



邓凡平◎著

Understanding Android Internals: Volume I

# 深入理解 Android

## 卷 I



机械工业出版社  
China Machine Press

# 深入理解 Android : 卷 I

邓凡平 著



机械工业出版社  
China Machine Press

这是一本以情景方式对 Android 的源代码进行深入分析的书。内容广泛,以对 Framework 层的分析为主,兼顾 Native 层和 Application 层;分析深入,每一部分源代码的分析都力求透彻;针对性强,注重实际应用开发需求,书中所涵盖的知识点都是 Android 应用开发者和系统开发者需要重点掌握的。

全书共 10 章,第 1 章介绍了阅读本书所需要做的准备工作,主要包括对 Android 系统架构和源码阅读方法的介绍;第 2 章通过对 Android 系统中的 MediaScanner 进行分析,详细讲解了 Android 中十分重要的 JNI 技术;第 3 章分析了 init 进程,揭示了通过解析 init.rc 来启动 Zygote 以及属性服务的工作原理;第 4 章分析了 Zygote、SystemServer 等进程的工作机制,同时还讨论了 Android 的启动速度、虚拟机 HeapSize 的大小调整、Watchdog 工作原理等问题;第 5 章讲解了 Android 系统中常用的类,包括 sp、wp、RefBase、Thread 等类,同步类,以及 Java 中的 Handler 类和 Looper 类,掌握这些类的知识后方能在后续的代码分析中做到游刃有余;第 6 章以 MediaServer 为切入点,对 Android 中极为重要的 Binder 进行了较为全面的分析,深刻揭示了其本质。第 7 章对 Audio 系统进行了深入的分析,尤其是 AudioTrack、AudioFlinger 和 AudioPolicyService 等的工作原理。第 8 章深入讲解了 Surface 系统的实现原理,分析了 Surface 与 Activity 之间以及 Surface 与 SurfaceFlinger 之间的关系、SurfaceFlinger 的工作原理、Surface 系统中的帧数据传输以及 LayerBuffer 的工作流程。第 9 章对 Vold 和 Rild 的原理和机制进行了深入的分析,同时还探讨了 Phone 设计优化的问题;第 10 章分析了多媒体系统中 MediaScanner 的工作原理。

本书适合有一定基础的 Android 应用开发工程师和系统工程师阅读。通过对本书的学习,大家将能更深刻地理解 Android 系统,从而自如应对实际开发中遇到的难题。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

## 图书在版编目(CIP)数据

深入理解 Android:卷 I/邓凡平著. —北京:机械工业出版社,2011.9

ISBN 978-7-111-35762-9

I. 深… II. 邓… III. 移动终端—应用程序—程序设计 IV. TN929.53

中国版本图书馆 CIP 数据核字(2011)第 176816 号

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:杨绣国 陈佳媛

印刷

2011 年 9 月第 1 版第 1 次印刷

186mm×240mm • 31.75 印张

标准书号:ISBN 978-7-111-35762-9

定价:69.00 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) 88378991; 88361066

购书热线:(010) 68326294; 88379649; 68995259

投稿热线:(010) 88379604

读者信箱:hzjsj@hzbook.com



近两年来，IT 行业的最热点聚焦到了移动互联网上。PC 时代，WINTEL 联盟成就了英特尔和微软各自的霸业。移动互联网时代，谁将上演新的传奇？新生的 Android 当年仅用短短一年多的时间就跻身全球智能操作系统的三甲行列。在北美市场，如今 Android 已经超过 iOS 和黑莓系统成为老大！Android 势不可挡，ARM+Android 组合的前景一片光明，越来越多的从业者加入了 Android 行列！

与带给人们良好用户体验的 iOS 不一样的是，Android 是一个开放的系统，其所有代码都是开源的。因此，对于开发者而言，不仅可以做到知其然，更可以做到知其所以然！

然而，要想知道其所以然，并不是一件简单的事情。回想当初，我开始接触 Android 的时候，除了 Android 源码外，其他资料甚少。Android 是基于 Linux 的完整操作系统，其代码量让人望而生畏。可以想象，在没有指导的情况下一头扎进操作系统庞大的代码中是一件让人多么痛苦的事情。时间过得很快，Android 生态链已经得到了充分的发展。现在市场上的 Android 资料已经开始泛滥，书籍已经数不胜数。然而，绝大部分书籍只限于讲解 Android 应用的开发（拜 Android 应用 API 所赐），没有深入到系统级的探讨，极少的所谓提供 Android 深入指导的资料也只是浅尝辄止。如果想深入了解 Android 系统，只有华山一条路：自己看 Android 源代码！

正是因为如此，当初凡平告诉我他要系统地整理其深入钻研 Android 源代码的心得时，我表示了强烈的赞同。这是一件极少有人做过的事情，这件事情将给已经或即将跨入 Android 世界的同仁们极大的帮助！这本书里，作者以代码框架为主线，用循序渐进的方式将框架中的

#### IV

关键点一一剖开，从而给读者一个架构清楚、细节完善的立体展现。另外，凡平还会用他的幽默给正在啃枯燥代码的您带来不少笑意和轻松。毫无疑问，如果您想深入了解 Android 系统，这本书就是您进入 Android 神秘世界的钥匙。

如果您看准了移动互联网的前景，想深入理解 Android，那就让这本书指导您前进吧！

邓必山

2011 年 6 月于北京





虽然前言位于书的最前面，但往往是最后才完成的。至今，本书的撰写工作算是基本完成了，在书稿付梓之前，心中却有些许忐忑和不安，因为拙著可能会存在 Bug。为此，我先为书中可能存在的 Bug 将给大家带来的麻烦致以真诚的歉意。另外，如果大家发现本书存在纰漏或有必要进一步探讨的地方，请发邮件<sup>①</sup>给我，我会尽快回复。非常乐意与大家交流。

## 本书主要内容

全书一共 10 章，其中一些重要章节中还设置了“拓展思考”部分。这 10 章的主要内容是：

第 1 章介绍了阅读本书所需要做的一些准备工作，包括对 Android 整个系统架构的认识，以及 Android 开发环境和源码阅读环境的搭建等。注意，本书分析的源码是 Android2.2。

第 2 章通过 Android 源码中的一处实例深入地介绍了 JNI 技术。

第 3 章围绕 init 进程，介绍了如何解析 init.rc 以启动 Zygote 和属性服务（property service）的工作原理。

第 4 章剖析了 zygote 和 system\_server 进程的工作原理。本章的拓展思考部分讨论了 Android 的启动速度、虚拟机 heapsize 的大小调整问题以及“看门狗”的工作原理。

第 5 章讲解了 Android 源码中常用的类，如 sp、wp、RefBase、Thread 类、同步类、Java 中的 Handler 类以及 Looper 类。这些类都是 Android 中最常用和最基本的，只有掌握这些类

---

<sup>①</sup> 我的 E-mail 是 fanping.deng@gmail.com。

的知识，才能在分析后续的代码时游刃有余。

第6章以 MediaServer 为切入点，对 Binder 进行了较为全面的分析。本章拓展思考部分讨论了与 Binder 有关的三个问题，它们分别是 Binder 和线程的关系、死亡通知以及匿名 Service。笔者希望，通过本章的学习，大家能更深入地认识 Binder 的本质。

第7章阐述了 Audio 系统中的三位重要成员 AudioTrack、AudioFlinger 和 AudioPolicyService 的工作原理。本章拓展思考部分分析了 AudioFlinger 中 DuplicatingThread 的工作原理，并且和读者一道探讨了单元测试、ALSA、Desktop check 等问题。通过对本章的学习，相信读者会对 Audio 系统有更深入的理解。

第8章以 Surface 系统为主，分析了 Activity 和 Surface 的关系、Surface 和 SurfaceFlinger 的关系以及 SurfaceFlinger 的工作原理。本章的拓展思考部分分析了 Surface 系统中数据传输控制对象的工作原理、有关 ViewRoot 的一些疑问，最后讲解了 LayerBuffer 的工作流程。这是全书中难度较大的一章，建议大家反复阅读和思考，这样才能进一步深入理解 Surface 系统。

第9章分析了 Vold 和 Rild，其中 Vold 负责 Android 平台中外部存储设备的管理，而 Rild 负责与射频通信有关的工作。本章的拓展思考部分介绍了嵌入式系统中与存储有关的知识，还探讨了 Rild 和 Phone 设计优化方面的问题。

第10章分析了多媒体系统中 MediaScanner 的工作原理。在本章的拓展思考部分，笔者提出了几个问题，旨在激发读者深入思考和学习 Android 的欲望。

## 本书特色

笔者认为，本书最大的特点在于，较全面、系统、深入地讲解了 Android 系统中的几大重要组成部分的工作原理，旨在通过直接剖析源代码的方式，引领读者一步步深入于诸如 Binder、Zygote、Audio、Surface、Vold、Rild 等模块的内部，去理解它们是如何实现的，以及如何工作的。笔者根据研究 Android 代码的心得，在本书中尝试性地采用了精简流程、逐个击破的方法进行讲解，希望这样做能帮助读者更快、更准确地把握各模块的工作流程及其本质。本书大部分章节中都专门撰写了“拓展思路”的内容，希望这部分内容能激发读者对 Android 代码进行深入研究的热情。

## 本书面向的读者

### （1）Android 应用开发工程师

对于 Android 应用开发工程师而言，本书中关于 Binder，以及 sp、wp、Handler 和 Looper 等常用类的分析或许能帮助你迅速适应 Android 平台上的开发工作。

### （2）Android 系统开发工程师

Android 系统开发工程师常常需要深入理解系统的运转过程，而本书所涉及的内容可能正是他们在工作和学习中最想了解的。那些对具体模块（如 Audio 系统和 Surface 系统）感兴趣



的读者也可以直接阅读相关章节的内容。

这里有必要提醒一下，要阅读此书，应具有 C++ 的基本知识，因为本书的大部分内容都集中在了 Native 层。

## 如何阅读本书

本书是在分析 Android 源码的基础上展开的，而源码文件所在的路径一般都很长，例如，文件 `AndroidRuntime.cpp` 的真实路径就是 `framework/base/core/jni/AndroidRuntime.cpp`。为了书写方便起见，我们在各章节开头把该章所涉及的源码路径全部都列出来了，而在具体分析源码时，则只列出该源码的文件名。下面就是一个示例：

```
[-->AndroidRuntime.cpp]
```

// 这里是源码分析和一些注释。

如有一些需要特别说明的地方，则会用下面的格式表示：

```
[-->AndroidRuntime.cpp:: 特别说明]
```

特别说明可帮助读者找到源码中的对应位置。

另外，本书在描述类之间的关系以及在函数调用流程上使用了 UML 的静态类图以及序列图。UML 是一个强大的工具，但它的建模规范过于烦琐，为更简单清晰地描述事情的本质，本书并未完全遵循 UML 的建模规范。这里仅举一例，如图 1 所示：

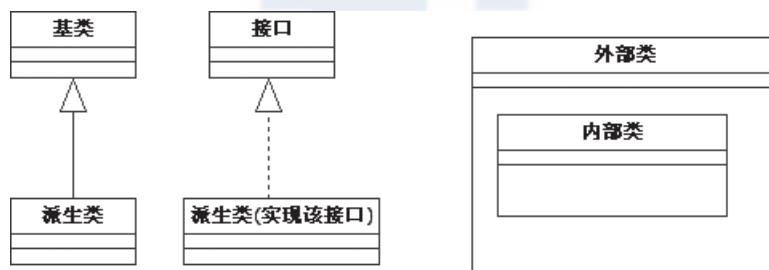


图 1 UML 示例图

如上图所示：

□ 画在外部类内部的方框用于表示内部类。

□ 接口和普通类用同一种框图表示。

本书所使用的 UML 图都比较简单，读者大可不必花费时间专门学习 UML。

本书的编写顺序，其实应该是 6、5、4、7、8、9、10、2、3、1 章，但出于逻辑连贯性的考虑，还是建议读者按本书的顺序阅读。其中，第 2、5、6 章分别讲述了 JNI、Android 常用类以及 Binder 系统，这些都是基础知识，我们有必要完全掌握。其他部分的内容都是针对单个模块的，例如 Zygote、Audio、Surface、MediaScanner 等，读者可各取所需，分别对其进行研究。





首先要感谢杨福川编辑。本书最初的内容来自我的博客<sup>①</sup>，但博客里的文章都没有图，格式也较混乱。是杨编辑最先鼓励我将这些博文整理修改成册，所以我对杨福川编辑的眼光佩服得五体投地。在他的同事杨绣国和白字的帮助下，我最终才将博客中那些杂乱的文章撰成了今天这本图文并茂、格式工整的书籍。

其次要感谢我的妻子。为写成此书，我几乎将周末所有的时间都花在了工作中，而长时间在生活上对妻子不闻不问。对丈夫呆若木鸡式的冷淡，妻子却给予了最大的宽容。另外，我的岳父母和我的父母亲都给予了我无私的帮助，他们都是平凡而伟大的父母亲。还有我和妻子的亲戚们，他们的宽厚和善良时刻感动着我。

在 IT 职业的道路上，非常感念前东家中科大洋公司的领导和同事们，他们是邓伟先生、刘运红先生、王宁先生等。当初，如果没有他们宽容的接纳和细心的指导，现在我不可能成为一名合格的程序员。

非常感谢我现在供职的单位中科创达公司<sup>②</sup>。在这里工作，我常有这样一种感慨：不是所有人都能自己开公司创业的，而又有多少人能够有机会和一个优秀的创业公司一起成长、一起发展呢？创达开明的领导、睿智而富有激情的工作伙伴正是孕育本书的沃土。公司领导赵鸿飞先生、吴安华女士等人更是给予了我最大的肯定和鼓励。

---

① 个人博客地址：<http://blog.csdn.net/Innost>。

② 中科创达公司主页：[www.thunderst.com](http://www.thunderst.com)。

这里要特别提及的是，我的大学同窗，即为本书作序的邓必山先生。如果没有他的推荐，凭自己那份简陋、单薄的简历，是根本无法与 Android 亲密接触的。另外，他还曾在技术和个人发展上给予过我很多的指导，对此，我将永志不忘！

谢谢那些共享 Android 知识的网友们！没有大家前期点滴的奉献，或许我至今还在琢磨着某段代码呢。

最后应感谢的是肯花费时间和精力阅读本书的读者，你们的意见和建议将会是我获得的巨大的精神财富！

邓凡平

2011 年 6 月于北京





## 第 1 章 阅读前的准备工作 / 1

- 1.1 系统架构 / 2
  - 1.1.1 Android 系统架构 / 2
  - 1.1.2 本书的架构 / 3
- 1.2 搭建开发环境 / 4
  - 1.2.1 下载源码 / 4
  - 1.2.2 编译源码 / 6
- 1.3 工具介绍 / 8
  - 1.3.1 Source Insight 介绍 / 8
  - 1.3.3 Busybox 的使用 / 11
- 1.4 本章小结 / 12

## 第 2 章 深入理解 JNI / 13

- 2.1 JNI 概述 / 14
- 2.2 学习 JNI 的实例：MediaScanner / 15
- 2.3 Java 层的 MediaScanner 分析 / 16

- 2.3.1 加载 JNI 库 / 16
- 2.3.2 Java 的 native 函数和总结 / 17
- 2.4 JNI 层 MediaScanner 的分析 / 17
  - 2.4.1 注册 JNI 函数 / 18
  - 2.4.2 数据类型转换 / 22
  - 2.4.3 JNIEnv 介绍 / 24
  - 2.4.4 通过 JNIEnv 操作 jobject / 25
  - 2.4.5 jstring 介绍 / 27
  - 2.4.6 JNI 类型签名介绍 / 28
  - 2.4.7 垃圾回收 / 29
  - 2.4.8 JNI 中的异常处理 / 32
- 2.5 本章小结 / 32

### 第 3 章 深入理解 init / 33

- 3.1 概述 / 34
- 3.2 init 分析 / 34
  - 3.2.1 解析配置文件 / 38
  - 3.2.2 解析 service / 42
  - 3.2.3 init 控制 service / 48
  - 3.2.4 属性服务 / 52
- 3.3 本章小结 / 60

