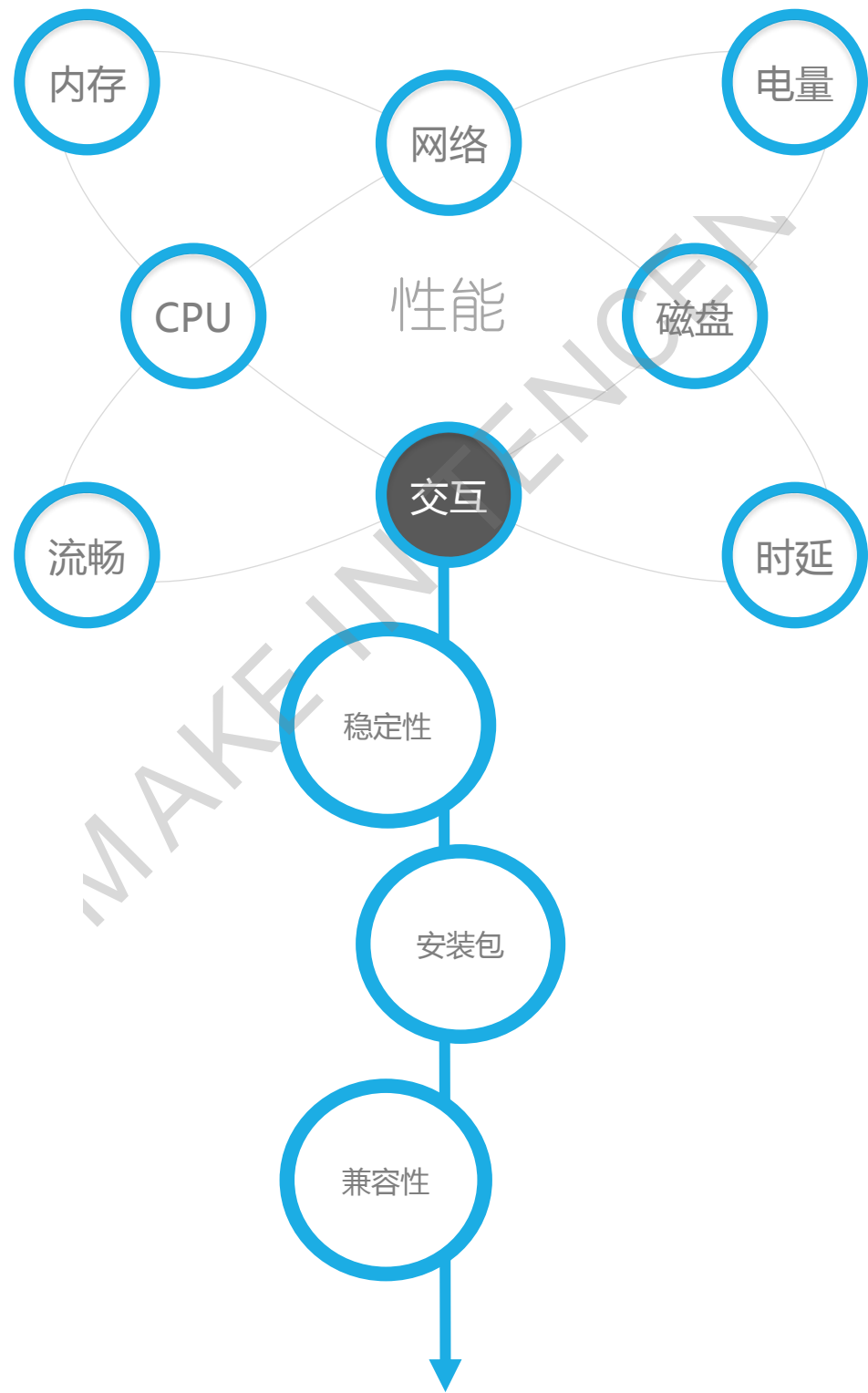


Android 终端专项宝典

(上篇 先行版)



引子

先行版必看说明！

这是我们的灰度版本，大家发现里面的错误或者有建议均可以到这个网址

<https://github.com/mobileperfman/BookResource/issues>，创建你们 issue。给我们宝贵的意见，让我们的正式版本早点出来。

谁适合读此书

- **终端专项测试：**位于此位置的测试，负责产品的性能，安全，稳定性，兼容性等各个方面。我们希望你通过此书，可以有效归纳总结知识，可以拓展思路。也可以作为你在专项测试领域的第一本《字典》。
- **终端系统测试：**专项测试是作为测试本身的一个空间最广阔、蕴含知识最丰富的分支，学习和了解，对职业生涯的发展有着不可或缺的重要作用。
- **高级终端开发：**终端开发必然需要面对许多性能上的难题，本书希望成为你的其中一部指南。还有，必须要说越是高级的终端开发，越是需要啃硬骨头，而专项恰巧就是个硬骨头。

致谢

老黑，小 V，云雷，谭力，杨洋，coco，gavin，专家，Emily，志斌，朱乔的鼎力支持。

目录

引子.....	2
先行版必看说明！	2
谁适合读此书.....	2
致谢.....	2
欢迎你离开桌面的世界.....	4
故事的起点.....	4
如何利用我.....	4
交互类性能.....	5
原理篇.....	5
流畅度.....	6
案例：红米手机上的手 Q 消息列表卡顿问题.....	6

案例：硬件加速中文字体渲染的坑.....	9
案例：圆角的前世今生.....	16
响应时延.....	22
案例：Android 应用发生黑屏的场景分析	22
案例：“首次打开聊天窗口”之痛	24
工具集.....	27
Perfbox 自研工具：Scrolltest.....	27
Systrace（分析）	30
traceview（分析）	38
gfxinfo（分析）	40
Intel 的性能测试工具：UXTune（测评+分析）	41
Hierarchy Viewer（分析）	41
Tracer for OpenGL ES	42
Slickr（测评+分析）	43
资源类性能.....	47
内存篇.....	47
工具集.....	47
案例：内类是有危险的编码方式.....	67
案例：使用统一界面绘制服务的内存问题.....	70
案例：结构化消息点击通知产生的内存问题.....	72
案例：为了不卡，所以可能泄漏.....	73
案例：图片缓存的内存问题.....	76
案例：登录界面有内存问题么.....	79
案例：使用 wifimanager 的内存问题	81
案例：产品图片缓存服务合格么.....	84

案例：关于图片解码配色设置的建议.....	88
案例：图片放错资源目录也会有内存问题.....	92
案例：要使用统一的“图片停车库”	97
案例：把 Webview 类型泄漏装进垃圾桶进程	98
案例：定时器、延时器的内存问题.....	101
案例：列表控件 ListView 的 view 泄漏	106
案例：处理“一车多位”	111
案例：大家伙要怎么进入小车库.....	117
案例：大车小车分开停.....	122
案例：我是这样想的，Android 要纠正内存世界观了	122

欢迎你离开桌面的世界

故事的起点

我们为什么会有这个宝典呢？

记得我从微博和 MAC QQ 项目中解放出来，就开始接手手 Q，组建专项测试团队。那时有几个小伙伴，我们一起做手 Q 的专项测试，发现推动专项问题解决都非常困难。产品的需求压力巨大，性能越来越差，我们开始用更严厉的标准像守护者一样守护手机 QQ，例如：安装包的大小，接手后的第一个版本，涨了 10M。我们看到了风险，顶着各个部门的 KPI 需求，定制了一系列严厉的指标，超过则需求都不让合进来，从此安装包也刹住了车。但 KPI 这东西真的很厉害，像是洪水，不泄掉，堤坝只能越建越高，我们的压力也越来越大。产品经理开始不禁地问，为什么安装包不能变大呢？为什么不能占用更多的内存？我提供更多服务，为什么不能消耗更多流量？为什么？

在这些质疑中，我们经历了许多，里面除了工具，流程之外，更多是真实的经验。例如，安装包不能再变大了。这里需要证据，运营同学找到了应用宝的数据，发现有不少用户是 3G 下载，还有对下载失败率的影响。数据跟老板汇报过后，我们拍定了更严厉的标准：0 增长。类似的这些故事，慢慢随着我们团队的人增加，也越来越多，跟大家知道的一样，有跟开发的 PK，有不服输，自己去解决专项 BUG，有跟产品经理 PK 需求与专项性能的平衡。但是知道故事的人并不多，知道“为什么”的人就更少了。我们觉得这些故事应该要记录和被分享下来，然后有了我们的专项宝典。宝典里面会介绍工具，原理，但更重要的是一个个真实的案例，BUG 解决方案。

如何利用我

每一个篇章会分为原理，案例，工具三个篇章。原理主要是为了说明一些不脱离实际的实用的基础知识。而案例会从按照分析专项问题的思路来划分我们的案例，力求做到让看书的小伙伴可以举一反三。而工具则是对前面案例中使用的工具的更全面的补充说明。

交互类性能

响应时延+流畅度，我把它称作交互类性能的基础。那什么时候算是交互性能问题呢？自己感觉到卡顿那就是有问题的。但是这个比较主观，每个人的感觉都不一样。如果你是公司老大，尽管任性，程序员和产品经理会尊重你的“感觉”的。但是如果你是一线测试，一线开发，那就要有理性的武器。下面是 Intel 网站上说的行业经验，也就是理性的武器。怎么理解这个武器呢？500ms 是用户进行操作到界面产生响应开始的相隔的时间，类似打开聊天窗口，用三星 i9100 为基准，能做到 500ms。但用户不会快乐，但是至少不会骂，这就叫做 Acceptable。另外一个指标是流畅度，可以看到图形动画和视频播放的 Acceptable 分别是 30fps 和 20fps。

那 Good 和 Best 怎么理解呢？这个事情有点像用习惯了 iPhone 6，突然让你用 Nexus S，让用户一直沉浸在 Good 和 Best 之间，Acceptable 自然就会变成 Bad，所以 100ms 和 200ms 还可以用来做竞品对比，如果竞品界面（同产品的其他界面），竞品产品都做到了 100ms 和 200ms 这个阶段，那么就要提升自我的修养了。下面我们会从流畅度和响应时延两个方面来介绍我们如何测评和分析应用，让应用达成 Acceptable，追求 good 和 best 的故事。

	Best	Good	Acceptable
Response delay	$\leq 100\text{ms}$	$\leq 200\text{ms}$	$\leq 500\text{ms}$
Graphics animation	$\geq 120\text{fps}$	$\geq 60\text{fps}$	$\geq 30\text{fps}$
Video playback	$\geq 60\text{fps}$	$\geq 30\text{fps}$	$\geq 20\text{fps}$

来自行业经验：<https://software.intel.com/zh-cn/android/articles/quantify-and-optimize-the-user-interactions-with-android-devices>

原理篇

之前有很多很多的文章，从各个维度去描述流畅度的影响因素，例如从 MVC 的角度，从资源类性能指标角度等等。但是大家可能会发现，这些纬度在解决 BUG 的时候，并不是最佳的。原因很简单，因为流畅度，响应时延，电量都是各种资源使用的最终结果，这里我们需要有可以拆解，找出最影响的部份的方法。

根据图 1.1 中，我们来了解下从我们操作到产生界面变化的整个流畅是怎样的。下图 1.1 中，我们从点击或者 Drag-Drop 事件开始，如果是 ListView 之类的就会触发 Adapter.getView()的操作，再之后，系统会根据情况来执行 Measure 测量和 Layout 定位的操作，最后就是绘制，绘制分为硬件加

速和软件加速两种，软件加速相对来说简单很多，在 Draw 的时候就是触发直接交给 skia 栅格化来完成，硬件加速相对比较常见和复杂，下图 1.1 描述的 Draw 的部分就是指硬件加速

更新：在 android 5.0 以上，情况稍微发生了一些改变，引入了 RenderThread。



图 1.1 绘制流程图

如图 1.2 所示，CPU 与 GPU 在于 getDisplayList(UpdateDisplayList)与 DrawDisplayList 之间，因为 getDisplayList 其实是生成一系列 OPENGGL 的指令，而 draw displaylist，就是通过 GPU 真正去根据这些指令去绘制。这里 systrace 和 traceview 可以快速帮助定位问题，根据问题的位置，可以再使用不同的工具进行分析，例如 dumpsys gfxinfo 与 opengl tracer 是专门针对 draw 的部分，而 HierachyView 对于 measurement,layout 就有到每个 view 的对应耗时的详尽信息。下面我们将详细地介绍这些工具。

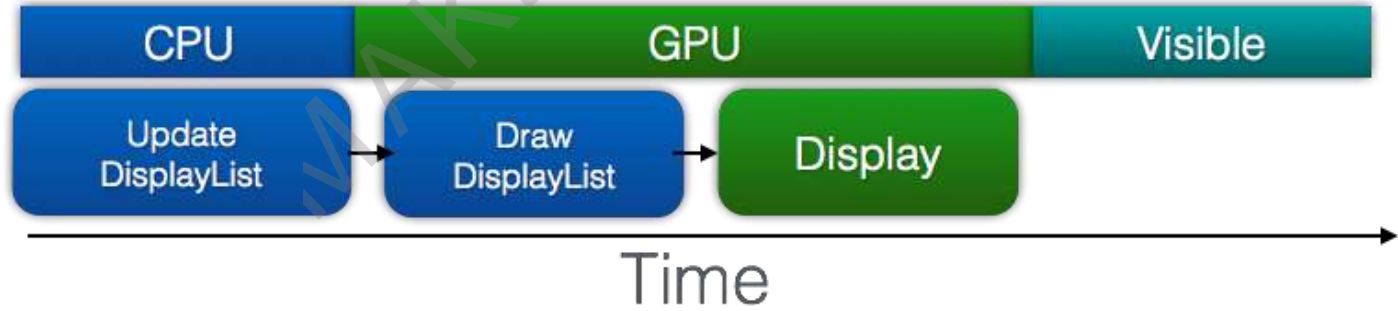


图 1.2 CPU&GPU

流畅度

案例：红米手机上的手 Q 消息列表卡顿问题

问题类型: Measurement

View.measure: 29.5%

ViewGroup.layout: 2.8%

ViewRootImpl.draw: 24.7%

HeaderListAdapter.getView: 6.5%

我们发现，好友列表在我们的常规测试机型 i9100 上一点不卡，但是在红米居然卡爆了。可惜这个只是人的感觉而已。有没有量化的方法呢？当然有，我们开始使用 `perfbox:scrolltest` 来测算这里的 FPS 值，发现，在相同的滑动频率下，红米仅仅只有 13FPS，而 i9100 却有 40 多 FPS。

如果要初步定位是哪里的问题的话，我们有两个选择，一个是 `systrace`，另外一个则是 `traceview`。这里我们选择了使用更加方便 `traceview`。如图 1.2，按照 Incl CPU Time(包含子函数的耗时)排序，只发现两个关键函数耗时特别长，一个是 `draw 34.4% CPU TIME`，一个是 `measure 32.2% CPU TIME`。

图 1.2 traceview 分析图

Name	Incl Cpu Time %	Incl Cpu Time
21 android/view/HardwareRenderer\$GIRenderer.draw (Landroid/view/View;Landroid	34.4%	1363.959
Parents		
22 android/view/ViewRootImpl.draw (Z)V	100.0%	1363.959
Children		
self	0.1%	1.121
29 android/view/HardwareRenderer\$GIRenderer.buildDisplayList (Landroid/view	68.5%	934.932
55 android/view/HardwareRenderer\$GIRenderer.drawDisplayList (Landroid/view	22.6%	308.492
138 android/view/HardwareRenderer\$GIRenderer.swapBuffers (I)V	5.3%	71.741
189 android/view/HardwareRenderer\$GIRenderer.beginFrame (Landroid/view/I	2.9%	39.016
751 android/view/HardwareRenderer\$GIRenderer.prepareFrame (Landroid/gra	0.2%	2.250
797 android/view/HardwareRenderer.getSystemTime ()J	0.2%	2.174
862 android/view/HardwareRenderer\$GI20Renderer.onPostDraw ()V	0.1%	1.611
864 android/view/ViewRootImpl.onHardwarePostDraw (Landroid/view/Hardwa	0.1%	1.607
740 java/util/concurrent/locks/ReentrantLock.lock ()V	0.1%	1.015
(context switch)	0.0%	0.000

图 1.3 draw 耗时分析

如图 1.3 所示，`buildDisplayList` 的耗时的主要构成，不过这是应该的，因为 `drawDisplayList` 就是 GPU 操作了，原理上 CPU 耗时就不应该高，再要深入分析，就要使用 `opengl tracer` 了，所以先放下。看另外一个耗时很长的 `measure`，如图 1.4，我们可以看到 `TextView` 的 `onMeasure` 特别高。这里很想说，根据经验 `TextView.onMeasure CPU Time` 高达 1008，这肯定是不合理的。

Name	Incl Cpu Time %	Incl Cpu Time
24 android/view/View.measure (II)V	32.2%	1275.289
▼ Parents		
27 com.tencent/widget/ListView.setupChild (Landroid/view/View;IIZIZI)V	85.8%	1093.818
69 com.tencent/widget/PinnedHeaderExpandableListView.configHeaderView (II)V	14.2%	181.471
▼ Children		
self	0.1%	1.100
25 android/widget/RelativeLayout.onMeasure (II)V	99.2%	1265.295
535 android/view/ViewGroup.resolveRtlPropertiesIfNeeded ()Z	0.3%	3.285
304 android/util/LongSparseLongArray.put (JJ)V	0.2%	2.306
176 java/lang/System.currentTimeMillis ()J	0.2%	2.204
178 android/view/View.isLayoutModeOptical (Ljava/lang/Object;)Z (context switch)	0.1%	1.099
0.0%	0.000	
► Parents while recursive		
▼ Children while recursive		
28 android/widget/TextView.onMeasure (II)V	79.1%	1008.326
171 android/widget/LinearLayout.onMeasure (II)V	3.6%	45.477

图 1.4 measure 耗时分析

再深挖下去发现，如图 1.5，StringBuilder 相关的系列操作，居然消耗了 onMeasure 将近一半的 cpu time。到这里，其实已经是系统的操作了，一般来说会认为不应该怀疑这里了。但是我们觉得 StringBuilder 实在不应该消耗这么多，况且现象就是告诉我们，这跟系统密切相关。所以为了进一步证明，我们还测试并录制了 i9100 的 traceview（图 1.6），发现果然！StringBuilder 什么的消失无踪。最终开发打算重写一个 textview 试一下，果然！流畅度一下子就上去了。

Name	Incl Cpu Time %	Incl Cpu Time
28 android/widget/TextView.onMeasure (II)V	25.5%	1008.326
▼ Parents		
24 android/view/View.measure (II)V	100.0%	1008.326
▼ Children		
self	1.2%	12.508
65 android/widget/TextView.makeNewLayout (II)Landroid/text/BoringLayout\$Metrics;	22.5%	227.045
47 java/lang/StringBuilder.append (Ljava/lang/Object;)Ljava/lang/StringBuilder;	22.3%	224.567
61 java/lang/StringBuilder.append (I)Ljava/lang/StringBuilder;	13.9%	140.236
60 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	9.8%	98.955
105 java/lang/StringBuilder.<init> ()V	6.5%	65.455
149 android/text/Layout.getDesiredWidth (Ljava/lang/CharSequence;Landroid/text/TextPaint;)I	6.5%	65.253
153 android/widget/TextView.getDesiredHeight ()I	6.0%	60.501
120 java/lang/StringBuilder.toString ()Ljava/lang/String;	3.7%	37.102

图 1.5 TextView measure 分析

Name	Incl Cpu Time %	Incl Cpu Time
62 com.tencent/widget/AbsListView.obtainView (I[Z)Lan	7.7%	186.828
63 android/widget/TextView.onMeasure (I)V	7.6%	186.028
▼ Parents		
19 android/view/View.measure (I)V	100.0%	186.028
▼ Children		
self	10.4%	19.408
87 android/widget/TextView.makeNewLayout (II)Lar	53.1%	98.773
141 android/text/Layout.getDesiredWidth (Ljava/lang	17.5%	32.529
175 android/text/BoringLayout.isBoring (Ljava/lang	4.3%	7.924
407 android/widget/TextView.getDesiredHeight ()I	3.5%	6.487
525 android/widget/TextView.registerForPreDraw ()V	2.2%	4.176
604 android/widget/TextView.getCompoundPaddin	1.2%	2.259
617 android/widget/TextView.getCompoundPaddin	1.2%	2.228
467 android/view/View\$MeasureSpec.getMode (I)I	0.9%	1.661
533 android/view/View\$MeasureSpec.getSize (I)I	0.9%	1.657

图 1.6 i9100 TextView measure 分析

同样是 TextView，但是下面这个缺陷的成因就完全不一样。

案例：硬件加速中文字体渲染的坑

问题类型：Draw

ViewRootImpl.draw: 92.8%

View.measure: N/A

ViewGroup.layout: N/A

getView: N/A

是否在进行界面滑动优化时，你曾经抓破脑袋也无法达到预期？在测试孩童、PM、和各位老大那催命符般的催促下祭出杀手锏：开硬件加速。通常情况下开启硬件加速能让你的界面滑动性能瞬间健步如飞。但当你的 UI 里有大量 TextView，或者 TextView 是个长文本时，而且内容还是人见人怕的中文时。开启硬件加速后，或许你会有一种想换一个键盘的冲动。下面和大家分享一个手 Q 在开发长文本阅读时遇到的一个硬件加速中文字体渲染的坑。

需求背景：

在手 Q 聊天界面，双击气泡，可以将气泡内容在一个新的界面中打开（在新界面中文本内容具有额外的阅读编辑功能），该需求即为长文本阅读。当消息内容较长时，TextView 可以上下滑动。

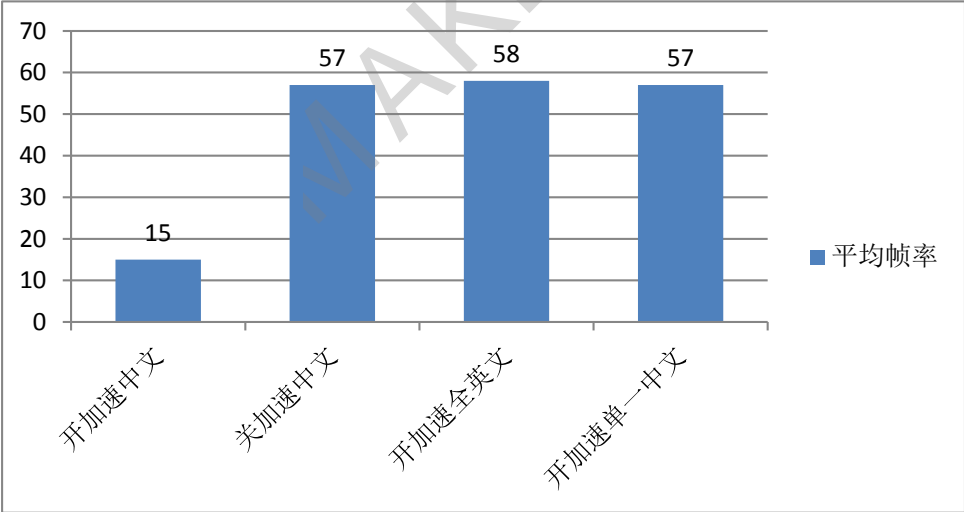


图 1 左

图 1 右

开发哥哥在做需求时，考虑到长文本界面用户总是喜欢上下滑动的，跟据经验开启了硬件加速，天真地以为可以提升滑动性能（这样又可以少一个 Bug 单了）。结果提测后，他掉坑里了：在我们测试孩童丢了一段《西游记》上去后，滑起来，简直卡爆了。

测试数据：



注：机型：三星 9100，Android 4.1.2

问题分析

滑动不流畅通常是由于帧渲染耗时较长，我们可以通过 DDMS 的 Method Profing 抓取 Trace 分析在滑动过程中哪个方法的执行耗时较长。

步骤：

- 1、 连接手机，打开 DDMS，找到要分析的应用，点击下图中的“小耙子”



- 2、 准备好测试场景，点击下图中的“带红点小耙子”开始抓 Trace



- 3、 做相应的滑动操作
- 4、 点击下图中的“带黑点小耙子”结束抓 Trace



- 5、 分析 Trace 文件

如下图，开启硬件加速时上下滑动长文本界面，抓取的 Trace，从图中我们看到 HardwareRenderer&GIRenderer.draw 方法平均耗时 49ms（平均每帧绘制耗时大于 49ms），占据了 92%的 CPU 时间。且 HardwareRenderer&GIRenderer.draw 的耗时主要产生在他调用的 GLES20Canva.drawDisplyList 上。GLES20Canva.drawDisplyList 正是使用 GPU 绘制显示列表执行绘图操作。从而定位到滑动卡顿的原因是因为开启了硬件加速。

Name	Incl C...	Incl Cpu...	Exc...	Excl...	Incl R...	Incl Re...	Excl...	Excl...	Calle...	Cpu...	Real T...
11 android/view/ViewRootImpl.draw (Z/V	92.8%	2307.524	0.1%	3.691	17.1%	3118.439	0.0%	3.655	63+0	36.627	49.499
12 android/view/HardwareRenderer\$GIRenderer.draw (Lar	92.4%	2299.955	0.3%	7.615	17.1%	3110.876	0.0%	7.692	63+0	36.507	49.379
13 android/view/GLES20Canvas.drawDisplayList (Landroid/v	76.5%	1902.706	0.1%	2.749	14.6%	2650.721	0.0%	3.379	293+0	6.494	9.047
14 android/view/GLES20Canvas.drawDisplayList (IILandro	76.3%	1899.070	76.3...	1898...	14.6%	2646.640	10.4%	189...	293+0	6.481	9.033
Parents											
13 android/view/GLES20Canvas.drawDisplayList (Lar	100.0%	1899.070			100.0%	2646.640			293/...		
Children											
self	100.0%	1898.610			71.7%	1898.527					
228 android/graphics/Rect.set (IIII)V	0.0%	0.460			0.0%	0.305			62/297		
(context_switch)	0.0%	0.000			28.3%	747.808			52/272		
15 android/view/View.getDisplayList (Landroid/view/Displ	9.7%	241.726	0.2%	4.036	1.4%	258.912	0.0%	3.351	63+5...	0.428	0.458
16 android/view/View.getDisplayList (Landroid/view/Displa	9.7%	241.146	0.5%	12.184	1.4%	258.363	0.1%	13.0...	63+5...	0.427	0.457

优化方法

关闭长文本 TextView 的硬件加速。

```
protected void onCreate(Bundle savedInstanceState) {

    .....

    super.onCreate(savedInstanceState);

    .....

    textView = (SelectableTextView) findViewById(R.id.content);
    textView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);

    scrollView = (ScrollView) findViewById(R.id.sv);
    textView.setLayerType(View.LAYER_TYPE_SOFTWARE, null);

    textView.setGravity(Gravity.CENTER);

    textView.outScrollView = scrollView;

    .....

}
```

建议:

当你的 view 需要处理长中文时，请禁用 view 的硬件加速

源码分析:

由于 Method Profiling 抓到的 Trace 只能抓到 Java 层的方法执行耗时，无法从 Trace 继续分析 GLES20Canvas.drawDisplyList 内部的执行耗时。不过我们可以去看看 Android 硬件加速相关的源码吧。由于完整的调用链条较长，这里不一一列出源码，只给出调用关键调用链条及关键代码。

从前面的 Trace 文件分析的调用链:

```
ViewRootImpl.draw()
->HardwareRenderer$GIRenderer.draw()
->GLES20Canvas.drawDisplayList()
->GLES20Canvas.nDrawDisplayList()。
```

这是 Java 层 Trace 可以追踪到的调用链，从 GLES20Canvas.nDrawDisplayList()开始通过 JNI 跳转到 Native 层。

GLS20CanvasnDrawDisplayList()

JNI -> android_view_GLES20Canvas_drawDisplayList()

->OpenGLRenderrer.drawDisplayList()

OpenGLRenderrer.drawDisplayList()方法通过调用 DisplayList 的 replay 方法以回放前面录制的 displaylist 执行绘制操作。

```
status_t OpenGLRenderrer::drawDisplayList(DisplayList* displayList,
    Rect& dirty, int32_t flags, uint32_t level) {
    // All the usual checks and setup operations (quickReject, setupDraw, etc.)
    // will be performed by the display list itself
    if (displayList && displayList->isRenderable()) {
        return displayList->replay(*this, dirty, flags, level);
    }
    return DrawGInfo::kStatusDone;
}
```

DisplayList 的 replay 方法遍历 DisplayList 中保存的每一个操作执行。渲染字体的操作名是 DrawText。当遍历到一个 DrawText 操作时，调用 OpenGLRenderrer::drawText 方法渲染字体。

OpenGLRenderrer::drawText()

```
status_t DisplayList::replay(OpenGLRenderrer& renderer, Rect& dirty, int32_t flags,
    uint32_t level) {
    status_t drawGlStatus = DrawGInfo::kStatusDone;
    .....
    while (!mReader.eof()) {
        int op = mReader.readInt();
        .....
        switch (op) {
            case DrawGLFunction: {
                .....
            }
            break;
            case Save: {
                ....
            }
            .....
            case DrawText: {
                getText(&text);
                .....
                drawGlStatus |= renderer.drawText(text.text(), text.length(), count,
                    x, y, paint, length);
            }
        }
    }
}
```



```

    }
    break;
    ....

    return drawGfStatus;
}

```

->FontRenderer::renderText()

->Font::render()

最终进入到一个 Font::render()方法去渲染字体，在这个方法有一个很关键的动作，获取字体缓存。看到这里，基本可以确定开启硬件加速在处理长中文文本时的卡顿原因。由于每个中文的编码是不同的，因此中文的缓存效果非常不理想。而对于英文，只需要缓存 26 个字母就可以了。

```

void Font::render(SkPaint* paint, const char *text, uint32_t start, uint32_t len,
    int numGlyphs, SkPath* path, float hOffset, float vOffset) {
    .....
    while (glyphsCount < numGlyphs && penX < pathLength) {
        glyph_t glyph = GET_GLYPH(text);
        if (IS_END_OF_STRING(glyph)) {
            break;
        }
        CachedGlyphInfo* cachedGlyph = getCachedGlyph(paint, glyph);
        penX += SkFixedToFloat(AUTO_KERN(prevRsbDelta,
            cachedGlyph->mLsbDelta));
        prevRsbDelta = cachedGlyph->mRsbDelta;
        if (cachedGlyph->mIsValid) {
            drawCachedGlyph(cachedGlyph, penX, hOffset, vOffset,
                measure, &position, &tangent);
        }
        penX += SkFixedToFloat(cachedGlyph->mAdvanceX);
        glyphsCount++;
    }
}

```

实验室：硬件渲染的其他坑

1. 在软件渲染的情况下，如果需要重绘某个 Parent View 中所有的子 View，只需要调用这个 Parent View 的 invalidate()方法即可，但如果开启了硬件加速，这么做是行不通的，需要遍历整个子 View 并调用 invalidate()。
2. 在软件渲染的情况下，会常常使用到 Bitmap 重用的情况来节省内存，如下面这段代码。但是如果开启了硬件加速，这将会不起作用

```

public void onDraw(Canvas canvas) {

```

```

// 擦掉所有像素
sBitmap.eraseColor(Color.TRANSPARENT);
Canvas buffer = new Canvas(sBitmap);
buffer.drawRect(mRect, mPaint);
canvas.drawBitmap(sBitmap, 0, 0, null);

sBitmap.eraseColor(Color.TRANSPARENT);
buffer.drawOval(mRectF, mPaint);
canvas.drawBitmap(sBitmap, 0, 0, null);
}

```

3. 当开启硬件加速的 UI 在前台运行时，需要耗费额外的内存。当硬件加速的 UI 切换到后台时，上述额外内存有可能不释放（多存在于 4.1.2）

可以在 `onStop` 时从 `ViewRoot` 中移除掉，在 `onResume` 中重新加载回来，但是这样会容易引入其他问题，建议慎重修改。

长或宽大于 2048px 的 Bitmap 无法绘制，显示为一片透明。原因是 opengl 的材质大小上限为 2048x2048，因此对于超过 2048 像素的 Bitmap，需要将其切割成 2048x2048 以内的图片块，然后在显示的时候拼起来

4. 有可能会花屏（主要集中在 4.0.x 版本）。当 ui 中存在 `overdraw`（过渡绘制）时会比较容易发生，过渡绘制可以通过手机开发者选项中“调试过渡绘制”开关来查看，一般来说绘制少于 5 层不会出现花屏现象，如果有大块红色区域就要小心了，这时候就需要优化你的 UI 结构。另外还有一种方法就是在较高层的 `ViewGroup` 上设置 `LayerType` 为 `Software` 来解决，原理是当你设置 `layerType` 为 `Software` 时，这个 `view` 会将自己先绘制到一个 `Bitmap` 上，最后再把这个 `Bitmap` 绘制到 `Canvas` 上，从而变相的减少了绘制的层数，原理跟开启 `drawingCache` 差不多，代码如下：

Java 代码

```

if (VersionUtils.isHoneycomb())
{
    gallery.setLayerType(View.LAYER_TYPE_SOFTWARE, null);
}

```

5. 在 4.0.x 版本，如果渲染含有大量中文字符文本块，会有明显的掉帧。在 4.1.2 版本修复。原因是因为底层渲染时对文本的 `buffer` 设置过小导致的（因为英文就 26 个），一般也不用刻意去管，如果实在不爽可以把 4.0.x 的硬件加速关了。代码如下：

Xml 代码:

```

<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"

```

问题类型: Invalidate(AdapterView)

```
android:theme="@style/AppTheme"
android:hardwareAccelerated="@bool/hardware_acceleration">
<activity
    android:name=".MyActivity"
    android:label="@string/app_name"
    >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</application>
```

最后在 value-v14, value-15 中设置相应的 bool 值即可:

Xml 代码

```
<bool name="hardware_acceleration">false</bool>
```

6. 补充个误区, 关于 LAYER_TYPE_SOFTWARE, 虽然无论在 app 打开硬件加速和没有开硬件加速的时候, 都会通过软件绘制 BITMAP 作为离屏缓存, 但区别在于打开硬件加速的时候, BITMAP 最终还是会通过硬件加速方式 drawDisplayList 渲染这个 bitmap。

案例: 圆角的前世今生



《乔布斯传》中的描述, 乔布斯为了确定电脑机箱的弧度, 他和团队成员讨论了将近一个月。一个菜单栏, 工程师们反复做了 20 多次才满足他的要求。他希望所有窗口都是圆角矩形, 但就当时的技术而言, 难度太大。于是乔布斯带着工程师走了三条街, 找出 17 处圆角矩形的例子。工程师终于被说服, 最终解决了这一难题。这是我能知道的圆角在软件设计上的唯一来历。而从 ios5-ios6 的像素级变化中, 也不难看出圆角无处不在的呀~~~~~

圆角图片与流畅度的关系

大家可能会问为什么会放到流畅度章节呢？很简单，因为圆角图片置于性能中，用户能体验到的其中一个重要的影响就是**流畅度**。所以流畅度将会是我们衡量圆角图片的最终落脚点。而且刚好属于 Invalidate 这个步骤，通常是在 getView 的时候把 Bitmap 放到 ImageView，因此解码图片，图片缓存，会是最影响这个步骤的因素。但是开始之前我们还要看一下制作圆角图片的四种方法和基本的区别（如表 1.0 所示）。包括 Bitmapshader，这也是官方 Romain Guy 大神文章推荐的方式。Let`s 开始测试。（测试过程中如果有遗漏，或者思考不周的，请回复，小 V 会尽快更正。）

方法	支持 antialiasing(无锯齿)	支持使用 RGB_565(省内存)	支持硬件加速	canvas 上只绘制一次
BitmapShader	√	√	√	√
AvoidXfermode	√		√	
clipPath				
9patch/xml drawable	√	√	√	

表 1.0 （第四种方法，其实是有点像相框的原理，只适用于纯色背景的图片，适用范围，可以是按钮等等，比较简单，范围也比较小，下面就不做过多介绍了）

利用 CLIPPATH 绘制的圆角

代码非常简单，关键是利用 clippath 在画布 canvas 上面裁切出圆角的画布，然后再把 bitmap 画上去。有个地方值得关注的，其实最终生成的还是一张 ARGB_8888 的图片。

```
public static Bitmap getRoundedShape(Bitmap scaleBitmapImage, float radius, int margin) {
    int targetWidth = scaleBitmapImage.getWidth() - margin;

    int targetHeight = scaleBitmapImage.getHeight() - margin;
    Bitmap targetBitmap = Bitmap.createBitmap(targetWidth,
        targetHeight, Bitmap.Config.ARGB_8888);

    Canvas canvas = new Canvas(targetBitmap);
    Path path = new Path();

    path.addRoundRect(new RectF(margin, margin, targetWidth, targetHeight), radius, radius, Path.Direction.CW);
    canvas.clipPath(path);
    Bitmap sourceBitmap = scaleBitmapImage;
    canvas.drawBitmap(sourceBitmap,
        new Rect(0, 0, sourceBitmap.getWidth(),
            sourceBitmap.getHeight()),
        new Rect(0, 0, targetWidth, targetHeight), null);
    return targetBitmap;
}
```

但是无论怎么样，因为 antialiasing 支持无效，所以这个还是锯齿太明显了。正如下面两张图，用 clipPath 的如左图，与右图对比就会明显看到有锯齿，非常不美观。



