|  |
| --- |
| 1、内存问题可以细分为如下三类:内存抖动、内存泄漏、内存溢出。  1）内存抖动  内存波动图形呈锯齿张、GC导致卡顿。这个问题在Dalvik虚拟机上会更加明显，而ART虚拟机在内存管理跟回收策略上都做了大量优化，内存分配和GC效率相比提升了5~10倍，所以 出现内存抖动的概率会小很多。  2）内存泄漏  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。内存泄漏就是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。  3）内存溢出  即OOM，OOM时会导致程序异常。Android设备出厂以后，java虚拟机对单个应用的最大内存分配就确定下来了，超出这个值就会OOM。  除了因内存泄漏累积到一定程度导致OOM的情况以外，也有一次性申请很多内存，比如说一次创建大的数组或者是载入大的文件如图片的时候会导致OOM。而且，实际情况下 很多OOM就是因图片处理不当而产生的。  2、内存检测工具：Memory Profiler；  LeakCanary自动化 内存泄漏检测神器。  3、Java 内存分配  Java的内存分配区域分为如下五部分：  1）、方法区：主要存放静态常量。  2）、虚拟机栈：Java变量引用。  3）、本地方法栈：native变量引用。  4）、堆：对象。  5）、程序计数器：计算当前线程的当前方法执行到多少行。  4、Java内存回收算法  （1）标记-清除算法  流程可简述为两步：  1）、标记所有需要回收的对象。  2）、统一回收所有被标记的对象。  优点：  实现比较简单。  缺点：  1）、标记、清除效率不高。  2）、产生大量内存碎片。  （2）复制算法  流程可简述为 三步：  1）、将内存划分为大小相等的两块。  2）、一块内存用完之后复制存活对象到另一块。  3）、清理另一块内存。  ##这里的话，是不是重在查询、复制，清除操作是很高效的。  优点  实现简单，运行高效，每次仅需遍历标记一半的内存区域。  缺点  会浪费一半的空间，代价大。  （3）标记-整理算法  流程可简述为 三步：  1）、标记过程与 标记-清除算法 一样。  2）、存活对象往一端进行移动。  3）、清理其余内存。  ##这个相对于复制算法有什么区别呢？是不是并发GC呢？在移动的过程，不能进行分配内存。  优点  1）、避免 标记-清除 导致的内存碎片。  2）、避免复制算法的空间浪费。  （4）分代收集算法  现在主流的虚拟机 一般用的比较多的还是分代收集算法，它具有如下 特点：  1）、结合多种算法优势。  2）、新生代对象存活率低，使用 复制算法。  3）、老年代对象存活率高，使用 标记-整理算法。  5、内存抖动  当内存频繁分配和回收导致内存不稳定，就会出现内存抖动，它通常表现为频繁GC、内存曲线呈锯齿状。并且，它的危害也很严重，通常会导致页面卡顿，甚至造成OOM。  为什么内存抖动会导致 OOM？主要原因有如下两点：  1）、频繁创建对象，导致内存不足及碎片（不连续）。  2）、不连续的内存片无法被分配，导致OOM。  排查方法：通常的技巧就是着重查看循环或频繁被调用的地方。  面列举一些导致内存抖动的常见案例，如下所示：  （1）字符串使用加号拼接  1）、使用StringBuilder替代。  2）、初始化时设置容量，减少StringBuilder的扩容。  （2）资源复用  1）、使用 全局缓存池，以 重用频繁申请和释放的对象。  2）、注意 结束 使用后，需要 手动释放对象池中的对象。  （3）减少不合理的对象创建  1）、ondraw、getView 中创建的对象尽量进行复用。  2）、避免在循环中不断创建局部变量。  （4）使用合理的数据结构  使用 SparseArray类族、ArrayMap 来替代 HashMap。  6、内存问题总结  1、内类是有危险的编码方式  每个类实例都具有一个 this$0，当它的内类需要访问它的成员时，内类就会持有外类的 this$0，通过 this$0 就可以访问外部类所有的成员。  解决方案是在 Activity 关闭，即触发 onDestory 时解除内类和外部的引用关系。  2、普通 Hanlder 内部类的问题  这也是一个 this$0 间接引用的问题，对于 Handler 的解决方案一般可以归结为如下三个步骤：  static + WeakReference;  removeCallbackAndMessages 来移除回调和消息。  3、登录界面的内存问题  如果在闪屏页跳转到登录界面时没有调用 finish()，则会造成闪屏页的内存泄漏，在碰到这种”过渡界面“的情况时，需要注意不要产生这样的内存 Bug。  4、使用系统服务时产生的内存问题  我们通常都会使用 getSystemService 方法来获取系统服务，但是当在 Activity 中调用时，会默认把 Activity 的 Context 传给系统服务，在某些不确定的情况下，某些系统服务内部会产生异常，从而 hold 住外界传入的 Context。  解决方案是 直接使用 Applicaiton 的 Context 去获取系统服务。  5、把 WebView 类型的泄漏装进垃圾桶进程  我们都知道，对应 WebView 来说，其 网络延时、引擎 Session 管理、Cookies 管理、引擎内核线程、HTML5调用系统声音、视频播放组件等产生的引用链条无法及时打断，造成的内存问题基本上可以用”无解“来形容。  解决方案是我们可以 把 WebView 装入另一个进程。 具体为在 AndroidManifes 中对当前的 Activity 设置 android:process 属性即可，最后，在 Activity 的 onDestory 中退出进程，这样即可基本上终结 WebView 造成的泄漏。  6、在适当的时候对组件进行注销  我们在平常开发过程中经常需要在Activity创建的时候去注册一些组件，如广播、定时器、事件总线等等。这个时候我们应该在适当的时候对组件进行注销，如 onPause 或 onDestory 方法中。  7、Handler / FrameLayout 的 postDelyed 方法触发的内存问题  不仅在使用 Handler 的 sendMessage 方法时，我们需要在 onDestory 中使用 removeCallbackAndMessage 移除回调和消息，在使用到 Handler / FrameLayout 的 postDelyed 方法时，我们需要调用 removeCallbacks 去移除实现控件内部的延时器对 Runnable 内类的持有。  8、图片放错资源目录也会有内存问题  在做资源适配的时候，因为需要考虑到 APK 的瘦身问题，无法为每张图片在每个 drawable / mipmap 目录下安置一张适配图片的副本。如果放到分辨率低的目录如 hdpi 目录，则可能会造成内存问题，这个时候建议尽量问设计人员要高品质图片然后往高密度目录下方，如 xxhdpi 目录，这样 在低密屏上”放大倍数“是小于1的，在保证画质的前提下，内存也是可控的。也可以使用 Drawable.createFromSream 替换 getResources().getDrawable 来加载，这样便可以绕过 Android 的默认适配规则。  9、列表 item 被回收时注意释放图片的引用  10、使用 ViewStub 进行占位  我们应该使用 ViewStub 对那些没有马上用到的资源去做延迟加载，并且还有很多大概率不会出现的 View更要去做懒加载，这样可以等到要使用时再去为它们分配相应的内存。  11、注意定时清理 App 过时的埋点数据  产品或者运营为了统计数据会在每个版本中不断地增加新的埋点。所以我们需要定期地去清理一些过时的埋点，以此来适当地优化内存以及CPU的压力。  12、针对匿名内部类 Runnable 造成内存泄漏的处理  7、搭建一套成体系化的图片优化/监控机制  1）统一图片库  在项目中，我们需要收拢图片的调用，避免使用 Bitmap.createBitmap、BitmapFactory 相关的接口创建 Bitmap，而应该使用自己的图片框架。  2）设备分级优化策略  内存优化首先需要根据设备环境来综合考虑，让高端设备使用更多的内存，做到 针对设备性能的好坏使用不同的内存分配和回收策略。  可以使用类似 device-year-class 的策略对设备进行分级。  3）建立统一的缓存管理组件  建立统一的缓存管理组件（参考 ACache），并合理使用 OnTrimMemory / LowMemory 回调，根据系统不同的状态去释放相应的缓存与内存。  4）低端机避免使用多进程  5）线下大图片检测  在开发过程中，如果检测到不合规的图片使用（如图片宽度超过View的宽度甚至屏幕宽度），应该立刻提示图片所在的Activity和堆栈，让开发人员更快发现并解决问题。  8、图片加载时的降级处理  使用 Glide、Fresco 等图片加载库，通过定制，在加载 bitmap 时，若发生 OOM，则使用 try catch 将其捕获，然后清除图片 cache，尝试降低 bitmap format（ARGB8888、RGB565、ARGB4444、ALPHA8）。  需要注意的是，OOM 是可以捕获的，只要 OOM 是由 try 语句中的对象声明所导致的，那么在 catch 语句中，是可以释放掉这些对象，解决 OOM 的问题的。 |