### 第 4 章 深入理解 zygote / 61

- 4.1 概述 / 62
- 4.2 zygote 分析 / 62
  - 4.2.1 AppRuntime 分析 / 63
  - 4.2.2 Welcome to Java World / 68
  - 4.2.3 关于 zygote 的总结 / 74
- 4.3 SystemServer 分析 / 74
  - 4.3.1 SystemServer 的诞生 / 74
  - 4.3.2 SystemServer 的重要使命 / 77

- 4.3.3 关于 SystemServer 的总结 / 83
- 4.4 zygote 的分裂 / 84
  - 4.4.1 ActivityManagerService 发送请求 / 84
  - 4.4.2 有求必应之响应请求 / 86
  - 4.4.3 关于 zygote 分裂的总结 / 88
- 4.5 拓展思考 / 88
  - 4.5.1 虚拟机 heapsize 的限制 / 88
  - 4.5.2 开机速度优化 / 89
  - 4.5.3 Watchdog 分析 / 90
- 4.6 本章小结 / 93

## 第 5 章 深入理解常见类 / 95

- 5.1 概述 / 96
- 5.2 以“三板斧”揭秘 RefBase、sp 和 wp / 96
  - 5.2.1 第一板斧——初识影子对象 / 96
  - 5.2.2 第二板斧——由弱生强 / 103
  - 5.2.3 第三板斧——破解生死魔咒 / 106
  - 5.2.4 轻量级的引用计数控制类 LightRefBase / 108
  - 5.2.5 题外话——三板斧的来历 / 109
- 5.3 Thread 类及常用同步类分析 / 109
  - 5.3.1 一个变量引发的思考 / 109
  - 5.3.2 常用同步类 / 114
- 5.4 Looper 和 Handler 类分析 / 121
  - 5.4.1 Looper 类分析 / 122
  - 5.4.2 Handler 分析 / 124
  - 5.4.3 Looper 和 Handler 的同步关系 / 127
  - 5.4.4 HandlerThread 介绍 / 129
- 5.5 本章小结 / 129

## 第 6 章 深入理解 Binder / 130

- 6.1 概述 / 131

- 6.2 庖丁解 MediaServer / 132
  - 6.2.1 MediaServer 的入口函数 / 132
  - 6.2.2 独一无二的 ProcessState / 133
  - 6.2.3 时空穿越魔术——defaultServiceManager / 134
  - 6.2.4 注册 MediaPlayerService / 142
  - 6.2.5 秋风扫落叶——StartThread Pool 和 join Thread Pool 分析 / 149
  - 6.2.6 你彻底明白了吗 / 152
- 6.3 服务总管 ServiceManager / 152
  - 6.3.1 ServiceManager 的原理 / 152
  - 6.3.2 服务的注册 / 155
  - 6.3.3 ServiceManager 存在的意义 / 158
- 6.4 MediaPlayerService 和它的 Client / 158
  - 6.4.1 查询 ServiceManager / 158
  - 6.4.2 子承父业 / 159
- 6.5 拓展思考 / 162
  - 6.5.1 Binder 和线程的关系 / 162
  - 6.5.2 有人情味的计告 / 163
  - 6.5.3 匿名 Service / 165
- 6.6 学以致用 / 166
  - 6.6.1 纯 Native 的 Service / 166
  - 6.6.2 扶得起的“阿斗”(aidl) / 169
- 6.7 本章小结 / 172

## 第 7 章 深入理解 Audio 系统 / 173

- 7.1 概述 / 174
- 7.2 AudioTrack 的破解 / 174
  - 7.2.1 用例介绍 / 174
  - 7.2.2 AudioTrack (Java 空间) 分析 / 179
  - 7.2.3 AudioTrack (Native 空间) 分析 / 188
  - 7.2.4 关于 AudioTrack 的总结 / 200
- 7.3 AudioFlinger 的破解 / 200

- 7.3.1 AudioFlinger 的诞生 / 200
- 7.3.2 通过流程分析 AudioFlinger / 204
- 7.3.3 audio\_track\_cblk\_t 分析 / 230
- 7.3.4 关于 AudioFlinger 的总结 / 234
- 7.4 AudioPolicyService 的破解 / 234
  - 7.4.1 AudioPolicyService 的创建 / 235
  - 7.4.2 重回 AudioTrack / 245
  - 7.4.3 声音路由切换实例分析 / 251
  - 7.4.4 关于 AudioPolicy 的总结 / 262
- 7.5 拓展思考 / 262
  - 7.5.1 DuplicatingThread 破解 / 262
  - 7.5.2 题外话 / 270
- 7.6 本章小结 / 272

## 第 8 章 深入理解 Surface 系统 / 273

- 8.1 概述 / 275
- 8.2 一个 Activity 的显示 / 275
  - 8.2.1 Activity 的创建 / 275
  - 8.2.2 Activity 的 UI 绘制 / 294
  - 8.2.3 关于 Activity 的总结 / 296
- 8.3 初识 Surface / 297
  - 8.3.1 和 Surface 有关的流程总结 / 297
  - 8.3.2 Surface 之乾坤大挪移 / 298
  - 8.3.3 乾坤大挪移的 JNI 层分析 / 303
  - 8.3.4 Surface 和画图 / 307
  - 8.3.5 初识 Surface 小结 / 309
- 8.4 深入分析 Surface / 310
  - 8.4.1 与 Surface 相关的基础知识介绍 / 310
  - 8.4.2 SurfaceComposerClient 分析 / 315
  - 8.4.3 SurfaceControl 分析 / 320
  - 8.4.4 writeToParcel 和 Surface 对象的创建 / 331



- 8.4.5 lockCanvas 和 unlockCanvasAndPost 分析 / 335
- 8.4.6 GraphicBuffer 介绍 / 344
- 8.4.7 深入分析 Surface 的总结 / 353
- 8.5 SurfaceFlinger 分析 / 353
  - 8.5.1 SurfaceFlinger 的诞生 / 354
  - 8.5.2 SF 工作线程分析 / 359
  - 8.5.3 Transaction 分析 / 368
  - 8.5.4 关于 SurfaceFlinger 的总结 / 376
- 8.6 拓展思考 / 377
  - 8.6.1 Surface 系统的 CB 对象分析 / 377
  - 8.6.2 ViewRoot 的你问我答 / 384
  - 8.6.3 LayerBuffer 分析 / 385
- 8.7 本章小结 / 394

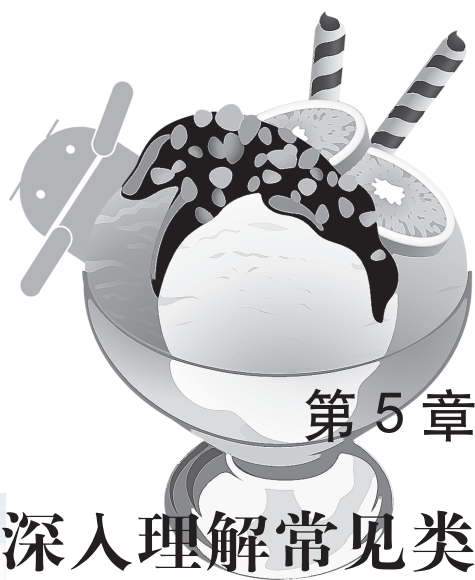
## 第 9 章 深入理解 Vold 和 Rild / 395

- 9.1 概述 / 396
- 9.2 Vold 的原理与机制分析 / 396
  - 9.2.1 Netlink 和 Uevent 介绍 / 397
  - 9.2.2 初识 Vold / 399
  - 9.2.3 NetlinkManager 模块分析 / 400
  - 9.2.4 VolumeManager 模块分析 / 408
  - 9.2.5 CommandListener 模块分析 / 414
  - 9.2.6 Vold 实例分析 / 417
  - 9.2.7 关于 Vold 的总结 / 428
- 9.3 Rild 的原理与机制分析 / 428
  - 9.3.1 初识 Rild / 430
  - 9.3.2 RIL\_startEventLoop 分析 / 432
  - 9.3.3 RIL\_Init 分析 / 437
  - 9.3.4 RIL\_register 分析 / 444
  - 9.3.5 关于 Rild main 函数的总结 / 447
  - 9.3.6 Rild 实例分析 / 447

- 9.3.7 关于 Rild 的总结 / 459
- 9.4 拓展思考 / 459
  - 9.4.1 嵌入式系统的存储知识介绍 / 459
  - 9.4.2 Rild 和 Phone 的改进探讨 / 462
- 9.5 本章小结 / 463

## 第 10 章 深入理解 MediaScanner / 464

- 10.1 概述 / 465
- 10.2 android.process.media 分析 / 465
  - 10.2.1 MSR 模块分析 / 466
  - 10.2.2 MSS 模块分析 / 467
  - 10.2.3 android.process.media 媒体扫描工作的流程总结 / 471
- 10.3 MediaScanner 分析 / 472
  - 10.3.1 Java 层分析 / 472
  - 10.3.2 JNI 层分析 / 476
  - 10.3.3 PVMediaScanner 分析 / 479
  - 10.3.4 关于 MediaScanner 的总结 / 485
- 10.4 拓展思考 / 486
  - 10.4.1 MediaScannerConnection 介绍 / 486
  - 10.4.2 我问你答 / 487
- 10.5 本章小结 / 488



## 第 5 章

# 深入理解常见类

### 本章涉及的源代码文件名称及位置

下面是本章分析的源码文件名和它的位置。

- ❑ `RefBase.h`(`framework/base/include/utils/RefBase.h`)
- ❑ `RefBase.cpp`(`framework/base/libs/utils/RefBase.cpp`)
- ❑ `Thread.cpp`(`framework/base/libs/utils/Thread.cpp`)
- ❑ `Thread.h`(`framework/base/include/utils/Thread.h`)
- ❑ `Atomic.h`(`system/core/include/cutils/Atomic.h`)
- ❑ `AndroidRuntime.cpp`(`framework/base/core/jni/AndroidRuntime.cpp`)
- ❑ `Looper.java`(`framework/base/core/Java/Android/os/Looper.java`)
- ❑ `Handler.java`(`framework/base/core/Java/Android/os/ Handler.java`)
- ❑ `HandlerThread.java`(`framework/base/core/Java/Android/os/ HandlerThread.java`)

## 5.1 概述

初次接触 Android 源码时，见到最多的一定是 sp 和 wp。即使你只是沉迷于 Java 世界的编码，那么 Looper 和 Handler 也是避不开的。本章的目的，就是把经常碰到的这些内容中的“拦路虎”一网打尽，将它们彻底搞懂。至于弄明白它们有什么好处，就仁者见仁，智者见智了。个人觉得 Looper 和 Handler 相对会更实用一些。

## 5.2 以“三板斧”揭秘 RefBase、sp 和 wp

RefBase 是 Android 中所有对象的始祖，类似于 MFC 中的 CObject 及 Java 中的 Object 对象。在 Android 中，RefBase 结合 sp 和 wp，实现了一套通过引用计数的方法来控制对象生命周期的机制。就如我们想像的那样，这三者的关系非常暧昧。初次接触 Android 源码的人往往会被那个随处可见的 sp 和 wp 搞晕了头。

什么是 sp 和 wp 呢？其实，sp 并不是我开始所想的 smart pointer（C++ 语言中有这个东西），它真实的意思应该是 strong pointer，而 wp 则是 weak pointer 的意思。我认为，Android 推出这一套机制可能是模仿 Java，因为 Java 世界中有所谓 weak reference 之类的东西。sp 和 wp 的目的，就是为了帮助健忘的程序员回收 new 出来的内存。

---


**说明** 我还是喜欢赤裸裸地管理内存的分配和释放。不过，目前 sp 和 wp 的使用已经深入到 Android 系统的各个角落，想把它去掉真是不太可能了。

---

这三者的关系比较复杂，都说程咬金的“三板斧”很厉害，那么我们就借用这三板斧，揭密其间的暧昧关系。

### 5.2.1 第一板斧——初识影子对象

我们的“三板斧”，其实就是三个例子。相信这三板斧劈下去，你会很容易理解它们。

 [--> 例子 1]

---

```
// 类 A 从 RefBase 派生，RefBase 是万物的始祖。
class A: public RefBase
{
    //A 没有任何自己的功能。
}
int main()
{
    A* pA = new A;
    {
        // 注意我们的 sp、wp 对象是在 {} 中创建的，下面的代码先创建 sp，然后创建 wp。
        sp<A> spA(pA);
        wp<A> wpA(spA);
    }
}
```

```

        // 大括号结束前，先析构 wp，再析构 sp。
    }
}

```

例子够简单吧？但也需一步一步分析这斧子是怎么劈下去的。

### 1. RefBase 和它的影子

类 A 从 RefBase 中派生。使用的是 RefBase 构造函数。代码如下所示：

👉 [-->RefBase.cpp]

```

RefBase::RefBase()
    : mRefs(new weakref_impl(this)) // 注意这句话
{
    // mRefs 是 RefBase 的成员变量，类型是 weakref_impl，我们暂且叫它影子对象。
    // 所以 A 有一个影子对象。
}

```

mRefs 是引用计数管理的关键类，需要进一步观察。它是从 RefBase 的内部类 weakref\_type 中派生出来的。

先看看它的声明：

```

class RefBase::weakref_impl : public RefBase::weakref_type
// 从 RefBase 的内部类 weakref_type 派生。

```

由于 Android 频繁使用 C++ 内部类的方法，所以初次阅读 Android 代码时可能会有点不太习惯，C++ 的内部类和 Java 的内部类相似，但有一点不同，即它需要一个显式的成员指向外部类对象，而 Java 的内部类对象有一个隐式的成员指向外部类对象的。

**说明** 内部类在 C++ 中的学名叫 nested class（内嵌类）。

👉 [-->RefBase.cpp::weakref\_impl 构造]

```

weakref_impl(RefBase* base)
    : mStrong(INITIAL_STRONG_VALUE) // 强引用计数，初始值为 0x1000000。
    , mWeak(0) // 弱引用计数，初始值为 0。
    , mBase(base) // 该影子对象所指向的实际对象。
    , mFlags(0)
    , mStrongRefs(NULL)
    , mWeakRefs(NULL)
    , mTrackEnabled(!DEBUG_REFS_ENABLED_BY_DEFAULT)
    , mRetain(false)
{
}

```

如你所见，new 了一个 A 对象后，其实还 new 了一个 weakref\_impl 对象，这里称它为

影子对象，另外我们称 A 为实际对象。

这里有一个问题：影子对象有什么用？

可以仔细想一下，是不是发现影子对象成员中有两个引用计数？一个强引用，一个弱引用。如果知道引用计数和对象生死有些许关联的话，就容易想到影子对象的作用了。

---

**说明** 按上面的分析来看，在构造一个实际对象的同时，还会悄悄地构造一个影子对象，在嵌入式设备的内存不是很紧俏的今天，这个影子对象的内存占用已经不成问题了。

---

## 2.sp 上场

程序继续运行，现在到了：

```
sp<A> spA(pA);
```

请看 sp 的构造函数，它的代码如下所示（注意，sp 是一个模板类，对此不熟悉的读者可以去翻翻书，或者干脆把所有出现的 T 都换成 A）：

 [-->RefBase.h::sp(T\* other)]

---

```
template<typename T>
sp<T>::sp(T* other) // 这里的 other 就是刚才创建的 pA。
    : m_ptr(other) // sp 保存了 pA 的指针。
{
    if (other) other->incStrong(this); // 调用 pA 的 incStrong。
}
```

---

OK，战场转到 RefBase 的 incStrong 中。它的代码如下所示：

 [-->RefBase.cpp]

---

```
void RefBase::incStrong(const void* id) const
{
    // mRefs 就是刚才在 RefBase 构造函数中 new 出来的影子对象。
    weakref_impl* const refs = mRefs;

    // 操作影子对象，先增加弱引用计数。
    refs->addWeakRef(id);
    refs->incWeak(id);
    .....
```

---

先来看看影子对象的这两个 weak 函数都干了些什么。

(1) 眼见而心不烦

下面看看第一个函数 addWeakRef，代码如下所示：

 [-->RefBase.cpp]

---

```
void addWeakRef(const void* /*id*/) { }
```

---

呵呵，addWeakRef啥都没做，因为这是 release 版走的分支。调试版的代码我们就不讨论了，它是给创造 RefBase、sp，以及 wp 的人调试用的。

