移动开发

Android 插件化开发指南

包建强　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Android 插件化开发指南 / 包建强著 . —北京：机械工业出版社，2018.8
（移动开发）

ISBN 978-7-111-60336-8

I. A…　II. 包…　III. 移动终端 - 应用程序 - 程序设计 - 指南　IV. TN929.53-62

中国版本图书馆 CIP 数据核字（2018）第 145767 号

Android 插件化开发指南

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：李秋荣
印　　刷：北京市兆成印刷有限责任公司　　版　　次：2018 年 8 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：22
书　　号：ISBN 978-7-111-60336-8　　定　　价：79.00 元

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

Preface 序　一

当接到包老师邀请写序时，我真是受宠若惊。大家都知道，作为一个程序员，写代码拿手那是自然的事情，文字工作实在不是我的强项，但是能给包老师的书写序，实在是荣幸之至，更何况盛情难却呢。

我与包老师相识是在 2015 年的一个关于 DroidPlugin 的分享会上，那会儿我还在 360 公司做手机助手。2014 年年底到 2015 年年初的时候，在公司比较空闲，所以写了一个插件化框架叫 DroidPlugin，并在 2015 年 7 月份以公司的名义在 GitHub 上开源了出来，承蒙大家抬爱，在极短的时间内便收获了数千颗星。因为在项目的介绍中我写上了“免修改、免重打包、免安装运行”的宣传语，所以也被一些人冠以“360 黑科技”，在知乎上甚至有人认为这是 360 的什么阴谋。不管是不是什么阴谋，但这个项目让我认识了很多行业中的大牛（包老师就是其中一位），确实是我意料之外的事情。后来跟包老师、任玉刚老师几位经常结成“饭醉团伙”，自然也少不了讨论插件化技术。既然与包老师是因为 DroidPlugin 开源项目相识，而包老师这本书的内容涉及的很多技术也跟其相关，所以我还是主要介绍一下 DroidPlugin 项目中的相关技术。

如你所见，那会儿正是插件化技术大热的时候，各大公司也相继开源了自己的插件化框架，但是总结来看，所涉及的技术原理也大同小异。但是 DroidPlugin 在其中的确算是比较“奇葩”的一个，因为它实际不止是一个插件化框架，更多的算是一个用户态虚拟机，后来大多数的双开软件也都参考了它的源码或者原理。

要在 Android 上实现免安装、免修改运行一个 App，并不是一件容易的事情。因为 Android 系统设计和权限的限制，我们需要做很多的工作……这跟 Docker 不一样，虽然都是 Linux 系统，但在 Android 上我们不能要求 root 权限，而 Docker 则没有这些限制，因此 Linux 内核提供的 namespace、cgroup、chroot 等能力也自然可以应用在 Docker 中。同时因为国内各大厂商的定制系统，我们也做了大量的适配工作。虽然现在看起来这个框架还有不少缺陷，但在当时确实算是比较先进的一个框架。

DroidPlugin 使用了一些比较 hack 的技巧，但是总结起来也就是一句话“利用 hook 技

术实现欺上瞒下，从而达到免安装运行的目的”。因为 Android 系统出于安全考虑，系统服务与 App 进程采用分进程设计，它们之间通讯使用 binder 技术，系统服务实际上是不知道 App 进程中运行的具体代码，这为我们实现“欺上瞒下”提供了可能。所谓“欺上瞒下”中的“欺上”是指我们可以通过某种技术手段，拦截所有插件发向系统服务的通信，让系统不知道插件在我们的宿主 App 中运行；“瞒下”是指通过拦截并模拟系统服务发向插件的通信，让插件“以为”自己已经被安装。这样我们就可以模拟一个环境，让插件运行在宿主的模拟进程中。

要做到欺上瞒下，hook 技术不可缺。hook 这个词是我从 Window 安全技术中借用而来的，这实际上是一种函数拦截技术。在某个函数调用流程中插入我们自己的函数，实现对目标函数的参数、返回值的修改。比如某个函数调用流程为：a 调用 b、b 调用 c，那么我们可以动态拦截对 b 函数的调用，插入我们自己的函数 h，修改后的调用流程为：a 调用 h，h 调用 b，b 调用 c，因为 h 是我们新插入的函数，由我们自己编写逻辑，那么在 h 函数中调用 b 的时候，我们就可以修改其参数、返回值，甚至可以中断调用流程，不调用 b 函数而伪造一个返回结果给 a。在此过程中，a 是不知道它调用的 b 函数已经被我们修改了。这就达到了我们“欺上瞒下”的目的。

hook 技术的思想非常类似于设计模式中的代理模式和 Java Spring 框架中的 AOP（Aspect Oriented Programming，面向切面编程）。尽管它们的实现原理完全不同，但目的却差不多。DroidPlugin 中的 Hook 技术实际上是使用了 Java 中的动态代理技术，它只是在一些关键点通过动态代理生成的对象替换掉系统原来的对象，从而完成了对系统通信的 hook。

这其中最关键的是对 AMS（Activity Manage Service）和 PMS(Package Manage Service）以 及 Handler 的 hook。AMS 负 责 管 理 Android 中 Activity、Service、Content Provider、Broadcast 四大组件的生命周期以及各种调度工作，我们 hook 它可以实现对插件四大组件的管理及调度；PMS 负责管理系统中安装的所有 App，我们 hook 它是为了让插件以为自己已经被安装；Handler 是系统向插件传递消息的一个桥梁，我们 hook 它则是为了把系统发向宿主的消息转发给插件。至于在 DroidPlugin 中看到的对其他系统服务的 hook，在大多数情况下都是为了“欺上”——让系统感知不到插件的运行。DroidPlugin 还实现了一个简单的 AMS 服务、一个 PMS 服务，将 hook 后的 AMS 和 PMS 系统调用转发到我们自己的 AMS 和 PMS 服务中去，由 DroidPlugin 自己管理。

除此之外，DroidPlugin 的另外一个特色是“占位”操作。众所周知，在 Android 四大组件中除了 BroadcastReceiver 之外，其他如 Activity 三大组件都需要在 AndroidManifest.xml 中注册，这是静态的，是 Android 框架要求我们预先写死的，我们没有办法动态地向系统注册一个 Activity 或者是 Service。所以插件中的 Activity、Service、Content Provider 则是不可能向系统真正注册。所以我们使用了“占位”的技术，也就是说先在宿主中注册很多的“坑位”，比如对于 Activity 来说，就是 Activity1、Activity2、Activity3 等。在我们需要启动某插件 Activity 的时候，可以通过 hook 技术，将其替换为某个坑位，如 Activity1，

让系统服务去启动坑位 Activity；而在真正的系统服务 AMS 回调插件进程要求插件进程去启动坑位 Activity 的时候，我们再换回插件的 Activity，这样我们就实现了插件 Activity 的免注册运行。当然，因为 Activity 的 Launch Mode 等各种参数问题，我们还需要做很多的细节工作，才能完美。

为了某种完美，DroidPlugin 中做了大量的适配工作，这让其初看起来复杂又臃肿，但是当了解到其中的原理和关键代码后，你会发现如果仅仅是满足“插件化”需求的话，那么其中很多适配并不必需。从现在来看，DroidPlugin 实际上算是一个用户态虚拟机的雏形，从插件化的角度来说则有些“重”了。但是它对于大家深入研究 Android 技术本身，则或多或少会有些帮助。

现在市面上也有各种各样的开源插件化框架，其中很多都已经在各大公司自己的产品中长期稳定使用，满足了各种现实的需求，它们的稳定性、可用性都还是不错的。包老师在本书中也对其中很多插件进行了介绍剖析。

如果我们不满足于业务研发，希望可以了解一些 Android 底层知识，研读这些开源框架的源码则大有裨益。当然，包老师的这本书是国内第一本介绍插件化技术的书籍，作为我们学习插件化技术的入门书籍，则相当合适。

张勇，2018 年 6 月于北京

序　二 Preface

很荣幸能为本书写下三言两语。

我和插件化技术之间有着难解的情缘，到目前为止我已经工作好几年了，如果简化一下，就是下面这个样子：

1）开源 dynamic-load-apk 插件化框架（业余时间）。

2）开发百度手机卫士 & 出版《Android 开发艺术探索》(百度)。

3）开源 VirtualAPK 插件化框架（滴滴出行）。

这么一看，自从我工作以来，有一半时间都在从事插件化相关的开发工作。我喜欢 Android，也喜欢插件化。最开始我对插件化只是兴趣使然，在工作之余我喜欢做一些研究，所以有了 dynamic-load-apk。到后面我加入滴滴出行则是使命使然，在我的内心深处，我觉得 dynamic-load-apk 不够完美，我想开发一款完美的插件化框架，于是有了后面的 VirtualAPK。回想起插件化的发展，也就仿佛看到了一路走来的自己。

2014 年 3 月 30 号，我在 CSDN 上发布了一篇文章《Android apk 动态加载机制的研究》。这篇文章现在看起来很傻，技术也很落后，但是很多人都无法感受当时的情形。在 2014 年年初，别说插件化知识了，就连高质量的 Android 技术文章都比较匮乏。

说起高质量的 Android 技术文章，大家可能会想到：可以看《第一行代码 Android》和《Android 开发艺术探索》呀！但是很遗憾，那个时候这两本书都还没出版，不止是这两本书，很多大家所熟知的书都没有出版。当时 Android 技术圈沉醉于下拉刷新、侧滑菜单等这种炫酷特效，对于 AIDL 和 View 原理不曾研究过，你要问插件化？ 90% 的 Android 工程师都不知道这是个什么东西。除了技术文章和书籍比较匮乏以外，开源也比较匮乏。在 2014 年，插件化技术只是一个概念，虽然当时阿里已经有了 Atlas，但是并没有开源，所以在这种情形下，我发的那篇文章就显得很专业了，当时引起了技术圈的广泛关注，获得了 7 万多的阅读量。

在 2014 年下半年，我和田啸、宋思宇等同学发起了 dynamic-load-apk 这个开源项目，现在大家都知道了，dynamic-load-apk 在插件化历史中有着浓厚的一笔。dynamic-load-apk

支持动态加载代码和资源，资源访问可以直接通过 R 来进行，在四大组件方面支持 Activity 和 Service。虽然说 dynamic-load-apk 谈不上多完善，但是业内却有不少公司基于 dynamic-load-apk 进行二次开发来定制自己的插件化框架，从这个角度来说 dynamic-load-apk 是很成功的一款开源框架。

到 2016 年，插件化框架才真正迎来了大繁荣时代，可谓百家争鸣。不管是 Atlas、Small、DroidPlugin 还是携程的 DynamicAPK，都极大地促进了 Android 插件化框架的发展。我也是在 2016 年年初离开百度，来到了现在的滴滴出行。如果说在百度的工作是做应用开发的话，那么在滴滴出行的工作就完全是热修复和插件化开发了。经过长时间的开发和验证，滴滴出行在 2017 年 6 月 30 日开源了一个更为完善的插件化框架 VirtualAPK，而我则在这个框架的开源中发挥了至关重要的作用。

如果给我这几年的职业生涯写一个总结，那就是：两款 Android 插件化框架 + 一个 App+ 一本书，而我还将在 Android 的道路上继续耕耘。

任玉刚，2018 年 6 月于北京

序　三 Preface

听说包猪猪的第二本书要出版了，很为他高兴，作为一个旁观者，眼见着本书由一个想法萌芽逐渐充实，颇有感触。我脑海里就像过电影一般，浮现这几个月中包猪猪的种种状态：为解决一个 bug 而连续工作十几个小时的苦闷，第二天一早起来灵感触发迎刃而解的喜悦，一边照顾父母一边因书的进度被某一难题阻滞而焦灼……身边的人会跟他说："事情这么多，歇歇再写吧，别让自己太辛苦。"可是一个想快点跟业界分享自己想法和成果的程序员又怎会因琐事耽搁？曾是微软 MVP 的他，入行十几年，仍然秉承着刚入行时的激情与热忱，吭哧吭哧地调 bug 到凌晨，这本书或许是这个"内心有团火"的家伙希望送给读者最好的礼物——智慧的分享与教学相长的领悟。

我是学中文和法律的，作为圈外人在本书前作序，多少有班门弄斧之嫌。如论代码，各位读者在自己的领域各有所长。不过我这种对技术一窍不通的人，竟在看一本技术书时捧腹大笑，足见本书的吸引力。比如说，书中调侃张勇和任玉刚是插件化领域的男乔峰女慕容，以至于我一直想亲眼见一下这两个人。比如说他在前言中拿娃娃开涮，连着感谢了霹雳娇娃、赵越和 Dinosaur，殊不知那却是一个人，可能是为了看上去人多一些，以壮声势。

程序员的世界我不太懂，尤其是包猪猪，他经常做出一些令人啼笑皆非的事情。比如说，情人节他第一次给我送花，却阴差阳错地寄来了两束。晚上的时候他跟我说，两束花别浪费了，另一束花就快递给邓凡平吧——据说那也是 Android 行业内的一位大神，他因为伺候老婆坐月子而忘记准备情人节礼物了。于是，作为情人节只收别人礼物的我，在这一天，第一次给别人送礼物。

他做饭很好吃，有很多招牌菜。用他的话讲，炒菜是设计模式中的装饰器模式。比如说西红柿炒鸡蛋，放锅里炒了几分钟后，加点糖，就是味道甜甜的西红柿炒鸡蛋，再加些盐，就是酸甜可口的西红柿炒鸡蛋，也许还会再加些其他调料，但这道菜永远都是西红柿炒鸡蛋，只是味道不同罢了。技术做到这一步已经接近于完美，但他后面的奇葩行为，却颠覆了我对他的认知。

他研究做鱼，第一天没做好，第二天再买条鱼继续做，直到他认为完美。把钻研计算

机技术的执着用于烹饪，结果就是我一连吃了 5 天鱼汤，上火，满嘴都是泡。

第一次见包猪猪是在酒吧，一边听着不知道是哪里的古老而嘈杂的乐队嘶吼，另一边是他给我讲关于五个海盗分赃的小故事。隐约记得故事是这样的：“5 个海盗抢到 100 个金币，他们决定依次由 A,B,C,D,E 五个海盗来分，他们订立了如下规则：当由 A 提分配方案时，剩下的海盗表决，如果 B,C,D,E 四人中有一半以上反对就把 A 扔下海，再由 B 分……如是这般，那么 A 海盗如何分，才能既保住性命又能获得更多的金币。”这是个有意思的小故事，轻松而又暗藏思维逻辑，我们可能有很多种解决方案，但是最优方案最后一定是“博弈与制衡”的平衡。此前，我一直以为程序员的世界充斥着代码，那是一套拥有独立计算机语言的系统，难以接近跟理解。可是包猪猪有种能力将难以理解或者比较复杂的事情用一种有趣的方式表达出来，诙谐有趣、通俗易懂又能在嘻嘻哈哈的氛围中有所感悟。

本书讲插件化，从插件化的历史讲起，说了不少这行的人跟事，还有八卦。接下来由基础知识开始讲起，后来又介绍了插件化解决方案及周边技术。文字所限，本书内容结构不赘述了，各位可以凭目录了解。代码方面我虽然读不懂，不过并不影响看书的心情，这是本书颇为神奇的地方。没有高深莫测的理论，没有艰深难懂的词汇，就像包猪猪站在我面前娓娓道来，举一些他觉得有意思的例子，让你觉得调皮又生动。对于对插件化不熟悉的读者，可能本书提及的有些词汇是新的，不太容易理解，感谢本书的编辑在成书过程中从旁指引，因而本书在前言部分增加了名词解释，并在各个章节多加了一些描述性的文字。

关于本书前言所提及未展开描述的部分，其实作者在写作前已经预留了位置，但是在成书前两天犹豫再三还是有所删减，因为总觉得讲得不透、不彻底是不好意思呈现给读者看的。因此，我跟很多读者一样很期待包猪猪过段时间能将那些本书没说透的东西再写本书好好讲一讲。比如说 Small，他半夜说梦话时经常念叨这个词。

谨以此书献给奋斗在一线的程序员们，作为家属深刻了解程序员的辛酸，希望本书对读者能够有所启发，运用到工作中可以提高效率，多一些休息时间陪伴家人、朋友，少一些熬夜加班、拼命赶工。

郭旻云，2018 年 6 月于北京

前　言 *Preface*

这是一本什么书

如果只把本书当作纯粹介绍 Android 插件化技术的书那就错了。本书在研究 Android 插件化之余，还详细介绍了 Android 系统的底层知识，包括 Binder 和 AIDL 的原理、四大组件的原理、App 的安装和启动流程、Context 和 ClassLoader 的家族史。没有罗列大量的 Android 系统中的源码，而是以一张张 UML 图把这些知识串起来。

本书详细介绍了 Android 中的资源机制，包括 aapt 命令的原理、resource 文件的组成以及 public.xml 的使用方式，顺带还提及了如何自定义一个 Gradle 插件化。

此外，本书还介绍了 so 的加载原理，尤其是动态加载 so 的技术，可以帮助 App 进行瘦身；探讨了 HTML5 降级技术，可以实现任何一个原生页面和 HTML5 页面的互换；介绍了反射技术，以及 jOOR 这个有趣的开源框架；介绍了 Android 中的动态代理技术 Proxy.newProxyInstance 方法。

如果读者能坚持把这本书从头到尾读完，那么不仅掌握了插件化技术，而且也把上述所有这些知识点全都系统地学习了一遍。也许 Android 插件化会随着 Google 的限制而有所变化甚至消亡，但我在本书中介绍的其他知识，仍然是大有用武之处的。

如何面对 Android P 的限制

写作这本书的时候，Google 推出了 Android P preview 的操作系统，会限制对 @hide api 的反射调用。目前会通过 log 发出警告，用户代码仍然能够获取到正确的 Method 或 Field，在后续版本中获取到的 Method 或 Field 极有可能为空。

但是道高一尺，魔高一丈。Google 对这次限制，很快就被技术极客们绕过去了[⊖]，有两种解决方法：

⊖ 详细内容，请参见田维术的文章：http://weishu.me/2018/06/07/free-reflection-above-android-p/

1）把通过反射调用的系统内部类改为直接调用。具体操作办法是，在 Android 项目中新建一个库，把要反射的类的方法和字段复制一份到这个库中，App 对这个库的引用关系设置为 provided。那么我们就可以在 App 中直接调用这个类和方法，同时，在编译的时候，又不会把这些类包含到 apk 中。

其实早在 2015 年，hoxkx 就在他的插件化框架中实现了这种技术[⊖]。但是这种解决方案，仅限于 Android 系统中标记为 public 的方法和字段，对于 protected 和 private 就无能为力了。比如 AssetsManager 的 addAssetPath 方法，ActivityThread 的 currentActivityThread 方法。

2）类的每个方法和字段都有一个标记，表明它是不是 hide 类型的。我们只要在 jni 层，把这个标记改为不是 hide 的，就可以绕过检查了。

然而，魔高一丈，道高一丈二。Google 在 Android P 的正式版中势必会推出更严厉的限制方案，到时候，又会有新的解决方案面世，让我们拭目以待。

其实，开发者是无意和 Google 进行技术对抗的，这是毫无意义的。泛滥成灾的修改导致了 App 大量的崩溃，Google 实在看不下去了，所以才搞出这套限制方案；另一方面，插件化技术是刚需，尤其在中国的互联网行业，App 崩溃会直接影响使用，很可能导致经济损失，所以开发者才会不惜一切代价走插件化这条路。

再回到限制方案来，Google 也不是清一色不要开发者使用系统底层的标记为 hide 的 API，而是推出了一组黑灰名单，如下所示：

名　单	影　响
light-greylist 浅灰名单	仅打印警告日志，Google 尽可能在未来版本提供 public API
dark-greylist 深灰名单	第三方 App 不能访问，开发者可以申请把这份清单中的某些 API 加入到浅灰名单
blacklist 黑名单	第三方 App 不能访问

所以，另一种应对策略是，在插件化中使用浅灰名单中的 API，比如说 ActivityThread 的 currentActivityThread 方法。

Google 的这组清单还在持续调整中，据我所知，给各大手机厂商的清单与其在社区中发布的清单略有出入。在 Android P 的正式版本中，这份清单会最终确定下来。所以现在中国的各个插件化框架的开发人员，都在等 Android P 的正式版本发布后再制定相应的策略。留给中国队的时间不多了。

这本书的来龙去脉

这是一本酝酿了 3 年的书。

早在 2015 年 Android 插件化技术百家争鸣时，我就看好这个技术，想写一本书介绍这

⊖ 项目地址参见：https://github.com/houkx/android-pluginmgr

个技术，但当时的积累还不够。那年，我在一场技术大会上发表了《 Android 插件化从入门到放弃》演讲，四十五分钟介绍了插件化技术的皮毛。后来这个演讲内容被整理成文章发布到网上，流传很广。

2017 年 1 月，有企业要我去讲 2 天 Android 插件化技术。为此，我花了一个月时间，准备了四十多个例子。这是我第一次系统地积累了素材。

2017 年 6 月，我在腾讯课堂做 Android 线上培训，为了宣传推广我的课程，我写了一系列文章《写给 Android App 开发人员看的 Android 底层知识》，共 8 篇，没列太多代码，完全以 UML 图的方式向读者普及 Binder、AIDL、四大组件、AMS、PMS 的知识。本书的第 2 章就是在这 8 篇文章的基础之上进行扩充的。

2018 年 1 月，我父亲住院一周。我当时在医院每天晚上值班。老爷子半夜打呼噜，吵得我睡不着，事后我才知道，我睡着了打呼噜声音比他还大。半夜睡不着时就开始了本书的写作，每晚坚持写到凌晨两三点。直到父亲出院，这本书写了将近五分之一。

碰巧的是，这一年 5 月底我结婚，促使我想在 5 月初完成这本书的一稿，为此，我宅在家里整整写了 3 个月。仅以此书作为新婚礼物献给我亲爱的老婆，感谢你的理解，这本书才得以面世。

这两年我在忙什么

2016 年 5 月写完《 App 研发录》后不久，我就从一线互联网公司出来，开始了长达两年的 App 技术培训工作。一改之前十几年在办公室闷头研究技术的工作方式，开始在全中国飞来飞去，给各大国企、传统公司、手机商的 Android 和 iOS 团队进行培训。这两年去过了近百家公司，有了很多与过去十几年不一样的感受。

App 开发人员的分布也呈金字塔型，在金字塔尖的自然是那些一线互联网的开发人员，他们掌握 Android 和 iOS 最先进的技术，比如组件化、插件化等技术，但这些人毕竟是少数。而位于金字塔底端的开发人员则是大多数，他们大都位于创业公司或者传统行业，相应的 App 侧重于业务的实现，对 App 的高端技术，用得不多，需要不断补充新知识。另外，我在腾讯课堂讲了几个月 App 开发课程的过程中，认识了很多学员，有几千粉丝，同样面临需要不断学习新知识。

写作这本书的目的，是向广大 Android 开发人员普及插件化技术。

这本书里讲些什么

战战兢兢写下这本书，有十几万字，仍不能覆盖 Android 插件化的所有技术。因为插件化技术千头万绪，流派众多，我想从最基本的原理讲起，配合大量的例子，希望能帮助完全不懂 Android 插件化技术的小白，升级为一个精通这类技术的高手。

面对业内各种成熟的插件化框架，我只选取了具有代表意义的DroidPlugin、DL、Small和Zeus进行介绍，这几个框架基本覆盖了插件化编程的所有思想，而且非常简单，像Zeus只有11个类，就支撑起掌阅App的插件化。而对于后期推出的VirtualApk、Atlas、Replugin等，在本书中并没有介绍，主要是因为这些框架都是大块头，代码量很多，我没有精力再去研究和学习了。但这些企业级插件化框架所用的技术，本书都有涉及。

本书的结构及内容

全书分为三大部分，共22章。第1部分“预备知识”包括第1～5章，是进行Android插件化编程的准备知识。第2部分“解决方案”包括第6～16章，详细介绍并分析了插件化编程的各种解决方案。第3部分“相关技术”包括第17～21章，介绍插件化编程的周边技术，并对纷繁复杂的插件化技术进行了总结。

第1章介绍的是Android插件化的历史，可以当作小说来读，茶余饭后，地铁站中就可以读完。

第2章介绍Android底层知识，涉及那些与Android插件化相关的知识，比如Binder和AIDL，Android App的安装流程和启动流程，ActivityThread，LoadedApk，Android四大组件的运行原理。这一章篇幅较多，需要仔细研读。其中，讲到一个音乐播放器的例子，帮助大家更加深刻地认识Android的四大组件。

第3章讲反射，详细介绍了构造函数、方法、字段、泛型的反射语法。这章介绍了Java领域很火的一个开源库jOOR，可惜，它对Android的支持并不是很好，所以这章还介绍了我们自己封装的RefInvoke类，这个类将贯穿本书，基本上所有源码例子都会使用到它。

第4章讲代理模式。这个模式在Android中最著名的实现就是Proxy.newProxyInstance方法。基于此，我们Hook了AMS和PMS中的一些方法。

第5章是第4章的延续，仍然是基于Proxy.newProxyInstance方法，Hook了Activity的启动流程，从而可以启动一个没有在AndroidManifest中声明的Activity，这是插件化的核心技术之一。

第6章介绍了如何加载插件App，以及如何对插件化项目的宿主App和插件App同时进行调试。说到插件化编程，离不开面向接口编程的思想，这章也花了很多笔墨介绍这个思想，以及具体的代码实现。

第7章介绍了资源的加载机制，包括AssetManager和Resources，并给出了资源的插件化解决方案，从而为Activity的插件化铺平了道路。另外还介绍了换肤技术的插件化实现。

第8章介绍了最简单的插件化解决方案，通过在宿主App的AndroidManifest中事先声明插件中的四大组件。为了能让宿主App随意加载插件的类，这章介绍了合并dex的技术方案。

第9章到第12章介绍了Android四大组件的插件化解决方案。四大组件的生命周期各不相同，所以它们各自的插件化解决方案也都不同。

第 13 章、第 14 章介绍了 Android 插件化的静态代理的解决方案。这是一种“牵线木偶”的思想，我们不用 Hook 太多 Android 系统底层的代码。

第 15 章再次讲到资源，这次要解决的是宿主和多个插件的资源 id 值冲突的问题。这一章介绍了多种解决方案，包括思路分析、代码示例。

第 16 章介绍一种古老的插件化解决方案，通过动态替换 Fragment 的方式。

第 17 章介绍了 App 的降级解决方案。一旦插件化方案不可用，那么我们仍然可以使用 H5，来替换任何一个 App 原生页面。

第 18 章介绍了插件的混淆技术。有时候宿主 App 和插件 App 都会引用 MyPluginLibrary 这个类库，这个公用类库是否要混淆，相应有两种不同的混淆方案。

