HZ BOOKS
华章科技

结合实际应用开发需求，以情景分析的方式有针对性地对Android的源代码进行了详尽的剖析，深刻揭示Android系统的工作原理

机锋网、51CTO、开源中国社区等专业技术网站一致鼎力推荐

移动开发



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

联袂推荐

作者是Thundersoft多媒体组的牛人，技术精深，乐于分享，对Android系统有真正的理解。本书内容给力，语言生动，全书几乎没有废话，各章中的“拓展思考”尤为精彩，体现了作者对Android实现原理的深入理解和批判性思考。为什么Android的短信群发很慢？为什么拔出SD卡时有的程序会退出？读者都能从本书中找到诸如此类的各种实际问题的答案。更重要的是，读者能够对Android的整个体系有一个全新的理解。如果你通读了这本书，请一定投一份简历给我们。

——**Thundersoft**（中科创达软件科技（北京）有限公司）

对于Android开发工程师而言，本书不可多得，分析透彻深入，针对性极强。Android系统本身极为庞大，如果要对整个系统进行面面俱到且细致入微地分析，恐怕不是一两本书能完成的。本书从开发者的实际需求出发，有针对性地对Android系统中的重要知识点和功能模块的源代码实现进行了剖析，这样既能帮助开发者解决实际问题，又能使分析深入透彻，而不是停留于表面。强烈推荐！

——**机锋网**（http://www.gfan.com/）

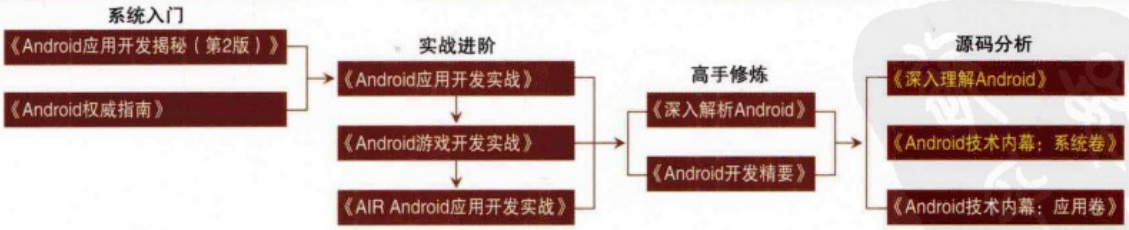
这本书非常实用，绝不是枯燥的源代码分析，是深入理解Android工作机制和实现原理的一本好书。为什么说它实用呢？因为它的最终目的并不是停留在源代码分析上，而是要帮助开发者解决实际问题，所以所有知识点的分析和讲解都是从开发者的实际需求出发的。与一般的源代码分析的书相比较而言，本书在语言上多了几分幽默，更加生动易懂。更重要的是，本书的分析十分深入，探讨了Android相关功能模块的本质。

——51CTO移动开发频道（http://mobile.51cto.com/）

随着Android 系统越来越流行，Android应用的需求也在不断变化，对于开发者而言，深入理解Android系统原理显得越来越重要。目前市面上Android 开发相关的图书已经很多，但真正能够系统、深入地讲解Android系统原理的书还乏善可陈。这本书的出版恰逢其时，该书深度和广度兼备，以循序渐进的方式，优雅的语言，深入分析到了各模块的源码与原理。另外，它启发性的讲解方式，更有助于读者的学习和思考。

——**开源中国社区**（http://www.oschina.net/）

Android开发者学习路线图



说明：黄色表示已出版

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

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

上架指导：程序设计 / 移动开发
ISBN 978-7-111-35762-9
9 787111 357629
定价：69.00元

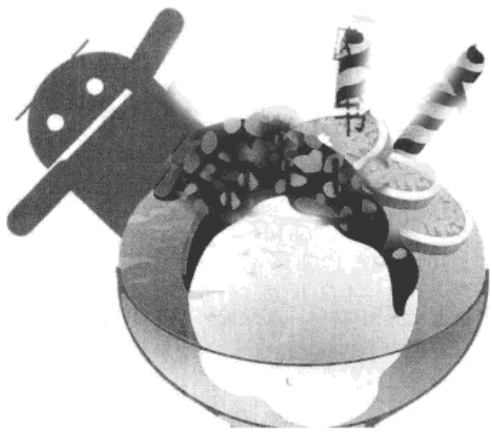
移动开发

Understanding Android Internals: Volume I

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
北京市荣盛彩色印刷有限公司印刷
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



推 荐 序

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

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



前 言

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[①]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 Andorid 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

① 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：

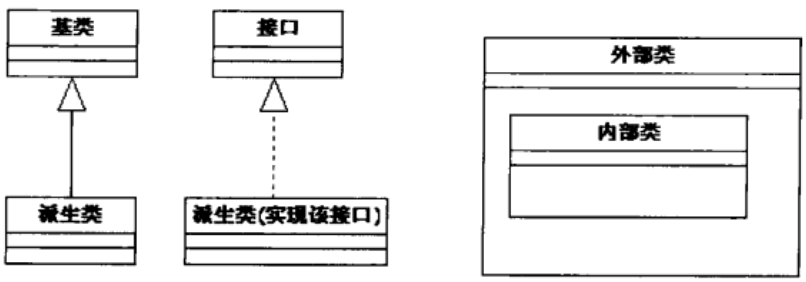


图 1　UML 示例图

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。



致　谢

首先要感谢杨福川编辑。本书最初的内容来自我的博客[①]，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宇的帮助下，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最无私的帮助，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[②]。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。

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



目　录

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 有人情味的讣告 / 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1章

阅读前的准备工作

1.1 系统架构

1.1.1 Android 系统架构

Android 是 Google 公司推出的一款智能手机平台。该平台本身是基于 Linux 内核的，图 1-1 展示了这个系统的架构：

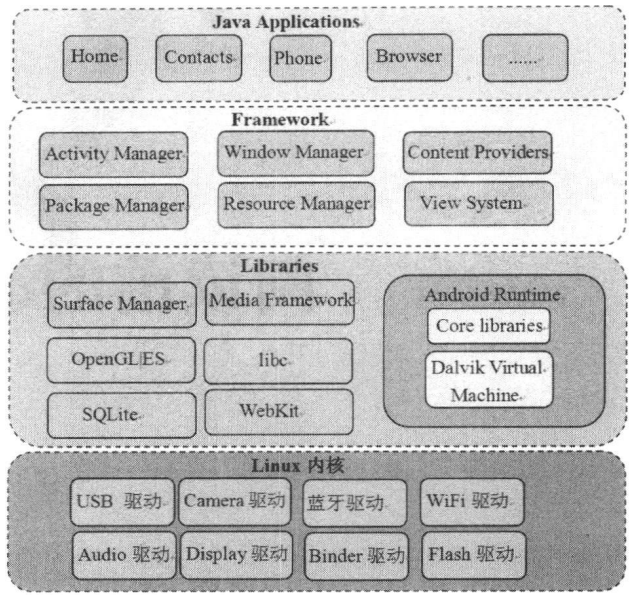


图 1-1 Android 系统架构

从上图中可以看出，Android 系统大体可分为四层，从下往上依次是：

- Linux 内核层：包含了 Linux 内核和一些驱动模块（比如 USB 驱动、Camera 驱动、蓝牙驱动等）。目前 Android2.2（代号为 Froyo）基于 Linux 内核 2.6 版本。
- Libraries 层：这一层提供动态库（也叫共享库）、Android 运行时库、Dalvik 虚拟机等。从编程语言角度来说，这一层大部分都是用 C 或 C++ 写的，所以也可以简单地把它看成是 Native 层。
- Framework 层：这一层大部分用 Java 语言编写，它是 Android 平台上 Java 世界的基石。
- Applications 层：与用户直接交互的就是这些应用程序，它们都是用 Java 开发的。

从上面的介绍可看出，Android 系统的最大特点之一就是搭建了一个被广大 Java 开发者热捧的 Java 世界。但这个世界并不是空中楼阁，它的运转依赖于另一个被 Google 极力隐藏的 Native 世界。两个世界的交互关系可用图 1-2 来表示：

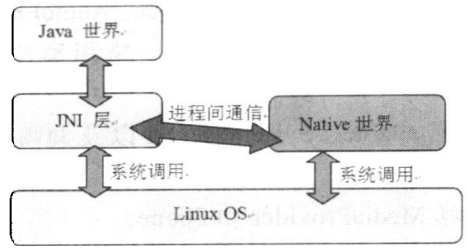


图 1-2　Java 世界和 Native 世界交互

从上图可知：

- Java 虽具有与平台无关的特性，但 Java 和具体平台之间的隔离却是由 JNI 层来实现的。Java 是通过 JNI 层调用 Linux OS 中的系统调用来完成对应的功能的，例如创建一个文件或一个 Socket 等。
- 除了 Java 世界外，还有一个核心的 Native 世界，它为整个系统高效和平稳地运行提供了强有力的支持。一般而言，Java 世界经由 JNI 层通过 IPC 方式与 Native 世界交互，而 Android 平台上最为神秘的 IPC 方法就是 Binder 了，第 6 章将详细分析 Binder。除此之外，Socket 也是常用的 IPC 方式。这些内容在后面的代码分析中都会见到。

1.1.2　本书的架构

本书所分析的模块也将遵循 Android 系统架构，如图 1-3 所示：

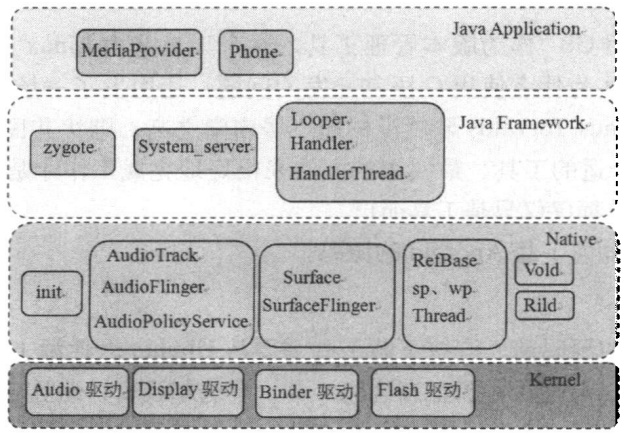


图 1-3　本书的架构图

从上图可知，本书所分析的各个模块除未涉及 Kernel 外，其他三层均有所涉及，它们分别是：

- Native 层包括 init、Audio 系统（包括 AudioTrack、AudioFlinger 和 AudioPolicyService）、Surface 系统（包括 Surface 和 SurfaceFlinger）、常用类（包括 RefBase、sp、wp 等）、Vold 和 Rild。
- Java Framework 层包括 zygote、System_server 以及 Java 中的常用类（包括 Handler 和 Looper 等）。
- Java Application 层包括 MediaProvider 和 Phone。

1.2 搭建开发环境

本节将介绍如何搭建 Android 源码开发环境。首先，需要一个 Linux 系统，笔者推荐安装 Ubuntu 10.04（32 位版本），读者可从网上下载该版本的系统。Windows 用户可以使用 VMWare 或 VirtualBox 作为虚拟机来安装 Ubuntu10.04。笔者推荐 VMWare，因为它的功能太强大了！

注意 如果要使用 VMWare，那么在安装完 Ubuntu 之后，一定要把 VMWare Tools 也安装上，因为这个工具会提供很多非常实用的功能。这里还有一个小建议，如果 Linux 系统只是个人使用，则建议用 root 账户登录系统。在工作中，曾发现很多用非 root 账户登录的同事整天都在执行 sudo 命令和输入密码，这样浪费了不少零碎的时间片。

假设读者已经安装好了 Ubuntu 10.04（32 位版本），并且以 root 账户登录到系统上了，接下来就要开始下面的工作。

1.2.1 下载源码

Android 源码采用 Git[①]作为版本管理工具，这个工具是由 Linux 之父 Linus Torvalds 采用纯 C 开发。关于 Git 为什么使用 C 语言开发的问题，还引发了一场关于 C 和 C++ 孰好孰坏的大讨论，不过 Linus Torvalds 显然没树起“居庙堂之高，则忧其民”的形象。对于普通“码农”而言，用最合适的工具、最实用的办法来很好地完成工作才是最重要的。所以，C、C++、Java、Python 等都仅仅只是工具而已。

下面将详细介绍如何下载 Android 的源码。

1. 设置软件源

在下载 Android 的源码前，有些下载工具需要从 Ubuntu 软件源上下载。可以为 Ubuntu 系统指定一个软件源。有些软件源上有这些工具，有些却没有，而且各个软件源的下载速度也不同，所以应首先找到一个合适的软件源。Ubuntu 软件源的设置界面如图 1-4 所示：

---

① 如果你对Git不熟悉或对它感兴趣，推荐阅读《Git权威指南》（机械工业出版社，2011.7月出版，蒋鑫著），这是目前最全面、最深入的一本Git专著。

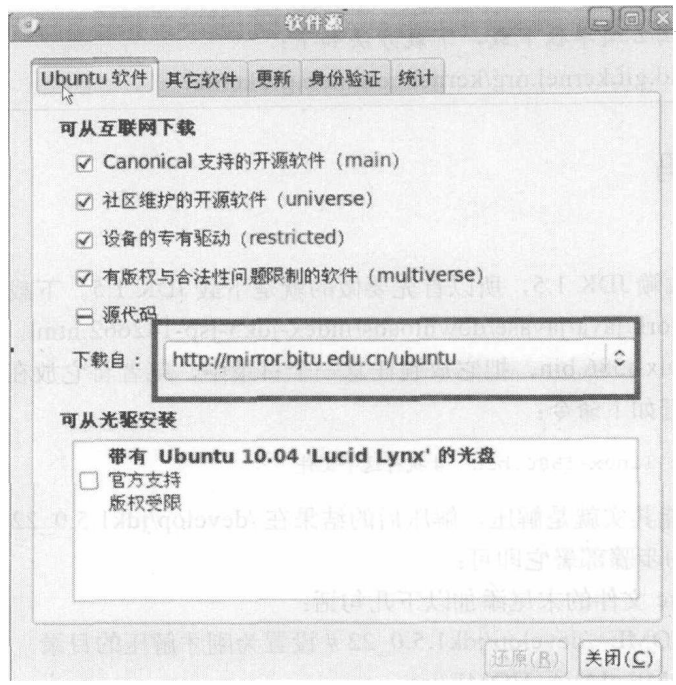


图 1-4　Ubuntu 软件源设置

从上图中可发现，将软件源地址设置成了 http://mirror.bjtu.edu.cn/ubuntu。每个人可根据自己的情况选择合适的软件源。

2. 下载 Android 源码

下面开始下载 Android 源码，工序比较简单，可一气呵成。

- apt-get install git-core curl　　# 先下载这两个工具
- mkdir –p /develop/download-froyo　# 在根目录下建立 develop 和 download-froyo 两个目录
- cd ~/develop/download-froyo　　# 进入这个目录
- curl http://Android.git.kernel.org/repo > ./repo　# 从源码网站下载 repo 脚本，该脚本是 Google 为了方便源码下载而提供的，通过该脚本可下载整套源码。
- chmod a+x repo　　# 设置该脚本为可执行
- ./repo init -u git://Android.git.kernel.org/platform/manifest.git –b froyo　　# 初始化 git 库
- ./repo sync　　# 下载源码，大小约为 2GB，如果网速快，估计也得要 2 个多小时。

下载完后，该目录中的内容如图 1-5 所示：

```
root@ubuntu:/develop/download-froyo# ls
bionic    cts          device      hardware  ndk       prebuilt  system
bootable  dalvik       external    kernel    out       repo
build     development  frameworks  Makefile  packages  sdk
```

图 1-5　源码下载结果

注意 Kernel 的代码必须单独下载，下载方法如下：
git clone git://android.git.kernel.org/kernel/common.git kernel

1.2.2 编译源码

1. 部署 JDK

Froyo 的编译依赖 JDK 1.5，所以首先要做的就是下载 JDK 1.5。下载网址是 http://www.oracle.com/technetwork/java/javase/downloads/index-jdk5-jsp-142662.html。下载得到的文件为 jdk-1_5_0_22-linux-i586.bin，把它放到任意一个目录中，笔者将它放在了 /develop 中，然后在这个目录中执行如下命令：

```
./jdk-1_5_0_22-linux-i586.bin  # 执行这个文件
```

这个命令的功能其实就是解压，解压后的结果在 /develop/jdk1.5.0_22 目录中。现在有了 JDK，再按照下面的步骤部署它即可：

（1）在 ~/.bashrc 文件的末尾添加以下几句话：

```
export JAVA_HOME=/develop/jdk1.5.0_22 # 设置为刚才解压的目录
export JRE_HOME=JAVA_HOME/jre
export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

（2）重新登录系统，这样 JDK 资源就能被正确找到了。

2. 编译源码

Android 的编译有自己的一套规则，主要利用的是 mk 文件。网上有太多关于它的解说了，这里不再赘述，只简单地介绍其编译工序：

进入源码目录（以笔者的开发环境为例，也就是 cd /develop/download_froyo）：

- 执行 . build/envsetup.sh，这个脚本用来设置 Android 的编译环境。
- 执行 choosecombo 命令，这个命令用来选择编译目标（如目标硬件平台、eng 还是 user 等）。一般而言，手机厂商会设置自己特有的编译选项。

执行完上面几个步骤后，就可以编译系统了。Android 平台提供了三个命令用于编译，它们分别是 make、mmm 和 mm，这三个命令的使用方法及其优劣如下：

- make：不带任何参数，它用于编译整个系统，时间较长，笔者不推荐这种做法，除非读者想编译整个系统。
- make MediaProvider：下面几个例子都以编译 MediaProvider 为例。这种方式对应于单个模块编译。它的优点是，会把该模块依赖的其他模块也一起编译。例如 make libmedia，就会把 libmedia 依赖的库全部编译好。其缺点也很明显，它需要搜索整个源码来定位 MediaProvider 模块所使用的 Android.mk 文件，并且还要判断该模块所依

赖的其他模块是否有修改。整体编译时间较长。

- mmm packages/providers/MediaProvider：该命令将编译指定目录下的目标模块，而不编译它所依赖的模块。所以，如果读者是初次编译，采用这种方式编译一个模块往往会报错。错误的原因是因为它依赖的模块没有被编译。
- mm：这种方式需要先用 cd 命令进入 packages/providers/MediaProvider 目录，然后执行 mm 命令。该命令会编译当前目录下的模块。它和 mmm 一样，只编译目标模块，mm 和 mmm 命令编译的速度都很快。

从使用的角度来看，笔者有如下建议：

- 如果只知道目标模块的名称，则应使用 make 模块名的方式来编译目标模块。例如，如果要编译 libmedia，则直接使用 make libmedia 即可。另外，初次编译时也要采用这种方法。
- 如果不知道目标模块的名称，但知道目标模块所处的目录，则可使用 mmm 或 mm 命令来编译。当然，初次编译还必须使用 make 命令，以后的编译就可使用 mmm 或 mm 了，这样会节约不少时间。

注意　一般的编译方式都使用增量编译，即只编译发生变化的目标文件，但有时则需重新编译所有目标文件，那么就可使用 make 命令的 -B 选项。例如 make -B 模块名，或者 mm -B、mmm -B。在 mm 和 mmm 内部，也是调用 make 命令的，而 make 的 -B 选项将强制编译所有目标文件。

Android 的编译工序比较简单，难点主要在 Android.mk 文件的编写，读者可上网搜索与此相关的学习资料。

3. 本书各模块的编译目标

本书各模块的编译目标如表 1-1 所示，这里仅列出几个有代表性的模块：

表 1-1　本书各模块的编译目标

目标模块	make 命令	mmm 命令
init	make init	mmm system/core/init
zygote	make app_process	mmm frameworks/base/cmds/app_process
system_server	make services	mmm frameworks/base/services/java
RefBase 等	make libutils	mmm frameworks/base/libs/utils
Looper 等	make framework	mmm frameworks/base
AudioTrack	make libmedia	mmm frameworks/base/media/libmedia
AudioFlinger	make libaudioflinger	mmm frameworks/base/libs/audioflinger
AudioPolicyService	make libaudiopolicy	mmm hardware/msm7k/libaudio-qsd8k（示例）

SurfaceFlinger	make libsurfaceflinger	mmm frameworks/base/libs/surfaceflinger
Vold	make vold	mmm system/vold/
Rild	make rild	mmm hardware/ril/rild/
MediaProvider	make MediaProvider	mmm packages/providers/MediaProvider
Phone	make Phone	mmm packages/apps/Phone/

假设 make framework，那么编译完的结果则如图 1-6 所示：

```
bj/NOTICE_FILES/src//system/framework/framework-res.apk.txt
Install: out/target/product/generic/system/framework/framework-res.apk
Install: out/target/product/generic/system/framework/framework.jar
```

图 1-6 make framework 的结果

从上图可看出，make 命令编译了 framework-res.apk 和 framework.jar 两个模块。它们编译的结果在 out/target/product/generic/system/framework 下。利用 adb 命令把这两个文件 push 到手机的 system/framework 目录即可替换旧的文件。如果想测试这个新模块，则需要先杀掉所有使用该模块的进程，进程重启后会重新加载模块，这时就能使用新的文件了。例如，想测试刚修改的 libaudioflinger 模块，通过 adb 命令 push 上去后，要先杀掉 mediaserver 进程，因为 libaudioflinger 库目前只有该进程在使用。当 mediaserver 重启后，就会加载新 push 上来的 libaudioflinger 库了。

注意 *系统服务被杀掉后一般都会自动重启（由 init 控制，见第 3 章中的分析）。*

1.3 工具介绍

本节将介绍 Android 应用开发和源码研究过程中会经常用到的实用工具。

1.3.1 Source Insight 介绍

Source Insight 是阅读源码的必备工具，它是 Windows 下的一个软件，在 Linux 平台上可通过 wine 安装。这里就不讲述如何安装 Source Insight 了，相信读者都会。下面介绍一下 Source Insight 的使用小技巧。

1. Source Insight 工作减负

使用 Source Insight 时，需要新建一个源码工程，通过菜单项 Project → New Project 可指定源码的目录。在工作中发现，很多同事常一股脑把所有 Android 源代码都加到工程中，从而导致 Source Insight 运行速度非常慢。实际上，只需要将当前分析的源码目录加到工程中即可。例如，新建一个 Source Insight 工程后，只把源码 /framework/base 目录加进去。另外，当一个目录下的源码分析完后，可以通过 Project → Add and Remove Project Files 选项

把无须再分析的目录从工程中去掉。如图 1-7 所示：

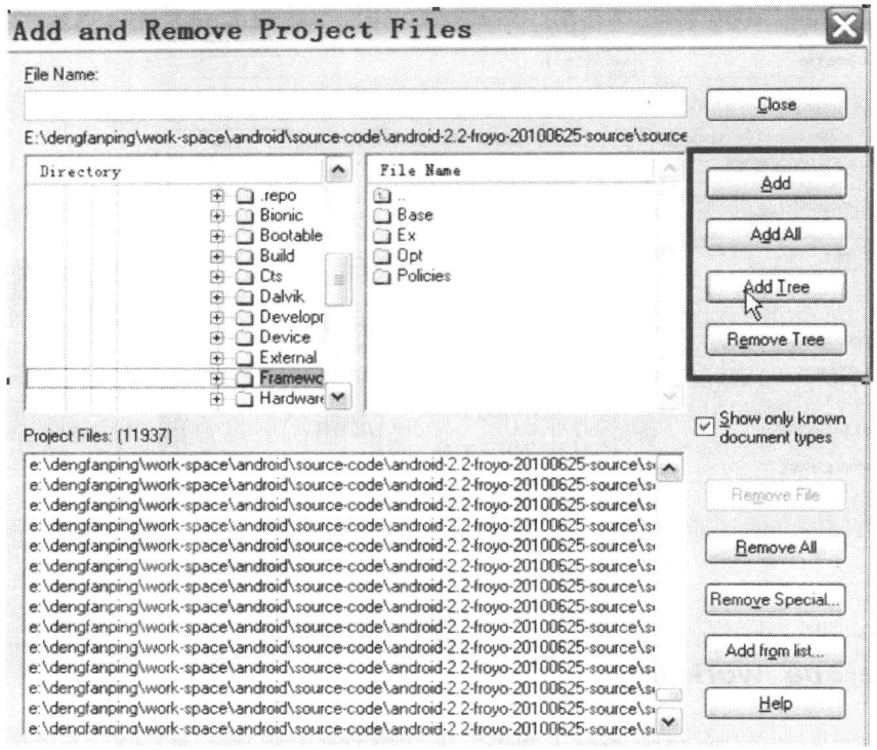


图 1-7　添加或删除工程中的目录

从图中的框线我们可以发现：Source Insight 支持动态添加或删除目录。通过这种方式可极大减少 Source Insight 的工作负担。

注意　一般是首先把 framework/base 下的目录加到工程中，以后如果有需要，再把其他目录加进来。

---

2. 调节字体

Source Insight 默认的字体比较小，看着很费眼。怎么办？

依次选择工具栏上的 Options → Document options 菜单项，会弹出 Document Options 对话框，其中左上部分有个 Screen Fonts 按钮，单击后会弹出一个字体对话框，在那里可选择大字体。如图 1-8 所示。

3. 快速定位文件

工程建立好后，需通过 Project → Rebuild Project 选项来解析源码。另外，在研究源码时常常会只记得源码文件名，而不记得是在哪个目录下。没关系，Source Insight 支持在源码中快速定位文件。使用方法如图 1-9 所示。

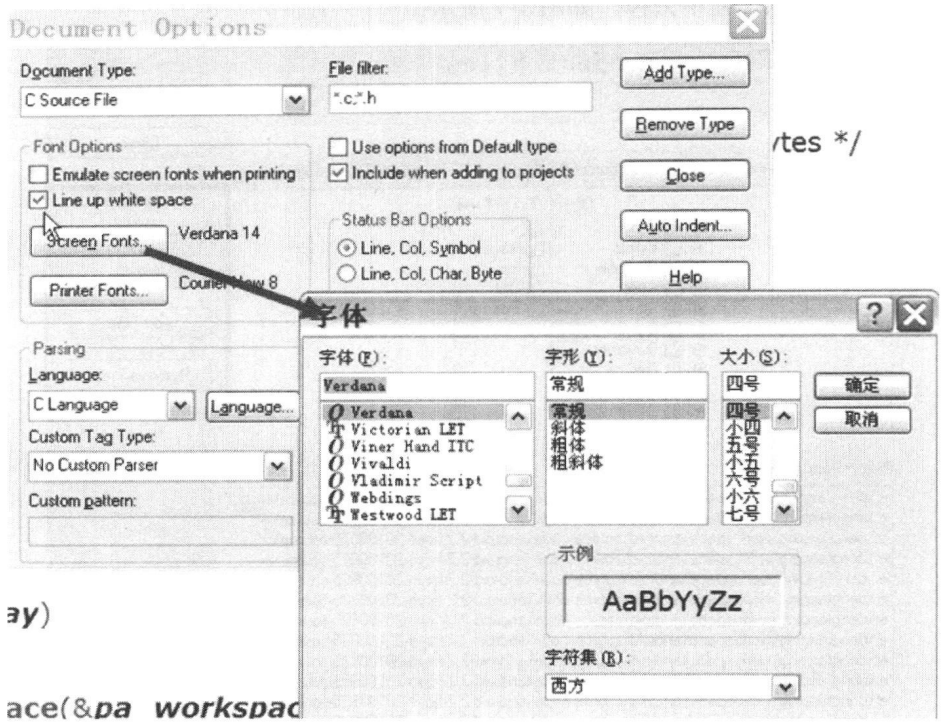


图 1-8　字体调节

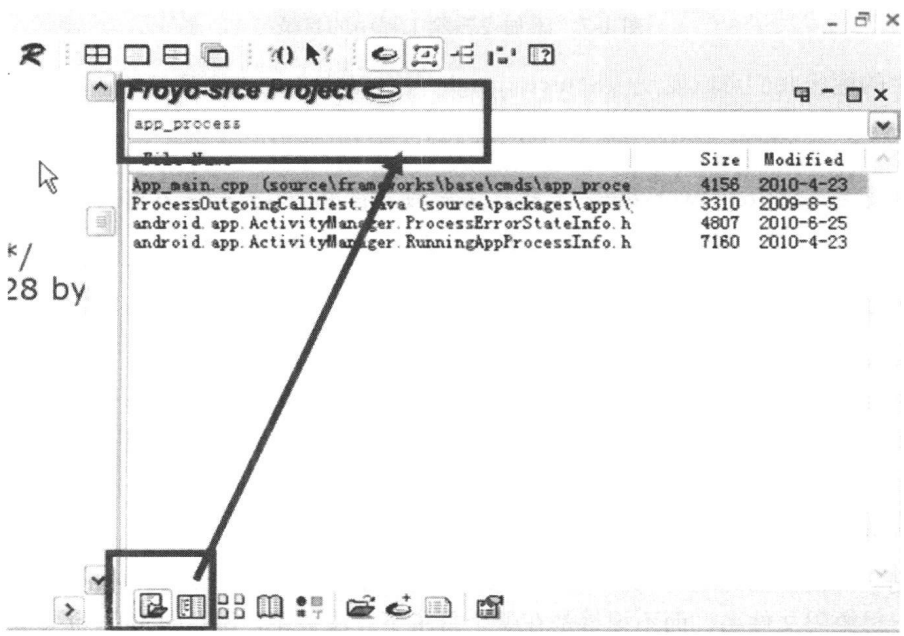


图 1-9　快速定位文件

使用方法是：

（1）首先选择图 1-8 中左下角的那个按钮。

（2）然后在左上角那个输入框中输入源码文件名，例如 app_process，最后结果栏中就会把对应的文件列出。

1.3.3　Busybox 的使用

Busybox 号称 Linux 平台上的“瑞士军刀”，它提供了很多常用的工具，例如 grep 和 find 等。这些工具在标准 Linux 上都有，但 Android 系统却去掉了其中的大多数工具。这导致我们在调试程序和研究 Android 系统时十分不便，所以我们需要在手机上安装 Busybox。

1. 下载 Busybox

我们可从下面这个网站中下载已编译好的 Busybox：

http://www.busybox.net/downloas/binaries/1.18.4/

图 1-10 显示了该网站已经根据不同平台编译好的 Busybox，我们可根据自己的手机下载对应的文件。例如，HTC G7 的 CPU 支持 armv7l，所以我下载了最接近的 busybox-armv6l。

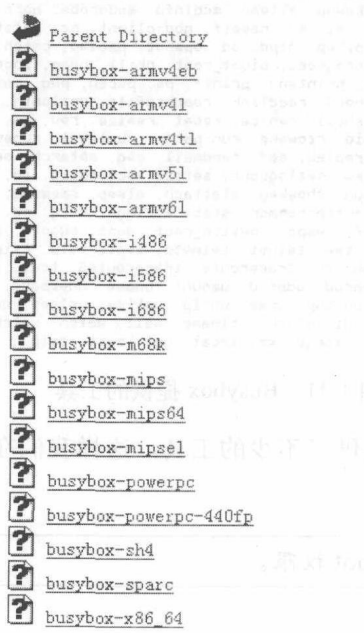


图 1-10　Busybox 下载

小知识　arm v7 表示的是 ARM 指令集为 v7，目前 ARM Cortex-A8/A9 系列的 CPU 支持该指令集。

2. 安装和使用 Busybox

下载完 Busybox 后，需将它 push 到手机上，具体操作如下：

```
adb push busybox /system/xbin # 为了避免冲突，我 push 到了 /system/xbin 目录下了。
cd /system/xbin        # 进入对应目录
chmod 755 busybox      # 更改 Busybox 权限为可执行
busybox --install      # 安装 Busybox
grep  # 执行 Busybox 提供的 grep 命令，或者 busybox xxx 执行 xxx 命令也行
```

Busybox 安装完成后，如果执行 busybox 命令，就会打印如图 1-11 的输出。

```
Currently defined functions:
        [, [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash,
        awk, base64, basename, beep, blkid, blockdev, bootchartd, brctl,
        bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod,
        chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm,
        cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd,
        deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff,
        dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap,
        dumpleases, echo, ed, egrep, eject, env, envdir, envuidgid, ether-wake,
        expand, expr, fakeidentd, false, fbset, fbsplash, fdflush, fdformat,
        fdisk, fgconsole, fgrep, find, findfs, flock, fold, free, freeramdisk,
        fsck, fsck.minix, fsync, ftpd, ftpget, ftpput, fuser, getopt, getty,
        grep, gunzip, gzip, halt, hd, hdparm, head, hexdump, hostid, hostname,
        httpd, hush, hwclock, id, ifconfig, ifdown, ifenslave, ifplugd, ifup,
        inetd, init, insmod, install, ionice, iostat, ip, ipaddr, ipcalc,
        ipcrm, ipcs, iplink, iproute, iprule, iptunnel, kbd_mode, kill,
        killall, killall5, klogd, last, length, less, linux32, linux64,
        linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread,
        losetup, lpd, lpq, lpr, ls, lsattr, lsmod, lspci, lsusb, lzcat, lzma,
        lzop, lzopcat, makedevs, makemime, man, md5sum, mdev, mesg, microcom,
        mkdir, mkdosfs, mke2fs, mkfifo, mkfs.ext2, mkfs.minix, mkfs.vfat,
        mknod, mkpasswd, mkswap, mktemp, modinfo, modprobe, more, mount,
        mountpoint, mpstat, mt, mv, nameif, nbd-client, nc, netstat, nice,
        nmeter, nohup, nslookup, ntpd, od, openvt, passwd, patch, pgrep, pidof,
        ping, ping6, pipe_progress, pivot_root, pkill, pmap, popmaildir,
        poweroff, powertop, printenv, printf, ps, pscan, pwd, raidautorun,
        rdate, rdev, readahead, readlink, readprofile, realpath, reboot,
        reformime, remove-shell, renice, reset, resize, rev, rm, rmdir, rmmod,
        route, rpm, rpm2cpio, rtcwake, run-parts, runlevel, runsv, runsvdir,
        rx, script, scriptreplay, sed, sendmail, seq, setarch, setconsole,
        setfont, setkeycodes, setlogcons, setsid, setuidgid, sh, sha1sum,
        sha256sum, sha512sum, showkey, slattach, sleep, smemcap, softlimit,
        sort, split, start-stop-daemon, stat, strings, stty, su, sulogin, sum,
        sv, svlogd, swapoff, swapon, switch_root, sync, sysctl, syslogd, tac,
        tail, tar, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time,
        timeout, top, touch, tr, traceroute, traceroute6, true, tty, ttysize,
        tunctl, udhcpc, udhcpd, udpsvd, umount, uname, unexpand, uniq,
        unix2dos, unlzma, unlzop, unxz, unzip, uptime, usleep, uudecode,
        uuencode, vconfig, vi, vlock, volname, wall, watch, watchdog, wc, wget,
        which, who, whoami, xargs, xz, xzcat, yes, zcat, zcip
```

图 1-11　Busybox 提供的工具

从上图中可看出，Busybox 提供了不少的工具，这样我们在研究 Android 系统时就如虎添翼了。

注意　给手机安装 Busybox 须有 root 权限。

1.4　本章小结

本章对 Android 系统架构、源码开发环境搭建、源码研究工具等做了部分介绍，相信大家现在已是迫不及待，跃跃欲试了吧？马上开始我们的源码征程！



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。

- MediaScanner.java(*framework/base/media/java/src/android/media/MediaScanner.java*)
- android_media_MediaScanner.cpp(*framework/base/media/jni/MediaScanner.cpp*)
- android_media_MediaPlayer.cpp(*framework/base/media/jni/android_media_MediaPlayer.cpp*)
- AndroidRunTime.cpp(*framework/base/core/jni/AndroidRunTime.cpp*)
- JNIHelp.c(*dalvik/libnativehelper/JNIHelp.c*)

2.1 JNI 概述

JNI 是 Java Native Interface 的缩写，中文译为“Java 本地调用”。通俗地说，JNI 是一种技术，通过这种技术可以做到以下两点：

- Java 程序中的函数可以调用 Native 语言写的函数，Native 一般指的是 C/C++ 编写的函数。
- Native 程序中的函数可以调用 Java 层的函数，也就是说在 C/C++ 程序中可以调用 Java 的函数。

在平台无关的 Java 中，为什么要创建一个与 Native 相关的 JNI 技术呢？这岂不是破坏了 Java 的平台无关特性吗？ JNI 技术的推出有以下几个方面的考虑：

- 承载 Java 世界的虚拟机是用 Native 语言写的，而虚拟机又运行在具体的平台上，所以虚拟机本身无法做到平台无关。然而，有了 JNI 技术后就可以对 Java 层屏蔽不同操作系统平台（如 Windows 和 Linux）之间的差异了（例如同样是打开一个文件，Windows 上的 API 使用 OpenFile 函数，而 Linux 上的 API 是 open 函数）。这样，就能实现 Java 本身的平台无关特性。其实 Java 一直在使用 JNI 技术，只是我们平时较少用到罢了。
- 早在 Java 语言诞生前，很多程序都是用 Native 语言写的，它们遍布在软件世界的各个角落。Java 出世后，它受到了追捧，并迅速得到发展，但仍无法将软件世界彻底改朝换代，于是才有了折中的办法。既然已经有 Native 模块实现了相关功能，那么在 Java 中通过 JNI 技术直接使用它们就行了，免得落下重复制造轮子的坏名声。另外，在一些要求效率和速度的场合还是需要 Native 语言参与的。

在 Android 平台上，JNI 就是一座将 Native 世界和 Java 世界间的天堑变成通途的桥。来看图 2-1，它展示了 Android 平台上 JNI 所处的位置：

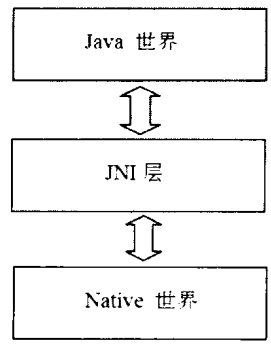


图 2-1 Android 平台中 JNI 的示意图

由上图可知，JNI 将 Java 世界和 Native 世界紧密地联系在了一起。在 Android 平台上尽情使用 Java 的程序员们不要忘了，如果没有 JNI 的支持，我们将寸步难行！

注意 虽然 JNI 层的代码是用 Native 语言写的，但本书还是把与 JNI 相关的模块单独归类到 JNI 层了。

俗话说，百闻不如一见，下面就来见识一下 JNI 技术吧。

2.2　学习 JNI 的实例：MediaScanner

初次接触 JNI，感觉最神奇的就是 Java 竟然能够调用 Native 的函数，可它是怎么做到的呢？由于 Android 大量使用了 JNI 技术，本章接下来的内容就将通过源码中的一处实例来讲解 JNI 相关的知识。

其中这个例子是和 MediaScanner 相关的，本书的最后一章会详细分析它的工作原理，这里先看与 JNI 相关的部分，如图 2-2 所示：

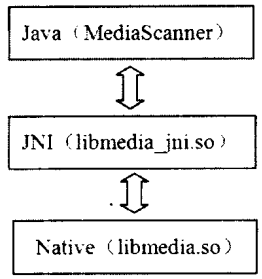


图 2-2　MediaScanner 和它的 JNI

将图 2-2 与图 2-1 结合来看，可以知道：

- Java 世界对应的是 MediaScanner，而这个 MediaScanner 类有一些函数需要由 Native 层来实现。
- JNI 层对应的是 libmedia_jni.so。media_jni 是 JNI 库的名字，其中，下划线前的“media”是 Native 层库的名字，这里就是 libmedia 库。下划线后的“jni”表示它是一个 JNI 库。注意，JNI 库的名字可以随便取，不过 Android 平台基本上都采用“lib 模块名 _jni.so”的命名方式。
- Native 层对应的是 libmedia.so，这个库完成了实际的功能。
- MediaScanner 将通过 JNI 库 libmedia_jni.so 和 Native 层的 libmedia.so 交互。

从上面的分析中还可知道：JNI 层必须实现为动态库的形式，这样 Java 虚拟机才能加载它并调用它的函数。

下面来看 MediaScanner。

提示　MediaScanner 是 Android 平台中多媒体系统的重要组成部分，它的功能是扫描媒体文件，得到诸如歌曲时长、歌曲作者等媒体信息，并将它们存入到媒体数据库中，供其他应用程序使用。

2.3 Java 层的 MediaScanner 分析

先来看 MediaScanner（简称 MS）的源码，这里将提取出与 JNI 有关的部分，其代码如下所示：

[-->MediaScanner.java]

```
public class MediaScanner
{
    static { static 语句
        /*
          ①加载对应的 JNI 库，media_jni 是 JNI 库的名字。在实际加载动态库的时候会将其拓展成
          libmedia_jni.so，在 Windows 平台上则拓展为 media_jni.dll
        */
        System.loadLibrary("media_jni");
        native_init();// 调用 native_init 函数
    }
    .......
    // 非 native 函数
    public void scanDirectories(String[] directories, String volumeName){
        ......
    }

    // ②声明一个 native 函数。native 为 Java 的关键字，表示它将由 JNI 层完成。
    private static native final void native_init();
    ......
    private native void processFile(String path, String mimeType,
                                    MediaScannerClient client);
    ......
}
```

上面代码中列出了两个比较重要的要点：一个是加载 JNI 库；另一个是 Java 的 native 函数。

2.3.1 加载 JNI 库

前面说过，如果 Java 要调用 native 函数，就必须通过一个位于 JNI 层的动态库来实现。顾名思义，动态库就是运行时加载的库，那么在什么时候以及什么地方加载这个库呢？

这个问题没有标准答案，原则上是：在调用 native 函数前，任何时候、任何地方加载都可以。通行的做法是在类的 static 语句中加载，调用 System.loadLibrary 方法就可以了。这一点在上面的代码中也见到了，我们以后就按这种方法编写代码即可。另外，System.loadLibrary 函数的参数是动态库的名字，即 media_jni。系统会自动根据不同的平台拓展成真实的动态库文件名，例如在 Linux 系统上会拓展成 libmedia_jni.so，而在 Windows 平台上则会拓展成 media_jni.dll。

解决了 JNI 库加载的问题，再来看第二个关键点。

2.3.2　Java 的 native 函数和总结

从上面代码中可以发现，native_init 和 processFile 函数前都有 Java 的关键字 native，它表示这两个函数将由 JNI 层来实现。

Java 层的分析到此结束。JNI 技术也很照顾 Java 程序员，只要完成下面两项工作就可以使用 JNI 了：

- 加载对应的 JNI 库。
- 声明由关键字 native 修饰的函数。

所以，对于 Java 程序员来说，使用 JNI 技术真的是太容易了。不过 JNI 层要完成任务可没这么轻松，下面来看对 JNI 层的 MediaScanner 的分析。

2.4　JNI 层 MediaScanner 的分析

MediaScanner（简称“MS”）的 JNI 层代码在 android_media_MediaScanner.cpp 中，如下所示：

[-->android_media_MediaScanner.cpp]

```
// ①这个函数是 native_init 的 JNI 层实现。
static void  android_media_MediaScanner_native_init(JNIEnv *env)
{
     jclass clazz;

    clazz = env->FindClass("android/media/MediaScanner");
    ......
    fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
    ......
    return;
}

// 这个函数是 processFile 的 JNI 层实现。
static void android_media_MediaScanner_processFile(JNIEnv *env, jobject thiz,
                              jstring path, jstring mimeType, jobject client)
{
    MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);
    ......
    const char *pathStr = env->GetStringUTFChars(path, NULL);
    ......
    if (mimeType) {
        env->ReleaseStringUTFChars(mimeType, mimeTypeStr);
    }
}
```

上面是 MS 的 JNI 层代码，不知道大家看了以后是否会产生一些疑惑？

我想最大的疑惑可能是，如何才能知道 Java 层的 native_init 函数对应的是 JNI 层的

android_media_MediaScanner_native_init 函数呢？下面就来回答这个问题。

2.4.1 注册 JNI 函数

正如代码中注释的那样，native_init 函数对应的 JNI 函数是 android_media_MediaScanner_native_init，可能细心的读者要问了，你怎么知道 native_init 函数对应的是这个 JNI 函数，而不是其他的呢？莫非是根据函数的名字来确定的？

大家知道，native_init 函数位于 android.media 这个包中，它的全路径名应该是 android.media.MediaScanner.native_init，而 JNI 层函数的名字是 android_media_MediaScanner_native_init。因为在 Native 语言中，符号“.”有着特殊的意义，所以 JNI 层需要把 Java 函数名称（包括包名）中的“.”换成“_”。也就是通过这种方式，native_init 找到了自己 JNI 层的本家兄弟 android.media.MediaScanner.native_init。

上面其实讨论的是 JNI 函数的注册问题，“注册”之意就是将 Java 层的 native 函数和 JNI 层对应的实现函数关联起来，有了这种关联，调用 Java 层的 native 函数时，就能顺利转到 JNI 层对应的函数执行了。而 JNI 函数的注册方法实际上有两种，下面分别做介绍。

1. 静态方法

我们从网上找到的与 JNI 有关的资料，一般都会介绍如何使用这种方法来完成 JNI 函数的注册，这种方法就是根据函数名来找对应的 JNI 函数。它需要 Java 的工具程序 javah 参与，整体流程如下：

- 先编写 Java 代码，然后编译生成 .class 文件。
- 使用 Java 的工具程序 javah，如 javah –o output packagename.classname，这样它会生成一个叫 output.h 的 JNI 层头文件。其中 packagename.classname 是 Java 代码编译后的 class 文件，而在生成的 output.h 文件里，声明了对应的 JNI 层函数，只要实现里面的函数即可。

这个头文件的名字一般都会使用 packagename_class.h 的样式，例如 MediaScanner 对应的 JNI 层头文件就是 android_media_MediaScanner.h。下面来看这种方式生成的头文件：

[-->android_media_MediaScanner.h:: 样例文件]

```
/* DO NOT EDIT THIS FILE - it is machine generated */
  #include <jni.h>  // 必须包含这个头文件，否则编译通不过
/* Header for class android_media_MediaScanner */

#ifndef _Included_android_media_MediaScanner
#define _Included_android_media_MediaScanner
#ifdef __cplusplus
extern "C" {
#endif
...... 略去一部分内容
//processFile 的 JNI 函数
JNIEXPORT void JNICALL Java_android_media_MediaScanner_processFile
```

```
                    (JNIEnv *, jobject, jstring, jstring, jobject);

......// 略去一部分内容
//native_init 对应的 JNI 函数
JNIEXPORT void JNICALL Java_android_media_MediaScanner_native_linit
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

从上面代码中可以发现，native_init 和 processFile 的 JNI 层函数被声明成：

```
//Java 层函数名中如果有一个"_"，转换成 JNI 后就变成了"_1"。
JNIEXPORT void JNICALL Java_android_media_MediaScanner_native_linit
JNIEXPORT void JNICALL Java_android_media_MediaScanner_processFile
```

需要解释一下静态方法中 native 函数是如何找到对应的 JNI 函数的。其实，过程非常简单：

当 Java 层调用 native_init 函数时，它会从对应的 JNI 库中寻找 Java_android_media_MediaScanner_native_linit 函数，如果没有，就会报错。如果找到，则会为这个 native_init 和 Java_android_media_MediaScanner_native_linit 建立一个关联关系，其实就是保存 JNI 层函数的函数指针。以后再调用 native_init 函数时，直接使用这个函数指针就可以了，当然这项工作是由虚拟机完成的。

从这里可以看出，静态方法就是根据函数名来建立 Java 函数和 JNI 函数之间的关联关系的，而且它要求 JNI 层函数的名字必须遵循特定的格式。这种方法也有几个弊端，即：

- 需要编译所有声明了 native 函数的 Java 类，每个所生成的 class 文件都得用 javah 生成一个头文件。
- javah 生成的 JNI 层函数名特别长，书写起来很不方便。
- 初次调用 native 函数时要根据函数名字搜索对应的 JNI 层函数来建立关联关系，这样会影响运行效率。

有什么办法可以克服上面三个弊端吗？根据上面的介绍可知，Java native 函数是通过函数指针来和 JNI 层函数建立关联关系的。如果直接让 native 函数知道 JNI 层对应函数的函数指针，不就万事大吉了吗？这就是下面要介绍的第二种方法：动态注册法。

2. 动态注册

既然 Java native 函数和 JNI 函数是一一对应的，那么是不是会有一个结构来保存这种关联关系呢？答案是肯定的。在 JNI 技术中，用来记录这种一一对应关系的，是一个叫 JNINativeMethod 的结构，其定义如下：

```
typedef struct {
   //Java 中 native 函数的名字，不用携带包的路径，例如"native_init"。
```

```
const char* name;
//Java 函数的签名信息，用字符串表示，是参数类型和返回值类型的组合。
    const char* signature;
    void*       fnPtr;  //JNI 层对应函数的函数指针，注意它是 void* 类型。
} JNINativeMethod;
```

应该如何使用这个结构体呢？来看 MediaScanner JNI 层是如何做的，代码如下所示：

[-->android_media_MediaScanner.cpp]

```
// 定义一个 JNINativeMethod 数组，其成员就是 MS 中所有 native 函数的一一对应关系。
static JNINativeMethod gMethods[] = {
    ......
{
"processFile" //Java 中 native 函数的函数名。
//processFile 的签名信息，签名信息的知识后面再做介绍。
"(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
 (void *)android_media_MediaScanner_processFile //JNI 层对应的函数指针。
},
 ......

{
"native_init",
"()V",
(void *)android_media_MediaScanner_native_init
},
  ......
};
// 注册 JNINativeMethod 数组
int register_android_media_MediaScanner(JNIEnv *env)
{
   // 调用 AndroidRuntime 的 registerNativeMethods 函数，第二个参数表明是 Java 中的哪个类
    return AndroidRuntime::registerNativeMethods(env,
                "android/media/MediaScanner", gMethods, NELEM(gMethods));
}
```

AndroidRunTime 类提供了一个 registerNativeMethods 函数来完成注册工作，下面来看 registerNativeMethods 的实现，代码如下：

[-->AndroidRunTime.cpp]

```
int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className,const JNINativeMethod* gMethods, int numMethods)
{
    // 调用 jniRegisterNativeMethods 函数完成注册
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

其中 jniRegisterNativeMethods 是 Android 平台中为了方便 JNI 使用而提供的一个帮助函数，其代码如下所示：

[-->JNIHelp.c]

```
int jniRegisterNativeMethods(JNIEnv* env, const char* className,
                             const JNINativeMethod* gMethods, int numMethods)
{
    jclass clazz;
    clazz = (*env)->FindClass(env, className);
    ......
    // 实际上是调用 JNIEnv 的 RegisterNatives 函数完成注册的
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        return -1;
    }
    return 0;
}
```

wow，好像很麻烦啊！其实动态注册的工作，只用两个函数就能完成。总结如下：

```
/*
  env 指向一个 JNIEnv 结构体，它非常重要，后面会讨论它。classname 为对应的 Java 类名，由于
  JNINativeMethod 中使用的函数名并非全路径名，所以要指明是哪个类。
*/
jclass clazz =  (*env)->FindClass(env, className);
// 调用 JNIEnv 的 RegisterNatives 函数，注册关联关系。
(*env)->RegisterNatives(env, clazz, gMethods, numMethods);
```

所以，在自己的 JNI 层代码中使用这种方法，就可以完成动态注册了。这里还有一个很棘手的问题：这些动态注册的函数在什么时候和什么地方被调用呢？这里就不卖关子了，直接给出该问题的答案：

当 Java 层通过 System.loadLibrary 加载完 JNI 动态库后，紧接着会查找该库中一个叫 JNI_OnLoad 的函数。如果有，就调用它，而动态注册的工作就是在这里完成的。

所以，如果想使用动态注册方法，就必须实现 JNI_OnLoad 函数，只有在这个函数中才有机会完成动态注册的工作。静态注册的方法则没有这个要求，但建议大家也实现这个 JNI_OnLoad 函数，因为有一些初始化工作是可以在这里做的。

那么，libmedia_jni.so 的 JNI_OnLoad 函数是在哪里实现的呢？由于多媒体系统很多地方都使用了 JNI，所以“码农”把它放到 android_media_MediaPlayer.cpp 中了，代码如下所示：

[-->android_media_MediaPlayer.cpp]

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
   // 该函数的第一个参数类型为 JavaVM，这可是虚拟机在 JNI 层的代表喔，每个 Java 进程只有一个
   // 这样的 JavaVM。
    JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        goto bail;
```

```
    }
    ..... // 动态注册 MediaScanner 的 JNI 函数。
    if (register_android_media_MediaScanner(env) < 0) {
        goto bail;
    }
    .....
    return JNI_VERSION_1_4;// 必须返回这个值，否则会报错。
}
```

JNI 函数注册的相关内容介绍完了。下面来关注一下 JNI 技术中其他的几个重要部分。

注意 JNI 层代码中一般要包含 jni.h 这个头文件。Android 源码中提供了一个帮助头文件 JNIHelp.h，它内部其实就包含了 jni.h，所以我们在自己的代码中直接包含这个 JNIHelp.h 即可。

2.4.2 数据类型转换

前面的分析解决了 JNI 函数的注册问题，下面来研究数据类型转换的问题。

在 Java 中调用 native 函数传递的参数是 Java 数据类型，那么这些参数类型到了 JNI 层会变成什么呢？

Java 数据类型分为基本数据类型和引用数据类型两种，JNI 层也是区别对待这二者的。先来看基本数据类型的转换。

1. 基本数据类型的转换

基本数据类型的转换很简单，可用表 2-1 表示：

表 2-1 基本数据类型的转换关系表

Java	Native 类型	符号属性	字长
boolean	jboolean	无符号	8 位
byte	jbyte	无符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jint	有符号	32 位
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

上面列出了 Java 基本数据类型与 JNI 层数据类型对应的转换关系，非常简单。不过，务必注意转换成 Native 类型后对应数据类型的字长，例如 jchar 在 Native 语言中是 16 位，占两个字节，这和普通的 char 占一个字节的情况是完全不一样的。

接下来看 Java 引用数据类型的转换。

2. 引用数据类型的转换

引用数据类型的转换如表 2-2 所示：

表 2-2　Java 引用数据类型的转换关系表

Java 引用类型	Native 类型	Java 引用类型	Native 类型
All objects	jobject	char[]	jcharArray
java.lang.Class 实例	jclass	short[]	jshortArray
java.lang.String 实例	jstring	int[]	jintArray
Object[]	jobjectArray	long[]	jlongArray
boolean[]	jbooleanArray	float[]	floatArray
byte[]	jbyteArray	double[]	jdoubleArray
java.lang.Throwable 实例	jthrowable		

由上表可知：除了 Java 中基本数据类型的数组、Class、String 和 Throwable 外，其余所有 Java 对象的数据类型在 JNI 中都用 jobject 表示。

这一点太让人惊讶了！看看 processFile 这个函数：

```
//Java 层 processFile 有三个参数。
processFile(String path, String mimeType,MediaScannerClient client);
//JNI 层对应的函数，最后三个参数与 processFile 的参数对应。
android_media_MediaScanner_processFile(JNIEnv *env, jobject thiz,
                jstring path, jstring mimeType, jobject client)
```

从上面这段代码中可以发现：

- Java 的 String 类型在 JNI 层对应为 jstring 类型。
- Java 的 MediaScannerClient 类型在 JNI 层对应为 jobject。

如果对象类型都用 jobject 表示，就好比是 Native 层的 void* 类型一样，对“码农”来说，它们是完全透明的。既然是透明的，那该如何使用和操作它们呢？在回答这个问题之前，再来仔细看看上面的 android_media_MediaScanner_processFile 函数，代码如下：

```
/*
Java 中的 processFile 只有三个参数，为什么 JNI 层对应的函数会有五个参数呢？第一个参数中的
JNIEnv 是什么？（稍后介绍。）第二个参数 jobject 代表 Java 层的 MediaScanner 对象，它表示是在哪个
MediaScanner 对象上调用的 processFile。如果 Java 层是 static 函数，那么
这个参数将是 jclass，表示是在调用哪个 Java Class 的静态函数。
*/
android_media_MediaScanner_processFile(JNIEnv *env,
                    jobject thiz,
                    jstring path, jstring mimeType, jobject client)
```

上面的代码，引出了下面几个小节的主角 JNIEnv。

2.4.3 JNIEnv 介绍

JNIEnv 是一个与线程相关的代表 JNI 环境的结构体，图 2-3 展示了 JNIEnv 的内部结构：

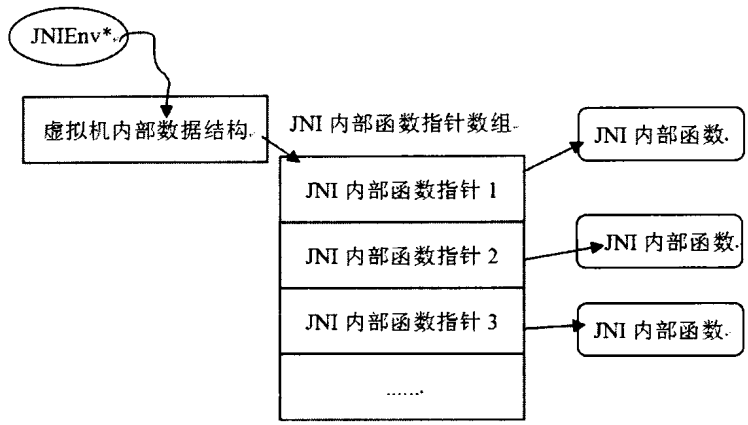


图 2-3 JNIEnv 内部结构简图

从上图可知，JNIEnv 实际上就是提供了一些 JNI 系统函数。通过这些函数可以做到：

- 调用 Java 的函数。
- 操作 jobject 对象等很多事情。

后面的小节中将具体介绍如何使用 JNIEnv 中的函数。这里，先介绍一个关于 JNIEnv 的重要知识点。

上面提到说 JNIEnv 是一个与线程相关的变量。也就是说，线程 A 有一个 JNIEnv，线程 B 有一个 JNIEnv。由于线程相关，所以不能在线程 B 中使用线程 A 的 JNIEnv 结构体。读者可能会问，JNIEnv 不都是 native 函数转换成 JNI 层函数后由虚拟机传进来的吗？使用传进来的这个 JNIEnv 总不会错吧？是的，在这种情况下使用当然不会出错。只是当后台线程收到一个网络消息，而又需要由 Native 层函数主动回调 Java 层函数时，JNIEnv 从何而来呢？根据前面的介绍可知，我们不能保存另外一个线程的 JNIEnv 结构体，然后把它放到后台线程中来用。这该如何是好？

还记得前面介绍的那个 JNI_OnLoad 函数吗？它的第一个参数是 JavaVM，它是虚拟机在 JNI 层的代表，代码如下所示：

```
// 全进程只有一个 JavaVM 对象，所以可以保存，并且在任何地方使用都没有问题。
jint JNI_OnLoad(JavaVM* vm, void* reserved)
```

正如上面代码所说，不论进程中有多少个线程，JavaVM 却是独此一份，所以在任何地方都可以使用它。那么，JavaVM 和 JNIEnv 又有什么关系呢？答案如下：

- 调用 JavaVM 的 AttachCurrentThread 函数，就可得到这个线程的 JNIEnv 结构体。这样就可以在后台线程中回调 Java 函数了。
- 另外，在后台线程退出前，需要调用 JavaVM 的 DetachCurrentThread 函数来释放对应的资源。

再来看 JNIEnv 的作用。

2.4.4　通过 JNIEnv 操作 jobject

前面提到过一个问题，即 Java 的引用类型除了少数几个外，最终在 JNI 层都会用 jobject 来表示对象的数据类型，那么该如何操作这个 jobject 呢？

从另外一个角度来解释这个问题。一个 Java 对象是由什么组成的？当然是它的成员变量和成员函数了。那么，操作 jobject 的本质就应当是操作这些对象的成员变量和成员函数。所以应先来看与成员变量及成员函数有关的内容。

1.jfieldID 和 jmethodID 介绍

我们知道，成员变量和成员函数都是由类定义的，它们是类的属性，所以在 JNI 规则中，用 jfieldID 和 jmethodID 来表示 Java 类的成员变量和成员函数，可通过 JNIEnv 的下面两个函数得到：

```
jfieldID GetFieldID(jclass clazz,const char *name, const char *sig);
jmethodID GetMethodID(jclass clazz, const char *name,const char *sig);
```

其中，jclass 代表 Java 类，name 表示成员函数或成员变量的名字，sig 为这个函数和变量的签名信息。如前所示，成员函数和成员变量都是类的信息，这两个函数的第一个参数都是 jclass。

在 MS 中是怎么使用它们的呢？来看代码，如下所示：

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 构造函数]

```
MyMediaScannerClient(JNIEnv *env, jobject client).....
{
  // 先找到 android.media.MediaScannerClient 类在 JNI 层中对应的 jclass 实例。
  jclass mediaScannerClientInterface =
            env->FindClass("android/media/MediaScannerClient");
  // 取出 MediaScannerClient 类中函数 scanFile 的 jMethodID。
  mScanFileMethodID = env->GetMethodID(
                          mediaScannerClientInterface, "scanFile",
                          "(Ljava/lang/String;JJ)V");
  // 取出 MediaScannerClient 类中函数 handleStringTag 的 jMethodID。
  mHandleStringTagMethodID = env->GetMethodID(
                            mediaScannerClientInterface, "handleStringTag",
                            "(Ljava/lang/String;Ljava/lang/String;)V");
  ......
}
```

在上面的代码中，将 scanFile 和 handleStringTag 函数的 jmethodID 保存为 MyMediaScannerClient 的成员变量。为什么这里要把它们保存起来呢？这个问题涉及一个关于程序运行效率的知识点：

如果每次操作 jobject 前都去查询 jmethodID 或 jfieldID，那么将会影响程序运行的效率，所以我们在初始化的时候可以取出这些 ID 并保存起来以供后续使用。

取出 jmethodID 后，又该怎么用它呢？

2. 使用 jfieldID 和 jmethodID

下面再看一个例子，其代码如下所示：

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 的 scanFile]

```
virtual bool scanFile(const char* path, long long lastModified,
                      long long fileSize)
    {
        jstring pathStr;
        if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;

        /*
          调用 JNIEnv 的 CallVoidMethod 函数，注意 CallVoidMethod 的参数：
          第一个是代表 MediaScannerClient 的 jobject 对象，
          第二个参数是函数 scanFile 的 jmethodID，后面是 Java 中 scanFile 的参数。
        */
        mEnv->CallVoidMethod(mClient, mScanFileMethodID, pathStr,
                             lastModified, fileSize);

        mEnv->DeleteLocalRef(pathStr);
        return (!mEnv->ExceptionCheck());
}
```

明白了，通过 JNIEnv 输出 CallVoidMethod，再把 jobject、jMethodID 和对应的参数传进去，JNI 层就能够调用 Java 对象的函数了！

实际上 JNIEnv 输出了一系列类似 CallVoidMethod 的函数，形式如下：

```
NativeType Call<type>Method(JNIEnv *env, jobject obj,jmethodID methodID, ...)。
```

其中 type 对应 Java 函数的返回值类型，例如 CallIntMethod、CallVoidMethod 等。

上面是针对非 static 函数的，如果想调用 Java 中的 static 函数，则用 JNIEnv 输出的 CallStatic<Type>Method 系列函数。

现在，我们已了解了如何通过 JNIEnv 操作 jobject 的成员函数，那如何通过 jfieldID 操作 jobject 的成员变量呢？这里，直接给出整体解决方案，如下所示：

```
// 获得 fieldID 后，可调用 Get<type>Field 系列函数获取 jobject 对应的成员变量的值。
NativeType Get<type>Field(JNIEnv *env, jobject obj,jfieldID fieldID)
// 或者调用 Set<type>Field 系列函数来设置 jobject 对应的成员变量的值。
void Set<type>Field(JNIEnv *env,jobject obj, jfieldID fieldID,NativeType value)
```

```
// 下面我们列出一些常用的 Get/Set 函数。
GetObjectField()          SetObjectField()
GetBooleanField()         SetBooleanField()
GetByteField()            SetByteField()
GetCharField()            SetCharField()
GetShortField()           SetShortField()
GetIntField()             SetIntField()
GetLongField()            SetLongField()
GetFloatField()           SetFloatField()
GetDoubleField()          SetDoubleField()
```

通过本节的介绍，相信读者已经了解了 jfieldID 和 jmethodID 的作用，也知道如何通过 JNIEnv 的函数来操作 jobject 了。虽然 jobject 是透明的，但有了 JNIEnv 的帮助，还是能轻松操作 jobject 背后的实际对象的。

2.4.5　jstring 介绍

Java 中的 String 也是引用类型，不过由于它的使用频率较高，所以在 JNI 规范中单独创建了一个 jstring 类型来表示 Java 中的 String 类型。虽然 jstring 是一种独立的数据类型，但是它并没有提供成员函数以便操作。而 C++ 中的 string 类是有自己的成员函数的。那么该怎么操作 jstring 呢？还是得依靠 JNIEnv 提供的帮助。这里看几个有关 jstring 的函数：

- 调用 JNIEnv 的 NewString(JNIEnv *env, const jchar *unicodeChars,jsize len)，可以从 Native 的字符串得到一个 jstring 对象。其实，可以把一个 jstring 对象看成是 Java 中 String 对象在 JNI 层的代表，也就是说，jstring 就是一个 Java String。但由于 Java String 存储的是 Unicode 字符串，所以 NewString 函数的参数也必须是 Unicode 字符串。
- 调用 JNIEnv 的 NewStringUTF 将根据 Native 的一个 UTF-8 字符串得到一个 jstring 对象。在实际工作中，这个函数用得最多。
- 上面两个函数将本地字符串转换成了 Java 的 String 对象，JNIEnv 还提供了 GetStringChars 函数和 GetStringUTFChars 函数，它们可以将 Java String 对象转换成本地字符串。其中 GetStringChars 得到一个 Unicode 字符串，而 GetStringUTFChars 得到一个 UTF-8 字符串。
- 另外，如果在代码中调用了上面几个函数，在做完相关工作后，就都需要调用 ReleaseStringChars 函数或 ReleaseStringUTFChars 函数来对应地释放资源，否则会导致 JVM 内存泄露。这一点和 jstring 的内部实现有关，读者写代码时务必注意这个问题。

为了加深印象，来看 processFile 是怎么做的。

[-->android_media_MediaScanner.cpp]

```
static void
android_media_MediaScanner_processFile(JNIEnv *env, jobject thiz, jstring path,
```

```
jstring mimeType, jobject client)
{
    MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);
    ......
    // 调用 JNIEnv 的 GetStringUTFChars 得到本地字符串 pathStr
    const char *pathStr = env->GetStringUTFChars(path, NULL);
    ......
    // 使用完后，必须调用 ReleaseStringUTFChars 释放资源
    env->ReleaseStringUTFChars(path, pathStr);
    ......
}
```

2.4.6 JNI 类型签名介绍

先来看动态注册中的一段代码：

```
tatic JNINativeMethod gMethods[] = {
    ......
{
"processFile"
//processFile 的签名信息，这么长的字符串，是什么意思？
"(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
 (void *)android_media_MediaScanner_processFile
},
  ......
}
```

上面代码中的 JNINativeMethod 前面已经见过了，不过其中那个很长的字符串 "(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V" 是什么意思呢？

根据前面的讲解可知，它是 Java 中对应函数的签名信息，由参数类型和返回值类型共同组成。不过为什么需要这个签名信息呢？

这个问题的答案比较简单。因为 Java 支持函数重载，也就是说，可以定义同名但不同参数的函数。但仅仅根据函数名是没法找到具体函数的。为了解决这个问题，JNI 技术中就将参数类型和返回值类型的组合作为了一个函数的签名信息，有了签名信息和函数名，就能很顺利地找到 Java 中的函数了。

JNI 规范定义的函数签名信息看起来很别扭，不过习惯就好了。它的格式是：

```
( 参数 1 类型标示参数 2 类型标示 ... 参数 n 类型标示 ) 返回值类型标示
```

来看 processFile 的例子：

```
Java 中的函数定义为 void processFile(String path, String mimeType)。
对应的 JNI 函数签名就是
(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V
其中，括号内是参数类型的标识，最右边是返回值类型的标识，void 类型对应的标识是 V。
当参数的类型是引用类型时，其格式是" L 包名 ;"，其中包名中的" ." 换成" /"。上面例子中的
 Ljava/lang/String; 表示是一个 Java String 类型。
```

函数签名不仅看起来麻烦，写起来更麻烦，稍微写错一个标点就会导致注册失败。所以，在具体编码时，读者可以定义字符串宏，这样改起来也方便。

表 2-3 是常见的类型标识：

表 2-3　类型标识示意表

类型标识	Java 类型	类型标示	Java 类型
Z	boolean	F	float
B	byte	D	double
C	char	L/java/langaugeString;	String
S	short	[I	int[]
I	int	[L/java/lang/object;	Object[]
J	long		

上面列出了一些常用的类型标识。请注意，如果 Java 类型是数组，则标识中会有一个“[”，另外，引用类型（除基本类型的数组外）的标识最后都有一个“;”。

再来看一个小例子，如表 2-4 所示：

表 2-4　函数签名的小例子

函数签名	Java 函数
“()Ljava/lang/String;”	String f()
“(ILjava/lang/Class;)J”	long f(int i, Class c)
“([B)V”	void f(byte[] bytes)

请大家结合表 2-3 和表 2-4 左栏的内容写出对应的 Java 函数。

虽然函数签名信息很容易写错，但 Java 提供一个叫 javap 的工具能帮助生成函数或变量的签名信息，它的用法如下：

```
javap -s -p xxx
```

其中 xxx 为编译后的 class 文件，s 表示输出内部数据类型的签名信息，p 表示打印所有函数和成员的签名信息，默认只会打印 public 成员和函数的签名信息。

有了 javap，就不用死记硬背上面的类型标示了。

2.4.7　垃圾回收

我们知道，Java 中创建的对象最后是由垃圾回收器来回收和释放内存的，可它对 JNI 有什么影响呢？下面看一个例子：

[--> 垃圾回收的例子]

```
static jobject save_thiz = NULL; // 定义一个全局的 jobject
static void
```

```
android_media_MediaScanner_processFile(JNIEnv *env, jobject thiz, jstring path,
                    jstring mimeType, jobject client)
{
  ......
  // 保存 Java 层传入的 jobject 对象，代表 MediaScanner 对象
  save_thiz = thiz;
  ......
  return;
}
// 假设在某个时间，有地方调用 callMediaScanner 函数
void callMediaScanner()
{
  // 在这个函数中操作 save_thiz，会有问题吗?
}
```

上面的做法肯定会有问题，因为和 save_thiz 对应的 Java 层中的 MediaScanner 很有可能已经被垃圾回收了，也就是说，save_thiz 保存的这个 jobject 可能是一个野指针，如果使用它，后果会很严重。

可能有人要问，对一个引用类型执行赋值操作，它的引用计数不会增加吗？而垃圾回收机制只会保证那些没有被引用的对象才会被清理。问得对，但如果在 JNI 层使用下面这样的语句，是不会增加引用计数的。

```
save_thiz = thiz; // 这种赋值不会增加 jobject 的引用计数。
```

引用计数没有增加，thiz 就有可能被回收，那该怎么办？不必担心，JNI 规范已很好地解决了这一问题，JNI 技术一共提供了三种类型的引用，它们分别是：

- Local Reference：本地引用。在 JNI 层函数中使用的非全局引用对象都是 Local Reference，它包括函数调用时传入的 jobject 和在 JNI 层函数中创建的 jobject。Local Reference 最大的特点就是，一旦 JNI 层函数返回，这些 jobject 就可能被垃圾回收。
- Global Reference：全局引用，这种对象如不主动释放，它永远不会被垃圾回收。
- Weak Global Reference：弱全局引用，一种特殊的 Global Reference，在运行过程中可能会被垃圾回收。所以在使用它之前，需要调用 JNIEnv 的 IsSameObject 判断它是否被回收了。

平时用得最多的是 Local Reference 和 Global Reference，下面来看一个实例，代码如下所示：

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 构造函数]

```
MyMediaScannerClient(JNIEnv *env, jobject client)
        :   mEnv(env),
        // 调用 NewGlobalRef 创建一个 Global Reference，这样 mClient 就不用担心被回收了。
            mClient(env->NewGlobalRef(client)),
            mScanFileMethodID(0),
```

```
            mHandleStringTagMethodID(0),
            mSetMimeTypeMethodID(0)
{
  ......
}
// 析构函数
virtual ~MyMediaScannerClient()
{
   mEnv->DeleteGlobalRef(mClient);// 调用 DeleteGlobalRef 释放这个全局引用。
 }
```

每当 JNI 层想要保存 Java 层中的某个对象时，就可以使用 Global Reference，使用完后记住释放它就可以了。这一点很容易理解。下面要讲有关 Local Reference 的一个问题，还是先看实例，代码如下所示：

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 的 scanFile]

```
 virtual bool scanFile(const char* path, long long lastModified,long long fileSize)
{
   jstring pathStr;
   // 调用 NewStringUTF 创建一个 jstring 对象，它是 Local Reference 类型。
   if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;
        // 调用 Java 的 scanFile 函数，把这个 jstring 传进去
        mEnv->CallVoidMethod(mClient, mScanFileMethodID, pathStr,
                             lastModified, fileSize);
     /*
      根据 Local Reference 的说明，这个函数返回后，pathStr 对象就会被回收。所以
      下面这个 DeleteLocalRef 调用看起来是多余的，其实不然，这里解释一下原因：
      1）如果不调用 DeleteLocalRef，pathStr 将在函数返回后被回收。
      2）如果调用 DeleteLocalRef，pathStr 会立即被回收。这两者看起来没什么区别，
      不过代码如果像下面这样，虚拟机的内存就会很快被耗尽：
      for(int i = 0; i < 100; i++)
      {
            jstring pathStr = mEnv->NewStringUTF(path);
            ......// 做一些操作
           //mEnv->DeleteLocalRef(pathStr); // 不立即释放 Local Reference
      }
      如果上面代码的循环中未调用 DeleteLocalRef，则会创建 100 个 jstring，
      那么内存的耗费就非常可观了！
     */
         mEnv->DeleteLocalRef(pathStr);
         return (!mEnv->ExceptionCheck());
}
```

所以，没有及时回收 Local Reference 或许是进程占用内存过多的一个原因，请务必注意这一点。

2.4.8 JNI 中的异常处理

JNI 中也有异常，不过它和 C++、Java 的异常不太一样。如果调用 JNIEnv 的某些函数出错了，则会产生一个异常，但这个异常不会中断本地函数的执行，直到从 JNI 层返回到 Java 层后，虚拟机才会抛出这个异常。虽然在 JNI 层中产生的异常不会中断本地函数的运行，但一旦产生异常后，就只能做一些资源清理工作了（例如释放全局引用，或者 ReleaseStringChars）。如果这时调用除上面所说函数之外的其他 JNIEnv 函数，则会导致程序死掉。

来看一个和异常处理有关的例子，代码如下所示：

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 的 scanFile 函数]

```
virtual bool scanFile(const char* path, long long lastModified,
                    long long fileSize)
{
        jstring pathStr;
        //NewStringUTF 调用失败后，直接返回，不能再干别的事情了。
        if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;
        ......
}
```

JNI 层函数可以在代码中截获和修改这些异常，JNIEnv 提供了三个函数给予帮助：

- ExceptionOccured 函数，用来判断是否发生异常。
- ExceptionClear 函数，用来清理当前 JNI 层中发生的异常。
- ThrowNew 函数，用来向 Java 层抛出异常。

异常处理是 JNI 层代码必须关注的事情，读者在编写代码时务小心对待。

2.5 本章小结

本章通过一个实例介绍了 JNI 技术中的几个重要方面，包括：

- JNI 函数的注册方法。
- Java 和 JNI 层数据类型的转换。
- JNIEnv 和 jstring 的使用方法，以及 JNI 中的类型签名。
- 垃圾回收在 JNI 层中的使用，以及异常处理方面的知识。

相信掌握了上面的知识后，我们会对 JNI 技术有一个比较清晰的认识。这里，还要建议读者再认真阅读一下 JDK 文档中的《Java Native Interface Specification》，它完整和细致地阐述了 JNI 技术的各个方面，堪称深入学习 JNI 的权威指南。



第3章 深入理解init

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。

- init.c(*system/core/init/init.c*)
- parser.c(*system/core/init/parser.c*)
- builtins.c(*system/core/init/builtins.c*)
- keywords.h(*system/core/init/keywords/h*)
- init.rc(*system/core/rootdir/init.rc*)
- properties_service.c(*system/core/init/properties_service.c*)
- libc_init_dynamic.c(*bionic/libc/bionic/libc_init_common.c*)
- libc_init_common.c(*bionic/libc/bionic/libc_init_common.c*)
- properties.c(*system/core/libcutils/properties.c*)

3.1 概述

init 是一个进程，确切地说，它是 Linux 系统中用户空间的第一个进程。由于 Android 是基于 Linux 内核的，所以 init 也是 Android 系统中用户空间的第一个进程，它的进程号是 1。作为天字第一号进程，init 被赋予了很多极其重要的工作职责，本章将关注其中两个比较重要的职责：

- init 进程负责创建系统中的几个关键进程，尤其是下一章要介绍的 zygote，它更是 Java 世界的开创者。那么，init 进程是如何创建 zygote 的呢？
- Android 系统有很多属性，于是 init 就提供了一个 property service（属性服务）来管理它们。那么这个属性服务是怎么工作的呢？

本章就将从以上两方面的内容着手来分析 init，即：

- init 如何创建 zygote。
- init 的属性服务是如何工作的。

3.2 init 分析

init 进程的入口函数是 main，它的代码如下所示：

[-->init.c]

```
int main(int argc, char **argv)
{
    int device_fd = -1;
    int property_set_fd = -1;
    int signal_recv_fd = -1;
    int keychord_fd = -1;
    int fd_count;
    int s[2];
    int fd;
    struct sigaction act;
    char tmp[PROP_VALUE_MAX];
    struct pollfd ufds[4];
    char *tmpdev;
    char* debuggable;

   // 设置子进程退出的信号处理函数，该函数为 sigchld_handler。
    act.sa_handler = sigchld_handler;
    act.sa_flags = SA_NOCLDSTOP;
    act.sa_mask = 0;
    act.sa_restorer = NULL;
    sigaction(SIGCHLD, &act, 0);

    ......// 创建一些文件夹，并挂载设备，这些是与 Linux 相关的，不做过多讨论。
    mkdir("/dev/socket", 0755);
```

```
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);

    // 重定向标准输入 / 输出 / 错误输出到 /dev/_null_。
    open_devnull_stdio();
    /*
      设置 init 的日志输出设备为 /dev/__kmsg__，不过该文件打开后，会立即被 unlink 了，
      这样，其他进程就无法打开这个文件读取日志信息了。
    */
    log_init();

   // 上面涉及很多与 Linux 系统相关的知识，不熟悉的读者可自行研究，它们不影响我们的分析。
   // 解析 init.rc 配置文件
    parse_config_file("/init.rc");

    ......
    // 下面这个函数通过读取 /proc/cpuinfo 得到机器的 Hardware 名，笔者的 HTC G7 手机为 bravo。
    get_hardware_name();
    snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
    // 解析这个和机器相关的配置文件，笔者的 G7 手机对应文件为 init.bravo.rc。
    parse_config_file(tmp);

    /*
      解析完上述两个配置文件后，会得到一系列的 Action（动作），下面两句代码将执行那些处于
      early-init 阶段的 Action。init 将动作执行的时间划分为四个阶段：early-init、init、
      early-boot、boot。由于有些动作必须在其他动作完成后才能执行，所以就有了先后之分。哪些
      动作属于哪个阶段由配置文件决定，后面会介绍配置文件的相关知识。
    */
    action_for_each_trigger("early-init", action_add_queue_tail);
    drain_action_queue();

    /*
      创建利用 Uevent 与 Linux 内核交互的 socket。关于 Uevent 的知识，第 9 章中对
      Vold 进行分析时会做介绍。
    */
    device_fd = device_init();
    // 初始化和属性相关的资源
    property_init();
    // 初始化 /dev/keychord 设备，这与调试有关，本书不讨论它的用法。读者可以自行研究，
    // 内容比较简单。
    keychord_fd = open_keychord();

    ......
  /*
    INIT_IMAGE_FILE 定义为 "/initlogo.rle"，下面这个函数将加载这个文件作为系统的开机
   画面，注意，它不是开机动画控制程序 bootanimation 加载的开机动画文件。
  */
if(load_565rle_image(INIT_IMAGE_FILE) ) {
   /*
```

```
    如果加载 initlogo.rle 文件失败（可能是没有这个文件），则会打开 /dev/ty0 设备，并
    输出 "ANDROID" 的字样作为开机画面。在模拟器上看到的开机画面就是它。
    */
     ......
     }
}

 if (qemu[0])
     import_kernel_cmdline(1);
......
 // 调用 property_set 函数设置属性项，一个属性项包括属性名和属性值。
 property_set("ro.bootloader", bootloader[0] ? bootloader : "unknown");

 ......// 执行位于 init 阶段的动作
 action_for_each_trigger("init", action_add_queue_tail);
 drain_action_queue();

 // 启动属性服务
 property_set_fd = start_property_service();

 /*
 调用 socketpair 函数创建两个已经 connect 好的 socket，socketpair 是 Linux 的系统调用，
 不熟悉的读者可以利用 man socketpair 查询相关信息。后面就会知道它们的用处了。
 */
 if (socketpair(AF_UNIX, SOCK_STREAM, 0, s) == 0) {
     signal_fd = s[0];
     signal_recv_fd = s[1];
     ......
 }

 ......

 // 执行配置文件中 early-boot 和 boot 阶段的动作。
 action_for_each_trigger("early-boot", action_add_queue_tail);
 action_for_each_trigger("boot", action_add_queue_tail);
 drain_action_queue();
   ......

 //init 关注来自四个方面的事情。
 ufds[0].fd = device_fd;//device_fd 用于监听来自内核的 Uevent 事件。
 ufds[0].events = POLLIN;
 ufds[1].fd = property_set_fd;//property_set_fd 用于监听来自属性服务器的事件。
 ufds[1].events = POLLIN;
 //signal_recv_fd 由 socketpair 创建，它的事件来自另外一个 socket。
 ufds[2].fd = signal_recv_fd;
 ufds[2].events = POLLIN;
 fd_count = 3;
 if (keychord_fd > 0) {
    // 如果 keychord 设备初始化成功，则 init 也会关注来自这个设备的事件。
     ufds[3].fd = keychord_fd;
```

```
        ufds[3].events = POLLIN;
        fd_count++;
    }
    ......

#if BOOTCHART
    ...... // 与 Boot char 相关，不做讨论了。
    /*
    Boot chart 是一个小工具，它能对系统的性能进行分析，并生成系统启动过程的图表，
    以提供一些有价值的信息，而这些信息最大的用处就是帮助提升系统的启动速度。
    */
#endif
  for(;;) {
        // 从此 init 将进入一个无限循环。
        int nr, i, timeout = -1;

        for (i = 0; i < fd_count; i++)
            ufds[i].revents = 0;

        // 在循环中执行动作
        drain_action_queue();
        restart_processes(); // 重启那些已经死去的进程
......
#if BOOTCHART
        ...... // Boot Chart 相关
#endif
        // 调用 poll 等待一些事情的发生
        nr = poll(ufds, fd_count, timeout);
        ......
        //ufds[2] 保存的是 signal_recv_fd，用于接收来自 socket 的消息。
        if (ufds[2].revents == POLLIN) {
            // 有一个子进程去世，init 要处理这个事情。
            read(signal_recv_fd, tmp, sizeof(tmp));
            while (!wait_for_one_process(0))
                ;
            continue;
        }

        if (ufds[0].revents == POLLIN)
            handle_device_fd(device_fd);// 处理 Uevent 事件。
        if (ufds[1].revents == POLLIN)
            handle_property_set_fd(property_set_fd);// 处理属性服务的事件。
        if (ufds[3].revents == POLLIN)
            handle_keychord(keychord_fd);// 处理 keychord 事件。
    }

    return 0;
}
```

从上面的代码可知，init 的工作任务还是很重的。上面的代码虽已省略了不少行，可结

果还是很长，不过从本章要分析的两个知识点来看，可将 init 的工作流程精简为以下四点：

- ❑ 解析两个配置文件，我们将分析其中对 init.rc 文件的解析。
- ❑ 执行各个阶段的动作，创建 zygote 的工作就是在其中的某个阶段完成的。
- ❑ 调用 property_init 初始化属性相关的资源，并且通过 property_start_service 启动属性服务。
- ❑ init 进入一个无限循环，并且等待一些事情的发生。重点关注 init 如何处理来自 socket 和来自属性服务器的相关事情。

提示 *精简工作流程，是本书后面分析代码时常用的方法。读者在分析代码的过程中，也可使用这种方法。*

3.2.1 解析配置文件

根据上面的代码可知，在 init 中会解析两个配置文件，其中一个是系统配置文件 init.rc，另外一个是与硬件平台相关的配置文件。以 HTC G7 手机为例，这个配置文件名为 init.bravo.rc，其中 bravo 是硬件平台的名称。对这两个配置文件进行解析，调用的是同一个 parse_config_file 函数。下面就来看这个函数，在分析过程中以 init.rc 为主。

[-->parser.c]

```
int parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);// 读取配置文件的内容，这个文件是 init.rc。
    if (!data) return -1;
    parse_config(fn, data); // 调用 parse_config 做真正的解析。
    return 0;
}
```

读取完文件的内容后，将调用 parse_config 进行解析，这个函数的代码如下所示：

[-->parser.c]

```
static void parse_config(const char *fn, char *s)
{
  struct parse_state state;
  char *args[SVC_MAXARGS];
  int nargs;

  nargs = 0;
  state.filename = fn;
  state.line = 1;
  state.ptr = s;
  state.nexttoken = 0;
```

```
    state.parse_line = parse_line_no_op; // 设置解析函数，不同的内容用不同的解析函数。
    for (;;) {
      switch (next_token(&state)) {
        case T_EOF:
              state.parse_line(&state, 0, 0);
              return;
        case T_NEWLINE:
              if (nargs) {
                  // 得到关键字的类型。
                  int kw = lookup_keyword(args[0]);
                  if (kw_is(kw, SECTION)) { // 判断关键字类型是不是 SECTION。
                      state.parse_line(&state, 0, 0);
                     parse_new_section(&state, kw, nargs, args);// 解析这个 SECTION。
                  } else {
                      state.parse_line(&state, nargs, args);
                  }
                  nargs = 0;
              }
              break;
          case T_TEXT:
             ......
              break;
          }
      }
}
```

上面就是 parse_config 函数，代码虽短，实际却比较复杂。从整体来说，parse_config 首先会找到配置文件的一个 section，然后针对不同的 section 使用不同的解析函数来解析。那么，什么是 section 呢？这和 init.rc 文件的组织结构有关。先不必急着去看 init.rc，还是先到代码中去寻找答案。

1. 关键字定义

keywords.h 这个文件定义了 init 中使用的关键字，它的用法很有意思，先来看这个文件，代码如下所示：

[-->keywords.h]

```
#ifndef KEYWORD  // 如果没有定义 KEYWORD 宏，则走下面的分支。
......// 声明一些函数，这些函数就是前面所说的 Action 的执行函数。
int do_class_start(int nargs, char **args);
int do_class_stop(int nargs, char **args);
......
int do_restart(int nargs, char **args);
......
#define __MAKE_KEYWORD_ENUM__  // 定义一个宏。
/*
  定义 KEYWORD 宏，虽然有四个参数，但这里只用第一个，其中 K_##symbol 中的 ## 表示连接
  的意思，即最后得到的值为 K_symbol。symbol 其实就是 init.rc 中的关键字。
```

```
*/
#define KEYWORD(symbol, flags, nargs, func) K_##symbol,
enum { // 定义一个枚举，这个枚举定义了各个关键字的枚举值。
    K_UNKNOWN,
#endif
    ......
    // 根据上面 KEYWORD 的定义，这里将得到一个枚举值 K_class。
     KEYWORD(class,         OPTION,  0, 0)
     KEYWORD(class_start,   COMMAND, 1, do_class_start)//K_class_start
     KEYWORD(class_stop,    COMMAND, 1, do_class_stop)//K_class_stop
     KEYWORD(on,            SECTION, 0, 0)//K_on
     KEYWORD(oneshot,       OPTION,  0, 0)
     KEYWORD(onrestart,     OPTION,  0, 0)
     KEYWORD(restart,       COMMAND, 1, do_restart)
     KEYWORD(service,       SECTION, 0, 0)
     ......
     KEYWORD(socket,        OPTION,  0, 0)
     KEYWORD(start,         COMMAND, 1, do_start)
     KEYWORD(stop,          COMMAND, 1, do_stop)
     ......
#ifdef __MAKE_KEYWORD_ENUM__
    KEYWORD_COUNT,
};
#undef __MAKE_KEYWORD_ENUM__
#undef KEYWORD  // 取消 KEYWORD 宏的定义。
#endif
```

keywords.h 好像没什么奇特之处，它不过是个简单的头文件，为什么说它的用法很有意思呢？接下来看在代码中是如何使用它的，如下所示：

[-->parser.c]

```
......//parser.c 中将包含 keywords.h 头文件，而且还不止一次！！
// 第一次包含 keywords.h，根据 keywords.h 的代码，我们首先会得到一个枚举定义。
#include "keywords.h"
/*
  重新定义 KEYWORD 宏，这回四个参数全用上了，看起来好像是一个结构体。其中 #symbol 表示
  一个字符串，其值为“symbol”。
*/
#define KEYWORD(symbol, flags, nargs, func) \
    [ K_##symbol ] = { #symbol, func, nargs + 1, flags, },

// 定义一个结构体 keyword_info 数组，它用来描述关键字的一些属性，请注意里面的注释内容。
struct {
    const char *name;  // 关键字的名称。
    int (*func)(int nargs, char **args);// 对应关键字的处理函数。
    unsigned char nargs;// 参数个数，每个关键字的参数个数是固定的。
  // 关键字的属性有三种：COMMAND、OPTION 和 SECTION，其中 COMMAND 有对应的处理函数。
    unsigned char flags;
```

```
} keyword_info[KEYWORD_COUNT] = {
[ K_UNKNOWN ] = { "unknown", 0, 0, 0 },
/*
  第二次包含 keywords.h，由于已经重新定了 KEYWORD 宏，所以以前那些作为枚举值的关键字
  现在变成 keyword_info 数组的索引了。
*/
#include "keywords.h"
};
#undef KEYWORD

// 一些辅助宏，帮助我们快速操作 keyword_info 中的内容。
#define kw_is(kw, type) (keyword_info[kw].flags & (type))
#define kw_name(kw) (keyword_info[kw].name)
#define kw_func(kw) (keyword_info[kw].func)
#define kw_nargs(kw) (keyword_info[kw].nargs)
```

现在领略了 keywords.h 的神奇之处了吧？原来它干了两件事情：

- 第一次包含 keywords.h 时，它声明了一些诸如 do_class_start 的函数，另外还定义了一个枚举，枚举值为 K_class，K_mkdir 等关键字。
- 第二次包含 keywords.h 后，得到了一个 keyword_info 结构体数组，这个 keyword_info 结构体数组以前面定义的枚举值为索引，存储对应的关键字信息，这些信息包括关键字名称、处理函数、处理函数的参数个数，以及属性。

目前，关键字信息中最重要的就是 symbol 和 flags 了。什么样的关键字被认为是 section 呢？根据 keywords.h 的定义，当 symbol 为 on 或 service 的时候表示 section：

```
KEYWORD(on,          SECTION, 0, 0)
KEYWORD(service,     SECTION, 0, 0)
```

有了上面的知识，再来看配置文件 init.rc 的内容就比较容易了。

2. 解析 init.rc

init.rc 的内容如下所示（这里只截取了部分内容，注意，其中的注释符号是 #）：

[-->init.rc]

```
on init    # 根据上面的分析可知，on 关键字标示一个 section，对应的名字是 "init"
 ......   # 下面所有的内容都属于这个 section，直到下一个 section 开始时。
 export PATH /sbin:/system/sbin:/system/bin:/system/xbin
 export LD_LIBRARY_PATH /system/lib
 export ANDROID_BOOTLOGO 1 # 根据 keywords.h 的定义可知，export 表示一个 COMMAND。
 export ANDROID_ROOT /system
 export ANDROID_ASSETS /system/app
...... # 省略部分内容
on boot  # 这是一个新的 section，名为 "boot"。
   ifup lo # 这是一个 COMMAND。
    hostname localhost
    domainname localdomain
```

```
    ......
   #class_start 也是一个 COMMAND，对应函数为 do_class_start，很重要，切记。
    class_start default
    ......
    # 下面这个 section 的意思是：待属性 persist.service.adb.enable 的值变为 1 后，
    # 需要执行对应的 COMMAND，这个 COMMAND 是 start adbd。
     on property:persist.service.adb.enable=1
         start adbd  //start 是一个 COMMAND
     on property:persist.service.adb.enable=0
         stop adbd
    ......
#service 也是 section 的标示，对应 section 的名为 "zygote"。
service zygote /system/bin/app_process -Xzygote /system/bin -zygote
 --start-system-server
    socket zygote stream 666  #socket 关键字表示 OPTION。
    onrestart write /sys/android_power/request_state wake #onrestart 也是 OPTION。
    onrestart write /sys/power/state on
    onrestart restart media
# 一个 service（同时也是一个 section），名为 "media"
service media /system/bin/mediaserver
    user media
    group system audio camera graphics inet net_bt net_bt_admin net_raw
ioprio rt 4
```

从上面对 init.rc 的分析可知：

- 一个 section 的内容从这个标识 section 的关键字开始，到下一个标识 section 的地方结束。
- init.rc 中出现了名为 boot 和 init 的 section，这里的 boot 和 init 就是前面介绍的 4 个动作执行阶段中的 boot 和 init。也就是说，在 boot 阶段执行的动作都是由 boot 这个 section 定义的。

另外还可发现，zygote 被放在了一个 service section 中。下面以 zygote 这个 section 为例，介绍 service 是如何解析的。

3.2.2 解析 service

zygote 对应的 service section 内容是：

[-->init.rc::zygote]

```
service zygote /system/bin/app_process -Xzygote /system/bin -zygote \
--start-system-server
    socket zygote stream 666  #socket 是 OPTION。
    # 下面的 onrestart 是 OPTION，而 write 和 restart 是 COMMAND。
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
```

解析 section 的入口函数是 parse_new_section，它的代码如下所示：

[-->parser.c]

```
void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{
    switch(kw) {
    case K_service:  // 用 parse_service 和 parse_line_service 解析 service。
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on: // 解析 on section。
        ..... // 读者可以自己研究。
        break;
    }
    state->parse_line = parse_line_no_op;
}
```

其中，解析 service 时，用到了 parse_service 和 parse_line_service 这两个函数，在分别介绍它们之前，先看 init 是如何组织这个 service 的。

1. service 结构体

init 中使用了一个叫 service 的结构体来保存与 service section 相关的信息，不妨来看看这个结构体，代码如下所示：

[-->init.h::service 结构体定义]

```
struct service {
  //listnode 是一个特殊的结构体，在内核代码中用得非常多，主要用来将结构体链接成一个
  // 双向链表。init 中有一个全局的 service_list，专门用来保存解析配置文件后得到的 service。
    struct listnode slist;
    const char *name; //service 的名字，与我们这个例子对应的就是 "zygote"。
    const char *classname; //service 所属 class 的名字，默认是 "defult"。
    unsigned flags;//service 的属性
    pid_t pid;     // 进程号
    time_t time_started;    // 上一次启动的时间
    time_t time_crashed;  // 上一次死亡的时间
    int nr_crashed;         // 死亡次数
    uid_t uid;      //uid,gid 相关
    gid_t gid;
    gid_t supp_gids[NR_SVC_SUPP_GIDS];
    size_t nr_supp_gids;
   /*
    有些 service 需要使用 socket，下面这个 socketinfo 用来描述 socket 的相关信息。
    我们的 zygote 也使用了 socket，配置文件中的内容是 socket zygote stream 666。
```

```
    它表示将创建一个 AF_STREAM 类型的 socket（其实就是 TCP socket），该 socket 的名为 "zygote"，
    读写权限是 666。
   */
    struct socketinfo *sockets;
    //service 一般运行在一个单独的进程中，envvars 用来描述创建这个进程时所需的环境变量信息。
    struct svcenvinfo *envvars;
   /*
  虽然关键字 onrestart 标识一个 OPTION，可是这个 OPTION 后面一般跟着 COMMAND，
  下面这个 action 结构体可用来存储 command 信息，马上就会分析到它。
  */
    struct action onrestart;

    // 与 keychord 相关的内容
    int *keycodes;
    int nkeycodes;
    int keychord_id;
    //io 优先级设置
    int ioprio_class;
    int ioprio_pri;
    // 参数个数
    int nargs;
    // 用于存储参数
    char *args[1];
};
```

我们现在已了解的 service 的结构体，相对来说还算是清晰易懂的。而 zygote 中的那三个 onrestart 该怎么表示呢？请看 service 中使用的这个 action 结构体：

[-->init.h::action 结构体定义]

```
struct action {
/*
    一个 action 结构体可存放在三个双向链表中，其中 alist 用于存储所有的 action，
    qlist 用于链接那些等待执行的 action，tlist 用于链接那些待某些条件满足后
    就需要执行的 action。
*/
    struct listnode alist;
    struct listnode qlist;
    struct listnode tlist;

    unsigned hash;
    const char *name;

   // 这个 OPTION 对应的是 COMMAND 链表，以 zygote 为例，它有三个 onrestart option，所以
  // 它对应会创建三个 command 结构体。
    struct listnode commands;
    struct command *current;
};
```

了解了上面的知识后，你是否能猜到 parse_service 和 parse_line_service 的作用了呢？马上就来看它们。

2. 了解 parse_service

parse_service 的代码如下所示：

[-->parser.c]

```
static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc; // 声明一个 service 结构体。
    ......
    //init 维护了一个全局的 service 链表，先判断是否已经有同名的 service 了。
    svc = service_find_by_name(args[1]);
    if (svc) {
       ......  // 如果有同名的 service，则不能继续后面的操作。
        return 0;
    }

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    ......
    svc->name = args[1];
    svc->classname = "default";// 设置 classname 为 "default"，这个很关键！
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);
    svc->args[nargs] = 0;
    svc->nargs = nargs;
    svc->onrestart.name = "onrestart";

    list_init(&svc->onrestart.commands);
    // 把 zygote 这个 service 加到全局链表 service_list 中。
    list_add_tail(&service_list, &svc->slist);
    return svc;
}
```

parse_service 函数只是搭建了一个 service 的架子，具体的内容尚需由后面的解析函数来填充。来看 service 的另外一个解析函数 parse_line_service。

3. 了解 parse_line_service

parse_line_service 的代码如下所示：

[-->parser.c]

```
static void parse_line_service(struct parse_state *state, int nargs,
                               char **args)
{
    struct service *svc = state->context;
    struct command *cmd;
    int i, kw, kw_nargs;
```

```
.....
svc->ioprio_class = IoSchedClass_NONE;
// 其实还是根据关键字来做各种处理。
kw = lookup_keyword(args[0]);
switch (kw) {
case K_capability:
    break;
case K_class:
    if (nargs != 2) {
        .....
    } else {
        svc->classname = args[1];
    }
    break;
.....
case K_oneshot:
   /*
      这是 service 的属性，它一共有五个属性，分别为：
      SVC_DISABLED：不随 class 自动启动。下面将会看到 class 的作用。
      SVC_ONESHOT：退出后不需要重启，也就是说这个 service 只启动一次就可以了。
      SVC_RUNNING：正在运行，这是 service 的状态。
      SVC_RESTARTING：等待重启，这也是 service 的状态。
      SVC_CONSOLE：该 service 需要使用控制台。
      SVC_CRITICAL：如果在规定时间内该 service 不断重启，则系统会重启并进入恢复模式。
      zygote 没有使用任何属性，这表明它会随着 class 的处理自动启动；
      退出后会由 init 重启；不使用控制台；即使不断重启也不会导致系统进入恢复模式。
   */
    svc->flags |= SVC_ONESHOT;
    break;
case K_onrestart: // 根据 onrestart 的内容，填充 action 结构体的内容。
    nargs--;
    args++;
    kw = lookup_keyword(args[0]);
    .....
    // 创建 command 结构体。
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    // 把新建的 command 加入到双向链表中。
    list_add_tail(&svc->onrestart.commands, &cmd->clist);
    break;
.....
case K_socket: { // 创建 socket 相关信息。
    struct socketinfo *si;
    .....
    si = calloc(1, sizeof(*si));
    if (!si) {
        parse_error(state, "out of memory\n");
        break;
```

```
            }
            si->name = args[1]; //socket 的名字
            si->type = args[2]; //socket 的类型
            si->perm = strtoul(args[3], 0, 8); //socket 的读写权限
            if (nargs > 4)
                si->uid = decode_uid(args[4]);
            if (nargs > 5)
                si->gid = decode_uid(args[5]);
            si->next = svc->sockets;
            svc->sockets = si;
            break;
        }
        ......
        default:
            parse_error(state, "invalid option '%s'\n", args[0]);
        }
    }
```

parse_line_service 将根据配置文件的内容填充 service 结构体，那么，zygote 解析完后会得到什么呢？图 3-1 表示了 zygote 解析后的结果：

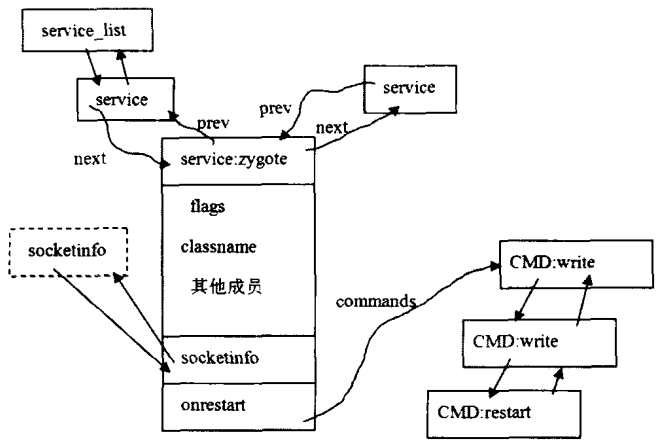


图 3-1　zygote 解析结果示意图

从上图中可知：

- service_list 链表将解析后的 service 全部链接到了一起，并且是一个双向链表，前向节点用 prev 表示，后向节点用 next 表示。
- socketinfo 也是一个双向链表，因为 zygote 只有一个 socket，所以画了一个虚框 socket 作为链表的示范。
- onrestart 通过 commands 指向一个 commands 链表，zygote 有三个 commands。

zygote 这个 service 解析完了，现在就是“万事俱备，只欠东风”了。接下来要了解的是 init 如何控制 service。

3.2.3 init 控制 service

先看 service 是如何启动的。

1. 启动 zygote

init.rc 中有这样一句话：

```
#class_start 是一个 COMMAND，对应的函数为 do_class_start，很重要，切记。
 class_start default
```

class_start 标识一个 COMMAND，对应的处理函数为 do_class_start，它位于 boot section 的范围内。为什么说它很重要呢？

还记得 init 进程中的四个执行阶段吗？当 init 进程执行到下面几句话时，do_class_start 就会被执行了。

```
// 将 boot section 节的 command 加入到执行队列。
action_for_each_trigger("boot", action_add_queue_tail);
// 执行队列里的命令，class 是一个 COMMAND，所以它对应的 do_class_start 会被执行。
drain_action_queue();
```

下面来看 do_class_start 函数：

[-->builtins.c]

```
int do_class_start(int nargs, char **args)
{
    /*
     args 为 do_class_start 的参数，init.rc 中只有一个参数，就是 default。
     下面这个函数将从 service_list 中找到 classname 为 "default" 的 service，然后
     调用 service_start_if_not_disabled 函数。现在读者明白了 service 结构体中
     classname 的作用了吗？
    */
     service_for_each_class(args[1], service_start_if_not_disabled);
     return 0;
}
```

我们已经知道，zygote 这个 service 的 classname 的值就是“default”，所以会针对这个 service 调用 service_start_if_not_disabled，这个函数的代码是：

[-->parser.c]

```
static void service_start_if_not_disabled(struct service *svc)
{
if (!(svc->flags & SVC_DISABLED)) {
     service_start(svc, NULL); //zygote 可不设置 SVC_DISABLED
 }
}
```

service_start 函数的代码如下所示：

[-->init.c]

```
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid_t pid;
    int needs_console;
    int n;

    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING));
    svc->time_started = 0;

    if (svc->flags & SVC_RUNNING) {
        return;// 如果这个 service 已经在运行，则不用处理
    }
    /*
    service 一般运行于另外一个进程中，这个进程也是 init 的子进程，所以启动 service 前需要判断
    对应的可执行文件是否存在，zygote 对应的可执行文件是 /system/bin/app_process。
    */
    if (stat(svc->args[0], &s) != 0) {
       svc->flags |= SVC_DISABLED;
        return;
    }
    ......
    pid = fork(); // 调用 fork 创建子进程
    if (pid == 0) {
        //pid 为零，表示现在运行在子进程中
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        // 得到属性存储空间的信息并加到环境变量中，后面在属性服务一节中会碰到使用它的地方。
        get_property_workspace(&fd, &sz);
        add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        // 添加环境变量信息。
        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);
        // 根据 socketinfo 创建 socket。
        for (si = svc->sockets; si; si = si->next) {
            int s = create_socket(si->name,
                                  !strcmp(si->type, "dgram") ?
                                  SOCK_DGRAM : SOCK_STREAM,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                // 在环境变量中添加 socket 信息。
                publish_socket(si->name, s);
            }
```

```
        }
        ......// 设置 uid, gid 等
        setpgid(0, getpid());
        if (!dynamic_args) {
         /*
         执行 /system/bin/app_process，这样就进入到 app_process 的 main 函数中了。
         fork、execve 这两个函数都是 Linux 系统上常用的系统调用。
         */
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
               ......
            }
        } else {
           ......
    }

    ......// 父进程 init 的处理，设置 service 的信息，如启动时间、进程号，以及状态等。
     svc->time_started = gettime();
     svc->pid = pid;
     svc->flags |= SVC_RUNNING;
     // 每一个 service 都有一个属性，zygote 的属性为 init.svc.zygote，现在设置它的值为 running。
     notify_service_state(svc->name, "running");
}
```

原来，zygote 是通过 fork 和 execv 共同创建的！但 service 结构中的那个 onrestart 好像没有派上用场，原因何在？

2. 重启 zygote

应该根据名字就可猜到 onrestart 是在 zygote 重启时用的。下面先看在 zygote 死后，它的父进程 init 会有什么动作：

[-->init.c]

```
static void sigchld_handler(int s)
{   // 当子进程退出时，init 的这个信号处理函数会被调用。
    write(signal_fd, &s, 1); // 往 signal_fd 中写数据。
}
```

signal_fd 就是在 init 中通过 socketpair 创建的两个 socket 中的一个，既然会往这个 signal_fd 中发送数据，那么另外一个 socket 就一定能接收到，这样就会导致 init 从 poll 函数中返回，代码如下所示：

[-->init.rc::main 函数代码片断]

```
nr = poll(ufds, fd_count, timeout);
......
if (ufds[2].revents == POLLIN) {
   read(signal_recv_fd, tmp, sizeof(tmp));
       while (!wait_for_one_process(0))// 调用 wait_for_one_process 函数处理。
```

```
        ;
        continue;
 }
 ......
// 直接看这个 wait_for_one_process 函数:
static int wait_for_one_process(int block)
{
    pid_t pid;
    int status;
    struct service *svc;
    struct socketinfo *si;
    time_t now;
    struct listnode *node;
    struct command *cmd;

    while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 &&
                  errno == EINTR );
    if (pid <= 0) return -1;
    // 找到死掉的那个 service，现在应该找到了代表 zygote 的那个 service。
    svc = service_find_by_pid(pid);
   ......

    if (!(svc->flags & SVC_ONESHOT)) {
        // 杀掉 zygote 创建的所有子进程，这就是 zygote 死后，Java 世界崩溃的原因。
        kill(-pid, SIGKILL);
   }

    // 清理 socket 信息，不清楚的读者可以通过命令 man 7 AF_UNIX 查询一下相关知识。
    for (si = svc->sockets; si; si = si->next) {
        char tmp[128];
        snprintf(tmp, sizeof(tmp), ANDROID_SOCKET_DIR"/%s", si->name);
        unlink(tmp);
    }

    svc->pid = 0;
    svc->flags &= (~SVC_RUNNING);

   if (svc->flags & SVC_ONESHOT) {
        svc->flags |= SVC_DISABLED;
    }
   ......
    now = gettime();
    /*
     如果设置了 SVC_CRITICAL 标识，则 4 分钟内该服务重启的次数不能超过 4 次，否则
     机器会在重启时进入 recovery 模式。根据 init.rc 的配置来看，只有 servicemanager 进程
     享有此种待遇。
    */
    if (svc->flags & SVC_CRITICAL) {
        if (svc->time_crashed + CRITICAL_CRASH_WINDOW >= now) {
            if (++svc->nr_crashed > CRITICAL_CRASH_THRESHOLD) {
```

```
                ......
                sync();
                __reboot(LINUX_REBOOT_MAGIC1, LINUX_REBOOT_MAGIC2,
                        LINUX_REBOOT_CMD_RESTART2, "recovery");
                return 0;
            }
        } else {
            svc->time_crashed = now;
            svc->nr_crashed = 1;
        }
    }

    svc->flags |= SVC_RESTARTING;
    // 设置标识为 SVC_RESTARTING，然后执行该 service onrestart 中的 COMMAND，这些内容就
    // 非常简单了，读者可以自行学习。
    list_for_each(node, &svc->onrestart.commands) {
        cmd = node_to_item(node, struct command, clist);
        cmd->func(cmd->nargs, cmd->args);
    }
    // 设置 init.svc.zygote 的值为 restarting。
    notify_service_state(svc->name, "restarting");
    return 0;
}
```

