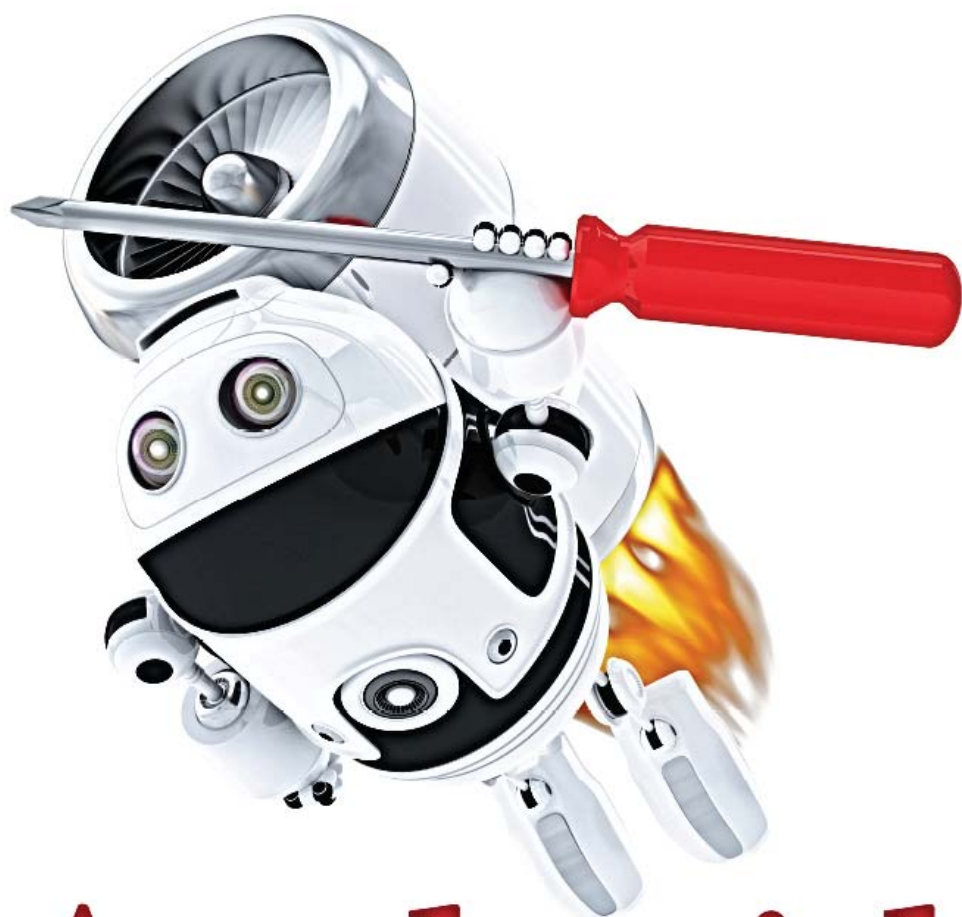


来自多年来**手机QQ、QQ空间、QQ音乐**等项目沉淀下来的**经典案例**
从内存、CPU、磁盘、网络、电量、流畅度、响应时延等多个方向进行介绍
是Android App性能和开发工程师的**必备案头手册**

Broadview[®]
www.broadview.com.cn



Android

移动性能实战

腾讯SNG专项测试团队 编著

 中国工信出版集团

 电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

内容简介

本书从资源类性能中的内存、CPU、磁盘、网络、电量和交互类性能中的流畅度、响应时延，多个性能测评和优化的方向出发。每个方向，都会帮助读者深入浅出地学习必须要懂得的原理和概念，区分众多专项工具使用的场景和对应的使用方法；同时提炼总结不同类型的性能缺陷和对应的排查手段、定位方法和解决方案，透过真实的案例，让大家身临其境地快速学习；提供建立专项性能标准的武器与武器的来源，让读者能快速落地项目并产生成效。本书的最后，还会帮助读者从全新的角度学习如何应对专项测评要面对的两个基础问题：UI 自动化测试和竞品测试。

本书适合从事移动 App 性能测评和优化的工程师阅读，内容有一定的技术深度和广度，建议读者在阅读本书的同时扩展阅读其他经典的技术类书籍。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

Android 移动性能实战 / 腾讯 SNG 专项测试团队编著. — 北京：电子工业出版社，2017.4
ISBN 978-7-121-31064-5

I. ①A… II. ①腾… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2017) 第 047562 号

策划编辑：付 睿

责任编辑：徐津平

特约编辑：顾慧芳

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：22.5 字数：504 千字

版 次：2017 年 4 月第 1 版

印 次：2017 年 4 月第 1 次印刷

定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

Android 移动性能实战

腾讯 SNG 专项测试团队 编著

電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内容简介

本书从资源类性能中的内存、CPU、磁盘、网络、电量和交互类性能中的流畅度、响应时延，多个性能测评和优化的方向出发。每个方向，都会帮助读者深入浅出地学习必须要懂得的原理和概念，区分众多专项工具使用的场景和对应的使用方法；同时提炼总结不同类型的性能缺陷和对应的排查手段、定位方法和解决方案，透过真实的案例，让大家身临其境地快速学习；提供建立专项性能标准的武器与武器的来源，让读者能快速落地项目并产生成效。本书的最后，还会帮助读者从全新的角度学习如何应对专项测评要面对的两个基础问题：UI 自动化测试和竞品测试。

本书适合从事移动 App 性能测评和优化的工程师阅读，内容有一定的技术深度和广度，建议读者在阅读本书的同时扩展阅读其他经典的技术类书籍。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

Android 移动性能实战 / 腾讯 SNG 专项测试团队编著. — 北京：电子工业出版社，2017.4
ISBN 978-7-121-31064-5

I. ①A… II. ①腾… III. ①移动终端—应用程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2017) 第 047562 号

策划编辑：付 睿

责任编辑：徐津平

特约编辑：顾慧芳

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：22.5 字数：504 千字

版 次：2017 年 4 月第 1 版

印 次：2017 年 4 月第 1 次印刷

定 价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888, 88258888。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

推荐序一

写在开头，送贾岛《剑客》诗一首：“十年磨一剑，霜刃未曾试。今日把似君，谁为不平事！”我们团队工作重心转到移动互联网领域已经好几年了，团队在移动领域测试技术积累可以说是从零基础开始的，几年来，配套的各类技术攻坚、工具平台建设都具备了很好的沉淀和规模，同时团队在这期间的自我实践提升和转变速度也非常快，如果自我吹捧一下，那么这就是一支优秀团队所具备的核心竞争力。这几年来，看着大家能不断探索攻克一个个难题并填坑，其实是一件很幸福开心的事情！这期间的学习、探索和实践，借用一句典故就是“工欲善其事，必先利其器”，我们都在说“磨刀不误砍柴工”，道理都是一致的，腾讯的专项技术测试团队从 2010 年开始组建，近 7 年来已经不断体现出其强大影响力和价值，成为了研发团队最坚实的战斗伙伴之一，而我们专项技术测试团队这几年也不断夯实了移动测试领域的重点攻坚领域、填补了几乎所有短板，并且也是努力从基础提升做起后到现在带来的结果体现！这好比在练武术时，早期教练会让学员练习扎马步，大家在健身房请私教时，会发现教练要求学员一定练深蹲，这些日久才能发挥威力体现基本功的基础动作，对武术提升和健身起到举足轻重的作用。我们团队过去持续保持平和心态，聚焦在短板上不断学习、钻研和沉淀，也在今天不断体现出了价值和给业务提供着重大支持。这是一支务实、踏实但又保持持续创新的团队，这也是我们团队的宝贵财富和一贯传承的管理思路。

每次看到行业里有新书出来时，我基本都会第一时间来了解获取，首先希望拜读理解作者的思路，然后看书内容里的技术实践深度，我这个人很懒又很挑剔，宁愿花很多时间来提炼内容写个 PPT 给大家做分享，也不愿花很多时间坐在电脑旁边码字、写出一摞看起来厚厚的很有“成就感”的文档来给人读，因此我是真心佩服那些能写出大部头书籍的同仁，过去几年里承蒙同仁抬爱，我给多本书写过序，虽然让我有些“愤愤然”，我是“作序君”嘛，但也总是很欣慰，佩服同仁和我的朋友 / 同事们熬夜码字的毅力，也佩服他们能抽丝剥茧，把自己的经验实践用一本书完整地呈现给读者的魄力。但其实我想说，写书本身是一件严肃的事情，也是把自己扒光了晾给大家看的一个过程，一本书如果东拼西凑，大部分内容要么是截图、纯图片，要么是“腾挪”了很多他人的内容，这种书其实出版出来也是体现了作者典型的“囊中羞涩”，此类书不出也罢，因为会食之无味，让读者读完基本没啥收

Android 移动性能实战

获，反而浪费时间，误人子弟。

我们 SNG 专项技术测试团队这次要出版的书籍，我不想给予太多的赞美，不然就是在自我吹捧了，毕竟这本书算是集我们团队之力的实践分享，同时也是团队工作日常点滴积累所得，希望对大家有用。本书所有内容产生的背景是日常工作开展过程中各个维度攻坚实践的过程，本书以看到问题—定位问题—解决问题—找共性—抽象化 / 平台工具化—提炼原理的方式积累呈现出来，每个维度每个领域的案例都是真实的，容不得取巧，更没有很多花哨的架子，放出来的是点滴积累出来的真实工作经验。初期来阅读时，很多行业同仁可能会觉着有点乱甚至晕，我建议先把书籍的目录章节仔细研究，梳理清楚这本书希望传递给大家的思想和体系，然后再有针对性地阅读和学习，这样才能事半功倍。

两年多前，整体回顾我们团队专项测试开展情况时，我勾画了一个所谓的“专项测试战略地图”，不过我们的团队属于有些“不按套路出牌的团队”，并没严格按我规划的在推进，或即使在推进，也并没老实地回头看这个“地图”，但让我们更开心的是团队自身的从下往上创新、创造的意识，使这期间诞生了很多创新型项目 / 工具，这其实是团队自己的“道”，非常值得鼓励，欣喜看到左冲右突的人在团队中大有人在，幸事！其实，让一个人的思想和行为不得自由的，有两个牢笼：一个是对过去的贪恋和自满，定死了自己的思维和进取；另一个是对未来的恐惧，以及对它的贪婪，定死了自己的勇敢和视野。打破这两个牢笼，会顿悟得“道”。我们在人际交往中，对一个人的评价经常因为生活中的小事情决定，这是因为一个人的秉性很难改变，不管是淳朴务实还是爱慕虚荣的表现，回归到工作上时，不外乎是持之以恒、不断聚焦和专业化，或是昙花一现、只做一时耀眼的流星，而很多人并没悟透这个最基本的道理。“逻辑思维”里说过一个小典故，僧侣得道前的日常作业是挑水、劈柴、做饭，得道后还是挑水、劈柴、做饭，做一行能做到如此才是“大道”。

今天我们走的这条路很不幸，不再有看板可以让我们比对，时代变化太快，过去的经验、经历未必还管用，未来的道路如何也无法预测，但这好比待在一个黑暗的道路上摸索前行，可能有很多弯路，可能不断跌倒，但只要有信念，坚定前行，无论荆棘坎坷，彼岸总会泛出微光指引我们不断前行，相信那也是我们心底最灿烂的光明之火！

腾讯社交网络质量部 吴凯华

推荐序二

现在的移动互联网是一个用户体验为王的时代，你的用户群会决定你的产品的成败。而移动无线测试中的专项测试就变得非常重要，功能和业务测试保证了一个产品的生命，而专项测试则能够延续一个产品的生命。

移动互联网到底是什么？带给老百姓的是生活的便捷，带给程序员的是新鲜的技术和更快的工作节奏。在早期大家都在谈论 Android、iOS 和 WP（Windows Phone），然后则开始谈论物联网。而如今神秘选手横扫全球围棋界，所有人都在讨论这个“选手”，结果 AlphaGo 的出现让 2016 年成为了人工智能元年，也掀起了人工智能在人类历史上的第三次浪潮。

- 当我们还在用诺基亚砸核桃的时候，Android 和 iPhone 来了；
- 当我们以为移动支付只有支付宝的时候，微信支付来了；
- 当我们觉得二维码这项发明没有意义的时候，微信和支付宝等 App 狠狠地给了我们响亮的耳光；
- 当我们开始玩朋友圈的时候，公众号出现了；
- 当我们开始熟练使用公众号的时候，小程序来了；
- 当我们觉得 AR 没有什么实际的有黏度的用户场景的时候，Pokémon Go 让所有人都拿着手机扫全世界，甚至在美国的高架上还有专门的路标提示不要玩 Pokémon Go；
- 当我们觉得手机只能用来打电话、玩游戏、支付、上网的时候，Google Cardboard 让我们知道原来我们可以进入手机的世界；
- 当我们还沉浸在抨击 VR 还不成熟的时候，Vive、PSVR、Oculus 等让我们欲罢不能；
- 当我们以为 Siri 已经能够打败人类的时候，AlphaGo 让我们明白其实人工智能才刚刚向人类发起挑战；
- 当我们在各个演唱会上面看到全息投影，觉得离我们还很遥远的时候，Gatebox 出现了（日本全息投影女管家）。

这一切的一切说明了移动互联网并没有具体的形态，它仅仅代表着一个高速发展的时代已经来了。我们很幸运，能够活着看到时代的发展和变迁，我们也会很累需要不停地去

Android 移动性能实战

接受和面对挑战。

测试这个行业就如同移动互联网一样发展迅速，我们完全可以去用“当我们还在学习，使用 xxx 的时候，yyy 已经成为了新的宠儿”这样的句式，相信所有的互联网从业人员都会有这样的感受。综合这些年所有人问我的问题，我总结两点在这里给大家分享：

- 在这样一个社会中，不要浪费时间在思考，实践才能够抓住“红利期”。
- 不要纠结于先有鸡，还是先有蛋。很多人纠结于自己没有这个，没有那个，所以不够级别去做一些事情。想做了就去做，我们不应该等到自己达到了一个等级才去做事情，而是要在做事情的过程中让自己达到对应的级别。

专项测试这个概念出现时间其实并不长，但其重要性和普及率都是非常高的。我自己也是最早做专项测试的人员之一，深知其中需要填坑无数。从 2015 年开始很多公司起步做专项，但对于具体的方法和策略以及专项测试基线往往都不是很清楚，导致专项的测试投入产出比不高，大家都期望能够有一种统一的标准和方法出现。

移动专项测试是不是只有大公司才需要做呢？答案肯定是“当然不是”，任何一个关心用户体验的企业都应该关心、重视专项测试。纵览全书，这可以说是至今为止我看到过最详细的专项测试宝典。从书中的内容我能感受到的不仅仅是腾讯 SNG 专项测试团队做专项测试的认真专业的态度，更多的是一种孜孜不倦的探索精神。书中涉及的内存、磁盘 I/O、电量、流量等方面的专项测试都会涵盖有案例、总结标准以及原理讲解。

再次感谢腾讯 SNG 专项测试团队能够为国内移动互联网行业产出这样一本专项测试宝典，我相信看到这本书的测试朋友都会像我一样欣喜若狂。在我看来，这本宝典不仅能够帮助更多企业的测试团队变得越来越专业，也对测试行业进步做出了不小的贡献。

书中最后提到，未来是什么？我们不是预言家，我们也不知道未来究竟是什么。但我们知道未来已经到来，你准备好了吗？在这样一个有的人每天在抱怨这个抱怨那个，有的人踏踏实实地在钻研技术，有的人有能力让影响力变现的时代，你是否明白自己要做什么？你想成为什么样的人？最后奉上我一直很喜欢的一句话，与大家共勉。

“It's not who I am underneath, but it's what I do that defines me”

——黑暗骑士

《大话移动 App 测试》系列作者 陈晔

前言

为什么会有这本书

记得笔者从微博和 MAC QQ 项目中解放出来后，就开始接手手机 QQ，组建专项测试团队。那时有几个小伙伴，我们一起做手机 QQ 的专项测试，发现推动专项问题解决非常困难。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。接手后的第一个手机 QQ 版本，涨了 10MB，这使我们看到了风险，顶着各部门的 KPI 需求，我们制定了一系列严厉的指标，超过的需求都不允许通过，从此安装包大小刹住了车。但 KPI 的压力巨大，像是洪水，不排解，堤坝只能越建越高，我们的压力也越来越大。产品经理开始不断地问，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的流量？Why？Why not？

在这些质疑中，我们经历了许多，除了工具、流程之外，更多带给我们的是真实的经验。例如，安装包不能再变大了。这里需要证据，运营同事找到了应用宝的数据，发现有不少用户是通过 3G 网络下载安装包的，另外安装包大小对下载失败率也有影响。在跟老板汇报过数据后，我们拍定了更严厉的标准：0 增长。慢慢地随着我们团队人数的增加，类似这样的故事也越来越多。跟大家想象的一样，其中有跟开发人员的 PK、有不服输自己去解决专项 Bug 的、有跟产品经理 PK 需求、与专项性能平衡的，等等。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被记录下来和分享出去，然后就有了本书。本书中会介绍工具、原理，但更重要的是提供了一个个真实的案例、Bug 解决方案。

