



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。



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



华章图书



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

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 序 —

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。

The logo consists of a stylized graphic element resembling a stack of books or a bar chart, composed of vertical bars of varying heights. Below this graphic, the text "HZ BOOKS" is written in a bold, sans-serif font. Underneath that, the Chinese characters "华章图书" are displayed in a larger, traditional-style font.

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、异常捕获和分析、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。如果你对其中有深刻的共鸣，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。

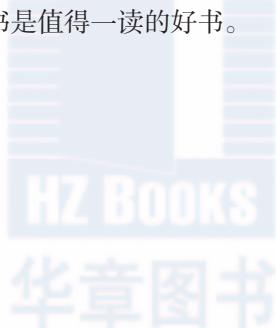
本书第 9 章我认为是最接地气也是最有特色的章节，从分析国内热门的 App 开始，帮助读者了解最前沿的大公司的移动开发的技术方向。有很多技术点是小的 App 开发团队并不会

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，到目前已经大部分的应用抛弃，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.cnblogs.com/Jax/p/4656789.html。

包建强

2015 年 8 月 3 日于北京

Contents 目 录

序一
序二
序三
前言

第一部分 高效 App 框架设计与重构

第 1 章 重构成，夜未眠	3
1.1 重新规划 Android 项目结构	3
1.2 为 Activity 定义新的生命周期	5
1.3 统一事件编程模型	7
1.4 实体化编程	9
1.4.1 在网络请求中使用实体	9
1.4.2 实体生成器	11
1.4.3 在页面跳转中使用实体	12
1.5 Adapter 模板	14
1.6 类型安全转换函数	16
1.7 本章小结	17
第 2 章 Android 网络底层框架设计	19
2.1 网络低层封装	19

2.1.1 网络请求的格式	19
2.1.2 AsyncTask 的使用和缺点	21
2.1.3 使用原生的 ThreadPoolExecutor + Runnable + Handler	24
2.1.4 网络底层的一些优化工作	28
2.2 App 数据缓存设计	32
2.2.1 数据缓存策略	32
2.2.2 强制更新	35
2.3 MockService	36
2.4 用户登录	38
2.4.1 登录成功后的各种场景	39
2.4.2 自动登录	41
2.4.3 Cookie 过期的统一处理	44
2.4.4 防止黑客刷库	45
2.5 HTTP 头中的奥妙	46
2.5.1 HTTP 请求	46
2.5.2 时间校准	48
2.5.3 开启 gzip 压缩	51
2.6 本章小结	52
第3章 Android 经典场景设计	53
3.1 App 图片缓存设计	53
3.1.1 ImageLoader 设计原理	53
3.1.2 ImageLoader 的使用	54
3.1.3 ImageLoader 优化	55
3.1.4 图片加载利器 Fresco	56
3.2 对网络流量进行优化	58
3.2.1 通信层面的优化	58
3.2.2 图片策略优化	59
3.3 城市列表的设计	61
3.3.1 城市列表数据	61
3.3.2 城市列表数据的增量更新机制	63

3.4 App 与 HTML5 的交互	64
3.4.1 App 操作 HTML5 页面的方法	64
3.4.2 HTML5 页面操作 App 页面的方法	65
3.4.3 App 和 HTML5 之间定义跳转协议	66
3.4.4 在 App 中内置 HTML5 页面	67
3.4.5 灵活切换 Native 和 HTML5 页面的策略	68
3.4.6 页面分发器	68
3.5 消灭全局变量	70
3.5.1 问题的发现	70
3.5.2 把数据作为 Intent 的参数传递	71
3.5.3 把全局变量序列化到本地	71
3.5.4 序列化的缺点	75
3.5.5 如果 Activity 也被销毁了呢	79
3.5.6 如何看待 SharedPreferences	80
3.5.7 User 是唯一例外的全局变量	80
3.6 本章小结	81