---

**说明** 调试版分支的代码很多，看来创造它们的人也在为不理解它们之间的暧昧关系痛苦不已。

---

总之，一共有这么几个不用考虑的函数，下面都已列出来了。以后再碰见它们，干脆就直接跳过去：

```
void addStrongRef(const void* /*id*/) { }
void removeStrongRef(const void* /*id*/) { }
void addWeakRef(const void* /*id*/) { }
void removeWeakRef(const void* /*id*/) { }
void printRefs() const { }
void trackMe(bool, bool) { }
```

继续我们的征程。再看 incWeak 函数，代码如下所示：

👉 [-->RefBase.cpp]

---

```
void RefBase::weakref_type::incWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->addWeakRef(id); // 上面说了，非调试版什么都不干。
    const int32_t c = android_atomic_inc(&impl->mWeak);
    // 原子操作，影子对象的弱引用计数加1。
    // 千万记住影子对象的强弱引用计数的值，这是彻底理解 sp 和 wp 的关键。
}
```

---

好，我们再回到 incStrong，继续看代码：

👉 [-->RefBase.cpp]

---

```
.....
// 刚才增加了弱引用计数。
// 再增加强引用计数。
refs->addStrongRef(id); // 非调试版这里什么都不干。
// 下面函数为原子加1操作，并返回旧值。所以 c=0x1000000，而 mStrong 变为 0x1000001。
const int32_t c = android_atomic_inc(&refs->mStrong);
if (c != INITIAL_STRONG_VALUE) {
    // 如果 c 不是初始值，则表明这个对象已经被强引用过一次了。
    return;
}
// 下面这个是原子加操作，相当于执行 refs->mStrong + (-0x1000000)，最终 mStrong=1。
android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
/*
    如果是第一次引用，则调用 onFirstRef，这个函数很重要，派生类可以重载这个函数，完成一些
    初始化工作。
*/
const_cast<RefBase*>(this)->onFirstRef();
}
```

---



**说明** android\_atomic\_xxx 是 Android 平台提供的原子操作函数，原子操作函数是多线程编程中的常见函数，读者可以学习原子操作函数的相关知识，本章后面也会对其进行介绍。

## (2) sp 构造的影响

sp 构造完后，它给这个世界带来了什么？

那就是在 RefBase 中影子对象的强引用计数变为 1，且弱引用计数也变为 1。


更准确的说法是，sp 的出生导致影子对象的强引用计数加 1，且弱引用计数也加 1。

## (3) wp 构造的影响

继续看 wp，例子中的调用方式如下：

```
wp<A> wpA(spA)
```

wp 有好几个构造函数，原理都一样。来看这个最常见的：

 [-->RefBase.h::wp(const sp<T>& other)]

```
template<typename T>
wp<T>::wp(const sp<T>& other)
    : m_ptr(other.m_ptr) //wp 的成员变量 m_ptr 指向实际对象。
{
    if (m_ptr) {
        // 调用 pA 的 createWeak，并且保存返回值到成员变量 m_refs 中。
        m_refs = m_ptr->createWeak(this);
    }
}
```

 [-->RefBase.cpp]

```
RefBase::weakref_type* RefBase::createWeak(const void* id) const
{
    // 调用影子对象的 incWeak，这个我们刚才讲过了，它会导致影子对象的弱引用计数增加 1。
    mRefs->incWeak(id);
    return mRefs; // 返回影子对象本身。
}
```

我们可以看到，wp 化后，影子对象的弱引用计数将增加 1，所以现在弱引用计数为 2，而强引用计数仍为 1。另外，wp 中有两个成员变量，一个保存实际对象，另一个保存影子对象。sp 只有一个成员变量，用来保存实际对象，但这个实际对象内部已包含了对应的影子对象。

OK，wp 创建完了，现在开始进行 wp 的析构。

## (4) wp 析构的影响

wp 进入析构函数，则表明它快要离世了，代码如下所示：

 [-->RefBase.h]

```
template<typename T>
```

```
wp<T>::~~wp()
{
    if (m_ptr) m_refs->decWeak(this); // 调用影子对象的 decWeak, 由影子对象的基类实现。
}
```

#### 👉 [-->RefBase.cpp]

```
void RefBase::weakref_type::decWeak(const void* id)
{
    // 把基类指针转换成子类(影子对象)的类型, 这种做法有些违背面向对象编程的思想。
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id); // 非调试版不做任何事情。

    // 原子减1, 返回旧值, c=2, 而弱引用计数从2变为1。
    const int32_t c = android_atomic_dec(&impl->mWeak);
    if (c != 1) return; // c=2, 直接返回。

    // 如果c为1, 则弱引用计数为0, 这说明没用弱引用指向实际对象, 需要考虑是否释放内存。
    // OBJECT_LIFETIME_XXX和生命周期有关系, 我们后面再说。
    if ((impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
        if (impl->mStrong == INITIAL_STRONG_VALUE)
            delete impl->mBase;
        else {
            delete impl;
        }
    } else {
        impl->mBase->onLastWeakRef(id);
        if ((impl->mFlags & OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
            delete impl->mBase;
        }
    }
}
```

在例1中, wp析构后, 弱引用计数减1。但由于此时强引用计数和弱引用计数仍为1, 所以没有对象被干掉, 即没有释放实际对象和影子对象占据的内存。

#### (5) sp析构的影响

下面进入sp的析构。

#### 👉 [-->RefBase.h]

```
template<typename T>
sp<T>::~~sp()
{
    if (m_ptr) m_ptr->decStrong(this); // 调用实际对象的 decStrong, 由 RefBase 实现。
}
```

#### 👉 [-->RefBase.cpp]

```
void RefBase::decStrong(const void* id) const
```

```

{
    weakref_impl* const refs = mRefs;
    refs->removeStrongRef(id); // 调用影子对象的 removeStrongRef, 啥都不干。
    // 注意, 此时强弱引用计数都是 1, 下面函数调用的结果是 c=1, 强引用计数为 0。
    const int32_t c = android_atomic_dec(&refs->mStrong);
    if (c == 1) { // 对于我们的例子, c 为 1
        // 调用 onLastStrongRef, 表明强引用计数减为 0, 对象有可能被 delete。
        const_cast<RefBase*>(this)->onLastStrongRef(id);
        // mFlags 为 0, 所以会通过 delete this 把自己干掉。
        // 注意, 此时弱引用计数仍为 1。
        if ((refs->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
            delete this;
        }
        .....
    }
}

```

---

先看 delete this 的处理, 它会导致 A 的析构函数被调用。再来看 A 的析构函数, 代码如下所示:

👉 [--> 例子 1::~~A()]

```

// A 的析构直接导致进入 RefBase 的析构。
RefBase::~~RefBase()
{
    if (mRefs->mWeak == 0) { // 弱引用计数不为 0, 而是 1。
        delete mRefs;
    }
}

```

---

RefBase 的 delete this 自杀行为没有把影子对象干掉, 但我们还在 decStrong 中, 可从 delete this 接着往下看:

👉 [--> RefBase.cpp]

```

.... // 接前面的 delete this
if ((refs->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
    delete this;
}
// 注意, 实际数据对象已经被干掉了, 所以 mRefs 也没有用了, 但是 decStrong 刚进来
// 的时候就把 mRefs 保存到 refs 了, 所以这里的 refs 指向影子对象。
refs->removeWeakRef(id);
refs->decWeak(id); // 调用影子对象 decWeak
}

```

---

👉 [--> RefBase.cpp]

```

void RefBase::weakref_type::decWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);

```

```
impl->removeWeakRef(id); // 非调试版不做任何事情。

// 调用前影子对象的弱引用计数为 1，强引用计数为 0，调用结束后 c=1，弱引用计数为 0。
const int32_t c = android_atomic_dec(&impl->mWeak);
if (c != 1) return;

// 这次弱引用计数终于变为 0 了，并且 mFlags 为 0，mStrong 也为 0。
if ((impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
    if (impl->mStrong == INITIAL_STRONG_VALUE)
        delete impl->mBase;
    else {
        delete impl; // impl 就是 this，把影子对象也就是自己干掉。
    }
} else {
    impl->mBase->onLastWeakRef(id);
    if ((impl->mFlags & OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
        delete impl->mBase;
    }
}
}
```

好，第一板斧劈下去了！来看看它的结果是什么。

### 3. 第一板斧的结果

第一板斧过后，来总结一下刚才所学的知识：

- ❑ RefBase 中有一个隐含的影子对象，该影子对象内部有强弱引用计数。
- ❑ sp 化后，强弱引用计数各增加 1，sp 析构后，强弱引用计数各减 1。
- ❑ wp 化后，弱引用计数增加 1，wp 析构后，弱引用计数减 1。

完全彻底地消灭 RefBase 对象，包括让实际对象和影子对象灭亡，这些都是由强弱引用计数控制的，另外还要考虑 flag 的取值情况。当 flag 为 0 时，可得出如下结论：

- ❑ 强引用为 0 将导致实际对象被 delete。
- ❑ 弱引用为 0 将导致影子对象被 delete。

## 5.2.2 第二板斧——由弱生强

再看第二个例子，代码如下所示：

👉 [--> 例子 2]

```
int main()
{
    A *pA = new A();
    wp<A> wpA(pA);
    sp<A> spA = wpA.promote(); // 通过 promote 函数，得到一个 sp。
}
```

对 A 的 wp 化，不再做分析了。按照前面所讲的知识，wp 化后仅会使弱引用计数加 1，所以此处 wp 化的结果是：

影子对象的弱引用计数为 1，强引用计数仍然是初始值 0x1000000。

wpA 的 promote 函数是从一个弱对象产生一个强对象的重要函数，试看——

### 1. 由弱生强的方法

代码如下所示：

👉 [-->RefBase.h]

---

```
template<typename T>
sp<T> wp<T>::promote() const
{
    return sp<T>(m_ptr, m_refs); // 调用 sp 的构造函数。
}
```

---

👉 [-->RefBase.h]

---

```
template<typename T>
sp<T>::sp(T* p, weakref_type* refs)
    : m_ptr((p && refs->attemptIncStrong(this)) ? p : 0) // 有点看不清楚。
{
    // 上面那行代码够简洁，但是不方便阅读，我们写成下面这样：
    /*
        T* pTemp = NULL;
        // 关键函数 attemptIncStrong
        if (p != NULL && refs->attemptIncStrong(this) == true)
            pTemp = p;

        m_ptr = pTemp;
    */
}
```

---

### 2. 成败在此一举

由弱生强的关键函数是 attemptIncStrong，它的代码如下所示：

👉 [-->RefBase.cpp]

---

```
bool RefBase::weakref_type::attemptIncStrong(const void* id)
{
    incWeak(id); // 增加弱引用计数，此时弱引用计数变为 2。
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    int32_t curCount = impl->mStrong; // 这个仍是初始值。
    // 下面这个循环，在多线程操作同一个对象时可能会循环多次。这里可以不去管它，
    // 它的目的就是使强引用计数增加 1。
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
    }
}
```

---

```

    }
    curCount = impl->mStrong;
}

if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
    bool allow;
/*
    下面这个 allow 的判断极为精妙。impl 的 mBase 对象就是实际对象，有可能已经被 delete 了。
    curCount 为 0，表示强引用计数肯定经历了 INITIAL_STRONG_VALUE->1->...->0 的过程。
    mFlags 就是根据标志来决定是否继续进行 || 或 && 后的判断，因为这些判断都使用了 mBase，
    如不做这些判断，一旦 mBase 指向已经回收的地址，你就等着 segment fault 吧！
    其实，咱们大可不必理会这些东西，因为它不影响我们的分析和理解。
*/
    if (curCount == INITIAL_STRONG_VALUE) {
        allow = (impl->mFlags & OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
            || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
    } else {
        allow = (impl->mFlags & OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
            && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
    }
    if (!allow) {
        // allow 为 false，表示不允许由弱生强，弱引用计数要减去 1，这是因为咱们进来时加过一次。
        decWeak(id);
        return false; // 由弱生强失败。
    }

    // 允许由弱生强，强引用计数要增加 1，而弱引用计数已经增加过了。
    curCount = android_atomic_inc(&impl->mStrong);
    if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
        impl->mBase->onLastStrongRef(id);
    }
}
impl->addWeakRef(id);
impl->addStrongRef(id); // 两个函数调用没有作用。
if (curCount == INITIAL_STRONG_VALUE) {
    // 强引用计数变为 1。
    android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
    // 调用 onFirstRef，通知该对象第一次被强引用。
    impl->mBase->onFirstRef();
}
return true; // 由弱生强成功。
}

```

### 3. 第二板斧的结果

promote 完成后，相当于增加了一个强引用。根据上面所学的知识可知：

由弱生强成功后，强弱引用计数均增加 1。所以现在影子对象的强引用计数为 1，弱引用计数为 2。

## 5.2.3 第三板斧——破解生死魔咒

### 1. 延长生命的魔咒

RefBase 为我们提供了一个这样的函数：

```
extendObjectLifetime(int32_t mode)
```

另外还定义了一个枚举：

```
enum {
    OBJECT_LIFETIME_WEAK      = 0x0001,
    OBJECT_LIFETIME_FOREVER = 0x0003
};
```

注意：FOREVER 的值是 3，用二进制表示是 B11，而 WEAK 的二进制是 B01，也就是说 FOREVER 包括了 WEAK 的情况。

上面这两个枚举值，是破除强弱引用计数作用的魔咒。先观察 flags 为 OBJECT\_LIFETIME\_WEAK 的情况，见下面的例子。

👉 [--> 例子 3]

```
class A:public RefBase
{
    public A()
    {
        extendObjectLifetime(OBJECT_LIFETIME_WEAK); // 在构造函数中调用。
    }
}

int main()
{
    A *pA = new A();
    wp<A> wpA(pA); // 弱引用计数加 1。
    {
        sp<A> spA(pA) // sp 后，结果是强引用计数为 1，弱引用计数为 2。
    }
    ....
}
```

sp 的析构将直接调用 RefBase 的 decStrong，它的代码如下所示：

👉 [--> RefBase.cpp]

```
void RefBase::decStrong(const void* id) const
{
    weakref_impl* const refs = mRefs;
    refs->removeStrongRef(id);
    const int32_t c = android_atomic_dec(&refs->mStrong);
    if (c == 1) { // 上面进行原子操作后，强引用计数为 0
        const_cast<RefBase*>(this)->onLastStrongRef(id);
        // 注意这句话。如果 flags 不是 WEAK 或 FOREVER 的话，将 delete 数据对象。
        // 现在我们的 flags 是 WEAK，所以不会 delete 它。
    }
}
```



```

        if ((refs->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
            delete this;
        }
    }
    refs->removeWeakRef(id);
    refs->decWeak(id); // 调用前弱引用计数是 2。
}

```

然后调用影子对象的 `decWeak`。再来看它的处理，代码如下所示：

👉 [-->RefBase.cpp::weakref\_type 的 `decWeak()` 函数]

```

void RefBase::weakref_type::decWeak(const void* id)
{
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id);
    const int32_t c = android_atomic_dec(&impl->mWeak);
    if (c != 1) return; // c 为 2，弱引用计数为 1，直接返回。
    /*
     * 假设我们现在到了例子中的 wp 析构之处，这时也会调用 decWeak，在调用上面的原子减操作后
     * c=1，弱引用计数变为 0，此时会继续往下运行。由于 mFlags 为 WEAK，所以不满足 if 的条件。
     */
    if ((impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK) {
        if (impl->mStrong == INITIAL_STRONG_VALUE)
            delete impl->mBase;
        else {
            delete impl;
        }
    } else { // flag 为 WEAK，满足 else 分支的条件。
        impl->mBase->onLastWeakRef(id);
        /*
         * 由于 flags 值满足下面这个条件，所以实际对象会被 delete，根据前面的分析可知，实际对象的
         * delete 会检查影子对象的弱引用计数，如果它为 0，则会把影子对象也 delete 掉。
         * 由于影子对象的弱引用计数此时已经为 0，所以影子对象也会被 delete。
         */
        if ((impl->mFlags&OBJECT_LIFETIME_FOREVER) != OBJECT_LIFETIME_FOREVER) {
            delete impl->mBase;
        }
    }
}

