无法继续使用ViewStub来控制布局。所以当需要在运行时不止一次显示和隐藏某个布局时，使用ViewStub是无法实现的。这时只能使用View的可见性来控制。

#### 布局复用

我们在开发应用时还会碰到另一个常见的场景，就是一个相同的布局在很多页面（Activity或Fragment）会用到，如果给这些页面的布局文件都统一加上相同的布局代码，维护起来就很麻烦，可读性也差，一旦需要修改，很容易有漏掉的地方，Android的布局复用可以通过<include>标签来实现，就像提取代码公用部分一样，在编写Android布局文件时，也可以将相同的部分提取出来，在使用时，用<include>添加进去。例如：

|  |
| --- |
| <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:background="@color/white"  android:orientation="vertical">  <include  android:id="@+id/topBar"  layout="@layout/common\_top\_bar"  android:layout\_width="match\_parent"  android:layout\_height="@dimen/topbar\_height" /> |

例如，在大部分应用中，基本上所有的应用都会带有头部栏（TopBar），主要是显示标题和返回键功能，这样只需要维护一份代码，就可以修改所有的显示效果

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　类似于TopBar的这类常用控件，包括菜单，可以把具体实现抽象到页面的基类（BaseActivity）中，这样布局和具体的实现都收归到一个地方，方便维护;如果公共的控件要做一些逻辑处理，可以抽象为一个自定义的View

提高布局效率的方法总体来说就是减少层级，提高绘制速度和布局复用。影响布局效率主要有以下几点：

* ·布局的层级越少，加载速度越快。
* ·减少同一层级控件的数量，加载速度会变快。
* ·一个控件的属性越少，解析越快。

根据本节的分析，对优化的总结如下：

* ·尽量多使用RelativeLayout或LinearLayout，不要使用绝对布局AbsoluteLayout。
* ·将可复用的组件抽取出来并通过<include/>标签使用。
* ·使用<ViewStub/>标签加载一些不常用的布局。
* ·使用<merge/>标签减少布局的嵌套层次。
* ·尽可能少用wrap\_content，wrap\_content会增加布局measure时的计算成本，已知宽高为固定值时，不用wrap\_content。
* ·删除控件中的无用属性。

## 避免过度绘制

过度绘制（Overdraw）是指在屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次重叠的UI结构（如带背景的TextView）中，如果不可见的UI也在做绘制的操作，就会导致某些像素区域被绘制了多次，从而浪费多余的CPU以及GPU资源。

当设计上追求更华丽的视觉效果时，我们很容易陷入采用复杂的多层次重叠视图来实现这种视觉效果的怪圈。这很容易导致大量的性能问题，为了获得最佳性能，必须尽量减少Overdraw情况发生。

我们一般在XML布局和自定义控件中绘制，因此可以看出导致过度绘制的主要原因是：

* ·XML布局->控件有重叠且都有设置背景
* ·View自绘->View.OnDraw里面同一个区域被绘制多次

### 过度绘制检测工具

要知道是否有过度绘制的情况，可以通过手机设置中的开发者选项，打开Show GPU Overdraw选项，打开后会有不同的颜色区域表示不同的过度绘制次数，如图2-34所示。

具体步骤如下：

1）系统版本要求：需要Android 4.1以上版本。

2）在手机的“设置”→“开发者选项”中打开“显示GPU过度重绘”开关（注：对未默认开启硬件加速的界面需要同时打开“强制进行GPU渲染”开关）。

3）在设置时，如果有App已经打开，需要终止App进程，重新启动。

4）然后即可通过界面的颜色判断界面重绘的严重程度。

打开后可以根据不同的颜色观察UI上的Overdraw情况，蓝色、淡绿、淡红、深红代表4种不同程度的Overdraw情况，不同颜色的含义如下：

·无色：没有过度绘制，每个像素绘制了1次。

·蓝色：每个像素多绘制了1次。大片的蓝色还是可以接受的。如果整个窗口是蓝色的，可以尝试优化减少一次绘制。

·绿色：每个像素多绘制了2次。

·淡红：每个像素多绘制了3次。一般来说，这个区域不超过屏幕的1/4是可以接受的。

·深红：每个像素多绘制了4次或者更多。严重影响性能，需要优化，避免深红色区域。

我们的目标是尽量减少红色Overdraw，看到更多的蓝色区域。

### 如何避免过度绘制

#### 布局上的优化

在XML布局上，如果出现了过度绘制的情况，可以使用Hierarchy View来查看具体的层级情况，可以通过XML布局优化来减少层级。需要注意的是，在使用XML文件布局时，会设置很多背景，如果不是必需的，尽量移除。布局优化总结为以下几点：

* ·移除XML中非必需的背景，或根据条件设置。
* ·移除Window默认的背景。
* ·按需显示占位背景图片。

使用Android自带的一些主题时，activity往往会被设置一个默认的背景，这个背景由DecorView持有。当自定义布局有一个全屏的背景时，比如设置了这个界面的全屏黑色背景，DecorView的背景此时对我们来说是无用的，但是它会产生一次Overdraw。因此没有必要的话，也可以移除，代码如下：（why 我的并没有work？）

|  |
| --- |
| protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  this.getWindow().setBackgroundDrawable(null); |

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　针对ListView中的Avatar ImageView的设置，在getView的代码中，判断是否获取对应的Bitmap，获取Avatar的图像之后，把ImageView的Background设置为Transparent，只有当图像没有获取到时，才设置对应的Background占位图片，这样可以避免因为给Avatar设置背景图而导致的过度渲染。

#### 自定义View优化

事实上，由于我们的产品设计总是追求更华丽的视觉效果，仅仅通过布局优化很难做到最好，这时可以对复杂的控件使用自定义View来实现，虽然**自定义View减少了Layout的层级**，但在实际绘制时也是会过度绘制的。原因是有些过于复杂的自定义View（通常重写了onDraw方法），Android系统无法检测在**onDraw**中具体会执行什么操作，无法监控并自动优化，也就无法避免Overdraw了。但是在自定义View中可以通过**canvas.clipRect**（）来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制，其他的区域会被忽视。canvas.clipRect（）可以很好地帮助那些有多组重叠组件的自定义View来控制显示的区域。clipRect方法还可以帮助节约CPU与GPU资源，在clipRect区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制，并且可以使用**canvas.quickreject（）来判断是否没和**某个矩形相交，从而跳过那些非矩形区域内的绘制操作。接下来介绍使用一个自定义View避免OverDraw的案例。

#### 案例：无过度绘制View的实现

我们来实现一个四张图片叠加的自定义View。为了方便看到效果，这四张图片都有一定的重合区域，如果直接绘制，由于系统是不知道有重合区域，就会导致过度绘制，打开Show GPU Overdraw后看到如图2-35所示的效果图，可看出叠加层次越多，过度绘制就越严重。

图2-35　过度绘制优化前后对比（后为优化结果）

从图2-35中可以看出，重叠部分都有过度绘制的情况，接下来通过一个例子来避免这种情况，以下代码是描述其中一张图片的类。详细代码参考MultiCardsView，SingleCard，OverDrawActivity，下面主要讲解MultiCardsView的核心代码drawCardsWithotOver-Draw

|  |
| --- |
| //没有过度绘制的实现  protected void drawCardsWithotOverDraw(Canvas canvas, int index) {  if (canvas == null || index < 0 || index >= cardsList.size())  return;  SingleCard card = cardsList.get(index);  //判断是否没和某个卡片相交，从而跳过那些非矩形区域内的绘制操作  if (card != null && !canvas.quickReject(card.area, Canvas.EdgeType.BW)) {  int saveCount = canvas.save(Canvas.CLIP\_SAVE\_FLAG);  //只绘制可见区域  if (canvas.clipRect(card.area, Region.Op.DIFFERENCE)) {  drawCardsWithotOverDraw(canvas, index - 1);  }  canvas.restoreToCount(saveCount);  saveCount = canvas.save(Canvas.CLIP\_SAVE\_FLAG);  //只绘制可见区域  if (canvas.clipRect(card.area)) {  KLogUtil.D("draw", "overdraw opt: draw cards index: " + index);  Rect clip = canvas.getClipBounds();  KLogUtil.D("draw", String.format("current clip bounds %d %d %d %d", clip.left, clip.top, clip.right, clip.bottom));  card.draw(canvas);  }  canvas.restoreToCount(saveCount);  }else{  drawCardsWithotOverDraw(canvas, index - 1);  }  } |

可以看出，使用drawCardsWithotOver-Draw（）避免了过度绘制，从代码清单2-6中可以看到，调用了两个关键的方法：

·快速判断Canvas是否需要绘制：Canvas.QuickReject。

在绘制一个单元之前，首先判断该单元的区域是否在Canvas的剪切域内。若不在，直接返回，避免CPU和GPU的计算和渲染工作。

·避免绘制越界：Canvas.ClipRect。

每个绘制单元都有自己的绘制区域，绘制前，Canvas.**ClipRect**（Region.Op.INTERSECT）帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内，才会被绘制，其他的区域被忽视。这个API可以很好地帮助那些有多组重叠组件的自定义View来控制显示的区域。clipRect方法还可以帮助节约CPU与GPU资源，在clipRect区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制。

这个案例可以避免层次很多的自定义View导致过度绘制的问题。比如纸牌堆叠。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　在listview或其他容器控件中，itemview如果比较复杂，建议实现成一个自绘View，使用此案例来绘制可以使listview滑动更流畅。

## 启动优化

随着应用的功能越来越丰富、启动时需要初始化的工作多、界面的元素复杂等，启动速度不可避免地受到影响，比如一开始单击时出现黑屏或者白屏，甚至在低端机型上出现假死的现象，本节通过学习应用的启动流程、启动速度的监控，发现影响启动速度的问题所在，并优化启动的逻辑，提高应用的启动速度。

### 应用启动流程

Android应用程序的载体是APK文件，其中包含了组件和资源，APK文件可能运行在一个独立的进程中，也有可能产生多个进程，还可以多个APK运行在同一个进程中，可以通过不同的方式来实现。但有两点需要注意，第一，每个应用只对应一个Application对象，并且启动应用一定会产生一个Application对象；第二，应用程序可视化组件Activity是应用的基本组成之一，因此要分析启动的性能，就有必要了解这两个对象的工作流程和生命周期。

#### Application

Application是Android系统框架中的一个系统组件，Android程序启动时，系统会创建一个Application对象，用来存储系统的一些信息。Android系统会自动在每个程序运行时创建一个Application类的对象，并且只创建一个，可以理解为Application是一个单例类。

应用可以不指定一个具体的Application，系统会自动创建，但一般在开发中都会创建一个继承于系统Application的类实现一些功能，比如一些数据库的创建、模块的初始化等。但这个派生类必须在AndroidManifest.xml中定义好，在application标签增加name属性，并添加自己的Application的类名。

启动Application时，系统会创建一个PID，即进程ID，所有的Activity都会在此进程上运行。在Application创建时初始化全局变量，同一个应用的所有Activity都可以取到这些全局变量的值，Application对象的生命周期是整个程序中最长的，它的生命周期就等于这个应用程序的生命周期，因为它是全局的单例的，所以在不同的Activity或者Service中获得的对象都是同一个对象。因此在安卓中要避免使用静态变量来存储长久保存的值，可以用Application，但并不建议使用太多的全局变量。

AndroidManifest.xml文件上的application标签指定了重写的Application类后，看看该类可以重载的几个抽象接口，代码清单2-7是一个自定义的Application。

代码清单2-7　自定义的Application

|  |
| --- |
|  |

从代码清单2-7中可以看到几个重要的抽象接口，这些接口的调用时机如下：

·attachBaseContext（Context base）：得到应用上下文的Context，在应用创建时首先调用。

·onCreate（）：应用创建时调用，晚于attachBaseContext（）方法。

·onTerminate（）：应用结束时调用。

·onConfigurationChanged（）：系统配置发生变化时调用。

·onLowMemory（）：系统低内存时调用。

·onTrimMemory（int level）：系统要求应用释放内存时调用，level为级别。

从上面的抽象方法可以看出，这些方法都在这个应用生命周期之中，attachBaseContext和onCreate在应用创建时必须调用，而其他需要满足一定的触发时机。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　在开发过程中，尽量使用Application中的Context实例，因为使用Activity中的Context可能会导致内存泄漏。也可以使用Activity的getApplicationContext方法。

#### Activity

Activity大家都非常熟悉了，这里也不做太多解释，只需要理解它的生命周期，因为这在启动优化的过程中非常重要。

在Activity的生命周期中，系统会按类似于阶梯金字塔的顺序调用一组核心的生命周期方法，如图2-37所示。也就是说，Activity生命周期的每个阶段就是金字塔上的一阶。当系统创建一个新Activity实例时，每个回调方法会将Activity状态向顶端移动一阶。金字塔顶端是Activity在前台运行并且用户可以与其交互的时间点。当用户开始离开Activity时，系统调用其他方法在金字塔中将Activity状态下移，从而销毁Activity。在有些情况下，Activity将只在金字塔中部分下移并等待（如当用户切换到其他应用时），Activity可从该点开始移回顶端（如果用户返回到该Activity），并在用户停止的位置继续。



图2-37　Activty生命周期金字塔模型

大多数应用包含若干不同的Activity，用户可通过这些Activity执行不同的操作。无论Activity是用户单击应用图标时创建的主Activity，还是应用在响应用户操作时开始的其他Activity，系统都会调用其onCreate（）方法创建Activity的每个新实例。因此必须实现onCreate（）方法，执行后，在Activity整个生命周期中只需要出现一次基本应用启动逻辑。例如，onCreate（）的实现应定义用户界面并且可能实例化某些类范围变量、声明用户界面（在XML布局文件中定义）、定义成员变量，以及配置某些UI。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　onCreate（）方法包括一个savedInstanceState参数，在有关重新创建Activity中非常有用。

从Application和Activity的介绍中，可以总结出应用启动的流程，如图2-38所示。其中，启动分为两种类型：冷启动和热启动。

·冷启动：因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化Application类，再创建和初始化Main-Activity类（包括一系列的测量、布局、绘制），最后显示在界面上，如图2-38所示。

·热启动：因为会从已有的进程中启动，所以热启动不会再创建和初始化Application，而是直接创建和初始化MainActivity（包括一系列的测量、布局、绘制），即Application只会初始化一次，只包含Activity中的生命周期流程。



2-38　应用启动流程

### 启动耗时监测

因为一个应用在启动或者跳入某个页面时是否流畅，时间是否太长，仅仅通过肉眼来观察是非常不准确的，并且在不同设备和环境会有完全不同的表现，所以要准确知道耗时，就需要有效准确的数据，首先通过shell来获取启动耗时。

#### adb shell am

应用启动的时间会受到很多因素的影响，比如首次安装后需要解压apk文件，绘制时GPU的耗时等，所以在应用层很难获取到启动耗时，但借助ADB可以得到准确的启动时间。

使用adb shell获得应用真实的启动时间，代码如下：

adb shell am start -W [packageName]/[packageName.AppstartActivity]

执行后可以得到三个时间：

* ·ThisTime：一般和TotalTime时间一样，如果在应用启动时开了一个过度的全透明的页面（Activity）预先处理一些事，再显示出主页面（Activity），这样将比TotalTime小。
* ·TotalTime：应用的启动时间，包括创建进程+Application初始化+Activity
* 初始化到界面显示。
* ·WaitTime：一般比TotalTime大些，包括系统影响的耗时。

但这个方法只能得到固定的某一个阶段的耗时，不能得到具体哪个方法的耗时，下面介绍第二个方案：代码打点输出耗时。

#### 代码打点

通过代码打点来准确获取记录每个方法的执行时间，知道哪些地方耗时，然后再有针对性地优化，下面通过一个简单的例子来讲解打点的方案。

以下代码（看随书代码）是一个统计耗时的数据结构，通过这个数据结构记录整个过程的耗时情况。

这个对象可以用在很多需要统计的地方，不仅可以统计应用启动的耗时，还可以统计其他模块，如统计一个Activity的启动耗时和一个Fragment的启动耗时。流程为：在创建这个对象时，需要传入一个ID，这个ID是需要统计的模块或者一个生命周期流程的ID，ID自定义并且是唯一的，一个TimeMonitor对应一个ID。其中end（Boolean writeLog）方法表示这个监控的流程结束，其中writeLog表示是否需要写入本地，建议实现这个方法，可以统计一系列的数据，最好上传到服务器，用来监控这个应用在外网的实际启动状况。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　上传到服务器时建议抽样上报，比如根据用户ID的尾号来抽样上报，虽然不影响性能，但还是尽量不要全部上报，用后台下发抽样比较好。

比如现在要统计启动应用在各阶段的耗时，就自定义一个ID，为了使代码更好管理，编写一个专门定义所有ID的类，方便以后的维护

因为耗时统计可能会在多个模块和类中需要打点，所以需要一个单例类来管理各个耗时统计的数据，这里使用了一个单例类来实现：TimeMonitorManager

在有需要的地方通过这个方法进行打点，为了得到有效的数据，总结起来主要在两个方面需要打点：

·应用程序的生命周期节点，如Application的onCreate、Activity或Fragment的回调函（onCreate、onResume等）。

·启动时需要初始化的重要方法，如数据库初始化、读取本地的一些数据等。

·其他耗时的一些算法。

例如，在启动时加入统计，在Application和第一个Activity加入打点统计，结合前面讲过的启动生命周期，首先进入的是Application的attachBaseContext（）方法，然后在Oncreate结束时打第一个点，在AppstartActivity结束打第二个点，在AppstartActivity中的onStart（）打最后一个点，代码如下（看随书代码）：

结果如下所示：



https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　可以在项目中核心基类的关键回调函数和核心方法加入打点，另外插桩也是一种不错的方式。

### 启动优化方案

在Android应用开发中，应用启动速度对用户体验非常重要，也是一个应用给用户的第一个性能方面的体验，因此应用启动优化是非常有必要的。应用启动优化的核心思想就是要快，在启动过程中做尽量少的事。但是应用功能越丰富，模块越多，需要初始化的地方也越多，导致了应用启动变慢。

为了优化启动的速度，首先要了解启动时做了什么，来看一个例子，启动源码中性能优化的启动应用（[应用源码在](http://github.com/lyc7898/AndroidTech)），通过打点，统计从单击打开应用到首页显示完成的时间，后面章节会讲到打点的注意事项和具体实现。表2-4是通过打点获取到这个应用启动时，各个模块占用的时间。



表2-4　各模块的时间占比

因为这个应用比较简单，没有什么模块和数据，所以大头是在绘制工作上，也就是闪屏页（显示启动LOGO的页面）和进入后首页的布局上，其次是一些初始化工作，但实际上一个稍大型的应用，模块初始化和数据准备工作的占比会高很多，因为这块的优化也是非常有必要。

从总体上看，启动主要完成三件事：UI布局、绘制和数据准备，因此启动速度的优化就是需要优化这三个过程，我们也可以通过一些启动界面策略进行优化。接下来从启动耗时最高的UI布局和启动加载逻辑两个方向优化，达到降低启动耗时的目的。因为这个应用非常简单，所以不具有代表性，但优化的流程和方案是通用的。

#### UI布局

这个应用启动过程为：启动应用→Application初始化→AppstartActivity→HomePageActivity。AppstartActivity是应用的闪屏页（也叫启动页），我们看到大部分应用都有这么一个页面，为什么要有闪屏页呢？闪屏页的存在主要有两个好处：一是可以作为品牌宣传展示，如节日运营或热点事件运营，也可以做广告展示（不要太低端）；其二，因为闪屏一般需要停留一段时间，在这段时间可以做很多事情，比如底层模块的初始化、数据的预拉取等。

首先需要优化AppstartActivity的布局，从前面的章节可以知道，要提高显示的效率，一是减少布局层级，二是避免过度绘制，因为前面已经有很详细的例子了，这里不做过多介绍，优化的步骤如下：

·使用Profile GPU Rendering检查启动时是否有严重的掉帧，见2.2.1节。

·使用Hierarchy View检查布局文件（XML）分析布局并优化，见2.3.1节。

#### 启动加载逻辑优化

一个应用越大，涉及的模块越多，包含的服务甚至进程就会越多，如网络模块的初始化、底层数据初始化等，这些加载都需要提前准备好，有些不必要的就不要放到应用中。可以用以下四个维度分整理启动的各个点：

·必要且耗时：启动初始化，**考虑用线程来初始化。**

·必要不耗时：首页绘制。

·非必要耗时：数据上报、**插件初始化。**

·非必要不耗时：不用想，这块直接去掉，在需要用的时再加载。

把数据整理出来后，按需实现加载逻辑，采取分步加载、异步加载、延期加载策略，如图2-39所示。



图2-39　启动优化方向

提高应用的启动速度，核心思想是在启动过程中少做事情，越少越好。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　在应用中，增加启动默认图或者自定义一个Theme，在Activity首先使用一个默认的界面可以解决部分启动短暂黑屏问题，如:

|  |
| --- |
| <**application**  android:name=".KApplication"  **android:theme="@style/KTheme**Loader**"**  …  > |

## 合理的刷新机制

在应用开发的过程中，因为数据的变化，需要刷新页面来展示新的数据，但频繁刷新会增加资源开销，并且可能导致卡顿发生，所以，需要一个合理的刷新机制来提高整体的UI流畅度。合理的刷新需要注意以下几点：

* ·尽量减少刷新次数。
* ·尽量避免后台有高CPU线程运行。
* ·缩小刷新区域。

### 减少刷新次数

毫无疑问，减少刷新次数可以减少系统的开销，在功耗和页面的性能上可以表现得更优秀，但不刷新就不能及时让用户看到最新的数据，可以从以下几个方面减少刷新次数：

#### 控制刷新频率

在有些功能上需要频繁刷新某个控件（View），比如下载进度条或者播放进度条，没有必要在数据每次变化时都更新对应的控件，需要注意的是一定不能出现过度刷新（进度刷新频率大于系统显示的刷新频率）的情况。可以通过定时控制刷新频率，相同的刷新只做一次，比如播放进度条的刻度是100，如果数据变化没有1%，完全没有必要刷新，这样可以减少UI的刷新负担。

#### 避免没有必要的刷新

首先需要判断是否需要刷新，比如数据没有变化、需要刷新的控件（View）不在可见区域，就没有必要刷新。但是需要注意，如果一个View从不可见到可见，一定要刷新一次。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　一般来说，在第一片数据（从无到有）和最后一片数据（结束了一个刷新周期）时，一定要刷新一次，以保证完整性。

### 避免后台线程影响

后台线程虽然不会直接影响到主线程的工作，但如果后台线程开销很大，占用CPU过高，导致系统GC频繁和CPU时间片资源紧张，还是有可能会导致页面的卡顿。因此在需要迅速刷新的情况下避免这类线程在高峰工作。比如ListView的滚动，如果ListView中的Item需要下载图片显示，在ListView滚动时可以暂停其他UI的操作，示例代码如下：

|  |
| --- |
| listView.setOnScrollListener(new AbsListView.OnScrollListener() {  @Override  public void onScrollStateChanged(AbsListView absListView, int scrollState) {  if (scrollState == AbsListView.OnScrollListener.SCROLL\_STATE\_FLING) {  MiniImageLoader.getInstance().setPauseWork(true);  } else {  MiniImageLoader.getInstance().setPauseWork(false);  }  } |

通过监听ListView的onScrollStateChanged事件，在滚动时暂停图片下载线程工作，结束后再开始，可以提高ListView的滚动平滑度。

### 缩小刷新区域

在以下两个场景下可以采用局部刷新的方法来节省更多的资源。

一是在自定义View中。自定义View一般采用invalidata方法刷新。如果需要更新的数据只是在某一个区域内改变，在调用invalidata方法更新这个区域时，也会更新整个视图，这就浪费了不需要更新区域资源。Android提供系统了两个局部更新数据的方法：

* ·**invalidate**（Rect dirty）；
* ·invalidate（int left，int top，int right，int bottom）；

使用这两个方法可以只更新需要更新的区域，其他不需要数据更新的区域不会更新，这样能节省部分资源。

第二种是容器中的某个Item发生了变化，只需要更新这一个Item即可。比如在ListView中，如果是单条操作，就必须调用Adapter的**notifyDataSetChanged**（）刷新。

## 提升动画性能

在打造优秀体验的应用和实现酷炫效果的过程中，动画是不可或缺的重要组成部分。Android平台提供了三个动画框架：帧动画（Frame Animation）、补间动画（Tween Animation）和属性动画（Property Animation）。属性动画在Android 3.0开始支持，开发者使用这些动画框架来实现各种动画效果，这三个框架都有其优势和局限性，要深入了解就需要明白它们的实现原理。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　属性动画只有Android API 11（Android 3.0）以上才支持，如果使用11以下的SDK，请导入NineOldAndroids动画库，用法完全一致。

虽然通过动画可以实现很多酷炫的动画，但带来的性能开销也有不同程度的影响。在实现动画的过程中，主要从以下三个纬度来对比性能：

* ·流畅度：流畅度是动画的核心，控制每一帧动画在16ms以内完成。
* ·内存：避免内存泄漏，减小内存开销。
* ·耗电：减小运算量，优化算法，减小CPU占用。

因篇幅所限，本文不从原理上讲解所有动画的具体实现，只简单了解下目前Android系统支持的三种动画实现方式，并且以应用启动动画为例来对比不同动画的实现效果和性能上的开销，这个启动动画由两部分组成：旋转360度和从0到原始尺寸的放大。

### 帧动画

帧动画很简单，就是将一组图片按顺序显示出来，可以使用AnimationDrawable来定义使用帧动画。但由于帧动画要求图片过多，如果图片较大，内存占用就非常大，效果也比较差，除非图片足够多。所以帧动画是消耗资源最多，效果最差的一种，这里就不再介绍使用方法，能不用就不用。

### 补间动画

补间动画是通过对某个View进行一系列的操作来改变显示效果，对比逐帧动画，它的使用更简单方便。补间动画不需要定义时间频率内的每一帧，只需要定义开始和结束关键帧的内容，两个关键帧之间的效果自动生成，使用时不需要另写代码控制，而逐帧动画的帧与帧之间的过渡并不连贯，并且消耗更多的内存。补间动画支持淡入淡出（AlphaAnimation）、缩放（ScaleAnimation）、平移（TranslationAnimation）和旋转（RotateAnimation）4种动画模式。

以缩放和旋转两种动画来实现一个组合动画，让一个ImagaView由小到大并同步旋转出来，代码如下随书：

运行后效果很流畅，接下来使用TraceView看看性能如何（TraceView的使用方法详见2.2.2节），如图2-40所示。



图2-40　补间动画的使用性能分析

从图2-40可以看出，当前只有主线程，这就更利于分析了，通过前面的绘制原理（详见2.1节）我们知道，最后把数据送到显示器上需要更新DisplayList，所以我们来看下android.view.View.updateDisplayListIfDirty（）方法，通过Calls+Recur Calls参数可以看到，这个方法调用了63次，递归调用了254次，说明使用补间动画实现动画导致View重绘非常频繁，这里先讲到这里，在后面再对比性能。

* ·补间动画虽然能很好地实现动画，也方便使用，但有以下几点局限性：
* ·补间动画只能用于View对象，也就是只有继承于View或者View的控件，才能用补间动画实现动画效果。
* ·补间动画只有4种动画操作，淡入淡出、缩放、平移和旋转。

·补间动画是改变View的显示效果，但是不会真正改变View的属性，比如为一个按钮添加了单击事件，通过补间动画让它移动到其他位置，这时单击事件是绝对不会触发的，原因是补间动画只是将该按钮绘制到屏幕的其他地方，实际上触发点还是停在原来的位置。

针对这些问题，Android 3.0版本开始推出了新的动画实现方式：Property Animation。

### 属性动画

属性动画由Android 3.0（API 11）及更高版本支持，通过修改动画的实际属性来实现动画效果。属性动画系统是一个非常全面的框架，几乎允许把任何对象变成动画。可以根据时间的推移通过改变任何对象的属性来定义一个动画，并且不用关心该对象是否要绘制在屏幕上，在指定的时间长度改变一个属性的值。属性动画只是表示一个值在一段时间内的改变，当值改变时要做什么事情完全由用户自己决定。相比于补间动画，属性动画不仅可以应用于View，还可以应用于任何对象。

属性动画系统可以定义以下动画特性：

* ·持续时间（Duration）：指定动画从开始到结束的持续时间。默认长度是300ms。
* ·时间插值（Time interpolation）：这个值能够指定为计算当前动画运行时间的函数的属性值它决定动画时间范围内的变化频率。
* ·重复次数和行为（Repeat count and behavior）：指定在动画结束时是否重新播放动画，以及重复播放的次数。还能够指定动画是否能够反向回播，如果设置了反向回播，动画就会先向前再向后，重复播放，在达到播放次数时结束。
* ·动画集合（Animator sets）：把动画组织到一个逻辑集合中，可以同时、指定播放顺序的或者延迟播放它们。
* ·帧刷新延迟（Frame refresh delay）：指定动画帧的刷新频率。默认是每10秒刷新一次，但是应用程序最终的刷新帧的速度取决于系统的繁忙程度以及系统能够提供的底层定时器的反应速度。

运行后启动动画，感观上区别不大，为了与补间动画进行对比，再使用TraceView来看看android.view.View.updateDisplayListIfDirty（）这个方法的使用情况，如图2-41所示。



图2-41　属性动画的使用性能分析

通过Calls+Recur Calls参数可以看到，android.view.View.updateDisplayListIfDirty（）方法调用了5次，递归调用了30次，这说明相比于补间动画，属性动画重绘明显少很多，性能也明显更优秀，所以，在实现动画时优先考虑使用属性动画，这样不但能实现更多效果，性能也能大大提高。

虽然属性动画很好，但还是有重绘的情况，有没有办法再次提高动画的性能呢？答案是有的。Android 3.0版本同时也引入了硬件加速的概念，可以提高渲染速度，实现更平滑的动画效果。

### 硬件加速

从2.1节我们知道，在软件渲染时，即使只有一个View发生改变，也会遍历整个ViewTree来进行重绘，而其中大部分的绘图操作却没有发生任何变化。为了解决这个种浪费资源的问题，Android 3.0引入了硬件加速的概念，用来提高渲染速度，并且可以达到更顺畅的动画效果。

#### 硬件加速原理

在硬件加速的渲染模型中有一个重要的核心类：DisplayList，每个View内部都会维护一个DisplayList。

在不支持硬件渲染的Android版本中，系统是通过draw（）方法或invalidate（）方法去通知屏幕更新并重新渲染，这两个方法的区别只是实际处理绘制的方式不同；而在支持硬件渲染的Android版本中，在打开硬件渲染后绘制View时，其中执行绘制的draw（）方法会把所有绘制命令记录到一个新的显示列表（DisplayList），这个显示列表（DisplayList）包含了输出的View层级的绘制代码，但并不是加入到显示列表（DisplayList）就立刻执行，当这个ViewTree的DisplayList全都记录完毕后，由OpenGLRender负责将Root View中的DisplayList渲染到屏幕上。而invalidate（）方法只是在显示列表中记录和更新显示层级，去标记不需要绘制的View，如图2-42所示。



图2-42　硬件渲染刷新流程

通过图2-42所示的流程图可以看出，使用DisplayList，如果只有一个View有变化，只需要重绘这个View，因此可以减少很多的绘图操作次数，仅仅需要重绘一个View的DisplayList，就能明显提高渲染效率。硬件渲染模型同时也存在一个问题，当一个View和脏区域有交集时，不一定会执行draw（）方法。确保Android系统记录了一个View的显示列表，如果没有调用invalidate（）方法，即使View发生了变化，看起来也会相同，这是需要注意的地方。

#### 硬件加速控制级别

如果应用程序中只使用了标准View或者Drawable，就可以为整个系统打开硬件加速的全局设置。但并不是所有2D绘制的操作都支持硬件加速，因此开启全局硬件加速可能会对使用自定义控件造成影响，为了避免这个问题，Android提供了一个级别控制，这样可以选择启动或者禁用以下不同级别的硬件加速，目前可以控制Application、Activity、Window和View。

（1）Application级别

在Android Manifest文件中添加属性，整个应用全局使用硬件加速，代码如下：

<application android:hardwareAccelerated="true" ...>

（2）Activity级别

对Activity进行单独控制。可以启动或者禁用一个Activity的硬件加速，即使设置过全局打开硬件加速，也可以在某个Activity中禁用。

<activity android:hardwareAccelerated="false/true" />

（3）Window级别

更细粒度的控制，通过如下代码可以对某个window进行加速：

[代码]

（4）View级别

对单独的View控制硬件加速。

View.setLayerType(View.LAYER\_TYPE\_SOFTWARE, null);

在编写代码中可以检查当前View是否已经硬件加速，可以使用iew.isHardware-Accelerated（），或者Canvas.isHardwareAccelerated（）来得到结果。即使View是存在于一个已经加速的Windows上，仍然可以使用没有硬件加速的Canvas进行绘制。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　layerType如果设置成Software，会将整个View绘制到一个Bitmap中，然后依然是通过硬件加速的方式将这个Bitmap绘制到Canvas。

#### 在动画上使用硬件加速

从前面章节可以知道，如果每秒没有达到60帧，就有可能发生卡顿现象，对一个View实现动画，本身就有很多绘制的操作，再加上View本身很复杂，不一定能保持达到60FPS。因此可以通过硬件加速来渲染提高动画的性能，实现更为快速、平滑的效果。硬件纹理操作对一个View进行动画绘制，如果不调用invalidate（）方法，就可以减少对View自身频繁的重绘。同时Android 3.0的属性动画也减小了重绘，当View通过硬件层返回时，最终所有的层叠画面显示到屏幕，View的属性同时被处理好，因此只要设置这些属性，就可以明显提高绘制的效率，它们不需要View重绘，设置属性后，View会自动刷新，这也是为什么属性动画中绘制的递归比补间动画少很多的原因，而打开硬件加速，动画会更平滑、流畅。

Android 3.0以前的版本也有能力对离屏缓冲进行渲染，可以使用View的绘制缓冲，或者Canvas.saveLayer（）函数。离屏缓冲或者Layer能够更高效地处理复杂view的动画效果或者UI合成效果。Android 3.0开始对何时以及如何使用层有了更多的控制，也就是新的离屏缓冲方式View.setLayerType（type，paint）方法，这个API所带的两个参数一个是使用的层类型，另外一个是可选参数Paint。可以把Paint参数应用到颜色过滤上，特别是混合模式或者是对一个Layer进行不透明处理。一个View可以使用如下三种Layer类型之一：

* ·LAYER\_TYPE\_NONE：普通渲染方式，不会返回一个离屏的缓冲，默认值。
* ·LAYER\_TYPE\_HARDWARE：如果这个应用使用了硬件加速，这个View将会在硬件中渲染为硬件纹理，如果应用程序并没有被硬件加速，则其效果和LAYER\_TYPE\_SOFTWARE相同。
* ·LAYER\_TYPE\_SOFTWARE：此View通过软件渲染为一个Bitmap。

在2.7.3节例子的基础上，打开硬件加速，代码如下：

|  |
| --- |
| mLogo.setLayerType(View.LAYER\_TYPE\_HARDWARE,null);  ObjectAnimator scalex = ObjectAnimator.ofFloat(mLogo, "scaleX", 0, 1);  ObjectAnimator scaley = ObjectAnimator.ofFloat(mLogo, "scaleY", 0, 1);  ObjectAnimator rotation = ObjectAnimator.ofFloat(mLogo, "rotation", 0.0f, 360f);  rotation.addListener(new ObjectAnimator.AnimatorListener() {  @Override  public void onAnimationStart(Animator animation) {  }  @Override  public void onAnimationEnd(Animator animation) {  mLogo.setLayerType(View.LAYER\_TYPE\_NONE,null);  mStartHandler.sendEmptyMessageDelayed(0, 0);  }  @Override  public void onAnimationCancel(Animator animation) {  }  @Override  public void onAnimationRepeat(Animator animation) {  }  });  AnimatorSet mSetPlayer = new AnimatorSet();  mSetPlayer.setDuration(TIME\_ANIMATION);  mSetPlayer.play(scalex).with(scaley).with(rotation);  mSetPlayer.start(); |