利用 AVOIDXFERMODE 的圆角

锯齿这么丑，你能接受，腾讯可爱的产品经理也肯定是不能接受的了。而在 google 或者百度上面最流行的一种圆角的方法，也是手 Q 与手空使用的方法。就是用 `avoidxfermode` 来做圆角。而且这个模式支持硬件加速。代码实现的过程，首先创建一个指定高宽的 `bitmap`，作为输出的内容，然后创建一个相同大小的矩形，利用画布绘制时指定圆角角度，这样画布上就有了一个圆角矩形，最后就是设置画笔的剪裁方式为 `Mode.SRC_IN`，将原图叠加到画布上，

```
int srcX = 0;
int srcY = 0;
int bitmapW = bitmap.getWidth();
int bitmapH = bitmap.getHeight();
if(width > height){
    width = height;
    srcX = (bitmapW-bitmapH)/2;
    bitmapW = bitmapH;
}else if(height > width){
    height = width;
    srcY = (bitmapH-bitmapW)/2;
    bitmapH = bitmapW;
}

Bitmap output = Bitmap.createBitmap(width, height, Config.ARGB_8888);
Canvas canvas = new Canvas(output);

final int color = 0xff424242;
final Paint paint = new Paint();
final Rect srcRect = new Rect(srcX, srcY, bitmapW, bitmapH);
final Rect destRect = new Rect(0, 0, width, height);
```



```

final RectF rectF = new RectF(destRect);

paint.setAntiAlias(true);
paint.setDither(true);
paint.setFilterBitmap(true);

canvas.drawARGB(0, 0, 0, 0);
paint.setColor(color);
canvas.drawRoundRect(rectF, roundPx, roundPx, paint);
paint.setXfermode(new PorterDuffXfermode(Mode.SRC_IN));
canvas.drawBitmap(bitmap, srcRect, destRect, paint);

return output;

```

这个方法的好处也明显，支持 `antialias(paint.setAntiAlias(true);)`，无锯齿是必须的，也支持硬件加速。但我们的脚本是否就此为止呢？



使用 BITMAPSHADER 绘制圆角

这时出现了官方介绍的方法，也就是 `bitmapshader`。这个方法代码原理也很简单，特点就是不需要额外创建一个图片，这里把原图构造成了一个 `BitmapShader`，然后就可以用画布直接画出圆角的内容。

```

public class StreamDrawable extends Drawable {
    private static final boolean USE_VIGNETTE = false;

    private final float mCornerRadius;
    private final RectF mRect = new RectF();
    private final BitmapShader mBitmapShader;
    private final Paint mPaint;
    private final int mMargin;

    public StreamDrawable(Bitmap bitmap, float cornerRadius, int margin) {
        mCornerRadius = cornerRadius;
    }

```

```

        mBitmapShader = new BitmapShader(bitmap,
            Shader.TileMode.CLAMP, Shader.TileMode.CLAMP);

        mPaint = new Paint();
        mPaint.setAntiAlias(true);
        mPaint.setShader(mBitmapShader);

        mMargin = margin;
    }

    @Override
    protected void onBoundsChange(Rect bounds) {
        super.onBoundsChange(bounds);
        mRect.set(mMargin, mMargin, bounds.width() - mMargin, bounds.height() - mMargin);

        if (USE_VIGNETTE) {
            RadialGradient vignette = new RadialGradient(
                mRect.centerX(), mRect.centerY() * 1.0f / 0.7f, mRect.centerX() * 1.3f,
                new int[] { 0, 0, 0x7f000000 }, new float[] { 0.0f, 0.7f, 1.0f },
                Shader.TileMode.CLAMP);

            Matrix oval = new Matrix();
            oval.setScale(1.0f, 0.7f);
            vignette.setLocalMatrix(oval);

            mPaint.setShader(
                new ComposeShader(mBitmapShader, vignette, PorterDuff.Mode.SRC_OVER));
        }
    }

    @Override
    public void draw(Canvas canvas) {
        canvas.drawRoundRect(mRect, mCornerRadius, mCornerRadius, mPaint);
    }

    @Override
    public int getOpacity() {
        return PixelFormat.TRANSLUCENT;
    }

    @Override
    public void setAlpha(int alpha) {
        mPaint.setAlpha(alpha);
    }

```

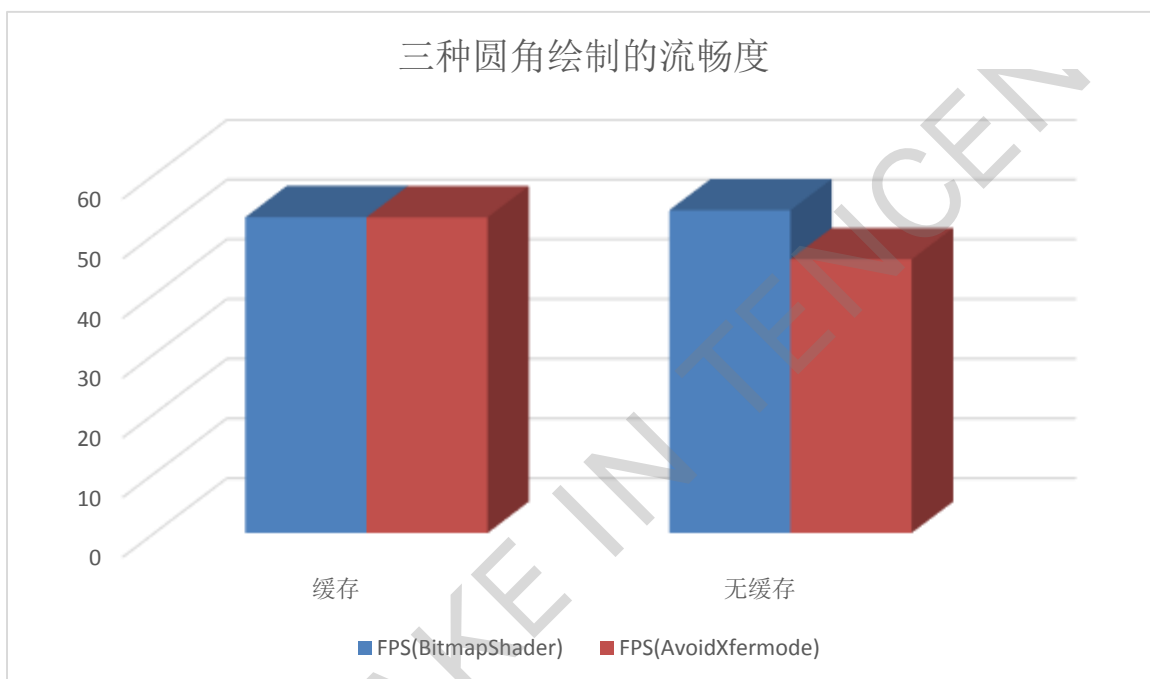
```

    }

    @Override
    public void setColorFilter(ColorFilter cf) {
        mPaint.setColorFilter(cf);
    }
}

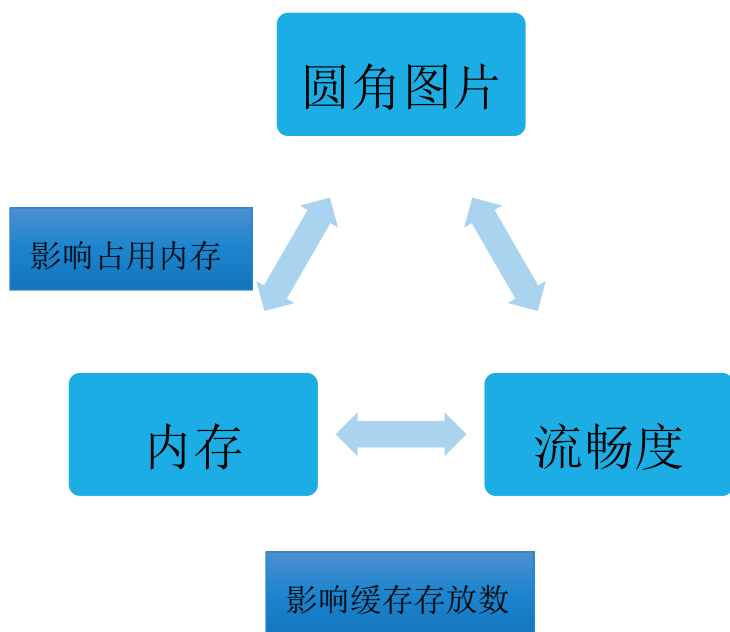
```

按照上面的代码实现了，我们不妨测试一下，这时需要定制一般应用基本的两个场景，模拟一个是无缓存的过程（首次滑动），另外一个是有缓存（二次滑动）的过程，使用 nexus s 测试，测试结果如下图，有缓存的话，fps 基本都一样，无缓存的场景 bitmapshader 方法的流畅度明显优于 avoidxfermode。



这样看来，用 bitmapshader 的价值并不是很大。但是真正应用的场景是复杂的，“缓存”毫无疑问是有限的，一般的设计思路是和 maxmemoryheap、剩余内存关联起来限制，并且使用 LRUCache 实现，这里就会产生淘汰的缓存，因此我们不难发现，无缓存的场景并非仅仅只有“首次滑动”，价值也就不只于“首次滑动”。另外有一个更重要的事情，使用 bitmapshader 因为他的原理所致，我们是可以使用 RGB_565 decode 图片，而 avoidxfermode 只能用 RGB_8888 来 decode 图片，这样会直接导致 bitmapshader 会节省约 2 倍的内存（有图有证据），同时也就意味着在有限缓存中，我们可以缓存更多的图片，这样也直接让流畅度持续保持在“有缓存”的状态。

cache-nogpu-method1.hprof			ImageUtil.java			PerformanceT...			cache-nogpu-...		
Overview			dominator_tree			dominator_tree			dominator_tree		
list_objects [selection...			list_objects [selection...			list_objects [sel...			list_objects [sel...		
Class Name	Shallow Heap	Retained Heap	Class Name	Shallow Heap	Retained Heap	Class Name	Shallow Heap	Retained Heap	Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Regex>	<Numeric>	<Numeric>	<Regex>	<Numeric>	<Numeric>	<Regex>	<Numeric>	<Numeric>
android.graphics.Bitmap @ 0x41378198	48	180,064	android.graphics.Bitmap @ 0x41378198	48	180,064	android.graphics.Bitmap @ 0x413bdb68	48	360,064	android.graphics.Bitmap @ 0x413bdb68	48	360,064



秉承美好事物都有缺陷的原则，我们在学习这部分的内容中，也发现 roy 大牛的 demo 有一个问题：如果把利用 bitmapshader 生成的 drawable 放到一个 warp_content 的 imageview 里面，是会显示不出来的问题。

但是在 <https://github.com/vinc3m1/RoundedImageView> 似乎有了很不错的解决。因此如果还没有用 bitmapshader 的同学吗？为什么不来尝尝这个老鲜肉呢~

响应时延

案例：ANDROID 应用发生黑屏的场景分析

黑屏产生的场景

1. 当应用启动时间超过 5 秒，几乎可以必现产生黑屏。
2. 启动新进程，未做优化，有可能会发生黑屏（如，当应用前台切后台主进程被杀，这时再从后台切前台，会出现黑屏）

实验 1：

我们在三星 9100 写了个简单的应用 DEMO，重写 onCreate 方法，加了一段执行超过 5 秒的代码。运行后，黑屏问题是必现的。

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_child);  
  
    float timenow= SystemClock.uptimeMillis();  
  
    while(SystemClock.uptimeMillis()-timenow<5200)  
    {  
        int i = 100;  
  
        i = i * i*i ;  
  
    }  
}
```

实验 2：

我们用酷派 8150，测试手 Q 启动，会出现两次黑屏。分别出现在登陆页面 前，和主页面消息 TAB 前。从时延日志中可以看到 LoginActivity 花了 7.2 秒，MainActivity 花了 6.3 秒。

这 2 个时间也验证，超过 5 秒会发生黑屏。

```
04-07 01:20:19.250 I/activity_launch_time< 1574>: [1084376400,com.tencent.mobile  
qq/.activity.SplashActivity,1898,1898]  
04-07 01:20:28.250 I/activity_launch_time< 1574>: [1084776184,com.tencent.mobile  
qq/.activity.MainActivity,5179,5179]  
04-07 01:21:09.578 I/activity_launch_time< 1574>: [1079201544,com.tencent.mobile  
qq/.activity.SplashActivity,1924,1924]  
04-07 01:21:27.125 I/activity_launch_time< 1574>: [1081758432,com.tencent.mobile  
qq/.activity.LoginActivity,7270,7270]  
04-07 01:22:58.820 I/activity_launch_time< 1574>: [1081745280,com.tencent.mobile  
qq/.activity.MainActivity,6361,6361]  
04-07 01:23:05.054 I/activity_launch_time< 1574>: [1085231728,com.tencent.mobile  
qq/.activity.phone.PhoneLaunchActivity,1518,1518]
```

实验 3：

这是手 Q 已知的一个 BUG，场景：

- 1) 启动手 Q
- 2) 进入一个好友会话
- 3) 手 Q 按 HOME 键，切入后台

4) 用 ADB Shell Kill 命令杀掉手 Q 进程

5) 从手机通知栏手 Q 图标进入手 Q(已设置手 Q 在通知栏显示),这时会出一段黑屏后,再进入 AIO

我们应该有这样一个疑问,在什么场景下,主进程会被杀?

除我们手动去杀进程外,Android 系统也会根据当前内存使用状态,自动地管理这些进程,具体见官方文档(Process lifecycle, <http://developer.android.com/guide/components/processes-and-threads.html>)

对如何避免这类黑屏问题,根据以上几个场景,其实已经不少解决方案,如:

1. 加闪屏

2. 优化加载

3. 处理进程被杀后,Activity 的加载顺序。

在实际的开发、测试 APP 时,我们应该考虑这几种场景、尽可能地避免黑屏。

案例:“首次打开聊天窗口”之痛

一、故事起点

手 Q 首次打开聊天窗口在我们核心性能监控中,耗时经历一次合流之后上升了不少。但是开发一直没有排查出来原因。我们尝试去定位问题。

二、分析定位问题

1. TraceView

录取“首次打开 AIO”过程的 trace,通过 Exclusive cpu time 排序发现,AbstractStringBuilder.append() 耗时严重,一层一层回溯其 parent 函数,能够定位到应用代码 QLogImpl.writeLogToFile 及 QLogImpl.getLogString:

并且耗时函数的调用次数颇多。

Name	Incl Cpu Time...	Incl Cpu Time
28 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	10.1%	199.177
Parents		
23 com.tencent/mobileqq/msf/sdk/QLogImpl.writeLogToFile Lcom/tence	41.8%	83.222
105 com.tencent/mobileqq/msf/sdk/QLogImpl.getLogString (Ljava/lang/	12.7%	25.257

2. Allocation Tracker

通过 Allocation Tracker 发现 StringBuilder 的 GC 频繁,并且很多是由于再次扩容导致的 GC。

Allocation Tracker					
File: <input type="text"/> [OK] [Trace]					
Alloc. Order	Allocation Size	Allocated Class	Thread	Allocated in	Allocated in
95	2064	char[]	58	com.tencent.mm.bkq.mf.a.k.QiaopingLL	InitialValue
418	518	char[]	42	java.lang.AbstractStringBuilder	<init>
167	528	char[]	65	java.lang.AbstractStringBuilder	<init>
462	500	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
504	496	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
222	496	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
509	376	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
470	376	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
225	376	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
377	360	char[]	30	java.lang.String	<init>
466	336	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
505	330	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
221	330	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
402	324	char[]	30	java.lang.String	<init>
308	308	char[]	69	java.lang.AbstractStringBuilder	enlargeBuffer
416	296	long[]	42	com.tencent.mm.bkq.app.MessageHandler	InitialGetFullTraceMsgHandler
389	268	String	30	java.lang.String	<init>
473	254	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
228	254	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
437	226	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
423	210	char[]	1	java.lang.String	<init>
112	208	char[]	69	java.lang.AbstractStringBuilder	enlargeBuffer
462	200	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer
4	200	char[]	20	java.lang.AbstractStringBuilder	enlargeBuffer
476	172	char[]	1	java.lang.AbstractStringBuilder	enlargeBuffer

3. 总结

经过查看相关函数的源码我们发现确实有较多的 `stringbuilder.append` 操作，例如这个。

```

StringBuilder stringBuilder = obtainStringBuilder();
stringBuilder.append(logTime);
stringBuilder.append("|");
stringBuilder.append(log.logTime);
stringBuilder.append("|");
stringBuilder.append(log.processName);
stringBuilder.append("[");
stringBuilder.append(log.processId);
stringBuilder.append("]");
stringBuilder.append(log.threadId);
stringBuilder.append("|");
stringBuilder.append(log.level);
stringBuilder.append("|");
stringBuilder.append(log.tag);
stringBuilder.append("|");
stringBuilder.append(log.msg);
stringBuilder.append("\n");
if(log.trace != null){
    stringBuilder.append("\n");
    stringBuilder.append(log.trace);
    stringBuilder.append("\n");
}

```

三、修改源码尝试优化

1. 修改点一：通过 `ThreadLocal` 减少同一线程重复生成 `StringBuild`
2. 修改点二：创建的时候传入预估的合适的容量大小，显示指定 `capacity`，避免二次扩容带来的时间开销及可能的 GC 开销。

```

public AbstractStringBuilder append(char[] str) {
    int len = str.length;
    ensureCapacityInternal(count + len);
    System.arraycopy(str, 0, value, count, len);
    count += len;
    return this;
}

void expandCapacity(int minimumCapacity) {
    int newCapacity = value.length * 2 + 2;
    if (newCapacity - minimumCapacity < 0)
        newCapacity = minimumCapacity;
    if (newCapacity < 0) {
        if (minimumCapacity < 0) // overflow
            throw new OutOfMemoryError();
        newCapacity = Integer.MAX_VALUE;
    }
    value = Arrays.copyOf(value, newCapacity);
}

```

通过查看 `append` 操作的源码我们发现 `append` 操作每次会先通过 `ensureCapacityInternal` 函数进行容量检查，默认是 16，如果容量不够则调用 `expandCapacity` 进行扩容，如果 `append` 操作频繁，会导致再次扩容耗时增加，并且可能导致 GC 开销。

因此我们可以在 `new StringBuilder()` 的时候，传入一个较为合适的参数，预估 `stringbuilder` 的可能大小，避免频繁扩容的开销。

如手 Q 中的以下代码：

```

char[] buffer = null;
StringBuilder result = null;
try {
    fis = new FileInputStream(file);
    reader = new InputStreamReader(fis, "UTF-8");
    buffer = new char[1024 * 4];
    // buf 9892
    result = new StringBuilder(4096);
    int n = 0;
    while (-1 != (n = reader.read(buffer))) {
        result.append(buffer, 0, n);
    }
} catch (Exception e) {
}

```

循环地从 `buffer` 中读取数据，加入到 `result` 这个 `StringBuilder` 中，在我们修改之前创建 `StringBuilder` 的操作是没有显示指定 `capacity` 的，但是代码可以看到 `buffer` 最大就是 4096，因此这个地方我们完全可以通过预估，指定一个合适的容量，避免再次扩容的耗时。

3. 修改点三：多个非常量字符串拼接的地方也需要显示指定容量。

如以下代码，存在很多字符串拼接操作：

```
@Override
public String toString() {
    return tag+" msName:"+msfCommand+" ssoSeq:"+getRequestSsoSeq()+" appId:"+appId+" appSeq:"+appSeq+
        +" uin:"+uin+" sCmd:"+serviceCmd+" t:"+timeout+" needResp:"+needResp;
}
```

而由于并非是常量字符串之间的拼接操作，因此 JAVA 本身会对其进行一次优化，优化成以下形式：

StringBuilder().append(...).append(...).toString()

然而由于拼接操作较多，如果不显示指定 capacity，又会存在再次扩容的开销。因此我们可以直接将上段代码修改成以下形式，减少再次扩容的发生：

四、优化前后效果对比

- ① Debug 版本的提升效果总体比 release 版本更好，这可能是因为 debug 版本的 log 比较多，这样我们针对优化的效果更加能够凸显出来。
- ② Debug 版本的提升大多在 6%，50ms 左右，其中首次打开群消息提升效果最突出，在 13.6%，100ms 左右，这可能是由于群组消息比较多，因此优化效果能够凸显。
- ③ Release 版本的优化效果没有那么明显，最多的才 60ms 左右，7%，并且还有两个场景是基本持平的。

工具集

PERFBOX 自研工具：SCROLLTEST

```

shell@android:/ $ dumsys SurfaceFlinger --latency com.tencent.mobileqq/com.tenc
ent.mobileqq.activity.SplashActivity
cy com.tencent.mobileqq/com.tencent.mobileqq.activity.SplashActivity      <
16383773
42999417419360  42999430969164  42999432586596
42999434814379  42999447509692  42999448638842
42999449096606  42999464019701  42999464721606
42999465301440  42999480255053  42999480956957
42999481536791  42999496612475  42999497497485
42999497894213  42999511122485  42999511422338

```

开始绘制

垂直同步

刷新该帧到屏幕

```

shell@android:/ # service call SurfaceFlinger 1013
service call SurfaceFlinger 1013
Result: Parcel(0000013ah '....')

```

Buffer交换次数

如图，这是我们利用 SurfaceFlinger 获取数据，然后统计出 fps 和 janky 的工具。涵盖了标准的拖拽能力设定，直接用于 fps 数据的获取，流畅度的测评。下面将介绍如何使用。

环境准备

开源工具下载：待定

检测工具：Scrolltest

手机需要有 Root 权限

安装 python2.7.X，<http://www.python.org/getit/>，并将 python 的安装目录设置到系统环境变量 PATH 中。注意，当前脚本仅支持 2.6.X、2.7.X 版本的 python。请不要使用 3.3.X 版本。

注：测试 FPS 值，为避免 debug 版本输出的大量日志对测试结果的影响，请务必使用 Release 版本

脚本使用方法

【快捷方式】:

*仅监控 FPS，执行 runFPSmonitor.bat。

*自动滑动并监控 FPS，执行 runScrolltest.bat

注：若电脑链接了多如手机，请在批处理文件中添加 -s 参数，指定设备序列号

【高级选项】:

cmd 命令行窗口定位到 Scrolltest 目录

*仅监控 FPS：执行 `python fpsmonitor.py [options]`

参数：

-s 设备序列号，当连接多台手机时，需指定设备序列号

-f fps 取样频率（默认 1 秒）

-o 测试结果的文件名，csv 格式（默认 result.csv）

-u 当指定该参数时总是使用 `page_flip` 统计帧率，此时反映的是全屏内容的刷新帧率。当不指定该参数时，对 4.1 以上的系统将统计当前获得焦点的 Activity 的刷新帧率。

*自动滑动并监控 FPS，执行 `python scrolltest.py [options]`

参数：

-a (必选参数)滑动的方向：u 向上滑动、d 向下滑动、ud 上下交替滑动、du 下上交替滑动（可指定多个，如： `-a u -a d`）

-c 滑动的次数(默认 20 次)

-f fps 取样频率（默认 1 秒）

-s 设备序列号，当连接多台手机时，需指定设备序列号

-o 测试结果的文件名，csv 格式（默认 result.csv）

-t 每次滑动的时间间隔(默认 1.0 秒)

-u 当指定该参数时总是使用 `page_flip` 统计帧率，此时反映的是全屏内容的刷新帧率。当不指定该参数时，对 4.1 以上的系统将统计当前获得焦点的 Activity 的刷新帧率。

--ds 指定屏幕分辨率（如： `--ds 宽*高`），通常不需要此参数，但当 monkey 取不到时，需要用户指定

打开要测试 FPS 值的界面，准备测试场景，执行 Scrolltest 脚本，并等待脚本自动执行结束。