第 4 章 Android 命名规范和编码规范 83

4.1 Android 命名规范	83
4.2 Android 编码规范	86
4.3 统一代码格式	89
4.4 本章小结	90

第二部分 App 开发中的高级技巧

第 5 章 Crash 异常收集与统计 93

5.1 异常收集	93
5.2 异常收集与统计	96
5.2.1 人工统计线上 Crash 数据	96
5.2.2 第一个线上 Crash 报表：Crash 分类	97

5.2.3 第二个线上 Crash 报表：Crash 去重	99
5.2.4 线上 Crash 的其他分析工作	104
5.3 本章小结	105
第6章 Crash 异常分析	107
6.1 Java 语法相关的异常	108
6.1.1 空指针	108
6.1.2 角标越界	109
6.1.3 试图调用一个空对象的方法	110
6.1.4 类型转换异常	110
6.1.5 数字转换错误	111
6.1.6 声明数组时长度为 -1	111
6.1.7 遍历集合同时删除其中元素	112
6.1.8 比较器使用不当	114
6.1.9 当除数为 0	115
6.1.10 不能随便使用的 asList	116
6.1.11 又有类找不到了（一）：ClassNotFoundException	116
6.1.12 又有类找不到了（二）：NoClassDefFoundError	117
6.2 Activity 相关的异常	117
6.2.1 找不到 Activity	117
6.2.2 不能实例化 Activity	118
6.2.3 找不到 Service	118
6.2.4 不能启动 BroadcastReceiver	119
6.2.5 startActivityForResult 不能回传	119
6.2.6 猴急的 Fragment	120
6.3 序列化相关的异常	120
6.3.1 实体对象不支持序列化	121
6.3.2 序列化时未指定 ClassLoader	121
6.3.3 反序列化时发现类找不到：被 ProGuard 混淆导致的崩溃	122
6.3.4 反序列化时发现类找不到：传入畸形数据	123
6.3.5 反序列化时出错	123

6.4	列表相关的异常	123
6.4.1	Adapter 数据源变化但是没通知 ListView	124
6.4.2	ListView 滚动时点击刷新按钮后崩溃	125
6.4.3	AbsListView 的 obtainView 返回空指针	125
6.4.4	Adapter 数据源变化但是没调用 notifyDataSetChanged	126
6.5	窗体相关的异常	126
6.5.1	窗口句柄泄露	126
6.5.2	View not attached to window manager	128
6.5.3	窗体在不恰当的时候获取了焦点	129
6.5.4	token null is not for an application	130
6.5.5	permission denied for this window type	131
6.5.6	is your activity running	131
6.5.7	添加窗体失败	133
6.5.8	AlertDialog.resolveDialogTheme	134
6.5.9	The specified child already has a parent	136
6.5.10	子线程不能修改 UI	137
6.5.11	不能在子线程操作 AlertDialog 和 Toast	141
6.6	资源相关的异常	143
6.6.1	Resources\$NotFoundException	143
6.6.2	StackOverflowError	144
6.6.3	UnsatisfiedLinkError	144
6.6.4	InflateException 之 FileNotFoundException	145
6.6.5	InflateException 之缺少构造器	145
6.6.6	InflateException 之 style 与 android:textStyle 的区别	146
6.6.7	TransactionTooLargeException	147
6.7	系统碎片化相关的异常	147
6.7.1	NoSuchMethodError	147
6.7.2	RemoteViews	148
6.7.3	pointerIndex out of range	149
6.7.4	SecurityException 之一： Intent 中图片太大	150
6.7.5	SecurityException 之二： 动态加载其他 apk 的 activity	151

