위키북스

2011年度
韩国文化体育观光部
优秀学术图书

Android框架揭秘

Inside the Android Framework

[韩] 金泰延 宋亨周 朴知勋 李白 林起永 著　　武传海 译

人民邮电出版社
POSTS & TELECOM PRESS

更多更新编程资源尽在 http://www.pin5i.com/

위키북스

Android框架揭秘

Inside the Android Framework

[韩] 金泰延 宋亨周 朴知勋 李白 林起永 著　　武传海 译

人 民 邮 电 出 版 社
北 京

更多更新编程资源尽在 http://www.pin5i.com/

·内容提要·

本书通过对 Android 系统源代码的分析，主要介绍 Android 框架的初始化过程及主要组件的工作原理。作者直接分析和整理了 Android 框架的主要源代码，并详细讲解了理解框架工作原理所需的各种基础知识和构成实际 Android 平台骨干的服务框架。其中的主要内容包括：通过启动程序了解 Android 框架的概要，移植 Android 以及如何开发适合各种机器的应用程序，分析 Android 框架所需的基础知识，JNI（Java Native Interface）与 Binder 基础知识，Zygote、Service Manager、Service Server 等 Android 核心组件，Android 服务框架的结构与理解，通过 Camera Service、Activity Manager Service 等服务分析实际的 Android 服务等。

本书不仅可以供具备一定开发经验的 Android 开发人员参考阅读，也可作为 Android 开发初学者的 Android 框架入门教材使用。

更多更新编程资源尽在 http://www.pin5i.com/

·作者序·

宋亨周

笔者第一次接触 Android 是在 2009 年 5 月，当时苹果的 iPhone 获得了巨大成功，作为唯一能与之抗衡的操作系统，Android 引起了众多研发者的关注。一些出版社也相继推出了基于 Android SDK 应用程序的图书。跟随这股潮流笔者开始学习编写 Android 应用程序。

也许此前一直从事系统编程工作的缘故，笔者在学习编写 Android 应用程序的过程中，慢慢对 Android 系统的内部运行机制产生了兴趣。Android 系统是开放源码的，笔者想深入研究 Android 系统的内部运行机理，但无论在网络上还是书店中相关资料都很少，笔者在研究过程中处处碰壁，逐渐认识到单枪匹马、孤军作战不是良策，很难取得实质性进展。

于是，笔者成立了“水原 Android 平台研究小组”，和一群志同道合的朋友开始了对 Android 平台的探索。尽管大家的工作领域不尽相同，但是在共同兴趣爱好的驱使下，每周六我们都会聚集在一起，分析研究 Android 系统，分享各自的研究心得并整理出相关的研究资料。

本书正是在这些研究资料的基础上编写而成的，它凝聚了小组所有人的心血，是集体智慧的结晶。虽然本书并未涵盖 Android 的所有内容，但相信对初学 Android 系统的朋友会有相当大的帮助。记得有个诗句：“不要在雪地中盲目前行，请循着我的脚印，它将为你引路。”希望本书能成为读者的引路人。

本书是 Android 研究小组所有成员共同努力的结果，借此机会向各位表示深深的谢意。最后，还要感谢我的妻子贤静、儿子承民，一直以来默默地关心、理解、支持我，在此将本书一同献给你们，感谢你们无私的付出。

金泰延

从诞生到现在，Android 系统不断升级，每次升级都会增添许多新的功能与特性，开发人员很难适应这种变化，并且 Android 系统糅合了各种先进技术，要完全理解它很难。所以，学习 Android 系统编程时，不仅要熟悉 Linux、C++、Java 等各种技术，还要理解 Android 系统平台，研究它的组成结构及内部运行机制。

更多更新编程资源尽在 http://www.pin5i.com/

学习 Android 平台有什么好方法吗？我认为，可以借鉴我高中时学习数学的方法，首先将相关的公式和基本原理弄清楚，然后再去做应用题目，所有问题将迎刃而解。同理，学习 Android 应用程序开发，首先要理解其系统内部结构及核心原理。本书对 Android 平台中的 init 进程、Binder、JNI、Zygote、服务框架、核心系统服务等运行原理与机制进行了剖析与说明。

在开始编写本书时，Android 的版本为 1.6，到本书完稿时，已更新到 2.2。但是一般而言，核心原理不会轻易改变，而且本书侧重基础原理的讲解，受版本更新影响不大，请放心阅读。对于编程人员而言，本书是学习 Android 平台最好的起点。

在 Android 平台相关资料匮乏的前提下，本书内容完全基于对源代码的分析编写而成。源码分析过程耗时长达 6 个月，本书是研究小组所有成员智慧的结晶，正是他们无私的奉献与付出，本书才最终得以出版。在此，对研究小组的所有成员表示最诚挚的谢意。

在此特别要感谢宋亨周，Android 平台研究小组的组织者，亦是本书的执笔者，他付出了大量劳动，并给我们提供了参与的相会。同时，还要感谢参与本书编写的李白、林起永、朴知勋，感谢他们长期以来的努力。本书凝聚了全体成员的心血，大家辛苦了！

最后，还要感谢我的朋友与家人。首先，感谢蔡兴锡教授 4 年来对我孜孜不倦的教诲与爱护；其次是燕赞，感谢他把我介绍到 Android 研究小组，还有 Optimus Q 开发小组的曹尚贤、薛美英，李东勋，以及 301 同仁们，感谢他们的鼓励与支持。此外，还要感谢研究室的朋友如善烈、郑浩、显宰、燕智等，感谢他们试读本书并提出了中肯的意见。最后，感谢我的家人，感谢父母的关心与爱护，感谢妻子珍淑的理解与支持，使我能够在蜜月旅行中编写稿子。

朴知勋

加入水原 Android 平台研究小组，研究 Android 系统对我来说是一次全新的挑战。之前我从事 SoC（System On Chip）工作，主要跟硬件开发打交道，对我而言，Android 融合的各种技术让我着迷，大大激发了我的好奇心与求知欲望。在分析 Android 系统内核的过程中，我学到了许多有关 Linux 深层次的知识，认识到了遍布整个框架的 Java 富于弹性的特征，还有了解了连接各种核心库的 JNI，以及进程孵化器 Zygote 等各种组成要素，这是一个充满乐趣的过程。而且，我们可以任意查看源码，这是 Android 的开源性带来的好处，也正是其魅力所在。

本书在 Android 源码分析的基础上编写而成。如果将分析源码比作一次 Android 框架探险，那么本书就是一本探险日志。那些在大学讲堂里开着投影仪，一点点分析源码的日子，至今仍历历在目，苦思冥想后豁然开朗的喜悦仍然记忆犹新。我们将这些过

更多更新编程资源尽在 http://www.pin5i.com/

程整理成书，希望借助本书跟大家分享知识、共同进步。

尽管没能做到面面俱到，我们还是尽可能地把各种重要的内容、细节纳入书中。从最终的成稿来看，我们做得相当不错。这得益于大家的大力支持，在此特别感谢林明柏，站在读者的立场上审校原稿，提出了很多宝贵意见；还要感谢高炫哲先生，在百忙之中抽出时间对 Binder IPC 原理部分进行审查，他的专业性意见是本书质量的重要保障。在此，感谢你们付出的每份努力，谢谢！

李白

从加入 Android 平台研究小组到现在，不知不觉已有一年多了。期间，很多其他兴趣小组解散了，但我们一直坚持了下来，并且将研究成果整理成书，这是小组所有成员共努力的结果。回首过去，小组成员在一起共同学习的情景仍然历历在目，那种攻克难题后的喜悦仍然在脑海中萦绕，这些都将成为我永不磨灭的美好回忆。借此机会，向小组的各位成员表示感谢，谢谢你们每个人的付出。

最后，感谢我的家人。由于参加研究小组与编写本书而没有更多的时间陪伴你们，但你们毫无怨言，一直默默地关心我、支持我，谢谢你们！妻子朴美拉、儿子俊民，我爱你们！

林起永

当谷歌发布 Android OS 时，我非常兴奋。在 20 年前的个人电脑市场上，Apple、Amiga、IBM、MSX 等公司竞争非常激烈，后来 IBM 发布了开放式结构电脑，随之兼容 IBM 的个人电脑迅速占领了市场。

嵌入式设备的发展也经历了类似的时期与阶段。各大手机厂商开发生产的操作系统与手机硬件都是封闭的，并且所生产的手机通常只能用于通话，功能比较单一。就这样，这些厂商在各自相对封闭的领域内独占着市场并慢慢发展着。但是随着时代的发展和技术的进步，人们提出的要求越来越高，手机从简单的通话工具逐步发展成为掌上多媒体设备，于是智能手机出现了。苹果公司推出的智能手机备受消费者喜爱与推崇，市场占有率越来越大。

但是，包括苹果在内的许多智能手机厂商所开发的操作系统，相对而言仍然处于封闭状态，直到谷歌发布开源的 Android 系统，这种相对封闭的状态才被打破。大批手机厂商开始关注 Android，并使用这种系统开发手机。渐渐地，搭载 Android 系统的手机受到越来越多消费者的青睐，市场份额不断攀升。相信在不远的将来，谷歌 Android 会像 IBM 的兼容 PC 一样颠覆整个行业，究竟对我们的生活会产生怎样的影响，让我们翘首以待。

然而，在谷歌 Android 兴起并不断发展之时，有关 Android 平台研究的资料并不多见。市面上所见到的大部分图书，多是关于如何开发 Android 应用程序的，而对于其

更多更新编程资源尽在 http://www.pin5i.com/

系统框架内部的运行机制，平台核心元素分析的资料、图书仍然非常稀少。

Android 是个非常庞大、复杂的系统，仅凭个人的力量，想完全了解掌握它几乎是不可能的。于是在 2008 年，一个特别炎热的夏天，我们几个对 Android 系统感兴趣的人聚集在一起，结成了 Android 平台研究小组。我们利用业余时间聚在一起，一同分析研究 Android 系统，相互交流心得与体会，慢慢地，我们对 Android 系统了解得越来越多。为了将研究成果分享给大家，我们编写了本书，这是我们小组所有成员勤奋努力、集体智慧的结晶。Android 犹如一座巨型山脉，任何一本书想要将其完全包含进去都是不可能的，本书也不例外，但书中包含了 Android 系统核心部分的重要内容，相信对初学 Android 的朋友会有一定帮助，这也是编写此书的初衷。

本书由多名作者集体写作而成，他们分别是宋亨周、朴知勋、金泰延、李白，他们的热心、真诚令人印象深刻，我非常喜欢跟他们在一起学习、分享知识。在编写本书的过程中，我们相互鼓励，相互配合，最终写成此书，在此对他们表示最诚挚的谢意。

最后，感谢我的妻子惠珠，谢谢她一直以来对我的关心、照顾与支持，在此将本书献给我亲爱的惠珠。惠珠，感谢有你，我永远爱你！

更多更新编程资源尽在 http://www.pin5i.com/

·前言·

在移动开发中，对开发者而言，哪些能力是必须具备的呢？

不同的人可能有不同的答案，但不管怎样，理解掌握目标设备中的系统框架是必不可少的。如果你是一名程序开发人员，那么必须要学习 Windows 或 Linux 等操作系统的运行原理；如果你是一名网站开发人员，那么你必须要学习浏览器的运行机制。同样，如果你想成为移动开发领域中专家级的开发人员，那么你必须要学习相关系统框架的工作原理。

换言之，在移动开发领域中，研究学习相关系统框架的内部运行机制原理是成为高级开发人员的必由之路。这条规则也适用于 Android 开发人员。但是，与 Android 框架运行原理相关的资料，无论在网络上，还是书店中都很难找到。事实上，除了谷歌或谷歌 IO（谷歌开发者大会）提供的有限的资料之外，研究 Android 框架的资料和图书非常少。

目前，研究学习 Android 框架运行原理的最基本、最原始的做法就是直接分析 Android 系统源代码。由于有关 Android 框架分析的资料非常少，笔者们也是通过分析系统源码来研究 Android 框架的。但是面对庞大的 Android 系统源码，仍然不知从何处入手分析。

幸运的是，在谷歌 IO 2008 Android 开发者大会上发布了一份名为“Anatomy & Physiology of an Android”（http://sites.google.com/site/io/anatomy--physiology-of-an-android）的资料，资料中包含了 Android 框架初始化过程等内容，为我们分析 Android 框架指明了方向，提供了便利。

开始写作本书时，Android 最新版本为 1.5（代号：Cupcake），至本书完稿时，Android 更新到 2.2 版本（代号：Froyo）。所以我们根据 Android 2.2 对全书内容重新做了调整，以使本书内容与 Android 版本保持一致。事实上，从 Android 1.5 版本到 Android 2.2 版本，有关系统框架的源码并未发生重大变化，因此本书的内容受 Android 版本升级的影响不大。也许你阅读本书时，Android 又升级到新的版本，请不要担心，本书讲解的都是最基本、最核心的内容，对 Android 版本更新不敏感，请放心阅读。

本书以讲解分析 Android 框架内部运行原理为主，涵盖了 Android 框架的核心和

更多更新编程资源尽在 http://www.pin5i.com/

·目录·

第 1 章　Android Framework 概要 …… 1

1.1　Android 源代码组成 …… 2
1.2　通过启动过程分析 Android Framework …… 3

第 2 章　搭建 Android 开发环境 …… 7

2.1　主机环境构成 …… 7
2.1.1　安装 VirtualBox …… 7
2.1.2　安装 Ubuntu …… 8
2.2　搭建 Android 平台编译环境 …… 9
2.2.1　编译工具 …… 10
2.2.2　安装 Repo …… 11
2.2.3　下载 Android 源代码 …… 11
2.2.4　编译 Android 源代码 …… 12
2.3　搭建 Android SDK 开发环境 …… 13
2.3.1　下载、安装 Eclipse …… 13
2.3.2　下载 Android SDK starter …… 13
2.3.3　安装 ADT 插件 …… 14
2.3.4　设置 Android SDK 路径 …… 16
2.3.5　安装 Android SDK …… 16
2.4　开发 Android 应用程序 …… 18
2.5　应用程序 Framework 源码级别调试 …… 21
2.5.1　加载应用程序 Framework 源 …… 21
2.5.2　调试 HelloWorld Framework（源码级） …… 24
2.6　小结 …… 27

更多更新编程资源尽在 http://www.pin5i.com/

第 3 章　init 进程 …… 29

3.1　init 进程运行过程 …… 29
3.2　init 进程源码分析 …… 31
3.3　init.rc 脚本文件分析与执行 …… 40
3.3.1　动作列表（Action List） …… 41
3.3.2　服务列表（Service List） …… 43
3.3.3　init.rc 文件分析函数 …… 44
3.3.4　动作列表与服务列表的运行 …… 48
3.4　创建设备节点文件 …… 52
3.4.1　创建静态设备节点 …… 52
3.4.2　动态设备感知 …… 57
3.5　进程的终止与再启动 …… 58
3.6　属性服务 …… 62
3.6.1　属性初始化 …… 63
3.6.2　属性变更请求处理 …… 65
3.7　小结 …… 67

第 4 章　JNI 与 NDK …… 69

4.1　Android 与 JNI …… 69
4.2　JNI 的基本原理 …… 72
4.2.1　在 Java 中调用 C 库函数 …… 72
4.2.2　小结 …… 83
4.3　调用 JNI 函数 …… 84
4.3.1　调用 JNI 函数的示例程序结构 …… 84
4.3.2　Java 层代码（JniFuncMain.java） …… 85
4.3.3　分析 JNI 本地函数代码 …… 87
4.3.4　编译及运行结果 …… 101
4.3.5　在 Android 中的应用举例 …… 102
4.4　在 C 程序中运行 Java 类 …… 102
4.4.1　Invocation API 应用示例 …… 103
4.4.2　编译及运行 …… 108
4.4.3　Invocation API 在 Android 中的应用举例：Zygote 进程 …… 110
4.5　直接注册 JNI 本地函数 …… 110
4.5.1　加载本地库时，注册 JNI 本地函数 …… 111

更多更新编程资源尽在 http://www.pin5i.com/

4.5.2 Android 中的应用举例 …… 115
4.6 使用 Android NDK 开发 …… 122
4.6.1 安装 Android NDK …… 123
4.6.2 使用 Android NDK 开发步骤 …… 127
4.6.3 小结 …… 136

第 5 章 Zygote …… 137

5.1 Zygote 是什么 …… 137
5.2 由 app_process 运行 ZygoteInit class …… 142
5.2.1 生成 AppRuntime 对象 …… 143
5.2.2 调用 AppRuntime 对象 …… 144
5.2.3 创建 Dalvik 虚拟机 …… 145
5.2.4 运行 ZygoteInit 类 …… 146
5.3 ZygoteInit 类的功能 …… 147
5.3.1 绑定/dev/socket/zygote 套接字 …… 149
5.3.2 加载应用程序 Framework 中的类与平台资源 …… 150
5.3.3 运行 SystemServer …… 155
5.3.4 运行新 Android 应用程序 …… 158

第 6 章 Android 服务概要 …… 163

6.1 示例程序：理解 Android 服务的运行 …… 163
6.2 Android 服务的种类 …… 166
6.3 Android 应用程序服务 …… 168
6.4 Android 系统服务 …… 182
6.5 运行系统服务 …… 185
6.5.1 分析媒体服务器（Media Server）的运行代码 …… 186
6.5.2 分析系统服务器(System Server)的运行代码 …… 188
6.6 Android Service Framework、Binder Driver 概要及相关术语 …… 192

第 7 章 Android Binder IPC …… 197

7.1 Linux 内存空间与 Binder Driver …… 197
7.2 Android Binder Model …… 199

更多更新编程资源尽在 http://www.pin5i.com/

7.2.1 Binder IPC 数据传递 …… 201
7.2.2 Binder IPC 数据流 …… 202
7.2.3 Binder 协议（Binder Protocol） …… 204
7.2.4 RPC 代码与 RPC 数据 …… 206
7.2.5 Binder 寻址（Binder Addressing） …… 206
7.3 Android Binder Driver 分析 …… 209
7.3.1 从进程的角度看服务的使用 …… 210
7.3.2 从 Binder Driver 角度看服务的使用 …… 214
7.3.3 Binder Driver 函数分析 …… 219
7.4 Context Manager …… 251
7.5 小结 …… 256

第 8 章 Android Service Framework …… 257

8.1 服务框架（Service Framework） …… 257
8.2 服务框架（Service Framework）的构成 …… 259
8.2.1 各层构成元素的配置 …… 260
8.2.2 各层构成元素间的相互作用 …… 261
8.2.3 类的结构 …… 264
8.3 运行机制 …… 266
8.3.1 服务接口 …… 267
8.3.2 服务 …… 273
8.3.3 服务代理（Service Proxy） …… 276
8.3.4 Binder IPC 处理 …… 280
8.4 本地服务管理器（Native Service Manager） …… 282
8.4.1 Service Manager 概要 …… 282
8.4.2 Service Manager 类 …… 284
8.4.3 Service Manager 的运行 …… 286
8.5 编写本地服务 …… 314
8.5.1 设计 HelloWorld 系统服务 …… 314
8.5.2 HelloWorld 服务接口 …… 315
8.5.3 HelloWorld 服务 …… 316
8.5.4 HelloWorld 服务代理 …… 319
8.5.5 运行 HelloWorld 服务 …… 320
8.6 小结 …… 325

更多更新编程资源尽在 http://www.pin5i.com/

第 9 章　本地系统服务（Native System Service）分析 ……327

9.1　相机服务（Cameral Service） ……327
9.2　相机应用程序 ……328
9.3　相机服务框架（Camera Service Framework） ……331
9.3.1　相机服务框架层次结构 ……331
9.3.2　相机服务框架类 ……333
9.4　相机服务框架的运行 ……334
9.4.1　初始化相机服务 ……334
9.4.2　连接相机服务 ……335
9.4.3　相机服务连接过程分析 ……337
9.4.4　相机设置与控制 ……340
9.4.5　相机设置与控制分析 ……341
9.4.6　相机事件处理 ……342
9.4.7　相机事件处理分析 ……343
9.5　小结 ……345

第 10 章　Java 服务框架（Java Service Framework） ……347

10.1　Java 服务框架（Java Service Framework） ……347
10.1.1　Java 服务框架的层次结构 ……348
10.1.2　Java 服务框架中各个类间的相互作用 ……351
10.2　运行机制 ……354
10.2.1　Java 服务框架初始化 ……355
10.2.2　Binder ……355
10.2.3　BinderProxy ……361
10.2.4　Parcel ……364
10.3　Java 系统服务的实现 ……367
10.3.1　闹钟服务（Alarm Manager Service）分析 ……368
10.3.2　编写 HelloWorldService 系统服务 ……372
10.3.3　使用 HelloWorldService 系统服务 ……375
10.3.4　编译 HelloWorldService 系统服务 ……378
10.4　Java Service Manager ……380
10.4.1　Java Service Manager 简介 ……380
10.4.2　BinderInternal ……381
10.4.3　Java Service Manager 的运行实例 ……383

更多更新编程资源尽在 http://www.pin5i.com/

10.5 使用 AIDL 生成服务代理与服务 Stub …… 389
10.5.1 在 AIDL 文件中定义服务接口 …… 390
10.5.2 使用 AIDL 编译器，生成服务接口、服务 Stub 以及服务代理 …… 391
10.5.3 继承 Stub 类创建服务 …… 392
10.5.4 服务接口的调用 …… 393
10.6 小结 …… 394

第 11 章 Java 系统服务运行分析 …… 395

11.1 Activity Manager Service …… 395
11.2 Activity Manager Service 创建服务分析 …… 397
11.2.1 Controller Activity-调用 startService()方法 …… 398
11.2.2 Activity Manager Service 的 startService()方法的调用过程（使用 Binder RPC） …… 399
11.2.3 Activity Manager Service——运行 startService()Stub 方法 …… 405
11.2.4 运行 ActivityThread 类的 main()方法 …… 409
11.2.5 Activity Manager Service——attachApplication() Stub 方法 …… 414
11.3 小结 …… 421