```

## 2. LIFETIME\_WEAK 的魔力

看完上面的例子，我们发现什么了？

在 `LIFETIME_WEAK` 的魔法下，强引用计数为 0，而弱引用计数不为 0 的时候，实际对象没有被 `delete`！只有当强引用计数和弱引用计数同时为 0 时，实际对象和影子对象才会被 `delete`。

### 3. 魔咒大揭秘

至于 LIFETIME\_FOREVER 的破解，就不用再来一斧子了，我直接给出答案：

- ❑ flags 为 0，强引用计数控制实际对象的生命周期，弱引用计数控制影子对象的生命周期。强引用计数为 0 后，实际对象被 delete。所以对于这种情况，应记住的是，使用 wp 时要由弱生强，以免收到 segment fault 信号。
- ❑ flags 为 LIFETIME\_WEAK，强引用计数为 0，弱引用计数不为 0 时，实际对象不会被 delete。当弱引用计数减为 0 时，实际对象和影子对象会同时被 delete。这是功德圆满的情况。
- ❑ flags 为 LIFETIME\_FOREVER，对象将长生不老，彻底摆脱强弱引用计数的控制。所以你要在适当的时候杀死这些“老妖精”，免得她祸害“人间”。

## 5.2.4 轻量级的引用计数控制类 LightRefBase

上面介绍的 RefBase，是一个重量级的引用计数控制类。那么，究竟有没有一个简单些的引用计数控制类呢？Android 为我们提供了一个轻量级的 LightRefBase。这个类非常简单，我们不妨一起来看看。

👉 [-->RefBase.h]

---

```
template <class T>
class LightRefBase
{
public:
    inline LightRefBase() : mCount(0) { }
    inline void incStrong(const void* id) const {
        //LightRefBase 只有一个引用计数控制量 mCount。incStrong 的时候使它增加 1。
        android_atomic_inc(&mCount);
    }
    inline void decStrong(const void* id) const {
        //decStrong 的时候减 1，当引用计数变为零的时候，delete 掉自己。
        if (android_atomic_dec(&mCount) == 1) {
            delete static_cast<const T*>(this);
        }
    }
    inline int32_t getStrongCount() const {
        return mCount;
    }

protected:
    inline ~LightRefBase() { }

private:
    mutable volatile int32_t mCount; // 引用计数控制变量。
};
```

---

LightRefBase 类够简单吧？不过它是一个模板类，我们该怎么用它呢？下面给出一个例子，其中类 A 是从 LightRefBase 派生的，写法如下：

```
class A:public LightRefBase<A> // 注意派生的时候要指明是 LightRefBase<A>。
{
public:
    A(){};
    ~A(){};
};
```

另外，我们从 LightRefBase 的定义中可以知道，它支持 sp 的控制，因为它只有 incStrong 和 decStrong 函数。

### 5.2.5 题外话——三板斧的来历

从代码量上看，RefBase、sp 和 wp 的代码量并不多，但里面的关系，尤其是 flags 的引入，曾一度让我眼花缭乱。当时，我确实很希望能自己调试一下这些例子，但在设备上调试 native 代码，需要花费很大的精力，即使是通过输出 log 的方式来调试也需要花很多时间。该怎么解决这一难题？

既然它的代码不多而且简单，那何不把它移植到台式机的开发环境下，整一个类似的 RefBase 呢？有了这样的构想，我便用上了 Visual Studio。至于那些原子操作，Windows 平台上有很直接的 InterlockedExchangeXXX 与之对应，真的是踏破铁鞋无觅处，得来全不费功夫！（在 Linux 平台上，不考虑多线程的话，将原子操作换成普通的非原子操作不是也可以吗？如果更细心更负责任的话，你可以自己用汇编来实现常用的原子操作，内核代码中有现成的函数，一看就会明白。）

如果把破解代码看成是攻城略地的话，我们必须学会灵活多变，而且应力求破解方法日臻极致！

## 5.3 Thread 类及常用同步类分析

Thread 类是 Android 为线程操作而做的一个封装。代码在 Thread.cpp 中，其中还封装了一些与线程同步相关的类（既然是封装，要掌握它，最重要的当然是掌握与 Pthread 相关的知识）。我们先分析 Threa 类，进而再介绍与常用同步类相关的知识。

### 5.3.1 一个变量引发的思考

Thread 类虽说挺简单，但其构造函数中的那个 canCallJava 却一度让我感到费解。因为我一直使用的是自己封装的 Pthread 类。当发现 Thread 构造函数中竟然存在这样一个东西时，很担心自己封装的 Pthread 类会不会有什么重大问题，因为当时我还从来没考虑过 Java 方面的问题。

// canCallJava 表示这个线程是否会使用 JNI 函数。为什么需要一个这样的参数呢？  
Thread(bool canCallJava = true)。

我们必须得了解它实际创建的线程函数是什么。Thread 类真实的线程是创建在 run 函数中的。

### 1. 一个变量，两种处理

先来看一段代码：

 [-->Thread.cpp]

---

```
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    Mutex::Autolock _l(mLock);
    ....
    // 如果 mCanCallJava 为真，则调用 createThreadEtc 函数，线程函数是 _threadLoop。
    // _threadLoop 是 Thread.cpp 中定义的一个函数。
    if (mCanCallJava) {
        res = createThreadEtc(_threadLoop, this, name, priority,
                               stack, &mThread);
    } else {
        res = androidCreateRawThreadEtc(_threadLoop, this, name, priority,
                                         stack, &mThread);
    }
}
```

---

上面的 mCanCallJava 将线程创建函数的逻辑分为两个分支，虽传入的参数都有 \_threadLoop，但它们调用的函数却不同。先直接看 mCanCallJava 为 true 的这个分支，代码如下所示：

 [-->Thread.h:createThreadEtc() 函数]

---

```
inline bool createThreadEtc(thread_func_t entryFunction,
                             void *userData,
                             const char* threadName = "android:unnamed_thread",
                             int32_t threadPriority = PRIORITY_DEFAULT,
                             size_t threadStackSize = 0,
                             thread_id_t *threadId = 0)
{
    return androidCreateThreadEtc(entryFunction, userData, threadName,
                                   threadPriority, threadStackSize, threadId) ? true : false;
}
```

---

它调用的是 androidCreateThreadEtc 函数，相关代码如下所示：

```
// gCreateThreadFn 是函数指针，它在初始化时和 mCanCallJava 为 false 时使用是同一个
// 线程创建函数。那么有地方会修改它吗？
static android_create_thread_fn gCreateThreadFn = androidCreateRawThreadEtc;
int androidCreateThreadEtc(android_thread_func_t entryFunction,
                           void *userData, const char* threadName,
```

```

        int32_t threadPriority, size_t threadStackSize,
        android_thread_id_t *threadId)
    {
        return gCreateThreadFn(entryFunction, userData, threadName,
                               threadPriority, threadStackSize, threadId);
    }

```

---

如果没有人修改这个函数指针，那么 `mCanCallJava` 就是虚晃一枪，并无什么作用。不过，代码中有的地方是会修改这个函数指针的指向的，请看——

## 2. zygote 偷梁换柱

在本书 4.2.1 节的第 2 点所介绍的 `AndroidRuntime` 调用 `startReg` 的地方，就有可能修改这个函数指针，其代码如下所示：

👉 [--> `AndroidRuntime.cpp`]

---

```

/*static*/ int AndroidRuntime::startReg(JNIEnv* env)
{
    // 这里会修改函数指针为 javaCreateThreadEtc。
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);
    return 0;
}

```

---

如果 `mCanCallJava` 为 `true`，则将调用 `javaCreateThreadEtc`。那么，这个函数有什么特殊之处呢？来看其代码，如下所示：

👉 [--> `AndroidRuntime.cpp`]

---

```

int AndroidRuntime::javaCreateThreadEtc(
    android_thread_func_t entryFunction,
    void* userData,
    const char* threadName,
    int32_t threadPriority,
    size_t threadStackSize,
    android_thread_id_t* threadId)
{
    void** args = (void**) malloc(3 * sizeof(void*));
    int result;
    args[0] = (void*) entryFunction;
    args[1] = userData;
    args[2] = (void*) strdup(threadName);
    // 调用的还是 androidCreateRawThreadEtc，但线程函数却换成了 javaThreadShell。
    result = androidCreateRawThreadEtc(AndroidRuntime::javaThreadShell, args,
                                       threadName, threadPriority, threadStackSize, threadId);
    return result;
}

```

---

👉 [-->AndroidRuntime.cpp]

---

```
int AndroidRuntime::javaThreadShell(void* args) {
    .....
    int result;
    // 把这个线程 attach 到 JNI 环境中，这样这个线程就可以调用 JNI 的函数了。
    if (javaAttachThread(name, &env) != JNI_OK)
        return -1;
    // 调用实际的线程函数干活。
    result = (*(android_thread_func_t)start)(userData);
    // 从 JNI 环境中 detach 出来。
    javaDetachThread();
    free(name);
    return result;
}
```

---

### 3. 费力能讨好

你明白 mCanCallJava 为 true 的目的了吗？它创建的新线程将：

- ❑ 在调用你的线程函数之前会 attach 到 JNI 环境中，这样，你的线程函数就可以无忧无虑地使用 JNI 函数了。
- ❑ 线程函数退出后，它会从 JNI 环境中 detach，释放一些资源。

---

**注意** 第二点尤其重要，因为进程退出前，dalvik 虚拟机会检查是否有 attach 了，如果最后有未 detach 的线程，则会直接 abort（这不是一件好事）。如果你关闭 JNI check 选项，就不会做这个检查，但我觉得，这个检查和资源释放有关系，建议还是重视。如果直接使用 POSIX 的线程创建函数，那么凡是使用过 attach 的，最后就都需要 detach！

---

Android 为了 dalvik 的健康真是费尽心机呀。

### 4. 线程函数 \_threadLoop 介绍

无论一分为二是如何处理的，最终都会调用线程函数 \_threadLoop，为什么不直接调用用户传入的线程函数呢？莫非 \_threadLoop 会有什么暗箱操作吗？下面我们来看：

👉 [-->Thread.cpp]

---

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

    #if HAVE_ANDROID_OS
        self->mTid = gettid();
    #endif
}
```

---

```

bool first = true;

do {
    bool result;
    if (first) {
        first = false;
        //self 代表继承 Thread 类的对象，第一次进来时将调用 readyToRun，看看是否准备好。
        self->mStatus = self->readyToRun();
        result = (self->mStatus == NO_ERROR);

        if (result && !self->mExitPending) {
            result = self->threadLoop();
        }
    } else {
        /*
        调用子类实现的 threadLoop 函数，注意这段代码运行在一个 do-while 循环中。
        这表示即使我们的 threadLoop 返回了，线程也不一定会退出。
        */
        result = self->threadLoop();
    }
}

/*
线程退出的条件：
1) result 为 false。这表明，如果子类在 threadLoop 中返回 false，线程就可以退出。这属于主动退出的情况，是 threadLoop 自己不想继续干活了，所以返回 false。读者在自己的代码中千万别写错 threadLoop 的返回值。
2) mExitPending 为 true，这个变量可由 Thread 类的 requestExit 函数设置，这种情况属于被动退出，因为由外界强制设置了退出条件。
*/
    if (result == false || self->mExitPending) {
        self->mExitPending = true;
        self->mLock.lock();
        self->mRunning = false;
        self->mThreadExitedCondition.broadcast();
        self->mLock.unlock();
        break;
    }
    strong.clear();
    strong = weak.promote();
} while(strong != 0);

return 0;
}

```

---

关于 `_threadLoop`，我们就介绍到这里。请读者务必注意下面一点：  
`threadLoop` 运行在一个循环中，它的返回值可以决定是否退出线程。



### 5.3.2 常用同步类

同步，是多线程编程中不可避免的话题，同时也是一个非常复杂的问题。这里只简单介绍一下 Android 提供的同步类。这些类，只对系统提供的多线程同步函数（这种函数我们称为 Raw API）进行了面向对象的封装，读者必须先理解 Raw API，然后才能真正掌握其具体用法。

---

**提示** 要了解 Windows 下的多线程编程，有很多参考资料，而有关 Linux 下完整系统阐述多线程编程的书籍目前较少，这里推荐一本含金量较高的著作《Programming with POSIX Thread》（本书只有英文版，由 Addison-Wesley 出版）。

---

Android 提供了两个封装好的同步类，它们是 Mutex 和 Condition。这是重量级的同步技术，一般内核都会有对应的支持。另外，OS 还提供了简单的原子操作，这些也算是同步技术中的一种。下面分别来介绍这三种东西。

#### 1. 互斥类——Mutex

Mutex 是互斥类，用于多线程访问同一个资源的时候，保证一次只有一个线程能访问该资源。在《Windows 核心编程》<sup>①</sup>一书中，对于这种互斥访问有一个很形象的比喻：想象你在飞机上如厕，这时卫生间的信息牌上显示“有人”，你必须等里面的人出来后才可进去。这就是互斥的含义。

下面来看 Mutex 的实现方式，它们都很简单。

##### (1) Mutex 介绍

其代码如下所示：

👉 [-->Thread.h::Mutex 的声明和实现]

---

```
inline Mutex::Mutex(int type, const char* name) {
    if (type == SHARED) {
        //type 如果是 SHARED，则表明这个 Mutex 支持跨进程的线程同步。
        // 以后我们在 Audio 系统和 Surface 系统中会经常见到这种用法。
        pthread_mutexattr_t attr;
        pthread_mutexattr_init(&attr);
        pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
        pthread_mutex_init(&mMutex, &attr);
        pthread_mutexattr_destroy(&attr);
    } else {
        pthread_mutex_init(&mMutex, NULL);
    }
}

inline Mutex::~Mutex() {
    pthread_mutex_destroy(&mMutex);
}
```

---

<sup>①</sup> 本书中文版由机械工业出版社出版，原书作者 Jeffrey Richter。

```

}
inline status_t Mutex::lock() {
    return -pthread_mutex_lock(&mMutex);
}
inline void Mutex::unlock() {
    pthread_mutex_unlock(&mMutex);
}
inline status_t Mutex::tryLock() {
    return -pthread_mutex_trylock(&mMutex);
}

```

关于 Mutex 的使用，除了初始化外，最重要的是 lock 和 unlock 函数的使用，它们的用法如下：

- ❑ 要想独占卫生间，必须先调用 Mutex 的 lock 函数。这样，这个区域就被锁住了。如果这块区域之前已被别人锁住，lock 函数则会等待，直到可以进入这块区域为止。系统保证一次只有一个线程能 lock 成功。
- ❑ 当你“方便”完毕，记得调用 Mutex 的 unlock 以释放互斥区域。这样，其他人的 lock 才可以成功返回。
- ❑ 另外，Mutex 还提供了一个 trylock 函数，该函数只是尝试去锁住该区域，使用者需要根据 trylock 的返回值来判断是否成功锁住了该区域。

**注意** 以上这些内容都和 Raw API 有关，不了解它的读者可自行学习相关知识。在 Android 系统中，多线程也是常见和重要的编程手段，务必请大家重视。

Mutex 类确实比 Raw API 方便好用，不过还是稍显麻烦。

## (2) AutoLock 介绍

AutoLock 类是定义在 Mutex 内部的一个类，它其实是一帮“懒人”搞出来的，为什么这么说呢？先来看看使用 Mutex 有多麻烦：

- ❑ 显示调用 Mutex 的 lock。
- ❑ 在某个时候记住要调用该 Mutex 的 unlock。

以上这些操作都必须一一对应，否则会出现“死锁”！在有些代码中，如果判断分支特别多，你会发现 unlock 这句代码被写得比比皆是，如果稍有不慎，在某处就会忘了写它。有什么好办法能解决这个问题吗？终于有人想出来一个好办法，就是充分利用了 C++ 的构造和析构函数，只需看一看 AutoLock 的定义就会明白。代码如下所示：

👉 [-->Thread.h Mutex::Autolock 声明和实现]

```

class Autolock {
public:
    // 构造的时候调用 lock。
    inline Autolock(Mutex& mutex) : mLock(mutex) { mLock.lock(); }
    inline Autolock(Mutex* mutex) : mLock(*mutex) { mLock.lock(); }

```