6.7.6 SecurityException 之三：No permission to modify thread	151
6.7.7 view 的 getDrawingCache() 返回 null	152
6.7.8 DeadObjectException	153
6.7.9 Android 2.1 不支持 SSL	153
6.7.10 ViewFlipper 引发的血案	153
6.7.11 ActivityNotFoundException	154
6.7.12 Android 2.2 不支持 xlargeScreens	154
6.7.13 Package manager has died	155
6.7.14 SpannableString 与富文本字符串	155
6.7.15 Can not perform this action after onSaveInstanceState	156
6.7.16 Service Intent must be explicit	157
6.8 SQLite 相关的异常	157
6.8.1 No transaction is active	158
6.8.2 忘记关闭 Cursor	158
6.8.3 数据库被锁定	159
6.8.4 试图再打开已经关闭的对象	159
6.8.5 文件加密了或无数据库	159
6.8.6 WebView 中 SQLite 缓存导致的崩溃	160
6.8.7 磁盘读写错误	161
6.8.8 android_metadata 表不存在	161
6.8.9 android_metadata 表中的 locale 字段	162
6.8.10 数据库或磁盘满了	162
6.9 不明觉厉的异常	162
6.9.1 内存溢出	163
6.9.2 Verify Failed	163
6.10 其他情况的异常	163
6.10.1 TimeoutException	164
6.10.2 JSON 解析异常	164
6.10.3 JSONArray 在初始化时为空	164
6.10.4 第三方 SDK 抛出的 Crash	165
6.10.5 两个不同类型的 View 有相同的 id	165

6.10.6 LayoutInfiater.from().inflate() 使用不当导致的崩溃	166
6.10.7 ViewGroup 中的玄机	166
6.10.8 Monkey 点击过快导致的崩溃	167
6.10.9 图片缩放很多倍	168
6.10.10 图片宽高为 0	168
6.10.11 不能重复添加组件	168
6.11 本章小结	169
第 7 章 ProGuard 技术详解	171
7.1 ProGuard 简介	171
7.2 ProGuard 工作原理	172
7.3 如何写一个 ProGuard 文件	172
7.3.1 基本混淆	172
7.3.2 针对 App 的量身定制	175
7.3.3 针对第三方 jar 包的解决方案	177
7.4 其他注意事项	178
7.5 本章小结	179
第 8 章 持续集成	181
8.1 版本管理策略	181
8.1.1 三种版本管理策略	181
8.1.2 特殊情况的版本管理策略	183
8.2 使用 Ant 脚本打包	184
8.2.1 Android 打包流程	184
8.2.2 打包时的注意事项	189
8.3 Monkey 包的生成	190
8.4 自动打包	191
8.4.1 安装和配置各种软件	192
8.4.2 准备 Ant 打包脚本	193
8.4.3 配置 CCNET	193
8.4.4 搭建 IIS 站点下载 apk 包	193

8.4.5 自动打包流程小结	193
8.5 批量打渠道包	194
8.5.1 基于 apk 包批量生成渠道包	194
8.5.2 基于代码批量生成渠道包	195
8.6 Android 发版流程	197
8.7 分类打渠道包	198
8.7.1 分门别类生成渠道包	198
8.7.2 批量上传 apk 的两种方式	199
8.8 灵活切换服务器	199
8.9 单元测试	201
8.10 本章小结	203
第 9 章 App 竞品技术分析	205
9.1 竞品分析概述	205
9.1.1 App 竞品定义	205
9.1.2 竞品分析要研究的几个方向	206
9.1.3 竞品分析与拿来主义	206
9.2 App 安装包的结构	207
9.2.1 Android 安装包的结构	207
9.2.2 iOS 安装包的结构	208
9.3 竞品技术一瞥：开机速度	208
9.4 竞品技术二瞥：HTML5 页面的打开速度	209
9.4.1 把 HTML5 页面嵌入到 Zip 包中	209
9.4.2 Zip 包的增量更新机制	209
9.4.3 制作 Zip 增量包	210
9.4.4 使用 WebView 预先加载 HTML5 并缓存到本地	211
9.5 竞品技术三瞥：安装包的大小	211
9.5.1 从几件小事说起	211
9.5.2 安装包为什么那么大	212
9.5.3 png 和 jpg 的区别及使用场景	212
9.5.4 Splash、引导图和背景图	213

