HZ BOOKS
华章科技

本书由业界多位移动团队技术负责人联袂推荐，为打造高质量App提供了有价值的实践指导。
书中总结了80多个Crash的分析与处理，是迄今为止最完整的Android异常分析资料。
剖析了国内上百款知名App的前沿技术实现，是最权威的竞品技术分析白皮书。

移动开发

包建强◎著

App研发录

架构设计、Crash分析和竞品技术分析

机械工业出版社
China Machine Press

移动开发

App 研发录

架构设计、Crash 分析和竞品技术分析

包建强　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

App 研发录：架构设计、Crash 分析和竞品技术分析 / 包建强著 . —北京：机械工业出版社，2015.9
（移动开发）

ISBN 978-7-111-51638-5

I. A…　II. 包…　III. 移动电话机 – 应用程序 – 程序设计　IV. TN929.53

中国版本图书馆 CIP 数据核字（2015）第 228298 号

本书是作者多年 App 开发的经验总结，重点介绍 Android 应用开发中常见的实用技巧和疑难问题解决方法，为打造高质量 App 提供了有价值的实践指导，可帮助读者迅速提升应用开发能力和解决疑难问题的能力。本书涉及的主题有：Android 项目的重构、网络底层框架设计、经典场景设计、命名规范和编程规范、Crash 的捕获与分析、持续集成、代码混淆、App 竞品技术分析、移动项目管理和团队建设等。本书内容丰富，文风幽默，不仅给出疑难问题的解决方案，而且结合示例代码深入剖析这些问题的实质和编程技巧，旨在帮助移动开发人员和管理人员提高编程效率，改进代码质量，打造高质量的 App。

App 研发录
架构设计、Crash 分析和竞品技术分析

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：吴　怡　　责任校对：殷　虹
印　　刷：　　版　　次：2015 年 10 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：20.5
书　　号：ISBN 978-7-111-51638-5　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

Preface 序 一

互联网时代什么人是核心驱动力

在我刚刚开始宣布要做奇酷手机的时候，我曾经发布公开信说我需要四类动物：程序猿、攻城狮、产品狗、设计猫。程序员被排在了第一位，而从我的个人经历来说，与程序员有着密切的关系：大学研究生时的程序员，上班时的工程师，创业后的产品经理，最近几年一直在学习和琢磨设计。

这本书的作者建强也是其中一种人，一种喜欢钻研技术的程序员。我曾经和《奇点临近》作者雷·库兹韦尔交流的时候提到，也许上帝就是一名程序员，因为程序员正在通过给基因重新编程的方式来解决人类很多疾病之类的问题。

当然，实现给基因编程解决人类疾病问题的过程是漫长的，但“程序员”的作用是重大的。而在互联网的世界里，程序员的重要性更明显。一个好的程序员能力固然重要，精神世界的升华也不能缺少，写书就是一种精神世界的升华，能说服自己，也能帮助和提高更多人。

互联网时代离不开各种移动 App，本书提到很多时下移动互联网很前沿的技术，像竞品技术分析部分就提到 ABTest、WaxPatch 等。而且据说，为了写这本书，作者分析了市场上有名的上百款 App，能够费这么多心血去研究技术实现的人，在我看来至少是一个充满好奇心的人。正是这种拥有好奇心并执着探索的人，推动了近百年来的科学发展。

移动互联网的世界更是如此，从手机产生至今，短短二三十年的时间，就已经发生了翻天覆地的变化。今天的手机已经快成为人类的器官了，未来手机是什么样子很难说，但对手机应用的要求越来越高。虽然 iOS 和安卓平台上开发 App 会有所不同，但用户在各方面体验的要求是一致的。所以在我做手机的过程中，一直要求自己要充满好奇心。

移动 App 是一个充满了未知和探索的领域，这也正是它的魅力所在，所以越来越多渴望探索的人加入到移动互联网的创业大潮中来。事实上，这些移动 App 正在改变着我们的生活，

从订餐、打车到游戏娱乐都被各种 App 所改变。

但 App 相关的技术发展、更新非常迅速，所以作为技术人员要保持对技术的敏锐嗅觉，永远抱着谦卑的心态去学习先进的技术和理念，才能时刻占据着主动。当我们认为自己对这个世界已经相当重要的时候，其实这个世界才刚刚准备原谅我们的幼稚。

互联网发展到今天，程序员功不可没。也许程序员真的就是上帝，但他们在创造出一个个绚丽多彩的世界之前，注定要沉浸在枯燥的代码之中。我相信，每个程序都有自己的一个小小世界，在程序世界里，一切都按照他们的设计规则运行。那么你说，这和上帝创造世界有什么不同？互联网的世界里谁才是核心驱动力？

当然，我也希望这本书能培养出更多的 App 领域高级人才，来共同繁荣移动互联网的世界。

周鸿祎

奇虎 360 董事长

Preface 序　二

十年写一本书

1998 年，Peter Norvig 曾经写过一篇很有名的文章，题为“ Teach Yourself Programming in Ten Years”（十年学会编程）。这文章怎么个有名法呢，自发表以来它的访问次数逐年增加，到 2012 年总数已经接近 300 万次。

文章的意思其实很简单，编程与下棋、作曲、绘画等专业技能一样，不花上十年以上有素的训练（deliberative practice）、忘我的（fearless）投入，是很难真正精通的。Malcolm Gladwell 后来的畅销书《异类》用 1 万小时这个概念总结了类似的观点。

那么，写书呢？

说到写书，我的另一位朋友在微博上说过一段话，让我一直耿耿于怀：“有的时候被鼓励、怂恿写书，就算书能卖 5000 册，40 块一本，8% 的版税，收益是 16000RMB，这还没缴税。我还不如跟读者化缘，把内容均匀地贡献给读者，一个礼拜就能募集到这个数额。为什么要去写书？浪费纸张，污染环境。为了名气？为了评职称？”这位朋友是 2001 年我出版的国内第一本 Python 书的译者，当时这本封面上是一只老鼠的书只有 400 页，现在英文原版最新版已经 1600 页了——时间真是最强大的重构工具。其实他的话说得挺实在的。这年头写专业图书，经济上直接的回报，的确很低。

可要是真的没人写书了，这个世界会好吗？

我曾经在出版社工作十几年，经手的书数以千计，有的书问世之初就门可罗雀，有的书一时洛阳纸贵但终归沉寂，只有少数一些，能够多年不断重印，一版再版。这后一种，往往是真正的好书，是某个领域知识系统整理的精华，其作用不可替代，Google 索引的成千上万的网页也不能——聚沙其实是不能成塔的。Google 图书计划的受挫，其实是人类文明发展的重大延迟，对此我深以为憾。

书（我说的是科技专业书）可以粗略分为两种，一种是入门教程性质的，一种是经验之谈

或者感悟心得。无论哪一种，都需要作者多年教学或者实践的积淀。很难想象，没有这种积淀，能够很好地引导读者入门，或者教授其他人需要花费十年才能掌握的专业技能。

所以，好书不易得。它不仅来之不易（有能力写的人本来就少，这些人还不一定有动力写），而且还经常面临烂书太多，可能劣币驱逐良币的厄运。最后的结果是，很多领域都缺乏真正的好书，导致整个圈子的水平偏低。因为，书这种成体系的东西，往往是最有效的交流与传承手段，是互联网（微博、微信、博客、视频等等）上碎片化的信息不能取代的。

几天前，我的 Gmail 邮箱里收到一封邮件：

> “还记得 2008 年我就想写一本书，但是感觉技术能力不够，就只好去翻译了一本。一晃 2015 年了，积累了 11 年的技术功底，写起书来游刃有余。这本书我整整写了 1 年，其中第 6 章和第 9 章是自认为写得最好的章节。请刘江老师为我这本书写一篇序言，介绍一下无线 App 的技术前瞻和趋势，以及看过样章后的一些感想吧。谢谢。”
>
> 包建强

包建强？我记得的。一个长得挺帅的大男孩儿，2004 年复旦大学毕业，2008 年被评为微软的 MVP。在技术上有追求，而且热心。多年前在博客园非常活跃，张罗着要将里面的精华文章结集出版成书。很爱翻译，曾找我推荐过国外的博客网站，想要把一个同学 WPF/Silverlight 系列文章翻译成英文。2009 年我在图灵出版了他翻译的 .NET IL 汇编语言方面的书，到现在也是这一主题唯一的一本。好多年没联系，没想到他已经从 .NET 各种技术（WPF、Silverlight、CLR 等），转向移动客户端开发了。更让人动容的是，他一直不忘初心，用了十年，终于完成了自己的书。

那么，这是一本什么样的书呢？

我将书的几个样章转给身边从事 Android 开发的一位美团同事，他看了以后有点小激动，给了这样的评价：

> “拿到这本书的目录和样章时，感到非常惊喜，因为内容全是一线工程师正在使用或者学习的一些热门技术和大家的关注点。比如网络请求的处理、用户登录的缓存信息、图片缓存、流量优化、本地网页处理、异常捕抓和分析、打包等这些平时使用最多的技术。
>
> 我本人从事 Android 开发两年，特别想找一本能提高技术、经验之谈的书，可惜很难找到。本书不光站在技术的层面上去谈论 Android，还通过市场上比较火的一些 App 和当今 Android 在国内发展的方向等各种角度，来分析怎么样去做好一个 App。
>
> 我个人感觉这本书不仅能让你从技术上有收获而且在其他层面上让你对 Android 有更深层次的了解。我已经迫不及待这本书能够尽快上市，一睹为快了。”

的确，目前国内外市面上数百种 Android 开发类图书，基本上可以分为两类：

- 一类是从系统内核和源代码入手，作者往往是 Linux 系统背景，从事底层系统定制等方面工作。书的内容重在分析 Android 各个模块的运行机制，虽然深入理解系统肯定对应用开发者有好处，但很多时候并不是那么实用。
- 一类是标准教程，作者往往是培训机构的老师，或者不那么资深但善于总结的年轻工程师，基本内容是 Android 官方文档的变形，围绕 API 的用法就事论事地讲开去。虽然其中比较好的，写法、教学思路和例子上也各有千秋，但你看完以后真上战场，就会发现远远不够。

本书与这两类书都完全不同，纯从实战出发，在官方文档之上，阐述实际开发中应该掌握的那些来之不易的经验，其中多是过来人踩过坑、吃过亏，才能总结出来的东西。不少章节类似于 Effective 系列名著的风格，有很高的价值。

书最后的部分讨论了团队和项目管理，既有比较宏观的建议，比如流程、趋势，更多的还是实用性非常强的经验，比如百宝箱、必备文档，等等。很多章节，不限于 Android，对其他平台的移动开发者也有很大的借鉴意义。

看得出来，这里很多内容都是包建强自己平时不断记录、积累的成果，其中少量在他的博客上能看到雏形。如果说多年前，包建强在组织和翻译图书时还有些青涩的话，本书中所显现出来的，则完全是一派大将风度，用他自己的话来说，“游刃有余”了。

我曾经不止一次和潜在的作者说过：“不写一本书，人生不完整。”我说这话是认真的。人生百年，如果最后没有什么可以总结、留之后人的东西，那可不是什么值得夸耀的事情。

而说到总结，互联网各种碎片化的媒体形式当然有各种方便，但到头来逃脱不了烟花易逝的命运（想想网上有多少好的文字，链接早已失效，现在只能到 archive.org 上寻找，甚至那里也不见踪影）。还是书这种物理形式最坚实，最像那么回事儿，也最是个东西。去国家图书馆看过宋版书的人肯定会有体会。

当然，真正能立住的，是那些真正的好书，那些花费十年写出来的东西。希望有更多的同学像包建强这样，十年写一本书。希望有更多好书不断涌现出来。

我更希望，包建强不止步于书的出版，而是能将书的内容互联网化，让读者和同行也加入进来，不断生长、丰富，不断改版重印，变成一种活的东西。写一本好书，不应该限于十年。

刘江

美团技术学院院长

CSDN 和《程序员》杂志前总编

序　三 Preface

这是一本很有特点的书，没有系统的知识介绍，也没有对细分领域钻牛角尖般的头头是道。第一次看完老包的样章时我很惊讶，他不仅一个人完成了全书内容的撰写，而且其中大部分章节都非常接地气并具有时代性。

当前移动开发技术处在一个野蛮增长的时代，在移动开发从业人员逐年递增的情况下，很多公司的移动开发团队都有几十人甚至上百人。当 App 越做越大，承载了越来越多的功能时，不断地累加代码也造成了很多问题。在解决这些问题的同时，很多人从单纯的业务开发转向深入研究技术细节，沉淀了很多经验，并诞生了不少有意思的开源项目。

在 2013 年我首次遇到 Android 65536 方法数限制的时候，网络上唯一能查询到的资料就是 Facebook 上的一篇博客，其中简单介绍了博主遇到的问题及解决的大致方法。当时在没有任何参考资料的情况下只能自己开发解决方案，并且由于需要分拆 dex 引入了不少其他的问题。今天看到本书中总结的这些经验和问题，发现本书能够给我很好的启示，原本那些踩过的坑和交过的学费其实都是可以避免的。虽然书中介绍每个问题时篇幅看上去并不大，但是提炼得很精简，如果你对其中的某段不是很理解，很可能它正是在你真正遇到问题时会联想到的内容和恰到好处的解决方案。

本书第 6 章常见的异常分析，就是完全基于实践积累完成的。就阅读这章本身来说，可能学到的知识点非常分散，但是包含了很多不为人知的冷门或者非常细节的知识。如果你对其有深刻的共鸣，多数都是因为自己曾有过被坑的经历。在我自己的异常分析过程中，会遇到一些非常难理解的异常，俗称“妖怪问题”。这类异常的表象很难和原因联系到一起，光读取栈信息不足以理解异常的机理，这时候就需要有更完善的异常收集系统，能够把应用的当前状态进行回溯，这对分析问题是很有帮助的。

本书第 9 章我认为是最接地气也是最有特色的章节，从分析国内热门的 App 开始，帮助读者了解最前沿的大公司的移动开发的技术方向。有很多技术点是小的 App 开发团队并不会

花精力关注的，比如资源文件如何组织，如何应对线上故障等，但是如果在应用规模急剧增长后再去解决相应的问题就会花费不小的代价，不如从一开始就遵循这些已经在其他成熟团队中积淀的经验和法则。对于应用开发来说，很多高深的技术和复杂的框架也许并不会对最终的结果带来很大的帮助，学习一些业界真实的方案，并对其进行扩展可能是更加稳妥的方式。

从 Android 和 iOS 诞生至今，技术虽然一直在进步，但它们分别是由 Google 和 Apple 主导的。开源社区虽然有很多热门的项目，但是不同于服务端的 Apache 扶持的大型开源项目，客户端受限于体积、硬件及部署方式的限制，一直没有形成大而全的框架，反而出色的开源项目都聚焦在一个点上。回想 Joe Hewitt 当年在 Facebook 开源的 Three20 项目引领了当时的 iOS 应用架构，到目前已经被大多数的应用抛弃，只能说这是一个大浪淘沙的时代，移动技术在飞速发展，技术被淘汰的速度非常之快。优秀的开发人员需要具备的不光是对平台的了解和写代码的能力，更重要的是对技术的整合和对发展趋势的理解。本书就像是对 2015 年整个移动技术的一份快照，非常富有这个时代的特征。整本书并不是从枯燥的文档提炼而来，而是真切地从一个互联网从业者的切身经历和与他人的交流中得来。对于一个需要时刻紧跟移动浪潮的 App 开发人员来说，本书是值得一读的好书。

屠毅敏

大众点评首席架构师

前　言 *Preface*

皇皇三十载，书剑两无成

在你面前娓娓而谈的我，曾经是一位技术宅男。我写了 6 年的技术博客，500 多篇技术文章。十年编程生涯，我学习了 .NET 的所有技术，但是从微软出来，踏上互联网这条路，却发现自己还是小学生水平，当时恰逢三十而立之年，感慨自己多年来一事无成，于是又开始了新一轮的学习。选择移动互联网这个方向，是因为这个领域所有人都是从零开始，大家都是摸索着做，初期没有高低上下之分。

在此期间，我做过 Window Phone 的 App，学会了 Android 和 iOS，慢慢由二把刀水平升级到如今的著书立说，本来我想写的是 iOS 框架设计，因为当时这方面的经验积累会更多一些，2013 年的时候我在博客上写了一系列这方面的文章，可惜没有写完。如今这本书是以 Android 为主，但是框架设计的思想是和 iOS 一致的。

作为程序员，不写本书流传于世，貌似对不起这个职业。2008 年的时候我就想写，可那时候积累不够，所知所会多是从书本上看到的，所以没敢动笔，而是选择翻译了一本书《 MSIL 权威指南》。翻译途中发现，我只能老老实实地按照原文翻译，而不能有所发挥。我渴望能有一个地方，天马行空地将自己的风格淋漓尽致地表现出来，在写这本书之前，只有我的技术博客。

终于给了自己一个交代，东隅已逝，桑榆非晚。

文章本天成，妙手偶得之

这是一本前后风格迥异的书，以至于完稿后，不知道该给本书起一个什么样的书名。只希望各位读者看过之后能得到一些启示，我就心满意足了。

下面介绍一下本书的章节概要。本书分为三个部分共计 12 章。

第 1 章讲重构。这是后续 3 章的基础。先别急着看其他章节，先看一下这一章介绍的内

容，你的项目是否都做到了。

第 2 章讲网络底层封装。各个公司都对 App 的网络通信进行了封装，但都稍显臃肿。我介绍的这套网络框架比较灵巧，而且摆脱了 AsyncTask 的束缚，可以在底层或上层快速扩展新的功能。这样讲多少有些自卖自夸，好不好还是要听读者的反馈，建议在新的 App 上使用。

第 3 章讲 App 中一些经典的场景设计，比如说城市列表的增量更新、缓存的设计、App 与 HTML5 的交互、全局变量的使用。对于这些场景，各位读者是否有似曾相识的感觉，是否能从我的解决方案中产生共鸣？

第 4 章介绍 Android 的命名规范和编码规范。网上的各种规范多如牛毛，但我们不能直接拿来就使用，要有批判地继承吸收，要总结出适合自己团队的规范。所以，即使是我这章内容，也请各位读者有选择地采纳。我写这一章的目的，就是要强调“无规矩不成方圆”，代码亦如是。

第 5 章和第 6 章组成了 Android 崩溃分析三部曲。写这本书用了一年，其中有半年多时间花在这两章上。一方面，要不断优化自己的算法，训练机器对崩溃进行分类；另一方面，则是对八十多种线上崩溃追根溯源，找到其真正的原因。

第 7 章讲 Android 中的代码混淆。本不该有这一章，只是在工作中发现网上关于 ProGuard 的介绍大都只言片语。官方倒是有一份白皮书，但是针对 Android 的介绍却不是很多，于是便写了这章，系统而全面地介绍了在 Android 中使用 ProGuard 的理论和实践。

第 8 章讲持续集成（CI）。十年传统软件的经验，使我在这方面得心应手。这一章所要解决的是，如何把传统软件的思想迁移到 App 上。

第 9 章讲 App 竞品分析，是研究了市场上几十款著名 App 并参阅了大量技术文章后写出的。之前积累了十年的软件研发经验，这时极大地帮助了我。

第 10 章讲项目管理，是为 App 量身打造的敏捷过程，是我在团队中一直坚持使用的开发模式。App 一般 2 周发一次版本，迭代周期非常快，适合用敏捷开发模式。

第 11 章讲日常工作中的问题解决办法。那是在一段刀尖上舔血的日子中总结出的办法，那时每天都在战战兢兢中度过，有问题要在最短时间内查找到原因并尽可能修复；那也是个人能力提升最快的一段时光，每一次成功解决问题都伴随着个人的成长。

第 12 章讲 App 团队建设。我是一个孔雀型性格的老板，所以我的团队中多是外向型的人，或者说，把各种闷骚型技术宅男改造成明骚；我是从技术社区走出来的，所以我会推崇技术分享，关心每个人的成长；我有 8 年软件公司的工作经验，所以我擅长写文档、画流程图，以确保一切尽在掌握之中。有这样一位奇葩老板，对面的你，还不快到我的碗里来，我的邮箱是 16230091@qq.com，我的团队，期待你的加入。

心如猛虎，细嗅蔷薇

话说，我也是无意间踏上编程这条道路的。如果不是在大三实在学不明白实变函数这门课的话，我现在也许是一个数学家，或者和我的那些同学一样做操盘手或是二级市场。

我真正的爱好是看书，最初是资治通鉴、二十四史，后来发现在饭桌上说这些会被师弟师妹们当做怪物，于是按照中文系同学的建议翻看张爱玲、王小波的小说，读梁实秋的随笔。在复旦的四年时光，熏出了一身的“臭毛病”，比如说看着夜空中的月亮会莫名其妙地流眼泪，会喜欢喝奶茶并且挑剔珍珠的口感。

不要以为程序员只会写代码。程序员做烘焙绝对是逆天的，因为这用到软件学中的设计模式，我也曾研发出失败的甜品，做饼干时把黄油错用成了淡奶油，然后把烤得硬邦邦的饼干第二天拿给同事们吃。

我涉及的领域还有很多，比如煮咖啡、唱 K、看老电影，都是在编程技术到了一定瓶颈后学会的，每一类都有很深的学问。不要一门心思地看代码，生活能教会我们很多，然后反过来让我们对编程有更深刻的认识。

心若有桃园，何处不是水云间。

会当凌绝顶，一览众山小

如果后续还有第二卷，我希望是讲数据驱动产品。就在本书写作期间，我的思想发生了一次升华，那是在 2015 年初的一个雪夜，我完成了从纠结于写代码的方法到放眼于数据驱动产品的转变。这也是这本书前面代码很多，越到后面代码越少的原因。

数据驱动产品是未来十年的战略布局。之前，我们过多地关注于写代码的方法了，却始终搞不清用户是否愿意为我们辛辛苦苦做出来的产品买单，技术人员不知道，产品人员更不知道。产品人员需要技术人员提供工具来帮助他们进行分析，比如说 ABTest，比如说精准推送平台，比如说用户画像，而我们检查自己的代码，却发现连 PV 和 UV 都不能确保准确。

这也是我接下来的研究和工作方向。

本书全部代码均可以从作者的博客上下载，地址是：www.cnblog.com/Jax/p/4656789.html。

包建强

2015 年 8 月 3 日于北京

Contents 目　录

第三部分 项目管理和团队建设

第一部分 Part 1

高效 App 框架设计与重构

对于 App 来说，要么就一次性把它设计好，否则，就只能重构了。

什么时候做重构？作为 App 技术团队的负责人，我每次想到这一点，都会掂量再三。在我看来，产品需求是优先级最高的。开发团队要使尽浑身解数，优先完成这些需求。但这样一来，就没有时间做重构了。长此以往，积弊难返，代码会越来越难维护。

另一个更重要的问题是，现在互联网严重缺人，各大公司的各个部门都不饱和。也就是说，我们可能连需求都做不完，更不要提重构的事情了。

对于新项目，一开始就要把它设计好了，因为我们不会再有重构的机会了。互联网的发展现状是：没有时间给我们来回折腾。

对于老项目，我们就得掰着手指头仔细盘算盘算是否要重构：

1）如果不重构，会导致很严重的 App 功能问题，那么这是很严重的 bug，宁肯砍掉 1 ~ 2 个功能也一定要修复的，需要和产品经理商量，由他们决策。

2）本次迭代是否还有空余的开发人员来做重构？有，就做；没有，也不勉强。同时，重构也会导致测试团队额外的工时，如果测试团队没有额外的人力对重构进行测试，那么开发人员就要把重构做在新建的代码分支上，本期迭代上线后，再把代码合并，放到下期迭代。

3）重构计划要事先给出，但是绝对不能超过两周，这对于重构小的功能是可以的，但是要重构大的模块或者底层架构，就要考虑拆分重构计划的事情了。我们可以把重构划分为几次迭代，分期上线，先把最严重的问题解决了。

当前移动互联网已经过了几年前的草创时期，目前各家公司比拼的都是内功，就是看谁做得细致。谁家的交互做得好，谁家的崩溃少，谁就占据了市场和用户，马虎不得。然而说得不客气些，目前市面上的 App 做得都很糙。

本书第一部分要讲的就是怎么设计 App 应用开发框架，怎么进行重构。

好戏即将上演。

第 1 章 Chapter 1

重构，夜未眠

本章将要讨论的主题是对项目进行重构，进而搭建一套简单实用的 Android 应用框架 AndroidLib。AndroidLib 框架将封装业务无关的逻辑，从而将业务逻辑独立出来，为 Android 模块化拆分和 Android 插件化编程打下了基础。

值得一提的是 1.4 节，尤其是那个经过改良的实体生成器，是为 App 量身打造的一款利器。

1.1 重新规划 Android 项目结构

庭院深深深几许，杨柳堆烟，帘幕无重数。用这首词来形容当前市面上 Android 项目的代码再贴切不过了。

我做过很多 App，少则 70 多个页面，多则 200 个页面左右，我的切身感受是，无论什么 App，开发人员都喜欢把所有的代码、类放在一个项目下，这也就罢了，更有甚者，无论是 Activity 还是 Adapter 都位于一个 Package 下，或者将 Adapter 内置在 Activity 中。这就相当于一个房间里既有餐桌又有马桶，床上还放着酱油瓶。

我们需要重新规划 Android 项目的目录结构，分两步走：

第一步：建立 AndroidLib 类库，将与业务无关的逻辑转移到 AndroidLib。重构后的项目结构请参见图 1-1，其中 YoungHeart 是主项目，保持了对 AndroidLib 类库的引用。

如何将 AndroidLib 项目设置为类库，以及如何在 YoungHeart 项目中添加对 AndroidLib 类库的引用，这些我就不多说了，请参考相关教程。

AndroidLib 中应该包括哪些业务无关的逻辑呢？应至少包括五大部分，如图 1-2 所示。

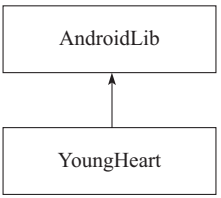


图 1-1 重构后的项目依赖关系

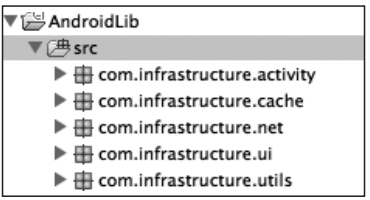


图 1-2 AndroidLib 项目结构

这几部分的说明如下：

- activity 包中存放的是与业务无关的 Activity 基类。Activity 基类要分两层，如图 1-3 所示。AndroidLib 下的基类 BaseActivity 封装的是业务无关的公用逻辑，主项目中的 AppBaseActivity 基类封装的是业务相关的公用逻辑。
- net 包里面存放的是网络底层封装。这里封装的是 AsyncTask。
- cache 包里面存放的是缓存数据和图片的相关处理。
- ui 包中存放的是自定义控件。
- utils 包中存放的是各种与业务无关的公用方法，比如对 SharedPreferences 的封装。

第二步：将主项目中的类分门别类地进行划分，放置在各种包中，如图 1-4 所示。

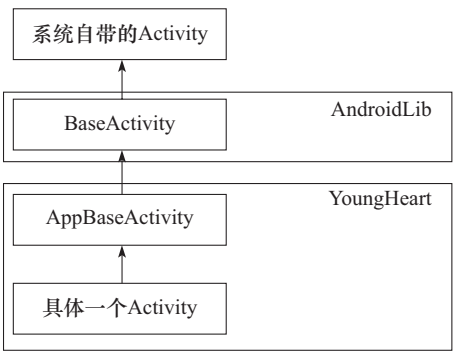


图 1-3 基类的继承关系

图 1-4 Android 重构后的项目结构

对图 1-4 中各个包的介绍如下：

- activity：我们按照模块继续拆分，将不同模块的 Activity 划分到不同的包下。
- adapter：所有适配器都放在一起。
- entity：将所有的实体都放在一起。
- db：SQLLite 相关逻辑的封装。
- engine：将业务相关的类都放在一起。
- ui：将自定义控件都放在这个包中。
- utils：将所有的公用方法都放在这里。
- interfaces：真正意义上的接口，命名以 I 作为开头。

❑ listener：基于 Listener 的接口，命名以 On 作为开头。

这些划分主要是为了以下两个目的：

1）每个文件只有一个单独的类，不要有嵌套类，比如在 Activity 中嵌套 Adapter、Entity。

2）将 Activity 按照模块拆分归类后，可以迅速定位具体的一个页面。此外，将开发人员按照模块划分后，每个开发人员都只负责自己的那个包，开发边界线很清晰。

曾经有人问我，Activity 按照模块拆分了，为什么 Adapter、Entity 不如法炮制也进行相应的拆分呢？这个问题其实是可以商量的。我不做拆分的原因是，看代码时，肯定是先找到页面从 Activity 看起，而不会从 Adapter 看起，所以把 Activity 分好类就够了。此外，Adapter 的逻辑大同小异，如果开发人员都严格遵守 Android 编码，那么代码中的方法和实现基本相同。这就有别于 Activity 了，每个 Activity 都有着很复杂的业务逻辑，所以 Activity 才是最重要的。

Entity 也是这个样子，Entity 中应该只有属性，否则就不叫 Entity。只是当 Entity 有上百个时，就需要考虑按照模块划分。

由于 Entity 中应该只有属性，不应该有业务逻辑的方法，那么如果确实需要，我们就要将其转移到 Engine 这个包中的某个类下面，这也是 Engine 这个包的存在意义。

主席说过：“打扫干净屋子再请客。”对于项目而言，划分好组织结构也是这个道理。我们只有把项目结构规划好，才能进行下一步的重构工作。

1.2　为 Activity 定义新的生命周期

学习过设计模式的人，应该对 SOLID 原则不陌生吧。其中有一条原则就是：单一职责。单一职责的定义是：一个类或方法，只做一件事情。

用这条原则来观察 Activity 中的 onCreate 方法，你会发现，这哥们儿怎么干那么多事啊，onCreate 中的代码如下所示：

```
public class LoginActivity extends Activity implements View.OnClickListener {
    private int loginTimes;

    private EditText etPassword;
    private EditText etEmail;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);

        loginTimes = -1;

        Bundle bundle = getIntent().getExtras();
```

```
        String strEmail = bundle.getString(AppConstants.Email);

        etEmail = (EditText) findViewById(R.id.email);
        etEmail.setText(strEmail);
        etPassword = (EditText) findViewById(R.id.password);

        // 登录事件
        Button btnLogin = (Button) findViewById(
                R.id.sign_in_button);
        btnLogin.setOnClickListener(this);

        // 获取 2 个 MobileAPI，获取天气数据，获取城市数据
        loadWeatherData();
        loadCityData();
    }
```

这段代码主要包括三部分逻辑：

- 接收从其他页面传递过来的 Intent 参数。
- 加载布局文件，并初始化页面的控件，为控件挂上点击事件方法。
- 调用 MobileAPI 获取数据。

那我们为什么不把 onCreate 方法拆成三个子方法呢？如图 1-5 所示。

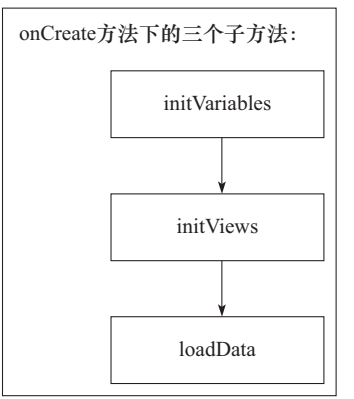


图 1-5 onCreate 方法下的三个子方法

对这些子方法介绍如下：

- initVariables：初始化变量，包括 Intent 带的数据和 Activity 内的变量。
- initViews：加载 layout 布局文件，初始化控件，为控件挂上事件方法。
- loadData：调用 MobileAPI 获取数据。

于是我们在 AndroidLib 这个类库的 BaseActivity 中，重写 onCreate 方法：

```
public abstract class BaseActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        initVariables();
        initViews(savedInstanceState);
        loadData();
    }

    protected abstract void initVariables();
    protected abstract void initViews(Bundle savedInstanceState);
    protected abstract void loadData();
}
```

同时这三个子方法，要声明为 abstract 的，从而要求所有子类必须实现这三个方法。

然后我们重写刚才那个 Activity 的实现，要让所有的 Activity 都继承自 BaseActivity 基类，如下所示：

```
public class LoginNewActivity extends BaseActivity
        implements View.OnClickListener {
    private int loginTimes;
    private String strEmail;

    private EditText etPassword;
    private EditText etEmail;
    private Button btnLogin;

    @Override
    protected void initVariables() {
        loginTimes = -1;

        Bundle bundle = getIntent().getExtras();
        strEmail = bundle.getString(AppConstants.Email);
    }

    @Override
    protected void initViews(Bundle savedInstanceState) {
        setContentView(R.layout.activity_login);

        etEmail = (EditText)findViewById(R.id.email);
        etEmail.setText(strEmail);
        etPassword = (EditText)findViewById(R.id.password);

        //登录事件
        btnLogin = (Button)findViewById(R.id.sign_in_button);
        btnLogin.setOnClickListener(this);
    }

    @Override
    protected void loadData() {
        //获取 2 个 MobileAPI，获取天气数据，获取城市数据
        loadWeatherData();
        loadCityData();
    }
```

对 Activity 生命周期重新定义是借鉴了 JavaScript 的做法。JavaScript 因为是脚本语言，所以必须要细化每个方法，才能保证结构清晰，不至于写错变量和语法。

1.3　统一事件编程模型

接上节，我们给按钮点击事件增加方法，如下所示：

```
public class LoginActivity extends Activity
        implements View.OnClickListener {

    @Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    // 以上省略一些无关代码

    // 登录事件
    Button btnLogin = (Button) findViewById(
            R.id.sign_in_button);
    btnLogin.setOnClickListener(this);

    // 以上省略一些无关代码
}

@Override
public void onClick(View view) {
    switch (view.getId()) {
    case R.id.sign_in_button:
        Intent intent = new Intent(LoginActivity.this,
                PersonCenterActivity.class);
        startActivity(intent);
    }
}
```

很多公司、很多团队、很多程序员都是这样写代码的，也不能说不对。但我反对这样写的原因是，大家看那个 onClick 方法，里面要使用 switch…case…语句来对 R.id.btnNext 中的值进行判断，我不希望 R 这个类在程序中反复出现，这会扰乱面向对象编程的风格，按照我的设想，我们在 initViews 方法中一次性把所有的控件都初始化了，今后就再也不会使用 R.id 了。

Android 中还有另一种事件编程方式，如下所示：

```
// 登录事件
btnLogin = (Button)findViewById(R.id.sign_in_button);
btnLogin.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            gotoLoginActivity();
        }
    });
```

这是我比较推崇的方式，有以下两个优点：

1）直接在 btnLogin 这个按钮对象上增加点击事件，是面向对象的写法。

2）将 onClick 方面的实现，封装成一个 gotoLoginActivity 方法，如下所示：

```
private void gotoLoginActivity() {
    Intent intent = new Intent(LoginNewActivity.this,
        PersonCenterActivity.class);
    startActivity(intent);
}
```

这样 onClick 事件方法就不那么臃肿了。设想当我们在 initViews 方法中声明了 10 个按

钮对象，并都给它们挂上不同的点击方法，那么 initViews 方法该有多少行代码呢？我写过上千行的，直接感受就是 initViews 方法很难维护。但是我们把这些点击方法都分别封装到私有方法中，代码就清晰多了。

但是，只要在一个团队内部达成了协议，决定使用某种事件编程方式，所有开发人员就要按照同样的方式编写代码。我认为这是没错的。只要不是各有各的编码风格就好。

1.4　实体化编程

听说过 fastJSON 吗？听说过 GSON 吗？我面试过很多 Android 开发人员，他们的项目大多不用 fastJSON 或者 GSON 这种实体化编程的思路。他们在获取 MobileAPI 网络请求返回的 JSON 数据时，使用 JSONObject 或者 JSONArray 来承载数据，然后把返回的数据当作一个字典，根据键取出相应的值。

1.4.1　在网络请求中使用实体

如果仅仅是在转换 MobileAPI 返回的 JSON 数据时手动取值也就算了，只要能把取到的值填充到一个实体中就成。但是我见过最糟糕的程序是，把 JSON 数据直接转成 JSONObject 或者 JSONArray，然后就一直使用这样的对象了，甚至将 JSONObject 从一个 Acivity 传递到另一个 Activity，要知道 JSONObject 和 JSONArray 都是不支持序列化的，所以只好将这种对象封装到一个全局变量中，在跳转前设置，在跳转后取出，写一个这样的糟糕示例：

先给出 MobileAPI 返回的 JSON 字符串：

```
{ "weatherinfo":{
    "city":"北京",
    "cityid":"101010100",
    "temp":"24",
    "WD":"南风",
    "WS":"2 级",
    "SD":"74%",
    "WSE":"2",
    "time":"17:45",
    "isRadar":"1",
    "Radar":"JC_RADAR_AZ9010_JB",
    "njd":"暂无实况",
    "qy":"1005"
  }
}
```

使用 JSONObject 的编码如下，代码中的 result 变量就是上面的 JSON 字符串，更详细的 Demo 请参见 WeatherByJsonObjectActivity：

```
try {
    JSONObject jsonResponse = new JSONObject(result);
    JSONObject weatherinfo = jsonResponse
            .getJSONObject("weatherinfo");
    String city = weatherinfo.getString("city");
    int cityId = weatherinfo.getInt("cityid");

    tvCity.setText(city);
    tvCityId.setText(String.valueOf(cityId));
} catch (JSONException e) {
    e.printStackTrace();
}
```

这样的写法有以下两个问题：

1）根据 key 值取 value，我们可以认为这是一个字典。同样的功能实现，字典比实体更晦涩难懂，容易产生 bug。

2）每次都要手动从 JSONObject 或者 JSONArray 中取值，很烦琐。

接下来我们分别使用 fastJSON 和 GSON，介绍一下实体编程的方式，相应的，请在项目中添加对 fastJSON 和 GSON 这两个 jar 的引用，如图 1-6 所示。

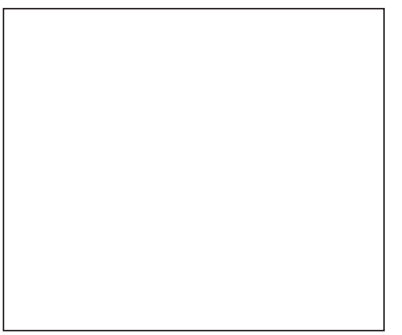


图 1-6 在 Android 项目中添加 fastJSON 和 GSON 的 jar 包

我们使用 fastJSON 对上述代码进行改造，要事先准备两个实体 WeatherEntity 和 WeatherInfo，用于 JSON 字符串到实体之间的映射：

```
WeatherEntity weatherEntity = JSON.parseObject(content, WeatherEntity.class);
WeatherInfo weatherInfo = weatherEntity.getWeatherInfo();
if (weatherInfo != null) {
    tvCity.setText(weatherInfo.getCity());
    tvCityId.setText(weatherInfo.getCityid());
}
```

使用 GSON 的方式也差不多：

```
Gson gson = new Gson();
WeatherEntity weatherEntity = gson.fromJson(content, WeatherEntity.class);
WeatherInfo weatherInfo = weatherEntity.getWeatherInfo();
if (weatherInfo != null) {
    tvCity.setText(weatherInfo.getCity());
    tvCityId.setText(weatherInfo.getCityid());
}
```

这里说一件非常狗血的事情，就是在我们使用 fastJSON 后，App 四处起火，主要表现为：

1）加了符号 Annotation 的实体属性，一使用就崩溃。

2）当有泛型属性时，一使用就崩溃。

在调试的时候没事，可是每次打签名混淆包，就会出现上述问题。我们几个开发人员曾经查到晚上十点半，最后才发现是混淆文件缺了以下两行代码导致的：

```
-keepattributes Signature          // 避免混淆泛型
-keepattributes *Annotation*       // 不混淆注解
```

1.4.2　实体生成器

当使用实体编程的时候，我有个切身感受，就是每次根据 JSON 字符串去编写一个实体的时候非常麻烦。不仅仅是 Android，当我们进行 iOS 和 WindowsPhone 编程时，也需要把 JSON 转换为相应的实体。

创建实体是一件很烦琐的事情，我们需要一个工具，帮助我们自动生成不同开发平台下的实体。于是便有了 EntityGenerater 这个工具。就像马云说的那样，工具都是懒人发明的。当初我在推进实体化编程的时候，我的 iOS 团队早已习惯了字典式取数据的方式，对建立实体这种新机制不是很感兴趣，除非我发明一个能够自动生成实体的工具，提高开发效率。于是我就到开源社区找到了一个类似的工具 JSON C# Class Generator[㊀]，但是它只能生成 WindowsPhone 的实体，于是我就稍微改造了一下这个工具，让它同时也可以生成 Android 和 iOS 实体，如图 1-7 所示。

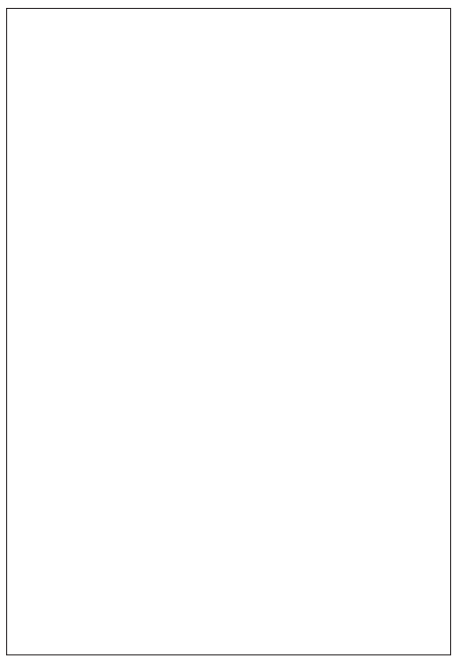


图 1-7　实体生成器的左边界面

在左边的文本框输出 JSON 字符串后，点击 Load 按钮，就会在右边的列表中预览到实体间的层次关系，以及 JSON 字符串中的字段与 JSON 实体中的属性之间的对应关系，如图 1-8 所示。

同时，这个列表还是可以编辑的。我们可以灵活修改要生成的 JSON 实体的属性名称。点击 Generate 按钮，就会在 C:\JSON 目录下生成 JSON 实体了。

再后来，考虑到 iOS 团队每次使用实体生成器都要切换到 Windows 系统，这是一件何其麻烦的事情啊。于是我就又开发了实体生成器的 Web 版本，这样就能满足所有团队的需要了。

经过我修改的 EntityGenerator 项目源码，请到我的博客下载，读者可以根据自己的需要定制自己的实体格式[㊁]。

㊀ 这个工具的地址如下：http://www.xamasoft.com/json-class-generator/

㊁ 项目地址如下：http://files.cnblogs.com/Jax/EntityGenerator.zip。

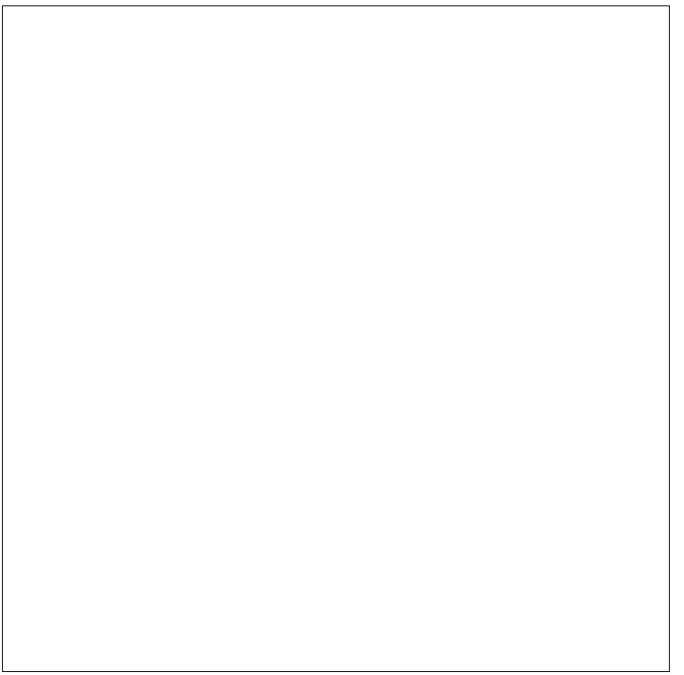


图 1-8 实体生成器的右边界面

1.4.3 在页面跳转中使用实体

在一个页面中，数据的来源有两种：

1）调用 MobileAPI 获取 JSON 数据。

2）从上一个页面传递过来。

我们上一小节介绍了如何将从 MobileAPI 请求到的 JSON 数据转换为实体，接下来，我们看一下 Activity 之间的数据应该如何传递。

一种偷懒的办法是，设置一个全局变量，在来源页设置全局变量，在目标页接收全局变量。

以下是来源页 MainActivity 的代码：

```
Intent intent = new Intent(MainActivity.this, LoginActivity.class);
intent.putExtra(AppConstants.Email, "jianqiang.bao@qq.com");

CinemaBean cinema = new CinemaBean();
cinema.setCinemaId("1");
cinema.setCinemaName(" 星美 ");

// 使用全局变量的方式传递参数
GlobalVariables.Cinema = cinema;

startActivity(intent);
```

以下是目标页 LoginActivity 的代码：

```
CinemaBean cinema = GlobalVariables.Cinema;
if (cinema != null) {
    cinemaName = cinema.getCinemaName();
} else {
    cinemaName = "";
}
```

这里的 GlobalVariables 类是一个全局变量，定义如下：

```
public class GlobalVariables {
    public static CinemaBean Cinema;
}
```

我是不建议使用全局变量的。App 一旦被切换到后台，当手机内存不足的时候，就会回收这些全局变量，从而当 App 再次切换回前台时，再继续使用全局变量，就会因为它们为空而崩溃。

如果必须使用全局变量，就一定要把它们序列化到本地。这样即使全局变量为空，也能从本地文件中恢复。在 3.5 节，我会专门讲解如何解决全局变量导致 App 崩溃的问题。

本节我们着重研究如何不使用全局变量，而是使用 Intent 在页面间来传递数据实体的机制。

首先，在来源页 MainActivity 要这样写：

```
Intent intent = new Intent(MainActivity.this, LoginNewActivity.class);
intent.putExtra(AppConstants.Email, "jianqiang.bao@qq.com");

CinemaBean cinema = new CinemaBean();
cinema.setCinemaId("1");
cinema.setCinemaName("星美");

//使用 intent 上挂可序列化实体的方式传递参数
intent.putExtra(AppConstants.Cinema, cinema);

startActivity(intent);
```

其次，目标页 LoginActivity 要这样写：

```
CinemaBean cinema = (CinemaBean)getIntent()
        .getSerializableExtra(AppConstants.Cinema);
if (cinema != null) {
    cinemaName = cinema.getCinemaName();
} else {
    cinemaName = "";
}
```

这里的 CinemaBean 要实现 Serializable 接口，以支持序列化：

```
public class CinemaBean implements Serializable {
```

```
    private static final long serialVersionUID = 1L;

    private String cinemaId;
    private String cinemaName;

    public CinemaBean() {

    }

    public String getCinemaId() {
        return cinemaId;
    }
    public void setCinemaId(String cinemaId) {
        this.cinemaId = cinemaId;
    }
    public String getCinemaName() {
        return cinemaName;
    }
    public void setCinemaName(String cinemaName) {
        this.cinemaName = cinemaName;
    }
}
```

1.5 Adapter 模板

我在进行重构的时候还发现，如果不对 Adapter 的写法进行规范，开发人员还是会根据自己的习惯，写出来各种各样的 Adapter，比如：

- 很多开发人员都喜欢将 Adapter 内嵌在 Activity 中，一般会使用 SimpleAdapter。
- 由于没有使用实体，所以一般会把一个字典作为构造函数的参数注入到 Adapter 中。

而我希望 Adapter 只有一种编码风格，这样发现了问题也很容易排查。

于是我们要求所有的 Adapter 都继承自 BaseAdapter，从构造函数注入 List< 自定义实体 > 这样的数据集合，从而完成 ListView 的填充工作，如下所示：

```
public class CinemaAdapter extends BaseAdapter {
    private final ArrayList<CinemaBean> cinemaList;
    private final AppBaseActivity context;

    public CinemaAdapter(ArrayList<CinemaBean> cinemaList,
            AppBaseActivity context) {
        this.cinemaList = cinemaList;
        this.context = context;
    }

    public int getCount() {
        return cinemaList.size();
    }
```

```
    public CinemaBean getItem(final int position) {
        return cinemaList.get(position);
    }

    public long getItemId(final int position) {
        return position;
    }
```

对于每个自定义的 Adapter，都要实现以下 4 个方法：

- ❑ getCount()
- ❑ getItem()
- ❑ getItemId()
- ❑ getView()

此外，还要内置一个 Holder 嵌套类，用于存放 ListView 中每一行中的控件。ViewHolder 的存在，可以避免频繁创建同一个列表项，从而极大地节省内存，如下所示：

```
class Holder {
    TextView tvCinemaName;
    TextView tvCinemaId;
}
```

你可能会觉得我老生常谈，但当有很多列表数据时，快速滑动列表会变得很卡，其实就是因为没有使用 ViewHolder 机制导致的，正确的写法如下所示：

```
public View getView(final int position, View convertView,
        final ViewGroup parent) {
    final Holder holder;
    if (convertView == null) {
        holder = new Holder();
        convertView = context.getLayoutInflater().inflate(
                R.layout.item_cinemalist, null);
        holder.tvCinemaName = (TextView) convertView.
                findViewById(R.id.tvCinemaName);
        holder.tvCinemaId = (TextView) convertView.
                findViewById(R.id.tvCinemaId);
        convertView.setTag(holder);
    } else {
        holder = (Holder) convertView.getTag();
    }

    CinemaBean cinema = cinemaList.get(position);
    holder.tvCinemaName.setText(cinema.getCinemaName());
    holder.tvCinemaId.setText(cinema.getCinemaId());
    return convertView;
}
```

那么，在 Activity 中，在使用 Adapter 的地方，我们按照下面的方式把列表数据传递过去：

```
@Override
Protected void initViews(Bundle savedlnstanceState) {
    setContentView(R.layout.activity_listdemo);
    lvCinemaList = (ListView) findViewByld(R.id. lvCinemalist);

    CinemaAdapter adapter = new CinemaAdapter(cinemaList, ListDemoActivity.this);
    lvCinemaList.setAdapter(adapter);
    lvCinemaList.setOnItemClickListener(
        new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent,
                    View view, int position, long id) {
                // do something
            }
        });
}
```

1.6 类型安全转换函数

在每天统计线上崩溃的时候，我们发现因为类型转换不正确导致的崩溃占了很大的比例。于是我们去检查程序中所有的类型转换，发现主要集中在两个地方：Object 类型的对象、substring 函数。下面分别说明。

1）对于一个 Object 类型的对象，我们对其直接使用字符串操作函数 toString，当其为 null 时就会崩溃。

比如，我们经常会写出下面这样的程序：

```
int result = Integer.valueOf(obj.toString());
```

一旦 obj 这个对象为空，那么上面这行代码会直接崩溃。

这里的 obj，一般是从 JSON 数据中取出来的，对于 MobileAPI 返回的 JSON 数据，我们无法保证其永远不为空。

比较好的做法是，我们需要编写一个类型安全转换函数 convertToInt，实现如下，其核心思想就是，如果转换失败，就返回默认值：

```
public final static int convertToInt(Object value, int defaultValue) {
    if (value == null || "".equals(value.toString().trim())) {
        return defaultValue;
    }
    try {
        return Integer.valueOf(value.toString());
    } catch (Exception e) {
        try {
            return Double.valueOf(value.toString()).intValue();
        } catch (Exception e1) {
```

```
            return defaultValue;
        }
    }
}
```

我们将这个方法放到 Utils 类下面，每当要把一个 Object 对象转换成整型时，都使用该方法，就不会崩溃了：

```
int result = Utils.convertToInt(obj, 0);
```

以上只是其中一种类型安全转换函数，相应的，我们在 Utils 类中还要提供诸如 Object 到 long、double、String 等类型的类型安全转换函数，以满足我们的不时之需。

2）如果长度不够，那么执行 substring 函数时，就会崩溃。

Java 的 substring 函数有 2 个参数：start 和 end。

对于第一个参数 start，我们的程序大多是设为 0，所以一般不会有问题。但是要设置为大于 0 的值时，就要仔细思量了，比如：

```
String cityName = "T";
String firstLetter = cityName.substring(1, 2);
```

这样的代码必然崩溃，所以每次在使用 substring 函数的时候，都要判断 start 和 end 两个参数是否越界了。应该这样写：

```
String cityName = "T";
String firstLetter = "";
if (cityName.length() > 1) {
    firstLetter = cityName.substring(1, 2);
}
```

以上两类问题的根源，都来自 MobileAPI 返回的数据，由此而引出另一个很严肃的问题，对于从 MobileAPI 返回的数据，可信度到底有多高呢?

首先，不能让 App 直接崩溃，应该在解析 JSON 数据的外面包一层 try…catch…语句，将截获到的异常在 catch 中进行处理，比如说，发送错误日志给服务器。

其次，对数据要分级别对待，例如：

1）对于那些不需要加工就能直接展示的数据，我们是不担心的，因为即使为空，页面上也就是不显示而已，不会引起逻辑的问题。

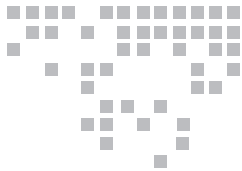
2）对于那些很重要的数据，比如涉及支付的金额不能为空的逻辑，这时候就应该弹出提示框提示用户当前服务不可用，并停止接下来的操作。

1.7　本章小结

本章介绍的内容都是为后面的章节打基础。有了 AndroidLib 这个业务无关的类库，我

们将在接下来的章节中封装更多的公用逻辑，比如第 2 章介绍的网络底层封装，以及第 9 章介绍的模块化拆分和插件化编程。实体化编程将极大提升代码可读性，从而进一步提高开发效率。而实体生成器的出现，将是解决重复劳动的一大利器。为 Activity 定义新的生命周期，把 onCreate 方法中的几百行代码拆分为 3 个具有不同功用的子方法，也是提升代码可读性的一个手段。

相比后面的章节，本章更多内容是写代码的方法，如果整本书都是这个内容，那就没意思了。接下来的各章，我将详细介绍移动 App 开发领域的常见问题以及解决问题的思路或方法。



第 2 章 Chapter 2

Android 网络底层框架设计

本章介绍 Android 网络底层的封装。很多公司、很多团队都只是把网络底层封装成一个好用的方法，而我接下来要介绍的内容将覆盖的范围很广：

- 抛弃 AsyncTask，自定义一套网络底层的封装框架。
- 设计一套 App 缓存策略。
- 设计一套 MockService 的机制，在没有 MobileAPI 的时候，也能假装获取到了网络返回的数据。
- 封装了用户 Cookie 的逻辑。

好了，让我们开始愉快地阅读吧。

2.1 网络低层封装

很多公司和团队都是用 AsyncTask 来封装网络底层，因为这个类非常好用，内部封装了很多好用的方法，但缺点是可扩展性不高。

本节将介绍一种自定义的网络底层框架，我们可以基于这个框架随心所欲地增加自定义的逻辑，完成很多高级功能。

让我们先从 MobileAPI 网络请求格式入手。

2.1.1 网络请求的格式

对于网络请求，我们一般定义为 GET 和 POST 即可，GET 为请求数据，POST 为修改数据（增删改）。

1. Request 格式

所有的 MobileAPI 都可以写作 http://www.xxx.com/aaaa.api 的形式。

- 对于 GET，我们可以写作：http://www.xxx.com/aaaa.api?k1=va&k2=v2 的形式，也就是说，把 key-value 这样的键值对存放在 URL 上。之所以这样设计，是为了更方便地定义数据缓存。我们尽量使 GET 的参数都是 string、int 这样的简单类型。
- 对于 POST，我们将 key-value 这样的键值对存放在 Form 表单中，进行提交。POST 经常会提交大量数据，所以有些键值对要定义成集合或者复杂的自定义实体，这时我们就需要将这样的值转换为 JSON 字符串进行提交，由 App 传递到 MobileAPI 后，再将 JSON 字符串转换为对应的实体。

上述介绍只是一家之言，不同公司有不同的实现方式，这取决于服务器端的设计。

2. Response 格式

我们一般使用 JSON 作为 MobileAPI 返回的结果。最规范的 JSON 数据返回格式如下。

JSON 数据格式 1：

```
{
    "isError" : true,
    "errorType" : 1,
    "errorMessage" : "网络异常",
    "result" : ""
}
```

JSON 数据格式 2：

```
{
    "isError" : false,
    "errorType" : 0,
    "errorMessage" : "",
    "result" : {
        "cinemaId" : 1,
        "cinemaName" : "星美"
    }
}
```

这里，isError 是调用 MobileAPI 成功与否，errorType 是错误类型（如果成功则为 0），errorMessage 是错误消息（如果成功则为空），result 是成功请求返回的数据结果（如果失败则返回空）。

既然所有的 JSON 都返回 isError、errorType、errorMessage、result 这 4 个字段，我们不妨定义一个 Response 实体类，作为所有 JSON 实体的最外层，代码如下所示：

```
public class Response
{
    private boolean error;
    private int errorType;        //1 为 Cookie 失效
    private String errorMessage;
```

```
    private String result;

    public boolean hasError() {
        return error;
    }

    public void setError(boolean hasError) {
        this.error = hasError;
    }

    public String getErrorMessage() {
        return errorMessage;
    }

    public void setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public String getResult() {
        return result;
    }

    public void setResult(String result) {
        this.result = result;
    }

    public int getErrorType() {
        return errorType;
    }

    public void setErrorType(int errorType) {
        this.errorType = errorType;
    }
}
```

如果成功返回了数据，数据会存放在 result 字段中，映射为 Response 实体的 result 属性。

上面的 JSON 数据返回的是一笔影院数据，如果返回的 result 是很多影院的数据集合，那么就要把 result 解析为相应的实体集合，如下所示：

```
{
    "isError" : false,
    "errorType" : 0,
    "errorMessage" : "",
    "result" : [
        {"cinemaId" : 1, "cinemaName" : "星美"},
        {"cinemaId" : 2, "cinemaName" : "万达"}
    ]
}
```

2.1.2　AsyncTask 的使用和缺点

对 AsyncTask 的封装属于网络底层的技术，所以 AsyncTask 应该封装在 AndroidLib 类库

中，而不是具体的项目里。

对网络异常的分类，也就是 Response 类中的 errorType 字段，分析如下：

- 一种是请求发送到 MobileAPI，MobileAPI 执行过程中发现的异常，这时候要自定义错误类型，也就是 errorType，比如说 1 是 Cookie 过期，2 是第三方支付平台不能连接，等等，这些已知的错误都是大于 0 的整数，因接口不同而各自定义不同。
- 另一种是在 App 访问 MobileAPI 接口时发生的异常，有可能 App 自身网络不稳定，有可能因为网络传输不好导致返回了空值，这些异常情况我们都标记为负数。

基于上述分析，AsyncTask 的 doInBackground 方法复写为：

```
@Override
protected Response doInBackground(String… url) {
    return getResponseFromURL(url[0]);
}

private Response getResponseFromURL(String url) {
    Response response = new Response();
    HttpGet get = new HttpGet(url);
    String strResponse = null;
    try {
        HttpParams httpParameters = new BasicHttpParams();
        HttpConnectionParams.setConnectionTimeout(httpParameters, 8000);
        HttpClient httpClient = new DefaultHttpClient(httpParameters);

        HttpResponse httpResponse = httpClient.execute(get);
        if (httpResponse.getStatusLine().getStatusCode()
                == HttpStatus.SC_OK) {
            strResponse = EntityUtils.toString(httpResponse.getEntity());
        }
    } catch (Exception e) {
        response.setErrorType(-1);
        response.setError(true);
        response.setErrorMessage(e.getMessage());
    }

    if (strResponse == null) {
        response.setErrorType(-1);
        response.setError(true);
        response.setErrorMessage("网络异常，返回空值");
    } else {
        strResponse = "{'isError':false,'errorType':0,'errorMessage':'',
                'result':{'city':'北京','cityid':'101010100','temp':'17',
                'WD':'西南风','WS':'2级','SD':'54%','WSE':'2','time':'23:15',
                'isRadar':'1','Radar':'JC_RADAR_AZ9010_JB',
                'njd':'暂无实况','qy':'1016'}}";
        response = JSON.parseObject(strResponse, Response.class);
    }
    return response;
}
```

相应的，在 AsyncTask 的 onPostExecute 方法中，我们要对错误类型进行分类，从而进一步回调：

```
public abstract class RequestAsyncTask
        extends AsyncTask<String, Void, Response> {
    public abstract void onSuccess(String content);

    public abstract void onFail(String errorMessage);

    @Override
    protected void onPreExecute() {
    }

    @Override
    protected void onPostExecute(Response response) {
        if(response.hasError()) {
            onFail(response.getErrorMessage());
        } else {
            onSuccess(response.getResult());
        }
    }
```

目前我们只定义了 onSuccess 和 onFail 两个回调函数，将网络返回值简单地分为成功与失败两种情况。在 2.4.3 节，我们将详细介绍如何在网络底层封装对 Cookie 过期时的异常处理。

在相应的 Activity 页面，调用 AysncTask 如下所示：

```
protected void loadData() {
    String url = "http://www.weather.com.cn/data/sk/101010100.html";

    RequestAsyncTask task = new RequestAsyncTask() {

        @Override
        public void onSuccess(String content) {
            // 第 2 种写法，基于 fastJSON
            WeatherEntity weatherEntity = JSON.parseObject(content,
                    WeatherEntity.class);
            WeatherInfo weatherInfo = weatherEntity.getWeatherInfo();
            if (weatherInfo != null) {
                tvCity.setText(weatherInfo.getCity());
                tvCityId.setText(weatherInfo.getCityid());
            }
        }

        @Override
        public void onFail(String errorMessage) {
            new AlertDialog.Builder(WeatherByFastJsonActivity.this)
                    .setTitle(" 出错啦 ").setMessage(errorMessage)
                    .setPositiveButton(" 确定 ", null).show();
        }
    };
    task.execute(url);
}
```

网上关于如何使用 AsyncTask 的文章不胜枚举，大家都在欣赏它的优点，却忽略了它的致命缺点，那就是不能灵活控制其内部的线程池。

线程池里面的每个线程存放的都是 MobileAPI 的调用请求，而 AsyncTask 中又没有暴露出取消这些请求的方法，也就是我们熟知的 CancelRequest 方法，所以，一旦从 A 页面跳转到 B 页面，那么在 A 页面发起的 MobileAPI 请求，如果还没有返回，并不会被取消。

对于一款频繁调用 MobileAPI 的应用类 App 而言，最严重的情况发生在首页到二级页面的跳转，因为在首页会调用十几个 MobileAPI 接口，视网络情况而定，如果是 WiFi，应该很快就能请求到数据，不会产生积压，但如果是 3G 或者 2G，那么请求就会花费很长时间，而我们在这期间就跳转到二级页面，而这个二级页面也会调用 MobileAPI 接口，那么将得不到任何结果，因为首页的请求还在排队处理中，之前的那十几个 MobileAPI 接口的数据还都遥遥无期在线程池里排队呢，就更不要说当前页面这个请求了。

如果你不信，我们可以做个试验。记录每次 MobileAPI 请求发起和接收数据的时间点，你会看到，在迅速进入二级页面后，首页的十几个 MobileAPI 请求只有发起时间并没有返回时间，说明它们还在处理过程中，都被堵塞了。

2.1.3 使用原生的 ThreadPoolExecutor + Runnable + Handler

既然 AsyncTask 有诸多问题，那么退而求其次，使用 ThreadPoolExecutor + Runnable + Handler 的原生方式，对网络底层进行封装。

接下来我将介绍一个非常轻量级的网络底层框架。它由以下 9 个类组成，如图 2-1 所示。

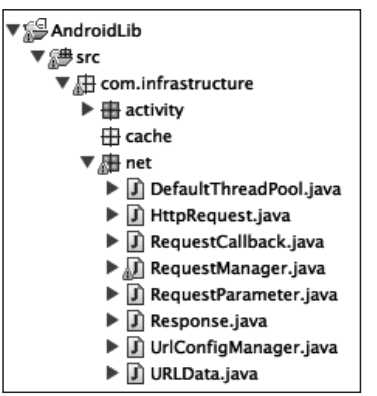


图 2-1 轻量级的网络底层框架

图中只列出了 8 个，还有 1 个 RemoteService 类，位于 YoungHeart 项目的 engine 包中。下面分别介绍。

1. UrlConfigManager 和 URLData

我们把 App 所要调用的所有 MobileAPI 接口的信息都放在 url.xml 文件中，如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<url>
    <Node
        Key="getWeatherInfo"
        Expires="300"
        NetType="get"
        Url="http://www.weather.com.cn/data/sk/101010100.html" />

    <Node
        Key="login"
        Expires="0"
        NetType="post"
        Url="http://www.weather.com.cn/data/login.api" />
</url>
```

在使用上，通过 UrlConfigManager 的 findURL 方法，在上述 xml 文件中找到当前 MobileAPI 调用的节点，其中每一个 MobileAPI 接口都对应一个 URLData 实体，如下所示：

```
public class URLData {
    private String key;
    private long expires;
    private String netType;
    private String url;
```

目前我们只用到 key、url 和 netType 这 3 个属性。expires 是用来做数据缓存的，我们 2.2 节会介绍到它的作用。

这样，发起一次 MobileAPI 网络请求的所有数据就都准备好了。

2. RemoteService 和 RequestCallback、RequestParameter

这里的 3 个类是暴露给 App 用来调用 MobileAPI 接口的，举个例子，在 WeatherByFastJsonActivity 中的调用形式如下：

```
@Override
protected void loadData() {
    weatherCallback = new RequestCallback() {

        @Override
        public void onSuccess(String content) {
            WeatherInfo weatherInfo = JSON.parseObject(content,
                    WeatherInfo.class);
            if (weatherInfo != null) {
                tvCity.setText(weatherInfo.getCity());
                tvCityId.setText(weatherInfo.getCityid());
            }
        }

        @Override
        public void onFail(String errorMessage) {
            new AlertDialog.Builder(WeatherByFastJsonActivity.this)
                    .setTitle("出错啦").setMessage(errorMessage)
                    .setPositiveButton("确定", null).show();
        }
    };

    ArrayList<RequestParameter> params = new ArrayList<RequestParameter>();
    RequestParameter rp1 = new RequestParameter("cityId", "111");
    RequestParameter rp2 = new RequestParameter("cityName", "Beijing");
    params.add(rp1);
    params.add(rp2);

    RemoteService.getInstance().invoke(this, "getWeatherInfo", params,
            weatherCallback);
}
```

对上述方法介绍如下：

- RequestCallback 是回调，目前有 onSuccess 和 onFail 两种。
- RequestParameter 是用来传递调用 MobileAPI 接口所需参数的键值对的。我们原本可以使用 HashMap<String, String> 这样的数据结构，但是 HashMap 比较耗费内存，虽然它的查找速度是 o(1)，而对于 MobileAPI 接口的参数而言，数据一般不会太多，查找速度快体现不出优势来，所以我们使用 ArrayList<RequestParameter> 这样的数据结构。
- RemoteService 这个单例是用来发起请求的，它会创建一个 request，并将其添加到 RequestManager 中，然后放到 DefaultThreadPool 的一个线程中去执行这个 request。

3. RequestManager

RequestManager 这个集合类是用于取消请求（cancelRequest）的。因为每次发起请求，都会把为此创建的 request 添加到 RequestManager 中，所以 RequestManager 保存了全部 request。

从 ActivityA 跳转到 ActivityB，为了不产生阻塞，要取消 ActivityA 中的所有未完成的请求。这时候就需要 RequestManager 的 cancelRequest 方法出力了，它会遍历之前保存的所有 request，不管三七二十一，全部终止。如下所示：

```
public void cancelRequest() {
    if ((requestList != null) && (requestList.size() > 0)) {
        for (final HttpRequest request : requestList) {
            if (request.getRequest() != null) {
                try {
                    request.getRequest().abort();
                    requestList.remove(request.getRequest());
                } catch (final UnsupportedOperationException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

我们在 BaseActivity 中，会保持对 RequestManager 的一个引用，这样在 onDestroy 和 onPause 的时候，执行 RequestManager 的 cancelRequest 方法就可以取消所有未完成的请求了：

```
public abstract class BaseActivity extends Activity {
    // 请求列表管理器
    protected RequestManager requestManager = null;

    protected void onDestroy() {
        // 在 activity 销毁的同时设置停止请求，停止线程请求回调
        if (requestManager != null) {
            requestManager.cancelRequest();
        }
        super.onDestroy();
    }

    protected void onPause() {
```

```
        //在 activity 停止的同时设置停止请求，停止线程请求回调
        if (requestManager != null) {
            requestManager.cancelRequest();
        }
        super.onPause();
    }

    public RequestManager getRequestManager() {
        return requestManager;
    }
}
```

4. DefaultThreadPool

DefaultThreadPool 只是对 ThreadPoolExecutor 和 ArrayBlockingQueue 的简单封装。我们可以认为它就是一个线程池，每发起一次请求（runnable），就由线程池分配一个新的线程来执行该请求。

网上关于 ThreadPoolExecutor 和 ArrayBlockingQueue 的文章不胜枚举，请大家自行参阅。线程池不是本书的重点，这里不再花篇幅去讨论。

5. HttpRequest

HttpRequest 是发起 Http 请求的地方，它实现了 Runnable，从而让 DefaultThreadPool 可以分配新的线程来执行它，所以，所有的请求逻辑都在 Runnable 接口的 run 方法中，其中：

- 对于 get 形式的 MobileAPI 接口，它会把从上层传递进来的 ArrayList<RequestParameter>，解析为 url?k1=v1&k2=v2 这样的形式。
- 对于 post 格式的 MobileAPI 接口，它会把从上层传递进来的 ArrayList<RequestParameter>，转为 BasicNameValuePair 的形式，放到表单中进行提交。

需要注意的是，因为我们把每个 HttpRequest 都放在了新的子线程上执行，所以回调 RequestCallback 的 onSuccess 方法时，不能直接操作 UI 线程上的控件，所以我们在 HttpRequest 类中使用了 Handler：

```
if (responseInJson.hasError()) {
    handleNetworkError(responseInJson.getErrorMessage());
} else {
    handler.post(new Runnable() {

        @Override
        public void run() {
            HttpRequest.this.requestCallback
                    .onSuccess(responseInJson.getResult());
        }

    });
}
```

这样就保证了 RequestCallback 的 onSuccess 方法是在 UI 线程上的，从而可以在 Activity

中编写这样的代码而不报错：

```
weatherCallback = new RequestCallback() {

    @Override
    public void onSuccess(String content) {
        WeatherInfo weatherInfo = JSON.parseObject(content,
                WeatherInfo.class);
        if (weatherInfo != null) {
            tvCity.setText(weatherInfo.getCity());
            tvCityId.setText(weatherInfo.getCityid());
        }
    }
```

Response 这个类就不讨论了，本章前面已经介绍过了。

2.1.4 网络底层的一些优化工作

我们的网络底层越来越强大了，是否有意犹未尽的感受？接下来将完善这个框架，修复其中的一些瑕疵，如 onFail 的统一处理机制、UrlConfigManager 的优化、ProgressBar 的处理等。

1. onFail 的统一处理机制

如果访问 MobileAPI 请求失败，我们一般希望只是在 App 上简单地弹出一个提示框，告诉用户网络有异常。

也就是说，对于每个在 Activity 中声明的 RequestCallback 实例而言，尽管每个 onSuccess 方法的处理逻辑各不相同，但每个 onFail 方法都是一样的逻辑和代码，如下所示：

```
weatherCallback = new RequestCallback() {

    @Override
    public void onSuccess(String content) {
        WeatherInfo weatherInfo = JSON.parseObject(content,
                WeatherInfo.class);
        if (weatherInfo != null) {
            tvCity.setText(weatherInfo.getCity());
            tvCityId.setText(weatherInfo.getCityid());
        }
    }

    @Override
    public void onFail(String errorMessage) {
        new AlertDialog.Builder(WeatherByFastJsonActivity.this)
                .setTitle("出错啦").setMessage(errorMessage)
                .setPositiveButton("确定", null).show();
    }
};
```

我不希望每次都编写同样的 onFail 方法，这会使程序很臃肿。于是在 AppBaseActivity

中写一个自定义类 AbstractRequestCallback，如下所示：

```
public abstract class AppBaseActivity extends BaseActivity {

    public abstract class AbstractRequestCallback
        implements RequestCallback {

        public abstract void onSuccess(String content);

        public void onFail(String errorMessage) {
            new AlertDialog.Builder(AppBaseActivity.this)
                    .setTitle("出错啦").setMessage(errorMessage)
                    .setPositiveButton("确定", null).show();
        }
    }
}
```

那么我们的 weatherRequestCallback 的实例化就可以改写如下，可以看到，不再需要重写 onFail 方法：

```
weatherCallback = new AbstractRequestCallback() {

    @Override
    public void onSuccess(String content) {
        WeatherInfo weatherInfo = JSON.parseObject(content,
                WeatherInfo.class);
        if (weatherInfo != null) {
            tvCity.setText(weatherInfo.getCity());
            tvCityId.setText(weatherInfo.getCityid());
        }
    }
};
```

当然，如果有些 MobileAPI 接口在返回错误时需要 App 特殊处理，比如重启 App 或者啥都不做，我们只需要在实例化 AbstractRequestCallback 时，重写 onFail 方法即可，如下所示。重写的 onFail 方法是一个空方法，表示出错时啥都不做：

```
weatherCallback = new AbstractRequestCallback() {

    @Override
    public void onSuccess(String content) {
        WeatherInfo weatherInfo = JSON.parseObject(content,
                WeatherInfo.class);
        if (weatherInfo != null) {
            tvCity.setText(weatherInfo.getCity());
            tvCityId.setText(weatherInfo.getCityid());
        }
    }

    @Override
    public void onFail(String errorMessage) {
```

```
            // 重启 App 或者啥都不做
        }
    };
```

2. UrlConfigManager 的优化

在 UrlConfigManager 的实现上，我们采取的策略是每发起一次 MobileAPI 请求，都会读取 url.xml 文件，把符合这次 MobileAPI 接口调用的参数取出来。

在一个大量调用 MobileAPI 的 App 中，这样的设计会造成频繁读 xml 文件，性能很差。于是我们对其进行改造，在 App 启动时，一次性将 url.xml 文件都读取到内存，把所有的 UrlData 实体保存在一个集合中，然后每次调用 MobileAPI 接口，直接从内存的这个集合中查找。考虑到内存中的数据会被回收，所以上述这个集合一旦为空，我们要从 url.xml 中再次读取。

基于上述方案，我们对 UrlConfigManager 的 findUrl 方法进行改造：

```
public static URLData findURL(final Activity activity,
        final String findKey) {
    // 如果 urlList 还没有数据（第一次）
    // 或者被回收了，那么（重新）加载 xml
    if (urlList == null || urlList.isEmpty())
        fetchUrlDataFromXml(activity);

    for (URLData data : urlList) {
        if (findKey.equals(data.getKey())) {
            return data;
        }
    }

    return null;
}
```

其中，fetchUrlDataFromXml 方法就不多说了，它的工作就是把 xml 的数据都搬到内存集合 urlList 中。

3. 不是每个请求都需要回调的

有些时候，我们调用一个 MobileAPI 接口，并不需要知道调用成功与否以及返回结果是什么，比如向 MobileAPI 发送打点统计数据。那就是说，我们不需要回调函数了，那么代码可以写为：

```
void loadAPIData3() {
    ArrayList<RequestParameter> params
        = new ArrayList<RequestParameter>();
    RequestParameter rp1 =
            new RequestParameter("cityId", "111");
    RequestParameter rp2 =
            new RequestParameter("cityName", "Beijing");
    params.add(rp1);
```

```
        params.add(rp2);

        RemoteService.getInstance()
                .invoke(this, "getWeatherInfo", params, null);
    }
```

我们将空的 RequestCallback 传给 HttpRequest，那么在 HttpRequest 处理请求返回的结果时，就需要添加 HttpRequest 是否为空的判断，不为空，才会处理返回结果；否则，发起 MobileAPI 请求后什么都不做。

有以下两个地方需要修改：

1）处理请求时：

```
response = httpClient.execute(request);

if ((requestCallback != null)) {
    //获取状态
    final int statusCode =
        response.getStatusLine().getStatusCode();
    if (statusCode == HttpStatus.SC_OK) {
        final ByteArrayOutputStream content =
                    new ByteArrayOutputStream();
```

2）遇到异常，是否要回调 onFail 方法：

```
public void handleNetworkError(final String errorMsg) {
    if ((requestCallback != null)) {
        handler.post(new Runnable() {
            @Override
            public void run() {
                HttpRequest.this.requestCallback
                                .onFail(errorMsg);
            }
        });
    }
}
```

4. ProgressBar 的处理

在调用 MobileAPI 的时候，会显示进度条 ProgressBar，直到返回结果到 onSuccess 或 onFail 回调方法，ProgressBar 才会消失。

由于 App 要保持风格统一，所以所有页面的 ProgressBar 应该长得一样。那么我们就可以将其定义在 AppBaseActivity 中，如下所示：

```
public abstract class AppBaseActivity extends BaseActivity {

    protected ProgressDialog dlg;

    public abstract class AbstractRequestCallback
        implements RequestCallback {
```

```
        public abstract void onSuccess(String content);

        public void onFail(String errorMessage) {
            dlg.dismiss();

            new AlertDialog.Builder(AppBaseActivity.this)
                    .setTitle("出错啦").setMessage(errorMessage)
                    .setPositiveButton("确定", null).show();
        }
    }
}
```

在使用的时候，在开始调用 MobileAPI 的地方，执行 show 方法；在 onSuccess 和 onFail 方法的开始，执行 dismiss 方法：

```
@Override
protected void loadData() {
    dlg = Utils.createProgressDialog(this,
            this.getString(R.string.str_loading));
    dlg.show();

    loadAPIData1();
}

void loadAPIData1() {
    weatherCallback = new AbstractRequestCallback() {

        @Override
        public void onSuccess(String content) {
            dlg.dismiss();
            WeatherInfo weatherInfo = JSON.parseObject(content,
                    WeatherInfo.class);
            if (weatherInfo != null) {
```

不要把 Dialog 的 show 方法和 dismiss 方法封装到网络底层。网络底层的调用经常是在子线程执行的，子线程是不能操作 Dialog、Toast 和控件的。

2.2 App 数据缓存设计

如果以为上一节内容就是网络底层框架的全部，那就错了。那只是网络底层框架的最核心的功能，我们还有很多高级功能没有介绍。在接下来的几节中，我将陆续介绍到这些高级功能。本节先介绍 App 本地的缓存策略。

2.2.1 数据缓存策略

对于任何一款应用类 App，如果访问 MobileAPI 的速度和牛车一样慢，那么就是失败之作。不要在 WiFi 下测试速度，那是自欺欺人，要把 App 放在 2G 或 3G 网络环境下进行测试，才能得到大部分用户的真实数据。

访问 MobileAPI，主要慢在一来一回的传输速度上，对于服务器的处理速度，不需要担心太多，大多数服务器逻辑原本就是支持网站端的，现在只是在外面包了一层，返回给 App 而已。

既然时间主要花在了数据传输上，那么我们就要想一些应对的措施。比如说，减少 MobileAPI 的调用次数。对于一个 App 页面，它一次性可能需要 3 部分数据，分别从 3 个 MobileAPI 接口获取，那么我们就可以做一个新的 MobileAPI 接口，将这 3 部分数据都获取到，然后一次性返回。

减少调用次数只是若干解决方案中的一种，更极端的做法是，App 调用一次 MobileAPI 接口后，在一个时间段内不再调用，仍然使用上次调用接口获取到的数据，这些数据保存在 App 上，我们称为 App 缓存，这个时间段我们称为 App 缓存时间。