```

        // 析构的时候调用 unlock。
        inline ~Autolock() { mLock.unlock(); }
private:
    Mutex& mLock;
};

```

AutoLock 的用法很简单：

❑ 先定义一个 Mutex，如 Mutex xlock。

❑ 在使用 xlock 的地方，定义一个 AutoLock，如 AutoLock autoLock (xlock)。

由于 C++ 对象的构造和析构函数都是自动被调用的，所以在 AutoLock 的生命周期内，xlock 的 lock 和 unlock 也就自动被调用了，这样就省去了重复书写 unlock 的麻烦，而且 lock 和 unlock 的调用肯定是一一对应的，这样就绝对不会出错。

## 2. 条件类——Condition

多线程同步中的条件类对应的是下面这种使用场景：

线程 A 做初始化工作，而其他线程比如线程 B、C 必须等到初始化工作完后才能工作，即线程 B、C 在等待一个条件，我们称 B、C 为等待者。

当线程 A 完成初始化工作时，会触发这个条件，那么等待者 B、C 就会被唤醒。触发这个条件的 A 就是触发者。

上面的使用场景非常形象，而且条件类提供的函数也非常形象，它的代码如下所示：

👉 [-->Thread.h:: Condition 的声明和实现]

```

class Condition {
public:
    enum {
        PRIVATE = 0,
        SHARED = 1
    };

    Condition();
    Condition(int type); // 如果 type 是 SHARED，表示支持跨进程的条件同步
    ~Condition();
    // 线程 B 和 C 等待事件，wait 这个名字是不是很形象呢？
    status_t wait(Mutex& mutex);
    // 线程 B 和 C 的超时等待，B 和 C 可以指定等待时间，当超过这个时间，条件却还不满足，则退出等待。
    status_t waitRelative(Mutex& mutex, nsecs_t reltime);
    // 触发者 A 用来通知条件已经满足，但是 B 和 C 只有一个会被唤醒。
    void signal();
    // 触发者 A 用来通知条件已经满足，所有等待者都会被唤醒。
    void broadcast();

private:
#ifdef HAVE_PTHREADS
    pthread_cond_t mCond;
#endif
};

```

```

#else
    void*    mState;
#endif
}

```

声明很简单，定义也很简单，代码如下所示：

```

inline Condition::Condition() {
    pthread_cond_init(&mCond, NULL);
}
inline Condition::Condition(int type) {
    if (type == SHARED) { // 设置跨进程的同步支持。
        pthread_condattr_t attr;
        pthread_condattr_init(&attr);
        pthread_condattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
        pthread_cond_init(&mCond, &attr);
        pthread_condattr_destroy(&attr);
    } else {
        pthread_cond_init(&mCond, NULL);
    }
}
inline Condition::~Condition() {
    pthread_cond_destroy(&mCond);
}
inline status_t Condition::wait(Mutex& mutex) {
    return -pthread_cond_wait(&mCond, &mutex.mMutex);
}
inline status_t Condition::waitRelative(Mutex& mutex, nsecs_t reltime) {
#ifdef HAVE_PTHREAD_COND_TIMEDWAIT_RELATIVE
    struct timespec ts;
    ts.tv_sec = reltime/1000000000;
    ts.tv_nsec = reltime%1000000000;
    return -pthread_cond_timedwait_relative_np(&mCond, &mutex.mMutex, &ts);
    ..... // 有些系统没有实现 POSIX 的相关函数，所以不同的系统需要调用不同的函数。
#endif
}
inline void Condition::signal() {
    pthread_cond_signal(&mCond);
}
inline void Condition::broadcast() {
    pthread_cond_broadcast(&mCond);
}

```

可以看出，Condition 的实现全是凭借调用了 Raw API 的 pthread\_cond\_xxx 函数。这里要重点说明的是，Condition 类必须配合 Mutex 来使用。什么意思？

在上面的代码中，不论是 wait、waitRelative、signal 还是 broadcast 的调用，都放在一个 Mutex 的 lock 和 unlock 范围中，尤其是 wait 和 waitRelative 函数的调用，这是强制性的。

来看一个实际的例子，加深一下对 Condition 类和 Mutex 类的印象。这个例子是 Thread

类的 `requestExitAndWait`，目的是等待工作线程退出，代码如下所示：

👉 [-->Thread.cpp]

---

```
status_t Thread::requestExitAndWait()
{
    .....
    requestExit(); // 设置退出变量 mExitPending 为 true。
    Mutex::Autolock _l(mLock); // 使用 Autolock, mLock 被锁住。
    while (mRunning == true) {
        /*
         * 条件变量的等待，这里为什么要通过 while 循环来反复检测 mRunning？
         * 因为某些时候即使条件类没有被触发，wait 也会返回。关于这个问题，强烈建议读者阅读
         * 前面推荐的《Programming with POSIX Thread》一书。
         */
        mThreadExitedCondition.wait(mLock);
    }

    mExitPending = false;
    // 退出前，局部变量 Mutex::Autolock _l 的析构会被调用，unlock 也就会被自动调用。
    return mStatus;
}
```

---

那么，什么时候会触发这个条件呢？是在工作线程退出前。其代码如下所示：

👉 [-->Thread.cpp]

---

```
int Thread::_threadLoop(void* user)
{
    Thread* const self = static_cast<Thread*>(user);
    sp<Thread> strong(self->mHoldSelf);
    wp<Thread> weak(strong);
    self->mHoldSelf.clear();

    do {
        .....
        result = self->threadLoop(); // 调用子类的 threadLoop 函数。
        .....
        // 如果 mExitPending 为 true，则退出。
        if (result == false || self->mExitPending) {
            self->mExitPending = true;
            // 退出前触发条件变量，唤醒等待者。
            self->mLock.lock(); // lock 锁住。
            // mRunning 的修改位于锁的保护中。如果你阅读了前面推荐的书，这里也就不难理解了。
            self->mRunning = false;
            self->mThreadExitedCondition.broadcast();
            self->mLock.unlock(); // 释放锁。
            break; // 退出循环，此后该线程函数会退出。
        }
        .....
    }
```

---

```

    } while(strong != 0);

    return 0;
}

```

关于 Android 多线程的同步类，暂时介绍到此吧。当然，这些类背后所隐含的知识及技术是读者需要倍加重视的。

**提示** 希望我们能养成一种由点及面的学习方法。以我们的同步类为例，假设你是第一次接触多线程编程，也学会了如何使用 Mutex 和 Condition 这两个类，不妨以这两个类代码中所传递的知识作为切入点，把和多线程相关的所有知识（这个知识不仅仅是函数的使用，还包括多线程的原理，多线程的编程模型，甚至是现在很热门的并行多核编程）普遍了解一下。只有深刻理解并掌握了原理等基础和框架性的知识后，才能以不变应万变，才能做到游刃有余。

### 3. 原子操作函数介绍

什么是原子操作？所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就是说，原子操作是最小的执行单位。

上面这句话放到代码中是什么意思？请看一个例子：

👉 [--> 例子]

```

static int g_flag = 0; // 全局变量 g_flag
static Mutex lock ;// 全局的锁
// 线程 1 执行 thread1。
void thread1()
{
    //g_flag 递减，每次操作前锁住。
    lock.lock();
    g_flag--;
    lock.unlock();
}
// 线程 2 中执行 thread2 函数。
void thread2()
{
    lock.lock();
    g_flag++; // 线程 2 对 g_flag 进行递增操作，每次操作前要取得锁。
    lock.unlock();
}

```

为什么需要 Mutex 来帮忙呢？因为 g\_flag++ 或 g\_flag-- 操作都不是原子操作。从汇编指令的角度看，C/C++ 中的一条语句对应了数条汇编指令。以 g\_flag++ 操作为例，它生成的汇编指令可能就是以下三条：

- ❑ 从内存中取数据到寄存器。
- ❑ 对寄存器中的数据进行递增操作，结果还在寄存器中。
- ❑ 寄存器的结果写回内存。

这三条汇编指令，如果按正常的顺序连续执行是没有问题的，但在多线程时就不能保证了。例如，线程 1 在执行第一条指令后，线程 2 由于调度的原因，抢在线程 1 之前连续执行完了三条指令。这样，线程 1 继续执行指令时，它所使用的值就不是线程 2 更新后的值，而是之前的旧值。再对这个值进行操作便没有意义了。

在一般情况下，处理这种问题可以使用 Mutex 来加锁保护，但 Mutex 的使用方法比它所保护的内容还要复杂，例如，锁的使用将导致从用户态转入内核态，有较大的浪费。那么，有没有简便些的办法让这些加、减等操作不被中断呢？

答案是肯定的，但这需要 CPU 的支持。在 X86 平台上，一个递增操作可以用下面的内嵌汇编语句来实现：

```
#define LOCK "lock;"
INT32 InterlockedIncrement(INT32* lpAddend)
{
    /*
     * 这是我们在 Linux 平台上实现 Windows API 时使用的方法。
     * 其中在 SMP 系统上，LOCK 定义成 "lock;" 表示锁总线，这样同一时刻就只能有一个 CPU 访问总线了。
     * 非 SMP 系统，LOCK 定义成空。由于 InterlockedIncrement 要返回递增前的旧值，所以我们
     * 使用了 xaddl 指令，它先交换源和目的的操作数，再进行递增操作。
     */
    INT32 i = 1;
    __asm__ __volatile__(
        LOCK "xaddl %0, %1"
        : "+r" (i), "+m" (*lpAddend)
        : : "memory");
    return *lpAddend;
}
```

Android 提供了相关的原子操作函数。这里有必要介绍一下各个函数的作用。

[-->Atomic.h]，注意该文件位于 system/core/include/cutils 目录中。

```
// 原子赋值操作，结果是 *addr=value。
void android_atomic_write(int32_t value, volatile int32_t* addr);
// 下面所有函数的返回值都是操作前的旧值。
// 原子加 1 和原子减 1。
int32_t android_atomic_inc(volatile int32_t* addr);
int32_t android_atomic_dec(volatile int32_t* addr);
// 原子加法操作，value 为被加数。
int32_t android_atomic_add(int32_t value, volatile int32_t* addr);
// 原子“与”和“或”操作。
int32_t android_atomic_and(int32_t value, volatile int32_t* addr);
int32_t android_atomic_or(int32_t value, volatile int32_t* addr);
/*
条件交换的原子操作。只有在 oldValue 等于 *addr 时，才会把 newValue 赋值给 *addr。
```

这个函数的返回值须特别注意。返回值非零，表示没有进行赋值操作。返回值为零，表示进行了原子操作。

```
*/
int android_atomic_cmpxchg(int32_t oldvalue, int32_t newvalue,
                           volatile int32_t* addr);
```

有兴趣的话，读者可以对上述函数的实现进行深入研究，其中：

❑ X86 平台的实现在 system/core/libcutils/Atomic.c 中，注意其代码在 #elif defined(\_\_i386\_\_) || defined(\_\_x86\_64\_\_) 所包括的代码段内。

❑ ARM 平台的实现在 system/core/libcutils/atomic-android-arm.S 汇编文件中。

原子操作的最大好处在于避免了锁的使用，这对整个程序运行效率的提高有很大帮助。目前，在多核并行编程中，最高境界就是完全不使用锁。当然，它的难度可想而知是巨大的。

## 5.4 Looper 和 Handler 类分析

就应用程序而言，Android 系统中 Java 的应用程序和其他系统上相同，都是靠消息驱动来工作的，它们大致的工作原理如下：

- ❑ 有一个消息队列，可以往这个消息队列中投递消息。
- ❑ 有一个消息循环，不断从消息队列中取出消息，然后处理。

我们用图 5-1 来展示这个工作过程：

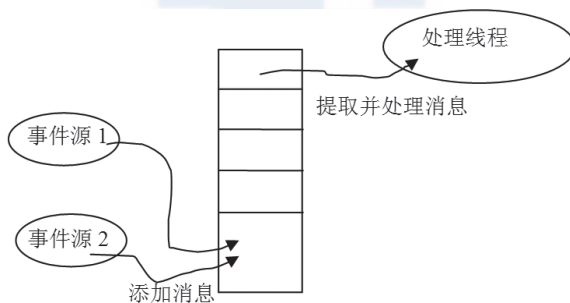


图 5-1 线程和消息处理的原理图

从图中可以看出：

- ❑ 事件源把待处理的消息加入到消息队列中，一般是加至队列尾，一些优先级高的消息也可以加至队列头。事件源提交的消息可以是按键、触摸屏等物理事件产生的消息，也可以是系统或应用程序本身发出的请求消息。
- ❑ 处理线程不断从消息队列头中取出消息并处理，事件源可以把优先级高的消息放到队列头，这样，优先级高的消息就会首先被处理。

在 Android 系统中，这些工作主要由 Looper 和 Handler 来实现：

- ❑ Looper 类，用于封装消息循环，并且有一个消息队列。



❑ Handler 类，有点像辅助类，它封装了消息投递、消息处理等接口。  
 Looper 类是其中的关键。先来看看它是怎么做的。

### 5.4.1 Looper 类分析

我们以 Looper 使用的一个常见例子来分析这个 Looper 类。

👉 [--> 例子 1]

---

```
// 定义一个 LooperThread。
class LooperThread extends Thread {
    public Handler mHandler;
    public void run() {
        // ① 调用 prepare。
        Looper.prepare();
        .....
        // ② 进入消息循环。
        Looper.loop();
    }
}
// 应用程序使用 LooperThread。
{
    .....
    new LooperThread().start(); // 启动新线程，线程函数是 run
}
```

---

上面的代码一共有两个关键调用（即①和②），我们对其逐一进行分析。

#### 1. 准备好了吗

第一个调用函数是 Looper 的 prepare 函数。它会做什么工作呢？其代码如下所示：

👉 [--> Looper.java]

---

```
public static final void prepare() {
    // 一个 Looper 只能调用一次 prepare。
    if (sThreadLocal.get() != null) {
        throw new RuntimeException("Only one Looper may be created per thread");
    }
    // 构造一个 Looper 对象，设置到调用线程的局部变量中。
    sThreadLocal.set(new Looper());
}
//sThreadLocal 定义
private static final ThreadLocal sThreadLocal = new ThreadLocal();
```

---

ThreadLocal 是 Java 中的线程局部变量类，全名应该是 Thread Local Variable。我觉得它的实现和操作系统提供的线程本地存储（TLS）有关系。总之，该类有两个关键函数：

- ❑ set：设置调用线程的局部变量。
- ❑ get：获取调用线程的局部变量。

**注意** set/get 的结果都和调用这个函数的线程有关。ThreadLocal 类可参考 JDK API 文档或 Android API 文档。

根据上面的分析可知，prepare 会在调用线程的局部变量中设置一个 Looper 对象。这个调用线程就是 LooperThread 的 run 线程。先看看 Looper 对象的构造，其代码如下所示：

👉 [-->Looper.java]

```
private Looper() {
    // 构造一个消息队列。
    mQueue = new MessageQueue();
    mRun = true;
    // 得到当前线程的 Thread 对象。
    mThread = Thread.currentThread();
}
```

prepare 函数很简单，它主要干了一件事：

在调用 prepare 的线程中，设置了一个 Looper 对象，这个 Looper 对象就保存在这个调用线程的 TLV 中。而 Looper 对象内部封装了一个消息队列。

也就是说，prepare 函数通过 ThreadLocal 机制，巧妙地把 Looper 和调用线程关联在一起了。要了解这样做的目的是什么，需要再看第二个重要函数。

## 2. Looper 循环

代码如下所示：

👉 [-->Looper.java]

```
public static final void loop() {
    Looper me = myLooper(); // myLooper 返回保存在调用线程 TLV 中的 Looper 对象。
    // 取出这个 Looper 的消息队列。
    MessageQueue queue = me.mQueue;
    while (true) {
        Message msg = queue.next();
        // 处理消息，Message 对象中有一个 target，它是 Handler 类型。
        // 如果 target 为空，则表示需要退出消息循环。
        if (msg != null) {
            if (msg.target == null) {
                return;
            }
            // 调用该消息的 Handler，交给它的 dispatchMessage 函数处理。
            msg.target.dispatchMessage(msg);
            msg.recycle();
        }
    }
}

// myLooper 函数返回调用线程的线程局部变量，也就是存储在其中的 Looper 对象。
public static final Looper myLooper() {
```