谁适合阅读本书

以下职位的小伙伴们适合阅读本书。

- 终端专项测试：这个职位的测试人员，负责产品的性能、安全、稳定性、兼容性等各个方面。我们希望你通过阅读本书，可以有效地归纳总结知识、拓展思路，也可以作为你在专项测试领域的一本“字典”，随时翻查。
- 终端系统测试：这个职位的测试人员，需要全面负责功能测试、专项测试等各个

Android 移动性能实战

方面，利用合适的测试策略发现和预防风险。而专项测试是测试本身一个空间最广阔、蕴含知识最丰富的分支，学习和了解专项测试，对系统测试人员本身职业生涯的发展有着不可或缺的重要作用，也有利于制定出最合适的测试策略。

- 高级终端开发：终端开发人员必然需要面对许多性能上的难题，本书希望成为你的一部指南书。还有，必须要说，越是高级的终端开发人员，越是需要啃硬骨头，而专项恰巧就是这个硬骨头。

另外，产品经理不能看这本书吗？答案是能。因为不懂测试的开发不是好的产品经理。

如何利用本书

本书力求做到以下三点。

第一，通过结构化的知识体系，让读者在心中建立起性能专项的知识体系。希望做到“授之以渔”，所以我们会从资源类性能和交互类性能入手。

第二，案例均来自手机 QQ、QQ 空间、QQ 音乐等的真实项目案例，结合工具集和原理，希望让读者对其中的技巧和知识使用更加得心应手。

第三，提炼专项标准。在测试行业中，很多测试人员都需要有标准在背后支撑，特别是对于性能这些不黑不白的东西。虽然制定让人信服的标准很难，但我们愿意踏出这一步。

因此大家阅读的时候会发现，为了上面的三点，本书的大部分章节会分为原理、工具集、案例、专项标准四部分来介绍。

原理 主要是为了说明一些不脱离实际的实用的基础知识。因为有好的基础知识，才能 PK 得过开发人员，说服得了产品经理，用“专业知识”武装自己。

工具集工欲善其事，必先利其器。但工具那么多，该选哪个呢？根据我们的经验，本书中对工具做了不同纬度的分类，助你消灭选择恐惧症。

案例 按照分析专项问题的思路来划分我们的案例，我们力求做到让读者可以举一反三。

专项标准 会从原则、标准、优先级、来源等来描述。原则像是宪法，在没有对应的具体标准的时候，可遵循原则。标准更多是直接案例中提炼的规则，可直接操作落地。优先级和来源都是为了让推动标准的时候更有把握。

在开始性能专项之旅之前

为了坚定你把这本稍微晦涩难懂的书读完的信心，笔者必须让你弄清楚性能的重要性的这本书将会告诉你些什么。下面，先从几个不同的角度来谈谈性能的重要性。

首先，性能是基础功能。这句话不是我说的，是 Pony Ma 在一次大会上说的，即使不算终端性能，也都能印证这句话的正确性。其中最经典的例子就是 PC 的传文件功能。对于这个功能来说，在不同的网络环境下，尽可能地利用好带宽，保证成功率和提升传输速度就是这个功能的描述。而对性能的不断打磨，也让这个功能成为用户使用 QQ 的重要原因之一。所以产品经理要升级，要打怪通关，怎么能忽略性能呢？

其次，性能可以给予更多丰富用户体验的空间，也可以彻底破坏用户体验。这里举两个例子。第一个例子，过年时候，上了一个有强迫症的功能，口令红包。这个功能就相当于一次对于客户端的消息压测，会带来前所未有的性能压力，幸好聊天窗口的性能还不错，才能承载起来。第二个例子，内存中 OOM 会带来 crash，卡顿到了极端会 ANR，这些都会严重破坏用户体验。

最后，性能可以直接跟钱产生关系，可以省钱也可以费钱。关于省钱，例如手机 QQ 的部分业务功能切换为使用 WebP 来压缩图片，这不仅节省了用户流量，更重要的是从带宽费用上为公司节省了不少支出。关于费钱，例如 http content length 设置错误带来的重复下载，就会浪费用户流量，甚至可能导致一次让公司损失大量金钱的事故。

移动专项性能是一个完整的体系，如图 1 所示的 Android 性能专项地图，它涉及很多方面知识，作为移动专项的一个重要分支，包括资源类性能、交互类性能两个方面，所以本书将从这两个方面，依据图 1 中的脉络，讲述这些重要的案例、经验和工具，让你快速成长。

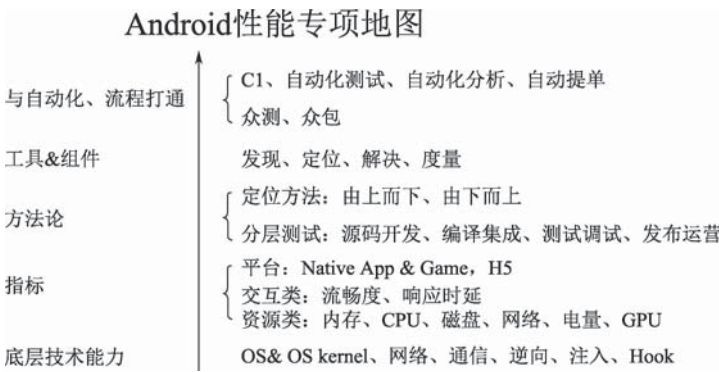


图 1

Android 移动性能实战

致谢

本书的作者是来自腾讯 SNG 专项测试团队的工程师们，他们负责手机 QQ、QQ 空间、QQ 音乐等的性能评测与优化工作，在 App 的资源类性能、交互类性能的分析与优化上挖掘很深，积累了不少案例和经验。

主要编著成员有：黄闻欣、杨阳、丁铎、谭力、付越、付云雷、黄天琳、欧阳霞、唐志彬、樊林。

感谢吴凯华、肖衡、邱俊、汪斐、石延龙、张金旭、闫石、潘在亮、刘海锋、周文乐、李昶博和其余专项测试团队成员的鼎力支持。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），您即可享受以下服务。

- 提交勘误：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 与作者交流：在页面下方【读者评论】处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31064>

二维码：



目录

第 1 部分 资源类性能

第 1 章 磁盘：最容易被忽略的性能洼地	2
1.1 原理	2
1.2 工具集	6
1.3 案例 A：手机 QQ 启动有 10 次重复读写 /proc/cpuinfo	16
1.4 案例 B：对于系统 API，只知其一造成重复写入	18
1.5 案例 C：手机 QQ 启动场景下主线程写文件	19
1.6 案例 D：Object Output Stream 4000 多次的写操作	20
1.7 案例 E：手机 QQ “健康中心” 使用的 Buffer 太小	22
1.8 案例 F：手机 QQ 解压文件使用的 Buffer 太小	24
1.9 案例 G：刚创建好表，就做大量的查询操作	37
1.10 案例 H：重复打开数据库	39
1.11 案例 I：AUTOINCREMENT 可没有你想的那么简单	40
1.12 案例 J：Bitmap 解码，Google 没有告诉你的方面	45
1.13 专项标准：磁盘	48

第 2 章 内存：性能优化的终结者	50
2.1 原理	50
2.2 工具集	57
2.3 案例 A：内类是有危险的编码方式	103
2.4 案例 B：使用统一界面绘制服务的内存问题	106
2.5 案例 C：结构化消息点击通知产生的内存问题	109
2.6 案例 D：为了不卡，所以可能泄漏	110
2.7 案例 E：登录界面有内存问题吗	114
2.8 案例 F：使用 WifiManager 的内存问题	116
2.9 案例 G：把 WebView 类型泄漏装进垃圾桶进程	120
2.10 案例 H：定时器的内存问题	123
2.11 案例 I：FrameLayout.POSTDELAY 触发的内存问题	126
2.12 案例 J：关于图片解码配色设置的建议	129
2.13 案例 K：图片放错资源目录也会有内存问题	134
2.14 案例 L：寻找多余的内存——重复的头像	139
2.15 案例 M：大家伙要怎么才能进入小车库	144
2.16 Android 要纠正内存世界观了	149
2.17 专项标准：内存	152
第 3 章 网络：性能优化中的不可控因素	154
3.1 原理	154
3.2 工具集	157
3.3 案例 A：WebView 缓存使用中的坑	189
3.4 案例 B：离线包下载失败导致重复下载	196
3.5 案例 C：使用压缩策略优化资源流量	197
3.6 案例 D：手机 QQ 发图速度优化	202
3.7 案例 E：手机 QQ 在弱网下 PTT 重复发送	206

目录

3.8 专项标准：网络	208
第 4 章 CPU：速度与负载的博弈	210
4.1 原理	210
4.2 工具集	211
4.3 案例 A：音乐播放后台的卡顿问题	215
4.4 案例 B：要注意 Android Java 中提供的低效 API	216
4.5 案例 C：用神器 renderscript 来减少你图像处理的 CPU 消耗	218
4.6 专项标准：CPU	220
第 5 章 电池：它只是结果不是原因	221
5.1 原理	221
5.2 工具集	226
5.3 案例 A：QQWi-Fi 耗电	243
5.4 案例 B：QQ 数据上报逻辑优化	244
5.5 案例 C：动画没有及时释放	245
5.6 案例 D：间接调用 WakeLock 没有及时释放	246
5.7 案例 E：带兼容性属性的 WakeLock 释放的巨坑	251
5.8 专项标准：电池	253
 第 2 部分 交互类性能	
第 6 章 原理与工具集	255
6.1 原理	255
6.2 工具集	257
6.2.1 Perfbox 自研工具：Scrolltest	257
6.2.2 Systrace（分析）	260

Android 移动性能实战

6.2.3	Trace View (分析)	269
6.2.4	gfxinfo (分析)	271
6.2.5	Intel 的性能测试工具: UxTune (测评 + 分析)	273
6.2.6	Hierarchy Viewer (分析)	274
6.2.7	Slickr (测评 + 分析)	277
6.2.8	图形引擎分析神器——Adreno Profiler 工具使用说明	281
6.2.9	Chrome DevTool	286

第 7 章 流畅度: 没有最流畅, 只有更流畅 295

7.1	案例 A: 红米手机 QQ 上的手机消息列表卡顿问题	295
7.2	案例 B: 硬件加速中文字体渲染的坑	298
7.3	案例 C: 圆角的前世今生	305
7.4	案例 D: 让企鹅更优雅地传递火炬	312
7.5	案例 E: H5 页面卡顿, 到底是谁闯的祸	314
7.6	专项标准: 流畅度	320

第 8 章 响应时延: 别让用户等待 322

8.1	案例 A: Android 应用发生黑屏的场景分析	322
8.2	案例 B: “首次打开聊天窗口”之痛	324
8.3	专项标准: 响应时延	328

第 3 部分 其他事项

第 9 章 还应该知道的一些事儿 330

9.1	UI 自动化测试	330
9.2	专项竞品测试攻略	335
9.3	未来的未来	344

第 2 章

内存：性能优化的终结者

2.1 原理

那天几个小伙伴在讨论重复下载的流量问题，一个负责内存的小伙伴云雷，默默地走过来强行插入说，一切最后都会变成内存问题，如图 2-1 所示。然后大家相视一笑，为什么呢？想想，质疑重复下载问题，可以缓存到存储中；缓存到存储要读出来，就变成磁盘 I/O 问题；为了避免磁盘 I/O 问题怎么办，用内存缓存起来。什么都用内存缓存起来，App 的常驻内存就会很大，变成内存问题，甚至最后成为 OOM 的导火索。



图 2-1

这里我们就以 OOM 为起点，介绍 Android 内存的原理。Out of Memory，OOM，通常会在 decode 图片的时候触发，但不一定是 decode 图片的问题，因为也许它只是压垮骆驼的稍微大一点的稻草而已。那什么时候会压垮骆驼？在虚拟机的 Heap 内存使用超过堆内存最大值（Max Memory Heap）的时候，那么在这里大家需要理解的第一个概念就是 Dalvik（ART）虚拟机的堆内存最大值。

第 2 章 内存：性能优化的终结者

1. 虚拟机的堆内存最大值

在虚拟机中，Android 系统给堆（Heap）内存设置了一个最大值，可以通过 `runtime.getRuntime().maxmemory()` 获取，而据我们 2016 年 2 月的统计，大部分用户使用的手机的最大堆内存应该都设置在 64MB 以上，而 128MB 的手机份额也在飞速增加，估计是因为屏幕分辨率变大了，解码图片的内存消耗相应变大，所以给予每个 App 的最大堆内存也与日俱增。而游戏作为消耗内存的特殊存在，Android 开通了一个绿色通道，可以在 manifest 里面设置 `LargeHeap` 为 `true`。

从这里可以发现，你的 App 真的不可能完全使用 1GB、2GB 的内存，系统只分给你一小部分。分这么小内存有一个重要的原因，是 Android 默认没有虚拟内存。在内存资源稀缺的大背景下，为了保证在极端情况下，前台 App 和系统还能稳定运行，就只有靠 low memory killer 机制。

2. Low Memory Killer

下面引出另一个重要概念 Low Memory Killer，也是 App 消耗内存过大导致的另外一个结果。在手机剩余内存低于内存警戒线的时候，就会召唤 Low Memory Killer 这个劫富济贫的“杀手”在后台默默干活。这里只要记住一句话，App 占用内存越多，被 Low Memory Killer 处理掉的机会就越大。

如果 OOM 和 Low Memory Killer 都没有干掉你的 App，那也不代表 App 就没有内存问题。因为还有一类问题，会直接导致 App 卡顿，即 GC。

3. GC（Garbage Collection）

最简单的理解就是没有被 GC ROOT 间接或直接引用的对象的内存会被回收。在具体执行中，ART 和 Dalvik（>2.3）会有很多不同，如图 2-2 所示，并发 GC 的时候 ART 比 Dalvik 少了一个 stop-the-world 的阶段，因此 Dalvik 比 ART 更容易产生 Jank（卡顿），当然，无论 ART 还是 Dalvik 并发 GC 的 stop-the-world 的时间并不长。然而，糟糕的情况是 GC for Alloc，这个情况在内存不足以分配给新的对象时触发，它 stop-the-world 的时间因为 GC 无法并发而变得更长（虽然在 Android 3.0 中增加了局部回收（Partial），在 Android 5.0 中增加了新增回收（Sticky）优化了不少，但时间依然很长），如图 2-3 所示。

Android 移动性能实战

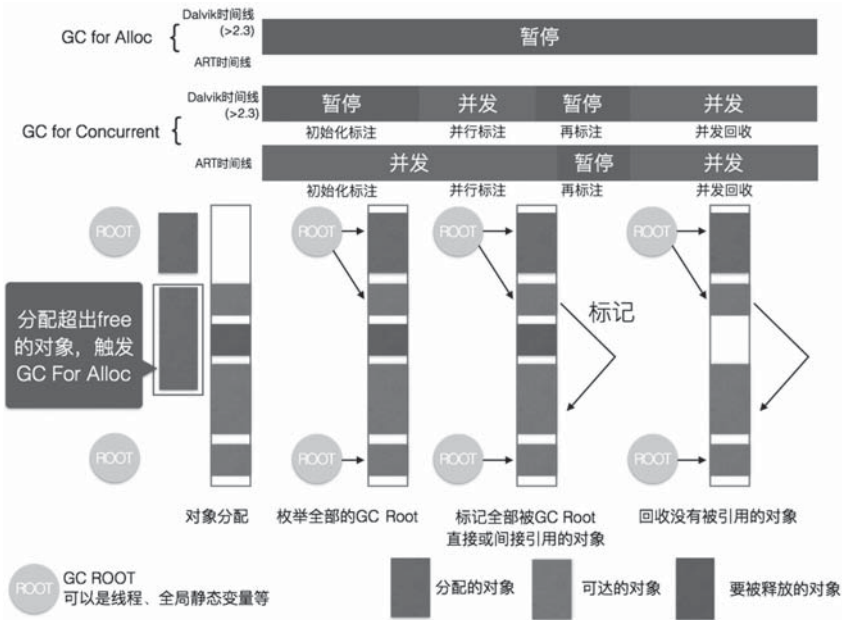


图 2-2

	<3.0	3.0-4.4	>5.0(ART)
GC原因	GC_FOR_MALLOC GC_EXPLICIT GC_EXTERNAL_ALLOC: BitmapHeap空间不足 GC_HPROF_DUMP_HEAP	+GC_CONCURRENT -GC_EXTERNAL_ALLOC	+Background +NativeAlloc +CollectorTransition +DisableMovingGC,HeapTrim +HomogeneousSpaceCompact
GC范围	全局(full)	+局部(Partial)	+新增(Sticky)
GC采集方式	Stop-the-world	+并发(Concurrent)	+并行(Parallel) +腾挪(Moving)
GC回收方式	标注-回收 (Mark-Sweep)	+并发标注-回收 (Concurrent Mark-Sweep)	减少碎片的新算法 +需要回收的空间压缩到一端 +移动被GC ROOT间接或者直接引用的对象到一个空间, 剩余在原空间统一删除
触发条件	没有空间分配对象	+触碰剩余内存的阈值则回收	+预估吞吐量回收 +预估剩余回收

图 2-3

第 2 章 内存：性能优化的终结者

那么说到底，我们还是要避免 GC FOR ALLOC，跟要避免 OOM 一样，关键是要管理好内存。什么是管理好内存？除了减少内存的申请回收外，更重要的是减少常驻内存和避免内存泄漏。说起内存泄漏，就必须提 Activity 内存泄漏。