# 一、重识内存优化

Android给每个应用进程分配的内存都是非常有限的，那么，为什么不能把图片下载下来都放到磁盘中呢？那是因为放在 内存 中，展示会更 “快”，快的原因有两点，如下所示：

1）、硬件快：内存本身读取、存入速度快。

2）、复用快：解码成果有效保存，复用时，直接使用解码后对象，而不是再做一次图像解码。

这里说一下解码的概念。Android系统要在屏幕上展示图片的时候只认 “像素缓冲”，而这也是大多数操作系统的特征。而我们 常见的jpg，png等图片格式，都是把 “像素缓冲” 使用不同的手段压缩后的结果，所以这些格式的图片，要在设备上 展示，就 必须经过一次解码，它的 执行速度会受图片压缩比、尺寸等因素影响。

(官方建议：把从内存中淘汰的图片，降低压缩比后存储到本地，以备后用，这样可以最大限度地降低以后复用时的解码开销。)

下面，我们来了解一下内存优化的一些重要概念。

## 1、手机RAM

手机不使用 PC 的 DDR内存，采用的是 LPDDR RAM，即 ”低功耗双倍数据速率内存“。其计算规则如下所示：

LPDDR系列的带宽 = 时钟频率 \* 内存总线位数 / 8

LPDDR4 = 1600MHZ \* 64 / 8 \* 双倍速率 = 25.6GB/s。

那么内存占用是否越少越好？

当系统 内存充足 的时候，我们可以 多用 一些获得 更好的性能。当系统 内存不足 的时候，我们希望可以做到 ”用时分配，及时释放“。

## 2、内存优化的纬度

对于Android内存优化来说又可以细分为如下两个维度，如下所示：

1）、RAM优化

2）、ROM优化

随机存取存储器（英语：Random Access Memory，缩写：RAM），也叫主存，是与CPU直接交换数据的内部存储器。它可以随时读写（刷新时除外），而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储介质。只读存储器（Read-Only Memory，ROM）以非破坏性读出方式工作，只能读出无法写入信息。信息一旦写入后就固定下来，即使切断电源，信息也不会丢失，所以又称为固定存储器。RAM与ROM常见的就是内存与硬盘。

### 1、RAM优化

主要是 降低运行时内存。它的 目的 有如下三个：

1）、防止应用发生OOM。

2）、降低应用由于内存过大被LMK机制杀死的概率。

3）、避免不合理使用内存导致GC次数增多，从而导致应用发生卡顿。

### 2、ROM优化

降低应用占ROM的体积，进行APK瘦身。它的 目的 主要是为了 降低应用占用空间，避免因ROM空间不足导致程序无法安装。

### 3、内存问题

那么，内存问题主要是有哪几类呢？内存问题通常来说，可以细分为如下 三类:

1）、内存抖动

2）、内存泄漏

3）、内存溢出

下面，我们来了解下它们。

1、内存抖动

内存波动图形呈 锯齿张、GC导致卡顿。

这个问题在 Dalvik虚拟机 上会 更加明显，而 ART虚拟机 在 内存管理跟回收策略 上都做了 大量优化，内存分配和GC效率相比提升了5~10倍，所以 出现内存抖动的概率会小很多。

2、内存泄漏

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。内存泄漏就是 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。简言之，就是 对象被持有导致无法释放或不能按照对象正常的生命周期进行释放。一般来说，可用内存减少、频繁GC，容易导致内存泄漏。

3、内存溢出

即OOM，OOM时会导致程序异常。Android设备出厂以后，java虚拟机对单个应用的最大内存分配就确定下来了，超出这个值就会OOM。单个应用可用的最大内存对应于 /system/build.prop 文件中的 dalvik.vm.heapgrowthlimit。

此外，除了因内存泄漏累积到一定程度导致OOM的情况以外，也有一次性申请很多内存，比如说 一次创建大的数组或者是载入大的文件如图片的时候会导致OOM。而且，实际情况下 很多OOM就是因图片处理不当 而产生的。

# 二、常见工具选择

在 Android性能优化之内存优化 中我们已经介绍过了相关的优化工具，这里再简单回顾一下。

## 1、Memory Profiler

作用

1）、实时图表展示应用内存使用量。

2）、用于识别内存泄漏、抖动等。

3）、提供捕获堆转储、强制GC以及根据内存分配的能力。

优点

1）、方便直观

2）、线下使用

## 2、Memory Analyzer

强大的 Java Heap 分析工具，查找 内存泄漏及内存占用， 生成 整体报告、分析内存问题 等等。建议 线下深入使用。

## 3、LeakCanary

自动化 内存泄漏检测神器。建议仅用于线下集成。

它的 缺点 比较明显，具体有如下两点：

1）、虽然使用了 idleHandler与多进程，但是 dumphprof 的 SuspendAll Thread 的特性依然会导致应用卡顿。

2）、在三星等手机，系统会缓存最后一个Activity，此时应该采用更严格的检测模式。

# 三、Android内存管理机制回顾

ART 和 Dalvik 虚拟机使用 分页和内存映射 来管理内存。下面我们先从Java的内存分配开始说起。

## 1、Java 内存分配

Java的 内存分配区域 分为如下 五部分：

1）、方法区：主要存放静态常量。

2）、虚拟机栈：Java变量引用。

3）、本地方法栈：native变量引用。

4）、堆：对象。

5）、程序计数器：计算当前线程的当前方法执行到多少行。

## 2、Java 内存回收算法

### 1、标记-清除算法

流程可简述为 两步：

1）、标记所有需要回收的对象。

2）、统一回收所有被标记的对象。

优点

实现比较简单。

缺点

1）、标记、清除效率不高。

2）、产生大量内存碎片。

### 2、复制算法

流程可简述为 三步：

1）、将内存划分为大小相等的两块。

2）、一块内存用完之后复制存活对象到另一块。

3）、清理另一块内存。

##这里的话，是不是重在查询、复制，清除操作是很高效的。

优点

实现简单，运行高效，每次仅需遍历标记一半的内存区域。

缺点

会浪费一半的空间，代价大。

### 3、标记-整理算法

流程可简述为 三步：

1）、标记过程与 标记-清除算法 一样。

2）、存活对象往一端进行移动。

3）、清理其余内存。

##这个相对于复制算法有什么区别呢？是不是并发GC呢？在移动的过程，不能进行分配内存。

优点

1）、避免 标记-清除 导致的内存碎片。

2）、避免复制算法的空间浪费。

### 4、分代收集算法

现在 主流的虚拟机 一般用的比较多的还是分代收集算法，它具有如下 特点：

1）、结合多种算法优势。

2）、新生代对象存活率低，使用 复制算法。

3）、老年代对象存活率高，使用 标记-整理算法。

## 3、Android 内存管理机制

Android 中的内存是 弹性分配 的，分配值 与 最大值 受具体设备影响。

对于 OOM场景 其实可以细分为如下两种：

1）、内存真正不足。

2）、可用（被分配的）内存不足。

我们需要着重注意一下这两种的区分。

## 4、小结

以Android中虚拟机的角度来说，我们要清楚 Dalvik 与 ART 区别，Dalvik 仅固定一种回收算法，而 ART 回收算法可在 运行期按需选择，并且，ART 具备 内存整理 能力，减少内存空洞。