```
        return (Looper)sThreadLocal.get();
    }
}
```

---

通过上面的分析会发现，Looper 的作用是：

- ❑ 封装了一个消息队列。
- ❑ Looper 的 prepare 函数把这个 Looper 和调用 prepare 的线程（也就是最终的处理线程）绑定在一起了。
- ❑ 处理线程调用 loop 函数，处理来自该消息队列的消息。

当事件源向这个 Looper 发送消息的时候，其实是把消息加到这个 Looper 的消息队列里了。那么，该消息就将由和 Looper 绑定的处理线程来处理。可事件源又是怎么向 Looper 消息队列添加消息的呢？来看下一节。

### 3. Looper、Message 和 Handler 的关系

Looper、Message 和 Handler 之间也存在暧昧关系，不过要比 RefBase 那三个简单得多，用两句话就可以说清楚：

- ❑ Looper 中有一个 Message 队列，里面存储的是一个个待处理的 Message。
- ❑ Message 中有一个 Handler，这个 Handler 是用来处理 Message 的。

其中，Handler 类封装了很多琐碎的工作。先来认识一下这个 Handler。

## 5.4.2 Handler 分析

### 1. 初识 Handler

Handler 中所包括的成员：

👉 [-->Handler.java]

```
final MessageQueue mQueue; //Handler 中也有一个消息队列。
final Looper mLooper; // 也有一个 Looper。
final Callback mCallback; // 有一个回调用的类。
```

---

这几个成员变量是怎么使用的呢？这首先得分析 Handler 的构造函数。Handler 一共有四个构造函数，它们主要的区别是在对上面三个重要成员变量的初始化上。我们试对其进行逐一的分析。

👉 [-->Handler.java]

```
// 构造函数 1
public Handler() {
    // 获得调用线程的 Looper。
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(.....);
    }
}
```

```

        // 得到 Looper 的消息队列。
        mQueue = mLooper.mQueue;
        // 无 callback 设置。
        mCallback = null;
    }

    // 构造函数 2
    public Handler(Callback callback) {
        mLooper = Looper.myLooper();
        if (mLooper == null) {
            throw new RuntimeException(.....);
        }
        // 和构造函数 1 类似，只不过多了一个设置 callback。
        mQueue = mLooper.mQueue;
        mCallback = callback;
    }

    // 构造函数 3
    public Handler(Looper looper) {
        mLooper = looper; //looper 由外部传入，是哪个线程的 Looper 不确定。
        mQueue = looper.mQueue;
        mCallback = null;
    }

    // 构造函数 4，和构造函数 3 类似，只不过多了 callback 设置。
    public Handler(Looper looper, Callback callback) {
        mLooper = looper;
        mQueue = looper.mQueue;
        mCallback = callback;
    }
}

```

在上述构造函数中，Handler 中的消息队列变量最终都会指向 Looper 的消息队列，Handler 为何要如此做？

## 2. Handler 的真面目

根据前面的分析可知，Handler 中的消息队列实际就是某个 Looper 的消息队列，那么，Handler 如此安排的目的何在？

在回答这个问题之前，我先来问一个问题：

怎么往 Looper 的消息队列插入消息？

如果不知道 Handler，这里有一个很原始的方法可解决上面这个问题：

- ❑ 调用 Looper 的 myQueue，它将返回消息队列对象 MessageQueue。
- ❑ 构造一个 Message，填充它的成员，尤其是 target 变量。
- ❑ 调用 MessageQueue 的 enqueueMessage，将消息插入消息队列。

这种原始方法的确很麻烦，且极容易出错。但有了 Handler 后，我们的工作就变得异常简单了。Handler 更像一个辅助类，帮助我们简化编程的工作。

### (1) Handler 和 Message

Handler 提供了一系列函数，帮助我们完成创建消息和插入消息队列的工作。这里只列举其中一二。要掌握详细的 API，则需要查看相关的文档。

```
// 查看消息队列中是否有消息码是 what 的消息。
final boolean      hasMessages(int what)
// 从 Handler 中创建一个消息码是 what 的消息。
final Message      obtainMessage(int what)
// 从消息队列中移除消息码是 what 的消息。
final void          removeMessages(int what)
// 发送一个只填充了消息码的消息。
final boolean       sendEmptyMessage(int what)
// 发送一个消息，该消息添加到队列尾。
final boolean       sendMessage(Message msg)
// 发送一个消息，该消息添加到队列头，所以优先级很高。
final boolean       sendMessageAtFrontOfQueue(Message msg)
```

只需对上面这些函数稍作分析，就能明白其他的函数。现以 sendMessage 为例，其代码如下所示：

 [-->Handler.java]

---

```
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0); // 调用 sendMessageDelayed
}
```

---

 [-->Handler.java]

---

```
// delayMillis 是以当前调用时间为基础的相对时间
public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    // 调用 sendMessageAtTime，把当前时间算上
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
```

---

 [-->Handler.java]

---

```
//uptimeMillis 是绝对时间，即 sendMessageAtTime 函数处理的是绝对时间
public boolean sendMessageAtTime(Message msg, long uptimeMillis){
    boolean sent = false;
    MessageQueue queue = mQueue;
    if (queue != null) {
        // 把 Message 的 target 设置为自己，然后加入到消息队列中
        msg.target = this;
        sent = queue.enqueueMessage(msg, uptimeMillis);
    }
}
```

---

```

        return sent;
    }

```

看到上面这些函数我们可以预见，如果没有 Handler 的辅助，当我们自己操作 MessageQueue 的 enqueueMessage 时，得花费多大工夫！

Handler 把 Message 的 target 设为自己，是因为 Handler 除了封装消息添加等功能外还封装了消息处理的接口。

## (2) Handler 的消息处理

刚才，我们往 Looper 的消息队列中加入了一个消息，按照 Looper 的处理规则，它在获取消息后会调用 target 的 dispatchMessage 函数，再把这个消息派发给 Handler 处理。Handler 在这块是如何处理消息的呢？

👉 [-->Handler.java]

```

public void dispatchMessage(Message msg) {
    // 如果 Message 本身有 callback，则直接交给 Message 的 callback 处理
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        // 如果本 Handler 设置了 mCallback，则交给 mCallback 处理
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        // 最后才是交给子类处理
        handleMessage(msg);
    }
}

```

dispatchMessage 定义了一套消息处理的优先级机制，它们分别是：

- ❑ Message 如果自带了 callback 处理，则交给 callback 处理。
- ❑ Handler 如果设置了全局的 mCallback，则交给 mCallback 处理。
- ❑ 如果上述都没有，该消息则会被交给 Handler 子类实现的 handleMessage 来处理。当然，这需从 Handler 派生并重载 handleMessage 函数。

在通常情况下，我们一般都是采用第三种方法，即在子类中通过重载 handleMessage 来完成处理工作的。

至此，Handler 知识基本上讲解完了，可是在实际编码过程中还有一个重要问题需要警惕，下一节内容就会谈及此问题。

### 5.4.3 Looper 和 Handler 的同步关系

Looper 和 Handler 会有什么同步关系呢？它们之间确实有同步关系，而且如果不注意此

关系，定会铸成大错！

同步关系肯定与多线程有关，我们来看下面的一个例子：

👉 [--> 例子 2]

---

```
// 先定义一个 LooperThread 类
class LooperThread extends Thread {
    public Looper myLooper = null; // 定义一个 public 的成员 myLooper，初值为空。
    public void run() { // 假设 run 在线程 2 中执行
        Looper.prepare();
        // myLooper 必须在这个线程中赋值
        myLooper = Looper.myLooper();
        Looper.loop();
    }
}
// 下面这段代码在线程 1 中执行，并且会创建线程 2
{
    LooperThread lpThread = new LooperThread();
    lpThread.start(); // start 后会创建线程 2
    Looper looper = lpThread.myLooper; // <===== 注意
    // thread2Handler 和线程 2 的 Looper 挂上钩
    Handler thread2Handler = new Handler(looper);
    // sendMessage 发送的消息将由线程 2 处理
    threadHandler.sendMessage(...)
}
```

---

上面这段代码的目的很简单：

- ❑ 线程 1 中创建线程 2，并且线程 2 通过 Looper 处理消息。
- ❑ 线程 1 中得到线程 2 的 Looper，并且根据这个 Looper 创建一个 Handler，这样发送给该 Handler 的消息将由线程 2 处理。

但很可惜，上面的代码是有问题的。如果我们熟悉多线程，就会发现标有“注意”的那行代码存在着严重问题。myLooper 的创建是在线程 2 中，而 looper 的赋值在线程 1 中，很有可能此时线程 2 的 run 函数还没来得及给 myLooper 赋值，这样线程 1 中的 looper 将取到 myLooper 的初值，也就是 looper 等于 null。另外，

```
Handler thread2Handler = new Handler(looper) 不能替换成
Handler thread2Handler = new Handler(Looper.myLooper())
```

这是因为，myLooper 返回的是调用线程的 Looper，即 Thread1 的 Looper，而不是我们想要的 Thread2 的 Looper。

对这个问题，可以采用同步的方式进行处理。你是不是有点迫不及待地想完善这个例子了？其实 Android 早就替我们想好了，它提供了一个 HandlerThread 来解决这个问题。

## 5.4.4 HandlerThread 介绍

HandlerThread 完美地解决了 myLooper 可能为空的问题。下面来看看它是怎么做的，代码如下所示：

👉 [-->HandlerThread]

---

```
public class HandlerThread extends Thread{
// 线程1 调用 getLooper 来获得新线程的 Looper
public Looper getLooper() {
    .....
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait(); // 如果新线程还未创建 Looper, 则等待
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}

// 线程2 运行它的 run 函数, looper 就是在 run 线程里创建的。
public void run() {
    mTid = Process.myTid();
    Looper.prepare(); // 创建这个线程上的 Looper
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll(); // 通知取 Looper 的线程1, 此时 Looper 已经创建好了。
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
}
```

---

HandlerThread 很简单，小小的 wait/ notifyAll 就解决了我们的难题。为了避免重复发明轮子，我们还是多用 HandlerThread 类吧！

## 5.5 本章小结

本章主要分析了 Android 代码中最常见的几个类：其中在 Native 层包括与对象生命周期相关的 RefBase、sp、wp、LightRefBase 类，以及 Android 为多线程编程提供的 Thread 类和相关的同步类；Java 层则包括使用最为广泛的 Handler 类和 Looper 类。另外，还分析了类 HandlerThread，它降低了创建和使用带有消息队列的线程的难度。





## 第 6 章

# 深入理解 Binder

### 本章涉及的源代码文件名及位置

下面是本章分析的源码文件名及其位置。

- ❑ Main\_mediaserver.cpp(`framework/base/Media/MediaServer/Main_mediaserver.cpp`)
- ❑ Static.cpp(`framework/base/libs/binder/Static.cpp`)
- ❑ ProcessState.cpp(`framework/base/libs/binder/ProcessState.cpp`)
- ❑ IServiceManager.cpp(`framework/base/libs/binder/IServiceManager.cpp`)
- ❑ BpBinder.cpp(`framework/base/libs/binder/BpBinder.cpp`)
- ❑ IInterface.h(`framework/base/include/binder/IInterface.h`)
- ❑ IServiceManager.h(`framework/base/include/binder/IServiceManager.h`)
- ❑ IServiceManager.cpp(`framework/base/libs/binder/IServiceManager.cpp`)
- ❑ binder.cpp(`framework/base/libs/binder/binder.cpp`)
- ❑ MediaPlayerService.cpp(`framework/base/media/libmediaplayerservice/MediaPlayerService.cpp`)
- ❑ IPCThreadState.cpp(`framework/base/libs/binder/IPCThreadState.cpp`)
- ❑ binder\_module.h(`framework/base/include/private/binder.h`)
- ❑ Service\_manager.c(`framework/base/cmds/ServiceManager/Service_manager.c`)
- ❑ Binder.c(`framework/base/cmds/ServiceManager/Binder.c`)
- ❑ IMediaDeathNotifier(`framework/base/media/libmedia/IMediaDeathNotifier.cpp`)
- ❑ MediaMetadataRetriever(`framework/base/media/libmedia/MediaMetadataRetriever.cpp`)

## 6.1 概述

Binder 是 Android 系统提供了一种 IPC（进程间通信）机制。由于 Android 是基于 Linux 内核的，因此，除了 Binder 以外，还存在其他的 IPC 机制，例如管道和 socket 等。Binder 相对于其他 IPC 机制来说，就更加灵活和方便了。对于初学 Android 的朋友而言，最难却又最想掌握的恐怕就是 Binder 机制了，因为 Android 系统基本上可以看作是一个基于 Binder 通信的 C/S 架构。Binder 就像网络一样，把系统的各个部分连接在了一起，因此它是非常重要的。

在基于 Binder 通信的 C/S 架构体系中，除了 C/S 架构所包括的 Client 端和 Server 端外，Android 还有一个全局的 ServiceManager 端，它的作用是管理系统中的各种服务（Service）。Client、Server 和 ServiceManager 这三者之间的交互关系如图 6-1 所示：

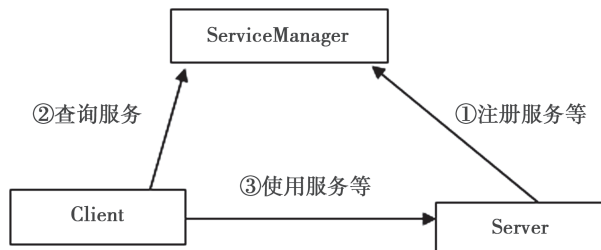


图 6-1 Client、Server 和 ServiceManager 三者之间的交互关系

---

**注意** 一个 Server 进程可以注册多个 Service，就像即将讲解的 MediaServer 一样。

---

根据图 6-1，可以得出如下结论：

- ❑ Server 进程要先注册一些 Service 到 ServiceManager 中，所以 Server 是 ServiceManager 的客户端，而 ServiceManager 就是服务端了。
- ❑ 如果某个 Client 进程要使用某个 Service，必须先到 ServiceManager 中获取该 Service 的相关信息，所以 Client 是 ServiceManager 的客户端。
- ❑ Client 根据得到的 Service 信息与 Service 所在的 Server 进程建立通信的通路，然后就可以直接与 Service 交互了，所以 Client 也是 Server 的客户端。
- ❑ 最重要的一点是，三者的交互都是基于 Binder 通信的，所以通过任意两者之间的关系，都可以揭示 Binder 的奥秘。

这里要重点强调的是，Binder 通信与 C/S 架构之间的关系。Binder 只是为 C/S 架构提供了一种通信的方式，我们完全可以采用其他 IPC 方式进行通信，例如，系统中有很多其他的程序就是采用 Socket 或 Pipe 方法进行进程间通信的。很多初学者可能会觉得 Binder 较复杂，尤其是看到诸如 BpXXX、BnXXX 之类的定义便感到头晕，这很有可能是把 Binder 通信层结构和应用的业务层结构搞混了。如果能搞清楚这二者的关系，完全可以自己实现一个不使用 BpXXX 和 BnXXX 的 Service。须知，ServiceManager 可并没有使用它们。

## 6.2 庖丁解 MediaServer

为了能像“庖丁”那样解析 Binder，我们必须得找一头“牛”来做解剖，而 MediaServer（简称 MS）正是一头比较好的“牛”。它是一个可执行程序，虽然 Android 的 SDK 提供 Java 层的 API，但 Android 系统本身还是一个完整的基于 Linux 内核的操作系统，所以并非所有的程序都是用 Java 编写的，这里的 MS 就是一个用 C++ 编写的可执行程序。

之所以选择 MediaServer 作为切入点，是因为这个 Server 是系统诸多重要 Service 的栖息地，它们包括：

- ❑ AudioFlinger：音频系统中的核心服务。
- ❑ AudioPolicyService：音频系统中关于音频策略的重要服务。
- ❑ MediaPlayerService：多媒体系统中的重要服务。
- ❑ CameraService：有关摄像 / 照相的重要服务。

可以看到，MS 除了不涉及 Surface 系统外，其他重要的服务基本上都涉及了，它不愧是“庖丁”所要的好“牛”。

这里以其中的 MediaPlayerService 为主切入点进行分析。先来分析 MediaServer 本身。

### 6.2.1 MediaServer 的入口函数

MS 是一个可执行程序，入口函数是 main，代码如下所示：

 [-->Main\_MediaServer.cpp]

---