第 19 章介绍了增量更新技术。这是插件化必备的技术，从而保证插件的升级，不需要从服务器下载太大的包。

第 20 章介绍了 so 的插件化解决方案。这章详细介绍了 so 的加载原理，以及从服务器动态加载 so 的方案，基于此，有两种 so 的插件化解决方案。

第 21 章介绍了 gradle-small 这个自定义 Gradle 插件。这章是对第 15 章的补充，是另一种解决插件资源 id 冲突的方案。

第 22 章作为整本书的结尾，系统总结了 Android 插件化的各种解决方案。如果读者能坚持读到这最后一章，可以帮助读者巩固这些知识。

关于本书名词解释

本书有很多专业术语，刚接触 Android 插件化的读者可能不容易理解。有一些专业术语还有别称或者简称，我在这里罗列出最常见的一些术语：

- HostApp，本书中有时也写作“宿主 App”。用于承载各种插件 App，是最终发版的 App。我们从 Android 市场上下载的都是 HostApp。
- Plugin，本书中有时也写作“插件”、“插件 App”。
- Receiver，是 BroadcastReceiver 的简称，Android 四大组件之一。
- AndroidManifest，也就是 AndroidManifest.xml。
- Hook，就是使用反射修改 Android 系统底层的方法和字段。
- AMS，是 ActivityManagerService 的简称，在 App 运行时和四大组件进行通信。
- PMS，是 PackageManagerService 的简称，用于 App 的安装和解析。

关于本书的源码

本书精心挑选了 70 多个例子，都可以直接下载使用，正文中都列出了代码名称，在相应网站可以找到。附录 B 还列出了所有源码对应的章节。

致谢

几乎所有的书都千篇一律地感谢很多人，却不写为什么要感谢他们。我在这里一定要把感谢的理由说清楚。

首先，感谢我那古灵精怪的老婆郭曼云。谢谢她在我人生迷惘的时候及时出现，陪我玩王者荣耀，带我骑小黄车去散心，看电影时一起八卦剧情然后被坐在旁边的观众出声制止。她每天要我做不一样的饭菜给她吃，把我锻炼成厨房小能手，我现在已经习惯于每天傍晚五点半就放下手中所有的活儿，愉快地投入买菜做饭的工作。

感谢张勇、任玉刚、罗迪、黄剑、林光亮、邓凡平、王尧波、田维术等 Android 领域的朋友们，我在写作这本书的时候，经常会遇到各种疑惑，每次问到他们，他们都会不厌其烦地给我详细解答。

在这里，尤其感谢田维术，他的技术博客（weishu.me）“介绍插件化的一系列文章”对我的影响很大，可惜没写完，只讲了 Binder 原理和四大组件的插件化方案。本书的部分章节参考了他的博客文章，对他提供的一些代码例子进行了二次加工。经过他本人同意后，收入这本书中。代码中的很多类上都标注了作者是 weishu，以表达对他的感谢。

感谢任正浩、霹雳娇娃、赵越、Dinosaur 等这群狐朋狗友的陪伴，我从互联网公司出来转型做 App 技术培训的过程中，整理了半年 PPT 教程后才开始陆陆续续接到单子，在这半年时间里，我就跟这帮学弟学妹厮混在一起，爬山、撕名牌、唱 K、密室逃脱、狼人杀，还有一阵时间沉迷于你画我猜。那是我最惬意的一段时光。

感谢我爸我妈以及咱爸咱妈，你们的女儿我一定照顾好。虽然北京与天津距离那么近，很抱歉还是不能常回家看看，我永远是那么忙，忙着去追求事业的成功，距离财务自由还很远，但是我一直在努力。

最后，感谢曹洪伟等 21 位社区朋友的辛勤劳动，把这本书翻译为英文，限于篇幅，这里就不一一致谢了。接下来，这本书的英文版本会在国外网站社区逐篇发布，乃至出版成书，让全世界的 Android 开发人员看到中国工程师们的智慧结晶。

包建强

2018 年 6 月于北京

目　录 Contents

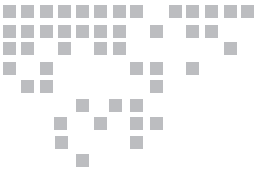
第二部分 解决方案

第三部分　相关技术

附录

第一部分 Part 1

预备知识



Chapter 1 第 1 章

插件化技术的昨天、今天与明天

这是最好的时代，国内各大应用市场对插件化技术的态度是开放的，因此，国内各大互联网 App 无一不有自己的插件化框架。有了开放的环境，才会有无数英雄大展身手，在 Android 插件化的领域中出现百家争鸣、百花齐放的局面。

这是最坏的时代，随着插件化技术在中国的普及，你会发现，去中国的各大互联网公司面试，一般都会聊聊插件化的技术。这就使得开发人员要去了解 Android 底层的知识，这无形中增加了学习难度。

本章将介绍插件化的概念、历史及应用，为后续学习插件化技术，提供基础。

1.1 插件化技术是什么

一个游戏平台，比如联众，支持上百种游戏，如象棋、桥牌、80 分。一个包括所有游戏的游戏平台往往有上百兆的体积，需要下载很长时间，但是用户往往只玩其中的 1～2 款游戏。让用户下载并不会去玩的上百款游戏，是不明智的做法。此外，任何一个游戏更新或者新上线一个游戏，都需要重新下载数百兆的安装包，也会让用户抓狂。

所以，游戏平台必然采用插件化技术。

通常的做法是，只让用户下载一个十几兆大小的安装包，其中只包括游戏大厅和一个全民类游戏，如“斗地主”。用户进入游戏大厅，可以看到游戏清单，点击“80 分”就下载 80 分的游戏插件，点击“中国象棋”就下载中国象棋的游戏插件，这称为“按需下载”。这就需要插件化编程，不过这是基于电脑上的游戏平台，是一个个 exe 可执行文件。

在 Android 领域，是没有 exe 这种文件的。所有的文件都会被压缩成一个 apk 文件，然后下载到本地。Android 应用中所谓的安装 App 的过程，其实就是把 apk 放到本地的一个

目录，然后使用 PMS 读取其中的权限信息和四大组件信息。所以 Android 领域的插件化编程，与电脑上的软件的插件化编程是不一样的。

其实，在 Android 领域，对于游戏而言，用的还真不是插件化技术，而是从服务器动态下发脚本，根据脚本中的信息，修改人物属性，增加道具和地图。

Android 插件化技术，主要用在新闻、电商、阅读、出行、视频、音乐等领域，比如说旅游类 App，因为涉及下单和支付，所以算是电商的一个分支，旅游类 App 可以拆分出酒店、机票、火车票等完全独立的插件。

1.2　为什么需要插件化

在那山的这边海的那边有一群程序员，
他们老实又腼腆，
他们聪明又有钱。
他们一天到晚坐在那里熬夜写软件，
饿了就咬一口方便面。
哦苦命的程序员，
哦苦命的程序员，
只要一改需求他们就要重新搞一遍，
但是期限只剩下两天。

这首改编自《蓝精灵》主题曲的《程序员之歌》，道出了中国互联网行业的程序员现状。

就在 Android 程序员疯狂编写新需求之际，自然会衍生出各种 bug，甚至是崩溃。App 有 bug，会导致用户下不了单，而一旦崩溃，那就连下单页面都进不去，因此我们要在最短时间内修复这些问题，重新发版到 Android 各大市场已经来不及，每分每秒都在丢失生意，因此，Android 插件化的意义就体现出来了，不需要用户重新下载 App，分分钟就能享受到插件新的版本。

另一方面，如果要和竞争对手抢占市场，那么谁发布新功能越快越多，对市场对用户的占有率就越高。如果隔三岔五就发布一个新版本到 Android 各大市场，那么用户会不胜其烦，发版周期固定为半个月，又会导致新功能长期积压，半个月后才能让用户见到，而竞争对手早就让用户在使用同样的新功能了。这时候，如果有插件化技术支持，那么新功能就可以在做完之后立刻让用户看到，这可是让竞争对手闻风丧胆的手段。

1.3　插件化技术的历史

2012 年 7 月 27 日，是 Android 插件化技术的第一个里程碑。大众点评的屠毅敏

（Github 名为 mmin18），发布了第一个 Android 插件化开源项目 AndroidDynamicLoader㊀，大众点评的 App 就是基于这个框架来实现的。这是基于 Fragment 来实现的一个插件化框架。通过动态加载插件中的 Fragement，来实现页面的切换，而 Activity 作为 Fragement 的容器却只有一个。我们也是在这个开源项目中第一次看到了如何通过反射调用 AssetManger 的 addAssetPath 方法来处理插件中的资源。

2013 年，出现了 23Code。23Code 提供了一个壳，在这个壳里可以动态下载插件，然后动态运行。我们可以在壳外编写各种各样的控件，在这个框架下运行。这就是 Android 插件化技术。这个项目的作者和开源地址，我不是很清楚，如果作者恰巧读到我这本书，请联系我，咱们一起喝杯咖啡。

2013 年 3 月 27 日，第 16 期阿里技术沙龙，淘宝客户端的伯奎做了一个技术分享，专门讲淘宝的 Atlas 插件化框架，包括 ActivityThread 那几个类的 Hook、增量更新、降级、兼容等技术。这个视频㊁，只是从宏观来讲插件化，具体怎么实现的并没说，更没有开源项目可以参考。时隔 5 年再看这个视频，会觉得很简单，但在 2013 年，这个思想还是很先进的，毕竟那时的我还处在 Android 入门阶段。

2014 年 3 月 30 日 8 点 20 分，是 Android 插件化的第二个里程碑。任玉刚开源了一个 Android 插件化项目 dynamic-load-apk㊂，这与后续介绍的很多插件化项目都不太一样，它没有对 Android 系统的底层方法进行 Hook，而是从上层，也就是 App 应用层解决问题——通过创建一个 ProxyActivity 类，由它来进行分发，启动相应的插件 Activity。因为任玉刚在这个框架中发明了一个 that 关键字，所以我在本书中把它称为 that 框架。其实作者不喜欢我给他的最爱起的这个外号，他一直称之为 DL。曾经和玉刚在一起吃饭聊天，他感慨当年如何举步维艰地开发这个框架，因为 2014 年之前没有太多的插件化技术资料可以参考。

that 框架一开始只有 Activity 的插件化实现，后来随着田啸和宋思宇的加入，实现了 Service 的插件化。2015 年 4 月 that 框架趋于稳定。那时我在途牛做 App 技术总监，无意中看到这个框架，毅然决定在途牛的 App 中引入 that 框架。具体实施的是汪亮亮和魏正斌，他们当时一个初为人父另一个即将为人父，还是咬牙把这个 that 框架移植到了途牛 App 中。that 框架经受住了千万级日活 App 的考验，这是它落地的第一个实际项目㊃。

与此同时，张涛也在研究插件化技术㊄，并于 2014 年 5 月发布了他的第一个插件化框架 CJFrameForAndroid㊅。它的设计思想和 that 框架差不多，只是把 ProxyActivity 和 ProxyService

㊀ 开源项目地址：https://github.com/mmin18/AndroidDynamicLoader
㊁ 视频地址：http://v.youku.com/v_show/id_XNTMzMjYzMzM2.html
㊂ 开源项目地址：https://github.com/singwhatiwanna/dynamic-load-apk
㊃ 参考文章：https://blog.csdn.net/lostinai/article/details/50496976
㊄ 张涛的开源实验室：https://kymjs.com
㊅ 开源项目地址：https://github.com/kymjs/CJFrameForAndroid

称为托管所。此外，CJFrameFor-Android 框架还给出了 Activity 的 LaunchMode 的解决方案，这是对插件化框架一个很重要的贡献，可以直接移植到 that 框架中。

2014 年 11 月，houkx 在 GitHub 上发布了插件化项目 android-pluginmgr ㊀，这个框架最早提出在 AndroidManifest 文件中注册一个 StubActivity 来“欺骗 AMS”，实际上却打开插件中的 ActivityA。但是作者并没有使用对 Instrumnetation 和 ActivityThread 的技术进行 Hook，而是通过 dexmaker.jar 这个工具动态生成 StubActivity，StubActivity 类继承自插件中的 ActivityA。现在看来，这种动态生成类的思想并不适用于插件化，但在当时能走到这一步已经很不容易了。

同时，houkx 还发现，在插件中申请的权限无法生效，所以要事先在宿主 App 中申请所有的权限。android-pluginmgr 有两个分支——dev 分支和 master 分支。作者的插件化思想位于 dev 分支。后来高中生 Lody 参与了这个开源项目，把 android-pluginmgr 设计为对 Instrumnetation 的思想进行 Hook，体现在 master 分支上，但这已是 2015 年 11 月的事情了。

2014 年 12 月 8 日有一个好消息，那就是 Android Studio1.0 版本出现了。Android 开发人员开始逐步抛弃 Eclipse，而投入 Android Studio 的怀抱。Android Studio 借助于 Gradle 来编译和打包，这就使插件化框架的设计变得简单了许多，排除了之前 Eclipse 还要使用 Ant 来运行 Android SDK 的各种不便。

时间到了 2015 年。高中生 Lody 此刻还是高二学生。他是从初中开始研究 Android 系统源码的。他的第一个著名的开源项目是 TurboDex ㊁，能够以迅雷不及掩耳之势加载 dex，这在插件化框架中尤其好用，因为首次加载所有的插件需要花很久的时间。

2015 年 3 月底，Lody 发布插件化项目 Direct-Load-apk ㊂。这个框架结合了任玉刚的 that 框架的静态代理思想和 Houkx 的 pluginmgr 框架的“欺骗 AMS”的思想，并 Hook 了 Instrumnetation。可惜 Lody 当时还是个学生，没有花大力气宣传这个框架，以至于没有太多的人知道这个框架的存在。Lody 的传奇还没结束，后来他投身于 VirtualApp，这是一个 App，相当于 Android 系统之上的虚机，这是一个更深入的技术领域，我们稍后再提及。

2015 年 5 月，limpoxe 发布插件化框架 Android-Plugin-Framework ㊃。

2015 年 7 月，kaedea 发布插件化框架 android-dynamical-loading ㊄。

2015 年 8 月 27 日，是 Android 插件化技术的第三个里程碑。张勇的 DroidPlugin 问世了。张勇当时在 360 的手机助手团队，DroidPlugin 就是手机助手使用的插件化框架。这个框架的神奇在于，能把任意的 App 都加载到宿主里面去。你可以基于这个框架写一个宿主 App，然后就可以把别人写的 App 都当作插件来加载。

㊀ 开源项目地址：https://github.com/houkx/android-pluginmgr
㊁ 开源项目地址：https://github.com/asLody/TurboDex
㊂ 开源项目地址：http://git.oschina.net/oycocean/Direct-Load-apk
㊃ 开源项目地址：https://github.com/limpoxe/Android-Plugin-Framework
㊄ 开源项目地址：https://github.com/kaedea/android-dynamical-loading

DroidPlugin 的功能很强大，但强大的代价就是要 Hook 很多 Android 系统的底层代码，而且张勇没有给 DroidPlugin 项目加任何说明文档，导致这个框架不太容易理解。网上有很多人写文章研究 DroidPlugin，但其中写得最好的是田维术[⊖]。他当时就在 360，刚刚毕业转正，写出一系列介绍 DroidPlugin 思想的文章，包括 Binder 和 AIDL 的原理、Hook 机制、四大组件的插件化机制等。

2015 年是 Android 插件化蓬勃发展的一年，不光有 that 框架和 DroidPlugin，很多插件化框架也在这个时候诞生。

OpenAtlas 这个项目是 2015 年 5 月发布在 Github 上的，后来改名为 ACDD。里面提出了通过修改并重新生成 aapt 命令，使得插件 apk 的资源 id 不再是固定的 0x7f，可以修改为 0x71 这样的值。这就解决了把插件资源合并到宿主 HostApp 资源后资源 id 冲突的问题。

OpenAtlas 也是基于 Hook Android 底层 Instrumentation 的 execStartActivity 方法，来实现 Activity 的插件化。此外，OpenAltas 还 Hook 了 ContextWrapper，在其中重写了 getResource 等方法，因为 Activity 是 ContextWrapper 的“孙子”，所以插件 Activity 就会继承这些 getResource 方法，从而取到插件中的资源——这种做法现在已经不用了，我们是通过为插件 Activity 创建一个基类 BasePluginActivity 并在其中重写 getResource 方法来实现插件资源加载的。

携程于 2015 年 10 月开源了他们的插件化框架 DynamicAPK[⊜]，这是基于 OpenAltas 框架基础之上，融入了携程自己特殊的业务逻辑。

2015 年 12 月底，林光亮的 Small 框架发布，他当时在福建一家二手车交易平台，这个框架是为这个二手车平台的 App 量身打造的，主要特点如下：

- Small 把插件的 ClassLoader 对应的 dex 文件，塞入到宿主 HostApp 的 ClassLoader 中，从而 HostApp 可以加载任意插件的任意类。
- Small 框架通过 Hook Instrumentation 来启动插件的 Activity，这一点和 DroidPlugin 相同，那么自然也会在 AndroidManifest 中声明一个 StubActivity，来“欺骗 AMS”。
- Small 框架对其他三大组件的支持，需要提前在宿主 HostApp 的 AndroidManifest 中声明插件的 Service、Receiver 和 ContentProvider。
- Small 对资源的解决方案独树一帜。使用 AssetManager 的 addAssetPath 方法，把所有插件的资源都合并到宿主的资源中，这时候就会发生资源 id 冲突的问题。Small 没有采用 Atlas 修改 AAPT 的思路，而是在生成插件 R.java 和 resources.arsc 这两个文件后，把插件 R.java 中所有资源的前缀从 0x7f 改为 0x71 这样的值，同时也把 resources.arsc 中出现 0x7f 的地方也改为 0x71。

⊖ 田维术的技术博客：http://weishu.me

⊜ 开源项目地址：https://github.com/CtripMobile/DynamicAPK

随着 2015 年的落幕，插件化技术所涉及的各种技术难点都已经有了一种甚至多种解决方案。在这一年，插件化技术领域呈现了百家争鸣的繁荣态势。这一时期以个人主导发明的插件化框架为主，基本上分为两类——以张勇的 DroidPlugin 为代表的动态替换方案，以任玉刚的 that 框架为代表的静态代理方案。

就在 2015 年，Android 热修复技术和 React Native 开始进入开发者的视线，与 Android 插件化技术平分秋色。Android 插件化技术不再是开发人员唯一的选择。

从 2016 年起，国内各大互联网公司陆续开源了自己研发的插件化框架。这时候已经没有什么新技术出现了，因为插件化所有的解决方案都已经在 2015 年由个人开发者给出来了。互联网公司是验证这些插件化技术是否可行的最好的平台，因为他们的 App 拥有动辄千万用户的日活。

按时间顺序列举插件化框架如下：

2016 年 8 月，掌阅推出 Zeus ㊀。

2017 年 3 月，阿里推出 Atlas ㊁。

2017 年 6 月 26 日，360 手机卫士的 RePlugin ㊂。

2017 年 6 月 29 日，滴滴推出 VisualApk ㊃。

仔细读这些框架的源码会发现，互联网公司开源的这些框架更多关注于：

- ❑ 插件的兼容性，包括 Android 系统的升级对插件化框架的影响，各个手机 ROM 的不同而对插件化的影响。
- ❑ 插件的稳定性，比如各种奇葩的崩溃。
- ❑ 对插件的管理，包括安装和卸载。

斗转星移，时光荏苒，虽然只有几年时间，但各个插件化框架已经渐趋稳定，现在做插件化技术的开发人员，只需要关注每年 Android 系统版本的升级对自身框架的影响，以及如果消除这种影响。

随着插件化领域的技术基本成型，我身边很多做插件化的朋友都开始转行，有的人还在 Android 这个领域，比如张勇基于他的 DroidPlugin 框架，在做他的创业项目闪电盒子；有的人转入区块链，每天沉浸于用 GO 语言写智能合约。

谨以此文献给那些在插件化领域中做出过贡献的朋友们，包括开源框架的作者，以及写文章传经布道的作者。我的见识有限，有些人、有些框架、有些文章可能会没有提及，欢迎广大读者多多指正。

㊀ 开源项目地址：https://github.com/iReaderAndroid/ZeusPlugin

㊁ 开源项目地址：https://github.com/alibaba/atlas

㊂ 开源项目地址：https://github.com/Qihoo360/RePlugin

㊃ 开源项目地址：https://github.com/didi/VirtualAPK

1.4 插件化技术的用途到底是什么

我们曾经天真地认为，Android 插件化是为了增加新功能，或者增加一个完整的模块。费了不少时间和精力，等项目实施了插件化后，我们才发现，插件化 80% 的使用场景，是为了修复线上 bug。在这一点上，插件化与 Tinker、Robust 这类热修复工具拥有相同的能力，甚至比热修复工具做得更好。

App 每半个月发一次版，新功能上线，一般都会等这个时间点。另一方面，很多公司的 Android 发版策略是受 iPhone 新版本影响的，新功能要等两个版本一起面世，那就只有等 Apple Store 审核通过 iPhone 的版本，Android 才能发版。所以，真的不是那么着急。

在没有插件化的年代，我们做开发都是战战兢兢的，生怕写出什么 bug，非常严重的话就要重新发版本。有了插件化框架，开发人员没有了后顾之忧，于是 App 上线后，每个插件化，每天都会有一到两个新版本发布。Android 插件化框架，已经沦落为 bug 修复的工具。这是我们不愿看到的场景。

其实，插件化框架更适合于游戏领域。比如王者荣耀，经常都会有新皮肤，或者隔几天上线一个新英雄，调整一下英雄的数据，这些都不需要重新发版。

插件化还有一种很好的使用场景，那就是 ABTest，只是没有深入人心罢了。当产品经理为两种风格的设计举棋不定时，那么把这两种策略做成两个插件包，让 50% 的用户下载 A 策略，另外 50% 的用户下载 B 策略，一周后看数据，比如说页面转化率，就知道哪种策略更优了。这就是数据驱动产品。

随着业务线的独立，Android 和 iOS 团队拆分到各自的业务线，有各自的汇报关系，因此有必要把酒店机票火车票这些不同的业务拆分成不同的模块。在 Android 组件化中，模块之间还是以 aar 的方式进行依赖的，所以我们可以借助 Maven 来管理这些 aar。

Android 的这种组件化模型，仅适用于开发阶段，一旦线上有了 bug，或者要发布新功能，那就需要将所有模块重新打包一起发布新版本。

Android 组件化再往前走一步，就是插件化。此时，各个业务模块提供的就不再是 aar 了，而是一个打包好的 apk 文件，放在宿主 App 的 assets 目录下。这样，发版后，某个模块有更新，只需重新打包这个模块的代码，生成增量包，放到服务器上供用户下载就可以了。这才是 Android 插件化在企业级开发中的价值所在。一般的小公司只做了 Android 组件化，没有做插件化，所以体会不到这个好处，这是因为插件化开发成本很高，投入产出比很低。

1.5 更好的替代品：React Native

2015 年，React Native（RN）横空出世，当时并没有多少人关注它，因为它还很不成熟，甚至连基本的控件都没几个。后来随着 RN 项目的迭代，功能日趋完善，虽然迄今为止还

没有一个 1.0 的 release 版本，我们还是欣喜地发现，这个东西就是 Android 和 iOS 的插件化啊。

外国人和中国人的思路不一样。就好像国际象棋与中国象棋不一样。当我们投入大量人力去钻研怎么 Hook Android 系统源码的时候，外国人走的是另一条路，那就是映射，让 Android 或 iOS 中的每个控件，在 RN 中都能找到相对应的控件。RN 是基于 JavaScript 代码来编写的，打包后放到服务器，供 Android 和 iOS 的 App 下载并使用。

RN 比 Android 插件化更好的地方在于它还支持 iOS，因此最大程度地实现了跨平台。于是当我们一厢情愿地以为 Android 插件化多么好用，而对 iOS 如何发布新功能一筹莫展时，便有了 RN 这个更好的选择。至于性能，二者差别不大，RN 在 iOS 和 Android 上都很流畅，这一点不用担心。

对于中小型公司和创业公司而言，缺少人力和财力自己研制一套插件化框架，一般就采用国内比较稳定的、开源的、持续更新的插件化框架。但 iOS 没有这方面的技术框架，尤其是在 jsPatch 热修复被 AppStore 明令禁止了之后，最好的选择就是 RN。只要招聘做 JavaScript 前端的技术人员，就能快速迭代、快速上线了，完全不受发版的限制。我是从研发的岗位走出来，在国内做了两年培训，全国各地走了上百家公司，包括大型国企、二三线互联网公司、传统行业，我发现国内 90% 的公司都属于这种类型，国内对 RN 的需求远大于 Android 插件化。

关于 RN 的话题，至少要一本书才能说清楚。本书主要介绍 Android 插件化，这里只指出 Android 插件化不如 RN 的地方。

1.6　只有中国这么玩吗

有读者会问，Android 插件化在中国如火如荼，为什么在国外却悄无声息？打开硅谷那些独角兽的 App，都没有发现插件化的影子。

一方面原因是，国外人都使用 Google Play，这个官方市场不允许插件化 App 的存在，审核会不通过，这就很像 Apple Store 了。

另一方面原因是，国外没有这样的需求。所以当你发现国外某款 App 显示数据错误了，或者莫名其妙崩溃了，就算你反馈给他们，得到的也是一副坐看闲云、宠辱不惊的回复——下个版本再修复吧。下个版本什么时候？一个月后。

这就和中国国内的 App 境遇不同了。在互联网公司，特别是有销售业务的公司，任何数据显示的错误或者崩溃，都会导致订单数量的下降，直接影响的是钱啊。所以，我经常半夜被叫醒去修 bug，然后快速出新版本的插件包，避免更多订单的损失。

国内的一二线互联网公司，会花很多钱雇佣一群做插件化框架的人，框架设计完，他们一般会比较闲。在 Android 每年发布新版本的时候，他们会很忙，去研究新版本改动了哪些 Android 系统源码，会对自家公司的插件化框架有什么影响。从长期来看，公司花的

这些钱是划算的，基本等于没有插件化而损失的订单数量的价值。

而国内的中小型公司以及创业公司，没有额外的财力来做自己的插件化框架，一般就采用国内比较稳定的、开源插件化框架。后来有了 RN，就转投 RN 的怀抱了。

就在中国的各路牛人纷纷推出自家的 Android 插件化框架之际，国外的技术人员在研究些什么呢？

国外的技术人员比较关注用户体验，所以在国外 Material Design 大行其道，而在中国，基本是设计师只设计出 iOS 的样稿，Android 保持做的一样就够了。国外的技术人员比较关注函数式编程，追求代码的优雅、实用、健壮、复用，而不像国内的 App，为了赶进度超越竞争对手，纯靠人力堆砌代码，甚至带 bug 上线，以后有时间了再进行重构，而那时当初写代码的人也许已经离职了。

所以，当硅谷那边层出不穷地推出 ButterKnife、Dagger、OKHttp、Retrofit、RxJava 的时候，国内能拿出来与之媲美的只有各种插件化框架和热修复框架，以及双开技术。

1.7 四大组件都需要插件化技术吗

在 Android 中，Activity、Service、ContentProvider 和 BroadcastReceiver 并称为四大组件。四大组件都需要插件化吗？这些年，我是一直带着这个问题做插件化技术的。

我所工作过的几家公司都属于 OTA（在线旅游）行业。这类 App 类似于电商，包括完整的一套下单支付流程，用得最多的是 Activity，达数百个；Service 和 Receiver 用得很少，屈指可数；ContentProvider 根本就没用过。

国内大部分 App 都是如此。根据技术栈来划分 App 行业：

- **游戏类 App**，有一套自己的在线更新流程，很多用的是 Lua 之类的脚本。
- **手机助手**、**手机卫士**，这类 App 对 Service、Receiver、ContentProvider 的使用比较多。所以四大组件的插件化都必须实现。
- **音乐类、视频类、直播类 App**，除了使用比较多的 Activity，对 Service 和 Receiver 的依赖很强。
- **电商类、社交类、新闻类、阅读类 App**，基本是 Activity，其他三大组件使用不是很多，可以只考虑对 Activity 插件化的支持。