App 缓存只能针对于 MobileAPI 中 GET 类型的接口，对于 POST 不适用。因为 GET 是获取数据，而 POST 是修改数据。

此外，即使是 GET 类型的接口，对于那些即时性很低的、不怎么改变的数据，比如获取商品的描述，缓存时间可以设置得比较长，比如 5 ~ 10 分钟，对于那些即时性比较高、频繁变动的数据，比如商品价格，缓存时间就会比较短，甚至不能进行缓存。

即使对于同一个需要做 App 缓存的 MobileAPI，参数不同，缓存也是不同的。比如 GetWeather.api 这个 MobileAPI 接口，它有一个参数也就是时间 date，对于 date=2014-9-8 和 date=2014-9-9，它们就分别对应两个缓存，不能存在一起。

接下来要说的是，App 缓存存在哪里，以及以什么方式进行存放。由于缓存数据比较大，所以我们将其存在 SD 卡上，而不是内存中。这样的话，App 缓存策略就仅限于那些有 SD 卡的手机用户了。

我们可以将 xxx.api?k1=va&k2=v2 这样的 URL 格式作为 key，存放 App 缓存数据。需要注意的是，我们要对 k1、k2 这些 key 进行排序，这样才能唯一，否则对于如下 URL：

```
xxxx.api?k1=v1&k2=v2
xxxx.api?k2=v2&k1=v1
```

就会被认为是两个不同的 key 存放在缓存中，但其实它们是一样的。

对上面的介绍总结如下：

1）对于 App 而言，它是感受不到取的是缓存数据还是调用 MobileAPI。具体工作由网络底层完成。

2）在 url.xml 中为每一个 MobileAPI 接口配置缓存时间 Expired。对于 post，一律设置为 0，因为 post 不需要缓存。

3）在 HttpRequest 类中的 run 方法中，改动 3 个地方：

a）写一个排序算法 sortKeys，对 URL 中的 key 进行排序。

b）将 newUrl 作为 key，检查缓存中是否有数据，有则直接返回；否则，继续调用 MobileAPI 接口。

```
// 如果这个 get 的 API 有缓存时间（大于 0）
if (urlData.getExpires() > 0) {
    final String content = CacheManager.getInstance()
            .getFileCache(newUrl);
    if (content != null) {
        handler.post(new Runnable() {
            @Override
            public void run() {
                requestCallback.onSuccess(content);
            }
        });

        return;
    }
}
```

c）MobileAPI 接口返回数据后，将数据存入缓存。

```
final Response responseInJson = JSON.parseObject(
        strResponse, Response.class);
if (responseInJson.hasError()) {
    handleNetworkError(responseInJson.getErrorMessage());
} else {
    // 把成功获取到的数据记录到缓存
    if (urlData.getNetType().equals(REQUEST_GET)
            && urlData.getExpires() > 0) {
        CacheManager.getInstance().putFileCache(newUrl,
                responseInJson.getResult(),
                urlData.getExpires());
    }

    handler.post(new Runnable() {
        @Override
        public void run() {
            requestCallback.onSuccess(responseInJson
                    .getResult());
        }
    });
}
```

4）CacheManager 用于操作读写缓存数据，并判断缓存数据是否过期。缓存中存放的实体就是 CacheItem。

5）在 App 项目中，创建 YoungHeartApplication 这个 Application 级别的类，在程序启动时，初始化缓存的目录，如果不存在则创建之。

```
public class YoungHeartApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
```

```
            CacheManager.getInstance().initCacheDir();
        }
    }
```

记得要在 AndroidManifest.xml 文件中，指定 application 的 android:name 为 YoungHeartApplication：

```
    <application
        android:allowBackup="true"
        android:name="com.youngheart.engine.YoungHeartApplication"
```

对缓存数据，我这里没有做加密，而是直接把明文以字符串类型存放到了 SD 卡。有这方面特殊需求的 App，可以将要缓存的数据转成 byte 数组再序列化到本地，要用的时候再反过来操作就是了。

2.2.2　强制更新

不光是 App 端需要记录缓存数据，在 MobileAPI 的很多接口，其实也需要一样的设计。

如果对于某个接口的数据，MobileAPI 缓存了 5 分钟，App 缓存了 3 分钟，那么最极端的情况是，用户在 8 分钟内是看不到数据更新的。因此，我们需要在页面上提供一个强制更新的按钮。

我们可以让 RemoteService 多暴露一个 boolean 类型的参数，用于判断是否要遵守 App 端缓存策略，如果是，则在从 url.xml 中取出 UrlData 实体后，将其 expired 强制设置为 0，这样就不会执行缓存策略了。

RemoteService 的改动如下：

```
    public void invoke(final BaseActivity activity,
            final String apiKey,
            final List<RequestParameter> params,
            final RequestCallback callBack) {
        invoke(activity, apiKey, params, callBack, false);
    }

    public void invoke(final BaseActivity activity,
            final String apiKey,
            final List<RequestParameter> params,
            final RequestCallback callBack,
            final boolean forceUpdate) {
        final URLData urlData =
                UrlConfigManager.findURL(activity, apiKey);
        if(forceUpdate) {
            // 如果强制更新，那么就把过期时间强制设置为 0
            urlData.setExpires(0);
        }

        HttpRequest request =
                activity.getRequestManager().createRequest(
```

```
                    urlData, params, callBack);
    DefaultThreadPool.getInstance().execute(request);
}
```

那么在调用的时候，只需要加一个参数就好：

```
RemoteService.getInstance().invoke(
        this, "getWeatherInfo", params,
        weatherCallback);
```

数据缓存是一把双刃剑，设置时间长了，数据长期不更新，用户体验就会不好。因此我们需要为那些强迫症类型的用户提供一个强制刷新的按钮，点击按钮后，页面会重新调用 MobileAPI 加载数据，无论缓存是否到期。

具体这个按钮放在什么位置，就需要产品经理来决策了。我见过很多 App 都是放在右上角。当然，这是见仁见智的事了。

2.3 MockService

在 App 团队与 MobileAPI 团队协同开发的过程中，经常会遇到因为 MobileAPI 接口还没好而 App 又急等着用的情况。

正常的流程是：

1）MobileAPI 开发人员会事先和 App 开发人员定义 MobileAPI 接口，包括 API 名称、参数、返回 JSON 格式。

2）MobileAPI 按照上述约定写一个 Mock 接口，部署到测试环境，该 Mock 接口中没有任何逻辑实现，在返回结果中硬编码返回一些 JSON 数据。我们假设这个工作应该是很快的。

3）App 开发人员基于上述测试环境 Mock 接口，进行开发。

4）MobileAPI 接口完成后，通知 App 开发人员，对真实逻辑进行联调。

以上 4 步，如果是正常实施，是没有问题的，但是问题经常出在第 2 步，MobileAPI 开发人员来不及提供 Mock 接口。

另一种情况是，随着 App 开发工作的进行，App 开发人员会发现原先约定的那些字段不够用，所以就会要求 MobileAPI 开发人员频繁修改 Mock 接口并部署到测试环境，这是一个很浪费时间的工作。

其实，就是因为 App 与 MobileAPI 之间有依赖，我们需要解除这种依赖。为此我们要在 App 端设计自己的 MockService，这样就在完成上述步骤 1——约定 MobileAPI 接口参数和返回 JSON 格式后，在 App 端 Mock 自己的数据，直到功能开发完成，而不会被任何人阻塞。App 开发完成后，肯定也积累了一些修改意见，这时候可以请 MobileAPI 开发人员汇总在一起进行修改。

设计 App 端 MockService 包括如下几个关键点：

1）对需要 Mock 数据的 MobileAPI 接口，通过在 url.xml 中配置 Node 节点 MockClass 属性，来指定要使用那个 Mock 子类生成的数据：

```
<Node
    Key="getWeatherInfo"
    Expires="300"
    NetType="get"
    MockClass="com.youngheart.mockdata.MockWeatherInfo"
    Url="http://www.weather.com.cn/data/sk/101010100.html" />
```

这里将使用 com.mockdata.mockdata 包下的 MockWeatherInfo 子类来解析。

2）我使用了反射工厂来设计 MockService。MockService 类是基类，它有一个抽象方法 getJsonData，用于返回手动生成的 Mock 数据。

```
public abstract class MockService {
    public abstract String getJsonData();
}
```

每个要 Mock 数据的 MobileAPI 接口，都对应一个继承自 MockService 的子类，都要实现各自的 getJsonData 方法，返回不同的 JSON 数据。

比如在上述 url.xml 中声明的 MockWeatherInfo，它对应的类实现如下：

```
public class MockWeatherInfo extends MockService {
    @Override
    public String getJsonData() {
        WeatherInfo weather = new WeatherInfo();
        weather.setCity("Beijing");
        weather.setCityid("10000");

        Response response = getSuccessResponse();
        response.setResult(JSON.toJSONString(weather));
        return JSON.toJSONString(response);
    }
}
```

以后每添加一个新的 Mock 类，我们都将其放置在 mockdata 包下，如图 2-2 所示。

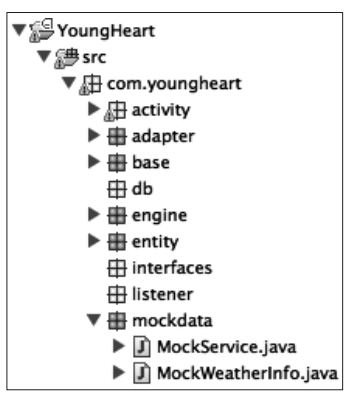


图 2-2　mockdata 包

3）接下来介绍如何实现反射机制。

主要的改造工作在 RemoteService 类的 invoke 方法中，根据是否在 url.xml 中指定了 MockClass 值来决定，是调用线上 MobileAPI 还是从本地 MockService 直接取假数据。

如果 MockClass 有值，就把这个值反射为一个具体的类，比如 MockWeatherInfo，然后调用它的 getJsonData 方法。

```
public void invoke(final BaseActivity activity,
```

```
            final String apiKey,
            final List<RequestParameter> params,
            final RequestCallback callBack) {
        final URLData urlData = UrlConfigManager.findURL(activity, apiKey);
        if (urlData.getMockClass() != null) {
            try {
                MockService mockService = (MockService) Class.forName(
                        urlData.getMockClass()).newInstance();
                String strResponse = mockService.getJsonData();
                final Response responseInJson =
                        JSON.parseObject(strResponse, Response.class);
                if (callBack != null) {
                    if (responseInJson.hasError()) {
                        callBack.onFail(responseInJson.getErrorMessage());
                    } else {
                        callBack.onSuccess(responseInJson.getResult());
                    }
                }
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            }
        } else {
            HttpRequest request =
                    activity.getRequestManager().createRequest(
                            urlData, params, callBack);
            DefaultThreadPool.getInstance().execute(request);
        }
    }
```

有了 MockService 这个利器，对于作者来说，本书接下来的内容将会轻松很多，因为不需要搭建自己的服务器，全都用 MockService 在本地编写假数据即可。

2.4 用户登录

登录是考查一个 App 开发人员是否合格的衡量标准。面试的时候我都会问候选人这个问题。

由于我手头没有任何 MobileAPI 登录接口，所以我准备用 2.3 节介绍的 MockService 来模拟登录的返回信息。

为此建立 MockLoginSuccessInfo 类如下：

```
public class MockLoginSuccessInfo extends MockService {
    @Override
    public String getJsonData() {
        UserInfo userInfo = new UserInfo();
```

```
        userInfo.setLoginName("jianqiang.bao");
        userInfo.setUserName("包建强");
        userInfo.setScore(100);

        Response response = getSuccessResponse();
        response.setResult(JSON.toJSONString(userInfo));
        return JSON.toJSONString(response);
    }
}
```

并在 url.xml 中如下配置 MockClass：

```
<Node
    Key="getWeatherInfo"
    Expires="300"
    NetType="get"
    MockClass="com.youngheart.mockdata.MockLoginSuccessInfo"
    Url="http://www.weather.com.cn/data/sk/101010100.html" />
```

2.4.1　登录成功后的各种场景

首先，贯穿 App 的，应该有一个 User 全局变量，在每次登录成功后，会将其 isLogin 属性设置为 true，在退出登录后，则将该属性设置为 false。这个 User 全局变量要支持序列化到本地的功能，这样数据才不会因内存回收而丢失。

其次，登录分为 3 种情形：

情形 1：点击登录按钮，进入登录页面 LoginActivity，登录成功后，直接进入个人中心 PersonCenterActivity。这种情况最直截了当，一路执行 startActivity(intent) 就能达到目的。

情形 2：在页面 A，想要跳转到页面 B，并携带一些参数，却发现没有登录，于是先跳转到登录页，登录成功后，再跳转到 B 页面，同时仍然带着那些参数。

这就主要是 setResult(intent, resultCode) 发挥作用的时候了，Activity 的回调机制这时候派上了用场，如下所示：

```
btnLogin2.setOnClickListener(new OnClickListener(){
    @Override
    public void onClick(View v) {
        if(User.getInstance().isLogin()) {
            gotoNewsActivity();
        } else {
            Intent intent = new Intent(LoginMainActivity.this,
                    LoginActivity.class);
            intent.putExtra(AppConstants.NeedCallback, true);
            startActivityForResult(intent,
                        LOGIN_REDIRECT_OUTSIDE);
        }
    }
});
```

情形 3：在页面 A，执行某个操作，却发现没有登录，于是跳转到登录页，登录成功后，

再回到页面 A，继续执行该操作。

处理方式同于情形 2，也是使用 setResult 来完成回调。

```
btnLogin3.setOnClickListener(new OnClickListener(){
    @Override
    public void onClick(View v) {
        if(User.getInstance().isLogin()) {
            changeText();
        } else {
            Intent intent = new Intent(LoginMainActivity.this,
                    LoginActivity.class);
            intent.putExtra(AppConstants.NeedCallback, true);
            startActivityForResult(intent,
                        LOGIN_REDIRECT_INSIDE);
        }
    }
});
```

无论是上述哪种情形，登录页面 LoginActivity 只有一个，所以要把上面的三个逻辑整合在一起，如下所示：

```
RequestCallback loginCallback = new AbstractRequestCallback() {

    @Override
    public void onSuccess(String content) {
        UserInfo userInfo = JSON.parseObject(content,
                UserInfo.class);
        if (userInfo != null) {
            User.getInstance().reset();
            User.getInstance().setLoginName(userInfo.getLoginName());
            User.getInstance().setScore(userInfo.getScore());
            User.getInstance().setUserName(userInfo.getUserName());
            User.getInstance().setLoginStatus(true);
            User.getInstance().save();
        }

        if(needCallback) {
            setResult(Activity.RESULT_OK);
            finish();
        } else {
            Intent intent = new Intent(LoginActivity.this,
                    PersonCenterActivity.class);
            startActivity(intent);
        }
    }
};
```

整合的关键在于从上个页面传过来 needCallback 变量，它决定了是否要回到上个页面。

另一方面，我们看到，在登录成功后，我们会把用户信息存储到 User 这个全局变量并序列化到本地，这是因为各个模块都有可能使用到用户的信息。其中 LoginStatus 是关键，接

下来的篇幅将着重谈论这个属性。

最后在 LoginMainActivity 中的 onActivityResult 回调函数，它负责处理登录后的事情，如下所示：

```
@Override
protected void onActivityResult(int requestCode,
        int resultCode, Intent data) {
    if (resultCode != Activity.RESULT_OK) {
        return;
    }

    switch (requestCode) {
    case LOGIN_REDIRECT_OUTSIDE:
        gotoNewsActivity();
        break;
    case LOGIN_REDIRECT_INSIDE:
        changeText();
        break;
    default:
        break;
    }
}
```

我们看到，对于情形 2，当用户在 LoginMainActivity 点击按钮想跳转到 NewsActivity，如果已经登录，就直接跳转过去；否则，先到 LoginActivity 登录，然后回调 LoginMainActivity 的 onActivityResult，仍然跳转到 NewsActivity。

2.4.2　自动登录

所谓自动登录，就是登录成功后，重启 App 后用户仍然是登录状态。

最直接的方法是，登录成功后，本地保存用户名和密码。重启 App 后，检查本地是否有保存用户名和密码，如果有，则将用户名和密码传入到登录接口，模拟用户登录的行为。

但这样就有安全风险了，分析如下：

- 本地保存用户密码，这种的敏感信息容易被人窃取。要么是在本地文件中看到这些信息，要么是侦听 App 的网络请求，获取到请求的数据。

  所以本地保存密码时，一定要进行加密。对称加密是不可靠的，因为很难确保 App 的源代码不外泄，所以别有用心的人还是可以根据源码中的对称加密算法，反向把密码推算出来。只有不对称加密才是安全的。

- 那么登录之后呢？市面上大多数 App 的逻辑都有问题，它们会在本地保存一个 isLogin 的全局变量，登录成功后设置为 true。接下来涉及用户相关的 MobileAPI，只有在这个值为 true 时才能调用，它们会把 UserId 传递给服务器。

  服务器的解决方案通常也很简陋，它没有任何安全机制，包括用户信息相关的 MobileAPI 接口，只要接口调用参数中有 UserId，它就会去把相关的数据取出并返回。

一种补救措施是，每次调用用户相关的 MobileAPI 接口时，都需要把 UserId 和加密后的密码一起传递。而服务器需要对那些用户相关的 MobileAPI 接口加上安全验证机制，每次请求都检查用户名和密码是否正确。我们要求密码是经过哈希散列算法不对称加密过的，是无法还原的。服务器的验证工作是根据传过来的 UserId 从数据库中取出相应的密码，然后进行比对。注意，数据库中存放的密码是在注册的时候经过哈希散列算法加密过的。

- 本地保存用户名和密码的另一个问题是，每次用户启动 App，登录页都会一闪而过，因为它要模拟用户登录的行为：假装输入用户名和密码，然后假装点击登录按钮。这样做用户体验很不好倒是其次，关键是这种做法有个无法自圆其说的硬伤——出于安全考虑，我们要修改登录接口，使其除了接收用户名和密码这两个参数外，还必须接收验证码，也就是动态口令，如图 2-3 所示。

图 2-3 App 的登录界面

我们知道，验证码必须是手动输入的，否则就失去了它存在的意义。但是当前这种自动登录的做法，我们只知道用户名和密码，而不知道每次生成的验证码是什么，所以就不能自动登录了。是时候该抛弃这种每次启动就进行一次登录的机制了，其实 Web 在这一点已经做得很成熟了，那就是 Cookie 机制。

也有的人管 Cookie 叫 Token，这是用户身份的唯一性标志。

首先，App 在登录成功后，会从服务器获取到一个 Cookie，这个 Cookie 存放在 HttpResponse 的 header 中，如下所示：

```
Set-Cookie: customer=huangxp; path=/foo; domain=.ibm.com;
expires= Wednesday, 19-OCT-05 23:12:40 GMT; [secure]
```

我们将其取出来，不用关心它是什么，只要把它存放在本地文件中即可。

我们需要修改 App 的网络底层，也就是 HttpRequest 类，分以下几步：

1）每次发起 MobileAPI 请求时，都要把本地保存的 Cookie 取出来，放到 HttpRequest 的 header 中。还是那句话，不用管 Cookie 是什么，也不管 Cookie 是否有值，都应如下操作：

```
// 添加 Cookie 到请求头中
addCookie();

// 发送请求
response = httpClient.execute(request);
```

2）每次接收 MobileAPI 的相应结果时，都把 HttpResponse 的 header 里面的 Cookie 取出来，覆盖本地保存的 Cookie。不用管 Cookie 有值与否，如下所示：

```
if (urlData.getNetType().equals(REQUEST_GET)
        && urlData.getExpires() > 0) {
```

```
    CacheManager.getInstance().putFileCache(newUrl,
            responseInJson.getResult(),
            urlData.getExpires());
}

handler.post(new Runnable() {
    @Override
    public void run() {
        requestCallback.onSuccess(responseInJson
                .getResult());
    }
});

// 保存 Cookie
saveCookie();
```

以下是 addCookie 和 saveCookie 方法的实现：

```
public void addCookie() {
    List<SerializableCookie> cookieList = null;
    Object cookieObj = BaseUtils.restoreObject(cookiePath);
    if (cookieObj != null) {
        cookieList = (ArrayList<SerializableCookie>) cookieObj;
    }

    if ((cookieList != null) && (cookieList.size() > 0)) {
        final BasicCookieStore cs = new BasicCookieStore();
        cs.addCookies(cookieList.toArray(new Cookie[] {}));
        httpClient.setCookieStore(cs);
    } else {
        httpClient.setCookieStore(null);
    }
}
public synchronized void saveCookie() {
    // 获取本次访问的 cookie
    final List<Cookie> cookies =
            httpClient.getCookieStore().getCookies();
    // 将普通 cookie 转换为可序列化的 cookie
    List<SerializableCookie> serializableCookies = null;

    if ((cookies != null) && (cookies.size() > 0)) {
        serializableCookies = new ArrayList<SerializableCookie>();

        for (final Cookie c : cookies) {
            serializableCookies.add(new SerializableCookie(c));
        }
    }

    BaseUtils.saveObject(cookiePath, serializableCookies);
}
```

而服务器的相应操作，对于来自 App 的请求：

3）如果是用户信息相关的，则判断 HttpRequest 中 Cookie 是否有效，如果有效，就去

执行后续的逻辑并返回结果；否则，返回 Cookie 过期失效的错误信息。

4）如果是用户无关的，则不需要检查 HttpRequest 中 Cookie，直接执行下面的逻辑即可。此外，还需要注意几个地方，都是些琐碎的工作：

- 用户注销功能，要把本地保存的 Cookie 清空。App 判断用户是否登录的标志，就是 Cookie 是否为空。
- 用户注册功能，一般在注册成功后，都会拿着用户名和密码再调用一次登录接口，这就又和验证码功能冲突了，解决方案是注册成功后直接跳转到登录页面，让用户手动再输入一次。这是从产品层面来解决问题。另一种解决方案是，注册成功后进入个人中心页面，不需要再登录一次，而是把注册和登录接口绑在一起。
- 对于 Cookie 过期，App 应该跳转到登录页面，让用户手动进行登录。这里有一个比较有挑战性的工作，就是登录成功后，应该返回手动登录之前的那个页面。我们在下一节再细说这个技术。[⊖]

2.4.3 Cookie 过期的统一处理

Cookie 不是一直有效的，到了一定时间就会失效。

Cookie 过期的表现是，当访问 MobileAPI 某个接口的时候，就不会返回数据了，代之以 Cookie 过期的错误消息，这时要统一处理。

我们要求 MobileAPI 在遇到这种情况时，直接返回以下内容的 JSON，其中，errorType 固定为 1：

```
{
    "isError" : true,
    "errorType" : 1,
    "errorMessage" : "Cookie 失效，请重新登录 ",
    "result" : ""
}
```

为此我们修改 AndroidLib，使之支持 Cookie 失效的场景。

1）在 RequestCallback 中增加一种 onCookieExpired 回调方法，如下所示：

```
public interface RequestCallback
{
    public void onSuccess(String content);

    public void onFail(String errorMessage);

    public void onCookieExpired();
}
```

2）在网络底层对 JSON 返回结果进行解析，如果发现是属于 Cookie 过期的错误类型，

⊖ 关于 Cookie 更详细的介绍，请参考博客园小坦克的这篇文章：http://blog.csdn.net/ToCpp/article/details/4680946。

就直接回调 onCookieExpired 方法，如下所示：

```
final Response responseInJson = JSON.parseObject(
        strResponse, Response.class);
if (responseInJson.hasError()) {
    if(responseInJson.getErrorType() == 1) {
        handler.post(new Runnable() {
            @Override
            public void run() {
                requestCallback.onCookieExpired();
            }
        });
    } else {
        handleNetworkError(responseInJson.getErrorMessage());
    }
```

我们模拟一种场景，在 CookieExpiredActivity 页面，访问天气预报这个 MobileAPI 接口，如果 Cookie 失效，则弹出对话框，通知用户“ Cookie 过期，请重新登录”，点击确定按钮，将跳转到登录页。登录成功后，将回到上一个页面，即 CookieExpiredActivity。

由于所有页面处理 Cookie 过期的逻辑都是相同的，所以我们将其封装到基类 AppBaseActivity 中，放在和 onFail 方法平级的位置：

```
public void onCookieExpired() {
    dlg.dismiss();

    new AlertDialog.Builder(AppBaseActivity.this)
        .setTitle(" 出错啦 ")
        .setMessage("Cookie 过期，请重新登录 ")
        .setPositiveButton(" 确定 ",
            new DialogInterface.OnClickListener() {

                @Override
                public void onClick(DialogInterface dialog,
                        int which) {
                    Intent intent = new Intent(
                        AppBaseActivity.this,
                        LoginActivity.class);
                    intent.putExtra(AppConstants.NeedCallback,
                            true);
                    startActivity(intent);
                }
            }).show();
}
```

其实就这么简单，因为我们事先对网络底层进行了高度的封装，所以增加一种新的回调类型并不麻烦。

2.4.4　防止黑客刷库

经常有一些网站的安全措施没做好，导致用户名和密码大量泄漏。城门失火，殃及池

鱼。要知道，对于用户而言，他们通常在各个网站上都使用相同的用户名和密码，这些信息在一个网站上泄漏了，会导致在其他网站上的信息也都失去了保障。于是，某些黑客就会写一个脚本，遍历他窃取到的某个网站成千上万的用户名和密码，去访问另一个网站的登录接口。1 万个用户还试不出来 1 个用户吗？

一种安全解决方案是为登录接口增加第三个参数，也就是上文提及的验证码。每次登录都必须输入验证码，其实就是为了防止被黑客刷库。

但是这个方案仅适用于那些新 App，一开始就要如此设计，我们要杜绝只有用户名和密码的登录接口，这是有安全隐患的。

有的网站是连续登录 3 次失败后，才要求用户输入验证码。难道他们不怕被黑客刷库吗？其实还有其他的解决方案，但同时需要 MobileAPI 和 App 配合工作：

- MobileAPI 在发现有同一 IP 短时间内频繁访问某一个 MobileAPI 接口时，就直接返回一段 HTML5，要求用户输入验证码。
- App 在接收到这段代码时，就在页面上显示一个浮层，里面一个 WebView，显示这个要求用户输入验证码的 HTML5。

这样就阻止了黑客刷库。试问哪个呆萌黑客愿意每隔一两分钟就手动输入一次验证码呢？

2.5 HTTP 头中的奥妙

对于 HTTP 头，我们并不陌生。我们在上一节中成功运用到了 HTTP 头中的 Cookie 属性。接下来，我们将继续发挥它的威力，看看它还能为我们做些什么。

我们先学习一下 HTTP 请求的定义。

2.5.1 HTTP 请求

HTTP 请求分为 HTTPRequest 和 HTTPResponse 两种。但无论哪种请求，都由 header 和 body 两部分组成。

1. HTTP Body

Body 部分就是存放数据的地方，回顾一下我们在 HTTPRequest 类中封装的网络请求：

1）对于 get 形式的 HTTPRequest，要发送的数据都以键值对的形式存放在 URL 上，比如 aaa.api?k1=va&k2=va。它的 Body 是空的，如下所示：

```
if (urlData.getNetType().equals(REQUEST_GET)) {
    //添加参数
    final StringBuffer paramBuffer = new StringBuffer();
    if ((parameter != null) && (parameter.size() > 0)) {

        //这里要对 key 进行排序
        sortKeys();

        for (final RequestParameter p : parameter) {
```

```
            if (paramBuffer.length() == 0) {
                paramBuffer.append(p.getName() + "="
                        + BaseUtils.UrlEncodeUnicode(p.getValue()));
            } else {
                paramBuffer.append("&" + p.getName() + "="
                        + BaseUtils.UrlEncodeUnicode(p.getValue()));
            }
        }

        newUrl = url + "?" + paramBuffer.toString();
    } else {
        newUrl = url;
    }

    request = new HttpGet(newUrl);
}
```

2）对于 post 形式的 HTTPRequest，要发送的数据都存在 Body 里面，也是以键值对的形式，所以代码编写与 get 情形完全不同，如下所示：

```
else if (urlData.getNetType().equals(REQUEST_POST)) {
    request = new HttpPost(url);
    //添加参数
    if ((parameter != null) && (parameter.size() > 0)) {
        final List<BasicNameValuePair> list =
                new ArrayList<BasicNameValuePair>();
        for (final RequestParameter p : parameter) {
            list.add(new BasicNameValuePair(
                    p.getName(), p.getValue()));
        }

        ((HttpPost) request).setEntity(
                new UrlEncodedFormEntity(list, HTTP.UTF_8));
    }
}
```

2. HTTP Header

与 Body 相比，HTTP header 就丰富的多了。它由很多键值对（key-value）组成，其中有些 key 是标准的，兼容于各大浏览器，比如：

- ❑ accept
- ❑ accept-language
- ❑ referrer
- ❑ user-agent
- ❑ accept-encoding

此外，我们还可以在 MobileAPI 端自定义一些键值对，然后要求 App 在调用 MobileAPI 时把这些信息传递过来。比如 MobileAPI 可以定义一个 check-value 这样的 key，然后要求 App 将 AppId（同一公司的不同 App 编号）、ClientType（Android 还是 iPhone、iPad）这些值

拼接在一起经过 MD5 加密后，作为这个 key 的值传递给 MobileAPI，然后由 MobileAPI 再去分析这些数据。

对于 App 开发人员而言，只要按照 MobileAPI 的要求，把这些 key 所需要的值拼接成 HTTPRequest 头正确传递过去即可。如下所示：

```
void setHttpHeaders(final HttpUriRequest httpMessage)
{
    headers.clear();
    headers.put(FrameConstants.ACCEPT_CHARSET, "UTF-8,*");
    headers.put(FrameConstants.USER_AGENT,
                                "Young Heart Android App ");

    if ((httpMessage != null) && (headers != null))
    {
        for (final Entry<String, String> entry : headers.entrySet())
        {
            if (entry.getKey()!=null)
            {
                httpMessage.addHeader(entry.getKey(), entry.getValue());
            }
        }
    }
}
```

我们在组装 Cookie 之前调用 setHttpHeaders 方法：

```
// 添加必要的头信息
setHttpHeaders(request);

// 添加 Cookie 到请求头中
addCookie();

// 发送请求
response = httpClient.execute(request);
```

而在返回数据时，也可以从 HTTP Response 头中把所需要的数据解析出来。Android SDK 将其封装成了若干方法以供调用。我们在下面的章节将会看到。

前面我们介绍过 Cookie，其实也是 HTTP 头的一部分。它的作用我们已经见识过了。下面将讨论 HTTP 头中的另几个重要字段。

2.5.2 时间校准

接下来要介绍的是 HTTP Response 头中另一重要属性：Date，这个属性中记录了 MobileAPI 当时的服务器时间。

为什么说这个属性很重要呢？ App 开发人员经常遇到的一个 bug 就是，App 显示的时间不准，经常会因为时区问题前后差几个小时，而接到用户的投诉。

为了解决这个问题，要从 MobileAPI 和 App 同时做一些工作。MobileAPI 永远使用 UTC

时间。包括入参和返回值，都不要使用 Date 格式，而是减去 UTC 时间 1970 年 1 月 1 日的差值，这是一个 long 类型的长整数。

在 App 端比较麻烦。这里我们只讨论中国，比如国内航班时间、电影上映时间等等，那么我们把 MobileAPI 返回的 long 型时间转换为 GMT8 时区的时间就万事大吉了——只需要额外加 8 个小时。无论使用的人身在哪个时区，他们看到的都应该是一个时间，也就是 GMT8 的时间。

由于 App 本地时间会不准，比如前后差十几分钟，又比如设置了 GMT9 的时区，这样在取本地时间的时候，就会差一个小时。遇到这种情况，就要依赖于 HTTP Response 头的 Date 属性了。

每调用一次 MobileAPI，就取出 HTTP Response 头的 Date 值，转换为 GMT 时间后，再减去本地取出的时间，得到一个差值 delta。这个值可能是因为手机时间不准而差出来的那十几分钟，也可能是因为时区不同导致的 1 个小时差值。我们将这个 delta 值保存下来。那么每当取本地当前时间的时候，再额外加上这个 delta 差值，就得到了服务器 GMT8 的时间，就做到了任何人看到的时间是一样的。

因为 App 会频繁调用 MobileAPI，所以这个 delta 值也会频繁更新，不用担心长期不调用 MobileAPI 而导致的这个 delta 值不太准的问题。

接下来我们修改 AndroidLib 框架，以支持上述的这些功能。

1）首先，在 HTTPRequest 类提供一个用于更新本地时间和服务器时间差值的方法 updateDeltaBetweenServerAndClientTime，如下所示，由于我们在这里补上了 UTC 和 GMT8 相差的那 8 个小时，所以 App 其他地方不再需要考虑时差的问题，如下所示：

```
void updateDeltaBetweenServerAndClientTime() {
    if (response != null) {
        final Header header = response.getLastHeader("Date");
        if (header != null) {
            final String strServerDate = header.getValue();
            try {
                if ((strServerDate != null) && !strServerDate.equals("")) {
                    final SimpleDateFormat sdf = new SimpleDateFormat(
                            "EEE, d MMM yyyy HH:mm:ss z", Locale.ENGLISH);
                    TimeZone.setDefault(TimeZone.getTimeZone("GMT+8"));

                    Date serverDateUAT = sdf.parse(strServerDate);

                    deltaBetweenServerAndClientTime = serverDateUAT
                            .getTime()
                            + 8 * 60 * 60 * 1000
                            - System.currentTimeMillis();
                }
            } catch (java.text.ParseException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}
```

我们会在发起 MobileAPI 网络请求得到响应结果后，执行该方法，更新这个差值：

```
// 发送请求
response = httpClient.execute(request);
// 获取状态
final int statusCode = response.getStatusLine().getStatusCode();
// 设置回调函数，但如果 requestCallback，说明不需要回调，不需要知道返回结果
if ((requestCallback != null)) {
    if (statusCode == HttpStatus.SC_OK) {
        // 更新服务器时间和本地时间的差值
        updateDeltaBetweenServerAndClientTime();
```

因为我们的 App 会频繁的调用 MobileAPI，所以为了避免频繁读写文件，我们没有将 deltaBetweenServerAndClientTime 存到本地文件，而是放在了内存中，当作一个全局变量来使用。

2）我们把这个 deltaBetweenServerAndClientTime 方法暴露出来，供外界调用：

```
public static Date getServerTime() {
    return new Date(System.currentTimeMillis()
            + deltaBetweenServerAndClientTime);
}
```

现在我们就可以模拟一个场景了。我把手机的时间改成任意一个值，然后再进入到 WeatherByFastJsonActivity 页面，因为页面加载的时候会调用 MobileAPI 获取天气的接口，所以本地会保存一个 deltaBetweenServerAndClientTime 差值。点击 WeatherByFastJsonActivity 页面上的“获取服务器时间”按钮，会因为我调用了 AndroidLib 中封装好的 getServerTime 方法，而弹出 GMT8 的当前时间：

```
btnShowTime.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        String strCurrentTime = Utils.getServerTime().toString();

        new AlertDialog.Builder(WeatherByFastJsonActivity.this)
                .setTitle("当前时间是: ").setMessage(strCurrentTime)
                .setPositiveButton("确定", null).show();
    }
});
```

我见过一个 App 中的 ntp.hosts 文件，里面罗列了若干亚洲区的时间校准服务器，比如说：cn.pool.ntp.org。[㊀]

我猜这是为了解决手机系统时间与服务器时间不同步的问题，身处不同时区是一种情况，另一种情况则是用户故意把手机时间提前五分钟，以防止赶不上班车。但不管怎样，那种通

㊀ 关于 NTP，请参见 http://www.cnblogs.com/TianFang/archive/2011/12/20/2294603.html。

过 App 强行修改手机系统时间的做法是不负责任的。

对于手机系统时间不准的问题，本文给出了比较好的解决方案，即通过每次调用 MobileAPI 来计算时间差，然后每次本地获取时间就加上这个时间差。

对于用户身处不同时区的问题，App 仍然返回同一个时间，只是要在 App 上注明这些时间都是北京时间，而不能是北京用户显示飞机 9 点起飞而日本用户显示 10 点起飞。另一方面，这两个时区的用户在一起聊天是个麻烦的事情，即使有人在日本时区 10 点说句话，对于北京用户而言，看到的也应该是 9 点发的消息，反之亦然。而服务器则要使用格林威治一套时间，具体怎么显示，那是 App 的事情。有些 App 就存在这样的 bug，出国旅游收不到即时聊天消息，到了晚上会莫名其妙冒出来几百条消息，就是因为这个时区问题没有处理好导致的。

2.5.3　开启 gzip 压缩

接下来要介绍的内容和 gzip 有关。HTTP 协议上的 gzip 编码是一种用来改进 Web 应用程序性能的技术。大流量的 Web 站点常常使用 gzip 压缩技术来减少传输量的大小，减少传输量大小有两个明显的好处，一是可以减少存储空间，二是通过网络传输时，可以减少传输的时间。

使用 gzip 的流程如下：

1）在 App 发起请求时，在 HTTPRequest 头中，添加要求支持 gzip 的 key-value，这里的 key 是 Accept-Encoding，value 是 gzip。如下所示，我们需要修改 setHttpHeaders 方法：

```
void setHttpHeaders(final HttpUriRequest httpMessage) {
    headers.clear();
    headers.put(FrameConstants.ACCEPT_CHARSET, "UTF-8,*");
    headers.put(FrameConstants.USER_AGENT, "Young Heart Android App ");
    headers.put(FrameConstants.ACCEPT_ENCODING, "gzip");

    if ((httpMessage != null) && (headers != null)) {
        for (final Entry<String, String> entry : headers.entrySet()) {
            if (entry.getKey() != null) {
                httpMessage.addHeader(entry.getKey(), entry.getValue());
            }
        }
    }
}
```

2）MobileAPI 的逻辑是，检查 HTTP 请求头中的 Accept-Encoding 是否有 gzip 值，如果有，就会执行 gzip 压缩。

如果执行了 gzip 压缩，那么在返回值也就是 HTTPResponse 的头中，有一个 content-encoding 字段，会带有 gzip 的值；否则，就没有这个值。

3）App 检查 HTTPResponse 头中的 content-encoding 字段是否包含 gzip 值，这个值的有无，导致了 App 解析 HTTPResponse 的姿势不同，如下所示（以下代码参见 HTTPRequest 这个类）：

```
String strResponse = "";
if ((response.getEntity().getContentEncoding() != null)
        && (response.getEntity().getContentEncoding()
                .getValue() != null)) {
    if (response.getEntity().getContentEncoding()
            .getValue().contains("gzip")) {
        final InputStream in = response.getEntity()
                .getContent();
        final InputStream is = new GZIPInputStream(in);
        strResponse = HttpRequest.inputStreamToString(is);
        is.close();
    } else {
        response.getEntity().writeTo(content);
        strResponse = new String(content.toByteArray()).trim();
    }
} else {
    response.getEntity().writeTo(content);
    strResponse = new String(content.toByteArray()).trim();
}
```

到此，一个比较完备的网络底层封装就全部完成了。

2.6 本章小结

本章介绍如何对网络底层进行封装，其中包括：新写了一个网络调用框架用以代替 AsyncTask；设计了 App 的缓存机制；设计了 MockService 的机制，以后即使没有 MobileAPI 接口也能开发新功能了。介绍用户 Cookie 的设计方法；巧妙运用 Http 头中的数据等。

下一章，我将介绍 App 中的一些经典场景的设计。

第 3 章 Chapter 3

Android 经典场景设计

同样是使用 Java 语言，为什么做 MobileAPI 的开发人员写不了 Android 程序，反之亦然。我想大概是各行有各行的规矩和做事法则，本章介绍的这几种 Android 经典场景就是如此，看似都是些平谈无奇的 UI，但其中却蕴藏着大智慧。

闲话少叙，且听我一一道来。

3.1 App 图片缓存设计

App 缓存分为两部分，数据缓存和图片缓存。

我们在第 2 章的 2.2 节介绍了 App 数据缓存，从而把从 MobileAPI 获取到的数据缓存到本地，减少了调用 MobileAPI 的次数。

本节将介绍图片缓存策略。

3.1.1 ImageLoader 设计原理

Android 上最让人头疼的莫过于从网络获取图片、显示、回收，任何一个环节有问题都可能直接 OOM。尤其是在列表页，会加载大量网络上的图片，每当快速划动列表的时候，都会很卡，甚至会因为内存溢出而崩溃。

这时就轮到 ImageLoader 上场表演了。ImageLoader 的目的是为了实现异步的网络图片加载、缓存及显示，支持多线程异步加载。㊀

ImageLoader 的工作原理是这样的：在显示图片的时候，它会先在内存中查找；如果没

㊀ ImageLoader 在 GitHub 的下载地址：https://github.com/nostra13/Android-Universal-Image-Loader。

有，就去本地查找；如果还没有，就开一个新的线程去下载这张图片，下载成功会把图片同时缓存到内存和本地。

基于这个原理，我们可以在每次退出一个页面的时候，把 ImageLoader 内存中的缓存全都清除，这样就节省了大量内存，反正下次再用到的时候从本地再取出来就是了。

此外，由于 ImageLoader 对图片是软引用的形式，所以内存中的图片会在内存不足的时候被系统回收（内存足够的时候不会对其进行垃圾回收）。[㊀]

3.1.2 ImageLoader 的使用

ImageLoader 由三大组件组成：

- ImageLoaderConfiguration——对图片缓存进行总体配置，包括内存缓存的大小、本地缓存的大小和位置、日志、下载策略（FIFO 还是 LIFO）等等。
- ImageLoader——我们一般使用 displayImage 来把 URL 对应的图片显示在 ImageView 上。
- DisplayImageOptions——在每个页面需要显示图片的地方，控制如何显示的细节，比如指定下载时的默认图（包括下载中、下载失败、URL 为空等），是否将缓存放到内存或者本地磁盘。

借用博客园上陈哈哈的博文[㊁]对三者关系的一个比喻，“他们有点像厨房规定、厨师、客户个人口味之间的关系。ImageLoaderConfiguration 就像是厨房里面的规定，每一个厨师要怎么着装，要怎么保持厨房的干净，这是针对每一个厨师都适用的规定，而且不允许个性化改变。ImageLoader 就像是具体做菜的厨师，负责具体菜谱的制作。DisplayImageOptions 就像每个客户的偏好，根据客户是重口味还是清淡，每一个 ImageLoader 根据 DisplayImageOptions 的要求具体执行。”

下面我们介绍如何使用 ImageView：

1）在 YoungHeartApplication 中总体配置 ImageLoader：

```
public class YoungHeartApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();

        CacheManager.getInstance().initCacheDir();

        ImageLoaderConfiguration config =
                new ImageLoaderConfiguration.Builder
                (getApplicationContext())
                .threadPriority(Thread.NORM_PRIORITY - 2)
                .memoryCacheExtraOptions(480, 480)
```

㊀ 关于 Java 强引用、软引用、弱引用、虚引用的介绍，请参考这篇文章：http://www.cnblogs.com/blogoflee/archive/2012/03/22/2411124.html。

㊁ 详细内容请参见 http://www.cnblogs.com/kissazi2/p/3886563.html。

```
                .memoryCacheSize(2 * 1024 * 1024)
                .denyCacheImageMultipleSizesInMemory()
                .discCacheFileNameGenerator(new Md5FileNameGenerator())
                .tasksProcessingOrder(QueueProcessingType.LIFO)
                .memoryCache(new WeakMemoryCache()).build();

        ImageLoader.getInstance().init(config);
    }
}
```

2）在使用 ImageView 加载图片的地方，配置当前页面的 ImageLoader 选项。有可能是 Activity，也有可能是 Adapter：

```
public CinemaAdapter(ArrayList<CinemaBean> cinemaList,
        AppBaseActivity context) {
    this.cinemaList = cinemaList;
    this.context = context;

    options = new DisplayImageOptions.Builder()
            .showStubImage(R.drawable.ic_launcher)
            .showImageForEmptyUri(R.drawable.ic_launcher)
            .cacheInMemory()
            .cacheOnDisc()
            .build();
}
```

3）在使用 ImageView 加载图片的地方，使用 ImageLoader，代码片段节选自 CinemaAdapter：

```
CinemaBean cinema = cinemaList.get(position);
holder.tvCinemaName.setText(cinema.getCinemaName());
holder.tvCinemaId.setText(cinema.getCinemaId());

context.imageLoader.displayImage(cinemaList.get(position)
        .getCinemaPhotoUrl(), holder.imgPhoto);
```

其中 displayImage 方法的第一个参数是图片的 URL，第二个参数是 ImageView 控件。

一般来说，ImageLoader 性能如果有问题，就和这里的配置有关，尤其是 ImageLoader-Configuration。我列举在上面的配置代码是目前比较通用的，请大家参考。

3.1.3　ImageLoader 优化

尽管 ImageLoader 很强大，但一直把图片缓存在内存中，会导致内存占用过高。虽然对图片的引用是软引用，软引用在内存不够的时候会被 GC，但我们还是希望减少 GC 的次数，所以要经常手动清理 ImageLoader 中的缓存。

我们在 AppBaseActivity 中的 onDestroy 方法中，执行 ImageLoader 的 clearMemoryCache 方法，以确保页面销毁时，把为了显示这个页面而增加的内存缓存清除。这样，即使到了下个页面要复用之前加载过的图片，虽然内存中没有了，根据 ImageLoader 的缓存策略，还是可以在本地磁盘上找到：

```
public abstract class AppBaseActivity extends BaseActivity {
    protected boolean needCallback;

    protected ProgressDialog dlg;

    public ImageLoader imageLoader = ImageLoader.getInstance();

    protected void onDestroy() {
        // 回收该页面缓存在内存的图片
        imageLoader.clearMemoryCache();

        super.onDestroy();
    }
```

本章没有过多讨论 ImageLoader 的代码实现，只是描述了它的实现原理。有兴趣的朋友可以参考下列文章，里面有很深入的研究：

1）简介 ImageLoader。

地址：http://blog.csdn.net/yueqinglkong/article/details/27660107

2）Android-Universal-Image-Loader 图片异步加载类库的使用（超详细配置）。

地址：http://blog.csdn.net/vipzjyno1/article/details/23206387

3）Android 开源框架 Universal-Image-Loader 完全解析。

地址：http://blog.csdn.net/xiaanming/article/details/39057201

3.1.4 图片加载利器 Fresco

就在本书写作期间，Facebook 开源了它的 Android 图片加载组件 Fresco。

我之所以关注这个 Fresco 组件，是因为我负责的 App 用一段时间后就占据了 180M 左右的内存，App 会变得很卡。我们使用 MAT 分析内存，发现让内存居高不下的罪魁祸首就是图片。于是我们把目光转向 Fresco，开始优化 App 占用的内存。

Fresco 使用起来很简单，如下所示：

- 在 Application 级别，对 Fresco 进行初始化，如下所示：

```
Fresco.initialize(getApplicationContext());
```

- 与 ImageLoader 等传统第三方图片处理 SDK 不同，Fresco 是基于控件级别的，所以我们把程序中显示网络图片的 ImageView 都替换为 SimpleDraweeView 即可，并在 ImageView 所在的布局文件中添加 fresco 命名空间，如下所示：

```
<LinearLayout
    xmlns:android="http:// schemas.android.com/apk/res/android"
    xmlns:fresco="http:// schemas.android.com/apk/res-auto">
<com.facebook.drawee.view.SimpleDraweeView
    android:id="@+id/imgView"
    android:layout_width="10dp"
    android:layout_height="10dp"
```

```
    fresco:placeholderImage="@drawable/placeholder" />
```

❑ 在 Activity 中为这个图片控件指定要显示的网络图片：

```
Uri uri = Uri.parse("http://www.bb.com/a.png");
draweeView.setImageURI(uri);
```

Fresco 的原理是，设计了一个 Image Pipeline 的概念，它负责先后检查内存、磁盘文件（Disk），如果都没有再老老实实从网络下载图片，如图 3-1 所示，箭头上标记了 jpg 或 bmp 格式的，表示 Cache 中有图片，直接取出；没有标记，则表示 Cache 中找不到。

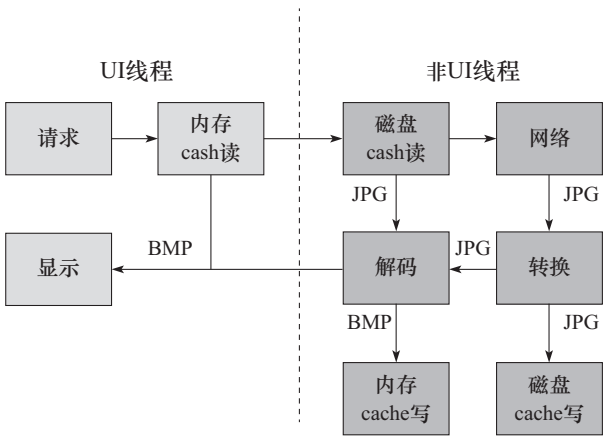


图 3-1　Image Pipeline 的工作流

我们可以像配置 ImageLoader 那样配置 Fresco 中的 Image Pipeline，使用 ImagePipelineConfig 来做这个事情。

Fresco 有 3 个线程池，其中 3 个线程用于网络下载图片，2 个线程用于磁盘文件的读写，还有 2 个线程用于 CPU 相关操作，比如图片解码、转换，以及放在后台执行的一些费时操作。

接下来介绍 Fresco 三层缓存的概念。这才是 Fresco 最核心的技术，它比其他图片 SDK 吃内存小，就在于这个全新的缓存设计。

第一层：Bitmap 缓存

❑ 在 Android 5.0 系统中，考虑到内存管理有了很大改进，所以 Bitmap 缓存位于 Java 的堆（heap）中。

❑ 而在 Android 4.x 和更低的系统，Bitmap 缓存位于 ashmem 中，而不是位于 Java 的堆（heap）中。这意味着图片的创建和回收不会引发过多的 GC，从而让 App 运行得更快。当 App 切换到后台时，Bitmap 缓存会被清空。

第二层：内存缓存

内存缓存中存储了图片的原始压缩格式。从内存缓存中取出的图片，在显示前必须先解码。当 App 切换到后台时，内存缓存也会被清空。

第三层：磁盘缓存

磁盘缓存，又名本地存储。磁盘缓存中存储的也是图片的原始压缩格式。在使用前也要先解码。当 App 切换到后台时，磁盘缓存不会丢失，即使关机也不会。

Fresco 有很多高级的应用，对于大部分 App 而言，基本还用不到。只要掌握上述简单的使用方法就能极大地节省内存了。我做的 App 原先占用 180MB 的内存，现在只会占据 80MB 左右的内存了。这也是我为什么要在本书中增加这一部分内容的原因。

关于 Fresco 的更多介绍请参见：

- Fresco 在 GitHub 上的源码：https://github.com/mkottman/AndroLua
- Fresco 官方文档：http://fresco-cn.org/docs/index.html

3.2 对网络流量进行优化

对 App 的最低容忍限度是，在 2G、3G 和 4G 网络环境下，每个页面都能打开，都能正常跳转到其他页面。要能够完成一次完整的支付流程。

慢点儿没关系，尤其是 2G 网络。但是动不动就弹出“无法连接到网络”或者“网络连接超时”的对话框，就是我们开发人员必须要解决的问题了。

3.2.1 通信层面的优化

让我们先从 MobileAPI 层面进行优化：

1）MobileAPI 接口返回的数据，要使用 gzip 进行压缩。注意：大于 1KB 才进行压缩，否则得不偿失。经过 gzip 压缩后，返回的数据量大幅减少。

2）App 与 MobileAPI 之间的数据传递，通常是遵守 JSON 协议的。JSON 因为是 xml 格式的，并且是以字符存在的，在数据量上还有可以压缩的空间。我这里推荐一种新的数据传输协议，那就是 ProtoBuffer。这种协议是二进制格式的，所以在表示大数据时，空间比 JSON 小很多。

3）接下来要解决的是频繁调用 MobileAPI 的问题。我们知道，发起一次网络请求，服务器处理的速度是很快的，主要花费的时间在数据传输上，也就是这一来一回走路的时间上。

走路时间的长度，网络运维人员会去负责解决。移动开发人员需要关注的是，减少网络访问次数，能调用一次 MobileAPI 接口就能取到数据的，就不要调用两次。

4）我们知道，传统的 MobileAPI 使用的是 HTTP 无状态短连接。使用 HTTP 协议的速度远不如使用 TCP 协议，因为后者是长连接。所以我们可以使用 TCP 长连接，以提高访问的速度。缺点是一台服务器能支持的长连接个数不多，所以需要更多的服务器集成。

5）要建立取消网络请求的机制。一个页面如果没有请求完网络数据，在跳转到另一个页面之前，要把之前的网络请求都取消，不再等待，也不再接收数据。

我遇到过一个真实的例子，首页要在后台调用十几个 MobileAPI 接口，用户一旦进入二级页面，在二级页面获取列表数据时，经常会取不到数据，并弹出“网络请求超时”的提示。

我们通过在 App 输出 log 的方式发现，二级页面还在调用首页没有完成的那些 MobileAPI 接口，App 网络底层的请求队列已经被阻塞了，原因是在进入下一个页面时，首页发起的网络请求仍然存在于网络请求队列中，并没有移除掉。

无论是 iOS 还是 Android，都应该在基类（BaseViewController 或者 BaseActivity）中提供一个 cancelRequest 的方法，用以在离开当前页面时清空网络请求队列。

6）增加重试机制。如果 MobileAPI 是严格的 RESTful 风格，那么我们一般将获取数据的请求接口都定义为 get；而把操作数据的请求接口都定义为 post。

这样的话，我们就可以为所有的 get 请求配置重试机制，比如 get 请求失败后重试 3 次。

有人会问 post 请求失败后，是否需要重试呢？我们举个例子吧，比如说下单接口是个 post 请求，如果请求失败那么就会重试 3 次，直到下单成功。但是有时候 post 请求并没有失败，而是超时了，超时时间是 30 秒，但是却 31 秒返回了，如果因此而重新发起下单请求，那么就会连续下单两次。所以 post 请求是不建议有重试机制的。此外，对所有的 post 请求，都要增加防止用户 1 分钟内频繁发起相同请求的机制，这样就能有效防止重复下单、重复发表评论、重复注册等操作。

如果 post 请求具有防重机制，那么倒是可以增加重试机制。但是要可以在服务器端灵活配置重试的次数，可以是 0 次，意味着不会重试。在 App 启动的时候，告诉 App 所有的 MobileAPI 接口的重试次数。

3.2.2　图片策略优化

首先，我们从图片层面进行优化，这里说的图片，是根据 MobileAPI 返回的图片 URL 地址新启一个线程下载到 App 本地并显示的。很多 App 崩溃的原因就是图片的问题没处理好。

以下是我遇到的几类问题以及相应的解决方案。

1. 要确保下载的每张图，都符合 ImageView 控件的大小

这对于 Android 是有难度的，因为手机分辨率千奇百怪，所以 App 中的图片，我们大多做成自适应的，有时是等比拉伸或缩放图片的宽和高，有时则固定高度而动态伸缩宽度，反之亦然。

于是我们要求运营人员要事先准备很多套不同分辨率的图片。我们每次根据 URL 请求图片时，都要额外在 URL 上加上两个参数，width 和 height，从而要求服务器返回其中某一张图，URL 如下所示：http://www.aaa.com/a.png?width=100&height=50。

如果认为每次准备很多套图片是件很浪费人力的事情，我还有另一种解决方案，这种方案只需要一张图。但我们需要事先准备一台服务器，称为 ImageServer。具体流程是这样的：

1）首先，App 每次加载图片，都会把 URL 地址以及 width 和 height 参数所组成的字符串进行 encode，然后发送给 ImageServer，新的 URL 如下所示：

http://www. ImageServer.com/getImage?param=(encode value)

2）然后，ImageServer 收到这个请求，会把 param 的值 decode，得到原始图片的 URL，

以及 App 想要显示的这张图片的 width 和 height。ImageServer 会根据 URL 获取到这张原始图片，然后根据 width 和 height，重新进行绘制，保存到 ImageServer 上，并返回给 App。

3）最后，App 请求到的是一张符合其显示大小的图片。

接下来收到同样的请求，直接返回 ImageServer 上保存的那种图片即可。但是要每天清一次硬盘，不然过不了几天硬盘就满了。

如果 width 和 height 的比例与原图的宽高比不一致呢？我们需要再加一个参数 imagetype，以下是定义：

- 1 表示等比缩放后，裁减掉多余的宽或者高。
- 2 表示等比缩放后，不足的宽或者高填充白色。

当然你也可以定义 0 表示不进行缩放，直接返回。

这种方案的缺点就是，ImageServer 频繁地写硬盘，硬盘坚持不到两周就坏掉。所以，我们在损失了几块硬盘后，决定事先规定几套 width 和 height，App 必须严格遵守，比如说 100 × 50，200 × 100，那么就不允许向服务器发送类似 99 × 51 这样的图片尺寸。

但这样规定，并不能防止 App 开发人员犯错，他在 UI 上就是不小心为某个 ImageView 控件指定了 99 × 51 这样的尺寸，那么 ImageServer 还是会生成这样的图片。

唯一的办法就是在出口加以控制，也就是向 ImageServer 发起请求的时候。我们会拿 99 × 51 这个实际的图片尺寸，去轮询我们事先规定好的那几个尺寸 100 × 50 和 200 × 100，看更接近哪个，比如说 99 × 51 更接近 100 × 50，那么就向 ImageServer 请求 100 × 50 这种尺寸的图片。

找最接近图片尺寸的办法是面积法：

```
S = (w1-a) × (w1-w) + (h1-h) × (h1-h)
```

w 和 h 是实际的图片宽和高，w1 和 h1 是事先规定的某个尺寸的宽和高。S 最小的那个，就是最接近的。

2. 低流量模式

在 2G 和 3G 网络环境下，我们应该适当降低图片的质量。降低图片质量，相应的图片大小也会降低，我们称为低流量模式。

还记得我们前面提到的 ImageServer 吗？我们可以在 URL 中再增加一个参数 quality，2G 网络下这个值为 50%，3G 网络下这个值为 70%，我们把这个参数传递给 ImageServer，从而 ImageServer 在绘制图片时，就会将 jpg 图片质量降低为 50% 或 70%，这样返回给 App 的数据量就大大减少了。

在列表页，这种效果最为明显，能极大的节省用户流量。

3. 极速模式

我们后来发现，在 2G 和 3G 网络环境下，用户大多对图片不感兴趣，他们可能就是想快速下单并支付，我们需要额外设计一些页面，区别于正常模式下图文并茂的页面，我们将

这些只有文字的页面称为极速模式。

比如，首页往往图片占据多数，而且这些图片大多数从网络动态下载的，在 2G 网络下，这些图片是很浪费流量的。所以在极速模式下，我们需要设计一个只有纯文字的首页。

在每次开启 App 进入首页前会先进行预判，如果发现当前网络环境为 2G、3G 或 4G，但是当前模式为正常模式，就会弹出一个对话框询问用户，是否要进入极速模式以节省流量。如果是 WiFi 网络环境，但当前模式是极速模式，也会提示用户是否要切换回正常模式，以看到最炫的效果。

仅在开启 App 时提示用户极速模式是不够的，我们在设置页也要提供这个开关，供用户手动切换。

3.3　城市列表的设计

很多 App 都有城市列表这一功能。看似简单，但就像登录功能一样，做好它并不容易。

3.3.1　城市列表数据

一份城市列表的数据包括以下几个字典：

- cityId：城市 Id。
- cityName：城市名称。
- pinyin：城市全拼。
- jianpin：城市简拼。

其中，全拼和简拼是用来在 App 本地做字母表排序和关键字检索的。

我曾经经历过把城市列表数据写死在本地文件的做法，日积月累，就会产生两个问题：

- Android 和 iOS 维护的数据，差异会越来越大。
- 一千多个城市，每次从本地加载都要很长时间。

针对问题 1 的解决办法是，写一个文本分析工具，找出 Android 和 iOS 各自维护文件的不同数据。

iOS 开发人员喜欢使用 plist 文件作为数据存储的载体，最好能和 Android 统一使用一份 xml 文件，这样便于管理类似城市列表这样的数据。

针对问题 2 的解决方案是，对于一千多个城市，意味着每次都要解析 xml 城市数据文件，既然每次读取数据都很慢，那么我们干脆就把序列化过的城市列表直接保存到本地文件，跟随 App 一起发布。这样，每次读取这个文件时，就直接进行反序列化即可，速度得到很大提升。

把城市列表数据保存在本地，有个很烦的事情，就是每次增加新的城市，都要等下次发版，因为数据是写死在 App 本地的。于是，我们把城市列表数据做成一个 MobileAPI 接口，由 MobileAPI 去后台采集数据，这样数据是最新最准的。

但是这样做的问题是，这个 MobileAPI 接口返回的数据量会很大，上千笔数据，还包括那么多字段，即使打开了 gzip 压缩，也会有 100k 的样子。于是我们又增加了版本号字段 version 的概念，这个 MobileAPI 接口的定义和返回的 JSON 格式是这样的：

1）入参。version，本地存储的城市列表数据对应的版本号。

2）返回值。如果传入参数 version 和线上最新版本号一致，则返回以下固定格式：

```
{
    "isMatch": false,
    "version": 1,
    "cities": [
        {
        },
    ]
}
```

如果传入参数 version 和线上最新版本号不一致，则返回以下格式：

```
{
    "isMatch": false,
    "version": 1,
    "cities": [
        {
            "cityId": 1,
            "cityName": "北京",
            "pinyin": "beijing",
            "jianpin": "bj"
        },
        {
            "cityId": 2,
            "cityName": "上海",
            "pinyin": "shanghai",
            "jianpin": "sh"
        },
        {
            "cityId": 3,
            "cityName": "平顶山",
            "pinyin": "pingdingshan",
            "jianpin": "pds"
        }
    ]
}
```

version 这个字段由 MobileAPI 进行更新，每当有城市数据更新时，version 可以立即自增 +1，也可以积累到一定数据后自增 +1。具体策略由 MobileAPI 来决定。

基于此，App 的策略可以是这样的：

1）本地仍然保存一份线上最新的城市列表数据（序列化后的）以及对应的版本号。我们要求每次发版前做一次城市数据同步的事情。

2）每次进入到城市列表这个页面时，将本地城市列表数据对应的版本号 version 传入到 MobileAPI 接口，根据返回的 isMatch 值来决定是否版本号一致。如果一致，则直接从本地文件中加载城市列表数据；否则，就解析 MobileAPI 接口返回的数据，在显示列表的同时，记得要把最新的城市列表数据和版本号保存到本地。

3）如果 MobileAPI 接口没有调用成功，也是直接从本地文件中加载城市列表数据，以确保主流程是畅通的。

4）每次调用 MobileAPI 时，会获取到大量的数据，一般我们会打开 gzip 对数据进行压缩，以确保传输的数据量最小。

3.3.2　城市列表数据的增量更新机制

上节中我们谈到，每当有城市数据更新时，version 可以立即自增 +1。我的问题是，如何判断有城市数据更新？一种解决方案是，在服务器建立一个 Timer，每十分钟跑一次，检查 10 分钟前后的数据是否有改动，如果有，version 就自增 +1，并返回这些有改动的数据（新增、删除和修改）。这样就保证了 10 分钟内，从 A 改成 B 又改回 A，这时候我们认为是没有改动的，版本号不需要自增 +1。

那么问题来了，对于 1000 笔城市数据，每次只改动其中的几笔，返回数据中包括那些没有改动过的数据是没有意义的，是否可以只返回这些改动的数据？

分析 1.0 和 2.0 版本的城市列表数据，每笔数据都有 cityId 和其他一些字段，比如说城市名称、简拼、全拼等。我画了一个表，如图 3-2 所示，试图展示出 1.0 和 2.0 这两个版本的城市数据之间的异同。

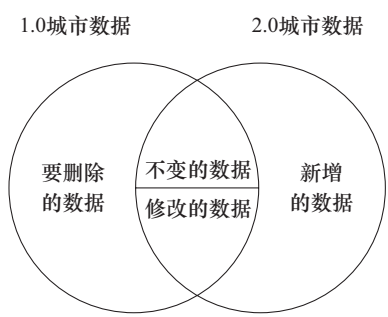


图 3-2　比较两个版本城市数据间的异同

我来解释一下图 3-2，以 cityId 作为唯一标识，只在 1.0 中出现的 cityId 是要删除的数据，只在 2.0 中出现的 cityId 是要增加的数据，二者的交集则是 cityId 相同的数据，这又分为两种情况，所有字段都相同的数据是不变的数据；cityId 相同但某个字段不相同，则是修改的数据。

增量更新的数据，就由增、删、改这 3 部分数据构成。

于是，我们可以重新定义城市列表的 JSON 格式，在每笔增量数据中增加一个字段 type，用来区别是增（c）、删（d）、改（u）中的哪种情况，如下所示：

```
{
    "isMatch": false,
    "version": 1,
    "cities": [
        {
            "cityId": 1,
            "cityName": "北京",
            "pinyin": "beijing",
```

```
            "jianpin": "bj",
            "type": "d"
        },
        {
            "cityId": 2,
            "cityName": "上海",
            "pinyin": "shanghai",
            "jianpin": "sh",
            "type": "c"
        },
        {
            "cityId": 3,
            "cityName": "平顶山",
            "pinyin": "pingdingshan",
            "jianpin": "pds",
            "type": "u"
        }
    ]
}
```

客户端在收到上述格式 JSON 数据后，会根据 type 值来处理存放在本地的数据。因为不是全量更新，所以处理起来很快。

这种增量更新城市数据的策略，会使得 App 的逻辑很简单，但是服务器的逻辑很复杂。这样做是划算的，我们要想尽办法确保 App 的轻量，把复杂的业务逻辑放在后端。

3.4 App 与 HTML5 的交互

App 与 HTML5 的交互，是一个可以大做文章的话题。有的团队直接使用 PhoneGap 来实现交互的功能，而我则认为 PhoneGap 太重了。我们完全可以把这些交互操作在底层封装好，然后给开发人员使用。

为了开发人员方便，我们要准备一台测试用的 PC 服务器，在上面搭建一个 IIS，这样可以快速搭建自己的 Demo，对于 App 开发人员而言，不需要等待 HTML5 团队就可以自行开发并测试了。他们只需知道一些基本的 Html 和 JavaScript 语法，而相应的培训非常简单。

3.4.1 App 操作 HTML5 页面的方法

为了演示方便，我在 assets 中内置了一个 HTML5 页面。现实中，这个 HTML5 页面是放在远程服务器上的。

首先要定好通信协议，也就是 App 要调用的 HTML5 页面中 JavaScript 的方法名称。

例如，App 要调用 HTML5 页面的 changeColor(color) 方法，改变 HTML5 页面的背景颜色。

1）HTML5

```
<script type="text/javascript">
    function changeColor (color) {
```

```
        document.body.style.backgroundColor = color;
    }
</script>
```

2）Android

```
wvAds.getSettings().setJavaScriptEnabled(true);
    wvAds.loadUrl("file:// /android_asset/104.html");

    btnShowAlert.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            String color = "#00ee00";
            wvAds.loadUrl("javascript: changeColor ('" + color + "');");
        }
    });
```

3.4.2　HTML5 页面操作 App 页面的方法

仍然是先定义通信协议，这次定义的是 JavaScript 要调用的 Android 中方法名称。

例如，点击 HTML5 的文字，回调 Java 中的 callAndroidMethod 方法：

1）HTML5

```
<a onclick="baobao.callAndroidMethod(100,100,'ccc',true)">
    CallAndroidMethod</a>
```

2）Android

新创建一个 JSInterface1 类，包括 callAndroidMethod 方法的实现：

```
class JSInteface1 {
    public void callAndroidMethod(int a, float b,
                String c, boolean d) {
        if (d) {
            String strMessage = "-" + (a + 1) + "-" + (b + 1)
                    + "-" + c + "-" + d;

            new AlertDialog.Builder(MainActivity.this)
                    .setTitle("title")
                    .setMessage(strMessage).show();
        }
    }
}
```

同时，需要注册 baobao 和 JSInterface1 的对应关系：

```
wvAds.addJavascriptInterface(new JSInteface1(), "baobao");
```

调试期间我发现对于小米 3 系统，要在方法前增加 @JavascriptInterface，否则，就不能触发 JavaScript 方法。

3.4.3 App 和 HTML5 之间定义跳转协议

根据上面的例子，运营团队就找到了在 App 中搞活动的解决方案。不必等到 App 每次发新版才能看到新的活动页面，而是每次做一个 HTML5 的活动页面，然后通过 MobileAPI 把这个 HTML5 页面的地址告诉 App，由 App 加载这个 HTML5 页面即可。

在这个 HTML5 页面中，我们可以定义各种 JavaScript 点击事件，从而跳转回 App 的任意 Native 页面。

为此，HTML5 团队需要事先和 App 团队约定好一个格式，例如：

```
gotoPersonCenter
gotoMovieDetail:movieId=100
gotoNewsList:cityId=1&cityName= 北京
gotoUrl:http:// www.sina.com
```

这个协议具体在 HTML5 页面中是这样的，以 gotoNewsList 为例：

```
<a onclick="baobao.gotoAnyWhere(
        'gotoNewsList:cityId=(int)12&cityName= 北京 ')">
            gotoAnyWhere</a>
```

其中，有些协议是不需要参数的，比如说 gotoPersonCenter，也就是个人中心；有些则需要跳转到具体的电影详情页，我们需要知道 movieId ；有时候 1 个参数不够用，我们需要更多的参数，才能准确获取到我们想要的数据，比如说 gotoNewsList，我们想要跳转到 2014 年 12 月 31 号北京的所有新闻信息，就不得不需要 cityId 和 createdTime 两个参数，处理协议的代码如下所示：

```
public void gotoAnyWhere(String url) {
    if (url != null) {
        if (url.startsWith("gotoMovieDetail:")) {
            String strMovieId = url.substring(24);
            int movieId = Integer.valueOf(strMovieId);

            Intent intent = new Intent(MainActivity.this, MovieDetailActivity.class);
            intent.putExtra("movieId", movieId);
            startActivity(intent);
        } else if (url.startsWith("gotoNewsList:")) {
            // as above
        } else if (url.startsWith("gotoPersonCenter")) {
            Intent intent = new Intent(MainActivity.this, PersonCenterActivity.class);
            startActivity(intent);
        } else if (url.startsWith("gotoUrl:")) {
            String strUrl = url.substring(8);
            wvAds.loadUrl(strUrl);
        }
    }
}
```

这里的 if 分支逻辑太多，我们要想办法将其进行抽象，参见后面 3.4.6 节介绍的页面分发器。

3.4.4　在 App 中内置 HTML5 页面

什么时候在 App 中内置 HTML5 页面？根据我的经验，当有些 UI 不太容易在 App 中使用原生语言实现时，比如画一个奇形怪状的表格，这是 HTML5 所擅长的领域，只要调整好屏幕适配，就可以很好地应用在 App 中。

下面详细介绍如何在页面中显示一个表格，表格里的数据都是动态填充的。

1）首先定义两个 HTML5 文件，放在 assets 目录下。

其中，102.html 是静态页：

```
<html>
    <head>
    </head>
    <body>
        <table>
            <data1DefinedByBaobao>
        </table>
    </body>
</html>
```

而 data1_template.html 是一个数据模板，它负责提供表格中一行的样式：

```
<tr>
    <td>
        <name>
    </td>
    <td>
        <price>
    </td>
</tr>
```

像 <name>、<price> 和 <data1DefinedByBaobao> 都是占位符，我们接下来会使用真实的数据来替换这些占位符。

2）在 MovieDetailActivity 中，通过遍历 movieList 这个集合，我们把数据填充到 sbContent 中，最终，把拼接好的字符串替换 <data1DefinedByBaobao> 标签：

```
String template = getFromAssets("data1_template.html");
StringBuilder sbContent = new StringBuilder();

ArrayList<MovieInfo> movieList = organizeMovieList();
for (MovieInfo movie : movieList) {
    String rowData;
    rowData = template.replace("<name>", movie.getName());
    rowData = rowData.replace("<price>", movie.getPrice());
    sbContent.append(rowData);
}

String realData = getFromAssets("102.html");
realData = realData.replace("<data1DefinedByBaobao>",
        sbContent.toString());

wvAds.loadData(realData, "text/html", "utf-8");
```

3.4.5 灵活切换 Native 和 HTML5 页面的策略

对于经常需要改动的页面，我们会把它做成 HTML5 页面，在 App 中以 WebView 的形式加载。这样就避免了 Native 页面每次修改，都要等一次迭代上线后才能看到——周期太长了，这不是产品经理所希望的。

此外，HTML5 的另一个好处是，开发周期短——相比 App 开发而言。

但是 HTML5 的缺点是慢。我们来看一下 HTML5 页面生成的步骤：

1）从服务器端动态获取数据并拼接成一个 HTML。

2）返回给客户端 WebView。

3）在 WebView 中解析并生成这个 HTML。

相对于 Native 原生页面加载 JSON 这种短小精悍的数据并展现在客户端而言，HTML5 肯定是慢了很多。鱼和熊掌不可兼得，于是我们只能在灵活性和性能上作出取舍。

但是我们可以换一个思路来解决这个问题。我同时做两套页面，Native 一套，HTML5 一套，然后在 App 中设置一个变量，来判断该页面将显示 Native 还是 HTML5 的。

这个变量可以从 MobileAPI 获取，这样的话，正常情况下，是 Native 页面，如果有类似双十一或双十二的促销活动，我们可以修改这个变量，让页面以 HTML5 的形式展现。这样，我们只要做个 HTML5 的页面发布到线上就行了。等活动结束后再撤回到 Native 页面。

以此类推，App 中所有的页面，都可以做成上述这种形式，为此，我们需要改变之前做 App 的思路，比如：

1）需要做一个后台，根据版本进行配置每个页面是使用 Native 页面还是 HTML5 页面。

2）在 App 启动的时候，从 MobileAPI 获取到每个页面是 Native 还是 HTML5。

3）在 App 的代码层面，页面之间要实现松耦合。为此，我们要设计一个导航器 Navigator，由它来控制该跳转到 Native 页面还是 HTML5 页面。最大的挑战是页面间参数传递，字典是一种比较好的形式，消除了不同页面对参数类型的不同要求。

接下来，就是 App 运营人员和产品经理随心所欲的进行配置了。

在实际的操作中，一定要注意，HTML5 页面只是权宜之计，可以快速上一个活动，比如类似于双十一的节假日，从而以迅雷不及掩耳之势打击竞争对手。随着 HTML5 和 Native 的不同步，当一个页面再从 HTML5 切换回 Native 时，我们会发现，它们的逻辑已经差了很多了，切回来就会有很多 bug，而我们又只能是在 App 发布后才发现这样的问题。

唯一的解决方案是，把 App 和 HTML5 划归到一个团队，由产品经理整理二者的差异性，要做到二者尽量同步，一言以蔽之，App 要时刻追赶 HTML5 的逻辑，追赶上了就切换回 Native。

3.4.6 页面分发器

我们知道，跳转到一个 Activity，需要传递一些参数。这些参数的类型简单如 int 和

String，复杂的则是列表数据或者可序列化的自定义实体。

但是，如果从 HTML5 页面跳转到 Native 页面，是不大可能传递复杂类型的实体的，只能传递简单类型。所以，并不是每个 Native 页面都可以替换为 HTML5。

接下来要讨论的是，对于那些来自 HTML5 页面、传递简单类型的页面跳转请求，我们将其抽象为一个分发器，放到 BaseActivity 中。

还记得我们在 3.4.3 节定义的协议吗，以 gotoMovieDetail 为例：

```
<a onclick="baobao.gotoAnyWhere(
        'gotoMovieDetail:movieId=12')">
            gotoAnyWhere</a>
```

我们将其改写为：

```
<a onclick="baobao.gotoAnyWhere(
        'com.example.youngheart.MovieDetailActivity,
        iOS.MovieDetailViewController:movieId=(int)123')">
            gotoAnyWhere</a>
```

我们看到，协议的内容分成 3 段，第一段是 Android 要跳转到的 Activity 的名称。第二段是 iOS 要跳转到的 ViewController 的名称，第三段是需要传递的参数，以 key-value 的形式进行组装。

我们接下来要做的就是从协议 URL 中取出第 1 段，将其反射为一个 Activity 对象，取出第 3 段，将其解析为 key-value 的形式，然后从当前页面跳转到目标页面并配以正确的参数。其中，写一个辅助函数 getAndroidPageName，用来获取 Activity 名称：

```
public class BaseActivity extends Activity {
    private String getAndroidPageName(String key) {
        String pageName = null;

        int pos = key.indexOf(",");
        if (pos == -1) {
            pageName = key;
        } else {
            pageName = key.substring(0, pos);
        }

        return pageName;
    }

    public void gotoAnyWhere(String url) {
        if (url == null)
            return;

        String pageName = getAndroidPageName(url);
        if (pageName == null || pageName.trim() == "")
            return;

        Intent intent = new Intent();
```

```
            int pos = url.indexOf(":");
            if (pos > 0) {
                String strParams = url.substring(pos);
                String[] pairs = strParams.split("&");
                for (String strKeyAndValue : pairs) {
                    String[] arr = strKeyAndValue.split("=");
                    String key = arr[0];
                    String value = arr[1];
                    if (value.startsWith("(int)")) {
                        intent.putExtra(key,
                            Integer.valueOf(value.substring(5)));
                    } else if (value.startsWith("(Double)")) {
                        intent.putExtra(key,
                            Double.valueOf(value.substring(8)));
                    } else {
                        intent.putExtra(key, value);
                    }
                }
            }

            try {
                intent.setClass(this, Class.forName(pageName));
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            startActivity(intent);
        }
    }
```

注意，在协议中定义这些简单数据类型的时候，String 是不需要指定类型的，这是使用最广泛的类型。对于 int、Double 等简单类型，我们要在值前面加上类似 (int) 这样的约定，这样才能在解析时不出问题。

3.5 消灭全局变量

本节我们要讨论的是一个深刻的话题。相信很多人都遇到过 App 莫名其妙就崩溃的情况，尤其是一些配置很低的手机，重现场景就是在 App 切换到后台，闲置了一段时间后再继续使用时，就会崩溃。

3.5.1 问题的发现

导致上述崩溃发生的罪魁祸首就是全局变量。下述代码就是在生成一个全局变量：

```
public class GlobalVariables {
    public static UserBean User;
}
```

在内存不足的时候，系统会回收一部分闲置的资源，由于 App 被切换到后台，所以之前存放的全局变量很容易被回收，这时再切换到前台继续使用，在使用某个全局变量的时候，就会因为全局变量的值为空而崩溃。这不是个例。我经历过最糟糕的 App 竟然使用了 200 多个全局变量，任何页面从后台切换回前台都有崩溃的可能。

想彻底解决这个问题，就一定要使用序列化技术。

3.5.2　把数据作为 Intent 的参数传递

想一劳永逸地解决上述问题就是不使用全局变量，使用 Intent 来进行页面间数据的传递。因为，即使目标 Activity 被系统销毁了，Intent 上的数据仍然存在，所以 Intent 是保存数据的一个很好的地方，比本地文件靠谱。但是 Intent 能传递的数据类型也必须支持序列化，像 JSONObject 这样的数据类型，是传递不过去的。对于一个有 200 多个全局变量的 App 而言，重构的工作量很大，风险也很大。

另外，如果 Intent 上携带的数据量过大，也会发生崩溃。第 7 章会对此有详细的介绍。

3.5.3　把全局变量序列化到本地

另一个比较稳妥的解决方案是，我们仍然使用全局变量，在每次修改全局变量的值的时候，都要把值序列化到本地文件中，这样的话，即使内存中的全局变量被回收，本地还保存有最新的值，当我们再次使用全局变量时，就从本地文件中再反序列化到内存中。