```
int main(int argc, char** argv)
{
    // ①获得一个 ProcessState 实例。
    sp<ProcessState> proc(ProcessState::self());

    // ② MS 作为 ServiceManager 的客户端，需要向 ServiceManager 注册服务。
    // 调用 defaultServiceManager，得到一个 IServiceManager。
    sp<IServiceManager> sm = defaultServiceManager();

    // 初始化音频系统的 AudioFlinger 服务。
    AudioFlinger::instantiate();
    // ③多媒体系统的 MediaPlayer 服务，我们将以它作为主切入点。
    MediaPlayerService::instantiate();
    // CameraService 服务。
    CameraService::instantiate();
    // 音频系统的 AudioPolicy 服务。
    AudioPolicyService::instantiate();

    // ④根据名称来推断，难道是要创建一个线程池吗？
    ProcessState::self()->startThreadPool();
    // ⑤下面的操作是要将自己加入到刚才的线程池中吗？
    IPCThreadState::self()->joinThreadPool();
}
```

---

上面的代码中，确定了5个关键点（即①～⑤），让我们通过对这5个关键点逐一进行深入分析，来认识和理解 Binder。

## 6.2.2 独一无二的 ProcessState

我们在 main 函数的开始处便碰见了 ProcessState。由于每个进程只有一个 ProcessState，所以它是独一无二的。它的调用方式如下面的代码所示：

👉 [-->Main\_MediaServer.cpp]

---

```
// ①获得一个 ProcessState 实例。
sp<ProcessState> proc(ProcessState::self());
```

---

下面来进一步分析这个独一无二的 ProcessState。

### 1. 单例的 ProcessState

ProcessState 的代码如下所示：

👉 [-->ProcessState.cpp]

---

```
sp<ProcessState> ProcessState::self()
{
    //gProcess 是在 Static.cpp 中定义的一个全局变量。
    // 程序刚开始执行，gProcess 一定为空。
    if (gProcess != NULL) return gProcess;
    AutoMutex _l(gProcessMutex);
    // 创建一个 ProcessState 对象，并赋值给 gProcess。
    if (gProcess == NULL) gProcess = new ProcessState;

    return gProcess;
}
```

---

self 函数采用了单例模式，根据这个以及 Process State 的名字这很明确地告诉了我们一个信息：每个进程只有一个 ProcessState 对象。这一点，从它的命名中也可看出些端倪。

### 2. ProcessState 的构造

再来看 ProcessState 的构造函数。这个函数非常重要，它悄悄地打开了 Binder 设备。代码如下所示：

👉 [-->ProcessState.cpp]

---

```
ProcessState::ProcessState()
    // Android 中有很多代码都是这么写的，稍不留神就容易忽略这里调用了 一个很重要的函数。
    : mDriverFD(open_driver())
    , mVMStart(MAP_FAILED) // 映射内存的起始地址。
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
```

---

```

        , mBinderContextUserData(NULL)
        , mThreadPoolStarted(false)
        , mThreadPoolSeq(1)
    {
        if (mDriverFD >= 0) {
            /*
             * BIDNER_VM_SIZE 定义为 (1*1024*1024) - (4096 * 2) = 1M-8K
             * mmap 的用法希望读者 man 一下，不过这个函数真正的实现和驱动有关系，而 Binder 驱动会分配一块
             * 内存来接收数据。
             */
            mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE,
                           mDriverFD, 0);
        }
        .....
    }
}

```

### 3. 打开 binder 设备

open\_driver 的作用就是打开 /dev/binder 这个设备，它是 Android 在内核中为完成进程间通信而专门设置的一个虚拟设备，具体实现如下所示：

👉 [-->ProcessState.cpp]

```

static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR); // 打开 /dev/binder 设备。
    if (fd >= 0) {
        .....
        size_t maxThreads = 15;
        // 通过 ioctl 方式告诉 binder 驱动，这个 fd 支持的最大线程数是 15 个。
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
    }
    return fd;
    .....
}

```

至此，Process::self 函数就分析完了。它到底干了什么呢？总结如下：

- ❑ 打开 /dev/binder 设备，这就相当于与内核的 Binder 驱动有了交互的通道。
  - ❑ 对返回的 fd 使用 mmap，这样 Binder 驱动就会分配一块内存来接收数据。
  - ❑ 由于 ProcessState 具有唯一性，因此一个进程只打开设备一次。
- 分析完 ProcessState，接下来将要分析第二个关键函数 defaultManager。

### 6.2.3 时空穿越魔术——defaultServiceManager

defaultServiceManager 函数的实现在 IServiceManager.cpp 中完成。它会返回一个 IServiceManager 对象，通过这个对象，我们可以神奇地与另一个进程 ServiceManager 进行交互。是不是有一种观看时空穿越魔术表演的感觉？

## 1. 魔术前的准备工作

先来看看 defaultServiceManager 都调用了哪些函数。返回的这个 IServiceManager 到底是什么？具体实现代码如下所示：

👉 [-->IServiceManager.cpp]

---

```
sp<IServiceManager> defaultServiceManager()
{
    // 看样子又是一个单例，英文名叫 Singleton，Android 是一个优秀的源码库，大量使用了
    // 设计模式，建议读者以此为契机学习设计模式，首推 GOF 的《设计模式：可复用面向对象软件的基础》。
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            // 真正的 gDefaultServiceManager 是在这里创建的。
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

---

哦，是调用了 ProcessState 的 getContextObject 函数！注意：传给它的参数是 NULL，即 0。既然是“庖丁解牛”，就还要一层一层地往下切。下面再看 getContextObject 函数，如下所示：

👉 [-->ProcessState.cpp]

---

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    /*
     * caller 的值为 0！注意，该函数返回的是 IBinder。它是什么？我们后面再说。
     * supportsProcesses 函数根据 openDriver 函数是否成功打开设备来判断它是否支持 process。
     * 真实设备肯定支持 process。
     */
    if (supportsProcesses()) {
        // 真实设备上肯定是支持进程的，所以会调用下面这个函数。
        return getStrongProxyForHandle(0);
    } else {
        return getContextObject(String16("default"), caller);
    }
}
```

---

getStrongProxyForHandle 这个函数名怪怪的，可能会让人感到些许困惑。请注意，它的调用参数名叫 handle，在 Windows 编程中经常使用这个名称，它是对资源的一种标识。说白了，其实就是有一个资源项，保存在一个资源数组（也可以是别的组织结构）中，handle 的值正是该资源项在数组中的索引。

👉 [-->ProcessState.cpp]

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    /*
    根据索引查找对应的资源。如果 lookupHandleLocked 发现没有对应的资源项，则会创建一个新的项并返回。
    这个新项的内容需要填充。
    */
    handle_entry* e = lookupHandleLocked(handle);
    if (e != NULL) {
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            // 对于新创建的资源项，它的 binder 为空，所以走这个分支。注意，handle 的值为 0。
            b = new BpBinder(handle); // 创建一个 BpBinder。
            e->binder = b; // 填充 entry 的内容。
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result; // 返回 BpBinder(handle)，注意，handle 的值为 0。
}
```

## 2. 魔术表演的道具——BpBinder

众所周知，玩魔术时必须有道具。这个穿越魔术的道具就是 BpBinder。BpBinder 是什么呢？有必要先来介绍它的孪生兄弟 BBinder。

BpBinder 和 BBinder 都是 Android 中与 Binder 通信相关的代表，它们都是从 IBinder 类中派生而来，如图 6-2 所示：

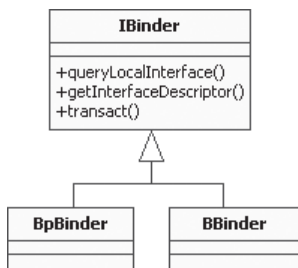


图 6-2 Binder 家族图谱

从上图中可以看出：

- BpBinder 是客户端用来与 Server 交互的代理类，p 即 Proxy 的意思。
- BBinder 则是与 proxy 相对的一端，它是 proxy 交互的目的端。如果说 Proxy 代表客



户端，那么 BBinder 则代表服务端。这里的 BpBinder 和 BBinder 是一一对应的，即某个 BpBinder 只能和对应的 BBinder 交互。我们当然不希望通过 BpBinderA 发送的请求，却由 BBinderB 来处理。

刚才我们在 defaultServiceManager() 函数中创建了这个 BpBinder。这里有两个问题：

1) 为什么创建的不是 BBinder？

因为我们是 ServiceManager 的客户端，当然得使用代理端与 ServiceManager 进行交互。

2) 前面说了，BpBinder 和 BBinder 是一一对应的，那么 BpBinder 如何标识它所对应的 BBinder 端呢？

答案是 Binder 系统通过 handler 来标识对应的 BBinder。以后我们会确认这个 Handle 值的作用。

---

**注意** 我们给 BpBinder 构造函数传的参数 handle 的值是 0。这个 0 在整个 Binder 系统中具有重要意义——因为 0 代表的就是 ServiceManager 所对应的 BBinder。

---

BpBinder 是如此重要，必须对它进行深入分析，其代码如下所示：

👉 [-->BpBinder.cpp]

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle) //handle 是 0。
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    // 另一个重要对象是 IPCThreadState，我们稍后会详细讲解。
    IPCThreadState::self()->incWeakHandle(handle);
}
```

看上面的代码，会觉得 BpBinder 确实简单，不过再仔细查看，你或许会发现，BpBinder、BBinder 这两个类没有任何地方操作 ProcessState 打开的那个 /dev/binder 设备，换言之，这两个 Binder 类没有和 binder 设备直接交互。那为什么说 BpBinder 与通信相关呢？注意本小节的标题，BpBinder 只是道具嘛！所以它后面一定还另有机关。不必急着揭秘，还是先回顾一下道具出场的过程。

我们是从下面这个函数开始分析的：

```
gDefaultServiceManager = interface_cast<IServiceManager>(
    ProcessState::self()->getContextObject(NULL));
```

现在这个函数调用将变成如下所示的样子：

```
gDefaultServiceManager = interface_cast<IServiceManager>(new BpBinder(0));
```

这里出现了一个 interface\_cast。它是什么？其实是一个障眼法！下面就来具体分析它。



### 3. 障眼法——interface\_cast

interface\_cast、dynamic\_cast 和 static\_cast 看起来是否非常眼熟？它们是指针类型转换的意思吗？如果是，那又是如何将 BpBinder\* 类型强制转化成 IServiceManager\* 类型的呢？BpBinder 的家谱我们刚才也看了，它的“爸爸的爸爸的爸爸”这条线上没有任何一个与 IServiceManager 有任何关系。

谈到这里，我们得去看看 interface\_cast 的具体实现，其代码如下所示：

👉 [-->IInterface.h]

---

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

哦，仅仅是一个模板函数，所以 interface\_cast<IServiceManager>() 等价于：

```
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
    return IServiceManager::asInterface(obj);
}
```

---

又转移到 IServiceManager 对象中去了，这难道不是障眼法吗？既然找到了“真身”，不妨就来见识见识它吧。

### 4. 拨开浮云见月明——IServiceManager

刚才提到，IBinder 家族的 BpBinder 和 BBinder 是与通信业务相关的，那么业务层的逻辑又是如何巧妙地架构在 Binder 机制上的呢？关于这些问题，可以用一个绝好的例子来解释，它就是 IServiceManager。

#### (1) 定义业务逻辑

先回答第一个问题：如何表述应用的业务层逻辑。可以先分析一下 IServiceManager 是怎么做的。IServiceManager 定义了 ServiceManager 所提供的服务，看它的定义可知，其中有很多有趣的内容。IServiceManager 定义在 IServiceManager.h 中，代码如下所示：

👉 [-->IServiceManager.h]

---

```
class IServiceManager : public IInterface
{
public:
    // 关键无比的宏！
    DECLARE_META_INTERFACE(ServiceManager);

    // 下面是 ServiceManager 所提供的业务函数。
    virtual sp<IBinder>    getService( const String16& name) const = 0;
    virtual sp<IBinder>    checkService( const String16& name) const = 0;
```

---

```

virtual status_t      addService( const String16& name,
                                   const sp<IBinder>& service) = 0;
virtual Vector<String16> listServices() = 0;
.....
};

```

## (2) 业务与通信的挂钩

Android 巧妙地通过 DECLARE\_META\_INTERFACE 和 IMPLEMENT 宏，将业务和通信牢牢地钩在了一起。DECLARE\_META\_INTERFACE 和 IMPLEMENT\_META\_INTERFACE 这两个宏都定义在刚才的 IInterface.h 中。先看 DECLARE\_META\_INTERFACE 这个宏，如下所示：

👉 [-->IInterface.h::DECLARE\_META\_INTERFACE]

```

#define DECLARE_META_INTERFACE(INTERFACE)                                \
    static const android::String16 descriptor;                          \
    static android::sp<I##INTERFACE> asInterface(                        \
        const android::sp<android::IBinder>& obj);                      \
    virtual const android::String16& getInterfaceDescriptor() const;     \
    I##INTERFACE();                                                       \
    virtual ~I##INTERFACE();

```

将 IServiceManager 的 DELCARE 宏进行相应的替换后得到的代码如下所示：

👉 [--->DECLARE\_META\_INTERFACE(IServiceManager)]

```

// 定义一个描述字符串。
static const android::String16 descriptor;

// 定义一个 asInterface 函数。
static android::sp< IServiceManager >
asInterface(const android::sp<android::IBinder>& obj)

// 定义一个 getInterfaceDescriptor 函数，估计就是返回 descriptor 字符串。
virtual const android::String16& getInterfaceDescriptor() const;

// 定义 IServiceManager 的构造函数和析构函数。
IServiceManager ();
virtual ~IServiceManager();

```

DECLARE 宏声明了一些函数和一个变量，那么，IMPLEMENT 宏的作用肯定就是定义它们了。IMPLEMENT 的定义在 IInterface.h 中，IServiceManager 是如何使用这个宏的呢？只有一行代码，在 IServiceManager.cpp 中，如下所示：

```
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

很简单，可直接将 IServiceManager 中 IMPLEMENT 宏的定义展开，如下所示：

```

const android::String16
IServiceManager::descriptor("android.os.IServiceManager");
// 实现 getInterfaceDescriptor 函数。
const android::String16& IServiceManager::getInterfaceDescriptor() const
{
    // 返回字符串 descriptor, 值是 "android.os.IServiceManager"。
    return IServiceManager::descriptor;
}
// 实现 asInterface 函数。
android::sp<IServiceManager>
IServiceManager::asInterface(const android::sp<android::IBinder>& obj)
{
    android::sp<IServiceManager> intr;
    if (obj != NULL) {
        intr = static_cast<IServiceManager *>(
            obj->queryLocalInterface(IServiceManager::descriptor).get());
        if (intr == NULL) {
            //obj 是我们刚才创建的那个 BpBinder(0)。
            intr = new BpServiceManager(obj);
        }
    }
    return intr;
}
// 实现构造函数和析构函数。
IServiceManager::IServiceManager () { }
IServiceManager::~IServiceManager() { }

```

我们曾提出过疑问：interface\_cast 是如何把 BpBinder 指针转换成一个 IServiceManager 指针的呢？答案就在 asInterface 函数的这一行代码中，如下所示：

```
intr = new BpServiceManager(obj);
```

明白了！interface\_cast 不是指针的转换，而是利用 BpBinder 对象作为参数新建了一个 BpServiceManager 对象。我们已经知道 BpBinder 和 BBinder 与通信有关系，这里怎么突然冒出来一个 BpServiceManager？它们之间又有什么关系呢？

### (3) IServiceManager 家族

要搞清这个问题，必须先了解 IServiceManager 家族之间的关系，先来看图 6-3，它展示了 IServiceManager 的家族图谱。

根据图 6-3 和相关的代码可知，这里有以下几个重要的点值得注意：

- ❑ IServiceManager、BpServiceManager 和 BnServiceManager 都与业务逻辑相关。
- ❑ BnServiceManager 同时从 IServiceManager BBinder 派生，表示它可以直接参与 Binder 通信。
- ❑ BpServiceManager 虽然从 BpInterface 中派生，但是这条分支似乎与 BpBinder 没有关系。
- ❑ BnServiceManager 是一个虚类，它的业务函数最终需要子类来实现。