4. Activity 内存泄漏

因为 Activity 对象会间接或者直接引用 View、Bitmap 等，所以一旦无法释放，会占用大量内存。案例部分将会介绍有不少关于 Activity 内存泄漏的案例。但是无论是什么案例，都离不开不同的 GC ROOT 对 Activity 的直接引用、this\$0 间接引用、mContext 间接引用，如表 2-1 所示。

表 2-1

引用的方式\GC ROOT	Class-（静态变量）	活着的线程	生命周期跟随 App 的特殊存在
mContext 间接引用	静态 View， InputMethodManager	SensorManager、WifiManager（其他 Service 进程都可以）	ViewRootImpl
this\$0 间接引用	内类引用	匿名类 /Timer/TimerTask/Handler	
直接引用	静态 Activity		

提示：大家学习完后面的案例之后，也不妨回到这里看一下。

那么另外一个情况就是内存常驻了，而通常在常驻内存中最大的就是图片。俗话说，互联网产品最讲究的体验精神，即“有图有真相”。但是这些图片在内存中的存储不合理会导致什么呢？

首当其冲的是 Crash 堆栈，如图 2-4 所示。

Android 移动性能实战



图 2-4

然后是疯狂 GC, 触发我们前面说到的 GC for Alloc, 导致 Stop-the-world 的“卡”, 如图 2-5 所示。

第 2 章 内存：性能优化的终结者

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

图 2-5

最后是功能异常，有损体验：内存没了，图还要加载，如图 2-6 所示。



图 2-6 内存没了，图还要加载

Android 移动性能实战

当然，以上的损害都说明，我们将“大卡车”停进的“内存”造成危害。既然有这么多的损害，为什么不能把图片下载来都放到磁盘（SD Card）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，可以简单地理解，Android 系统要在屏幕上展示图片的时候只认“像素缓冲”，而这也是大多数操作系统的特征。而我们常见的 jpg、png 等图片格式，都是把“像素缓冲”使用不同的手段压缩后的结果，所以相对而言，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。

5. 图片缓存

两害相权取其轻，官方建议使用 LRU 算法来做图片缓存，而不是之前推荐的 WeekReference，因为 WeekReference 会导致大量 GC。原理示意图如图 2-7 所示。

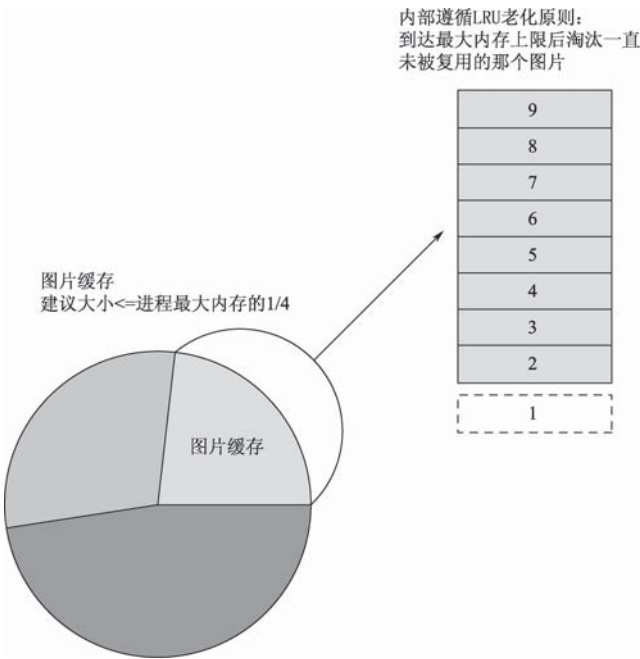


图 2-7

第 2 章 内存：性能优化的终结者

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，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 算法做淘汰，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。但是对于许多图片类 App，内存对于它们实在是捉襟见肘，因此官方有两个非常著名的硬盘缓存方案。

1. DiskLruCache（<https://android.googlesource.com/platform/libcore/+/-/jb-mr2-release/luni/src/main/java/libcore/io/DiskLruCache.java>，<https://github.com/JakeWharton/DiskLruCache>），简单理解就是 LruCache 的硬盘版本，容错性强，但是对比 BlobCache，I/O 性能很一般。

2. BlobCache（http://androidxref.com/6.0.1_r10/xref/packages/Apps/Gallery2/gallerycommon/src/com/android/gallery3d/common/BlobCache.java），这个方案源自 Android 原生的相册，仅仅利用三个文件，包括索引（Index）文件、活动（Active）文件和非活动（Unactive）文件，通过 `FileInputStream().getChannel().map()` 把索引文件直接映射到内存，通过索引（其实就是偏移）来读取活动文件中的图片缓存。清除旧图片的方法简单粗暴，直接 seek 到文件头部，覆盖写入就可以。这一切都是为了用最小的磁盘 I/O 代价完成磁盘缓存。

另外官方也建议，把从内存淘汰的图片，降低压缩比存储到本地，以备后用。这样就可以最大限度地降低以后复用时的解码开销。

现在我们来归纳一下，内存问题主要包括常驻问题（主要是图片缓存）、泄漏问题（主要是 Activity 泄漏）、GC 问题（关键是 GC For Alloc），后果会导致 App Crash、闪退、后台被杀、卡顿，而且这是各种资源类性能问题积压的最后一环。因此可见其重要性，下面，我们来看看有什么好工具和实用的案例可以帮助我们理解和解决这些终极性能问题。

2.2 工具集

这里要特别强调，Android 关于内存的工具不少，如表 2-2 所示，灵活地选择工具就显得特别重要。我们特别推荐涵盖一定初步定位和定位能力的工具，可以让我们一步到位地剖析问题、提升效率。实在有些场景无法覆盖，这时可以使用仅有“发现”能力的工具。

Android 移动性能实战

表 2-2

工 具	问 题	能 力
top/procrank	内存占用过大,内存泄漏	发现
STRICTMODE	Activity 泄漏	发现
meminfo	Native 内存泄漏、是否存在 Activity、ApplicationContext 泄漏、数据库缓存命中率低	发现 + 初步定位
MAT、Finder、JHAT	Java 层的重复内存、不合理图片解码、内存泄漏等	发现 + 定位
libc_malloc_deBug_leak.so	Native 内存泄漏 (JNI 层)	发现 + 定位
LeakCanary	Activity 内存泄漏	自动发现 + 定位
StrictMode	Activity 内存泄漏	自动发现 + 初步定位
APT	内存占用过大,内存泄漏	发现
GC Log from Logcat、GC Log 生成图表	人工触发 GC for Explicit 而导致的卡顿, Heap 内存不足触发 GC for Alloc 而导致的卡顿	发现 + 初步定位
Systrace	GC 导致的卡顿	发现
Allocation Tracer	申请内存次数过多和过大、辅助定位 GC Log 发现的问题	发现 + 定位
chrome devtool	HS 的内存问题	发现 + 定位

1. top/procrank

得到内存曲线的方法很多, top 就是其中一种, 但是很遗憾, 它的输出列信息 (如图 2-8 所示) 中只包含 RSS 与 VSS, 所以 Android 中 Top 的使用更多地集中在某个进程的 CPU 负载方面。下面借着介绍 top 的契机, 说明 VSS、RSS、PSS、USS 的含义。

VSS: Virtual Set Size 虚集合大小。

RSS: Resident Set Size 常驻集合大小。

PSS: Proportional Set Size 比例集合大小。

USS: Unique Set Size 独占集合大小。

第 2 章 内存：性能优化的终结者

PID	Vss	Rss	Pss	Uss	cmdline
871	1127632K	146220K	113053K	109588K	com.android.systemui
764	1118200K	91996K	53031K	48104K	system_server
1886	1278432K	88192K	49528K	45136K	com.tencent.android.qqdownloader
1092	1122192K	83492K	44669K	37904K	com.android.launcher3
1698	1166252K	82616K	39727K	32288K	com.google.android.gms.persistent
2813	1230140K	78052K	39565K	35000K	com.google.android.googlequicksearchbox:search
1717	1275336K	78704K	36543K	29280K	com.google.android.gms
932	1293096K	56956K	23560K	21232K	com.tencent.mobileqq:MSF
2224	1051816K	55016K	23145K	21192K	com.tencent.weread:gap
2252	1197636K	52704K	19084K	16836K	com.taptap

图 2-8

RSS 与 PSS 相似，也包含进程共享内存，但有一个麻烦，RSS 并没有把共享内存大小平分到使用共享的进程头上，以至于所有进程的 RSS 相加会超过物理内存很多。而 VSS 是虚拟地址，它的上限与进程的可访问地址空间有关，和当前进程的内存使用关系并不大，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 A Size 加 B Size，而至于内存是什么属性，它并不关心。所以很多 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，它与进程使用内存并没有多少关系。

而 PSS、USS 最大的不同在于“共享内存”（比如两个 App 使用 MMAP 方式打开同一个文件，那么打开文件而使用的这部分内存就是共享的），USS 不包含进程间共享的内存，而 PSS 包含。这也造成了 USS 因为缺少共享内存，所有进程的 USS 相加要小于物理内存大小的原因。

最早的时候官方推荐使用 PSS 曲线图来衡量 App 的物理内存占用，同时，用户在原生的 Android 操作系统上唯一能看到的内存指标（在“设置 - 应用程序 - 正在运行的某程序”）就是 PSS，而 Android 4.4 之后加入 USS（如图 2-8 所示）。

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 SO 库，那么 SO 库中初始化所用的那部分内存就会被平分到 A 与 B 的头上。但是 A 是在 B 之后启动的，那么对于 B 的 PSS 曲线图而言，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的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，这些都会影响曲线的异常波动，比如异步 GC 是 Android 4.0 以上系统很重要的新特性，

Android 移动性能实战

但是 GC 什么时候结束？曲线什么时候“降”？就变得很诡异了，而测试人员通常希望退出某个界面后可以明显地看到曲线有大的降落。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，为 Java 代码运行时预留，避免 Java 运行时再申请而造成卡顿），但是这个预申请大小是动态变化的，这也会造成 USS 忽大忽小。

另外：

（1）Android 4.4 以后增加了一个新的名词，被称为 ProcessStats（PS），用于反映内存负载，其最终计算也用到了 PSS。

（2）PROCRAK 在模拟器里面会存在，而大多数真机则没有。不过下面介绍的 Meminfo 才是正道，所以也不多介绍了。

2. meminfo

介绍完 top/procrank，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 `dumpsys meminfo`。

meminfo 的使用如下：

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

其中，-h 帮助信息。

-a 打印所有进程的内存信息，以及当前设备的内存概况。

--oom 按照 OOM Adj 值进行排序。

[process] 可以是进程名称，也可以是进程 id，用于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，如图 2-9 和图 2-10 所示，这两张图由三个部分构成。

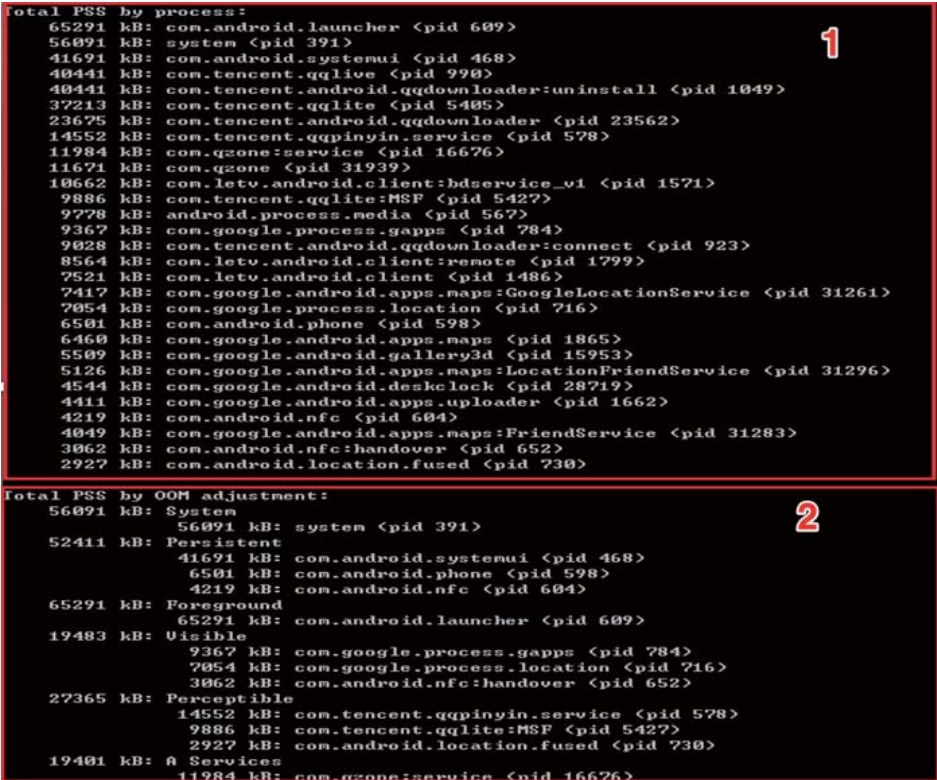


图 2-9

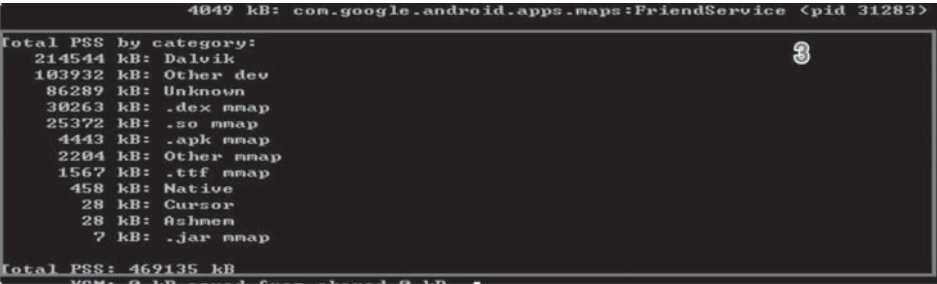


图 2-10

- (1) 按照 PSS 排队，用于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不相上下的 PSS。
- (2) OOM Adj 排队，展示当前系统内部运行的所有 Android 进程的内存状态和被杀

Android 移动性能实战

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，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，没过多久出现竞品的排名在被测产品上方的情况（即被测产品退回后台更容易被系统杀掉，不过 Android 4.4 以后出现了“内存负载”概念，这也不一定是坏事了）。

（3）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，也可以查看物理内存的使用是否已接近物理内存的最大值。

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，如图 2-11 和图 2-12 所示。

MEMINFO in pid 12477 (com.tencent.mobileqq) **

	Pss	Private	Private	Swapped	Heap	Heap	Heap
	Total	Dirty	Clean	Dirty	Size	Alloc	Free
Native Heap	2671				27433		25814
Dalvik Heap	58844	58180	0	0	73285	71417	1868
Dalvik Other	1392	1392	0	0			
Stack	1112	1112	0				
Other dev	20	0	20	0			
.so mmap	1353	380	184	0			
.apk mmap	130	0	84	0			
.dex mmap	8730	0	8840	0			
.oat mmap	1655	0	344	0			
.art mmap	2088	1584	40	0			
Other mmap	16	4	0	0			
Unknown	246	244	0	0			
TOTAL	102202	80540	9712	0	126533	98850	27682

Views: 407 ViewRootImpl: 1
AppContexts: 5 Activities: 1
Assets: 8 AssetManagers: 8
Local Binders: 74 Proxy Binders: 38
Parcel memory: 54 Parcel count: 216
Death Recipients: 2 OpenSSL Sockets: 3
Bitmaps(preloaded): 446 Bitmaps(client): 250
WebView: 0

Dalvik
isLargeHeap: false

图 2-11

第 2 章 内存：性能优化的终结者

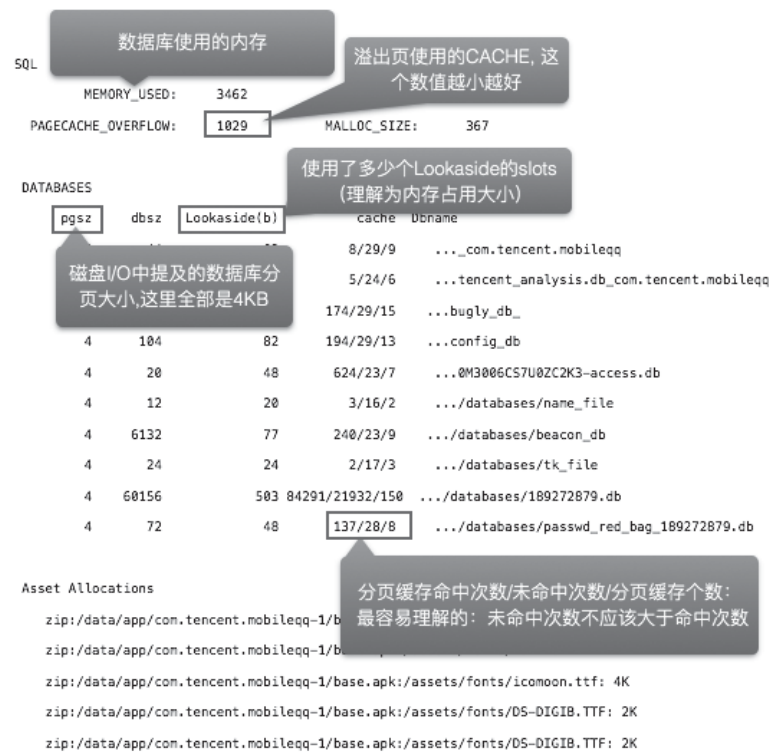


图 2-12

另外，配合命令行 `watch` 来观察 `meminfo` 也是不错的选择，例如，每隔 5 秒刷新一次，`watch -n 5 dumphys meminfo com.tencent.mobileqq`。