这样就解了燃眉之急，数据不再丢失。但长远之计还是要一个模块一个模块地将全局变量转换为 Intent 上可序列化的实体数据。但这是后话，眼前，我们先要把全局变量序列化到本地文件，如下所示，我们对全局 GlobalsVariables 变量进行改造：

```
public class GlobalVariables implements Serializable, Cloneable {
    /**
     * @Fields: serialVersionUID
     */
    private static final long serialVersionUID = 1L;

    private static GlobalVariables instance;

    private GlobalVariables() {

    }

    public static GlobalVariables getInstance() {
        if (instance == null) {
            Object object = Utils.restoreObject(
                            AppConstants.CACHEDIR + TAG);
            if(object == null) {    //App 首次启动，文件不存在则新建之
                object = new GlobalVariables();
                Utils.saveObject(
```

```
                        AppConstants.CACHEDIR + TAG, object);
            }

            instance = (GlobalVariables)object;
        }

        return instance;
    }

    public final static String TAG = "GlobalVariables";

    private UserBean user;

    public UserBean getUser() {
        return user;
    }

    public void setUser(UserBean user) {
        this.user = user;
        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
    }

    //— — — — —以下 3 个方法用于序列化— — — — — — — —
    public GlobalVariables readResolve()
            throws ObjectStreamException,
            CloneNotSupportedException {
        instance = (GlobalVariables) this.clone();
        return instance;
    }

    private void readObject(ObjectInputStream ois)
            throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
    }

    public Object Clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public void reset() {
        user = null;

        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
    }
}
```

就是这短短的六十多行代码，解决了全局变量 GlobalsVariables 被回收的问题。我们对其进行详细分析：

1）首先，这是一个单例，我们只能以如下方式来读写 user 数据：

```
UserBean user = GlobalsVariables.getInstance().getUser();
GlobalsVariables.getInstance().setUser(user);
```

同时，GlobalsVariables 还必须实现 Serializable 接口，以支持序列化自身到本地。然而，为了使一个单例类变成可序列化的，仅仅在声明中添加“implements Serializable”是不够的。因为一个序列化的对象在每次反序列化的时候，都会创建一个新的对象，而不仅仅是一个对原有对象的引用。为了防止这种情况，需要在单例类中加入 readResolve 方法和 readObject 方法，并实现 Cloneable 接口。

2）我们仔细看 GlobalsVariables 这个类的构造函数。这和一般的单例模式写的不太一样。我们的逻辑是，先判断 instance 是否为空，不为空，证明全局变量没有被回收，可以继续使用；为空，要么是第一次启动 App，本地文件都不存在，更不要说序列化到本地了；要么是全局变量被回收了，于是我们需要从本地文件中将其还原回来。

为此，我们在 Utils 类中编写了 restoreObject 和 saveObject 两个方法，分别用于把全局变量序列化到本地和从本地文件反序列化到内存，如下所示：

```
public static final void saveObject(String path, Object saveObject) {
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;
    File f = new File(path);
    try {
        fos = new FileOutputStream(f);
        oos = new ObjectOutputStream(fos);
        oos.writeObject(saveObject);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (oos != null) {
                oos.close();
            }
            if (fos != null) {
                fos.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public static final Object restoreObject(String path) {
    FileInputStream fis = null;
    ObjectInputStream ois = null;
    Object object = null;
    File f = new File(path);
    if (!f.exists()) {
        return null;
    }
    try {
        fis = new FileInputStream(f);
        ois = new ObjectInputStream(fis);
```

```
            object = ois.readObject();
            return object;
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                if (ois != null) {
                    ois.close();
                }
                if (fis != null) {
                    fis.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return object;
    }
```

3）全局变量的 User 属性，具有 getUser 和 SetUser 这两个方法。我们就看这个 setUser 方法，它会在每次设置一个新值后，执行一次 Utils 类的 saveObject 方法，把新数据序列化到本地。

值得注意的是，如果全局变量中有一个自定义实体的属性，那么我们也要将这个自定义实体也声明为可序列化的，UserBean 实体就是一个很好的例子。它作为全局变量的一个属性，其自身也必须实现 Serializable 接口。

接下来我们看如何使用全局变量。

1）在来源页：

```
    private void gotoLoginActivity() {
        UserBean user = new UserBean();
        user.setUserName("Jianqiang");
        user.setCountry("Beijing");
        user.setAge(32);

        Intent intent = new Intent(LoginNew2Activity.this,
                    PersonCenterActivity.class);

        GlobalVariables.getInstance().setUser(user);

        startActivity(intent);
    }
```

2）在目标页 PersonCenterActivity：

```
    protected void initVariables() {
```

```
    UserBean user = GlobalVariables.getInstance().getUser();
    int age = user.getAge();
}
```

3）在 App 启动的时候，我们要清空存储在本地文件的全局变量，因为这些全局变量的生命周期都应该伴随着 App 的关闭而消亡，但是我们来不及在 App 关闭的时候做，所以只好在 App 启动的时候第一件事情就是清除这些临时数据：

```
GlobalVariables.getInstance().reset();
```

为此，需要在 GlobalVariables 这个全局变量类中增加一个 reset 方法，用于清空数据后把空值强制保存到本地。

```
public void reset() {
    user = null;
    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}
```

3.5.4　序列化的缺点

再次强调，把全局变量序列化到本地的方案，只是一种过渡型解决方案，它有几个硬伤：

1）每次设置全局变量的值都要强制执行一次序列化的操作，容易造成 ANR。

我们看一个例子，写一个新的全局变量 GlobalVariables3，它有 3 个属性，如下所示：

```
private String userName;
private String nickName;
private String country;

public void reset() {
    userName = null;
    nickName = null;
    country = null;

    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}

public String getUserName() {
    return userName;
}

public void setUserName(String userName) {
    this.userName = userName;
    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}

public String getNickName() {
    return nickName;
}

public void setNickName(String nickName) {
```

```
        this.nickName = nickName;
        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
    }
```

那么在给 GlobalVariables3 设值的时候，如下所示：

```
private void simulateANR() {
    GlobalVariables3.getInstance().setUserName("jianqiang.bao");
    GlobalVariables3.getInstance().setNickName("包包");
    GlobalVariables3.getInstance().setCountry("China");
}
```

我们会发现，每次设置值的时候，都要将 GlobalVariables3 强制序列化到本地一次。性能会很差，如果属性多了，强制序列化的次数也会变多，因为读写文件的次数多了，就会造成 ANR。

相应的解决方案很丑陋，如下所示：

```
public void setUserName(String userName, boolean needSave) {
  this.userName = userName;
  if(needSave) {
      Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
  }
}

public void setNickName(String nickName, boolean needSave) {
  this.nickName = nickName;
  if(needSave) {
      Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
  }
}

public void setCountry(String country, boolean needSave) {
  this.country = country;
  if(needSave) {
      Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
  }
}
```

也就是说，为每个 set 方法多加一个 boolean 参数，来控制是否要在改动后做序列化。同时在 GlobalVariables3 中提供一个 save 方法，就是做序列化的操作。

这样改动之后，我们再给 GlobalVariables3 设值的时候就要这样写了：

```
private void simulateANR2() {
    GlobalVariables3.getInstance().setUserName("bao", false);
    GlobalVariables3.getInstance().setNickName("包包", false);
    GlobalVariables3.getInstance().setCountry("China", false);
    GlobalVariables3.getInstance().save();
}
```

也就是说，每次 set 后不做序列化，都设置完后，一次性序列化到本地。这么写代码很恶心，但我之前说过，这只是权宜之计，相当于打补丁，是临时的解决方案。

2）序列化生成的文件，会因为内存不够而丢失。

这个问题也是在把全局变量都序列化到本地后发现的，究其原因，就是因为我们将序列化的本地文件放在了内存 /data/data/com.youngheart/cache/ 这个目录下。内存空间十分有限，因而显得可贵，一旦内存空间耗尽，手机也就无法使用了。因为我们的全局变量非常多，所以内部空间会耗尽，这个序列化文件会被清除。其实 SharedPreferences 和 SQLite 数据库也都是存储在内存空间上，所以这个文件如果太大，也会引发数据丢失的问题。

有人问我为什么不存在 SD 卡上，嗯，SD 卡确实空间大得很，但是不稳定，不是所有的手机 ROM 对其都有完好的支持，我不能相信它。

临时解决方案是，每次使用完一个全局变量，就要将其清空，然后强制序列化到本地，以确保本地文件体积减小。

3）Android 提供的数据类型并不全都支持序列化。

我们要确保全局变量的每个属性都可以序列化。然而，并不是所有的数据类型都可以序列化的。那么，哪些数据可以序列化呢？表 3-1 是我经过测试得到的结果。

表 3-1　各种类型数据对序列化的支持程度

类　　型	是否支持序列化
简单类型 int, String, Boolean 等	支持
String[]	支持
Boolean[]	支持
int[]	支持
String[][]	支持
int[][]	支持
ArrayList	支持
Calendar	支持
JSONObject	不支持
JSONArray	不支持
HashMap<String, Object>	因为 Object 可能是不支持序列化的 JSONObject 类型，所以 HashMap<String, Object> 不一定支持序列化
ArrayList<HashMap<String, Object>>	因为 Object 可能是不支持序列化的 JSONObject 类型，所以 ArrayList<HashMap<String, Object>> 不一定支持序列化

这就从另一方面证明了，我们尽量不要使用不能序列化的数据类型，包括 JSONObject、JSONArray、HashMap<String, Object>、ArrayList<HashMap<String, Object>>。

新项目可以尽量规避这些数据类型，但是老项目可就棘手了。好在天无绝人之路，我经过大量实践，得到一些解决方案，如下所示。

1）JSONObject 和 JSONArray

虽然 JSONObject 不支持序列化，但是可以在设置的时候将其转换为字符串，然后序列化到本地文件。在需要读取的时候，就从本地文件反序列化处理这个字符串，然后再把字符串转换为 JSONObject 对象，如下所示：

```
private String strCinema;
public JSONObject getCinema() {
    if(strCinema == null)
        return null;

    try {
        return new JSONObject(strCinema);
    } catch (JSONException e) {
        return null;
    }
}

public void setCinema(JSONObject cinema) {
    if(cinema == null) {
        this.strCinema = null;
        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
        return;
    }

    this.strCinema = cinema.toString();
    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}
```

JSONArray 如法炮制。只需要把上述代码中的 JSONObject 替换为 JSONArray 即可。

2）HashMap<String, Object> 和 ArrayList<HashMap<String, Object>>

因为 Object 可以是各种类型，有可能是 JSONObject 和 JSONArray，所以以上两种类型不一定支持序列化。

首选的解决方案是，如果 HashMap 中所有的对象都不是 JSONObject 和 JSONArray，那么以上两种类型就是支持序列化的。建议将 Object 全都改为 String 类型的。

```
private HashMap<String, String> rules;
public HashMap<String, String> getRules() {
    return rules;
}

public void setRules(HashMap<String, String> rules) {
    this.rules = rules;
    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}
```

其次，如果 HashMap 中存放有 JSONObject 或 JSONArray，那么我们就要在 set 方法中，遍历 HashMap 中存放的每个 Object，将其转换为字符串。

以下是代码实现，你会看到算法超级繁琐，效率也非常差：

```
HashMap<String, Object> guides;

public HashMap<String, Object> getGuides() {
    return guides;
}

public void setGuides(HashMap<String, Object> guides) {
    if (guides == null) {
        this.guides = new HashMap<String, Object>();
        Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
        return;
    }

    this.guides = new HashMap<String, Object>();
    Set set = guides.entrySet();
    java.util.Iterator it = guides.entrySet().iterator();
    while (it.hasNext()) {
        java.util.Map.Entry entry = (java.util.Map.Entry) it.next();

        Object value = entry.getValue();
        String key = String.valueOf(entry.getKey());

        this.guides.put(key, String.valueOf(value));
    }

    Utils.saveObject(AppConstants.CACHEDIR + TAG, this);
}
```

对于 HashMap<String, Object> 类型，无论是 get 方法还是 set 方法，都非常慢，因为要遍历 HashMap 中存放的所有对象。

ArrayList<HashMap<String, Object>> 是 HashMap<String, Object> 的集合，所以对其进行遍历，会更加慢。

在遇到了 N 多次以上解决方案导致的 ANR 之后，我决定将这两种超级复杂的数据结构，全部改造为可序列化的实体。好在这样的数据类型在 App 中不太多，重构的成本不是很大。

3.5.5　如果 Activity 也被销毁了呢

如果内存不足导致当前 Activity 也被销毁了呢？比如说旋转屏幕从竖屏到横屏。

即使 Activity 被销毁了，传递到这个 Activity 的 Intent 并不会丢失，在重新执行 Activity 的 onCreate 方法时，Intent 携带的 bundle 参数还是在的。所以，我们的解决方案是重新执行当前 Activity 的 onCreate 方法，这样做最安全。

但是另一个问题就又浮出水面了：Activity 需要保存页面状态吗？

想必各位亲们都看过 Android SDK 中的贪食蛇游戏，它讲的就是在 Activity 被销毁后保存贪食蛇的位置，这样的话，恢复该页面时就能根据之前保存的贪食蛇的位置继续游戏。

这个 Demo 用到了 Activity 的以下 2 个方法：

- onSaveInstanceState()
- onRestoreInstanceState()

网上关于以上两个方法的介绍和讨论不胜枚举，下面只是分享我的使用心得。

对于游戏以及视频播放器而言，保存页面上每个控件的状态是必须的，因为每当 Activity 被销毁，用户都希望能恢复销毁之前的状态，比如游戏进行到哪个程度了，视频播放到哪个时间点了。

但是对于社交类或者电商类 App 而言，页面繁多，多于 100 个页面的 App 比比皆是。如果每个页面都保存所有控件的状态，工作量就会很大，要知道这样的 App，每个页面都有大量的控件和交互行为，需要记录的状态会很多。

所以，不记录状态，直接让页面重新执行一遍 onCreate 方法，是一种比较稳妥的方法。丢失的数据，是页面加载完成之后的用户行为，让用户重新操作一遍就是了。

额外说一句，想保存页面状态，是件很难的事情。这一点 WindowsPhone 做得很好，因为它是基于 MVVM 的编程模型，它把业务逻辑 ViewModel 和页面 View 彻底分开，同时，View 中的每个控件的状态，都与 ViewModel 中的属性进行了绑定，这样的话，View 中控件状态变化，ViewModel 中的属性也会相应变化，反之亦然。所以把 ViewModel 序列化到本地，即使 View 被销毁了，重新创建 View，并把保存到本地的 ViewModel 与之绑定，就可以重现 View 被销毁之前的状态——我们称为墓碑机制。

不得不说，微软的墓碑机制确实做得很好，它吸取了 iOS 和 Android 的经验，让恢复页面状态变得容易很多。

3.5.6 如何看待 SharedPreferences

在我们决定禁止使用全局变量后，曾经一段时间确实有了很好的效果，但是我后来仔细一看项目，新的全局变量倒是真的不再有了，大家都改为存取 SharedPreferences 的方式了。

在我看来，SharedPreferences 是全局变量序列化到本地的另一种形式。SharedPreferences 中也是可以存取任何支持序列化的数据类型的。

我们应该严格控制 SharedPreferences 中存放的变量的数量。有些数据存在 SharedPreferences 中是合理的，比如说当前所在城市名称、设置页面的那些开关的状态等等。但不要把页面跳转时要传递的数据放在 SharedPreferences 中。这时候，要优先考虑使用 Intent 来传递数据。

3.5.7 User 是唯一例外的全局变量

依我看来，App 中只有一个全局变量的存在是合理的，那就是 User 类。我们在任何地方都有可能使用到 User 这个全局变量，比如获取用户名、用户昵称、身份证号码等等。

User 这个全局变量的实现，可以参考本章讲解的例子。

每次登录，都要把登录成功后获取到的用户信息保存到 User 类。以后，每当 User 的属性有变动时，我们都要把 User 保存一次。退出登录，就把 User 类的信息进行清空。与之前我们所设计的全局变量不同，App 启动时不需要清空 User 类的数据。因为我们希望 App 记住上次用户的登录状态以及用户信息。再讲下去就涉及用户 Cookie 的机制了。

3.6　本章小结

本章讨论了 App 中的集中几种场景的设计，其中包括：如何设计 App 图片缓存，如何优化网络流量，对城市列表的重新思考，如何让 HTML5 在 App 中发挥更大的作用，如何解决全局变量过多导致的内存回收问题，等等。

下一章，我将介绍 Android 的编码规范和命名规范。

第 4 章 Chapter 4

Android 命名规范和编码规范

写本章的时候，我正在恶补《灌篮高手》。在狂笑过后，冷静地思考赤木军团的成与败。这个团队的每个成员都是个性强到爆表，这恰恰是该团队的软肋，即每个成员都不会为了团队的利益而抛弃个性。

由此想到写程序中，编码规范是泯灭程序员个性的一项制度，但对于整个团队而言，却是一件利器。它能让代码整齐有序，任何人都能接手其他人的工作，而不会因为代码风格迥异而花费过多时间。

代码是程序员的第二张脸。每个程序员在嘲笑别人代码的同时，有没有想过自己的代码也会成为别人的谈资呢？

制定规范不需要太多的理论知识，只要记住两点就够了：尽量简单，多写注释。

4.1 Android 命名规范

无规矩不成方圆。

一个项目必须有统一的命名规范，只有这样，才像是一个团队做的产品。命名规范有以下几点需要注意：

首先，命名规范不能反人类。

我曾经见过有的 Team Leader 这样为 Acitivty 设计命名规范：

PersonActivityAddCustomer.java

我看过他们团队的项目，所有的 Activity 全都是上述这种“模块名 +Activity+ 页面名”

的命名方式。我很奇怪的是，为什么不设计成以下方式呢，如图4-1所示。

这就涉及人生观、价值观和审美观了，作为Team Leader，不能因为自己的癖好，制定出一些变态的规范，而让其他团队成员跟着受罪。

其次，要望文而知义，清晰准确。比如说登录页面的登录按钮，命名时就不能像button1这样随心所欲，要类似于login_button（资源文件）或btnLogin（java代码中的按钮实例）这样的名称。

图4-1 Activity的命名规范

此外，遇到MyGridView之类的命名，就可以把创建该文件的同学拉出去打80大板了，而且要肚皮朝上的那种打法。

最后，命名规范千万别制定太多，多了会让人烦，没人看，更没人遵守。要做到简单易记，适可而止。

下面说点具体的规则：

1）Java类文件命名规范。

- Activity命名规范：以Activity作为后缀。比如说PersonActivity。
- Adapter命名规范：以Adapter作为后缀。比如说PersonAdapater。
- Entity命名规范：大多以Entity作为后缀。比如说PersonEntity。值得注意的是，User是全局变量，不算是实体，不受此约束。

2）资源文件命名规范。

- layout目录下的文件命名规范：
  - 页面布局文件。以act_为前缀，以Activity所在的Package作为中缀，以Activity的名称（去掉Activity后缀）作为后缀。注意都是小写。

    例如，对于Person这个模块下的AddCustomerActivity，它的layout文件就应该是：act_person_addcustomer.xml。
  - ListView中的item布局文件。以item_作为固定前缀，列表项的名称为后缀。注意都是小写。例如，某个页面下有一个用户列表，控件名为lvUserList，那么item的layout就应该是：item_lvUserList.xml。
  - Dialog布局文件。

  以dlg_作为固定前缀，Dialog的功能名称为后缀。注意都是小写，例如：dlg_hint.xml。
- drawable目录下文件命名规范。drawable目录下的资源，大部分是图片，此外，还有一部分xml文件，用于Selector。但无论是图片，亦或Selector文件，都应该遵守下述命名规范：
  - 对于只在一个页面使用的资源，就以该页面的名称作为前缀。
  - 对于只在一个模块下多个页面使用的资源，就以该模块的名称作为前缀。
  - 对于在各个模块、各个页面都有可能使用的资源，比如说上导航、下导航，以common作为前缀。

3）Java 类中控件对象的命名规范。

控件类型缩写 + 控件的逻辑名称（首字母大写），比如登录按钮，就可以命名为 btnLogin。表 4-1 列出了一些常用控件的缩写。

表 4-1　常用控件的缩写

控　件	缩　写	控　件	缩　写
LayoutView	lv	EditText	et
RelativeView	rv	TimePicker	tp
TextView	tv	toggleButton	tb
Button	btn	ProgressBar	pb
ImageButton	img	WdbView	wv
ImageView	iv	RantingBar	rb
CheckBox	chk	Tab	tab
RadioButton	rb	ListView	lv
DatePicker	dp	MapView	mv

4）Layout 中控件对象的命名规范。

这里我建议，与 Activity 中相对应的控件名称保持一致。这样的好处是可以迅速 copy-paste 出以下代码而杜绝任何的潜在错误：

```
Button btnLogin = (Button)findViewById(R.id.btnLogin);
```

但是这样做与传统 Android 的命名规范就不一致了，至少违反了 2 条：不应该出现大写字母，btn 和 Login 之间没有以下划线进行连接，以下的命名才是中规中矩的：

```
Button btnLogin = (Button)findViewById(R.id.sign_in_button);
```

我认为以上两种命令方式都是可行的，只要认定其中一种并坚持用下去，就是我们需要的规范，只要不反人类即可。

5）strings.xml 中常量的命名规范。

因为这些值大多在 Layout 中的控件上使用，所以以该常量所在的 Activity 名称作为前缀，后面接控件名称，再后面就自由发挥了，比如登录页面的登录按钮上显示的文字，就可以命名为：loginActivity_btnLogin_text。

另一种使用场景则是在 Java 代码中使用，可能出现在 Activity 中，也可能出现在工具类 Utils 中，这时候，如果是和具体 Activity 相关，那么规则和上面的一样，以所在的 Activity 名称作为前缀，如果涉及和公共模块和控件相关，就以 common_ 作为前缀。

strings.xml 的规则可以灵活一些。我们甚至可以将其按照模块拆分为多个 strings 文件，只要 resoures 标签下都是 string 标签就行，编译打包时会自动将同类文件进行合并，如图 4-2 所示。

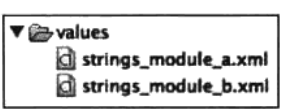


图 4-2　strings.xml 的命名规范

这样做的好处是，各个模块维护各自的strings.xml。但为常量命名时就一定要以模块名作为前缀了，不然很容易产生重名的情况，从而编译报错。

6）常量命名。

这一点遵守Java的命名规范，即只能包含字母和下划线_，字母全部大写，单词之间用下划线_隔开。

关于命名规范的事情，可以写十几页密密麻麻的规则，我这里只提到最关键的几点。其他的，要么是不重要，要么是使用场景少，所以都可以自由发挥。

记住，命名规范的作用在于：

- 好的文件命名规范，让几千个文件分门别类的放在好找的位置。
- 好的对象命名规范，让整个项目的代码风格整齐一致。

切记，不能为了规范而规范，网上的各种规范不胜枚举，让人眼花缭乱，但是过多的规范，会让App这个轻量级的应用背上越来越沉重的包袱，举步维艰。

制定一套切实可行、易于遵守的命名规范，是每个Team Leader的必备技能。

4.2 Android编码规范

前面写的内容就是为了编码规范做铺垫。

有人说，编码规范网上多的是，我就见过有人网上摘抄了一些然后跟我讲这是他们团队的编码规范的。这不对，不能为了规范而规范，规范是为了解决实际问题的。我个人经过血和泪的后的经验总结如下：

1）要分门别类存放各种类，如图4-3所示。

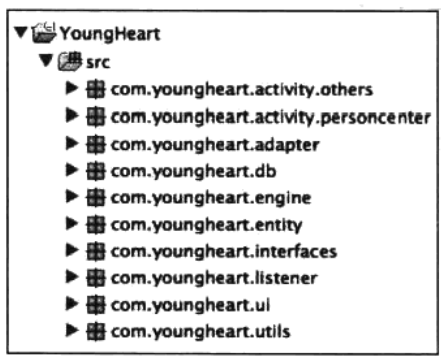


图4-3 分门别类存放各种类

2）要怎么使用findViewById语句？

看过项目中很多人都这么使用findViewById语句：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  ((TextView)findViewById(R.id.login_status_message))
        .setVisibility(View.VISIBLE);
}
```

从面向对象的角度出发，以下写法是不是更好呢：

```
TextView tvLoginStatusMessage;

@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
```

```
    setContentView(R.layout.activity_main);

    tvLoginStatusMessage =
        (TextView)findViewById(R.id.login_status_message);
    tvLoginStatusMessage.setVisibility(View.VISIBLE);
}
```

我们观察到，把控件对象声明在 Activity 级别会更好一些，在 initViews 中对其赋值，而在 Activity 的其他地方还有使用的机会。

也许有人会质疑，如果这个控件只使用了一次，那么第一种写法其实是最好的。但这毕竟是少数，我们现在要统一编码规范，只能牺牲一部分人的利益，来达到协同工作的效率最大化。

3）Layout 中的常量，要在资源 strings.xml 中定义。

以下的使用方式是错误的：

```
<TextView android:text="评论" ……../>
```

我们要将“评论”这个常量定义在 strings.xml 中：

```
<resources>
    <string name="tvPersonCenter">评论</string>
</resources>
```

然后在 Layout 布局文件中这样使用：

```
<TextView android:text="@string/tvPersonCenter" ……../>
```

另一方面，在 Activity 中也需要设置一些常量，不能把它写死，要将其定义在 strings.xml 中，然后每次都从资源文件中取值，如下所示：

```
String loadingMessage = this.getString(R.string.loadingmessage);
```

Eclipse 编译时会检查上述问题，显示在 Warnings 列表中，开发人员要定期修复 Warning 中类似的问题。

4）Layout 中所有控件的字体大小，都定义在 dimens.xml 中，它相当于网站的 CSS 样式表，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <!— 字体定义 —>
    <dimen name="font_size_tiny">10sp</dimen>
    <dimen name="font_size_small">12sp</dimen>
    <dimen name="font_size_normal">14sp</dimen>
    <dimen name="font_size_normal_high">16sp</dimen>
    <dimen name="font_size_large">18sp</dimen>
    <dimen name="font_size_large_high">20sp</dimen>
    <dimen name="font_size_xlarge">22sp</dimen>

    <!— 边距 —>
```

```
<dimen name="offset_2dp">2dp</dimen>
<dimen name="offset_4dp">4dp</dimen>
<dimen name="offset_6dp">6dp</dimen>
```

使用方式如下：

```
<TextView android:textSize= "@dimen/font_size_normal" …… />
```

此外，对于所有控件的Margin偏移量，我们也需要统一规格，正如上面的dimens.xml中的定义，有若干种尺寸事先定义好供我们选择。

```
<LnearLayout
    android:layout_marginLeft= "@dimen/offset_2dp"
    android:layout_marginRight= "@dimen/offset_4dp"
```

这样做的好处是，只要稍微修改一下dimens.xml中的定义，就可以批量修改页面的样式。Android的手机千奇百怪，各种分辨率都存在，在一些特殊机型上，font_size_normal字体可能会过大或者过小，我们将其修改为13sp或15sp，就可以迅速看到修改是否符合我们的审美观了。

做得更彻底些，是使用style来统一控件的风格。如果有必要，请使用之。

5）在Acitivity中，定义新的生命周期，从而将onCreate方法拆分为以下3部分：

- initVariables：初始化变量（包括Intent上的数据和Activity内部使用的变量）。
- initViews：加载layout布局文件，初始化控件。
- loadData：调用MobileAPI。

拆分onCreate方法是设计模式中单一职责原则的体现。

6）坚持使用fastJSON自定义实体来作为MobileAPI的数据载体。

像JSONObject、JSONArray、HashMap<String, Object>、ArrayList<String, Object>这些不能序列化的实体，都禁止使用。除非它们仅仅是为了实现某个算法，在方法内部临时使用。

千万别偷懒哦。如果觉得自定义实体很麻烦，建议使用我在第1章1.4节介绍的那个实体自动化生成工具EntityGenerator。

7）页面之间传值，坚持使用Intent携带序列化实体数据的方式。禁止为了省事使用全局变量进行传值的方式。

8）为控件添加事件。以按钮为例，为按钮添加点击事件，统一使用以下这种方式：

```
btnLogin = (Button)findViewById(R.id.sign_in_button);
btnLogin.setOnClickListener(
        new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                login();
            }
        });
```

严禁在 Layout 的控件声明中直接声明事件方法，以下代码是不允许的：

```
<Button android:onClick="gotoLogin" …… />
```

9）Activity 中不要嵌套内部类，尽量都独立出来，该放哪儿就放哪儿。

10）Adapter 的编码规范如下：

- 所有 Adapter，都放在 adapter 这个包中。
- Adapter 绑定的数据，一律为 ArrayList< 自定义可序列化实体 >。
- 在 Adapter 中创建适合于列表自身的 ViewHolder 实体类。请统一命名为 ViewHolder。

11）实体不要在不同模块间共享，但是可以在同一模块下的不同页面间共享。比如说，不要在美食模块和酒店模块共用同一个实体，但是在美食模块的列表页和详情页，可以共用同一个实体。

12）为节省内存，请使用 ArrayList< 自定义实体 >，而不是 HashMap。

ArrayList 虽然慢一点，每次查找一个元素，都是 o(n)，而 HashMap 则是 o(1)，但是 ArrayList 在内存的使用上要少于 HashMap。对于 Android 手机，尤其是配置很低的手机，我们开发 App 的策略是尽量不占用太多内存，所以请优先选择 ArrayList< 自定义实体 >。

13）图片的处理，请统一使用第三方组件 ImageLoader 或 Fresco 来进行异步加载。

14）什么时候使用 SharedPreferences？对于简单的配置信息，设置页面的各种开关，这些都是要保存在 SharedPreferences 中的。对于复杂的对象，比如说 User 类，比如说城市基础数据，这些数据还是要存储到本地文件中。

15）尽量使用 ApplicationContext 代替 Context，否则会引起内存泄漏。当然，也不是任何地方 ApplicationContext 都可以代替 Context，请参见第 6 章，将详细介绍因为使用不当而导致的崩溃。

16）数据类型转换一定要进行校验。请使用第 1 章 1.6 节介绍的类型安全转换函数，这些函数能帮你做两件事，一是转换失败会有默认值；二是由 try…catch…保护，不会轻易抛出空指针或者类型转换失败的崩溃。

17）使用常量来代替枚举。众所周知，枚举的每个值只能是一个整数，而没有 toString 这样的方法，所以不如在类中定义一个字符串常量方便。此外，有人说枚举的内存开销要比常量大，但我觉得这不是判断常量比枚举好的理由。

4.3　统一代码格式

最后说说代码格式的问题。这就完全是个人偏好了。有人喜欢把方法的左括号写在下一行，有人则把方法的左括号与方法名称放在同一行；有人喜欢一句话就是一行代码，有人则喜欢把一句话分成若干行来增强可读性。总之是各有各的喜好。

但是作为一个团队，我们希望在一个项目中的代码，看上去像是一个人写的。那么除了

要求所有开发人员遵守编码规范和命名规范外，统一的代码格式也是非常重要的。

Android 源码中包含了一份 android-formatting.xml，专门用于统一代码格式。每个开发人员在 Eclipse 导入这个文件后，以后在执行快捷键 ctrl+shift+f 时，Eclipse 都会根据这个文件来调整代码格式。导入方法如下：

```
window->preferences->java->Code style->Formatter 中导入 android-formatting.xml
```

这个文件我们可以签入到 SVN 上，这样就能所有开发人员导入的是同一份代码格式文件。[⊖]

一方面，我们统一代码格式、制定编码规范和命名规范，另一方面，我们需要检查开发人员是否严格遵守这些规范，人工检查会累死人的，需要有一个自动检查的工具。这里我推荐 checkstyle[⊜]。这是个很有趣的工具，博客园上有专门的介绍[⊜]。

但是，这个工具只是锦上添花，不必花太多精力在上面。我们还是要把更多功课做足在优化 App 性能、Crash 修复上。

4.4 本章小结

本章讨论了 Android 开发过程中需要遵守的 2 个规范：命名规范，编码规范。

在此基础上，为了统一 Android 项目中的编码格式，我们还引进了 android-formatting.xml 和 checkstyle。

下面的章节，我将介绍线上 Crash 的收集、分析和修复。

⊖ 更详尽的介绍，可以参考 CSDN 上这篇文章：http://blog.csdn.net/thl789/article/details/8040603。

⊜ 官方网站地址：http://checkstyle.sourceforge.net/。

⊜ 详细内容请参见 http://www.cnblogs.com/qianxudetianxia/archive/2012/01/01/2309102.html。

第二部分 Part 2

App 开发中的高级技巧

- 第 5 章　Crash 异常收集与统计
- 第 6 章　Crash 异常分析
- 第 7 章　ProGuard 技术详解
- 第 8 章　持续集成
- 第 9 章　App 竞品技术分析

工欲善其事，必先利其器。

这一部分讨论4个主题，都和Android日常开发工作无关，但如果有了这些机制，将极大提高App项目的质量和开发效率。

- 首先是Android线上崩溃的收集、分析和修复。有了这个利器，Android的稳定性将极大提高。一开始我只准备了五十多个崩溃情况，后来越写越多，整整写了6个月，扩充到一百多个。其实每个Crash在网上都有人进行介绍，只是众说纷纭，有真有假。于是我做了很多Demo试图重现崩溃以分辨网上文章的真假，其中很多优秀的思想，我会详细介绍。
- 其次是ProGuard。除了一份官方文档，市面上还没有一份详尽介绍ProGuard的文章，网上的文章倒是很多，但大都很简单。于是我仔细研究了官方文档，参考了网上大量的技术文章，并结合自身经验，写下这一篇专门给Android开发人员看的文章。
- 再次是适用于App的持续集成（CI）。无论是使用Ant还是Maven，亦或是当下最流行的Gradle，都要确保DailyBuild和BatchBuild的机制。
- 最后是竞品分析。不光是竞品，因为我研究的是技术实现，所以会覆盖到市面上口碑比较好的100款App，每款都包括iOS和Android两种，其中在iOS上花的时间会更多一些。我发现每个App在技术实现上都有若干闪光点，当然也有做得不好的地方。把这些闪光点总结下来，很有必要，这章干货很多，很多都是第一手的研究心得，分享给读者。

第 5 章 Chapter 5

Crash 异常收集与统计

本书第 5 章和第 6 章，将给出 Android 线上 Crash 的解决方案，也就是 Crash 分析三部曲：收集、统计、分析。

1）收集：把 Crash 收集到本地数据库。

2）统计：对每天线上大量的 Crash 进行去重、分类。

3）分析：逐个分析各类 Crash，重现异常发生的例子，给出解决方案。

其中第 1 和第 2 点由于篇幅不长，不能独立成篇，所以合并为第 5 章。

5.1 异常收集

一个健壮的 App 应该能搜集运行中所有的 Crash 信息，并将其发送到服务器以便程序员进行分析。

对于任何一款 App 而言，无论页面数量多少，我们也不可能在每个页面的每个方法都加上 try…catch…语句来捕获 Crash，而是需要一套统一的解决方案，将 Crash 一网打尽。

为此我们需要了解一个很重要的类：UncaughtExceptionHandler，用来处理未捕获的异常。未捕获异常指的是在程序中未使用 try…catch…语句而抛出的异常。我们需要在 App 级别处理这些未捕获到的异常，算是最后一道关卡。

如果程序出现了未捕获异常，默认会弹出系统的强制关闭对话框。我们需要实现此接口，并在 App 中对其进行注册。这样当未捕获异常发生时，就可以做一些个性化的异常处理操作。

于是我们设计一个 CrashHandler 类，使之继承自 UncaughtExceptionHandler，来定义我

们自己的异常捕获逻辑，如下所示：

```
/**
 * UncaughtException处理类，当程序发生Uncaught异常的时候，
 * 由该类来接管程序，并记录发送错误报告.
 * 需要在Application中注册，为了要在程序启动器就监控整个程序。
 */
public class CrashHandler implements UncaughtExceptionHandler {
    public static final String TAG = "CrashHandler";
    public static final String APP_CACHE_PATH =
            Environment.getExternalStorageDirectory().getPath()
            + "/YoungHeart/crash/";
```

我们要实现 CrashHandler 的 uncaughtException 方法，详细的代码如下所示：

```
/**
 * 当UncaughtException发生时会转入该函数来处理
 */
@Override
public void uncaughtException(Thread thread, Throwable ex) {
    if (!handleException(ex) && mDefaultHandler != null) {
        //如果用户没有处理则让系统默认的异常处理器来处理
        mDefaultHandler.uncaughtException(thread, ex);
    } else {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            Log.e(TAG, "error : ", e);
        }
        //退出程序
        android.os.Process.killProcess(
                android.os.Process.myPid());
        System.exit(1);
    }
}
```

这里只介绍其中最关键的方法，也就是 handleException 方法，这个方法做三件事情：

1）发错误日志到服务器。

2）给用户崩溃前的友好提示。

3）把错误日志记录到 SD 卡。

其代码如下所示：

```
/**
 * 自定义错误处理，收集错误信息
 * 发送错误报告等操作均在此完成
 *
 * @param ex
 * @return true:如果处理了该异常信息；否则返回false.
 */
private boolean handleException(Throwable ex) {
    if (ex == null) {
```

```
            return false;
        }

        //把 crash 发送到服务器
        sendCrashToServer(context, ex);

        //使用 Toast 来显示异常信息
        new Thread() {
            @Override
            public void run() {
                Looper.prepare();
                Toast.makeText(context,
                        "很抱歉，程序出现异常，即将退出 .",
                        Toast.LENGTH_SHORT).show();
                Looper.loop();
            }
        }.start();

        //保存日志文件
        saveCrachInfoInFile(ex);
        return true;
    }
```

sendCrashToServer 方法负责将捕获的异常发送到服务器，为此需要 MobileAPI 提供一个接口。表 5-1 中的信息都是很重要的，我们要事先准备这些数据。

表 5-1　Crash 数据表结构

字段名称	描　　述
id	自增 id
client_type	Crash 所在的 App
page_name	Crash 所在的 Activity 名称
exception_name	Crash 名称
exception_stack	Crash 详细信息
crash_type	1 表示崩溃了，0 表示被 try-catch 捕获到了
app_version	当前 App 的版本
os_version	Android 系统的版本
device_model	Android 手机型号
device_id	Android 手机设备号
network_type	网络类型，是否为 WIFI
channel_id	渠道号
client_type	标记 Crash 发生在 Android 还是 iPhone
memory_info	Crash 发生时的内存使用情况
crash_time	Crash 发生时间，在插入数据库时，由数据库自动生成

有了上述机制，所有的异常就全都能被捕获到了。但并不是所有的异常都导致崩溃——我们希望尽可能留住用户，而不是 App 崩溃后重启。因为用户是不会重启打开 App 的，至

少我不会。

有些异常是不严重的。比如说 MobileAPI 的数据不规范，该返回数值的却返回了字符串，不能为空的字段却返回了空值。这些数据中，有些数据仅仅是为了显示，显示与否无伤大雅，所以即使解析时出了问题抛出异常，也不应该崩溃。我们应该在相应的 Activity，在具体解析数据的地方，加一层自定义的 try…catch…语句，来捕获这些已知的异常。

需要注意的是，如果异常在 Activity 中就被捕获到了，就不会将其再交由 Application 级别的 CrashHandler 类去处理了。所以我们要在这个 Activity 的 try…catch…语句中，手动把异常信息发送到服务器。在具体的 Activity 中，我们会将 CrashType 设置为 0，而在 CrashHandler 中才会将 CrashType 设置为 1。

5.2 异常收集与统计

目前业界对 App 线上 Crash 的收集一般有 2 种，要么记录到第三方平台，要么记录到自己的数据库中。

使用第三方 Crash 收集分析平台的好处是，他们能提供一套完整的 Crash 分类和报表统计工具。比如腾讯的 Bugly 平台，他们还能提供技术支持，告诉你某类要怎么修复。

接下来我要介绍的是，如何记录到自己的数据库中，然后自行统计分析这些 Crash 数据。其实并不难。

5.2.1 人工统计线上 Crash 数据

最一开始，我们是通过人工的方式手动统计这些 Crash 数据的，当时是把这活儿分给了新来的 3 个 Android 开发人员，因为新人往往有股子冲劲儿。

第一次我们用了 3 天时间，分析了 1 天的 Crash 数据，大约 2000 多笔，对每个 Crash 进行了分类，我们在分析中就发现：

1）有很多重复的 Crash。这其中分很多种情况。

- ❑ 有不同设备在不同时间发出来重复的 Crash，这时候要检查是否只对某些机型或 Android 版本才会发生类似问题，比如说 Android2.1 不支持 https。
- ❑ 有不同设备在一个时间段发出来重复的 Crash。这时候要检查 MobileAPI 是否返回了脏数据而 App 没有使用 try…catch…语句捕获到。
- ❑ 有相同设备在很短的时间段内频繁发送了重复的 Crash。这是因为 App 没有做好崩溃后的善后工作导致的，它试图重新启动发生崩溃的那个 Activity，然后重启过程中因为要重新执行 onCreate 方法而这个方法有空指针，于是就会造成“崩溃—重启—崩溃—重启”的死循环，直到用户强制关闭 App。对此，我们需要去除重复数据。

2）每笔异常信息都包括以下 2 部分数据信息：

- ❑ exception_name：Crash 对应的异常名称。

❑ exception_stack：Crash 的详细信息。

不要以 exception_name 作为 Crash 分类的标准，这是不准确的。比如说，因为空指针 NullPointer 导致的崩溃，但是 exception_name 却是 RuntimeException。所以 exception_name 只能作为 Crash 的参考标准，而产生 Crash 的真正原因，则隐藏在 exception_stack 中。

3）exception_stack 中含有 OutOfMemory 内容的，都是内容溢出导致的。但是逆命题不成立。因为有些内容溢出导致的崩溃，抛出的异常信息却不包括 OutOfMemory 内容，比如说 ResourcesNotFoundException，有很多情况是资源明明存在于 App 中但还是说找不到，“睁眼说瞎话”，于是我们也只好眼睁睁地看着它崩溃了而无能为力。

4）对于空指针 NullPointerException 这个“不治之症”，我们观察到的情况是，NullPointerException 只是导致崩溃的结果，而不是原因。导致空指针的情况五花八门，有时，我们要留意 exception_stack 中 Cause by 后面的内容，如下所示：

```
java.lang.RuntimeException: Failure delivering result ResultInfo
{who=null, request=3, result=-1, data=Intent{ (has extras)
contextId=0, taskId=0 }} to activity {包名称/Activiyy名称}
```

如果只看前半段信息，根本不知道问题所在，继续向下看，会发现在 Cause by 处有空指针的提示信息，如下所示：

```
Caused by: java.lang.NullPointException at
包名称.Activity名称.onActivityResult(Unknown Source) at
android.app.Activity.dispatchActivity(Activity.java:5352) at
```

5）窗体泄露这类问题，基本都是想关闭弹出框的时候，却发现承载它的宿主已经不在。

6）ListView 和 Adapter 相关的 Crash 基本都发生在分页获取数据的场景，数据源发生了改变，却没有及时通知 ListView 和 Adapter。

5.2.2　第一个线上 Crash 报表：Crash 分类

在找到这些 Crash 的共性后，我们开始调整 Crash 分析的策略。我会引入 SQL Server 和 C# 作为分析的工具。

1）首先，每天上班，我会把昨天 24 小时的 Crash 数据从服务器上取下来，导出为 excel 文件，然后再把 excel 还原到本地 SQL Server 数据库的 CrashDB 这个表。这样我就可以在本地数据库上写各种各样的 SQL 语句来分析这些数据了，不必担心直接在线上直接操作 SQL 而把线上数据库搞死。

2）接下来我会执行一个存储过程 UpdateCrashDesc，把这些 Crash 数据分门别类，为此，我要为存放 Crash 的表 CrashDB 加一个字段 crash_desc，用来表明 Crash 是哪个类别的。

存储过程 UpdateCrashDesc 的逻辑就是把符合某类特征的那些 Crash，设置其 crash_desc 字段为同一个值，如下所示：

```
CREATE PROCEDURE [dbo].[UpdateCrashDesc]
```

```
AS
BEGIN
  SET NOCOUNT ON;

update CrashDB
set crash_desc = '内存溢出'
where exception_stack like '%OutOfMemory%'
and crash_desc is null

update CrashDB
set crash_desc = 'ClassCastException'
where crash_desc is null
and exception_stack like '%java.lang.ClassCastException%'

update CrashDB
set crash_desc = '数组越界'
where crash_desc is null
and exception_stack like '%OutOfBoundsException%'

update CrashDB
set crash_desc = 'java.lang.VerifyError'
where crash_desc is null
and exception_stack like '%java.lang.VerifyError%'

update CrashDB
set crash_desc = '各个页面的空指针'
where crash_desc is null
and exception_stack like '%NullPointerException%'

update CrashDB
set crash_desc = 'is your activity running?'
where crash_desc is null
and exception_stack like '%is your activity running?%'

- -中间省略若干 Update 语句

update CrashDB
set crash_desc = '不明觉厉'
where crash_desc is null

END
```

考虑到章节限制，我只贴出了 UpdateCrashDesc 这个存储过程的部分代码，全部代码请参见我博客上的源码[⊖]。值得注意的是：

每条 Update 语句代表做一次分类操作。一开始我也只有十几条 Update 语句，后来慢慢扩充到五十几条。每次新增一个 Crash 分类，就加在“不明觉厉”这条 Update 语句之上即可。

“不明觉厉”这个 Crash 类别，是经过上述五十几条 Update 语句筛选后，剩下的 Crash。这类 Crash 数量不能超过 100，否则，就应该从中继续寻找共性，拆分成新的 Crash 类别，编写一个新的 Update 语句。

⊖ 代码地址为：http://www.cnblogs.com/Jax/p/4573575.html。

3）接下来我会执行一个存储过程 GroupOnlineCrash，用以统计各类 Crash 的数量。

```
CREATE PROCEDURE [dbo].[GroupOnlineCrash]
AS
BEGIN
  SET NOCOUNT ON;

select * into #temp1 from CrashDB
where client_type=20
order by page_name, exception_name, exception_stack

select crash_desc, COUNT(crash_desc) as count from #temp1
group by crash_desc
order by COUNT(crash_desc) desc

END
```

执行结果如图 5-1 所示。

	crash_desc	count
1	java.lang.VerifyError	350
2	Package manager has died	308
3	各个页面的空指针	198
4	InflateException	164
5	UnsatisfiedLinkError	122
6	内存溢出	109
7	doInBackground	100
8	Failure delivering result ResultInfo	48
9	数组越界	43
10	不明觉厉	39
11	NumberFormatException	17
12	Permission相关	13
13	Resources$NotFoundException	9
14	Fragment相关,百度查Can not perform this action after ...	7
15	is your activity running?	4
16	ClassCastException	3
17	ListView刷新数据	3
18	SQLiteException相关	2
19	view not attached to window manager	1
20	libcore.io.DiskLruCache	1
21	parameter must be a descendant of this view	1
22	TransactionTooLargeException	1

图 5-1　线上 Crash 统计图

对于图 5-1 中排名前 10 的线上 Crash，我们要花大力气去分析、修复它们。

5.2.3　第二个线上 Crash 报表：Crash 去重

我们在前面手工统计分析的时候就已经发现，Crash 有很多重复。我们接下来要去重，

从而看出每天到底有多少种不同的 Crash。

去重工作由 4 部分组成，如图 5-2 所示。对这 4 部分工作介绍如下。

图 5-2 去重工作的 4 个步骤

1. 去除数字不同导致的重复

去重主要是在 exception_stack 字段上做文章，也就是 Crash 的详细信息。我们发现，很多时候同一类 Crash，它们的 exception_stack 字段仅仅是数字的不同，比较典型的有以下几种情况：

- 发生崩溃时的代码行不同，如下所示：

```
package manager has died at android.app.ActivityThread
.performLaunchActivity(ActivityThread.java:2215)

package manager has died at android.app.ActivityThread
.performLaunchActivity(ActivityThread.java:2296)
```

- 运行时的数值不同，如下所示。
  - 崩溃信息中的 # 后面的数字不同：

```
android.view.InflateException:
Binary XML file line #8: Error inflating class<unknown>

android.view.InflateException:
Binary XML file line #32: Error inflating class<unknown>
```

  - 崩溃信息中的 result= 后面的数字不同：

```
java.lang.RuntimeException: Failure delivering result
ResultInfo {who=null, request=5, result=-1

java.lang.RuntimeException: Failure delivering result
ResultInfo {who=null, request=3, result=-1
```

  - 崩溃信息中的 ViewRootImpl$W@@ 后面的数字不同：

```
android.view.WindowManager$BadTokenException:
Unable to add window - - token android.app.LocalActivityManager
$LocalActivityRecord@45a58ee0 is not valid;
 is your activity running?

android.view.WindowManager$BadTokenException:
Unable to add window - - token android.app.LocalActivityManager
$LocalActivityRecord@4012ef33 is not valid;
 is your activity running?
```

虽然数值不同，但其实是相同的 Crash。一种好的去重方案是将这些数值都统一改为 1000。这样 exception_stack 字段就全都一样了。但是一旦改成 1000，就没机会恢复为之前的值了，所以我的做法是，在 CrashDB 这个表再增加一个字段 dis_info，把 exception_stack 这

个字段的数据复制一份到 dis_info 字段，然后我们在 dis_info 字段上进行修改。修改的方法是借助于正则表达式，进行批量替换。

按照上述思路，我使用 C# 写了一个小工具，它可以遍历 CrashDB 表中的每个 Crash，取出它的 exception_stack，使用正则表达式进行替换，然后赋值给 dis_info 字段。

以下是 C# 中使用正则表达式进行替换的实现代码：

```
string dis_info = (String)read["exception_stack"];

//from    .java:836)
//to      .java:1000)
string str = Regex.Replace(dis_info, @".java:\d*", @".java:1000");

//from    window android.view.ViewRootImpl$W@41ec8258
//to        window android.view.ViewRootImpl$W@12345678
//@后面是8位字符
string str2 = Regex.Replace(str, @"\w{8}*", @"@12345678");

//from    request=327681
//to      request=1000
string str3 = Regex.Replace(str2,
        @"request=\d*", @"request=1000");

//from    #4:
//to      #1000:
string str4 = Regex.Replace(str3, @"#d*", @"#1000");

dicCrash[(String)(read["id"].ToString())] = str4;
```

因为数字的不同而导致的 Crash 不能去重的问题，不仅限于上述这几种情况。我们应该具体问题具体分析，每发现一种新情况，就在程序中增加相应的正则表达式，进行批量替换。

那么接下来，我们只要使用下述 SQL 语句就能取得去除重复的数据了，不受 Crash 信息中数字不同的影响：

```
select distinct page_name, dis_info from CrashDB
order by page_name, dis_info
```

在对 CrashDB 表中的四万笔数据执行这个 SQL 语句后，得到 3000 多笔数据，重复数据大幅减少。

我对上述 C# 程序进行封装，做成一个工具，可以在配置文件中增加新的正则表达式，我将这个工具称为 AnalysisCrash，图形界面如图 5-3 所示。

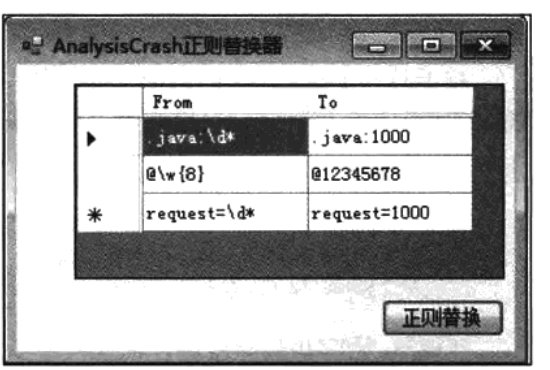


图 5-3　AnalysisCrash

相应的配置文件 RegexRules.xml，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>

<Rules>

    <Rule name="r1" from=".java:\d*" to=".java:1000" />
    <Rule name="r2" from="@\w{8}" to="@12345678" />

    <Rule name="r3" from="request=\d*" to="request=1000" />
</Rules>
```

2. 去除其他情况的重复

我还观察到，有很多Crash信息，它们仅仅是长度的不同，比如说B的Crash信息比A多了一块。相应的解决方案是，对exception_stack从起始位置取150个字符，再进行distinct去重。这样就又能少大量的Crash数据了，SQL语句如下所示：

```
select distinct page_name,
    SUBSTRING(dis_info, 1, 150) from CrashDB
order by page_name, SUBSTRING(dis_info, 1, 150)
```

这样Crash数量就从3000降低到了1200个。

也许你会问我为什么是150，我只能说这是试出来的。如果设置为200，会略显宽松，执行上述语句后，Crash数量会变成1300个；如果设置为100，则又太严格，Crash数量会变成1100个。我们可以根据实际情况动态调整这个值。

不要轻易满足于筛选后的这1200笔数据。这里面还是有很多水分的。纵观去重后的1200笔数据，里面重复的Crash数据还是很多。我们只能根据不同Crash，相应的给出不同的去重方案。

比如说，对于VerifyError这样的Crash，它的page_name字段不是某个Activity页面，而是具有相同的值Application，而对于exception_stack字段则仅仅是类的名称的不同，如下所示：

```
java.lang.VerifyError: Rejecting class
com.company.app.activity.ActivityA
that attempts to sub-class erroneous class
com.company.app.activity.BaseActivity
(declaration of 'com.company.app.activity.ActivityA')
appears in /data/app/com.company.app.ui-2.apk
```

对于这个Crash，我们的解决方案是，只要exception_stack的前38个字符是java.lang.VerifyError: Rejecting class，都视为一个Crash。在执行distinct语句之前，先排除这类Crash。

```
select * into #temp1 from CrashDB

delete from #temp1
where SUBSTRING(dis_info, 1, 38)
    = 'java.lang.VerifyError: Rejecting class'

select distinct page_name,
    SUBSTRING(dis_info, 1, 150) from #temp1
order by page_name, SUBSTRING(dis_info, 1, 150)
```

执行上述语句后，Crash 数据从 1200 降低为 1100。

我们需要不断地增加新的规则，从而进一步优化我们的去重结果。这就需要投入人力砸在上面去做了。很多第三方 Crash 收集平台也是基于这个思路去设计的。

3. 去除同一版本之前的重复

如何确保昨天统计过的 Crash，今天不会再统计？

相应的解决方案是把今天的线上 Crash 放到一个数据表 CrashStore 中，对于第二天的线上 Crash 数据，先到 CrashStore 表中去重，那么剩下来的 Crash 数据就是新的了。

为此我们设计 CrashStore 表结构如图 5-4 所示。

列名	数据类型
categoty_id	int
page_name	nvarchar(255)
sub_crash_desc	nvarchar(500)
sub_crash_length	int
app_version	nvarchar(50)
fix_status	nchar(10)
first_find_date	datetime

图 5-4　CrashStore 表结构

然后编写一个存储过程 UpdateCrashStore，我为每个 SQL 操作都添加了注释，仅供参考：

```
CREATE PROCEDURE [dbo].[UpdateCrashStore]
    @version varchar(30)
AS
BEGIN
  SET NOCOUNT ON;

select * into #temp1 from CrashDB

-- 1. 排除 java.lang.VerifyError 之类 Crash 对去重结果的影响
delete from #temp1
where SUBSTRING(dis_info, 1, 38)
  = 'java.lang.VerifyError: Rejecting class'

-- 1.x 这里可以添加其他排除语句，减少对去重逻辑的干扰

-- 2. 取 dis_info 的前 150 个字符，去重
select distinct page_name,
  SUBSTRING(dis_info, 1, 150) as sub_crash_desc
  into #temp2 from #temp1
  order by page_name, SUBSTRING(dis_info, 1, 150)

-- 3. 在 CrashStore 表中，取出当前版本的之前已经统计过的 Crash
select * into #tempCrashStore from CrashStore
  where app_version=@version

-- 4. 使用 left join 语句，筛选出今天的、未统计过的 Crash
select t.page_name, t.sub_crash_desc
  into #temp4 from #temp2 t
  left join #tempCrashStore c
  on c.page_name = t.page_name
  and c.sub_crash_desc = t.sub_crash_desc
  where c.categoty_id is null

-- 5. 将今天统计的 Crash 放入 CrashStore 表
```

```
insert CrashStore(page_name, sub_crash_desc,
    sub_crash_length, app_version)
  select distinct page_name, sub_crash_desc,
  150, @version from #temp4

END
```

4. 按照 Activity，把 Crash 自动分发到人

这一步不属于去重工作，而是一件锦上添花的工作。我们在给出 Crash 报表后，发现并没有把每个 Crash 落实到具体的开发人员身上。

为此，我们设计 PageOwner 表，用来记录每个 Activity 应该由哪位开发人员负责修复的对应关系，表结构如图 5-5 所示。

表中的数据可以如图 5-6 所示。

列名	数据类型
Activity	nvarchar(100)
Owner	nvarchar(100)

图 5-5 PageOwner 的表结构

Activity	Owner
HomeActivity	张三
UserActivity	李四

图 5-6 PageOwner 中的数据

那么在出报表的时候，与 PageOwner 这个表进行匹配，就能得出每个 Crash 应该谁来负责修复了。

至此，一套完整的 Crash 去重流程就做完了。我们再重新梳理一下上述去重的流程，如图 5-7 所示。

图 5-7 去重流程

本节所介绍的线上 Crash 分析流程，在 Android 发版后每天都要做一遍，把昨天 24 小时内产生的线上 Crash 分析一遍。一般而言，发版后的头 2 天，Crash 数据不太多，因为很多人还没有升级 App 到最新的版本，发版后的第 3 到 5 天，基本就能收集到这个版本 95% 的线上 Crash。再往后，虽然 Crash 数量也很多，但大都重复，再投入时间分析，意义不大。

我们可以把上述过程中的建表、执行 SQL 脚本、执行 C# 程序这些操作串起来，做成自动化执行脚本，这样就能大大节省人力成本了。

5.2.4 线上 Crash 的其他分析工作

对于我而言，上述两节所整理出来的报表基本就能满足我的需求了。但这还远远不够，如果有人力，应该把以下工作也完成：

1）对 Crash 进行归纳，从而知道每类 Crash 发生的次数、涉及的机型、涉及的 Android 系统版本。

我们曾经按照 page_name，dis_info 这两个维度对 Crash 进行了去重，。对这些去重了的 Crash 数据，当我们点击其中一个 Crash 时，应该能够看到有多少种机型、哪些版本的 Android 系统，发生过这类 Crash。

这是个一对多的关系，我们需要根据去重后的 Crash 数据，反向查找每种 Crash 在 CrashDB 表中出现过多少次，以及相应的 Crash 信息。

我们在前面创建了 CrashStore 表，这个表中的 category_id 字段是自增的，相应的，我们要在 CrashDB 这个表也增加 category_id 字段，它们是一对多的关系，这样就做到了点击一种 Crash，能够知道每类 Crash 发生的次数、涉及的机型、涉及的 Android 系统版本。

接下来就是写个 C# 程序，把 CrashDB 表中的 category_id 字段值都反向填充上。

2）目前第三方平台的 Crash 统计工具是即时的，也就是说服务器每收到一个 Crash，就会将其归类，而不是要等到一天结束后才一起进行分析。

此外，我们应该基于线上的 Crash 数据，做一个 Crash 查询平台，开发人员可以根据 App 版本、Crash 发生时间段、机型、Activity 页面等条件来查询相应相应的 Crash。

这个平台还应该提供 Crash 趋势图，它能绘制出一天 24 小时的线上 Crash 趋势图。如果在某个时间段 Crash 数量激增，一定是有重大事情发生，比如 MobileAPI 返回了脏数据。

5.3　本章小结

本章介绍的 Crash 三部曲的第 1 部和第 2 部：异常收集和异常统计。异常收集是基于 App 端的，在发生崩溃的最后一道“关卡”，将崩溃信息发送到服务器。异常统计是基于数据库的，针对于线上每天几千笔崩溃数据，如何自己编写工具将其去重、分类。

在得知线上每天有哪些崩溃后，下一章将介绍针对于每类崩溃的发生原因和解决方案。

第 6 章 Chapter 6

Crash 异常分析

对于一款 App 而言，最重要的莫过于稳定性，没有之一。

Android 之所以存在千奇百怪的 Crash，主要归结于以下几种情况：

1）Android 系统的碎片化。各种硬件厂商都定制自己的 ROM，改写了 Android 系统的很多方法。对于 App 而言，在大多数手机上没有问题，但是到了该厂商的手机系统里，使用到这些方法就会崩溃。当然，也不能排除 ROM 上的方法被改写后存在 bug 的情况。

2）MobileAPI 返回了脏数据。比如说当 MobileAPI 返回空值或空数组时，App 收到数据后就会发生空指针或数组越界的 Crash。有时则是某个字段返回 0，而这个字段作为除数时，也会发生不能除以 0 的 Crash。

3）混淆时没有 Keep 要使用的类或方法，也会发生找不到类或方法的 Crash。

Android 正是因为有这些光怪陆离的 Crash 而显得比 iOS 开发有趣得多。

我们继续聊，每个 Crash 都对应 Android 中的一类 Exception。本章就是要介绍 Android 中的各类 Exception，这些都是我亲身经历过的，其中的大部分都已经修复，当然也有一些 Crash 直到现在仍是不明觉厉。

Crash 信息会因为 App 进行了混淆处理而看不懂具体是在哪个方法哪行代码崩溃的，所以我们需要把每次发版打包时生成的 ProGuardMapping 文件保留下来，然后根据这个文件中方法混淆前后的对应关系，找到发生崩溃所在的原始的类、方法和代码行。

此外，我还发现，有些 Crash 是开发人员在调试的时候发到线上的 Crash 数据库中的。为此，本地开发版本的渠道号，要与发到线上的渠道号区分开。否则，就会误认为是线上 Crash 而白花很多时间去查原因。

提示 Unknown Source

异常信息中经常会出现“方法名”(Unknown Source)的内容。这就加大了我们准确定位 Crash 发生原因的难度。

导致 Unknown Source 的出现有以下两点原因：

1）执行 javac 时丢失了文件名和行号

为此我们在进行 javac 编译时要保留 debug 信息，如下所示：

```
<javac debug="true" debuglevel="source,lines" ……>
```

2）执行混淆时丢失了文件名和行号

为此，我们要在 ProGuard 文件中增加以下语句：

```
-keepattributes SourceFile,LineNumberTable
```

感谢腾讯 Bugly 平台的“精神哥”在审阅本章的时候所提出的宝贵意见，关于 Unknown Source 的更详细介绍，请参见：http://bugly.qq.com/blog/?p=110。

同理，对于测试人员使用的测试包，所使用的渠道号，也要与发到线上的渠道号区分开。

对于每晚跑 Monkey 的包，由此产生的 Crash 信息不应该上传到线上，存到测试机上即可。每天有人去排查 Monkey 日志即可。这样就避免了线上 Crash 数据中有太多的冗余数据。

接下来我们就对 Android 中的 Crash 逐一讲解，共计 84 个，分为 10 大类。

6.1 Java 语法相关的异常

这类 Crash，通常是和 Java 语法有关系。同样的错误，在 Java 项目中也屡见不鲜。所幸的是，这些纯语言相关的异常都有相应的解决方案。

6.1.1 空指针

异常中的关键字：

NullPointException

发生频率：★★★★★

笼统地说，80% 的 Crash 在异常信息中都带有 NullPointException 这样的关键字，散布在各个 Activity 和 Adapter 中，但其实有很多是其他原因导致的。比如说窗体泄露很多时候也表现为 NullPointException。我们这里只讨论几种最简单的几种情况（其他复杂的情况散布

在后续的异常分析中）：

1）方法需要对传入的参数判空后再使用。调用 MobileAPI 的接口时，过于相信返回的数据，一旦使用了空的 JSON 值，App 就有可能崩溃。这类原因导致的 Crash，修复是比较容易的，只要在 MobileAPI 接口返回的数据上增加非空判断或 try-catch 语句即可。

很多 App 开发都使用了 AsyncTask 来调用 MobileAPI 接口并返回数据，在 AsyncTask 的 doInBackground 中，会因为有空指针而崩溃。

2）对于外部接口调用，需要确保返回值中不为空，甚至需要确保执行该接口不会抛出其他异常导致程序退出。比如，页面跳转前后，跳转前没准备好数据，跳转到目标页执行 onCreate 方法时解析传过来的 Intent 时，发现 bundle 这个字典中的某些数据为空，那么使用的时候就崩溃了。

3）在 App 中过多使用全局变量，一旦发生内存回收，这些全局变量会被设置为空，而我们的程序又没有考虑如何处理这种情况。针对于这类原因导致的 Crash，我们要避免使用全局变量，如果万不得已必须要使用全局变量，也要使全局变量支持序列化到本地的机制，一旦我们要使用全局变量而又发现其为空的时候，就从本地反序列化回来。

全局变量这类问题多发生在把 App 切换到后台，过一段时间后再切换到前台，因为要执行所在页面的 onResume 和 onCreate 方法，如果这些方法中有全局变量并且被回收，那么会立刻就崩溃。

此外，即使切换回前台不会崩溃，由于这个页面所使用的全局变量被回收了，那么在页面跳转时，还是会发生崩溃。

关于全局变量的详细介绍，参见第 3 章的 3.5 节“消灭全局变量”。

6.1.2　角标越界

异常中的关键字：

- 关键字 1：IndexOutOfBoundsException
- 关键字 2：StringIndexOutOfBoundsException
- 关键字 3：ArrayIndexOutOfBoundsException

发生频率：★★★

如图 6-1 所示，IndexOutOfBoundsException 是基类。对于字符串截取时发生的越界，会抛出 StringIndexOutOfBoundsException 的异常信息；而对于数组越界，则会抛出 ArrayIndexOutOfBoundsException。

这类 Crash 也是由于程序的不严谨导致的。相应的解决方案是：

❑ 在遍历一个数组 / 集合时，要预判数组 / 集合是否为空，长度是否大于 0。

❑ 在使用数组 / 集合中的元素时，要预判数组 / 集合长度是否有这么长。

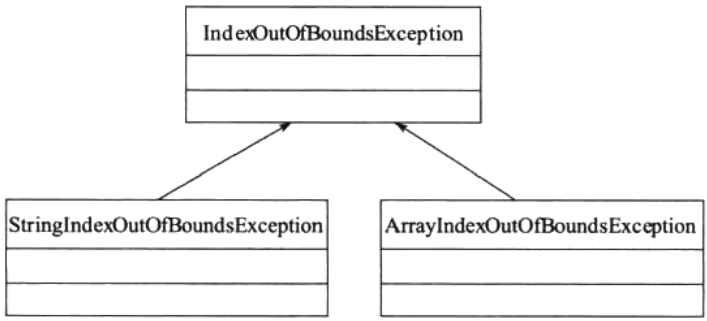


图 6-1 IndexOutOfBoundsException 与其子类的继承关系

字符串也是一种数组，我们经常会使用 subString（start，end）这样的函数，如果 start 或 end 超过了字符串的长度，就会崩溃。解决方案是，每次使用该函数时，都要判断字符串的长度。

ListView 操作不当也会导致 IndexOutOfBoundsException 的异常，请参阅 6.4.3 节的相关介绍。

6.1.3 试图调用一个空对象的方法

异常中的关键字：

Attempt to invoke virtual method on a null object reference

发生频率：★★★

这种 Crash 的产生，是因为在使用一个对象的某个方法时，这个对象为空，就是说没有实例化。比如，我们经常犯的一个错误是，将实例化的语句写在 if-else 的一个分支中，日常开发和测试工作只保证了带有实例化的情况，所以不会崩溃。发版后，没有实例化的分支才会被大量的用户群所点到，于是就崩溃了。

我还经常看见这样的程序，在一个 Activity 中，调用另一个 Activity B 的方法，为此在 B 中建立一个 static 变量。当这个 static 变量被回收时，就会有上述异常。

还有一种可能，那就是推送，点击推送消息，根据事先定好的协议，跳过首页直接进入二级甚至三级页面。这时，二级页面要使用首页某个对象时，这个对象势必为空，那也会引发同样的异常。

6.1.4 类型转换异常

异常中的关键字：

ClassCastException: classA cannot be cast to classB

发生频率：★★★★

这类 Crash 都是由于强制类型转换导致的，如下所示：

```
Object x = new Integer(0);
String str = (String)x;
```

这就会抛出 ClassCastException 的异常了。

解决方案是，使用安全类型转换函数，参见本书第 1 章中 1.6 节介绍的类型安全转换函数。在把字符串转换为整数、小数或布尔类型时，我们要为其指定转换失败时的默认值。否则，就会得到一个空值，放到哪里使用都会崩溃。

6.1.5　数字转换错误

异常中的关键字：

NumberFormatException

发生频率：★★★★

在数据类型转换过程中，如果转换不成功，一般抛出 ClassCastException 的异常。只有一个例外情况，当字符型转换为数字失败时，Android 系统会抛出 NumberFormatException 异常，如下所示：

```
String abc = "123xxx45";
int result = Integer.parseInt(abc);
```

这种情况多发生在服务器返回数据，没有按照约定返回整数而是字符串，客户端必须要事先考虑到这种情况，如果转换失败，必须有默认值而不是直接就崩溃了。

6.1.6　声明数组时长度为 -1

异常中的关键字：

NegativeArraySizeException

发生频率：★★

数组大小为负值异常。当使用负数大小值创建数组时抛出该异常。

我认为程序员不可能犯 int arr = new int[-1]；这样的低级错误，所以我继续试图寻找其他导致这个异常的场景。我在网上找了很久，直到有一天，我发现了下述语句：

```
String[] arg 1 = new String[args.length - 1];
```

当 args 数组中没有元素时，就会出现 int[-1] 的场景。

此外，我还尝试声明 int arr = new int[0]；的语句，发现程序并不会报错，但是这样的语

句声明得到的变量 arr 毫无意义，因为 arr 的长度为 0，arr 只能是一个空数组，不能设置其中的任何一个元素。所以我们在声明数组时，不能出现类似 int[0] 这样的语句。

综上所述，在声明一个数组时，如果数组长度是由另一个变量动态得到的，要保证中括号 [ ] 中的值必须大于 0。

6.1.7 遍历集合同时删除其中元素

异常中的关键字：

ConcurrentModificationException

发生频率：★★

能犯这种错误的人，还是拖出去打八十大板吧，而且要翻过来打的那种。

但凡有点编程常识的程序员都知道在遍历一个集合时不能删除该集合中的元素，如下所示，必然产生这样的崩溃：

```
HashMap<Integer, String> map = new HashMap<Integer, String>();

for (int i = 0; i < 10; i++) {
    map.put(i, "value" + i);
}

for (Map.Entry<Integer, String> entry : map.entrySet()) {
    Integer key = entry.getKey();
    if (key % 2 == 0) {
        map.remove(key);
    }
}
```

该问题的解决方案是，需要再定义一个列表结合 delList，用来保存需要删除的对象，如下所示：

```
HashMap<Integer, String> map = new HashMap<Integer, String>();

for (int i = 0; i < 10; i++) {
    map.put(i, "value" + i);
}

List delList = new ArrayList();

for (Map.Entry<Integer, String> entry : map.entrySet()) {
    Integer key = entry.getKey();
    if (key % 2 == 0) {
        delList.add(key);
    }
}

for (int i = 0; i < delList.size(); i++) {
```

```
        map.remove(delList.get(i));
    }
```

还有另一种产生这种崩溃的情况，那就是在多个线程中删除同一个集合中的元素。

如下列代码所示，vector 是一个集合，我们建立了两个线程，线程 1 对其进行遍历，线程 2 对其进行插入操作。由于这两个线程同时在执行，所以就会产生 ConcurrentModification Exception 的异常了：

```
static ArrayList<Integer> list = new ArrayList<Integer>();

void testScenario2() {
    for (int i = 0; i < 100; i++) {
        list.add(i);
    }

    Thread1 thread1 = new Thread1();
    thread1.start();

    Thread2 thread2 = new Thread2();
    thread2.start();
}

class Thread1 extends Thread {
    public void run() {
        while (true) {
            Iterator<Integer> iterator = list.iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
            }
        }
    }
}

class Thread2 extends Thread {
    public void run() {
        while (true) {
            for (int j = 101; j < 200; j++) {
                list.add(j);
            }
        }
    }
}
```

ArrayList 继承自 AbstractList，这是一个迭代器，所有继承自 AbstractList 的集合类，都是线程不安全的。

相应的解决方案是，将 Vector 换为 CopyOnWriteArrayList，这是一个线程安全的集合类。

6.1.8 比较器使用不当

异常中的关键字：

Comparison method violates its general contract!

发生频率：★★

这个错误是因为 Comparator 的 compare 方法使用姿势不正确导致的。

说起 Comparator，是基于插入排序算法与归并排序算法相结合的产物[⊖]，要比我们日常所使用的冒泡排序算法快很多，但缺点就是不易掌握，于是就产生了这里所讨论的异常。

我们先写一个正确的用法：

```
List<Double> list = new ArrayList<Double>();
list.add(22.1);
list.add(22.1);
list.add(19.7);
list.add(26.3);

Comparator<Double> comparator = new Comparator<Double>() {
    public int compare(Double d1, Double d2) {
        if (d1 < d2) {
            return -1;
        } else if (d1 > d2) {
            return 1;
        } else {
            return 0;
        }
    }
};

Collections.sort(list, comparator);
```

但是，我们经常会偷懒，把这个 compare 方法写成这样：

```
Comparator<Double> comparator = new Comparator<Double>() {
    public int compare(Double d1, Double d2) {
        return p1 > p2 ? 1 : -1;
    }
};
```

这就忽略了 p1 和 p2 的 age 相等的情况，这时应该返回 0。当数组或集合中的元素以某种方式排列的时候，就会报 Comparison method violates its general contract! 的异常了，如下所示[⊜]：

⊖ 关于 Comparator 的算法实现机制，详细信息请参见 http://blog.2baxb.me/993/?utm_source=tuicool

⊜ 关于这个 bug 的详细介绍，网上已经找不到原创，请参见其中一篇转载文章：http://blog.csdn.net/sells2012/article/details/18947849，隐约能查到的是，此文为 HuangWei 所写，相关代码请到 GitHub 下载：https://github.com/Huang-Wei/understanding-timsort-java7。

```
public static void compare() {
    int[] sample = new int[] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            0, -2, 1, 0, -2, 0, 0, 0, 0 };

ArrayList<Integer> list = new ArrayList<Integer>();
for (int i : sample) {
    list.add(i);
}

    Comparator<Integer> comparator = new Comparator<Integer>() {
        public int compare(Integer o1, Integer o2) {
            if (o1 < 02)
                return -1;
            else if (o1 > o2)
                return 1;
            else
                return 0;

        }
    };

    Collections.sort(list, comparator);
}
```

为了预防这类 Crash 的发生，我的解决方案是对每个自定义的比较器进行单元测试，用充足的测试数据来保障逻辑没有问题。参见本书第 8 章中 8.9 节介绍的单元测试。

6.1.9　当除数为 0

异常中的关键字：

java.lang.ArithmeticException: divide by zero

发生频率：★★

当在程序中执行一个除法时，如果除数为 0，就会发生上述崩溃。

我们一般不会直接写出除数为 0 的异常来。这样的 Crash 多发生在第三方控件中，比如说 GifView，这个框架很有名，用于显示 gif 动画。

GifView 这个开源项目有很多变体，但是无论如何，都应该注意其中 movie 的 duration 方法，这个值表示动画持续的时间，在接下来的代码中将会作为除数，如果为 0，就会抛出上述的异常信息了，这时候要将其设置为默认值 1 秒。[⊖]

⊖ 详细内容请参见 http://blog.csdn.net/loongggdroid/article/details/21166563。

6.1.10 不能随便使用的 asList

异常中的关键字：

java.lang.UnsupportedOperationException at
java.util.AbstractList.remove(AbstractList.java:144) at
java.util.AbstractList$Itr.remove(AbstractList.java:360) at
java.util.AbstractCollection.remove(AbstractCollection.java:252) at

发生频率： ★★

这个异常是因为对 asList 方法的理解有误导致。Arrays.asList() 的返回值类型为 java.util.Arrays$ArrayList，而不是 ArrayList。画一个类的继承关系图，如图 6-2 所示。

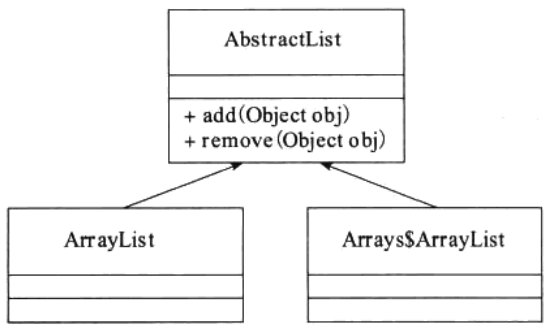


图 6-2 AbstractList 类的继承关系图

从图中看到，AbstractList 这个基类有两个方法 add 和 remove。但是它的两个子类，只有 ArrayList 实现了 add 和 remove 这两个方法，而 Arrays$ArrayList 却没有实现这两个方案，而直接抛出 UnsupportedOperationException 异常。

写一段导致这个异常的代码，如下所示：

```
String str = "1,2,3,4,5";
List<String> test = Arrays.asList(str.split(","));
test.remove("1");
```

相应的解决方案是，将 java.util.Arrays$ArrayList 转换为 ArrayList，如下所示：

```
String str = "1,2,3,4,5";
List<String> list = Arrays.asList(str.split(","));
List arrayList = new ArrayList(list);
arrayList.remove("1");
```

6.1.11 又有类找不到了（一）：ClassNotFoundException

异常中的关键字：

ClassNotFoundException

发生频率： ★★★★

当我们动态加载一个类的时候，如果这个类在运行时找不到，就会抛出这个异常。比如说，Class 会有一个 forName 方法：

```
Class.forName("com.company.package.class");
```

由于类的全名称是字符串形式，这个值极有可能可能是不正确的，那自然就会加载不成功了。类似的方法还有：

- ClassLoader 中的 findSystemClass（“classname”）方法。
- ClassLoader 中的 loadClass（“classname”）方法。

我们在 6.2 节中会介绍导致 ClassNotFoundException 的几种情况，比如说使用 Proguard 会把一些类混淆了，但是 Class.forName 中的参数值并不会改变，那么自然就会找不到类了。

6.1.12　又有类找不到了（二）：NoClassDefFoundError

异常中的关键字：

NoClassDefFoundError

发生频率：★★★★

当我们在 B 类中声明一个 A 类的实例，如下所示：

```
ClassA  obj = new ClassA();
```

但是打包时 B 和 A 分别位于不同的 dex 中，这时如果在 A 所在的 dex 中把 A 类删除了，那么在运行时执行到这句话时就会抛出 NoClassDefFoundError 的异常信息。

通常插件化编程的时候会牵扯出这个异常，因为要使用到 DexClassLoader。也许你的项目中没有用到插件化编程但是也有类似的问题，那么就看一下你所使用的第三方 SDK 吧。

6.2　Activity 相关的异常

Android 四大组件，对于应用类 App 而言，使用最多的是 Activity，偶尔会用到 Service 和 BroadCastReceiver，线上关于 Actvity 的崩溃也是最多的。

对于这些异常，只从字面上分析是远远不够的。它们通常是由外界环境所导致的，比如说找不到一个 Activity 或 Service，有可能是混淆或 dex 拆分不当导致的。

6.2.1　找不到 Activity

异常中的关键字：

android.content.ActivityNotFoundException: No Activity found to handle Intent {…}

发生频率：★★★

出现错误的原因是，URL 不是以 http 开头，代码就会抛出异常：

```
Uri uri = Uri.parse("www.baidu.com");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

这类 Crash 还有一种发生的场景是，当我们要打开 SD 卡上的一个 HTML 页面时，没有为 Intent 指定打开该 HTML 页面所需要的浏览器，如下所示：

```
Intent intent = new Intent(Intent.ACTION_VIEW,
    Uri.parse("file:// sdcard/101.html"));

// 此处指定系统自带浏览器包名和 Activity 名称 .
// intent.setClassName("com.android.browser",
// "com.android.browser.BrowserActivity");

startActivity(intent);
```

就是因为指定浏览器的语句被注释了，所以就崩溃了。我最初想重现这个异常的时候，因为手机上装了爱奇艺 App，所以即使没有指定系统自带的浏览器，也会弹出爱奇艺的播放器。只有卸载了爱奇艺 App，才会复现该问题。

还有一个原因，如果是调用百度地图的 openBaiduMapNavi 方法导致的 Crash，有可能是手机没有安装百度地图的客户端，而这个方法就是要打开这个客户端。解决方案是判断其是否安装了，没有的话就提示用户有问题要么就干脆不显示。

6.2.2 不能实例化 Activity

异常中的关键字：

java.lang.RuntimeException: Unable to instantiate activity ComponentInfo

发生频率： ★★★★

这种 Crash，通常是因为没有在 AndroidManifest.xml 清单中注册该 activity，或者在创建完 activity 后，修改了包名或者 activity 的类名，而配置清单中没有修改，造成不能实例化。

如果还不能解决问题，有可能是系统处于异常状态（关机，内存不足）等，导致部件初始化失败。

6.2.3 找不到 Service

异常中的关键字：

java.lang.RuntimeException: Unable to instantiate receiver

发生频率： ★★

对于应用类 App 而言，不可能开发期间没有问题，而发布到线上却发现上述的崩溃，所

以我们接下来的讨论也基于此。对于 Manifest.xml 文件中写错了的类似问题我们就不研究了。

首先检查代码中是否有 Class.forName("class1") 这样的语句。对于此，ProGuard 会将 class1 混淆，从而就是找不到 class1 这个类。

6.2.4　不能启动 BroadcastReceiver

异常中的关键字：

Unable to start receiver

发生频率：★★

在推送的时候，会和 App 事先定好协议，点击推送消息就能跳过首页直接进入二级页面，如下所示，我们要在一个 BroadcastReceiver 中编写如下代码：

```
Intent intent = new Intent(context, S13Activity.class);
intent.putExtra(bundle);
intent.setFlags(intent.FLAG_ACTIVITY_NEW_TASK);
context.startActivity(intent);
```

使用 Activity 以外的 content 来 startActivity，比如 BroadcastReceiver，就必须指定为 Intent.FLAG_ACTIVITY_NEW_TASK，否则就会抛出上述异常信息。

此外，还有类似的一个异常，如下所示：

```
Caused by: android.util.AndroidRuntimeException:
Calling startActivity() from outside of an Activity context
requires the FLAG_ACTIVITY_NEW_TASK fiag.
Zs this really what you want?
```

众所周知，Context 中有一个 startActivity 方法，Activity 继承自 Context，重载了 startActivity 方法。如果使用 Activity 的 startActivity 方法，不会有任何限制，而如果使用 Context 的 startActivity 方法的话，就需要开启一个新的 task，遇到上面那个异常的，都是因为使用了 Context 的 startActivity 方法。解决办法是，加一个 fiag：

```
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

这样就可以在新的 task 里面启动这个 Activity 了。[⊖]

6.2.5　startActivityForResult 不能回传

异常中的关键字：

Failure delivering result ResultInfo{who=null, request=0, result=-1

发生频率：★★★

⊖ 关于这个 Crash 的更多信息请参见：http://www.it165.net/pro/html/201406/15547.html。

这类问题是 startActivityForResult 导致的。就是说传回来的 key 是 A，但是却按照 B 这个 key 来取值。如下所示：

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    switch (resultCode) {
    case 1:
        Bundle bundle = data.getExtras();
        String number = bundle.getString("number");
        if (!("".equals(number)))
            textview1.setText(number);
        break;
    default:
        break;
    }
}
```

我们看到，number 这个字符串为 null 时，在执行 equal 语法时就会崩溃。其实是空指针导致的，但是表现为 Failure delivering result ResultInfo 的异常。

6.2.6 猴急的 Fragment

异常中的关键字：

Fragment not attached to Activity

发生频率：★★★

发生这个异常，是因为 Fragment 在还没有 Attach 到 Activity 时，调用了诸如 getResource() 这样的方法：

```
getResources().getString(R.string.app_name);
```

相应的解决方案是，在获取资源前先使用 isAdded 方法进行判断，如下所示：

```
if(isAdded()){
    getResources().getString(R.string.app_name);
}
```

isAdd 方法是 Android 系统提供的，它只有在 Fragment 被添加到所属的 Activity 后才会返回 true。

6.3 序列化相关的异常

Android 中的序列化分两种，一种是原始的 Serializable，另一种是 Android 为了提升性

能而量身打造的 Parcelable。接下来将介绍序列化不当导致的异常。

6.3.1　实体对象不支持序列化

异常中的关键字：

Parcelable encountered IOException writing serializable object (name = xxx)……

发生频率：★★★

看下面这个实体类，它看上去是支持序列化的：

```
public class UserInfo implements Serializable {
    private static final long serialVersionUID = 1L;
    private String username;
    private CreditCard creditCard;

    public UserInfo(){

    }
```

看其中的 CreditCard 实体，它的定义如下：

```
public class CreditCard {
    public String cardNO;
```

也就是说，CreditCard 类不支持序列化。那么，当 UserInfo 对象中 CreditCard 属性的值为空时，没有任何问题；而一旦 CreditCard 属性值不为空，那么 UserInfo 在序列化的时候，就会因为这个属性不能序列化而崩溃。

JSONObject 和 JSONArray 不支持序列化

对于 JSONObject 和 JSONArray 这样的类型，也是不支持序列化的，所以实体中一旦有这样的属性，必然崩溃。

6.3.2　序列化时未指定 ClassLoader

异常中的关键字：

BadParcelableException: ClassNotFoundException when unmarshalling……

发生频率：★★

在使用 Parcelable 机制的时候，会遇到上述异常信息。

比如说下面这个序列号类 MyParcelable，有个自定义类型 ClassA 的属性 a：

```
public class MyParcelable implements Parcelable {
    private String mStr;
    private ClassA a;

    ..... //省略若干语句

    private MyParcelable(Parcel in) {
        mStr = in.readString();
        a = in.readParcelable(null);
    }
}
```

崩溃出在最后一句上，对 a 的反序列化上：

```
a = in.readParcelable(null);
```

当把它改为下面这样，就不会再崩溃了：

```
a = in.readParcelable(ClassA.class.getClassLoader());
```

提示 ClassLoader 的概念

当 ClassLoader 为空时，系统会采取默认的 ClassLoader。

Android 有两种不同的 ClassLoader：framework ClassLoader 和 apk ClassLoader，其中 framework ClassLoader 知道怎么加载 Android 系统内部的类；apk ClassLoader 知道怎么加载我们自己写的类，也知道怎么加载 Android 系统内部的类。

在 App 刚启动时，默认 ClassLoader 是 apk ClassLoader，但在系统内存不足应用被系统回收会再次启动，这个默认 ClassLoader 会变为 framework ClassLoader，所以对于我们自己的类会报 ClassNotFoundException。

6.3.3 反序列化时发现类找不到：被 ProGuard 混淆导致的崩溃

异常中的关键字：

Parcelable encountered ClassNotFoundException reading a Serializable object ……

发生频率：★★

在反序列化的时候，发现有个类找不到。一般而言，这样的崩溃，在开发调试期间就会暴露出来。但为什么开发期间没事发到线上就出问题了呢？ StackOverfiow 上有个哥们契而不舍的花了 2 年时间查这个问题，最后发现是 ProGuard 导致的。

ProGuard 对于 Class.forName（className）中的 class 是无能为力的，它会将这个 class 混淆得面目全非，于是在反序列化这个类的时候却发现找不到这个类了，自然就会抛出这种

异常信息了。相应的解决方案就是，在 ProGuard 文件中 keep 这个类。[⊖]

6.3.4　反序列化时发现类找不到：传入畸形数据

异常中的关键字：

Parcelable encountered ClassNotFoundException reading a Serializable object（name = 某个类名称）

发生频率：★★

这是一个最近发现的安全漏洞。

由于在 App 中使用了 getSerializableExtra() 的 API，App 开发人员没有对传入的数据做异常判断，别有企图的人可以通过传入畸形数据，导致本地拒绝服务。

例如传入简单类型，比如 Integer，就会抛出类型转换异常 ClassCastException。

而当传入自定义的可序列化对象时，就会抛出上述带有 ClassNotFoundException 的异常信息了。[⊜]

6.3.5　反序列化时出错

异常中的关键字：

Could not read input channel file descriptors from parcel ……

发生频率：★★★

出现这个异常，一般是因为 Intent 传递的数据太大了，貌似大于 1MB 就会崩溃。

此外，网上也有人说是因为 FileDescripter 太多而且没有关闭，或 looper 太多没有退出导致的，我没有验证过，仅供参考。

6.4　列表相关的异常

有 Adapter 在的地方，就有 ListView，就有因此而产生的异常。这些异常基本是因为下拉列表刷新数据时处理不当导致的。这主要是 Android 本身没有提供标准的下拉刷新数据的列表控件，而网上千奇百怪的下拉刷新控件又都有这样那样的缺陷。封装得再完善的下拉列表控件，也只能确保在大部分机型上工作良好。

⊖ 详细信息请参见 http://stackoverfiow.com/questions/6014806/android-classnotfoundexception-when-passing-serializable-object-to-activity。

⊜ 这个安全漏洞由 360 近期发现，参见 http://blogs.360.cn/360mobile/2015/01/06/android-app 通用型拒绝服务漏洞分析报告 /。

6.4.1 Adapter 数据源变化但是没通知 ListView

异常中的关键字：

The content of the adapter has changed but ListView did not receive a notification. Make sure the content of your adapter is not modified from a background thread, but only from the UI thread.