9.5.5 iOS 的 1 倍图、2 倍图和 3 倍图	213
9.5.6 在 iOS 中进行图片拉伸和旋转	214
9.5.7 使用 XML 配置动画	214
9.5.8 iOS 使用 storyboard 还是 xib	215
9.5.9 字体文件的学问	215
9.5.10 表情图片打包下载	217
9.5.11 清除未使用图片	218
9.5.12 Proguard 不只是用来混淆的	218
9.5.13 在 iOS 中使用 pdf 格式的图片	218
9.5.14 iOS 的包永远比 Android 包体积大吗	219
9.5.15 从代码层面减少 iOS 包的体积	220
9.6 竞品技术四瞥：性能优化	220
9.6.1 App 自动选取最佳服务器的策略	220
9.6.2 使用 TCP+Protobuf	222
9.7 竞品技术五瞥：数据采集工具	223
9.7.1 页面跳转器	223
9.7.2 打点统计	226
9.7.3 ABTest	230
9.8 竞品技术六瞥：热修补	232
9.8.1 Native 页面和 HTML5 页面的相互切换	232
9.8.2 在 iOS 中使用脚本编程	233
9.9 竞品技术七瞥：曲径通幽	237
9.9.1 一切皆可配置	237
9.9.2 App 后门	238
9.9.3 Android 包中 META-INF 目录的妙用	239
9.9.4 classes.dex 的拆与合	241
9.10 竞品技术八瞥：模块化拆分	242
9.10.1 iOS 资源拆分与模块化	242
9.10.2 Android 模块化拆分	243
9.11 竞品技术九瞥：第三方 SDK	244
9.11.1 HTML5 篇	244

9.11.2 iOS 篇	245
9.11.3 Android 篇	245
9.11.4 其他	246
9.12 竞品技术十瞥：版本策略与 App 彩蛋	246
9.12.1 版本策略	246
9.12.2 App 彩蛋	246
9.13 本章小结	247

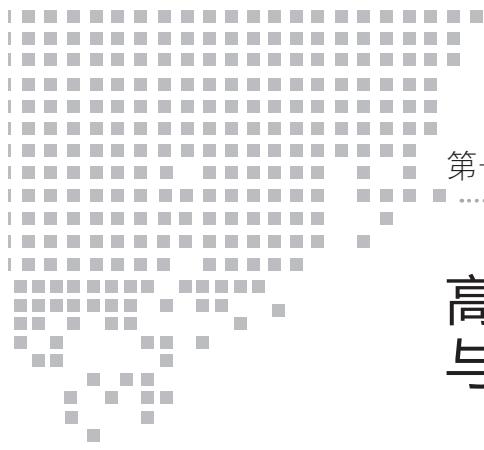
第三部分 项目管理和团队建设

第 10 章 项目管理决定了开发速度	251
10.1 项目管理中的三驾马车	251
10.1.1 为什么不能没有测试团队	252
10.1.2 产品经理应做的事	253
10.1.3 开发人员的喜怒哀乐	254
10.1.4 项目经理的职责	254
10.2 优化团队结构，让敏捷流程跑得更快	255
10.2.1 平行模式还是垂直模式	255
10.2.2 让 HTML5 站点和 MobileAPI 的进度提前一个迭代	256
10.2.3 如何进行模块化分工	256
10.3 App 敏捷开发流程	257
10.3.1 四周时间的开发流程	257
10.3.2 两周时间的开发流程	261
10.3.3 一周时间的开发流程	262
10.3.4 即时更新策略	263
10.4 项目经理的百宝箱	263
10.4.1 项目经理的任务评估表	263
10.4.2 贴小纸条的艺术	264
10.4.3 敏捷迭代中的会议纪要	265
10.4.4 开站例会的技巧	266

10.4.5 如何确保项目不延期	268
10.4.6 迭代风险管理	268
10.5 迭代中的测试工作	269
10.5.1 冒烟测试	269
10.5.2 探索性测试	271
10.5.3 Monkey 测试	271
10.6 高层对敏捷流程的干预	272
10.6.1 重构与产品需求的平衡	272
10.6.2 提高效率，拒绝 6×12	273
10.6.3 无线部门的座位安排	274
10.6.4 静时	276
10.7 本章小结	277
第 11 章 日常工作中的问题解决	279
11.1 使用二分法排查问题	279
11.2 找到能稳定重现问题的人	281
11.3 小流量包	282
11.4 建立全国范围的测试群	283
11.5 如何与用户沟通	284
11.6 日志与 App 性能	286
11.7 从新人入职作业入手	286
11.8 本章小结	287
第 12 章 无线团队的组建和管理	289
12.1 从面试谈起	289
12.1.1 如今是卖方市场	289
12.1.2 名校论不适用无线开发	290
12.1.3 如何搞到更多的简历	290
12.1.4 面试时需要考察的几个点	291
12.2 无线团队必备的 10 份文档	292
12.2.1 新员工入职文档	292

12.2.2 加强版新员工入职文档	292
12.2.3 测试机清单	293
12.2.4 模块分工表	293
12.2.5 页面逻辑流程文档	293
12.2.6 MobileAPI 接口分布图	295
12.2.7 版本管理策略文档	295
12.2.8 框架设计文档	295
12.2.9 发版流程文档	296
12.2.10 App 启动流程图	296
12.3 一对一沟通	297
12.4 每周技术分享	298
12.5 代码评审	299
12.6 对 Android 团队 Leader 的定位	300
12.7 Android 应用开发所需技能自我评测	301
12.8 App 开发人员的学习路线	302
12.9 本章小结	303





第一部分 *Part 1*

高效 App 框架设计 与重构



- 第1章 重构，夜未眠
 - 第2章 Android 网络底层框架设计
 - 第3章 Android 经典场景设计
 - 第4章 Android 命名规范和编码规范
-

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：

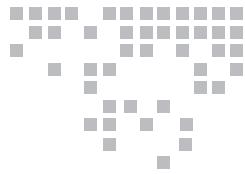
1) 如果不重构，会导致很严重的 App 功能问题，那么这是很严重的 bug，宁肯砍掉 1 ~ 2 个功能也一定要修复的，需要和产品经理商量，由他们决策。

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。



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。

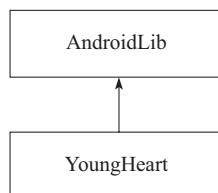


图 1-1 重构后的项目依赖关系

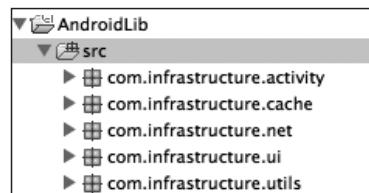


图 1-2 AndroidLib 项目结构

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 App BaseActivity 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。

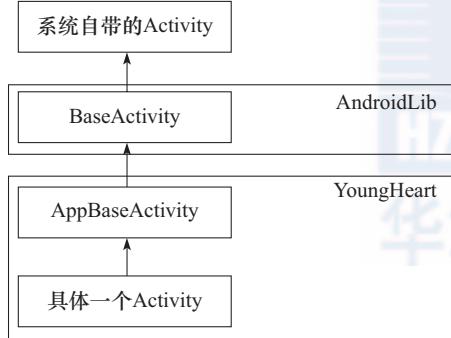


图 1-3 基类的继承关系

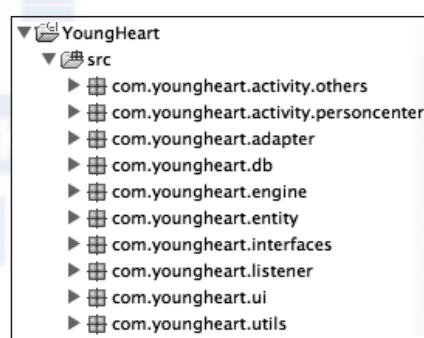


图 1-4 Android 重构后的项目结构

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：SQLite 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。

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。

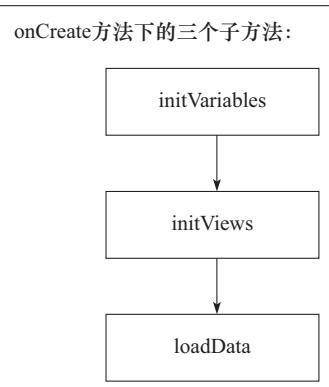


图 1-5 onCreate 方法下的三个子方法

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) {
            goto LoginActivity();
        }
    });

```

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 Activity 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。

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。

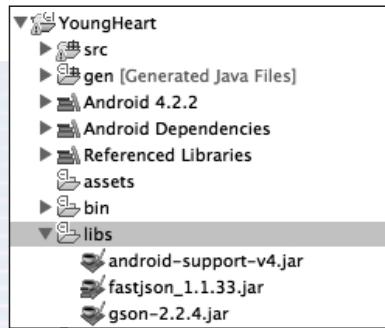


图 1-6 在 Android 项目中添加 fastJSON 和 GSON 的 jar 包

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 EntityGenerator 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^①，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，让它同时也可生成 Android 和 iOS 实体，如图 1-7 所示。

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^②。

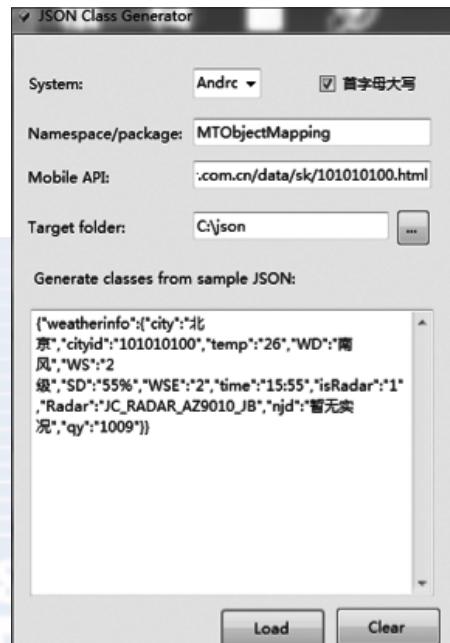


图 1-7 实体生成器的左边界面

^① 这个工具的地址如下：<http://www.xamasoft.com/json-class-generator/>

^② 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_cinelist, 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 savedInstanceState) {
    setContentView(R.layout.activity_listdemo);
    lvCinemaList = (ListView) findViewById(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。

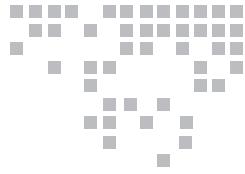
1.7 本章小结

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

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。





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 onFailure(String errorMessage);

    @Override
    protected void onPreExecute() {
    }

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

目前我们只定义了 onSuccess 和 onFailure 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 onFailure(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。

图中只列出了 8 个，还有一个 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>
```

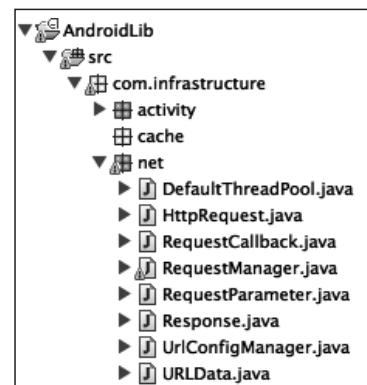


图 2-1 轻量级的网络底层框架

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 onFailure 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.getErrorMassage());
} 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.getId());
        }
    }
}
```

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.getId());
        }
    }

    @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.getId());
        }
    }
};
```