3. Procstats

从基础的曲线图 PSS 引出了 `meminfo`，那么怎样才能把 `meminfo` 这样一个“点”上的值变成一个“统计”的值呢？绘制二维曲线是一种方法（纵轴 PSS、横轴时间），但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 被提了出来，那么负载是怎么计算的呢？见如下公式：

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 在设备上的运行状态一致）时长，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

Android 移动性能实战

观展示它们，就有了 Procstats 工具（如图 2-13 所示），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布于众。



图 2-13

Procstats 的查看方式很简单，有如下两种。

（1）每个进程后面都有一个百分比数值，它用于统计“此状态下的”运行时间，所谓的此状态即上文所述的——前台、后台、缓冲。默认展示的是“后台负载”，所以按照图 2-13 所展示的，QQ MSF 进程在被统计的 2 小时 6 分钟内，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，由图 2-13 可见，微信的内存负载高过 QQ 空间。

第 2 章 内存：性能优化的终结者

引申一下，三个状态中对于 Android 系统来说，有如下的潜规则。

（1）对于前台而言，是用户正在使用的，所以这部分内存一定会保证，是用户不关注的，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并回收内存，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），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，可以通过竞品对比查看哪个 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），认可度高的 App 相比认可度低的 App 而言肯定更容易被用户所青睐。

4. DDMS (Monitor)

DDMS 的全称是：Dalvik DeBug Monitor Server，即 Dalvik 虚拟机调试监控服务，其界面如图 2-14 所示。

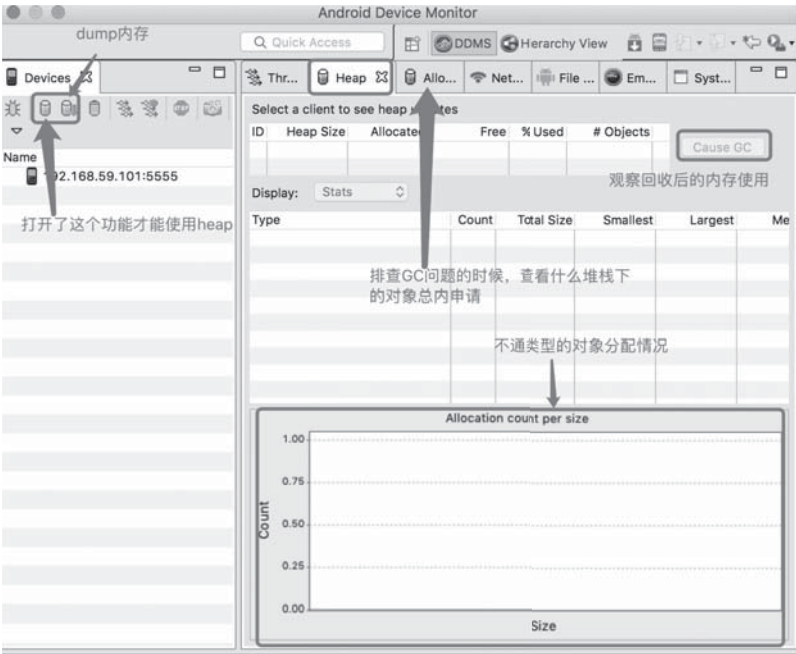


图 2-14

Android 移动性能实战

DDMS 是一个调试信息合集，里面包含时延、内存、线程、CPU、文件系统、流量等一系列信息的获取和展示，其中和内存相关的要提到两个功能：Update Heap、Allocation Tracker，以及内存快照 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，还有一个分配大小的分布柱状图，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（如图 2-15 所示），官方推荐用它来提升流畅度。因为申请内存多，GC 就多，而正如原理所介绍的，GC 会挂起全部线程，引入卡顿问题。

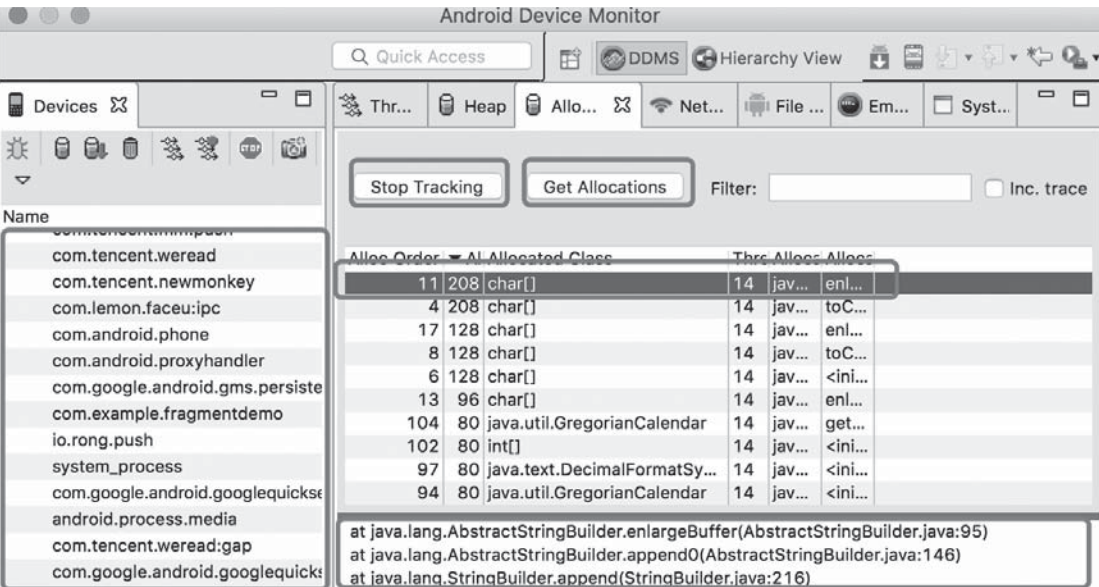


图 2-15

Dump Hprof file：用于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 部分中介绍。

另外，在 Android Studio 也有类似的功能，而且更加好用，其截图如 2-16 所示。

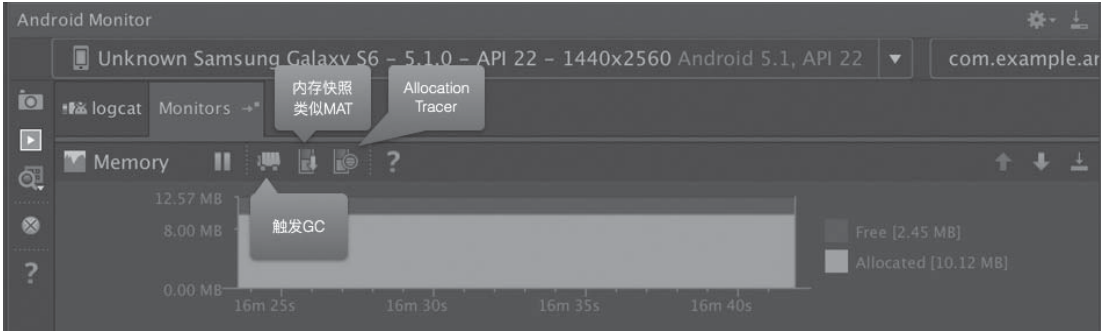


图 2-16

5. MAT

MAT 的全称为 Memory Analyzer（内存分析器），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，而沿用了 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，前文说的 DDMS 的 Dump Hprof file 功能就是其中最简单、通用性最强的一种。以下还有两种不常用的方法。

- （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 Android 4.x 以上系统已经不支持这条命令了，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 文件不能直接用于 MAT，需要通过 Android SDK Tools 中的 Hprof-conv 工具转换一下，才能用于 MAT。不过在安装 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 2-17 所示的界面。

Android 移动性能实战

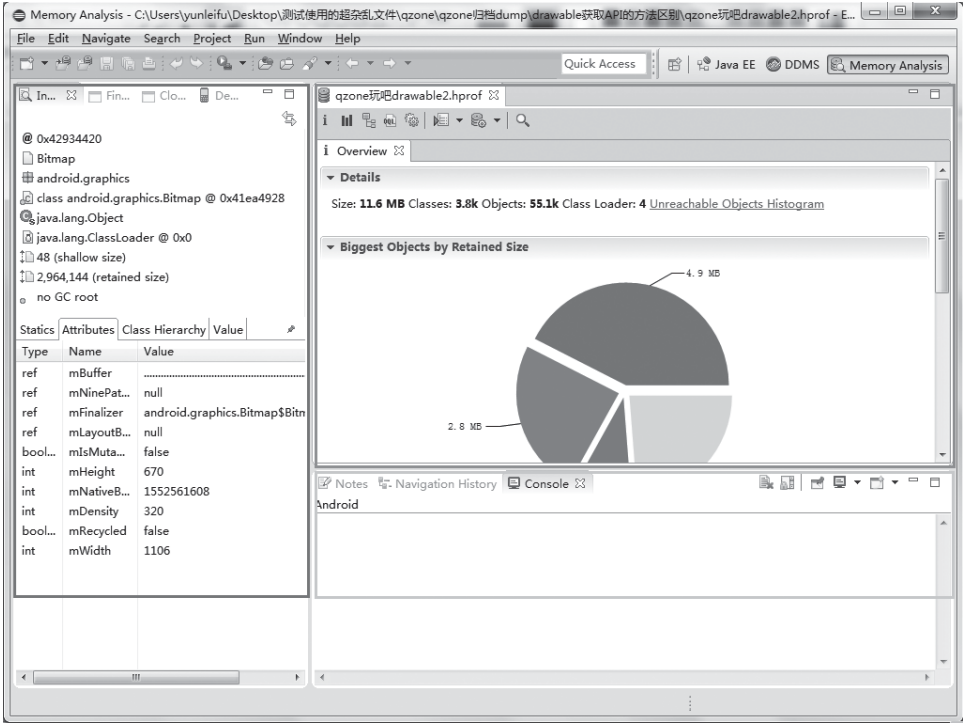


图 2-17

Insepector: 左侧框出来的部分，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: 右侧上方框出来的部分，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: 在右侧下方框出来部分，可以看到操作记录、工作日志、命令行信息等，可以辅助书写工作摘要。

在使用 MAT 前有几个重要的知识点需要掌握。

- (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，比如在手机 QQ 聊天窗口场景下，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，主要包含如图 2-18 所示的几个方面。

第 2 章 内存：性能优化的终结者



图 2-18

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，所以有些信息没有做得足够详细、易用，比如统治者视图，虽然它是 MAT 的重要组件，但是里面的信息做过大量过滤，如果过度依赖它来发现问题，很可能出现遗漏测试的悲剧。更多有关 MAT 的使用方法，因为其功能过多就不在这里一一详述了，会在后续案例中有更加生动、详尽的描述。

6. 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是一场噩梦。Finder 是我们基于 MAT 进行的二次开发，主要为 Android 内存测试人员提供一个更系统化、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，其界面如图 2-19 所示。

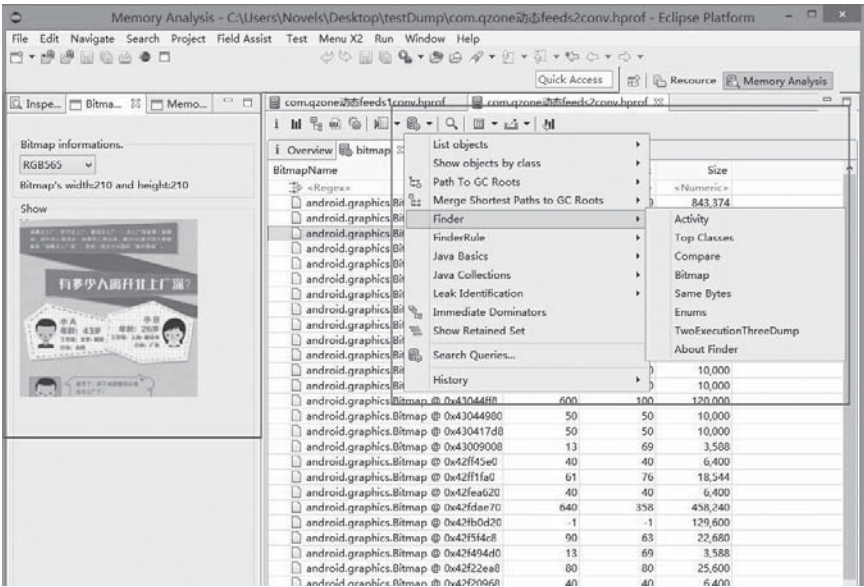


图 2-19

Android 移动性能实战

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 中的所有 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”用于分析三个 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。

右边 FinderRule 菜单栏下扩展区菜单是 Run Memory rules（执行内存规则），主要配合 Memory Rule 使用。

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 → Other，调出 FinderView（Bitmaps View），用于查看某个 Bitmap 对象中的图像，如图 2-20 所示。

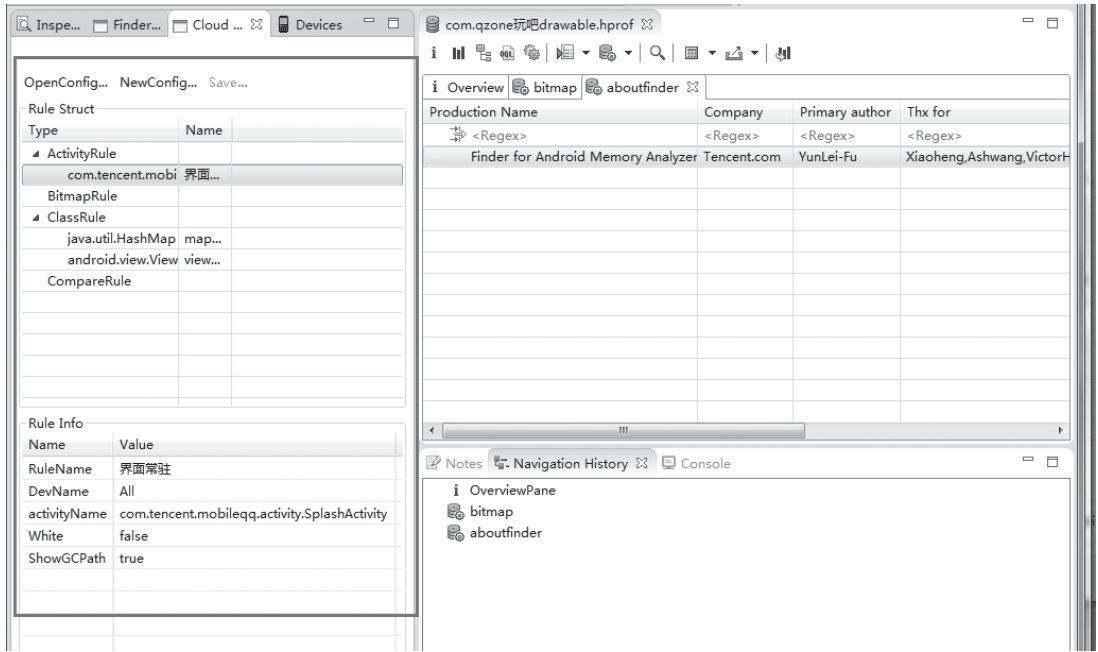


图 2-20

第2章 内存：性能优化的终结者

图 2-20 左边功能区展示的是 Finder 中另一个很好用的功能 Memory Rule：通过 Eclipse 的 Windows → Show View → Other，调出 FinderView（Memory rules View），它用来积累用例和类对象规则，比如登录成功后内存中应该存在多少 map 对象。

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达 80% 以上）。
- （2）更够发现细微的内存问题（通过曲线工具发现问题后，不太敏感，常常出现小泄漏测试不出来，但是在用户那里频繁爆发的情况）。
- （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，大量地了解产品架构，才能开始内存测试工作，且测试效率基本上为两个工作日一个需求，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，就开始内存测试工作，且半天就可以测试完成一个需求）。

7. 推荐 LeakCanary

LeakCanary 是 Square 出品的一款非常优秀的 Activity 内存泄漏检测工具。值得注意的是，在 `leakcanary/leakcanary-android/src/main/java/com/squareup/leakcanary/AndroidExcludedRefs.java` 这个文件中，定义了不少 Android 系统导致的内存泄漏坑和解决的黑科技，其中就包括 `InputMethodManager` 泄漏等。

8. LeakInspector

天网是 Android 手机经过长期积累和提炼，集内存泄漏检测、自动修复系统 Bug、自动回收已泄漏 Activity 内资源、自动分析 GC 链、白名单过滤等功能于一体，并深度对接研发流程、自动分析责任人并提缺陷单的一站式内存泄漏解决方案。前面推荐了 Square 开源的内存泄漏检测组件 LeakCanary，与之相比有什么不同呢？LeakInspector 与 LeakCanary 两个工具的功能对比如表 2-3 所示。