发生频率：★★★★★

上述异常信息的大体意思是，adapter 的内容变化了，但是相应的 ListView 并不知情。请保证 adapter 的数据在主线程中进行更改！

首先，一种极端的解决方案是，每次设置 adapter 中的集合数据时，都要将其 clone 一份，而不是直接传递一个集合过来。但是这样会比较消耗性能。

其次，要确保每次在 Activity 中设置 adapter 的值，而不是在后台线程，有以下几个办法：

1）调用 Activity 的 runOnUiThread() 方法，如下所示：

```
private class OnClickListenerImpl
    implements View.OnClickListener {
    @Override
    public void onClick(View arg0){
        new Thread(){
            public void run(){
                MainActivity.this.runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        textView1.setText("Hello World!");
                    }
                });
            }
        }.start();
    }
}
```

2）调用 Handler，通知主线程修改 adapter。

3）使用 AsyncTask 也是一个不错的选择，虽然它也有很多缺陷。

最后，无论何时何地，只要修改了 adapter 中集合数据的值（比如设置一个集合数据、加一笔数据、清空集合数据），就要马上调用 notifyDataSetChanged 方法，以确保列表同步更新。

这个异常在 Android 技术圈儿里可算是大名鼎鼎了，基本上所有的 App 应用都存在这样的崩溃。

6.4.2　ListView 滚动时点击刷新按钮后崩溃

异常中的关键字：

java.lang.IndexOutOfBoundsException: Invalid index 30, size is 1 at java.util.ArrayList.throwIndexOutOfBoundsException(ArrayList.java:251) at java.util.ArrayList.get(ArrayList.java:304) at android.widget.HeaderViewListAdapter.getView(HeaderViewListAdapter.java:225)

发生频率：★★★

Listview 滚动的时候，表示它已经获取了 adapter 的 getCounts()，可能是 30，也可能更大。回调用 getView()，这个时候将数据 clear 掉了。当然会报 IndexOutOfBoundsException: Invalid index 30，size is 1。这个 1 是那个 header，因为我们使用的是 HeaderViewList Adapter。

这种 Crash 的解决方案是，Listview 滚动的时候，将刷新按钮设置为不可点击，如下所示：

```
public void refresh() {
    startLocation();
    pageNo = 0;
    hasMore = true;
    dataList.clear();
    moreBtn.setVisibility(View.GONE);
    loadFirstPageData();
}
```

6.4.3　AbsListView 的 obtainView 返回空指针

异常中的关键字：

java.lang.NullPointerException at android.widget.AbsListView. obtain View (AbsListView.java:1521) at android.widget. ListView.makeAndAddView

发生频率：★★★

导致空指针的罪魁祸首是 AbsListView 的 obtainView 方法获取不到 View，究其原因是 getView 方法在某些时候返回 null。

解决方案很简单，getView 的第二个参数 convertView 是不会为 null 的，在 getView 返回值的时候，判断一下是否为 null，如果为 null，则返回 convertView。

6.4.4 Adapter 数据源变化但是没调用 notifyDataSetChanged

异常中的关键字：

The application's PagerAdapter changed the adapter's contents without calling PagerAdapter#notifyDataSetChanged

发生频率：★★

PagerAdapter 对于 notifyDataSetChanged() 和 getCount() 的执行顺序是非常严格的，系统跟踪 count 的值，如果这个值和 getCount 返回的值不一致，就会抛出这个异常。所以为了保证 getCount 总是返回一个正确的值，那么在初始化 ViewPager 时，应先给 adapter 初始化内容后再将该 adapter 传给 ViewPager，如果不这样处理，在更新 adapter 的内容后，应该调用一下 Adapter 的 notifyDataSetChanged 方法。

6.5 窗体相关的异常

这种 Crash 很有名，原因基本都是在执行 dismiss 方法销毁对话框的时候，Activity 已经不再存在。但是随着场景的不同，抛出的异常信息却又大不相同。本节我还会顺带讲一下在非主线程操作 UI 导致的异常。

6.5.1 窗口句柄泄露

异常中的关键字：

android.view.WidnowLeaded: Activity xxx has leaked window com.android.internal.policy.impl.PhoneWindow$DecorView{xxxx} that was originally added here.

发生频率：★★

我们试着写这样一段代码，来重现这个异常：

```
Dialog dialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s1_scenario1);

        AlertDialog.Builder info = new AlertDialog.Builder(this);
        info.setTitle("Dialog").setPositiveButton("OK", null)
                .setMessage("This is a Dialog");
```

```
        dialog = info.show();

        finish();
    }
```

以上代码在运行时必然崩溃，这是因为，最后 finish 语句销毁了当前 Activity，但是在它基础上创建的 AlertDialog 对话框还在，窗口句柄泄露，未能及时销毁。

finish() 语句是我故意写的，是为了重现这个异常。现实中当然不会这么写代码，往往是因为我们在非主线程中的某些操作不当而产生了一个严重的异常，从而强制关闭当前 Activity。而在关闭的同时，却没能及时调用 dismiss 来解除对 ProgressDialog 等的引用，从而系统抛出了上述崩溃信息。

可以再写一个 Demo，来模拟 Activity 被销毁的情景：

```
Dialog dialog;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s1_scenario2);

        AlertDialog.Builder info = new AlertDialog.Builder(this);
        info.setTitle("Dialog").setPositiveButton("OK", null)
                .setMessage("This is a Dialog");
        dialog = info.show();

        findViewById(R.id.btnStartThread).setOnClickListener(
            new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    new Thread(new Runnable() {
                        @Override
                        public void run() {
                            try {
                                Thread.sleep(10000);
                            } catch (InterruptedException e) {
                                e.printStackTrace();
                            }

                            dialog.dismiss();
                        }
                    }).start();
                }
            });
    }
```

同时，要设置这个 Activity 支持横竖屏旋转，这时就会产生崩溃信息了。

相应的解决办法是，重写 Activity 的 onDestroy 方法，在方法中调用 dismiss 来解除对 ProgressDialog 等的引用：

```
@Override
    public void onDestroy() {
```

```
        super.onDestroy();

        // 成败就在这句话，注释了就会 Crash
        dialog.dismiss();
    }
```

6.5.2 View not attached to window manager

异常中的关键字：

java.lang.IllegalArgumentException: View not attached to window manager at
android.view.WindowManagerImpl.findViewLocked(WindowManagerImpl.java:356) at
android.view.WindowManagerImpl.removeView(WindowManagerImpl.java:201) at
android.view.Window$LocalWindowManager.removeView(Window.java:400) at
android.app.Dialog.dismissDialog(Dialog.java:268) at
android.app.Dialog.access$000(Dialog.java:69) at
android.app.Dialog$1.run(Dialog.java:103) at
android.app.Dialog.dismiss(Dialog.java:252)

发生频率： ★★★★★

发生这类 Exception 的场景是，有一个费时的线程任务，在任务开始的时候显示一个对话框，然后当任务完成了再销毁对话框，在此期间如果 Activity 因为某种原因被杀掉且又重新启动了，那么当 Dialog 调用 dismiss 方法的时候 WindowManager 检查发现 Dialog 所属的 Activity 已经不存在了，所以会报 View not attached to window manager。[⊖]

要想避免此类 Exception，就要正确的使用对话框，也要正确的使用线程，有以下几点需要注意。

1）正确使用对话框。不要在非 UI 线程中使用对话框创建，显示和取消对话框。

那么对于异步操作显示对话框怎么办呢？ Activity 都有相应的操作对话框的回调，比如：

- onCreateDialog()
- showDialog()
- dimissDialog()
- removeDialog()

以上这些都是 Activity 的方法，因此使用起来更方便，也不用显示创建和操控 Dialog 对象，一切都由框架操控，相对来说比较安全。

2）一定要让对话框对象在 Activity 的可控制范围之内和生命周期之内。比如对话框一定要是 Activity 的成员变量，并且在让对话框变量活跃在 Activity 的 onCreate() 和 onDestroy() 这两个方法之间。

写一个引发此 Crash 的例子：

⊖ 有关这个 Crash 的更详细描述，请参见 http://blog.csdn.net/yihongyuelan/article/details/9829313。

```
private ProgressDialog mProgressDialog;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s10);

        mProgressDialog = new ProgressDialog(this);
        mProgressDialog.show();

        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                mProgressDialog.dismiss();
            }
        }, 1000);

        finish();
    }
```

后来我在网上看到另一种完美的解决方案[⊖]：从 ProgressDialog 中派生出 SafeProgressDialog 子类，通过覆写 dismiss 方法，在 ProgressDialog.dismiss 方法执行之前判断 Activity 是否存在。

```
class SafeProgressDialog extends ProgressDialog {

    Activity mParentActivity;

    public SafeProgressDialog(Context context) {
        super(context);
        mParentActivity = (Activity) context;
    }

    @Override
    public void dismiss() {
        if (mParentActivity != null
            && !mParentActivity.isFinishing()) {
            super.dismiss();
        }
    }
}
```

6.5.3　窗体在不恰当的时候获取了焦点

异常中的关键字：

java.lang.NullPointerException:android.widget.PopupWindow$PopupViewCo ntainer.dispatchKeyEvent

发生频率：★★

⊖ 该解决方案摘自码农场的这篇文章：http://www.hankcs.com/program/mobiledev/solution-java-lang-illegalargumentexception-view-not-attached-to-window-manager.html。

这个问题是因为在 PopupWindow 显示之前，就把焦点赋予了它，结果当然会 Crash 了。

这类问题只在 Android 2.3 版本才会偶然出现，我看到 Android 系统 4.0 的源码修改了方法，在底层对这个问题进行了规避。[⊖]

但是对于 2.3 的 Android 系统，我们还是要进行兼容。相应的解决方法是，在创建 PopupWindow 的时候不立即调用 setFocusable(true)，而是在 showAtLocation 后再调用 setFocusable(true)，同时，在调用 dismiss 的时候，调用 setFocusable(false)。[⊜]


注意 PopupWindow 调用 setFocusable(true) 是为了让它里面的控件能够实现监听事件。

6.5.4 token null is not for an application

异常中的关键字：

android.view.WindowManager$BadTokenException: Unable to add window -- token null is not for an application

发生频率：★★★

在实现 Android 浮窗时，有时会报这个异常，根据以往的经验，出现这问题一般是我们的 Context 不正确。以下代码会报这个异常：

```
new AlertDialog.Builder(getApplicationContext())
    .setIcon(android.R.drawable.ic_dialog_alert)
    .setTitle("Warnning")
    .setMessage("Hello world!")
    .show();
```

问题出在 AlertDialog.Builder（mcontext）这句话，所接受的参数不能是 getApplicationContext() 获得的 Context，而应该是 Activity 实例，因为只有一个 Activity 才能添加一个窗体，如下所示：

```
new AlertDialog.Builder(S4Activity.this)
    .setIcon(android.R.drawable.ic_dialog_alert)
    .setTitle("Warnning")
    .setMessage("Hello world!")
    .show();
```

⊖ 参考 http://stackoverfiow.com/questions/7768728/popupwindow-crash-on-dispatch-event 和 http://www.eoeandroid.com/thread-109193-1-1.html 这两篇文章。

⊜ 可参考 http://www.cnblogs.com/loulijun/p/3267958.html。

6.5.5　permission denied for this window type

异常中的关键字：

Android.view.WindowManager$BadTokenException: Unable to add window android.view.ViewRootImpl$W@411da608 -- permission denied for this window type

发生频率：★★★

在使用 WindowManager.LayoutParams.TYPE_SYSTEM_ALERT 涉及 window type 权限问题。

这种错误多发生在使用 WindowManager 自定义弹出框时，没有设置权限。

相应的解决方案是，在 AndroidManifest.xml 配置文件中添加以下两个 uses-permission：

```
<!-- 显示系统窗口权限 -->
<uses-permission
    android:name="android.permission.SYSTEM_ALERT_WINDOW" />
<!-- 在屏幕最顶部显示 addview -->
<uses-permission
    android:name="android.permission.SYSTEM_OVERLAY_WINDOW" />
```

前者允许应用使用 TYPE_SYSTEM_ALERT 来打开窗口，并将窗口显示于其他应用的顶端；后者允许使用窗体覆盖在 window 上。

6.5.6　is your activity running

异常中的关键字：

android.view.WindowManager$BadTokenException: Unable to add window -- token android.app.LocalActivityManager$LocalActivityRecord@45a58ee0 is not valid; is your activity running?

发生频率：★★★★

当我回来，你已不在。说的就是这个 Crash。

这种 Crash 与弹出框 Dialog 密切相关，是由于 Activity A 依附于另一个 Activity B 的，当被依附的 Activity B 产生错误的时候，Activity A 因为没有了靠山而产生错误（或者是调用了一个已经被 finish() 的 Activity）。

比如，在 onCreate 方法中，想要弹出 PopupWindow，如下所示：

```
public class S6CrashActivity extends Activity {

    @Override
```

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s6_crash);

        PopupWindow popupWindow = new PopupWindow(
            getLayoutInfiater().infiate(
            R.layout.activity_s6_crash, null),
            ViewGroup.LayoutParams.WRAP_CONTENT,
            ViewGroup.LayoutParams.WRAP_CONTENT);

        popupWindow.showAtLocation(
            findViewById(R.id.btnScenario1),
            Gravity.CENTER, 0, 0);
        popupWindow.update();
    }
}
```

我们看一下 PopupWindow 的 showAtLocation 方法：

```
void android.widget.PopupWindow.showAtLocation(
    View parent, int gravity, int x, int y)
```

当参数 parent 为空时，就会报上述的错误，说 token 为空了，无效了，由于 popupwindow 要依附于一个 activity，而 activity 的 onCreate() 还没执行完，那么肯定会出错了。

因此，我们要做的就是让这个 showAtLocation 的调用再晚一点，这里使用 handler 来解决这个问题，如下所示：

```
public class S6CrashFixActivity extends Activity {
    private PopupWindow popupWindow;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        popupWindow = new PopupWindow(getLayoutInfiater().infiate(
            R.layout.activity_s6, null),
            WindowManager.LayoutParams.WRAP_CONTENT,
            WindowManager.LayoutParams.WRAP_CONTENT);

        new Thread() {
            public void run() {
                try {
                    handler.sendEmptyMessageDelayed(0, 1000);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }.start();
```

```
    }

    private Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
            case 1000:
                popupWindow.showAtLocation(
            findViewById(R.id.btnScenario1),
                        Gravity.CENTER | Gravity.CENTER, 0, 0);
                popupWindow.update();
            }

            super.handleMessage(msg);
        }
    };
}
```

6.5.7　添加窗体失败

异常中的关键字：

java.lang.RuntimeException: Adding window failed at
android.view.ViewRootImpl.setView(ViewRootImpl.java:511) at
android.view.WindowManagerImpl.addView(WindowManagerImpl.java:301) at
android.view.WindowManagerImpl.addView(WindowManagerImpl.java:215) at ……

发生频率： ★

这个 Crash 我不能复现，只能在线上看到异常信息。不知道发生的原因，也暂时没有解决方案。

检查 Android 系统源码，这个 Crash 是在 ViewRoot 的 setView 方法中捕获到的，如下所示：

```
try {
    res = sWindowSession.add(mWindow, mWindowAttributes,
          getHostVisibility(), mAttachInfo.mContentInsets);
} catch(RemoteException ex) {
    mAdded = false;
    mView = null;
    mAttachInfo.mRootView = null;
    unscheduleTraversals();
    throw new RuntimeException("Adding windows failed", ex);
}
```

考虑到窗体类 Crash 的完整性，我没有把这个 Crash 归类到 6.9 不明觉厉中。还请知道其中缘由的朋友不吝赐教。

6.5.8 AlertDialog.resolveDialogTheme

异常中的关键字：

java.lang.NullPointerException at
android.app.AlertDialog.resolveDialogTheme(AlertDialog.java:142) at
android.app.AlertDialog$Builder.<init>(AlertDialog.java:359) at
com.radzik.devadmin.MainActivity$5.onClick(MainActivity.java:140) at
android.view.View.performClick(View.java:4084)……

发生频率： ★★★

这是一个很有趣的异常。我在重现 is your activity running 这个异常时，阴差阳错发现了这个新的异常，上网一查，这类 Crash 发生次数还是蛮多的。

场景 1：在 B 页面写了一个 show 方法，控制 AlertDialog.Builder 的弹出和隐藏。在 A 页面却要调用 B 页面的 show 方法，于是就崩溃了，代码如下所示：

```
public class S8Activity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s8);

        Button btnCrash1 = (Button) findViewById(R.id.btnCrash1);
        btnCrash1.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                AnotherS8Activity s8 = new AnotherS8Activity();
                s8.show();
            }
        });
    }
}


public class AnotherS8Activity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    public void show() {
        AlertDialog.Builder dialog = new AlertDialog.Builder(
                AnotherS8Activity.this);
        dialog.setTitle("Test");
        dialog.setMessage("Hello World");
```

```
        dialog.setPositiveButton("OK",
                new DialogInterface.OnClickListener() {

            @Override
            public void onClick(DialogInterface dialog,
                    int which) {
                dialog.cancel();

            }
        });
        dialog.show();
    }
}
```

这种崩溃的解决方案有以下几种：

- 最简单的解决方案，就是把 AnotherActivity 中的 show 方法，复制到 S8Activity 中。
- 也可以把这个 show 方法放在 BaseActivity 中。
- 创建一个单独的类，把 AnotherActivity 中的 show 方法转移过去，只要传递正确的 context 参数即可。

场景 2：在 TabActivity 中切换 Tab 时，容易产生这个 Crash。这是因为，在 new 对话框的时候，参数 content 指定成了 this，即指向当前子 Activity 的 content。但子 Activity 是动态创建的，不能保证一直存在。其父 Activity 的 content 则是稳定存在的，所以将 this 替换为 getParent() 即可，如下代码所示：

```
@Override
public void onTabChanged(String tagString) {
    if (tagString.equals("One")) {
        myMenuSettingTag = 1;
        ProgressDialog dialog = ProgressDialog.show(
                getParent() "提示",
                "正在获取数据，请稍等_1", true, true);
    }
    if (tagString.equals("Two")) {
        myMenuSettingTag = 2;
        ProgressDialog dialog = ProgressDialog.show(
                S8CrashFixActivity.this, "提示",
                "正在获取数据，请稍等_2", true, true);
    }
    if (tagString.equals("Three")) {
        myMenuSettingTag = 3;
        ProgressDialog dialog = ProgressDialog.show(
                S8CrashFixActivity.this, "提示",
                "正在获取数据，请稍等_3", true, true);
    }
    if (myMenu != null) {
        onCreateOptionsMenu(myMenu);
    }
}
```

6.5.9 The specified child already has a parent

异常中的关键字：

The specified child already has a parent. You must call removeView() on the child's parent first.

发生频率：★★★

这个异常，我们从字面上就能理解。在使用儿子的时候，要先调用其父亲的 removeView 方法，解除父子关系。[⊖]

我们在一个 Activity 中加载 layout，一般这样写：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_s9);
```

但殊不知，换个写法也能达到同样的效果：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    LayoutInfiater infiater = (LayoutInfiater)
            getSystemService(LAYOUT_INFLATER_SERVICE);
    LinearLayout parent = (LinearLayout) infiater.infiate(
            R.layout.activity_s9_crash, null);
    setContentView(parent);
}
```

我们尝试着改写 setContent 方法的内容，从 layout 布局中抓取它的子控件 ImageView，当我们把 ImageView 放到 setContent 方法中时，就会报上述的错误了：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    LayoutInfiater infiater = (LayoutInfiater)
            getSystemService(LAYOUT_INFLATER_SERVICE);
    LinearLayout parent = (LinearLayout) infiater.infiate(
            R.layout.activity_s9_crash, null);

    ImageView child = (ImageView)parent.findViewById(
            R.id.imageView1);
    setContentView(child);
}
```

⊖ 关于这个异常的分析，还有一篇文章，http://blog.csdn.net/lissdy/article/details/8453433，我不能复现，仅供参考。

这是因为 ImageView 是其所在 layout 的儿子，它必须跟它的父亲（parent）共存亡，除非我们使用 removeView 先把它从其父亲中移除，如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    LayoutInfiater infiater = (LayoutInfiater)
            getSystemService(LAYOUT_INFLATER_SERVICE);
    LinearLayout parent = (LinearLayout) infiater.infiate(
            R.layout.activity_s9_crash, null);
    ImageView child = (ImageView) parent.findViewById(
            R.id.imageView1);

    parent.removeView(child);
    setContentView(child);
}
```

6.5.10　子线程不能修改 UI

异常中的关键字：

android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.……

发生频率：★★★★★

从字面上翻译是，只有原始创建这个视图层次（view hierarchy）的线程才能修改它的视图（view）。也就是说必须在程序的主线程（UI）线程中更新界面显示的工作。

话虽如此，但是我写了一个 Demo，试图在子线程中更新 TextView 中的值，如下所示：

```
public class Scenario1Activity extends Activity {
    TextView mLoadingText;
    Button btnStartThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s11_scenario1);

        mLoadingText = (TextView) findViewById(R.id.textView1);
        btnStartThread = (Button) findViewById(R.id.btnStartThread);

        new Thread(new Runnable() {
            @Override
            public void run() {
                mLoadingText.setText("hello world");
            }
        }).start();
    }
}
```

但是奇迹出现了，居然能运行良好，不会有崩溃。这不由得使我对之前从书本上看到的概念产生了怀疑，不是说在子线程操作 UI 就会崩溃吗？

后来我加了 1 秒的等待时间，然后再修改 TextView 上的值，这个 Crash 就能稳定复现了（如果不能复现就把时间拉长到 2 ~ 5 秒），代码如下所示：

```
public class Scenario2Activity extends Activity {
    TextView mLoadingText;
    Button btnStartThread;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_s11_scenario2);

        mLoadingText = (TextView) findViewById(R.id.textView1);
        btnStartThread = (Button) findViewById(R.id.btnStartThread);

        // 在 onCreate 方法中执行不会 Crash
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // 刷新页面的文字
                mLoadingText.setText("hello world");
            }
        }).start();
    }
}
```

继续探索，在按钮点击事件中重复刚才的试验，Crash 稳定重现：

```
btnStartThread.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 刷新页面的文字
                mLoadingText.setText("hello");
            }
        }).start();
    }
});
```

于是我重新检查了 Android 的定义，发现是自己对这句话有理解上的误区：“不建议在子线程中更新 UI，会因此而产生不可预知的错误。”

这就是多线程编程，有时候你运行若干次，结果正确，并不表明你的逻辑就是对的。我们一定要遵循代码的规范，保持清晰的思维。

接下来解释一下在 onCreate 方法中操作 UI 为什么有时候不崩溃？就像前面所说，一定要等一会儿才会出现崩溃，肯定是这段时间内某种检查机制还没起作用，晚于后续对 UI 的操作。检查 Android 源码，这个方法是 viewRoot 的 requestLayout()。只有在 requestLayout 方法的子方法 checkThread 中，才会抛出这个异常。

```
public void requestLayout() {
    checkThread();
    mLayoutRequested = true;
    scheduleTraversals();
}

void checkThread() {
    if(mThread != Thread.currentThread()) {
        throw new CalledFromWrongThreadException(
        "Only the original thread that created
        a view hierarchy can touch its views.");
    }
}
```

由此而推测，在 onCreate 的时候，是 requestLayout 方法没有执行——layout 布局文件还没有创建完成，导致我们可以在 onCreate 方法内在其他子线程中操作 UI。

问题查出来了，接下来是如何正确解决问题，因为有时会碰到在非主 UI 线程更新视图的需要。这个时候我们有两种处理的方式。一种是 Handler，另一种是 Activity 中的 runOnUiThread(Runnable) 方法。

方法 1：使用 Handler。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_s11_scenario4);

    mLoadhandler = new LoadHandler();

    mLoadingText = (TextView) findViewById(R.id.textView1);
    btnStartThread = (Button) findViewById(R.id.btnStartThread);

    btnStartThread.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    mLoadhandler.sendEmptyMessage(101);
                }
            }).start();
        }
    });
}
```

```
// 主线程中的 handler
class LoadHandler extends Handler {
    // 接受子线程传递的消息机制
    @Override
    public void handleMessage(Message msg) {
        super.handleMessage(msg);
        int what = msg.what;

        switch (what) {
        case 101: {
            // 刷新页面的文字
            mLoadingText.setText("test");
            break;
        }
        }
    }
}
```

方法 2：利用 Activity 的 runOnUiThread 方法把更新 UI 的代码创建在 Runnable 中，这样 Runnable 对像就能在 UI 程序中被调用。如果当前线程是 UI 线程，那么行动是立即执行。如果当前线程不是 UI 线程，操作是发布到事件队列的线程中。

```
public void onClick(View v) {
    runOnUiThread(new Runnable() {
        public void run() {
            // 刷新页面的文字
            mLoadingText.setText("test");
        }
    });
}
```

方法 3：使用 AsyncTask。

```
private class MyTask extends AsyncTask<Void, Void, Void> {

    @Override
    protected Void doInBackground(Void... params) {
        publishProgress();
        return null;
    }

    @Override
    protected void onProgressUpdate(Void... values) {
        super.onProgressUpdate(values);

        // 刷新页面的文字
        mLoadingText.setText("test");
    }

    @Override
    protected void onPostExecute(Void result) {
```

```
            //刷新页面的文字
            mLoadingText.setText("test2");

            super.onPostExecute(result);
        }
    }
```

简单介绍一下这三个方法：

- onProgressUpdate 方法的执行在收到 publishProgress 方法调用后，运行于 UI 线程中，对 UI 控件进行处理。
- onPostExecute() 方法，则在 doInBackground() 方法结束后运行在 UI 线程，对 result 进行处理。
- doInBackground() 方法中，就是在后台线程中处理我们的异步任务，不能做类似 Toast 的操作，同样会抛出 Can't create handler inside thread that has not called Looper.prepare() 异常。

接下来，在使用的时候就很简单了：

```
btnStartThread.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        MyTask myTask = new MyTask();
        myTask.execute();
    }
});
```

6.5.11　不能在子线程操作 AlertDialog 和 Toast

异常中的关键字：

Can't create handler inside thread that has not called Looper.prepare()

发生频率：★★★★★

我们继续讨论在子线程操作 UI 的事情。这次是要显示弹出框 AlertDialog 和吐司 Toast。

AlertDialog，只要是在子线程中操作它，就会报上述的错误信息。我测试过，无论是在 onCreate() 还是按钮的点击方法中，都是一样：

```
btnStartThread1.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                new AlertDialog.Builder(S12Activity.this)
```

```
                .setTitle("标题")
                .setMessage("简单消息框")
                .setPositiveButton("确定", null).show();
        }
    }).start();

    }
});
```

相应的解决方案有多种：

方案 1：在外面包一层 Looper.prepare() 和 Looper.loop()，如下所示：

```
btnStartThread2.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                Looper.prepare();

                new AlertDialog.Builder(S12Activity.this)
                        .setTitle("标题")
                        .setMessage("简单消息框")
                        .setPositiveButton("确定", null).show();

                Looper.loop();
            }
        }).start();
    }
});
```

方案 2：Looper 的变形

```
btnStartThread3.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                showAlertByRunnable(S12Activity.this, "", 101);
            }
        }).start();
    }
});

private void showAlertByRunnable(final Context context,
        final CharSequence text, final int duration) {
    Handler handler = new Handler(Looper.getMainLooper());
    handler.post(new Runnable() {
        @Override
        public void run() {
```

```
                new AlertDialog.Builder(S12Activity.this)
                        .setTitle("标题")
                        .setMessage("简单消息框")
                        .setPositiveButton("确定", null)
                        .show();
            }
        });
    }
```

我试过其他三个方案：Handler、runOnUiThread 或 Async，也能解决 AlertDialog 的问题，详细内容请参考 6.5.10 节，但是 Looper 的解决方案，针对操作 UI 控件却是无效的。

吐司 Toast，这个控件和弹出框 AlertDialog 是一样的问题和解决方案，这里不再赘述。代码参见我博客上的源码。[⊖]

6.6　资源相关的异常

资源相关的异常，基本都容易解决。但是有一种情况非常恶心，就是明明 apk 包中有这个资源文件，但是仍然抛出该资源找不到的异常，对此我们也只好认为是内存溢出（OOM）了。

6.6.1　Resources$NotFoundException

异常中的关键字：

android.content.res.Resources$NotFoundException: String resource ID #0x1

发生频率：★★★

这种异常一般是因为参数 int resId 错误，我们把 String 赋值给 int 的 resId，所以编译器找不到正确的 resource 而报错。

最简单的例子，检查一下项目中以下语句的使用：

```
Toast.makeText();
textView1.setText();
```

类似还有一些，这里不列举出来了。这样的函数通常有几个重载，如 TextView 的重载函数如下：

```
TextView.setText(CharSequence text);
TextView.setText(int resId);
```

如果不小心将一个 int 值传给了它，那它不会显示该 int 值，而是跑到工程下去找一个对应的 resource 的 id，那当然是找不到的，于是就报错了。

比如我这里是这样的：

⊖ 代码下载地址：http://www.cnblogs.com/Jax/p/4656789.html。

```
count.setText(incall.getCount());
```

incall.getCount()；返回的是一个 int 值，直接执行 setText 方法是肯定不行的，就会发生上述的 Crash。

解决办法如下：

```
count.setText(String.valueOf(incall.getCount()));
```

或者

```
count.setText(incall.getCount() + "");
```

6.6.2 StackOverfiowError

异常中的关键字：

StackOverfiowError

发生频率： ★★★

发生这种事情，主要是因为 Layout 布局文件结构嵌套层次太深。我们应尽量控制在 5 层以下。要经常使用 Hierarchy View 对其进行优化，移除不必要的视图。

产生这种 Crash 的第二种原因是，在 App 退出的时候，如果 App 中有多个线程，那么在退出 App 的时候可能不能完全关闭 App，即使使用 finish 方法也做不到，必须使用 System.exit(0) 这样的语句才可以。

这是因为 finish 方法只能退出当前 Activity，但还可能有其他 Activity 未关闭，这些 Activity 中有没结束的线程，从而会有一些资源没有释放。

而 exit(int code) 方法可以使进程退出能保证把所有线程的栈空间释放，否则残留的线程栈空间无法回收，将会导致该进程新建线程时栈空间不足，而发生 StackOverfiowError 的异常。

无论是哪种情况导致的 StackOverFlowError，都是由无限递归引起的。在 JVM 中有一个栈，预设了一个深度，当超出这个深度时，就会抛出 StackOverFlowError。我们上述种种解决方案，都是在避免无限循环调用。

6.6.3 UnsatisfiedLinkError

异常中的关键字：

java.lang.UnsatisfiedLinkError:dalvik.system.PathClassLoader [DexPathList[[zip file "/data/app/appname-1.apk"]·····."

发生频率： ★★★★★

遇到这个 Crash，肯定是 so 格式的文件没有加载到。检查 libs 的 armeabi 目录下的 so 文件是否存在。

此外，不能只看 armeabi 下是否有 so 文件，还要看 x86 目录下 so 文件是否存在，如果没有，在 x86 的设备上仍然是加载不到。

由此而上升到 CPU 指令集，Android 上一共有 4 种，armeabi、armeabi-v7a、mips 和 x86。处理 so 文件时要格外小心。[⊖]

如图 6-3 所示，armeabi 和 armeabi-v7a 的 so 数量不一致，是典型的会导致 UnsatisfiedLinkError 的场景。

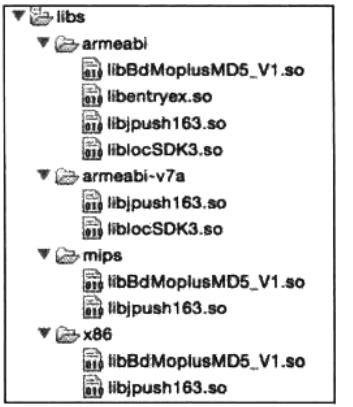


图 6-3　某 App 下的 so 文件

6.6.4　InfiateException 之 FileNotFoundException

异常中的关键字：

Caused by: android.view.InfiateException: Binary XML file line #18: Error infiating class <unknown> at android.view.LayoutInfiater.createView(LayoutInfiater.java:518)

Caused by: java.io.FileNotFoundException: res/drawable-hdpi/add.png at android.content.res.AssetManager.openNonAssetNative(Native Method)

发生频率： ★★★★★

咋一看，还以为是资源找不到，于是去相应的 drawable 目录下去寻找，就发现这个 add.png 文件确实存在啊。

我在网上找了好久好久关于这个 Crash 的描述，但大都不满意。目前看到的一种比较靠谱的说法是 GC 导致的。Activity 销毁了，但是里面涉及的资源并没有被回收，于是便产生内存泄露了，但是表现为 FileNotFoundException。

对此，相应的解决方案是，在 Activity 的 onStop 方法中，手动释放每一张图片资源。[⊜]

6.6.5　InfiateException 之缺少构造器

异常中的关键字：

android.view.InfiateException: Binary XML file line # : Error infiating class com.example.activity1.TestButton

发生频率： ★★★

⊖ 详细情况请参见“Androidndk 开发打包时我们应该如何注意平台的兼容”，文章地址：http://www.cnblogs.com/devinzhang/archive/2012/02/29/2373729.html。

⊜ 关于这个 Crash 的详细描述，请参见 http://blog.csdn.net/yiding_he/article/details/38597703。

创建自定义 view 的时候，碰到上述这个异常，反复研究后发现是缺少一个构造器造成。其中第二个参数用来将 xml 文件中的属性初始化。

自定义控件若需要在 xml 文件中使用，就必须重写带如上两个参数的构造方法。添加后即可正常使用了：

```
public MyView(Context context, AttributeSet paramAttributeSet) {
    super(context, paramAttributeSet);
}
```

补齐这个构造器，异常就消失了。

6.6.6 InfiateException 之 style 与 android:textStyle 的区别

异常中的关键字：

android.view.InfiateException: Binary XML file line #14: Error infiating class

发生频率：★★

在一个 xml 布局文件中，对于实现已经定义好的样式：

```
<style name="NormalText">
    <item name="android:textSize">14sp</item>
    <item name="android:textStyle">normal</item>
    <item name="android:textColor">@color/Gray1</item>
</style>
```

去引用：

```
<TextView
    android:id="@+id/tvUserName"
    android:text="@string/hello_world"
    android:layout_width="230dp"
    android:layout_height="30dp"
    android:layout_marginLeft="10dp"
    android:layout_marginTop="10dp"
    android:textStyle="@style/NormalText"
    />
```

结果发现运行时出错，抛出 android.view.InfiateException：Binary XML file line #14 异常。

按图索骥，我们找到资源文件的第 14 行，发现是 style 的问题，后来去参考 Android 官方文档，感觉应该是把：

```
android:textStyle="@style/NormalText"
```

改为：

```
style="@style/NormalText"
```

修改后的布局文件如下所示：

```
<TextView
    android:id="@+id/tvUserName"
    android:text="@string/hello_world"
    android:layout_width="230dp"
    android:layout_height="30dp"
    android:layout_marginLeft="10dp"
    android:layout_marginTop="10dp"
    style="@style/NormalText"
    />
```

6.6.7　TransactionTooLargeException

异常中的关键字：

android.view.InfiateException: Binary XML file line #14: Error infiating class

发生频率：★★★

官方文档里的解释是，Binder 最大通常限制为 1MB，如果大于 1MB 的话，就会抛出 TransactionTooLargeException 的异常。

相应的解决方法是：不要将大量数据传入 Binder，比如说图片。

这个 Crash 经常出现在图片分享的功能中，因为我们要给第三方分享 SDK 传递很大的图片。此外，使用采集打点数据时也会看到这类 Crash，因为打点的机制不是每点击一次按钮就发一次，而是数据积累到一定量后再发，这个阀值太大就会导致抛出 TransactionTooLargeException 异常。

6.7　系统碎片化相关的异常

这类 Crash 由两部分组成，一方面是和 Android 系统的版本不同有关，比如说在 Android 4.2 的手机执行了 Android 5.0 的语法，崩溃是必然的；另一方面和 ROM 的不同有关，即使是相同的 Android 4.2 版本，由于各个硬件厂商随意定制自己的 ROM，改写其中的系统方法，那么就会表现为 App 的某个页面，不同手机看到不同的效果，甚至是崩溃。

6.7.1　NoSuchMethodError

异常中的关键字：

java.lang.NoSuchMethodError

发生频率：★★★★★

举个例子，错误信息如下：

```
java.lang.NoSuchMethodError:android.os.Bundle.getString
```

android.os.Bunde 中怎么可能没有 getString 这个方法呢?

其实吧，getString 方法有两种参数类型：

```
getString(key);
getString(key, defaultValue);
```

而前面一种是旧的版本，后面这种加了 defaultValue 参数的，是在 2.3 之后的 Android 版本里才加入的，所以，如果你的 android project 设置的 target version 是 2.3.3，而你又用了后面这种新的 getString() 方法的话，那么在 2.3 系统上就会报这个异常。

NoSuchMethodError 异常，只能防范，不能根治，因为 Android 碎片化问题很严重：

一方面 Android 系统的升级，会提供一些新方法，程序员在 App 中使用了这些新方法，而这在老版本的 Android 系统中不存在，就会崩溃。相应的解决方案是，在 DailyBuild 机器上准备不同版本的 SDK，每天晚上自动打包时，把 App 在所有这些 SDK 上都编译一遍，如果缺少方法，首先在编译期间就会报错，自动打包会第一时间发现这些错误。

另一方面，随着 Android 系统的升级，也会有一些旧方法会被废弃，有些厂商的 ROM 有可能删除这些被废弃的方法，于是当程序员在 App 中使用了这些废弃的方法，该 App 在这些厂商的 ROM 中运行就会崩溃。

相应的解决方案是，在开发阶段检查 Android Lint，里面有被废弃的方法的警告，谨慎使用就是了。

如果在项目中一定要使用上述这些新方法或者废弃的旧方法，那么在使用时，要进行 Android 系统版本的判断：

```
int sysVersion = Integer.parseInt(android.os.Build.VERSION.SDK);
if(sysVersion > 9){
    //do something
} else {
    //do other this
}
```

Android 碎片化问题很严重，getString 只是其中的一种情况，类似的问题还有很多，但基本逃不出我上述的分析。

6.7.2 RemoteViews

异常中的关键字：

android.widget.RemoteViews$RefiectionAction. writeToParcel(RemoteViews.java :763)

发生频率：★★★

一直以为在项目中使用 RemoteViews 是件很逼格的事情。这玩意儿一般用在两个地方，一个是在 AppWidget，另外一个是在 Notification。对于应用类 App 而言，有机会用到的是

后者。

比如说，App 应用都有下载更新的功能，一般都是用 AsyncTask 来做这个事情。下载过程中显示进度条，就是使用 Notification，它有一个 contentView 属性，就是 RemoteViews 类型的，我们要为其设置 2 个很关键的值：

- 给 ImageView 绑定图片资源 id。
- 给 TextView 绑定字符串资源 Id。

如下面的例子所示：

```
notification.contentView = new RemoteViews(
    context.getPackageName(), R.layout.notification);
notification.contentView.setImageViewResource(
    R.id.imageview, R.drawable.icon);
notification.contentView.setProgressBar(
    R.id.progressbar, 100, 0, false);
notification.contentView.setTextViewText(
    R.id.textview, "正在更新" + "\n" + "0%");
```

异常就是在绑定时出现的，而且有特定的情况：

1）当你的 Bitmap 为 null 时；

2）当你的 String 为 "" 或者 null 时；

3）Android 版本是 4.0.3 和 4.0.4 时；

如果 Android 版本是 4.1 以上的，则不会出现上述的异常，读不到图片就是控件不显示图片而已，并不会导致程序崩溃。

6.7.3　pointerIndex out of range

异常中的关键字：

java.lang.IllegalArgumentException: pointerIndex out of range　at android.view.MotionEvent.nativeGetAxisValue(Native Method)

发生频率：★★★

关于这个异常有好几种说法，我们逐个进行分析。

首先我们定位问题，在做多点触控放大缩小，操作自己所绘制的图形时发生这个异常，如果是操作图片的放大缩小、多点触控不会出现这个错误。这个 bug 是 Android 系统原因导致的，所以简单有效的办法是在绘图时捕获这个异常，如下所示：

```
public fioat spacing(MotionEvent event) {
    try {
        x = event.getX(0) - event.getX(1);
        y = event.getY(0) - event.getY(1);
    } catch(IllegalArgumentException e) {
```

```
            e.printStackTrace();
        }

        // 以下省略若干语句
    }
```

另一种解决方案是：

1）让你的 view（可能是 ScrollView、WebView、MapView 等），创建一个子 view 继承自它们中的某一个。

2）重写这个 view 的 onInterceptTouchEvent 和 onTouchEvent 方法。

3）为上述这两个方法增加 try…catch…语句，捕获已知的异常，如下所示：

```
try {
    super.onInterceptTouchEvent(MotionEvent ev);
} catch (IllegalArgumentException ex) {

}
return false;

try {
    super.onTouchEvent(MotionEvent ev);
} catch (IllegalArgumentException ex) {

}
return false;
```

这种解决方案，至少在 Android4.1 上是好用的。

按照这个思路，还是有点问题，如果是用 ViewPager 的话，onInterceptTouchEvent 返回 false 会导致 ViewPager 翻页出现 bug，CSDN 上有人给出了相应的解决方案，可以参考。

6.7.4 SecurityException 之一：Intent 中图片太大

异常中的关键字：

Unable to find app for caller android.app.ApplicationThreadProxy@41868f10 (pid=24370) when stopping service Intent { cmp=xxxx }

发生频率：★★

在跳转 activity 的过程中携带的 extras 中有 Bitmap，应尽量减小要传输的图片的体积，或者通过保存图片到 SD 卡中或者通过 URI 方式传递图片参数；否则，图片太大，就会报上述的异常信息。

果然，在去掉了 resultIntent.putExtra（"bitmap"，bitmap）；这条语句后，就不报错了。

一般而言，超过 1MB 的数据，就不要通过 Intent 来传递了。

6.7.5　SecurityException 之二：动态加载其他 apk 的 activity

异常中的关键字：

java.lang.SecurityException: Given caller package com.jianqiang.abc is not running in process ProcessRecord {41e74e50 28637:com.zhao3546.launcher/u0a10142}

发生频率：★★

如果在 apk 中使用了动态注册 BroadcastReceiver，那么 Launcher 动态加载该 apk 时，就有可能出现 java.lang.SecurityException 异常。

相应的解决方案是，修改之前注册 BroadcastReceiver 的地方，通过 ContextHolder() 来注册 BroadcastReceiver，把 apk 重新部署验证即可。[⊖]

6.7.6　SecurityException 之三：No permission to modify thread

异常中的关键字：

java.lang.SecurityException: No permission to modify given thread at
android.os.Process.setThreadPriority(Native Method) at
android.webkit.WebViewCore$WebCoreThread$1.handleMessage (WebViewCore.java:764)

发生频率：★★★

在很多设备上，Android 4.0.4 系统都会有这个问题发生。[⊜]

App 经常会申请一些权限，而有些手机的 ROM 出于安全考虑，则会禁止这些权限，那么当 App 使用到这些权限时，就会发生崩溃。

相应的解决方案是，在执行某些安全相关的操作时，要么加上 if 语句跳过这个操作，要么使用 try…catch…捕获这类异常，宁肯点击后没有反应，也不能崩溃了。

比如拨打电话，我们一般会直接这么写：

```
Intent intent = new Intent(
            Intent.ACTION_CALL,Uri.parse("tel:13800000000"));
            startActivity(intent);
```

但是有些手机系统会禁止 App 拨打电话，即使 AndroidManifest.xml 配置了拨打电话的权限也不行。这时我们就要改写上述代码，预判是否有打电话的权限，以确保不发生崩溃，如下所示：

⊖ 关于这个 Crash 的更详细信息，请参见 http://blog.csdn.net/zhao_3546/article/details/11195881。

⊜ 关于这个 Crash 的更详细信息，请参见 http://stackoverfiow.com/questions/11025182/webview-java-lang-securityexception-no-permission-to-modify-given-thread。

```
PackageManager pm = getPackageManager();
boolean hasPermission =
    pm.checkPermission(Manifest.permission.CALL_PHONE,
        getPackageName()) == PackageManager.PERMISSION_GRANTED;

if (hasPermission) {
    Intent intent = new Intent(
        Intent.ACTION_CALL,Uri.parse("tel:13800000000"));
    startActivity(intent);
}
```

6.7.7 view 的 getDrawingCache() 返回 null

异常中的关键字：

java.lang.NullPointerException at
android.view.View.buildDrawingCache(View.java:6578) at
android.view.View.getDrawingCache(View.java:6428) at ……

发生频率：★★★

当背景图太大，超过了屏幕的大小，就会导致 getDrawingCache() 返回的结果是 null，从而抛出 NullPointException 的异常。

查看 Android 源码，会发现 buildDrawingCache 方法中有这样几行代码：

```
if (width <= 0 || height <= 0 ||
    (width * height * (opaque && !translucentWindow ? 2 : 4) >
      ViewConfiguration.get(mContext)
        .getScaledMaximumDrawingCacheSize())) {
    destroyDrawingCache();
    return;
}
```

在上面的代码中，width 和 height 是所要 cache 的 view 绘制的宽度和高度，所以 width * height *(opaque && !translucentWindow ? 2 : 4) 计算的是当前所需要的 cache 大小。

Android 系统在计算当前所需要的 DrawingCache 大小时，发现这个值超过了系统所提供的最大 DrawingCache 值，这时会直接返回 null。

总之，万恶之源在于图片太大，那我们就控制一下图片的大小，裁减或者等比例缩放，总之不要超过系统所提供的最大 DrawingCache 值，这个值是这么计算的：当前屏幕的分辨率的高和宽相乘，再乘以 4。[⊖]

⊖ 关于这个 Crash 的更详细分析，请参见 http://zartzwj.iteye.com/blog/1098839。社区上关于这个 Crash 的解决方案众说纷坛，目前还没有统一的解决方案。

6.7.8　DeadObjectException

异常中的关键字：

DeadObjectException

发生频率：★★★★

很多开发者在想如何通过编写代码的方式重启 Android 设备。大多数设备都没有 Root 权限，想让设备重启比较简单的方法就是想办法制造一些系统级的错误，强迫 Android 系统自动重启，类似于 Windows 上的 Ring0 级应用崩溃出现蓝屏。对于 Android 来说产生一个 android.os.DeadObjectException 异常是一个不错的方法。

对于 App 应用而言，我从未写过这样的语句来重启系统，网上各路达人对此异常的讨论、发生场景和解决方案也不尽相同，但基本上都是停留在 App 的某个页面，放置一段时间后就崩溃，有的机器能坚持的时间长一些，半个多小时，有些机器也就十几秒的样子。

由此可推测出来，发生 DeadObjectException，其实就是某个对象已经被系统回收了，可我们却还在使用它。[⊖]

6.7.9　Android 2.1 不支持 SSL

异常中的关键字：

java.lang.NullPointerException at

android.webkit,SslErrorHandler.handleMessage (SslErrorHandler.java:62)

发生频率：★★

Android 2.1 版本不支持 SSL，所以发起 https 的请求会导致崩溃。解决方案是，调用 https 的网络请求时，要事先判断 Android 系统的版本，版本过低要提示用户不能进行操作。

6.7.10　ViewFlipper 引发的血案

异常中的关键字：

java.lang.IllegalArgumentException: Receiver not registered:

android.widget.ViewFlipper$1@4083a4d0 at

android.app.LoadedApk.forgetReceiverDispatcher (LoadedApk.java:634)

发生频率：★★★

⊖ 在 StackOverfiow 上对 DeadObjectException 有更详细的讨论，请参见 http://stackoverfiow.com/questions/7037093/android-dead-object-exception。

在 Activity 中使用 ViewFlipper 控件，进行横竖屏切换操作时就会发生这种异常。这是由于 onDetachedFromWindow() 在 onAttachedToWindow() 之前被调用所致。

这个 Crash 很有名，业界公认的解决方案是，重写 ViewFlipper 的 onDetachedFromWindow() 方法：

```
@Override
protected void onDetachedFromWindow() {
    try {
        super.onDetachedFromWindow();
    } catch(IllegalArgumentException e) {
        stopFlipping();
    }
}
```

6.7.11 ActivityNotFoundException

异常中的关键字：

android.content.ActivityNotFoundException: Unable to find explicit activity class {com.android.settings/com.android.settings.WirelessSettings}; have you declared this activity in your AndroidManifest.xml?

发生频率：★★★

看了一下发生错误的操作系统分布，发现都是在 4.0 以上才会出现这类错误信息。究其原因，是 4.0 以上把原来的打开网络设置方式舍弃了，如下修改代码可以解决这个问题：

```
// 3.2 以上打开设置页面
// 也可以直接用 ACTION_WIRELESS_SETTINGS 打开到 WiFi 页面
if (Build.VERSION.SDK_INT > 13) {
    startActivity(new Intent(
                android.provider.Settings.ACTION_SETTINGS));
} else {
    startActivity(new Intent(
                android.provider.Settings.ACTION_WIRELESS_SETTINGS));
}
```

6.7.12 Android 2.2 不支持 xlargeScreens

异常中的关键字：

No resource identifier found for attribute 'xlargeScreens' in package 'android'

发生频率：★★

错误出现在 AndroidManifest.xml 文件的 supports-screens 标记中，原因是 xlargeScreens

属性在 API9（Android 2.3）中才支持。

解决办法：将 Android 2.2 移除，添加 Android 2.3 即可解决。

6.7.13　Package manager has died

异常中的关键字：

Package manager has died at

android.app.ApplicationPackageManager. getApplicationInfo(ApplicationPackageMana

ger.java:213)

发生频率：★★★★

我们一般这样使用 PackageManager，如下所示：

```
try {
    String channelId = getPackageManager()
        .getApplicationInfo(
            getPackageName(),
            PackageManager.GET_META_DATA)
        .metaData.getString("UMENG_CHANNEL");

    PackageInfo info = this.getPackageManager()
            .getPackageInfo(getPackageName(), 0);
} catch (PackageManager.NameNotFoundException e) {

}
```

PackageManager 如果已经 died，说明该进程不存在了，由于某些错误原因 PackageManager 进程已经退出，此时任何向它进行的请求都将失效，让设备重启可能是一个办法。还有一种情况是，App 本身已经处于崩溃状态，这个时候如果 App 已经弹出错误框，再调用 PackageManager 也会出错或卡死。

解决方案就是每次获取 PackageManager 的时候用 try…catch…捕获异常。

6.7.14　SpannableString 与富文本字符串

异常中的关键字：

java.lang.IndexOutOfBoundsException: setSpan(−1 …−1) starts before 0 at

android.text.SpannableStringBuilder.checkRange(SpannableStringBuilder.java:951) at……

发生频率：★★

有一种异常表面看起来是数组越界，但其实并非如此。

从上面的异常信息中能看出，是 SpannableString 的 setSpan 方法越界导致出现崩溃。但是检查相应的代码，并没有刻意使用这个方法。

TextView 要显示的富文本恰好要被换行符截断的时候，因为富文本是使用 SpannableString 技术来显示的，所以会报这种异常。所幸，这个 Crash 不是必现的，取决于机型、分辨率、字体大小、文字和样式很多因素。

相应的解决方案是，在执行 TextView 的 setText 方法时，加上 try…catch…语句捕获 IndexOufOfBoundsException。因为这种情况发生的概率极小，所以即使抛出异常，最多是不显示文本，也不会让 App 崩溃。[⊖]

还有一种情况是，在长按一段文本时，有些 Android 系统对于 EditText 的 getSelectionStart 方法，会返回 -1，这就会导致上述异常情况的抛出，如下所示：

```
public void afterTextChanged(Editable s){
    if(StringUtils.isNullOrEmpty(s.toString()))
        return;

    int editStart = mTxInput.getSelectionStart();
    int editEnd = mTxInput.getSelectionEnd();
    mTxInput.removeTextChangedListener(this);

    while((s.toString().length()) > MAX_INPUT){
        s.delete(editStart-1,editEnd);
        editStart--;
        editEnd--;
    }

    mTxInput.setSelection(editStart);
}
```

所以在使用 getSelectionStart 方法获得值的时候，要判断这个值是否为 -1。[⊖]

6.7.15 Can not perform this action after onSaveInstanceState

异常中的关键字：

java.lang.IllegalStateException:

Can not perform this action after onSaveInstanceState at

android.support.v4.app.FragmentManagerImpl .checkStateLoss(FragmentManager.java:1314)……

android.support.v4.app.BackStackRecord.commit(BackStackRecord.java:595)

发生频率：★★★

⊖ 关于这种情况的详细描述，请参见 http://hold-on.iteye.com/blog/1943437。

⊖ 关于这种情况的详细描述，请参见 http://stackoverfiow.com/questions/22810147/error-when-selecting-text-from-textview-java-lang-indexoutofboundsexception-se。

commit 方法在 Activity 的 onSaveInstanceState() 之后调用就会出错，因为 onSaveInstanceState 方法是在 Activity 即将被销毁前调用，以保存 Activity 数据的，如果在保存完状态后再给它添加 Fragment 就会出错。

解决办法就是把 commit() 方法替换成 commitAllowingStateLoss()，其效果是一样的，如下代码所示：

```
FragmentTransaction ft =
        fragmentActivity.getSuppotFragmentManager().beginTransaction();
ft.add(fragmentContentId, fragments.get(0),
        fragments.get(0).getClass().toString());
```

此外，有时候按后退键触发 onBackPressed 方法也会引发类似的异常，网上有一篇文章详细分析了这类问题的发生原因和解决方案，这里不再赘述。[⊖]

6.7.16　Service Intent must be explicit

异常中的关键字：

Service Intent must be explicit

发生频率：★★★

Android 在升级到 5.0 系统后会产生这样的崩溃。直接通过 action 启动 Service，就会导致这个问题，所以我们必须指定 component 或 package 才能避免这类问题，如下所示：

```
Intent intent = new Intent();
intent.setAction("your action name");
intent.setPackage(getPackageName());
context.startService(intent);
```

很多第三方 SDK 都存在这个问题，我们需要更新 SDK 到最新版本，才能保证 Android 5.0 系统下的 App 不会因此而崩溃。

6.8　SQLite 相关的异常

在 App 中，一般都使用 SQLite 这个数据库，本节介绍的 Crash 也是围绕着这个主题发生的。SQLite 相关的异常大都和 IO 操作不当有关，由于我们无法猜测用户手机发生崩溃时的状态，所以这类异常是最难修复的。

⊖ 详细内容请参见：http://zhiweiofii.iteye.com/blog/1539467。

6.8.1 No transaction is active

异常中的关键字：

android.database.sqlite.SQLiteException: cannot commit – no transaction is active

发生频率：★★★

在事务中，逐条循环插入（for+insert）大量数据时会导致这类崩溃。Android 中在 SqlLite 插入数据的时候默认一条语句就是一个事务，有多少条数据就有多少次磁盘操作，而且不能保证所有数据都能同时插入。

相应的解决方案是使用 SQLLite 提供的批量插入语法，一次性地把这些数据都插入到数据库中，如下所示：

```
public void insertOrUpdateDataBatch() {
    SQLiteDatabase db = getWritableDatabase();
    db.beginTransaction();
    try {
        for (String sql : sqls) {
            db.execSQL(sql);
        }
        //设置事务标志为成功，当结束事务时就会提交事务
        db.setTransactionSuccessful();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        db.endTransaction();
        db.close();
    }
}
```

这段代码的成功与否，就在于 setTransactionSuccessful() 这个方法。在这个方法执行前，所有的 execSQL 方法都不会更新到数据库；等这个方法执行后，会一次性把所有 execSQL 方法都执行完成，数据会同步到数据库。不信的话可以在 for 循环处打个断点，看数据库是否有变化。

6.8.2 忘记关闭 Cursor

异常中的关键字：

android.database.CursorWindowAllocationException: Cursor window allocation of 2048 kb failed

发生频率：★★★

这个异常是因为使用 SQLLite 时忘记释放游标导致的，内存泄漏得多了，就崩溃了。相

应的解决办法就是手动关闭 Cursor，如下所示：

```
cursor.close();
```

6.8.3　数据库被锁定

异常中的关键字：

android.database.sqlite.SQLiteDatabaseLockedException: database is locked

发生频率：★

当我们试图在不同的线程中创建多个连接时，就会抛出这个异常。相应的解决方案是将数据库做成一个单例。[⊖]

单例固然能解决单进程操作数据库的情况，但是对于多进程 App 而言，还是需要 ContentProvider。

6.8.4　试图再打开已经关闭的对象

异常中的关键字：

java.lang.IllegalStateException: attempt to re-open an already-closed object

发生频率：★

这个问题是上一个问题的延续。即使做成了单例，如果在不同的线程中创建多个连接，就会报当前的错误信息。

频繁地操作 SQLite 数据库容易产生这个崩溃。我们习惯于每执行一次数据库操作，都打开和关闭数据库各一次。这就会导致当两个线程同时操作数据库时，比如，A 为读数据，B 为写数据，当 A 读完就会关闭数据库，而 B 这时正在写数据，那么上述 Crash 就会产生了。

在实际应用中，App 中的 IM，因为要把聊天信息存放到本地 SQLite，最容易看到这类异常，这时好的做法是，在当前聊天室，保持数据库一直处于 Open 状态，等退出聊天室再执行 close 方法。

6.8.5　文件加密了或无数据库

异常中的关键字：

SQLiteDatabaseCorruptException: file is encrypted or is not a database

发生频率：★

⊖ 单例的实现请参见：http://zhiwei.neatooo.com/blog/detail?blog=5343818a9d4869f0310000de。

请注意 SQLLite DB 文件的版本，如果有两个 DB，一个是 2.8.17，另一个是 3.7.7.1，那么就会出现这个异常。将其统一成一个版本即可。

此外，如果 DB 破损，也可能出现这种异常。当我们将 App 安装在 SD 卡上，多次插拔就会导致部分文件破损。

6.8.6 WebView 中 SQLLite 缓存导致的崩溃

异常中的关键字：

SQLiteDiskIOException: disk I/O error……at

android.webkit.WebViewDatabase$1.run(WebViewDatabase.java:1000)

发生频率： ★

注意 这个异常信息中还带有 WebViewDatabase 的内容，说明我们的程序使用了 WebView 控件的缓存技术。但是原因不详。有人说把数据库删除了就会崩溃，但我试过了，对 WebView 是无效的。

由此而谈到 Android 中 WebView 的缓存策略。WebView 中存在着两种缓存：

- 网页数据缓存，存储打开过的页面及资源。
- Html5 缓存，即 appcache。

缓存数据的构成如图 6-4 所示。

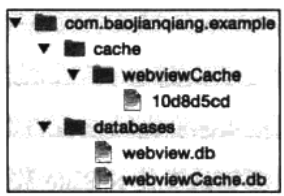


图 6-4 WebView 中的 cache 数据

WebView 自带的缓存机制里面，会将 url 保存在 webviewCashe.db 中，将 url 内容保存在 webviewCashe 文件夹下，比如说图中的 10d8d5cd，就是 url 对应的一张图片，此外 html、js 等文件也会存下来。

而对于 databases 目录下的 webview.db 和 webviewCashe.db，都会自动生成 1 个名为 android_metadata 的表，只要创建 SQLLite 数据库中的表，就会自动创建这个表，表中只有一个 locale 字段，里面存放的是 en-US 或者 zh-CN 这样的值（是哪个值取决于 Android 系统），用于标示语言文化。从数据库中读取的文本是否为乱码，就取决于这个值了。

我的系统是英文的，所以我检查过 locale 值是 en-US；而我同事的中文系统则显示 zh-CN。

缓存模式有五种，见表 6-1。

表 6-1 缓存模式

模　式	简　介
LOAD_CACHE_ONLY	不使用网络，只读取本地缓存数据
LOAD_DEFAULT	根据 cache-control 决定是否从网络上取数据

（续）

模　式	简　介
LOAD_CACHE_NORMAL	API level 17 中已经废弃，从 API level 11 开始作用同 LOAD_DEFAULT 模式
LOAD_NO_CACHE	不使用缓存，只从网络获取数据
LOAD_CACHE_ELSE_NETWORK	只要本地有，无论是否过期，或者 no-cache，都使用缓存中的数据

根据以上几种模式，建议缓存策略为：判断是否有网络，有的话，使用 LOAD_DEFAULT，无网络时，使用 LOAD_CACHE_ELSE_NETWORK。

6.8.7　磁盘读写错误

异常中的关键字：

android.database.sqlite.SQLiteDiskIOException: disk I/O error (code 1802)

发生频率：★

我曾经认为，在 UI 线程执行 dbHelper.getWritableDatabase() 这句话的时候，UI 线程会把数据库锁住。但是后来 Bugly 的“精神哥”告诉我，dbHelper 只有在创建数据库、进行事务处理时才会锁住数据库。默认情况下 dbHelper 会缓存 DB 实例，执行类似于 getWritableDatabase 的操作是立即返回的，并不会上锁。

disk I/O error 这类异常的抛出，是因为多线程修改 DB，比如一个线程在写数据，另一个线程却在删除数据。

6.8.8　android_metadata 表不存在

异常中的关键字：

android.database.sqlite.SQLiteException: no such table: android_metadata SQLiteOpenHelper.getReadableDatabase

发生频率：★

开发中需要连接 SQLite 数据库，当使用如下方法打开数据库时就会抛出上述错误：

```
SQLiteDatabase database = SQLiteDatabase.openDatabase(
            PATH, null, SQLiteDatabase.OPEN_READONLY);
```

解决办法是，将 openDatabase 方法中最后一个参数修改为 SQLiteDatabase.NO_LOCALIZED_COLLATORS 即可。

6.8.9 android_metadata 表中的 locale 字段

异常中的关键字：

android.database.sqlite.SQLiteException: Failed to change locale for db '/data/data/appname/databases/webview.db' to 'zh_CN'.

发生频率： ★

根据对 6.8.8 中 Crash 的分析，我们知道 android_metadata 这个表中有个 locale 字段。

这里要介绍的 Crash 发生在 WebView 控件生成的缓存数据库中，但是发生的概率极小（个位数）。对此，众说纷坛。甚至有美国人在 StackOverFlow 上说中国产的手机也报这个异常，只是不能转换为 en-US 而已。我只能怀疑是 ROM 的问题。

6.8.10 数据库或磁盘满了

异常中的关键字：

android.database.sqlite.SQLiteFullException: database or disk is full

发生频率： ★

当数据库文件存放在内存中时，就和存文件或者 SharedPreferences 一样，会因为内存满了而报错，只是这次的错误信息更具体，会提示我们数据库 / 磁盘满了。

6.9 不明觉厉的异常

不是所有的 Crash 都能找到原因，比如说内存爆了，也就是 OOM，但是表现为其他的症状，也有可能是混淆的问题。

对于线上的 Crash，我们要本着“为什么开发期间没有发现”的思路来进行修复，调试期间没有问题但是到了线上就有问题了，原因有几种：

- 测试不充分，有些场景没有考虑到。
- 服务器返回给 App 的数据不规范，而 App 又没做容错处理。恰恰测试时数据是没问题的，上线后服务器返回的数据不规范，就会有各种崩溃。
- 也有可能是在线上调试，没写好的代码抛出来的异常。这种 Crash 永远不会重现。我们需要排除这样的 Crash，否则会浪费大量的人力在上面。

其实，很多线上 Crash 都是无厘头的，让人无从下手修复。我在这里教大家一种简单有效的办法，那就是发现这样的线上 Crash，在出错的代码行加上 try… catch…语句，专门捕获这种异常。记得在 catch 语句中将这次异常发送到服务器，并把 crash_type 标记为 0（线上 Crash

的这个值为 1），这样我们就能在每天几千个 Crash 中捕获到一些，而阻止应用不崩溃了。

当我们捕获到这种异常，尽量不要让程序崩溃，而是退回到上一个页面，请用户重新操作一遍。之前的操作可能会因为各种各样的原因而引发异常，重新操作一遍能极大减少这种情形。但如果每次退回后重新操作还是这种崩溃，那么就要重点对待了，有可能是 MobileAPI 脏数据，有可能是某款机型不兼容，有可能就是一个代码上的 bug，具体情况具体分析。

6.9.1　内存溢出

异常中的关键字：

OutOfMemoryException

发生频率：★★★★★

我们时常抱怨 Android 系统为每个 App 分配的内存太小，只有几十兆，殊不知在 AndroidManifest.xml 中有个参数可以设置：

```
<application android:largelHeap="true"
```

这样就能增加系统为当前 App 分配的内存了，甚至到 100MB 以上。使用后，OOM 的崩溃明显减少很多。

但是，天底下没有免费的午餐。当内存很大的时候，每次 GC 的时间也会长一些，性能就会下降。Android 官方给的建议是，作为程序员的我们应该努力减少内存的使用，使用回收和复用的方法，而不是想方设法增大内存。

6.9.2　Verify Failed

异常中的关键字：

java.lang.VerifyError: Rejecting class xxxx.package.activityA that attempts to sub-class erroneous class xxxx.package.Activity 基类

发生频率：★★★★

这个问题至今没有查到原因。

6.10　其他情况的异常

最后，是一些不太好归类的异常信息。这就像武侠小说中的独行大侠，无门无派但也不能小觑。

6.10.1 TimeoutException

异常中的关键字：

com.android.internal.BinderInternal$GcWatcher.finalize() timed out after 10 seconds

发生频率：★

GC 回收超时会抛出该异常，注意重写 finalize 方法时不要有超时的操作。[⊖]

6.10.2 JSON 解析异常

异常中的关键字：

org.json.JSONException: No value for UserName at

org.json.JSONObject.get(JSONObject.java:354) at……

发生频率：★★★

在 JSON 解析中经常会遇到这种异常。

这是因为我们在解析 JSON 的时候，使用了 getString("UserName") 而不是 optString ("UserName")，如果 UserName 这个 key 在 JSON 字符串中不存在，前者会抛出上述异常，后者则会返回空。

类似地，还有 getJsonArray 方法，建议的解决方案是改用 optJsonArray 方法，才不会发生崩溃。

6.10.3 JSONArray 在初始化时为空

异常中的关键字：

java.lang.NullPointerException at

org.json.JSONTokener.nextCleanInternal(JSONTokener.java:116) at

org.json.JSONTokener.nextValue(JSONTokener.java:94)t……

发生频率：★★★

我们知道 JSONArray 的初始化如下所示：

```
public void simulateJSONException() throws JSONException {
    String jsonString = "";
```

⊖ 关于这个异常的不完全诊断，请参见 http://stackoverfiow.com/questions/24021609/how-to-handle-java-util-concurrent-timeoutexception-android-os-binderproxy-fin。

```
    JSONArray array = new JSONArray(jsonString);
    for (int i = 0; i < array.length(); i++) {
        JSONObject jsonObject = array.getJSONObject(i);
    }
}
```

当 jsonString 这个值为空时，就会报上述的异常信息。

6.10.4　第三方 SDK 抛出的 Crash

在引入第三方 SDK 的同时，也会引入 SDK 导致的崩溃。举个例子：GoogleAnalytics，简称 GA。Google 提供的这个工具，很多公司用来搜集线上 Crash。殊不知，有些手机只要启动这个记录 Crash 的功能，就会 Crash，每天会有一两千个崩溃就是因为这个原因导致的，异常信息如下所示：

```
java.lang.RuntimeException: Package manager has died at
android.app.ApplicationPackageManager.getPackageInfo(Application
PackageManager.java:82) at
com.google.analytics.tracking.android.StandardExceptionParser.setIn
cludedPackages(Unknown Source)
```

我也是在把线上 Crash 收集到自己的服务器后，才发现这个问题的。后来把 GA 的这个 Crash 发送功能禁用掉，就不再因此而崩溃了。

6.10.5　两个不同类型的 View 有相同的 id

异常中的关键字：

java.lang.IllegalArgumentException: Wrong state class, expecting View State but received class android.widget.ScrollView$SavedState instead. This usually happens when two views of different type have the same id in the same hierarchy. This view's id is id/0xff0000. Make sure other views do not use the same id.