我们应该根据 App 对四大组件的依赖程度，来选择合适的插件化技术。四大组件全都实现插件化固然是最好的，但是如果 App 中主要是 Activity，那么选择静态代理 that 框架就够了。

1.8 双开和虚拟机

既然插件化会慢慢被 RN 所取代，那么插件化的未来是什么？答案是，虚拟机技术。

各位读者应该有过在 PC 机上安装虚拟机的经历。只要电脑的内存足够大，那么就可以

同时打开多个虚拟机，在每个虚拟机上都安装 QQ 软件，使用不同的账号登录，然后自己跟自己聊天。

在 Android 系统上，是否也能支持安装一个或多个虚拟机呢？国内已经有人在做了，我所知道的，一个是高中生 Lody（当你阅读这本书的时候，他应该已经是大学生了吧），他有一个很著名的开源项目 VirtualApp[⊖]，这个项目现在已经商业化运作了。另一个是 DroidPlugin 的作者张勇，他现在创业专职做这个，在 DroidPlugin 的基础上研发了闪电盒子，可以极速加载各种 apk。

有了这样一个虚拟机系统，我们就可以在手机上打开两个不同账号的 QQ，自己和自己聊天了。

我们称同时打开一个 App 的多个分身的技术叫“双开”。现在国内有些手机系统已经支持双开技术了，可以在设置中看到这一选项。

关于双开和虚拟机的技术，我们就介绍这么多，毕竟这已经不是本书所涉及的技术范畴了。

1.9　从原生页面到 HTML 5 的过渡

无线技术越来越成熟，已经从 2012 年时的初步开荒，发展到现在的蔚为壮观。对于国人来说，我们比较关注的是：热修复、性能、开发效率、App 体积、数据驱动产品。这些点目前都已经有了很好的解决方案，也涌现出 RxJava、LeakCanary 这样优秀的框架。这个话题很大，本文就不展开说了。

由 App 技术的无比繁荣，回想起我 2004 年刚工作的时候，IT 行业正从 CS（Service-Client）转型为 BS（Browser-Server）。2004 年之前大都是 CS 这样的软件，比如 Windows 上安装一个联众的客户端就可以和网友斗地主了，后来互联网的技术成熟起来了，就把原先的系统都搬到网站上，这就是 BS。

后来 BS 做多了，大家觉得 BS 太“单薄”，很多功能不支持，不如 CS，于是就提出 SmartClient 的概念，也就是智能客户端，Outlook 就是一个很好的例子，你可以脱机读和写邮件，没网络也可以，什么时候有网络了，再将写好的邮件发送出去。

再后来 Flash 火起来了，这个本来是网页制作工具三剑客之一，却阴差阳错地成为了网页富客户端的鼻祖。在此基础上便有了 Flex，现在还有些公司在使用。微软这时候提出了 Silverlight，是搭载在网页上的。与此同时，JavaScript 也在发力，并逐渐取代前者，成为富客户端的最后赢家，那时候《JavaScript 设计模式》一书非常畅销。

JavaScript 在 2004 年仅是用来做网页特效的。时至今日，我们发现，JavaScript 经历了 Ajax、jQuery、ECMAScript 从 1 到 6、Webpack 打包，以及 Angular、React、Vue 三大主

⊖　开源项目地址：https://github.com/asLody/VirtualApp

流框架，已经变得无比强大，被封装成一门“面向对象”的语言了。

前面铺垫了那么多，就是想说明 App 也在走同样的发展道路，先沉淀几年，把网站的很多技术都搬到 App 上，也就是目前的发展阶段，差不多该有的也都有了。下一个阶段就是从 CS 过渡到 BS，Hybird 技术就类似于 BS，但是有很多缺陷，尤其是 Web Browser 性能很差，然后便出现了 React Native，HTML 5 很慢，但可以把 HTML 5 翻译成原生的代码啊。再往前发展是什么样，我不知道，但是这个发展方向是很清晰的。一方面，Android 和 iOS 技术不会消亡；另一方面 HTML 5 将慢慢成为 App 开发的主流。

1.10 本章小结

本章回顾了 Android 插件化技术的发展历史，基本上分为两大流派：静态代理和动态替换，所有的插件化框架都基于此。看完这段历史你会发现，这门技术也不是一蹴而就的，期间也经历了从无到有，以及逐步完善的过程。

插件化技术不仅仅用于修复 bug 和动态发布新功能，我们在研究插件化技术的过程中，顺带开发出了 Android 虚拟机和双开技术，这是一个新的技术领域，可以摆脱 Android 原生系统的束缚，更快地运行 App。

本章还谈到了 ReactNative，它也能修复 bug 和动态发布新功能，和 Android 插件化有异曲同工之妙。具体该采用哪门技术，取决于研发团队以 H5 为主还是以 Android 为主，取决于是否要发布到 Google Play。

第 2 章 Chapter 2

Android 底层知识

这一章，改编自 2017 年我的系列文章《写给 Android App 开发人员看的 Android 底层知识》[⊖]。在此基础上，扩充了 PMS、双亲委托、ClassLoader 等内容。这些 Android 底层知识都是学习 Android 插件化技术所必需的。

本章介绍的这些 Android 底层知识基于 Android 6.0.1 版本。我把本章以及整本书涉及的 Android 系统底层的类或 aidl 都搜集在一起，放在我的 GitHub 上[⊜]，读者可以下载并研读这些代码。

2.1 概述

在我还是 Android 菜鸟的时候，有很多技术我都不太明白，也都找不到答案，比如，apk 是怎么安装的？资源是怎么加载的？再比如，每本书都会讲 AIDL，但我却从来没用过。四大组件也是这个问题，我只用过 Activity，其他三个组件不但没用过，甚至连它们是做什么的，都不是很清楚。

之所以这样，是因为我一直从事的是电商类 App 开发的工作，这类 App 基本是由列表页和详情页组成的，所以我每天面对的是 Activity，会写这两类页面，把网络底层封装得足够强大就够了。绝大多数 App 开发人员都是如此。但直到接触 Android 的插件化编程和热修复技术，我才发现只掌握上述这些技术是远远不够的。

市场上有很多介绍 Android 底层的书籍，网上也有很多文章，但大都是给 ROM 开发人员看的——动辄贴出几页代码，这类书不适合 App 开发人员去阅读学习。

⊖ 文章地址：http://www.cnblogs.com/Jax/p/6864103.html

⊜ 项目地址：https://github.com/BaoBaoJianqiang/AndroidSourceCode

于是，这几年来，我一直在寻找这样一类知识，App 开发人员看了能有助于他们更好地编写 App 程序，而又不需要知道太多这门技术底层的代码实现。

这类知识分为两种：

- 知道概念即可，比如 Zygote，其实 App 开发人员是不需要了解 Zygote 的，知道有这么个东西是“孕育天地”的就够了，类似的还有 SurfaceFlinger、WMS 这些概念。
- 需要知道内部原理，比如 Binder，关于 Binder 的介绍铺天盖地，但对于 App 开发者，需要了解的是它的架构模型，只要有 Client、Server 以及 ServiceManager 就足够了。

四大组件的底层通信机制都是基于 Binder 的，我们需要知道每个组件中，分别是哪些类扮演了 Binder Client，哪些类扮演了 Binder Server。知道这些概念有助于 App 开发人员进行插件化编程。

接下来的章节将介绍以下概念，掌握了这些底层知识，就算是迈进 Android 插件化的大门了：

- Binder；
- AIDL；
- AMS；
- 四大组件的工作原理；
- PMS；
- App 安装过程；
- ClassLoader 以及双亲委托。

2.2 Binder 原理

Binder 的目的是解决跨进程通信。关于 Binder 的文章实在是太多了，每篇文章都能从 Java 层讲到 C++ 层，App 开发人员其实是没必要了解这么多内容的。我们看看对 App 开发有用的几个知识点：

1）Binder 分为 Client 和 Server 两个进程。

注意，Client 和 Server 是相对的。谁发消息，谁就是 Client，谁接收消息，谁就是 Server。举个例子，进程 A 和进程 B 之间使用 Binder 通信，进程 A 发消息给进程 B，那么这时候 A 是 Binder Client，B 是 Binder Server；进程 B 发消息给进程 A，那么这时候 B 是 Binder Client，A 是 Binder Server。其实，这么说虽然简单，但是不太严谨，我们先这么理解。

2）Binder 的组成。

Binder 的架构如图 2-1 所示，图中的 IPC 即为进程间通信，ServiceManager 负责把 Binder Server 注册到一个容器中。

有人把 ServiceManager 恰当地比喻成电话局，存储着每个住宅的座机电话。张三给李四打电话，拨打电话号码，会先转接到电话局，电话局的接线员查到这个电话号码的地址，因为李四的电话号码之前在电话局注册过，所以就能拨通；如果没注册，就会提示该号码不存在。

对照着 Android Binder 机制和图 2-1，张三就是 Binder Client，李四就是 Binder Server，电话局就是 ServiceManager，电话局的接线员在这个过程中做了很多事情，对应着图中的 Binder 驱动。

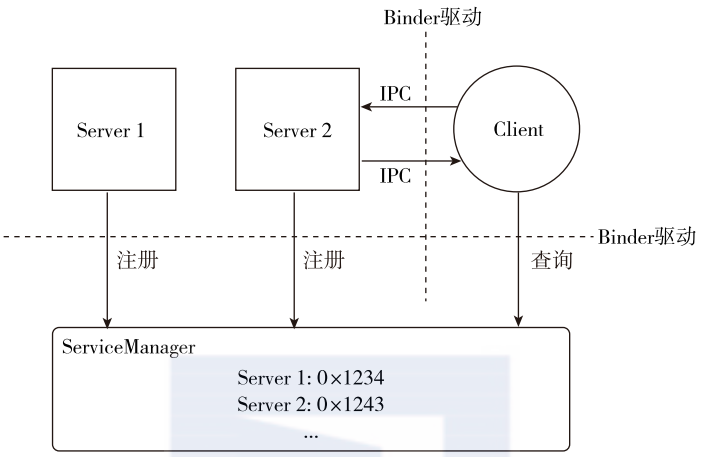


图 2-1　Binder 的组成（摘自田维术的博客）

3）Binder 的通信过程。

Binder 通信流程如图 2-2 所示，图中的 SM 即为 ServiceManager。我们看到，Client 不可以直接调用 Server 的 add 方法，因为它们在不同的进程中，这时候就需要 Binder 来帮忙了。

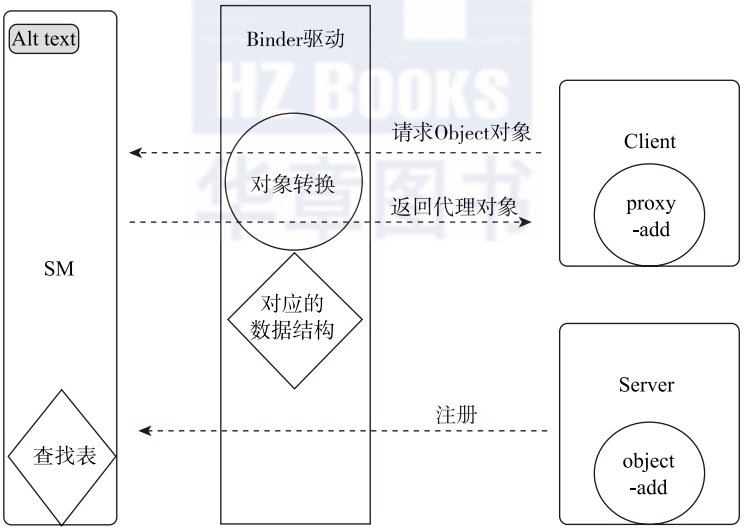


图 2-2　Binder 的通信流程（摘自田维术的博客）

- 首先，Server 在 SM 容器中注册。
- 其次，Client 若要调用 Server 的 add 方法，就需要先获取 Server 对象，但是 SM 不会把真正的 Server 对象返回给 Client，而是把 Server 的一个代理对象，也就是

Proxy，返回给 Client。

- 再次，Client 调用 Proxy 的 add 方法，ServiceManager 会帮它去调用 Server 的 add 方法，并把结果返回给 Client。

以上这 3 步，Binder 驱动出了很多力，但我们不需要知道 Binder 驱动的底层实现，这涉及 C 或 C++ 的代码。我们要把有限的时间用在更有意义的事情上。

App 开发人员对 Binder 的掌握，这些内容就足够了。

综上所述：

1）学习 Binder 是为了更好地理解 AIDL，基于 AIDL 模型，进而了解四大组件的原理。

2）理解了 Binder 再看 AMS 和四大组件的关系，就像是 Binder 的两个进程 Server 和 Client 通信。

2.3 AIDL 原理

AIDL 是 Binder 的延伸。一定要先了解前文介绍的 Binder，再来看 AIDL。要按顺序阅读。

Android 系统中很多系统服务都是 AIDL，比如剪切板。举这个例子是为了让 App 开发人员知道 AIDL 和我们距离非常近，无处不在。

学习 AIDL 需要知道下面几个类：

- IBinder
- IInterface
- Binder
- Proxy
- Stub

当我们自定义一个 aidl 文件时（比如 MyAidl.aidl，里面有一个 sum 方法），Android Studio 会帮我们生成一个类文件 MyAidl.java，如图 2-3 所示。

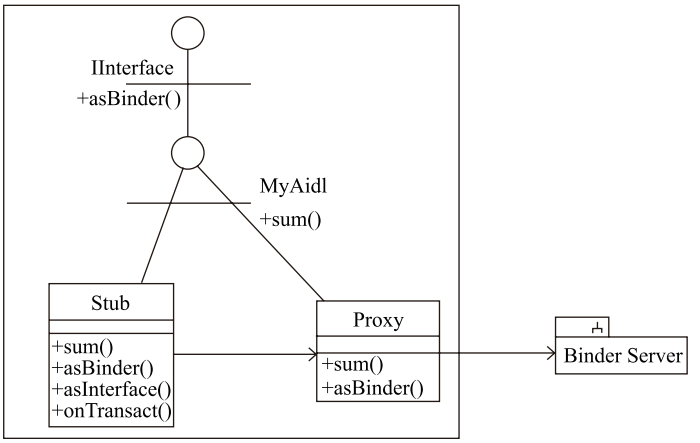


图 2-3 AIDL 中涉及的类图

我们把 MyAidl.java 中的三个类拆开，就一目了然了，如下所示：

```
public interface MyAidl extends android.os.IInterface {
    public int sum(int a, int b) throws android.os.RemoteException;
}

public abstract class Stub extends android.os.Binder implements jianqiang.com.
    hostapp.MyAidl {
    private static final java.lang.String DESCRIPTOR = "jianqiang.com.hostapp.
        MyAidl";

    static final int TRANSACTION_sum = (android.os.IBinder.FIRST_CALL_TRANSACTION
        + 0);

    /**
     * Construct the stub at attach it to the interface.
     */
    public Stub() {
        this.attachInterface(this, DESCRIPTOR);
    }

    /**
     * Cast an IBinder object into an jianqiang.com.hostapp.MyAidl interface,
     * generating a proxy if needed.
     */
    public static jianqiang.com.hostapp.MyAidl asInterface(android.os.IBinder
        obj) {
        if ((obj == null)) {
            return null;
        }
        android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
        if (((iin != null) && (iin instanceof jianqiang.com.hostapp.MyAidl))) {
            return ((jianqiang.com.hostapp.MyAidl) iin);
        }
        return new jianqiang.com.hostapp.MyAidl.Stub.Proxy(obj);
    }

    @Override
    public android.os.IBinder asBinder() {
        return this;
    }

    @Override
    public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel
        reply, int flags) throws android.os.RemoteException {
        switch (code) {
            case INTERFACE_TRANSACTION: {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_sum: {
```

```
                data.enforceInterface(DESCRIPTOR);
                int _arg0;
                _arg0 = data.readInt();
                int _arg1;
                _arg1 = data.readInt();
                int _result = this.sum(_arg0, _arg1);
                reply.writeNoException();
                reply.writeInt(_result);
                return true;
            }
        }

        return super.onTransact(code, data, reply, flags);
    }
}

class Proxy implements jianqiang.com.hostapp.MyAidl {
    private android.os.IBinder mRemote;

    Proxy(android.os.IBinder remote) {
        mRemote = remote;
    }

    @Override
    public android.os.IBinder asBinder() {
        return mRemote;
    }

    public java.lang.String getInterfaceDescriptor() {
        return DESCRIPTOR;
    }

    @Override
    public int sum(int a, int b) throws android.os.RemoteException {
        android.os.Parcel _data = android.os.Parcel.obtain();
        android.os.Parcel _reply = android.os.Parcel.obtain();
        int _result;
        try {
            _data.writeInterfaceToken(DESCRIPTOR);
            _data.writeInt(a);
            _data.writeInt(b);
            mRemote.transact(Stub.TRANSACTION_sum, _data, _reply, 0);
            _reply.readException();
            _result = _reply.readInt();
        } finally {
            _reply.recycle();
            _data.recycle();
        }
        return _result;
    }
}
```

我曾经很不理解，为什么不是生成 3 个文件——一个接口，两个类，清晰明了。都放在一个文件中，这是导致很多人看不懂 AIDL 的一个门槛。其实，Android 这样设计是有道理的。当有多个 AIDL 类的时候，Stub 和 Proxy 类就会重名，把它们放在各自的 AIDL 接口中，就区分开了。

对照图 2-3，我们继续来分析，Stub 的 sum 方法是怎么调用到 Proxy 的 sum 方法的，然后又是怎么调用另一个进程的 sum 方法的？

起决定作用的是 Stub 的 asInterface 方法和 onTransact 方法。其实图 2-3 没有画全，把完整的 Binder Server 也加上，应该如图 2-4 所示。

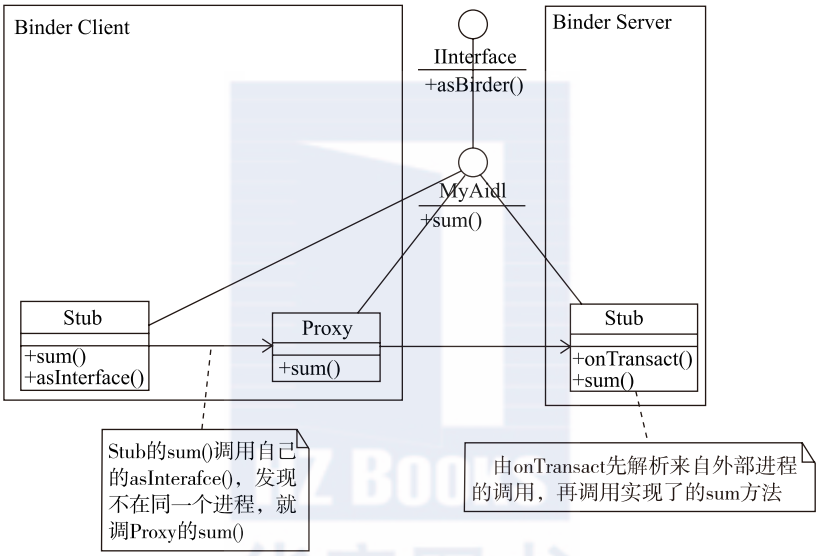


图 2-4　完整的 AIDL 类图

1）从 Client 看，对于 AIDL 的使用者，我们写程序：

```
MyAidl.Stub.asInterface(某IBinder对象).sum(1, 2);   //最好在执行sum方法前判空。
```

asInterface 方法的作用是判断参数，也就是 IBinder 对象，和自己是否在同一个进程，如果

- 是，则直接转换、直接使用，接下来则与 Binder 跨进程通信无关；
- 否，则把这个 IBinder 参数包装成一个 Proxy 对象，这时调用 Stub 的 sum 方法，间接调用 Proxy 的 sum 方法，代码如下：

```
return new MyAidl.Stub.Proxy(obj);
```

2）Proxy 在自己的 sum 方法中，会使用 Parcelable 来准备数据，把函数名称、函数参数都写入 _data，让 _reply 接收函数返回值。最后使用 IBinder 的 transact 方法，就可把数据传给 Binder 的 Server 端了。

```
mRemote.transact(Stub.TRANSACTION_addBook, _data, _reply, 0); //这里的mRemote就是
    asInterface方法传过来的obj参数
```

3）Server 则是通过 onTransact 方法接收 Client 进程传过来的数据，包括函数名称、函数参数，找到对应的函数（这里是 sum），把参数喂进去，得到结果，返回。所以 onTransact 函数经历了读数据→执行要调用的函数→把执行结果再写数据的过程。

下面要介绍的四大组件的原理，我们都可以对照图 2-4 来理解，比如，四大组件的启动和后续流程，都是在与 ActivityManagerService（AMS）来来回回地通信，四大组件给 AMS 发消息，四大组件就是 Binder Client，而 AMS 就是 Binder Server；AMS 发消息通知四大组件，那么角色互换。

在四大组件中，比如 Activity，是哪个类扮演了 Stub 的角色，哪个类扮演了 Proxy 的角色呢？这也是本章下面要介绍的，包括 AMS、四大组件各自的运行原理。

好戏即将开始。

2.4 AMS

如果站在四大组件的角度来看，AMS 就是 Binder 中的 Server。

AMS（ActivityManagerService）从字面意思上看是管理 Activity 的，但其实四大组件都归它管。

由此而说到了插件化，有两个困惑我已久的问题：

1）App 的安装过程，为什么不把 apk 解压缩到本地，这样读取图片就不用每次从 apk 包中读取了。这个问题，我们放到 2.12 节再详细说。

2）为什么 Hook 永远是在 Binder Client 端，也就是四大组件这边，而不是在 AMS 那一侧进行 Hook。

这里要说清楚第二个问题。就拿 Android 剪切板举例吧。前面说过，这也是个 Binder 服务。

AMS 要负责和所有 App 的四大组件进行通信。如果在一个 App 中，在 AMS 层面把剪切板功能进行了 Hook，那会导致 Android 系统所有的剪切板功能被 Hook——这就是病毒了，如果是这样的话，Android 系统早就死翘翘了。所以 Android 系统不允许我们这么做。

我们只能在 AMS 的另一侧，即 Client 端，也就是四大组件这边做 Hook，这样即使我们把剪切板功能进行了 Hook，也只影响 Hook 代码所在的 App，在别的 App 中，剪切板功能还是正常的。

关于 AMS 我们就说这么多，下面的小节在介绍四大组件时，会反复提到四大组件和 AMS 的跨进程通信。

2.5　Activity 工作原理

对于 App 的开发人员而言，Activity 是四大组件中用得最多的，也是最复杂的。这里只讲述 Activity 的启动和通信原理。

2.5.1　App 是怎么启动的

在手机屏幕上点击某个 App 的图标，假设是斗鱼 App，这个 App 的首页（或引导页）就出现在我们面前了。这个看似简单的操作，背后经历了 Activity 和 AMS 的反反复复的通信过程。

首先要搞清楚，在手机屏幕上点击 App 的快捷图标，此时手机屏幕就是一个 Activity，而这个 Activity 所在的 App，业界称之为 Launcher。Launcher 是各手机系统厂商提供的，比拼的是谁的 Launcher 绚丽和人性化。

Launcher 这个 App，其实和我们做的各种应用类 App 没有什么不同，我们大家用过华为、小米之类的手机，预装 App 以及我们下载的各种 App，都显示在 Launcher 上，每个 App 表现为一个图标。图标多了可以分页，可以分组，此外，Launcher 也会发起网络请求，调用天气的数据，显示在屏幕上，即人性化的界面。

还记得我们在开发一款 App 时，在 AndvoidManifest 文件中是怎么定义默认启动 Activity 的吗？代码如下所示：

```
<activity android:name=".MainActivity">
    <intent-filter>
            <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

而 Launcher 中为每个 App 的图标提供了启动这个 App 所需要的 Intent 信息，如下所示（以斗鱼 App 为例）：

```
action: android.intent.action.MAIN
category: android.intent.category.LAUNCHER
cmp: 斗鱼的包名+ 首页Activity名
```

这些信息是 App 安装（或 Android 系统启动）的时候，PackageManager-Service 从斗鱼的 apk 包的 AndroidManifest 文件中读取到的。所以点击图标就启动了斗鱼 App 的首页。

2.5.2　启动 App 并非那么简单

前文只是 App 启动的一个最简单的描述。

仔细看，我们会发现，Launcher 和斗鱼是两个不同的 App，它们位于不同的进程中，它们之间的通信是通过 Binder 完成的——这时候 AMS 出场了。

仍然以启动斗鱼 App 为例，整体流程分为以下 7 个阶段[⊖]。

1）Launcher 通知 AMS，要启动斗鱼 App，而且指定要启动斗鱼 App 的哪个页面（也就是首页）。

2）AMS 通知 Launcher，“好了我知道了，没你什么事了”。同时，把要启动的首页记下来。

3）Launcher 当前页面进入 Paused 状态，然后通知 AMS，“我睡了，你可以去找斗鱼 App 了”。

4）AMS 检查斗鱼 App 是否已经启动了。是，则唤起斗鱼 App 即可。否，就要启动一个新的进程。AMS 在新进程中创建一个 ActivityThread 对象，启动其中的 main 函数。

5）斗鱼 App 启动后，通知 AMS，“我启动好了”。

6）AMS 翻出之前在 2）中存的值，告诉斗鱼 App，启动哪个页面。

7）斗鱼 App 启动首页，创建 Context 并与首页 Activity 关联。然后调用首页 Activity 的 onCreate 函数。

至此启动流程完成，可分成两部分：第 1～3 阶段，Launcher 和 AMS 相互通信；第 4～7 阶段，斗鱼 App 和 AMS 相互通信。

这会涉及一堆类，列举如下，在接下来的分析中，我们会遇到这些类。

- Instrumentation；
- ActivityThread；
- H；
- LoadedApk；
- AMS；
- ActivityManagerNative 和 ActivityManagerProxy；
- ApplicationThread 和 ApplicationThreadProxy。

第 1 阶段：Launcher 通知 AMS

第 1 步和第 2 步——点击图标启动 App。

从图 2-5 中我们看到，点击 Launcher 上的斗鱼 App 的快捷图标，这时会调用 Launcher 的 startActivitySafely 方法，其实还是会调用 Activity 的 startActivity 方法，intent 中带着要启动斗鱼 App 所需要的如下关键信息：