根据前面的描述，结合代码，总结出设计一个动画的流程如下：

1. 将要执行动画的View的LayerType设置为LAYER\_TYPE\_HARDWARE。
2. 计算动画View的属性等信息，更新View的属性。
3. 若动画结束，将LayerType设置为NONE。

虽然硬件加速可以带来更好的绘制效果，但也有一些问题，以下几点需要注意：

* ·在软件渲染时，可以使用重用Bitmap的方法来节省内存，但是如果开启了硬件加速，这个方案就不起作用。
* ·开启硬件加速的View在前台运行时，需要耗费额外的内存，加速的UI切换到后台时，产生的额外内存有可能不释放。
* 当UI中存在过渡绘制时，硬性加速会比较容易发生问题，减少过度绘制的方法请参考2.4节。

## 卡顿监控方案与实现

从用户感受上来讲卡顿一般不是必现的问题，存在明显的随机性，我们在开发和测试过程中，由于设备和应用所处环境的局限性，不一定能发现所有的卡顿情况，而在用户反馈卡顿时又无从下手，即使在测试过程中发生了卡顿，然后抓了一大堆的logcat信息，要从这里找到具体的卡顿还是很难，这时候就需要一个方案来解决这个问题，那就是监控卡顿，一旦发生卡顿，就把当前有利于分析卡顿的数据抓下来发送到服务器，开发人员拿到对应的数据做分析。卡顿监控的方案和一些开源工具也挺多，大家也可以在网上找到，笔者在分析过几个工具后发现，都是利用了Looper中的Printer来实现监控。

### 监控原理

系统层Looper的关键代码如代码清单2-9所示。可以看到，在一个应用进程中，只有一个MainLooper，而在loop的方法中可以看出，Printer logging=me.mLogging；，它在每个message处理的前后被调用，而如果主线程卡住了，说明dispatchMessage内部出了问题。根据这个原理，可以做一个UI卡顿的监控方案。

代码清单2-9　Looper.java

|  |
| --- |
| public static void loop() {  ...  for (;;) {  Message msg = queue.next(); // might block  ...  // This must be in a local variable, in case a UI event sets the logger  Printer logging = me.mLogging;  if (logging != null) {  logging.println(">>>>> Dispatching to " + msg.target + " " +  msg.callback + ": " + msg.what);  }  msg.target.dispatchMessage(msg);  if (logging != null) {  logging.println("<<<<< Finished to " + msg.target + " " + msg.callback);  }  }  ...  } |

从代码清单2-9可以看出，只要重写Printer方法，通过Looper.getMainLooper（）.setMess-ageLogging（LogPrinter）方法，在自定义的LogPrinter中把时间抓下来就可以实现监控了。

根据以上分析，就可以得到一个监控的方案，如图2-43所示。该方案利用了主线程的消息队列处理机制，通过自定义Printer，然后在Printer中获取到两次被调用的时间差，这个时间差就是执行时间。在logPrinter中通过判断start和end时间差值，这个差值是主线程dispatch的message的开始和结束时间，并判定该时间超过阈值（如1000ms）时，主线程卡顿发生，并抛出各种有用信息，供开发者分析。



图2-43　卡顿监控原理

也可以在UI线程以外开启一个异步线程，定时向UI线程发送一个任务（runnable），并记下发送时间。任务的内容是将执行时间同步到发送线程，如果UI线程被阻塞，那么发送过去的任务不能被准时执行，同步回去的执行时间会与发送时间有较大的差值，设定几个阈值（1000ms或5000ms），用来评测不同情况下卡UI线程的情况，然后可以通过log Message.what的值和Message.callback的类型来判断发生场景。但这个方法会增加系统开销，一般来说没有必要，只需要监控应用中产生的数据。

如果发生了卡顿，我们需要哪些信息呢，系统提供了以下信息可以在代码中实现并抓取：

* 基础信息：系统版本、机型、进程名、应用版本号、磁盘空间等，如果有登录，则可以上报用户的ID。
* ·耗时信息：卡顿开始时间和结束时间。
* ·CPU信息：CPU的信息、本进程CPU使用率等。
* ·堆栈信息：发生卡顿的时间段内的堆栈数据，详细记录了该时间段内的系统信息，更容易定位问题，但会增加系统开销，所以这里可以抽样上报，只要有足够多的样本，就可以分析到具体原因。

### 代码实现

前面2.8.1节讲解了监控原理，本节将介绍具体的代码实现，以一个简单的卡顿监控为例，这个例子的主要功能如下：

* ·监控卡顿。
* ·抓取相关数据，这里只讲解采集CPU信息。
* ·日志保存/上传监控。

#### 卡顿监控

首先实现监控，先实现一个自定义的Printer，这里是LogPrinter，然后使用Looper.getMainLooper（）.setMessageLogging（LogPrinter），可以计算和监听耗时。LogPrinter类的代码如代码清单2-10所示。

从代码清单2-10可以看到，重写println（）方法可以拦截MainLooper中的dispatch事件前后时间差，若高于我们的最低级别时间，就进入execuTime（）方法，根据级别回调不同的监听方法，把处理权交到监控的管理层，在管理层做相应的处理。其中耗时级别在UiPerfMonitorConfig类中定义，这里只定义两个级别，可以根据需要定义更细的级别。定义耗时级别的代码如下

最低级别时间根据不同业务和场景由用户自己控制，一般来说1s就应该告警。通过以下代码可以启动和停止监控：

#### 获取相关数据

如果在发生卡顿后获取数据，数据的价值就不大，所以需要提前保存一定量的数据，在发生卡顿时把数据保存起来。这就需要一个线程来定时抓取需要的信息，推荐使用HandlerThread来实现。HandlerThread适合只需要一个工作线程（非UI线程）的情况，优点是不会堵塞，减少对性能的消耗。本节以采集一个CPU使用情况的信息为例，如果要实现更多数据的采集器，都可以通过这个采集框架来相应地扩展，这里先实现一个定时采样的类BaseSampler，具体实现如代码清单2-11所示。

从代码清单2-11可以看到，BaseSampler只是实现一个定时采样的类，intervalTime变量是采样时间间隔，但具体采集什么信息需要其子类来完成。在采样时调用了doSample（）虚方法，只要在子类中重写这个方法即可。接下来通过继承BaseSampler来实现一个采集CPU信息的功能类：CpuInfoSampler。

要采集CPU使用率，需要得到两个数据：整体CPU的使用率和当前应用进程的占用率。通过这两个数据，可以粗略判断是当前应用消耗CPU资源太多导致的卡顿，还是其他原因。然后通过获取总的CPU时间、进程的CPU时间、线程的CPU时间以及CPU的个数的信息，采样两个足够短的时间间隔的CPU快照与进程快照来计算进程的CPU使用率。可以得出以下几个数据：

* 总的CPU使用率。
* ·用户进程占比。
* ·系统进程占比。
* ·等待时间占比。
* ·当前App CPU使用率。

因此需要定义一个类CpuInfo来管理这些数据，

按以下步骤来获取这些数据。

1. 采样两个时间的CPU快照，分别记作c1、c2，其中CPU快照的数据结构是一个九元数组（user、nice、system、idle、iowait、irq、softirq、stealstolen、guest），把c1和c2中所有的CPU时间片求和得到t1、t2。
2. 2）得到总的CPU时间片totalTime=t2-t1、空闲时间idleTime=c2.idle-c1.id le、用户进程时间c1.user和c2.user、系统进程时间c1.system和c2.system、IO等待时间c1.ioWait和c2.ioWait。
3. 3）获取进程的总CPU时间appCpu\_time=utime+stime+cutime+cstime（该值包括其所有线程的CPU时间），同样采样两个时间的进程快照，进而计算得到appCpu\_time和mAppCpuTimePre。

各参数的计算方法如表2-5所示。



表2-5　各参数的计算公式

CpuInfoSampler继承于BaseSampler，实现了doSampler（）方法，doSample采样所需的数据，当然只有CPU信息是不能够定位到具体原因的，因些需要更多数据，特别是发生卡顿这段时间的堆栈信息。增加获取堆栈的信息可以和增加CPU信息一样，继承于BaseSampler实现doSample方法即可，这里就不一一实现。

#### 存储与上报

在拿到数据后，需要上报到后台，在后台通过大数据样本来定位发生卡顿最多的地方，但如果每次发生卡顿都上报（实时上报），就会增加很多系统开销，特别是本身就发生了卡顿的，更应该减少系统开销，因此可以先把信息存到本地，在合适的时机以及达到一定的量时，再压缩上报到服务器，供开发者分析。实现代码如代码清单2-13所示。

代码清单2-13实现了一个通过异步线程写日志到本地文件功能，其中上传到服务器的方法未实现，因后台差异，可根据自己的需求实现。

#### 监控管理类

至此，我们实现了一个卡顿监控各个子步骤，但需要一个管理类来管理这些功能，并将这些功能封装成一个监控器UiPerfMonitor。UiPerfMonitor是一个单例类，可以完整地实现了一个应用中的卡顿监控功能，代码如下：

该监控器的使用流程如下：

1. 初始化单例，构造方法UiPerMonitor初始化监控器LogPrinter、CPU采样器CpuInfo-Sampler和日志管理器LogWriteThread。
2. 2）通过startMonitor方法开始监控卡顿情况，监控过程通过回调方法onEndLoop（starttime，endtime，loginfo，level）实现，其中level为自定义级别，根据不同级别保存不同的堆栈信息。
3. 3）上传到服务器。

通过以上流程，可以从后台拿到大量卡顿数据，了解当前应用在外网用户使用过程中的实际卡顿情况，分析TOP数据，然后定位具体原因并改善，使应用使用更流畅。

本节介绍了一个卡顿监控的基本实现，在实际应用中，需要做更多的扩展，比如增加采样数据、完善日志管理系统，以便更有效地分析，在分析过程中，参考前面的章节讲解的优化方案对具体的模块进行优化。

## 本章小结

本章学习如何在UI绘制相关编程时，有效减少卡顿，提高应用的流畅度，优化分为以下几个过程：

* ·发现问题：除了在使用时明显体验到卡顿情况外，也可以通过卡顿监控来发现整体的耗时情况，或者打开开发者选项中的一些辅助工具来发现问题。
* ·分析问题：分析绘制和耗时的问题，前面提供了一系列的工具，可以使用Systrace和TraceView分析耗时，使用Hierarchy Viewer来分析页面层级。
* ·找到导致问题的原因：深入了解导致发生这些问题的根本原因。
* ·解决问题：在发现问题后，分析导致这些问题的原因，最后根据本章节的内容一一解决各种问题。

应用出现卡顿，除了绘制不合理和优化不合理以外，另一个影响应用使用流畅性的原因和内存有关，不合理使用内存除了会导致卡顿以外，对耗电和应用的稳定性也有很大的影响，下一章将详细讲解如何合理地使用内存。

# 内存优化

在内存管理上，和C/C++开发不同的是，Java虚拟机拥有垃圾内存回收的机制，在虚拟机层自动分配和释放内存，因此不需要在代码中分配和释放某一块内存，从应用层面上不容易出现内存泄漏和内存溢出等问题，开发者不用担心对象创建和销毁时产生的额外负担。

Android系统的内存管理也是如此，通过new关键字来为对象分配内存，内存的释放由垃圾收集器（GC）来回收。在开发的过程中，不需要显式地管理内存。同时Android系统在内存管理上有一个Generational Heap Memory模型，内存回收的大部分压力不需要应用层关心，Generational Heap Memory有一套自己管理的机制，当内存达到某一个阈值时，系统会根据不同的规则自动释放系统认为可以释放的内存，也正是因为Android程序把内存控制的权力交给了Generational Heap Memory，一旦出现内存泄漏和溢出方面的问题，排查错误将会成为一项异常艰难的工作。另一方面，内存的不合理使用也会造成一系列的性能问题，比如在短时间内分配大量的内存对象，即使没有内存泄漏但内存占用仍过高等。

正是因为系统提供了优秀的内存管理机制，部分Android应用开发人员在开发过程中并没有特别关注内存的合理使用，**也没有在内存方面做太多的优化，然而当应用程序同时运行越来越多的任务，加上越来越复杂的业务需求时，完全依赖Android的内存管理机制就会导致一系列性能问题逐渐显现，对应用的稳定性和性能带来不可忽视的影响，因此，解决内存问题和合理优化内存是非常有必要的**。

本章讲解Android内存管理机制，了解内存会导致哪些性能问题，掌握多个内存分析工具，优化应用程序中的内存使用，使应用更加稳定、流畅。

## Android内存管理机制

Android应用都是在Android的虚拟机上运行，应用程序的内存分配与垃圾回收都是由虚拟机完成的。在Android系统，虚拟机有两种运行模式：Dalvik和ART。下面学习Android内存管理机制，了解系统如何分配和回收内存。

### Java对象生命周期

在Android应用程序中，大部分是Java程序，而Java对象被创建前的Java类字节码（编译后的.class文件）需要从文件系统加载到虚拟机。Java对象在虚拟机上运行有7个阶段，也就是对象的生命周期，如图3-1所示。



图3-1　Java对象生命周期

#### 创建阶段（Created）

创建Java对象阶段的具体步骤如下：

1. 为对象分配存储空间。
2. 构造对象。
3. 从超类到子类对static成员进行初始化，类的static成员的初始化在ClassLoader加载该类时进行。
4. 超类成员变量按顺序初始化，递归调用超类的构造方法。
5. 子类成员变量按顺序初始化，一旦对象被创建，子类构造方法就调用该对象并为某些变量赋值，完成后这个对象的状态就切换到了应用阶段。

#### 应用阶段（InUse）

对象至少被一个强引用（Strong Reference）持有，除非在系统中显式地使用了软引用（Soft Reference）、弱引用（Weak Reference）或虚引用（Phantom Reference）。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　**软引用**可以加速虚拟机对垃圾内存的回收速度，更可靠地维护系统的运行安全，防止内存溢出（Out Of Memory）等问题产生。

#### 不可见阶段（Invisible）

处于不可见阶段的对象在虚拟机的对象引用根集合中再也找不到直接或间接地强引用，这些对象一般是所有线程栈中的临时变量。所有已经装载的静态变量或者是对本地代码接口的引用。

当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该对象仍然是存在的。简单的例子就是程序的执行已经超出了该对象的作用域了。该对象仍可能被虚拟机下的某些已装载的静态变量线程或JNI等强引用持有，这些特殊的强引用称为“GC Root”。存在这些GC root会导致对象的内存泄漏，无法被回收。

#### 不可达阶段（Unreachable）

对象处于不可达阶段是指该对象不再被任何强引用持有，回收器发现该对象已经不可达。

#### 收集阶段（Collected）

当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，对象进入“收集阶段”。如果该对象已经重写了finalize（）方法，则执行该方法的操作。

#### 终结阶段（Finalized）

当对象执行完finalize（）方法后仍然处于不可达状态时，该对象进入终结阶段。在该阶段，等待垃圾回收器回收该对象空间。

#### 7.对象空间重新分配阶段（Deallocated）

若垃圾回收器对该对象占用的内存空间进行回收或者再分配，则该对象彻底消失，这个阶段称为“对象空间重新分配阶段”。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　创建对象后，在确定不需要使用该对象时，使对象置空，这样更符合垃圾回收标准。比如Object=null，可以提高内存使用效率，并且不要采用过深的继承层次。访问本地变量优于访问类中的变量。

### 内存分配

在了解Android内存模型前，先要了解堆的概念，在Android系统中，堆实际上就是一块匿名共享内存。Android虚拟机并没有直接管理这块匿名共享内存，而是把它封装成一个mSpace，由底层C库来管理，并且仍然使用libc提供的函数malloc和free来分配和释放内存，当然也是在一个mSpace进行的。Android应用的进程都是从一个叫作Zygote的进程衍生出来的（调用fork方法）。系统启动Zygote进程后，为了启动一个新的应用程序进程，系统会衍生**Zygote进程**生成一个新的进程，然后在新的进程中加载并运行应用程序的代码。这就使得大多数的RAM pages被用来分配给framework的代码，同时促使RAM资源能够在应用的所有进程之间共享。

大多数静态数据被映射到一个进程中。这不仅使得同样的数据能够在进程间共享，而且使得它能够在需要时能共享使用。**常见的静态数据包括Dalvik Code、app resources、so文件等**。

在大多数情况下，Android通过显式分配共享内存区域（如ashmem或者gralloc）来实现动态RAM区域能够在不同进程之间共享的机制。例如，Window Surface在App与Screen Compositor之间使用共享的内存，Cursor Buffers在Content Provider与Clients之间共享内存。

为了整个系统的内存控制需要，Android系统为每一个应用程序都设置一个硬性的Dalvik Heap Size最大限制阈值，这个阈值在不同的设备上会因为RAM大小不同而有所差异。如果应用占用内存空间已经接近这个阈值时，再尝试分配内存的话，就很容易引发Out Of Memory Error错误。Android系统的内存堆被划分为不同的区块，根据对数据配置的类型分配不同的区域内存，垃圾回收时，也会根据这些配置执行不同的回收处理过程，并且每一个区块都有指定的单位大小。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　ActivityManager.getMemoryClass（）可以用来查询当前应用的Heap Size阈值，这个方法会返回一个整数，表明应用的Heap Size阈值是多少MB。

前面提到过，在Android Runtime（Android运行环境）有两种虚拟机，Dalvik和ART，它们分配的内存区域块是不同的，如图3-2所示。



图3-2　Android内存运行时堆

从图3-2可以看出，虚拟机对整个堆又划分出了不同的空间，这些空间用来存储不同的数据。

不论是在Dalvik虚拟机模式，还是在ART虚拟机模式，运行时堆都划分为三个空间：LinearAlloc、Zygote Space（Zygote Heap）和Allocation Space（Active Heap）。Dalvik中的LinearAlloc是一个线性内存空间，并且是一个只读区域，主要用来存储虚拟机中的类，因为类加载后只需要读的属性，并且不会改变它。把这些只读属性，以及在整个进程的生命周期都不能结束的永久数据放到线性分配器中管理，能很好地减少堆混乱和垃圾扫描，加快内存管理性能。

ART虚拟机Zygote Space和Allocation Space与Dalvik虚拟机中的Zygote Space和Allocation Space一样。Zygote Space在Zygote进程和应用程序进程之间共享，Allocation Space则是每个进程独占。Android系统的第一个虚拟机由Zyg ote进程创建并且只有一个Zygote Space。但是当Zygote进程在fork第一个应用程序进程之前，会将已经使用的那部分堆内存划分为一部分，还没有使用的堆内存划分为另外一部分，也就是Allocation Space。但无论是应用程序进程，还是Zygote进程，当它们需要分配对象时，都是在Allocation Space堆上进行。

在ART运行时，堆除了Zygote Space、Allocation Space，又多了两个空间，即Image Space和Large Object Space。其中Image Space用来存放一些预加载类，和Dalvik中的Linear Alloc类似，而Large Object Space是一些离散地址的集合，用来分配一些大对象，这样可以提高GC的管理效率和整体性能。其中Image Space和Zygote Space在Zygote进程和应用程序进程之间共享，而Allocation Space是每个进程都独立拥有一份。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　Image Space和Zygote Space都是在Zygote进程和应用程序进程之间共享，但是Image Space的对象只创建一次，而Zygote Space的对象需要在系统每次启动时，根据运行情况都重新创建一遍。

### 内存回收机制

在Android的高级系统版本中，针对Heap空间有一个Generational Heap Memory的模型，如图3-3所示，最近分配的对象会存放在Young Generation区域。对象在某个时机触发GC（Garbage Collection）回收垃圾，而没有回收的就根据不同规则，有可能被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。系统会根据内存中不同的内存数据类型分别执行不同的GC操作。GC通过确定对象是否被活动对象引用来确定是否收集该对象，进而动态回收无任何引用的对象占据的内存空间。

从图3-3中可以看到整个内存分为三个区域：Young Generation（年轻代）、Old Genera-tion（年老代）和Permanent Generation（持久代）。



图3-3　Generational Heap Memory内存模型

#### Young Generation

年轻代分为三个区，一个Eden区，另外的两个S0和S1都是Survivor区（S0和S1只是为了说明，两者实质上一样，方向可互换）。程序中生成的大部分新的对象都在Eden区中，当Eden区满时，还存活的对象将被复制到其中一个Survivor区，当此Survivor区的对象占用空间满时，此区存活的对象又被复制到另外一个Survivor区，当这个Survivor区也满时，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制到年老代。

#### Old Generation

年老代存放的是上面年轻代复制过来的对象，也就是在年轻代中还存活的对象，并且区满了复制过来的。一般来说，年老代中的对象生命周期都比较长。

#### Permanent Generation

用于存放静态的类和方法，持久代对垃圾回收没有显著影响。

内存对象的处理过程如下：

1）对象创建后在Eden区。

2）执行GC时，如果对象仍然存活，则复制到S0区。

3）当S0区满时，该区存活对象将复制到S1区，然后S0清空，接下来S0和S1角色互换。

4）当第3）步达到一定次数（系统版本不同会有差异）后，存活对象将被复制到Old Generation。

5）当这个对象在Old Generation区域停留的时间达到一定程度时，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。Permanent Generation区域也存放一些静态文件，如Java类等。



图3-4　Generational Heap Memory内存数据流转模型

整体数据流转模型如图3-4所示。

这样，系统在Young Generation、Old Generation上采用不同的回收机制。每一个Generation的内存区域都有固定的大小。随着新的对象陆续被分配到此区域，当对象总的大小临近这一级别内存区域的阀值时，会触发GC操作，以便腾出空间来存放其他新的对象。

执行GC占用的时间和它发生在哪一个Generation也有关系，Young Generation中的每次GC操作时间是最短的，Old Generation其次，Permanent Generation最长。同时GC的执行时间也和当前Generation中的对象数量有关，Generation中的对象数量越多，执行时间越长。

#### Young Generation GC

Young Generation通常存活时间较短，因此基于Copying算法来回收，所谓Copying算法，就是扫描出存活的对象，并复制到一块新的完全未使用的空间中，对应于Young Generation，就是在Eden、FromSpace或ToSpace之间copy。新生代采用空闲指针的方式来控制GC触发，指针保持最后一个分配的对象在Young Generation区间的位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。当连续分配对象时，对象会逐渐从Eden到Survivor，最后到Old Generation。

#### Old Generation GC

Old Generation与Young Generation不同，对象存活的时间比较长，比较稳定，因此采用标记（Mark）算法来回收。所谓标记，就是扫描出存活的对象，然后再回收未被标记的对象，回收后对空出的空间要么合并，要么标记出来便于下次分配，以减少内存碎片带来的效率损耗。

### GC类型

在Android系统中，GC有以下三种类型：

* ·kGcCauseForAlloc：在分配内存时发现内存不够的情况下引起的GC，这种情况下的GC会Stop World。Stop World是由于并发GC时，其他线程都会停止，直到GC完成。
* ·kGcCauseBackground：当内存达到一定的阈值时触发GC，这个时候是一个后台GC，不会引起Stop World。
* ·kGcCauseExplicit：显式调用时进行的GC，如果ART打开了这个选项，在system.gc时会进行GC。

下面来看一段虚拟机打印出来的日志：

D/dalvikvm( 7030): GC\_CONCURRENT freed **1049K**, 60% free 2341K/9351K, external 3502K/6261K, paused 3ms 3ms

GC\_CONCURRENT是当前GC时的类型，在Android的虚拟机中GC日志有以下几种类型：

* ·GC\_CONCURRENT：当应用进程中的Heap内存占用上涨时，避免因Heap内存满了而触发GC。
* ·GC\_FOR\_MALLOC：这是由于Concurrent GC没有及时执行完，而应用又需要分配更多的内存，这时不得不停下来进行Malloc GC。
* ·GC\_EXTERNAL\_ALLOC：这是为external分配的内存执行的GC。
* ·GC\_HPROF\_DUMP\_HEAP：创建一个HPROF profile的时候执行。
* ·GC\_EXPLICIT：显式地调用了System.gc（）。一般来说，可以信任系统的GC机制，尽量不去显式调用System.gc（），减少不必要的系统开销，影响应用的流畅度

再回到上面打印出的日志，其中：

freed 1049K表明在这次GC中回收了多少内存。

60%free 3571K/9991K是Heap的一些统计数据，表明这次回收后60%的Heap可用，存活的对象大小为2341KB，Heap大小是9351KB。

external 3502K/6261K是Native Memory的数据。存放位图数据（Bitmap Pixel Data）或者堆以外内存（NIO Direct Buffer）之类的。第一个数字表明Native Memory中已分配了多少内存，第二个值有点类似一个浮动的阈值，表明分配内存达到这个值，系统就会触发一次GC进行内存回收。

paused 3ms 3ms表明GC暂停的时间。从这里可以看到，越大的Heap Size在GC时导致暂停的时间越长。如果是Concurrent GC，会看到两个时间：一个开始，一个结束，且时间很短，但如果是其他类型的GC，很可能只会看到一个时间，且这个时间是相对比较长的。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　在ART模式下日志多了一个Large Object Space（大对象占用的空间），这部分内存并不是分配在堆上的，但仍属于应用程序内存空间，主要用来管理Bitmap等占内存大的对象，避免因分配大内存导致堆频繁GC。

在Dalvik虚拟机下，GC的操作都是并发的，也就意味着每次触发GC都会导致其他线程暂停工作（包括UI线程）。而在ART模式下，在GC时，不像Dalvik仅有一种回收算法，ART在不同的情况下会选择不同的回收算法，比如Alloc内存不够时会采用非并发GC，但在Alloc后，发现内存达到一定阈值时又会触发并发GC。所以在ART模式下，并不是所有的GC都是非并发的。

总体来看，在GC方面，与比Dalvik相比，ART更为高效，不仅仅是GC的效率，大大地缩短了Pause时间，而且在内存分配上对大内存分配单独的区域，还能有算法在后台做内存整理，减少内存碎片。因此在ART虚拟机下，可以避免较多的类似GC导致的卡顿问题

## 优化内存的意义

内存优化对应用程序有哪些具体的意义？对应用有哪些性能方面的提高？在内存优化前先抛出以下4个问题：

* ·系统在内存收回（GC）时对性能会造成什么影响？
* ·明明有一定剩余内存，为什么有时会产生OOM（Out Of Memory）导致程序崩溃？
* ·为什么发生OOM是概率性的，并且每次在执行不同的代码时产生OOM？
* ·内存占用高对应用的存活有什么影响？
* 开发的实际主要会遇到哪些情况下的oom

接下来带着这4个问题，具体分析在应用开发过程中内存导致的问题以及在Android系统内存回收过程中，对性能会产生哪些影响。

在使用C和C++时，应用开发者在自己代码中管理内存，也就是说在写代码时，在程序中从系统申请内存，在不使用时释放这些内存，因此所有的内存对于应用来说都是可控的，然而当代码量很大时，会因为细节未处理好，结果并没有按意愿释放已经申请的内存，导致内存泄漏。而在Android（包括其他Java虚拟机）上，申请内存和释放内存这个工作在系统层承担，自动分配对象所需内存，然后虚拟机会追踪每个内存对象，一旦决定哪个对象已经不再使用了，就会释放到内存堆中，这个过程不需要应用开发者去干涉。这种内存再生的机制，在Android系统有一个垃圾内存回收器，也就是GC。

应用在使用过程中随着内存增加，当到达一定的条件后，GC开始工作，为新的对象释放出更多的内存，但是在GC的过程中，任何其他在工作的线程会暂停，包括负责绘制的UI线程（在Google I/O的发布会上将这种情况称为世界终结者），并且在不同区域的内存释放速度也有一定的差异，但不管在哪个区域，都是要到这次GC内存回收完成后，才会继续执行原来的线程。在图3-5所示的情况下，开发者并没有在乎和注意到这一点，因为仅增加一点耗时并不会影响应用的使用。



图3-5　不影响UI线程的GC情况

但是，如果大量这样的重复，其他事情处理时间就被压缩了，进而会影响到应用的渲染工作，造成垃圾回收动作太频繁。这种情况很容易发生在短时间内申请大量的对象时，并且它们在极少的情况下能得到有效的释放，这样会出现内存泄漏的情况。一旦达到了剩余内存的阈值，垃圾回收活动就会启动。即使有时内存申请很小，它们仍然会给应用程序的堆内存造成压力，还是会启动垃圾回收，在GC频繁的工作过程中消耗了非常多的时间，并且可能导致卡顿。为了避免这样的情况，设置一个16ms界线，只要GC消耗的时间超过了16ms的阈值，就会有丢帧的情况出现（原因见第2章），并可能会导致卡顿，如图3-6所示。



图3-6　GC导致丢帧

**由此得出结论：频繁的GC会增加应用的卡顿情况，影响应用的流畅性，因此，需要尽量减少系统GC行为，以便提高应用的流畅度，减小卡顿发生的概率。**

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　在ART模式中，虽然减少了大部分的GC停顿，但还是会在部分垃圾回收出现小的停顿。虽然系统也会让垃圾回收以最快速度进行，以免中断其他工作，但还是会导致应用出现性能问题。

虽然系统的GC机制经过Android版本的发展，已经越来越智能、高效，但如果内存使用不合理，系统仍然不能保证内存保持在充足的阶段，这也是不管什么版本的系统仍然会出现OOM情况的原因。如果内存在某一阶段的峰值到达了内存空间的阈值，或者频繁地发生这种内存峰值（毛刺现象），刚好在这个峰值时，需要申请一块较大的内存，就会由于堆内存空间不足而导致OOM异常。总地来说，导致OOM的主要原因是申请的内存仍然不能满足应用程序这次需要的内存，如图3-7所示。



图3-7　内存不足的情况

而另一方面，由于在Android应用开发过程中，虽然开发者不用担心对象创建和销毁时产生的额外负担，但是大家都忽略了一个问题，即使Android Runtime会管理内存，内存泄漏的问题还是会存在，内存泄漏是复杂应用程序中常见的问题，如果处理不好会导致很严重的性能问题。内存泄漏是指应用已经不会再使用的内存对象，但垃圾回收时没有把这些辨认出来，不能及时地回收，仍然一直保留在内存中，占用了一定的空间，并且最终会到GC耗时最长的Old Generation（年老代），不释放给其他对象，因此当更多的内存泄漏时（也可能是频繁地运行了导致内存泄漏这块的程序），系统为应用分配可用堆上的空间就会不断变小，这会导致不断启动垃圾回收（在Logcat上可以看到系统不停地打印出GC日志），以便释放空间用于执行其他程序。但是这时大部分情况下并不有效，因为这些内存泄漏的对象都无法被释放，所以也会出现OOM的异常。

**出现OOM是因为内存溢出导致，这种情况不一定会发生在相同的代码，也不一定是出现OOM的代码使用内存有问题，而是刚好执行到这段代码**。找到并解决这些问题是很困难的，首先需要清晰理解你的代码，确认代码库的关键活动的每一项工作都在正常进行。为了确认这些类型的行为是正常的，需要一些详细的检查。

除了导致OOM和UI卡顿，在应用程序后台运行时，特别是服务进程，即使通过一系列的方法提高进程的优先级，但如果内存占用过高，在系统资源紧张的情况下，仍然会被系统Kill。

通过上面的分析，可以总结出内存优化主要有以下几个意义：

* ·减少OOM，提高应用稳定性。
* ·减少卡顿，提高应用流畅度。
* ·减少内存占用，提高应用后台运行时的存活率。
* ·减少异常发生，减少代码逻辑隐患。

要达到这些目标，根据本节和3.1节的分析，主要目标综合起来就是避免内存泄漏，提高内存效率，节省对象占用内存空间。这些目标实现起来并不容易，需要借助一系列的辅助和分析工具，接下来介绍这一系列工具的使用方法。

## 内存分析工具

做内存优化前，需要了解当前应用的内存使用现状，通过现状去分析哪些数据类型有问题，各种类型的分布情况如何，以及在发现问题后如何发现是哪些具体对象导致的。非常幸运的是，Android官方在内存分析上提供了一系列非常强大的工具。下面将一一介绍这些工具的作用和使用方法。

### Memory Monitor

Memory Monitor是一款使用非常简单的图形化工具，可以很好地监控系统或应用的内存使用情况，主要有以下几个功能：

* ·显示可用和已用内存，并且以时间为维度实时反应内存分配和回收情况。
* ·快速判断应用程序的运行缓慢是否是由于过度的内存回收导致。
* ·快速判断应用是否是由于内存不足导致程序崩溃。

通过观察以时间为维度实时反应内存分配和回收情况，可以快速发现内存抖动、大内存分配，甚至由于CG导致卡顿。

#### Memory Monitor的使用介绍

在使用Memory Monitor前，需要确认手机是否打开了开发者模式，并且打开了USB调试模式，确定后，根据以下步骤使用Memory Monitor。

1. 在Android Studio上运行需要监控的应用。
2. 2）从Android Studio菜单栏中选择Tools->Android->Memory Monitor。或者单击Android Studio应用程序面板右下角的Android图标，直接运行Memory Monitor。
3. 3）一旦Memory Monitor开始运行，图形就开始实时显示当前内存使用情况，如图3-8所示。这时就可以观察内存的实时使用情况了。



图3-8　Memory Monitor视图

图3-8是Memory Monitor的视图结构，在第一行，分别选择需要监控的设备和应用。左边的一列按钮（从上到下）的功能如下：

* Initiate GC（见①）：手动触发GC。
* ·Dump Java heap（见②）：保存内存快照。
* ·Start/Stop Allocation Tracking（见③）：打开Allocation Tracker工具。

在内存显示区域可以看到有深蓝和浅蓝两种的区域，其中深蓝表示当前App使用的内存大小，浅蓝为可用的未分配内存大小。

#### 典型场景

使用Memory Monitor可以观察到多个不同的场景，接下来介绍两个基本场景在Memory Monitor的视图情况。

1. **内存分配与释放**

如图3-9所示，在第一段触发了系统GC，迅速释放了一大块内存，在第二段，进入新的界面，同时申请了新的内存。通过Memory Monitor可以清楚地发现这种大内存分配的场景，定位观察，以判断是否为合理分配内存，是Bitmap还是其他数据，然后对数据进行优化，减小内存开销。



图3-9　内存释放与申请

1. **大内存申请与内存抖动**

内存抖动，一般是指在很短的时间内发生了多次的内存分配和释放，并且通常在发生严重内存抖动时，也能感觉到应用卡顿，可以看出来是由于执行了GC操作造成的。频繁的GC会导致界面卡顿，具体原因可参见3.2节。在Memory Monitor上，内存抖动也很容易发现。如图3-10所示，内存的分配过程有明显的锯齿状，带有毛剌，在这种情况下要仔细分析内容的分配情况，可以使用另外两个分析工具来查看分配了什么对象、以前分配内存的大小和分析对象的堆栈，从而结合代码进行优化。



图3-10　内存抖动

### Heap Viewer

Heap Viewer的主要功能是查看不同数据类型在内存中的使用情况，可以看到当前进程中的Heap Size的情况，分别有哪些类型的数据，以及各种类型数据占比情况。通过分析这些数据来找到大的内存对象，再进一步分析这些大对象，进而通过优化减少内存开销，也可以通过数据的变化发现内存泄漏。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　Heap Viewer目前只能由5.0及以上系统支持，并且需要开发者模式。

#### Heap Viewer启动

Heap Viewer在Android Device Monitor工具中，Android Device Monitor可以从快捷工具栏上打开，或者选择Tools->Android->Android Device Monitor命令。



