深入解析 Android 5.0系统

刘超 著

基于新版 Android 5 系统

- 全面细致讲解系统调用、内存管理、管道、线程管理、Log 模块、Binder 驱动、同步和消息机制、init 进程、Zygote 进程、资源管理、应用管理、软件管理、管理进程、图形显示系统、窗口系统、输入管理系统、电源管理、存储系统、网络系统、音频系统、SELinux 模块、最新 ART 虚拟机、垃圾回收、Recovery 模块、内存监控等核心模块。
- 书中从可视化角度给出主要模块的架构、原型和主干实现，很多模块可以直接拿来当作代码以帮助读者融会贯通。本书可以帮助读者快速理解内核的设计思想，进而获得对 Android 系统进行二次开发的能力。

人民邮电出版社
POSTS & TELECOM PRESS

深入解析 Android 5.0 系统

刘　超　著

人 民 邮 电 出 版 社

北　京

内 容 提 要

本书详细剖析了最新 Android 5.0 系统框架的原理和具体实现。本书共 24 章，覆盖了 Android 5.0 系统中重要的模块，对于每个模块都详细介绍了它们的架构、原理及代码实现等各个方面，尽量让读者知其然，又知其所以然，达到学以致用的目的。

本书主要内容为 Android Build 系统核心、Android 的 Bionic、系统调用的实现、Binder 应用层的核心类、JNI、同步和消息机制、进程间的消息传递、Init 进程、Zygote 进程、资源管理、SystemServer 进程、应用管理、组件管理、多用户模式、图形显示系统、窗口系统、输入管理、电源管理、存储系统、网络管理框架、音频系统、SELinux 模块、Dalvik 和 ART 虚拟机、Recovery 模块、调试方法、内存泄露分析、Android 的自动化测试等系统的核心知识。

本书中尽可能详细地给出了代码的注释、各种属性和常量的解释，以及系统中使用的各种文件格式的介绍，希望读者能通过本书，获得对 Android 5.0 系统进行二次开发的能力，本书是系统开发人员的案头必备书。

本书面向的读者主要是进行系统开发的工程师，包括应用开发工程师、ROM 开发工程师和各种使用 Android 作为开发平台的 TV 和可穿戴式设备（Wear）的开发工程师。本书也可以作为大专院校相关专业师生的学习用书及培训学校的教材。

推 荐 序

这是一本有 6 年安卓系统开发经验的、中国顶级 Android 系统工程师的心血之作！

这是一本可以推荐给任何从事 Android 系统开发或应用开发工程师看的书！

——原 Motorola 软件总监，播思通讯 CTO，饶宏

这本书介绍 Android 系统的翔实和认真程度可能在市面上无出其右。从 Android 下载安装到配置编译，从 JNI/Boinic 到 Loop/Init，从 SystemServer 到 Provider，从包管理到图形系统，从窗口系统到输入管理，从电源管理到睡眠唤醒机制，从网络管理到音、视频系统，甚至从 Vold 到 Recovery，从虚拟机到自动化测试，本书都有详细解释和说明。翻开这本书，就能感觉到作者是在认认真真地撰写这本书，生怕错了一个标点；全面细致的讲解，又生怕有哪一句话读者不明白。

作为一个工作十多年的资深工程师，作为一个从 Android 1.0 版本开始接触 Android 系统的工程师，作为一个量产过多款产品的 Android 一线架构师，我想没有这样的经历是很难将这本书写得如此详尽。

感谢刘超给我们的 Android 系统工程师写了这么优秀一本书，一本 Andoird 工程笔记，一本建筑 Android“大厦”的详细图纸。非常感谢作者的辛勤付出，希望读者可以从中得到有益的启发，开启自己完美的 Android 开发之旅！

——小米电视系统软件部总监，茹忆

一本非常优秀的介绍 Android 内部机制的图书，详细地分析了 Android 系统的大部分模块，值得每一个希望深入学习 Android 系统的工程师拥有。

——德信无线软件部经理，陈行星

前　言

Android 5.0（代号 Lollipop，简称 Android L）是 Android 系统自 1.0 版本以来变化最大的一个升级版，它包含了很多激动人心的改进，其中最明显的变化在于用户体验方面。“Material Design”扁平、简洁，且其色彩丰富、动感十足的设计风格，给用户带来了更新鲜的感觉。除了界面风格的重大变化外，包括搜索、应用菜单、通知中心等众多细节在 Android 5.0 上也有很大改进。

Android 5.0 在系统性能提升方面也做了相当大的努力，首先是 ART 彻底代替了 Dalivk 虚拟机，同时开始支持 64 位编译，因而，用户很快就能使用运行 64 位 CPU 的手机，重要的是，系统的耗电量也通过“Porject Volte”项目有了显著的改善。

从 2008 年 9 月 23 日 Google 发布 Android 1.0 版本到 2014 年 6 月 26 日 Google 正式推出 Android 5.0，已经 6 年过去了。笔者第一次接触 Android 系统还是 2008 年在 Motorola 工作的时候，那时 Android 才崭露头角，而现在 Android 系统已经占据了手机市场的一半多份额。由于工作关系，笔者 6 年来一直致力于 Android Framework 的开发和研究，也算是略有收获。Android 是一套开源系统，官方（Google）提供的技术资料很有限，其官方网站（www.android.com）中提供的资料更多地是在介绍 Android 应用程序的开发。这 6 年间，国内也涌现了大批介绍 Android 的书籍，主要是介绍 Android 应用程序的开发，分析 Android 内核源码的书籍较少，为此特意编写了本书。

本书内容介绍

本书介绍了 Android 主要模块的架构、原理和主干实现。希望读者能通过本书，获得对 Android 系统进行二次开发的能力，而笔者在书中，也尽可能详细地给出代码的注释、各种属性和常量的解释，以及系统中使用的各种文件格式的介绍，希望本书能成为系统开发人员的案头工具书。

本书共分 24 章，基本覆盖了 Android 系统中重要的模块，对于每个模块，笔者尽量详细地介绍了它们的架构、原理和代码等各个方面，但是，由于篇幅限制以及本人的理解有限，如果读者发现了其中的谬误，欢迎来信赐教，本人不胜感激。关于本书的一些后续的信息，可以访问笔者的博客（http://blog.csdn.net/u013234805），答疑 QQ 群为 216840480。

本书有 3 个主要特点：一是新，全书基于最新的 Android 5.0 的代码分析讲解（Android 5.0 中很多代码细节都有改进）；二是全，对 Android 的覆盖面比较广，Android 4.4 和 Android 5.0 中增加的新功能，如 SELinux、ART 等都有详细介绍，同时整本书基于一套源码分析讲解，很多模块前后能相互印证；三是细，除了对各个模块的功能进行详细讲解说明外，对书中摘录的 Android 源码也做了详细注释。

本书面向的读者主要是进行系统开发的工程师，包括手机开发工程师、ROM 开发工程师

和各种使用 Android 作为开发平台的 TV 和可穿戴式设备（Wear）的开发工程师。当然，即使是 Android 应用开发工程师，如果能了解更多的 Android 系统知识，也会对应用开发有相当大的帮助，毕竟 Android 底层的开发资料比较少，一些 API 的解释也不尽人意，在开发应用的过程中，当遇到问题时，如果能打开源码进行查看，很多问题都能轻松解决。

本书导读

阅读系统分析类的书籍，从来不是一件轻松的事情。Android 是一套非常复杂的系统，为了能减轻读者的负担，笔者在介绍每个模块时，尽量先介绍模块的使用、原理或者架构，让读者对某个模块有了初步了解后再通过剖析源码来更深入地介绍细节。

本书摘录了大量的 Android 源码，读者第一次阅读某个章节时，如果觉得枯燥，可以先忽略源码，只阅读笔者的分析文字，有了整体的概念之后，再从头开始理解书中的代码会更容易。

其实，真正掌握某个模块的内容，是必须要能读懂源码的，为了帮助读者达到这个目的，笔者会做尽量详细的分析解释，对于一些复杂的函数，同时会在代码中加上详细的注释。

为了减小代码占用的版面，笔者将代码中的 Log 语句、英文注释，以及一些进行参数检查、错误处理的代码都删除了，然后再加上注释，改动后的代码和原始代码看上去可能有区别，但是逻辑上并没有差别，而且更加容易理解。

如果纯粹使用自然语言去描述某个函数的代码逻辑，可能会非常晦涩难懂。对于软件工程师而言，通常更习惯阅读代码来理解逻辑，因此，笔者在书中尽可能保留一个函数源码的全貌。但是大段的代码也会让人觉得枯燥，因此，对于代码量大的函数，笔者一般会把代码分成小段来分析讲解，这样能保持一页中代码和文字的合适比例，减轻读者的阅读负担。

在阅读本书时，有一点要注意：在 Java 语言中，通常使用“方法”这个称谓，而 C/C++语言中，则习惯使用“函数”。在本书中，笔者会同时使用这两种称谓，因为 Android 的很多模块中，Java 和 C++的代码中会有同名的函数，为了不混淆两者，书中讲述 Java 代码时，尽量使用“方法”这个称谓，讲述 C/C++代码时，则使用“函数”。当然两者并没有本质区别，这是笔者特意为之，是为了更好理解。