发生频率：★★★

异常信息中不一定每次都是 ScrollView，也有 TextView 或者其他控件。

异常信息已经解释得很清楚了，在一个页面中，两个不同类型的 View 有相同的 id，就会导致崩溃。

这个悲剧的发生，是 Android 系统的内部机制导致的。ViewPager 中有两个页面，每个页面的 layout 布局文件中都有一个 id 名叫 scroll_view 的控件，那么当我们重写 onSaveInstanceState 这个方法的时候，如果要保存 scroll_view 的状态，比如 scrollX 和 scrollY 的值，那么在 onRestoreInstanceState 方法中恢复这两个值时，就会分不清楚究竟是哪一个。

Android 官方建议最好保证每个 View 的 id 都是唯一的，或者至少在一个局部的 layout 文件中这么做，因为很显然，如果同一个 layout 文件中有两个 id 都是 "android:id="@+id/button" 的按钮，那么通过 findViewById 的时候只能找到前面的按钮，后面的那个就没机会被找到了，所以 Android 官方的说法是合理的。

此外，还应该加上特别重要的一条：当在 Activity 中，确定要保存 / 恢复一个 View 的状态的时候，一定要保证它们有唯一的 id，因为 Android 内部用 id 作为保存、恢复状态时使用的 key，否则就会发生一个覆盖另一个的悲剧。

6.10.6 LayoutInfiater.from().infiate() 使用不当导致的崩溃

异常中的关键字：

No package identifier when getting value for resource number 0x00000001

发生频率：★★★

在程序中使用 LayoutInfiater.from().infiate() 语句时，必须写在具体的子类中，一定不能工作在父类或虚类里，如下所示：

```
View view = LayoutInfiater.from(mContext)
                  .infiate(LAYOUT_ID, this, true);
```

这里有个 this 指针的问题，当 initView 方法让虚类调用时，这个 this 指向谁？是虚类自己还是子类？正是因为 Android 系统搞不清楚所以就崩溃了，另外这个 infiate 本身就有一定的特殊性，是不能随便乱用 this 的。我尝试过把 BaseGuideView 里的 initView 方法不写成虚方法，而是一个空的函数，但依旧报错。所以遇到这种情况，加载布局一定要由各个子 View 自行加载并初始化。[⊖]

6.10.7 ViewGroup 中的玄机

异常中的关键字：

java.lang.IllegalArgumentException: parameter must be a descendant of this view

发生频率：★★★

这个崩溃，是通过 ViewGroup 的 offsetRectBetweenParentAndChild 方法抛出来的。

```
void offsetRectBetweenParentAndChild(void descendant,
     Rect rect, boolean offsetFromChildToParent,
     boolean clipToBounds)
```

⊖ 关于这个崩溃的详细信息，请参见 http://blog.csdn.net/yanzi1225627/article/details/37338565。

该方法就是用来计算父子重叠的区域。它是通过所给的 descendant 这个 View 逐级向上寻找 Parent View，同时将 Rect 转换为同级坐标系来计算的。

在这个方法的末尾，如果最终找到的 Parent View 和当前 View 不一致，则会抛出这个异常。说白了，就是 descendant 参数必须是当前 View 的子孙。[⊖]

那么什么时候 descendant 不是当前 View 的子孙呢？在 UI 调整的时候，会改变当前界面中拥有焦点的控件。我们应该实时确保这个控件是当前 View 的子孙，所以相应的解决方案也很简单，每次都重新设置一下焦点，让当前 View 始终获得焦点。与此同时，如果是 ListView，还要清空 ListView 中其他控件抢到的焦点。

6.10.8　Monkey 点击过快导致的崩溃

异常中的关键字：

java.lang.NullPointerException at

android.view.ViewRootImpl$ViewRootHandler.handleMessage(ViewRootImpl.java:3046) at ……

发生频率：★★

有种 Crash，只有执行 Monkey 脚本时才会抛出来。没办法，人手点的速度远不及 Monkey 点击的速度。这种 Crash，我们就不深究了，只要确保手点的时候不崩溃即可。

有一种相对成熟的解决方案，那就是为每个点击事件加一个延迟函数，如下所示：

```
public void onClick(View v) {
    if(isWindowLocked())
        return;
    // 接下来的代码执行点击按钮后的逻辑
}
```

我们把 isWindowLocked 这个延迟方法写到 BaseActivity 中：

```
public Boolean isWindowLocked() {
    long current = SystemClock.elapsedRealtime();
    if (current - mLastOnClickTime > 500) {
        mLastOnClickTime = current;
        return false;
    }

    return true;
}
```

这样 Monkey 就不会跑那么快了。

代码中 500 的意思是延迟 0.5 秒。这取决于 Monkey 中事件的间隔时间，一般我们设置为 0.5 秒。

⊖ 关于这个崩溃的详细信息，请参见 http://blog.sina.com.cn/s/blog_5704bfaf0102v3bn.html。

6.10.9 图片缩放很多倍

异常中的关键字：

java.lang.IllegalArgumentException: bitmap size exceeds 32bits

发生频率：★★★

当图片缩放了很多倍时，导致内存溢出，就会抛出这个异常，多发生在全屏显示一张图片的时候。

如下所示，postScale 方法中的参数就是宽和高比例，要在这里增加 try… catch…捕获这个异常。

```
// srcWindth 和 srcHeight 是缩放前
// tagetWidth 和 targetHeight 缩放后
Float scaleW = (fioat)targetWidth / (fioat)srcWidty;
Float scaleH = (fioat)targetHeight / (fioat)srcHeight;

Matrix matrix = new Matrix();
Matrix.postScale(scalew,scaleH);
```

6.10.10 图片宽高为 0

异常中的关键字：

java.lang.IllegalArgumentException: width and height must be > 0 at
android.graphics.Bitmap.nativeCreate(Native Method)

发生频率：★★

产生这个异常，通常是因为没有取到图片的宽和高，于是就返回默认值 0 了。

这是件很诡异的事情，因为任何一张图片都是有宽和高的，那么唯一的一种解释就是，没加载到这个图片（比如说缓冲数据被清空），或者提前调用了获取图片的宽和高的方法，这时候就得到 0 值了。

暂时还没有完美的解决方案，只能看到哪个页面有这样的异常信息，就加 try…catch…语句防止获取图片宽高时出错。

6.10.11 不能重复添加组件

异常中的关键字：

View xxxx has already been added to the window manager

发生频率：★★★

这个异常发生在 windowmanger.addView（view）这行代码中，意思大体是说这个 view 在 Window Manager 中已经存在，不能再添加相同的了。

通常的解决办法是在添加 view 时，捕获这个异常，但是并没有解决问题，想要添加的 view 并没有被加入到 Window Manager 中。

于是我们想到，先执行 windowmanger.removeView（view），再执行 addView 方法，这样就不会出问题了。但是问题接踵而至，当 Window Manager 中并不存在这个 view 时，执行 remove 方法反而会抛出 View not attached to window manager 的异常信息。基于此，得到终极解决方案，如下所示：

```
try {
    windowmanager.removeView(view);
} catch(IllegalStateException ex) {
    e.printStackTrace();
}

try {
    windowmanager.addView(view);
} catch(IllegalStateException ex) {
    e.printStackTrace();
}
```

也就是说，即使 removeView 失败，也能继续执行接下来的 addView 操作。

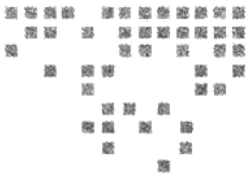
6.11　本章小结

这是极其枯燥无味的一章，我努力让自己的语言生动一些，也不一定有效。原设想一个月就完成这一章，谁知却整整写了 6 个月。

从第一批收集到的 40 多个异常，越写越多，慢慢扩充到现在的 80 多个异常。网络上早就有人在讨论 Android 千奇百怪的异常信息，每篇文章都要仔细看一遍，与此同时，还要为每一个崩溃做两个 Demo，一个用来演示崩溃，另一个则用来演示如何修复崩溃。只有这样才能辨别真伪，但却最费时间和精力。

另一个困扰我的问题是，如何辨别开发期间发现的崩溃和线上环境发生的崩溃。前者是可以在上线前就能发现的，如果不修复，功能流程根本不能走下去，所以，这类崩溃，我尽量不去分析。我着力解决的是那些开发期间发现不了，只有通过线上环境几十万用户才能点出来的崩溃。我总在想，为什么这个崩溃在开发和测试期间不能发现？

我不能确保本章中每个异常分析都是正确的，有些情况我只能是大胆猜测，甚至还没有结论，如果读者有更好的解释，请在我的博客上留言，我将非常感激。



第 7 章 Chapter 7

ProGuard 技术详解

ProGuard 是一个很枯燥且让人没有成就感的技术，至少我是这么认为的。但不可否认的是，Android 项目没有了 ProGuard 还真就不行。既然投身程序员这个行业，就要耐得住寂寞，在夜深人静的时候，加班给代码做混淆。本章专门介绍 ProGuard 的工作原理，以及使用方法。

7.1 ProGuard 简介

在 Android 中一提起 ProGuard，我们就会认为它是用来混淆代码的，殊不知 ProGuard 一共包括以下 4 个功能。

- 压缩（Shrink）：侦测并移除代码中无用的类、字段、方法和特性（Attribute）。
- 优化（Optimize）：对字节码进行优化，移除无用的指令。
- 混淆（Obfuscate）：使用 a、b、c、d 这样简短而无意义的名称，对类、字段和方法进行重命名。
- 预检（Preveirfy）：在 Java 平台上对处理后的代码进行预检。

如果仅仅是为了代码混淆，ProGuard 有一个兄弟产品 DexGuard 可以试试，地址如下：
http://www.saikoa.com/dexguard
常常看到有人诟病 ProGuard 不会混淆字符串常量，DexGuard 可以做这个事情。

ProGuard 是一个开源项目，在 SourceForge 上进行维护，地址如下：
http://ProGuard.sourceforge.net。

从上述地址下载 ProGuard 之后，能同时看到官方文档和示例，不过是英文的，目前市面上没有相应的中文翻译版，也没有一篇详尽的介绍文章。

如果你的项目已经使用了某个版本的 ProGuard，比如，现在市面上最流行的是 4.7 版本，我建议不要进行升级。一切以稳定为首，如果一定要升级到最新版本，请在使用 ProGuard 后，对项目的所有模块进行全功能回归测试。

7.2 ProGuard 工作原理

ProGuard 由 shrink、optimize、obfuscate 和 preverify 四个步骤组成，其中每个步骤都是可选的，我们可以通过配置脚本来决定执行其中的哪几个步骤，如图 7-1 所示。

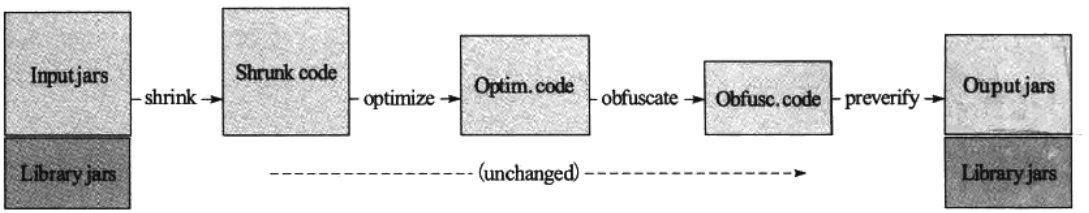


图 7-1 ProGuard 执行流程

这里，我们引入 Entry Point 的概念。Entry Point 是在 ProGuard 过程中不会被处理的类或方法。在压缩的步骤中，ProGuard 会从上述的 EntryPoint 开始递归遍历，搜索哪些类和类的成员在使用。对于没有被使用的类和类的成员，就会在压缩阶段丢弃。

接下来在优化的步骤中，那些非 EntryPoint 的类、方法都会被设置为 private、static 或 final，不使用的参数会被移除，此外，有些方法会被标记为内联的。在混淆的步骤中，ProGuard 会对非 EntryPoint 的类和方法进行重命名。

7.3 如何写一个 ProGuard 文件

接下来，我们只讲 ProGuard.cfg 混淆文件要怎么写。这是一个三步走的过程。

7.3.1 基本混淆

以下是混淆最基本的配置信息，任何 App 都要使用，可以作为模板使用，我为每行代码都增加了注释：

1. 基本指令

```
# 代码混淆压缩比，在 0~7 之间，默认为 5，一般不需要改
-optimizationpasses 5

# 混淆时不使用大小写混合，混淆后的类名为小写
```

```
-dontusemixedcaseclassnames

# 指定不去忽略非公共的库的类
-dontskipnonpubliclibraryclasses

# 指定不去忽略非公共的库的类的成员
-dontskipnonpubliclibraryclassmembers

# 不做预校验，preverify 是 proguard 的 4 个步骤之一
# Android 不需要 preverify，去掉这一步可加快混淆速度
-dontpreverify

# 有了 verbose 这句话，混淆后就会生成映射文件
# 包含有类名 -> 混淆后类名的映射关系
# 然后使用 printmapping 指定映射文件的名称
-verbose
-printmapping proguardMapping.txt

# 指定混淆时采用的算法，后面的参数是一个过滤器
# 这个过滤器是谷歌推荐的算法，一般不改变
-optimizations !code/simplification/arithmetic,!field/*,!class/merging/*

# 保护代码中的 Annotation 不被混淆
# 这在 JSON 实体映射时非常重要，比如 fastJson
-keepattributes *Annotation*

# 避免混淆泛型，
# 这在 JSON 实体映射时非常重要，比如 fastJson
-keepattributes Signature

// 抛出异常时保留代码行号，在第 6 章异常分析中我们提到过
-keepattributes SourceFile,LineNumberTable
```

-dontskipnonpubliclibraryclasses 用于告诉 ProGuard，不要跳过对非公开类的处理。默认情况下是跳过的，因为程序中不会引用它们，有些情况下人们编写的代码与类库中的类在同一个包下，并且对包中内容加以引用，此时需要加入此条声明。

对于 -dontusemixedcaseclassnames，Microsoft Windows 用户请注意：默认情况下，ProGuard 假定你使用的操作系统能够区分两个只是大小写不同的文件名（比如，A.java 和 a.java 被认为是两个不同的文件）。显然 Microsoft Windows 不是这样的操作系统（Windows 是对文件名是大小写不敏感的）。因此 Windows 用户必须为 ProGurad 指定 –dontusemixedcaseclassnames 选项。如果不这么做并且你的项目中有超过 26 个类的话，那么 ProGuard 就会默认混用大小写文件名，而导致 class 文件相互覆盖。安全起见，从 0.9.0 版本开始，EclipseME 默认为 ProGuard 设置 –dontusemixedcaseclassnames 选项。项目中有很多类的 UNIX 用户可以删除这个选项，这样最终产生的 JAR 文件的大小可以进一步缩小。

2. 需要保留的东西

```
# 保留所有的本地 native 方法不被混淆
-keepclasseswithmembernames class * {
```

```
    native <methods>;
}

# 保留了继承自 Activity、Application 这些类的子类
# 因为这些子类都有可能被外部调用
# 比如说，第一行就保证了所有 Activity 的子类不要被混淆
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider
-keep public class * extends android.app.backup.BackupAgentHelper
-keep public class * extends android.preference.Preference
-keep public class * extends android.view.View
-keep public class com.android.vending.licensing.ILicensingService

# 如果有引用 android-support-v4.jar 包，可以添加下面这行
-keep public class com.tuniu.app.ui.fragment.** {*;}

# 保留在 Activity 中的方法参数是 view 的方法，
# 从而我们在 layout 里面编写 onClick 就不会被影响
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}

# 枚举类不能被混淆
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

# 保留自定义控件（继承自 View）不被混淆
-keep public class * extends android.view.View {
    *** get*();
    void set*(***);
    public <init>(android.content.Context);
    public <init>(android.content.Context, android.util.AttributeSet);
    public <init>(android.content.Context, android.util.AttributeSet, int);
}

# 保留 Parcelable 序列化的类不被混淆
-keep class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator *;
}

# 保留 Serializable 序列化的类不被混淆
-keepclassmembers class * implements java.io.Serializable {
    static final long serialVersionUID;
    private static final java.io.ObjectStreamField[] serialPersistentFields;
    private void writeObject(java.io.ObjectOutputStream);
    private void readObject(java.io.ObjectInputStream);
    java.lang.Object writeReplace();
```

```
    java.lang.Object readResolve();
}

# 对于 R（资源）下的所有类及其方法，都不能被混淆
-keep class **.R$* {
    *;
}

# 对于带有回调函数 onXXEvent 的，不能被混淆
-keepclassmembers class * {
    void *(**On*Event);
}
```

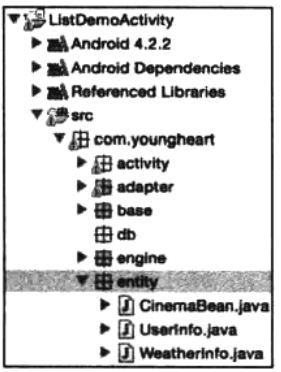


图 7-2　一个 Android 项目的目录结构

7.3.2　针对 App 的量身定制

我们创建一个 Android 项目，它的包名和项目结构图如图 7-2 所示：

1. 保留实体类和成员不被混淆

对于实体，要保留它们的 set 和 get 方法，对于 boolean 型 get 方法，有人喜欢命名为 isXXX 的方式，所以不要遗漏了。

```
-keep public class com.youndheart.entity.** {
    public void set*(***);
    public *** get*();
    public *** is*();
}
```

一种好的做法是把所有实体都放在一个包下进行管理，这样只写一次混淆就够了。避免以后在别的包中新增的实体而忘记保留，代码在混淆后因为找不到相应的实体类而崩溃。

2. 内嵌类

内嵌类经常会被混淆，结果在调用的时候为空就崩溃了。最好的解决办法就是把这个内嵌类拿出来，单独成为一个类。

如果一定要内置，那么这个类就必须在混淆时进行保留。比如说 com.example.youngheart 包下面的 MainActivity，它有一些内嵌类，以下指令保留 MainActivity 的所有内嵌类：

```
-keep class com.example.youngheart.MainActivity$* { *; }
```

$ 这个符号就是用来分割内嵌类与其母体的标志。还记得 4.1.2 中保留 R（资源）下面的所有类及其方法的指令吗？如出一辙：

```
-keep class **.R$* {*;}
```

3. 对 WebView 的处理

如果项目中用到了 WebView 的复杂操作，请加入以下这两段代码：

```
-keepclassmembers class *
        extends android.webkit.webViewClient {
    public void *(android.webkit.WebView,
        java.lang.String, android.graphics.Bitmap);
    public boolean *(android.webkit.WebView,
            java.lang.String)
}
-keepclassmembers class * extends android.webkit.webViewClient {
    public void *(android.webkit.webView,
            java.lang.String)
}
```

4. 对 JavaScript 的处理[⊖]

App 应用要经常与 HTML5 页面的 JavaScript 进行交互，如下所示：

```
class JSInteface1 {
    @JavascriptInterface
    public void callAndroidMethod(int a, fioat b, String c, boolean d) {
        if (d) {
            String strMessage = "-" + (a + 1) + "-" + (b + 1) + "-" + c
                    + "-" + d;

            new AlertDialog.Builder(MainActivity.this).setTitle("title")
                    .setMessage(strMessage).show();
        }
    }
}
```

这个例子参见第 3 章中 3.4 节介绍的 App 与 HTML5 之间的交互。我接下来要讨论的是，如何确保这些 js 要调用的原生方法不被混淆。

JSInterface 是 MainActivity 的子类，所以保留指令要这么写：

```
-keepclassmembers
    class com.example.youngheart.MainActivity$JSInterface1 {
    <methods>;
}
```

请在项目中搜索 addJavascriptInterface，我们要对所有使用的地方设置保留指令。

5. 处理反射

也许有人会问，在程序中使用 SomeClass.class.method1 这样的静态方法，ProGuard 如何处理？

答案是，被引用的类，如 SomeClass，肯定会在压缩过程中被保留。

那么对于 Class.forName("SomeClass") 呢？ SomeClass 不会在压缩过程中被移除，ProGuard 在这点上还是蛮聪明的，它会检查程序中使用的 Class.forName 方法，对参数 SomeClass 这样的字符串则法外开恩，不会移除。

但是，在混淆过程中，无论是 Class.forName("SomeClass")，还是 SomeClass.class，就都不能蒙混过关了。SomeClass 这个类的名称会被混淆。因此，我们要在 ProGuard.cfg 文件中，

⊖ 对 JavaScript 的处理，详细内容请参见 http://blog.csdn.net/span76/article/details/9065941。

保留这个类名称。

不光是 Class.forName("SomeClass")，以下方法也同样适用：

- SomeClass.class.getField("someField")
- SomeClass.class.getDeclaredField("someField")
- SomeClass.class.getMethod("someMethod", new Class[] {})
- SomeClass.class.getMethod("someMethod", new Class[] { A.class })
- SomeClass.class.getMethod("someMethod", new Class[] { A.class, B.class })
- SomeClass.class.getDeclaredMethod("someMethod", new Class[] {})
- SomeClass.class.getDeclaredMethod("someMethod", new Class[] { A.class })
- SomeClass.class.getDeclaredMethod("someMethod", new Class[] { A.class, B.class })
- AtomicIntegerFieldUpdater.newUpdater(SomeClass.class, "someField")
- AtomicLongFieldUpdater.newUpdater(SomeClass.class, "someField")
- AtomicReferenceFieldUpdater.newUpdater(SomeClass.class, SomeType.class, "someField")

在混淆的时候，要在项目中搜索一下上述这些方法，将相应的类或者方法的名称进行保留而不被混淆。做混淆的开发人员，应该对代码比较熟悉，以确保万无一失。

6. 对于自定义 View 的解决方案

但凡是在 layout 目录下的 xml 布局文件中配置的自定义 View，都不能进行混淆。为此要遍历 layout 下所有的 xml 布局文件，找到那些自定义 View，然后确认其是否在 proguard 文件中保留了。

这就需要我们写一个脚本了，遍历所有 layout 下的 xml 布局文件，列举出 layout 中常用的那些标签，将其添加到一个字典中。凡是不在这个字典中的，就算做是自定义 view。

另一种查找思路是，在使用我们的自定义 View 时，前面都必须加上我们自己的包名，例如 com.a.b.customeview，我们可以遍历所有 layout 下的 xml 布局文件，查找所有匹配 com.a.b 的标签即可。

7.3.3　针对第三方 jar 包的解决方案

我们在 Android 项目中不可避免地要使用到很多第三方提供的 SDK。一般而言，这些 SDK 都是经过 ProGuard 混淆了的。而我们所要做的，是避免这些 SDK 的类和方法在我们的 App 中被混淆。

1. 针对 android-support-v4.jar 的解决方案

```
-libraryjars libs/android-support-v4.jar
-dontwarn android.support.v4.**
-keep class android.support.v4.**  { *; }
-keep interface android.support.v4.app.** { *; }
```

```
-keep public class * extends android.support.v4.**
-keep public class * extends android.app.Fragment
```

这里介绍一下 android-support-v4.jar[⊖]。由于我们一直使用 eclipse 之类的 IDE 进行 Android 开发，IDE 会自动帮我们把 android-support-v4.jar 这个 jar 添加到 lib 目录下并进行引用，以致很多开发人员搞不清这个 jar 到底是用来干嘛的。

其实这个 jar 包是 google 提供的，全称是 Android Support Library package，它有 v4、v7 和 v13 一共 3 个版本

v4 这个包是为了照顾 Android 1.6 及更高版本而设计的，这个包是使用最广泛的，eclipse 新建工程时，都默认带有这个包。而 v7 和 v13 这两个版本向下兼容的版本很高，所以用的人不多。

android-support-v4.jar 这个 jar 包有不同的版本，所以我们在使用一些第三方 jar 包时，因为它们也用到了 android-support-v4.jar，但是版本不一样，就会在运行期抛出 NoClassDefFoundError 异常：

相应的解决办法就是将两个 android-support-v4.jar 都用一个就行了。

2. 其他的第三方 jar 包的解决方案

这个就要取决于第三方 jar 包的混淆策略了。它们会在各自的 SDK 中有关于混淆的说明文字。比如支付宝，相应的混淆规则是：

```
-libraryjars libs/alipaysdk.jar
-dontwarn com.alipay.android.app.**
-keep public class com.alipay.**  { *; }
```

不胜枚举，为了避免有 SDK 遗漏没有进行混淆处理，一个好的做法是，打开 libs 目录，看看有多少个 jar 包，每个都进行类似的处理，如图 7-3 所示。

值得注意的是，不是每个第三方 SDK 都需要 -dontwarn 指令，这取决于混淆时第三方 SDK 是否会出现警告。需要的时候再加上。

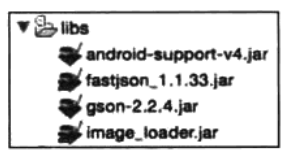


图 7-3 第三方 jar 包

7.4 其他注意事项

接下来介绍一些使用 ProGuard 过程中需要注意的事项。

1. 如何确保混淆不会对项目产生影响

如果一个 Android 项目从一开始就进行了混淆工作，那么：

- ❑ 测试工作要基于混淆包进行，才能尽早发现问题。
- ❑ 每天开发团队的冒烟测试，也要基于混淆包进行。
- ❑ 发版前，要额外测试正式版的推送、分享、打点、二维码扫描等功能。

⊖ 关于 android-support-v4.jar 的详细介绍，请参见 http://blog.csdn.net/hh2000/article/details/39718623。

2. 打包时忽略警告

当在导出时，发现很多 could not reference class 之类的 warning 信息，如果确认 App 在运行中和那些引用没有什么关系的话，可以添加 –dontwarn 标签，就不会再提示这些 warning 信息了。如：–dontwarn org.apache.**。

不要使用 –ignorewarnings 语句，它会忽略所有警告，这会有很大的潜在风险。

3. 对于自定义类库的混淆处理

回顾第 1 章，我们编写了一个 AndroidLib 类库，我们的 App 应用要引用这个类库。我们努力在做的是，把业务无关的逻辑抽离到 AndroidLib 类库中，而在 App 应用中只关心业务逻辑。

我们需要对 Lib 也进行混淆，然后在主项目的混淆文件中保留 AndroidLib 中的类和类的成员。

4. 使用 annotation 避免混淆

另一种避免类或者属性被混淆的方式是，使用 annotation。在需要保留的类中加上如下语法：

```
@Keep
@KeepPublicGettersSetters
public class Bean {
    public boolean booleanProperty;
    public int intProperty;
    public String stringProperty;

    public boolean isBooleanProperty() {
        return booleanProperty;
    }
}
```

这种使用方式多出现在 fastJSON 的使用上。

5. 在项目中指定混淆文件

说到最后，发现没有介绍如何在项目中指定混淆文件。

在项目中有一个 project.properties 文件，在其中写这么一句话，就可以确保每次手动打包生成的 apk 是混淆过的：

```
proguard.config = proguard.cfg
```

其中，proguard.cfg 是混淆文件的名称。

7.5　本章小结

本章系统全面地介绍了 ProGuard，够无聊吧。市面上没有一本书肯花这么多篇幅来介绍这些能让读者读着读着就睡着的内容。我也是醉了，花了这么多精力，干的可能是一件极其吃力又不一定讨好的事情。

衷心希望每个程序员都能练好基本功，技术本身就是件朴实无华的事情，来不得半点投机取巧。

第 8 章 Chapter 8

持续集成

持续集成（Continuous Integration），一个高大上的概念，说得简单些，就是持续提交代码、持续编译、持续测试、持续修复 bug。

持续集成一方面可以提前发现风险，另一方面，可以把构建的过程交给服务器来做，避免了本地手动打包中的各种人为错误。

本章将覆盖持续集成的几个最重要的策略：版本管理、自动构建、单元测试。

8.1 版本管理策略

版本管理策略是一个很古老的话题。业界关于这个话题的介绍不胜枚举。这里，我的讨论只限于 App 版本管理策略。App 的独特之处在于：

首先，App 是一款软件，而不是网站。所以，每次只能通过发布一个新版本的 App，来增加新功能和修复 bug，而网站当天修复 bug 当天上线。这其实是 CS 和 BS 的区别。

其次，App 因为目前的受众人群多，动则上亿的用户群，所以这就要求 App 的发版周期多，可以大约 2 周发一次新版本。这区别于传统软件慢条斯理的迭代周期。

基于上述这两点，App 的版本管理策略与以往其他项目都不同。本章和第 10 章都将围绕这个主题而展开讨论。

8.1.1 三种版本管理策略

无论是 SVN 还是 GIT，都有主干（Trunk），有分支（Branch）。相应的版本管理策略就有 3 种：

1）分支开发，分支上线。

2）主干开发，主干上线。

3）主干开发，分支上线。

策略 1：分支开发，分支上线。我带团队的时候，曾经使用过这种策略。就是说，每次迭代开始，就打一个分支，接下来一个月的迭代工作，包括开发和测试，全都在分支上进行。迭代期间，看起来没啥事，一切正常。等上线后，往主干上合并代码可就麻烦了，改了那么多文件，几千处需要合并的地方，自动合并功能我是从来不敢太相信的，一个个文件手动合并又没有时间，所以我只好把主干上的代码全都删除了，然后把分支上的代码一次性粘贴到主干上，直接签入代码。

这样做最大的问题就是，主干长时间没有嵌入，成了摆设；另一个问题是代码文件的修改历史不连贯，要到各个分支上去看。

策略 2：主干开发，主干上线。就是说，我们总是在主干上进行开发和测试。只要是本期迭代的需求，都是这么操作，直到发版上线。

策略 3：主干开发，分支上线，就是说，在主干上开发，直到写完代码，然后开分支，在分支上测试和修 bug，直到上线，最后再合并回主干，这样做的好处是要合并的代码并不多，如图 8-1 所示。

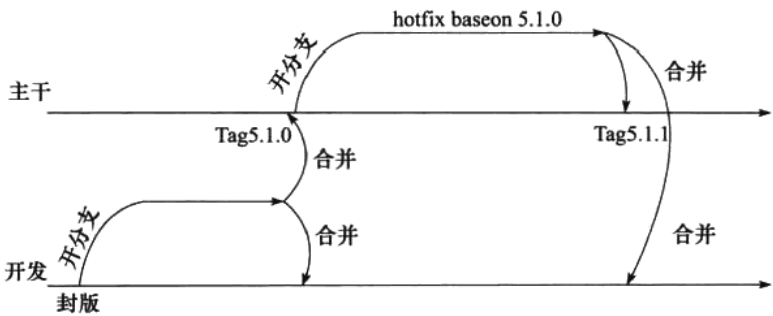


图 8-1 主干开发、分支上线的版本管理策略

策略 2 和策略 3 没有孰好孰坏的说法。下面详细介绍这两种常见的版本管理策略。

场景 1：

版本策略：主干开发，主干上线。

使用工具：SVN

迭代周期：4 周

所有开发人员都在主干上进行开发，测试也是在上面进行，直到有一天，项目经理说，我们要发版了。于是大家手忙脚乱地在主干上修改 bug，直到所有人都满意了，然后基于这个点——对应 SVN 某次提交的 ChangeSet，组织发版工作。之后，我们会基于这个点打一个 Tag，需要强调的是，一定要在注释中注明是基于哪个 ChangeSet。

SVN 没有真正的分支和 Tag，所谓的打 Tag 工作，就是基于某次提交，把代码复制一份放在一个新的目录下面。分支也是如此。

场景 2：

版本策略：主干开发，分支上线。

使用工具：GIT

迭代周期：1 ~ 2 周

迭代周期短，就会经常发生上轮迭代还没完成，下轮迭代就要开始了的情况。于是我们留一小撮人去收拾上轮迭代的遗留问题，大部队还是要在主干上进行下轮迭代的开发工作。

为此，我们要为这一小撮人开一个新的分支，让他们在上面工作，直到上一轮迭代发版上线，然后再把代码改动合并到主干上。

这样，我们就在分支上发版，并在分支上打 Tag。

GIT 比较适合干这种分支间合并代码的技术活儿，GIT 中有一个 Cherry Pick 的功能，就是干这事的。此外，GIT 中的 Tag 就是一个指针，所以不必担心又折腾出一套冗余代码的事情。

对比这两种场景，我们发现，版本管理工具的选择对选择使用哪种策略有一定的影响。SVN 本身的局限性导致了合并代码时心里会没底，需要更多的回归测试时间。

另一方面，迭代周期的长短，对使用哪种策略也有影响，尤其是项目周期发生重叠的时候。

8.1.2 特殊情况的版本管理策略

特殊情况 1：有时候，有些需求，我们发现开发有时间但是测试没有时间了，只能放到下期迭代进行，我们就在主干上找一个相对稳定的点，基于这个点开一个新分支，专门用来做这个需求，等本次迭代结束后，我们立刻就把这个分支上的功能合并到主干上，这样测试团队也可以马上测试该功能了。新分支的命名规则要规范好，能一眼看出它的功用，比如 DevForLoginBaseOn20140909，一看就知道是为了 Login 这个新需求而基于 2014 年 9 月 9 日打的分支。

特殊情况 2：接下来说到定制渠道包和手机预装的版本管理策略。如果只是简单的修改渠道号，是不需要执行版本管理策略的。但是，经常有些渠道，他们会要求我们的 App 换个闪屏页。对于在某款手机上做预装，就更麻烦了，手机厂商也会有测试人员，他们会检查我们的 App 里面的一些 bug，勒令我们修复。我们的版本管理测试是，在发给厂商的那个版本对应的 Tag 上，比如 release1.1.0，创建一个新分支，专门用于做这些小改动，测试团队验收后，打包发给渠道商和预装商。这个分支的命名规范是 channel BBB base on release1.1.0，其中 BBB 为渠道名。

特殊情况 3：最后就是上线后发现重大 bug，需要 hotfix 并紧急上线的版本管理策略。

比如说我们发布了版本 1.1.0，然后发现该版本有重大问题，需要紧急修复并上线。我们会在 release1.1.0 这个 Tag 上新建一个分支，命名为 Hotfix base on Release1.1.0，我们在这个分支上修 bug、测试并发 hotfix 版本 1.1.1。发版后，我们基于这个 hotfix 分支的稳定节点打一个新的 Tag，比如 release1.1.1。

8.2 使用 Ant 脚本打包

在开始本节的内容之前，我们先要做一些准备工作，比如说准备好一份需要安装的软件清单，如下所示：

- ❑ Ant 1.9.2
- ❑ IIS 6
- ❑ Antcontrib
- ❑ Android SDK 19
- ❑ Java SDK 1.6
- ❑ SVN
- ❑ CCNET

接下来，我们开始安装上述这些软件，需要注意以下几点：

1）事先准备一个 Android 项目 ProjectForAntBuild。

2）在服务器上安装 Java SDK。注意，请安装 1.6.0 版本的 jdk，1.7 版本的打包时会有问题。

3）在服务器上安装 Ant，版本为 1.9.2。注意，请安装带有 antcontrib 扩展的 Ant，它提供了 for 和 if 语句，能帮我们做更多的事情。

要定义 3 个全局变量，末尾记得加分号，如表 8-1 所示。

表 8-1 定义一些全局变量

全局变量名	路 径
ANT_HOME	C:\apache-ant-1.9.2
JAVA_HOME	C:\jdk1.6.0_43
CLASSPATH	%ANT_HOME%\lib;
PATH	%ANT_HOME%\bin;
PATH	%JAVA_HOME%\bin;

4）在服务器上安装 Android SDK，我的 demo 是基于 sdk-19 的，大家可以根据自己的 sdk 版本配置自己的安装包。

5）对于 Android 3.0 以上版本的 SDK，我们会发现 apkbuilder.bat 文件找不到了，我们需要上网去下载一个，或者从老版本的 SDK 把这个文件复制出来，然后粘贴到 ddms.bat 文件所在的目录中。

8.2.1 Android 打包流程

一套完整的 Android APK 打包流程如图 8-2 所示，有的同学还会在最后一步加上 adb 指

令将生成的 apk 包自动安装到手机上，这里没有包括这个步骤，因为我认为打出一个正式的安装包就算完成任务了。

打包脚本 build.xml 放在 ProjectForAntBuild 项目的根目录下，打包流程如图 8-2 所示，大家可以一边看着流程图一边看 Ant 打包脚本。

Android 打包步骤如下所示：

1）初始化。准备打包使用的目录，同时声明各种全局变量。

```
<target name="init">
    <delete dir="${outdir-gen}" />
    <delete dir="${outdir}" />
    <delete file="${basedir}/proguardMapping.txt" />

    <mkdir dir="${outdir-gen}" />
    <mkdir dir="${outdir-classes}" />
    <mkdir dir="${outdir}/${appname}" />
    <mkdir dir="${basedir}/${output.dir}" />
</target>
```

2）使用 aapt 生成 R 文件。根据 res 目录下的资源生成 R.java 文件。同时生成 AndroidManifest.xml 对应的 Manifest.java 文件。这两个文件位于 Android 项目的根目录下的 gen 子目录中。

```
<target name="aapt_gererateR" depends="init">
    <exec executable="${aapt}" failonerror="true">
        <arg value="package" />
        <arg value="-m" />
        <arg value="-J" />
        <arg value="${outdir-gen}" />
        <arg value="-M" />
        <arg value="${manifest-xml}" />
        <arg value="-S" />
        <arg value="${resource-dir}" />
        <arg value="-I" />
        <arg value="${android-jar}" />
    </exec>
</target>
```

3）aidl。将项目中的 .aidl 文件转换为 .java 代码。

```
<target name="aidl" depends="aapt_gererateR">
    <apply executable="${aidl}" failonerror="true">
        <arg value="-p${android-framework}" />
        <arg value="-I${srcdir}" />
        <arg value="-o${outdir-gen}" />
            <fileset dir="${srcdir}">
                <include name="**/*.aidl" />
            </fileset>
    </apply>
</target>
```

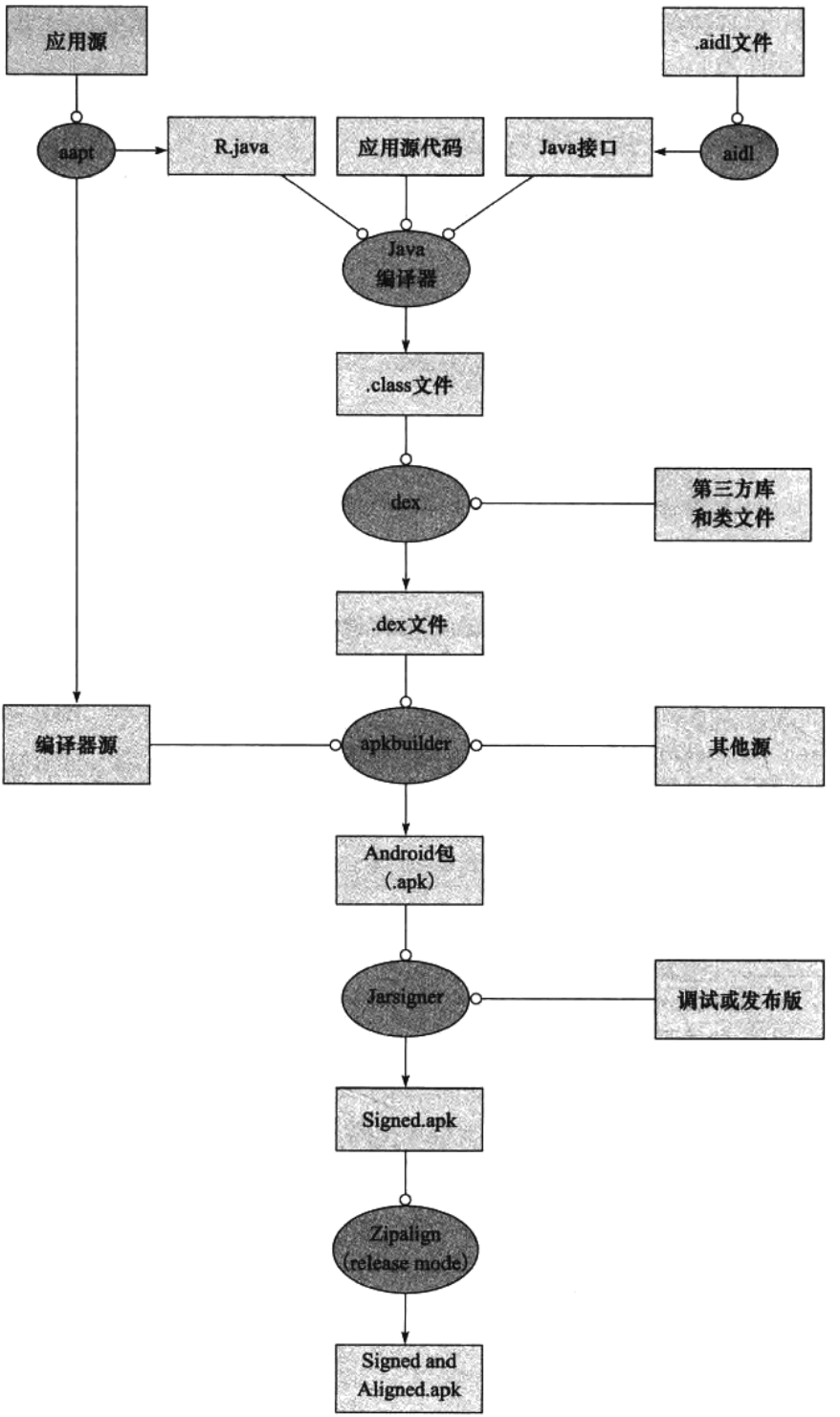


图 8-2 Android 打包流程图

4）javac。将项目中的所有 Java 代码编译为 .class 文件。

```
<target name="compile" depends="aidl">
    <javac debug="true" extdirs="" srcdir="." includeantruntime="on"
        destdir="${outdir-classes}" bootclasspath="${android-jar}"
        encoding="UTF-8">
        <compilerarg line="-encoding UTF-8 " />
        <classpath>
            <fileset dir="${external-libs}" includes="*.so" />
            <fileset dir="${external-libs}" includes="**/*.so" />
            <fileset dir="${external-libs}" includes="*/*.so" />
            <fileset dir="${external-libs}" includes="**/*.jar" />
        </classpath>
    </javac>
</target>
```

5）混淆。对项目进行混淆。同时生成 proguardMapping.txt 文件。

```
<target name="obfuscate" depends="compile">
    <jar basedir="${outdir-classes}" destfile="temp.jar" />

    <java jar="${proguard-home}/proguard.jar"
        fork="true" failonerror="true">
        <jvmarg value="-Dmaximum.inlined.code.length=32" />
        <arg value="-injars temp.jar" />
        <arg value="-outjars optimized.jar" />
        <arg value="-libraryjars '${annotations-jar}'" />
        <arg value="-libraryjars '${android-jar}'" />
        <arg value="@proguard-project.txt" />
    </java>
    <delete file="temp.jar" />
    <delete dir="${outdir-classes}" />
    <mkdir dir="${outdir-classes}" />
    <unzip src="optimized.jar" dest="${outdir-classes}" />
    <delete file="optimized.jar" />
</target>
```

6）dex。将项目中的所有 .class 文件（包括第三方库的 .class 文件）转换为 .dex 文件。

```
<target name="dex" depends="obfuscate">
    <apply executable="${dx}" failonerror="true" parallel="true">
        <arg value="--dex" />
        <arg value="--output=${intermediate-dex-ospath}" />
        <arg path="${outdir-classes-ospath}" />
        <arg path="${external-libs-ospath}" />
        <fileset dir="${external-libs}" includes="*.so" />
        <fileset dir="${external-libs}" includes="**/*.so" />
    </apply>
</target>
```

7）使用 aapt 打包资源。将 res 目录下的资源打包为一个 .ap_ 文件。注意，不要忽略了 assets 目录下的资源。

```
<target name="aapt-package-res" depends="dex">
    <echo>Packaging resources and assets…</echo>
```

```
    <echo>${resource-dir}</echo>
    <exec executable="${aapt}" failonerror="true">
        <arg value="package" />
        <arg value="-f" />
        <arg value="-M" />
        <arg value="${manifest-xml}" />
        <arg value="-S" />
        <arg value="${resource-dir}" />
        <arg value="-A" />
        <arg value="${asset-dir}" />
        <arg value="-I" />
        <arg value="${android-jar}" />
        <arg value="-F" />
        <arg value="${resources-package}" />
    </exec>
</target>
```

8）apkbuilder。将所有的 dex 文件、ap_ 文件、AndroidManifest.xml 打包为 .apk 文件，这是一个未签名的包。

```
<target name="apkbuilder" depends="aapt-package-res">
    <exec executable="${apk-builder}" failonerror="true">
        <arg value="${out-unsigned-package-ospath}" />
        <arg value="-u" />
        <arg value="-z" />
        <arg value="${resources-package-ospath}" />
        <arg value="-f" />
        <arg value="${intermediate-dex-ospath}" />
        <arg value="-rf" />
        <arg value="${srcdir-ospath}" />
        <arg value="-nf" />
        <arg value="${external-libs-ospath}" />
        <arg value="-rj" />
        <arg value="${basedir}\${external-libs}" />
    </exec>
</target>
```

9）jarsigner。对 apk 进行签名。

```
<target name="jarsigner" depends="apkbuilder">
    <exec executable="${jarsigner}" failonerror="true">
        <arg value="-verbose" />
        <arg value="-keystore" />
        <arg value="${key.store}" />
        <arg value="-storepass" />
        <arg value="${key.store.password}" />
        <arg value="-keypass" />
        <arg value="${key.alias.password}" />
        <arg value="-signedjar" />
        <arg value="${out-signed-package-ospath}" />
        <arg value="${out-unsigned-package-ospath}" />
        <arg value="${key.alias}" />

        <arg value="-digestalg" />
```

```
        <arg value="SHA1" />
        <arg value="-sigalg" />
        <arg value="MD5withRSA" />
    </exec>
</target>
```

10）zipalign。对要发布的 apk 文件进行对齐操作，以便在运行时节省内存。

```
<target name="zipalign" depends="jarsigner">
    <exec executable="${zipalign}" failonerror="true">
        <arg value="-v" />
        <arg value="-f" />
        <arg value="4" />
        <arg value="${out-signed-package-ospath}" />
        <arg value="${zipalign-package-ospath}" />
    </exec>
</target>
```

至此，Ant 的 build 脚本都已经介绍完毕，我们只要执行下列语句，就可以对 Android 项目进行打包：

```
c:\ProjectForAntBuild>ant -buildfile build.xml
```

注意，上述 Ant 脚本打出来的包是签名包。

8.2.2 打包时的注意事项

容我再多说几句，以下内容是我在日常打包过程中的经验总结。

1）打包工作是件很枯燥的事情，一定要细心，要多使用 echo 输出日志。在 cmd 中看日志的问题是，一旦日志内容多了，前面的日志会被冲掉，所以请使用标签，把日志记录到本地文件中：

```
<project name="apkTargets" default="zipalign" basedir=".">
    <record name="C:/build.log" loglevel="info" append="no" action="start" />
```

2）一定要确保打包服务器上的 Android SDK 版本与开发人员所使用的开发版本一致。尤其是 proguard 程序，版本低了，会导致混淆不能进行。

3）如果打包机器上安装了杀毒软件，它会妨碍 Android 的打包工作，尤其是 dex 文件，会被视作一个病毒，所以 apkbuilder 会不能正常执行。切记，在打包机器上，一定要把杀毒软件关闭。

4）有时，我们需要打未签名的包，于是我们在上述打包脚本 build.xml 中补充以下语句：

```
<target name="debug" depends="aapt-package-res">
    <exec executable="${apk-builder}" failonerror="true">
        <arg value="${out-debug-package-ospath}" />
        <arg value="-z" />
        <arg value="${resources-package-ospath}" />
        <arg value="-f" />
```

```
        <arg value="${intermediate-dex-ospath}" />
        <arg value="-rf" />
        <arg value="${srcdir-ospath}" />
        <arg value="-nf" />
        <arg value="${external-libs-ospath}" />
        <arg value="-rj" />
        <arg value="${external-libs-ospath}" />
    </exec>
</target>
```

这条语句与前面介绍的打包流程第 8 步虽然都使用了 apkbuilder 指令，但是参数略有不同，所以打出来的包是未签名的。我们将 Ant 中 project 标签的 default 属性改为 debug，执行以下指令即可：

```
c:\ProjectForAntBuild>ant -buildfile build.xml
```

8.3 Monkey 包的生成

在打包这个工具做好之后，运行 build.xml 脚本就能得到一个经过签名混淆的 apk 包，这与最终发版上线打包的机制是一样的。

在发版前，我们经常要对 App 进行 Monkey 测试，由于 Monkey 是乱点的，所以我们要防止它执行以下几个操作：

1）点击拨打电话的按钮，从而跳出 App。

2）进入支付流程，这样会生成很多无效的订单。

这就要求我们要在程序中设置一个开关 isMonkey，只有打 Monkey 包时这个值才为 true，考查 ProjectForAntBuild 项目中下面的代码：

```
public interface Config {
    public final static boolean isMonkey = true;
}
```

在 MainActivity 中，使用这个 isMonkey 开关控制电话按钮是否禁用，如下所示：

```
Button btnPhone = (Button)findViewById(R.id.btnPhone);
btnPhone.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if(!Config.isMonkey) {
                startActivity(
                    new Intent(Intent.ACTION_DIAL,
                        Uri.parse("tel:13000000000")));
            }
        }
    });
```

在打包阶段，生成 Monkey 测试包的脚本时，就把 isMonkey 这个值设为 true，而生成要

发布到线上的正式包时，又要把这个 isMonkey 值设为 false。

我们希望执行一次脚本，就同时打出这两个包来。于是我们在 build.xml 这个 Ant 脚本之外，新建了一个 dailybuild.xml 脚本，由它来修改 isMonkey 的值，然后调用我们之前编写的 build.xml 脚本。先生成正式包，后生成 Monkey 包，脚本中的关键代码如下所示：

```
<target name="begin">
    <!-- 正式签名包，关闭 monkey 开关 -->
    <close_monkey />
    <generateApk />

    <!-- Monkey 包，打开 monkey 开关 -->
    <open_monkey />
    <generateApk-monkey />
</target>
```

generateApk 和 generateApk-monkey 的实现基本上是相同的，唯一的区别是在 copy 时生成不同的文件名，然后转移到同一个目录下。

执行下述脚本，就能同时生成两个 apk 安装包：

```
c:\ProjectForAntBuild>ant -buildfile dailybuild.xml
```

8.4 自动打包

如何判断一个公司的无线 App 技术水平是作坊式开发，还是企业级开发？其中很重要的一个指标就是 App 是否支持自动打包。

对于只有几个人的软件作坊，往往是测试人员找开发人员用 Eclipse 打一个包，安装在测试机上，然后进行测试。这种手动打包的方式问题很多，经常发生测试人员发现新包有问题，然后又去找开发人员检查问题，重新打包。这样往返几次，极大地浪费了开发人员和测试人员的时间。

新包有问题一般是因为：开发人员没有获取最新的代码就进行打包工作了，于是其他人提交的代码和功能不在这个包中。另一方面，如果有人提交了不能编译的代码，会导致其他开发人员更新代码后不能编译调试。

想解决这些问题，只能引入自动打包机制，大致的思路是：

1）我们需要有一台打包服务器，它能从代码服务器自动获取最新的代码、编译、打包，发邮件通知团队成员打包结果。

2）提供一个大家都可以访问的 Web 页面，为不同项目建立不同的打包机制。要同时提供自动和手动两种触发打包的方式。

自动打包，也就是 Daily Build，每天设定一个时间，一般是深夜大家都下班的时间。自动打包可以确保如果打包失败，会发邮件通知，第二天上班，会有人立刻修复导致编译不通过的 bug。

手动打包，为测试人员提供一个“打包”按钮，这样他们就可以根据需要随时打包，比如开发人员提交代码修复了一个 bug，测试人员要验证这个 bug，就在上述的 Web 页面上点击“打包”按钮就可以了。

3）在这台测试服务器上部署 Web 服务器，可以浏览每天打出的安装包清单，从而可以直接下载任意安装包并安装到测试机上。

基于此，我们选用 CCNET 这个工具。CCNET 提供手动打包的按钮，以及自动打包的设置。CCNET 来驱动 Ant 执行打包脚本进行打包工作。因为 CCNET 仅支持在 Windows 环境安装，所以我们选用 Windows 2003 作为我们的打包服务器。同时，我们在这台服务器上安装 IIS，使包的存放地址可以通过 http 进行访问。当然，你也可以选用别的服务，比如 Tomcat。

接下来我将详细介绍怎样组装这些技术和工具，搭建出我们想要的自动化打包机制。

8.4.1 安装和配置各种软件

安装步骤如下：

1）在服务器上安装 Java SDK。注意，请安装 1.6.0 版本的 jdk，1.7 版本的打包时会有问题。

2）在服务器上安装 Ant，版本为 1.9.2。注意，请安装带有 antcontrib 扩展的 Ant，它提供了 for 和 if 语句，能帮我们做更多的事情。

定义 3 个全局变量，末尾记得加分号，如表 8-2 所示。

表 8-2 定义一些全局变量

全局变量名	路 径
ANT_HOME	C:\apache-ant-1.9.2
CLASSPATH	%ANT_HOME%lib;
PATH	%ANT_HOME%bin;

3）在服务器上安装 Android SDK，我的 demo 是基于 sdk-19 的，大家可以根据自己的 sdk 版本配置自己的安装包。

4）在服务器上安装 IIS。

5）在服务器上安装 .NET Framework 3.5 或以上版本。

6）到 CCNET 官方网站下载 CCNET 的最新版本，目前为 1.8.5。

定义 1 个全局变量，末尾记得加分号，如表 8-3 所示。

表 8-3 定义一些全局变量

全局变量名	路 径
PATH	%ANT_HOME%bin;
ANT_HOME	C:\Apache-Subversion-1.8.10\bin;

注意，在安装 CCNET 之前，请确保已经安装了 IIS。

8.4.2 准备 Ant 打包脚本

我们仍然使用上一节介绍的 daily.xml 脚本，它将生成两个包，正式包和 Monkey 包。如果大家还想生成其他的包，只需要配置 dailybuild.xml 脚本即可，在打包前使用正则表达式修改某个文件的值。

因为 CCNET 目前不支持直接执行 Ant 脚本，所以我们要额外编写一个 bat 脚本，由 CCNET 通过执行 bat 文件来间接执行 Ant 脚本 dailybuild.xml。

这个 bat 脚本的内容如下，我们将其命名为 dailybuild_1.1.0.bat：

```
ant -file C:\Source\ProjectForAntBuild_1.1.0\dailybuild.xml
-D app.source.path="C:\Source\ProjectForAntBuild_1.1.0"
```

8.4.3 配置 CCNET

CCNET 的关键就在 ccnet.config 这个配置文件上，它位于以下目录中：

```
C:\Program Files\CruiseControl.NET\server
```

我们使用 CCNET 主要做 3 件事情：

- 根据 SVN 地址获取相应的代码。
- 执行打包脚本。
- 发邮件通知，定制成功和失败两种情况下的邮件格式。

8.4.4 搭建 IIS 站点下载 apk 包

执行 CCNET 每日自动打包，日积月累，在存放打包文件的目录下将存在大量的子目录，如图 8-3 所示。

我们需要提供一个内部的 Web 站点，指向 ProjectForAntBuild 这个目录，从而公司内部的所有同事随时都可以下载 apk 进行测试。

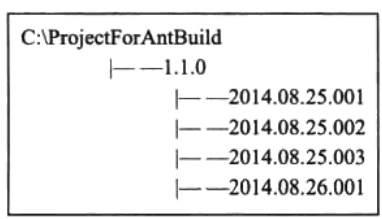


图 8-3 ProjectForAntBuild 目录下的子目录

配置 CCNET 和 IIS

原本写了 8 页来介绍如何配置 CCNET 和 IIS，后来发现这与本书主题不符，于是就把这部分内容上传到我的博客空间，请访问以下地址下载这份配置文档：

http://files.cnblogs.com/files/Jax/config.zip

8.4.5 自动打包流程小结

至此，一套自动打包的流程机制全都介绍完毕。最后补充一下，如果过渡到下一次迭

代，版本从 1.1.0 变为 1.2.0，我们又要在自动打包中做哪些工作呢？

1）在 C:\ProjectForAntBuild\bat 目录下，新建一个 dailybuild_1.2.0.bat 文件，内容与 dailybuild_1.1.0.bat 类似，只是要修改传递到 Ant 脚本的参数，如图 8-4 所示。

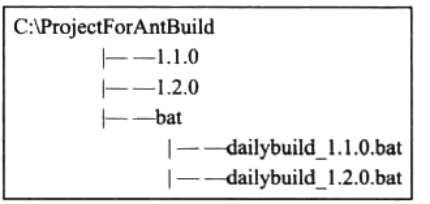


图 8-4 在 bat 目录下新建一个 dailybuild_1.2.0.bat 文件

2）修改源代码中 AndroidManifest.xml 文件中的版本号。

3）修改 ccnet.config 文件，在里面新增一个 project，可以复制一份 1.1.0 版本的 project 节点内容，但是其中的 1.1.0 要全都改为 1.2.0。

8.5 批量打渠道包

所谓的渠道包，从代码层面讲，就是 AndroidManifest 中的 UMENG_CHANNEL 这个 key 的值，将其替换为相应的渠道号，比如 360 市场，这个值就是 360Android，然后再进行打包。

从商业角度讲，360Android 这个渠道号是财务部门用来和 360 市场做结算的，我们每月会根据友盟上 360Android 这个渠道有多少下载量（或激活量），来向 360 公司支付相应的推广费用，于是无线推广部门应运而生，他们的一部分工作就是干这个事情，有的渠道包是手动上传到各大市场，有的渠道包是分发给市场的工作人员，由他们帮忙发布。

除了发布到各大市场，渠道包的另一种出现场景是，外链。比如说公司网站首页上会提供下载；比如说推广活动的 Html 链接；比如说公交车站、电梯上的二维码（它其实也是一个 Html 链接）。我们会把这些链接对应的渠道包都放在公司的服务器上，以提供下载。

我们需要建立批量打渠道包的机制。目前，批量打渠道包有两种方式，接下来我会逐一介绍。

8.5.1 基于 apk 包批量生成渠道包

基于一个 apk 包，我们将其反编译，然后遍历渠道列表获取每一个渠道号，修改 AndroidManfest.xml 中的渠道号后，重新进行打包工作，包括签名、混淆和对其操作，如图 8-5 所示。

对图 8-5 中的打包流程详细分析如下：

1）反编译 apk 文件。

```
apktool.bat d --no-src -f "C:\jianqiang_app.apk" "temp"
```

反编译 apk 文件后，在 temp 目录中能看到 AndroidManifest.xml 文件。

2）for 循环渠道列表，逐个打渠道包。

2.1）替换 AndroidManifest.xml 中的渠道号。

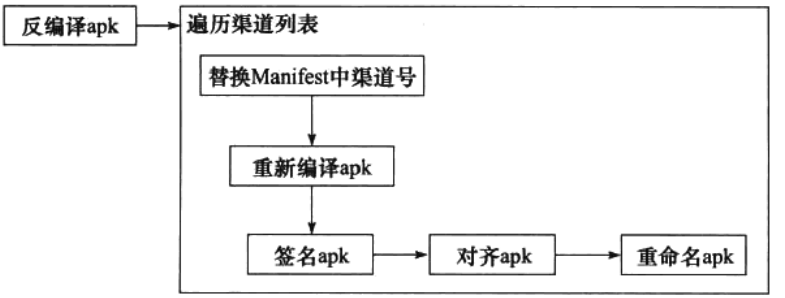


图 8-5 基于 apk 包批量生成渠道包的流程图

2.2）将反编译后的文件重新编译成 apk，放到 apk_temp 目录下：

```
apktool.bat b "temp" "apk_temp\unsigned-App 名称 .apk"
```

2.3）签名 apk 文件。

```
java -jar SignApk.jar "C: \a.b"
    "123456" "JianqiangApp" "123456"
    "apk_temp\unsigned-appname.apk"
    "apk_temp\unzipAligned-appname.apk"
```

2.4）对要发布的 apk 文件进行对齐操作。

```
zipalign.exe -v 4
    "apk_temp\unzipAligned-App 名称 .apk"
    "apk_temp\App 名称 - 渠道号 .apk"
```

2.5）把打好的渠道包重命名为：App 名称 _ 渠道号 .apk，然后将其转移到 output\App 名称 \App 名称 _ 渠道号 .apk。

上述这些操作都有相应的 Android SDK 命令，我们可以使用任何语言编写一个程序来一次执行这些命令。

这里我们可以使用友盟提供的渠道批量打包工具，它是基于 C# 来实现的上述批量打包流程。[⊖]

8.5.2 基于代码批量生成渠道包

对于基于代码进行打包的方法，我们在前面的章节介绍过 build.xml，但是这个脚本每次只能打一个包，我们需要写一个新的脚本 batchbuild.xml，它会依次执行以下操作：

1）读取 channel.txt 文件中的渠道列表。

```
<target name="foreach_manager">
    <loadfile property="listVerText"
        srcfile="${app.source.path}/${channel.filename}"
        encoding="GBK" />
```

⊖ 该工具下载地址：https://github.com/umeng/umeng-muti-channel-build-tool。

```
    <propertyregex override="true" property="apk-channel"
        input="${listVerText}" regexp="\r\n" replace=";"/>

    <!— 遍历 ${apk-channel} 打包 —>
    <foreach target="build-apk" param="channelName"
        list="${apk-channel}" delimiter=";"/>
</target>
```

2）执行 for 循环语句，执行 build.xml 脚本文件中 build-apk 这个 target。

2.1）替换 AndroidManifest.xml 中的渠道号。

2.2）执行 build.xml 脚本进行打包。

2.3）将打好的包并转移到指定的目录 build/1.1.0 下面，1.1.0 为版本号。

2.4）清理打包过程中生成的临时文件，为打下一个渠道包做准备。

```
<target name="build-apk">
    <echo>build-path:${build-path}, 目录 :${channelName} ,
        渠道 :${channelName}</echo>

    <!-- 创建放 APK 的目录 (FFFFFF 就是为了进行一次字符串的 override 操作 ) -->
    <propertyregex override="true" property="build-path"
        input="${build-path}/${channelName}FFFFFF"
        regexp="FFFFFF" replace="" />
    <echo> 创建目录 :${build-path}</echo>
    <mkdir dir="${build-path}" />
    <!-- 替换 Manifest 中的 UMENG_CHANNEL 字段 -->
    <replaceregexp file="AndroidManifest.xml"
        match="(android:name=&quot;UMENG_CHANNEL
            &quot;\s+android:value=&quot;)(.*)(&quot;)"
        replace="\1${channelName}\3"
        encoding="UTF-8"
        byline="false"/>

    <!-- 开始打包 -->
    <ant antfile="build.xml" inheritAll="true" target="zipalign" />

    <!-- 移动 APK 至相应目录 -->
    <copy file="${basedir}/${output.dir}/${appname}_for_android_
                ${android_version}_${temp.dir}.apk"
        tofile="${build-path}/${appname}_${appversion}_
                ${channelName}.apk" />

    <!-- 清理生成的临时文件，为 build 下一个渠道包做准备 -->
    <cleanTmpFolder />
</target>
```

上述流程如图 8-6 所示。

我们只要执行下述命令，就可以批量打渠道包了：

```
c:\ProjectForAntBuild>ant -buildfile batch_build.xml
```

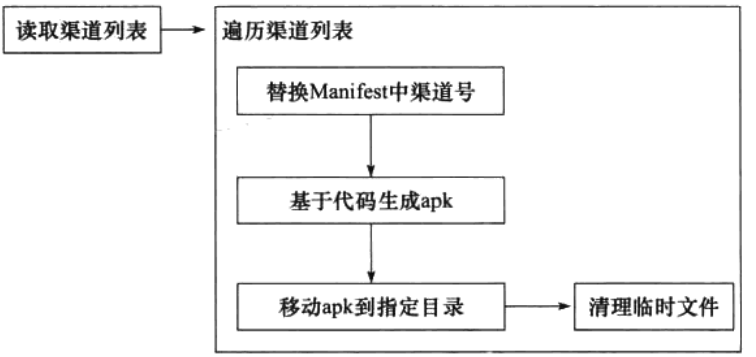


图 8-6 基于代码批量生成渠道包的流程图

8.6 Android 发版流程

前面已经絮絮叨叨地说了很多发版相关的事情，这里做一下总结：

假设即将发布 1.1.0 版本，apk 的名字是 ProjectForAntBuild。

1）远程登录到这台批量打渠道包工具所在的服务器上，假设是 192.168.1.14。

2）将项目的代码从 SVN 或者 GIT 手动签出，放在 C 盘根目录下。

3）在 Android 项目的 AndroidManifest.xml 中，修改以下两个地方并提交：

```
android:versionCode= "110"
android:versionName= "1.1.0"
```

4）执行批量打渠道包的命令：

```
c:\ProjectForAntBuild>ant -buildfile batch_build.xml
```

打包后，生成的 apk 文件都会放在 C:\build\1.1.0 目录下面。

在渠道包全都打完之后，随机选取其中 1-2 个包进行测试，需要检查几个地方，如表 8-4 所示。

表 8-4 对渠道包进行抽样测试

步 骤	检查方法
1. 版本号	将 apk 文件后缀改为 zip，解压，观察其中的 AndroidManifest.xml 文件，如果 versionCode 和 versionName 与所要发布的版本号一致，就认为是正确的
2. 渠道号	检查方法同步骤 1，只是这次检查的是渠道号是否与所要发布的渠道包一致，如下所示： <meta-data android:name= "UMENG_CHANNEL" android:value= "360android" />
3. 是否混淆	需要反编译这个包，看代码是否有混淆

（续）

步　骤	检查方法
4. 检查配置文件中开关是否正常	1）是否可以下单和打电话，如果不能，说明是 Monkey 包，是有问题的 2）Menu 中是否有切换服务器的按钮，如果有，说明是测试包，是有问题的 3）是否可以唤起微信支付，是，证明是签名包；否则是有问题的
5. 检查主流程是否可以走通	个人中心是否可以登录。如果有支付流程，要下一个单并支付以验证主流程是否正常

8.7 分类打渠道包

每次 Android 发版都要打几百个渠道包，把这些渠道包都放在一个目录下，对于推广人员来说是一种灾难。本节我们要研究如何把这几百个渠道包分门别类放在合适的地方。

8.7.1 分门别类生成渠道包

根据我的经验，渠道包基本分为 4 类：

1）需要我们自己的推广人员手动上传到各大市场的渠道包。

2）HTML5 短链接上提供下载的渠道包。

3）交付给第三方 Android 市场的工作人员，由他们帮忙更新。

4）需要额外定制的渠道包。

其中，第 4 类不列入批量打渠道包的清单中。因为这种渠道包有额外定制的功能，每次都是在某个稳定版本的基础上修改一些功能后单独打包，然后交付给推广人员即可。

在实际操作中，我们发现，前 3 类渠道包，是有优先级顺序的，一般而言，在发版当天，第 1 类和第 2 类渠道包就要同步更新了，第 3 类可以放在夜里进行打包，第二天再发给推广人员就可以了。

我们之前编写的 batchbuild.xml 太一厢情愿了，它把所有的渠道包全都打出来而不会进行分类，这对于市场人员太痛苦了，而我们开发人员的工作就是要救世人于水火之中，所以我们将原先的 channel.xml 按类别拆分为 3 个文件，分别存放以上 3 类渠道列表：

1）channel_manual.txt，存放需要手动上传的包。

2）channel_h5.txt，存放 HTML5 短链上的包。

3）channel_tomorrow.txt，存放第二天再上传的渠道包。

我们在 batchbuild.xml 的外面做了一层包装，也就是 batch_build_ext.txt，其中 ext 是扩展的意思，它会先后读取以上 3 个存放渠道列表的 txt 文件，然后进行批量打包工作。

batch_build_ext.xml 脚本的关键代码如下：

```
<target name="foreach_manager_all">
    <!-- 根据 channel_manual.txt 进行打包 -->
    <var name="channel.filename" value="channel_manual.txt" />
```

```
    <var name="build-path" value="C:\build\${appversion}\manual" />
    <ant antfile="batch_build.xml" inheritAll="true" />

    <!-- 根据 channel_h5.txt 进行打包 -->
    <var name="channel.filename" value="channel_h5.txt" />
    <var name="build-path" value="C:\build\${appversion}\h5" />
    <ant antfile="batch_build.xml" inheritAll="true" />

    <!-- 根据 channel_tomorow.txt 进行打包 -->
    <var name="channel.filename" value="channel_tomorrow.txt" />
    <var name="build-path" value="C:\build\${appversion}\tomorrow" />
    <ant antfile="batch_build.xml" inheritAll="true" />
</target>
```

我们只要执行下面的脚本就可以批量生成渠道包了：

```
c:\ProjectForAntBuild>ant -buildfile batch_build_ext.xml
```

生成的目录格式如图 8-7 所示。

```
C:\build
    |----1.1.0
    |       |----h5
    |       |----manual
    |       |----tomorrow
    |----1.2.0
```

图 8-7 批量生成渠道包的目录结构

8.7.2 批量上传 apk 的两种方式

每次发版时，推广人员都要手动上传所有的 apk 包到市场。对于推广人员而言是非常痛苦的事情。[⊖]

为了把推广人员解脱出来，我们经过调研，发现市面上有很多这样的一键式提交工具，我们预先把这些市场的账户和密码输入到这个工具中，就可以一劳永逸了。当然这期间还有如何输入更新信息、不同渠道上传不同的渠道包等若干问题，这就都是细节了。

另一方面，推广人员还要手动更新所有的 HTML5 短链接。每次都有 100 多个，要耗费大量的人力。经过调研，我们发现，其实这也是可以实现自动化的。我们需要写一个工具，批量更新 HTML5 短链接上的 apk 包。事先需要规定好渠道包的命名规范，如下所示：

```
渠道号 _ 版本号 _App 名称 .apk
```

例如：ProjectForAntBuild _1.1.0_360android.apk

那么我们的批量打渠道包工具，就会按照这个约定，在一个目录下生成 HTML5 短链接所需要的所有 apk。然后推广人员点击“发布”按钮，就可以把所有的 HTML5 短链接都更新为最新的版本。

8.8 灵活切换服务器

我们在开发 App 功能的时候，会使用到 MobileAPI 提供的接口。但实际的情况是，在我

⊖ 详细信息请参见博客园“谦虚的天下”的文章《App 应用之提交到各大市场渠道》，地址如下：http://www.cnblogs.com/qianxudetianxia/archive/2012/12/05/2803894.html。

们开发 App 新功能的时候，这些接口有可能还没有上线，仅仅在测试环境可以使用。

一种方法是把不同环境的 IP 写到配置文件中，每次打包时指定其中一个环境的 IP。但这样的缺点是每个包只能针对于一种环境。

对于 Android 我们可以这么做，在 Menu 里加入 IP 的列表，点击其中一项后将会把全局变量 Globals.IP 设置为相应的 IP。

为了每个页面都能切换服务器 IP，我们将这个逻辑封装到基类 BaseActivity 中：

```
public class BaseActivity extends Activity {

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);

        if (Config.isDebug) {
            getMenuInfiater().infiate(R.menu.activity_main, menu);
        }

        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
      switch (item.getItemId()) {
        case R.id.menu_ip1:
            Globals.IP = "http://212.1.2.3";
            break;
        case R.id.menu_ip2:
            Globals.IP = "http://192.168.1.14";
            break;
        case R.id.menu_ip3:
            Globals.IP = "http://192.168.2.28";
            break;
        default:
           return super.onOptionsItemSelected(item);
        }
        return true;
    }
}
```

这样做的好处是，在任何页面都可以通过 Menu 切换 IP，从而连接不同的环境，马上就会生效。

当然，为了避免正式版也有这个功能，需要在 Config 文件中增加一个开关 isDebug，只有这个值为 true 时，才能在 Menu 中看到那个按钮。

相应的，要修改 dailybuild.xml 和 batch_build.xml 文件，以控制这个 isDebug 开关。这里就不再多说了，原理和前面介绍过的开关 isMonkey 相同。

8.9 单元测试

“春色满园关不住，一枝红杏出墙来。”之所以想起这两句，是因为虽然最近这两周项目紧张，我们一直在赶进度，但是忙里偷闲，我们还是做了一件对 Android 项目而言很有意义的事情，那就是单元测试。

开始讲述我的故事之前，先来扫扫盲：

- 什么是单元测试？请参见文章：http://baike.baidu.com/link?url=DtllYiDKetRaM2z1uKgLG_BDGYDYU3gNFzOQnd13i9k7lqnLHEelYuoAVd0WwYMy。
- TDD（在微软时，我们戏称为“踢弟弟”）请参见文章：http://www.ibm.com/developerworks/cn/linux/l-tdd/index.html。
- Android 单元测试的相关文章。请参见文章：http://www.oschina.net/question/54100_27061。
- iOS 单元测试的相关文章。请参见文章：http://blog.csdn.net/fengsh998/article/details/8109293。

有人认为单元测试（UT）是测试人员写的，错！大错特错！！测试人员可以写 TestCase，写 UAT case，但就是写不来 UT。UT 是开发人员写的。

面试时发现，绝大多数的 App 开发人员，没有写过单元测试，原因有三：

- 在客户端这个领域，业界没有写 UT 的风气。
- 基于 UI 的单元测试，不知道怎么写。
- 高强度开发，没有时间写 UT。

其实，在客户端写单元测试的好处有很多：

- 对蕴藏在客户端中的复杂逻辑或者算法，如果有相应的单元测试，可以确保每次小的逻辑变动，而不用再手动测试其他情况，只需要跑一遍所有的 UT 即可。
- 单元测试要求编码时将 UI 与业务逻辑相剥离。但凡做不到的，都是代码写的有问题，耦合性太高。

但是，绝对不能以偏概全，为客户端的所有代码都加上单元测试，那是不现实的。我的经验是，只为那些复杂的业务逻辑（有很多 if-else 分支语句）写单元测试。

下面以验证身份证号码是否有效作为例子，来介绍如何编写单元测试。项目请参见我博客上的源码[⊖]。

身份证的业务规则如下所示：

1）15 位或 18 位长度。

2）15 位，必须全数字。

3）18 位，前 17 位必须全数字。

4）检查出生日期是否为有效的日期。注意 18 位和 15 位的取值规则是不一样的。

5）检查 18 位的最后一位是否有效（这个值有可能是 X）。

上述业务逻辑的实现，请参见 Utils 类的 isIdCardNumberValid 方法。可以看到这个方法

⊖ 下载地址：http://www.cnblogs.com/Jax/p/4656789.html。

非常复杂，有太多的 if-else 逻辑判断。动一动牵发全身，导致后面的逻辑有问题。代码量很大，由于我这一节介绍的是单元测试，所以就不贴出来了，大家可以去 TestCode 项目下去看具体的实现。

如果我们想增加一个新的业务规则，或者发现某个 bug 而对上述某个规则进行了修改，那么该如何确保其他业务规则不受影响呢？

只有单元测试能解决这个棘手的问题。

于是我们为每条业务规则都准备了若干单元测试用例，每次做出修改，都把这些用例全都执行一遍，这些用例集中放在 TestIdCard 类的 testIdCard 方法中，如下所示（截取部分代码）：

```
public void testIdCard() throws Exception {
    // 测试长度为 0 或者输入为空的情况
    Assert.assertEquals(AppConstants.IDCARD_LENGTH_SHOULD_NOT_BE_NULL,
            Utils.isIdCardNumberValid("").getIdCardDesc());
    Assert.assertEquals(AppConstants.IDCARD_LENGTH_SHOULD_NOT_BE_NULL,
            Utils.isIdCardNumberValid(null).getIdCardDesc());

    // 测试长度不为 15 或者 18 的情况
    StringBuilder idCard = new StringBuilder();
    for (int i = 0; i < 20; i++) {
        idCard.append("1");

        if (idCard.length() == 15 || idCard.length() == 18)
            continue;
```

图 8-8 是 Android 单元测试用例的执行结果，标记√的表示单元测试通过，标记 × 表示测试不通过：

我们看到，具体错误发生在 testIdCard 这个方法上，双击它能定位到具体有问题的测试代码，一路跟踪到 Utils 类的 isIsCardNumberValid 方法，发现问题出在对身份证号码的最后一位校验上，代码中逻辑仅支持小写的 x，对大写 X 并不支持。

把这个问题上升到需求层面，对于用户而言，输入身份证号码是不要去区分大小写的，所以这确实是一个 bug，于是我们修改这个逻辑，比较时不区分大小写。再次运行单元测试，如图 8-9 所示，可以看到所有测试用例都通过了。

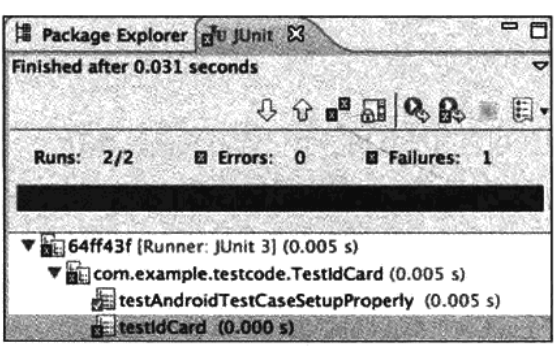


图 8-8 单元测试的执行结果

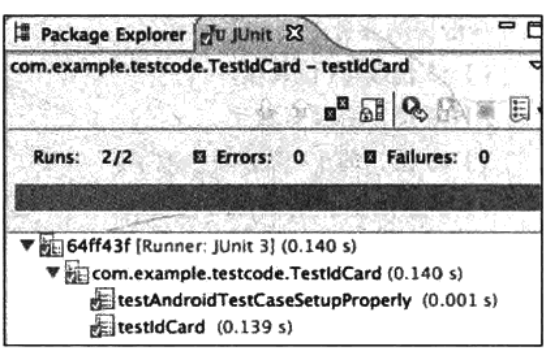


图 8-9 单元测试的执行结果

通过编写单元测试发现 bug、修复 bug 的例子，证明了单元测试是确实有很大帮助的。

说起单元测试，往事历历在目，有甜蜜，有心酸。接下来是八卦时间，大家可以去抢沙发和板凳了。感谢读者们花钱买我写的书，接下来我给大家分享一个苦逼程序员的故事。

话说我每天加班都要到晚上 10 点多，终于有一次约到了女神吃饭，我还清晰地记得那是第一次下班的时候天还亮着。6 点半我已经在出租车上了，车上还坐着我的一个兄弟，他要蹭我的车去地铁站。快到目的地的时候，女神微信我说已经在餐厅排队等位子了，再后来跟我说已经排到了位子就等我过去了。这时悲催的事情发生了，还在公司的兄弟打电话来说线上出事了，有个模块频繁崩溃。我当时好纠结啊，去约会还是回公司？最后还是咬咬牙，让司机调头开回公司。我还记得在出租车上和女神解释放鸽子的原因的时候，女神只回了我六个句号，然后就再也没有然后了。

当然和我同车的那个兄弟也同样悲催，因为他被我带回了公司一起查问题。多年之后，我们喝酒时说起这件事，仍然感慨万千。

我们到公司后发现，问题时有时无，并不稳定重现。那是一个用 Comparator 实现的排序算法，数据源来自 MobileAPI，我们要把其中状态为 0 的数据都排到前面，状态为 1 的数据都排到后面。

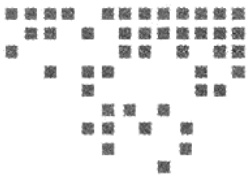
但是 Comparator 排序算法写的有问题，而这个问题很隐蔽，仅在某些特定的情况下才会发生崩溃，而我们在做功能测试时，并不包括那些特殊的测试场景，所以只有等到发版后根据线上的真实数据发现问题了。

想要规避这种情况的发生，只有写单元测试。由开发人员准备各种测试数据，以证明算法的正确性。

8.10 本章小结

本章介绍持续集成。持续集成是个很宏大的概念，本章只涉及了版本管理策略、打包、单元测试这几部分。

本章的知识比较零碎，看似和 Android 日常开发工作关系不大，所以很多程序员不愿意涉及这个领域，他们更愿意埋头写几个 Activity。殊不知，掌握了本章的这些技能，才能完成从小工到技术大牛——思想上的飞跃。



第 9 章 Chapter 9

App 竞品技术分析

我仔细研究了市面上百款 App 的技术实现。管窥到很多先进的思想和技术，总结到本章中，内容很多，如安装包的结构与大小、开机速度、HTML5 页面的打开速度、性能优化、数据采集工具、ABTest、热修补、模块化拆分等。希望抛砖引玉，使各个公司能意识到竞品分析这个重要领域，成立专门团队，从产品和技术两个维度进行竞品分析的研究工作。

9.1 竞品分析概述

9.1.1 App 竞品定义

我们通常将同行业内竞争对手的产品定义为竞品，所以竞品分析通常就是分析竞争对手的产品。

对于 App 而言，这样定义竞品还远远不够。同行业内的竞品固然重要，但是对于行业外的优秀 App，对我们而言，也是有很大参考意义的。比如：

- 社区类和视频类 App，他们的广告系统做得是最好的，因为他们就靠在 App 中投放第三方公司的广告来赚取广告费，这是他们生存的手段，所以一定是花大力气做的。
- 电商类（包括 OTA 和 O2O）App 的产品详情页和订单填写页做得是最好的，因为他们要确保订单转化率就靠产品详情页来吸引用户眼球，靠订单填写页良好的用户体验来促进用户下单。
- 活动运营做得最好的仍然是电商类 App。就靠首页的那几个轮播广告位，能做出各种意想不到的促销效果。此外，各种秒杀、满减，也都是电商类 App 的拿手好戏。

- 社交类 App 的聊天功能做得是最好的，尤其是高并发的架构实现，随着其他行业 App 陆陆续续引入在线客服系统或者支持用户和商家直接点对点沟通，一定要学习社交类 App 的在线聊天技术。
- 新闻类 App 比拼的是推送的及时性和到达率，所以大都是自己搭建推送服务器，而不依赖于第三方推送平台。

越来越多的 App 都意识到数据的重要性，开始采集用户行为数据，以助于更准确地做出战略上的决策，优化自己的产品和功能。由于这些都涉及公司机密，所以往往不使用第三方的服务，而都是自己采集数据，自己分析。老牌移动互联网公司在这方面会比较有优势，毕竟做得久了，积累了很多经验。