附录 AIDL 语法 …… 423

更多更新编程资源尽在 http://www.pin5i.com/

第 1 章

Android Framework 概要

安卓（Android）是一个移动终端操作系统平台，由谷歌在 2007 年 11 月 5 日发布，它采用软件堆层（software stack，又名软件叠层）的架构，主要由操作系统、中间件、核心应用程序组成。安卓提供了一整套的软件框架，方便开发者开发基于移动终端的各种应用程序。关于安卓更详细的介绍，在安卓开发者网站（http://developer.android.com）中，可以查找到相关资料。

如果你有过开发 Android 应用程序的经验，即使你对 Android Framework 掌握得不深，通过谷歌提供的 Android SDK，也能非常容易地开发出基于 Android 的交互应用程序。这得益于 Android 为开发者提供了一套定义良好的软件框架，开发者即使不具备特别高深的专业知识，在短时间内同样能开发出强大的 Android 应用程序。

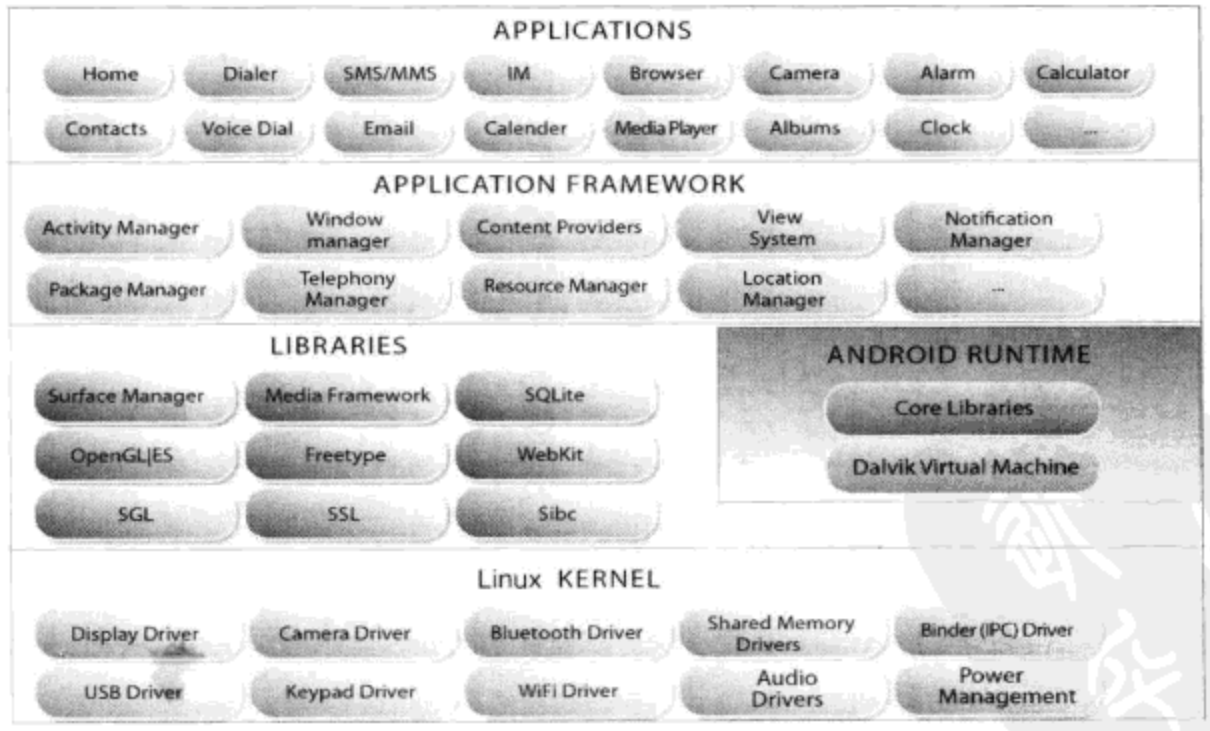


图 1-1 | Android 系统架构

图 1-1 是 Android 系统架构图，如图所示，Android 采用了分层架构，分为 4 个层，从高层到低层分别是应用程序层、应用程序框架层、系统运行库层和 Linux 核心层。

更多更新编程资源尽在 http://www.pin5i.com/

观察 Android 系统架构图可以发现，其提供的基本应用程序（Home，Camera，Dialer，Browser 等）运行在应用程序框架层之上。同样，开发者借助 Android SDK，调用应用程序框架 API 开发出的应用也运行在应用程序框架层之上。

既然 Android 框架如此重要，那是否意味着开发者要全面、系统地掌握它呢？

关于这点，前面已略有提及。对于一个开发者而言，即使没有完全掌握 Android 框架，同样能够编写出一些不错的应用程序。当然，这并不是鼓励大家不去学习它。事实上，如果你对该框架了解得越多、越熟悉，掌握了相关原理，你会开发出更适合于 Android 框架，更优化的应用程序。在本书中，不会讨论 Android 自带的 AlarmClock、Contacts 等程序，但这些程序是经过严格测试的、高质量的程序，研读这些程序的源代码，对于开发者编写优秀的应用程序非常有益。

当然，如果你想成为一名优秀的 Android 平台应用程序开发者，那么你有必要深入了解、学习 Android 框架。Android 是一个真正开放的移动开发平台，访问其网站即可轻松地获取其源代码。每个硬件厂商可以根据自身需要定制基本的 Android 框架，开发出与竞争对手不同的产品。图 1-2 中显示的是各硬件厂商分别定制不同的 Android UI（安卓用户界面），从左往右依次为 Android 模拟器基本 UI、HTC UI、三星 UI、索尼爱立信 UI。

Android 模拟器基本 UI　HTC Sense UI　Samsung Touchwiz UI　Sony Ericsson Rachel UI

图 1-2 | Android 模拟器基本 UI 及各硬件厂商定制的 Android UI

像这样，若想构建基于 Android 框架的个性化移动终端系统，则必须对 Android 框架进行系统、全面地学习与研究。

1.1 Android 源代码组成

那么，我们该如何学习 Android 框架呢？其实，学习 Android 框架时，最准确、最

更多更新编程资源尽在 http://www.pin5i.com/

权威的参考资料就是 Android 的源代码，这些源代码从 http://android.git.kernel.org/可免费下载。关于这方面的内容，请参考第 2 章中的相关部分。

Android 主要源代码组成如下。

- Kernel：Android Linux 内核 2.6
- bionic：Android 标准 C 运行支持库
- bootloader：Android 内核加载器参考
- build：Android 的 Bulid 系统
- cts：Android 兼容性测试源
- dalvik：Dalvik 虚拟机
- external：Android 使用的开放源
- frameworks：Android 框架
- hardware：Android HAL（Hardware Abstraction Layer，硬件抽象层）库源
- packages：包含 Android 基本应用，Content Provider 等
- system：Android 初始化进程、蓝牙工具集等

本书主要针对 Kernel、frameworks、packages、system 文件夹内的源代码进行分析。如果你想进一步深入分析 Android 框架，建议你优先分析 frameworks 中的其他源代码（本书在讲 Android 框架源时，相关代码路径会以脚注的形式标示出来）。

1.2 通过启动过程分析 Android Framework

Android 源码数量极其庞大，以 Android 2.2 为例，除去 Linux 代码，代码数量大于 4GB。若想理解和掌握这么庞大的 Android 系统，需要耗费大量的时间，付出极大的努力。并且，到现在为止，也没有相关资料对 Android Frame 作系统完整的讲解说明。

那么，分析 Android Framework 用什么方法好呢？回答这一问题之前，先回想一下我们是如何分析他人编写的程序代码的。在分析程序代码时，我们通常从程序的入口 main()函数开始，一点点地理清程序流，把握程序的运行过程。同样，在分析结构庞大的 Android Framework 时，也要从 Android 平台启动过程着手。Android 启动过程包含从 Linux 内核加载到 Home 应用程序启动的整个过程，依次分析这一过程，有利于我们系统地理解 Android Framework 运行的原理。

如图 1-3 所示，简单地描述了 Android 启动过程，本书在后续章节中讲解 Android Framework 初始化过程、各模块间如何相互作用等内容时，均以此图所描述的启动过程为基础。

更多更新编程资源尽在 http://www.pin5i.com/

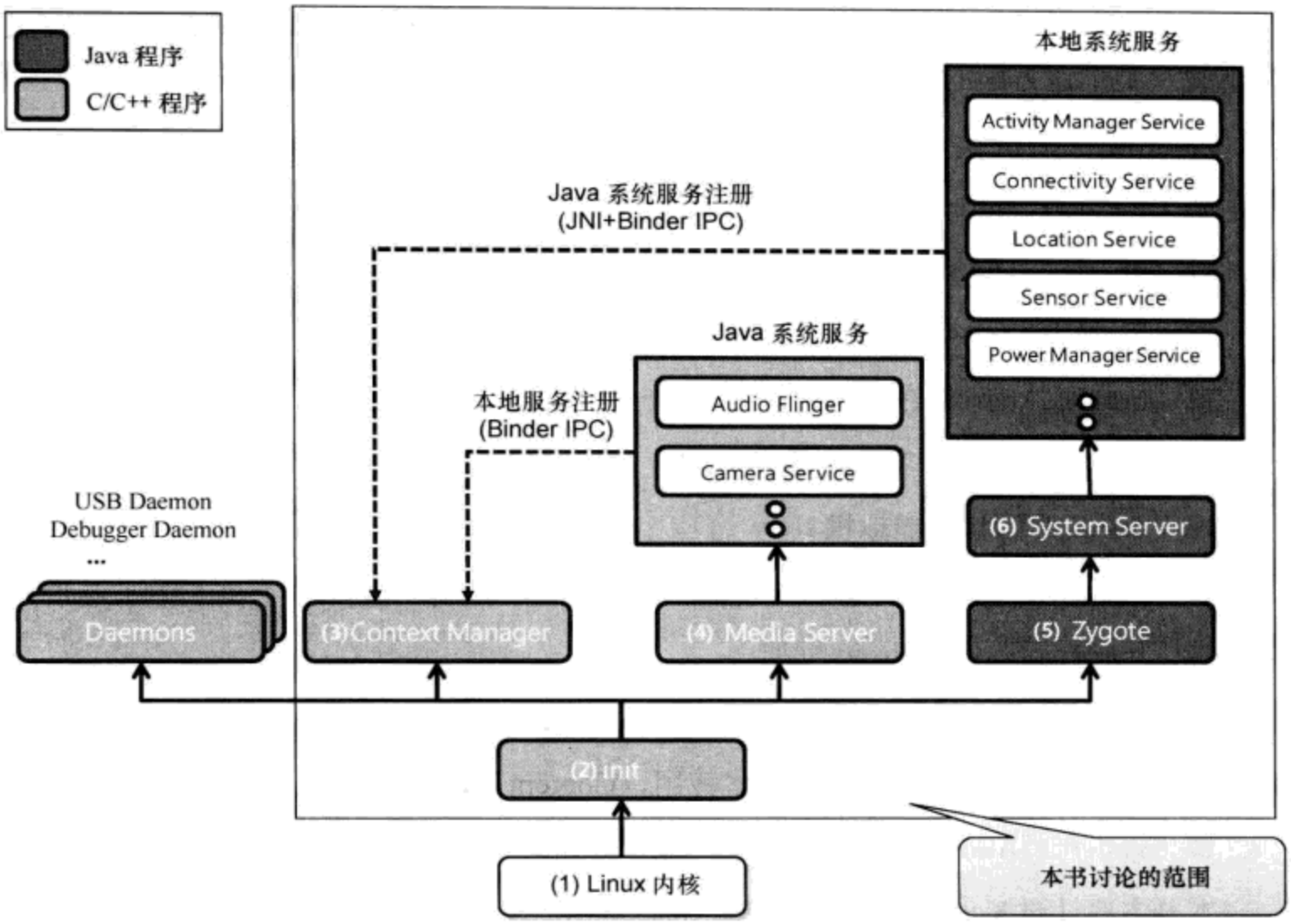


图 1-3 | Android 启动过程

对照图 1-3 Android 启动过程图，简单地讲解一下。

1．Linux 内核

Android 是基于 Linux 内核的系统平台。启动时，首先通过 bootloader（系统加载器），加载 Linux 内核。在 Linux 加载启动时，与普通的 Linux 启动过程相同，先初始化内核，然后调用 init 进程。

2．init

Android init 进程对各种设备进行初始化，运行 Android Framework 所需用的各种 Daemon、Context Manager、Media Server、Zygote 等。

以下是 init 进程执行的 Daemon 进程。

- USB Daemon (usbd)：管理 USB 连接。
- Android Debug Bridge Daemon (adbd)：Android Debug Bridge 连接管理。
- Debugger Daemon (debuggerd)：启动 Debugger 系统。
- Radio Interface Layer Daemon (rild)：管理无线通信连接。

更多更新编程资源尽在 http://www.pin5i.com/

3．Context Manager

Context Manager 是一个管理 Android 系统服务的重要进程。系统服务是组成 Android Framework 的重要组件，提供从相机、音频、视频处理到各种应用程序制作所需要的重要的 API。

Context Manager 提供运行于 Android 内的各种系统服务信息。应用程序或 Framework 内部模块在调用系统服务时，需要先向服务管理器申请，而后通过 Binder IPC（Interprocess communication）调用系统服务。

在系统启动时，Android 所有系统服务都要把各自的 handle 信息注册到 Context Manager，此时，Binder IPC 用来进行进程间的通信。

4．Media Server

Media Server 用于运行基于 C/C++ 的本地系统服务，如 Audio Flinger（负责音频输出）、Camera 等。

5．Zygote

Zygote 进程用于缩短 Android 应用程序加载的时间，每当执行 Java 应用程序时，Zygote 就会派生出一个子进程来执行应用程序，该子进程就是用来执行 Java 应用程序的虚拟机。

6．System Server

System Server 是 Android 系统的一个核心进程，它是由 Zygote 进程创建的，在 Android 启动过程中位于 Zygote 之后。在 System Server 中可以看到它建立的 Android 中的大部分服务，如 Activity Manager Service（管理应用程序的生命周期）、Location Manager Service（提供终端的地理位置信息）等。

为了将运行在 System Server 中的 Java 系统服务提供给 Android 应用程序或 Framework 内部模块调用，需要先将它们注册到 Context Manager 中。

在通过 Binder IPC 将 Java 系统服务注册到基于 C 语言的服务管理器时，需要使用 JNI（Java Native Interface）本地编程接口。JNI 允许 Java 代码与其他编程语言（如 C、C++、汇编语言）编写的应用程序和库进行交互操作。

以上就是对 Android 启动过程以及 Framework 初始化的简单介绍。当然这仅仅是 Android 启动过程的一部分，如图 1-3 所示，当所有 Java 系统服务加载完毕后，Activity Manager Service 会运行 HOME 应用，启动过程继续进行。这部分已超出本书所要讨论的范围，如果读者感兴趣，请参照 Android 源码进行分析。

出于篇幅的考量，本书不可能对 Android Framework 所有模块的所有动作作出说明。即便如此，沿着本书提供的思路，参考相关章节内容，分析相关源码，你会很快、很容易地掌握 Android Framework。

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 2 章

搭建 Android 开发环境

本章讲解的主要内容有，编译 Android 平台源代码、安装 SDK 开发工具与模拟器，以及调试应用程序 Framework 的方法。首先讲解基本的主机环境组成，然后讲解 Android 平台编译环境、SDK 开发环境搭建等内容。最后通过一个 Hello World 应用程序，学习在源代码级别上如何调试 Android 应用程序 Framework，并简单了解 Android 应用程序 Framework 的运行过程。

2.1 主机环境构成

虽然 Android 开发环境多种多样，但本书构建 Android 平台、模拟器的驱动，以及对应用程序 Framework 的调试都是在 Ubuntu Linux[1]操作系统平台上进行的。由于大部分机器都运行在微软公司的 Windows XP 操作系统下，所以需要先在 Windows XP 操作系统下安装 VirtualBox 虚拟机，再在此虚拟机上安装 Ubuntu Linux 操作系统。

首先简单地讲一下在 Windows XP 操作系统下如何安装 VirtualBox 虚拟机，然后再讲解在虚拟机上安装 Ubuntu Linux 操作系统的方法。最后，下载 Android 平台源代码，并进行编译。

2.1.1 安装 VirtualBox

VirtualBox 是由 Oracle 提供的一款开源、免费的虚拟机软件，其版本更新速度很快，与其他虚拟机软件相比更轻巧，运行速度更快，安装界面如图 2-1 所示。各位可从以下网站下载 VirtualBox：

http://www.virtualbox.org/wiki/Downloads

---

1 从 Android 官方网站上，下载完 Android 源代码后，编译代码，都在 Ubuntu Linux 系统上进行。http://source.android.com/ source/download.html。

更多更新编程资源尽在 http://www.pin5i.com/

图 2-1 | VirtualBox 安装界面

成功安装 VirtualBox 后，运行它，单击“新建”（快捷键 Ctrl+N）图标，新建虚拟机。在“新建虚拟电脑”窗口中，选择“系统类型”为 Linux 与 Ubuntu，如图 2-2 所示。在创建虚拟机时，应保证有足够的内存与硬盘空间，内存建议设置在 1.5GB 以上，硬盘空间要大于 10GB，编译 Android 源代码时，要求硬盘空间至少为 10GB。

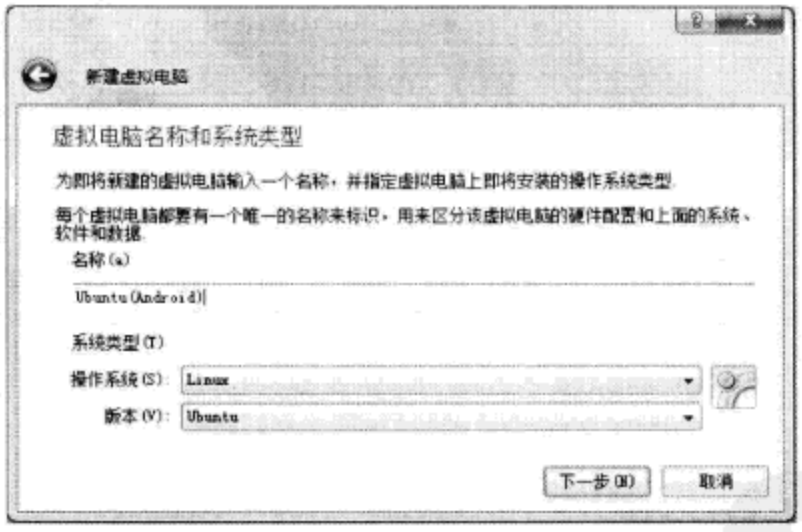


图 2-2 | 新建虚拟机

在使用 VirtualBox 搭建完开发环境后，将其保存为 VirtualBox 映像。在其他 PC 上，只要安装 VirtualBox，即可把 Android 开发环境移植到指定 PC 上，使用起来非常方便。

2.1.2 安装 Ubuntu

Ubuntu 是一个以桌面应用为主的 Linux 操作系统，应用非常广泛，各位从以下网

更多更新编程资源尽在 http://www.pin5i.com/

站即可下载：

http://www.ubuntu.com/desktop/get-ubuntu/download

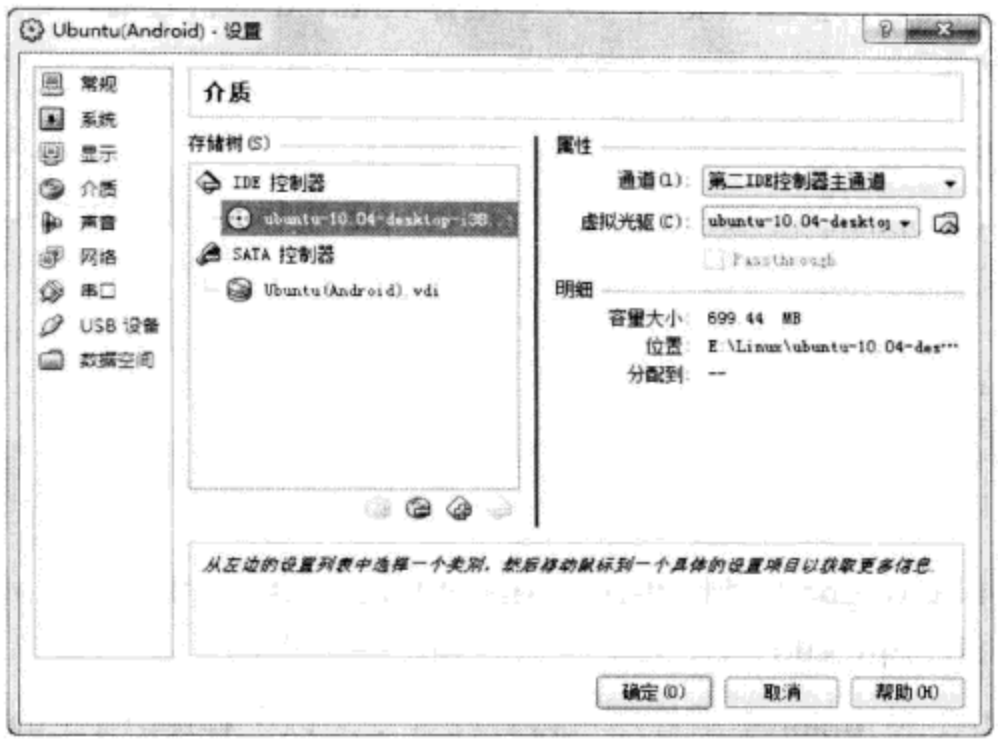


图 2-3 | 在 VirtualBox 的介质设置中，选择 Ubuntu 映像

在 VirtualBox 中，单击“设置”图标，弹出虚拟机设置窗口如图 2-3 所示，在其左侧列表中，选择“介质”，再在 IDE 控制器属性的虚拟光驱中，选择下载的 Ubuntu 映像文件[1]，再单击“开始”按钮，在虚拟机中安装 Ubuntu。

TIP Ubuntu?