```
C:\windows\system32\cmd.exe

D:\workspace\MobileTestTools_proj\speed\android-fps\scrolltest>python D:\workspace\MobileTestTools_proj\speed\android-fps\scrolltest\scrolltest.py -a ud
Starting ...
正在连接设备...
正在连接设备...
正在连接设备...
屏幕分辨率: 480*800
FPS monitor has start!
FPS:55 jank:0
FPS:54 jank:0
FPS:47 jank:1
FPS:50 jank:0
FPS:51 jank:0
FPS monitor has stop!
结果保存在: D:\workspace\MobileTestTools_proj\speed\android-fps\scrolltest\result.csv
终止批处理操作吗(Y/N)?
```

SYSTRACE（分析）

systrace 则提供了强大的初步定位能力，作为 Android 4.1 引入的一套用于做性能分析的工具，它可以输出各个线程当前的函数调用状态，并且可以跟当前 CPU 的线程运行状态、VSYNC、SurfaceFlinger 等系统信息在同一个时间轴上进行对比。但是很可惜的是，这个工具要求很多，只有寥寥可数的机型可以用，遇到一些不能用的机型的卡顿问题，就束手无策。但是它确实有无可替代的地位，所以我们下面还是具体介绍一下。下图就是用 systrace 得到的手空滚动好友动态时的日志：



如果 SurfaceFlinger 服务在每个 VSYNC 信号中断时就调用一次，就意味这应用显示非常流畅。对应到上图，就是每个 VSYNC 信号的上升沿或下降沿都有一个对应的 SurfaceFlinger 服务的调用。那我们就来放大看一看 VSYNC 和 SurfaceFlinger 服务（由于截图原因，线没对齐，大家将就看看）。



显，在 VSYNC 的上升沿 SurfaceFlinger 服务并没有被调用，导致了丢帧。有三种原因会导致这种情况的发生。



1. CPU 负载过大，这种情况多发生在单核低端机型上，以放大的 CPU 信息图为例：



如果在竖线处（即 VSYNC 中断信号），CPU 负载过大，无暇调用 SurfaceFlinger 服务，就会发生一次丢帧。如果此时 CPU 是在运行其它应用，那么我们无能为力，但如果此时 CPU 是在运行手空，那么就需要进一步分析（下文中会提到）。

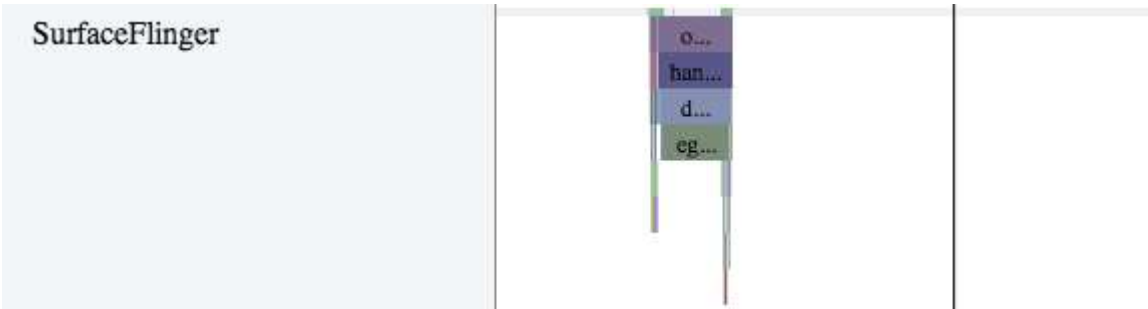
2. 应用侧没有完成绘制，而这个原因又可以细分为三个原因：

1) 应用内部忙于处理逻辑，如下图所示：



由于手空忙于执行其他的逻辑，没有进行绘制，这里还需要进一步分析（下文中会提到）。

2) 应用忙于分发响应事件，如下图所示：



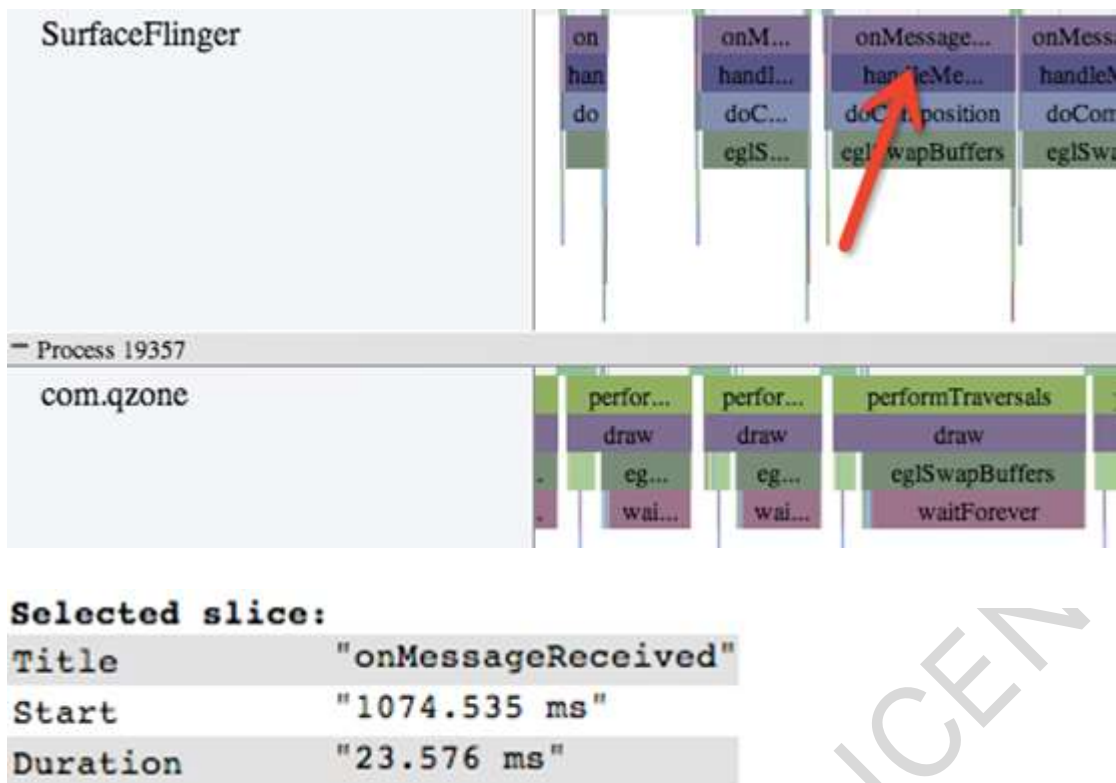
这说明当前窗口的视图过多、布局嵌套太深，导致查找响应输入事件的控件耗时太长，最后导致应用无暇绘制 UI。

3) 当前帧绘制耗时过长，如下图所示：



可以看到，当前帧应用侧绘制时间超过 39ms，造成了 1 到 2 帧的卡顿。追究其原因，还是因为视图过度、布局嵌套太深造成的。

3. 系统侧渲染时间过长



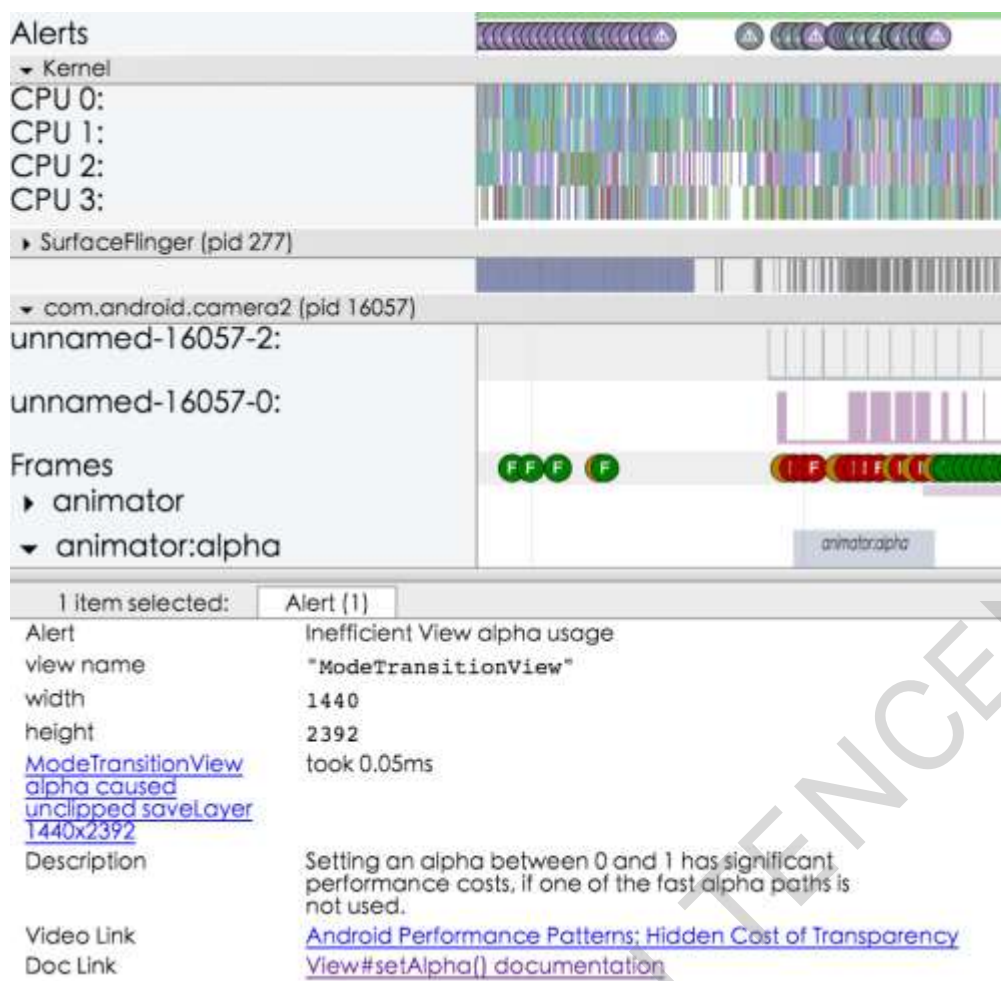
SurfaceFlinger 服务渲染的时间达到了 23ms，至少造成了 1 帧的卡顿。从交换 Buffer 的时间过长（几乎 23ms）推测可能是使用了三重缓冲机制，导致在处理缓冲区排序、交换时耗时过长。回想使用三重缓冲机制的原因，还是因为视图过多、布局嵌套太深导致。

综上所述，表现层影响应用流畅度的主要原因还是在应用侧视图过多、布局嵌套太深，优化措施可以参照 Lint 的 12 条规则。

更新：在 Google I/O 2015, SYSTRACE 有了一些不错的新功能。里面会提示下面的 alerts

Inefficient View alpha usage

这个告警要 android 5.1 才会有。这个错误是基于 render thread 信息提供。所以说的是设置 0~1 的透明度对 GPU 执行渲染的性能的消耗是有比较大的影响。



Expensive rendering with Canvas.saveLayer()

Canvas.saveLayer() 会有高昂代价的渲染性能的损耗。它们会打断绘制过程中的 rendering pipeline。替换使用 View.LAYER_TYPE_HARDWARE 或者 static Bitmaps。这会让我们离屏缓存在两帧之间被复用，同时避免了渲染目标由于切换被打断。

Path texture churn

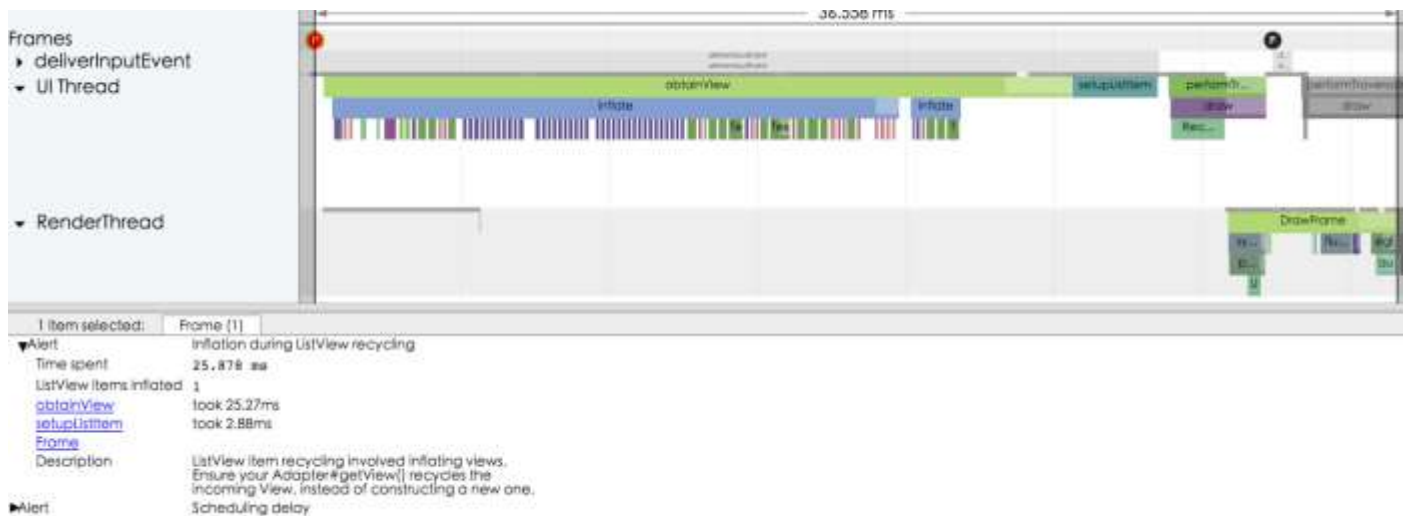
在使用遮罩纹理绘制的时候可以利用“绘制路径”减少性能损耗。当路径修改或者更新的时候，纹理必须重新生成并上传到 GPU 进行处理的时候，确认各个帧有共同的缓存路径，并且不需要调用 Path.reset()。这样就可以通过共享路径的方式减少更改路径的次数，体现在 Drawables 和 Views 之间绘制的性能损耗减小。

Expensive Bitmap uploads

Bitmaps 在硬件加速下，修改和图像的变化都会上传到 GPU，当这个像素的总量比较大，对 GPU 来说就有比较大的成本，建议在每个 FRAME 中减少这些图片的修改。

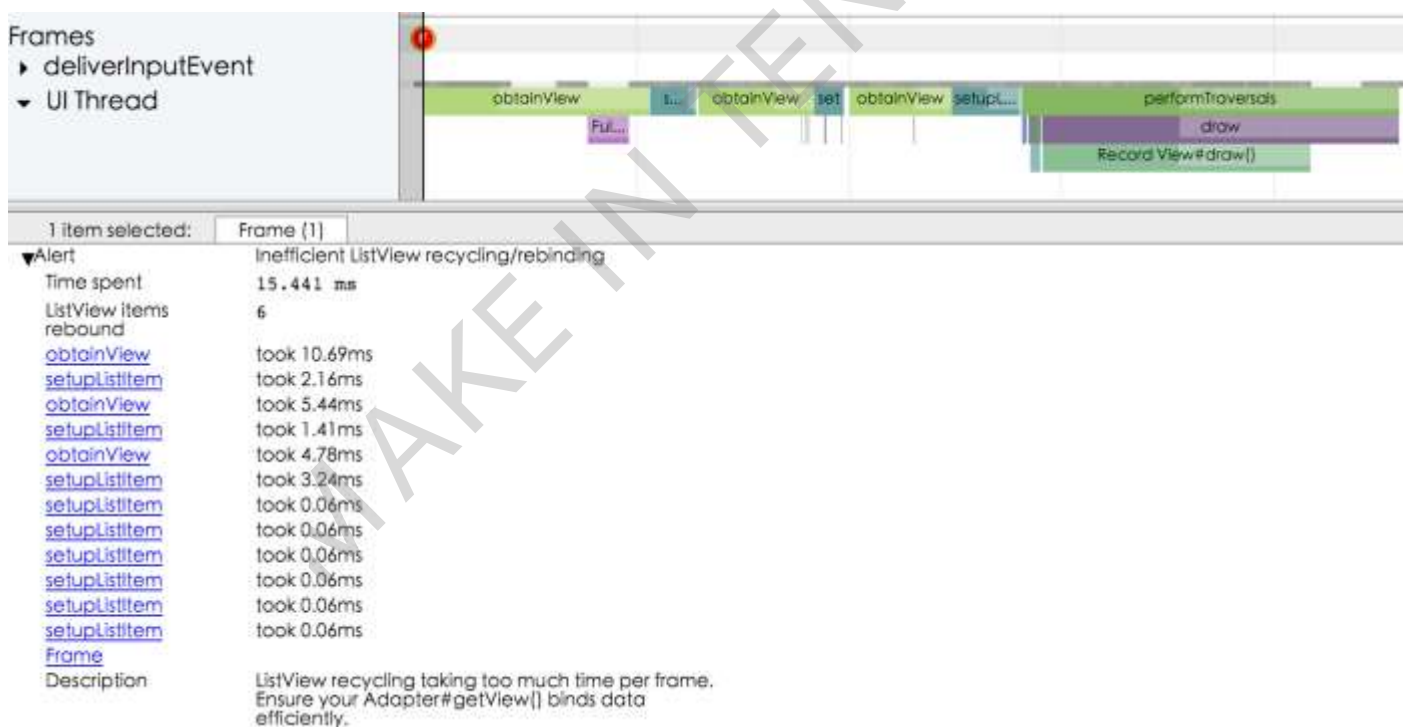
Inflation during ListView recycling

没有使用 ListView 的复用机制造成 inflate 单个 Item 的 getView 成本过高



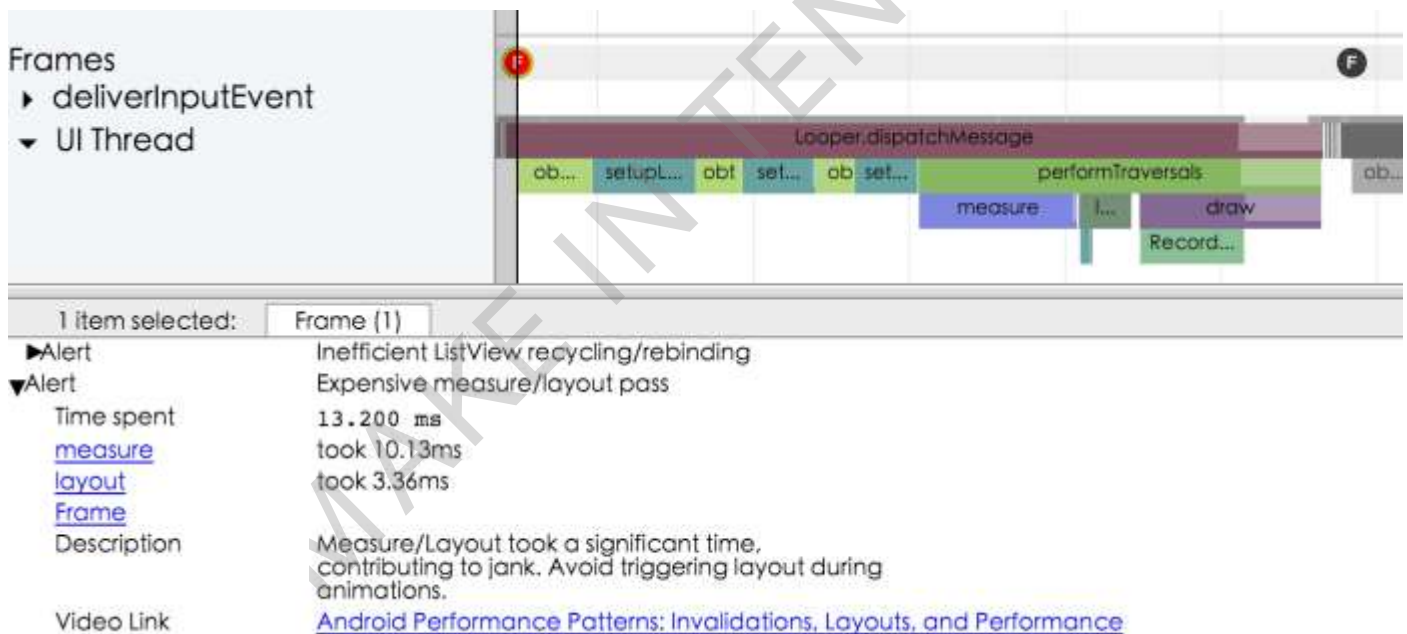
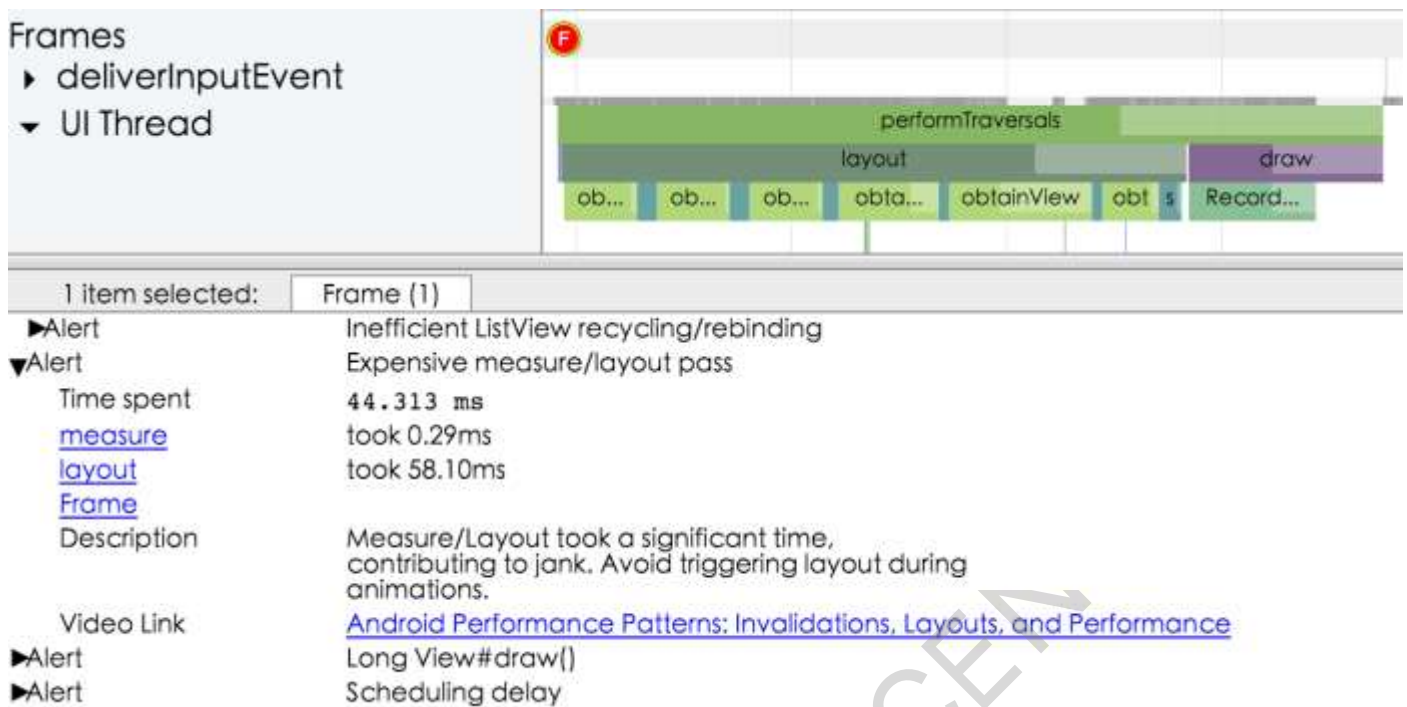
Inefficient ListView recycling/rebinding

每帧的 ListView recycling 耗时过长，需要看一下 Adapter.getView() 绑定数据的时候是否高效。



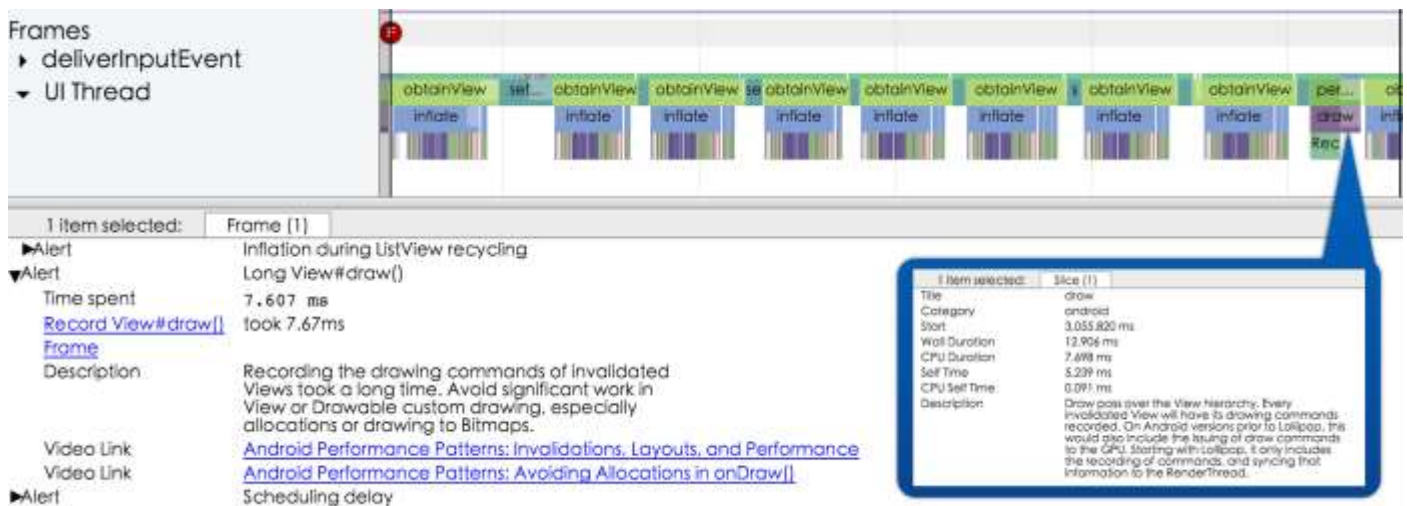
Expensive measure/layout pass

Measure/Layout 耗费了一定的时间从而导致 jank。当动画的时候，避免触发 Layout。



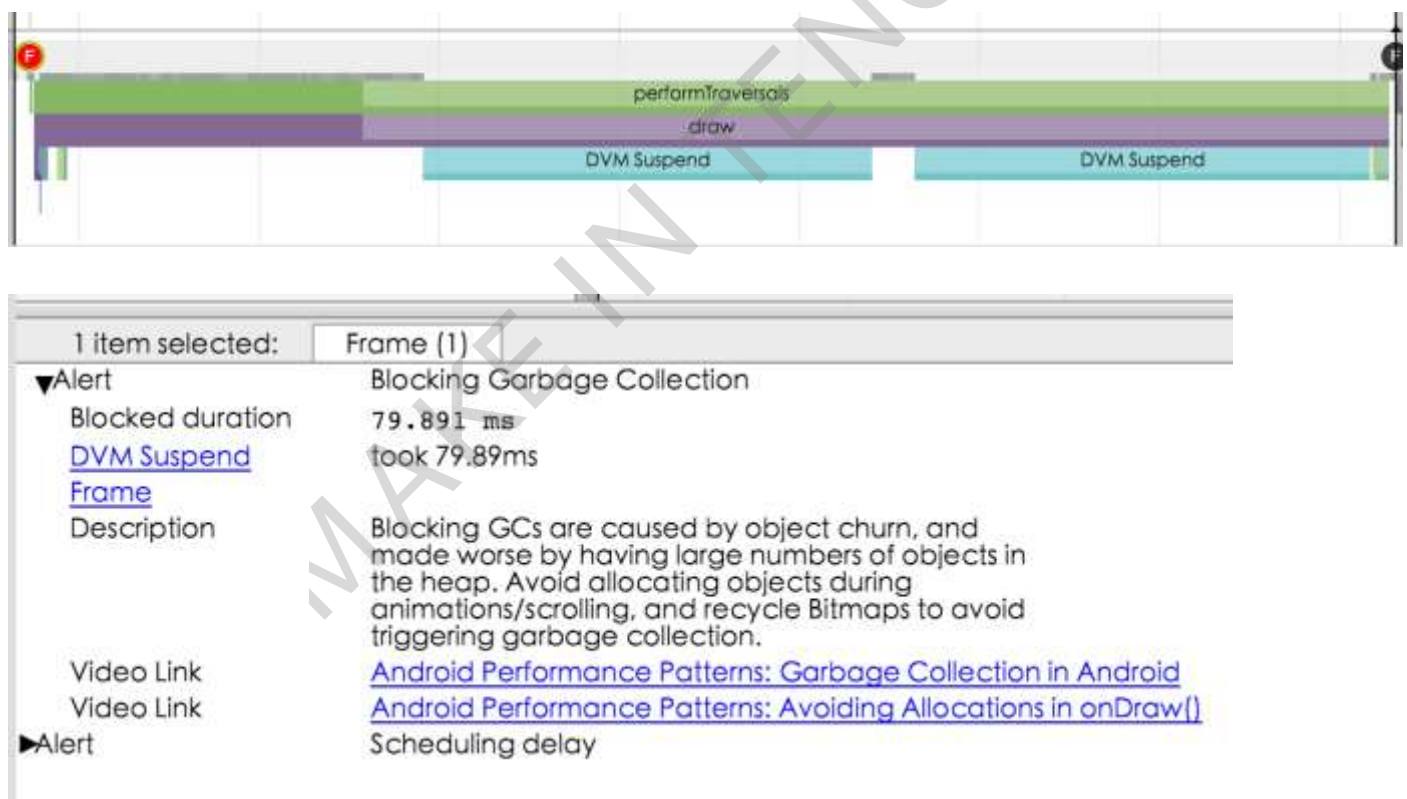
Long View.draw()

Draw 本身耗费了很长的时间。避免在 View 或者 Drawable 的 onDraw 里面执行任务繁重的自定义的操作，特别是申请内存或者绘制 Bitmap。



Blocking Garbage Collection

因为垃圾回收导致的卡顿。一定要避免在动画的时候生成对象，重用 bitmap 也可以避免触发垃圾回收。



Lock contention

UI 线程锁竞争的出现是因为 UI 线程尝试去使用其他线程持有的锁。当出现这种情况，UI 线程就会被 block，直到锁被释放。检查现有在 UI 线程的锁，并且确认它锁的区间耗时并不长。


Scheduling delay

因为网络 I/O, 磁盘 I/O 等线程资源争抢, 导致有一定时间的 UI 线程实际耗时长 (非 CPU 耗时), 而导致了卡顿。确认这些后台线程, 都运行在较低的线程优先级 (比 Thread_Priority_background 要低)。

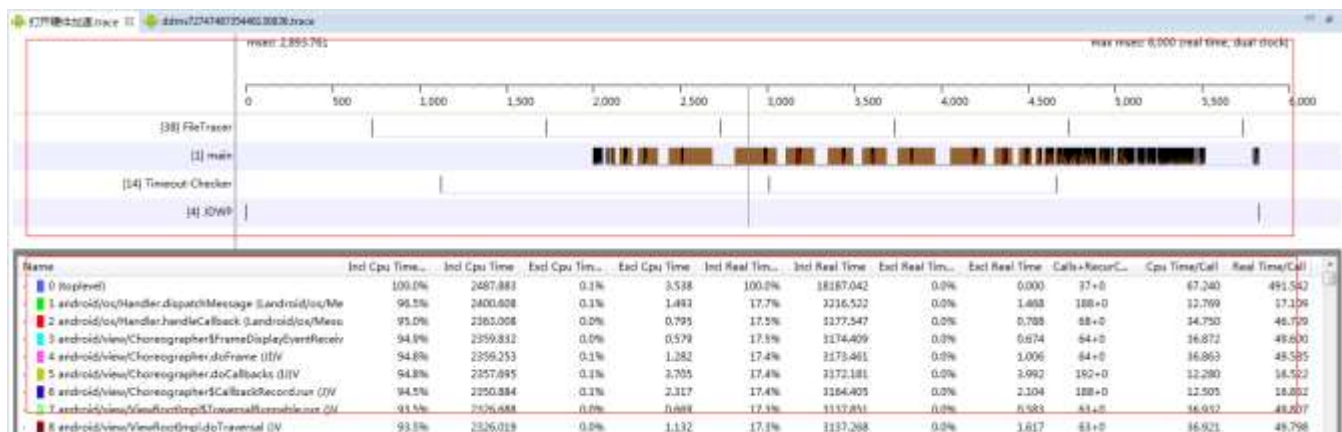
TRACEVIEW (分析)

录取 TRACE 方式

先说 TRACEVIEW 有三种启动方式, 不同的启动方式有不同的使用场景

方式	实现方式	使用场景
代码启动	<code>android.os.Debug.startMethodTracing();</code> 和 <code>android.os.Debug.stopMethodTracing()</code>	为了定位某个精准的区间的函数耗时问题。配合自动化测试最佳选择。
命令行启动	<code>adb shell am start -a android.intent.action.VIEW -start-profiler a.trace</code> <code>adb shell am start -a android.intent.action.VIEW -P b.trace</code> (当 app 进入 idle 状态的时候, profile 才停下来) 与 ddms 的使用相近: <code>am profile com.example.android.apis start /sdcard/android.trace</code> 和 <code>am profile com.example.android.apis stop /sdcard/android.trace</code>	定位程序启动过程的耗时问题
ddms 中启动		对于没有严格开始和结束的场景, 如动画卡顿, 流畅度类型的问题比较适用。

理解 TRACEVIEW



如上图所示，Trace 文件主要分两个区域。

上半区为时间片面板，X 轴表示的是时间消耗，单位为毫秒，Y 轴表示的是线程，每个线程中不同的颜色表示了不同的方法，颜色越宽表示该方法占用 CPU 时间越长。

下半区为方法耗时分析面板，给出了所有方法消耗时间的概况。

每个字段的含义：

列名	含义
Name	列出的是所有的调用方法，展开可以看到有 Parent 和 Children 子项，分别指该方法的调用者以及该方法调用的方法；
Incl CPU Time	某函数占用 CPU 时间，包含该方法调用的其他函数的 CPU 占用时间
Excl CPU Time	某函数自身占用的 CPU 时间，即不包含该方法调用的其他函数的 CPU 占用时间
Incl Real Time	某函数真实执行耗时，包含该方法调用的其他函数的耗时
Excl Real Time	某函数自身真实执行耗时，即不包含该方法调用的其他函数的耗时
Call+Recur Calls/Total	某函数被调用次数（含递归调用）占总调用次数的百分比
CPU Time/Call	某函数平均占用 CPU 时间
Real Time/Call	某函数平均真实执行耗时

这里重点说说 Calls + Recur Calls/Total 和 Cpu Time/Call。先说前者，有两个数值，方法调用次数、递归调用次数。下面这个例子就是方法调用了 632 次。

Name	Cpu Time/Call	Real Time/Call
getView (ILand...	1.124	2.536
or.getView (ILar	1.124	2.536
▼ Parents		
getView (I		
▼ Children		
self		

而后者，取得每次方法调用的 CPU 耗时，例如可以用来看 getView 的执行的耗时。一般比较复杂的应用都会有 1ms~2ms 左右的耗时。另外值得一提的是 real time，其实小 V 一直认为，real time 与 cpu time 的差值应该能帮助找 I/O，block，线程竞争，GC 的问题，只可惜据说这个实际耗时只有 5.0 的版本才比较准确。

GFXINFO（分析）

作为渲染流程中初步定位的工具，GFXINFO 非常合适。使用的方法也很简单。

1. 打开 GFXINFO



a. 打开开发者选项-> 选择 GPU 呈现模式分析

b. 选择在 adb shell dump gfxinfo

2. 执行命令行 `adb shell dumsys gfxinfo [app packagename]`, 例如：`adb shell dumpsys gfxinfo com.tencent.mobileqq`, 会获取一系列数据，包括：

Recent DisplayList operations: 最近 Displaylist 操作，其实就是一系列 OpenGL 的指令

Caches: 打开硬件加速后，会使用内存，上限是 8M。这个部分可以看到具体是什么东西构成这 8M

Profile data in ms: 这是最常用的，通过展示 128 帧，process, draw, execute 的耗时来分析卡顿问题。最终可以绘制出一个图标。Android 6.0 更新后，这里的数据更加丰富了，详见 [Slickr](#) 的介绍

View hierarchy: 包含 view 的个数，渲染的帧的数量以及 displaylist 的数据量

INTEL 的性能测试工具：UXTUNE（测评+分析）

<https://software.intel.com/zh-cn/android/articles/how-to-debug-an-app-for-android-x86-and-the-tools-to-use>

HIERARCHY VIEWER（分析）

Hierarchy Viewer 只是一个客户端，真正连接的是手机端的 view server，两者通过 socket 进行连接并传递数据。一般来说，默认都不会启用 View Server 的。

启用方法

要启用 View Server 有两个方法，

方法 1：

(通常 google 的一种方法是替换“/system/framework/services.odex”文件，需要先反编译生成 smali 文件，然后修改 smali 文件,在 isSystemSecure 函数中强制返回“false”,再编译、压缩、优化,替换。但是容易让手机变砖)

但其实对于 view server 来讲，其代码中会判断系统属性，只有对于 ro.secure 为 0 或 ro.debuggable 为 1 的系统才能启动。修改这个东西就可以了。

1. root 手机，为后续执行该工具做准备

2.将 setpropex push 到手机中（非 SD 卡），建议 push 到/data/local/tmp 下

[执行] adb push setpropex /data/local/tmp

3.修改 setpropex，将其设置成可执行。

[执行] chmod 777 setpropex

4.运行 setpropex 修改 ro.secure 为 0 或者 ro.debuggable 为 1

[执行] setpropex ro.secure 0

5.连接 HierarchyView 查看你想要查看的程序吧

注意：ro 属性在重启后会自动还原，因此，重启之后如果想再次使用 hierarchy view 需要重新执行第四步

(setpropex 可以到对应章节的 github 下载)

方法 2：

在项目里面引用并执行 <http://github.com/romainguy/ViewServer>。

使用方法：

通过上面的方法把 ViewServer 启动后， 就可以使用 HierarchyView 启动了。[待补充]

TRACER FOR OPENGL ES

[待补充]

功能

Slickr 集合 bash 及 python 脚本，用来对安卓应用的帧渲染性能进行耗时数据收集及分析，可通过分析每个阶段的耗时判断是否存在相关性能问题，可以自动模拟滑动当前屏幕，用来加入自动化测试，亦可将收集的数据通过图表的形式展现，易于进一步分析。

对比 DUMPSYS GFXINFO

Slickr 是基于 `dumpsys gfxinfo`（仅仅适用于硬件加速的平台）收集帧渲染性能数据的，但是具有更多的功能：

- 1) 可通过 `input touchscreen` 进行自动化模拟滑动操作，可通过参数指定滑动次数及滑动的垂直距离范围，可加入自动化测试中。
- 2) 对于 Android M，能收集到更为详尽的帧渲染过程的数据，将过程分为更加具体的阶段，如：`stat`、`input`、`animations` 到 `execute`、`process`
- 3) 能够将收集到的各帧性能数据，基于 `matplotlib` 画出直观的图表，查看到每帧对应的各个渲染阶段的耗时。

工具的使用说明

Github 链接：<https://github.com/ericleong/slickr>

1. 依赖环境

python 及 matplotlib 用来画图，源码的脚本应在 linux 下运行，可将 `slick.sh` 改造成 `slick.bat`，在 windows 下能够运行。

2. API 接口说明

源码中的 `slickr.sh` 是程序的入口，调用方式如下：

```
$ slickr.sh <package> <iterations> <distance>
```

第一个参数是指渲染性能监控的对象，传入的是包名，如 `com.tencent.mobileqq`;

第二个参数是指每 250ms 内滑动的次数，可缺省，默认为 4 次。

第三个参数是指滑动的垂直距离，也就是在如下命令中的 `VERTICAL` 参数

```
do input touchscreen swipe 100 $VERTICAL 100 0 250;
```

Swipe 后面的四个参数分别为 `<x1,y1,x2,y2>`，`VERTICAL` 对应 `y1`，也就是由 `(100,$VERTICAL)` 滑动到 `(100,0)`，`$VERTICAL` 决定了滑动的垂直距离，可缺省，默认值根据 `adb shell wm density` 或者 `adb shell getprop | grep density` 取出 `density` 后乘以 3 得出。

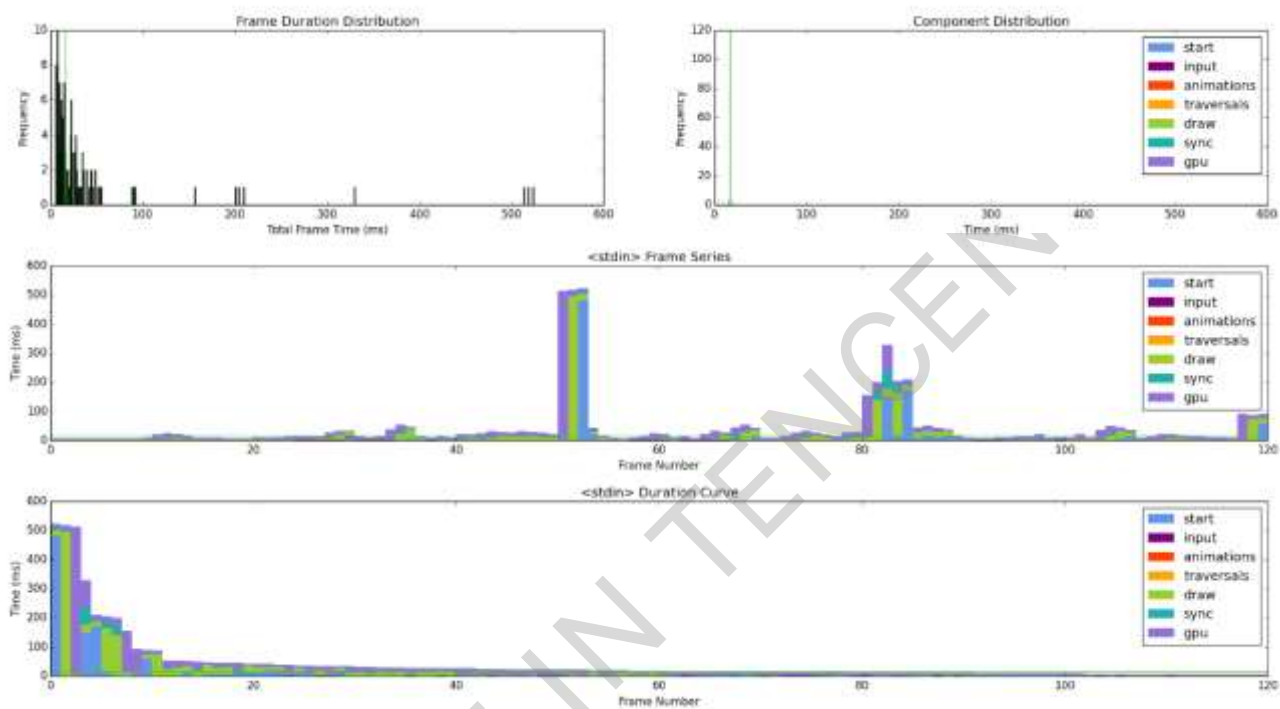
3. 使用示例

具体各个时间节点的含义可参考：

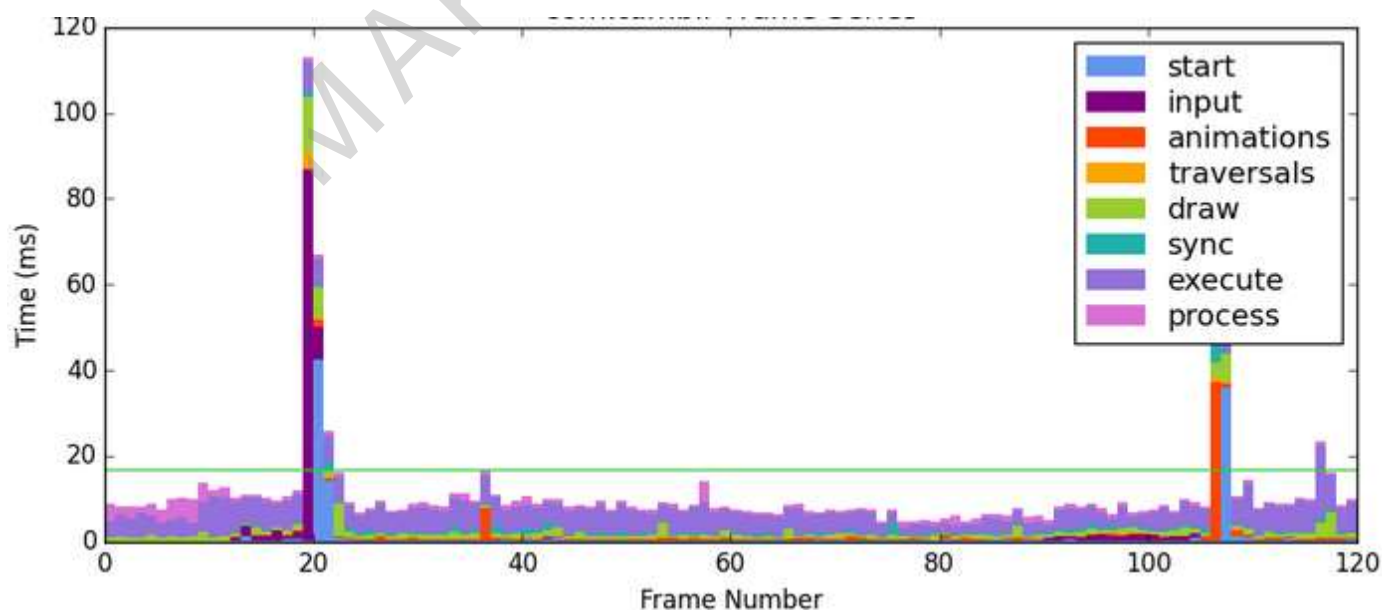
<http://developer.android.com/preview/testing/performance.html#fs-data-format>

而下图中各个时间段的值，正是由每一行数据的某两列相减算出来的，如 start 时间是由 `HANDLE_INPUT_START-INTENDED_VSYNC` 得到。

每一次调用 `plot.py` 会画出四种类型的图：



以下为输出的帧渲染耗时直方图：



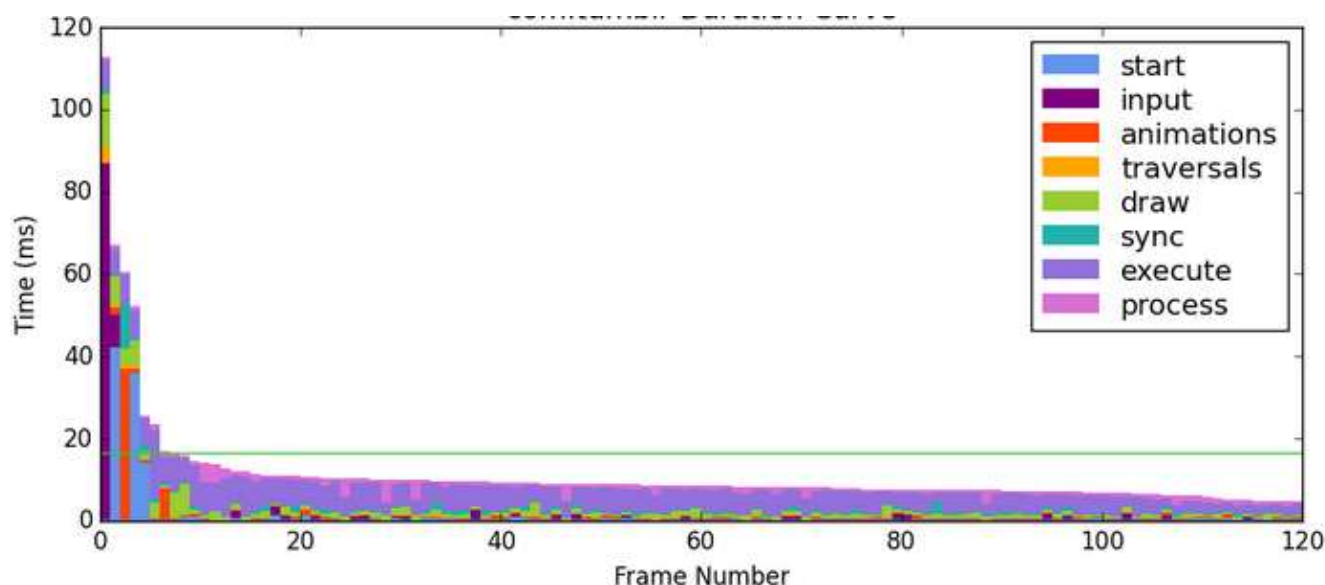
时间段	Gfxinfo 对应阶段	对应 framestats 的起始时间节点	含义
start		INTENDED_VSYNC →HANDLE_INPUT_START	系统处理开始的时间
input		HANDLE_INPUT_START →ANIMATION_START	处理输入事件的时间
animations		ANIMATION_START →PERFORM_TRAVERSALS_START	评估运行动画的时间
traversals		PERFORM_TRAVERSALS_START →DRAW_START	Measure 和 layout 阶段的耗时
draw	draw	DRAW_START →SYNC_START	View.draw() 的耗时, 构建 displaylist 的时间
sync	prepare	SYNC_START →ISSUE_DRAW_COMMANDS_START	传递数据给 GPU 花的时间
execute	execute		执行 display list 命令的时间
gpu	process	ISSUE_DRAW_COMMANDS_START →FRAME_COMPLETED	等待 GPU 的时间

我们可以根据上述详细的过程耗时进行分析, 如: 若 animation 的耗时大于 2ms, 那么可能是 APP 存在有不合适的自定义动画等。

具体的各个时间含义可参考:

<http://developer.android.com/preview/testing/performance.html#fs-data-format>

同时该工具也可根据数据生成时间曲线图:



该图根据帧渲染时间从大到小重新排序，我们可以一目了然地查看到多少帧超过了 16ms 的渲染时间，多少帧是在 16ms 之内的。

资源类性能

内存篇

内存是手机中最纠结的部分，特别是 android 手机，图片 OOM，甚至衍生到初始化界面都会 CRASH，内存成为开发最头痛的问题。而其中，最简单的莫过于内存泄漏，特别是大部分的泄漏会体现在 Activity 上。而更复杂的则是内存常驻，例如 Bitmap 使用 8888 而没有使用更优的 565，图片缓存里面有重复的图片，static 变量太多等等。常驻太多、占用内存太大自然就会在系统低内存的时候让 APP 成为被系统 KILL 的对象。但是过程总没有那么简单，还要伴随着 GC，而 GC 会影响 APP 的交互类性能，下面我们一起来看一下这些工具和案例。

工具集

曲线图

PSS: Proportional Set Size，比例集合大小

USS: Unique Set Size，独占集合大小

从 Google 对 Android 的设计可以得知：所有 Android 进程的 PSS 相加等于终端的物理内存大小；USS 则是杀死一个进程后能够释放的物理内存大小。

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。

官方推荐使用曲线图来衡量 APP 的物理内存占用，所以用户在纯净的 android OS 上所能看到的唯一内存指标（在【设置-应用程序-正在运行的 xxx】）就是 PSS，4.4 之后加入了 USS。

但是 PSS，有个很大的麻烦，就是“共享内存”，这种情况发生在 A 进程与 B 进程都会使用一个共享库 x.SO，那么 x.SO 中初始化所用的那部分内存就会被平分在 A 与 B 的头上。但是 A 是在 B 之后启动的，那么，对于 B 的 PSS 曲线图而言，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行为造成致命的麻烦。

USS 虽然没有这个麻烦，但是由于 Dalvik 虚拟机申请内存，牵扯到 gc 时延和多种 gc 策略，都会影响曲线的异常波动。比如异步 GC 是 4.0 以上系统很重要的新特性，但是 GC 什么时候结束呢？曲线什么时候“降”，就变得很诡异了，而测试通常希望退出某个界面后可以明显看到曲线有个大的降落。还有 GC 策略，什么时候开始增加 Dalvik 虚拟机的预申请内存大小（Dalvik 启动时是有一个标称的 start 内存大小的，给 java 代码运行时预留，避免 java 运行到时再申请造成卡顿），但是这个预申请大小是动态变化的，这也会造成 USS 忽大忽小。

注：4.4 以后有增加了一个新的名词叫做 processStats 用以反映内存负载，其最终计算也是用到了 PSS）。

TOP

得到内存曲线的方法很多，其中 top 就是一种，但是很遗憾它的输出列信息中只包含了 RSS 与 VSS，所以 android 中 top 的使用更多的集中在某个进程的 CPU 负载方面。借着介绍 top 的契机，说下上一章没有提到的 RSS、VSS

RSS: Resident Set Size 常驻集合大小

VSS: Virtual Set Size 虚集合大小

RSS 与 PSS 相似，也包含了进程共享内存，但有一个麻烦，RSS 并没有把共享内存大小去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，它的上限和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，就比如在 A 地址有一块内存，在 B 地址也有一块内存，那么 VSS 就等于 ASzie 加 BSize，至于内存是什么属性，它并不关心。所以以至于很多 file 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，它和进程使用内存并没有多少关系（因为我们关注的是“运行时内存”）。

Procrank

Procrank 拥有打印进程 VSS, USS, RSS, PSS, 以及排序功能。