图3-11　启动Heap Viewer

进入Android Device Monitor面板后，在进程列表中选择需要查看的应用进程，单击Update Heap按钮，在右边的Heap Viewer开始更新数据，右边面板中的数值会在每次GC时改变，包括应用自动触发或者在面板上手动触发。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　Heap Viewer不仅可以用来检测是否有内存泄漏，还可以检测到内存抖动，因为内存抖动时，会频繁发生GC，这时只需要开启Heap Viewer，观察内存数据的变化，如果发生内存抖动，就可以观察到数据在这段时间内频繁更新。

#### .Heap Viewer面板

Heap Viewer面板总共有三个区域，如图3-12所示。

（1）总览页（A区域）

查看整体内存的使用情况，包括已使用和未使用内存的占比。

2）详情页（B区域）

在详情页可以看到各种类型数据的内存开销。

（3）具体类型内存分配柱状图（C区域）

可以查看选中类型的内存分析情况。



图3-12　Heap Viewer面板

在总览页和详情页，可以分别通过总览视图和详情视图了解更多关于内存的情况。

·总览视图

通过总览视图可以看到所有的内存情况，单击总览视图上最右边的Cause GC按钮可以手动触发GC，表中数值的对应关系如下：

·Heap Size：堆栈分配给App的内存大小。

·Allocated：已分配使用的内存大小。

·Free：空闲的内存大小。

·%Used：Allocated/Heap Size，使用率。

·#Objects：对象数量。

·详情视图

详情视图页展示所有数据类型的内存情况，其中第一列为所有数据类型，类型如下：

·free：空闲的对象。

·data object：数据对象，Java类类型对象，是最主要的观察对象。

·class object：Java类类型的引用对象。

·1-byte array（byte[]，boolean[]）：一字节的数组对象。

·2-byte array（short[]，char[]）：两字节的数组对象。

·4-byte array（long[]，double[]）：4字节的数组对象。

·8-byte array（long[]，double[]）：8字节的数组对象。

·non-Java object：非Java对象。

在每个类型的数据值对应如下：

·Count：数量。

·Total Size：总共占用的内存大小。

·Smallest：将对象占用内存从小到大排列，排在第一个的对象占用内存大小。

·Largest：将对象占用内存的大小从小到大排列，排在最后一个的对象占用的内存大小。

·Median：将对象占用内存的大小从小到大排列，排在中间的对象占用的内存大小。

·Average：平均值。

选择一个具体数据类型后，会显示对应的内存对象柱状图，柱状图横坐标是对象的内存大小，这些值随着不同对象而不同，纵坐标是在某个内存大小上的对象的数量。

### Allocation Tracker

Memory Monitor和Heap Viewer都可以很直观且实时地监控内存使用情况，还能发现内存问题，但发现内存问题后不能再进一步找到原因，或者发现一块异常内存，但不能区别是否正常，同时在发现问题后，也不能定位到具体的类或方法。这时就需要使用另一个内存分析工具Allocation Tracker进行更详细的分析，Allocation Tracker可以分配跟踪记录应用程序的内存分配，并列出了它们的调用堆栈，可以查看所有对象内存分配的周期。

Allocation Tracker的主要功能如下：

* ·在一段时间内以对象类型为纬度，跟踪在此时间内的内存分配和释放情况。
* ·寻找代码中内存使用不合理的地方。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　Allocation Tracker是分析较短一段时间内的内存使用情况，在使用Allocation Tracker前，可以先用Memory Monitor或者Heap Viewer找到内存异常的场景，然后使用Allocation Tracker分析这个场景的内存使用情况。

#### Allocation Tracker的使用

Allocation Tracker在Android Studio和Eclipse上都支持，在Android Studio上使用Allocation Tracker界面更加清晰并更有条理，但两个IDE上使用Allocation Tracker的功能都是相同的。运行应用后，切换到Android选项卡（和Memory Monitor是同一个视图）。

1）单击启动追踪按钮（Start Allocation Tracking）。

2）操作应用，怀疑有内存泄漏或者内存变化较大的操作。

3）单击结束追踪按钮，与启动追踪按钮是同一个位置，如图3-13所示。



图3-13　启动/停止Allocation Tracke

1. 自动生成一个alloc结尾的文件，这个文件记录了这次追踪到的所有内存数据，并且在Android Studio中自动打开一个数据面板，显示当前生成alloc文件的内存数据，如图3-14所示。



图3-14　Allocation Tracker面板

#### .查看面板信息

面板信息如图3-14所示，整个面板分成两个区域，上面是内存对象列表，下面是对象引用堆栈。

（1）内存对象列表

内存对象列表每一列字段的定义如表3-1所示。



表3-1　Allocation Tracker对象列表字段

（2）对象引用堆栈

在内存对象列表中，选中某一个对象后，在下面的窗口显示调用堆栈，单击具体的堆栈可以进入具体的代码行。

Allocation Tracker可以非常方便快捷地得到程序在某一段时间内内存的分配情况，并且能跟踪到分配该内存的具体代码。但是它不能提供任何程序Heap的总体情况和Heap具体信息。而分析Heap的信息，能够查找出更隐蔽的内存泄漏问题，下一节将讲述查找内存泄漏的过程，并介绍一个更强大的内存分析工具：MAT。

## 避免内存泄漏

在Android应用开发过程中，开发者不用担心对象创建和销毁时的内存如何管理这一额外负担，但并不意味着不会产生任何问题，即使Android Runtime高效并且智能地管理内存，仍然会有内存泄漏的情况存在。内存泄漏是复杂应用程序中比较常见的现象，开发者在应用开发阶段更重视功能需求的开发，却忽略了内存是否合理使用以及使用完如何处理，进而导致在不知不觉产生了内存泄漏的情况。

内存泄漏是指应用不再使用的内存对象，但垃圾回收时没有把这些辨认出来，不能及时回收，一直保留在内存中长期占用一定的空间，再也不释放给其他对象。在这种情况下，如果不去很好地处理，就会导致很严重的性能问题。当泄漏点越多，或者频繁地执行了这段内存泄漏的代码时，会导致更多的内存泄漏，这样可用堆上的空间会不断变小，不断地触发启动垃圾回收，以便释放空间用于执行其他程序。而GC的频繁工作是导致卡顿的罪魁祸首（原因见3.2节），更坏的情况是GC有时并不有效，因为产生泄漏的对象一直都无法被释放，最终会耗尽内存导致OOM的情况。因此内存泄漏是必须避免的。

本节通过了解内存泄漏产生的原因，在开发过程中写出优质的代码尽量避免发生内存泄漏，并且通过一系列的工具和方法查找出内存泄漏的具体位置并修复，使应用程序更加高效健壮地运行。

### 内存泄漏定义

通过3.1.1节的学习，可以知道Java对象有自己的生命周期，当这个对象不需要再使用时，应该完整地走完生命周期，但因为某些原因，对象虽然已经不再使用，仍然在内存中并没有结束整个生命周期，这就意味着这个对象已经泄漏。

Android系统虚拟机的垃圾回收是通过虚拟机GC机制来实现的。GC会选择一些还存活的对象作为内存遍历的根节点GC Roots，通过对GC Roots的可达性来判断是否需要回收。GC Roots是系统选择的对象根节点，比如Thread Stack中的变量Zygote中的Class Loader对象等，对Heap进行遍历，没有被直接或间接遍历到的引用会被GC回收，能被遍历到的不能被回收，如图3-15所示。这类在当前应用周期内不再使用的对象被GC Roots引用，导致不能回收，使实际可使用内存变小，这种现象在Android应用中称为内存泄漏。



图3-15　GC Roots判断对象是否可以回收

图3-15是一组对象的引用关系，其中A、B、C、D四个对象直接或间接被GC Roots引用链相连，这类对象被认为是还需要使用的对象，将不会回收，而E、F、G三个对象未被GC Roots直接或间接引用，即使E、F两个对象互相有引用，但相对GC Roots来说仍然不可达，所以E、F、G三个对象可以被回收掉。此时如果C对象在实际中已经不需要使用，但因为仍然对GC Roots可达，就说明对象C已经内存泄漏。

大部分内存泄漏的现象不是很明显，比如Context、Handler相关的使用导致的内存泄漏，或者引用第三方库，由于单位时间内消耗资源并不多，一般是不会引起开发者的足够注意去手动释放它们。在应用代码达到一定的复杂度时，如果结构不是很清晰，即使是很大的内存对象没有及时释放，在开发过程中也有可能忽视了，从而导致内存泄漏。这种泄漏并不好找到，因此需要借助一些工具来查找内存泄漏。

### 使用MAT查找内存泄漏

对于大型Java应用程序来说，再精细的测试也难以堵住所有的漏洞，即便在测试阶段进行了大量卓有成效的工作，很多问题还是会在生产环境下暴露出来，并且很难在测试环境中重现。在没有发现或者并不知道哪有内存泄漏的情况下，可以使用在3.3节提到的一系列内存分析工具去分析。通常情况下，可以使用Heap View粗略查看堆的使用情况，又或者使用Allocation Tracker跟踪内存分配情况，当发现内存持续上涨并没有释放时，说明有内存泄漏的可能性，这时再深入分析这个场景的内存情况。Android虚拟机能够记录下问题发生时，系统的部分运行状态和内存使用情况，并将其存储在堆转储（Heap Dump）文件中，而这个文件为开发者分析和诊断问题提供了重要的依据。

抓取这个疑似有内存问题的使用场景的Heap信息，然后进行分析，目前来看最好的分析Java Heap工具就是MAT。

Memory Analyzer Tool（MAT）是一个快速、功能丰富的Java Heap分析工具，通过分析Java进程的内存快照HPROF文件，从众多的对象中分析，快速计算出在内存中对象的占用大小，查看哪些对象不能被垃圾收集器回收，并可以通过视图直观地查看可能造成这种结果的对象。

通常大家理解的内存泄漏分析被认为是一件很有难度的工作，一般由团队中的资深工程师负责。不过，在这里要介绍的MAT（Eclipse Memory Analyzer）被认为是一个“傻瓜式”的堆转储文件分析工具，和其他内存泄漏分析工具相比，MAT的使用非常容易，基本使用几次就能理解，即使是新手，也能够很快上手使用。

MAT工具可以帮助开发者定位导致内存泄漏的对象，以及发现大的内存对象，然后解决内存泄漏并通过优化内存对象，达到减少内存消耗的目的。

1.使用步骤

在Eclipse上使用MAT比较简单，只需要安装相应的Eclipse插件，安装方法和安装其他插件相同，在线安装MAT插件地址为：http://download.eclipse.org/mat/1.3/update-site/。

相比Eclipse上可以直接集成MAT插件，并且能自动转换成视图来分析，Android Studio的使用流程要略显复杂些，因为Android Studio并没有集成MAT工具，所以需要下载MAT独立客户端，使用步骤如下：

1）下载MAT客户端。

下载地址为https://eclipse.org/mat/downloads.php，在这里可以下载不同的版本。

2）获取HPROF文件。

从Android Studio进入Android Device Monitor（DDMS），选择需要分析的应用进程，单击Update Heap按钮，对应用进行怀疑有内存问题的操作，也可以整体操作一段时间，结束操作后，多主动进行几次GC，最后单击Dump HPROF File按钮，保存HPROF文件，如图3-16所示。



图3-16　获取HPROF文件

3) 因为Android Studio保存的是Android Dalvik格式.hprof文件，所以需要转换成J2SE HPROF格式才能被MAT识别和分析。Android SDK自带一个转换工具hprof-conv，转换语句如下：

./hprof-conv path/file.hprof exitPath/heap-converted.hprof

其中path为转换前的文件路径，exitPath为转换后文件的路径。

4）通过MAT工具打开转换后的HPROF文件，进入分析界面。

在Android Studio1.2以上版本中获取HPROF文件和转换有更快捷的方式是在Memory Monitor工具中，单击Dump Java Heap按钮，在左侧的Capture栏中的Heap Snapshot列表中看到Dump下来的HPROF文件，右击文件，在弹出的菜单中选择Export to standard.hprof选项，即可转换成标准的HPROF文件，再使用MAT打开，如图3-17所示。



图3-17　Memory Monitor保存HPROF文件

2.MAT视图

通过前一节的学习，我们已经可以拿到HPROF文件，使用MAT打开HPROF文件后，可以看到MAT的分析内存视图，如图3-18所示。



图3-18　MAT视图

在MAT窗口上，OverView是一个总体概览，显示总体的内存消耗情况和疑似问题。MAT提供了多种分析维度，其中Histogram、Dominator Tree、Top Consumers和Leak Suspects的分析纬度不同，如图3-19所示。

分析内存最常用的是Histogram和Dominator Tree两个视图，这两个视图的区别是统计的纬度不一样，但使用Dominator Tree可以更方便地看出其引用关系。下面介绍几个常用的分析数据视图和报告。



图3-19　MAT提供的四种Action

（1）Histogram

列出内存中的所有实例类型对象、对象的个数以及大小，并支持正则表达式查找。

（2）Dominator Tree

列出最大的对象及其依赖存活的Object。分析流程和Histogram大同小异，但Dominator Tree能更方便地看出引用关系。

（3）Top Consumers

通过图形列出最大的Object。

（4）Leak Suspects

通过MAT自动分析泄漏的原因和泄漏的一份总体报告。Leak Suspects列出了工具怀疑的内存泄漏点，以及泄漏的内存大小，在后面有问题列表和所有对象，单击对应的<Details>可以看到更深入的分析情况，如图3-20所示。



图3-20　Leak Suspects分析示意图

一般都在Histogram或者Dominator Tree视图中分析内存是否异常，如图3-21所示。一共有四列：Class Name、Objects、Shallow Heap和Retained Heap。

从图3-21可以看到具体的列表，其中Class Name是类名，Objects是对象实例个数，另外有两个很少见过的名词Shallow Heap和Retained Heap。



图3-21　Hisogram视图

·Shallow Heap：对象自身占用的内存大小，不包括它引用的对象。非数组的常规对象的Shallow Heap Size由其成员变量的数量和类型决定，数组的Shallo常规对象的Shallow Heap Size由其成员变量的数量和类型决定，数组的Shallow Heap Size由数组元素的类型（对象类型、基本类型）和数组长度决定。真正的内存都在堆上，看起来是一堆原生的byte[]、char[]、int[]，对象本身的内存都很小。因此Shallow Heap对分析内存泄漏意义不是很大。

·Retained Heap：是当前对象大小与当前对象可直接或间接引用到的对象的大小总和，包括被递归释放的。也可以理解为，Retained Size就是当前对象被GC后，从Heap上总共能释放掉的内存大小。

3.查找内存泄漏具体位置

为了了解查找泄漏的过程，制造一个内存泄漏，在Activity中把一个TextView一直让一个单例执有，这样这个对象就会一直得不到释放。根据前面的介绍，拿到HPROF文件使用MAT打开，进入Dominator Tree视图。

右击选中实例类型android.widget.TextView，选择Merge Shortest Paths to GC Root→exclude all phantom/weak/soft etc refereneces.如图3-22所示。



图3-22　查找内存泄漏可疑点

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　直接按照包名类型显示查看GC引用链，可以一次性筛选多个类。

Merge Shortest Paths to GC Roots主要显示距GC Root最短的路径。通过3.1.3节的介绍我们知道Java的垃圾回收机制，如果一个对象有引用，则GC Roots到这个对象之间是有路径可达的。只有GC Roots不可达时，才能被系统回收。exclude all phantom/weak/soft etc references则是排除虚引用、弱引用及软引用，因为这些引用一般是可以回收的。进入结果页，如图3-23所示。



图3-23　Merge Shortest Paths to GC Root结果页

其中可以看到TextView对象有三个实例，并且都被TestDataModel这个单例引用，TestDataModel的mRetaindeTextView引用的三个TextView实例都不能被释放，因此可以结合代码去解决。简单的办法就是将其中的某处置为null或者remove掉，使其到GC Root无路径可达，即处于不可触及状态，这样垃圾回收器就可以回收这个不需要再使用的TextView对象。

这个例子很简单，所以很容易找到泄漏的对象，但这种分析方法并不容易确定所有的内存泄漏场景，还需要通过代码的逻辑找出原因。另一种更快速的方法就是对比HPROF数据。通过对比两个Dump结果来发现内存泄漏的对象，步骤如下：

1）进入需要分析的应用页面，按前面介绍的抓取HPROF的方法拿到第一个HPROF。

2）进行一段时间的操作，再抓取第二个HPROF文件。

3）使用MAT打开两个HPROF文件。

4）在两个HPROF文件中，把Histogram或者Dominator Tree增加到Compare Basket，如图3-24所示。



图3-24　增加到对比视图

1. 在Compare Basket中单击对比按钮[图https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=53a84b3a1fe92be0536c28f987b80cab.jpg片]，生成对比结果视图，如图3-25所示。这样就可以对比相同的对象在不同阶段的对象实例个数和内存占用大小，如明显只需要一个实例的对象，或者不应该增加的对象实例个数却增加了，说明发生了内存泄漏，就需要去代码中定位具体的原因并解决。



图3-25　MAT对比两个dump

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　当目标不太明确时，可以直接定位到RetainedHeap最大的Object，通过Select incoming references查看引用链，定位到可疑的对象，然后通过Path to GC Roots分析引用链。

MAT的功能非常强大，本节只介绍了基本的使用，另外还有一个检查引起性能问题的功能：Map Collision Ratio，当Hash集合中过多的对象返回相同Hash值时，会严重影响性能（Hash算法原理自行搜索），这时可以用Map Collision Ratio查找导致Hash集合的碰撞率较高的罪魁祸首。

另外大家在使用过程中会慢慢积累，发现更多实用的小技巧，这里就不过多介绍。

### 常见内存泄漏场景

如果在内存泄漏发生后再去找原因并修复会增加开发的成本，最好是在编写代码时就能够很好地考虑到内存问题，写出更高质量的代码，这里列出一些常见的内存泄漏场景，在以后的开发过程中需要避免这类问题。

**1.资源性对象未关闭**

资源性对象（比如Cursor、File文件等）往往都用了一些缓冲，在不使用时，应该及时关闭它们，以便它们的缓存数据能够及时回收。它们的缓存不只存在于Java虚拟机内，还存在于Java虚拟机外。如果仅仅是把它的引用设置为null，而不关闭它们，往往会造成内存泄漏。因为有些资源性对象，比如SQLite Cursor（在析构函数finalize（）中，如果没有关闭它，它自己会调用close（）关闭），如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用时，应该立即调用它的close（）函数，将其关闭，然后再置为null。在程序退出时，一定要确保资源性对象已经关闭。

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。如果查询结果集比较小，对内存的消耗不容易被发现，只有在长时间大量操作的情况下，才会复现内存问题，这样会给以后的测试和问题排查带来困难和风险。

因此，在编写资源文件读写时，都需要在finally中关闭资源性对象，避免在异常情况下资源对象未被释放的隐患。

**2.注册对象未注销**

如果事件注册后未注销，会导致观察者列表中维持着对象的引用，阻止垃圾回收，一般发生在**注册广播接收器**、注册观察者等。

假设在Activity中，监听系统中的电话服务，以获取一些信息（如信号强度等），可以在Activity中定义一个PhoneStateListener的对象，同时将它注册到TelephonyManager服务中。对于Activity对象，理论上要求Activity在退出后该Activity对象就会被释放掉。

但是如果在释放Activity对象时，忘记取消之前注册的PhoneStateListener对象，则会导致Activity无法被GC回收。如果不断地进出这个Activity，则最终会由于大量的Activity对象没有办法被回收而引起频繁的GC情况，甚至导致Out Of Memory。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　虽然有些系统程序本身好像是可以自动取消注册的（不一定及时），但是还是应该在程序中明确取消注册，程序结束时，应该取消所有的注册。

**3.类的静态变量持有大数据对象**

静态变量长期维持对象的引用，阻止垃圾回收，如果静态变量持有大的数据对象，如Bitmap等，就很容易引起内存不足等问题。

**4.非静态内部类的静态实例**

非静态内部类会维持一个到外部类实例的引用，如果非静态内部类的实例是静态的，就会间接长期维持着外部类的引用，阻止被系统回收。如下代码是一个非静态内部类的实现。

在Activity内部创建了一个非静态内部类的静态实例mTestModule，每次启动Activity时都会使用mTestModule的数据，这样虽然避免了资源的重复创建，但是TestModule为非静态内部类默认持有外部类的引用，而这里又使用该非静态内部类创建了一个静态的实例，该实例的生命周期和应用一样长，这就导致该静态实例一直持有该Activity的引用，**Activity的内存资源不能正常回收。**

可以这样避免内存泄漏，即将内部类TestModule设为静态内部类或将内部类TestModule抽取出来封装成一个单例，如果需要使用Context，就在没有特殊要求的情况下；**Application Context**，如果需要使用Activity Context，就记得用完后**置空**让GC可以回收，否则还是会内存泄漏。

**5.Handler临时性内存泄漏**

Handler通过发送Message与主线程交互是应用开发中非常常见的使用场景之一，Message发出之后存储在MessageQueue中，有些Message也不是马上就被处理到。在Message中存在一个target，它是Handler的一个引用，Message在Queue中存在的时间过长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。Handler发送Message与主线程交互的实现代码如下：

由于mHandler是Handler的非静态匿名内部类的实例，所以它持有外部类Activity的引用，并且消息队列是在一个Looper线程中不断轮询处理消息，那就有一种情况，当这个Activity退出时，消息队列中还有未处理的消息或者正在处理的消息，并且消息队列中的Message持有mHandler实例的引用，mHandler又持有Activity的引用，所以导致该Activity的内存资源无法及时回收，引发内存泄漏。避免这种内存泄漏需要修改以下两个地方：

1）使用一个静态Handler内部类，然后对Handler持有的对象使用弱引用，这样在回收时，也可以回收Handler持有的对象。

2）在Activity的Destroy或者Stop时，应该移除消息队列中的消息，避免Looper线程的消息队列中有待处理的消息需要处理。

修改后的代码如下。

|  |
| --- |
| private static class NewHandler extends Handler{  private WeakReference<Context> mContext = null;  public NewHandler(Context ctx){  mContext = new WeakReference<Context>(ctx);  }  @Override  public void handleMessage(Message msg) {  super.handleMessage(msg);  }  }  private void doGetDataAsyncTask(){  Message msg = Message.obtain();  mHandler.sendMessage(msg);  }  @Override  protected void onDestroy() {  super.onDestroy();  mHandler.removeCallbacksAndMessages(null);  } |

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　由于AsyncTask内部也是Handler机制，同样存在内存泄漏的风险。但这种内存泄漏一般是临时性的。

**6.容器中的对象没清理造成的内存泄漏**

通常把一些对象的引用加入集合中，在不需要该对象时，如果没有把它的引用从集合中清理掉，这个集合就会越来越大。如果这个集合是static，情况就更严重。

**7.WebView**

大家一定要注意到，Android中的WebView不仅仅存在很大的兼容性问题，不同Android系统版本中的WebView会有较大的差异，加上不同厂商定制的ROM中的WebView也存在着差异。更严重的是WebView都存在内存泄漏的问题，在应用中只要使用一次Webview，内存就不会被释放掉。通常解决这个问题的办法是为**WebView开启独立的一个进程**，使用AIDL与应用的主进程进行通信，WebView所在的进程可以根据业务的需要选择合适的时机进行销毁，达到正常释放内存的目的。

### 内存泄漏监控

从前面我们知道分析和查找内存泄漏的过程很繁琐，还不一定能排查所有的内存泄漏，因此有必要使用一个工具来监控全局，特别是高风险模块的内存是否泄漏。

LeakCanary是一个检测内存泄漏的开源类库，使用非常简单，可以在发生内存泄漏时告警，并且生成leak trace分析泄漏位置，同时可以提供Dump文件，LeakCanary实际上就是在本机上自动做了Heap Dump，然后分析生成的hprof文件（和手工分析MAT步骤基本一致），展示结果。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　有时仅仅通过leak trace不一定能找到问题更深层次的原因，因此也有必要使用MAT手动分析hprof文件。

#### 实现监控

首先需要导入LeakCanary的SDK，建议通过远程导入第三方Jar包的方法，在build.gradle文件（需要使用的Module，非全局的build.gradle）中加入相关引用：

|  |
| --- |
| debugCompile 'com.squareup.leakcanary:leakcanary-android:1.4-beta2'  releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.4-beta2'  testCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.4-beta2' |

LeakCanary对应用的性能有一定的影响，特别是Heap Dump操作会消耗更多的系统资源，并且有可能引起卡顿现象。对此，LeakCanary也已经考虑到这个问题，因此在增加LeakCanary依赖库时有三个配置可以选择：debugCompile、releaseCompile和testCompile，其中releaseCompile和testCompile这两个的依赖明显不同于debugCompile的依赖。它们的依赖属于NOOP（No Operation Performed）操作。编译器技术会检测无操作指令并出于优化的目的将无操作指令剔除。所以使用LeakCanary只要配置好releaseCompile和testCompile的依赖，就不需要担心对正式版本性能产生影响。

引入相关依赖后，在应用的自定义Application中安装LeakCanary，代码如下：

|  |
| --- |
| @Override  public void onCreate() {  super.onCreate();  mRefWatcher = LeakCanary.install(this);  …  } |

LeakCanary.install() 会返回一个预定义的 RefWatcher，同时也会启用一个 ActivityRefWatcher，用于自动监控调用Activity.onDestroy() 之后泄露的 activity。其实现原理是设置Application的ActivityLifecycleCallbacks方法监控所有的Activity的生命周期回调。

出现内存泄漏时会有Toast弹出提示，在Leaks App中可以看到所有的内存泄漏记录，同时会在Logcat中打印出如下日志：

默认情况下，只对Activity进行了检测。另一个需要监控的重要对象就是Fragment实例或者其他自定义的UI容器窗口组件，因为它和Activity实例一样，可能持有大量的视图以及视图需要的资源（如Bitmap），需要监控这一类组件，可以在Fragment onDestroy方法中，或者自定义组件的周期结束回调接口加入如下实现：

KApplication.getRefWatcher().watch(this);

除了视图类的，其他模块也可以监控的对象如下。

* ·BroadcastReceiver
* ·Service
* ·其他有生命周期的对象
* ·持有大内存占用的对象（即Retained Heap值比较大的对象）

从3.4.1章节可以知道，内存泄漏是某个对象在该释放时被其他对象持有没有被释放而造成的。因此，监控也需要设置在对象被释放时，如Activity和Fragment的onDestroy方法。如果放在一个Activity的onCreate方法中，那么每次都会收到Activity的泄漏通知。

#### 自定义处理结果

仅仅依靠默认的监控结果处理方式，在实际的开发和测试过程中体验不是很好，必须依赖安装的Leaks应用，并且不好定制化，因此，实现自定义的监控结果处理非常有必要。

首先，继承DisplayLeakService实现一个自定义的监控处理Service，代码如下：

|  |
| --- |
| public class LeakCanaryService extends DisplayLeakService{  private final String TAG = "LeakCanaryService";  @Override  protected void afterDefaultHandling(HeapDump heapDump, AnalysisResult result, String leakInfo) {  KLogUtil.D(TAG,"afterDefaultHandling:" + leakInfo);  super.afterDefaultHandling(heapDump, result, leakInfo);  }  } |

重写afterDefaultHanding方法，在其中处理需要的数据，afterDefaultHanding回调方法有三个参数，定义如下：

·heapDump：堆内存文件，可以拿到完成的hprof文件，可以用MAT分析。

·leakInfo：监控到内存的状态，如是否泄漏等。

·leakInfo：leak trace详细信息，除了内存泄漏对象，还有设备信息。

然后在intall时，使用自定义的LeakCanaryService，代码如下：

ref = LeakCanary.install(this, LeakCanaryService.class, AndroidExcludedRefs.createAppDefaults().build());

这样，就可以在LeakCanaryService中实现自己的处理方式，如丰富提示信息、把数据保存到本地、上传到服务器进行分析。



https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　LeakCanaryService需要在AndroidManifest中注册。

## 优化内存空间

没有内存泄漏，并不意味着内存就不需要优化了，在移动设备上，由于物理设备的存储空间有限，Android系统对每个应用进程也都分配了有限的堆内存备的存储空间有限，Android系统对每个应用进程也都分配了有限的堆内存，因此使用最小内存的对象或者资源可以减小内存开销，同时让GC能更高效地回收不再需要使用的对象，让应用堆内存保持充足的可用内存，使应用更稳定高效地运行。

### 对象引用

从Java1.2版本开始引入了三种对象引用方式：软引用（SoftReference）、弱引用（WeakReference）和虚引用（PhantomReference）三个引用类，引用类的主要功能就是能够引用但仍可以被垃圾收集器回收的对象。在引入引用类之前，只能使用强引用（Strong Reference），如果没有指定对象引用类型，默认是强引用，如下代码段就产生了一个强引用对象：Object A = new Object();Object B = A;

在不妨碍内存收集的情况下，软引用、弱引用和虚引用对象提供了三种方式来引用堆对象。每种引用对象都有不同的行为，而且它们与垃圾收集器之间的交互也有所不同。需要说明的是，内存中的一个对象可以被多个引用（可以是强引用、软引用、弱引用或虚引用）引用。

#### 强引用

强引用是使用最普遍的引用。如果一个对象具有强引用，垃圾回收器（GC）就绝不会回收它。当内存空间不足时，Java虚拟机会抛出OutOfMemoryError错误，不会回收具有强引用的对象来解决内存不足的问题。因此，如果是强引用的对象，在应用的生命周期中如果不再需要使用，一定要记得释放或转成弱引用，以便让系统回收。

#### 软引用

软引用在保持引用对象的同时，保证在虚拟机报告内存不足的情况之前，清除所有的软引用。关键之处在于，垃圾收集器在运行时可能会（也可能不会）释放软引用对象。对象是否被释放取决于垃圾收集器的算法以及垃圾收集器运行时可用的内存数量。

如果一个对象只具有软引用，则内存空间足够，GC时就不会回收它；如果内存空间不足，就会回收这些对象的内存，并且在没有回收该对象时，该对象可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用引用的对象被垃圾回收器回收，Java虚拟机会把这个软引用加入与之关联的引用队列中。

#### .弱引用

弱引用类的一个典型用途就是规范化映射（canonicalized mapping）。另外，对于那些生存期相对较长，而且重新创建的开销也不高的对象来说，弱引用也比较有用。关键之处在于，垃圾收集器运行时如果碰到了弱可及对象，将释放WeakReference引用的对象。

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，**不管当前内存空间足够与否**，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　可能需要运行多次GC，才能找到并释放弱引用对象。

#### 虚引用

虚引用类只能用于跟踪即将对被引用对象进行的收集。同样，它还能用于执行pre-mortem清除操作。虚引用必须与ReferenceQueue类联合使用。需要ReferenceQueue是因为它能够充当通知机制。当垃圾回收器确定某个对象是虚可及对象时，PhantomReference对象就被放在它的ReferenceQueue上。将Phantom Reference对象放在ReferenceQueue上会有一个通知，表明虚引用对象引用的对象已经结束，可供收集了。这使用户能够刚好在对象占用的内存被回收之前采取行动。

虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，它就和没有任何引用一样，在任何时候都可能被垃圾收集器回收。

从以上我们知道不同引用类型，在GC时的策略是不同的，因此根据业务需求合理使用不同引用，以提高内存的使用效率。

### 减少不必要的内存开销

#### AutoBoxing

在性能优化时，有些小细节也需要注意，小问题积累多了就是一个大问题，和内存泄漏一样，都是积少成多后才导致了严重的性能问题。

Java语言中有一个AutoBoxing（自动装箱）功能，我们知道Java提供了基本数据类型，如boolean（8 bits）、int（32 bits）、float（32 bits）、long（64 bits）和复杂数据类型（通过一个类描述），如Boolean、Integer、Float。为了能够让这些基础数据类型在大多数Java容器中运作，需要做一个AutoBoxing的操作，把这些基础类型转换成Boolean、Integer、Float，如表3-2所示。



表3-2　基础数据类型与复杂数据类型对应关系

这些复杂数据类型提供了与基本数据类型相通的功能，但可以使用于泛型集合，在程序运行过程中，这种自动转换的过程就是自动装箱的过程，自动装箱的核心就是把基础数据类型换成对应的复杂类型，不需要开发者自己转换。自动装箱的典型情况如图3-26所示。



图3-26　自动装箱的典型情况

因此在编程时不需要关心这种转换，Java会自动把整数类型换成复杂类型，但如果代码中出现以下情况：

|  |
| --- |
| Integer test2 = 0;  for (int i = 0;i<1000\*1000;i++){  test2 += i;  } |

Int 消耗64ms，Integer 1000ms！！！

这就意味着要消耗更多的性能，每次循环虚拟机都必须创建一个新的整数对象，并把它加到其他整数对象前面，创建一个新的整数对象。在自动装箱转化时，都会产生一个新的对象，这样就会产生更多的内存和性能开销，因为这些对象比基础数据类型对象还要大。基础整数（int）对象只有4字节，而Integer对象有16字节，因此在内存和时间性能上都有额外的开销。

因此，在写程序时需要避免这类情况。特别是HashMap这类的容器，基本上只要使用这个设计基元的容器，进行增、删、改、查操作时，都会产生大量的自动装箱操作，为了避免这些AutoBoxing带来的效率问题，Android特地提供了一些Map容器用来替代HashMap，不仅减少了时间开销，还减少了内存占用。

对于不必要的内存开销等问题应该及早发现，不要让问题积少成多。一般可以通过TraceView查看耗时，如果发现调用了大量的integer.value，就说明发生了AutoBoxing，需要立即优化代码。

#### 内存复用

在Android系统中，有些数据是可以复用的，并且在开发过程中，系统也提供了可利用的接口或方法。比如以下三种复用的场景，通过内存复用可以有效减少应用的内存开销。

（1）有效利用系统自带的资源

Android系统本身内置了大量的资源，比如一些通用的字符串、颜色定义、常用Icon图片，还有些动画和页面的样式以及简单布局，如果没有特别的要求，这些资源都可以在应用程序中直接引用。

直接使用系统资源不仅可以在一定程度上减少内存的开销，还可以减少应用程序的自身负重，减小APK的大小，并且复用性更好。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　需要留意Android系统版本的差异性，对于那些不同系统版本上的表现存在很大差异，不符合需求的情况，还是需要应用程序自身实现。

（2）视图复用

出现大量重复子组件，而子组件是大量重复的，可以使用ViewHolder实现ConvertView复用，这基本上是所有容器控件的处理方式，如ListView、GridView等。

（3）对象池

可以在设计程序时显式地在程序中创建对象池，然后实现复用逻辑，对相同的类型数据使用同一块内存空间，也可以利用系统框架既有的具有复用特性的组件减少对象的重复创建，从而减少内存的分配与回收。

（4）Bitmap对象的复用

利用Bitmap中的inBitmap的高级特性，提高Android系统在Bitmap的分配与释放效率，不仅可以达到内存复用，还提高了读写速度。使用inBitmap属性可以告知Bitmap解码器尝试使用已经存在的内存区域，新解码的bitmap会尝试使用之前那张bitmap在heap中占据的pixel data内存区域，而不是向内存重新申请一块区域来存放bitmap。具体的使用方法会在后面具体讲到。

### 使用最优的数据类型

Android应用开发，大部分使用的是Java语言编程，在Java中有很多的数据结构和类型，但不一定都是最省内存的数据结构，需要根据实际的开发需求，选择最好的数据类型或类类型。在Android上也意识到这一类情况，因此Android针对移动开发提出了一系列的数据类容器结构优化。

#### HashMap与ArrayMap

HashMap是Android应用开发过程中经常使用的容器之一，当然它是非常有用的，但它并不是最节约的容器，会占用大量内存。一个典型HashMap的工作原理如图3-27所示。



图3-27　HashMap模型

从图3-27可以看出，HashMap是一个散列链表，向HashMap中put元素时，先根据key的HashCode重新计算hash值，根据hash值得到这个元素在数组中的位置，如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。也就是说，向HashMap插入一个对象前，会给一个通向Hash阵列的索引，在索引的位置中，保存了这个Key对象的值。这意味着需要考虑的一个最大问题是冲突，当多个对象散列于阵列相同位置时，就会有散列冲突的问题。因此，HashMap会配置一个大的数组来减少潜在的冲突，并且会有其他的逻辑防止链接算法和一些冲突的发生。

很明显，这个向稀疏阵列填入对象的大阵列，从内存节省的角度来看，是非常不理想的。因此，Android为了解决HashMap的这个问题，提供一个替代容器：ArrayMap。

ArrayMap提供了和HashMap一样的功能，但避免了过多的内存开销，方法是使用两个小数组，而不是一个大数组。其中一个数组记录对象Key Hash过后的顺序列表，另外一个数组按Key的顺序记录Key-Value值，根据Key数组的顺序，交织在一起，如图3-28所示。

在需要获取某个Value时，ArrayMap会计算输入Key转换过后的hash值，然后使用二分查找法对Hash数组寻找到对应的index，然后可以通过这个index在另外一个数组中直接访问需要的键值对。如果在第二个数组键值对中的key和前面输入的查询key不一致，就认为发生了碰撞冲突。为了解决这个问题，ArrayMap会以该key为中心点，分别上下展开，逐个对比查找，直到找到匹配的值。因此会带来一个问题，就是随着ArrayMap中对象数量的增加，需要访问单独对象的时间也会变长。

从上面的描述我们知道，ArrayMap在内存上是连续不间断的，那么在删除和插入时要如何处理呢？删除一个元素时，把将要删除的元素放到最后，其他元素向前移，或者是把所有元素的重置大小和一个副本，来删除想要的值。插入时需要重新配置数组，添加完成后移动所有元素来保证ArrayMap选择的顺序。删除和插入的示意图如图3-29所示



图3-28　ArrayMap数据模型



图3-29　ArrayMap删除与插入一个对象

总地来说，在ArrayMap中执行插入或者删除操作时，从性能角度上看，比HashMap还要更差一些，但如果只涉及很小的对象数，比如1000以下，就不需要担心这个问题了。因为小的连续阵列，当值特别小时，相比HashMap，ArrayMap能节省更多的内存，对于空的映射，没有什么配置会占用空间。HashMap和ArrayMap内存占用对比如图3-30所示。



图3-30　HashMap和ArrayMap内存占用对比

与HashMap相比，ArrayMap在循环遍历（顺序遍历）时也更加简单高效，从如下代码中可以看出HashMap和ArrayMap之间的差异。

因此，可以总结出在以下两个场景考虑优先使用Array-Map：

* ·当对象的数目非常小（1000以内），但是访问特别多，或者删除和插入频率不高时。
* ·当有映射容器，有映射发生，并且所有映射的容器也是ArrayMap时。

#### 枚举类型

JDK 1.5版本开始支持枚举类型，枚举类型采用关键字enum来定义，所有的枚举类型都是继承自enum类型，使用枚举来实现。如果使用枚举类型（Enums）来定义常量，会使代码更易读并且更安全，但在性能上却比普通常量定义差很多。我们知道Android系统在应用启动后，会给应用单独分配一块内存。应用的dex code、Heap以及运行时的内存分配都会在这块内存中。而使用枚举类型的dex size是普通常量定义的dex size的13倍以上（只是dex code增加），同时，运行时的内存分配，一个enum值的声明会消耗至少20 bytes，这还没有算上其中的对象数组需要保持对enum值的引用。每个枚举项都会被声明成一个静态变量，并被赋值。因此，当应用程序中的代码或包含的Lib中大量使用enum时，对本身内存小的手机会带来不可忽视的影响。下面看下如何普通定义一个常量，并且达到类型安全的效果。

首先通过以下方法定义常量，代码如下：



从以上代码可以看出，对于getLevel（int level）中的参数level是不安全的，如果传入的值不是0或1，就会抛出异常，也就是说对输入的参数没有约束，会导致上层的业务带来一个处理异常的逻辑，增加了代码的不安全因素和维护成本。

而枚举可以避免此问题，下面通过枚举类型来实现相同的代码逻辑。可以看出，输入参数level已经约束在枚举类型PER\_LEVEL中，这样就不用再做容错处理。

枚举的最大优点是类型安全，但在Android平台上，枚举的内存开销是直接定义常量的三倍以上，所以Android的官方文档也提醒了开发者尽量避免使用枚举类型，同时提供注解的方式检查类型安全，目前提供了int型和String型两种注解方式：IntDef和StringDef，用来提供编译期的类型检查，以下代码通过IntDef来实现。

|  |
| --- |
| public static final int UI\_PERF\_LEVEL\_0 = 0;  public static final int UI\_PERF\_LEVEL\_1 = 2;  @IntDef({UI\_PERF\_LEVEL\_0,UI\_PERF\_LEVEL\_1})  @Retention(RetentionPolicy.SOURCE)  public @interface PERF\_LEVEL {  }  public int getLevel(@PERF\_LEVEL int level){ |

这里可以看到，@PER\_LEVEL注解放到getLevel方法中，对参数的数据类型进行限制。如果在使用getLevel时直接赋值一个Int参数，IDE会报错，@PER\_LEVEL也可以放到返回值等其他整型属性的地方对参数进行限制，通过这个方法既保证了类型安全，又不会给内存带来额外的开销。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　使用IntDef和StringDef需要在Gralde配置中引入相应的依赖包：compile'com.android.support：support-annotations：22.0.0'。

#### LruCache

LruCache在android.util包下（android-support-v4的包中提供），可以翻译为最近最少使用缓存，它用强引用保存需要缓存的对象，它内部维护一个队列（实际上是LinkedHashMap内部的双向链表，不支持线程安全，LruCache对它进行封装，添加了线程安全操作），当其中的一个值被访问时，它被放到队列的尾部，当缓存将满时，队列头部的值（也就是最近最少被访问的）被丢弃，之后可以被垃圾回收。

LruCache比较重要的几个方法如下。

·public final V get（K key）

返回cache中key对应的值，调用这个方法后，被访问的值会移动到队列的尾部。

·public final V put（K key，V value）

根据key存放value，存放的value会移动到队列的尾部。

·protected int sizeOf（K key，V value）

返回每个缓存对象的大小，用来判断缓存是否快要满了，这个方法必须重写。

·protected void entryRemoved（boolean evicted，K key，V oldValue，V newValue）

当一个缓存对象被丢弃时调用的方法，这是个空方法，可以重写，不是必需的，第一个参数为true：当缓存对象是为了腾出空间而被清理时（trimToSize时）。第一个参数为false：缓存对象的entry被remove移除或者被put覆盖时。

Android的官网也一直推荐使用LruCache作为图片内存缓存，里面保存了一定数量的对象强引用。在使用时需要注意LruCache的容量，这个容量既不能太大，太大会造成其他可用内存变小，容易导致OOM，但又不能太小，否则起不到缓存的作用。Google官方文档给出以下意见作为参考：

分配LruCache大小时考虑应用剩余内存有多大。

·一次屏幕显示多少张图片，有多少张图片是缓存起来准备显示的。

·考虑设备的分辨率和尺寸，缓存相同的图片数，手机的dpi越大，需要的内存也越大。

·图片分辨率和像素质量决定了占用内存的大小。

图片访问的频繁程度是多少，如果存在多个不同要求的图片类型，可以考虑用多个LruCache来做缓存，按照访问的频率分配到不同的LruCache中。

总之分配的LruCache既不能太大，又不能太小，具体在应用中还要综合考虑。

### 图片内存优化

在应用中有很多图片需要显示，这种图片可以是多种格式，如JPEG、PNG或者WEBP，这种图片文件并不是很大，为了使文件变小，以提高传输速度，都要经过不同的压缩算法把位图压缩成不同的格式，但要在Android中显示图片，就和图片文件的大小无关了，在Android设备上显示前需要把图片解码成位图格式，而位图格式只和位图的属性相关，压缩格式仅仅是减小了文件大小。

因此，图片占用的内存在整个应用中是非常突出的，特别是Dalvik版本虚拟机在GC后不能压缩内存，也就是说，当对象被释放时，留下的空间不是连续的。如果需要显示更大的一张图片，虽然剩余内存大于图片所需的内存，但由于剩余内存并不是连续的，仍然会导致位图的内存不够触发GC，从其他位置释放空间。如图3-31所示。

前面讲过，Android中的一张图片解码成位图格式后，占用的内存只和位图的质量和大小相关，位图的主要属性有：图片长度、图片宽度、单位像素占用的字节数。一张图片（BitMap）占用的内存=图片长度×图片宽度×单位像素占用的字节数（图片长度和图片宽度的单位是像素）。如果从应用资源中加载（Drawable），图片占用的内存也应该和屏幕密度有一定关系。

在Android默认情况下，当图片文件解码成位图时，会被处理成32bit/像素。红色、绿色、蓝色和透明通道各8bit，即使是没有透明通道的图片，如JEPG格式是没有透明通道的，但然后会处理成32bit位图，这样分配的32bit中的8bit透明通道数据是没有任何用处的，这完全没有必要，并且在这些图片被屏幕渲染之前，它们首先要被作为纹理传送到GPU，这意味着每一张图片会同时占用CPU内存和GPU内存。接下来介绍减少图片内存开销的常用技术。

#### 设置位图规格

Android提供了多种位图格式，根据开发者或产品需求，可以选择不同的格式解码图片，安卓支持的像素格式如图3-32所示。

最高的是RGB\_8888，也就是系统默认的位图格式，其他几种都减小了位图通道位，可以减少内存开销并提升图片显示的性能。

当然，这种空间的节省是要付出视觉质量受损的代价的，比如从RGB\_8888改成使用RGB\_565，会损失较多的图片数据，但不是不能用，根据不同场景可以选择不同的规格。



图3-31　内存不够时申请模型



图3-32　Android支持的位图格式

除了大图模式，一般都可以使用，并且几乎看不出差别。只要满足以下其中一点就可以考虑使用RGB\_565：

·显示局部图片，比如列表中的小图片。

·小屏幕手机或者对图片质量要求不高的场景，可以使用RGB\_565，但实际上是，根据应用开发经验，不需要Alpha通道。

如果需要更小的格式，但又需要透明通道，可以尝试ARGB\_4444图像格式。它减少了一半的数据，但保留了透明通道，视觉差异变化较大，一般用于用户头像，特别是圆角的头像，可以尝试使用ARGB\_4444看看效果。

Aplha\_8格式主要用于Alpha通道模板，相当做一个染色。图像要渲染两次，虽然减少内存，但增加了绘制的开销。

值得注意的是，在Android的基本文件结构中不支持PNG、JPEG和WEBP格式，因此，需要解码后的位图功能，通过设置inPreferredConfig参数来实现不同的位图规格，代码如下：

BitmapFactory.Options options = new BitmapFactory.Options();

options.inPreferredConfig = Bitmap.Config.RGB\_565;

设置位图规格并不是减少内存的唯一方法，特别是在现在的应用中，图片场景非常多，为了更高效地显示图片，Android提供了一系列的接口来减少图片对堆内存空间的占用。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　图片加载完成时，系统会加载数据，并转换好。但还要注意，系统这样做时，会花费额外的处理时间，但一般是可以忽略的。

#### inSampleSize

如果内存中的图片大于屏幕显示出的图片大小，或者大于指定屏幕区域的大小，这些高分辨率图片会导致严重的性能问题。因此，需要改变内存的占用，避免此类问题。这里的问题在于，占据内存中实际未使用的区块，这些图片占用了内存堆中的大量空间，使应用空间变少，然后重置这些图片大小，让它们符合实际显示的大小，既能减小内存的开销，也能提高显示的效率，这样载入内存的图片规格符合实际显示规格，而不是完整的分辨率。

位图功能对象中的inSampleSize属性实现了位图的缩放功能，代码如下：

options.inSampleSize = findBestSampleSize(options.outWidth, options.outHeight, cardWidth, cardHeight);

将这个属性设置为1时，可以在不加载完整大小图片的前提下，生成一张只有原始图片部分大小的新图片，如inSampleSize为2时获得只有1/2大小的图片，同理设置为4就是1/4大小的图片。因此，图片大小总会比原始图片小一倍以上。

inSampleSize是一个很便捷的操作。它会根据设置的数值，每隔相应像素读入一次。

#### inScaled，inDensity和inTargetDensity

虽然inSampleSize可以实现图片的缩放，都是指数幂的缩放，如果想更细地缩放图片，就需要使用位图的inScaled、inDensity和inTargetDensity功能。代码如下：

BitmapFactory.Options options = new BitmapFactory.Options();

options.inJustDecodeBounds = true;

options.inDensity = DisplayMetrics.DENSITY\_DEFAULT;

options.inTargetDensity = DisplayMetrics.DENSITY\_DEFAULT;

options.inScreenDensity = DisplayMetrics.DENSITY\_DEFAULT;

BitmapFactory.decodeResource(getResources(), imageResId, options);

当inScaled设置为true时，系统会按照现有的密度来划分目标密度，通过派生绽放数来应用到位图上，使用这个方法会重设图片大小，并对它应用一个新的过滤。

虽然这些方法都非常好用，并且减少图片显示需要的内存，但因为过多的算法，导致图片显示的过程需要更多的时间开销，如果图片很多的话，就影响到图片的显示效果。最好的方案是结合这两个方法，达到最佳的性能结合，首先使用insamplsesie处理图片，转换为接近目标的2次幂，然后用inDensity和inTargetdensy生成最终想要的准确大小，因为inSamplesize会减少像素的数量，而基于输出密码的需要对像素重新过滤。但获取资源图片的大小，需要设置位图对象的inJustDecodeBounds值为true，然后继续解码图片文件，这样才能生成图片的宽高数据，并允许继续优化图片。整体代码如下：

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　有几个非常好的开源库，如picasso库，提供了更多、更优秀的处理位图的方式，并且实现了异步加载图片。

#### inBitmap

Android 3.0（API Level 11）引进了BitmapFactory.Options.inBitmap字段，如果设置了该属性，那么当使用了带有该Options参数的decode方法加载内容时，decode方法会尝试重用一个已经存在的位图。这就意味着位图内存已经被重用了，从而改善了性能，并且没有内存的分配和释放过程。

常见的使用方案可以结合LruCache来实现，在LruCache移除超出cache size的图片时，暂时缓存Bitmap到一个软引用集合，需要创建新的Bitmap时，可以从这个软引用集合中找到最适合重用的Bitmap，来重用它的内存区域。

Android 2.3.3（API level 10），以及更低的版本由于还不支持inBitmap内存复用，推荐使用Bitmap.recycle方法加快Bitmap内存的回收。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　新申请的Bitmap与旧的Bitmap必须有相同的解码格式，并且在Android 4.4之前，只能重用相同大小的Bitmap的内存区域，而Android 4.4之后可以重用任何bitmap的内存区域。

合理地处理图片，降低内存的占用并提高显示速度，在应用中非常重要，但通常来讲一个应用使用图片的地方非常多，特别是很多的ListView或者GridView等需要显示大量图片的容器，如果都各自管理图片的显示，很难保证整体的质量，并且对总内存占用不可控。因此，在应用中需要一个图片管理模块，把所有的图片加载和管理收到一个模块中管理，并对此管理模块整体优化，使应用更加流畅和稳定。下一节，通过设计一个简单的图片管理模块，了解优化图片显示的流程和一个管理模块的设计思路。

## 图片管理模块设计与实现

本节通过图片管理模块的设计与实现，熟悉内存优化的整体过程，图片管理目前也有很多成熟的开源框架，如UIL、Imageloader等，功能非常强大，在内存方面也做得很出色，可以灵活地扩展功能等。接下来通过MiniImageDownloader，实现在线加载图片的功能。

在设计一个模块时，需要考虑以下几点：

1.单一职责

也就是这个模块具有功能的单一性，不要把太多的功能都加到同一个模块中，比如要实现一个图片显示模块，就只是显示图片的功能，不要再增加其他功能进去。

2.避免不同功能之间的耦合

高层模块不应该依赖底层模块。两者都应该依赖其抽象，而抽象不应该依赖细节（具体实现），细节应该依赖抽象。

3.接口隔离

整个模块中不同类间的依赖关系应该建立在最小的接口上。不管具体实现的子类拥有多少其他特性，但是基本的还是可以通过父类或者接口来确定，在不需要或者不关心子类特有的方法属性时，把它作为接口或者父类使用。而在当作接口使用时，又不用担心其他特有的方法被暴露出去，只需要暴露出需要暴露的方法/接口，其他特有的方法调用者不需要了解。

在编写代码前先画好UML图，确定每一个对象、方法、接口的功能，首先尽量做到功能单一原则，在这个基础上，再明确模块与模块的直接关系，最后使用代码实现。

### 实现异步加载功能

在设计一个模块时，为了使模块更利于维护和扩展，可读性高，在类的设计过程中，开发者应该遵守单一职责原则（Single Responsibility Principle，SRP），也就是对于一个类而言，仅有一个引起它变化的原因。但如何划分一个类和一个函数的职责，不同的开发者有不同的看法，与开发者的水平以及业务的逻辑都有些不同的定义，但最基本的是两个完全不同的功能肯定不能放到一个类中，因此工程师需要合理设计，在编写代码过程中多加思考，合理拆分，是优化架构的第一步。

#### 实现网络图片显示

下面实现从网络加载图片，网络加载是一个耗时的操作，肯定要在异步线程中加载，避免在主线程做太多的等待（在第二章详细解释了为什么在主线程中执行耗时操作会导致卡顿），这里通过AsyncTask实现一个异步的线程池，并发加载图片并显示。异步加载图片的UML图如图3-33所示。



图3-33　异步加载图片UML图

结构很简单，ImageLoader是实现图片加载的基类，其中ImageLoader有一个内部类BitMapLoadTask是继承AsyncTask的异步下载管理类，负责图片的下载和刷新，MiniImageLoader是ImageLoader的子类，维护了一个ImageLoader的单例，并且实现了基类的网络加载功能，因为具体的下载在应用中有不同的下载引擎，抽像成接口便于替换。首先来看ImageLoader的代码，如代码清单3-1所示(见随书代码)。

ImageLoader中的内部类BitmapLoadTask继承了AsyncTask类，AsyncTask类是一个多线程管理和异步线程的接口实现，具体的原理这里不过多介绍，网上有很多介绍文档，下面介绍BitmapLoadTask重写的几个方法的具体作用。

（1）BitmapLoadTask（String url，ImageView imageView）

构造函数，参数为图片URL和对应的imageView。

（2）Bitmap doInBackground（Void...params）

此方法在异步子线程中运行，所有的耗时任务都应该放到这个方法中执行。最后通过return语句来返回任务的执行结果，如果AsyncTask的第三个泛型参数指定的是Void，就可以不返回任务执行结果。

在这里通过构造函数传入的URL获取图片数据并解码成Bitmap，然后返回bitmap数据再进入AsyncTask的下一个周期onPostExecute。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　在这个方法中是不可以进行UI操作的，如果需要更新UI元素，如反馈当前任务的执行进度，则可以调用publishProgress（Progress...）方法来完成。

（3）onPostExecute（Bitmap result）

当后台任务执行完毕并通过return语句返回时，这个方法很快被调用。也就是说，doInBackground返回的Bitmap数据会作为参数传递到此方法中，通过方法设置到构造函数传入的ImageView中显示。

（4）onCancelled（Bitmap value）

任务取消后回调此方法。BitmapLoadTask只在ImageLoader内部执行，对于加载图片的业务层来说，不需要关心具体的实现，对业务层，只开放了三个接口：

（5）loadImage（String url，ImageView imageView）

加载图片方法，只需要传入图片的网络地址和显示这张图片的imageView，即完成了一张图片的显示。

（6）setPauseWork（boolean pauseWork）

图片加载线程控制接口，pauseWork控制图片模块的暂停和继续工作，一般用在listView等容器类控件中，在ListView滑动时，为了减少系统开销，可以暂停线程的工作来保证滑动的流畅度。

另外，具体的图片下载和解码是和业务强相关的，因此在ImageLoader中不做具体实现，只是定义了一个抽象方法protected abstract Bitmap downLoadBitmap（String url）。在doInBackground异步线程中调用此方法实现图片下载与解码，方法的具体实现在ImageLoader的子类中，这里ImageLoader的子类是MiniImageLoader。

MiniImageLoader是一个单例，保证一个应用只维护一个ImageLoader，减少对象的开销，并管理应用中所有的图片加载，继承于ImageLoader，实现了ImageLoader中的虚方法downLoadBitmap（），如代码清单3-2所示。

代码清单3-2　MiniImageLoader.java

MiniImageLoader实现了一个单例，单例模式是应用最广泛的模式之一，单例模式的定义是一个类在应用的整个生命周期中只有一个实例，自动实例化并向整个应用提供这个实例。如果一个应用的整个应用生命周期只需要一个对象就可以实现单例模式，这里ImageLoader随着功能的增加，越来越庞大，加上为了对图片全局的管理，单例是最好的选择。

可以看到MiniImageLoader中的getInstance方法，为了确保单例模式避免出现多个实例的情况，需要注意getInstance方法的调用，通常是对getInstance加方法锁，但这对性能有一定的影响。getInstance方法是单例类中调用最频繁的方法，加方法锁会导致异步线程调用排队的情况，因此，这里把方法锁改成区域对象锁，只在创建实现时加锁，但并不能完全避免，所以增加了volatile关键字，用来保证MiniImageLoader对象只在主内存中加载。这种实现单例的方式叫双重锁模式（Double Check Lock），双重锁模式既能在有需要时，才实例化对象，又能保证线程安全。

downLoadBitmap方法是具体化了基类，也就是ImageLoader类中的虚方法，实现了具体的图片下载和解码，过程很简单，都是系统的API，但有一点可以优化，我们看到，在downLoadBitmap方法中，数据流读完后，都需要关闭资源对象，代码如下：

这段代码的可读性很差，try..catch中是非常简单的代码，虽然对功能和性能没有影响，但由于可读性差，层级多，很容易把代码写到错误的层级中。在Java中有一个Closeable接口，该接口标识了一个可关闭的对象，由此，可以写一个静态的工具类，用来统一关闭资源，代码如下：

|  |
| --- |
| public class CloseUtil {  public static void closeQuietly(Closeable closeable){  if(null != closeable){  try{  closeable.close();  }catch (IOException e){  e.printStackTrace();  }  }  }  } |

这样，在所有可关闭的对象中都可以调用此方法，提高了代码的简洁性，比如前面的finally实现可以用以下代码代替：

finally {

if (urlConnection != null) {

urlConnection.disconnect();

}

CloseUtil.closeQuietly(in);

}

这种编码方式也是面向对象程序设计的一个原则：接口隔离原则：类间的依赖关系应该建立在最小的接口上，把臃肿的代码拆分成更小和更具体的类来实现。

至此，就实现了异步加载图片的功能。例如，在一个listView实现加载网络图片并显示，代码如下：

#### 单个图片内存优化

前面我们实现了图片加载和显示，接下来使用Memory Monitor来看看打开一个图片列表时内存的情况（Memory Monitor的使用方法请参考3.3.1节），如图3-34所示。



图3-34　异步加载图片内存增长情况

从图3-34中可以看到，在加载图片时，内存增加峰值非常高，堆内存从32MB冲到接近80MB，但这里只是显示了10张图片，并且ListView中的ImageView的分辨率都是50×50。为了查看内存开销在什么地方，再使用Allocation Tracker来查看具体消耗内存的对象，Allocation Tracker的使用请查看3.3.3节。查看结果如图3-35所示。



图3-35　ListView加载网络图片内存占用

从图3-34可以看到，内存都是byte[]对象，从具体的引用路径可以看出，都是Bitmap对象占用了大量内存，这时可以根据3.5.4节介绍的图片内存优化知识来做一轮优化。本着类的单一原则，使用BitmapConfig类来实现参数的配置，代码如代码清单3-3所示。

代码清单3-3　BitmapConfig类实现参数配置

·BitmapConfig（int width，int height，Bitmap.Config preferred）

BitmapConfig类构造函数，参数为控件需要显示图片的参数。

·getBitmapOptions（InputStream is）

根据目标参数得到对应的BitmapFactory.Options，如果不传图片的Stream，则使用固定的samplesize。

·calculateInSampleSize（）

根据图片原始大小和目标大小计算出准确的SampleSize。

调用MiniImageLoader（代码清单3-2）中的downLoadBitmap方法，增加获取Bitmap-Factory.Options的步骤，代码如下：

因为需要显示的ImageView的分辨率是50×50，所以按这个标准新建一个Bitmap-Config，加载10张相同图片的内存开销明显降低，只需要增加4MB的内存开销，优化效果非常明显，如图3-36所示。



图3-36　加载优化后的图片内存开销

每张图片的大小，相比于图3-35，都控制在65KB左右，如图3-37所示。



图3-37　单张图片的内存占用

虽然图片得到了很好的优化，但仍然存在两个问题：

·相同的图片，每次显示都需要重新加载，浪费了内存和流量。

·整体内存开销不可控，虽然减少了单个图片开销，但在图片非常多的情况下，没有合理的管理机制仍然对性能有严重的影响。

为解决这两个问题，就需要有内存池的设计理念，通过内存池控制整体图片内存，不重新加载和解码已经显示过的图片，下面实现图片模块的缓存。

### 实现三重缓存

在Android应用中不可避免地要显示很多图片，按3.6.1节的实现方案，不管图片是否显示过，每次启动时都需要从网络拉取，这就极大影响了图片加载速度和浪费用户流量，并且整个应用中的图片内存无法控制在一个总的范围内。因此，图片缓存在一个图片加载模块中很重要并且不可缺少。

目前比较流行的图片框架，如Fresco、UIL等，都使用了“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卡和内存中，当程序再一次需要加载图片时，先判断内存中是否有缓存，有则直接从内存中拉取，否则查看本地缓存目录中是否有缓存，本地缓存目录中如果存在缓存，则从本地缓存卡中拉取，否则从网络加载图片。三级缓存的实现逻辑如图3-38所示。



图3-38　三级缓存实现逻辑

在图3-37中，Y轴是获取图片Bitmap的流程，X轴是图片处理的过程。这样在3.6.1节的基础上增加缓存模块，如图3-39所示。



图3-39　增加缓存模块UML图

为了实现这个缓存机制，该图片模块要在3.6.1节的基础上扩展，其实网络级的客户端不用太多的设置，取数据就可以了，在代码中需要实现内存缓存和磁盘缓存。使用三级缓存机制，可以让应用程序避免重复下载和解码相同的图片，有效减少内存开销，同时快速显示已经加载过的图片。

#### 内存缓存

使用软引用或弱引用（SoftReference or WeakReference）来实现内存池是以前的常用做法，但现在不建议开发者使用这种方案。从API 9（Android 2.3）开始，Android系统垃圾回收器更倾向于回收持有软引用或弱引用的对象，这让软引用和弱引用变得不再可靠，并且从Android 3.0（API Level 11）开始，图片的数据无法用一种可预见的方式将其释放，这就存在潜在的内存溢出风险。

使用LruCache来实现图片内存管理是一种可靠的方式，它的主要算法原理是把最近使用的对象用强引用存储在LinkedHashMap中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除，具体介绍详见3.5.3节。

接下来使用LruCache实现一个图片的内存缓存，如代码清单3-4所示。

代码清单3-4　实现图片的内存缓存MemoryCache.java

MemoryCache实现了将图片缓存到一个固定存池中，在下载图片前，先从Memory-Cache中获取，如果获取到图片就不需要重新下载。下面是缓存的几个主要方法：

·Init（float sizePer）

初始化LruCache，参数sizePer是设置图片内存池的size，因为不同的ROM，会根据硬件配置给应用下发不同的最大堆大小，所以使用一个百分比作为参数，一般的应用分配1/8的堆大小就能满足，图片较大的应用可以适当调整大小。

其中LruCache重写了两个方法，sizeOf（String key，Bitmap value），这个方法是必须重写的，返回加入的缓存对象大小，这里是返回一个新的Bitmap对象内存大小。需要注意的是，不同API计算Bitmap内存大小使用的系统API有些差异。重写entryRemoved（boolean evicted，String key，Bitmap oldValue，Bitmap newValue），这个方法在把最近最少使用的对象在缓存值达到预设定值之前从内存中移除时会回调，其中参数oldValue是即将要从LruCache中移除的对象，这里重写是因为后面会用到。

·addBitmapToCache（String url，Bitmap bitmap）

增加一个新的Bitmap到内存池中，url为KEY。

·getBitmap（String url）

通过url获取内存中的Bitmap对象，如果没有，则返回null，需要进行下一步的加载图片逻辑。

这样就实现了一个简单的图片缓存管理类，在对图片解码后，把图片放到内存中，代码如下：

在加载图片前，先尝试从MemoryCache中获取，实现如下代码：

否则走加载

这样，当解码出一组新图片时，都会缓存到LruCache中，在下次获取图片时，首先从LruCache中拉取，但LruCache的大小是有一定限制的，从前面的内容可以看到，LruCache的大小是系统分配给应用的最大堆内存的百分比，具体如下所示：

cacheSize = Math.round(sizePer \* Runtime.getRuntime().maxMemory() / 1024)

这个百分比多少合适呢，可以从以下几个点来考虑。

·应用中内存的占用情况，除了图片以外，是否还有大内存的数据需要缓存到内存。

·在应用中大部分情况要同时显示多少张图片，优先保证最大图片的显示数量的缓存支持。

·Bitmap的规格，计算出一张图片占用的内存大小。

·图片访问的频率。

缓存的大小需要设计好，一个过小的缓存不但没有任何好处，还会引起额外的开销，一个过大的缓存可能使java.lang.OutOfMemory异常的概率增加，并且应用剩余部分只留下很小的内存，导致其他数据缓存空间变小。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　在应用中，如果有一些图片的访问频率要比其他的大一些，或者必须一直显示出来，就需要一直保持在内存中，这种情况可以使用多个LruCache对象来管理多组Bitmap，对Bitmap进行分级，不同级别的Bitmap放到不同的LruCahe中。

#### Bitmap内存复用

在3.5.4章节中介绍了Bitmap内存复用的方法，从Android 3.0版本开始支持Bitmap内存复用，也就是BitmapFactory.Options.inBitmap属性，如果这个属性被设置有效的目标复用对象，decode方法就在加载内容时重用已经存在的Bitmap。这意味着Bitmap的内存被重新利用，这可以减少内存的分配与回收，提高图片的性能。在MemoryCache的基础上增加内存复用的功能。使用mReusableBitmaps存放可重用Bitmap的软引用的集合，代码如下：

因为inBitmap属性在Android 3.0以后才支持，所以只在Android 3.0以上的系统初始化，在后面使用时也需要判断版本号。我们知道在LruCache中重写了entryRemoved方法，但什么都没有做，这个方法是把最后一张Bitmap从LruCahce中移除了，这里可把这个Bitmap加入软引用集合中，作为内存复用的源对象。代码如下：

同样都是在Android 3.0以上版本才这样操作，这样，在需要分配一个新的Bitmap对象时，首先检查是否有可内存复用的Bitmap对象。代码如下：

从代码中可以看到addInBitmapOptions方法在可重用的集合中，通过cache.getBitmap-FromReusableSet（options）方法查找一个合适的bitmap赋值给inBitmap。getBitmapFrom-ReusableSet方法如下：

getBitmapFromReusableSet方法从软引用集合mReusableBitmaps中查找规格可利用的Bitmap作为内存复用对象，因为使用inBitmap有一些限制，在Android 4.4之前，只支持同等大小的位图。因此通过canUseForInBitmap（item，options）方法来判断该Bitmap是否可以复用，代码如下：

只要满足条件，就实现内存复用。

#### 磁盘缓存

内存缓存在访问最近浏览过的Bitmap是很有效的，能明显提高显示的速度，然而并不是所有的图片在这个缓存中都是有效的。最典型的场景就是GridView等一类带有大数据的组件，可以很快填满内存缓存，一旦用户继续浏览，应用就不得不再次处理每一张图片。

如果增加一级磁盘缓存，在磁盘缓存中保存加载过的图片，在图片在内存缓存中失效的地方从本地读取图片数据，再解码，就可以减少图片从网络加载所需的时间。当然，从磁盘上获取这些图片要比从内存中加载慢，并且由于磁盘读取时间是不可预知的，所以图片的解码和文件读取都应该在后台进程中完成。

DiskLruCache是Android提供的一个管理磁盘缓存的类。该类可用于在程序中把从网络加载的数据保存到磁盘上作为缓存数据，DiskLruCache并不是Google官方提供的，是获得官方认可的一个开源工具类，[DiskLruCache源码地址](https://developer.android.com/samples/DisplayingBitmaps/src/com.example.android.displayingbitmaps/util/DiskLruCache.html)

本节使用DiskLruCache来实现一个图片的磁盘缓存DiskCache。

1. 初始化DiskLruCache

DiskLruCache初始化是调用它的open（）静态方法，接口如下：

public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)

directory是缓存图片数据的文件夹，一般建议缓存到SD卡上，SD卡有足够的空间。appVersion是应用/引擎版本号，根据应用的业务需求来维护，如果appVersion发生变化，会自动删除前一个版本的数据。valueCount是指Key与Value的对应关系，一般情况下是1对1关系，这里就写1。maxSize是缓存图片的最大缓存数据大小，根据业务自身需求设置。初始化DiskLruCache的代码如下：

因为open方法中有IO操作，所以建议在异步线程中初始化，但如果在初始化前就要操作写或者读操作会导致失败，所以在初始化过程中加锁，避免同步问题。

（2）写入DiskLruCache

DiskLruCache.Editor类实现写入的操作。通过调用DiskLruCache的edit（）方法来获取实例，写入操作流程如下：

1）获取Editor实例。

Editor实例需要传入一个Key来获取参数，Key必须与图片URL有唯一对应关系，但由于URL中的字符可能会带文件名不支持的字符类型，所以取URL的MD5值作为文件名，实现Key与图片的对应关系，通过URL获取MD5值的代码如下：

2）写入需要保存的图片数据。

有了DiskLruCache.Editor的实例之后，可以调用它的newOutputStream（）方法来创建一个输出流，把需要保存的数据写入OutputStream。在写入操作执行完之后，再调用commit（）方法提交才能使写入生效，调用abort（）方法的话则表示放弃此次写入。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　newOutputStream（）方法接收一个index参数，由于前面在设置valueCount时指定的是1，所以这里index传0就可以了。

图片数据写入本地缓存的整体代码如下：

（3）读取图片缓存

读取的方法要比写入简单一些，通过DiskLruCache的get（）方法实现，代码如下：

这个方法不但把图片数据读取出来，而且一并完成图片解码。

但需要注意的是，读取并解码成Bitmap数据和保存图片数据都是有一定耗时的IO操作。所以一定要记住不能放在主线程中调用这些方法。在本案例中，这些方法都是ImageLoader中的doInBackground方法在异步线程中调用的，代码如下：

从以上代码可以看到，首先在本地缓存中读取，如果读取不到再从网络上下载图片，这样，整个三级缓存就实现了。

### 开源图片组件

前面通过设计一个最简单的图片管理模块，让开发者掌握模块的设计规则以及内存的优化方案。但设计一个成熟的模块，需要做的事情还有很多，比如网络下载部分，这里都只是最简单的实现。如果有兴趣可以在本节的基础上扩展，这里整体架构非常清晰，扩展并非难事。

但在实际应用中，因为开发成本和时间关系，可以考虑使用一些开源的组件，成熟的开源项目不但功能完善、设计优秀，并且在细节上考虑周全，本身也是好的学习材料，通过开源组件的使用和深入学习也能提高开发者的技能。

接下来介绍当前最流行、使用最广泛的三个图片管理组件：UIL（Universal Image-Loader）、Fresco和Picasso，如表3-3所示。



表3-3　开源图片管理组件

这三个图片组件目前都已经成熟并得到广泛使用，它们具有以下几个共同的特点：

·可配置

下载组件、显示控件、内存和本地缓存大小都可以根据业务自身需求灵活配置。

·智能

如果没有特别的要求，在不通过接口指定参数的情况下，组件可以通过对硬件的识别智能配置参数，如线程并发量、缓存大小等。

·多数据源支持

支持大部分主流图片格式，同时对来源也都支持，除了网络数据，还包括资源图片、Asserts资源和本地图片。

·多种控制支持

除了ImageView，也支持其他继承于View的显示控制，包括自定义控件。

·显示速度快

这几个开源组件都实现了多级缓存，可以提高显示的效率。

·使用门槛低

如果没有特别的要求，都可以通过几行代码实现图片显示。

总结来说，这三个组件都是做了相同的一件事情：获取图片并显示在相应的控件上。但这三个并不是没有区别，接下来介绍这三个开源的图片组件各自的特点。

1.UIL

·可配置度高。支持任务线程池、下载器、解码器、内存及磁盘缓存、显示选项等的配置。

·包含内存缓存和磁盘缓存两级缓存，支持定义本地缓存文件名的规则。

·支持多线程，支持异步和同步加载，以及下载进度监听。

·支持多种缓存算法、下载进度监听、ListView图片错乱解决等。

·接口灵活可控，支持暂停图片加载，把CPU时间交回给其他线程。

2.Picasso

·在Adapter中需要取消已经不在视野范围的ImageView图片资源的加载，否则会导致图片错位，Picasso已经解决了这个问题。

·使用复杂的图片压缩转换来尽可能地减少内存消耗。

·自带内存和硬盘二级缓存功能。

3.Fresco

Fresco一经Facebook推出，就得到开发者的认可，并得到广泛使用，它最大的亮点在于它的内存管理，特别是在低端机和Android 5.0以下的机器上的优势更加明显，而使用Fresco将很好地解决了图片占用内存大的问题，Fresco会将图片放到一个特别的内存区域，当图片不再显示时，占用的内存会自动释放，这使得APP更流畅，减少因图片内存占用而引发的OOM。当APP包含的图片较多时，这个效果尤其明显。以下是Fresco的优势。

（1）内存管理

Android中的Bitmap会占用大量的内存，很容易引发更加频繁的GC。在Android 5.0以下，GC将会显著引发界面卡顿（原因见3.2节）。而在Android 5.0以下系统，Fresco将图片放到一个特别的内存区域。在图片不显示时，占用的内存会自动释放。这使得应用更加流畅，同时减少因图片占用内存而引发的OOM。

（2）渐进式呈现

渐进式图片格式是指先呈现大致的图片轮廓，然后随着图片下载的继续，呈现逐渐清晰的图片，这在移动设备，特别是在网络环境差的环境下，可以带来更好的用户体验。

使用时很简单，仅仅需要提供图片的URI即可。

（3）更多的图片格式

支持Gif图和WebP格式。

（4）图像加载策略丰富

Fresco中的Image Pipeline，使开发者可以在多方面控制图片的加载，为同一个图片指定不同的远程路径，比如先显示已经存在本地缓存中的图片，等高清图下载完之后再显示高清图。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　虽然Fresco功能强大，并且性能更优秀，但它有一个最大的劣势是SDK安装包是这三个组件中最大的，如果应用本身需要显示的图片不多或者场景很少，建议使用更轻量的组件。

通过三者的对比，结合业务自身选择最适合当前应用的组件。让应用更快、更稳定地实现。

## 本章小结

通过本章的学习，了解了Android系统的内存管理机制，以及内存对性能的影响，在内存优化上可以参考如图3-40所示的流程。



图3-40　内存优化流程

先检查或者监控来发现问题，再分析问题原因，解决发现的问题或者对当前的实现逻辑进行优化，优化完成后再检查，直到达到设定的性能指标。

1.检查

掌握应用的内存现状需要通过一系列工具，使用Memory Monitor和HeapViewer可以很好地帮助我们查看程序的内存使用情况，可以发现内存峰值和内存抖动，一旦出现这些问题，可以进一步使用Allocation Tracker和MAT详细分析内存，找到内存占用大的对象，发现内存泄漏。

2.监控

工具可以用来检查和分析，但要做到实时防范，使用一些监控工具非常有必要，如本章讲解过的内存泄漏监控工具和大内存分配的报警工具等。

3.优化

优化主要从两个方面进行，一是节省内存开销，比如使用最优的数据结构，减少不必要的内存开销；二是避免内存泄漏，内存泄漏是应用程序中不能容忍的情况。

Random Access Memory（RAM）在任何软件开发环境中都是很宝贵的资源。这一点在物理内存通常很有限的移动操作系统上，显得尤为突出。尽管Android的虚拟机扮演了常规的垃圾回收的角色，但这并不意味着可以忽视应用的内存分配与释放的时机与地点。内存的不合理使用对UI卡顿、耗电、稳定性都有不同程度的影响，因此内存优化是应用性能优化中最重要的优化之一。

但我们知道有些数据从内存中读写的速度是最快的，但内存又是有限的，因此数据保存到本地磁盘中是提高应用读写数据速度的另一种方式，比如3.6节实现的图片管理模块使用了三级缓存，其中磁盘缓存是非常重要的一环，但磁盘的读写非常耗时，频繁的IO操作对CPU和卡顿会产生不可忽视的影响，在下一章，将通过了解Android系统的存储方式，掌握如何选择最佳的存储方法以及更好的优化方案。

# 存储优化

## 存储方式

Android系统提供4种基本的数据存储方式，分别是SharedPreferences存储方式、文件存储方式、SQLite数据库存储方式和ContentProvider存储方式。

·SharedPreference：一个轻量级数据存储类，适用于保存软件配置参数。使用Shared Preferences保存数据，最终是使用xml格式文件存放数据，文件存放在/data/data/<package name>/shared\_prefs目录下。

·SQLite：一个轻量级的数据库，支持基本SQL语法，是常被采用的一种数据存储方式。Android为此数据库提供了一个名为SQLiteDatabase的类，封装了一些操作数据库的API。

·File：通用的文件（I/O）存储方法，常用于存储大数量的数据，但缺点是更新数据比较麻烦。

·ContentProvider：Android系统中所有应用程序实现数据共享的一种数据存储方式，由于数据通常在各应用间的是相对私密的，为了数据安全，应用相互之间不能直接实现共享，特别是音频、视频、图片和通信录，一般都可以采用此种方式存储。每个Content Provider都会对外提供一个公共的URI（包装成URI对象），如果应用程序有数据需要共享，就需要使用Content Provider为这些数据定义一个URI，然后其他应用程序就通过Content Provider传入这个URI来对数据进行操作。

选择哪种数据存储方式，需要根据数据量大小、数据格式以及数据结构来选择性价比最高的存储方式，接下来介绍这4种存储方式的基本特性。

### SharedPreferences

SharedPreferences是一种简单便捷的存储方式，在Android应用中，常用来存储一些简单配置信息，例如保存应用中的一个开关状态和一些数据的独家属性等。

1）特点

用Map数据结构来存储数据，以键值（Key-Value）的方式存储，可以简单地读取与写入；数据读取与写入的方法都非常简单。

（2）存储路径

SharedPreferences将数据保存到一个XML文件中，该文件位于/data/data/Package Name/shares\_prefs下。

（3）创建模式

·Activity.MODE\_APPEND：如果该文件已经存在，就将数据写入，而不是抹掉现有文件的末尾。

·Activity.MODE\_PRIVATE：默认模式，只能由创建该文件的应用程序调用，即为私有的。

·Activity.MODE\_WORLD\_READABLE：允许所有其他应用程序读取和创建文件的权限。

·Activity.MODE\_WORLD\_WRITEABLE：允许所有其他应用程序具有写入、访问和创建的文件权限。

SharedPreferences的最大优点就是使用简单，速度也快，是一些配置数据的首选存储方式。比如创建一个私有的SharedPreferences，如下代码就可以实现SharedPrefenrences的创建与读写功能。

但SharedPreferences也有明显的缺点，就是只能存储boolean、int、float、long和string五种简单的数据类型。

### 文件存储

Android文件系统与其他平台基于磁盘的文件系统类似，文件存储对象适合于读写大量的流式数据，如媒体文件（图片、音视频等）和其他网络传输内容。

1）特点

Android提供了openFileOutput（）方法可以把数据输出到文件中并保存，具体的实现过程与在J2SE环境中保存数据到文件中相同，所以上手比较简单。

（2）存储方式

Android系统将文件存储区域分为内部存储和外部存储两部分。这种命名来源于Linux的文件系统，Android早期的设备都有一个内置的不可变的内存（内部存储），另外还有一个可移除的存储介质，如SD卡（外部存储）。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　后来即使设备没有可移除的存储介质，Android文件系统也将永久存储空间划分为“内部”和“外部”，并且无论外部存储是否可以移除，这两部分存储空间的API行为都是相同的。

两种存储方式的主要区别如下：

·内部存储空间内的文件默认只有创建文件的应用可以访问，而外部存储所有应用都可以访问。

·内部存储在应用卸载后，数据也一起删除。

·外部存储在有的设备上是可以移除的，所以使用外部存储，需要判断是否已挂载。

一般一些应用的配置文件放到内部存储，在应用创建的文件不需要访问限制、应用需要将文件分享给其他的应用，或者允许用户通过计算机来访问时，优先考虑外部存储。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　尽管应用默认被安装到内部存储空间中，但是可以在AndroidManifest中指定android：installLocation属性来使应用安装到外部存储空间。

### SQLite（需要扩展）

SQLite是一款开源嵌入式数据库引擎，同时也是一款轻型的数据库，遵守了ACID的关系型数据库管理系统，支持SQL查询，并且只用很少的内存。因为Android在运行时集成了SQLite，所以每个Android应用程序都可以使用SQLite数据库。对于熟悉SQL的开发人员来说，使用SQLite相当简单。

（1）特点

·轻量级：使用SQLite只需要带一个动态库，就可以享受它的全部功能，而且动态库的尺寸相当小。

·独立性：SQLite数据库的核心引擎不需要依赖第三方软件，也不需要所谓的“安装”。

·隔离性：SQLite数据库中的所有信息（如表、视图、触发器等）都包含在一个文件夹内，方便管理和维护。

·跨平台：SQLite目前支持大部分操作系统，除了计算机操作系统，在众多的手机系统也能够运行，如Android。

·多语言接口：SQLite数据库支持多语言编程接口。

·安全性：SQLite数据库通过数据库级上的独占性和共享锁来独立处理事务。这意味着多个进程可以在同一时间从同一数据库读取数据，但只能有一个可以写入数据。

（2）优势

·多线程访问数据。

·需要事物处理。

·应用程序需要处理需要变化的复杂数据结构。

·数据库对于创建它们的包套件是私有的。

### ContentProvider

Android系统为了实现数据的安全，有些数据存储是私有的，比如数据库和Shared Preference，这一类数据在Android当中是私有的，当然也包括某些文件数据、数据库数据以及一些其他类型的数据。在两个程序之间对数据进行交换可以通过ContentProvider来实现。

在Android系统中，一个程序可以通过实现Content Provider的一个抽象接口将自己的数据暴露出去。外界根本看不到，也不用看到这个应用暴露的数据在应用当中是如何存储的，或者是用数据库存储、用文件存储，还是通过网上获得，这些一切都不重要，重要的是外界可以通过这一套标准及统一的接口和程序中的数据打交道，可以读取程序的数据，也可以删除程序的数据，当然，中间也会涉及一些权限的问题。

## 序列化

数据的序列化是程序代码中必不可少的组成部分，Java提供了一种对象序列化的机制，在该机制中，一个对象可以表示为一个字节序列，该字节序列包括该对象的数据、有关对象类型的信息和存储在对象中数据的类型。将序列化对象写入文件之后，可以从文件中读取出来，并且对它进行反序列化，也就是说，对象的类型信息、对象的数据，以及对象中的数据类型可以用来在内存中新建对象。在讨论数据序列化的性能时，需要了解有哪些候选的方案，它们各自的优缺点是什么。

数据序列化通常在以下场景中使用：

·永久性保存对象，将对象的字节序列保存到本地文件中。

·对象在网络中传递。

·对象在IPC间传递。

接下来介绍几种在Android上序列化数据方法。

### Serializable与Parcelable

大部分开发者会通过实现类的Serializable接口来实现序列化，Serializable接口是来自Java中的序列化接口，代码如下。

但是这种实现方法在性能方面不理想，Serializable在序列化时会产生大量的临时变量，在序列化的过程中会消耗更多的内存，从而引起频繁的GC。

Android也提供了一个序列化接口Parcelable。但Parcelable有个明显的缺点，不能使用在要将数据存储在磁盘上的情况，如永久性保存对象、保存对象的字节序列到本地文件中，因为Parcel本质上是为了更好地实现对象在IPC间传递，并不是一个通用的序列化机制，改变Parcel中任何数据的底层实现都可能导致之前的数据不可读取，所以此时还是建议使用Serializable。

可以总结出Serializable的性能较差，而Parcelable又有使用限制。以下介绍几种其他的序列化方法。

### Gson实现JSON的序列化和反序列化

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式，它基于JS的一个子集。JSON采用完全独立于语言的文本格式，这使得JSON成为理想的数据交换语言，易于阅读和编写，也易于机器解析和生成，并且JSON格式与语言无关，扩展性更强。

使用Gson库来处理数据序列化的问题，不仅执行速度更快，内存的使用效率也更高。

Gson有两个重要的方法，toJson和fromJson，也就是序列化和反序列化这两个方法的代码如下：

Gson库非常强大，这里不再详细介绍，读者可以查看相关的文档和书籍。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　Android的XML布局文件会在编译的阶段被转换成更加复杂的格式，具备更加高效的执行性能与更高的内存使用效率。

### Nano Proto Buffers

在讲解Nano Proto Buffers前，需要先了解Protocal Buffers。Google Protocol Buffer（简称Protobuf）是Google公司内部的混合语言数据标准，目前正在使用的有超过48？162种报文格式定义和超过12？183个.proto文件。它们用于RPC系统和持续数据存储系统。

Protocol Buffers是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者序列化。它很适合做为数据存储或RPC数据交换格式。可用于通信协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式，目前提供了C++、Java、Python三种语言的API。

虽然Protobuf强大、灵活，但是对内存的消耗会比较大，所以并不是移动终端上的最佳选择。因此基于Protobuf修改出了一个新的协议Nano Protobuf，相比于Protobuf，Nano Protobuf去掉了一些复杂的特性，生成的代码更加小巧、简单，并且根据移动终端做了特殊的优化，代码执行效率更高，内存使用效率更佳，更加适用于移动设备。

### FlatBuffers

FlatBuffers是一个开源的、跨平台的、高效的、提供了C++/Java接口的序列化工具库。它是Google专门为游戏开发或其他性能敏感的应用程序需求而创建的。尤其适用于移动平台。

主要优点：

·对序列化数据的访问不需要打包和拆包，它将序列化数据存储在缓存中，这些数据既可以存储在文件中，又可以通过网络原样传输，而没有任何解析开销；

·内存效率和速度：访问数据时的唯一内存需求就是缓冲区，不需要额外的内存分配。这里可查看详细的基准测试。

·扩展性、灵活性：它支持的可选字段意味着不仅能获得很好的前向/后向兼容性（对于长生命周期的游戏来说尤其重要，因为不需要每个新版本都更新所有数据）。

·最小代码依赖：仅仅需要自动生成的少量代码和一个单一的头文件依赖，很容易集成到现有系统中。

·强类型设计：尽可能使错误出现在编译期，而不是等到运行期才手动检查和修正。

·使用简单：生成的C++代码提供了简单的访问和构造接口；而且如果需要，通过一个可选功能可以在运行时，高效解析Schema和类JSON格式的文本。

·跨平台：支持C++11、Java，而不需要任何依赖库；在最新的gcc、clang、vs2010等编译器上工作良好。

### 小结

Protocol Buffers和FlatBuffers比较类似，但其主要区别在于FlatBuffers在访问数据前，不需要解析/拆包这一步，而且Protocol Buffers既没有可选的文本导入/导出功能，也没有Schemas语法特性。

JSON非常具有可读性，而且当和动态类型语言（如JavaScript）一起使用时非常方便。然而在静态类型语言中序列化数据时，JSON不但具有运行效率低的明显缺点，而且会让你写更多的代码来访问数据。上面介绍的这些方案在性能方面的数据对比如表4-1所示。



表4-1　Encode/Decode耗时对比

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　如果没有特殊的需求，为了避免序列化带来的性能问题，可以考虑使用Shared Preference或者SQLite来存储那些数据，避免先把那些复杂的数据进行序列化的操作。

## SharedPreferences优化

SharedPreferences实际上是对一个XML文件存储key-value键值对，每一次的commit和apply操作都是一次I/O写操作。众所周知，I/O操作是最慢的操作之一，在主线程中操作会导致主线程缓慢。SharedPreferences性能优化主要是两个方面：

·IO性能。

·同步锁问题。

IO性能瓶颈是导致SharedPreferences性能差最大的原因，SharedPreferences上的IO分为读取数据到内存与数据写入磁盘两种情况。

·当SharedPreferences文件还没有被加载到内存时，调用getSharedPreferences方法会初始化文件并读入内存，这容易导致耗时更长。

·Editor的commit或者apply方法每次执行时，同步写入磁盘耗时较长。

需要说明的是，editor的commit和apply方法的区别在于同步写入和异步写入，以及是否需要返回值。在不需要返回值的情况下，使用**apply**方法可以极大提高性能。

另一方面就是同步锁的问题，SharedPreferences类中的commitToMemory（）方法会锁定SharedPreference对象，Put（）和getEditor（）方法会锁定Editor对象，在写入磁盘时更会锁定一个写入锁。

因此，最好的优化方法就是避免频繁地读写SharedPreferences，减少无谓的调用，如下伪代码在同一生命周期内，SharedPreferences读一次即可。

if(开关未读取过&没有发生过变化)

[代码]

而对于SharedPreferences的批量操作，最好先获取一个editor，进行批量操作，然后调用apply方法。这样会比commit方法略快。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　跨进程读写SharedPreferences需要用到ContentProvider方案支持，对所有SP操作套上了ContentProvider进行访问，耗时增加三倍左右，因此尽量避免跨进程操作。

## ？ 数据库使用及优化

数据库是Android应用开发中广泛使用的一种数据存储方式，也是比较复杂的存储方式，本节实现一个简单的联系人管理模块，通过保存和更新联系人列表来学习数据库的使用和优化方法。这个模块列表实现两个功能：数据库的创建以及一个具体表（联系人列表）的实现。联系人管理模块的示意图如图4-1所示



图4-1　联系人管理模块

代码和优化不展开了，等到需要用到的时候再看这部分！！

## 本章小结

在Android应用中，数据库是应用开发中常用的技术。Android默认使用了SQLite数据库，而且SDK提供了接口非常友好的操作类和方法。因此，对于Android应用开发者来说，使用门槛并不高，也容易上手，并且在应用开发中使用最多的无外乎增删改查。然而纵使操作简单，由于本身就是耗时的操作，如果没有很好地优化，也可能出现查找数据缓慢、插入数据耗时严重等情况，特别是数据量大或者表结构复杂时，会把问题放大，因此，优化数据库的操作是非常有必要的。

和图片管理模块一样，数据库作为一个应用的底层通用模块，也有需要开源的框架，这些开源框架基本上都采用了对象关系映射（ORM）的模式。应用比较广泛的有AFinal、LitePal和greenDAO，这三种都是成熟稳定的开源数据库框架，开发者可以在网上找到丰富的使用资料，这里就不详细讲解如何使用。需要说明的是，greenDAO与AFinal和LitePal两种ORM框架有所不同，它的原理不是根据反射进行数据库的各个具体操作，而是直接使用人工生成业务需要的DAO和Model文件，业务中可以直接调用相应的DAO文件进行数据库操作，从而避免了因反射带来的性能损耗并且导致效率降低。但是因为需要人工生成model和DAO文件，所以greenDAO的配置就相比另外两种会复杂一些。

同时可以看到在数据库的底层方法实现中，有大量的try/catch语句，如果数据来源不可靠，可能会导致程序崩溃，所以catch是为了保证应用的稳定性。而应用的稳定性差除了体现为程序崩溃（Crash），还体现为假死（ANR）情况，下一章我们将具体介绍这两种情况的监控方案。

# 稳定性优化

Android应用的稳定性定义很宽泛，影响稳定性的原因也很多，比如内存使用不合理、代码异常场景考虑不周全、代码逻辑不合理等，都会对应用的稳定性造成影响。本章讨论的是稳定性中的两个常见场景：Crash和ANR，这两种错误将使程序无法使用。因为引发这两种情况的原因无法系统归类，因此本章不会介绍如何修改某个Crash或者ANR，而是介绍一系列的方法和工具，探索如何发现问题和规避问题，把风险降低。

本章从如何提高代码的质量以减少这些情况的发生，到通过对Crash和ANR的监控，以对应用的稳定性有一个全局的掌握，再到采取有效的上报机制逐个解决。

## 提高代码质量

代码质量是一个很大的话题，不同的编程语言都有优秀的书籍介绍如何编写高质量的代码，因此这里不介绍如何编写。然而实际的应用开发团队都会面临以下两个问题：

1）程序员的水平差异：如果程序员基础扎实并且开发经验丰富，毫无疑问，可以写出更高质量的代码，然而在开发团队中，难免会有一些经验较少、代码编写质量并不好的程序员存在。

2）开发人员变动：人员变动是不可避免的，一个模块的开发人员变更后，新的开发者可能对以前的设计理解不全面，从而产生偏差。而一般前期也不会出现问题，并且满足完成需求，但随着时间的推移会发现开发越来越吃力，不停地打补丁修复问题。

因此，要解决这两个问题可以从两个方向入手：代码编写阶段的优化和完成后借助工具检查发现问题。

本节介绍在实际项目开发中，非常重要的提高代码质量的两种方法：代码审查与重构。

### 代码审查

代码审查也就是我们常说的Code Review，是指程序员在完成自己的代码后，由其他人来复查他写的代码，目的是发现应用开发初期未发现的错误并修改，同时提升软件质量及开发者的技术。

在开发阶段增加Code Review环节，可以大大提高代码的质量，不但能发现缺陷，也能提高代码的可读性和可维护性。特别是较大的开发团队，开发人员的变动会很频繁，代码审查可以督促开发者思考如何提高编码质量。做代码审查的人也是受益者，通过读别人的代码，可以更深刻地理解什么是代码的可读性。代码审查还可以促进开发团队人员之间的技术交流。

虽然代码审查有如此大的好处，但使用不当带来的坏处也不能轻视。因为代码审查的代价不小，需要从时间、人力、代码质量方面综合考虑，形成有效的代码审查机制需要解决以下三个问题：

·何时代码审查？

确定什么时候需要代码审查。

·谁来代码审查？

确定代码审查的对象是谁。

·代码审查的内容是什么？

有目的地代码审查。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　代码审查不仅仅是找Bug，更多是使代码更易读，有更高的维护性，同时可以达到知识共享。

#### 何时代码审查

何时代码审查分两个方面：一是这个模块是否需要审查，明确代码审查的必要性，二是在开发阶段的哪个时间点代码审查，确定代码审查合理的时间点。

在实际的应用开发过程中，一个主要的矛盾是需求开发量大，但开发人力有限，而代码审查增加了不少的时间。权衡上线时间压力与稳定性需求，在确定模块是否要代码审查和何时代码审查时可以从以下几个方面考虑。

（1）底层公共模块

比如网络模块、图片管理模块这类底层公共模块，因为整个应用都使用这些模块，并且因为要提供给上层模块的开发者使用，如果有问题影响面大。这类模块因为不涉及具体的业务，不可避免地会对设计考虑不全，通过代码审查可以及早发现模块的缺陷并修复，同时让接口做得更友好。因此，这类模块不仅在开发完成后需要代码审查，以后的维护、修改都需要代码审查。

2）重大特性业务

由于重大特性业务的复杂性，更容易出现问题，并且大的特性维护的时间也会比较长，通过代码审查及早发现问题，达到更好的可维护性。重大特性业务尽量不要出现重大问题，否则影响应用的用户口碑。

（3）与其他模块有耦合

设计一个新的业务模块时，因为产品特性的原因，或多或少会有些耦合，通过代码审查发现两个（或者多个）模块之间是否对业务理解一致，避免业务或者体验有矛盾。

（4）新手

新来的开发者，特别是刚毕业的学生，也许他们做的功能比较简单，并不是前面两种模块中的一种，但通过代码审查可以发现一些代码规范方面的不足和一些明显的缺陷，使他们更快地向有经验的开发者学习。

（5）应用即将发布前

全量测试已经完成，在准备上线的情况下，有些紧急Bug的修复需要代码审查，主要是发现真正的问题以修复，并且没有带来新问题，避免上线后出现严重问题再重新发版本。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　代码审查是一个耗时耗力的过程，过度的代码审查不但不能达到预期的效果，并且容易引起一些经验丰富的程序员反感，因此要避免没有必要的代码审查，影响团队开发的效率。

#### 2.谁来代码审查

如何选择合适的人去做代码审查非常重要，由于应用开发需要敏捷思维，所以也不能固定一种审查方式，而是根据不同的业务/模块选择审查者和审查的方式，代码审查分为三种方式：团队审查、模块负责人审查、结对审查。可以根据不同的情况选择。

（1）团队审查

俗话说“三个臭皮匠，顶个诸葛亮”。即使是经验欠缺的团队，如果能群策群力，也能交付出高质量的应用。这里先不论代码审查对团队的成长的帮助，单单是每一次修改都让其他人审查的行为，就能提高初级程序员的信心、对产品的参与度和责任心。这种审查方式适用于底层公用的模块，通过审查让大家都熟悉底层的实现原理和接口调用方式，在以后使用该模块时可以很快地理解如何使用，并且根据上层业务特点查看模块实现是否合理，对上层业务是否有影响。

但这种审查方式，要整个项目组全体成员去审查，毫无疑问付出的代价太大，因此只适用于底层通用的大模块，如网络引擎、图片管理模块等。而其他的模块可以考虑使用后面两种审查方式。

（2）模块负责人审查

一个应用的开发，可以根据功能模块划分成多个开发小组，每个开发小组的负责人无疑是开发经验最丰富的人员，也是对业务最了解的人员，由他负责组织和指导其他成员是非常合适的。而这位负责人，有责任有效地完成代码审查工作，一般重大业务模块可以使用这个方案，因为重大业务比较复杂，需要一个经验丰富和业务熟悉的人去代码审查，避免风险，同时因为业务模块和底层公用模块不同，所以没有必要团队集体审查。

（3）结对审查

结对代码审查和结对编程（pair-programming）一样，两个开发者，一个人是Driver角色，另一个人是Observer。Driver编写完代码后，由Observer查看Driver的代码，两人一起讨论代码的实现方案，而这两个角色是可以互换的。结对代码审查除了提高代码质量外，还因为一个模块有两个人熟悉，在碰到紧急事情，原开发者没有时间跟进时，原来的Observer可以很快代替原Driver来工作，减少熟悉代码的过程。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　结对编程也有一些弊端，对于不同习惯的程序员，有时候会对问题坚持自己的看法，容易产生争吵，一定层面上有些内耗，因此是否使用结对编程需要根据具体问题考虑，同时需要很好地处理过程中的矛盾。

#### 代码审查内容

在代码审查时，需要确定好审查的基本方向，否则漫无目的地查看代码，没有针对性地分析代码作用不大，这样也容易使审查流于形式。审查的流程为：先审查设计实现思路，然后审查设计模式，接着审查成形的骨干代码，最后审查完成的代码。一般来说审查代码时可以从以下几个方面查看代码。

（1）实现思路与设计思想

·单一职责原则：一个模块有且只能有一个职责。如果一个模块或者一个类提供了不同类型的功能，或者一个功能需要几个模块共同完成，这就有可能在抽象层上设计不合理。

·开闭原则，在面向对象的语言中，对象对可扩展开放、对修改关闭。所以需要考虑添加/扩展另外的内容时是否会带来新的问题。

1. 代码设计

代码复用：根据“三振法”，如果代码被复制一次，通常没什么问题。但如果复制超过三次，就应该通过提取公共的部分来重构它。

·更合理的代码：是否有更好的实现。

·潜在的缺陷：异常情况考虑是否全面，错误的传参是否会引起的其他错误？循环是否以我们期望的方式终止？

严重缺陷有两种类型：业务实现是否有缺陷或者有矛盾的地方。代码缺陷，如溢出、空指针等，这些审查非常重要，但简单而且机械，比较耗时，建议通过代码静态扫描工具发现，在5.12节介绍常用的静态代码扫描工具。

·错误处理：要考虑错误确定后是否被优雅地修改，是否会导致其他错误，修改是否有用。

·效率：如果代码中包含算法，这种算法是否高效？

（3）设计逻辑

设计逻辑和设计模式是复杂并且有深度的审查，需要有一定理论基础和开发经验的人才能有效发现设计逻辑方面的错误。

（4）代码风格

每个人都有一些不同的编码习惯，不用规范得太严格，只需要关注以下几点：

·命名：在计算机科学中，命名是一个难题。一个不准确的函数命名会误导使用者的预期。数据结构尽量写得详细些，在别人查看或自己重新查看代码时，那些见名知意的命名可以让人更容易理解它们的定义。

·方法长度：对于一个方法的长度，一般的经验值是小于100行，如果一个函数在100行以上，最好是分成多个方法。但在Android系统上存在不同的情况，因为在Android应用中有个最大方法数的问题，所以一个方法的长度多少倒不是很重要，需要根据应用自身的特点去考虑。

·函数参数个数：建议越少越好，如果需要参数很多，建议封装成一个新的数据结构。

（5）需求理解是否到位

这需要对业务特别熟悉，特别是复杂的业务需求，需要审查实现后的结果是否和预期一致。

#### 小结

一直强调代码审查是一个比较重的环节，合理地代码审查，不要流于形式，也不要成为开发者的负担。

然而再好的代码审查也不能发现所有的问题，特别是一些代码上的缺陷，再仔细认真的人也会有疏漏，所以需要借助一些代码审查工具，代替部分人工审查，提高开发效率。

### 代码静态扫描工具

静态代码分析是指通过分析或检查源代码的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的错误和缺陷，比如参数类型不匹配、错误的递归、线程安全、非法计算，以及可能出现的空指针引用等。

在Android开发过程中，开发人员不可避免地需要花费大量的时间去发现并修改代码缺陷，而静态代码分析工具可以作为开发辅助工具，在构建过程中帮助开发人员快速发现问题，然后快速修改，这样可以在一定程度上提高软件的稳定性，并且节省开发和测试成本。目前在Android应用开发中，大部分是Java代码，Java静态代码分析工具种类繁多，且各有优缺点，本节介绍4种常用的Java代码分析工具：Android Lint、Findbugs、Checkstyle以及PMD，如表5-1所示。



表5-1　4种静态代码扫描工具对比

（1）Checkstyle

Checkstyle是SourceForge的开源项目，通过对代码编码格式、命名约定、Javadoc、类设计等方面进行代码规范和风格的检查，从而有效约束开发人员更好地遵循代码编写规范。

（2）FindBugs

FindBugs是由马里兰大学提供的一款开源Java静态代码分析工具。FindBugs通过检查类文件或JAR文件，将字节码与一组缺陷模式进行对比，从而发现代码缺陷，完成静态代码分析。FindBugs既提供可视化UI界面，也可以作为Eclipse插件使用。文本主要将FindBugs作为Eclipse插件。在安装成功后会在Eclipse中增加FindBugs perspective，用户可以对指定Java类或JAR文件运行FindBugs，此时FindBugs会遍历指定文件，进行静态代码分析，并将代码分析结果显示在FindBugs perspective的bugs explorer中。

1. PMD

PMD是由DARPA在SourceForge上发布的开源Java代码静态分析工具。PMD通过其内置的编码规则对Java代码进行静态检查，主要检查潜在的bug、未使用的代码、重复的代码、循环体创建新对象等问题。PMD提供了和多种Java IDE的集成，如Android Studio、Eclipse，IDEA、NetBean等。

（4）Android Lint

在前面的章节中介绍过Android Lint部分功能的使用，这里不再过多介绍。Android Lint是一个非常强大的Android应用程序代码检查利器，除了代码缺陷外，还能检查代码布局的合理性等。

由于这4种工具内置的编程规范各有不同，因此它们发现不同种类的代码问题的发现能力也有所不同。其中Checkstyle更加偏重于检查代码编写格式，而FindBugs、PMD更偏重于发现代码缺陷，Android Lint则非常明显地除了检查代码外，也包括了针对Android平台特点的更丰富的检查。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　利用Gradle，在Android平台使用代码质量检查工具是非常容易。比在自己的计算机上使用质量工具局部检查更全面、效率更高，Gradle可以用于自动构建如Jenkins/Hudson这样的平台，让你自动进行质量检查，并自动建立过程。

## Crash监控

Crash（应用崩溃）是由于代码异常而导致App非正常退出，导致应用程序无法继续使用，所有工作都停止的现象。发生Crash后需要重新启动应用（有些情况会自动重启），而且不管应用在开发阶段做得多么优秀，也无法避免Crash发生，特别是在Android系统中，系统碎片化严重、各ROM之间的差异，使Android在稳定性方面需要付出更多的代价。

在应用中发生Crash具有以下两个特点。

·非必现：一般来说Crash不是必现的，可能是一个很少使用的场景触发，在开发和测试阶段都没有考虑到，并且是有概率地触发。

·原因多，无法系统性解决：导致Crash的原因有很多，有代码逻辑缺陷、系统兼容问题、硬件兼容问题，而在应用开发时较少关注到，特别是硬件和ROM的兼容性问题，需要特定的机型/ROM和特定的场景才触发。

因此，要降低Crash发生的概率，需要Crash监控和发生Crash时的堆栈信息，开发者拿到这些信息后分析导致Crash的原因并修复。

在Android应用中发生的Crash有两种类型，Java层的Crash和Native层Crash。这两种Crash的监控和获取堆栈信息有所不同。接下来介绍发生Crash时，如何监控和获取日志。

### Java层Crash监控

在Android中，Java虚拟机为每个进程都设置了一个默认UncaughtExceptionHandler，用于处理本进程中未被try catch的Exception。因此只要实现UncaughtExceptionHandler接口，并在进程启动时调用Thread.setDefaultUncaughtExceptionHandler（...）传入自定义的Uncaught ExceptionHandler，当出现没被catch的异常时，就会回调uncaughtException（Thread thread，Throwable ex）方法，可以在这里记录crash日志，并上报给服务器，也可以执行一些个性化的异常处理操作。