Ubuntu 一词来自南非的祖鲁语或科萨语，原意为“因为有你所以有我”，是非洲传统的一种价值观，类似于我们所说的“仁爱”，Ubuntu 操作系统将这种思想带入软件世界。Ubuntu 操作系统每隔 6 个月发布一次新版本，标注在名称后的数字代表发布的年份与月份。至 2010 年 7 月，发布的最新版本为 Ubuntu 10.04 LTS (Long Term Support)。LTS 版本每两年发布一次，是长期支持版本，其桌面版本提供 3 年支持，服务器版本则提供长达 5 年的支持。

2.2　搭建 Android 平台编译环境

前面，我们讲解了在 Windows 平台下安装虚拟机，以及在虚拟机下安装 Ubuntu 的相关知识。下面我们将讲解 Android 所需要的一些编译工具，以及下载 Android 平台源码的方法。

1　ubuntu-10.01-desktop-i386.iso

更多更新编程资源尽在 http://www.pin5i.com/

2.2.1 编译工具

在 Ubuntu Linux（32-bit x86 环境）中编译 Android 平台之前，首先要在 Linux 中安装如下工具或包。

（1）Git 1.5.4 版本以上，GNU Privacy Guard。

（2）JDK 5.0，update 12 版本以上（因@override 注释问题，JDK 6.0 不支持）[1]若非 JDK 5 版本，编译时会产生如下错误：

```
You are attempting to build with the incorrect version of java.
Your version is: java version "1.6.0_15".
The correct version is: 1.5.
Please follow the machine setup instructions at
    http://source.android.com/download
```

（3）Flex，bison，gperf，libsdl-dev，libesd0-dev，libwxgtk2.6-dev(optional)，build-essential，zip，curl。

在 Ubuntu 中，使用 apt 实用程序可以轻松地安装以上程序。在 Linux 的 Shell 提示符下输入如下命令：

```
$ sudo apt-get install git-core gnupg flex bison gperf libsdl-dev libesd0-dev
libwxgtk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev[2]
```

若需要，可以安装 valgrind 工具，此工具用于检测内存泄露、堆栈溢出等问题。

```
$ sudo apt-get install valgrind
```

Ubuntu intrepid(8.10)的用户需要将系统更新到 libreadline 最新版本。

```
$ sudo apt-get install lib32readline5-dev
```

下面开始安装 Java。在 Ubuntu 10.04 软件源中，仅支持 Java 1.6，安装 Java 1.5 时，请按以下步骤进行。

（1）添加下载源：

```
$ sudo add-apt-repository "deb http://us.archive.ubuntu.com/ubuntu/  hard multiverse"
$ sudo add-apt-repository "deb http://us.archive.ubuntu.com/ubuntu/ hard-updates multiverse"
```

（2）更新包索引：

```
$ sudo apt-get update
```

---

1 @override 注释用在方法上，用来告诉编译器此方法是改写自父类或接口。此注释在 java 1.5 中只能用于对父类方法的重写，而不能用于对实现的接口中的方法的实现，否则编译器会生成一个错误信息。在 Android Froyo 发布后，在 Java 5 与 Java 6 中，对哪个版本是 Java 的编译器，曾有过讨论，据谷歌的 Jean-Baptiste Queru 说，1.5 用于内部开发 Froyo，是测试版本。

2 Zlib1g-dev：中间的 1 是数字 1，非字母 l。

更多更新编程资源尽在 http://www.pin5i.com/

（3）安装 Java 1.5：

```
$ sudo apt-get install sun-java5-jdk
```

（4）确认所有安装的 Java 版本：

```
$ sudo update-java-alternatives -l
```

（5）若系统存在多个 Java 版本，转换为 Java 1.5：

```
$ sudo update-java-alternatives -s java-1.5.0-sun
```

（6）确认 Java 版本：

```
$ java -version
java version "1.5.0_19"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_19-b02)
Java HotSpot(TM) Server VM (build 1.5.0_19-b02, mixed mode)
```

设置完 JDK 之后，再设置 JAVA_HOME。关于 Java 设置目录信息，可以在 Shell Prompt 中输入 find/usr-name"java-1.5*"查询。

```
$ export JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
$ export ANDROID_JAVA_HOME=$JAVA_HOME
$ export PATH=$JAVA_HOME/bin:$PATH
```

2.2.2　安装 Repo

Android 源代码十分庞大，通过 Git 版本控制工具管理源码，Android 是由 kernel、Dalvik、Bionic、prebuilt、build 等多个 Git 项目组成，如果分别使用 Git 来逐个获取显然很麻烦，所以 Android 项目编写了一个名为 Repo 的 Python 脚本来统一管理这些项目的仓库，使得项目的获取更加简单。

使用 curl 工具下载 Repo 脚本文件后，再更改运行权限。

```
$ cd ~
$ mkdir bin
$ export PATH=$PATH:~/bin
$ curl http://android.git.kernel.org/repo >~/bin/repo
$ chmod a+x ~/bin/repo
```

2.2.3　下载 Android 源代码

下载完 Repo 脚本文件后，执行如下命令，下载包含 Android 源码发布信息的 manifest.git 文件。在-b 选项后输入版本名称，即可下载指定版本的源码。比如 froyo 表示下载 froyo 版本的 manifest.git 文件。若去除-b 选项，则下载当前主版本的 Android 源码。[1]

1　2010 年 7 月最新版本为 froyo 版本，下载时，需要在-b 选项后标出 froyo 字样。

更多更新编程资源尽在 http://www.pin5i.com/

```
$ cd ~
$ mkdir mydroid
$ cd mydroid
$ repo init -u git://android.git.kernel.org/platform/manifest.git -b froyo-plus-aosp
```

敲入以上代码后，会输出与 froyo Branch 相关的源码信息，而后提示输入用户信息，依次输入用户名与邮件地址。命令执行完毕后，输出目录信息，并生成.repo 目录，如下所示。

```
android@android-desktop:~/mydroid$ ls -al
total 80
drwxr-xr-x 19 android android 4096 2010-06-29 20:08 .
drwxr-xr-x 30 android android 4096 2010-06-29 20:08 ..
drwxr-xr-x  6 android android 4096 2010-06-29 17:11 .repo
```

最后，执行 repo sync 命令，下载 Android 源代码。Android 源代码总大小大于 2GB，下载过程非常漫长。

```
$repo sync
```

2.2.4 编译 Android 源代码

Android 源码下载完成后，开始编译源代码。若需要移植，则需要进行很多设置。当然，如果只是搭建一个模拟环境，编译时，只要保持默认设置即可。首先进入 Android 源码所在的目录，而后敲入 make 命令，执行编译，命令如下。

```
$ cd ~/mydroid
$ make
```

在笔者的编译环境下，编译结果如图 2-4 所示。

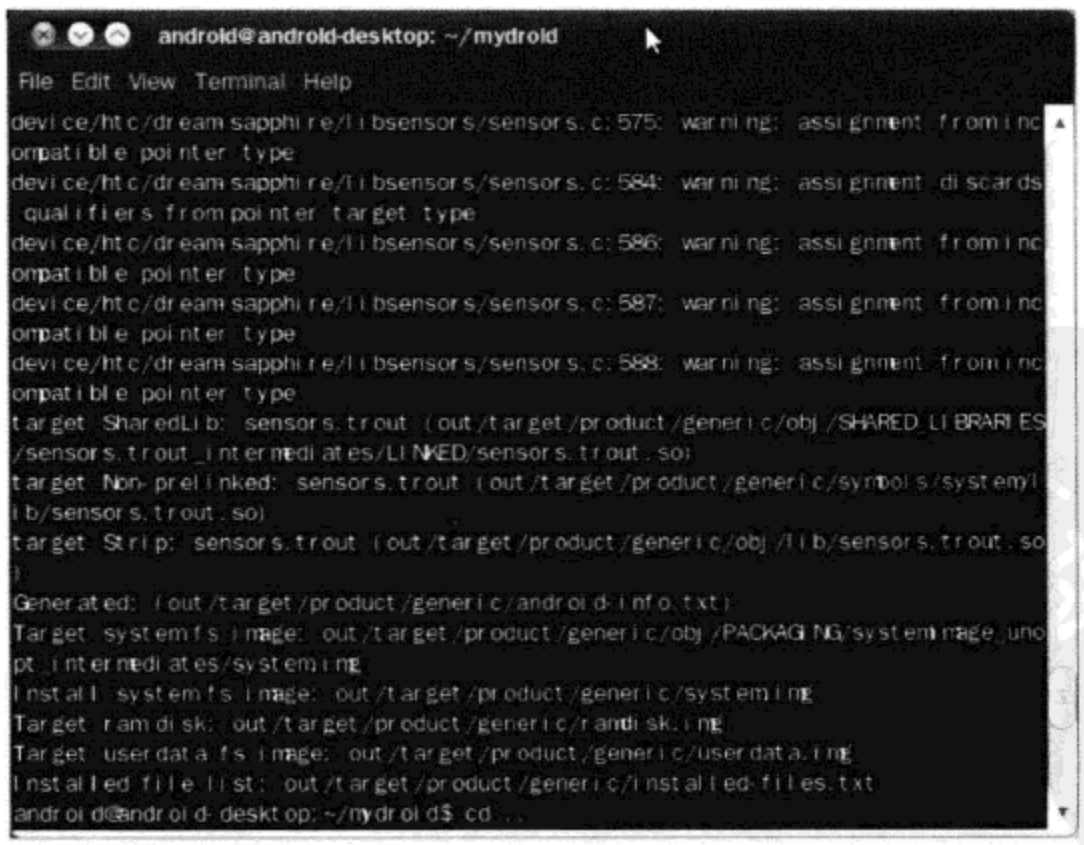


图 2-4 | 成功编译 Android 源代码后，显示的信息

更多更新编程资源尽在 http://www.pin5i.com/

2.3　搭建 Android SDK 开发环境

前面，讲解了 Android 平台的构成，从 Git 服务器下载源码并进行编译的内容。接下来，讲解如何搭建 Android SDK 开发环境，包括 Eclipse、Android SDK、ADT Eclipse 插件下载与安装等内容，这些都与 Android 应用程序开发与调试相关。

2.3.1　下载、安装 Eclipse

首先，进入 http://www.eclipse.org/downloads 网站，下载“EclipseIDE for Java Developers”，如图 2-5 所示。

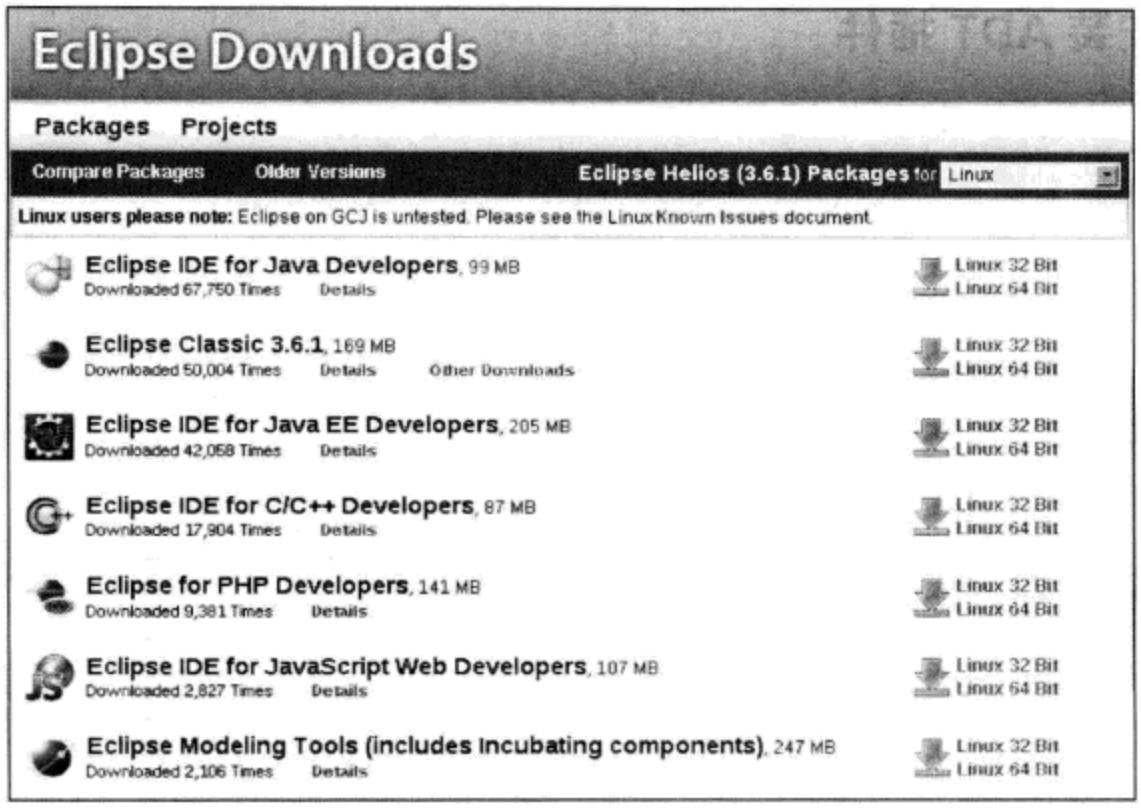


图 2-5 | Eclipse 下载页面

下载完成后，解压缩，运行 Eclipse。

```
$ tar -xvzf eclipse-java-helios-SR1-linux-gtk.tar.gz
$ cd eclipse
$ ./ eclipse &
```

2.3.2　下载 Android SDK starter

进入 Android 官网（http://developer.android.com/sdk/index.html），下载 Android SDK starter 后，解压缩。

更多更新编程资源尽在 http://www.pin5i.com/

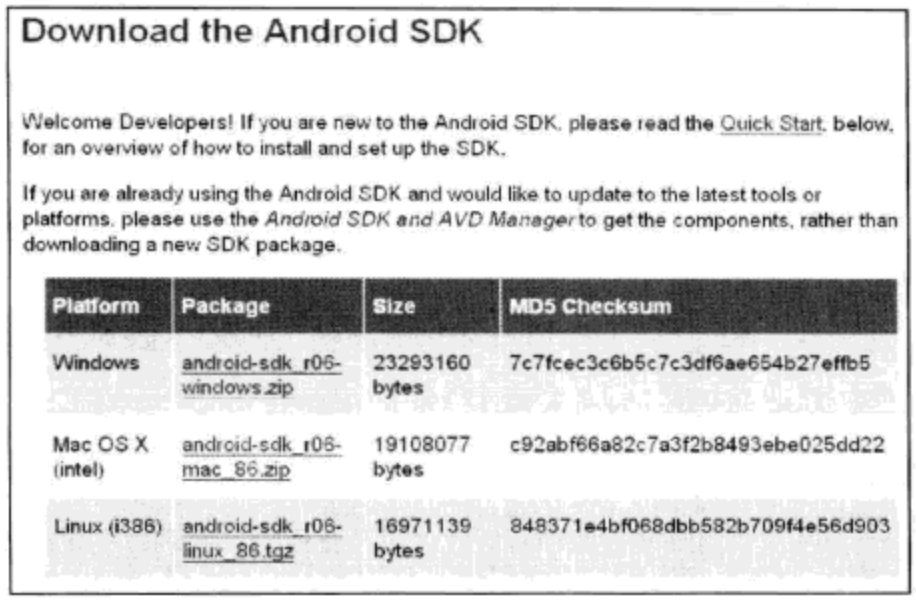


图 2-6 | Android SDK 下载页面

2.3.3 安装 ADT 插件

下面我们开始一起安装 ADT（Android Development Tool）插件。

在 Eclipse 菜单中，依次单击 Help>Install New Software 菜单，如图 2-7 所示，在弹出的窗口中，单击 Add 按钮，在弹出的添加源对话框中，如图 2-8 所示，输入名称 ADT 与地址（https://dl-ssl.google.com/android/eclipse），而后单击 OK 按钮。

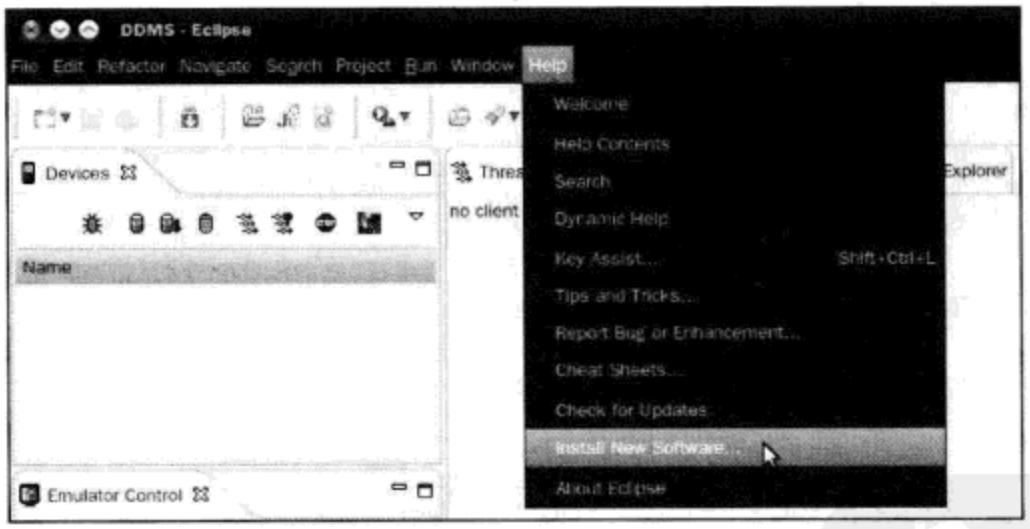


图 2-7 | 安装 Android ADT（1/4）

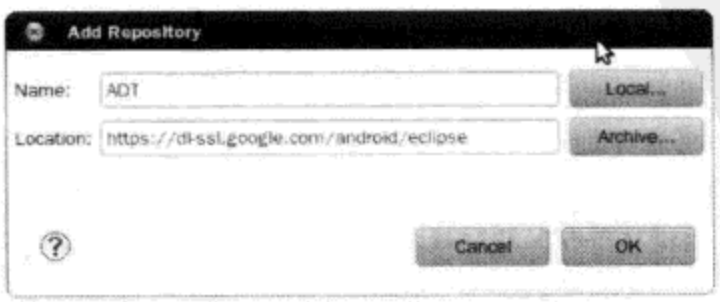


图 2-8 | 安装 Android ADT（2/4）

在列表中，选择 Android DDMS 与 Android Development Tools，而后单击 Next 按

更多更新编程资源尽在 http://www.pin5i.com/

钮，进入用户协议窗口，如图 2-9 所示。

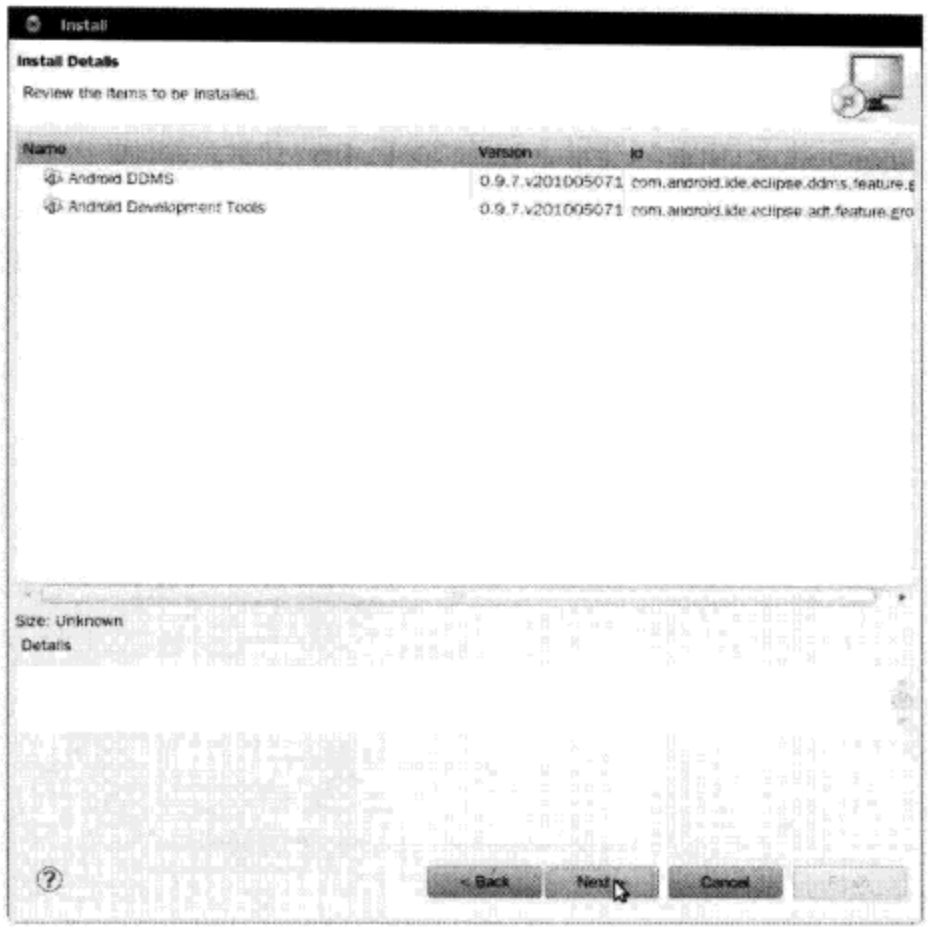


图 2-9 | Android ADT 安装（3/4）

同意所有协议，开始安装，如图 2-10 所示。

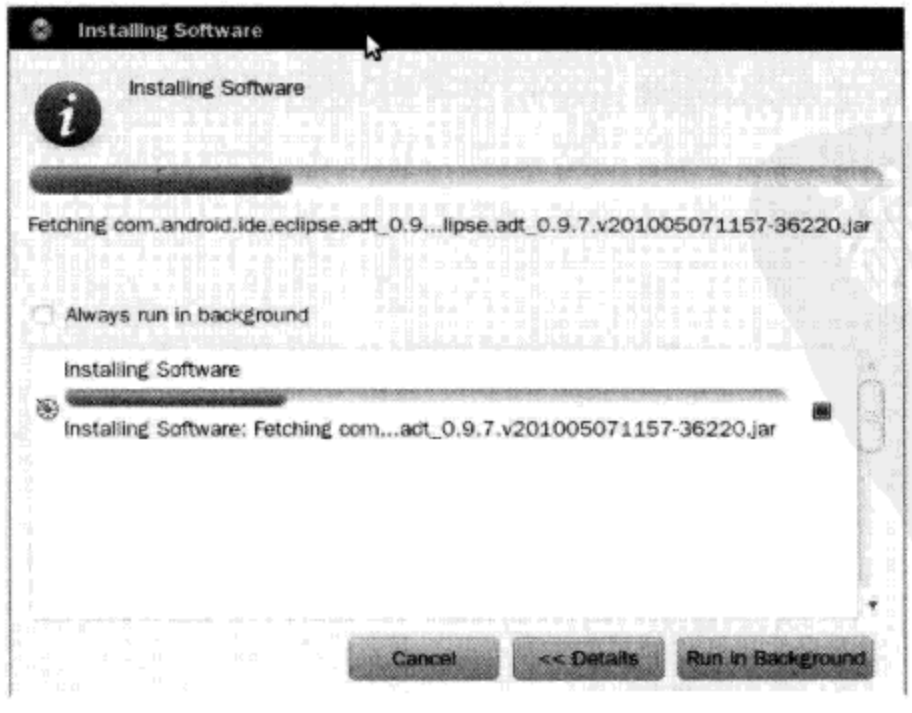


图 2-10 | Android ADT 安装（4/4）

安装完成后，重启 Eclipse。

更多更新编程资源尽在 http://www.pin5i.com/

2.3.4 设置 Android SDK 路径

重启 Eclipse 后，在菜单栏中，依次单击 Windows>Preferences 菜单，打开 Preferences 窗口，准备将 Android SDK Starter 路径设置至 Eclipse 中，如图 2-11 所示。