```
action = "android.intent.action.MAIN"
category = "android.intent.category.LAUNCHER"
cmp = "com.douyu.activity.MainActivity"
```

第 3 行代码是我推测的，就是斗鱼 App 在 AndroidManifest 文件中指定为首页的那个 Activity。这样，我们终于明白，为什么在 AndroidManifest 中，给首页指定 action 和

⊖ 这个流程分析，基本上是基于 Android 6.0 的源码进行的。

category 了。在 App 的安装过程中，会把这个信息“记录”在 Launcher 的斗鱼启动快捷图标中。关于 App 的安装过程，我会在后面的文章详细介绍。

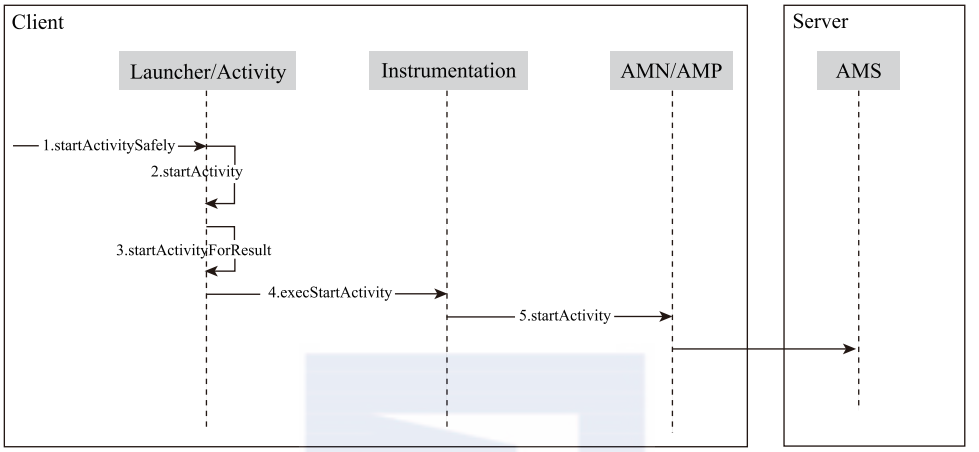


图 2-5　Launcher 通知 AMS 的流程

startActivity 这个方法，如果我们看它的实现，会发现它调来调去，经过一系列 startActivity 的重载方法，最后会走到 startActivityForResult 方法。代码如下：

```
public void startActivity(Intent intent, @Nullable Bundle options) {
    if (options != null) {
        startActivityForResult(intent, -1, options);
    } else {
        startActivityForResult(intent, -1);
    }
}
```

我们知道 startActivityForResult 需要两个参数，一个是 intent，另一个是 code，这里 code 是 -1，表示 Launcher 才不关心斗鱼的 App 是否启动成功的返回结果。

第 3 步——startActivityForResult。

Activity 内部会保持一个对 Instrumentation 的引用，但凡是做过 App 单元测试的读者，对这个类都很熟悉，习惯上称之为“仪表盘”。

在 startActivityForResult 方法的实现中，会调用 Instrumentation 的 execStartActivity 方法。代码如下：

```
public void startActivityForResult(Intent intent, int requestCode, @Nullable
    Bundle options) {
    //前后省略一些代码
    Instrumentation.ActivityResult ar =
        mInstrumentation.execStartActivity(
            this, mMainThread.getApplicationThread(), mToken, this,
            intent, requestCode, options);
    }
```

看到这里，我们发现有个 mMainThread 变量，这是一个 ActivityThread 类型的变量。这个家伙的来头可不小。

ActivityThread，就是主线程，也就是 UI 线程，它是在 App 启动时创建的，它代表了 App 应用程序。读者不禁会问，ActivityThread 代表了 App 应用程序，那 Application 类岂不是被架空了？其实，Application 对我们 App 开发人员来说也许很重要，但是在 Android 系统中还真的没那么重要，它就是个上下文。Activity 不是有一个 Context 上下文吗？Application 就是整个 ActivityThread 的上下文。

ActivityThread 则没有那么简单。它里面有 main 函数。我们知道大部分程序都有 main 函数，比如 Java 和 C#，iPhone App 用到的 Objective-C 也有 main 函数。那么，Android 的 main 函数藏在哪里？就在 ActivityThread 中，如下所示，代码太多，此处只截取了一部分：

```
public final class ActivityThread {
    public static void main(String[] args) {
        Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "ActivityThreadMain");
        SamplingProfilerIntegration.start();
        CloseGuard.setEnabled(false);
        Environment.initForCurrentUser();

        //以下省略了很多代码
    }
}
```

又有人会问：不是说谁写的程序，谁就要提供 main 函数作为入口吗？但 Android App 却不是这样的。Android App 的 main 函数在 ActivityThread 里面，而这个类是 Android 系统提供的底层类，不是我们提供的。

所以这就是 Android 有趣的地方。Android App 的入口是 AndroidManifest 中定义默认启动 Activity。这是由 Android AMS 与四大组件的通信机制决定的。

上面的代码中传递了两个很重要的参数：

- 通过 ActivityThread 的 getApplicationThread 方法取到一个 Binder 对象，这个对象的类型为 ApplicationThread，代表了 Launcher 所在的 App 进程。
- mToken 也是一个 Binder 对象，代表 Launcher 这个 Activity 也通过 Instrumentation 传给 AMS，AMS 一查电话簿，就知道是谁向 AMS 发起了请求。

这两个参数是伏笔，传递给 AMS，以后 AMS 想反过来通知 Launcher，就能通过这两个参数找到 Launcher。

第 4 步——Instrumentation 的 execStartActivity 方法。

Instrumentation 绝对是 Android 测试团队的最爱，因为它可以帮助我们启动 Activity。

回到 App 的启动过程来，在 Instrumentation 的 execStartActivity 方法中：

```
public class Instrumentation {

    public ActivityResult execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity target,
            Intent intent, int requestCode, Bundle options) {

        //省略一些代码

        try {
            //省略一些代码

            int result = ActivityManagerNative.getDefault()
                .startActivity(whoThread, who.getBasePackageName(), intent,
                    intent.resolveTypeIfNeeded(who.getContentResolver()),
                    token, target != null ? target.mEmbeddedID : null,
                    requestCode, 0, null, options);

            //省略一些代码
        } catch (RemoteException e) {
            throw new RuntimeException("Failure from system", e);
        }
        return null;
    }
}
```

这就是一个透传，借助 Instrumentation，Activity 把数据传递给 ActivityManagerNativ。

第 5 步——AMN 的 getDefault 方法。

ActivityManagerNative（AMN），这个类后面会反复用到。

ServiceManager 是一个容器类。AMN 通过 getDefault 方法，从 ServiceManager 中取得一个名为 activity 的对象，然后把它包装成一个 ActivityManagerProxy 对象（AMP），AMP 就是 AMS 的代理对象。

AMN 的 getDefault 方法返回类型为 IActivityManager，而不是 AMP。IActivityManager 是一个实现了 IInterface 的接口，里面定义了四大组件所有的生命周期。

AMN 和 AMP 都实现了 IActivityManager 接口，AMS 继承自 AMN，对照着前面 AIDL 的 UML，就不难理解了。AMN 和 AMP 如图 2-6 所示。

第 6 步——AMP 的 startActivity 方法。

看到这里，你会发现 AMP 的 startActivity 方法和 AIDL 的 Proxy 方法，是一模一样的，写入数据到另一个进程，也就是 AMS，然后等待 AMS 返回结果。

至此，第 1 阶段的工作就做完了。

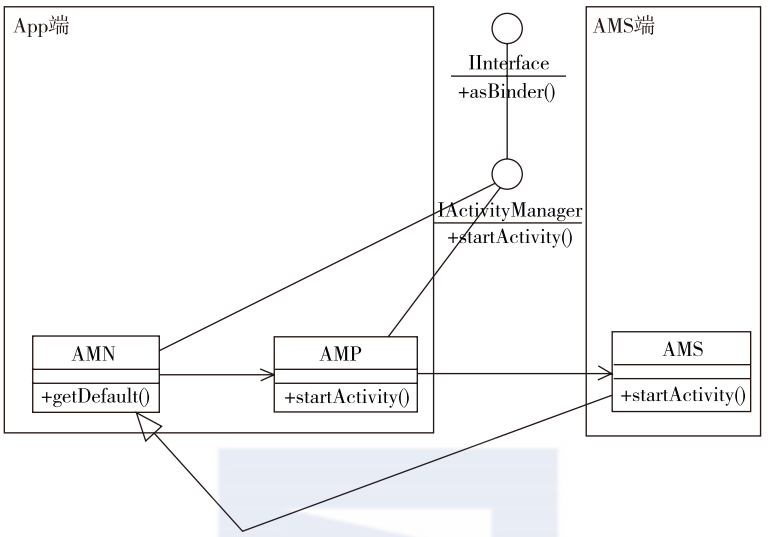


图 2-6 AMN/AMP 的类图

第 2 阶段：AMS 处理 Launcher 传过来的信息

先来看一下 AMP 的 startActivity 方法的实现：

```
class ActivityManagerProxy implements IActivityManager
{
    public int startActivity(IApplicationThread caller, String callingPackage,
        Intent intent,
            String resolvedType, IBinder resultTo, String resultWho, int requestCode,
            int startFlags, ProfilerInfo profilerInfo, Bundle options) throws
                RemoteException {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        data.writeInterfaceToken(IActivityManager.descriptor);
        data.writeStrongBinder(caller != null ? caller.asBinder() : null);
        data.writeString(callingPackage);

        mRemote.transact(START_ACTIVITY_TRANSACTION, data, reply, 0);
        reply.readException();
        int result = reply.readInt();
        reply.recycle();
        data.recycle();
        return result;
    }
}
```

这个阶段主要是 Binder 的 Server 端在做事情。因为我们没有机会修改 Binder 的 Server 端逻辑，所以这个阶段看起来非常“枯燥”，主要过程如下。

1）Binder（也就是 AMN/AMP）和 AMS 通信，肯定每次是做不同的事情，比如这次

Launcher 要启动斗鱼 App，那么会发送类型为 START_ACTIVITY 的请求给 AMS，同时会告诉 AMS 要启动哪个 Activity。

2）AMS 说，“好，我知道了。”然后它会干一件很有趣的事情——检查斗鱼 App 中的 AndroidManifest 文件，是否存在要启动的 Activity。如果不存在，就抛出 Activity not found 的错误，各位做 App 的读者对这个异常应该再熟悉不过了，经常写了个 Activity 而忘记在 AndroidManifest 中声明，就报这个错误。这是因为 AMS 在这里做检查。不管是新启动一个 App 的首页，还是在 App 内部跳转到另一个 Activity，都会做这个检查。

3）AMS 通知 Launcher，“没你什么事了，你洗洗睡吧。”那么 AMS 是通过什么途径告诉 Launcher 的呢？

前面讲过，Binder 的双方进行通信是平等的，谁发消息谁就是 Client，接收的一方就是 Server。Client 这边会调用 Server 的代理对象。对于从 Launcher 发来的消息，通过 AMS 的代理对象 AMP 发送给 AMS。

那么当 AMS 想给 Launcher 发消息，又该怎么办呢？前面不是把 Launcher 以及它所在的进程给传过来了吗？它在 AMS 这边保存为一个 ActivityRecord 对象，这个对象里面有一个 ApplicationThreadProxy，顾名思义，这就是一个 Binder 代理对象。它的 Binder 真身，也就是 ApplicationThread。

站在 AIDL 的角度，来画这张图，如图 2-7 所示。

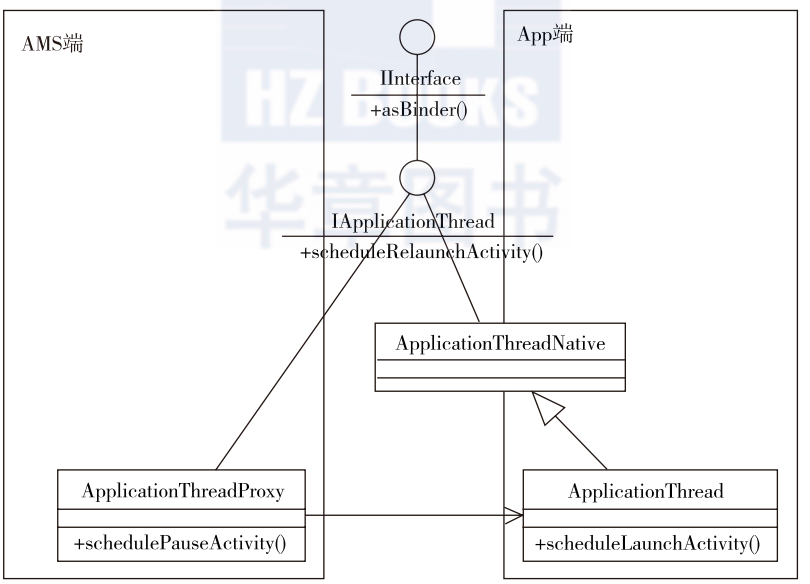


图 2-7　IApplicationThread 的类簇

结论是，AMS 通过 ApplicationThreadProxy 发送消息，而 App 端则通过 ApplicationThread 来接收这个消息。

第 3 阶段：Launcher 去休眠，然后通知 AMS：“我真的已经睡了”

此阶段的过程如图 2-8 所示。

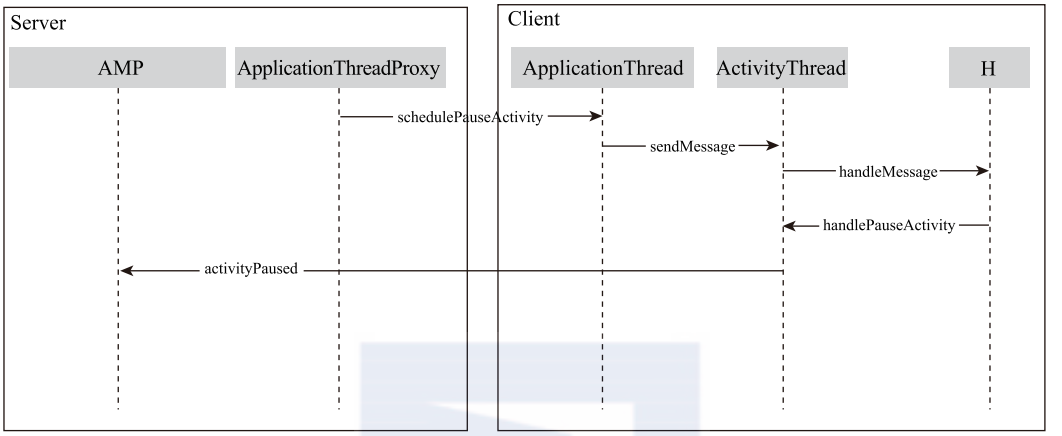


图 2-8 Launcher 再次通知 AMS

ApplicationThread（APT），它和 ApplicationThreadProxy（ATP）的关系，我们在第 2 阶段已经介绍过了。

APT 接收到来自 AMS 的消息后，调用 ActivityThread 的 sendMessage 方法，向 Launcher 的主线程消息队列发送一个 PAUSE_ACTIVITY 消息。

前面说过，ActivityThread 就是主线程（UI 线程）

看到下面的代码是不是很亲切？

```
private void sendMessage(int what, Object obj, int arg1, int arg2, boolean async) {
    if (DEBUG_MESSAGES) Slog.v(
        TAG, "SCHEDULE " + what + " " + mH.codeToString(what)
        + ": " + arg1 + " / " + obj);
    Message msg = Message.obtain();
    msg.what = what;
    msg.obj = obj;
    msg.arg1 = arg1;
    msg.arg2 = arg2;
    if (async) {
        msg.setAsynchronous(true);
    }
    mH.sendMessage(msg);
}
```

发送消息是通过一个名为 H 的 Handler 类的完成的，这个 H 类的名字真有个性，很容易记住。

做 App 的读者都知道，继承自 Handler 类的子类，就要实现 handleMessage 方法，这里是一个 switch…case 语句，处理各种各样的消息，PAUSE_ACTIVITY 消息只是其中一种。

由此也能预见，AMS 给 Activity 发送的所有消息，以及给其他三大组件发送的所有消息，都从 H 这里经过。为什么要强调这一点呢？既然四大组件都走这条路，那么就可以从这里入手做插件化技术，这个我们以后介绍插件化技术的时候会讲到。

代码如下：

```
public final class ActivityThread {
    private class H extends Handler {

    //省略一些代码
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
        switch (msg.what) {
            case PAUSE_ACTIVITY:
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityPause");
            handlePauseActivity((IBinder)msg.obj, false, (msg.arg1&1) != 0, msg.
                arg2, (msg.arg1&2) != 0);
                    maybeSnapshot();
                    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                    break;
            };
        //省略一些代码
    }
}
```

H 对 PAUSE_ACTIVITY 消息的处理，如上面的代码，是调用 ActivityThread 的 handlePauseActivity 方法。这个方法做两件事：

- ActivityThread 里面有一个 mActivities 集合，保存当前 App 也就是 Launcher 中所有打开的 Activity，把它找出来，让它休眠。
- 通过 AMP 通知 AMS，“我真的休眠了。”

你可能会找不到 H 和 APT 这两个类文件，那是因为它们都是 ActivityThread 的内嵌类。

至此，Launcher 的工作完成了。你可以看到在这个过程中，各个类都起到了什么作用。

第 4 阶段：AMS 启动新的进程

接下来又轮到 AMS 做事了，你会发现我不太喜欢讲解 AMS 的流程，甚至都不画 UML 图，因为这部分逻辑和 App 开发人员关系不是很大，我尽量说得简单一些，把流程说清楚即可。

AMS 接下来要启动斗鱼 App 的首页，因为斗鱼 App 不在后台进程中，所以要启动一个新的进程。这里调用的是 Process.start 方法，并且指定了 ActivityThread 的 main 函数为入口函数。代码如下：

```
int pid = Process.start(“android.app.ActivityThread”,
     mSimpleProcessManagement ? app.processName : gid, debugFlags, null);
```

第 5 阶段：新的进程启动，以 ActivityThread 的 main 函数作为入口

启动新进程，其实就是启动一个新的 App，如图 2-9 所示。

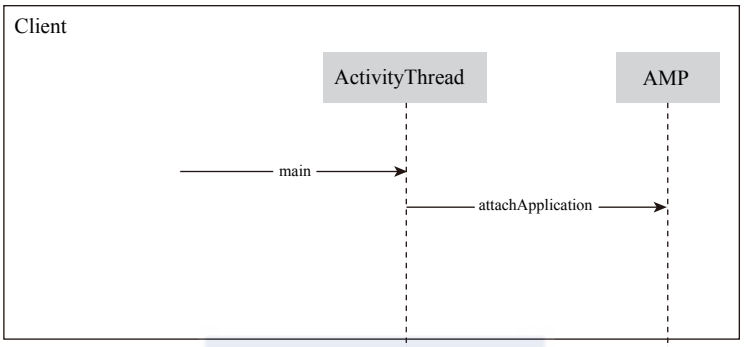


图 2-9 ActivityThread 启动

在启动新进程的时候，为这个进程创建 ActivityThread 对象，这就是我们耳熟能详的主线程（UI 线程）。

创建好 UI 线程后，立刻进入 ActivityThread 的 main 函数，接下来要做两件具有重大意义的事情：

1）创建一个主线程 Looper，也就是 MainLooper。注意，MainLooper 就是在这里创建的。

2）创建 Application。注意，Application 是在这里生成的。

主线程在收到 BIND_APPLICATION 消息后，根据传递过来的 ApplicationInfo 创建一个对应的 LoadedApk 对象（标志当前 APK 信息），然后创建 ContextImpl 对象（标志当前进程的环境），紧接着通过反射创建目标 Application，并调用其 attach 方法，将 ContextImpl 对象设置为目标 Application 的上下文环境，最后调用 Application 的 onCreate 函数，做一些初始工作。

App 开发人员对 Application 非常熟悉，因为我们可以在其中写代码，进行一些全局的控制，所以我们通常认为 Application 是掌控全局的，其实 Application 的地位在 App 中并没有那么重要，它就是一个 Context 上下文，仅此而已。

App 中的灵魂是 ActivityThread，也就是主线程，只是这个类对于 App 开发人员是访问不到的，但使用反射是可以修改这个类的一些行为的。

创建新 App 的最后就是告诉 AMS“我启动好了”，同时把自己的 ActivityThread 对象发送给 AMS。从此以后，AMS 的电话簿中就多了这个新的 App 的登记信息，AMS 以后就通过这个 ActivityThread 对象，向这个 App 发送消息。

第 6 阶段：AMS 告诉新 App 启动哪个 Activity

AMS 把传入的 ActivityThread 对象转为一个 ApplicationThread 对象，用于以后和这个

App 跨进程通信。还记得 APT 和 ATP 的关系吗？参见图 2-7。

在第 1 阶段，Launcher 通知 AMS，要启动斗鱼 App 的哪个 Activity。在第 2 阶段，这个信息被 AMS 存下来。在第 6 阶段，AMS 从过去的记录中翻出来要启动哪个 Activity，然后通过 ATP 告诉 App。

第 7 阶段：启动斗鱼首页 Activity

毕其功于一役，尽在第 7 阶段。这是最后一步，过程如图 2-10 所示。

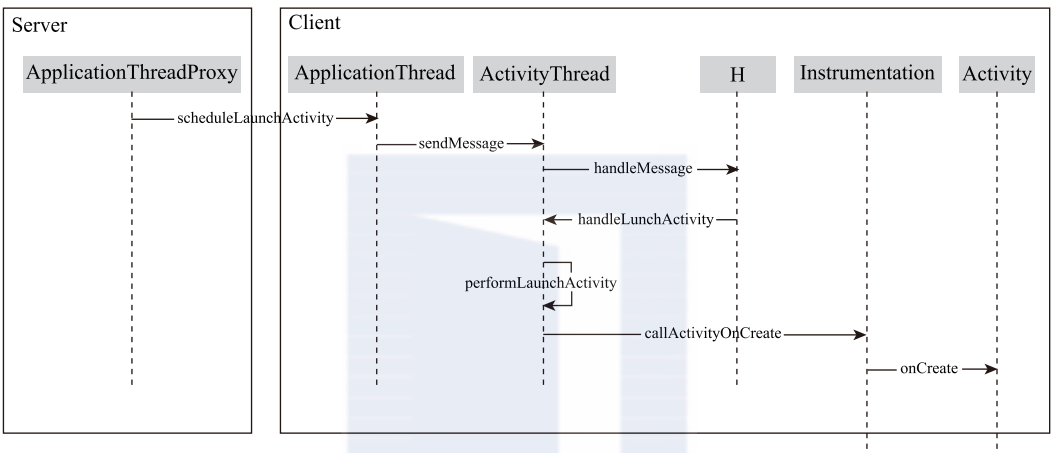


图 2-10　首页 Activity 最终被启动

在 Binder 的另一端，App 通过 APT 接收到 AMS 的消息，仍然在 H 的 handleMessage 方法的 switch 语句中处理，只不过，这次消息的类型是 LAUNCH_ACTIVITY：

```
public final class ActivityThread {
    private class H extends Handler {

    //省略一些代码
    public void handleMessage(Message msg) {
        if (DEBUG_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));
        switch (msg.what) {
            case LAUNCH_ACTIVITY: {
                Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "activityStart");
                final ActivityClientRecord r = (ActivityClientRecord) msg.obj;

                r.packageInfo = getPackageInfoNoCheck(
                    r.activityInfo.applicationInfo, r.compatInfo);
                handleLaunchActivity(r, null);
                Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
            } break;
```

```
        //省略一些代码
        }
    }
```

ActivityClientRecord 是什么？这是 AMS 传递过来的要启动的 Activity。

我们仔细看那个 getPackageInfoNoCheck 方法，这个方法会提取 apk 中的所有资源，然后设置 r 的 packageInfo 属性。这个属性的类型很有名，叫做 LoadedApk。注意，这个地方也是插件化技术渗入的一个点。

在 H 的这个分支中，又反过来回调 ActivityThread 的 handleLaunchActivity 方法（图 2-11），你一定会觉得很绕。其实我一直觉得，ActivityThread 和 H 合并成一个类也没问题。

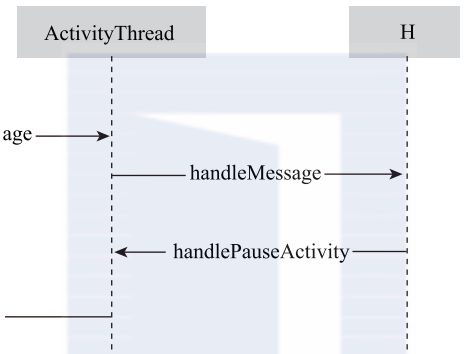


图 2-11 ActivityThread 与 H 的交互

重新看一下这个过程，每次都是 APT 执行 ActivityThread 的 sendMessage 方法，在这个方法中，把消息拼装一下，然后扔给 H 的 swicth 语句去分析，来决定要执行 ActivityThread 的那个方法。每次都是这样，习惯就好了。

handleLaunchActivity 方法都做哪些事呢？

1）通过 Instrumentation 的 newActivity 方法，创建要启动的 Activity 实例。

2）为这个 Activity 创建一个上下文 Context 对象，并与 Activity 进行关联。

3）通过 Instrumentation 的 callActivityOnCreate 方法，执行 Activity 的 onCreate 方法，从而启动 Activity。看到这里是不是很熟悉很亲切？

至此，App 启动完毕。这个流程经过了很多次握手，App 和 ASM 频繁地向对方发送消息，而发送消息的机制，是建立在 Binder 的基础之上的。

2.6 App 内部的页面跳转

在介绍完 App 的启动流程后，我们发现，其实就是启动一个 App 的首页。接下来我们看 App 内部页面的跳转。

从 ActivityA 跳转到 ActivityB，其实可以把 ActivityA 看作 Launcher，那么这个跳转过程和 App 的启动过程就很像了。有了前面的分析基础，你会发现这个过程不需要重新启动一个新的进程，所以可以省略 App 启动过程中的一些步骤，流程简化为：

1）ActivityA 向 AMS 发送一个启动 ActivityB 的消息。

2）AMS 保存 ActivityB 的信息，然后通知 App，“你洗洗睡吧”。

3）ActivityA 进入休眠，然后通知 AMS，“我休眠了（onPaused）。”

4）AMS 发现 ActivityB 所在的进程就是 ActivityA 所在的进程，所以不需要重新启动新的进程，它就会通知 App 启动 ActivityB。

5）App 启动 ActivityB。

为了更好地理解上述文字，可参考图 2-12。

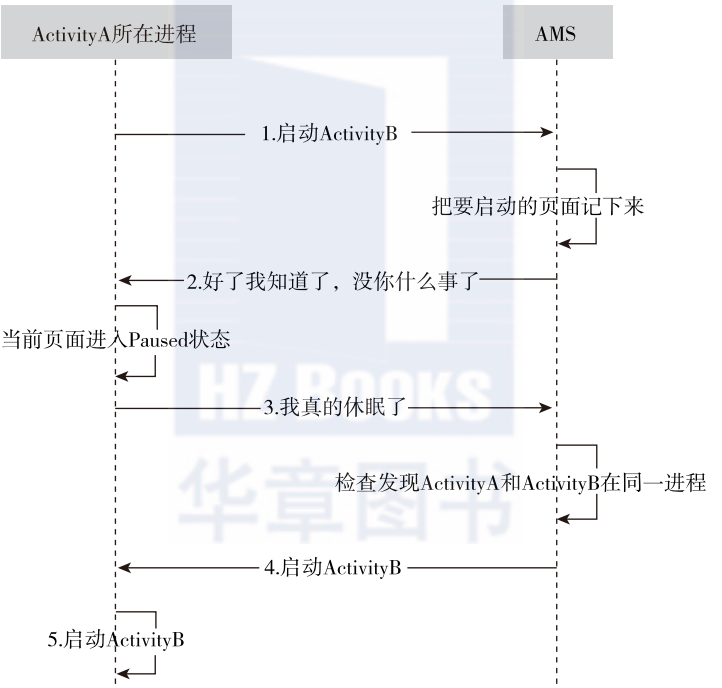


图 2-12　启动一个新的 Activity

整体流程此处不再赘述，和上一小节介绍的 App 启动流程是基本一致的。

以上的分析，仅限于 ActivityA 和 ActivityB 在相同的进程中，如果在 AndroidManifest 中指定不在同一个进程中的两个 Activity，那么就又是另一套流程了，但是整体流程大同小异。

2.7 Context 家族史

Activity、Service、Application 其实是亲戚关系，如图 2-13 所示。

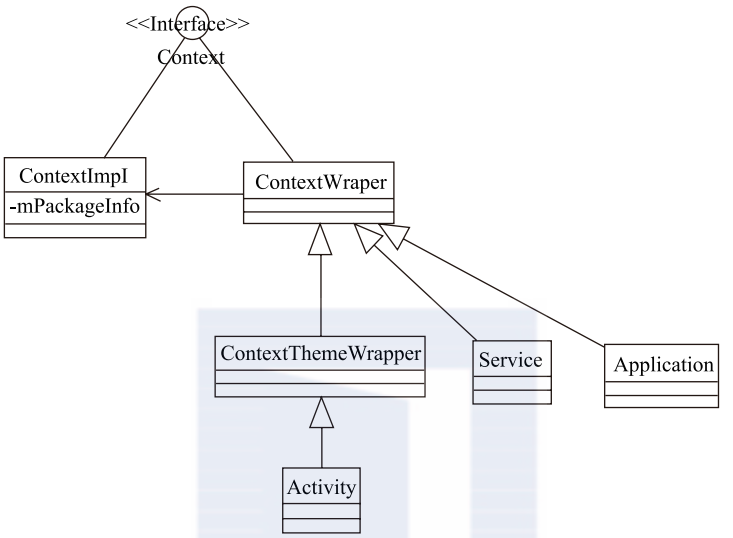


图 2-13 Context 家族

Activity 因为有了一层 Theme，所以中间有个 ContextThemeWrapper，相当于它是 Service 和 Application 的侄子。

ContextWrapper 只是一个包装类，没有任何具体的实现，真正的逻辑都在 ContextImpl 里面。

一个应用包含的 Context 个数＝Service 个数 +Activity 个数 +1(Application 类本身对应一个 Context 对象)。

应用程序中包含多个 ContextImpl 对象，而其内部变量 mPackageInfo 指向同一个 PackageInfo 对象。我们以 Activity 为例，看看 Activity 和 Context 的联系和区别。

我们知道，跳转到一个新的 Activity 要写如下代码：

```
btnNormal.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
           Intent intent = new Intent(Intent.Action.VIEW);
           intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
           intent.setData(Uri.parse("https://www.baidu.com"));
           startActivity(intent);
    }
});
```

我们还知道，也可以在 Activity 中使用 getApplicationContext 方法获取 Context 上下文

信息，然后使用 Context 的 startActivity 方法，启动一个新的 Activity：

```
btnNormal.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(Intent.Action.VIEW);
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        intent.setData(Uri.parse("https://www.baidu.com"));
        getApplicationContext().startActivity(intent);
    }
});
```

这二者的区别是什么？通过图 2-14 便可明了。

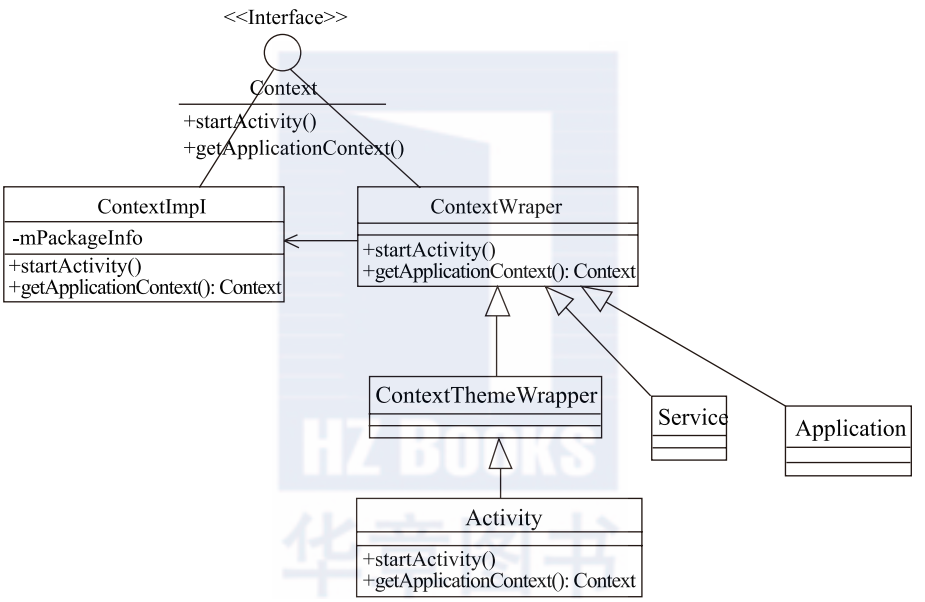


图 2-14　两种 startActivity

关于 Context 的 startActivity 方法，我看了在 ContextImpl 中的源码实现，仍然是从 ActivityThread 中取出 Instrumentation，然后执行 execStartActivity 方法，这和使用 Activity 的 startActivity 方法的流程是一样的。

还记得我们前面分析的 App 启动流程么？在第 5 阶段，创建 App 进程的时候，先创建的 ActivityThread，再创建的 Application。Application 的生命周期是跟整个 App 相关联的。而 getApplicationContext 得到的 Context，就是从 ActivityThread 中取出来的 Application 对象。代码如下：

```
class ContextImpl extends Context {
    @Override
    public void startActivity(Intent intent, Bundle options) {
```

```
            warnIfCallingFromSystemProcess();
            if ((intent.getFlags()&Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
                throw new AndroidRuntimeException(
                    "Calling startActivity() from outside of an Activity "
                    + " context requires the FLAG_ACTIVITY_NEW_TASK flag."
                    + " Is this really what you want?");
            }
            mMainThread.getInstrumentation().execStartActivity(
                    getOuterContext(), mMainThread.getApplicationThread(), null,
                    (Activity) null, intent, -1, options);
        }
    }
```

2.8 Service 工作原理

众所周知，Service 有两套流程，一套是启动流程，另一套是绑定流程，如图 2-15 所示。

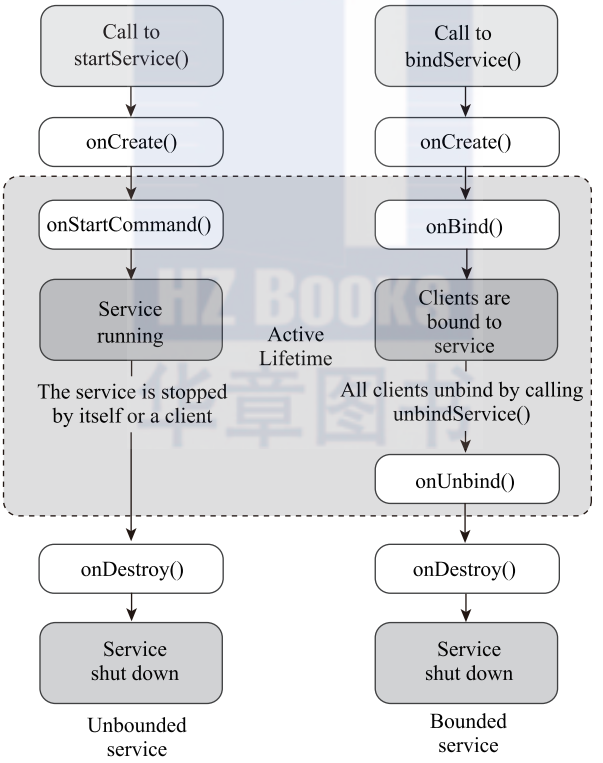


图 2-15 Service 的两套流程

2.8.1 在新进程启动 Service

我们先看 Service 启动过程，假设要启动的 Service 是在一个新的进程中，启动过程可

分为 5 个阶段：

1）App 向 AMS 发送一个启动 Service 的消息。

2）AMS 检查启动 Service 的进程是否存在，如果不存在，先把 Service 信息存下来，然后创建一个新的进程。

3）新进程启动后，通知 AMS，“我可以啦”。

4）AMS 把刚才保存的 Service 信息发送给新进程。

5）新进程启动 Service。

我们详细分析这 5 个阶段。

第 1 阶段：App 向 AMS 发送一个启动 Service 的消息

和 Activity 非常像，仍然是通过 AMM/AMP 把要启动的 Service 信息发送给 AMS，如图 2-16 所示。

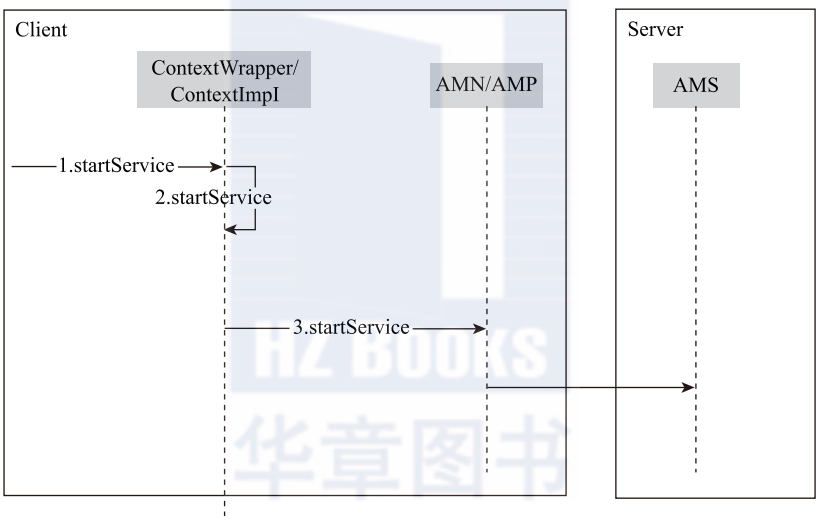


图 2-16　App 向 AMS 发送一个启动 Service 的消息

第 2 阶段：AMS 创建新的进程

AMS 检查 Service 是否在 AndroidManifest 中声明了，若没声明则会直接报错。AMS 检查启动 Service 的进程是否存在，如果不存在，先把 Service 信息存下来，然后创建一个新的进程。在 AMS 中，每个 Service，都使用 ServiceRecord 对象来保存。

第 3 阶段：新进程启动后，通知 AMS，“我可以啦。”

Service 所在的新进程启动的过程，与前面介绍 App 启动时的过程相似。

新进程启动后，也会创建新的 ActivityThread，然后把 ActivityThread 对象通过 AMP 传递给 AMS，告诉 AMS，“新进程启动成功了”。

第 4 阶段：AMS 把刚才保存的 Service 信息发送给新进程

AMS 把传进来的 ActivityThread 对象改造为 ApplicationThreadProxy，也就是 ATP，通过 ATP 把要启动的 Service 信息发送给新进程。

第 5 阶段：新进程启动 Service

新进程启动 Service 过程如图 2-17 所示。

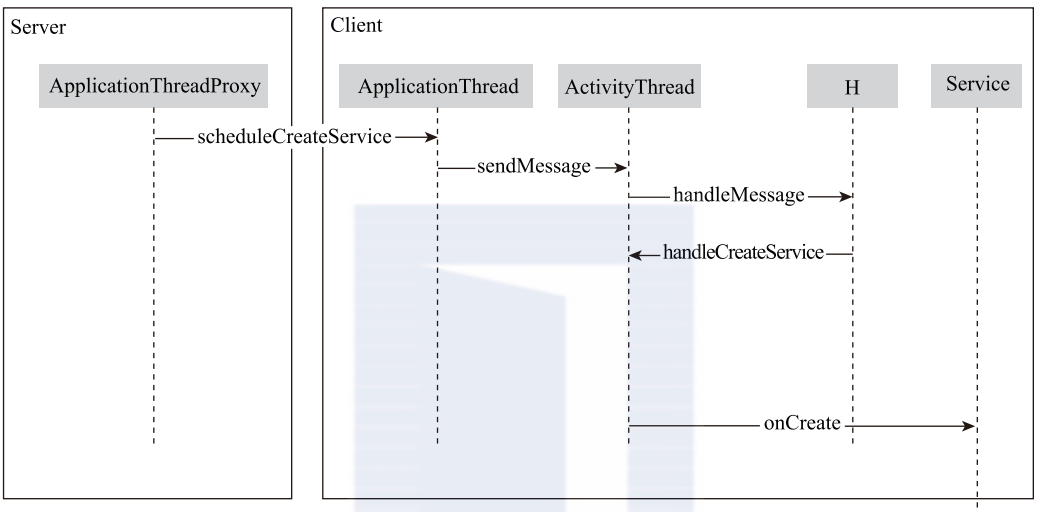


图 2-17 启动一个 Service

新进程通过 ApplicationThread 接收到 AMS 的信息，和前面介绍的启动 Activity 的最后一步相同，借助 ActivityThread 和 H，执行 Service 的 onCreate 方法。在此期间，为 Service 创建了 Context 上下文对象，并与 Service 相关联。

需要重点关注的是 ActivityThread 的 handleCreateService 方法，代码如下：

```
private void handleCreateService(CreateServiceData data) {
    LoadedApk packageInfo = getPackageInfoNoCheck(
        data.info.applicationInfo, data.compatInfo);
    Service service = null;
    try {
        java.lang.ClassLoader cl = packageInfo.getClassLoader();
        service = (Service) cl.loadClass(data.info.name).newInstance();
    }

    //省略一些代码
    }
```

你会发现，这段代码和前面介绍的 handleLaunchActivity 差不多，都是从 PMS 中取出包的信息 packageInfo，这是一个 LoadedApk 对象，然后获取它的 classloader，反射出来一个类的对象，在这里反射的是 Service。

四大组件的逻辑都是如此，所以我们要做插件化，可以在这里做文章，换成插件的 classloader，加载插件中的四大组件。

至此，我们在一个新的进程中启动了一个 Service。

2.8.2　启动同一进程的 Service

如果是在当前进程启动这个 Service，那么上面的步骤就简化为：

1）App 向 AMS 发送一个启动 Service 的消息。

2）AMS 例行检查，比如 Service 是否声明了，把 Service 在 AMS 这边注册。AMS 发现要启动的 Service 就是 App 所在的 Service，于是通知 App 启动这个 Service。

3）App 启动 Service。

我们看到，没有了启动新进程的过程。

2.8.3　在同一进程绑定 Service

如果要在当前进程绑定这个 Service，可分为以下 5 个阶段：

1）App 向 AMS 发送一个绑定 Service 的消息。

2）AMS 例行检查，比如 Service 是否声明了，把 Service 在 AMS 这边注册。AMS 发现要启动的 Service 就是 App 所在的 Service，就先通知 App 启动这个 Service，然后再通知 App 对 Service 进行绑定操作。

3）App 收到 AMS 第 1 个消息，启动 Service。

4）App 收到 AMS 第 2 个消息，绑定 Service，并把一个 Binder 对象传给 AMS。

5）AMS 把接收到的 Binder 对象发送给 App。

你也许会问，都在一个进程，App 内部直接使用 Binder 对象不就好了，其实，要考虑不在一个进程的场景，代码又不能写两份，两套逻辑，所以就都放在一起了，即使在同一个进程，也要绕着 AMS 走一圈。接下来，我们详细分析一下这 5 个阶段。

第 1 阶段：App 向 AMS 发送一个绑定 Service 的消息

具体过程如图 2-18 所示。

第 2 阶段：AMS 创建新的进程

AMS 检查 Service 是否在 AndroidManifest 中声明了，没声明会直接报错。

AMS 检查启动 Service 的进程是否存在，如果不存在，先把 Service 信息存下来，然后创建一个新的进程。在 AMS 中，每个 Service，都使用 ServiceRecord 对象来保存。

第 3 阶段：新进程启动后，通知 AMS，“我可以啦”

Service 所在的新进程启动的过程，就和前面介绍的 App 启动时的过程差不多。

新进程启动后，也会创建新的 ActivityThread，然后把 ActivityThread 对象通过 AMP 传递给 AMS，告诉 AMS，“新进程启动成功了”。

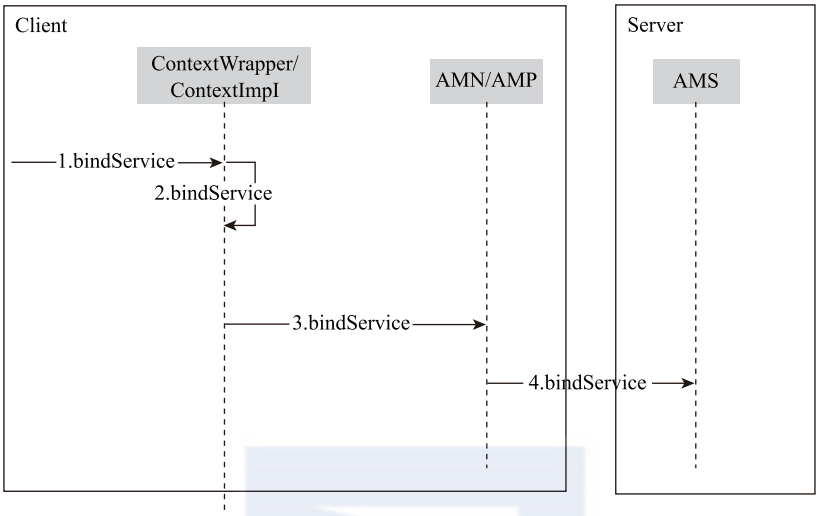


图 2-18 App 向 AMS 发送一个绑定 Service 的消息

第 4 阶段：处理第 2 个消息，绑定 Service

具体过程如图 2-19 所示。

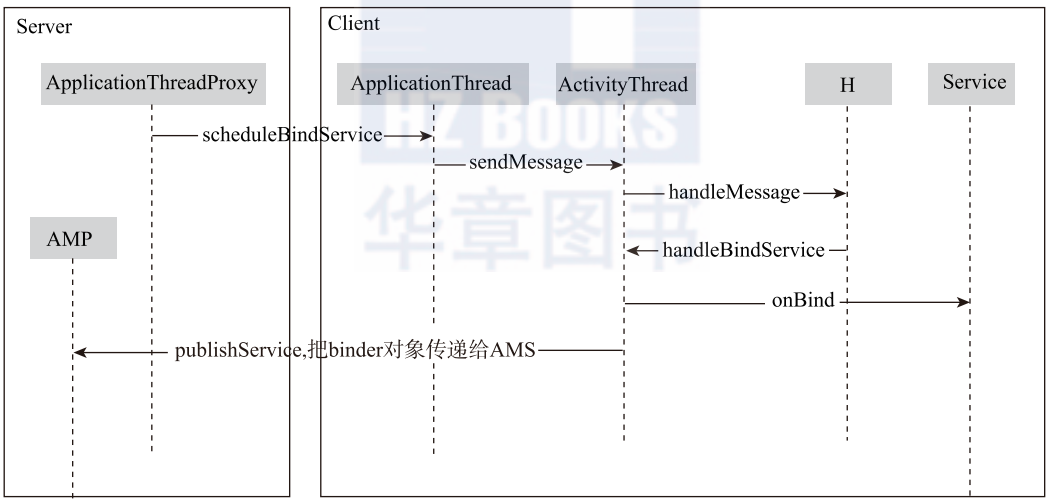


图 2-19 处理第 2 个消息

第 5 阶段：把 Binder 对象发送给 App

这一步是要仔细说的，因为 AMS 把 Binder 对象传给 App，这里没用 ATP 和 APT，而是利用 AIDL 来实现的，这个 AIDL 的名字是 IServiceConnection，如图 2-20 所示。

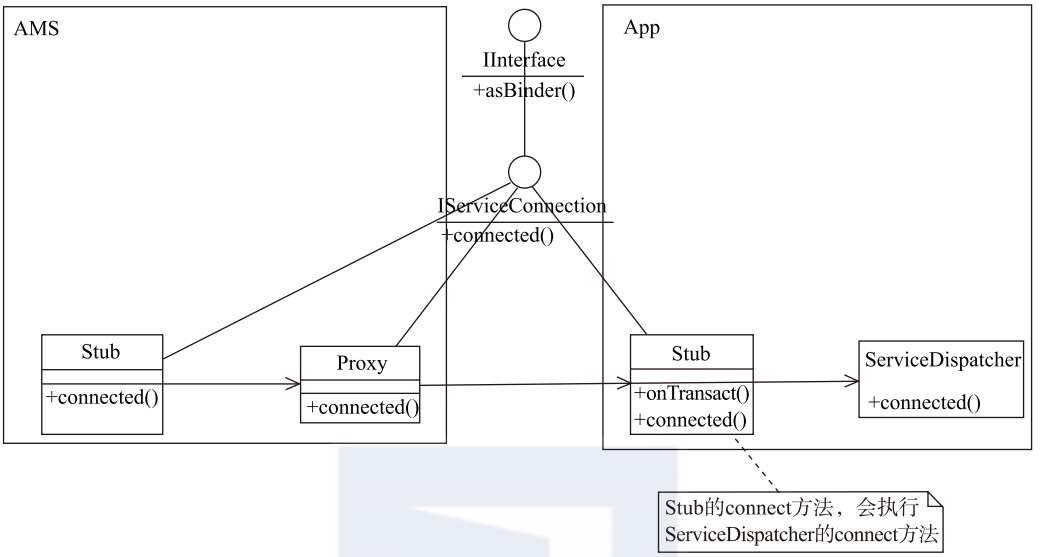


图 2-20　AIDL 原理

ServiceDispatcher 的 connect 方法最终会调用 ServiceConnection 的 onServiceConnected 方法，这个方法我们已经很熟悉了。App 开发人员在这个方法中拿到 connection，就可以做自己的事情了。

好了，关于 Service 的底层知识，我们就全都介绍完了。当你再去编写一个 Service 时，一定能感到对这个组件理解得更透彻了。

2.9　BroadcastReceiver 工作原理

BroadcastReceiver 就是广播，简称 Receiver。

很多 App 开发人员表示，从来没用过 Receiver。其实，对于音乐播放类 App，Service 和 Receiver 用的还是很多的，如果你用过 QQ 音乐，App 退到后台，音乐照样播放不会停止，这就是 Service 在后台起的作用。

在前台的 Activity，点击“停止”按钮，就会给后台 Service 发送一个 Receiver，通知它停止播放音乐；点击“播放”按钮，仍然发送这个 Receiver，只是携带的值变了，所以 Service 收到请求后播放音乐。

反过来，后台 Service 每播放完一首音乐，接下来准备播放下一首音乐的时候，就会给前台 Activity 发 Receiver，让 Activity 显示下一首音乐的名称。

所以音乐播放器的原理，就是一个前后台 Activity 和 Service 互相发送和接收 Receiver 的过程，如图 2-21 所示。

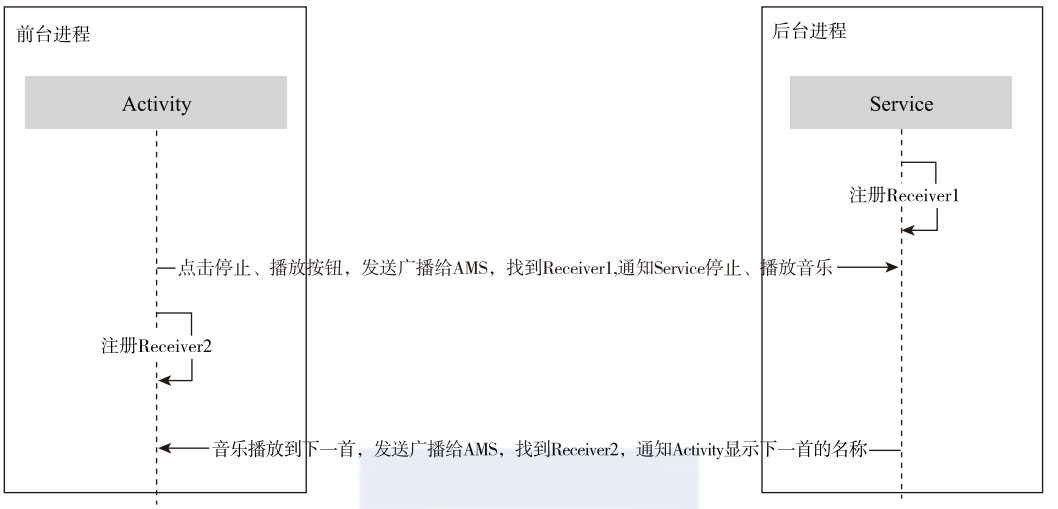


图 2-21 音乐播放器的两个 Receiver

Receiver 分静态广播和动态广播两种。

在 AndroidManifest 中声明的 Receiver，是静态广播：

```
<receiver android:name=".MyReceiver">
    <intent-filter>
        <action android:name="baobao" />
    </intent-filter>
</receiver>
```

在程序中手动写注册代码的是动态广播：

```
activityReceiver = new ActivityReceiver();
IntentFilter filter = new IntentFilter();
Filter.addAction(UPDATE_ACTION);
registerReceiver(activityReceiver, filter);

Intent intent = new Intent(this, MyService.class);
startService(intent);
```

二者具有相同的功能，只是写法不同。既然如此，我们就可以把所有静态广播都改为动态广播，这就避免在 AndroidManifest 文件中声明了，也避免了 AMS 检查。你想到什么？对，Receiver 的插件化解决方案就是这个思路。

接下来我们看 Receiver 是怎么和 AMS 打交道的，分为两部分：一是注册，二是发送广播。

你只有注册了这个广播，发送这个广播时，才能通知你执行 onReceive 方法。

我们就以音乐播放器为例，在 Activity 注册 Receiver，在 Service 发送广播。Service 播放下一首音乐时，会通知 Activity 修改当前正在播放的音乐名称。

2.9.1　注册过程

注册过程如下：

1）在 Activity 中，注册 Receiver，并通知 AMS，如图 2-22 所示。

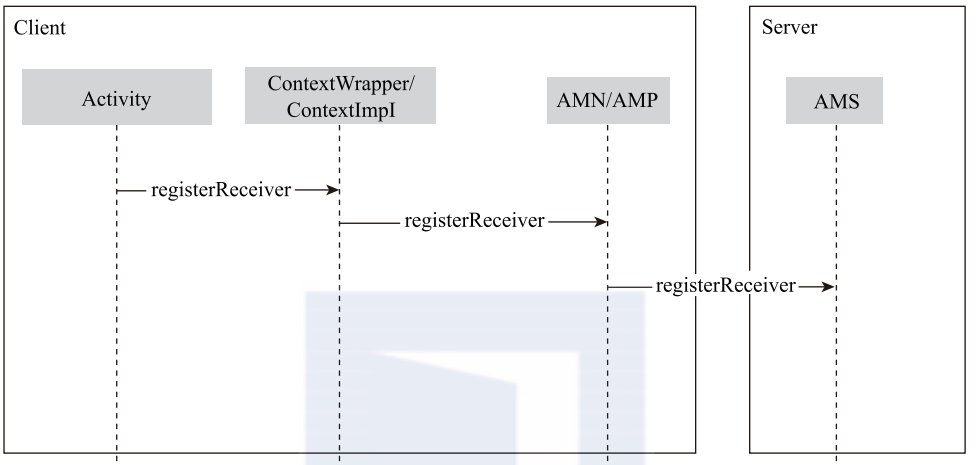


图 2-22　注册 Receiver 的流程

这里 Activity 使用了 Context 提供的 registerReceiver 方法，然后通过 AMN/AMP，把一个 Receiver 传给 AMS。在创建这个 Receiver 对象的时候，需要为 Receiver 指定 IntentFilter，这个 filter 就是 Receiver 的身份证，用来描述 Receiver。

在 Context 的 registerReceiver 方法中，它会使用 PMS 获取到包的信息，也就是 LoadedApk 对象。就是这个 LoadedApk 对象，它的 getReceiverDispatcher 方法，将 Receiver 封装成一个实现了 IIntentReceiver 接口的 Binder 对象。

我们就是将这个 Binder 对象和 filter 传递给 AMS。但只传递 Receiver 给 AMS 是不够的，当发送广播时，AMS 不知道该发给谁，所以 Activity 所在的进程还要把自身对象也发送给 AMS。

2）AMS 收到消息后，就会把上面这些信息，存在一个列表中，这个列表中保存了所有的 Receiver。

注意，这里都是在注册动态 Receiver。静态 Receiver 什么时候注册到 AMS 呢？是在 App 安装的时候。PMS 会解析 AndroidManifest 中的四大组件信息，把其中的 Receiver 存起来。

动态 Receiver 和静态 Receiver 分别存在 AMS 不同的变量中，在发送广播的时候，会把两种 Receiver 合并到一起，然后依次发送。其中动态的排在静态的前面，所以动态 Receiver 永远优先于静态 Receiver 收到消息。

此外，Android 系统每次启动的时候，也会把静态广播接收者注册到 AMS。因为 Android 系统每次启动时，都会重新安装所有的 apk，详细流程我们会在后面 PMS 的相关

章节看到。

2.9.2 发送广播的流程

发送广播的流程大致有三个步骤：

1）在 Service 中，通过 AMM/AMP，发送广播给 AMS，广播中携带着 filter。

2）AMS 收到这个广播后，在 Receiver 列表中，根据 filter 找到对应的 Receiver，可能是多个，把它们都放到一个广播队列中。最后向 AMS 的消息队列发送一个消息。

当消息队列中的这个消息被处理时，AMS 就从广播队列中找到合适的 Receiver，向广播接收者所在的进程发送广播。

3）Receiver 所在的进程收到广播，并没有把广播直接发给 Receiver，而是将广播封装成一个消息，发送到主线程的消息队列中，当这个消息被处理时，才会把这个消息中的广播发送给 Receiver。

下面通过图 2-23，仔细看一下这三个步骤：

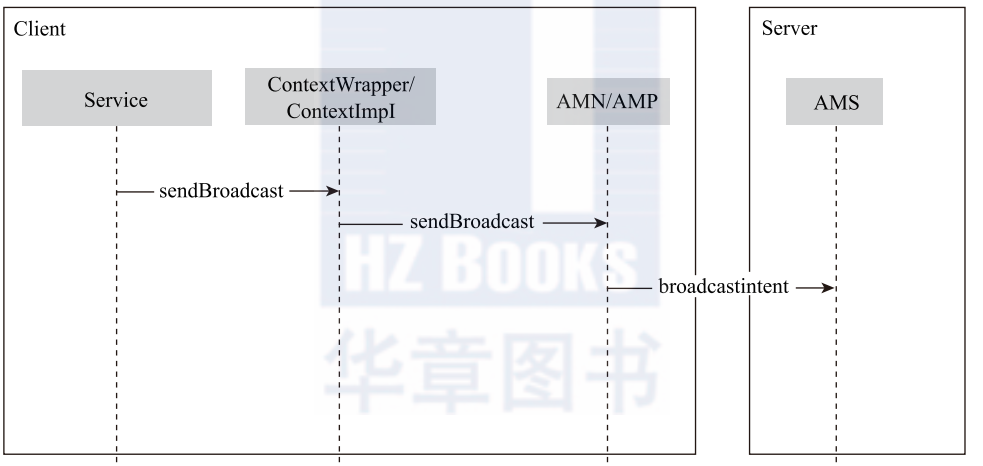


图 2-23 Service 发送广播给 AMS

第 1 步：Service 发送广播给 AMS

发送广播，是通过 Intent 这个参数携带了 filter，从而告诉 AMS 什么样的 Receiver 能接收这个广播。

第 2 步：AMS 接收广播，发送广播

接收广播和发送广播是不同步的。AMS 每接收到一个广播，就把它扔到广播发送队列中，至于发送是否成功，它就不管了。

因为 Receiver 分为无序 Receiver 和有序 Receiver，所以广播发送队列也分为两个，分别发送这两种广播。

AMS 发送广播给客户端，这又是一个跨进程通信，还是通过 ATP 把消息发给 APT。因为要传递 Receiver 这个对象，所以它也是一个 Binder 对象才可以传过去。我们前面说过，在把 Receiver 注册到 AMS 的时候，会把 Receiver 封装为一个 IIntentReceiver 接口的 Binder 对象。那么接下来，AMS 就是把这个 IIntentReceiver 接口对象传回来。

第 3 步：App 处理广播

处理流程如图 2-24 所示：

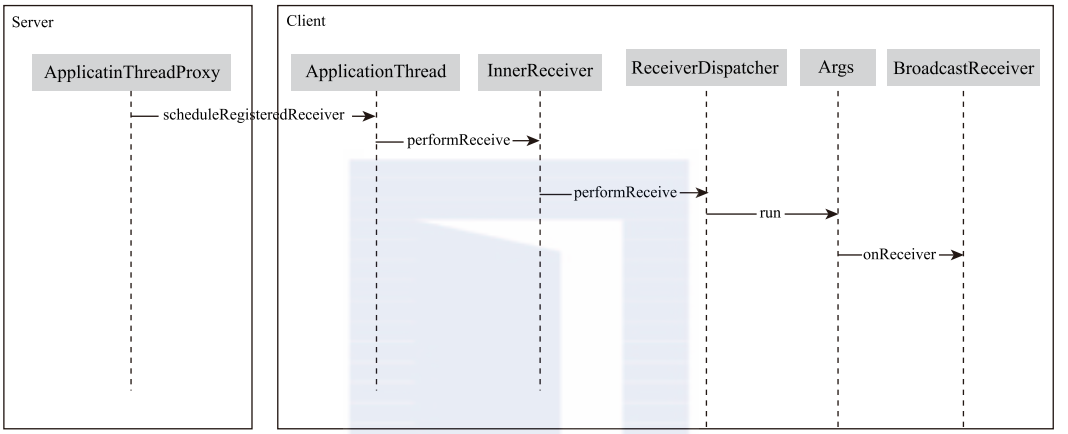


图 2-24　App 处理广播

1）消息从 AMS 传到客户端，把 AMS 中的 IIntentReceiver 对象转为 InnerReceiver 对象，这就是 Receiver，这是一个 AIDL 跨进程通信。

2）然后在 ReceiverDispatcher 中封装一个 Args 对象（这是一个 Runnable 对象，要实现 run 方法），包括广播接收者所需要的所有信息，交给 ActivityThread 来发送。

3）接下来要做的就是我们所熟悉的了，ActivityThread 把 Args 消息扔到 H 这个 Hanlder 中，向主线程消息队列发送消息。等到执行 Args 消息的时候，自然是执行 Args 的 run 方法。

4）在 Args 的 run 方法中，实例化一个 Receiver 对象，调用它的 onReceiver 方法。

5）最后，在 Args 的 run 方法中，随着 Receiver 的 onReceiver 方法调用结束，会通过 AMN/AMP 发送一个消息给 AMS，告诉 AMS“广播发送成功了”。AMS 得到通知后，就发送广播给下一个 Receiver。

InnerReceiver 是 IIntentReceiver 的 stub，是 Binder 对象的接收端。

2.9.3　广播的种类

Android 广播按发送方式分为三种：无序广播、有序广播（OrderedBroadcast）和粘性广

播（StickyBroadcast）。

1）无序广播是最普通的广播。

2）有序广播区别于无序广播，就在于它可以指定优先级。

这两种 Receiver 在 AMS 不同的变量中，可以认为是两个 Receiver 集合发送不同类别的广播。

3）粘性广播是无序广播的一种。我们平常见的不多，但我说一个场景你就明白了，那就是电池电量。当电量小于 20% 的时候，就会提示用户。而获取电池的电量信息，就是通过广播来实现的。但是一般的广播，发完就完了。我们需要有这样一种广播，发出后，还能一直存在，未来的注册者也能收到这个广播，这种广播就是粘性广播。

由于动态 Receiver 只有在 Activity 的 onCreate() 方法调用时才能注册再接收广播，所以当程序没有运行就不能收到广播；但是静态注册的则不依赖于程序是否处于运行状态。

至此，关于广播的所有概念就全都介绍完了，虽然本节列出的代码很少，但我希望上述文字能引导 App 开发人员进入一个神奇的世界。

2.10 ContentProvider 工作原理

ContentProvider，简称 CP。App 开发人员，尤其是电商类 App 开发人员，对 ContentProvider 并不熟悉，对这个概念的最大程度的了解，也仅仅是建立在书本上，它是 Android 四大组件中的一个。开发系统管理类 App，比如手机助手，则有机会频繁使用 ContentProvider。

而对于应用类 App，数据通常存在服务器端，当其他应用类 App 也想使用时，一般都是从服务器取数据，所以没机会使用到 ContentProvider。

有时候我们会在自己的 App 中读取通讯录或者短信数据，这时候就需要用到 ContentProvider 了。通讯录或者短信数据，是以 ContentProvider 的形式提供的，我们在 App 这边，是使用方。

对于应用类 App 开发人员，很少有机会自定义 ContentProvider 供其他 App 使用。

我们快速回顾一下在 App 中怎么使用 ContentProvider。

1）定义 ContentProvider 的 App1。

在 App1 中定义一个 ContentProvider 的子类 MyContentProvider，并在 AndroidManifest 中声明，为此要在 MyContentProvider 中实现 ContentProvider 的增删改查 4 个方法：

```
<provider
    android:name=".MyContentProvider"
        android:authorities="baobao"
        android:enabled="true"
```

```
        android:exported="true"></provider>

public class MyContentProvider extends ContentProvider {
    public MyContentProvider() {
    }

    @Override
    public boolean onCreate() {
        //省略一些代码
    }

    @Override
    public String getType(Uri uri) {
        //省略一些代码
    }

    @Override
    public Uri insert(Uri uri, ContentValues values) {
        //省略一些代码
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String where,
                        String[] whereArgs, String sortOrder){
        //省略一些代码
    }

    @Override
    public int delete(Uri uri, String where, String[] whereArgs) {
        //省略一些代码
    }

    @Override
    public int update(Uri uri, ContentValues values, String where,
                      String[] whereArgs){
        //省略一些代码
    }
}
```

2）使用 ContentProvider 的 App2。

在 App2 访问 App1 中定义的 ContentProvider，为此，要使用 ContentResolver（如图 2-25 所示），它也提供了增删改查 4 个方法，用于访问 App1 中定义的 ContentProvider：

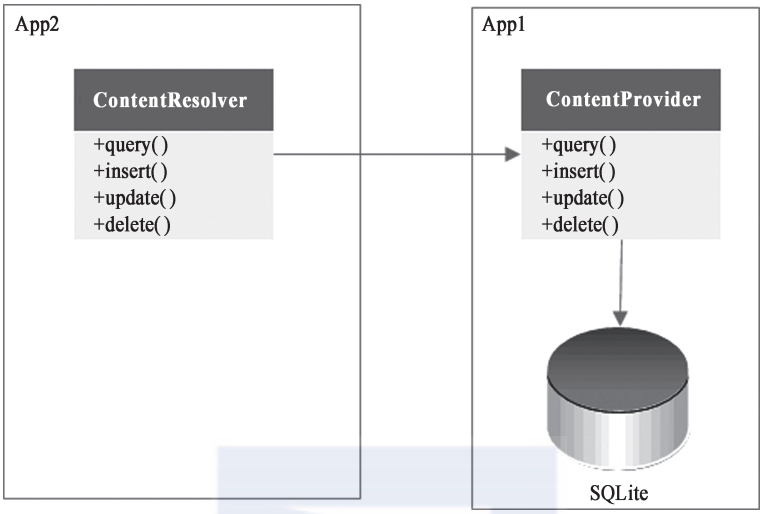


图 2-25 App2 访问 App1 提供的 ContentProvider

```
public class MainActivity extends Activity {

    ContentResolver contentResolver;
    Uri uri;

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

        uri = Uri.parse("content://baobao/");
        contentResolver = getContentResolver();
    }

    public void delete(View source) {
        int count = contentResolver.delete(uri, "delete_where", null);
        Toast.makeText(this, "delete uri:" + count, Toast.LENGTH_LONG).show();
    }

    public void insert(View source) {
        ContentValues values = new ContentValues();
        values.put("name", "jianqiang");
        Uri newUri = contentResolver.insert(uri, values);
        Toast.makeText(this, "insert uri:" + newUri, Toast.LENGTH_LONG).show();
    }

    public void update(View source) {
        ContentValues values = new ContentValues();
        values.put("name", "jianqiang2");
        int count = contentResolver.update(uri, values, "update_where", null);
```

```
            Toast.makeText(this, "update count:" + count, Toast.LENGTH_LONG).show();
        }
    }
```

首先，我们看一下 ContentResolver 的增删改查这 4 个方法的底层实现，其实都是和 AMS 通信，最终调用 App1 的 ContentProvider 的增删改查 4 个方法，后面我们会讲到这个流程是怎么样的。

其次，URI 是 ContentProvider 的唯一标识。我们在 App1 中为 ContentProvider 声明 URI，也就是 authorities 的值为 baobao，那么在 App2 中想使用它，就在 ContentResolver 的增删改查 4 个方法中指定 URI，格式为：

```
uri = Uri.parse("content://baobao/");
```

接下来把两个 App 都进入 debug 模式，就可以从 App2 调试进入 App1 了，比如，query 操作。

2.10.1　ContentProvider 的本质

ContentProvider 的本质是把数据存储在 SQLite 数据库中。

不同的数据源具有不同的格式，比如短信、通讯录，它们在 SQLite 中就是不同的数据表，但是对外界的使用者而言，就需要封装成统一的访问方式，比如对于数据集合而言，必须提供增删改查 4 个方法，于是我们在 SQLite 之上封装了一层，也就是 ContentProvider。

2.10.2　匿名共享内存（ASM）

ContentProvider 读取数据使用了匿名共享内存（ASM），所以你看上面 ContentProvider 和 AMS 通信忙得不亦乐乎，其实下面别有一番风景。

ASM 实质上也是个 Binder 通信，如图 2-26 所示。

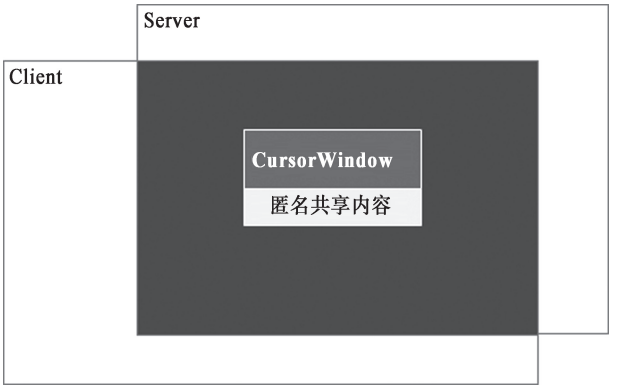


图 2-26　匿名共享内存的架构图

图 2-27 为 ASM 的类的交互关系图。

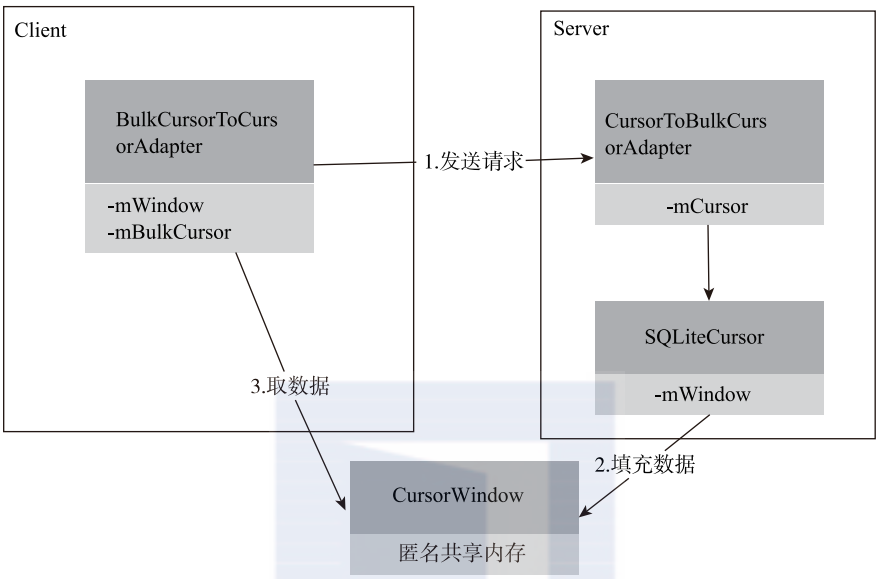


图 2-27 匿名共享内存的类图

这里的 CursorWindow 就是匿名共享内存。这个流程简单来说包含 3 个步骤：

1）Client 内部有一个 CursorWindow 对象，发送请求的时候，把这个 CursorWindow 类型的对象传过去，这个对象暂时为空。

2）Server 收到请求，搜集数据，填充到这个 CursorWindow 对象中。

3）Client 读取内部的这个 CursorWindow 对象，获取数据。

由此可见，这个 CursorWindow 对象就是匿名共享内存，这是同一块匿名内存。

举个生活中的例子，你定牛奶，在你家门口放个箱子，送牛奶的人每天早上往这个箱子里放一瓶牛奶，你睡醒了去箱子里取牛奶。这个牛奶箱就是匿名共享内存。

2.10.3 ContentProvider 与 AMS 的通信流程

接下来我们看一下 ContentProvider 是怎么和 AMS 通信的。

还是以 App2 想访问 App1 中定义的 ContentProvider 为例。我们仅看 ContentProvider 的 insert 方法：

```
ContentResolver contentResolver = getContentResolver ();
Uri uri = Uri.parse(“content://baobao/”);

ContentValues values = new ContentValues();
values.put(“name”, “jianqiang”);
Uri newUri = contentResolver.insert(uri, values);
```

上面这 5 行代码，包括了启动 ContentProvider 和执行 ContentProvider 方法两部分，分水岭在 insert 方法，insert 方法的实现，前半部分仍然是在启动 ContentProvider，当 ContentProvider 启动后获取到 ContentProvider 的代理对象，后半部分便通过代理对象调用 insert 方法。

整体的流程如图 2-28 所示。

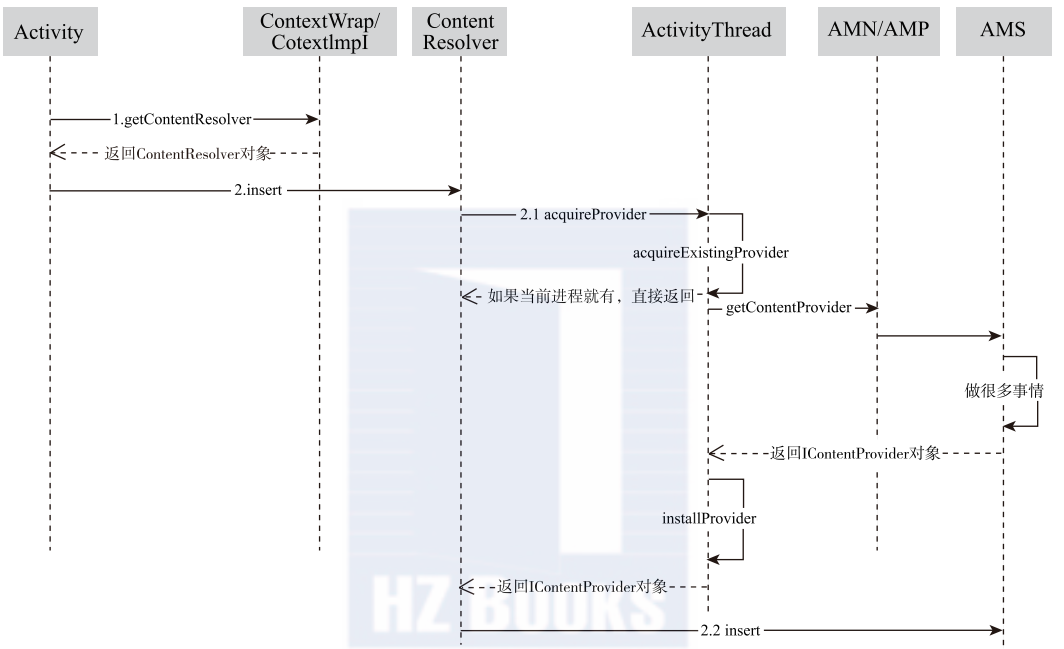


图 2-28　ContentProvider 与 AMS 的通信流程

1）App2 发送消息给 AMS，想要访问 App1 中的 ContentProvider。

2）AMS 检查发现，App1 中的 ContentProvider 没启动过，为此新开一个进程，启动 App1，然后获取 App1 启动的 ContentProvider，把 ContentProvider 的代理对象返回给 App2。

3）App2 拿到 ContentProvider 的代理对象，也就是 IContentProvider，就调用它的增删改查 4 个方法。接下来使用 ASM 传输数据或者修改数据了，也就是上面提到的 CursorWindow 这个类，取得数据或者操作结果即可，作为 App 的开发人员，可以不必知道太多底层的详细信息。

至此，关于 ContentProvider 的介绍就结束了。下一小节，我们讨论 App 的安装流程，也就 PMS。

2.11 PMS 及 App 安装过程

2.11.1 PMS 简介

PackageManagerService（PMS）是用来获取 apk 包的信息的。

在前面分析四大组件与 AMS 通信的时候，我们介绍过，AMS 总是会使用 PMS 加载包的信息，将其封装在 LoadedApk 这个类对象中，然后我们就可以从中取出在 AndroidManifest 声明的四大组件信息了。

在下载并安装 App 的过程中，会把 apk 存放在 data/app 目录下。

apk 是一个 zip 压缩包，在文件头会记录压缩包的大小，所以就算后续在文件尾处追加一部小电影，也不会对解压造成影响——木马其实就是这个思路，在可执行文件 exe 尾巴上挂一个木马病毒，执行 exe 的同时也会执行这个木马，然后你就中招了。

我们可以把这一思想运用在 Android 多渠道打包上。在比较老的 Android 4.4 版本中，我们会在 apk 尾巴上追加几个字节，来标志 apk 的渠道。apk 启动的时候，从 apk 中的尾巴上读取这个渠道值。

后来 Google 也发现这个安全漏洞了，在新版本的系统中，会在 apk 安装的时候，检查 apk 的实际大小，看这个值与 apk 的头部记录的压缩包大小是否相等，不相等就会报错说安装失败。

回答前面提及的一个问题：为什么 App 安装时，不把它解压呢？直接从解压文件中读取资源文件（比如图片）是不是更快呢？其实并不是这样的，这部分逻辑需要到底层 C++ 的代码去寻找，我没有具体看过，只是道听途说问过 Lody，他是这么给我解释的：

每次从 apk 中读取资源，并不是先解压再找图片资源，而是解析 apk 中的 resources.arsc 文件，这个文件中存储着资源的所有信息，包括资源在 apk 中的地址、大小等等，按图索骥，从这个文件中快速找到相应的资源文件。这是一种很高效的算法。

不解压 apk 的好处自然是节省空间。

2.11.2 App 的安装流程

Android 系统使用 PMS 解析这个 apk 中的 AndroidManifest 文件，包括：

- 四大组件的信息，比如，前面讲过的静态 Receiver，默认启动的 Activity。
- 分配用户 Id 和用户组 Id。用户 Id 是唯一的，因为 Android 是一个 Linux 系统。用户组 Id 指的是各种权限，每个权限都在一个用户组中，比如读写 SD 卡或网络访问，分配了哪些用户组 Id，就拥有了哪些权限。
- 在 Launcher 生成一个 icon，icon 中保存着默认启动的 Activity 的信息。
- 在 App 安装过程的最后，把上面这些信息记录在一个 xml 文件中，以备下次安装时再次使用。

在 Android 手机系统每次启动的时候，都会使用 PMS，把 Android 系统中的所有 apk 都安装一遍，一共 4 个步骤，如图 2-29 所示。

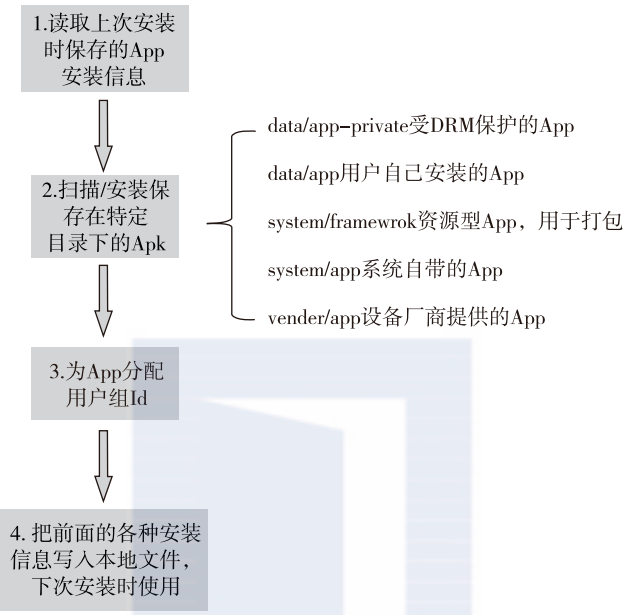


图 2-29　App 的安装流程

第 1 步，因为结束安装的时候，都会把安装信息保存在 xml 文件中，所以 Android 系统再次启动时，会重新安装所有的 apk，就可以直接读取之前保存的 xml 文件了。

第 2 步，从 5 个目录中读取并安装所有的 apk。

第 3 步和第 4 步，与单独安装一个 App 的步骤是一样的，不再赘述。

2.11.3　PackageParser

Android 系统重启后，会重新安装所有的 App，这是由 PMS 这个类完成的。App 首次安装到手机上，也是由 PMS 完成的。PMS 是 Android 的系统进程，我们是不能 Hook 的。

PMS 中有一个类，对于我们 App 开发人员来说很重要，那就是 PackageParser，其类图如图 2-30 所示，它是专门用来解析 AndroidManifest 文件的，以获取四大组件的信息以及用户权限。

PackageParser 类中，有一个 parsePackage 方法，接收一个 apkFile 的参数，既可以是当前 apk 文件，也可以是外部 apk 文件。我们可以使用这个类，来读取插件 apk 的 Android Manifest 文件中的信息，但是 PackageParser 是隐藏的不对 App 开发人员开放，但这并不是什么难题，可以通过反射来获取到这个类。

parsePackage 方法返回的是 Package 类型的实体对象，里面存储着四大组件的信息，但

Package 类型我们一般用不到，取而代之的是 PackageInfo 对象，所以要使用 PackageParser 类的 generatePackageInfo 方法，把 Package 类型转换为 PackageInfo 类型。

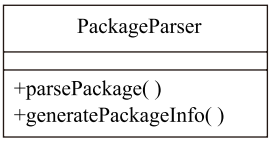


图 2-30 PackageParser 的类图

在插件化编程中，我们反射 PackageParser，一般是用来获取插件 AndroidManifest 中的四大组件信息。

2.11.4 ActivityThread 与 PackageManager

条条大路通罗马，对于 App 开发人员而言，也可以使用 Context.getPackageManager() 方法，来获取当前 Apk 的信息。

getPackageManager 的具体实现自然是在 ContextImpl 类中：