最后，LMK（Low Memory killer） 机制保证了进程资源的合理利用，它的实现原理主要是 根据进程分类和回收收益来综合决定的一套算法集。

# 四、内存抖动

当 内存频繁分配和回收 导致内存 不稳定，就会出现内存抖动，它通常表现为 频繁GC、内存曲线呈锯齿状。

并且，它的危害也很严重，通常会导致 页面卡顿，甚至造成 OOM。

## 1、那么，为什么内存抖动会导致 OOM？

主要原因有如下两点：

1）、频繁创建对象，导致内存不足及碎片（不连续）。

2）、不连续的内存片无法被分配，导致OOM。

## 2、内存抖动解决实战

这里我们假设有这样一个场景：点击按钮使用 handler 发送一个空消息，handler 的 handleMessage 接收到消息后创建内存抖动，即在 for 循环创建 100个容量为10万 的 strings 数组并在 30ms 后继续发送空消息。

一般使用 Memory Profiler （表现为 频繁GC、内存曲线呈锯齿状）结合代码排查即可找到内存抖动出现的地方。

通常的技巧就是着重查看 循环或频繁被调用 的地方。

## 3、内存抖动常见案例

下面列举一些导致内存抖动的常见案例，如下所示：

### 1、字符串使用加号拼接

1）、使用StringBuilder替代。

2）、初始化时设置容量，减少StringBuilder的扩容。

### 2、资源复用

1）、使用 全局缓存池，以 重用频繁申请和释放的对象。

2）、注意 结束 使用后，需要 手动释放对象池中的对象。

### 3、减少不合理的对象创建

1）、ondraw、getView 中创建的对象尽量进行复用。

2）、避免在循环中不断创建局部变量。

### 4、使用合理的数据结构

使用 SparseArray类族、ArrayMap 来替代 HashMap。

# 五、内存优化体系化搭建

在开始我们今天正式的主题之前，我们先来回归一下内存泄漏的概念与解决技巧。

所谓的内存泄漏就是 内存中存在已经没有用的对象。它的 表现 一般为 内存抖动、可用内存逐渐减少。 它的 危害 即会导致 内存不足、GC频繁、OOM。

而对于 内存泄漏的分析 一般可简述为如下 两步：

1）、使用 Memory Profiler 初步观察。

2）、通过 Memory Analyzer 结合代码确认。

## 1、MAT回顾

MAT查找内存泄漏

对于MAT来说，其常规的查找内存泄漏的方式可以细分为如下三步：

1）、首先，找到当前 Activity，在 Histogram 中选择其 List Objects 中的 with incoming reference（哪些引用引向了我）。

2）、然后，选择当前的一个 Path to GC Roots/Merge to GC Roots 的 exclude All 弱软虚引用。

3）、最后，找到的泄漏对象在左下角下会有一个小圆圈。

此外，在 Android性能优化之内存优化 还有几种进阶的使用方式，这里就不一一赘述了，下面，我们来看看关于 MAT 使用时的一些关键细节。

MAT的关键使用细节

要全面掌握MAT的用法，必须要先了解 隐藏在 MAT 使用中的四大细节，如下所示：

1）、善于使用 Regex 查找对应泄漏类。

2）、使用 group by package 查找对应包下的具体类。

3）、明白 with outgoing references 和 with incoming references 的区别。

with outgoing references：它引用了哪些对象。

with incoming references：哪些对象引用了它。

4）、了解 Shallow Heap 和 Retained Heap 的区别。

Shallow Heap：表示对象自身占用的内存。

Retained Heap：对象自身占用的内存 + 对象引用的对象所占用的内存。

MAT 关键组件回顾

除此之外，MAT 共有 5个关键组件 帮助我们去分析内存方面的问题，分别如下所示：

1）、Dominator\_tree

2）、Histogram

3）、thread\_overview

4）、Top Consumers

5）、Leak Suspects

## 2、搭建体系化的图片优化 / 监控机制

在介绍图片监控体系的搭建之前，首先我们来回顾下 Android Bitmap 内存分配的变化。

Android Bitmap 内存分配的变化

在Android 3.0之前

1）、Bitmap 对象存放在 Java Heap，而像素数据是存放在 Native 内存中的。

2）、如果不手动调用 recycle，Bitmap Native 内存的回收完全依赖 finalize 函数回调，但是回调时机是不可控的。

Android 3.0 ~ Android 7.0

将 Bitmap对象 和 像素数据 统一放到 Java Heap 中，即使不调用 recycle，Bitmap 像素数据也会随着对象一起被回收。

但是，Bitmap 全部放在 Java Heap 中的缺点很明显，大致有如下两点：

1）、Bitmap是内存消耗的大户，而 Max Java Heap 一般限制为 256、512MB，Bitmap 过大过多容易导致 OOM。

2）、容易引起大量 GC，没有充分利用系统的可用内存。

Android 8.0及以后

1）、使用了能够辅助回收 Native 内存的 NativeAllocationRegistry，以实现将像素数据放到 Native 内存中，并且可以和 Bitmap 对象一起快速释放，最后，在 GC 的时候还可以考虑到这些 Bitmap 内存以防止被滥用。

2）、Android 8.0 为了 解决图片内存占用过多和图像绘制效率过慢 的问题新增了 硬件位图 Hardware Bitmap。

那么，我们如何将图片内存存放在 Native 中呢？

将图片内存存放在Native中的步骤有 四步，如下所示：

1）、调用 libandroid\_runtime.so 中的 Bitmap 构造函数，申请一张空的 Native Bitmap。对于不同 Android 版本而言，这里的获取过程都有一些差异需要适配。

2）、申请一张普通的 Java Bitmap。

3）、将 Java Bitmap 的内容绘制到 Native Bitmap 中。

4）、释放 Java Bitmap 内存。

我们都知道的是，当 系统内存不足 的时候，LMK 会根据 OOM\_adj 开始杀进程，从 后台、桌面、服务、前台，直到手机重启。并且，如果频繁申请释放 Java Bitmap 也很容易导致内存抖动。对于这种种问题，我们该 如何评估内存对应用性能的影响 呢？

对此，我们可以主要从以下 两个方面 进行评估，如下所示：

1）、崩溃中异常退出和 OOM 的比例。

2）、低内存设备更容易出现内存不足和卡顿，需要查看应用中用户的手机内存在 2GB 以下所占的比例。

对于具体的优化策略与手段，我们可以从以下 七个方面 来搭建一套 成体系化的图片优化 / 监控机制。

### 1、统一图片库

在项目中，我们需要 收拢图片的调用，避免使用 Bitmap.createBitmap、BitmapFactory 相关的接口创建 Bitmap，而应该使用自己的图片框架。

### 2、设备分级优化策略

内存优化首先需要根据 设备环境 来综合考虑，让高端设备使用更多的内存，做到 针对设备性能的好坏使用不同的内存分配和回收策略。

因此，我们可以使用类似 device-year-class 的策略对设备进行分级，对于低端机用户可以关闭复杂的动画或”重功能“，使用565格式的图片或更小的缓存内存 等等。

业务开发人员需要 考虑功能是否对低端机开启，在系统资源不够时主动去做降级处理。

### 3、建立统一的缓存管理组件

建立统一的缓存管理组件（参考 ACache），并合理使用 OnTrimMemory / LowMemory 回调，根据系统不同的状态去释放相应的缓存与内存。

在实现过程中，需要 解决使用 static LRUCache 来缓存大尺寸 Bitmap 的问题。

并且，在通过实际的测试后，发现 onTrimMemory 的 ComponetnCallbacks2.TRIM\_MEMORY\_COMPLETE 并不等价于 onLowMemory，因此建议仍然要去监听 onLowMemory 回调。

### 4、低端机避免使用多进程

一个 空进程 也会占用 10MB 内存，低端机应该尽可能减少使用多进程。

针对低端机用户可以推出 4MB 的轻量级版本，如今日头条极速版、Facebook Lite。

### 5、线下大图片检测

在开发过程中，如果检测到不合规的图片使用（如图片宽度超过View的宽度甚至屏幕宽度），应该立刻提示图片所在的Activity和堆栈，让开发人员更快发现并解决问题。在灰度和线上环境，可以将异常信息上报到后台，还可以计算超宽率（图片超过屏幕大小所占图片总数的比例）。