综上所述，从技术层面而言，同行业内竞争对手的 App 产品一定要经常研究，而对于整个 App 应用领域，各个行业都有其优势，我们要学习他们各自的优点，用到自己的 App 中，这才是竞品分析的意义所在。

因此，做竞品分析，紧盯着竞争对手固然没错，但是只盯着他们，就会把自己的逼格也降低了。一定要把眼界放大，立足于整个 App 行业，一步步的、不知不觉地就会超越竞争对手，自然就会让竞争对手跟着我们的节奏走了。所谓“胸有多大，舞台就有多大”就是这个道理。

于是，我把市面上所有优秀的 App 都定义为我的竞品。不气吞山河，又怎能兼济天下？

9.1.2 竞品分析要研究的几个方向

对于竞品，我们要研究其做得好的地方，从技术层面讲，有以下几点是重点研究方向：

- 为什么他们的 App 体积比我们小？
- 为什么他们的 App 访问速度比我们快？
- 为什么他们的 App 不发版也能上新功能？
- 为什么他们的 App 基本就不怎么崩溃？
- 为什么同样的产品，我们的价格更有优势，但是却卖不过竞争对手？

这些问题和答案在后面会陆续介绍。

9.1.3 竞品分析与拿来主义

第一次听到“竞品分析”这个词语，是从产品经理的口中。

从产品层面讲，“竞品分析”就是把竞争对手优秀的产品仔细研究一番，然后原封不动照搬到自家产品上。这样的抄袭多了，以至于几年前有分析师在比较了某个领域的几款 App 首页后，得到的结论是这些 App 看起来都是同一个设计师设计的，因为排版风格都是一样的。

对此我也只能呵呵一笑。我观察到的情况是，这种通过竞品分析后抄袭得到的产品，只学习到了人家的皮毛，而没有领会到产品内在的精髓，以至于产品上线了，但效果并不如竞

争对手。因为没有把“为什么要这么做、这样做的好处是什么”理解透，这就是盲目抄袭的后果。短期内效果还不明显，因为移动互联网现如今是烧钱的时代，大家都是赔本赚吆喝，都追求的是用户量，但是等钱烧完了开始追求利润的时候，就会发现这种反噬。所以研究竞品，如果纯粹是为了抄袭，就意义不大了。

从技术层面讲，竞品分析是为了取长补短。每个 App 在技术上都有做得好和不好的地方。我们看到了别人家 App 的长处，就要思考自家 App 如何取长补短。

这就是鲁迅先生倡导的“拿来主义”，在拿来的同时，又不能生搬硬套，并不是所有外来的技术都适合我们，要有选择地吸收。

9.2　App 安装包的结构

9.2.1　Android 安装包的结构

Android 的安装包是 apk 格式的文件。我们将其后缀名 apk 改为 zip，就可以看到安装包中的内容。

如图 9-1 所示，所有的 Android 安装包解压后都具有这样的目录结构：

图 9-1　Android 安装包解压后的目录结构

简单介绍一下这些目录和文件的用途：

- resources.arscz 这个文件是编译后的二进制资源文件的索引，也就是 apk 文件的资源表（索引）。
- lib 目录下的子目录 armeabi 存放的是一些 so 文件。
- META-INF 目录下存放的是签名信息，用来保证 apk 包的完整性和系统的安全。但这个目录下的文件却不会被签名，从而给了我们无限的想象空间。
- assets 目录下面可以看到很多基础数据，以及一些本地会使用到的 HTML、CSS 和 JavaScript 文件。
- res 目录下面的 anim 子目录很值得研究，这个目录存放 App 所有的动画效果。Android 做动画可以使用 xml 来配置，而不是写代码。iOS 的动画都是使用代码写出来的，这是件很费力气的事情。一种好的解决方案是，在 App 的 Android 版本中找到某个动画对应的 xml，将其翻译为 iOS 的动画语言即可。

注意，res 目录中的很多 xml 文件打开后是乱码，AndroidManifest.xml 也是如此，那是因为打包的时候对 xml 文件进行了压缩，所以看到的往往是全角的字符和乱码，不便于查找到我们想要看的内容。有一款神器用于看到 apk 包中正常的内容，AXMLPrinter2.jar，它可以将 apk 中已经处理过的 xml 还原为可读格式。命令如下所示：

```
java -jar AXMLPrinter2.jar AndroidManifest.xml
```

9.2.2 iOS 安装包的结构

iOS 的安装包是 ipa 格式的文件。我们将其后缀名 ipa 改为 zip，就可以看到安装包中的内容。

所有的 iOS 安装包解压后都具有如图 9-2 的目录结构：

其中 Payload 目录下是一个包，里面有这个 App 所需要的所有图片、音频、布局文件、配置文件和可执行文件、bundle 文件、HTML5 相关文件。

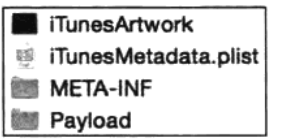


图 9-2 iOS 安装包解压后的目录结构

很多 png 图片是打不开的，那是因为在 iOS 打包时，对一部分 png 图片进行了压缩。

9.3 竞品技术一瞥：开机速度

无论是哪个 App，它的启动步骤都大体相同，如图 9-3 所示。

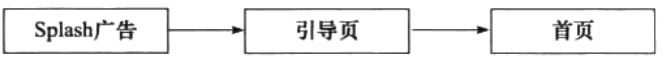


图 9-3 App 启动流程

我们仔细研究一下每一步都做了哪些事情：

1）Splash 广告的逻辑是，首次加载 App 包中的图片，同时调用 MobileAPI 的一个接口，获取下一次打开的图片 URL，把这张图片存放在本地。那么下次再打开这个 App 时，就加载这张新图片，同时，仍然调用 MobileAPI 的那个接口，看是否有新的 Splash 图片要下载。为了确保首页打开速度，MobileAPI 的这个接口一定是异步调用的。

2）引导页，不要超过 4 页，甚至 4 页我都认为多。最近流行在引导页加入动画，让 App 变得活泼生动一些。因为做原生动画比较耗费人力和时间，所以很多公司要么不加，要么用 gif 动画来实现。

3）进入首页之前，很多 App 会要求用户选择所在城市，有的 App 是默认选一个城市进入，有的 App 则是异步定位当前城市，同时给用户选择所在城市的机会。

4）App 首页的设计，则经历过几次大的革命。过去是把公司的主要产品放在首页很显眼的位置，次要产品则放在二级页面，也有的公司是每个品类做一个 App。现在通用的做法是，尽可能多地把所有产品都显示在首页，会有轮播广告，会有搜索框，会有滚动条。首页这个位置太重要了，只要出现在首页的产品，卖的都很好。

以上都是看得见的东西，接下来说一些在后台做的看不见的事情：

1）友盟打点统计，统计激活数。

2）注册推送。

3）如果是从消息推送点击进入的 App，则要根据推送协议，跳转到具体的页面。

4）初始化崩溃收集机制，如果上次崩溃时没有来得及发送崩溃信息，那么这次发送。

总结一下上述这些事情的共性，都是要调用 MobileAPI 接口获取数据的。为了不影响首页打开速度，这些操作都是在后台异步执行的。

不同公司的 App，它们所使用的第三方服务不同，所以还会做一些别的事情。对于其中比较耗时的，也都是要放在后台异步执行。

由此而引入一款嗅探器，WireShark，也有使用 fiddler 的。当我们试图探索别人家 App 为什么首页加载速度那么快的时候，使用嗅探器可以观察出首页加载期间该 App 调用了哪些 MobileAPI 接口，以及返回用了多长时间，下载了哪些 Zip 包以及 Zip 包中有哪些东西。

很多 App 升级新版本后会直接崩溃，这是程序员没有做好 App 兼容导致的，肯定是上一个版本遗留下来什么脏数据，在升级后新版本没有处理好如何兼容这些脏数据就崩溃了。所以 App 发版前必须要做兼容性测试，以确保稳定性。最好的解决方案是，App 升级后，除了用户信息要保留之外，所有遗留数据都要清除。

9.4　竞品技术二瞥：HTML5 页面的打开速度

9.4.1　把 HTML5 页面嵌入到 Zip 包中

App 中会使用很多 HTML5 页面。我们一般使用内置的 WebView 来打开一个外部的 URL 地址，这样一来，速度就肯定不如 App 原生的页面快了。

我们可以打开几个 App 的 HTML5 页面来进行比较，差距立刻就能看出来。当年我就是被老板追着问为什么竞争对手的 App 打开 HTML5 也就 1 ~ 2 秒，而我们的 App 加载 HTML5 页面就跟牛车一样慢。

我看过很多 App 的内部结构，发现无论是 ipa 还是 apk 包中都会有一个 Zip 压缩包，里面存放着要加载的 HTML5 页面、图片、CSS 和 JS 文件。App 每次启动的时候，会启动一个线程，异步把 Zip 包解压到本地的某个目录下，然后每次从本地读取 HTML5 页面，这样就不用每次从服务器加载 HTML5 页面了。

也许有人会问，如果这个 Zip 包里的内容有变化怎么办？比如说新增了图片或是修改了 HTML5 页面的内容。我们需要有个版本控制机制。每次加载 HTML5 页面之前，先问一下服务器，当前 HTML5 页面的版本是什么，如果与本地保存的版本号相同，就直接加载本地的 HTML5；否则，就从服务器重新下载一个新的 Zip 包，仍然解压到本地相同的目录下。

如果客户端自带 Zip 包版本比较旧，那么每个新下载的用户打开 App 都要下载服务器最新版本的 Zip 包。这样不好，会导致 Zip 包很大，要下载很久，所以每次发版前，都要把服务器上最新的 Zip 压缩包放到 App 安装包中。

9.4.2　Zip 包的增量更新机制

即使如此，每次有新版本的 HTML5，都要下载一个最新的 Zip 包，还是很慢。为此我

们要减小 Zip 的体积。我们知道，Zip 包中包括 HTML5 页面、图片、CSS 和 JS 文件，但并不是每次升级每个文件都要更新，我们要把那些不随版本升级而变化的文件挑出来，压缩成 common.zip，放到 App 包中，仍然是第一次启动 App 后解压缩到本地。这样每次 HTML5 页面的版本要升级，确保要下载的 Zip 包中只包括新增的和修改的文件就可以了，从而确保了 Zip 包的体积最小，可以快速下载到 App，仍然解压到相同的目录下，如果有相同的文件则将其覆盖。我们称这种机制为“增量更新”。

我说的这种增量包，只包括新增的和修改的文件，对于删除的文件，我们不用去管它，就把它扔在手机的本地目录下好了。

也许有人会问，当 App 正在访问本地一个 HTML 页面的时候，恰好本地解压 Zip 包时要覆盖这个文件，那么会不会像 PC 机那样弹出个窗口提示“该文件正在使用中，复制工作不能进行”？经过测试，在手机上不存在这个问题。

就算是增量更新，也要控制增量包的大小在 100KB 以内。

9.4.3 制作 Zip 增量包

那么问题来了，如何制作增量包？比如说，HTML5 内置在 App 中的版本号是 1.0，两天后线上 HTML5 版本更新为 1.1，于是我们需要提供一个增量压缩包供用户下载，这是 1.1 和 1.0 之间的增量。又过了两天，线上 HTML5 版本更新到 1.2 了，因为我们不确保所有用户的 HTML5 本地版本都从 1.0 升级到 1.1 了，所以这时我们就要提供两个增量压缩包，一个是 1.2 和 1.1 之间的增量包，另一个则是 1.2 和 1.0 之间的增量包。随着 HTML5 版本的不断升级，每次要生成的增量压缩包会越来越多。

我们不能每次手动打增量包，这是要累死人的。我们需要记录每个 HTML5 版本对应的目录和文件，放到 GIT 服务器上是个不错的选择。GIT 提供这样的命令行工具，比较两次提交之间的区别。此外，需要做一个小工具，它具有一个“发布”按钮，点击这个按钮后将会从 GIT 服务器中逐个比较当前版本 1.2 和历史版本 1.0、1.1 之间的区别，分别打包成 Zip，然后发布到服务器上提供下载。

这是件很繁琐的事情。但是你会发现，虽然后台生成增量压缩包的逻辑超级复杂，但是 App 的业务逻辑却非常简单了，App 只要知道增量压缩包的下载地址就够了。

即使如此，如果增量包中的图片过多，那么这个增量包还是会很大。这时我们就要控制增量包中图片的数量，只要保证 App 首屏显示的图片在增量压缩包中即可，至于屏幕外的图片，还是使用 http://www.aaa.com/aa.jpg 这样的 URL，同时要在 App 的 WebView 控件上建立图片缓存，以确保再次访问这个 URL 时不会重新加载图片浪费用户的时间和流量。

对于后台运营人员，她们要经常上一些新活动，以至于每时每刻都有可能更新 HTML5 的内容并希望用户能立刻看到这些变化。这时候手动点击 Publish 按钮就会跟不上节奏了。一种好的解决方案是，在服务器创建一个 Task，每 10 分钟执行一次，把这 10 分钟内有改动

的内容重新打增量压缩包。服务器端业务逻辑仍然会很复杂，但是极大地提高了客户端的信息更新频率。

有的 HTML5 页面会在 HTML 页面中使用 AJAX 调用网络接口获取数据，当我们将其打包成 Zip 解压到手机本地再去加载的时候，AJAX 因为存在跨域的问题而不能访问到网络接口数据，这时我们就要同时从 App 的代码和 HTML 的代码进行配置，才能解决这个问题。

9.4.4　使用 WebView 预先加载 HTML5 并缓存到本地

前面的设计太复杂了。

为了快速加载 HTML5，我们可以尝试多种方法。比如，利用 WebView 控件的缓存技术。如果在 App 的某个页面设置了 WebView 的缓存，那么再次打开相同的 URL 时，如果没有过期，就会使用本地的缓存数据。但即使如此，也不能解决第一次加载 HTML5 时很慢的问题，但是我们可以在上一个页面创建一个 WebView，让它预先加载这个 URL，这样就能提前把 HTML5 页面缓存到本地，一定要记住，要把这个 WebView 设置为不可见，否则就露馅了。

这样做虽然大幅提升了 HTML5 加载的速度，但是却非常耗流量，采用这个策略的时候要谨慎。

9.5　竞品技术三瞥：安装包的大小

9.5.1　从几件小事说起

春节在家帮姐姐的 iPhone 手机安装市面上形形色色的 App，忘记她是使用 4G 流量包月了，于是在下载了 10 个 App 后，不但耗尽了流量，还按照 0.3 元 / 兆的价格扣了七八十元的流量费。后来我检查了这几个 App 的体积，发现每个 App 体积都是 40 ~ 50MB 的样子，这让我很吃惊，因为我记得两年前这些 App 也就在 10 ~ 20MB 的样子。

另一件记忆犹新的事情，是去公园景点游玩，当时公园门口有个活动“扫二维码下载 App 下单立减 10 元”，但是我发现下载这个 40MB 的 App 要花费 12 元的流量，这样其实是要额外多花 2 元钱，所以“扫码立减”这件事情对于我这种“小市民”而言是很不划算的。

由此而得到一个结论，App 安装包的体积一定要小，至少要比竞争对手的 App 体积小。

对于 Android 而言，国内的各大市场商店已经发现这个问题了，所以对于用户升级 App，会为每个 App 提供增量下载的功能，所以 App 版本升级不再是几十兆的流量，而只是下载 1 ~ 2MB 的增量包就能升级到最新版本，这样就极大节省了流量。[⊖]

对于 iOS 而言，AppStore 从 iOS6 开始提供增量更新功能。对于 iOS 6.x 和 iOS7.0，只要有文件改动过，这个文件就会进入到增量更新包中，比如说 1 个 10MB 的文件，只改动了

⊖ 关于 Android 增量更新技术，请参见 http://blog.csdn.net/hmg25/article/details/8100896。

1KB 的内容，这个 10MB 文件就会进入到增量更新包中，包还是很大。到了 iOS7.1 及更高版本，这个机制进行了改良，它会把这 1KB 的改动内容放到增量更新包中，从而极大地减少了增量更新包的大小，但是安装的时候会变慢，因为要把这 1KB 的改动内容合并到 10MB 文件中，这是个很繁琐很费时的工作。[⊖]

尽管如此，以上种种措施只能解决升级用户的流量困扰，对新用户并无帮助。我们必须减小安装包的大小，才能吸引更多的新用户。

9.5.2 安装包为什么那么大

是什么让 App 安装包的体积变得如此之大？

我们在前面的章节看到了 iOS 和 Android 安装包的内部结构，对于可执行文件，我们无能为力；对于 xml 文件，这些文件在 App 打包压缩后会极大减小体积，所以也不用管它们；那么就只能在图片和音频文件上做文章了。

各位读者看到这里，都请停下手中的工作，检查一下自家 App 包中图片和音频文件的大小。图片但凡是大于 1MB 的，都是需要瘦身的。对于 500KB ~ 1MB 这个区间内的，也有瘦身的可能。我研究过很多知名的 App，其中有很多图片都在 2 ~ 3MB 的样子，其实真没有必要，之所以这么大，是因为 UI 设计人员提供的设计稿就是这么大，开发人员拿过来也不看文件体积大小直接就往项目里放，久而久之，App 包的体积就大了。

在众多 App 之中，我印象最深的是一款旅游类软件，它的所有图片都不超过 100KB，甚至说 50KB 以上的图片都屈指可数。这是把品质做到家的表现。

接下来说音频文件，对于应用类 App 而言，我见到的大都是 App 推送时发出的声音，这个声音很简单，不应该超过 10KB。但我在很多 App 中看到的音频，都在 100KB 左右。这是我们优化的一个方向。网上有很多这样的软件，可以对音频进行大幅压缩。

9.5.3 png 和 jpg 的区别及使用场景

设计师曾经问过我，App 为什么不使用 jpg 图片，因为同样的尺寸，png 格式的图片要比 jpg 图片大很多。

众所周知，png 有透明通道，而 jpg 没有，此外 png 是无损压缩的，而 jpg 是有损压缩的，所以 png 中存储的信息会很多，体积自然就大了。

但是手机却偏偏对 png 情有独钟，会对其进行硬件加速，所以我们会发现，同样一张背景图，png 虽然体积比 jpg 大但是加载速度却要快一些。

综上所述，对于 App 包中的图片，我们都使用 png 格式的，而对于要从网上加载的图片，考虑到流量以及下载速度，则使用 jpg 格式的，因为它有较高的压缩率，体积很小。

但是对于背景图、引导页，这种大尺寸的图片，我们还是倾向于使用 jpg 格式，虽然加

⊖ 关于 iOS 增量更新机制，请参见 https://developer.apple.com/library/ios/qa/qa1779/_index.html?utm_source=iOS+Dev+Weekly&utm_campaign=iOS_Dev_Weekly_Issue_114&utm_medium=email。

载慢一些，但是体积小，减少了包的体积。我看过的 App 基本都是这么做的。

对于 Splash 广告图，就是那个每次开启 App 一闪而过的广告，由于我们隔三岔五就要从线上下载新的广告图并展示在 Spalsh 页面上，所以这里使用 jpg 格式的图片。

对于 iOS，苹果规定启动页（Launch image）必须是 png 图片，否则审核时就会被拒。

Google 后来发布了一种新的图片格式，WebP，它的压缩率比 jpg 更好，已经慢慢普及。Android 自然是支持的，iOS 想要使用这种格式的图片，需要在程序中引入 WebP 解码器。

9.5.4　Splash、引导图和背景图

通过对 50 多款 App 中的图片逐个分析，我发现有 3 种比较典型的场景，大多数公司的解决方案是雷同的：

1）Splash 默认广告是体积最大的，而且对应不同机型，要做多套，根据我的经验，每张图控制在 300 ~ 500KB 左右就可以了。分辨率再高，对于手机而言，看不出效果。

2）引导图，设计师每次都会给几张高分辨率的图片，然后程序员不加思索地直接放到 App 里，这样 App 体积自然就变大了。其实，仔细观察，你会发现，为了保持风格统一，这些图片的背景都是一样的。所以我们完全可以这样做，比如说背景上有一只小兔子：

- 把背景与小兔子拆分成 2 张图片。如果另一个引导图的背景上有一只小鸭子，那么就只需要这张小鸭子的图片了，背景图可以复用。
- 根据分辨率，动态放置小兔子的位置，动态拉伸背景图，使之铺满整个屏幕。

3）对于背景图，为了达到一种视觉效果，这张图片经常被添加虚化等效果，既然如此，没有必要做得太清晰，应该控制在 50KB 左右，看到很多 App 中类似的背景图都在 1MB 左右，实在没有必要。背景图一般使用 jpg 文件。

9.5.5　iOS 的 1 倍图、2 倍图和 3 倍图

iOS 不使用像素作为单位，而是使用点这个单位，对于 iPhone4 及之后，1 点等于 2 个像素；而对于 iPhone3GS 及之前，1 点等于 1 个像素。这样就保证了之前在 iPhone3GS 上运行的 App，不用修改也能在 iPhone4 上运行。[⊖]

但是原先适用于 iPhone3GS 的图片，比如 a.png 的尺寸是 30 × 40 像素，在 iPhone4 中看起来就模糊了。于是我们必须为 a.png 再准备一张 60 × 80 像素的图片，命名为 a@2x.png，也放到 App 项目中，这样 App 在运行时会根据屏幕是否为 iPhone3GS 来选择相应的图片。iPhone3GS 会选择 a.png，iPhone4 会选择 a@2x.png。对于 iPhone4 而言，如果没有这张 2 倍图，则选择 a.png，所以就模糊了。

iPhone4S、iPhone5、iPhone5c、iPhone5s、iPhone6，它们都使用 a@2x.png 这张 2 倍图。

直到 iPhone6 Plus，才需要提供 a@3x.png 的图片。如果没有这张 3 倍图呢，它会选择 1

⊖　详细内容请参见知乎上的这篇文章：http://www.zhihu.com/question/25421514/answer/31623909。

倍图或 2 倍图，我尝试过只有 2 倍图的情况，在 iPhone6 Plus 上确实是模糊的效果。

那么问题就来了，我们需要为每张图都提供 1 倍图、2 倍图和 3 倍图这 3 张图片吗？

我看到一款国际版的 App 是这么处理图片的。它在提供了多国语言文字的同时，还为每张图片生成了 1 倍图、2 倍图和 3 倍图。这就导致了这款 App 的体积非常大。看上去有点“宁可错杀一千，不可放走一个”的感觉，但只要反过来想，图片一张也不缺，永远不会模糊。

我查看过很多 App 的图片，发现 1 倍图铺天盖地，但并不是每张 1 倍图都有相应的 2 倍图和 3 倍图，或者是只有相应的 2 倍图而没有 3 倍图，当然，也有只存在 2 倍图和 3 倍图而找不到 1 倍图的情况。图片管理五花八门，乱七八糟。

但是在中国，可不是这样哦。我看过友盟给出的数据报告，中国 iPhone3GS 用户不足 0.1%。于是，我有一个大胆的设想，就是把 iOS App 的包中所有的 1 倍图都干掉，为每张图生成 2 倍图和 3 倍图。

很多公司都有根据 1 倍图批量生成 2 倍图和 3 倍图的工具，我也曾用 C# 写过一个。但是我发现有问题，并不是每张图片转换后都清晰，矢量图可以拉伸，但是拉伸位图就会失真。当我反过来根据 3 倍图批量生成 1 倍图和 2 倍图时，却发现位图可以压缩，而矢量图压缩后会失真。于是一种好的解决方案是，先把所有图片按照位图和矢量图进行分类，属于矢量图的，要提供 1 倍图，然后批量转换为 2 倍图和 3 倍图；而属于位图的，则提供 3 倍图，然后批量转换为 1 倍图和 2 倍图。

这个解决方案并不能有效减小 iOS 包的体积，说不定反而会增大包的大小，但是却能系统地对图片进行管理，从而确保每张图片都是清晰的。

9.5.6 在 iOS 中进行图片拉伸和旋转

在 Android 技术领域，流行 .9 图这个概念，从而极大地节省了图片的体积。iOS 其实也可以这么干，使用 iOS 的图片拉伸语法，可以把一张 .9 图铺满一个区域，比如说按钮，如下所示：

```
(UIImage *)resizableImageWithCapInsets:(UIEdgeInsets)capInsets
```

其中 capInsets 这个参数是一个 UIEdgeInsets 类型的结构体，被 capInsets 覆盖到的区域将会保持不变，而未覆盖到的部分将会用于平铺。

以上这个方法只适用于 iOS5.0 及以上版本，5.0 以下版本有另外的解决方案，但是目前国内的 App 都只支持 5.0 以上版本了，所以这里我就不提及了。

对于箭头，更没必要准备上下左右 4 张图片，准备一张图片就够了，使用的时候在方向上进行旋转即可。

9.5.7 使用 XML 配置动画

动画主要用在引导图中以及加载进度条上。

做应用类 App 的开发人员做动画不是很在行，所以他们会要求设计师提供 gif 格式的动

画，或者二十多张图片进行轮播，以达到 gif 动画的效果，殊不知，在编程上简单了，但是 App 的体积却相应变大了。

比较简单的解决方案是，减少动画中的关键帧，来降低动画的大小。

稍微正规一点，还是要使用原生的 Android 或 iOS 原生代码来实现。任何复杂的动画，都是由四种简单的动画组成的，分别是：移动、旋转、缩放、渐变。在 Android 中，是使用 XML 来配置的，上述这四种简单动画都有对应的 XML 语法，可以很快拼凑出一个复杂的动画；而对于 iOS，只好使用编码方式了。

我们为什么不仿照 Android 的 XML 动画实现技术，为 iOS 也量身定制一套 XML 的动画标签呢？从而不用写任何 Objective-C 代码，配置几行 XML 就展现一个动画。

我见过一家 App 就是使用这个思想在 plist 中配置属性来做 iOS 动画的，如图 9-4 所示，就是一个平移的动画。

▼ animation1	Dictionary	(7 items)
startX	Number	2.8
endX	Number	1
startY	Number	1.2
endY	Number	1.2
imageURL	String	a.png
duration	Number	0.35
delay	Number	0

图 9-4　配置文件中的平移动画

基于这个配置，还需要有一个动画引擎，来解析这个配置文件，将其翻译成 Objective-C 原生语言。在设计模式中，我们称之为解释器模式。

不单如此，我还需要有个测试页面，通过在这个页面中修改动画的属性，然后点击按钮能立刻看到改动后的效果，而不需要重新运行 App 程序。点个按钮就能执行输入框中的 XML 脚本。

使用上述的若干方法，我们可以把 1 个 500KB 左右的 gif 文件，减小到 50KB 的几张图片，并且极大地节省了而开发成本。

9.5.8　iOS 使用 storyboard 还是 xib

抱歉，我始终不喜欢 storyboard，但是存在即合理。我曾经认为 storyboard 比 xib 大，是导致 iOS 安装包体积变大的一个原因，于是我做了一件探索性的工作，就是把 storyboard 中的页面拆分为若干个 xib 文件，然后重新打包，但是结果却是前后大小一致。

结论是，是否使用 storyboard，对 ipa 包大小没有影响。

9.5.9　字体文件的学问

我在某个 ipa 包中发现了 ttf 格式的字体文件。起初还以为是他们的 App 使用了某种特

定字体，但打开这个 ttf 文件后才发现，这里面存放的居然是图片，如图 9-5 所示。

图 9-5 字体文件中的 icon 图片

每个 icon 对应一个十六进制的数字，比如第一个是 \Ue600，这个值是唯一的。

观察这个字体文件，我们看到所有的 icon 具有以下共性：

- 这些 icon 都是单色的，可以在 App 中的页面里设置这些 icon 为其他颜色，但也必须是单色。
- 这些 icon 可大可小，因为它们是一种“字体”，字体是矢量图，所以拉伸不会失真。
- 这个 ttf 文件体积很小，比做成单独的 png 图片要小。

有人立刻就会联想到 iOS 的 1 倍图、2 倍图和 3 倍图，每次都要准备 3 张图片，分别适用于不同的手机型号。如果做成一个字体，就可以减少体积，再也不用设计 @2x 和 @3x 两套图了——这种方案仅限于单色图片。

我们一般到下述网站来把单色 icon 转换成字体文件：https://icomoon.io。或者使用 FontLab 这样的工具自己来制作。

接下来我们来看如何在 App 中推广这门技术。

1）Android

首先把这个字体文件放到 assets 目录下，如图 9-6 所示：

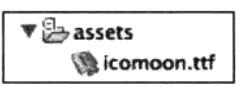


图 9-6 assets 目录下的字体文件

接下来，我们将 icon 和十六进制编码的映射关系保存在 drawable 资源文件中：

```
<string name="font_icon_1_normal">&#xe606;</string>
<string name="font_icon_1_pressed">&#xe607;</string>
```

这样就可以使用 R.id.font_icon_1_normal 这样的语法来取出这个图片了。

```
TextView textView1 = (TextView) findViewById(R.id.textView1);
Typeface font = Typeface.createFromAsset(
        getAssets(), "icomoon.ttf");
textView1.setTypeface(font);
textView1.setTextSize(12);
textView1.setText(
        getResources().getString(R.string.font_icon_1_normal));
```

也可以将其设计为一个 Drawable 对象，然后设置给 ImageView 这样的控件。

2）对于 iOS，实现思路差不多。

首先我们要把 icmoon.ttf 文件添加到项目中，如图 9-7 所示。

图 9-7　UseIconInTTF 中的 icomoon.ttf 文件

在 Supporting Files 目录下的 UserIconInTTF-Info.plist 文件中，增加一个配置，类型指定为 Fonts provided by app-lication，在其中添加对 icomoon.ttf 字体文件的声明，如图 9-8 所示。

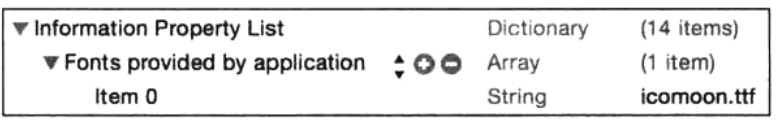


图 9-8　在 UseIconInTTF-Info.plist 配置 icomoon.ttf

与 Android 类似，为了不直接使用 \ue605 这样的十六机制编码数字，我们将 icon 和十六进制编码的映射关系定义为一个宏 TTFConstants：

```
#define font_icon_1_normal  "\ue605"
#define font_icon_1_pressed  "\ue606"
```

接下来只要两行代码就能显示这个字体文件中的图片：

```
[self.label1 setFont:[UIFont fontWithName:@"icomoon" size:12]];
[self.label1 setText:
          [NSString stringWithUTF8String: font_icon_1_normal]];
```

9.5.10　表情图片打包下载

对于表情图片。很多 App 中集成了聊天功能，有了聊天，自然就要提供各种表情图片，有静态图 png，也有动画 gif，虽然每个都不大，但是数量多啊，都打到包里面一起发布，会直接导致包变大。

考虑到实际的场景，用户不会一打开 App 就使用聊天功能，所以我们可以把这些表情图片打包成一个 Zip 包，在启动 App 的时候，在一个新的线程中异步下载这个 Zip 包然后解压到本地。这样以后聊天的时候就可以使用本地的图片了。对此，我们要做好版本增量升级功能，以确保有新表情图片的时候也能下载到本地后使用。

9.5.11 清除未使用图片

对于 Android 而言，Eclipse 可以自动检查出哪些图片没有用到。

对于 iOS 而言，则需要写个小程序，逐一检查哪些图片没有使用到，注意，对于 a@2x.png 和 a@3x.png 的处理，要先将 @2x 和 @3x 过滤掉。

无论是 Android 还是 iOS，即使发现到冗余图片，也不能直接删除，因为我们的程序经常会在代码中动态决定要显示哪些图片，我们只能检查这些图片在版本库的修改历史，来决定这些图片是否真的不需要了。

9.5.12 Proguard 不只是用来混淆的

一提起 Android 中的 Proguard，我们首先想到的是代码混淆，那是因为我们要经常去修改 proguard.cfg 文件，去 keep 那些不需要被混淆的类和方法。

其实，Proguard 还能瘦身 apk，在打包时它会帮忙检查那些不使用的类和方法，将其移除。最有效果的就是那些第三方 SDK 了，比如说 aSmack 这个用于 xmpp 的 SDK 有几万个方法，但其实我们只使用其中很小的一部分。Proguard 会帮助我们把不使用的部分移除，从而极大地减小了 apk 的体积。

9.5.13 在 iOS 中使用 pdf 格式的图片

在研究各家 App 的过程中，我发现某款 App 的 ipa 文件中有几张 pdf 格式的图片。

在研究中还发现，有几款 App 的 ipa 包中的每张图片都做成 imageset 的形式，每个 imageset 目录中都同时存在这张图片的 1 倍图、2 倍图和 3 倍图，如图 9-9 所示。

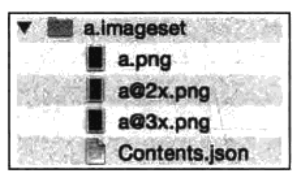


图 9-9 cancelSelectedListBtn.imageset 目录下的 3 张图片

上述这两件看似无关的事情，其实是使用了 iOS 的一个新技术。让我们从这些蛛丝马迹中探踪索隐。

我请设计师把这几张 pdf 图片做成同样的 png 图片，体积相差不大，所以和 png 相比毫无优势。由于这个 App 的 ipa 包中有几百张图片，其中只有这 3 张图片是 pdf 格式的，所以我怀疑，这只是他们的新技术尝试。

再观察 imageset 目录下的那 3 张图片，我发现每张图片都是这样的。这不由使我意识到，一定是用了什么工具，一次性生成的这些图片。

搜索关键字 ios+pdf，直到找到“Using Vector Images in Xcode 6”这篇文章[⊖]，才发现这是 iOS8 才出现的一种新技术，只能在 XCode6 上使用。

在这里简单介绍一下这门技术，先绘制一张 pdf 矢量图，然后 XCode6 在编译的时候，会生成 3 张 pdf 格式的图片，分别是 1 倍图、2 倍图、3 倍图。这样就避免了图片不全导致的模糊，也避免了每次都要设计师准备 3 套图的麻烦。

⊖ 文章参见 http://martiancraft.com/blog/2014/09/vector-images-xcode6/。

但是，我们为什么要在用户的 iPhone 上装一些永远用不到的图片呢？苹果煞费苦心搞出来这样一门技术，仍然没有解决 App 体积日益膨胀的现实，而且这门技术只会让 App 的体积变得更庞大——而这才是用户的痛点所在。是否可以让 App 中只包括 pdf 矢量图，只有在用户下载完 App 开始安装的时候，才根据用户的机型，把 pdf 转换为相应的图片，比如 iPhone 6+ 上 App 生成的图片就是 3 倍图。

苹果公司在 iOS9 中推出了 App 瘦身功能，据说能大幅减少要下载的 App 包的体积，具体效果如何，我们拭目以待。

9.5.14　iOS 的包永远比 Android 包体积大吗

我比较了 100 多款 App 后发现，同一款 App 的 iOS 和 Android 版本，iOS 的 ipa 包一定比 Android 的 apk 包在体积上大很多。

但总是有特立独行的 App，比如某款著名视频播放软件，Android 版本 23.2MB，而 iOS 版本才 20.5MB。我起初以为这个 App 的 Android 版本做得有问题，于是仔细研究了这个 Android 包里的内容，我就发现这家公司 Android 技术做得很精致，之所以比 iOS 版本体积大，是因为 Android 版本为几十种分辨率都适配了不同的图片和布局，以确保用户体验在任何分辨率下都是一致的。

如图 9-10 所示，居然有 12 种 layout 布局。

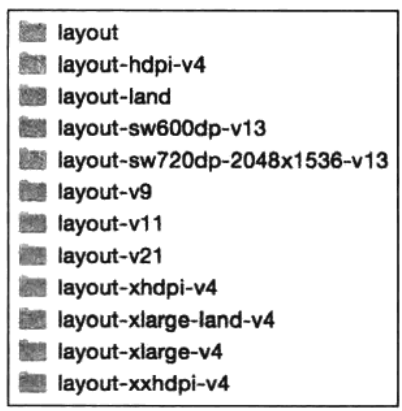


图 9-10　Android 项目中的 Layout 文件夹

drawable 文件夹就更多了，高达 28 个，限于篇幅，这里就不贴图了。

以上只是特例，而大多数 App 并没有做得那么细致，比如：

1）首先是不支持那么多分辨率。这是由当前 App 的开发现状导致的。一方面是产品经理和设计师人力不足，另一方面则因为设计师偏爱 iPhone，一般只会给出 iPhone 版本的设计稿，然后让 Android 开发人员根据 iPhone 的设计稿去适配。于是 Android 开发人员只好去做 UI 自适应，使用 .9 图拉伸技术，实在搞不定了，才去找设计师重新给画一张。所以我们会看到 iPhone 的 App 大都很精致，相应的 Android 版本都很粗糙，这是因为 Android App 的 UI 很多都是开发人员凭着自己的审美观去二次加工的。

2）其次，不同 drawable 目录下放置着不同内容的图片。用开发人员的话讲，好找。比如 drawable 目录下放各种 Selector 文件，drawable–hdpi 目录下放美食类图片，drawable-large 目录下放门票图片。所以，目录虽多，但其实只有一套图。殊不知，这样反而降低了 App 运行的速度，因为它在相应分辨率的 drawable 目录下找不到某张图片时，就会逐个遍历每个 drawable 目录下的图片，直至找到该图片的位置。

9.5.15 从代码层面减少 iOS 包的体积

对于 iOS 而言，在 ipa 包中会有一个 .a 格式的二进制文件，这是代码编译后生成的文件，往往占据了整个 ipa 包的 50% 到 80% 的体积。苹果曾要求所有的 App 都支持 64 位，于是在此基础上，ipa 包的体积又扩大了将近一倍，主要是那个 .a 文件编译后变大了。

我们要想办法减少这个 .a 文件的大小，其实就是要减少项目中的冗余代码。经过不断地摸索和尝试，我发现这些冗余代码分为以下 3 部分：

1）*已经不使用的类*。为此，我们需要写一个 Python 脚本，逐个检查哪些类不再使用了。检查的过程中我发现，某个类即使不使用了，但是在其他类中仍然保持对它的引用，所以我们要排除掉这种特殊情况，不让它对我们的检查工作造成影响。

还存在这么一种情况，在 A 类中使用了 B。A 类不再使用了，第一遍执行 Python 脚本找出来 A 类，将其删除了。这时 B 类就孤零零地放在那里，也不再使用了，所以我们有必要再次执行 Python 脚本，将 B 也找出来。以此类推，不停地执行这个 Python 脚本，直到再也找不到不再使用的类为止。

2）*已经不再使用的方法*。这个找起来有些费劲，因为 Objective-C 独特的方法签名形式（方法签名由三部分组成，包括方法名称、参数和返回类型）。

仍然需写一个 Python 脚本，逐个遍历每个类中的方法，然后到项目中查找是否使用到了。

在执行过程中，遇到这么一种情况，A 类和 B 类都有 loveBaobao 这个方法，方法签名也完全相同。这时 Python 是区分不出来到底是使用了哪个类的 loveBaobao 方法的。我们也只能将其汇总起来，然后用手动检查。

此外，有很多方法是系统自带的，比如说 UITableView 的那 6 个方法，只要使用了 UITableView 的页面，都有这 6 个方法。我们在执行 Python 脚本的时候，不应该统计这样的方法。所以需要做一个白名单，事先把这些方法填进去。

3）*代码相似度问题*。初级程序员在写代码时，喜欢把一段代码从 A 类粘贴到 B 类中，然后修改其中的几个变量名称，这个功能就算做完了。于是两段相似度极高的代码就产生了。

稍微懂得些面向对象思想的人，都知道这时候需要把这样的代码抽象出来，比如在 Utils 类中新建一个方法，然后要用到这段逻辑的人调用 Utils 类的这个方法即可。

但并不是所有的程序员都有这样的境界，即使是有几年开发经验的人，也会采用复制粘贴大法敷衍了事。久而久之，冗余代码就多了，包的体积自然就大了。为此，我们需要有一个检查代码相似度的工具。在 iOS 领域，我推荐 Simian 这个工具。有兴趣的读者可以尝试一下，对你们的项目使用一下这个工具，看能找出来多少相似的代码来。

9.6 竞品技术四瞥：性能优化

9.6.1 App 自动选取最佳服务器的策略

我们经常看到 App 中会包含一个服务器列表文件，开发人员和测试人员可以随意切换到

任意服务器进行开发测试工作。

这只是服务器列表文件的一种功用，是给开发和测试人员使用的，为此我们需要为 App 设计一个后门，由他们手动进行切换，相关内容请参见 9.9.2 章节。

服务器列表文件还有另一种作用，就是由 App 自己来决定选用哪个服务器作为 MobileAPI 服务器。

众所周知，App 发起 MobileAPI 请求到接收到数据，这个过程所耗费的时间由 3 部分组成：从 App 到达服务器的时间，服务器处理的时间，从服务器到 App 的时间。其中，从 App 到达服务器的时间，加上从服务器到 App 的时间，我们称为来回走路时间。对于 2G、3G、4G 和 WiFi 用户，因为网络环境的不同，来回走路时间大相径庭。

于是我们会准备多台服务器，可能是放在全国各地，也可能是分别接入电信、移动或联通的专线。这些服务器有可能是配置相同的，也有可能是由若干高配和低配组成。我们把这些服务器的域名罗列在 App 的服务器列表文件中，如下所示：

```
<Servers>
    <Server key="s1" type="3G" url="http://login1.company.com/">
    <Server key="s2" type="3G" url="http://login2.company.com/">
    <Server key="s3" type="4G" url="http://login3.company.com/">
    <Server key="s4" type="4G" url="http://login4.company.com/">
    <Server key="s5" type="2G" url="http://login5.company.com/">
    <Server key="s6" type="2G" url="http://login6.company.com/">
    <Server key="s7" type="WiFi" url="http://login7.company.com/">
    <Server key="s8" type="WiFi" url="http://login8.company.com/">
</Servers>
```

接下来，我们会让 MobileAPI 提供一个接口服务 A，该接口不需要任何入参，直接返回 1 这个结果。这样就确保了 App 从发起 MobileAPI 请求到接收到数据的时间，就是来回走路的时间。

在 App 第一次启动的时候，我会让 App 根据当前的网络情况，遍历服务器列表文件中的域名，访问这些域名下的接口服务 A，计算出哪个域名的访问速度最快。同一个域名只访问一次，得不到准确的数据，一般而言，我会调用 10 次后取平均值，来作为参考标准。

当网络环境发生变化的时候，也要把上述这个操作执行一遍，测算出该网络环境下哪个域名的访问速度最快。为了避免频繁做这个事情，我会设置一个缓存，记录最后一次测算每种网络环境的时间，以确保 1 个小时之内不会测算 2 次。

一旦测算出当前网络环境下哪个域名的访问速度最快，那么接下来 1 个小时内，访问 MobileAPI 就会使用这个域名了。1 个小时后，我们将在 App 后台线程再次发起测算工作，重新选择最佳的域名。

上述这种解决方案，能帮助用户选择最快的 MobileAPI 服务器，但是由此会导致另一种负面效果，App 一厢情愿地认为网络环境好所对应的服务器访问速度也最快，于是这台服务器的 CPU 会迅速被占满，无法处理后续接踵而至的网络请求。所以，我们要将服务器的处理能力划分为优良中差四种级别，并在 App 发起测评请求（调用 MobileAPI 接口服务 A）的

时候把这个值返回给 App，当达到中（CPU 占用 60%）这个级别时，即使网速很快，也不能采用这个域名对应的服务器。

9.6.2 使用 TCP+Protobuf

当大多数公司还在纠结于如何能更好提高 MobileAPI 的性能时，已经有公司开始抛弃 HTTP+JSON，开始走 TCP+ProtoBuf 的路线了。

TCP 是长连接，ProtoBuf 则是基于二进制的协议，可读性差但是体积小。这里我不讨论 Protobuf 协议中的 required、optional 或 repeated 关键字，也不讨论 Android 和 iOS 大小端对齐的问题。这些都属于 App 和服务器能使用 Protobuf 进行通信的第一步。

我只说三点，一是工具，二是架构，三是性能。

1. 工具

我们需要做一个工具，能帮助开发人员把 ProtoBuf 协议自动转换为 Android 或 iOS 的实体类和相应的方法。使用该方法就可以发起一次 ProtoBuf 请求并获取到服务器返回的实体数据，这将极大地加速开发人员的工作效率。

2. 架构

传统 MobileAPI 返回 HTTP+JSON，当我们改为使用 TCP+ProtoBuf 的时候，之前的 JSON 仍然要维护，因为我们要给自己留一条后路，一旦服务器上的 TCP+ProtoBuf 扛不住了，要立刻能切换回 HTTP+JSON。

那么问题就来了。难道我们要为 App 同时维护两套 MobileAPI 逻辑吗？当然不行，一种理想的设计方案如图 9-11 所示。

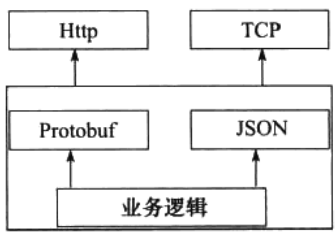


图 9-11 新的 MobileAPI 架构设计

但是反观我们的 MobileAPI 代码，却不是这样的，你会发现业务逻辑和 JSON 绑定很紧，往往是从后台取到数据就立刻填充到 JSON 字段中了。我们需要重构，把取数据的业务逻辑和返回什么样的数据（JSON 或 ProtoBuf）剥离开，最好能拆分成 3 个项目，最差也应该是在一个项目中拆分为不同的目录。这样业务逻辑如果有变动，只需要修改一个地方，然后在 JSON 或 ProtoBuf 中追加字段。

生成器模式（Builder）这时候就能派上用场了，它能很好地弥合 ProtoBuf 和 JSON 这两种数据格式的差异性。

我们把业务逻辑、JSON 生成器、ProtoBuf 生成器框在一起后，下一步要面临的就是以 HTTP 还是 TCP 的协议返回给 App 数据了。HTTP 协议由 Header 和 Body 两部分组成，都需要填充数据，其实我们也可以在 TCP 协议中定义 Header 和 Body，把之前填充在 HTTP 的 Header 中的版本信息、Cookie 传递过去。

策略模式（Strategy）可以用于指定使用 Http 协议还是 TCP 协议。

3. 性能

TCP 要解决的技术难点就在于，服务器上长连接数量多会导致服务器性能压力，如果解决不了，用起来还不如 HTTP。

于是我们采取 TCP 长连接和短连接混合的模式。

TCP 长连接就是每个 App 客户端都是作为一个连接，保存在服务器的长连接池中。但是这个池子中的长连接数量是有上限的，所以我们持续清理池子中长期不使用的长连接，比如说几分钟内不使用就关闭这个连接，大不了以后再连上来。

资源是有限的，对于日活几十万的 App 而言，我们要保证服务器至少能支撑这几十万个长连接。如果超过了这个池子的上限，那么我们就要使用短连接作为补充。短连接就是连接后完成一次调用就把连接关闭了。

服务器要根据当前长连接池的情况，来决定建立长连接还是短连接。如果 TCP 长连接和短连接都没有资源了，那就切换到 HTTP，这其实也是一种短连接。

网络请求的场景不同，也会影响 TCP 长连接和短连接的选择。比如说 xmpp 聊天，就比较适合 TCP 长连接。用户的活跃度，也可以作为选择 TCP 长连接还是短连接的依据。活跃用户往往会长时间使用 App，频繁发起网络请求，这时候要使用长连接。对于那些偶尔打开 App 随便点一点看一看的用户，可以先使用短连接。等用户发起网络请求的次数超过某个阈值时，就切换到长连接。

网络环境是影响 App 选择 TCP 长连接还是短连接的又一个因素。对于 WiFi 环境，网络请求普遍比 2G、3G 和 4G 要好。接下来的策略有两种：

- 快的更快、慢的更慢，为使用 WiFi 的客户端建立长连接，而为 2G、3G、4G 网络环境下的客户端分配短连接。
- 均衡策略，反正 WiFi 已经很快了，分配给它短连接不会有太大影响，而为了提高 2G、3G、4G 网络环境下的客户端访问速度，尽量为它们建立长连接。关于在 2G、3G、4G 网络环境使用 TCP 长连接是一个很热的话题，经常会出现网络不给力导致 TCP 连接频繁断开的情况，所以我们要做好随时可以把这部分用户切换到 HTTP 短连接的机制，以备突发情况的发生。

不得不说的是，WiFi 不一定快过 4G，甚至是 3G 和 2G，所以上述策略有不准的情况。

9.7　竞品技术五瞥：数据采集工具

9.7.1　页面跳转器

页面跳转器是页面打点的前提。

对于 Android 而言，有 Intent 来帮助我们进行页面跳转和传值。但是你会发现，想从 A 页面跳转到 B 页面，在 A 页面要声明 B 页面的实例，这是一个强引用，如下所示：

```
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
startActivity(intent);
```

对于 iOS 而言，就连 Intent 这样的机制都没有了。我们不但要在 A 页面声明 B 页面实例，还要通过为 B 设置属性的方式，进行页面间传值。如下所示：

```
- (void) jumpTo {
    APageViewController* b = [[APageViewController alloc] init];
    b.version = "5.7.1";
    [self.navigationController pushViewController: b animated: YES];
    [b release];
}
```

我们一直在强调解耦，但是在 iOS 和 Android 的页面传值上却不遵守这个原则。于是很多公司开始致力于解决这个问题。写一个 Navigator 类，通过使用反射技术可以接触页面间的耦合性，这样我们就可以把所有的页面都定义在一个 XML 配置文件中，每个节点包括该页面的 key、对应的类名称、打开方式。

我们先解决 iOS 的页面传参。使用一个字典作为页面间参数传递的载体，为此，在 ViewController 的基类中定义一个字典参数，这样在 Navigator 反射的时候，将传递进来的参数设置给页面实例即可，下面，分别是 Navigator 的 h 和 m 文件：

```
#import <Foundation/Foundation.h>

@interface Navigator : NSObject {
}

+ (Navigator *)sharedInstance;

+ (void)navigateTo:(NSString *)viewController;

+ (void)navigateTo:(NSString *)viewController
    withData:(NSDictionary *)param;

@end
#import "Navigator.h"
#import "BaseViewController.h"
#import "SynthesizeSingleton.h"

@implementation Navigator

SYNTHESIZE_SINGLETON_FOR_CLASS(Navigator);

+ (void)navigateTo:(NSString *)viewController {
    [self navigateTo:viewController withData:nil];
}

+ (void)navigateTo:(NSString *)viewController
      withData:(NSDictionary *)param {
    BaseViewController * classObject = (BaseViewController *)
```

```
    [[NSClassFromString(viewController) alloc] init];
    classObject.param = param;

    [classObject.navigationController
      pushViewController:classObject animated:YES];

    [classObject release];
}

@end
```

为了解决页面间传参的问题，我们需要在 BaseViewController 中增加一个 params 属性，这是一个字典，在跳转前把要传递的属性塞进去，在跳转后把字典中的值再取出来：

```
@interface BaseViewController : UIViewController {
    NSDictionary* _param;
}

@property (nonatomic, retain) NSDictionary* param;
```

那么在使用时就非常简单了，如下所示：

```
- (void) jumpTo {
    NSMutableDictionary* dict = [NSMutableDictionary dictionary];
    [dict setObject: @"5.7.1" forKey:@"version"];

    [Navigator navigateTo: @"BViewController" withData: dict];
}
```

而在目标页 BViewController 要接收这个参数：

```
if(self.param!=nil){
    version = [self.param objectForKey: @"version"];
}
```

接下来要解决的是 Android 的页面耦合。不必新建一个 Navigator 类，我们完全可以利用 Activity 基类，增加一个 navigatorTo 方法，利用反射把要跳转的页面实例化出来，如下所示：

```
public abstract class AppBaseActivity extends BaseActivity {
    public void navigatorTo(final String activityName, final Intent intent) {
        Class<?> clazz = null;
        try {
            clazz = Class.forName(activityName);
            if (clazz != null) {
                intent.setClass(this, clazz);
                this.startActivity(intent);
            }
        } catch (ClassNotFoundException ignore) {
            return;
        }
    }
}
```

相应的，我们要创建 ActivityNameConstants 这个类，用来存放每个 Activity 的用于反射的全名称，如下所示：

```
public class ActivityNameConstants {
    public final static String SecondActivity
        = "com.example.navigator.SecondActivity";
}
```

在 Activity 使用 navigatorTo 方法的时候就非常简单了，如下所示：

```
Intent intent = new Intent();
intent.putExtra("name", "Jianqiang");
navigateTo(ActivityNameConstants.SecondActivity, intent);
```

相应的，还应该有一个 startActivityForResult 方法，实现原理差不多，我这里就不赘述了。

9.7.2 打点统计

1. 打点统计的两大痛点

如何寻找一种好的打点统计方法，是整个 App 业界都在做的一件事情。我这里只是抛砖引玉，把我这三年来的实战经验和切身感受分享给大家。

确保 App 打点数据的准确和无遗漏，是实现“数据驱动产品”的第一步，非常重要。纵观各大公司的打点办法，都非常原始，往往是哪个页面或哪个事件需要打点，就在相应的方法体中写一行打点的语句。

这种原始的打点方式直接导致以下问题：

- 不全，经常漏打。
- 不准，经常打错。

一旦发生了上述问题，要等下次发版后，数据才会恢复正常。基于此，我们需要解决 2 个痛点：

1）如何在发版前就能检查出漏打的和打错的点。

2）如果在发版后发现漏打的和打错的点，快速修复快速上线，而不必等新版本发布。

打点分为两种，页面打点，事件打点。接下来我们逐个讨论。

2. 页面打点

相比较而言，页面打点比较容易实现。我们可以统一在页面跳转时，进行页面打点统计。还记得前面章节介绍的跳转器吗？我们只要在这个地方加上页面打点语句即可。

iOS 的实现是在 Navigator 的 navigateTo 方法中，我们在 9.7.1 节介绍过这个类，如下所示：

```
+ (void)navigateTo:(NSString *)viewController
      withData:(NSDictionary *)param {

    //在这里执行页面打点的操作
```

```
    BaseViewController * classObject = (BaseViewController *)
                [[NSClassFromString(viewController) alloc] init];
    classObject.param = param;

    [classObject.navigationController
                pushViewController:classObject animated:YES];

    [classObject release];
}
```

Android 的实现则是在 BaseActivity 基类的 navigateTo 方法中，我们在 9.7.1 节中介绍过这个方法，如下所示：

```
public void navigateTo(final String activityName,
        final Intent intent) {

    //在这个位置执行 PV 打点的操作

    Class<?> clazz = null;
    try {
        clazz = Class.forName(activityName);
        if (clazz != null) {
            intent.setClass(this, clazz);
            this.startActivity(intent);
        }
    } catch (final ClassNotFoundException e) {
        return;
    }
}
```

只要把页面打点语句写在上面代码片段的注释位置就好了。在这个位置，我们可以搜集到页面名称（viewController 或 activityName 参数），也可以解析 param 字典或 Intent 参数，从中找出一些重要的参数记录下来，比如说 movieId。

采取上述机制，能有效防止页面打点遗漏的问题。

此外，为了防止打点错误，应该动态传递当天 ViewController 或 Activity 的名称，而不是手动去拼写这个字符串，这就增加了出错的可能性。

相比较而言，页面打点的解决方案比较简单，我们甚至可以使用这种机制，计算出页面停留时间。接下来要介绍的事件打点的优化方案，可就不那么简单了。

3. 事件打点

事件打点是比较棘手的。一般而言，我们为事件打点都是在事件方法中，增加一行事件打点的代码。这样的代码多了，就很难维护，经常发生打错点或者有遗漏的情况，有时则是这个迭代有某个事件的打点数据，但是下个迭代却不小心删除了。

我们系望 App 开发人员在写代码的时候，不需要考虑打点的事情，不需要额外准备打点所需要的信息，比如说哪个页面哪个控件以及相关的数据。为此，我们写一个基类，把打点逻辑封装在这个基类中。任何继承自这个基类的控件，就能自动打点，而不用把打点逻辑写

在业务代码中。

这里我们先看按钮，因为绝大多数打点，都是基于按钮的点击。

对于 iOS，为一个按钮添加点击事件是通过 addTarget 方法，如下所示：

```
UIButton* getInfoButton;
[getInfoButton addTarget: self
                  action: @selector(getInfo)
        forControlEvents:UIControlEventTouchUpInside];
```

那么我们要写一个继承自 UIButton 的新控件，比如就叫 UVButton。我发现，所有的 UI 控件都继承自 UIControl 这个基类，它有一个 sendAction 方法，这个方法会在点击事件发生后第一个执行，之后才执行 addTarget 上绑定的方法。于是就可以在 UVButton 中重写这个 sendAction 方法，如下所示：

```
@interface UVButton : UIButton

@end
#import "UVButton.h"

@implementation UVButton

- (void)sendAction:(SEL)action to:(id)target
        forEvent:(UIEvent *)event
{
    //在这里写一个方法，执行打点操作

    [super sendAction:action to:target forEvent:event];
}

@end
```

打点操作的方法我这里就不提供了，反正就是搜集一些信息，存在某个地方，等待发送出去。

那么在程序中，所有的按钮我们都将使用 UVButton，你会发现，之前的逻辑是什么，完全不需要修改。只要把按钮声明为 UVButton 即可。

对于 Android，其实也可以这么做，创建一个新的按钮，重写它的 click 方法。但是我们发现 Android 为控件绑定响应方法的语法是通过 setOnClickListener，如下所示：

```
btnLogin.setOnClickListener(
    new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            gotoLoginActivity();
        }
    });
```

我们已经习惯在程序中使用 OnClickListener，复写它的 onClick 方法，来实现按钮点击后的业务逻辑。我们为什么不能从 OnClickListener 中派生出一个子类呢？比如叫

OnUVClickListener，复写它的 onClick 方法，实现事件打点的逻辑。

那么接下来在程序中就使用 OnUVClickListener 来代替 OnClickListener 了，除此之外，代码逻辑和之前一样。

上面的讨论虽然只是按钮，但也可以适用于 Image 控件。而对于列表控件、Tab 之类的复合控件，则需要特殊情况特殊处理。

4. 事件打点的验证

如果可能，我们希望采集每个页面和每个事件的点。但并不总是这样，所以我们需要有一个配置文件，每次页面跳转或点击控件的时候，都检查这个动作是否需要采集打点。

按照上述这种解决方案，我们需要写一个 Python 小程序，每次发版前验证一下这个配置文件，确保打点数据是全的，而且没有错误。

做得再极致一些，这个配置文件可以设计成从服务器动态下载。这样发现错了或者漏了，就可以在服务器提供一份新的配置文件供用户下载。

对于大多数 App 而言，是没有这个配置文件的。代码已经写成这样的，再改一遍不划算，那么要使用 Python 做静态代码检查就不能依赖于配置文件这个统一的出口了。那么我们有必要统计代码中所需要打点的地方，所在的类和方法，具体的代码行位置，然后每次执行 Python 就检查这些地方。

静态代码检查只能确保打点的代码都存在，但并不能确保在运行期间相应的打点代码被执行到了。

为此，需要引入 App 自动化测试。

首先在 App 端编写一组能够完整覆盖打点的自动化测试用例，在即将发版前，执行一遍这组测试用例。

然后，在服务器端，也需要编写一个自动化脚本，每当 App 端打点的自动化测试用例执行完，我们就执行服务器端的这个自动化脚本，检查是否所有点都打上了，以及打点是否正确。

5. 如何在发版后即时修复线上打点的错误

目前我所想到的解决方案有：

1）iOS 使用 Lua，临时把漏打或者打错的点修好。

2）Android 使用插件化编程，更新有问题的插件。

3）还记得我们上面说到的那个记录打点的配置文件吗？把这个配置文件做成服务器下载的，如果漏打或者打错，那么就更新这个配置文件。

6. 处理 App 中的 HTML5 页面打点

App 中有很多 HTML5 页面，它们也需要打点统计数据。

一种做法则是回调 App 的打点机制，让 App 把打点数据传到服务器。这时候经常发生的情况是，App 有 bug，某个版本上线后突然就不能回传数据了。所以 HTML5 和 Native 之间的协议是非常重要的，每次发版前都要逐一测试。

另一种做法是 HTML5 页面自己编写打点语句，然后上传到服务器，这样即使出错了，也能够立刻修复立刻上线。

9.7.3 ABTest

很多产品经理做一个功能是根据主观臆断出来的，拿不出切实的数据来证明方案的可行性，只能根据上线后的订单转化率，来猜测该方案是否有效。

这样做就像是赌博，这个问题也是最近一两年才暴露出来，于是很多 App 开始在所有页面打点，采集用户行为，把这些数据放到 Hadoop 中做大数据分析，最后基于数据来决定哪种方案是可行的。于是我们采用 ABTest 这种强大工具，用于判断：

- 做一个新功能，做之前和做之后哪个更有效果。
- 做一个新功能，方案 A 和方案 B 哪个更有效果。

1. 什么是 ABTest

我们可以将 ABTest 的定义归纳为以下几点：

- 场景：对于某一个页面，UI 样式的修改。
- 结果：得到旧版和新版（或者 A 方案和 B 方案）的订单转化率，比较后决定使用哪种 UI。
- 策略：产品经理和运营人员在新版本上线后比较一周，最终确定使用哪一种。这个决定必须在一个迭代内迅速作出，否则 App 接下来的版本就要维护两套页面的代码逻辑。
- 规则：ABTest 不一定是 A 和 B 各占 50%，也有可能是 A 占 20% 而 B 占 80%，也有可能是 ABC 三种策略各占一定的比例。

ABTest 的设计难点在于如何确保数据准确。

对于同一个设备，在第一次获取到 A 策略后，今后每次重启 App 访问那个页面都将一直是 A 策略了，除非我们关闭了该页面的 ABTest 并决定从此以后使用 B 策略的页面，该设备才有机会看到另一种页面。这样就避免了 ABTest 期间，同一个用户每次看这个页面都随即有不同的 UI 样式，这样我们就不能判断这位用户下单是受 A 策略还是 B 策略的影响。

2. 为 App 量身打造 ABTest

根据上述策略，我们对 App 和 MobileAPI 改造如下：

- App 对于要做 ABTest 的页面，如果是新页面，那么要做两套 UI，A 方案和 B 方案；如果是改造原有页面，那么不是在原有页面上进行修改，而是 copy 一份这个页面的副本，然后在副本上进行修改。总之，无论是哪种情况，都要确保有两套 UI。
- App 每次启动时就调用 MobileAPI 的一个接口 A，获取哪些页面要进行 ABTest，以及要采用哪种策略，把这些数据保存到本地文件中，注意，这里是覆写，也就是说之前保存的数据都不要了。

那么每次跳转到一个页面，都要判断一下，该页面是否要做 ABTest 以及相应的策略，

就从本地文件中读取到这些数据。还记得我在 9.7.1 节介绍的页面跳转器吗？我们可以把判断逻辑写在这个统一的页面跳转器中。

- 每次启动 App 时才会调用 MobileAPI 的接口 A 获取 ABTest 策略，但是大多数用户是不会关闭 App 的，只是简单地将其切换到后台，为了确保 ABTest 的策略及时更新，在 App 每次从后台切换到前台时，都要调用一次 MobileAPI 的接口 A。

3. 如何确保 ABTest 公平

接下来介绍 MobileAPI 中 ABTest 的策略分配算法。MobileAPI 应该有一个自增的整数 count，每次请求都会加 1。如果是 AB 各占 50% 的话，那么策略就是 count 除以 2，根据余数来分配 A 和 B 两种策略。如果 ABC 各三分之一，则对 count 除以 3 取余数来分配 ABC 三种策略。这里的 count 取余数的算法直接决定了 ABTest 策略的公平性。

策略分配后，每次有新的设备号来请求 ABTest 策略，就要把设备和分配到的策略保存下来，此外还要保存要做 ABTest 的 App 版本号、页面名称、Android 还是 iPhone。把这些数据保存在数据库中是不划算的，频繁的 IO 操作会导致性能问题，我们可以将每笔数据写成日志保存在服务器，然后每隔几个小时就发送到 Hadoop 上，进行大数据分析。

4. 如何衡量 ABTest 的结果

对 ABTest 的结果进行衡量，自然还是要使用大数据分析。

比如，对某个页面做 ABTest，AB 两种策略各占 50%。我们观察了一周后得到的数据是这样的：

1）订单的总转化率是 40%，分子 40，分母 100。100 是点击搜索的次数，而 40 是下单的次数。

2）分子 40 由 10 个 A 和 30 个 B 组成，分母 100 由 30 个 A 和 70 个 B 组成，那么 A 的转化率就是 10/30=33%，B 的转化率就是 30/70=42%，于是我们愉快的决定采用策略 B。

也许会有人问，为什么 A 的分母是 30，而 B 的分母是 70，取样儿怎么差距这么大。这是因为我们在 App 启动时就为用户分配了 ABTest 策略，但是用户不一定会进入搜索页和要做 ABTest 的那个页面，这样无形中就白白分配了很多策略。

比较精准的办法是，在需要做 ABTest 的页面，才会分配策略，但是这样做就要求每进入一个页面都要请求 MobileAPI 获取策略，这无疑会对 App 性能产生影响。

另一种折中的解决方案是，把这些只进入过搜索页但是没进入到 ABTest 页面的数据，从分母中剔除。这就需要 9.7.2 节中采集的 PV 打点数据来协助了。

5. 为产品经理和运营人员提供 ABTest 的配置后台和报表

我们要为运营人员或产品经理设计一个配置 ABTest 策略的工具，可以灵活配置在哪个页面、哪个版本做 ABTest，包括有几种 UI 样式（枚举），每个样式的百分比是多少，等等。

对于每个品类，一次只做一个页面的 ABTest。比如说火车票，如果有两个页面同时做 ABTest，将难以判断转化率的提升，是受哪一个页面修改后的影响。

我们可以每次测一个页面，得到结论后，再去测另一个页面。如果每次发版的间隔是 2 周的话，那就每个策略测试一周。

此外，还需要有报表，能在采集到数据后，看到 ABTest 的结果，以便于运营人员或产品经理迅速做决策。

对于 Android 和 iOS，应该可以分开看报表，也可以合在一起看数据。

6. 如何快速采用 ABTest 得到的策略

一旦通过 ABTest 收集的数据分析出最终使用 B 策略了，那么如何能快速的通知 App 该页面将不再进行 ABTest 并永远进入 B 页面呢？在下个版本删除 A 页面然后永远进入 B 页面，这件事情是肯定要做的。但这样就太晚了，我们要等待很久才能看到新版本的上线，所以我们要在当前线上的版本就立刻把页面切到 B。因此，我们要在刚才提到的那个 MobileAPI 接口 A 中，永远返回策略 B。这样就能解决及时更新策略的问题了。

7. 实施 ABTest 中遇到的一些问题和解决方案

我在设计 ABTest 的实现方案时，被质疑最多的是，每做一次 ABTest，都要设计两套 UI，App 开发人员的工时倍增。其实呢，这是一个磨刀不误砍柴工的概念。如果我们猜着在本轮迭代中开发 A 方案，两周后发现效果不好，然后在下个迭代再开发 B 方案——开发的人力没有省，但是开发的周期拉长了，除非你中途离职，不然活儿永远也躲不掉。

另一种做 ABTest 的方法是使用 Lua 脚本。MobileAPI 返回不同的 Lua 脚本，动态绘制不同的 UI 样式。这样就不用在 App 中准备两套 UI 了。

本文介绍了多套 UI 的 ABTest 方案。但其实还有一种仅限于数据层面的 ABTest 方案，比如说，点击搜索按钮，50% 的用户看到 A 方案的数据列表，而 50% 的用户看到 B 方案的数据列表，然后分别统计这两种方案下的订单转化率，最终采用转化率高的那种方案。由于不需要 App 的介入，所以稍微容易一些。

9.8 竞品技术六瞥：热修补

9.8.1 Native 页面和 HTML5 页面的相互切换

Native 页面和 HTML5 页面的相互切换是最激动人心的技术，比我一直在研究的 App 插件化技术还要震撼。因为插件化技术只能适用于 Android，对 iOS 无能为力。即使如此，搞 Android 插件化技术需要投入大量的人力物力，如果团队不够大是不建议搞插件化编程的。记得两年前我去一家公司面试，他们当时就在搞 App 插件化，面试时问我这方面的东西，被我当场泼了一头冷水，然后就没有然后了。

我们知道，Android 插件化更多是为了解决线上严重的崩溃或者 bug，有时也可以紧急上线一个新功能，而不用等到新版本发布。但问题恰恰出在这里，真正需要紧急修复的是 iOS，因为每次审核都要 1 ~ 2 周的时间，而 Android 可以随时发版到国内各大市场。我们不能做

亏本的买卖，费了巨大人力结果发现并没有解决主要矛盾。

于是我们会选择 HTML5，如果发现 App 出事了，就把那个模块临时切换到 HTML5 网站。但注意，我们通常是把整个模块切换为 HTML5 站点，这个模块再也不会有 Native 页面了。这种做法有些得不偿失。于是我开始思考，能否只修改有问题的那个页面，将其临时换成 HTML5，而这个模块的其他页面仍然使用 Native 的？

我仔细研究了一个页面——无论是 Android 还是 iOS，所必备的几个要素，列举如下：

首先是入口和出口，把入口和出口控制住了，尤其是传进来的参数和传出去的参数，我们就能做到随时在 Native 和 HTML5 之间切换。我们不能再随意的在 A 页面中实例化 B 页面了，我们应该使用 9.7.1 节介绍的页面跳转器，来解耦各个页面之间的依赖，才能把任何 Native 页面切换为 HTML5。

注意，直接使用 9.7.1 节的 Navigator 是有问题的。我们在 BaseActivity 和 BaseViewController 中定义的字典，用来在页面间传递参数。但是 HTML5 可不认这一套机制。所以有必要定义一套新的协议，同时适用于 Android、iOS 和 HTML5，pagename?k1=v1&k2=v2 是一种比较合适的协议。比如说，从 HTML5 跳转到 Android 或 iOS 页面，协议如下所示，其中单引号中的内容是协议，由 3 部分组成，Android 页面名称，iOS 页面名称，参数键值对，分别用逗号和分号分隔开。

```
<a onclick="baobao.gotoAnyWhere(
    'com.example.youngheart.MovieDetailActivity,
    iOS.MovieDetailViewController:movieId=(int)123')">
    gotoAnyWhere</a>
```

其次是状态，这其中包括全局变量、本地存储。一个 Native 页面通常要读写全局变量和本地存储，如果切换成 HTML5 页面，就不能干这些事情了，因此，我们要提供 Native 和 HTML5 之间的交互方法，以便于 HTML5 页面能读写 Native 中的全局变量和本地存储。

最后是公共组件，比如说网络请求和打点统计。这些要在 Native 中封装成公用方法，以便于 HTML5 回调这些方法。

如果把以上三点都做到了，就可以随时更换线上的某个页面了，我们只要在 App 启动的时候调用一个 MobileAPI 接口，获取一份页面清单，指定哪些页面是 Native 的哪些页面是 HTML5 的即可。

9.8.2　在 iOS 中使用脚本编程

1. 寻找快速修复 App 线上 bug 的办法

我们前面提到了在 App 中使用 HTML5，这其实就是脚本编程的一种，只不过要在 WebView 中展现。

我见过有些 App 通过返回 XML 格式或者 JSON 格式的数据，通知 App 绘制 UI。这其实也是一门脚本语言，但这么做只能把 UI 绘制出来，并不能动态返回一个 Native 的方法，

比如，点击按钮该做些什么事情。

我接下来要介绍的脚本编程，是指在 iOS 使用 Lua 或 JavaScript 这样的脚本语言。对于应用类 App 而言，也确实需要脚本语言介入了，尤其是那些对转化率要求很高的电商 App，线上一旦有致命的 bug 或者 Crash，可以迅速用脚本语言改好。这就好比身体受伤了，帖一个创可贴，等伤口愈合了（下次发新版本），再把创可贴摘掉。

在手机游戏领域，已经广泛采用 Lua 进行编程了。这样的好处是，每天都能通过 Lua 修改代码，增加个新的地图或者道具，然后通过 MobileAPI 把 Lua 脚本返回给 App，达到新功能迅速上线的效果，而不用受发版上线的制约。接下来我们看 iOS 中是如何植入 Lua 或 JavaScript 脚本的。

2. 在 iOS 中使用脚本语言的八卦史

首先隆重介绍 Wax 这个第三方开源库。Wax 是使用 Lua 脚本语言来编写 iOS 原生应用的一个框架，它建立了 iOS 原生 Objective-C 语言和 Lua 脚本语言之间的映射关系。

但是发明 Wax 的这哥们从 2013 年开始就不维护这个框架了，导致了 Wax 中的很多遗留问题没有得到解决，比如说不支持自定义的结构体和结构体指针，不支持多线程等等。

后来，2013 年年底，屠毅敏在 Wax 的基础上开发出 WaxPatch，这也是 GitHub 上的一个开源项目，它的神奇之处就在于，在 App 启动时会加载服务器上的 zip 包，zip 包中是用 Lua 脚本编写的补丁，在 App 运行期间，这些补丁文件中的方法能替换 iOS 中的任何一个类的任何一个方法的实现。它的实现原理是重写了运行时的 class_replaceMethod 方法。[㊀]

就在我们庆幸 iOS 找到了快速修复线上 bug 的解决方案，再不用因为线上有 bug 而要忍受老板能杀死你的眼神时，苹果在 2015 年 2 月强制要求所有新提交的应用必须兼容 64 位，但原来使用 Lua 的框架 Wax 是不支持 64 位的。

人生不如意事，十有八九。

于是又等了几个月，开源社区给出了 Wax 的 64 位版本，在此基础上，我们把 WaxPatch 的改动也移植过去，就有了 WaxPatch 的 64 位版本。[㊁]

2015 年 5 月，JSPatch 面世。它的原理和 WaxPatch 一样，都是在 App 运行期间替换 iOS 中的任何一个类的任何一个方法的实现，只是它是基于 JavaScript 来实现的。估计是 JSPatch 的作者等不及 Wax 和 WaxPatch 迟迟不更新所以才另起炉灶了吧。与此同时，JSPatch 的作者还提供了大量的实例来帮助我们理解这个开源项目。[㊂]

Wax 和 WaxPatch 毕竟很久不维护了，它不支持 iOS 的多线程语法以及自定义结构体和结构体指针，而 JSPatch 是支持这些 iOS 特性的，所以建议大家使用 JSPatch。本书即将出版的时候，JSPatch 已经比较成熟了，而且还在持续更新，优化因反射而带来的性能问题。让我们拭目以待。

㊀ WaxPatch 的源码地址：https://github.com/mmin18/WaxPatch

㊁ WaxPatch 的 64 位版本，参见 https://github.com/felipejfc/n-wax

㊂ JSPatch 的下载地址，参见 https://github.com/bang590/JSPatch

本书不打算过多介绍如何把 Objective-C 代码转换为 Lua 或者 JavaScript，官方文档已经讲得很清楚了。下面我将以 WaxPatch 为例，介绍一下它的使用策略。JSPatch 的使用思路也是一样的。

3. Zip 包下载策略

接下来介绍 WaxPatch 中压缩包的下载规则。压缩包中的内容就是用于热修补的 Lua 脚本。

首先返回 Lua 下载地址的 MobileAPI 接口，要区分 App 的版本。比如当前版本有一个严重的 bug，为了修复它引入了 lua001.zip，而我们在下一个版本修复了这个 bug，就不需要 lua001.zip 包，或者说等下个版本上线后又发现了新的 bug，这时候要引入 lua002.zip。所以这个 MobileAPI 接口应该根据版本号返回不同的 Lua 压缩包下载地址。

如何控制 App 不重复下载相同的 Lua 压缩包呢？每次调用 MobileAPI 接口获取到 Lua 压缩包的地址，比如说 lua001.zip，我们在解压 lua001.zip 这个压缩包到本地 lua001 这个目录下的同时会把 lua001 这个值存到本地文件的变量 luaVer 中。下次再调用 MobileAPI 接口，就会根据返回的 Lua 压缩包的地址进行判断：

如果值为空，说明不需要 Lua 脚本来修复 bug，那么就把 luaVer 设置为空。

如果值仍然是 lua001.zip 没有变化，就什么都不做。

如果值是一个新的 Lua 压缩包的地址，比如 lua002.zip，那么就下载这个压缩包，将其解压到 lua002 这个新的目录，并把 luaVer 这个值设置为 lua002。

按照上述策略，我们就可以根据 luaVer 的值，来控制 App 能加载到最新的 lua 压缩包，而且避免重复下载。

4. 调试策略

我们的策略是依赖 MobileAPI 返回的 Lua 压缩包的下载地址，但是不可能每次开发调试时，都把一个用于测试 Lua 压缩包发布到服务器上，因为我们在调试期间会频繁地修改 Lua 压缩包中的文件。

基于此，在调试期间，我们绕开从服务器下载 Lua 压缩包并比较版本的做法，改为把 Lua 压缩包中的文件直接复制到本地目录的方式，比如，lua001.zip 包中有 2 个 Lua 文件，我们把这两个文件集成到 App 项目中，在 App 每次启动的时候，就把这两个 Lua 文件复制到本地，然后就可以直接使用了。

在全部调试完成，就把代码切回到仍然从服务器下载 Lua 压缩包的模式。

5. Lua 不支持的场景及解决方案

并不是所有的 iOS 代码都能转换为 Lua 脚本。以下是我遇到的情况以及相应的解决方案。

1）如果变量或属性声明错了呢？

我们知道 WaxPacth 编程的思想是在 iOS 运行时注入，动态修改任何一个类的任何一个方法的实现。也就是说任何一个方法体都可以替换为 Lua 脚本，但就是不能修改方法的签名。但这还好，遇到这种情况，我们在 Lua 中重写一个方法，简单地包装一下 Objective-C

中不符合我们要去的方法即可。

但是如果是一个属性或类级别的变量的类型声明错了，我们就真的没办法了。仔细检查 WaxPacth 这个框架，还真没有定义一个属性或变量的地方。遇到这种情况，我们的解决方案是，在项目中增加一个 LuaClass 类，里面只有一个字典属性 dicLuaObject。

在 Lua 脚本中，我们把错误类型的属性或者变量所出现的任何地方替换为正确类型的变量，而这个变量则定义在 LuaClass 类的 dicLuaObject 字典属性中。

2）对于 block 块该如何处理呢？

Lua-Wax 不支持 block 块。因此一旦 block 块内的代码有问题，就要重写这个 block 块所在的方法，同时将 block 块中的代码封装成另成一个方法，也在 Lua 脚本中重写。

6. 如果 zip 包被劫持了呢？

不要以为 MobileAPI 返回了 Lua 压缩包下载的地址，就可以直接下载并使用了。经常有恶意攻击者劫持了服务器返回给我们的下载地址，而让我们去下载一个恶意的压缩包。我们一旦下载并解压缩这个恶意的包，接下来可能发生各种意想不到的事情。

为此，我们不能认为网上下载的任何压缩包都是安全的。我们需要一套校验机制，来保证这个下载到的压缩包是我们自己提供的，如果验证不过，就删除或者隔离这个文件。

SSH 是最简单的解决方案，但就是 HTTPS 协议访问起来太慢了，能否做成 HTTP 的呢？可以，我们需要准备一对公钥和私钥：把 zip 包使用私钥进行签名后再放到服务器提供下载；而 App 下载这个 zip 包到本地，则使用保存在 App 中的公钥进行校验。我们要对私钥进行严格的保密，不能泄漏给他人，这样即使有人在 App 中取到了公钥，因为没有配套的私钥，也没办法生成一个符合我们要取的 zip 包。

7. Lua 对 iOS 的深远影响

有了 Lua 这个利器，线上的任何 bug 或者 Crash 都能以最快的速度修复，而不需要重新提交审核新的版本并等待超长的时间。比如，我们最苦恼的是页面打点经常发现打错了或者漏打了，为了能不影响数据的采集，使用 Lua 能及时缝补这个漏洞。