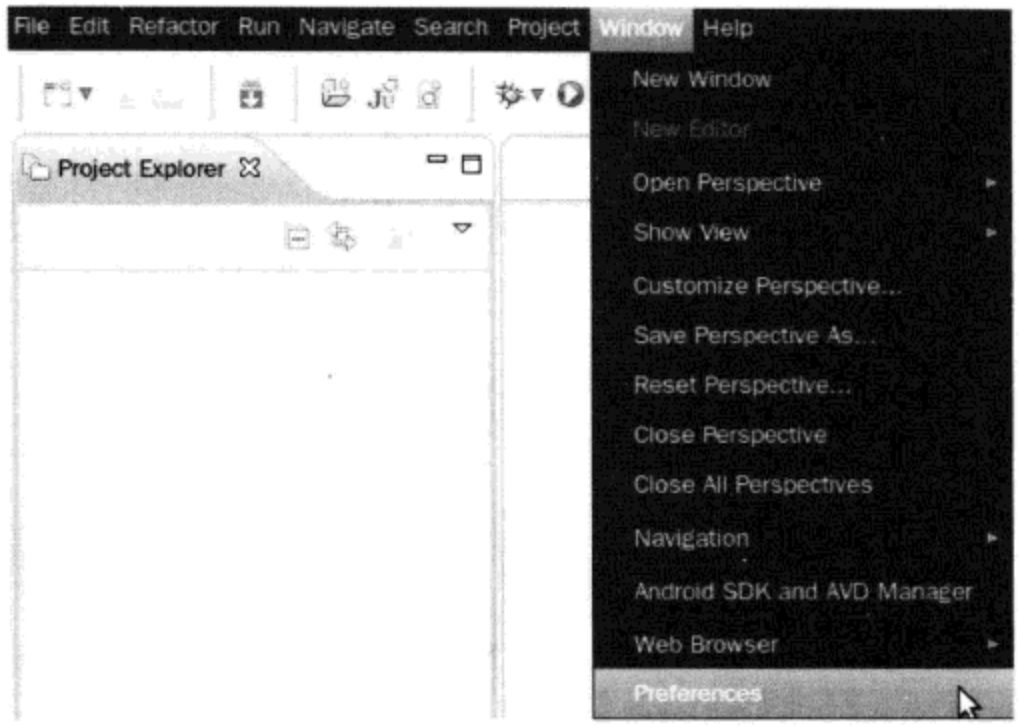


图 2-11 | 设置 Android SDK 路径（1/2）

在 Preferences 窗口的左侧列表中，可以看到一项 Android，选中它，在右侧的 SDK Location 中，指定解压后的 Android SDK Starter 路径，如图 2-12 所示。

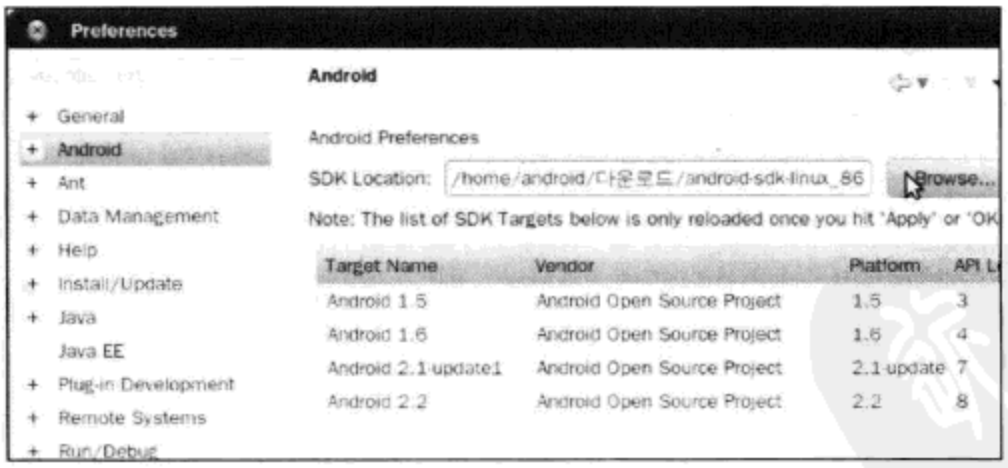


图 2-12 | 设置 Android SDK 路径（2/2）

2.3.5 安装 Android SDK

在 Android SDK Starter 包中只包含开发 Android 应用程序所用到的几个工具，并不包含 Android 类库、平台模拟器、示例程序、参考文档等，而这些又是开发 Android 应

更多更新编程资源尽在 http://www.pin5i.com/

用程序所必需的，需要使用 Eclipse ADT 插件中的 Android SDK and AVD Manager 工具，下载并添加到 Android SDK 中，如图 2-13 所示。

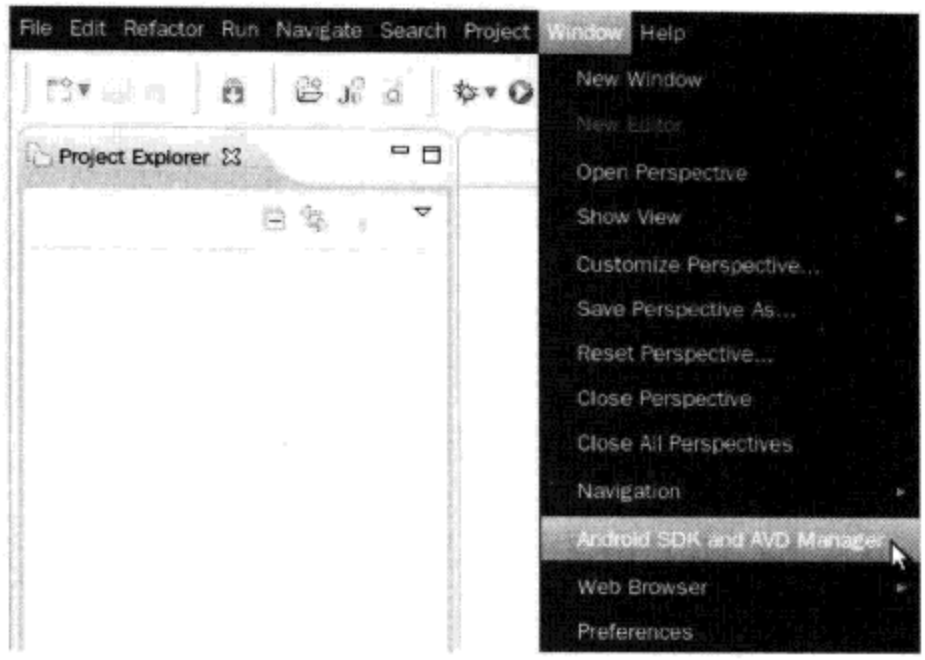


图 2-13 | 运行 Android SDK and AVD Manager

在 Eclipse 中，依次单击 Window>Android SDK and AVD Manager 菜单，打开 Android SDK and AVD Manager 窗口。

如图 2-14 所示，选择 Available Package 菜单，在窗口右侧显示出许多待添加项目，主要有 Android SDK 平台版本、谷歌 API、示例代码、文档等，选中需要下载安装的项目后，单击 Install Selected 按钮。

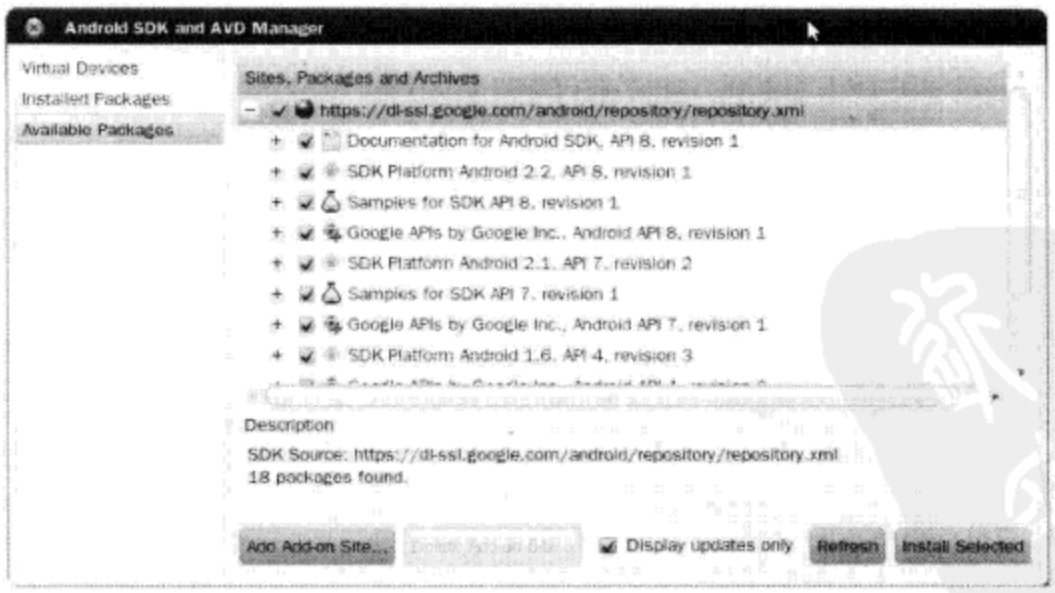


图 2-14 | 选择添加到 Android SDK 中的项目

同意所有 Package Description & Licence 后，进行下载安装，如图 2-15 所示。

更多更新编程资源尽在 http://www.pin5i.com/

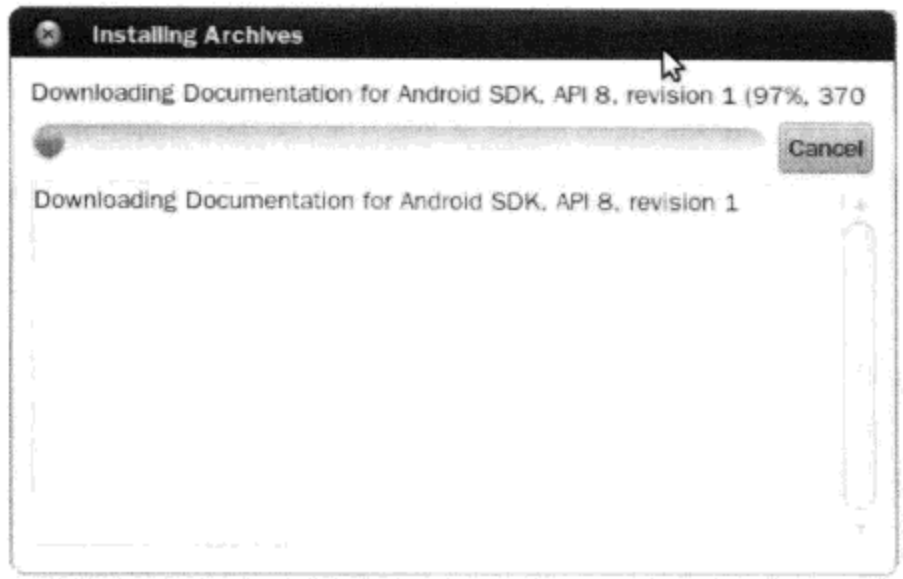


图 2-15 | 下载选中的项目

2.4 开发 Android 应用程序

在上一节中，我们已经安装好 Eclipse、Eclipse ADT 插件、Android SDK，搭建好了 Android 应用程序开发环境。下面我们将编写一个 Hello 应用程序，并在模拟器中运行它，以测试开发环境是否搭建成功。

编写 Hello 应用程序

在开始编写 Hello 应用程序之前，首先创建 Android 工程，在 Eclipse 菜单栏中，依次选择 File>New> Project>Android Project 命令，如图 2-16 所示。

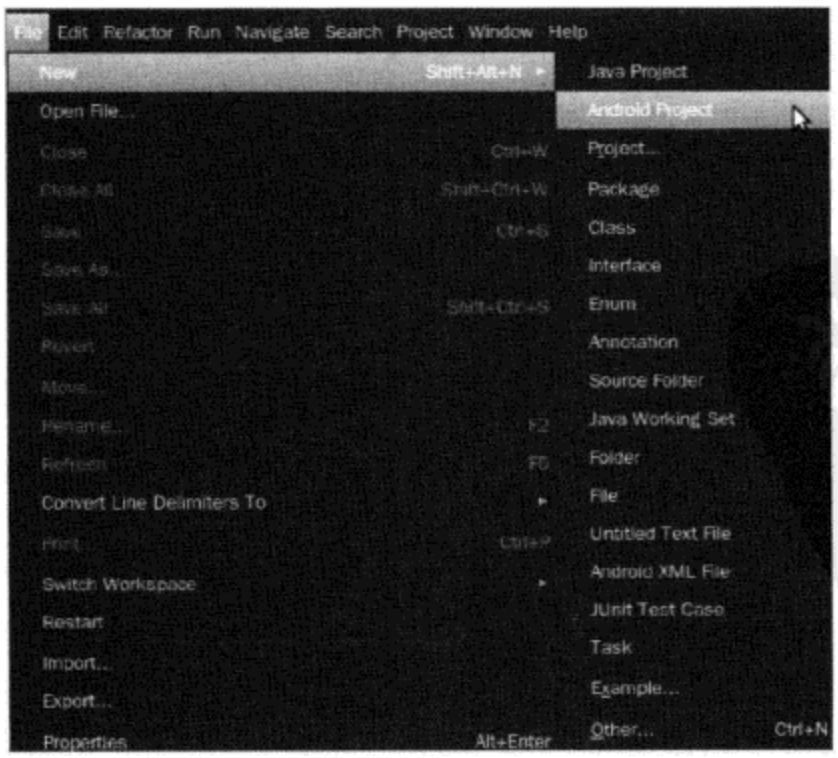


图 2-16 | 创建 Android 工程

更多更新编程资源尽在 http://www.pin5i.com/

在 New Android Project 窗口中，输入相关信息，单击 Finish 按钮后，在 Eclipse 左侧的 Package Explorer 中，可以看到创建好的 HelloWorld 工程。

代码 2-1 是 Eclipse 自动生成的程序代码。

```
package org.example.hello;

import android.app.Activity;
import android.os.Bundle;
public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

代码 2-1 | HelloWorld.java 源代码

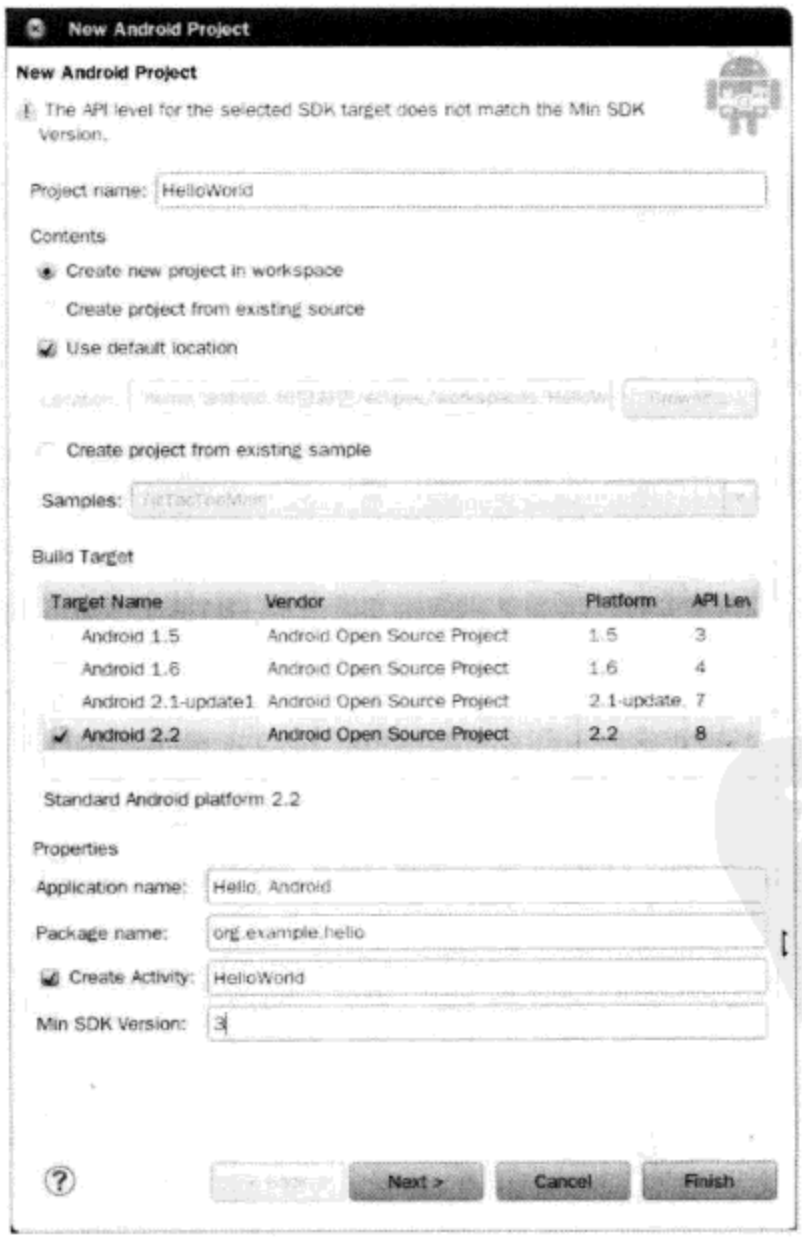


图 2-17 | 新建 Android 工程窗口

更多更新编程资源尽在 http://www.pin5i.com/

运行 HelloWorld 应用程序时，会弹出如图 2-18 所示的错误信息提示窗口，提示用户没有可运行应用程序的 Android 设备。

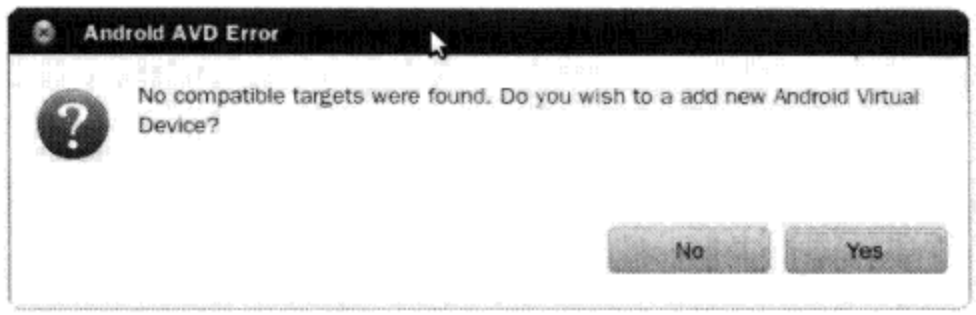


图 2-18 | 提示缺少程序运行模拟器

下面开始创建 Android 虚拟设备。在 Eclipse 菜单栏中，依次单击 Window>Android SDK and AVD Manager 菜单，在 Android SDK and AVD Manager 左侧列表中，选择 Virtual Devices，在弹出的新建虚拟设备窗口中，输入虚拟设备的相关信息，而后单击 Create AVD 按钮，创建 Android 虚拟设备，如图 2-19 所示。

在 Eclipse 左侧 Package Explorer 窗口中，选中 HelloWorld 工程，单击鼠标右键，在弹出的菜单中，选择 Run as>Android Application，启动模拟器，运行 HelloWorld 程序，输出如图 2-20 所示的信息。

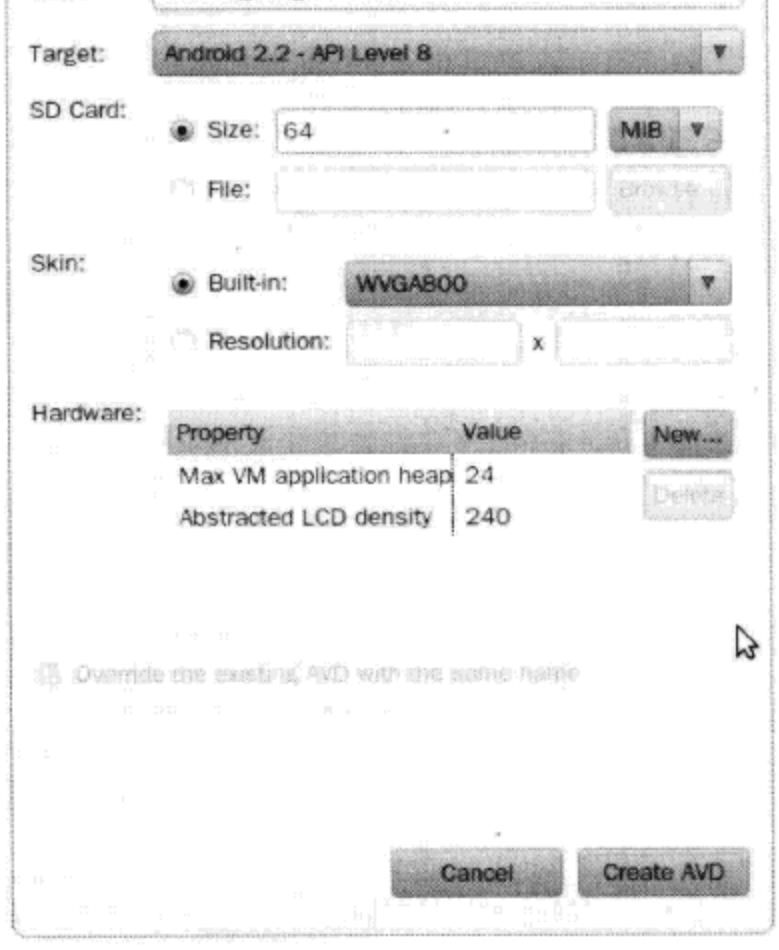


图 2-19 | Create new AVD 窗口

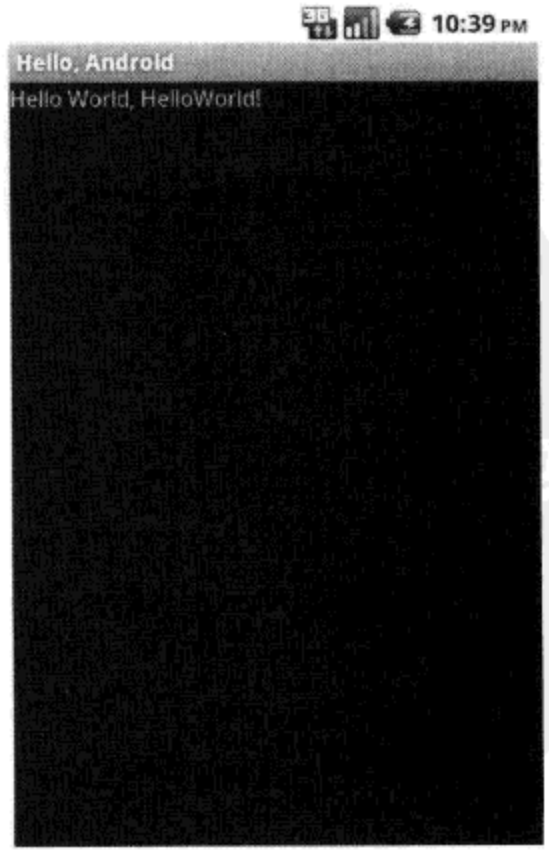


图 2-20 | HelloWorld 程序运行界面

更多更新编程资源尽在 http://www.pin5i.com/

关于如何编写程序代码的内容已超出本书所涉及的范围，请参考相关书籍，在此省略。

2.5　应用程序 Framework 源码级别调试

Android 平台由采用 C/C++开发的 Linux 驱动、本地库、应用程序 Framework（采用 Java 开发）以及应用程序四部分组成。在开发 Android 平台代码时，要经常调试编好的程序。下面以 Java 代码为例，讲解如何进行应用程序 Framework 源码级别的调试。

2.5.1　加载应用程序 Framework 源

在调试系统源码之前，首先要把系统源码调入 Eclipse 中。但在 Android 平台目录中存在数量庞大的 Framework 源码，若想将它们全部调入到 Eclipse 中，需要花费大量功夫。值得庆幸的是在 Android 中包含 Eclipse 设置文件，利用它可以把 Framework 的所有源码路径保存下来。下面列出了操作步骤，依据这些步骤，即可把 Framework 源码调入 Eclipse 之中。

（1）如 2.2 节所述，先搭建好 Android 编译环境，下载 Android 平台源代码，再使用 make 命令，编译 Android 系统。

（2）再把 Android 提供的.classpath 文件复制到 Android 平台源码所在的最上层目录中。