```
# procrank -h
Usage: procrank [ -W ] [ -v | -r | -p | -u | -h ]
  -v  Sort by VSS.
  -r  Sort by RSS.
  -p  Sort by PSS.
  -u  Sort by USS.
      (Default sort order is PSS.)
  -R  Reverse sort order (default is descending).
  -w  Display statistics for working set only.
  -W  Reset working set of all processes.
  -h  Display this help screen.
```

但是可惜这样一个工具注定因为功能过于单一, 不适合 android 测试而退出了历史舞台。据测试: 4.2 以上的 android 系统已不支持 procrank (当然也可以自助去下载 libpagemap.so, 来完善 4.2 系统, 但明显测试代价巨大, 而且数据准确度也有待商榷)。

```
root@android:/system # procrank
procrank
soinfo_link_image(linker.cpp:1635): could not load library "libpagemap.so" needed by "procrank"; caused by is_prelinked(linker.cpp:667): prelinked libraries no longer supported: libpagemap.soCANNOT LINK EXECUTABLE
255!root@android:/system #
```

No longer supported

MEMINFO

介绍完曲线图, 我们知道了它们的适用范围, 以及局限性。接下来介绍一个 android 官方非常推荐的工具 meminfo, 它是 android 基础服务接口其中的一个接口, 有兴趣的读者可以使用【adb shell service list】来展示 dumpsys 支持的打印信息。

```

C:\Users\Novels>adb shell service list
Found 65 services:
0    ijinshan_cleanmaster_cn_rtsrv: [com.ijinshan.rt.common.IRootKeeper]
1    sip: [android.net.sip.ISipService]
2    phone: [com.android.internal.telephony.ITelephony]
3    iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
4    simphonebook: [com.android.internal.telephony.IIccPhoneBook]
5    isms: [com.android.internal.telephony.ISms]
6    assetredirection: [com.android.internal.app.IAssetRedirectionManager]
7    samplingprofiler: []
8    diskstats: []
9    appwidget: [com.android.internal.appwidget.IAppWidgetService]
10   backup: [android.app.backup.IBackupManager]
11   uimode: [android.app.IUIModeManager]
12   usb: [android.hardware.usb.IUsbManager]
13   audio: [android.media.IAudioService]
14   wallpaper: [android.app.IWallpaperManager]
15   dropbox: [com.android.internal.os.IDropBoxManagerService]
16   search: [android.app.ISearchManager]
17   country_detector: [android.location.ICountryDetector]
18   location: [android.location.ILocationManager]
19   devicestoragemonitor: []
20   notification: [android.app.INotificationManager]
21   profile: [android.app.IProfileManager]

```

Meminfo 的使用如下：

meminfo dump options: [-a] [--oom] [process]

-h 帮助信息

-a 打印所有进程的内存信息，以及当前设备内存概况

--oom 按照 oom adj 值进行排序

[process] 可以使进程名称，也可以是进程 id，来打印某个进程的内存信息

如果不输入参数，meminfo 只会打印当前设备的内存概况，由三个部分构成

```
管理员: C:\windows\system32\cmd.exe - adb shell
Total PSS by process:
65291 kB: com.android.launcher <pid 609>
56091 kB: system <pid 391>
41691 kB: com.android.systemui <pid 468>
40441 kB: com.tencent.qqlive <pid 990>
40441 kB: com.tencent.android.qqdownloader:uninstall <pid 1049>
37213 kB: com.tencent.qqlite <pid 5405>
23675 kB: com.tencent.android.qqdownloader <pid 23562>
14552 kB: com.tencent.qqinyin.service <pid 578>
11984 kB: com.qzone.service <pid 16676>
11671 kB: com.qzone <pid 31939>
10662 kB: com.letv.android.client:hdservice_v1 <pid 1571>
9886 kB: com.tencent.qqlite:MSF <pid 5427>
9778 kB: android.process.media <pid 567>
9367 kB: com.google.process.gapps <pid 784>
9028 kB: com.tencent.android.qqdownloader:connect <pid 923>
8564 kB: com.letv.android.client:remote <pid 1799>
7521 kB: com.letv.android.client <pid 1486>
7417 kB: com.google.android.apps.maps:GoogleLocationService <pid 31261>
7054 kB: com.google.process.location <pid 716>
6501 kB: com.android.phone <pid 598>
6460 kB: com.google.android.apps.maps <pid 1865>
5509 kB: com.google.android.gallery3d <pid 15953>
5126 kB: com.google.android.apps.maps:LocationFriendService <pid 31296>
4544 kB: com.google.android.deskclock <pid 28719>
4411 kB: com.google.android.apps.uploader <pid 1662>
4219 kB: com.android.nfc <pid 604>
4049 kB: com.google.android.apps.maps:FriendService <pid 31283>
3062 kB: com.android.nfc:handover <pid 652>
2927 kB: com.android.location.fused <pid 730>

Total PSS by OOM adjustment:
56091 kB: System
52411 kB: Persistent
65291 kB: Foreground
19483 kB: Visible
27365 kB: Perceptible
19401 kB: A Services
```

```
10662 kB: com.letv.android.client:hdservice_v1 <pid 1571>
9778 kB: android.process.media <pid 567>
8564 kB: com.letv.android.client:remote <pid 1799>
6460 kB: com.google.android.apps.maps <pid 1865>
193629 kB: Background
40441 kB: com.tencent.qqlive <pid 990>
40441 kB: com.tencent.android.qqdownloader:uninstall <pid 1049>
37213 kB: com.tencent.qqlite <pid 5405>
23675 kB: com.tencent.android.qqdownloader <pid 23562>
11671 kB: com.qzone <pid 31939>
9028 kB: com.tencent.android.qqdownloader:connect <pid 923>
7521 kB: com.letv.android.client <pid 1486>
5509 kB: com.google.android.gallery3d <pid 15953>
5126 kB: com.google.android.apps.maps:LocationFriendService <pid 31296>
4544 kB: com.google.android.deskclock <pid 28719>
4411 kB: com.google.android.apps.uploader <pid 1662>
4049 kB: com.google.android.apps.maps:FriendService <pid 31283>

Total PSS by category:
214544 kB: Dalvik
103932 kB: Other dev
86289 kB: Unknown
30263 kB: .dex mmap
25372 kB: .so mmap
4443 kB: .apk mmap
2204 kB: Other mmap
1567 kB: .ttf mmap
458 kB: Native
28 kB: Cursor
28 kB: Ashmem
? kB: .jar mmap

Total PSS: 469135 kB
```

pss 排队，用以查看进程的内存占用，一般用它来做初步的竞品分析，同样功能的应用程序，应该具有相同的 pss，这是竞品分析最根本的理由。

oomAdj 排队，展示当前系统内部运行的所有 android 进程内存状态，和被杀顺序，越靠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，服务或界面，可见与否，老化等等，但其查看的意义大于测试，可以做竞品对比，比如先后退回后台的被测产品与竞品，过没多久出现竞品排名在被测产品的上方的情况（即被测产品退回后台更容易被系统杀掉，不过在 4.4 以后出现“内存负载”概念后，这也不一定是坏事儿了）。

整机 pss 分布，按照降序排列各类 pss 占用，此部分仅用于粗略查看设备内存概况，也可以看看物理内存的使用是否已接近物理内存的最大值。（注：据作者测试，在 pss 的值为 80% 的设备物理内存时，Android 系统开始杀死进程）

输入进程标识参数后，meminfo 会打出一份有关进程的详细内存概况：

```

** MEMINFO in pid 31939 [com.qzone] **

```

	Pss	Shared Dirty	Private Dirty	Heap Size	Heap Alloc	Heap Free
Native	16	12	16	3040	2678	89
Dalvik	5276	10688	4916	10860	9678	1182
Cursor	12	0	12			
ashmem	0	0	0			
Other dev	4	28	0			
.so mmap	733	2184	544			
.jar mmap	1	0	0			
.apk mmap	348	0	0			
.ttf mmap	0	0	0			
.dex mmap	3776	0	84			
Other mmap	98	16	40			
Unknown	1844	420	1836			
TOTAL	12108	13348	7448	13900	12356	1271

Objects			
Views:	0	ViewRootImpl:	0
AppContexts:	2	Activities:	0
Assets:	2	AssetManagers:	2
Local Binders:	18	Proxy Binders:	14
Death Recipients:	1		
OpenSSL.Sockets:	0		

SQL			
MEMORY_USED:	115	MALLOC_SIZE:	62
PAGECACHE_OVERFLOW:	28		

DATABASES				
pss	dbss	Lookaside(b)	cache	Dbname
4	76	108	4/27/5	/data/data/com.qzone/database

介绍了一个进程的 PSS 构成，比较重要的指标有：

1. PSS 的 total，它可以当作整个 APP 当前消耗的物理内存大小
2. Dalvik 行的 Heap Alloc，可以作为当前 APP java 层内存分配的大小
3. Native 层的内存大小在 4.4 以下的系统，会用 Unknown 方式展示，而 4.4 则展示在 Native 的 pss 字段
4. 绘图的硬件加速消耗内存存在 4.4 以下系统，会用 Other dev 方式展示，而 4.4 则展示为 GL 和 Graphics
5. Objects 的 Views，是当前 APP 内存中所包含的激活态 view 数量，如果 view 数量随着操作持续上升，那么可能出现界面控件泄漏了

6.Objects 的 Activities，是当前 APP 内存中所包含的激活态的 Activity 数量（窗口数量），和 view 一样，如果持续增加，可能出现界面窗口泄漏了

PROCSTATS

从基础的曲线图 PSS 引出了 Meminfo，那么怎么样才能把 Meminfo 这样一个“点”上的值变成一个“统计”的值呢？画二维曲线是一种方法（纵轴 pss，横轴时间的画法），但不能这并不能为用户选择 app 提出好的建议（太难于理解，不好简易量化），所以 Android 想出了更好的办法，即“内存负载”。这个概念在 Android 4.4 被提出来，那么负载是怎么计算的呢？如下公式：

内存负载=PSS*运行时长

运行时长被分为前台、后台和缓冲（与 app 在设备上运行状态一致），与之对应内存负载就出现了前台内存负载、后台内存负载和缓冲内存负载这三个概念。为了更直观展示它们就有了 Procstats 工具，它位于 Android L 的开发者选项中，虽然官方也没有说什么时候要把它移出来给用户看，但相信很快就会被公诸于众。



Procstats 的查看方式很简单：

1. 每个进程背后都有个百分比，他是统计“此状态下的”运行时间，所谓的此状态下即上文中所述的一一前台、后台、缓冲。默认展示的是“后台负载”，所以按照这个图所展示的，QQ MSF 进程在被统计的 2 小时 16 分钟中，位于后台的运行时长也是 2 小时 16 分钟。
2. 每个进程都有一条绿色的进度条，越长表示负载越高，没有一个统一的刻度值，只是一种展示而已，它的长短由 app 的 PSS 和此状态下的运行时间的【积】来决定，如图：微信的内存负载一定高过 QQ 空间。

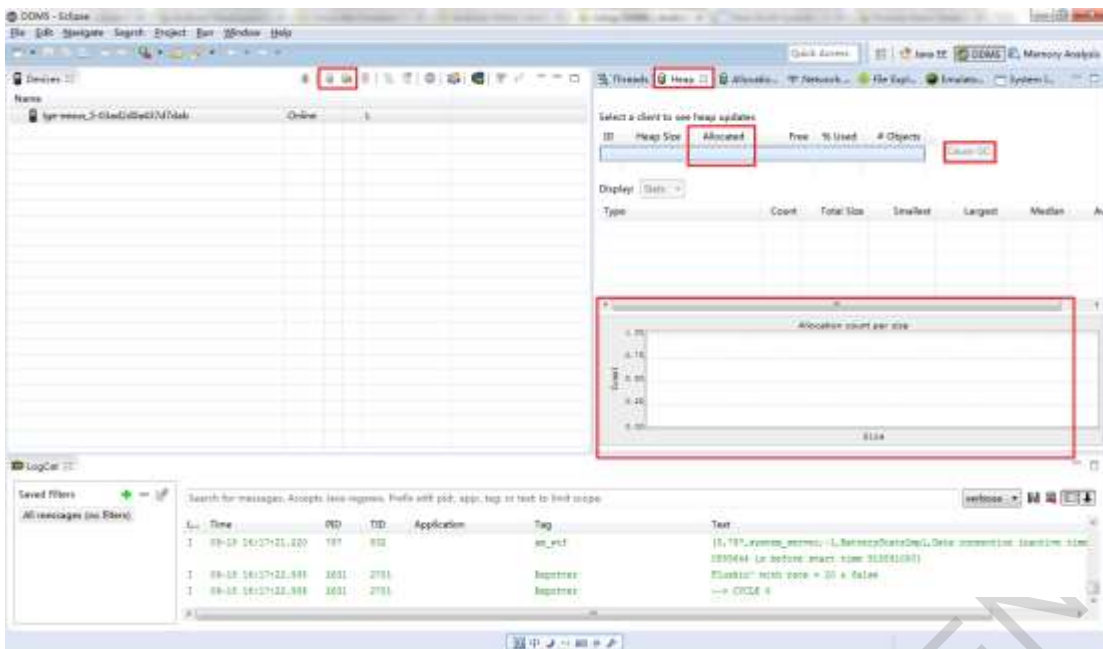
引申一下：三个状态中对 Android 系统来说，有如下潜规则：

1. 对于【前台】而言，是用户正在使用的，所以这部分内存一定要保证，是用户不 Care 的，所以在内存负载中不应该“默认展示”
2. 对于【缓冲】而言，Android 认为可以回收，此类软件有良好的被杀恢复能力，所以，没有将它杀死完全是系统的责任，在内存负载中也不应该“默认展示”
3. 对于【后台】而言，Android 认为这完全是 APP 行为，而且系统因为种种原因也无权杀死它回收内存，而且并非用户当前所使用（正在使用的 APP 是运行于“前台”状态下的），所以有可能是用户不想支付的代价，所以这部分内存负载应该被“默认展示”

作为测试人员，可以通过竞品对比看看那个 APP 在后台的内存负载更高，用以说明被测软件在完成自我特性的同时，内存指标是否被 Android 系统认可（即软件的全局观），认可度越高的 APP 相比认可度低的 APP 而言肯定更容易被用户所青睐。

DDMS

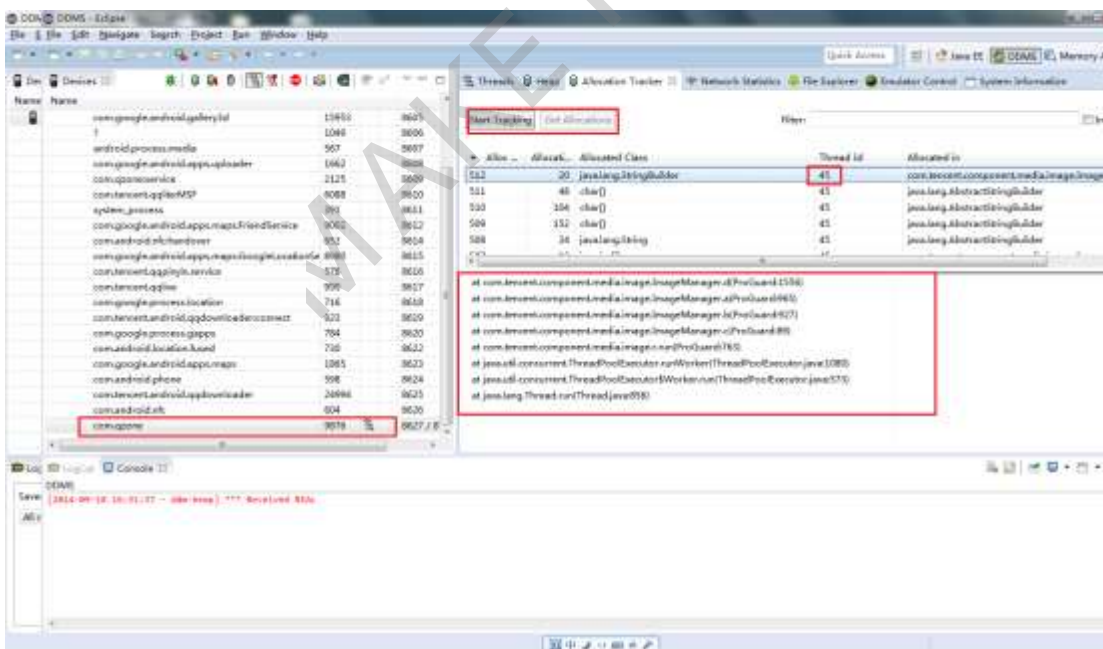
DDMS 的全称是：Dalvik Debug Monitor Server，Dalvik 虚拟机调试监控服务。



DDMS 是一个调试信息合集，里面包含了时延、内存、线程、cpu、文件系统、流量等一系列信息获取和展示，其中和内存相关的要提到两个功能：Update Heap、Allocation Tracker、Dump Hprof file。

Update Heap: 会获取 GC 的信息，包括当前已分配内存，当前存活的对象个数，所剩内存，动态虚拟机 heapSize，还有一个分配大小分布柱状图，主要用于查看，问题定位能力有限。

Allocation Tracker: 会展示最近的 500 条内存分配，以及分配发生时刻的线程堆栈信息，官方推荐用它来做流畅度问题判定（官方建议不要在主线程 main 中有大内存申请行为，这会影响流畅度）。



Dump Hprof file:

用以对选中的进程进行内存快照，至于内存快照的使用将会在 MAT 节中介绍。

MAT

全称 Memory Analyzer 内存分析器，MAT 是 IBM Eclipse 顶级开源项目，但 MAT 的设计初衷并非专门用于分析 Android 应用程序内存，它最初的作用是分析运行在 J2SE 或 J2ME 下的 java 类型应用程序的内存问题。由于 Dalvik 在一定程度上也可以理解为 java 虚拟机，google 就没有重复开发内存分析工具，而延用了 MAT。

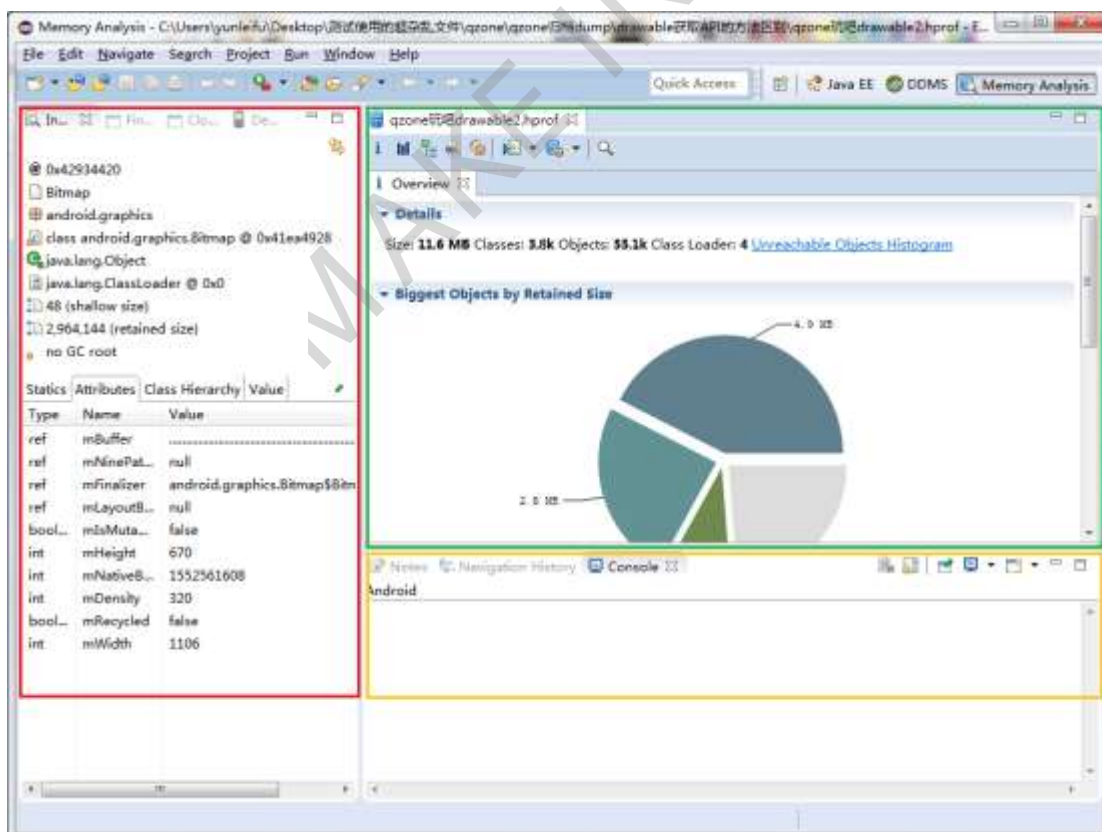
使用 MAT 需要抓取 Hprof 文件（内存快照），抓取 Android 应用程序快照的方法有很多，上一节说的 DDMS 的 Dump Hprof file 功能就是其中最简单，通用性最强的一种。以下还有两种比较不常用的：

1. 在 adb shell 模式下使用 kill -10 pid
2. 在 adb shell 模式下使用 am dumpheap pid outfilePath

通过 1 抓取的 hprof 文件位于 /data/misc 文件夹下，但是 4.x 以上系统就不支持这条命令了，am 命令虽然可以指定手机端的输出目录，但是分析过程仍需要把它从手机端复制到电脑端。

通过 DDMS 抓取的 Hprof 文件不能直接用给 MAT，需要通过 android SDK tools 中的 hprof-conv 工具转换一下，才能用给 MAT。不过在安装有 DDMS 的 eclipse 上继续安装 MAT 插件，这样使用 DDMS 抓取 Hprof 文件，就不会出现“另存为”窗口，而会直接自动转化后，在 MAT 插件中展示了。

使用 MAT 打开 Hprof 文件后，通常会见到如下界面：



Insepctor: 红色框出来的部分，用于展示对象的一些信息，比如：继承关系、成员、内部静态变量等，非常重要。

内存详情: 绿色框出来的部分，用于展示一个快照的内部数据，这是分析工作主要的操作台。

状态以及扩展窗口: 黄色部分，在这里可以看到操作记录，工作记要，命令行信息等等，可以辅助记录工作摘要。

使用 **MAT** 前有几个重要的知识需要掌握

1. **GC** 的原理，即对象之间的引用关系的理解
2. 被测产品的一般内存特征，比如：在 **xxx** 场景下，内存中有多少个 **Map** 对象和多少个 **List** 对象等特征
3. 有一定的 **java** 代码阅读能力

MAT 的使用技巧，主要有

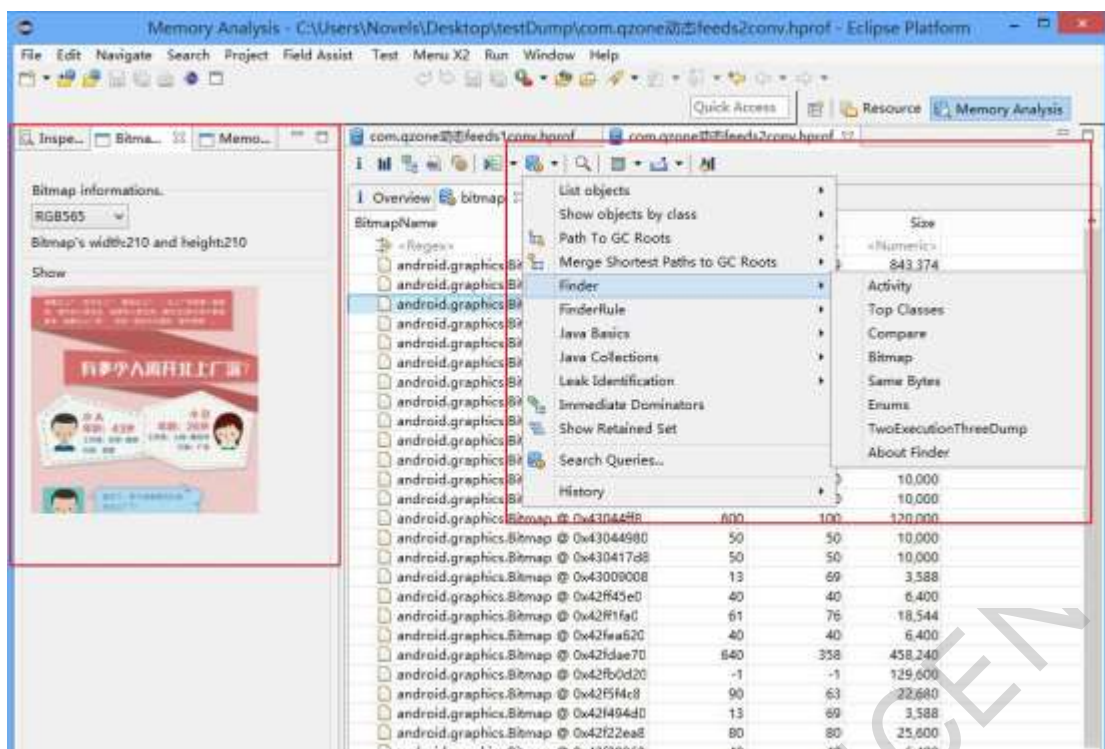


MAT 非常灵活且强大，但使用门槛颇高，不易于上手，因为是开源软件，所以有些信息并非做的足够详细、易用，比如统治者视图，虽然它是 **MAT** 的重要组件，但是里面信息做过大量过滤，如果过度依赖它来发现问题，很可能出现漏测的悲剧。

更多有关 **MAT** 的使用，因为其功能过多就不在这里一一述说了，会在案例篇中有更加生动，详尽的描述。

FINDER

MAT 足够强大，但是对于初级或者需要大量内存覆盖测试的测试人员来说，其强大的功能、复杂的操作、适配 **android** 内存测试时的小误差无疑构成了一场噩梦。**Finder** 是 **MAT** 的插件，由腾讯出品，主要是给 **Android** 内存测试人员提供一个系统化，简单化，流程化的内存测试体验，降低测试门槛，提高测试效率，稳定测试成果。



功能点

右边 Finder 菜单栏下扩展区菜单是主要操作区，有如下功能

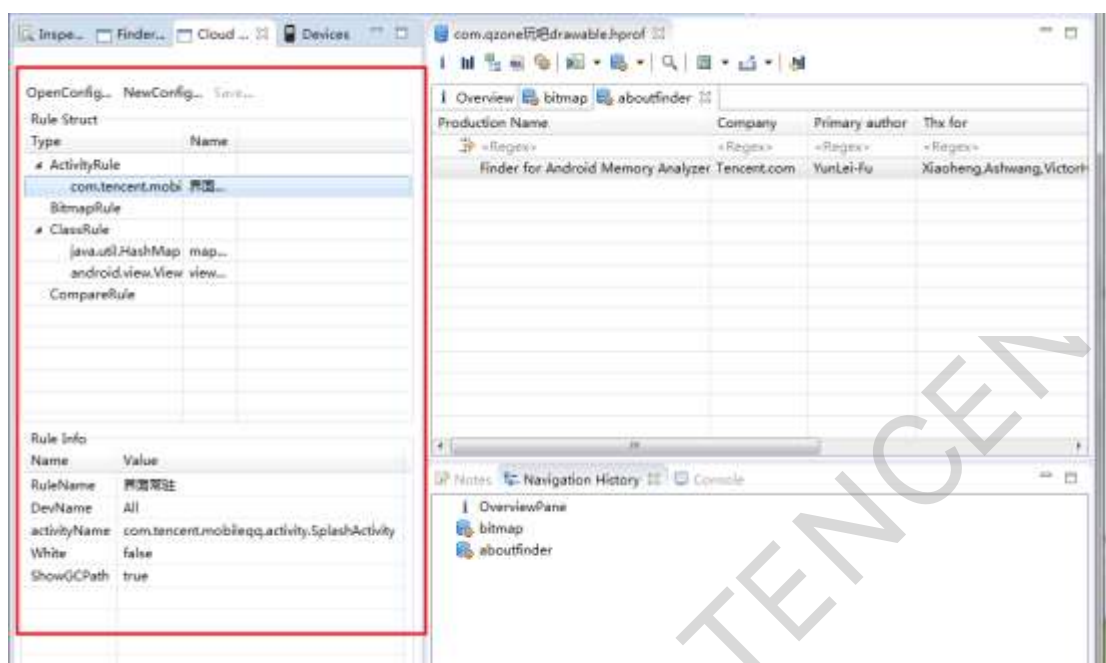
- Activity: 获取 dump 中所有 Activity 对象
- Top Classes: 以对象数量或对象大小为维度来获取对象降序列表
- Compare: 对比两个 Hprof 文件内容的差别
- Bitmap: 获取 dump 中所有的 bitmap 对象
- Same Bytes: 查询 dump 中以 byte[] 类型出现，并且内容重复的对象
- TwoExecutionThreeDump: “两遍三 dump”用以分析三个 dump 中持续增长的对象
- Singleton: 查询 dump 中的单例
- About Finder: 作者和感谢人

1.2.2 右边 FinderRule 菜单栏下扩展区菜单是

Run Memory rules 执行内存规则，主要是配合【1.2.4 Memory Rule】

1.2.3 左边功能区展示的是 Finder 中一个很好用的功能：

BitmapView: 通过 eclipse 的 Windows-Show View 中的 Other 里，FinderView(Bitmap View)呼出，用以查看某个 bitmap 对象中的图像。



1.2.4 左边功能区展示的是 Finder 中另一个很好用的功能，Memory Rule

Memory Rule: 通过 eclipse 的 Windows-Show View 中的 Other 里，FinderView(Memory rules View)，它用来积累用例和类对象规则，比如在登录成功后，内存中应该存在多少 map 对象。

1.3 使用效果

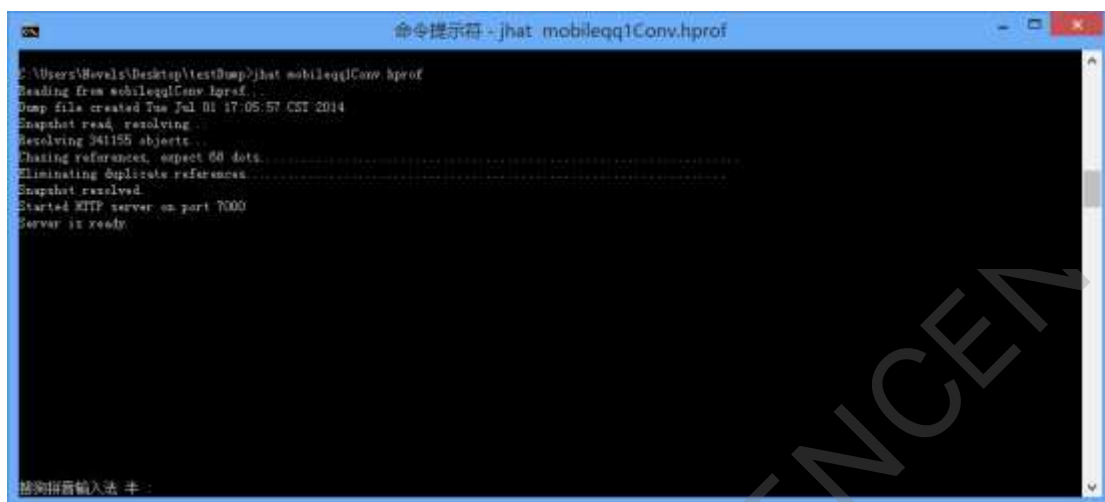
1.能够准确定位泄漏问题（通过 finder 定位的泄漏缺陷修改率 80%以上）

2.更够发现细微内存问题（通过曲线工具发现问题置后，且不敏感，常常出现小的泄漏测不出来，但是在用户那里就频繁爆发）

3.更加节省测试成本（以前内存测试，一个测试人员需要经过数月的培训，大量的了解产品架构，才能开始内存测试工作，且测试效率基本上为 2 工作日一个需求，使用 finder 后可以达到经过 3-5 小时培训，在不了解产品架构情况下，就可以开始内存测试工作，且半天就可以测试完成一个需求）

Jhat 是 Oracle 推出的一款 hprof 分析软件，它和 MAT 并列为 JAVA 内存静态分析利器，但是两者最初认知不同，MAT 更注重单人界面式分析，而 JHat 起初就认为 JAVA 的内存分析是联合多人协作的过程，所以使用多人界面式分析（BS 结构）。因为是 Oracle 推出的，也就是传统意义上的“正统”，所以 jhat 被置于 JDK 中，安装了 JDK 的读者，设置好 JAVA_HOME 与 Path 后，在命令行下如入 jhat 看看有没有相应的命令吧。

Jhat 的使用如下：

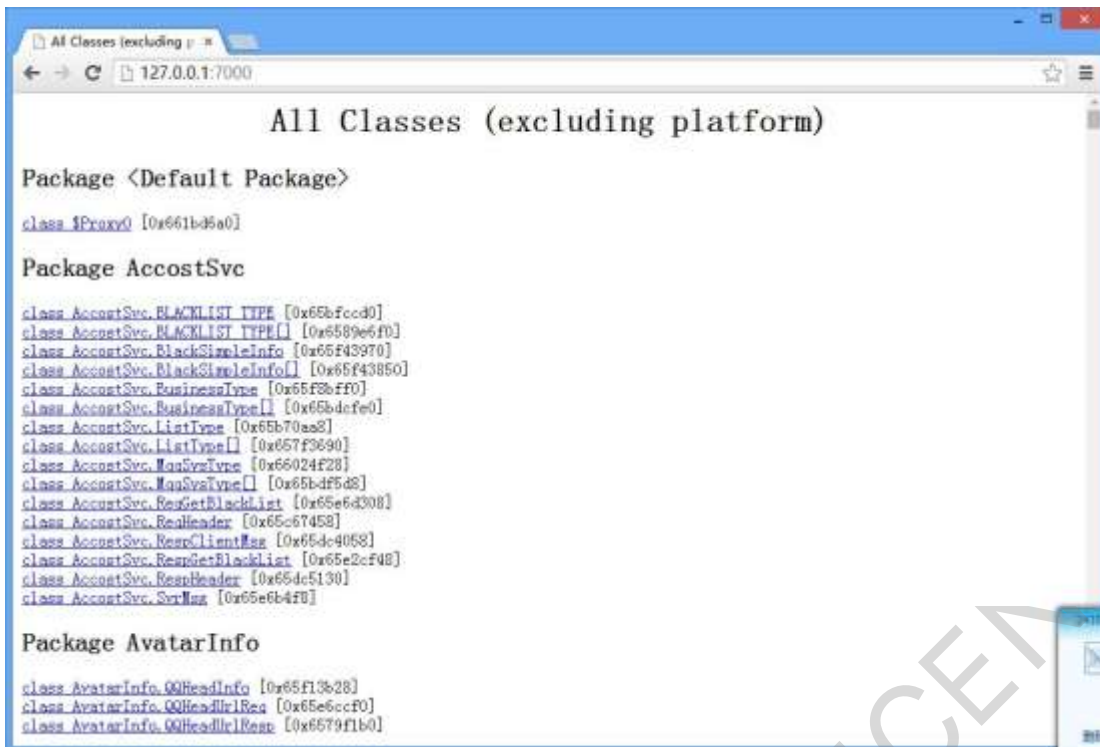


正常的使用非常简单：

jhat xxx.hprof

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。

执行结果：

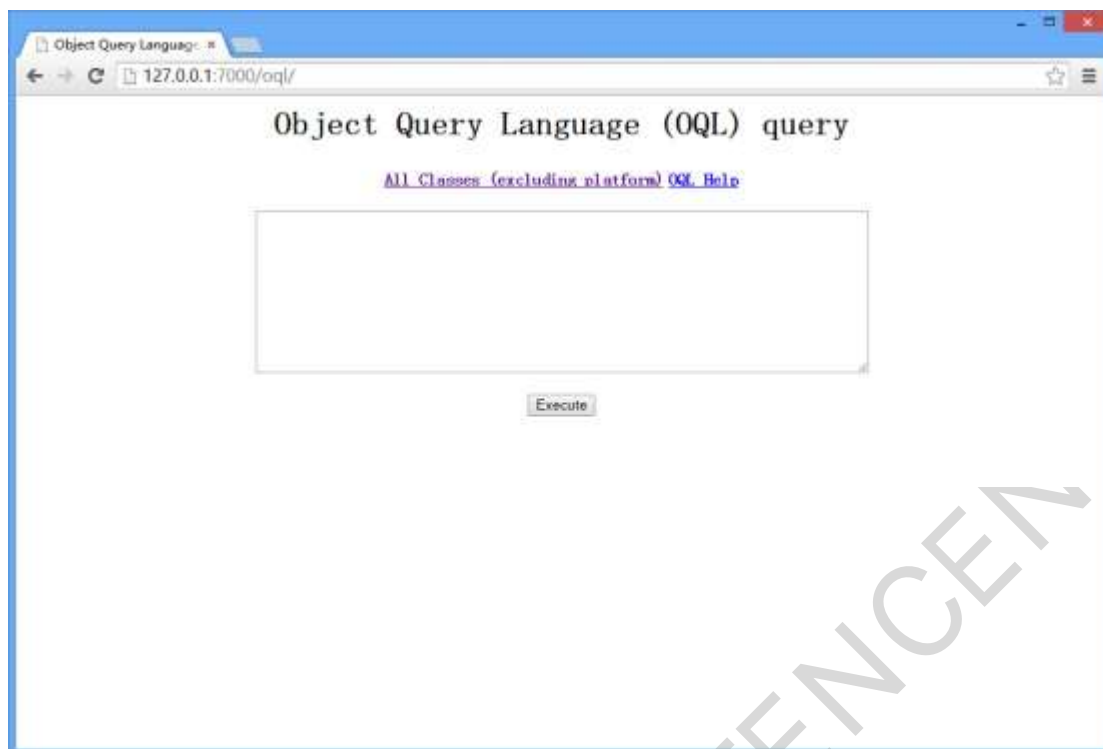


还有两个比较重要的功能：

1.统计表

Class	Instance Count	Total Size
class byte[]	3796	23230021
class java.lang.reflect.ArtMethod	69801	5584080
class char[]	65318	4187472
class java.lang.reflect.ArtMethod[]	18888	2391464
class java.lang.String[]	2141	1716856
class java.lang.String	65038	1560912
class java.lang.reflect.ArtField	52513	1260312
class java.lang.reflect.ArtField[]	6134	1021800
class java.lang.Class	6076	631904
class java.lang.Object[]	3987	340328
class int[]	4102	184300
class java.util.HashMap\$HashMapEntry	6325	151800
class java.lang.Class[]	65	150308
class java.util.HashMap\$HashMapEntry[]	453	73840
class java.lang.Integer	5809	69708
class long[]	262	61880
class java.lang.ref.FinalizerReference	1567	56412
class android.widget.TextView	64	43776
class short[]	12	43246
class java.util.LinkedHashMap\$LinkedEntry	948	30336

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，或者同源类数量）



建议对于中小型团队就不要考虑 Jhat 工具了，jhat 比 mat 更加灵活，且符合大型团队安装简易，团队协作的需求，并不是非常符合中小型高效沟通型团队使用。

STRICTMODE

前面的节点说的都是需要人工介入的内存测试，而 StrictMode 则是 Android 所特有的“规则桩”，桩的概念可以简单的认为，这种测试需要在产品中增加测试代码，以提高产品的可测性，这种做法即通俗的“打桩、下桩”，而被增加的测试代码即“桩”。

桩，按照用途可以分为日志桩、规则桩、监听桩，上报桩等等。而 StrictMode 是由 google 官方开发，并推荐 app 使用，用来提高运行效率的规则桩。它的规则主要分为两类：

- 1.虚拟机规则
- 2.线程规则

这两个规则的作用域不同，虚拟机规则，是针对单个运行时态的 App 而言的；而线程规则则是针对单个运行时态 APP 中的某个线程而言。简单的说，下放线程规则要知道目标线程的上下代码，最好能在产品的统一线程管理器中下桩，启动一个线程前，就立刻调用 strictMode 下桩。而虚拟机规则就不同了，在那里下，都不会影响其发挥作用。

StrictMode 的规则主要有如下：

主规则	子规则
线程规则	调用过慢
	硬盘读取
	硬盘写入
	网络访问
虚拟机规则	Activity 泄漏
	File 型 URI 暴露
	Closeable 类型未关闭
	广播监听器或服务状态监听未关闭
	SQL 游标或 SQL 对象未关闭
	设置某种类实例数量上限

其中虚拟机规则除了 File 型 URI 牵扯安全，权限问题外，其余均牵扯到泄漏问题。产品可以在发给测试的调试版打开 `strictMode`，实时监控 App 在运行态下的泄漏情况。开启参考如下代码：

开启线程规则：