通过上面的代码可以知道 onrestart 的作用了，但 zygote 本身又在哪里重启呢？答案就在下面的代码中：

[-->init.c::main 函数代码片断]

```
for(;;) {
        int nr, i, timeout = -1;
        for (i = 0; i < fd_count; i++)
            ufds[i].revents = 0;
        drain_action_queue(); //poll 函数返回后，会进入下一轮的循环
        restart_processes(); // 这里会重启所有 flag 标志为 SVC_RESTARTING 的 service。
        ......
}
```

这样，zygote 又回来了！

3.2.4 属性服务

我们知道，Windows 平台上有一个叫注册表的东西。注册表可以存储一些类似 key/value 的键值对。一般而言，系统或某些应用程序会把自己的一些属性存储在注册表中，即使系统重启或应用程序重启，它还能够根据之前在注册表中设置的属性，进行相应的初始化工作。Android 平台也提供了一个类似的机制，称之为属性服务（property service）。应用程序可通过这个属性机制，查询或设置属性。读者可以用 adb shell 登录到真机或模拟器上，然后用 getprop 命令查看当前系统中有哪些属性。比如我的 HTC G7 测试结果，如图 3-2 所示（图中

只显示了部分属性）。

```
[dhcp.eth0.ipaddress]: [192.168.1.103]
[dhcp.eth0.gateway]: [192.168.1.1]
[dhcp.eth0.mask]: [255.255.255.0]
[dhcp.eth0.leasetime]: [7200]
[dhcp.eth0.server]: [192.168.1.1]
[dhcp.eth0.leasedfrom]: [1305379532]
[net.dns1]: [219.239.26.42]
[net.dns2]: [124.207.160.110]
[net.rmnet0.dns1]: [0.0.0.0]
[net.rmnet0.dns2]: [0.0.0.0]
[net.rmnet0.gw]: [10.141.18.133]
[ro.ril.current.ip.address]: [10.143.228.94]
[net.rmnet0.proxy]: [10.0.0.172]
[net.rmnet0.mms_proxy]: [10.0.0.172]
[net.gprs.mms.ru.number]: [0]
```

图 3-2　HTC G7 属性示意图

这个属性服务是怎么实现的呢？下面来看代码，其中与 init.c 和属性服务有关的代码有下面两行：

```
property_init();
property_set_fd = start_property_service();
```

分别来看看它们。

1. 属性服务初始化

（1）创建存储空间

先看 property_init 函数，代码如下所示：

[-->property_service.c]

```
void property_init(void)
{
    init_property_area(); //初始化属性存储区域。
    //加载 default.prop 文件。
    load_properties_from_file(PROP_PATH_RAMDISK_DEFAULT);
}
```

在 properyty_init 函数中，先调用 init_property_area 函数，创建一块用于存储属性的存储区域，然后加载 default.prop 文件中的内容。再看 init_property_area 是如何工作的，它的代码如下所示：

[-->property_service.c]

```
static int init_property_area(void)
{
    prop_area *pa;

    if(pa_info_array)
        return -1;
```

```
    /*
    初始化存储空间，PA_SIZE 是这块存储空间的总大小，为 32768 字节，pa_workspace
    为 workspace 类型的结构体，下面是它的定义：
    typedef struct {
        void *data;    // 存储空间的起始地址
        size_t size;   // 存储空间的大小
        int fd;    // 共享内存的文件描述符
    } workspace;
    init_workspace 函数调用 Android 系统提供的 ashmem_create_region 函数创建一块
    共享内存。关于共享内存的知识我们在第 7 章会接触，这里只需把它当做一块普通的内存就
    可以了。
    */
    if(init_workspace(&pa_workspace, PA_SIZE))
        return -1;

    fcntl(pa_workspace.fd, F_SETFD, FD_CLOEXEC);

    // 在 32768 个字节的存储空间中，有 PA_INFO_START（1024）个字节用来存储头部信息。
    pa_info_array = (void*) (((char*) pa_workspace.data) + PA_INFO_START);

    pa = pa_workspace.data;
    memset(pa, 0, PA_SIZE);
    pa->magic = PROP_AREA_MAGIC;
    pa->version = PROP_AREA_VERSION;
       //__system_property_area__ 这个变量由 bionic libc 库输出，有什么用呢？
       __system_property_area__ = pa;

    return 0;
}
```

上面的内容比较简单，不过最后的赋值语句可是大有来头。__system_property_area__ 是 bionic libc 库中输出的一个变量，为什么这里要给它赋值呢？

原来，虽然属性区域是由 init 进程创建的，但 Android 系统希望其他进程也能读取这块内存里的东西。为了做到这一点，它便做了以下两项工作：

- 把属性区域创建在共享内存上，而共享内存是可以跨进程的。这一点，已经在上面的代码中见到了，init_workspace 函数内部将创建这个共享内存。
- 如何让其他进程知道这个共享内存呢？Android 利用了 gcc 的 constructor 属性，这个属性指明了一个 __libc_prenit 函数，当 bionic libc 库被加载时，将自动调用这个 __libc_prenit，这个函数内部就将完成共享内存到本地进程的映射工作。

（2）客户端进程获取存储空间

关于上面的内容，来看相关代码：

[-->libc_init_dynamic.c]

```
//constructor 属性指示加载器加载该库后，首先调用 __libc_prenit 函数。这一点和 Windows 上
// 动态库的 DllMain 函数类似。
```

```
void __attribute__((constructor)) __libc_prenit(void);
void __libc_prenit(void)
{
    ......
      __libc_init_common(elfdata); // 调用这个函数。
     ......
}
```

__libc_init_common 函数为：

[-->libc_init_common.c]

```
void __libc_init_common(uintptr_t *elfdata)
{
   ......
    __system_properties_init();// 初始化客户端的属性存储区域。
}
```

[-->system_properties.c]

```
int __system_properties_init(void)
{
    prop_area *pa;
    int s, fd;
    unsigned sz;
    char *env;

    .....
    // 还记得在“启动 zygote”一节中提到的添加环境变量的地方吗？属性存储区域的相关信息
    // 就是在那儿添加的，这里需要取出来使用了。
    env = getenv("ANDROID_PROPERTY_WORKSPACE");
    // 取出属性存储区域的文件描述符。关于共享内存的知识，第 7 章中将会进行介绍。
    fd = atoi(env);
    env = strchr(env, ',');
    if (!env) {
        return -1;
    }
    sz = atoi(env + 1);
    // 映射 init 创建的那块内存到本地进程空间，这样本地进程就可以使用这块共享内存了。
    // 注意，映射的时候指定了 PROT_READ 属性，所以客户端进程只能读属性，而不能设置属性。
    pa = mmap(0, sz, PROT_READ, MAP_SHARED, fd, 0);

    if(pa == MAP_FAILED) {
        return -1;
    }

    if((pa->magic != PROP_AREA_MAGIC) || (pa->version != PROP_AREA_VERSION)) {
        munmap(pa, sz);
        return -1;
    }

    __system_property_area__ = pa;
```

```
    return 0;
}
```

上面的代码中有很多地方与共享内存有关，在第 7 章中会对与共享内存有关的问题进行介绍，大家也可先行学习有关共享内存的知识。

总之，通过这种方式，客户端进程可以直接读取属性空间，但没有权限设置属性。客户端进程又是如何设置属性的呢？

2. 属性服务器的分析

（1）启动属性服务器

init 进程会启动一个属性服务器，而客户端只能通过与属性服务器交互来设置属性。先来看属性服务器的内容，它由 start_property_service 函数启动，代码如下所示：

[-->Property_servie.c]

```
int start_property_service(void)
{
    int fd;

   /*
       加载属性文件，其实就是解析这些文件中的属性，然后把它设置到属性空间中去。Android 系统
      一共提供了四个存储属性的文件，它们分别是：
     #define PROP_PATH_RAMDISK_DEFAULT  "/default.prop"
     #define PROP_PATH_SYSTEM_BUILD     "/system/build.prop"
     #define PROP_PATH_SYSTEM_DEFAULT   "/system/default.prop"
     #define PROP_PATH_LOCAL_OVERRIDE   "/data/local.prop"
    */

    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_properties_from_file(PROP_PATH_LOCAL_OVERRIDE);
    // 有一些属性是需要保存到永久介质上的，这些属性文件则由下面这个函数加载，这些文件
    // 存储在 /data/property 目录下，并且这些文件的文件名必须以 persist. 开头。这个函数
    // 很简单，读者可自行研究。
    load_persistent_properties();
    // 创建一个 socket，用于 IPC 通信。
    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return -1;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);
    listen(fd, 8);
    return fd;
}
```

属性服务创建了一个用来接收请求的 socket，可这个请求在哪里被处理呢？事实上，在 init 中的 for 循环处已经进行相关处理了。

（2）处理设置属性请求

接收请求的地方在 init 进程中，代码如下所示：

[-->init.c::main 函数片断]

```
if (ufds[1].revents == POLLIN)
            handle_property_set_fd(property_set_fd);
```

当属性服务器收到客户端请求时，init 会调用 handle_property_set_fd 进行处理。这个函数的代码如下所示：

[-->property_service.c]

```
void handle_property_set_fd(int fd)
{
    prop_msg msg;
    int s;
    int r;
    int res;
    struct ucred cr;
    struct sockaddr_un addr;
    socklen_t addr_size = sizeof(addr);
    socklen_t cr_size = sizeof(cr);
    // 先接收 TCP 连接。
    if ((s = accept(fd, (struct sockaddr *) &addr, &addr_size)) < 0) {
        return;
    }

    // 取出客户端进程的权限等属性。
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
        ......
        return;
    }
   // 接收请求数据。
    r = recv(s, &msg, sizeof(msg), 0);
    close(s);
    ......

    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        msg.name[PROP_NAME_MAX-1] = 0;
        msg.value[PROP_VALUE_MAX-1] = 0;
        /*
            如果是 ctl 开头的消息，则认为是控制消息，控制消息用来执行一些命令，例如用
            adb shell 登录后，输入 setprop ctl.start bootanim 就可以查看开机动画了，
            如果要关闭就输入 setprop ctl.stop bootanim，是不是很有意思呢？
         */
        if(memcmp(msg.name,"ctl.",4) == 0) {
            if (check_control_perms(msg.value, cr.uid, cr.gid)) {
```

```
                handle_control_message((char*) msg.name + 4, (char*) msg.value);
            }
            ......
        } else {
            //检查客户端进程是否有足够的权限。
            if (check_perms(msg.name, cr.uid, cr.gid)) {
                //然后调用 property_set 设置。
                property_set((char*) msg.name, (char*) msg.value);
            }
            ......
        }
        break;

    default:
        break;
    }
}
```

当客户端的权限满足要求时，init 就调用 property_set 进行相关处理，这个函数比较简单，代码如下所示：

[-->property_service.c]

```
int property_set(const char *name, const char *value)
{
    prop_area *pa;
    prop_info *pi;

    int namelen = strlen(name);
    int valuelen = strlen(value);
    ......
    //从属性存储空间中寻找是否已经存在该属性。
    pi = (prop_info*) __system_property_find(name);

    if(pi != 0) {
        //如果属性名以 ro. 开头，则表示是只读的，不能设置，所以直接返回。
        if(!strncmp(name, "ro.", 3)) return -1;

        pa = __system_property_area__;
        //更新该属性的值。
        update_prop_info(pi, value, valuelen);
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    } else {
       //如果没有找到对应的属性，则认为是增加属性，所以需要新创建一项。注意，Android 最多支持
       //247 项属性，如果目前属性的存储空间中已经有 247 项，则直接返回。
        pa = __system_property_area__;
        if(pa->count == PA_COUNT_MAX) return -1;
```

```
        pi = pa_info_array + pa->count;
        pi->serial = (valuelen << 24);
        memcpy(pi->name, name, namelen + 1);
        memcpy(pi->value, value, valuelen + 1);

        pa->toc[pa->count] =
            (namelen << 24) | (((unsigned) pi) - ((unsigned) pa));

        pa->count++;
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    }
    // 有一些特殊的属性需要特殊处理，这里主要是以 net.change 开头的属性。
    if (strncmp("net.", name, strlen("net.")) == 0)  {
        if (strcmp("net.change", name) == 0) {
            return 0;
        }
        property_set("net.change", name);
    } else if (persistent_properties_loaded &&
       strncmp("persist.", name, strlen("persist.")) == 0) {
       // 如果属性名以 persist. 开头，则需要把这些值写到对应的文件中去。
       write_persistent_property(name, value);
    }
    /*
    还记得 init.rc 中的下面这句话吗?
    on property:persist.service.adb.enable=1
         start adbd
    待 persist.service.adb.enable 属性置为 1 后，就会执行 start adbd 这个 command,
    这是通过 property_changed 函数来完成的，它非常简单，读者可以自己阅读。
    */
    property_changed(name, value);
    return 0;
}
```

好，属性服务端的工作已经了解了，下面看客户端是如何设置属性的。

（3）客户端发送请求

客户端通过 property_set 发送请求，property_set 由 libcutils 库提供，代码如下所示：

[-->properties.c]

```
int property_set(const char *key, const char *value)
{
    prop_msg msg;
    unsigned resp;

   ......
    msg.cmd = PROP_MSG_SETPROP;// 设置消息码为 PROP_MSG_SETPROP。
    strcpy((char*) msg.name, key);
    strcpy((char*) msg.value, value);
```

```
    // 发送请求。
    return send_prop_msg(&msg);
}

static int send_prop_msg(prop_msg *msg)
{
    int s;
    int r;
    // 建立和属性服务器的 socket 连接。
    s = socket_local_client(PROP_SERVICE_NAME,
                        ANDROID_SOCKET_NAMESPACE_RESERVED,
                        SOCK_STREAM);
    if(s < 0) return -1;
    // 通过 socket 发送出去。
    while((r = send(s, msg, sizeof(prop_msg), 0)) < 0) {
        if((errno == EINTR) || (errno == EAGAIN)) continue;
        break;
    }

    if(r == sizeof(prop_msg)) {
        r = 0;
    } else {
        r = -1;
    }

    close(s);
    return r;
}
```

至此，属性服务器就介绍完了。总体来说，还算比较简单。

3.3 本章小结

本章讲解了 init 进程如何解析 zygote，以及属性服务器的工作原理，旨在帮助读者认识这个天字号第一进程。相对来说，init.rc 的解析难度最大。相信读者通过以上的实例分析，已经理解了 init.rc 的解析原理。另外，init 涉及很多和 Linux 系统相关的知识，有兴趣的读者可以自行研究。



第4章 深入理解 zygote

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。

- App_main.cpp(*framework/base/cmds/app_process/App_main.cpp*)
- AndroidRuntime.h(*framework/base/include/android_runtime/AndroidRuntime.h*)
- android_debug_JNITest.cpp(*framework/base/core/jni/android_debug_JNITest.cpp*)
- ZygoteInit.java(*framework/base/core/java/com/android/internal/os/ZygoteInit.java*)
- dalvik_system_Zygote.c(*dalvik/vm/native/dalvik_system_Zygote.c*)
- RuntimeInit.java(*framework/base/core/java/com/android/internal/os/RuntimeInit.java*)
- SystemServer.java(*framework/base/services/java/com/android/server/SystemServer.java*)
- com_android_server_SystemServer.cpp(*framework/base/services/jni/com_android_server_SystemServer.cpp*)
- system_init.cpp(*framework/base/cmds/system_server/library/system_init.cpp*)
- Watchdog.java(*framework/base/services/java/com/android/server/Watchdog.java*)
- ActivityManagerService.java(*framework/base/services/java/com/android/server/am/ActivityManagerService.java*)
- Process.java(*framework/base/core/java/android/os/Process.java*)
- ZygoteConnection.java(*framework/base/core/java/com/android/internal/os/ ZygoteConnection.java*)

4.1 概述

读者可能已经知道，Android 系统存在着两个完全不同的世界：

- Java 世界，Google 提供的 SDK 主要就是针对这个世界的。在这个世界中运行的程序都是基于 Dalvik 虚拟机的 Java 程序。
- Native 世界，也就是用 Native 语言 C 或 C++ 开发的程序，它们组成了 Native 世界。

初次接触 Android 的人可能会有如下几个疑问：

- Android 是基于 Linux 内核构建的，那么最早存在的肯定是 Native 世界，可 Java 世界是什么时候创建的呢？
- 我们都知道，程序运行时一定要有一个进程，但是我们在编写 Activity、Service 的时候却极少接触到“进程”这一概念（这是 Google 有意为之），但这些 Activity 或 Service 却又不能脱离“进程”而存在。那么，这个“进程”是怎么创建和运行的呢？这是一个值得琢磨的问题。
- 在程序中，我们经常使用系统的 Service，那么，这些 Service 在哪里呢？

这些问题的答案都与我们本章的两位主人公 zygote 和 system_server 有关。zygote 这个词的中文意思是“受精卵”，它和 Android 系统中的 Java 世界有着重要关系。而 system_server 则“人如其名”，系统中重要的 Service 都驻留于 Java 世界中。

zygote 和 system_server 这两个进程分别是 Java 世界的半边天，任何一个进程的死亡，都会导致 Java 世界崩溃，够厉害吧？下面我们就来见识见识这两个重量级人物。

4.2 zygote 分析

zygote 本身是一个 Native 的应用程序，与驱动、内核等均无关系。根据第 3 章对 init 的介绍我们可以知道，zygote 是由 init 进程根据 init.rc 文件中的配置项创建的。在分析它之前，我们有必要先简单介绍一下“zygote”这个名字的来历。zygote 最初的名字叫“app_process”，这个名字是在 Android.mk 文件中指定的，但在运行过程中，app_process 通过 Linux 下的 pctrl 系统调用将自己的名字换成了“zygote”，所以我们通过 ps 命令看到的进程名是“zygote”。

zygote 玩的这一套“换名把戏”并不影响我们的分析，它的原型 app_process 所对应的源文件是 App_main.cpp，代码如下所示：

[-->App_main.cpp]

```
int main(int argc, const char* const argv[])
{
  /*
   zygote 进程由 init 通过 fork 而来，我们回顾一下 init.rc 中设置的启动参数：
    -Xzygote /system/bin --zygote --start-system-server
  */
```

```
        mArgC = argc;
        mArgV = argv;

        mArgLen = 0;
        for (int i=0; i<argc; i++) {
            mArgLen += strlen(argv[i]) + 1;
        }
        mArgLen--;
        AppRuntime runtime;
        // 调用 Appruntime 的 addVmArguments，这个函数很简单，读者可以自行分析。
        int i = runtime.addVmArguments(argc, argv);
        if (i < argc) {
           // 设置 runtime 的 mParentDir 为 /system/bin。
            runtime.mParentDir = argv[i++];
        }

        if (i < argc) {
            arg = argv[i++];
            if (0 == strcmp("--zygote", arg)) {
               // 我们传入的参数满足 if 的条件，而且下面的 startSystemServer 的值为 true。
                bool startSystemServer = (i < argc) ?
                        strcmp(argv[i], "--start-system-server") == 0 : false;
                setArgv0(argv0, "zygote");
                // 设置本进程的名称为 zygote，这正是前文所讲的“换名把戏”。
                set_process_name("zygote");
                // ①调用 runtime 的 start，注意第二个参数 startSystemServer 为 true。
                runtime.start("com.android.internal.os.ZygoteInit",
                              startSystemServer);
            }
             ......
        }
             ......
    }
```

zygote 的这个 main 函数虽然很简单，但其重要功能却是由 AppRuntime 的 start 来完成的。下面，我们就来具体分析这个 AppRuntime。

4.2.1　AppRuntime 分析

AppRuntime 类的声明和实现均在 App_main.cpp 中，它是从 AndroidRuntime 类派生出来的，图 4-1 显示了这两个类的关系和一些重要函数。

由图 4-1 我们可知：

AppRuntime 重载了 onStarted、onZygoteInit 和 onExit 函数。

前面的代码调用了 AndroidRuntime 的 start 函数，由图 4-1 可知，这个 start 函数使用的是基类 AndroidRuntime 的 start，我们来分析一下它，注意它的调用参数。

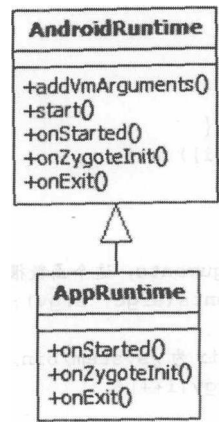


图 4-1　AppRuntime 和 AndroidRuntime 的关系

[-->AndroidRuntime.cpp]

```
void AndroidRuntime::start(const char* className, const bool startSystemServer)
{
    //className 的值是 "com.android.internal.os.ZygoteInit"。
    //startSystemServer 的值是 true。
    char* slashClassName = NULL;
    char* cp;
    JNIEnv* env;
    blockSigpipe();// 处理 SIGPIPE 信号。
     ......

    const char* rootDir = getenv("ANDROID_ROOT");
    if (rootDir == NULL) {
    // 如果环境变量中没有 ANDROID_ROOT，则新增该变量，并设置值为 “/system"。
        rootDir = “/system";
        ......
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //① 创建虚拟机
    if (startVm(&mJavaVM, &env) != 0)
        goto bail;

     //② 注册 JNI 函数
    if (startReg(env) < 0) {
         goto bail;
     }

    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
```

```
    jstring startSystemServerStr;

    stringClass = env->FindClass("java/lang/String");
    // 创建一个有两个元素的 String 数组，即 Java 代码 String strArray[] = new String[2]。
    strArray = env->NewObjectArray(2, stringClass, NULL);

   classNameStr = env->NewStringUTF(className);
   // 设置第一个元素为 "com.android.internal.os.ZygoteInit"。
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    startSystemServerStr = env->NewStringUTF(startSystemServer ? "true" : "false");
   // 设置第二个元素为 "true"，注意这两个元素都是 String 类型，即字符串。
    env->SetObjectArrayElement(strArray, 1, startSystemServerStr);

    jclass startClass;
    jmethodID startMeth;

    slashClassName = strdup(className);
   /*
     将字符串 “com.android.internal.os.ZygoteInit” 中的 “.” 换成 “/”，
     这样就变成了 “com/android/internal/os/ZygoteInit”，这个名字符合 JNI 规范，
     我们可将其简称为 ZygoteInit 类。
   */
    for (cp = slashClassName; *cp != '\0'; cp++)
        if (*cp == '.')
            *cp = '/';

    startClass = env->FindClass(slashClassName);
     ......
    // 找到 ZygoteInit 类的 static main 函数的 jMethodId。
    startMeth = env->GetStaticMethodID(startClass, "main",
                                       "([Ljava/lang/String;)V");
      ......
     /*
        ③通过 JNI 调用 Java 函数，注意调用的函数是 main，所属的类是
          com.android.internal.os.ZygoteInit，传递的参数是
          “com.android.internal.os.ZygoteInit true”，
          在调用 ZygoteInit 的 main 函数后，zygote 便进入了 Java 世界！
          也就是说，zygote 是开创 Android 系统中 Java 世界的盘古。
     */
      env->CallStaticVoidMethod(startClass, startMeth, strArray);
    //zygote 退出，在正常情况下，zygote 不需要退出。
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        LOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        LOGW("Warning: VM did not shut down cleanly\n");

bail:
    free(slashClassName);
}
```

通过上面的分析，我们找到了三个关键点（见代码中的①、②、③），它们共同组成了开创 Android 系统中 Java 世界的三部曲。现在让我们来具体地观察它们。

1. 创建虚拟机——startVm

我们先看三部曲中的第一部：startVm，这个函数没有特别之处，就是调用 JNI 的虚拟机创建函数，但是创建虚拟机时的一些参数却是在 startVm 中确定的，其代码如下所示：

[-->AndroidRuntime.cpp]

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
    // 这个函数绝大部分代码都是设置虚拟机的参数，我们只分析其中的两个。
   /*
     下面的代码是用来设置 JNI check 选项的。JNI check 指的是 Native 层调用 JNI 函数时，
     系统所做的一些检查工作。例如，调用 NewUTFString 函数时，系统会检查传入的字符串是不是符合
      UTF-8 的要求。JNI check 还能检查资源是否被正确释放。但这个选项也有副作用，比如：
      1）因为检查工作比较耗时，所以会影响系统运行速度。
      2）有些检查过于严格，例如上面的字符串检查，一旦出错，则调用进程就会 abort。
      所以，JNI check 选项一般只在调试的 eng 版设置，在正式发布的 user 版中则不设置该选项了。
      下面这几句代码就控制着是否启用 JNI check，这是由系统属性决定的，eng 版如经过特殊配置，也可
      以去掉 JNI check。
    */
    property_get("dalvik.vm.checkjni", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkJni = true;
    } else if (strcmp(propBuf, "false") != 0) {
       property_get("ro.kernel.android.checkjni", propBuf, "");
        if (propBuf[0] == '1') {
            checkJni = true;
        }
        }
    ......
   /*
     设置虚拟机的 heapsize，默认为 16MB。绝大多数厂商都会修改这个值，一般是 32MB。
     heapsize 不能设置得过小，否则在操作大尺寸的图片时无法分配所需内存。
     这里有一个问题，即 heapsize 既然是系统级的属性，那么能否根据不同应用程序的需求来进行动
     态调整呢？我开始也考虑过能否实现这一构想，不过希望很快就破灭了。对这一问题，我们将在本章的
     拓展内容中深入讨论。
    */
    strcpy(heapsizeOptsBuf, "-Xmx");
    property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
    opt.optionString = heapsizeOptsBuf;
    mOptions.add(opt);

    if (checkJni) {
        opt.optionString = "-Xcheck:jni";
        mOptions.add(opt);
        //JNI check 中的资源检查，系统中创建的 Global reference 个数不能超过 2000。
        opt.optionString = "-Xjnigreflimit:2000";
```

```
        mOptions.add(opt);
    }
    // 调用 JNI_CreateJavaVM 创建虚拟机，pEnv 返回当前线程的 JNIEnv 变量。
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        LOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }

    result = 0;

bail:
    free(stackTraceFile);
    return result;
}
```

关于 dalvik 虚拟机的详细参数，读者可以参见 Dalvik/Docs/Dexopt.html 中的说明。这个 Docs 目录下的内容，或许可帮助我们更深入地了解 dalvik 虚拟机。

2. 注册 JNI 函数——startReg

前面已经介绍了如何创建虚拟机，下一步则需要给这个虚拟机注册一些 JNI 函数。正是因为后续 Java 世界用到的一些函数是采用 native 方式实现的，所以才必须提前注册这些函数。

下面我们来看看这个 startReg 函数，代码如下所示：

[-->AndroidRuntime.cpp]

```
int AndroidRuntime::startReg(JNIEnv* env)
{
 // 注意，设置 Thread 类的线程创建函数为 javaCreateThreadEtc。
 // 它的作用将在对 Thread 进行分析时（第 5 章）详细介绍。
 androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

  env->PushLocalFrame(200);
  // 注册 JNI 函数，gRegJNI 是一个全局数组。
  if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);
    // 下面这句话应当是“码农”休闲时的小把戏。在日新月异的 IT 世界中，它现在绝对是“文物”了。
    //createJavaThread("fubar", quickTest, (void*) "hello");
    return 0;
}
```

我们来看看 register_jni_procs，代码如下所示：

[-->AndroidRuntime.cpp]

```
static int register_jni_procs(const RegJNIRec array[], size_t count, JNIEnv* env)
```

```
{
    for (size_t i = 0; i < count; i++) {
        if (array[i].mProc(env) < 0) {// 仅仅是一个封装，调用数组元素的 mProc 函数
        return -1;
        }
        }
        return 0;
}
```

上面的函数调用的不过是数组元素的 mProc 函数，让我们再直接看看这个全局数组的 gRegJNI 变量。

[-->AndroidRuntime.cpp::gRegJNI 声明]

```
static const RegJNIRec gRegJNI[] = {
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
     ...// 共有 100 项
};
```

REG_JNI 是一个宏，宏里面包括的就是那个 mProc 函数，这里我们来分析一个例子。

[-->android_debug_JNITest.cpp]

```
int register_android_debug_JNITest(JNIEnv* env)
{
   // 为 android.debug.JNITest 类注册它所需要的 JNI 函数。
   return jniRegisterNativeMethods(env, "android/debug/JNITest",
                                        gMethods, NELEM(gMethods));
}
```

哦，原来 mProc 就是为 Java 类注册了 JNI 函数！

至此，虚拟机已创建好，JNI 函数也已注册，下一步就要分析 CallStaticVoidMethod 了。通过这个函数，我们将进入 Android 精心打造的 Java 世界，而且最佳的情况是，永远也不回到 Native 世界。

4.2.2 Welcome to Java World

这个 Java 世界的入口在哪里？根据前面的分析可知，CallStaticVoidMethod 最终将调用 com.android.internal.os.ZygoteInit 的 main 函数，下面就来看看这个入口函数。代码如下所示：

[-->ZygoteInit.java]

```
public static void main(String argv[]) {
```

```
try {
        SamplingProfilerIntegration.start();
        //①注册 zygote 用的 socket。
        registerZygoteSocket();
        //②预加载类和资源。
        preloadClasses();
        preloadResources();
        ......
        // 强制执行一次垃圾收集。
        gc();
        // 我们传入的参数满足 if 分支。
        if (argv[1].equals("true")) {
            startSystemServer();//③启动 system_server 进程。
        } else if (!argv[1].equals("false")) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }
        // ZYGOTE_FORK_MODE 被定义为 false，所以满足 else 的条件。
        if (ZYGOTE_FORK_MODE) {
            runForkMode();
        } else {
            runSelectLoopMode();// ④ zygote 调用这个函数。
        }
        closeServerSocket(); // 关闭 socket。
        } catch (MethodAndArgsCaller caller) {
            caller.run();//⑤很重要的 caller run 函数，以后分析。
        } catch (RuntimeException ex) {
            closeServerSocket();
            throw ex;
        }
        ......
    }
```

在 ZygoteInit 的 main 函数中，我们列举出了 5 大关键点（即代码中的①～⑤），下面对其一一进行分析。先看第一点：registerZygoteSocket。

1. 建立 IPC 通信服务端——registerZygoteSocket

zygote 及系统中其他程序的通信没有使用 Binder，而是采用了基于 AF_UNIX 类型的 Socket。registerZygoteSocket 函数的使命正是建立这个 Socket。代码如下所示：

[-->ZygoteInit.java]

```
private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
        try {
            // 从环境变量中获取 Socket 的 fd，还记得第 3 章中介绍的 zygote 是如何启动的吗？
            // 这个环境变量由 execv 传入。
```

```
            String env = System.getenv(ANDROID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
        }
        try {
          // 创建服务端 Socket，这个 Socket 将 listen 并 accept Client。
          sServerSocket = new LocalServerSocket(createFileDescriptor(fileDesc));
        }
        }
}
```

registerZygoteSocket 很简单，就是创建一个服务端的 Socket。不过读者应该提前想到下面两个问题：

- 谁是客户端？
- 服务端会怎么处理客户端的消息？

提示 读者应掌握与 Socket 相关的知识，这些知识对网络编程或简单的 IPC 使用是会有帮助的。

2. 预加载类和资源

现在我们要分析的就是 preloadClasses 和 preloadResources 函数了。先来看看 preloadClasses。

[-->ZygoteInit.java]

```
private static void preloadClasses() {
     final VMRuntime runtime = VMRuntime.getRuntime();
     // 预加载类的信息存储在 PRELOADED_CLASSES 变量中，它的值为 "preloaded-classes"。
     InputStream is = ZygoteInit.class.getClassLoader().getResourceAsStream(
                              PRELOADED_CLASSES);
        if (is == null) {
            Log.e(TAG, "Couldn't find " + PRELOADED_CLASSES + ".");
        } else {
               ..... // 做一些统计和准备工作。

            try {
                BufferedReader br
                    = new BufferedReader(new InputStreamReader(is), 256);
                // 读取文件的每一行，忽略 # 开头的注释行。
                int count = 0;
                String line;
                String missingClasses = null;
                while ((line = br.readLine()) != null) {
                      line = line.trim();
                    if (line.startsWith("#") || line.equals("")) {
                        continue;
                    }

                    try {
```

```
                    // 通过 Java 反射来加载类，line 中存储的是预加载的类名。
                    Class.forName(line);
                    ......
                    count++;
                } catch (ClassNotFoundException e) {
                    ......
                } catch (Throwable t) {
                    ......
                }
                }
            ...... // 扫尾工作
        }
        }
```

preloadClasses 看起来是如此简单，但是你知道它有多少个类需要预先加载吗？

用 coolfind 工具程序在 framework 中搜索名为“preloaded-classes”的文件，最后会在 framework/base 目录下找到。它是一个文本文件，内容如下：

```
# Classes which are preloaded by com.android.internal.os.ZygoteInit.
# Automatically generated by
# frameworks/base/tools/preload/WritePreloadedClassFile.java.
# MIN_LOAD_TIME_MICROS=1250   // 超时控制
android.R$styleable
android.accounts.AccountManager
android.accounts.AccountManager$4
android.accounts.AccountManager$6
android.accounts.AccountManager$AmsTask
android.accounts.AccountManager$BaseFutureTask
android.accounts.AccountManager$Future2Task
android.accounts.AuthenticatorDescription
android.accounts.IAccountAuthenticatorResponse$Stub
android.accounts.IAccountManager$Stub
android.accounts.IAccountManagerResponse$Stub
......// 一共有 1268 行
```

这个 preload-class 一共有 1268 行，试想，加载这么多类得花多少时间！

说明　preload_class 文件由 framework/base/tools/preload 工具生成，它需要判断每个类加载的时间是否大于 1250 微秒，超过这个时间的类就会被写到 preload-classes 文件中，最后由 zygote 预加载。这方面的内容读者可参考有关 preload 工具中的说明，这里就不再赘述。

preloadClass 函数的执行时间比较长，这是导致 Android 系统启动慢的原因之一。可对比进行一些优化，但优化是基于对整个系统有比较深入的了解才能实现的。

注意　在本章的拓展思考内容中，我们会讨论 Android 的启动速度问题。

preloadResources 和 preloadClass 类似，它主要是加载 framework-res.apk 中的资源。这

里就不再介绍它了。

说明 在 UI 编程中常使用的 com.android.R.XXX 资源是系统默认的资源，它们就是由 zygote 加载的。

3. 启动 system_server

我们现在要分析的是第三个关键点：startSystemServer。这个函数会创建 Java 世界中系统 Service 所驻留的进程 system_server，该进程是 framework 的核心。如果它死了，就会导致 zygote 自杀。先来看看这个核心进程是如何启动的。

[-->ZygoteInit.java]

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    // 设置参数
    String args[] = {
        "--setuid=1000",//uid和gid等的设置
        "--setgid=1000",
        "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,
                     3001,3002,3003",
        "--capabilities=130104352,130104352",
        "--runtime-init",
        "--nice-name=system_server",  // 进程名，叫system_server
        "com.android.server.SystemServer", // 启动的类名
    };
    ZygoteConnection.Arguments parsedArgs = null;
    int pid;
    try {
      // 把上面字符串数组参数转换成Arguments对象。具体内容请读者自行研究。
        parsedArgs = new ZygoteConnection.Arguments(args);
        int debugFlags = parsedArgs.debugFlags;
       //fork一个子进程，看来，这个子进程就是system_server进程。
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids, debugFlags, null);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
/*
 关于fork的知识，请读者务必花些时间去研究。如果对fork的具体实现也感兴趣，可参考
 《Linux内核源代码情景分析》一书（该书由浙江大学出版社出版，作者为毛德操、胡希明）。
 在下面代码中，如果pid为零，则表示处于子进程中，也就是处于system_server进程中。
*/
    if (pid == 0) {
      // ① system_server进程的工作
        handleSystemServerProcess(parsedArgs);
    }
```

```
        //zygote 返回 true
        return true;
    }
```

这里出现了一个分水岭，即 zygote 进行了一次无性繁殖，分裂出了一个 system_server 进程（即代码中的 Zygote.forkSystemServer 这句话）。关于它的故事，我们会在后文专门分析，这里先说 zygote。

4. 有求必应之等待请求——runSelectLoopMode

待 zygote 从 startSystemServer 返回后，将进入第四个关键函数：runSelectLoopMode。前面在第一个关键点 registerZygoteSocket 中注册了一个用于 IPC 的 Socket，不过那时还没有地方用到它。它的用途将在这个 runSelectLoopMode 中体现出来，请看下面的代码：

[-->ZygoteInit.java]

```
private static void runSelectLoopMode()
throws MethodAndArgsCaller {
        ArrayList<FileDescriptor> fds = new ArrayList();
        ArrayList<ZygoteConnection> peers = new ArrayList();
        FileDescriptor[] fdArray = new FileDescriptor[4];
       //sServerSocket 是我们先前在 registerZygoteSocket 中建立的 Socket。
        fds.add(sServerSocket.getFileDescriptor());
        peers.add(null);

        int loopCount = GC_LOOP_COUNT;
        while (true) {
            int index;
              try {
                fdArray = fds.toArray(fdArray);
        /*
           selectReadable 内部调用 select，使用多路复用 I/O 模型。
           当有客户端连接或有数据时，则 selectReadable 就会返回。
         */
                index = selectReadable(fdArray);
            }
          else if (index == 0) {
             // 有一个客户端连接上。请注意，客户端在 Zygote 的代表是 ZygoteConnection。
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
               // 客户端发送了请求，peers.get 返回的是 ZygoteConnection。
              // 后续处理将交给 ZygoteConnection 的 runOnce 函数完成。
                done = peers.get(index).runOnce();
        }
    }
```

runSelectLoopMode 比较简单，就是：

- 处理客户连接和客户请求。其中客户在 zygote 中用 ZygoteConnection 对象来表示。
- 客户的请求由 ZygoteConnection 的 runOnce 来处理。

提示 runSelectLoopMode 比较简单，但它使用的 select 背后所代表的思想却并非那么简单。建议读者以此为契机，认真学习常用的 I/O 模型，包括阻塞式、非阻塞式、多路复用、异步 I/O 等，掌握这些知识，对于未来编写大型系统很有帮助。

关于 zygote 是如何处理请求的，将单独用一节内容进行讨论。

4.2.3 关于 zygote 的总结

zygote 是在 Android 系统中创建 Java 世界的盘古，它创建了第一个 Java 虚拟机，同时它又是女娲，它成功地繁殖了 framework 的核心 system_server 进程。做为 Java 语言的受益者，我们理应回顾一下 zygote 创建 Java 世界的步骤：

- 第一天：创建 AppRuntime 对象，并调用它的 start。此后的活动则由 AppRuntime 来控制。
- 第二天：调用 startVm 创建 Java 虚拟机，然后调用 startReg 来注册 JNI 函数。
- 第三天：通过 JNI 调用 com.android.internal.os.ZygoteInit 类的 main 函数，从此进入了 Java 世界。然而在这个世界刚开创的时候，什么东西都没有。
- 第四天：调用 registerZygoteSocket。通过这个函数，它可以响应子孙后代的请求。同时 zygote 调用 preloadClasses 和 preloadResources，为 Java 世界添砖加瓦。
- 第五天：zygote 觉得自己的工作压力太大，便通过调用 startSystemServer 分裂一个子进程 system_server 来为 Java 世界服务。
- 第六天：zygote 完成了 Java 世界的初创工作，它已经很满足了。下一步该做的就是调用 runSelectLoopMode 后，便沉沉地睡去了。

以后的日子：zygote 随时守护在我们的周围，当接收到子孙后代的请求时，它会随时醒来，为它们工作。

如果支持中文编码的话，我一定要为 zygote 取名为“盘古_女娲”。

4.3 SystemServer 分析

SystemServer 的进程名实际上叫做“system_server”，这里我们可将其简称为 SS。SS 作为 zygote 的嫡长子，其重要性不言而喻。关于这一点，通过代码分析便可马上知晓。

4.3.1 SystemServer 的诞生

我们先回顾一下 SS 是怎么创建的，代码如下所示：

```
String args[] = {
            "--setuid=1000",
            "--setgid=1000",
            "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,
                         3001,3002,3003",
            "--capabilities=130104352,130104352",
            "--runtime-init",
            "--nice-name=system_server",
            "com.android.server.SystemServer",
        };
        ZygoteConnection.Arguments parsedArgs = null;

        int pid;
       parsedArgs = new ZygoteConnection.Arguments(args);
       int debugFlags = parsedArgs.debugFlags;
       pid = Zygote.forkSystemServer(  // 调用 forkSystemServer
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids, debugFlags, null);
```

从上面的代码中可以看出，SS 是由 Zygote 通过 Zygote.forkSystemServer 函数 fork 诞生出来的。这里会有什么玄机吗？先一起来看看 forkSystemServer 的实现。它是一个 native 函数，实现在 dalvik_system_Zygote.c 中，如下所示：

[-->dalvik_system_Zygote.c]

```
static void Dalvik_dalvik_system_Zygote_forkSystemServer(
                        const u4* args, JValue* pResult)
{
     pid_t pid;
    // 根据参数，fork 一个子进程。
    pid = forkAndSpecializeCommon(args);
    if (pid > 0) {
        int status;
        gDvm.systemServerPid = pid;// 保存 system_server 的进程 id。
        // 函数退出前须先检查刚创建的子进程是否退出了。
        if (waitpid(pid, &status, WNOHANG) == pid) {
            // 如果 system_server 退出了，Zygote 直接干掉了自己。
            // 看来 Zygote 和 SS 的关系异常紧密，简直是生死与共！
            kill(getpid(), SIGKILL);
        }
    }
    RETURN_INT(pid);
}
```

下面，再看看 forkAndSpecializeCommon，代码如下所示：

[-->dalvik_system_Zygote.c]

```
static pid_t forkAndSpecializeCommon(const u4* args)
{
```

```
    pid_t pid;
    uid_t uid = (uid_t) args[0];
    gid_t gid = (gid_t) args[1];
    ArrayObject* gids = (ArrayObject *)args[2];
    u4 debugFlags = args[3];
    ArrayObject *rlimits = (ArrayObject *)args[4];
   // 设置信号处理，待会儿要看看这个函数。
    setSignalHandler();
    pid = fork(); //fork 子进程。
   if (pid == 0) {
     // 要根据传入的参数对子进程做一些处理，例如设置进程名，设置各种 id（用户 id，组 id 等）。
   }
.....
}
```

最后看看 setSignalHandler 函数，它由 Zygote 在 fork 子进程前调用，代码如下所示：

[-->dalvik_system_Zygote.c]

```
static void setSignalHandler()
{
    int err;
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = sigchldHandler;
    err = sigaction (SIGCHLD, &sa, NULL);// 设置信号处理函数，该信号是子进程死亡的信号。
}
// 我们直接看这个信号处理函数 sigchldHandler。
static void sigchldHandler(int s)
{
    pid_t pid;
    int status;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
              } else if (WIFSIGNALED(status)) {
            }
        }
        // 如果死去的子进程是 SS，则 Zygote 把自己也干掉了，这样就做到了生死与共！
        if (pid == gDvm.systemServerPid) {
            kill(getpid(), SIGKILL);
        }
    }
```

OK，作为 Zygote 的嫡长子，SS 确实具有非常高的地位，竟然到了与 Zygote 生死与共的地步！它为什么这么重要呢？我们现在就从 forkSystemServer 下手来分析 SS 究竟承担了怎样的工作使命。

注意 关于源代码定位的问题，不少人在面对浩瀚的代码时，常常不知道具体函数是在哪个

文件中定义的。这里，就 Source insight 的使用提几点建议：

1）加入工程的时候，不要把所有目录全部加进去，否则会导致解析速度异常缓慢。我们可以先加入 framework 目录，在以后另有需要时，再加入其他目录。

2）除了 Source insight 的工具外，还需要有一个能搜索文件中特定字符串的工具，我用的是 coolfind。我就是通过它在源码中搜索到 forkSystemServer 这个函数的，并且找到了它的实现文件 dalvik_system_Zygote.c。在 Linux 下也有对应的工具，但其工作速度比 coolfind 缓慢。

3）在 Linux 下，可通过 wine（一个支持 Linux 平台安装 Windows 软件的工具）安装 Source insight。

4.3.2　SystemServer 的重要使命

SS 诞生后，便和生父 Zygote 分道扬镳了，它有了自己的使命。它的使命是什么呢？代码如下所示：

```
pid = Zygote.forkSystemServer();
 if (pid == 0) { //SS 进程返回 0，那么下面这句话就是 SS 的使命：
        handleSystemServerProcess(parsedArgs);
}
```

SS 调用 handleSystemServerProcess 来承担自己的职责。

[-->ZygoteInit.java]

```
private static void handleSystemServerProcess(
       ZygoteConnection.Arguments parsedArgs)
       throws ZygoteInit.MethodAndArgsCaller {
        // 关闭从 Zygote 那里继承下来的 Socket。
        closeServerSocket();
      // 设置 SS 进程的一些参数。
        setCapabilities(parsedArgs.permittedCapabilities,
                  parsedArgs.effectiveCapabilities);
        // 调用 ZygoteInit 函数。
         RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
  }
```

现在 SS 走到 RuntimeInit 中了，它的代码在 RuntimeInit.java 中，如下所示：

[-->RuntimeInit.java]

```
public static final void zygoteInit(String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {
    // 做一些常规初始化。
    commonInit();
    // ① native 层的初始化。
    zygoteInitNative();
```

```
        int curArg = 0;
        for ( /* curArg */ ; curArg < argv.length; curArg++) {
              String arg = argv[curArg];

               if (arg.equals("--")) {
                   curArg++;
                   break;
               } else if (!arg.startsWith("--")) {
                   break;
               } else if (arg.startsWith("--nice-name=")) {
                   String niceName = arg.substring(arg.indexOf('=') + 1);
                  // 设置进程名为 niceName，也就是 "system_server"。
                   Process.setArgV0(niceName);
               }
           }
           //startClass 名为 "com.android.server.SystemServer"。
           String startClass = argv[curArg++];
           String[] startArgs = new String[argv.length - curArg];
           System.arraycopy(argv, curArg, startArgs, 0, startArgs.length);
          // ②调用 startClass，也就是 com.android.server.SystemServer 类的 main 函数。
           invokeStaticMain(startClass, startArgs);
    }
```

对于上面列举出的两个关键点（即代码中的①和②），我们一个一个地分析。

1. zygoteInitNative 分析

先看 zygoteInitNative，它是一个 native 函数，实现在 AndroidRuntime.cpp 中。

[-->AndroidRuntime.cpp]

```
static void com_android_internal_os_RuntimeInit_zygoteInit(
JNIEnv* env,jobject clazz)
{
    gCurRuntime->onZygoteInit();
}
//gCurRuntime 是什么？还记得我们在本章开始说的 app_process 的 main 函数吗？
int main(int argc, const char* const argv[])
{
   AppRuntime runtime;// 就是这个。当时我们没顾及它的构造函数，现在回过头看看。
}
//AppRuntime 的定义
class AppRuntime : public AndroidRuntime
static AndroidRuntime* gCurRuntime = NULL; // gCurRuntime 为全局变量。
AndroidRuntime::AndroidRuntime()
{
    SkGraphics::Init();//Skia 库初始化
    SkImageDecoder::SetDeviceConfig(SkBitmap::kRGB_565_Config);
    SkImageRef_GlobalPool::SetRAMBudget(512 * 1024);
    gCurRuntime = this; //gCurRuntime 被设置为 AndroidRuntime 对象自己。
}
```

由于 SS 是从 zygote fork 出来的，所以它也拥有 zygote 进程中定义的这个 gCurRuntime，也就是 AppRuntime 对象。那么，它的 onZygoteInit 会干些什么呢？它的代码在 App_main.cpp 中，我们一起来看：

[-->App_main.cpp]

```
    virtual void onZygoteInit()
    {
        // 下面这些东西和 Binder 有关，但读者可以先不管它。
        sp<ProcessState> proc = ProcessState::self();
        if (proc->supportsProcesses()) {
            proc->startThreadPool();// 启动一个线程，用于 Binder 通信。
        }
}
```

一言以蔽之，SS 调用 zygoteInitNative 后，将与 Binder 通信系统建立联系，这样 SS 就能够使用 Binder 了。关于 Binder 的知识，将在第 6 章中详细介绍，大家现在不必关注。

2. invokeStaticMain 分析

再来看第二个关键点 invokeStaticMain。代码如下所示：

[-->RuntimeInit.java]

```
private static void invokeStaticMain(String className, String[] argv)
        throws ZygoteInit.MethodAndArgsCaller {

    ......// 注意我们的参数，className 为 "com.android.server.SystemServer"。
    Class<?> cl;

    try {
        cl = Class.forName(className);
    } catch (ClassNotFoundException ex) {
        throw new RuntimeException(
                "Missing class when invoking static main " + className,
                ex);
    }

    Method m;
    try {
       // 找到 com.android.server.SystemServer 类的 main 函数，肯定有地方要调用它。
        m = cl.getMethod("main", new Class[] { String[].class });
    } catch (NoSuchMethodException ex) {
        ......
    } catch (SecurityException ex) {
        ......
    }

    int modifiers = m.getModifiers();
    if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
```

```
            ......
        }
        // 抛出一个异常，为什么不在这里直接调用上面的 main 函数呢?
        throw new ZygoteInit.MethodAndArgsCaller(m, argv);
    }
```

invokeStaticMain 竟然抛出了一个异常，它是在哪里被截获的呢？原来是在 ZygoteInit 的 main 函数中。请看这段代码：

注意 我们所在的进程是 system_server。

[-->ZygoteInit.java]

```
    ....
if (argv[1].equals("true")) {
     // 在 SS 进程中，抛出一个异常 MethodAndArgsCaller。
     startSystemServer();
     ......
    catch (MethodAndArgsCaller caller) {
     // 被截获，调用 caller 的 run 函数。
       caller.run();
  }
```

再来看看 MethodAndArgsCaller 的 run 函数。

```
public void run() {
   try {
   // 这个 mMethod 为 com.android.server.SystemServer 的 main 函数。
   mMethod.invoke(null, new Object[] { mArgs });
  } catch (IllegalAccessException ex) {
     ......
  }
}
```

抛出的这个异常最后会导致 com.android.server.SystemServer 类的 main 函数被调用。不过这里有一个疑问，为什么不在 invokeStaticMain 那里直接调用，而是采用这种抛异常的方式呢？我对这个问题的看法是：

这个调用是在 ZygoteInit.main 中，相当于 Native 的 main 函数，即入口函数，位于堆栈的顶层。如果不采用抛异常的方式，而是在 invokeStaticMain 那里调用，则会浪费之前函数调用所占用的一些调用堆栈。

注意 关于这个问题的深层思考，读者可以利用 fork 和 exec 的知识。个人认为这种抛异常的方式是对 exec 的一种近似模拟，因为后续的工作将交给 com.android.server.SystemServer 类来处理。

3. SystemServer 的真面目

ZygoteInit 分裂产生的 SS，其实就是为了调用 com.android.server.SystemServer 的 main 函数，这简直就是改头换面！下面就来看看这个真实的 main 函数，代码如下所示：

[-->SystemServer.java]

```
public static void main(String[] args) {
    ......
     // 加载 libandroid_servers.so。
    System.loadLibrary("android_servers");
   // 调用 native 的 init1 函数。
   init1(args);
}
```

其中 main 函数将加载 libandroid_server.so 库，这个库所包含的源码文件在文件夹 framework/base/services/jni 下。

（1）init1 分析

init1 是 native 函数，在 com_android_server_SystemServer.cpp 中实现。来看看它，代码如下所示：

[-->com_android_server_SystemServer.cpp]

```
extern "C" int system_init();

static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
    system_init();// 调用另外一个函数。
}
```

system_init 的实现在 system_init.cpp 中，它的代码如下所示：

[-->system_init.cpp]

```
extern "C" status_t system_init()
{
    // 下面这些调用和 Binder 有关，我们会在第 6 章中讲述，这里先不必管它。
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();

    sp<GrimReaper> grim = new GrimReaper();
    sm->asBinder()->linkToDeath(grim, grim.get(), 0);
    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        //SurfaceFlinger 服务在 system_server 进程中创建。
        SurfaceFlinger::instantiate();
    }
```

```
        ......

        // 调用 com.android.server.SystemServer 类的 init2 函数。
        AndroidRuntime* runtime = AndroidRuntime::getRuntime();
        runtime->callStatic("com/android/server/SystemServer", "init2");

    // 下面这几个函数的调用和 Binder 通信有关，具体内容在第 6 章中介绍。
        if (proc->supportsProcesses()) {
            ProcessState::self()->startThreadPool();
           // 调用 joinThreadPool 后，当前线程也加入到 Binder 通信的大潮中。
            IPCThreadState::self()->joinThreadPool();
           }
        return NO_ERROR;
    }
```

init1 函数创建了一些系统服务，然后把调用线程加入到 Binder 通信中了。不过其间还通过 JNI 调用了 com.android.server.SystemServer 类的 init2 函数，下面就来看看这个 init2 函数。

（2）init2 分析

init2 在 Java 层，代码在 SystemServer.java 中，如下所示：

[-->SystemServer.java]

```
public static final void init2() {
    Thread thr = new ServerThread();
    thr.setName("android.server.ServerThread");
    thr.start(); // 启动一个 ServerThread。
}
```

启动了一个 ServerThread 线程。请直接看它的 run 函数。这个函数比较长，大概看看它干了什么即可。

[-->SystemServer.java::ServerThread 的 run 函数]

```
public void run(){
                ....
  // 启动 Entropy Service。
  ServiceManager.addService("entropy", new EntropyService());
  // 启动电源管理服务。
  power = new PowerManagerService();
  ServiceManager.addService(Context.POWER_SERVICE, power);
  // 启动电池管理服务。
  battery = new BatteryService(context);
  ServiceManager.addService("battery", battery);

   // 初始化看门狗，在本章的拓展内容中将介绍关于看门狗的知识。
   Watchdog.getInstance().init(context, battery, power, alarm,
```

```
                                    ActivityManagerService.self());

    // 启动 WindowManager 服务。
    wm = WindowManagerService.main(context, power,
                        factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL);
    ServiceManager.addService(Context.WINDOW_SERVICE, wm);

    // 启动 ActivityManager 服务。
     (ActivityManagerService)ServiceManager.getService("activity"))
                        .setWindowManager(wm);

    ......// 总之，系统各项重要的服务都在这里启动。
     Looper.loop();   // 进行消息循环，然后处理消息。关于这部分内容参见第 5 章。
}
```

init2 函数比较简单，就是单独创建一个线程，用以启动系统的各项服务，至此，读者或许能理解 SS 的重要性了吧？

Java 世界的核心 Service 都在这里启动，所以它非常重要。

说明　本书不对这些 Service 做进一步的分析，今后有机会再专门介绍。

4.3.3　关于 SystemServer 的总结

SS 曲折的调用流程真让人眼花缭乱，我们用图 4-2 来展示这一过程：

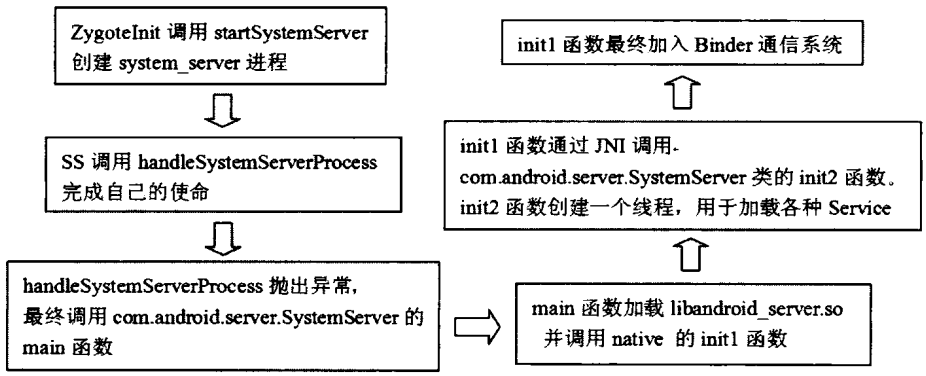


图 4-2　SystemServer 的调用流程

注意　init1 函数最终导致进程的主线程加入到 Binder 通信的大潮中，关于 Binder 的知识将在第 6 章中介绍。

4.4 zygote 的分裂

前文已经讲到，zygote 分裂出嫡长子 system_server 后，就通过 runSelectLoopMode 等待并处理来自客户的消息了，那么，谁会向 zygote 发送消息呢？这里以一个 Activity 的启动为例，具体分析 zygote 是如何分裂和繁殖的。

4.4.1 ActivityManagerService 发送请求

ActivityManagerService 也是由 SystemServer 创建的。假设通过 startActivity 来启动一个新的 Activity，而这个 Activity 附属于一个还未启动的进程，那么这个进程该如何启动呢？先来看看 ActivityManagerService 中的 startProcessLocked 函数，代码如下所示：

[-->ActivityManagerService.java]

```
private final void startProcessLocked(ProcessRecord app,
            String hostingType, String hostingNameStr){
        ......// 这个 ActivityManagerService 类很复杂，有 14657 行！！！
    if ("1".equals(SystemProperties.get("debug.checkjni"))) {
            debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
        }
        if ("1".equals(SystemProperties.get("debug.assert"))) {
            debugFlags |= Zygote.DEBUG_ENABLE_ASSERT;
        }
     // 这个 Process 类是 Android 提供的，并非 JDK 中的 Process 类。
     int pid = Process.start("android.app.ActivityThread",
                mSimpleProcessManagement ? app.processName : null, uid, uid,
                    gids, debugFlags, null);
    ......
}
```

接着来看看 Process 的 start 函数，这个 Process 类是 android.os.Process，它的代码在 Process.java 中，代码如下所示：

[-->Process.java]

```
public static final int start(final String processClass,final String niceName,
int uid, int gid, int[] gids,int debugFlags,String[] zygoteArgs)
{
    // 注意，processClass 的值是 "android.app.ActivityThread"。
    if (supportsProcesses()) {
        try {
            // 调用 startViaZygote。
            return startViaZygote(processClass, niceName, uid, gid, gids,
                    debugFlags, zygoteArgs);
        }
    }
}
```

[-->Process.java::startViaZygote()]

```
private static int startViaZygote(final String processClass,
     final String niceName,final int uid, final int gid,final int[] gids,
     int debugFlags,String[] extraArgs) throws ZygoteStartFailedEx {
     int pid;
     ......// 一些参数处理，最后调用 zygoteSendArgsAndGetPid 函数。
    argsForZygote.add("--runtime-init"); // 这个参数很重要。
    argsForZygote.add("--setuid=" + uid);
    argsForZygote.add("--setgid=" + gid);
    pid = zygoteSendArgsAndGetPid(argsForZygote);
     return pid;
}
```

[-->Process.java::zygoteSendArgsAndGetPid()]

```
private static int zygoteSendArgsAndGetPid(ArrayList<String> args)
                                          throws ZygoteStartFailedEx {

   int pid;
   // openZygoteSocketIfNeeded？是不是打开了和 Zygote 通信的 Socket？
   openZygoteSocketIfNeeded();

   try {
           // 把请求的参数发到 Zygote。
            sZygoteWriter.write(Integer.toString(args.size()));
            sZygoteWriter.newLine();
            sZygoteWriter.write(arg);
            sZygoteWriter.newLine();
       }
       // 读取 Zygote 处理完的结果，便可得知是某个进程的 pid！
       sZygoteWriter.flush();
       pid = sZygoteInputStream.readInt();
       return pid;
}
[-->Process.java]
private static void openZygoteSocketIfNeeded() throws ZygoteStartFailedEx {
try {
         sZygoteSocket = new LocalSocket();// 果真如此！！
         // 连接 Zygote。
          sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
                              LocalSocketAddress.Namespace.RESERVED));
          sZygoteInputStream
                         = new DataInputStream(sZygoteSocket.getInputStream());
          sZygoteWriter = new BufferedWriter(
                                       new OutputStreamWriter(
                                    sZygoteSocket.getOutputStream()),256);
               }
          }
    }
}
```

好了，ActivityManagerService 终于向 zygote 发送请求了。请求的参数中有一个字符串，它的值是“android.app.ActivityThread”。现在该回到 zygote 处理请求那块去看看了。

注意 由于 ActivityManagerService 驻留于 SystemServer 进程中，所以正是 SS 向 Zygote 发送了消息。

4.4.2 有求必应之响应请求

前面有一节，题目叫“有求必应之等待请求”，那么这一节“有求必应之响应请求”会回到 ZygoteInit 中。下面就看看它是如何处理请求的。

[--->ZygoteInit.java]

```
private static void runSelectLoopMode() throws MethodAndArgsCaller{
    ......
   try {
          fdArray = fds.toArray(fdArray);
           ......
           else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
                // 调用 ZygoteConnection 的 runOnce。
                done = peers.get(index).runOnce();
        }
     ......
}
```

每当有请求数据发来时，zygote 就会调用 ZygoteConnection 的 runOnce 函数。ZygoteConnection 代码在 ZygoteConnection.java 文件中，来看看它的 runOnce 函数：

[-->ZygoteConnection.java]

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
    try {
            args = readArgumentList();// 读取 SS 发送过来的参数。
            descriptors = mSocket.getAncillaryFileDescriptors();
        }
          ......
          int pid;
     try {
            parsedArgs = new Arguments(args);
            applyUidSecurityPolicy(parsedArgs, peer);
            // 根据函数名，可知 Zygote 又分裂出了一个子进程。
            pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
```

```
                    parsedArgs.gids, parsedArgs.debugFlags, rlimits);
        }
         ......
        if (pid == 0) {
            // 子进程处理，这个子进程是不是我们要创建的 Activity 对应的子进程呢?
            handleChildProc(parsedArgs, descriptors, newStderr);
            return true;
        } else {
          //zygote 进程
           return handleParentProc(pid, descriptors, parsedArgs);
        }
    }
```

接下来，看看新创建的子进程在 handleChildProc 中做了些什么。

[-->ZygoteConnection.java]

```
    private void handleChildProc(Arguments parsedArgs,FileDescriptor[] descriptors,
                        PrintStream newStderr) throws ZygoteInit.MethodAndArgsCaller {

        ......// 根据传入的参数设置新进程的一些属性。
        //SS 发来的参数中有 "--runtime-init"，所以 parsedArgs.runtimeInit 为 true。
        if (parsedArgs.runtimeInit) {
            RuntimeInit.zygoteInit(parsedArgs.remainingArgs);
        } else {
        ......
       }
    }
```

[-->RuntimeInit.java]

```
    public static final void zygoteInit(String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        // 重定向标准输出和错误输出。
        System.setOut(new AndroidPrintStream(Log.INFO, "System.out"));
        System.setErr(new AndroidPrintStream(Log.WARN, "System.err"));

        commonInit();
        // 下面这个函数为 native 函数，最终会调用 AppRuntime 的 onZygoteInit，在那个函数中
        // 建立了和 Binder 的关系。
        zygoteInitNative();
        int curArg = 0;
        ......
        String startClass = argv[curArg++];
        String[] startArgs = new String[argv.length - curArg];
        System.arraycopy(argv, curArg, startArgs, 0, startArgs.length);
        // 最终还是调用 invokeStaticMain 函数，这个函数我们已经见识过了。
        invokeStaticMain(startClass, startArgs);
    }
```

zygote 分裂子进程后，自己将在 handleParentProc 中做一些扫尾工作，然后继续等待请求进行下一次分裂。

提示 这个 android.app.ActivityThread 类，实际上是 Android 中 apk 程序所对应的进程，它的 main 函数就是 apk 程序的 main 函数。从这个类的命名（android.app）中也可以看出些端倪。

通过这一节的分析，读者可以想到，Android 系统运行的那些 apk 程序，其父都是 zygote。这一点，可以通过 adb shell 登录后，用 ps 命令查看进程和父进程号来确认。

4.4.3 关于 zygote 分裂的总结

zygote 的分裂由 SS 控制，这个过程我们用图 4-3 来表示：

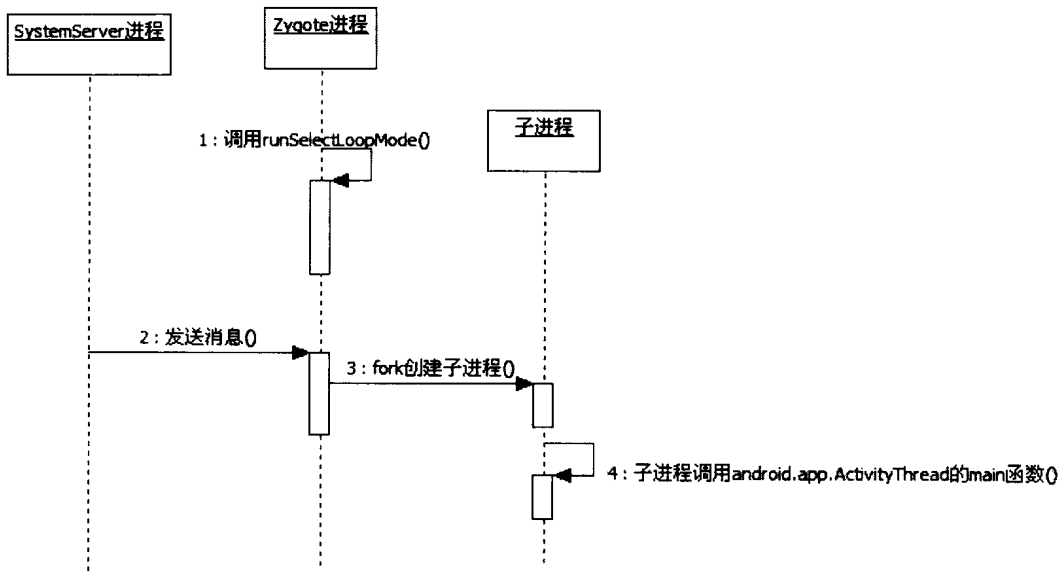


图 4-3 zygote 响应请求的过程

说明 这里借用了 UML 的时序图来表达 zygote 响应请求的过程。

4.5 拓展思考

4.5.1 虚拟机 heapsize 的限制

在分析 zygote 创建虚拟机的时候，我们说过系统默认设置的 Java 虚拟机堆栈最大为

16MB，这个值对于需要使用较大内存的程序（例如图片处理程序）来说远远不够。当然，可以修改这个默认值，例如我在 HTC G7 上就将其修改为 32MB 了，但这个改动是全局性的，也就是所有的 Java 程序都会是这个 32MB。我们能动态配置这个值吗？例如：

设置一个配置文件，每个进程启动的时候根据配置文件的参数来设置堆大小。

不过正如前面所说，我的这一美好愿望最终破灭了，原因只有一个：

zygote 是通过 fork 来创建子进程的，Zygote 本身设置的信息会被子进程全部继承，例如 Zygote 设置的堆栈为 16MB，那么它的子进程也会用这个 16MB。

关于这个问题，我目前想到了两个解决方案：

- 为 Dalivk 增加一个函数，这个函数允许动态调整最大堆的大小。
- zygote 通过 fork 子进程后，调用 exec 家族的函数来加载另外一个映像，该映像对应的程序会重新创建虚拟机，重新注册 JNI 函数，也就是模拟 zygote 创世界中前两天的工作，最后调用 android.app.ActivityThread 的 main 函数。这种方式应该是可行的，但难度较大，而且会影响运行速度。

提示　*关于本节所提出的问题，欢迎广大读者踊跃讨论。*

4.5.2　开机速度优化

Android 的开机速度慢这一现象一直受人诟病，Google 好像也没有要做这方面优化的意向，那么，在实际工作中又可以在哪些地方做一些优化呢？根据我目前所掌握的资料来看，开机时有三个地方耗时比较长：

- ZygoteInit 的 main 函数中 preloadClasses 加载了一千多个类，这耗费了不少时间。
- 开机启动时，会对系统内所有的 apk 文件进行扫描并收集信息，这个动作耗费的时间非常长。
- SystemServer 创建的那些 Service，会占用不少时间。

我们这里讨论第一个问题，如何减少 preloadClasses 的时间。其实，这个函数是可以去掉的，因为系统最终还是会在使用这些类时去加载，但这样就破坏了 Android 采用 fork 机制来创建 Java 进程的本意，而 fork 机制的好处是显而易见的：

- zygote 预加载的这些 class，在 fork 子进程时，仅需做一个复制即可。这就节约了子进程的启动时间。
- 根据 fork 的 copy-on-write 机制可知，有些类如果不做改变，甚至都不用复制，它们会直接和父进程共享数据。这样就会省去不少内存的占用。

开机速度优化是一项比较复杂的研究，目前有人使用 Berkeley Lab Checkpoint/Restart（BLCR）技术来提升开机速度。这一技术的构想其实挺简单，就是对系统做一个快照，将系统当前信息保存到一个文件中，当系统重启时，直接根据文件中的快照信息来恢复重启之前的状态。当然想法很简单，实现却是很复杂的，这里就不做进一步的讨论了，读者可自行展开深入的思考和研究。

提示 我在 VMWare 虚拟机上使用过类似的技术，它叫 Snapshort。开机速度的问题我更希望 Google 自己能加以重视并推动解决之。

4.5.3 Watchdog 分析

本章我们没有对 SystemServer 做更进一步的分析，不过作为拓展内容，这里想介绍一下 Watchdog。Watch Dog 的中文意思是 看门狗 。我依稀记得其最初存在的意义是因为早期嵌入式设备上的程序经常 跑飞 （比如说电磁干扰等），所以专门设置了一个硬件看门狗，每隔一段时间，看门狗就去检查一下某个参数是不是被设置了，如果发现该参数没有被设置，则判断为系统出错，然后就会强制重启。

在软件层面上，Android 对 SystemServer 的参数是否被设置也很谨慎，所以专门为它增加了一条看门狗，可它看的是哪个门呢？就是看几个重要 Service 的门，一旦发现 Service 出了问题，就会杀掉 system_server，而这也会使 zygote 随其一起自杀，最后导致重启 Java 世界。

我们先把 SystemServe 使用 Watchdog 的调用流程总结一下，然后以此为切入点来分析 Watchdog。SS 和 Watchdog 的交互流程可以总结为以下三个步骤：

- Watchdog. getInstance().init()
- Watchdog.getInstance().start()
- Watchdog. getInstance().addMonitor()

这三个步骤都非常简单。先看第一步。

1. 创建和初始化 Watchdog

getInstance 用于创建 Watchdog，一起来看看，代码如下所示：

[-->Watchdog.java]

```
public static Watchdog getInstance() {
 if (sWatchdog == null) {
      sWatchdog = new Watchdog(); // 使用了单例模式。
  }
   return sWatchdog;
}
public class Watchdog extends Thread
//Watchdog 从线程类派生，所以它会在一个单独的线程中执行。
private Watchdog() {
      super("watchdog");
      // 构造一个 Handler，Handler 的详细分析见第 5 章，读者可以简单地把它看作是消息处理的地方。
      // 它在 handleMessage 函数中处理消息。
      mHandler = new HeartbeatHandler();
      //GlobalPssCollected 和内存信息有关。
      mGlobalPssCollected = new GlobalPssCollected();
}
```

这条看门狗诞生后，再来看看 init 函数，代码如下所示：

[-->Watchdog.java]

```
public void init(Context context, BatteryService battery,
            PowerManagerService power, AlarmManagerService alarm,
            ActivityManagerService activity) {

        mResolver = context.getContentResolver();
        mBattery = battery;
        mPower = power;
        mAlarm = alarm;
        mActivity = activity;
         ......

        mBootTime = System.currentTimeMillis();// 得到当前时间
        ......
}
```

至此，看门狗诞生的知识就介绍完了，下面我们就让它动起来。

2. 看门狗跑起来

SystemServer 调用 Watchdog 的 start 函数，这将导致 Watchdog 的 run 在另外一个线程中被执行。代码如下所示：

[-->Watchdog.java]

```
public void run() {
     boolean waitedHalf = false;
     while (true) {// 外层 while 循环。
     mCompleted = false; //false 表明各个服务的检查还没完成。
     /*
       mHandler 的消息处理是在另外一个线程上，这里将给那个线程的消息队列发条消息，
       请求 Watchdog 检查 Service 是否工作正常。
   */
    mHandler.sendEmptyMessage(MONITOR);
    synchronized (this) {
            long timeout = TIME_TO_WAIT;
            long start = SystemClock.uptimeMillis();
            // 注意这个小 while 循环的条件，mForceKillSystem 为 true 也会导致退出循环。
                while (timeout > 0 && !mForceKillSystem) {
                  try {
                        wait(timeout);  // 等待检查的结果。
                    } catch (InterruptedException e) {
                      }
                    timeout = TIME_TO_WAIT - (SystemClock.uptimeMillis() - start);
                }
              //mCompleted 为 true，表示 service 一切正常。
               if (mCompleted && !mForceKillSystem) {
                   waitedHalf = false;
                    continue;
                }
```

```
                    // 如果 mCompleted 不为 true，看门狗会尽责地再检查一次。
                    if (!waitedHalf) {
                        ......
                        waitedHalf = true;
                        continue;// 再检查一次
                    }
                }
                // 已经检查过两次了，还是有问题，这回是真有问题了，所以 SS 需要把自己干掉。
                if (!Debug.isDebuggerConnected()) {
                      Process.killProcess(Process.myPid());
                       System.exit(10); // 干掉自己
                }
                ......
                waitedHalf = false;
            }
    }
```

这个 run 函数看起来还是比较简单，就是：

隔一段时间给另外一个线程发送一条 MONITOR 消息，那个线程将检查各个 Service 的健康情况。而看门狗会等待检查结果，如果第二次还没有返回结果，那么它会杀掉 SS。

下面来看看检查线程究竟是怎么检查 Service 的。

3. 列队检查

这么多 Service，哪些是看门狗比较关注的呢？一共有三个 Service 是需要交给 Watchdog 检查的：

- ❑ ActivityManagerService
- ❑ PowerManagerService
- ❑ WindowManagerService

要想支持看门狗的检查，就需要让这些 Service 实现 monitor 接口，然后 Watchdog 就会调用它们的 monitor 函数进行检查了。检查的地方是在 HeartbeatHandler 类的 handleMessage 中，代码如下所示：

[-->Watchdog.java::HeartbeatHandler]

```
final class HeartbeatHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
                ......
                case MONITOR: {
                    ......
                   long now = SystemClock.uptimeMillis();
                   final int size = mMonitors.size();
                   // 检查各个服务，并设置当前检查的对象为 mCurrentMonitor。
                    for (int i = 0 ; i < size ; i++) {
                        mCurrentMonitor = mMonitors.get(i);
```

```
                    mCurrentMonitor.monitor();// 检查这个对象。
                }
                // 如果没问题，则设置 mCompleted 为真。
                synchronized (Watchdog.this) {
                    mCompleted = true;
                    mCurrentMonitor = null;
                }
            } break;
        }
    }
}
```

那么，Service 的健康情况是怎么判断的呢？我们以 PowerManagerService 为例，先看看它是怎么把自己交给看门狗检查的，代码如下所示：

[-->PowerManagerService.java]

```
PowerManagerService()
{
    ......
    // 在构造函数中把自己加入 Watchdog 的检查队列。
    Watchdog.getInstance().addMonitor(this);
}
```

而 Watchdog 调用各个 monitor 函数到底又检查了些什么呢？再看看它实现的 monitor 函数吧。

[-->PowerManagerService.java]

```
public void monitor() {
  // 原来 monitor 检查的就是这些 Service 是不是发生死锁了！
    synchronized (mLocks) { }
}
```

原来，Watchdog 最怕系统服务死锁了，对于这种情况也只能采取杀系统的办法了。

说明　这种情况，我只碰到过一次，原因是有一个函数占着锁，但长时间没有返回。没返回的原因是这个函数需要和硬件交互，而硬件又没有及时返回。

关于 Watchdog，我们就介绍到这里。另外，它还能检查内存的使用情况，这一部分内容读者可以自行研究。

4.6　本章小结

本章对 zygote 进程做了较为深入的分析，zygote 的主要工作是开创 Java 世界，本章介

绍了它创世纪的七大步骤。另外，本章还分析了 zygote 的“嫡长子”——System_server 进程，这个进程是 Java 世界中的系统 Service 的驻留地，所以它非常重要。对于 System_server 进程，本章重点关注的是它的创建和初始化过程。此外，我们还分析了一个 Activity 所属进程的创建过程，原来这个进程是由 ActivityManagerService 发送请求给 zygote，最后由 zygote 通过 fork 的方式创建的。

在本章的拓展部分，我们讨论了 Dalvik 虚拟机对 heap 大小的设置，以及可能的修改方法，另外还探讨了 Android 系统开机速度的问题。最后还分析了 System_server 中 Watchdog 的工作流程。



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_imple 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 时使用的是同一个
// 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回避的话题，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》[①]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);
```

① 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:
#if defined(HAVE_PTHREADS)
    pthread_cond_t mCond;
```

```
#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) {
#if defined(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_flags++ 或 g_flags-- 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_flags++ 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 的使用方法比它所要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：

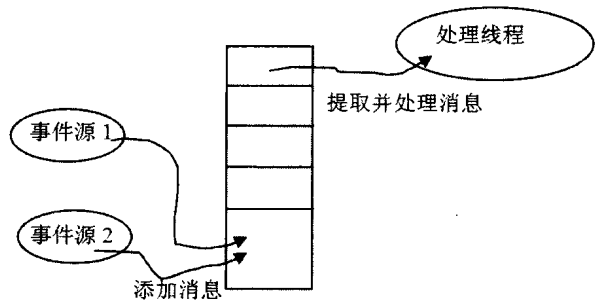


图 5-1　线程和消息处理的原理图

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，这需要从 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]

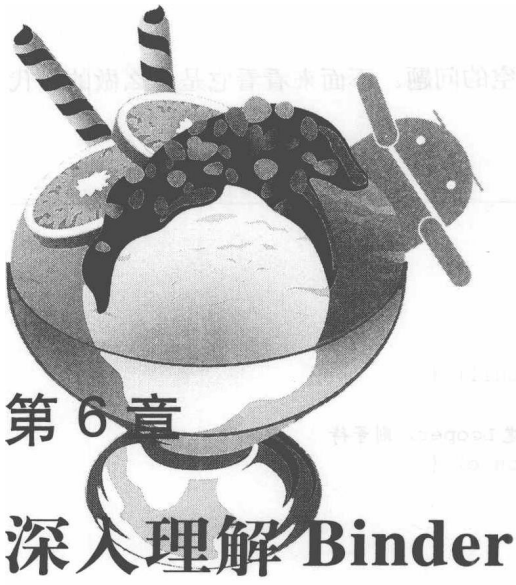
```
public class HandlerThread extends Thread{
// 线程 1 调用 getLooper 来获得新线程的 Looper
 public Looper getLooper() {
        ......
        synchronized (this) {
            while (isAlive() && mLooper == null) {
                try {
                    wait();// 如果新线程还未创建 Looper，则等待
                } catch (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 系统提供的一种 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：

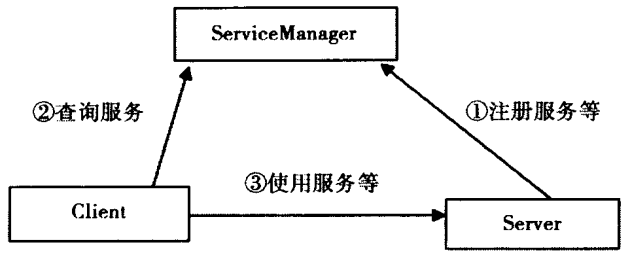


图 6-1　Client、Server 和 ServiceManager 三者之间的交互关系

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());

 //②MS 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 defaultServiceManager。

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 到底又是什么？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 函数是否可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：

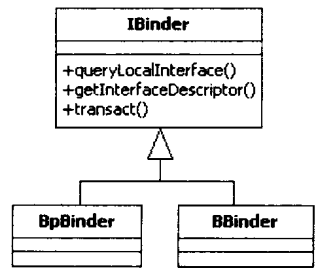


图 6-2 Binder 家族图谱

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 系统中有重要含义——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 IMPLENT 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。

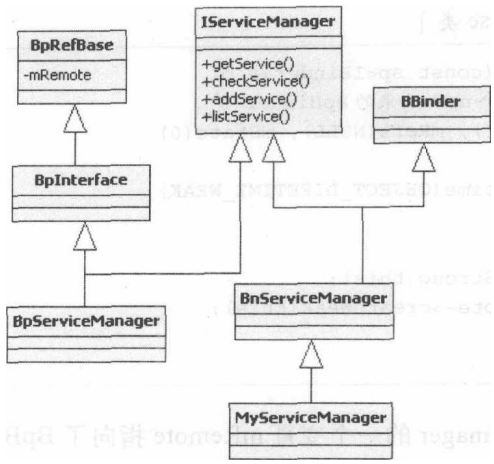


图 6-3　IServiceManager 的家族图谱

重要说明　以上这些关系很复杂，但 ServiceManager 并没有使用错综复杂的派生关系，它直接打开 Binder 设备并与之交互。后文还会详细分析它的实现代码。

图 6-3 中的 BpServiceManager，既然不像它的兄弟 BnServiceManager 那样与 Binder 有直接的血缘关系，那么它又是如何与 Binder 交互的呢？简言之，BpRefBase 中 mRemote 值就是 BpBinder。如果你不相信，仔细看 BpServiceManager 左边派生分支树上的一系列代码，它们都在 IServiceManager.cpp 中，如下所示：

[-->IServiceManager.cpp::BpServiceManager 类]

```
// 通过它的参数可得知，impl 是 IBinder 类型，看来与 Binder 有间接关系，它实际上是 BpBinder 对象。
BpServiceManager(const sp<IBinder>& impl)
    // 调用基类 BpInterface 的构造函数。
    : BpInterface<IServiceManager>(impl)
{
}
```

BpInterface 的实现代码如下所示：

[-->IInterface.h::BpInterface 类]

```
template<typename INTERFACE>
inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
    : BpRefBase(remote)// 基类构造函数
{
}
```

BpRefBase() 的实现代码如下所示：

[-->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(BC_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：

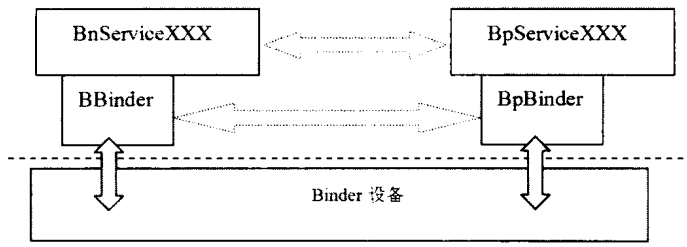


图 6-4 binder 通信层和业务层的关系图

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。

注意　通过这件事情是否能感悟到什么？我们确实可以抛开前面所有的那些封装，直接与 binder 设备打交道。

下面来看 ServiceManager 是怎么放弃华丽的封装去做 Manager 的。

1. ServiceManager 的入口函数

ServiceManager 的入口函数如下所示。

[-->ServiceManager.c]

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    //BINDER_SERVICE_MANAGER 的值为 NULL，是一个 magic number。
    void *svcmgr = BINDER_SERVICE_MANAGER;
    // ①应该是打开 binder 设备吧?
    bs = binder_open(128*1024);
    // ②成为 manager，是不是把自己的 handle 置为 0 ?
    binder_become_context_manager(bs)
    svcmgr_handle = svcmgr;
    // ③处理客户端发过来的请求。
     binder_loop(bs, svcmgr_handler);
}
```

这里，一共有三个重要关键点（即①～③）。必须对其逐一进行分析。

注意　有一些函数是在 Binder.c 中实现的，此 Binder.c 不是前面碰到的那个 Binder.cpp。

2. 打开 binder 设备

binder_open 函数用于打开 binder 设备，它的实现如下所示：

[-->Binder.c]

```
/*
  这里的 binder_open 应该与我们之前在 ProcessState 中看到的一样：
  1）打开 binder 设备
  2）内存映射
*/
struct binder_state *binder_open(unsigned mapsize)
{
    struct binder_state *bs;
    bs = malloc(sizeof(*bs));
    ....
    bs->fd = open("/dev/binder", O_RDWR);
    ....
    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
  }
```

果然如此，有了之前所掌握的知识，理解这里真的就不难了。

3. 成为老大

怎么才成为系统中独一无二的 manager 了呢？ manager 的实现如下面的代码所示：

[-->Binder.c]

```
int binder_become_context_manager(struct binder_state *bs)
{
   // 实现太简单了！这个 0 是否就是设置自己的 handle 呢?
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

4. 死磕 Binder

binder_loop 是一个很尽责的函数。为什么这么说呢？因为它老是围绕着 binder 设备转悠，实现代码如下所示：

[-->Binder.c]

```
/*
  注意 binder_handler 参数，它是一个函数指针，binder_loop 读取请求后将解析
  这些请求，最后调用 binder_handler 完成最终的处理。
*/
void binder_loop(struct binder_state *bs, binder_handler func)
{
    int res;
    struct binder_write_read bwr;
    readbuf[0] = BC_ENTER_LOOPER;
    binder_write(bs, readbuf, sizeof(unsigned));
    for (;;) {// 果然是循环。
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
        // 接收到请求，交给 binder_parse，最终会调用 func 来处理这些请求。
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
  }
```

5. 集中处理

往 binder_loop 中传的那个函数指针是 svcmgr_handler，它的代码如下所示：

[-->Service_manager.c]

```
int svcmgr_handler(struct binder_state *bs,struct binder_txn *txn,
                   struct binder_io *msg,struct binder_io *reply)
{
    struct svcinfo *si;
```

```
    uint16_t *s;
    unsigned len;
    void *ptr;
    // svcmgr_handle 就是前面说的那个 magic number，值为 NULL。
    //这里要比较 target 是不是自己。
    if (txn->target != svcmgr_handle)
        return -1;
    s = bio_get_string16(msg, &len);

    if ((len != (sizeof(svcmgr_id) / 2)) ||
        memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
       return -1;
    }

    switch(txn->code) {
    case SVC_MGR_GET_SERVICE://得到某个 service 的信息，service 用字符串表示。
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);//s 是字符串表示的 service 名称。
        ptr = do_find_service(bs, s, len);
        if (!ptr)
            break;
        bio_put_ref(reply, ptr);
        return 0;
    case SVC_MGR_ADD_SERVICE://对应 addService 请求。
        s = bio_get_string16(msg, &len);
        ptr = bio_get_ref(msg);
        if (do_add_service(bs, s, len, ptr, txn->sender_euid))
            return -1;
        break;
    //得到当前系统已经注册的所有 service 的名字。
    case SVC_MGR_LIST_SERVICES: {
        unsigned n = bio_get_uint32(msg);
        si = svclist;
        while ((n-- > 0) && si)
            si = si->next;
        if (si) {
            bio_put_string16(reply, si->name);
            return 0;
        }
        return -1;
    }
    default:
       return -1;
    }
    bio_put_uint32(reply, 0);
    return 0;
}
```

6.3.2　服务的注册

上面提到的 switch/case 语句，将实现 IServiceManager 中定义的各个业务函数，我

们重点看 do_add_service 这个函数，它最终完成了对 addService 请求的处理，代码如下所示：

[-->Service_manager.c]

```
int do_add_service(struct binder_state *bs,uint16_t *s, unsigned len,
                        void *ptr, unsigned uid)
{
    struct svcinfo *si;
    if (!ptr || (len == 0) || (len > 127))
        return -1;
     //svc_can_register 函数比较注册进程的 uid 和名字。
    if (!svc_can_register(uid, s)) {
        return -1;
    ......
```

将上面的函数暂时放一下，先介绍 svc_can_register 函数。

1. 不是什么都可以注册的

do_add_service 函数中的 svc_can_register，是用来判断注册服务的进程是否有权限的，代码如下所示：

[-->Service_manager.c]

```
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;
    // 如果用户组是 root 用户或 system 用户，则权限够高，允许注册。
    if ((uid == 0) || (uid == AID_SYSTEM))
        return 1;

    for (n = 0; n < sizeof(allowed) / sizeof(allowed[0]); n++)
        if ((uid == allowed[n].uid) && str16eq(name, allowed[n].name))
            return 1;

    return 0;
}
```

allowed 结构数组控制那些权限达不到 root 和 system 的进程，它的定义如下所示：

```
static struct {
    unsigned uid;
    const char *name;
} allowed[] = {
#ifdef LVMX
    { AID_MEDIA, "com.lifevibes.mx.ipc" },
#endif
    { AID_MEDIA, "media.audio_flinger" },
    { AID_MEDIA, "media.player" },
```

```
    { AID_MEDIA, "media.camera" },
    { AID_MEDIA, "media.audio_policy" },
    { AID_RADIO, "radio.phone" },
    { AID_RADIO, "radio.sms" },
    { AID_RADIO, "radio.phonesubinfo" },
    { AID_RADIO, "radio.simphonebook" },
    { AID_RADIO, "phone" },
    { AID_RADIO, "isms" },
    { AID_RADIO, "iphonesubinfo" },
    { AID_RADIO, "simphonebook" },
};
```

所以，如果 Server 进程权限不够 root 和 system，那么请记住要在 allowed 中添加相应的项。

2. 添加服务项

再回到我们的 do_add_service 中，如下所示：

[-->Service_manager.c]

```
int do_add_service(struct binder_state *bs,uint16_t *s, unsigned len,
                   void *ptr, unsigned uid){

    ..... //接前面的代码
    si = find_svc(s, len);
    if (si) {
        if (si->ptr) {
            return -1;
        }
        si->ptr = ptr;
    } else {
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        if (!si) {
            return -1;
        }
        //ptr是关键数据，可惜为void*类型。只有分析驱动的实现才能知道它的真实含义。
        si->ptr = ptr;
        si->len = len;
        memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
        si->name[len] = '\0';
        si->death.func = svcinfo_death;//service退出的通知函数。
        si->death.ptr = si;
        //这个svclist是一个list，保存了当前注册到ServiceManager中的信息。
        si->next = svclist;
        svclist = si;
    }

    binder_acquire(bs, ptr);
    /*
      我们希望当服务进程退出后，ServiceManager能有机会做一些清理工作，例如释放前面malloc出来的si。
```

```
        binder_link_to_death 会完成这项工作，每当有服务进程退出时，ServiceManager 都会得到来自
        binder 设备的通知。
    */
        binder_link_to_death(bs, ptr, &si->death);
        return 0;
    }
```

至此，服务注册分析完毕。可以知道，ServiceManager 不过就是保存了一些服务的信息。那么，这样做又有什么意义呢？

6.3.3 ServiceManager 存在的意义

为何需要一个 ServiceManager，其重要作用何在？

- ServiceManager 能集中管理系统内的所有服务，它能施加权限控制，并不是任何进程都能注册服务的。
- ServiceManager 支持通过字符串名称来查找对应的 Service。这个功能很像 DNS。
- 由于各种原因的影响，Server 进程可能生死无常。如果让每个 Client 都去检测，压力实在太大了。现在有了统一的管理机构，Client 只需要查询 ServiceManager，就能把握动向，得到最新信息。这可能正是 ServiceManager 存在的最大意义吧。

6.4 MediaPlayerService 和它的 Client

前面一直在讨论 ServiceManager 和它的 Client，现在我们以 MediaPlayerService 的 Client 来进行分析，换换口味吧。由于 ServiceManager 不是从 BnServiceManager 中派生的，所以之前没有讲请求数据是如何从通信层传递到业务层并进行处理的。本节，我们以 MediaPlayerService 和它的 Client 作为分析对象，试着解决这些遗留问题。

6.4.1 查询 ServiceManager

前文曾分析过 ServiceManager 的作用，一个 Client 想要得到某个 Service 的信息，就必须先和 ServiceManager 打交道，通过调用 getService 函数来获取对应 Service 的信息。请看来源于 IMediaDeathNotifier.cpp 中的例子 getMediaPlayerService()，它的代码如下所示：

[-->IMediaDeathNotifier.cpp]

```
/*
  这个函数通过与 ServiceManager 通信，获得一个能够与 MediaPlayerService 通信的 BpBinder，
  然后再通过障眼法 interface_cast，转换成一个 BpMediaPlayerService。
*/
IMediaDeathNotifier::getMediaPlayerService()
{
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
```

```
        do {
        // 向 ServiceManager 查询对应服务的信息，返回 BpBinder。
                binder = sm->getService(String16("media.player"));
                if (binder != 0) {
                break;
            }
        // 如果 ServiceManager 上还没有注册对应的服务，则需要等待，直到对应服务注册
        // 到 ServiceManager 中为止。
             usleep(500000);
        } while(true);

    /*
     通过 interface_cast，将这个 binder 转化成 BpMediaPlayerService，
     binder 中 handle 标识的一定是目的端 MediaPlayerService。
   */
    sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);
    }
    return sMediaPlayerService;
}
```

有了 BpMediaPlayerService，就能够使用任何 IMediaPlayerService 提供的业务逻辑函数了。例如 createMediaRecorder 和 createMetadataRetriever 等。

显而易见的是，调用的这些函数都将把请求数据打包发送给 Binder 驱动，并由 BpBinder 中的 handle 值找到对应端的处理者来处理。这中间的过程如下所示：

（1）通信层接收到请求。

（2）递交给业务层处理。

想进一步了解这中间的过程吗？下面就对此做详细分析。

6.4.2　子承父业

根据前面的分析可知，MediaPlayerService 驻留在 MediaServer 进程中，这个进程有两个线程在 talkWithDriver。假设其中有一个线程收到了请求信息，它最终会通过 executeCommand 调用来处理这个请求，实现代码如下所示：

[-->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 实际就是实现 BnServiceXXX 的那个对象，这样就直接定位到了业务层的对象。
           */
            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;
......
```

BBinder 和业务层有什么关系？还记得图 6-3 吗？我们以 MediaPlayerService 为例，来梳理一下其派生关系，如图 6-5 所示：

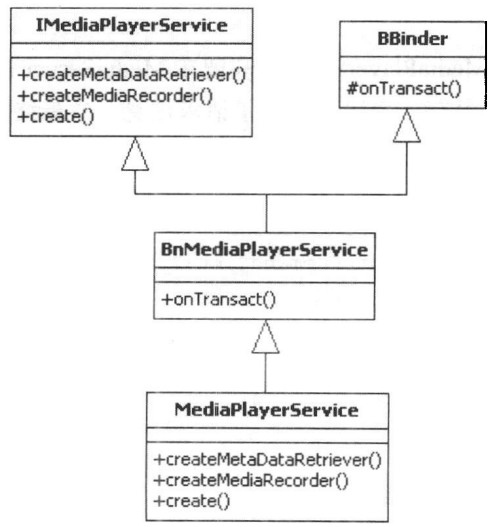


图 6-5　MediaPlayerService 家谱

BnMediaPlayerService 实现了 onTransact 函数，它将根据消息码调用对应的业务逻辑函数，这些业务逻辑函数由 MediaPlayerService 来实现。这一路的历程，如下面的代码所示：

[-->Binder.cpp]

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);
    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
         // 调用子类的 onTransact，这是一个虚函数。
            err = onTransact(code, data, reply, flags);
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
    }
    return err;
}
```

[-->IMediaPlayerService.cpp]

```
status_t BnMediaPlayerService::onTransact(uint32_t code, const Parcel& data,
                                          Parcel* reply, uint32_t flags)
{
    switch(code) {
        ......
        case CREATE_MEDIA_RECORDER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            // 从请求数据中解析对应的参数。
            pid_t pid = data.readInt32();
            // 子类要实现 createMediaRecorder 函数。
            sp<IMediaRecorder> recorder = createMediaRecorder(pid);
            reply->writeStrongBinder(recorder->asBinder());
            return NO_ERROR;
        } break;
        case CREATE_METADATA_RETRIEVER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            pid_t pid = data.readInt32();
            // 子类要实现 createMetadataRetriever 函数。
            sp<IMediaMetadataRetriever> retriever = createMetadataRetriever(pid);
            reply->writeStrongBinder(retriever->asBinder());
            return NO_ERROR;
        } break;
       default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

通过上面的分析，我们是否能更清晰地理解了图 6-3、图 6-4 和图 6-5 所表达的意义了呢？

6.5 拓展思考

这一节，让我们进一步讨论和思考几个与 Binder 有关的问题。这几个问题和 Binder 的实现有关系。先简单介绍一下与 Binder 驱动相关的内容：

- Binder 的驱动代码在 kernel/drivers/staing/android/binder.c 中，另外该目录下还有一个 binder.h 头文件。Binder 是一个虚拟设备，所以它的代码相比而言还算简单，读者只要有基本的 Linux 驱动开发方面的知识就能读懂它。
- /proc/binder 目录下的内容可用来查看 Binder 设备的运行状况。

6.5.1 Binder 和线程的关系

以 MS 为例，如果现在程序运行正常，此时 MS 则：

1）通过 startThreadPool 启动一个线程，这个线程在 talkWithDriver。

2）主线程通过 joinThreadPool 也在 talkWithDriver。

至此我们知道，有两个线程在和 Binder 设备打交道。这时在业务逻辑上需要与 ServiceManager 交互，比如要调用 listServices 打印所有服务的名字，假设这是 MS 中的第三个线程。按照之前的分析，它最终会调用 IPCThreadState 的 transact 函数，这个函数会 talkWithDriver 并把请求发到 ServiceManager 进程，然后等待来自 Binder 设备的回复。那么现在一共有三个线程（不论是在等待来自其他 Client 的请求，还是在等待 listService 的回复）都在 talkWithDriver。

ServiceManager 处理完了 listServices，把回复结果写回 Binder 驱动，那么，MS 中哪个线程会收到回复呢？此问题如图 6-6 所示：

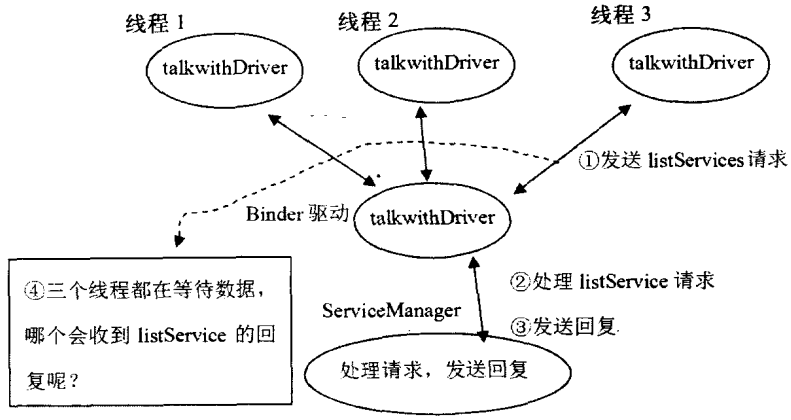


图 6-6 本问题的示意图

显而易见，当然是调用 listServices 的那个线程会得到结果。为什么？因为如果不这么

做，则会导致下面的情况发生：

- 如果是没有调用 listServices 的线程 1 或线程 2 得到回复，那么它们应该唤醒调用 listServices 的线程 3。因为这时已经有了结果，线程 3 应该从 listServices 函数调用中返回。
- 这其中的线程等待、唤醒、切换会浪费不少宝贵的时间片，而且代码逻辑会极其复杂。

看来，Binder 设备把发起请求的线程牢牢地拴住了，必须收到回复才会放它离开。这种一一对应的方式极大简化了代码层的处理逻辑。

说明　*想想 socket 编程吧，同一时刻不能有多个线程操作 socket 的读 / 写，否则数据会乱套。*

另外，我们在分析 executeCommand 的时候，特别提到了 BR_SPAWN_LOOPER 这个 case 的处理，它用于建立新的线程来和 Binder 通信。什么会收到这个消息呢？请读者研究 Binder 的驱动。如果有新发现，请告诉我，大家再一起学习。

6.5.2　有人情味的讣告

在 socket 编程中，如果一个 socket 关闭了，我们会无比希望另一端的 select/poll/epoll/WaitForXXX 有相应的返回，以示通知。

说明　*在广域网中，我们常常会因为收不到或延时收到 socket 的 close 消息而烦恼。*

在 Binder 系统中，要是明确表示了你对 BnXXX 的生死非常关心，那么在它“离世”后你会收到一份讣告。你可以嚎啕大哭，或者什么也不做。

关于这个问题，请直接看源码中的例子吧。

1. 表达你的关心

要想收到讣告，必须先表达你的关心，做下面两件事：

- 从 IBinder::DeathRecipient 派生一个类，并实现其中的通知函数 binderDied。这个函数一旦被调用，就相当于你收到了讣告。
- 把这个类注册到系统中，告诉它你关心哪一个 BnXXX 的生死。

看示例代码，它在 MediaMetadataRetriever.cpp 中，如下所示：

[-->MediaMetadataRetriever.cpp]

```
const sp<IMediaPlayerService>& MediaMetadataRetriever::getService()
{
    Mutex::Autolock lock(sServiceLock);
    if (sService.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
```

```
            sp<IBinder> binder;
            do {
                binder = sm->getService(String16("media.player"));
                if (binder != 0) {
                    break;
                }
                usleep(500000); // 0.5 s
            } while(true);
            if (sDeathNotifier == NULL) {
                sDeathNotifier = new DeathNotifier();
            }
            // 调用下面这个函数，告诉系统我们对这个 binder 的生死有兴趣。
            // 这个 binder 是一个 BpBinder，它关心的是对端 BBinder，也即 BnXXX 的父类。
            binder->linkToDeath(sDeathNotifier);
            sService = interface_cast<IMediaPlayerService>(binder);
        }
        return sService;
    }
```

2. 讣告是怎么收到的

那么，这份讣告是怎么收到的呢？答案也在 executeCommand 中，代码如下所示：

[-->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_DEAD_BINDER:
        {
            //Binder 驱动会通知死亡消息。下面的 proxy 对应着已经死亡的远端 BBinder。
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
             // 发送讣告，Obituary 是讣告的意思。最终会传递到你的 DeathNotifier 中。
             proxy->sendObituary();
             mOut.writeInt32(BC_DEAD_BINDER_DONE);
             mOut.writeInt32((int32_t)proxy);
        } break;
       default:
         result = UNKNOWN_ERROR;
        break;
    }
```

3. 你“死”了，我怎么办

收到讣告后该怎么办呢？有一些代码看起来非常“薄情寡义”，如下所示：

[-->MediaMetadataRetriever.cpp]

```
/*
  DeathNotifier 是 MediaMetadataRetriever 的内部类，前面在 getService 函数中
  我们注册了它对 BnMediaPlayerService 的关心。
*/
void MediaMetadataRetriever::DeathNotifier::binderDied(const wp<IBinder>& who) {
    Mutex::Autolock lock(MediaMetadataRetriever::sServiceLock);
  // 把自己保存的 BpMediaPlayerService 对象干掉！
  MediaMetadataRetriever::sService.clear();
  LOGW("MediaMetadataRetriever server died!");// 打印一下 LOG，这样就完事大吉了。
}
```

4. 承受不住的诺言

虽然我答应收到讣告后给你送终，可是如果我死在你前面或中途我不想接收讣告，又该怎么办呢？ 来看下面的代码：

[-->MediaMetadataRetriever.cpp]

```
MediaMetadataRetriever::DeathNotifier::~DeathNotifier()
{
   Mutex::Autolock lock(sServiceLock);
   // DeathNotifier 对象不想活了，但是 BnMediaPlayerService 还活着，
   // 或者 DeathNotifier 中途变卦。怎么办？
   //unlinkToDeath 调用可以取消对 BnMediaPlayerService 的关心。
    if (sService != 0) {
        sService->asBinder()->unlinkToDeath(this);
    }
}
```

Binder 的这个讣告是不是很有人情味呢？想知道它是怎么做到的吗？还是先去看看驱动的实现吧。

6.5.3 匿名 Service

匿名 Service 就是没有注册的 Service，这句话是什么意思？

- 没有注册意味着这个 Service 没有在 ServiceManager 上注册。
- 它是一个 Service 又表示它确实是一个基于 Binder 通信的 C/S 结构。

再看下面的代码，或许就会明白是什么意思了。

[-->IMediaPlayerService.cpp]

```
status_t BnMediaPlayerService::onTransact(uint32_t code, const Parcel& data,
```

```
                                                    Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CREATE_URL: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
             ...
             //player 是一个 IMediaPlayer 类型的对象。
            sp<IMediaPlayer> player = create(
                    pid, client, url, numHeaders > 0 ? &headers : NULL);
            // 下面这句话也很重要。
            reply->writeStrongBinder(player->asBinder());
            return NO_ERROR;
        } break;
```

当 MediaPlayerClient 调用 create 函数时，MediaPlayerService 会返回一个 IMediaPlayer 对象，此后，MediaPlayerClient 即可直接使用这个 IMediaPlayer 来进行跨进程的函数调用了。

请看，这里确实也存在 C/S 的两端：

- BpMediaPlayer，由 MediaPlayerClient 使用，用来调用 IMediaPlayer 提供的业务服务。
- BnMediaPlayer，由 MediaPlayerService 使用，用来处理来自 Client 端的业务请求。

上面明显是一个 C/S 结构，但在 ServiceManager 中，肯定没有 IMediaPlayer 的信息，那么 BpMediaPlayer 是如何得到 BnMediaPlayer 的 handle 值的呢？

注意 handle 事关通信的目的端，因此它非常重要。

答案可能就在下面这句话中：

```
reply->writeStrongBinder(player->asBinder());// 将 Binder 类型作为一种特殊数据类型处理。
```

当这个 reply 写到 Binder 驱动中时，驱动可能会特殊处理这种 IBinder 类型的数据，例如为这个 BBinder 建立一个独一无二的 handle，这其实相当于在 Binder 驱动中注册了一项服务。

通过这种方式，MS 输出了大量的 Service，例如 IMediaPlayer 和 IMediaRecorder 等。

说明 关于这个问题，也可以查看驱动的实现来验证这一想法。

6.6 学以致用

全书中可能惟有 Binder 系统有如此大的魅力，让我单独用一节来介绍如何使用它。

6.6.1 纯 Native 的 Service

纯 Native 的 Service 表示代码都在 Native 层。Native 层有很多 Service，前面的 MS 不就

是一个重量级的吗？

假设 Service 叫 Test，我们该如何实现呢？完全可以模仿 MS！具体实现过程如代码所示：

[-->Test.cpp:: 范例]

```
int main()
{
   sp<ProcessState> proc(ProcessState::self());
  sp<IServiceManager> sm = defaultServiceManager();
  // 记住注册你的服务，否则谁也找不着你！
  sm->addService( "service.name" ,new Test());
  // 如果压力不大，可以不用单独搞一个线程。
  ProcessState::self()->startThreadPool();
  // 这个是必须的，否则主线程退出了，你也完了。
  IPCThreadState::self()->joinThreadPool();
}
```

Test 是怎么定义的呢？我们是跨进程的 C/S，所以本地需要一个 BnTest，对端需要提供一个代理 BpTest。为了不暴露 Bp 的身份，Bp 的定义和实现都放在 BnTest.cpp 中了。

注意　你虽可以暴露 Bp 的身份（输出它的头文件），但却没有必要，因为客户端用的是基类 ITest 指针。

1. 我能干什么

ITest 接口表明了它所提供的服务，例如 getTest 和 setTest 等，这个与业务逻辑相关，代码如下所示：

说明　getTest 也可以返回一个 ITestService 类型的 Service！

[-->ITest.h:: 声明 ITest]

```
// 需要从 IInterface 派生
class ITest: public IInterface。
{
public:
  // 神奇的宏 DECLARE_META_INTERFACE。
  DECLARE_META_INTERFACE(Test);
  virtual void getTest() = 0;
  virtual void setTest() = 0;
}//ITest 是一个接口类。
```

2. 定义 BnTest 和 BpTest

为了把 ITest 融入到 Binder 系统，需要定义 BnTest 和对客户端透明的 BpTest。BnTest

定义既可以与上面的 Test 定义放在一块，也可以分开，如下所示：

[-->ITest.h:: 声明 BnTest]

```
class BnTest: public BnInterface<ITest>
{
public:
   // 由于 ITest 是个纯虚类，而 BnTest 只实现了 onTransact 函数，所以 BnTest 依然是一个纯虚类。
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);

};
```

另外，我们还要使用 IMPLEMENT 宏。参考 BnMediaPlayerService 的方法，把 BnTest 和 BpTest 的实现都放在 ITest.cpp 中，如下所示：

[-->ITest.cpp::BnTest 的实现]

```
IMPLEMENT_META_INTERFACE(Test, "android.Test.ITest");//IMPLEMENT 宏

status_t BnTest::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case GET_Test: {
           CHECK_INTERFACE(ITest, data, reply);
             getTest();// 子承父业，由 Test 完成。
            return NO_ERROR;
        } break; //SET_XXX 类似。
.......
```

BpTest 也在这里实现吧，如下所示：

[-->ITest.cpp::BpTest 的实现]

```
class BpTest: public BpInterface<ITest>
{
public:
    BpXXX (const sp<IBinder>& impl)
        : BpInterface< ITest >(impl)
    {
}
vitural getTest()
{
Parcel data, reply;
   data.writeInterfaceToken(ITest::getInterfaceDescriptor());
   data.writeInt32(pid);
   // 打包请求数据，然后交给 BpBinder 通信层处理。
```

```
    remote()->transact(GET_Test, data, &reply);
    return;
}
//setTest 类似。
......
```

纯 Native 的 Service 写起来量大一些，上面的代码还只是把 C/S 的框架写好了，真正的业务处理尚未开始，不过感觉却很踏实，很厚重。那么，Java 层的 Service 该怎么写呢？

6.6.2　扶得起的“阿斗”（aidl）

阿斗（aidl 的谐音）本来是扶不起的，可是我们有了 AIDL 工具，就有可能将他扶起！

1. 我能干什么

在 Java 层中，如果想要利用 Binder 进行跨进程的通信，也得定义一个类似 ITest 的接口，不过这是一个 aidl 文件。现在假设服务端程序都在 com.test.service 包中。

ITest.aidl 文件内容如下：

[-->ITest.aidl]