表 2-3

对比项		LeakInspector	LeakCanary
基础功能	Activity 泄漏检测	✓	✓
	自定义对象泄漏检测	✓	✓
	位图检测	✓	✗
	显示泄漏对象	✗ 只显示类名	✓
	泄漏提醒	✓	✓
	自动 dump	✓	✓
	Hprof 分析	✓ 云端分析	✓ 本地分析
	提单跟进	✓	✓
	白名单配置	✓ 动态配置	✓ 源码写死
	配合自动化	✓	✗
兜底功能	修复系统泄漏	✓	✗
	回收资源	✓	✗

第一，检测能力与原理方面不同

（1）检测能力

两个工具都支持对 Activity、Fragment 以及其他自定义类（比如 QQAppInterface）泄漏的检测，但 LeakInspector 还有针对 Bitmap 的检测能力：

①检测有没有在 View 上 decode 超过该 View 尺寸的图片，若有则上报出现问题的 Activity 及与其对应的 View id，并记录它的个数与平均占用内存的大小。

②检测图片尺寸是否超过所有手机屏幕大小，违规则提单。

而 LeakCanary 没有这个能力。

（2）检测原理

检测原理图如图 2-21 所示，两个工具的泄漏检测原理都是在 onDestroy 时检查弱引用，不同的地方在于 LeakInspector 使用 WeakReference 来检测对象是否已经释放，而 LeakCanary 使用 ReferenceQueue，两者效果没什么区别。

第 2 章 内存：性能优化的终结者

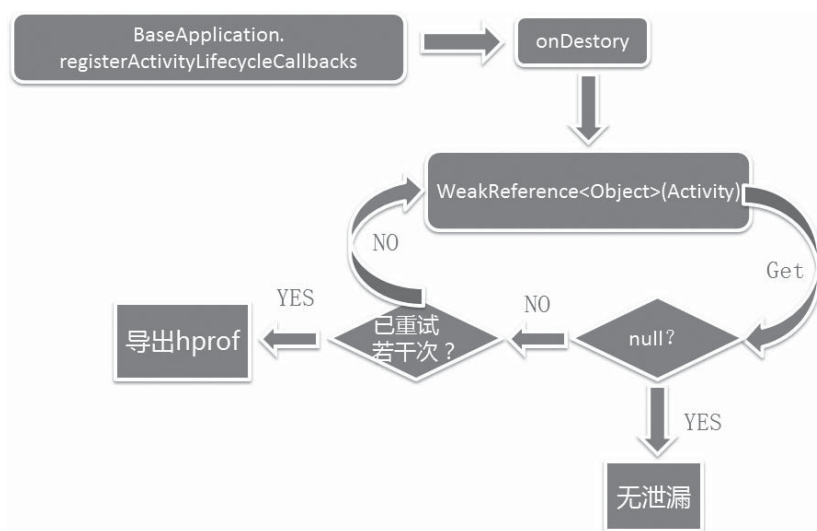


图 2-21

针对 Activity，如何实现在 `onDestory` 时启动监控呢？在 Android 4.0 以上系统中，两者都通过注册 Activity 的生命周期，重写 `onActivityDestroyed` 方法实现。然而在 Android 4.0 以下的系统上，LeakCanary 需要手动在每一个 `Activity.onDestroy` 中添加启动检测的代码，而 LeakInspector 反射了 `Instrumentation` 来截获 `onDestory`，接入时修改成本更低。

下面我们用自己的 `MonitorInstrumentation` 替换系统原来的 `Instrumentation` 对象代码，如图 2-22 所示。

```
Class<?> clazz = Class.forName("android.app.ActivityThread");
Method method = clazz.getDeclaredMethod("currentActivityThread", null);
method.setAccessible(true);
sCurrentActivityThread = method.invoke(null, null);
Field field = sCurrentActivityThread.getClass().getDeclaredField("mInstrumentation");
field.setAccessible(true);
field.set(sCurrentActivityThread, new MonitorInstrumentation());
```

图 2-22

第二，泄漏现场处理方面不同

(1) dump 采集

两者都能采集 dump，但 LeakInspector 提供了回调方法，能让用户增加自定义信息，比如运行时 LOG、TRACE、DUMPSYS 等信息，辅助分析定位问题，如图 2-23 所示。



图 2-23

（2）白名单定义

所谓的白名单，主要是为了处理一些系统引起的泄漏问题，以及一些因为业务逻辑需要开后门的情形而设置的。分析时如果碰到白名单上标识的类，则不对这个泄漏做后续处理。

二者的配置有以下两个差异。

① LeakInspector 的白名单以 XML 配置的形式存放在服务器上。

- 优点：跟产品（甚至用例）绑定，测试、开发同学可以很方便地修改相应配置。
- 缺点：白名单里的类不区分系统版本一刀切。

LeakCanary 的白名单写死在其源码的 AndroidExcludedRefs.java 类里，如图 2-24 所示。

- 优点：定义非常详细，区分系统版本。
- 缺点：每次修改必定得重新编译。

```

<GCWhiteNode>
...<Rule name="WindowInputEventReceiver">
...<Rule name="LoadedApk">
...<Rule name="ViewConfiguration">
...<Rule name="PhoneFallbackEventHandler">
...<Rule name="DecorView">
...<Rule name="QzoneGPUPluginProxyActivity">
...<Rule name="FinalizerWatchdogDaemon">
...<Rule name="IClipboardDataPasteEventImpl">
</GCWhiteNode>

```

分析云配置

与

LeakCanary配置

```

.RESOURCES_MCONTEXT(SAMSUNG.equals(MANUFACTURER) && SDK_INT == KITKAT) {
    @Override void add(ExcludedRefs.Builder excluded) {
        // In AOSP the Resources class does not have a context.
        // Here we have ZygoteInit.mResources (static field) holding on to a Resources instance that
        // has a context that is the activity.
        // Observed here: https://github.com/square/leakcanary/issues/1#issue-74450184
        excluded.instanceField("android.content.res.Resources", "mContext");
    }
},

.VIEW_CONFIGURATION_MCONTEXT(SAMSUNG.equals(MANUFACTURER) && SDK_INT == KITKAT) {
    @Override void add(ExcludedRefs.Builder excluded) {
        // In AOSP the ViewConfiguration class does not have a context.
        // Here we have ViewConfiguration.sConfigurations (static field) holding on to a
        // ViewConfiguration instance that has a context that is the activity.
        // Observed here: https://github.com/square/leakcanary/issues/1#issuecomment-100324683
        excluded.instanceField("android.view.ViewConfiguration", "mContext");
    }
},

```

图 2-24

② LeakCanary 的系统白名单里定义的类比 LeakInspector 方案中定义的多很多，因为它没有下面所述的自动修复系统泄漏功能。

(3) 修复系统泄漏

针对系统泄漏，LeakInspector 进行了预处理，通过反射自动修复目前碰到的一些系统泄漏，只要在 onDestroy 里调用一个修复系统泄漏的方法即可。LeakCanary 也能识别系统泄漏，但它仅仅对该类问题给出了分析，没有提供实际可用的解决方案。

(4) 回收资源

如果已经发生了泄漏，LeakInspector 会对整个 Activity 的 View 进行遍历，把图片资源等一些占内存的数据释放掉，保证此次泄漏只会泄漏一个 Activity 的空壳，尽量减少对内存的影响。而 LeakCanary 没有类似逻辑，只能依赖人工修改来解决内存问题。

LeakInspector 回收资源的大体方法如图 2-25 所示。

Android 移动性能实战

```
if (view instanceof ImageView) {
    //ImageView ImageButton都会走这里
    recycleImageView(app, (ImageView) view);
} else if (view instanceof TextView) {
    //释放TextView、Button周边图片资源
    recycleTextView((TextView) view);
} else if (view instanceof ProgressBar) {
    //ProgressBar
    recycleProgressBar((ProgressBar) view);
} else {
    //ListView
    if (view instanceof android.widget.ListView) {
        recycleListView((android.widget.ListView) view);
    } else if (view instanceof FrameLayout) {
        recycleFrameLayout((FrameLayout) view);
    } else if (view instanceof LinearLayout) {
        recycleLinearLayout((LinearLayout) view);
    }
}

if (view instanceof ViewGroup) {
    recycleViewGroup(app, (ViewGroup) view);
}
```

图 2-25

以 recycleTextView 为例，我们回收资源的方式如图 2-26 所示。

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

图 2-26

第 2 章 内存：性能优化的终结者

第三，后期处理（如图 2-27 所示）方面不同

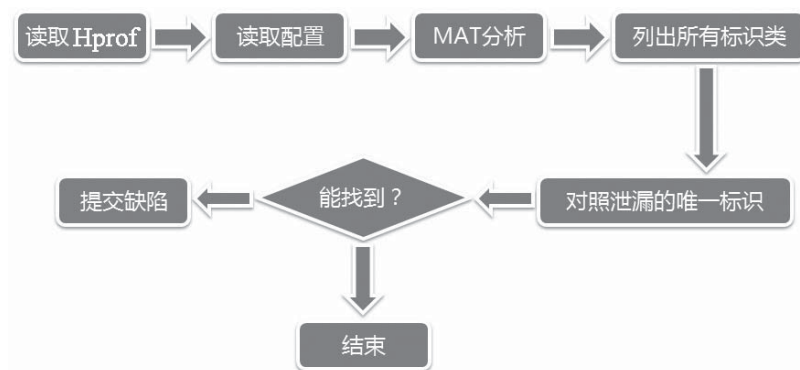


图 2-27

（1）分析与展示

采集 dump 之后，LeakInspector 会自动通过 Magnifier 上传 dump 文件，并调用 MAT 命令行来进行分析，得到这次泄漏的 GC 链；LeakCanary 则用开源组件 HAHA 来分析，同样返回一条 GC 链。

从分析过程来看，两者都不需要用户介入。但是整个分析流程比较耗时，LeakInspector 将分析放在服务器上，极大地减轻了手机的负担，而且马上能开始下一次测试；LeakCanary 连分析也放在手机上做，此时基本无法执行其他测试，只能等分析完毕。

从获取的 GC 链展示来看，LeakCanary 得到的 GC 链包含被 hold 住的类对象，用户很可能不需要用 MAT 打开 Hprof 即可解决问题；而 LeakInspector 方案得到的 GC 链里只有类名，用户还得用 MAT 打开 Hprof 看才能定位问题，有点不方便，如图 2-28 所示。

Android 移动性能实战



图 2-28

(2) 后续跟进闭环

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

第四，配合自动化测试方面不同

LeakInspector 方案跟自动化测试可以无缝结合，当自动化脚本执行过程中发现内存泄漏，即由它采集 dump，然后发送到服务器进行分析，生成 JSON 格式的结果，最后提单，整个流程一气呵成无须人力。而 LeakCanary 把分析结果通过通知栏告知用户，该结果无法传到自动化下一个流程，必须有人工介入。

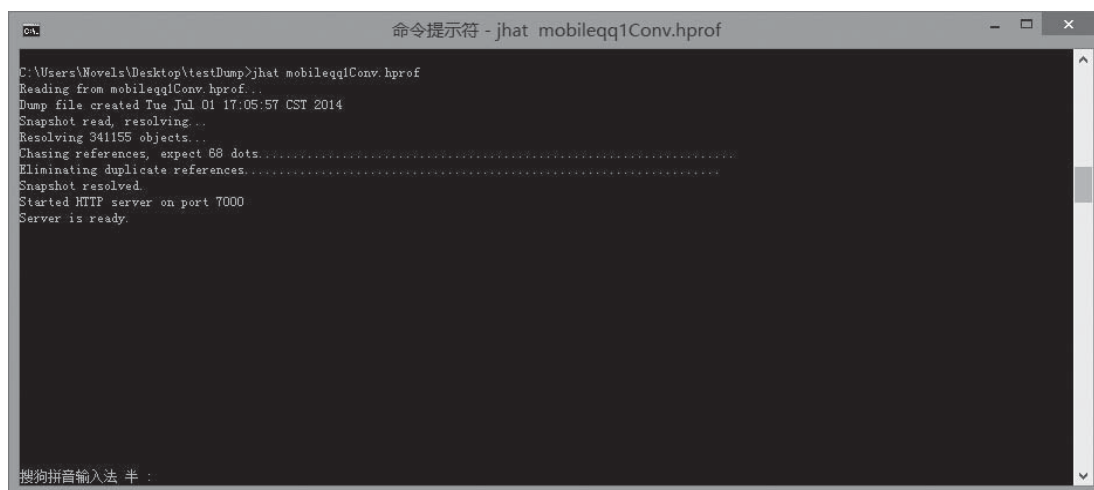
9. JHat 节点

JHat 是 Oracle 推出的一款 Hprof 分析软件，它和 MAT 并称为 Java 内存静态分析利器，

第 2 章 内存：性能优化的终结者

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）。因为这款软件是 Oracle 推出的，也就是传统意义上的“正统”软件，所以 jhat 被置于 JDK 中，安装了 JDK 的读者，设置好 Java_Home 与 Path 后，在命令行中输入 jhat 命令可查看有没有相应的命令。