```
StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder()
    .detectAll()
    .penaltyLog()
    .build();
StrictMode.setThreadPolicy(policy);
```

开启虚拟机规则：

```
StrictMode.VmPolicy policy = new StrictMode.VmPolicy.Builder()
    .detectAll()
    .penaltyLog()
    .build();
StrictMode.setVmPolicy(policy);
```

如果 APP 产生违规，会在 `logcat` 中以“`StrictMode`”为 tag 打印出违规信息。当然这里也需要注意一个特殊的情况，`android` 在回收某些无根对象或者弱引用对象时需要时间，而 `StrictMode` 会毫不留情的

把这个过程中出现的违规也打印出来，而这部分其实不是泄漏。因此读者在享受它带来的方便同时，请小心它带来麻烦也不少。

LIBC_MALLOC_DEBUG_LEAK.SO

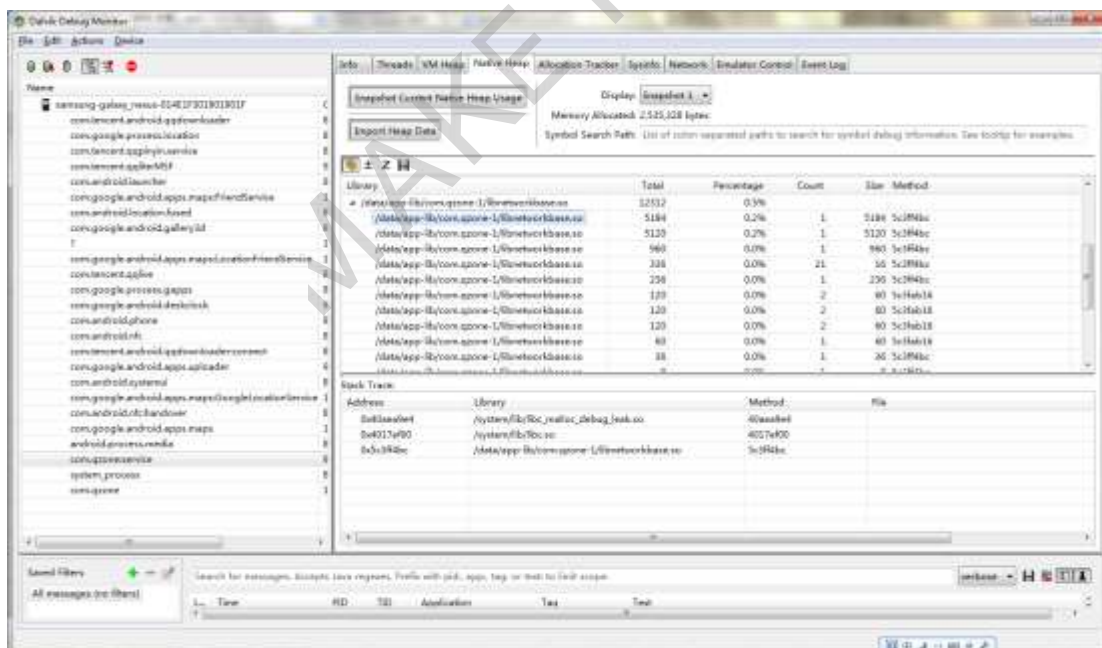
Android 是构建在一个被精简的 Linux 上，Dalvik 虚拟机是一个 Java 的运行时环境，而 Dalvik 本身其实是在 Linux 上运行的，和所有别的 Java 运行时环境一样，Dalvik 为了能够叫更多的 C/C++ 人员能够融入，提供了 NDK 以便能够叫 C 类开发者开发 Android APP。官方虽然提供了工具，但是在工具介绍的一开始，就严厉的说：并非所有的 app 都需要使用 NDK 技术，而且 NDK 技术并不会带来通常猜想的那些性能优势。它仅仅应该在两种情况下被使用，一、你有大量的 C++ 库要被复用 二、你编写的程序是高 CPU 负载的，比如游戏引擎，物理模仿等。

虽然官方明确限制了 **NDK** 的作用，但是依然有不少的产品使用了混合架构，即有一部分功能由 **JAVA** 编写，另外的使用 **NDK**。**NDK** 使用的内存是透传出 **Dalvik** 的，因此在 **Hprof** 分析过程中，是见不到这部分内存分配的。前面介绍的分析级别工具在 **NDK** 的面前都是没有作用的。为了要检测 **NDK** 所编写的 **C** 代码在运行时耗费的内存，我们就必须用到一个特殊的工具，或者你可以称它为库--

libc_malloc_debug_leak.so。

Android 底层 linux 申请内存所用到的库是 `libc.so`，而 `libc_malloc_debug_leak` 就是专门来监视 `libc.so` 内部接口（`malloc`、`calloc` 等）被调用的调试库。把 `libc_malloc_debug_leak.so` 放到 `libc.so` 的旁边，并且设置 Android 框架的 `libc.debug.malloc` 属性为“1”，然后重启 Android 框架，就打开了 Android C 类内存申请监控的开关。

界面展示:



在独立版 DDMS 中是可以展示 Native Heap 的，这部分就是 C 类内存申请，通常用 NDK 编出的程序，需要以 so 形式出现，并在安装后放入系统的/data 目录，这和系统本身的 so 是有区别的，所以

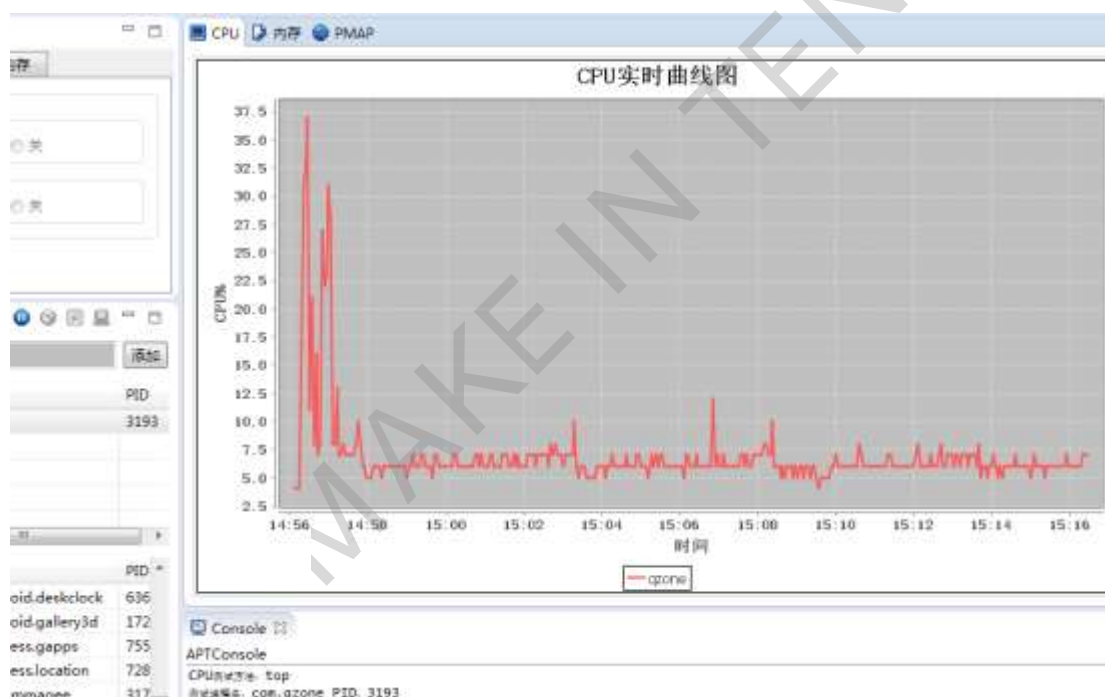
我们只用看这部分内存的大小，就是 **Size** 字段，点击每个申请，都拥有一个申请过程的调用栈，根据栈后的 **method** 字段，就可以知道方法所在的内存偏移，最后使用 NDK 自带的 **addr2line.exe**，就可以将地址转化为方法名称了。

NDK 在 Android 下并非是一种推荐的编程方式，但是由于其内存自管理与 CPU 高负载支撑特性，却俘获了很多开发者的心灵，Adnroid 对于这块的内存测试方案并没有多少新意，仅仅能够做到的，就是看看分配大小，看看申请此大小的方法是谁而已，如此直白就没有必要再过多叙述了，不过需要注意的是，在使用 NDK 的时候，一定要把 **Application.mk** 文件中加入编译选项“**-Wl,-Map=xxx.map -g**”，否则会引起无法使用 **addr2line** 的麻烦。

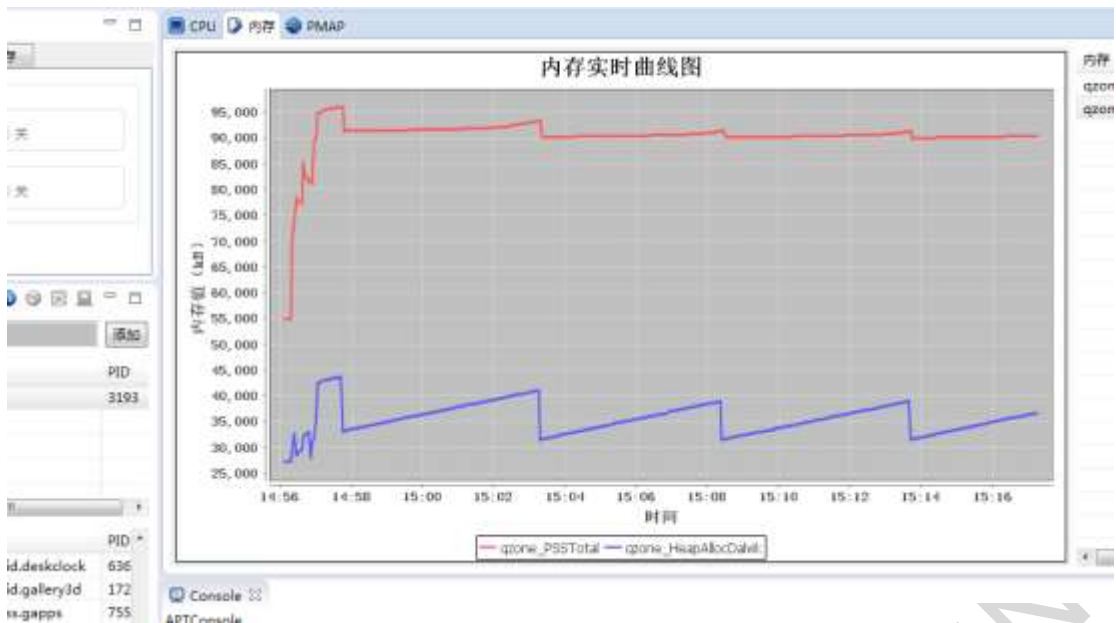
APT

APT 是腾讯出品的另外一款 Android 应用测试工具，与 Finder 同样作为 IDE 的插件形式出现，不同的是 Finder 是 MAT 的插件，而 APT 是 DDMS 的插件（DDMS 有独占问题，所以使用 APT 的机器上不能正常使用 DDMS），APT 实现的是实时监控能力，它可以监控多个 APP 的 CPU，内存指标，并且把它们画成图标形式，很直观。

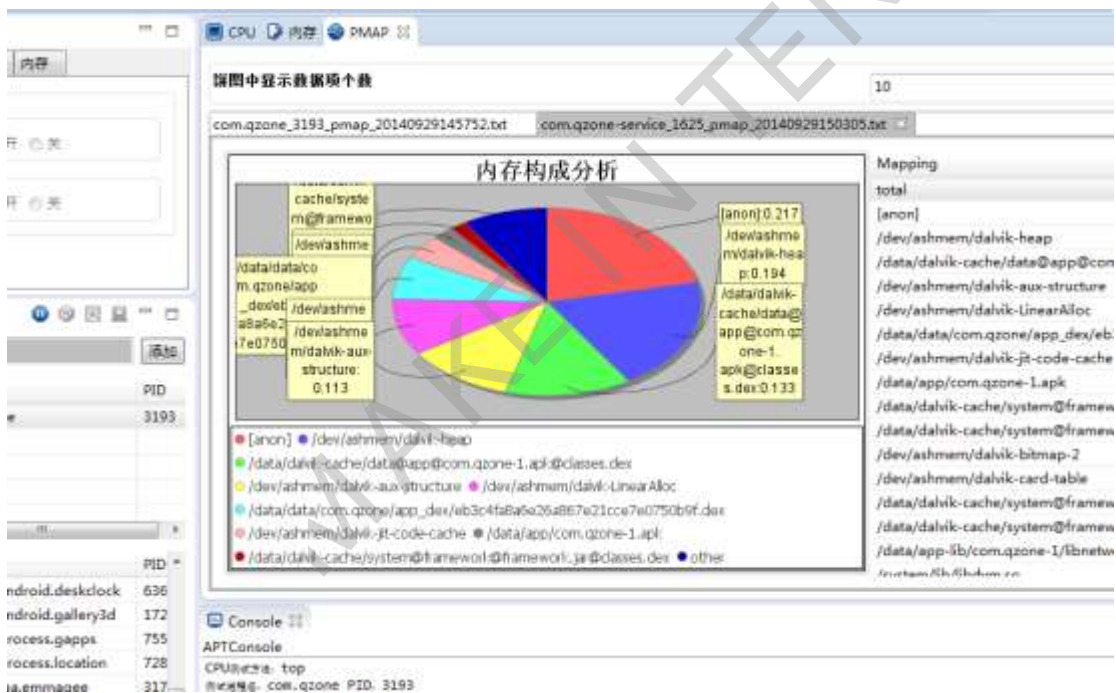
CPU 曲线展示：



内存曲线展示：



SMap 展示



注：因为 APT 实现的内存监控是调用 meminfo 分析其结果来实现，而 meminfo 每次调用会增加 dalvik 的 HeapAlloc 值（即使 app 什么事情也没有做），但是这并不影响 app 本身功能，到一定值 HeapAlloc 会释放这部分 meminfo 调用造成增加值，所以用 APT 监控 heapAlloc 很有可能会出现上面展示图的特征。

SMap 从一定程度上可以反映 native 的内存分配量，但是毕竟没有监控 malloc、calloc、delete 来的直接、准确，得到的数据也混入了大量代码段占据内存空间，而这部分内存空间是 APP 编码所必备的或者说很难减少，很难精简定位的，所以也不是很建议使用 SMap 来做 native 内存测试。

小结

通过工具篇，我们知道了多种可以用于 Android 应用程序内存测试的工具与使用方法。在思维中也建立起了 Android 内存测试的主要方向，对于 Android 应用，官方建议尽量少用 NDK，因此 JAVA 代码占据了 Android 应用的主要部分，而运行 JAVA 的 dalvik 虚拟机内存指标就变得非常重要，分析 dalvik 内存主要靠静态分析，这里面我们介绍了 Meminfo、MAT、Jhat 和 Finder。其次如果产品中包含了 NDK 成分，那么就要检查 native，这里我们介绍了 libc_malloc_debug_leak 和 APT-smap。使用工具并非是最重要的，工具因为专注的面不同，因此或多或少会存在某些局限性，但是知道了 Android 应用的内存测试方向，灵活的选择工具，就能够把内存测试变成一件并不那么困难的事情。

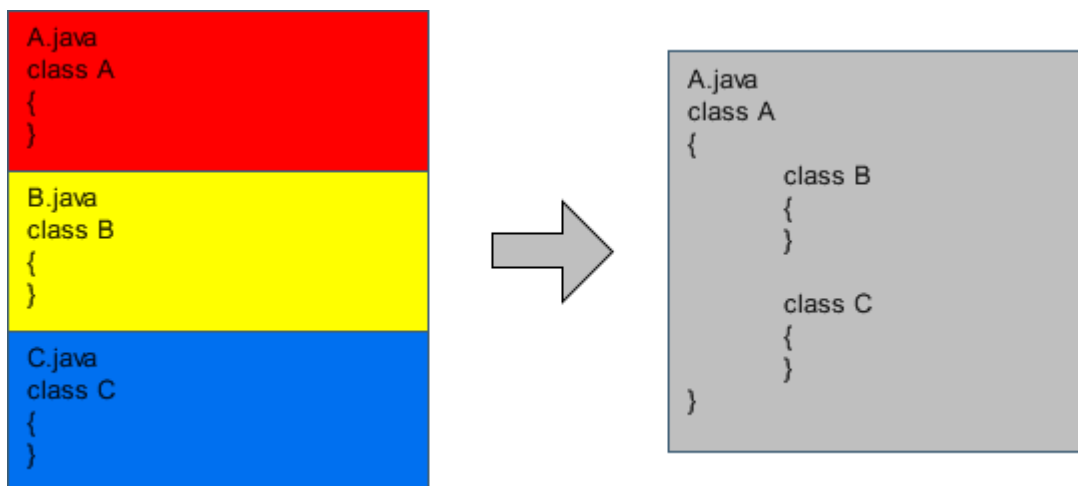
工具	覆盖范围	定位深度	推荐
曲线图	大	低	
TOP	大	低	
Meminfo	大	中	🌸
DDMS	大	中	
MAT	中	高	🌸
Finder	中	高	🌸
JHat 节点	中	中	
StrictMode	小	高	
APT	中	中	

测试案例是测试手法最好的说明与测试方案最好的验证，门槛低，收益高，易于分享和深挖，是测试团队成长的最好经验食粮。从本节开始用一个个案例来向您呈现腾讯做产品内存测试的方法、方案与 bug 经历。作者在数百个 bug 中抽样出十几例具有代表性的 bug，力争覆盖一个社交软件的主要构成部分，为读者在开发、测试软件提供一些比较全面的内存注意事项。

案例：内类是有危险的编码方式

问题类型：InnerClass

内类是一种对 java 语言的特有说法，通常情况下一个 java 的类必须占据整个与之同名的“.java”，但是也可以在一个类的内部去定义别的类，只要保证最外层的类与 java 文件同名，编译器就不会报错。这种方法省去了创建多个类就要建立多个 java 文件的麻烦。但每个好处的背后都一定潜伏着一个麻烦。



上面说的是内类在文件个数方面的好处，其实还有一个好处，内类对外类即图中的 B 或 C 对 A 的成员是具有直接访问能力的，也就是说比如 A 中有个成员 m1，那么在 B 中是直接可以访问或修改 m1 的，试想如果没有这个特征，那么开发者要在 B 中访问 A 的成员应该怎么做？先要实例化一个 A，然后通过实例访问，或者建立 A 的单例然后再访问，还牵扯到各种的访问权限问题，简直是太麻烦了。

但就是因为第二个优点的出现，给开发者创造方便的同时，却给测试员带来了不小的麻烦。说到内类就不得不提到“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了。引用关系是 java 内存泄漏的根本，所以如果开发者只图一时方便，而不清楚这一原理的话，很容易就造成内存问题。

接下来我们看下手 Q 的 bug，来更加具体的了解一下 this\$0。QQ 为了能够时刻保持用户的多终端情况下消息通知及时，曾经做了一个需求：QQ 时刻在线。

测试步骤：

1. 开启手 Q 的“PC QQ 离线时自动启动手 Q”功能
2. 手 Q 登录 A 帐号，然后下线
3. PC-QQ 登录 A 帐号并下线
4. 手 Q 上 A 帐号自动上线

使用 finder 查看 activity 列表，发现有泄漏，XxxPCActivityActivity 泄漏

android2958274355998597619.hprof

Overview finderactivity path2gc [selection of 'NotifyPCActiveActivity @ 0x430a38b8'] -excludes java

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.tencent.mobileqq. [redacted] nActivity @ 0x430aa098	344	2,224
com.tencent.mobileqq. [redacted] ntManageActivity @ 0x437a4480	496	1,952
com.tencent.mobileqq. [redacted] PCActiveActivity @ 0x430a38b8	280	17,392
com.tencent.mobileqq. [redacted] ttingSettingActivity @ 0x432453e8	384	896
com.tencent.mobileqq. [redacted] cationActivity @ 0x43221310	304	696
Σ Total: 5 entries		

分析:

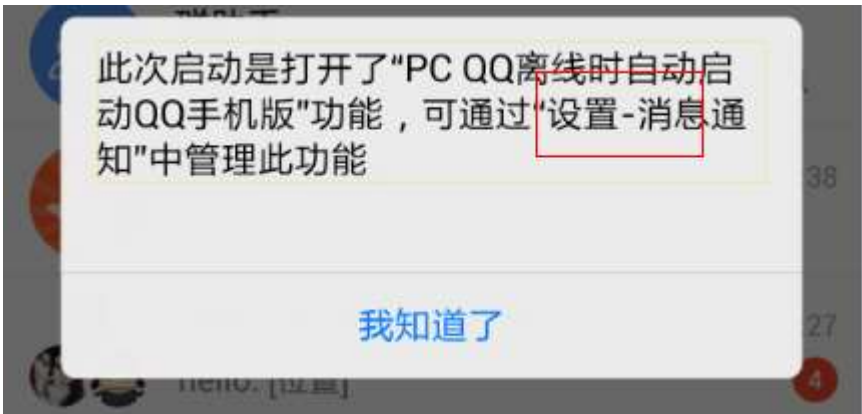
Overview finderactivity path2gc [selection of 'NotifyPCActiveActivity @ 0x430a38b8'] -excludes java.lang.ref.R

Status: Found 30 paths so far.

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.tencent.mobileqq. [redacted] PCActiveActivity @ 0x430a38b8	280	17,392
this\$0 com.tencent.mobileqq. [redacted] PCActiveActivity\$4 @ 0x430a3a6f	16	16
[0] java.lang.Object[20] @ 0x43819368	96	96
elementData java.util.Vector @ 0x437f4b90	24	120
bgObservers com.tencent. [redacted] nterface @ 0x43	456	212,728
app class com.tencent. [redacted] @ 0x4305f258	168	4,400
<Java Local> java.lang. [redacted] MSF-Receiver	80	1,992
mAppRuntime com.te [redacted] eApplicationImp	112	1,256
this\$0 com.tencent.mol [redacted] face\$1 @ 0x430	16	16
app, mRuntime com.te [redacted] .SplashActivity @	344	2,224
app, mRuntime com.te [redacted] .NotificationActi	304	696

查看引用路径不难发现，有一个熟悉的 this\$0 出现在第二行，在第二行末尾有一个奇怪\$4。这行的正确解读是：XxxPCActiveActivity 内部包含了一个匿名内类（4 在一般情况下是说第四个匿名内类），通过 this\$0 引用了 XxxPCActiveActivity 实例。所谓匿名内类的意思就是直接 new 出来的接口，抽象类等等，它并没有按照标准的继承，实现来创建一个新类，然后实例化，而是采用实现、实例化一起的做法，当然它也是内类的范畴。

这个匿名内类的用途并不难理解，主要对网络层接收到 PC 下线通知后做出反应。但是恰恰开发同学在关闭窗口的时候，忘掉了将内类在网络层反注册掉，因此通过内类引用外类原则，此时的外类就和内类一起泄漏了。



解决方法：

在窗口关闭时，反注册匿名内类。

案例：使用统一界面绘制服务的内存问题

问题类型：Activity 驻留

Activity 是 Android 交互接口的基础单位，可以简单的理解成“窗口”。如果需要更新窗口中的某种状态，就需要得到窗口的 Context。下面将要展示 bug，就是源于支付服务直接把 Activity 本身缓存下来，以便将来更新界面。

Bug 也是发生在手 Q 支付需求

操作步骤如下：

- 1. 通过个人设置，进入手 Q 钱包，然后返回
- 2. 将第一步操作录制成 monkeyrunner 脚本，持续操作 50 次
- 3. 抓取内存 hprof，分析
- 4. 使用 Finder-Activity 发现支付界面泄漏

Class Name	Shallow Heap	Retained Heap
com.pay.ui.[REDACTED]ListNumActivity @ 0x453c92e0	288	159,656
com.pay.ui.[REDACTED]Activity @ 0x449789d0	288	4,408
com.pay.ui.[REDACTED]Activity @ 0x4293cbf8	288	3,608

支付中心的activity泄漏

Class Name	Shallow Heap	Retained Heap
com.pay.ui.XXXChannelActivity @ 0x449789d0	288	4,408
- value java.util.HashMap\$HashMapEntry @ 0x449eb560	24	24
- [2] java.util.HashMap\$HashMapEntry[4] @ 0x41e5fd48	32	66,840
- table java.util.HashMap @ 0x41dd2b90	48	66,904
- b class com.pay.ui.XXXCommonMethod @ 0x41dd2ab8 System Class	16	67,064
- mOuterContext android.app.ContextImpl @ 0x449851e8	104	728
- Total: 2 entries		

从引用路径上很容易就可以看出 XXXChannelActivity 是被 XXXCommonMethod hold 住了。为什么要这样设计呢？原因很简单，支付渠道的普通方法中抽象了类似“转菊花”（腾讯的 T 族把 Loading 图标都叫菊花，因为最早的 loading 图标通常被做成像菊花的模样）之类的概念接口，而 XXXChannelActivity 只用把自己（this）作为参数传入，并调用这个普通方法，就可以在自己的界面上绘制出一张“转菊花”的动态 loading。这样 XXXCommonMethod 就变身成通用的绘制服务，能够在多种 Activity 上绘制“转菊花”。



但因为 XXXCommonMethod 要把 UIN 传给财付通服务器后等待其返回数据，不能立刻在传入的 Activity 上绘制，所以就要把传入的 Activity 缓存下来，等服务器返回数据后，才能绘制。这样就对传入的 Activity 有了强引用，而针对 XXXCommonMethod 的编码者而言，恰恰忘掉了把缓存的 Activity 在用完之后移除，也就造成了这个“低级”的泄漏。

注：其实缓存 Activity 风险极高，因为 Activity 本身有状态，当一个 destroyed 状态的 Activity 因为缓存存在强引用而无法被垃圾回收器回收时，这个 Activity 中的界面是不能给更新的，一旦执行更新操作程序就会异常。

解决方法:

XXXCommonMethod 渲染完成后移除 Activity。

案例：结构化消息点击通知产生的内存问题

问题类型：Context 泄露

聊天窗口中有一种消息，并非为用户与用户之间发送，而是一种结构化的分享消息，它有如下特点：

1. 有固定的排布，并非用户打字构成
2. 通过分享等渠道产生数据源
3. 可以点击，呼起被分享模块



因为有如上特点，开发在设计结构化消息的时候，实现了一个模块间的结构化消息通知服务，这样有助于统一实现从聊天窗口通知被分享模块。但这里产生了一个内存问题：

操作步骤：

1. 打开聊天窗口，点击分享消息
2. 进入被分享模块，如地图后，回退
3. 退出聊天窗口
4. 抓取内存 hprof，查看 Finder-Activity

5. 发现聊天窗口泄漏

聊天窗口引用路径如下：

Class Name	Shallow Heap	Retained Heap
com.tencent.mob[REDACTED]ChatActivity @ 0x42df55d0	608	6,992
- mContext com.tencent.mobileqq.structmsg[REDACTED]ClickHandler @ 0x420e1e00	24	24
- mSourceClickHandler com.tencent.mob[REDACTED]ShareMsg @ 0x4264a360	152	440
- structingMsg com.tencent.mob[REDACTED]ageForStructing @ 0x42743ab0	168	168
- [0] java.lang.Object[12] @ 0x4263b5b8	64	64

分析：

从引用路径上可知，聊天窗口被结构化消息内部的消息 hold 住。分析原因如下：在结构化消息内部会统一处理从聊天窗口传入的用户点击操作，为了使接口通用，于是设计为传入参数为聊天窗口的 context，这样可以最大限度的访问聊天窗口中的界面元素，以得到当前被点击结构消息的信息。将获取来的结构体消息存储以备提供给被分享模块做启动参数。

但是在从聊天窗口获取完信息并包装成结构体消息后，这个 context 应该没有用了，但不知开发是否处于方便扩展原因，便把这个 context 存入了包装后的结构体消息内部，待用，这样就造成了这个 bug。

解决方法：

包装后的结构体消息不存储 context。

案例：为了不卡，所以可能泄漏

问题类型：Context 泄露

界面开启速度对于用户体验来说比较重要，专项有个专门的名词用来描述它，叫做“时延”。所谓“时延”大或者“卡”范畴非常广泛，下面就简单分下类（如果有兴趣的读者可以转去“时延篇”专门阅读这块的知识）：

1. cpu 耗尽型，运算量很大，视音频解码场景容易出现，此类问题通常只能优化算法已达到降“卡”
2. 耗时型，运算量不是很大，起码不会达到 cpu 满载，耗时比较巨大，比如网络访问，硬盘读写等，此类问题通常把“耗时”部分投放到界面线程以外的线程处理，已达到降“卡”。当然“耗时”有时也发生在线程互锁或切换方面，这样的“卡”就不一定能够靠投放任务开辟新线程来解决。
3. 内存耗尽型，内存耗尽型和 cpu 耗尽型差不多，都是 cpu 满载，因为 android 在物理内存吃紧时候，会大量的 gc，甚至会大量的杀进程，而紧接着又会启动一些必要进程（比如电话簿，

短信等等)，而这种杀了起，起了杀，gc 了申请，申请了又 gc，会吃掉大量的 CPU，一般出现在某个进程有较大的内存需求，或使用了 JNI 技术来申请 native 内存块以规避 Android 框架对进程内存最大值限制的情况，此类问题只能尽可能降低内存损耗。

我们要讲的是 2 类型“卡”，使用投放耗时任务达到降卡目的的场景，以及应该注意的内存问题点。投放任务的方式无非有如下几种：

1. 界面自己创建一个线程来处理“耗时”任务
2. 有计算型服务，正好可以处理这类“耗时”型任务，能把任务直接丢给它

计算型服务的问题，我们在服务篇里面已经介绍过了，最要注意的是，正确调用服务对外产生引用（服务有可能会缓存 callback 或者 context 之类的对象）的接口。而自己创建线程来处理“耗时”，则是我们接下来要着重介绍的，一样我们通过一个 bug 来展示。

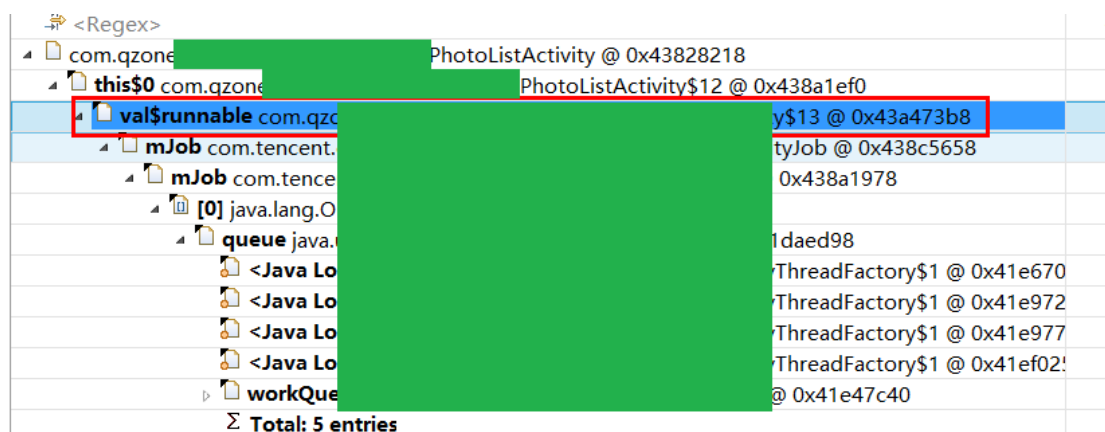
这个 bug 来自于 QQ 空间 Android 独立版，众所周知相册在空间的业务中属于核心级别的业务之一，而相册内容基本都是照片，这些照片的加载需要巨大耗时，如果不采用异步线程处理，会影响相册的启动速度。当然这一点腾讯的开发是非常注意的，所以专门编写了一种叫做异步 imageView 的类来做这件事情，而这个类也相对稳定，我们的 bug 并非出现在它。

图片加载完成后，并不代表完成相册的所有任务。相册还有一些子业务，比如一个叫做“圈人”的业务，它是一种人脸识别技术的社交化应用。如果某张照片中存在“人脸”，那么会提示当前操作用户去把这张“脸”对应的好友给“圈”出名字来。这样方便正在浏览我相册的朋友去认识我的别的朋友，或者是我的照片中存在一个陌生人，但是浏览相册的朋友恰恰知道这个人，于是他可以帮我把人脸对应上名字。

对于这个需求，相册不但要加载照片，还要加载对应照片的人脸信息（人脸识别是在云端计算的），而恰恰是这一步出了一点问题。

操作步骤：

1. 进入相册，查看照片列表
2. 浏览某张包含人脸的照片，查看人脸列表
3. 退出人脸列表
4. 退出相册
5. 抓取内存快照 hprof
6. 使用 finder-activity 查看界面泄漏情况，发现 xxxPhotoListActivity 界面泄漏
7. 查看引用关系



分析:

Bug 造成的原因非常简单，开发在界面的内部使用了一个继承于 `Runnable` 的内类来实例化一个任务（工作），并将其投递给了自定义线程工厂，但是线程工厂由于任务优先级的设计，并没有及时的启动这个任务。而后测试人员在未等任务启动并完成的情况下，就退出了相册列表，抓取了内存快照，这样就有了以上的 bug 单。

从技术的角度分析就更加简单，因为 `xxxPhotoListActivity` 的第 12 个匿名内类持有了 `Activity` 的 `Context`（`this$0`），而这个匿名内类恰恰被一个线程池内未被执行的任务持有，于是 `xxxPhotoListActivity` 在执行回退操作后，依然无法被有效回收。

解决方法:

这里的解决方法可以分成两类，对应不同的情况

1.按照官方推荐的方法，线程都应该具有一个开关来决定它自己是否继续执行，通常由一个线程互斥的边界变量来控制。而位于线程池内的任务，更应该在设置互斥边界值后将其从线程池内移除（有兴趣的读者可以在官网的《在线程池里运行代码》查阅相关内容）。

2.不去停止线程，而是从界面层（引用链的顶端）规避引用关系，要做的方法非常简单，使用 `ApplicationContext` 来替换 `ActivityContext`。

第 2 种方法的好处在于，不会把做到一半的工作停下来，比如从云端拉取了一半的图片停下来代价是挺大的。但如果任务的状态需要到界面反馈，比如下载进度，这种实现就有点麻烦，需要任务内部具备接口来获取此任务的状态。在某些地方保存对任务实例的引用，在需要知道任务状态的时候，使用这个引用来访问任务接口获取执行状态。但这这就要求编码者特别注意对这个引用的释放，否则会把界面泄漏变成任务对象泄漏。

在这里还要着重提出一种比较“好”的方法，大体的实现代码如下：

need to invoke the outer activity's methods from within the `Handler`, have the `Handler` hold a `WeakReference` to the activity so you don't accidentally leak a context. To fix the memory leak that occurs when we instantiate the anonymous `Runnable` class, we make the variable a static field of the class (since static instances of anonymous classes do not hold an implicit reference to their outer class):