👉 [-->Binder.cpp::BpRefBase 类]

---

```
BpRefBase::BpRefBase(const sp<IBinder>& o)
//mRemote 最终等于那个 new 出来的 BpBinder(0)。
: mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);
        mRefs = mRemote->createWeak(this);
    }
}
```

---

原来，是 BpServiceManager 的一个变量 mRemote 指向了 BpBinder。至此，我们的魔术表演结束，回想一下 defaultServiceManager 函数，可以得到以下两个关键对象：

- ❑ 有一个 BpBinder 对象，它的 handle 值是 0。
- ❑ 有一个 BpServiceManager 对象，它的 mRemote 值是 BpBinder。

BpServiceManager 对象实现了 IServiceManager 的业务函数，现在又有 BpBinder 作为通信的代表，接下来的工作就简单了。下面，要通过分析 MediaPlayerService 的注册过程，进一步分析业务函数的内部是如何工作的。

## 6.2.4 注册 MediaPlayerService

### 1. 业务层的工作

再回到 MS 的 main 函数，下一个要分析的是 MediaPlayerService，它的代码如下所示：

👉 [-->MediaPlayerService.cpp]

---

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
        String16("media.player"), new MediaPlayerService());
}
```

---

根据前面的分析可知，defaultServiceManager() 实际返回的对象是 BpServiceManager，它是 IServiceManager 的后代，代码如下所示：

👉 [-->IServiceManager.cpp::BpServiceManager 的 addService() 函数]

---

```
virtual status_t addService(const String16& name, const sp<IBinder>& service)
{
    //Parcel: 就把它当作是一个数据包。
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    data.writeStrongBinder(service);
```

```

//remote 返回的是 mRemote, 也就是 BpBinder 对象。
status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
return err == NO_ERROR ? reply.readInt32() : err;
}

```

别急着往下走，应先思考以下两个问题：

- ❑ 调用 BpServiceManager 的 addService 是不是一个业务层的函数？
- ❑ 在 addService 函数中把请求数据打包成 data 后，传给了 BpBinder 的 transact 函数，这是不是把通信的工作交给了 BpBinder？

两个问题的答案都是肯定的。至此，业务层的工作原理应该是很清晰了，它的作用就是将请求信息打包后，再交给通信层去处理。

## 2. 通信层的工作

下面分析 BpBinder 的 transact 函数。前面说过，在 BpBinder 中确实找不到任何与 Binder 设备交互的地方。那它是如何参与通信的呢？原来，秘密就在这个 transact 函数中，它的实现代码如下所示：

👉 [-->BpBinder.cpp]

```

status_t BpBinder::transact(uint32_t code, const Parcel& data, Parcel* reply,
                           uint32_t flags)
{
    if (mAlive) {
        //BpBinder 果然是道具，它把 transact 工作交给了 IPCThreadState。
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags); //mHandle 也是参数
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}

```

这里又遇见了 IPCThreadState，之前也见过一次。看来，它确实与 Binder 通信有关，所以必须对其进行深入分析！

### (1) “劳者一份”的 IPCThreadState

谁是“劳者”？线程，它是进程中真正干活的伙计，所以它正是劳者。而“劳者一份”，就是每个伙计一份的意思。IPCThreadState 的实现代码在 IPCThreadState.cpp 中，如下所示：

👉 [-->IPCThreadState.cpp]

```

IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) { // 第一次进来为 false。
        restart:
    }
}

```

```

        const pthread_key_t k = gTLS;
    /*
    TLS 是 Thread Local Storage (线程本地存储空间) 的简称。
    这里只需知晓：这种空间每个线程都有，而且线程间不共享这些空间。
    通过 pthread_getspecific/pthread_setspecific 函数可以获取 / 设置这些空间中的内容。
    从线程本地存储空间中获得保存在其中的 IPCThreadState 对象。
    有调用 pthread_getspecific 的地方，肯定也有调用 pthread_setspecific 的地方。
    */
    IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
    if (st) return st;
    // new 一个对象，构造函数中会调用 pthread_setspecific。
    return new IPCThreadState;
}

if (gShutdown) return NULL;
pthread_mutex_lock(&gTLSMutex);
if (!gHaveTLS) {
    if (pthread_key_create(&gTLS, threadDestructor) != 0) {
        pthread_mutex_unlock(&gTLSMutex);
        return NULL;
    }
    gHaveTLS = true;
}
pthread_mutex_unlock(&gTLSMutex);
// 其实 goto 没有我们说的那么不好，汇编代码也有很多跳转语句，关键是要用好。
goto restart;
}

```

接下来，有必要转向分析它的构造函数 IPCThreadState() 了，如下所示：

👉 [-->IPCThreadState.cpp]

```

IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()), mMyThreadId(androidGetTid())
{
    // 在构造函数中，把自己设置到线程本地存储中去。
    pthread_setspecific(gTLS, this);
    clearCaller();
    // mIn 和 mOut 是两个 Parcel。把它看成是发送和接收命令的缓冲区即可。
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}

```

每个线程都有一个 IPCThreadState，每个 IPCThreadState 中都有一个 mIn、一个 mOut，其中，mIn 是用来接收来自 Binder 设备的数据的，而 mOut 则是用来存储发往 Binder 设备的数据的。

## (2) 勤劳的 transact

传输工作是很辛苦的。我们刚才看到 BpBinder 的 transact 调用了 IPCThreadState 的

transact 函数，这个函数实际完成了与 Binder 通信的工作，如下面的代码所示：

👉 [-->IPCThreadState.cpp]

---

```
// 注意，handle 的值为 0，代表了通信的目的端。
status_t IPCThreadState::transact(int32_t handle,
                                   uint32_t code, const Parcel& data,
                                   Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    .....

    /*
    注意这里的第一个参数 BC_TRANSACTION，它是应用程序向 binder 设备发送消息的消息码，
    而 binder 设备向应用程序回复消息的消息码以 BR_ 开头。消息码的定义在 binder_module.h 中，
    请求消息码和回应消息码的对应关系，需要查看 Binder 驱动的实现才能将其理清楚，我们这里暂时用不上。
    */
    err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    .....
    err = waitForResponse(reply);
    .....

    return err;
}
```

---

多熟悉的流程：先发数据，然后等结果。再简单不过了！不过，我们有必要确认一下 handle 这个参数到底起了什么作用。先来看 writeTransactionData 函数，它的实现如下所示：

👉 [-->IPCThreadState.cpp]

---

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
                                               int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    //binder_transaction_data 是和 binder 设备通信的数据结构。
    binder_transaction_data tr;

    // 果然，handle 的值传递给了 target，用来标识目的端，其中 0 是 ServiceManager 的标志。
    tr.target.handle = handle;
    //code 是消息码，是用来 switch/case 的！
    tr.code = code;
    tr.flags = binderFlags;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    }
```

---



```

    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = statusBuffer;
        tr.offsets_size = 0;
        tr.data.ptr.offsets = NULL;
    } else {
        return (mLastError = err);
    }
    // 把命令写到 mOut 中，而不是直接发出去，可见这个函数有点名不副实。
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));
    return NO_ERROR;
}

```

现在，已经把 addService 的请求信息写到 mOut 中了。接下来再看发送请求和接收回复部分的实现，代码在 waitForResponse 函数中，如下所示：

👉 [-->IPCThreadState.cpp]

```

status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
    int32_t cmd;
    int32_t err;

    while (1) {
        // 好家伙，talkWithDriver！
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = mIn.readInt32();
        switch (cmd) {
            case BR_TRANSACTION_COMPLETE:
                if (!reply && !acquireResult) goto finish;
                break;

            .....
            default:
                err = executeCommand(cmd); // 看这个！
                if (err != NO_ERROR) goto finish;
                break;
        }
    }

    finish:
        if (err != NO_ERROR) {
            if (acquireResult) *acquireResult = err;

```

```

        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}

```

OK, 我们已发送了请求数据, 假设马上就收到了回复, 后续该怎么处理呢? 来看 executeCommand 函数, 如下所示:

👉 [-->IPCThreadState.cpp]

```

status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;
        .....
    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            if (result != NO_ERROR) break;
            Parcel buffer;
            Parcel reply;
            if (tr.target.ptr) {
                /*
                 * 看到 BBinder 想起图 6-3 了吗? BnServiceXXX 从 BBinder 派生,
                 * 这里的 b 实际上就是实现 BnServiceXXX 的那个对象, 关于它的作用, 我们要在 6.5 节中讲解。
                 */
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, 0);
                if (error < NO_ERROR) reply.setError(error);
            } else {
                /*
                 * the_context_object 是 IPCThreadState.cpp 中定义的一个全局变量,
                 * 可通过 setTheContextObject 函数设置。
                 */
                const status_t error =
                    the_context_object->transact(tr.code, buffer, &reply, 0);
                if (error < NO_ERROR) reply.setError(error);
            }
            break;
            .....
        }
    }
}

```

```

case BR_DEAD_BINDER:
{
    /*
        收到 Binder 驱动发来的 service 死掉的消息，看来只有 Bp 端能收到了，
        后面我们将会对此进行分析。
    */
    BpBinder *proxy = (BpBinder*)mIn.readInt32();
    proxy->sendObituary();
    mOut.writeInt32(BR_DEAD_BINDER_DONE);
    mOut.writeInt32((int32_t)proxy);
} break;
.....
case BR_SPAWN_LOOPER:
// 特别注意，这里将收到来自驱动的指示以创建一个新线程，用于和 Binder 通信。
mProcess->spawnPooledThread(false);
break;
default:
    result = UNKNOWN_ERROR;
    break;
}
.....
if (result != NO_ERROR) {
    mLastError = result;
}
return result;
}

```

### (3) 打破砂锅问到底

你一定想知道如何和 binder 设备交互吧？是通过 write 和 read 函数来发送和接收请求的吗？来看 talkWithDriver 函数，如下所示：

👉 [-->IPCThreadState.cpp]

```

status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    // binder_write_read 是用来与 binder 设备交换数据的结构。
    binder_write_read bwr;
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    // 请求命令的填充。
    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();

    if (doReceive && needRead) {
        // 接收数据缓冲区信息的填充。如果以后收到数据，就直接填在 mIn 中了。
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    } else {

```

```

        bwr.read_size = 0;
    }

    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        #if defined(HAVE_ANDROID_OS)
            // 看来不是 read/write 调用, 而是 ioctl 方式。
            if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
                err = NO_ERROR;
            else
                err = -errno;
        #else
            err = INVALID_OPERATION;
        #endif
    } while (err == -EINTR);

    if (err >= NO_ERROR) {
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        return NO_ERROR;
    }
    return err;
}

```

较为深入地分析了 MediaPlayerService 的注册过程后, 下面还剩最后两个函数, 就让我们向它们发起进攻吧!

### 6.2.5 秋风扫落叶——StartThread Pool 和 join Thread Pool 分析

重要的内容都已讲过了, 现在就剩下最后两个函数 startThreadPool() 和 joinThreadPool 没有分析了。它们太简单了, 不是吗?

#### 1. 创造劳动力——startThreadPool()

startThreadPool() 的实现, 如下面的代码所示:

👉 [-->ProcessState.cpp]

// 太简单, 没什么好说的。

```

void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    // 如果已经 startThreadPool 的话，这个函数就没有什么实质作用了。
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true); // 注意，传进去的参数是 true。
    }
}

```

---

上面的 spawnPooledThread() 函数的实现如下所示：

 [-->ProcessState.cpp]

---

```

void ProcessState::spawnPooledThread(bool isMain)
{
    // 注意，isMain 参数是 true。
    if (mThreadPoolStarted) {
        int32_t s = android_atomic_add(1, &mThreadPoolSeq);
        char buf[32];
        sprintf(buf, "Binder Thread #%d", s);
        sp<Thread> t = new PoolThread(isMain);
        t->run(buf);
    }
}

```

---

PoolThread 是在 IPCThreadState 中定义的一个 Thread 子类，它的实现如下所示：

 [-->IPCThreadState.h::PoolThread 类]

---

```

class PoolThread : public Thread
{
public:
    PoolThread(bool isMain)
        : mIsMain(isMain){}
protected:
    virtual bool threadLoop()
    {
        // 线程函数如此简单，不过是在这个新线程中又创建了一个 IPCThreadState。
        // 你还记得它是每个伙计都有一个的吗？
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }
    const bool mIsMain;
};

```

---

## 2. 万众归一——joinThreadPool

还需要看看 IPCThreadState 的 joinThreadPool 的实现，因为新创建的线程也会调用这个函数，具体代码如下所示：

👉 [-->IPCThreadState.cpp]

```
void IPCThreadState::joinThreadPool(bool isMain)
{
    // 注意, 如果 isMain 为 true, 我们则需要循环处理。把请求信息写到 mOut 中, 待会儿一起发出去。
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

    androidSetThreadSchedulingGroup(mMyThreadId, ANDROID_TGROUP_DEFAULT);

    status_t result;
    do {
        int32_t cmd;

        if (mIn.dataPosition() >= mIn.dataSize()) {
            size_t numPending = mPendingWeakDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                    refs->decWeak(mProcess.get());
                }
                mPendingWeakDerefs.clear();
            }
            // 处理已经死亡的 BBinder 对象。
            numPending = mPendingStrongDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    BBinder* obj = mPendingStrongDerefs[i];
                    obj->decStrong(mProcess.get());
                }
                mPendingStrongDerefs.clear();
            }
        }
        // 发送命令, 读取请求。
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
            result = executeCommand(cmd); // 处理消息
        }

        .....
    } while (result != -ECONNREFUSED && result != -EBADF);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

原来, 我们的两个伙计在 talkWithDriver, 它们希望能从 binder 设备那里找到点可做的事情。

### 3. 有几个线程在服务

到底有多少个线程在为 Service 服务呢？目前看来是两个：

- ❑ startThreadPool 中新启动的线程通过 joinThreadPool 读取 binder 设备，查看是否有请求。
- ❑ 主线程也调用 joinThreadPool 读取 binder 设备，查看是否有请求。看来，binder 设备是支持多线程操作的，其中一定是做了同步方面的工作。

MediaServer 这个进程一共注册了 4 个服务，繁忙的时候，两个线程会不会显得有点少呢？另外，如果实现的服务负担不是很重，完全可以不调用 startThreadPool 创建新的线程，使用主线程即可胜任。

## 6.2.6 你彻底明白了吗

我们以 MediaServer 为例，分析了 Binder 的机制，这里还是有必要再次强调一下 Binder 通信和基于 Binder 通信的业务之间的关系。

- ❑ Binder 是通信机制。
- ❑ 业务可以基于 Binder 通信，当然也可以使用别的 IPC 方式通信。

Binder 之所以复杂，重要原因之一在于 Android 通过层层封装，巧妙地把通信和业务融合在了一起。如果透彻地理解了这一点，Binder 对我们来说就较为简单了。它们之间的交互关系可通过图 6-4 来表示：

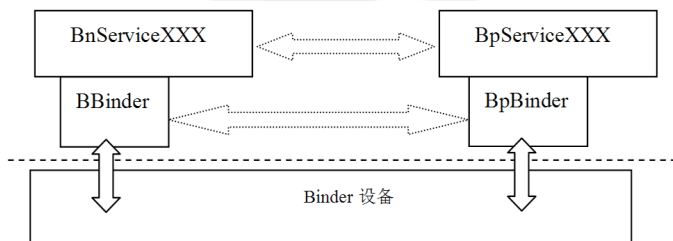


图 6-4 binder 通信层和业务层的关系图

## 6.3 服务总管 ServiceManager

### 6.3.1 ServiceManager 的原理

前面说过，defaultServiceManager 返回的是一个 BpServiceManager，通过它可以把命令请求发送给 handle 值为 0 的目的端。按照图 6-3 所示的 IServiceManager “家谱” 来看，无论如何也应该有一个类从 BnServiceManager 派生出来并处理这些来自远方的请求吧？

很可惜，源码中竟然没有这样一个类存在！但确实又有这么一个程序完成了 BnServiceManager 未尽的工作，这个程序就是 ServiceManager，它的代码在 Service\_manager.c 中。