下面，我们介绍下如何实现对大图片的检测。

常规实现

继承 ImageView，重写实现计算图片大小。但是侵入性强，并且不通用。

因此，这里我们介绍一种更好的方案：ARTHook。

ARTHook优雅检测大图

ARTHook，即 挂钩，用额外的代码勾住原有的方法，以修改执行逻辑，主要可以用于以下四个方面：

1）、AOP编程

2）、运行时插桩

3）、性能分析

4）、安全审计

具体我们是使用 Epic 来进行 Hook，Epic 是 一个虚拟机层面，以 Java 方法为粒度的运行时 Hook 框架。简单来说，它就是 ART 上的 Dexposed，并且它目前 支持 Android 4.0~10.0。

Epic github 地址

使用步骤

Epic通常的使用步骤为如下三个步骤：

1、在项目 moudle 的 build.gradle 中添加

compile 'me.weishu:epic:0.6.0'

复制代码

2、继承 XC\_MethodHook，实现 Hook 方法前后的逻辑。如 监控Java线程的创建和销毁：

class ThreadMethodHook extends XC\_MethodHook{

@Override

protected void beforeHookedMethod(MethodHookParam param) throws Throwable {

super.beforeHookedMethod(param);

Thread t = (Thread) param.thisObject;

Log.i(TAG, "thread:" + t + ", started..");

}

@Override

protected void afterHookedMethod(MethodHookParam param) throws Throwable {

super.afterHookedMethod(param);

Thread t = (Thread) param.thisObject;

Log.i(TAG, "thread:" + t + ", exit..");

}

}

复制代码

3、注入 Hook 好的方法：

DexposedBridge.findAndHookMethod(Thread.class, "run", new ThreadMethodHook());

复制代码

知道了 Epic 的基本使用方法之后，我们便可以利用它来实现大图片的监控报警了。

项目实战

以 Awesome-WanAndroid 项目为例，首先，在 WanAndroidApp 的 onCreate 方法中添加如下代码：

DexposedBridge.hookAllConstructors(ImageView.class, new XC\_MethodHook() {

@Override

protected void afterHookedMethod(MethodHookParam param) throws Throwable {

super.afterHookedMethod(param);

// 1

DexposedBridge.findAndHookMethod(ImageView.class, "setImageBitmap", Bitmap.class, new ImageHook());

}

});

复制代码

在注释1处，我们 通过调用 DexposedBridge 的 findAndHookMethod 方法找到所有通过 ImageView 的 setImageBitmap 方法设置的切入点，其中最后一个参数 ImageHook 对象是继承了 XC\_MethodHook 类，其目的是为了 重写 afterHookedMethod 方法拿到相应的参数进行监控逻辑的判断。

接下来，我们来实现我们的 ImageHook 类，代码如下所示：

public class ImageHook extends XC\_MethodHook {

@Override

protected void afterHookedMethod(MethodHookParam param) throws Throwable {

super.afterHookedMethod(param);

// 1

ImageView imageView = (ImageView) param.thisObject;

checkBitmap(imageView,((ImageView) param.thisObject).getDrawable());

}

private static void checkBitmap(Object thiz, Drawable drawable) {

if (drawable instanceof BitmapDrawable && thiz instanceof View) {

final Bitmap bitmap = ((BitmapDrawable) drawable).getBitmap();

if (bitmap != null) {

final View view = (View) thiz;

int width = view.getWidth();

int height = view.getHeight();

if (width > 0 && height > 0) {

// 2、图标宽高都大于view的2倍以上，则警告

if (bitmap.getWidth() >= (width << 1)

&& bitmap.getHeight() >= (height << 1)) {

warn(bitmap.getWidth(), bitmap.getHeight(), width, height, new RuntimeException("Bitmap size too large"));

}

} else {

// 3、当宽高度等于0时，说明ImageView还没有进行绘制，使用ViewTreeObserver进行大图检测的处理。

final Throwable stackTrace = new RuntimeException();

view.getViewTreeObserver().addOnPreDrawListener(new ViewTreeObserver.OnPreDrawListener() {

@Override

public boolean onPreDraw() {

int w = view.getWidth();

int h = view.getHeight();

if (w > 0 && h > 0) {

if (bitmap.getWidth() >= (w << 1)

&& bitmap.getHeight() >= (h << 1)) {

warn(bitmap.getWidth(), bitmap.getHeight(), w, h, stackTrace);

}

view.getViewTreeObserver().removeOnPreDrawListener(this);

}

return true;

}

});

}

}

}

}

private static void warn(int bitmapWidth, int bitmapHeight, int viewWidth, int viewHeight, Throwable t) {

String warnInfo = "Bitmap size too large: " +

"\n real size: (" + bitmapWidth + ',' + bitmapHeight + ')' +

"\n desired size: (" + viewWidth + ',' + viewHeight + ')' +

"\n call stack trace: \n" + Log.getStackTraceString(t) + '\n';

LogHelper.i(warnInfo);

}

}

复制代码

首先，在注释1处，我们重写了 ImageHook 的 afterHookedMethod 方法，拿到了当前的 ImageView 和要设置的 Bitmap 对象。然后，在注释2处，如果当前 ImageView 的宽高大于0，我们便进行大图检测的处理：ImageView 的宽高都大于 View 的2倍以上，则警告。接着，在注释3处，如果当前 ImageView 的宽高等于0，则说明 ImageView 还没有进行绘制，则使用 ImageView 的 ViewTreeObserver 获取其宽高进行大图检测的处理。至此，我们的大图检测检测组件就已经实现了。如果有小伙伴对 epic 的实现原理感兴趣的，可以查看这篇文章。

ARTHook方案实现小结

1）、无侵入性

2）、通用性强

3）、兼容性问题大，开源方案不能带到线上环境。

6、线下重复图片检测

首先我们来了解一下这里的 重复图片 所指的概念： 即 Bitmap 像素数据完全一致，但是有多个不同的对象存在。

重复图片检测的原理其实就是 使用内存 Hprof 分析工具，自动将重复 Bitmap 的图片和引用堆栈输出。

已完全配置好的项目请参见这里

使用说明

使用非常简单，只需要修改 Main 类的 main 方法的第一行代码，如下所示：

// 设置我们自己 App 中对应的 hprof 文件路径

String dumpFilePath = "//Users//quchao//Documents//heapdump//memory-40.hprof";

复制代码

然后，我们执行 main 方法即可在 //Users//quchao//Documents//heapdump 这个路径下看到生成的 images 文件夹，里面保存了项目中检测出来的重复的图片。images 目录如下所示：

注意：需要使用 8.0 以下的机器，因为 8.0 及以后 Bitmap 中的 buffer 已保存在 native 内存之中。

实现步骤

具体的实现可以细分为如下三个步骤：

1）、首先，获取 android.graphics.Bitmap 实例对象的 mBuffer 作为 ArrayInstance ，通过 getValues 获取的数据为 Object 类型。由于后面计算 md5 需要为 byte[] 类型，所以通过反射的方式调用 ArrayInstance#asRawByteArray 直接返回 byte[] 数据。

2）、然后，根据 mBuffer 的数据生成 png 图片文件，这里直接参考了 github.com/JetBrains/a… 的实现方式。

3）、最后，获取堆栈信息，直接 使用LeakCanary 获取 stack 的方法，使用 leakcanary-analyzer-1.6.2.jar 和 leakcanary-watcher-1.6.2.jar 这两个库文件。并用 反射 的方式调用了 HeapAnalyzer#findLeakTrace 方法。

其中，获取堆栈 的信息也可以直接使用 haha 库来进行获取。这里简单说一下 使用 haha 库获取堆栈的流程，其具体可以细分为八个步骤，如下所示：

1）、首先，预备一个已经存在重复 bitmap 的 hprof 文件。

2）、利用 haha 库上的 MemoryMappedFileBuffer 读取 hrpof 文件 [关键代码 new MemoryMappedFileBuffer(heapDumpFile) ]。

3）、解析生成 snapshot，获取 heap，这里我只获取了 app heap [关键代码 snapshot.getHeaps(); heap.getName().equals("app") ]。

4）、从 snapshot 中根据指定 class 查找出所有的 Bitmap Classes [关键代码snapshot.findClasses(Bitmap.class.getName()) ]。

5）、从 heap 中获得所有的 Bitmap 实例 instance [关键代码 clazz.getHeapInstances(heap.getId()) ]。