```
1 public class SampleActivity extends Activity {
2
3     /**
4      * Instances of static inner classes do not hold an implicit
5      * reference to their outer class.
6      */
7     private static class MyHandler extends Handler {
8         private final WeakReference<SampleActivity> mActivity;
9
10        public MyHandler(SampleActivity activity) {
11            mActivity = new WeakReference<SampleActivity>(activity);
12        }
13
14        @Override
15        public void handleMessage(Message msg) {
16            SampleActivity activity = mActivity.get();
17            if (activity != null) {
18                // ...
19            }
20        }
21    }
22
23    private final MyHandler mHandler = new MyHandler(this);
24}
```

作者解决界面泄漏的着眼点在内类的“this\$0 引用”（这和我们上面介绍的 `bug` 类似），他变了个戏法，下面我们仔细看看他是怎么做到的：

1. 把内类申明成 `static`，来断绝 `this$0` 的引用。因为 `static` 描述的内类从 `Java` 编译原理上看，“内类”与“外类”相互独立，互相都没有访问对方成员变量能力。
2. 使用 `WeakReference` 来引用外部类实例。弱引用并不产生无法 `GC` 的强制引用，所以垃圾回收器并不关心它，即当被弱引用指向的对象生命周期还未结束时，通过弱引用可以得到被指向对象，但是如果被指向的对象生命周期已结束后（所有强引用都释放了对这个对象持有），这类弱引用就只能返回空对象。这样在 `activity` 被用户返回前并不影响内类对 `activity` 的操作，但是在 `activity` 执行返回后，又不会影响 `activity` 本身的回收。

这种方法为什么被列为“好”，原因很简单，它在最大程度上简化了修改这种内存 `bug` 的代价，但是有可能会在某些情况下引发功能 `bug`，比如没有对 `WeakReference` 判空造成功能 `crash`，或者虽然判空了，但没有把 `else` 逻辑加完善，造成二次打开异常等等。

案例：图片缓存的内存问题

问题类型：Context 泄露

图片缓冲的内存问题在前面图片缓冲章已经介绍了很多，但都集中在其自身产生的问题，这里要讲的是服务和外界发生作用时产生的 `bug`。

图片缓冲一般除了存储位图能力外，还具备 **Bitmap** 对象加载功能，也就是具备图片解码功能（在之前的图片缓冲中有介绍，比如从硬盘上的 **jpg** 文件加载，或网络流加载）。

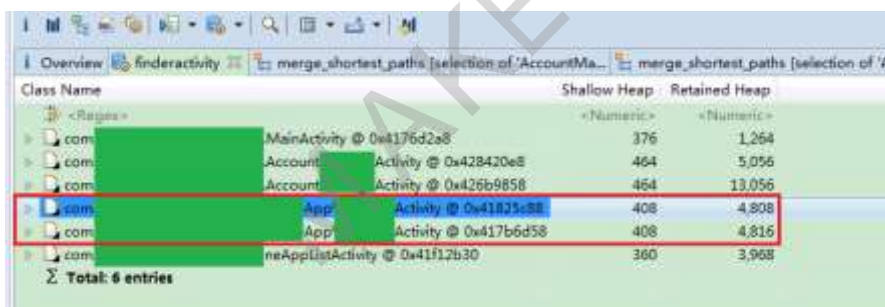
入参可以是本地路径或者是安装包中的某个图片资源 ID（图片是可以直接当作资源编译在安装包内的，被编译入安装包的资源都会有一个相对安装包而言唯一的 ID），这两种情况，图片缓冲可以同步的返回解码后的图片对象给外界，因为这个时延是可控的。

但是，如果入参是远端 **URL**，那么图片缓冲是不可能同步的返回一个图片对象的。因为时延是不可控的，其中多了一个下载图片过程，而不可控因素远远多余本地读取文件。所以图片缓冲在获取图片接口的设计上，就会有两套，一套为本地，一套为远端。远端接口是异步的，除了给 **URL** 参数外还要有回调函数。当缓冲服务下载完图片流后解码成对象，再把图片对象回调给外界。

下面介绍的这个 **bug**，就发生使用图片缓冲获取远端图片的场景下。

操作步骤：

1. 将手机网络置于 **2G** 或 **wifi** 限速状态（弱网环境）
2. 从 **QQ** 动态 **tab**，进入游戏中心，应用中心，查看游戏或应用详情
3. 返回 **QQ** 动态 **tab**
4. 重复 2、3 操作数次
5. 抓取 **hprof** 文件
6. 使用 **Finder-Activity** 功能，发现游戏中心 **Activity** 泄漏

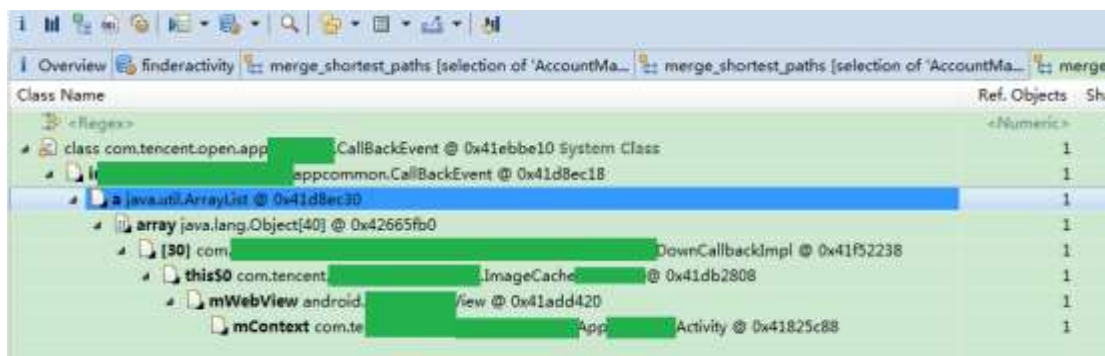


Class Name	Shallow Heap	Retained Heap
<Root>	<Numeric>	<Numeric>
com.tencent.activity.MainActivity @ 0x4176d2a8	376	1,264
com.tencent.activity.AccountActivity @ 0x428420e8	464	5,056
com.tencent.activity.AccountActivity @ 0x426b9858	464	13,056
com.tencent.activity.AppActivity @ 0x41825c88	408	4,808
com.tencent.activity.AppActivity @ 0x417b6d58	408	4,816
com.tencent.activity.AppListActivity @ 0x41f12b30	360	3,968
Total: 6 entries		

Finder-Activity 运行结果

Bug 类型：内存泄漏

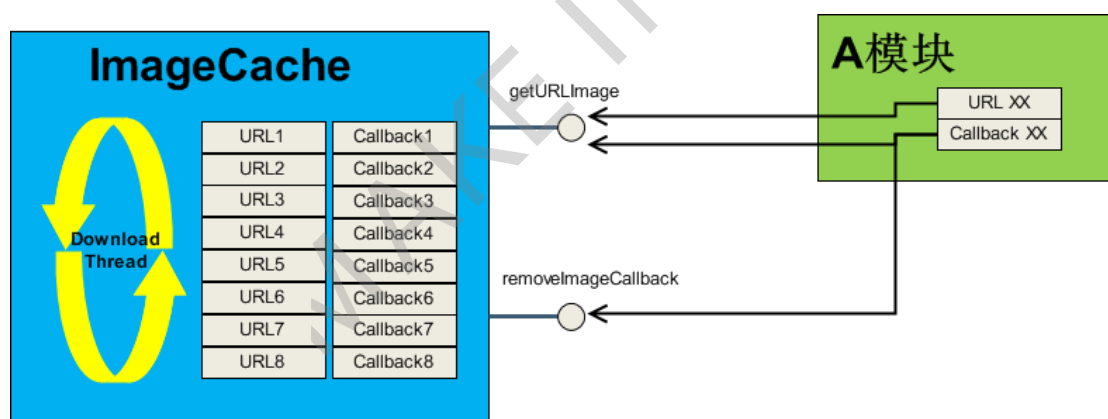
要分析为何会产生泄漏，还需要看界面对象的引用路径，通过 **MAT** 的 **Merge shortest GC Path** 功能。



从引用路径上看，是一个 callback 被 ImageCacheXXX（图片缓冲）持有了，Callback 本身又通过 View 持有了 Activity 的 context，于是就有了这次泄漏。从引用路径上可以直观的看到问题就是 callback，但是具体是什么原因造成 callback 常驻与缓冲中呢。

原因很简单，就是远端图片加弱网络问题。在图片缓冲的远端图片下载接口的使用存在缺陷，下载图片完成后，需要接口调用方主动指明删除那个回调。而弱网络造成图片迟迟下载不回来，然后多次重复启动页面，又给缓冲下达了很多重复的图片下载任务。调用方却只保存最后一个回调函数，以便将来传给图片的回调移除接口。

于是等多个图片下载完成后，虽然调用方多次调用了图片缓存服务移除回调，但是传入参数总是最后的那个，所以图片缓冲服务也只会移除一个回调，其余的回调就被长久的存储在缓冲服务里了，也就造就了这个 bug。



解决方法：

A 模块把 callback 记录权交给 Activity，在 Activity 的 onDestory 回调函数中执行 removeImageCallback，图片缓存中做判空容错。

案例：登录界面有内存问题么

问题类型：Activity 驻留

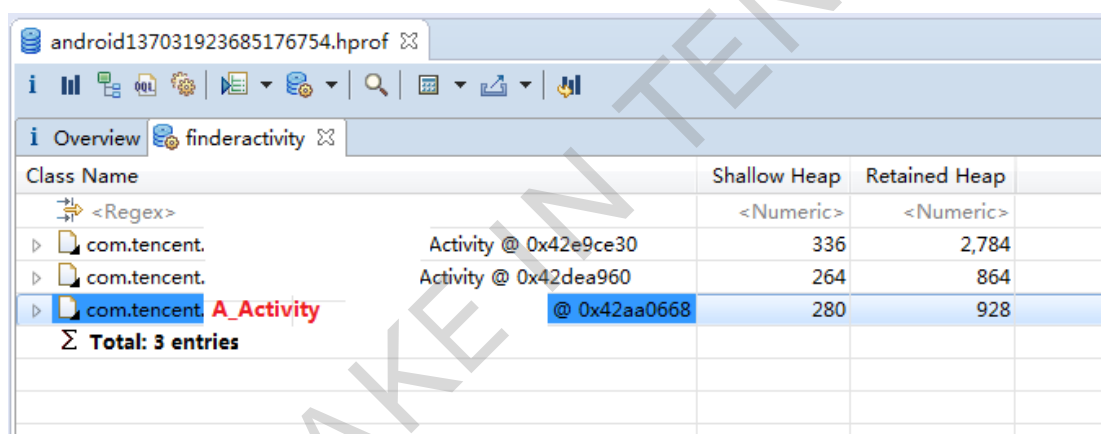
背景：

登录界面通常是一个互联网软件最先见到的界面之一，当然有的软件也会做运营“闪屏”。在阅读下文之前，读者可以想想登录界面有什么独特之处呢？我们在这个 bug 的总结处来告知这个答案。

操作步骤：

腾讯有款新的互联网产品上线前需要做内存检测，拿到产品的测试同学首先对软件的登录界面 A_Activity 做了内存检测，遗憾的发现这个登录界面是泄漏的。测试步骤是如下做的：

- 使用帐号登录软件
- 在软件登录成功后，使用 DDMS 抓取 hprof
- 用 MAT-Finder 的 activity 功能来查看当前内存中的 Activity

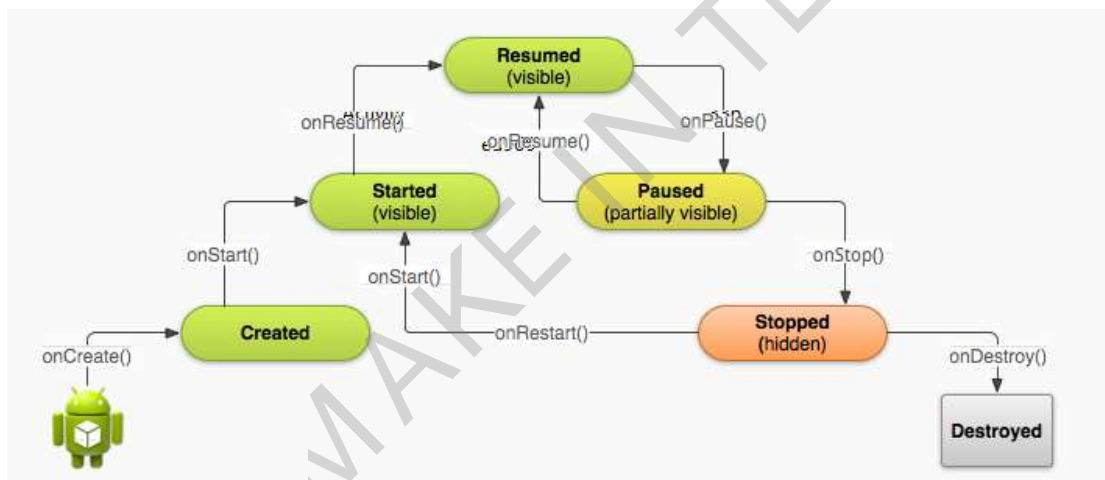


Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.tencent. Activity @ 0x42e9ce30	336	2,784
com.tencent. Activity @ 0x42dea960	264	864
com.tencent. A_Activity @ 0x42aa0668	280	928
Σ Total: 3 entries		

- 发现 hprof 中存在 A_Activity 对象，查看引用路径

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.tencent.A_Activity @ 0x42aa0668	280	928
context1 android.view.GestureDetector @ 0x42ae5c68	104	2,592
mGesture android.view.ViewRootImpl @ 0x42afa880	496	4,632
this\$0 android.view.ViewRootImpl\$WindowInputEventReceiver @ 0x42e32730 Native Stack	40	192
this\$0 android.view.ViewRootImpl\$AccessibilityInteractionConnectionManager @ 0x42ab0408	16	16
Σ Total: 2 entries		
[0] java.lang.Object[12] @ 0x42afd1f8	64	64
array java.util.ArrayList @ 0x42afd1e0	24	88
observers com.pusha @ 0x42afd1c8	16	104
a class com.pusha @ 0x42afd108 System Class	8	152
mContext com.android.internal.policy.impl.PhoneWindow\$DecorView @ 0x42e0ddc0	656	1,944
[0] android.view.View[2] @ 0x42ab9bc0	24	24
mViews android.view.WindowManagerGlobal @ 0x42aa8028	32	112
sDefaultWindowManager class android.view.WindowManagerGlobal @ 0x42053118 System Class	96	640
this\$0 android.view.WindowManagerGlobal\$1 @ 0x42de1d30	16	16
mGlobal android.view.WindowManagerImpl @ 0x42df18e8	24	24
Σ Total: 3 entries		
activity android.app.ActivityThread\$ActivityClientRecord @ 0x42af6250	112	576
value java.util.HashMap\$HashMapEntry @ 0x42dfaad0	24	600
next java.util.HashMap\$HashMapEntry @ 0x42e0eec0	24	760
[3] java.util.HashMap\$HashMapEntry[4] @ 0x42dd9dd0	32	792
table java.util.HashMap @ 0x42a9b7a0	48	840
mActivities android.app.ActivityThread @ 0x42a9b660	192	4,792

登录成功后会进入产品主界面 B_Activity，而且并不存在一种逻辑能够叫用户再从 B_Activity 退回到 A_Activity（逻辑回退），所以按照“所见即所得”的测试基本思路，A_Activity 应该从内存中释放掉，但是在上述 bug 中 A_Activity 在用户进入 B_Activity 后依然存在，所以测试人员定义这个现象为“泄漏”。什么原因造成这个泄漏呢？首先我们要简单的看下一个 Activity 的生命周期：



上图是在介绍 Activity 的生命周期时比较著名的一副。官方叫它“生命周期金字塔”，在塔尖的状态是可见且可交互状态，其余的状态都是部分可见、不可见、或者不可交互状态。那么就一个完整的 Activity 生命周期而言，至于状态是如何迁移，这个读者可以自己去体会或者查阅相关文档，状态如何迁移不是这个 bug 的主要构成原因，就不再这里发散了。但是我们要记住一个原则，“每个 activity 在不做特殊处理的时候，都必须拥有这么一个完全的生命周期状态节点，才能保证它最终释放，缺少或停留某些状态都会造成 activity 无法消亡，进而造成内存泄漏”。

在最后一个 Destroyed 状态是指：用户点击了回退键，或者显式调用了 Activity 的 finish() 方法，否则被闪一下然后退到后台的 activity。如果仅仅退回有台的 activity，只会进入 stopped 状态。没有走到 Destroyed 状态的 activity 系统是不会回收它的。

而前面我们说了登录界面进入主界面后会退到后台（自动退到后台或者说被主界面盖住，并非用户手动点击回退键），也就是登录界面进入了 `stopped` 状态。而 `stopped` 状态的界面并不会被系统回收，这样它就泄漏了。

那么，我们应该怎么才能把 `A_Activity` 从 `Resumed` 状态转换到 `Destoryed` 呢？官方是有这方面的建议的，原文如下：

Note: The system calls `onDestroy()` after it has already called `onPause()` and `onStop()` in all situations except one: when you call `finish()` from within the `onCreate()` method. In some cases, such as when your activity operates as a temporary decision maker to launch another activity, you might call `finish()` from within `onCreate()` to destroy the activity. In this case, the system immediately calls `onDestroy()` without calling any of the other lifecycle methods.

原来官方早就意识到了，可能存在这么一种“过渡界面”，而这种“过渡”界面应该在 `onCreate` 方法中主动的调用 `finish` 方法，来完成 `activity` 的状态转变，使它能够被系统回收。

总结：通过对这个案例的学习，我们知道了 `Android` 的界面基础类“`Activity`”生命周期和状态迁移的一些知识，了解到存在一种特殊的内存泄漏，它是由状态变更不到位而引起，常常出现在“过渡”界面场景下，“过渡”界面由于其缺失回退操作，而无法完成整个 `Activity` 的所有状态，进而会造成内存泄漏。因此，当测试人员再碰到类似有“过渡”界面的产品时，一定要注意这种内存 `bug` 是否存在。

案例：使用 `WIFIMANAGER` 的内存问题

问题类型：Context 泄露

下面要介绍的这个 `bug` 发生在手 `Q` 需要获取网络状态时。手 `Q` 很多的业务都需要测试当时的网络状态，会用到底层服务组件，比如 `wifiManager` 等。根据网络的不同状态可以优化用户体验，比如在 `wifi` 下用户一般不会非常关心流量，就可以给用户展示更加清晰的图片或者展示视频等等。



操作步骤:

1. 使用 New Monkey 执行 2500 下压力点击
2. 等压力点击结束后，手动回退到主界面
3. 抓取内存 hprof，使用 Finder-Activity 分析界面泄漏
4. 发现界面泄漏

com.ten	PluginActivity @ 0x4540b1d8	584	99,696
com.ten	PluginActivity @ 0x44a69e90	584	99,696
com.ten	PluginActivity @ 0x449df728	584	93,328
com.ten	PluginActivity @ 0x42d65628	584	99,696
com.ten	PluginActivity @ 0x42bc6998	584	99,696
com.ten	PluginActivity @ 0x429bb118	584	93,328
com.ten	PluginActivity @ 0x425b2e50	584	99,696
com.ten	PluginActivity @ 0x41f13798	584	99,696
com.ten	PluginActivity @ 0x41e56a78	584	99,696
com.ten	PluginActivity @ 0x41d9bf98	584	99,696
com.ten	PluginActivity @ 0x41d4baa0	584	99,696
com.ten	PluginActivity @ 0x41c188e8	584	93,328
com.ten	PluginActivity @ 0x41c03448	584	99,696
com.ten	PluginActivity @ 0x41c003f0	584	99,696
com.ten	PluginActivity @ 0x4195fbb8	584	99,696

i Overview finderactivity path2gc [selection of 'TenpayPluginActivity @ 0x449df728'] -excludes java.lang.ref.Reference:		
Status: Found 1 paths. No more paths left.		
Class Name	Shallow Heap	Retain
<Regex>	<Numeric>	
com.ten[REDACTED]PluginActivity @ 0x449df728	584	
mContext android.net.wifi.WifiManager @ 0x44c0cbe0	48	
this\$0 android.net.wifi.WifiManager\$ServiceHandler @ 0x44cecf20	32	
this\$0 android.os.Handler\$MessengerImpl @ 0x44cecf48 Native Stack	24	

从引用路径来看都是 Activity 的 Context 被 WifiManager 持有所造成，开发的代码编写方式很简单：

```

WifiManager mWifiM = getSystemService(Context.WIFI_SERVICE);

if(mWifiM.isWifiEnable())
{
.....// wifi 逻辑
}

else
{
.....// 非 wifi 逻辑
}

```

注：这个 bug 是非必现的，所以要使用 MTTF 这样的压力测试才能保证出现率。

标红的这段代码，如果在 Activity 中调用，那么会默认把 Activity 的 Context 传给 WifiManager 服务，在某些不确定情况下，WifiManager 内部会产生异常，从而会 hold 住外界传入的 Context。

解决办法：

把 `getSystemService(Context.WIFI_SERVICE);`修改为以下代码：

```
getApplicationContext().getSystemService(Context.WIFI_SERVICE);
```

同样的情况也发生在 AudioManager 等服务上，比如要判断当前是耳机模式或者外放模式，一样会产生这样的问题，所以都应该使用 `getApplicationContext().getSystemService` 来获取服务实例。

服务节点总结：

对于服务来说它们的生命周期一般是跟随 app 的完整生命周期，所以它如果对外有引用，按照 java 的生命周期延长法则，这些外部对象也都会被延长生命周期，进而产生内存泄漏。解决服务对外持有的方法可以总结为几点：

1. 有清除引用逻辑
2. 在使用系统服务的时候尽量避免界面的 Context
3. 提供异步工作的服务，一定要注意回调函数，handle，observe 等通知类型对象的注册与反注册成对出现

案例：产品图片缓存服务合格么

问题类型：图片类问题

俗说：互联网产品最讲究什么体验精神？有图有真相。

通过上一节我们已经可以成功登录被测产品了，如果被测产品是一个互联网产品，那么这个时候，它就要开始落入上面那个“俗套”里了。这个时候它会开足马力，从服务器或者云端，拉取互联网数据了。

互联网是个超大的资料库，媒体库。图片，视频，音频犹如一辆辆满载的卡车，在互联网的高速通道上疯狂的你追我赶。这些大块头，带来的不仅仅是“交通拥挤”还有“停车风险”。我们可以简单的把带宽拥挤理解为“交通拥挤”，那么“停车风险”，就是怎么把这些下载回来的媒体，存储或缓存下来了。

这一节所要展示的 bug，都是由这些大块头造成的，如果你测试的产品也拥有“有图有真相”的体验精神，那么这节对于你测试内存来说相当必要。先看看“停车风险”都有哪些吧。

首当其冲的是 crash：

Crash堆栈

[查看还原前堆栈](#)

```
- java.lang.OutOfMemoryError: bitmap size exceeds VM budget
- android.graphics.BitmapFactory.nativeDecodeAsset(Native Method)
- android.graphics.BitmapFactory.decodeStream(BitmapFactory.java:460)
- android.graphics.BitmapFactory.decodeResourceStream(BitmapFactory.java:336)
- android.graphics.drawable.Drawable.createFromResourceStream(Drawable.java:697)
- android.content.res.Resources.loadDrawable(Resources.java:1709)
- android.content.res.Resources.getDrawable(Resources.java:581)
- com.tencent.████████████████████.Drawable
  getDrawable(int) (ProGuard:71)
- android.graphics.drawable.StateListDrawable.inflate(StateListDrawable.java:162)
- android.graphics.drawable.Drawable.createFromXmlInner(Drawable.java:787)
- android.graphics.drawable.Drawable.createFromXml(Drawable.java:728)
- android.content.res.Resources.loadDrawable(Resources.java:1694)
- android.content.res.Resources.getDrawable(Resources.java:581)
- com.tencent.████████████████████.Drawable
  getDrawable(int) (ProGuard:73)
- com.tencent.████████████████████.$Item
  Add(int, java.lang.CharSequence, int) (ProGuard:126)
- com.qzone.████████████████████ void add(int, java.lang.CharSequence)
  (ProGuard:100)
- com.qzone.████████████████████PictureViewer.void createMore(████████ Menu())
  (ProGuard:899)
- com.qzone.████████████████████Viewer.void init() (ProGuard:415)
- com.qzone.ui.████████████████████PictureViewer.void onCreate(android.os.Bundle)
  (ProGuard:342)
- android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
- android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1630)
```

（加载图片 crash，出师不利）

然后是“大兴调度”，疯狂 GC，“卡”：

03-21 11:27:33.008: D/dalvikvm(2493): GC_FOR_ALLOC freed 4062K, 12% free 31476K/35580K, paused 26ms, total 26ms
03-21 11:27:33.038: D/dalvikvm(2493): GC_FOR_ALLOC freed 8K, 11% free 33698K/37812K, paused 21ms, total 21ms
03-21 11:27:33.068: D/dalvikvm(2493): GC_FOR_ALLOC freed 80K, 11% free 33705K/37812K, paused 28ms, total 28ms
03-21 11:27:33.098: D/dalvikvm(2493): GC_FOR_ALLOC freed <1K, 11% free 35935K/40044K, paused 21ms, total 21ms
03-21 11:27:33.168: D/dalvikvm(2493): GC_FOR_ALLOC freed 4915K, 21% free 33788K/42276K, paused 20ms, total 20ms
03-21 11:27:33.228: D/dalvikvm(2493): GC_FOR_ALLOC freed 6859K, 12% free 31589K/35576K, paused 26ms, total 26ms
03-21 11:27:33.258: D/dalvikvm(2493): GC_FOR_ALLOC freed <1K, 11% free 33818K/37808K, paused 22ms, total 22ms
03-21 11:27:33.298: D/dalvikvm(2493): GC_FOR_ALLOC freed 20K, 11% free 33886K/37808K, paused 31ms, total 31ms
03-21 11:27:33.318: D/dalvikvm(2493): GC_FOR_ALLOC freed <1K, 10% free 36115K/40040K, paused 23ms, total 23ms
03-21 11:27:33.388: D/dalvikvm(2493): GC_FOR_ALLOC freed 4585K, 18% free 34874K/42328K, paused 24ms, total 24ms
03-21 11:27:33.458: D/dalvikvm(2493): GC_FOR_ALLOC freed 4593K, 18% free 34895K/42328K, paused 21ms, total 21ms
03-21 11:27:33.528: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 34917K/42328K, paused 21ms, total 21ms
03-21 11:27:33.588: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 34941K/42328K, paused 22ms, total 22ms
03-21 11:27:33.648: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 34977K/42328K, paused 20ms, total 20ms
03-21 11:27:33.718: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35002K/42328K, paused 31ms, total 31ms
03-21 11:27:33.788: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35024K/42328K, paused 20ms, total 20ms
03-21 11:27:33.848: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35048K/42328K, paused 20ms, total 20ms
03-21 11:27:33.908: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35073K/42328K, paused 20ms, total 20ms
03-21 11:27:33.968: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35098K/42328K, paused 28ms, total 28ms
03-21 11:27:34.038: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35131K/42328K, paused 25ms, total 25ms
03-21 11:27:34.118: D/dalvikvm(2493): GC_FOR_ALLOC freed 4634K, 18% free 35110K/42328K, paused 33ms, total 33ms
03-21 11:27:34.188: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 18% free 35130K/42328K, paused 33ms, total 33ms
03-21 11:27:34.248: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 17% free 35152K/42328K, paused 20ms, total 20ms
03-21 11:27:34.308: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 17% free 35177K/42328K, paused 20ms, total 20ms
03-21 11:27:34.378: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 17% free 35198K/42328K, paused 29ms, total 29ms
03-21 11:27:34.458: D/dalvikvm(2493): GC_FOR_ALLOC freed 4588K, 17% free 35223K/42328K, paused 20ms, total 21ms

（一秒钟疯狂申请内存，末日前的狂欢）

最后是功能异常，有损体验：



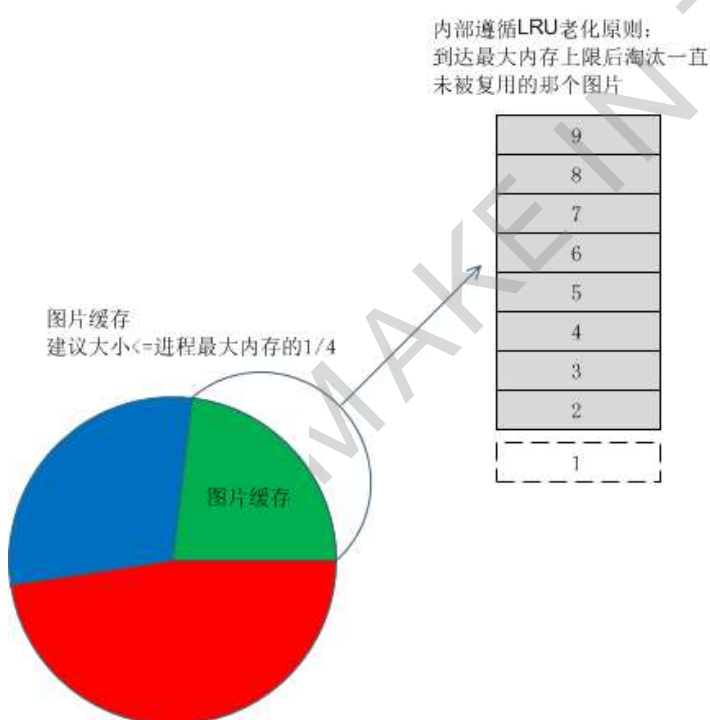
（内存没了，图还要加载）

当然，以上的损害都是在说，我们将“大卡车”停进了“内存”所造成的危害。既然有这么多的损害，为什么不能把图片下载回来都放到磁盘（SDCard）上呢？其实答案不难猜，放在内存中，展示起来会“快”那么一些。快的原因有如下两点：

- 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，而不是再做一次图片解码）

很多同学不知道所谓“解码”的概念，可以简单的理解，Android 系统要在屏幕上展示图片的时候只认“像素缓冲”，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，所以相对而言，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，是影响图片展示速度的一个重要因素。

两害相权，取其轻，官方建议应该适量的使用图片缓存概念。当然我们并非要在这里大讲图片缓存是怎么建立的，应该怎么实现，那就偏离本书的目的了（当然有兴趣的同学也可以去 <http://developer.android.com/training> 中看看 Building Apps with Graphics & Animation 章节），所以在这里我们只是简单提一下原理示意图：



官方建议使用一个进程所能申请的最大内存的四分之一作为图片缓存（Android 进程都有最大内存上限，这依据手机 Rom 而定，在手机出厂的那一刻就被固定下来了，一般情况下无法更改）。图片缓存达到容积上限时，内部使用 LRU 思维做淘汰，淘汰那些“又老又少被用到”的图片。这就是内存图

片缓存的大体设计思维，当然，在官方还有硬盘缓存建议，它是增加缓存容量，降低解码开销的设计思路，我们也是简单做下原理示意：



前面提到了两点：

1. 内存缓存会淘汰一些内存中解码好的图片
2. 图片的压缩比会影响解码速度，而解码速度是影响图片展示速度的主要原因之一

于是官方建议，把从内存被淘汰的图片，降低压缩比存储到本地，以备后用。这样就可以最大限度的降低以后复用时的解码开销。

接着，着重介绍一下使用图片缓存时，可能会产生的问题。

案例：关于图片解码配色设置的建议

问题类型：图片类问题

我觉得在 **Android** 内存专项中，“一张图，毁十优”，意思是说，一张图片的非合理常驻，会造成十次优化的结果都白费。当然说法有点夸张，但图片在 **Android** 进程运行时内存中，都是些块头健硕的大家伙。

简单的量化一下，一个简单的 **Activity** 界面泄漏，通常泄漏大小在十几 k 到 **1MB** 之间。但如果换做一个图片，造成的内存耗损却会达到几十 KB 甚至几十 MB，两者完全不是一个等量级。因此“防火防盗放图片”也是测试员理所当然的条件反射。

以下要讲述的是有关图片的“配色设置”的话题，它在一定程度上会非常影响图片内存大小。当然，我们一样先从内存 bug 说起。

这个 bug 发生在手机空间，空间在发布之前会做一次必须的“专项横评”。所谓“专项横评”就是把将要发布的版本，与已经发布过的，上一个版本做专项方面的对比。对比项包括内存，FPS，时延，CPU，耗电等一系列指标，这样做是为了保证将要发布的版本，在专项方面不低于已发布版。

操作步骤

1. 取 4.2 版本作为基线，取 4.5 即将发布版本作为被测目标

2. 测试登录后滑动 30 条 Feeds 后内存对比
3. 发现 4.5 滑动 30 条 feeds 比 4.2 滑动 30 条 feeds 消耗的内存（DalvikVM heapAlloc）要多
4. 使用 Finder-Compare 功能，分别查看 4.2 和 4.5 的内存对象新增
5. 发现位图的内存占用差别很大

4.5 30条feeds

Class&ObjectName	ChangedObjectsCount	NewCount	GCCount	CurrentCount	Incr Retained
com.tencent.compon [REDACTED] Bitmap [REDACTED]	40	40	0	47	16,539,640

4.2 30条feeds

Class&ObjectName	ChangedObjectsCount	NewCount	GCCount	CurrentCount	Incr Retained
com.tencent.compon [REDACTED] Bitmap [REDACTED]	45	45	0	51	6,642,624

Bug 类型：内存浪费

分析：

就新增对象个数分析，同是拉取 30 条 feeds，4.5 增加的图片个数比 4.2 少。但内存耗损增量却比 4.2 要大出不少，这是为什么？

测试员取出了其中一个图片查看 Finder bitmap view，发现 4.2 加载的图片与 4.5 加载的图片，拥有同样的尺寸，内容。但内存耗损 4.5 的却比 4.2 的翻了一翻。

取其中一张4.5图片

Class&ObjectName	ChangedObjectsCount	NewCount	GCCount	CurrentCount	Incr Retained
com.tencent.compon [REDACTED] Bitmap [REDACTED] @ 0x41d1bb28	1	1	0	1	2,910,936

和4.2图片

Class&ObjectName	ChangedObjectsCount	NewCount	GCCount	CurrentCount	Incr Retained
com.tencent.com [REDACTED] Bitmap [REDACTED] @ 0x419ddcd0	1	1	0	1	1,455,456

（内存大小对比）



(4.2 feed 图片)



(4.5 feed 图片)

从肉眼判断两张图完全一样，这下在“黑盒测试”方面已经无路可循，只能才用“白盒”方式一探究竟。跟踪开发的代码变更，根据差异化分析发现，开发在 4.2 版本上拉取 feeds 图片解码图片时，使用了 `inPreferredConfig` 参数。它是 `BitmapFactory.Options` 的一个字段，`BitmapFactory.Options` 可以被绝大多数 Android 图片解码 API 当作“设置”参数传入。

看下 `inPreferredConfig` 字段的含义是什么：