```
$ cd mydroid
$ cp development/ide/eclipse/.classpath ~/mydroid
$ chmod u+w .classpath
```

（3）运行 Eclipse，在菜单栏，依次单击 File-New-Java Project，如图 2-21 所示。在弹出的 New Java Project 窗口中，输入工程名称后，指定 Location 为 Android 源码所在的目录（.classpath 文件复制的位置），而后单击 Finish 按钮。

（4）查看 Eclipse 左侧的 Package Explorer 窗口，可以看到 Android Framework 源码被导入进来，如图 2-22 所示。

更多更新编程资源尽在 http://www.pin5i.com/

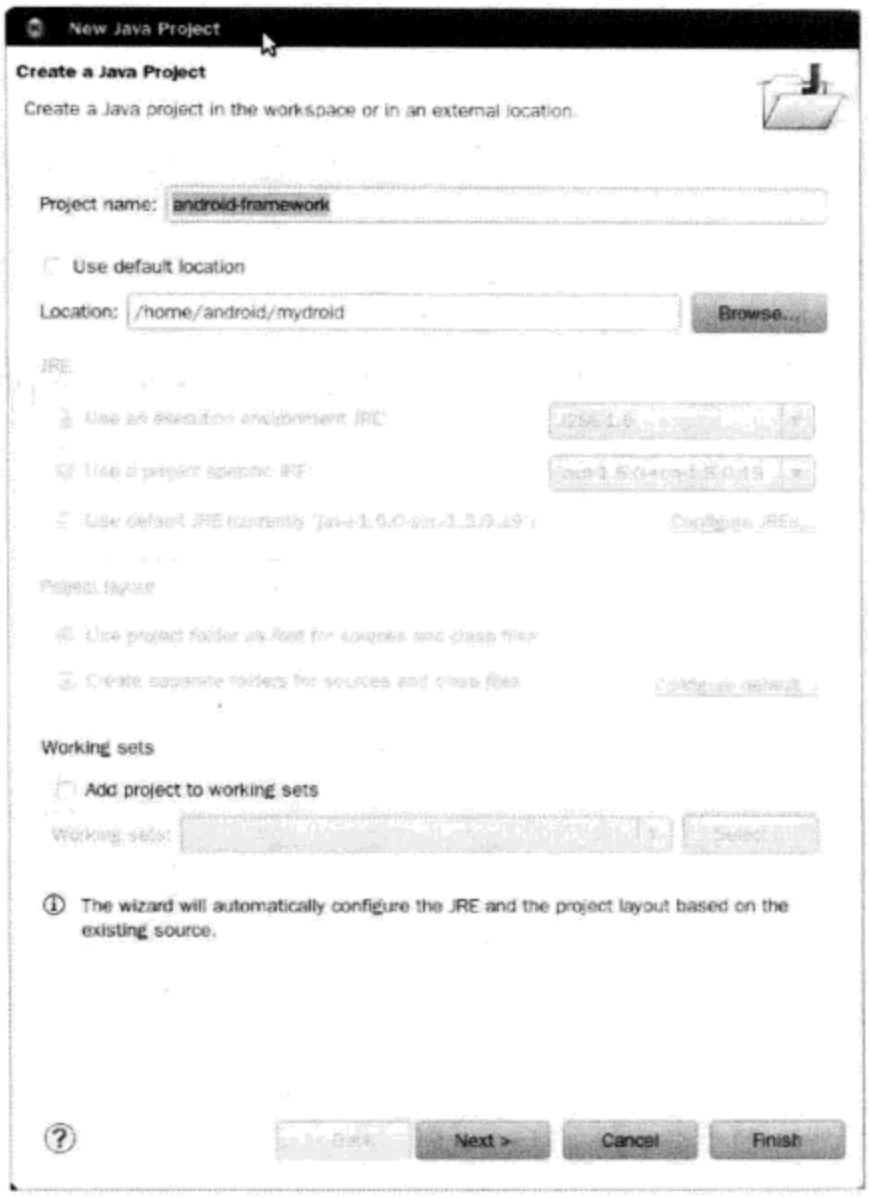


图 2-21 | 创建 Java 工程时，指定 Android Framework 源路径

图 2-22 | 导入到工程中的 Android Framework 源代码

TIP

在创建工程时，若出现如下错误。

Description Resource Path Location Type
Project 'Generic-Android' is missing required library: 'out/target/common/obj/JAVA_LIBRARIES/google-common_intermediates/javalib.jar'
Generic-Android Build path Build Path Problem

Description Resource Path Location Type
Project 'Generic-Android' is missing required library: 'out/target/common/obj/JAVA_LIBRARIES/gsf-client_intermediates/javalib.jar'
Generic-Android Build path Build Path Problem

请参考谷歌“android-platform”Groups 中 Quiring，Sam 的答复。

http://groups.google.com/group/android-platform/browse_thread/thread/5c86d1f1929eed3c

更多更新编程资源尽在 http://www.pin5i.com/

（5）接下来，设置调试器，以便调试 Android 平台源代码。如图 2-23 所示，在 HelloWorld 工程上单击鼠标右键，在弹出的菜单中，依次选择 Debug As>Debug Configurations 选项，打开 Debug Configurations 窗口。

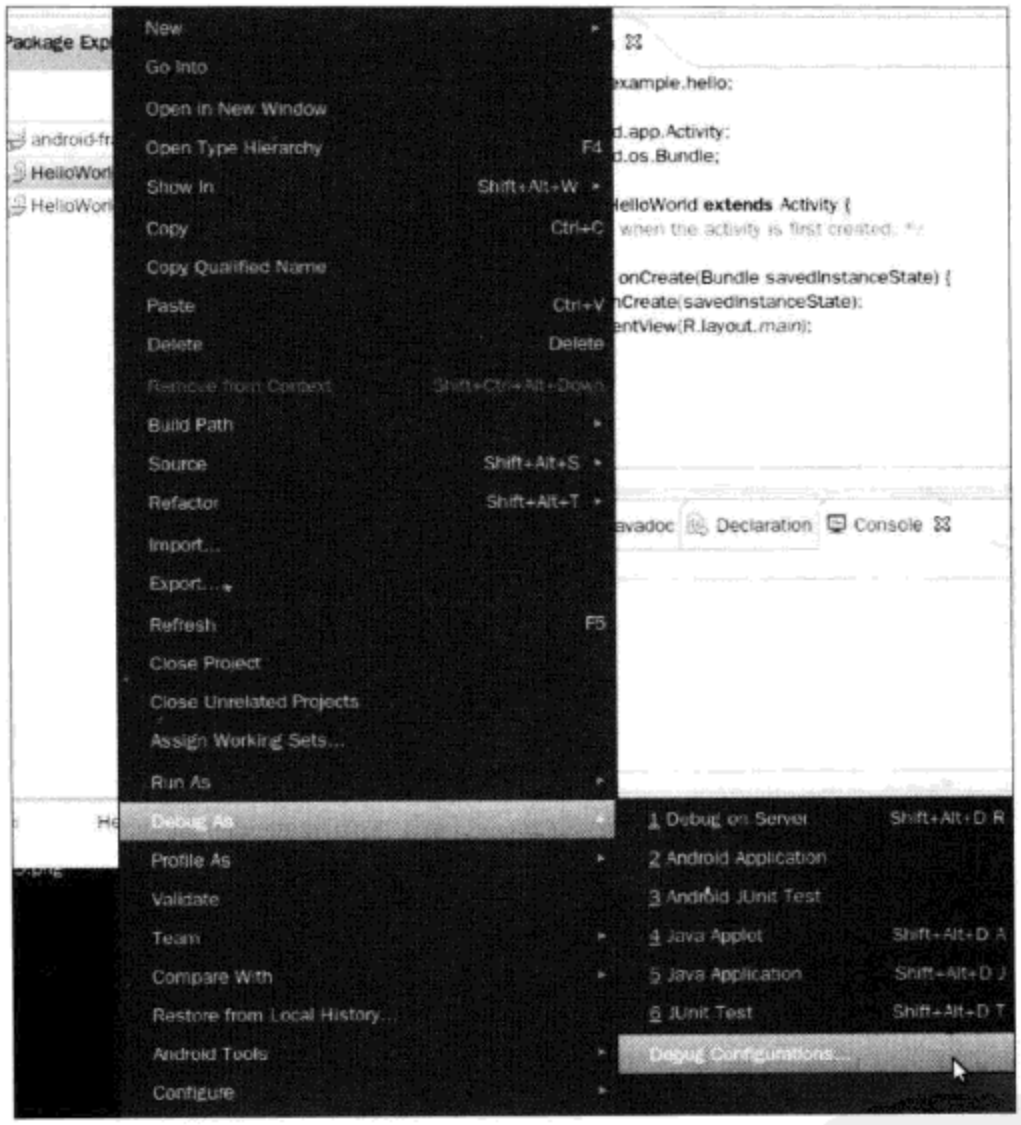


图 2-23 | Android 平台调试器设置

（6）在 Debug Configurations 窗口左侧列表中，选择 Remote Java Application，单击鼠标右键，在弹出的菜单中，选择 New，在右侧窗框内进行相应设置后，单击 Apply 按钮，应用更改项目，如图 2-24 所示。

更多更新编程资源尽在 http://www.pin5i.com/

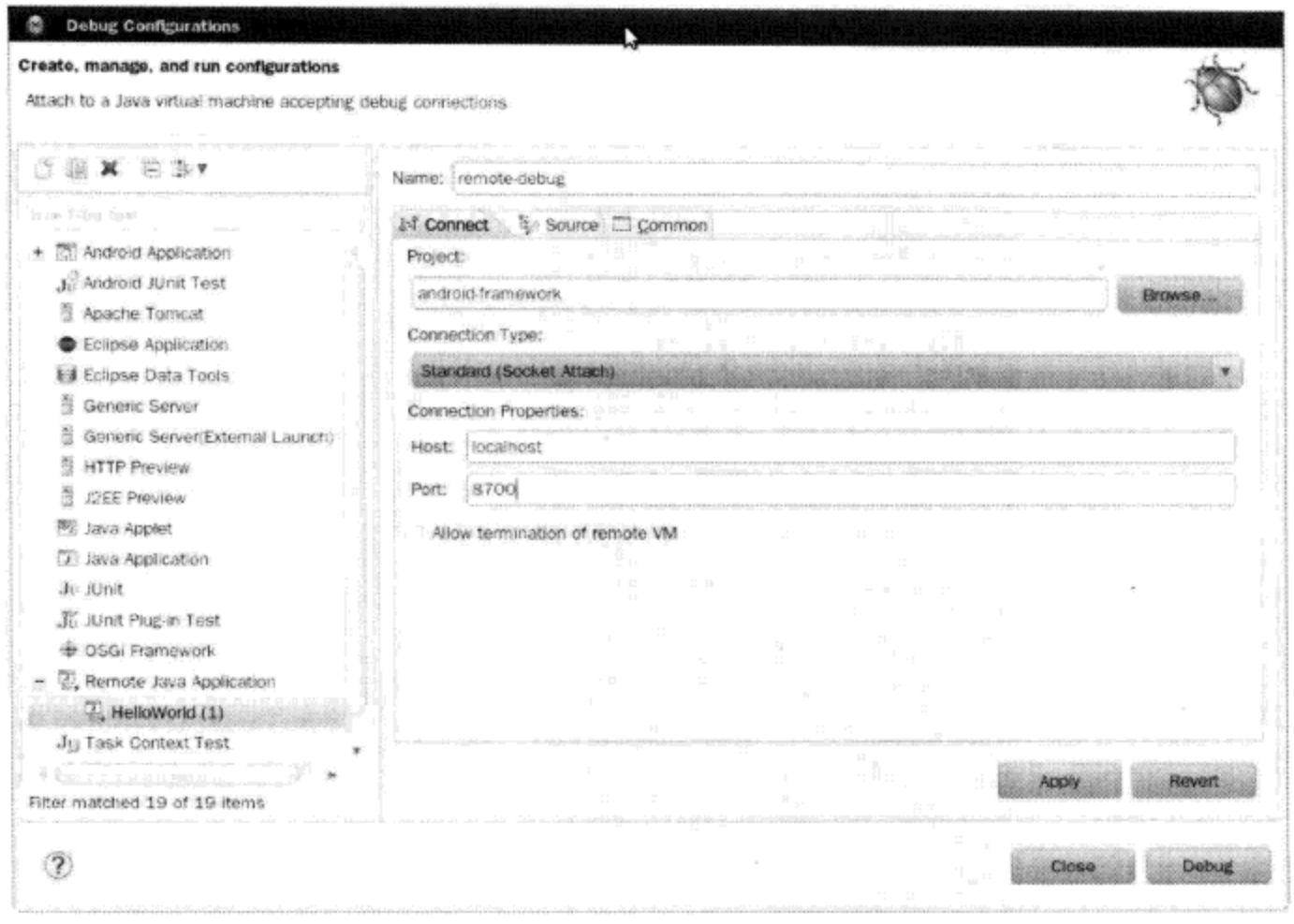


图 2-24 | 设置 Remote Java Application

2.5.2 调试 HelloWorld Framework（源码级）

调试 Android 应用程序 Framework 的准备工作完成后，接下来开始调试 HelloWorld 程序，跟随与应用程序相关的 Framework 源代码，查看变量的值。

（1）首先在 HelloWorld 应用程序起始的地方，设置一个断点，而后在 HelloWorld 工程上单击鼠标右键，在弹出的菜单中，依次选择 Debug As>Android Application，打开调试窗口，程序运行到断点的位置停止，并把控制权交给 Eclipse 的 Java 调试器，如图 2-25 所示。

（2）为了在源码级别上调试应用程序 Framework，在左上 Debug 窗口中，选择 ActivityThread 类，出现源码无法找到的错误，如图 2-26 所示。此时在 ActivityThread.perform LaunchActivity 上，单击鼠标右键，在弹出的菜单中，选择 Edit Source Lookup 菜单，弹出 Edit Source Lookup Path 窗口。

更多更新编程资源尽在 http://www.pin5i.com/

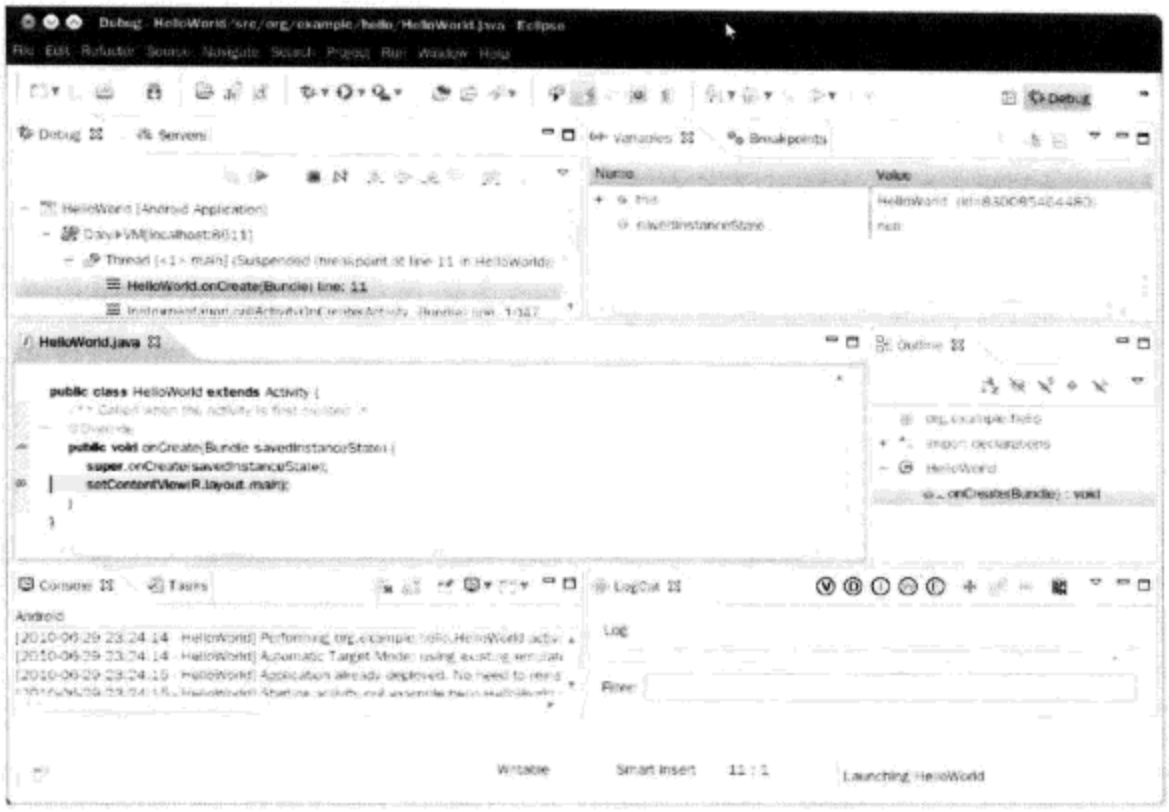


图 2-25 | 调试 HelloWorld 应用程序

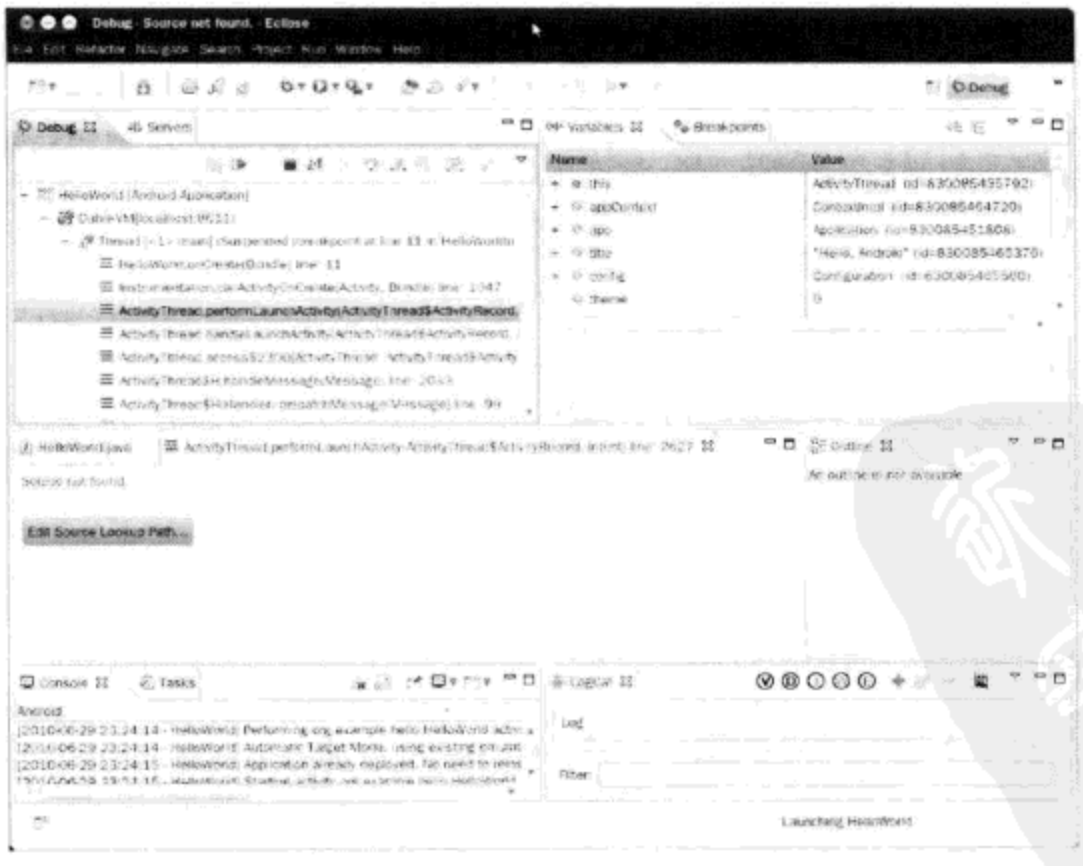


图 2-26 | 调试时，发生无法找到源代码错误

（3）在 Edit Source Lookup Path 窗口中，单击 Add 按钮，如图 2-27 所示。

（4）选中 Java Project 后，再选择 2.5.1 节中创建的 Android-Framework 工程，如图 2-28 所示。

更多更新编程资源尽在 http://www.pin5i.com/

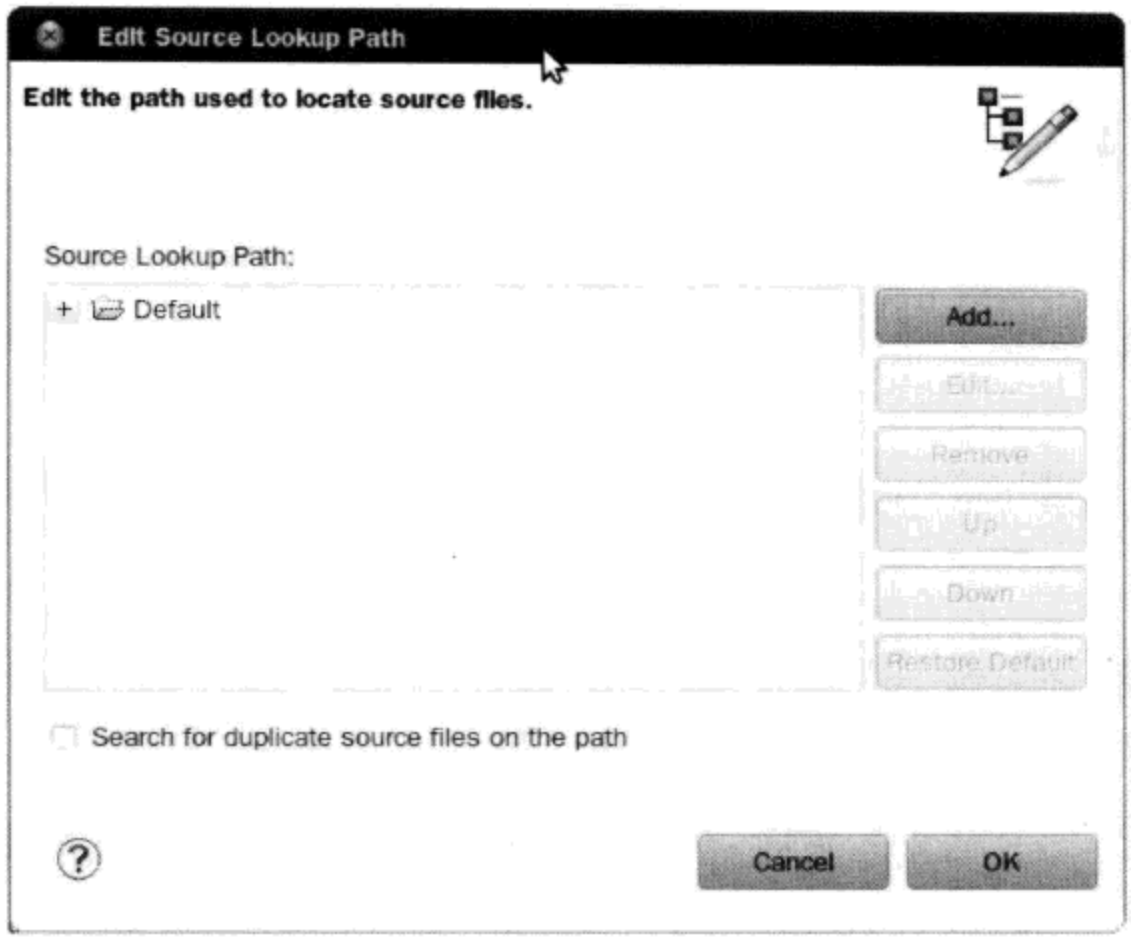


图 2-27 | Edit Source Lookup Path 窗口

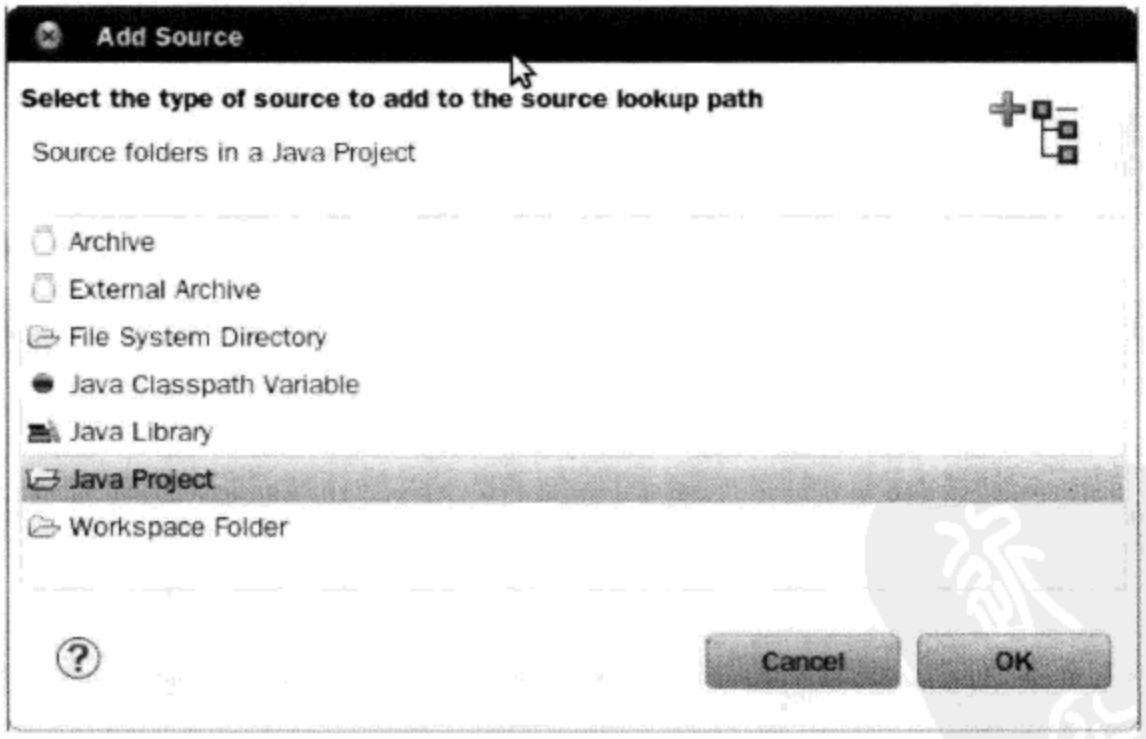


图 2-28 | Add Source 窗口

（5）如图 2-29 所示，若能看到 ActivityThread 类的源码，表示源码添加成功。在右上侧的 Variables 窗口中，可以看到应用程序 Framework 中的所有变量值。

更多更新编程资源尽在 http://www.pin5i.com/

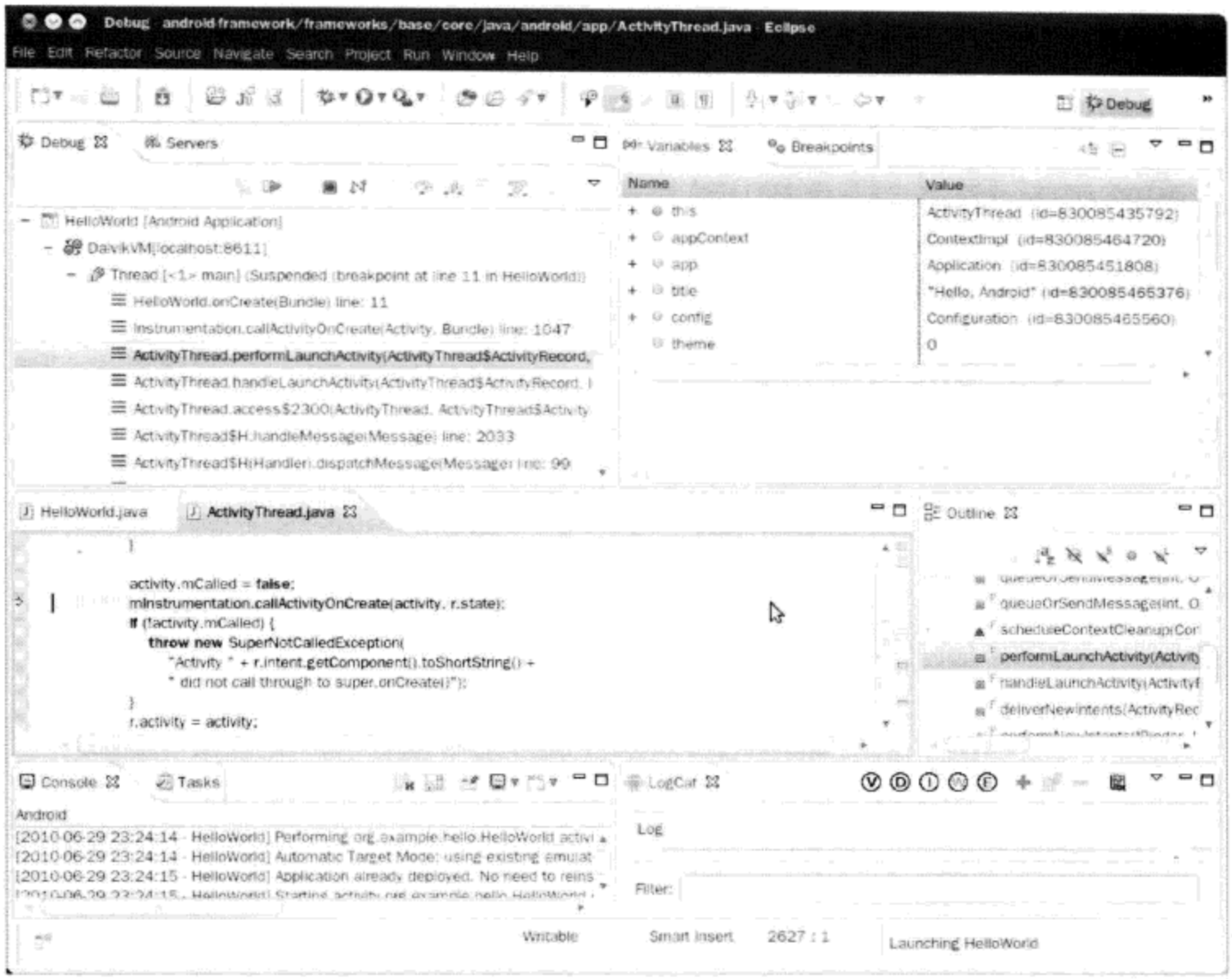


图 2-29 | 调试 ActivityThread 类源码

2.6 小结

本章主要讲解的内容有 Android 平台编译、Android SDK 安装，以及调试应用程序 Framework 的方法，其中用到的主要程序整理如下。

- VirtualBox：一款运行在 Windows 平台下的虚拟机软件，用于安装 Ubuntu 操作系统，以便编译 Android 平台源码。
- Ubuntu：一款以桌面为主的 Linux 操作系统，是编译 Android 源码所需的环境。
- Git：一个版本控制工具，采用分布式版本库的方式，管理 Android 源码。
- Repo：一个 Python 脚本，用于更方便地下载 Android 源码。

更多更新编程资源尽在 http://www.pin5i.com/

- Eclipse：一个开放源代码的、基于 Java 的可扩展开发平台，用于 Android 应用程序开发与调试。
- Android SDK：Android 程序开发工具包，包含程序开发工具与库等。
- ADT Plugin for Eclipse：一个 Eclipse 插件，是 Android 在 Eclipse 上的开发工具。
- Development/ide/eclipse/.classpath：一个设置文件，包含 Android 应用程序 Framework 的 Java 源码及编译后的类文件的路径。

更多更新编程资源尽在 http://www.pin5i.com/

第 3 章

init 进程

众所周知，Linux 中的所有进程都是由 init 进程创建并运行的。首先 Linux 内核启动，然后在用户空间中，启动 init 进程，再依次启动系统运行所需的其他进程。在系统启动完成后，init 进程会作为守护进程监视其他进程。若某个监视中的进程一旦终结，进入僵死状态时，它就会释放进程所占用的系统资源。在 Android 平台（下称 Android）中也存在 init 进程，除了提供以上常见的功能外，还提供几种额外的功能。

在本章中，将讨论学习 Android 的 init 进程的功能，需要读者具有一定的 Linux 系统编程知识，如果您对 Linux 编程知识感到陌生，建议事先翻阅相关的书籍进行学习。

3.1 init 进程运行过程

与 Linux 类似，init 进程是 Android 启动后，由内核启动的第一个用户级进程。

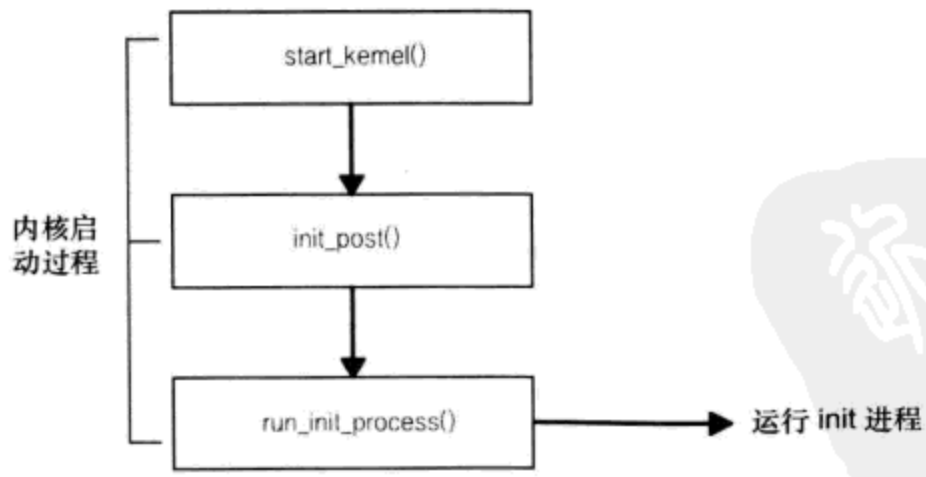


图 3-1 | init 进程启动过程

由图 3-1 init 进程启动过程图可以看出，init 进程是在顺次执行完 start_kernel() 函数、init_post()函数、run_init_process()函数后，最后启动执行的。代码 3-1 是内核内部

更多更新编程资源尽在 http://www.pin5i.com/

实现的 init 进程启动代码。

```
static int noinline init_post(void)
{
    if (execute_command) {          ❶
        run_init_process(execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");
}
```

代码 3-1 | 由 init_post()[1]启动 init 进程

❶ init_post()函数调用 run_init_process()函数，获取注册在 execute_command 中的进程文件路径[2]，执行 execve()系统调用。execve()函数执行由参数传递过来的文件路径下的进程。注意在设置内核启动选项时，应设置为“init = /init”，以便正常运行 init 进程。因为在编译完 Android 后生成的根文件系统中，init 进程位于最顶层目录中，如图 3-2 所示。

```
# ls -l /
ls -l /
drwxrwxrwt root     root              2010-03-22 14:21 sqlite_stmt_journals
dr-x------ root     root              2010-03-22 14:21 config
drwxrwx--- system   cache             2010-01-31 14:39 cache
d---rwxr-x system   sdcard_rw          1970-01-01 00:00 sdcard
lrwxrwxrwx root     root              2010-03-22 14:21 d -> /sys/kernel/debug
lrwxrwxrwx root     root              2010-03-22 14:21 etc -> /system/etc
drwxr-xr-x root     root              2009-12-19 00:45 system
drwxr-xr-x root     root              1970-01-01 00:00 sys
drwxr-x--- root     root              1970-01-01 00:00 sbin
dr-xr-xr-x root     root              1970-01-01 00:00 proc
-rwxr-x--- root     root         12215 1970-01-01 00:00 init.rc
-rwxr-x--- root     root          1677 1970-01-01 00:00 init.goldfish.rc
-rwxr-x--- root     root        103092 1970-01-01 00:00 init
-rw-r--r-- root     root           118 1970-01-01 00:00 default.prop
drwxrwx--x system   system            2010-03-01 21:08 data
drwx------ root     root              2009-11-02 22:59 root
drwxr-xr-x root     root              2010-03-22 14:21 dev
#
```


图 3-2 | Android 根文件系统

当根文件系统顶层目录中不存在 init 进程，或未指定启动选项“init =”时，内核会到/sbin、/ect、/bin 目录下查找 init 文件。如果在这些目录中仍未找到 init 文件，内核就会终止执行 init 进程，并引发 Kernel Panic。

若代码 3-1 中代码❶正常执行，那么 init 进程就会正常启动。Android 是个开源系统，只要下载并查看源代码，即可把握某些进程进行的动作。在下一节中，我们将一起

1 此函数被定义在 init/main.c 中。

2 进程文件路径被定义成内核启动选项，此选项可在内核 make menuconfig 上的 boot option 中确认，或者在内核源码顶层目录中的.config 文件中查看“CONFIG_CMDLINE”。

更多更新编程资源尽在 http://www.pin5i.com/

分析 init 进程的源代码[1]，了解 init 进程都做了哪些事情。

3.2　init 进程源码分析

本节将依次分析 init 进程源码中 main()函数内的代码。由于篇幅所限，无法把源代码一一列出，希望读者参考 init 进程的实际代码，一起研究学习。

Android 的 init 进程主要提供四大功能，如下所述。

init 进程不仅进行子进程的终止处理，当应用程序访问设备驱动时，还会生成设备节点文件，同时提供属性服务，保存系统运行所需的环境变量。此外，它还会分析 init.rc 启动脚本文件，并根据相关文件中包含的内容，执行相应的功能。图 3-3 列出了 init 的主要功能，接下来参考源码 main()函数，逐一分析代码。图 3-3 中同时标注出了相应源代码所在的位置或章节，在分析 init 进程时，请读者参考。

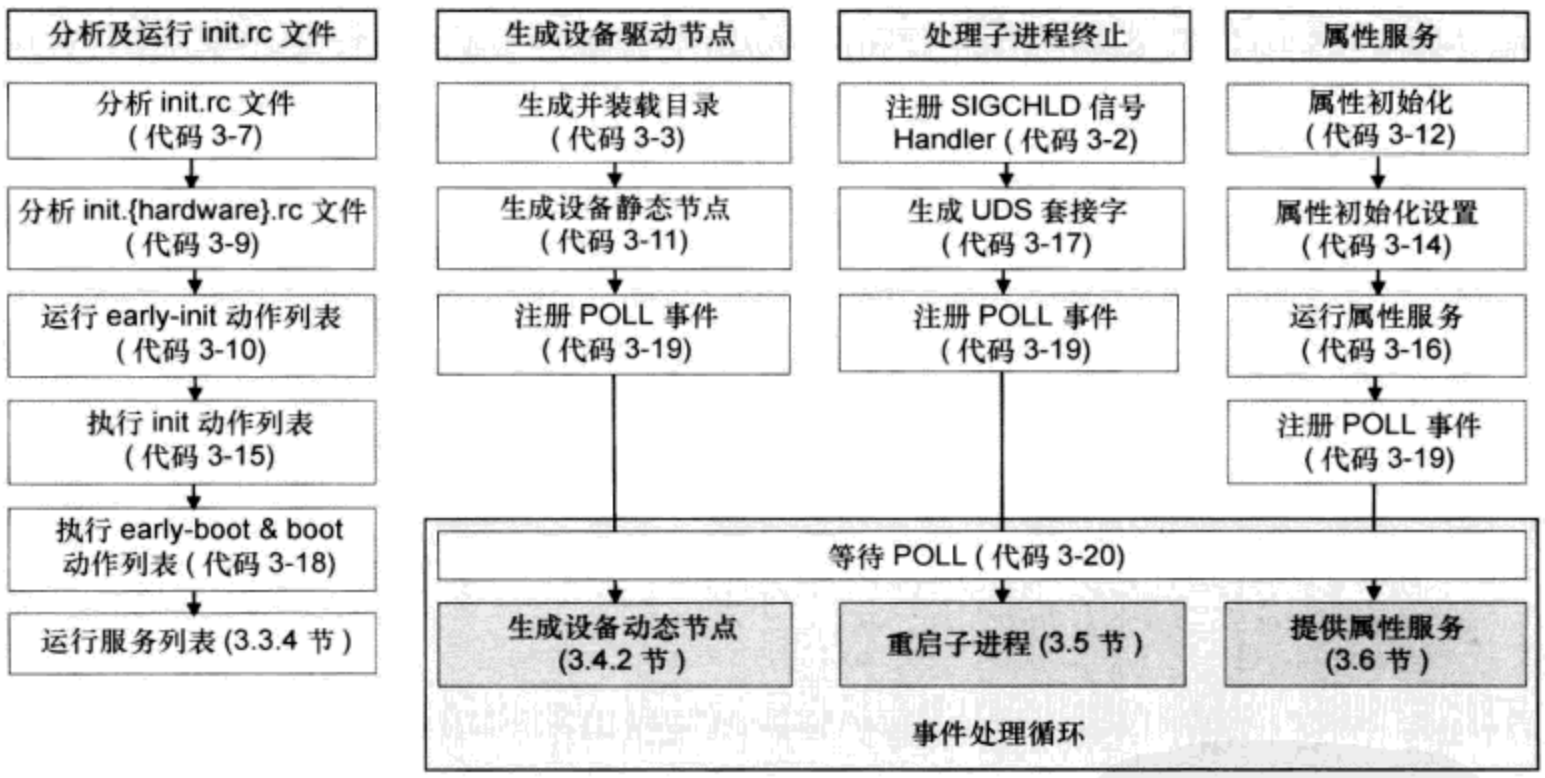


图 3-3 | init 进程主要功能及事件处理循环

```
static int signal_fd = -1;
static void sigchld_handler(int s)
{
   write(signal_fd, &s, 1);
}
int main(int argc, char **argv)
{
```

1　在 http://android.git.kernel.org/?p=platform/system/core.git;a=tree 的 init 目录下。

更多更新编程资源尽在 http://www.pin5i.com/

```
struct sigaction act;
act.sa_handler = sigchld_handler;
act.sa_flags = SA_NOCLDSTOP;
act.sa_mask = 0;
act.sa_restorer = NULL;
sigaction(SIGCHLD, &act, 0);
```

代码 3-2 | main() – SIGCHLD 注册信号处理器

Linux 进程通过互相发送接收消息来实现进程间的通信，这些消息被称为“信号”。每个进程在处理其他进程发送的信号时都要注册程序，此程序被称为信号处理器。当进程的运行状态改变或终止时，就会产生某种信号，init 进程是所有进程的父进程，当其子进程终止产生 SIGCHLD 信号时，init 进程需要调用信号安装函数 sigaction()[1]，并通过参数传递至 sigaction[2]结构体中，以完成信号处理器的安装。

Init 进程通过代码 3-2 注册与子进程相关的 SIGCHLD 信号处理器，并把 sigaction 结构体的 sa_flags 设置为 SA_NOCLDSTOP[3]，该值表示仅当进程终止时才接收 SIGCHLD 信号[4]。sigchld_handler 函数用于通知全局变量 signal_fd[5]，SIGCHLD 信号已发生。对于产生的信号的实际处理，在 init 进程的事件处理循环中进行。

init 进程在注册完信号处理器后，创建并挂载启动所需的文件目录，如下所示。

```
mkdir("/dev", 0755);
mkdir("/proc", 0755);
mkdir("/sys", 0755);

mount("tmpfs", "/dev", "tmpfs", 0, "mode=0755"); [2]
mkdir("/dev/pts", 0755);
mkdir("/dev/socket", 0755);
mount("devpts", "/dev/pts", "devpts", 0, NULL); [3]
mount("proc", "/proc", "proc", 0, NULL); [4]
mount("sysfs", "/sys", "sysfs", 0, NULL); [5]
```

代码 3-3 | main()目录生成与挂载

1 sigaction()是信号安装函数，由 signal()函数改进而来，带有 struct sigaction 参数，允许设置更多选项。

2 tmpfs 是一种虚拟内存的文件系统，典型的 tmpfs 文件系统完全驻留在 RAM 中，读写速度远快于闪存或硬盘文件系统。/dev 目录保存着硬件设备访问所需要的设备驱动程序。在 Android 中，将相关目录用作 tmpfs，可以大幅提升设备访问的速度。

3 devpts 是一种虚拟终端文件系统。

4 proc 是一种虚拟文件系统，只存在于内存中，而不占用外存空间。借助此文件系统，应用程序可以与内核内部数据结构进行交互。

5 sysfs 文件是一种特殊的文件系统，在 Linux Kernel 2.6 中引入，用于将系统中的设备组织成层次结构，并向用户模式程序提供详细的内核数据结构信息，将 proc、devfs、devpts 三种文件系统统一起来。

更多更新编程资源尽在 http://www.pin5i.com/

编译 Android 系统源码时，在生成的根文件系统中，并不存在/dev、/proc、/sys 这类目录，它们是系统运行时的目录，由 init 进程在运行中生成，当系统终止时，它们就会消失。

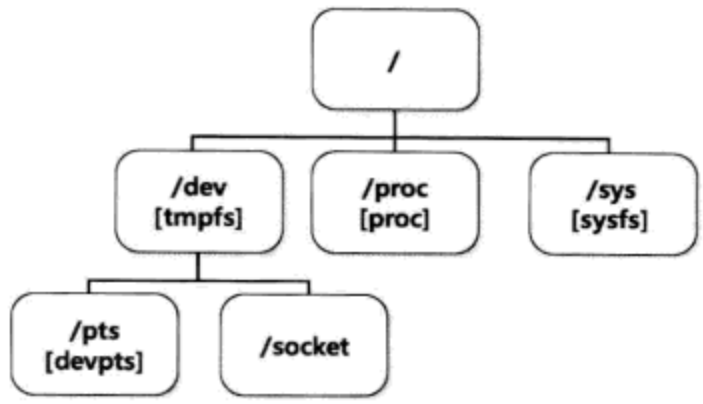


图 3-4 | init 进程生成的目录层次结构

init 进程通过执行代码 3-3 中代码，创建系统运行所需的目录，形成图 3-4 所示的层次目录结构。在图 3-4 中，[]内表示挂载在相应目录下的文件系统。

下列代码用于生成 log 设备，以便输出 init 进程的运行信息。

```
open_devnull_stdio();
log_init();

INFO("reading config file\n");
```

代码 3-4 | main()-初始化 log 输出设备

init 进程通过执行代码 3-3，生成/dev 目录，包含系统中使用的设备，而后调用 open_devnull_stdio()函数，创建运行日志输出设备。open_devnull_stdio()函数会在/dev 目录下生成_ _null_ _设备节点文件，并将标准输入、标准输出、标准错误输出[1]全部重定向到_ _null_ _设备中。

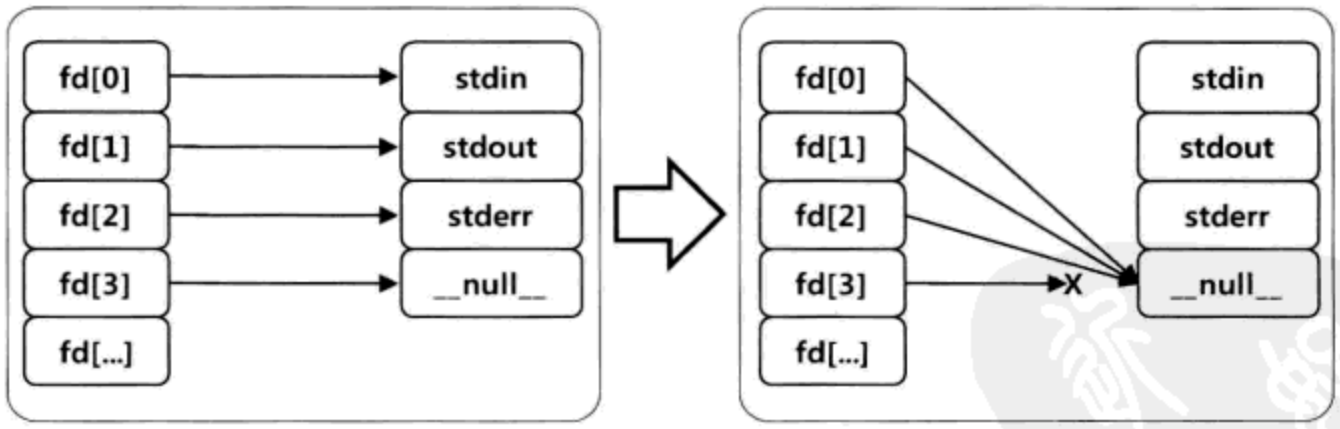


图 3-5 | 由 open_devnull_stdio()函数改变 log 信息输出设备

如图 3-5 所示，open_devnull_stdio()函数将与输入输出相关的文件全部重定向至_ _null_ _设备中。因此 init 进程调用执行 open_devnull_stdio()函数时，通过标准的输入输出无法输出信息。这是否意味着 init 进程输出的 log 信息无法查看呢？不是

1　在 Unix（包括 Linux）系统中，所有的输入、输出都被处理为文件，标准输入为 0，标准输出为 1，标准错误输出为 2。

更多更新编程资源尽在 http://www.pin5i.com/

的。init 进程通过 log_init()函数，提供输出 log 信息的设备。

```
void log_init(void)
{
   static const char *name = "/dev/__kmsg__";
   if (mknod(name, S_IFCHR | 0600, (1 << 8) | 11) == 0) {
      log_fd = open(name, O_WRONLY);
      fcntl(log_fd, F_SETFD, FD_CLOEXEC);
      unlink(name);
   }
}
```

代码 3-5 | 由 log_init()函数生成输出设备

init 进程通过调用 log_init()函数，生成“/dev/_ _kmsg_ _”设备节点文件。_ _kmsg_ _设备调用内核信息输出函数 printk()，init 进程即是通过该函数来输出 log 信息。

```
#define ERROR(x...)   log_write(3, "<3>init: " x)
#define NOTICE(x...)  log_write(5, "<5>init: " x)
#define INFO(x...)    log_write(6, "<6>init: " x)
```

代码 3-6 | 输出 init 进程的 log 信息

如代码 3-6 所示，init 进程通过_ _kmsg_ _设备定义用于输出信息的宏。关于宏输出信息，可以通过 dmesg 实用程序进行确认，dmesg 实用程序用于显示内核信息。

init 进程在生成输出设备后，便开始解析 init.rc 文件。init.rc 文件是在 init 启动后执行的启动脚本，文件中记录着 init 进程执行的功能。在 Linux 系统中，它被定义在根文件系统的“/etc/rc.d/”目录下，是启动时的可执行文件，在“/etc”目录下保存着设置环境变量的脚本。但在 Android 系统中，仅使用 init.rc 与 init.{hardware[1]}.rc 两个文件，用来定义与执行文件与环境变量。init.rc 文件在 Android 系统运行过程中用于通用的环境设置及与进程相关的定义，init.{hardware}.rc 用于定义 Android 在不同平台下的特定进程和环境设置等。

```
parse_config_file("/init.rc");
```

代码 3-7 | main()-解析 init.rc 文件

parse_config_file()函数用于分析*.rc 配置文件，带有一个参数，用来指定 init.rc 文件

1 每个 Android 编译平台，hardware 都不相同。比如，在三星的 SMDK 开发平台下，生成的是 init.smdk.rc。

更多更新编程资源尽在 http://www.pin5i.com/

的路径。执行 parse_config_file() 函数，读取并分析 init.rc 文件后，生成服务列表与动作列表[1]。动作列表与服务列表会以链表的形式注册到 service_list 与 action_list 中，service_list 与 action_list 是 init 进程中声明的全局结构体[2]。然后，初始化 QEMU 设备，设置模拟器环境。

```
qemu_init();
```

代码 3-8 | main-初始化 QEMU 设备

QEMU 模拟器允许 Android 应用开发者在缺少 Android 实际设备的情况下运行处于开发中的应用程序。QEMU 是面向 PC 的开源模拟器，能够模拟具有特定处理器的设备，此外还提供虚拟网络、视频设备等。Android 模拟器是虚拟的硬件平台，用来模拟 ARM 核的移动设备，基于 QEMU 开发，被命名为 Goldfish。运行在模拟器中的软件都可以执行 ARM 命令集，LCD、相机、SD 卡控制器等硬件设备都可以存在于 Goldfish 这种虚拟的硬件平台中。

如在代码 3-7 中分析 init.rc 文件一样，以下代码用于分析 init.{hardware}.rc 文件。

```
snprintf(tmp, sizeof(tmp), "/init.%s.rc", hardware);
parse_config_file(tmp);
```

代码 3-9|main()-分析 init.{hardware}.rc 文件

如代码 3-9 所示，init 进程通过 parse_config_file() 函数，依据 init.{hardware}.rc 文件生成服务列表与动作列表，这些列表会分别被添加到 init.rc 文件已生成的服务列表与动作列表中。

init 进程会依次执行“early-init，init，early-boot，boot”片段中的命令。如代码 3-10 所示，init 进程会执行动作列表 early-init 片段中的命令。

```
action_for_each_trigger("early-init", action_add_queue_tail);
drain_action_queue();
```

代码 3-10|main()-执行 early-init 动作

action_for_each_trigger() 函数会将第一个参数中的命令保存到运行队列 action_add_

1 在 init.rc 文件中，含 service 关键字的被称为服务列表，含其他关键字的则被称为动作列表。

2 相关内容将在 3.3 节“init.rc 文件分析与执行”中进行说明。

更多更新编程资源尽在 http://www.pin5i.com/

queue_tail 中，而后通过 drain_action_queue() 函数将运行队列中的命令逐一取出执行。

接下来，创建 init 进程中已经定义好的设备节点文件。

```
device_fd = device_init();
```

代码 3-11| main()-生成静态设备节点

如代码 3-11 所示，init 进程通过调用 device_init() 函数生成静态设备节点。关于生成节点文件的内容，在 3.4 节“生成设备节点文件”中将进行详细说明。

然后，开始初始化属性服务。

```
property_init();
```

代码 3-12| main()-初始化属性服务

在 Android 系统中，所有的进程共享系统设置值，为此提供了一个名称为属性的保存空间。init 进程调用代码 3-12 中的 property_init() 函数，在共享内存区域中，创建并初始化属性域[1]。而后通过执行中的进程所提供的 API，访问属性域中的设置值。但更改属性值仅能在 init 进程中进行。当修改属性值时，要预先向 init 进程提交值变更申请，然后 init 进程处理该申请，并修改属性值。

接下来，系统会将 Android 启动 Logo 显示在 LCD 屏幕上。

```
#define INIT_IMAGE_FILE "/initlogo.rle"
load_565rle_image(INIT_IMAGE_FILE);
```

代码 3-13 | main()-显示启动 Logo

在代码 3-13 中，load_565rle_image()函数将加载由参数传递过来的图像文件，而后将该文件显示在 LCD 屏幕上。如果想更改启动 Logo，只需修改 INIT_IMAGE_FILE 即可。由于函数只支持 rle565 格式[2]图像的显示，在更改图像时，要注意所选图像的格式。

```
if (!strcmp(bootmode,"factory"))
    property_set("ro.factorytest", "1");
else if (!strcmp(bootmode,"factory2"))
```

1 在 ASHMEM（Android Shared Memory）中生成。

2 rle565 格式的图像在 Android 编译后，可以转换成 out/host/linux-x86/bin/rgb2565-rle<img24bit.bmp> initlogo.rle 格式。

更多更新编程资源尽在 http://www.pin5i.com/


```
    property_set("ro.factorytest", "2");
else
    property_set("ro.factorytest", "0");

property_set("ro.serialno", serialno[0] ? serialno : "");
property_set("ro.bootmode", bootmode[0] ? bootmode : "unknown");
property_set("ro.baseband", baseband[0] ? baseband : "unknown");
property_set("ro.carrier", carrier[0] ? carrier : "unknown");
property_set("ro.bootloader", bootloader[0] ? bootloader : "unknown");
property_set("ro.hardware", hardware);
snprintf(tmp, PROP_VALUE_MAX, "%d", revision);
property_set("ro.revision", tmp);
```

代码 3-14 | main()-属性初始设置

如代码 3-14 所示，init 进程会通过 property_set()函数向属性域设置系统所需的一些初始值。分析由 bootloader 传递至内核中的启动选项，可以看到 ro.serailno、ro.bootmode 等属性值，如代码所示。这些设置的属性值由执行中的多种进程调用 property_get() API 来访问。

接下来，执行与动作列表的 init 区块相关的命令。

```
/* execute all the boot actions to get us started */
action_for_each_trigger("init", action_add_queue_tail);
drain_action_queue();
```

代码 3-15 | main()-执行 init 动作

然后，启动属性服务，它是 init 进程最主要的功能之一。

```
/* read any property files on system or data and
 * fire up the property service.  This must happen
 * after the ro.foo properties are set above so
 * that /data/local.prop cannot interfere with them.
 */
property_set_fd = start_property_service();
```

代码 3-16 | main()-启动属性服务

除了前面设置的属性外，start_property_service() 函数还会读取几个设置文件，并对属性进行初始化。在根文件系统的/data/property 目录下，保存着进程生成或修改的属性值。如前所述，属性值的更改仅能在 init 进程中进行，即一个进程若想修改属性值，必须向 init 进程提交申请，为此 init 进程生成“/dev/socket/property_service”套接字，以接收其他进程提交的申请。

更多更新编程资源尽在 http://www.pin5i.com/

接下来，创建套接字，以便 init 进程在收到子进程终止的 SIGCHLD 信号时调用相应的 handler。

```
/* create a signalling mechanism for the sigchld handler */
    if (socketpair(AF_UNIX, SOCK_STREAM, 0, s) == 0) {
        signal_fd = s[0];
        signal_recv_fd = s[1];
        fcntl(s[0], F_SETFD, FD_CLOEXEC);
        fcntl(s[0], F_SETFL, O_NONBLOCK);
        fcntl(s[1], F_SETFD, FD_CLOEXEC);
        fcntl(s[1], F_SETFL, O_NONBLOCK);
    }
```

代码 3-17 | main()-收到 SIGCHLD 信号时，创建 UDS 套接字

init 进程另外定义了 handler，用于处理子进程的终止，当发生 SIGCHLD 信号时，为了调用相关的 handler，init 进程会通过套接字连接 SIGCHLD 信号的 handler。socketpair()函数创建一对已经连接的套接字，即代码 3-17 中的 s[2]数组。在代码 3-2 的 SIGCHLD 信号 handler 中，可以看到 signal_fd 被设置为 1，此时通过套接字对（s）与 signal_fd 相连接的 signal_recv_fd 也被设置为 1。事件处理 handler 会监视 signal_recv_fd 的值，当其值为 1 时，init 进程就会调用子进程停止处理 handler。更详细的内容，将在 3.5 节“进程的终止与重启”中进行讲解。

请看代码 3-18，程序开始执行与 Action List 的 early-boot、boot、property 相关的命令。

```
/* execute all the boot actions to get us started */
action_for_each_trigger("early-boot", action_add_queue_tail);
action_for_each_trigger("boot", action_add_queue_tail);
drain_action_queue();

/* run all property triggers based on current state of the properties */
queue_all_property_triggers();
drain_action_queue();
```

代码 3-18 | main()- 执行 early-boot、boot、property 区段的 action

在 init.rc 文件的 boot 区段有一条 class_start 命令，用来逐一执行存在于服务列表（Service List）中的进程列表。关于 class_start 命令的相关内容，将在 3.3.4 节“Aciton List 与 Service List 的运行”中详细进行说明。

接下来，设置事件处理循环的监视事件。注册在 POLL 中的文件描述符会在 poll()函数中等待事件，若事件发生，则从 poll()函数中跳出并处理事件。

更多更新编程资源尽在 http://www.pin5i.com/


```
// device_fd = device_init();  - 代码 3-11
// property_set_fd = start_property_service(); - 代码 3-16
// signal_recv_fd = s[1];    - 代码3-17

    ufds[0].fd = device_fd;                    ←❶
    ufds[0].events = POLLIN;
    ufds[1].fd = property_set_fd;              ←❷
    ufds[1].events = POLLIN;
    ufds[2].fd = signal_recv_fd;               ←❸
    ufds[2].events = POLLIN;
    fd_count = 3;
```

代码 3-19 | main()-注册 POLL 事件

代码 3-19 会注册 init 进程监视的文件描述符。即将注册在 POLL 中的文件描述符❶既是设备节点生成事件处理文件描述符，❷也是属性服务申请事件处理文件描述符，❸还是 SIGCHLD 信号处理文件描述符。上述文件描述符分别被注册在 pollfd 结构体的变量 ufds 数组中，在事件处理循环中，它被传递给 poll() 函数的参数，监视相关文件描述符的事件。pollfd 结构体的 fd 是所要监视的文件描述符的号码，events 是要监视的事件。在把文件描述符注册至 POLL 之后，如下事件处理循环将通过 poll() 函数监视事件。