本书共 24 章，基本上按照从基础到高级的顺序进行。读者阅读本书时最好能够按照本书的顺序进行，因为后面的章节可能会依赖前面的内容。

下面是本书中所有章节的简单介绍。

第 1 章介绍了如何建立 Android 系统开发的环境。包含开发环境的建立、各种软件的安装，以及系统和内核源码的下载方法。

第 2 章介绍了 Android 的 Build 系统。包括 Build 系统的组成、产品配置文件的内容、Android 模块的编译方式，以及系统和应用签名的原理。

第 3 章介绍了 Android 的 Bionic 库。包括系统调用、内存管理的原理、管道、线程管理的方法、Futex 同步机制、Log 模块、Linker 模块等内容。

第 4 章介绍了 Android 的 Binder 原理。包括 Binder 的使用方式、Binder 服务的开发、Binder 的原理、Binder 驱动，以及匿名共享内存等内容。

第 5 章介绍了 JNI。包括 JNI 的用法和 JNI 环境的创建，以及使用中的各种注意事项。

第 6 章介绍了 Android 的同步和消息机制。包括原子操作的原理、native 层和 Java 层的同步方法及机制、Android 的消息机制、Android 应用进程间消息传递的机制等。

第 7 章介绍了 Android 的 Init 进程。包括 Init 进程的初始化、如何解析启动脚本文件、Android 的属性系统、ueventd 守护进程等内容。

第 8 章介绍了 Android 的 Zygote 进程。包括 Zygote 进程如何初始化、如何启动虚拟机、如何启动应用进程、如何加载系统类和资源等。

第 9 章介绍了 Android 的资源管理系统。包括资源的定义和使用，系统资源的匹配方式以及资源管理的实现方式。

第 10 章介绍了 Android 的 SystemServer 进程。包括进程的初始化过程以及用于监测 System Server 中服务和进程的 WatchDog 模块。

第 11 章介绍了 Android 的应用管理。包括 PackageManagerService 的整个初始化过程、应用的安装过程、系统运行时对应用的管理以及 installd 守护进程。

第 12 章介绍了 Android 对组件的管理。包括 ActivityManagerService 如何管理进程、如何管理 Activity、Service、ContentProvider 和 BroadReceiver 4 大组件。

第 13 章介绍了 Android 的多用户模式。包含 UserManagerService 对用户账号的管理以及 PackageMangerService 服务中如何创建用户等内容。

第 14 章介绍了 Android 的图形显示系统。包含 Surface 的创建过程、Android 图像显示的原理、VSync 信号的生成和分发、SurfaceFlinger 中图像的输出过程等。

第 15 章介绍了 Android 的窗口系统。包括 WindowManagerService 如何管理窗口、如何确定窗口的大小、Z 轴位置、如何播放窗口动画等。

第 16 章介绍了 Android 的输入管理系统。包括 InputManagerService 如何接收和发送输入消息，以及应用进程处理 InputEvent 的过程。

第 17 章介绍了 Android 的电源管理，包括 PowerManagerService 的功能、WakeLock 的实现原理，以及电池管理相关的模块。

第 18 章介绍了 Android 的存储系统，包括 Vold 守护进程、MountService 服务，以及其他存储系统的服务。

第 19 章介绍了 Android 的网络系统。包括 Netd 守护进程、ConnectivityService 服务的主要功能，以及 NetworkManagerService 的作用。

第 20 章介绍了 Android 的音频系统。包括音频系统的架构、AudioPolicyService 的作用、AudioFlingerService 的功能，以及一次完整的播放过程的分析。

第 21 章介绍了 Android 的 SELinux 模块。包括 SELinux 的简介，以及 Android 中如何实现 SELinux 的功能。

第 22 章介绍了 Dalvik 和 ART 虚拟机。包括 Dalvik 虚拟机的创建过程、字节码的执行、内存的管理机制、垃圾回收的过程，以及 ART 的原理等。

第 23 章介绍了 Android 的 Recovery 模块。包含 Recovery 的运行机制、系统升级过程的详细介绍，以及升级脚本的语法规则和执行流程等。

第 24 章介绍了 Android 的调试方法。包括如何分析 Android 的 Log、查找内存泄露的方法，以及如何进行 Android 的自动化测试。

作者简介

刘超，资深 Android 专家、系统架构师。曾任职于四通利方、Motorola、小米等多家著名公司。国内最早的 Android 系统开发者之一，研究 Android 内核多年。主持研发过天语 W606、

酷派 W711、华为 T8301 等多款 Android 手机系统。

在编写本书的过程中，我的父母一直帮忙照看两岁多的女儿，让我能够专心写作，这里首先感谢他们为我辛苦付出。我的女儿每隔一段时间都会来打断我的工作，“提醒”我该休息了，感谢她给我带来的欢乐，支持我能将这项艰苦的工作进行下去。

最后感谢人民邮电出版社的各位领导和编辑，特别是本书的责任编辑张涛，没有你们的辛苦工作，本书的出版不会这么顺利，这里一并表示感谢！本书技术交流 QQ 群 216840480，编辑联系邮箱：zhangtao@ptpress.com.cn。

编　者

目　　录

第 4 章　进程间通信——Android

Binder 是 Android 特有的一种进程间通信（IPC）方式。Android Binder 的前身是 OpenBinder，最早由 Dianne Hackborn 开发并用于 PalmOS 上，后来 Dianne Hackborn 加入 Google，在 OpenBinder 的基础上开发了 Android Binder。

4.1 Binder 简介

Binder 和传统的 IPC 机制相比，融合了远程过程调用（RPC）的概念，而且这种远程调用不是传统的面向过程的远程调用，而是一种面向对象的远程调用。

从 Unix 发展而来的 IPC 机制，只能提供比较原始的进程间通信手段，通信的双方必须处理线程同步、内存管理等复杂问题，不但工作量大，而且很容易出错。除了 Socket、匿名管道（Pipe）以外，传统的 IPC，如命名管道（FIFO）、信号量（Semaphore）、消息队列等已经从 Android 中去掉了。和其他 IPC 相比较，Socket 是一种比较成熟的通信手段，同步控制也很容易实现。Socket 用于网络通信非常合适，但是用于进程间通信，效率就不太高了。

Andoid 在架构上一直希望模糊进程的概念，取而代之以组件的概念。应用不需要关心组件存放的位置、组件运行在哪个进程中、组件的生命周期等问题。随时随地，只要拥有 Binder 对象，就能使用组件的功能。Binder 就像一张网，将整个系统的组件，跨越进程和线程，组织在了一起。

Binder 很强大，也很复杂。但是无论调用 Binder 服务，还是开发一个 Binder 服务都是一件很轻松的事情，不需要考虑线程同步、内存分配等问题，和开发一个普通的类没有太大的区别，所有这些麻烦问题都在 Binder 框架中解决了。

正因为如此，Android 系统的服务都是利用 Binder 构建的。Android 系统中的服务有几十种之多，这是任何一个其他嵌入式平台都不具备的。

Binder 是整个系统运行的中枢，因此，Android 在提高 Binder 的效率方面也下足了功夫。Android 在进程间传递数据使用的是共享内存的方式，这样数据只需要复制一次就能从一个进程到达另一个进程（一般的 IPC 都需要两步，从用户进程复制到内核，再从内核复制到服务进程），这样数据传输的效率就大大提高了。

在安全性方面 Android 也做了考虑，Binder 调用时会传递调用进程的 euid 到服务端，因此，服务端可以通过检查调用进程的权限来决定是否允许其使用所调用的服务。

4.1.1 Binder 对象定义

Binder 涉及的对象比较多，很容易混淆，为了行文方便，这里对 Binder 对象做出了定义，这些并非官方的定义，只是本文使用的代称。

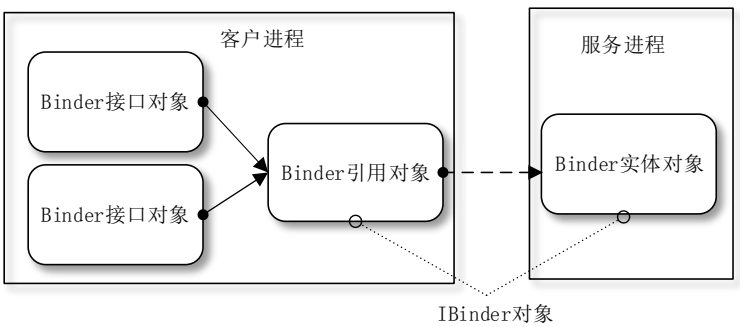
（1）Binder 实体对象：Binder 实体对象就是 Binder 服务的提供者。一个提供 Binder 服务的类必须继承 BBinder 类，因此，有时为了强调对象的类型，也用“BBinder 对象”来代替“Binder 实体对象”。

（2）Binder 引用对象：Binder 引用对象是 Binder 实体对象在客户进程的代表，每个引用对象的类型都是 BpBinder 类，同样可以用名称“BpBinder 对象”来代替“Binder 引用对象”。

（3）Binder 代理对象：代理对象也称为接口对象，它主要是为客户端的上层应用提供接口服务，从 IInterface 类派生。它实现了 Binder 服务的函数接口，当然只是一个转调的空壳。通过代理对象，应用能像使用本地对象一样使用远端的实体对象提供的服务。

（4）IBinder 对象： BBinder 和 BpBinder 类都是从 IBinder 类中继承而来。在很多场合，不需要刻意地去区分实体对象和引用对象，这时可以使用“IBinder 对象”来统一称呼它们。

Binder 代理对象主要和应用程序打交道，将 Binder 代理对象和引用对象分开的好处是代理对象可以有很多实例，但是，它们包含的是同一个引用对象，这样方便了用户层的使用，如图 4.1 所示。