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.getId());
        }
    }

    @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 onFailure 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 AppCompatActivity 中，如下所示：

```

public abstract class AppCompatActivity 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(AppCompatActivity.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 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=v1&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。

- 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,
```

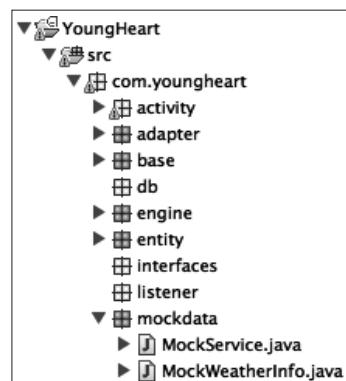


图 2-2 mockdata 包

```

        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);
            startActivityForResult(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，它就会去把相关得数据取出并返回。

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。

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 `Http-Response` 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) {
```



图 2-3 App 的登录界面

```

        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。^④

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，

^④ 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 App BaseActivity 中，放在和 onFail 方法平级的位置：

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

    new AlertDialog.Builder(App BaseActivity.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(
                        App BaseActivity.this,
                        LoginActivity.class);
                    intent.putExtra(App Constants.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。[⊖]

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

[⊖] 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）：

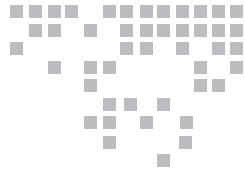
```
String strResponse = "";
if ((response.getEntity().getContentEncoding() != null)
    && (response.getEntity().getContentEncoding()
        .getValue() != null)) {
    if (response.getEntity().getContentEncoding()
        .getValue().contains("gzip")) {
        final InputStream in = response.getEntity()
            .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。



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，看似都是些平淡无奇的 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。[⊖]

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

[⊖] 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 内存中的缓存全部清除，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）。^①

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^②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)
```

^① 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>。

^② 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 ImageLoaderConfiguration。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 AppCompatActivity 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。

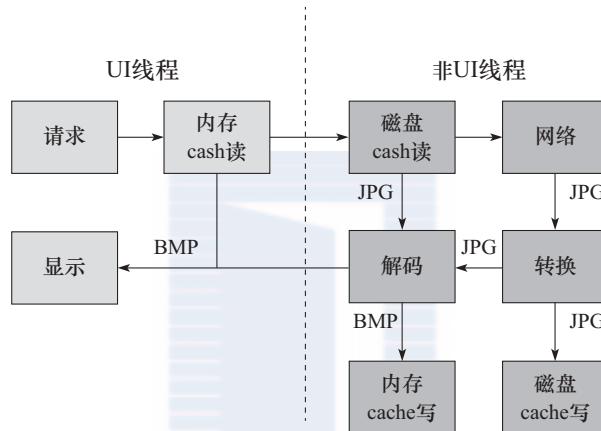


图3-1 Image Pipeline的工作流

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=\(encode value\)](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：

$$S = (w_1 - a) \times (w_1 - w) + (h_1 - h) \times (h_1 - h)$$

w 和 h 是实际的图片宽和高， w_1 和 h_1 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一段时间。

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。

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",
            "type": "u"
        }
    ]
}
```

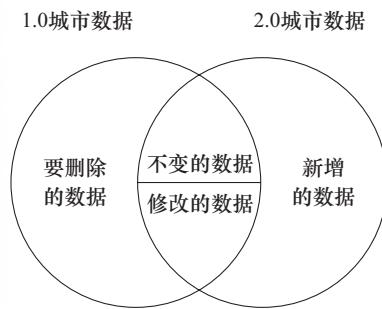


图 3-2 比较两个版本城市数据间的异同

```

        "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 JSInterface1 {
    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 JSInterface1(), "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 createTime 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，追赶上来了就切换回 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。