```
class ContextImpl extends Context {
    private PackageManager mPackageManager;

    @Override
    public PackageManager getPackageManager() {
        if (mPackageManager != null) {
            return mPackageManager;
        }

        IPackageManager pm = ActivityThread.getPackageManager();
        if (pm != null) {
            // Doesn't matter if we make more than one instance.
            return (mPackageManager = new ApplicationPackageManager(this, pm));
        }

        return null;
    }
}
```

从上面代码看出，Context 的 getPackageManager 方法返回的是 ApplicationPackageManager 类型的对象。ApplicationPackageManager 是 PackageManager 的子类。

如果读者熟悉设计模式，那么能看出 ApplicationPackageManager 其实是一个装饰器模式。它就是一个壳，它装饰了 ActivityThread.getPackageManager()，真正的逻辑也是在 ActivityThread 中完成的，如下所示：

```
public final class ActivityThread {
    private static ActivityThread sCurrentActivityThread;
```

```
    static IPackageManager sPackageManager;

    public static IPackageManager getPackageManager() {
        if (sPackageManager != null) {
            return sPackageManager;
        }
        IBinder b = ServiceManager.getService("package");
        sPackageManager = IPackageManager.Stub.asInterface(b);
        return sPackageManager;
    }
}
```

还记得我们前面介绍 Binder 时讲的 ServiceManager 吗？这就是一个字典容器，里面存放着各种系统服务，key 为 clipboard 时，对应的是剪切板；key 为 package 时，对应的是 PMS。

IPackageManager 是一个 AIDL。根据前面章节对 AIDL 的介绍，我们发现以下语句是相同的对象：

- Context 的 getPackageManager()；
- ActivityThread 的 getPackageManager()；
- ActivityThread 的 sPackageManager；
- ApplicationPackageManager 的 mPM 字段。

所以当你在程序中看到上述这些语句时，它们都是 PMS 在 App 进程的代理对象，都能获得当前 Apk 包的信息，尤其是我们感兴趣的四大组件信息。在插件化编程中，我们反射 ActivityThread 获取 Apk 包的信息，一般用于当前的宿主 Apk ，而不是插件 Apk。

ApplicationPackageManager 实现了 IPackageManager.Stub。

2.12　ClassLoader 家族史

Android 插件化能从外部下载一个 apk 插件，就在于 ClassLoader。ClassLoader 是一个家族。ClassLoader 是老祖先，它有很多儿孙，ClassLoader 家族如图 2-31 所示。

其中最重要的是 PathClassLoader 和 DexClassLoader，及其父类 BaseDexClassLoader。

PathClassLoader 和 DexClassLoader 这两个类都很简单，以至于看不出什么区别，但仔细看，你会发现，构造函数的第 2 个参数 optimizedDirectory 的值不一样，PathClassLoader 把这个参数设置为 null，DexClassLoader 把这个参数设置为一个非空的值。

```
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String librarySearchPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), librarySearchPath, parent);
    }
}
```

```
public class PathClassLoader extends BaseDexClassLoader {
    public PathClassLoader(String dexPath, ClassLoader parent) {
        super(dexPath, null, null, parent);
    }

    public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader
parent) {
        super(dexPath, null, librarySearchPath, parent);
    }
}
```

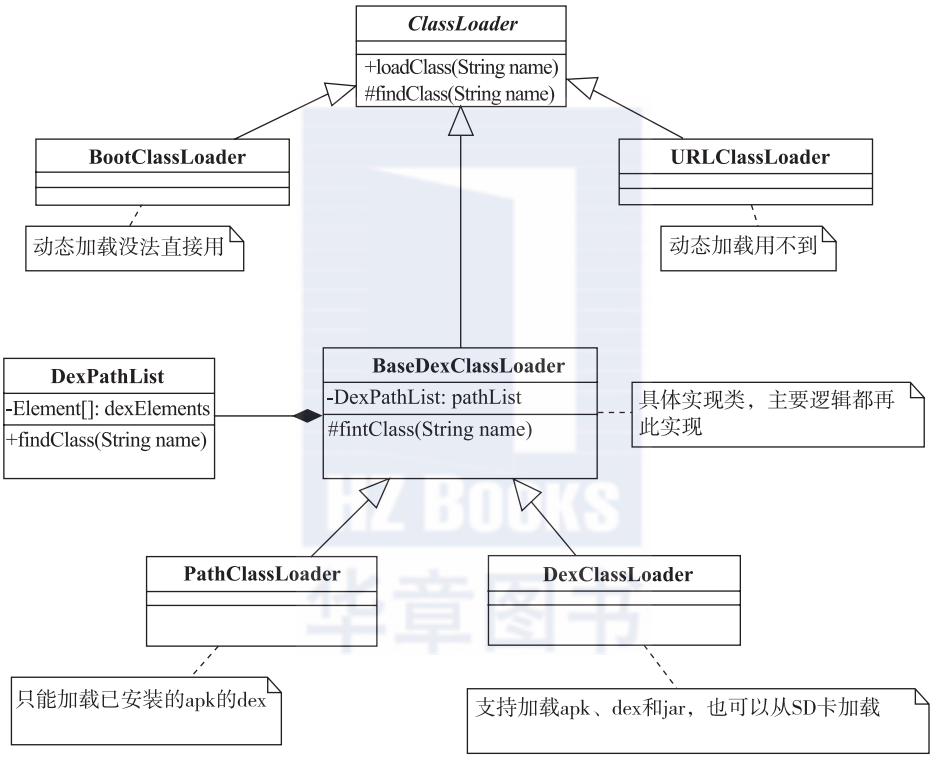


图 2-31 ClassLoader 的家谱

我们不打算深究更底层的代码，只需要知道，optimizedDirectory 是用来缓存我们需要加载的 dex 文件的，并创建一个 DexFile 对象，如果它为 null，那么会直接使用 dex 文件原有的路径来创建 DexFile 对象。

DexClassLoader 可以指定自己的 optimizedDirectory，所以它可以加载外部的 dex，因为这个 dex 会被复制到内部路径的 optimizedDirectory；而 PathClassLoader 没有 optimizedDirectory，所以它只能加载内部的 dex，这些大都是存在于系统中已经安装过的 Apk 里面的。

2.13　双亲委托

说完了 ClassLoader，就要讲双亲委托（Parent-Delegation Model）。对于 ClassLoader 家族，它们的老祖先是 ClassLoader 类，它的构造函数很有趣：

```
ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {
    if (parentLoader == null && !nullAllowed) {
        throw new NullPointerException("parentLoader == null && !nullAllowed");
    }
    parent = parentLoader;
}
```

主要看第 1 个参数，也就是创建一个 ClassLoader 对象的时候，都要使用一个现有的 ClassLoader 作为参数 parentLoader。这样在 Android App 中，会形成一棵由各种 ClassLoader 组成的树。

在 CLassLoader 加载类的时候，都会优先委派它的父亲 parentLoader 去加载类，沿着这棵数一路向上找，如果没有哪个 ClassLoader 能加载，那就只好自己加载这个类。

这样做的意义是为了性能，每次加载都会消耗时间，但如果父亲加载过，就可以直接拿来使用了。

对于 App 而言，Apk 文件中有一个 classes.dex 文件，那么这个 dex 就是 Apk 的主 dex，是通过 PathClassLoader 加载的。

在 App 的 Activity 中，通过 getClassLoader 获取到的是 PathClassLoader，它的父类是 BootClassLoader。

对于插件化而言，有一种方案是将 App 的 ClassLoader 替换为自定义的 ClassLoader，这样就要求自定义 ClassLoader 模拟双亲委托机制。比较典型的代码就是 Zeus 插件化框架。

2.14　MultiDex

记得还是在 Android 5 之前的版本，在 App 的开发过程中，随着业务的扩展，App 中的代码会急速增长，直到有一天，编译代码进行调试或者打包的时候，遇到下面的错误：

```
Conversion to Dalvik format failed:Unable to execute dex: method ID not in [0,
0xffff]: 65536
```

这就是著名的“65536 问题”，业内戏称为“爆棚”。

其实我们的项目中，自己写的代码功能再多，一般也不会超过这个 65536 的上限。往往是引入一些第三方的 SDK，它提供了很多的功能，而我们只用到其中几个功能或几个方法，那么其他数万个方法虽然用不到，但还是驻留在 App 中了，跟着一起打包。

在 Apk 打包的过程中，我们一般会开启 proguard。这个工具不仅是做混淆的，它还会把 App 中用不到的方法全部删除，所以第三方 SDK 中那些无用的方法就这样被移除了，方法数量大幅减少，又降低到了 65536 之下。

但是代码调试期间，是不会开启 proguard 的，所以“65536 问题”还是会出现。我当时的解决方案是，在开发其他功能时，把这个第三方 SDK 临时删掉，相应的功能注释掉——只要你不开发用到这个 SDK 的功能点，这种方案还是可行的，但毕竟不是长久之计。

后来 Google 官方解决了这个问题。一方面，Android 5.0 上修复了这个爆棚的 bug，我们拭目以待，看这个新的界限哪一天被突破，再次爆棚。

另一方面，在当时那个年代，还是要对 Android 5.0 之前的版本做兼容，类似于 2.3 和 4.4 版本的市场占用率还是很高的，于是 Google 推出了 MultiDex 这个工具。顾名思义，MultiDex 就是把原先的一个 dex 文件，拆分成多个 dex 文件。每个 dex 的方法数量不超过 65536 个，如图 2-32 所示。

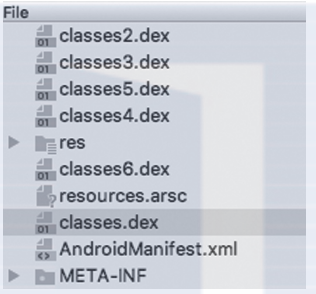


图 2-32 把 dex 拆分为多个

classes.dex 称为主 dex，由 App 使用 PathClassLoader 进行加载。而 classes2.dex 和 classes3.dex 这些子 dex，会在 App 启动后使用 DexClassLoader 进行加载。

时光荏苒，转眼已到 2018 年，市场上对 Android 系统的最低版本的支持已经到了 Android 5.0，再也不会遇到 65536 的爆棚问题了。那么 MultiDex 就没有用武之地了吗？并不是这样。在 Android 5.0 下，虽然 dex 中已经可以容纳比 65536 还多的方法数量，但是 dex 的体积却变大了。所以我们还是会对 dex 进行拆分，classes.dex 中只保留 App 启动时所需要的类以及首页的代码，从而确保 App 花最少时间启动并进入到首页；而把其他模块的代码转移到 classes2.dex 和 classes3.dex。

如何手动指定 classes2.dex 中包含 SecondActivity 的代码，而 classes3.dex 中包含 ThirdActivity 的代码呢？这个技术称为“手动分包”，我们会在 18.3 节介绍这个技术。

这个技术再往前走一步，就是插件化。我们可以按照模块把原先的项目拆分成多个 App，各自打包出 Apk，把这些 Apk 放到主 App 的 assets 目录下，然后使用 DexClass 进行加载。

有人说插件化能减少 App 的体积，这是不正确的。在 App 打包的时候，就要把插件的 1.0 版本预先放在 assets 目录下一起打包，然后有更新版本时（比如版本 1.1），才会下载新的插件或者增量包。

如果 App 打包时不包括插件，那么就会在 App 启动的时候才下载，这就慢了，用户体验很差。

所以 App 打包时一定要有 1.0 版本。用不用插件化，体积至少是相同的。

2.15　实现一个音乐播放器 App

相信很多读者还没接触过 Service 和 BroadcastReceiver 的编程，这里我们举一个音乐播放器的例子，来带领大家熟悉这两个组件的工作机制。

2.15.1　基于两个 Receiver 的音乐播放器[⊖]


提示　本节示例代码参见 https://github.com/Baobaojianqiang/ReceiverTestBetweenActivityAndService1。

音乐播放器有几个有趣的特点：

- 即使你切换到另一个 App，当前在播放的音乐，仍然播放而没有停止，这是利用了 Service 在后台播放音乐，其实直播也是基于这个思路来做的。音乐 App 的前台也就是 Activity，负责展示当前哪个音乐在播放，有播放和停止两个按钮，无论点击哪个按钮，都是通知后台 Service 播放或停止音乐。这个通知就是通过 BroadcastReceiver 从 Activity 发送给 Service 的。
- 每当后台 Service 播放完一首歌曲，就会通知前台 Activity，于是在后台 Service 播放下一首歌的同时，前台 Activity 的展示内容将从当前播放的歌曲名称和作者，切换到下一首歌的名称和作者。这个通知则是通过另一个 BroadcastReceiver 从 Service 发送给 Activity。

所以一个音乐播放 App，至少需要一个 Service 和两个 BroadcastReceiver。

1）在 AndroidManifest.xml 中，声明 Activity 和 Service：

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<service android:name=".MyService" />
```

2）在 Activity 这边，点击播放、停止按钮，都会发消息给 Service：

⊖ 这个例子参考自网上的一篇文章和示例，由于资料缺失，已经找不到它的地址了，还请知晓的读者把地址发给我，在本书的修订版本中更新。

```
public class MainActivity extends Activity {
    TextView tvTitle, tvAuthor;
    ImageButton btnPlay, btnStop;

    Receiver1 receiver1;

    //0x11: stoping; 0x12: playing; 0x13:pausing
    int status = 0x11;

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

        tvTitle = (TextView) findViewById(R.id.tvTitle);
        tvAuthor = (TextView) findViewById(R.id.tvAuthor);

        btnPlay = (ImageButton) this.findViewById(R.id.btnPlay);
        btnPlay.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //send message to receiver in Service
                Intent intent = new Intent("UpdateService");
                intent.putExtra("command", 1);
                sendBroadcast(intent);
            }
        });

        btnStop = (ImageButton) this.findViewById(R.id.btnStop);
        btnStop.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                //send message to receiver in Service
                Intent intent = new Intent("UpdateService");
                intent.putExtra("command", 2);
                sendBroadcast(intent);
            }
        });

        //register receiver in Activity
        receiver1 = new Receiver1();
        IntentFilter filter = new IntentFilter();
        filter.addAction("UpdateActivity");
        registerReceiver(receiver1, filter);

        //start Service
        Intent intent = new Intent(this, MyService.class);
        startService(intent);
    }
```

```
    public class Receiver1 extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            status = intent.getIntExtra("status", -1);
            int current = intent.getIntExtra("current", -1);
            if (current >= 0) {
                tvTitle.setText(MyMusics.musics[current].title);
                tvAuthor.setText(MyMusics.musics[current].author);
            }

            switch (status) {
                case 0x11:
                    btnPlay.setImageResource(R.drawable.play);
                    break;
                case 0x12:
                    btnPlay.setImageResource(R.drawable.pause);
                    break;
                case 0x13:
                    btnPlay.setImageResource(R.drawable.play);
                    break;
                default:
                    break;
            }
        }
    }
}
```

也许有人看不懂 Receiver1 和 Receiver2 的对应关系，为了便于理解，请参见图 2-33。

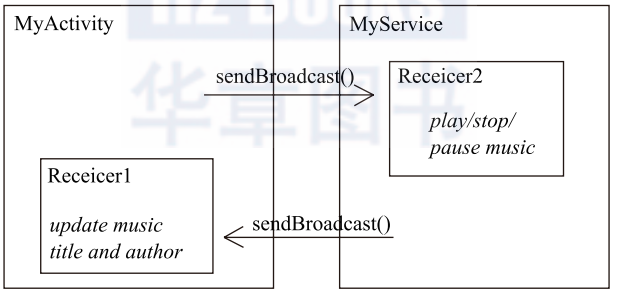


图 2-33　音乐播放器中的两个 Receiver

3）在 Service 这边，当一首音乐播放完，在播放下一首音乐的同时，会发消息给 Activity：

```
public class MyService extends Service {

    Receiver2 receiver2;
    AssetManager am;

    MediaPlayer mPlayer;
```

```
int status = 0x11;
int current = 0;

@Override
public IBinder onBind(Intent intent) {
    return null;
}

@Override
public void onCreate() {
    am = getAssets();

    //register receiver in Service
    receiver2 = new Receiver2();
    IntentFilter filter = new IntentFilter();
    filter.addAction("UpdateService");
    registerReceiver(receiver2, filter);

    mPlayer = new MediaPlayer();
    mPlayer.setOnCompletionListener(new OnCompletionListener() {
        @Override
        public void onCompletion(MediaPlayer mp) {
            current++;
            if (current >= 3) {
                current = 0;
            }
            prepareAndPlay(MyMusics.musics[current].name);

            //send message to receiver in Activity
            Intent sendIntent = new Intent("UpdateActivity");
            sendIntent.putExtra("status", -1);
            sendIntent.putExtra("current", current);
            sendBroadcast(sendIntent);
        }
    });
    super.onCreate();
}

private void prepareAndPlay(String music) {
    try {
        AssetFileDescriptor afd = am.openFd(music);
        mPlayer.reset();
        mPlayer.setDataSource(afd.getFileDescriptor()
                , afd.getStartOffset()
                , afd.getLength());
        mPlayer.prepare();
        mPlayer.start();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
    }

    public class Receiver2 extends BroadcastReceiver {
        @Override
        public void onReceive(final Context context, Intent intent) {
            int command = intent.getIntExtra("command", -1);
            switch (command) {
                case 1:
                    if (status == 0x11) {
                        prepareAndPlay(MyMusics.musics[current].name);
                        status = 0x12;
                    }
                    else if (status == 0x12) {
                        mPlayer.pause();
                        status = 0x13;
                    }
                    else if (status == 0x13) {
                        mPlayer.start();
                        status = 0x12;
                    }
                    break;
                case 2:
                    if (status == 0x12 || status == 0x13) {
                        mPlayer.stop();
                        status = 0x11;
                    }
            }

            //send message to receiver in Activity
            Intent sendIntent = new Intent("UpdateActivity");
            sendIntent.putExtra("status", status);
            sendIntent.putExtra("current", current);
            sendBroadcast(sendIntent);
        }
    }
}
```

也许有人看不懂 0x11（stoping），0x12（playing）和 0x13（pausing）三个状态之间的切换关系，为便于理解，请参见图 2-34。

2.15.2　基于一个 Receiver 的音乐播放器

本节的示例代码参见 https://github.com/Baobaojianqiang/ReceiverTestBetweenActivityAndService2。

上一节介绍了音乐播放器 App 的第一种实现——使用两个 Receiver 的方式，在 Activity 和 Service 中各注册一个 Receiver 给对方使用。

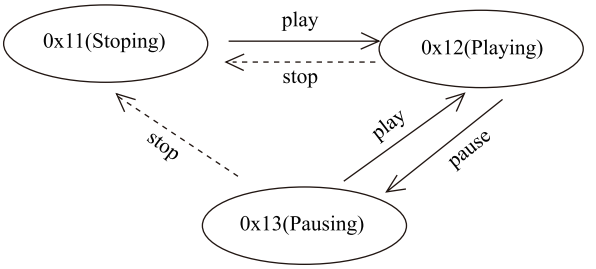


图 2-34 音乐按钮的状态机

其实，很多音乐类播放器中只有一个 Receiver，也就是上一节中介绍的 Receiver1，在 Activity 中注册，而后台音乐 Service 一旦播放完某首音乐，就发送广播给 Receiver1，更新 Activity 界面，

另一方面，我们使用 Service 的 onBind 方法，通过 ServiceConnection 获取到在 Service 中定义的 Binder 对象，在 Activity 点击播放或暂停音乐的按钮，会调用这个 Binder 对象的 play 和 stop 方法来操作后台 Service，如图 2-35 所示。

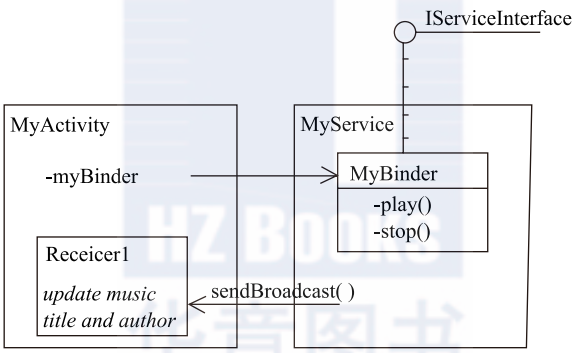


图 2-35 音乐播放器的流程图

为了解耦 Activity 和 Service，不在 Activity 中直接使用 Service 中定义的 MyBinder 对象，我们创建 IServiceInterface 接口，让 Activity 和 Service 都面向 IServiceInterface 接口编程：

```
public interface IServiceInterface {
    public void play();
    public void stop();
}
```

音乐播放器的 Activity 代码如下：

```
public class MainActivity extends Activity {
    TextView tvTitle, tvAuthor;
    ImageButton btnPlay, btnStop;
```

```
//0x11: stoping; 0x12: playing; 0x13:pausing
int status = 0x11;

Receiver1 receiver1;

IServiceInterface myService = null;

ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName name, IBinder binder) {
        myService = (IServiceInterface) binder;
        Log.e("MainActivity", "onServiceConnected");
    }

    public void onServiceDisconnected(ComponentName name) {
        Log.e("MainActivity", "onServiceDisconnected");
    }
};

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

    tvTitle = (TextView) findViewById(R.id.tvTitle);
    tvAuthor = (TextView) findViewById(R.id.tvAuthor);

    btnPlay = (ImageButton) this.findViewById(R.id.btnPlay);
    btnPlay.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            myService.play();
        }
    });

    btnStop = (ImageButton) this.findViewById(R.id.btnStop);
    btnStop.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            myService.stop();
        }
    });

    //register receiver in Activity
    receiver1 = new Receiver1();
    IntentFilter filter = new IntentFilter();
    filter.addAction("UpdateActivity");
    registerReceiver(receiver1, filter);

    //bind Service
    Intent intent = new Intent(this, MyService.class);
```

```
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE);
    }

    public class Receiver1 extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            status = intent.getIntExtra("status", -1);
            int current = intent.getIntExtra("current", -1);
            if (current >= 0) {
                tvTitle.setText(MyMusics.musics[current].title);
                tvAuthor.setText(MyMusics.musics[current].author);
            }

            switch (status) {
                case 0x11:
                    btnPlay.setImageResource(R.drawable.play);
                    break;
                case 0x12:
                    btnPlay.setImageResource(R.drawable.pause);
                    break;
                case 0x13:
                    btnPlay.setImageResource(R.drawable.play);
                    break;
                default:
                    break;
            }
        }
    }
}
```

音乐播放器的 Service 代码如下：

```
public class MyService extends Service {

    AssetManager am;

    MediaPlayer mPlayer;
    int status = 0x11;
    int current = 0;

    private class MyBinder extends Binder implements IServiceInterface {

        @Override
        public void play() {
            if (status == 0x11) {
                prepareAndPlay(MyMusics.musics[current].name);
                status = 0x12;
            } else if (status == 0x12) {
                mPlayer.pause();
                status = 0x13;
            } else if (status == 0x13) {
```

```
            mPlayer.start();
            status = 0x12;
        }

        sendMessageToActivity(status, current);
    }

    @Override
    public void stop() {
        if (status == 0x12 || status == 0x13) {
            mPlayer.stop();
            status = 0x11;
        }

        sendMessageToActivity(status, current);
    }
}

MyBinder myBinder = null;

@Override
public void onCreate() {
    myBinder = new MyBinder();

    am = getAssets();

    mPlayer = new MediaPlayer();
    mPlayer.setOnCompletionListener(new OnCompletionListener() {
        @Override
        public void onCompletion(MediaPlayer mp) {
            current++;
            if (current >= 3) {
                current = 0;
            }
            prepareAndPlay(MyMusics.musics[current].name);

            sendMessageToActivity(-1, current);
        }
    });
    super.onCreate();
}


@Override
public IBinder onBind(Intent intent) {
    return myBinder;
}

@Override
public boolean onUnbind(Intent intent) {
```

```
            myBinder = null;
            return super.onUnbind(intent);
        }

        private void sendMessageToActivity(int status1, int current1) {
            Intent sendIntent = new Intent("UpdateActivity");
            sendIntent.putExtra("status", status1);
            sendIntent.putExtra("current", current1);
            sendBroadcast(sendIntent);
        }

        private void prepareAndPlay(String music) {
            try {
                AssetFileDescriptor afd = am.openFd(music);
                mPlayer.reset();
                mPlayer.setDataSource(afd.getFileDescriptor()
                        , afd.getStartOffset()
                        , afd.getLength());
                mPlayer.prepare();
                mPlayer.start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

希望通过上面两个例子，能让各位读者更加深入了解 Receiver 和 Service，毕竟这两个组件平常的出镜率并不是很高。

2.16 本章小结

本章详细介绍了 Android 底层的知识点。本章没有贴太多代码，而是画了几十张图，这是本章的特色。

对于 Binder，掌握其原理即可，不需要深入源码，尤其是那些读起来让人头大的 C++ 代码。Binder 在 App 中的代表是 AIDL，所以要牢记，在 AIDL 生成的 Java 代码中 stub 和 proxy 的作用。

要牢记 App 的启动流程和安装流程。

四大组件非常重要。四大组件要和 AMS 打交道，要牢记 App 端是怎么和 AMS 打交道的，而不需要了解太多 AMS 内部的逻辑。这就引出了 App 端参与交互的那些类，比如：

- ActivityThread；
- Instrumentation；
- H；
- AMN 和 AMP；

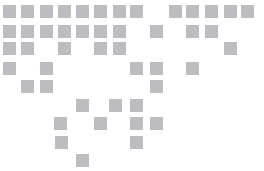
- Context 家族；
- LoadedApk。

我们一般对 Activity 很熟悉，而对其他三大组件就很少了解了，这是我们需要去补充的知识。音乐播放器是一个很好的例子。

对于 PMS，不需要深入 PMS 读取 Apk 的过程，而是要关注使用什么手段获取 Apk 的信息。重结果不重过程。

对于 ClassLoader 家族，要掌握 DexClassLoader，这是插件化编程的关键。此外对双亲委托机制要理解。

对于 MultiDex，要掌握手动拆包的办法，我们会在插件混淆的章节中用到这个技术。



Chapter 3 第 3 章

反　　射

本章介绍 Java 中最强大的技术：反射。

Java 原生的反射语法艰涩难懂，于是我们一般将这些反射语法封装成 Utils 类，包括反射类、反射方法（构造函数）、反射字段。这其中，做得最好的莫过于 jOOR 这个开源反射封装库。但是它有个缺点，就是不适用于 Android 中定义为 final 的字段，就连作者也承认，jOOR 只为了 Java 而设计，而没有考虑 Android。

所有反射语法中最难的莫过于反射一个泛型类，而这又是我们在做插件化编程中不可避免的。

3.1　基本反射技术

反射包括以下技术：

- ❑ 根据一个字符串得到一个类的对象。
- ❑ 获取一个类的所有公用或私有、静态或实例的字段、方法、属性。
- ❑ 对泛型类的反射。

相比于其他语言，Java 反射的语法是非常艰涩难懂的，我们按照上面的三点依次介绍。

提示　本节的示例代码参见 https://github.com/BaoBaoJianqiang/TestReflection。

3.1.1　根据一个字符串得到一个类

1. getClass

通过一个对象，获取它的类型。类型用 Class 表示：