最后需要补充的是，虽然 Lua 语言很简单，尤其是 WaxPacth 这个框架的支持，使得我们可以改写任何方法都很容易。但是我经常看到的是很多 Objective-C 方法都有成百上千行代码，这就给改写带来了很大的工作量。这就又回到了编码规范的层面，尽量把方法写的短小。每个方法只做一件事情。

提示 在 Android 中使用 Lua

iOS 因为有了 WaxPatch 而重新焕发了活力，而 Android 在 Lua 方向的进展却不温不火。

Android 因为可以使用插件化编程，而且即使线上有了严重的 bug，到各大市场发一次新版本就解决了，所以，相比 iOS，Android 有更多的选择。

其实 Android 也可以使用 Lua 脚本语言编程，业界比较公认的技术是 AndroLua 这个开源项目。我对 AndroLua 的研究还在进行中。也请越来越多的人关注这个项目。

本书临近出版的时候，听说淘宝有个团队推出一个名为 Dexposed 的开源项目，它是基于 AOP 思想来设计的，能解决性能监控、在线热修复等问题。这个开源项目还很年轻，但是我非常看好它。

9.9　竞品技术七脊：曲径通幽

9.9.1　一切皆可配置

1. 使用 XML 配置首页，防止因加载不到数据而没有入口

在很多电商类 App 中，我们会看到有一个配置文件或者 JSON 文件里面存放着首页展示所需要的所有数据，包括图片、文字等等，点击后能进入各个品类这些二级页面，如图 9-12 所示，我们可以看到，这个首页由 3 个 Tab 组成：首页、发现、个人中心，配置文件中指定了每个 Tab 的显示文字、点击后对应的 ViewController、所需的默认图和高亮图。

MainPageData.plist 〉No Selection

Key	Type	Value
▼Root	Array	(3 items)
▼Item 0	Dictionary	(5 items)
selected	Boolean	YES
className	String	MainPageController
highlightedImage	String	home-highlight
defaultImage	String	home-default
tabTitle	String	首页
▼Item 1	Dictionary	(5 items)
selected	Boolean	NO
className	String	PublishViewController
highlightedImage	String	publish-highlight
defaultImage	String	publish-default
tabTitle	String	发布
▼Item 2	Dictionary	(5 items)
selected	Boolean	NO
className	String	UserCenterViewController
highlightedImage	String	user-highlight
defaultImage	String	user-default
tabTitle	String	个人中心

图 9-12　某款 App 首页的 plist 配置文件

这么做是因为，如果获取首页信息的 MobileAPI 接口挂了，或者，就在我们调用该接口

的时候挂了，那么首页仍然能通过读取这个本地的配置文件或者 JSON 文件而正常显示，仍然能看到各个品类的入口，点击后进入，这样不影响生意。

但是这个配置文件或者 JSON 文件可能不是线上最新的数据，所以一种好的解决方案是，第一次启动 App 的时候把这个文件复制到本地，然后每次调用首页 MobileAPI 接口渠道数据后就把数据同步到这个文件，这样就确保了下次如果调用 MobileAPI 接口不通，仍然能显示比较新的数据。

2. 配置页面的公共行为

把首页的数据配置在 XML 中只是第一步，这个世界上不乏野心者，他们想把更多公用的东西做成可配置化。

比如，调用 MobileAPI 时是否要显示进度条，进度条中是否有取消按钮，点击取消按钮后是后退到上一页还是停留在当前页面，调用 MobileAPI 错误是否要显示错误提示，如下所示：

```
<ShowSetting showLoading="1"
    showCancel="0" goBackAfterCancel="1" showErrorInfo="1" />
```

又比如，进这个页面是否要登录，如下所示：

```
<WindowType needLogin="1"/>
```

所有这些信息都定义在配置文件中。我们应该在 App 中编写一套页面引擎，自动读取配置信息，这样就能少写很多很多代码。开发人员就可以把更多精力放在业务逻辑的实现上。

9.9.2 App 后门

任何成熟 App 都会为自己留一个后门，目前业界有两种做法：

- 只有 Debug 版本能看到这个后门，而 Release 版本看不到。
- 在线上 Release 版本中很深的一个页面，比如设置页面，点击某个特定的区域很多次后弹出一个对话框，要求输入密码，输入正确就能进入这个后门。

留一个后门有很多好处列举如下：

- 做一个能切换服务器的页面。这样就可以在开发期间，从线上环境切换到测试环境而不需要重新打个包，极大方便了测试团队对新功能进行验收。
- 要测试某个页面请求了哪些 MobileAPI 接口，打印出调用这些接口时输入的参数和返回 JSON 数据。这样就能够在线上 App 发现某个页面有问题时，及时在 App 后门中检查数据是否正常，而不用 App 开发人员和 MobileAPI 开发人员坐在一起逐行联调代码，极大节省了人力。
- 对于 App 崩溃，我们将最后一次崩溃的信息记录在本地，然后可以通过后门看到这个崩溃信息。这对于测试期间不经意点出来的崩溃，可以迅速追踪到问题的所在。当然，另一种方案是把崩溃信息发送到服务器，然后我们去服务器抓取崩溃信息，但是

这样不及时；而对于那些发现 App 崩溃然后来找我们的同事朋友来说，通过后门看崩溃日志是最好的途径。

- 提供一个后门页面供 HTML5 团队进行调试，该页面内置一个 WebView，加载 HTML5 团队正在开发的 HTML 页面，要支持调试。
- 对我们的 App 进行流量测试，统计某个页面所花费的流量，包括调用 MobileAPI、下载图片、上传文件、XMPP 聊天等等。其中，从 App 启动到首页加载完成所花费的流量是我们关心的一个关键点，而手机待机时，App 所花费的流量也是我们所关心的。我们需要这样一个后门页面，看到这些数据统计。
- 对我们的 App 进行电池电量消耗测试。需要有个后门页面记录每次打开 App 和退出 App 的时间，以及这段时间内我们的 App 所消耗的电量。为了确保数据的准确性，需要确保手机上只安装了一个 App，而且处于相同的网络环境下，比如 3G。

前面说到开一个后门，提供切换服务器的功能。这样测试人员可以在这个后门页面灵活配置当前 MobileAPI 要连接哪个服务器。基于此，这个后台页面需要显示服务器清单列表，而这个列表从 App 包中的一个文件读取，此外，还要支持手动输入服务器地址，因为有时候要直接连接到 MobileAPI 开发人员的机器，把他们的开发机器作为临时服务器。

9.9.3　Android 包中 META-INF 目录的妙用

对于 Android 批量打渠道包，每个团队都有切身的痛。每个包经过混淆和签名，都至少要 3 分钟时间，300 多个包就是十几个小时才能全打出来，所以一般在晚上干这个事情。

一般而言，我们在 App 每次启动时从 AndroidManifest.xml 这个文件读取渠道名称，如下所示，其中 360Android 是渠道名称：

```
<application>
    <meta-data
        android:name= "UMENG_CHANNEL"
        android:value= "360android"/>
```

然后在 App 中，每次从 AndroidManifest.xml 中取出这个渠道名称，传递给友盟或者我们自己的 MobileAPI 接口，如下所示，演示了如何取得渠道名称的方法：

```
private String getChannel(Context context) {
    try {
        PackageManager pm = context.getPackageManager();
        ApplicationInfo appInfo = pm.getApplicationInfo(
            context.getPackageName(), PackageManager.GET_META_DATA);
        return appInfo.metaData.getString("channel");
    } catch (PackageManager.NameNotFoundException ignored) {
    }
    return "";
}
```

上述是传统的做法，我们接下来介绍一种更快的做法。

我也是偶然的机会，看到一些知名的 App 包里面的 META-INF 目录，会有一个 0 字节的文件，文件名是某个渠道的值，于是我就大胆猜测，这个文件是用来批量打渠道包的。

我上网查了一下这个 META-INF 目录的功用，发现修改这个目录里面的文件，是不需要重新签名 App 的。于是我们可以如下进行优化。

1. 打包流程上的优化

打一个签名混淆过的正式包，我们称之为“母体”，然后往这个 apk 包中插入一个名为 channel_360Android 的空文件。这样一个渠道包就完成了，如图 9-13 所示。

图 9-13 META-INF 目录下的空文件

之所以在空文件的名称前面加上 channel_ 的前缀，是为了在运行期查找这个文件的时候，可以快速找到。

准备一个渠道列表文件 channel.txt，文件内容由 3 个渠道组成，每个渠道占一行，如下所示：

```
360Android
91Android
baidu
```

接下来我们使用 Python 脚本 build.py，遍历这个渠道列表文件，逐个生成渠道包，脚本如下所示：

```
import zipfile
import shutil
import os

base_dir = '/Users/Shared/'
apk_name = 'ChannelDemo'
apk_path = base_dir + apk_name + '.apk'

empty_file = base_dir + 'baojianqiang'
f = open(empty_file, 'w')
f.close()

channel_file = base_dir + 'channel.txt'
f = open(channel_file)
lines = f.readlines()
f.close()

output_dir = base_dir + 'output'
```

```
if not os.path.exists(output_dir)
    os.mkdir(output_dir)

for line in lines
    target_channel = line.strip()
    target_apk = output_dir + '/ChannelDemo_'
            + target_channel + '.apk'

    shutil.copy(apk_path, target_apk)

    zipped = zipfile.ZipFile(target_apk, 'a', zipfile.ZIP_DEFLATED)
    empty_channel_file = 'META-INF/channel_' + target_channel
    zipped.write(empty_file, empty_channel_file)
    zipped.close()
```

这样，生成第一个“母体”包需要几分钟时间，但是之后生成其他渠道包的时间就快了，就都是在 apk 中插入一个空文件的时间了。

2. 运行期间的优化

我们改为从 META-INF 目录读取那个 0 字节的文件名称，从中得到这个 apk 包的渠道号，网上有很多这样的代码例子，我就不过多说了。

按照上述打包新流程，一分钟打几百个渠道包不成问题。

9.9.4　classes.dex 的拆与合

一般而言，Android App 中的 dex 文件只有一个。但是对于很多业务逻辑复杂的 App，当方法数量超过 dex 的最大限制 65535 时，编译就会出错。业界称之为“爆棚”。

一种解决方案就是插件化。把那些独立的模块做成一个 apk，然后在使用的时候，使用 DexClassLoader 进行加载。对那些暂时还没有插件化编程的 App 而言，这种解决方案太遥远了。

另一种解决方案就是 dex 分包。就是将 classes.dex 拆分为 2 个 dex，让每个 dex 包的方法数量都小于 65535。很多 App 都是这么做的，有的 App 甚至拆分成 6 个 dex 之多。

Google 提供了 dex 拆包工具 multidex。关于这个工具的使用方法，请参见 CSDN 上“时之沙”的博客文章[⊖]。

但 multidex 仍然解决不了开发调试期间方法数超过 dex 上限的问题，开发人员不能每次调试都使用 Gradle 去打包，于是我们只好把不重要的 SDK 引用临时去掉，比如 GA 打点，同时注释掉引用了这个 SDK 的代码，这样就能正常编译和调试了，最后在提交测试或发布市场打包的时候再把删除的引用和注释掉的代码恢复过来。

每次都这么搞可不行，严重扰乱了开发节奏，于是我们采取在代码中动态加载 dex 的方式。对于那些第三方 SDK，比如 Umeng、Google Analytics、aSmack、JPush，都是导致 dex

⊖　博文地址：http://blog.csdn.net/t12x3456/article/details/40837287。

方法数徒增最终达到上限的“杀手”级 SDK，所以我们优先把这些 jar 包提出来，放到一个 apk 中，作为第二个 dex。这样我们就能使用 DexClassLoader 加载这些 SDK 啦。

只要能把方法数降低到 65535 以下，就又可以在各种 IDE 中正常开发调试了。

关于动态加载 dex 的技术，请参见以下文章，有更加详尽的介绍：

- custom class loading in dalvik[⊖]
- 美团 Android DEX 自动拆包及动态加载简介[⊜]
- Android dex 分包方案[⊝]

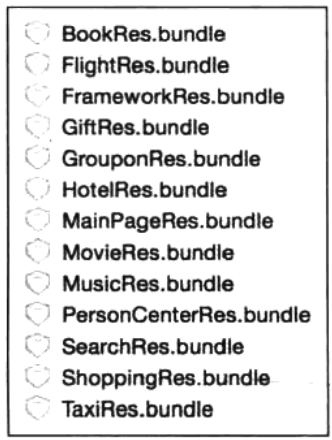


图 9-14 某款 App 包中，对资源进行了模块化拆分

9.10 竞品技术八瞥：模块化拆分

9.10.1 iOS 资源拆分与模块化

对于 iOS，很多 App 已经注意到图片会散落在各个地方，于是会把图片、配置文件、xib 按照模块进行归类，放到各自的 bundle 包中。做得最好的是一家电商 App，会在 App 包中的一级目录下面看不到任何图片，而只有若干 bundle，如图 9-14 所示。

只对资源进行模块化拆分是远远不够的。一定要对代码进行模块化拆分。把不同模块的代码放到各自的 GIT 仓库中，这样各个部门只对各自 GIT 仓库中的代码负责，而不会产生代码级别的依赖，如图 9-15 所示。

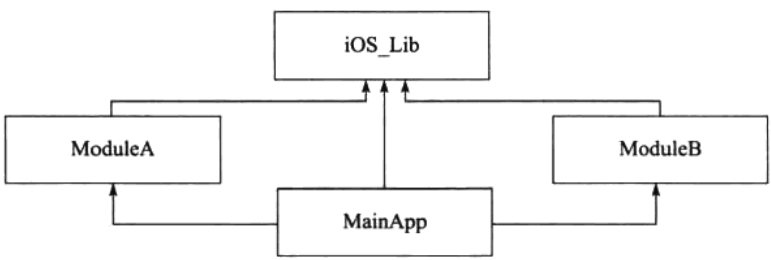


图 9-15 iOS 模块化架构

在 iOS 中，我们可以使用 .a 文件进行模块化拆分。把每个模块都以 .a 文件的形式嵌入到 MainApp 这个主模块中。

但是 .a 文件不能动态下载，所以也就不能使用类似于 Android 的插件化思想。要想动态更新模块还要另辟蹊径。

⊖ 博文地址：http://android-developers.blogspot.hk/2011/07/custom-class-loading-in-dalvik.html。

⊜ 博文地址：http://tech.meituan.com/mt-android-auto-split-dex.html。

⊝ 博文地址：http://my.oschina.net/853294317/blog/308583。

9.10.2　Android 模块化拆分

家大业大，子女多了以后就要考虑分家的事情，大家各过各的，出了问题尽量自己搞定。公司大了，也会面临同样的问题，我们会把 App 按照模块进行拆分，代码按照模块拆分到不同的 GIT 仓库中，不同部门负责各自不同的模块，他们会对自己的模块负责。

如果还按照之前的做法，把模块按照 Package 进行划分，看起来也不错，但是这样做会有问题。比如发版时间为 1 月 14 号，但是 A 部门负责的 A 模块却延期了，难道我们要延期发版吗？那不行。所以我们要把 A 模块从主项目中迁移出去，A 模块会作为一个 jar 包，主项目会保持对该 jar 包的引用。这样 A 模块如果延期了，那么主项目就仍然保持对 A 项目原有 jar 包的引用，这样就不耽误 1 月 14 号的正常发版了。

另一方面，各部门如果继续在一个版本库下工作，经常会搞出互相干扰的情况。比如说 A 提交的代码会导致 B 编译不过，A 提交的代码会冲掉 B 的代码，A 修改了公共方法会导致其他地方都报错。当我们把代码按照模块都拆开了就不会有问题了，A 随便提交自己的代码，只会影响自己的 GIT 仓库，不会祸及他人。

然而问题接踵而至，如图 9-16 所示。

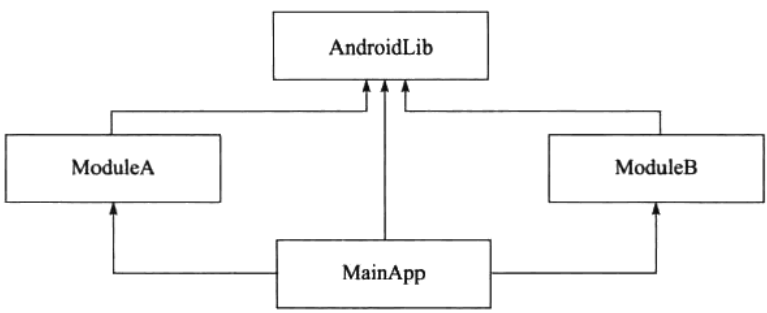


图 9-16　Android 模块化拆分示意图

我们看一个这个模块拆分图，有以下几个问题需要解决：

1）ModuleA 模块和 ModuleB 模块共用的类和方法，要怎么处理？

2）ModuleA 模块和 ModuleB 模块共用的资源，要怎么处理？

3）如何能在不同模块间共享数据？比如全局变量、模块之间页面跳转时传值。

对于问题 1），我们要解决代码上的依赖。所以我才会在本书第一部分介绍了如何剥离出一个业务无关的 AndroidLib 类库。在此基础上，我们可以轻松地把一个模块所涉及的那些 Activity 转移到另一个模块去。

对于问题 2），我们要解决资源上的依赖。首先是要制定模块的命名规范，所有资源前面都要加上模块名称，这样才能确保资源名称不冲突。对于公用资源，还是要放在 AndroidLib 目录下，AndroidLib 类库会为每个公共资源生成一个 R.id.xxx 的对应属性，我们要把这个 R 文件连同资源、AndroidLib 目录下的代码一起打成 jar 包，放到要用到它的 MainApp、ModuleA、ModuleB 模块中，这样手动打包时才不会出错。

对于问题 3），我们有很多手段，来传递数据。比如，从 ModuleA 模块的 A1 页面，跳转到 ModuleB 模块的 B1 页面，传递一些简单类型还好办，如果要传递自定义的实体，就只能把这个实体定义在 AndroidLib 类库中了。但是 AndroidLib 类库毕竟是放业务无关的代码，所以不适合存放这样的业务实体类，所以还是尽量不要改动 AndroidLib 类库。

比较靠谱的做法是，再新建一个存放实体的 AndroidEntity 类库，这些实体专门用于传递模块间要传递的数据。所有模块都保持对这个类库的引用。

我还见过模块间通信使用 JSON 文本的，这样就不用在 AndroidKit 类库中建实体类了。

对于用户身份信息，也就是 User 单例类，也是这么处理，把 User 类放到 AndroidLib 类库中。

如果一定要使用全局变量，而且要在不同的模块间读写，也可以这么处理。

接下来说一下开发人员如何创建自己的工作区：

1）最简单无脑的办法就是把所有的项目都打开，项目之间是代码级依赖关系。ModuleA 模块的开发人员只能修改 ModuleA 的代码，尽管能看到 MainApp 模块和 ModuleB 模块的代码，但是却没有权限修改。项目之间是代码级的依赖关系，那么自动打包脚本就要相应修改，我们要同时编译若干个项目，而且有先后顺序。请参见博客园“谦虚的天下”的文章“App 自动化之使用 Ant 编译项目多渠道打包”[⊖]。

2）比较高级的做法是 jar 包依赖的方式，只打开自己部门所属模块的代码。比如，对于 MainApp 模块的开发人员，他们只打开 MainApp 模块的代码，而把 ModuleA 模块和 ModuleB 模块对应的两个 jar 包引入项目中。这两个 jar 包是由 ModuleA 和 ModuleB 这两个模块的开发人员生成后上传到 MainApp 模块的 lib 目录的。而对于 ModuleA 模块的开发人员，则是要打开 MainApp 模块和 ModuleA 模块的代码，而把 ModuleB 模块对应的 jar 包，引入项目中。因为 MainApp 模块是宿主（也就是一个壳），所以不得不打开它的源码，才能编译调试代码。按照这种 jar 依赖的方式，自动打包脚本就非常简单了，仍然是单项目的打包机制，我们基于 MainApp 项目进行打包，其他所有模块都事先做成 jar 包放到 MainApp 项目的 lib 目录下。

9.11 竞品技术九瞥：第三方 SDK

App 是一个全新的领域，充满了未知，但这也正是它的魅力所在。开源社区上有各种千奇百怪的发明创造，以 GitHub 名气最大，其中一些开源项目已经为很多 App 所广泛使用，比如说，本章 9.5 节已经介绍过如何在字体文件中使用 icon。接下来我们就要看看还有哪些优秀的开源 SDK。

9.11.1 HTML5 篇

关于跨平台交互的开源项目有很多，以下几个比较有名：

⊖ 文章地址：http://www.cnblogs.com/qianxudetianxia/archive/2012/07/04/2573687.html。

- PhoneGap　这是跨平台开源项目的老大哥。我研究过一段时间，个人感觉这个框架太重了，所以才有下面这些开源项目的面世。
- WebViewJavascriptBridge.js　这是一个优秀的开源小项目，国内很多大公司的 App 都在使用它。它优雅的实现了 HTML5 和 App 之间的互相调用。就像项目的名称一样，它是连接 JavaScript 和 WebView 的桥梁。[㊀]
- zepto.js　这个开源项目兼容于 jQuery，和 jQuery 这个老前辈相比算是青出于蓝而胜于蓝。[㊁]
- CryptoJS　为 JavaScript 提供了各种各样的加密算法。
- mraid.js　MARID 是 Mobile Rich Media Ad Interface Definitions 的缩写，即移动富媒体广告接口定义，基于 JavaScript 实现。[㊂]

9.11.2　iOS 篇

- CocoaPods　iOS 最有名的类库管理工具，解决类库之间依赖关系的开源项目。
- EGOImageLoading　异步加载图片的第三方类库，有点类似于 Android 的 ImageLoader。关于 EGO-ImageLoading 的详细介绍，参见 http://blog.csdn.net/duxinfeng2010/article/details/9000693。
- CocoaLumberjack　这是一个集快捷、简单、强大和灵活于一身的日志框架。关于 CocoaLumberjack 的详细介绍，参见 http://www.cocoachina.com/industry/20140414/8157.html。
- YAJL（Yet Another JSON Library）　是一个小型事件驱动 (SAX 风格) 的 JSON 解析器，采用 ANSI C 编写。关于 YAJL 的详细介绍，参见 http://mobile.51cto.com/iphone-386666.htm。
- zlib　用于解压缩 Zip 包。我们在 App 中打包 HTML5 页面时会用到这个东西。关于 zlib 的详细介绍，参见 http://xzhoumin.blog.163.com/blog/static/408811362013143824 39/。

9.11.3　Android 篇

- aSmack　说到 aSmack，自然要先提提 Smack。Smack API 是一个完整的实现了 XMPP 协议的开源 API 库，而 aSmack 则是 Smack 在 Android 上的构建版本，于 2013 年 2 月初迁移到 GitHub 上，该资源库并不包含太多的代码，只是一个构建环境。开发者可以利用该 API 进行基于 XMPP 协议的即时消息应用程序开发。项目地址：http://www.open-open.com/lib/view/home/1368327419922。
- EventBus　是一个发布 – 订阅的事件总线，是为 Android 量身打造的开源项目。看到发布 – 订阅，我们自然就会想起观察者模式，其实这个开源项目就是按照这个思路实现的。关于 EventBus 的详细描述，请参见：http://blog.csdn.net/lmj623565791/article/details/40794879。

㊀ 关于 WebViewJavascriptBridge 的详细介绍，请参见 http://www.cocoachina.com/industry/20131230/7628.html。

㊁ 关于 zepto.js，请参见 http://www.cnblogs.com/huangtenghui/archive/2013/03/05/2944614.html。

㊂ 关于 MRAID 的详细介绍，请参见 http://blog.chinaunix.net/uid-22312037-id-4238431.html。

9.11.4 其他

- Pinyin4j 它是 sourceforge.net 上的一个开源项目，可以将汉字转化为拼音，这样的话，当我们从服务器取出中文城市列表的数据后，就可以通过输入全拼或者拼音首字母，迅速的查找到相应的中文城市了。关于 Pinyin4j 的详细描述，请参见：http://blog.csdn.net/woshixuye/article/details/7462081。

  在此，我谈一下对这个技术的一点看法。我认为不该在客户端做这个事情，太重了。应该由服务器端在返回中文城市数据时，额外返回该城市的全拼或者拼音首字母这两个字段。把复杂的业务逻辑放在服务器端。
- Countly 精益化运营，需要一个优秀的统计分析平台，其中比较优秀的有 Countly 和 Google Analytics，后者又简称为 GA。

  市面上的 App 对 GA 使用得比较多，对 Countly 了解不多。Countly 是一款专门给移动应用的统计分析平台，而且它居然是开源的。Countly 由两部分组成，APP SDK 和服务器，服务器是建立在 Node.js 和 MongoDB 之上的。如果厌倦了第三方平台的局限性，可以考虑使用该开源平台。

9.12 竞品技术十瞥：版本策略与 App 彩蛋

9.12.1 版本策略

同一时间比较了 100 款 App 的 iPhone 和 Andriod 版本，有以下几种版本策略：

- 保持一致。比如，当前版本都叫 6.0.0。下一个迭代版本都叫 6.1.0。但下一个迭代版本绝对不能是 6.0.1，因为 6.0.0 版本在使用中发现严重问题要紧急修复紧急发版时，就不好定义版本号了。
- 第二位用于版本递增。比如 6.1.0，6.2.0，6.3.0；第三位用于紧急发版，比如 6.1.1，6.1.2，6.1.3。
- 一个奇数，另一个偶数。如 iPhone 3.1.3 和 Android 3.1.4。下个版本则是 iPhone 3.1.5 和 Android 3.1.6。其实这也是没给自己留后路，如果 3.1.3 发现问题要紧急发版，将没有版本号可用。

我还见过 3.1563 这样的版本号，也曾见过 6.0.1.2 这样的版本号，但都属于非主流，这里就不多做介绍了。

9.12.2 App 彩蛋

1. 我发现一些 App 的包中总是有些有趣的文件：

- 比如 apk 包中掺杂 gradle 等文件，这一看就是 Android 打渠道包时留下的垃圾。

- 比如 ipa 包中掺杂 .h 文件，其中有程序员的签名，让自己的大名出现在几千万用户的手机中，我也是醉了。
- 比如包中有些文件会带有 Test 前缀，这明显是用于自动化测试的。

以上种种，从侧面表现出 App 的开发团队的水平很业余。

2. 哥们，裤子拉链开了！

有时还能从 App 的设置页面看到“调试”这样的后门，点进去看到的是专门给开发人员联调、测试人员验收时使用的页面。这就上升到线上故障了。

3. 图片不要使用中文名称

建议还是全都使用英文名称的图片名称。中文名称多少显得有些业余。我还见过“+.png”这样的图片名称，对应的就是一个加号图片。

4. 大文件

我见过有些 App 里面会有 7.6M 的图片，我打开一看，其实就是“添加收藏”的按钮图片。把这张图片扔进 App 中的程序员，可以吊起来暴打一顿了。

我还见过有的 App 嵌入 1 个 2.6MB 的字体文件，这相当不划算。如果只用到这个字体的几个字，那么还不如将它做成 icon 放到 ttf 字体文件中。

5. Zip 压缩包用密码

如果你觉得自己的配置文件、Lua 脚本、HTML5 真的很重要，不想被别人看到，就把它们压缩为 Zip 包吧，并加上一个密码。

9.13　本章小结

“当我们认为自己对这个世界已经相当重要的时候，其实这个世界才刚刚准备原谅我们的幼稚。”这句话时刻警醒着我，不要沉迷于以往取得的成绩，作为技术负责人，要与时俱进，要有敏锐的嗅觉，才能跟得上时代的潮流。要永远抱着谦卑的心态，去学习竞争对手先进的技术和理念，才能时刻在这个行业占据着主动地位。

第三部分 Part 3

项目管理和团队建设

- 第 10 章　项目管理决定了开发速度
- 第 11 章　日常工作中的问题解决
- 第 12 章　无线团队的组建和管理

打造一支李云龙风格的独立团。

这部分讨论三个主题：移动项目管理、线上问题分析与解决、团队建设。

个人技术水平再高，不懂得怎么带项目带团队，也是白搭。我做过很多失败的项目，也按期交付过优质的项目。成功不可复制，但是失败的经验教训一定要吸取。我的经验心得是，项目是否成功，一半取决于团队的技术水平，另一半取决于项目经理的经验，比如说，如何拆分需求到若干次迭代，如何激励士气，如何控制风险。

对于移动互联网公司，日常工作中除了做项目，还要解决线上各种各样的问题。如何快速准确地定位问题需要经验积累，每一个成熟的方法背后，都有一个充满血和泪的故事。

我讨厌教条式的KPI考核，我对员工的评价是基于他的潜力以及成长。有潜力的员工永远是我欣赏的。但是一个团队不能全都是这样的人，要像西天取经组合那样兼容并包。我喜欢招有潜力、有悟性、性格比较外向的员工，即使当前的技术水平还不够，但我会通过各种方式将其培养成为一流人才，看人才逐步成长，就像雕琢玉器一样，是一个享受的过程。

第 10 章 Chapter 10

项目管理决定了开发速度

想改变现状，首先要深入一线，熟悉现状，知道了一线人员的苦与痛，然后才能一小步一小步地优化，步子太大，容易扯着蛋，后期可以把步子迈得大一些，最终朝着你所期望的那个方向逼近。

一次性把流程全都改变了，一线人员首先会不习惯，从而达不到效果，但是各种报表是好看的，上报给大老板的结果都是好的，直到最后一天捂不住了，才会发现延期或者驴唇不对马嘴，而项目负责人这时候总能找到脱身的理由，比如团队执行不到位，其他部门配合不够，然后轻描淡写地说“先解决问题而不追究责任”。于是，项目每次迭代都会延期而得不到本质上的改变。

王安石变法不就是个很好的反面教材吗？那次变法具备了项目管理中最忌讳的几件事情：

1）领导者高高在上，执行者欺上瞒下。

2）理想美好但是不切实际，最后连农民阶层这样的“受益者”都反对。

3）一次性改变太多，导致树敌太多。很多人不理解不支持，尤其是既得利益受到损害的阶层。

无线项目的管理，与之前的所有项目都不同，因为它涉及 iOS、Android、MobileAPI 和 QA 团队的相互依赖、分工协作的事情。以下是我这几年来在无线领域摸着石头过河的经验总结，其中也走过不少弯路，仅供大家参考。

10.1 项目管理中的三驾马车

对于无线研发部门而言，一个完整的团队，应该包括产品经理、开发、测试这 3 个团

队，我们称之为“三驾马车”。其中，

- 产品经理是三驾马车的灵魂。
- 开发团队是三驾马车的主力。
- 测试团队是三驾马车中的保证。

要想项目跑得快，一定要搞好三驾马车之间的关系。团队之间越默契，效率越高，质量越高。

我们需要为三驾马车配备一个驾驶员，也就是项目经理。因为现在互联网公司都走敏捷流程，所以又称这个角色为 Scrum Master，他负责把三驾马车快速而平稳地驾驶到终点。

10.1.1 为什么不能没有测试团队

我见过有些公司没有测试团队，而是让开发人员自测，产品经理验收。这是一件非常不靠谱的事情，原因如下：

1）开发人员自测，只会按照自己编程的逻辑进行测试，很多时候，局外人——也就是测试人员，因看事情的角度不同，才能发现更多的问题。

2）测试过多地占用了产品经理的精力。产品经理应该更多地关注产品本身，包括页面转化率、用户体验，等等。其实他们只要在发版前验收一下需求（逻辑、UI）是他们想要的就可以了。而测试人员的工作就是一天到晚执行测试用例，想方设法发现 bug。

3）测试工作不是产品经理的专长，很多情况，比如边界条件，就是产品经理测不到的地方。试想一个登录功能，产品经理的验收标准仅仅是点击登录按钮能进入个人中心就够了，而测试人员的测试用例却有 50 多个。

4）产品经理对 bug 的关注度不够，于是经常出现开发人员修复一个 bug，但是产品经理几天后才会验收的情况——他们常常是把 bug 积压到一定数量后才批量处理，这样比较省事，殊不知这样的风险很大，如果最后才发现有 bug 并没有完全修复而此时又临近发版，那么就只能是项目延期或者忍痛屏蔽该功能了。

以上 4 点，是我这几年来作为项目管理者所观察到的情况，由此而验证测试团队在敏捷开发流程中不可或缺的地位。

一个好的移动团队，至少要有 2 名测试人员，开发和测试比大约是 6:1。也就是说，1 个测试人员对 2 个 iOS 开发人员 +2 个 Android 开发人员 +2 个 MobileAPI 开发人员。测试团队应该担负的工作如下：

- 召开测试用例评审会——相当于需求二次评审。测试人员、开发人员、产品经理在会议上对需求达成一致。
- 手动测试。
- 全功能回归测试。
- 探索性测试。
- 渠道包测试。

- MobileAPI 发布上线前的测试工作。
- 压力测试。
- Monkey 测试。
- 客人投诉回访。

在很多公司里，因为过度强调开发团队的重要性，测试团队往往沦为附庸。于是，测试团队往往是被项目经理安排去做某项工作，而不是自主选择该去做什么事情。被动的久了，自然就形成鸡肋了。

测试团队应该在项目中有自己的话语权，一方面他们要对质量负责，另一方面，他们要及时反馈迭代过程中发现的各种风险，比如：

- 当测试资源不足时，应该告诉项目经理哪些功能因为没有测试资源是不能上线的。
- 在发版前如果发现 bug 很多，应该通知项目经理这次迭代的风险。要么延期，要么砍功能。但是决不能带着严重的 bug 上线。

10.1.2　产品经理应做的事

当产品经理不再承担测试的工作时，就应该把更多精力放在需求本身了。他们应该花 80% 时间在需求上，以确保需求尽量清晰，至少自己想明白了，才能让开发人员和测试人员也明白。

另外 20% 时间干什么呢？

首先他要参加开发和测试人员的每日站例会，这样才会知道开发人员在哪些需求上遇到了逻辑问题，从而及时做出调整。这个站例会很重要，如果产品经理不能保证每天都参加，有可能直到最后一天才告诉团队这不是他想要的产品。

其次，测试团队提的 bug，经过开发人员分析后，发现是产品需求的问题，会将 bug 转给产品经理。产品经理每天都要检查分配给自己的 bug，要么重新定义业务逻辑，让开发人员照此修改；要么降低 bug 优先级，本期迭代不修复。

验收需求。在开发工作结束、测试工作接近尾声时，产品经理要安装一个开发版 App，验证实际开发出来的功能是否与他的需求一致。

最后，也就是发版前，产品经理要根据本次迭代的 bug 清单，根据测试团队的反馈，决定是否发版——bug 太多可能会延期。

产品经理在项目中的职责很简单，就是定义什么是对的，然而很多公司为其赋予了太多的责任，比如他们要对项目进度负责，所以每天要组织站例会，他们要对项目质量负责，所以每天要进行测试。请问，这样的产品经理还会有什么天马行空的想法呢？用我老板的话讲，把产品经理当牲口使。

很多互联网公司在设计 App 时，都是把网站上的成熟产品搬到 App 上，这不需要太多的产品设计，所以把产品经理当牲口使这种策略一度是可行的。但随着网站内容和 App 内容渐趋一致后，如果还想在 App 上有所突破，比如 App 上的订单超过网站上的订单，这就需要在用户体验、运营策略上下功夫了。

10.1.3 开发人员的喜怒哀乐

我是程序员出身，也曾过着上班时穿拖鞋短裤的IT男生活。我深知技术男喜欢什么，不喜欢什么。

一个软件/互联网公司的成功与否，很大程度上取决于这些技术男也就是开发人员的存在，只有他们能把产品经理的想法或者尝试付诸实践，这才是其价值所在。

开发人员要想尽一切办法实现需求，而不是一天到晚发牢骚，说这个需求做不了那个需求做了也没用。值得欣慰的是，牢骚归牢骚，再苦再累，绝大多数一线开发人员还是会咬着牙把需求按时做完，诚然，熬夜加班是必须的。

这众多的牢骚之中，我听到抱怨的最多是：

1）一句话需求。

2）开发过程中，产品需求频繁变动。

3）产品经理搞不清楚业务逻辑。直到开发过程中才发现有问题。

4）UI设计图、切图、标注图不到位。

所有这一切，只能怪互联网公司节奏太快，不能像软件公司那样按部就班的工作。

我在接下来的章节，就是要想尽各种办法，来解决这些问题。

10.1.4 项目经理的职责

在敏捷流程中，项目经理也称为Scrum Master。根据我多年做Scrum Master的经验，我的切身体会是：

1）项目经理不需要知道太多的业务逻辑，他只关心项目进度就够了。

2）一个团队是否高效，完全取决于项目经理的水平。

项目经理的事情非常琐碎，他不需要技术和业务知识，但是却一天到晚跟着项目进度走，和各个团队沟通，协调资源。以下是项目经理的几项职责：

1）搜集开发计划和测试计划。分配开发任务和测试任务是开发和测试团队各自的Team Leader的工作。他们会把工时汇总给项目经理和产品经理，然后由项目经理协调三驾马车，在规定的期限内，尽可能多的排进更多的需求。

2）主持每天的站例会，并发送会议记要。绝对禁止发送报喜不报忧的会议记要。

3）积极面对风险，及时调整计划，以减少风险。这句话说起来简单，实际操作起来绝非易事，很大程度上取决于项目经理的经验。

4）及时解决各个地方的瓶颈。

5）推动bug的修复情况。

6）监督开发团队的冒烟测试、测试团队的探索性测试、产品经理的验收工作。

7）如果开发流程需要同步到jira，那么项目经理要负责创建Story和Task。为了提高开发人员的工作效率，项目经理可以在每天开完站例会，了解完所有人员的进度后，根据会议记要，帮助开发人员在jira上同步进度。

我个人是不喜欢 jira 的，因为操作起来太麻烦，不如 Excel 简单明了。不同项目经理有各自使用顺手的工具，半个小时内能完成同步进度的工具都是好工具。

8）项目结束后，召开总结会，好的地方继续保持，做的不好的地方，集思广益想办法解决。

以上介绍了无线部门敏捷开发中各个团队的作用，接下来介绍如何搞敏捷开发。

10.2　优化团队结构，让敏捷流程跑得更快

敏捷流程中，切忌僵化的团队组织结构。为了让敏捷流程跑得更快，我们应该不断地优化团队的结构和开发模式，不断地尝试，发现好的地方要坚持，发现行不通，观察一两个迭代后果断撤回来，再去想别的办法。本节我将介绍敏捷过程中的一些优化方案。

10.2.1　平行模式还是垂直模式

由于移动互联网的开发模式有别于传统互联网——它是由 Android、iOS 和 MobileAPI 三个团队组成的，所以选择什么样的开发模式是很有讲究的：

平行模式，就是 Android、iOS 和 MobileAPI 各自为一个独立的团队，在项目初期，团队间制定好 MobileAPI 接口的格式，约定好联调时间，就可以各自开工了。然后到了联调时间，再进行集成测试。

垂直模式，就是按照模块，拆分出若干小的团队，比如说会员中心，就由一个小团队负责这个模块，有相应的 Android、iOS 和 MobileAPI 开发人员，以及产品经理和测试人员。

这两种模式我都尝试过，分别介绍如下：

1. 垂直开发模式

我曾经做过一个 B2C 项目，使用的就是垂直模式。团队 10 个人，其中：

- 1 个产品经理。
- 1 个项目经理。
- 2 个 Android 开发人员。
- 2 个 iOS 开发人员。
- 2 个 MobileAPI 开发人员。
- 2 个测试人员。

这个项目做了 2 个月，延期 4 天上线，排除掉过程中遇到的很多不可抗因素（比如公司的新人培训、测试环境的不稳定性，等等），算是一个比较成功的项目。

我一直在思考这个项目成功的原因，因为之前做的很多项目都要延期很久，其中有一点非常关键：垂直模式的开发模式使得这只团队非常高效。当 App 开发人员发现有个 MobileAPI 接口不能使用时，他会抱着笔记本坐到 MobileAPI 开发人员旁边的座位上，一起

联调，直到解决问题。测试人员从前端发现 bug，会从 App 往下一路查到 MobileAPI，直到 bug 修复。所有人都在对一个团队负责，为一个目标而努力。

2. 平行开发模式

仍然是上述这种拆分成若干独立小团队的开发模式，在其他公司却行不通。我们虽然将开发团队按照业务模块拆分为若干个独立小团队了，但是战斗力并没有得到加强，因为拆分前并没有确保每个开发人员都熟悉自己所负责的模块。后来，有开发人员离职，随着 2 ~ 3 名技术骨干的离开，这种模式就走不下去了。有些组只剩下一些实习生，难以维持下去，只能合并到其他组。

另一方面，由于 Android 和 iOS 开发人员被分散到各个组，以至于我想做重构的时候，每个组的进度不一致，有的组有时间，有的组还在做需求，导致重构的事情推不下去。

于是，我们又退回到平行模式，重新把团队按照技能划分为 Android、iOS 和 MobileAPI 团队。

由此而吸取的教训是，在团队没有成规模之前，不宜拆分。这就好比一只手有五根手指，攥成拳头打出去才有力量。另一方面，即使是要做拆分，比如一支 10 人的 Android 技术团队，也是每次拆分出 2 个人，一步步的进行，而不是一下子就把 10 个人拆成 5 支 2 人团队了。

每次拆分出的这 2 个人，就雷打不动做这个模块了。不能说哪天其他模块没人了，把他们调回去，临时支援 1 ~ 2 周，这是不行的。必须把人固定在模块上，才能培养出这 2 个人的业务知识。

介绍完上述两种开发模式，可以观察到适用于无线开发团队的开发模式。从短期看，人少的时候，平行模式比较有优势；从长期看，随着业务规模的扩大，垂直开发模式是大势所趋。毕竟，对于 Team Leader 而言，手下超过 6 个人就会有管理上的问题。

10.2.2 让 HTML5 站点和 MobileAPI 的进度提前一个迭代

做了这几年的迭代，我的切身感受是，一个功能点，只要是 MobileAPI 和 App 同时开发，就会延期。而那些现成的 MobileAPI 接口，App 开发人员可以直接拿来使用，一般都不会延期。

我尝试过每次让 HTML5 网站先行，请他们先去扫雷，他们会和 MobileAPI 早一个迭代把这个功能在 HTML5 站点实现了，下一个迭代再接入 App。HTML5 网站的特点是开发周期短，往往一个页面 App 需要 1 天，而 HTML 5 页面一个小时就做好了。

10.2.3 如何进行模块化分工

任何一个企业级 App，都是由若干个业务模块组成，比如说会员中心、美食、电影等等。我们要确保每一个模块都有 1 ~ 2 个开发人员非常熟悉它的业务逻辑，长时间在该模块上开发和维护。

我见过 10 人的 Android 开发团队，因为没有明确的业务模块分工，导致每次迭代，负责开发某个功能的开发人员额外还需要 1 ~ 2 天熟悉代码和业务的时间，直接导致了开发效

率的下降。

当模块化工作落实到每一个开发人员身上时，你会发现，每当产品经理提一个需求，比如说美食模块，那么 Android 开发、iOS 开发、MobileAPI 开发、产品经理、测试人员会自发组建一个 QQ 群，在里面讨论、沟通该功能点的所有事情，直到开发完成、测试人员和产品经理验收通过。他们会协调时间，以确保该需求准时完成。

在模块化的实际操作中，被划分到某个模块的开发人员，不仅仅要熟悉该模块的业务逻辑，从代码角度来说，还要清晰地知道该模块包括哪些 Activity、Adapter、Entity 和其他一些类。我们在第 1 章 1.1 节介绍过，要对项目进行重构，把项目按照业务模块进行组织，也是基于这个目的。

模块化分工是一个需要长时间磨合、调整的过程。我的切身体会是，要确保“让合适的人做合适的事”，比如说：

- 并不是每个人都能接手“会员中心”这个模块的，这个模块包括个人信息、各种订单信息、消息盒子、充值、红包、积分等等很零碎的功能，通常没有太多的技术含量，而大多是脏活累活，所以需要一个沙僧型任劳任怨的开发人员来负责。
- 对于公司最重要的业务模块，要委派踏实勤奋的开发人员，踏实是确保质量高，不会犯愚蠢的错误以至于影响公司生意，勤奋是确保任务做不完时能加班。因为往往这块业务每次迭代都有大量的需求，所以还要配备候补开发人员，以备不时之需，从而才能消化所有的需求。
- 技术能力强的人往往效率要高于其他开发人员，所以要经常把有挑战的工作交给他们去做，比如说 Monkey 日志分析，比如说线上 Crash 分析并修复。
- 沟通能力强的，这种人适合解决每天的用户投诉，从而准确地定位问题。

由此而想到另一个问题，如何开发工时估算得更准？只要做到了模块化分工，让熟悉该模块的开发人员估计工时，就能评估出精准的工时，从而知道每次迭代最多能做多少需求。

10.3　App 敏捷开发流程

每个公司都有自己的开发迭代周期，有 4 周的，有 2 周的，也有 1 周的。也不好判断究竟哪个开发节奏更好，只能说各家有各家的打法，各家有各家的烦心事。下面就让我来逐一介绍一下这几种开发流程。

10.3.1　四周时间的开发流程

1. 巧妙安排迭代间隙

敏捷开发的周期，包括从需求准备、排期、开发、测试到上线、发版，可长可短。

在一个月迭代周期开始之前，我先介绍一下，我们都干些什么？

这期间，通常会有 1 ~ 2 天时间，除了让团队休整，该约会的约会，该学车的学车，还要做以下这些事情：

1）总结上次迭代的若干问题。也就是所谓的 post mortem，这个总结会议很重要，需要把上次迭代做得好的和不好的，都列举出来。好的，我们下次迭代要继续遵守；不好的，要在下次迭代想办法解决，落实到具体的负责人。

2）修复上次迭代来不及处理的 bug。每次迭代都会有一些 bug 遗留下来，之所以不修复，是因为改动这个 bug 可能会导致很大的隐患，或者测试团队没有时间去验证，或者需求不清楚，需要产品经理将其细化，在下期迭代作为一个 Task 来完成。

3）做一些代码上的重构工作。包氏法则之一：永远以产品需求为最高优先级的 Task，在想方设法完成了需求之后，再利用剩余的时间来做代码优化、项目重构的事情。重构工作一般放在迭代前期进行，这样测试人员才可以将其也作为一个测试任务去评估时间。此外，在项目前期完成重构，可以通过接下来长达 4 周的迭代时间来发现重构所带来的各种问题并及时修复。

4）讨论新需求，划分到具体的开发人员和测试人员，评估出工时和工期。这时要求产品经理的需求文档已经到位了。

项目经理召开一次全部人员参加的需求确认会，由产品经理讲解每个需求。为此，要确保每个模块都有 1 ~ 2 名开发负责人，从而保证该模块有需求时至少有 1 个人立刻能上，当该模块需求过多时，迅速把第 2 个人也补上来。同时，也降低了项目对人的依赖，以确保任何一个人都有备胎。这是团队建设必须做、并持之以恒去做的一件事。

把需求划分到人，听产品经理讲完需求之后，就该评估工时了。当收集到所有开发人员报上来的工时后，你会发现：

- 有人工时过多，超过了 2 周，有人则不足 2 周，这时项目管理者就要局部调整 Task，以确保每个人员的工时都控制在 2 周以内。对此，我称之为“拆东墙补西墙”。开发工作控制在 2 周是绝对有必要的。2 周之后不再做额外的需求（除非很紧急），不再做任何重构（除非问题很严重），以确保测试阶段项目的稳定性。
- 一个简单的 Task，却需要 3 天才能做完。有的程序员喜欢给自己多留一些 buffer，以确保各种天灾人祸所导致的 Task 延期，但作为项目管理者，则更希望每个 Task 的 buffer 控制在半天以内，这样才能制定出比较准的迭代计划。有的程序员则属于偷奸耍滑的类型，他们会把工时估的很宽裕，从而每天有充足的时间去逛淘宝、QQ 聊天。这时，项目管理者所要做的是，拎起笔记本，到每一个开发人员座位上，对有水分的 Task，一起分析需求，重新评估工时，把“水分”挤出来。

如果绞尽脑汁排出来的开发工时还是超过 2 周，项目经理这时就要联系产品经理，砍掉一些不必要的需求，从而踩住 2 周 code complete 那个时间点，以确保本次迭代不会有太大风险。

我们漏了一个环节，那就是测试团队的测试工时。有时候，即使是开发能在这 2 周把所有需求都做完，测试资源不足，也会需要产品经理适度砍掉一些需求，以确保测试时间够

用，保证那些重要的功能点。或者把那些只涉及 UI 改动的需求转给产品经理来验收。

工时安排妥当之后，接下来需要每个开发人员为自己分到的 Task 制定工期，即先做哪个、后做哪个。

要想把工期排好，首先要解决 App 对原型图、MobileAPI 的依赖性：

- 有些需求需要美工给出原型图和切图，什么时候给出，对工期有很大影响。
- 有些需求需要后端 MobileAPI 提供数据，什么时候 MobileAPI 能完工，或者退而求其次，事先制定好 MobileAPI 接口，给出假数据也能接受。
- 如果 MobileAPI 的进度比 App 的进度能提前一个迭代周期，那么就能避免 App 和 MobileAPI 并行开发所带来的风险。

以上都是项目管理者所要去协调沟通的。

5）在迭代正式开始的前一天，开一个冲刺会，标志着本次迭代正式开始。

如果前戏都做得很充分了，这个冲刺会其实就是走个形式，开发人员、测试人员、产品经理聚在一起，然后宣布下期迭代从明天起正式开始，上线时间点是哪一天。

会议控制在 10 分钟。也许有人会问，10 分钟够吗？通常会有团队在冲刺会上把项目分配、评估、工时和工期也一起讨论，所以开一天才能结束。其实大可不必，只要在会前把这些工作做足了，和每个开发人员都充分够通过，有了结论，那么在动员大会上，只要宣布这些结论就可以了，不需要再讨论。

从以上 5 点看出，迭代开始前的这几天，是项目经理最忙碌的日子，他们要使尽浑身解数，在迭代开始前把这些准备工作都做好。稍有延迟，项目进度就会受到影响，10 多个开发和测试人员就会等你，项目经理耽误 1 个小时，整个团队耽误就是 10 多个小时。

项目经理切记，永远不要让自己成为瓶颈。

接下来的 4 周就是真刀真枪的迭代时间了。

2. 控制 4 周迭代的节奏

在这 4 周的迭代时间里，要干的事情很多，把这些事情标注在时间轴上，如图 10-1 所示。

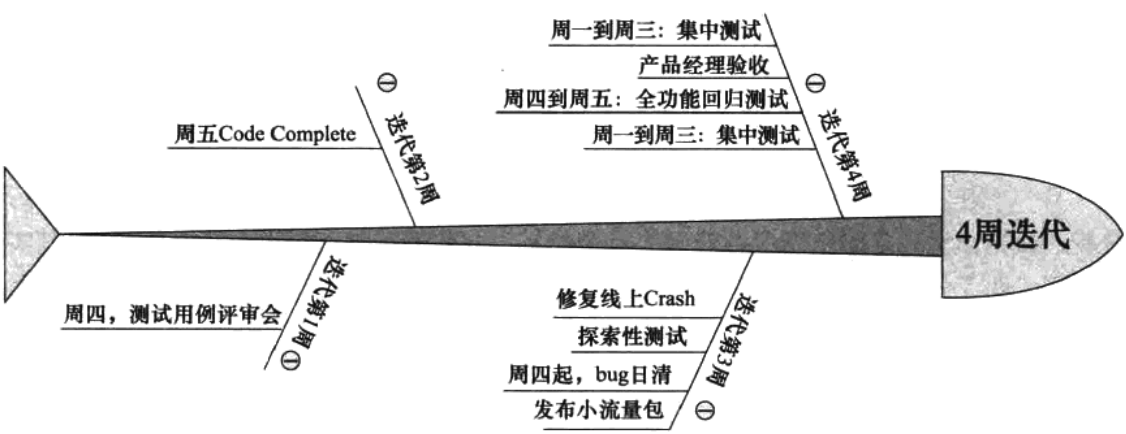


图 10-1　4 周迭代流程

1）开始两周，是开发时间。

在这两周时间内，初期会有一个测试用例评审会，根据我的经验，就是需求二次确认会。一般而言，第一次需求确认会，开发和测试只是了解需求，当时提不出太多的意见。只有经过几天的沉淀，才会发现，有些需求并不合理，所以我们每次开测试用例评审会，就会发现这也有问题，那也有问题，于是这个会议就变成了需求二次确认会。我们在评审测试用例的时候，也把需求最终确认了下来。

在这 2 周的开发时间里，会遇到各种狗血的事情：

❑ 上一个版本发现了重大 bug，要紧急修复并发版。

这个是比较费开发和测试团队时间和精力的一件事。我每次的处理方式是调整 1 个 Task 延期到下期迭代完成，以此来解决资源的不足。

❑ 陆陆续续发现一些线上的 bug，虽然不是很严重，但要放到本次迭代内修复。

作为项目经理，我一般将此当作正常迭代中的 bug 来修复，不会影响迭代的工期，但是遇到架构上的问题导致的 bug，可能就要排到下期迭代来完成了。

❑ 产品经理经常插入一些紧急的需求。

如果开发团队和测试团队都能消化掉这类紧急需求，那么就排到 2 周开发的工时中；如果测试团队时间不够，就开新分支来完成，下期迭代合并进来再进行测试；如果开发团队时间不够，那么就把还没开始的一个低优先级 Task 放到下期迭代，以优先完成这个新需求。

❑ 一些需求，临时决定本次迭代不做。

这样开发人员就有额外时间了，我一般安排他们去做之前一直拖欠的 Task，或者去做一些重构，但是考虑到测试资源的问题，所以每次都是做在新分支上，本次迭代并不上线，放到下次迭代去测试。

在这两周里，随着新需求一个个的提交测试，测试工作也慢慢展开了，并开始报了一些 bug。

第二周周五，所有功能都已经开发完成了，也就是所谓的 Code Complete。

2）第三周，测试工作进入全面测试阶段，每天的测试包都比前一天更加稳定。开发人员的日常工作是修 bug。

第三周要把高优先级的 bug 全都修复，不然难以确保下周 bug 日清。

此外，本周可以跑 Monkey 了，每天要有专人对 Crash 日志进行分析、归类，然后指派给相应的开发人员去修复。

另一项开发人员需要做的是，修复线上的 Crash。需要有专人对线上每天的 Crash 进行分析、归类，然后分配给相应的开发人员去修复。我的经验是，每天的 Crash 种类，其实差不多，昨天的某个 Crash，接下来一周都会出现，所以我每次只会分析连续三天的线上 Crash，基本能囊括线上的 90% 的 Crash。

为了确保开发质量，开发人员每天还会集中坐在一起进行冒烟测试。即每天集中测试一个模块，把发现的问题及时修复。

第三周的最后一天，应该确保所有的高优先级 bug 都修复了，可以进行正常的下单支付

流程。这时我们要打一个正式包，用于：

- 发给全国各地的分公司同事，请他们帮忙进行测试。我主要想知道全国各地是否都可以登录，包括 WiFi、2G、3G 和 4G 网络环境。
- 发布小流量包。有关小流量包的介绍参见 11.3 节的内容。

3）第四周，这周主要是测试人员进行集中测试、探索性测试和全功能回归测试。

前三天是集中测试，他们会集中所有测试人力，对所有新功能一起测试一遍。开发人员则要保证 bug 日清。与此同时，测试团队这几天需要每天组织探索性测试，及早发现 bug 及早修复，要逐个模块推进探索性测试工作。

第三天晚上是 Code Freeze。我们认为这个版本是比较稳定的，除非发现很严重的 bug，否则，不再改动代码。

周一到周三这三天中，产品经理要进行验收工作，把问题及时反馈给开发人员，及时修复。

周四周五是全功能回归测试，又名 Regression Test。这是最后一轮测试，这期间发现的任何 bug，我们（开发、测试和产品经理）都要评估，如果不是很严重，我们本期迭代就不修复了。可以按照先 iOS 后 Android 的顺序，每个 App 使用一天的时间进行全功能回归测试。

这周，我们还要密切观察小流量包发布后的线上 Crash 情况和安装新版本体验包的同事的反馈，对发现的严重问题进行评估和修复。

10.3.2　两周时间的开发流程

敏捷是什么？敏捷就是为了按时交付，不断调整策略，做到资源利用率最优化。至少我是这么认为的。

上一节我们介绍了 4 周一次迭代的敏捷开发流程。还能不能更快一些？可以，那就是接下来我要介绍的 2 周一次迭代的敏捷开发流程，如图 10-2 所示。

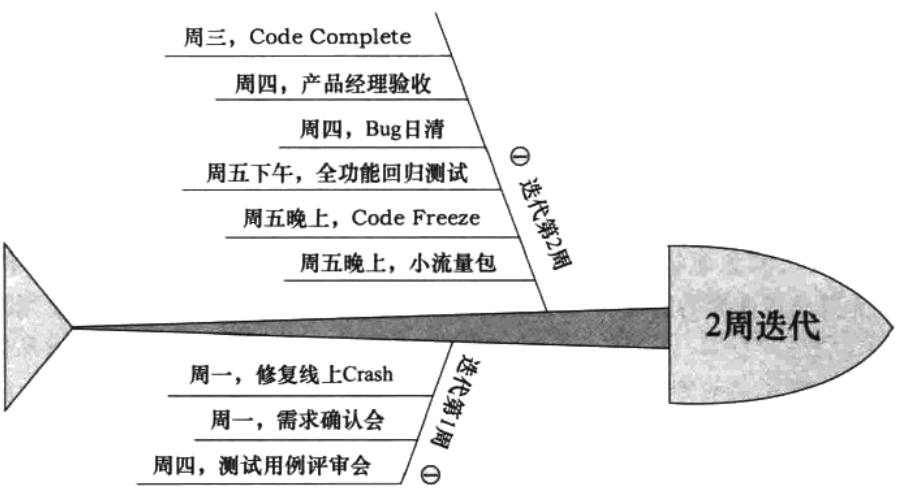


图 10-2　2 周迭代流程

时间少了，那就意味着每次迭代的功能也减少了，也不会有休整时间。每次迭代都是从周一开始，下周五晚上发版作为结束。

1）第一周的工作安排：

周一用于产品经理讲解需求、开发人员和测试人员分 Task、评估工时、排工期。此外，周一开发人员会比较空，一般用来修复线上的 Crash。

周二到第二周的周三，共计 7 天，所有需求都必须排在这 7 天完成。同时，要求 MobileAPI 在这天完成全部联调工作。

周四或周五，测试用例评审会。在这之前，测试团队编写测试用例。

2）第二周的工作安排：

开发工作持续到第二周周三下班前。周三晚上这个时间点，我们称之为 Code Complete。这个点延期了，后面的工作都要顺延。

周三起，项目经理组织开发人员进行冒烟测试，周三是 Android 和 iOS 各测各的，周四是两个团队交叉测试。

周四一天，产品经理对功能进行验收，如果可能，这一天尽量提前，以便于产品经理不满意，开发人员有更多时间进行修改。

周四要求开发人员 bug 日清。同时测试人员要对周三开发人员提测的功能进行测试。

周五上午测试人员验证 bug 是否全都修复。

周五下午测试人员进行全功能回归测试工作。

对于周五发现的所有 bug，我们只修那些严重程度高的 bug，bug 是否严重，由产品经理说了算。

周五晚上封版。我们称之为 Code Freeze。iOS 会提交 AppStore 审核，为保证 iOS 和 Android 同时发布，Android 当天是不能发布的，因此只是在主干上新建一个分支，该分支用于 Android 发版。一般来说，AppStore 审核通过 iOS 新版本要 1 周时间，在此一周内，Android 发现紧急 bug 还有机会修复，但原则上不再修改代码。

按照这个节奏，我们能确保每隔两周就能提供一个新的 App 版本，如果有延期，就需要周末加班补齐。

10.3.3 一周时间的开发流程

随着无线团队的急速扩充，我们会把无线团队按照业务线拆分到各个部门，你会发现，无论是 4 周还是 2 周的迭代，都难以协调各个部门，让他们按时完成功能，以保证准时发版。

我们不妨每周五 App 发一次版本，这是雷打不动的节奏。但是各个部门可以自行安排自己的发版时间，比如有些大功能要做两周，那就两周之后再发布这个新功能，而对于那些零零散散的小功能或者 bug 修复，则放到每周的发版中，不至于让用户等很久。

这就好比在地铁站等地铁，每 3 分钟都会开过去一趟，永远不会等乘客。而乘客有赶上

第一班的，也有赶上第二班的，这取决于他们的到达站台时间和着急程度。

App 一个月发 4 次版是很恐怖的，这会让竞争对手永远跟不上你的节奏。但缺点是用户不胜其烦，每周都要提示更新。

10.3.4　即时更新策略

还有没有更短的迭代流程？比 1 周还要短？有，那就是随时开发测试完成，随时提交到线上，而不借助于发版。

那就要用到插件化编程和脚本编程技术了。插件化编程仅限于 Android，这是一个庞大的主题，本书不会涉及这门技术。脚本编程就同时适用于 Android 和 iOS 了。目前业界普遍使用 Lua，以手机游戏行业用得最多。他们等不了 iOS 漫长的审核期，因为手游可能随时新增或修改地图、装备和剧情，所以他们会在已经审核通过的 App 中用 Lua 脚本做这些事情。其实应用类 App 也可以这么干，我接下来就准备招几个 Lua 程序员到 iOS 团队从事这方面的工作。

如果能做到上述的插件化编程或脚本编程技术，那么就可以随时发布新功能了。这是一件梦里都会笑醒的事。由此而回顾我们的敏捷开发流程，就没有迭代周期这样的概念了，我们将实现真正的敏捷流程，把所有 Task 都贴到白板上，做完哪个就发布哪个到线上。

10.4　项目经理的百宝箱

很多公司不设置项目经理，这是导致项目经常失控的原因之一。是否需要项目经理，取决于团队的负责人是技术型还是管理型，对于前者，是需要项目经理的出现的。

项目经理主要和人打交道，要具备良好的沟通技巧和协调能力，同时，他还必须具备其他几项技能，接下来我会逐一介绍。

10.4.1　项目经理的任务评估表

每次迭代的初期，最忙的就是项目经理了。他要在一天时间内完成以下工作：

1）汇总产品经理的需求，形成一个 excel。把这个 excel 下发给设计团队、Android 团队、iOS 团队、MobileAPI 团队、QA 团队，由各个团队的 Leader 把需求分配个具体的开发人员和测试人员。

我们以 Android 项目举例，这个 excel 应该由以下列组成：

- 需求名称
- 产品经理
- 需求地址（往往是 wiki）
- 设计师
- UI 提供时间
- MobileAPI 接口负责人（如果有）

- ❑ MobileAPI 联调时间
- ❑ Android 开发人员
- ❑ Android 工时
- ❑ Android 工期
- ❑ 测试人员
- ❑ 测试工时
- ❑ 测试工期

2）召开需求确认会，请产品经理为开发和测试人员讲解需求。在此之前，开发人员和测试人员应该按照自己分到的 Task 阅读相应的需求，以便于讲解过程中理解深刻。

3）搜集各个团队每个需求的负责人和工时、工期。设计团队提供设计师和 UI 提供的时间，MobileAPI 团队提供 MobileAPI 接口负责人和联调时间。Android 团队则提供每个 Task 的工时。

每个人的工时会不太均匀，比如说，每个开发人员只有 7 天的开发时间，必然有人超过 7 天，有人不足七天，这就需要开发团队的 Team Leader 来协调，对 Task 的分配进行微调，以保证每个人的工时都控制在 7 天，并且尽最大可能的消化需求。如果安排不下来，就要和产品经理商量，根据优先级删减需求。

在 Android 开发人员给出工时后，要根据 MobileAPI 的联调时间、设计师提供 UI 的时间来调整 Android 开发人员每个 task 的工期，以确保开发时 UI 和数据接口都是可用的。

4）把每个 Task 都写到小纸条上，贴到敏捷白板上，接下来的几周时间，就看这些小纸条的威力了。

5）有些公司倾向于把所有 Task 都用 Jira 来管理，这就要求项目经理额外再花一些时间去维护 Jira。

由于每天的站例会上都会过每个人的进度，并且还会把每天的进度作为会议纪要的一部分，发送给所有人。项目经理清晰的知道每个人每个 Task 的进度，所以可以由项目经理每天来更新所有人员在 Jira 中 Task 的进度，从而把开发人员和测试人员从 Jira 中解放出来，专心致志进行开发或测试工作。

此外，不允许有超过 2 天的 Task。对超过 2 天的 Task，需要开发人员和测试人员对其进行细化，直到每个子 Task 都控制在 1-2 天之内。

开发人员往往因为对某个模块不熟悉而预估出很多时间。这是不好的，会导致我们永远不知道每次迭代我们最多能消化多少需求。想解决这个问题，只能把 App 按照模块进行拆分，确保每个模块都有 1-2 名开发人员长期进行维护，这样估算出来的工时，就是相当准确的了。

10.4.2 贴小纸条的艺术

在敏捷白板上贴小纸条，是一门艺术。如图 10-3 所示。

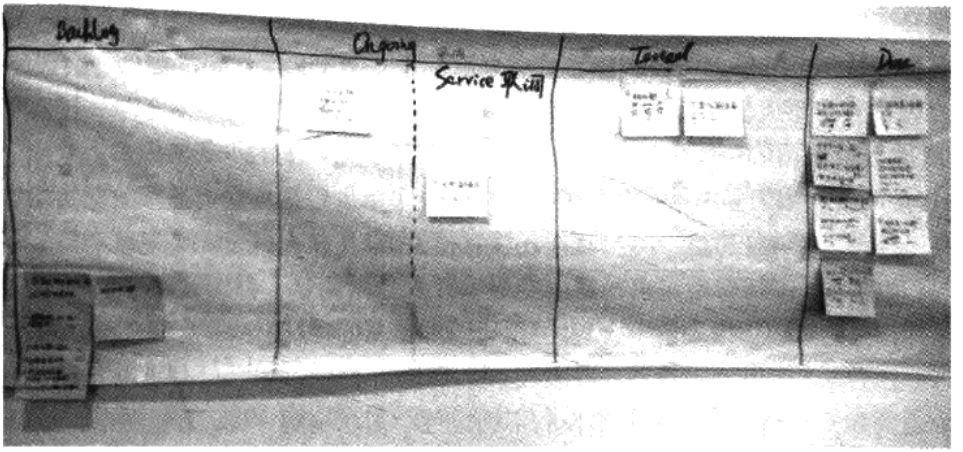


图 10-3　敏捷白板

这项工作最好在每次迭代正式开工前做好。每个小纸条上需要有以下几项内容：

- 需求标题
- 开发人员
- 工时
- 工期
- 测试人员

通常而言，白板上会有一个时间轴，按照敏捷流程而分为几个阶段：

- BackLog：待办列表。
- Doing：开发进行中。
- CC：开发完成，等待测试。
- Testing：测试中。
- Done：测试完成。

通常，Doing 阶段中还会细分出另一个子阶段：与 MobileAPI 联调。当然，这一步是可选的，因为有些需求不需要 MobileAPI 的支持。

迭代期间，会陆陆续续发现线上的 bug，或者加入新的需求，或者项目本身的代码优化，我们会将其写到小纸条上，暂时贴到 BackLog 中，有时间再做。这里的时间，不光指开发时间，测试所需的额外工时也要考虑。

最后，要防止小纸条粘性不够，经常掉地上，风一吹就不见了。我的经验是用胶带，这样比较牢靠一些。此外，小纸条的材质也很讲究，经常会发生写不上字的情况。要注意贴纸的正反面，只有一面是可以正常写字的。

10.4.3　敏捷迭代中的会议纪要

只要是一群人在一起开会，一定要有人做会议记录，然后把会议记录群发邮件给大家。

下面介绍敏捷开发过程中的四种必不可少的会议纪要及邮件。

第一种：站例会邮件。项目经理在站例会后，要立即发会议纪要的邮件，会议纪要的格式如下：

1）每个开发人员的进度。基本就是流水账，与敏捷白板上的小纸条同步。

2）提测功能，当天新提测的功能要用红色高亮显示，以区分之前提测的那些功能。

3）UI 和 MobileAPI 进度，列出目前还没有提供的 UI 和 MobileAPI 接口

4）发现问题，包括新增需求、需求变更、开发计划调整，都应该在这里列举出来。此外，还包括在敏捷过程中发现的不合理之处，比如 MobileAPI 与 App 的配合不默契、

5）风险评估，任何风吹草动，都要反映在风险评估中。项目经理要有足够的敏感度，在项目中遇到的人员请假、第三方依赖的不确定性、需求变更、bug 数量激增，等等，都是潜在的风险点，要如实反映在邮件中。

以上 5 点中，最重要的是第 5 点，不要怕得罪人，要如实反映项目中的潜在风险，只报喜不报忧的邮件是没有任何意义的。

第二种：测试团队邮件

在站例会的会议纪要中，我们会发现，这份会议记要中没有每日的测试进度和 Bug 情况。这是因为，测试相关的邮件要单独由测试团队于每天下班前发出，包括本次迭代中每个需求的测试进度，每个开发人员当天的剩余 bug 数量，每个测试人员目前还没验收的 bug 数量。

第三种：分析 Monkey 邮件

每天下班前，开发团队和测试团队要执行 Monkey 测试，跑一个通宵。每天上午，由测试人员统一把昨天晚上所有 Monkey 测试的结果发出来，然后由开发人员分析这些 Monkey 日志，尤其是崩溃的地方，发一封邮件出来，列举出每个崩溃发生在哪个页面，指派该模块的负责人去修复。

第四种：项目总结邮件

每次迭代结束后，都要举行项目总结会议，请每个团队成员给出本次迭代做的好的和不好的地方各 3 点，好的要继续发扬光大，并且看是否能做得更好，不好的地方要想办法解决，下次迭代不能还是这样，至少要减轻它的影响。由项目经理总结后发出邮件。

每次项目总结会上，都要对上次总结的内容进行回顾，看做得不好的地方是否有了改善。

10.4.4 开站例会的技巧

站例会，英文名为 Stand Meeting，因为是一群人每天都站着开会过进度，所以也有的人称之为站立会（或站例会）。

1. 早上开会效果会更好

每天我们都要开站例会，开发人员、测试人员、产品经理聚在白板前。有的团队早上开站例会，有的团队则是下班前开站例会。

早上开站例会的好处是，作为一天的开始，可以安排今天要做些什么。下班前开站例会的好处是，作为一天的结束，可以知道每天的进度是否正常，如果有问题，可以及时做出调整，等到明天早上才知道就晚了。两种方式我都试过。一开始是每天早上开站例会，但是一段时间后发现，虽然早上把工作都安排好了，但是当天的进度只有第二天早上才知道。久而久之，每天早上，总会有开发人员给我一个惊喜——各种延期。后来就改为每天下班前开站例会了。虽然能提前知道每天的工作进度，但是明天要做些什么，虽然今天晚上站例会都安排好了，但是睡了一觉后，第二天就忘记 80% 了。

于是过了几个月后，我又改回早例会的方式了，但是每天下班前我会走到开发人员座位旁，简单询问每个人当天的进度，以确保没有太大的惊喜。一段时间后，发现效果显著，每个开发人员的剩余价值都被榨了出来，在效率提升的同时，我也发现自己的强迫症更加严重了。

2. 务必全员准时参加

开站例会一定要准时。定好了 9 点半，就一定在那个时间把人都召集到白板前。项目经理作为会议的组织者首先不能迟到，否则整个团队也都会上行下效。

任何人都不希望中途被打断，希望集中精力做事情，尤其对于工程师而言，他们最抵触开会，抵触的直接表现就是开站例会的时候懒懒散散，不准时参加，一定要忙完自己手里的事情再过来——我也遇到过这样的情况，我的经验是，提前 5 分钟走到团队工位，提醒每个开发人员和测试人员把手头工作收一收，5 分钟后准备开会。

此外，每个人的“生物钟”不太一样，慢慢调整每个人员的生物钟，不要与站例会冲突。当然，人有三急，遇到突发情况，也没有办法。对于因故不能参加会议的同学，等他有空了，再单独和他同步进度。

开站例会一定要确保开发人员、测试人员、产品经理都在场。其中，开发人员和测试人员很重要，要确保他们尽可能都参加。如果再把七八个产品经理也包括进来，那么二十多人的站例会就不是敏捷了。这说明团队大了，需要拆分了。一个敏捷团队要控制在 10 人以内。

曾经有一段时间，站例会每次都有将近 20 人参加。于是，我尝试过把站例会按照模块拆成两个小的站例会，这样每次就有 10 个人参加会议了。但这样做的前提是，开发和测试团队都已经实现了模块化，每个模块都有固定的开发和测试人员。

3. 站例会控制在 15 分钟

就算是 10 个人的站例会，也要控制在 15 分钟。每人介绍一下自己的开发进度和测试进度，各个团队的 Leader 说一下今天要做的一些公共的事情。需要牢记的是，每件事讨论不能超过 2 分钟，一旦发现 2 分钟说不清楚，那么项目经理就要站出来打断他，记下这个事情，会后叫上相关的人再详细讨论。

项目经理要控制站例会的节奏，不能跑题。我经常犯这样的错，说着说着就不正经了。

另一方面，因为参加会议的人很多，所以大家不要私下开小会。问到自己就说，否则就不要开另一个话题和旁人聊下去。

10.4.5 如何确保项目不延期

我带团队做过很多新项目，新项目就是从无到有。说老实话，开始的几个项目我做得并不好，原因有几个：

1）估算工时过于乐观，以至于虽然每天我也参与大量的开发工作，和团队加班到九、十点钟，但是仍然延期。

2）新项目因为一切从零开始，所以会有各种狗血的事情中途发生，会严重影响士气。

3）新项目要做的功能往往比较多，所以一次性评估出一两个月的工时和工期，会有很大的风险，比如说：

- 首先是计划赶不上变化，每次需求变动都会调整事先排好的工期；
- 其次是时间太长，开发人员会看不到尽头，士气会逐渐降低，直到崩溃。士气低的直接反映就是质量差。
- 再次是测试团队的介入点，工期排的很紧，我并没有给开发人员预留修之前提测功能的 bug 修复时间。做到后面，我会发现，开发人员一边在做新功能，一边在修之前的 bug。两线作战，疲于应付。

吃过几次亏，决不能再犯同样的错误，比如我最近做的一个新项目，就将其拆分成 3 次迭代，每次迭代做一个完整的功能，包括 App 开发、MobileAPI 开发、测试、修 bug、产品验收，每次迭代 2 周时间。最后再预留出一个迭代（2 周时间）做 buffer，用来处理一些突发事件，比如之前的架构设计得不好需要修改，比如我们对外界的依赖不可用了，比如上线前的一堆准备工作。

这样把一个 2 个月的新项目拆分成 4 次小的迭代，每次迭代都能发布一些新功能给产品经理甚至是大老板看，大家每次迭代的目标都很明确，每次迭代如果都能按时完成任务，士气就会很高涨。这样即使中途加一些新需求，也能消化掉（当然随便加新需求这样不好）。

更多时候，我们所做的项目是有外界依赖的，比如说无线部门往往依赖于公司的底层部门，比如说搜索及产品信息、支付、安全、运维这些部门，尤其是在项目上线之前，对测试环境的依赖性非常大。经常会发生测试环境上午是好的，吃过午饭后就不能使用的现象，所以项目经理在保证自己团队项目进度的同时，还肩负着与其他部门沟通、协作的工作。

10.4.6 迭代风险管理

不夸张地说，无项目不延期。

所以，尽管机关算计，无论是两周的迭代，还是四周的迭代，都会有延期的可能。我接下来讨论的，是如何规避风险、以及遇到了风险如何把风险降到最低。

就以两周的迭代为例子吧。第二周的周三晚上，应该完成所有的功能，称为 Code Complete。如果这个点踩不住，那就有风险了，开发人员往后延期几天，测试也就相应的延期几天，这就导致 App 发版时间也会顺延。

延期一般发生在 MobileAPI。当然不能全都怪从事 MobileAPI 开发人员，因为他们只是

一个中间层，问题出在底层的系统上，包括搜索、产品信息、支付系统、会员体系等等，传统互联网公司的这些系统原型都是为网站服务的，不能直接搬到移动互联网上。

要想规避因此而导致的延期，有 3 种解决方案：

- 让 MobileAPI 的进度提前两周（一个迭代）。只有这样，MobileAPI 才能告诉 App 开发人员，下期迭代能做什么和不能做什么。
- 让 HTML5 网站先行，App 下个迭代后续跟进。考虑到 HTML5 页面开发起来很快，发布起来也很简单，能迅速上线并收集到用户反馈，所以可以让 HTML5 网站先去“趟雷”，以确保 App 开发时少走弯路。
- 如果算来算去，MobileAPI 还是要和 App 一起开发。那么 MobileAPI 一定要在开工前就做好技术调研，需要提供哪些接口，用到底层哪些功能，这些功能是否满足所有需求，这些功能是否都能正常工作。要第一时间知道本次迭代能做多少，否则每走几步就会遇到一个坑，所有人停下来等解决方案；再走几步又遇到一个坑，然后大家又只能停下来等结论。

接下来说第二周的周四，这一天应该做到 bug 日清，如果达不到，说明有风险。对于 bug 重灾区对应的那部分功能，要么是相应的开发人员技术能力不够，要么是需求和交互设计过于复杂了。我们有必要建议产品经理弱化需求，以便该功能能够平稳上线。

做任何需求，我们都要为自己留一条后路，也就是最坏打算，如果未能按期完工，或者质量很差，该如何面对？就算是砍需求，也是需要人工成本去做这个事情的，把代码回滚到最初的状态，所以一定要有个最晚的时间点，过了这个时间点如果还有很严重的问题就要采取断然措施。

最后是第二周的周五，即最后一天，全功能回归测试及发版。这一天，即使是发现了 bug，也不能急着去修复了。这时大家要坐下来一起商量，只修复那些最重要的 bug；对于影响不大的 bug，匆匆忙忙修复反而有可能引起更严重的 bug，这才是风险所在。

要做好带 bug 上线的心理准备，这些 bug 一类是小问题，影响不大，可以延期到下次迭代解决，另一类是大问题但是改动量很大，所以也只能忍痛延期到下次迭代，当作一个 Task 来做。

综上所述，我们会发现，每次迭代的最后三天，是至关重要的，是风险的汇集地，作为管理者，这三天一定要睁大眼睛盯着任何风吹草动，盯着 bug 报表的波动情况。

10.5　迭代中的测试工作

接下来我们说测试，不光是测试人员的日常测试工作，还包括开发人员组织的自测工作。

10.5.1　冒烟测试

真正的冒烟测试，是针对修复了一个 bug 而进行的一系列专门的测试。我接下来说的冒烟测试机制，并不是这个意思，只是为了好听，叫起来朗朗上口，就像前几天我去饭店吃饭，那里有道菜叫枫桥夜泊，其实就是把牛肉块炖一炖，吃的就是那份雅致。

当开发人员开发完成了所有功能，接下来的几天，将主要是测试团队提 bug、开发人员修 bug 的过程。这期间，开发人员是比较空闲的。不要安排开发人员去做新的需求，而是每天找一个时间段（一般一个小时），把他们集中起来，围坐在一张圆桌上，把 App 的所有功能都测一遍，我们称之为“冒烟测试”。

原本我是想帮着测试团队一起做测试的，因为开发人员都比较自信，他们在测试自己的代码时会漫不经心，但是大家都集中测试一个功能时，其他开发人员就会像见到杀父仇人一样，拼命找对方的问题，那种成就感真是妙不可言。

当然了，为了不至于把气氛搞得太紧张，我每次都买些黄飞红或者橘子来作为奖赏。有人提议发现 bug 奖励一碗牛肉面，被我否了，因为发现两个的时候，总不能给一个人买两碗吧，加一份肉倒是可以考虑。

“冒烟测试”主要是解决测试人力不足、覆盖场景不全的问题。在移动互联网公司，开发测试比大约是 6:1，由于很多功能都是集中在最后几天提测，所以测试人员越到后期越紧张，而开发人员介入测试工作，是对产品质量的保证。