6）、根据 instance 中获取所有的属性信息 Field[]，并从 Field[] 查找出我们需要的 "mWidth" "mHeight" "mBuffer" 信息。

7）、通过 "mBuffer" 属性即可获取到他们的 hashcode 来判断是否是重复图片。

8）、最后，通过 instance 中 mNextInstanceToGcRoot 获取整个引用链信息并打印。

7、建立全局的线上 Bitmap 监控

为了建立全局的 Bitmap 监控，我们必须 对 Bitmap 的分配和回收 进行追踪。我们先来看看 Bitmap 有哪些特点：

1）、创建场景比较单一：在 Java 层调用 Bitmap.create 或 BitmapFactory 等方法创建，可以封装一层对 Bitmap 创建的接口，注意要 包含调用第三方库产生的 Bitmap，这里我们具体可以使用 ASM 编译插桩 + Gradle Transform 的方式来高效地实现。

2）、创建频率比较低。

3）、和 Java 对象的生命周期一样服从 GC，可以使用 WeakReference 来追踪 Bitmap 的销毁。

根据以上特点，我们可以建立一套 Bitmap 的高性价比监控组件：

1）、首先，在接口层将所有创建出来的 Bitmap 放入一个 WeakHashMap 中，并记录创建 Bitmap 的数据、堆栈等信息。

2）、然后，每隔一定时间查看 WeakHashMap 中有哪些 Bitmap 仍然存活来判断是否出现 Bitmap 滥用或泄漏。

3）、最后，如果发生了 Bitmap 滥用或泄露，则将相关的数据与堆栈等信息打印出来或上报至 APM 后台。

这个方案的 性能消耗很低，可以在 正式环境 中进行。但是，需要注意的一点是，正式与测试环境需要采用不同程度的监控。

## 建立线上应用内存监控体系

（其实，这个是自己实现的，可能Bugly应该也有相关的功能吧。）

要建立线上应用的内存监控体系，我们需要 先获取 App 的 DalvikHeap 与 NativeHeap，它们的获取方式可归结为如下四个步骤：

1、首先，通过 ActivityManager 的 getProcessMemoryInfo => Debug.MemoryInfo 获取内存信息数据。

2、然后，通过 hook Debug.MemoryInfo 的 getMemoryStat 方法（os v23 及以上）可以获得 Memory Profiler 中的多项数据，进而获得 细分内存的使用情况。

3、接着，通过 Runtime 获取 DalvikHeap。

4、最后，通过 Debug.getNativeHeapAllocatedSize 获取 NativeHeap。

对于监控场景，我们需要将其划分为两大类，如下所示：

1）、常规内存监控

2）、低内存监控

### 1、常规内存监控

根据 斐波那契数列 每隔一段时间（max：30min）获取内存的使用情况。常规内存的监控方法有多种实现方式，下面，我们按照 项目早期 => 壮大期 => 成熟期 的常规内存监控方式进行 演进式 讲解。

项目早期：针对场景进行线上 Dump 内存的方式

具体使用 Debug.dumpHprofData() 实现。

其实现的流程为如下四个步骤：

1）、超过最大内存的 80%。

2）、内存 Dump。

3）、回传文件至服务器。

4）、MAT 手动分析。

但是，这种方式有如下几个缺点：

1）、Dump文件太大，和对象数正相关，可以进行裁剪。

2）、上传失败率高，分析困难。

壮大期：LeakCanary带到线上的方式

在使用 LeakCanary 的时候我们需要 预设泄漏怀疑点，一旦发现泄漏进行回传。但这种实现方式缺点比较明显，如下所示：

1）、不适合所有情况，需要预设怀疑点。

2）、分析比较耗时，容易导致 OOM。

成熟期：定制 LeakCanary 方式

那么，如何定制线上的LeakCanary？

定制 LeakCanary 其实就是对 haha组件 来进行 定制。haha库是 square 出品的一款 自动分析Android堆栈的java库。这是haha库的 链接地址。

对于haha库，它的 基本用法 一般遵循为如下四个步骤：

1、导出堆栈文件

File heapDumpFile = ...

Debug.dumpHprofData(heapDumpFile.getAbsolutePath());

2、根据堆栈文件创建出内存映射文件缓冲区

DataBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);

3、根据文件缓存区创建出对应的快照

Snapshot snapshot = Snapshot.createSnapshot(buffer);

复制代码

4、从快照中获取指定的类

ClassObj someClass = snapshot.findClass("com.example.SomeClass");

我们在实现线上版的LeakCanary的时候主要要解决的问题有三个，如下所示：

1）、解决 预设怀疑点 时不准确的问题 => 自动找怀疑点。

2）、解决掉将 hprof 文件映射到内存中的时候可能导致内存暴涨甚至发生 OOM 的问题 => 对象裁剪，不全部加载到内存。即对生成的 Hprof 内存快照文件做一些优化：裁剪大部分图片对应的 byte 数据 以减少文件开销，最后，使用 7zip 压缩，一般可 节省 90% 大小。

3）、分析泄漏链路慢而导致分析时间过长 => 分析 Retain size 大的对象。

成熟期：实现内存泄漏监控闭环

在实现了线上版的 LeakCanary 之后，就需要 将线上版的 LeakCanary 与服务器和前端页面结合 起来。具体的 内存泄漏监控闭环流程 如下所示：

1）、当在线上版 LeakCanary 上发现内存泄漏时，手机将上传内存快照至服务器。

2）、此时服务器分析 Hprof，如果不是系统原因导致误报则通过 git 得到该最近修改人。

3）、最后将内存泄漏 bug 单提交给负责人。该负责人通过前端实现的 bug 单系统即可看到自己新增的bug。

此外,在实现 图片内存监控 的过程中，应注意 两个关键点，如下所示：

1）、在线上可以按照 不同的系统、屏幕分辨率 等纬度去 分析图片内存的占用情况。

2）、在 OOM 崩溃时，可以将 图片总内存、Top N 图片占用内存 写入 崩溃日志。

### 2、低内存监控

对于低内存的监控，通常有两种方式，分别如下所示：

1、利用 onTrimMemory / onLowMemory 监听系统回调的物理内存警告。

2、在后台起一个服务定时监控系统的内存占用，只要超过虚拟内存大小最大限制的 90% 则直接触发内存警告。

### 3、内存监控指标

为了准确衡量内存性能，我们需要引入一系列的内存监控指标，如下所示：

1）、发生频率

2）、发生时各项内存使用状况

3）、发生时App的当前场景

4）、内存异常率

内存 UV 异常率 = PSS 超过 400MB 的 UV / 采集UV

PSS 获取：调用 Debug.MemoryInfo 的 API 即可

如果出现 新的内存使用不当或内存泄漏 的场景，这个指标会有所 上涨。

5）、触顶率

内存 UV 触顶率 = Java 堆占用超过最大堆限制的 85% 的 UV / 采集UV

计算触顶率的代码如下所示：

long javaMax = Runtime.maxMemory();

long javaTotal = Runtime.totalMemory();

long javaUsed = javaTotal - runtime.freeMemory();

float proportion = (float) javaUsed / javaMax;

如果超过 85% 最大堆 的限制，GC 会变得更加 频发，容易造成 OOM 和 卡顿。

### 4、小结

在具体实现的时候，客户端 尽量只负责 上报数据，而 指标值的计算 可以由 后台 来计算。这样便可以通过 版本对比 来监控是否有 新增内存问题。因此，建立线上内存监控的完整方案 至少需要包含以下四点：

1）、待机内存、重点模块内存、OOM率。

2）、整体及重点模块 GC 次数、GC 时间。

3）、增强的 LeakCanry 自动化内存泄漏分析。

4）、低内存监控模块的设置。

## 4、建立全局的线程监控组件

每个线程初始化都需要 mmap 一定的栈大小，在默认情况下初始化一个线程需要 mmap 1MB 左右的内存空间。

在 32bit 的应用中有 4g 的 vmsize，实际能使用的有 3g+，这样一个进程 最大能创建的线程数 可以达到 3000个，但是，linux 对每个进程可创建的线程数也有一定的限制（/proc/pid/limits），并且，不同厂商也能修改这个限制，超过该限制就会 OOM。

因此，对线程数量的限制，在一定程度上可以 有效地避免 OOM 的发生。那么，实现一套 全局的线程监控组件 便是 刻不容缓 的了。

全局线程监控组件的实现原理

在线下或灰度的环境下通过一个定时器每隔 10分钟 dump 出应用所有的线程相关信息，当线程数超过当前阈值时，则将当前的线程信息上报并预警。

## 5、GC 监控组件搭建

通过 Debug.startAllocCounting 来监控 GC 情况，注意有一定 性能影响。

在 Android 6.0 之前 可以拿到 内存分配次数和大小以及 GC 次数，其对应的代码如下所示：

long allocCount = Debug.getGlobalAllocCount();

long allocSize = Debug.getGlobalAllocSize();

long gcCount = Debug.getGlobalGcInvocationCount();

并且，在 Android 6.0 及之后 可以拿到 更精准 的 GC 信息：

Debug.getRuntimeStat("art.gc.gc-count");

Debug.getRuntimeStat("art.gc.gc-time");

Debug.getRuntimeStat("art.gc.blocking-gc-count");

Debug.getRuntimeStat("art.gc.blocking-gc-time");

对于 GC 信息的排查，我们一般关注 阻塞式GC的次数和耗时，因为它会 暂停线程，可能导致应用发生 卡顿。建议 仅对重度场景使用。

## 6、建立线上 OOM 监控组件：Probe

美团的 Android 内存泄漏自动化链路分析组件 Probe 在 OOM 时会生成 Hprof 内存快照，然后，它会通过 单独进程 对这个 文件 做进一步 分析。

Probe 组件的缺陷及解决方案

它的缺点比较多，具体为如下几点：

1、在崩溃的时候生成内存快照容易导致二次崩溃。

2、部分手机生成 Hprof 快照比较耗时。

3、部分 OOM 是由虚拟内存不足导致。

在实现自动化链路分析组件 Probe 的过程中主要要解决两个问题，如下所示：

1、链路分析时间过长

1）、使用链路归并：将具有 相同层级与结构 的链路进行 合并。

2）、使用 自适应扩容法：通过不断比较现有链路和新链路，结合扩容因子，逐渐完善为完整的泄漏链路。

2、分析进程占用内存过大

分析进程占用的内存 跟 内存快照文件的大小 不成正相关，而跟 内存快照文件的 Instance 数量 呈 正相关。所以在开发过程中我们应该 尽可能排除不需要的Instance实例。

Prope 分析流程揭秘

Prope 的 总体架构图 如下所示：

而它的整个分析流程具体可以细分为八个步骤，如下所示：

1、hprof 映射到内存 => 解析成 Snapshot & 计数压缩：

解析后的 Snapshot 中的 Heap 有四种类型，具体为：

1）、DefaultHeap

2）、ImageHeap

3）、App Heap：包括 ClassInstance、ClassObj、ArrayInstance、RootObj。

4）、System Heap

解析完 后使用了 计数压缩策略，对 相同的 Instance 使用 计数，以 减少占用内存。超过计数阈值的需要计入计数桶（计数桶记录了 丢弃个数 和 每个 Instance 的大小）。

2、生成 Dominator Tree。

3、计算 RetainSize。

4、生成 Reference 链 && 基础数据类型增强:

如果对象是 基础数据类型，会将 自身的 RetainSize 累加到父节点 上，将 怀疑对象 替换为它的 父节点。

5、链路归并。

6、计数桶补偿 & 基础数据类型和父节点融合：

使用计数补偿策略计算 RetainSize，主要是 判断对象是否在计数桶中，如果在的话则将 丢弃的个数和大小补偿到对象上，累积计算RetainSize，最后对 RetainSize 排序以查找可疑对象。

7、排序扩容。

8、查找泄露链路。

## 7、实现 单机版 的 Profile Memory 自动化内存分析

项目地址请点击此处

在配置的时候要注意两个问题：

1、liballoc-lib.so在构建后工程的 build => intermediates => cmake 目录下。将对应的 cpu abi 目录拷贝到新建的 libs 目录下。

2、在 DumpPrinter Java 库的 build.gradle 中的 jar 闭包中需要加入以下代码以识别源码路径:

sourceSets.main.java.srcDirs = ['src']

使用步骤

具体的使用步骤如下所示：

1、首先，点击 ”开始记录“ 按钮可以看到触发对象分配的记录，说明对象已经开始记录对象的分配，log如下所示：

12-26 10:54:03.963 30450-30450/com.dodola.alloctrack I/AllocTracker: ====current alloc count 388=====

2、然后，点击多次 ”生成1000个对象“ 按钮，当对象达到设置的最大数量的时候触发内存dump，会得到保存数据路径的日志。如下所示：

12-26 10:54:03.963 30450-30450/com.dodola.alloctrack I/AllocTracker: ====current alloc count 388=====

12-26 10:56:45.103 30450-30450/com.dodola.alloctrack I/AllocTracker: saveARTAllocationData write file to /storage/emulated/0/crashDump/1577329005

3、此时，可以看到数据保存在 sdk 下的 crashDump 目录下。

4、接着，通过 gradle task :buildAlloctracker 任务编译出存放在 tools/DumpPrinter-1.0.jar 的 dump 工具，然后采用如下命令来将数据解析 到dump\_log.txt 文件中。

java -jar tools/DumpPrinter-1.0.jar dump文件路径 > dump\_log.txt

5、最后，就可以在 dump\_log.txt 文件中看到解析出来的数据，如下所示：

Found 4949 records:

tid=1 byte[] (94208 bytes)

dalvik.system.VMRuntime.newNonMovableArray (Native method)

android.graphics.Bitmap.nativeCreate (Native method)

android.graphics.Bitmap.createBitmap (Bitmap.java:975)

android.graphics.Bitmap.createBitmap (Bitmap.java:946)

android.graphics.Bitmap.createBitmap (Bitmap.java:913)

android.graphics.drawable.RippleDrawable.updateMaskShaderIfNeeded (RippleDrawable.java:776)

android.graphics.drawable.RippleDrawable.drawBackgroundAndRipples (RippleDrawable.java:860)

android.graphics.drawable.RippleDrawable.draw (RippleDrawable.java:700)

android.view.View.getDrawableRenderNode (View.java:17736)

android.view.View.drawBackground (View.java:17660)

android.view.View.draw (View.java:17467)

android.view.View.updateDisplayListIfDirty (View.java:16469)

android.view.ViewGroup.recreateChildDisplayList (ViewGroup.java:3905)

android.view.ViewGroup.dispatchGetDisplayList (ViewGroup.java:3885)

android.view.View.updateDisplayListIfDirty (View.java:16429)

android.view.ViewGroup.recreateChildDisplayList (ViewGroup.java:3905)

## 8、搭建线下 Native 内存泄漏监控体系

在 Android 8.0 及之后，可以使用 Address Sanitizer、Malloc 调试和 Malloc 钩子 进行 native 内存分析，参见 native\_memory

对于线下 Native 内存泄漏监控的建立，主要针对 是否能重编 so 的情况 来记录分配的内存信息。

针对无法重编so的情况

1）、首先，使用 PLT Hook 拦截库的内存分配函数，然后，重定向到我们自己的实现后去 记录分配的 内存地址、大小、来源so库路径 等信息。

2）、最后，定期 扫描分配与释放 的配对内存块，对于 不配对的分配 输出上述记录的信息。

针对可重编的so情况

1）、首先，通过 GCC 的 ”-finstrument-functions“ 参数给 所有函数插桩，然后，在桩中模拟调用栈的入栈与出栈操作。

2）、接着，通过 ld 的 ”--warp“ 参数 拦截内存分配和释放函数，重定向到我们自己的实现后记录分配的 内存地址、大小、来源so以及插桩调用栈此刻的内容。