▲图 4.1　Binder 对象关系图

应用完全可以抛开接口对象直接使用 Binder 的引用对象，但是这样开发的程序兼容性不好。也正是在客户端将引用对象和代理对象分离，Android 才能用一套架构来同时为 Java 和 native 层提供 Binder 服务。隔离后，Binder 底层不需要关心上层的实现细节，只需要和 Binder 实体对象和引用对象进行交互。

4.1.2　Binder 的架构

Binder 通信的参与者由 4 部分组成。

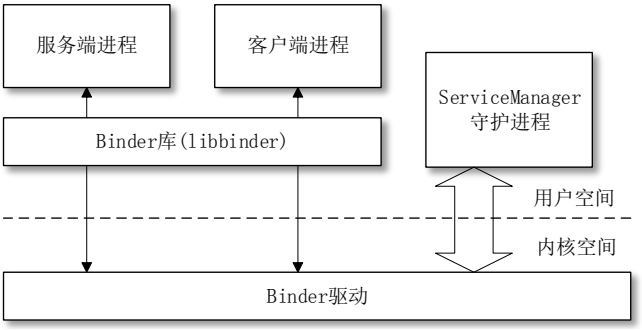
（1）Binder 驱动：Binder 的核心，实现各种 Binder 的底层操作。

（2）ServiceManager：提供 Binder 的名称到引用对象的转换服务。

（3）服务端：Binder 服务的提供者。

（4）客户端：Binder 服务的使用者。

这 4 部分的关系如图 4.2 所示。



▲图 4.2　Binder 架构图

Binder 驱动位于 Binder 架构的核心，通过文件系统的标准接口，如 open()、ioctl()、mmap()等，向用户层提供服务。应用层和 Binder 驱动之间的数据交换是通过 ioctl()接口完成的，read()和 write()两个常用的接口反而没有使用。Binder 的数据交换过程比较复杂，而且涉及两个进程间的数据传输，如果读写用两个不同的接口来分别实现，会导致实现比现在更加复杂。使用 ioctl()的好处是一次系统调用就能完成用户系统和驱动之间的双向数据交换，不但简化了控制逻辑，也提高了传输效率。Binder 驱动的主要功能是提供 Binder 通信的通道，维护 Binder 对象的引用计数，转换传输中的 Binder 实体对象和引用对象以及管理数据缓存区。

ServiceManager 是一个守护进程，它的作用是提供 Binder 服务的查询功能，返回被查询服务的引用。对于一个 Binder 服务，通常会有一个惟一的字符串标识，只要它向 ServiceMananger 注册了这个标识，应用就可以通过标识来获得服务的引用对象。ServiceManager 是一个单独的进程，它是通过什么方式来给其他进程提供查询服务呢？实际上 ServiceManager 也是通过 Binder 框架来提供服务。

作为一个 Binder 服务的提供者，ServiceManager 也需要通过某种方式让使用者得到它的引用对象。Android 使用了一种很巧妙，也很简单的方法来解决这个问题。Binder 引用对象的核心数据是一个实体对象的引用号，它是在驱动内部分配的一个值。Binder 框架硬性规定了 0 代表 ServiceManager。这样用户进程可以使用参数“0”直接构造出 ServiceManager 的引用对象，然后开始使用 ServiceManager 的查询服务。

既然驱动是 Binder 的核心，为什么不直接把查找引用对象的功能放在驱动中完成呢，还要在 ServiceManager 的进程中绕一圈。其实这和 Android 的安全管理有关系。Android 中并不容许任意进程都能注册 Binder 服务，虽然任意一个进程都能创建 Binder 服务，但是，只有 root 进程或 system 进程才可以不受限制地向 ServiceManager 注册服务。ServiceManager 中有一张表，里面规定了能够注册服务的进程的用户 ID，以及进程能够注册的服务名称，ServiceManager 通过这张表来控制普通进程的注册请求（Android 5.0 通过 SELinux 而不是查表的方式来检查权限）。这也是 ServiceManager 存在的原因。

Binder 服务可以分成两种：实名服务和匿名服务。它们从开发到使用没有任何区别，惟一的区别是实名服务能够通过 ServiceManager 查询到。Android 中的实名 Binder 服务都是系统提供的，如 ActivityManagerService、PowerManagerService、WindowManagerService 等。普通应用开发的 Binder 服务，只能是匿名服务。

如果匿名服务不能通过 ServiceManger 查询到，使用者通过什么方式才得到它的引用对象呢？答案还是通过 Binder。匿名服务经常使用的场景是服务进程回调客户进程中的函数。整

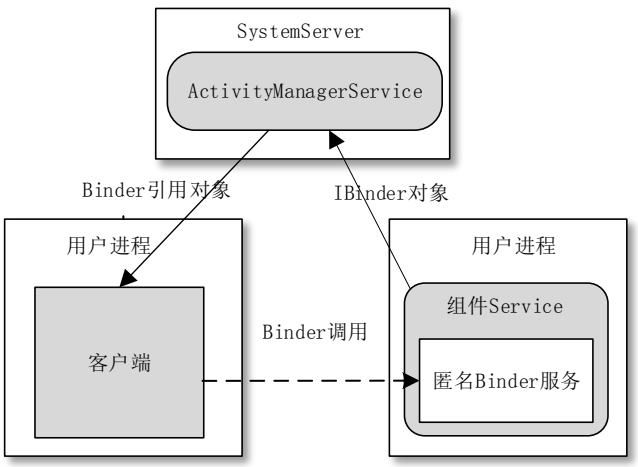
个过程是：客户端和服务端通过 Binder 连接上后，客户端把本进程中创建的匿名服务的实体对象作为函数参数传递到服务端，驱动会在中间把实体对象“转换”成引用对象，这样服务进程就得到了客户进程创建的 Binder 服务的引用对象，然后就能回调客户进程中 Binder 服务的函数了。

在 Java 层 Android 还提供了通过组件 Service 的方式来包装和使用匿名服务。下面详细介绍这种方式。

4.1.3 组件 Service 和匿名 Binder 服务

如果对 Android 的理解不够深入，可能会将 Android4 大组件之一的 Service 和 Binder 服务混为一谈。不过这两者的确关系紧密：组件 Service 其实包含了 Binder 服务。

这里为什么要强调它们的区别呢？首先是为了澄清概念。我们谈论组件 Service 时，更多的是关心它的生命周期、启动和结束之类的问题，而 Binder 服务和生命周期没有太大关系，两者不能混为一谈，否则在开发带有组件 Service 的程序时，可能会对里面的一些细节感到迷惑。同时借这个问题可以更深入地讨论匿名 Binder 服务的使用。匿名服务因为没有 ServiceManager 来提供名称解析服务，因此一般只能用于服务进程回调客户进程。但是，Android 还通过 Framework 提供了一种启动 Java 匿名 Binder 服务的方法。这种方法的过程如下：首先某个应用通过调用 bindService()方法发出一个 Intent，Framework 根据 Intent 找到对应的组件 Service 并启动它，包在组件 Service 中的 Binder 服务也将同时创建出来。随后 Framework 会把服务的 IBinder 对象通过 ConnectivityManager 的回调方法 onServiceConnected()传回到应用，这样应用就得到匿名 Binder 服务的引用对象，也就能使用组件 Service 中的匿名 Binder 服务了。在这里 Android 的 Framework 用 Intent 代替了 Binder 服务的名称来查找对应的服务，同时也承担了 ServiceManager 的工作，解析 Intent 并传回服务的引用对象，如图 4.3 所示。

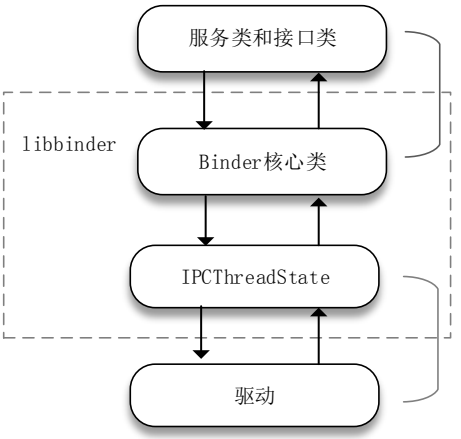


▲图 4.3 组件 Service 和匿名 Binder 服务

Android 的 Framework 并没有使用新的技术来传递 Binder 对象，因为 Framework 中担任这个角色的 ActivityManagerService 本身就是一个 Binder 服务，最终还是通过 Binder 框架在服务端和客户端之间传递了 IBinder 对象。

4.1.4 Binder 的层次

从代码实现上划分，Binder 设计的类可以分成 4 个层次，如图 4.4 所示。



▲图 4.4 Binder 的层次关系图

最上层是位于 Framework 中的各种 Binder 服务类和它们的接口类。这一层的类非常多，例如常见的 ActivityMangerService、WindowManagerService、Package MangerService 等，它们为应用程序提供了各种各样的服务。

最底层的当然是 Binder 驱动。

中间则分成两层，上层是用于服务类和接口类开发的基础，如 Ibinder、Bbinder、BpBinder 等。下层是和驱动交互的 IPCThreadState 和 ProcessState 类。