下面是一个实现Thread.UncaughtExceptionHandler接口的类：CrashHandler，代码如代码清单5-1所示。

|  |
| --- |
| package com.android.androidtech;  import android.content.Context;  import android.content.pm.PackageInfo;  import android.content.pm.PackageManager;  import android.os.Build;  import android.os.Environment;  import com.android.androidtech.utils.GLog;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileWriter;  import java.io.IOException;  import java.io.PrintWriter;  import java.text.SimpleDateFormat;  import java.util.Date;  import android.os.Process;  public class CrashHandler implements Thread.UncaughtExceptionHandler {  private static final String TAG = "CrashHandler";  private static final String CRASH\_FILE\_NAME = "crash";  private static final String CRASH\_FILE\_PATH = Environment.getExternalStorageDirectory().getPath() + "/log/";  //log文件的后缀名  private static final String CRASH\_FILE\_NAME\_SUFFIX = ".txt";  //系统默认的异常处理（默认情况下，系统会终止当前的异常程序）  private Thread.UncaughtExceptionHandler mDefaultCrashHandler;  private Context mContext;  //这里主要完成初始化工作  public void init(Context context) {  //获取系统默认的异常处理器  mDefaultCrashHandler = Thread.getDefaultUncaughtExceptionHandler();  //将当前实例设为系统默认的异常处理器  Thread.setDefaultUncaughtExceptionHandler(this);  //获取Context，方便内部使用  mContext = context;  //可以在初始化后在异步线程中上报上次保存的Crash信息  }  /\*\*  \* 这个是最关键的方法，当程序中有未被捕获的异常，系统将会自动调用#uncaughtException方法  \* thread为出现未捕获异常的线程，ex为未捕获的异常，有了这个ex，我们就可以得到异常信息。  \*/  @Override  public void uncaughtException(Thread thread, Throwable ex) {  try {  //导出异常信息到SD卡中  dumpExceptionToSDCard(ex);  //这里可以通过网络上传异常信息到服务器，便于开发人员分析日志从而解决bug  } catch (IOException e) {  e.printStackTrace();  }  //打印出当前调用栈信息  ex.printStackTrace();  //如果系统提供了默认的异常处理器，则交给系统去结束我们的程序，否则就由我们自己结束自己  if (mDefaultCrashHandler != null) {  mDefaultCrashHandler.uncaughtException(thread, ex);  } else {  Process.killProcess(Process.myPid());  }  }  private void dumpExceptionToSDCard(Throwable ex) throws IOException {  //如果SD卡不存在或无法使用，则无法把异常信息写入SD卡  if (!Environment.getExternalStorageState().equals(Environment.MEDIA\_MOUNTED)) {  GLog.e(TAG, "sdcard unmounted,skip dump exception");  return;  }  File dir = new File(CRASH\_FILE\_PATH);  if (!dir.exists()) {  dir.mkdirs();  }  long current = System.currentTimeMillis();  String time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date(current));  //以当前时间创建log文件  File file = new File(CRASH\_FILE\_PATH + CRASH\_FILE\_NAME + time + CRASH\_FILE\_NAME\_SUFFIX);  try {  PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(file)));  //导出发生异常的时间  pw.println(time);  //获取手机信息  getPhoneInfo(pw);  pw.println();  //导出异常的调用栈信息  ex.printStackTrace(pw);  pw.close();  } catch (Exception e) {  GLog.e(TAG, "dump crash info failed");  }  }  private void getPhoneInfo(PrintWriter pw) throws PackageManager.NameNotFoundException {  //TODO 上报一些辅助信息，如应用版本号，系统版本号，手机型号等，方便数据分析和归类  }  } |

|  |
| --- |
| /\*\*  \* 读取当前设备信息 并追加到异常日志中  \*/  private void dumpPhoneInfo(PrintWriter pw) throws PackageManager.NameNotFoundException {  // 获得应用包管理者 并获取存储当前应用的信息对象  PackageManager pm = mContext.getPackageManager();  PackageInfo pi = pm.getPackageInfo(mContext.getPackageName(), PackageManager.GET\_ACTIVITIES);  pw.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*设备信息\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  // 开始写入 应用信息  pw.println("App Version ");  pw.print(" VersionName: ");  pw.println(pi.versionName);  pw.print(" VersionCode: ");  pw.println(pi.versionCode);  // 开始写入 Android版本信息  pw.println("OS Version ");  pw.println(" SDK\_NAME: " + Build.VERSION.RELEASE);  pw.println(" SDK\_INT: " + Build.VERSION.SDK\_INT);  // 开始写入 手机制造商  pw.println("Vendor ");  pw.println(" " + Build.MANUFACTURER);  // 开始写入 手机型号  pw.println("Model ");  pw.println(" " + Build.MODEL);  // 开始写入 CPU架构  pw.println("CPU ABI ");  String[] supportedAbis = SUPPORTED\_ABIS;  for (int i = 0; i < supportedAbis.length; i++) {  pw.println(" " + supportedAbis[i]);  }  pw.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  }  } |

当发生Crash后，CrashHandler类通过回调方法uncaughtException，在这里可以拿到错误的信息，也可以上报更多的辅助数据去分析。而CrashHandler最好是在AppApplication中注册

CrashHandler crashHandler = new CrashHandler();

crashHandler.init(this);

这样就可以监听Java层的Crash，并保存发生Crash的错误堆栈，或者根据需求增加一些辅助信息。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　有些文章指出Error不能通过try/catch捕获异常，实际上因为Exception和Error都是Throwable的子类，所以，Error也是可以被try/catch的。

### ？Native层Crash监控：写NDK代码

5.2.1节讲述了在Java层发生Crash后，如何捕获并保存Crash日志，Native层和Java层不一样，Native层发生Crash后，虽然可以通过Logcat查看日志，但在应用运行过程中不能通过5.2.1节的方式捕获到异常并上报。

我们知道，Android系统是Linux内核。在Linux上，当程序发生Crash时，也会生成core dump文件，然后分析dump文件查找崩溃的原因。和Linux一样，在Android系统的Linux内核中，一种重要的进程间通信方式就是Linux的信号机制。Linux信号除了用于正常的进程间通信和同步外，还负责监控系统异常和中断，当应用程序发生异常时，Linux内核会生成错误信号并通知当前进程。应用进程接收到错误信号后，可以捕获该信号并执行对应的信号处理函数。而在应用运行过程中发生严重错误时会发生Crash。Linux有一类专门用于描述Crash的信号：SIGALRM、SIGHUP、SIGINT、SIGKILL、SIGPIPE、SIGPOLL、SIGPROF、SIGSYS、SIGTERM、SIGUSR1、SIGUSR2、SIGVTALRM，在Android系统上，Native层最常见的导致Crash的信号量如表5-2所示。



表5-2　常见Android Native Crash信号

只要在应用程序中注册了这些信号的处理函数，当JNI crash时，我们的处理函数就会被调用到，然后获取dump文件再上传，后续的工作就和Java层异常逻辑一致了。Android系统除了保留Linux对信号的默认处理方式外，还定义了一些额外的行为，例如SIGQUIT对SIGQUIT信号的默认行为是“终止+CORE”，即产生core dump文件后，立即终止运行。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　可以通过adb输入kill-l，获取当前手机支持的kill信号列表。

和Java层捕获Crash一样，捕获Native层Crash分两个步骤，捕获Native Crash和获取堆栈信息。

前面讲到，要捕获Native层的Crash，需要先在应用程序中注册导致应用Crash的信号量处理函数，在Android Native层注册信号量函数的代码如下：

|  |
| --- |
|  |

在Android应用程序中，native层代码通过Dalvik执行System.loadLibrary（），把so库加载进来，然后执行so中的jint JNI\_OnLoad（JavaVM\*vm，void\*reserved）函数。

。。。。。，ndk单独研究，不在此处展开

### Crash上报机制

不管是Java层，还是Navtice导致了Crash，因为应用已经异常了，所以发生Crash后，尽量先保存日志到本地，在下一次网络正常时再上传日志。Crash日志上报机制如图5-1所示。



图5-1　Crash日志上报机制

## ANR剖析

Application Not Responding, Android系统对于一些事件需要在一定的时间范围内完成，如果超过预定时间能未能得到有效响应或者响应时间过长，都会造成ANR。一般地，这时往往会弹出一个提示框，告知用户当前xxx未响应，用户可选择继续等待或者Force Close。.

和发生Crash一样，ANR也是Android应用程序无法继续运行的一种异常，和Crash的区别是，它不一定是由于程序的异常错误导致的，一般是应用处理长时间没有结果响应导致主进程不能处理下一件事情。

本节介绍ANR的定义和发生ANR的原因，掌握如何快速定位导致ANR的原因以及如何在开发过程中尽量避免发生ANR。

### ANR介绍

ANR（Application Not Responding）即应用无响应，在Android系统中，应用发生的ANR有以下几种类型：

1.KeyDispatchTimeout(InputDispatching Timeout)

最常见的ANR类型是对输入事件**5s**内无响应，比如**按键或触摸事件**在此时间内无响应。

2.BroadcastTimeout(BroadcastQueue)

BroadcastReceiver在指定时间（原生系统默认是**10s**）内无法处理完成，并且没有结束执行onReceive。对于前台广播，则超时为BROADCAST\_FG\_TIMEOUT = 10s；对于后台广播，则超时为BROADCAST\_BG\_TIMEOUT = 60s; 超时检测机制为超过一定时间没有执行完相应操作来触发移除延时消息，则会触发anr;

3. ContentProvider Timeout：内容提供者, CONTENT\_PROVIDER\_PUBLISH\_TIMEOUT = 10s;

4. ServiceTimeout：对于前台服务，则超时为SERVICE\_TIMEOUT = 20s；对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT = 200s；超时检测机制：有序广播的总执行时间超过 2\* receiver个数 \* timeout时长，则会触发anr;有序广播的某一个receiver执行过程超过 timeout时长，则会触发anr;这种类型在Android应用中出现的概率很小

引起ANR的根本原因总体来说有以下两种：

1）应用程序自身逻辑有缺陷，或者在某些异常场景触发了此缺陷，如主线程堵塞、死循环等导致。

2）由于Android设备**其他进程的CPU**占用高、事件没有得到及时的响应、死锁，导致当前应用进程无法抢占到CPU时间片。

所有类型的ANR的过程可分为三个步骤: 埋炸弹, 拆炸弹, 引爆炸弹，下面以Service Timeout为例。

### Service Timeout

Service Timeout是位于”ActivityManager”线程中的AMS.MainHandler收到SERVICE\_TIMEOUT\_MSG消息时触发。

对于Service有两类:对于前台服务，则超时为SERVICE\_TIMEOUT = 20s；

对于后台服务，则超时为SERVICE\_BACKGROUND\_TIMEOUT = 200s

由变量ProcessRecord.execServicesFg来决定是否前台启动

[理解Android ANR的触发原理](http://gityuan.com/2016/07/02/android-anr/)

#### 埋炸弹

文章[startService流程分析](http://gityuan.com/2016/03/06/start-service/)详细介绍Service启动流程. 其中在Service进程attach到system\_server进程的过程中会调用realStartServiceLocked()方法来埋下炸弹. 通过发送delay消息(SERVICE\_TIMEOUT\_MSG). 炸弹已埋下, 我们并不希望炸弹被引爆, 那么就需要在炸弹爆炸之前拆除炸弹.

|  |
| --- |
| ActiveServices.realStartServiceLocked(ServiceRecord r,  ProcessRecord app, boolean execInFg) throws RemoteException {  //发送delay消息(SERVICE\_TIMEOUT\_MSG)，  bumpServiceExecutingLocked(r, execInFg, "create");  try {  //最终执行服务的onCreate()方法  app.thread.scheduleCreateService(r, r.serviceInfo,  mAm.compatibilityInfoForPackageLocked(r.serviceInfo.applicationInfo),  app.repProcState);  } catch (DeadObjectException e) {  mAm.appDiedLocked(app);  throw e;  }  }  ActiveServices.bumpServiceExecutingLocked(ServiceRecord r, boolean fg, String why) {  scheduleServiceTimeoutLocked(r.app);  }  ActiveServices.scheduleServiceTimeoutLocked(ProcessRecord proc) {  long now = SystemClock.uptimeMillis();  Message msg = mAm.mHandler.obtainMessage(  ActivityManagerService.SERVICE\_TIMEOUT\_MSG);  msg.obj = proc;  //当超时后仍没有remove该SERVICE\_TIMEOUT\_MSG消息，则执行service Timeout流程【引爆】  **mAm.mHandler.sendMessageAtTime(msg,**  **proc.execServicesFg ? (now+SERVICE\_TIMEOUT) : (now+ SERVICE\_BACKGROUND\_TIMEOUT));**  } |

#### 拆炸弹

在system\_server进程AS.realStartServiceLocked()调用的过程会埋下一颗炸弹, 超时没有启动完成则会爆炸. 那么什么时候会拆除这颗炸弹的引线呢? 经过Binder等层层调用进入目标进程的主线程handleCreateService()的过程.

|  |
| --- |
| ActivityThread.handleCreateService(CreateServiceData data) {  java.lang.ClassLoader cl = packageInfo.getClassLoader();  Service service = (Service) cl.loadClass(data.info.name).newInstance();  try {  //创建ContextImpl对象  ContextImpl context = ContextImpl.createAppContext(this, packageInfo);  context.setOuterContext(service);  //创建Application对象  Application app = packageInfo.makeApplication(false, mInstrumentation);  service.attach(context, this, data.info.name, data.token, app,  ActivityManagerNative.getDefault());  //调用服务onCreate()方法  service.onCreate();    **//拆除炸弹引线**  ActivityManagerNative.getDefault().serviceDoneExecuting(  data.token, SERVICE\_DONE\_EXECUTING\_ANON, 0, 0);  }  }  AS.serviceDoneExecutingLocked(ServiceRecord r, boolean inDestroying,  boolean finishing) {  if (r.executeNesting <= 0) {  if (r.app != null) {  r.app.execServicesFg = false;  r.app.executingServices.remove(r);  if (r.app.executingServices.size() == 0) {  //当前服务所在进程中没有正在执行的service,则移除服务超时消息SERVICE\_TIMEOUT\_MSG  **mAm.mHandler.removeMessages(ActivityManagerService.SERVICE\_TIMEOUT\_MSG, r.app**);  }  } |

#### 引爆炸弹

前面介绍了埋炸弹和拆炸弹的过程, 如果在炸弹倒计时结束之前成功拆卸炸弹,那么就没有爆炸的机会, 但是世事难料. 总有些极端情况下无法即时拆除炸弹,导致炸弹爆炸, 其结果就是App发生ANR. 接下来,带大家来看看炸弹爆炸的现场:

在system\_server进程中有一个Handler线程, 名叫”ActivityManager”.当倒计时结束便会向该Handler线程发送 一条信息SERVICE\_TIMEOUT\_MSG,

|  |
| --- |
| ActivityManagerService.MainHandler **extends** Handler{  **public void** handleMessage(Message msg){  **switch**(msg.what){  **case** SERVICE\_TIMEOUT\_MSG:{  mServices.serviceTimeout((ProcessRecord)msg.obj);  }**break**;  }  }  } AS.serviceTimeout(ProcessRecord proc){  String anrMessage=**null**;  **synchronized**(mAm){  **if**(proc.executingServices.size()==0||proc.thread==**null**) **return**;  **final long** now=SystemClock.uptimeMillis();  **final long** maxTime=now-  (proc.execServicesFg?SERVICE\_TIMEOUT:SERVICE\_BACKGROUND\_TIMEOUT);  ServiceRecord timeout=**null**;  **long** nextTime=0;  **for**(**int** i=proc.executingServices.size()-1;i>=0;i--){  ServiceRecord sr=proc.executingServices.valueAt(i);  **if**(sr.executingStart<maxTime){  timeout=sr;  **break**;  }  **if**(sr.executingStart>nextTime){  nextTime=sr.executingStart;  }  }  **if**(timeout!=**null**&&mAm.mLruProcesses.contains(proc)){  Slog.w(TAG,**"Timeout executing service: "**+timeout);  StringWriter sw=**new** StringWriter();  PrintWriter pw=**new** FastPrintWriter(sw,**false**,1024);  pw.println(timeout);  timeout.dump(pw,**" "**);  pw.close();  mLastAnrDump=sw.toString();  mAm.mHandler.removeCallbacks(mLastAnrDumpClearer);  mAm.mHandler.postDelayed(mLastAnrDumpClearer,LAST\_ANR\_LIFETIME\_DURATION\_MSECS);  anrMessage=**"executing service "**+timeout.shortName;  }  }   **if**(anrMessage!=**null**){   ***//当存在timeout的service，则执行appNotResponding***mAm.appNotResponding(proc,**null**,**null**,**false**,anrMessage);  } } |

### KeyDispatchTimeout具体的原因

我们需要关注的是第一种情况导致的ANR。**KeyDispatchTimeout具体的原因有**：

(1). 当前的事件没有机会得到处理（即UI线程正在处理前一个事件，没有及时的完成或者looper被某种原因阻塞住了）

(2). 当前的事件正在处理，但没有及时完成

首先要知道事件发生的线程，一般来说大多数可能是ui线程操作超时，那么ui线程都有哪些呢：

(1). Activity 生命周期

(2). View post 的runnable方法 、 handler（MainLooper） 的 handleMessage()

(3). Asycktask 的 onPreExecute(), onPostExecute() , onProgressUpdate()方法

(4). Broadcast 的onReceive()

(5). Service

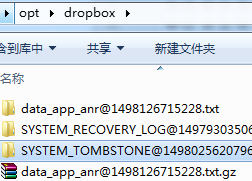
其他原因造成的anr：

(1). 线程死锁

(2). cpu 饥饿

### ANR分析

在Android系统上，如果发生ANR，Logcat会产生对应的日志和一个**/data/anr/traces.txt**文件，只保留最后一次发生ANR时的信息，“/data/system/dropbox则是保存最近3d的anr。”是DB指定的文件存放位置.分析ANR原因主要是分析这两个信息。但是Logcat不一定保证在发生ANR时可以拿到，特别是在非开发人员手上发生ANR情况时，再去拿logcat已经晚了，并且ANR的原因非常复杂。



#### Traces系统源码

SystemServer在启动时， 会创建并添加DROPBOX\_SERVICE.

|  |
| --- |
| //SystemServer.java  private void startOtherServices() {  try {  Slog.i(TAG, "DropBox Service");  ServiceManager.addService(Context.DROPBOX\_SERVICE,  new DropBoxManagerService(context, new File("/data/system/dropbox")));  } catch (Throwable e) {  reportWtf("starting DropBoxManagerService", e);  }  } |

framework是如何输出ANR信息到traces.txt文件中去的？

ActivityManagerService.java中的appNotResponding()方法

|  |
| --- |
| final void appNotResponding(ProcessRecord app, ActivityRecord activity,  ActivityRecord parent, boolean aboveSystem, final String annotation) {  //firstPids和lastPids两个集合存放那些将会在traces中输出信息的进程的进程号  ArrayList<Integer> firstPids = new ArrayList<Integer>(5);  SparseArray<Boolean> lastPids = new SparseArray<Boolean>(20);  //mController是IActivityController接口的实例，是为Monkey测试程序预留的，默认为null  if (mController != null) {  try {  // 0 == continue, -1 = kill process immediately  int res = mController.appEarlyNotResponding(app.processName, app.pid, annotation);  if (res < 0 && app.pid != MY\_PID) {  app.kill("anr", true);  }  } catch (RemoteException e) {  mController = null;  Watchdog.getInstance().setActivityController(null);  }  }  long anrTime = SystemClock.uptimeMillis();  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow(); //更新CPU使用率  }  synchronized (this) {  //一些特定条件下会忽略ANR  // PowerManager.reboot() can block for a long time, so ignore ANRs while shutting down.  if (mShuttingDown) {  Slog.i(TAG, "During shutdown skipping ANR: " + app + " " + annotation);  return;  } else if (app.notResponding) {  Slog.i(TAG, "Skipping duplicate ANR: " + app + " " + annotation);  return;  } else if (app.crashing) {  Slog.i(TAG, "Crashing app skipping ANR: " + app + " " + annotation);  return;  }  // In case we come through here for the same app before completing  // this one, mark as anring now so we will bail out.  app.notResponding = true;  // Log the ANR to the event log.  EventLog.writeEvent(EventLogTags.AM\_ANR, app.userId, app.pid,  app.processName, app.info.flags, annotation);  //当前发生ANR的应用进程被第一个添加进firstPids集合中.  //**所以会第一个向traces文件中写入信息**。反过来说，traces文件中出现的第一个进程正常情况下就是发生ANR的那个进程。  // Dump thread traces as quickly as we can, starting with "interesting" processes.  firstPids.add(app.pid);  int parentPid = app.pid;  if (parent != null && parent.app != null && parent.app.pid > 0) parentPid = parent.app.pid;  if (parentPid != app.pid) firstPids.add(parentPid);  if (MY\_PID != app.pid && MY\_PID != parentPid) firstPids.add(MY\_PID);  for (int i = mLruProcesses.size() - 1; i >= 0; i--) {  ProcessRecord r = mLruProcesses.get(i);  if (r != null && r.thread != null) {  int pid = r.pid;  if (pid > 0 && pid != app.pid && pid != parentPid && pid != MY\_PID) {  if (r.persistent) {  firstPids.add(pid);  } else {  lastPids.put(pid, Boolean.TRUE);  }  }  }  }  }  // Log the ANR to the main log.  StringBuilder info = new StringBuilder();  info.setLength(0);  info.append("ANR in ").append(app.processName);  if (activity != null && activity.shortComponentName != null) {  info.append(" (").append(activity.shortComponentName).append(")");  }  info.append("\n");  info.append("PID: ").append(app.pid).append("\n");  if (annotation != null) {  info.append("Reason: ").append(annotation).append("\n");  }  if (parent != null && parent != activity) {  info.append("Parent: ").append(parent.shortComponentName).append("\n");  }  final ProcessCpuTracker processCpuTracker = new ProcessCpuTracker(true);  //dumpStackTraces()是输出ANR信息到traces文件的函数  File tracesFile = dumpStackTraces(true, firstPids, processCpuTracker, lastPids,  NATIVE\_STACKS\_OF\_INTEREST);  String cpuInfo = null;  if (MONITOR\_CPU\_USAGE) {  updateCpuStatsNow();  synchronized (mProcessCpuTracker) {  cpuInfo = mProcessCpuTracker.printCurrentState(anrTime);  }  //输出ANR发生前一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentLoad());  info.append(cpuInfo);  }  //输出ANR发生后一段时间内的CPU使用率  info.append(processCpuTracker.printCurrentState(anrTime));  //此处， 把ANR的信息输出到main log中.  Slog.e(TAG, info.toString());  if (tracesFile == null) {  // There is no trace file, so dump (only) the alleged culprit's threads to the log  Process.sendSignal(app.pid, Process.SIGNAL\_QUIT);  }  //将ANR信息同时输出到DropBox中  addErrorToDropBox("anr", app, app.processName, activity, parent, annotation,  cpuInfo, tracesFile, null);  //在>=Android4.0中可以设置是否不显示ANR提示对话框，如果设置的话就不会显示对话框，并且会杀掉ANR进程  // Unless configured otherwise, swallow ANRs in background processes & kill the process.  boolean showBackground = **Settings.Secure.getInt(mContext.getContentResolver(),**  **Settings.Secure.ANR\_SHOW\_BACKGROUND, 0) != 0;**  synchronized (this) {  mBatteryStatsService.noteProcessAnr(app.processName, app.uid);  if (!showBackground && !app.isInterestingToUserLocked() && app.pid != MY\_PID) {  app.kill("bg anr", true);  return;  }  // Set the app's notResponding state, and look up the errorReportReceiver  makeAppNotRespondingLocked(app,  activity != null ? activity.shortComponentName : null,  annotation != null ? "ANR " + annotation : "ANR",  info.toString());  // 显示ANR提示对话框  // Bring up the infamous App Not Responding dialog  Message msg = Message.obtain();  HashMap<String, Object> map = new HashMap<String, Object>();  msg.what = SHOW\_NOT\_RESPONDING\_MSG;  msg.obj = map;  msg.arg1 = aboveSystem ? 1 : 0;  map.put("app", app);  if (activity != null) {  map.put("activity", activity);  }  mUiHandler.sendMessage(msg);  }  }  public static File dumpStackTraces(boolean clearTraces, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  //系统属性“dalvik.vm.stack-trace-file”用来配置trace信息输出文件  //之所以trace信息会输出到“/data/anr/traces.txt”文件中，就是系统属性“dalvik.vm.stack-trace-file”设置的  //adb shell下， 可以使用setprop和getprop对系统属性进行设置和读取  //getprop dalvik.vm.stack-trace-file  //setprop dalvik.vm.stack-trace-file /tmp/stack-traces.txt  String tracesPath = SystemProperties.get("dalvik.vm.stack-trace-file", null);  if (tracesPath == null || tracesPath.length() == 0) {  return null;  }  File tracesFile = new File(tracesPath);  try {  File tracesDir = tracesFile.getParentFile();  if (!tracesDir.exists()) {  tracesDir.mkdirs();  if (!SELinux.restorecon(tracesDir)) {  return null;  }  }  //FileUtils.setPermissions是个很有用的函数，设置文件属性时经常会用到  FileUtils.setPermissions(tracesDir.getPath(), 0775, -1, -1); // drwxrwxr-x  //clearTraces为true，会删除旧文件，创建新文件  if (clearTraces && tracesFile.exists()) tracesFile.delete();  tracesFile.createNewFile();  FileUtils.setPermissions(tracesFile.getPath(), 0666, -1, -1); // -rw-rw-rw-  } catch (IOException e) {  Slog.w(TAG, "Unable to prepare ANR traces file: " + tracesPath, e);  return null;  }  dumpStackTraces(tracesPath, firstPids, processCpuTracker, lastPids, nativeProcs);  return tracesFile;  }  private static void dumpStackTraces(String tracesPath, ArrayList<Integer> firstPids,  ProcessCpuTracker processCpuTracker, SparseArray<Boolean> lastPids, String[] nativeProcs) {  // Use a FileObserver to detect when traces finish writing.  // The order of traces is considered important to maintain for legibility.  //使用FileObserver监听SignalCatcher线程是否已经完成写入traces文件的操作  FileObserver observer = new FileObserver(tracesPath, FileObserver.CLOSE\_WRITE) {  @Override  public synchronized void onEvent(int event, String path) {  notify(); //让其他阻塞在observer.wait()这行代码的线程继续往下执行.  }  };  try {  observer.startWatching();  //首先输出firstPids集合中指定的进程，这些也是对ANR问题来说最重要的进程  // First collect all of the stacks of the most important pids.  if (firstPids != null) {  try {  int num = firstPids.size();  for (int i = 0; i < num; i++) {  synchronized (observer) {  //前面提到的SIGNAL\_QUIT, 由SignalCatcher线程处理Process.SIGNAL\_QUIT信号， 并把进程中的各个线程输出到traces.txt文件中.  //具体代码可以查看/dalvik/vm/目录下的SignalCatcher.cpp::logThreadStacks函数和Thread.cpp:: dvmDumpAllThreadsEx函数。  Process.sendSignal(firstPids.get(i), Process.SIGNAL\_QUIT);  observer.wait(200); // Wait for write-close, give up after 200msec  //block here, 直到另一个线程调用了observer.notify() or observer.notifyAll()方法， 再或是等待了200毫秒超时后， 才会继续往下执行代码.  }  }  } catch (InterruptedException e) {  Slog.wtf(TAG, e);  }  }  } finally {  observer.stopWatching();  }  }  Object.java  final void notify()  //Causes a thread which is waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void notifyAll()  //Causes all threads which are waiting on this object's monitor (by means of calling one of the wait() methods) to be woken up.  final void wait()  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method of this object.  final void wait(long millis)  //Causes the calling thread to wait until another thread calls the notify() or notifyAll() method  //of this object or until the specified timeout expires. |

getprop dalvik.vm.stack-trace-file

#### 分析工具Analyze Stacktrace

在Android Studio上提供了一个分析trace文件的工具：**Analyze Stacktrace**。Analyze Stacktrace可以更直观地分析导致ANR的原因。

Analyze Stacktrace使用方法如下：

1）在Android Studio的工具栏中，选择Analyze→Analyze Stacktrace，打开Analyze Stacktrace工具窗口。

2）将traces.txt中的内容复制到窗口，单击Normalize按钮，生成Thread Dump列表，如图5-2所示，左边为所有线程列表，右边为选中线程的具体信息。注意有可能不显示没有上锁的线程。值得注意的是， 当前运行的其他进程也会把当前的函数堆栈信息输出到traces.txt文件中, 但发生ANR的进程正常情况下会第一个输出，所以一般情况下我们只看traces.txt的开头就行了。



图5-2　Thread Dump列表

3）如果某个线程被标红，说明此线程被堵塞了，然后在右边的详细信息中查看堵塞的具体原因。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　部分ANR是由于多个线程之间的同步锁导致，在使用Analyze Stacktrace分析时，可以在详情页查看是在等待哪个线程，再定位到对应的线程。

从以上的分析能够知道， 发生ANR时Android为我们提供了两种“利器”：traces文件和CPU使用率。**线程有很多状态，了解这些状态的意义对分析ANR的原因是有帮助的.**



开发中定位ANR问题日志有个很简单的规律，就是直接找到我们自己开发App所使用的包名（包括第三方Library库）信息开始定位找就可以了

[浅析android 线程状态](http://www.jianshu.com/p/3fb50ddc36d0)，java的6种线程状态定义在/java/lang/Thread.java中:在VMThread.java中, 可以看到下面的代码， native thread有10种状态, 对应着java thread的6种状态.

//

static final Thread.State[] STATE\_MAP = new Thread.State[] {

Thread.State.TERMINATED, // ZOMBIE

Thread.State.RUNNABLE, // RUNNING

Thread.State.TIMED\_WAITING, // TIMED\_WAIT

Thread.State.BLOCKED, // MONITOR

Thread.State.WAITING, // WAIT

Thread.State.NEW, // INITIALIZING

Thread.State.NEW, // STARTING

**Thread.State.RUNNABLE, // NATIVE**

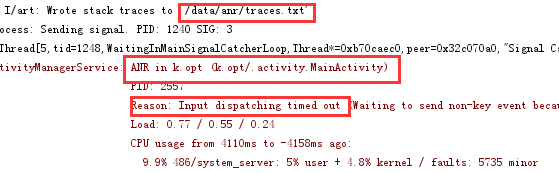
Thread.State.WAITING, // VMWAIT

**Thread.State.RUNNABLE // SUSPENDED**

};

#### KeyDispatchTimeout分析

这里简单构造一个ANR，在HomePageActivity中的单击处理函数中等待100s（Thread，sleep），然后在屏幕上滑动，投递触屏事件，结果出现ANR。发生ANR后，首先看Logcat的信息，**Logcat**信息如下：

从以上Logcat信息中，关注标粗的几个值的信息，它们的意义如下：

·ANR IN：发生ANR的具体类。

·PID：发生ANR的进程，系统在此时会生成trace文件，当前的时间点也是发生ANR的具体时间，以及生成trace文件的时间。

·Reason：当前ARN的类型以及导致ANR的原因。

·CPU usage：CPU的使用情况，在日志中CPU usage有两个时间点，第一个是发生ANR前的CPU使用情况，第二个是发生ANR后的CPU使用情况。

此时用户如果在屏幕上滑动，会产生InputDispatcher事件得不到处理

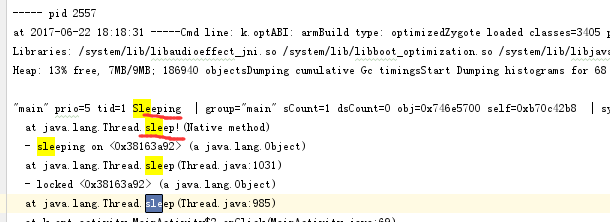
从Logcat中除了能看出在哪个类发生ANR以及ANR的类型，具体的原因主要还是看CPU的使用情况，如果**CPU使用量很少，说明主线程可能阻塞**，如果IOwait很高，说明ANR有可能是由于主线程进行耗时的I/O操作造成的。

接下来分析这个文件**/data/anr/traces.txt**。

可以直接使用adb工具获取traces.txt文件：adb pull/data/anr/traces.txt.