```
package com.test.service;
import com.test.complicatedDataStructure
interface ITest{
  // complicatedDataStructure 类是自己定义的复杂数据结构，in 表示输入参数，out 表示输出参数。
  //in 和 out 的表示一定要准确。切记！
  int getTest(out complicatedDataStructure);
  int setTest(in String name,in boolean reStartServer);
}
```

定义完后，如果用 Eclipse 进行编译，会在 gen 目录下生成一个 com.test.ITest.java 文件（也会生成对应包结构的目录）。内容就不具体罗列了，我们只关注它是如何实现服务端的。

说明　Eclipse 用的也是 aidl 工具，可以手动使用这个工具编译 aidl 文件。

2. 实现服务端

com.test.ITest.java 只是实现了类似 BnTest 的一个东西，具体的业务实现还需从 ITest.Stub 派生，实现代码如下所示：

[-->ITestImpl.java]

```
/*
  ITest.Stub 是在 aidl 生成的那个 Java 文件中定义的，非常类似 Native 层的 BnTest。
  ITestImpl 必须从 ITest.Stub 中派生，用来实现具体的业务函数。
*/
package com.test.service
```

```
class ITestImpl extends ITest.Stub{
 public void getTest(complicatedDataStructure cds)  throws RemoteException {
         //在这里实现具体的getTest。
 }
  public void setTest(in String name,in boolean reStartServer)
   throws RemoteException
  {
   //在这里实现具体的setTest。
  }
}
```

这时，你的 Eclipse 下会有如下两个目录：

- src 下有一个 com.test.service 包结构目录。
- gen 下也有一个 com.test.service 包结构目录，其中的内容是由 aidl 工具生成的。

3. 实现代理端

代理端往往在另外一个程序中使用。假设是 com.test.client 包，把刚才 com.test.service 工程中 gen 下的 com.test.service 目录全部复制到 com.test.client 中了。这样，client 工程也就有两个包结构目录了：

- com.test.client。
- com.test.service。不过这个目录中仅有 aidl 生成的 Java 文件。

服务端一般驻留于 Service 进程，所以可以在 Client 端的 onServiceConnected 函数中获得代理对象，实现代码如下所示：

[-->Client 端示例代码]

```
// 不一定是在onServiceConnected中，但它是比较合适的。
private ServiceConnection serviceConnection = new ServiceConnection() {
        //@Override
        public void onServiceConnected(ComponentName name, IBinder service) {
           if(ITestProxy == null)
               ITestProxy = ITest.Stub.asInterface(service);//这样你就得到BpTest了。
}
```

4. 传递复杂的数据结构

AIDL 支持简单数据结构与 Java 中 String 类型的数据进行跨进程传递，如果想做到跨进程传递复杂的数据结构，还须另做一些工作。

以 ITest.aidl 文件中使用的 complicatedDataStructure 为例：

- 它必须实现 implements Parcelable 接口。
- 内部必须有一个静态的 CREATOR 类。
- 定义一个 complicatedDataStructure.aidl 文件。

说明 可参考 Android API 文档的 parcelable 类，里面有一个很简单的例子。

来看这个 Java 文件的实现：

[-->complicatedDataStructure.java]

```
package com.test.service;
import android.os.Parcel;
import android.os.Parcelable;
public class complicatedDataStructure implements Parcelable {

    public static final int foo1 = 0;
    public static final int foo2 = 1;
    public String fooString1 = null;
    public String fooString2 = null;
    // 静态 Creator 类
public static final Parcelable.Creator< complicatedDataStructure > CREATOR =
      new Parcelable.Creator< complicatedDataStructure >() {
        public complicatedDataStructure createFromParcel(Parcel in)
        {
            return new complicatedDataStructure (in);
        }
        public complicatedDataStructure [] newArray(int size)
        {
          return new complicatedDataStructure [size];// 用于传递数组
        }
    };
    public complicatedDataStructure(complicatedDataStructure other) {
        fooString1 = other. fooString1;
        fooString2 = other. fooString2;
        foo1 = other. foo1;
        foo2 = other. foo2;
   }
    private complicatedDataStructure (Parcel in) {
        readFromParcel(in);
    }
    /* @Override */
    public int describeContents() {
        return 0;
    }
    public void readFromParcel(Parcel in) {
        foo1 = in. readInt ();
        foo2 = in. readInt ();
        fooString1 = in.readString();
        fooString2 = in.readString();
     }
    /* @Override */
    public void writeToParcel(Parcel dest, int flags) {
       dest.writeInt(foo1);
       dest.writeInt(foo2);
```

```
            dest.writeString(fooString1);
            dest.writeString(fooString2);
        }
    }
```

complicatedDataStructure.aidl 该怎么写呢？如下所示：

[-->complicatedDataStructure.aidl]

```
package com.test.service;
parcelable complicatedDataStructure;
```

然后，在使用它的 aidl 文件中添加下行代码即可：

```
import com.test.complicatedDataStructure
```

有了 AIDL，再看我们的阿斗是不是能扶得起了呢？当然，想让上面的程序正确工作，还得再努把力，把未尽的业务层事业完成。另外，还要经得起残酷环境的考验（即通过测试来检验自己的程序）。

6.7 本章小结

本章对 Android 平台中最为重要的 IPC 通信机制 Binder 进行了较为全面、详细的分析。我们首先以 MediaServer 为切入点，讲述了 Android 是如何通过层层封装将 Binder 机制集成到应用程序中的；然后对服务总管 ServiceManager 进行了介绍；最后分析了 MediaPlayerService 是如何通过 Binder 处理来自客户端的请求的。

在拓展思考部分，我们重点讨论了和 Binder 有关的三个问题。要弄清这三个问题，都需要从 Binder 驱动的实现中寻找答案。

考虑到 Binder 的重要性，本章还专门增加了“学以致用”一节，旨在研讨怎么写纯 Native 的 Service，如何通过编写 AIDL 文件来实现 Java 中的 Service。



第7章 深入理解 Audio 系统

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。

- AudioTrack.java(*framework/base/media/java/com/android/media/AudioTrack.java*)
- android_media_track.cpp(*framework/base/core/jni/android_media_track.cpp*)
- MemoryHeapBase(*framework/base/libs/binder/MemoryHeapBase.cpp*)
- MemoryBase.h(*framework/base/include/binder/MemoryBase.h*)
- AudioTrack.cpp(*framework/base/libmedia/AudioTrack.cpp*)
- audio_track_cblk_t 声明 (*framework/base/include/private/media/AudioTrackShared.h*)
- audio_track_cblk_t 定义 (*framework/base/media/libmedia/AudioTrack.cpp*)
- Main_MediaServer.cpp(*framework/base/media/mediaserver/Main_MediaServer.cpp*)
- AudioFlinger.cpp(*framework/base/libs/audioFlinger/AudioFlinger.cpp*)
- AudioHardwareInterface.h(*hardware/libhardware_legacy/include/hardware_legacy/AudioHardwareInterface.h*)
- AudioMixer.cpp(*framework/base/libs/audioflinger/AudioMixer.cpp*)
- AudioSystem.h(*framework/base/include/media/AudioSystem.h*)
- AudioSystem.cpp(*framework/base/media/libmedia/ AudioSystem.cpp*)
- AudioPolicyInterface.h(*hardware/libhardware_legacy/include/hardware_legacy*)
- AudioPolicyManagerBase.cpp(*framework/base/libs/audioflinger/AudioPolicyManagerBase.cpp*)
- AudioService.java(*framework/base/media/java/com/android/media/AudioService.java*)
- Android_media_AudioSystem.cpp(*framework/base/core/Jni/Android_media_AudioSystem.cpp*)

7.1 概述

Audio 系统是 Android 平台的重要组成部分，它主要包括三方面的内容：

- AudioRcorder 和 AudioTrack：这两个类属于 Audio 系统对外提供的 API 类，通过它们可以完成 Android 平台上音频数据的采集和输出任务。
- AudioFlinger：它是 Audio 系统的工作引擎，管理着系统中的输入输出音频流，并承担音频数据的混音，以及读写 Audio 硬件等工作以实现数据的输入输出功能。
- AudioPolicyService：它是 Audio 系统的策略控制中心，具体掌管系统中声音设备的选择和切换、音量控制等功能。

Android 的 Audio 系统是我们分析的第一个具有相当难度的复杂系统。对于这种系统，我采取的学习方法是，以一个常见用例为核心，沿着重要函数调用的步骤逐步进行深入分析。中途若出现所需且不熟悉的知识，则以此为契机，及时学习、思考、研究，当不熟悉的知识逐渐被自己了解掌握时，该系统的真面目也随之清晰了。

下面是破解 Audio 系统的战略步骤：

- 首先，从 API 类的 AudioTrack 开始，从 Java 层到 Native 层一步步了解其工作原理。其中 AudioTrack 和 AudioFlinger 有较多的交互工作，但在这一步中，我们暂时只集中关注 AudioTrack 的流程。
- 提炼上一步中 AudioTrack 和 AudioFlinger 的交互流程，以这个交互流程作为分析 AudioFlinger 的突破口。
- 在前面两个步骤中还会有一些剩余的“抵抗分子”，我们将在 AudioPolicyService 的破解过程中把它们彻底消灭掉。另外，在分析 AudioPolicyService 时，还会通过一个耳机插入事件的处理实例来帮助分析 AudioPolicyService 的工作流程。
- 最后，在本章的拓展部分，我们会介绍一下 AudioFlinger 中 DuplicatingThread 的工作原理。

说明 *在下文中 AudioTrack 被简写为 AT，AudioFlinger 被简写为 AF，AudioPolicyService 被简写为 AP。*

让我们整装上阵，开始代码的征程吧！

7.2 AudioTrack 的破解

AudioTrack 属于 Audio 系统对外提供的 API 类，所以它在 Java 层和 Native 层均有对应的类，先从 Java 层的用例开始。

7.2.1 用例介绍

这个用例很简单，但其中会有一些重要概念，应注意理解。

注意　要了解 AudioTrack Java API 的具体信息，需要仔细阅读 Android API 中的相关文档。阅读 API 文档，是一个快速掌握相关知识的好方法。

[-->AudioTrack API 使用例子（Java 层）]

```
// ① 根据音频数据的特性来确定所要分配的缓冲区的最小 size。
int bufsize =
          AudioTrack.getMinBufferSize(8000,// 采样率：每秒 8000 个点。
          AudioFormat.CHANNEL_CONFIGURATION_STEREO,// 声道数：双声道。
          AudioFormat.ENCODING_PCM_16BIT// 采样精度：一个采样点 16 比特，相当于 2 个字节。
);

// ② 创建 AudioTrack。
AudioTrack trackplayer = new AudioTrack(
           AudioManager.STREAM_MUSIC, // 音频流类型
           8000, AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
           AudioFormat.ENCODING_PCM_16BIT, bufsize,
           AudioTrack.MODE_STREAM // 数据加载模式 );
// ③ 开始播放。
trackplayer.play() ;

......
// ④ 调用 write 写数据。
trackplayer.write(bytes_pkg, 0, bytes_pkg.length) ;// 往 track 中写数据。
......
// ⑤ 停止播放和释放资源。
trackplayer.stop();// 停止播放。
trackplayer.release();// 释放底层资源。
```

上面的用例引入了两个新的概念，一个是数据加载模式，另一个是音频流类型。下面进行详细介绍。

1. AudioTrack 的数据加载模式

AudioTrack 有两种数据加载模式：MODE_STREAM 和 MODE_STATIC，它们对应着两种完全不同的使用场景。

- MODE_STREAM：在这种模式下，通过 write 一次次把音频数据写到 AudioTrack 中。这和平时通过 write 系统调用往文件中写数据类似，但这种工作方式每次都需要把数据从用户提供的 Buffer 中拷贝到 AudioTrack 内部的 Buffer 中，这在一定程度上会使引起延时。为了解决这一问题，AudioTrack 就引入了第二种模式。
- MODE_STATIC：这种模式下，在 play 之前只需要把所有的数据通过一次 write 调用传递到 AudioTrack 的内部缓冲区中，后续就不必再传递数据了。这种模式适用于像铃声这种内存占用量较小，延时要求较高的文件。但它也有一个缺点，就是一次 write 的数据不能太多，否则系统无法分配足够的内存来存储全部数据。

这两种模式中 MODE_STREAM 模式相对更常见，并且也更复杂，我们的分析将以它为主。

注意 如果采用 STATIC 模式，须先调用 write 写数据，然后再调用 play。

2. 音频流的类型

在 AudioTrack 的构造函数中，会接触到 AudioManager.STREAM_MUSIC 这个参数。它的含义与 Android 系统对音频流的管理和分类有关。

Android 将系统的声音分为好几种流类型，下面是几个常见的：

- STREAM_ALARM：警告声
- STREAM_MUSIC：音乐声，例如 music 等
- STREAM_RING：铃声
- STREAM_SYSTEM：系统声音，例如低电提示音、锁屏音等
- STREAM_VOCIE_CALL：通话声

注意 上面这些类型的划分和音频数据本身并没有关系。例如 MUSIC 和 RING 类型都可以是某首 MP3 歌曲。另外，声音流类型的选择没有固定的标准，例如，铃声预览中的铃声可以设置为 MUSIC 类型。

音频流类型的划分和 Audio 系统对音频的管理策略有关。其具体的作用在以后的分析中再详细介绍。在目前的用例中，把它当做一个普通数值即可。

3. Buffer 分配和 Frame 的概念

在用例中碰到的第一个重要函数就是 getMinBufferSize。这个函数对于确定应用层分配多大的数据 Buffer 具有重要指导意义。先回顾一下它的调用方式：

[-->AudioTrack API 使用例子（Java 层）]

```
// 注意这些参数的值。想象我们正在一步步的 Trace，这些参数都会派上用场。
AudioTrack.getMinBufferSize(8000,// 每秒 8000 个点
         AudioFormat.CHANNEL_CONFIGURATION_STEREO,// 双声道
         AudioFormat.ENCODING_PCM_16BIT);
```

来看这个函数的实现：

[-->AudioTrack.java]

```
static public int getMinBufferSize(int sampleRateInHz, int channelConfig,
                                   int audioFormat) {
        int channelCount = 0;
        switch(channelConfig) {
        case AudioFormat.CHANNEL_OUT_MONO:
        case AudioFormat.CHANNEL_CONFIGURATION_MONO:
            channelCount = 1;
            break;
        case AudioFormat.CHANNEL_OUT_STEREO:
        case AudioFormat.CHANNEL_CONFIGURATION_STEREO:
```

```
            channelCount = 2;// 目前最多支持双声道。
            break;
        default:
            return AudioTrack.ERROR_BAD_VALUE;
        }
        // 目前只支持 PCM8 和 PCM16 精度的音频数据。
        if ((audioFormat != AudioFormat.ENCODING_PCM_16BIT)
            && (audioFormat != AudioFormat.ENCODING_PCM_8BIT)) {
            return AudioTrack.ERROR_BAD_VALUE;
        }
      // 对采样频率也有要求，太低或太高都不行。
        if ( (sampleRateInHz < 4000) || (sampleRateInHz > 48000) )
            return AudioTrack.ERROR_BAD_VALUE;

        /*
          调用 Native 函数，先想想为什么，如果是简单计算，那么 Java 层做不到吗？
          原来，还需要确认硬件是否支持这些参数，当然得进入 Native 层查询了。
        */
        int size = native_get_min_buff_size(sampleRateInHz,
                  channelCount, audioFormat);
        if ((size == -1) || (size == 0)) {
            return AudioTrack.ERROR;
        }
        else {
            return size;
        }
    }
```

Native 的函数将查询 Audio 系统中音频输出硬件 HAL 对象的一些信息，并确认它们是否支持这些采样率和采样精度。

说明　HAL 对象的具体实现和硬件厂商有关系，如果没有特殊说明，我们则把硬件和 HAL 作为一种东西讨论。

来看 Native 的 native_get_min_buff_size 函数，它在 android_media_track.cpp 中。

[-->android_media_track.cpp]

```
/*
注意我们传入的参数是：

sampleRateInHertz = 8000, nbChannels = 2
audioFormat = AudioFormat.ENCODING_PCM_16BIT
*/
static jint android_media_AudioTrack_get_min_buff_size(
          JNIEnv *env,  jobject thiz,
          jint sampleRateInHertz, jint nbChannels, jint audioFormat)
{
    int afSamplingRate;
```

```
    int afFrameCount;
    uint32_t afLatency;
     /*
        下面这些调用涉及了 AudioSystem，这个和 AudioPolicy 有关系。这里仅把它们看成是
        信息查询即可。
    */
    // 查询采样率，一般返回的是所支持的最高采样率，例如 44100。
    if (AudioSystem::getOutputSamplingRate(&afSamplingRate) != NO_ERROR) {
        return -1;
}
     // ① 查询硬件内部缓冲的大小，以 Frame 为单位。什么是 Frame？
    if (AudioSystem::getOutputFrameCount(&afFrameCount) != NO_ERROR) {
        return -1;
    }
    // 查询硬件的延时时间。
    if (AudioSystem::getOutputLatency(&afLatency) != NO_ERROR) {
        return -1;
}
......
```

这里有必要插入一个说明，因为代码中出现了音频系统中的一个重要概念：Frame（帧）。

说明 Frame 是一个单位，经多方查寻，最终在 ALSA 的 wiki 中找到了对它的解释。Frame 被用来直观地描述数据量的多少，例如，一帧等于多少字节。1 单位的 Frame 等于 1 个采样点的字节数 × 声道数（比如 PCM16，双声道的 1 个 Frame 等于 2×2=4 字节）。

我们知道，1 个采样点只针对一个声道，而实际上可能会有一或多个声道。由于不能用一个独立的单位来表示全部声道一次采样的数据量，也就引出了 Frame 的概念。Frame 的大小，就是一个采样点的字节数 × 声道数。另外，在目前的声卡驱动程序中，其内部缓冲区也是采用 Frame 作为单位来分配和管理的。

OK，继续 native_get_min_buff_size 函数。

```
    ......
    // minBufCount 表示缓冲区的最少个数，它以 Frame 作为单位。
     uint32_t minBufCount = afLatency / ((1000 * afFrameCount)/afSamplingRate);
     if (minBufCount < 2) minBufCount = 2;// 至少要两个缓冲。

    // 计算最小帧个数。
    uint32_t minFrameCount =
            (afFrameCount*sampleRateInHertz*minBufCount)/afSamplingRate;
  // 下面根据最小的 FrameCount 计算最小的缓冲大小。
     int minBuffSize = minFrameCount // 计算方法完全符合我们前面关于 Frame 的介绍。
             * (audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1)
             * nbChannels;

     return minBuffSize;
}
```

getMinBufSize 会在综合考虑硬件的情况（诸如是否支持采样率，硬件本身的延迟情况等）后，得出一个最小缓冲区的大小。一般我们分配的缓冲大小会是它的整数倍。

好了，介绍完一些基本概念后，开始要分析 AudioTrack 了。

7.2.2　AudioTrack（Java 空间）分析

注意　Java 空间的分析包括 JNI 这一层，因为它们二者的关系最为紧密。

1. AudioTrack 的构造

回顾一下用例中调用 AudioTrack 构造函数的代码：

```
AudioTrack trackplayer = new AudioTrack(
          AudioManager.STREAM_MUSIC,
          8000, AudioFormat.CHANNEL_CONFIGURATION_ STEREO,
          AudioFormat.ENCODING_PCM_16BIT,bufsize,
          AudioTrack.MODE_STREAM);
```

AudioTrack 构造函数的实现在 AudioTrack.java 中。来看这个函数：

[-->AudioTrack.java]

```
public AudioTrack(int streamType, int sampleRateInHz, int channelConfig,
          int audioFormat,int bufferSizeInBytes, int mode)
           throws IllegalArgumentException {

           mState = STATE_UNINITIALIZED;
        // 检查参数是否合法。
        audioParamCheck(streamType, sampleRateInHz, channelConfig,
                        audioFormat, mode);
       //bufferSizeInBytes 是通过 getMinBufferSize 得到的，所以下面的检查肯定能通过。
        audioBuffSizeCheck(bufferSizeInBytes);

        /*
           调用 native 层的 native_setup，构造一个 WeakReference 传进去。
           不了解 Java WeakReference 的读者可以上网查一下，很简单。
       */
         int initResult = native_setup(new WeakReference<AudioTrack>(this),
         mStreamType, // 这个值是 AudioManager.STREAM_MUSIC。
         mSampleRate, // 这个值是 8000。
         mChannels,     // 这个值是 2。
         mAudioFormat,// 这个值是 AudioFormat.ENCODING_PCM_16BIT。
         mNativeBufferSizeInBytes,// 这个值等于 bufferSizeInBytes。
         mDataLoadMode);//DataLoadMode 是 MODE_STREAM。
         ....
}
```

OK，native_setup 对应的 JNI 层函数是 android_media_AudioTrack_native_setup。一起

来看看：

[-->android_media_AudioTrack.cpp]

```
static int
android_media_AudioTrack_native_setup(JNIEnv *env, jobject thiz,
                  jobject weak_this,jint streamType,
                  jint sampleRateInHertz, jint channels,
                  jint audioFormat, jint buffSizeInBytes,
                  jint memoryMode)
{
    int afSampleRate;
    int afFrameCount;
    //进行一些信息查询。
   AudioSystem::getOutputFrameCount(&afFrameCount, streamType);
   AudioSystem::getOutputSamplingRate(&afSampleRate, streamType);
   AudioSystem::isOutputChannel(channels);
   //popCount 用于统计一个整数中有多少位为 1，有很多经典的算法。
   int nbChannels = AudioSystem::popCount(channels);
    //Java 层的值和 JNI 层的值转换。
    if (streamType == javaAudioTrackFields.STREAM_MUSIC)
         atStreamType = AudioSystem::MUSIC;

   int bytesPerSample = audioFormat == javaAudioTrackFields.PCM16 ? 2 : 1;
   int format = audioFormat == javaAudioTrackFields.PCM16 ?
                  AudioSystem::PCM_16_BIT : AudioSystem::PCM_8_BIT;

    // 计算以帧为单位的缓冲大小。
    int frameCount = buffSizeInBytes / (nbChannels * bytesPerSample);

     // ① AudioTrackJniStorage 对象，它保存了一些信息，后面将详细分析。
    AudioTrackJniStorage* lpJniStorage = new AudioTrackJniStorage();
      ......
     // ②创建 Native 层的 AudioTrack 对象。
    AudioTrack* lpTrack = new AudioTrack();
       if (memoryMode == javaAudioTrackFields.MODE_STREAM) {
            // ③ STREAM 模式
            lpTrack->set(
                atStreamType,// 指定流类型。
                sampleRateInHertz,
                format,// 采样点的精度，一般为 PCM16 或 PCM8。
                channels,
                frameCount,
                0,// flags
                audioCallback, // 该回调函数定义在 android_media_AudioTrack.cpp 中。
                &(lpJniStorage->mCallbackData),
                0,
                0,// 共享内存，STREAM 模式下为空。实际使用的共享内存由 AF 创建。
                true);// 内部线程可以调用 JNI 函数，还记得“zygote 偷梁换柱”那一节吗？
        } else if (memoryMode == javaAudioTrackFields.MODE_STATIC) {
```

```
        // 如果是 static 模式，需要先创建共享内存。
        lpJniStorage->allocSharedMem(buffSizeInBytes);
        lpTrack->set(
          atStreamType,// stream type
          sampleRateInHertz,
          format,// word length, PCM
          channels,
          frameCount,
          0,// flags
          audioCallback,
          &(lpJniStorage->mCallbackData),
          0,
          lpJniStorage->mMemBase, //STATIC 模式下，需要传递该共享内存。
          true);
    }

    ......
    /*
      把 JNI 层中 new 出来的 AudioTrack 对象指针保存到 Java 对象的一个变量中，
      这样就把 JNI 层的 AudioTrack 对象和 Java 层的 AudioTrack 对象关联起来了，
      这就是 Android 的常用技法。
   */
     env->SetIntField(thiz,javaAudioTrackFields.nativeTrackInJavaObj,
              (int)lpTrack);
    // lpJniStorage 对象指针也保存到 Java 对象中。
     env->SetIntField(thiz, javaAudioTrackFields.jniData, (int)lpJniStorage);
  }
```

上面的代码列出了三个要点（即①～③），这一节仅分析 AudioTrackJniStorage 这个类，其余的作为 Native AudioTrack 部分放在后面进行分析。

2. AudioTrackJniStorage 分析

AudioTrackJniStorage 是一个辅助类，其中有一些有关共享内存方面的较重要的知识，这里先简单介绍一下。

（1）共享内存介绍

共享内存，作为进程间数据传递的一种手段，在 AudioTrack 和 AudioFlinger 中被大量使用。先简单了解一下有关共享内存的知识：

- 每个进程的内存空间是 4GB，这个 4GB 是由指针长度决定的，如果指针长度为 32 位，那么地址的最大编号就是 0xFFFFFFFF，为 4GB。
- 上面说的内存空间是进程的虚拟地址空间。换言之，在应用程序中使用的指针其实是指向虚拟空间地址的。那么，如何通过这个虚拟地址找到存储在真实物理内存中的数据呢？

上面的问题引出了内存映射的概念。内存映射让虚拟空间中的内存地址和真实物理内存地址之间建立了一种对应关系。也就是说，进程中操作的 0x12345678 这块内存地址，在经

过 OS 内存管理机制的转换后，它实际对应的物理地址可能会是 0x87654321。当然，这一切对进程来说都是透明的，这些活都由操作系统悄悄地完成了。不过，这会和我们的共享内存有什么关系吗？

当然有，共享内存和内存映射有着重要关系。来看图 7-1“共享内存示意图”：

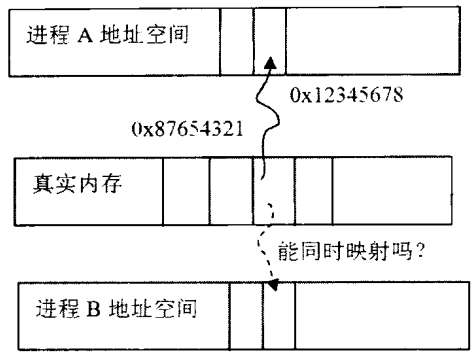


图 7-1 共享内存示意图

图 7-1 提出了一个关键性的问题，即真实内存中 0x87654321 标示的这块内存页（OS 的内存管理机制将物理内存分成了一个个的内存页，一块内存页的大小一般是 4KB）现在已经映射到了进程 A 中。可它能同时映射到进程 B 中吗？如果能，那么在进程 A 中对这块内存页所写的数据在进程 B 中就能看见了，这岂不就成了内存在两个进程间共享吗？

事实确实如此，否则我们的生活就不会像现在这么美好了。这个机制是由操作系统提供和实现的，原理很简单，实现起来却很复杂，这里就不深究了。

如何创建和共享内存呢？不同的系统会有不同的方法。Linux 平台的一般做法是：

- 进程 A 创建并打开一个文件，得到一个文件描述符 fd。
- 通过 mmap 调用将 fd 映射成内存映射文件。在 mmap 调用中指定特定的参数表示要创建进程间共享内存。
- 进程 B 打开同一个文件，也得到一个文件描述符，这样 A 和 B 就打开了同一个文件。
- 进程 B 也要用 mmap 调用指定参数表示想使用共享内存，并传递打开的 fd。这样 A 和 B 就通过打开同一个文件并构造内存映射，实现了进程间的内存共享。

注意 *这个文件也可以是设备文件。一般来说，mmap 函数的具体工作由参数中的那个文件描述符所对应的驱动或内核模块来完成。*

除上述的一般方法外，Linux 还有 System V 的共享内存创建方法，这里就不再介绍了。总之，AT 和 AF 之间的数据传递，就是通过共享内存方式来完成的。这种方式对于跨进程的大数据量传输来说，是非常高效的。

（2）MemoryHeapBase 和 MemoryBase 类介绍

AudioTrackJniStorage 用到了 Android 对共享内存机制所设置的封装类。所以我们有必

要先看看 AudioTrackJniStorage 的内容。

[-->android_media_AudioTrack.cpp::AudioTrackJniStorage 相关]

```
// 下面这个结构就是保存一些变量，没有什么特别的作用。
struct audiotrack_callback_cookie {
    jclass      audioTrack_class;
    jobject     audioTrack_ref;
 };
class AudioTrackJniStorage {
    public:
        sp<MemoryHeapBase>      mMemHeap;// 这两个 Memory 很重要。
        sp<MemoryBase>          mMemBase;

        audiotrack_callback_cookie mCallbackData;
        int                        mStreamType;

      bool allocSharedMem(int sizeInBytes) {
       /*
        注意关于 MemoryHeapBase 和 MemoryBase 的用法。
        先 new 一个 MemoryHeapBase，再以它为参数 new 一个 MemoryBase。
       */
      // ① MemoryHeapBase
       mMemHeap = new MemoryHeapBase(sizeInBytes, 0, "AudioTrack Heap Base");
      // ② MemoryBase
       mMemBase = new MemoryBase(mMemHeap, 0, sizeInBytes);

       return true;
      }
};
```

注意代码中标识①和②的地方，它们很好地展示了这两个 Memory 类的用法。在介绍它们之前，先来看图 7-2 中与这两个 Memory 有关的家谱。

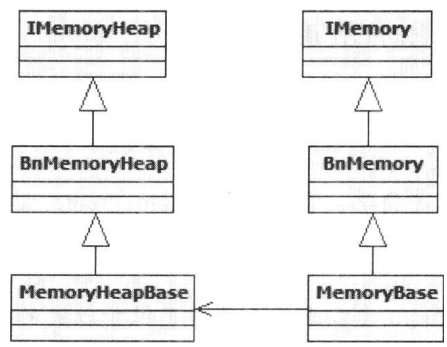


图 7-2　MemoryHeapBase 和 MemoryBase 的家谱

MemoryHeapBase 是一个基于 Binder 通信的类，根据前面所讲的 Binder 知识可知，

BpMemoryHeapBase 由客户端使用，而 MemoryHeapBase 完成 BnMemoryHeapBase 的业务工作。

从 MemoryHeapBase 开始分析。它的使用方法是：

```
mMemHeap = new MemoryHeapBase(sizeInBytes, 0, "AudioTrack Heap Base");
```

它的代码在 MemoryHeapBase.cpp 中，如下所示。

[-->MemoryHeapBase.cpp]

```
/*
   MemoryHeapBase 有两个构造函数，我们用的是第一个。
  size 表示共享内存的大小，flags 为 0，name 为 "AudioTrack Heap Base"。
*/
MemoryHeapBase::MemoryHeapBase(size_t size, uint32_t flags,char const * name)
    : mFD(-1), mSize(0), mBase(MAP_FAILED), mFlags(flags),
      mDevice(0), mNeedUnmap(false)
{
    const size_t pagesize = getpagesize();// 获取系统中的内存页大小，一般为 4KB。
    size = ((size + pagesize-1) & ~(pagesize-1));
   /*
     创建共享内存，ashmem_create_region 函数由 libcutils 提供。
     在真实设备上将打开 /dev/ashmem 设备得到一个文件描述符，在模拟器上则创建一个 tmp 文件。
  */
    int fd = ashmem_create_region(name == NULL ? "MemoryHeapBase" : name, size);
  // 下面这个函数将通过 mmap 方式得到内存地址，这是 Linux 的标准做法，有兴趣的读者可以看看。
   mapfd(fd, size);
}
```

MemoryHeapBase 构造完后，得到了以下结果：

- mBase 变量指向共享内存的起始位置。
- mSize 是所要求分配的内存大小。
- mFd 是 ashmem_create_region 返回的文件描述符。

另外，MemoryHeapBase 提供了以下几个函数，可以获取共享内存的大小和位置。由于这些函数都很简单，这里仅把它们的作用描述一下。

```
MemoryHeapBase::getBaseID() //返回 mFd，如果为负数，表明刚才创建共享内存失败了。
MemoryHeapBase::getBase()   //共享内存起始地址。
MemoryHeapBase::getSize() //返回 mSize，表示内存大小。
```

MemoryHeapBase 确实比较简单，它通过 ashmem_create_region 得到一个文件描述符。

说明 Android 系统通过 ashmem 创建共享内存的原理，和 Linux 系统中通过打开文件创建共享内存的原理类似，但 ashmem 设备驱动在这方面做了较大的改进，例如增加了引用计数、延时分配物理内存的机制（即真正使用的时候才去分配内存）等。这些内容，感兴趣的读者可以自行研究。

那么，MemoryBase 是何物？它又有什么作用？

MemoryBase 也是一个基于 Binder 通信的类，它比起 MemoryHeapBase 来就更显得简单了，看起来更像是一个辅助类。它的声明在 MemoryBase.h 中。一起来看：

[-->MemoryBase.h::MemoryBase 声明]

```
class MemoryBase : public BnMemory
{
public:
    MemoryBase(const sp<IMemoryHeap>& heap,ssize_t offset, size_t size);
    virtual sp<IMemoryHeap> getMemory(ssize_t* offset, size_t* size) const;

protected:
    size_t getSize() const { return mSize; }// 返回大小
    ssize_t getOffset() const { return mOffset;}// 返回偏移量
    // 返回 MemoryHeapBase 对象
    const sp<IMemoryHeap>& getHeap() const { return mHeap;}
};
//MemoryBase 的构造函数
MemoryBase::MemoryBase(const sp<IMemoryHeap>& heap,ssize_t offset, size_t size)
    : mSize(size), mOffset(offset), mHeap(heap)
{
}
```

MemoryHeapBase 和 MemoryBase 都够简单吧？总结起来不过是：

- 分配了一块共享内存，这样两个进程可以共享这块内存。
- 基于 Binder 通信，这样使用这两个类的进程就可以交互了。

这两个类在后续的讲解过程中会频繁碰到，但不必对它们做深入分析，只需把它当成普通的共享内存看待即可。

提醒　这两个类没有提供同步对象来保护这块共享内存，所以后续在使用这块内存时，必然需要一个跨进程的同步对象来保护它。这一点，是我在 AT 中第一次见到它们时想到的，不知道你是否注意过这个问题。

3. play 和 write 分析

还记得用例中的③和④关键代码行吗？

```
// ③ 开始播放
trackplayer.play() ;
// ④ 调用 write 写数据
trackplayer.write(bytes_pkg, 0, bytes_pkg.length) ;// 往 track 中写数据
```

现在就来分析它们。我们要直接转向 JNI 层来进行分析。相信你现在已有能力从 Java 层直接跳转至 JNI 层了。

（1）play 分析

先看看 play 函数对应的 JNI 层函数，它是 android_media_AudioTrack_start。

[-->android_media_AudioTrack.cpp]

```
static void
android_media_AudioTrack_start(JNIEnv *env, jobject thiz)
{
/*
  从 Java 的 AudioTrack 对象中获取对应 Native 层的 AudioTrack 对象指针。
  从 int 类型直接转换成指针，不过要是以后 ARM 平台支持 64 位指针了，代码就得大修改了。
*/
    AudioTrack *lpTrack = (AudioTrack *)env->GetIntField(
          thiz, javaAudioTrackFields.nativeTrackInJavaObj);
    lpTrack->start(); //很简单的调用。
}
```

play 函数太简单了，至于它调用的 start，等到 Native 层进行 AudioTrack 分析时，我们再去观察。

（2）write 分析

Java 层的 write 函数有两个：

- 一个是用来写 PCM16 数据的，它对应的一个采样点的数据量是两个字节。
- 另外一个是用来写 PCM8 数据的，它对应的一个采样点的数据量是一个字节。

我们的用例中采用的是 PCM16 数据。它对应的 JNI 层函数是 android_media_AudioTrack_native_write_short，一起来看：

[-->android_media_AudioTrack.cpp]

```
static jint android_media_AudioTrack_native_write_short(
          JNIEnv *env,  jobject thiz,
          jshortArray javaAudioData,jint offsetInShorts,
          jint sizeInShorts,jint javaAudioFormat) {

   return (android_media_AudioTrack_native_write(
          env, thiz,(jbyteArray)javaAudioData,offsetInShorts*2,
          sizeInShorts*2,javaAudioFormat) / 2);
}
```

无论 PCM16 还是 PCM8 数据，最终都会调用 writeToTrack 函数。代码如下所示：

[-->android_media_AudioTrack.cpp]

```
jint writeToTrack(AudioTrack* pTrack, jint audioFormat,
          jbyte* data,jint offsetInBytes, jint sizeInBytes) {

   ssize_t written = 0;
  /*
     如果是 STATIC 模式，sharedBuffer() 返回不为空。
```

```
        如果是 STREAM 模式，sharedBuffer() 返回空。
    */
        if (pTrack->sharedBuffer() == 0) {
            // 我们的用例是 STREAM 模式，调用 write 函数写数据。
            written = pTrack->write(data + offsetInBytes, sizeInBytes);
        } else {
            if (audioFormat == javaAudioTrackFields.PCM16) {
                if ((size_t)sizeInBytes > pTrack->sharedBuffer()->size()) {
                    sizeInBytes = pTrack->sharedBuffer()->size();
                }
                // 在 STATIC 模式下，直接把数据 memcpy 到共享内存，记住在这种模式下要先调用 write，
                // 后调用 play。
                memcpy(pTrack->sharedBuffer()->pointer(),
                       data + offsetInBytes, sizeInBytes);
                written = sizeInBytes;
            } else if (audioFormat == javaAudioTrackFields.PCM8) {
                // 如果是 PCM8 数据，则先转换成 PCM16 数据再拷贝。
                ......
            }
        return written;
    }
```

看上去 play 和 write 这两个函数还真是比较简单，须知，大部分工作还都是由 Native 的 AudioTrack 来完成的。继续 Java 层的分析。

4. release 分析

如果数据都 write 完了，则需要调用 stop 停止播放，或者直接调用 release 来释放相关资源。由于 release 和 stop 有一定的相关性，这里只分析 release 调用。代码如下所示：

[-->android_media_AudioTrack.cpp]

```
static void android_media_AudioTrack_native_release(JNIEnv *env,  jobject thiz) {

    // 调用 android_media_AudioTrack_native_finalize 真正释放资源。
    android_media_AudioTrack_native_finalize(env, thiz);
    // 之前保存在 Java 对象中的指针变量此时都要设置为零。
    env->SetIntField(thiz, javaAudioTrackFields.nativeTrackInJavaObj, 0);
    env->SetIntField(thiz, javaAudioTrackFields.jniData, 0);
}
```

[-->android_media_AudioTrack.cpp]

```
static void android_media_AudioTrack_native_finalize(JNIEnv *env, jobject thiz) {
    AudioTrack *lpTrack = (AudioTrack *)env->GetIntField(
                          thiz, javaAudioTrackFields.nativeTrackInJavaObj);
    if (lpTrack) {
        lpTrack->stop();// 调用 stop
        delete lpTrack; // 调用 AudioTrack 的析构函数
```

```
    }
    ......
}
```

扫尾工作也很简单，没什么需要特别注意的。

至此，在 Java 空间的分析工作就完成了。但在进入 Native 空间的分析之前，要总结一下 Java 空间使用 Native 的 AudioTrack 的流程，只有这样，在进行 Native 空间分析时才能有章可循。

5. AudioTrack（Java 空间）的分析总结

AudioTrack 在 JNI 层使用了 Native 的 AudioTrack 对象，总结一下调用 Native 对象的流程：

- new 一个 AudioTrack，使用无参的构造函数。
- 调用 set 函数，把 Java 层的参数传进去，另外还设置了一个 audiocallback 回调函数。
- 调用 AudioTrack 的 start 函数。
- 调用 AudioTrack 的 write 函数。
- 工作完毕后，调用 stop。
- 最后就是 Native 对象的 delete。

说明 为什么要总结流程呢？

第一：控制了流程，就把握了系统工作的命脉，这一点至关重要。

第二：有些功能的实现纵跨 Java/Native 层，横跨两个进程，这中间有很多封装、很多的特殊处理，但是其基本流程是不变的。通过精简流程，我们才能把注意力集中在关键点上。

7.2.3 AudioTrack（Native 空间）分析

1. new AudioTrack 和 set 分析

Native 的 AudioTrack 代码在 AudioTrack.cpp 中。这一节将分析它的构造函数和 set 调用。

[-->AudioTrack.cpp]

```
AudioTrack::AudioTrack()// 我们使用无参构造函数。
    : mStatus(NO_INIT)
{
  // 把状态初始化成 NO_INIT。Android 的很多类都采用了这种状态控制。
}
```

再看看 set 调用，这个函数有很多内容。代码如下所示：

[-->AudioTrack.cpp]

```
/*
   还记得我们传入的参数吗？
```

```
  streamType=STREAM_MUSIC,sampleRate=8000,format=PCM_16
  channels=2, frameCount 由计算得来，可以假设一个值，例如 1024，不影响分析。
  flags=0, cbf=audiocallback, user 为 cbf 的参数，notificationFrames=0
  因为是流模式，所以 sharedBuffer=0。threadCanCallJava 为 true。
*/
status_t AudioTrack::set(int streamType, uint32_t sampleRate,int format,
        int channels,int frameCount,uint32_t flags,callback_t cbf,void* user,
        int notificationFrames,const sp<IMemory>& sharedBuffer,
        bool threadCanCallJava)
{
    // 前面有一些判断，都是和 AudioSystem 有关的，以后再分析。
    ......
  /*
     audio_io_handle_t 是一个 int 类型，通过 typedef 定义，这个值的来历非常复杂，
     涉及 AudioFlinger 和 AudioPolicyService，后面会试着将其解释清楚。
     这个值主要被 AudioFlinger 使用，用来表示内部的工作线程索引号。AudioFlinger 会根据
     情况创建几个工作线程，下面的 AudioSystem::getOutput 会根据流类型等其他参数最终选
     取一个合适的工作线程，并返回它在 AF 中的索引号。
     而 AudioTrack 一般使用混音线程 (Mixer Thread)。
 */
    audio_io_handle_t output = AudioSystem::getOutput(
                (AudioSystem::stream_type)streamType,
                       sampleRate, format, channels,
                (AudioSystem::output_flags)flags);
   // 调用 creatTrack
    status_t status = createTrack(streamType, sampleRate, format, channelCount,
                                  frameCount, flags, sharedBuffer, output);


//cbf 是 JNI 层传入的回调函数 audioCallback，如果用户设置了回调函数，则启动一个线程。
  if (cbf != 0) {
        mAudioTrackThread = new AudioTrackThread(*this, threadCanCallJava);
   }
   return NO_ERROR;
}
```

再来看 createTrack 函数，代码如下所示：

[-->AudioTrack.cpp]

```
status_t AudioTrack::createTrack(int streamType,uint32_t sampleRate,
        int format,int channelCount,int frameCount, uint32_t flags,
        const sp<IMemory>& sharedBuffer, audio_io_handle_t output)
{
   status_t status;

  /*
    得到 AudioFlinger 的 Binder 代理端 BpAudioFlinger。
    关于这部分内容，我们已经很熟悉了，以后的讲解会跨过 Binder，直接分析 Bn 端的实现。
  */
  const sp<IAudioFlinger>& audioFlinger = AudioSystem::get_audio_flinger();
```

```
    /*
        向 AudioFinger 发送 createTrack 请求。注意其中的几个参数，
        在 STREAM 模式下 sharedBuffer 为空。
        output 为 AudioSystem::getOutput 得到一个值，代表 AF 中的线程索引号。
        该函数返回 IAudioTrack（实际类型是 BpAudioTrack）对象，后续 AF 和 AT 的交互就是
        围绕 IAudioTrack 进行的。
    */
        sp<IAudioTrack> track = audioFlinger->createTrack(getpid(),
            streamType,sampleRate,format,channelCount,frameCount,
             ((uint16_t)flags) << 16,sharedBuffer,output,&status);

       /*
         在 STREAM 模式下，没有在 AT 端创建共享内存，但前面提到了 AT 和 AF 的数据交互是
        通过共享内存完成的，这块共享内存最终由 AF 的 createTrack 创建。我们以后分析 AF 时
        再做介绍。下面这个调用会取出 AF 创建的共享内存。
    */
        sp<IMemory> cblk = track->getCblk();
        mAudioTrack.clear();//sp 的 clear
        mAudioTrack = track;
        mCblkMemory.clear();
        mCblkMemory = cblk;//cblk 是 control block 的简写。
      /*
        IMemory 的 pointer 在此处将返回共享内存的首地址，类型为 void*，
        static_cast 直接把这个 void* 类型转成 audio_track_cblk_t，表明这块内存的首部中存在
         audio_track_cblk_t 这个对象。
      */
        mCblk = static_cast<audio_track_cblk_t*>(cblk->pointer());
        mCblk->out = 1;//out 为 1 表示输出，out 为 0 表示输入。
        mFrameCount = mCblk->frameCount;
        if (sharedBuffer == 0) {
            //buffers 指向数据空间，它的起始位置是共享内存的首部加上 audio_track_cblk_t 的大小。
            mCblk->buffers = (char*)mCblk + sizeof(audio_track_cblk_t);
      } else {
            //STATIC 模式下的处理
             mCblk->buffers = sharedBuffer->pointer();
             mCblk->stepUser(mFrameCount);// 更新数据位置，后面要分析 stepUser 的作用。
        }
        return NO_ERROR;
    }
```

（1）IAudioTrack 和 AT、AF 的关系

上面的 createTrack 函数中突然冒出来一个新面孔，叫 IAudioTrack。关于它和 AT 及 AF 的关系，我们用图 7-3 来表示：

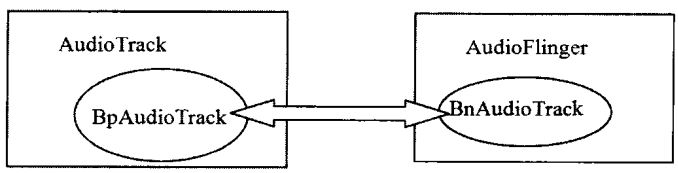


图 7-3 IAudioTrack 和 AT、AF 的关系

从图 7-3 中可以发现：

IAudioTrack 是联系 AT 和 AF 的关键纽带。

至于 IAudioTrack 在 AF 端到底是什么，在分析 AF 时会有详细解释。

（2）共享内存及其 Control Block

通过前面的代码分析，我们发现 IAudioTrack 中有一块共享内存，其头部是一个 audio_track_cblk_t（简称 CB）对象，在该对象之后才是数据缓冲。这个 CB 对象有什么作用呢？

还记得前面提到的那个深层次思考的问题吗？即 MemoryHeapBase 和 MemoryBase 都没有提供同步对象，那么，AT 和 AF 作为典型的数据生产者和消费者，如何正确协调二者生产和消费的步调呢？

Android 为顺应民意，便创造出了这个 CB 对象，其主要目的就是协调和管理 AT 和 AF 二者数据生产和消费的步伐。先来看 CB 都管理些什么内容。它的声明在 AudioTrackShared.h 中，而定义却在 AudioTrack.cpp 中。

[-->AudioTrackShared.h::audio_track_cblk_t 声明]

```
struct audio_track_cblk_t
{

        Mutex       lock;
        Condition   cv;// 这是两个同步变量，初始化的时候会设置为支持跨进程共享。
    /*
     一块数据缓冲同时被生产者和消费者使用，最重要的就是维护它的读写位置了。
     下面定义的这些变量就和读写的位置有关，虽然它们的名字并不是那么直观。
     另外，这里提一个扩展问题，读者可以思考一下：
     volatile 支持跨进程吗？要回答这个问题需要理解 volatile、CPU Cache 机制和共享内存的本质。
   */
      volatile    uint32_t    user;     // 当前写位置（即生产者已经写到什么位置了）
      volatile    uint32_t    server;   // 当前读位置
   /*
      userBase 与 serverBase 要和 user 及 server 结合起来用。
      CB 巧妙地通过上面几个变量把一块线性缓冲当作环形缓冲来使用，以后将单独分析这个问题。
    */
                  uint32_t    userBase;  //
                  uint32_t    serverBase;
      void*       buffers; // 指向数据缓冲的首地址。
      uint32_t    frameCount;// 数据缓冲的总大小，以 Frame 为单位。

      uint32_t    loopStart; // 设置打点播放（即设置播放的起点和终点）。
      uint32_t    loopEnd;
       int         loopCount;// 循环播放的次数。

      volatile    union {
                      uint16_t    volume[2];
                      uint32_t    volumeLR;
                  }; // 和音量有关系，可以不管它。
       uint32_t    sampleRate;// 采样率
```

```
    uint32_t    frameSize;// 一单位 Frame 的数据大小。
    uint8_t     channels;// 声道数
    uint8_t     flowControlFlag;// 控制标志，见下文分析。
    uint8_t     out; // AudioTrack 为 1，AudioRecord 为 0。
    uint8_t     forceReady;
    uint16_t    bufferTimeoutMs;
    uint16_t    waitTimeMs;
     // 下面这几个函数很重要，后续会详细介绍它们。
    uint32_t    stepUser(uint32_t frameCount);// 更新写位置。
    bool        stepServer(uint32_t frameCount);// 更新读位置。
    void*       buffer(uint32_t offset) const;// 返回可写空间的起始位置。
    uint32_t    framesAvailable();// 还剩多少空间可写。
    uint32_t    framesAvailable_l();
    uint32_t    framesReady();// 是否有可读数据。
}
```

关于 CB 对象，这里要专门讲解一下其中 flowControlFlag 的意思：

- 对于音频输出来说，flowControlFlag 对应着 underrun 状态，underrun 状态是指生产者提供数据的速度跟不上消费者使用数据的速度。这里的消费者指的是音频输出设备。由于音频输出设备采用环形缓冲方式管理，当生产者没有及时提供新数据时，输出设备就会循环使用缓冲中的数据，这样就会听到一段重复的声音。这种现象一般被称作“machinegun”。对于这种情况，一般的处理方法是暂停输出，等数据准备好后再恢复输出。
- 对于音频输入来说，flowControlFlag 对应着 overrun 状态，它的意思和 underrun 一样，只是这里的生产者变成了音频输入设备，而消费者则变成了 Audio 系统的 AudioRecord。

说明 目前这个参数并不直接和音频输入输出设备的状态有关系。它在 AT 和 AF 中的作用必须结合具体情况才能分析。

图 7-4 表示 CB 对象和它所驻留的共享内存间的关系：

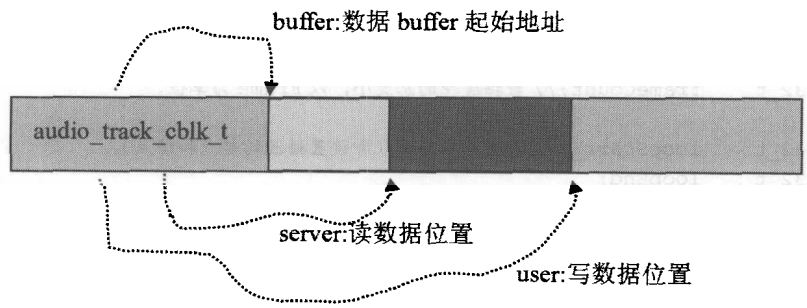


图 7-4 共享内存和 CB 的关系

注意　CB 实际是按照环形缓冲来处理数据的读写的，所以 user 和 server 的真实作用还需要结合 userBase 和 serverBase 来看。图 7-4 只是一个示意图。

另外，关于 CB，还有一个神秘的问题。先看下面这行代码：

```
mCblk = static_cast<audio_track_cblk_t*>(cblk->pointer());
```

这看起来很简单，但仔细琢磨会发现其中有一个很难解释的问题：

cblk->pointer 返回的是共享内存的首地址，怎么把 audio_track_cblk_t 对象塞到这块内存中呢？

这个问题将通过对 AudioFlinger 的分析得到答案。

说明　关于 audio_track_cblk_t 的使用方式，后文会有详细分析。

（3）数据的 Push or Pull

在 JNI 层的代码中可以发现，在构造 AudioTrack 时，传入了一个回调函数 audioCallback。由于它的存在，导致了 Native 的 AudioTrack 还将创建另一个线程 AudioTrackThread。它有什么用呢？

这个线程与外界数据的输入方式有关系，AudioTrack 支持两种数据输入方式：

- Push 方式：用户主动调用 write 写数据，这相当于数据被 push 到 AudioTrack。MediaPlayerService 一般使用这种这方式提供数据。
- Pull 方式：AudioTrackThread 将利用这个回调函数，以 EVENT_MORE_DATA 为参数主动从用户那 pull 数据。ToneGenerator 则使用这种方式为 AudioTrack 提供数据。

这两种方式都可以使用，不过回调函数除了 EVENT_MORE_DATA 外，还能表达其他许多意图，这是通过回调函数的第一个参数来表明的。一起来看：

[-->AudioTrack.h::event_type]

```
enum event_type {
        EVENT_MORE_DATA = 0, // 表示 AudioTrack 需要更多数据。
        EVENT_UNDERRUN = 1,// 这是 Audio 的一个术语，表示 Audio 硬件处于低负荷状态。
        //AT 可以设置打点播放，即设置播放的起点和终点，LOOP_END 表示已经到达播放终点。
        EVENT_LOOP_END = 2,
    /*
      数据使用警戒通知。该值可通过 setMarkerPosition () 设置。
      当数据的使用超过这个值时，AT 会且仅通知一次，有点像 Water Marker。
       这里所说的数据使用，是针对消费者 AF 消费的数据量而言的。
    */
        EVENT_MARKER = 3,
     /*
       数据使用进度通知。进度通知值由 setPositionUpdatePeriod() 设置，
       例如每使用 500 帧通知一次。
     */
        EVENT_NEW_POS = 4,
```

```
        EVENT_BUFFER_END = 5    // 数据全部被消耗
    };
```

请看 AudioTrackThread 的线程函数 threadLoop。代码如下所示：

[-->AudioTrack.cpp]

```
bool AudioTrack::AudioTrackThread::threadLoop()
{
  //mReceiver 就是创建该线程的 AudioTrack。
  return mReceiver.processAudioBuffer(this);
}
```

[-->AudioTrack.cpp]

```
bool AudioTrack::processAudioBuffer(const sp<AudioTrackThread>& thread)
{
    Buffer audioBuffer;
    uint32_t frames;
    size_t writtenSize;

    // 处理 underun 的情况。
    if (mActive && (mCblk->framesReady() == 0)) {
        if (mCblk->flowControlFlag == 0) {
            mCbf(EVENT_UNDERRUN, mUserData, 0);//under run 通知
            if (mCblk->server == mCblk->frameCount) {
            /*
               server 是读位置，frameCount 是 buffer 中的数据总和。
               当读位置等于数据总和时，表示数据都已经使用完了。
           */
                mCbf(EVENT_BUFFER_END, mUserData, 0);
            }
            mCblk->flowControlFlag = 1;
            if (mSharedBuffer != 0) return false;
        }
    }

    // 循环播放通知
    while (mLoopCount > mCblk->loopCount) {
        int loopCount = -1;
        mLoopCount--;
        if (mLoopCount >= 0) loopCount = mLoopCount;
        // 一次循环播放完毕，loopCount 表示还剩多少次。
        mCbf(EVENT_LOOP_END, mUserData, (void *)&loopCount);
    }

     if (!mMarkerReached && (mMarkerPosition > 0)) {
        if (mCblk->server >= mMarkerPosition) {
          // 如果数据的使用超过警戒值，则通知用户。
            mCbf(EVENT_MARKER, mUserData, (void *)&mMarkerPosition);
```

```
            //只通知一次，因为该值被设为 true。
             mMarkerReached = true;
        }
    }

    if (mUpdatePeriod > 0) {
        while (mCblk->server >= mNewPosition) {
       /*
          进度通知，但它不是以时间为基准的，而是以帧数为基准的。
          例如设置每 500 帧通知一次，假设消费者一次就读了 1500 帧，那么这个循环会连续通知 3 次。
       */
            mCbf(EVENT_NEW_POS, mUserData, (void *)&mNewPosition);
            mNewPosition += mUpdatePeriod;
        }
    }

    if (mSharedBuffer != 0) {
        frames = 0;
    } else {
        frames = mRemainingFrames;
    }

    do {

        audioBuffer.frameCount = frames;
       //得到一块可写的缓冲
        status_t err = obtainBuffer(&audioBuffer, 1);
        ......

       //从用户那 pull 数据
        mCbf(EVENT_MORE_DATA, mUserData, &audioBuffer);
        writtenSize = audioBuffer.size;

        ......
        if (writtenSize > reqSize) writtenSize = reqSize;
        //PCM8 数据转 PCM16
        .......

        audioBuffer.size = writtenSize;
        audioBuffer.frameCount = writtenSize/mCblk->frameSize;

        frames -= audioBuffer.frameCount;

        releaseBuffer(&audioBuffer);//写完毕，释放这块缓冲
    }
    while (frames);
     ......
    return true;
}
```

关于 obtainBuffer 和 releaseBuffer，后面再分析。这里有一个问题值得思考：

用例会调用 write 函数写数据，AudioTrackThread 的回调函数也让我们提供数据。难道我们同时在使用 Push 和 Pull 模式？

这太奇怪了！来查看这个回调函数的实现，了解一下究竟是怎么回事。该回调函数是通过 set 调用传入的，对应的函数是 audioCallback。代码如下所示：

[-->android_media_AudioTrack.cpp]

```
static void audioCallback(int event, void* user, void *info) {
    if (event == AudioTrack::EVENT_MORE_DATA) {
        //很好，没有提供数据，也就是说，虽然AudioTrackThread通知了EVENT_MORE_DATA，
        //但是我们并没有提供数据给它。
        AudioTrack::Buffer* pBuff = (AudioTrack::Buffer*)info;
        pBuff->size = 0;
    }
    ......
```

悬着的心终于放下来了，还是老老实实地看 Push 模式下的数据输入吧。

2. write 输入数据

write 函数涉及 Audio 系统中最重要的问题，即数据是如何传输的，在分析它的时候，不妨先思考一下它会怎么做。回顾一下我们已了解的信息：

- 有一块共享内存。
- 有一个控制结构，里面有一些支持跨进程的同步变量。

有了这些东西，write 的工作方式就非常简单了：

- 通过共享内存传递数据。
- 通过控制结构协调生产者和消费者的步调。

重点强调 *带着问题和思考来分析代码相当于“智取”，它比一上来就直接扎入源码的“强攻”要高明得多。希望我们能掌握这种思路和方法。*

好了，现在开始分析 write，看看它的实现是不是如我们所想的那样。

[-->AudioTrack.cpp]

```
ssize_t AudioTrack::write(const void* buffer, size_t userSize)
{
  if (mSharedBuffer != 0) return INVALID_OPERATION;

    if (ssize_t(userSize) < 0) {
       return BAD_VALUE;
    }
    ssize_t written = 0;
    const int8_t *src = (const int8_t *)buffer;
```

```
    Buffer audioBuffer; // Buffer是一个辅助性的结构。

    do {
        // 以帧为单位
        audioBuffer.frameCount = userSize/frameSize();
        //obtainBuffer从共享内存中得到一块空闲的数据块。
        status_t err = obtainBuffer(&audioBuffer, -1);
        ......

        size_t toWrite;

        if (mFormat == AudioSystem::PCM_8_BIT &&
                        !(mFlags & AudioSystem::OUTPUT_FLAG_DIRECT)) {
            //PCM8 数据转成 PCM16。
        } else {
            // 空闲数据缓冲的大小是 audioBuffer.size。
            // 地址在 audioBuffer.i8 中，数据传递通过 memcpy 完成。
            toWrite = audioBuffer.size;
            memcpy(audioBuffer.i8, src, toWrite);
            src += toWrite;
        }
        userSize -= toWrite;
        written += toWrite;
        //releaseBuffer 更新写位置，同时会触发消费者。
        releaseBuffer(&audioBuffer);
    } while (userSize);

    return written;
}
```

通过 write 函数，会发现数据的传递其实是很简单的 memcpy，但消费者和生产者的协调，则是通过 obtainBuffer 与 releaseBuffer 来完成的。现在来看这两个函数。

3. obtainBuffer 和 releaseBuffer

这两个函数展示了作为生产者的 AT 和 CB 对象的交互方法。先简单看看，然后把它们之间交互的流程记录下来，以后在 CB 对象的单独分析部分，我们再来做详细介绍。

[-->AudioTrack.cpp]

```
status_t AudioTrack::obtainBuffer(Buffer* audioBuffer, int32_t waitCount)
{
    int active;
    status_t result;
    audio_track_cblk_t* cblk = mCblk;
    ......
    // ①调用 framesAvailable，得到当前可写的空间大小。
    uint32_t framesAvail = cblk->framesAvailable();

    if (framesAvail == 0) {
```

```
            ......
            // 如果没有可写空间，则要等待一段时间。
            result = cblk->cv.waitRelative(cblk->lock,milliseconds(waitTimeMs));
              ......
        }

        cblk->waitTimeMs = 0;

        if (framesReq > framesAvail) {
            framesReq = framesAvail;
        }

        //user 为可写空间的起始地址。
        uint32_t u = cblk->user;
       uint32_t bufferEnd = cblk->userBase + cblk->frameCount;

        if (u + framesReq > bufferEnd) {
            framesReq = bufferEnd - u;
        }

         ......
        // ②调用 buffer，得到可写空间的首地址。
        audioBuffer->raw = (int8_t *)cblk->buffer(u);
        active = mActive;
        return active ? status_t(NO_ERROR) : status_t(STOPPED);
    }
```

obtainBuffer 的功能，就是从 CB 管理的数据缓冲中得到一块可写空间，而 releaseBuffer，则是在使用完这块空间后更新写指针的位置。

[-->AudioTrack.cpp]

```
void AudioTrack::releaseBuffer(Buffer* audioBuffer)
{
    audio_track_cblk_t* cblk = mCblk;
    cblk->stepUser(audioBuffer->frameCount);// ③调用 stepUser 更新写位置。
}
```

obtainBuffer 和 releaseBuffer 与 CB 交互，一共会有三个函数调用，如下所示：

- framesAvailable 判断是否有可写空间。
- buffer 得到写空间的起始地址。
- stepUser 更新写位置。

请记住这些流程，以后在分析 CB 时会发现它们有重要作用。

4. delete AudioTrack

到这里，AudioTrack 的使命就进入倒计时阶段了。来看它在生命的最后还会做一些什么工作。代码如下所示：

[-->AudioTrack.cpp]

```
AudioTrack::~AudioTrack()
{
    if (mStatus == NO_ERROR) {
        stop();// 调用 stop。
        if (mAudioTrackThread != 0) {
            // 通知 AudioTrackThread 退出。
            mAudioTrackThread->requestExitAndWait();
            mAudioTrackThread.clear();
        }
        mAudioTrack.clear();
     // 将残留在 IPCThreadState 发送缓冲区的信息发送出去。
        IPCThreadState::self()->flushCommands();
    }
}
```

如果不调用 stop，析构函数也会先调用 stop，这个做法很周到。代码如下所示：

[-->AudioTrack.cpp]

```
void AudioTrack::stop()
{
    sp<AudioTrackThread> t = mAudioTrackThread;
    if (t != 0) {
        t->mLock.lock();
    }

    if (android_atomic_and(~1, &mActive) == 1) {
        mCblk->cv.signal();
     /*
        mAudioTrack 是 IAudioTrack 类型，其 stop 的最终处理在 AudioFlinger 端。
      */
        mAudioTrack->stop();
       // 清空循环播放设置。
         setLoop(0, 0, 0);
         mMarkerReached = false;

        if (mSharedBuffer != 0) {
            flush();
        }
        if (t != 0) {
            t->requestExit();// 请求退出 AudioTrackThread。
        } else {
            setpriority(PRIO_PROCESS, 0, ANDROID_PRIORITY_NORMAL);
        }
    }

    if (t != 0) {
        t->mLock.unlock();
    }
}
```

stop 的工作比较简单，就是调用 IAudioTrack 的 stop，并且还要求退出回调线程。要重点关注 IAudioTrack 的 stop 函数，这个将作为 AT 和 AF 交互流程中的一个步骤来分析。

7.2.4 关于 AudioTrack 的总结

AudioTrack 就这样完了吗？它似乎也不是很复杂。其实，在进行 AT 分析时，对于一些难度比较大的地方暂时没做介绍。不过，在将 AudioFlinger 分析完之后，肯定不会怕它们的。

在完成对 AudioTrack 的分析之前，应把它和 AudioFlinger 交互的流程总结一下，如图 7-5 所示。这些流程是以后攻克 AudioFlinger 的重要武器。

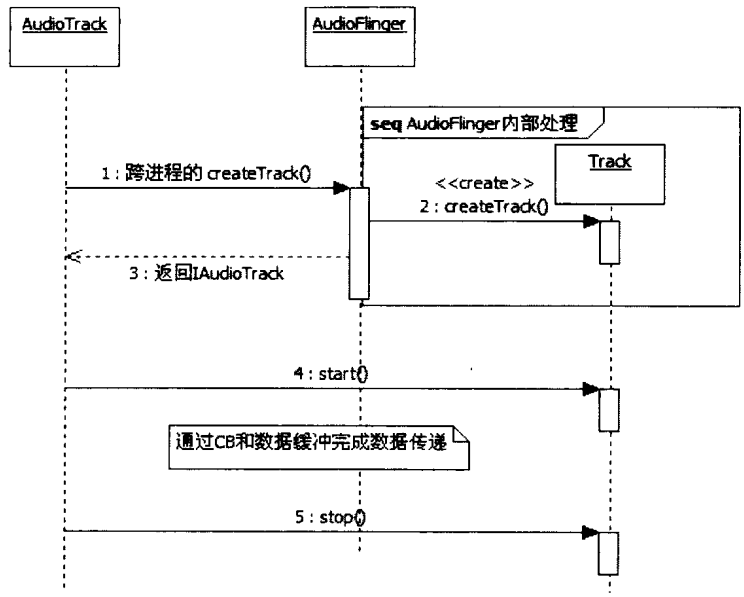


图 7-5 AT 和 AF 的交互流程图

7.3 AudioFlinger 的破解

AudioFlinger 是 Audio 系统的核心，来自 AudioTrack 的数据，最终都会在这里得到处理并被写入 Audio HAL 层。虽然破解 AudioFlinger 的难度比较大，但既然已经攻破了桥头堡 AudioTrack，并掌握了重要的突破口，那么对 AudioFlinger 的破解也就能手到擒来了。接下来，就一步步地破解它。

7.3.1 AudioFlinger 的诞生

AudioFlinger 驻留于 MediaServer 进程中。回顾一下它的代码，如下所示：

[-->Main_MediaServer.cpp]

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
     ....
    //很好，AF和APS都驻留在这个进程中。
     AudioFlinger::instantiate();
     AudioPolicyService::instantiate();
      ....
     ProcessState::self()->startThreadPool();
     IPCThreadState::self()->joinThreadPool();
}
```

1. AudioFlinger 的构造

先看一段代码，如下所示：

[-->AudioFlinger.cpp]

```
void AudioFlinger::instantiate() {
    defaultServiceManager()->addService( //把AF添加到ServiceManager中。
            String16("media.audio_flinger"), new AudioFlinger());
}
```

再来看它的构造函数，如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::AudioFlinger(): BnAudioFlinger(),
        mAudioHardware(0), //代表Audio硬件的HAL对象。
        mMasterVolume(1.0f), mMasterMute(false), mNextThreadId(0)
{
   mHardwareStatus = AUDIO_HW_IDLE;
   //创建代表Audio硬件的HAL对象。
    mAudioHardware = AudioHardwareInterface::create();

    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
     //设置系统初始化的一些值，有一部分通过Audio HAL设置到硬件中。
        setMode(AudioSystem::MODE_NORMAL);
        setMasterVolume(1.0f);
        setMasterMute(false);
    }
}
```

AudioHardwareInterface 是 Android 对代表 Audio 硬件的封装，属于 HAL 层。HAL 层的具体功能，由各个硬件厂商根据所选硬件的情况来实现，多以动态库的形式提供。这里简

单分析一下 Audio HAL 的接口，至于其具体的实现就不做过多的探讨了。

2. AudioHardwareInterface 介绍

AudioHardwareInterface 接口的定义在 AudioHardwareInterface.h 中。先来看看它。

[-->AudioHardwareInterface.h::AudioHardwareInterface 声明]

```
class AudioHardwareInterface
{
public:
    virtual ~AudioHardwareInterface() {}

     // 用于检查硬件是否初始化成功，返回的错误码定义在 include/utils/Errors.h 中。
    virtual status_t    initCheck() = 0;

    // 设置通话音量，范围从 0 到 1.0。
    virtual status_t    setVoiceVolume(float volume) = 0;

    /*
       设置除通话音量外的其他所有音频流类型的音量，范围从 0 到 1.0，如果硬件不支持的话，
       这个功能会由软件层的混音器完成。
     */
    virtual status_t    setMasterVolume(float volume) = 0;

    /*
       设置模式，NORMAL 的状态为普通模式，RINGTONE 表示来电模式（这时听到的声音是来电铃声），
       IN_CALL 表示通话模式（这时听到的声音是手机通话过程中的语音）。
     */
    virtual status_t    setMode(int mode) = 0;

    // 和麦克相关
    virtual status_t    setMicMute(bool state) = 0;
    virtual status_t    getMicMute(bool* state) = 0;

    // 设置 / 获取配置参数，采用 key/value 的组织方式。
    virtual status_t    setParameters(const String8& keyValuePairs) = 0;
    virtual String8     getParameters(const String8& keys) = 0;

    // 根据传入的参数得到输入缓冲的大小，返回 0 表示其中某个参数的值 Audio HAL 不支持。
     virtual size_t     getInputBufferSize(uint32_t sampleRate, int format,
                                           int channelCount) = 0;

        /* 下面这几个函数非常重要 */
     /*
         openOutputStream：创建音频输出流对象（相当于打开音频输出设备）。
         AF 可以往其中 write 数据，指针型参数将返回该音频输出流支持的类型、声道数、采样率等。
    */
    virtual AudioStreamOut* openOutputStream(
                                uint32_t devices,
                                int *format=0,
```

```
                                uint32_t *channels=0,
                                uint32_t *sampleRate=0,
                                status_t *status=0) = 0;
    // 关闭音频输出流。
     virtual    void        closeOutputStream(AudioStreamOut* out) = 0;

     /* 创建音频输入流对象（相当于打开音频输入设备），AF 可以 read 数据 */
     virtual AudioStreamIn* openInputStream(
                                uint32_t devices,
                                int *format,
                                uint32_t *channels,
                                uint32_t *sampleRate,
                                status_t *status,
                                AudioSystem::audio_in_acoustics acoustics) = 0;

     virtual    void        closeInputStream(AudioStreamIn* in) = 0;
     // 关闭音频输入流。
     virtual status_t dumpState(int fd, const Vector<String16>& args) = 0;
     // 静态 create 函数，使用设计模式中的工厂模式，具体返回的对象由厂商根据硬件的情况决定。
     static AudioHardwareInterface* create();
     ......
};
```

根据上面的代码，可以得出以下结论：

- AudioHardwareInterface 管理音频输出设备对象（AudioStreamOut）和音频输入设备对象（AudioStreamIn）的创建。
- 通过 AudioHardwareInterface 可设置音频系统的一些参数。

图 7-6 表示 AudioHardwareInterface 和音频输入输出对象之间的关系，以及它们的派生关系：

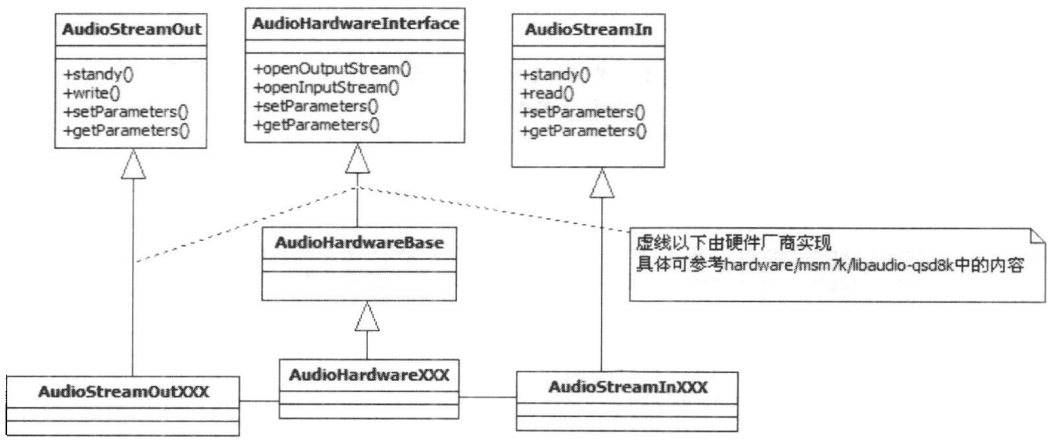


图 7-6　AudioHardwareInterface 关系图

从图 7-6 中还可看出：

音频输出 / 输入对象均支持设置参数（由 setParameters 完成）。

说明 AudioHardwareInterface 最重要的功能是创建 AudioStreamOut 和 AudioStreamIn，它们分别代表音频输出设备和音频输入设备。从这个角度说，是 AudioHardwareInterface 管理着系统中所有的音频设备。Android 引入的 HAL 层，大大简化了应用层的工作，否则不管是使用 libasound（AlSA 提供的用户空间库）还是 ioctl 来控制音频设备，都会非常麻烦。

7.3.2 通过流程分析 AudioFlinger

图 7-5 中说明的 AT 和 AF 交互流程，对于分析 AF 来说非常重要。先来回顾一下图 7-5 的流程：

- AT 调用 createTrack，得到一个 IAudioTrack 对象。
- AT 调用 IAudioTrack 对象的 start，表示准备写数据了。
- AT 通过 write 写数据，这个过程和 audio_track_cblk_t 有着密切关系。
- 最后 AT 调用 IAudioTrack 的 stop 或 delete IAudioTrack 结束工作。

至此，上面的每一步都很清楚了。根据 Binder 知识可知，AT 调用的这些函数最终都会在 AF 端得到实现，所以可直接从 AF 端开始。

1. createTrack 分析

按照前面的流程步骤来看，第一个被调用的函数会是 createTrack，请注意在用例中传的参数。代码如下所示：

[-->AudioFlinger.cpp]

```
sp<IAudioTrack> AudioFlinger::createTrack(
        pid_t pid,//AT 的 pid 号
        int streamType,// 流类型，用例中是 MUSIC 类型
        uint32_t sampleRate,//8000 采样率
        int format,//PCM_16 类型
        int channelCount,//2，双声道
        int frameCount,// 需要创建的缓冲大小，以帧为单位。
        uint32_t flags,
        const sp<IMemory>& sharedBuffer,//AT 传入的共享 buffer，这里为空。
        int output,// 这个值前面提到过，是 AF 中的工作线程索引号。
        status_t *status)
{
    sp<PlaybackThread::Track> track;
    sp<TrackHandle> trackHandle;
    sp<Client> client;
    wp<Client> wclient;
    status_t lStatus;
```

```
    {
        Mutex::Autolock _l(mLock);
        //output 代表索引号，这里根据索引号找到一个工作线程，它是一个 PlaybackThread。
        PlaybackThread *thread = checkPlaybackThread_l(output);
       // 看看这个进程是否已经是 AF 的 Client，AF 根据进程 pid 来标识不同的 Client。
         wclient = mClients.valueFor(pid);
        if (wclient != NULL) {
       } else {
            // 如果还没有这个 Client 信息，则创建一个，并加入到 mClients 中去。
            client = new Client(this, pid);
            mClients.add(pid, client);
        }

      // 在找到的工作线程对象中创建一个 Track，注意它的类型是 Track。
        track = thread->createTrack_l(client, streamType, sampleRate, format,
                channelCount, frameCount, sharedBuffer, &lStatus);
    }
    /*
     TrackHandle 是 Track 对象的 Proxy，它支持 Binder 通信，而 Track 不支持 Binder。
     TrackHandle 所接收的请求最终会由 Track 处理，这是典型的 Proxy 模式。
    */
    trackHandle = new TrackHandle(track);
    return trackHandle;
}
```

这个函数相当复杂，主要原因之一，是其中出现了几个我们没接触过的类。我刚接触这个函数的时候，大脑也曾因为看到这些眼生的东西而“死机”！不过暂时先不用去理会它们，等了解了这个函数后，再回过头来收拾它们。先进入 checkPlaybackThread_l 看看。

（1）选择工作线程

checkPlaybackThread_l 的代码如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::PlaybackThread *
          AudioFlinger::checkPlaybackThread_l(int output) const
{
    PlaybackThread *thread = NULL;
    // 根据 output 的值找到对应的 thread。
    if (mPlaybackThreads.indexOfKey(output) >= 0) {
        thread = (PlaybackThread *)mPlaybackThreads.valueFor(output).get();
    }
    return thread;
}
```

上面函数中传入的 output，就是之前在分析 AT 时提到的工作线程索引号。看到这里，是否感觉有点困惑？困惑的原因可能有二：

- 目前的流程中尚没有见到创建线程的地方，但在这里确实能找到一个线程。

❑ Output 的含义到底是什么？为什么它会作为 index 来找线程呢？

关于这两个问题，待会儿再做解释。现在只需知道 AudioFlinger 会创建几个工作线程，AT 会找到对应的工作线程即可。

（2）createTrack_l 分析

找到工作线程后，会执行 createTrack_l 函数，请看这个函数的作用：

[-->AudioFlinger.cpp]

```
// Android 的很多代码都采用了内部类的方式进行封装，看习惯就好了。
sp<AudioFlinger::PlaybackThread::Track>
     AudioFlinger::PlaybackThread::createTrack_l(
        const sp<AudioFlinger::Client>& client,int streamType,
        uint32_t sampleRate,int format,int channelCount,int frameCount,
        const sp<IMemory>& sharedBuffer,// 注意这个参数，从 AT 中传入，为 0。
        status_t *status)
{
    sp<Track> track;
    status_t lStatus;
    {
        Mutex::Autolock _l(mLock);
      // 创建 Track 对象。
     track = new Track(this, client, streamType, sampleRate, format,
                       channelCount, frameCount, sharedBuffer);
      // 将新创建的 Track 加入到内部数组 mTracks 中。
       mTracks.add(track);
   }
   lStatus = NO_ERROR;
   return track;
}
```

上面的函数调用传入的 sharedBuffer 为空，那共享内存又是在哪里创建的呢？可以注意到 Track 构造函数中关于 sharedBuffer 这个参数的类型是一个引用，莫非是构造函数创建的？

（3）Track 创建共享内存和 TrackHandle

在 createTrack_l 中，会 new 出来一个 Track，请看它的代码：

[-->AudioFlinger.cpp]

```
AudioFlinger::PlaybackThread::Track::Track( const wp<ThreadBase>& thread,
            const sp<Client>& client,int streamType,uint32_t sampleRate,
            int format,int channelCount,int frameCount,
             const sp<IMemory>& sharedBuffer)
:   TrackBase(thread, client, sampleRate, format, channelCount,
    frameCount, 0, sharedBuffer),//sharedBuffer 仍然为空。
    mMute(false), mSharedBuffer(sharedBuffer), mName(-1)
{
   // mCblk !=NULL? 什么时候创建的呢？只能看基类 TrackBase 的构造函数了。
   if (mCblk != NULL) {
```

```
        mVolume[0] = 1.0f;
        mVolume[1] = 1.0f;
        mStreamType = streamType;
        mCblk->frameSize = AudioSystem::isLinearPCM(format) ?
                                channelCount * sizeof(int16_t) : sizeof(int8_t);
    }
}
```

对于这种重重继承，我们只能步步深入分析，一定要找到共享内存创建的地方，继续看代码：

[-->AudioFlinger.cpp]

```
AudioFlinger::ThreadBase::TrackBase::TrackBase(
            const wp<ThreadBase>& thread,const sp<Client>& client,
            uint32_t sampleRate,int format,int channelCount,int frameCount,
            uint32_t flags,const sp<IMemory>& sharedBuffer)
    :   RefBase(), mThread(thread),mClient(client), mCblk(0),
        mFrameCount(0),mState(IDLE),mClientTid(-1),mFormat(format),
        mFlags(flags & ~SYSTEM_FLAGS_MASK)
{
    size_t size = sizeof(audio_track_cblk_t);// 得到 CB 对象的大小。
   // 计算数据缓冲的大小。
   size_t bufferSize = frameCount*channelCount*sizeof(int16_t);
   if (sharedBuffer == 0) {
    //还记得图 7-4 吗？共享内存的最前面一部分是 audio_track_cblk_t，后面才是数据空间。
      size += bufferSize;
   }
   // 根据 size 创建一块共享内存。
    mCblkMemory = client->heap()->allocate(size);
   /*
      pointer() 返回共享内存的首地址，并强制转换 void* 类型为 audio_track_cblk_t* 类型。
      其实把它强制转换成任何类型都可以，但是这块内存中会有 CB 对象吗？
   */
     mCblk = static_cast<audio_track_cblk_t *>(mCblkMemory->pointer());
    //①下面这句代码看起来很独特。什么意思？？？
     new(mCblk) audio_track_cblk_t();

     mCblk->frameCount = frameCount;
     mCblk->sampleRate = sampleRate;
     mCblk->channels = (uint8_t)channelCount;
     if (sharedBuffer == 0) {
       // 清空数据区
       mBuffer = (char*)mCblk + sizeof(audio_track_cblk_t);
       memset(mBuffer, 0, frameCount*channelCount*sizeof(int16_t));
      // flowControlFlag 初始值为 1。
       mCblk->flowControlFlag = 1;
     }
       ......
}
```

这里需要重点讲解下面这句话的意思。

```
new(mCblk) audio_track_cblk_t();
```

注意它的用法，new 后面的括号里是内存，紧接其后的是一个类的构造函数。

重点说明 这个语句就是 C++ 语言中的 placement new。其含义是在括号里指定的内存中创建一个对象。我们知道，普通的 new 只能在堆上创建对象，堆的地址由系统分配。这里采用 placement new 将使 audio_track_cblk_t 创建在共享内存上，它就自然而然地能被多个进程看见并使用了。关于 placement new 较详细的知识，还请读者自己搜索一下。

通过上面的分析可以知道：

- Track 创建了共享内存。
- CB 对象通过 placement new 方法创建于这块共享内存中。

AF 的 createTrack 函数返回的是一个 IAudioTrack 类型的对象，可现在碰到的 Track 对象是 IAudioTrack 类型的吗？来看代码：

[-->AudioFlinger.cpp]

```
sp<IAudioTrack> AudioFlinger::createTrack (......)
{
    sp<TrackHandle> trackHandle;
     ......
    track = thread->createTrack_l(client, streamType, sampleRate,
                    format,channelCount, frameCount, sharedBuffer, &lStatus);

   trackHandle = new TrackHandle(track);
   return trackHandle; //① 这个trackHandle对象竟然没有在AF中保存!
 }
```

原来，createTrack 返回的是 TrackHandle 对象，它以 Track 为参数构造。这二者之间又是什么关系呢？

Android 在这里使用了 Proxy 模式，即 TrackHandle 是 Track 的代理，TrackHandle 代理的内容是什么呢？分析 TrackHandle 的定义可以知道：

- Track 没有基于 Binder 通信，它不能接收来自远端进程的请求。
- TrackHandle 能基于 Binder 通信，它可以接收来自远端进程的请求，并且能调用 Track 对应的函数。这就是 Proxy 模式的意思。

讨论 Android 为什么不直接让 Track 从 IBinder 派生，直接支持 Binder 通信呢？关于这个问题，在看到后面的 Track 家族图谱后，我们或许就明白了。

另外，注意代码中的注释①：

trackHandle 被 new 出来后直接返回，而 AF 中并没有保存它，这岂不是成了令人闻之

色变的野指针？

说明　关于这个问题的答案，请读者自己思考并回答。提示，可从 Binder 和 RefBase 入手。

分析完 createTrack 后，估计有些人会晕头转向的。确实，这个 createTrack 比较复杂。仅仅是对象类型就层出不穷。到底它有多少种对象，它们之间又有怎样的关系呢？下面就来解决这几个问题。

2. 到底有多少种对象？

现在不妨把 AudioFlinger 中出现的对象总结一下，以了解它们的作用和相互之间的关系。

（1）AudioFlinger 对象

作为 Audio 系统的核心引擎，首先要介绍 AudioFlinger。它的继承关系很简单：

```
class AudioFlinger : public BnAudioFlinger, public IBinder::DeathRecipient
```

AudioFlinger 的主要工作由其定义的许多内部类来完成，我们用图 7-7 来表示。图中大括号所指向的类为外部类，大括号所包含的为该外部类所定义的内部类。例如，DuplicatingThread、RecordThread 和 DirectOutputThread 都包括在一个大括号中，这个大括号指向 AudioFlinger，所以它们三个都是 AudioFlinger 的内部类，而 AudioFlinger 则是它们三个的外部类：

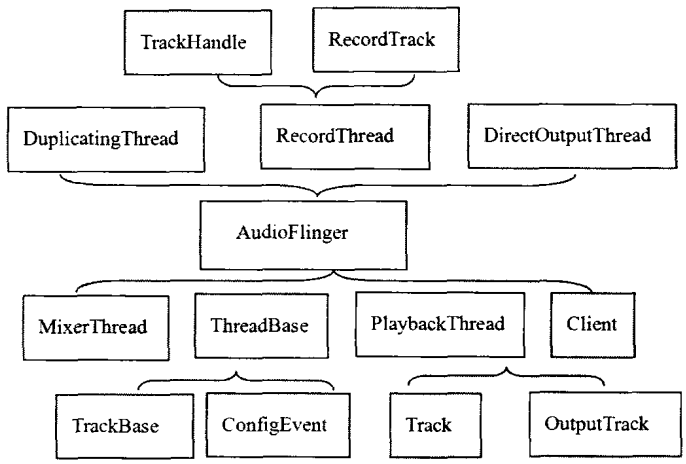


图 7-7　AF 中的所有类

看，AF 够复杂吧？要不是使用了 Visual Studio 的代码段折叠功能，我画这个图，也会颇费周折的。

（2）Client 对象

Client 是 AudioFlinger 对客户端的封装，凡是使用了 AudioTrack 和 AudioRecord 的进

程，都被会当作是 AF 的 Client，并且 Client 用它的进程 pid 作为标识。代码如下所示：

```
class Client : public RefBase {
    public:
        Client(const sp<AudioFlinger>& audioFlinger, pid_t pid);
        virtual             ~Client();
        const sp<MemoryDealer>&     heap() const;
        pid_t               pid() const { return mPid; }
        sp<AudioFlinger>    audioFlinger() { return mAudioFlinger; }

    private:
                            Client(const Client&);
                            Client& operator = (const Client&);
        sp<AudioFlinger>    mAudioFlinger;
        sp<MemoryDealer>    mMemoryDealer;// 内存分配器
        pid_t                  mPid;
    };
```

Client 对象比较简单，因此就不做过多的分析了。

注意 一个 Client 进程可以创建多个 AudioTrack，这些 AudioTrack 都属于同一个 Client。

（3）工作线程介绍

AudioFlinger 中有几种不同类型的工作线程，它们之间的关系如图 7-8 所示：

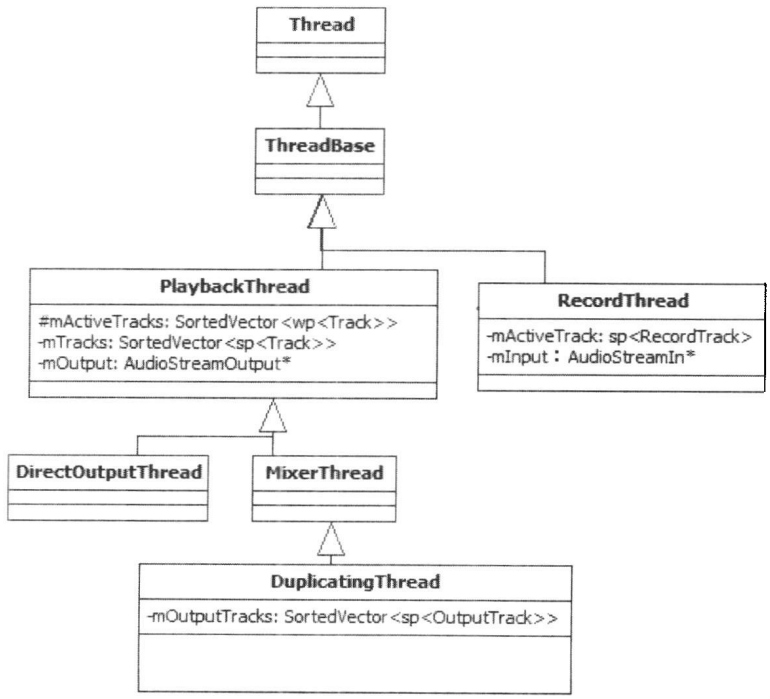


图 7-8 AF 中的工作线程家谱

下面来解释图 7-8 中各种类型工作线程的作用：

- PlaybackThread：回放线程，用于音频输出。它有一个成员变量 mOutput，为 AudioStreamOutput* 类型，这表明 PlaybackThread 直接和 Audio 音频输出设备建立了联系。
- RecordThread：录音线程，用于音频输入，它的关系比较单纯。它有一个成员变量 mInput，为 AudioStreamInput* 类型，这表明 RecordThread 直接和 Audio 音频输入设备建立了联系。

从 PlaybackThread 的派生关系上可看出，手机上的音频回放应该比较复杂，否则也不会派生出三个子类了。其中：

- MixerThread：混音线程，它将来自多个源的音频数据混音后再输出。
- DirectOutputThread：直接输出线程，它会选择一路音频流后将数据直接输出，由于没有混音的操作，这样可以减少很多延时。
- DuplicatingThread：多路输出线程，它从 MixerThread 派生，意味着它也能够混音。它最终会把混音后的数据写到多个输出中，也就是一份数据会有多个接收者。这就是 Duplicate 的含义。目前在蓝牙 A2DP 设备输出中使用。

另外从图 7-8 中还可以看出：

- PlaybackThread 维护两个 Track 数组，一个是 mActiveTracks，表示当前活跃的 Track；另一个是 mTracks，表示这个线程创建的所有 Track。
- DuplicatingThread 还维护了一个 mOutputTracks，表示多路输出的目的端。后面分析 DuplicatingThread 时再对此进行讲解。

说明　大部分常见音频输出使用的是 MixerThread，后文会对此进行详细分析。另外，在拓展内容中，也将深入分析 DuplicatingThread 的实现。

（4）PlaybackThread 和 AudioStreamOutput

从图 7-8 中，可以发现 PlaybackThread 有一个 AudioStreamOutput 类型的对象，这个对象提供了音频数据的输出功能。可以用图 7-9 来表示音频数据的流动轨迹。该图以 PlaybackThread 最常用的子类 MixerThread 作为代表。

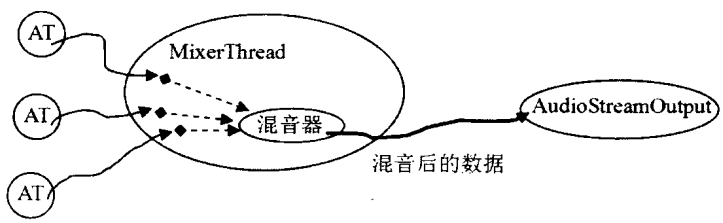


图 7-9　音频数据的流动轨迹

根据图 7-9，就能明白 MixerThread 的大致工作流程了：

- 接收来自 AT 的数据。
- 对这些数据进行混音。
- 把混音的结果写到 AudioStreamOut 中，这样就完成了音频数据的输出。

（5）Track 对象

前面所说的工作线程，其工作就是围绕 Track 展开的，图 7-10 展示了 Track 的家族：

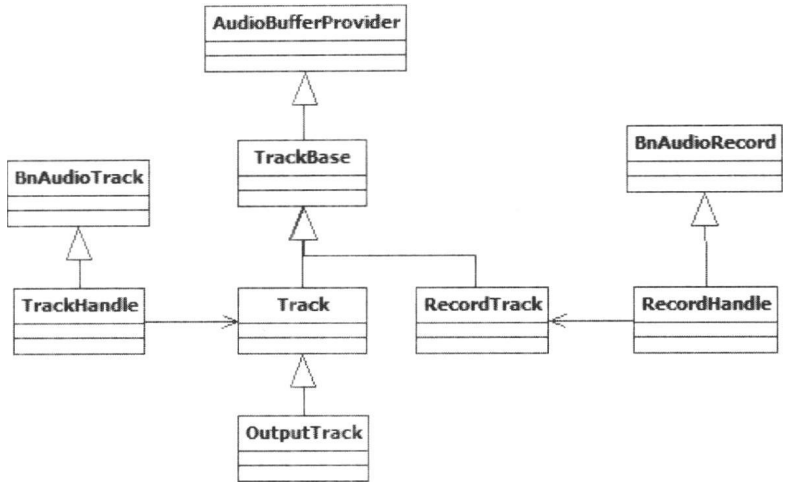


图 7-10 Track 家族

注意 这里把 RecordTrack 也统称为 Track。

从图 7-10 中可看出，TrackHandle 和 RecordHandle 是基于 Binder 通信的，它作为 Proxy，用于接收请求并派发给对应的 Track 和 RecordTrack。

说明 从图 7-10 也能看出，之所以不让 Track 继承 Binder 框架，是因为 Track 本身的继承关系和它所承担的工作已经很复杂了，如再让它掺合 Binder，只会乱上添乱。

Track 类作为工作线程的内部类来实现，其中：

- TrackBase 定义于 ThreadBase 中。
- Track 定义于 PlaybackThread 中，RecordTrack 定义于 RecordThread 中。
- OutputTrack 定义于 DuplicatingThread 中。

根据前面的介绍可知，音频输出数据最后由 Playback 线程来处理，用例所对应的 Playback 线程实际上是一个 MixerThread，那么它是如何工作的呢？一起来分析。

3.MixerThread 分析

MixerThread 是 Audio 系统中负担最重的一个工作线程。先来了解一下它的来历。

（1）MixerThread 的来历

前面，在 checkplaybackThread_l 中，有一个地方一直没来得及解释。回顾一下它的代码：

[-->AudioFlinger.cpp]

```
AudioFlinger::PlaybackThread *
   AudioFlinger::checkPlaybackThread_l(int output) const
{
  PlaybackThread *thread = NULL;
  // 根据 output 的值找到对应的 thread
    if (mPlaybackThreads.indexOfKey(output) >= 0) {
        thread = (PlaybackThread *)mPlaybackThreads.valueFor(output).get();
  }
    return thread;
}
```

上面这个函数的意思很明确：就是根据 output 值找到对应的回放线程。

但在前面的流程分析中，并没有见到创建线程的地方，那这个线程又是如何得来的？它又是何时、怎样创建的呢？

答案在 AudioPolicyService 中。提前看看 AudioPolicyService，分析一下它为什么和这个线程有关系。

AudioPolicyService 和 AudioFlinger 一样，都驻留在 MediaServer 中，直接看它的构造函数：

[-->AudioPolicyService.cpp]

```
AudioPolicyService::AudioPolicyService()
    : BnAudioPolicyService() , mpPolicyManager(NULL)
{
    char value[PROPERTY_VALUE_MAX];

    // Tone 音播放线程。
    mTonePlaybackThread = new AudioCommandThread(String8(""));
    // 命令处理线程。
    mAudioCommandThread = new AudioCommandThread(String8("ApmCommandThread"));

#if (defined GENERIC_AUDIO) || (defined AUDIO_POLICY_TEST)
    // 这里属于 Generic 的情况，所以构造 AudioPolicyManagerBase，注意构造函数的参数。
    mpPolicyManager = new AudioPolicyManagerBase(this);
#else
......
    // 创建和硬件厂商相关的 AudioPolicyManager。
#endif
    ......
}
```

看 AudioPolicyManagerBase 的构造函数，注意传给它的参数是 this，即把 AudioPolicyService 对象传进去了。代码如下所示：

[-->AudioPolicyManagerBase.cpp]

```
AudioPolicyManagerBase::AudioPolicyManagerBase(
     AudioPolicyClientInterface *clientInterface)
   : mPhoneState(AudioSystem::MODE_NORMAL), mRingerMode(0),
     mMusicStopTime(0), mLimitRingtoneVolume(false)
{
    mpClientInterface = clientInterface;

    // 先把不相关的内容去掉。
    ......
  /*
     返回来调用 mpClientInterface 的 openOutput，实际就是 AudioPolicyService。
     注意 openOutput 函数是在 AP 的创建过程中调用的。
  */
    mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,
                                      &outputDesc->mSamplingRate,
                                      &outputDesc->mFormat,
                                      &outputDesc->mChannels,
                                      &outputDesc->mLatency,
                                      outputDesc->mFlags);
    ......
}
```

真是山不转水转！咱们还得回到 AudioPolicyService 中去看看：

[-->AudioPolicyService.cpp]

```
audio_io_handle_t AudioPolicyService::openOutput(uint32_t *pDevices,
                                 uint32_t *pSamplingRate,
                                 uint32_t *pFormat,
                                 uint32_t *pChannels,
                                 uint32_t *pLatencyMs,
                                 AudioSystem::output_flags flags)
{
  sp<IAudioFlinger> af = AudioSystem::get_audio_flinger();
  // 下面会调用 AudioFlinger 的 openOutput，这个时候 AF 已经启动了。
   return af->openOutput(pDevices, pSamplingRate, (uint32_t *)pFormat,
                  pChannels, pLatencyMs, flags);
}
```

真是曲折啊，又得到 AF 去看看：

[-->AudioFlinger.cpp]

```
int AudioFlinger::openOutput(
uint32_t *pDevices,uint32_t *pSamplingRate,uint32_t *pFormat,
       uint32_t *pChannels,uint32_t *pLatencyMs,uint32_t flags)
{
   ......
```

```
    Mutex::Autolock _l(mLock);
    // 创建 Audio HAL 的音频输出对象，和音频输出扯上了关系。
     AudioStreamOut *output = mAudioHardware->openOutputStream(*pDevices,
                                                               (int *)&format,
                                                               &channels,
                                                               &samplingRate,
                                                               &status);
     mHardwareStatus = AUDIO_HW_IDLE;
     if (output != 0) {
         if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) ||
             (format != AudioSystem::PCM_16_BIT) ||
             (channels != AudioSystem::CHANNEL_OUT_STEREO)) {
             // 如果标志为 OUTPUT_FLAG_DIRECT，则创建 DirectOutputThread。
             thread = new DirectOutputThread(this, output, ++mNextThreadId);
            } else {
          // 一般创建的都是 MixerThread，注意代表 AudioStreamOut 对象的 output 也传进去了。
           thread = new MixerThread(this, output, ++mNextThreadId);
         }
         // 把新创建的线程加入线程组 mPlaybackThreads 中保存，mNextThreadId 是它的索引号。
         mPlaybackThreads.add(mNextThreadId, thread);
         ......
       return mNextThreadId;// 返回该线程的索引号。
     }
     return 0;
}
```

明白了吗？是否感觉有点绕？可用一个简单的示意图来观察三者的交互流程，如图 7-11 所示：

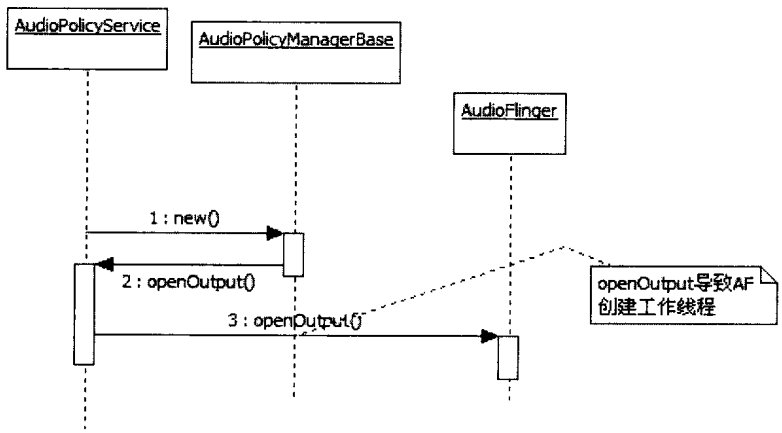


图 7-11　MixerThread 的曲折来历示意图

图 7-11 表明：

❑ AF 中工作线程的创建，受到了 AudioPolicyService 的控制。从 AudioPolicyService 的

角度出发，这也是应该的，因为 APS 控制着整个音频系统，而 AF 只是管理音频的输入和输出。

❑ 另外，注意这个线程是在 AP 的创建过程中产生的。也就是说，AP 一旦创建完 Audio 系统，就已经准备好要工作了。

关于 AF 和 AP 的恩恩怨怨，在后面 APS 的分析过程中再去探讨。目前，读者只需了解系统中第一个 MixerThread 的来历即可。下面来分析这个来之不易的 MixerThread。

（2）MixerThread 的构造和线程启动

[-->AudioFlinger.cpp]

```
AudioFlinger::MixerThread::MixerThread(
    const sp<AudioFlinger>& audioFlinger,
    AudioStreamOut* output, // AudioStreamOut 为音频输出设备的 HAL 抽象。
    int id)
      :    PlaybackThread(audioFlinger, output, id),mAudioMixer(0)
{
    mType = PlaybackThread::MIXER;
   // 混音器对象，这个对象比较复杂，它完成多路音频数据的混合工作。
    mAudioMixer = new AudioMixer(mFrameCount, mSampleRate);
}
```

再来看 MixerThread 的基类 PlaybackThread 的构造函数：

[-->AudioFlinger.cpp]

```
AudioFlinger::PlaybackThread::PlaybackThread(const sp<AudioFlinger>&
        audioFlinger, AudioStreamOut* output, int id)
    :    ThreadBase(audioFlinger, id),
     mMixBuffer(0), mSuspended(0), mBytesWritten(0),
     mOutput(output), mLastWriteTime(0), mNumWrites(0),
     mNumDelayedWrites(0), mInWrite(false)
{
     // 获取音频输出 HAL 对象的一些信息，包括硬件中音频缓冲区的大小（以帧为单位）。
     readOutputParameters();
     mMasterVolume = mAudioFlinger->masterVolume();
     mMasterMute = mAudioFlinger->masterMute();
   // 设置不同类型音频流的音量及静音情况。
     for (int stream = 0; stream < AudioSystem::NUM_STREAM_TYPES; stream++)
     {
         mStreamTypes[stream].volume =
                       mAudioFlinger->streamVolumeInternal(stream);
         mStreamTypes[stream].mute = mAudioFlinger->streamMute(stream);
     }
   // 发送一个通知消息给监听者，这部分内容较简单，读者可自行研究。
   sendConfigEvent(AudioSystem::OUTPUT_OPENED);
 }
```

此时，线程对象已经创建完毕。根据对 Thread 的分析可知，应该调用它的 run 函数才能

真正创建新线程。在首次创建 sp 时调用了 run，这里利用了 RefBase 的 onFirstRef 函数。根据 MixerThread 的派生关系来看，该函数最终由父类 PlaybackThread 的 onFirstRef 实现：

[-->AudioFlinger.cpp]

```
void AudioFlinger::PlaybackThread::onFirstRef()
{
    const size_t SIZE = 256;
    char buffer[SIZE];

    snprintf(buffer, SIZE, "Playback Thread %p", this);
   // 下面的 run 就真正创建了线程并开始执行 threadLoop。
    run(buffer, ANDROID_PRIORITY_URGENT_AUDIO);
}
```

好，线程已经 run 起来了。继续按流程分析，下一个轮到的调用函数是 start。

4. start 分析

AT 调用的是 IAudioTrack 的 start 函数，由于 TrackHandle 的代理作用，这个函数的处理实际会由 Track 对象来完成。

[-->AudioFlinger.cpp]

```
status_t AudioFlinger::PlaybackThread::Track::start()
{
    status_t status = NO_ERROR;
    sp<ThreadBase> thread = mThread.promote();
  // 该 Thread 是用例中的 MixerThread。
    if (thread != 0) {
        Mutex::Autolock _l(thread->mLock);
        int state = mState;
         if (mState == PAUSED) {
            mState = TrackBase::RESUMING;
           } else {
            mState = TrackBase::ACTIVE;// 设置 Track 的状态
        }
        PlaybackThread *playbackThread = (PlaybackThread *)thread.get();
        //addTrack_l 把这个 track 加入到 mActiveTracks 数组中。
        playbackThread->addTrack_l(this);
    return status;
}
```

看看这个 addTrack_l 函数，代码如下所示：

[-->AudioFlinger.cpp]

```
status_t AudioFlinger::PlaybackThread::addTrack_l(const sp<Track>& track)
{
    status_t status = ALREADY_EXISTS;
```

```
        // ① mRetryCount：设置重试次数，kMaxTrackStartupRetries 值为 50。
        track->mRetryCount = kMaxTrackStartupRetries;
      if (mActiveTracks.indexOf(track) < 0) {
          // ② mFillingUpStatus：缓冲状态。
          track->mFillingUpStatus = Track::FS_FILLING;
          // 原来是把调用 start 的这个 track 加入到活跃的 Track 数组中了。
          mActiveTracks.add(track);
          status = NO_ERROR;
      }
     // 广播一个事件，一定会触发 MixerThread 线程，通知它有活跃数组加入，需要开工干活。
      mWaitWorkCV.broadcast();
      return status;
  }
```

start 函数把这个 Track 加入到活跃数组后，将触发一个同步事件，这个事件会让工作线程动起来。虽然这个函数很简单，但有两个关键点必须指出，这两个关键点其实指出了两个问题的处理办法：

- mRetryCount 表示重试次数，它针对的是这样一个问题：如果一个 Track 调用了 start 却没有 write 数据，该怎么办？如果 MixerThread 尝试了 mRetryCount 次后还没有可读数据，工作线程就会把该 Track 从激活队列中去掉了。
- mFillingUpStatus 能解决这样的问题：假设分配了 1MB 的数据缓冲，那么至少需要写多少数据的工作线程才会让 Track 觉得 AT 是真的需要它工作呢？难道 AT 写一个字节就需要工作线程兴师动众吗？其实，这个状态最初为 Track::FS_FILLING，表示正在填充数据缓冲。在这种状态下，除非 AT 设置了强制读数据标志（CB 对象中的 forceReady 变量），否则工作线程是不会读取该 Track 的数据的。该状态还有其他的值，读者可以自行研究。

说明 我们在介绍大流程的同时也把一些细节问题指出来，希望这些细节问题能激发读者深入研究的欲望。

Track 加入工作线程的活跃数组后，又触发了一个同步事件，MixerThread 是否真的动起来了呢？一起来看。

（1）MixerThread 动起来

Thread 类的线程工作都是在 threadLoop 中完成的，那么 MixerThread 的线程又会做什么呢？

[-->AudioFlinger.cpp]

```
bool AudioFlinger::MixerThread::threadLoop()
{
    int16_t* curBuf = mMixBuffer;
    Vector< sp<Track> > tracksToRemove;
```

```
uint32_t mixerStatus = MIXER_IDLE;
nsecs_t standbyTime = systemTime();
......
uint32_t sleepTime = idleSleepTime;

while (!exitPending())
{
 //① 处理一些请求和通知消息，如之前在构造函数中发出的 OUTPUT_OPEN 消息等。
  processConfigEvents();

    mixerStatus = MIXER_IDLE;
    { // scope for mLock

        Mutex::Autolock _l(mLock);
       //检查配置参数，如有需要则重新设置内部参数值。
        if (checkForNewParameters_l()) {
            mixBufferSize = mFrameCount * mFrameSize;
            maxPeriod = seconds(mFrameCount) / mSampleRate * 3;
            ......
        }
       // 获得当前的已激活 track 数组。
        const SortedVector< wp<Track> >& activeTracks = mActiveTracks;
          ......
        /*
          ② prepareTracks_l 将检查 mActiveTracks 数组，判断是否有 AT 的数据需要处理。
          例如有些 AudioTrack 虽然调用了 start，但是没有及时 write 数据，这时就无须
          进行混音工作。我们待会再分析 prepareTracks_l 函数。
       */
        mixerStatus = prepareTracks_l(activeTracks, &tracksToRemove);
   }
    // MIXER_TRACKS_READY 表示 AT 已经把数据准备好了。
    if (LIKELY(mixerStatus == MIXER_TRACKS_READY)) {
        //③ 由混音对象进行混音工作，混音的结果放在 curBuf 中。
        mAudioMixer->process(curBuf);
        sleepTime = 0;// 等待时间设置为零，表示需要马上输出到 Audio HAL 中。
        standbyTime = systemTime() + kStandbyTimeInNsecs;
    }
       .......

    if (sleepTime == 0) {
        ......
       //④ 往 Audio HAL 的 OutputStream 中 write 混音后的数据，这是音频数据的最终归宿。
        int bytesWritten = (int)mOutput->write(curBuf, mixBufferSize);

        if (bytesWritten < 0) mBytesWritten -= mixBufferSize;
          ......
        mStandby = false;
    } else {
        usleep(sleepTime);
    }
```

```
            tracksToRemove.clear();
        }

        if (!mStandby) {
            mOutput->standby();
        }
        return false;
    }
```

从上面的分析可以看出，MixerThread 的线程函数大致工作流程是：

- 如果有通知信息或配置请求，则先完成这些工作。比如向监听者通知 AF 的一些信息，或者根据配置请求进行音量控制、声音设备切换等。
- 调用 prepareTracks _l 函数，检查活跃的 Tracks 是否有数据已准备好。
- 调用混音器对象 mAudioMixer 的 process，并且传入一个存储结果数据的缓冲，混音后的结果就存储在这个缓冲中。
- 调用代表音频输出设备的 AudioOutputStream 对象的 write，把结果数据写入设备。

其中，配置请求处理的工作将在 AudioPolicyService 的分析中以一个耳机插入处理实例进行讲解。这里主要分析代码中的②③两个步骤。

（2）prepareTracks_l 和 process 分析

prepareTracks_l 函数检查激活的 Track 数组，看看其中是否有数据等待使用，代码如下所示：

[-->AudioFlinger.cpp]

```
uint32_t AudioFlinger::MixerThread::prepareTracks_l(
                    const SortedVector<wp<Track>>& activeTracks,
                    Vector<sp<Track>> *tracksToRemove)
{

    uint32_t mixerStatus = MIXER_IDLE;
    // 激活 Track 的个数。
    size_t count = activeTracks.size();

    float masterVolume = mMasterVolume;
    bool  masterMute = mMasterMute;

  // 依次查询这些 Track 的情况。
   for (size_t i=0 ; i<count ; i++) {
        sp<Track> t = activeTracks[i].promote();
        if (t == 0) continue;

        Track* const track = t.get();
     // 怎么查？通过 audio_track_cblk_t 对象。
        audio_track_cblk_t* cblk = track->cblk();
   /*
```

```
    一个混音器可支持 32 个 Track，它内部有一个 32 元素的数组，name 函数返回的就是 Track 在
    这个数组中的索引。混音器每次通过 setActiveTrack 设置一个活跃 Track，
    后续所有操作都会针对当前设置的这个活跃 Track。
*/
    mAudioMixer->setActiveTrack(track->name());

    //下面这个判断语句决定了什么情况下 Track 数据可用。
 if (cblk->framesReady() && (track->isReady() || track->isStopped())
         && !track->isPaused() && !track->isTerminated())
    {
        ......
       /*
         设置活跃 Track 的数据提供者为 Track 本身，因为 Track 从 AudioBufferProvider
         派生。混音器工作时，需从 Track 得到待混音的数据，也就是 AT 写入的数据由混音
         器取出并消费。
       */
        mAudioMixer->setBufferProvider(track);
       //设置对应 Track 的混音标志。
        mAudioMixer->enable(AudioMixer::MIXING);
        ......
        //设置该 Track 的音量等信息，这在以后的混音操作中会使用。
        mAudioMixer->setParameter(param, AudioMixer::VOLUME0, left);
        mAudioMixer->setParameter(param, AudioMixer::VOLUME1, right);
        mAudioMixer->setParameter(
            AudioMixer::TRACK,
            AudioMixer::FORMAT, track->format());

        ......
        mixerStatus = MIXER_TRACKS_READY;
    } else {//如果不满足上面的条件，则走 else 分支。
        if (track->isStopped()) {
            track->reset();//reset 会清零读写位置，表示没有可读数据。
        }
        //如果处于这三种状态之一，则加入移除队列。
        if (track->isTerminated() || track->isStopped()
                || track->isPaused()) {
            tracksToRemove->add(track);
            mAudioMixer->disable(AudioMixer::MIXING);
        } else {
         //不处于上面三种状态时，表示暂时没有可读数据，则重试 mRetryCount 次。
         if (--(track->mRetryCount) <= 0) {
              tracksToRemove->add(track);
            } else if (mixerStatus != MIXER_TRACKS_READY) {
                mixerStatus = MIXER_TRACKS_ENABLED;
            }
          //禁止这个 Track 的混音。
            mAudioMixer->disable(AudioMixer::MIXING);
        ......
     }
    }
```

```
        }
        // 对那些被移除的 Track 做最后的处理。
        ......
        return mixerStatus;
    }
```

如果所有 Track 都准备就绪了，最重要的工作就是混音了。混音对象的 process 也就派上了用场。来看这个 process 函数，代码如下所示：

[-->AudioMixer.cpp]

```
void AudioMixer::process(void* output)
{
    mState.hook(&mState, output);//hook？这是一个函数指针
}
```

hook 是函数指针，它根据 Track 的个数和它的音频数据格式（采样率等）等情况，使用不同的处理函数。为了进一步了解混音器是如何工作的，需要先分析 AudioMixer 对象。

（3）AudioMixer 对象分析

AudioMixer 实现在 AudioMixer.cpp 中，先看构造函数：

[-->AudioMixer.cpp]

```
AudioMixer::AudioMixer(size_t frameCount, uint32_t sampleRate)
    :   mActiveTrack(0), mTrackNames(0), mSampleRate(sampleRate)
{
    mState.enabledTracks= 0;
    mState.needsChanged = 0;

    mState.frameCount    = frameCount;// 这个值等于音频输出对象的缓冲大小。

    mState.outputTemp    = 0;
    mState.resampleTemp = 0;
    //hook 初始化的时候为 process__nop，这个函数什么都不会做。
    mState.hook = process__nop;

    track_t* t = mState.tracks;//track_t 是和 Track 相对应的一个结构。

    // 最大支持 32 路混音，也很不错了。
    for (int i=0 ; i<32 ; i++) {
        ......
        t->channelCount = 2;
        t->enabled = 0;
        t->format = 16;
        t->buffer.raw = 0;

        t->bufferProvider = 0; // bufferProvider 为这一路 Track 的数据提供者。
        t->hook = 0;// 每一个 Track 也有一个 hook 函数。
```

```
        ......
    }
    int             mActiveTrack;
    uint32_t        mTrackNames;
    const uint32_t  mSampleRate;
    state_t         mState

}
```

其中，mState 是在 AudioMixer 类中定义的一个数据结构。

```
struct state_t {
        uint32_t        enabledTracks;
        uint32_t        needsChanged;
        size_t          frameCount;
        mix_t           hook;
        int32_t         *outputTemp;
        int32_t         *resampleTemp;
        int32_t         reserved[2];
  /*
    aligned 表示 32 字节对齐，由于 source insight 不认识这个标志，导致
    state_t 不能被解析。在看代码时，可以注释掉后面的 attribute，这样 source insight
     就可以识别 state_t 结构了。
 */
        track_t         tracks[32]; __attribute__((aligned(32)));
    };
```

AudioMixer 为 hook 准备了多个实现函数，来看：

- process__validate：根据 Track 的格式、数量等信息选择其他的处理函数。
- process__nop：什么都不做。
- process__genericNoResampling：普通无需重采样。
- process__genericResampling：普通需重采样。
- process__OneTrack16BitsStereoNoResampling：一路音频流，双声道，PCM16 格式，无需重采样。
- process__TwoTracks16BitsStereoNoResampling：两路音频流，双声道，PCM16 格式，无需重采样。

hook 最初的值为 process__nop，这一定不会是混音中最终使用的处理函数，难道有动态赋值的地方？是的。一起来看——

（4）杀鸡不用宰牛刀——根据情况选择处理函数

在 AF 的 prepare_l 中，会为每一个准备好的 Track 使能混音标志：

```
mAudioMixer->setBufferProvider(track);
mAudioMixer->enable(AudioMixer::MIXING);// 使能混音
```

请看 enable 的实现：

[-->AudioMixer.cpp]

```
status_t AudioMixer::enable(int name)
{
    switch (name) {
        case MIXING: {
            if (mState.tracks[ mActiveTrack ].enabled != 1) {
                mState.tracks[ mActiveTrack ].enabled = 1;
                // 注意这个 invalidateState 调用。
                invalidateState(1<<mActiveTrack);
            }
        } break;
        default:
            return NAME_NOT_FOUND;
    }
    return NO_ERROR;
}
```

[-->AudioMixer.cpp]

```
void AudioMixer::invalidateState(uint32_t mask)
 {
    if (mask) {
        mState.needsChanged |= mask;
        mState.hook = process__validate;// 将 hook 设置为 process_validate。
    }
 }
```

process_validate 会根据当前 Track 的情况选择不同的处理函数，所以不会出现杀鸡却用宰牛刀的情况。

[-->AudioMixer.cpp]

```
void AudioMixer::process__validate(state_t* state, void* output)
{
    uint32_t changed = state->needsChanged;
    state->needsChanged = 0;

   uint32_t enabled = 0;
    uint32_t disabled = 0;
    ......
    if (countActiveTracks) {
        if (resampling) {
            ......
         // 如果需要重采样，则选择 process__genericResampling。
            state->hook = process__genericResampling;
        } else {
            ......
            state->hook = process__genericNoResampling;
```

```
            if (all16BitsStereoNoResample && !volumeRamp) {
                if (countActiveTracks == 1) {
                //如果只有一个Track，则使用process__OneTrack16BitsStereoNoResampling。
                    state->hook = process__OneTrack16BitsStereoNoResampling;
                }
            }
        }
    }
    state->hook(state, output);
    ......
}
```

假设用例运行时，系统只有这么一个 Track，那么 hook 函数使用的就是 process__OneTrack16BitsStereoNoResampling 处理。process_XXX 函数会涉及很多数字音频处理的专业知识，先不用去讨论它。数据缓冲的消费工作是在这个函数中完成的，因此应重点关注它是如何通过 CB 对象使用数据缓冲的。

说明　这个数据消费和之前破解 AT 的过程中所讲的数据生产是对应的，先来提炼 AT 和 AF 在生产和消费这两个环节上与 CB 交互的流程。

（5）怎么消费数据

在 AudioTrack 中，曾讲到数据的生产流程：

- ObtainBuffer 得到一块数据缓冲。
- memcpy 数据到该缓冲。
- releaseBuffer 释放这个缓冲。

那么作为消费者，AudioFlinger 是怎么获得这些数据的呢？

[-->AudioMixer.cpp]

```
void AudioMixer::process__OneTrack16BitsStereoNoResampling(
    state_t* state, void* output)
{
    // 找到被激活的Track，此时只能有一个Track，否则就不会选择这个process函数了。
    const int i = 31 - __builtin_clz(state->enabledTracks);
    const track_t& t = state->tracks[i];

    AudioBufferProvider::Buffer& b(t.buffer);

    ......
    while (numFrames) {
        b.frameCount = numFrames;
        //BufferProvider就是Track对象，调用它的getNextBuffer获得可读数据缓冲。
        t.bufferProvider->getNextBuffer(&b);
        int16_t const *in = b.i16;
```

```
        ......
         size_t outFrames = b.frameCount;

           do {// 数据处理，也即混音。
                uint32_t rl = *reinterpret_cast<uint32_t const *>(in);
                in += 2;
                int32_t l = mulRL(1, rl, vrl) >> 12;
                int32_t r = mulRL(0, rl, vrl) >> 12;
               // 把数据复制给 out 缓冲。
                *out++ = (r<<16) | (l & 0xFFFF);
             } while (--outFrames);
         }
         numFrames -= b.frameCount;
        // 调用 Track 的 releaseBuffer 释放缓冲。
         t.bufferProvider->releaseBuffer(&b);
    }
}
```

bufferProvider 就是 Track 对象，总结一下它使用数据缓冲的调用流程：

- 调用 Track 的 getNextBuffer，得到可读数据缓冲。
- 调用 Track 的 releaseBuffer，释放数据缓冲。

现在来分析上面这两个函数：getNextBuffer 和 releaseBuffer。

（6）getNextBuffer 和 releaseBuffer 分析

先看 getNextBuffer。它从数据缓冲中得到一块可读空间，代码如下所示：

[-->AudioFlinger.cpp]

```
status_t AudioFlinger::PlaybackThread::Track::getNextBuffer(
                  AudioBufferProvider::Buffer* buffer)
{
     audio_track_cblk_t* cblk = this->cblk();// 通过 CB 对象完成
     uint32_t framesReady;
     //frameCount 为 AudioOutput 音频输出对象的缓冲区大小
     uint32_t framesReq = buffer->frameCount;

     ......
     // 根据 CB 的读写指针计算有多少帧数据可读。
     framesReady = cblk->framesReady();

     if (LIKELY(framesReady)) {
        uint32_t s = cblk->server; // 当前读位置
        // 可读的最大位置，为当前读位置加上 frameCount。
         uint32_t bufferEnd = cblk->serverBase + cblk->frameCount;
        //AT 可以通过 setLooping 设置播放的起点和终点，如果有终点的话，需要以 loopEnd
       // 作为数据缓冲的末尾。
        bufferEnd = (cblk->loopEnd < bufferEnd) ? cblk->loopEnd : bufferEnd;
        if (framesReq > framesReady) {
```

```
            // 如果要求的读取帧数大于可读帧数，则只能选择实际可读的帧数。
              framesReq = framesReady;
          }
         // 如果可读帧数的最后位置超过了 AT 设置的末端点，则需要重新计算可读帧数。
          if (s + framesReq > bufferEnd) {
              framesReq = bufferEnd - s;
          }
          // 根据读起始位置得到数据缓冲的起始地址，framesReq 参数用来做内部检查，防止出错。
           buffer->raw = getBuffer(s, framesReq);
           if (buffer->raw == 0) goto getNextBuffer_exit;

           buffer->frameCount = framesReq;
          return NO_ERROR;
      }

getNextBuffer_exit:
      buffer->raw = 0;
      buffer->frameCount = 0;
      return NOT_ENOUGH_DATA;
}
```

getNextBuffer 非常简单，不过就是根据 CB 记录的读写位置等计算可读的缓冲位置。下面来看 releaseBuffer 的操作。代码如下所示：

[-->AudioFlinger.cpp]

```
void AudioFlinger::ThreadBase::TrackBase::releaseBuffer(
            AudioBufferProvider::Buffer* buffer)
{
    buffer->raw = 0;
    mFrameCount = buffer->frameCount;//frameCount 为 getNextBuffer 中分配的可读帧数。
    step();// 调用 step 函数
    buffer->frameCount = 0;
}
```

[-->AudioFlinger.cpp]

```
bool AudioFlinger::ThreadBase::TrackBase::step() {
    bool result;
    audio_track_cblk_t* cblk = this->cblk();
  // 调用 stepServer 更新读位置。
    result = cblk->stepServer(mFrameCount);
    if (!result) {
        mFlags |= STEPSERVER_FAILED;
    }
    return result;
}
```

getNextBuffer 和 releaseBuffer 这两个函数相对比较简单。把它和 CB 交互的流程总结一下，为后面进行 CB 对象的分析做铺垫：

- getNextBuffer 通过 frameReady 得到可读帧数。
- getBuffer 函数将根据可读帧数等信息得到可读空间的首地址。
- releaseBuffer 通过 stepServer 更新读位置。

5. stop 分析

（1）TrackHandle 和 Track 的回收

来自 AT 的 stop 请求最终会通过 TrackHandle 这个 Proxy 交给 Track 的 stop 处理。请直接看 Track 的 stop，代码如下所示：

[-->AudioFlinger.cpp]

```
void AudioFlinger::PlaybackThread::Track::stop()
{
    sp<ThreadBase> thread = mThread.promote();
    if (thread != 0) {
        Mutex::Autolock _l(thread->mLock);
        int state = mState;// 保存旧的状态。
        if (mState > STOPPED) {
            mState = STOPPED;// 设置新状态为 STOPPED。
            PlaybackThread *playbackThread = (PlaybackThread *)thread.get();
            if (playbackThread->mActiveTracks.indexOf(this) < 0) {
                reset();// 如果该线程的活跃数组中没有 Track，则重置读写位置。
            }
        }
        // 和 APS 相关，我们不在这里讨论，它不直接影响 AudioFlinger。
        if (!isOutputTrack() && (state == ACTIVE || state == RESUMING)) {
            thread->mLock.unlock();
            AudioSystem::stopOutput(thread->id(),
                (AudioSystem::stream_type)mStreamType);
            thread->mLock.lock();
        }
    }
}
```

如果 Track 最初处于活跃数组中，那么这个 stop 函数无非是把 mState 设置为 STOPPED 了，但播放该怎么停止呢？请再回头看 prepareTrack_l 中的那个判断：

```
if (cblk->framesReady() && (track->isReady() || track->isStopped())
 && !track->isPaused() && !track->isTerminated())
```

假设 AT 写数据快，而 AF 消耗数据慢，那么上面这个判断语句在一定时间内是成立的，换言之，如果仅仅调用了 stop，还是会听到声音，该怎么办？在一般情况下，AT 端 stop 后会很快被 delete，这将导致 AF 端的 TrackHandle 也被 delete。

说明　在介绍 Track 和 TrackHandle 的一节中，曾在最后提到了那个野指针问题。相信读者这时候会知道那个问题的答案了，是吗？

看 TrackHandle 的析构函数，代码如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::TrackHandle::~TrackHandle() {
    mTrack->destroy();
}
```

[-->AudioFlinger.cpp]

```
void AudioFlinger::PlaybackThread::Track::destroy()
{
    sp<Track> keep(this);
    {
        sp<ThreadBase> thread = mThread.promote();
        if (thread != 0) {
            if (!isOutputTrack()) {
               // 和 AudioSystem 相关，以后再分析
                if (mState == ACTIVE || mState == RESUMING) {
                    AudioSystem::stopOutput(thread->id(),
                  (AudioSystem::stream_type)mStreamType);
                }
                AudioSystem::releaseOutput(thread->id());
            }
            Mutex::Autolock _l(thread->mLock);
            PlaybackThread *playbackThread = (PlaybackThread *)thread.get();
            // 调用回放线程对象的 destroyTrack_l。
            playbackThread->destroyTrack_l(this);
        }
    }
}
```

[-->AudioFlinger.cpp]

```
void AudioFlinger::PlaybackThread::destroyTrack_l(const sp<Track>& track)
{
    track->mState = TrackBase::TERMINATED;// 设置状态为 TERMINATED。
    if (mActiveTracks.indexOf(track) < 0) {
        mTracks.remove(track);// 如果不在 mActiveTracks 数组中，则把它从 mTracks 中去掉。
       // 由 PlaybackThread 的子类实现，一般就是做回收一些资源等工作。
        deleteTrackName_l(track->name());
    }
}
```

TrackHandle 的 delete 最后会导致它所代理的 Track 对象也被删除，那么 Client 对象什

么时候被回收呢？

（2）Client 的回收

直接看 TrackBase 的析构，因为 Track 的析构会导致它的基类 TrackBase 析构函数被调用，代码如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::ThreadBase::TrackBase::~TrackBase()
{
if (mCblk) {
      //placement new 出来的对象需要显示调用的析构函数。
        mCblk->~audio_track_cblk_t();
        if (mClient == NULL) {
            delete mCblk;// 先调用析构，再释放内存，这是 placement new 的用法。
        }
    }
    mCblkMemory.clear();
    if (mClient != NULL) {
        Mutex::Autolock _l(mClient->audioFlinger()->mLock);
        mClient.clear();// 如果 mClient 的强弱引用计数都为 0，则会导致该 Client 被 delete。
    }
}
```

资源回收的工作相对比较简单，这里就不做过多的讨论了，读者可自行分析研究。

说明 其实，要找到 TrackHandle 是什么时候被 delete 会更有难度。

7.3.3 audio_track_cblk_t 分析

前面讲解了 AudioFlinger 的工作方式，但 AT 和 AF，以及那个神秘的 CB 对象的工作原理一直都还没能讲解。对于 Audio 系统来说，如果最终也解决不了这个，真会有当年岳飞在朱仙镇被十二道金牌召回时一样的悲愤心情。幸好我们没遇到秦桧，那就奋起直追 ，去解决这个 CB 对象吧。

解决问题要有好的对策。还是从 AT 和 AF 这两端关于 CB 对象的调用流程开始分析吧，这一招可是屡试不爽啊！

1.AT 端的流程

AT 端作为数据的生产者（可称它为写者），它在 CB 对象中用 user 表示。它的调用流程是：

- 调用 framesAvailable，看看是否有空余的可写空间。
- 调用 buffer，获得写空间起始地址。
- 调用 stepUser，更新 user 的位置。

一起来分析一下，这几个函数都相当简单，力争一气呵成。

先调用 framesAvailable，看看当前剩余多少可写空间。假设是第一次进来，读者还在那等待数据，这样就不用考虑竞争等问题了，代码如下所示：

[-->AudioTrack.cpp::audio_track_cblk_t 的 framesAvailable() 及相关]

```
uint32_t audio_track_cblk_t::framesAvailable()
{
    Mutex::Autolock _l(lock);
    return framesAvailable_l();// 调用 framesAvailable_l。
}
int32_t audio_track_cblk_t::framesAvailable_l()
{
    uint32_t u = this->user; // 当前写者位置，此时为 0。
    uint32_t s = this->server; // 当前读者位置，此时也为 0。
    if (out) { // 对于音频输出，out 为 1。
        uint32_t limit = (s < loopStart) ? s : loopStart;
      // 由于不设置播放端点，所以 loopStart 是初始值 INT_MAX，limit=0。
        return limit + frameCount - u;
        // 返回 0+frameCount-0，也就是数据缓冲的全部大小。假设 frameCount=1024 帧。
    }
}
```

然后，调用 buffer 获得起始位置，buffer 返回一个地址，代码如下所示：

[-->AudioTrack.cpp]

```
void* audio_track_cblk_t::buffer(uint32_t offset) const
{
   //buffers 是数据缓冲的起始位置，offset 是计算出来的基于 userBase 的偏移。
  // 通过这种方式巧妙地把数据缓冲当做环形缓冲来处理。
   return (int8_t *)this->buffers + (offset - userBase) * this->frameSize;
}
```

当把数据写到缓冲后，调用 stepUser，代码如下所示：

[-->AudioTrack.cpp]

```
uint32_t audio_track_cblk_t::stepUser(uint32_t frameCount)
{
 /*
   framecount 表示写了多少帧，前面分配了 1024 帧，但写的数据可以比这个少。
    假设这一次写了 512 帧。
  */
    uint32_t u = this->user;//user 位置还没更新，此时 u=0；

    u += frameCount;//u 更新了，u=512。
    ......
/*
   userBase 还是初始值 0。可惜只写了 1024 的一半，所以 userBase 加不了。
   但这句话很重要，还记得前面的 buffer 调用吗？取数据地址的时候用 offset-userBase，
```

```
        一旦 user 位置到达缓冲的尾部，则 userBase 也会更新，这样 offset-userBase 的位置就会
        回到缓冲的头部，从头到尾这么反复循环，不就是一个环形缓冲了吗？非常巧妙！
    */
        if (u >= userBase + this->frameCount) {
            userBase += this->frameCount;
        }
        this->user = u;// 哦，user 位置也更新为 512 了，但是 useBase 还是 0。
        return u;
    }
```

假设写者这时因某种原因停止了写数据，而读者却被唤醒。

2.AF 端的流程

AF 端作为数据的消费者，它在 CB 中的表示是 server，可称它为读者。读者的使用流程是：

- 调用 framesReady 看是否有可读数据。
- 获得可读数据的起始位置，这个和上面的 buffer 调用基本一样，都是根据 offset 和 serverBase 来获得可读数据块的首地址的。
- 调用 stepServer 更新读位置。

现在来分析 framesReady 和 stepServer 这两个函数，framesReady 的代码如下所示：

[-->AudioTrack.cpp]

```
uint32_t audio_track_cblk_t::framesReady()
{
    uint32_t u = this->user; //u 为 512。
    uint32_t s = this->server;// 还没读呢，s 为零。

    if (out) {
        if (u < loopEnd) {
            return u - s;//loopEnd 也是 INT_MAX，所以这里返回 512，表示有 512 帧可读了。
        } else {
            Mutex::Autolock _l(lock);
            if (loopCount >= 0) {
                return (loopEnd - loopStart)*loopCount + u - s;
            } else {
                return UINT_MAX;
            }
        }
    } else {
        return s - u;
    }
}
```

可读数据地址的计算方法和前面的 buffer 调用一样，都是通过 server 和 serverBase 来计算的。接着看 stepServer，代码如下所示：

[-->AudioTrack.cpp]

```
bool audio_track_cblk_t::stepServer(uint32_t frameCount)
{
    status_t err;
     err = lock.tryLock();
    uint32_t s = this->server;

    s += frameCount; // 读了 512 帧，所以 s=512。
     ......
   // 没有设置循环播放，所以不走这个。
    if (s >= loopEnd) {
       s = loopStart;
       if (--loopCount == 0) {
            loopEnd = UINT_MAX;
            loopStart = UINT_MAX;
        }
}
     // 和 userBase 一样的处理。
    if (s >= serverBase + this->frameCount) {
        serverBase += this->frameCount;
    }
    this->server = s; //server 为 512 了。
    cv.signal(); // 读者读完了，触发一个同步信号，因为读者可能在等待可写的数据缓冲。
    lock.unlock();
    return true;
}
```

3. 真的是环形缓冲？

满足下面场景的缓冲可称为环形缓冲（假设数据缓冲最大为 1024 帧）：

- 写者先写 1024 帧，此后便无剩余空间可写。
- 读者读了前面的 512 帧，那么这 512 帧的数据空间就空出来了。
- 所以，写者就可以重新利用这空余的 512 帧空间了。

关键是第三步，写者是否跟踪了读者的位置，并充分利用了读者已使用过的数据空间。所以得回头看看写者 AT 是否把这 512 帧利用了。

先看写者写完 1024 帧后的情况，stepUser 中会有下面几句话：

```
if (u >= userBase + this->frameCount) {
    //u 为 1024，userBase 为 0，frameCount 为 1024。
    userBase += this->frameCount;// 好，userBase 也为 1024 了。
}
```

此时 userBase 更新为 1024 帧。再看写者获取可写空间的 framesAvailable_l 函数，按照以前的假设，应该返回 512 帧可写空间，代码如下所示：

[-->AudioTrack.cpp]

```
uint32_t audio_track_cblk_t::framesAvailable_l()
{
    uint32_t u = this->user;  //1024，写者上一次写完了整个1024帧空间。
    uint32_t s = this->server;//512，读者当前读到的位置。

    if (out) {
        uint32_t limit = (s < loopStart) ? s : loopStart;
        return limit + frameCount - u;//返回512。
    }
}
```

framesAvailable 返回了 512 帧，但可写空间的地址是否是从头开始的呢？要是从其他地方开始，情况就惨了。来看 buffer 中最后返回的可写空间地址：

```
return (int8_t *)this->buffers + (offset - userBase) * this->frameSize;
//offset是外界传入的基于userBase的一个偏移量，它的值是userBase+512，所以
//offset-userBase将得到从头开始的那段数据空间。真的是一个环形缓冲。
```

从上面的分析可看出，CB 对象通过 userBase 和 user 等几个变量，将一段有限长度的线性缓冲变成了一段无限长的缓冲，这不正是环形缓冲的精髓吗！

7.3.4 关于 AudioFlinger 的总结

总体来说，AF 比较复杂，再加上其他一些辅助类，cpp 文件中的代码有近 7000 行。其中 AudioFlinger.cpp 就有 4000 多行。这仅是从代码量来看，而使 AF 复杂的另外一个重要因素是它定义的内部类和它们之间的关系。

不过，从生产者和消费者的角度来看，AF 的工作还是比较简单明了的：

MixerThread 获取 Track 的数据，混音后写入音频输出设备。

关于 AudioFlinger 的学习和理解，有几个建议供大家参考：

- 首先要搞清数据传输的流程。虽然这只涉及 AT 和 AF 两个进程，但可以只在一端使用流程进行分析，例如 AF 的 start、stop 等。AT 和 AF 的工作流程也是它们的工作步骤，流程分析在 AT 和 AF 的破解过程中起到了重要作用，希望大家能掌握这个方法。
- 搞清 AF 中各个类的作用和派生关系。这样，在分析时就能准确定位到具体的实现函数。
- 搞清 CB 对象的工作原理和方式。如自己觉得只理解 AF 工作流程即可，CB 对象就不必过于深究。

7.4 AudioPolicyService 的破解

前面关于 AudioTrack 和 AudioFlinger 的分析，主要是针对 Audio 系统中数据传输方面

的，它们是 Audio 系统中不可或缺的部分。但 Audio 系统仅限于此吗？如果是这样，那么 AudioPolicyService 又是怎么一回事？另外，还要问几个实际问题：插入耳机后，声音是怎么从最开始的听筒输出变成耳机输出的呢？音量又是怎么控制的？MixerThread 的来历和 AudioPolicy 有怎样的关系？这些都与后面要分析的 AudioPolicyService 有关。

顾名思义，AudioPolicyService 是和 Audio 策略有关的，依本人对 AudioPolicy 的理解，策略比流程更复杂和难懂，对 APS 的分析与对 AT 及 AF 不同，因此不宜采用固定流程分析法，而应从下面三个步骤入手：

- 在分析 AudioPolicyService 的创建过程中，会讲解一些重要的概念和定义。
- 重新回到 AudioTrack 的分析流程，介绍其中和 AudioPolicy 有关的内容。
- 以一个耳机插入事件为实例，讲解 AudioPolicy 的处理。

7.4.1　AudioPolicyService 的创建

AudioPolicyService 和 AudioFlinger 都驻留于一个进程，之前在“MixerThread 的来历”一节中，曾简单介绍过 APS 的创建，现在需要仔细观察其中的内容。

1. 创建 AudioPolicyService

AudioPolicyService 的代码如下所示：

[-->AudioPolicyService.cpp]

```
AudioPolicyService::AudioPolicyService()
            : BnAudioPolicyService() ,
//mpPolicyManager 是 Audio 系统中的另一种 HAL 对象，它的类型是 AudioPolicyInterface。
            mpPolicyManager(NULL)
{
     char value[PROPERTY_VALUE_MAX];
    //TonePlayback 用于播放 Tone 音，Tone 包括按键音等。
    mTonePlaybackThread = new AudioCommandThread(String8(""));
   // 用于处理控制命令，例如路由切换、音量调节等。
    mAudioCommandThread = new AudioCommandThread(String8("ApmCommandThread"));

#if (defined GENERIC_AUDIO) || (defined AUDIO_POLICY_TEST)
    // 注意 AudioPolicyManagerBase 的构造函数，把 this 传进去了。
     mpPolicyManager = new AudioPolicyManagerBase(this);

#else
...
   // 使用硬件厂商实现的 AudioPolicyInterface。
    mpPolicyManager = createAudioPolicyManager(this);
#endif
  // 根据系统属性来判断照相机拍照时是否强制发声。为了防止偷拍，强制按快门的时候必须发出声音。
    property_get("ro.camera.sound.forced", value, "0");
    mpPolicyManager->setSystemProperty("ro.camera.sound.forced", value);
}
```

和 AudioFlinger 中的 AudioHardwareInterface 一样，在 APS 中可以见到另外一个 HAL 层对象 AudioPolicyInterface，为什么在 APS 中也会存在 HAL 对象呢？

如前所述，APS 主要是用来控制 Audio 系统的，由于各个硬件厂商的控制策略不可能完全一致，所以 Android 把这些内容抽象成一个 HAL 对象。下面来看这个 AudioPolicyInterface。

2. AudioPolicyInterface 分析

AudioPolicyInterface 比 AudioHardwareInterface 简单直接。这里，只需看几个重点函数即可，代码如下所示：

[-->AudioPolicyInterface.h]

```
class AudioPolicyInterface
{

public:
......
     // 设置设备的连接状态，这些设备包括耳机、蓝牙等。
    virtual status_t setDeviceConnectionState(
                          AudioSystem::audio_devices device,
                          AudioSystem::device_connection_state state,
                          const char *device_address) = 0;
    // 设置系统 Phone 状态，这些状态包括通话状态、来电状态等。
    virtual void setPhoneState(int state) = 0;
    // 设置 force_use 的 config 策略，例如通话中强制使用扬声器。
   virtual void setForceUse(AudioSystem::force_use usage,
                     AudioSystem::forced_config config) = 0;

     /*
        audio_io_handle_t 是 int 类型。这个函数的目的是根据传入的参数类型
      找到合适的输出句柄。这个句柄，在目前的 Audio 系统代表 AF 中的某个工作线程。
      还记得创建 AudioTrack 的时候传入的那个 output 值吗？它就是通过这个函数得到的。
      关于这个问题，马上会分析到。
    */
   virtual audio_io_handle_t getOutput(
          AudioSystem::stream_type stream,
                     uint32_t samplingRate = 0,
                     uint32_t format = AudioSystem::FORMAT_DEFAULT,
                     uint32_t channels = 0,
                     AudioSystem::output_flags flags =
                     AudioSystem::OUTPUT_FLAG_INDIRECT) = 0;

    // 下面两个函数以后会介绍。它们的第二个参数表示使用的是音频流类型。
  virtual status_t startOutput(audio_io_handle_t output,
                       AudioSystem::stream_type stream) = 0;

  virtual status_t stopOutput(audio_io_handle_t output,
                     AudioSystem::stream_type stream) = 0;
    ......

    // 音量控制：设置不同音频流的音量级别范围，例如 MUSIC 有 15 个级别的音量。
```

```
    virtual void initStreamVolume(AudioSystem::stream_type stream,
                                  int indexMin,
                                  int indexMax) = 0;

    // 设置某个音频流类型的音量级，例如觉得 music 声音太小时，可以调用这个函数提高音量级。
    virtual status_t setStreamVolumeIndex(AudioSystem::stream_type stream,
                                  int index) = 0;
};
```

从上面的分析可知，AudioPolicyInterface 主要提供了一些设备切换管理和音量控制的接口。每个厂商都有各自的实现方式。目前，Audio 系统提供了一个通用的实现类 AudioPolicyManagerBase，以前这个类是放在 hardware 目录下的，现在放到 framework 目录中了。图 7-12 展示了 AP 和 HAL 类之间的关系：

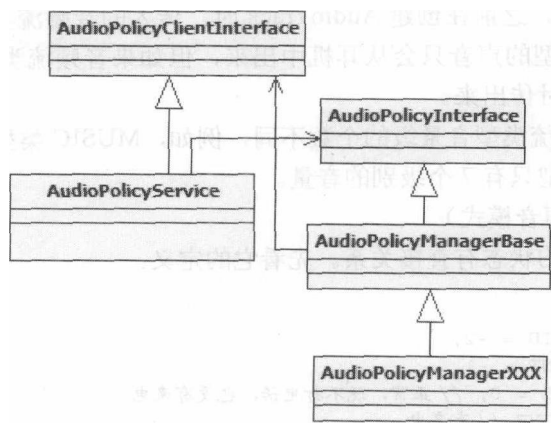


图 7-12　AudioPolicy 和 AudioPolicyInterface 的关系

其中：

- AudioPolicyService 有一个 AudioPolicyInterface 类型的对象。
- AudioPolicyManagerBase 有一个 AudioPolicyClientInterface 的对象。
- AudioPolicyInterface 中的一些函数在后面会分析到，这些函数中有很多参数都是以 AudioSystem::xxx 方式出现的，那么 AudioSystem 又是什么呢？

3. AudioSystem 介绍

AudioSystem 是一个 Native 类，这个类在 Java 层有对应的 Java 类，其中定义了一些重要的类型，比如音频流流程、音频设备等，这些都在 AudioSystem.h 中。下面来看其中的一些定义。

（1）stream type（音频流类型）

音频流类型，我们已在 AudioTrack 中见识过了，其完整定义如下：

```
enum stream_type {
```

```
        DEFAULT           =-1,// 默认
        VOICE_CALL        = 0,// 通话声
        SYSTEM            = 1,// 系统声，例如开关机提示
        RING              = 2,// 来电铃声
        MUSIC             = 3,// 媒体播放声
        ALARM             = 4,// 闹钟等的警告声
        NOTIFICATION      = 5,// 短信等的提示声
        BLUETOOTH_SCO     = 6,// 蓝牙 SCO
        ENFORCED_AUDIBLE = 7,// 强制发声，照相机的快门声就属于这个类型
        DTMF              = 8,//DTMF，拨号盘的按键声
        TTS               = 9,// 文本转语音，Text to Speech
        NUM_STREAM_TYPES
    };
```

音频流类型有什么用呢？为什么要做这种区分呢？它主要与两项内容有关：

- 设备选择：例如，之前在创建 AudioTrack 时，传入的音频流类型是 MUSIC，当插上耳机时，这种类型的声音只会从耳机中出来，但如果音频流类型是 RING，则会从耳机和扬声器中同时传出来。
- 音量控制：不同流类型音量级的个数不同，例如，MUSIC 类型有 15 个级别可供用户调节，而有些类型只有 7 个级别的音量。

（2）audio mode（声音模式）

audio mode 和电话的状态有直接关系。先看它的定义：

```
enum audio_mode {
        MODE_INVALID = -2,
        MODE_CURRENT = -1,
        MODE_NORMAL = 0, // 正常，既不打电话，也没有来电
        MODE_RINGTONE,// 有来电
        MODE_IN_CALL,// 通话状态
        NUM_MODES
};
```

为什么 Audio 需要特别强调 Phone 的状态呢？这必须和智能手机的硬件架构联系上。先看智能手机的硬件架构，如图 7-13 所示：

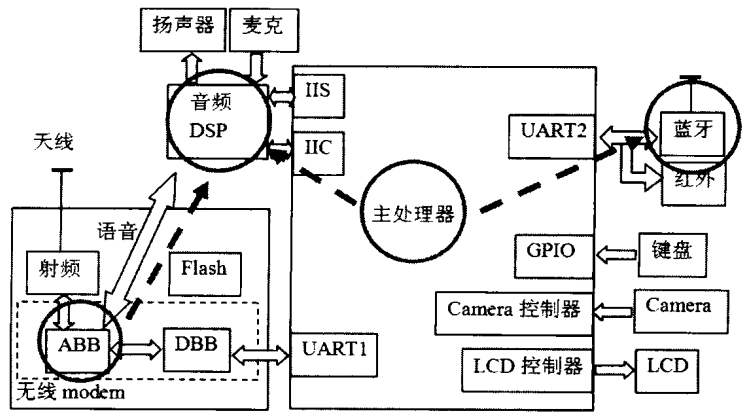


图 7-13　智能手机的硬件架构图

从图 7-13 中看出了什么？

- 系统有一个音频 DSP，声音的输入输出都要经过它（不考虑蓝牙的情况）。但它处理完的数字信号，需通过 D/A（数 / 模）转换后输出到最终的设备上，这些设备包括扬声器、听筒、耳机等。

注意　所谓的设备切换，是指诸如扬声器切换到听筒的情况，而前面常提到的音频输出设备，应该指的是 DSP。

- 系统有两个核心处理器，一个是应用处理的核心，叫 AP（Application Processor），可把它当做台式机上的 CPU，在这上面可以运行操作系统。另一个和手机通信相关，一般叫 BP（Baseband Processor 基带处理器），可把它当做台式机上的“猫”。
- AP 和 BP 都能向音频 DSP 发送数据，它们在硬件的通路上互不干扰。于是就出现了一个问题，即如果两个 P 同时往 DSP 发送数据，而互相之间没有协调，那么就可能出现通话声和音乐声混杂的情况。谁还会用这样的手机？所以打电话时，将由 AP 上的 Phone 程序主动设置 Audio 系统的 mode，在这种 mode 下，Audio 系统会做一些处理，例如把 music 音量调小等。
- 注意图中的蓝牙了吗？它没有像 AP 那样直接和音频 DSP 相连，所以音频数据需要单独发给蓝牙设备。如果某种声音要同时从蓝牙和扬声器发出，亦即一份数据要往两个地方发送，便满足了 AudioFlinger 中 DuplicatingThread 出现的现实要求。

注意　蓝牙设备实际上会建立两条数据通路：SCO 和 A2DP。A2DP 和高质量立体声有关，且必须由 AudioFlinger 向它发送数据。所以“音频数据需要单独发送给蓝牙设备”，其中这个设备实际上是指蓝牙的 A2DP 设备。蓝牙技术很复杂，有兴趣的读者可以自行研究。

（3）force use 和 config（强制使用及配置）

大家知道，手机通话时可以选择扬声器输出，这就是强制使用的案例。Audio 系统对此有很好的支持。它涉及两个方面：

- 强制使用何种设备，例如使用扬声器、听筒、耳机等。它由 forced_config 控制，代码如下所示：

```
enum forced_config {
        FORCE_NONE,
        FORCE_SPEAKER,   // 强制使用扬声器
        FORCE_HEADPHONES,
        FORCE_BT_SCO,
        FORCE_BT_A2DP,
        FORCE_WIRED_ACCESSORY,
        FORCE_BT_CAR_DOCK,
        FORCE_BT_DESK_DOCK,
        NUM_FORCE_CONFIG,
        FORCE_DEFAULT = FORCE_NONE
}
```

- 在什么情况下需要强制使用，是通话的强制使用，还是听音乐的强制使用？这须由 force_use 控制，代码如下所示：

```
enum force_use {
    FOR_COMMUNICATION,// 通话情况，注意前缀是 FOR_XXX。
    FOR_MEDIA,// 听音乐等媒体相关的情况。
    FOR_RECORD,
    FOR_DOCK,
    NUM_FORCE_USE
}
```

所以，AudioPolicyInterface 的 setForceUse 函数，就是设置在什么情况下强制使用什么设备：

```
virtual void setForceUse(AudioSystem::force_use usage,// 什么情况
                AudioSystem::forced_config config // 什么设备
                ) = 0;
```

（4）输出设备的定义

前面曾反复提到输出设备。这些设备在软件中是怎么表示的呢？ Audio 定义了很多输出设备，来看其中几个：

```
enum audio_devices {
        // output devices
        DEVICE_OUT_EARPIECE = 0x1,  // 听筒
        DEVICE_OUT_SPEAKER = 0x2,   // 扬声器
        DEVICE_OUT_WIRED_HEADSET = 0x4,   // 耳机
        DEVICE_OUT_WIRED_HEADPHONE = 0x8,  // 另外一种耳机
        DEVICE_OUT_BLUETOOTH_SCO = 0x10,  // 蓝牙相关，SCO 用于通话的语音传输。
        DEVICE_OUT_BLUETOOTH_SCO_HEADSET = 0x20,
        DEVICE_OUT_BLUETOOTH_SCO_CARKIT = 0x40,
        DEVICE_OUT_BLUETOOTH_A2DP = 0x80, // 蓝牙相关，A2DP 用于立体声传输。
        DEVICE_OUT_BLUETOOTH_A2DP_HEADPHONES = 0x100,
        DEVICE_OUT_BLUETOOTH_A2DP_SPEAKER = 0x200,
        DEVICE_OUT_AUX_DIGITAL = 0x400,
        DEVICE_OUT_DEFAULT = 0x8000,
        ......
}
```

至此，AudioSystem 中常用的定义都已见过了，现在要回到 APS 的创建上了。对于这个例子，将使用 Generic 的设备，所以会直接创建 AudioPolicyManagerBase 对象，这个对象实现了 AudioPolicyInterface 的所有功能。一起来看。

说明 实际上很多硬件厂商实现的 AudioPolicyInterface，基本上是直接使用这个 AudioPolicyManagerBase 的。

4. AudioPolicyManagerBase 分析

AudioPolicyManagerBase 类在 AudioPolicyManagerBase.cpp 中实现，先来看它的构造函数：

[-->AudioPolicyManagerBase.cpp]

```
AudioPolicyManagerBase::AudioPolicyManagerBase(
```

```
                                AudioPolicyClientInterface *clientInterface)
                               :mPhoneState(AudioSystem::MODE_NORMAL), mRingerMode(0),
                                mMusicStopTime(0),mLimitRingtoneVolume(false)
{
    //APS 实现了 AudioPolicyClientInterface 接口。
    mpClientInterface = clientInterface;// 这个 clientInterface 就是 APS 对象。

     // 清空强制使用配置。
     for (int i = 0; i < AudioSystem::NUM_FORCE_USE; i++) {
         mForceUse[i] = AudioSystem::FORCE_NONE;
     }

     // 输出设备有听筒和扬声器。
     mAvailableOutputDevices = AudioSystem::DEVICE_OUT_EARPIECE |
                               AudioSystem::DEVICE_OUT_SPEAKER;
    // 输入设备是内置的麦克（学名叫传声器）。
     mAvailableInputDevices = AudioSystem::DEVICE_IN_BUILTIN_MIC;

#ifdef WITH_A2DP  // 和蓝牙立体声有关。
     mA2dpOutput = 0;
     mDuplicatedOutput = 0;
     mA2dpDeviceAddress = String8("");
#endif
     mScoDeviceAddress = String8(""); //SCO 主要用于通话。
   /*
     ①创建一个 AudioOutputDescriptor 对象，这个对象用来记录并维护与
     输出设备（相当于硬件的音频 DSP）相关的信息，例如使用该设备的流个数、各个流的音量、
     该设备所支持的采样率、采样精度等。其中，有一个成员 mDevice 用来表示目前使用的输出设备，
     例如耳机、听筒、扬声器等。
  */
     AudioOutputDescriptor *outputDesc = new AudioOutputDescriptor();
     outputDesc->mDevice = (uint32_t)AudioSystem::DEVICE_OUT_SPEAKER;

   /*
     ②还记得 MixerThread 的来历吗？ openOutput 导致 AF 创建了一个工作线程。
     该函数返回的是一个工作线程索引号。
   */
     mHardwareOutput = mpClientInterface->openOutput(&outputDesc->mDevice,
                                        &outputDesc->mSamplingRate,
                                        &outputDesc->mFormat,
                                        &outputDesc->mChannels,
                                        &outputDesc->mLatency,
                                        outputDesc->mFlags);

      ......
     //AMB 维护了一个与设备相关的 key/value 集合，下面将对应的信息加到该集合中。
     addOutput(mHardwareOutput, outputDesc);
     // ③设置输出设备，就是设置 DSP 的数据流到底从什么设备出去，这里设置的是从扬声器出去。
      setOutputDevice(mHardwareOutput,
                      (uint32_t)AudioSystem::DEVICE_OUT_SPEAKER, true);
     }
    // ④更新不同策略使用的设备。
```

```
        updateDeviceForStrategy();
    }
```

关于 AMB 这个小小的构造函数，有几个重要点需要介绍：

（1）AudioOutputDescriptor 和 openOutput

AudioOutputDescriptor 对象，是 AMB 用来控制和管理音频输出设备的，从硬件上看，它代表的是 DSP 设备。关于这一点已在注释中做出了说明，这里就不再赘述。

另一个重要点是 openOutput 函数。该函数的实现由 APS 来完成。之前曾分析过，它最终会在 AF 中创建一个混音线程（不考虑 DirectOutput 的情况），该函数返回的是该线程在 AF 中的索引号，亦即：

```
mHardwareOutput = mpClientInterface->openOutput(.....)
```

mHardwareOutput 表示的是 AF 中一个混音线程的索引号。这里涉及一个非常重要的设计问题：AudioFlinger 到底会创建多少个 MixerThread？有两种设计方案：

- 一种是一个 MixerThread 对应一个 Track。如果这样，AMB 仅使用一个 mHardwareOutput 恐怕不够用。
- 另一种是用一个 MixerThread 支持 32 路的 Track 数据，多路数据通过 AudioMixer 混音对象在软件层面进行混音。

这里用的是第二种方案，当初设计时为何不用一个 MixerThread 支持一路 Track，然后把混音的工作交给硬件来完成呢？我觉得原因之一是如果采用一个线程一个 Track 的方式，会非常难管理和控制，另一个原因是多线程比较浪费资源。

采用第二种方法（也就是现有的方案），极大地简化了 AMB 的工作量。图 7-14 展示了 AMB 和 AF 及 MixerThread 之间的关系：

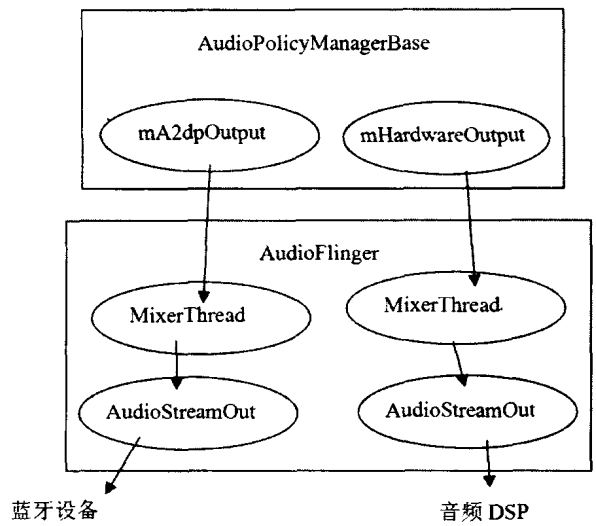


图 7-14　AF、AMB 及 MixerThread 之间的关系

图 7-14 表明：

- AMB 中除了 mHardwareOutput 外，还有一个 mA2dpOutput，它对应的 MixerThread 专往代表蓝牙 A2DP 设备的 AudioStreamOut 上发送数据。关于这个问题，在后面分析 DuplicatingThread 时可以见到。

注意　使用 mA2dpOutput 需要蓝牙设备连接上才会有意义。

- 除了蓝牙外，系统中一般也就只有图 7-14 右边的这么一个 MixerThread 了，所以 AMB 通过 mHardwareOutput 就能控制整个系统的声音，这真是一劳永逸。

说明　关于这一点，现在通过 setOutputDevice 来分析。

（2）setOutputDevice

现在要分析的调用是 setOutputDevice，目的是为 DSP 选择一个合适的输出设备。注意它的第一个参数是传入的 mHardwareOutput，它最终会找到代表 DSP 的 AudioStreamOut 对象，第二个参数是一个设备号，代码如下所示：

[-->AudioPolicyManagerBase.cpp]

```
void AudioPolicyManagerBase::setOutputDevice(audio_io_handle_t output,
                              uint32_t device, bool force, int delayMs)
{
    AudioOutputDescriptor *outputDesc = mOutputs.valueFor(output);
    // 判断是否是 Duplicate 输出，和蓝牙 A2DP 有关，后面再做分析。
    if (outputDesc->isDuplicated()) {
        setOutputDevice(outputDesc->mOutput1->mId, device, force, delayMs);
        setOutputDevice(outputDesc->mOutput2->mId, device, force, delayMs);
        return;
  }
 // 初始设置的输出设备为听筒和扬声器。
  uint32_t prevDevice = (uint32_t)outputDesc->device();
  if ((device == 0 || device == prevDevice) && !force) {
        return;
   }
 // 现在设置新的输出设备为扬声器，注意这是软件层面上的设置。
  outputDesc->mDevice = device;
    ......
 /*
   还需要硬件也做相应的设置，主要是告诉 DSP 把它的输出切换到某个设备上，根据之前的分析可知，
   这个请求要发送到 AF 中的 MixerThread 上，因为只有它拥有代表输出设备的 AudioStreamOut
   对象。
*/
   AudioParameter param = AudioParameter();
   param.addInt(String8(AudioParameter::keyRouting), (int)device);
/*
 上面的配置参数将投递到 APS 的消息队列，而 APS 中创建的 AudioCommandThread
 会取出这个配置参数，再投递给 AF 中对应的 MixerThread，最终由 MixerThread 处理。
```

```
        这个流程，将在耳机插拔事件处理中进行分析。
    */
        mpClientInterface->setParameters(mHardwareOutput,
                                                param.toString(),delayMs);
        ......
    }
```

setOutputDevice 要实现的目的已经很明确，只是实现的过程比较烦琐而已。其间没有太多复杂之处，读者可自行研究，以加深对 Audio 系统的了解。

（3）Audio Strategy

现调用的函数是 updateDeviceForStrategy，这里会引出一个 strategy 的概念。先看 updataDeviceForStrategy 函数：

[-->AudioPolicyManagerBase.cpp]

```
void AudioPolicyManagerBase::updateDeviceForStrategy()
{
    for (int i = 0; i < NUM_STRATEGIES; i++) {
        mDeviceForStrategy[i] =
                            getDeviceForStrategy((routing_strategy)i, false);
    }
}
```

关于 getDeviceForStrategy，在耳机插拔事件中再做分析，现在先看 routing_strategy 的定义，代码如下所示：

[-->getDeviceForStrategy.h::routing_strategy]

```
//routing_strategy: 路由策略
enum routing_strategy {
            STRATEGY_MEDIA,
            STRATEGY_PHONE,
            STRATEGY_SONIFICATION,
            STRATEGY_DTMF,
            NUM_STRATEGIES
        }
```

它是在 AudioPolicyManagerBase.h 中定义的，一般的应用程序不会使用这个头文件。这个 routing_strategy 有什么用处呢？从名字上看，似乎和路由的选择有关系，但 AudioSystem 定义的是 stream type，这两者之间会有什么关系吗？有，而且还很紧密。这个关系通过 AMB 的 getStrategy 就可以看出来。它会从指定的流类型得到对应的路由策略，代码如下所示：

[-->AudioPolicyManagerBase.cpp]

```
AudioPolicyManagerBase::getStrategy(AudioSystem::stream_type stream)
{
```

```
    switch (stream) {
    case AudioSystem::VOICE_CALL:
    case AudioSystem::BLUETOOTH_SCO:
        return STRATEGY_PHONE; //PHONE 路由策略
    case AudioSystem::RING:
    case AudioSystem::NOTIFICATION:
    case AudioSystem::ALARM:
    case AudioSystem::ENFORCED_AUDIBLE:
        return STRATEGY_SONIFICATION; //SONIFICATION 路由策略
    case AudioSystem::DTMF:
        return STRATEGY_DTMF;  //DTMF 路由策略
    default:
        LOGE("unknown stream type");
    case AudioSystem::SYSTEM:
    case AudioSystem::TTS:
    case AudioSystem::MUSIC:
        return STRATEGY_MEDIA;//media 路由策略
    }
}
```

从这个函数中可看出，AudioSystem 使用的流类型并不是和路由直接相关的，AMB 或 AudioPolicy 内部是使用 routing_strategy 来控制路由策略的。

5. 小结

这一节涉及不少新东西，但本人觉得最重要的还是图 7-13 和图 7-14。其中：

- 图 7-13 展示了智能手机的硬件架构，通过和 Audio 相关的架构设计，我们能理解 Audio 系统的设计缘由。
- 图 7-14 展示了 APS 和 AF 内部联系的纽带，后续 APS 的控制无非就是找到对应的 MixerThread，给它发送控制消息，最终由 MixerThread 将控制信息传给对应的代表音频输出设备的 HAL 对象。

7.4.2　重回 AudioTrack

从前文所介绍的内容可知，AudioTrack 在调用 createTrack 时，会传入一个 audio_handle_t，这个值表示 AF 中某个工作线程的索引号，而它又是从 APS 中得到的。那么，这中间又有哪些曲折的经历呢？

先回顾一下 AudioTrack 的 set 函数。

1. 重回 set

先来看相应的代码，如下所示：

[--->AudioTrack.cpp]

```
status_t AudioTrack::set(int streamType,uint32_t sampleRate,int format,
                         int channels,int frameCount,uint32_t flags,
```

```
                                callback_t cbf,void* user,int notificationFrames,
                                const sp<IMemory>& sharedBuffer, bool threadCanCallJava)
{

    ......
    // 得到 AF 中一个工作线程的索引号。
      audio_io_handle_t output = AudioSystem::getOutput(
                                        (AudioSystem::stream_type)streamType,
                                         sampleRate, format, channels,
                                         (AudioSystem::output_flags)flags);

    ......
// 创建 Track，最终会调到 AF 的 createTrack。
status_t status = createTrack(streamType, sampleRate, format, channelCount,
                                       frameCount, flags, sharedBuffer, output);

.......
}
```

再看 AudioSystem 是如何实现 getOutput 的，代码如下所示：

[-->AudioSystem.cpp]

```
audio_io_handle_t AudioSystem::getOutput(stream_type stream,
                                         uint32_t samplingRate,
                                         uint32_t format,
                                         uint32_t channels,
                                         output_flags flags)
{
    audio_io_handle_t output = 0;
    .......
    if (output == 0) {
        const sp<IAudioPolicyService>& aps =
                                          AudioSystem::get_audio_policy_service();
        if (aps == 0) return 0;
        // 调用 AP 的 getOutput 函数。
        output = aps->getOutput(stream, samplingRate, format, channels, flags);
        if ((flags & AudioSystem::OUTPUT_FLAG_DIRECT) == 0) {
            Mutex::Autolock _l(gLock);
          // 把这个 stream 和 output 的对应关系保存到 map 中。
            AudioSystem::gStreamOutputMap.add(stream, output);
        }
    }
    return output;
}
```

这里调用了 AP 的 getOutput，来看：

[-->AudioPolicyService.cpp]

```
audio_io_handle_t AudioPolicyService::getOutput(
                            AudioSystem::stream_type stream, uint32_t samplingRate,
```

```
                                uint32_t format,uint32_t channels,
                                AudioSystem::output_flags flags)
{
    //和硬件厂商的实现相关，所以交给 AudioPolicyInterface 处理。
    //这里将由 AudioPolicyManagerBase 处理。
     Mutex::Autolock _l(mLock);
     return mpPolicyManager->getOutput(stream, samplingRate, format, channels,
                                                    flags);
}
```

[->AudioPolicyManagerBase.cpp]

```
audio_io_handle_t AudioPolicyManagerBase::getOutput(
                        AudioSystem::stream_type stream, uint32_t samplingRate,
                        uint32_t format,uint32_t channels,
                        AudioSystem::output_flags flags)
{
    audio_io_handle_t output = 0;
    uint32_t latency = 0;
   //根据流类型得到对应的路由策略，这个我们已经见过了，MUSIC 类型返回 MUSIC 策略。
   routing_strategy strategy = getStrategy((AudioSystem::stream_type)stream);
  //根据策略得到使用这个策略的输出设备（指扬声器之类的），以后再看这个函数。
    uint32_t device = getDeviceForStrategy(strategy);
......
   //看这个设备是不是与蓝牙的 A2DP 相关。
   uint32_t a2dpDevice = device & AudioSystem::DEVICE_OUT_ALL_A2DP;
   if (AudioSystem::popCount((AudioSystem::audio_devices)device) == 2) {
#ifdef WITH_A2DP
     //对于有 A2DP 支持的，a2dpUsedForSonification 函数直接返回 true。
  if (a2dpUsedForSonification() && a2dpDevice != 0) {
       //和 DuplicatingThread 相关，以后再看。
        output = mDuplicatedOutput;
 } else
#endif
       {
         output = mHardwareOutput; //使用非蓝牙的混音输出线程。
        }
    } else {
#ifdef WITH_A2DP
        if (a2dpDevice != 0) {
            //使用蓝牙的混音输出线程。
            output = mA2dpOutput;
        } else
#endif
        {
           output = mHardwareOutput;
        }
    }
  return output;
}
```

终于明白了！原来，AudioSystem 的 getOutput 就是想找到 AF 中的一个工作线程。为什么这个线程号会由 AP 返回呢？是因为 Audio 系统需要：

- 根据流类型找到对应的路由策略。
- 根据该策略找到合适的输出 device（指扬声器、听筒之类的）。
- 根据 device 选择 AF 中合适的工作线程，例如是蓝牙的 MixerThread，还是 DSP 的 MixerThread，或者是 DuplicatingThread。
- AT 根据得到的工作线程索引号，最终将在对应的工作线程中创建一个 Track。之后，AT 的数据将由该线程负责处理。

下面用图 7-15 来回顾一下上面 AT、AF、AP 之间的交互关系。

图 7-15 充分展示了 AT、AF 和 AP 之间复杂微妙的关系。关系虽复杂，但目的却单纯。读者在分析时一定要明确目的。下面从目的开始，反推该流程：

- AT 的目的是把数据发送给对应的设备，例如蓝牙、DSP 等。
- 代表输出设备的 HAL 对象由 MixerThread 线程持有，所以要找到对应的 MixerThread。
- AP 维护流类型和输出设备（耳机、蓝牙耳机、听筒等）之间的关系，不同的输出设备使用不同的混音线程。
- AT 根据自己的流类型向 AudioSystem 查询，希望得到对应的混音线程号。

这样，三者精妙配合，便达到了预期目的。

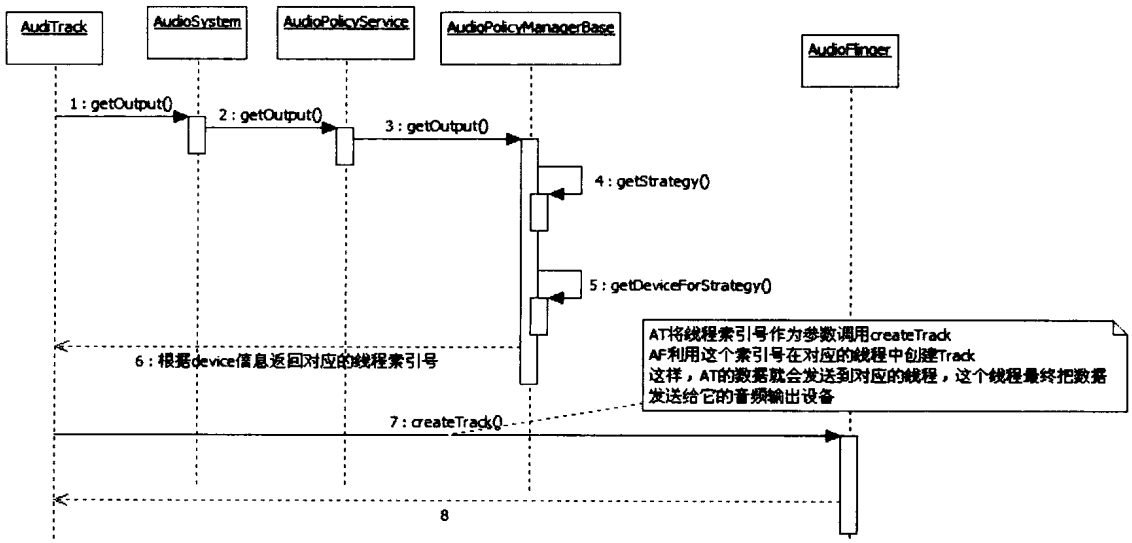


图 7-15　Audio 三巨头的交互关系

2. 重回 start

现在要分析的就是 start 函数。AT 的 start 虽没有直接与 AP 交互，但在 AF 的 start 中却

和 AP 有着交互关系。其代码如下所示：

[-->AudioFlinger.cpp]

```
status_t AudioFlinger::PlaybackThread::Track::start()
{
    status_t status = NO_ERROR;
     sp<ThreadBase> thread = mThread.promote();
     ......
        if (!isOutputTrack() && state != ACTIVE && state != RESUMING) {
            thread->mLock.unlock();
          // 调用 AudioSystem 的 startOutput。
            status = AudioSystem::startOutput(thread->id(),
                                    (AudioSystem::stream_type)mStreamType);
             thread->mLock.lock();
        }
       PlaybackThread *playbackThread = (PlaybackThread *)thread.get();
       playbackThread->addTrack_l(this);// 把这个 Track 加入到活跃的 Track 数组中。
    return status;
}
```

下面来看 AudioSystem 的 startOutput，代码如下所示：

[-->AudioSystem.cpp]

```
status_t AudioSystem::startOutput(audio_io_handle_t output,
                                        AudioSystem::stream_type stream)
{
   const sp<IAudioPolicyService>& aps =
                                      AudioSystem::get_audio_policy_service();
  if (aps == 0) return PERMISSION_DENIED;
    // 调用 AP 的 startOutput，最终由 AMB 完成实际功能。
    return aps->startOutput(output, stream);
}
```

[-->AudioPolicyManagerBase.cpp]

```
status_t AudioPolicyManagerBase::startOutput(audio_io_handle_t output,
                                        AudioSystem::stream_type stream)
{
    // 根据 output 找到对应的 AudioOutputDescriptor。
    ssize_t index = mOutputs.indexOfKey(output);
    AudioOutputDescriptor *outputDesc = mOutputs.valueAt(index);

   // 找到对应流使用的路由策略。
    routing_strategy strategy = getStrategy((AudioSystem::stream_type)stream);

   // 增加 outputDesc 中该流的使用计数，1 表示增加 1。
    outputDesc->changeRefCount(stream, 1);
   //getNewDevice 将得到一个设备，setOutputDevice 将使用这个设备进行路由切换。
```

```
        // 至于 setOutputDevice，我们在分析耳机插入事件时再来讲解。
        setOutputDevice(output, getNewDevice(output));
        // 设置音量，读者可自行分析。
        checkAndSetVolume(stream, mStreams[stream].mIndexCur, output,
                          outputDesc->device());

    return NO_ERROR;
}
```

再看 getNewDevice，它和音频流的使用计数有关系：

[-->AudioPolicyManagerBase.cpp]

```
uint32_t AudioPolicyManagerBase::getNewDevice(audio_io_handle_t output,
                                              bool fromCache)
{
    uint32_t device = 0;

    AudioOutputDescriptor *outputDesc = mOutputs.valueFor(output);
    /*
     isUsedByStrategy 判断某个策略是否正在被使用，之前曾通过 changeRefCount 为
     MUSIC 流使用计数增加了 1，所以使用 MUSIC 策略的个数至少为 1，这表明，此设备正在使用该策略。
     一旦得到当前 outputDesc 使用的策略，便可根据该策略找到对应的设备。
     注意 if 和 else 的顺序，它代表了系统优先使用的策略，以第一个判断为例，
     假设系统已经插上耳机，并且处于通话状态中，而且强制使用了扬声器，那么声音都从扬声器出。
     这时，如果想听音乐的话，则应首先使用 STRATEGY_PHONE 的对应设备，此时就是扬声器。
     所以音乐将从扬声器出来，而不是耳机。上面仅是举例，具体的情况还要综合考虑 Audio
     系统中的其他信息。另外如果 fromCache 为 true，将直接从内部保存的旧信息中得到设备，
     关于这个问题，在后面的耳机插入事件处理中再做分析。
    */
    if (mPhoneState == AudioSystem::MODE_IN_CALL ||
        outputDesc->isUsedByStrategy(STRATEGY_PHONE)) {
        device = getDeviceForStrategy(STRATEGY_PHONE, fromCache);
    } else if (outputDesc->isUsedByStrategy(STRATEGY_SONIFICATION)) {
        device = getDeviceForStrategy(STRATEGY_SONIFICATION, fromCache);
    } else if (outputDesc->isUsedByStrategy(STRATEGY_MEDIA)) {
        device = getDeviceForStrategy(STRATEGY_MEDIA, fromCache);
    } else if (outputDesc->isUsedByStrategy(STRATEGY_DTMF)) {
        device = getDeviceForStrategy(STRATEGY_DTMF, fromCache);
    }
    return device;
}
```

这里，有一个问题需要关注：

为什么 startOutput 函数会和设备切换有关系呢？

仅举一个例子，帮助理解这一问题。AudioTrack 创建时可设置音频流类型，假设第一个 AT 创建时使用的是 MUSIC 类型，那么它将使用耳机出声（假设耳机已经连接上）。这时第二个 AT 创建了，它使用的是 RING 类型，它对应的策略应是 SONIFACATION，这个策略的优

先级比 MUSIC 要高（因为 getNewDevice 的判断语句首先会判断 isUsedByStrategy(STRATEGY_SONIFICATION)），所以这时需要把设备切换为耳机加扬声器（假设这种类型的声音需要从耳机和扬声器同时输出）。startOutput 的最终结果，是这两路的 Track 声音都将从耳机和扬声器中听到。当第二路 AT 调用 stop 时，对应音频流类型使用计数会减一，这会导致新的路由切换，并重新回到只有耳机的情况，这时第一路 AT 的声音会恢复为只从耳机输出。

提醒　读者可自行分析 stop 的处理方式，基本上是 start 的逆向处理过程。

3. 小结

这一节主要讲解了 AudioTrack 和 AP 之间的交互，总结为以下两点：

- AT 通过 AP 获取 AF 中的工作线程索引号，这决定了数据传输的最终目标是谁，比如是音频 DSP 还是蓝牙。
- AT 的 start 和 stop 会影响 Audio 系统的路由切换。

读完这一节，读者可能只会对与工作线程索引有关的内容印象较深刻，毕竟这个决定了数据传输的目的地。至于与路由切换有关的知识，可能就不太了解了。下面通过分析一个应用场景来启发、加深对路由切换相关知识的理解。

7.4.3　声音路由切换实例分析

路由这个词听上去很专业，其实它的目的很简单，就是为 DSP 选择数据出口，例如是从耳机、听筒还是扬声器传出。下面分析这样一个场景：

假设我们在用扬声器听歌，这时把耳机插上，会发生什么呢？

1. 耳机插拔事件处理

耳机插上后，系统会发一个广播，Java 层的 AudioService 会接收这个广播，其中的内部类 AudioServiceBroadcastReceiver 会处理该事件，处理函数是 onReceive。

这段代码在 AudioSystem.java 中。一起来看——

（1）耳机插拔事件接收

看这段代码，如下所示：

[-->AudioSystem.java::AudioServiceBroadcastReceiver 的 onReceive()]

```
private class AudioServiceBroadcastReceiver extends BroadcastReceiver{
@Override
        public void onReceive(Context context, Intent intent) {
            String action = intent.getAction();
        ......
                    // 如果该事件是耳机插拔事件。
        else if (action.equals(Intent.ACTION_HEADSET_PLUG)) {
                // 取得耳机的状态。
```

```
                int state = intent.getIntExtra("state", 0);
                int microphone = intent.getIntExtra("microphone", 0);
                if (microphone != 0) {
                  // 查看已连接设备是不是已经有了耳机，耳机的设备号为 0x4，
                  // 这个和 AudioSystem.h 定义的设备号是一致的。
                    boolean isConnected = mConnectedDevices.containsKey(
                            AudioSystem.DEVICE_OUT_WIRED_HEADSET);
                    // 如果之前有耳机而现在没有，则认为是耳机拔出事件。
                    if (state == 0 && isConnected) {
                        // 设置 Audio 系统的设备连接状态，耳机为 Unavailable。
                        AudioSystem.setDeviceConnectionState(
                                AudioSystem.DEVICE_OUT_WIRED_HEADSET,
                                AudioSystem.DEVICE_STATE_UNAVAILABLE,
                                "");
                    // 从已连接设备中去掉耳机设备。
                    mConnectedDevices.remove(AudioSystem.DEVICE_OUT_WIRED_HEADSET);
                } // 如果 state 为 1，并且之前没有耳机连接，则处理这个耳机插入事件。
                  else if (state == 1 && !isConnected){
                    // 设置 Audio 系统的设备连接状态，耳机为 Available。
                        AudioSystem.setDeviceConnectionState(
                            AudioSystem.DEVICE_OUT_WIRED_HEADSET,
                            AudioSystem.DEVICE_STATE_AVAILABLE,
                            "");
                    // 在已连接设备中增加耳机。
                    mConnectedDevices.put(
                             new Integer(AudioSystem.DEVICE_OUT_WIRED_HEADSET),
                             "");
                    }
                  }
    ......
    }
```

从上面的代码中可看出，不论耳机是插入还是拔出，都会调用 AudioSystem 的 setDeviceConnectionState 函数。

（2）setDeviceConnectionState：设置设备连接状态

这个函数被定义为 Native 函数。下面是它的定义：

[-->AudioSystem.java]

```
    public static native int setDeviceConnectionState(int device, int state,
                                                       String device_address);
// 注意我们传入的参数，device 为 0X4 表示耳机，state 为 1，device_address 为 ""。
```

该函数的 Native 实现，在 android_media_AudioSystem.cpp 中，对应函数是：

[->android_media_AudioSystem.cpp]

```
static int android_media_AudioSystem_setDeviceConnectionState(
                                        JNIEnv *env, jobject thiz, jint
                                        device, jint state, jstring device_address)
```

```
{
    const char *c_address = env->GetStringUTFChars(device_address, NULL);
    int status = check_AudioSystem_Command(
    // 调用 Native AudioSystem 的 setDeviceConnectionState。
    AudioSystem::setDeviceConnectionState(
                static_cast <AudioSystem::audio_devices>(device),
                static_cast <AudioSystem::device_connection_state>(state),
                c_address));
    env->ReleaseStringUTFChars(device_address, c_address);
    return status;
}
```

从 AudioSystem.java 转入到 AudioSystem.cpp，现在来看 Native 的对应函数：

[-->AudioSystem.cpp]

```
status_t AudioSystem::setDeviceConnectionState(audio_devices device,
                                                device_connection_state state,
                                                const char *device_address)
{
    const sp<IAudioPolicyService>& aps =
                                        AudioSystem::get_audio_policy_service();
    if (aps == 0) return PERMISSION_DENIED;
    // 转到 AP 去，最终由 AMB 处理。
    return aps->setDeviceConnectionState(device, state, device_address);
}
```

Audio 代码不厌其烦地把函数调用从这一类转移到另外一类，请直接看 AMB 的实现：

[-->AudioPolicyManagerBase.cpp]

```
status_t AudioPolicyManagerBase::setDeviceConnectionState(
                  AudioSystem::audio_devices device,
                  AudioSystem::device_connection_state state,
                  const char *device_address)
{

    // 一次只能设置一个设备。
    if (AudioSystem::popCount(device) != 1) return BAD_VALUE;
     ......
     // 根据设备号判断是不是输出设备，耳机肯定属于输出设备。
   if (AudioSystem::isOutputDevice(device)) {
       switch (state)
        {
        case AudioSystem::DEVICE_STATE_AVAILABLE:
          // 处理耳机插入事件，mAvailableOutputDevices 保存已连接的设备。
           // 这个耳机是刚连上的，所以不走下面 if 分支。
            if (mAvailableOutputDevices & device) {
                // 启用过了，就不再启用了。
                return INVALID_OPERATION;
```

```
            }

            // 现在已连接设备中多了一个耳机。
            mAvailableOutputDevices |= device;
              ....
        }
        // ① getNewDevice 之前已分析过了，这次再看。
        uint32_t newDevice = getNewDevice(mHardwareOutput, false);
        // ② 更新各种策略使用的设备。
         updateDeviceForStrategy();
        // ③ 设置新的输出设备。
         setOutputDevice(mHardwareOutput, newDevice);

    ......
}
```

这里面有三个比较重要的函数，前面也已提过，现将其再进行一次较深入的分析，旨在加深读者对它的理解。

（3）getNewDevice

来看代码，如下所示：

[->AudioPolicyManagerBase.cpp]

```
uint32_t AudioPolicyManagerBase::getNewDevice(audio_io_handle_t output,
                                                bool fromCache)
{  // 注意我们传入的参数，output 为 mHardwardOutput，fromCache 为 false。
    uint32_t device = 0;
    // 根据 output 找到对应的 AudioOutputDescriptor，这个对象保存了一些信息。
    AudioOutputDescriptor *outputDesc = mOutputs.valueFor(output);

    if (mPhoneState == AudioSystem::MODE_IN_CALL ||
         outputDesc->isUsedByStrategy(STRATEGY_PHONE))
    {
        device = getDeviceForStrategy(STRATEGY_PHONE, fromCache);
    }
     else if (outputDesc->isUsedByStrategy(STRATEGY_SONIFICATION))
    {
        device = getDeviceForStrategy(STRATEGY_SONIFICATION, fromCache);
     }
     else if (outputDesc->isUsedByStrategy(STRATEGY_MEDIA))
    {
         // 应用场景是正在听歌，所以会走这个分支。
        device = getDeviceForStrategy(STRATEGY_MEDIA, fromCache);
     }
    else if (outputDesc->isUsedByStrategy(STRATEGY_DTMF))
    {
        device = getDeviceForStrategy(STRATEGY_DTMF, fromCache);
     }
     return device;
}
```

策略是怎么和设备联系起来的呢？秘密就在 getDeviceForStrategy 中，来看代码：

[-->AudioPolicyManagerBase.cpp]

```
uint32_t AudioPolicyManagerBase::getDeviceForStrategy(
                                        routing_strategy strategy, bool fromCache)
{
    uint32_t device = 0;

    if (fromCache) {// 如果为 true，则直接取之前的旧值。
       return mDeviceForStrategy[strategy];
    }
  // 如果 fromCache 为 false，则需要重新计算策略所对应的设备。
    switch (strategy) {
    case STRATEGY_DTMF:// 先处理 DTMF 策略的情况。
        if (mPhoneState != AudioSystem::MODE_IN_CALL) {
           // 如果不处于电话状态，则 DTMF 策略和 MEDIA 策略对应同一个设备。
            device = getDeviceForStrategy(STRATEGY_MEDIA, false);
            break;
        }
         // 如果处于电话状态，则 DTMF 策略和 PHONE 策略用同一个设备。
    case STRATEGY_PHONE:
         // 是 PHONE 策略的时候，先要考虑是不是用户强制使用了某个设备，例如强制使用扬声器。
        switch (mForceUse[AudioSystem::FOR_COMMUNICATION]) {
         ......
        case AudioSystem::FORCE_SPEAKER:
         ...... // 如果没有蓝牙，则选择扬声器。
            device = mAvailableOutputDevices &
                                            AudioSystem::DEVICE_OUT_SPEAKER;
            break;
        }
    break;
    case STRATEGY_SONIFICATION://SONIFICATION 策略
        if (mPhoneState == AudioSystem::MODE_IN_CALL) {
            /*
              如果处于来电状态，则和 PHONE 策略用同一个设备。例如通话过程中我们强制使用
              扬声器，那么如果这个时候按拨号键，按键声则也会从扬声器出来。
            */
            device = getDeviceForStrategy(STRATEGY_PHONE, false);
            break;
        }
        device = mAvailableOutputDevices & AudioSystem::DEVICE_OUT_SPEAKER;
        // 如果不处于电话状态，则 SONIFICATION 和 MEDIA 策略用同一个设备。
    case STRATEGY_MEDIA: {
        //AUX_DIGITAL 值为 0x400，耳机不满足该条件。
        uint32_t device2 = mAvailableOutputDevices &
                                        AudioSystem::DEVICE_OUT_AUX_DIGITAL;
        if (device2 == 0) {
           // 也不满足 WIRED_HEADPHONE 条件。
            device2 = mAvailableOutputDevices &
```

```
                                    AudioSystem::DEVICE_OUT_WIRED_HEADPHONE;
        }
        if (device2 == 0) {
            // 满足这个条件，所以 device2 为 0x4，WIRED_HEADSET。
            device2 = mAvailableOutputDevices &
                                    AudioSystem::DEVICE_OUT_WIRED_HEADSET;
        }
        if (device2 == 0) {
            device2 = mAvailableOutputDevices &
                                    AudioSystem::DEVICE_OUT_SPEAKER;
        }
        device |= device2; // 最终 device 为 0x4，WIRED_HEADSET
      } break;
    default:
        break;
    }
    return device;
}
```

getDeviceForStrategy 是一个比较复杂的函数。它的复杂在于选取设备时，需考虑很多情况。简单的分析仅能和读者一起领略一下它的风采，在实际工作中反复琢磨，或许才能掌握其中的奥妙。

好，getNewDevice 将返回耳机的设备号 0x4。下一个函数是 updateDeviceForStrategy。这个函数和 getNewDevice 没有什么关系，因为它没用到 getNewDevice 的返回值。

（4）updateDeviceForStrategy

同样来看相应的代码，如下所示：

[-->AudioPolicyManagerBase.cpp]

```
void AudioPolicyManagerBase::updateDeviceForStrategy()
{
    for (int i = 0; i < NUM_STRATEGIES; i++) {
    // 重新计算每种策略使用的设备，并保存到 mDeviceForStrategy 中，起到了 cache 的作用。
    mDeviceForStrategy[i] = getDeviceForStrategy((routing_strategy)i, false);
    }
}
```

updateDeviceForStrategy 会重新计算每种策略对应的设备。

另外，如果 updateDeviceForStrategy 和 getNewDevice 互换位置，会节省很多不必要的调用。如：

```
updateDevicdForStrategy();// 先更新策略
// 使用 cache 中的设备，节省一次重新计算。
uint32_t newDevice = getNewDevice(mHardwareOutput, true);
```

不必讨论这位码农的功过了，现在来看最后一个函数 setOutputDevice。它会对新选出来

的设备做如何处理呢？

（5）setOutputDevice

继续看 setOutputDevice 的代码，如下所示：

[-->AudioPolicyManagerBase.cpp]

```
void AudioPolicyManagerBase::setOutputDevice(audio_io_handle_t output,
                                    uint32_t device,bool force, int delayMs)
{
    ......
   // 把这个请求要发送到 output 对应的 AP 工作线程中。
    AudioParameter param = AudioParameter();
   // 参数是 key/vlaue 键值对的格式。
    param.addInt(String8(AudioParameter::keyRouting), (int)device);
   //mpClientInterface 是 AP 对象，由它处理。
    mpClientInterface->setParameters(mHardwareOutput,
                                        param.toString(), delayMs);

   // 设置音量，不做讨论，读者可自行分析。
    applyStreamVolumes(output, device, delayMs);
 }
```

setParameters 最终会调用 APS 的 setParameters，代码如下所示：

[-->AudioPolicyService.cpp]

```
void AudioPolicyService::setParameters(audio_io_handle_t ioHandle,
                              const String8& keyValuePairs, int delayMs)
{
    // 把这个请求加入到 AudioCommandThread 中处理。
    mAudioCommandThread->parametersCommand((int)ioHandle,
                                            keyValuePairs, delayMs);
}
```

AudioPolicyService 创建时会同时创建两个线程，其中一个用于处理各种请求。现在看看它是怎么做的。

2. AudioCommandThread

AudioCommandThread 有一个请求处理队列，AP 负责往该队列中提交请求，而 AudioCommandThread 在它的线程函数 threadLoop 中处理这些命令。请直接看命令是如何处理的。

说明　这种通过一个队列来协调两个线程的方法，在多线程编程中非常常见，它也属于生产者 / 消费者模型。

（1）AudioCommandThread 中的处理

先来看相应的代码，如下所示：

[-->AudioPolicyService.cpp]

```
bool AudioPolicyService::AudioCommandThread::threadLoop()
{
    nsecs_t waitTime = INT64_MAX;

    mLock.lock();
    while (!exitPending())
    {
        while(!mAudioCommands.isEmpty()) {
            nsecs_t curTime = systemTime();
            if (mAudioCommands[0]->mTime <= curTime) {
                AudioCommand *command = mAudioCommands[0];
                mAudioCommands.removeAt(0);
                mLastCommand = *command;

                switch (command->mCommand) {
                case START_TONE:
                   ......
                case STOP_TONE:
                   ......  //TONE 处理
                    mLock.lock();
                    }break;
                case SET_VOLUME: {
                    // 设置音量
                    delete data;
                    }break;
                case SET_PARAMETERS: {
                    // 处理路由设置请求。
                     ParametersData *data = (ParametersData *)command->mParam;
                    // 转到 AudioSystem 处理，mIO 的值为 mHardwareOutput。
                     command->mStatus = AudioSystem::setParameters(
                                              data->mIO,
                                             data->mKeyValuePairs);
                     if (command->mWaitStatus) {
                         command->mCond.signal();
                         mWaitWorkCV.wait(mLock);
                     }
                     delete data;
                     }break;
                ......
                default:
                }
}
```

Audio 系统真是非常绕！来看 AudioSystem 的 setParameters。

（2）AudioSystem 的 setParameters

AudioSystem 将设置请求转移给 AudioFlinger 处理，代码如下所示：

[-->AudioSystem.cpp]

```
status_t AudioSystem::setParameters(audio_io_handle_t ioHandle,
                                    const String8& keyValuePairs)
{
     const sp<IAudioFlinger>& af = AudioSystem::get_audio_flinger();
    // 果然是交给 AF 处理，ioHandle 看来一定就是工作线程索引号了。
    return af->setParameters(ioHandle, keyValuePairs);
}
```

离真相越来越近了，接着看代码，如下所示：

[-->AudioFlinger.cpp]

```
status_t AudioFlinger::setParameters(int ioHandle,
                                     const String8& keyValuePairs)
{
    status_t result;
    // ioHandle == 0 表示和混音线程无关，需要直接设置到 HAL 对象中。
    if (ioHandle == 0) {
        AutoMutex lock(mHardwareLock);
        mHardwareStatus = AUDIO_SET_PARAMETER;
       // 调用 AudioHardwareInterface 的参数设置接口。
        result = mAudioHardware->setParameters(keyValuePairs);
        mHardwareStatus = AUDIO_HW_IDLE;
        return result;
    }

     sp<ThreadBase> thread;
    {
        Mutex::Autolock _l(mLock);
       // 根据索引号找到对应的混音线程。
        thread = checkPlaybackThread_l(ioHandle);
      }
      // 我们只有一个 MixerThread，交给它处理，这又是一个命令处理队列。
      result = thread->setParameters(keyValuePairs);
      return result;
    }
    return BAD_VALUE;
}
```

好了，最终的请求处理在 MixerThread 的线程函数中，接着往下看。

（3）MixerThread 最终处理

代码如下所示：

[-->AudioFlinger.cpp]

```
bool AudioFlinger::MixerThread::threadLoop()
{
    ....
    while (!exitPending())
    {
        processConfigEvents();
        mixerStatus = MIXER_IDLE;
        { // scope for mLock
            Mutex::Autolock _l(mLock);
           // checkForNewParameters_l 最有嫌疑
            if (checkForNewParameters_l()) {
                ...
            }
       ......// 其他处理
}
```

[-->AudioFlinger.cpp]

```
bool AudioFlinger::MixerThread::checkForNewParameters_l()
{
    bool reconfig = false;

    while (!mNewParameters.isEmpty()) {
        status_t status = NO_ERROR;
        String8 keyValuePair = mNewParameters[0];
        AudioParameter param = AudioParameter(keyValuePair);
        int value;
       ......
      // 路由设置需要硬件参与，所以直接交给代表音频输出设备的 HAL 对象处理。
       status = mOutput->setParameters(keyValuePair);

      return reconfig;
}
```

至此，路由设置所经历的一切轨迹，我们都已清晰地看到了，可总还有点意犹未尽的感觉，HAL 的 setParameters 到底是怎么工作的呢？不妨再来看一个实际的 HAL 对象处理例子。

（4）真实设备的处理

这个实际的 Hardware，位于 hardware/msm7k/libaudio-qsd8k 的 Hardware.cpp 中，它提供了一个音频处理实际的例子，这个 Hardware 针对的是高通公司的硬件。直接看它是怎么处理音频输出对象 setParameters 的，代码如下所示：

[-->AudioHardware.cpp AudioStreamOutMSM72xx::setParameters()]

```
status_t AudioHardware::AudioStreamOutMSM72xx::setParameters(
const String8& keyValuePairs)
{
```

```
    AudioParameter param = AudioParameter(keyValuePairs);
    String8 key = String8(AudioParameter::keyRouting);
    status_t status = NO_ERROR;
    int device;

    if (param.getInt(key, device) == NO_ERROR) {
        mDevices = device;
        // 调用 doRouting，mHardware 就是 AudioHardware 对象。
        status = mHardware->doRouting();
        param.remove(key);
    }
    ......
    return status;
}
```

[-->AudioHardware.cpp]

```
status_t AudioHardware::doRouting()
{
    Mutex::Autolock lock(mLock);
    uint32_t outputDevices = mOutput->devices();
    status_t ret = NO_ERROR;
    int sndDevice = -1;
      ......
      // 做一些判断，最终由 doAudioRouteOrMute 处理。
     if ((vr_mode_change) || (sndDevice != -1 && sndDevice != mCurSndDevice)) {
        ret = doAudioRouteOrMute(sndDevice);
        mCurSndDevice = sndDevice;
      }

    return ret;
}
```

[-->AudioHardware.cpp]

```
status_t AudioHardware::doAudioRouteOrMute(uint32_t device)
{
    uint32_t rx_acdb_id = 0;
    uint32_t tx_acdb_id = 0;
   // 只看看就行，对于硬件相关的代码，咱们就是打打酱油。
   return do_route_audio_dev_ctrl(device,
   mMode == AudioSystem::MODE_IN_CALL, rx_acdb_id, tx_acdb_id);
}
```

[-->AudioHardware.cpp]

```
static status_t do_route_audio_dev_ctrl(uint32_t device, bool inCall,
                    uint32_t rx_acdb_id, uint32_t tx_acdb_id)
{
```

```
    uint32_t out_device = 0, mic_device = 0;
    uint32_t path[2];
    int fd = 0;
  //打开音频控制设备
   fd = open("/dev/msm_audio_ctl", O_RDWR);
   path[0] = out_device;
   path[1] = rx_acdb_id;
    // 通过 ioctl 切换设备，一般系统调用都是返回 -1 表示出错，这里返回 0 表示出错。
    if (ioctl(fd, AUDIO_SWITCH_DEVICE, &path)) {
       close(fd);
       return -1;
    }
.....
}
```

7.4.4 关于 AudioPolicy 的总结

AudioPolicy 是 Audio 系统中最难懂的内容，重要原因之一是，它不像 AT 和 AF 那样有比较固定的工作流程，所以对它的把握和理解一定要结合具体的使用场景，尤其是路由切换这一块，涉及很多方面的知识，如音频流类型、当前可用设备等。我希望读者至少要理解以下两点：

- ❑ AP 和 AF 的关系，关于这一部分的内容，读者应彻底弄懂图 7-13。
- ❑ 关于设备连接导致的路由切换处理，读者要理解这期间的处理流程。

7.5 拓展思考

7.5.1 DuplicatingThread 破解

DuplicatingThread 需要与蓝牙结合起来使用，它的存在与 Audio 硬件结构息息相关。读者可参考图 7-12“智能手机硬件架构图”来理解。当一份数据同时需要发送给 DSP 和蓝牙 A2DP 设备时，DuplicatingThread 就派上用场了。在分析 DuplicatingThread 前，还是应该了解一下它的来龙去脉。

1. DuplicatingThread 的来历

DuplicatingThread 和蓝牙的 A2DP 设备有关系。可先假设有一个蓝牙立体声耳机已经连接上了，接着从 setDeviceConnectionState 开始分析，代码如下所示：

[-->AudioPolicyManagerBase.cpp]

```
status_t AudioPolicyManagerBase::setDeviceConnectionState(
                        AudioSystem::audio_devices device,
                        AudioSystem::device_connection_state state,
                        const char *device_address)
```

```
{
        ......
        switch (state)
        {
          case AudioSystem::DEVICE_STATE_AVAILABLE:

            mAvailableOutputDevices |= device;

#ifdef WITH_A2DP
            if (AudioSystem::isA2dpDevice(device)) {
                 // 专门处理 A2DP 设备的连接。
                status_t status = handleA2dpConnection(device, device_address);
            }
#endif
            ......
}
```

对于 A2DP 设备，有专门的函数 handleA2dpConnection 处理，代码如下所示：

[-->AudioPolicyManagerBase.cpp]

```
status_t AudioPolicyManagerBase::handleA2dpConnection(
                                        AudioSystem::audio_devices device,
                                        const char *device_address)
{
    AudioOutputDescriptor *outputDesc = new AudioOutputDescriptor();
    outputDesc->mDevice = device;
    // 先为 mA2dpOutput 创建一个 MixerThread，这个和 mHardwareOutput 一样。
    mA2dpOutput = mpClientInterface->openOutput(&outputDesc->mDevice,
                                            &outputDesc->mSamplingRate,
                                            &outputDesc->mFormat,
                                            &outputDesc->mChannels,
                                            &outputDesc->mLatency,
                                            outputDesc->mFlags);
    if (mA2dpOutput) {
    /*
    a2dpUsedForSonification 永远返回 true，表示属于 SONIFCATION 策略的音频流声音需要
    同时从蓝牙和 DSP 中传出。至于 SONIFCATION 策略的音频流类型可查看前面关于 getStrategy 的
    分析，来电铃声、短信通知等属于这一类。
    */
        if (a2dpUsedForSonification()) {
           /*
            创建一个 DuplicateOutput，注意它的参数，第一个是蓝牙 MixerThread，
            第二个是 DSP MixerThread。
          */
            mDuplicatedOutput = mpClientInterface->openDuplicateOutput(
                                                mA2dpOutput, mHardwareOutput);
        }
        if (mDuplicatedOutput != 0 ||
            !a2dpUsedForSonification()) {
```

```
                if (a2dpUsedForSonification()) {
                    // 创建一个 AudioOutputDescriptor 对象。
                    AudioOutputDescriptor *dupOutputDesc = newAudioOutputDescriptor();
                    dupOutputDesc->mOutput1 = mOutputs.valueFor(mHardwareOutput);
                    dupOutputDesc->mOutput2 = mOutputs.valueFor(mA2dpOutput);
                    ......
                    // 保存 mDuplicatedOutput 和 dupOutputDesc 键值对。
                    addOutput(mDuplicatedOutput, dupOutputDesc);
                    ......
                }
            }
    }
    ......
    }
```

这里最重要的函数是 openDuplicateOutput。它和 openOutput 一样，最终的处理都是在 AF 中。去那里看看，代码如下所示：

[-->AudioFlinger.cpp]

```
int AudioFlinger::openDuplicateOutput(int output1, int output2)
{
    Mutex::Autolock _l(mLock);
    //output1 对应蓝牙的 MixerThread。
    MixerThread *thread1 = checkMixerThread_l(output1);
   //output2 对应 DSP 的 MixerThread。
    MixerThread *thread2 = checkMixerThread_l(output2);

    // ①创建 DuplicatingThread，注意它第二个参数使用的是代表蓝牙的 MixerThread。
     DuplicatingThread *thread = new DuplicatingThread(this,
                                                       thread1, ++mNextThreadId);
    // ②加入代表 DSP 的 MixerThread。
    thread->addOutputTrack(thread2);
    mPlaybackThreads.add(mNextThreadId, thread);
    return mNextThreadId;// 返回 DuplicatingThread 的索引
}
```

从现在起，MixerThread 要简写为 MT，而 DuplicatingThread 则简写为 DT。

这里面有两个重要的函数调用，一起来看。

2.DuplicatingThread 和 OutputTrack

先看 DT 的构造函数，代码如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::DuplicatingThread::DuplicatingThread(const sp<AudioFlinger>&
                audioFlinger, AudioFlinger::MixerThread* mainThread,int id)
                :   MixerThread(audioFlinger, mainThread->getOutput(), id),
                    mWaitTimeMs(UINT_MAX)
```

```
{
//DT 是 MT 的派生类，所以先要完成基类的构造，还记得 MT 的构造吗？它会创建一个 AudioMixer 对象。
  mType = PlaybackThread::DUPLICATING;
// 把代表 DSP 的 MT 加入进来，咱们看看。
    addOutputTrack(mainThread);
}
```

[-->AudioFlinger.cpp]

```
void AudioFlinger::DuplicatingThread::addOutputTrack(MixerThread *thread)
{
   int frameCount = (3 * mFrameCount * mSampleRate) / thread->sampleRate();
   // 构造一个 OutputTrack，它的第一个参数是 MT。
    OutputTrack *outputTrack = new OutputTrack((ThreadBase *)thread,
                                            this, mSampleRate, mFormat,
                                            mChannelCount,frameCount);
    if (outputTrack->cblk() != NULL) {
        thread->setStreamVolume(AudioSystem::NUM_STREAM_TYPES, 1.0f);
        // 把这个 outputTrack 加入到 mOutputTracks 数组保存。
        mOutputTracks.add(outputTrack);
        updateWaitTime();
    }
}
```

此时，当下面两句代码执行完：

```
DuplicatingThread *thread = new DuplicatingThread(this,thread1, ++mNextThreadId);
thread->addOutputTrack(thread2);
```

DT 则分别构造了两个 OutputTrack，一个对应蓝牙的 MT，另一个对应 DSP 的 MT。现在来看 OutputTrack 为何方神圣，代码如下所示：

[-->AudioFlinger.cpp]

```
AudioFlinger::PlaybackThread::OutputTrack::OutputTrack(
        const wp<ThreadBase>& thread, DuplicatingThread *sourceThread,
        uint32_t sampleRate, int format,int channelCount,int frameCount)
        :Track(thread, NULL, AudioSystem::NUM_STREAM_TYPES, sampleRate,
               format, channelCount, frameCount, NULL),// 最后这个参数为 NULL。
         mActive(false), mSourceThread(sourceThread)
{
     /*
       OutputTrack 从 Track 派生，所以需要先调用基类的构造，还记得 Track 构造函数
       中的事情吗？它会创建一块内存，至于是不是共享内存，由 Track 构造函数的最后一个参数决定。
        如果该值为 NULL，表示没有客户端参与，则会在本进程内创建一块内存，这块内存的结构如
        图 7-4 所示，前面为 CB 对象，后面为数据缓冲。
     */
    // 下面的这个 thread 对象为 MT。
    PlaybackThread *playbackThread = (PlaybackThread *)thread.unsafe_get();
    if (mCblk != NULL) {
```

```
            mCblk->out = 1;// 表示 DT 将往 MT 中写数据。
            // 和前面所分析的 AT、AF 中的处理何其相似!
            mCblk->buffers = (char*)mCblk + sizeof(audio_track_cblk_t);
            mCblk->volume[0] = mCblk->volume[1] = 0x1000;
            mOutBuffer.frameCount = 0;
            // 把这个 Track 加到 MT 的 Track 中。
            playbackThread->mTracks.add(this);
        }
    }
```

明白了吗？图 7-16 表示的是 openDuplicateOutput 的结果：

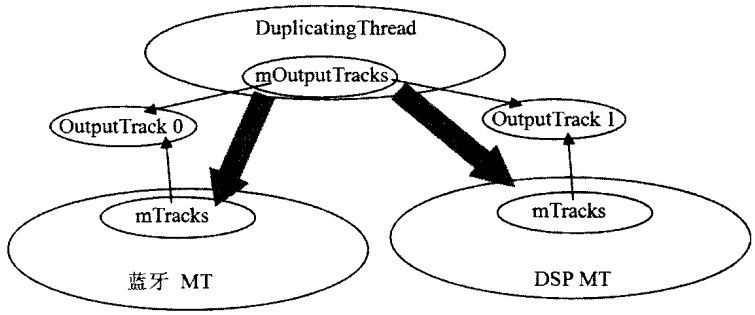


图 7-16 openDuplicateOutput 的结果示意图

图 7-16 说明（以蓝牙 MT 为例）：

- 蓝牙 MT 的 Track 中有一个成员为 OutputTrack0。
- DT 的 mOutputTracks 也有一个成员指向 OutputTrack0。这就好像 DT 是 MT 的客户端一样，它和前面分析的 AT 是 AF 的客户端类似。
- 灰色部分代表数据传递用的缓冲。

3.DT 的客户端 AT

DT 是从 MT 中派生的，根据 AP 和 AT 的交互流程可知，当 AT 创建的流类型对应策略为 SONIFACATION 时，它会从 AP 中得到代表 DT 的线程索引号。由于 DT 没有重载 createTrack_l，所以这个过程也会创建一个 Track 对象（和 MT 创建 Track 对象一样）。此时的结果，将导致图 7-16 变成图 7-17。

图 7-17 把 DT 的工作方式表达得非常清晰了。一个 DT 配合两个 OutputTrack 中的进程内缓冲，把来自 AT 的数据原封不动地发给蓝牙 MT 和 DSP MT，这简直就是个数据中继器。不过俗话说得好，道理虽简单，实现却复杂。来看 DT 是如何完成这一复杂而艰巨的任务的吧。

4.DT 的线程函数

DT 的线程函数代码如下所示：

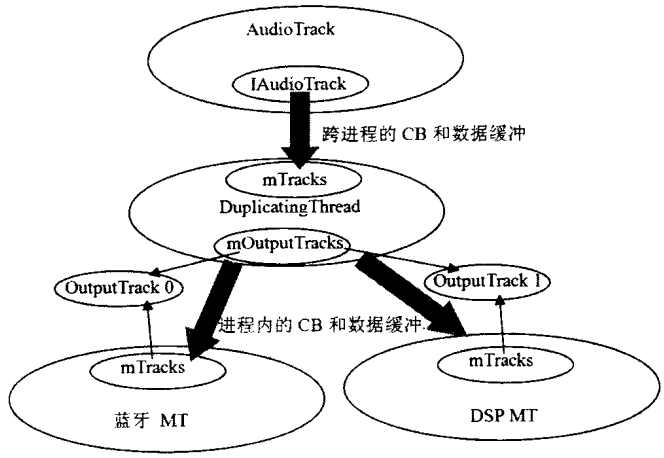


图 7-17　有 AT 的 DT 全景图

[-->AudioFlinger.cpp]

```
bool AudioFlinger::DuplicatingThread::threadLoop()
{
    int16_t* curBuf = mMixBuffer;
    Vector< sp<Track> > tracksToRemove;
    uint32_t mixerStatus = MIXER_IDLE;
    nsecs_t standbyTime = systemTime();
    size_t mixBufferSize = mFrameCount*mFrameSize;
    SortedVector< sp<OutputTrack> > outputTracks;
    while (!exitPending())
    {
        processConfigEvents();

        mixerStatus = MIXER_IDLE;
        {
            ......
            // 处理配置请求，和 MT 处理一样。
            const SortedVector< wp<Track> >& activeTracks = mActiveTracks;

            for (size_t i = 0; i < mOutputTracks.size(); i++) {
                outputTracks.add(mOutputTracks[i]);
            }
        // 如果 AT 的 Track 停止了，则需要停止和 MT 共享的 OutputTrack。
    if UNLIKELY((!activeTracks.size() && systemTime() > standbyTime)
                || mSuspended) {
                if (!mStandby) {
                    for (size_t i = 0; i < outputTracks.size(); i++) {
                        outputTracks[i]->stop();
                    }
                    mStandby = true;
                    mBytesWritten = 0;
```

```
                }
                ......
            //DT 从 MT 派生，天然具有混音的功能，所以这部分功能和 MT 一致。
            mixerStatus = prepareTracks_l(activeTracks, &tracksToRemove);
        }

        if (LIKELY(mixerStatus == MIXER_TRACKS_READY)) {
            //outputsReady 将检查 OutputTracks 对应的 MT 状态。
            if (outputsReady(outputTracks)) {
                mAudioMixer->process(curBuf);// 使用 AudioMixer 对象混音。
            } else {
                memset(curBuf, 0, mixBufferSize);
            }
            sleepTime = 0;
            writeFrames = mFrameCount;
        }
            ......
            if (sleepTime == 0) {
                standbyTime = systemTime() + kStandbyTimeInNsecs;
            for (size_t i = 0; i < outputTracks.size(); i++) {
                // 将混音后的数据写到 outputTrack 中。
                outputTracks[i]->write(curBuf, writeFrames);
            }
            mStandby = false;
            mBytesWritten += mixBufferSize;
        } else {
            usleep(sleepTime);
        }
        tracksToRemove.clear();
        outputTracks.clear();
    }

    return false;
}
```

现在，来自远端进程 AT 的数据已得到了混音，这一份混音后的数据还将通过调用 OutputTrack 的 write 完成 DT 到其他两个 MT 的传输。注意，这里除了 AT 使用的 Track 外，还有 DT 和两个 MT 共享的 OutputTrack。AT 调用的 start，将导致 DT 的 Track 加入到活跃数组中，但另外两个 OutputTrack 还没调用 start。这些操作又是在哪里做的呢？来看 write 函数，代码如下所示：

[-->AudioFlinger.cpp]

```
bool AudioFlinger::PlaybackThread::OutputTrack::write(int16_t* data,
                                                      uint32_t frames)
{
    // 注意，此处的 OutputTrack 是 DT 和 MT 共享的。
    Buffer *pInBuffer;
```

```
  Buffer inBuffer;
  uint32_t channels = mCblk->channels;
  bool outputBufferFull = false;
  inBuffer.frameCount = frames;
  inBuffer.i16 = data;

  uint32_t waitTimeLeftMs = mSourceThread->waitTimeMs();

if (!mActive && frames != 0) {
    // 如果此 Track 没有活跃，则调用 start 激活。
      start();
      ......
  }
   /*
     现在，AF 中的数据传递有三个线程：一个 DT，两个 MT。MT 作为 DT 的二级消费者，
     可能由于某种原因来不及消费数据，所以 DT 中提供了一个缓冲队列 mBufferQueue，
     把 MT 来不及消费的数据保存在这个缓冲队列中。注意这个缓冲队列容纳的临时缓冲
     个数是有限制的，其限制值由 kMaxOverFlowBuffers 控制，初始化为 10 个。
 */
  while (waitTimeLeftMs) {
      // 先消耗保存在缓冲队列的数据。
      if (mBufferQueue.size()) {
          pInBuffer = mBufferQueue.itemAt(0);
      } else {
          pInBuffer = &inBuffer;
      }
      ......
     // 获取可写缓冲，下面这句代码是否和 AT 中对应的代码很相似?
     if (obtainBuffer(&mOutBuffer, waitTimeLeftMs) ==
                            (status_t)AudioTrack::NO_MORE_BUFFERS) {
            ......
             break;
       }

      uint32_t outFrames = pInBuffer->frameCount > mOutBuffer.frameCount ?
                            mOutBuffer.frameCount : pInBuffer->frameCount;
      // 将数据拷贝到 DT 和 MT 共享的那块缓冲中去。
      memcpy(mOutBuffer.raw, pInBuffer->raw,
               outFrames * channels * sizeof(int16_t));
      // 更新写位置
      mCblk->stepUser(outFrames);
      pInBuffer->frameCount -= outFrames;
      pInBuffer->i16 += outFrames * channels;
      mOutBuffer.frameCount -= outFrames;
      mOutBuffer.i16 += outFrames * channels;
      ......
    }//while 结束

     if (inBuffer.frameCount) {
      sp<ThreadBase> thread = mThread.promote();
```

```
            if (thread != 0 && !thread->standby()) {
                if (mBufferQueue.size() < kMaxOverFlowBuffers) {
                    pInBuffer = new Buffer;
                    pInBuffer->mBuffer = new int16_t[inBuffer.frameCount * channels];
                    pInBuffer->frameCount = inBuffer.frameCount;
                    pInBuffer->i16 = pInBuffer->mBuffer;
                    //拷贝旧数据到新的临时缓冲。
                    memcpy(pInBuffer->raw, inBuffer.raw,
                            inBuffer.frameCount * channels * sizeof(int16_t));
                    //保存这个临时缓冲。
                    mBufferQueue.add(pInBuffer);
                }
            }
        }
        //如果数据全部写完
        if (pInBuffer->frameCount == 0) {
            if (mBufferQueue.size()) {
                mBufferQueue.removeAt(0);
                delete [] pInBuffer->mBuffer;
                delete pInBuffer;//释放缓冲队列对应的数据缓冲。
            } else {
                break;
            }
        }
    }
    ......
    return outputBufferFull;
}
```

数据就这样通过 DT 的帮助，从 AT 传输到蓝牙的 MT 和 DSP 的 MT 中了。这种方式的数据传输比直接使用 MT 传输要缓慢。

到这里，对 DT 的讲解就告一段落了。本人觉得，DT 的实现是 AF 代码中最美妙的地方，多学习这些优秀代码，有助于提高学习者的水平。

说明 DT 还有别的一些细节本书中没有涉及，读者可以结合自己的情况进行分析和理解。

7.5.2 题外话

下面说一点题外话，希望借此和读者一起探讨交流，共同进步。开源世界，开放平台，关键就是能做到兼容并包。

1. CTS 和单元测试

对驱动开发有所了解的人可能都会知道，芯片的类型太多了，它们的操作也不太一样。这必然会导致开发一个东西就得写一套代码，非常烦琐，并且可能是在反复创造低价值。

作为一个应用层开发者，我非常不希望的是，下层驱动的变动影响上层的应用（在工作

中经常会发现问题像病毒一样随意扩散）。当然，Android 已经提供了 HAL 层，任何硬件厂商都需要实现这些接口。硬件厂商的这些代码是需要编译成程序来进行验证的，可我不想拿应用层程序来做测试程序。因为应用层程序有自己的复杂逻辑，可能触发一个声音的 bug，需要满足很多预期的条件，否则会非常影响 HAL 的测试。有没有办法解决这一问题呢？像 Google 这样的公司，面临着很多硬件厂商，它又是怎么解决的呢？

我从 CTS 上看到了希望。CTS 是 Google 为 Android 搞的一个兼容性测试，即不管是怎么实现硬件驱动的，反正得通过我的 CTS 测试。当然，CTS 并不是用来测试硬件的，但是它的这种思想可以参考和借鉴。

我很羡慕 iOS 的应用开发者，他们面临的由于硬件变化导致的问题要少得多。

其实，如果做驱动移植的同仁们能以测试驱动开发的态度来严格测试，可能我们这些上层开发人员就不会总怀疑是驱动的问题了。

2. ALSA——Advanced Linux Sound Architecture

ALSA 是什么，大伙儿可以网上 google 之。现在在大力推广 ALSA，但在 Android 这块，我个人感觉它还不是很好用。“不好用”是从上层用户的角度说的，ALSA 提供了一个用户空间的 libasound 库，而这个库的确比较难用。不过有了 Audio HAL 的帮助，应用层就不用做改动了，但是实现 HAL 层的厂商要做的改动就比较大了。相比较而言，我觉得现在的源码中使用的 open/ioctl 方法更为方便。

说明　*这可能和我做过的应用太简单有一定关系（就是 ffmpeg 编解码 MP3，然后播出来即可）。而 libasound 提供的 API 较多，在权衡各种情况后，我觉得它不适合快速简单的应用开发。*

---

3. Desktop Check

Desktop Check 虽然是一种行为，但我更觉得它的产生是基于了一种态度。Desktop Check 本意是桌面检查。起因是在计算机技术刚兴起时，程序员调试代码非常费劲，因为那时机器配置很差，调试工具也不像现在这么发达，有时要跑到机房，预约机器然后启动调试器，做这一切所需时间远远多于坐在电脑前修改一个 bug 的时间。对于这种情况怎么办？为什么不像考试那样对自己的代码多检查几遍呢？自己虚拟一些应用场景，结合参数代入程序，在大脑中 Trace 岂不更好？这正是 Desktop Check 行为的本意。

今天，很多开发人员不厌其烦得添加 log，然后运行看输出。当然，这是解决问题的一种比较好的办法，但是在时间充裕的情况下，我还是希望开发人员能像我们前辈那样，用 Desktop Check 的这种方式先反复阅读和检查程序，争取在大脑中模拟程序的运行，最后才用打印 log 的方法来验证自己的想法。

另外，Desktop check 对提升阅读代码的能力有重要帮助。

说明　*已不记得第一次接触 Desktop Check 一词是什么时候了，或许当时还不叫 Desktop Check，但我觉得它所蕴涵的思想是正确的，是颇有价值的。*

7.6 本章小结

Audio 是本书碰到的第一个复杂系统，这个系统的整体示意图如图 7-18 所示：

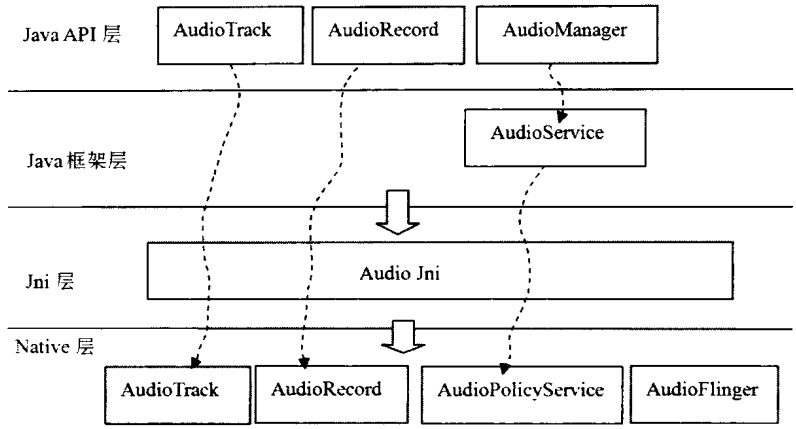


图 7-18 Audio 系统大家族

从图 7-18 中可以看出：

- 音频数据的输入输出不论是 Java 层还是 Native 层，都是通过 AudioTrack 和 AudioRecord 类完成的。事实上，Audio 系统提供的 I/O 接口就是 AudioTrack 和 AudioRecord 类。音频 I/O 是 Audio 系统最重要的部分。建议读者反复阅读，加深理解。
- AudioManager 用来做音量调节、audio 模式的选择、设备连接控制等。这些都会和 Native 的 AP 交互。从我个人博客和其他技术论坛的统计来看，较少有人关注 AudioPolicy，毕竟在这一块 Android 已提供了一个足够好用的 AudioPolicyManagerBase 类。不过作为 Audio 系统不可或缺的一部分，AudioPolicy 的重要性是不言而喻的。

建议 *无论怎么说，数据 I/O 毕竟是 Audio 系统中的关键之关键，所以请读者一定要仔细阅读，体会其中的精妙之所在。*

Audio 系统中还有其他部分（例如 AudioRecord、Java 层的 AudioSystem，AudioService 等），本书没有涉及，读者可结合个人需要自行分析。在现有的基础上，要学习并掌握这些内容都不会太难。



第8章 深入理解Surface系统

本章涉及的源代码文件名及位置：

- ActivityThread.java(*framework/base/core/java/android/app/ActivityThread.java*)
- Activity.java(*framework/base/core/java/android/app/Activity.java*)
- Instrumentation.java(*framework/base/core/java/android/app/Instrumentation.java*)
- PolicyManager.java(*frameworks/policies/base/phone/com/android/internal/policy/impl/PolicyManager.java*)
- Policy.java(*frameworks/policies/base/phone/com/android/internal/policy/impl/Policy.java*)
- PhoneWindow.java(*frameworks/policies/base/phone/com/android/internal/policy/impl/PhoneWindow.java*)
- Window.java(*framework/base/core/java/android/view/Window.java*)
- WindowManagerImpl(*framework/ base/core/java/android/view/WindowManagerImpl.java*)
- ViewRoot.java(*framework/base/core/java/android/view/ViewRoot.java*)
- Surface.java(*framework/base/core/java/android/view/Surface.java*)
- WindowManagerService.java(*framework/base/services/java/com/android/server/WindowManagerService.java*)
- IWindowSession.aidl(*framework/base/core/java/android/view/IWindowSession.aidl*)
- IWindow.aidl(*framework/base/core/java/android/view/IWindow.aidl*)
- SurfaceSession.java(*framework/base/core/java/android/view/SurfaceSession.java*)
- android_view_Surface.cpp(*framework/base/core/jni/android_view_Surface.cpp*)

- framebuffer_service.c(*system/core/adb/framebuffer_service.c*)
- SurfaceComposerClient.cpp(*framework/base/libs/surfaceflinger_client/SurfaceComposerClient.cpp*)
- SurfaceFlinger.cpp(*framework/base/libs/surfaceflinger/SurfaceFlinger.cpp*)
- ISurfaceComposer.h(*framework/base/include/surfaceflinger/ISurfaceComposer.h*)
- Layer.h(*framework/base/include/surfaceflinger/Layer.h*)
- Layer.cpp(*framework/base/libs/surfaceflinger/Layer.cpp*)
- LayerBase.cpp(*framework/base/libs/surfaceflinger/LayerBase.cpp*)
- Surface.cpp(*framework/base/libs/surfaceflinger_client/Surface.cpp*)
- SharedBufferStack.cpp(*framework/base/libs/surfaceflinger_client/SharedBufferStack.cpp*)
- GraphicBuffer.h(*framework/base/include/ui/GraphicBuffer.h*)
- GraphicBuffer.cpp(*framework/base/libs/ui/GraphicBuffer.cpp*)
- GraphicBufferAllocator.h(*framework/base/include/ui/GraphicBufferAllocator.h*)
- GraphicBufferAllocator.cpp(*framework/base/libs/ui/GraphicBufferAllocator.cpp*)
- GraphicBufferMapper.cpp(*framework/base/libs/ui/GraphicBufferMapper.cpp*)
- Android_natives.h(*framework/base/include/ui/egl/Android_natives.h*)
- android_native_buffer.h(*framework/base/include/ui/android_native_buffer.h*)
- native_handle.h(*system/core/include/cutils/native_handle.h*)
- gralloc.h(*hardware/libhardware/include/hardware/gralloc.h*)
- ISurface.cpp(*framework/base/libs/surfaceflinger_client/ISurface.cpp*)
- DisplayHardware.cpp(*framework/base/libs/surfaceflinger/DisplayHardware.cpp*)

8.1　概述

Surface 是继 Audio 系统后要破解第二个复杂的系统。它的难度和复杂度远远超过了 Audio。基于这种情况，本章将集中精力打通 Surface 系统的“任督二脉”，这任督二脉分别是：

- 任脉：应用程序和 Surface 的关系。
- 督脉：Surface 和 SurfaceFlinger 之间的关系。

当这二脉打通后，我们就可以自行修炼更高层次的功夫了。图 8-1 显示了这二脉的关系：

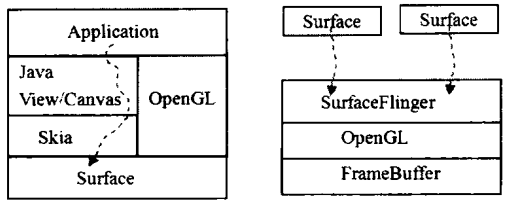


图 8-1　Surface 系统的任督二脉

其中，左图是任脉，右图是督脉。

- 先看左图。可以发现，不论是使用 Skia 绘制二维图像，还是用 OpenGL 绘制三维图像，最终 Application 都要和 Surface 交互。Surface 就像是 UI 的画布，而 App 则像是在 Surface 上作画。所以要想打通任脉，就需破解 App 和 Surface 之间的关系。
- 再看右图。Surface 和 SurfaceFlinger 的关系，很像 Audio 系统中 AudioTrack 和 AudioFlinger 的关系。Surface 向 SurfaceFlinger 提供数据，而 SurfaceFlinger 则混合数据。所谓打通督脉的关键，就是破解 Surface 和 SurfaceFlinger 之间的关系。

目标已清楚，让我们开始“运功”破解代码吧！

说明　*为了书写方便，后文将 SurfaceFlinger 简写为 SF。*

8.2　一个 Activity 的显示

一般来说，应用程序的外表是通过 Activity 来展示的。那么，Activity 是如何完成界面绘制工作的呢？根据前面所讲的知识可知，应用程序的显示和 Surface 有关，那么具体到 Activity 上，它和 Surface 又是什么关系呢？

本节就来讨论这些问题。首先从 Activity 的创建说起。

8.2.1　Activity 的创建

我们已经知道了 Activity 的生命周期，如 onCreate 表示创建、onDestroy 表示销毁等，但大家是否考虑过这样一个问题：

如果没有创建 Activity，那么 onCreate 和 onDestroy 就没有任何意义了，可这个 Activity 究竟是在哪里创建的呢？

第 4 章中的“zygote 分裂”一节已讲过，zygote 在响应请求后会 fork 一个子进程，这个子进程是 App 对应的进程，它的入口函数是 ActivityThread 类的 main 函数。ActivityThread 类中有一个 handleLaunchActivity 函数，它就是创建 Activity 的地方。一起来看这个函数，代码如下所示：

[-->ActivityThread.java]

```
private final void handleLaunchActivity(ActivityRecord r, Intent customIntent) {
        // ① performLaunchActivity 返回一个 Activity
        Activity a = performLaunchActivity(r, customIntent);

         if (a != null) {
             r.createdConfig = new Configuration(mConfiguration);
             Bundle oldState = r.state;
            // ②调用 handleResumeActivity
             handleResumeActivity(r.token, false, r.isForward);
    }
        ......
}
```

handleLaunchActivity 函数中列出了两个关键点（即①和②），下面对其分别进行介绍。

1. 创建 Activity

第一个关键函数 performLaunchActivity 返回一个 Activity，这个 Activity 就是 App 中的那个 Activity（仅考虑 App 中只有一个 Activity 的情况），它是怎么创建的呢？其代码如下所示：

[-->ActivityThread.java]

```
private final Activity performLaunchActivity(ActivityRecord r,
Intent customIntent) {

         ActivityInfo aInfo = r.activityInfo;
         ......// 完成一些准备工作。
        //Activity 定义在 Activity.java 中。
         Activity activity = null;
         try {
             java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
    /*
    mInstrumentation 为 Instrumentation 类型，源文件为 Instrumentation.java。
    它在 newActivity 函数中根据 Activity 的类名通过 Java 反射机制来创建对应的 Activity，
    这个函数比较复杂，待会我们再分析它。
    */
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
```

```
            r.intent.setExtrasClassLoader(cl);
            if (r.state != null) {
                r.state.setClassLoader(cl);
            }
        } catch (Exception e) {
            ......
        }

        try {
            Application app =
              r.packageInfo.makeApplication(false, mInstrumentation);

             if (activity != null) {
                // 在 Activity 中 getContext 函数返回的就是这个 ContextImpl 类型的对象。
                ContextImpl appContext = new ContextImpl();
                ......
               // 下面这个函数会调用 Activity 的 onCreate 函数。
                mInstrumentation.callActivityOnCreate(activity, r.state);
                ......
        return activity;
    }
```

好了，performLaunchActivity 函数的作用明白了吧？

- 根据类名以 Java 反射的方法创建一个 Activity。
- 调用 Activity 的 onCreate 函数，开始在 SDK 中大书特书 Activity 的生命周期。

那么，在 onCreate 函数中，我们一般会做什么呢？在这个函数中，和 UI 相关的重要工作就是调用 setContentView 来设置 UI 的外观。接下去，需要看 handleLaunchActivity 中的第二个关键函数 handleResumeActivity。

2. handleResumeActivity 分析

上面已创建好了一个 Activity，再来看 handleResumeActivity。它的代码如下所示：

[-->ActivityThread.java]

```
final void handleResumeActivity(IBinder token, boolean clearHide,
boolean isForward) {
boolean willBeVisible = !a.mStartedActivity;

if (r.window == null && !a.mFinished && willBeVisible) {
      r.window = r.activity.getWindow();
      // ①获得一个 View 对象。
      View decor = r.window.getDecorView();
      decor.setVisibility(View.INVISIBLE);
      // ②获得 ViewManager 对象。
      ViewManager wm = a.getWindowManager();
       ......
```

```
        //③把刚才的 decor 对象加入到 ViewManager 中。
        wm.addView(decor, l);
    }
        ......// 其他处理
}
```

上面有三个关键点（即①～③）。这些关键点似乎已经和 UI 部分（如 View、Window）有联系了。那么这些联系是在什么时候建立的呢？在分析上面代码中的三个关键点之前，请大家想想在前面的过程中，哪些地方会和 UI 挂上钩呢？

答案是就在 onCreate 函数中，Activity 一般都在这个函数中通过 setContentView 设置 UI 界面。

看来，必须先分析 setContentView，才能继续后面的征程了。

3. setContentView 分析

setContentView 有好几个同名函数，现在只看其中的一个就可以了。代码如下所示：

[-->Activity.java]

```
public void setContentView(View view) {
//getWindow 返回的是什么呢？一起来看看。
  getWindow().setContentView(view);
}

public Window getWindow() {
  return mWindow; //返回一个类型为 Window 的 mWindow，它是什么？
}
```

上面出现了两个和 UI 有关系的类：View 和 Window[①]。来看 SDK 文档是怎么描述这两个类的。这里先给出原文描述，然后进行对应的解释：

- Window：abstract base class for a top-level window look and behavior policy. An instance of this class should be used as the top-level view added to the window manager. It provides standard UI policies such as a background, title area, default key processing, etc.

中文的意思是：Window 是一个抽象基类，用于控制顶层窗口的外观和行为。作为顶层窗口它有什么特殊的职能呢？即绘制背景和标题栏、默认的按键处理等。

这里面有一句比较关键的话：它将作为一个顶层的 view 加入到 Window Manager 中。

- View：This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling.

① 说实话，我刚接触Android UI的时候也有点分不清楚View和Window的区别。

View 的概念就比较简单了，它是一个基本的 UI 单元，占据屏幕的一块矩形区域，可用于绘制，并能处理事件。

根据上面对 View 和 Window 的描述，再加上 setContentView 的代码，我们可以想象一下这三者的关系，如图 8-2 所示：

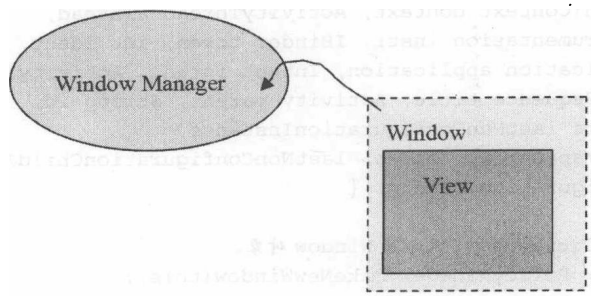


图 8-2　Window/View 的假想关系图

看了上面的介绍，大家可能会产生两个疑问：

- Window 是一个抽象类，它实际的对象到底是什么类型？
- Window Manager 究竟是什么？

如果能有这样的疑问，就说明我们非常细心了。下面来试着解决这两个问题。

（1）Activity 的 Window

据上文的讲解可知，Window 是一个抽象类。它实际的对象到底属于什么类型？先回到 Activity 创建的地方去看看。下面正是创建 Activity 时的代码，可当时没有深入地分析。

```
activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
```

代码中调用了 Instrumentation 的 newActivity，再去那里看看，如下所示：

[-->Instrumentation.java]

```
public Activity newActivity(Class<?> clazz, Context context,
            IBinder token, Application application, Intent intent,
            ActivityInfo info, CharSequence title, Activity parent,
String id,Object lastNonConfigurationInstance)
throws InstantiationException, IllegalAccessException {

Activity activity = (Activity)clazz.newInstance();
        ActivityThread aThread = null;
        // 关键函数 attach!!
        activity.attach(context, aThread, this, token, application, intent,
        info, title,parent, id, lastNonConfigurationInstance,
        new Configuration());
        return activity;
    }
```

看到关键函数 attach 了吧？ Window 的真相马上就要揭晓了，让我们用咆哮体[①]来表达内心的激动之情吧！！！！

[-->Activity.java]

```
final void attach(Context context, ActivityThread aThread,
            Instrumentation instr, IBinder token, int ident,
            Application application, Intent intent, ActivityInfo info,
            CharSequence title, Activity parent, String id,
            Object lastNonConfigurationInstance,
            HashMap<String,Object> lastNonConfigurationChildInstances,
            Configuration config) {
        ......
        // 利用 PolicyManager 来创建 Window 对象。
        mWindow = PolicyManager.makeNewWindow(this);
        mWindow.setCallback(this);
        ......
        // 创建 WindowManager 对象。
        mWindow.setWindowManager(null, mToken, mComponent.flattenToString());
        if (mParent != null) {
            mWindow.setContainer(mParent.getWindow());
        }
       // 保存这个 WindowManager 对象。
        mWindowManager = mWindow.getWindowManager();
        mCurrentConfig = config;
}
```

此刻又有一点失望吧？这里冒出了个 PolicyManager 类，Window 是由它的 makeNewWindow 函数所创建的，因此还必须再去看看这个 PolicyManager。

（2）水面下的冰山——PolicyManager

PolicyManager 定义于 PolicyManager.java 文件，该文件在一个非常独立的目录下，现将其单独列出来：

frameworks/policies/base/phone/com/android/internal/policy/impl

注意 上面路径中的灰色目录 phone 是针对智能手机这种小屏幕的；另外还有一个平级的目录叫 mid，是针对 Mid 设备的。mid 目录的代码比较少，可能目前还没有开发完毕。

下面来看这个 PolicyManager，它比较简单，代码如下所示：

[-->PolicyManager.java]

```
public final class PolicyManager {
    private static final String POLICY_IMPL_CLASS_NAME =
        "com.android.internal.policy.impl.Policy";
```

① 近期网络中流行的一种文体，其特点就是会用很多感叹号。

```
    private static final IPolicy sPolicy;

    static {
        //
        try {
            Class policyClass = Class.forName(POLICY_IMPL_CLASS_NAME);
            // 创建 Policy 对象。
            sPolicy = (IPolicy)policyClass.newInstance();
        } catch (ClassNotFoundException ex) {
            ......
        }

     private PolicyManager() {}

    // 通过 Policy 对象的 makeNewWindow 创建一个 Window。
    public static Window makeNewWindow(Context context) {
        return sPolicy.makeNewWindow(context);
    }
    ......
}
```

这里有一个单例的 sPolicy 对象，它是 Policy 类型，请看它的定义。

（3）真正的 Window

Policy 类型的定义代码如下所示：

[-->Policy.java]

```
public class Policy implements IPolicy {
    private static final String TAG = "PhonePolicy";

    private static final String[] preload_classes = {
        "com.android.internal.policy.impl.PhoneLayoutInflater",
        "com.android.internal.policy.impl.PhoneWindow",
        "com.android.internal.policy.impl.PhoneWindow$1",
        "com.android.internal.policy.impl.PhoneWindow$ContextMenuCallback",
        "com.android.internal.policy.impl.PhoneWindow$DecorView",
        "com.android.internal.policy.impl.PhoneWindow$PanelFeatureState",
"com.android.internal.policy.impl.PhoneWindow$PanelFeatureState$SavedState",
    };

    static {
        // 加载所有的类。
        for (String s : preload_classes) {
            try {
                Class.forName(s);
            } catch (ClassNotFoundException ex) {
                ......
            }
        }
```

```
    }

public PhoneWindow makeNewWindow(Context context) {
        //makeNewWindow 返回的是 PhoneWindow 对象。
        return new PhoneWindow(context);
    }

    ......
}
```

至此，终于知道了代码：

```
mWindow = PolicyManager.makeNewWindow(this);
```

返回的 Window，原来是一个 PhoneWindow 对象。它的定义在 PhoneWindow.java 中。mWindow 的真实身份搞清楚了，还剩下个 WindowManager。现在就来揭示其真面目。

（4）真正的 WindowManager

先看 WindowManager 创建的代码，如下所示：

[-->Activity.java]

```
    ......// 创建 mWindow 对象。
    // 调用 mWindow 的 setWindowManager 函数。
    mWindow.setWindowManager(null, mToken, mComponent.flattenToString());
    .....
```

上面的函数设置了 PhoneWindow 的 WindowManager，不过第一个参数是 null，这是什么意思？在回答此问题之前，先来看 PhoneWindow 的定义，它是从 Window 类派生的。

[-->PhoneWindow.java::PhoneWindow 定义]

```
public class PhoneWindow extends Window implements MenuBuilder.Callback
```

前面调用的 setWindowManager 函数，其实是由 PhoneWindow 的父类 Window 类来实现的，来看其代码，如下所示：

[-->Window.java]

```
public void setWindowManager(WindowManager wm,IBinder appToken, String appName)
{       // 注意，传入的 wm 值为 null。
        mAppToken = appToken;
        mAppName = appName;
        if (wm == null) {
            // 如果 wm 为空的话，则创建 WindowManagerImpl 对象。
            wm = WindowManagerImpl.getDefault();
        }
        //mWindowManager 是一个 LocalWindowManager。
        mWindowManager = new LocalWindowManager(wm);
    }
```

LocalWindowManager 是在 Window 中定义的内部类，请看它的构造函数，其定义如下所示：

[-->Window.java::LocalWindowManager 定义]

```
private class LocalWindowManager implements WindowManager {
        LocalWindowManager(WindowManager wm) {
            mWindowManager = wm;// 还好，只是简单地保存了传入的 wm 参数。
            mDefaultDisplay = mContext.getResources().getDefaultDisplay(
                    mWindowManager.getDefaultDisplay());
        }
    ......
```

如上面代码所示，LocalWindowManager 将保存一个 WindowManager 类型的对象，这个对象的实际类型是 WindowManagerImpl。而 WindowManagerImpl 又是什么呢？来看它的代码，如下所示：

[-->WindowManagerImpl.java]

```
public class WindowManagerImpl implements WindowManager {
......

public static WindowManagerImpl getDefault()
{
     return mWindowManager; //返回的就是 WindowManagerImpl 对象。
}
private static WindowManagerImpl mWindowManager = new WindowManagerImpl();
}
```

看到这里是否有点头晕眼花？很多朋友读了我所写的与此相关的一篇博客后，普遍也有此反应。所以，我试着配制了一剂治晕药方，如图 8-3 所示：

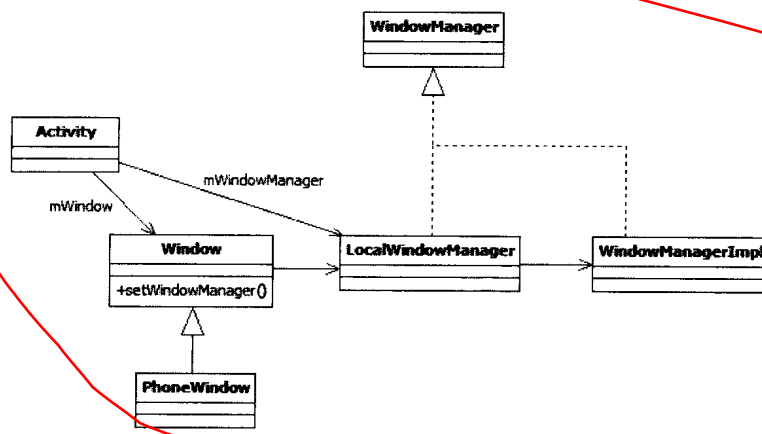


图 8-3　Window 和 WindowManager 的家族图谱

根据上图可得出以下结论：

- Activity 的 mWindow 成员变量其真实类型是 PhoneWindow，而 mWindowManager 成员变量的真实类型是 LocalWindowManager。
- LocalWindowManager 和 WindowManagerImpl 都实现了 WindowManager 接口。这里采用的是 Proxy 模式，表明 LocalWindowManager 将把它的工作委托给 WindowManagerImpl 来完成。

（5）关于 setContentView 的总结

了解了上述知识后，重新回到 setContentView 函数。这次希望能分析得更深入些。代码如下所示：

[-->Activity.java]

```
public void setContentView(View view) {
        getWindow().setContentView(view);//getWindow 返回的是 PhoneWindow。
}
```

一起来看 PhoneWindow 的 setContentView 函数，代码如下所示：

[-->PhoneWindow]

```
public void setContentView(View view) {
   // 调用另一个 setContentView。
   setContentView(view,
   new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));
}

public void setContentView(View view, ViewGroup.LayoutParams params) {
   //mContentParent 为 ViewGroup 类型，它的初值为 null。
     if (mContentParent == null) {
           installDecor();
     } else {
           mContentParent.removeAllViews();
     }
    // 把 view 加入到 ViewGroup 中。
     mContentParent.addView(view, params);
     ......
}
```

mContentParent 是一个 ViewGroup 类型，它从 View 中派生，所以也是一个 UI 单元。从它名字“Group”中所表达的意思来看，它应该还可以包含其他的 View 元素。这又是什么意思呢？

也就是说，在绘制一个 ViewGroup 时，它不仅需要把自己的样子画出来，还需要把它包含的 View 元素的样子也画出来。读者可将它想象成一个容器，容器中的元素就是 View。

注意 这里采用的是 23 种设计模式中的 Composite 模式，它是 UI 编程中常用的模式之一。

再来看 installDecor 函数，其代码如下所示：

[-->PhoneWindow.java]

```
private void installDecor() {
    if (mDecor == null) {
     // 创建 mDecor，它为 DecorView 类型，从 FrameLayout 派生。
     mDecor = generateDecor();
            ......
   }
  if (mContentParent == null) {
     // 得到这个 mContentParent。
mContentParent = generateLayout(mDecor);
// 创建标题栏。
mTitleView = (TextView)findViewById(com.android.internal.R.id.title);
......
}
```

generateLayout 函数的输入参数为 mDecor，输出为 mContentParent，代码如下所示：

[-->PhoneWindow]

```
protected ViewGroup generateLayout(DecorView decor){
  ......
  int layoutResource;
  int features = getLocalFeatures();
  if ((features & ((1 << FEATURE_LEFT_ICON) |(1 << FEATURE_RIGHT_ICON))) != 0) {
      if (mIsFloating) {
      // 根据情况取得对应标题栏的资源 id。
      layoutResource =  com.android.internal.R.layout.dialog_title_icons;
     }
        ......
}

  mDecor.startChanging();

 View in = mLayoutInflater.inflate(layoutResource, null);
 // 加入标题栏
 decor.addView(in, new ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));
   /*
    ID_ANDROID_CONTENT 的值为" com.android.internal.R.id.content",
    这个 contentParent 由 findViewById 返回，实际上就是 mDecorView 的一部分。
   */
   ViewGroup contentParent = (ViewGroup)findViewById(ID_ANDROID_CONTENT);
   ......
   mDecor.finishChanging();
   return contentParent;
}
```

下面看 findViewById 是如何实现的。它定义在 Window.java 中，代码如下所示：

[-->Window.java]

```
public View findViewById(int id) {
    //getDecorView 将返回 mDecorView，所以 contentParent 确实是 DecorView 的一部分。
    return getDecorView().findViewById(id);
 }
```

大家还记得图 8-2 吗？介绍完上面的知识后，可在图 8-2 的基础上绘制出更细致的图 8-4：

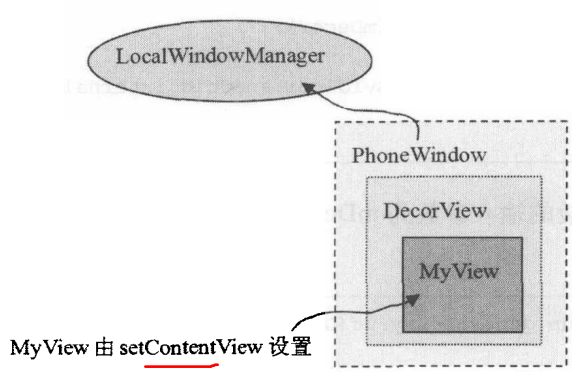


图 8-4 一个 Activity 中的 UI 组件

从上图中可看出，在 Activity 的 onCreate 函数中，通过 setContentView 设置的 View，其实只是 DecorView 的子 View。DecorView 还处理了标题栏显示等一系列的工作。

注意 这里使用了设计模式中的 Decorator（装饰）模式，它也是 UI 编程中常用的模式之一。

4. 重回 handleResumeActivity

看完 setContentView 的分析后，不知大家是否还记得我们为什么要分析这个 setContentView 函数？在继续前行之前，先来回顾一下被 setContentView 打断的流程。

当时，我们正在分析 handleResumeActivity，代码如下所示：

[-->ActivityThread.java]

```
final void handleResumeActivity(IBinder token, boolean clearHide,
boolean isForward) {
 boolean willBeVisible = !a.mStartedActivity;
......
if (r.window == null && !a.mFinished && willBeVisible) {
     r.window = r.activity.getWindow();
    //①获得一个 View 对象。现在知道这个 view 就是 DecorView 了。
   View decor = r.window.getDecorView();
```

```
    decor.setVisibility(View.INVISIBLE);
    //②获得 ViewManager 对象，这个 wm 就是 LocalWindowManager。
    ViewManager wm = a.getWindowManager();
    WindowManager.LayoutParams l = r.window.getAttributes();
    a.mDecor = decor;
    l.type = WindowManager.LayoutParams.TYPE_BASE_APPLICATION;
    if (a.mVisibleFromClient) {
        a.mWindowAdded = true;
        //③把刚才的 decor 对象加入到 ViewManager 中。
       wm.addView(decor, l);
    }
.....// 其他处理
}
```

在上面的代码中，由于出现了多个之前不熟悉的东西，如 View、ViewManager 等，而这些东西的来源又和 setContentView 有关，所以我们才转而去分析 setContentView 了。想起来了吧？

注意　由于代码比较长，跳转关系也很多，在分析代码时，请读者把握流程，在大脑中建立一个代码分析的堆栈。

下面就从 addView 的分析开始。如前面所介绍的，它的调用方法是：

```
wm.addView(decor, l);//wm 类型实际是 LocalWindowManager。
```

来看这个 addView 函数，它的代码如下所示：

[-->Window.java LocalWindowManager]

```
public final void addView(View view, ViewGroup.LayoutParams params) {

 WindowManager.LayoutParams wp = (WindowManager.LayoutParams)params;
 CharSequence curTitle = wp.getTitle();
 ..... //做一些操作，可以不管它。
// 还记得前面提到过的 Proxy 模式吗？ mWindowManager 对象实际上是 WindowManagerImpl 类型。
mWindowManager.addView(view, params);
}
```

看来，要搞清楚这个 addView 函数还是比较麻烦的，因此现在必须到 WindowManagerImpl 中去看看。它的代码如下所示：

[-->WindowManagerImpl.java]

```
private void addView(View view, ViewGroup.LayoutParams params, boolean nest)
{
  ViewRoot root; //ViewRoot，幕后的主角终于登场了！
  synchronized (this) {
  //①创建 ViewRoot
```

```
    root = new ViewRoot(view.getContext());
    root.mAddNesting = 1;
    view.setLayoutParams(wparams);

    if (mViews == null) {
        index = 1;
        mViews = new View[1];
        mRoots = new ViewRoot[1];
        mParams = new WindowManager.LayoutParams[1];
     } else {
        ......
      }
     index--;
     mViews[index] = view;
     mRoots[index] = root;// 保存这个 root
     mParams[index] = wparams;

  // ② setView，其中 view 是刚才我们介绍的 DecorView。
    root.setView(view, wparams, panelParentView);//
  }
```

“ViewRoot，ViewRoot……”，主角终于出场了！即使没介绍它的真实身份，不禁也想欢呼几声。为了避免高兴得过早，还是先冷静地分析一下它吧！代码中列出了 ViewRoot 的两个重要关键点（即①和②）。

（1）ViewRoot 是什么？

ViewRoot 是什么？看起来好像和 View 有些许关系，至少名字非常像。事实上，它的确和 View 有关系，因为它实现了 ViewParent 接口。SDK 的文档中有关于 ViewParent 的介绍。但它和 Android 基本绘图单元中的 View 却不太一样，比如：ViewParent 不处理绘画，因为它没有 onDraw 函数。

如上所述，ViewParent 和绘画没有关系，那么，它的作用是什么？先来看它的代码，如下所示：

[-->ViewRoot.java::ViewRoot 定义]

```
public final class ViewRoot extends Handler implements ViewParent,
        View.AttachInfo.Callbacks // 从 Handler 类派生。
{
private final Surface mSurface = new Surface();// 这里创建了一个 Surface 对象。
final W mWindow;  // 这个是什么？
View mView;
}
```

上面这段代码传达出了一些重要信息：

- ViewRoot 继承了 Handler 类，看来它能处理消息。ViewRoot 果真重写了 handleMessage 函数。稍后再来看它。

- ViewRoot 有一个成员变量叫 mSurface，它是 Surface 类型。
- ViewRoot 还有一个 W 类型的 mWindow 和一个 View 类型的 mView 变量。

其中，W 是 ViewRoot 定义的一个静态内部类：

```
static class W extends IWindow.Stub
```

这个类将参与 Binder 的通信，以后再对此做讲解，先来介绍 Surface 类。

（2）Android 是神笔马良吗？

这里冒出来一个 Surface 类。它是什么？在回答此问题之前，先来考虑这样一个问题：

前文介绍的 View、DecorView 等都是 UI 单元，这些 UI 单元的绘画工作都在 onDraw 函数中完成。如果把 onDraw 想象成画图过程，那么画布是什么？

Android 肯定不是“马良”，它也没有那支可以在任何物体上作画的“神笔”，所以我们需要一块实实在在的画布，这块画布就是 Surface。SDK 文档对 Surface 类的说明是：Handle on to a raw buffer that is being managed by the screen compositor。这句话的意思是：

- 有一块 Raw buffer，至于是内存还是显存，不必管它。
- Surface 操作这块 Raw buffer。
- Screen compositor（其实就是 SurfaceFlinger）管理这块 Raw buffer。

Surface 和 SF、ViewRoot 有什么关系呢？相信聪明的你此时已经有些明白了，这里用图 8-5 描绘一下心中的想法：

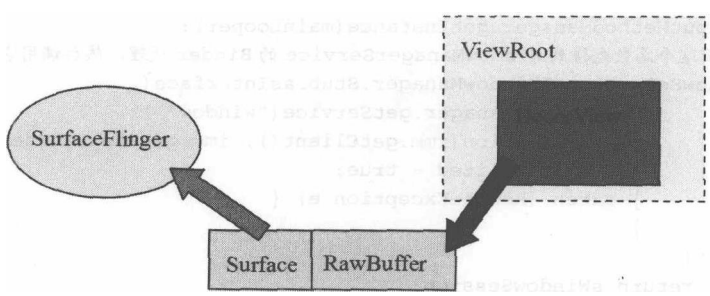


图 8-5　马良的神笔工作原理

结合之前所讲的知识来看图 8-5 可知其清晰地传达了如下几条信息：

- ViewRoot 有一个成员变量 mSurface，它是 Surface 类型，它和一块 Raw Buffer 有关联。
- ViewRoot 是一个 ViewParent，它的子 View 的绘画操作，是在画布 Surface 上展开的。
- Surface 和 SurfaceFlinger 有交互，这非常类似 AudioTrack 和 AudioFlinger 之间的交互。

既然本章题目为“深入理解 Surface 系统”，那么就需要重点关注 Surface 和 SurfaceFlinger 间的关系。建立这个关系需要 ViewRoot 的参与，所以应先来分析 ViewRoot 的创建和它的 setView 函数。

（3）ViewRoot 的创建和对 setView 的分析

来分析 ViewRoot 的构造。它所包含内容的代码如下所示：

[-->ViewRoot.java]

```
public ViewRoot(Context context) {
        super();
        ....
        // getWindowSession？我们进去看看。
        getWindowSession(context.getMainLooper());
       ......//ViewRoot 的 mWindow 是一个 W 类型，注意它不是 Window 类型，而是 IWindow 类型。
        mWindow = new W(this, context);
}
```

getWindowsession 函数，将建立 Activity 的 ViewRoot 和 WindowManagerService 的关系。代码如下所示：

[-->ViewRoot.java]

```
ublic static IWindowSession getWindowSession(Looper mainLooper) {
synchronized (mStaticInit) {
  if (!mInitialized) {
   try {
      InputMethodManager imm =
         InputMethodManager.getInstance(mainLooper);
      // 下面这个函数先得到 WindowManagerService 的 Binder 代理，然后调用它的 openSession。
      sWindowSession = IWindowManager.Stub.asInterface(
                  ServiceManager.getService("window"))
                 .openSession(imm.getClient(), imm.getInputContext());
                   mInitialized = true;
               } catch (RemoteException e) {
               }
          }
          return sWindowSession;
       }
    }
```

WindowSession？ WindowManagerService？第一次看到这些东西时，我快疯了。复杂，太复杂，无比复杂！要攻克这些难题，应先来回顾一下与 z ygote 相关的知识：

WindowManagerService（以后简称 WMS）由 System_Server 进程启动，SurfaceFlinger 服务也在这个进程中。

看来，Activity 的显示还不单纯是它自己的事，还需要和 WMS 建立联系才行。继续往下看，先看 setView 的处理。这个函数很复杂，注意其中关键的几句。

注意 openSession 的操作是一个使用 Binder 通信的跨进程调用，暂且记住这个函数，在精简流程之后再来分析。

代码如下所示：

[-->ViewRoot.java]

```
public void setView(View view, WindowManager.LayoutParams attrs,
                    View panelParentView) {// 第一个参数 view 是 DecorView。
        ......
        mView = view;// 保存这个 view。
         synchronized (this) {
             requestLayout(); // 待会先看看这个。
                 try {
                     // 调用 IWindowSession 的 add 函数，第一个参数是 mWindow。
                     res = sWindowSession.add(mWindow, mWindowAttributes,
                             getHostVisibility(), mAttachInfo.mContentInsets);
                 }
            ......
}
```

ViewRoot 的 setView 函数做了三件事：

- 保存传入的 view 参数为 mView，这个 mView 指向 PhoneWindow 的 DecorView。
- 调用 requestLayout。
- 调用 IWindowSession 的 add 函数，这是一个跨进程的 Binder 通信，第一个参数是 mWindow，它是 W 类型，从 IWindow.stub 派生。

先来看这个 requestLayout 函数，它非常简单，就是往 handler 中发送了一个消息。注意，ViewRoot 是从 handler 派生的，所以这个消息最后会由 ViewRoot 自己处理，代码如下所示：

[-->ViewRoot.java]

```
public void requestLayout() {
        checkThread();
        mLayoutRequested = true;
        scheduleTraversals();
}
public void scheduleTraversals() {
        if (!mTraversalScheduled) {
            mTraversalScheduled = true;
            sendEmptyMessage(DO_TRAVERSAL); // 发送 DO_TRAVERSAL 消息
        }
}
```

那么 requestLayout 就分析完毕了。

从上面的代码可发现，ViewRoot 和远端进程 SystemServer 的 WMS 有交互，先来总结一下它和 WMS 的交互流程：

- ViewRoot 调用 openSession，得到一个 IWindowSession 对象。

- 调用 WindowSession 对象的 add 函数，把一个 W 类型的 mWindow 对象作为参数传入。

5. ViewRoot 和 WMS 的关系

上面总结了 ViewRoot 和 WMS 的交互流程，其中一共有两个跨进程的调用，一起去看看。

（1）调用流程分析

WMS 的代码在 WindowManagerService.java 中，如下所示：

[-->WindowManagerService.java]

```
public IWindowSession openSession(IInputMethodClient client,
                                  IInputContext inputContext) {
        ......
return new Session(client, inputContext);
}
```

Session 是 WMS 定义的内部类。它支持 Binder 通信，并且属于 Bn 端，即响应请求的服务端。

再来看它的 add 函数。代码如下所示：

[-->WindowManagerService.java::Session]

```
public int add(IWindow window, WindowManager.LayoutParams attrs,
               int viewVisibility, Rect outContentInsets) {
    // 调用外部类对象的 addWindow，也就是 WMS 的 addWindow。
    return addWindow(this, window, attrs, viewVisibility,
                                   outContentInsets);
}
```

[-->WindowManagerService.java]

```
public int addWindow(Session session, IWindow client,
            WindowManager.LayoutParams attrs, int viewVisibility,
            Rect outContentInsets) {
            ......
            // 创建一个 WindowState。
            win = new WindowState(session, client, token,
                      attachedWindow, attrs, viewVisibility);
          ......
          // 调用 attach 函数。
           win.attach();
           ......
           return res;
}
```

WindowState 类也是在 WMS 中定义的内部类，直接看它的 attach 函数，代码如下所示：

[-->WMS.java::WindowState]

```
void attach() {
        //mSession 就是 Session 对象，调用它的 windowAddedLocked 函数。
        mSession.windowAddedLocked();
}
```

[-->WMS.java::Session]

```
void windowAddedLocked() {
  if (mSurfaceSession == null) {
          ......
         // 创建一个 SurfaceSession 对象。
         mSurfaceSession = new SurfaceSession();
         ......
      }
       mNumWindow++;
}
```

这里出现了另外一个重要的对象 SurfaceSession。在讲解它之前，急需理清一下现有的知识点，否则可能会头晕。

（2）ViewRoot 和 WMS 的关系梳理

ViewRoot 和 WMS 之间的关系，可用图 8-6 来表示：

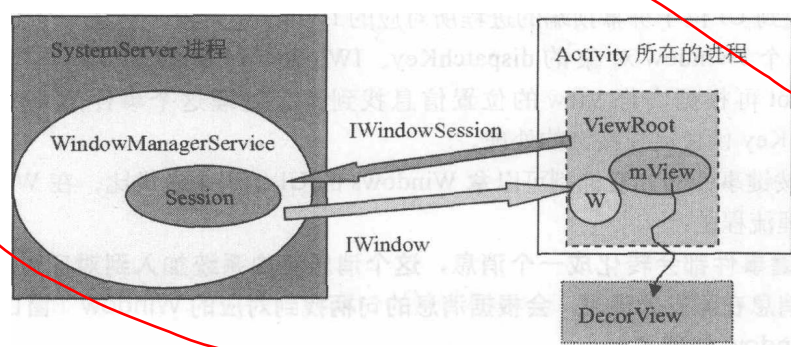


图 8-6　ViewRoot 和 WMS 的关系

总结一下图 8-6 中的知识点：

- ViewRoot 通过 IWindowSession 和 WMS 进程进行跨进程通信。IWindowSession 定义在 IWindowSession.aidl 文件中。这个文件在编译时由 aidl 工具处理，最后会生成类似于 Native Binder 中 Bn 端和 Bp 端的代码，后文会介绍它。
- ViewRoot 内部有一个 W 类型的对象，它也是一个基于 Binder 通信的类，W 是 IWindow 的 Bn 端，用于响应请求。IWindow 定义在另一个 aidl 文件 IWindow.aidl 中。

为什么需要这两个特殊的类呢？下面简单介绍一下。

首先，来看 IWindowSession.aidl 对自己的描述：

System private per-application interface to the window manager：也就是说每个 App 进程都会和 WMS 建立一个 IWindowSession 会话。这个会话被 App 进程用于和 WMS 通信。后面会介绍它的 requestLayout 函数。

再看对 IWindow.adil 的描述：

API back to a client window that the Window Manager uses to inform it of interesting things happening：这句话的大意是 IWindow 是 WMS 用来进行事件通知的。每当发生一些事情时，WMS 就会把这些事情告诉某个 IWindow。可以把 IWindow 想象成一个回调函数。

IWindow 的描述表达了什么意思呢？不妨看看它的内容，代码如下所示：

[-->IWindow.aidl 定义]

```
void dispatchKey(in KeyEvent event);
void dispatchPointer(in MotionEvent event, long eventTime,
boolean callWhenDone);
void dispatchTrackball(in MotionEvent event, long eventTime,
boolean callWhenDone);
```

明白了？这里的事件指的就是按键、触屏等事件。那么，一个按键事件是如何被分发的呢？下面是它大致的流程：

- WMS 所在的 SystemServer 进程接收到按键事件。
- WMS 找到 UI 位于屏幕顶端的进程所对应的 IWindow 对象，这是一个 Bp 端对象。
- 调用这个 IWindow 对象的 dispatchKey。IWindow 对象的 Bn 端位于 ViewRoot 中，ViewRoot 再根据内部 View 的位置信息找到真正处理这个事件的 View，最后调用 dispatchKey 函数完成按键的处理。

其实这些按键事件的分发机制可以拿 Windows 的 UI 编程来做类比，在 Windows 中应用程序的按键处理流程是：

每一个按键事件都会转化成一个消息，这个消息将由系统加入到对应进程的消息队列中。该进程的消息在派发处理时，会根据消息的句柄找到对应的 Window（窗口），继而该消息就由这个 Window 处理了。

注意 上面的描述实际上大大简化了真实的处理流程，读者可在大体了解相关知识后进行更深入的研究。

上面介绍的是 ViewRoot 和 WMS 的交互，但是我们最关心的 Surface 还没有正式介绍，不过在此之前，还是要先介绍 Activity 的流程。

8.2.2 Activity 的 UI 绘制

ViewRoot 的 setView 函数中会有一个 requestLayout。根据前面的分析可知，它会向

ViewRoot 发送一个 DO_TRAVERSAL 消息，来看它的 handleMessage 函数，代码如下所示：

[-->ViewRoot.java]

```
public void handleMessage(Message msg) {
        switch (msg.what) {
        ......
        case DO_TRAVERSAL:
            ......
            performTraversals();// 调用 performTraversals 函数。
            ......
            break;
       ......
    }
}
```

再去看 performTraversals 函数，这个函数比较复杂，先只看它的关键部分，代码如下所示：

[-->ViewRoot.java]

```
private void performTraversals() {
 final View host = mView;// 还记得这 mView 吗？它就是 DecorView 哦。

  boolean initialized = false;
  boolean contentInsetsChanged = false;
  boolean visibleInsetsChanged;
  try {
    relayoutResult = //①关键函数 relayoutWindow。
                relayoutWindow(params, viewVisibility, insetsPending);
   }
......
draw(fullRedrawNeeded);// ②开始绘制
......
}
```

1. relayoutWindow 分析

performTraversals 函数比较复杂，暂时只关注其中的两个函数 relayoutWindow 和 draw 即可。先看第一个 relayoutWindow，代码如下所示：

[-->ViewRoot.java]

```
private int relayoutWindow(WindowManager.LayoutParams params,
int viewVisibility, boolean insetsPending) throws RemoteException {

        // 原来是调用 IWindowSession 的 relayOut，暂且记住这个调用。
         int relayoutResult = sWindowSession.relayout(
                 mWindow, params,
```

```
                (int) (mView.mMeasuredWidth * appScale + 0.5f),
                (int) (mView.mMeasuredHeight * appScale + 0.5f),
                viewVisibility, insetsPending, mWinFrame,
                mPendingContentInsets, mPendingVisibleInsets,
                mPendingConfiguration, mSurface); mSurface 作为参数传进去了。
        }
    ......
}
```

relayoutWindow 中会调用 IWindowSession 的 relayout 函数，暂且记住这个调用，在精简流程后再进行分析。

2. draw 分析

再来看 draw 函数。这个函数非常重要，它可是 Activity 漂亮脸蛋的塑造大师啊，代码如下所示：

[-->ViewRoot.java]

```
private void draw(boolean fullRedrawNeeded) {
        Surface surface = mSurface;//mSurface 是 ViewRoot 的成员变量。
        ......
        Canvas canvas;
        try {
            int left = dirty.left;
            int top = dirty.top;
            int right = dirty.right;
            int bottom = dirty.bottom;
            // 从 mSurface 中 lock 一块 Canvas。
            canvas = surface.lockCanvas(dirty);
            ......
            mView.draw(canvas);// 调用 DecorView 的 draw 函数，canvas 就是画布的意思啦！
            ......
            //unlock 画布，屏幕上马上就会见到漂亮宝贝的长相了。
            surface.unlockCanvasAndPost(canvas);
        }
           ......
    }
```

UI 的显示好像很简单嘛！真的是这样的吗？在揭露这个“惊天秘密”之前我们先总结一下 Activity 的显示流程。

8.2.3 关于 Activity 的总结

不得不承认的是前面几节的内容很多也很繁杂，为了让后面分析的过程更流畅轻松一些，所以我们必须总结一下。关于 Activity 的创建和显示，前面几节的信息可提炼成如下几条：

- Activity 的顶层 View 是 DecorView，而我们在 onCreate 函数中通过 setContentView 设置的 View 只不过是这个 DecorView 中的一部分罢了。DecorView 是一个 FrameLayout 类型的 ViewGroup。
- Activity 和 UI 有关，它包含一个 Window（真实类型是 PhoneWindow）和一个 WindowManager（真实类型是 LocalWindowManager）对象。这两个对象将控制整个 Activity 的显示。
- LocalWindowManager 使用了 WindowManagerImpl 作为最终的处理对象（Proxy 模式），这个 WindowManagerImpl 中有一个 ViewRoot 对象。
- ViewRoot 实现了 ViewParent 接口，它有两个重要的成员变量，一个是 mView，它指向 Activity 顶层 UI 单元的 DecorView，另外一个是 mSurface，这个 Surface 包含了一个 Canvas（画布）。除此之外，ViewRoot 还通过 Binder 系统和 WindowManagerService 进行了跨进程交互。
- ViewRoot 能处理 Handler 的消息，Activity 的显示就是由 ViewRoot 在它的 performTraversals 函数中完成的。
- 整个 Activity 的绘图流程就是从 mSurface 中 lock 一块 Canvas，然后交给 mView 去自由发挥画画的才能，最后 unlockCanvasAndPost 释放这块 Canvas。

这里和显示有关的就是最后三条了，其中最重要的内容都和 Surface 相关，既然 mSurface 是 ViewRoot 的本地变量，那就直接去看 Surface。上面的代码分析似乎一路都比较流畅，波澜不惊，可事实果真如此吗？

8.3　初识 Surface

本节将介绍 Surface 对象。它可是纵跨 Java/JNI 层的对象，想必读者已经摩拳擦掌，跃跃欲试了。

8.3.1　和 Surface 有关的流程总结

这里先总结一下前面讲解中和 Surface 有关的流程：

- 在 ViewRoot 构造时，会创建一个 Surface，它使用无参构造函数，代码如下所示：

```
private final Surface mSurface = new Surface();
```

- ViewRoot 通过 IWindowSession 和 WMS 交互，而 WMS 中调用的一个 attach 函数会构造一个 SurfaceSession，代码如下所示：

```
void windowAddedLocked() {
    if (mSurfaceSession == null) {
        mSurfaceSession = new SurfaceSession();
        mNumWindow++;
    }
}
```

- ViewRoot 在 performTransval 的处理过程中会调用 IWindowSession 的 relayout 函数。这个函数目前还没有分析。
- ViewRoot 调用 Surface 的 lockCanvas，得到一块画布。
- ViewRoot 调用 Surface 的 unlockCanvasAndPost 释放这块画布。

下面从 relayout 函数开始分析。

8.3.2 Surface 之乾坤大挪移

1. 乾坤大挪移的表象

relayout 的函数是一个跨进程的调用，由 WMS 完成实际处理。先到 ViewRoot 中看看调用方的用法，代码如下所示：

[-->ViewRoot.java]

```
private int relayoutWindow(WindowManager.LayoutParams params,
int viewVisibility, boolean insetsPending)
throws RemoteException {
       int relayoutResult = sWindowSession.relayout(
                 mWindow, params,
                (int) (mView.mMeasuredWidth * appScale + 0.5f),
                (int) (mView.mMeasuredHeight * appScale + 0.5f),
               viewVisibility, insetsPending, mWinFrame,
               mPendingContentInsets, mPendingVisibleInsets,
               mPendingConfiguration, mSurface);//mSurface 传了进去。
        ......
        return relayoutResult;
    }
```

再看接收方的处理。它在 WMS 的 Session 中，代码如下所示：

[-->WindowManagerService.java::Session]

```
public int relayout(IWindow window, WindowManager.LayoutParams attrs,
                int requestedWidth, int requestedHeight, int viewFlags,
                boolean insetsPending, Rect outFrame, Rect outContentInsets,
                Rect outVisibleInsets, Configuration outConfig,
                Surface outSurface) {
                // 注意最后这个参数的名字，叫 outSurface。

   // 调用外部类对象的 relayoutWindow。
   return relayoutWindow(this, window, attrs,
                    requestedWidth, requestedHeight, viewFlags, insetsPending,
                    outFrame, outContentInsets, outVisibleInsets, outConfig,
                    outSurface);
}
```

[-->WindowManagerService.java]

```
public int relayoutWindow(Session session, IWindow client,
            WindowManager.LayoutParams attrs, int requestedWidth,
            int requestedHeight, int viewVisibility, boolean insetsPending,
            Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,
            Configuration outConfig, Surface outSurface){
    .....
  try {
         //win 就是 WinState，这里将创建一个本地的 Surface 对象。
         Surface surface = win.createSurfaceLocked();
          if (surface != null) {
            // 先创建一个本地 surface，然后在 outSurface 的对象上调用 copyFrom。
            // 将本地 Surface 的信息拷贝到 outSurface 中，为什么要这么麻烦呢?
            outSurface.copyFrom(surface);
          }
  }
    ......
}
```

[-->WindowManagerService.java::WindowState]

```
Surface createSurfaceLocked() {
      ......
   try {
     //mSurfaceSession 就是在 Session 上创建的 SurfaceSession 对象。
     // 这里以它为参数，构造一个新的 Surface 对象。
         mSurface = new Surface(
                 mSession.mSurfaceSession, mSession.mPid,
                  mAttrs.getTitle().toString(),
                  0, w, h, mAttrs.format, flags);
      }
         Surface.openTransaction();// 打开一个事务处理。
         ......
         Surface.closeTransaction();// 关闭一个事务处理。关于事务处理以后再分析。
         ......
}
```

上面的代码段好像有点混乱。用图 8-7 来表示一下这个流程：

根据图 8-7 可知：

- WMS 中的 Surface 是“乾坤”中的“乾”，它的构造使用了带 SurfaceSession 参数的构造函数。
- ViewRoot 中的 Surface 是“乾坤”中的“坤”，它的构造使用了无参构造函数。
- copyFrom 就是挪移，它将乾中的 Surface 信息，拷贝到坤中的 Surface 即 outSurface 里。

要是觉得乾坤大挪移就是这两三下，未免就太小看它了。为了彻底揭示这期间的复杂过程，我们将使用必杀技——aidl 工具。

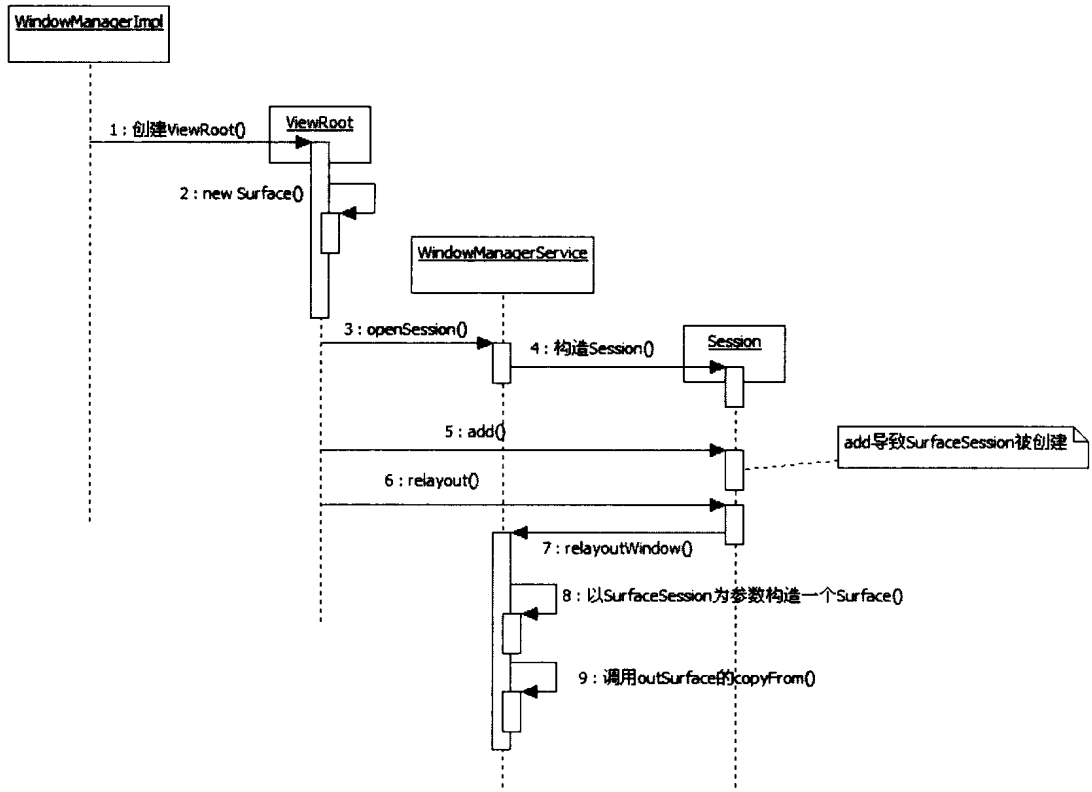


图 8-7　复杂的 Surface 创建流程

2. 揭秘 Surface 的乾坤大挪移

aidl 可以把 XXX.aidl 文件转换成对应的 Java 文件。刚才所说的乾坤大挪移发生在 ViewRoot 调用 IWindowSession 的 relayout 函数中，它在 IWindowSession.adil 中的定义如下：

[-->IWindowSesson.aidl]

```
interface IWindowSession {
    ......
 int relayout(IWindow window, in WindowManager.LayoutParams attrs,
            int requestedWidth, int requestedHeight, int viewVisibility,
            boolean insetsPending, out Rect outFrame, out Rect outContentInsets,
            out Rect outVisibleInsets, out Configuration outConfig,
            out Surface outSurface);
```

下面，拿必杀技 aidl 来编译一下这个 aidl 文件，使用方法如下：

在命令行下可以输入：

```
aidl -Ie:\froyo\source\frameworks\base\core\java\ -Ie:\froyo\source\frameworks\
```

```
base\Graphics\java  e:\froyo\source\frameworks\base\core\java\android\view\
IWindowSession.aidl test.java
```

新生成的 Java 文件叫 test.java。其中，-I 参数指定 include 目录，这是因为 aidl 文件中可能使用了别的 Java 文件中的类，所以需要指定这些 Java 文件所在的目录。

先看 ViewRoot 这个客户端生成的代码，如下所示：



[-->test.java::Bp 端 ::relayout]

```
public int relayout(android.view.IWindow window,
                    android.view.WindowManager.LayoutParams attrs,
                    int requestedWidth, int requestedHeight,
                    int viewVisibility, boolean insetsPending,
                    android.graphics.Rect outFrame,
                    android.graphics.Rect outContentInsets,
                    android.graphics.Rect outVisibleInsets,
                    android.content.res.Configuration outConfig,
                    android.view.Surface outSurface)//outSurface 是第 11 个参数。
                                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.writeStrongBinder((((window!=null))?(window.asBinder()):(null)));
    if ((attrs!=null)) {
      _data.writeInt(1);
      attrs.writeToParcel(_data, 0);
    }
   else {
    _data.writeInt(0);
   }
  _data.writeInt(requestedWidth);
  _data.writeInt(requestedHeight);
  _data.writeInt(viewVisibility);
  _data.writeInt(((insetsPending)?(1):(0)));
 // 奇怪，outSurface 的信息没有写到请求包 _data 中，就直接发送请求消息了。
  mRemote.transact(Stub.TRANSACTION_relayout, _data, _reply, 0);
  _reply.readException();
  _result = _reply.readInt();
  if ((0!=_reply.readInt())) {
   outFrame.readFromParcel(_reply);
  }
  ......
  if ((0!=_reply.readInt())) {
     outSurface.readFromParcel(_reply); // 从 Parcel 中读取信息来填充 outSurface。
    }
  }
  ......
```

```
    return _result;
}
```

奇怪！ ViewRoot 调用 requestlayout 竟然没有把 outSurface 信息传进去，这么说，服务端收到的 Surface 对象应该就是空吧？那怎么能调用 copyFrom 呢？还是来看服务端的处理，首先看收到消息的 onTransact 函数，代码如下所示：

[-->test.java::Bn 端 ::onTransact]

```
public boolean onTransact(int code, android.os.Parcel data,
                          android.os.Parcel reply, int flags)
                  throws android.os.RemoteException
{
  switch (code)
  {
    case TRANSACTION_relayout:
    {
      data.enforceInterface(DESCRIPTOR);
      android.view.IWindow _arg0;
      android.view.Surface _arg10;
      // 刚才讲了，Surface 信息并没有传过来，那么在 relayOut 中看到的 outSurface 是怎么
      // 出来的呢？看下面这句话便可知，原来在服务端这边竟然 new 了一个新的 Surface!!!
      _arg10 = new android.view.Surface();
      int _result = this.relayout(_arg0, _arg1, _arg2, _arg3, _arg4,
      _arg5, _arg6, _arg7, _arg8, _arg9, _arg10);
      reply.writeNoException();
      reply.writeInt(_result);
      //_arg10 就是调用 copyFrom 的那个 outSurface，那怎么传到客户端呢？
      if ((_arg10!=null)) {
            reply.writeInt(1);
            // 调用 Surface 的 writeToParcel，把信息写到 reply 包中。
            // 注意最后一个参数为 PARCELABLE_WRITE_RETURN_VALUE。
            _arg10.writeToParcel(reply,
                  android.os.Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        }
    }
    ......
   return true;
}
```

看完这个你是否会觉得大吃一惊？我最开始一直在 JNI 文件中寻找大挪移的踪迹，但是有几个关键点始终不能明白，万不得已就使用了这个 aidl 必杀技，终于揭露了真相。

3. 乾坤大挪移的真相

这里总结一下乾坤大挪移的整个过程，如图 8-8 表示：

上图非常清晰地列出了乾坤大挪移的过程，我们可结合代码来加深理解。

注意 这里将 BpWindowSession 作为了 IWindowSessionBinder 在客户端的代表。

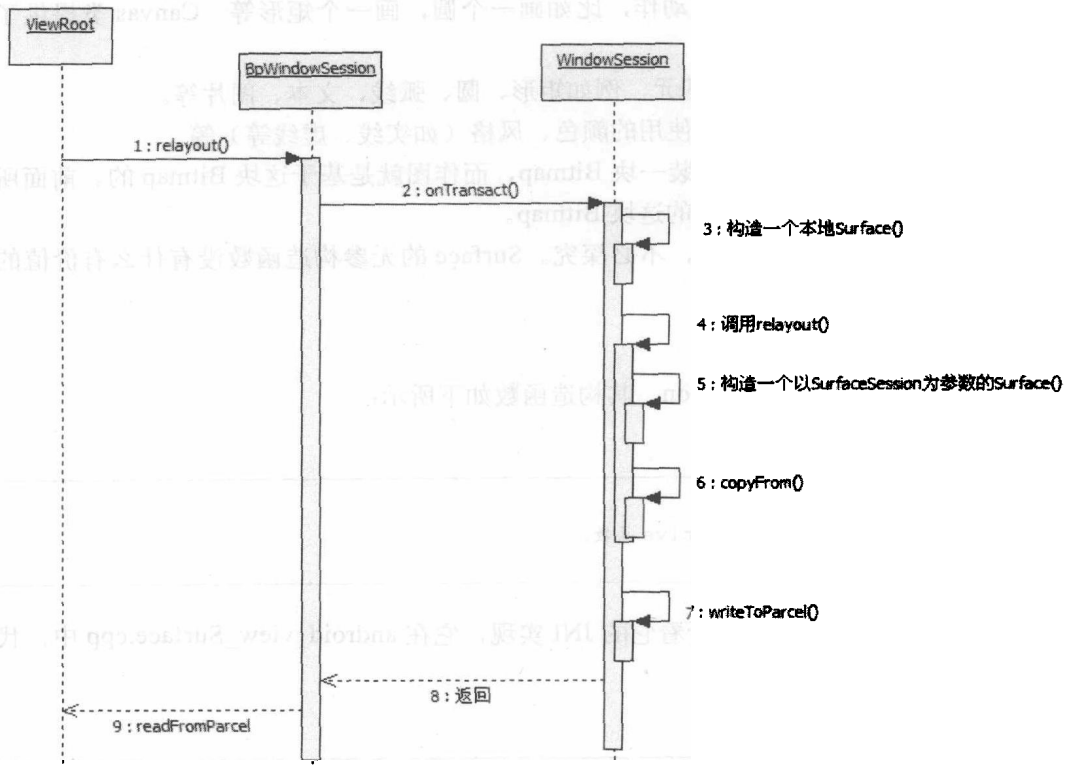


图 8-8　乾坤大挪移的真面目

8.3.3　乾坤大挪移的 JNI 层分析

前文讲述的内容都集中在 Java 层，按照流程顺序下面要分析 JNI 层的内容了。

1. Surface 的无参构造分析

在 JNI 层，第一个被调用的是 Surface 的无参构造函数，其代码如下所示：

[-->Surface.java]

```
public Surface() {
        ......
       //CompatibleCanvas 从 Canvas 类派生。
        mCanvas = new CompatibleCanvas();
    }
```

Canvas 是什么？根据 SDK 文档的介绍可知，画图需要“四大金刚”相互合作，这四大金刚是：

- Bitmap：用于存储像素，也就是画布。可把它当做一块数据存储区域。

- Canvas：用于记载画图的动作，比如画一个圆，画一个矩形等。Canvas 类提供了这些基本的绘图函数。
- Drawing primitive：绘图基元，例如矩形、圆、弧线、文本、图片等。
- Paint：它用来描述绘画时使用的颜色、风格（如实线、虚线等）等。

在一般情况下，Canvas 会封装一块 Bitmap，而作图就是基于这块 Bitmap 的。前面所说的画布，其实指的就是 Canvas 中的这块 Bitmap。

这些知识稍微了解一下即可，不必深究。Surface 的无参构造函数没有什么有价值的内容，接着看下面的内容。

2.SurfaceSession 的构造

现在要分析的是 SurfaceSession，其构造函数如下所示：

[-->SurfaceSession.java]

```
public SurfaceSession() {
        init();// 这是一个 native 函数。
}
```

init 是一个 native 函数。去看看它的 JNI 实现，它在 android_view_Surface.cpp 中，代码如下所示：

[-->android_view_Surface.cpp]

```
static void SurfaceSession_init(JNIEnv* env, jobject clazz)
{
     // 创建一个 SurfaceComposerClient 对象。
    sp<SurfaceComposerClient> client = new SurfaceComposerClient;
    client->incStrong(clazz);
    // 在 Java 对象中保存这个 client 对象的指针，类型为 SurfaceComposerClient。
    env->SetIntField(clazz, sso.client, (int)client.get());
}
```

这里先不讨论 SurfaceComposerClient 的内容，咱们还是继续把乾坤大挪移的流程走完。

3. Surface 的有参构造

下一个调用的是 Surface 的有参构造，其参数中有一个 SurfaceSession。先看 Java 层的代码，如下所示：

[-->Surface.java]

```
public Surface(SurfaceSession s,// 传入一个 SurfaceSession 对象。
        int pid, String name, int display, int w, int h, int format, int flags)
    throws OutOfResourcesException {
    ......
    mCanvas = new CompatibleCanvas();
  // 又一个 native 函数，注意传递的参数：display，以后再说。w,h 代表绘图区域的宽高值。
```

```
        init(s,pid,name,display,w,h,format,flags);
        mName = name;
    }
```

Surface 的 native init 函数的 JNI 实现，也在 android_view_Surface.cpp 中，一起来看看：

[-->android_view_Surface.cpp]

```
static void Surface_init(
        JNIEnv* env, jobject clazz,
        jobject session,
        jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
{
    // 从 SurfaceSession 对象中取出之前创建的那个 SurfaceComposerClient 对象。
    SurfaceComposerClient* client =
            (SurfaceComposerClient*)env->GetIntField(session, sso.client);

     sp<SurfaceControl> surface;// 注意它的类型是 SurfaceControl。
     if (jname == NULL) {
     /*
     调用 SurfaceComposerClient 的 createSurface 函数，返回的 surface 是一个
     SurfaceControl 类型。
     */
         surface = client->createSurface(pid, dpy, w, h, format, flags);
     } else {
         ......
}
    // 把这个 surfaceControl 对象设置到 Java 层的 Surface 对象中，对这个函数就不再分析了。
     setSurfaceControl(env, clazz, surface);
}
```

4. copyFrom 分析

现在要分析的就是 copyFrom 了。它是一个 native 函数。看它的 JNI 层代码：

[-->android_view_Surface.cpp]

```
static void Surface_copyFrom(JNIEnv* env, jobject clazz, jobject other)
{
    // 根据 JNI 函数的规则可知，clazz 是 copyFrom 的调用对象，而 other 是 copyFrom 的参数。
    // 目标对象此时还没有设置 SurfaceControl，而源对象在前面已经创建了 SurfaceControl。
    const sp<SurfaceControl>& surface = getSurfaceControl(env, clazz);
    const sp<SurfaceControl>& rhs = getSurfaceControl(env, other);
    if (!SurfaceControl::isSameSurface(surface, rhs)) {
        // 把源 SurfaceControl 对象设置到目标 Surface 中。
        setSurfaceControl(env, clazz, rhs);
     }
}
```

这一步还是比较简单的，下面看第 5 步 writeToParcel 函数的调用。

5.writeToParcel 分析

多亏了必杀技 aidl 工具的帮忙，才挖出这个隐藏的 writeToParcel 函数调用，下面就来看看它，代码如下所示：

[-->android_view_Surface.cpp]

```
static void Surface_writeToParcel(JNIEnv* env, jobject clazz,
jobject argParcel, jint flags)
{
    Parcel* parcel = (Parcel*)env->GetIntField(argParcel, no.native_parcel);
//clazz 就是 Surface 对象，从这个 Surface 对象中取出保存的 SurfaceControl 对象。
const sp<SurfaceControl>& control(getSurfaceControl(env, clazz));
/*
把 SurfaceControl 中的信息写到 Parcel 包中，然后利用 Binder 通信传递到对端，
对端通过 readFromParcel 来处理 Parcel 包。
*/
SurfaceControl::writeSurfaceToParcel(control, parcel);
if (flags & PARCELABLE_WRITE_RETURN_VALUE) {
       // 还记得 PARCELABLE_WRITE_RETURN_VALUE 吗？ flags 的值就等于它，
       // 所以本地 Surface 对象的 SurfaceControl 值被置空了。
        setSurfaceControl(env, clazz, 0);
    }
}
```

6.readFromParcel 分析

再看作为客户端的 ViewRoot 所调用的 readFromParcel 函数。它也是一个 native 函数，JNI 层的代码如下所示：

[-->android_view_Surface.cpp]

```
static void Surface_readFromParcel(
        JNIEnv* env, jobject clazz, jobject argParcel)
{
    Parcel* parcel = (Parcel*)env->GetIntField( argParcel, no.native_parcel);

   // 注意，下面定义的变量类型是 Surface，而不是 SurfaceControl。
   const sp<Surface>& control(getSurface(env, clazz));
   // 根据服务端传递的 Parcel 包来构造一个新的 surface。
    sp<Surface> rhs = new Surface(*parcel);
if (!Surface::isSameSurface(control, rhs)) {
    // 把这个新 surface 赋给 ViewRoot 中的 mSurface 对象。
    setSurface(env, clazz, rhs);
    }
}
```

7. Surface 乾坤大挪移的小结

可能有人会问，乾坤大挪移怎么这么复杂？这期间出现了多少对象？来总结一下，在此

期间一共有三个关键对象（注意我们这里只考虑 JNI 层的 Native 对象），它们分别是：

- SurfaceComposerClient。
- SurfaceControl。
- Surface，这个 Surface 对象属于 Native 层，和 Java 层的 Surface 相对应。

其中转移到 ViewRoot 成员变量 mSurface 中的，就是最后这个 Surface 对象了。这一路走来，真是异常坎坷，来回顾并概括总结一下这段历程。至于它的作用应该是很清楚了，以后要破解 SurfaceFlinger，靠的就是这个精简的流程。

- 创建一个 SurfaceComposerClient。
- 调用 SurfaceComposerClient 的 createSurface 得到一个 SurfaceControl 对象。
- 调用 SurfaceControl 的 writeToParcel 把一些信息写到 Parcel 包中。
- 根据 Parcel 包的信息构造一个 Surface 对象。把这个 Surface 对象保存到 Java 层的 mSurface 对象中。这样，大挪移的结果是 ViewRoot 得到一个 Native 的 Surface 对象。

注意　精简流程后，寥寥数语就可把过程说清楚。以后我们在研究代码时，也可以采取这种方式。

这个 Surface 对象非常重要，可它到底有什么用呢？这正是下一节要讲的内容。

8.3.4　Surface 和画图

下面来看最后两个和 Surface 相关的函数调用：一个是 lockCanvas；另外一个是 unlockCanvasAndPost。

1. lockCanvas 分析

要对 lockCanvas 进行分析，须先来看 Java 层的函数，代码如下所示：

[-->Surface.java::lockCanvas()]

```
public Canvas lockCanvas(Rect dirty)
throws OutOfResourcesException, IllegalArgumentException
 {
        return lockCanvasNative(dirty);// 调用 native 的 lockCanvasNative 函数。
}
```

[-->android_view_Surface.cpp::Surface_lockCanvas()]

```
static jobject Surface_lockCanvas(JNIEnv* env, jobject clazz, jobject dirtyRect)
{
    // 从 Java 中的 Surface 对象中，取出费尽千辛万苦得到的 Native 的 Surface 对象。
    const sp<Surface>& surface(getSurface(env, clazz));
    ......
```

```
    // dirtyRect 表示需要重绘的矩形块，下面根据这个 dirtyRect 设置 dirtyRegion。
    Region dirtyRegion;
    if (dirtyRect) {
        Rect dirty;
        dirty.left  = env->GetIntField(dirtyRect, ro.l);
        dirty.top   = env->GetIntField(dirtyRect, ro.t);
        dirty.right = env->GetIntField(dirtyRect, ro.r);
        dirty.bottom= env->GetIntField(dirtyRect, ro.b);
        if (!dirty.isEmpty()) {
            dirtyRegion.set(dirty);
        }
    } else {
        dirtyRegion.set(Rect(0x3FFF,0x3FFF));
    }

// 调用 Native Surface 对象的 lock 函数，
// 传入了一个参数 Surface::SurfaceInfo info 和一块表示脏区域的 dirtyRegion。
Surface::SurfaceInfo info;
status_t err = surface->lock(&info, &dirtyRegion);
......
//Java 的 Surface 对象构造的时候会创建一个 CompatibleCanvas。
// 这里就取出这个 CompatibleCanvas 对象。
jobject canvas = env->GetObjectField(clazz, so.canvas);
env->SetIntField(canvas, co.surfaceFormat, info.format);
// 从 Canvas 对象中取出 SkCanvas 对象。
SkCanvas* nativeCanvas = (SkCanvas*)env->GetIntField(
                canvas, no.native_canvas);
                SkBitmap bitmap;
                ssize_t bpr = info.s * bytesPerPixel(info.format);
    bitmap.setConfig(convertPixelFormat(info.format), info.w, info.h, bpr);
   ......
if (info.w > 0 && info.h > 0) {
//info.bits 指向一块存储区域。 .
        bitmap.setPixels(info.bits);
    } else {
        bitmap.setPixels(NULL);
}
// 给这个 SkCanvas 设置一个 Bitmap，还记得前面说的画图需要的四大金刚吗？
// 这里将 Bitmap 设置到这个 Canvas 中，这样进 UI 绘画时就有画布了。
nativeCanvas->setBitmapDevice(bitmap);
......

    return canvas;
}
```

lockCanvas 还算比较简单：

先获得一块存储区域，然后将它和 Canvas 绑定到一起，这样，UI 绘画的结果就记录在这块存储区域里了。

注意　本书不打算讨论 Android 系统上 Skia 和 OpenGL 方面的知识，有兴趣的读者可自行研究。

接下来看 unlockCanvasAndPost 函数，它也是一个 native 函数。

2. unlockCanvasAndPost 分析

来看 unlockCanvasAndPost 的代码，如下所示：

[-->android_view_Surface.cpp]

```
static void Surface_unlockCanvasAndPost(JNIEnv* env, jobject clazz,
jobject argCanvas)
{
    jobject canvas = env->GetObjectField(clazz, so.canvas);
    // 取出 Native 的 Surface 对象。
    const sp<Surface>& surface(getSurface(env, clazz));
    // 下面这些内容就讨论了，读者若有兴趣，可结合 Skia 库自行研究。
    SkCanvas* nativeCanvas = (SkCanvas*)env->GetIntField(canvas,
                                        no.native_canvas);
    int saveCount = env->GetIntField(clazz, so.saveCount);
    nativeCanvas->restoreToCount(saveCount);
    nativeCanvas->setBitmapDevice(SkBitmap());
    env->SetIntField(clazz, so.saveCount, 0);

    // 调用 Surface 对象的 unlockAndPost 函数。
    status_t err = surface->unlockAndPost();
    ......
}
```

unlockCanvasAndPost 也很简单，这里就不再多说了。

8.3.5　初识 Surface 小结

在本节的最后，我们来概括总结一下这一节所涉及的和 Surface 相关的调用流程，以备攻克下一个难关，如图 8-9 所示：

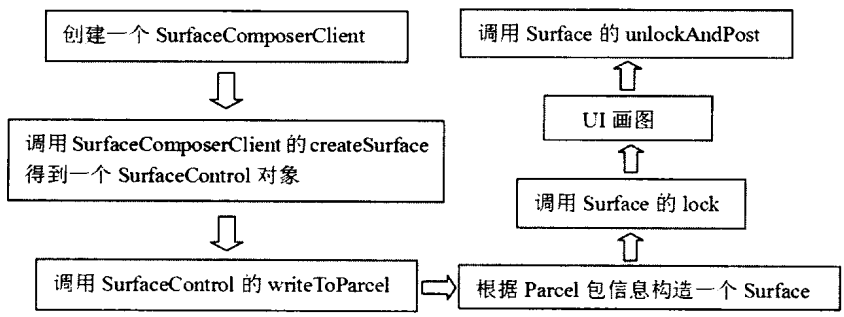


图 8-9　Surface 的精简流程图

8.4 深入分析 Surface

这一节将基于图 8-9 中的流程，对Surface 进行深入分析。在分析之前，还需要介绍一些 Android 平台上图形 / 图像显示方面的知识，这里统称为与 Surface 相关的基础知识。

8.4.1 与 Surface 相关的基础知识介绍

1. 显示层（Layer）和屏幕组成

你了解屏幕显示的漂亮界面是如何组织的吗？来看图 8-10 所展示的屏幕组成示意图。

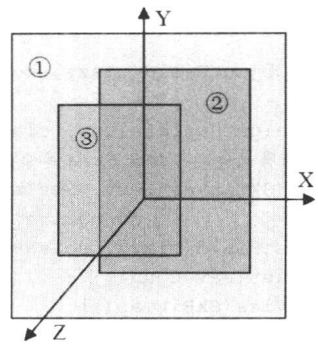


图 8-10 屏幕组成示意图

从图 8-10 中可以看出：

- 屏幕位于一个三维坐标系中，其中 Z 轴从屏幕内指向屏幕外。
- 编号为①②③的矩形块叫显示层（Layer）。每一层有自己的属性，例如颜色、透明度、所处屏幕的位置、宽、高等。除了属性之外，每一层还有自己对应的显示内容，也就是需要显示的图像。

在 Android 中，Surface 系统工作时，会由 SurfaceFlinger 对这些按照 Z 轴排好序的显示层进行图像混合，混合后的图像就是在屏幕上看到的美妙画面了。这种按 Z 轴排序的方式符合我们在日常生活中的体验，例如前面的物体会遮挡住后面的物体。

注意 *Surface 系统中定义了一个名为 Layer 类型的类，为了区分广义概念上的 Layer 和代码中的 Layer，这里称广义层的 Layer 为显示层，以免混淆。*

Surface 系统提供了三种属性，一共四种不同的显示层。简单介绍一下：

- 第一种属性是 eFXSurfaceNormal 属性，大多数的 UI 界面使用的就是这种属性。它有两种模式：

  1）Normal 模式，这种模式的数据，是通过前面的 mView.draw(canvas) 画上去的。这也是绝大多数 UI 所采用的方式。

  2）PushBuffer 模式，这种模式对应于视频播放、摄像机摄录 / 预览等应用场景。以摄

像机为例，当摄像机运行时，来自 Camera 的预览数据将直接 push 到 Buffer 中，无须应用层自己再去 draw 了。

- 第二种属性是 eFXSurfaceBlur 属性，这种属性的 UI 有点朦胧美，看起来很像隔着一层毛玻璃。
- 第三种属性是 eFXSurfaceDim 属性，这种属性的 UI 看起来有点暗，好像隔了一层深色玻璃。从视觉上讲，虽然它的 UI 看起来有点暗，但并不模糊。而 eFXSurfaceBlur 不仅暗，还有些模糊。

图 8-11 展示了最后两种类型的视觉效果图，其中左边的是 Blur 模式，右边的是 Dim 模式。

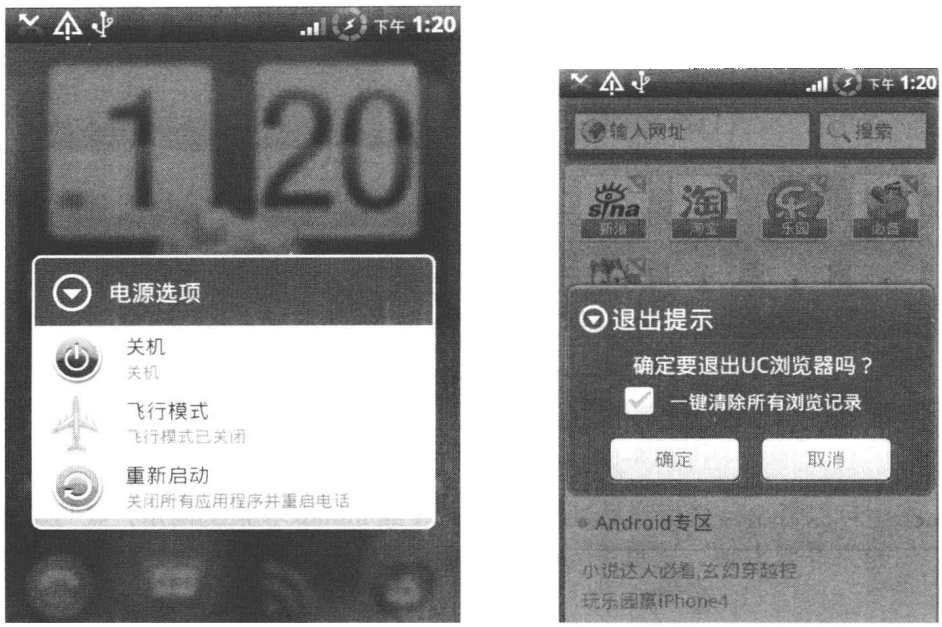


图 8-11　Blur 和 Dim 效果图

注意　关于 Surface 系统的显示层属性定义，读者可参考 ISurfaceComposer.h。

本章将重点分析第一种属性的两类显示层的工作原理。

2.FrameBuffer 和 PageFlipping

我们知道，在 Audio 系统中音频数据传输的过程是：

- 由客户端把数据写到共享内存中。
- 然后由 AudioFlinger 从共享内存中取出数据再往 Audio HAL 中发送。

根据以上介绍可知，在音频数据传输的过程中，共享内存起到了数据承载的重要作用。无独有偶，Surface 系统中的数据传输也存在同样的过程，但承载图像数据的是鼎鼎大名的

FrameBuffer（简称 FB）。下面先来介绍 FrameBuffer，然后再介绍 Surface 的数据传输过程。

（1）FrameBuffer 介绍

FrameBuffer 的中文名叫帧缓冲，它实际上包括两个不同的方面：

❑ Frame：帧，就是指一幅图像。在屏幕上看到的那幅图像就是一帧。

❑ Buffer：缓冲，就是一段存储区域，不过这个区域存储的是帧。

FrameBuffer 的概念很清晰，它就是一个存储图形 / 图像帧数据的缓冲。这个缓冲来自哪里？理解这个问题，需要简单介绍一下 Linux 平台的虚拟显示设备 FrameBuffer Device（简称 FBD）。FBD 是 Linux 系统中的一个虚拟设备，设备文件对应为 /dev/fb%d（比如 /dev/fb0）。这个虚拟设备将不同硬件厂商实现的真实设备统一在一个框架下，这样应用层就可以通过标准的接口进行图形 / 图像的输入和输出了。图 8-12 展示了 FBD 示意图：

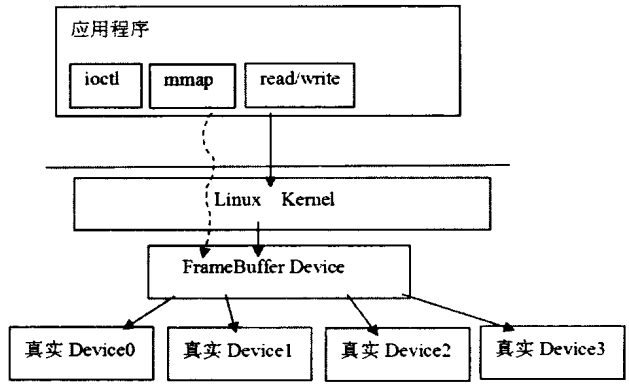


图 8-12 Linux 系统中的 FBD 示意图

从上图中可以看出，应用层通过标准的 ioctl 或 mmap 等系统调用，就可以操作显示设备了，用起来非常方便。这里把 mmap 的调用列出来，相信大部分读者都知道它的作用了。

FrameBuffer 中的 Buffer，就是通过 mmap 把设备中的显存映射到用户空间的，在这块缓冲上写数据，就相当于在屏幕上绘画。

注意 上面所说的框架将引出另外一个概念 Linux FrameBuffer（简称 LFB）。LFB 是 Linux 平台提供的一种可直接操作 FB 的机制，依托这个机制，应用层通过标准的系统调用，就可以操作显示设备了。从使用的角度来看，它和 Linux Audio 中的 OSS 有些类似。

为了加深读者对此节内容的理解，这里给出一个小例子，就是在 DDMS 工具中实现屏幕截图功能，其代码在 framebuffer_service.c 中，如下所示：

[-->framebuffer_service.c]

```
struct fbinfo {// 定义一个结构体。
    unsigned int version;
```

```
    unsigned int bpp;
    unsigned int size;
    unsigned int width;
    unsigned int height;
    unsigned int red_offset;
    unsigned int red_length;
    unsigned int blue_offset;
    unsigned int blue_length;
    unsigned int green_offset;
    unsigned int green_length;
    unsigned int alpha_offset;
    unsigned int alpha_length;
} __attribute__((packed));
//fd 是一个文件的描述符，这个函数的目的，是把当前屏幕的内容写到一个文件中。
void framebuffer_service(int fd, void *cookie)
{
    struct fb_var_screeninfo vinfo;
    int fb, offset;
    char x[256];

    struct fbinfo fbinfo;
    unsigned i, bytespp;
    //Android 系统上的 fb 设备路径在 /dev/graphics 目录下。
    fb = open("/dev/graphics/fb0", O_RDONLY);
    if(fb < 0) goto done;
    // 取出屏幕的属性。
    if(ioctl(fb, FBIOGET_VSCREENINFO, &vinfo) < 0) goto done;
    fcntl(fb, F_SETFD, FD_CLOEXEC);

    bytespp = vinfo.bits_per_pixel / 8;
    // 根据屏幕的属性填充 fbinfo 结构，这个结构要写到输出文件的头部。
    fbinfo.version = DDMS_RAWIMAGE_VERSION;
    fbinfo.bpp = vinfo.bits_per_pixel;
    fbinfo.size = vinfo.xres * vinfo.yres * bytespp;
    fbinfo.width = vinfo.xres;
    fbinfo.height = vinfo.yres;
/*
下面几个变量和颜色格式有关，以 RGB565 为例简单介绍一下。
RGB565 表示一个像素点中的 R 分量为 5 位，G 分量为 6 位，B 分量为 5 位，并且没有 Alpha 分量。
这样一个像素点的大小为 16 位，占两个字节，比 RGB888 格式的一个像素少一个字节（它一个像素是三个字节）。
x_length 的值为 x 分量的位数，例如，RGB565 中 R 分量就是 5 位。
x_offset 的值代表 x 分量在内存中的位置。如 RGB565 一个像素占两个字节，那么 x_offeset
表示 x 分量在这两个字节内存区域中的起始位置，但这个顺序是反的，也就是 B 分量在前，
R 在最后。所以 red_offset 的值就是 11，而 blue_offset 的值是 0，green_offset 的值是 6。
这些信息在做格式转换时（例如从 RGB565 转到 RGB888 的时候）有用。
*/
    fbinfo.red_offset = vinfo.red.offset;
    fbinfo.red_length = vinfo.red.length;
    fbinfo.green_offset = vinfo.green.offset;
    fbinfo.green_length = vinfo.green.length;
```

```
    fbinfo.blue_offset = vinfo.blue.offset;
    fbinfo.blue_length = vinfo.blue.length;
    fbinfo.alpha_offset = vinfo.transp.offset;
    fbinfo.alpha_length = vinfo.transp.length;

    offset = vinfo.xoffset * bytespp;

    offset += vinfo.xres * vinfo.yoffset * bytespp;
    // 将 fb 信息写到文件头部。
    if(writex(fd, &fbinfo, sizeof(fbinfo))) goto done;

    lseek(fb, offset, SEEK_SET);
    for(i = 0; i < fbinfo.size; i += 256) {
      if(readx(fb, &x, 256)) goto done;// 读取 FBD 中的数据。
      if(writex(fd, &x, 256)) goto done;// 将数据写到文件中。
    }

    if(readx(fb, &x, fbinfo.size % 256)) goto done;
    if(writex(fd, &x, fbinfo.size % 256)) goto done;

done:
    if(fb >= 0) close(fb);
    close(fd);
}
```

上面函数的目的就是截屏，这个例子可加深我们对 FB 的直观感受，相信读者下次再碰到 FB 时就不会犯怵了。

注意 我们可根据这段代码，写一个简单的 Native 可执行程序，然后 adb push 到设备上运行。注意上面写到文件中的是 RGB565 格式的原始数据，如想在台式机上看到这幅图片，可将它转换成 BMP 格式。我的个人博客上提供一个 RGB565 转 BMP 的程序，读者可以下载或自己另写一个，这样或许有助于更深入地理解图形 / 图像方面的知识。

在继续分析前，先来问一个问题：

前面在 Audio 系统中讲过，CB 对象通过读写指针来协调生产者 / 消费者的步调，那么 Surface 系统中的数据传输过程，是否也需通过读写指针来控制呢？

答案是肯定的，但不像 Audio 中的 CB 那样复杂。

（2）PageFlipping

图形 / 图像数据和音频数据不太一样，我们一般把音频数据叫音频流，它是没有边界的，而图形 / 图像数据是一帧一帧的，是有边界的。这一点非常类似 UDP 和 TCP 之间的区别。所以在图形 / 图像数据的生产 / 消费过程中，人们使用了一种叫 PageFlipping 的技术。

PageFlipping 的中文名叫画面交换，其操作过程如下所示：

- 分配一个能容纳两帧数据的缓冲，前面一个缓冲叫 FrontBuffer，后面一个缓冲叫 BackBuffer。

- 消费者使用 FrontBuffer 中的旧数据，而生产者用新数据填充 BackBuffer，二者互不干扰。
- 当需要更新显示时，BackBuffer 变成 FrontBuffer，FrontBuffer 变成 BackBuffer。如此循环，这样就总能显示最新的内容了。这个过程很像我们平常的翻书动作，所以它被形象地称为 PageFlipping。

说明　说白了，PageFlipping 其实就是使用了一个只有两个成员的帧缓冲队列，以后在分析数据传输的时候还会见到诸如 dequeue 和 queue 的操作。

3. 图像混合

我们知道，在 AudioFlinger 中有混音线程，它能将来自多个数据源的数据混合后输出，那么，SurfaceFlinger 是不是也具有同样的功能呢？

答案是肯定的，否则它就不会叫 Flinger 了。Surface 系统支持软硬两个层面的图像混合：

- 软件层面的混合：例如使用 copyBlt 进行源数据和目标数据的混合。
- 硬件层面的混合：使用 Overlay 系统提供的接口。

无论是硬件还是软件层面，都需将源数据和目标数据进行混合，混合需考虑很多内容，例如源的颜色和目标的颜色叠加后所产生的颜色。关于这方面的知识，读者可以学习计算机图形 / 图像学。这里只简单介绍一下 copyBlt 和 Overlay。

- copyBlt，从名字上看是数据拷贝，它也可以由硬件实现，例如现在很多的 2D 图形加速就是将 copyBlt 改由硬件来实现，以提高速度的。但不必关心这些，我们只需关心如何调用 copyBlt 相关的函数进行数据混合即可。
- Overlay 方法必须有硬件支持才可以，它主要用于视频的输出，例如视频播放、摄像机摄像等，因为视频的内容往往变化很快，所以如改用硬件进行混合效率会更高。

总体来说，Surface 是一个比较庞大的系统，由于篇幅和精力所限，本章后面的内容将重点关注 Surface 系统的框架和工作流程。在掌握框架和流程后，读者就可以在大的脉络中迅速定位到自己感兴趣的地方，然后展开更深入的研究了。

下面通过图 8-9 所示的精简流程，深入分析 Android 的 Surface 系统。

8.4.2　SurfaceComposerClient 分析

SurfaceComposerClient 的出现是因为：

Java 层 SurfaceSession 对象的构造函数会调用 Native 的 SurfaceSession_init 函数，而该函数的主要目的就是创建 SurfaceComposerClient。

先回顾一下 SurfaceSession_init 函数，代码如下所示：

[-->android_view_Surface.cpp]

```
static void SurfaceSession_init(JNIEnv* env, jobject clazz)
{
```

```
    //new 一个 SurfaceComposerClient 对象。
sp<SurfaceComposerClient> client = new SurfaceComposerClient;
//sp 的使用也有让人烦恼的地方，有时需要显式地增加强弱引用计数，要是忘记可就麻烦了。
client->incStrong(clazz);
 env->SetIntField(clazz, sso.client, (int)client.get());
}
```

上面代码中，显式地构造了一个 SurfaceComposerClient 对象。接下来看它是何方神圣。

1. 创建 SurfaceComposerClient

SurfaceComposerClient 这个名字隐含的意思是：

这个对象会和 SurfaceFlinger 进行交互，因为 SurfaceFlinger 派生于 SurfaceComposer。

通过它的构造函数来看是否是这样的。代码如下所示：

[-->SurfaceComposerClient.cpp]

```
SurfaceComposerClient::SurfaceComposerClient()
{
  //getComposerService() 将返回 SF 的 Binder 代理端的 BpSurfaceFlinger 对象。
  sp<ISurfaceComposer> sm(getComposerService());
// 先调用 SF 的 createConnection，再调用 _init。
   _init(sm, sm->createConnection());

    if (mClient != 0) {
        Mutex::Autolock _l(gLock);
       //gActiveConnections 是全局变量，把刚才创建的 client 保存到这个 map 中去。
        gActiveConnections.add(mClient->asBinder(), this);
    }
}
```

果然如此，SurfaceComposerClient 建立了和 SF 的交互通道，下面直接转到 SF 的 createConnection 函数去观察。

（1）createConnection 分析

直接看代码，如下所示：

[-->SurfaceFlinger.cpp]

```
sp<ISurfaceFlingerClient> SurfaceFlinger::createConnection()
{
    Mutex::Autolock _l(mStateLock);
    uint32_t token = mTokens.acquire();
   // 先创建一个 Client。
    sp<Client> client = new Client(token, this);
    // 把这个 Client 对象保存到 mClientsMap 中，token 是它的标识。
    status_t err = mClientsMap.add(token, client);
    /*
    创建一个用于 Binder 通信的 BClient，BClient 派生于 ISurfaceFlingerClient，
    它的作用是接受客户端的请求，然后把处理提交给 SF，注意，并不是提交给 Client。
```

```
    Client 会创建一块共享内存，该内存由 getControlBlockMemory 函数返回。
    */
    sp<BClient> bclient =
        new BClient(this, token, client->getControlBlockMemory());
    return bclient;
}
```

上面代码中提到，Client 会创建一块共享内存。熟悉 Audio 的读者或许会想到，这可能是 Surface 的 ControlBlock 对象！确实是的。CB 对象在协调生产 / 消费步调时，起到了决定性的控制作用，所以非常重要，下面来看：

[-->SurfaceFlinger.cpp]

```
Client::Client(ClientID clientID, const sp<SurfaceFlinger>& flinger)
    : ctrlblk(0), cid(clientID), mPid(0), mBitmap(0), mFlinger(flinger)
{
    const int pgsize = getpagesize();
    // 下面这个操作会使 cblksize 为页的大小，目前是 4096 字节。
    const int cblksize = ((sizeof(SharedClient)+(pgsize-1))&~(pgsize-1));
    //MemoryHeapBase 是我们的老朋友了，不熟悉的读者可以回顾 Audio 系统中所介绍的内容。
    mCblkHeap = new MemoryHeapBase(cblksize, 0,
                "SurfaceFlinger Client control-block");

    ctrlblk = static_cast<SharedClient *>(mCblkHeap->getBase());
    if (ctrlblk) {
        new(ctrlblk) SharedClient; // 再一次觉得眼熟吧？使用了 placement new。
    }
}
```

原来，Surface 的 CB 对象就是在共享内存中创建的这个 SharedClient 对象。先来认识一下这个 SharedClient。

（2）SharedClient 分析

SharedClient 定义了一些成员变量，代码如下所示：

```
class SharedClient
{
public:
    SharedClient();
    ~SharedClient();
    status_t validate(size_t token) const;
    uint32_t getIdentity(size_t token) const;// 取出标识本 Client 的 token。

private:
    Mutex lock;
Condition cv; // 支持跨进程的同步对象。
    //NUM_LAYERS_MAX 为 31，SharedBufferStack 是什么？
    SharedBufferStack surfaces[ NUM_LAYERS_MAX ];
};
//SharedClient 的构造函数，没什么新意，不如 Audio 的 CB 对象复杂。
SharedClient::SharedClient()
```

```
    : lock(Mutex::SHARED), cv(Condition::SHARED)
{
}
```

SharedClient 的定义似乎简单到极致了，不过不要高兴得过早，在这个 SharedClient 的定义中，没有发现和读写控制相关的变量，那怎么控制读写呢？

答案就在看起来很别扭的 SharedBufferStack 数组中，它有 31 个元素。关于它的作用就不必卖关子了，答案是：

一个 Client 最多支持 31 个显示层。每一个显示层的生产 / 消费步调都由会对应的 SharedBufferStack 来控制。而它内部就是用几个成员变量来控制读写位置的。

认识一下 SharedBufferStack 的这几个控制变量，如下所示：

[-->SharedBufferStack.h]

```
class  SharedBufferStack{
     ......
    //Buffer 是按块使用的，每个 Buffer 都有自己的编号，其实就是数组中的索引号。
    volatile int32_t head;      //FrontBuffer 的编号。
    volatile int32_t available; // 空闲 Buffer 的个数。
    volatile int32_t queued;  // 脏 Buffer 的个数，有脏 Buffer 表示有新数据的 Buffer。
    volatile int32_t inUse; //SF 当前正在使用的 Buffer 的编号。
    volatile status_t status; // 状态码
     ......
  }
```

注意，上面定义的 SharedBufferStack 是一个通用的控制结构，而不仅是针对于只有两个 Buffer 的情况。根据前面介绍的 PageFlipping 知识可知，如果只有两个 FB，那么，SharedBufferStack 的控制就比较简单了：

要么 SF 读 1 号 Buffer，客户端写 0 号 Buffer，要么 SF 读 0 号 Buffer，客户端写 1 号 Buffer。

图 8-13 是展示了 SharedClient 的示意图：

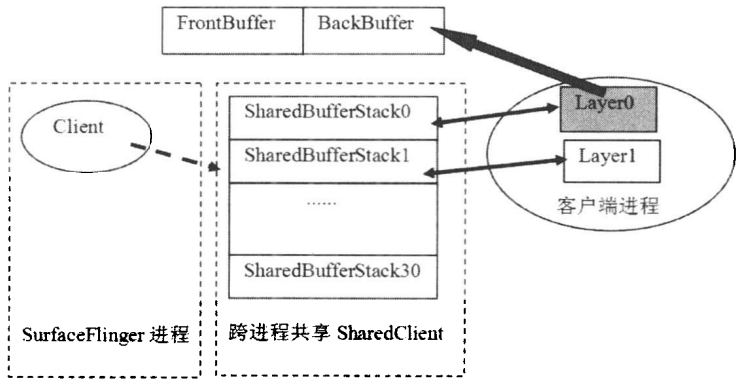


图 8-13　SharedClient 的示意图

从上图可知：

- SF 的一个 Client 分配一个跨进程共享的 SharedClient 对象。这个对象有 31 个 SharedBufferStack 元素，每一个 SharedBufferStack 对应于一个显示层。
- 一个显示层将创建两个 Buffer，后续的 PageFlipping 就是基于这两个 Buffer 展开的。

另外，每一个显示层中，其数据的生产和消费并不是直接使用 SharedClient 对象来进行具体控制的，而是基于 SharedBufferServer 和 SharedBufferClient 两个结构，由这两个结构来对该显示层使用的 SharedBufferStack 进行操作，这些内容在以后的分析中还会碰到。

注意　这里的显示层指的是 Normal 类型的显示层。

来接着分析后面的 _init 函数。

（3）_init 函数分析

先回顾一下之前的调用，代码如下所示：

[-->SurfaceComposerClient.cpp]

```
SurfaceComposerClient::SurfaceComposerClient()
{
    ......
    _init(sm, sm->createConnection());
    ......
}
```

来看这个 _init 函数，代码如下所示：

[-->SurfaceComposerClient.cpp]

```
void SurfaceComposerClient::_init(
        const sp<ISurfaceComposer>& sm, const sp<ISurfaceFlingerClient>& conn)
{
    mPrebuiltLayerState = 0;
    mTransactionOpen = 0;
    mStatus = NO_ERROR;
    mControl = 0;

    mClient = conn;//mClient 就是 BClient 的客户端
    mControlMemory = mClient->getControlBlock();
    mSignalServer = sm;// mSignalServer 就是 BpSurfaceFlinger。
    //mControl 就是那个创建于共享内存之中的 SharedClient。
    mControl = static_cast<SharedClient *>(mControlMemory->getBase());
}
```

_init 函数的作用，就是初始化 SurfaceComposerClient 中的一些成员变量。最重要的是得到了三个成员：

- mSignalServer，它其实是 SurfaceFlinger 在客户端的代理 BpSurfaceFlinger，它的

主要作用是，在客户端更新完 BackBuffer 后（也就是刷新了界面后），通知 SF 进行 PageFlipping 和输出等工作。

- mControl，它是跨进程共享的 SharedClient，是 Surface 系统的 ControlBlock 对象。
- mClient，它是 BClient 在客户端的对应物。

2. 到底有多少种对象？

这一节出现了好几种类型的对象，通过图 8-14 来看看它们：

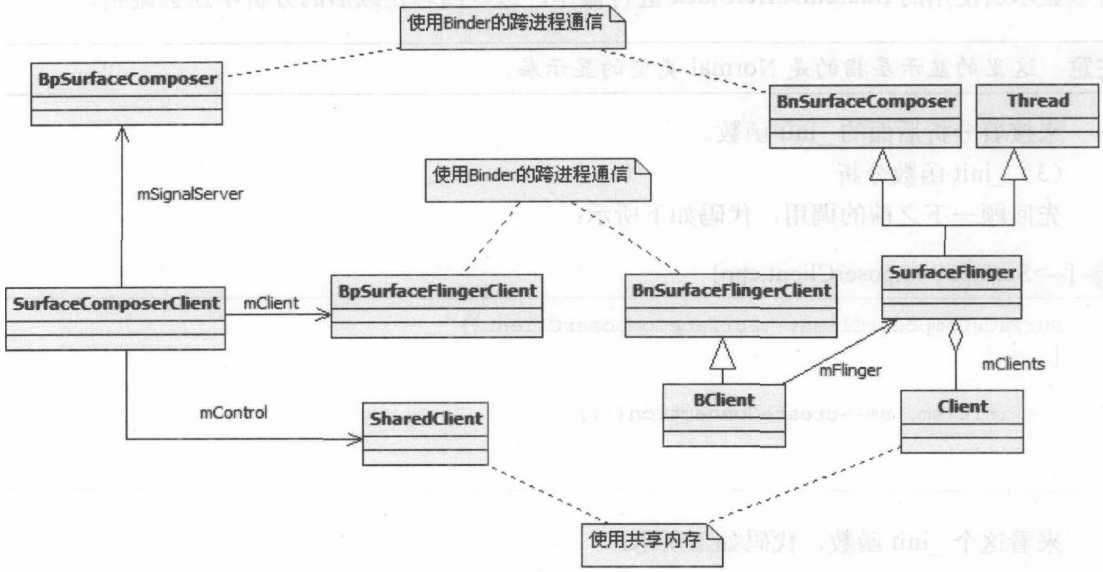


图 8-14　类之间关系的展示图

从上图中可以看出：

- SurfaceFlinger 是从 Thread 派生的，所以它会有一个单独运行的工作线程。
- BClient 和 SF 之间采用了 Proxy 模式，BClient 支持 Binder 通信，它接收客户端的请求，并派发给 SF 执行。
- SharedClient 构建于一块共享内存中，SurfaceComposerClient 和 Client 对象均持有这块共享内存。

在精简流程中，关于 SurfaceComposerClient 就分析到这里，下面分析第二个步骤中的 SurfaceControl 对象。

8.4.3　SurfaceControl 分析

1.SurfaceControl 的来历

根据精简的流程可知，这一节要分析的是 SurfaceControl 对象。先回顾一下这个对象的创建过程，代码如下所示：

[-->android_view_Surface.cpp]

```
static void Surface_init(JNIEnv* env, jobject clazz, jobject session,
        jint pid, jstring jname, jint dpy, jint w, jint h, jint format, jint flags)
{
     SurfaceComposerClient* client =
             (SurfaceComposerClient*)env->GetIntField(session, sso.client);
    //注意这个变量，类型是SurfaceControl，名字却叫surface，稍不留神就出错了。
    sp<SurfaceControl> surface;
   if (jname == NULL) {
    //调用Client的createSurface函数，得到一个SurfaceControl对象。
    surface = client->createSurface(pid, dpy, w, h, format, flags);
    }
    ......
   //将这个SurfaceControl对象设置到Java层的对象中保存。
   setSurfaceControl(env, clazz, surface);
}
```

通过上面的代码可知，SurfaceControl 对象由 createSurface 得来，下面看看这个函数。

注意　此时，读者或许会被代码中随意起的变量名搞糊涂，而我的处理方法是碰到了容易混淆的地方，尽量以对象类型来表示这个对象。

（1）createSurface 的请求端分析

在 createSurface 内部会使用 Binder 通信将请求发给 SF，所以它分为请求和响应两端，先看请求端，代码如下所示：

[-->SurfaceComposerClient.cpp]

```
sp<SurfaceControl> SurfaceComposerClient::createSurface(
        int pid,
        DisplayID display,//DisplayID是什么意思?
        uint32_t w,
        uint32_t h,
        PixelFormat format,
        uint32_t flags)
{
    String8 name;
    const size_t SIZE = 128;
    char buffer[SIZE];
    snprintf(buffer, SIZE, "<pid_%d>", getpid());
    name.append(buffer);
   //调用另外一个createSurface，多一个name参数。
    return SurfaceComposerClient::createSurface(pid, name, display,
            w, h, format, flags);

}
```

在分析另外一个 createSurface 之前，应先介绍一下 DisplayID 的含义：

```
typedef int32_t    DisplayID;
```

DisplayID 是一个 int 整型，它的意义是屏幕编号，例如双屏手机就有内屏和外屏两块屏幕。由于目前 Android 的 Surface 系统只支持一块屏幕，所以这个变量的取值都是 0。

再分析另外一个 createSurface 函数，它的代码如下所示：

[-->SurfaceComposerClient.cpp]

```
sp<SurfaceControl> SurfaceComposerClient::createSurface(
        int pid,const String8& name,DisplayID display,uint32_t w,
        uint32_t h,PixelFormat format,uint32_t flags)
{
    sp<SurfaceControl> result;
    if (mStatus == NO_ERROR) {
        ISurfaceFlingerClient::surface_data_t data;
        // 调用 BpSurfaceFlingerClient 的 createSurface 函数
        sp<ISurface> surface = mClient->createSurface(&data, pid, name,
                                    display, w, h, format, flags);
        if (surface != 0) {
            if (uint32_t(data.token) < NUM_LAYERS_MAX) {
                // 以返回的 ISurface 对象创建一个 SurfaceControl 对象。
                result = new SurfaceControl(this, surface, data, w, h,
                                            format, flags);
            }
        }
    }
    return result;// 返回的是 SurfaceControl 对象。
}
```

请求端的处理比较简单：

- 调用跨进程的 createSurface 函数，得到一个 ISurface 对象，根据第 6 章的 Binder 相关知识可知，这个对象的真实类型是 BpSurface。不过以后统称为 ISurface。
- 以这个 ISurface 对象为参数，构造一个 SurfaceControl 对象。

createSurface 函数的响应端在 SurfaceFlinger 进程中，下面去看这个函数。

注意 在 Surface 系统定义了很多类型，咱们也中途休息一下，不妨来看看和字符串“Surface”有关的类有多少个，权当小小的娱乐：

Native 层有 Surface、ISurface、SurfaceControl、SurfaceComposerClient。

Java 层有 Surface、SurfaceSession。

上面还只列出了一部分，后面还有呢！ *&@&*% ¥*

（2）createSurface 的响应端分析

前面讲过，可把 BClient 看作是 SF 的 Proxy，它会把来自客户端的请求派发给 SF 处理，

通过代码来看看是不是这样的。如下所示：

[-->SurfaceFlinger.cpp]

```
sp<ISurface> BClient::createSurface(
        ISurfaceFlingerClient::surface_data_t* params, int pid,
        const String8& name,
        DisplayID display, uint32_t w, uint32_t h, PixelFormat format,
        uint32_t flags)
{
  // 果然是交给 SF 处理，以后我们将跳过 BClient 这个代理。
return mFlinger->createSurface(mId, pid, name, params, display, w, h,
            format, flags);
}
```

来看 createSurface 函数，它的目的就是创建一个 ISurface 对象，不过这中间的玄机还挺多，代码如下所示：

[-->SurfaceFlinger.cpp]

```
sp<ISurface> SurfaceFlinger::createSurface(ClientID clientId, int pid,
        const String8& name, ISurfaceFlingerClient::surface_data_t* params,
        DisplayID d, uint32_t w, uint32_t h, PixelFormat format,
        uint32_t flags)
{
sp<LayerBaseClient> layer;//LayerBaseClient 是 Layer 家族的基类。
// 这里又留出一个 LayerBaseClient 的内部类，它也叫 Surface，是不是有点头晕了？
sp<LayerBaseClient::Surface> surfaceHandle;


Mutex::Autolock _l(mStateLock);
// 根据 clientId 找到 createConnection 时加入的那个 Client 对象。
sp<Client> client = mClientsMap.valueFor(clientId);
    ......
   // 注意这个 id，它的值表示 Client 创建的是第几个显示层，根据图 8-14 可以看出，这个 id

   // 同时也表示将使用 SharedBufferStatck 数组的第 id 个元素。
   int32_t id = client->generateId(pid);
    // 一个 Client 不能创建多于 NUM_LAYERS_MAX 个的 Layer。
    if (uint32_t(id) >= NUM_LAYERS_MAX) {
        return surfaceHandle;
    }
   // 根据 flags 参数来创建不同类型的显示层，我们在 8.4.1 节介绍过相关知识。
    switch (flags & eFXSurfaceMask) {
        case eFXSurfaceNormal:
            if (UNLIKELY(flags & ePushBuffers)) {
              // 创建 PushBuffer 类型的显示层，我们将在本章的拓展思考部分分析它。
             layer = createPushBuffersSurfaceLocked(client, d, id,
                        w, h, flags);
            } else {
```

```
                // ①创建 Normal 类型的显示层，我们待会儿分析这个。
                layer = createNormalSurfaceLocked(client, d, id,
                        w, h, flags, format);
            }
            break;
        case eFXSurfaceBlur:
            // 创建 Blur 类型的显示层。
            layer = createBlurSurfaceLocked(client, d, id, w, h, flags);
            break;
        case eFXSurfaceDim:
            // 创建 Dim 类型的显示层。
            layer = createDimSurfaceLocked(client, d, id, w, h, flags);
            break;
    }

    if (layer != 0) {
        layer->setName(name);
        setTransactionFlags(eTransactionNeeded);
        // 从显示层对象中取出一个 ISurface 对象赋值给 SurfaceHandle。
        surfaceHandle = layer->getSurface();
        if (surfaceHandle != 0) {
            params->token = surfaceHandle->getToken();
            params->identity = surfaceHandle->getIdentity();
            params->width = w;
            params->height = h;
            params->format = format;
        }
    }
    return surfaceHandle;//ISurface 的 Bn 端就是这个对象。
}
```

上面代码中的函数倒是很简单，只是代码里面冒出来的几个新类型和它们的名字让人有点头晕。先用文字总结一下：

- LayerBaseClient：前面提到的显示层在代码中的对应物就是这个 LayerBaseClient，不过这是一个大家族，不同类型的显示层将创建不同类型的 LayerBaseClient。
- LayerBaseClient 中有一个内部类，名字叫 Surface，这是一个支持 Binder 通信的类，它派生于 ISurface。

关于 Layer 的故事，后面会有单独的章节来介绍。这里先继续分析 createNormalSurfaceLocked 函数。它的代码如下所示：

[-->SurfaceFlinger.cpp]

```
sp<LayerBaseClient> SurfaceFlinger::createNormalSurfaceLocked(
        const sp<Client>& client, DisplayID display,
        int32_t id, uint32_t w, uint32_t h, uint32_t flags,
        PixelFormat& format)
{
```

```
    switch (format) { // 一些图像方面的参数设置，可以不去管它。
    case PIXEL_FORMAT_TRANSPARENT:
    case PIXEL_FORMAT_TRANSLUCENT:
        format = PIXEL_FORMAT_RGBA_8888;
        break;
    case PIXEL_FORMAT_OPAQUE:
        format = PIXEL_FORMAT_RGB_565;
        break;
    }
    // ①创建一个 Layer 类型的对象。
    sp<Layer> layer = new Layer(this, display, client, id);
    // ②设置 Buffer。
    status_t err = layer->setBuffers(w, h, format, flags);
    if (LIKELY(err == NO_ERROR)) {
         // 初始化这个新 layer 的一些状态。
        layer->initStates(w, h, flags);
       // ③ 还记得在图 8-10 中提到的 Z 轴吗？下面这个函数把这个 layer 加入到 Z 轴大军中。
        addLayer_l(layer);
}
......
    return layer;
}
```

createNormalSurfaceLocked 函数有三个关键点，它们是：

- 构造一个 Layer 对象。
- 调用 Layer 对象的 setBuffers 函数。
- 调用 SF 的 addLayer_l 函数。

暂且记住这三个关键点，后面有单独的章节分析它们。先继续分析 SurfaceControl 的流程。

（3）创建 SurfaceControl 对象

当跨进程的 createSurface 调用返回一个 ISurface 对象时，将通过下面的代码创建一个 SurfaceControl 对象：

```
result = new SurfaceControl(this, surface, data, w, h,format, flags);
```

下面来看这个 SurfaceControl 对象为何物。它的代码如下所示：

[-->SurfaceControl.cpp]

```
SurfaceControl::SurfaceControl(
        const sp<SurfaceComposerClient>& client,
        const sp<ISurface>& surface,
        const ISurfaceFlingerClient::surface_data_t& data,
        uint32_t w, uint32_t h, PixelFormat format, uint32_t flags)
    //mClient 为 SurfaceComposerClient，而 mSurface 指向跨进程 createSurface 调用
    // 返回的 ISurface 对象。
    : mClient(client), mSurface(surface),
```

```
        mToken(data.token), mIdentity(data.identity),
        mWidth(data.width), mHeight(data.height), mFormat(data.format),
        mFlags(flags)
{
}
```

SurfaceControl 类可以看作是一个 wrapper 类：

它封装了一些函数，通过这些函数可以方便地调用 mClient 或 ISurface 提供的函数。

在分析 SurfaceControl 的过程中，还遗留了和 Layer 相关的部分，下面就来解决它们。

2.Layer 和它的家族

我们在 createSurface 中创建的是 Normal 的 Layer，下面先看这个 Layer 的构造函数。

（1）Layer 的构造

Layer 是从 LayerBaseClient 派生的，其代码如下所示：

[-->Layer.cpp]

```
Layer::Layer(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& c, int32_t i)// 这个 i 表示 SharedBufferStack 数组的索引。
    :   LayerBaseClient(flinger, display, c, i),// 先调用基类构造函数。
        mSecure(false),
        mNoEGLImageForSwBuffers(false),
        mNeedsBlending(true),
        mNeedsDithering(false)
{
     //getFrontBuffer 实际取出的是 FrontBuffer 的位置。
    mFrontBufferIndex = lcblk->getFrontBuffer();
}
```

再来看基类 LayerBaseClient 的构造函数，代码如下所示：

[-->LayerBaseClient.cpp]

```
LayerBaseClient::LayerBaseClient(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& client, int32_t i)
    : LayerBase(flinger, display), lcblk(NULL), client(client), mIndex(i),
      mIdentity(uint32_t(android_atomic_inc(&sIdentity)))
{
   /*
    创建一个 SharedBufferServer 对象，注意它使用了 SharedClient 对象，
    并且传入了表示 SharedBufferStack 数组索引的 i 和一个常量 NUM_BUFFERS。
*/
lcblk = new SharedBufferServer(
            client->ctrlblk, i, NUM_BUFFERS,// 该值为常量 2，在 Layer.h 中定义
            mIdentity);
}
```

SharedBufferServer 是什么？它和 SharedClient 有什么关系？

其实，之前在介绍 SharedClient 时曾提过与此相关的内容，这里再来认识一下，先看图 8-15：

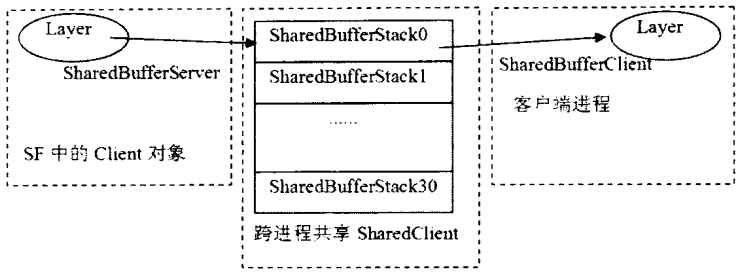


图 8-15　ShardBufferServer 的示意图

根据上图并结合前面的介绍，可以得出以下结论：

- 在 SF 进程中，Client 的一个 Layer 将使用 SharedBufferStack 数组中的一个成员，并通过 SharedBufferServer 结构来控制这个成员，我们知道 SF 是消费者，所以可由 SharedBufferServer 来控制数据的读取。
- 与之相对应的是，客户端的进程也会有一个对象来使用这个 SharedBufferStatck，可它是通过另外一个叫 SharedBufferClient 的结构来控制的。客户端为 SF 提供数据，所以可由 SharedBufferClient 控制数据的写入。在后文的分析中还会碰到 SharedBufferClient。

注意　*在本章的拓展思考部分，会有单独小节来分析生产 / 消费过程中的读写控制。*

通过前面的代码可知，Layer 对象被 new 出来后，传给了一个 sp 对象，读者还记得 sp 中的 onFirstRef 函数吗？Layer 家族在这个函数中还有一些处理。但这个函数是由基类 LayerBaseClient 实现的，一起去看看。

[-->LayerBase.cpp]

```
void LayerBaseClient::onFirstRef()
{
    sp<Client> client(this->client.promote());
if (client != 0) {
// 把自己加入到 client 对象的 mLayers 数组中，这部分内容比较简单，读者可以自行研究。
        client->bindLayer(this, mIndex);
    }
}
```

Layer 创建完毕，下面来看第二个重要的函数 setBuffers。

（2）setBuffers 分析

setBuffers、Layer 类以及 Layer 的基类都有实现。由于创建的是 Layer 类型的对

象，所以请读者直接到 Layer.cpp 中寻找 setBuffers 函数。这个函数的目的就是创建用于 PageFlipping 的 FrontBuffer 和 BackBuffer。一起来看，代码如下所示：

[-->Layer.cpp]

```
status_t Layer::setBuffers( uint32_t w, uint32_t h,
                            PixelFormat format, uint32_t flags)
{
    PixelFormatInfo info;
    status_t err = getPixelFormatInfo(format, &info);
    if (err) return err;

    //DisplayHardware 是代表显示设备的 HAL 对象，0 代表第一块屏幕的显示设备。
    // 这里将从 HAL 中取出一些和显示相关的信息。
    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    uint32_t const maxSurfaceDims = min(
            hw.getMaxTextureSize(), hw.getMaxViewportDims());

    PixelFormatInfo displayInfo;
    getPixelFormatInfo(hw.getFormat(), &displayInfo);
    const uint32_t hwFlags = hw.getFlags();

    ......

/*
创建 Buffer，这里将创建两个 GraphicBuffer。这两个 GraphicBuffer 就是我们前面
所说的 FrontBuffer 和 BackBuffer。
*/
for (size_t i=0 ; i<NUM_BUFFERS ; i++) {
    // 注意，这里调用的是 GraphicBuffer 的无参构造函数，mBuffers 是一个二元数组。
    mBuffers[i] = new GraphicBuffer();
}
    // 又冒出来一个 SurfaceLayer 类型，#￥%……&*！@
    mSurface = new SurfaceLayer(mFlinger, clientIndex(), this);
    return NO_ERROR;
}
```

setBuffers 函数的工作内容比较简单，就是：

- 创建一个 GraphicBuffer 缓冲数组，元素个数为 2，即 FrontBuffer 和 BackBuffer。
- 创建一个 SurfaceLayer，关于它的身世我们后面再介绍。

注意 GraphicBuffer 是 Android 提供的显示内存管理的类，关于它的故事将在 8.4.7 节中介绍。我们暂把它当作普通的 Buffer 即可。

setBuffers 中出现的 SurfaceLayer 类是什么？读者可能对此感觉有些晕乎。待把最后一个关键函数 addLayer_l 介绍完，或许就不太晕了。

（3）addLayer_l 分析

addLayer_l 把这个新创建的 layer 加入到自己的 Z 轴大军，下面来看：

[-->SurfaceFlinger.cpp]

```
status_t SurfaceFlinger::addLayer_l(const sp<LayerBase>& layer)
{
    /*
    mCurrentState 是 SurfaceFlinger 定义的一个结构，它有一个成员变量叫
    layersSortedByZ，其实就是一个排序数组。下面这个 add 函数将把这个新的 layer 按照
    它在 Z 轴的位置加入到排序数组中。mCurrentState 保存了所有的显示层。
    */
    ssize_t i = mCurrentState.layersSortedByZ.add(
                                layer, &LayerBase::compareCurrentStateZ);
    sp<LayerBaseClient> lbc =
                    LayerBase::dynamicCast< LayerBaseClient* >(layer.get());
    if (lbc != 0) {
        mLayerMap.add(lbc->serverIndex(), lbc);
    }
    return NO_ERROR;
}
```

对 Layer 的三个关键函数都已分析过了，下面正式介绍 Layer 家族。

（4）Layer 家族介绍

前面的内容确让人头晕眼花，现在应该帮大家恢复清晰的头脑。先来“一剂猛药”，见图 8-16：

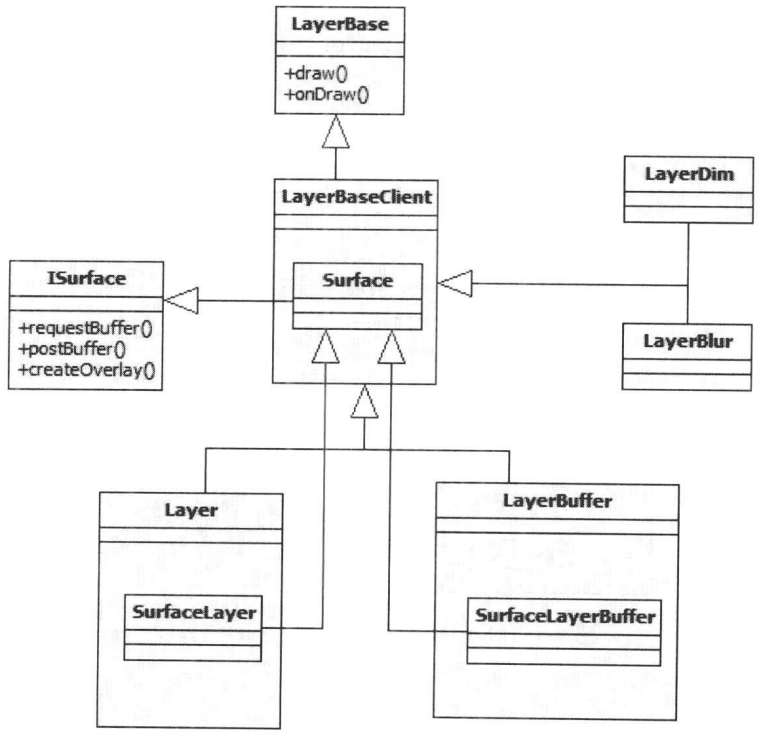


图 8-16　Layer 家族

通过上图可知：

- LayerBaseClient 从 LayerBase 类派生。
- LayerBaseClient 还有四个派生类，分别是 Layer、LayerBuffer、LayerDim 和 LayerBlur。
- LayerBaseClient 定义了一个内部类 Surface，这个 Surface 从 ISurface 类派生，它支持 Binder 通信。
- 针对不同的类型，Layer 和 LayerBuffer 分别有一个内部类 SurfaceLayer 和 SurfaceLayerBuffer，它们继承了 LayerBaseClient 的 Surface 类。所以对于 Normal 类型的显示层来说，getSurface 返回的 ISurface 对象的真正类型是 SurfaceLayer。
- LayerDim 和 LayerBlur 类没有定义自己的内部类，所以对于这两种类型的显示层来说，它们直接使用了 LayerBaseClient 的 Surface。
- ISurface 接口提供了非常简单的函数，如 requestBuffer、postBuffer 等。

这里大量使用了内部类。我们知道，内部类最终都会把请求派发给外部类对象来处理，既然如此，在以后的分析中，如果没有特殊情况，就会直接跳到外部类的处理函数中。

注意 强烈建议 Google 把 Surface 相关的代码好好整理一下，至少让类型名取得更直观些，现在这样确实有点让人头晕。好了，咱们来小小娱乐一下。之前介绍的和“Surface”有关的名字为：

Native 层有 Surface、ISurface、SurfaceControl、SurfaceComposerClient。

Java 层有 Surface、SurfaceSession。

在介绍完 Layer 家族后，看看与它相关的名字又多了几个，它们是：

LayerBaseClient::Surface、Layer::SurfaceLayer、LayerBuffer::SurfaceLayerBuffer。

---

3. 关于 SurfaceControl 的总结

SurfaceControl 创建后得到了什么呢？可用图 8-17 来表示：

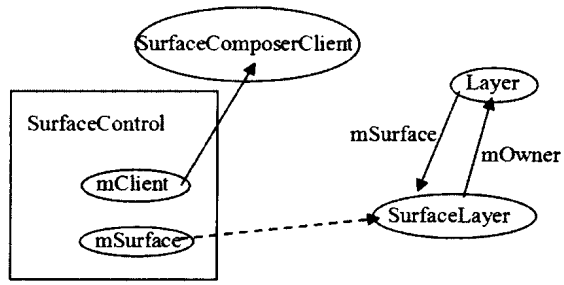


图 8-17 SurfaceControl 创建后的结果图

通过上图可以知道：

- mClient 成员变量指向 SurfaceComposerClient。

- mSurface 的 Binder 通信响应端为 SurfaceLayer。
- SurfaceLayer 有一个变量 mOwner 指向它的外部类 Layer，而 Layer 有一个成员变量 mSurface 指向 SurfaceLayer。这个 SurfaceLayer 对象由 getSurface 函数返回。

注意　mOwner 变量由 SurfaceLayer 的基类 Surface（LayBaseClient 的内部类）定义。

接下来就是 writeToParcel 分析和 Native Surface 对象的创建了。注意，这个 Native 的 Surface 可不是 LayBaseClient 的内部类 Surface。

8.4.4　writeToParcel 和 Surface 对象的创建

从乾坤大挪移的知识可知，前面创建的所有对象都在 WindowManagerService 所在的进程 system_server 中，而 writeToParcel 则需要把一些信息打包到 Parcel 后，发送到 Activity 所在的进程中。到底哪些内容需要回传给 Activity 所在的进程呢？

注意　后文将 Activity 所在的进程简称为 Activity 端。

1. writeToParcel 分析

writeToParcel 比较简单，就是把一些信息写到 Parcel 中去。代码如下所示：

[-->SurfaceControl.cpp]

```
status_t SurfaceControl::writeSurfaceToParcel(
        const sp<SurfaceControl>& control, Parcel* parcel)
{
    uint32_t flags = 0;
    uint32_t format = 0;
    SurfaceID token = -1;
    uint32_t identity = 0;
    uint32_t width = 0;
    uint32_t height = 0;
    sp<SurfaceComposerClient> client;
    sp<ISurface> sur;
    if (SurfaceControl::isValid(control)) {
        token    = control->mToken;
        identity = control->mIdentity;
        client   = control->mClient;
        sur      = control->mSurface;
        width    = control->mWidth;
        height   = control->mHeight;
        format   = control->mFormat;
        flags    = control->mFlags;
}
//SurfaceComposerClient 的信息需要传递到 Activity 端，这样客户端那边会构造一个
//SurfaceComposerClient 对象。
parcel->writeStrongBinder(client!=0  ? client->connection() : NULL);
```

```
    //把 ISurface 对象信息也写到 Parcel 中，这样 Activity 端那边也会构造一个 ISurface 对象。
        parcel->writeStrongBinder(sur!=0? sur->asBinder(): NULL);
        parcel->writeInt32(token);
        parcel->writeInt32(identity);
        parcel->writeInt32(width);
        parcel->writeInt32(height);
        parcel->writeInt32(format);
        parcel->writeInt32(flags);
        return NO_ERROR;
    }
```

Parce 包发到 Activity 端后，readFromParcel 将根据这个 Parcel 包构造一个 Native 的 Surface 对象，一起来看相关代码。

2. 分析 Native 的 Surface 创建过程

[-->android_view_Surface.cpp]

```
static void Surface_readFromParcel(
        JNIEnv* env, jobject clazz, jobject argParcel)
{
    Parcel* parcel = (Parcel*)env->GetIntField( argParcel, no.native_parcel);
    const sp<Surface>& control(getSurface(env, clazz));
    //根据服务端的 parcel 信息来构造客户端的 Surface。
    sp<Surface> rhs = new Surface(*parcel);
    if (!Surface::isSameSurface(control, rhs)) {
        setSurface(env, clazz, rhs);
    }
}
```

Native 的 Surface 是怎么利用这个 Parcel 包的呢？代码如下所示：

[-->Surface.cpp]

```
Surface::Surface(const Parcel& parcel)
:mBufferMapper(GraphicBufferMapper::get()),
mSharedBufferClient(NULL)
{
/*
  Surface 定义了一个 mBuffers 变量，它是一个 sp<GraphicBuffer> 的二元数组，也就是说 Surface
  也存在两个 GraphicBuffer，可之前在创建 Layer 的时候也有两个 GraphicBuffer，难道一共有四个
  GraphicBuffer？这个问题后面再解答。
*/
sp<IBinder> clientBinder = parcel.readStrongBinder();
//得到 ISurface 的 Bp 端 BpSurface。
    mSurface     = interface_cast<ISurface>(parcel.readStrongBinder());
    mToken       = parcel.readInt32();
    mIdentity    = parcel.readInt32();
```

```
    mWidth      = parcel.readInt32();
    mHeight     = parcel.readInt32();
    mFormat     = parcel.readInt32();
    mFlags      = parcel.readInt32();

if (clientBinder != NULL) {
     /*
     根据 ISurfaceFlingerClient 对象构造一个 SurfaceComposerClient 对象，注意我们
     现在位于 Activity 端，这里还没有创建 SurfaceComposerClient 对象，所以需要创建一个。
      */
        mClient = SurfaceComposerClient::clientForConnection(clientBinder);
       //SharedBuffer 家族的最后一员 ShardBufferClient 终于出现了。
        mSharedBufferClient = new SharedBufferClient(
                            mClient->mControl, mToken, 2, mIdentity);
    }

    init();// 做一些初始化工作。
}
```

在 Surface 创建完后，得到什么了呢？看图 8-18 就可知道：

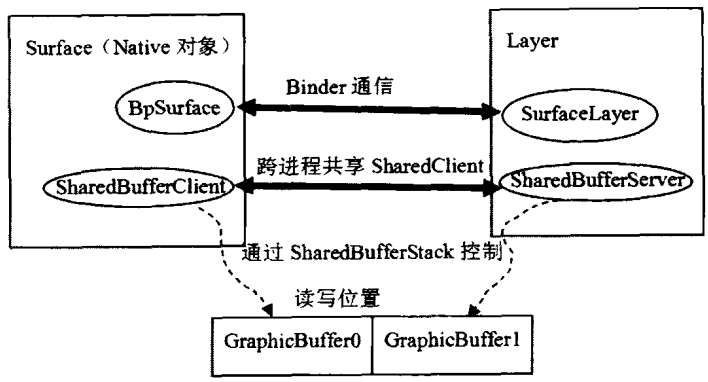


图 8-18　Native Surface 的示意图

上图很清晰地说明：

- ShardBuffer 家族依托共享内存结构 SharedClient 与它共同组成了 Surface 系统生产 / 消费协调的中枢控制机构，它在 SF 端的代表是 SharedBufferServer，在 Activity 端的代表是 SharedBufferClient。
- Native 的 Surface 将和 SF 中的 SurfaceLayer 建立 Binder 联系。

另外，图中还特意画出了承载数据的 GraphicBuffer 数组，在代码的注释中也针对 GraphicBuffer 提出了一个问题：Surface 中有两个 GraphicBuffer，Layer 中也有两个，一共就有四个 GraphicBuffer 了，可是为什么这里只画出了两个呢？

答案是，咱们不是有共享内存吗？这四个 GraphicBuffer 其实操纵的是同一段共享内存，

为了简单，所以就只画了两个 GraphicBuffer。在 8.4.7 节再介绍 GraphicBuffer 的故事。

下面，来看中枢控制机构的 SharedBuffer 家族。

3.SharedBuffer 家族介绍

（1）SharedBuffer 家族成员

SharedBuffer 是一个家族名称，它包括多少成员呢？来看 SharedBuffer 的家族图谱，如图 8-19 所示：

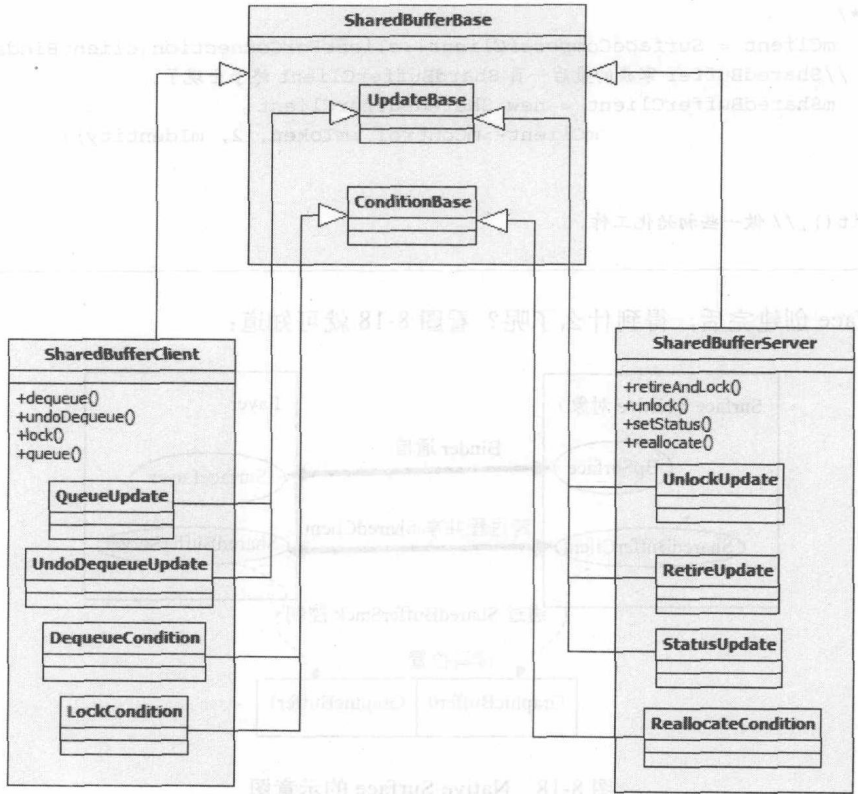


图 8-19　SharedBuffer 家族介绍

从上图可以知道：

- XXXCondition、XXXUpdate 等都是内部类，它们主要是用来更新读写位置的。不过这些操作为什么要通过类来封装呢？因为 SharedBuffer 的很多操作都使用了 C++ 中的 Function Object（函数对象），而这些内部类的实例就是函数对象。函数对象是什么？它怎么使用？对此，在以后的分析中会介绍。

（2）SharedBuffer 家族和 SharedClient 的关系

前面介绍过，SharedBufferServer 和 SharedBufferClient 控制的其实只是 SharedBufferStack 数组中的一个，下面通过 SharedBufferBase 的构造函数，来看是否如此。代码如下所示：

[-->SharedBufferStack.cpp]

```
SharedBufferBase::SharedBufferBase(SharedClient* sharedClient,
        int surface, int num, int32_t identity)
: mSharedClient(sharedClient),
  mSharedStack(sharedClient->surfaces + surface),
  mNumBuffers(num), // 根据前面 PageFlipping 的知识可知，num 值为 2
mIdentity(identity)
{
 /*
   上面的赋值语句中最重要的是第二句：
   mSharedStack(sharedClient->surfaces + surface)
   这条语句使得这个 SharedBufferXXX 对象和 SharedClient 中 SharedBufferStack 数组
   的第 surface 个元素建立了关系。
*/
}
```

4. 关于 Native Surface 的总结

至此，Activity 端 Java 的 Surface 对象，终于和一个 Native Surface 对象挂上了钩，并且这个 Native Surface 还准备好了绘图所需的一切，其中包括：

- 两个 GraphicBuffer，这就是 PageFlipping 所需的 FrontBuffer 和 BackBuffer。
- SharedBufferServer 和 SharedBufferClient 结构，这两个结构将用于生产 / 消费的过程控制。
- 一个 ISurface 对象，这个对象连接着 SF 中的一个 SurfaceLayer 对象。
- 一个 SurfaceComposerClient 对象，这个对象连接着 SF 中的一个 BClient 对象。

资源都已经准备好了，可以开始绘制 UI 了。下面，分析两个关键的函数 lockCanvas 和 unlockCanvasAndPost。

8.4.5　lockCanvas 和 unlockCanvasAndPost 分析

这一节，分析精简流程中的最后两个函数 lockCanvas 和 unlockCanvasAndPost。

1. lockCanvas 分析

根据前文的分析可知，UI 在绘制前都需要通过 lockCanvas 得到一块存储空间，也就是所说的 BackBuffer。这个过程中最终会调用 Surface 的 lock 函数。其代码如下所示：

[-->Surface.cpp]

```
status_t Surface::lock(SurfaceInfo* other, Region* dirtyIn, bool blocking)
{
    // 传入的参数中，other 用来接收一些返回信息，dirtyIn 表示需要重绘的区域。
    ......
    if (mApiLock.tryLock() != NO_ERROR) {// 多线程的情况下要锁住。
     ......
     return WOULD_BLOCK;
```

```
}
    // 设置 usage 标志，这个标志在 GraphicBuffer 分配缓冲时有指导作用。
    setUsage(GRALLOC_USAGE_SW_READ_OFTEN | GRALLOC_USAGE_SW_WRITE_OFTEN);

    // 定义一个 GraphicBuffer，名字就叫 backBuffer。
    sp<GraphicBuffer> backBuffer;
   // ①还记得我们说的 2 个元素的缓冲队列吗？下面的 dequeueBuffer 将取出一个空闲缓冲。
    status_t err = dequeueBuffer(&backBuffer);
   if (err == NO_ERROR) {
        // ② 锁住这块 buffer。
        err = lockBuffer(backBuffer.get());
        if (err == NO_ERROR) {
            const Rect bounds(backBuffer->width, backBuffer->height);
            Region scratch(bounds);
            Region& newDirtyRegion(dirtyIn ? *dirtyIn : scratch);

            ......
           //mPostedBuffer 是上一次绘画时使用的 Buffer，也就是现在的 frontBuffer。
            const sp<GraphicBuffer>& frontBuffer(mPostedBuffer);
            if (frontBuffer !=0 &&
                backBuffer->width  == frontBuffer->width &&
                backBuffer->height == frontBuffer->height &&
                !(mFlags & ISurfaceComposer::eDestroyBackbuffer))
            {
                const Region copyback(mOldDirtyRegion.subtract(newDirtyRegion));
                if (!copyback.isEmpty() && frontBuffer!=0) {
                    // ③把 frontBuffer 中的数据拷贝到 BackBuffer 中，这是为什么？
                    copyBlt(backBuffer, frontBuffer, copyback);
                }
            }

            mDirtyRegion = newDirtyRegion;
            mOldDirtyRegion = newDirtyRegion;


            void* vaddr;
           // 调用 GraphicBuffer 的 lock 得到一块内存，内存地址被赋值给了 vaddr，
           // 后续的作画将在这块内存上展开。
            status_t res = backBuffer->lock(
                    GRALLOC_USAGE_SW_READ_OFTEN | GRALLOC_USAGE_SW_WRITE_OFTEN,
                    newDirtyRegion.bounds(), &vaddr);

            mLockedBuffer = backBuffer;
            //other 用来接收一些信息。
            other->w      = backBuffer->width;  // 宽度信息
            other->h      = backBuffer->height;
            other->s      = backBuffer->stride;
            other->usage  = backBuffer->usage;
            other->format = backBuffer->format;
            other->bits   = vaddr; // 最重要的是这个内存地址
```

```
        }
    }
    mApiLock.unlock();
    return err;
}
```

在上面的代码中，列出了三个关键点：

- 调用 dequeueBuffer 得到一个空闲缓冲，也可以叫空闲缓冲出队。
- 调用 lockBuffer。
- 调用 copyBlt 函数，把 frontBuffer 数据拷贝到 backBuffer 中，这是为什么？

来分析这三个关键点。

（1）dequeueBuffer 分析

dequeueBuffer 的目的很简单，就是选取一个空闲的 GraphicBuffer，其代码如下所示：

[-->Surface.cpp]

```
status_t Surface::dequeueBuffer(sp<GraphicBuffer>* buffer) {
    android_native_buffer_t* out;
    status_t err = dequeueBuffer(&out);// 调用另外一个 dequeueBuffer。
    if (err == NO_ERROR) {
        *buffer = GraphicBuffer::getSelf(out);
    }
    return err;
}
```

这其中又调用了另外一个 dequeueBuffer 函数。它的代码如下所示：

[-->Surface.cpp]

```
int Surface::dequeueBuffer(android_native_buffer_t** buffer)
{
sp<SurfaceComposerClient> client(getClient());
// ①调用 SharedBufferClient 的 dequeue 函数，它返回当前空闲的缓冲号。
  ssize_t bufIdx = mSharedBufferClient->dequeue();
const uint32_t usage(getUsage());
/*
mBuffers 就是我们前面在 Surface 创建中介绍的那个二元 sp<GraphicBuffer> 数组。
这里定义的 backBuffer 是一个引用类型，也就是说如果修改 backBuffer 的信息，
就相当于修改了 mBuffers[bufIdx]。
*/
const sp<GraphicBuffer>& backBuffer(mBuffers[bufIdx]);
//mBuffers 定义的 GraphicBuffer 使用的也是无参构造函数，所以此时还没有真实的存储被创建。
   if (backBuffer == 0 || // 第一次进来满足 backBuffer 为空这个条件。
      ((uint32_t(backBuffer->usage) & usage) != usage) ||
        mSharedBufferClient->needNewBuffer(bufIdx))
{
   // 调用 getBufferLocked，需要进去看看。
```

```
            err = getBufferLocked(bufIdx, usage);
            if (err == NO_ERROR) {
                mWidth  = uint32_t(backBuffer->width);
                mHeight = uint32_t(backBuffer->height);
            }
        }
    ......
    }
```

上面列出了一个关键点，就是 SharedBufferClient 的 dequeue 函数，暂且记住这个调用，后面会有单独的章节分析生产 / 消费步调控制。先看 getBufferLocked 函数，其代码如下所示：

[-->Surface.cpp]

```
tatus_t Surface::getBufferLocked(int index, int usage)
{
    sp<ISurface> s(mSurface);
    status_t err = NO_MEMORY;
    // 注意这个 currentBuffer 也被定义为引用类型。
    sp<GraphicBuffer>& currentBuffer(mBuffers[index]);
    // 终于用上了 ISurface 对象，调用它的 requestBuffer 得到指定索引 index 的 Buffer。
    sp<GraphicBuffer> buffer = s->requestBuffer(index, usage);
    if (buffer != 0) {
        err = mSharedBufferClient->getStatus();
        if (!err && buffer->handle != NULL) {
            //getBufferMapper 返回 GraphicBufferMapper 对象。
            // 调用它的 registerBuffer 干什么？这个问题我们在 8.4.7 节回答。
            err = getBufferMapper().registerBuffer(buffer->handle);
            if (err == NO_ERROR) {
          // 把 requestBuffer 得到的值赋给 currentBuffer，由于 currentBuffer 是引用类型，
          // 实际上相当于 mBuffers[index]=buffer。
                currentBuffer = buffer;
               // 设置 currentBuffer 的编号。
                currentBuffer->setIndex(index);
                mNeedFullUpdate = true;
            }
        } else {
            err = err<0 ? err : NO_MEMORY;
        }
        return err;
}
```

至此，getBufferLocked 的目的已比较清晰了：

调用 ISurface 的 requestBuffer 得到一个 GraphicBuffer 对象，这个 GraphicBuffer 对象被设置到本地的 mBuffers 数组中。看来 Surface 定义的这两个 GraphicBuffer 和 Layer 定义的两个 GraphicBuffer 是有联系的，所以图 8-18 中只画了两个 GraphicBuffer。

我们已经知道，ISurface 的 Bn 端实际上是定义在 Layer 类中的 SurfaceLayer，下面来

看它实现的 requestBuffer。由于 SurfaceLayer 是 Layer 的内部类，它的工作最终都会交给 Layer 来处理，所以这里可直接看 Layer 的 requestBuffer 函数：

[-->Layer.cpp]

```
sp<GraphicBuffer> Layer::requestBuffer(int index, int usage)
{
    sp<GraphicBuffer> buffer;

    sp<Client> ourClient(client.promote());
  //lcblk 就是那个 SharedBufferServer 对象，下面这个调用确保 index 号 GraphicBuffer
  //没有被 SF 当做 FrontBuffer 使用。
    status_t err = lcblk->assertReallocate(index);
    ......
    if (err != NO_ERROR) {
         return buffer;
    }

    uint32_t w, h;
    {
        Mutex::Autolock _l(mLock);
        w = mWidth;
        h = mHeight;
   /*
   mBuffers 是 SF 端创建的一个二元数组，这里取出第 index 个元素，之前说过，
   mBuffers 使用的也是 GraphicBuffer 的无参构造函数，所以此时也没有真实存储被创建。
   */
        buffer = mBuffers[index];
        mBuffers[index].clear();
    }

    const uint32_t effectiveUsage = getEffectiveUsage(usage);
   if (buffer!=0 && buffer->getStrongCount() == 1) {
        //①分配物理存储，后面会分析这个。
        err = buffer->reallocate(w, h, mFormat, effectiveUsage);
    } else {
        buffer.clear();
       //使用 GraphicBuffer 的有参构造，这也使得物理存储被分配。
        buffer = new GraphicBuffer(w, h, mFormat, effectiveUsage);
        err = buffer->initCheck();
    }

    ......

    if (err == NO_ERROR && buffer->handle != 0) {
        Mutex::Autolock _l(mLock);
        if (mWidth && mHeight) {
            mBuffers[index] = buffer;
            mTextures[index].dirty = true;
        } else {
```

```
                buffer.clear();
            }
        }
        return buffer;// 返回
    }
```

不管怎样，此时跨进程的这个 requestBuffer 返回的 GraphicBuffer，已经和一块物理存储绑定到一起了。所以 dequeueBuffer 顺利返回了它所需的东西。接下来则需调用 lockBuffer。

（2）lockBuffer 分析

lockBuffer 的代码如下所示：

[-->Surface.cpp]

```
int Surface::lockBuffer(android_native_buffer_t* buffer)
{
    sp<SurfaceComposerClient> client(getClient());
    status_t err = validate();
    int32_t bufIdx = GraphicBuffer::getSelf(buffer)->getIndex();
    err = mSharedBufferClient->lock(bufIdx); // 调用 SharedBufferClient 的 lock。
     return err;
}
```

来看这个 lock 函数，代码如下所示：

[-->SharedBufferStack.cpp]

```
status_t SharedBufferClient::lock(int buf)
{
    LockCondition condition(this, buf);// 这个 buf 是 BackBuffer 的索引号。
    status_t err = waitForCondition(condition);
    return err;
}
```

注意，给 waitForCondition 函数传递的是一个 LockCondition 类型的对象，前面所说的函数对象的作用将在这里见识到，先看 waitForCondition 函数，代码如下所示：

[-->SharedBufferStack.h]

```
template <typename T> // 这是一个模板函数。
status_t SharedBufferBase::waitForCondition(T condition)
{
    const SharedBufferStack& stack( *mSharedStack );
    SharedClient& client( *mSharedClient );
    const nsecs_t TIMEOUT = s2ns(1);
    Mutex::Autolock _l(client.lock);
    while ((condition()==false) && // 注意这个 condition() 的用法。
            (stack.identity == mIdentity) &&
```

```
            (stack.status == NO_ERROR))
        {
            status_t err = client.cv.waitRelative(client.lock, TIMEOUT);
            if (CC_UNLIKELY(err != NO_ERROR)) {
                if (err == TIMED_OUT) {
                    if (condition()) {// 注意这个：condition()，condition 是一个对象。
                        break;
                    } else {
                    }
                } else {
                    return err;
                }
            }
        }
        return (stack.identity != mIdentity) ? status_t(BAD_INDEX) : stack.status;
    }
```

waitForCondition 函数比较简单，就是等待一个条件为真，这个条件是否满足由 condition() 这条语句来判断。但这个 condition 不是一个函数，而是一个对象，这又是怎么回事？

说明　这就是 Funcition Object（函数对象）的概念。函数对象的本质是一个对象，不过是重载了操作符 ()，这和重载操作符 +、- 等没什么区别。可以把它当作是一个函数来看待。

为什么需要函数对象呢？因为对象可以保存信息，所以调用这个对象的 () 函数就可以利用这个对象的信息了。

来看 condition 对象的 () 函数。刚才传进来的是 LockCondition，它的 () 定义如下：

[-->SharedBufferStack.cpp]

```
bool SharedBufferClient::LockCondition::operator()() {
    //stack、buf 等都是这个对象的内部成员，这个对象的目的就是根据读写位置判断这个 buffer 是
    // 否空闲。
     return (buf != stack.head ||
             (stack.queued > 0 && stack.inUse != buf));
}
```

SharedBufferStack 的读写控制比 Audio 中的环形缓冲看起来要简单，实际上它却比较复杂。本章会在扩展内容中进行分析。这里给读者准备一个问题，也是我之前百思不得其解的问题：

既然已经调用 dequeue 得到了一个空闲缓冲，为什么这里还要 lock 呢？

（3）拷贝旧数据

在第三个关键点中，可看到这样的代码：

[-->Surface.cpp]

```
status_t Surface::lock(SurfaceInfo* other, Region* dirtyIn, bool blocking)
{
    ......
    const sp<GraphicBuffer>& frontBuffer(mPostedBuffer);
    if (frontBuffer !=0 &&
        backBuffer->width  == frontBuffer->width &&
        backBuffer->height == frontBuffer->height &&
        !(mFlags & ISurfaceComposer::eDestroyBackbuffer))
    {
        const Region copyback(mOldDirtyRegion.subtract(newDirtyRegion));
        if (!copyback.isEmpty() && frontBuffer!=0) {
        // ③把 frontBuffer 中的数据拷贝到 BackBuffer 中，这是为什么？
        copyBlt(backBuffer, frontBuffer, copyback);
        }
    }
  ......
 }
```

上面这段代码所解决的，其实是下面这个问题：

在大部分情况下，UI 只有一小部分会发生变化（例如一个按钮被按下去，导致颜色发生变化），这一小部分 UI 只对应整个 GraphicBuffer 中的一小块存储（就是在前面代码中见到的 dirtyRegion），如果整块存储都更新，则会极大地浪费资源。怎么办？

这就需要将变化的图像和没有发生变化的图像进行叠加了。上一次绘制的信息保存在 mPostedBuffer 中，而这个 mPostedBuffer 则要在 unLockAndPost 函数中设置。这里将根据需要，把 mPostedBuffer 中的旧数据拷贝到 BackBuffer 中。后续的绘画只要更新脏区域就可以了，这会节约不少资源。

lockCanvas 返回后，应用层将在这块画布上尽情作画。假设现在已经在 BackBuffer 上绘制好了图像，下面就要通过 unlockCanvasAndPost 进行后续工作了，一起来看。

2. unlockCanvasAndPost 分析

进入精简流程的最后一步，就是 unlockCanvasAndPost 函数，它的代码如下所示：

[-->Surface.cpp]

```
status_t Surface::unlockAndPost()
{
    // 调用 GraphicBuffer 的 unlock 函数。
    status_t err = mLockedBuffer->unlock();
    //get 返回这个 GraphicBuffer 的编号，queueBuffer 将含有新数据的缓冲加入队中。
    err = queueBuffer(mLockedBuffer.get());
    mPostedBuffer = mLockedBuffer; // 保存这个 BackBuffer 为 mPostedBuffer。
    mLockedBuffer = 0;
    return err;
}
```

来看 queueBuffer 调用，代码如下所示：

[-->Surface.cpp]

```
int Surface::queueBuffer(android_native_buffer_t* buffer)
{
    sp<SurfaceComposerClient> client(getClient());

    int32_t bufIdx = GraphicBuffer::getSelf(buffer)->getIndex();
    // 设置脏 Region。
    mSharedBufferClient->setDirtyRegion(bufIdx, mDirtyRegion);
    // 更新写位置。
    err = mSharedBufferClient->queue(bufIdx);
     if (err == NO_ERROR) {
     //client 是 BpSurfaceFlinger，调用它的 signalServer，这样 SF 就知道新数据准备好了。
        client->signalServer();
    }
    return err;
}
```

这里与读写控制有关的是 queue 函数，其代码如下所示：

[-->SharedBufferStack.cpp]

```
status_t SharedBufferClient::queue(int buf)
{
    //QueueUpdate 也是一个函数对象。
    QueueUpdate update(this);
    // 调用 updateCondition 函数。
    status_t err = updateCondition( update );
    SharedBufferStack& stack( *mSharedStack );
    const nsecs_t now = systemTime(SYSTEM_TIME_THREAD);
    stack.stats.totalTime = ns2us(now - mDequeueTime[buf]);
    return err;
}
```

这个 updateCondition 函数的代码如下所示：

[-->SharedBufferStack.h]

```
template <typename T>
status_t SharedBufferBase::updateCondition(T update) {
    SharedClient& client( *mSharedClient );
    Mutex::Autolock _l(client.lock);
    ssize_t result = update();// 调用 update 对象的 () 函数。
    client.cv.broadcast(); // 触发同步对象。
    return result;
}
```

updateCondition 函数和前面介绍的 waitForCondition 函数一样，使用的都是函数对象。

queue 操作使用的是 QueueUpdate 类，关于它的故事，将在本章拓展内容中讨论。

3. 关于 lockCanvas 和 unlockCanvasAndPost 的总结

总结一下 lockCanvas 和 unlockCanvasAndPost 这两个函数的工作流程，用图 8-20 表示：

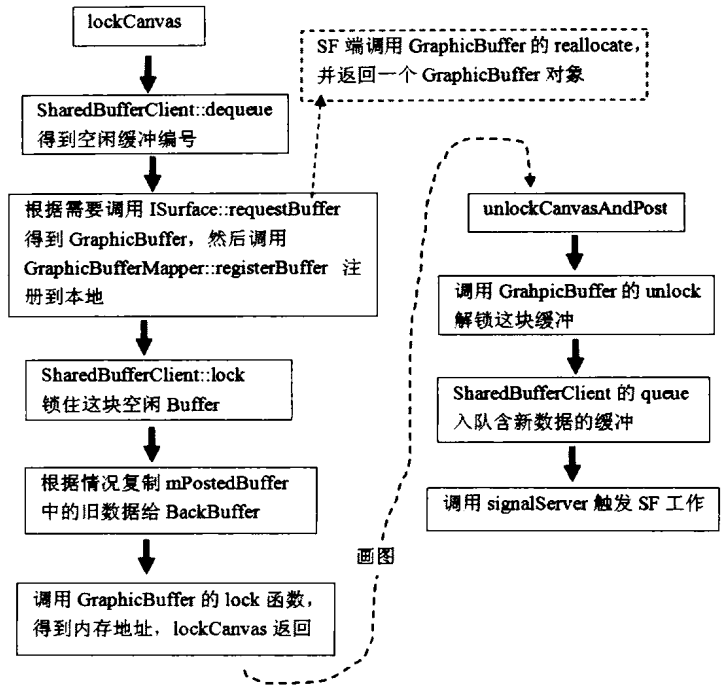


图 8-20　lockCanvas 和 unlockCanvasAndPost 的流程总结

8.4.6　GraphicBuffer 介绍

GraphicBuffer 是 Surface 系统中一个高层次的显示内存管理类，它封装了和硬件相关的一些细节，简化了应用层的处理逻辑。先来认识一下它。

1. 初识 GraphicBuffer

GraphicBuffer 的代码如下所示：

[-->GraphicBuffer.h]

```
class GraphicBuffer
    : public EGLNativeBase<android_native_buffer_t,
             GraphicBuffer,LightRefBase<GraphicBuffer> >,
public Flattenable
```

其中，EGLNativeBase 是一个模板类。它的定义代码如下所示：

[-->Android_natives.h]

```
template <typename NATIVE_TYPE, typename TYPE, typename REF>
class EGLNativeBase : public NATIVE_TYPE, public REF
```

通过替换，可得到 GraphicBuffer 的派生关系，如图 8-21 所示：

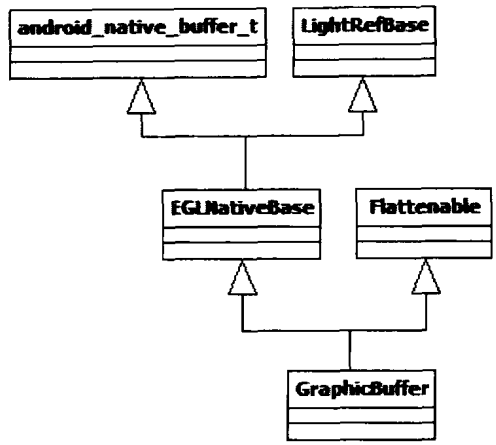


图 8-21　GraphicBuffer 派生关系的示意图

从图中可以看出：

- 从 LightRefBase 派生使 GraphicBuffer 支持轻量级的引用计数控制。
- 从 Flattenable 派生使 GraphicBuffer 支持序列化，它的 flatten 和 unflatten 函数用于序列化和反序列化，这样，GraphicBuffer 的信息就可以存储到 Parcel 包中并被 Binder 传输了。

另外，图中的 android_native_buffer_t 是 GraphicBuffer 的父类，它是一个 struct 结构体。可以将 C++ 语言中的 struct 和 class 当作同一个东西，所以 GraphicBuffer 能从它派生出来。其代码如下所示：

[-->android_native_buffer.h]

```
typedef struct android_native_buffer_t
{
#ifdef __cplusplus
    android_native_buffer_t() {
        common.magic = ANDROID_NATIVE_BUFFER_MAGIC;
        common.version = sizeof(android_native_buffer_t);
        memset(common.reserved, 0, sizeof(common.reserved));
    }
#endif
   // 这个 android_native_base_t 是 struct 的第一个成员，根据 C/C++ 编译的特性可知，这个成员
   // 在它的派生类对象所占有的内存中也是排第一个。
    struct android_native_base_t common;
```

```
    int width;
    int height;
    int stride;
    int format;
    int usage;
    void* reserved[2];
    // 这是一个关键成员，保存一些和显示内存分配 / 管理相关的内容。
    buffer_handle_t handle;

    void* reserved_proc[8];
} android_native_buffer_t;
```

GraphicBuffer 和显示内存分配相关的部分主要集中在 buffer_handle_t 这个变量上，它实际上是一个指针，定义如下：

[-->gralloc.h]

```
typedef const native_handle* buffer_handle_t;
```

native_handle 的定义如下：

[-->native_handle.h]

```
typedef struct
{
    int version;        /* version 值为 sizeof(native_handle_t) */
    int numFds;
    int numInts;
    int data[0];        /* data 是数据存储空间的首地址 */
} native_handle_t;
typedef native_handle_t native_handle;
```

读者可能要问，一个小小的 GraphicBuffer 为什么这么复杂？要回答这个问题，应先对 GraphicBuffer 有比较全面的了解。按照图 8-20 中的流程来看 GraphicBuffer。

2.GraphicBuffer 和存储的分配

GraphicBuffer 的构造函数最有可能分配存储了。注意，流程中使用的是无参构造函数，所以应先看无参构造函数。

（1）无参构造函数分析

代码如下所示：

[-->GraphicBuffer.cpp]

```
GraphicBuffer::GraphicBuffer()
    : BASE(), mOwner(ownData), mBufferMapper(GraphicBufferMapper::get()),
      mInitCheck(NO_ERROR),  mVStride(0), mIndex(-1)
{
```

```
    /*
    其中 mBufferMapper 为 GraphicBufferMapper 类型，它的创建采用的是单例模式，也就是每个
    进程只有一个 GraphicBufferMapper 对象，读者可以去看看 get 的实现。
    */
     width  =
     height =
     stride =
     format =
     usage  = 0;
     handle = NULL; //handle 为空
}
```

在无参构造函数中没有发现和存储分配有关的操作。那么，根据流程来看，下一个有可能的地方就是 reallocate 函数了。

（2）reallocate 分析

Reallocate 的代码如下所示：

[-->GraphicBuffer.cpp]

```
status_t GraphicBuffer::reallocate(uint32_t w, uint32_t h, PixelFormat f,
        uint32_t reqUsage)
{
    if (mOwner != ownData)
        return INVALID_OPERATION;

    if (handle) {//handle 值在无参构造函数中初始化为空，所以不满足 if 的条件。
        GraphicBufferAllocator& allocator(GraphicBufferAllocator::get());
        allocator.free(handle);
        handle = 0;
    }
    return initSize(w, h, f, reqUsage);// 调用 initSize 函数。
}
```

InitSize 函数的代码如下所示：

[-->GraphicBuffer.cpp]

```
status_t GraphicBuffer::initSize(uint32_t w, uint32_t h, PixelFormat format,
        uint32_t reqUsage)
{
    if (format == PIXEL_FORMAT_RGBX_8888)
        format = PIXEL_FORMAT_RGBA_8888;
   /*
    GraphicBufferAllocator 才是真正的存储分配的管理类，它的创建也是采用的单例模式，
    也就是说每个进程只有一个 GraphicBufferAllocator 对象。
   */
    GraphicBufferAllocator& allocator = GraphicBufferAllocator::get();
    // 调用 GraphicBufferAllocator 的 alloc 来分配存储，注意 handle 作为指针
    // 被传了进去，看来 handle 的值会被修改。
```

```
    status_t err = allocator.alloc(w, h, format, reqUsage, &handle, &stride);
    if (err == NO_ERROR) {
        this->width  = w;
        this->height = h;
        this->format = format;
        this->usage  = reqUsage;
        mVStride = 0;
    }
    return err;
}
```

（3）GraphicBufferAllocator 介绍

从上面的代码中可以发现，GraphicBuffer 的存储分配和 GraphicBufferAllocator 有关。一个小小的存储分配为什么需要经过这么多道工序呢？还是先来看 GraphicBufferAllocator 吧，代码如下所示：

[-->GraphicBufferAllocator.cpp]

```
GraphicBufferAllocator::GraphicBufferAllocator()
    : mAllocDev(0)
{
    hw_module_t const* module;
    // 调用 hw_get_module，得到 hw_module_t
    int err = hw_get_module(GRALLOC_HARDWARE_MODULE_ID, &module);
    if (err == 0) {
    // 调用 gralloc_open 函数，注意我们把 module 参数传了进去。
        gralloc_open(module, &mAllocDev);
    }
}
```

GraphicBufferAllocator 在创建时，首先会调用 hw_get_module 取出一个 hw_module_t 类型的对象。从名字上看，它和硬件平台有关系。它会加载一个叫“libgralloc. 硬件平台名 .so”的动态库。比如，我的 HTC G7 手机上加载的库是 /system/lib/hw/libgraolloc.qsd-8k.so。这个库的源代码在 hardware/msm7k/libgralloc-qsd8k 目录下。

这个库有什么用呢？简言之，就是为了分配一块用于显示的内存，但为什么需要这种层层封装呢？答案很简单：

封装的目的就是为了屏蔽不同硬件平台的差别。

说明 读者可通过执行 adb getprop ro.board.platform 命令，得到具体手机上硬件平台的名字。图 8-22 总结了 GraphicBufferAllocator 分配内存的途径。这部分代码读者可参考 hardware/libhardware/hardware.c 和 hardware/msm7k/libgralloc-qsd8k/gralloc.cpp，后文将不再深入探讨和硬件平台有关的知识。

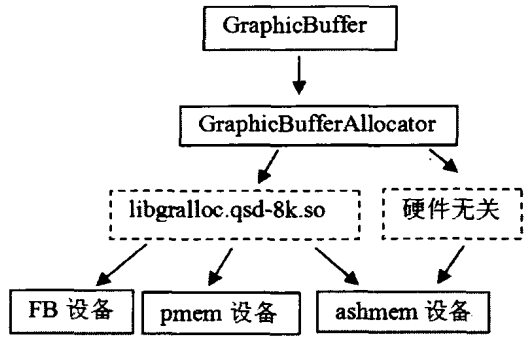


图 8-22　GraphicBufferAllocator 内存的分配途径

注意　这里是以 G7 的 libgralloc.qsk-8k.so 为示例的。其中 pmem 设备用来创建一块连续的内存，因为有些硬件设备（例如 Camera）工作时需要使用一块连续的内存，对于这种情况，一般就会使用 pmem 设备来分配内存了。

这里仅讨论图 8-22 中与硬件无关的分配方式。在这种情况下，将使用 ashmem 分配共享内存。下面看 GraphicBufferAllocator 的 alloc 函数，其代码如下所示：

[-->GraphicBufferAllocator.cpp]

```
status_t GraphicBufferAllocator::alloc(uint32_t w, uint32_t h, PixelFormat
format,int usage, buffer_handle_t* handle, int32_t* stride)
{
   // 根据前面的定义可知 buffer_handle_t 为 native_handle_t* 类型。
    status_t err;

    if (usage & GRALLOC_USAGE_HW_MASK) {
    err = mAllocDev->alloc(mAllocDev, w, h, format, usage, handle, stride);
} else {
     //SW 分配，可以做到和 HW 无关了。
     err = sw_gralloc_handle_t::alloc(w, h, format, usage, handle, stride);
    }
......
    return err;
}
```

下面，来看软件分配的方式，代码如下所示：

[-->GraphicBufferAllocator.cpp]

```
status_t sw_gralloc_handle_t::alloc(uint32_t w, uint32_t h, int format,
        int usage, buffer_handle_t* pHandle, int32_t* pStride)
{
    int align = 4;
    int bpp = 0;
```

```
    ..... //格式转换
    size_t bpr = (w*bpp + (align-1)) & ~(align-1);
    size_t size = bpr * h;
    size_t stride = bpr / bpp;
    size = (size + (PAGE_SIZE-1)) & ~(PAGE_SIZE-1);
    //直接使用了 ashmem 创建共享内存。
    int fd = ashmem_create_region("sw-gralloc-buffer", size);

   ......
    //进行内存映射，得到共享内存的起始地址。
    void* base = mmap(0, size, prot, MAP_SHARED, fd, 0);

    sw_gralloc_handle_t* hnd = new sw_gralloc_handle_t();
    hnd->fd = fd;//保存文件描述符。
    hnd->size = size;//保存共享内存的大小。
    hnd->base = intptr_t(base);//intptr_t 将 void* 类型转换成 int* 类型。
    hnd->prot = prot;//保存属性。
    *pStride = stride;
    *pHandle = hnd; //pHandle 就是传入的那个 handle 变量的指针，这里对它进行赋值。

    return NO_ERROR;
}
```

我们知道，调用 GraphicBuffer 的 reallocate 函数后，会导致物理存储被分配。前面曾说过，Layer 会创建两个 GraphicBuffer，而 Native Surface 端也会创建两个 GraphicBuffer，那么这两个 GraphicBuffer 是怎么建立联系的呢？为什么说 native_handle_t 是 GraphicBuffer 的精髓呢？

3. flatten 和 unflatten 分析

试想，Native Surface 的 GraphicBuffer 是怎么和 Layer 的 GraphicBuffer 建立联系的？

先通过 requestBuffer 函数返回一个 GraphicBuffer，然后这个 GraphicBuffer 被 Native Surface 保存。

这中间的过程其实是一个 mini 版的乾坤挪移，来看看，代码如下所示：

[-->ISurface.cpp]

```
//requestBuffer 的响应端
status_t BnSurface::onTransact(
   uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
      case REQUEST_BUFFER: {
            CHECK_INTERFACE(ISurface, data, reply);
            int bufferIdx = data.readInt32();
            int usage = data.readInt32();
            sp<GraphicBuffer> buffer(requestBuffer(bufferIdx, usage));
            ......
```

```
            /*
            requestBuffer 的返回值被写到 Parcel 包中，由于 GraphicBuffer 从
            Flattenable 类派生，这将导致它的 flatten 函数被调用。
            */
            return reply->write(*buffer);
        }
        .......
}
// 再来看请求端的处理，在 BpSurface 中。
virtual sp<GraphicBuffer> requestBuffer(int bufferIdx, int usage)
{
     Parcel data, reply;
     data.writeInterfaceToken(ISurface::getInterfaceDescriptor());
     data.writeInt32(bufferIdx);
     data.writeInt32(usage);
     remote()->transact(REQUEST_BUFFER, data, &reply);
     sp<GraphicBuffer> buffer = new GraphicBuffer();
     reply.read(*buffer);//Parcel 调用 unflatten 函数把信息反序列化到这个 buffer 中。
     return buffer;//requestBuffer 实际上返回的是本地 new 出来的这个 GraphicBuffer。
}
```

通过上面的代码可以发现，挪移的关键体现在 flatten 和 unflatten 函数上。请看——

（1）flatten 分析

flatten 的代码如下所示：

[-->GraphicBuffer.cpp]

```
status_t GraphicBuffer::flatten(void* buffer, size_t size,
       int fds[], size_t count) const
{
    //buffer 是装载数据的缓冲区，由 Parcel 提供。
     ......

    if (handle) {
        buf[6] = handle->numFds;
        buf[7] = handle->numInts;
        native_handle_t const* const h = handle;
       // 把 handle 的信息也写到 buffer 中。
        memcpy(fds,     h->data,             h->numFds*sizeof(int));
        memcpy(&buf[8], h->data + h->numFds, h->numInts*sizeof(int));
    }

    return NO_ERROR;
}
```

flatten 的工作就是把 GraphicBuffer 的 handle 变量信息写到 Parcel 包中。那么接收端如何使用这个包呢？这就是 unflatten 的工作了。

（2）unflatten 分析

unflatten 的代码如下所示：

[-->GraphicBuffer.cpp]

```
status_t GraphicBuffer::unflatten(void const* buffer, size_t size,
        int fds[], size_t count)
{
        ......

    if (numFds || numInts) {
        width  = buf[1];
        height = buf[2];
        stride = buf[3];
        format = buf[4];
        usage  = buf[5];
        native_handle* h = native_handle_create(numFds, numInts);
        memcpy(h->data,          fds,     numFds*sizeof(int));
        memcpy(h->data + numFds, &buf[8], numInts*sizeof(int));
        handle = h;// 根据 Parcel 包中的数据还原一个 handle。
    } else {
        width = height = stride = format = usage = 0;
        handle = NULL;
    }
    mOwner = ownHandle;
    return NO_ERROR;
}
```

unflatten 最重要的工作是，根据 Parcel 包中 native_handle 的信息，在 Native Surface 端构造一个对等的 GraphicBuffer。这样，Native Surface 端的 GraphicBuffer 实际上就和 Layer 端的 GraphicBuffer 管理着同一块共享内存了。

4. registerBuffer 分析

registerBuffer 有什么用呢？上一步调用 unflatten 后得到了代表共享内存的文件句柄，regiserBuffer 的目的就是对它进行内存映射，代码如下所示：

[-->GraphicBufferMapper.cpp]

```
status_t sw_gralloc_handle_t::registerBuffer(sw_gralloc_handle_t* hnd)
{
    if (hnd->pid != getpid()) {
        // 原来是做一次内存映射操作。
        void* base = mmap(0, hnd->size, hnd->prot, MAP_SHARED, hnd->fd, 0);
        ......
        //base 保存着共享内存的起始地址。
        hnd->base = intptr_t(base);
    }
    return NO_ERROR;
}
```

5.lock 和 unlock 分析

GraphicBuffer 在使用前需要通过 lock 来得到内存地址，使用完后又会通过 unlock 释放这块地址。在 SW 分配方案中，这两个函数的实现却非常简单，如下所示：

[-->GraphicBufferMapper.cpp]

```
//lock 操作
int sw_gralloc_handle_t::lock(sw_gralloc_handle_t* hnd, int usage,
        int l, int t, int w, int h, void** vaddr)
{
    *vaddr = (void*)hnd->base;// 得到共享内存的起始地址，后续作画就使用这块内存了。
    return NO_ERROR;
}
//unlock 操作
status_t sw_gralloc_handle_t::unlock(sw_gralloc_handle_t* hnd)
{
    return NO_ERROR;// 没有任何操作
}
```

对 GraphicBuffer 的介绍就到这里。虽然采用的是 SW 方式，但是相信读者也能通过树木领略到森林的风采。从应用层角度看，可以把 GraphicBuffer 当作一个构架在共享内存之上的数据缓冲。对于想深入研究的读者，我建议可按图 8-20 中的流程来分析。因为流程体现了调用顺序，表达了调用者的意图和目的，只有把握了流程，分析时才不会迷失在茫茫的源码海洋中，才不会被不熟悉的知识阻拦前进的脚步。

8.4.7　深入分析 Surface 的总结

Surface 系统最难的部分，是这个 Native Surface 的创建和使用，它包括三个方面：

- Activity 的 UI 和 Surface 的关系是怎样的？这是 8.2 节回答的问题。
- Activity 中所使用的 Surface 是怎么和 SurfaceFlinger 挂上关系的？这是 8.3 节回答的问题。
- 本节则对第 2 个问题进行了进一步的研究，更深入地分析了 Surface 和 SurfaceFlinger 之间的关系，以及生产 / 消费步调的中枢控制机构 SharedBuffer 家族和数据的承载者 GraphicBuffer。

从上面分析可看出，本章前四节均围绕着这个 Surface 进行讲解的，一路下来确实遇到了不少曲折和坎坷，望读者跟着源码反复阅读、体会。

8.5　SurfaceFlinger 分析

这一节要对 SurfaceFlinger 进行分析。相比较而言，SurfaceFlinger 不如 AudioFlinger 复杂。

8.5.1 SurfaceFlinger 的诞生

SurfaceFlinger 驻留于 system_server 进程，这一点和 Audio 系统的几个 Service 不太一样。它创建的位置在 SystemServer 的 init1 函数中（本书 4.3.2 节的第 3 点有介绍）。虽然位于 SystemServer 这个重要进程中，但是 SF 创建的代码却略显波澜不惊，没有什么特别之处。SF 的创建首先会调用 instantiate 函数，代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::instantiate() {
    defaultServiceManager()->addService(
            String16("SurfaceFlinger"), new SurfaceFlinger());
}
```

前面在图 8-14 中指出了 SF 同时从 BnSurfaceComposer 和 Thread 类中派生，相关代码如下所示：

```
class SurfaceFlinger : public BnSurfaceComposer, protected Thread
```

从 Thread 派生这件事给了我们一个很明确的提示：

SurfaceFlinger 会单独启动一个工作线程。

我们知道，Thread 类的工作线程要通过调用它的 run 函数来创建，那这个 run 函数是在什么地方调用的呢？当然，最有可能的就是在构造函数中：

[-->SurfaceFlinger.cpp]

```
SurfaceFlinger::SurfaceFlinger()
    :   BnSurfaceComposer(), Thread(false),
        mTransactionFlags(0),
        mTransactionCount(0),
        mResizeTransationPending(false),
        mLayersRemoved(false),
        mBootTime(systemTime()),
        mHardwareTest("android.permission.HARDWARE_TEST"),
        mAccessSurfaceFlinger("android.permission.ACCESS_SURFACE_FLINGER"),
        mDump("android.permission.DUMP"),
        mVisibleRegionsDirty(false),
        mDeferReleaseConsole(false),
        mFreezeDisplay(false),
        mFreezeCount(0),
        mFreezeDisplayTime(0),
        mDebugRegion(0),
        mDebugBackground(0),
        mDebugInSwapBuffers(0),
        mLastSwapBufferTime(0),
        mDebugInTransaction(0),
        mLastTransactionTime(0),
```

```
        mBootFinished(false),
        mConsoleSignals(0),
        mSecureFrameBuffer(0)
{
    init();// 上面没有调用 run。必须到 init 去检查一番。
}
//init 函数更简单了。
void SurfaceFlinger::init()
{
    char value[PROPERTY_VALUE_MAX];
    property_get("debug.sf.showupdates", value, "0");
    mDebugRegion = atoi(value);
    property_get("debug.sf.showbackground", value, "0");
    mDebugBackground = atoi(value);
}
```

嗯？上面的代码竟然没有创建工作线程？难道在其他地方？读者别急着在文件中搜索“run”，先猜测一下答案。

根据之前所学的知识，另外一个最有可能的地方就是 onFirstRef 函数了，这个函数在对象第一次被 sp 化后调用，很多初始化的工作也可以在这个函数中完成。

事实是这样吗？一起来看——

1. onFirstRef 分析

onFirstRef 的代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::onFirstRef()
{
    // 梦里寻他千百度，却是在 onFirstRef 中创建了工作线程。
    run("SurfaceFlinger", PRIORITY_URGENT_DISPLAY);
    /*
    mReadyToRunBarrier 类型为 Barrier，这个类封装了一个 Mutex 对象和一个 Condition
    对象。如果读者还记得第 5 章有关同步类的介绍，理解这个 Barrier 就非常简单了。下面调用的
    wait 函数表示要等待一个同步条件满足。
    */
    mReadyToRunBarrier.wait();
}
```

onFirstRef 创建工作线程后，将等待一个同步条件，那么这个同步条件在哪里被触发呢？相信不用多说大家也知道：

在工作线程中被触发，而且极有可能是在 readyToRun 函数中。

注意　不清楚 Thread 类的读者可以复习一下与第 5 章有关 Thread 类的知识。

2.readyToRun 分析

SF 的 readyToRun 函数将完成一些初始化工作，代码如下所示：

[-->SurfaceFlinger.cpp]

```
status_t SurfaceFlinger::readyToRun()
{
    int dpy = 0;
    {
        // ① GraphicPlane 是什么？
        GraphicPlane& plane(graphicPlane(dpy));
        // ②为这个 GraphicPlane 设置一个 HAL 对象——DisplayHardware。
        DisplayHardware* const hw = new DisplayHardware(this, dpy);
        plane.setDisplayHardware(hw);
    }

// 创建 Surface 系统中的“CB”对象，按照老规矩，应该先创建一块共享内存，然后使用 placment new。
    mServerHeap = new MemoryHeapBase(4096,
                                     MemoryHeapBase::READ_ONLY,
                                     "SurfaceFlinger read-only heap");
  /*
   注意这个“CB”对象的类型是 surface_flinger_cblk_t。为什么在 CB 上打引号呢？因为这个对象
   谈不上什么控制，只不过被用来存储一些信息罢了。其控制作用完全达不到 audio_track_cblk_t
   的程度。基于这样的事实，我们把前面提到的 SharedBuffer 家族称之为 CB 对象。
   */
  mServerCblk =
     static_cast<surface_flinger_cblk_t*>(mServerHeap->getBase());
  //placement new 创建 surface_flinger_cblk_t。
   new(mServerCblk) surface_flinger_cblk_t;

   const GraphicPlane& plane(graphicPlane(dpy));
   const DisplayHardware& hw = plane.displayHardware();
   const uint32_t w = hw.getWidth();
   const uint32_t h = hw.getHeight();
   const uint32_t f = hw.getFormat();
   hw.makeCurrent();

   // 当前只有一块屏。
   mServerCblk->connected |= 1<<dpy;
   // 屏幕在“CB”对象中的代表是 display_cblk_t。
   display_cblk_t* dcblk = mServerCblk->displays + dpy;
   memset(dcblk, 0, sizeof(display_cblk_t));
   dcblk->w            = plane.getWidth();
   dcblk->h            = plane.getHeight();
   ......// 获取屏幕信息。

  // 还用上了内联汇编语句。
  asm volatile ("":::"memory");
 /*
  下面是一些和 OpenGL 相关的函数调用。读者如果感兴趣，可以研究一下，
```

```
        至少 SurfaceFlinger.cpp 中所涉及的相关代码还不算难懂。
      */
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, 0);
        ......
        glOrthof(0, w, h, 0, 0, 1);

        //LayerDim 是 Dim 类型的 Layer。
        LayerDim::initDimmer(this, w, h);

        // 还记得在 onFirstRef 函数中的 wait 吗？下面的 open 将触发这个同步条件。
        mReadyToRunBarrier.open();
        // 资源准备好后，init 将启动 bootanim 程序，这样就见到开机动画了。
        property_set("ctl.start", "bootanim");

        return NO_ERROR;
    }
```

在上面的代码中，列出了两个关键点（即①和②），下面一一进行分析。

（1）GraphicPlane 介绍

GraphicPlane 是屏幕在 SF 代码中的对应物，根据前面的介绍可知，目前 Android 只支持一块屏幕，所以 SF 定义了一个一元数组：

```
GraphicPlane        mGraphicPlanes[1];
```

GraphicPlane 虽无什么特别之处，但它有一个重要的函数，叫 setDisplayHardware，这个函数把代表显示设备的 HAL 对象和 GraphicPlane 关联起来了。这也是下面要介绍的第二个关键点 DisplayHardware。

（2）DisplayHardware 介绍

从代码上看，这个和显示相关的 HAL 对象是在工作线程中 new 出来的，先看它的构造函数，代码如下所示：

[-->DisplayHardware.cpp]

```
DisplayHardware::DisplayHardware(
        const sp<SurfaceFlinger>& flinger,
        uint32_t dpy)
    : DisplayHardwareBase(flinger, dpy)
{
    init(dpy); // 最重要的是这个 init 函数。
}
```

init 函数非常重要，应进去看看。下面先思考一个问题。

前面在介绍 FrameBuffer 时说过，显示这一块需要使用 FrameBuffer，但在 GraphicBuffer 中用的却是 ashmem 创建的共享内存。也就是说，之前在共享内存中绘制的图像和 FrameBuffer 没有什么关系。那么 FrameBuffer 是在哪里创建的呢？

答案就在 init 函数中，代码如下所示：

[-->DisplayHardware.cpp]

```
void DisplayHardware::init(uint32_t dpy)
{
    //FrameBufferNativeWindow 实现了对 FrameBuffer 的管理和操作，该类中创建了两个
    //FrameBuffer，分别起到 FrontBuffer 和 BackBuffer 的作用。
    mNativeWindow = new FramebufferNativeWindow();

    framebuffer_device_t const * fbDev = mNativeWindow->getDevice();

    mOverlayEngine = NULL;
    hw_module_t const* module;//Overlay 相关
    if (hw_get_module(OVERLAY_HARDWARE_MODULE_ID, &module) == 0) {
        overlay_control_open(module, &mOverlayEngine);
    }
    ......

    EGLint w, h, dummy;
    EGLint numConfigs=0;
    EGLSurface surface;
    EGLContext context;
    mFlags = CACHED_BUFFERS;
   //EGLDisplay 在 EGL 中代表屏幕。
    EGLDisplay display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    ......
   /*
    surface 是 EGLSurface 类型，下面这个函数会将 EGL 和 Android 中的 Display 系统绑定起来，
    后续就可以利用 OpenGL 在这个 Surface 上绘画，然后通过 eglSwappBuffers 输出图像了。
    */
    surface = eglCreateWindowSurface(display, config,
    mNativeWindow.get(), NULL);
   ......
    mDisplay = display;
    mConfig  = config;
    mSurface = surface;
    mContext = context;
    mFormat  = fbDev->format;
    mPageFlipCount = 0;
}
```

看了上面的代码，现在可以回答前面的问题了：

SF 创建 FrameBuffer，并将各个 Surface 传输的数据（通过 GraphicBuffer）混合后，再由自己传输到 FrameBuffer 中显示。

注意 本节的内容，实际上涉及另外一个比 Surface 更复杂的 Display 系统，出于篇幅和精力的原因，本书目前不打算讨论它。

8.5.2　SF 工作线程分析

SF 中的工作线程就是用来做图像混合的，比起 AudioFlinger 来，它相当简单，下面是它的代码：

[-->SurfaceFlinger.cpp]

```
bool SurfaceFlinger::threadLoop()
{
    waitForEvent();// ① 等待什么事件呢?

   if (UNLIKELY(mConsoleSignals)) {
       handleConsoleEvents();
    }
    if (LIKELY(mTransactionCount == 0)) {
       const uint32_t mask = eTransactionNeeded | eTraversalNeeded;
        uint32_t transactionFlags = getTransactionFlags(mask);
        if (LIKELY(transactionFlags)) {
            //Transaction (事务) 处理，放到本节最后来讨论。
            handleTransaction(transactionFlags);
        }
    }

    // ②处理 PageFlipping 工作。
    handlePageFlip();

    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    if (LIKELY(hw.canDraw() && !isFrozen())) {
        // ③处理重绘。
        handleRepaint();
        hw.compositionComplete();
        // ④投递 BackBuffer。
        unlockClients();
        postFramebuffer();
    } else {
        unlockClients();
        usleep(16667);
    }
    return true;
}
```

ThreadLoop 一共有四个关键点（即①～④），这里分析除 Transaction 外的三个关键点。

1. waitForEvent

SF 工作线程一上来就等待事件，那会是什么事件呢？来看代码：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::waitForEvent()
```

```
{
    while (true) {
        nsecs_t timeout = -1;
        const nsecs_t freezeDisplayTimeout = ms2ns(5000);
        ......

        MessageList::value_type msg = mEventQueue.waitMessage(timeout);

        ...... //还有一些和冻屏相关的内容。
        if (msg != 0) {
            switch (msg->what) {
                //千辛万苦就等这一个重绘消息。
                case MessageQueue::INVALIDATE:
                   return;
            }
        }
    }
}
```

SF 收到重绘消息后，将退出等待。那么，是谁发送的这个重绘消息呢？还记得在 unlockCanvasAndPost 函数中调用的 signal 吗？它在 SF 端的实现代码如下：

[-->SurfaceFlinger]

```
void SurfaceFlinger::signal() const {
     const_cast<SurfaceFlinger*>(this)->signalEvent();
}
void SurfaceFlinger::signalEvent() {
    mEventQueue.invalidate(); //往消息队列中加入 INVALIDATE 消息。
}
```

2.handlePageFlip 分析

SF 工作线程从 waitForEvent 中返回后，下一步要做的就是处理事务和 handlePageFlip 了。先看 handlePageFlip，从名字上可知，它和 PageFlipping 工作有关。

注意 事务处理将在 8.5.3 节中介绍。

代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::handlePageFlip()
{
    bool visibleRegions = mVisibleRegionsDirty;
    /*
    还记得前面所说的 mCurrentState 吗？它保存了所有显示层的信息，而绘制的时候使用的
    mDrawingState 则保存了当前需要显示的显示层信息。
    */
```

```
    LayerVector& currentLayers =
                    const_cast<LayerVector&>(mDrawingState.layersSortedByZ);
    // ①调用 lockPageFlip。
    visibleRegions |= lockPageFlip(currentLayers);
    const DisplayHardware& hw = graphicPlane(0).displayHardware();
    // 取得屏幕的区域。
    const Region screenRegion(hw.bounds());
    if (visibleRegions) {
          Region opaqueRegion;
          computeVisibleRegions(currentLayers, mDirtyRegion,opaqueRegion);
          mWormholeRegion = screenRegion.subtract(opaqueRegion);
          mVisibleRegionsDirty = false;
      }
    // ② 调用 unlockPageFlip。
    unlockPageFlip(currentLayers);
    mDirtyRegion.andSelf(screenRegion);
}
```

hanldePageFlip 调用了两个看起来是一对的函数：lockPageFlip 和 unlockPageFlip。这两个函数会干些什么呢？

（1）lockPageFlip 分析

先看 lockPageFlip 函数，代码如下所示：

[-->SurfaceFlinger.cpp]

```
bool SurfaceFlinger::lockPageFlip(const LayerVector& currentLayers)
{
    bool recomputeVisibleRegions = false;
    size_t count = currentLayers.size();
    sp<LayerBase> const* layers = currentLayers.array();
    for (size_t i=0 ; i<count ; i++) {
        const sp<LayerBase>& layer = layers[i];
        // 调用每个显示层的 lockPageFlip。
        layer->lockPageFlip(recomputeVisibleRegions);
    }
    return recomputeVisibleRegions;
}
```

假设当前的显示层是 Layer 类型，那么得转到 Layer 类去看它的 lockPageFlip 函数了，代码如下所示：

[-->Layer.cpp]

```
void Layer::lockPageFlip(bool& recomputeVisibleRegions)
{
  //lcblk 是 SharedBufferServer 类型，调用 retireAndLock 函数将返回 FrontBuffer 的
  // 索引号。
    ssize_t buf = lcblk->retireAndLock();
    ......
```

```
    mFrontBufferIndex = buf;

    // 得到 FrontBuffer 对应的 GraphicBuffer。
    sp<GraphicBuffer> newFrontBuffer(getBuffer(buf));
    if (newFrontBuffer != NULL) {
        // 取出脏区域。
       const Region dirty(lcblk->getDirtyRegion(buf));
        // 和 GraphicBuffer 所表示的区域进行裁剪，得到一个脏区域。
        mPostedDirtyRegion = dirty.intersect( newFrontBuffer->getBounds() );

        const Layer::State& front(drawingState());
        if (newFrontBuffer->getWidth()  == front.requested_w &&
            newFrontBuffer->getHeight() == front.requested_h)
        {
            if ((front.w != front.requested_w) ||
                (front.h != front.requested_h))
            {
                ..... // 需要重新计算可见区域。
                recomputeVisibleRegions = true;
            }
            mFreezeLock.clear();
        }
    } else {
        mPostedDirtyRegion.clear();
    }
    if (lcblk->getQueuedCount()) {
        mFlinger->signalEvent();
    }
/*
如果脏区域不为空，则需要绘制成纹理，reloadTexture 将绘制一张纹理保存在
mTextures 数组中，里边涉及很多 OpenGL 的操作，读者有兴趣可以自己研究。
*/
if (!mPostedDirtyRegion.isEmpty()) {
        reloadTexture( mPostedDirtyRegion );
    }
}
```

我们知道，Layer 的 lockPageFlip 将根据 FrontBuffer 的内容生成一张纹理。那么，unlockPageFlip 会做些什么呢？

（2）unlockPageFlip 分析

unlockPageFlip 的代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::unlockPageFlip(const LayerVector& currentLayers)
{
     const GraphicPlane& plane(graphicPlane(0));
    const Transform& planeTransform(plane.transform());
```

```
    size_t count = currentLayers.size();
    sp<LayerBase> const* layers = currentLayers.array();
    for (size_t i=0 ; i<count ; i++) {
        const sp<LayerBase>& layer = layers[i];
        // 调用每个显示层的 unlockPageFlip，Layer 的 unlockPageFlip 主要做一些
        // 区域的清理工作，读者可以自己看看。
        layer->unlockPageFlip(planeTransform, mDirtyRegion);
    }
}
```

（3）关于 handlePageFlip 的总结

handlePageFlip 的工作其实很简单，以 Layer 类型为例来总结一下：

各个 Layer 需要从 FrontBuffer 中取得新数据，并生成一张 OpenGL 中的纹理。纹理可以看做是一个图片，这个图片的内容就是 FrontBuffer 中的图像。

现在每一个 Layer 都准备好了新数据，下一步的工作当然就是绘制了。来看 handleRepaint 函数。

3.handleRepaint 函数分析

handleRepaint 函数的代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::handleRepaint()
{
    mInvalidRegion.orSelf(mDirtyRegion);
    if (mInvalidRegion.isEmpty()) {
        return;
    }

    ......

    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    uint32_t flags = hw.getFlags();
    if ((flags & DisplayHardware::SWAP_RECTANGLE) ||
        (flags & DisplayHardware::BUFFER_PRESERVED))
    {
        ......// 计算 mDirtyRegion。
    }

    // 在脏区域上进行绘制。
    composeSurfaces(mDirtyRegion);
    mDirtyRegion.clear();
}
```

其中，composeSurfaces 将不同的显示层内容进行混合，其实就是按 Z 轴的顺序由里到

外依次绘制。当然，最后绘制的数据有可能遮盖前面绘制的数据，代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::composeSurfaces(const Region& dirty)
{
    const SurfaceFlinger& flinger(*this);
    const LayerVector& drawingLayers(mDrawingState.layersSortedByZ);
    const size_t count = drawingLayers.size();
    sp<LayerBase> const* const layers = drawingLayers.array();
    for (size_t i=0 ; i<count ; ++i) {
        const sp<LayerBase>& layer = layers[i];
        const Region& visibleRegion(layer->visibleRegionScreen);
        if (!visibleRegion.isEmpty())  {
            const Region clip(dirty.intersect(visibleRegion));
            if (!clip.isEmpty()) {
                layer->draw(clip); // 调用各个显示层的 layer 函数。
            }
        }
    }
}
```

draw 函数在 LayerBase 类中实现，代码如下所示：

[-->LayerBase.cpp]

```
void LayerBase::draw(const Region& inClip) const
{
    ......
    glEnable(GL_SCISSOR_TEST);
    onDraw(clip);// 调用子类的 onDraw 函数。
}
```

至于 Layer 是怎么实现这个 onDraw 函数的，代码如下所示：

[-->Layer.cpp]

```
void Layer::onDraw(const Region& clip) const
{
    int index = mFrontBufferIndex;
    if (mTextures[index].image == EGL_NO_IMAGE_KHR)
        index = 0;
        GLuint textureName = mTextures[index].name;
        ......
        Region holes(clip.subtract(under));
        if (!holes.isEmpty()) {
            clearWithOpenGL(holes);
        }
        return;
}
    //index 是 FrontBuffer 对应生成的纹理，在 lockPageFlip 函数中就已经生成了。
```

```
    drawWithOpenGL(clip, mTextures[index]);// 将纹理画上去，里面有很多和 OpenGL 相关的内容。
}
```

drawWithOpenGL 函数由 LayerBase 实现，看它是不是使用了这张纹理，代码如下所示：

[-->LayerBase.cpp]

```
void LayerBase::drawWithOpenGL(const Region& clip, const Texture& texture) const
{
    const DisplayHardware& hw(graphicPlane(0).displayHardware());
    const uint32_t fbHeight = hw.getHeight();
    const State& s(drawingState());

    //validateTexture 函数内部将绑定指定的纹理。
    validateTexture(texture.name);
    // 下面就是 OpenGL 操作函数了。
    glEnable(GL_TEXTURE_2D);

    ......

    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();

    // 坐标旋转。
    switch (texture.transform) {
        case HAL_TRANSFORM_ROT_90:
            glTranslatef(0, 1, 0);
            glRotatef(-90, 0, 0, 1);
            break;
        case HAL_TRANSFORM_ROT_180:
            glTranslatef(1, 1, 0);
            glRotatef(-180, 0, 0, 1);
            break;
        case HAL_TRANSFORM_ROT_270:
            glTranslatef(1, 0, 0);
            glRotatef(-270, 0, 0, 1);
            break;
    }

    if (texture.NPOTAdjust) {
        // 缩放处理。
        glScalef(texture.wScale, texture.hScale, 1.0f);
    }
    // 使能纹理坐标
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    // 设置顶点坐标。
    glVertexPointer(2, GL_FIXED, 0, mVertices);
    // 设置纹理坐标。
    glTexCoordPointer(2, GL_FIXED, 0, texCoords);
```

```
        while (it != end) {
            const Rect& r = *it++;
            const GLint sy = fbHeight - (r.top + r.height());
            // 裁剪。
            glScissor(r.left, sy, r.width(), r.height());
            // 画矩形。
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        // 禁止纹理坐标。
        glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    }
```

纹理绑定是 OpenGL 的常用函数，其代码如下所示。

[-->LayerBase.cpp]

```
void LayerBase::validateTexture(GLint textureName) const
{
    // 下面这个函数将绑定纹理。
    glBindTexture(GL_TEXTURE_2D, textureName);
    ......// 其他一些设置
}
```

handleRepaint 这个函数基本上就是按 Z 轴的顺序对每一层进行重绘，重绘的方法就是使用 OpenGL。

注意 我在 Android 平台上有几个月的 OpenGL 开发经历，还谈不上很深刻，不过整理了一些资料，希望能够给感兴趣的读者提供参考。

1）OpenGL 的入门教材当选 NeHe 的资料，大略看前几章即可。

2）Android 平台上关于 OpenGL ES 的开发，有一篇很详细的 Word 文档叫《OpenGL ES Tutorial for Android》。该文详细地描述了在 Android 平台上进行 OpenGL 开发的流程。大家可跟着这篇教材，在模拟器上做一些练习。那里面所涉及的一些基础知识，可从前面介绍的入门教材中学到。

3）有了前面两点的基础后，就需要对整个 OpenGL 有比较完整深入的了解了。我在那时所看的书是《OpenGL Programming Guide (7th Edition)》。该书很厚，有 1000 多页。里面有一些内容可能与工作无关，只要大概知道有那回事就行了，暂时不必深入学习，等需要时再进一步学习并运用。我在开发的项目中曾用到的光照、雾化等效果，都是之前先知道有这个东西，后来在项目中才逐渐学习运用的。

4）嵌入式平台上用的其实是 OpenGL ES。这里，还有一本书叫《OpenGL ES 2.0 Programming Guide》，它介绍了 OpenGL ES 的开发，读者可认真学习。

5）在 Android SDK 文档中，对 OpenGL API 的描述只有寥寥数语。怎么办？不过，由于它使用了 J2ME 中的 javax.microedition.khronos.opengles 包，所以 J2ME 的 SDK 文档中对 OpenGL 的 API 有着非常详细的描述，读者手头应该要有一个 J2ME 的文档。

6）如果想做深入开发，就不得不学习计算机图形学了。我后来买了书，可惜没时间学了。

4. unlockClients 和 postFrameBuffer 分析

在绘制完图后，还有两项工作需要做，一个涉及 unlockClients 函数，另外一个涉及 postFrameBuffer 函数，这两个函数分别干了什么呢？ unlockClients 的代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::unlockClients()
{
    const LayerVector& drawingLayers(mDrawingState.layersSortedByZ);
    const size_t count = drawingLayers.size();
    sp<LayerBase> const* const layers = drawingLayers.array();
    for (size_t i=0 ; i<count ; ++i) {
        const sp<LayerBase>& layer = layers[i];
        layer->finishPageFlip();
    }
}
```

再看 Layer 的 finishPageFlip 函数，代码如下所示：

[-->Layer.cpp]

```
void Layer::finishPageFlip()
{
    // 释放 FrontBufferIndex。
    status_t err = lcblk->unlock( mFrontBufferIndex );
}
```

原来，unlockClients 会释放之前占着的 FrontBuffer 的索引号。下面看最后一个函数 postFrameBuffer，代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::postFramebuffer()
{
    if (!mInvalidRegion.isEmpty()) {
        const DisplayHardware& hw(graphicPlane(0).displayHardware());
        const nsecs_t now = systemTime();
        mDebugInSwapBuffers = now;
        // 调用这个函数后，混合后的图像就会传递到屏幕中显示了。
        hw.flip(mInvalidRegion);
        mLastSwapBufferTime = systemTime() - now;
        mDebugInSwapBuffers = 0;
        mInvalidRegion.clear();
    }
}
```

flip 将调用在 DisplayHardware 一节中提到的 eglSwapBuffer 函数，以完成 FrameBuffer 的 PageFlip 操作，代码如下所示：

[-->DisplayHardware.cpp]

```
void DisplayHardware::flip(const Region& dirty) const
{
    checkGLErrors();

    EGLDisplay dpy = mDisplay;
    EGLSurface surface = mSurface;

    ......
    if (mFlags & PARTIAL_UPDATES) {
        mNativeWindow->setUpdateRectangle(dirty.getBounds());
    }

    mPageFlipCount++;
    eglSwapBuffers(dpy, surface);//PageFlipping，此后图像终于显示在屏幕上了！

}
```

8.5.3 Transaction 分析

Transaction 是“事务”的意思。在我脑海中，关于事务的知识来自数据库。在数据库操作中，事务意味着一次可以提交多个 SQL 语句，然后一个 commit 就可让它们集中执行，而且数据库中的事务还可以回滚，即恢复到事务提交前的状态。

SurfaceFlinger 为什么需要事务呢？从上面对数据库事务的描述来看，是不是意味着一次执行多个请求呢？如直接盯着 SF 的源码来分析，可能不太容易搞清楚事务的前因后果，我想还是用老办法，从一个例子入手吧。

在 WindowManagerService.java 中有一个函数之前已分析过，现在再看看，代码如下所示：

[-->WindowManagerService.java::WinState]

```
Surface createSurfaceLocked() {
   Surface.openTransaction(); // 开始一次 transaction。
   try {
     try {
          mSurfaceX = mFrame.left + mXOffset;
          mSurfaceY = mFrame.top + mYOffset;
          // 设置 Surface 的位置。
          mSurface.setPosition(mSurfaceX, mSurfaceY);
          ......
        }
      } finally {
            Surface.closeTransaction(); // 关闭这次事务。
      }
```

这个例子很好地展示了事务的调用流程，它会依次调用：

- openTransaction
- setPosition
- closeTransaction

下面就来分析这几个函数的调用。

1. openTransaction 分析

看 JNI 对应的函数，代码如下所示：

[-->android_View_Surface.cpp]

```
static void Surface_openTransaction(JNIEnv* env, jobject clazz)
{
    // 调用 SurfaceComposerClient 的 openGlobalTransaction 函数。
    SurfaceComposerClient::openGlobalTransaction();
}
```

下面转到 SurfaceComposerClient，代码如下所示：

[-->SurfaceComposerClient.cpp]

```
void SurfaceComposerClient::openGlobalTransaction()
{
    Mutex::Autolock _l(gLock);
    ......

    const size_t N = gActiveConnections.size();
    for (size_t i=0; i<N; i++) {
        sp<SurfaceComposerClient>
        client(gActiveConnections.valueAt(i).promote());
        //gOpenTransactions 存储当前提交事务请求的 Client。
        if (client != 0 && gOpenTransactions.indexOf(client) < 0) {
            //Client 是保存在全局变量 gActiveConnections 中的 SurfaceComposerClient
            // 对象，调用它的 openTransaction。
            if (client->openTransaction() == NO_ERROR) {
                if (gOpenTransactions.add(client) < 0) {
                    client->closeTransaction();
                }
            }
            ......
        }
    }
}
```

上面是一个静态函数，内部调用了各个 SurfaceComposerClient 对象的 openTranscation，代码如下所示：

[-->SurfaceComposerClient.cpp]

```
status_t SurfaceComposerClient::openTransaction()
{
    if (mStatus != NO_ERROR)
        return mStatus;
    Mutex::Autolock _l(mLock);
    mTransactionOpen++; //一个计数值，用来控制事务的提交。
    if (mPrebuiltLayerState == 0) {
        mPrebuiltLayerState = new layer_state_t;
    }
    return NO_ERROR;
}
```

layer_state_t 是用来保存 Surface 的一些信息的，比如位置、宽、高等信息。实际上，调用的 setPosition 等函数，就是为了改变这个 layer_state_t 中的值。

2. setPosition 分析

上文说过，SFC 中有一个 layer_state_t 对象用来保存 Surface 的各种信息。这里以 setPosition 为例来看它的使用情况。这个函数是用来改变 Surface 在屏幕上的位置的，代码如下所示：

[-->android_View_Surface.cpp]

```
static void Surface_setPosition(JNIEnv* env, jobject clazz, jint x, jint y)
{
    const sp<SurfaceControl>& surface(getSurfaceControl(env, clazz));
    if (surface == 0) return;
    status_t err = surface->setPosition(x, y);
}
```

[-->Surface.cpp]

```
status_t SurfaceControl::setPosition(int32_t x, int32_t y) {
    const sp<SurfaceComposerClient>& client(mClient);
    status_t err = validate();
    if (err < 0) return err;
    //调用 SurfaceComposerClient 的 setPosition 函数。
    return client->setPosition(mToken, x, y);
}
```

[-->SurfaceComposerClient.cpp]

```
status_t SurfaceComposerClient::setPosition(SurfaceID id, int32_t x, int32_t y)
{
    layer_state_t* s = _lockLayerState(id); //找到对应的 layer_state_t。
    if (!s) return BAD_INDEX;
    s->what |= ISurfaceComposer::ePositionChanged;
```

```
    s->x = x;
    s->y = y;  // 上面几句修改了这块 layer 的参数。
    _unlockLayerState(); // 该函数将 unlock 一个同步对象，其他没有做什么工作。
    return NO_ERROR;
}
```

setPosition 修改了 layer_state_t 中的一些参数，那么，这个状态是什么时候传递到 SurfaceFlinger 中的呢？

3.closeTransaction 分析

相信读者此时已明白为什么叫“事务”了。原来，在 openTransaction 和 closeTransaction 中可以有很多操作，然后会由 closeTransaction 一次性地把这些修改提交到 SF 上，来看代码：

[-->android_View_Surface.cpp]

```
static void Surface_closeTransaction(JNIEnv* env, jobject clazz)
{
    SurfaceComposerClient::closeGlobalTransaction();
}
```

[-->SurfaceComposerClient.cpp]

```
void SurfaceComposerClient::closeGlobalTransaction()
{
    ......

    const size_t N = clients.size();
    sp<ISurfaceComposer> sm(getComposerService());
    // ①先调用 SF 的 openGlobalTransaction。
    sm->openGlobalTransaction();
    for (size_t i=0; i<N; i++) {
       // ②然后调用每个 SurfaceComposerClient 对象的 closeTransaction。
       clients[i]->closeTransaction();
    }
    // ③最后调用 SF 的 closeGlobalTransaction。
    sm->closeGlobalTransaction();
}
```

上面一共列出了三个函数，它们都是跨进程的调用，下面对其一一进行分析。

（1）SurfaceFlinger 的 openGlobalTransaction 分析

这个函数其实很简单，略看一下就行了。

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::openGlobalTransaction()
{
    android_atomic_inc(&mTransactionCount);// 又是一个计数控制。
}
```

（2）SurfaceComposerClient 的 closeTransaction 分析

代码如下所示：

[-->SurfaceComposerClient.cpp]

```
status_t SurfaceComposerClient::closeTransaction()
{
    if (mStatus != NO_ERROR)
        return mStatus;

    Mutex::Autolock _l(mLock);
   ......
    const ssize_t count = mStates.size();
    if (count) {
     //mStates 是这个 SurfaceComposerClient 中保存的所有 layer_state_t 数组，也就是
    // 每个 Surface 一个。然后调用跨进程的 setState。
        mClient->setState(count, mStates.array());
        mStates.clear();
    }
    return NO_ERROR;
}
```

BClient 的 setState，最终会转到 SF 的 setClientState 上，代码如下所示：

[-->SurfaceFlinger.cpp]

```
    status_t SurfaceFlinger::setClientState(ClientID cid, int32_t count,
                            const layer_state_t* states)
{
    Mutex::Autolock _l(mStateLock);
    uint32_t flags = 0;
    cid <<= 16;
    for (int i=0 ; i<count ; i++) {
        const layer_state_t& s = states[i];
        sp<LayerBaseClient> layer(getLayerUser_l(s.surface | cid));
        if (layer != 0) {
            const uint32_t what = s.what;
            if (what & ePositionChanged) {
                if (layer->setPosition(s.x, s.y))
                     //eTraversalNeeded 表示需要遍历所有显示层。
                    flags |= eTraversalNeeded;
            }
            ....
    if (flags) {
        setTransactionFlags(flags);// 这里将会触发 threadLoop 的事件。
    }
    return NO_ERROR;
}
```

[-->SurfaceFlinger.cpp]

```
uint32_t SurfaceFlinger::setTransactionFlags(uint32_t flags, nsecs_t delay)
```

```
{
    uint32_t old = android_atomic_or(flags, &mTransactionFlags);
    if ((old & flags)==0) {
        if (delay > 0) {
            signalDelayedEvent(delay);
        } else {
            signalEvent();  // 设置完 mTransactionFlags 后，触发事件。
        }
    }
    return old;
}
```

（3）SurfaceFlinger 的 closeGlobalTransaction 分析

来看代码：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::closeGlobalTransaction()
{
    if (android_atomic_dec(&mTransactionCount) == 1) {
    // 注意下面语句的执行条件，当 mTransactionCount 变为零时才执行，这意味着
    //openGlobalTransaction 两次的话，只有最后一个 closeGlobalTransaction 调用
    // 才会真正地提交事务。
        signalEvent();

        Mutex::Autolock _l(mStateLock);
       // 如果这次事务涉及尺寸调整，则需要等一段时间。
        while (mResizeTransationPending) {
            status_t err = mTransactionCV.waitRelative(mStateLock, s2ns(5));
            if (CC_UNLIKELY(err != NO_ERROR)) {
                mResizeTransationPending = false;
                break;
            }
        }
    }
}
```

关于事务的目的，相信读者已经比较清楚了：

就是将一些控制操作（例如 setPosition）的修改结果一次性地传递给 SF 进行处理。

那么，哪些操作需要通过事务来传递呢？查看 Surface.h 可以知道，下面这些操作需要通过事务来传递（这里只列出了几个经常用的函数）：setPosition、setAlpha、show/hide、setSize、setFlag 等。

由于这些修改不像重绘那么简单，有时它会涉及其他的显示层，例如在显示层 A 的位置调整后，之前被 A 遮住的显示层 B，现在可能变得可见了。对于这种情况，所提交的事务会设置 eTraversalNeeded 标志，这个标志表示要遍历所有显示层进行处理。关于这一点，我们可以看看工作线程中的事务处理。

4. 工作线程中的事务处理

还是从代码入手分析，如下所示：

[-->SurfaceFlinger.cpp]

```
bool SurfaceFlinger::threadLoop()
{
    waitForEvent();
    if (LIKELY(mTransactionCount == 0)) {
      const uint32_t mask = eTransactionNeeded | eTraversalNeeded;
      uint32_t transactionFlags = getTransactionFlags(mask);
        if (LIKELY(transactionFlags)) {
            handleTransaction(transactionFlags);
        }
    }
    ......
 }
```

getTransactionFlags 函数的实现蛮有意思，不妨看看其代码，如下所示：

[-->SurfaceFlinger.cpp]

```
uint32_t SurfaceFlinger::getTransactionFlags(uint32_t flags)
{
    // 先通过原子操作去掉 mTransactionFlags 中对应的位。
    // 而后原子操作返回的旧值和 flags 进行与操作
    return android_atomic_and(~flags, &mTransactionFlags) & flags;
}
```

getTransactionFlags 所做的工作不仅仅是 get 那么简单，它还设置了 mTransactionFlags，从这个角度来看，getTransactionFlags 这个名字有点名不副实。

接着来看最重要的 handleTransaction 函数，代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::handleTransaction(uint32_t transactionFlags)
{
    Vector< sp<LayerBase> > ditchedLayers;

    {
        Mutex::Autolock _l(mStateLock);
        // 调用 handleTransactionLocked 函数处理。
        handleTransactionLocked(transactionFlags, ditchedLayers);
    }


    const size_t count = ditchedLayers.size();
    for (size_t i=0 ; i<count ; i++) {
        if (ditchedLayers[i] != 0) {
        //ditch 是丢弃的意思，有些显示层可能被 hide 了，所以这里做些收尾的工作。
```

```
            ditchedLayers[i]->ditch();
        }
    }
}
```

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::handleTransactionLocked(
        uint32_t transactionFlags, Vector< sp<LayerBase> >& ditchedLayers)
{
    // 这里使用了 mCurrentState，它的 layersSortedByZ 数组存储了 SF 中所有的显示层。
    const LayerVector& currentLayers(mCurrentState.layersSortedByZ);
    const size_t count = currentLayers.size();

     const bool layersNeedTransaction = transactionFlags & eTraversalNeeded;
     // 如果需要遍历所有显示层的话。
    if (layersNeedTransaction) {
        for (size_t i=0 ; i<count ; i++) {
            const sp<LayerBase>& layer = currentLayers[i];
            uint32_t trFlags = layer->getTransactionFlags(eTransactionNeeded);
            if (!trFlags) continue;
            // 调用各个显示层的 doTransaction 函数。
            const uint32_t flags = layer->doTransaction(0);
            if (flags & Layer::eVisibleRegion)
                mVisibleRegionsDirty = true;
        }
    }
    if (transactionFlags & eTransactionNeeded) {
        if (mCurrentState.orientation != mDrawingState.orientation) {
          // 横竖屏如果发生切换，需要对应变换设置。
            const int dpy = 0;
            const int orientation = mCurrentState.orientation;
            const uint32_t type = mCurrentState.orientationType;
            GraphicPlane& plane(graphicPlane(dpy));
            plane.setOrientation(orientation);

            ......
        }
     /*
      mLayersRemoved 变量在显示层被移除的时候设置，例如 removeLayer 函数，这些函数
      也会触发 handleTranscation 函数的执行。
     */
      if (mLayersRemoved) {
            mLayersRemoved = false;
            mVisibleRegionsDirty = true;
            const LayerVector& previousLayers(mDrawingState.layersSortedByZ);
            const size_t count = previousLayers.size();
            for (size_t i=0 ; i<count ; i++) {
                const sp<LayerBase>& layer(previousLayers[i]);
                if (currentLayers.indexOf( layer ) < 0) {
                   ditchedLayers.add(layer);
                   mDirtyRegionRemovedLayer.orSelf(layer->visibleRegionScreen);
```

```
                }
            }
        }
        free_resources_l();
    }
    // 提交事务处理，有必要进去看看。
    commitTransaction();
}
```

每个显示层对事务的具体处理，都在它们的 doTransaction 函数中，读者若有兴趣，可进去看看。需要说明的是，每个显示层内部也有一个状态变量，doTransaction 会更新这些状态变量。

回到上面的函数，最后它将调用 commitTransaction 提交事务，代码如下所示：

[-->SurfaceFlinger.cpp]

```
void SurfaceFlinger::commitTransaction()
{
    //mDrawingState 将使用更新后的 mCurrentState。
    mDrawingState = mCurrentState;
    mResizeTransationPending = false;
    // 触发一个条件变量，这样等待在 closeGlobalTransaction 函数中的线程可以放心地返回了。
    mTransactionCV.broadcast();
}
```

8.5.4 关于 SurfaceFlinger 的总结

前面的分析让我们感受了 SurfaceFlinger 的风采。从整体上看，SurfaceFlinger 不如 AudioFlinger 复杂，它的工作集中在工作线程中，下面用图 8-23 来总线一下 SF 工作线程：

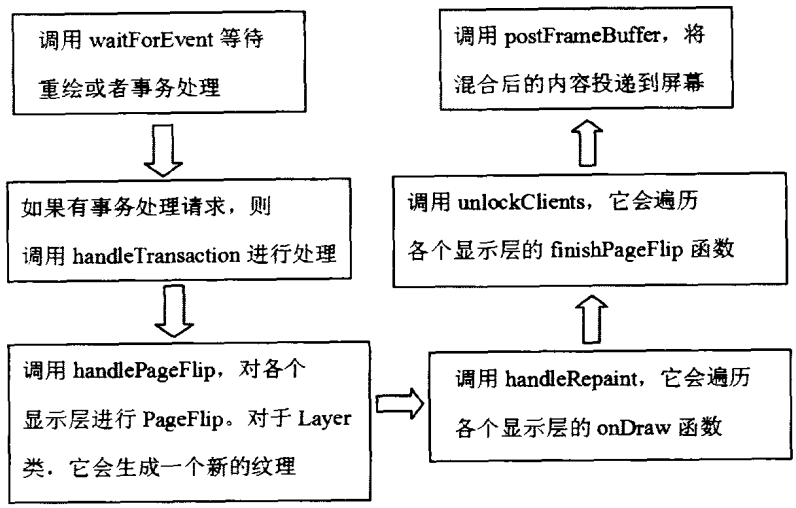


图 8-23 SF 工作线程的流程总结

8.6　拓展思考

本章的拓展思考分三个部分：

- 介绍 SharedBufferServer 和 SharedBufferClient 的工作流程。
- 关于 ViewRoot 的一些问题的总结。
- LayerBuffer 的工作原理分析。

8.6.1　Surface 系统的 CB 对象分析

根据前文的分析可知，Surface 系统中的 CB，其实是指 SharedBuffer 家族，它们是 Surface 系统中对生产者和消费者进行步调控制的中枢机构。先通过图 8-24 来观察整体的工作流程是怎样的。

为了书写方便，我们简称：

- SharedBufferServer 为 SBS。
- SharedBufferClient 为 SBC。
- SharedBufferStack 为 SBT。

其中 SBC 和 SBS 都是建立在同一个 SBT 上的，所以应先看 SBT，下面的代码列出了其中几个与读写控制有关的成员变量：

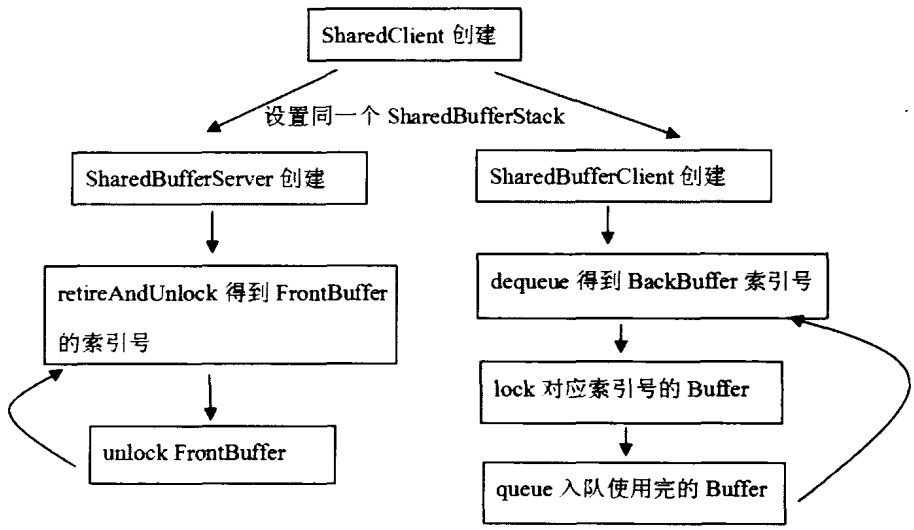


图 8-24　SharedBuffer 家族的使用流程

[-->SharedBufferStack.h]

```
class SharedBufferStack{
 ......
 /*
```

```
虽然 PageFlipping 使用 Front 和 Back 两个 Buffer 就可以了，但是 SBT 的结构和相关算法
是支持多个缓冲的。另外，缓冲是按照块来获取的，也就是一次获得一块缓冲，每块缓冲用
一个编号表示（这一点在之前的分析中已经介绍过了）。
 */
int32_t head;
int32_t available;          // 当前可用的空闲缓冲个数。
int32_t queued;             //SBC 投递的脏缓冲个数。
int32_t inUse;              //SBS 当前正在使用的缓冲编号。
......// 上面这几个参数联合 SBC 中的 tail，我称之为控制参数。
}
```

SBT 创建好后，下面就是 SBS 和 SBC 的创建了，它们会做什么特殊工作吗？

1. SBS 和 SBC 的创建

下面分别看 SBS 和 SBC 的创建，代码如下所示：

[-->SharedBufferStack.cpp]

```
SharedBufferServer::SharedBufferServer(SharedClient* sharedClient,
        int surface, int num, int32_t identity)
    : SharedBufferBase(sharedClient, surface, num, identity)
{
    mSharedStack->init(identity);// 这个函数将设置 inUse 为 -1。
    // 下面设置 SBT 中的参数，我们关注前三个。
    mSharedStack->head = num-1;
    mSharedStack->available = num;
    mSharedStack->queued = 0;
    // 设置完后，head=2-1=1,available=2,queued=0, inUse=-1
    mSharedStack->reallocMask = 0;
    memset(mSharedStack->dirtyRegion, 0, sizeof(mSharedStack->dirtyRegion));
}
```

再看 SBC 的创建，代码如下所示：

[-->SharedBufferStack.cpp]

```
SharedBufferClient::SharedBufferClient(SharedClient* sharedClient,
        int surface, int num, int32_t identity)
    : SharedBufferBase(sharedClient, surface, num, identity), tail(0)
{
    tail = computeTail(); //tail 是 SBC 定义的变量，注意它不是 SBT 定义的。
}
```

看 computeTail 函数的代码，如下所示：

[-->SharedBufferStack.cpp]

```
int32_t SharedBufferClient::computeTail() const
{
```

```
    SharedBufferStack& stack( *mSharedStack );
    int32_t newTail;
    int32_t avail;
    int32_t head;
    do {
        avail = stack.available; //available=2,head=1
        head = stack.head;
    } while (stack.available != avail);
    newTail = head - avail + 1;//newTail=1-2+1=0
    if (newTail < 0) {
        newTail += mNumBuffers;
    } else if (newTail >= mNumBuffers) {
        newTail -= mNumBuffers;
    }
    return newTail;// 计算得到 newTail=0
}
```

来看 SBC 和 SBS 创建后控制参数的变化，如图 8-25 所示：

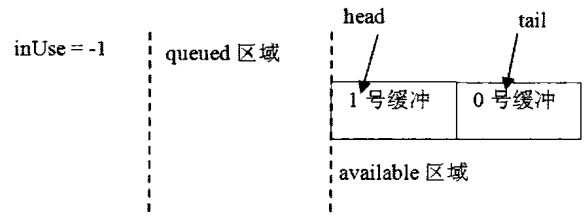


图 8-25　SBC/SBS 创建后的示意图

2.SBC 端流程分析

下面看 SBC 端的工作流程。

（1）dequeue 分析

先看 SBC 的 dequeue 函数：

[-->SharedBufferStack.cpp]

```
ssize_t SharedBufferClient::dequeue()
{
    SharedBufferStack& stack( *mSharedStack );
......
    //DequeueCondition 函数对象。
    DequeueCondition condition(this);
    status_t err = waitForCondition(condition);
    // 成功以后，available 减 1，表示当前可用的空闲 buffer 只有 1 个。
    if (android_atomic_dec(&stack.available) == 0) {
     ......
    }

    int dequeued = tail; //tail 值为 0，所以 dequeued 的值为 0。
```

```
    //tail 加 1。如果超过 2，则重新置为 0，这表明 tail 的值在 0,1 间循环。
    tail = ((tail+1 >= mNumBuffers) ? 0 : tail+1);
    ......
    // 返回的这个 dequeued 值为零，也就是 tail 加 1 操作前的旧值。这一点请读者务必注意。
    return dequeued;
}
```

其中 DequeueCondition 的操作函数很简单，代码如下所示：

```
bool SharedBufferClient::DequeueCondition::operator()() {
    return stack.available > 0;// 只要 available 大于 0 就算满足条件，第一次进来肯定满足。
}
```

用图 8-26 来表示 dequeue 的结果：

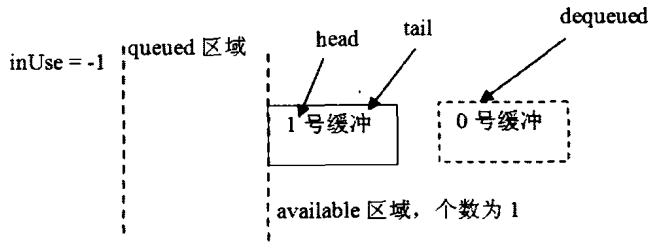


图 8-26 dequeue 结果图

注意 在图 8-26 中，0 号缓冲用虚线表示，SBC 的 dequeue 函数的返回值用 dequeued 表示，它指向这个 0 号缓冲。正如代码中注释的那样，由于 dequeued 的值用的是 tail 的旧值，而 tail 是 SBC 定义的变量，不是 SBT 定义的变量，所以 tail 在 SBS 端是不可见的。这就带来了一个潜在危险，即 0 号缓冲不能保证当前是真正空闲的，因为 SBS 可能正在用它，怎么办？试看下面的 lock。

（2）lock 分析

lock 使用了 LockCondition，其中传入的参数 buf 的值为 0，也就是上图中的 dequeue 的值，代码如下所示：

[-->SharedBufferStack.cpp]

```
status_t SharedBufferClient::lock(int buf)
{
    LockCondition condition(this, buf);
    status_t err = waitForCondition(condition);
    return err;
}
```

来看 LockCondition 的 () 函数，代码如下所示：

```
bool SharedBufferClient::LockCondition::operator()() {
```

```
    /*
     这个条件其实就是判断编号为 buf 的 Buffer 是不是被使用了。
     buf 值为 0，head 值为 1，queued 为 0，inUse 为 -1
    */
     return (buf != stack.head ||
           (stack.queued > 0 && stack.inUse != buf));
}
```

现在知道为什么 SBC 需要调用 dequeue 和 lock 函数了吗？原来：

- dequeue 只是根据本地变量 tail 计算一个本次应当使用的 Buffer 编号，其实也就是在 0,1 之间循环。上次用 0 号缓冲，那么这次就用 1 号缓冲。
- lock 函数要确保这个编号的 Buffer 没有被 SF 当作 FrontBuffer 使用。

（3）queue 分析

Activity 端在绘制完 UI 后，将把 BackBuffer 投递出去以便显示。接着看上面的流程，这个 BackBuffer 的编号是 0。待 Activity 投递完后，才会调用 signal 函数触发 SF 消费，所以在此之前格局不会发生变化。试看投递用的 queue 函数，注意传入的 buf 参数为 0，代码如下所示：

[-->SharedBufferStack.cpp]

```
status_t SharedBufferClient::queue(int buf)
{
    QueueUpdate update(this);
    status_t err = updateCondition( update );
    ......
    return err;
}
// 直接看这个 QueueUpdate 函数对象。
ssize_t SharedBufferClient::QueueUpdate::operator()() {
    android_atomic_inc(&stack.queued);//queued 增加 1，现在该值由零变为 1。
    return NO_ERROR;
}
```

至此，SBC 端走完一个流程了，结果是什么？如图 8-27 所示：

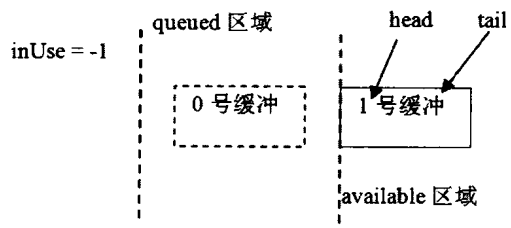


图 8-27　queue 结果图

0 号缓冲被移到了 queue 的区域，可目前还没有变量指向它。假设 SBC 端此后没有绘制 UI 的需求，那么它就会沉默一段时间。

3. SBS 端分析

SBS 的第一个函数是 retireAndLock，它使用了 RetireUpdate 函数对象，代码如下所示：

[-->SharedBufferStack.cpp]

```
ssize_t SharedBufferServer::retireAndLock()
{
    RetireUpdate update(this, mNumBuffers);
    ssize_t buf = updateCondition( update );
    return buf;
}
```

这个 RetireUpdate 对象的代码如下所示：

```
ssize_t SharedBufferServer::RetireUpdate::operator()() {
    // 先取得 head 值，为 1。
    int32_t head = stack.head;

    //inUse 被设置为 1。表明要使用 1 吗？目前的脏缓冲应该是 0 才对。
    android_atomic_write(head, &stack.inUse);

   int32_t queued;
    do {
        queued = stack.queued; //queued 目前为 1。
        if (queued == 0) {
            return NOT_ENOUGH_DATA;
        }
      // 下面这个原子操作使得 stack.queued 减 1。
    } while (android_atomic_cmpxchg(queued, queued-1, &stack.queued));
    //while 循环退出后，queued 减 1，又变为 0。
    //head 值也在 0,1 间循环，现在 head 值变为 0 了。
    head = ((head+1 >= numBuffers) ? 0 : head+1);

    //inUse 被设置为 0。
    android_atomic_write(head, &stack.inUse);

    // head 值被设为 0。
    android_atomic_write(head, &stack.head);

    // available 加 1，变成 2。
    android_atomic_inc(&stack.available);
    return head;// 返回 0。
}
```

retireAndLock 的结果是什么呢？看看图 8-28 就知道了。

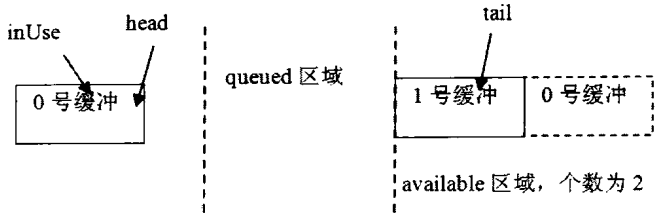


图 8-28　retireAndLock 结果图

注意上面的 available 区域，1 号缓冲右边的 0 号缓冲是用虚线表示的，这表示该 0 号缓冲实际上并不存在于 available 区域，但 available 的个数却变成 2 了。这样不会出错吗？当然不会，因为 SBC 的 lock 函数要确保这个缓冲没有被 SBS 使用。

我们来看 SBS 端的最后一个函数，它调用了 SBS 的 unlock，这个 unlock 使用了 UnlockUpdate 函数对象，就直接了解它好了，代码如下所示：

[-->SharedBufferStack.cpp]

```
ssize_t SharedBufferServer::UnlockUpdate::operator()() {
    ......
    android_atomic_write(-1, &stack.inUse);//inUse 被设置为 -1。
    return NO_ERROR;
}
```

unlock 后最终的结果是什么呢？如图 8-29 所示：

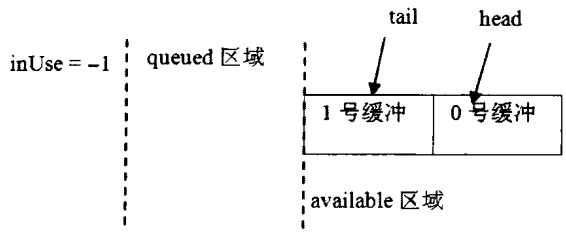


图 8-29　unlock 结果图

比较一下图 8-29 和图 8-25，可能会发现两图中 tail 和 head 刚好反了，这就是 PageFlip。另外，上面的函数大量使用了原子操作。原子操作的目的就是为了避免锁的使用。值得指出的是，updateConditon 函数和 waitForCondition 函数都使用了 Mutex，也就是说，上面这些函数对象又都是在 Mutex 锁的保护下执行的，为什么会这样呢？先来看一段代码：

```
像下面这样的代码，如果有锁控制的话根本用不着一个 while 循环，因为有锁的保护，没有其他线程
能够修改 stack.queued 的值，所以用 while 来循环判断 android_atomic_cmpxchg 没有什么意义。
    int32_t queued;
    do {
        queued = stack.queued;
        if (queued == 0) {
            return NOT_ENOUGH_DATA;
```

```
        }
    } while (android_atomic_cmpxchg(queued, queued-1, &stack.queued));
```

对于上面这个问题，我目前还不知道答案，但对其进行修改后，把函数对象放在锁外执行，结果在真机上运行没有出现任何异常现象。也许 Google 或哪位读者能给这个问题一个较好的解释。

说明 为什么我对生产/消费的同步控制如此感兴趣呢？这和自己工作的经历有些关系。因为之前曾做过一个单写多读的跨进程缓冲类，也就是一个生产者，多个消费者。为了保证正确性和一定的效率，我们在算法上曾做了很多改进，但还是大量使用了锁，所以我很好奇 Google 是怎么做到的，这也体现了一个高手的内功修养。要是由读者自己来实现，结果会怎样呢？

8.6.2 ViewRoot 的你问我答

ViewRoot 是 Surfac 系统甚至 UI 系统中一个非常关键的类，下面把网上一些关于 ViewRoot 的问题做个总结，希望这能帮助读者对 ViewRoot 有更加清楚的认识。

ViewRoot 和 View 类的关系是什么？

ViewRoot 是 View 视图体系的根。每一个 Window（注意是 Window，比如 PhoneWindow）有一个 ViewRoot，它的作用是处理 layout 和 View 视图体系的绘制工作。那么视图体系又是什么呢？它包括 Views 和 ViewGroups，也就是 SDK 中能看到的 View 类都属于视图体系。根据前面的分析可知，这些 View 是需要通过 draw 画出来的。而 ViewRoot 就是用来 draw 它们的，ViewRoot 本身没有 draw/onDraw 函数。

ViewRoot 和它所控制的 View 及其子 View 使用同一个 Canvas 吗？

这个问题的答案就很简单了，我们在 ViewRoot 的 performTraversals 中见过。ViewRoot 提供 Canvas 给它所控制的 View，所以它们使用同一个 Canvas。但 Canvas 使用的内存却不是固定的，而是通过 Surface 的 lockCanvas 得到的。

View、Surface 和 Canvas 之间的关系是怎样的？我认为，每一个 view 将和一个 Canvas，以及一个 surface 绑定到一起（这里的“我”表示提问人）。

这个问题的答案也很简单。一个 Window 将和一个 Surface 绑定在一起，绘制前 ViewRoot 会从 Surface 中 lock 出一个 Canvas。

Canvas 有一个 bitmap，那么绘制 UI 时，数据是画在 Canvas 的这个 bitmap 中吗？

答案是肯定的，bitmap 实际上包括了一块内存，绘制的数据最终都在这块内存上。

同一个 ViewRoot 下，不同类型的 View（不同类型指不同的 UI 单元，例如按钮、文本框等）使用同一个 Surface 吗？

是的，但是 SurfaceView 要除外。因为 SurfaceView 的绘制一般在单独的线程上，并且由应用层主动调用 lockCanvas、draw 和 unlockCanvasAndPost 来完成绘制流程。应用层相当

于抛开了 ViewRoot 的控制直接和屏幕打交道，这在 camera、video 方面用得最多。

8.6.3　LayerBuffer 分析

前面介绍了Normal 属性显示层中的第一类Layer，这里将介绍其中的第二类LayerBuffer。LayerBuffer 会在视频播放和摄像机预览等场景中用到，下面就以 Camera 的 preView（预览）为例，来分析 LayerBuffer 的工作原理。

1.LayerBuffer 的创建

先看 LayerBuffer 的创建，它通过 SF 的 createPushBuffersSurfaceLocked 得到，代码如下所示：

[-->SurfaceFlinger.cpp]

```
sp<LayerBaseClient> SurfaceFlinger::createPushBuffersSurfaceLocked(
        const sp<Client>& client, DisplayID display,
        int32_t id, uint32_t w, uint32_t h, uint32_t flags)
{
    sp<LayerBuffer> layer = new LayerBuffer(this, display, client, id);
    layer->initStates(w, h, flags);
    addLayer_l(layer);
    return layer;
}
```

LayerBuffer 的派生关系如图 8-30 所示：

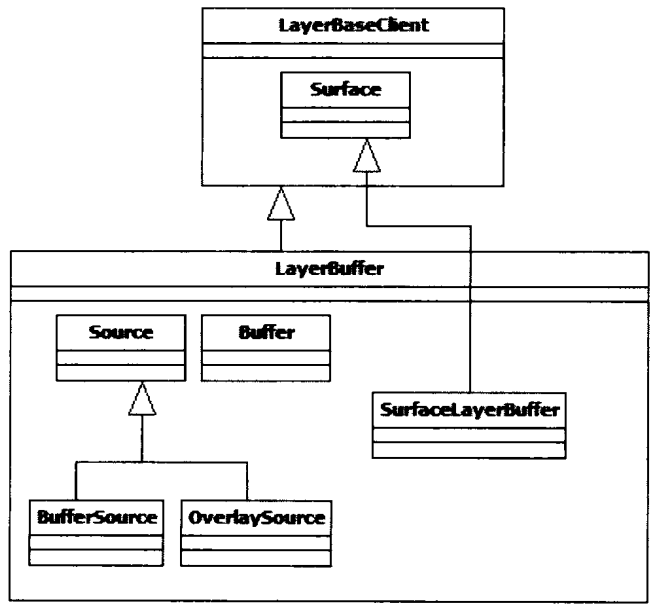


图 8-30　LayerBuffer 的派生关系示意图

从上图中可以发现：

- LayerBuffer 定义了一个内部类 Source 类，它有两个派生类 BufferSource 和 OverlaySource。根据它们的名字，可以猜测到 Source 代表数据的提供者。
- LayerBuffer 中的 mSurface 其真实类型是 SurfaceLayerBuffer。

LayerBuffer 创建好了，不过该怎么用呢？和它相关的调用流程是怎样的呢？下面来分析 Camera。

2. Camera preView 分析

Camera 是一个单独的 Service，全称是 CameraService，先看 CameraService 的 registerPreviewBuffers 函数。这个函数会做什么呢？代码如下所示：

[-->CameraService.cpp]

```
status_t CameraService::Client::registerPreviewBuffers()
{
    int w, h;
    CameraParameters params(mHardware->getParameters());
    params.getPreviewSize(&w, &h);

   /*
    ① mHardware 代表 Camera 设备的 HAL 对象。本书讨论 CameraHardwareStub 设备，它其实是
   一个虚拟的设备，不过其代码却具有参考价值。
   BufferHeap 定义为 ISurface 的内部类，其实就是对 IMemoryHeap 的封装。
   */
   ISurface::BufferHeap buffers(w, h, w, h,
                                HAL_PIXEL_FORMAT_YCrCb_420_SP,
                                mOrientation,
                                0,
                                mHardware->getPreviewHeap());
    // ②调用 SurfaceLayerBuffer 的 registerBuffers 函数。
    status_t ret = mSurface->registerBuffers(buffers);
    return ret;
}
```

上面代码中列出了两个关键点，逐一来分析它们。

（1）创建 BufferHeap

BufferHeap 是 ISurface 定义的一个内部类，它的声明如下所示：

[-->ISurface.h]

```
class BufferHeap {
   public:
      ......
       // 使用这个构造函数。
       BufferHeap(uint32_t w, uint32_t h,
              int32_t hor_stride, int32_t ver_stride,
              PixelFormat format, const sp<IMemoryHeap>& heap);
```

```
    ......
      ~BufferHeap();

      uint32_t w;
      uint32_t h;
      int32_t hor_stride;
      int32_t ver_stride;
      PixelFormat format;
      uint32_t transform;
      uint32_t flags;
      sp<IMemoryHeap> heap; //heap 指向真实的存储对象。
};
```

从上面的代码可发现，BufferHeap 基本上就是封装了一个 IMemoryHeap 对象，根据我们对 IMemoryHeap 的了解，它应该包含了真实的存储对象，这个值由 CameraHardwareStub 对象的 getPreviewHeap 得到，这个函数的代码如下所示：

[-->CameraHardwareStub.cpp]

```
sp<IMemoryHeap> CameraHardwareStub::getPreviewHeap() const
{
    return mPreviewHeap;// 返回一个成员变量，它又是在哪创建的呢?
}
// 上面的 mPreivewHeap 对象由 initHeapLocked 函数创建，该函数在 HAL 对象创建的时候被调用
void CameraHardwareStub::initHeapLocked()
{
  ......
 /*
创建一个 MemoryHeapBase 对象，大小是 mPreviewFrameSize * kBufferCount，其中
 kBufferCount 为 4。注意这是一段连续的缓冲。
*/
 mPreviewHeap = new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
 //mBuffer 为 MemoryBase 数组，元素为 4。
 for (int i = 0; i < kBufferCount; i++) {
        mBuffers[i] = new MemoryBase(mPreviewHeap,
i * mPreviewFrameSize, mPreviewFrameSize);
    }
}
```

从上面这段代码中可以发现，CameraHardwareStub 对象创建的用于 preView 的内存结构是按图 8-31 所示的方式来组织的：

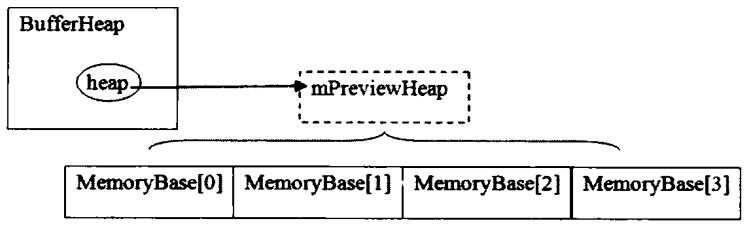


图 8-31　CameraHardwareStub 用于 preView 的内存结构图

其中：

- BufferHeap 的 heap 变量指向一块 MemoryHeap，这就是 mPreviewHeap。
- 在这块 MemoryHeap 上构建了 4 个 MemoryBase。

（2）registerBuffers 分析

BufferHeap 准备好后，要调用 ISurface 的 registerBuffers 函数，ISurface 在 SF 端的真实类型是 SurfaceLayerBuffer，所以要直接地看它的实现，代码如下所示：

[-->LayerBuffer.cpp]

```
status_t LayerBuffer::SurfaceLayerBuffer::registerBuffers(
        const ISurface::BufferHeap& buffers)
{
    sp<LayerBuffer> owner(getOwner());
    if (owner != 0)
      // 调用外部类对象的 registerBuffers，所以 SurfaceLayerBuffer 也是一个 Proxy 哦。
        return owner->registerBuffers(buffers);
    return NO_INIT;
}
// 外部类是 LayerBuffer，调用它的 registerBuffers 函数。
status_t LayerBuffer::registerBuffers(const ISurface::BufferHeap& buffers)
{
Mutex::Autolock _l(mLock);
// 创建数据的来源 BufferSource，注意我们其实把 MemoryHeap 设置上去了。
    sp<BufferSource> source = new BufferSource(*this, buffers);
    status_t result = source->getStatus();
    if (result == NO_ERROR) {
        mSource = source;// 保存这个数据源为 mSource。
    }
    return result;
}
```

BufferSource，曾在图 8-30 中见识过，它内部有一个成员变量 mBufferHeap 指向传入的 buffers 参数，所以 registerBuffers 过后，就得到了图 8-32：

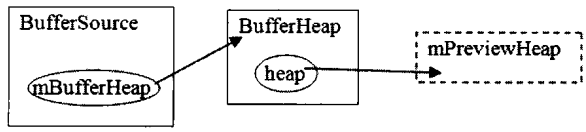


图 8-32 registerBuffers 的结果示意图

请注意上图的箭头指向，不论中间有多少层封装，最终的数据存储区域还是 mPreivewHeap。

3. 数据的传输

至此，Buffer 在 SF 和 Camera 两端都准备好了，那么数据是怎么从 Camera 传递到 SF 的呢？先来看数据源是怎么做的。

（1）数据传输分析

CameraHardwareStub 有一个 preview 线程，这个线程会做什么呢？代码如下所示：

[-->CameraHardwareStub.cpp]

```
//preview 线程从 Thread 类派生，下面这个函数在 threadLoop 中循环调用。
int CameraHardwareStub::previewThread()
{
    mLock.lock();
    // 每次进来 mCurrentPreviewFrame 都会加 1。
    ssize_t offset = mCurrentPreviewFrame * mPreviewFrameSize;

    sp<MemoryHeapBase> heap = mPreviewHeap;

    FakeCamera* fakeCamera = mFakeCamera;// 虚拟的摄像机设备。
    // 从 mBuffers 中取一块内存，用于接收来自硬件的数据。
    sp<MemoryBase> buffer = mBuffers[mCurrentPreviewFrame];

    mLock.unlock();
    if (buffer != 0) {
       int delay = (int)(1000000.0f / float(previewFrameRate));
       void *base = heap->base();//base 是 mPreviewHeap 的起始位置。

        // 下面这个 frame 代表 buffer 在 mPreviewHeap 中的起始位置，还记得图 8-31 吗?
       // 四块 MemoryBase 的起始位置由下面这个代码计算得来。
        uint8_t *frame = ((uint8_t *)base) + offset;
        // 取出一帧数据，放到对应的 MemoryBase 中。
        fakeCamera->getNextFrameAsYuv422(frame);
        // ①把含有帧数据的 buffer 传递到上层。
        if (mMsgEnabled & CAMERA_MSG_PREVIEW_FRAME)
            mDataCb(CAMERA_MSG_PREVIEW_FRAME, buffer, mCallbackCookie);

        //mCurrentPreviewFrame 递增，在 0 到 3 之间循环。
        mCurrentPreviewFrame = (mCurrentPreviewFrame + 1) % kBufferCount;
        usleep(delay);// 模拟真实硬件的延时。
    }

    return NO_ERROR;
}
```

读者是否明白 Camera preview 的工作原理了？就是从四块内存中取一块出来接收数据，然后再把这块内存传递到上层去处理。从缓冲使用的角度来看，mBuffers 数组构成了一个成员个数为四的缓冲队列。preview 通过 mData 这个回调函数，把数据传递到上层，而 CameraService 则实现了 mData 这个回调函数，这个回调函数最终会调用 handlePreviewData，直接看 handlePreviewData 即可，代码如下所示：

[-->CameraService.cpp]

```
void CameraService::Client::handlePreviewData(const sp<IMemory>& mem)
{
    ssize_t offset;
    size_t size;
    // 注意传入的 mem 参数，它实际上是 Camera HAL 创建的 mBuffers 数组中的一个。
    //offset 返回的是这个数组在 mPreviewHeap 中的偏移量。
    sp<IMemoryHeap> heap = mem->getMemory(&offset, &size);
    if (!mUseOverlay)
    {
        Mutex::Autolock surfaceLock(mSurfaceLock);
        if (mSurface != NULL) {
            // 调用 ISurface 的 postBuffer，注意我们传入的参数是 offset。
            mSurface->postBuffer(offset);
        }
}
......
}
```

上面的代码是什么意思？我们到底给 ISurface 传什么了？答案很明显：

handlePreviewData 就是传递了一个偏移量，这个偏移量是 mBuffers 数组成员的首地址。可用图 8-33 来表示：

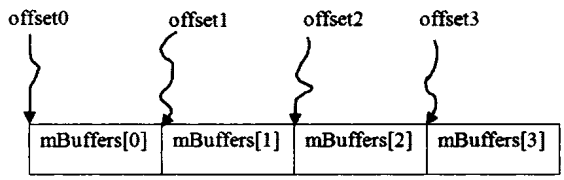


图 8-33　handlePreviewData 示意图

有了图 8-33，读者明白数据传递的工作原理了吗？

下面看 SurfaceLayerBuffer 的 postBuffer 函数，不过它只是一个小小的代理，真正的工作由外部类 LayerBuffer 完成，直接看它好了，代码如下所示：

[-->LayerBuffer.cpp]

```
void LayerBuffer::postBuffer(ssize_t offset)
{
    sp<Source> source(getSource());//getSource 返回 mSource，为 BufferSource 类型。
    if (source != 0)
        source->postBuffer(offset);// 调用 BufferSource 的 postBuffer 函数。
}
```

[-->LayerBuffer.cpp]

```
void LayerBuffer::BufferSource::postBuffer(ssize_t offset)
{
```

```
    ISurface::BufferHeap buffers;
    {
        Mutex::Autolock _l(mBufferSourceLock);
        buffers = mBufferHeap;// 还记得图 8-32 吗?
        if (buffers.heap != 0) {
            //BufferHeap 的 heap 变量指向 MemoryHeap, 下面取出它的大小。
            const size_t memorySize = buffers.heap->getSize();
           // 做一下检查, 判断这个 offset 是不是有问题。
            if ((size_t(offset) + mBufferSize) > memorySize) {
                LOGE("LayerBuffer::BufferSource::postBuffer() "
                     "invalid buffer (offset=%d, size=%d, heap-size=%d",
                     int(offset), int(mBufferSize), int(memorySize));
                return;
            }
        }
    }

    sp<Buffer> buffer;
    if (buffers.heap != 0) {
    // 创建一个 LayerBuffer::Buffer。
        buffer = new LayerBuffer::Buffer(buffers, offset, mBufferSize);
        if (buffer->getStatus() != NO_ERROR)
            buffer.clear();
        setBuffer(buffer);//setBuffer ? 我们要看看。

        //mLayer 就是外部类 LayerBuffer, 调用它的 invalidate 函数将触发 SF 的重绘。
        mLayer.invalidate();
    }
}

void LayerBuffer::BufferSource::setBuffer(
                            const sp<LayerBuffer::Buffer>& buffer)
{
   //setBuffer 函数就是简单地将 new 出来的 Buffer 设置给成员变量 mBuffer, 这么做会有问题吗?
   Mutex::Autolock _l(mBufferSourceLock);
    mBuffer = buffer; // 将新的 buffer 设置为 mBuffer, mBuffer 原来指向的那个被 delete。
}
```

从数据生产者的角度看，postBuffer 函数将不断地 new 一个 Buffer 出来，然后将它赋值给成员变量 mBuffer，也就是说，mBuffer 会不断变化。现在从缓冲的角度来思考一下这种情况的结果：

- 数据生产者有一个含四个成员的缓冲队列，也就是 mBuffers 数组。
- 而数据消费者只有一个 mBuffer。

这种情况会有什么后果呢？请记住这个问题，我们到最后再来揭示。下面先看 mBuffer 的类型 Buffer 是什么。

（2）数据使用分析

Buffer 被定义成 LayerBuffer 的内部类，代码如下所示：

[-->LayerBuffer.cpp]

```
LayerBuffer::Buffer::Buffer(const ISurface::BufferHeap& buffers,
        ssize_t offset, size_t bufferSize)
    : mBufferHeap(buffers), mSupportsCopybit(false)
{
   //注意，这个src被定义为引用，所以修改src的信息相当于修改mNativeBuffer的信息。
    NativeBuffer& src(mNativeBuffer);
    src.crop.l = 0;
    src.crop.t = 0;
    src.crop.r = buffers.w;
    src.crop.b = buffers.h;

    src.img.w       = buffers.hor_stride ?: buffers.w;
    src.img.h       = buffers.ver_stride ?: buffers.h;
    src.img.format  = buffers.format;
    //这个base将指向对应的内存起始地址。
    src.img.base    = (void*)(intptr_t(buffers.heap->base()) + offset);
    src.img.handle  = 0;
    gralloc_module_t const * module = LayerBuffer::getGrallocModule();
    //做一些处理，有兴趣的读者可以去看看。
    if (module && module->perform) {
        int err = module->perform(module,
                GRALLOC_MODULE_PERFORM_CREATE_HANDLE_FROM_BUFFER,
                buffers.heap->heapID(), bufferSize,
                offset, buffers.heap->base(),
                &src.img.handle);

        mSupportsCopybit = (err == NO_ERROR);
    }
 }
```

上面是Buffer的定义，其中最重要的就是这个mNativeBuffer了，它实际上保存了mBuffers数组成员的首地址。

下面看绘图函数，也就是LayerBuffer的onDraw函数，这个函数由SF的工作线程调用，代码如下所示：

[-->LayerBuffer.cpp]

```
void LayerBuffer::onDraw(const Region& clip) const
{
    sp<Source> source(getSource());
    if (LIKELY(source != 0)) {
        source->onDraw(clip);//source实际类型是BufferSource，我们去看看。
    } else {
        clearWithOpenGL(clip);
    }
}
void LayerBuffer::BufferSource::onDraw(const Region& clip) const
```

```
{
    sp<Buffer> ourBuffer(getBuffer());
    ......// 使用这个 Buffer，注意使用的时候没有锁控制。
    mLayer.drawWithOpenGL(clip, mTexture);// 生成一个贴图，然后绘制它。
 }
```

其中 getBuffer 函数返回 mBuffer，代码如下所示：

```
sp<LayerBuffer::Buffer> LayerBuffer::BufferSource::getBuffer() const
{
    Mutex::Autolock _l(mBufferSourceLock);
    return mBuffer;
}
```

从上面的代码中能发现，mBuffer 的使用并没有锁的控制，这会导致什么问题发生呢？请再次回到前面曾强调要记住的那个问题上。此时生产者的队列有四个元素，而消费者的队列只有一个元素，它可用图 8-34 来表示：

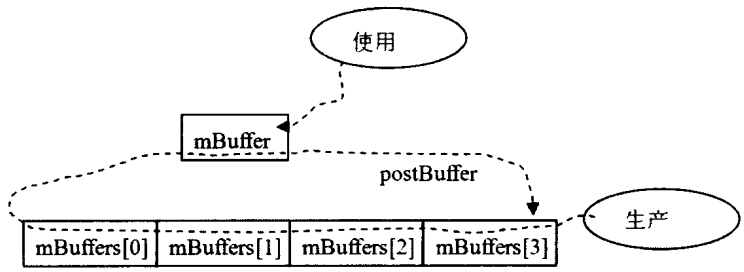


图 8-34　数据传递的问题示意图

从上图可以知道：

- 使用者使用 mBuffer，这是在 SF 的工作线程中做到的。假设 mBuffer 实际指向的内存为 mBuffers[0]。
- 数据生产者循环更新 mBuffers 数组各个成员的数据内容，这是在另外一个线程中完成的。由于这两个线程之间没有锁同步，这就造成了当使用者还在使用 mBuffers[0] 时，生产者又更新了 mBuffers[0]。这会在屏幕上产生混杂的图像。

经过实际测试得知，如果给数据使用端加上一定延时，屏幕就会出现不连续的画面，即前一帧和后一帧的数据混杂在一起输出。

说明　从代码的分析来看，这种方式确实有问题。我在真实设备上测试的结果也在一定程度上验证了这一点。通过修改 LayerBuffer 来解决这问题的难度比较大，是否可在读写具体缓存时加上同步控制呢（例如使用 mBuffers[0] 的时候调用一下 lock，用完后调用 unlock）？这样就不用修改 LayerBuffer 了。读者可再深入研究这个问题。

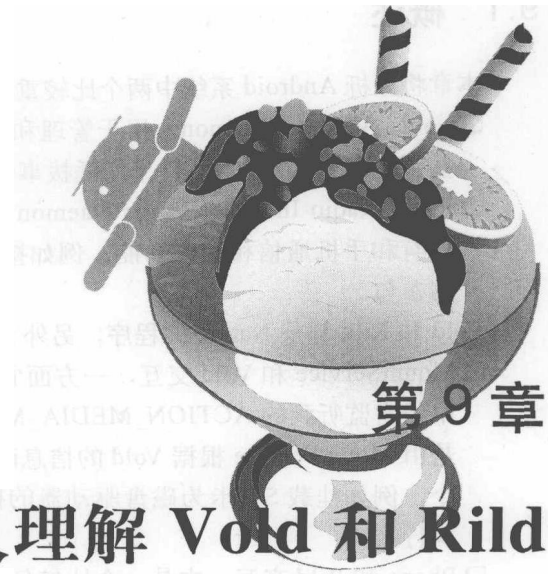
8.7 本章小结

本章可能是全书难度最大的一章了。在这一章的讲解中，我们把打通任督二脉做为破解 Surface 系统的突破口：

- 应用程序和 Surface 的关系，这是任脉。
- Surface 和 SurfaceFlinger 的关系，这是督脉。

其中，打通任脉的过程是比较曲折的，从应用程序的 Activity 开始，一路追踪到 ViewRoot、WindowManagerService。任脉被打通后，还只是解决了 Java 层的问题，而督脉则集中在 Native 层。在必杀技 aidl 工具的帮助下，我们首先成功找到了 Surface 乾坤大挪移的踪迹。此后在精简流程方法的帮助下，乘胜追击，对 Surface 及 SurfaceFlinger 进行了深入分析。我希望读者在阅读过程中，也要把握流程，这样就不至于迷失在代码中了。

在拓展部分，对 Surface 系统中 CB 对象的工作流程、ViewRoot 的一些问题，以及 LayerBuffer 进行了较为详细的介绍。



第9章 深入理解 Vold 和 Rild

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及其位置。

- Main.cpp(*system/vold/Main.cpp*)
- NetlinkManager.cpp(*system/vold/NetlinkManager.cpp*)
- NetlinkManager.h(*system/vold/NetlinkManager.h*)
- NetlinkHandler.cpp(*system/vold/NetlinkHandler.cpp*)
- NetlinkListener.cpp(*system/core/libsysutils/src/NetlinkListener.cpp*)
- SocketListener.cpp(*system/core/libsysutils/src/SocketListener.cpp*)
- VolumeManager.cpp(*system/vold/VolumeManager.cpp*)
- DirectVolume.cpp(*system/vold/DirectVolume.cpp*)
- FrameworkListener.cpp(*system/core/libsysutils/src/FrameworkListener.cpp*)
- MountService.java(*framework/base/services/java/com/android/server/MountService.java*)
- Rild.c(*hardware/ril/rild/Rild.c*)
- Ril.cpp(*hardware/ril/libril/Ril.cpp*)
- Ril_event.h(*hardware/ril/lilbril/Ril_event.h*)
- Reference_ril.c(*hardware/ril/reference_ril/Reference_ril.c*)
- Atchannle.c(*hardware/ril/reference_ril/Atchannle.c*)
- Ril.h(*hardware/ril/include/telephony/Ril.h*)
- PhoneApp.java(*package/apps/Phone/src/com/android/phone/PhoneApp.java*)
- PhoneFactory.java(*framework/base/telephony/java/com/android/internal/telephony/PhoneFactory.java*)
- RIL.java(*framework/base/telephony/java/com/android/internal/telephony/RIL.java*)
- PhoneUtils.java(*package/apps/Phone/src/com/android/phone/PhoneUtils.java*)

9.1 概述

本章将分析 Android 系统中两个比较重要的程序，它们分别是：

- Vold：Volume Daemon，用于管理和控制 Android 平台外部存储设备的后台进程，这些管理和控制，包括 SD 卡的插拔事件检测、SD 卡挂载、卸载、格式化等。
- Rild：Radio Interface Layer Daemon，用于智能手机的通信管理和控制的后台进程，所有和手机通信相关的功能，例如接打电话、收发短信 / 彩信、GPRS 等都需要 Rild 的参与。

Vold 和 Rild 都是 Native 的程序，另外 Java 世界还有和它们交互的模块，它们分别是：

- MountService 和 Vold 交互，一方面它可以接收来自 Vold 的消息，例如，在应用程序中经常监听到的 ACTION_MEDIA_MOUNTED/ACTION_MEDIA_EJECT 等广播，就是由 MountService 根据 Vold 的信息而触发的。另一方面，它可以向 Vold 发送控制命令，例如挂载 SD 卡为磁盘驱动器的操作，就是由 MountService 发送命令给 Vold 来执行的。
- Phone 和 Rild 交互，它是一个比较复杂的应用程序。简单来说，Phone 拨打电话时需要发送对应的命令给 Rild 来执行。后面在 Rild 的实例分析中会有相关介绍。

这两个 Daemon 代码的结构都不算太复杂。本章将和大家一起来领略一下它们的风采。

9.2 Vold 的原理与机制分析

Vold 是 Volume Daemon 的缩写，它是 Android 平台中外部存储系统的管控中心，是一个比较重要的进程。虽然它的地位很重要，但其代码结构却远没有前面的 Audio 和 Surface 系统复杂。欣赏完 Audio 和 Surface 的大气磅礴后，再来感受一下 Vold 的小巧玲珑也会别有一番情趣。Vold 的架构可用图 9-1 来表示：

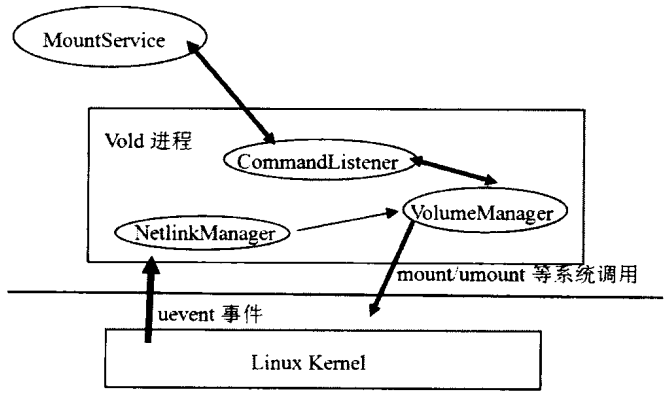


图 9-1 Vold 架构图

从上图中可知：

- Vold 中的 NetlinkManager 模块（简称 NM）接收来自 Linux 内核的 uevent 消息。例如 SD 卡的插拔等动作都会引起 Kernel 向 NM 发送 uevent 消息。
- NM 将这些消息转发给 VolumeManager 模块（简称 VM）。VM 会对应做一些操作，然后把相关信息通过 CommandListener（简称 CL）发送给 MountService，MountService 根据收到的消息会发送相关的处理命令给 VM 做进一步的处理。例如待 SD 卡插入后，VM 会将来自 NM 的“Disk Insert”消息发送给 MountService，而后 MountService 则发送“Mount”指令给 Vold，指示它挂载这个 SD 卡。
- CL 模块内部封装了一个 Socket 用于跨进程通信。它在 Vold 进程中属于监听端（即服务端），而它的连接端（即客户端）则是 MountService。它一方面接收来自 MountService 的控制命令（例如卸载存储卡、格式化存储卡等），另一方面 VM 和 NM 模块又会通过它将一些信息发送给 MountService。

相比于 Audio 和 Surface 系统，Vold 的架构确实比较简单，并且 Vold 和 MountService 所在的进程（这个进程其实就是 system_server）在进行进程间通信时，也没有利用 Binder 机制，而是直接使用了 Socket，这样，在代码量和程序中类的派生关系上就会简单不少。

9.2.1　Netlink 和 Uevent 介绍

在分析 Vold 的代码前，先介绍一下 Linux 系统中的 Netlink 和 Uevent。

1. Netlink 介绍

Netlink 是 Linux 系统中用户空间进程和 Kernel 进行通信的一种机制，通过这种机制，位于用户空间的进程可以接收来自 Kernel 的一些信息（例如 Vold 中用到的 USB 或 SD 的插拔消息），同时应用层也可通过 Netlink 向 Kernel 发送一些控制命令。

目前，Linux 系统并没有为 Netlink 单独设计一套系统调用，而是复用了 Socket 的操作接口，只是在创建 Socket 时会有一些特殊的地方。Netlink 的具体使用方法在进行代码分析时再来了解，读者目前只需知道，通过 Netlink 机制应用层，可接收来自 Kernel 的消息即可。

2.Uevent 介绍

Uevent 和 Linux 的 Udev 设备文件系统及设备模型有关系，它实际上就是一串字符串，字符串的内容可告知发生了什么事情。下面通过一个实例来直观感受 Uevent。

在 SD 卡插入手机后（我们这里以 SD 卡为例），系统会检测到这个设备的插入，然后内核会通过 Netlink 发送一个消息给 Vold，Vold 将根据接收到的消息进行处理，例如挂载这个 SD 卡。内核发送的这个消息，就是 Uevent，其中 U 代表 User space（应用层空间）。下面看 SD 卡插入时 Vold 截获到的 Uevent 消息。在我的 G7 手机上，Uevent 的内容如下，注意，其中 // 号或 /**/ 号中的内容是为方便读者理解而加的注释：

[-->SD 卡插入的 Uevent 消息]

```
//mmc 表示 MultiMedia Card，这里统称为 SD 卡。
add@/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0
ACTION=add //add 表示设备插入，另外还有 remove 和 change 等动作。
//DEVPATH 表示该设备位于 /sys 目录中的设备路径。
DEVPATH=/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0
/*
SUBSYSTEM 表示该设备属于哪一类设备，block 为块设备，磁盘也属于这一类设备，另外还有
character（字符）设备等类型。
*/
SUBSYSTEM=block
MAJOR=179//MAJOR 和 MINOR 分别表示该设备的主次设备号，二者联合起来可以标识一个设备。
MINOR=0
DEVNAME=mmcblk0
DEVTYPE=disk// 设备 Type 为 disk。
NPARTS=3   // 这个表示该 SD 卡上的分区，我的 SD 卡上有三块分区。
SEQNUM=1357// 序号
```

由于我的 SD 卡上有分区，所以还会接收到和分区相关的 Uevent。简单看一下：

[-->SD 卡插入后和分区相关的 Uevent 消息]

```
add@/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0/mmcbl k0p1
ACTION=add

// 比上面那个 DEVPATH 多了一个 mmcblk0p1。
DEVPATH=/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0/mmcblk0p1
SUBSYSTEM=block
MAJOR=179
MINOR=1
DEVNAME=mmcblk0p1
DEVTYPE=partition // 设备类型变为 partition，表示分区。
PARTN=1
SEQNUM=1358
```

通过上面的实例，我们和 Uevent 来了一次亲密接触，具体到 Vold，也就是内核通过 Uevent 告知外部存储系统发生了哪些事情，那么 Uevent 在什么情况下会由 Kernel 发出呢？

- 当设备发生变化时，这会引起 Kernel 发送 Uevent 消息，例如设备的插入和拔出等。如果 Vold 在设备发生变化之前已经建立了 Netlink IPC 通信，那么 Vold 可以接收到这些 Uevent 消息。这种情况是由设备发生变化而触发的。
- 设备一般在 /sys 对应的目录下有一个叫 uevent 的文件，往该文件中写入指定的数据，也会触发 Kernel 发送和该设备相关的 Uevent 消息，这是由应用层触发的。例如 Vold 启动时，会往这些 uevent 文件中写数据，通过这种方式促使内核发送 Uevent 消息，这样 Vold 就能得到这些设备的当前信息了。

根据上面的介绍可知，Netlink 和 Uevent 的目的，就是让 Vold 随时获悉外部存储系统

的信息，这至关重要。我们总不会希望发生诸如 SD 卡都被拔了，而 Vold 却一无所知的情况吧？

9.2.2　初识 Vold

下面来认识一下 Vold，它的代码在 main.cpp 中，如下所示：

[-->Main.cpp]

```
int main() {

    VolumeManager *vm;
    CommandListener *cl;
    NetlinkManager *nm;

    SLOGI("Vold 2.1 (the revenge) firing up");
    // 创建文件夹 /dev/block/vold。
    mkdir("/dev/block/vold", 0755);

    // ①创建 VolumeManager 对象。
    if (!(vm = VolumeManager::Instance())) {
        SLOGE("Unable to create VolumeManager");
        exit(1);
    };
   // ②创建 NetlinkManager 对象。
    if (!(nm = NetlinkManager::Instance())) {
        SLOGE("Unable to create NetlinkManager");
        exit(1);
    };

    // ③创建 CommandListener 对象。
    cl = new CommandListener();

    vm->setBroadcaster((SocketListener *) cl);
    nm->setBroadcaster((SocketListener *) cl);
    // ④启动 VM。
    if (vm->start()) {
        ......
        exit(1);
    }
   // ⑤根据配置文件来初始化 VM。
    if (process_config(vm)) {
        ......
    }
   // ⑥启动 NM。
    if (nm->start()) {
        ......
        exit(1);
    }
```

```
// 通过往 /sys/block 目录下对应的 uevent 文件写 "add\n" 来触发内核发送 Uevent 消息。
     coldboot("/sys/block");
    {
        FILE *fp;
        char state[255];
       /*

          Android 支持将手机上的外部存储设备作为磁盘挂载到电脑上。下面的代码可查看是否打开了
         磁盘挂载功能。这里涉及 UMS（USB Mass Storage，USB 大容量存储）方面的知识。*/
        if ((fp = fopen("/sys/devices/virtual/switch/usb_mass_storage/state",
                        "r"))) {
            if (fgets(state, sizeof(state), fp)) {
                if (!strncmp(state, "online", 6)) {
                   // ⑦ VM 通过 CL 向感兴趣的模块（如 MountService）通知 UMS 的状态。
                    vm->notifyUmsConnected(true);
                } else {
                    vm->notifyUmsConnected(false);
                }
            }
            ......
            fclose(fp);
        }
    ......
    }
      ......
  // ⑧启动 CL。
   if (cl->startListener()) {
        ......
        exit(1);
    }
   // 无限循环。
    while(1) {
        sleep(1000);
    }

    SLOGI("Vold exiting");
    exit(0);
}
```

上面的代码中列出了 8 个关键点（即①～⑧）。由于 Vold 将其功能合理地分配到了各个模块中，所以这 8 个关键点将放到图 9-1 所示的 Vold 的三个模块中去讨论。

下面看第一个模块 NetlinkManager，简称 NM。

9.2.3 NetlinkManager 模块分析

在 Vold 代码中，使用 NM 模块的流程是：

- 调用 Instance 创建一个 NM 对象。

❑ 调用 setBroadcaster 设置 CL 对象。

❑ 调用 start 启动 NM。

接下来，按这三个步骤来分析 NM 模块。

1. 创建 NM

Vold 调用 Instance 函数创建了一个 NM 对象。看到 Instance 这个函数，读者应能想到，这里可能是采用了单例模式。来看是否如此，代码如下所示。

[-->NetlinkManager.cpp]

```
NetlinkManager *NetlinkManager::Instance() {
    if (!sInstance)
        sInstance = new NetlinkManager();// 果然是单例模式。
    return sInstance;
}
```

NM 的创建真是非常简单。再看第二个被调用的函数 setBroadcaster。

2. setBroadcaster 分析

setBroadcaster 就更简单了，它的实现在 NetlinkManager 类的声明中，如下所示：

[-->NetlinkManager.h]

```
void setBroadcaster(SocketListener *sl) { mBroadcaster = sl; }
```

setBroadcaster 参数中的那个 sl 其实际类型为 CommandListener。需要说明的是，虽然 NM 设置了 CL 对象，但 Vold 的 NM 并没有通过 CL 发送消息和接收命令，所以在图 9-1 中，NM 模块和 CL 模块并没有连接线，这一点务必注意。

下面看最后一个函数 start。

3. start 分析

前面说过，NM 模块将使用 Netlink 和 Kernel 进行 IPC 通信，那么它是怎么做到的呢？来看代码，如下所示：

[-->NetlinkManager.cpp]

```
int NetlinkManager::start() {
   //PF_NETLINK 使用的 socket 地址结构是 sockaddr_nl，而不是一般的 sockaddr_in。
    struct sockaddr_nl nladdr;
    int sz = 64 * 1024;

    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    nladdr.nl_pid = getpid(); // 设置自己的进程 pid。
    nladdr.nl_groups = 0xffffffff;
```

```
    /*
    创建 PF_NETLINK 地址簇的 socket，目前只支持 SOCK_DGRAM 类型，第三个参数。
    NETLINK_KOBJECT_UEVENT 表示要接收内核的 Uevent 事件。
       */
        if ((mSock = socket(PF_NETLINK,
                            SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
            ......
            return -1;
        }
        // 设置 Socket 接收缓冲区大小。
        if (setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz)) < 0) {
            ......
            return -1;
        }
        // 必须对该 socket 执行 bind 操作。
        if (bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
            ......
            return -1;
        }
       // 创建一个 NetlinkHandler 对象，并把创建好的 Socket 句柄传给它。
       mHandler = new NetlinkHandler(mSock);

       // 调用 NetlinkHandler 对象的 start。
        if (mHandler->start()) {
            SLOGE("Unable to start NetlinkHandler: %s", strerror(errno));
            return -1;
        }
        return 0;
    }
```

从代码上看，NM 的 start 函数分为两个步骤：

- 创建地址簇为 PF_NETLINK 类型的 socket 并做一些设置，这样 NM 就能和 Kernel 通信了。关于 Netlink 的使用技巧网上有很多资料，读者可在 Linux 系统上通过 man netlink 命令来查询相关信息。
- 创建 NetlinkHandler 对象，并调用它的 start。看来，后续工作都是由 NetlinkHandler 完成的。

根据上文的分析可看出，NetlinkHandler 才是真正的主角，下面就来分析它。为了书写方便，我们将 NetlinkHandler 简称为 NLH。

4. NetlinkHandler 分析

（1）创建 NLH

在代码结构简单的 Vold 程序中，NetlinkHandler 有一个相对不简单的派生关系，如图 9-2 所示：

直接看代码，来认识这个 NLH：

[-->NetlinkHandler.cpp]

```
NetlinkHandler::NetlinkHandler(int listenerSocket) :
                NetlinkListener(listenerSocket) {
 // 调用基类 NetlinkListener 的构造函数。注意传入的参数是和 Kernel 通信的 socket。
 // 句柄。注意，文件描述符和句柄表示的是同一个东西，这里不再区分二者。
}
```

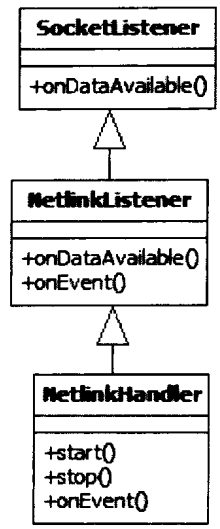


图 9-2　NLH 的派生关系图

再看基类 NetlinkListener 的构造函数，如下所示：

[-->NetlinkListener.cpp]

```
NetlinkListener::NetlinkListener(int socket) :
                    SocketListener(socket, false) {
 // 调用基类 SocketListener 的构造函数，第二个参数为 false。
}
```

基类 SocketListener 的构造函数是：

[-->SocketListener.cpp]

```
SocketListener::SocketListener(int socketFd, bool listen) {
    mListen = listen; // 这个参数是 false。
    mSocketName = NULL;
    mSock = socketFd;// 保存和 Kernel 通信的 socket 描述符。
    // 初始化一个 mutex，看来会有多个线程存在。
    pthread_mutex_init(&mClientsLock, NULL);
    /*
    SocketClientCollection 的声明如下，它是一个列表容器。
    typedef android::List<SocketClient *> SocketClientCollection
```

```
        其中，SocketClient 代表和 Socket 服务端通信的客户端。
        */
          mClients = new SocketClientCollection();
    }
```

NLH 的创建分析完了。此过程中没有什么新鲜内容。下面看它的 start 函数。

注意 本章内容会大量涉及 Socket，所以读者应先了解与 Socket 有关的知识，如果需要深入研究，建议阅读《Unix Networking Programming Volume I》一书。

（2）start 分析

在分析前面的代码时，曾看到 NetlinkHandler 会创建一个同步互斥对象，这表明 NLH 会在多线程环境中使用，那么这个线程会在哪里创建呢？来看 start 的代码，如下所示：

[-->NetlinkHandler.cpp]

```
int NetlinkHandler::start() {
    return this->startListener();//startListener 由 SocketListener 实现。
}
```

[-->SocketListener.cpp]

```
int SocketListener::startListener() {

    if (!mSocketName && mSock == -1) {
        errno = EINVAL;
        return -1;
    } else if (mSocketName) {
        if ((mSock = android_get_control_socket(mSocketName)) < 0) {
            return -1;
        }
    }
/*
还记得构造 NLH 时的参数吗？ mListen 为 false，这表明 NLH 不是监听（listen）端。
这里为了代码和操作的统一，用 mSock 做参数构造了一个 SocketClient 对象，
并加入到 mClients 列表中了，但这个 SocketClient 并不是真实客户端的代表。
*/
    if (mListen && listen(mSock, 4) < 0) {
        ......
        return -1;
} else if (!mListen)// 以 mSock 为参数构造 SocketClient 对象，并加入到对应的列表中。
        mClients->push_back(new SocketClient(mSock));
    /*
    pipe 系统调用将创建一个匿名管道，mCtrlPipe 是一个 int 类型的二元数组。
    其中 mCtrlPipe[0] 用于从管道读数据，mCtrlPipe[1] 用于往管道写数据。
    */
    if (pipe(mCtrlPipe)) {
        ......
```

```
        return -1;
    }
    // 创建一个工作线程，线程函数是 threadStart。
    if (pthread_create(&mThread, NULL, SocketListener::threadStart, this)) {
        ......
        return -1;
    }

    return 0;
}
```

如果熟悉 Socket 编程，理解上面的代码就非常容易了。下面来看 NLH 的工作线程。

（3）工作线程分析

工作线程的线程函数 threadStart 的代码如下所示：

[-->SocketListener.cpp]

```
void *SocketListener::threadStart(void *obj) {
    SocketListener *me = reinterpret_cast<SocketListener *>(obj);

    me->runListener();// 调用 runListener。
    pthread_exit(NULL);
    return NULL;
}
// 直接分析 runListener。
void SocketListener::runListener() {

    while(1) {
        SocketClientCollection::iterator it;
        fd_set read_fds;
        int rc = 0;
        int max = 0;

        FD_ZERO(&read_fds);

        if (mListen) {//mListen 为 false，所以不走这个 if 分支。
            max = mSock;
            FD_SET(mSock, &read_fds);
        }
       /*
        计算 max，为什么要有这个操作？这是由 select 函数决定的，它的第一个参数的取值。
        必须为它所监视的文件描述符集合中最大的文件描述符加 1。
       */
        FD_SET(mCtrlPipe[0], &read_fds);
        if (mCtrlPipe[0] > max)
            max = mCtrlPipe[0];
        // 还是计算 fd 值最大的那个。
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
```

```
        FD_SET((*it)->getSocket(), &read_fds);
        if ((*it)->getSocket() > max)
            max = (*it)->getSocket();
    }
    pthread_mutex_unlock(&mClientsLock);
    /*
     注意 select 函数的第一个参数，为 max+1。读者可以通过 man select 来查询
      select 的用法。注意，在 Windows 平台上的 select 对第一个参数没有要求。
    */
    if ((rc = select(max + 1, &read_fds, NULL, NULL, NULL)) < 0) {
        sleep(1);
        continue;
    } else if (!rc)
        continue;
    // 如果管道可读的话，表示需要退出工作线程。
    if (FD_ISSET(mCtrlPipe[0], &read_fds))
        break;
    if (mListen && FD_ISSET(mSock, &read_fds)) {
        // 如果是 listen 端的话，mSock 可读表示有客户端 connect 上。
        struct sockaddr addr;
        socklen_t alen = sizeof(addr);
        int c;
         // 调用 accept 接受客户端的连接，返回用于和客户端通信的 Socket 描述符。
        if ((c = accept(mSock, &addr, &alen)) < 0) {
            SLOGE("accept failed (%s)", strerror(errno));
            sleep(1);
            continue;
        }
        pthread_mutex_lock(&mClientsLock);
        // 根据返回的客户端 Socket 描述符构造一个 SocketClient 对象，并加入到对应的 list 中。
        mClients->push_back(new SocketClient(c));
        pthread_mutex_unlock(&mClientsLock);
    }

    do {
        pthread_mutex_lock(&mClientsLock);
        for (it = mClients->begin(); it != mClients->end(); ++it) {
            int fd = (*it)->getSocket();
            if (FD_ISSET(fd, &read_fds)) {
                pthread_mutex_unlock(&mClientsLock);
             /*
              有数据通过 Socket 发送过来，所以调用 onDataAvailable 进行处理。
              如果在 onDataAvailable 返回 false,, 则表示需要关闭该连接。
             */
                if (!onDataAvailable(*it)) {
                    close(fd);
                    pthread_mutex_lock(&mClientsLock);
                    delete *it;
                    it = mClients->erase(it);
                    pthread_mutex_unlock(&mClientsLock);
                }
                FD_CLR(fd, &read_fds);
```

```
                continue;
            }
        }
        pthread_mutex_unlock(&mClientsLock);
    } while (0);
  }
}
```

从代码中可看到：

- 工作线程退出的条件是匿名管道可读，但在一般情况下它不需要退出，所以可以忽略此项内容。
- 不论是服务端还是客户端，收到数据后都会调用 onDataAvailable 进行处理。

下面就来看 NLH 的数据处理。

（4）数据处理

根据前面的分析可知，收到数据后首先会调用 onDataAvailable 函数进行处理，这个函数由 NLH 的基类 NetlinkListener 实现。代码如下所示：

[-->NetlinkListener]

```
bool NetlinkListener::onDataAvailable(SocketClient *cli)
{
    int socket = cli->getSocket();
    int count;

    /*
    调用 recev 接收数据，如果接收错误，则返回 false，这样这个 socket 在
    上面的工作线程中就会被 close。
    */
    if ((count = recv(socket, mBuffer, sizeof(mBuffer), 0)) < 0) {
        SLOGE("recv failed (%s)", strerror(errno));
        return false;
    }
   //new 一个 NetlinkEvent，并调用 decode 来解析接收到的 Uevent 数据。
    NetlinkEvent *evt = new NetlinkEvent();
    if (!evt->decode(mBuffer, count)) {
         goto out;
    }
    // 调用 onEvent，并传递 NetlinkEvent 对象。
    onEvent(evt);
out:
    delete evt;
return true;
}
```

decode 函数就是将收到的 Uevent 信息填充到一个 NetlinkEvent 对象中，例如 Action 是什么，SUBSYSTEM 是什么等，以后处理 Uevent 时就不用再解析字符串了来。

来看 onEvent 函数，此函数是由 NLH 自己实现的，代码如下所示：

[-->NetlinkHandler.cpp]

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();

    if (!subsys) {
        return;
    }

if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt); // 调用 VM 的 handleBlockEvent。
    } else if (!strcmp(subsys, "switch")) {
        vm->handleSwitchEvent(evt);// 调用 VM 的 handleSwitchEvent。
} else if (!strcmp(subsys, "battery")) {
      // 这两个事件和外部存储系统没有关系，所以不处理。
    } else if (!strcmp(subsys, "power_supply")) {
    }
}
```

NLH 的工作已介绍完，下面总结一下 NM 模块的工作。

5. 关于 NM 模块的总结

NM 模块的功能就是从 Kernel 接收 Uevent 消息，然后转换成一个 NetlinkEvent 对象，最后会调用 VM 的处理函数来处理这个 NetlinkEvent 对象。

9.2.4 VolumeManager 模块分析

Vold 使用 VM 模块的流程是：

- 调用 Instance 创建一个 VM 对象。
- 调用 setBroadcaster 设置 CL 对象，这个函数和 NM 的 setBroadcaster 一样，所以本节不再介绍它。
- 调用 start 启动 VM。
- 调用 process_config 配置 VM。

现在来看除 setBroadcaster 之外的三个函数。

1. 创建 VM 和对 start 的分析

VM 的创建及 start 函数都非常简单，代码如下所示。

[-->VolumeManager.cpp]

```
VolumeManager *VolumeManager::Instance() {
    if (!sInstance)
        sInstance = new VolumeManager();
    return sInstance;
```

```
}
```

可以看到，VM 也采用了单例模式，所以全进程只会存在一个 VM 对象。

下面看 VM 的 start 函数，如下所示：

[-->VolumeManager.cpp]

```
int VolumeManager::start() {
    return 0;
}
```

start 很简单，没有任何操作。

2. process_config 分析

process_config 函数会根据配置文件配置 VM 对象，其代码如下所示：

[-->Main.cpp]

```
static int process_config(VolumeManager *vm) {
    FILE *fp;
    int n = 0;
    char line[255];
   // 读取 /etc/vold.fstab 文件。
    if (!(fp = fopen("/etc/vold.fstab", "r"))) {
        return -1;
    }

    while(fgets(line, sizeof(line), fp)) {
        char *next = line;
        char *type, *label, *mount_point;

        n++;
        line[strlen(line)-1] = '\0';

        if (line[0] == '#' || line[0] == '\0')
            continue;

        if (!(type = strsep(&next, " \t"))) {
            goto out_syntax;
        }
        if (!(label = strsep(&next, " \t"))) {
            goto out_syntax;
        }
        if (!(mount_point = strsep(&next, " \t"))) {
            goto out_syntax;
        }

        if (!strcmp(type, "dev_mount")) {
            DirectVolume *dv = NULL;
            char *part, *sysfs_path;
```

```
            if (!(part = strsep(&next, " \t"))) {
                ......
                goto out_syntax;
            }
            if (strcmp(part, "auto") && atoi(part) == 0) {
                goto out_syntax;
            }

            if (!strcmp(part, "auto")) {
               // ①构造一个 DirectVolume 对象。
                dv = new DirectVolume(vm, label, mount_point, -1);
            } else {
                dv = new DirectVolume(vm, label, mount_point, atoi(part));
            }

            while((sysfs_path = strsep(&next, " \t"))) {
                // ② 添加设备路径。
                if (dv->addPath(sysfs_path)) {
                    ......
                    goto out_fail;
                }
            }
           // 为 VolumeManager 对象增加一个 DirectVolume 对象。
            vm->addVolume(dv);
        }
        ......
    }
     ......
    return -1;
}
```

从上面的代码可发现，process_config 的主要功能就是解析 /etc/vold.fstab。这个文件的作用和 Linux 系统中的 fstab 文件很类似，就是设置一些存储设备的挂载点，我的 HTC G7 手机上这个文件的内容如图 9-3 所示：

```
## Vold 2.0 fstab for HTC Passion
#
## - San Mehat (san@android.com)
##

########################
## Regular device mount
##
## Format: dev_mount <label> <mount_point> <part> <sysfs_path1...>
## label        - Label for the volume
## mount_point  - Where the volume will be mounted
## part         - Partition # (1 based), or 'auto' for first usable partition.
## <sysfs_path> - List of sysfs paths to source devices
########################

# Mounts the first usable partition of the specified device
#dev_mount sdcard /mnt/sdcard auto /devices/platform/goldfish_mmc.0 /devices/platform/msm_sdcc.4/mmc_host/mmc2
dev_mount sdcard /mnt/sdcard 1 /devices/platform/goldfish_mmc.0 /devices/platform/msm_sdcc.2/mmc_host/mmc
```

注意黑框中的内容

图 9-3　我手机上的 vold.fstab 内容

从上图灰色框中的内容可知：

- sdcard 为 volume 的名字。
- /mnt/sdcard 表示 mount 的位置。
- 1 表示使用存储卡上的第一个分区，auto 表示没有分区。现在有很多定制的 ROM 要求 SD 卡上存在多个分区。
- /devices/xxxx 等内容表示 MMC 设备在 sysfs 中的位置。

根据 G7 的 vold.fstab 文件可以构造一个 DirectVolume 对象。

注意　根据手机刷的 ROM 的不同，vold.fstab 文件会有较大差异。

3.DirectVolume 分析

DirectVolume 从 Volume 类派生，可把它看成是一个外部存储卡（例如一张 SD 卡）在代码中的代表。它封装了对外部存储卡的操作，例如加载 / 卸载存储卡、格式化存储卡等。

下面是 process_config 函数中和 DirectVolume 相关的地方：

- 一个是创建 DirectVolume。
- 另一个是调用 DirectVolume 的 addpath 函数。

它们的代码如下所示：

[-->DirectVolume.cpp]

```
DirectVolume::DirectVolume(VolumeManager *vm, const char *label,
                           const char *mount_point, int partIdx) :
                Volume(vm, label, mount_point) {// 初始化基类。
    /*
      注意其中的参数：
      label 为" sdcard" ,mount_point 为" /mnt/sdcard" ,partIdx 为 1
     */
     mPartIdx = partIdx;
     //PathCollection 定义为 typedef android::List<char *> PathCollection
     // 其实就是一个字符串 list。
     mPaths = new PathCollection();
     for (int i = 0; i < MAX_PARTITIONS; i++)
         mPartMinors[i] = -1;
     mPendingPartMap = 0;
     mDiskMajor = -1;  // 存储设备的主设备号。
     mDiskMinor = -1;  // 存储设备的次设备号，一个存储设备将由主次两个设备号标识。
     mDiskNumParts = 0;
    // 设置状态为 NoMedia
     setState(Volume::State_NoMedia);
}
// 再来看 addPath 函数，它主要目的是添加设备在 sysfs 中的路径，G7 的 vold.fstab 上有两个路
// 径，见图 9-3 中的最后一行。
int DirectVolume::addPath(const char *path) {
    mPaths->push_back(strdup(path));
```

```
    return 0;
}
```

这里简单介绍一下 addPath 的作用。addPath 会把和某个存储卡接口相关的设备路径与这个 DirectVolume 绑定到一起，并且这个设备路径和 Uevent 中的 DEVPATH 是对应的，这样就可以根据 Uevent 的 DEVPATH 找到是哪个存储卡的 DirectVolume 发生了变动。当然手机上目前只有一个存储卡接口，所以 Vold 也只有一个 DirectVolume。

4. NM 和 VM 交互

在分析 NM 模块的数据处理时发现，NM 模块接收到 Uevent 事件后，会调用 VM 模块进行处理，下面来看这块的内容。

先回顾一下 NM 调用 VM 模块的地方，代码如下所示：

[-->NetlinkHandler.cpp]

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();
    ......
    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt); // 调用 VM 的 handleBlockEvent。
    } else if (!strcmp(subsys, "switch")) {
        vm->handleSwitchEvent(evt);// 调用 VM 的 handleSwitchEvent。
    }
    ......
}
```

在上面的代码中，如果 Uevent 是 block 子系统，则调用 handleBlockEvent；如果是 switch，则调用 handleSwitchEvent。handleSwitchEvent 主要处理 SD 卡挂载磁盘的通知，比较简单。这里只分析 handleBlockEvent 事件。代码如下所示：

[-->VolumeManager.cpp]

```
void VolumeManager::handleBlockEvent(NetlinkEvent *evt) {
    const char *devpath = evt->findParam("DEVPATH");

    /*
    前面在 process_config 中构造的 DirectVolume 对象保存在了 mVolumes 中，它的定义如下：
    typedef android::List<Volume *> VolumeCollection，也是一个列表。
    注意它保存的是 Volume 指针，而我们的 DirectVolume 是从 Volume 派生的。
    */
    VolumeCollection::iterator it;
    bool hit = false;
    for (it = mVolumes->begin(); it != mVolumes->end(); ++it) {
        // 调用每个 Volume 的 handleBlockEvent 事件，就我的 G7 手机而言，实际上将调用
        //DirectVolume 的 handleBlockEvent 函数。
        if (!(*it)->handleBlockEvent(evt)) {
```

```
            hit = true;
            break;
        }
    }
}
```

NM 收到 Uevent 消息后，DirectVolume 也将应声而动，它的 handleBlockEvent 的处理是：

[-->DirectVolume.cpp]

```
int DirectVolume::handleBlockEvent(NetlinkEvent *evt) {
    const char *dp = evt->findParam("DEVPATH");

    PathCollection::iterator  it;
    // 将 Uevent 的 DEVPATH 和 addPath 添加的路径进行对比，判断属不属于自己管理的范围。
    for (it = mPaths->begin(); it != mPaths->end(); ++it) {
        if (!strncmp(dp, *it, strlen(*it))) {
            int action = evt->getAction();
            const char *devtype = evt->findParam("DEVTYPE");

            if (action == NetlinkEvent::NlActionAdd) {
                int major = atoi(evt->findParam("MAJOR"));
                int minor = atoi(evt->findParam("MINOR"));
                char nodepath[255];

                snprintf(nodepath,
                         sizeof(nodepath), "/dev/block/vold/%d:%d",
                         major, minor);
                  // 创建设备节点。
                if (createDeviceNode(nodepath, major, minor)) {
                    ......
                }
                if (!strcmp(devtype, "disk")) {
                    handleDiskAdded(dp, evt);// 添加一个磁盘。
                } else {
                    /*
                    对于有分区的 SD 卡，先收到上面的“disk”消息，然后每个分区就会收到
                    一个分区添加消息。
                    */
                    handlePartitionAdded(dp, evt);
                }
            } else if (action == NetlinkEvent::NlActionRemove) {
                ......
            } else if (action == NetlinkEvent::NlActionChange) {
                ......
            }
            ......
            return 0;
        }
    }
```

```
    errno = ENODEV;
    return -1;
}
```

关于 DirectVolume 针对不同 Uevent 的具体处理方式，后面将通过一个 SD 卡插入案例来分析。

5. 关于 VM 模块的总结

从前面的代码分析可知，VM 模块的主要功能是管理 Android 系统中的外部存储设备。图 9-4 描述了 VM 模块的功能：

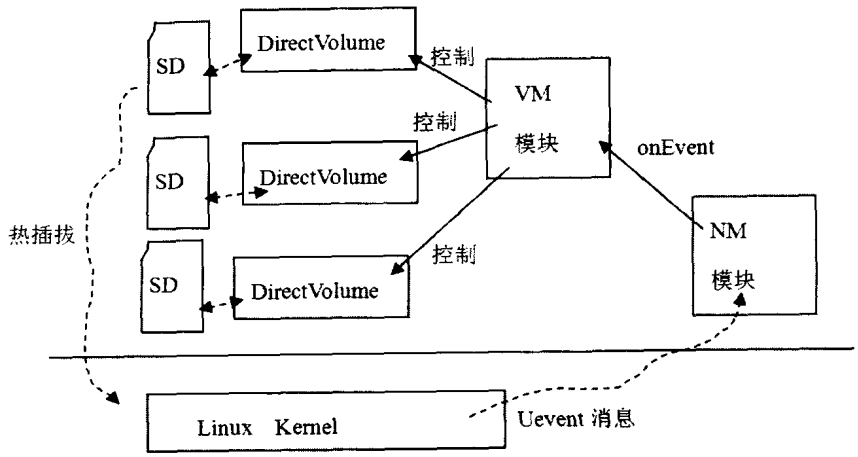


图 9-4 VM 模块的职责

通过对上图和前面代码的分析可知：

- SD 卡的变动（例如热插拔）将导致 Kernel 发送 Uevent 消息给 NM 模块。
- NM 模块调用 VM 模块处理这些 Uevent 消息。
- VM 模块遍历它所持有的 Volume 对象，Volume 对象根据 addPath 添加的 DEVPATH 和 Uevent 消息中的 DEVPATH 来判断自己是否可以处理这个消息。

至于 Volume 到底如何处理 Uevent 消息，将通过一个实例来分析。

9.2.5 CommandListener 模块分析

Vold 使用 CL 模块的流程是：

- 使用 new 创建一个 CommandListener 对象；
- 调用 CL 的 startListener 函数。

来看这两个函数。

1. 创建 CommandListener

和 NetlinkerHandler 一样，CommandListener 也有一个相对不简单的派生关系，它的家

族图谱如图 9-5 所示：

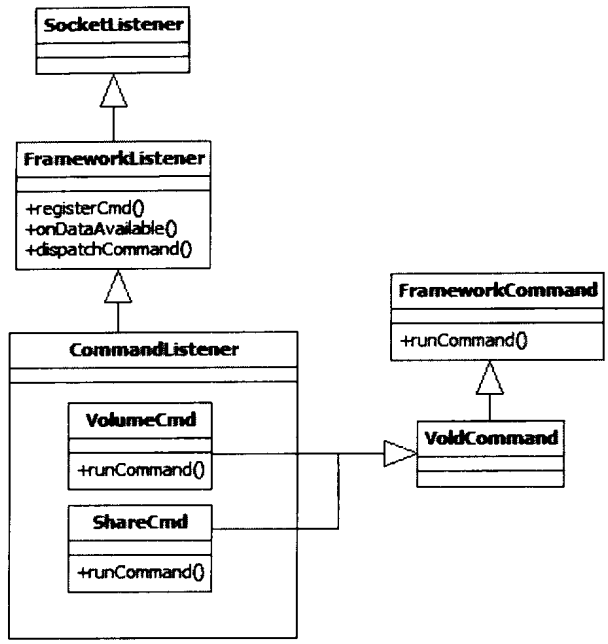


图 9-5　CommandListener 家族图谱

从上图可知：

- CL 定义了一些和 Command 相关的内部类，这里采用了设计模式中的 Command 模式，每个命令的处理函数都是 runCommand。注意，上图只列出了部分 Command 类。
- CL 也是从 SocketListener 派生的，不过它是 Socket 的监听（listen）端。

下面看它的代码：

[-->CommandListener.cpp]

```
CommandListener::CommandListener() :
                    FrameworkListener("vold") {
    //CL 模块支持的命令。
    registerCmd(new DumpCmd());
    registerCmd(new VolumeCmd());
    registerCmd(new AsecCmd());
    registerCmd(new ShareCmd());
    registerCmd(new StorageCmd());
    registerCmd(new XwarpCmd());
}
/*
registerCmd 函数将 Command 保存到 mCommands 中，mCommands 的定义为一个列表，如下：
typedef android::List<FrameworkCommand *> FrameworkCommandCollection;
*/
```

```
void FrameworkListener::registerCmd(FrameworkCommand *cmd) {
    mCommands->push_back(cmd);
}
```

从上面的代码可知，CommandListener 的基类是 FrameworkListener，而 FrameworkListener 又从 SocketListener 类派生。之前在分析 NM 模块的 NetLinkerHandler 时，已介绍过 SocketListener 相关的知识了，所以此处不再赘述，只总结一下 CL 创建后的结果，它们是：

- CL 会创建一个监听端的 socket，这样就可以接收客户端的链接了。
- 客户端发送命令给 CL，CL 则从 mCommands 中找到对应的命令，并交给该命令的 runCommand 函数处理。

下面来关注第二个函数 startListener，这个函数由 SocketListener 实现。

2. startListener 的分析和数据处理

其实在分析 NetlinkerHandler 时，已经介绍了 startListener 函数，这里再简单回顾一下，有些具体内容和本章对 NetlinkerHandler 的分析有关。

[-->SocketListener.cpp]

```
int SocketListener::startListener() {

    if (!mSocketName && mSock == -1) {
        ......
        errno = EINVAL;
        return -1;
    } else if (mSocketName) {
    //mSOcketName 为字符串 "vold"。android_get_control_socket 函数返回
    // 对应的 socket 句柄。
        if ((mSock = android_get_control_socket(mSocketName)) < 0) {
            ......
            return -1;
        }
    }
    //CL 模块是监听端
    if (mListen && listen(mSock, 4) < 0) {
        ......
        return -1;
    } else if (!mListen)
        mClients->push_back(new SocketClient(mSock));

    if (pipe(mCtrlPipe)) {
        ......
        return -1;
    }
    // 创建工作线程 threadStart。
    if (pthread_create(&mThread, NULL, SocketListener::threadStart, this)) {
        return -1;
    }
```

```
    return 0;
}
```

当 CL 收到数据时，会调用 onDataAvailable 函数，它由 FrameworkListener 实现。代码如下所示：

[-->FrameworkListener.cpp]

```
bool FrameworkListener::onDataAvailable(SocketClient *c) {
    char buffer[255];
    int len;
    // 读取数据。
    if ((len = read(c->getSocket(), buffer, sizeof(buffer) -1)) < 0) {
        ......
        return errno;
    } else if (!len)
        return false;

    int offset = 0;
    int i;

    for (i = 0; i < len; i++) {
        if (buffer[i] == '\0') {
           // 分发命令，最终会调用对应命令对象的 runCommand 进行函数处理。
            dispatchCommand(c, buffer + offset);
            offset = i + 1;
        }
    }
    return true;
}
```

dispatchCommand 最终会根据收到的命令名（如“Volume”、“Share”等）来调用对应的命令对象（如 VolumeCmd、ShareCmd）的 runCommand 函数以处理请求。这一块非常简单，这里就不再详述了。

3. 关于 CL 模块的总结

CL 模块的主要工作是：

- 建立一个监听端的 socket。
- 接收客户端的连接和请求，并调用对应 Command 对象的 runComand 函数处理。

目前，CL 模块唯一的客户端就是 MountService，下面我们通过一个实例来看看它。

9.2.6　Vold 实例分析

这一节将分析一个实际案例，即插入一张 SD 卡引发的事件，以及它的处理过程。在分析之前，先介绍 MountService。

1.MountService 介绍

有些应用程序需要检测外部存储卡的插入 / 拔出事件，这些事件是由 MountService 通过 Intent 广播发出的，例如外部存储卡插入后，MountService 就会发送 ACTION_MEDIA_MOUNTED 消息。从某种意义上来说，可以把 MountService 看成是 Java 世界的 Vold。来简单认识一下这个 MountService，它的代码如下所示：

[-->MountService.java]

```
class MountService extends IMountService.Stub
        implements INativeDaemonConnectorCallbacks {
//MountService 实现了 INativeDaemonConnectorCallbacks 接口。
......
public MountService(Context context) {
   mContext = context;

    ......
   // 创建一个 HandlerThread，在第 5 章中曾介绍过。
    mHandlerThread = new HandlerThread("MountService");
    mHandlerThread.start();
    /*
      创建一个 Handler，这个 Handler 使用 HandlerThread 的 Looper，也就是说，派发给该 Handler
      的消息将在另外一个线程中处理。可回顾第 5 章的内容，以加深印象。
    */
    mHandler = new MountServiceHandler(mHandlerThread.getLooper());

   ......
  /*
    NativeDaemonConnector 用于 Socket 通信，第二个参数“vold”表示将和 Vold 通信，也就是
    和 CL 模块中的那个 socket 建立通信连接。第一个参数为 INativeDaemonConnectorCallbacks
    接口。它提供两个回调函数：
    onDaemonConnected：当 NativeDaemonConnector 连接上 Vold 后回调。
    onEvent：当 NativeDaemonConnector 收到来自 Vold 的数据后回调。
  */
    mConnector = new NativeDaemonConnector(this, "vold", 10, "VoldConnector");
    mReady = false;
    // 再启动一个线程用于和 Vold 通信。
    Thread thread = new Thread(mConnector,
                               NativeDaemonConnector.class.getName());
    thread.start();
}
  ......
}
```

MountService 通过 NativeDaemonConnector 和 Vold 的 CL 模块建立通信连接，这部分内容比较简单，读者可自行研究。下面来分析 SD 卡插入后所引发的一连串处理。

2. 设备插入事件的处理

（1）Vold 处理 Uevent 事件

在插入 SD 卡后，Vold 的 NM 模块接收到 Uevent 消息，假设此消息的内容是前面介绍 Uevent 知识时使用的 add 消息，它的内容如下所示：

[-->SD 卡插入的 Uevent 消息]

```
add@/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0
ACTION=add //add 表示设备插入动作，另外还有 remove 和 change 等动作。
//DEVPATH 表示该设备位于 /sys 目录中的设备路径。
DEVPATH=/devices/platform/msm_sdcc.2/mmc_host/mmc1/mmc1:c9f2/block/mmcblk0
/*
SUBSYSTEM 表示该设备属于哪一类设备，block 为块设备，磁盘也属于这一类设备，另外还有
character（字符）设备等类型。
*/
SUBSYSTEM=block
MAJOR=179//MAJOR 和 MINOR 分别表示该设备的主次设备号，二者联合起来可以标识一个设备。
MINOR=0
DEVNAME=mmcblk0
DEVTYPE=disk// 设备 Type 为 disk。
NPARTS=3   // 表示该 SD 卡上的分区，我的 SD 卡上有三块分区。
SEQNUM=1357// 序号
```

根据前文的分析可知，NM 模块中的 NetlinkHandler 会处理此消息，请回顾一下相关代码：

[-->NetlinkHandler.cpp]

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();
  ......
    // 根据上面 Uevent 消息的内容可知，它的 subsystem 对应为 block，所以我们会走下面这个 if 分支。
    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt); // 调用 VM 的 handleBlockEvent。
    }
  ......
}
```

[-->VolumeManager.cpp]

```
void VolumeManager::handleBlockEvent(NetlinkEvent *evt) {
    const char *devpath = evt->findParam("DEVPATH");

    VolumeCollection::iterator it;
    bool hit = false;
    for (it = mVolumes->begin(); it != mVolumes->end(); ++it) {
      // 调用各个 Volume 的 handleBlockEvent。
        if (!(*it)->handleBlockEvent(evt)) {
```

```
                hit = true;
                break;
            }
        }
        ......
    }
```

我的 G7 手机只有一个 Volume，其实际类型就是之前介绍过的 DirectVolume。请看它是怎么对待这个 Uevent 消息的，代码如下所示：

[-->DirectVolume.cpp]

```
int DirectVolume::handleBlockEvent(NetlinkEvent *evt) {
    const char *dp = evt->findParam("DEVPATH");

    PathCollection::iterator  it;
    for (it = mPaths->begin(); it != mPaths->end(); ++it) {
        if (!strncmp(dp, *it, strlen(*it))) {
            int action = evt->getAction();
            const char *devtype = evt->findParam("DEVTYPE");

            if (action == NetlinkEvent::NlActionAdd) {
                int major = atoi(evt->findParam("MAJOR"));
                int minor = atoi(evt->findParam("MINOR"));
                char nodepath[255];

                snprintf(nodepath,
                         sizeof(nodepath), "/dev/block/vold/%d:%d",
                         major, minor);
                // 内部调用 mknod 函数创建设备节点。
                if (createDeviceNode(nodepath, major, minor)) {
                    SLOGE("Error making device node '%s' (%s)", nodepath,
                          strerror(errno));
                }
                if (!strcmp(devtype, "disk")) {
                    // 对应 Uevent 消息的 DEVTYPE 值为 disk，所以走这个分支。
                    handleDiskAdded(dp, evt);
                } else {
                   // 处理 DEVTYPE 为 Partition 的情况。
                    handlePartitionAdded(dp, evt);
                }
            } else if (action == NetlinkEvent::NlActionRemove) {
               // 对应 Uevent 的 ACTION 值为 remove。
                ......
            } else if (action == NetlinkEvent::NlActionChange) {
               // 对应 Uevent 的 ACTION 值为 change。
                ......
            }
             ......
```

```
            return 0;
        }
    }
    errno = ENODEV;
    return -1;
}
```

插入 SD 卡，首先收到的 Uevent 消息中 DEVTYPE 的值是“disk”，这将导致 DirectVolume 的 handleDiskInserted 被调用。下面来看它的工作。

[-->DirectVolume.cpp]

```
void DirectVolume::handleDiskAdded(const char *devpath, NetlinkEvent *evt) {
    mDiskMajor = atoi(evt->findParam("MAJOR"));
    mDiskMinor = atoi(evt->findParam("MINOR"));

    const char *tmp = evt->findParam("NPARTS");
    if (tmp) {
        mDiskNumParts = atoi(tmp);// 这个 disk 上的分区个数。
    } else {
       ......
        mDiskNumParts = 1;
    }

    char msg[255];

    int partmask = 0;
    int i;
    /*
      Partmask 会记录这个 Disk 上分区加载的情况。前面曾介绍过，如果一个 Disk 有多个分区，
    它后续则会收到多个分区的 Uevent 消息。
    */
      for (i = 1; i <= mDiskNumParts; i++) {
        partmask |= (1 << i);
    }
    mPendingPartMap = partmask;

    if (mDiskNumParts == 0) {
        ......// 如果没有分区，则设置 Volume 的状态为 Idle。
        setState(Volume::State_Idle);
    } else {
    ......// 如果还有分区未加载，则设置 Volume 状态为 Pending。
        setState(Volume::State_Pending);
    }
    /*
    设置通知内容，snprintf 调用完毕后 msg 的值为：
     "Volume sdcard /mnt/sdcard disk inserted (179:0)"
    */
    snprintf(msg, sizeof(msg), "Volume %s %s disk inserted (%d:%d)",
```

```
                getLabel(), getMountpoint(), mDiskMajor, mDiskMinor);
        /*
         getBroadcaster 函数返回 setBroadcaster 函数设置的那个 Broadcaster，也就是 CL 对象。
         然后调用 CL 对象的 sendBroadcast 给 MountService 发送消息，注意它的第一个参数是 ResponseC
         ode::VolumeDiskInserted。
        */
         mVm->getBroadcaster()->sendBroadcast(ResponseCode::VolumeDiskInserted,
                                                    msg, false);
    }
```

handleDiskAdded 把 Uevent 消息做一些转换后发送给了 MountService，实际上可认为 CL 模块和 MountService 通信使用的是另外一套协议。那么，MountService 会做什么处理呢？

（2）MountService 的处理

[-->MountService.java onEvent 函数]

```
public boolean onEvent(int code, String raw, String[] cooked) {
        Intent in = null;
        // 关于 onEvent 函数，在 MountService 的介绍中曾提过，当 NativeDaemonConnector 收到
         // 来自 vold 的数据后都会调用这个 onEvent 函数。
        ......
        if (code == VoldResponseCode.VolumeStateChange) {
           ......
        } else if (code == VoldResponseCode.ShareAvailabilityChange) {
            ......
        } else if ((code == VoldResponseCode.VolumeDiskInserted) ||
                   (code == VoldResponseCode.VolumeDiskRemoved) ||
                   (code == VoldResponseCode.VolumeBadRemoval)) {

            final String label = cooked[2]; //label 值为 "sdcard"

            final String path = cooked[3]; //path 值为 "/mnt/sdcard"
            int major = -1;
            int minor = -1;

            try {
                String devComp = cooked[6].substring(1, cooked[6].length() -1);
                String[] devTok = devComp.split(":");
                major = Integer.parseInt(devTok[0]);
                minor = Integer.parseInt(devTok[1]);
            } catch (Exception ex) {
                ......
            }
          if (code == VoldResponseCode.VolumeDiskInserted) {
               // 收到 handleDiskAdded 发送的 VolumeDiskInserted 消息了。
              // 单独启动一个线程来处理这个消息。
                new Thread() {
                    public void run() {
```

```
                    try {
                        int rc;
                        // 调用 doMountVolume 处理。
                        if ((rc = doMountVolume(path)) !=
                                StorageResultCode.OperationSucceeded) {
                            }
                    } catch (Exception ex) {
                        ......
                    }
                }
            }.start();
        }
```

doMountVolume 函数的代码如下所示：

[-->MountService.java]

```
private int doMountVolume(String path) {
        int rc = StorageResultCode.OperationSucceeded;
        try {
            // 通过 NativeDaemonConnector 给 Vold 发送请求，请求内容为：
           //volume mount /mnt/sdcard
            mConnector.doCommand(String.format("volume mount %s", path));
        } catch (NativeDaemonConnectorException e) {
           ......// 异常处理
       }
```

走了一大圈，最后又回到 Vold 了。CL 模块将收到这个来自 MountService 的请求，请求的内容为字符串“volume mount /mnt/sdcard”，其中的 volume 表示命令的名字，CL 会根据这个名字找到 VolumeCmd 对象，并交给它处理这个命令。

（3）Vold 处理 MountService 的命令

Vold 处理 MountService 命令的代码如下所示：

[-->CommandListener.cpp VolumeCmd 类]

```
int CommandListener::VolumeCmd::runCommand(SocketClient *cli,int argc, char **argv)
{
    ......
    VolumeManager *vm = VolumeManager::Instance();
    int rc = 0;

    if (!strcmp(argv[1], "list")) {
        return vm->listVolumes(cli);
    } else if (!strcmp(argv[1], "debug")) {
        ......
    } else if (!strcmp(argv[1], "mount")) {
       ......// 调用 VM 模块的 mountVolume 来处理 mount 命令，参数是 "/mnt/sdcard"。
        rc = vm->mountVolume(argv[2]);
    } else if (!strcmp(argv[1], "unmount")) {
```

```
            ......
            rc = vm->unmountVolume(argv[2], force);
        } else if (!strcmp(argv[1], "format")) {
            ......
            rc = vm->formatVolume(argv[2]);
        } else if (!strcmp(argv[1], "share")) {
            ......
            rc = vm->shareVolume(argv[2], argv[3]);
        } else if (!strcmp(argv[1], "unshare")) {
            ......
            rc = vm->unshareVolume(argv[2], argv[3]);
    }
     ......

    if (!rc) {
        // 发送处理结果给 MountService。
        cli->sendMsg(ResponseCode::CommandOkay, "volume operation succeeded", false);
    }
        ......
        return 0;
    }
```

来看 mountVolume 函数，代码如下所示：

[-->VolumeManager.cpp]

```
int VolumeManager::mountVolume(const char *label) {
    /*
     根据 label 找到对应的 Volume。label 这个参数的名字的含义有些歧义，根据 loopupVolume
     的实现来看，它其实比较的是 Volume 的挂载路径，也就是 vold.fstab 中指定的那个。
     /mnt/sdcard。而 vold.fstab 中指定的 label 叫 sdcard。
     */
     Volume *v = lookupVolume(label);
     ......
     return v->mountVol();//mountVol 由 Volume 类实现。
}
```

找到对应的 DirectVolume 后，也就找到了代表真实存储卡的对象。它是如何处理这个命令的呢？代码如下所示：

[-->Volume.cpp]

```
int Volume::mountVol() {
    dev_t deviceNodes[4];
    int n, i, rc = 0;
    char errmsg[255];
    ......
    //getMountpoint 返回挂载路径，即 /mnt/sdcard。
    //isMountpointMounted 判断这个路径是不是已经被 mount 了。
```

```
if (isMountpointMounted(getMountpoint())) {
     setState(Volume::State_Mounted);// 设置状态为 State_Mounted。
     return 0;// 如果已经被 mount 了，则直接返回。
 }

 n = getDeviceNodes((dev_t *) &deviceNodes, 4);
 ......
 for (i = 0; i < n; i++) {
     char devicePath[255];

     sprintf(devicePath, "/dev/block/vold/%d:%d", MAJOR(deviceNodes[i]),
             MINOR(deviceNodes[i]));
    ......
     errno = 0;
     setState(Volume::State_Checking);
    // 默认 SD 卡为 FAT 分区，只有这样，当加载为磁盘的时候才能被 Windows 识别。
     if (Fat::check(devicePath)) {
         ......
         return -1;
     }

     /*
        先把设备 mount 到 /mnt/secure/staging，这样 /mnt/secure/staging 下的内容
        就是该设备的存储内容了。
     */
     errno = 0;
     if (Fat::doMount(devicePath, "/mnt/secure/staging", false, false, 1000,
                      1015, 0702, true)) {
        ......
         continue;
     }
     /*
       下面这个函数会把存储卡中的 autorun.inf 文件找出来并删掉，这个文件就是“臭名昭著”的
       自动运行文件，在 Windows 系统上，把 SD 卡挂载为磁盘后，双击这个磁盘就会自动运行
       这个文件，很多病毒和木马都是通过它传播的。为了安全起见，要把这个文件删掉。
    */
    protectFromAutorunStupidity();
     // ①下面这个函数比较有意思，需要看看：
     if (createBindMounts()) {
         ......
         return -1;
     }

     // 将存储卡 mount 路径从 /mnt/secure/staging 移到 /mnt/sdcard。
     if (doMoveMount("/mnt/secure/staging", getMountpoint(), false)) {
        ......
         return -1;
     }
     // ②设置状态为 State_Mounted，这个函数将发送状态信息给 MountService。
     setState(Volume::State_Mounted);
```

```
            mCurrentlyMountedKdev = deviceNodes[i];
            return 0;
        }
        ......
        setState(Volume::State_Idle);

        return -1;
    }
```

上面的代码中有个比较有意思的函数，就是 createBindMounts，其代码如下所示：

[-->Volume.cpp]

```
int Volume::createBindMounts() {
    unsigned long flags;

   /*
    将 /mnt/secure/staging/android_secure 目录名改成
    /mnt/secure/staging/.android_secure，SEC_STG_SECIMGDIR 的值就是
    /mnt/secure/staging/.android_secure，也就是把它变成 Linux 平台上的隐藏目录。
   */
    if (!access("/mnt/secure/staging/android_secure", R_OK | X_OK) &&
         access(SEC_STG_SECIMGDIR, R_OK | X_OK)) {
        if (rename("/mnt/secure/staging/android_secure", SEC_STG_SECIMGDIR)) {
            SLOGE("Failed to rename legacy asec dir (%s)", strerror(errno));
        }
    }

    ......