JHat 的使用如图 2-29 所示。



```
C:\Users\Novels\Desktop\testDump>jhat mobileqq1Conv.hprof
Reading from mobileqq1Conv.hprof...
Dump file created Tue Jul 01 17:05:57 CST 2014
Snapshot read, resolving...
Resolving 341155 objects...
Chasing references, expect 68 dots...
Eliminating duplicate references...
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

图 2-29

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。执行结果如图 2-30 所示。

Android 移动性能实战

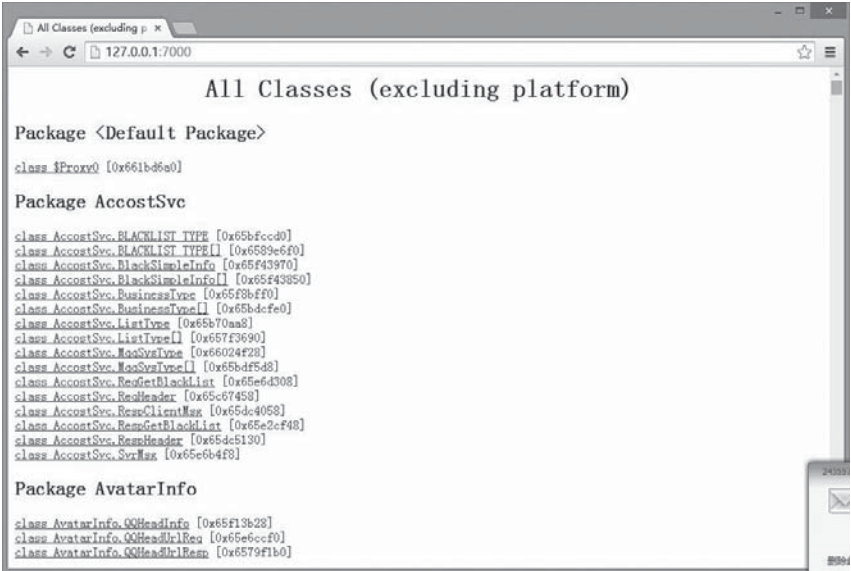


图 2-30

JHat 还有两个比较重要的功能分别如下。

(1) 统计表，如图 2-31 所示。

Heap Histogram

All Classes (excluding platform)

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

图 2-31

第 2 章 内存：性能优化的终结者

(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), 如图 2-32 所示。

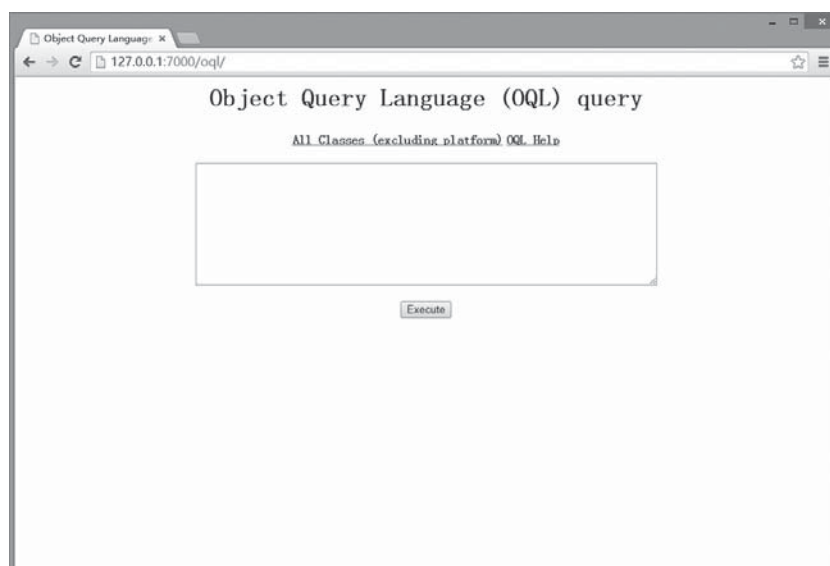


图 2-32

对于中小型团队, 建议就不要考虑 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, 并不非常适合中小型高效沟通型团队使用。

10. libc_malloc_deBug_leak.so

Android 构建在一个被精简的 Linux 上, Dalvik 虚拟机是一个 Java 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 为了能够使更多的 C/C++ 人员融入, 提供了 NDK 以便 C 类开发者开发 Android App。官方虽然提供了相应的工具, 但是在工具介绍的一开始, 就严厉地说: 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: 第一, 你有大量的 C++ 库要被复用; 第二, 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 面前都

Android 移动性能实战

没作用。为了检测 NDK 所编写的 C 代码在运行时耗费的内存，我们就必须使用一个特殊的工具，或者可以称它为库——`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。

界面展示如图 2-33 所示。

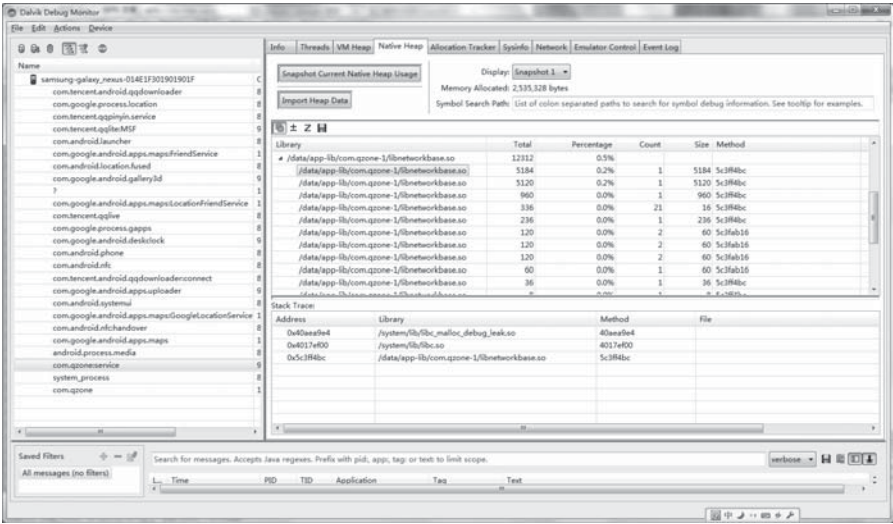


图 2-33

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` 目录中。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。

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 高负载支撑的特性，却俘获了很多开发者的心灵，Android 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过多叙述了。不过需要注意的是，在使用 NDK 的时候，一定要在 `Application.mk`

第 2 章 内存：性能优化的终结者

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。

11. APT

APT 是腾讯另外一个团队出品的一款 Android 应用测试工具，并且是开源的。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 曲线展示如图 2-34 所示。

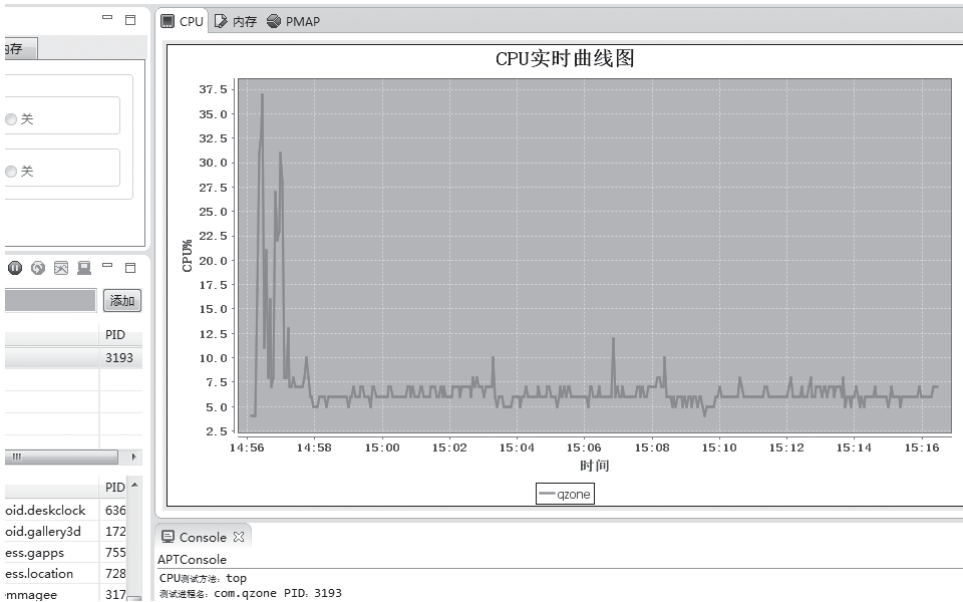


图 2-34

内存曲线展示，如图 2-35 所示。

Android 移动性能实战

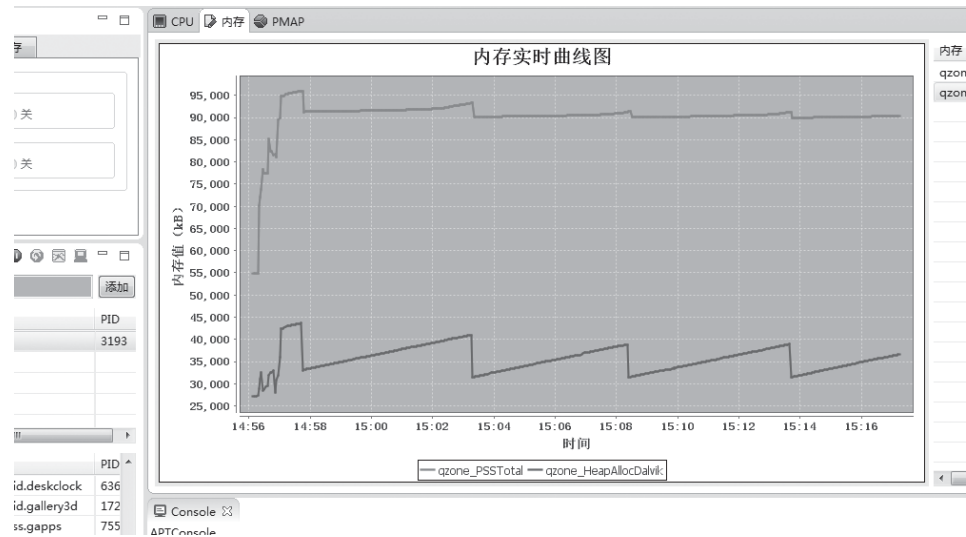


图 2-35

SMap 展示如图 2-36 所示。

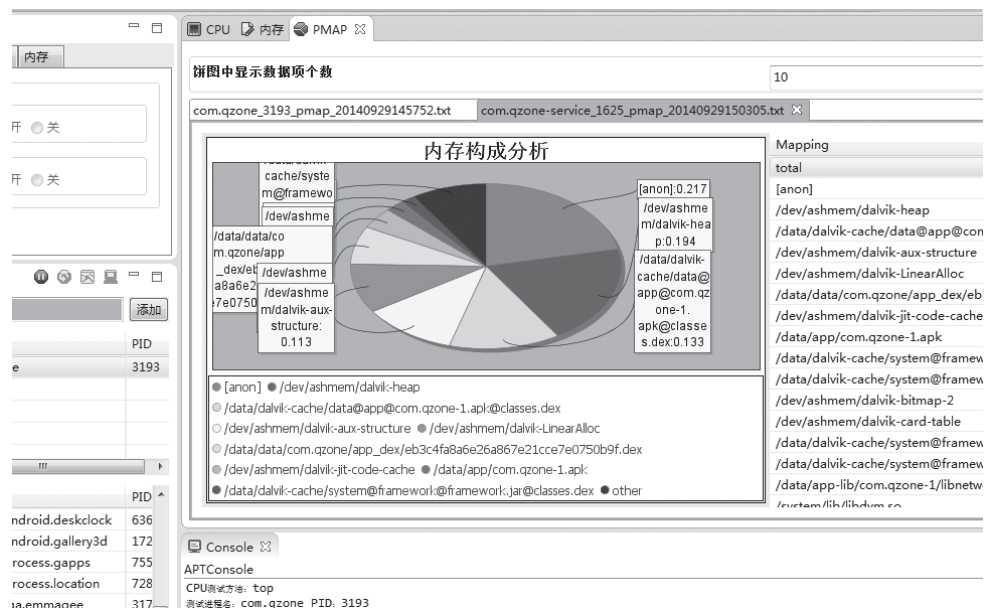


图 2-36

另外，因为 APT 实现的内存监控是调用 meminfo 分析结果来实现的，而 meminfo 每次

第 2 章 内存：性能优化的终结者

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 在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 来得直接、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，所以也不建议使用 SMap 来做 Native 内存测试。

12. GC Log

跟 GC 相关的有两个工具，一个是在 Logcat 中输出的 GC 日志，另外一个就是 Allocation Tracer。这里先介绍 Logcat 输出的日志。而日志分成 Dalvik 的 GC 日志与 ART 的 GC 日志两种。

（1）Dalvik GC 日志解析（如表 2-4 所示）

表 2-4

GC_CONCURRENT freed 2049K,	65% free 3571K/9991K,	external 4703K/5261K,	paused 2ms+2ms
GC 产生原因 回收的内存大小	currAllocated/ currFootprint	extAllocated/extLimit	暂停时间

GC 产生原因如下。

GC_EXPLICIT：通过 Runtime.gc() 与 VMRuntime.gc(), SIGUSR1 触发产生的 GC，虽不支持局部 GC, 但稍微幸运一点的是支持并发 GC。然而在列表滑动和动画播放的时候，最好还是不要出现这类日志。因为在这种高 CPU 低响应时延的场景，人工触发 GC 来消耗 CPU 的行为应该尽可能避免。

GC_FOR_[M]ALLOC：没有足够的内存空间给予即将分配的内存，这时会触发 GC。这种 GC 因为不是并发 GC, 所以对卡顿的影响更大，应该尽量避免。QQ 以内存触顶率（MaxMemoryHeap 的 80% 作为阈值）作为内存对用户影响的外网上报指标，其实归根结底就是根据 GC for Alloc 的概念提出的。

GC_FOR_CONCURRENT：当超过堆占用阈值时会自动触发。应该是最常见的 GC 了，也是最健康的 GC，因为它支持局部 GC、并发 GC。所以暂停时间也因为它是并发 GC, 所以

Android 移动性能实战

会分成 mark 和被修改对象的 remark 两部分的耗时。对于详情,读者可以再次阅读前面的“原理: GC (Garbage Collection)”一节。

GC_BEFORE_OOM: 在触发 OOM 之前触发的 GC。这种 GC 不能并发,不能局部 GC,所以耗时长,也容易卡住界面。

GC_HPROF_DUMP_HEAP: 在 dump 内存之前触发的 GC。这种 GC 不能并发,不能局部 GC,所以耗时长,也容易卡住界面。

- currAllocated/currFootprint: 就是在 App 实际使用的堆中对象的数量 / 堆大小。
- extAllocated/extLimit: 这部分主要包括系统 Bitmap、user-defined 和 view-inflated 的 Bitmap。这部分日志,在内存存在 3.1 版本之后,回归到 Java Heap 之后就没有了。
- 暂停时间: 一个耗时和两个耗时的区别在于是否是并发 GC。正如原理中所说,并发 GC 需要第二次挂起线程的机会来处理那些被标记又因并发过程中被修改的对象。

另外有一个工具 (<https://github.com/oba2cat3/logcat2memorygraph>), 可以轻易地把 Dalvik 的 GC 日志绘制成如图 2-37 所示的图表, 可谓功能强大。

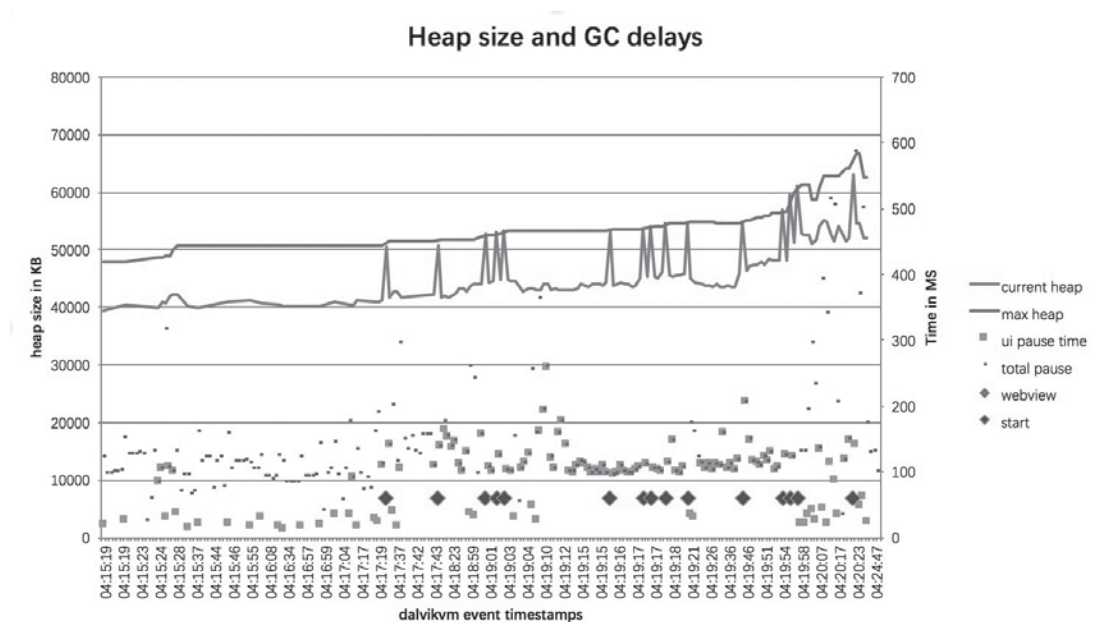


图 2-37

第 2 章 内存：性能优化的终结者

(2) ART 日志解析 (如表 2-5 所示)

ART 的日志与 Dalvik 的日志差距非常大,除了格式不同之外,打印的时机也不同,非要在慢 GC 时才打印出来。

表 2-5

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

GC 产生原因如下。

- Concurrent、Alloc、Explicit: 跟 Dalvik 的基本上一样,这里就不重复介绍了。
- NativeAlloc: Native 内存不足以分配内存时触发,跟 Alloc 类似。
- Background: 后台 GC,触发是为了给后面的内存申请预留更多空间。
- CollectorTransition、HomogeneousSpaceCompact、DisableMovingGc、HeapTrim: 产生的主要原因是 ART 的 GC 算法更加复杂。

GC 类型如下。

- Full: 简单可以理解为跟 Dalvik 的 FULL GC 差不多。
- Partial: 简单可以理解为跟 Dalvik 的局部 GC 差不多,策略是不包括 Zygote Heap。
- Sticky: 可以理解为另外一种局部中的局部 GC,选择局部的策略是上次垃圾回收后新分配的对象。

GC 方式如下。

- mark sweep: 先记录全部对象,然后从 GC ROOT 开始找出间接和直接的对象并标注。利用之前记录的全部对象和标注的对象对比,其余的对象就应该需要垃圾回收了。
- concurrent mark sweep: 使用 mark sweep 采集器的并发 GC。
- mark compact: 在标记存活对象的时候,所有的存活对象压缩到内存的一端,而另外一端可以更加高效地回收。

Android 移动性能实战

- semispace: 在做垃圾扫描的时候,把所有引用的对象从一个空间放到另外一个空间,剩余在旧的空间中的对象,只要直接 GC 整个空间就可以。

通过 GC 日志,我们可以知道 GC 的量和它对卡顿的影响,也可以初步定位一些如主动调用 GC、可分配的内存不足、过多使用 Weak Reference 等问题。但是还是不知道代码行,这时就必须使用 Allocation Tracer 或者前面的 MAT 了。

13. Allocation Tracer

在 Android Studio 里面打开 Android Monitor, 选择进程, 这里选择了比较火的名为 FaceU 的 App 来做例子。点击 Allocation Tracer, 然后经过几秒或者同步观察 GC 日志发现问题的时候, 再次点击 Allocation Tracer 暂停录制, 如图 2-38 和图 2-39 所示。



图 2-38



图 2-39

这时候, 观察 Android Studio 的代码区, 会出现录制结果。一般来说按照 Size 排序, 然后从最大的一层层展开, 如图 2-40 和图 2-41 所示。

第 2 章 内存：性能优化的终结者

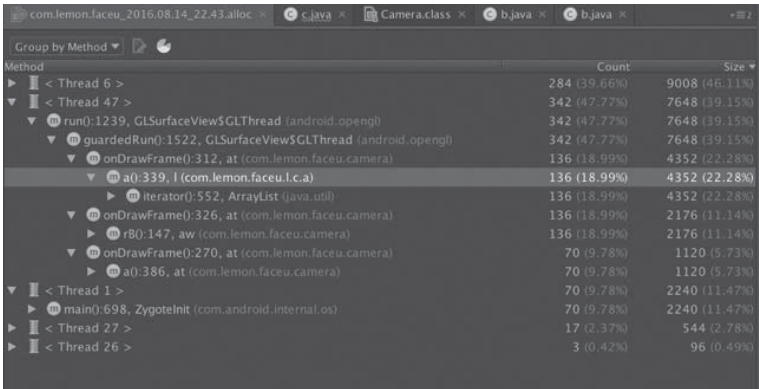


图 2-40

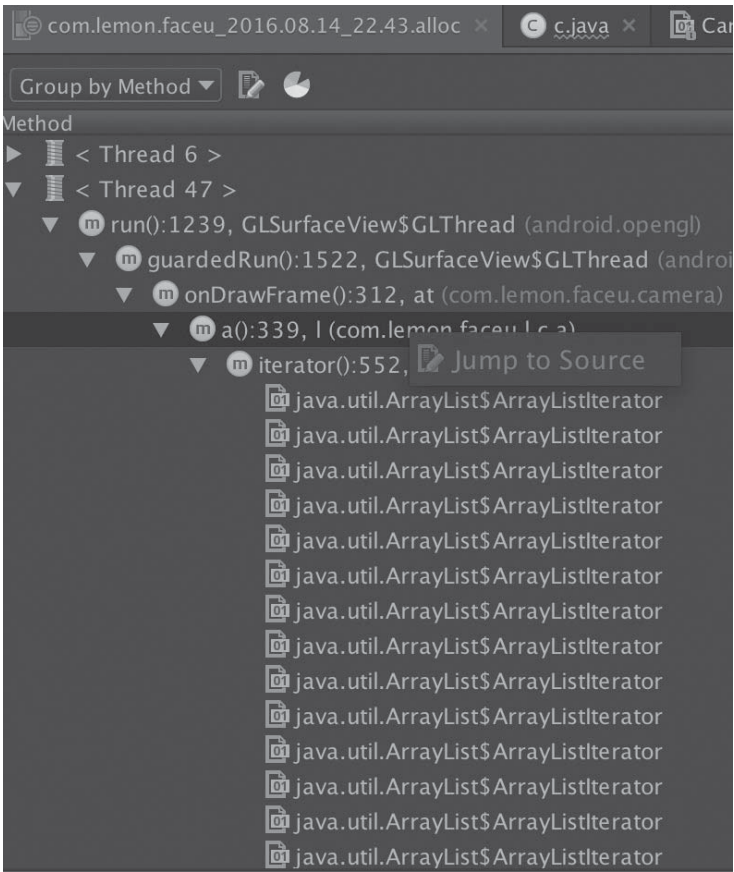


图 2-41

Android 移动性能实战

然后通过 dump to source 跳转到对应的代码行,结合申请了大量的 ArrayListIterator 结果,就可以思考一下有没有优化的方法,例如用 ArrayList.toArray 来优化。

14. 自带防泄漏功能的线程池组件

开发人员做子线程操作的时候,喜欢用匿名内部类 Runnable 来操作,敲起代码来简单、快捷、方便。然而,如果某个 Activity 放在线程池里的任务不能及时执行完毕,在 Activity 销毁时很容易导致内存泄漏。

原因为何?下面来看一段很简单的代码,如图 2-42 所示。

```
public class Outer {
    public Runnable getRunnable() {
        return new Runnable() {
            public void run() {
                System.out.println("hello");
            }
        };
    }
}
```

图 2-42

用 javac 编译之后,结果如图 2-43 所示。

```
Compiled from "Outer.java"
class Outer$1 extends java.lang.Object implements java.lang.Runnable{
    final Outer this$0;

    Outer$1(Outer);
    Code:
        0:   aload_0
        1:   aload_1
        2:   putfield      #1; //Field this$0:LOuter;
        5:   aload_0
        6:   invokespecial #2; //Method java/lang/Object."<init>":()V
        9:   return

    public void run();
    Code:
        0:   getstatic     #3; //Field java/lang/System.out:Ljava/io/PrintStream;
        3:   ldc          #4; //String hello
        5:   invokevirtual #5; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
        8:   return
}
```

图 2-43

第 2 章 内存：性能优化的终结者

可见这个匿名内部 Runnable 类持有一个指向 Outer 类的引用，这样一来如果某 Activity 里面的 Runnable 不能及时执行，就会使它外围的 Activity 无法释放，产生内存泄漏。那么，我们要想自动避免这个问题，有没有办法呢？从上面分析可见，只要在 Activity 退出时没有这个引用就可以，那我们就通过反射，在 Runnable 入线程池前先干掉它，如图 2-44 所示。

```
Field f = job.getClass().getDeclaredField("this$0");
f.setAccessible(true);
f.set(job, null);
```

图 2-44

这个任务是我们的 Runnable 对象，而“this\$0”就是上面所述指向外部类的引用了。当然，等到执行它的时候，没了这个引用可能会出问题。因此干掉它之后得“留个全尸”，找个 WeakReference“墓地”放起来，要执行了先 get 一下，如果是 null 说明 Activity 已经回收，任务就放弃执行。

15. Chrome Devtool

什么表现可能是内存泄漏问题引起的？
先看一个例子：

《疯狂打怪兽》	进入游戏黑屏	vivo y11 华为 t8954 华为 c8813dq
《疯狂打怪兽》	进入游戏 ANR	索尼爱立信 lt18i

“玩吧”这个功能的卡帕莱自动化测试结果显示，《疯狂打怪兽》这个游戏在某些机型上有黑屏、ANR。我们用已有的测试机无法重现 Bug，不过发现游戏进行一段时间后确实有一些卡顿。用 Chrome Devtool Timeline 工具检查了一下，游戏过程中内存一直在涨，进一步抓内存快照分析发现有泄漏，如图 2-45 所示。

Android 移动性能实战

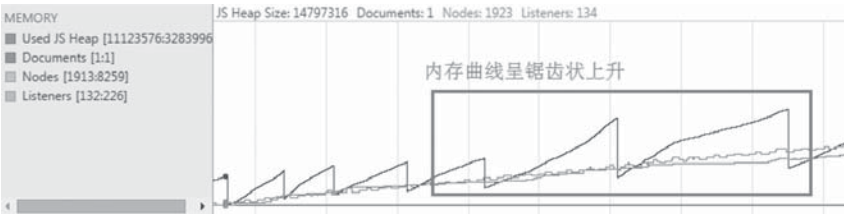


图 2-45

跟 Java 层内存泄漏类似，页面出现黑屏、ANR、卡顿，这些都很有可能是内存问题。

启动远程调试

首先，我们要有一个工具，可以实时抓到手机上的内存消耗。Java 层使用的是 ADT 工具，但是对于 HTML5 页面而言，抓取 JavaScript 的内存需要用到 Chrome Devtools 来进行远程调试。方式有两种，第一种可以先把 URL 抓取出来放到 Chrome 里访问，第二种用 Android H5 远程调试。下面具体介绍稍微复杂一点的 Android H5 远程调试，如表 2-6 所示。

表 2-6

场 景	是否支持调试
纯 H5（手机浏览器里打开）	Chrome 支持远程调试
默认 Hybrid H5 （App 内嵌 H5 页）	系统自带的浏览器内核，Android 4.4 以前是 WebKit 内核，不支持；Android 4.4 系统开始为 Chromium 内核，支持远程调试
TBS hybrid H5 （QQ 空间、手机 QQ、微信内嵌 H5）	使用 QQ 浏览器提供的 TBS 内核，支持远程调试

不同的场景需要不同的启用远程调试的方法

前面提到，Chrome 和 TBS 都支持远程调试，接下来介绍三种场景下的调试方法。

（1）纯 H5

这个最方便，适用于 Android 4.0+ 系统。下面具体说明需要准备什么。

- 一台 PC（安装最新版 Chrome）
- 一根 USB 线
- 一部手机（安装手机 Chrome）

第 1 步：手机安装 Chrome，打开 USB 调试模式，通过 USB 连上电脑，在 Chrome 里

第 2 章 内存：性能优化的终结者

打开一个页面，比如百度页面。然后在 PC Chrome 地址栏里访问 `Chrome://inspect`，如图 2-46 展示了调试目标设备和页面。



图 2-46

第 2 步：点击调试页面下的 `inspect`，弹出开发者工具界面，如图 2-47 所示，通过最上方的 url 可以确认调试目标。

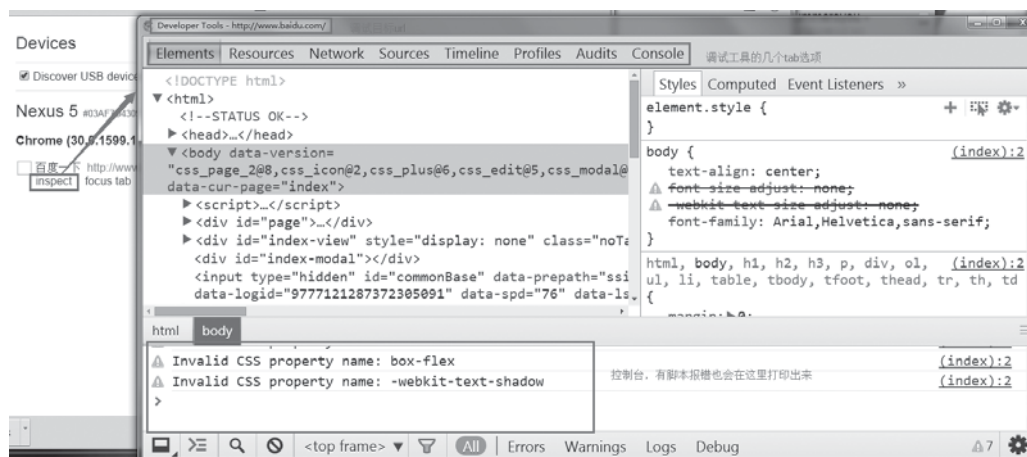


图 2-47

Android 移动性能实战

(2) 默认 hybrid H5 调试

Android 4.4 及以上系统的系统原生浏览器就是 Chrome 浏览器，可以使用 Chrome Devtool 远程调试 WebView，前提是需要 App 的代码里把调试开关打开，如图 2-48 所示。

Debugging WebViews

On Android 4.4 (KitKat) or later, you can use DevTools to debug WebView content in native Android applications.

Configure WebViews for debugging

WebView debugging must be enabled from within your application. To enable WebView debugging, call the static method **setWebContentsDebuggingEnabled** on the WebView class.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {  
    WebView.setWebContentsDebuggingEnabled(true);  
}
```

This setting applies to all of the application's WebViews.

Tip: WebView debugging is **not** affected by the state of the `debuggable` flag in the application's manifest. If you want to enable WebView debugging only when `debuggable` is `true`, test the flag at runtime.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {  
    if (0 != (getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE))  
    { WebView.setWebContentsDebuggingEnabled(true); }
```

图 2-48

第 2 章 内存：性能优化的终结者

打开后调试方法跟纯 H5 页面调试方法一致。直接在 App 里打开 H5 页面，再到 PC Chrome 的 inspector 页面，即可看到调试目标页面。

（3）TBS hybrid H5 调试

（更详尽的内容可见：<http://x5.tencent.com/tbs/document/debug-detail-wifi.html>）

适用于 Android 4.0+ 系统，手机 QQ 空间 / 手机 QQ / 微信都默认安装使用了 TBS 内核，支持远程调试。操作步骤如下。

第 1 步：打开 <http://debugx5.qq.com>，进入 X5 调试页面，点击“信息”选项卡，下拉找到“是否打开 TBS 内核 inspector 调试功能”复选框，勾选该复选框之后也会弹出信息框提示设置成功，重启后再进入 WebView 时会打开调试开关，如图 2-49 所示。



图 2-49

第 2 步：Aandroid 手机通过 USB 连接电脑，PC 打开 Chrome 浏览器，在地址栏中输入 `chrome://inspect/#devices`。

第 3 步：手机上打开你想要调试的 H5 页面，就能看到调试对象，如图 2-50 所示，QQ 空间、手机 QQ、微信都可以，点击 inspect 即可打开调试页面。

备注：如果点击 inspect 后新打开的页面为白屏，确认一下是不是 Google 被墙了，是否需要 VPN；如果首次安装后立马打开 H5 页面，可能还没有使用 TBS 内核，等 1 分钟后

Android 移动性能实战

再杀进程重启即可，无须 ROOT 手机，无须额外安装任务工具。

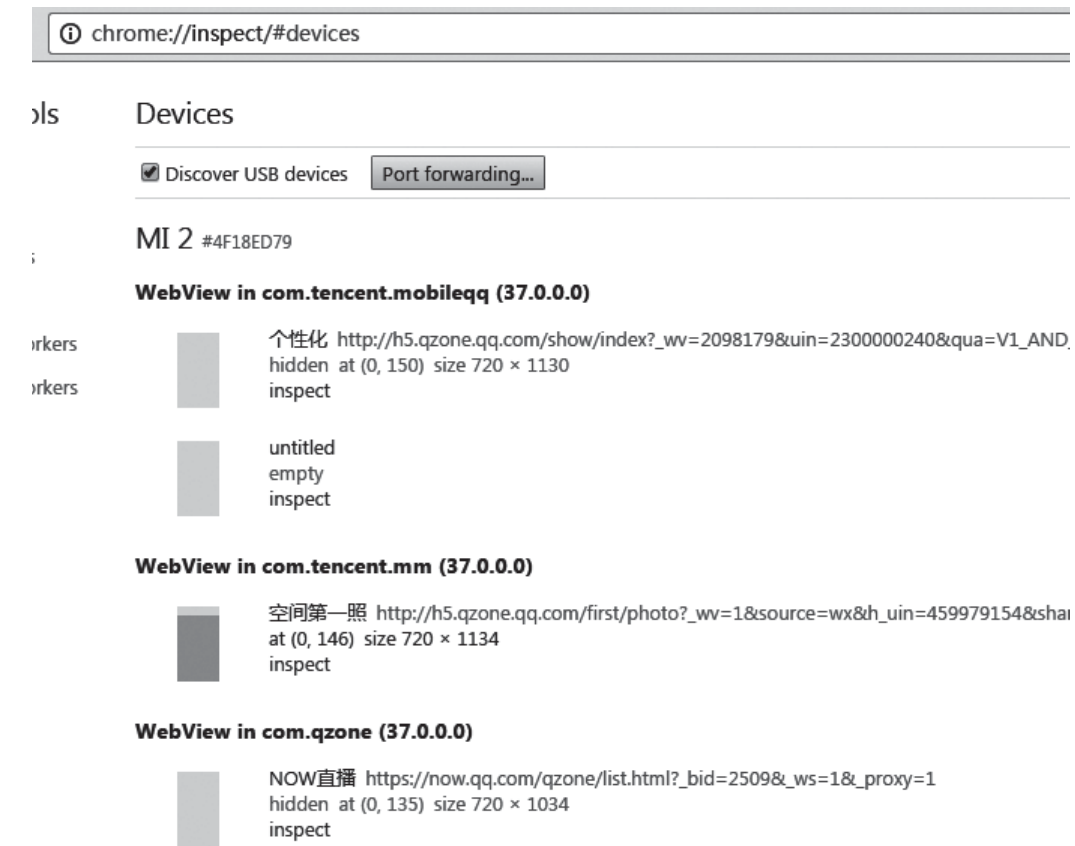


图 2-50

分析内存泄漏

掌握远程调试方法后，就可以利用 Devtool 来做内存测试了。H5 内存测试方法可以总结为以下几个步骤。

第 1 步：观察内存曲线。拿到一个新需求以后，开启 Timeline 工具的 Memory Record，重复功能操作的同时记录下内存曲线，观察内存是否一直增长。

第 2 步：记录场景并抓取内存快照。对于某些引起内存增长的操作，比如重复切换 tab、进二级页面再退回，把它们记录下来，使用类似于两遍三 dump 的方法，在 Profile 工具里抓取内存快照 Take Heap Snapshot。

第 2 章 内存：性能优化的终结者


第 3 步：快照分析。通过快照的 Summery、Caomparison 等视图，找出染色的对象和不合理的新增对象；分析对象的引用关系，找出是被 DOM 树中的节点 hold 住了，或是被 JS 代码里的某个对象 hold 住了。

第 4 步：确认并提单。跟开发人员确认，比如新增的对象是否合理，是不是缓存需要的；确认后提 Bug 单跟进问题，并附上截图和快照信息。

第 5 步：回归验证。修复 Bug 单后，重新执行第 2 和第 3 步，查看泄漏的地方是否修复。

详细介绍

（1）使用 Timeline 直观地查看内存曲线

在 Timeline 标签下，勾选 Capture memory，工具栏中左边有个小圆点按钮，开始是灰色的，单击后变成红色并开始记录数据，然后再单击旁边第 4 个  按钮，执行 GC 操作。在相应的 H5 页面进行一些操作，可以看到内存曲线相应地产生了变化，操作结束后再执行一次 GC，然后单击小圆点按钮停止记录。

如图 2-51 所示，记录的是游戏通关过程中内存的变化，中间有一些锯齿形的波形，是因为有内存被 GC 掉。GC 后内存没有降回游戏开始的水平，而是一直在增长。

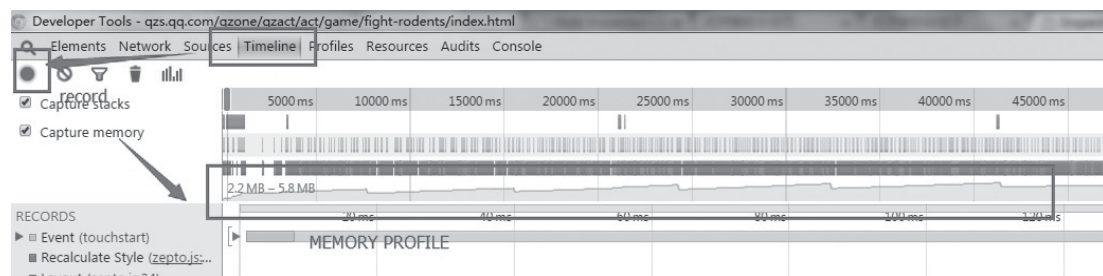


图 2-51

（2）抓取内存快照

这一步用到 Profile 里的“take haep snapshot”，分析某些可能出现泄漏的场景，在操作前后分别抓取内存，通过比较内存的变化，来定位哪些新增的 object、函数或是 DOM 节点是不合理的。

抓取内存快照的方法跟终端的两遍三 dump 类似，介绍如下。

- ①操作前在页面 A 先抓一个快照 1。
- ②操作后在页面 B 抓一个快照 2。

Android 移动性能实战

③返回页面 A 再抓一个快照 3。

④重复②和③（可以重复多次，如果有泄漏会更明显），再抓一个快照 4。

⑤通过对比快照 1 和快照 2，可看到页面 B 新增了些对象，新增的对象是否合理；对比快照 1 和快照 3 可看到，退出页面 B 后还有哪些页面 B 的对象在内存中没有被回收，是否为需要缓存的对象；对比快照 1 和快照 3、快照 3 和快照 4，如果 3 和 4 对比结果与 1 和 3 对比结果一样，有很多新增对象，那么很有可能是泄漏。

上面的例子，通过 Timeline 曲线，发现每次新开始一轮游戏，内存都会有增长。初步推测可能是开始新游戏的时候，上一轮的一些资源没有释放。

使用 Profile 工具，可以抓取内存快照，类似于 ADT 工具的 dump。每次抓取时，工具都会自动 GC，直接单击按钮即可，如图 2-52 所示。

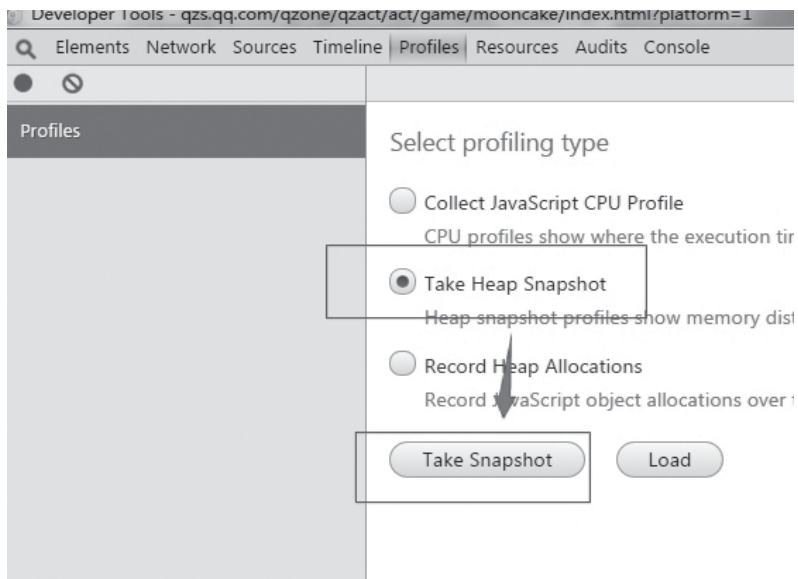


图 2-52

比如游戏中，要分析是否有泄漏，需要采用操作步骤①、步骤③、步骤④。

- 在游戏首页，抓取 heap-mainpage。
- 玩一轮游戏后，抓取 heap-round1。
- 玩两轮游戏后，抓取 heap-round2。

对比这几个文件的大小，玩一轮游戏后，内存从 2.0MB 涨到了 3.3MB，是因为加载了

第 2 章 内存：性能优化的终结者

游戏过程中需要的一些页面元素，而两轮游戏后内存涨到了 8.5MB，这明显不合理，如图 2-53 所示。

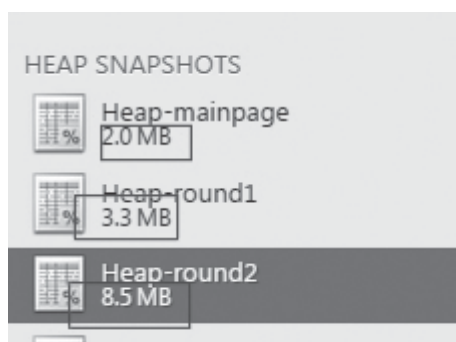


图 2-53

通过内存快照的对比定位具体问题

如图 2-54 所示，选择某个内存快照，右侧会展示内存里具体都有哪些对象及对象的大小。工具栏的下面有一个下拉选项，可以选择视图：Summary、Comparison、Containment、Statistics 四类。分析内存问题最常用到的两个视图是 Summary 和 Comparison。

选择快照 heap-round2，选择 Comparison 视图，在旁边的下拉选项里选取要对比的另一个内存快照 heap-round1，即可查看两份内存快照的详细对比，包括新增了、回收了哪些对象、节点等。类似于分析 App 泄漏的 Finder compare 操作。箭头指向的地方可以看到具体是哪里用到了，注意有一行黄色背景。一般黄色背景和红色背景都需要特别注意，这些都是工具帮你分析出来的疑似内存泄漏的点。红色背景代表这个对象被某个标了黄色背景的对象引用了，而黄色背景代表本对象被 JS 代码直接引用了。

例子中的问题是 DOM 树里的 iframe 泄漏了。

Android 移动性能实战

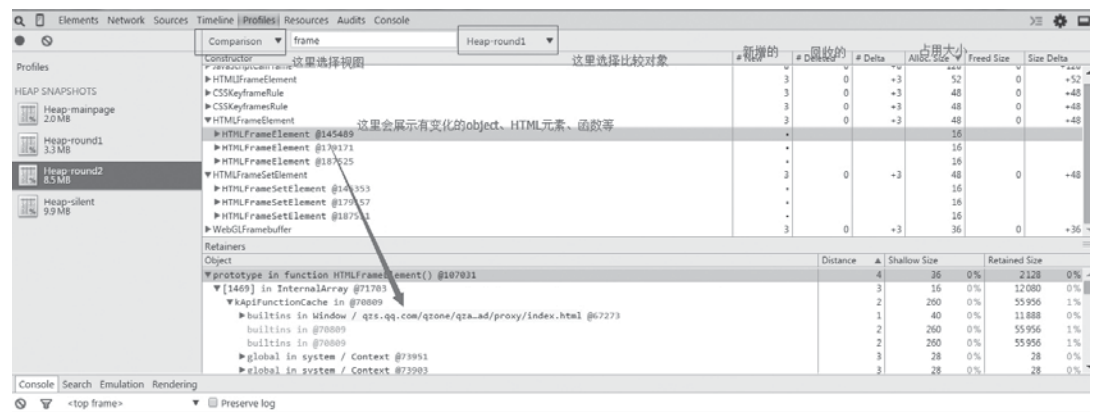
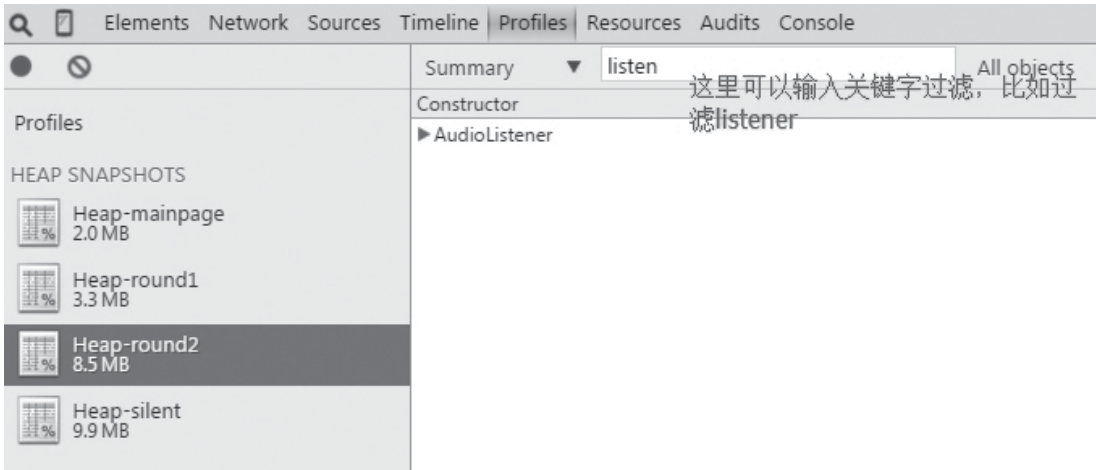


图 2-54

除了这些有颜色标记的部分,其他的对象变化需要依靠测试人员对业务的了解来判断,哪些新增的是不合理的。

Summary 视图另有一个好处。在视图下拉框旁边有个 class filter, 可以根据关键字来过滤对象。选择 Summary 视图, 再输入 detached 关键字, 可以看到有一项 Detached DOM tree, 指的就是 DOM 树中不再使用却无法回收的 DOM 节点。展开可看到具体的节点列表。选中某个节点, 在控制台里输入 “\$0”, 可以看到节点的内容。从 Detached 的第一个开始看, 因为其他节点很有可能是第一个节点的子节点。从 Shallow size 比 Retained size 小很多也可以看出, 第一个节点不是叶子节点, 其还包含了其他子节点, 如图 2-55 和图 2-56 所示。



第 2 章 内存：性能优化的终结者

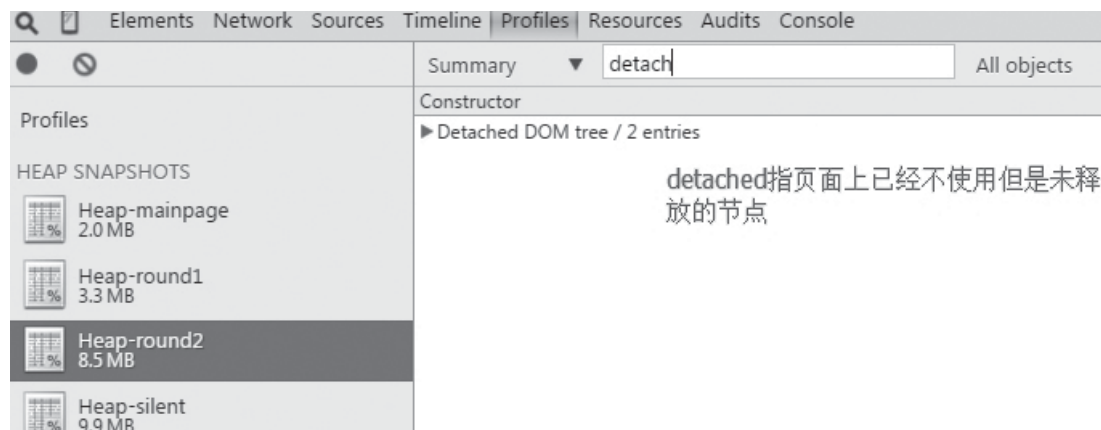


图 2-55

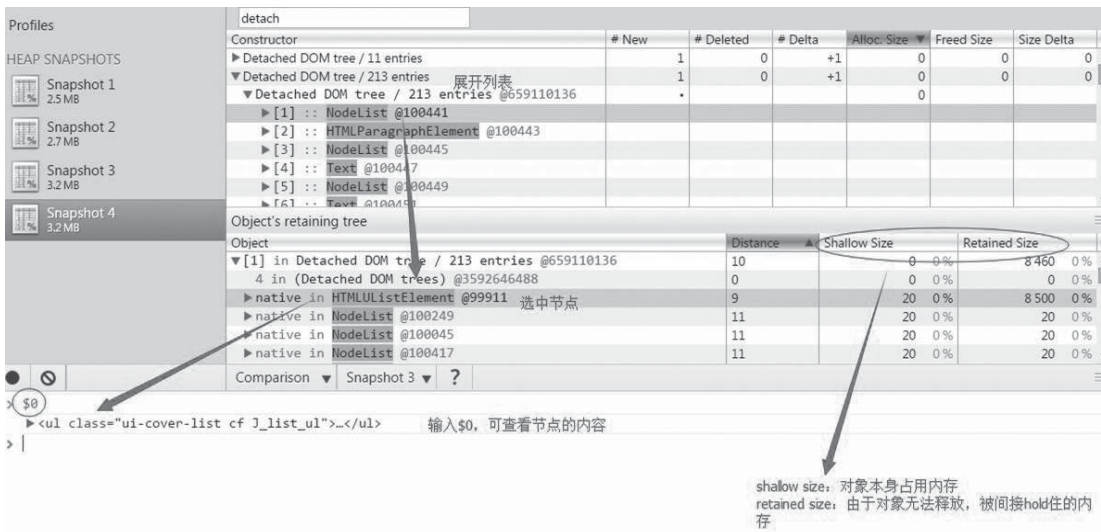


图 2-56

找到一些疑似泄漏的点，最好先和开发人员确认一下，达成共识后就可以提单，注意要附上 heapsnapshot 文件，选择对应的文件可以保存，如图 2-57 所示。

Android 移动性能实战

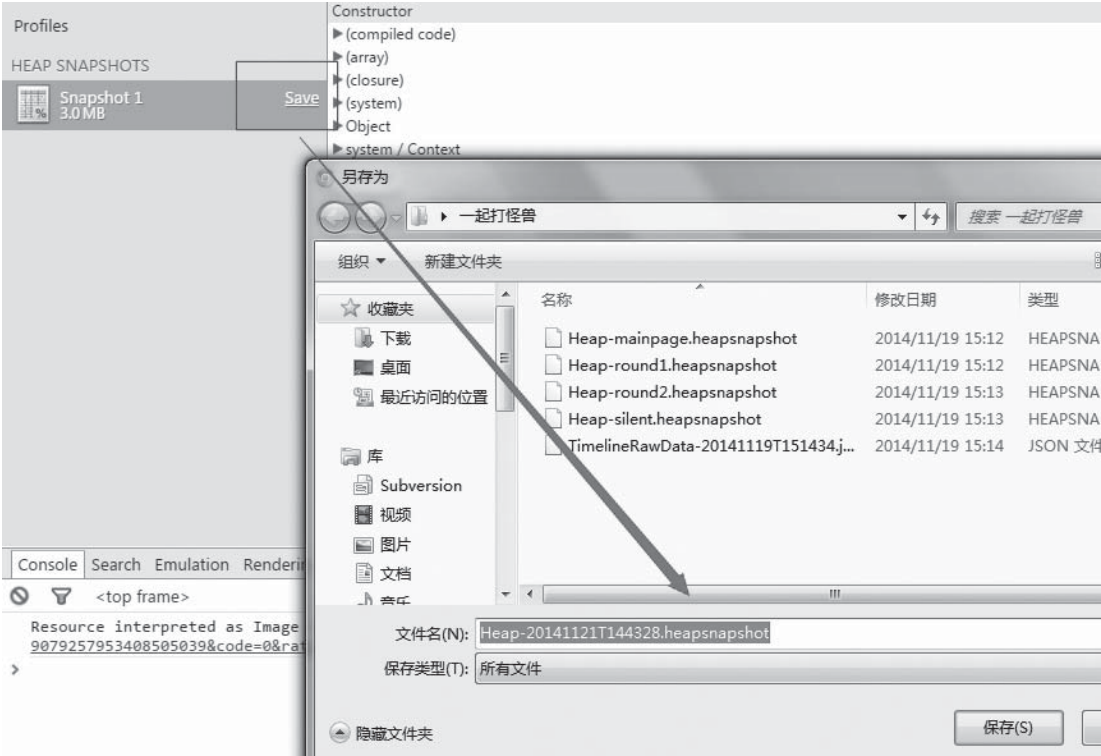


图 2-57

最后总结一下，一般来说，容易产生问题的有以下几种，需要特别留意。

- closure 闭包函数（如图 2-58 所示）。
- 事件监听。
- 变量作用域使用不当，全局变量的引用导致无法释放。
- DOM 节点的泄漏。

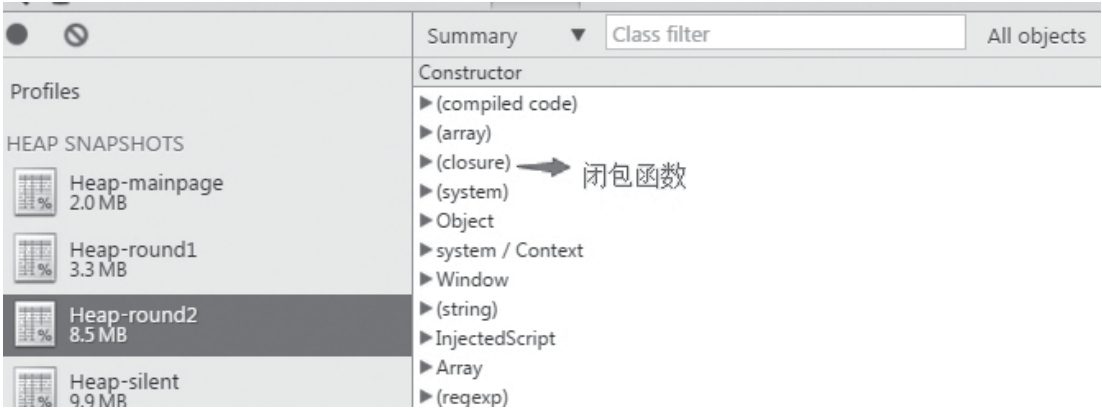


图 2-58

读者若想学习 Chrome 开发者工具使用方法，可以查看《Chrome 开发者工具中文手册》（<https://github.com/CN-Chrome-DevTools/CN-Chrome-DevTools>）。

2.3 案例 A：内类是有危险的编码方式

问题类型：Activity 泄漏

GC ROOT：静态变量

引用方式：this\$0 间接引用

解决策略：解除引用关系

内类是一种 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，如图 2-59 所示。但每个好处的背后都一定潜伏着一个麻烦。