3）、最后，定期扫描分配与释放是否配对，对于不配对的分配输出我们记录的信息。

## 9、设置内存兜底策略

设置内存兜底策略的目的，是为了 在用户无感知的情况下，在接近触发系统异常前，选择合适的场景杀死进程并将其重启，从而使得应用内存占用回到正常情况。

通常执行内存兜底策略时至少需要满足六个条件，如下所示：

1）、是否在主界面退到后台且位于后台时间超过 30min。

2）、当前时间为早上 2~5 点。

3）、不存在前台服务（通知栏、音乐播放栏等情况）。

4）、Java heap 必须大于当前进程最大可分配的85% || native内存大于800MB。

5）、vmsize 超过了4G（32bit）的85%。

6）、非大量的流量消耗（不超过1M/min） && 进程无大量CPU调度情况。

只有在满足了以上条件之后，我们才会去杀死当前主进程并通过 push 进程重新拉起及初始化。

## 10、更深入的内存优化策略

除了在 Android性能优化之内存优化 => 优化内存空间 中讲解过的一些常规的内存优化策略以外，在下面列举了一些更深入的内存优化策略。

1、使 bitmap 资源在 native 中分配

对于 Android 2.x 系统，使用反射将 BitmapFactory.Options 里面隐藏的 inNativeAlloc 打开。

对于 Android 4.x 系统，使用或借鉴 Fresco 将 bitmap 资源在 native 中分配的方式。

2、图片加载时的降级处理

使用 Glide、Fresco 等图片加载库，通过定制，在加载 bitmap 时，若发生 OOM，则使用 try catch 将其捕获，然后清除图片 cache，尝试降低 bitmap format（ARGB8888、RGB565、ARGB4444、ALPHA8）。

需要注意的是，OOM 是可以捕获的，只要 OOM 是由 try 语句中的对象声明所导致的，那么在 catch 语句中，是可以释放掉这些对象，解决 OOM 的问题的。

3、前台每隔 3 分钟去获取当前应用内存占最大内存的比例，超过设定的危险阈值（如80%）则主动释放应用 cache（Bitmap 为大头），并且显示地除去应用的 memory，以加速内存收集的过程。

计算当前应用内存占最大内存的比例的代码如下：

max = Runtime.getRuntime().maxMemory();

available = Runtime.getRuntime.totalMemory() - Runtime.getFreeMemory();

ratio = available / max;

显示地除去应用的 memory，以加速内存收集过程的代码如下所示：

WindowManagerGlobal.getInstance().startTrimMemory(TRIM\_MEMORY\_COMPLETE);

5、由于 webview 存在内存系统泄漏，还有 图库占用内存过多 的问题，可以采用单独的进程。

6、当UI隐藏时释放内存

当用户切换到其它应用并且你的应用 UI 不再可见时，应该释放应用 UI 所占用的所有内存资源。这能够显著增加系统缓存进程的能力，能够提升用户体验。

在所有 UI 组件都隐藏的时候会接收到 Activity 的 onTrimMemory() 回调并带有参数 TRIM\_MEMORY\_UI\_HIDDEN。

7、Activity 的兜底内存回收策略

在 Activity 的 onDestory 中递归释放其引用到的 Bitmap、DrawingCache 等资源，以降低发生内存泄漏时对应用内存的压力。

8、使用类似 Hack 的方式修复系统内存泄漏

LeakCanary 的 AndroidExcludeRefs 列出了一些由于系统原因导致引用无法释放的例子，可使用类似 Hack 的方式去修复。具体的实现代码可以参考 Booster => 系统问题修复。

9、应用发生 OOM 时，需要上传更加详细的内存相关信息。

10、当应用使用的Service不再使用时应该销毁它，建议使用 IntentServcie。

11、谨慎使用第三方库，避免为了使用其中一两个功能而导入一个大而全的解决方案。

# 六、内存优化演进

1、自动化测试阶段

内存达到阈值后自动触发 Hprof Dump，将得到的 Hprof 存档后由人工通过 MAT 进行分析。

2、LeakCanary

检测和分析报告都在一起，批量自动化测试和事后分析都不太方便。

3、使用基于 LeakCannary 的改进版 ResourceCanary

Matrix => ResourceCanary 实现原理

主要功能

目前，它的主要功能有 三个部分，如下所示：

1、分离 检测和分析 两部分流程

自动化测试由测试平台进行，分析则由监控平台的服务端离线完成，最后再通知相关开发解决问题。

2、裁剪 Hprof文件，以降低 传输 Hprof 文件与后台存储 Hprof 文件的开销

获取 需要的类和对象相关的字符串 信息即可，其它数据都可以在客户端裁剪，一般能 Hprof 大小会减小至原来的 1/10 左右。

3、增加重复 Bitmap 对象检测

方便通过减少冗余 Bitmap 的数量，以降低内存消耗。

4、小结

在研发阶段需要不断实现 更多的工具和组件，以此系统化地提升自动化程度，以最终 提升发现问题的效率。

# 七、内存优化工具

除了常用的内存分析工具 Memory Profiler、MAT、LeakCanary 之外，还有一些其它的内存分析工具。

# 八、内存问题总结

在我们进行内存优化的过程中，有许多内存问题都可以归结为一类问题，为了便于以后快速地解决类似的内存问题，我将它们归结成了以下的多个要点：

## 1、内类是有危险的编码方式

说道内类就不得不提到 ”this$0“，它是一种奇特的内类成员，每个类实例都具有一个 this$0，当它的内类需要访问它的成员时，内类就会持有外类的 this$0，通过 this$0 就可以访问外部类所有的成员。

解决方案是在 Activity 关闭，即触发 onDestory 时解除内类和外部的引用关系。

## 2、普通 Hanlder 内部类的问题

这也是一个 this$0 间接引用的问题，对于 Handler 的解决方案一般可以归结为如下三个步骤：

1）、把内类声明成 static：用来断绝 this$0 的引用。因为 static 描述的内类从 Java 编译原理的角度看，”内类“与”外类“相互独立，互相都没有访问对方成员变量的能力。

2、使用 WeakReference 来引用外部类的实例。

3、在外部类（如 Activity）销毁的时候使用 removeCallbackAndMessages 来移除回调和消息。

这里需要在使用过程中注意对 WeakReference 进行判空。

## 3、登录界面的内存问题

如果在闪屏页跳转到登录界面时没有调用 finish()，则会造成闪屏页的内存泄漏，在碰到这种”过渡界面“的情况时，需要注意不要产生这样的内存 Bug。

## 4、使用系统服务时产生的内存问题

我们通常都会使用 getSystemService 方法来获取系统服务，但是当在 Activity 中调用时，会默认把 Activity 的 Context 传给系统服务，在某些不确定的情况下，某些系统服务内部会产生异常，从而 hold 住外界传入的 Context。

解决方案是 直接使用 Applicaiton 的 Context 去获取系统服务。

## 5、把 WebView 类型的泄漏装进垃圾桶进程

我们都知道，对应 WebView 来说，其 网络延时、引擎 Session 管理、Cookies 管理、引擎内核线程、HTML5 调用系统声音、视频播放组件等产生的引用链条无法及时打断，造成的内存问题基本上可以用”无解“来形容。

解决方案是我们可以 把 WebView 装入另一个进程。 具体为在 AndroidManifes 中对当前的 Activity 设置 android:process 属性即可，最后，在 Activity 的 onDestory 中退出进程，这样即可基本上终结 WebView 造成的泄漏。

## 6、在适当的时候对组件进行注销

我们在平常开发过程中经常需要在Activity创建的时候去注册一些组件，如广播、定时器、事件总线等等。这个时候我们应该在适当的时候对组件进行注销，如 onPause 或 onDestory 方法中。

## 7、Handler / FrameLayout 的 postDelyed 方法触发的内存问题

不仅在使用 Handler 的 sendMessage 方法时，我们需要在 onDestory 中使用 removeCallbackAndMessage 移除回调和消息，在使用到 Handler / FrameLayout 的 postDelyed 方法时，我们需要调用 removeCallbacks 去移除实现控件内部的延时器对 Runnable 内类的持有。

## 8、图片放错资源目录也会有内存问题

在做资源适配的时候，因为需要考虑到 APK 的瘦身问题，无法为每张图片在每个 drawable / mipmap 目录下安置一张适配图片的副本。很多同学不知道图片应该放哪个目录，如果放到分辨率低的目录如 hdpi 目录，则可能会造成内存问题，这个时候建议尽量问设计人员要高品质图片然后往高密度目录下方，如 xxhdpi 目录，这样 在低密屏上”放大倍数“是小于1的，在保证画质的前提下，内存也是可控的。也可以使用 Drawable.createFromSream 替换 getResources().getDrawable 来加载，这样便可以绕过 Android 的默认适配规则。