起先我只是尝试解决迭代后期开发人员闲置的问题，但几次迭代后我发现这种“冒烟测试”能发现很多 bug，于是便将其纳入到敏捷开发流程中。

后来我们开发团队做过一次代码重构，把 JSONObject 全都换成了 fastJSON，并重写了部分页面的逻辑。但是发版后却发现很多地方显示有问题，最后只好紧急修复、重新发版。事后我们痛定思痛，如何没有在发版前发现这些问题？测试工作固然没有做好，需要另外总结，但是开发团队的“冒烟测试”也没有发现问题，形同虚设，问题又出在哪儿呢？

问题的根结在于，每次冒烟测试的时候，我们都只拿一台测试机安装了最新的开发版本进行测试，并不知道线上版本长得是什么样子。于是我们就改进了冒烟测试的方法，每次都拿两个手机进行比较测试，一台手机上当然是最新的开发版本，另一台手机，有人会拿线上的版本，也有人会拿 iPhone 和 Android 对比着看，总而言之，就是要检查本次迭代是不是改坏了什么地方。我们后来称之为“找不同”——因为我是在游戏厅玩“找不同”时想到的这个办法。

执行“找不同”方案后，我们还发现了 Android 和 iPhone 上很多数据显示不一致的地方，有些是 Android 一直就有的 bug（iPhone 也有一直不对的地方），经过和产品经理确认后，就一并也改正了。

大约在 3 次迭代之后，那时候同时进来很多新的开发人员，他们需要熟悉业务逻辑但是又没有什么文档可供参考学习，当时的办法就是让他们一起参加冒烟测试，每测到一个模块，就请该模块的负责人把业务逻辑讲一下。不光是培养了新人，对于长期做某个模块的开发人员，也是需要了解其他业务模块的，而冒烟测试是最好的培训课，

我清晰地记得，第一次迭代，冒烟测试用了 5 天时间，每天 1 个小时测一个模块，遇到的问题大都是点着点着就崩溃了，到了第 7 次迭代时，每天冒烟测试还是一个小时，但 3 天时间就够了。问题越来越少，App 的稳定性已不再是问题，iPhone 和 Android 的数据展示和业务逻辑也基本一致了。

有人兴许会问，测试工作应该是测试团队要做的啊，开发人员更多的是开发工作。其实不然，我的经验告诉我：

1）测试团队经常面临资源不足的情况，尤其是 Android 和 iPhone 同时发版。

2）开发团队没有那么多的开发工作要做，因为产品团队经常会没有太多的新需求。

3）App 不同于 MobileAPI 开发。MobileAPI 开发可以使用单元测试来保证质量，但是 App 就很难做单元测试了。也就是说，App 前端开发人员并没有做单元测试的 Task，那么这些时间要用来做什么？

4）App 自动化测试的事情就别指望了，那是大公司才愿意投入大量人力去烧钱的事。尤其是互联网行业，需求变动频繁，往往是刚写好一个自动化测试用例，一次迭代后就废弃了。

10.5.2　探索性测试

前面介绍的开发人员自发组织的“冒烟测试”，其实就是探索性测试，只是执行人员是开发团队而不是测试团队罢了。

测试团队在新功能测试结束后，应该做一轮探索性测试。操作方法和前面介绍的“冒烟测试”类似，把所有测试人员组织在一起，逐个模块进行测试，可以每天一小时，分为 3-4 天进行。这样就确保了有 bug 可以及早发现，而不是等到最后一天全功能回归测试时一次性提出几十个 bug。此外，测试团队所有成员都参与，可以保证每个测试人员对每个模块都熟悉，而不是长期只负责自己那个模块。

10.5.3　Monkey 测试

Android 项目每天下班前都要跑 Monkey，然后每天会有专人分析 Crash 日志。Crash 一般分三种：

1）代码逻辑上的空指针，这个比较容易看出来，有助于我们查找 bug。

2）系统问题，比如说不同手机 ROM，问题也不太一样。

3）ANR，这个就没办法了，因为在 A 页面发生的 ANR，并不一定是 A 页面的逻辑导致的，可能在前面很多页面持续积累下来的内存占用过多，就像我的一个兄弟说过的，A 页面可能是压死骆驼的最后一根稻草。

编写 Monkey 脚本，我们要注意几点：

1）要把 App 中的电话拨打按钮都禁止。否则就会因为误点了电话按钮而跳出 App。

2）要确保 Monkey 能进入到 App 的所有页面。

有些模块、有些页面因为层级比较深，所以 Monkey 进入的概率比较小。我们可以订制 Monkey 包，让 Monkey 每次只跑一个模块。

比如说首页由 8 个模块的入口，我们为每个模块创建一个开关，如果今天我跑 Monkey 只想进入第 8 个模块，那么我就把第 8 个模块对应的开关设置为 true，其他 7 个都设置为 false。这样 App 运行时，首页就只有第 8 个模块可以点击进入，其他页面因为开关为 false，

所以都不可以点击。

3）有很多页面需要用户登录后才能进入。为了让这些页面也能跑 Monkey，我们需要每晚跑 Monkey 的包与发版到线上的包略有不同。

最简单的做法是，在程序中新建一个变量 isMoney，以标记当前打的包是否为 Monkey 所准备的。在 Monkey 包中为 true，在正式包中为 false。

那么在登录页，我们把代码改为如下形式：

```
if(isMonkey){
    password=baobao
    userName="qwer";
}else{
    userName=etUserName.getText().toString();
    password=etPassword.getText().toString();
}
```

也就是说，如果是 monkey 包，我把用户名和密码写死在程序中，这样 Monkey 点击登录按钮肯定能够成功，接下来就能进入其他用户相关的页面了。

但是后来我们发现一个问题，这段含有用户名和密码的代码会一起编译到线上的包中，即使做了代码混淆，用户名和密码在反编译后还是能看到的。这是极不安全的。于是便有了解决方案 2：把用户名和密码放在一个文件上，每次读取这个文件。对于跑 Monkey 包的测试机，要把这个文件事先存到 SD 卡上；正式包就不需要这个文件了。

这是我们开发人员一厢情愿想出来的办法，按照这个思路把代码改写完才发现，跑 Monkey 包的测试机上大都没有 SD 卡。

于是我们在碰了一鼻子灰之后，给出了终极解决方案：

在打包脚本上做文章。把这个文件放在项目中。只有 Monkey 包才会在打包时把这个文件包含进来，而正式包不会包括这个文件。这样就彻底解决了安全性问题，只是编写打包脚本时要额外小心，同时，在每次发版前，都要检查一下 apk 包中是否有这个文件。

4）要把设计支付的按钮都禁止，以防止在线上下单而造成的各种纠纷。

10.6 高层对敏捷流程的干预

一般而言，一个敏捷流程是不需要总监级别的高层直接参与的。但是总监应该对敏捷流程适当干预，一方面要把握重构和产品的平衡，以确保一个“度”，另一方面则要提高人力的利用率，可以从开发效率、座位安排、静时这些点入手，从而让团队始终具有高产出。

10.6.1 重构与产品需求的平衡

App 兴起的早期，各大互联网公司都急急忙忙把自己网站的功能搬到了 App 上，而没有

考虑更为长远的事情，久而久之，每开发一个新功能，花的时间很长，质量也不高，App 的代码架构急需重构和优化。

本节讨论什么时候做重构。

在我的项目排期中，是永远不会有重构的任务的。我对产品经理的承诺是，优先把所有产品需求都做完。

我一般会在两次迭代的间隙，来进行重构。因为这时候大家都在确认需求制定计划，最忙的是产品经理和项目经理，开发人员是有时间进行重构而不影响项目进度的。

另外，在迭代过程中，会有需求被砍掉或者弱化的时候，省下来的时间也可以用来做项目重构。实践证明，这样的情况是很多的，而以往，由于没有事先规划好，这些时间是被荒废掉的。

每次重构都要事先规划好：

- ❑ 解决方案
- ❑ 工时
- ❑ 影响范围
- ❑ 测试方案

经常出现重构时没有预估好工时、越做越大、收不了尾的情况——我都见怪不怪了。开发人员总是太自信，以为自己能搞定一切，而不做好规划，殊不知改动越大，风险越大。

好的重构方法是，拆分重构工作，循序渐进，每次做一点。这样既可以尽可能多的完成需求，也可以降低重构的风险。

你可能会说我老了，思想越来越保守了。但你要知道我肩负的责任有多大，对于一个千万级用户的 App 而言，稍有闪失都会对公司的生意造成重大损失。

10.6.2　提高效率，拒绝 6×12

我曾经经历过 6 周时间的 6×12 工作制，包括 Android 和 iOS 两个项目的 Scrum Master，带领着团队艰难地熬过这段时间。

说是熬，一点也不夸张。开始时三周，大家的精神状态还好。三周之后，就发现团队和之前不一样了，主要表现为：

- ❑ 战斗力急剧下降。
- ❑ 质量下降，bug 激增。
- ❑ 脾气开始变得暴躁，容易发生冲突。
- ❑ 每天就是在耗时间。周六基本就是中午来吃个饭，然后四点多就下班了。
- ❑ 上班越来越晚，午休时间变长，晚餐后还要散步半个小时。

综合而言，表面上看起来是 6×12，但实际上只有 5×8+4，也就是说，每天实际工作 8 小时，再加上周六的 4 个小时。

另一个只有项目经理才能感觉到的问题是，随着开发人员每天的工作时间延长到 12 小

时，项目经理的工作时间会变得更长，每天甚至会超过 12 小时，因为有更多的项目上的事情需要去沟通解决。我记得项目到了后期，我基本上是 7×12 的节奏了。

我还发现，违背项目管理流程的是，6×12 相当于没有了项目缓冲时间，也就是说如果 6×12 还是发现有事情做不完，那么就真的做不完了，因为不会让团队周日也过来加班而不休息一天。

6 周后得到的经验是，6×12 适合于搞突击，但时间应控制在 3 周以内。想提高开发人员的效率，还要想别的办法。

我一向是反对硬性要求开发人员加班的。研发人员不同于其他工种，他们写了一天代码，需要很好的休息，才能保证第二天继续高效的工作。偶尔加班 1 ~ 2 小时，因为程序员大多吃青春这口饭，所以可以凭借年轻缓过来。但是长期的加班就不同了，只会使得代码质量下降，bug 变多。

我曾经计算过每天上班 8 小时（朝 9 晚 6，午饭 1 小时不计入）的实际利用率。以下时间是要扣除的：

- 上班整理工位、吃早饭时间。
- WC 时间。
- 饭后散步时间。
- 午休时间。
- QQ 闲聊时间。
- 淘宝购物时间。
- 各种被打扰时间，比如线上投诉的跟踪解决、各种紧急会议、其他部门咨询，等等。

其中前 4 项是不能省的，每项约半小时，那么每天就有 2 小时不在工作，每天工作 6 个小时是极限了，但如果算上 QQ 闲聊和淘宝购物时间，那就只剩下 4 个小时不到了。

所以，作为团队负责人或项目经理应注意以下几点：

- 要减少团队在 QQ 闲聊和淘宝购物上花费的时间，充分利用好这实打实的 6 个小时。我的做法是，只要事情提前做完了，剩下的时间开发人员干什么都可以。当然我更鼓励员工闲下来去学校新技术，为自己增值。
- 另一方面，还是要控制每个 Task 的工时，精细到 0.5 天。拒绝那种有很大水分的 Task 评估，这就是项目经理的职责了。开发人员往往喜欢给自己留一些 buffer，其实半天时间就够了。
- 减少被打扰时间。我在微软时，所在的团队有一项很好的制度，每周三下午是 Quiet Time，也就是静时。这段时间不和外界任何人沟通，专心做自己的事情，效率是非常高的。

10.6.3 无线部门的座位安排

一种排摆工位的办法如图 10-4 所示（空白处的表示过道）。

Android开发4	Android开发3	Android开发2	Android开发1		产品经理1	产品经理3
MobileAPI开发4	MobileAPI开发3	MobileAPI开发2	MobileAPI开发1		产品经理2	产品经理4
iOS开发4	iOS开发3	iOS开发2	iOS开发1		设计人员1	设计人员3
H5测试2	H5测试1	App测试2	App测试1		设计人员2	设计人员4
H5开发8	H5开发6	H5开发4	H5开发2		会议室1	
运营人员4	运营人员3	运营人员2	运营人员1			

图 10-4　无线部门的座位图 1

这种座位的排列，对于刚刚成型规模不大的无线部门比较有利，主要体现为：

- App 开发人员，无论是 iOS 还是 Android，都可以快速与 MobileAPI 开发人员进行沟通，联调。因为后者坐在中间位置。
- App 开发人员、MobileAPI 开发人员、测试人员可以快速找到产品经理和设计人员。
- 测试人员可以快速地找到开发人员，尤其是 iOS 开发人员。

随着人员的极速扩充，以上座位图不能满足需求，一种新的方案如图 10-5 所示。

MobileAPI开发8	MobileAPI开发6	MobileAPI开发4	MobileAPI开发2		MobileAPI测试人员2	自动化测试人员2	
MobileAPI开发7	MobileAPI开发5	MobileAPI开发3	MobileAPI开发1		MobileAPI测试人员1	自动化测试人员1	
Android开发8	Android开发6	Android开发4	Android开发2		App测试人员1	App测试人员3	App测试人员5
Android开发7	Android开发5	Android开发3	Android开发1		App测试人员2	App测试人员4	App测试人员6
iOS开发8	iOS开发6	iOS开发4	iOS开发2		产品经理1	产品经理3	产品经理5
iOS开发7	iOS开发5	iOS开发3	iOS开发1		产品经理2	产品经理4	产品经理6
H5开发7	H5开发5	H5开发3	H5开发1		设计人员1	设计人员3	设计人员5
H5开发8	H5开发6	H5开发4	H5开发2		设计人员2	设计人员4	设计人员6
		运营人员4	运营人员2		运营人员1	运营人员3	

图 10-5　无线部门的座位图 2

这种座位的排列，对于“大兵团”作战非常有利，主要体现为：

- 以 Android 团队为例，他们背靠背坐成两排，有技术上的问题找左右或者转个身就能最快寻求到帮助。
- 即使测试人员坐到过道的另一边，仍能快速地找到开发人员和产品经理。
- 开发、测试、产品经理、设计人员之间的沟通仍然很便捷。
- 每次增加新人，就向两边扩充，比如新来一个 Android 开发人员，就让他坐在 Android 开发 7 的左边。

每个团队招多少人是有预算的，每年年初都定好指标了，所以每年需要调整一下座位。HR 和行政人员往往以为招的都是你无线中心的人，坐在哪里都是一样的，所以找个角落随便给安排个座位就算完成任务了，于是你会看到一个团队大部分人坐在一起，而还有 2 个人分别坐在天南地北的两个角落里，其实这是有问题的，至少说明了在无线互联网飞速发展的今天，全民都已经学会连去厕所都会带上手机把玩自己心爱的 App 的同时，HR 却在工作上

未能与时俱进，没搞清楚自己所在公司的无线部门怎么排摆座位才能达到工作效率最高。

如果团队继续扩充呢？这就不是简单的排摆座位就能解决的了。我们知道，任何一个精干的团队，超过 8 人都是有问题的。首先 Team Leader 管理多于 8 人的团队就会捉襟见肘。所以要进行拆分，目前看起来，根据业务线进行拆分，是个不错的办法。每条业务线都有自己的产品经理、设计人员、Android 开发、iOS 开发、MobileAPI 开发、HTML5 开发、测试人员、运营人员。于是每条业务线的座位排摆方式就又回到了第一张图那样——可以认为这是一个由量变引起质变的过程。

10.6.4 静时

软件公司的很多理念，在互联网行业是行不通的，比如说软件开发流程就不一样，前者是敏捷流程而后者是瀑布流程。因为互联网永远是快节奏，所以会不按规则出牌，一切以快速上线为最高优先级，为此而牺牲了流程，所以你会看到，互联网公司，永远是乱哄哄的，没有一方“静”土。一方面，大家都在忙着处理快速上线后的各种问题，于是讨论的时间会多于静下来工作的时间；另一方面，大家都在为下一次迭代上线赶进度，却发现需求不到位、设计稿不到位、后台数据不到位，为此又不得不一轮又一轮进行讨论，等都讨论完，却又发现留给自己的工作时间已经不多了，所以加班是不可避免的。

抽丝剥茧，你会发现，开发人员的时间被严重碎片化了。任何人都可能来打扰他们。比如说突然发现的线上 bug，比如说客人投诉、比如说产品经理临时修改需求、比如说领导视察工作、各种谈心。

要想办法把这些碎片化的时间汇集在一起，开发人员的效率就能大幅提高了。这比多招几个开发人员、在架构和代码上进行优化要更管用。

为此，我们引入“静时”的概念。

静时（Quiet Time）是软件公司的术语，就是说，每周有几个下午，开发人员关掉所有通讯方式、不再进行沟通或者被沟通，全身心的投入编程工作，不被任何人任何事打扰。我在微软切身经历过这种机制，每周三下午的效率是最高的。整个办公区域会安静下来，当然，副作用是容易犯困，开始几次还真不适应。

软件公司的任何理念，想在互联网公司着陆，都是需要修正的，否则就会因为水土不服而达不到效果。我记得一开始施行静时的时候，App 团队倒是安静下来了，专心去做项目赶进度修 bug 了，但是其他团队就乱套了，其中最突出的问题是线上 bug 没人去查了，而这些问题往往很急，需要立刻解决，越快越好。

于是我们为每次静时指定一个值班人员，由他在这段时间作为外界的统一接口，处理这些乱七八糟的线上问题。开始由各团队的 Leader 担当值班人员，慢慢地就由团队成员轮流担任。这就相当于过春节放长假大家都回家休息了，但一定要有值班人员，能够处理线上各种紧急状况。

在 App 团队彻底安静下来之后，我们就发现，MobileAPI 团队也可以静下来啊，产品经

理团队也可以静下来啊，于是各个团度都设定了自己的静时。实施后我就发现，各个团队的静时设置为同一天，只能保证那一天效率很高，其他时间还是会很乱很低效。把各个团队的静时设置为一周的不同时间，反而能达到效率的最大化，因为每天都有团队要安静下来，只能投入 1 个人参与讨论，这就间接减少了其他团队想沟通的愿望。

静时是为了解决频繁沟通、无效沟通的问题。但是搞过火了会导致信息不同步，从而引发更严重的问题。为此，每天静时后，还是要留出半个小时，处理一下其他团队的诉求。另一方面，要提前协调好和其他团队协同工作的时间，要让其他团队知道，在什么时候可以来找你商量事情。

10.7　本章小结

本章介绍了敏捷开发中的项目管理。管理学分两种：团队管理和项目管理。团队管理我推崇弱管理，别给团队成员加诸太多的条条框框，给他们一个主题，让他们自由发挥，往往能得到惊喜。别让他们去做太多不擅长的事情，这种事情让部门秘书去统一去做就好了。而项目管理我执行强管理，我要清楚知道每个人每天的进度，以避免劳动力的荒废。这时候需要一个强力的项目经理来推进，以确保每次迭代能最大程度的完成需求，及时汇报剩余工时用于重构工作。

评价一个团队的好坏，不是看技术能力有多强，而是能否按时交货。为此，要想办法提高工作效率。本章涉及的各种技巧，都是我在实际项目管理中的经验总结。

第 11 章 Chapter 11

日常工作中的问题解决

自从我踏入 App 这个行业，就过上了在刀尖上舔血的日子，每天在战战兢兢中度过，线上一旦有什么风吹草动，比如逻辑错误或页面崩溃，都要立刻放下手中的事情，组织人力去查明原因。能查明原因并快速解决还好办，找不出原因是最头痛的，或找出了原因却无计可施则是更悲催的事情，因为会影响公司生意。

我三十岁前在微软，每天过着晚上六点下班就伙同张三李四王二麻子满世界吃喝玩乐的生活，经常周一上班没精神是因为周日唱 K 把嗓子喊哑了。三十岁后投身互联网后，每天下班都是九、十点钟，经常是办公室最后只剩下我一个人在那里分析线上 Crash，或者带着团队加班赶进度，然后周末倒在家里睡一天。我的青春就是以三十岁为分水岭，前松后紧的度过。我只希望老了以后，回忆起这段充实、刺激的人生经历，不会因为自己的碌碌无为而遗憾。

本章我要介绍的内容，就是在工作中逼出来的解决方案。

11.1 使用二分法排查问题

二分法是编程课中讲解递归函数时提到的概念，但其实这个方法在现实中往往用于排查问题。

记得有一次 Android 发版，测试人员发现，收银台突然就不能支付了，一点击支付按钮就崩溃。负责该模块的开发人员东瞅瞅西看看找不出崩溃的原因，最后一口咬定是后台 MobileAPI 有问题。

这件事上升到我这个层面，我当时就觉得很奇怪，首先，线上的版本是没有问题的，iPhone 上也是好的，而且，据测试人员反映，Android 的测试包前几天还是好的。那么基本

可以断定：

1）与 MobileAPI 无关，肯定是这次迭代改出问题来的。

2）导致崩溃的改动，肯定就是这几天的代码签入导致的。

于是我使用二分法来查找问题。我们要查找的是，导致 App 支付崩溃的那次代码签入点。换句话说，找到 App 最后一次不崩溃的代码签入点。

首先看一下每日自动打包（DailyBuild）的服务器上备份的历史版本，如图 11-1 所示：

```
C:\ProjectForAntBuild
        |——5.4.2
                |——2015.06.01.001
                |——2015.06.02.002
                |——2015.06.03.003
                |——2015.06.04.001
                ……中间省略若干次打包版本
                |——2015.06.30.021
```

图 11-1 打包服务器上的历史版本

也就是说，6 月 1 号开始开发，6 月 30 号最后一次提交代码。我们先以天为单位，找到崩溃发生的那个临界点。

1）我们先检查 6 月 1 号的包是好的还是坏的。如果 6 月 1 号的包是坏的，那么就是 6 月 1 号的某次提交导致了崩溃，我们直接在 6 月 1 号的提交历史中进行二分法排查。如果 6 月 1 号的包是好的，那就是 1 到 30 号之间的某天的包有问题。我们使用二分法，看一下 15 号这个包是好的还是坏的，如图 11-2 所示：

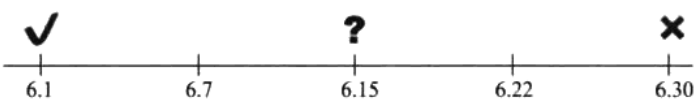


图 11-2 使用二分法查找错误提交点

接下来会有两种情况：如果 15 号的包是好的，那就是说 15 到 30 号之间某天的包有问题，如图 11-3 所示；如果 15 号的包是坏的，那就是说 1 到 15 号之间某天的包有问题，如图 11-4 所示。

图 11-3 使用二分法查找错误提交点

图 11-4 使用二分法查找错误提交点

选择 15 号作为第一次二分法的时间点，帮助我们缩小了排查范围。以此类推，下一次二分法的点是 7 号或者 22 号。以此类推，逐步缩小排查范围。对于时间长度为 30 天而言，这么找 5 次就能找到出问题的那一天（2 的 5 次方最接近 30）。

2）在出问题的那一天，我们继续使用二分法，找出问题的那一次提交。这次二分法不再是以天为单位，而是以每次提交为单位。如果当天有 100 次提交的话，大约找 7 次就够了（2 的 7 次方最接近 100）。

虽然上述这种做法土了点，每次都是把那次签入相对应的整个 App 代码都签出，然后打包安装到手机上验证是否会有崩溃。但越是土办法越有效，我花了 1 个小时，就把问题定位在昨天下午的一次代码签入，开发人员误把一个 if 语句直接返回 true 而不走下面的业务逻辑，没有给一个变量赋值，所以点击按钮的时候因为那个变量为空而崩溃。

后来这个方法就推而广之了。有一次发版后，大量用户打电话反馈说不能登录——对于一个电商平台而言，不能登录意味着不能下单，没有生意做，这是绝对不允许的。

但奇怪的是，我们开发人员无论如何也不能复现问题，就这样猜着查了一下午原因，始终不得要领。直到傍晚下班前客服妹子的一个电话打了进来。预知后事如何，且听下文分解。

11.2　找到能稳定重现问题的人

上节说到，我们发版后接到大量用户反馈说不能登录，但是我们在北京却不能复现问题，一筹莫展。

我们通过分析发现，这些反馈用户来自全国各地，但就是没有北京的，所以我们需要找一个能稳定重新这个问题的人，来协助我们排查问题。

就在我们急得像热锅上的蚂蚁时，有一个身在外地的客服妹子打电话给我们反馈同样的问题。当时我们激动死了，因为这个客服人员，就是我们要找的人，她具备两个特质：

1）能够稳定重现问题。

2）愿意花费大量时间不厌其烦的帮我们测试。

于是我们就基于这一个月来每天晚上 9 点通过 DailyBuild 机制生成的安装包，使用上一节介绍的二分法来排查问题。就这样一步步缩小范围，直到我们发现是某一天开发人员的一次错误导致的。这时已经是第二天晚上 9 点了。客服妹子陪我们整整一天进行枯燥的测试工作，装了卸，卸了装，反反复复就是验证登录那个功能。

查明原因并修复问题就要紧急发版了，但是只有那一个客服妹子测试过，只能说看似修复了这个 bug，对于其他用户，升级后是否就可以登录了，还不得而知。于是我就逐个给之前投诉的客人打电话，请他们帮忙，安装我发给他们的测试版，看是否可以正常登录了。因为当时已经晚上 9 点多了，所以大部分客人要么是已经休息了，怪我半夜吵醒他们，要么是不接电话，还有怀疑我是骗子的。总之打到晚上 10 点，二十多人中只有 2 个人愿意帮我们做测试，但聊胜于无，多一个人测试成功，就多一分上线信心。

经过这件事，我就总结出两点：

1）二分法再好用，找不到能帮助我们复现问题的人也是白搭。客服部门是比较好的人选，她们可以在接到客人投诉后，亲自用自己的手机点一点，看看问题是否真的存在。而且，最重要的是，她们在复现后，能帮我们长时间进行测试，因为我们是同一个公司，这属于部门之间协作，都是为公司做事，责无旁贷。另一个可以建立协作关系的部门是遍布全国各地的销售部门和分公司，可以帮我们测试全国各地的网络情况。

2）对于互联网公司，一定要建立公司的忠实用户群。这些用户可以在新版本发布前，帮我们进行测试。他们遍布全国各地，有各种不同的需求，从而试用的业务场景也不尽相同。要适当给他们一些奖励，比如 VIP 用户或者红包代金券什么的。不要以为全国人民都像开发人员似的，每天忙得焦头烂额，哪还有时间去帮别人去做小白鼠，其实有闲人和好心人还是很多的。

一般来说，能打电话来投诉的用户，都是愿意花时间帮我们进行测试的用户。

11.3 小流量包

Android 有一个比 iOS 好的地方，就是可以随时发版，发现问题后立刻修复立刻发新版。至少很多书上都是这么说的。

但我从事无线领域这几年的经验告诉我，实际情况并非如此。频繁发版会导致用户要频繁更新，他们会认为这家公司是不是要倒闭了？怎么一周内接二连三发 hotfix 版本？所以每次发现线上 bug，我们都会很谨慎的评估，是否一定要发 hotfix 版本。不同行业发 hotfix 版本的衡量标准是不一样的：

- 电商类公司，他们会很在乎订单数和订单转化率，所以一定要确保支付主流程能走通，同时，对于一切影响生意的 UI 展示问题都很在意，比如票券的有效期会误导用户，比如酒店的好评率如果不展示将直接影响订单的转化率。
- 社交类公司，他们会很在乎各个页面的广告是否能正确投放，因为这类公司就是靠收取其他公司的广告费来盈利的。另外，他们比较关心 PV 和 UV，因为不同页面上的广告费用是不一样的，PV 和 UV 高的页面，广告费用也高，比如首页。
- 推送，更是一个重中之重的功能。尤其是个性化推送，能在公司和用户之间建立很好的互动。如果推送功能坏了，是必须要紧急发版的。
- 地图定位，对所有公司都很重要，所以使用一款好的 SDK 至关重要，我们最关心的是地图 SDK 的稳定性和性能，还有准确性。我一天到晚接到全国各地销售部门的投诉，说某某定位错了，导致客人找不到具体地方，直接影响生意。

每次紧急发版都会把所有 200 多个渠道包全都打一遍，就算是自动化批量打包，每个包需要 5 分钟，也要等服务器运行将近一天时间[⊖]。然后由推广人员执行以下操作：

⊖ 有一种超快速打渠道包的机制，请参见本书 9.9.3 节。

- 一部分 apk 包手动上传到各大市场，有些是立即生效，有些则需要 Android 市场那边审核，如果是周末对方不上班，还要打电话请人家帮忙加速审核。
- 一部分 apk 分发给 Android 市场的市场人员，由他们帮忙上传。
- 一部分 apk 放到公司的服务器，然后更新所有 HTML5 短链的地址。

每次 Hotfix 发版，都会这么折腾一遍，所有涉及的人都苦不堪言。经历了几次 Hotfix 后，我就开始在想如何提前发现 App 的这些严重问题。

我们先尝试使用 Google Play，每次发版前一两天都提前发布到 Google Play 的灰度环境，设置为 50%，也就是说每两个用户就有一个用户能使用到新版本。我们原本以为这样就能收到用户的反馈了，但是我们试了几次后，发现这种方法不可行，因为在中国，Google Play 的用户很少，每天 100 多用户，所以收不到任何反馈，也看不到新版本发生 Crash 发送到后台服务器的异常日志。

既然 Google Play 行不通，因为用户少，那我们就在想，能否在用户量比较大的渠道上提前几天发布我们的测试版本？

我们发现，网站首页上的主渠道，用户量比较大，每天大约有 1000 的新用户激活，所以我们尝试将主渠道上的包提前一周替换为测试版本，我们称这个测试版本为小流量包。

小流量包的版本号仍然是当前线上的版本号。比如说，线上版本是 6.0，过几天要发新版本 6.1，在这期间我要在主渠道发布一个小流量包，版本号只能是 6.0，而不能是 6.1，否则第二天你就会发现各个渠道上的 App 都升级为 6.1 版本了，渠道商之间的竞争很激烈，他们会尽量保证每个 App 都是最新的版本，从而获得更多的下载量。但这样一来，小流量包就失去了原先的意义，相当于提前发版上线了，所以版本号一定不能变，仍然是 6.0，就不会被抓包了。

那么如何区分线上版本和小流量包呢？比如说激活数、下单数、转化率，甚至是线上 Crash 数据，因为版本号一样，都会混在一起。我的经验是申请一个新的渠道号，比如我今天要发布小流量包，那么我就找财务申请 20140802 这个新渠道，这样在后台就能看到渠道号为 20140802 的 Crash 信息了。到时候只要在计算每日激活量时，把这个渠道产生的激活划归到主渠道就是了。

小流量包的版本号不变，使得只有新用户才能下载到小流量包，老版本的用户，因为版本号一致，所以不会进行更新。这样就避免了频繁升级 App 版本对用户造成的麻烦。

11.4　建立全国范围的测试群

我们在本章 11.1 节和 11.2 节介绍了如何通过二分法排查线上 bug，以及如何请客服人员帮我们一起排查问题。像这类问题，只要能找到能稳定复现的用户，能帮助我们不停地进行测试，就肯定能解决线上的各种疑难问题。

这件事过后，我就在想，如何能提前发现类似的网络问题。像上面遇到的这件事，在

开发期间，在北京研发总部，无论是开发人员还是测试人员，都没有遇到过，但只要一到外地，就有可能发生，这是我们研发团队所面临的一个难题。

后来我在公司里面四处转悠的时候，我就发现销售部门可以帮我们做测试啊。要知道只要公司大一些，全国各地都会有销售办事处的。

有人会说，你说得轻松，人家也有要忙的事情，哪有空理你啊！其实，测试工作如果做好了，反过来可以帮销售部门争取到更多的客户，对公司也是有益处的，只是之前没有人意识到这一点而已。

于是我写了一封邮件给全国 30 多个省会的销售负责人，大抵是说，请大家配合提供各个地区的有 Android 手机可以配合测试工作的部门助理的联系方式，但正和事先料想的一样，回复者寥寥无几。没关系，我开始改变策略，一封封地发这个邮件，而且全都抄送给整个销售部门的总负责人。刚发完第二封，那个总负责人坐不住了，估计是被我骚扰烦了，他回邮件说这邮件你小子别再发了，我来帮你催。

就这样在一天之内要到了全国 30 多个省会的有 Android 手机可以配合测试工作的部门助理的联系方式，把她们加到了一个新创建的全国销售 QQ 群中，每次发版前一周都会给群里的每个人发一个测试包，测试在 WiFi 和 2G、3G、4G 网络环境下主流程是否能走通。我记得每次干这事的时候，都要一个小时同时和 30 多个人进行 QQ 聊天，我打字又不快，经常脸憋得通红，屏幕上的 30 多个 QQ 窗口全都在闪烁而我又回应不过来。

后来我老板听说这事，专门跑到我屏幕前看我创建的这个全国销售 QQ 群，他说你是工作泡妞两不误啊，这群里怎么这么多 90 后美女？我说 TMD 还不是为了你的生意，不然我一个做技术的，这每天干的事哪件和技术有关啊？

这个 QQ 群创建后，还有另一个意想不到的效果，就是销售部门的同事发现 App 的问题后，直接就在 QQ 群里说了，我可以第一时间收到反馈并立即组织开发人员查找原因，而这在过去，往往要中转好几个人，才能把问题和邮件转到我这里。

11.5 如何与用户沟通

用户投诉非同小可，我们要把每天的用户投诉作为一个长期的工作来抓。

每个打电话来投诉的用户，背后都有 99 个发现了同样问题但是却懒得打电话的用户，所以如果连这个用户都不理不睬，那么我们的 App 就真的没救了。

我作为 App 用户，也经常会打电话投诉或者反映问题，我希望问题很快解决，即使不能百分百解决，我希望有人愿意为此负责，而不是推卸责任。

所以，我在代表公司处理用户投诉时，我会这么做：

- 使用公司座机打过去，这样能表明自己不是骗子。不要使用手机。
- 首先表明自己的身份。我的经验是，当用户听到你是技术人员时，会比较乐于沟通，因为他会认为他的投诉得到了足够的重视，已经一层层传达到了公司的技术部门。

- 详细问清楚问题发生的场景，包括手机型号、网络是 2G、3G、4G 还是 WiFi、所在城市、App 版本号。
- 留下用户的 QQ 号或者微信号，这是为了方便以后能继续沟通。如果有可能，可以把这个用户加入到公司的活跃用户群。既然用户能打电话来跟我们讲问题，那就可以认为这些用户是我们潜在的忠实用户了。
- 如果需要用户花时间来配合我们的测试工作或者重现问题，最好能给用户一些实惠，比如升级为 VIP 用户，或者充 50 元话费等。

与各种各样用户沟通多了，发现用户的问题基本分为以下几种：

1）用户操作行为错误。这其实应该怪产品经理设计了屎一样的产品，让用户抓狂，才会点错。我听说美团的成功，就在于给商户使用的后台非常简单，所以能抢到更多的商户。交互逻辑非常复杂的功能我做过很多，上线后就是没人用。由此而验证了一条真理：简单，才是美。

2）业务逻辑有 bug，甚至导致崩溃。这主要是 App 开发人员的问题，也跟测试人员没有覆盖足够的测试场景有关系。目前，大多数公司遇到这样的问题，如果很严重，只能紧急修复、紧急发版；如果发新版成本很高，就只能忍到下次迭代发版了。想快速解决这类问题，Android 可以走插件化编程的路。对于 iOS，可以考虑 Lua 脚本编程技术。

3）GPS 定位不准。这是我遇到的投诉最多问题。这是要最优先解决的问题，这个数据不准，尤其是定位错了城市，或者把酒店定位到海里去了，都会闹笑话。很多时候，这是因为 Android 使用了百度地图而 iOS 使用了高德地图的原因，他们的坐标值不一样，所以在地图上的位置会不一样。

4）App 版本低。低版本有 bug，我们发现后在新版本修复了。用户升级到最新版本后，就没有问题了。

5）问题不能复现。如果不能复现，或者说时好时坏，那多半是 MobileAPI 返回了脏数据的原因。

6）客服记录问题与客人投诉问题完全不符。因为客服只管记录，对 App 并不熟悉，所以经常会以讹传讹，把客人投诉订单列表页的问题，反馈为产品列表页的问题。总之驴唇不对马嘴的事情很多，所以开发人员如果想知道真实情况，一定要打电话亲自询问客人原委。

鉴于每天都有大量的用户投诉，我们在与用户电话沟通后，要找一个地方备案，Excel 也好，Wiki 也好，自己做的系统也好，总之：

1）成功解决的，要写下来解决方案，以便于以后有类似问题，可以不用排查，直接答复用户。我们称之为 Trouble Shooting。

2）不能复现、成为无头公案的，如果不严重，就当作优先级不高的 bug 来处理，要记下来用户联系方式以及问题的来龙去脉，时刻保持警惕。

保证产品的质量不光是靠发版前的测试工作，还包括产品上线后的线上问题的跟踪和处理。

与用户沟通，不同于在公司里做项目，需要另一套沟通的技巧。要和颜悦色、要循循善诱、要不卑不亢、尽可能多的从用户那里获取信息，尽最大程度地请用户帮我们复现问题。

我们开发人员，遇到问题不要一上来就想看 log，用户手机上是没有 log 的，就算记录了 log，也不会拿给我们看的，要从多个角度综合分析问题，比如说追踪发生问题的时间点，我们可以沿着这个方向去后台查找 MobileAPI 日志、检查 Crash 信息。有关这方面排查问题的方法论，我们下一节再介绍。

11.6 日志与 App 性能

日志这玩意儿非常强大，关键看你会不会用。

在 Android 日常开发中，我会输出每次调用 MobileAPI 时的接口地址、返回的 JSON 字符串。但是这还远远不够。

对于一次完整的请求，我们需要记录以下信息：

1）发起请求的时间点，注意这个时间不是点击请求按钮的时间点，而是点击请求后调用 HttpRequest 执行一次 MobileAPI 网络请求的那个时间点。

2）接收到 MobileAPI 网络请求的响应时间。注意这个时间点，是接收到 JSON 字符串的时间。

3）从客户端接收到 JSON 字符串到页面生成的时间。这主要用于测试列表页的生成时间，用于优化列表页的加载性能。

将 1）和 2）这两个时间点相减，得到调用一次网络接口的时间，这期间包括服务器处理该请求的时间，以及来回传输数据的时间。我们请 MobileAPI 将每次响应的时间记录在 HttpResponse 响应头中，返回给客户端，就可以计算出每次请求中到底哪一段最耗费时间。

以上就是客户端网络性能的检测方案。我们将这些信息作为日志记录到 SD 卡上。每天晚上跑 Monkey，基本上每个页面都会走好几遍，那么每个 MobileAPI 接口的性能数据就都能得到了。

每天只在 WiFi 网络环境下跑 Monkey 测试，是得不到真实的数据的。跑 Monkey 测试时一定要使用 2G、3G 和 4G，虽然多花点钱，但是能模拟出大部分用户的真实性能数据，其中哪个页面是痛点就一目了然了。

另一种采集性能数据的做法就是把每次 MobileAPI 的性能数据放在内存中，然后每隔半分钟就发送到服务器，由服务器进行分析。这种解决方案的缺点就是一旦没有网络，MobileAPI 网络请求就会在客户端产生积压，所以要对积压过久的网络请求及时清理。

11.7 从新人入职作业入手

“不识庐山真面目，只缘身在此山中。”这两句诗讲的是，当局者迷，局外人往往看得更清楚。

对于从事 App 研发的人来说，包括开发人员、测试人员、设计师和产品经理，每天的工作就是丰富完善产品，等做到一定程度，就会有瓶颈，再难突破。这时就需要局外人来“搅搅局”了。

最好的“搅局者”是新入职的员工，他们刚到公司，身上还带有上一家公司的痕迹，所以能提出比较中肯的问题。这时候，请他们试用一下我们的 App，作为新人入职培训的课后作业布置下去，能收集到各种深刻的意见。

比新员工更有效果的是实习生，他们身上有一股初出茅庐的锐气，不像在职场上混了一两年的人那样有诸多顾虑。我曾经收到过一封从 CEO 那里直接转过来的邮件，是一位实习生使用 App 的意见反馈，其中虽然有些个人色彩夹杂其中，但有些意见是一针见血的，逼着我立刻就要组织人力去解决。之后的一段日子，每天都会有实习生使用心得的邮件转到我这边，他们这些人会拿着手机开着 2G、3G 和 4G 在北京的大街小巷使用我们的 App，于是各种网络问题、各种用户体验问题（我们称之为反人类设计）纷涌而至。

请实习生给 App 提意见的另一个好处是，他们的年龄都是在 23 ~ 25 这个年龄段，精力旺盛，对新生事物接受快，与 App 这种新兴事物的适用人群正好匹配。四五十岁的大叔是没时间也没精力给出太多太好的建议的，他们可能已经被生活折磨得身心俱疲了。

于是，我们可以在新人入职培训中加入本公司的 App 产品介绍这堂课，并为每个新员工布置一个作业，使用 App 一周，把使用心得和意见反馈以邮件的形式发出来。另一方面，要求无线部门负责人要回复每一封意见反馈的邮件，逐条解答各个问题，并对确认的问题给出排期解决。打开意见入口，后面一定要有人收尾，否则就是形象工程。

充分利用好这个通道，这比每天去 AppleStore 和 Android 各大市场看用户反馈要好得多。因为你可以直接找到发现问题的新员工获取更详细的信息，如果是 bug，甚至可以要到他的手机进行调试或者看手机上存储的日志。

11.8　本章小结

本章介绍了 App 的日常管理工作中的各种技巧，都是实际工作中点点滴滴的回忆。

本章写给最懂我的人看。

致那些和我一起加班熬夜奋斗过的兄弟们。

第 12 章 Chapter 12

无线团队的组建和管理

团队管理者决定了这只团队的高度。

想要成为 CTO，只会无线那些技术是不够的，还需要补习大数据和搜索、数据库等技术。

我希望我的团队像李云龙的独立团那样，平常一个个看上去都不起眼，但是打起仗来嗷嗷叫。为了达到这个目标，需要隔三差五地激励士气，让团队的每个成员都挑战自己的极限，尽早地完成技术上的飞跃；需要组织各种技术培训，建立一个良好的技术氛围；需要经常一对一沟通，对症下药，才能让每个人都产生团队归属感；此外，摒弃公司的陈规陋习，甩掉工作中阻碍团队发展的一切束缚，轻装上阵，这样每个人才会有干劲儿。

所有这一切，从招人开始。

12.1 从面试谈起

一个团队的整体风貌，和团队负责人有很大关系。如果团队负责人比较外向，那么他的团队也必然很火爆；如果团队负责人是内向型，那么他的团队也会很闷，日常工作中基本没什么声音。闹有闹的打法，静有静的风格，没有对错之分。

一个人性格是外向还是内向，面试时就能看出来。

12.1.1 如今是卖方市场

“面试的时候看人的短处，用人的时候看人的长处。”这是我曾经的一位老板跟我讲的，经过我这些年的实践，感觉并不全对。对于谷歌、微软、BAT 这类公司，每天有成千上万人挤破头颅要进去，所以他们永远不缺人，可以在一流人才中，慢慢找候选人的短板。

但是对于二线公司，情况就不容乐观了。众所周知，移动互联网迅速爆发，人才缺口很大，基本上所有的互联网公司都缺人。一流人才，基本见不到，都去 BAT 了，只能从二流人才和三流人才中下手，同时还要手快，稍微慢一拍就被其他公司抢走了，所以对于二线公司，要适当降低标准，一个强力的 Team Leader，外加一些能干活的人就行了。

人一旦招进来，接下来就要把他培养成一流人才，让他具备进入 BAT 的水平。于是我们要招那些有潜力有灵气的但是经验欠缺或者背景不好的开发人员，太笨的、太懒的、慢条斯理的都不行，如果要组建一支嗷嗷叫的团队，切记要守好这最后的底线。

作为部门主管，一旦你发现候选人不错，就要留个心眼了，无论是电话还是 QQ 还是微信，尽快与候选人后续建立长期联系。一言以蔽之，对于 App 开发人员，现在是卖方市场，我们招人时要改变以往高高在上的姿态，否则，就招不到人。

12.1.2 名校论不适用无线开发

有些公司要求招人必须是名校，尤其是研发部门，我觉得是不妥的。

我带过的团队成员，什么学校的都有。水平高者，往往来自那些名不见经传的学校，甚至是二本三本。我想，这大概是外界对研发二字的误解吧。一提起研发，所有外行人都会认为这是件很高深的工作，必须是 211 或者 985 高校的博士教授做的事情对于学术也许如此，但是对于软件研发其实不然，类似于搜索之类涉及复杂算法的软件行业，固然需要较高学历良好背景的人去研究，但是对于 App 应用类软件而言，每天的开发工作大都是重复性画 UI 和调用 MobileAPI 获取数据，就如同流水线工人那样做事，所以真的不需要名校出身。

12.1.3 如何搞到更多的简历

这年头，想要优先拿到简历，必须和 HR 搞好关系，不动点脑筋是不行的。可以把公司 HR 的妹子泡到手做老婆。我自酌没有这样的条件，可是我会做饼干蛋糕面包千层酥这样的甜点啊，于是亲手做了一份蛋挞和提拉米苏给 HR 的美女们送了过去，可想而知，接下来就陆陆续续有简历到我手里了。

再后来，简历又少了，因为不能总优先照顾我啊。于是我就着急了，我让 HR 把我的邮箱加到招聘组中，只要有人投开发职位的简历，就也会发给我一份，于是每天我会收到几十封简历，开始我还是收到一封看一封，可是后来我就发现自己的工作时间就被碎片化了，因为要时时刻刻接收并筛选简历，后来我就每天晚上 8 点统一筛一遍当天所有的简历，这样就把零散的时间利用起来了，与此同时我还发现，HR 确实帮我们挡住了一些完全不合适的简历节省了我们的时间，此外，有一部分简历则是因为学历原因，其实把候选人约过来聊聊还是很合适的，这时候就需要不拘一格降人才了。还有一部分简历就比较奇葩了，因为 HR 要帮不同的部门招人，所以经常会出现这样的情况，一份好的简历，先送到 A 部门，合适就留下来约面试，不合适就直接拒了，而我所在的 B 部门则完全不知道还有这样一个人的存在。

我不晓得其他部门是如何操作的，反正自从我把自己的邮箱加入到招聘组后，我就有了

优先筛选简历的权力，每天几十份简历，虽然额外增加了工作量，但是每天都能确保筛选到有合适的简历并约来面试。

12.1.4　面试时需要考察的几个点

面试时，主要考察候选人的 3 个方面：

- 技术水平，主要是候选人的编程技术水平。
- 领域知识，主要是候选人对业务的了解程度。
- 软性技能，包括沟通能力、抗压能力、性格。

每个公司面试的流程不太一样。一般而言，有两轮最重要。第一轮是 Team Leader 面试，考察技术水平。第二轮是用人部门的负责人面试，考察领域知识和软性技能。这两轮过了，只要薪水不是太离谱，基本就算过了，这也符合互联网公司简单高效的节奏。

如何考察面试者的技术水平？对于 App 而言，分为 3 个方向：

- 应用类，比如说京东、携程、大众点评、美团这样的 App，它们共同的特点是页面多，都需要频繁地调用 MobileAPI 获取数据，都涉及支付流程，所以这类 App 的开发人员需要对 UI、网络、登录、支付流程都非常熟悉。应用市场也属于这一类，比如豌豆荚。
- 手机管家类。这类 App 虽然也算是应用类，但是很少调用 MobileAPI，它更多关注的是手机系统内部数据的读写，所以这类 App 的开发人员需要对 ActivityManager、Service、BroadcastReceiver 之类的知识很熟悉。
- 游戏类，必须对动画引擎很熟悉，比如说 Cocos2d 和 Lua。

此外，还有一类 Android 从业人员，是在华为、三星这样的硬件厂商做手机系统的二次开发，包括手机系统上自带的一些软件，严格地说，不属于 App 开发。

我本人是从事应用类 App 开发的，这本书也是针对于此的，所以我在面试时一般会考察以下几个方面：

1）Activity 的生命周期。

2）Activity 的 4 种启动方式及使用场合。

3）做过的项目中，Activity 是否有基类，如果有，封装了哪些共用的逻辑？

4）事件的各种使用方式及优缺点。

5）与 HTML5 页面的相互调用。

6）UI 线程的阻塞与解决方案（Runnable 与 Handler）。

7）采用什么姿势调用 MobileAPI 并解析返回的数据？

8）怎样做列表的分页和刷新。

9）登录的实现，包括从哪儿来、到哪儿去的页面跳转机制，记住密码的逻辑设计。

10）性能调优，包括 Layout 调优、Activity 中如何使用 CONST 常量、时间换空间策略、ViewHolder、图集的优化策略、数据缓存和图片缓存，等等。

11）全局变量过多怎么办？

12）写过 UT 没？

13）是否做过自动打包？ Ant、Maven 或 Gradle 任意一种都可以。

大家会看到，我对 Activity 问的很详细，因为它们占据了应用类 App 日常开发工作的绝大部分，但是对 Android 的其他三大组件基本不问，因为在应用类 App 中很少使用。

以上 13 道问题，不一定要求候选人全都会。满足大部分就能干活了，剩下不会的知识点，接下来在工作中会慢慢补齐。

对于 TeamLeader 的要求会更高一些，包括如何检查内存泄露，如何优化内存、多线程、自动打包、框架设计、版本管理等诸多方面。

12.2 无线团队必备的 10 份文档

一个团队成熟与否的标志是文档。文档太多，就违反了敏捷的原则，但有几个文档是必须要提供的，下面分别介绍。

12.2.1 新员工入职文档

这份文档包括：

- 部门组织结构，新员工所在的团队和将要担当的角色。
- 个人简介，用于群发给部门其他成员。
- 要加入的公司邮件组，部门内部用于沟通的 QQ 群或微信群。
- Android 项目的地址，权限申请。
- Bug 管理工具及权限申请。
- 测试环境和仿真环境的地址。
- 产品需求的地址。
- WIFI 设置、VPN 申请、手机邮箱配置、打印机安装，等等。

12.2.2 加强版新员工入职文档

我们针对 Android 开发团队，编写了一份适用于 Android 团队新员工的入职文档。这份文档包括：

- SVN 或 GIT 的权限申请。
- Android 开发常用软件下载。
- 迭代的节奏。
- 业务名词解释。
- Android App 的项目结构。
- Android 自动打包地址（如果有）。
- 模板（模范标准）页面。这里指的是新人写程序时可以用来参考的类或方法。
- 代码规范。

12.2.3　测试机清单

App 开发团队一定要有一份测试机清单，如表 12-1 所示。

表 12-1　测试机清单

测试机型号	操作系统	使　用　人
小米 2	4.1.1	张三
三星 4S	4.3	李四
小米 1	2.3	王五
HUAWEI C8816	4.3	赵六

这样线上有类似机型或系统出了问题，就有机会复现这个问题。Android 几千款机型我们不可能全都采购，一种好的方案是，到友盟上看使用我们 App 的排名前 10 的 Android 手机，采购这些手机，确保开发团队和测试团队各有 1 部这些型号的手机。

12.2.4　模块分工表

把开发人员按照业务线（模块）进行划分。

对于小的团队，每个模块上有 1 个主要开发人员，1 个后备开发人员，二者互为备份。在另一个模块上，这两个人的身份则反过来。如表 12-2 所示。

表 12-2　模块分工表

	主力开发人员	后备开发人员
模块 A	张三	李四
模块 B	李四	张三
模块 C	王五	赵六

分工表一旦制定，就不能随意调整了。不能因为模块 A 忙不过来，就把模块 C 的王五调过去。人员频繁流动，会导致代码质量降低。

对于规模大的公司，每个模块都会有一个 3 ~ 4 人的小团队，所以无所谓主从的关系，但这个小团队会有 1 个 Team Leader。

另一方面，要尽早对 Android 项目进行模块拆分，按照业务线进行模块划分是个不错的选择，把各个独立的业务模块从一个大的 apk 中独立出来，这样才能让负责这个模块的人或者团队独立开发而不受其他团队的影响。

12.2.5　页面逻辑流程文档

每条业务线的业务逻辑都是非常复杂的，表现在 Android 项目中就是十几个 Activity 页面。其中，每个 Activity 中，跳转到其他 Activity 的情况就很多，包括 startActivityForResult 这样跳过去又跳回来的场景；另一方面，每个 Activity 都可能有多个入口。

当我们想修改页面跳转逻辑及传参时，往往会因为考虑不全面而引发灾难性的问题，直

到发版后才发现（多发生于推送）。

于是我们迫切需要每条业务线的页面流程图，在修改业务流程时，这个页面流程图有很好的参考价值。我画过很多这样的页面流程图，一般而言，各条业务线的页面流程都差不多，如图 12-1 所示。

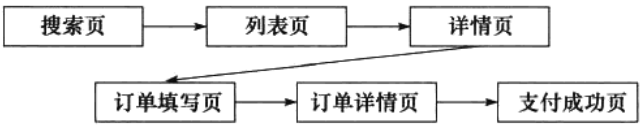


图 12-1 业务流程图

主流程就这么 6 个步骤，各家 App 的区别就在于每个页面上会有一些子页面，用于加强信息收集。基于此，才有了这份页面逻辑流程文档，图 12-2 和图 12-3 是我设计的一款奢侈品 App 的页面流程图。

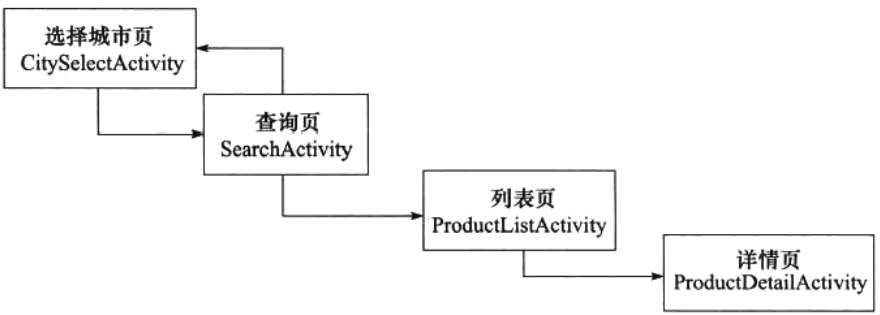


图 12-2 一款奢侈品 App 的页面流程图 -1

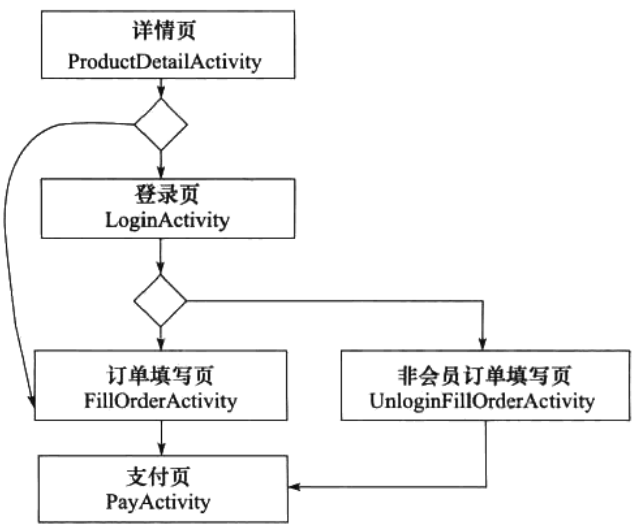


图 12-3 一款奢侈品 App 的页面流程图 -2

不要把所有页面都画在一个图中，线太多，没人能看懂。拆开画，效果会更好。

12.2.6　MobileAPI 接口分布图

一般用 XMind 思维导图来描述一款 App 所用到的 MobileAPI 接口，如图 12-4 所示。

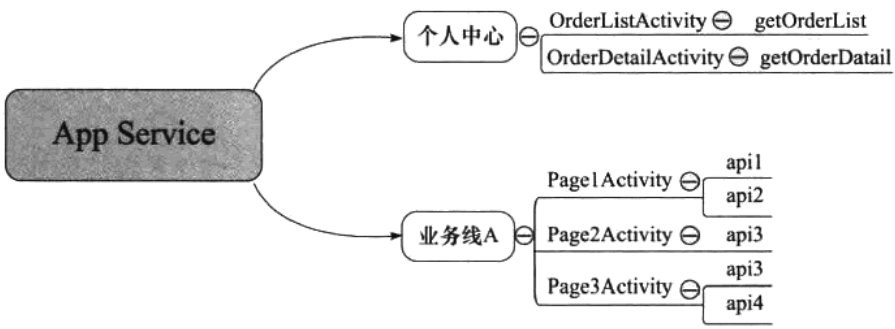


图 12-4　MobileAPI 接口分布图

有了这个图表，我们就可以：

- 定期检查 iOS 和 Android 在做同一功能时所使用到的 MobileAPI 是否一致。
- 每次 MobileAPI 发版上线，相关的测试人员，就可以根据这张图，找到这些 MobileAPI 接口改动影响了哪些页面和功能，需要进行相应的回归测试。

要定期更新这份文档，可以写一个脚本，定期从 Android 代码中，捞出所使用到的 MobileAPI 列表，同步到这份文档中。

12.2.7　版本管理策略文档

无论是使用 SVN 还是 GIT，都要制定一套发版流程。Android 团队中要有专门的开发人员熟悉并遵守这套流程，包括：

- 正常迭代的流程。
- 开新分支做技术调研的流程。
- 紧急上线流程。

流程一般有两种，要么是主干开发主干上线，要么是主干开发分支上线，无论是哪一种，都要落实为文档，切忌口口相传。

12.2.8　框架设计文档

当我们把 AndroidLib 这个业务无关的类库从 App 中抽象出来的时候，就该有一份框架设计文档了。

这份文档我曾经写过，本书第 1 部分的第 1 ~ 4 章就是这份文档的扩充版，请仔细阅读。

12.2.9 发版流程文档

Android 发版并不像 iOS 那样只提交 AppStore 审核，Android 要发布到各大市场，为此，需要修改 AndroidManifest.xml 中的友盟渠道号，才能统计出各大市场的下载量。此外，对外发布的 apk 包要混淆，否则外界可以通过反编译看到我们辛辛苦苦写的代码。

其实考虑问题最多的是测试团队，他们往往会担心：

- 渠道号是否正确？
- 代码是否混淆？
- 版本号是否正确？
- 是否 release 包（而不是 debug 包）？
- 临时决定关闭的功能是否露出来了？
- 是否可以支付、分享、扫描二维码？
- 升级安装是否会引起崩溃？

鉴于以上各点，我们需要制定发版流程并形成文档，包括：

1）产品经理准备发版所需要的描述文字、图片等材料。

2）开发人员进行批量打包工作。

3）测试人员要随机抽取一个 apk 包进行测试，包括我上面谈到的那些测试功能点。

4）推广人员发布到各大市场，要有邮件持续跟踪各个渠道的版本更新进度。

5）在版本仓库上打 Tag，合并分支上的代码到主干（如果采用的是主干开发分支上线的策略）。

12.2.10 App 启动流程图

如果要做 App 性能优化，最好的着手点是 App 从启动到进入首页的流程。

大多数 Android App 的启动 Activity 并不是首页 HomeActivity，而是一个叫做 LaunchActivity 的页面，它的 UI 就是简单的 Splash 动画，同时它肩负着更多的职责，如下所示：

- 注册友盟、推送等第三方组件。
- 加载 Splash 图，同时下载新的 Splash 图以便下次开启时使用。
- 如果是首次打开，则进入引导页。
- 友盟打点，统计激活数
- 如果有消息推送到达，点击消息后想不经过首页而直接进入某个二级页面，其实在代码层面还是要经过 LaunchActivity 的，由它对推送消息进行分发，以决定该跳转到哪个二级页面。

以上这些逻辑交织在一起，非常复杂，尤其是要区分升级版和全新安装版的时候，为此我们需要用 Visio 之类的软件绘制一个 App 启动流程图。

在业界，我们将 LaunchActiity 称为 Bootstrapper。LaunchActivity 把上述这些事情都做完，才会进入到首页 HomeActivity。

12.3　一对一沟通

作为部门管理者，我和团队的每个成员每隔一段时间进行一次一对一沟通，每次半个小时。这是不可缺少的一件工作，比任何其他工作都重要，宁肯少做一个需求，或少参加一个会议。

每每有团队管理者抱怨，每次谈话都得到相同的反馈，抱怨同样的问题而又长期得不到解决，由于每次都老生常谈，所以这样的沟通没有太大意义。

外企的文化是，比如微软，员工每两周都要和直属领导做一次 1:1 沟通，由员工组织材料，汇报最近两周的工作进度，展望接下来两周做什么事情。想当年我每次都要准备半天时间，每次做完沟通之后都是汗流浃背，因为经常会被问得体无完肤不寒而栗，比如原计划为何进展缓慢、新计划为何不切实际、哪里有不足为何还没有提高、计划什么时候提高，等等。

在互联网这几年，我深深地感受到，外企的这套玩法在互联网行业是行不通的，原因如下：

1）互联网工作节奏太快，产品需求都做不完，团队成员不会有半天的时间来准备。

2）团队成员都是为了生计而疲于奔波，没有太长远的规划，都是做完了这期的需求，等待老板分配新的工作而不是主动想要做些什么。

基于以上两点原因，互联网的员工不会提前准备，而是在一对一沟通时被问到什么就回答什么，就像挤牙膏那样。

所以，我对团队的要求是，沟通前花十分钟时间想一想：

1）最近一个月做了哪些事情，有什么提高？

2）自身想要有什么提高？需要我帮助做些什么？

于是，在和团队所有人做完第一轮一对一沟通后，我发现大家还都是蛮有想法的，只是平常被繁重的工作所累，一直都只能压抑在潜意识里罢了，只要加以引导，都是可以挖掘出来的，比如以下几点是我常听到的：

1）强烈要求团建。小规模的团建，找个特色饭馆吃吃饭就好了，然后去唱唱 K。如果是下午，还可以组织大家去看电影。大规模的团建，就要把团队拉出去嗨皮了，注意，这是要消耗额外工时的。

2）有的程序员以后想转行做产品经理，想得到一些锻炼的机会。那我们作为老板，就要给他们提供更多沟通交流的机会。

3）初级程序员希望能分配到一些更高级的 Task。他们渴望新知识，而不是天天画 UI。

4）有些程序员比较好学，他希望团队中有大牛，能学到东西。

5）渴望被表扬。

每轮一对一沟通都要花费一周的时间。不光是沟通，还包括事先准备和沟通后整理反馈的时间。每次做完这件事，我都要生一天病——胸口疼，从来没说过那么多的话。但从长线看，绝对是值得的。

12.4 每周技术分享

技术分享是提高团队技术水平的 3 个方法之一，另外两个是 Code-Review 和修复线上 Crash，本节只谈如何组织技术分享。

技术分享的关键在于坚持。有些公司、部门或者团队往往就是搞个一两次就因为各种忙而夭折了。技术分享短期内是看不到效果的，所以对于急于求成的管理者而言，他们会转而把精力用于做那些短平快的事情。

接下来分享一下我在部门内实施技术分享的经验。

- 每周一次，每次 1 个小时。由于我们的 App 迭代周期是两周，开发人员会很忙，尤其是第二周的周三周四周五，是三个非常重要的时间点，所以我把技术分享的时间定在每周一下班前的一个小时。中途也有周一没有准备好的情况，可以延期到这一周的某一天，但是不能取消。
- 单周由我来讲，双周由团队成员轮流进行。这样每个人就都有 2 周的充足准备时间。我讲的主题偏内功修炼，比如说设计模式、算法、框架设计，等等，团队成员讲的主题，偏实战中的经验和心得体会，会具体到代码和项目层面，比如 xmpp、内存泄漏、Activity 加载模式，等等。

在初期执行的时候，我也是走了一些弯路的。比如我的开发团队整体水平还不是很高，而我讲的又都是高大上的东西，比如我讲过 Android 打包流程，把一群人讲得云山雾罩。

在和开发人员一对一沟通得到反馈后，我把“逼格”适当调整，改为讲有趣的算法题目，就明显受欢迎很多。进一步，我又每次讲几个设计模式，结合着 Android 的实际情况进行讲解，慢慢地提高团队的内功修为——要知道，很多 Android 开发人员都是半路出家，没学过正规的软件开发所需要的这几门基本功，所以他们是需要补上这一课的。

同时，我还发现大家使用 GIT 命令行不是很熟练，我就从给大家介绍一款我用了 3 年的 GIT 图形化操作工具——SmartGit，从而提高开发效率，每天不用为合并代码花费过多的时间。

在团队成员轮流进行技术分享的时候，也遇到了问题，就是每个人都介绍自己感兴趣的东西，往往就变成了讲的人眉飞色舞，听的人不明觉厉。也就是说，没有形成一个体系，比如，通过半年的技术分享，为团队灌输了哪些必备的技术，大家是否在这些技术上有了提高。

于是我和客户端的几个技术经理一起罗列了 Android 和 iOS 必须掌握的若干技术点，然后发给大家去给自己打分，每个技术点都是 5 分制，量化如下：

- 完全不知道：0 分。
- 听说过：1 分。
- 看过介绍的文章：2 分。
- 亲手做过 demo：3 分。
- 项目中使用过：4 分。
- 非常熟悉：5 分。

把大家的自我打分收集上来进行汇总，对团队的整体技术水平就一目了然了。对于团队的技术短板，在每周的技术分享上，会安排团队成员专门进行讲解——当然这个人需要事先花大量的时间去学习、研究并准备 Demo。

对于 Android 应用类开发人员所需要掌握的 20 个技术点，我会在本章后面第 7 节进行介绍。

根据我的经验，按照这种形式坚持下去，半年就能够培养出一批 App 新型技术人才，他们在技术水平、开发效率上都会有质的飞越。技术团队能力不强这一问题，很多高管往往通过招更优秀的人优胜劣汰来解决，其实通过技术培训也能得到一批精兵强将。

12.5　代码评审

我刚到一家互联网公司时发现整个 App 团队在使用 Gerrit 进行代码评审（Code-Review）。搭设 Gerrit 这样一个服务器并不难，难的是整个 App 团队都在坚定不移地贯彻 Code-Review，每个人提交代码，都必须由另一个人审核批准后才能提交到 GIT 上——这不由得让我叹为观止。

但是我观察了一段时间后发现不是那么回事，Code-Review 的具体执行和最初的美好愿景并不匹配。首先我们是个互联网公司，App 迭代的周期只有 2 周，所有开发人员都疲于奔命做需求，哪里还有时间去审核别人的代码，于是就会产生以下几种情况：

- 技术能力强并且责任心强的开发人员，一天 80% 时间用于审核别人提交的代码。
- 技术能力强但是责任心差的开发人员，代码看都不看直接就审核通过了。
- 技术能力弱的开发人员，要他们审核别人的代码，也看不出什么问题来。即使责任心强也是心有余而力不足。

另一个副作用是，因为每次请别人 Code-Review 都要等，所以开发人员倾向于每天下班前一次性提交所有改动，并没有遵守持续开发、持续提交、持续测试的持续集成思想。而审核代码的人就更是辛苦了。

我曾经一度想把 Gerrit 机制废弃了，但是想想还是不妥，主要是因为：

- 好习惯很难养成，坏习惯一句话就能达到了。今天我把 Gerrit 废弃了，等哪天想恢复重新来可就难了。

- 目前线上有各种 bug，倒是还可以归咎为新人经验不足、开发资源不足、测试不充分等各种原因；而废弃 Gerrit 之后，接下来的线上 bug，可就都是没有 Code-Review 导致的了。

思前想后，我的解决方案是：

- 对老员工不再进行 Code-Review。
- 对新员工和实习生、应届生，要为他们每个人指定一个 Code-Review 的老员工，至少 3 个月之内，对他们的 Code-Review 还是要严格执行的。

此外，关于 Code-Review 的标准，每个人心里的秤也不一样。有的人看编码规范，有的人看编码逻辑，你问我哪个对？我也说不出来。Code-Review 我在软件公司也经历过，那时是每周一晚上，所有开发人员坐在一个会议室，在各自的笔记本上看分配给自己的要审核的代码。这期间，每个人都可以提出他认为不妥的各种问题，由被审核人进行回答，只要能自圆其说就行，否则就记下来，Code-Review 会议结束后进行修改。慢慢地，几个月下来，大家的编程风格渐趋一致，这就是 Code-Review 所要达成的效果。

在互联网公司，没空搞我上面说的那套。毕竟两周一次迭代逼死人啊！于是我把 Code-Review 的策略改为，每周一下午，技术经理从上周提交的代码中找出 10 处写的有问题的代码片段，然后给大家进行讲解和讨论。在达成共识后，今后就再也不能写类似的代码了。

那么对于有问题的代码，该怎么处理呢？我的做法是，对于首页、会员中心这种一级页面，代码写的再烂，也不要改，之前毕竟是稳定的，你改了后可能就不好用了，重构这部分代码是件长期的工作。对于二级或三级页面，我们倒是可以分配到具体的开发人员，把问题都改了，毕竟即使改错了，也只是影响局部某个功能。

每周进行一次整个团队的 Code-Review，把每周发现的问题汇总，坚持半年时间，整个团队的代码质量会有很大改善。

在进行 Code-Review 的同时，有一个东西可以顺带搞出来，那就是模板页面，即符合编码规范要求、可以作为编写其他页面的模范页面。如果项目中没有这样的页面，那就找到符合 60% 要求的页面，然后把它改造为符合 100% 要求的。对于 Android 应用类 App 而言，一个模板页面是不够的，至少要提供 Activity、Adapter、Entity、Fragment 这 4 个模板页，其中 Activity 要包括对 MobileAPI 的调用。

有了模板页，所有开发人员的编码就有章可循，单纯搞 Code-Review 和编码规范都太抽象，一定要有能落地的东西，那就是模板页。

12.6 对 Android 团队 Leader 的定位

Android 团队 Leader 要负责的工作罗列如下，其中绝大部分也适用于 iOS 团队 Leader：

- 每次迭代把 Task 分配到具体开发人员。

- 组织线上 Crash 的修复。
- 处理线上突发 bug。
- 排查每日客人投诉的问题。
- 解决团队遇到的技术难题。
- 组织每周 Code-Review。
- 组织每周例会。

团队 Leader 一定要明确自己的职责，注意以下两点：

- 不要给自己分配具体的需求开发，你会发现，上述管理工作会消耗掉你大量的时间。
- 努力不要使自己成为瓶颈。很耗费时间的事情，及时分配到具体的开发人员。bug 如果都集中到自己手里，那么一定要及时分下去。

哪些工作是要尽早分出去给具体的开发人员的呢？具体包括：

- Android 项目的打包。
- 代码混淆。
- 设计 Android 的 Lib 框架，交给架构组去做。
- 技术调研。
- Monkey 日志分析。

12.7　Android 应用开发所需技能自我评测

有个开发人员曾经跟我说，他很迷茫，接下来是该去看 Android 系统源码，还是每天继续做应用，但是感觉每天都是画 UI 和调用 MobileAPI 处理 JSON，没有技术上的提升空间。

这个问题我思考了一个晚上，列出来一个从事 Android 应用的开发人员所需要精通的 20 个技能点，如下所示：

1）Activity 相关。App 应用开发，以 Activity 使用最多，涉及 LaunchMode、onSaveInsatnce-State、生命周期等技术。

2）Fragment 相关技术。用的人不少，想明白是咋回事的人不多。这里推荐一本书：《Creating Dynamic UI with Android Fragments》。

3）序列化技术。有 Parcelable 和 Serializable 两种。前者是基于 Service 的，后者是基于 Bundle 的，二者实现原理不同，但是达到的效果差不多。

4）ImageLoader 的原理和使用。类似的，还可以学习 Facebook 新近开源的 Fresco，它对图片的处理会更好一些。

5）fastJSON 或 GSON 的使用。做 App 不会用实体自动匹配 JSON 数据，相当于白做。

6）多线程相关。包括 Handler、Looper、ExecutorService 等。

7）Adapter 和 ListView。这两个技术捆在一起，经常容易崩溃，尤其是分页的时候，要

仔细研究深刻领会。

8）用户 Cookie 设计。需要把登录机制彻底搞清楚，包括在 HttpRequest 头中夹带 Cookie 来进行用户身份验证的技术。

9）网络请求封装。使用 AsyncTask 的网络底层封装，使用 Handler+Runnable 的网络底层封装。

10）Android 与 HTML5 的交互。包括 Android 调用 HTML5 的方法，以及 HTML5 调用 Android 的方法。

11）代码混淆。没用过 ProGuard，不知道 keep 相关语法，就还是初级水平。

12）Android 打包机制。涉及 Android SDK 中的若干命令。对 Android 打包过程做的每一件事都很清楚。进一步是 Android 多项目依赖的打包技术。Ant、Gradle 或者 Maven，掌握其中任何一种打包机制即可。

13）线上 Crash 分析并修复。要具备通过分析 Crash 信息修复线上 Crash 的能力。

14）内存泄漏。包括内存优化、内存泄漏的场景、MAT 工具的使用。

15）调试工具。包括 DDMS、Eclipse 或 Android Studio 的调试功能。

16）Monkey 机制。Android 开发人员如何对一款 App 进行 Monkey 测试。这算是附加技能吧。

17）单元测试。这里指的是 JUnit。对复杂的算法写过单元测试以保证其没有问题。

18）GIT 的高级功能。包括 Stage、Rebase、Revert、Stash、Cherry Pick 和 Sub Module 等概念。如果项目中使用的是 SVN，那么要掌握 SVN 的版本管理策略。

19）插件化编程。哪怕知道一点 DexClassLoader 的概念也好。这年头，没做过插件化编程，出门面试都不好意思说自己是做 Android 开发的。

20）设计模式。对常见的设计模式如工厂、生成器、适配器、代理、策略模式耳熟能详。

由此而看到，做 Android 应用开发，不需要花太多精力去看 Android 系统源码，要先确保我上面罗列的 20 点所涉及的技术都掌握了。

12.8 App 开发人员的学习路线

上节我介绍了从事 Android 应用类开发所需要具备的 20 项技能。这里再唠叨几句。

对于设计模式，要逼着自己都实现一遍，然后，把这 23 个模式都忘了，只需要记住 SOLID 原则就够了。这就像金庸笔下的独孤九剑，以无招胜有招。我学习设计模式这门技术有 10 年了，就是这个套路，至今受益匪浅。

无论是 iOS 还是 Android 技术，你会发现，很多人比拼的是谁知道更多的 API，从而能快速地做出 PM 想要的功能。其实我一直不那么认为，人脑的容量就像内存一样是有限的，没必要记那么多 API，我只要记遇到问题时哪里能找到 API 就好了。打个比方，之前我们脑子里记的是值类型，接下来我将记引用类型，这明显能节省出很大的空间，用来记那些更重

要的信息。在微软，我们称之为 SMART。

开发人员一定要解放思想，才能打破陈规，做出有创造性的工作。有一道题目非常好，我曾经问过很多人：4 个 0，使用任何规则，如何得到 24 点。很多人在网上看过这道题目，于是告诉我答案是用阶乘可以得到结果。但其实我们的思维已经被外界的条条框框束缚住了。最无厘头的答案是 00:00，这也是 24 点，你可以说我耍赖，但是我的确解出了，而且是用最简单有效的办法。

解放思想的最佳实践就是跨界。我曾经做技术遇到了瓶颈，沉沦过一段时间，这期间我开始学习烹饪。我就发现炒菜是装饰者模式（Decorator），因为在炒菜的时候我们会依次放不同的作料，不断地给这道菜增加新的味道。

以下是我看过的一些书籍，推荐给读者：

1）《疯狂 Android 讲义》 我就是看这本书入门的。这本书很实际，比较适合于应用类 App 开发人员做入门教材。已经入门的，建议也看一遍，梳理一下知识，做进一步提高。

2）《Creating Dynamic UI with Android Fragments》 这本书是专门讲 Fragment 的。关于 Fragment，很多书都只言片语，语焉不详。唯独这本书把 Fragment 从头到尾仔仔细细讲了一遍。目前国内没有中文版。Fragment 是 Android 技术中比较高大上的部分。

3）《Android 应用测试与调试实战》[㊀] 乍一看这本书是讲测试的，其实不然，书中的很多章节涉及依赖注入、内存分析、打包部署等开发人员必知必会的技术。强烈建议仔仔细细通读之。

4）《Java 与模式》 这是本古董级的书了，所有介绍设计模式的书，论厚度，无出其右。另一点好处是，这本书是基于 Java 的，对 Android 开发人员比较适合。

5）《Git 权威指南》[㊁] 这本书名副其实，算是把 Git 讲明白了。说到这里，我还要推荐一款非常好用的 Git 图形化工具。除了能用来进行日常的 Pull、Push 和 Rebase 操作外，还能教会你 Git 的高级用法，比如 Cherry Pick、Stash、Sub Module 等。

12.9　本章小结

本章介绍的 Android 的团队组建和日常管理。制度是死的，人是活的。管理团队，千万别形而上学。尤其在移动互联网这个日息万变的行业，照搬软件和互联网的那套管理方式是行不通的。“短、平、快”是移动互联网一切工作的核心。

移动互联网的开发人员属于供不应求的状况，我们要学会尊重人才，逐步转变原先“买方市场”的传统思维模式。现在是卖方市场，各大公司的 HR 和老板，你们准备好了吗？

㊀ 此书已由机械工业出版社出版，书号为 978-7-111-46018-3。——编辑注

㊁ 此书已由机械工业出版社出版，书号为 978-7-111-34967-9。——编辑注

为了写这本书，作者分析了市场上有名的上百款App，能够费这么多心血去研究技术实现的人，在我看来至少是一个充满好奇心的人。正是这种拥有好奇心并执着探索的人，才推动了近百年来的科学发展。

—— 奇虎360董事长 **周鸿祎**

本书与其他书籍完全不同，纯从实战出发，在官方文档之上，阐述实际开发中应该掌握的那些来之不易的经验，其中多是过来人踩过坑、吃过亏才能总结出来的东西。不少章节类似于Effective系列名著的风格，有很高的价值。

—— 美团技术学院院长，CSDN和《程序员》杂志前总编 **刘江**

整本书并不是从枯燥的文档中提炼而来，而是真切地从一个互联网从业者的亲身经历和交流中得来。作为需要时刻紧跟移动浪潮的App开发人员，这本书是值得一读的好书。

—— 大众点评首席架构师 **屠毅敏**

这本书针对有经验的Android开发者，你会发现很多场景都是曾经或即将带给你疑问的，作者针对Android开发过程中一个个具体问题给出了解决方案，非常实用。特别是异常处理的部分，这是我第一次发现有这么完整地介绍Android异常的书籍，非常有学习价值。

—— 腾讯无线研发工具总监 **欧阳骏**

这本书在老包的诙谐笔法之下凝聚了他多年来奋战在一线的研发和管理经验，具备很强的实战参考意义，并非一般的App开发入门或者泛泛之谈。从菜鸟成长为大拿如果有捷径的话，莫过于经常与高手过招。在开发HTML的Web时代，前端开发人员很喜欢用右键点击网页查看源代码的方式来学习优秀网站的开发思路，后来从中提炼成了很多浏览器的插件，比如Firefox的Firebug等。进入App开发时代后，如何从高手的作品中进行类似的学习呢？老包在第9章App竞品技术分析里给出了相当精彩的解决方案。整本书的结构清晰，从最为痛苦的考古重构讲起，到Crash异常等的分析处理，再到持续集成与团队协作的App开发项目管理，包含每个小工在成长之路上都可能碰到的问题。相信您阅读之后必定会有所收获。

—— 途牛旅游网无线中心总经理 **陈世宏**

上架指导：计算机/移动开发
ISBN 978-7-111-51638-5
9 787111 516385 >
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn