深入探索Android内存优化

转载自：深入探索Android内存优化

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

本篇是Android内存优化的进阶篇，难度会比较大，建议对内存优化不是非常熟悉的前仔细看看在前几篇文章中，笔者曾经写过的一篇Android性能优化之内存优化，其中详细分析了以下几大模块：

- Android的内存管理机制
- 优化内存的意义
- 避免内存泄漏
- 优化内存空间
- 图片管理模块的设计与实现

如果你对以上基础内容都比较了解了，那么我们便开始接下来的Android内存优化探索之旅吧。

一、内存优化相关概念

Android的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：

- 硬件快：内存本身读取、存入速度快。
- 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：把从内存淘汰的图片，降低压缩比存储到本地，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。)

接下来，我们来了解一下内存优化的一些重要概念。

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“。

```
1 LPDDR系列的带宽 = 时钟频率 ×内存总线位数 / 8
2 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，希望可以做到”用时分配，及时释放“。

内存优化的纬度

对于Android内存优化来说又可以细分为两个维度：

1、RAM优化

主要是降低运行时内存。它的目的如下：

- 防止应用发生OOM。
- 降低应用由于内存过大被LMK机制杀死的概率。
- 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。APK瘦身。它的目的为：

- 降低应用占用空间，避免因ROM空间不足导致程序无法安装

内存问题

那么，内存问题主要是有哪几类呢？下面我来一一叙述：

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。

2、内存泄漏

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。
可用内存减少、频繁GC，容易导致内存泄漏。

3、内存溢出

OOM、程序异常。

二、常见工具选择

在内存优化的上一篇我们已经介绍过了相关的工具，这里再简单回忆一下。

1、Memory Profiler

它的作用如下：

- 实时图表展示应用内存使用量
- 识别内存泄漏、抖动等
- 提供捕获堆转储、强制GC以及根据内存分配的能力

它的优点即：

- 方便直观
- 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、分析问题等。建议线下深入使用。

3、LeakCanary

自动内存泄漏检测神器。仅用于线下集成。

它的缺点比较明显，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。
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。

4、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**。haha库的链接地址。
它的基本用法如下所示：

```
// 导出堆栈文件
File heapDumpFile = ...
Debug.dumpHprofData(heapDumpFile.getAbsolutePath());
// 根据堆栈文件创建出内存映射文件缓冲区
DataBuffer buffer = new MemoryMappedFileBuffer(heapDumpFile);
// 根据文件缓存区创建出对应的快照
Snapshot snapshot = Snapshot.createSnapshot(buffer);
// 从快照中获取指定的类
ClassObj someClass = snapshot.findClass("com.example.SomeClass");
```

在实现线上版的LeakCanary的时候主要要做2个工作：

- 1、在过程中加上对大对象的分析过程。
- 2、解决掉将hprof文件映射到内存中的时候可能内存暴涨的问题。

5、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。例如，当LeakCanary上发现内存泄漏时，手机将上传内存快照至服务器，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，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。

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的内存分配区域为如下几部分：

- 方法区:主要存放静态常量
- 虚拟机栈：Java变量引用
- 本地方法栈：native变量引用
- 堆：对象
- 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：

- 标记所有需要回收的对象
- 统一回收所有被标记的对象

它的优点实现比较简单，缺点也很明显：

- 标记、清除效率不高
- 产生大量内存碎片

2、复制算法

流程可简述为三步：

- 将内存划分为大小相等的两块
- 一块内存用完之后复制存活对象到另一块
- 清理另一块内存

它的优点为 实现简单，运行高效，每次仅需遍历标记一半的内存区域。而缺点则会浪费一半空间，代价大。

3、标记-整理算法

流程可简述为三步：

- 标记过程与”标记-清除算法“一样
- 存活对象往一端进行移动
- 清理其余内存

它的优点如下：

- 避免标记-清除导致的内存碎片
- 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带收集算法，它具有如下特点：

- 结合多种算法优势
- 新生代对象存活率低，复制
- 老年代对象存活率高，标记-整理

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场景其实由细分为两种，一种是内存真正不足
了，二另一种则是可用内存不足。要注意一下这两种的区分。
以Android中的虚拟机的角度来说，我们要清楚Dalvik与Art区别，Dalvik仅固定一种回收算法，而Art回收算法可运行期选择，并且，Art具备内存整理能力，减少内存空洞。
最后，LMK机制（Low Memory killer）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。

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**。
它的危害也很严重，通常会导致页面卡顿，甚至造成OOM。

那么为什么内存抖动会导致OOM?

主要原因有两点：

- 频繁创建对象，导致内存不足及碎片（不连续）
- 不连续的内存片无法被分配，导致OOM

内存抖动解决实战

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或CPU Profiler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调用的地方。
下面列举一些导致内存抖动的常见案例：

1、字符串使用加号拼接：

- 使用StringBuilder替代。
- 初始化时设置容量，减少StringBuilder的扩容。

2、资源复用

- 使用全局缓存池，以重用频繁申请和释放的对象。
- 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

- ondraw、getView中对象的创建尽量进行复用。
- 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类族来替代HashMap。

五、内存优化体系搭建

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。
内存泄漏的分析一般可简述为两步：

- 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

首先找到当前Activity，在Histogram中选择其List Objects中的 with incoming reference（哪些强引用引向了我），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。
最后找到最后的泄漏对象在左下角下会有一个小圆圈。

MAT的关键使用细节

要全面掌握MAT的用法，必须先了解下面的一些细节：

- 善于使用Regex查找对应泄漏类。
- 使用group by package查找对应包下的具体类。

其次，要明白with outgoing references和with incoming references的区别。
with outgoing references为它引用了哪些对象，with incoming references为哪些对象引用了它。
还需要了解Shallow Heap和Retained Heap的区别。
Shallow Heap为对象自身占用的内存，而Retained Heap则还包含对象引用的对象所占用的内存。
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，他们分别是Dominator_tree、Histogram、thread_overview、Top Consumers、Leak Suspects。下面我们简单地了解一下它们。

Dominator（支配者）：

如果从GC Root到达对象A的路径上必须经过对象B，那么B就是A的支配者。

Histogram和dominator_tree的区别：

- Histogram显示Shallow Heap、Retained Heap、Objects，而dominator_tree显示的是Shallow Heap、Retained Heap、Percentage。
- Histogram基于类的角度，dominator_tree是基于实例的角度。Histogram不会具体显示每一个泄漏的对象，而dominator_tree会。

thread_overview

查看有多少线程和线程的Shallow Heap、Retained Heap、Context Class Loader与is Daemon。

Top Consumers

通过图形的形式列出占用内存比较多的对象。
在下方的Biggest Objects还可以查看其相对比较详细的信息，如Shallow Heap、Retained Heap。

Leak Suspects

列出有内存泄漏的地方，点击Details可以查看其产生内存泄漏的引用链。
最后，我列举一些内存泄漏优化的技巧：

- 1、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。
- 2、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，降低发生内存泄漏对应用内存的压力。

2、建立线上内存泄漏监控组件：使用定制化的LeakCanary

在线上也可以使用类似LeakCanary的自动化检测方案，但是需要对生成的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、建立线上OOM监控组件：Probe

美团Android内存泄漏自动化链路分析组件Probe
在OOM时生成Hprof内存快照，然后通过单独进程对这个文件做进一步分析。
它的缺点比较多，具体为如下几点：

- 在崩溃的时候生成内存快照容易导致二次崩溃。
- 部分手机生成Hprof快照比较耗时。
- 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如下问题：

1、链路分析时间过长

- 使用链路归并，将具有相同层级与结构的链路进行合并。
- 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。所以应该尽可能排除不需要的Instance实例。

Prope分析流程

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：

- DefaultHeap
- ImageHeap
- App Heap：包括ClassInstance、ClassObj、ArrayInstance、RootObj。
- 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。
总体架构图如下：

4、实现单机版的Profile – Memory自动化内存分析

项目地址点击此处

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']
```

具体的使用步骤如下：

1、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。

```
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。

```
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、可以看到数据保存在sdk下的crashDump目录下。

4、此时，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)
```

5、图片监控体系搭建

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：

在Android 3.0之前

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

Bitmap全部放在Java Heap中的缺点很明显：

- 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及之后

- 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的时候还可以考虑这些Bitmap内存以防止被滥用。
- 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，如何将图片内存存放在Native中呢?

- 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，当系统内存不足，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，我们如何评估内存对应用性能的影响呢?

主要从以下两个方面进行评估:

- 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，我们可以从以下几个方面来进行。

1、设备分级

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。

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。

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。

2、建立统一的缓存管理组件

建立统一的缓存管理组件，合理使用OnTrimMemory回调，根据系统不同的状态去释放相应的内存。

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

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。

3、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。

4、统一图片库

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

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的宽度甚至图片宽度），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）。

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。

下面介绍一下ARTHook的方案。

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，主要用于以下几方面:

- 1、AOP变成

- 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

Epic的使用可简述为：

1、在build.gradle中添加

```
compile 'me.weishu:epic:0.6.0'
```

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..");
    }
}
```

3、注入Hook好的方法：

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

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

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) thro
```

```
ws Throwable {
            super.afterHookedMethod(param);
        // 这里找到所有通过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类以便于
        // 重写afterHookedMethod方法拿到相应的参数进行监控逻辑的判断
        DexposedBridge.findAndHookMethod(ImageView.class, "setImageB
itmap", Bitmap.class, new ImageHook());
        }
    });
```

接下来，我们来实现我们的ImageHook类，如下所示：

```
public class ImageHook extends XC_MethodHook {
    @Override
    protected void afterHookedMethod(MethodHookParam param) throws T
hrowable {
        super.afterHookedMethod(param);
        // 实现我们的逻辑
        ImageView imageView = (ImageView) param.thisObject;
        checkBitmap(imageView,((ImageView) param.thisObject).getDraw
able());
    }
    private static void checkBitmap(Object thiz, Drawable drawable)
 {
        if (drawable instanceof BitmapDrawable && thiz instanceof Vi
ew) {
            final Bitmap bitmap = ((BitmapDrawable) drawable).getBit
map();
            if (bitmap != null) {
                final View view = (View) thiz;
                int width = view.getWidth();
                int height = view.getHeight();
                if (width > 0 && height > 0) {
                    // 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(), widt
h, height, new RuntimeException("Bitmap size too large"));
                }
                } else {
```

```
23                      // 当宽高度等于0时，说明ImageView还没有进行绘制，使用Vie
wTreeObserver进行大图检测的处理。
24                      final Throwable stackTrace = new RuntimeExceptio
n();
25                      view.getViewTreeObserver().addOnPreDrawListener
(new ViewTreeObserver.OnPreDrawListener() {
26                          @Override
27                          public boolean onPreDraw() {
28                              int w = view.getWidth();
29                              int h = view.getHeight();
30                              if (w > 0 && h > 0) {
31                                  if (bitmap.getWidth() >= (w << 1)
32                                      && bitmap.getHeight() >= (h <<
 1)) {
33                                      warn(bitmap.getWidth(), bitmap.g
etHeight(), w, h, stackTrace);
34                                  }
35                                  view.getViewTreeObserver().removeOnP
reDrawListener(this);
36                              }
37                              return true;
38                          }
39                      });
40                  }
41              }
42          }
43      }
44      private static void warn(int bitmapWidth, int bitmapHeight, int
 viewWidth, int viewHeight, Throwable t) {
45          String warnInfo = "Bitmap size too large: " +
46              "\n real size: (" + bitmapWidth + ',' + bitmapHeight +
 ')' +
47              "\n desired size: (" + viewWidth + ',' + viewHeight +
 ')' +
48              "\n call stack trace: \n" + Log.getStackTraceString(t) +
'\n';
49          LogHelper.i(warnInfo);
50      }
51  }
```

在上面，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，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，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ViewTreeObserer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实现了。

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

项目地址

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。

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。具体实现步骤如下：

- 1、获取 android.graphics.Bitmap 实例对象的 mBuffer 为 ArrayInstance ，通过 getValues 获取数据为 Object 类型，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、根据 mBuffer 的数据生成 png 图片文件，参考了 https://github.com/JetBrains/adt-tools-base/blob/master/ddmlib/src/main/java/com/android/ddmlib/BitmapDecoder.java 实现。
- 3、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。

- 1、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”属性即可获取他们的hashcode来判断相同
- 8、最后通过instance中mNextInstanceToGcRoot获取整个引用链信息并打印。

在实现图片内存监控的过程中，应注意一下两点：

- 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。

7、建立全局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：

- 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，注意要包含调用外部库产生的Bitmap。
- 创建频率比较低。
- 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，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。
- 2、这个方案性能消耗很低，可以在正式环境中进行。注意正式与测试环境需要采用不同程度的监控。

6、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。
对线程数量的限制，一定程度上可以避免OOM的发生。

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，将当前的线程信息上报并预警。

7、建立线上应用内存监控体系

具体的相关数据获取方式如下：

- 1、首先，ActivityManager的getProcessMemoryInfo -> Debug.MemoryInfo数据。
- 2、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，进而获得细分内存使用情况。
- 3、通过Runtime获取DalvikHeap。
- 4、通过Debug.getNativeHeapAllocatedSize获取NativeHeap。

对于监控场景，需要划分为两大类：

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。内存监控方法有多种实现方式，我们先来介绍几种常规方式。
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：

- 1、超过最大内存的80%
- 2、内存Dump
- 3、回传文件

- 4、MAT手动分析

但是有如下缺点：

- 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。

LeakCanary带到线上的方式：

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：

- 不适合所有情况，需要预设怀疑点。
- 分析比较耗时，容易导致OOM。

定制LeakCanary方式

定制LeakCanary需要解决以上产生的一些问题，下面这里分别列出对应的解决方案：

- 1、预设怀疑点->自动找怀疑点。
- 2、分析泄漏链路慢->分析Retain size大的对象。
- 3、分析OOM->对象裁剪，不全部加载到内存。

2、低内存监控

- 利用onLowMemory、onTrimMemory监听物理内存警告。
- 代码设置超过虚拟内存大小最大限制的90%则直接触发内存警告。
- 对于监控指标，一般为：发生频率、发生时各项内存使用状况、发生时App的当前场景。

并且，为了准确衡量内存性能，我们引入了内存异常率和触顶率的指标。

内存异常率

内存UV异常率 = PSS 超过400MB的UV / 采集UV，PSS获取：通过Debug.MemoryInfo。

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。

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%最大堆限制，GC会变得更加频发，容易造成OOM和卡顿。

这里小结一下，客户端只负责上报数据，由后台来计算平均PSS、图片内存、Java内存、异常率、触顶率等指标值，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，建立线上监控完整方案需包含以下几点：

- 待机内存、重点模块内存、OOM率。
- 整体及重点模块GC次数、GC时间。
- 增强的LeakCanry自动化内存泄漏分析。
- 低内存监控模块的设置。

8、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，代码如下所示：

```
long allocCount = Debug.getGlobalAllocCount();
long allocSize = Debug.getGlobalAllocSize();
long gcCount = Debug.getGlobalGcInvocationCount();
```

并且，在Android 6.0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");
```

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。

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。
一般进行执行内存兜底策略时需要满足以下条件：

- 是否在主界面退到后台且位于后台时间超过30min。
- 当前时间为早上2~5点。
- 不存在前台服务（通知栏、音乐播放栏等情况）。
- java heap必须大于当前进程最大可分配的85% || native内存大于800MB
- vmsize超过了4G（32bit）的85%。
- 非大量的流量消耗（不超过1M/min） && 进程无大量CPU调度情况。

满足以上条件则杀死当前主进程并通过push进程重新拉起及初始化。

10、内存优化的一些策略

下面列举一些我在内存优化过程中常用的一些策略。

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Fresco将bitmap资源在native中分配。

2、使用inSampleSize避免不必要的大图加载。

3、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）。

4、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，以加速内存收集的过程的代码如下：

```
WindowManagerGlobal.getInstance().startTrimMemory(TRIM_MEMORY_COMPLET
E);
```

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、应用发生OOM时，需要上传更加详细的内存相关信息。

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

8、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。

9、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、线下Native内存泄漏监控搭建

在Android 8.0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的情况来进行记录分配的内存信息。

针对无法重编so的情况

- 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。
- 定期扫描分配与释放释放配对，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

- 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。
- 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。
- 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。

七、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不太方便。

3、使用基于LeakCannary的改进版ResourceCanary

它的主要特点如下：

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，再通知相关开发解决问题。

2、裁剪Hprof文件，以降低后台存储Hprof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。

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。

八、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，下面我将一一为大家进行介绍。

1、top

top命令是Linux下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于Windows的任务管理器。top命令提供了实时的对系统处理器的状态监视。它将显示系统中CPU最“敏感”的任务列表。该命令可以按CPU使用、内存使用和执行时间对任务进行排序。
接下来，我们输入以下命令查看top命令的用法：

```
quchao@quchaodeMacBook-Pro ~ % adb shell top --help
usage: top [-Hbq] [-k FIELD,] [-o FIELD,] [-s SORT] [-n NUMBER] [-d
 SECONDS] [-p PID,] [-u USER,]
Show process activity in real time.
-H    Show threads
-k    Fallback sort FIELDS (default -S,-%CPU,-ETIME,-PID)
-o    Show FIELDS (def PID,USER,PR,NI,VIRT,RES,SHR,S,%CPU,%MEM,TIME
+,CMDLINE)
-O    Add FIELDS (replacing PR,NI,VIRT,RES,SHR,S from default)
-s    Sort by field number (1-X, default 9)
```

```
 -b    Batch mode (no tty)
 -d    Delay SECONDS between each cycle (default 3)
 -n    Exit after NUMBER iterations
 -p    Show these PIDs
 -u    Show these USERs
 -q    Quiet (no header lines)
Cursor LEFT/RIGHT to change sort, UP/DOWN move list, space to force
update, R to reverse sort, Q to exit.
```

这里使用top仅显示一次进程信息，以便来讲解进程信息中各字段的含义。

前四行是当前系统情况整体的统计信息区。下面我们看每一行信息的具体意义。
第一行，Tasks — 任务（进程），具体信息说明如下：
系统现在共有729个进程，其中处于运行中的有1个，715个在休眠（sleep），stoped状态的有0个，zombie状态（僵尸）的有8个。
第二行,内存状态，具体信息如下：
5847124k total — 物理内存总量（5.8GB）
5758016k used — 使用中的内存总量（5.7GB）
89108k free — 空闲内存总量（89MB）
112428k buffers — 缓存的内存量 （112M）
第三行，swap交换分区信息，具体信息说明如下：
2621436k total — 交换区总量（2.6GB）
612572k used — 使用的交换区总量（612MB）
2008864k free — 空闲交换区总量（2GB）
2657696k cached — 缓冲的交换区总量（2.6GB）
第四行，cpu状态信息，具体属性说明如下：
800%cpu – 8核CPU。
39%user – 39%CPU被用户进程使用。
0%nice – 优先值为负的进程占0%。
42%sys — 内核空间占用CPU的百分比为42%。
712%idle – 除IO等待时间以外的其它等待时间为712%。
0%iow – IO等待时间占0%。
0%irq – 硬中断时间占0%。
6%sirq – 软中断时间占0%。
对于内存监控，在top里我们要时刻监控第三行swap交换分区的used，如果这个数值在不断的变化，说明内核在不断进行内存和swap的数据交换，这是真正的内存不够用了。
在第五行及以下，就是各进程（任务）的状态监控，项目列信息说明如下：
PID — 进程id。
USER — 进程所有者。
PR — 进程优先级。

NI — nice值。负值表示高优先级，正值表示低优先级。
VIRT — 进程使用的虚拟内存总量。VIRT = SWAP + RES。
RES — 进程使用的、未被换出的物理内存大小。RES = CODE + DATA。
SHR — 共享内存大小。
S — 进程状态。D=不可中断的睡眠状态、R=运行、 S=睡眠、T=跟踪/停止、Z=僵尸进程。
%CPU — 上次更新到现在的CPU时间占用百分比。
%MEM — 进程使用的物理内存百分比。
TIME+ — 进程使用的CPU时间总计，单位1/100秒。
ARGS — 进程名称（命令名/命令行）。
这里可以看到第一行的就是Awesome-WanAndroid这个应用的进程，它的进程名称为json.chao.com.w+，PID为23104，进程所有者USER为u0_a714，进程优先级PR为10，nice置NI为-10。进程使用的虚拟内存总量VIRT为4.3GB，进程使用的、未被换出的物理内存大小RES为138M，共享内存大小SHR为66M，进程状态S是睡眠状态，上次更新到现在的CPU时间占用百分比%CPU为21.2。进程使用的物理内存百分比%MEM为2.4%，进程使用的CPU时间TIME+为1:47.58/100小时。

2、dumpsys meminfo

在讲解dumpsys meminfo命令之前，我们必须先了解下Android中的几个内存指标的概念：

	内存指标	英文全称	含义	等价
USS	Unique Set Size	物理内存	进程独占的内存	
PSS	Proportional Set Size	物理内存	PSS = USS + 按比例包含共享库	
RSS	Resident Set Size	物理内存	RSS= USS+ 包含共享库	
VSS	Virtual Set Size	虚拟内存	VSS= RSS+ 未分配实际物理内存	

从上可知，它们之间内存的大小关系为VSS >= RSS >= PSS >= USS。
RSS与PSS相似，也包含进程共享内存，但比较麻烦的是RSS并没有把共享内存大小全都平分到使用共享的进程头上，以至于所有进程的RSS相加会超过物理内存很多。而VSS是虚拟地址，它的上限与进程的可访问地址空间有关，和当前进程的内存使用关系并不大。比如有很多的map内存也被算在其中，我们都知道，file的map内存对应的可能是一个文件或硬盘，或者某个奇怪的设备，它与进程使用内存并没有多少关系。
而PSS、USS最大的不同在于“共享内存“（比如两个App使用MMAP方式打开同一个文件，那么打开文件而使用的这部分内存就是共享的），USS不包含进程间共享的内存，而PSS包含。这也造成了USS因为缺少共享内存，所有进程的USS相加要小于物理内存大小的原因。
最早的时候官方就推荐使用PSS曲线图来衡量App的物理内存占用，而Android 4.4之后才加入USS。但是PSS，有个很大的问题，就是”共享内存“，考虑一种情况，如果A进程与B进程都会使用一个共享SO库，那么so库中初始化所用掉的那部分内存就会平分到A与B的头上。但是A是在B之后启动的，那么对于B的

PSS曲线而言，在A启动的那一刻，即使B没有做任何事情，也会出现一个比较大的阶梯状下滑，这会给用曲线图分析软件内存的行为造成致命的麻烦。
USS虽然没有这个问题，但是由于Dalvik虚拟机申请内存牵扯到GC时延和多种GC策略，这些都会影响到曲线的异常波动。比如异步GC是Android 4.0以上系统很重要的特性，但是GC什么时候结束？曲线什么时候”降低“？就无法预计了。还有GC策略，什么时候开始增加Dalvik虚拟机的预申请内幕才能大小（Dalvik启动时是由一个标称的start内存大小的,为Java代码运行时预留，避免Java运行时再申请而造成卡顿），但是这个预申请大小是动态变化的，这一点也会造成USS忽大忽小。
了解完Android内存的性能指标之后，下面我们便来说说dumpsys meminfo这个命令的用法，首先我们输入adb shell dumpsys meminfo -h查看它的帮助文档：

```
quchao@quchaodeMacBook-Pro ~ % adb shell dumpsys meminfo -h
meminfo dump options: [-a] [-d] [-c] [-s] [--oom] [process]
-a: include all available information for each process.
-d: include dalvik details.
-c: dump in a compact machine-parseable representation.
-s: dump only summary of application memory usage.
-S: dump also SwapPss.
--oom: only show processes organized by oom adj.
--local: only collect details locally, don't call process.
--package: interpret process arg as package, dumping all
            processes that have loaded that package.
--checkin: dump data for a checkin
If [process] is specified it can be the name or
pid of a specific process to dump.
```

接着，我们之间输入adb shell dumpsys meminfo命令：

```
quchao@quchaodeMacBook-Pro ~ % adb shell dumpsys meminfo
Applications Memory Usage (in Kilobytes):
Uptime: 257501238 Realtime: 257501238
// 根据进程PSS占用值从大到小排序
Total PSS by process:
    308,049K: com.tencent.mm (pid 3760 / activities)
    225,081K: system (pid 2088)
    189,038K: com.android.systemui (pid 2297 / activities)
    188,877K: com.miui.home (pid 2672 / activities)
    176,665K: com.plan.kot32.tomatotime (pid 22744 / activities)
    175,231K: json.chao.com.wanandroid (pid 23104 / activities)
    126,918K: com.tencent.mobileqq (pid 23741)
    ...
```

```
// 以oom来划分，会详细列举所有的类别的进程
Total PSS by OOM adjustment:
    432,013K: Native
        76,700K: surfaceflinger (pid 784)
        59,084K: android.hardware.camera.provider@2.4-service (pid 743)
        26,524K: transport (pid 23418)
        25,249K: logd (pid 597)
        11,413K: media.codec (pid 1303)
        10,648K: rild (pid 1304)
        9,283K: media.extractor (pid 1297)
        ...
    661,294K: Persistent
        225,081K: system (pid 2088)
        189,038K: com.android.systemui (pid 2297 / activities)
        103,050K: com.xiaomi.finddevice (pid 3134)
        39,098K: com.android.phone (pid 2656)
        25,583K: com.miui.daemon (pid 3078)
        ...
    219,795K: Foreground
        175,231K: json.chao.com.wanandroid (pid 23104 / activities)
        44,564K: com.miui.securitycenter.remote (pid 2986)
    246,529K: Visible
        71,002K: com.sohu.inputmethod.sogou.xiaomi (pid 4820)
        52,305K: com.miui.miwallpaper (pid 2579)
        40,982K: com.miui.powerkeeper (pid 3218)
        24,604K: com.miui.systemAdSolution (pid 7986)
        14,198K: com.xiaomi.metoknlp (pid 3506)
        13,820K: com.miui.voiceassist:core (pid 8722)
        13,222K: com.miui.analytics (pid 8037)
        7,046K: com.miui.hybrid:entrance (pid 7922)
        5,104K: com.miui.wmsvc (pid 7887)
        4,246K: com.android.smspush (pid 8126)
    213,027K: Perceptible
        89,780K: com.eg.android.AlipayGphone (pid 8238)
        49,033K: com.eg.android.AlipayGphone:push (pid 8204)
        23,181K: com.android.thememanager (pid 11057)
        13,253K: com.xiaomi.joyose (pid 5558)
        10,292K: com.android.updater (pid 3488)
        9,807K: com.lbe.security.miui (pid 23060)
```

```
        9,734K: com.google.android.webview:sandboxed_process0 (pid 11150)
        7,947K: com.xiaomi.location.fused (pid 3524)
    308,049K: Backup
        308,049K: com.tencent.mm (pid 3760 / activities)
    74,250K: A Services
        59,701K: com.tencent.mm:push (pid 7234)
        9,247K: com.android.settings:remote (pid 27053)
        5,302K: com.xiaomi.drivemode (pid 27009)
    199,638K: Home
        188,877K: com.miui.home (pid 2672 / activities)
        10,761K: com.miui.hybrid (pid 7945)
    53,934K: B Services
        35,583K: com.tencent.mobileqq:MSF (pid 14119)
        6,753K: com.qualcomm.qti.autoregistration (pid 8786)
        4,086K: com.qualcomm.qti.callenhancement (pid 26958)
        3,809K: com.qualcomm.qti.StatsPollManager (pid 26993)
        3,703K: com.qualcomm.qti.smcinvokepkgmgr (pid 26976)
    692,588K: Cached
        176,665K: com.plan.kot32.tomatotime (pid 22744 / activities)
        126,918K: com.tencent.mobileqq (pid 23741)
        72,928K: com.tencent.mm:tools (pid 18598)
        68,208K: com.tencent.mm:sandbox (pid 27333)
        55,270K: com.tencent.mm:toolsmp (pid 18842)
        24,477K: com.android.mms (pid 27192)
        23,865K: com.xiaomi.market (pid 27825)
        ...
// 按内存的类别来进行划分
Total PSS by category:
    957,931K: Native
    284,006K: Dalvik
    199,750K: Unknown
    193,236K: .dex mmap
    191,521K: .art mmap
    110,581K: .oat mmap
    101,472K: .so mmap
    94,984K: EGL mtrack
    87,321K: Dalvik Other
    84,924K: Gfx dev
```

```
    77,300K: GL mtrack
    64,963K: .apk mmap
    17,112K: Other mmap
    12,935K: Ashmem
     3,364K: Stack
     2,343K: .ttf mmap
     1,375K: Other dev
     1,071K: .jar mmap
        20K: Cursor
         0K: Other mtrack
// 手机整体内存使用情况
Total RAM: 5,847,124K (status normal)
Free RAM: 3,711,324K (  692,588K cached pss + 2,428,616K cached kernel +   117,492K cached ion +   472,628K free)
Used RAM: 2,864,761K (2,408,529K used pss +   456,232K kernel)
Lost RAM:   184,330K
    ZRAM:   174,628K physical used for   625,388K in swap (2,621,436K total swap)
Tuning: 256 (large 512), oom   322,560K, restore limit   107,520K (high-end-gfx)
```

根据dumpsys meminfo的输出结果，可归结为如下表格：

	划分类型	排序指标	含义
process	PSS	以进程的PSS从大到小依次排序显示，每行显示一个进程，一般用来做初步的竞品分析	
OOM adj	PSS	展示当前系统内部运行的所有Android进程的内存状态和被杀顺序，越靠近下方的进程越容易被杀，排序按照一套复杂的算法，算法涵盖了前后台、服务或节目、可见与否、老化等	

	划分类型	排序指标	含义
category	PSS	以Dalvik/Native/.art mmap/.dex map等划分并按降序列出各类进程的总PSS分布情况	
total	-	总内存、剩余内存、可用内存、其他内存	

此外，为了查看单个App进程的内存信息，我们可以输入如下命令：

```
dumpsys meminfo <pid> // 输出指定pid的某一进程
dumpsys meminfo --package <packagename> // 输出指定包名的进程，可能包含多个进程
```

这里我们输入adb shell dumpsys meminfo 23104这条命令，其中23104为Awesome-WanAndroid App的pid，结果如下所示：

```
quchao@quchaodeMacBook-Pro ~ % adb shell dumpsys meminfo 23104
Applications Memory Usage (in Kilobytes):
Uptime: 258375231 Realtime: 258375231
** MEMINFO in pid 23104 [json.chao.com.wanandroid] **
                   Pss  Private  Private  SwapPss     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap    46674    46620        0      164    80384    60559    19824
  Dalvik Heap     6949     6912       16       23    12064     6032     6032
 Dalvik Other     7672     7672        0        0
        Stack      108      108        0        0
      Ashmem      134      132        0        0
    Gfx dev     16036    16036        0        0
   Other dev        12        0       12        0
    .so mmap     3360      228     1084       27
   .jar mmap        8        8        0        0
   .apk mmap    28279    11328    11584        0
   .ttf mmap      295        0       80        0
```

```
  .dex mmap     7780       20     4908        0
  .oat mmap      660        0       92        0
  .art mmap     8509     8028      104       69
 Other mmap      982        8      848        0
 EGL mtrack    29388    29388        0        0
  GL mtrack    14864    14864        0        0
    Unknown     2532     2500        8       20
      TOTAL   174545   143852    18736      303    92448    66591
25856
App Summary
                   Pss(KB)
                    ------
       Java Heap:    15044
     Native Heap:    46620
            Code:    29332
           Stack:      108
        Graphics:    60288
   Private Other:    11196
          System:    11957
           TOTAL:   174545       TOTAL SWAP PSS:      303
Objects
           Views:      171         ViewRootImpl:        1
     AppContexts:        3           Activities:        1
          Assets:       18        AssetManagers:        6
   Local Binders:       32        Proxy Binders:       27
   Parcel memory:       11         Parcel count:       45
Death Recipients:        1      OpenSSL Sockets:        0
        WebViews:        0
SQL
        MEMORY_USED:      371
 PAGECACHE_OVERFLOW:       72          MALLOC_SIZE:      117
DATABASES
    pgsz     dbsz   Lookaside(b)          cache  Dbname
        4       60            109      151/32/18  /data/user/0/json.
chao.com.wanandroid/databases/bugly_db_
        4       20             19         0/15/1  /data/user/0/json.
chao.com.wanandroid/databases/aws_wan_android.db
```

该命令输出的进程内存概括，我们应该着重关注几个点，下面我将进行一一讲解。

1、查看Native Heap的Heap Alloc与Dalvik Heap的Heap Alloc

我们可以查看Native Heap的Heap Alloc的数值变化，它表示native的内存占用，如果持续上升，则可能有泄漏，而Dalvik Heap的Heap Alloc则表示Java层的内存占用。

2、查看Views、Activities、AppContexts数量变化情况

如果Views与Activities、AppContexts持续上升，则表明有内存泄漏的风险。

3、SQL的MEMORY_USED与PAGECACHE_OVERFLOW

SQL的MEMOERY_USED表示数据库使用的内存，而PAGECACHE_OVERFLOW则表示溢出也使用的缓存，这个数值越小越好。

4、查看DATABASES信息

其中pgsz表示数据库分页大小，这里全是4KB；Lookaside(b)表示使用了多少个Lookaside的slots，可理解为内存占用的大小；而cache一栏中的 151/32/18 则分别表示分页缓存命中次数/未命中次数/分页缓存个数，这里的未命中次数不应该大于命中次数。

3、LeakInspector

LeakInspector是腾讯内部的使用的一站式内存泄漏解决方案，它是Android手机经过长期积累和提炼、集内存泄漏检测、自动修复系统Bug、自动回收已泄露Activity内资源、自动分析GC链、白名单过滤等功能于一体，并深度对接研发流程、自动分析责任人并提缺陷单的全链路体系。

那么LeakInspector与LeakCanary又有什么不同呢?

它们之间主要有四个方面的不同：
一、检测能力与原理方面不同
1、检测能力
它们都支持对Activity、Fragment及其它自定义类的泄漏检测，但是LeakInspector还增加了Btiamp的检测能力：

- 检测有没有在View上decode超过该View尺寸的图片，若有则上报出现问题的Activity及与其对应的View id，并记录它的个数与平均占用内存的大小。
- 检测图片尺寸是否超过所有手机屏幕大小，违规则报警。

这一个部分的实现原理，主要是采用ARTHook来实现，还不清楚的朋友请再仔细看看大图检测的部分。
2、检测原理
两个工具的泄漏检测原理都是在onDestroy时检查弱引用，不同之处在于LeakInspector直接使用WeakReference来检测对象是否已经被释放，而LeakCanary则使用ReferenceQueue，两者效果是一样的。
并且针对Activity，我们通常都会使用Application的registerActivityLifecycleCallbacks来注册Activity的生命周期，以重写onActivityDestroyed方法实现。但是在Android 4.0以下，系统并没有提供这个方法，为了避免手动在每一个Activity的onDestroy中去添加这份代码，我们可以使用发射Instrumentation来截获onDestory，以降低接入成本。代码如下所示：

```
1 Class<?> clazz = Class.forName("android.app.ActivityThread");
2 Method method = clazz.getDeclaredMethod("currentActivityThread", nul
  l);
```

```
method.setAccessible(true);
sCurrentActivityThread = method.invoke(null, null);
Field field = sCurrentActivityThread.getClass().getDeclaredField("mInstumentation");
field.setAccessible(true);
field.set(sCurrentActivityThread, new MonitorInstumentation());
```

二、泄漏现场处理方面不同

1、dump采集

两者都能采集dump，但是LeakInspector提供了回调方法，我们可以增加更多的自定义信息，如运行时Log、trace、dumpsys meminfo等信息，以辅助分析定位问题。

2、白名单定义

这里的白名单是为了处理一些系统引起的泄漏问题，以及一些因为业务逻辑要开后门的情形而设置的。分析时如果碰到白名单上标识的类，则不对这个泄漏做后续的处理。二者的配置差异如下所示：

（1）LeakInspector的白名单以XML配置的形式存放在服务器上。

- 优点：跟产品甚至不同版本的应用绑定，我们可以很方便地修改相应的配置。
- 缺点：白名单里的类不区分系统版本一刀切。

而LeakCanary的白名单是直接写死在其源码的AndroidExcludedRefs类里。

- 优点：定义非常详细，并区分系统版本。
- 缺点：每次修改必定得重新编译。

（2）LeakCanary的系统白名单里定义的类比LeakInspector中定义的多很多，因为它没有自动修复系统泄漏功能。

3、自动修复系统泄漏

针对系统泄漏，LeakInspector通过反射自动修复了目前碰到的一些系统泄漏，只要在onDestory里面调研一个修复系统泄漏的方法即可。而LeakCanary虽然能识别系统泄漏，但是它仅仅对该类问题给出了分析，没有提供实际可用的解决方案。

4、回收资源

如果检测到发生了内存泄漏，LeakInspector会对整个Activity的View进行遍历，把图片资源等一些占内存的数据释放掉，保证此次泄漏只会泄漏一个Activity的空壳，尽量减少对内存的影响。代码大致如下所示：

```
if (View instanceof ImageView) {
    // ImageView ImageButton处理
    recycleImageView(app, (ImageView) view);
} else if (view instanceof TextView) {
    // 释放TextView、Button周边图片资源
    recycleTextView((TextView) view);
} else if (View instanceof ProgressBar) {
    recycleProgressBar((ProgressBar) view);
} else {
    if (view instancof android.widget.ListView) {
```

```
        recycleListView((android.widget.ListView) view);
    } else if (view instanceof android.support.v7.widget.RecyclerVie
w) {
        recycleRecyclerView((android.support.v7.widget.RecyclerView)
view);
    } else if (view instanceof FrameLayout) {
        recycleFrameLayout((FrameLayout) view);
    } else if (view instanceof LinearLayout) {
        recycleLinearLayout((LinearLayout) view);
    }
    if (view instanceof ViewGroup) {
        recycleViewGroup(app, (ViewGroup) view);
    }
}
```

这里以recycleTextView为例，它回收资源的方式如下所示：

```
private static void recycleTextView(TextView tv) {
    Drawable[] ds = tv.getCompoundDrawables();
    for (Drawable d : ds) {
        if (d != null) {
            d.setCallback(null);
        }
    }
    tv.setCompoundDrawables(null, null, null, null);
    // 取消焦点，让Editor$Blink这个Runnable不再被post，解决内存泄漏。
    tv.setCursorVisible(false);
}
```

三、后期处理不同

1、分析与展示

采集dump之后，LeakInspector会上传dump文件，并调用MAT命令行来进行分析，得到这次泄漏的GC链；而LeakCanary则用开源组件HAHA来分析得到一个GC链。但是LeakCanary得到的GC链包含被hold主的类对象，一般都不需要用MAT打开Hporf即可解决问题；而LeakInpsector得到的GC链李只有类名，还需要MAT打开Hprof才能具体去定位问题，不是很方便。

2、后续跟进闭环

LeakInspector在dump分析结束之后，会提交缺陷单，并且把缺陷单分配给对应类的负责人；如果发现重复的问题则更新旧单，同时具备重新打开单等状态转换罗家。而LeakCanary仅会在通知栏提醒用户，需要用户自己记录该问题并做后续处理。

四、配合自动化测试方面不同

LeakInspector跟自动化测试可以无缝结合，当自动化脚本执行中发现内存泄漏，可以由它采集dump并发送到服务进行分析，最后提单，整个流程是不需要人力介入的。而LeakCanary则把分析结果通过通知栏告知用户，需要人工介入才能进入下一个流程。

4、JHat

JHat是Oracle推出的一款Hprof分析软件，它和MAT并称为Java内存静态分析利器。不同于MAT的单人界面式分析，jHat使用多人界面式分析。它被内置在JDK中，在命令行中输入jhat命令可查看没有有相应的命令。

```
quchao@quchaodeMacBook-Pro ~ % jhat
ERROR: No arguments supplied
Usage:  jhat [-stack <bool>] [-refs <bool>] [-port <port>] [-baseline <file>] [-debug <int>] [-version] [-h|-help] <file>
    -J<flag>          Pass <flag> directly to the runtime system. For
            example, -J-mx512m to use a maximum heap size of 512MB
    -stack false:     Turn off tracking object allocation call stack.
    -refs false:      Turn off tracking of references to objects
    -port <port>:     Set the port for the HTTP server.  Defaults to 7000
    -exclude <file>:  Specify a file that lists data members that should
            be excluded from the reachableFrom query.
    -baseline <file>: Specify a baseline object dump.  Objects in
            both heap dumps with the same ID and same class will
            be marked as not being "new".
    -debug <int>:     Set debug level.
                0:  No debug output
                1:  Debug hprof file parsing
                2:  Debug hprof file parsing, no server
    -version          Report version number
    -h|-help          Print this help and exit
    <file>            The file to read
For a dump file that contains multiple heap dumps,
you may specify which dump in the file
by appending "#<number>" to the file name, i.e. "foo.hprof#3".
```

如上，则表明存在jhat命令。它的使用很简单，直在命令行输入jhat xxx.hprof即可：

```
quchao@quchaodeMacBook-Pro ~ % jhat Documents/heapdump/new-33.hprof
```

```
Snapshot read, resolving...
Resolving 408200 objects...
Chasing references, expect 81 dots.....................................
Eliminating duplicate references.....................................
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

jHat的执行过程是解析Hprof文件，然后启动httpsrv服务，默认是在7000端口监听Web客户端链接，维护Hprof解析后数据，以持续供给Web客户端的查询操作。
启动服务器后，我们打开入口地址127.0.0.1:7000即可查看All Classes界面：

jHat还有两个比较重要的功能：

1、统计表

打开127.0.0.1:7000/histo/:

2、OQL查询

OQL是一种模仿SQL语句的查询语句，通常用来查询某个类的实例数量，打开127.0.0.1:7000/oql/并输入java.lang.String查询String实例的数量，如下所示：

JHat比MAT更加灵活，且符合大型团队安装简单、团队协作的需求你，并不适合中小型高效沟通型团队使用。

5、GC Log

GC Log分为Dalvik和ART的GC日志，关于Dalvik的GC日志，在前篇Android性能优化之内存优化已经详细讲解过了，接下来我们说说ART的GC日志。
ART的日志与Dalvik的日志差距非常大，除了格式不同之外，打印的时间也不同，非要在慢GC时才打印除了。下面我们看看这条ART GC Log：

	Explicit	(full)	concurrent mark sweep GC	freed 104710 (7MB) AllocSpace objects,	21 (416KB) LOS objects,	33% free,25MB/38MB	paused 1.230ms total 67.216ms

	Explicit	（full）	concurrent mark sweep GC	freed 104710（7MB） AllocSpace objects,	21（416KB） LOS objects,	33% free,25MB/38MB	paused 1.230ms total 67.216ms
GC产生的原因	GC类型	采集方法	释放的数量和占用的空间	释放的大对象数量和所占用的空间	堆中空闲空间的百分比和（对象的个数）/（堆的总空间）	暂停耗时	

GC产生的原因如下：

- Concurrent、Alloc、Explicit跟Dalvik的基本一样，这里就不重复介绍了。
- NativeAlloc：Native内存分配时，比如为Bitmaps或者RenderScript分配对象， 这会导致Native内存压力，从而触发GC。
- Background:后台GC，触发是为了给后面的内存申请预留更多空间。
- CollectorTransition：由堆转换引起的回收，这是运行时切换GC而引起的。收集器转换包括将所有对象从空闲列表空间复制到碰撞指针空间（反之亦然）。当前，收集器转换仅在以下情况下出现：在内存较小的设备上，App将进程状态从可察觉的暂停状态变更为可察觉的非暂停状态（反之亦然）。
- HomogeneousSpaceCompact：齐性空间压缩是指空闲列表到压缩的空闲列表空间，通常发生在当App已经移动到可察觉的暂停进程状态。这样做的主要原因是减少了内存使用并对堆内存进行碎片整理。
- DisableMovingGc：不是真正的触发GC原因，发生并发堆压缩时，由于使用了
- GetPrimitiveArrayCritical，收集会被阻塞。一般情况下，强烈建议不要使用
- GetPrimitiveArrayCritical，因为它在移动收集器方面具有限制。
- HeapTrim：不是触发GC原因，但是请注意，收集会一直被阻塞，直到堆内存整理完毕。

GC类型如下：

- Full：与Dalvik的FULL GC差不多。
- Partial：跟Dalvik的局部GC差不多，策略时不包含Zygote Heap。
- Sticky：另外一种局部中的局部GC，选择局部的策略是上次垃圾回收后新分配的对象。

GC采集的方法如下：

- mark sweep：先记录全部对象，然后从GC ROOT开始找出间接和直接的对象并标注。利用之前记录的全部对象和标注的对象对比，其余的对象就应该需要垃圾回收了。
- concurrent mark sweep：使用mark sweep采集器的并发GC。
- mark compact：在标记存活对象的时候，所有的存活对象压缩到内存的一端，而另一端可以更加高效地被回收。

- semispace：在做垃圾扫描的时候，把所有引用的对象从一个空间移到另外一个空间，然后直接GC剩余在旧空间中的对象即可。

通过GC日志，我们可以知道GC的量和它对卡顿的影响，也可以初步定位一些如主动调用GC、可分配的内存不足、过多使用Weak Reference等问题。

6、自带防泄漏功能的线程池组件

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。

7、Chrome Devtool

对于HTML5页面而言，抓取JavaScript的内存需要使用Chrome Devtools来进行远程调试。方式有如下两种：

- 直接把URL抓取出来放到Chrome里访问。
- 用Android H5远程调试。

纯H5

1、手机安装Chrome，打开USB调试模式，通过USB连上电脑，在Chrome里打开一个页面，比如百度页面。然后在PC Chrome地址栏里访问Chrome://inspect，如下图所示：

2、最后，直接点击Chrome下面的inspect选项即可弹出开发者工具界面。如下图所示：

默认Hybrid H5调试

Android 4.4及以上系统的原生浏览器就是Chrome浏览器，可以使用Chrome Devtool远程调试WebView，前提是需要在App的代码里把调试开关打开，如下代码所示：

```
if (Build.VERSION_SDK_INT >= Build.VERSION_CODES.KITKAT && 是debug模式) {
    WebView.setWebContentsDebuggingEnabled(ture);
}
```

打开后的调试方法跟纯H5页面调试方法一样，直接在App中打开H5页面，再到PC Chrome的inpsector页面就可以看到调试目标页面。
这里总结一下JS中几种常见的内存问题点：

- closure闭包函数。
- 事件监听。
- 变量作用域使用不当，全局变量的引用导致无法释放。
- DOM节点的泄漏。

若想更深入地学习Chrome开发者工具的使用方法，请查看《Chrome开发者工具中文手册》。

九、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、内类时有危险的编码方式

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关闭时，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的解决方案一般博阿凯如下三个要点：

- 1、把内类声明成static，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“相互怒独立，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。

在AndroidManifest中对当前的Activity设置android:process属性即可，最后再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Handlerh/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，如果包含了以下几点，则不会被轻易杀死。

- 进程包含了服务startService，而服务本身调用了startForeground（需通过反射调用）。
- 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，然后重启它，这样可以有效地降低内存负载。

十、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Memory Profiler、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。

十一、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。
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

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，如下：

- 1、LargeHeap属性
- 2、onTrimMemory
- 3、使用优化过后的集合：如SparseArray
- 4、谨慎使用SharedPreference
- 5、谨慎使用外部库
- 6、业务架构设计合理

在本篇文章，我们除了建立了内存的监控闭环这一核心体系之外，还是实现了以下组件：

- 1、实现了线下的native内存泄漏监控。

- 2、根据设备分级来使用不同的内存和分配回收策略。
- 3、针对低端机做了功能或图片加载格式的降级处理。
- 4、针对缓存滥用的问题实现了统一的缓存管理组件。
- 5、实现了大图监控和重复图片的监控。
- 6、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。
- 7、当UI隐藏时释放内存以增加系统缓存应用进程的能力。
- 8、高效实现了应用全局内的Bitmap监控。
- 9、实现了全局的线程监控
- 10、针对内存使用的重度场景实现了GC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。
从性能分类的纬度来看，除了内存监控方面外，是否也同样建立了卡顿、缓存、电量、异常流量、布局、包体积、IO、存储相关的监控与优化体系。总的来说，要想实现更健全的功能、更深层次的定位问题、线上问题快速准确的发现，还是有很多事情可以做的。

```
1 路漫漫其修远兮，吾将上下而求索。
```

参考链接：

1、国内Top团队大牛带你玩转Android性能分析与优化 第四章 内存优化
2、极客时间之Android开发高手课 内存优化
3、微信 Android 终端内存优化实践
4、GMTC－Android内存泄漏自动化链路分析组件Probe.key
5、Manage your app's memory
6、Overview of memory management
7、Android内存优化杂谈
8、Android性能优化之内存篇
9、管理应用的内存
10、《Android移动性能实战》第二章 内存
11、每天一个linux命令（44）：top命令
12、Android内存分析命令