```
String str = “abc”;
Class c1 = str.getClass();
```

2. Class.forName

这个方法用得最多。

通过一个字符串获取一个类型。这个字符串由类的命名空间和类的名称组成。而通过getSuperclass 方法，获取对象的父类类型：

```
try {
    Class c2 = Class.forName("java.lang.String");
    Class c3 = Class.forName("android.widget.Button");

    //通过getSuperClass，每个Class都有这个函数
    Class c5 = c3.getSuperclass();  //得到TextView
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

3. class 属性

每个类都有 class 属性，可以得到这个类的类型：

```
Class c6 = String.class;
Class c7 = java.lang.String.class;
Class c8 = MainActivity.InnerClass.class;
Class c9 = int.class;
Class c10 = int[].class;
```

4. TYPE 属性

基本类型，如 BOOLEAN，都有 TYPE 属性，可以得到这个基本类型的类型：

```
Class c11 = Boolean.TYPE;
Class c12 = Byte.TYPE;
Class c13 = Character.TYPE;
Class c14 = Short.TYPE;
Class c15 = Integer.TYPE;
Class c16 = Long.TYPE;
Class c17 = Float.TYPE;
Class c18 = Double.TYPE;
Class c19 = Void.TYPE;
```

看到这里，读者也许会问，为什么不厌其烦地想要得到类或对象的类型。在后面的章节，我们在使用 Proxy.newProxyInstance() 的时候，会大量用到这些类型值作为参数。

3.1.2 获取类的成员

1. 获取类的构造函数

获取类的构造函数，包括 private 和 public 两种，也支持无参数和有参数这两种类型的

构造函数。

比如 TestClassCtor 这个类，就有很多构造函数：

```
public TestClassCtor() {
    name = "baobao";
}

public TestClassCtor(int a) {

}

public TestClassCtor(int a, String b) {
    name = b;
}

private TestClassCtor(int a, double c) {

}
```

1）获取类的所有构造函数。

通过 Class 的 getDeclaredConstructors 方法，获取类的所有构造函数，包括 public 和 private 的构造函数，然后就可以通过 for 循环遍历每一个构造函数了：

```
TestClass r = new TestClass();
    Class temp = r.getClass();
    String className = temp.getName();        // 获取指定类的类名

    Log.v("baobao", "获取类的所有ctor，不分public还是private----------------------");
    //获取类的所有ctor，不分public还是private
    try {
        Constructor[] theConstructors = temp.getDedaredConstructors();
                                        // 获取指定类的公有构造方法

        for (int i = 0; i < theConstructors.length; i++) {
            int mod = theConstructors[i].getModifiers();    // 输出修饰域和方法名称
            Log.v("baobao", Modifier.toString(mod) + " " + className + "(");

            Class[] parameterTypes = theConstructors[i].getParameterTypes();
                                        //获取指定构造方法参数的集合
            for (int j = 0; j < parameterTypes.length; j++) {
                                        // 输出打印参数列表
                Log.v("baobao", parameterTypes[j].getName());
                if (parameterTypes.length > j + 1) {
                    Log.v("baobao", ", ");
                }
            }
            Log.v("baobao", ")");
        }
    } catch (Exception e) {
        e.printStackTrace();
        }
```

如果只想获取类的所有 public 构造函数，就不能再使用 Class 的 getConstructors 方法了，而要使用 getDeclaredConstructors 方法。

2）获取类的某个构造函数。

获取无参数的构造函数：

```
Constructor c1 = temp.getDeclaredConstructor();
```

获取有一个参数的构造函数，参数类型 int：

```
Class[] p2 = {int.class};
Constructor c2 = temp.getDeclaredConstructor(p2);
```

获取有两个参数的构造函数，参数类型依次是 int 和 String：

```
Class[] p3 = {int.class, String.class};
Constructor c3 = temp.getDeclaredConstructor(p3);
```

反射到类的构造函数很重要，这是下述流程中至关重要的一步：通过字符串反射出一个类，然后通过反射获取到类的构造函数，执行构造函数就得到了类的实例。有了实例，就可以通过反射进一步得到实例的所有字段和方法。

3）调用构造函数。

接下来通过反射调用构造函数，得到类的实例，这要借助于 Constructor 的 newInstance 方法：

```
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");

//含参
Class[] p3 = {int.class, String.class};
Constructor ctor = r.getDeclaredConstructor(p3);
Object obj = ctor.newInstance(1, "bjq");

//无参
Constructor ctor2 = r.getDeclaredConstructor();
Object obj2 = ctor2.newInstance();
```

如果构造函数是无参数的，那么可以直接使用 Class 的 newInstance 方法：

```
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");
Object obj4 = r.newInstance();
```

2. 获取类的私有实例方法并调用它

在 TestClassCtor 中，有一个私有方法 doSOmething：

```
private String doSOmething(String d) {
    Log.v("baobao", "TestClassCtor, doSOmething");

    return "abcd";
}
```

想获取这个私有方法并执行它，要写如下代码：

```
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");
Class[] p3 = {int.class, String.class};
Constructor ctor = r.getDeclaredConstructor(p3);
Object obj = ctor.newInstance(1, "bjq");

//以下4句话，调用一个private方法
Class[] p4 = {String.class};
Method method = r.getDeclaredMethod("doSOmething", p4); //在指定类中获取指定的方法
method.setAccessible(true);

Object argList[] = {"jianqiang"};   //这里写死，下面有个通用的函数getMethodParamObject
Object result = method.invoke(obj, argList);
```

3. 获取类的静态的私有方法并调用它

在 TestClassCtor 中，有一个静态的私有方法 work：

```
private static void work() {
    Log.v("baobao", "TestClassCtor, work");
}
```

想获取这个静态的私有方法并执行它，要写如下代码：

```
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");
//以下3句话，调用一个private静态方法
Method method = r.getDeclaredMethod("work"); //在指定类中获取指定的方法
method.setAccessible(true);
method.invoke(null);
```

4. 获取类的私有实例字段并修改它

在 TestClassCtor 中，有一个私有的实例字段 name：

```
public class TestClassCtor {
    private String name;

    public String getName() {
        return name;
    }
}
```

想获取这个私有实例字段并修改它的值，要写如下代码：

```
//以下4句话，创建一个对象
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");
Class[] p3 = {int.class, String.class};
Constructor ctor = r.getDeclaredConstructor(p3);
Object obj = ctor.newInstance(1, "bjq");

//获取name字段，private
Field field = r.getDeclaredField("name");
```

```
field.setAccessible(true);
Object fieldObject = field.get(obj);

//只对obj有效
field.set(obj, "jianqiang1982");
```

值得注意的是，这次修改仅对当前这个对象有效，如果接下来我们再次创建一个TestClassCtor对象，它的name字段的值为空而不是jianqiang1982：

```
TestClassCtor testClassCtor = new TestClassCtor(100);
testClassCtor.getName(); //仍然返回null，并没有修改
```

5. 获取类的私有静态字段并修改它

在TestClassCtor中，有一个静态的私有字段address，想获取这个私有的静态字段并修改它的值，要写如下代码：

```
//以下4句话，创建一个对象
Class r = Class.forName("jianqiang.com.testreflection.TestClassCtor");

//获取address静态字段，private
Field field = r.getDeclaredField("address");
field.setAccessible(true);

Object fieldObject = field.get(null);

field.set(fieldObject, "ABCD");

//静态变量，一次修改，终生受用
TestClassCtor.printAddress();
```

与前面介绍的实例字段不同，静态字段的值被修改了，下次再使用，这个字段的值是修改后的值。所谓“一次修改，终生受用”。

3.1.3 对泛型类的反射

Android系统源码中存在大量泛型，所以插件化技术离不开对泛型进行反射，比如单例模式（Singleton），下述代码是从Android源码中找出来的：

```
public abstract class Singleton<T> {
    private T mInstance;

    protected abstract T create();

    public final T get() {
        synchronized (this) {
            if (mInstance == null) {
                mInstance = create();
```

```
            }
            return mInstance;
        }
    }
}
```

Singleton 是一个泛型类，我们可以通过以下三行代码，取出 Singleton 中的 mInstance 字段：

```
Class<?> singleton = Class.forName("jianqiang.com.testreflection.Singleton");
Field mInstanceField = singleton.getDeclaredField("mInstance");
mInstanceField.setAccessible(true);
```

同时，Singleton 也是一个抽象类，在实例化 Singleton 的时候，一定要实现 create 这个抽象方法。

接下来我们看 ActivityManagerNative（AMN）这个类，其中和 Singleton 有关的是下面几行代码：

```
public class AMN {
    private static final Singleton<ClassB2Interface> gDefault = new Singleton
        <ClassB2Interface>() {
        protected ClassB2Interface create() {
            ClassB2 b2 = new ClassB2();
            b2.id = 2;
            return b2;
        }
    };

    static public ClassB2Interface getDefault() {
        return gDefault.get();
    }
}
```

上面的代码中 gDefault 是 AMN 的静态私有变量，它是 Singleton 类型的，所以要实现 create 方法，返回一个 ClassB2 类型的对象。

在 Android 的源码中，可通过 AMN.getDefault() 来获取 create 方法创建的 ClassB2 对象。

我们可以通过以下几行代码来获取 AMN 的 gDefault 静态私有字段，进一步得到生成的 ClassB2 类型对象 rawB2Object：

```
Class<?> activityManagerNativeClass = Class.forName("jianqiang.com.testreflection.
    AMN");
Field gDefaultField = activityManagerNativeClass.getDeclaredField("gDefault");
gDefaultField.setAccessible(true);
Object gDefault = gDefaultField.get(null);
// AMN的gDefault对象里面原始的 B2对象
Object rawB2Object = mInstanceField.get(gDefault);
```

后来我们可能发现，rawB2Object 不是我们所需要的，我们希望把它换成 ClassB2Mock 类型的对象 proxy。

ClassB2Mock 是对 rawB2Object 的动态代理，这里使用了 Proxy.newProxyInstance 方法，额外打印了一行日志：

```
// 创建一个这个对象的代理对象ClassB2Mock, 然后替换这个字段, 让我们的代理对象帮忙干活
Class<?> classB2Interface = Class.forName("jianqiang.com.testreflection.ClassB2Interface");
Object proxy = Proxy.newProxyInstance(
    Thread.currentThread().getContextClassLoader(),
    new Class<?>[] { classB2Interface },
    new ClassB2Mock(rawB2Object));
mInstanceField.set(gDefault, proxy);
```

最后一行代码就是把 AMN 中的 gDefault 字段的 mInstance 字段，设置为代理对象 proxy。

经过 Hook，AMN.getDefault().doSomething() 将执行 ClassB2Mock 里面的逻辑。

Android 源码中 AMN 的思路和我的这些代码在思路上是一致的，只是这里用 ClassB2 和 ClassB2Mock 来模拟，比较简单。

3.2 jOOR

上面例子的语法都是基于原始的 Java 语法，有没有觉得这些语法很艰涩？我们希望用一种自然的、简单的、面向对象的语法，来取代上面这些艰涩的语言，于是便有了 jOOR 这个开源库。⊖

jOOR 库就两个类，Reflect.java 和 ReflectException.java，所以我一般不依赖于 gradle，而是直接把这两个类拖到项目中来。

其中，Reflect.java 最为重要，包括 6 个核心方法：

- on：包裹一个类或者对象，表示在这个类或对象上进行反射，类的值可以是 Class，也可以是完整的类名（包含包名信息）。
- create：用来调用之前的类的构造方法，有两种重载，一种有参数，一种无参数。
- call：方法调用，传入方法名和参数，如有返回值还需要调用 get。
- get：获取（field 和 method 返回）值相关，会进行类型转换，常与 call 组合使用。
- set：设置属性值。

我们使用 jOOR 把上一节的代码重构一下。

本节的示例代码参见 https://github.com/BaoBaoJianqiang/TestReflection2。

⊖ 开源地址：https://github.com/jOOQ/jOOR

3.2.1 根据一个字符串得到一个类

1. getClass

通过一个字符串，获取它的类型。类型用 Class 表示：

```
String str = "abc";
Class c1 = str.getClass();
```

这些代码还是传统的语法，并没有改变。

2. 根据字符串获取一个类

对于 jOOR，我们一般 import 它的 Reflect.on 方法，这样我们就可以直接在代码中使用 on 了，让编码更加简单。代码如下：

```
import static jianqiang.com.testreflection.joor.Reflect.on;

    //以下3个语法等效
    Reflect r1 = on(Object.class);
    Reflect r2 = on("java.lang.Object");
    Reflect r3 = on("java.lang.Object", ClassLoader.getSystemClassLoader());

    //以下2个语法等效，实例化一个Object变量，得到Object.class
    Object o1 = on(Object.class).<Object>get();
    Object o2 = on("java.lang.Object").get();

    String j2 = on((Object)"abc").get();
    int j3 = on(1).get();

    //等价于Class.forName()
    try {
        Class j4 = on("android.widget.Button").type();
    }
    catch (ReflectException e) {
        e.printStackTrace();
    }
```

3.2.2 获取类的成员

1. 调用类的构造函数

调用类的构造函数，包括 private 和 public 两种，也支持无参数和有参数这两种类型的构造函数。

jOOR 认为构造函数就是用来调用的，所以没有给出获取构造函数的方法，而是直接给出了调用构造函数的方法 create：

```
TestClassCtor r = new TestClassCtor();
Class temp = r.getClass();
String className = temp.getName();         // 获取指定类的类名

//public构造函数
Object obj = on(temp).create().get();    //无参
Object obj2 = on(temp).create(1, "abc").get();   //有参

//private构造函数
TestClassCtor obj3 = on(TestClassCtor.class).create(1, 1.1).get();
String a = obj3.getName();
```

2. 获取类的私有实例方法

获取类的私有实例方法并调用它：

```
//以下4句话，创建一个对象
    TestClassCtor r = new TestClassCtor();
    Class temp = r.getClass();
    Reflect reflect = on(temp).create();

    //调用一个实例方法
    String a1 = reflect.call("doSOmething", "param1").get();
```

3. 获取类的私有静态方法

获取类的私有静态方法并调用它：

```
//以下4句话，创建一个对象
TestClassCtor r = new TestClassCtor();
Class temp = r.getClass();
Reflect reflect = on(temp).create();

//调用一个静态方法
on(TestClassCtor.class).call("work").get();
```

4. 获取类的私有实例字段

获取类的私有实例字段并修改它：

```
Reflect obj = on("jianqiang.com.testreflection.TestClassCtor").create(1, 1.1);
obj.set("name", "jianqiang");
Object obj1 = obj.get("name");
```

5. 获取类的私有静态字段

获取类的私有静态字段并修改它：

```
on("jianqiang.com.testreflection.TestClassCtor").set("address", "avcccc");
Object obj2 = on("jianqiang.com.testreflection.TestClassCtor").get("address");
```

3.2.3 对泛型类的反射

这个例子用 jOOR 写起来会更简单：

```
//获取AMN的gDefault单例gDefault, gDefault是静态的
Object gDefault = on("jianqiang.com.testreflection.AMN").get("gDefault");

// gDefault是一个 android.util.Singleton对象; 我们取出这个单例里面的mInstance字段
// mInstance就是原始的ClassB2Interface对象
Object mInstance = on(gDefault).get("mInstance");

// 创建一个这个对象的代理对象ClassB2Mock, 然后替换这个字段, 让我们的代理对象帮忙干活
Class<?> classB2Interface = on("jianqiang.com.testreflection.ClassB2Interface").type();
Object proxy = Proxy.newProxyInstance(
    Thread.currentThread().getContextClassLoader(),
    new Class<?>[] { classB2Interface },
    new ClassB2Mock(mInstance));

on(gDefault).set("mInstance", proxy);
```

外界对 jOOR 的评价很高，我看了相关的文章，大多是一些浮于表面的介绍，然后就被媒体无限放大，殊不知 jOOR 在 Android 领域有个很大的缺陷，那就是不支持反射 final 类型的字段。

看一个例子，User 类有两个 final 字段，其中 userId 是静态字段，name 是实例字段：

```
public class User {
    private final static int userId = 3;
    private final String name = "baobao";
}
```

在使用 jOOR 反射时的语法如下：

```
//实例字段
Reflect obj = on("jianqiang.com.testreflection.User").create();
obj.set("name", "jianqiang");
Object newObj = obj.get("name");

//静态字段
Reflect obj2 = on("jianqiang.com.testreflection.User");
obj2.set("userId", "123");
Object newObj2 = obj2.get("userId");
```

上面这段代码在执行 set 语法时必然报错，抛出一个 NoSuchFieldException 异常。究其原因，是 jOOR 的 Reflect 的 set 方法会在遇到 final 时，尝试反射出 Field 类的 modifiers 字段，在 Java 环境是有这个字段的，但是 Android 版本的 Field 类并没有这个字段，于是就报错了。

3.3 对基本反射语法的封装

考虑到 jOOR 的局限性，我们不得不另辟蹊径。对基本的 Java 反射语法进行封装，以得到简单的语法。

考察前面介绍的种种语法，无论是反射出一个类，还是反射出一个构造函数并调用它，都是为了进一步读写类的方法和字段，所以我们只要封装以下几个方法即可：

- 反射出一个构造函数并调用它。
- 调用静态方法。
- 调用实例方法。
- 获取和设置一个字段的值。
- 对泛型的处理。

本节的示例代码参见 https://github.com/BaoBaoJianqiang/TestReflection3。

3.3.1 反射出一个构造函数

在 RefInvoke 类定义方法如下：

```
public static Object createObject(String className, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Class r = Class.forName(className);
        Constructor ctor = r.getDeclaredConstructor(pareTyples);
        ctor.setAccessible(true);
        return ctor.newInstance(pareVaules);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

以下是对这个封装函数的调用：

```
Class r = Class.forName(className);

//含参
Class[] p3 = {int.class, String.class};
Object[] v3 = {1, "bjq"};
Object obj = RefInvoke.createObject(className, p3, v3);

//无参
Object obj2 = RefInvoke.createObject(className, null, null);
```

3.3.2 调用实例方法

在 RefInvoke 类定义方法如下：

```
public static Object invokeInstanceMethod(Object obj, String methodName, Class[]
    pareTyples, Object[] pareVaules) {
    if(obj == null)
```

```
            return null;

        try {
            //调用一个private方法
            Method method = obj.getClass().getDeclaredMethod(methodName, pareTyples);
                //在指定类中获取指定的方法
            method.setAccessible(true);
            return method.invoke(obj, pareVaules);

        } catch (Exception e) {
            e.printStackTrace();
        }

        return null;
    }
```

以下是调用这个封装函数：

```
Class[] p3 = {};
Object[] v3 = {};
RefInvoke.invokeStaticMethod(className, "work", p3, v3);
```

3.3.3 调用静态方法

在 RefInvoke 类定义方法如下：

```
public static Object invokeStaticMethod(String className, String method_name,
    Class[] pareTyples, Object[] pareVaules) {
    try {
            Class obj_class = Class.forName(className);
        Method method = obj_class.getDeclaredMethod(method_name, pareTyples);
        method.setAccessible(true);
        return method.invoke(null, pareVaules);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

以下是调用这个封装函数：

```
Class[] p4 = {String.class};
Object[] v4 = {"jianqiang"};
Object result = RefInvoke.invokeInstanceMethod(obj, "doSOmething", p4, v4);
```

3.3.4 获取并设置一个字段的值

在 RefInvoke 类定义方法如下：

```
public static Object getFieldObject(String className, Object obj, String filedName) {
    try {
```

```
            Class obj_class = Class.forName(className);
            Field field = obj_class.getDeclaredField(filedName);
            field.setAccessible(true);
            return field.get(obj);
        } catch (Exception e) {
            e.printStackTrace();
        }

        return null;
    }

    public static void setFieldObject(String classname, Object obj, String filedName,
        Object filedVaule) {
        try {
            Class obj_class = Class.forName(classname);
            Field field = obj_class.getDeclaredField(filedName);
            field.setAccessible(true);
            field.set(obj, filedVaule);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

以下是调用这两个封装函数：

```
//获取实例字段
Object fieldObject = RefInvoke.getFieldObject(className, obj, "name");
RefInvoke.setFieldObject(className, obj, "name", "jianqiang1982");

//获取静态字段
Object fieldObject = RefInvoke.getFieldObject(className, null, "address");
RefInvoke.setFieldObject(className, null, "address", "ABCD");
```

3.3.5 对泛型类的处理

借助于前面封装的 5 个方法，我们重写对泛型的反射调用：

```
//获取AMN的gDefault单例gDefault，gDefault是静态的
Object gDefault = RefInvoke.getFieldObject("jianqiang.com.testreflection.AMN",
    null, "gDefault");

// gDefault是一个 android.util.Singleton对象; 我们取出这个单例里面的mInstance字段
Object rawB2Object = RefInvoke.getFieldObject(
    "jianqiang.com.testreflection.Singleton",
    gDefault, "mInstance");

// 创建一个这个对象的代理对象ClassB2Mock, 然后替换这个字段, 让我们的代理对象帮忙干活
Class<?> classB2Interface = Class.forName("jianqiang.com.testreflection.
    ClassB2Interface");
Object proxy = Proxy.newProxyInstance(
```

```
        Thread.currentThread().getContextClassLoader(),
        new Class<?>[] { classB2Interface },
        new ClassB2Mock(rawB2Object));

//把Singleton的mInstance替换为proxy
RefInvoke.setFieldObject("jianqiang.com.testreflection.Singleton", gDefault,
        "mInstance", proxy);
```

虽然不是面向对象的语法，但总要简单得多。

3.4 对反射的进一步封装

我们在 3.3 节对反射语法进行了简单的封装，但在实际的使用中，我们发现，有的时候，有很多不方便的地方。在本节中会对此进行优化。

本节示例代码参考 https://github.com/BaoBaoJianqiang/TestReflection4

1. 对于无参数和只有一个参数的处理

通过反射调用方法或者构造函数的时候，它们有时只需要一个参数，有时根本不需要参数，但我们每次都要按照多个参数的方式来编写代码，如下所示：

```
Class r = Class.forName(className);

//含参
Class[] p3 = {int.class, String.class};
Object[] v3 = {1, "bjq"};
Object obj = RefInvoke.createObject(className, p3, v3);

//无参
Object obj2 = RefInvoke.createObject(className, null, null);
```

我们希望把代码写的更简单一些，比如说这样：

```
//无参
public static Object createObject(String className) {
    Class[] pareTyples = new Class[]{};
    Object[] pareVaules = new Object[]{};

    try {
        Class r = Class.forName(className);
        return createObject(r, pareTyples, pareVaules);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return null;
}
```

```
//一个参数
public static Object createObject(String className, Class pareTyple, Object pareVaule) {
    Class[] pareTyples = new Class[]{ pareTyple };
    Object[] pareVaules = new Object[]{ pareVaule };

    try {
        Class r = Class.forName(className);
        return createObject(r, pareTyples, pareVaules);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return null;
}

//多个参数
public static Object createObject(String className, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Class r = Class.forName(className);
        return createObject(r, pareTyples, pareVaules);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return null;
}

//多个参数
public static Object createObject(Class clazz, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Constructor ctor = clazz.getDeclaredConstructor(pareTyples);
        ctor.setAccessible(true);
        return ctor.newInstance(pareVaules);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

以此类推，构造函数就是方法，我们封装的 invokeStaticMethod 和 invokeInstanceMethod 方法也可以有这许多种重载方式。

2. 字符串可以替换为 Class

截止到现在，我们加载类的方式，都是通过字符串的方式，比如 createObject 的实现：

```
public static Object createObject(String className, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Class r = Class.forName(className);
        Constructor ctor = r.getConstructor(pareTyples);
        return ctor.newInstance(pareVaules);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

但有时候，我们直接就拥有这个类的 Class 类型，而不再用 Class.forName(className) 的方式再去生成，于是就可以有字符串和 Class 这两种形式的方法重载，如下所示：

```
//多个参数
public static Object createObject(String className, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Class r = Class.forName(className);
        return createObject(r, pareTyples, pareVaules);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

    return null;
}

//多个参数
public static Object createObject(Class clazz, Class[] pareTyples, Object[]
    pareVaules) {
    try {
        Constructor ctor = clazz.getConstructor(pareTyples);
        return ctor.newInstance(pareVaules);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

RefInvoke 中的所有方法，大都拥有 String 和 Class 这两种方法的重载，限于篇幅，这里就不一一介绍了，具体情况可以参考源码例子。

3. 区分静态字段和实例字段

在反射字段的时候，我们发现，对静态字段和实例字段的处理，区别就在一个地方，由于静态字段的反射不需要 obj 参数，所以设置为 null，如下所示：

```
//获取实例字段
Object fieldObject = RefInvoke.getFieldObject(className, obj, "name");
RefInvoke.setFieldObject(className, obj, "name", "jianqiang1982");

//获取静态字段
Object fieldObject = RefInvoke.getFieldObject(className, null, "address");
RefInvoke.setFieldObject(className, null, "address", "ABCD");
```

为了区分静态字段和实例字段这两种场景，我们把静态字段的读写方法改名为getStaticFieldObject 和 setStaticFieldObject，它们间接的调用了实例字段的 getFieldObject 和 setFieldObject 方法，但是省略了 obj 参数，如下所示，

```
public static Object getStaticFieldObject(String className, String filedName) {
    return getFieldObject(className, null, filedName);
}

public static void setStaticFieldObject(String classname, String filedName, Object
    filedVaule) {
    setFieldObject(classname, null, filedName, filedVaule);
}
```

那么在反射静态字段的时候，就可以优雅的写出下列代码了：

```
Object fieldObject = RefInvoke.getFieldObject(className, null, "address");
RefInvoke.setStaticFieldObject(className, "address", "ABCD");
```

4. 对反射读写字段的优化

继续观察对实例字段的封装方法，以 getFieldObject 为例：

```
public static Object getFieldObject(Class clazz, Object obj, String filedName) {
    try {
        Field field = clazz.getDeclaredField(filedName);
        field.setAccessible(true);
        return field.get(obj);
    } catch (Exception e) {
        e.printStackTrace();
    }

    return null;
}
```

我们发现，大多数情况下，obj 的类型就是 clazz，所以可以省略 clazz 参数，于是编写两个简易版的重载方法，如下所示：

```
public static Object getFieldObject(Object obj, String filedName) {
    return getFieldObject(obj.getClass(), obj, filedName);
}

public static void setFieldObject(Object obj, String filedName, Object filedVaule) {
    setFieldObject(obj.getClass(), obj, filedName, filedVaule);
}
```

然后就可以优雅的编写代码了：

```
Object fieldObject = RefInvoke.getFieldObject(className, "address");
RefInvoke.setStaticFieldObject(className, "address", "ABCD");
```

但是也有例外，如果 obj 的类型不是 clazz（obj 有可能是 clazz 的孙子，甚至辈分更低），那么就只能老老实实的使用原先封装的 getFieldObject 和 setFieldObject 方法。

3.5 本章小结

本章给出了反射语法的三种编写方式：

1）基本反射语法。

2）jOOR 语法。

3）对基本反射语法的封装。

由于 jOOR 对 Android 的支持不是很好，所以业内开源框架一般采用的是方式 1 和 3。在本书后面的章节中，使用方式 3 来编写插件化的反射代码。