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　trace文件在非root手机上也可以读取，虽然在非root手机上没有权限查看data目录下的内容，**但anr目录可以读取**。

用工具使用，读取出来的traces.txt内容如下：



因此当前这个例子故意写了一个ANR，所以很容易定位到就是在onClick事件中超时导致，在这个日志中需要关注的是线程的状态。

#### [死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)分析

参考：[通过Android trace文件分析死锁ANR](http://blog.csdn.net/oujunli/article/details/9102101)

|  |
| --- |
| 对应的部分trace文件内容如下：  "PowerManagerService" prio=5 tid=24 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41dd0eb0 self=0x5241b218   | sysTid=567 nice=0 sched=0/0 cgrp=apps handle=1380038664   | state=S schedstat=( 6682116007 11324451214 33313 ) utm=450 stm=219 core=1   at com.[**android**](http://lib.csdn.net/base/android).server.am.ActivityManagerService.broadcastIntent(ActivityManagerService.[**Java**](http://lib.csdn.net/base/java):~13045)   - waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)   at android.app.ContextImpl.sendBroadcast(ContextImpl.java:1144)   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.unblankAllDisplays(PowerManagerService.java:3442)   at com.android.server.power.DisplayPowerState$PhotonicModulator$1.run(DisplayPowerState.java:456)   at android.os.Handler.handleCallback(Handler.java:800)   at android.os.Handler.dispatchMessage(Handler.java:100)   at android.os.Looper.loop(Looper.java:194)   at android.os.HandlerThread.run(HandlerThread.java:60)      "Binder\_B" prio=5 tid=85 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x42744770 self=0x58329e88   | sysTid=3700 nice=-20 sched=0/0 cgrp=apps handle=1471424616   | state=S schedstat=( 1663727513 2044643318 6806 ) utm=132 stm=34 core=1   at com.android.server.power.PowerManagerService$DisplayBlankerImpl.toString(PowerManagerService.java:~3449)   - waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)   at java.lang.StringBuilder.append(StringBuilder.java:202)   at com.android.server.power.PowerManagerService.dump(PowerManagerService.java:3052)   at android.os.Binder.dump(Binder.java:264)   at android.os.Binder.onTransact(Binder.java:236)   at android.os.IPowerManager$Stub.onTransact(IPowerManager.java:373)   at android.os.Binder.execTransact(Binder.java:351)   at dalvik.system.NativeStart.run(Native Method)    "android.server.ServerThread" prio=5 tid=12 MONITOR   | group="main" sCount=1 dsCount=0 obj=0x41a76178 self=0x507837a8   | sysTid=545 nice=-2 sched=0/0 cgrp=apps handle=1349936616   | state=S schedstat=( 15368096286 21707846934 69485 ) utm=1226 stm=310 core=0   at com.android.server.power.PowerManagerService.isScreenOnInternal(PowerManagerService.java:~2529)   - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)   at com.android.server.power.PowerManagerService.isScreenOn(PowerManagerService.java:2522)   at com.android.server.wm.WindowManagerService.sendScreenStatusToClientsLocked(WindowManagerService.java:7749)   at com.android.server.wm.WindowManagerService.setEventDispatching(WindowManagerService.java:7628)   at com.android.server.am.ActivityManagerService.updateEventDispatchingLocked(ActivityManagerService.java:8083)   at com.android.server.am.ActivityManagerService.wakingUp(ActivityManagerService.java:8077)   at com.android.server.power.Notifier.sendWakeUpBroadcast(Notifier.java:474)   at com.android.server.power.Notifier.sendNextBroadcast(Notifier.java:455)   at com.android.server.power.Notifier.access$700(Notifier.java:62)   at com.android.server.power.Notifier$NotifierHandler.handleMessage(Notifier.java:600)   at android.os.Handler.dispatchMessage(Handler.java:107)   at android.os.Looper.loop(Looper.java:194)   at com.android.server.ServerThread.run(SystemServer.java:1328) |

从trace文件看，是因为TID为24的线程等待一个TID为12的线程持有的锁，TID为12的线程等待一个TID为85的线程持有的锁，而TID为85的线程确等待一个TID为24的线程持有的锁，导致了循环等待的现象，对应的trace文件的语句如下：

TID 24：- waiting to lock <0x41a874a0> (a com.android.server.am.ActivityManagerService) held by tid=12 (android.server.ServerThread)

TID 12： - waiting to lock <0x41a7e2e8> (a java.lang.Object) held by tid=85 (Binder\_B)

TID 85：- waiting to lock <0x41a7e420> (a com.android.server.power.PowerManagerService$DisplayBlankerImpl) held by tid=24 (PowerManagerService)

    既然是死锁，那么先看各线程都有那些锁。

先看TID=24的线程的栈顶，ActivityManagerService的broadcastIntent函数代码如下:

public final int broadcastIntent(IApplicationThread caller,

Intent intent, String resolvedType, IIntentReceiver resultTo,

int resultCode, String resultData, Bundle map,

String requiredPermission, boolean serialized, boolean sticky, int userId) {

enforceNotIsolatedCaller("broadcastIntent");

synchronized(this) {

intent = verifyBroadcastLocked(intent);

final ProcessRecord callerApp = getRecordForAppLocked(caller);

final int callingPid = Binder.getCallingPid();

final int callingUid = Binder.getCallingUid();

final long origId = Binder.clearCallingIdentity();

int res = broadcastIntentLocked(callerApp,

callerApp != null ? callerApp.info.packageName : null,

intent, resolvedType, resultTo,

resultCode, resultData, map, requiredPermission, serialized, sticky,

callingPid, callingUid, userId);

Binder.restoreCallingIdentity(origId);

return res;

}

可以看到TID=24需要ActivityManagerService这个锁。再看TID=12线程的栈顶，PowerManagerService的isScreenOnInternal函数代码如下：

private boolean isScreenOnInternal() {

synchronized (mLock) {

return !mSystemReady

|| mDisplayPowerRequest.screenState != DisplayPowerRequest.SCREEN\_STATE\_OFF;

}

}

可以看到需要PowerManagerService的mlock这个锁。最后看TID=85线程的栈顶，同样在PowerManagerService里面，内部类DisplayBlankerImpl的toString函数：

public String toString() {

synchronized (this) {

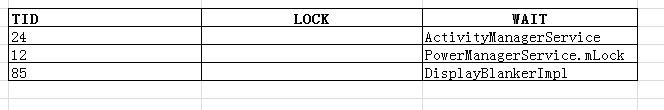
return "blanked=" + mBlanked;

}

}

这是在内部类DisplayBlankerImpl里面实现的，所以需要DisplayBlankerImpl这个锁。

对应的表格如下：



各线程等待的锁情况

死锁的原因是持有锁，又等待锁，下面我们要填充LOCK列，死锁原因就一目了然了。

跟踪TID=24线程的堆栈，在PowerManagerService内部类**DisplayBlankerImp**l的unblankAllDisplays函数中持有锁，持有了this就是持有了**DisplayBlankerImpl**

public void unblankAllDisplays() {

**synchronized (this)** {

mDisplayManagerService.unblankAllDisplaysFromPowerManager()}

}

跟踪TID=12线程的堆栈，在ActivityManagerService的wake\_up函数中持有锁，持有了this就是持有了**ActivityManagerService**

public void wakingUp() {

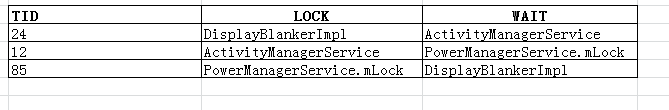
synchronized(this) { } }

跟踪TID=85线程的堆栈，在PowerManagerService的dump有持有锁**mLock**的操作

protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {

synchronized (mLock) {}

因此得到以下的变量lock和wait图，可以清晰的看出死锁了，资源循环等待了。



表二 各线程锁的情况

分析方法：先填充wait列的同步变量，然后在每个线程执行的堆栈的方法中，找出这些同步变量，添加到lock里面就好！

[非常经典的一个案例：Android ANR问题原因分析](http://blog.csdn.net/sinat_22657459/article/details/52780759)

[ANR问题的分析流程](http://blog.csdn.net/jiangguohu1/article/details/52636470)

### 如何避免

* 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；

long currTime = SystemClock.uptimeMillis();

if (currTime - lastUpdateTime >= MIN\_RATE&&!progressModel.isDone()) {

lastUpdateTime = currTime;

* 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。

### ANR监控

发生ANR的体验非常糟糕，因此有必要掌握外网发生ANR的情况。ANR的监控原理和卡顿监控的原理类似，我们知道，卡顿监控可以通过一个子线程向主线程发消息，通过时间差来判断是否发生卡顿，如图5-3所示。卡顿监控详见2.8.2节。

从图5-3中可以看到，是否卡顿取决于T2-T1的时间，发生ANR后，只有T1的时间，没有T2的时间，发生消息后，由于主线程卡住导致消息一直没有执行，这时就不会再回到打印T2的时间戳了，因此，可以使用这个原理监控和上报ANR。



图5-3　卡顿监控原理

在代码清单2-10的基础上，为卡顿监控代码增加ANR的线程监控，在发送消息时，在ANR线程中保存一个状态，主线程消息执行完后再Reset标志位。如果在ANR线程中收到发送消息后，超过一定时间（也就是超时时间），没有复位，就可以认为发生了ANR，这时可以上报相关trace数据给开发人员分析，代码如下：



这个方案有个缺点就是它是通过在主线程监控执行时间的，但不同手机系统的ANR时间不一致，阈值的设定不好掌握，在这个阈值内，主线程还没有处理完，则认为发生了ANR；或者主线程已经发生ANR了，但还要等线程执行完后才能记录该ANR。会有一定概率的误报。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　和卡顿监控一样，数据的上报和Dump信息最好不要在全量版本中打开，否则会影响性能。

## 提高后台进程存活率

在Android系统中，应用进程停止运行有以下几个原因。

* + ·用户主动退出。
  + ·Crash异常退出。
  + ·系统通过杀掉进程回收内存。

其中用户主动退出是合理行为，Crash在前面分析过，这里不再重复分析。而系统通过杀掉应用进程回收内存，理论上这也是合理的形为。因为Android有一个淘汰机制，这个机制会根据应用的运行状态设置一个进程的优先级，然后根据系统整体内存使用状态进行调整，当内存占用达到一定阈值时，就需要把一些优先级低的进程杀掉，以保证其他进程能够有足够的内存使用。

但是杀掉应用进程对体验会有一些影响，比如杀掉后，用户再打开应用耗时会更长（冷启动），如果能从最大程序上提高应用进程的存活率，可以提高热启动的概率，减少冷启动的概率，让用户更快进入应用中，并且提高进程存活率在一些有后台服务进程的应用意义更加明显，比如播放器，如果在后台听歌过程中应用进程被杀，会中断歌曲播放。本节介绍Android系统进程优先级，学习提高应用进程存活的方法。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　降低应用内存开销是减小进程被系统回收概率的一个非常有效的方法，降低应用内存详见第3章。

### 应用进程优先级

在Android系统上，应用进程的优先级（oom\_adj值）分为以下等级，不同定义的进程系统采用了不同的管理机制。

·NATIVE\_ADJ=-17

系统创建的Native进程。

·SYSTEM\_ADJ=-16

系统进程，在运行的过程中永远不会杀掉，如果杀掉可能会导致严重问题。

·PERSISTENT\_PROC\_ADJ=-12

核心进程，系统不会杀掉这类进程，但即使杀掉，影响面也没有SYSTEM\_ADJ进程那么严重。

·PERSISTENT\_SERVICE\_ADJ=-11

正在运行的服务进程，一般不会被杀掉。

·FOREGROUND\_APP\_ADJ=0

前台进程，是指正在前台运行的应用，被杀概率不大。

·VISIBLE\_APP\_ADJ=1

可见进程，用户正在使用，或者有界面在显示，除非出现异常，否则系统不会杀掉这类进程。

·PERCEPTIBLE\_APP\_ADJ=2

可感知的进程，虽然不在前台，但应用进程还在状态，系统除非到内存非常紧张才会杀掉这类进程，比如播放音乐的应用。

·BACKUP\_APP\_ADJ=3

正在备份的进程。

·HEAVY\_WEIGHT\_APP\_ADJ=4

高权重进程。

·SERVICE\_ADJ=5

有Service的进程。

·HOME\_APP\_ADJ=6

与Home有交互的进程，比如有桌面部件和应用正在通信，widget小挂件之类，一般尽量避免杀掉此类进程。

·PREVIOUS\_APP\_ADJ=7

切换进程，可以理解为从可见进程切换过来的进程的状态。

·CACHED\_APP\_MIN\_ADJ=8

缓存进程，也就是空进程。

·SERVICE\_B\_ADJ=9

不活跃的进程。

·HIDDEN\_APP\_MIN\_ADJ=15

缓存进程，空进程，在内存不足的情况下会被优先杀掉。

·UNKNOWN\_ADJ=16

最低级别进程，只有缓存的进程，才有可能设置成这个级别。

当内存不足时，进程优先级低的（oom\_adj越大的）、占内存大的App进程将会被优先杀掉，系统杀进程的规则如下：

·进程优先级设置为PERSISTENT\_PROC\_ADJ（-12）被杀概率较低。

·进程优先级HEAVY\_WEIGHT\_APP\_ADJ（4），这种是Activity仅次于主进程，系统认为是高权重进程。

·前台进程FOREGROUND\_APP\_ADJ（0）不会被杀掉。

·当Activity、Service的生命周期发生变化时都会调整进程的优先级（oo\_adj）。

·进程中没有任何Activity存在会优先被杀。

·空进程最容易被杀。

Android系统进程管理机制的出发点是很好的，将一些空进程和挂起进程杀掉，从用户体验上来讲不会有太大影响，但是在部分应用的特殊场景下，如消息推送和后台播放，需要应用在后台时也尽量不要杀掉，因此有时需要适当提高或维持进程的高优先级，以免被系统所杀。

### 利用SyncAdapter提高进程优先级

可以提高进程优先级使应用在后台的存活时间更长，一般使用以下几种方法实现。

（1）网络连接

通过长连接心跳和进程保持通信，使进程保持活动状态，但是如果系统内存非常紧张，也有可能被杀。

（2）利用系统现有机制

一般可以注册系统消息（AlarmReceiver、BootReceiver等），通过系统消息响应挂起进程。

（3）SyncAdapter

利用Android系统提供的账号同步机制实现进程优先级提高。

前两种使用比较广泛，这里主要介绍第三种：SyncAdapter的使用方法。SyncAdapter是一个系统服务，通过系统的定时器更新应用程序数据ContentProvide r，因为Sync服务工作在独立进程，并且由操作系统调度，进程属于核心进程级别，系统不会杀掉，而使用了SyncAdatper的进程优先级本身也会提高，从而降低应用进程被杀的概率，下面通过具体代码实现这个机制。

1.创建ContentProvider实现数据同步

首先通过一个ContentProvider来实现数据同步，这个示例不会有具体的数据同步，只建立一个空的ContentProvider，代码如下：





同时在AndroidMainfest.xml中注册，代码如下：



2.关联SyncAdapter通信服务

因为SyncAdapter是独立进程，所以要应用实现自己的Binder接口，不过Android系统提供了很简单的抽象类AbstractThreadedSyncAdapter来实现这个AIDL接口，实现进程间通信，代码如下：

|  |
| --- |
| public class TestSyncService extends Service {  private static final Object sSyncAdapterLock = new Object();  private static TestSyncAdapter mSyncAdapter = null;  @Override  public void onCreate() {  synchronized (sSyncAdapterLock) {  if (mSyncAdapter == null) {  mSyncAdapter = new TestSyncAdapter(getApplicationContext(), true);  }  }  }  @Override  public IBinder onBind(Intent intent) {  return mSyncAdapter.getSyncAdapterBinder();  }  static class TestSyncAdapter extends AbstractThreadedSyncAdapter {  public TestSyncAdapter(Context context, boolean autoInitialize) {  super(context, autoInitialize);  }  @Override  public void onPerformSync(Account account, Bundle extras, String authority, ContentProviderClient provider, SyncResult syncResult) {  //具体的同步操作，这里主要是为了提高进程优先级  //getContext().getContentResolver().notifyChange(XXAccountProvider.CONTENT\_URI, null, false);  }  }  } |

关联SyncAdapter进程后，只需重写AbstractThreadedSyncAdapter中的onPerformSync方法。

当然需要在AndroidMainfest.xml中声明这个服务，需要注意的是，这个服务必须exported，并且必须指定android.content.SyncAdapter为过滤器，这样系统才可以在启动时注册这个服务，代码如下。









3.使用Sync服务

1）通过ContentProvider实现手动更新，代码如下：





2）添加账号。



建立账号系统可参考官方文档：[https://developer.android.com/training/s ync-adapters/creating-authenticator.html](https://developer.android.com/training/s%20ync-adapters/creating-authenticator.html)

3）设置同步周期。



关联SyncAdapter后，进程的优先级变为1，仅仅低于前台正在运行的进程，因此可以降低应用被系统杀掉的概率。

## 本章小结

提高应用进程存活率最好不要为了提高而提高，如果不遵循系统的管理机制，采用一些“黑科技”的手段提高进程的存活率，会消耗移动设备更多的电量，因此需要谨慎考虑。减少应用内存消耗，合理使用系统管理机制，才是避免进程被系统杀掉最好的方法。

而Crash与ANR，出现的大部分场景不可预知，也不全是必现的问题，特别有些概率性的Crash在测试阶段不一定能发现，所以需要完善的监控机制记录发生问题的点以及有效的日志信息，再修复这类问题。同时在开发阶段提高开发者的技术水平，合理的代码监控与审查机制，将这类问题发生的概率降到最低。

做了稳定性优化，用户不一定能明显地感受到。但如果不做，在用户使用过程中碰到此类问题对体验来说是灾难性的，另外类似的一种优化也有这种情况，做了用户不一定能感受得到，但不做会或多或少带来一定的影响，这个优化就是耗电的优化。下一章，通过分析Android系统最耗电的几大模块，以及相应的优化方法，逐步了解和掌握优化耗电的方案。

## QA

为啥有多个main线程？

超时机制实现原理？跟看门狗有用么？

网上没有找到一个Android死锁了例子？难道有什么保护？

自己的代码加入了主线程

为啥我的死锁代码，分析不出死锁原因？

ui耗时分析还是不是很明确

## REF

[实现ANR原理源码分析](http://www.jianshu.com/p/8964812972be)

[理解Android ANR的触发原理](http://gityuan.com/2016/07/02/android-anr/)

# 耗电优化

在移动设备中，电池的重要性不言而喻，没有电什么都干不成。对于操作系统和设备开发商来说，耗电优化一直没有停止，去追求更长的待机时间，而对于一款应用来说，并不是就可以忽略电量使用问题，特别是那些被归为“电池杀手”的应用，最终的结果就是被卸载，影响产品的口碑，特别是在Android平台上，由于应用程序可以无限制地注册系统事件（BroadcastReceiver），在这些事件中被不停地唤醒，即使应用已经退到后台，做了很多毫无必要的事情，却导致电量消耗。因此，应用开发者在实现需求的同时，需要尽量减少电量的消耗。

本章学习在Android系统上如何测量应用的耗电情况，以及如何在省电的同时，不影响用户体验的优化方案。

## 耗电检测工具

在Android 5.0以前，在应用中测试电量消耗比较麻烦，也不准确，并且在应用内记录电量消耗本身也是一种耗电操作，唯一可行的方案是使用第三方监测电量的设备，这样才能够获取到真实的电量消耗。Android 5.0引入了一个专门获取设备上电量消耗信息的API：Battery Historian。

Battery Historian是一款由Google提供的Android系统电量分析工具，和Systrace一样，是一款图形化数据分析工具，直观地展示出手机的电量消耗过程，通过输入电量分析文件，显示消耗情况，最后提供一些可供参考电量优化的方法。

### Battery Historian使用步骤

1）初始化Battery Historian，使用以下两个adb命令

adb shell dumpsys batterystats --enable full-wake-history

adb shell dumpsys batterystats --reset

首先需要获取电池数据权限，并reset电池数据，执行成功后，结果如图6-1所示。



2）初始化完成后，操作需要测试电量的一些场景。

3）保存数据。

经过第2）步的操作后，运行以下命令将bugreport信息保存为bugreport.txt文件。(默认在C:\Users\key\bugreport)

adb bugreport > bugreport.txt

成功后打开bugreport.txt，可以看到应用的耗电数据，虽然数据很详细，但可读性非常差，因此需要生成可读性更高的html格式。

4）生成HTML报告。

将historian.py脚本转换成HTML，命令如下：

python historian.py -a bugreport.txt > battery.html

因为historian.py脚本是用Python写的，所以需要python环境，historian.py脚本可以[从github上下载，下载地址为](https://github.com/google/battery-historian)。

5）使用Chrome打开生成的HTML文件，即可查看详细的报告，如图6-2所示。

### 报告参数解析

从图6-2可以看到具体的耗电情况，这个页面和SysTrace工具生成的文件很类似，可读性比第4）步生成的数据文件高很多，但和其他图形化工具一样，虽然可以很好地看出各种数据，但是是否正常，以及异常后如何找到原因，还需要具体分析各项指标，下面我们来了解下各项指标的意义。

在图6-2中，横坐标表示时间周期，以60s为一个时间周期，刻度上的时间单位是秒，但实际上坐标间隔可以根据总测量时间的长度发生改变，所以要以实际情况为准。其中起始时间就是前面初始化的时间（完成第一步），结束时间是获取bugreport的时间（完成第四步）。而纵坐标就是各项具体的指标，详情如表6-1所示。



图6-2　使用Chrome打开Battery Historian文件



表6-1　Android 5.0 Battery Historian属性

以上是在Android 5.0上可以获取到的电池相关数据信息，可以查看的数据非常多，因此需要结合应用的实际场景分析，而Android 6.0又增加了几个属性，如表6-2所示。



表6-2　Android 6.0新增属性

通过分析这些属性数值的变化，可以看出具体的耗电在哪些模块，根据几款机型的测试，以及网上公布的一些数据，移动设备上的耗电基本集中在三个模块：显示（屏幕）、CPU、网络模块（3G/3G Module），图6-3是一款主流Android Pad的电量消耗图。接下来探讨这三大模块的耗电优化方案。



图6-3　Android Pad电量消耗分布

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　如果要求电量消耗的数据精确度非常高，可以采用专业的物理设备检测，因为这类设备的采样率高，数据会更精确。

## 三大模块省电优化

### 显示

从图6-3可以看出，显示屏幕的耗电占比超过了总耗电的四分之一，而屏幕显示的耗电对于应用程序来说基本上没有优化的空间，但是由于显示屏幕的材质不同，对于显示元素的类型有不同的电量消耗。

屏幕作为移动设备组成中最为重要的一部分，其显示效果会直接影响到整机的体验。目前关于屏幕概念的名词非常多（如AMOLED、Retina、TFT等），让人难以区分，如表6-3所示。很多人会认为它们代表屏幕的材质，其实不是，从技术分类上讲，目前主流的移动设备屏幕材质都可归结为两类：LCD与OLED。



表6-3　屏幕类型

LCD与OLED这两种屏幕从发光方式上有着本质的区别，一种是靠外部光源照亮，一种是自发光。LCD在背光灯泡亮度一定的情况下，不论显示什么颜色，耗电是相同，并不会有区别，**而OLED是自发光，在显示不同颜色时**，每个LED单元的功耗不同，于是整体功耗不同。

**因此这里的结论是，在OLED上，深色会比浅色（亮色）更省电，而在LCD上影响耗电的只有亮度。**

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　在应用内调节亮度不太现实，一般由系统控制，在设计页面时可以考虑在不影响美观的情况下，优化使用深色，当然还是以应用的实际场景考虑。

### 网络

目前移动设备的网络连接方式分为移动运营商网络和Wi-Fi，通常情况下，使用Wi-Fi连接网络时的功耗要低于使用移动网络的功耗。

使用移动网络传输数据，电量的消耗有以下3种种状态：

·Full power：高功率状态，移动网络连接被激活，允许设备以最大的传输速率进行操作。

·Low power：低功耗状态，对电量的消耗差不多是Full power状态下的50%。

·Standby：空闲态，没有数据连接需要传输，电量消耗最少。

图6-4是一个典型的移动网络状态转换图，在三种不同状态下电量的消耗不同，如果要达到省电的目的，应尽量保持在空闲或低功耗状态下。

从图6-4可以看出，从低功耗状态转换到高功率状态大约花费1.5秒，而从空闲状态到高功耗大概花费2秒。在应用中每创建一个新的网络连接，网络（射频）模块都会转换到高功率状态（Radio Full Power），在数据传输完后再转回低功耗状态（Radio Low Power），转换的过程需要5秒，这5秒的耗电量保持在高功率状态，最后再转换空闲态需要12秒。因此，对于一个典型的移动网络设备，每个数据传输都会导致网络模块消耗20秒的电量。



图6-4　3G模式下的状态流转图

而Wi-Fi的耗电与包率（每秒发送和接收的包数）和通道率（网速）这两个因素有关，Wi-Fi组件在active状态下有4种模式：低功率、高功率、低传输、高传输。当Wi-Fi组件从低或高功率状态开始传输数据时，短暂进入相应的低或高传输状态，发送完毕就回到之前相应的状态。进行高速传输时，Wi-Fi组件在高传输状态维持的时间非常短。而在低传输状态下维持的时间更短。

因此根据前面描述影响网络连接耗电的情况，可以使用以下几个方案优化网络连接，降低电量消耗。

1）使用Wi-Fi传输数据时，应尽可能增大每个包的大小（不超过MTU）并降低发包的频率。

2）在蜂窝移动网络下，最好做到批量执行网络请求，尽量避免频繁的间隔网络请求，尽量多地保持在Radio Standby状态。

3）尽量在Wi-Fi环境下使用数据传输。

4）使用效率高的数据格式和解析方法，在数据格式方面，JSON和Protobuf效率明显比XML好很多，XML和JSON大家都很熟悉，Protobuf前面的章节也讲到过，是一种语言无关、平台无关、扩展性好的用于通信协议、数据存储的结构化数据串行化方法。

5）压缩数据格式，比如采用GZIP压缩，虽然解压需要消耗更多的CPU，但可以明显提高下载速度，使网络传输更快结束，可以节省更多的电量，并提高了数据的获取速度。

这些优化，除了可以达到省电的目的，也使上传/下载更快，上传/下载速度提高，数据刷新和操作响应也能得到提高，这也是提高速度的一种方式。

### CPU

CPU是智能移动设备的核心，也是最繁忙的硬件模块，要减少CPU的耗电量，也就是要保持CPU在最合适的状态，首先了解几个CPU相关的名词。

1.CPU频率

CPU频率表示CPU的计算能力。频率决定了CPU周期的大小，它们互为倒数。CPU周期就是每条CPU指令执行的最小单位（1条指令可能需要多个CPU周期）。如果应用程序编译后的指令越多越复杂，那么需要的CPU周期数就越多。这也意味着，执行相同周期数时，CPU主频越快，耗时就越短。

2.CPU时间片

在计算机中，每隔N个高电频脉冲，时钟计数器就加1，可以把自然时间分成固定的小块，每一块就叫一个时间片，大部分移动设备的CPU时间片是10ms，单位是jiffies。于是，机器时间都转化为用时间片的个数来衡量。

3.CPU利用率

CPU时间分为用户态、系统态和空闲态，分别表示CPU处于用户态执行的时间、系统内核执行的时间和空闲系统进程执行的时间。当没有用户进程需要执行时，CPU就执行系统缺省空闲进程。我们所指的CPU利用率是指CPU执行非系统空闲进程的时间除以CPU总的执行时间。在Linux内核中有个全局变量Jiffies，表示系统执行最小时间片的个数。CPU利用率就是用执行用户态+系统态的Jiffies除以总Jiffies来表示。

**4.变频**

为了省电及更高效地管理CPU，Linux内核中提供了performance、powersave、userspace、conservative和ondemand五种变频模式供用户选择使用，它们在选择CPU合适的运行频率时，使用的是各自不同的标准并分别适用于不同的应用场景。

·Ondemand：是最常见的CPU调速器，几乎所有的官方内核都默认使用这种调速器。规则很简单，有高需求时，迅速跳到最大频率，有低需求时，迅速降到最小频率。

·Conservative：很常见的调速器，规则是“慢升快降”，注重省电，有高需求时，逐渐提高频率，有低需求时，迅速跳至低频率。

·Interactive：很常见的调速器，规则和Conservation相反，是“快升慢降”，注重响应速度（性能），有高需求时，迅速跳到高频率，有低需求时，逐渐降低频率。

·Lulzactive：较新的一款调速器，根据负载逐级升高或降低频率。

·Powersave：很常见的调速器，正如其名字一样，很省电，因为它的作用就是把频率锁定在设定范围的最小值，负载再高也不升高频率。

·Performance：很常见的调速器，性能最好也最费电，因为它将CPU频率锁定在设定范围的最大值，无论有多少负载，CPU都全速运行。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　因为CPU利用率高不一定会导致CPU运转频率高，这取决于CPU的调频策略，因此网上有些资料说CPU利用率高会更耗电，这种说法是错误的，CPU频率高才更耗电。

CPU在不同的频率下对电量的消耗不同，当然在应用程序中调频需要有Root权限，在有Root权限的情况下，可以使用以下代码实现调频：

|  |
| --- |
| public class CpuFreqSet {  public class CPUFreqSetting {  /\*\*  \* cpu cat命令大全  \* cat [%cpuFreqPath%]/cpuinfo\_cur\_freq (当前cpu频率)  \* cat [%cpuFreqPath%]/cpuinfo\_max\_freq (最大cpu频率)  \* cat [%cpuFreqPath%]/cpuinfo\_min\_freq (最小cpu频率)  \* cat [%cpuFreqPath%]/related\_cpus (cpu数量标号,从0开始,如果是双核,结果为0,1)  \* cat [%cpuFreqPath%]/scaling\_available\_frequencies (cpu所有可用频率)  \* cat [%cpuFreqPath%]/scaling\_available\_governors (cpu所有可用调控模式)  \* cat [%cpuFreqPath%]/scaling\_available\_governors (cpu所有可用调控模式)  \* cat [%cpuFreqPath%]/scaling\_cur\_freq (?????)  \* cat [%cpuFreqPath%]/scaling\_driver (?????)  \* cat [%cpuFreqPath%]/scaling\_governor (?????)  \* cat [%cpuFreqPath%]/scaling\_max\_freq (?????)  \* cat [%cpuFreqPath%]/scaling\_min\_freq (?????)  \* cat [%cpuFreqPath%]/scaling\_setspeed (?????)  \* cat [%cpuFreqPath%]/cpuinfo\_transition\_latency (?????)  \*/  private final String TAG = "CpuFreqSet";  private final String cpuFreqPath = "/sys/devices/system/cpu/cpu0/cpufreq";  // private final static String PERFORMANCE\_GOVERNOR = "performance";  // private final static String POWER\_SAVE\_GOVERNOR = "performance";  // private final static String ONDEMAND\_GOVERNOR = "performance";  // private final static String CONSERVATIVE\_GOVERNOR = "performance";  // private final static String USERSAPCE\_GOVERNOR = "performance";  /\*\*  \* 获取当前CPU调控模式  \*/  public void getCpuCurGovernor() {  try {  DataInputStream is = null;  Process process = Runtime.getRuntime().exec("cat " + cpuFreqPath + "/scaling\_governor");  is = new DataInputStream(process.getInputStream());  String line = is.readLine();  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* 设置CPU调控模式  \*/  private boolean writeCpuGovernor(String governor) {  DataOutputStream os = null;  byte[] buffer = new byte[256];  String command = "echo " + governor + " > " + cpuFreqPath + "/scaling\_governor";  try {  Process process = Runtime.getRuntime().exec("su");  os = new DataOutputStream(process.getOutputStream());  os.writeBytes(command + "\n");  os.writeBytes("exit\n");  os.flush();  process.waitFor();  } catch (IOException e) {  return false;  } catch (InterruptedException e) {  e.printStackTrace();  }  return true;  }  /\*\*  \* 获得CPU所有调控模式  \* @return  \*/  private ArrayList<String> readCpuGovernors() {  ArrayList<String> governors = new ArrayList<String>();  DataInputStream is = null;  try {  Process process = Runtime.getRuntime().exec("cat " + cpuFreqPath + "/scaling\_available\_governors");  is = new DataInputStream(process.getInputStream());  String line = is.readLine();  String[] strs = line.split(" ");  for (int i = 0; i < strs.length; i++)  governors.add(strs[i]);  } catch (IOException e) {  }  return governors;  }  }  } |

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　由于调频的实现需要在Root环境下，因此在应用开发中大部分场景无法使用，这里的代码只是让大家理解CPU的运行模式，CPU的运行模式只能交给系统层维护，在应用开发中，需要注意如何减少CPU的开销，以达到省电的目的。

## 应用常用优化方案

### 计算优化

缩短代码产生指令运行的时间，进而减少某个应用程序对CPU时间片的总占用时间，就能减少单位时间内，该应用程序占整个系统耗电的百分比。

大家都知道，浮点运算比整数运算更消耗CPU时间片，因此耗电也会增加，在编写代码的过程中应该尽量减少浮点运算。

避开浮点运算的优化方法如下：

* ·除法变乘法。
* ·充分利用移位。
* ·查表法，直接使用映射关系，但这会增加内存开销，视具体场景而定。
* ·利用arm neon指令集做并行运算，需要ARM V7及以上架构CPU才能支持。

### 避免WakeLock使用不当

虽然Android会不断关闭各种硬件来延长手机的待机时间，比如经过一段时间未做任何操作时，屏幕会逐渐变暗直至关闭，然后CPU进入低功耗的睡眠状态，系统这些机制都是为了节约更多的电量资源。

然而现实并非如此，由于应用需要满足一些特定的场景，比如一款社交类应用，即使手机处于待机状态，也需要把应用唤醒检查同步新的数据信息，或者播放器停留在看歌词页面。因此即使在这种睡眠状态下，应用为了达到这些目的和需求，它们会不断尝试唤醒手机。在Android操作系统中，最常用的唤醒手机的方法是使用PowerManager.WakeLock来保持CPU工作并防止屏幕自动变暗关闭。这使得手机可以保持在唤醒状态，继续执行相应的工作。PowerManager负责对Android设备电源相关进行管理，WakeLock也是一种锁机制，只要应用中有WakeLock，通过相应参数去获取对应的锁，既可以达到对应的电源管理目的，也可以使系统无法进入休眠阶段。

获取WakeLock比较简单，执行如下代码即可。

|  |
| --- |
| PowerManager.WakeLock mWakeLock = null;  private void acquireWakeLock(Context ctx)  {  if (null == mWakeLock)  {  PowerManager pm = (PowerManager)ctx.getSystemService(Context.POWER\_SERVICE);  mWakeLock = pm.newWakeLock(PowerManager.PARTIAL\_WAKE\_LOCK|PowerManager.ON\_AFTER\_RELEASE, "TestLocknService");  if (null != mWakeLock)  {  mWakeLock.acquire();  }  }  }  //释放设备电源锁  private void releaseWakeLock()  {  if (null != mWakeLock)  {  mWakeLock.release();  mWakeLock = null;  }  } |

如上代码所示，首先需要获取PowerManager的实例pm，然后通过newWakeLock方法获取WakeLock的相应锁，其中第一个参数是指定要获取哪种类型的锁，不同的锁对系统CPU、屏幕和键盘有不同的管理策略，第二个参数是锁的自定义名称。

Android系统中的WakeLock类型有以下几种，它们对CPU、屏幕、键盘的影响有不同的策略。

·PARTIAL\_WAKE\_LOCK：保持CPU正常运转，屏幕和键盘灯有可能会关闭。

·SCREEN\_DIM\_WAKE\_LOCK：保持CPU运转，允许保持屏幕显示但有可能变暗，允许关闭键盘灯。

·SCREEN\_BRIGHT\_WAKE\_LOCK：保持CPU运转，允许保持屏幕高亮显示，允许关闭键盘灯。

·FULL\_WAKE\_LOCK：保持CPU运转，保持屏幕高亮显示，键盘灯也保持亮度。

·ACQUIRE\_CAUSES\_WAKEUP：强制使屏幕亮起，这种锁主要用于一些必须通知用户的操作。

·ON\_AFTER\_RELEASE：当锁被释放时，保持屏幕亮起一段时间。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　使用WakeLock需要注册相应的权限：



拿到WakeLock后，可以保持完成需要完成的事，但完成后，或者离开这个场景后，需要及时释放WakeLock，这点非常重要的，否则会带来不可预估的电量消耗。

WakeLock是应用中为了正常完成相应工作最普遍的做法，但不恰当地使用WakeLock会导致严重的耗电问题，使用Battery Historian工具（使用方法参考6.1）可以查看详细的WakeLock数据，发现WaleLock使用不当带来的耗电问题。

[使用案例](http://landerlyoung.github.io/blog/2014/10/31/androidzhong-de-wakelockshi-yong/)

#### 查看方法

使用adb命令可以查看电量使用情况：

adb shell dumpsys power

|  |
| --- |
| Wake Locks: size=2  PARTIAL\_WAKE\_LOCK 'WifiStateMachine' (uid=1000, pid=453, ws=null)  SCREEN\_BRIGHT\_WAKE\_LOCK 'WindowManagerService' ON\_AFTER\_RELEASE (uid=1000, pid=453, ws=WorkSource{10017}) |

#### WakeLock控制级别

和硬件加速类似，Android也提供了一个WakeLock的级别控制，但是这么做有啥好处呢？目前可以控制Application、Activity、Window和View。

（1）Application级别

在Android Manifest文件中添加属性，整个应用全局使用硬件加速，代码如下：

<application android:hardwareAccelerated="true" ...>

（2）Activity级别

对Activity进行单独控制。可以启动或者禁用一个Activity的硬件加速，即使设置过全局打开硬件加速，也可以在某个Activity中禁用。

<activity android:hardwareAccelerated="false/true" />

（3）Window级别

更细粒度的控制，通过如下代码可以对某个window进行加速：

[代码]

（4）View级别

对单独的View控制硬件加速。

View.setLayerType(View.LAYER\_TYPE\_SOFTWARE, null);

在编写代码中可以检查当前View是否已经硬件加速，可以使用iew.isHardware-Accelerated（），或者Canvas.isHardwareAccelerated（）来得到结果。即使View是存在于一个已经加速的Windows上，仍然可以使用没有硬件加速的Canvas进行绘制。

### 使用Job Scheduler

在应用开发中，有以下一些常见的耗电场景：

·经常为了使一些特殊模块正常工作，而通过唤醒CPU去执行对应程序，但Google测试发现，每次唤醒CPU，即使程序只运行1秒钟，但实际上消耗了大约两分钟的耗电量。

·一些并不重要的数据，在非Wi-Fi环境下上报，如上传日志等。

·在CPU高负荷状态下去做一些数据清理工作。

以上这些操作在应用中是有必要的，但并不紧急，可以选择在合适的环境下完成，因此Google为了提高电池的续航能力，在Android 5.0（API 21）提供了一个JobScheduler组件，只有一系列的预置条件满足时，才执行对应的操作，这样既能省电，又保证了功能的完整性。也就是说可以将不紧急的任务交给Job Scheduler来处理，Job Scheduler集中处理收到的任务，选择合适的时间、合适的网络，再一起进行执行。

在以下场景可以考虑使用Job Scheduler：

·重要不紧急的任务，可以延迟执行，如定期数据库数据更新和数据上报。

·耗电量较大的任务，比如充电时才希望执行的备份数据操作。

·不紧急可以不执行的网络任务，如在Wi-Fi环境预加载数据。

·可以批量执行的任务。

使用JobScheduler主要两个步骤，创建一个JobScheduler对象预设置任务执行条件以及创建Job Service执行具体的任务，接下来详细介绍使用方法。