```
public Bitmap.Config inPreferredConfig; // If this is non-null, the decoder will try to decode into this internal configuration.
```

如果不为空的情况，解码器将会使用这个内部设置。

字段的官方说明比较晦涩，接着要看看它类型的官方说明，`inPreferredConfig` 的类型是 `Bitmap.Config`：

public static final enum Bitmap.Config extends Enum<E> extends Enum<E>			Summary Enums Methods Inherited Methods Expand All Added in API level 1
java.lang.Object ↳ java.lang.Enum<E> extends java.lang.Enum<E> ↳ android.graphics.Bitmap.Config			
Class Overview			
Possible bitmap configurations. A bitmap configuration describes how pixels are stored. This affects the quality (color depth) as well as the ability to display transparent/translucent colors.			
Summary			
Enum Values			
Bitmap.Config	ALPHA_8	Each pixel is stored as a single transluency (alpha) channel.	
Bitmap.Config	ARGB_4444	This field was deprecated in API level 13. Because of the poor quality of this configuration, it is advised to use <code>ARGB_8888</code> instead.	
Bitmap.Config	ARGB_8888	Each pixel is stored on 4 bytes.	
Bitmap.Config	RGB_565	Each pixel is stored on 2 bytes and only the RGB channels are encoded: red is stored with 5 bits of precision (32 possible values), green is stored with 6 bits of precision (64 possible values) and blue is stored with 5 bits of precision.	

Bitmap.Config 用来配置 bitmap 用怎样的像素格式来存储。这会影响质量（色深），也能影响显示半透明，透明颜色。

ALPHA_8：只有透明通道

ARGB_4444：质量太差，建议更换 ARGB_8888

ARGB_8888：每个像素四个字节

RGB_565：每个像素使用两个字节，只有 RGB 通道被解码：红色 5 位，绿色 6 位，蓝色 5 位

到这里就彻底明白了，看回代码发现原来 4.2 空间使用了 RGB_565 来解码 Feeds 图片。而 4.5 却没有使用 inPreferredConfig。

对于没有给定 BitmapFactory.Options 参数，而直接调用图片解码函数的情况，Android 系统会默认使用 ARGB_8888。透过官方文档，可以知道 RGB_565 每个像素占用的内存只有 ARGB_8888 的一半（RGB_565 每个像素使用两个字节，ARGB_8888 每个像素四个字节）。这样 bug 现象中 4.5 中看似相同的图片为什么比 4.2 中的大一倍的疑问就找到答案了。

但为什么 4.2 版本使用 RGB_565 来加载 feeds 图片，而 4.5 版本却弃用了呢？官网说不同的设置会影响图片的显示质量，和透明度。开发是否因为质量的原因刻意去掉 RGB_565 设置呢？带着问题，测试员询问了开发，开发首先解释了 RGB_565 是如何降低图片显示质量的。

用 5 位（0-32）来解析整个的红色空间，与使用 8 位（0-255）来解析整个红色空间相比，就像一个稀疏的栅栏和一个紧密的栅栏相比一样，前者粗糙许多。以下可以大概展示一下红色空间丰富的图片，在使用 RGB_565 与 ARGB_8888 产生的不同显示质量吧：

开发也承认，虽然 RGB_565 会粗糙一些，开发的确是因为在重构代码过程中漏了 RGB_565 的设置，这才造成了测试发现的内存差异。拉取 Feeds 操作，下载的其实都是“缩略图”，用户只想大概知道是什么内容，对质量并没有很高的要求，如果想要看高清大图，是可以通过点击操作进入浏览界面仔细查看的。



解决方案:

在 feeds 加载并解码图片的代码中, 增加 `BitmapFactory.Options.inPreferredConfig` 设置。

注意: 使用 `RGB_565` 的条件

缩略图, 用户感官上认为它本来就不应该是一张高清的图片, 如果需要高清查看, 有“点入”操作给到他们。

没有透明效果, 因为 `RGB_565` 不解码透明通道内容, 因此会造成透明效果丢失。

以此为基础, 测试人员也去手 Q 做了一下简单的排查, 也发现手 Q 也存在一种情况可以使用 `RGB_565`。虽然手 Q 没有大量的图片展示区, 这点不像空间, 但是手 Q 却存在各式各样的聊天背景, 这些背景图片尺寸巨大, 色泽单一, 恰恰也没有透明效果。而开发一直在琢磨怎么降低聊天时候出现的内存峰值。这下测试员娓娓道来这个既简单, 又实用的小技巧, 真可谓雪中送炭。

案例: 图片放错资源目录也会有内存问题

问题类型: 图片类问题

上面讲述的多数问题都是界面切换, 控件操作发生的内存问题。下面介绍的问题和交互操作没什么关系, 它们通常是因为开发没有明确设计而造成, 这不同于因为粗心丢了某些调用, 错过了某些设置。

首先, 打开一个 Android 的工程目录, 在 `res` 目录下面不难发现有很多以 `drawalbe` 开头的文件夹, 随便打开其中一个, 就会发现其中存储的是一些图片。和代码相结合阅读发现, 这些图片都是随 `apk` 发布, 将在某个场景下展示的“本地”图片。

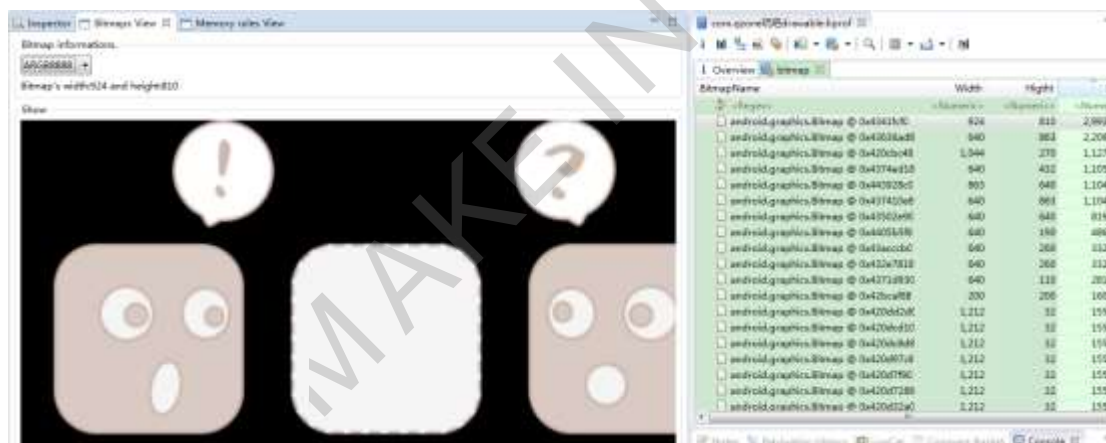
但 res 下有多个 drawable 目录，拿到一张设计给的图片，应该被放在那个目录下才合适呢（其实放在任何目录，都不会影响应用正常功能，但是在内存性能方面却有很大不同）？

我们一样通过一个 bug 来了解这个知识点：

Android 空间独立版有个叫做玩吧的业务。用户通过它可以边刷空间边玩游戏，比排名。在一次测试玩吧业务的合流测试过程中，但发现了一个奇怪的内存问题：

测试步骤：

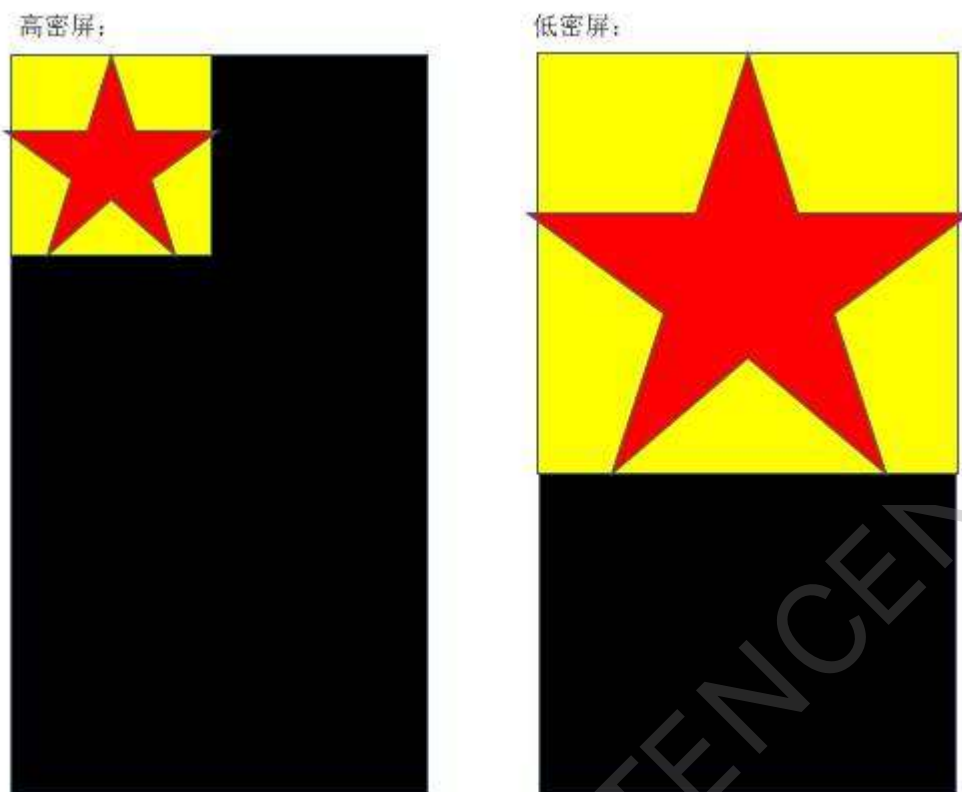
1. 安装 Qzone
2. 进入玩吧面板
3. 点击某个游戏，运行
4. 回退到玩吧界面，并抓取内存快照 hprof
5. 使用 finder-Btmap 查看内存中图片尺寸
6. 发现“未找到游戏”运营图片尺寸超过被测设备屏幕尺寸（检查图片超尺寸加载是合理必做的内存检测项之一）



被测设备是 Nexus3，分辨率 720*1280，但是这幅图片的尺寸是 924*810，宽度超过屏幕的分辨率尺寸 200 多个像素点，是“图片超尺寸加载”类型违规。所以当时的 bug 单中给开发的建议是，将图片缩小比例后加载入内存，避免没必要的内存浪费。但开发发现并没有用任何代码来加载这幅图片，这幅图片是通过 xml 布局元素，直接由系统加载的，于是开发就怀疑是系统问题。

难道真的是系统问题？Android 系统因为某些原因，在巨大的硬件环境差异下的确存在某些系统 bug。但是本着实事求是的原则，还是去考证一下。通过分析，叫人大跌眼镜的是，内存问题引出的却是一个屏幕密度的概念。

怎么理解屏幕密度呢？首先看下一下图片：



以上用来示意两个手机的屏幕，它们尺寸大小相同，但一个低密度，一个高密度。均在其屏幕上展示 500*500 的图片，情况就会变成上图这样。直观感受过现象后，我们来理解一下显示密度是什么：

Screen density

The quantity of pixels within a physical area of the screen; usually referred to as dpi (dots per inch). For example, a "low" density screen has fewer pixels within a given physical area, compared to a "normal" or "high" density screen.

For simplicity, Android groups all actual screen densities into six generalized densities: low, medium, high, extra-high, extra-extra-high, and extra-extra-extra-high.

Managing Virtual Devices

官网是这样解释显示密度的，某个屏幕物理面积下像素的总数；通常使用 **dpi**（每英寸下的点）为单位。比如：一个“低”密屏在给定的物理面积下，比“高”密屏或“正常”密度屏的像素点要少。简单的分类，**android** 把目前市面上的屏幕分为六个等级，低，中，高，超高，超超高，超超超高。

有了定义就不难理解上面的现象了，同样一张 500*500 的图片，在两个不同密度的手机屏幕上，的确需要“不同大小”的物理面积来显示。但对于要求体验一致的 **Android** 来说，这无疑是不可行的。**Android** 就专门为此问题，做了一套“与像素无关”的 UI 方案来应对。这里我们就不详细解说了，当然我们上面讲到的空间玩吧合流 **bug** 也是因为这套方案而产生的。但如果要通篇介绍这套方案，篇幅可能会很长，而且多数知识点离“内存”太远。

注：这套适配方案中包含比较著名的，建议使用“dp”来描述控件尺寸以及布局，它的大小等于“中密度屏幕”（160dpi）的一个“像素”大小。

回到正题，通过上面我们知道了一张相同的图片在不同密度的屏幕上会显示出不同的物理尺寸，而 Android 为了显示布局的体验统一，做了套适配方案，其中图片部分的方案，就会在内存中按照一定“比例放缩”这张要显示的图片。以达到“相同尺寸图片在不同密度屏幕上有相同的物理尺寸”的目的。那么，它会怎么放缩这张图片呢？

对于资源（就是放到工程目录 `res` 下的图片）图片而言，首先 Android 会要求资源助手寻找此图片所在的 `drawable` 目录，因为每个目录代表了不同的显示密度，如下表：

目录名称	Density
<code>res/drawable</code>	0
<code>res/drawable-hdpi</code>	240
<code>res/drawable-ldpi</code>	120
<code>res/drawable-mdpi</code>	160
<code>res/drawable-xhdpi</code>	320
<code>res/drawable-xxhdpi</code>	480

Android 会认为开发者非常清楚这些目录的代表密度，也清楚图片在屏幕上的用户体验，因此它要求开发者要尽量放置和屏幕密度相匹配的图片到对应目录下，比如：同样内容的图片，放在 `res/drawable-mdpi` 目录下的比放在 `res/drawable-xhdpi` 的像素尺寸要小一半。而 Android 最奇葩的要求是，希望每张图片，在不同的 `res/drawable` 目录下都应该有一个同名副本，它们拥有不同的像素尺寸，但是拥有一样的画面内容。这样可以保证好的画质，与性能（这里可能存在时延问题，有兴趣的同学可以深入了解下）。

Provide Alternative Bitmaps

Since Android runs in devices with a wide variety of screen densities, you should always provide your bitmap resources tailored to each of the generalized density buckets: low, medium, high and extra-high density. This will help you achieve good graphical quality and performance on all screen densities.

Android 在加载这些图片前，会先一步得到当前设备的显示密度，然后到相匹配的 `drawable` 目录去找寻图片资源。但是如果开发并没有按照官方推荐的方式，每个 `res/drawable` 目录都放置图片呢？Android 会按照当前设备显示密度就“近”获取图片资源，然后将其所在的目录所代表的密度与当前设

备密度相比，以这个比例来放缩图片，已得到一个“合适”的图片（有对应图片就不用放缩，这也是上面官网说“好”性能的原因）。

比如：一张备显图片只放置在 `mdpi` 目录，而当前的设备显示器为 `480dpi` 的超超高密屏，这时 `Android` 就会按照 3 倍大小放缩这幅图片，将它加载入内存。

这样就非常危险了，如果有一张 `800*480` 图片放置在 `ldpi` 目录，展示在 `480dpi` 的超超高密屏上时，会在内存中产生一张 `3200*1920` 的巨大图片，这一般会耗损 `23mb` 的内存，这个大玩意儿基本等于 `Android` 最低适配标准手机，单进程内存阈值的 `1/5`。官方建议图片缓冲的总大小应该为单进程内存阈值的 `1/4`。那么如果把这个大家伙塞进图片缓冲，得有多少无辜的小图片惨遭淘汰呀。而这恰恰是我们一开始讲述的那个空间的 `bug` 所产生的原因。

Screen Size	Screen Density	Application Memory
small / normal / large	ldpi / mdpi	16MB
small / normal / large	tvdpi / hdpi	32MB
small / normal / large	xhdpi	64MB
small / normal / large	400dpi	96MB
small / normal / large	xxhdpi	128MB
xlarge	mdpi	32MB
xlarge	tvdpi / hdpi	64MB
xlarge	xhdpi	128MB
xlarge	400dpi	192MB
xlarge	xxhdpi	256MB

空间因为安装包大小问题，无法为每张图在每个 `drawable` 目录下安置一张适配图片副本，这和大多数要控制安装包大小的产品类似。所以空间的开发只能选择其中一个目录来安放自己的业务图片，而这个开发因为对 `Android` 这块的适配并不是很清晰，于是就跟着前人，把自己的图片放在了那个 `res/drawable` 目录下，这目录后面没有带任何的密度相关的后缀，但是它所代表的屏幕密度值是“0”。可能开发认为这个背后不带有任意密度后缀的目录，`Android` 会做自动适配吧，这样就有了以上的 `bug`。

细心的读者可能会发现上面的解释似乎说不通了，因为一个“0”密度的目录，而目标设备为任何密度，都会除出来一个无限大的放大倍数。而这个 `bug` 其实只是将原来的图片放大了一倍，开发放到 `res/drawable` 目录下的图片尺寸为 `462*405`。原因很简单，在不同的 `rom` 下 `res/drawable` 会被替代成不同的密度，这个密度被称为“默认密度”。我的这台被测机“默认密度”为 `160`，而显示密度为 `320`，这样就有了这个 `bug`。

解决方案：

1. 目录法：抓不准该放那个目录的图片，就尽量问设计要高品质图片然后往高密度目录下放吧，这样在低密屏上“放大倍数”是小于 1 的，这样在保证画质的前提下，内存也是可控的。
2. API 调用法：拿不准的图片，使用 `Drawable.createFromStream` 来替换 `getResources().getDrawable` 来加载，这样可以绕过 Android 的以上这套默认适配法则。

案例：要使用统一的“图片停车库”

问题类型：图片类问题

无可厚非，我们所接触的大部分被测产品都并非是一个人设计、编码出来的（当然也不排除那些独立开发者存在）。所以“模块”概念会被当作软件设计者的必修课，那么这些被测产品就会拥有数个相对独立的模块，它们由不同的开发者开发、维护。比如下面我们要将的这个 bug，它发生在 QQ。

在 SNG 网络事业群中，负责群业务的部门和负责 QQ 基础业务的部门是相互独立的两个部门，他们虽然在共同的代码版本服务器上联合开发，但是对于对方的编程规范，设计思路都知之甚少。有一天，群业务部门做了一个需求，需要拉取云端的群头像（有点类似好友头像，是给群一个鲜明直观的标志），那么群业务部门就用 http 链路，加上认证信息，通过业务服务器把头像从云端拉到了客户端，但怎么把这些头像缓存在内存中呢？群业务部门自己建立了一个头像缓冲服务（服务单例），用一个静态的 Hashmap 容器，来存储这些图片，的确达到了头像缓存的目的，但是这样是最佳方案么？

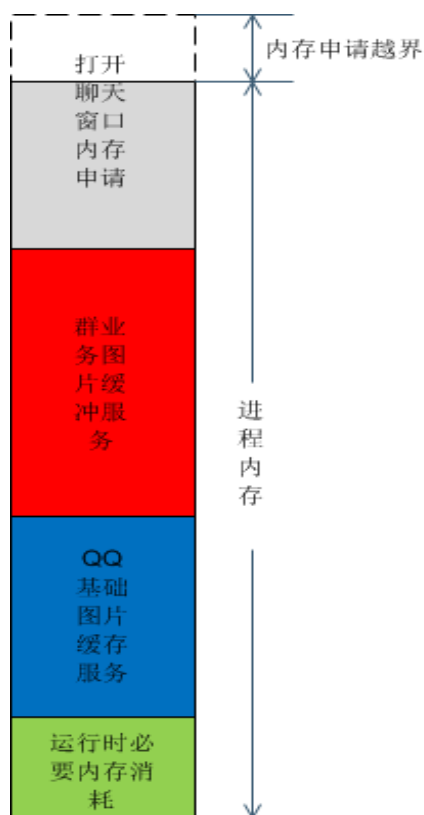
其实在 QQ 基础业务部分，专门有一个底层组在负责开发一些底层技术组件，其中就有图片缓存服务，它整合了缓存服务的大部分基础特征，查询，添加，更新，混合存储，老化算法等等。但要想真的使用起来，需要的沟通成本并不低，因为当时的底层组并没有文档输出的习惯，这也造成了群业务组并没有快速找到这个缓存服务组件，于是就自己开发了一个简易版。于是就早就了这个 bug。

操作步骤：

1. 在联系人列表中做分组展开操作
2. 查看 10 个联系人的个人详情
3. 查看群组下群列表
4. 查看 10 个群的资料，以及群成员
5. 进入消息列表，点击好友消息（图片消息），查看消息内容

Bug 类型：聊天窗口 Crash

分析如下：



(bug 示意图)

情况就是上图描述的样子，在进过了一系列使用后，打开聊天窗口后突然 crash，分析内存镜像发现，占用内存的两个大户是：群的图片缓存和 QQ 基础的图片缓存。群业务和 QQ 基础侧都按照官方推荐的作法，做图片缓存，所以他们各占了进程最大内存的 1/4，以至于聊天窗口再需要内存的时候发现内存不足了。

解决方法很简单，群业务删除自己的图片缓存服务，QQ 基础侧给基础图片缓存增加易用的接口，完善文档，大家公用一套图片缓存。

总结一下：一个产品最好只拥有“有限个”、“已知”的图片缓存，业务层都复用它，来缓存自己的图片，这样可以控制图片缓存所占内存“大小”，不至于造成上面这种“各自为政”的失控场面。

案例：把 WEBVIEW 类型泄漏装进垃圾桶进程

问题类型：WebView 类泄露

Webview 是 Android 的 web 页面展示控件，它继承于 view，使用 Webkit 引擎执行 web 请求，然后将获取的数据按照 html 规则渲染出来。它并非是一个完整的 web 浏览器，所以并没有导航控制，

或者地址栏。官方推荐它的适用范围也只有，展示用户协议（因为产品的每个版本用户协议都可能不同，或者要不定期更新用户协议），展示 mail 内容（mail 有相对复杂的字体，格式，段落，插画等等，用 Android layout 来做太麻烦）。

但是常常开发并不会注意官方建议的用途，而更加关注这个控件能够做什么。因为有 Webkit 引擎的支持，Webview 有了 HTML5 代码的执行能力，于此同时 Android 的 activity 框架还赋予了 Webview 完善 JavaScript 接口能力（web 中的 js 可以调用客户端 activity 已实现的标准接口）。这下 Webview 似乎成了产品、运营、甚至开发最中意的控件之一。

但网络延时，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。

当然，民间的牛人总是很多，他们尽然用“反射”也解决掉了其中的某些内存问题，比如下面这个：

```
1 public void setConfigCallback(WindowManager windowManager) {
2     try {
3         Field field = WebView.class.getDeclaredField("mWebViewCore");
4         field = field.getType().getDeclaredField("mBrowserFrame");
5         field = field.getType().getDeclaredField("sConfigCallback");
6         field.setAccessible(true);
7         Object configCallback = field.get(null);
8
9         if (null == configCallback) {
10             return;
11         }
12
13         field = field.getType().getDeclaredField("mWindowManager");
14         field.setAccessible(true);
15         field.set(configCallback, windowManager);
16     } catch (Exception e) {
17     }
18 }
```

它就成功的解决了下面这样的引用链：

com.tencent.open.applist.QZc	Activity @ 0x42f4b030		448	4,928
- mContext android.webkit.BrowserFrame @ 0x42b70718			80	312
- <JNI Local, Java Local> java.lang.Thread @ 0x42b4dc90	WebViewCoreThread Thread		80	1,224
- target android.os.Message @ 0x42e4a018			56	56
'- Total: 2 entries				
- mContext android.webkit.WebViewCore @ 0x42f5fd38			144	232
- mOuterContext android.app.ContextImpl @ 0x44adfb78			96	848
'- Total: 3 entries				

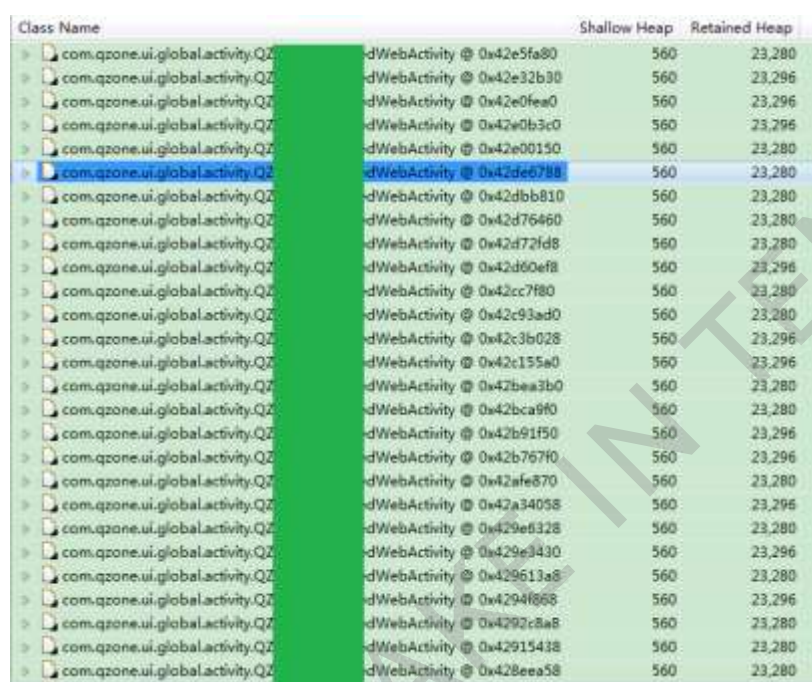
Class Name		Shallow Heap Retained Heap		

不难看出，作者使用了几次“反射”，首先“反射”得到 WebView 的内核，然后再通过“反射”从内核的 xxx 变量中得到窗口管理器回调配置，最后把一个空的窗口管理者赋给它，来替代原先的那个。这样就打断了底层的引用联调，成功的解决了 WebView 内核造成的上层 Activity 泄漏，但重点是作者最后补充了一句：这并不能适应所有的 Android 系统，因为它们 WebView 内核字段差异很大。

“反射”作为一种“补丁”来解决系统问题是推荐，并且鼓励的。但是将它用在内存问题的解决上，笔者还是建议大家持谨慎态度，因为这并不是解决内存问题的正确道路，正确理解组件本身，正确的调用接口，给出正确参数，采用正确调用顺序，这才是真正的解决内存的根本所在。

好了，上面说了 Webview 的内存问题，通过一个网络上的极端解决方法，也从一定侧面反映了广大的开发同胞对 Webview 泄漏的多么无奈。以下将要介绍的是腾讯在产品中用来统一解决 webview 问题的方法。

早期作为专项测试也曾经在产品内存中看到这样的情况，当时的感觉是真叫人“触目惊心”，要知道当时是使用自动化 monkeyrunner 在跑内存测试脚本，跑到一半我们手头的 Nexus5 就黑屏了，是假死状态，屏幕上的返回键，切换键完全失灵，只有 home 键勉强有所响应。抓取内存快照后，使用 finder-activity 就看到了以下的一幕：



Class Name	Shallow Heap	Retained Heap
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42e5fa80	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42e32b30	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42e0fea0	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42e0b3c0	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42e00150	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42de6788	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42dbb810	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42d76460	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42d72fd8	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42d60ef8	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42cc7f80	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42c93ad0	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42c3b028	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42c155a0	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42bea3b0	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42bca9f0	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42b91f50	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42b767f0	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42afe870	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42a34058	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x429e8328	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x429e3430	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x429613a8	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x4294f868	560 23,296
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x4292c8a8	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x42915438	560 23,280
com.qzone.ui.globalActivity.QZ	dWebActivity @ 0x428eea58	560 23,280

操作步骤：

1. 使用 monkeyrunner 运行脚本，进入 250 次“来消星星的你”（这是一款 QZone 玩吧的 web 小游戏）
2. 结束后抓取内存快照
3. 使用 finder-activity 分析界面泄漏

Bug 类型：泄漏（致命）

分析：

这里并没有展示引用联调，也不会做任何内部分析，原因很简单问题类似本节开始的那个引用链。我们要怎么解决这样严重的 bug 呢？以下摘抄了部分腾讯开发解决这个问题的邮件：



独立进程来解决内存问题，在官网上也并非没有提及，在《管理你的 app 内存》文章的最后，就是讲述如何用独立内存来解决内存问题，当然问题的角度有点不同。但是这并不影响“独立进程”是一种解决内存的方法的正确性。

独立进程在 android 框架下，非常简单，在官网的 app manifest 中<activity>的介绍中，有讲解有关 android:process 属性的设置，一旦设置了这个属性，这个 activity 的启动就会被投射到一个你所命名的进程当中了，最后在 activity 的 onDestroy 函数中，退出进程，这样就基本上终结了此类泄漏。

案例：定时器、延迟器的内存问题

问题类型：Context 泄露

定时器的适用场景非常广阔，比如定时刷新数据。常常用到 timer 类，它是一个 java 的基础类，Android 框架上可以直接使用。但它作为一种引用根，使用起来应该非常小心，因为非常容易造成内存泄漏。引用根的概念，通过前面的章节也介绍了不少，比如：线程、activity 和服务等，读者可以细心体会，这对内存问题的判别非常有好处。

我们同样使用一个 bug 来展示定时器使用时应该注意的内存问题：

这个 bug 出现在手机 QQ Android 版本上，手 Q 从 4.5 版本后就为打通 PC 与移动终端互联互通做了非常大的努力，这有助于实现多终端资源共享，同步的用户体验。其中有一个需求叫做“数据线”，它可以实现手机文件与电脑文件的互相发送、接收。这对于没有高品质摄像头的电脑来说非常管用，当然使用蓝牙传递文件也一样可以达到这样的效果，但速度与多文件传输对于默认的蓝牙服务来说简直是噩梦。

通常牵扯到文件传输的问题，都会出现“进度条”，这在上一节有讲到一些，当然它着重于“加载进度”，这里将到的进度是“处理或传输进度”。这种进度可以展示在界面，也可以脱离界面展示，数

数据线因为工作在多终端上，因此要实时的发送自己的进度给另一个终端，这点很容易理解，但是恰巧就是这个不难理解的场景会出现严重的内存泄漏。

测试步骤：

- 1. 使用 PC-QQ 发送文件给手机 QQ
- 2. 等待手机 QQ 提示，并在消息 tab 点击传文件消息
- 3. 接收文件过程，注销手机 QQ 帐号，重新登录
- 4. 抓取内存 hprof
- 5. 使用 finder-top class 功能
- 6. 发现出现两套帐号服务

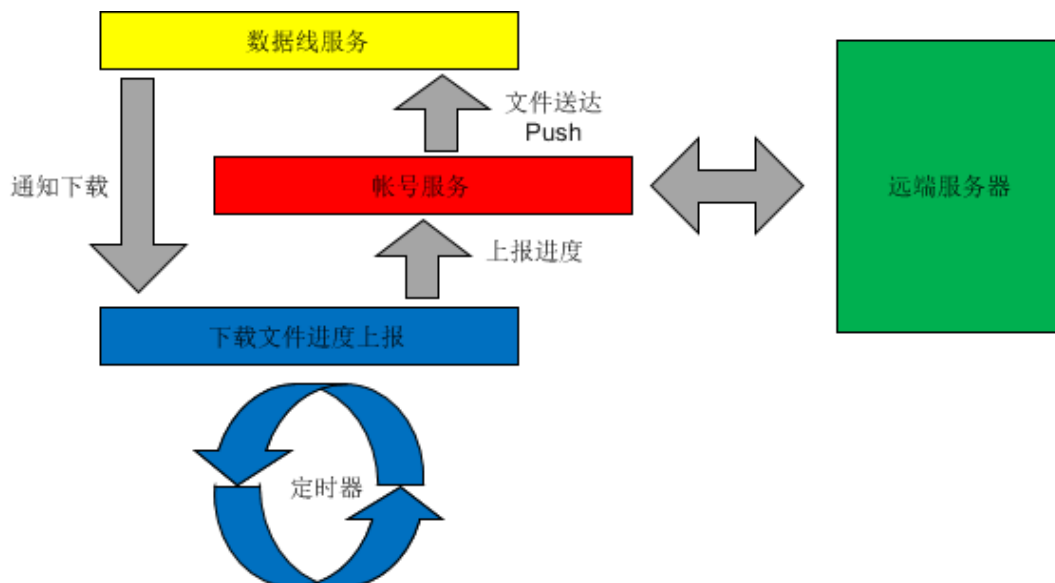
ClassName	ObjectCount	HeapSize	Percent
.com.	<Regex>	<Numeric>	<Numeric>
class com.tencent.the...eLoader @ 0x42118e38	1	3,069,288	0.12
class com.tencent.mobileqq.app.QQApp...ce @ 0x41ea5d68	2	2,036,544	0.079
com.tencent.mobileqq.app.QQAppIr... @ 0x41efc648	1	1,052,584	0
com.tencent.mobileqq.app.QQAppIr... @ 0x4249e098	1	983,960	0
Σ Total: 2 entries			
class com.tencent.mobileqq.star...emoryCache...MQL 1		1,715,000	0.067

分析：

在实现定时上报下载进度功能方面，手 Q 要通过帐号服务来接受 Push，所谓的 Push 是由服务器主动发送此命令给客户端，客户端对此做出响应动作。传文件的 Push 内容主要是说“XXX 的 PC 帐号 XXX 于 XXX 时间发送来了一个 XXX 文件”。这个 Push 里面除了包含必要的文件传输信息外，Push 本身也是启动手 Q 接收文件的启动命令。因此“数据线”服务对帐号服务的依赖是很强的，于是在内部就保存了对帐号服务的引用。

保存帐号服务的作用并非只有接收 Push 这么简单，下一步的上传下载进度也用到了帐号服务，帐号服务不单单存储了本帐号的信息，各种认证，同时包含了上传变更信息，接收服务器下发命令的“管道”。因此“数据线”服务就把自己持有的帐号服务，原封不动的也要传给上层接收文件服务。

接收文件服务会启动一个定时器，定时为 2 秒钟查询一下下载进度，并将其封装成 Tcp 命令，交给帐号服务发送到服务器。这样一个完整的 PC-QQ 发文件给手机 QQ 的流程就走通了。



从设计上就可以看出，帐号服务要被两个组件引用，数据先服务和下载进度上报定时器。现在问题出现了，测试员并没有在下载完成后结束测试，而是做了一个帐号注销的操作（可能在测试断点下载功能）。但在手 Q 的框架里，帐号一旦注销，那么帐号服务也应该被释放，等重新登录了新的帐号后，重新构建它。但这时“数据线”的开发忘记了在响应帐号登出接口里面停止下载文件上报定时器，这样就有了这个 bug。从引用路径上可以清楚的看到这一点：

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.tencent.***.app.QQApp @ 0x41efc648	464	1,052,584
app com.tencent.mob***.DataLine @ 0x428c7af8	112	608
this\$0 com.tencent.mobiled***.DataLine \$2 @ 0x425733d8	48	48
<Java Local> java.util.Timer\$TimerImpl @ 0x42571058 Timer-12 Th	88	1,552

解决方法：

在帐号注销响应函数中，停止正在运行的 timer

```

5      if (timerSendRead != null) {
6          timerSendRead.cancel();
7          timerSendRead = null;
8      }

```

也有读者会说上面这种释放方法是不是太残暴，的确官方有种比较温和的建议,只停止 timer 中的 task（使用 timertask 的 cancel 方法），然后调用 timer 的 purge 方法，来移除这些已经停止的 timertask，当然这种方法也是可以的，但还是要记得功能退出前把 timer 也要 cancel 掉。

定时器的内存问题我们就讲到这里，下面我们说说“延时器”，“延时器”在什么情况下使用呢？它和定时器又有什么区别呢？

首先应该确定一点，在一定程度上定时器是可以取代延时器的，比如服务器希望客户端延时做某些事情，或者客户端自己为了启动速度考虑延时去读取某些本地文件，这些场景下都可以使用定时器来完成延时任务。

但某些情况是不能随便使用定时器来玩的，比如定时界面刷新。启动的定时器并非与启动线程相同，多线程控制界面 UI 元素会导致异常这是最基础的常识，也就是说如果在 activity 里面启动一个定时器，并在回调函数里面刷新界面元素内容，这是不可行的。

当然也有方法来完善它，那就是在定时器的回调函数里面对外发送消息，使用一个 Handle 来俘获这个消息，并刷新界面。因为 Handle 的最基本特质就是如果没有指定线程创建，那么被启动的 Handle 会粘附于启动它的线程，从上面俘获消息栈中的消息，并处理。这样就可以在 Android 框架下做最基础的线程切换。

但是上面是否是最简单，有效的方法呢？消息满天飞，会大大增加产品维护成本。如果能像用定时器一样在回调函数里面直接表明要做的操作，而又能够达到“延时”，“线程切换”的要求是多好的事情。Android 早就考虑到开发者的这一诉求，于是“延时器”呼之欲出。

这个 Bug 出现在手机空间 Android 版本，进入空间会发现应用程序的最上部有一块图片展示区域，点击它可以进入一个叫做“背景商城”的业务块。



这个业务是用来更换空间顶部图片展示区域内容的。可以注意到在这个界面的最上部，是一个可以左右滑动的图片展示控件（梦境 xxx）。它不但可以左右滑动，而且还能自己定时轮播，从左至右循环播放这几幅图片。当前展示图片会在这个控件上停留几秒钟，然后后一副图慢慢“滚入”，如果当前图片被用户点击然后滑动，新划入的图片也要停留几秒钟，然后在开始“自动滚”下一幅。

操作步骤：

- 1. 进入空间背景商城
- 2. 退回空间好友动态 tab
- 3. 抓取内存快照 hprof，使用 finder-Activity
- 4. 发现存在界面泄漏

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.qzonex.module.feed.StoreFirstTabActivity @ 0x4416f568	440	27,312
com.qzonex.module.feed.FeedActivity @ 0x42554638	432	1,872
com.qzonex.module.h...portal.QZoneActivity @ 0x4231e498	504	1,936
com.qzonex.module.facade.ui.QZoneFacadeStoreActivity @ 0x44151758	384	3,472
com.qzone.ui.tab.QZoneTab @ 0x424fb9c0	312	872
Σ Total: 5 entries		

查看引用路径：

Class Name	Shallow Heap	Retained Heap	Id Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
com.qzonex.module.facade.ui.QZoneFacadeStoreFirstTabActivity @ 0x4416f568	440	27,312	27,312
mContext com.qzone.util.widget.WorkView @ 0x44192348	536	1,320	1,872
this\$0 com.qzone.util.widget.WorkView\$1 @ 0x44192728	16	16	1,936
<Java Local> java.lang.Thread @ 0x418dd700 main Thread	80	4,128	3,472
callback android.os.Message @ 0x432bcdc0	56	56	872
<Java Local> java.lang.Thread @ 0x418dd700 main Thread	80	4,128	
mMessages android.os.MessageQueue @ 0x423243c8	32	152	
<JNI Local Java Local> java.lang.Thread @ 0x418dd700 main Thread	80	4,128	

分析：

从引用路径上查看，发现这是一个内类引用 this\$0 非常明显，通过查询代码发现\$1 代表的一号内类，是一个 Runnable 对象，作用是当作“延时器”的回调。开发想实现的效果是，当图片划入完成后，开始设置一个延时，当延时完成后，开始滚动下一幅图进入控件。而用户点击图片滑动时，要清除原有延时。具体是通过 FrameLayout 的 postDelayed 方法实现的，其原理类似 Handle 的 postDelayed，两者在一定程度上可互相替代。

开发在调用 postDelayed 方法时，塞入了延时回调内类 Runnable，但在 XXXStroeFirstTabActivity 销毁的时候，并没有调用 removeCallbacks 移除掉这个内类，以至于导致泄漏。

注：延时器的泄漏特征在于其根部通常是一个主线程内的 MessageQueue，这也是它与定时器泄漏最显著的区别。

解决方法：

在 XXXStroeFirstTabActivity 注销的时候，移除自实现控件内部延时器对 Runnable 内类的持有。

FrameLayout:

boolean	<u>removeCallbacks(Runnable action)</u> Removes the specified Runnable from the message queue.
---------	---

Handler:

final void	<u>removeCallbacks(Runnable r)</u> Remove any pending posts of Runnable r that are in the message queue.
final void	<u>removeCallbacks(Runnable r, Object token)</u> Remove any pending posts of Runnable r with Object token that are in the message queue.
final void	<u>removeCallbacksAndMessages(Object token)</u> Remove any pending posts of callbacks and sent messages whose obj is token.
final void	<u>removeMessages(int what)</u> Remove any pending posts of messages with code 'what' that are in the message queue.
final void	<u>removeMessages(int what, Object object)</u> Remove any pending posts of messages with code 'what' and whose obj is 'object' that are in the message queue.

案例：列表控件 LISTVIEW 的 VIEW 泄漏

问题类型：其他问题

ListView 是 Android 界面 UI 组件中一个非常重要，而又强大的家伙。它定义了绝大部分互联网终端产品的 UI 交互总体验，因为手机多是一个“长条”型屏幕，列表在上面显示变得更加贴切，更加舒服。



知道了 ListView 的重要性，接着我们介绍一下 ListView 需求陷阱，所谓的 ListView 需求陷阱是指：因为和互联网互通后，ListView 中展示的内容不再是本地固定不变，数量有限的项，而是成了数量，内容都在时时刻刻变化的。

ListView 继承于 GroupView，内部用来存储 view，这点官方不少解释，这里就不多说了，GroupView 和 view 的主要区别是：GroupView 是一种容器，而 view 只是一种展示。按照前面对于服务篇的介绍，我们知道容器应该具有自清理能力，或者说“到达上限的淘汰”能力，否则越来越大的容器就会撑破内存阈值，造成 OOM。那么 ListView 存了过多的子 View 后，也会呈现这种情况。

Android 为了解决 ListView-OOM，就创造了 convertView->ViewHolder 模式，相信很多读者对此并不陌生，这里说一下两个名词的含义：

convertView：划出屏幕的 view 将会被回收转化成将要划入的 view，避免 ListView 里面不断增加 View，避免 OOM

ViewHolder：从回收的 view 中获取 View 实例，降低重新创建 View 开销，变得更加流畅

流畅度篇会主要介绍 ViewHolder，而这里我们主要介绍一下 convertView，因为 convertView 才是和内存指标强相关的。要叫 convertView 模式开启，必须要具备以下几个要素

1. `ListView` 和一个自定义的 `Adapter` 绑定
2. 自定义的 `Adapter` 实现了 `getViewTypeCount`、`getItemViewType` 与 `getView` 接口
3. 在 `getView` 里面使用第二个入参即 `convertView`，是它转化为即将出现在 `ListView` 中的 `view`

`ListView` 会在初始化的时候调用 `getItemViewType` 来得知当前的 `ListView` 中有几种 `View`，然后每次 `ListView` 有 `View` 对象将要划出或划入的时候，就会到 `View` 回收池（之前划出去部分 `View` 存在这里）存入或捡取对应类型的 `View`，其中获取类型的这一步就是通过调用 `getItemViewType` 实现的。把从回收池里取出的 `View` 当作 `convertView` 传递给接口 `getView` 复用，就完成了 `convertView` 机制的全部。

上面只是做了最简短的概括性说明，其实从 `google` 可以搜出更多的关于这个话题的研究，有兴趣的读者可以进一步研读一下。当然如果只了解使用层面上，具备有上述的知识就已经足够了。

`convertView` 机制的原理和必要元素我们都知道了，如果有那么一个需求他是互联网 `ListView`，可能拥有无数个子 `View` 而没有启用 `convertView` 机制，那么这必定是个低级内存 `bug`。但接下来我们介绍的 `bug` 会高级很多，它在开发实现了 `convertView` 机制的基础上，会打破 `convertView` 机制。

`Bug` 发生在 `QQ` 空间的独立 `Android` 版

操作步骤：

1. 在空间的搜索功能区输入“Wo”
2. 滑动搜索出的列表，不停的向下滑动，大概每 5 页抓取 1 个 `hprof`，总共抓取三个
3. 使用 `finder- TwoExecutionThreeDump`，发现 `android.view.View[]`持续增加



ClassObjectName	ChangedObjectsCount	NewCount	GCCount	CurrentCount	Incr Retained
java.util.ArrayList	3,000	3,000	0	3,426	152,000
java.lang.Object[]	2,002	2,001	1	2,431	1,494,616
android.text.TextPaint	2,000	2,000	0	2,148	208,000
java.lang.String	1,503	1,503	0	15,890	36,128
android.view.GLES20DisplayList	1,500	1,500	0	1,598	236,000
android.view.GLES20DisplayList\$DisplayListFinalizer	1,500	1,500	0	1,598	24,000
int[]	1,001	1,000	1	3,398	31,976
java.util.concurrent.locks.ReentrantLock\$NonfairSyn	1,000	1,000	0	1,076	24,000
java.util.concurrent.locks.ReentrantLock	1,000	1,000	0	1,076	40,000
android.text.BoringLayout	1,000	1,000	0	1,070	88,000
android.widget.LinearLayout\$LayoutParams	1,000	1,000	0	1,082	64,000
android.text.BoringLayout\$Metrics	1,000	1,000	0	1,071	32,000
android.widget.TextView	1,000	1,000	0	1,060	972,464
android.graphics.Paint	1,000	1,000	0	1,218	80,000
android.view.View[]	502	501	1	578	1,004,472
android.widget.LinearLayout	500	500	0	538	1,405,864
android.view.animation.Transformation	500	500	0	572	12,000
android.graphics.Matrix	500	500	0	595	8,000
android.graphics.PointF	500	500	0	575	8,000
android.widget.AbsListView\$LayoutParams	500	500	0	526	20,000
android.view.ViewGroup\$3	500	500	0	569	8,000
char[]	3	3	0	13,123	56

Bug 类型：View 泄漏

分析：

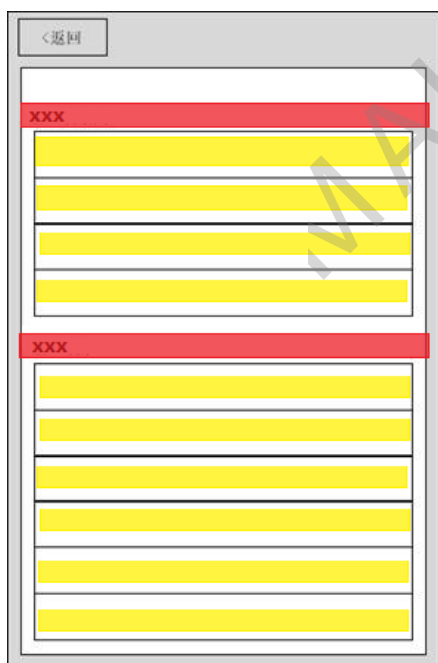
搜索界面可以简单的抽象成如下的模样：



它由两子列表构成，分别是“好友”、“非好友”，两个列表的内容都不确定。于是就出现了一个直白的设计，外部的“大”ListView 包含里面两个“小”的“ListView”，大小 ListView 均实现了 `convertView`。但 bug 还是发生了，这是为什么呢？

ListView 的 `convertView` 机制是在子 View“划入、划出”父 ListView 时刻生效，而以上的这种设计，“大”ListView 中的两个“小”ListView 其实并没有“划入、划出”了，而是变成了类是树形控件的“展开”。`convertView` 缺少了启动的机制，当然就失效了。

修改方案：



只用一个“大”的 `ListView`，而使用不同的“子”`View` 展示模式，其中红色的是一种 `View`，而黄色的是另外一种 `view`，这样就不会影响 `convertView` 生效了。

案例：处理“一车多位”

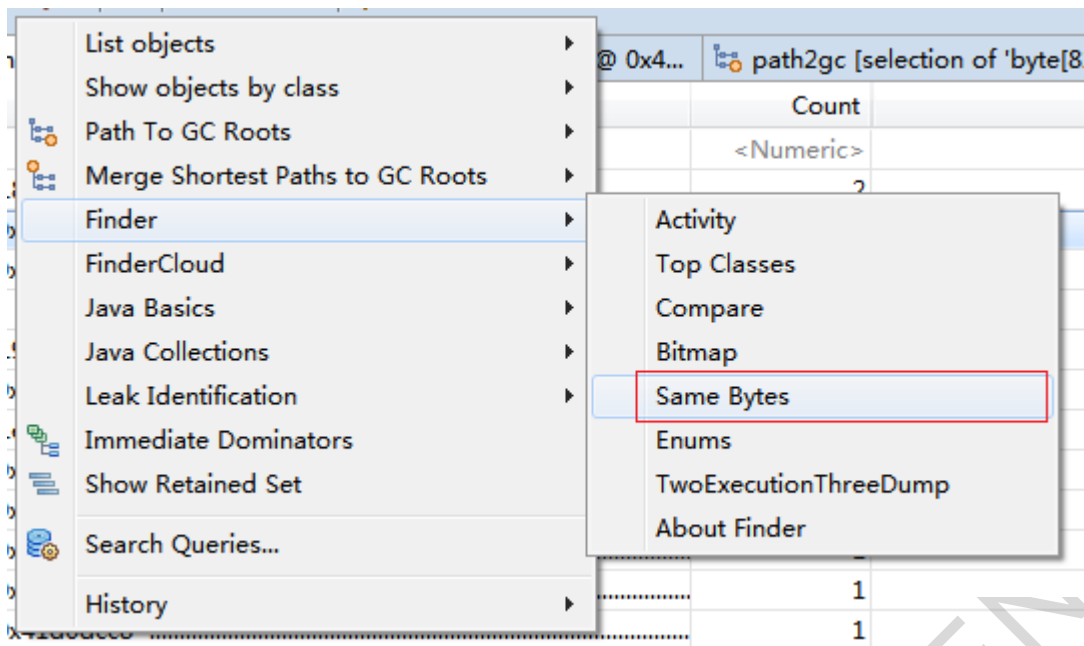
问题类型：内存资源重复

有了“公用”的图片缓存还只是把第一个坑填满了，那么紧接着产品就会遇到第二个坑：“一车多位”。图片缓存的寻址是按照经典的 `Key-Value` 模式，但如果我们把 `Key` 设置的不恰当，就很有可能出现“重复车”现象。也就是下面要讲的这个 `bug`：

`Bug` 一样是发生在 `QQ` 中，`QQ` 中有很大大块资源耗损是被“头像”创造的，不管是流量或者是内存，用户对于“头像”的情有独钟造成这块耗损变成了合理的强需求，这里简单说个计算公式，一般 `QQ` 会员使用的高清头像是 `144*144` 的，使用 `32bit` 方式加载，也就是说一个像素点需要 `4` 个字节来支撑，一个头像所耗损的内存不难算出，大概是 `81k`。那么当前主流 `Android` 设备的进程最大内存大概是多少呢？大体是 `64-128m`。那么对于拥有百个好友的用户而言，我们不需要做任何事情，只是把 `QQ` 登录，我们的内存就会有 `10m` 左右被头像吃掉，而且有很多用户不止百个好友。当时开发使用了 `UIN` 作为“头像”缓存的 `Key`，所谓 `UIN` 就是 `QQ` 号码，这样缓存不难理解。因为业务要使用缓存的“头像”时，也会根据 `UIN` 来取。这里看似没有问题，那么我们展示下 `bug` 的操作流吧：

操作步骤：

1. 在 `QQ` 联系人面板中，展开所有分组
2. 用 `DDMS` 抓取内存快照 `1.hprof`
3. 使用 `finder` 的 `Same Bytes` 功能
4. 对重复的 `byte[]` 查看 `gc` 路径
5. 使用 `finder-bitmap view` 查看引用路径上的 `bitmap` 对象



注：Same bytes 会排查内存中所有非空 `byte[]` 数组，检查他们是否存在内容相同，但对象不同的情况发生。

Bug 类型：重复缓存

Finder-Same Bytes 对这个 bug 的运行结果如下：

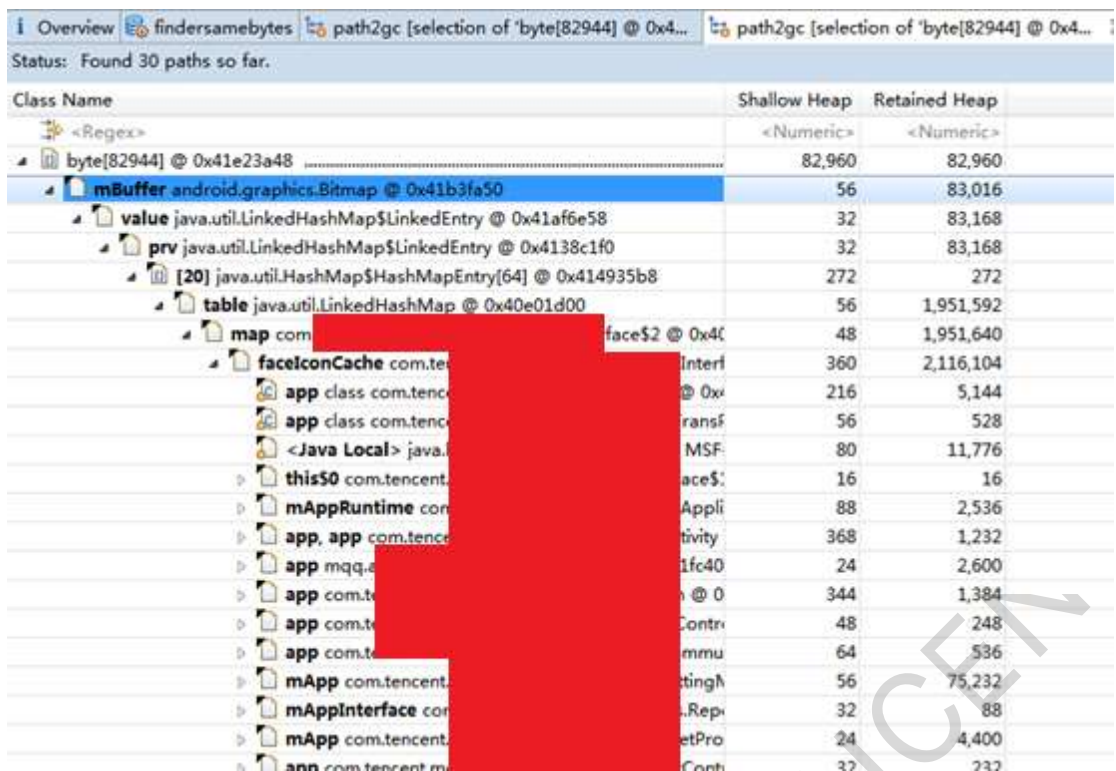
Overview		findersamebytes	path2gc [selection of 'byte[82944] @ 0x4...	path2gc [select
Bytes			Count	
<Regex>			<Numeric>	
▷	byte[40000] @ 0x418ea558		2	
▷	byte[47328] @ 0x419156a0		1	
▲	byte[82944] @ 0x41ec3e90		26	
	byte[82944] @ 0x41e23a48		1	
	byte[82944] @ 0x41e0f630		1	
	byte[82944] @ 0x41d98980		1	
	byte[82944] @ 0x41d220e0		1	
	byte[82944] @ 0x41d0dcc8		1	
	byte[82944] @ 0x41cd3318		1	
	byte[82944] @ 0x41c12ce0		1	
	byte[82944] @ 0x41bc67f8		1	
	byte[82944] @ 0x41bb23e0		1	
	byte[82944] @ 0x41b9dfc8		1	
	byte[82944] @ 0x41b870d0		1	
	byte[82944] @ 0x41aae088		1	
	byte[82944] @ 0x41a99c70		1	
	byte[82944] @ 0x41a5e508		1	
	byte[82944] @ 0x4189b730		1	
	byte[82944] @ 0x41747190		1	
	byte[82944] @ 0x4171e960		1	
	byte[82944] @ 0x416f6130		1	
	byte[82944] @ 0x41680dd0		1	
	byte[82944] @ 0x415e6ce0		1	

从运行结果来看，上面的操作，在内存中产生了 27 个（26 个复制+1 个自我）重复的 `byte[82944]`，这显然是存在一定浪费的，所以，我们随机抽取了两个重复的 `byte[82944]` 对象，查看它们的引用路径，查看引用路径是 Android 内存分析最重要的一环，也是 java 语言和 C 类在内存管理方面迥然不同的体现。

第一个 `byte[82944]` 对象引用路径：

Overview findersamebytes path2gc [selection of 'byte[82944] @ 0x4... path2gc [selection of 'byte[82		
Status: Found 30 paths so far.		
Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
byte[82944] @ 0x41e0f630	82,960	82,960
mBuffer android.graphics.Bitmap @ 0x4156ac20	56	83,016
value java.util.LinkedHashMap\$LinkedEntry @ 0x414e68e0	32	83,168
[13] java.util.HashMap\$HashMapEntry[64] @ 0x414935b8	272	272
table java.util.LinkedHashMap @ 0x40e01d00	56	1,951,592
map [REDACTED] interface\$2 @ 0x40e0...	48	1,951,640
faceIconCache com.tencent.[REDACTED]	360	2,116,104
app class com.tencent.[REDACTED] @ 0x40f8...	216	5,144
app class com.tencent.mobileq[REDACTED]	56	528
<Java Local> java.lang.Thread @ 0x41028358 MSF-Re	80	11,776
this\$0 com.tencent.[REDACTED] interface\$1 @	16	16
mAppRuntime com.tencent.[REDACTED] Applicati	88	2,536
app, app com.tencent.[REDACTED] activity @ (368	1,232
app mqq.ap[REDACTED] rImpl @ 0x411fc408	24	2,600
app com.tencent.[REDACTED] @ 0x41...	344	1,384
app com.tencent.mobile[REDACTED] trolle	48	248
app com.tencent.mobile[REDACTED] ica	64	536
mApp com.tencent.mo[REDACTED] an	56	75,232
mAppInterface com.te[REDACTED] rtC	32	88
mApp com.tencent.sc.a[REDACTED] idi	24	4,400
app com.tencent.mobil[REDACTED] olle	32	232
this\$0 com.tencent.mo[REDACTED] tSt	16	16

第二个 byte[82944]对象引用路径:



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
byte[82944] @ 0x41e23a48	82,960	82,960
mBuffer android.graphics.Bitmap @ 0x41b3fa50	56	83,016
value java.util.LinkedHashMap\$LinkedEntry @ 0x41af6e58	32	83,168
prv java.util.LinkedHashMap\$LinkedEntry @ 0x4138c1f0	32	83,168
[20] java.util.HashMap\$HashMapEntry[64] @ 0x414935b8	272	272
table java.util.LinkedHashMap @ 0x40e01d00	56	1,951,592
map com.tencent.mface\$2 @ 0x40e01d00	48	1,951,640
faceIconCache com.tencent.mface\$1	360	2,116,104
app class com.tencent.mface\$0	216	5,144
app class com.tencent.mface\$0	56	528
<Java Local> java.lang.Object	80	11,776
this\$0 com.tencent.mface\$0	16	16
mAppRuntime com.tencent.mface\$0	88	2,536
app, app com.tencent.mface\$0	368	1,232
app mqq.app com.tencent.mface\$0	24	2,600
app com.tencent.mface\$0	344	1,384
app com.tencent.mface\$0	48	248
app com.tencent.mface\$0	64	536
mApp com.tencent.mface\$0	56	75,232
mAppInterface com.tencent.mface\$0	32	88
mApp com.tencent.mface\$0	24	4,400
app com.tencent.mface\$0	32	232

看完两个 byte[82944]对象引用路径，发现都是属于与一个叫做 faceIconCache 变量持有的，而这个持有者就是头像服务的核心容器，由它来维护一个对头像的引用，以保证头像 bitmap 对象不被垃圾回收器回收。

接着我们还可以看看这幅图片到底是什么，将 Finder bitmap view 从 Windows 菜单中呼出，点击“引用树”上的“android.graphics.Bitmap”对象，然后调整 Bitmap informations，调至 ARGB8888，就可以看到这幅图片了，情况如下：



按照上面的分析，一个产品中存在 27 份相同的图片在内存中，这显然并不是那么合理的，合理情况是，存在一份图片，而业务来复用它，这才真正达到了图片缓存的意义。所以可以确定以上情况肯定是个 bug。

简单说下这个 Bug 的产生原因吧，前面我们说到开发使用 UIN 作为头像缓存的 Key，而测试环境下，测试号码加了数个“头像一样的不同测试号码”为好友，所以按照不同 Key 不同存储的原则，自然会有多个相同的头像缓存在内存中了。现实生活中也存在这样的情况，很多用户喜欢用 QQ 的默认头像，默认头像总共也就 40 多个，那么对于拥有几百个好友的用户来说，有那么三四十个头像重复的好友是很容易出现的。



问题的解决方法：头像服务器本身对于头像是除重的，所以每个头像在腾讯的服务器都有一个根据其内容而生成的唯一编号，内容相同的头像如果上传服务器，服务器也只会保存一份，所以客户端的 key 应该延续这一做法，客户端保存一份 UIN 与头像编号的映射表，而将缓存的头像按照以头像编号为 key 的方式存储，这样就从逻辑上除重了。



案例：大家伙要怎么进入小车库

问题类型：图片类问题

好了，到了这里可以说我们的“停车场”已经差不多稳固了，它有了一个众所周知的地址，和一个简单的“车辆验证”。那么现在停车场还会碰到什么问题呢？下面我们来说说超标车辆。

在服务器的思维中，它们是没有权利知道资源是被什么终端请求的（当然也能通过某些手段获取，但一般只做简单的统计，而不做区别逻辑），所以服务器不知道设备的详情，也就是说它们只会根据请求找到图片，并把它发给请求设备。那么请求设备很有可能会得到一张“大家伙”，比如单反照的照片（它们充斥了整个空间相册服务器），这些“大家伙”有如下特点：

1. 分辨率大，1000*1000 分辨率司空见怪

2. 压缩比大，通常使用很厉害的无损压缩技术来处理它们，以节省存储空间，比如 webp
3. 颜色绚丽，夕阳，日出，海滩，雨后花瓣美不胜收

看到这些特征，一般用户会感觉自己买的单反相机物超所值极其欣慰，但是对于内存测试来说或者是开发者来说无疑是头痛至极。分辨率大耗损的内存就多，压缩比大使用我们必须使用图片缓存来降低展示时延，这在前面提到颜色绚丽逼得我们只能用比较多的字节来存储每个像素点的信息，比如 32 位。而这一切都会把辛辛苦苦建立起来的“停车库”，冲的七零八落，“大家伙”解码后，很可能本身体积都超过了停车场的容积。比如一款低端手机，最大进程阈值是 32m，那么“停车场”也就是 8m (32/4)，一张 1500*1500 的照片（来自 300 万像素相机，300 万像素已经低于大多数主流相机的分辨率，现实情况更加恶略，他们都在使用 1000 万以上的大炮），解码后在内存中将会占据多大呢？

$$1500*1500*4/1024/1024=8.58m$$

看到如上的数据大体已经感觉到，有问题了。其实车库被撑爆也并非悲剧本身，悲剧的情况会发生在这时：有那么一刻“车库”停满了车，而“车库外面”也没有空地给一个“大家伙”栖身，那么这时一张“重型”图片风尘滚滚的而来，会造成什么情况？我们展示下空间遇到的 bug 吧：

操作步骤：

1. 空间某个版本符合灰度发布指标（灰度，即体验包，给小部分用户体验新功能，接受用户反馈的做法，大多数互联网公司都有这个发布阶段。达标，是指 bug 解决率超过某个指标且无严重问题存在）
2. 发布量控制在 30 万左右，半天后，查看 crash 上报（如果空间崩溃了，会把当时的崩溃原因以及堆栈信息汇报给产品稳定性统计服务器）
3. 发现 OOM（内存溢出）占据 50% 以上的 crash 量
4. 查看堆栈，OOM 主要发生在图片缓存解码堆栈（也就是图片要存入缓存的那一刻）

Bug 类型：crash


```

BitmapFactory.Options options = new BitmapFactory.Options();
options.inJustDecodeBounds = true;
BitmapFactory.decodeResource(getResources(), R.id.myimage, options);
int imageHeight = options.outHeight;
int imageWidth = options.outWidth;
String imageType = options.outMimeType;

```

（得到图片的高和宽）

```

public static int calculateInSampleSize(
    BitmapFactory.Options options, int reqWidth, int reqHeight) {
    // Raw height and width of image
    final int height = options.outHeight;
    final int width = options.outWidth;
    int inSampleSize = 1;

    if (height > reqHeight || width > reqWidth) {

        final int halfHeight = height / 2;
        final int halfWidth = width / 2;

        // Calculate the largest inSampleSize value that is a power of 2 and keeps both
        // height and width larger than the requested height and width.
        while ((halfHeight / inSampleSize) > reqHeight
            && (halfWidth / inSampleSize) > reqWidth) {
            inSampleSize *= 2;
        }
    }

    return inSampleSize;
}

```

（计算放缩比）

```

public static Bitmap decodeSampledBitmapFromResource(Resources res, int resId,
    int reqWidth, int reqHeight) {

    // First decode with inJustDecodeBounds=true to check dimensions
    final BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeResource(res, resId, options);

    // Calculate inSampleSize
    options.inSampleSize = calculateInSampleSize(options, reqWidth, reqHeight);

    // Decode bitmap with inSampleSize set
    options.inJustDecodeBounds = false;
    return BitmapFactory.decodeResource(res, resId, options);
}

```

（放缩式解码）

```

WindowManager wm = (WindowManager) context.getSystemService(Context.WINDOW_SERVICE);
Display display = wm.getDefaultDisplay();
int width = display.getWidth(); // deprecated
int height = display.getHeight(); // deprecated

```

（获取显示屏宽、高）

总结：有很多读者可能心里还在打鼓，那么按照屏幕来缩减的图片难道就可以说不是“大家伙”了么？难道不会有进程阈值 32mb 的机器却有个 1280*960（1080p）的巨屏么？如果不是山寨货，可以保证不会发生这种情况，因为虽然 google 对于 android 的态度是开源谁都可以用，但是也做了适用硬件的要求：

Screen Size	Screen Density	Application Memory
small / normal / large	ldpi / mdpi	16MB
small / normal / large	tvdpi / hdpi	32MB
small / normal / large	xhdpi	64MB
small / normal / large	400dpi	96MB
small / normal / large	xxhdpi	128MB
xlarge	mdpi	32MB
xlarge	tvdpi / hdpi	64MB
xlarge	xhdpi	128MB
xlarge	400dpi	192MB
xlarge	xxhdpi	256MB

这个表格来自于 Source.android.com 上的适配章节，目的是要求 rom 编写者在知道硬件情况的前提下，要给每个进程的最低内存，而刚才打的比方 1080p 的屏幕应该是属于 large 屏，而如果不像把手机做的像盾牌那么大，最起码应该才用的屏幕密度也应该是 xhdpi，也就是起码每个进程的内存阈值不应该低于 64mb。

有了这样的限制，那些“大家伙”经过按照屏幕放缩自己身材后，与和屏幕大小有关联的“停车场容积”而言也就不会那么危险了。

案例：大车小车分开停

问题类型：图片类问题

通过上面的操作，现在的图片缓冲已经可以说基本完成，具备直接上战场的能力了，但是还有没有方法叫它更加高效呢？所谓的高效是指“低淘汰、高复用”。试想一下，如果停车场为了停的下一辆“大”车，是不是很有可能需要挤出很多“小”车呢？答案是肯定的，图片缓存的容积是一定的，一旦容积满了，就要开始执行淘汰规则，这个时候如果新加入一张比较大（像素点多）的图片，势必需要挤出很多以前在缓存中的“小”图片。

下面要介绍的不是一个 bug，而是开发的一种设计，也是关于 QQ 头像的，头像在内存中的大小基本保持在 41k 左右，而从聊天窗口或者浏览相册组件来的图片通常都在 1m 左右，这种不平衡，就像是不同重量级别的拳击手上到了同一个擂台一样，因为要缓存大图，头像缓存惨遭大面积淘汰，但是从这些界面出来后，常用的头像又要重新加载回内存。

为了解决这种情况的发生，开发就把头像缓存和大图缓存分开了，就相当于多增加了一个擂台，叫不同重量级别的选手在不同的平台上竞争，保证公平，大举提高了缓存效率。

总结：未必所有的产品都有头像缓存这一特殊的图片缓存，所以没有必要所有的产品都搞出一套大小图缓存，也没有必要刻意的图片尺寸分区域保存，因为这都会打乱 LRU 原则。正因为有这个必要性，所以把它排在了第四位。希望读者要因事而为，万不可生搬硬套。

案例：我是这样想的，ANDROID 要纠正内存世界观了

在工具篇 Meminfo 和 Proccstats 就已经提到了【内存负载】的话题。由于 Android 独特的软硬件架构，会导致对一种命题的辩证，“产品的内存专项质量，是否应该用‘越小越好’来评判”。

要判别这个命题，就要理解 Android 框架对内存使用的管控。Android 首先使用的是一个去掉 swap 的 linux 内核（至少在 4.4 以前的版本是这样），这样就阻碍了 Android 上的应用程序使用 Page out（应用程序使用的内存，对操作系统而言都是一张张 Page，而对于老化的 Page，操纵系统可以将它们从内存中置换到硬盘上，这种操作叫做 Page out），这一常规的内存操作。那么是不是可以理解，Android 应用就更应该省着用内存了呢？答案还不一定。

Android 框架对与进程内存的第二个管控特征是，每个进程都有一个内存最高阈值（纯净的 Native 内存申请不算在这里），一旦进程申请内存突破了这个阈值，将会产生异常，并退出运行时物理内存空间。简单的说也就是 Android 为每个进程已经分好了一块蛋糕，至于你吃或者不吃，是你自己的事情。但这是否意味着 Android 应用程序为了效率考虑，应该玩命儿申请内存，使自己的内存沿着天花板滑行，这样是最健康呢？答案也不一定。

Android 的第三个管控特征是，进程都有可能被杀。在物理内存吃紧的时候（通常是使用 Meminfo 查看内存概况的 PSS 总值达到设备物理内存的 80%左右时），Android 框架就开始根据一套自由的 LRU 进程 Cache 列表来杀死进程，被杀死的进程，在死前将会得到通知，用以保存现场。而这部分被杀死的进程，所腾出来的物理内存，就可以用于某些应用程序的内存申请需求。那么是不是为了不被杀死，Android 应用应该尽量减少自己内存，以降低在 LRU 进程 Cache 列表中的排名呢？答案还是不一定。

到这里我们已经了解到 Android 内存框架下的各种管控特征了：

- 没有 Page out，所以物理内存更加金贵
- 每个进程都有一个内存上限，所以蛋糕是已经分配好的
- 所有的进程都有被杀可能，所以要做好被杀准备

读到这里，相信很多读者都开始发晕了，作者说了这么多，到底建议怎么来定性的去解释内存专项的“好与坏”呢？其实这不是多数读者的疑惑，也是众多开发者对于 Android 平台内存专项评判标准的疑惑，感觉怎么做都不能称之为“好”，或者称之为“不好”。

于是 Android 就退出了一个概念【内存负载】。

其计算方式是：应用在“某种状态”下的运行时长，乘以其平均物理内存占用（PSS）。简单的说，就是要告诉用户，某个应用在“后台”、“前台”或者是“缓冲”中的内存累计消耗。内存累计消耗高，会增加运行“新”应用的耗时，即用户所说的“手机”很卡。

因此也就多了一个统计工具 Procstats



Meminfo 在 4.4 以后的版本也会多出一项叫做“Cached”的列表：


```

48933 kB: System
48933 kB: system <pid 614>
124805 kB: Persistent
105247 kB: com.android.systemui <pid 901>
8512 kB: com.android.phone <pid 1047>
4114 kB: com.android.nfc <pid 1079>
2981 kB: com.redbend.odmc <pid 1086>
2265 kB: com.bel.android.dspmanager <pid 1055>
1686 kB: com.android.incallui <pid 1095>
50835 kB: Foreground
50835 kB: com.cyanogenmod.trebuchet <pid 1114 / activities>
1746 kB: Visible
1746 kB: com.android.smspush <pid 1207>
65519 kB: Perceptible
31398 kB: com.tencent.mobileqq <pid 7338>
22260 kB: com.iflytek.inputmethod <pid 1026>
11861 kB: com.tencent.mobileqq:MSF <pid 1011>
41695 kB: A Services
21797 kB: com.taobao.taobao <pid 1939>
12921 kB: com.taobao.taobao:push <pid 3256>
6977 kB: com.tencent.mm:push <pid 2858>
163541 kB: B Services
25385 kB: com.tencent.mm <pid 2897>
17344 kB: com.qiyi.video <pid 2575>
16226 kB: com.tencent.Alice:xg_service_v2 <pid 1510>
13128 kB: com.tencent.qqmusic:QQPlayerService <pid 3571>
12122 kB: com.qzone:service <pid 2352>
11872 kB: com.tencent.qqmusic <pid 2506>
11308 kB: com.qiyi.video:hdservice_v1 <pid 2663>
10263 kB: com.android.mms <pid 1625>
10263 kB: com.tudou.android:push <pid 2562>
9778 kB: com.UCMobile:push <pid 3243>
9380 kB: android.process.media <pid 1191>
9275 kB: cn.goapk.market <pid 2390>
5236 kB: com.android.de <pid 2213>
1961 kB: com.qualcomm.qcrilmsgtunnel <pid 2300>
61021 kB: Cached
16272 kB: com.taobao.taobao:notify <pid 3161>

```

至于 Procstats 的使用，和三种“状态”的含义，在工具篇的 Procstats 下已经讲的非常清晰了，这里我们主要将一个应用场景和案例，它发生在一次老板（腾讯内部戏称自己的上级为“老板”）问我，“嘿，咱们 SNG 的终端产品，内存到底做的怎么样了？”，于是我做了一个简单的产品间对比，汇报给了他。

操作步骤：

1. 抽取一些明星产品：微信，QQ，QQ 空间，应用宝，QQ 浏览器（其中 QQ 和 QQ 空间是 SNG 的，而其他的是别的部门的）
2. 逐个使用死号登录这些应用（这里所谓死号即好友数量少，测试时段不会收到好友消息，不会收到 push 信息等），然后将其挂入后台
3. 过 3-8 个小时再查看内存负载 TOP 列表

发现如下情况：

7 小时负载降序排列：微信>应用宝>QQ 浏览器>QQ(MSF)>QQ 空间(service)>应用宝(connect)>微信(push)>应用宝(uninstall)

3 小时负载降序排列：QQ 浏览器>微信>应用宝>QQ 空间>QQ>QQ(MSF)>QQ 空间(service)>应用宝(connect)>微信(push)>QQ 浏览器(service)>应用宝(uninstall)

于是拿着这份数据去给老板解读：QQ 系产品的内存负载，在 Android 上的表现均优于非 QQ 系的产品。

分析：

虽然测试结论非常乐观，但作为一个“技术族”的成员，必须要问一下“为什么”，以及 QQ 系的产品是怎么做到在内存负载方面优于别的目标产品的。Procstats 章节我们已经介绍了所谓的“潜规则”，以上的对比结论也是展示“后台”内存负载排行榜。因此要了解排行的原因，先要看看“后台”的定义。

所谓的“后台”就是指那些进程呢？它要满足以下特点：

已被用户使用物理“返回键”，退回的进程

1. 进程包含了服务 `startService`，而服务本身调用了 `startForeground`（要通过反射调用）
2. 主 Activity 没有实现 `onSaveInstanceState` 接口

这些特点都会告诉 Android 的 `ActivityManager`，你的进程是一个不可轻易被杀的应用，一旦杀死它可能会造成“用户体验”问题。这并不是个好情况，越来越多的应用编写者为了叫自己的应用“最大限度”的待在“运行时”，会故意做出以上的特点。这样会把整个设备的物理内存用到“无可可用”的状态。

解决方法：

QQ 本身没有去实现“不可杀”特点，所以 QQ 本身运行状态很容易击中 Cache 状态。

QQ 空间，虽然实现了“不可杀”的特点，但在三小时后会主动保存并重启它，这样内存负载会跟随进程的退出而下降。