这里刻意把中间的 libbinder 中的类划分为两个层次的原因，是在这 4 层中，第一层的和第二层联系很紧密，第二层中的各种 Binder 类用来支撑服务类和代理类的开发。但是第三层的 IPCThread 和第四层的驱动之间耦合得很厉害，单独理解 IPCTherad 或者是驱动都是一件很难的事，必须把它们结合起来理解，这一点正是 Binder 架构被人诟病的地方，驱动和应用层之间过于耦合，违反了 Linux 驱动设计的原则，因此，主流的 Linux 并不愿意接纳 Binder。

4.2 如何使用 Binder

本节将介绍如何使用 Binder，包括两部分的内容：一是如何使用已有的 Binder 服务；二是如何开发 Binder 服务。Android 中有大量使用 Binder 的例子，但是比较庞大，不容易看清楚脉络，这里给出一些最简单的例子，便于后面对 Binder 实现原理的分析讲解。

就开发语言而言，Binder 服务可以用 Java 实现，也可以用 C++实现。通常，我们在 Java 代码中调用 Java 语言实现的服务，在 C++代码中调用 C++编写的服务。但是从原理上讲，Binder 并没有这种限制，混合调用也是可以的。

应用可以在任意的进程和线程中使用 Binder 服务的功能，非常方便、灵活、也不用关心同步、死锁等棘手的问题。

4.2.1 使用 Binder 服务

C++层和 Java 层使用 Binder 服务的方式基本一样，包括函数的接口类型都相同，这里介绍就不区分 Java 和 C++了。

使用 Binder 服务前要先得到它的引用对象。例如要得到 CameraService 的引用对象。可以这样做：

```
sp<IServiceManager> sm = defaultServiceManager();
sp<IBinder> binder = sm->getService("media.camera");
```

defaultServiceManager()方法用来得到 ServiceManager 服务的引用对象。前面已经介绍了，ServiceManager 的引用对象是直接用数值 0 作为引用号构造出来的。ServiceManager 的 getService()方法的作用是查找注册的 Binder 服务。如果 getService()找到名称对应的服务，它会返回服务的

IBinder 对象，否则返回 NULL。

getService()返回值的类型是 IBinder，实际上是引用对象，但是，应用需要使用的是 Binder 的代理对象，因此，在使用前需要把引用对象转换成代理对象，转换的方法如下：

```
ICameraService service = ICameraService.Stub.asInterface(binder);
```

Binder 中提供了 asInterface()方法来完成这种转换，asInterface()方法会检查参数中的 IBinder 对象的服务类型是否相符，如果不符将返回 NULL。得到了服务的代理对象后，接下来就可以像使用普通对象一样使用 Binder 服务了。

正常情况下，ServiceManager 进程会在所有应用启动前启动，而且也不会停止服务，因此，使用 defaultServiceManager()方法时可以不检查它的返回值是否为 NULL，但是，对方法 getService()的返回值则需要检查，因为查找的服务有可能还没有启动。

4.2.2 Binder 的混合调用

Binder 是可以混合调用的，在 C++的代码中可以调用 Java 语言编写的 Binder 服务，从 Java 语言中也可以调用 C++语言编写的服务。调用的方法其实并不神秘，就是直接使用服务的引用对象。

Binder 的服务端用号码来表示它所提供的函数，客户端调用某个 Binder 服务时要使用函数号来表示要调用的函数。下面看一个例子，从 C++层调用 Java 服务 ActivityManagerService 的方法 cehckUriPermission()。这个方法的原型是：

```
int checkUriPermission(Uri uri, int pid, int uid, int mode)。
```

客户端中的调用代码可以这样写：

```
sp<IBinder> binder = defaultServiceManager()->getService("activity");
Parcel data = Parcel.obtain();
Parcel reply = Parcel.obtain();
data.writeInterfaceToken("android.app.IActivityManager ");
uri.writeToParcel(data, 0);
data.writeInt(pid);
data.writeInt(uid);
data.writeInt(mode);
binder->transact(54, data, reply, 0);           // 54 代表了方法 checkUriPermission
reply.readException();
int res = reply.readInt();
```

这段代码使用 Parcel 类打包参数，然后调用 IBinder 的函数 transact()来调用远程服务。这里最关键的是传递给 transact()函数的第一个参数：远程函数号 54。这个号码在服务端的代码中定义，而且有可能随着 Android 版本的升级而改变。正是因为函数号不能很方便地获得，所以，这种方法很少使用。笔者在这里介绍混合调用方法并不是推荐大家这样来使用，只是通过这种方式来让读者更进一步了解 Binder 的本质，而不是被一些表面的东西给束缚了。

4.2.3 用 Jave 开发 Binder 服务

Binder 使用简单，开发一个 Binder 服务就稍微复杂些。比较而言，开发一个 Java 服务还是比 C++服务简单得多。

下面我们一起写一个简单的组件 Service，前面已经介绍过，它里面包含有一个匿名的 Binder 服务。

（1）第一步，编写 AIDL 文件，定义两个简单的方法 get()和 set()，如下所示：

```
interface IExampleService {
   int get ();
   void set (int value);
}
```

AIDL 是 Android Interface Describation Language 的缩写，它是 Android 提供的一种用来简化 Binder 编程的脚本语言。开发人员只需要在 AIDL 文件中定义出方法接口，AIDL 解释器能自动生成服务器和客户端需要的 Java 代码，这也是 Java 服务比 C++服务更容易开发的原因。

（2）第二步，编写 Service 的代码：

```
public class ExampleService extends Service {
   int mValue = 0;
   private final IExampleService.Stub mInstance = new IExampleService.Stub() {
      public int get () {
         return mValue;
      }
      public void set (int value) {
         mValue = value;
      }
    };
    @Override
    public IBinder onBind(Intent intent) {
      return mInstance ;
    }
}
```

ExampleService 需要从 Service 类继承，而且必须重载底层的 onBind()方法，并在其中返回 Binder 服务的实体对象 mInstance。IExampleService.Stub 类是真正的 Binder 服务类，它是通过前面的 aidl 文件自动生成的，因此，这里还需要继承 IExampleService.Stub 并重载它的方法 get()和 set()，并实现它们的功能。

（3）第三步，在 AndroidManifest.xml 中加入服务的声明：

```
<service android:name=".ExampleSerivce" >
   <intent-filter>
      <action android:name="com.android.IExampleService" />
   </intent-filter>
 </service>
```

这里加入声明的作用是为了让 Framework 能通过 Intent 来找到 Service。

在 Java 层开发 Binder 服务并不一定要使用组件 Service，我们也可以开发一个单纯的 Java Binder 服务，但是，在应用中写出来的 Binder 服务无法被外界使用，还需要通过某种途径把它的引用对象传递到使用者手中，因此这里给出了一个使用组件 Service 的例子。

4.2.4 用 C++开发 Binder 服务

用 C++来实现 Binder 服务比较麻烦的原因是没有 AIDL 的辅助，必须手工来写中间层的代码，但是，这样反而有利于我们了解 Binder 的框架。下面我们把前面这个简单的服务用 C++再写一遍。

首先，编写服务类 ExampleService 的代码：

```
class ExampleService : public BnExampleService
{
public:
   ExampleService ():mValue(0);
   public int get () { return mValue; }
```

```
    public void set (int value) { mValue = value;}
private
    int mValue;
};
```

服务类的代码很简单，基本上和写一个普通的 C++类没有太大的区别，惟一的要求就是必须继承自定义的类 BnExampleService。BnExampleService 类的定义如下：

```
class IExampleService : public IInterface
{
public:
    DECLARE_META_INTERFACE(ExampleService);
    virtual int get () = 0;
    virtual void set(int value) = 0;
};
// ----------------------------------------------------------
class BnExampleService : public BnInterface< IExampleService >
{
public:
    virtual status_t  onTransact(uint32_t code, const Parcel& data,
                        Parcel* reply, uint32_t flags = 0);
};
```

BnExampleService 类看上去很简单，只重载了底层的方法 onTransact()，但是它继承了模板类 BnIntface< IExampleService.>。

IExampleService 是一个接口类，也需要我们自己定义，IExampleService 必须从 IInterface 类派生，主要的功能是用“纯虚函数”定义服务的接口函数，因此，在 ExampleService 类中必须实现这些接口函数。至于这些模板类的作用，后面会详细介绍。IExampleService 里还使用了一个看上去比较陌生的宏 DECLARE_META_INTERFACE，后面也会对这个宏详细介绍。这里先实现 BnExampleService 类的代码。

BnExampleService 类的代码实现如下，同时下面的代码中还定义和实现了 BpExampleService 类：

```
enum {
    EXAMPLE_GET = IBinder::FIRST_CALL_TRANSACTION,
    EXAMPLE_SET,
};
class BpExampleService : public BpInterface< IExampleService >
{
public:
    BpExampleService (const sp<IBinder>& impl)
    : BpInterface< IExampleService >(impl){ }
    virtual int get()
    {
        Parcel data, reply;
        data.writeInterfaceToken(IExampleService::getInterfaceDescriptor());
        remote()->transact(EXAMPLE_GET, data, &reply);
        return reply.readInt();
    }
    virtual void set(int value)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IExampleService::getInterfaceDescriptor());
        data.wriitInt(value);
        remote()->transact(EXAMPLE_SET, data, &reply);
        return;
    }
};
IMPLEMENT_META_INTERFACE(ExampleService, "ExampleDescriptor");
// ----------------------------------------------------------------------
status_t BnExampleService::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code){
```

```
        case EXAMPLE_GET:
            CHECK_INTERFACE(IExampleService, data, reply);
            reply->writeInt(get());
            break;
        case EXAMPLE_SET:
            CHECK_INTERFACE(IExampleService, data, reply);
            set(data->readInt());
            break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
    return NO_ERROR;
}
```

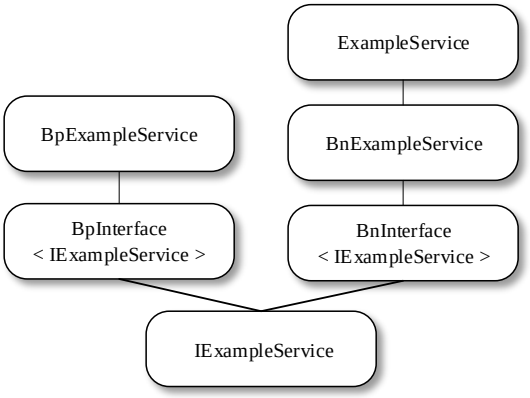
BnExampleService 类只需要实现函数 onTransact()，这个函数在运行时会被底层的 IPCThread State 类调用。onTransact()的实现比较公式化，就是根据参数中的函数号来调用对应的函数。每段调用代码的写法也很固定，首先调用宏 CHECK_INTERFACE 来检查 Parcel 对象中的数据是否正确，检查的原理是取出 Parcel 对象中从客户端传递过来的类描述字串，并和类中定义的字符串相比较，如果相同就认为正确。接着按照函数参数的排列顺序取出参数，并使用这些参数调用服务端的函数。调用结束后通过 reply 对象把函数的返回值传递回去。

和 BnExampleService 类的实现同在一个文件中的还有 BpExampleService 类，它是 Binder 服务在客户端的接口类，将来它的对象实例会运行在客户端的进程中，但是 BnExampleService 类的对象会运行在服务进程中。这里为什么要把它们的代码放在一个文件中呢？主要是因为两个类中都使用了相同的函数号码定义和类的描述字符串。如果把它们放在不同的文件中，这些定义可能要写两份，容易产生不一致。因此包含这两个类的文件 IExampleService.cpp 既要参与服务器模块的编译，也要参与客户端模块的编译，但是不要误认为它们一定会在一个进程中运行。

BpExampleService 类被要求从模板类 BpInterface< IExampleService >中派生。同样因为 IExampleService 把接口都定义成了“纯虚函数”，因此 BpExampleService 中也必须实现这些接口。BpExampleService 类中这些接口的作用是把函数调用的信息发送给远端，每个接口的实现方式基本上一致。首先是把类的描述字串放到 parcel 中，接着把调用的参数依次放入，最后通过调用 remote()函数得到 Binder 引用对象的指针，然后调用它的 transcat()函数把数据传递给底层，如图 4.5 所示。

IBinder 的 transcat()函数其实有 4 个参数，由于第四个参数 flags 定义了缺省值 0，因此调用 transcat()时常常只使用 3 个参数。如果客户端调用 Binder 服务时不打算等待对方运行结束就继续运行，可以使用 TF_ONE_WAY 作为第四个参数来调用 transcat()函数。这表明可以使用一种异步的方式调用 Binder 的服务，当然这种方式也意味着无法得到返回值。

这样，Binder 服务就开发完成了。通过下面的代码就能将这个简单的服务运行起来：



▲图 4.5　C++开发 Binder 服务所设计的类

```
ExampleService * service = new ExampleService()
defaultServiceManager()->addService(String16 ("example_service" ), service);
```

最后还有一个问题，应用如何调用开发的新服务呢？细心的读者可能注意到了，我们并没有

给客户端的接口类 BpExampleService 提供头文件。实际上，客户端的程序使用的是头文件 IExampleService.h 中定义的 IExampleService，Android 能保证最后调用的是 BpExampleService 中的函数。具体的原理下节会详细介绍。

4.3 Binder 应用层的核心类

上节的例子中我们使用了一些 libbinder 库中提供的类来开发 Binder 服务。本节将分析这些类的作用和实现。了解了这些类，才能完全掌握 Binder 服务开发的要点，同时也对 Binder 框架有更深入的了解。

libbinder 库中的 IInterface 类、BpInterface 类、BnInterface 类、BBinder 类、BpBinder 类和 IBinder 类共同构成了 Binder 应用层的核心类。

第 6 章　Android 的同步和消息机制

前面介绍了，Android 程序运行时会根据需要自动产生 Binder 线程，因此，即使上层应用代码不创造任何线程，Android 应用进程中还是会有多个线程在运行。包含有多个运行线程的程序会不可避免地遇到资源竞争的问题，这里谈到的资源可能是一个全局变量，也可能是系统的硬件资源，如扬声器等。Linux 下线程的运行模式是抢占式，因此，程序可能会在任何时刻失去对 CPU 的占用。为了防止使用全局资源时因为线程的切换出现错误，通常需要使用系统提供的同步机制来“独占”全局资源的访问权。

虽然同步机制能解决资源访问的冲突问题，但也不可避免地带来了性能上的损失。因此，在不影响正确性的前提下，应当尽量避免使用同步机制。

6.1 原子操作

对简单类型的全局变量进行操作时，即使是一些简单的操作，如加法、减法等，在汇编级别上也需要多条指令才能完成。整个操作的完成需要先读取内存中的值，在 CPU 中计算，然后再写回内存中。如果中间发生了线程切换并改变了内存中的值，这样最后执行的结果就会发生错误。避免这种问题发生的最好办法就是使用原子操作。

原子操作中没有使用锁，从效率上看要比使用锁来保护全局变量划算。但是，原子操作也不是没有一点性能上的代价，因此还是要尽量避免使用。

Android 中用汇编语言实现了一套原子操作函数，这些函数在同步机制的实现中被广泛使用。

6.1.1 Android 的原子操作函数

1．原子变量的加法操作

```
int32_t android_atomic_add(int32_t value, volatile int32_t* addr);
```

原子变量的减法操作可以通过传递负值给加法操作函数来完成。

2．原子变量的自增和自减操作

```
int32_t android_atomic_inc(volatile int32_t* addr);
int32_t android_atomic_dec(volatile int32_t* addr);
```

3．原子变量的与操作

```
int32_t android_atomic_and(int32_t value, volatile int32_t* addr);
```

4．原子变量的或操作

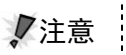
```
int32_t android_atomic_or(int32_t value, volatile int32_t* addr);
```

5．原子变量的设置

```
void android_atomic_acquire_store(int32_t value, volatile int32_t* addr);
void android_atomic_release_store(int32_t value, volatile int32_t* addr);
```

6．原子变量的读取

```
int32_t android_atomic_acquire_load(volatile const int32_t* addr);
int32_t android_atomic_release_load(volatile const int32_t* addr);
```



上面这两个函数从功能上看是一样的，区别只是内存屏障位于读取前还是读取后，下面的两组函数也类似。

7．原子变量的比较并交换

```
int  android_atomic_acquire_cas(int32_t  oldvalue,  int32_t  newvalue,  volatile
int32_t* addr);
int  android_atomic_release_cas(int32_t  oldvalue,  int32_t  newvalue,  volatile
int32_t* addr);
```

8．还有两个原子变量的宏定义

```
#define android_atomic_write android_atomic_release_store
#define android_atomic_cmpxchg android_atomic_release_cas
```

6.1.2 原子操作的实现原理

Android 原子操作的实现方式和 CPU 的架构有密切关系，现在的原子操作一般都是在 CPU 指令级别实现的。这种实现方式不但简单，而且效率非常高。

虽然原子操作的接口函数有 10 多个，但是，只有两个函数通过汇编代码真正实现了原子操作，它们是函数 android_atomic_add()和 android_atomic_cas()，其他函数都只是在内部调用它们而已。这两个函数的原理差不多。

ARM 平台上的实现更复杂一点，下面以 ARM 平台的加法函数为例来分析原子变量的实现原理：

```
extern ANDROID_ATOMIC_INLINE
int32_t android_atomic_add(int32_t increment, volatile int32_t *ptr)
{
    int32_t prev, tmp, status;
    android_memory_barrier();
    do {
        __asm__ __volatile__ ("ldrex %0, [%4]\n"
                              "add %1, %0, %5\n"
                              "strex %2, %1, [%4]"
                              : "=&r" (prev), "=&r" (tmp),
                                "=&r" (status), "+m" (*ptr)
```

```
                          : "r" (ptr), "Ir" (increment)
                          : "cc");
    } while (__builtin_expect(status != 0, 0));
    return prev;
}
```

上面的代码中第一行使用的宏 ANDROID_ATOMIC_INLINE 的定义如下：

```
#define ANDROID_ATOMIC_INLINE inline __attribute__((always_inline))
```

这个宏的作用是把函数定义成了 inline 函数。

代码中第二行调用 android_memory_barrier()函数的作用表示这里需要内存屏障（下节会介绍内存屏障）。

接下来是一段“内嵌汇编”（如果对“内嵌汇编”不了解，可以参考笔者的博客），“内嵌汇编”比较难懂，但是可以用下面这段展开的伪代码来表示它：

```
do {
    ldrex  prev , [ptr]
    add  tmp,  prev,  increment
    strex  status,  tmp, [ptr]
} whiile(status != 0)
```

在 add 指令的前后有两条看上去比较陌生的指令：ldrex 和 strex，这两条是 AMRV6 新引入的同步指令。ldrex 指令的作用是将指针 ptr 指向的内容放到 prev 变量中，同时给执行处理器做一个标记(tag)，标记上指针 ptr 的地址，表示这个内存地址已经有一个 CPU 正在访问。当执行到 strex 指令时，它会检查是否存在 ptr 的地址标记，如果标记存在，strex 指令会把 add 指令执行的结果写入指针 ptr 指向的地址，并且返回 0，然后清除该标记。返回的结果 0 将保存在 status 变量中，这样循环结束，函数返回结果。

如果在 strex 指令执行前发生了线程的上下文切换，在切换回来后，ldrx 指令设置的标志将会被清除。这时再执行 strex 指令时，由于没有了这个标志，strex 指令将不会完成对 ptr 指针的存储操作，而且 status 变量中的返回结果将是 1。因此，循环将重新开始执行，直到成功为止。

__builtin_expect()是 gcc 的内建函数，有两个参数，第一个参数是一个表达式，第二个参数是一个值。表达式的计算结果也是函数的结果。__builtin_expect()用来告诉 gcc 预测表达式更可能的值是什么，这样 gcc 会根据预测值来优化代码。代码中表达的含义是预测“status!=0”这个表达式的值为“0”，预测 while 循环将结束。

原子操作并没有禁止中断的发生或上下文切换，而是让它们不影响操作的结果。

6.1.3 内存屏障和编译屏障

现代 CPU 中指令的执行次序不一定按顺序执行，没有相关性的指令可以打乱次序执行，以充分利用 CPU 的指令流水线，提高执行速度。同时，编译器也会对指令进行优化，例如，调整指令顺序来利用 CPU 的指令流水线。这些优化方式，大部分时候都工作良好，但是在一些比较复杂的情况可能会出现错误，例如，执行同步代码时就有可能因为优化导致同步原语之后的指令在同步原语前执行。

内存屏障和编译屏障就是用来告诉 CPU 和编译器停止优化的手段。编译屏障是指使用伪

指令“memory”告诉编译器不能把“memory”执行前后的代码混淆在一起，这时“memory”起到了一种优化屏障的作用。内存屏障是在代码中使用一些特殊指令，如 ARM 中的 dmb、dsb 和 isb 指令，x86 中的 sfence、lfence 和 mfence 指令。CPU 遇到这些特殊指令后，要等待前面的指令执行完成才执行后面的指令。这些指令的作用就好像一道屏障把前后指令隔离开了，防止 CPU 把前后两段指令颠倒执行。

（1）ARM 平台的内存屏障指令。

- dsb：数据同步屏障指令。它的作用是等待所有前面的指令完成后再执行后面的指令。
- dmb：数据内存屏障指令。它的作用是等待前面访问内存的指令完成后再执行后面访问内存的指令。
- isb：指令同步屏障。它的作用是等待流水线中所有指令执行完成后再执行后面的指令。

（2）x86 平台上的内存屏障指令。

- sfence：存储屏障指令。它的作用是等待前面写内存的指令完成后再执行后面写内存的指令。
- lfence：读取屏障指令。它的作用是等待前面读取内存的指令完成后再执行后面读取内存的指令。
- mfence：混合屏障指令。它的作用是等待前面读写内存的指令完成后再执行后面读写内存的指令。

要精确地理解这些指令的含义，需要去查阅处理器的说明。这里只是对它们做了一点简单的介绍。下面看看 Android 是如何利用这些指令来实现内存屏障和编译屏障的：

1．ARM 平台的函数代码

（1）编译屏障：

```
void android_compiler_barrier()
{
    asm_volatile_("" : : : "memory");
}
```

编译屏障的实现只是使用了伪指令 memory。

（2）内存屏障：

```
void android_memory_barrier()
{
#if ANDROID_SMP == 0
    android_compiler_barrier();
#else
    __asm__volatile_("dmb" : : : "memory");
#endif
}

void android_memory_store_barrier()
{
#if ANDROID_SMP == 0
    android_compiler_barrier();
#else
    __asm_volatile_("dmb st" : : : "memory");
#endif
}
```

内存屏障的函数中使用了宏 ANDROID_SMP。它的值为 0 时表示是单 CPU，这种情况下只使用编译屏障就可以了。在多 CPU 情况下，同时使用了内存屏障指令“dmb”和编译屏障的伪指令“memory”。函数 android_memory_store_barrier()中的 dmb 指令还使用了选项 st，它表示要等待前面所有存储内存的指令执行完后再执行后面的存储内存的指令。

2．x86 平台下的函数代码

（1）编译屏障：

```
void android_compiler_barrier(void)
{
    __asm__ __volatile__ ("" : : : "memory");
}
```

和 ARM 平台下一样，编译屏障的实现只是使用了伪指令 memory。

（2）内存屏障：

```
#if ANDROID_SMP == 0
void android_memory_barrier(void)
{
    android_compiler_barrier();
}

void android_memory_store_barrier(void)
{
    android_compiler_barrier();
}
#else
void android_memory_barrier(void)
{
    asm__volatile_("mfence" : : : "memory");
}
void android_memory_store_barrier(void)
{
    android_compiler_barrier();
}
#endif
```

x86 平台也一样，如果是单 CPU，内存屏障的实现只使用了编译屏障。在多 CPU 情况下，函数 android_memory_barrier()使用了 CPU 指令“mfence”，对读写内存的情况都进行了屏障。但是 android_memory_store_barrier()函数只使用了编译屏障，这是因为 Intel 的 CPU 不对写内存的指令重新排序。所以不需要内存屏蔽指令。

Android native 层的同步方法

第 23 章　系统升级模块

——Android 的 Recovery 模

Recovery 是 Android 用来执行系统升级的模块。Android 缺省的 Recovry 使用不是很方便，因此，第三方的 Recovry 更加流行。这些第三方的 Recovery 主要是对操作方式进行了改进，核心的升级方式和 Android 的实现还是一致的。从研究 Android 的实现原理出发，原生的 recovery 更容易理解。

Android 设备启动时会根据检测到的按键组合来决定是否进入 Recovery 模式。Recovery 模式下同样会装载 Linux 内核，这个内核和正常启动时的内核一致的，只是不会进入图像模式。Recovry 模块的主要功能是使用更新包升级系统。我们先看看更新包的组成，如图 23.1 所示。

```
|--boot.img
|--system/
|--recovery/
    |--recovery-from-boot.p
    |--etc/
       |--install-recovery.sh
 |--META-INF/
   |--CERT.RSA
   |--CERT.SF
   |--MANIFEST.MF
   |---com/
       |--google/
            |--android/
               |--update-binary
               |--updater-script
               |--metadata
```

图 23.1　更新包的组成

- boot.img 用来更新 boot 分区，它由 Linux kernel 和根文件系统的映像 ramdisk 组成。boot.img 不是升级包中必须包含的文件。
- system 目录下的文件用来替换系统的 System 分区中的文件，包括各种应用、系统库和系统配置文件等。
- recovery 目录下存放的文件用于 Recovery 模块的升级。其中 recovery-from-boot.p 文件用来更新 recovery 分区，子目录 etc 中的 install-recovery.sh 文件是更新脚本。
- META-INF 目录下存放的是更新包的签名文件和更新脚本。只有更新包的签名和设备的签名相匹配才能进行系统更新。其中的 CERT.RSA 和 CERT.SF 文件就是签名文件。而 com/google/android 目录中的 update-script 是升级用的脚本文件，update-binary 是一个可执行文件，

用来解释执行 update-script 文件，完成系统的更新操作，metadata 文件包含了一些设备信息，如设备型号、时间、版本类型等。一些第三方的 Recovery 模块会去掉升级包的签名校验，这样手机就能刷第三方的 ROM 了。

23.1 Recovery 模块的执行

Recovery 模块的源码位于目录 bootable/recovry 中。下面我们从模块的入口函数 main()开始分析，一步步了解整个 Recovery 模块的工作原理。

23.1.1 Recovery 模块的启动

main()函数的代码如下：

```
int main(int argc, char **argv) {
    time_t start = time(NULL);
    if (argc == 2 && strcmp(argv[1], "--adbd") == 0) {
        adb_main();    // 如果启动参数带有 adbd，则作为 adbd 的 daemon 进程运行
        return 0;
    }
    load_volume_table();                    // 读取/etc/recovery.fstab 文件中的分区内容
    ensure_path_mounted(LAST_LOG_FILE);
    rotate_last_logs(10);
    get_args(&argc, &argv);                                 // 获得启动参数

    const char *send_intent = NULL;
    const char *update_package = NULL;
    int wipe_data = 0, wipe_cache = 0, show_text = 0;
    bool just_exit = false;
    bool shutdown_after = false;
    // 解析启动参数
    int arg;
    while ((arg = getopt_long(argc, argv, "", OPTIONS, NULL)) != -1) {
        switch (arg) {
        case 's': send_intent = optarg; break;
        case 'u': update_package = optarg; break;       // 升级系统
        case 'w': wipe_data = wipe_cache = 1; break;    // 擦除数据
        case 'c': wipe_cache = 1; break;                // 擦除 Cache
        case 't': show_text = 1; break;                 // 指示升级时是否显示 UI
        case 'x': just_exit = true; break;              // 退出
        case 'l': locale = optarg; break;               // 指定 locale
        case 'p': shutdown_after = true; break;         // 退出 recover 后关闭系统
        case '?':
            LOGE("Invalid command argument\n");
            continue;
        }
    }
    if (locale == NULL) {                   // 如果没有指定 locale
        load_locale_from_cache();           // 从文件/cache/recovery/last_locale 中读取
    }
    Device* device = make_device();         //  创建 DefaultDevice 对象
    ui = device->GetUI();                   //  得到 Device 中创建的 RecoveryUI 对象
    gCurrentUI = ui;
    // 初始化 RecoveryUI 对象
    ui->SetLocale(locale);                  // 为 UI 设置 locale，这样就能以合适的语音显示
    ui->Init();
    ui->SetBackground(RecoveryUI::NONE);
    if (show_text) ui->ShowText(true);      // 如果启动参数指定了显示 UI，设置在 UI 对象中

    struct selinux_opt seopts[] = {
      { SELABEL_OPT_PATH, "/file_contexts" }
    };
    sehandle = selabel_open(SELABEL_CTX_FILE, seopts, 1);
    if (!sehandle) {
        ui->Print("Warning: No file_contexts\n");
    }
    device->StartRecovery();                // 空函数
    // 打印输出命令
    printf("Command:");
```

```
for (arg = 0; arg < argc; arg++) {
    printf(" \"%s\"", argv[arg]);
}
printf("\n");
if (update_package) {                                   // 预处理更新命令
    // 如果更新命令以"CACHE:"，把它更换成实际的 cache 目录路径
    if (strncmp(update_package, "CACHE:", 6) == 0) {
        int len = strlen(update_package) + 10;
        char* modified_path = (char*)malloc(len);
        strlcpy(modified_path, "/cache/", len);
        strlcat(modified_path, update_package+6, len);
        printf("(replacing path \"%s\" with \"%s\")\n",
              update_package, modified_path);
        update_package = modified_path;
    }
}
printf("\n");
// 获取一些 property 的值
property_list(print_property, NULL);
property_get("ro.build.display.id", recovery_version, "");
printf("\n");

int status = INSTALL_SUCCESS;
if (update_package != NULL) {
    // 如果命令是更新系统
    status = install_package(update_package, &wipe_cache,
                             TEMPORARY_INSTALL_FILE);
    if (status == INSTALL_SUCCESS && wipe_cache) {
        // 如果更新成功，并且启动参数要求擦除 cache
        if (erase_volume("/cache")) {   // 擦除 cache 目录
            LOGE("Cache wipe (requested by package) failed.");
        }
    }
    if (status != INSTALL_SUCCESS) {
        // 安装失败，打印提示
        ui->Print("Installation aborted.\n");   // 在屏幕上打印失败的提示
        char buffer[PROPERTY_VALUE_MAX+1];
        property_get("ro.build.fingerprint", buffer, "");
        if (strstr(buffer, ":userdebug/") || strstr(buffer, ":eng/")) {
            ui->ShowText(true);     // 在屏幕上打印失败的版本信息
        }
    }
} else if (wipe_data) {     // 如果是擦除数据区的命令
    if (device->WipeData()) status = INSTALL_ERROR;          // 成功返回 0
    if (erase_volume("/data")) status = INSTALL_ERROR;       // 删除/data 目录
    if (wipe_cache && erase_volume("/cache")) status = INSTALL_ERROR;
    if (status != INSTALL_SUCCESS) ui->Print("Data wipe failed.\n");
} else if (wipe_cache) {    //如果是擦除 cache 目录的命令
    if (wipe_cache && erase_volume("/cache")) status = INSTALL_ERROR;
    if (status != INSTALL_SUCCESS) ui->Print("Cache wipe failed.\n");
} else if (!just_exit) {    // 只是退出命令
    status = INSTALL_NONE;  // No command specified
    ui->SetBackground(RecoveryUI::NO_COMMAND);
}
if (status == INSTALL_ERROR || status == INSTALL_CORRUPT) {
    copy_logs();
    ui->SetBackground(RecoveryUI::ERROR);    // 执行命令错误，在屏幕上显示标志
}
if (status != INSTALL_SUCCESS || ui->IsTextVisible()) {
    prompt_and_wait(device, status);  // 进入菜单模式
}
// 结束 recovery，根据参数，关闭或重启系统
if (shutdown_after) {
    ui->Print("Shutting down...\n");
    property_set(ANDROID_RB_PROPERTY, "shutdown,");
} else {
    ui->Print("Rebooting...\n");
    property_set(ANDROID_RB_PROPERTY, "reboot,");
}
return EXIT_SUCCESS;
```

```
}
```

main()函数的主要流程是。

（1）调用 load_volume_table()函数，读取/etc/recovery.fstab 文件的内容，这个文件中保存的是进入 recovery 模式后系统的分区情况，包括分区名称、参数等。

（2）调用 get_args()函数读取 recovey 的启动参数，如果命令行中有 recovery 命令，则优先执行命令行的 recovery 命令，否则读取 misc 分区中的命令。如果 misc 分区中不存在命令，则执行/cache/recovery/command 文件中的命令。get_args()函数则通过 get_bootloader_message()来读取 misc 分区中的命令。

（3）调用 make_device()创建 Device 对象，这里实际上创建的是 DefaultDevice 对象，它继承自 Device 类，DefaultDevice 对象又创建了 DefaultUI 对象，如下所示：

```
    DefaultDevice() :
        ui(new DefaultUI) {
}
```

我们会看到 Device 的很多函数都是空函数，并没有实现。为什么会这样呢？因为 Android 提供的 Recovery 模块可以由厂商进行个性化开发，厂商通过重载 Device 类来加入自己特有的功能。这些空函数就是留给厂商实现的。

（4）执行 DefaultUI 对象的初始化。Recovery 的 UI 就是简单的控制台文本输出界面。

（5）执行更新系统的命令。如果从参数中得到的是更新系统的命令，则调用 install_package()函数来更新，这个命令后面会重点介绍。

（6）执行删除/data 分区和/cache 分区的命令

（7）执行完命令后，如果命令执行不成功，或者调用 DefaultUI 对象的 IsTextVisible()函数返回 true，则调用 prompt_and_wait 显示菜单。IsTextVisible()函数根据参数中的 show_text 选项来进行判断，如下所示

```
bool ScreenRecoveryUI::IsTextVisible()
{
    pthread_mutex_lock(&updateMutex);
    int visible = show_text;
    pthread_mutex_unlock(&updateMutex);
    return visible;
}
```

（8）退出 recovery。根据参数，选择关闭或重启系统。

main()函数的逻辑并不复杂，下面再看看这个过程中的一些细节，例如，recovery 模块启动时是如何得到参数的。

23.1.2 如何传递启动参数

Bootloader 和 Recovery 模块以及主系统之间的通信是通过系统的 misc 分区来完成的。Misc 分区只有 3 页（Page）大小。描述 misc 分区的数据结构是 bootloader_message，定义如下：

```
struct bootloader_message {
    char command[32];
    char status[32];
    char recovery[768];
};
```

- command 字段中存储的是命令，如果它的值是“boot-recovery”，系统将进入 Recovery 模

式。如果它的值是“update-radia”或者“update-hboot”，系统会进入更新 firmware 的模式，这个更新过程由 bootloader 完成。如果 command 中的值为 NULL，则进入主系统，正常启动。

- status 字段存储的是更新的结果。更新结束后，由 Recovery 或者 Bootloader 将更新结果写入到这个字段中。

- recovery 字段存放的是 recovry 模块的启动参数。内容以“recovery”字符串开头，后面是 recovery 中执行的命令，命令之间以“\n”分割。

下面看看 Recovery 模块获取参数的函数 get_args()，代码如下：

```
static void get_args(int *argc, char ***argv) {
    struct bootloader_message boot;
    memset(&boot, 0, sizeof(boot));
    get_bootloader_message(&boot);  // 读取/misc 分区中的命令到 boot 变量中

    if (boot.command[0] != 0 && boot.command[0] != 255) {
        LOGI("Boot command: %.*s\n", sizeof(boot.command), boot.command);
    }
    if (boot.status[0] != 0 && boot.status[0] != 255) {
        LOGI("Boot status: %.*s\n", sizeof(boot.status), boot.status);
    }
    if (*argc <= 1) { // 如果命令行没有传递参数
        boot.recovery[sizeof(boot.recovery) - 1] = '\0';  // Ensure termination
        const char *arg = strtok(boot.recovery, "\n"); // arg 指向 boot 变量的 recvory
        if (arg != NULL && !strcmp(arg, "recovery")) {
            // 如果字符串以 recovery 开头，则使用从/misc 分区中读取的命令串建立启动参数
            *argv = (char **) malloc(sizeof(char *) * MAX_ARGS);
            (*argv)[0] = strdup(arg);
            for (*argc = 1; *argc < MAX_ARGS; ++*argc) {
                if ((arg = strtok(NULL, "\n")) == NULL) break;
                (*argv)[*argc] = strdup(arg);
            }
            LOGI("Got arguments from boot message\n");
        } else if (boot.recovery[0] != 0 && boot.recovery[0] != 255) {
            LOGE("Bad boot message\n\"%.20s\"\n", boot.recovery);
        }
    }
    if (*argc <= 1) { // 如果从/misc 分区中也没有读到命令
        FILE *fp = fopen_path(COMMAND_FILE, "r"); // 打开/cache/recovery/command 文件
        if (fp != NULL) {
            char *token;
            char *argv0 = (*argv)[0];
            *argv = (char **) malloc(sizeof(char *) * MAX_ARGS);
            (*argv)[0] = argv0;  // use the same program name
            char buf[MAX_ARG_LENGTH];
            // 使用读取的文件内容建立启动参数
            for (*argc = 1; *argc < MAX_ARGS; ++*argc) {
                if (!fgets(buf, sizeof(buf), fp)) break;
                token = strtok(buf, "\r\n");
                if (token != NULL) {
                    (*argv)[*argc] = strdup(token);  // Strip newline.
                } else {
                    --*argc;
                }
            }
            check_and_fclose(fp, COMMAND_FILE);
        }
    }
    // 把启动参数也放到 boot 对象中
    strlcpy(boot.command, "boot-recovery", sizeof(boot.command));
    strlcpy(boot.recovery, "recovery\n", sizeof(boot.recovery));
    int i;
    for (i = 1; i < *argc; ++i) {
        strlcat(boot.recovery, (*argv)[i], sizeof(boot.recovery));
        strlcat(boot.recovery, "\n", sizeof(boot.recovery));
    }
```

```
    set_bootloader_message(&boot);
}
```

get_args()函数的主要作用是建立 recovery 的启动参数，如果系统启动 recovery 时已经传递了启动参数，那么这个函数只是把启动参数的内容复制到函数的参数 boot 对象中，否则函数会首先从/misc 分区中获取命令字符串来构建启动参数。如果/misc 分区下没有内容，则尝试打开/cache/recovery/command 文件并读取文件的内容来建立启动参数。从这个函数我们可以看到，更新系统最简单的方式是把更新命令写到/cache/recovery/command 文件中。get_args()函数是通过 get_bootloader_message()函数来读取/misc 分区的数据的，get_bootloader_message()函数的代码如下所示：

```
int get_bootloader_message(struct bootloader_message *out) {
    Volume* v = volume_for_path("/misc");    // 打开 misc 分区
    if (v == NULL) {
      return -1;
    }
    if (strcmp(v->fs_type, "mtd") == 0) { // 如果文件系统类型是 mtd
        return get_bootloader_message_mtd(out, v);
    } else if (strcmp(v->fs_type, "emmc") == 0) {  // 如果是 emmc 类型
        return get_bootloader_message_block(out, v);
    }
    return -1;
}
```

从 get_bootloader_message()函数的代码可以看到，它打开/misc 分区来读取数据，函数中还区分了分区的类型是 mtd 还是 emmc 来选择不同的函数去读取数据，这些函数只是调用底层函数去读取数据块的内容，我们就不分析了。

get_args()函数的结尾调用了 set_bootloader_message()函数，它的代码如下：

```
int set_bootloader_message(const struct bootloader_message *in) {
    Volume* v = volume_for_path("/misc");
    if (v == NULL) {
      LOGE("Cannot load volume /misc!\n");
      return -1;
    }
    if (strcmp(v->fs_type, "mtd") == 0) {
        return set_bootloader_message_mtd(in, v);
    } else if (strcmp(v->fs_type, "emmc") == 0) {
        return set_bootloader_message_block(in, v);
    }
    return -1;
}
```

set_bootloader_message()函数的作用是把启动参数的信息又保存到了/misc 分区中。这样做的目的是防止升级过程中发生崩溃，这样重启后仍然可以从/misc 分区中读取更新的命令，继续进行更新操作。这也是为什么 get_args()函数要从几个地方读取启动参数的原因。

如果 recovery 正常退出，将调用 finish_recovery()函数，这个函数将清除/misc 分区的内容，这样就不会重复更新系统了。如下所示：

```
static void finish_recovery(const char *send_intent) {
    ......
    struct bootloader_message boot;
    memset(&boot, 0, sizeof(boot));
    set_bootloader_message(&boot);  // 向/misc 分区中写入 0
    ......
}
```

23.1.3 执行菜单命令

下面我们看看 Recovery 是如何执行菜单命令的。前面介绍了，prompt_and_wait()函数用来打印屏幕菜单并接收用户输入，函数代码如下：

```
static void prompt_and_wait(Device* device, int status) {
    const char* const* headers = prepend_title(device->GetMenuHeaders());
    for (;;) {
        finish_recovery(NULL);
        // 根据命令的执行情况改变 UI 的背景
        switch (status) {
            case INSTALL_SUCCESS:
            case INSTALL_NONE:
                ui->SetBackground(RecoveryUI::NO_COMMAND);
                break;
            case INSTALL_ERROR:
            case INSTALL_CORRUPT:
                ui->SetBackground(RecoveryUI::ERROR);
                break;
        }
        ui->SetProgressType(RecoveryUI::EMPTY);
        // 系统将在 get_menu_selection 函数中等待用户的输入
        int chosen_item = get_menu_selection(headers, device->GetMenuItems(),
                          0, 0, device);
        // 把用户的选择先交给 Device 对象处理
        chosen_item = device->InvokeMenuItem(chosen_item);
        int wipe_cache;
        // 处理菜单命令
        switch (chosen_item) {
            case Device::REBOOT:                    // 退出 Recovry
                return;
            case Device::WIPE_DATA:                 // 擦除数据区
                wipe_data(ui->IsTextVisible(), device);
                if (!ui->IsTextVisible()) return;
                break;
            case Device::WIPE_CACHE:                // 擦除 cache 分区
                ui->Print("\n-- Wiping cache...\n");
                erase_volume("/cache");
                ui->Print("Cache wipe complete.\n");
                if (!ui->IsTextVisible()) return;
                break;
            case Device::APPLY_EXT:                 // 从 SDCard 上更新
                status = update_directory(SDCARD_ROOT, SDCARD_ROOT, &wipe_cache,
                                          device);
                if (status == INSTALL_SUCCESS && wipe_cache) {
                    ui->Print("\n-- Wiping cache (at package request)...\n");
                    if (erase_volume("/cache")) {
                        ui->Print("Cache wipe failed.\n");
                    } else {
                        ui->Print("Cache wipe complete.\n");
                    }
                }
                ......
                break;
            case Device::APPLY_CACHE:               // 从 Cache 中更新
                status = update_directory(CACHE_ROOT, NULL, &wipe_cache, device);
                if (status == INSTALL_SUCCESS && wipe_cache) {
                    ui->Print("\n-- Wiping cache (at package request)...\n");
                    if (erase_volume("/cache")) {
                        ui->Print("Cache wipe failed.\n");
                    } else {
                        ui->Print("Cache wipe complete.\n");
                    }
                }
                ......
                break;
            case Device::APPLY_ADB_SIDELOAD:        // 启动 ADBD
                status = apply_from_adb(ui, &wipe_cache, TEMPORARY_INSTALL_FILE);
                ......
                break;
        }
```

```
        }
    }
```

prompt_and_wait()函数的逻辑是，在屏幕上打印出菜单，然后等待用户按键输入。用户只能使用上下音量键来移动菜单的选择条，然后使用 Power 键进行选择。得到用户的输入后，接下来就是根据输入的命令进行处理。函数中处理的命令如下。

- REBOOT 命令：退出 Recovery 就会重新启动。
- WIPE_DATA 命令：这条命令其实就是“恢复出厂设置”命令所进行的操作，它将清除手机上所有用户数据，也包括 cache 分区下的内容。
- WIPE_CACHE 命令：WIPE_CACHE 命令只会擦除 Cache 分区下的内容。
- APPLY_EXT 命令：这条命令可以让用户通过 UI 从 SDCard 上挑选一个文件进行系统更新操作。
- APPLY_CACHE 命令：APPLY_CACHE 命令和 APPLY_EXT 命令类似，只不过是从/cache 目录下挑选文件来执行更新。
- APPLY_ADB_SIDELOAD：这条命令将启动 ADBD，不过这个 ADBD 只是一个很简单的版本，主要的作用是让用户能通过 adb 连接来执行 sideload 命令上传，更新文件到/tmp/update.zip，然后再执行更新操作。

深入解析 Android 5.0系统

专家推荐：

这本书介绍Android系统的翔实和认真程度上可能在市面上无出其右者。从JNI/Bionic到Loop/Init，从SystemServer到Provider，从包管理到图形系统，从窗口系统到输入管理，从电源管理到唤醒机制，从网络管理到音视频系统，甚至从Vold到Recovery，从虚拟机到自动化测试，本书都有详细解释和说明。作为一个工作十多年的资深工程师，作为一个从Android 1.0版本开始接触Android系统的工程师，作为一个设计过多款Android手机系统的一线架构师，我想没有这样的经历是很难将这本书写得如此详尽。希望读者可以从中得到有益的启发，开启自己完美的Android开发之旅！

——小米电视系统软件部总监，茹忆

这是一本有6年安卓系统开发经验的、中国顶级Android系统工程师的心血之作！这是一本可以推荐给任何从事Android系统开发或应用开发工程师看的书！

——原Motorola软件总监，播思通讯CTO，饶宏

一本非常优秀的、介绍Android内部机制的书，详细地分析了Android系统的大部分模块，值得每一位希望深入学习Android系统的工程师拥有。

——佛倍无线软件部经理，陈行星

本书支持社区

极客学院 kaoyuan.com

随书赠送极客学院提供的 VIP 激活码（5159143894），VIP 激活码激活后，可在极客学院官网享受 2 个月 VIP 服务，畅快学习所有在线课程。极客学院网址为 www.jikexueyuan.com，任何疑问可联系客服小雷（QQ：2905494756）

封面设计：董志桐

分类建议：计算机/ 程序设计/ 移动开发

人民邮电出版社网址：www.ptpress.com.cn