对于已经被用户使用物理“返回键”退回到后台的进程，如果包含了以下 两点，则 不会被轻易杀死。

1）、进程包含了服务 startService，而服务本身调用了 startForeground（低版本需通过反射调用）。

2）、主 Activity 没有实现 onSaveInstanceState 接口。

但建议 在运行一段时间（如3小时）后主动保存界面进程（位于后台），然后重启它，这样可以有效地降低内存负载。

## 9、列表 item 被回收时注意释放图片的引用

我们应该在 item 被回收不可见时去释放掉对图片的引用。如果你使用的是 ListView，由于每次 item 被回收后被再次利用都会去重新绑定数据，所以只需在 ImageView 回调其 onDetchFromWindow 方法的时候区释放掉图片的引用即可。如果你使用的是 RecyclerView，因为被回收不可见时第一次选择是放进 mCacheView中，但是这里面的 item 被复用时并不会去执行 bindViewHolder 来重新绑定数据，只有被回收进 mRecyclePool 后拿出来复用才会重新绑定数据。所以此时我们应该在 item 被回收进 RecyclePool 的时候去释放图片的引用，这里我们只要去 重写 Adapter 中的 onViewRecycled 方法 就可以了，代码如下所示：

@Override

public void onViewRecycled(@Nullable VH holder) {

super.onViewRecycled(holder);

if (holder != null) {

//做释放图片引用的操作

}

}

## 10、使用 ViewStub 进行占位

我们应该使用 ViewStub 对那些没有马上用到的资源去做延迟加载，并且还有很多大概率不会出现的 View 更要去做懒加载，这样可以等到要使用时再去为它们分配相应的内存。

## 11、注意定时清理 App 过时的埋点数据

产品或者运营为了统计数据会在每个版本中不断地增加新的埋点。所以我们需要定期地去清理一些过时的埋点，以此来 适当地优化内存以及CPU的压力。

## 12、针对匿名内部类 Runnable 造成内存泄漏的处理

我们在做子线程操作的时候，喜欢使用匿名内部类 Runnable 来操作。但是，如果某个 Activity 放在线程池中的任务不能及时执行完毕，在 Activity 销毁时很容易导致内存泄漏。因为这个匿名内部类 Runnable 类持有一个指向 Outer 类的引用，这样一来如果 Activity 里面的 Runnable 不能及时执行，就会使它外围的 Activity 无法释放，产生内存泄漏。从上面的分析可知，只要在 Activity 退出时没有这个引用即可，那我们就通过反射，在 Runnable 进入线程池前先干掉它，代码如下所示：

Field f = job.getClass().getDeclaredField("this$0");

f.setAccessible(true);

f.set(job, null);

这个任务就是我们的 Runnable 对象，而 ”this$0“ 就是上面所指的外部类的引用了。这里注意使用 WeakReference 装起来，要执行了先 get 一下，如果是 null 则说明 Activity 已经回收，任务就放弃执行。

# 九、内存优化常见问题

## 1、你们内存优化项目的过程是怎么做的？

1、分析现状、确认问题

我们发现我们的 APP 在内存方面可能存在很大的问题，第一方面的原因是我们的线上的 OOM 率比较高。

第二点呢，我们经常会看到在我们的 Android Studio 的 Profiler 工具中内存的抖动比较频繁。

这是我们一个初步的现状，然后在我们知道了这个初步的现状之后，进行了问题的确认，我们经过一系列的调研以及深入研究，我们最终发现我们的项目中存在以下几点大问题，比如说：内存抖动、内存溢出、内存泄漏，还有我们的Bitmap 使用非常粗犷。

2、针对性优化

比如 内存抖动的解决 => Memory Profiler 工具的使用（呈现了锯齿张图形） => 分析到具体代码存在的问题（频繁被调用的方法中出现了日志字符串的拼接），也可以说说 内存泄漏或内存溢出的解决。

3、效率提升

为了不增加业务同学的工作量，我们使用了一些工具类或 ARTHook 这样的 大图检测方案，没有任何的侵入性。同时，我们将这些技术教给了大家，然后让大家一起进行 工作效率上的提升。

我们对内存优化工具Profiler Memory、MAT 的使用比较熟悉，因此 针对一系列不同问题的情况，我们写了 一系列解决方案的文档，分享给大家。这样，我们 整个团队成员的内存优化意识就变强 了。

## 2、你做了内存优化最大的感受是什么？

1、磨刀不误砍柴工

我们一开始并没有直接去分析项目中代码哪些地方存在内存问题，而是先去学习了 Google 官方的一些文档，比如说学习了 Memory Profiler 工具的使用、学习了 MAT 工具的使用，在我们将这些工具学习熟练之后，当在我们的项目中遇到内存问题时，我们就能够很快地进行排查定位问题进行解决。

2、技术优化必须结合业务代码

一开始，我们做了整体 APP 运行阶段的一个内存上报，然后，我们在一些重点的内存消耗模块进行了一些监控，但是，后面发现这些监控并没有紧密地结合我们的业务代码，比如说在梳理完项目之后，发现我们项目中存在使用多个图片库的情况，多个图片库的内存缓存肯定是不公用的，所以 导致我们整个项目的内存使用量非常高。所以进行技术优化时必须结合我们的业务代码。

3、系统化完善解决方案

我们在做内存优化的过程中，不仅做了 Android 端的优化工作，还将我们 Android 端一些数据的采集上报到了我们的服务器，然后传到我们的 APM 后台，这样，方便我们的无论是 Bug 跟踪人员或者是 Crash 跟踪人员进行一系列问题的解决。

## 3、如何检测所有不合理的地方？

比如说 大图片的检测，我们最初的一个方案是通过继承 ImageView，重写 它的 onDraw 方法来实现。但是，我们在推广它的过程中，发现很多开发人员并不接受，因为很多 ImageView 之前已经写过了，你现在让他去替换，工作成本是比较高的。所以说，后来我们就想，有没有一种方案可以 免替换，最终我们就找到了 ARTHook 这样一个 Hook 的方案。

# 十、总结

对于内存优化的专项优化 而言，我们要着重注意两点，即 优化大方向 和 优化细节。

1、优化大方向

对于 优化的大方向，我们应该 优先去做见效快的地方，主要有以下三部分：

1）、内存泄漏

2）、内存抖动

3）、Bitmap

2、优化细节

对于 优化细节，我们应该 注意一些系统属性或内存回调的使用 等等，主要可以细分为如下六部分：

1）、LargeHeap 属性

2）、onTrimMemory / onLowMemory

3）、使用优化过后的集合：如 SparseArray 类簇

4）、谨慎使用 SharedPreference

5）、谨慎使用外部库

6）、业务架构设计合理

3、内存优化体系化建设总结

在这篇文章中，我们除了建立了 内存的监控闭环 这一核心体系之外，还实现了以下 十大组件 / 策略：

1）、根据设备分级来使用不同的内存和分配回收策略。

2）、针对低端机做了功能或图片加载格式的降级处理。

3）、针对缓存滥用的问题实现了统一的缓存管理组件。

4）、实现了大图监控和重复图片的监控。

5）、在前台每隔一定时间去获取当前应用内存占最大内存的比例，当超过设定阈值时则主动释放应用 cache。

6）、当 UI 隐藏时释放内存以增加系统缓存应用进程的能力。

7）、高效实现了应用全局内的 Bitmap 监控。

8）、实现了全局的线程监控。

9）、针对内存使用的重度场景实现了 GC 监控。

10）、实现了线下的 native 内存泄漏监控。

最后，当监控到 应用内存超过阈值时，还定制了 完善的兜底策略 来 重启应用进程。

总的来看，要建立一套 全面且成体系的内存优化及监控 是非常重要也是极具挑战性的一项工作。并且，目前各大公司的 内存优化体系 也正处于 不断演进的历程 之中，其目的不外乎：实现更健全的功能、更深层次的定位问题、快速准确地发现线上问题。