#### 创建JobScheduler

创建JobScheduler，用来初始化一个JobScheduler以及设置触发JobScheduler执行任务的条件。

1）通过getSystemService（）获取一个JobScheduler的对象：

2）创建一个JobInfo，描述一个任务的执行ID，以及触发这个任务的条件。代码清单6-1，是添加一个任务的方法。

代码清单6-1　添加一个JobSchedule任务

|  |
| --- |
| **public class** JobScheduleManager {  **private static** JobScheduleManager *mInstance* = **null**;  **private** Context **mContext** = **null**;  **private** JobScheduler **mJS** = **null**;   **public static** JobScheduleManager getmInstance(Context ctx) {  **if** (**null** == *mInstance*) {  *mInstance* = **new** JobScheduleManager(ctx);  }  **return** *mInstance*;  }   **public** JobScheduleManager(Context ctx) {  **mContext** = ctx;  **mJS** = (JobScheduler) **mContext**.getSystemService(Context.***JOB\_SCHEDULER\_SERVICE***);  }   **public boolean** addJobScheduleTask(**int** task\_id) {  JobInfo.Builder builder = **new** JobInfo.Builder(task\_id,  **new** ComponentName(**"PackgeName"**,  JobSchedulerService.**class**.getName()));  **switch** (task\_id){  **case** 1:  builder.setPeriodic(1000);  **break**;  **case** 2:  builder.setPersisted(**false**);  **break**;  **default**:   }  **if**(**mJS** != **null**) {  **return mJS**.schedule(builder.build()) > 0;  }**else**{  **return false**;  }  }  *//create JobInfo* } |

从代码清单6-1看到，首先通过JobInfo的build方法创建一个JobInfo，有两个参数，第一个参数task\_id是任务ID，这个ID有两个作用：

·添加任务时，对不同的ID做不同的触发条件，即用switch代码实现。

·在执行时需要根据任务ID执行具体的任务。

第二个参数ComponentName是具体执行JobSchedluer任务的服务，参数为进程包名以及服务类名。

在switch代码段中，针对不同的任务设置不同的触发条件，JobInfo支持以下触发条件：

·setMinimumLatency（long minLatencyMillis）：设置任务的延迟执行时间（单位是ms），需要注意的是，setMinimumLatency与setPeriodic（long time）方法不兼容，同时调用这两个方法会引起异常。

·setOverrideDeadline（long maxExecutionDelayMillis）：设置任务最晚的延迟时间。如果到了规定的时间，其他条件还未满足，这个任务也会被启动。与setMinimumLatency（long time）一样，setOverrideDeadline与setPeriodic（long time）同时调用会引发异常。

·setPersisted（boolean isPersisted）：设备重启之后，任务是否还要继续执行。

·setRequiredNetworkType（int networkType）：只有在满足指定的网络条件时，才会被执行。共有三种类型：

·JobInfo.NETWORK\_TYPE\_NONE：不管是否有网络，这个任务都会被执行（如果未设置，这个参数就是默认参数）。

·JobInfo.NETWORK\_TYPE\_ANY：只有在有网络的情况下，任务才可以执行，和网络类型无关。

·JobInfo.NETWORK\_TYPE\_UNMETERED：非运营商网络（比如在Wi-Fi连接时），时任务才会被执行。

·setRequiresCharging（boolean requiresCharging）：只有当设备在充电时，这个任务才会被执行。

·setRequiresDeviceIdle（boolean requiresDeviceIdle）：只有当用户没有在使用该设备且有一段时间没有使用时，才会启动该任务。

[https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg]注意　setRequiredNetworkType（int networkType）、setRequiresCharging（boolean requireCharging）and setRequiresDeviceIdle（boolean requireIdle）这几个方法可能会使任务无法执行，除非调用setOverrideDeadline（long time）设置了最大延迟时间，使任务在为满足条件的情况下也会被执行。

这样就完成添加一个任务到JobScheduler中。在添加一个任务时，要指定一个完成任务的Service，也就是代码清单6-1中的JobSchedulerService，在任务条件触发后，进入JobSchedulerService执行具体的工作。接下来实现一个JobService，也就是JobSchedulerService。

#### 创建JobSchedulerService

JobSchedulerService只需要继承JobService，并实现相应的方法，执行具体的任务，建议在异步线程,代码如下：

|  |
| --- |
| **public class** JobSchedulerService **extends** JobService {  @Override  **public boolean** onStartJob(JobParameters params) {  **return false**;  }   @Override  **public boolean** onStopJob(JobParameters params) {  **return false**;  } } |

同时需要在AndroidManifest.xml中注册这个服务，代码如下：



JobSchedulerService类必须实现两个方法onStartJob（JobParameters params）和onStopJob（JobParameters params）。

（1）onStartJob（JobParameters params）

任务开始时，执行onStartJob（JobParameters params）方法，系统用来触发已经被执行的任务。任务执行完毕，需要调用jobFinished（JobParameters params，boolean needsRescheduled）来通知系统。

（2）void jobFinished（JobParameters params，boolean needsReschedule）

任务执行完后，调用jobFinished方法通知JobScheduler。

（3）onStopJob（JobParameters params）

系统接收到一个取消请求时，调用onStopJob（JobParameters params）方法取消正在等待执行的任务。如果系统在接收到一个取消请求时，实际任务队列中已经没有正在运行的任务，onStopJob（JobParameters params）不会被调用。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　JobService运行在主线程，如果是耗时任务，一定要使用ThreadHandler或者一个异步任务来运行耗时的操作，以防止阻塞主线程。

## Doze模式

从Android 4.4开始，开始更加重视系统耗电问题，其Wear系统就尝试使用了一些节省电量的模式，在此基础上，Android 6.0推出了两个新的省电技术Doze和App Standby，通过管理应用程序的操作行为延长电池的续航时间。

在Android 6.0系统上运行的应用，不论是否使用API 23 SDK编译的APK，都会受到这两种模式的限制，因此在Android 6.0系统上应用需要根据Doze和App Standby两个省电模式做些兼容性调整，避免应用进入这两种模式后影响应用的体验。

### Doze模式介绍

Doze模式是通过限制应用访问网络以及其他一些操作的频率，来减少CPU的开销，达到省电目的，比如限制应用的网络访问，对异步任务挂起并延时执行等。在Android 6.0系统上，只要满足以下两个条件，就会进入Doze模式：

·未连接电源。

·屏幕关闭。

同时为了保证应用正常工作，系统会周期性地退出休眠模式，在退出休眠模式这个时间里去执行那些在休眠模式下挂起的任务，这个时间窗口被称作维护窗口（maintenance window）。同时随着系统处于Doze模式时长的增长，进入维护窗口的频率会逐步降低。Doze模式执行时序图如图6-5所示。

图6-5　Doze模式执行时序图

如图6-5所示，在进入Doze模式后，CPU的执行情况会降到最低，并随时Doze模式时间的增长，进入维护窗口的频率也在减小。

在Doze模式下，维护了一个状态机，一共5种状态。

·ACTIVE：手机设备处于激活活动状态，设备在使用或者在充电中。

·INACTIVE：刚脱离ACTIVE状态，进入非活动状态，关闭屏幕并没有充电。

·IDLE\_PENDING：IDLE预备态，准备进入IDLE态，在每隔30分钟让应用进入等待空闲预备状态。

·IDLE：设备进入空闲状态。

·IDLE\_MAINTENANCE：处理挂起任务，在此状态可以执行在INACTIVE态挂起的任务（闹钟或者JobSchedule任务）。

如果设备在被使用，状态机就是ACTIVE状态。当用户屏幕状态关闭并且没有连接电源时，状态机进入INACTIVE。在30分钟后（厂商可能会改这个时间），状态机会进入IDLE\_PENDING，再经过一定的时间，设备就会进入IDLE状态，在这个状态下，应用程序的很多行为会被限制，后面设备会在IDLE和IDLE\_MAINTENANCE两个状态之间来回切换，在后者状态下，应用程序的一些事件可以被触发，之后仍然会进入IDLE状态。

在Doze模式中的状态进入IDLE时，有以下行为会受到限制：

·断开网络连接。

·系统忽略Wake Lock。

·标准闹钟AlarmManager定时任务延迟到下一个maintenance window进行处理，如果应用仍需要在Doze时使闹钟事件生效，可以使用setAndAllowWhileIdle（）或setExactAndAllowWhileIdle（）。

·系统不会扫描热点Wi-Fi。

·禁止同步工作。

·停止JobScheduler任务调度。

一旦点亮了屏幕或者插上电源充电，就会退出Doze模式。

### Doze模式应用策略

从6.4.1节的分析可以看到，在Doze模式的IDLE状态时，会有一系列的限制，对应用影响较大的是禁止闹钟和断开网络连接。前面也讲过，如果在Doze模式中仍需要在Doze时使闹钟事件生效，可以使用setAndAllowWhileIdle（）或setExactAndAllowWhileIdle（）。现在大部分应用程序依赖于实时信息，通信应用或者需要PUSH消息，这将严重影响应用的体验。这时可以使用GCM（Google Cloud Messaging）谷歌云推送服务。

GCM是一个云到设备的服务，可以支持实时在云端服务和Android设备上应用程序之间的消息传递。GCM提供了一个持久连接到云端的链接，让所有需要实时消息传递的应用程序可以共享此链接。这个共享链接显著优化电池消耗，使其不必让多个应用程序为维护自己单独的长链接而导致更多的电量消耗。同时GCM消息拥有高优先级，不影响Doze模式，且不会影响其他应用程序的状态。同时最大限度地减少电池在整个系统和设备的影响。

**因为国内无法使用GCM，这里不再讲解如何使用GCM**，如果有需要，可以在官网查询，使用并不复杂。

[https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg]注意　虽然在国内由于某些原因无法使用GCM，但几个大手机厂商都有自己的云推送服务，可以接入，不但可以在Doze模式下推送消息，到达率也会大幅提高。

部分应用由于场景特殊，使用GCM（或其他推送服务）仍不能满足应用的需求，这时可以加入白名单，白名单机制使应用在Doze模式时仍能够访问网络和使用WakeLocks，但是其他的限制仍不会执行，如作业调度、异步操作以及闹钟。

开发者可以通过接口isIgnoringBatteryOptimizations（）来判断当前应用是否处于白名单中。为了让用户能灵活控制以及应用对用户的引导，系统提供了以下功能：

·使用用户可以手动配置白名单，位置在设置→电池→电池优化。

·在应用中通过发送Action为REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS的Intent来调起电池优化配置页面。

·通过添加权限REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS使其可以触发系统对话框，询问用户是否将该应用添加到白名单中。触发添加白名单确认对话框方法是：发送Action为ACTION\_REQUEST\_IGNORE\_BATTERY\_OPTIMIZATIONS。

需要注意的是，用户可以随时在电池优化配置中把App从白名单中移除。

### 测试Doze模式应用工作状态

1）连接USB，运行被测App，保持App在活动状态。

2）关闭设备屏幕。

3）通过下面的adb命令强制系统进入Doze模式：

[代码] adb shell dumpsys battery unplug

adb shell dumpsys deviceidle step

第二条命令需要执行多次才能使系统进入idle状态。

4）退出Doze状态，检查App的工作状态和预期是否一致。

## 本章小结

虽然节省电量这些优化对用户来说感知不大，但如果不做这些优化，用户有很大可能感知到，并且影响手机的电池续航能力。

节省电量的优化，需要使用耗电检测工具对应用测试，数据不准确会使优化难度增加，同时耗电的优化对用户体验会有一定的影响，因此在优化的同时，一定要尽量降低对应用体验的影响，找到最佳的平衡点。

# 安装包大小优化

应用安装包大小对应用的使用没有影响，但应用的安装包越大，用户下载的门槛越高，特别是在移动网络情况下，用户在下载应用时，对安装包大小的要求更高，这点在游戏类应用上更加突出。从一些统计网站的数据上看，一款游戏的安装包越来越大，玩家的用户转化率也在逐步降低（当然明星级应用影响不大）。同时减小安装包大小不但可以降低用户下载的门槛，也能减小安装包下载的带宽成本。

因此，减小安装包大小可以让更多的用户愿意下载和体验产品，在减小应用安装包大小之前，先了解安装包（APK文件）的构成，然后对每个组成部分做相应的优化工作。

## 应用装包的构成

Android应用工程使用Android SDK编译代码，并且把所有的数据和资源文件打包成一个APK（Android Package）文件，后缀名为.apk。APK文件包含Android应用程序的所有内容，是Android平台用于安装应用程序的文件。APK是一个压缩文件，解压缩APK文件后看到如图7-1所示的目录结构。



图7-1　APK文件目录

在AS下双击apk直接可以看到目录结构

1.assets

assets目录可以根据应用需求存放任何文件夹架构，如配置文件、资源文件（如WebView本地资源、图片资源等），这些文件的内容在程序运行过程中可以通过AssetManager类获得。和res的不同点在于，res目录下的文件会在.R文件中生成对应的资源ID，assets不会自动生成对应的ID，而是通过AssetManager类的接口获取。

2.lib

该目录存放应用程序依赖的用C/C++编写的native库文件，该目录下可以包含4种类型，根据CPU类型的不同加载不同目录下的so库，这4种类型分别为ARM（ARM架构）、ARM-v7a（ARM-V7架构）、MIPS（MIPS架构）、X86（X86架构）、APK包的lib目录构成如图7-2所示。



图7-2　lib目录结构

如图7-2所示，不同的CPU架构设备在应用程序运行时，根据CPU架构类型加载对应的目录，每个目录可以存放很多对应版本的so库，同时这个目录结构固定，用户必须严格按照这个目录存放自己的so库。

[https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg]提示　目前的移动设备大部分是基于ARM或者ARM-V7a架构的，X86和MIPS架构的移动智能终端比较少，并且X86的设备基本都兼容了ARM指令集（但会影响功耗），所以在应用程序中，native的程序使用频率不高或者运算并不复杂的情况下，只包含ARM和ARM-V7a的so，如果不需要用到neon指令集，就只需要包含ARM架构下编译的so即可。

3.res

res是resource的缩写，这个目录存放资源文件，在这个文件夹下的所有文件都会生成对应的ID映射到Android工程的.R文件中，访问时可以直接使用资源ID，即R.id.filename，res文件夹可以包含多个文件夹，这些文件夹由两个纬度区分：资源的类型和设备硬件的类型，如图7-3所示。



图7-3　典型res目录文件夹架

随着采用Android技术且配置各异的设备越来越多，这些资源的重要性也日益增加。为了提供与不同配置的兼容性，必须使用各种按类型和配置对资源进行分组的子目录，如图7-3中的mipmap-xhdpi、mipmap-xxhdpi等目录。因此可以得到一个结论，如果依靠这类方法使应用达到更好的兼容性，需要增加更多资源，这样会导致安装包增长。

因此，既要减少安装包大小，又要使应用体验不受到影响，可以考虑在代码中实现自适应，但有些特别要求的资源文件还是要通过这个方法达到最好的兼容性。可以考虑优先提供xhdpi的图片，根据需要提供有差异的部分，看是否有必要在mdpi、ldpi或者xxxhdpi中提供。

4.META-INF

保存应用的签名信息，签名信息可以验证APK文件的完整性。Android SDK在打包APK时会计算APK包中所有文件的完整性，并且把这些完整性保存到META-INF文件夹下，应用程序在安装时首先根据META-INF文件夹校验APK的完整性，这样可以保证APK中的每一个文件都不能被篡改。以此来确保APK应用程序不被恶意修改或者病毒感染，有利于确保Android应用的完整性和系统的安全性。META-INF目录下包含的文件有CERT.RSA、CERT.DSA、CERT.SF和MANIFEST.MF，其中CERT.RSA是开发者利用私钥对APK进行签名的签名文件，CERT.SF、MANIFEST.MF记录了文件中文件的SHA-1哈希值。

5.AndroidManifest.xml

Android应用程序的配置文件是用来描述Android应用“整体资讯”的设定文件，简单来说，相当于Android应用向Android系统“自我介绍”的配置文件。Android系统可以根据这个“自我介绍”完整地了解APK应用程序的资讯，每个Android应用程序都必须包含一个AndroidManifest.xml文件，且它的名字固定的，不能修改。在开发Android应用程序时，一般都在AndroidManifest.xml中注册代码中的每个Activity、Service、Provider和Receiver，只有这样系统才能启动对应的组件，另外这个文件还包含一些权限声明以及使用的SDK版本信息等。程序打包时，会简单编译AndroidManifest.xml，便于Android系统识别，编译之后的格式是AXML格式，其中包括以下内容：

1）axml头：其中的axml头是固定标识axml文件的，其值固定为0x00080003。

2）axml文件长度：标识AXML文件的大小。

3）StringDataSegment：XML文件中所有字符串类型保存在此。

4）ResourceIdSegment：XML文件中声明的资源文件ID保存于此。

5）XmlContentSegment：是XML的内容段，按照XML文件中的结构依次排开，保存XML的数据内容。

6.classes.dex

Java可执行程序，需要先把Java文件编译成class文件，字节码都保存在class文件中，Java虚拟机可以通过解释并执行这些class文件。而Dalvik虚拟机在Java虚拟机进行了优化，执行的是Dalvik字节码，这些Dalvik字节码由Java字节码转换而来，一般情况下，Android应用在打包时通过AndroidSDK中的dx工具将Java字节码转换为Dalvik字节码。dx工具可以对多个class文件进行合并重组、优化，达到减小体积、缩短运行时间的目的。

7.proguard.cfg

代码混淆配置文件。

8.rsources.arsc

记录资源文件和资源ID之间的映射关系，用来根据资源ID寻找资源。Android的开发是分模块的，res目录专门用来存放资源文件，在代码中需要调用资源文件时，只需要调用findviewbyId（）就可以得到资源文件，每当在res文件夹下放一个文件，aapt就会自动生成对应的ID保存在.R文件中，调用这个ID就可以，但是只有这个ID还不够，.R文件只是保证编译程序不报错，实际上在程序运行时，系统要根据ID寻找对应的资源路径，而resources.arsc文件就是用来记录这些ID和资源文件位置对应关系的文件。

由此可见，APK中的classes.dex、lib、资源文件是大头，APK瘦身主要就是优化这三类，下一节讲解具体的优化方案。

## 减少安装包大小的常用方案

本节讲述一些常用的减小安装包技巧，但需要结合实际情况，在不影响用户体验的情况下进行取舍。

### 代码混淆

ProGuard是一个开源的Java代码混淆器（obfuscation），并且默认集成到Android SDK中，它具有以下功能：

·压缩：移除无效的类、属性、方法等。

·优化：移除没用的结构。

·混淆：把类名、属性名、方法名替换为一到两个字母。

#### ProGuard常用参数

ProGuard配置参数非常多，以下列出一些常用的参数，[具体的参数可参考](http://proguard.sourceforge.net/manual/usage.html#classspecification)

（1）ProGuard配置





（2）保留选项





（3）压缩



（4）优化





（5）混淆





#### 在Android Studio中使用ProGuard

按照上一节的语法规则编写proguard-rules.pro后，需要在build.gradle中配置，同时设置minifyEnabled为true，在编译工程时自动混淆代码，代码如下：





### 资源优化

1.使用Android Lint删除冗余资源

前面讲过的ProGuard只能对代码进行分析和优化，但对资源文件就没有类似的效果了。2.3.1节讲述过Android Lint工具用来扫描布局层级，Android Lint也能扫描到没有使用的资源文件。下面介绍使用Lint检查冗余资源文件的流程。

1）单击菜单栏中的Analyze→Run Inspection by Name，弹出工具选择框，如图7-4所示。

2）在弹出的输入框中Unused resources，回车进入扫描前目录设置对话框，如图7-5所示。



图7-4　打开资源扫描对话框



图7-5　扫描设置对话框

3）选择需要扫描的目录或者整个工程，单击OK按钮开始扫描。

4）扫描完成后生成报告，可在Android Studio中直接查看并删除确认无用的资源文件，如图7-6所示。



图7-6　扫描结果

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　Android Lint不会分析assets文件夹下的资源，因为assets文件可以通过文件名直播访问，不需要通过具体的引用，Lint无法判断资源是否被用到。

2.资源文件最少化

从图7-1中可以看到系统根据不同分辨率会自动加载不同图片资源（ldpi、mdpi、tvdpi、hdpi、xhdpi、xxhdpi等），如果为了适配不同屏幕，每个文件夹都放一套图，对安装包大小的影响非常严重，虽然Android支持这么多的屏幕密度，但是不代表应用为每一个屏幕都提供一套资源，在这方面可以参考以下几个建议：

·尽量使用一套图片资源，遇到一些图片在不同分辨率手机上变化差异过大的情况时，再考虑在相应文件夹下放入这个特定的图片。

·使用一套图、一套布局、多套dimens.xml文件，在使用最小资源的情况下，解决多分辨率适配。

·使用轻量级的第三方库。

·减少项目中的预置图片，预置图片改成有服务器下发，尽可能地将程序与资源分离。

在图7-1中可以看到lib目录下也有多个文件夹，这些也不是必须都要适配，原因和方案参考7.1节。

3.图片优化

Android应用的资源中主要用到两种格式的图片：PNG和JPG。PNG（Portable Network Graphic Format）用来存储灰度图像时，灰度图像的深度可多到16位，存储彩色图像时，彩色图像的深度可多到48位，并且可存储多达16位的透明通道数据。JPG（JPEG）图片以24位颜色存储单个位图。JPEG是与平台无关的格式，支持最高级别的压缩，但这种压缩有损耗，并且没有透明通道。

在开发Android应用时，PNG资源图片在应用打包过程中，资源打包工具AAPT（Android Asset Packaging Tool）自动对PNG图片做压缩处理，但减小的大小比较有限，除了靠工具自动打包，一般还能采用以下两种方式。

（1）降低图片色彩位数

如果使用的图片色彩比较单一，可以降低图像位数来减少图片大小。

（2）PNG图片压缩工具

PNG格式的压缩工具很多，比较主流的有Zopfli和PNGoo。Zopfli是Google官方的一个开源项目，使用了deflate压缩算法。PNGoo工具使用更简单，可以手动指定单个图片压缩，也可以写脚本批量压缩图片，在网上可以找到具体使用方法。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=742396fc29b0e6bb918ccbc448ddc3bb.jpg提示　对于非透明的大图，JPG文件会比PNG文件小很多，通常会减小到一半以上。一般使用于应用中的一些闪屏介绍页，全图背景等。

### 其他优化

除了前面几节讲述的优化方案以外，还可以根据应用需求尝试在以下几个方面优化：

1.避免重复功能的库

在应用开发中，比如几种图片开源库，可能一个开源库不能支持所有业务功能，而需要用到多个开源库，若一个应用放了两个图片库，不但增加了安装包大小，也增加了维护成本，此时，建议选择性价比更高、更适合当前应用业务需求的一个，在这个基础上增加相应的功能，扩展当前的图片库，而非引入新的开源库。

2.使用WebP图片格式

相对于PNG和JPG格式，WebP是一种新的图片格式，它支持透明度，压缩比比JPG更高，但效果不比JPG差，但由于只有Android 4.2.2才很好地支持WebP格式（Android 4.0支持但不包含透明度），使用的话需要注意，最好考虑在不支持的系统版本上增加解码能力。

3.插件化

插件化除了可以解决方法数超标的问题，同时如果插件放到服务器，在需要时再下载下来，可以减少安装包大小。但是需要注意的是，因为使用时需要临时下载，所以建议在用户使用率不高的功能模块上使用，或者使用预加载。

## 本章小结

通过本章介绍的一系列减包方案，可以从一定程度上减小安装包，但有些方案可能对体验有损，比如图片的压缩，所以需要更好地权衡。

## 参考

[Android傻瓜式分包插件](https://github.com/TangXiaoLv/Android-Easy-MultiDex)

## multidex解决方法数越界

Android中单个dex文件所能包含的最大方法数为65536, 这包含了FrameWork, 依赖的jar包以及应用本身的代码中的所有方法. 会爆出:

com.android.dex.DexIndexOverflowException: method ID not in[0, 0xffff] :65536

在一些低版本的手机, 即使没有超过方法数的上限却还是出现错误:

E/dalvikvm: Optimization failed

E/installd: **dexopt** failed on '/data/dalvik-cache/.....'

原因： 这在大型应用会遇到，首先**dexpot**是一个程序, 应用在安装时, 系统会通过dexopt来优化dex文件, 在优化过程中dexopt采用一个固定大小的缓冲区来存储应用中所有方法信息, 这个缓冲区就是linearAlloc. LinearAlloc缓冲区在新版本的Android系统中大小为8MB或者16MB. 在Android 2.2和2.3中却只有5MB. 这是如果方法过多, 即使方法数没有超过65535也有可能会因为存储空间失败而无法安装.

解决方案

插件化: 是一套重量级的技术方案, 通过将一个dex拆分成两个或者多个dex,可以在一定程度上解决方法数的越界问题. 但是还有兼容性问题需要考虑, 所以需要权衡是否需要使用这个方案.

multidex: 这是Google在2014年提出的解决方案

### multidex

#### 使用方法

在Android 5.0之前使用multidex需要引入android-support-multidex.jar包，从Android 5.0开始，系统默认支持了multidex，它可以从apk中加载多个dex。Multidex方案主要针对AndroidStudio和Gradle编译环境。

使用Multidex的步骤：

* 1. 在build.gradle文件中android-> defaultConfig添加multiDexEnabled true
  2. 添加对multidex的依赖compile 'com.android.support:multidex:1.0.0'
  3. 在代码中添加对multidex的支持：在super.attachBaseContext(base)后添加MultiDex.install(this)，因此这里有三种方案：
* 在AndroidManifest文件中指定Application为MultiDexApplication
* 让应用的Application继承自MultiDexApplication
* 重写Application的attachBaseContext方法，这个方法要先于onCreate方法执行

public class TestApplication extends Application {

@Override

protected void attachBaseContext(Context base) {

super.attachBaseContext(base);

MultiDex.install(this); //

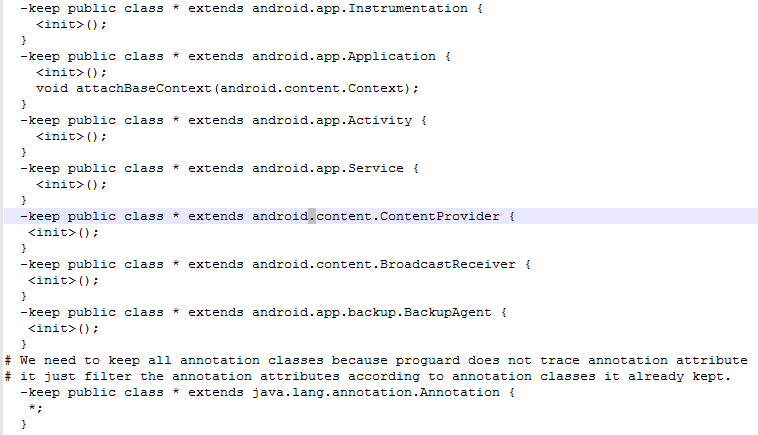
}

}

采用上面的配置之后，如果应用的方法数没有越界，那么Gradle并不会生成多个dex文件；如果方法数越界后，Gradle就会在apk中打包2个或者多个dex文件，具体会打包多少个dex文件要看当前项目的代码规模。

#### 主dex生成原理

**在**sdk\build-tools\platform-version\mainDexClasses.rules有定义，放入主Dex相关类有Instrumentation，Application，Activity，Service，ContentProvider，BroadcastReceiver，BackupAgent的所有子类。



有了规则文件之后，gradle会自动执行类扫描，

在有些情况下，可能需要指定**主dex**文件中所要包含的类，这个可以通过**--main-dex-list**选项来实现这个功能。

|  |
| --- |
| afterEvaluate {  println "afterEvaluate"  tasks.matching {  it.name.startsWith('dex')  }.each { dx ->  def listFile = project.rootDir.absolutePath + '/app/**maindexlist**.txt'  println "root dir:" + project.rootDir.absolutePath  println "dex task found: " + dx.name  if (dx.additionalParameters == null) {  dx.additionalParameters = []  }  dx.additionalParameters += '--multi-dex'  dx.additionalParameters += '--**main-dex-list**=' + listFile  dx.additionalParameters += '--minimal-main-dex'  }  } |

--multi-dex表明当方法数越界时生成多个dex文件，--main-dex-list指定了要在主dex中打包的类的列表，--minimal-main-dex表明只有--main-dex-list所指定的类才能打包到主dex中,最小化主dex。multidex的jar包中的9个类必须要打包到主dex中，其次不能在Application中成员以及代码块中访问其他dex中的类，否个程序会因为无法加载对应的类而中止执行。

**Maindexlist可以用脚本来实现，内容格式：**

Android/support/multidex/MultiDexApplication.class

Android/support/multidex/MultiDex$V4.class

#### 不足

Multidex方案可能带来的问题：

1.应用启动速度会降低，常见问题，甚至ANR，尤其是其他dex大的时候因为应用启动的时候会加载额外的dex文件，所以要避免生成较大的dex文件；

2. Dalvik LinearAlloc的bug，少见问题。需要做大量的兼容性测试，可能导致使用multidex的应用无法在Android 4.0以前的手机上运行。甚至大量消耗内存而奔溃

## Android的动态加载技术

(1)动态加载技术又称插件化技术，将应用插件化可以减轻应用的内存和CPU占用，还可以在**不发布新版本的情况下**更新某些模块。不同的插件化方案各有特色，但是都需要解决三个基础性问题：资源访问，Activity生命周期管理和插件ClassLoader的管理。

(2)宿主和插件：宿主是指普通的apk，插件是经过处理的dex或者apk。在主流的插件化框架中多采用特殊处理的apk作为插件，处理方式往往和编译以及打包环节有关，另外很多插件化框架都需要用到代理Activity的概念，插件Activity的启动大多数是借助一个代理Activity来实现的。

(3)资源访问：宿主程序调起未安装的插件apk，插件中凡是R开头的资源都不能访问了，因为宿主程序中并没有插件的资源，通过R来访问插件的资源是行不通的。

Activity的资源访问是通过ContextImpl来完成的，它有两个方法getAssets()和getResources()方法是用来加载资源的。

**具体实现方式是通过反射，调用AssetManager的addAssetPath方法添加插件的路径，然后将插件apk中的资源加载到Resources对象中即可。**

(4)Activity生命周期管理：有两种常见的方式，反射方式和接口方式。反射方式就是通过反射去获取Activity的各个生命周期方法，然后在代理Activity中去调用插件Activity对应的生命周期方法即可。

反射方式代码繁琐，性能开销大。接口方式将Activity的生命周期方法提取出来作为一个接口，然后通过代理Activity去调用插件Activity的生命周期方法，这样就完成了插件Activity的生命周期管理。

(5)插件ClassLoader的管理：为了更好地对多插件进行支持，需要合理地去管理各个插件的DexClassLoader，这样同一个插件就可以采用同一个ClassLoader去加载类，从而避免了多个ClassLoader加载同一个类时所引起的类型转换错误。

其他详细信息看作者插件化框架singwhatiwanna/dynamic-load-apk

# DL动态加载框架技术文档

# 参考

[综合技术- 7.4multidex解决方法数越界](https://www.kancloud.cn/kancloud/art-of-android-development-reading-notes/90458)

# 结束语

性能优化和功能开发不同，性能优化会涉及更广的知识点，在核心技能上也需要有一定的技术深度，只有找到问题的根本原因和技术背景，才能真正达到优化目的。通过本书的学习，我们知道性能优化的核心目的是提升用户的使用体验，这些体验是用户能明显感知到的，如下图所示。



本书对以上用户能感知到的性能做了系统化的分析并提供了相应的解决方案。所有的性能优化过程都差不多，即发现问题，再去寻找问题解决方案，最后解决问题。这个流程中最重要的两个过程如下：

1）深挖问题的根本原因。

2）合理使用各种性能相关工具。

需要知道的是，性能优化是持续性的工作，始终贯穿了应用的所有开发周期，每个版本发布前，都需要考虑性能是否达到最优，因此建议打造一个可持续的监控体系，也就是要做好数据上报，数据上报有两个目的：

·发现问题。

·验证性能优化后是否达到目的。

但数据也不能太多，本身收集日志也是对性能有一定影响的操作，所以可以提炼出一些核心数据，以下数据是非常有必要的。

·Crash。

·卡顿。

·核心功能成功/失败率。

性能优化的另一个重点在于平衡，比如内存和速度之间的平衡。众所周知，从内存中读取数据是最快的，但同时要付出更多的内存空间，因此需要一个平衡点。性能优化过程中平衡点的选择，需要根据不同的场景和环境决定，大家要知道，性能优化的最终目的是提高用户体验，而非某一个点的优化，需要全面考虑。

# QA

* 阅读系统显示源码SurfaceFlinger
* VSYNC的成功，告诉我们不同技术的融合能取得很好的效果
* 双缓冲技术为甚么还有卡顿情况
* CPU以主动查询的方式为啥低效
* VSYNC的中断机制为何效率就高了？
* TRAVERSAL为何叫与UI控件绘制有关
* 60的帧率是怎么得到的，肉眼生物特性？显存的工作频率？
* Merge工作原理，还是没有看明白
* this.getWindow().setBackgroundDrawable(null);为啥并没有work？
* 而如果主线程卡住了，说明dispatchMessage内部出了问题?
* 在基类的this是指的那个？
* getMemoryClass，一般应用程序的堆多大？
* ArrayMap为何比HashMap更节省空间？
* 如何跨进程读写SharedPreferences？解决app通信问题
* 插件话会增加类的复杂度么？模块通信是否更慢？插件化框架都有哪些？
* classes2 为啥方法数少也会出现？没有指定mudex也会有？

# 启发

* 硬件加速（GPU）至少需要8MB内存，导致系统奔溃？
* 16ms，如果某个操作花费的时间是24ms，丢帧：可能是你的Layout太过复杂，无法在16ms内完成渲染；可能是UI上有层叠太多的绘制单元；还有可能是动画执行的次数过多
* 绘制任务太重;主线程太忙了
* 优化FPV: 使用TraceView查看耗时，主要关注Calls+Recur Calls/Total和Cpu Time/Call这两个值，也就是关注调用次数多和耗时久的方法，然后优化这些方法的逻辑和调用次数，减少耗时。
* 搭建性能测试框架，指定指标？为测试提供
* 云测试是怎么做的呢？
* 并且RelativeLayout允许子元素指定它们相对于其他元素或父元素的位置，那么对于动画的UI尽量用relative？
* ViewStub优化状态图
* na的log可以抽样上报啊
* 耗时统计工具可以开始研发了
* FPV冷启动方案：必要且耗时：启动初始化，**考虑用线程来初始化。**·必要不耗时：首页绘制。·非必要耗时：数据上报、**插件初始化。**·非必要不耗时：不用想，这块直接去掉，在需要用的时再加载。
* **Invalidate在信道和农机那里用起来？**
* notifyDataSetChanged来check一下开机引导和wifi热点
* 开启硬件加速消耗内存，不可释放，用空间换时间？
* JAVA编程思想也要整理成为note文档，之前的一些笔记也要类似整理
* 如何使用Weak Reference来降低内存泄漏的风险，@bo，@na
* 既然在ART虚拟机更好，那么为甚我们不直接用这类虚拟机呢？
* 有效利用系统自带的资源和函数，color分辨率之类的
* 对象池 实现内存复用
* FlatBuffers替代gson？
* BroadcastTimeout，不要做复杂的事情！
* 压缩数据格式，比如采用GZIP压缩
* **系统变频调速设计？**
* JobService优化系统升级？
* 了解系统的省电模式？我们的设备也应这么处理吧
* main-dex-list eventbusIndex失败？或者是其他类加载不了？升级问题可以解决了？Dalvik LinearAlloc的bug？
* 广播和服务处理不当，都超时了吧？

# task

* ViewStub实践，看热点开启的代码，qs的page设置能否优化
* 实践用命令/代码方式取得trace，并分析
* android.os.Debug这个工具还有什么东西可以挖掘的？
* 2.3.1.1. Hierarchy Viewer用debug系统测试一下这个功能
* 验证一个控件的属性越少，解析越快？
* drawCardsWithotOver-Draw还是没有看懂？？？
* TimeMonitorManager用起来
* 帧动画优化的实践写出来，upgrade
* 学习openGL，捡起来！
* Looper研究学习，真正掌握ui主线程的概念
* 根据本文，实现性能监控框架，卡顿监控方案与实现
* 写NDK代码，实现crash监控？
* Cpu调频实现一下？WakeLock