/*
    使用 mount 命令的 bind 选项，可将 /mnt/secure/staging/.android_secure 这个目录
    挂载到 /mnt/secure/asec 目录下。/mnt/secure/asec 目录是一个 secure container，
    目前主要用来保存一些安装在 SD 卡上的 APP 信息。APP2SD 是 Android 2.2 引入的新机制，它
    支持将 APP 安装在 SD 卡上，这样可以节约内部的存储空间。
    mount 的 bind 选项允许将文件系统的一个目录挂载到另外一个目录下。读者可以通过 man mount
    查询具体信息。
     */
    if (mount(SEC_STG_SECIMGDIR, SEC_ASECDIR, "", MS_BIND, NULL)) {
         ......
        return -1;
    }

......
/*
将 tempfs 设备挂载到 /mnt/secure/staging/.android_secure 目录，这样之前
.android_secure 目录中的内容就只能通过 /mnt/secure/asec 访问了。由于那个目录只能
由 root 访问，所以可以起到安全保护的作用。
*/
if (mount("tmpfs", SEC_STG_SECIMGDIR, "tmpfs", MS_RDONLY,
```

```
                    "size=0,mode=000,uid=0,gid=0")) {
            ......
            umount("/mnt/asec_secure");
            return -1;
        }

        return 0;
    }
```

createBindMounts 的作用就是将存储卡上的 .android_secure 目录挂载到 /mnt/secure/asec 目录下，同时对 .android_secure 进行一些特殊处理，这样，没有权限的用户就不能更改或破坏 .android_secure 目录中的内容了，因此它起到了一定的保护作用。

注意　在手机上，受保护的目录内容只能通过 adb shell 登录后，进入 /mnt/secure/asec 目录来查看。注意，这个 asec 目录的内容就是 .android_secure 未挂载 tmpfs 时的内容（亦即它保存着那些安装在存储卡上的应用程序信息）。另外，可把 SD 卡拔出来，通过读卡器直接插到台式机上，此时，这些信息就能在 .android_secure 目录中直接看到了。

（4）关于 MountService 处理状态的通知

volume 的 mountVol 完成相关工作后，就通过下面的函数发送信息给 MountService：

```
setState(Volume::State_Mounted); // 感兴趣的读者可自行分析此函数的实现。
```

MountService 依然会在 onEvent 函数中收到这个消息。

[-->MountService.java]

```
public boolean onEvent(int code, String raw, String[] cooked) {
        Intent in = null;

        ......
        if (code == VoldResponseCode.VolumeStateChange) {
            /*
              状态变化由 notifyVolumeStateChange 函数处理，由于 Volume 的状态
              被置成了 Mounted，所以下面这个 notifyVolumeStateChange 会发送
              ACTION_MEDIA_MOUNTED 这个广播。我们就不再分析这个函数了，读者
              可自行研究。
            */
            notifyVolumeStateChange(
                    cooked[2], cooked[3], Integer.parseInt(cooked[7]),
                            Integer.parseInt(cooked[10]));
        }
```

实例分析就到这里。中间略去了一些处理内容，例如对分区的处理等，读者可自行研究，相信已没有太大难度了。另外，在上述处理过程中，稍微难懂的是 mountVol 这个函数在挂载方面的处理过程。用图 9-6 来总结一下这个处理过程：

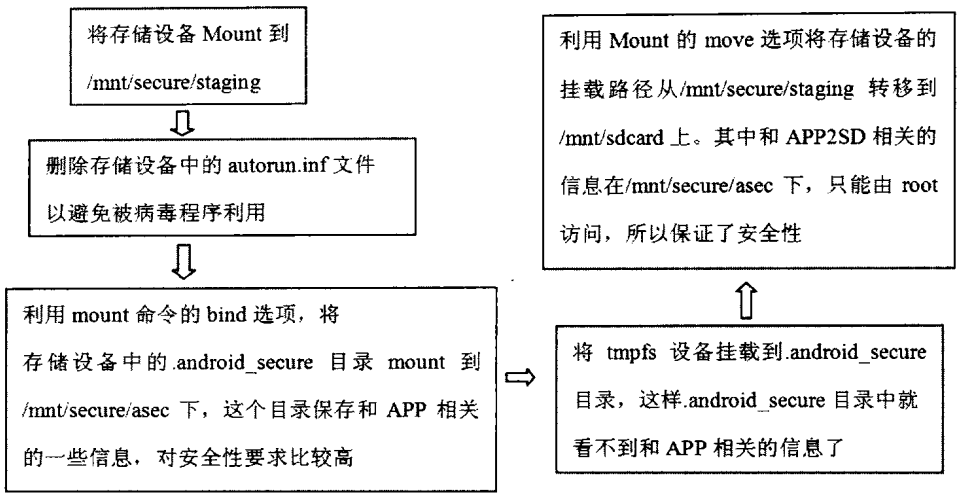


图 9-6 SD 卡插入事件处理流程图

由上图可知，Vold 在安全性上还是进行了一定考虑的。如果没有特殊需要，读者了解上面这些知识也就够了。

9.2.7 关于 Vold 的总结

Vold 和 MountService 之间的通信使用了 Socket。之前在第 6 章介绍 Binder 时也提到过它。除了 Binder 外，Socket 是 Android 系统中最常用的 IPC 通信机制了。本章介绍的 Vold 和 Rild 都是使用 Socket 进行 IPC 通信的。

Vold 及 Java 层的 MountService 都比较简单，所以我在工作中碰到它们出问题的几率基本为零。虽然二者比较简单，这里还是要提个小小的问题，以帮助大家加深印象：

当 SD 卡拔出，或者挂载到磁盘上时，都会导致 SD 卡被卸载，在这个切换过程中，有一些应用程序会被系统 kill 掉，这是为什么？

请读者阅读相关代码寻找答案，这样或许能解释很多测试人员在做测试时提出这种 Bug 的原因：为什么 SD 卡 mount 到电脑后，有些应用程序突然退出了？

9.3 Rild 的原理与机制分析

这里先回顾一下智能手机的架构。目前，很多智能手机的硬件架构都是两个处理器：一个处理器用来运行操作系统，上面可以跑应用程序，这个处理器称作 Application Processor，简称 AP；另一个处理器负责和射频无线通信相关的工作，叫 Baseband Processor，简称 BP。AP 和 BP 芯片之间采用串口进行通信，通信协议使用的是 AT 指令。

说明 什么是 AT 指令呢？AT 指令最早用在 Modem 上，后来几大手机厂商如摩托罗拉、爱

立信、诺基亚等为 GSM 通信又设计了一整套 AT 指令。AT 指令的格式比较简单，是一个以 AT 开头，后跟字母和数字表示具体功能的字符串。要了解具体的 AT 指令，可参考相关的规范或手机厂商提供的手册，这里就不再多说了。

在 Android 系统中，Rild 运行在 AP 上，它是 AP 和 BP 在软件层面上通信的中枢，也就是说，AP 上的应用程序将通过 Rild 发送 AT 指令给 BP，而 BP 的信息通过 Rild 传送给 AP 上的应用程序。

下面介绍一下在 Rild 代码中常会碰到的两个词语：

- 第一个 solicited Response，即经过请求的回复。它代表的应用场景是 AP 发送一个 AT 请求指令给 BP 进行处理，处理后，BP 会对应回复一个 AT 指令告知处理结果。这个回复指令是针对之前的那个请求指令的，此乃一问一答式，所以叫 solicited Response。
- 第二个 unsolicited Response，即未经请求的回复。很多时候，BP 主动给 AP 发送 AT 指令，这种指令一般是 BP 通知 AP 当前发生的一些事情，例如一个电话打了过来，或者网络信号中断等。从 AP 的角度来看，这种指令并非由它发送的请求所引起，所以称之为 unsolicited Response。

上面这两个词语，实际指明了 AP 和 BP 的两种交互类型：

- AP 发送请求给 BP，BP 响应并回复 AP。
- BP 发送通知给 AP。

这两种类型对软件而言有什么意义呢？先来看 Rild 在软件架构方面遇到的挑战：

- 有很多把 AP 和 BP 集成在一块芯片上的智能手机，它们之间的通信可能就不是 AT 指令了。
- 即使 AP 和 BP 通信使用的是 AT 指令，不同的手机厂商在 AT 指令上也会有很大的不同，而且这些都属于商业秘密，所以手机厂商不可能共享源码，它只能给出二进制的库。

Rild 是怎么解决这个问题的呢？结合前面提到的 AP/BP 交互的两种类型，大体可以勾画出图 9-7 来：

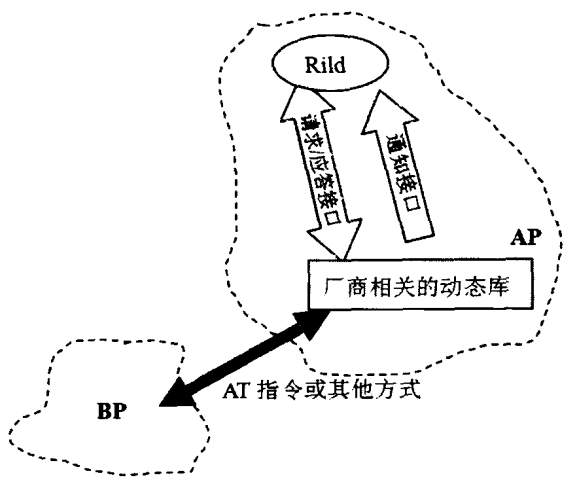


图 9-7　Rild 解决问题的方法

从上图中可以看出：

- Rild 会动态加载厂商相关的动态库，这个动态库加载在 Linux 平台上则使用 dlopen 系统调用。
- Rild 和动态库之间通过接口进行通信，也就是说 Rild 输出接口供动态库使用，而动态库也输出对应的接口供 Rild 使用。
- AP 和 BP 交互的工作由动态库去完成。

注意 Rild 和动态库运行在同一个进程上，为了方便理解，可把这两个东西分离开来。

根据上面的分析可知，对 Rild 的分析包括两部分：

- 对 Rild 本身的分析。
- 对动态库的分析。Android 提供了一个用作参考的动态库叫 libReference_ril.so，这个库实现了一些标准的 AT 指令。另外，它的代码结构也颇具参考价值，所以我们的动态库分析就以它为主。

分析 Rild 时，为了书写方便，我们将这个动态库简称为 RefRil 库。

9.3.1 初识 Rild

Rild 的代码在 Rild.c 中，它是一个应用程序。从它的 main 开始分析，代码如下所示：

[-->Rild.c]

```
int main(int argc, char **argv)
{
    // 动态库的位置由 rilLibPath 决定。
    const char * rilLibPath = NULL;
    char **rilArgv;
    void *dlHandle;

    /*
      Rild 规定动态库必须实现一个叫 Ril_init 的函数，这个函数的第一个参数指向结构体
      RIL_Env，而它的返回值指向结构体 RIL_ RadioFunctions。这两个结构体就是在
      图 9-7 中提到的接口。这两个接口的具体内容，后文再做分析。
    */
    const RIL_RadioFunctions *(*rilInit)(const struct RIL_Env *, int, char **);
    const RIL_RadioFunctions *funcs;
    char libPath[PROPERTY_VALUE_MAX];
    unsigned char hasLibArgs = 0;

    int i;
   //Rild 由 init 启动，没有对应的启动参数，所以这个 for 循环不会进来。
    for (i = 1; i < argc ;) {
        if (0 == strcmp(argv[i], "-l") && (argc - i > 1)) {
            rilLibPath = argv[i + 1];
            i += 2;
```

```
        } else if (0 == strcmp(argv[i], "--")) {
            i++;
            hasLibArgs = 1;
            break;
        } else {
            usage(argv[0]);
        }
    }

if (rilLibPath == NULL) {
    /*
       读取系统属性，LIB_PATH_PROPERTY 的值为 "rild.libpath"，模拟器上
       和 RIL 相关的属性值有两个，分别是：
       rild.libpath=/system/lib/libreference-ril.so
       rild.libargs=-d /dev/ttyS0
       上面这些值都定义在 build/target/board/generic/system.prop 文件中。
       不同厂商可以有自己对应的实现。
    */
    if ( 0 == property_get(LIB_PATH_PROPERTY, libPath, NULL)) {
            goto done;
        } else {
           /*
            这里，使用参考的动态库进行分析，它的位置为
            /system/lib/libreference-ril.so。
            */
            rilLibPath = libPath;
        }
    }

    ......  // 和模拟器相关的一些内容。
    switchUser();// 设置 Rild 的组用户为 radio。

   // 通过 dlopen 系统加载动态库。
    dlHandle = dlopen(rilLibPath, RTLD_NOW);
    ......
    // ① 启动 EventLoop，事件处理。
    RIL_startEventLoop();

   // 得到 RefRil 库中 RIL_Init 函数的地址。
   rilInit = (const RIL_RadioFunctions *(*)(const struct RIL_Env *, int,
                    char **))dlsym(dlHandle, "RIL_Init");

    ......

    rilArgv[0] = argv[0];
    // ②调用 RefRil 库输出的 RIL_Init 函数，注意传入的第一个参数和它的返回值。
    funcs = rilInit(&s_rilEnv, argc, rilArgv);
    // ③注册上面 rilInit 函数的返回值（一个 RIL_RadioFunctions 类型的结构体）到 Rild 中。
    RIL_register(funcs);
```

```
done:

    while(1) {
        // 主线程 sleep，具体工作交给工作线程完成。
        sleep(0x00ffffff);
    }
}
```

将上面的代码和分析结合起来，就知道 Rild 解决问题的方法了，代码中列出了三个关键点（即①～③），我们将逐一对其进行分析。

9.3.2 RIL_startEventLoop 分析

第一个关键点是 RIL_startEventLoop 函数，这个函数实际上是由 libRil.so 实现的，它的代码在 Ril.cpp 中，代码如下所示：

[-->Ril.cpp]

```
extern "C" void RIL_startEventLoop(void) {
    int ret;
    pthread_attr_t attr;

    s_started = 0;
    pthread_mutex_lock(&s_startupMutex);

    pthread_attr_init (&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    // 创建工作线程 eventLoop。
    ret = pthread_create(&s_tid_dispatch, &attr, eventLoop, NULL);

    /*
      工作线程 eventLoop 运行后会设置 s_started 为 1，并触发 s_startupCond。
      这几个语句的目的是保证在 RIL_startEventLoop 返回前，工作线程一定是已经创建并运行了。
    */
    while (s_started == 0) {
        pthread_cond_wait(&s_startupCond, &s_startupMutex);
    }
    pthread_mutex_unlock(&s_startupMutex);
    if (ret < 0) {

        return;
    }
}
```

从上面的代码可知，RIL_startEventLoop 会等待工作线程创建并运行成功。这个线程为什么会如此重要呢？下面就来了解一下工作线程 eventLoop。

1. 工作线程 eventLoop

工作线程 eventLoop 的代码如下所示：

[-->Ril.cpp]

```
static void * eventLoop(void *param) {
    int ret;
    int filedes[2];

    //①初始化请求队列。
    ril_event_init();

    //下面这几个操作告诉 RIL_startEventLoop 函数本线程已经创建并成功运行了。
    pthread_mutex_lock(&s_startupMutex);

    s_started = 1;
    pthread_cond_broadcast(&s_startupCond);

    pthread_mutex_unlock(&s_startupMutex);

   //创建匿名管道。
    ret = pipe(filedes);

   ......

    s_fdWakeupRead = filedes[0];
    s_fdWakeupWrite = filedes[1];
    //设置管道读端口的属性为非阻塞。
    fcntl(s_fdWakeupRead, F_SETFL, O_NONBLOCK);

   //②下面这两句话将匿名管道的读写端口加入到 event 队列中。
    ril_event_set (&s_wakeupfd_event, s_fdWakeupRead, true,
                processWakeupCallback, NULL);
    rilEventAddWakeup (&s_wakeupfd_event);

    //③进入事件等待循环中，等待外界触发事件并做对应的处理。
    ril_event_loop();
    return NULL;
}
```

工作线程的工作并不复杂，主要有三个关键点。

（1）ril_event_init 分析

工作线程，顾名思义就是用来干活的。要让它干活，是否得有一些具体的任务呢？它是如何管理这些任务的呢？对这两问题的回答是：

- 工作线程使用了一个叫 ril_event 的结构体来描述一个任务，并且它将多个任务按时间顺序组织起来，保存在任务队列中。这个时间顺序是指该任务的执行时间，由外界设定，可以是未来的某时间。

❑ ril_event_init 函数就是用来初始化相关队列和管理结构的，代码如下所示：

注意 在代码中，“任务”也称为“事件”，如没有特殊说明必要，这两者以后不再做区分。

[-->Ril.cpp]

```
void ril_event_init()
{
    MUTEX_INIT();// 初始化一个 mutex 对象 listMutex。

    FD_ZERO(&readFds); // 初始化 readFds，看来 Ril 会使用 select 来做多路 IO 复用。

    // 下面的 timer_list 和 pending_list 分别是两个队列。
    init_list(&timer_list);// 初始化 timer_list，任务插入的时候按时间排序。
    init_list(&pending_list);// 初始化 pendling_list，保存每次需要执行的任务。
    /*
      watch_table（监控表）定义如下：
      static struct ril_event * watch_table[MAX_FD_EVENTS];
      其中 MAX_FD_EVENTS 的值为 8。监控表主要用来保存那些 FD 已经加入到 readFDs 中的
      任务。
    */
    memset(watch_table, 0, sizeof(watch_table));
}
```

此 ril_event_init 函数没什么新鲜的内容。任务在代码中的对等物 Ril_event 结构的代码如下所示：

[-->Ril_event.h]

```
struct ril_event {
    struct ril_event *next;
    struct ril_event *prev;//next 和 prev 将 ril_event 组织成了一个双向链表。

    int fd;    // 该任务对应的文件描述符，以后简称 FD。
    int index;  // 这个任务在监控表中的索引。
    /*
      是否永久保存在监控表中，一个任务处理完毕后将根据这个 persist 参数来判断。
       是否需要从监控表中移除。
    */
    bool persist;
    struct timeval timeout; // 该任务的执行时间。
    ril_event_cb func;    // 任务函数。
    void *param;  // 传给任务函数的参数。
};
```

ril_event_init 刚初始化完任务队列，下面就有地方添加任务了。

（2）任务加入队列

下面这两行代码初始化一个 FD 为 s_wakeupfd_event 的任务，并将其加入到监控表中：

```
/*
  s_wakeupfd_event 定义为一个静态的 ril_event，ril_event_set 函数将初始化它的
  FD 为管道的读端，任务函数 ril_event_cb 对应为 processWakeupCallback，
  并设置 persist 为 true。
*/
ril_event_set (&s_wakeupfd_event, s_fdWakeupRead, true,
                processWakeupCallback, NULL);
// 来看这个函数：
rilEventAddWakeup (&s_wakeupfd_event);
```

rilEventAddWakeup 比较有意思，来看这个函数，代码如下所示：

[-->Ril.cpp]

```
static void rilEventAddWakeup(struct ril_event *ev) {
    ril_event_add(ev);//ev 指向一项任务
    triggerEvLoop();
}
// 直接看 ril_event_add 函数和 triggerEvLoop 函数。
void ril_event_add(struct ril_event * ev)
{
    ......
    MUTEX_ACQUIRE();// 锁保护
    for (int i = 0; i < MAX_FD_EVENTS; i++) {
    // 从监控表中找到第一个空闲的索引，然后把这个任务加到监控表中，
    //index 表示这个任务在监控中的索引。
        if (watch_table[i] == NULL) {
            watch_table[i] = ev;
            ev->index = i;
            ......
            // 将任务的 FD 加入到 readFds 中，这是 select 使用的标准方法。
            FD_SET(ev->fd, &readFds);
            if (ev->fd >= nfds) nfds = ev->fd+1;
            ......
            break;
        }
    }
    MUTEX_RELEASE();
    ......
}
// 再来看 triggerEvLoop 函数，这个就更简单了：
static void triggerEvLoop() {
    int ret;
    /*
    s_tid_dispatch 是工作线程 eventLoop 的线程 ID，pthread_self 返回调用线程的线程 ID。
    由于这里调用 triggerEvLoop 的就是 eventLoop 自己，所以不会走 if 分支，但是可以看看
    里面的内容。
    */
    if (!pthread_equal(pthread_self(), s_tid_dispatch)) {
        do {
```

```
            //s_fdWakeupWrite 为匿名管道的写端口，看来触发 eventLoop 工作的条件就是
            // 往这个端口写一点数据了。
            ret = write (s_fdWakeupWrite, " ", 1);
        } while (ret < 0 && errno == EINTR);
    }
}
```

一般的线程间通信使用同步对象来触发，而 rild 是通过往匿名管道写数据来触发工作线程工作的。

（3）ril_event_loop 分析

来看最后一个关键函数 ril_event_loop，其代码如下所示：

[-->Ril.cpp]

```
void ril_event_loop()
{
    int n;
    fd_set rfds;
    struct timeval tv;
    struct timeval * ptv;

    for (;;) {
        memcpy(&rfds, &readFds, sizeof(fd_set));
        /*
         根据 timer_list 来计算 select 函数的等待时间，timer_list 已经
         按任务的执行时间排好序了。
        */
        if (-1 == calcNextTimeout(&tv)) {
            ptv = NULL;
        } else {
            ptv = &tv;
        }
        ......;
        // 调用 select 进行多路 IO 复用。
        n = select(nfds, &rfds, NULL, NULL, ptv);
        ......
        // 将 timer_list 中那些执行时间已到的任务移到 pending_list 队列。
        processTimeouts();
        // 从监控表中转移那些有数据要读的任务到 pending_list 队列，如果任务的 persisit 不为
        //true，则同时从监控表中移除这些任务。
        processReadReadies(&rfds, n);
        // 遍历 pending_list，执行任务的任务函数。
        firePending();
    }
}
```

根据对 ril_event_Loop 函数的分析可知，Rild 支持两种类型的任务：

- 定时任务。它的执行由执行时间决定，和监控表没有关系，在 Ril.cpp 中由 ril_timer_

add 函数添加。

- 非定时任务，也叫 Wakeup Event。这些任务的 FD 将加入到 select 的读集合（readFDs）中，并且在监控表中存放了对应的任务信息。它们触发的条件是这些 FD 可读。对于管道和 Socket 来说，FD 可读意味着接收缓冲区中有数据，这时调用 recv 不会因为没有数据而阻塞。

注意　对于处于 listen 端的 socket 来说，FD 可读表示有客户端连接上了，此时需要调用 accept 接受连接。

2. 关于 RIL_startEventLoop 的总结

总结一下 RIL_startEventLoop 的工作。从代码中看，这个函数将启动一个比较重要的工作线程 eventLoop，该线程主要用来完成一些任务处理，而目前还没有给它添加任务。

9.3.3　RIL_Init 分析

下面看第二个关键函数 RIL_Init。这个函数必须由动态库实现，对于下面这个例子来说，它将由 RefRil 库实现，这个函数定义在 Reference_ril.c 中：

[-->Reference_ril.c]

```
pthread_t s_tid_mainloop;// 看来又会创建一个线程。

// 动态库必须实现的 RIL_Init 函数。
const RIL_RadioFunctions *RIL_Init(const struct RIL_Env *env,
                                   int argc, char **argv)
{
    int ret;
    int fd = -1;
    int opt;
    pthread_attr_t attr;

    s_rilenv = env; // 将外部传入的 env 保存为 s_rilenv。

     ......// 一些参数处理，不必管它。

    pthread_attr_init (&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    // 创建一个工作线程 mainLoop。
    ret = pthread_create(&s_tid_mainloop, &attr, mainLoop, NULL);
    /*
      s_callbacks 也为一个结构体。
    static const RIL_RadioFunctions s_callbacks = {
        RIL_VERSION,   //RIL 的版本。
        onRequest,   // 下面是一些函数指针。
        currentState,
        onSupports,
```

```
        onCancel,
        getVersion
    };

    */
    return &s_callbacks;
}
```

RefRil 的 RIL_Init 函数比较简单，主要有三项工作要做：

- 保存 Rild 传入的 RIL_Env 结构体。
- 创建一个叫 mainLoop 的工作线程。
- 返回一个 RIL_RadioFunctions 的结构体。

上面的 RIL_Env 和 RIL_RadioFunctions 结构体，就是 Rild 架构中用来隔离通用代码和厂商相关代码的接口。先来看 RIL_RadioFunctions，这个结构体由厂商的动态库实现，它的代码如下：

```
// 函数指针定义。
typedef void (*RIL_RequestFunc) (int request, void *data,
                                 size_t datalen, RIL_Token t);
typedef RIL_RadioState (*RIL_RadioStateRequest)();
typedef int (*RIL_Supports)(int requestCode);
typedef void (*RIL_Cancel)(RIL_Token t);
typedef void (*RIL_TimedCallback) (void *param);
typedef const char * (*RIL_GetVersion) (void);

typedef struct {
    int version;  //RIL 的版本

    // 通过这个接口可向 BP 提交一个请求，注意这个函数的返回值为空，这是为什么？
    RIL_RequestFunc onRequest;
    RIL_RadioStateRequest onStateRequest;// 查询 BP 的状态。
    RIL_Supports supports;
    RIL_Cancel onCancel;
    // 查询动态库的版本，RefRil 库中该函数的实现将返回字符串 "android reference-ril 1.0"
    RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

对于上面的结构体，应重点关注函数 onRequest，它被 Rild 用来向动态库提交一个请求，也就是说，AP 向 BP 发送请求的接口就是它，但是这个函数却没有返回值，那么该请求的执行结果是怎么得到的呢？

这里不卖关子，直接告诉大家。Rild 架构中最大的特点就是采用了异步请求 / 处理的方式。这种方式和异步 I/O 有异曲同工之妙。那么什么是异步请求 / 处理呢？它的执行流程如下：

- Rild 通过 onRequest 向动态库提交一个请求，然后返回去做自己的事情。
- 动态库处理这个请求，请求的处理结果通过回调接口通知。

这种异步请求 / 处理的流程和酒店的 Morning Call 服务很类似，具体相似之处如下所示：

- 在前台预约了一个 Morning Call，这好比向酒店提交了一个请求。预约完后，就可以放心地做自己的事情了。
- 酒店登记了这个请求，记录是哪个房间申请的服务，然后由酒店安排工作人员值班，这些都是酒店对这个请求的处理，作为房客则无须知道处理细节。
- 第二天早上，约好的时间一到，酒店给房客打电话，房客就知道这个请求被处理了。为了检查一下宾馆服务的效果，最好是拿表看看接到电话的时间是不是之前预约的时间。

有了上面的介绍，读者对异步请求 / 处理机制或许有了一些直观的感受。那么，动态库是如何通知请求的处理结果的呢？这里用到了另外一个接口 RIL_Env 结构，它的定义如下所示：

[-->Ril.h]

```
struct RIL_Env {
    // 动态库完成一个请求后，通过下面这个函数通知处理结果，其中第一个参数标明是哪个请求
    // 的处理结果。
    void (*OnRequestComplete)(RIL_Token t, RIL_Errno e,
                              void *response, size_t responselen);
    // 动态库用于进行 unsolicited Response 通知的函数。
    void (*OnUnsolicitedResponse)(int unsolResponse, const void *data,
                                  size_t datalen);

    // 给 Rild 提交一个超时任务。
    void* (*RequestTimedCallback) (RIL_TimedCallback callback,
                                   void *param, const struct timeval *relativeTime);
   // 从 Rild 的超时任务队列中移除一个任务。
    void (*RemoveTimedCallback) (void *callbackInfo);
};
```

结合图 9-7 和上面的分析可以发现，Rild 在设计时将请求的应答接口和动态库的通知接口都放在了 RIL_Env 结构体中。

关于 Rild 和动态库的交互接口就分析到这里。相信读者已经明白其中的原理了。下面来看 RefRil 库创建的工作线程 mainLoop。

1. 工作线程 mainLoop 分析

RefRil 库的 RIL_Init 函数会创建一个工作线程 mainLoop，其代码如下所示：

[-->Reference_Ril.c]

```
static void *
mainLoop(void *param)
{
    int fd;
    int ret;
```

```
    ......
    /*
      为 AT 模块设置一些回调函数，AT 模块用来和 BP 交互，对于 RefRil 库来说，AT 模块就是对
      串口设备通信的封装，这里统称为 AT 模块。
    */
    at_set_on_reader_closed(onATReaderClosed);
    at_set_on_timeout(onATTimeout);

    for (;;) {
        fd = -1;
        // 下面这个 while 循环的目的是为了得到串口设备的文件描述符，我们省略其中的一些内容
        while  (fd < 0) {
            if (s_port > 0) {
                fd = socket_loopback_client(s_port, SOCK_STREAM);
            } else if (s_device_socket) {
                if (!strcmp(s_device_path, "/dev/socket/qemud")) {
                    ......
            } else if (s_device_path != NULL) {
                fd = open (s_device_path, O_RDWR);
                if ( fd >= 0 && !memcmp( s_device_path, "/dev/ttyS", 9 ) ) {
                    struct termios  ios;
                    tcgetattr( fd, &ios );
                    ios.c_lflag = 0;
                    tcsetattr( fd, TCSANOW, &ios );
                }
            }

            ......
        }

        s_closed = 0;
        // ①打开 AT 模块，传入一个回调函数 onUnsolicited。
        ret = at_open(fd, onUnsolicited);

        ......
        // ②下面这个函数向 Rild 提交一个超时任务，该任务的处理函数是 initializeCallback。
        RIL_requestTimedCallback(initializeCallback, NULL, &TIMEVAL_0);
        sleep(1);
       /*
         如果 AT 模块被关闭，则 waitForClose 返回，但是该线程并不会退出，而是从 for 循环那
         开始重新执行一次。所以这个 mainLoop 工作线程是用来监控 AT 模块的，一旦它被关闭，就
         需要重新打开，也就是说不允许 AT 模块被关闭。
      */
        waitForClose();
        ......
    }
}
```

可以看到，mainLoop 的工作其实就是初始化 AT 模块，并监控 AT 模块，一旦 AT 模块被关闭，那么 mainLoop 就要重新打开并初始化它。这几项工作主要由 at_open 和超时任务

的处理函数 initializeCallback 完成。

（1）at_open 分析

来看 at_open 这个函数，其代码如下所示：

[-->Atchannle.c]

```
int at_open(int fd, ATUnsolHandler h)
{
   //at_open 的第一个参数是一个代表串口设备的文件描述符。
    int ret;
    pthread_t tid;
    pthread_attr_t attr;

    s_fd = fd;
    s_unsolHandler = h;
    s_readerClosed = 0;

    s_responsePrefix = NULL;
    s_smsPDU = NULL;
    sp_response = NULL;

    ..... //和电源管理相关的操作。

    pthread_attr_init (&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    // 创建一个工作线程 readerLoop，这个线程的目的就是从串口设备读取数据。
    ret = pthread_create(&s_tid_reader, &attr, readerLoop, &attr);
    ......
    return 0;
}
```

at_open 函数会另外创建一个工作线程 readerLoop，从名字上看，它会读取串口设备。下面来看它的工作，代码如下所示：

[-->Atchannle.c]

```
static void *readerLoop(void *arg)
{
    for (;;) {
        const char * line;

        line = readline(); // 从串口设备读取数据。
        ......

        if(isSMSUnsolicited(line)) {
            char *line1;
            const char *line2;

            line1 = strdup(line);
```

```
            line2 = readline();

            if (line2 == NULL) {
                break;
            }

            if (s_unsolHandler != NULL) {
                s_unsolHandler (line1, line2);// 调用回调，处理 SMS 的通知。
            }
            free(line1);
        } else {
          // 处理接收到的数据，也就是根据 line 中的 AT 指令调用不同的回调。
            processLine(line);
        }

      ......// 电源管理相关。
    // 这个线程退出前会调用通过 at_set_on_reader_closed 设置的回调函数，以通知
    //AT 模块关闭。
     onReaderClosed();

     return NULL;
}
```

readerLoop 工作线程比较简单，就是从串口设备中读取数据，然后进行处理。这些数据有可能是 solicited response，也有可能是 unsolicited response，具体的处理函数我们在后续的实例分析中再来介绍，下面我们看第二个函数 RIL_requestTimedCallback。

（2）initializeCallback 分析

在分析 initializeCallback 函数前，我们先看看 RefRil 向 Rild 提交超时任务的 RIL_requestTimedCallback 函数，它其实是一个宏，不过这个宏比较简单，就是封装了 RIL_Env 结构体中对 RequestTimedCallback 函数的调用，代码如下所示：

```
#define RIL_requestTimedCallback(a,b,c) \
        s_rilenv->RequestTimedCallback(a,b,c)
// 向 Rild 提交一个超时处理函数。
```

下面我们看看 Rild 实现的这个 RequestTimedCallback 函数，代码如下所示。

[-->Ril.cpp]

```
extern "C" void *
RIL_requestTimedCallback (RIL_TimedCallback callback, void *param,
                          const struct timeval *relativeTime) {
   return internalRequestTimedCallback (callback, param, relativeTime);
}
/*
调用 internalRequestTimedCallback，其实就是构造一个 Ril_event 事件然后加入到
timer_list 中，并触发 event_loop 工作线程执行。
*/
```

```
static UserCallbackInfo * internalRequestTimedCallback (
                                RIL_TimedCallback callback, void *param,
                               const struct timeval *relativeTime)
{
    struct timeval myRelativeTime;
    UserCallbackInfo *p_info;

    p_info = (UserCallbackInfo *) malloc (sizeof(UserCallbackInfo));

    p_info->p_callback = callback;
    p_info->userParam = param;
    if (relativeTime == NULL) {
        memset (&myRelativeTime, 0, sizeof(myRelativeTime));
    } else {
        memcpy (&myRelativeTime, relativeTime, sizeof(myRelativeTime));
    }

    ril_event_set(&(p_info->event), -1, false, userTimerCallback, p_info);
    // 将该任务添加到 timer_list 中去。
    ril_timer_add(&(p_info->event), &myRelativeTime);

    triggerEvLoop(); // 触发 eventLoop 线程。
    return p_info;
}
```

从上面的代码可知，RIL_requestTimedCallback 函数就是向 eventLoop 提交一个超时任务，这个任务的处理函数则为 initialCallback，下面直接来看该函数的内容，如下所示：

[-->Reference_ril.c]

```
static void initializeCallback(void *param)
{
   /*
    这个函数就是通过发送一些 AT 指令来初始化 BP 中的无线通信 Modem，不同的 modem 可能有
    不同的 AT 指令。这里仅列出部分代码。
   */
    ATResponse *p_response = NULL;
    int err;
    setRadioState (RADIO_STATE_OFF);
    at_handshake();
    ......
    err = at_send_command("AT+CREG=2", &p_response);
    ......
    at_response_free(p_response);
    at_send_command("AT+CGREG=1", NULL);
    at_send_command("AT+CCWA=1", NULL);
     ......
    if (isRadioOn() > 0) {
        setRadioState (RADIO_STATE_SIM_NOT_READY);
```

```
    }
    ......
    }
```

2. 关于 RIL_Init 的总结

RIL_Init 函数由动态库提供，以上面 RefRil 库的代码为参考，这个函数执行完后，将完成 RefRil 库的几项重要工作，它们是：

- 创建一个 mainLoop 工作线程，mainLoop 线程的任务是初始化 AT 模块，并监控 AT 模块，一旦 AT 模块被关闭，则会重新初始化 AT 模块。
- AT 模块内部会创建一个工作线程 readerLoop，该线程的作用是从串口设备中读取信息，也就是直接和 BP 打交道。
- mainLoop 通过向 Rild 提交超时任务，完成了对 Modem 的初始化工作。

在 Rild 的 main 函数中还剩下最后一个关键函数 RIL_register 没有分析，下面来看看它。

9.3.4 RIL_register 分析

1. 建立对外通信的链路

RIL_register 函数将创建两个监听端 socket，它们的名字分别是：

- “rild”：这个 socket 用来和 Java 层的应用通信。这一点与 Vold 中的 MountService 类似。
- “rild-debug”：这个 socket 用来接收测试程序的测试命令。

下面来看 RIL_register 函数的代码，如下所示：

[-->Ril.cpp]

```
extern "C" void RIL_register (const RIL_RadioFunctions *callbacks) {
   //RIL_RadioFunctions 结构体由 RefRil 库输出。
    int ret;
    int flags;

    ..... // 版本检测

    if (s_registerCalled > 0) {
        return;
    }

    // 拷贝这个结构体的内容到 s_callbacks 变量中。
    memcpy(&s_callbacks, callbacks, sizeof (RIL_RadioFunctions));

    s_registerCalled = 1;

    //Rild 定义了一些 Command，这里做一个小小的检查。
    for (int i = 0; i < (int)NUM_ELEMS(s_commands); i++) {
```

```
        assert(i == s_commands[i].requestNumber);
    }

    for (int i = 0; i < (int)NUM_ELEMS(s_unsolResponses); i++) {
        assert(i + RIL_UNSOL_RESPONSE_BASE
                == s_unsolResponses[i].requestNumber);
    }

    ......

    // start listen socket

#if 0
    ......
#else
    //SOCKET_NAME_RIL 的值为 "Ril"，这个 socket 由 init 进程根据 init.rc 的配置创建。
    s_fdListen = android_get_control_socket(SOCKET_NAME_RIL);
    ......
   // 监听
    ret = listen(s_fdListen, 4);
    ......
#endif

/*
构造一个非超时任务，处理函数是 listenCallback。这个任务会保存在监控表中，一旦它的 FD
可读就会导致 eventLoop 的 select 函数返回。根据前面的介绍可知，listen 端的 socket
可读表示有客户 connect 上。由于该任务的 persist 被设置为 false，待 listenCallback
处理完后，这个任务就会从监控表中移除。也就是说下一次 select 的 readFDs 中将不会有
这个监听 socket 了，这表明 Rild 只支持一个客户端的连接。
*/
    ril_event_set (&s_listen_event, s_fdListen, false,
                listenCallback, NULL);
    // 触发 eventLoop 工作。
    rilEventAddWakeup (&s_listen_event);

#if 1
    /*
      Rild 为了支持调试，还增加了一个 Ril_debug 的 socket，这个 socket 专门用于
      测试程序发送测试命令。
   */
    s_fdDebug = android_get_control_socket(SOCKET_NAME_RIL_DEBUG);
    ret = listen(s_fdDebug, 4);

     ......
   // 添加一个非超时任务，该任务对应的处理函数是 debugCallback，它是专门用来处理测试命令的。
    ril_event_set (&s_debug_event, s_fdDebug, true,
                debugCallback, NULL);
    rilEventAddWakeup (&s_debug_event);
#endif
}
```

根据上面的分析可知，如果有一个客户端 connect 上 Rild，eventLoop 就会被触发，并且对应的处理函数 listenCallback 会被调用，下面就去看看这个函数的实现，代码如下所示：

[-->Ril.cpp]

```
static void listenCallback (int fd, short flags, void *param) {
    int ret;
    int err;
    int is_phone_socket;
    RecordStream *p_rs;

    struct sockaddr_un peeraddr;
    socklen_t socklen = sizeof (peeraddr);

    struct ucred creds;
    socklen_t szCreds = sizeof(creds);

    struct passwd *pwd = NULL;

    // 接收一个客户端的连接，并将返回的 socket 保存在 s_fdCommand 中。
    s_fdCommand = accept(s_fdListen, (sockaddr *) &peeraddr, &socklen);

    ......

    errno = 0;
    is_phone_socket = 0;// 权限控制，判断连接的客户端有没有对应的权限。
    ......// 如果没有对应的权限则中止后面的流程。
    // 设置这个 socket 为非阻塞，所以后续的 send/recv 调用都不会阻塞。
    ret = fcntl(s_fdCommand, F_SETFL, O_NONBLOCK);
    ......
   /*
      p_rs 为 RecordStream 类型，它内部会分配一个缓冲区来存储客户端发来的数据，
      这些都是 socket 编程常用的做法。
   */
    p_rs = record_stream_new(s_fdCommand, MAX_COMMAND_BYTES);
    /*
     构造一个新的非超时任务，这样在收到来自客户端的数据后就会由 eventLoop 调用对应的
     处理函数 processCommandsCallback 了。
    */
    ril_event_set (&s_commands_event, s_fdCommand, 1,
                   processCommandsCallback, p_rs);
    rilEventAddWakeup (&s_commands_event);
    onNewCommandConnect();// 作一些后续处理，有兴趣的读者可以看看。
}
```

2. 关于 RIL_register 的总结

RIL_register 函数的主要功能是初始化了两个用来和外部进程通信的 socket，并且向 eventLoop 添加了对应的任务。

至此，Rild 的 main 函数就都分析完了。下面对 main 函数进行总结。

9.3.5　关于 Rild main 函数的总结

前面所有的内容都是在 main 函数中处理的，下面给出 main 函数执行后的结果，如图 9-8 所示：

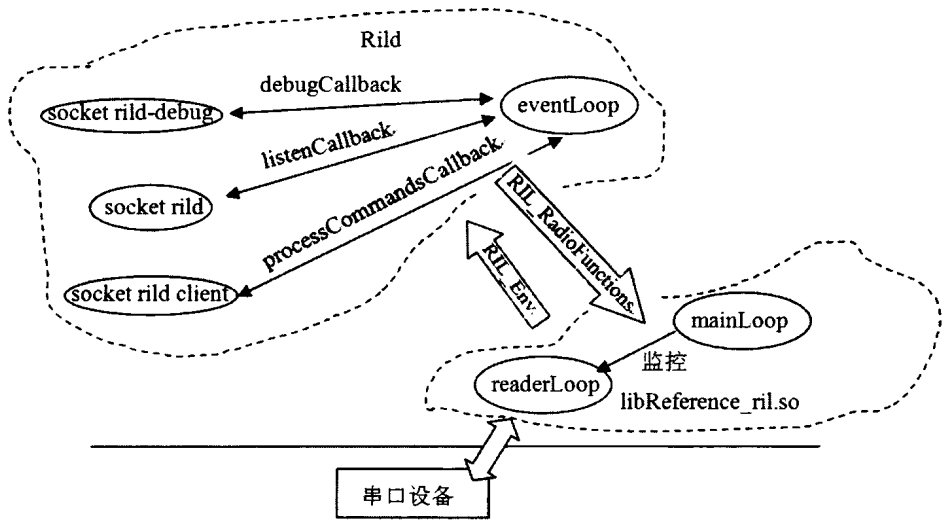


图 9-8　Rild main 函数执行后的结果示意图

其中：

- Rild 和 RefRil 库的交互通过 RIL_Env 和 RIL_RadioFunctions 这两个结构体来完成。
- Rild 的 eventLoop 处理任务。对于来自客户端的任务，eventLoop 调用的处理函数是 processCommandsCallback。
- RefRil 库的 readerLoop 用来从串口设备中读取数据。
- RefRil 库中的 mainLoop 用来监视 readerLoop。

上图画出的模块都是静态的，前面提到的异步请求 / 处理的工作方式并不能体现出来。那么来分析一个实例，看看这些模块之间是如何配合与联动的。

9.3.6　Rild 实例分析

其实，Rild 没什么难度，相信见识过 Audio 和 Surface 系统的读者都会有同感。但 Java 层的 Phone 应用及相关的 Telephony 模块却相当复杂，这里不去讨论 Phone 的实现，而是通过实例来分析一个电话是如何拨打出去的。这个例子和 Rild 有关的东西比较简单，但在分析代码的路途上，读者可以领略到 Java 层 Phone 代码的复杂。

1. 创建 Phone

Android 支持 GSM 和 CDMA 两种 Phone，到底创建哪种 Phone 呢？来看 PhoneApp.java

是怎么做的：

[-->PhoneApp.java]

```
public void onCreate() {
        ......
  if (phone == null) {
      // 创建一个 Phone，这里使用了设计模式中的 Factory（工厂）模式。
      PhoneFactory.makeDefaultPhones(this);
      phone = PhoneFactory.getDefaultPhone();
      ......
}
```

工厂模式的好处在于，将 Phone（例如代码中的 GSMPhone 或 CDMAPhone）创建的具体复杂过程屏蔽起来了，因为用户只关心工厂的产出物 Phone，而不关心创建过程。通过工厂模式可降低使用者和创建者代码之间的耦合性，即使以后增加 TDPhone，使用者也不需要修改太多的代码。

下面来看这个 Phone 工厂，代码如下所示：

[-->PhoneFactory.java]

```
public static void makeDefaultPhones(Context context) {
    makeDefaultPhone(context);// 调用 makeDefaultPhone 函数，直接去看看。
}

public static void makeDefaultPhone(Context context) {
     synchronized(Phone.class) {
      ......
     // 根据系统设置获取通信网络的模式。
     int networkMode = Settings.Secure.getInt(context.getContentResolver(),
             Settings.Secure.PREFERRED_NETWORK_MODE, preferredNetworkMode);
     int cdmaSubscription =
               Settings.Secure.getInt(context.getContentResolver(),
               Settings.Secure.PREFERRED_CDMA_SUBSCRIPTION,
               preferredCdmaSubscription);

      //RIL 这个对象就是 rild socket 的客户端，AT 命令由它发送给 Rild。
      sCommandsInterface = new RIL(context, networkMode,cdmaSubscription);

       int phoneType = getPhoneType(networkMode);
       if (phoneType == Phone.PHONE_TYPE_GSM) {
         // 先创建 GSMPhone，然后创建 PhoneProxy，这里使用了设计模式中的 Proxy 模式。
           sProxyPhone = new PhoneProxy(new GSMPhone(context,
               sCommandsInterface, sPhoneNotifier));
         } else if (phoneType == Phone.PHONE_TYPE_CDMA) {
          // 创建 CDMAPhone。
          sProxyPhone = new PhoneProxy(new CDMAPhone(context,
                       sCommandsInterface, sPhoneNotifier));
```

```
            }
              sMadeDefaults = true;
            }
        }
    }
```

假设创建的是 GSMPhone，makeDefaultPhones 函数将返回 PhoneProxy 对象，不过这是一个代理 Phone，具体工作还是会由 GSMPhone 完成。

Phone 创建完后，就要拨号了。

2. Dial 拨号

Phone 应用提供了一个 PhoneUtils 类，最终的拨号是由它完成的，代码如下所示：

[-->PhoneUtils.java]

```
static int placeCall(Phone phone, String number, Uri contactRef) {
   int status = CALL_STATUS_DIALED;
   try {
   // 调用 Phone 的 dial 函数，这个 Phone 的真实类型是 PhoneProxy，number 就是电话号码。
      Connection cn = phone.dial(number);
      ......
      }
     ......
}
```

前面说过，PhoneProxy 代理的对象是 GSMPhone，直接去看它的 dial 函数，如下所示：

[-->GSMPhone.java]

```
public Connection
    dial(String dialString) throws CallStateException {
        return dial(dialString, null);// 调用另外一个 dial 函数
    }
public Connection dial (String dialString, UUSInfo uusInfo)
                    throws CallStateException {
   String newDialString = PhoneNumberUtils.stripSeparators(dialString);
   if (mmi == null) {
       //mCT 是 GsmCallTrack 类型，调用它的 dial 函数。
      return mCT.dial(newDialString, uusInfo);
    } else if (mmi.isTemporaryModeCLIR()) {
      return mCT.dial(mmi.dialingNumber, mmi.getCLIRMode(), uusInfo);
    }
......
}
```

又调用 GSMCallTrack 的 dial 函数，它的代码如下所示：

[-->GSMCallTrack.java]

```
Connection dial (String dialString, int clirMode, UUSInfo uusInfo)
throws CallStateException {
      ......
// 构造一个 GsmConnection 对象。
pendingMO = new GsmConnection(phone.getContext(), dialString,
                              this, foregroundCall);
hangupPendingMO = false;
/*
cm 对象的真实类型就是我们前面提到的 RIL 类，它实现了 CommandInterface。
下面将调用它的 dial 函数。
*/
 cm.dial(pendingMO.address, clirMode, uusInfo,
                   obtainCompleteMessage());
......
 return pendingMO;
}
```

3. Java 层 RIL 类的分析

Phone 应用是不是很复杂？从创建 Phone 开始，颇费周折才来到了 Java 层的 RIL 类，RIL 将是 Rild 中 rild socket 的唯一客户端。下面来认识一下 RIL。

（1）RIL 的架构

RIL 的构造函数的代码如下所示。

[-->RIL.java]

```
public RIL(Context context, int networkMode, int cdmaSubscription) {
        super(context);
        mCdmaSubscription  = cdmaSubscription;
        mNetworkMode = networkMode;

        // 创建一个 HandlerThread，从名字上看它是用来发送消息的。
         mSenderThread = new HandlerThread("RILSender");
         mSenderThread.start();

         Looper looper = mSenderThread.getLooper();
         mSender = new RILSender(looper);

         mReceiver = new RILReceiver();
         // 创建一个 RILReceiver 线程，从名字上看它是用来接收消息的。
         mReceiverThread = new Thread(mReceiver, "RILReceiver");
         mReceiverThread.start();
         ......
}
```

那么和 Rild 中 rild socket 通信的 socket 是在哪创建的呢？答案是在接收线程中，其代码如下所示：

[-->RIL.java]

```
class RILReceiver implements Runnable {
        byte[] buffer;
        ......
        public void
        run() {
            int retryCount = 0;
            try {for (;;) {
                LocalSocket s = null;
                LocalSocketAddress l;
                try {
                    s = new LocalSocket();
                    l = new LocalSocketAddress(SOCKET_NAME_RIL,
                            LocalSocketAddress.Namespace.RESERVED);
                    // 和 Rild 进行连接。
                    s.connect(l);
                  ......
                }
               mSocket = s;
               int length = 0;
                try {
                    InputStream is = mSocket.getInputStream();
                    for (;;) {
                        Parcel p;
                         // 读数据。
                        length = readRilMessage(is, buffer);
                       // 解析数据。
                        p = Parcel.obtain();
                        p.unmarshall(buffer, 0, length);
                        p.setDataPosition(0);
                        // 处理请求，以后再看。
                        processResponse(p);
                        p.recycle();
                    }
                }
                ......
        }
```

从上面的代码可知，RIL 封装了两个线程：

- mSenderThread，用来向 Rild 发送消息。
- mReceiverThread，用来从 Rild 中接收消息。

待 RIL 创建后，dial 函数该干什么呢？

（2）发送 dial 请求

dial 处理过程的代码如下所示：

[-->RIL.java]

```
public void dial(String address, int clirMode, UUSInfo uusInfo, Message result)
{
```

```
        //创建一个Java层的RIL请求包。
        RILRequest rr = RILRequest.obtain(RIL_REQUEST_DIAL, result);
        rr.mp.writeString(address);
        rr.mp.writeInt(clirMode);
        rr.mp.writeInt(0);
        if (uusInfo == null) {
             rr.mp.writeInt(0); // UUS information is absent
        } else {
             rr.mp.writeInt(1); // UUS information is present
             rr.mp.writeInt(uusInfo.getType());
             rr.mp.writeInt(uusInfo.getDcs());
             rr.mp.writeByteArray(uusInfo.getUserData());
        }
            //发送数据。
            send(rr);
        }
    private void send(RILRequest rr) {
        Message msg;
        //发送EVENT_SEND消息，由mSender这个Handler处理。
        msg = mSender.obtainMessage(EVENT_SEND, rr);
        acquireWakeLock();
        msg.sendToTarget(); //由发送线程处理。
    }
```

下面看 handleMessage 函数，代码如下所示：

[-->RIL.java]

```
    public void  handleMessage(Message msg) {
        RILRequest rr = (RILRequest)(msg.obj);//请求消息。
        RILRequest req = null;
        ......
        switch (msg.what) {
            case EVENT_SEND:
            boolean alreadySubtracted = false;
            try {
                    LocalSocket s;
                    s = mSocket; //这个mSocket就是和Rild通信的socket。
                    /*
                     执行异步请求/处理时，请求方需要将请求包保存起来，待收到完成通知后再从请求队列
                     中找到对应的那个请求包并做后续处理。请求包一般会保存请求时的上下文信息。
                     以酒店的Morning Call服务为例。假设预约了7、8、9点的服务，那么当7点钟
                     接到电话时，一看表便知道是7点的那个请求完成了，而不是8点或9点的请求完成了。
                     这个7便是请求号的标示，而且完成通知必须回传这个请求号。至于上下文信息，则
                     保存在请求包中。例如酒店会在电话中通知说7点钟要开一个会，这个开会的信息是
                     预约服务的时候由你提供给酒店的。
                     保存请求包是异步请求/处理或异步I/O中常见的做法，不过这种做法有一个
                     很明显的缺点，就是当请求量比较大的时候，会占用很多内存来保存请求包信息。
                    */
                    synchronized (mRequestsList) {
```

```
                mRequestsList.add(rr);
            }

            byte[] data;
            data = rr.mp.marshall();
            rr.mp.recycle();
            rr.mp = null;
            ......
            s.getOutputStream().write(dataLength);
            s.getOutputStream().write(data); // 发送数据。
    }
        ......
}
```

至止，应用层已经通过 RIL 对象将请求数据发送了出去。由于是异步模式，请求数据发送出去后应用层就直接返回了，而且目前还不知道处理结果。那么 Rild 是如何处理这个请求的呢？

4. Rild 处理请求

根据前面对 Rild 的分析可知，当收到客户端的数据时会由 eventLoop 调用对应的任务处理函数进行处理，而这个函数就是 processCommandsCallback。看它的代码——

（1）Rild 接收请求

Rild 接收请求的代码如下所示：

[-->Ril.cpp]

```
static void processCommandsCallback(int fd, short flags, void *param) {
    RecordStream *p_rs;
    void *p_record;
    size_t recordlen;
    int ret;

    //RecordStream 为 processCommandsCallback 的参数，里面维护了一个接收缓冲区并
    // 有对应的缓冲读写位置控制。
    p_rs = (RecordStream *)param;

    for (;;) {

        /*
            下面这个函数将从 socket 中 read 数据到缓冲区，并从缓冲区中解析命令。
            注意，该缓冲区可能累积了多条命令，也就是说，客户端可能发送了多个命令，而
            Rild 通过一次 read 就全部接收到了。这个特性是由 TCP 的流属性决定的。
            所以这里有一个 for 循环来接收和解析命令。
         */
        ret = record_stream_get_next(p_rs, &p_record, &recordlen);

        if (ret == 0 && p_record == NULL) {
            /* end-of-stream */
```

```
                break;
            } else if (ret < 0) {
                break;
            } else if (ret == 0) {
                // 处理一条命令。
                processCommandBuffer(p_record, recordlen);
            }
        }

        if (ret == 0 || !(errno == EAGAIN || errno == EINTR)) {
            ..... // 出错处理，例如 socket read 出错。
        }
    }
```

每解析出一条命令，就调用 processCommandBuffer 函数进行处理，看这个函数，如下所示：

[-->Ril.cpp]

```
static int processCommandBuffer(void *buffer, size_t buflen) {
    Parcel p;
    status_t status;
    int32_t request;
    int32_t token;
    RequestInfo *pRI;
    int ret;

    p.setData((uint8_t *) buffer, buflen);
    status = p.readInt32(&request);
    status = p.readInt32 (&token);

     ......
    //s_commands 定义了 Rild 支持的所有命令及其对应的处理函数。
    if (request < 1 || request >= (int32_t)NUM_ELEMS(s_commands)) {
        ......
        return 0;
    }

   //Rild 内部处理也是采用的异步模式，所以它也会保存请求，又分配一次内存。
    pRI = (RequestInfo *)calloc(1, sizeof(RequestInfo));
    pRI->token = token;
    //s_commands 是什么？
    pRI->pCI = &(s_commands[request]);
    // 请求信息保存在一个单向链表中。
    ret = pthread_mutex_lock(&s_pendingRequestsMutex);
    pRI->p_next = s_pendingRequests;//p_next 指向链表的后继结点。
    s_pendingRequests = pRI;
    ret = pthread_mutex_unlock(&s_pendingRequestsMutex);
    // 调用对应的处理函数。
```

```
    pRI->pCI->dispatchFunction(p, pRI);

    return 0;
}
```

上面的代码中，出现了一个 s_commands 数组，它保存了一些 CommandInfo 结构，这个结构封装了 Rild 对 AT 指令的处理函数。另外 Rild 还定义了一个 s_unsolResponses 数组，它封装了 unsolicited Response 对应的一些处理函数。这两个数组如下所示：

[-->Ril.cpp]

```
typedef struct {// 先看看 CommandInfo 的定义。
    int requestNumber; // 请求号，一个请求对应一个请求号。
    // 请求处理函数。
    void (*dispatchFunction) (Parcel &p, struct RequestInfo *pRI);
    // 结果处理函数。
    int(*responseFunction) (Parcel &p, void *response, size_t responselen);
} CommandInfo;

// 下面是 s_commands 的定义。
static CommandInfo s_commands[] = {
#include "ril_commands.h"
};
// 下面是 s_unsolResponses 的定义。
static UnsolResponseInfo s_unsolResponses[] = {
#include "ril_unsol_commands.h"  // 这个头文件读者可以自己去看看。
};
```

再来看 ril_commands.h 的定义，代码如下所示：

[-->ril_commands.h]

```
{0, NULL, NULL}, // 除了第一条外，一共定义了 103 条 CommandInfo。
{RIL_REQUEST_GET_SIM_STATUS, dispatchVoid, responseSimStatus},
 ......
{RIL_REQUEST_DIAL, dispatchDial, responseVoid}, // 打电话的处理。
 ......
{RIL_REQUEST_SEND_SMS, dispatchStrings, responseSMS}, // 发短信的处理。
 ......
```

根据上面的内容可知，在 Rild 中打电话的处理函数是 dispatchDial，它的结果处理函数是 responseVoid。

（2）Rild 处理请求

Rild 处理请求的代码如下所示：

[-->Ril.c]

```
static void dispatchDial (Parcel &p, RequestInfo *pRI) {
    RIL_Dial dial; // 创建一个 RIL_Dial 对象，它存储打电话时所需要的一些参数。
```

```
    RIL_UUS_Info uusInfo;
    int32_t sizeOfDial;
    int32_t t;
    int32_t uusPresent;
    status_t status;

    memset (&dial, 0, sizeof(dial));

    dial.address = strdupReadString(p);

    status = p.readInt32(&t);
    dial.clir = (int)t;

    ..... // 中间过程略去。
    // 调用 RIL_RadioFunctions 的 onRequest 函数，也就是向 RefRil 库发送一个请求。
    s_callbacks.onRequest(pRI->pCI->requestNumber, &dial, sizeOfDial, pRI);
    ......
    return;
}
```

下面去 RefRil 库看一下这个 onRequest 的处理，代码如下所示：

[-->Reference_Ril.c]

```
static void onRequest (int request, void *data, size_t datalen, RIL_Token t)
{
    ATResponse *p_response;
    int err;

    ......

    switch (request) {
        ......
        case RIL_REQUEST_DIAL: // 打电话处理。
            requestDial(data, datalen, t);
            break;
            ......
        case RIL_REQUEST_SEND_SMS:  // 发短信处理。
            requestSendSMS(data, datalen, t);
            break;
        default:
            RIL_onRequestComplete(t, RIL_E_REQUEST_NOT_SUPPORTED, NULL, 0);
            break;
    }
}
// 我们看看 requestDial 函数。
static void requestDial(void *data, size_t datalen, RIL_Token t)
{
    RIL_Dial *p_dial;
    char *cmd;
```

```
    const char *clir;
    int ret;

    p_dial = (RIL_Dial *)data;

    ......
   //at_send_command 将往串口设备发送这条 AT 指令。
    ret = at_send_command(cmd, NULL);

    free(cmd);
/*
  对于 dail 请求，把数据发送给串口就算完成了，所以 dial 发送完数据后直接调用
  RIL_onRequestComplete 函数来通知请求处理的结果。而有一些请求需要先由
  AT 模块的 readLoop 线程从串口中读取 BP 的处理结果后再行通知。
*/
    RIL_onRequestComplete(t, RIL_E_SUCCESS, NULL, 0);
}
```

来看 RIL_onRequestComplete 函数，这个函数的实现由 RIL_Env 提供，代码如下所示：

[-->Ril.c]

```
extern "C" void
RIL_onRequestComplete(RIL_Token t, RIL_Errno e, void *response,
                      size_t responselen) {
    RequestInfo *pRI;
    int ret;
    size_t errorOffset;

    pRI = (RequestInfo *)t;
    // 由于已经收到了请求的处理结果，这表明该请求已经完成，所以需要从请求队列中去掉这个请求。
    if (!checkAndDequeueRequestInfo(pRI)) {
    }
    ......

    if (pRI->cancelled == 0) {
        Parcel p;

        p.writeInt32 (RESPONSE_SOLICITED);
        p.writeInt32 (pRI->token);
        errorOffset = p.dataPosition();

        p.writeInt32 (e);

        if (response != NULL) {
            //dial 请求的 responseFunction 函数是 responseVoid，读者可以看这个函数。
            ret = pRI->pCI->responseFunction(p, response, responselen);
            if (ret != 0) {
```

```
                p.setDataPosition(errorOffset);
                p.writeInt32 (ret);
            }
        }

        ......
        sendResponse(p);// 将结果发送给 Java 的 RIL。
    }

done:
    free(pRI);
}
```

Rild 内部也采用了异步请求 / 处理的结构，这样做有它的道理，因为有一些请求执行的时间较长，例如在信号不好的地方搜索网络信号往往会花费较长的时间。采用异步的方式，能避免工作线程阻塞在具体的请求函数中，从而腾出手来做一些别的工作。

（3）Java RIL 处理完成的通知

Rild 将 dial 请求的结果通过 socket 发送给 Java 中的 RIL 对象。前面说过，RIL 中有一个接收线程，它收到数据后会调用 processResponse 函数进行处理，来看这个函数，如下所示：

[-->RIL.java]

```
private void  processResponse (Parcel p) {
        int type;
        type = p.readInt();
        if (type == RESPONSE_UNSOLICITED) {
            processUnsolicited (p);
        } else if (type == RESPONSE_SOLICITED) {
            processSolicited (p); //dial 是应答式的，所以走这个分支。
        }
        releaseWakeLockIfDone();
    }

private void processSolicited (Parcel p) {
    int serial, error;
    boolean found = false;

    serial = p.readInt();
    error = p.readInt();

    RILRequest rr;
    // 根据完成通知中的请求包编号从请求队列中去掉对应的请求，以释放内存。
    rr = findAndRemoveRequestFromList(serial);
    Object ret = null;
```

```
    if (error == 0 || p.dataAvail() > 0) {
        try {
            switch (rr.mRequest) {
                    ......
            // 调用 responseVoid 函数处理结果。
            case RIL_REQUEST_DIAL: ret =  responseVoid(p); break;
            ......
            if (rr.mResult != null) {
        /*
            RILReceiver 线程将处理结果投递到一个 Handler 中，而这个 Handler 属于
            另外一个线程，也就是处理结果最终将交给另外一个线程做后续处理，例如切换界面显示等工作，
            具体内容就不再详述了。为什么要投递到别的线程进行处理呢？因为 RILReceiver
            负责从 Rild 中接收数据，而这个工作是比较关键的，所以这个线程除了接收数据外，最好
            不要再做其他的工作了。
       */
            AsyncResult.forMessage(rr.mResult, ret, null);
            rr.mResult.sendToTarget();
        }
        rr.release();
    }
```

实例分析就到此为止，相信读者已经掌握了 Rild 的精髓。

9.3.7　关于 Rild 的总结

从整体来说，Rild 并不复杂，其程序框架非常清晰，它和其他系统唯一不同的是，Rild 采用了异步请求 / 处理的工作方式，而异步方式对代码编写能力的要求是几种 I/O 模式中最高的。读者在阅读 Rild 这一节内容时，要牢记异步处理模式的流程。

另外，和 Rild 对应的 Java 中的 Phone 程序非常复杂，个人甚至觉得有些过于复杂了。读者如有兴趣，可以看看 Phone 的代码，写得很漂亮，其中也使用了很多设计模式，但我觉得这个 Phone 应用在设计上还有很多地方可以改进。这一点，在本章的拓展思考内容中再来讨论。

9.4　拓展思考

本章的拓展思考包括嵌入式系统的存储知识介绍，以及 Phone 应用的改进探讨两部分。

9.4.1　嵌入式系统的存储知识介绍

用 adb shell 登录到我的 G7 手机上，然后用 mount 查看信息后，会得到如图 9-9 所示的结果：

```
/dev/block/mtdblock3 /system yaffs2 rw,relatime 0 0
/dev/block/mtdblock4 /cache yaffs2 rw,nosuid,nodev,relatime 0 0
tmpfs /app-cache tmpfs rw,relatime,size=8192k 0 0
/dev/block/mtdblock5 /mnt/asec/mtddata yaffs rw,nosuid,nodev,rel
/dev/block/mmcblk0p2 /data ext4 rw,nosuid,nodev,noatime,nodirati
/dev/block/mmcblk0p3 /mnt/asec/extdata ext4 rw,nosuid,nodev,noat
```

图 9-9　mount 命令的执行结果

其中，可以发现系统的几个重要分区，例如 /system 对应的设备是 mtdblock3，那么 mtdblock 是什么呢？

1.MTD 的介绍

Linux 系统提供了 MTD（Memory Technology Device，内存技术设备）系统来建立针对 Flash 设备的统一、抽象的接口，也就是说，有了 MTD，就可以不用考虑不同 Flash 设备带来的差异了，这一点和 FBD（FrameBuffer Device）的作用很类似。下面看 Linux MTD 的系统层次图，如图 9-10 所示。

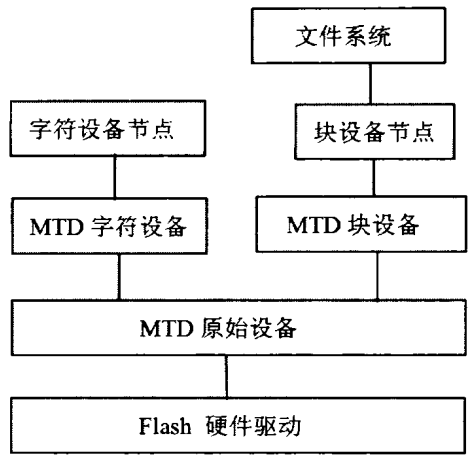


图 9-10 Linux MTD 系统层次图

从上图中可以看出：

- MTD 将文件系统与底层的 Flash 存储器进行了隔离，这样应用层就无须考虑真实的硬件情况了。
- 图中的 mtdblock 表示 MTD 块设备。

有了 MTD 后，就不用关心 Flash 是 NOR 还是 NAND 了。另外，我们从图 9-10 “mount 命令的执行结果”中还可看见 mount 指定的文件系统中有一个 yaffs2，它又是什么呢？

2.Flash 文件系统

先来说说 Flash 的特性。常见的文件系统（例如 FAT32、NTFS、Ext2 等）是无法直接用在 Flash 设备上的，因为无法重复地在 Flash 的同一块存储位置上做写入操作（必须事先擦除该块后才能写入）。为了能够在 Flash 设备上使用这些文件系统，必须透过一个层转换层（Translation Layer）将逻辑块地址对应到 Flash 存储器的物理地址上，以便系统能把 Flash 当作普通的磁盘处理，可称这一层为 FTL（Flash Translation Layer）。Flash 转换层的示意图如图 9-11 所示：

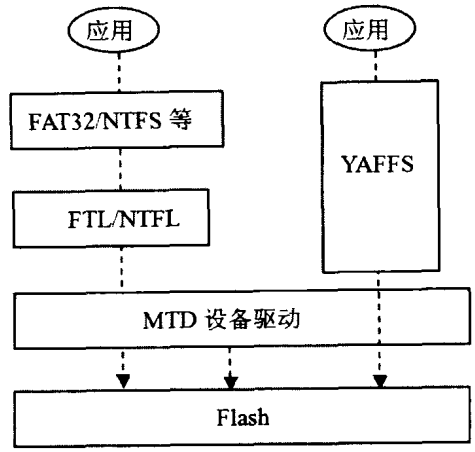


图 9-11　FTL 和 NFTL

从上图中可以看到：

- 如果想使用 FAT32 或 NTFS 文件系统，必须通过 FTL 或 NTFL 进行转换，其中 FTL 针对 NOR Flash，而 NTFL 针对 NAND Flash。
- 尽管有了 FTL，但毕竟多了一层处理，这样对 I/O 效率的影响较大，所以人们开发了专门针对 Flash 的文件系统，其中 YAFFS 就是应用比较广泛的一种。

YAFFS 是 Yet Another Flash File System 的简称，目前有 YAFFS 和 YAFFS2 两个版本。这两个版本的主要区别是，YAFFS2 可支持大容量的 NADN Flash，而 YAFFS 只支持页的大小为 512 字节的 NAND Flash。YAFFS 使用 OOB（Out Of Bind）来组织文件的结构信息，所以在 Vold 代码中，可以见到 OOB 相关的字样。

说明　*关于嵌入式存储方面的知识就介绍到这里。有兴趣深入了解的读者可阅读有关驱动开发方面的书籍。*

---

3. Android mtd 设备介绍

这里以我的 HTC G7 手机为例，分析 Android 系统中 MTD 设备的使用情况。

通过 adb cat /proc/mtd，得到图 9-12 所示的 MTD 设备的使用情况：

```
dev:    size   erasesize  name
mtd0: 00100000 00040000 "misc"
mtd1: 00480000 00040000 "recovery"
mtd2: 00340000 00040000 "boot"
mtd3: 0fa00000 00040000 "system"
mtd4: 02800000 00040000 "cache"
mtd5: 096c0000 00040000 "userdata"
```

图 9-12　G7 MTD 设备的使用情况

这几个设备对应存储空间的大小和作用如下：

- MTD0，主要用于存储开机画面。此开机画面在 Android 系统启动前运行，由 Bootloader 调用，大小为 1MB。
- MTD1，存储恢复模式的镜像，大小为 4.5MB。
- MTD2，存储 kernel 镜像，大小为 3.25MB。
- MTD3，存储 system 镜像，该分区挂载在 /system 目录下，大小为 250MB。
- MTD4，缓冲临时文件，该分区挂载在 /cache 目录下，大小为 40MB。
- MTD5，存储用户安装的软件和一些数据，我的 G7 把这个设备挂载在 /mnt/asec/mtddata 目录下，大小为 150.75MB。

注意 上面的设备和挂载点与具体的机器及所刷的 ROM 有关。

9.4.2 Rild 和 Phone 的改进探讨

在使用 G7 的时候，最不满意的就是，群发短信的速度太慢，而且有时会出现 ANR 的情况，就 G7 的硬件配置来说，按理不至于发生这种情况。原因究竟何在？通过对 Rild 和 Phone 的分析，我认为和 Rild 及 Phone 的设计有些许关系，下面来探讨一下这个问题。

下面以 Rild 和 RefRil 库为例，来分析 Rild 和 Phone 的设计上有哪些特点和问题。注意，这里将短信程序和 Phone 程序统称为 Phone。

- Rild 没有使用 Binder 通信机制和 Phone 进行交互，这一点虽感觉较奇怪，不过也好理解，因为实现一个用 Socket 进行 IPC 通信的架构，比用 Binder 要简单，至少代码量要少一些。
- Rild 使用了异步请求 / 处理的模式，这种模式对于 Rild 来说是合适的。因为一个请求的处理可能耗时很长，另外一点就是 Rild 会收到来自 BP 的 unsolicited Response。
- Phone 这个应用也使用了异步模式。其实，这也好理解，因为 Phone 和 Rild 采用了 Socket 进行通信，如果把 Phone 中的 Socket 看作是 Rild 中的串口设备，你就发现这个 Phone 竟然是 Rild 在 Java 层的翻版。这样设计有问题吗？其明显缺陷就是一个请求消息在 Java 层的 Phone 中要保存一个，传递到 Rild 中还要保存一个。另外，Phone 和 Rild 交互的是 AT 命令。这种直接使用 AT 命令的方式，对以后的扩展和修改都会造成不少麻烦。
- 再来看群发短信问题。群发短信的实现，就是同一个信息发送到不同的号码上。对于目前 Phone 的实现而言，就是一个 for 循环中调用一个发送函数，参数中仅号码不同，而短信内容是一样的。这种方式是否太浪费资源了呢？假设群发目标人数为 200 个，那么 Java 层要保存 200 个请求信息，而 Rild 层也要保存 200 个请求信息。并且 Rild 每处理一个命令就会来一个完成通知。对于群发短信功能来说，本人更关心的是，所

有短信发送完后的统一结果，而非单条短信发送的结果。

以上是我关于 Rild 和 Phone 设计特点的一些总结。如果由我来实现 Phone，我会怎么做呢？这里将自己的一些想法与读者分享一下。

- 在 Phone 和 Rild 的进程间通信上，将使用 Binder 机制。这样，首先需定义一个 Phone 和 Rild 通信的接口，这个接口的内容和 Rild 提供的服务有关，例如定义一个 dial 函数，定义一个 sendSMS 函数。除此之外，需要定义 Rild 向 Phone 回传 Response 的通知接口。也就是说，Rild 直接利用 Binder 回调 Phone 进程中的函数，把结果传过去。采用 Binder 至少有三个好处。第一，Phone 和 Rild 的交互基于接口函数，就不用在 Phone 中做 AT 命令的转换了，另外基于接口的交互使得程序的可扩展性也得到了提高。第二，可以定义自己的函数，例如提供一个函数用来实现群发短信功能，通过这个函数可将一条短信内容和多个群发目标打包传递给 Rild，然后由 Rild 自己去解析成多条 AT 命令进行处理。第三，Phone 代码将会被精简不少。
- 在内存使用方面，有可能 Phone 和 Rild 都需保存请求，这时可充分利用共享内存的优势，将请求信息保存在共享内存中。这样，可减少一部分内存的使用。另外，这块内存中存储的信息可能需要使用一定的结构来组织，例如采用优先级队列。这样，那些优先级高的请求就能首先得到处理了。

以上是本人在研究 Rild 和 Phone 代码过程中一些不成熟的想法，希望能引起读者共同的思考。读者还可以参考网上名为《RIL 设计思想解析》的一篇文章。

9.5　本章小结

本章对 Vold 和 Rild 两个重要的 daemon 程序进行了分析。其中：

- Vold 负责 Android 平台上存储系统的管理和控制。重点关注 Vold 两方面的内容，一是它如何接收和处理来自内核的 Uevent 事件，二是如何处理来自 Java 层 MountService 的请求。
- Rild 是 Android 平台上的射频通信控制中枢，接打电话、收发短信等都需要 Rild 的参与。本章对 Rild 的架构进行了重点分析，尤其对异步请求 / 响应的知识进行了较详细的介绍。另外，还分析了 Phone 中拨打电话的处理流程。

本章的拓展内容中，首先介绍了嵌入式系统中和存储、文件系统相关的知识。另外，还探讨了 Phone 和 Rild 设计的特点，以及可以改进的某些地方。



第10章 深入理解MediaScanner

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。

- MediaProvider.java(*packages/providers/MediaProvider/MediaProvider.java*)
- MediaScannerReceiver.java(*packages/providers/MediaProvider/MediaScannerReceiver.java*)
- MediaScannerService.java(*packages/providers/MediaProvider/MediaScannerService.java*)
- MediaScanner.java(*framework/base/media/java/com/android/media/MediaScanner.java*)
- MediaThumbRequest.java(*packages/providers/MediaProvider/MediaThumbRequest.java*)
- android_media_MediaScanner.cpp(*framework/base/media/jni/android_media_MediaScanner.cpp*)
- MediaScanner.cpp(*framework/base/media/libmedia/MediaScanner.cpp*)
- PVMediasScanner.cpp(*external/opencore/android/PVMediasScanner.cpp*)

10.1　概述

多媒体系统，是 Android 平台中非常庞大的一个系统。不过由于篇幅所限，本章只介绍多媒体系统中的重要一员 MediaScanner。MediaScanner 有什么用呢？可能有些读者还不是很清楚。MediaScanner 和媒体文件扫描有关，例如，在 Music 应用程序中见到的歌曲专辑名、歌曲时长等信息，都是通过它扫描对应的歌曲而得到的。另外，通过 MediaStore 接口查询媒体数据库，从而得到系统中所有媒体文件的相关信息也和 MediaScanner 有关，因为数据库的内容就是由 MediaScanner 添加的。所以 MediaScanner 是多媒体系统中很重要的一部分。

说明　伴随着 Android 的成长，多媒体系统也发生了非常大的变化。对于开发者来说一个非常好的消息是，那个令人极度郁闷的 OpenCore 从 Android 2.3 开始，终于有被干掉的可能了。从此，也将迎来 Stagefright 时代。但 Android 2.2 在很长一段时间内还会存在，所以希望以后能有机会深入地剖析一下这个 OpenCore。

下面就来分析媒体文件扫描的工作原理。

10.2　android.process.media 分析

多媒体系统的媒体扫描功能，是通过一个 APK 应用程序提供的，它位于 package/providers/MediaProvider 目录下。通过分析 APK 的 Android.mk 文件可知，该 APK 运行时指定了一个进程名，如下所示：

```
application android:process=android.process.media
```

原来，通过 ps 命令经常看到的进程就是它啊！另外，从这个 APK 程序所处的 package\providers 目录也可以知道，它还是一个 ContentProvider。事实上从 Android 应用程序的四大组件来看，它使用了其中的三个组件：

- MediaScannerService（从 Service 派生）模块负责扫描媒体文件，然后将扫描得到的信息插入到媒体数据库中。
- MediaProvider（从 ContentProvider 派生）模块负责处理针对这些媒体文件的数据库操作请求，例如查询、删除、更新等。
- MediaScannerReceiver（从 BroadcastReceiver 派生）模块负责接收外界发来的扫描请求。也就是 MS 对外提供的接口。

注意　除了支持通过广播发送扫描请求外，MediaScannerService 也支持利用 Binder 机制跨进程调用扫描函数。这部分内容将在本章的拓展内容中介绍。

本章仅关注 android.process.media 进程中的 MediaScannerService 和 MediaScannerReceiver 模块，为了书写方便，下文将这两个模块简称为 MSS 和 MSR，另外将 MediaScanner 简称

MS，将 MediaProvider 简称 MP。

下面，开始分析 android.process.media 中和媒体文件扫描相关的工作流程。

10.2.1 MSR 模块分析

MSR 模块的核心类 MediaScannerReceiver 从 BroadcastReceiver 派生，它是专门用来接收广播的，那么它感兴趣的广播有哪几种呢？其代码如下所示：

[-->MediaScannerReceiver.java]

```
public class MediaScannerReceiver extends BroadcastReceiver
{
    private final static String TAG = "MediaScannerReceiver";
    @Override  //MSR 在 onReceive 函数中处理广播。
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        Uri uri = intent.getData();
        // 一般手机外部存储的路径是 /mnt/sdcard。
        String externalStoragePath =
                    Environment.getExternalStorageDirectory().getPath();

        // 为了简化书写，所有 Intent 的 ACTION_XXX_YYY 字串都会简写为 XXX_YYY。
        if (action.equals(Intent.ACTION_BOOT_COMPLETED)) {
             // 如果收到 BOOT_COMPLETED 广播，则启动内部存储区的扫描工作，内部存储区
            // 实际上扫描的是 /system/media 目录，这里存储了系统自带的铃声等媒体文件。
             scan(context, MediaProvider.INTERNAL_VOLUME);
        } else {
            if (uri.getScheme().equals("file")) {
                 String path = uri.getPath();
              /*
                注意下面这个判断，如果收到 MEDIA_MOUNTED 消息，并且外部存储挂载的路径
                和 "/mnt/sdcard" 一样，则启动外部存储也就是 SD 卡的扫描工作。
                */
                if (action.equals(Intent.ACTION_MEDIA_MOUNTED) &&
                        externalStoragePath.equals(path)) {
                    scan(context, MediaProvider.EXTERNAL_VOLUME);
                } else if (action.equals(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE)
                        && path != null
                        && path.startsWith(externalStoragePath + "/")) {
                    /*
                      外部应用可以发送 MEDIA_SCANNER_SCAN_FILE 广播让 MSR 启动单个文件
                      的扫描工作。注意这个文件必须位于 SD 卡上。
                    */
                scanFile(context, path);
                }
            }
        }
    }
```

从上面的代码中发现了 MSR 接收的三种请求，也就是说，它对外提供三个接口函数：

- 接收 BOOT_COMPLETED 请求，这样 MSR 会启动内部存储区的扫描工作，注意这个内部存储区实际上是 /system/media 这个目录。
- 接收 MEDIA_MOUNTED 请求，并且该请求携带的外部存储挂载点路径必须是 /mnt/sdcard，通过这种方式 MSR 会启动外部存储区也就是 SD 卡的扫描工作，扫描目标是文件夹 /mnt/sdcard。
- 接收 MEDIA_SCANNER_SCAN_FILE 请求，并且该请求必须是 SD 卡上的一个文件，即文件路径须以 /mnt/sdcard 开头，这样，MSR 会启动针对这个文件的扫描工作。

不知读者是否注意到，MSR 和跨 Binder 调用的接口（在本章拓展内容中将介绍）都不支持对目录的扫描（除了 SD 卡的根目录外）。实现这个功能并不复杂，有兴趣的读者可自行完成该功能，如果方便，请将自己实现的代码与大家共享。

大部分的媒体文件都已放在 SD 卡上了，那么来看收到 MEDIA_MOUNTED 请求后 MSR 的工作。还记得第 9 章中对 Vold 的分析吗？这个 MEDIA_MOUNTED 广播就是由 MountService 发送的，一旦有 SD 卡被挂载，MSR 就会被这个广播唤醒，接着 SD 卡的媒体文件就会被扫描了。真是一气呵成！

SD 卡根目录扫描时调用的函数 scan 的代码如下：

[-->MediaScannerReceiver.java]

```
private void scan(Context context, String volume) {
        //volume 的值为 /mnt/sdcard。
        Bundle args = new Bundle();
        args.putString("volume", volume);
        // 启动 MSS。
        context.startService(
                new Intent(context, MediaScannerService.class).putExtras(args));
    }
```

scan 将启动 MSS 服务。下面来看 MSS 的工作。

10.2.2　MSS 模块分析

MSS 从 Service 派生，并且实现了 Runnable 接口。下面是它的定义：

[-->MediaScannerService.java]

```
MediaScannerService extends Service implements Runnable
//MSS 实现了 Runnable 接口，这表明它可能会创建工作线程。
```

根据 SDK 中对 Service 生命周期的描述可知，Service 刚创建时会调用 onCreate 函数，接着就是 onStartCommand 函数，之后外界每调用一次 startService 都会触发 onStartCommand 函数。接下来去了解一下 onCreate 函数及 onStartCommand 函数。

1. onCreate 分析

onCreate 函数的代码如下所示（这是 MSS 被系统创建时调用的，在它的整个生命周期内仅调用一次）：

[-->MediaScannerService.java]

```
public void onCreate(){
   // 获得电源锁，防止在扫描过程中休眠。
   PowerManager pm = (PowerManager)getSystemService(Context.POWER_SERVICE);
   mWakeLock = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, TAG);
   // 扫描工作是一个漫长的工程，所以这里单独创建一个工作线程，线程函数就是
   //MSS 实现的 Run 函数。
    Thread thr = new Thread(null, this, "MediaScannerService");
    thr.start();
}
```

onCreate 将创建一个工作线程，代码如下所示：

```
public void run()
    {
        /*
          设置本线程的优先级，这个函数的调用有很重要的作用，因为媒体扫描可能会耗费很长的
          时间，如果不调低优先级的话，CPU 将一直被 MSS 占用，这会导致用户感觉系统变得很慢。
         */
        Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND +
                                  Process.THREAD_PRIORITY_LESS_FAVORABLE);
        Looper.prepare();

        mServiceLooper = Looper.myLooper();
        /*
          创建一个 Handler，以后发送给这个 Handler 的消息都会由工作线程处理。
          这一部分内容已在第 5 章 Handler 中分析过了。
        */
        mServiceHandler = new ServiceHandler();

        Looper.loop();
}
```

onCreate 后，MSS 将会创建一个带消息处理机制的工作线程，那么消息是怎么投递到这个线程中的呢？

2. onStartCommand 分析

还记得 MSR 的 scan 函数吗？如下所示：

[-->MediaScannerReceiver.java::scan 函数]

```
context.startService(
            new Intent(context, MediaScannerService.class).putExtras(args));
```

其中 Intent 包含了目录扫描请求的目标 /mnt/sdcard。这个 Intent 发出后，最终由 MSS 的 onStartCommand 收到并处理，其代码如下所示：

[-->MediaScannerService.java]

```
@Override
 public int onStartCommand(Intent intent, int flags, int startId)
 {
     /*
       等待 mServiceHandler 被创建。耕耘这段代码的码农难道不知道
       HandlerThread 这个类吗？不熟悉它的读者请再阅读第 5 章的 5.4 节。
     */
     while (mServiceHandler == null) {
            synchronized (this) {
                try {
                    wait(100);
                } catch (InterruptedException e) {
                }
            }
        }
       ......
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent.getExtras();
        // 往这个 Handler 投递消息，最终由工作线程处理。
        mServiceHandler.sendMessage(msg);
        ......
}
```

onStartCommand 将把扫描请求信息投递到工作线程中去处理。

3. 处理扫描请求

扫描请求由 ServiceHandler 的 handleMessage 函数处理，其代码如下所示：

[-->MediaScannerService.java]

```
private final class ServiceHandler extends Handler
{
     @Override
     public void handleMessage(Message msg)
        {
            Bundle arguments = (Bundle) msg.obj;
            String filePath = arguments.getString("filepath");

            try {
                  ......
                } else {
                    String volume = arguments.getString("volume");
                    String[] directories = null;
```

```
                if (MediaProvider.INTERNAL_VOLUME.equals(volume)) {
                 // 如果是扫描内部存储的话，实际上扫描的目录是 /system/media。
                  directories = new String[] {
                            Environment.getRootDirectory() + "/media",
                     };
                }
                else if (MediaProvider.EXTERNAL_VOLUME.equals(volume)) {
                  // 扫描外部存储，设置扫描目标位置 /mnt/sdcard。
                   directories = new String[]{
                      Environment.getExternalStorageDirectory().getPath()};
                }
                if (directories != null) {
                /*
                  调用 scan 函数开展文件夹扫描工作，可以一次为这个函数设置多个目标文件夹，
                  不过这里只有 /mnt/sdcard 一个目录。
                */
                scan(directories, volume);
                 ......
                stopSelf(msg.arg1);
            }
    }
```

下面单独用一小节来分析这个 scan 函数。

4.MSS 的 scan 函数分析

scan 的代码如下所示：

[-->MediaScannerService.java]

```
private void scan(String[] directories, String volumeName) {
    mWakeLock.acquire();

    ContentValues values = new ContentValues();
    values.put(MediaStore.MEDIA_SCANNER_VOLUME, volumeName);
    //MSS 通过 insert 这个特殊 Uri 让 MediaProvider 做一些准备工作。
    Uri scanUri = getContentResolver().insert(
                                    MediaStore.getMediaScannerUri(), values);

   Uri uri = Uri.parse("file://" + directories[0]);
   // 向系统发送一个 MEDIA_SCANNER_STARTED 广播。
   sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_STARTED, uri));
        try {
           //openDatabase 函数也是通过 insert 这个特殊 Uri 让 MediaProvider 打开数据库的。
            if (volumeName.equals(MediaProvider.EXTERNAL_VOLUME)) {
                openDatabase(volumeName);
            }
        // 创建媒体扫描器，并调用它的 scanDirectories 函数扫描目标文件夹
        MediaScanner scanner = createMediaScanner();
          scanner.scanDirectories(directories, volumeName);
        }
```

```
            ......
            // 通过特殊 Uri 让 MediaProvider 做一些清理工作。
            getContentResolver().delete(scanUri, null, null);
            // 向系统发送 MEDIA_SCANNER_FINISHED 广播。
            sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_FINISHED, uri));

            mWakeLock.release();
    }
```

上面的代码中，比较复杂的是 MSS 和 MP 的交互。除了后文即将看到的正常数据库操作外，MSS 还经常会使用一些特殊的 Uri 来做数据库操作，而 MP 针对这些 Uri 会做一些特殊处理，例如打开数据库文件等。

注意　本章不打算对 MediaProvider 做过多的讨论，这部分知识对那些读完前 9 章的读者来说，应该不是什么难题。如果有可能，请读者自己整理 MediaProvider 的工作流程，然后提供给大家一起学习、探讨。

来看 MSS 中创建媒体扫描器的函数 createMediaScanner，如下所示：

```
private MediaScanner createMediaScanner() {
        // 下面这个 MediaScanner 在 framework/base/ 中，稍后再分析。
        MediaScanner scanner = new MediaScanner(this);
        // 获取当前系统使用的区域信息，扫描的时候将把媒体文件中的信息转换成当前系统使用的语言。
        Locale locale = getResources().getConfiguration().locale;
        if (locale != null) {
            String language = locale.getLanguage();
            String country = locale.getCountry();
            String localeString = null;
            if (language != null) {
                if (country != null) {
                    // 为扫描器设置当前系统使用的国家和语言。
                    scanner.setLocale(language + "_" + country);
                } else {
                    scanner.setLocale(language);
                }
            }
        }
        return scanner;
}
```

MSS 模块的扫描工作就到此为止了，下面轮到主角 MediaScanner 登场了。在介绍主角之前，不妨先总结一下本节的内容。

10.2.3　android.process.media 媒体扫描工作的流程总结

媒体扫描工作流程涉及 MSR 和 MSS 的交互，来总结一下相关的流程：

- MSR 接收外部发来的扫描请求，并通过 startService 方式启动 MSS 处理。
- MSS 的主线程接收 MSR 所收到的请求，然后投递给工作线程去处理。
- 工作线程做一些前期处理工作后（例如向系统广播扫描开始的消息），就创建媒体扫描器 MediaScanner 来处理扫描目标。
- MS 扫描完成后，工作线程再做一些后期处理，然后向系统发送扫描完毕的广播。

10.3 MediaScanner 分析

现在分析媒体扫描器 MediaScanner 的工作原理，它将纵跨 Java 层、JNI 层，以及 Native 层。先看它在 Java 层中的内容。

10.3.1 Java 层分析

1. 创建 MediaScanner

认识一下 MediaScanner，它的代码如下所示：

[-->MediaScanner.java]

```
public class MediaScanner
{
static {
        /*
         加载 libmedia_jni.so，这么重要的库竟然放在如此不起眼的 MediaScanner 类中加载。
         个人觉得，可能是因为开机后多媒体系统中最先启动的就是媒体扫描工作吧。
        */
        System.loadLibrary("media_jni");
         native_init();
}
// 创建媒体扫描器。
public MediaScanner(Context c) {
         native_setup();// 调用 JNI 层的函数做一些初始化工作。
         ......
}
```

在上面的 MS 中，比较重要的几个调用函数是：

native_init 和 native_setup，关于它们的故事，在分析 JNI 层时再做介绍。

MS 创建好后，MSS 将调用它的 scanDirectories 开展扫描工作，下面来看这个函数。

2. scanDirectories 分析

scanDirectories 的代码如下所示：

[-->MediaScanner.java]

```
public void scanDirectories(String[] directories, String volumeName) {
```

```
try {
        long start = System.currentTimeMillis();
        initialize(volumeName);// ①初始化
        prescan(null);// ②扫描前的预处理。
        long prescan = System.currentTimeMillis();

        for (int i = 0; i < directories.length; i++) {
          /*
            ③ processDirectory是一个native函数，调用它来对目标文件夹进行扫描，
            其中MediaFile.sFileExtensions是一个字符串，包含了当前多媒体系统所支持的
            媒体文件的后缀名，例如 .MP3、.MP4 等。mClient 为 MyMediaScannerClient 类型，
            它是从MediaScannerClient 类派生的。它的作用我们后面再做分析。

           */
           processDirectory(directories[i], MediaFile.sFileExtensions,
                            mClient);
          }
           long scan = System.currentTimeMillis();
           postscan(directories);// ④扫描后处理。
           long end = System.currentTimeMillis();
           ......// 统计扫描时间等。
}
```

上面一共列出了四个关键点（即①～④），下面逐一对其分析。

（1）initialize 分析

initialize 主要是初始化一些 Uri，因为扫描时需把文件的信息插入媒体数据库中，而媒体数据库针对 Video、Audio、Image 文件等都有对应的表，这些表的地址则由 Uri 表示。下面是 initialize 的代码：

[-->MediaScanner.java]

```
private void initialize(String volumeName) {
// 得到 IMediaProvider 对象，通过这个对象可以对媒体数据库进行操作。
  mMediaProvider =
              mContext.getContentResolver().acquireProvider("media");
      // 初始化 Uri，下面分别介绍一下。
      // 音频表的地址，也就是数据库中的 audio_meta 表。
      mAudioUri = Audio.Media.getContentUri(volumeName);
      // 视频表地址，也就是数据库中的 video 表。
      mVideoUri = Video.Media.getContentUri(volumeName);
      // 图片表地址，也就是数据库中的 images 表。
      mImagesUri = Images.Media.getContentUri(volumeName);
      // 缩略图表地址，也就是数据库中的 thumbs 表。
      mThumbsUri = Images.Thumbnails.getContentUri(volumeName);
      // 如果扫描的是外部存储，则支持播放列表、音乐的流派等内容。
       if (!volumeName.equals("internal")) {
            mProcessPlaylists = true;
            mProcessGenres = true;
            mGenreCache = new HashMap<String, Uri>();
```

```
            mGenresUri = Genres.getContentUri(volumeName);
            mPlaylistsUri = Playlists.getContentUri(volumeName);
            if ( Process.supportsProcesses()) {
                //SD 卡存储区域一般使用 FAT 文件系统，所以文件名与大小写无关。
                mCaseInsensitivePaths = true;
            }
        }
    }
```

下面看第二个关键函数 prescan。

（2）prescan 分析

在媒体扫描过程中，有个令人头疼的问题，来举个例子，这个例子会贯穿在对这个问题的整体分析过程中。例子：假设某次扫描之前 SD 卡中有 100 个媒体文件，数据库中有 100 条关于这些文件的记录，现因某种原因删除了其中的 50 个媒体文件，那么媒体数据库什么时候会被更新呢？

说明 读者别小瞧这个问题。现在有很多文件管理器支持删除文件和文件夹，它们用起来很方便，却没有对应地更新数据库，这导致了查询数据库时还能得到这些媒体文件的信息，但这个文件实际上已不存在了，而且后面所有和此文件有关的操作都会因此而失败。

其实，MS 已经考虑到这一点了，prescan 函数的主要作用是在扫描之前把数据库中和文件相关的信息取出并保存起来，这些信息主要是媒体文件的路径，所属表的 Uri。就上面这个例子来说，它会从数据库中取出这 100 个文件的文件信息。

prescan 的代码如下所示：

[-->MediaScanner.java]

```
private void prescan(String filePath) throws RemoteException {
        Cursor c = null;
        String where = null;
        String[] selectionArgs = null;
        //mFileCache 保存从数据库中获取的文件信息。
        if (mFileCache == null) {
            mFileCache = new HashMap<String, FileCacheEntry>();
        } else {
            mFileCache.clear();
        }
        ......
       try {
            // 从 Audio 表中查询其中和音频文件相关的文件信息。
            if (filePath != null) {
                where = MediaStore.Audio.Media.DATA + "=?";
                selectionArgs = new String[] { filePath };
            }
            // 查询数据库的 Audio 表，获取对应的音频文件信息。
```

```
            c = mMediaProvider.query(mAudioUri, AUDIO_PROJECTION, where,
                                     selectionArgs, null);
            if (c != null) {
                try {
                    while (c.moveToNext()) {
                        long rowId = c.getLong(ID_AUDIO_COLUMN_INDEX);
                        // 音频文件的路径。
                        String path = c.getString(PATH_AUDIO_COLUMN_INDEX);
                        long lastModified =
                          c.getLong(DATE_MODIFIED_AUDIO_COLUMN_INDEX);

                         if (path.startsWith("/")) {
                            String key = path;
                            if (mCaseInsensitivePaths) {
                                key = path.toLowerCase();
                            }
                           // 把文件信息存到 mFileCache 中。
                            mFileCache.put(key,
                                 new FileCacheEntry(mAudioUri, rowId, path,
                                  lastModified));
                        }
                    }
                } finally {
                    c.close();
                    c = null;
                }
            }
          ......// 查询其他表，取出数据中关于视频、图像等文件的信息并存入到 mFileCache 中。
        finally {
            if (c != null) {
                c.close();
            }
        }
    }
```

读懂了前面的例子，在阅读 prescan 函数时可能就比较轻松了。prescan 函数执行完后，mFileCache 保存了扫描前所有媒体文件的信息，这些信息是从数据库中查询得来的，也就是旧有的信息。

接下来看最后两个关键函数。

（3）processDirectory 和 postscan 分析

processDirectory 是一个 native 函数，其具体功能放到 JNI 层再分析，这里先简单介绍一下，它在解决上一节那个例子中的问题时所做的工作。答案是：

processDirectory 将扫描 SD 卡，每扫描一个文件，都会设置 mFileCache 中对应文件的一个叫 mSeenInFileSystem 的变量为 true。这个值表示这个文件目前还存在于 SD 卡上。这样，待整个 SD 卡扫描完后，mFileCache 的那 100 个文件中就会有 50 个文件的 mSeenInFileSystem 为 true，而剩下的另 50 个文件其初始值则为 false。

看到上面的内容，可以知道 postscan 的作用了吧？就是它把不存在于 SD 卡的文件信息从数据库中删除，而使数据库得以彻底地更新。来看 postscan 函数是否是这样处理的：

[-->MediaScanner.java]

```
private void postscan(String[] directories) throws RemoteException {

  Iterator<FileCacheEntry> iterator = mFileCache.values().iterator();
  while (iterator.hasNext()) {
            FileCacheEntry entry = iterator.next();
            String path = entry.mPath;

            boolean fileMissing = false;
            if (!entry.mSeenInFileSystem) {
                if (inScanDirectory(path, directories)) {
                    fileMissing = true; //这个文件确实丢失了。
                } else {
                    File testFile = new File(path);
                    if (!testFile.exists()) {
                        fileMissing = true;
                    }
                }
            }
        // 如果文件确实丢失，则需要把数据库中和它相关的信息删除。
        if (fileMissing) {
           MediaFile.MediaFileType mediaFileType = MediaFile.getFileType(path);
           int fileType = (mediaFileType == null ? 0 : mediaFileType.fileType);
          if (MediaFile.isPlayListFileType(fileType)) {
                    ......// 处理丢失文件是播放列表的情况。
            } else {
               /*
                 由于文件信息中还携带了它在数据库中的相关信息，所以从数据库中删除对应的信息会
                 非常快。
               */
               mMediaProvider.delete(ContentUris.withAppendedId(
                         entry.mTableUri, entry.mRowId), null, null);
             iterator.remove();
             }
          }
    }
    ..... // 删除缩略图文件等工作。
}
```

Java 层中的四个关键点，至此已介绍了三个，另外一个 processDirectory 是媒体扫描的关键函数，由于它是一个 native 函数，所以下面将转战到 JNI 层来进行分析。

10.3.2 JNI 层分析

现在分析 MS 的 JNI 层。在 Java 层中，有三个函数涉及 JNI 层，它们是：

- native_init，这个函数由 MediaScanner 类的 static 块调用。
- native_setup，这个函数由 MediaScanner 的构造函数调用。
- processDirectory，这个函数由 MS 扫描文件夹时调用。

分别来分析它们。

1.ative_init 函数分析

下面是 native_init 对应的 JNI 函数，其代码如下所示：

[-->android_media_MediaScanner.cpp]

```
static void
android_media_MediaScanner_native_init(JNIEnv *env)
{
     jclass clazz;
     clazz = env->FindClass("android/media/MediaScanner");
     //取得 Java 中 MS 类的 mNativeContext 信息。待会创建的 Native 对象的指针会保存
     //到 Java MS 对象的 mNativeContext 变量中。
     fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
     ......
}
```

native_init 函数没什么新意，这种把 Native 对象的指针保存到 Java 对象中的做法，已经屡见不鲜了。下面看第二个函数 native_setup。

2. native_setup 函数分析

native_setup 对应的 JNI 函数如下所示：

[-->android_media_MediaScanner.cpp]

```
android_media_MediaScanner_native_setup(JNIEnv *env, jobject thiz)
{
    // 创建 Native 层的 MediaScanner 对象。
    MediaScanner *mp = createMediaScanner();
    ......
    // 把 mp 的指针保存到 Java MS 对象的 mNativeContext 中去。
    env->SetIntField(thiz, fields.context, (int)mp);
}
// 下面的 createMediaScanner 这个函数将创建一个 Native 的 MS 对象。
static MediaScanner *createMediaScanner() {
#if BUILD_WITH_FULL_STAGEFRIGHT
    char value[PROPERTY_VALUE_MAX];
    if (property_get("media.stagefright.enable-scan", value, NULL)
        && (!strcmp(value, "1") || !strcasecmp(value, "true"))) {
        return new StagefrightMediaScanner; // 使用 Stagefright 的 MS。
    }
#endif
#ifndef NO_OPENCORE
    return new PVMediaScanner(); // 使用 Opencore 的 MS，我们会分析这个。
```

```
#endif
    return NULL;
}
```

native_setup 函数将创建一个 Native 层的 MS 对象，不过可惜的是，它使用的还是 Opencore 提供的 PVMediaScanner，所以后面不可避免地还会和 Opencore “正面交锋”。

3. processDirectory 函数分析

来看 processDirectory 函数，它对应的 JNI 函数代码如下所示：

[-->android_media_MediaScanner.cpp]

```
android_media_MediaScanner_processDirectory(JNIEnv *env, jobject thiz,
                            jstring path, jstring extensions, jobject client)
{
    /*
     注意上面传入的参数，path 为目标文件夹的路径，extensions 为 MS 支持的媒体文件后缀名集合，
     client 为 Java 中的 MediaScannerClient 对象。
     */

     MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);

     const char *pathStr = env->GetStringUTFChars(path, NULL);
     const char *extensionsStr = env->GetStringUTFChars(extensions, NULL);
     ......

    // 构造一个 Native 层的 MyMediaScannerClient，并使用 Java 那个 Client 对象做参数。
    // 这个 Native 层的 Client 简称为 MyMSC。
     MyMediaScannerClient myClient(env, client);
     // 调用 Native 的 MS 扫描文件夹，并且把 Native 的 MyMSC 传进去。
     mp->processDirectory(pathStr, extensionsStr, myClient,
                            ExceptionCheck, env);
     ......
     env->ReleaseStringUTFChars(path, pathStr);
     env->ReleaseStringUTFChars(extensions, extensionsStr);
     ......
}
```

processDirectory 函数本身倒不难，但又冒出了几个我们之前没有接触过的类型，下面先来认识一下它们。

4. 到底有多少种对象？

图 10-1 展示了 MediaScanner 所涉及的相关类和它们之间的关系：

为了便于理解，所以将 Java 和 Native 层的对象都画于图中了。从上图可知：

- Java MS 对象通过 mNativeContext 指向 Native 的 MS 对象。
- Native 的 MyMSC 对象通过 mClient 保存 Java 层的 MyMSC 对象。

- Native 的 MS 对象调用 processDirectory 函数的时候会使用 Native 的 MyMSC 对象。
- 另外，图中 Native MS 类的 processFile 是一个虚函数，需要派生类来实现。

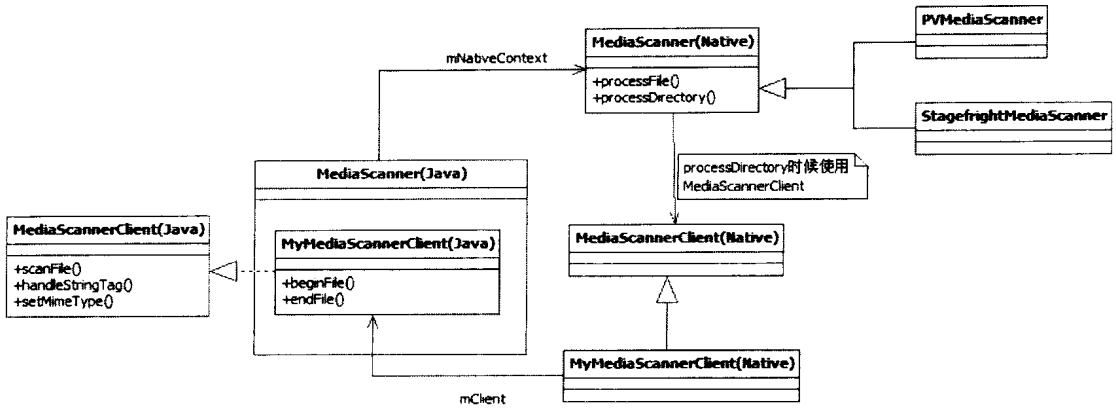


图 10-1　MS 相关类示意图

其中比较费解的是 MyMSC 对象。它们有什么用呢？这个问题真是一言难尽。下面通过 processDirectory 来探寻其中的原因，这回得进入 PVMediaScanner 的领地了。

10.3.3　PVMediaScanner 分析

1.PVMS 的 processDirectory 分析

来看 PVMediaScanner（以后简称为 PVMS，它就是 Native 层的 MS）的 processDirectory 函数。这个函数是由它的基类 MS 实现的。注意，源码中有两个 MediaScanner.cpp，它们的位置分别是：

- framework/base/media/libmedia
- external/opencore/android/

接着看 libmedia 下的那个 MediaScanner.cpp，其中 processDirectory 函数的代码如下所示：

[-->MediaScanner.cpp]

```
status_t MediaScanner::processDirectory(const char *path,
                        const char *extensions, MediaScannerClient &client,
                        ExceptionCheck exceptionCheck, void *exceptionEnv) {

    ......// 做一些准备工作。
    client.setLocale(locale()); // 给 Native 的 MyMSC 设置 locale 信息。
   // 调用 doProcessDirectory 函数扫描文件夹。
    status_t result =  doProcessDirectory(pathBuffer, pathRemaining,
                       extensions, client,exceptionCheck, exceptionEnv);

    free(pathBuffer);
```

```
    return result;
}
// 下面直接看这个 doProcessDirectory 函数。
status_t MediaScanner::doProcessDirectory(char *path, int pathRemaining,
                  const char *extensions,MediaScannerClient &client,
                  ExceptionCheck exceptionCheck,void *exceptionEnv) {

   ..... // 忽略 .nomedia 文件夹。

    DIR* dir = opendir(path);
    ......

    while ((entry = readdir(dir))) {
        // 枚举目录中的文件和子文件夹信息。
        const char* name = entry->d_name;
        ......
        int type = entry->d_type;
         ......
        if (type == DT_REG || type == DT_DIR) {
            int nameLength = strlen(name);
            bool isDirectory = (type == DT_DIR);
           ......
            strcpy(fileSpot, name);
            if (isDirectory) {
                ......
                 // 如果是子文件夹，则递归调用 doProcessDirectory。
                int err = doProcessDirectory(path, pathRemaining - nameLength - 1,
                            extensions, client, exceptionCheck, exceptionEnv);
                ......
            } else if (fileMatchesExtension(path, extensions)) {
                // 如果该文件是 MS 支持的类型（根据文件的后缀名来判断）。
                struct stat statbuf;
                stat(path, &statbuf); // 取出文件的修改时间和文件的大小。
                if (statbuf.st_size > 0) {
                    // 如果该文件大小非零，则调用 MyMSC 的 scanFile 函数！！？
                    client.scanFile(path, statbuf.st_mtime, statbuf.st_size);
                }
                if (exceptionCheck && exceptionCheck(exceptionEnv)) goto failure;
            }
        }
    }
.....
}
```

假设正在扫描的媒体文件类型是属于 MS 支持的，那么，上面代码中最不可思议的是，它竟然调用了 MSC 的 scanFile 来处理这个文件，也就是说，MediaScanner 调用 MediaScannerClient 的 scanFile 函数。这是为什么呢？还是来看看这个 MSC 的 scanFile 吧。

2.MyMSC 的 scanFile 分析

（1）JNI 层的 scanFile

其实，在调用 processDirectory 时，所传入的 MSC 对象的真实类型是 MyMediaScannerClient，下面来看它的 scanFile 函数，代码如下所示：

[-->android_media_MediaScanner.cpp]

```
virtual bool scanFile(const char* path, long long lastModified,
                      long long fileSize)
    {
        jstring pathStr;
        if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;
        //mClient 是 Java 层的那个 MyMSC 对象，这里调用它的 scanFile 函数。
        mEnv->CallVoidMethod(mClient, mScanFileMethodID, pathStr,
                             lastModified, fileSize);

        mEnv->DeleteLocalRef(pathStr);
        return (!mEnv->ExceptionCheck());
}
```

太没有“天理”了！ Native 的 MyMSC scanFile 主要的工作就是调用 Java 层 MyMSC 的 scanFile 函数。这又是为什么呢？

（2）Java 层的 scanFile

现在只能来看 Java 层的这个 MyMSC 对象了，它的 scanFile 代码如下所示：

[-->MediaScanner.java]

```
public void scanFile(String path, long lastModified, long fileSize) {
            ......
            // 调用 doScanFile 函数。
            doScanFile(path, null, lastModified, fileSize, false);
        }
// 直接来看 doScanFile 函数。
 public Uri doScanFile(String path, String mimeType, long lastModified,
                       long fileSize, boolean scanAlways) {
/*
上面参数中的 scanAlways 用于控制是否强制扫描，有时候一些文件在前后两次扫描过程中没有
发生变化，这时候 MS 可以不处理这些文件。如果 scanAlways 为 true，则这些没有变化
的文件也要扫描。
*/
Uri result = null;
long t1 = System.currentTimeMillis();
try {
     /*
      beginFile 的主要工作，就是将保存在 mFileCache 中的对应文件信息的
      mSeenInFileSystem 设为 true。如果这个文件之前没有在 mFileCache 中保存，
      则会创建一个新项添加到 mFileCache 中。另外它还会根据传入的 lastModified 值
```

```
        做一些处理，以判断这个文件是否在前后两次扫描的这个时间段内被修改，如果有修改，则
        需要重新扫描。
      */
          FileCacheEntry entry = beginFile(path, mimeType,
                                  lastModified, fileSize);
          if (entry != null && (entry.mLastModifiedChanged || scanAlways)) {
               String lowpath = path.toLowerCase();
               ......

               if (!MediaFile.isImageFileType(mFileType)) {
                 // 如果不是图片，则调用 processFile 进行扫描，而图片不需要扫描就可以处理。
                 // 注意在调用 processFile 时把这个 Java 的 MyMSC 对象又传了进去。
                 processFile(path, mimeType, this);
               }
             // 扫描完后，需要把新的信息插入数据库，或者要将原有的信息更新，而 endFile 就是做这项工作的。
              result = endFile(entry, ringtones, notifications,
                         alarms, music, podcasts);
                 }
             } ......
             return result;
        }
```

下面看这个 processFile，这又是一个 native 的函数。

说明 上面代码中的 beginFile 和 endFile 函数比较简单，读者可以自行研究。

（3）JNI 层的 processFile 分析

MediaScanner 的代码有点绕，是不是？总感觉我们像追兵一样，追着 MS 在赤水来回地绕，现在应该是二渡赤水了。来看这个 processFile 函数，代码如下所示：

[-->android_media_MediaScanner.cpp]

```
android_media_MediaScanner_processFile(JNIEnv *env, jobject thiz,
                        jstring path, jstring mimeType, jobject client)
{
    //Native 的 MS 还是那个 MS，其真实类型是 PVMS。
    MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);
    // 又构造了一个新的 Native 的 MyMSC，不过它指向的 Java 层的 MyMSC 没有变化。
    MyMediaScannerClient myClient(env, client);
    // 调用 PVMS 的 processFile 处理这个文件。
    mp->processFile(pathStr, mimeTypeStr, myClient);
}
```

看来，现在得去看看 PVMS 的 processFile 函数了。

3. PVMS 的 processFile 分析

（1）扫描文件

这是我们第一次进入到 PVMS 的代码中进行分析：

[-->PVMediaScanner.cpp]

```
status_t PVMediaScanner::processFile(const char *path, const char* mimeType,
                                     MediaScannerClient& client)
{
    status_t result;
    InitializeForThread();

    // 调用 Native MyMSC 对象的函数做一些处理。
    client.setLocale(locale());
    /*
    beginFile 由基类 MSC 实现，这个函数将构造两个字符串数组，一个叫 mNames，另一个叫 mValues。
    这两个变量的作用和字符编码有关，后面会碰到。
   */
    client.beginFile();
    ......
    const char* extension = strrchr(path, '.');
    // 根据文件后缀名来做不同的扫描处理。
    if (extension && strcasecmp(extension, ".mp3") == 0) {
        result = parseMP3(path, client);//client 又传进去了，我们看看对 MP3 文件的处理
     ......
}
  /*
   endFile 会根据 client 设置的区域信息来对 mValues 中的字符串做语言转换，例如一首 MP3
   中的媒体信息是韩文，而手机设置的语言为简体中文，endFile 会尽量对这些韩文进行转换。
   不过语言转换向来是个大难题，不能保证所有语言的文字都能相互转换。转换后的每一个 value 都
   会调用 handleStringTag 做后续处理。
*/
  client.endFile();
  ......
}
```

下面再到 parseMP3 这个函数中去看看，它的代码如下所示：

[-->PVMediaScanner.cpp]

```
static PVMFStatus parseMP3(const char *filename, MediaScannerClient& client)
{
  // 对 MP3 文件进行解析，得到诸如 duration、流派、标题的 TAG（标签）信息。在 Windows 平台上
// 可通过千千静听软件查看 MP3 文件的所有 TAG 信息。
   ......
    //MP3 文件已经扫描完了，下面将这些 TAG 信息添加到 MyMSC 中，一起来看看。
   if (!client.addStringTag("duration", buffer))
        ......
}
```

（2）添加 TAG 信息

文件扫描完了，现在需要把文件中的信息通过 addStringTag 函数告诉给 MyMSC。下面来看 addStringTag 的工作。这个函数由 MyMSC 的基类 MSC 处理。

[-->MediaScannerClient.cpp]

```
bool MediaScannerClient::addStringTag(const char* name, const char* value)
{
    if (mLocaleEncoding != kEncodingNone) {
        bool nonAscii = false;
        const char* chp = value;
        char ch;
        while ((ch = *chp++)) {
            if (ch & 0x80) {
                nonAscii = true;
                break;
            }
        }
      /*
        判断 name 和 value 的编码是不是 ASCII，如果不是的话则保存到
        mNames 和 mValues 中，等到 endFile 函数的时候再集中做字符集转换。
      */
        if (nonAscii) {
            mNames->push_back(name);
            mValues->push_back(value);
            return true;
        }
}
// 如果字符编码是 ASCII 的话，则调用 handleStringTag 函数，这个函数由子类 MyMSC 实现。
    return handleStringTag(name, value);
}
```

[-->android_media_MediaScanner.cpp::MyMediaScannerClient 类]

```
virtual bool handleStringTag(const char* name, const char* value)
{
    ......
    // 调用 Java 层 MyMSC 对象的 handleStringTag 进行处理。
    mEnv->CallVoidMethod(mClient, mHandleStringTagMethodID, nameStr, valueStr);
}
```

[-->MediaScanner.java]

```
  public void handleStringTag(String name, String value) {
            // 保存这些 TAG 信息到 MyMSC 对应的成员变量中去。
            if (name.equalsIgnoreCase("title") || name.startsWith("title;")) {
                mTitle = value;
            } else if (name.equalsIgnoreCase("artist") ||
                        name.startsWith("artist;")) {
                mArtist = value.trim();
            } else if (name.equalsIgnoreCase("albumartist") ||
                        name.startsWith("albumartist;")) {
```

```
                mAlbumArtist = value.trim();
            }
            ......
    }
```

到这里，一个文件的扫描就算做完了。不过，读者还记得是什么时候把这些信息保存到数据库的吗？

是在 Java 层 MyMSC 对象的 endFile 中，这时它会把文件信息组织起来，然后存入媒体数据库。

10.3.4　关于 MediaScanner 的总结

下面总结一下媒体扫描的工作流程，它并不复杂，就是有些绕，如图 10-2 所示。

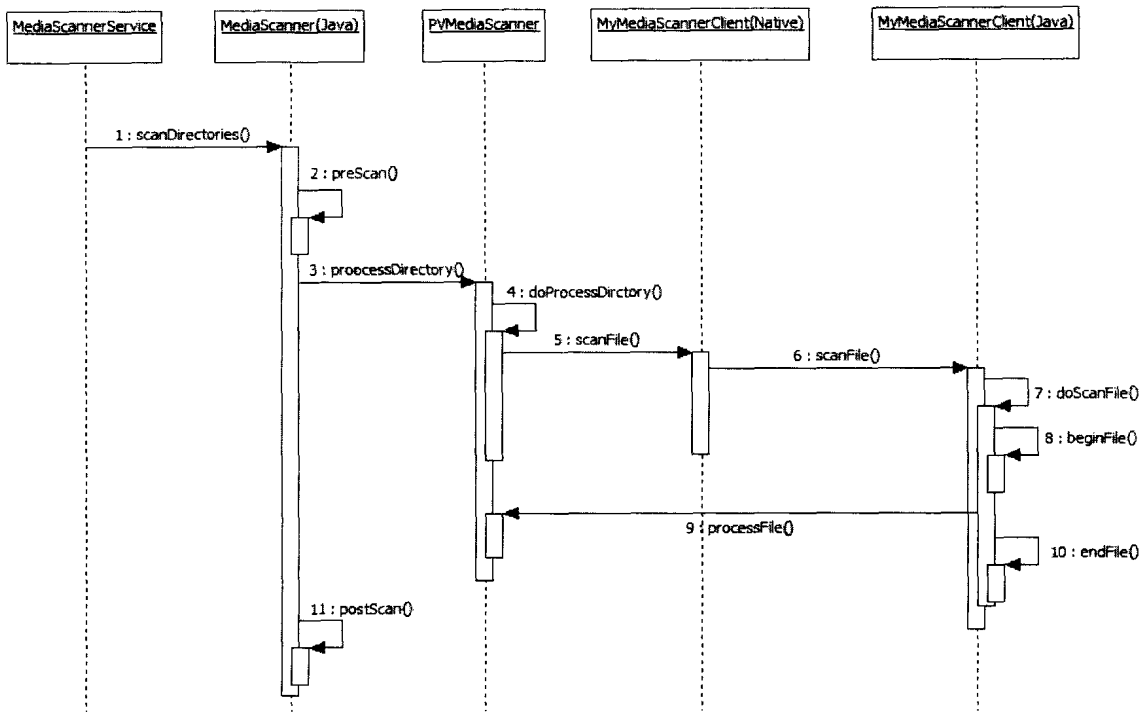


图 10-2　MediaScanner 扫描流程图

通过上图可以发现，MS 扫描的流程还是比较清晰的，就是四渡赤水这一招，让很多初学者摸不着头脑。不过读者千万不要像我当初那样，觉得这是垃圾代码的代表。实际上这是码农有意而为之，在 MediaScanner.java 中通过一段比较详细的注释，对整个流程做了文字总结，这段总结非常简单，这里就不翻译了。

[-->MediaScanner.java]

```
// 前面还有一段话，读者可自行阅读。下面是流程的文件总结。
* In summary:
 * Java MediaScannerService calls
 * Java MediaScanner scanDirectories, which calls
 * Java MediaScanner processDirectory (native method), which calls
 * native MediaScanner processDirectory, which calls
 * native MyMediaScannerClient scanFile, which calls
 * Java MyMediaScannerClient scanFile, which calls
 * Java MediaScannerClient doScanFile, which calls
 * Java MediaScanner processFile (native method), which calls
 * native MediaScanner processFile, which calls
 * native parseMP3, parseMP4, parseMidi, parseOgg or parseWMA, which calls
 * native MyMediaScanner handleStringTag, which calls
 * Java MyMediaScanner handleStringTag.
 * Once MediaScanner processFile returns, an entry is inserted in to the database.
```

看完这么详细的注释，想必你也会认为码农真是故意这么做的。但他们为什么要设计成这样呢？以后会不会改呢？注释中也说明了目前设计的流程是这样，估计以后有可能改。

10.4 拓展思考

10.4.1 MediaScannerConnection 介绍

通过前面的介绍，我们知道 MSS 支持以广播方式发送扫描请求。除了这种方式外，多媒体系统还提供了一个 MediaScannerConnection 类，通过这个类可以直接跨进程调用 MSS 的 scanFile，并且 MSS 扫描完一个文件后会通过回调来通知扫描完毕。MediaScannerConnection 类的使用场景包括浏览器下载了一个媒体文件，彩信接收到一个媒体文件等，这时都可以用它来执行媒体文件的扫描工作。

下面来看这个类输出的几个重要 API，由于它非常简单，所以这里就不再进行流程的分析了。

[-->MediaScannerConnection.java]

```
public class MediaScannerConnection implements ServiceConnection {

 // 定义 OnScanCompletedListener 接口，如果媒体文件扫描完，MSS 就调用这个接口进行通知。
 public interface OnScanCompletedListener {
        public void onScanCompleted(String path, Uri uri);
    }
    // 定义 MediaScannerConnectionClient 接口，派生自 OnScanCompletedListener，
    // 它增加了 MediaScannerConnection connect 上 MSS 的通知。
    public interface MediaScannerConnectionClient extends
                                        OnScanCompletedListener {
```

```
        public void onMediaScannerConnected();// 连接 MSS 的回调通知。
        public void onScanCompleted(String path, Uri uri);
    }
    // 构造函数。
    public MediaScannerConnection(Context context,
                                  MediaScannerConnectionClient client);
    // 封装了和 MSS 连接及断开连接的操作。
    public void connect();
    public void disconnect()
    // 扫描单个文件。
    public void scanFile(String path, String mimeType);
    // 我更喜欢下面这个静态函数，它支持多个文件的扫描，实际上间接提供了文件夹的扫描功能。
    public static void scanFile(Context context, String[] paths,
                    String[] mimeTypes,OnScanCompletedListener callback);

    ......
}
```

从使用者的角度来看，本人更喜欢静态的 scanFile 函数，一方面它封装了和 MSS 连接等相关的工作，另一方面它还支持多个文件的扫描，所以如果没什么特殊要求，建议读者还是使用这个静态函数。

10.4.2　我问你答

本节是本书的最后一小节，相信一路走来读者对 Android 的认识和理解已有提高。下面将提几个和媒体扫描相关的问题请读者思考，或者说是提供给读者自行钻研。在解答或研究过程中，读者如有什么心得，不妨也记录并与我们共享。那些对 Android 有深刻见地的读者，说不定会收到我们公司 HR 的电话哦！

下面是我在研究 MS 过程中，觉得读者可以进行拓展研究的内容：

- 本书还没有介绍 android.process.media 中的 MediaProvider 模块，读者不妨分别把扫描一个图片、MP3 歌曲、视频文件的流程走一遍，不过这个流程分析的重点是 MediaProvider。
- MP 中最复杂的是缩略图的生成，读者在完成上一步的基础上，可集中精力解决缩略图生成的流程。对于视频文件缩略图的生成还会涉及 MediaPlayerService。
- 到这一步，相信读者对 MP 已有了较全面的认识。作为深入学习的跳板，我建议有兴趣的读者可以对 Android 平台上和数据库有关的模块，以及 ContentProvider 进行深入研究。这里还会涉及很多问题，例如 query 返回的 Cursor 是怎么把数据从 MediaProvider 进程传递到客户端进程的？为什么一个 ContentProvider 死掉后，它的客户端也会跟着被 kill 掉？

10.5 本章小结

本章是全书最后一章，也是最轻松的一章。这一章重点介绍了多媒体系统中和媒体文件扫描相关的知识，相信读者对媒体扫描流程中的“四渡赤水”会有较深印象。

本章的拓展内容中介绍了 API 类 MediaScannerConnection 的使用方法，另外，提出了几个和媒体扫描相关的问题请读者与我们共同思考。

[General Information]
书名=深入理解Android 卷I
SS号=12842633
DX号=000008163802
fenlei=1816111203
作者=邓凡平著
出版社=北京市：机械工业出版社
出版日期=2011.09
ISBN号=978-7-111-35762-9
页数=488
尺寸=26cm
丛书名=
原书定价=69.00
主题词=
中图法分类号=TN929.53 （工业技术>无线电电子学、电信技术>无线通信>移动通信>蜂窝式移动通信系统（大哥大、移动电话手机））
参考文献格式=邓凡平著.
深入理解Android 卷I.
北京市：机械工业出版社,
2011.09.
内容提要=本书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。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。
pdgpath=http://sxt.5read.com/300-06/diskqjs/qjs1783/02/