```
for(;;) {
    drain_action_queue();                      ←❶
    restart_processes();                       ←❷
    nr = poll(ufds, fd_count, timeout);        ←❸
    if (ufds[2].revents == POLLIN) {           ←❹
        /* we got a SIGCHLD - reap and restart as needed */
        read(signal_recv_fd, tmp, sizeof(tmp));
        while (!wait_for_one_process(0))    ;
        continue;
    }

    if (ufds[0].revents == POLLIN)
        handle_device_fd(device_fd);                    ←❺
    if (ufds[1].revents == POLLIN)
        handle_property_set_fd(property_set_fd);        ←❻
}
```

代码 3-20 | main()-事件处理循环

代码 3-20 是 init 进程的事件处理循环。

❶ 在确认事件发生前，先要在 action list 的命令中确认是否有尚未执行的命令，并执行之。起初，在事件处理循环中，action list 与 service list 未含有需要执行的事情，但是在处理过注册的事件之后，init 进程要做的事情会

更多更新编程资源尽在 http://www.pin5i.com/

重新注册到 action list 与 service list 中。代码❶是事件处理循环执行一次后的结果。

❷ 当子进程终止退出时，此命令用于重启或终止子进程。关于 restart_processes()函数的详细介绍，将在 3.5.1 节“分析进程重启代码”中进行。

❸ poll()函数用于等待代码 3-19 中已注册的文件描述符发生的事件。若事件发生，事件信息会保存在 pollfd 结构体的 udfs.revents 变量中。当 poll()函数返回后，可以在 ufds 数组的 revents 中查看哪些事件已经发生。

❹ 当子进程终止时，会产生 SIGCHLD 信号，POLLIN 事件会被注册至 udfs[2].revents 中。关于事件处理的内容，将在 3.5.1 节“分析进程重启代码”中详细说明。

❺ 在 Android 系统运行过程中，插入热的拔插设备时，将生成设备节点文件。与这部分相关的内容，将在 3.4.2 节“热拔插设备感知”中讲解。

❻ 处理属性变更请求。相关内容将在 3.6 节中详细地进行说明。

3.3 init.rc 脚本文件分析与执行

本节主要讲解两个方面的内容，一是 init.rc 启动脚本，Android 启动时，此脚本文件用来设置系统环境，记录待执行的进程；二是 action list 与 service list 相关的内容，它们是由 init 进程根据 init.rc 生成的。此外，系统还存在着 init.{hardware}.rc 文件，其作用类似于 init.rc 文件，init 进程按 init.rc 文件的处理方式处理 init.{hardware}.rc 文件。init.{hardware}.rc 文件所用的语法与 init.rc 相同，因此这里仅以 init.rc 文件为例进行分析说明。

init.rc 文件不同于 init 进程，init 进程仅当编译完 Android 后才会生成，而 init.rc 文件存在于 Android 平台源码[1]中。init.rc 文件的结构大致如图 3-6 所示。

图 3-6 | init.rc 文件结构

图 3-6 中 init.rc 文件大致分为两大部分，一部分是以“on”关键字开头的动作列表（action list），另一部分是以“service”关键字开头的服务列表（service list）。借助系统环境变量或 Linux 命令，动作列表用于创建所需目

1 init.rc 文件在 http://android.git.kernel.org/?p=platform/system/core.git;a=tree 的 rootdir 目录下。

更多更新编程资源尽在 http://www.pin5i.com/

录，以及为某些特定文件指定权限。而服务列表用来记录初始化程序需要启动的一些程序。动作列表在 3.3.1 节中介绍，服务列表在 3.3.2 节“服务列表”中详细介绍。

3.3.1　动作列表（Action List）

在动作列表的“on init”段落中，主要设置环境变量，生成系统运行所需的文件或目录，修改相应的权限，并挂载与系统运行相关的目录。

代码 3-21 是 init.rc 文件中“on init”段落部分。在“on init”段落的环境变量设置部分，主要设置运行根文件系统命令的目录，以及程序编译时需要的库目录。

```
on init
    loglevel 3
    # setup the global environment                          // 设置环境变量
    export PATH /sbin:/system/sbin:/system/bin:/system/xbin
    export LD_LIBRARY_PATH /system/lib
    export ANDROID_BOOTLOGO 1
    export ANDROID_ROOT /system
    # mount mtd partitions
    # Mount /system rw first to give the filesystem a chance to save a check point
    mount yaffs2 mtd@system /system                         // 挂载根文件系统
    mount yaffs2 mtd@system /system ro remount

    # We chown/chmod /data again so because mount is run as root + defaults
    mount yaffs2 mtd@userdata /data nosuid nodev
    chown system system /data
    chmod 0771 /data…
```

代码 3-21 | init.rc-on init 段落

在“on init”根文件系统挂载部分，主要挂载/system 与/data 两个目录。两个目录挂载完毕后，Android 的根文件系统就准备好了。

如图 3-7 所示，根文件系统大致可分为 Shell 实用程序、system 目录（提供库与基本应用）、data 目录（保存用户应用、照片等用户数据）这几部分。其中，system/bin 目录用来保存运行在 Shell 中的实用程序，类似于 Linux 中常用的 busybox[1]命令工具。在 Android 平台中，未使用 busybox 工具集，而是使用了 toolbox 工具集。

Android 是针对手机开发的 OS，必须充分考虑手机的硬件状态。

手机领域有多种不同的内存设备，其中 NAND 闪存设备以其功耗低、重量轻、性

1　虽然 Android 开源，但在嵌入实际机器中时，为了保护开发企业的 S/W 知识产权，未采用遵循开放源码的许可政策。因此 Android 系统未使用遵循 GPL 许可的 busybox，而使用了 toolbox。

更多更新编程资源尽在 http://www.pin5i.com/

能佳等优良特性，受到绝大多数手机厂商的青睐。NAND 闪存采用 yaffs2 文件系统，启动时要将其挂载至/system 与/data 目录下。若开发者想变更要挂载的目录或文件系统，则必须修改代码 3-21 中的根文件系统挂载部分。

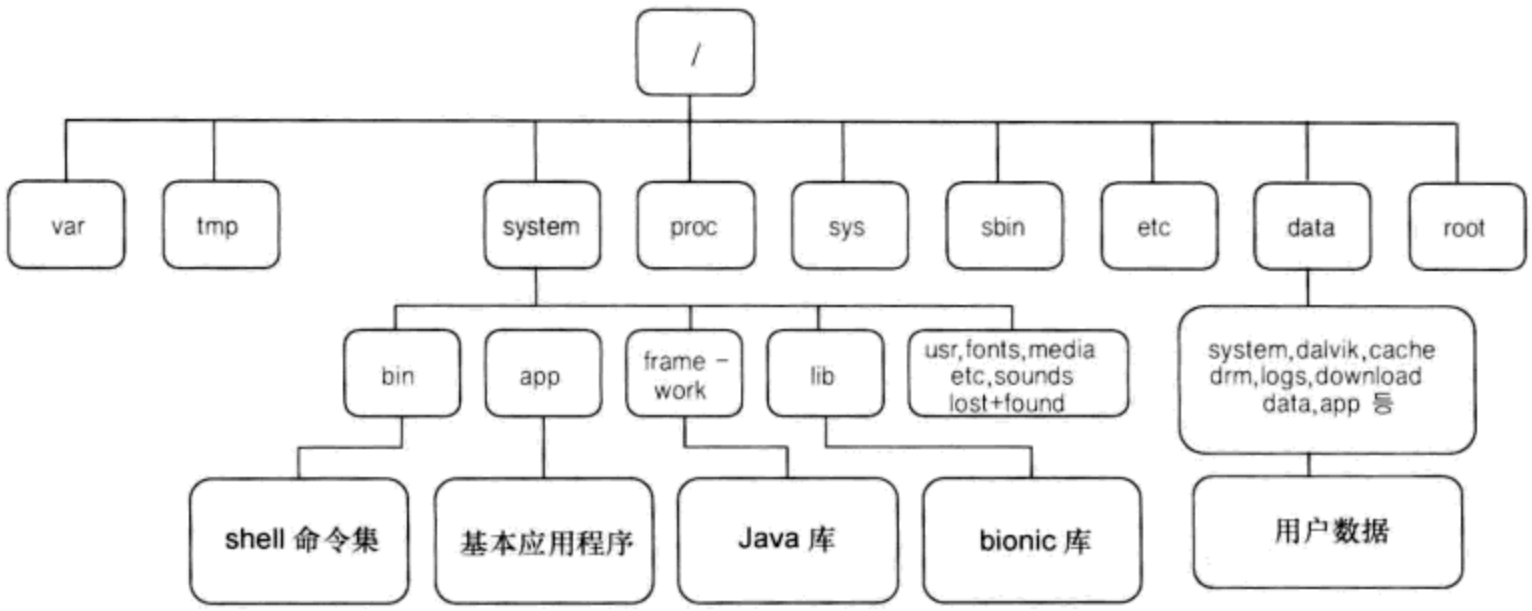


图 3-7 | Android 根文件系统结构

接下来，查看“on boot”段落部分，该部分主要用于设置应用程序终止条件、应用程序驱动目录及文件权限等。

```
on boot
# Define the oom_adj values for the classes of processes that can be   // 应用程序
# killed by the kernel. These are used in ActivityManagerService.     // 设置终止条件
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
.
.
class_start default                                                    // 运行服务
```

代码 3-22 | init.rc-on boot 段落

代码 3-22 是 init.rc 文件的 on boot 段落，在此段落的设置应用程序终止条件部分中，分别为各应用程序指定 OOM（Out of Memory）调整值（Adjustment Value：ADJ）。Android 中的众多应用程序根据驱动状态的不同划分成几个组，如图 3-8 所示。

OOM 用来监视内核分配给应用程序的内存，当内存不足时，应用程序会被终止执

更多更新编程资源尽在 http://www.pin5i.com/

行。OOM 值用于区分内存不足即将终止的应用程序，其值越大，代表应用程序的终止优先权越高。

应用程序组	ADJ 值	说明
FOREGROUND_APP	0	前台程序
VISIBLE_APP	1	仅显示非运行程序
SECONDARY_SERVER	2	服务 Daemon
HOME_APP	4	注册到开始画面的程序
HIDDEN_APP_MIN	7	最小化程序
CONTENT_PROVIDER	14	内容提供者
EMPTY_APP	15	

图 3-8 | 应用程序组划分

```
# adbd on at boot in emulator
on property:ro.kernel.qemu=1
    start adbd

on property:persist.service.adb.enable=1
    start adbd

on property:persist.service.adb.enable=0
    stop adbd
```

代码 3-23 | init.rc 文件的 on property 设置

在“on property:<name>=<value>”段落中，记录属性值改变时执行的命令，在默认的 init.rc 初始化脚本中，它还记录 adbd 服务启动、终止的条件。关于属性的内容，将在 3.6 节“属性服务”中详细说明。

3.3.2 服务列表（Service List）

在 init.rc 脚本文件中，“service”段落用来记录 init 进程启动的进程。由 init 进程启动的子进程或是一次性程序，或是运行在后台的与应用程序、系统管理相关的 Daemon 进程。

```
## Daemon processes to be run by init.
##
service console /system/bin/sh
    console

# adbd is controlled by the persist.service.adb.enable system property
```

更多更新编程资源尽在 http://www.pin5i.com/

```
service adbd /sbin/adbd
    disabled

service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media
service vold /system/bin/vold
    socket vold stream 0660 root mount
```

代码 3-24 | init.rc-service 段落

代码 3-24 是 init.rc 文件 service 段落部分，在 service 段落中，关键字 service 后的第一个字符串表示服务的名称，第二个字符串表示服务的路径。第二行是服务的附加内容，用于配合服务使用，主要包含运行权限、条件以及重启等相关选项。service 段落中的服务全部注册在服务列表中，init 进程从该列表中依次取出相应服务，并启动它。关于运行服务列表中服务的内容，将在本书第 55 页的 3.3.4 节“动作列表与服务列表的运行”中进行说明。

3.3.3 init.rc 文件分析函数

在 3.2 节“init 进程源码分析”的代码 3-7 中，有一个名为“parse_config_file()”的函数调用，该函数用来分析 init.rc 脚本文件。parse_config_file() 函数带有一个参数，用来指定待分析文件的路径。执行时，parse_config_file() 函数先从参数指定的位置读取文件，生成连续的字符串，而后再分析各字符串。parse_config_file() 函数主要调用了两个函数，一个是 read_file()函数，用来读取文件；另一个是 parse_config() 函数，用来分析读入的字符串。

```
int parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    parse_config(fn, data);
}
```

代码 3-25 | parse_config_file() 函数

read_file()函数用于将指定的文件读取到内存，保存为字符串形式，并返回字符串在内存中的初始地址。parse_config() 函数分析 read_file() 函数返回的字符串，并生成动作列表（Action List）与服务列表（Service List）。代码 3-26 是 parse_config() 函数，该函数会逐行分析参数传递过来的字符串。

更多更新编程资源尽在 http://www.pin5i.com/


```
static void parse_config(const char *fn, char *s)
{
    for (;;) {
        switch (next_token(&state)) {                    ← ❶
            case T_NEWLINE:
            if (nargs) {
                int kw = lookup_keyword(args[0]);        ← ❷
                if (kw_is(kw, SECTION)) {                ← ❸
                    parse_new_section(&state, kw, nargs, args);
                }
            }
}
```

代码 3-26 | parse_config()函数

❶ 首先 next_token()函数以行为单位分割参数传递过来的字符串，而后调用 lookup_keyword()函数。

❷ lookup_keyword()函数用于返回 init.rc 脚本中每行首个单词在 keyword_list 结构体数组中的数组编号。keyword_list 结构体数组由代码 3-27 中的 KEYWORD 列表构成，每个列表由代码 3-28 中的 KEYWORD 宏生成。

```
//              关键字、组、参数个数、映射函数
KEYWORD(capability,  OPTION,  0, 0)
KEYWORD(class,       OPTION,  0, 0)
.
.
KEYWORD(mkdir,       COMMAND, 1, do_mkdir)
KEYWORD(mount,      COMMAND, 3, do_mount)
KEYWORD(on,          SECTION, 0, 0)
KEYWORD(oneshot,     OPTION,  0, 0)
KEYWORD(onrestart,   OPTION,  0, 0)
KEYWORD(restart,     COMMAND, 1, do_restart)
KEYWORD(service,     SECTION, 0, 0)
.
.
KEYWORD(chown,      COMMAND, 2, do_chown)
KEYWORD(chmod,      COMMAND, 2, do_chmod)
KEYWORD(loglevel,   COMMAND, 1, do_loglevel)
KEYWORD(device,     COMMAND, 4, do_device)
```

代码 3-27 | KEYWORD 列表

KEYWORD 宏分别定义在 parse.c 与 keyword.h[1]两个文件中，它们的作用各不相同。定义在 parse.c 中的 KEYWORD 宏用来返回 KEYWORD 列表，返回值类型为

1　在 http://android.git.kernel.org/?p=platform/system/core.git;a=tree 的 init 目录下。

更多更新编程资源尽在 http://www.pin5i.com/

keyword_list 结构体数组，而定义在 keyword.h 中的 KEYWORD 宏用来为 KEYWORD 列表分配编号，编号从 1 开始。

```
#define SECTION 0x01
#define COMMAND 0x02
#define OPTION  0x04

#define KEYWORD(symbol, flags, nargs, func) \
    [ K_##symbol ] = { #symbol, func, nargs + 1, flags, },

struct {
    const char *name;
    int (*func)(int nargs, char **args);
    unsigned char nargs;
    unsigned char flags;
} keyword_info[KEYWORD_COUNT] = {
    [ K_UNKNOWN ] = { "unknown", 0, 0, 0 },
    #include "keywords.h"
};
```

代码 3-28 | 定义在 parse.c 中的 KEYWORD 宏

代码 3-28 中的 KEYWORD 宏将根据代码 3-27 中注释的“组”字段对 init.rc 中的关键字进行分组。COMMAND 分组表示 init 进程执行的命令，为处理相关命令，需将关键字与相关函数对应起来。例如，mkdir 关键字与 do_mkdir 函数相对应，do_mkdir 用来创建目录。

SECTION 分组用于区分动作列表与服务列表，其中动作列表以“on”关键字开头，而服务列表以“service”关键字开头。当执行命令或运行服务列表中的程序时，OPTION 分组用来指定运行条件、选项。

KEYWORD 列表由上面的 KEYWORD 宏转换为 keyword_info 结构体数组形式的列表，其形式为：[“列表编号”]={“关键字”，“组”，“参数个数”，“映射函数”}，其中“列表编号”由代码 3-29 中的 KEYWORD 宏创建。而定义在 keyword.h 中的 KEYWORD 列表被转换为“K_关键字”形式的枚举数据，编号从 1 开始。

```
#define __MAKE_KEYWORD_ENUM__
#define KEYWORD(symbol, flags, nargs, func) K_##symbol,
enum {
   K_UNKNOWN,                                    // 0
#endif
   KEYWORD(capability,  OPTION,  0, 0)           // 1
   KEYWORD(class,       OPTION,  0, 0)           // 2
```

代码 3-29 | 定义在 keyword.h 中的 KEYWORD 宏

更多更新编程资源尽在 http://www.pin5i.com/

代码 3-30 是 lookup_keyword() 函数，该函数返回与参数传递过来的关键字相对应的 KEYWORD 列表的编号。

```
int lookup_keyword(const char *s)
{
    switch (*s++) {
    case 'c':
        if (!strcmp(s, "capability")) return K_capability;
     .
     .
    case 'm':
        if (!strcmp(s, "mkdir")) return K_mkdir;
        if (!strcmp(s, "mount")) return K_mount;
        break;
    case 'o':
        if (!strcmp(s, "on")) return K_on;
        if (!strcmp(s, "oneshot")) return K_oneshot;
        if (!strcmp(s, "onrestart")) return K_onrestart;
        break;
    case 's':
        if (!strcmp(s, "service")) return K_service;
    .
    .
}
```

代码 3-30 | lookup_keyword() 函数

用户若想向 init.rc 中添加新命令，只需重新向上述 KEYWORD 列表中定义新命令，并定义与之对应的函数即可。

❸ 查找带有 KEYWORD 列表编号的 keyword_list 结构体数组，KEYWORD 列表编号由 lookup_keyword()函数返回。数组内的 flag 用于判断是否为 SECTION。带有 SECTION flag 的命令仅有“on”与“service”两个关键字，通过 kw_is() 宏筛选出“on”段落的动作列表与“service”段落的服务列表，并调用 parse_new_section()函数分析其含义。

parse_new_section() 函数将 kw_is() 宏筛选出的命令，分别注册到动作列表或服务列表中。比如，在 init.rc 文件中，“on”段落的命令都会被注册到动作列表中，直至碰到“service”段落，而“service”段落的程序会被注册到服务列表中。而后 parse_new_section() 函数会将服务列表与动作列表分别保存到全局变量 service_list 与 action_list 中。在 parse_new_section() 函数执行完毕后，形成如图 3-9 所示的动作列表与服务列表。

关于动作列表与服务列表如何运行，将在下一节中进行讲解。

更多更新编程资源尽在 http://www.pin5i.com/

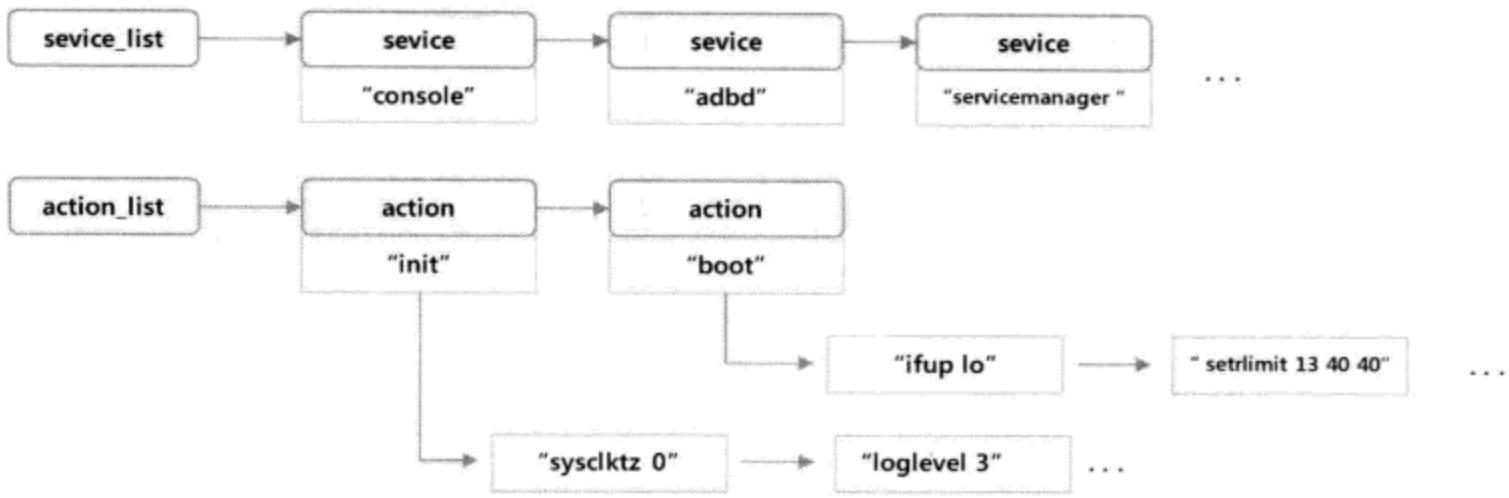


图 3-9 | 服务列表与动作列表

3.3.4 动作列表与服务列表的运行

动作列表与服务列表通过代码 3-10 中的 drain_action_queue() 函数来运行。

```
static void drain_action_queue(void)
{
    struct listnode *node;
    struct command *cmd;
    struct action *act;
    int ret;

    while ((act = action_remove_queue_head())) {                    ❶
        INFO("processing action %p (%s)\n", act, act->name);
        list_for_each(node, &act->commands) {
            cmd = node_to_item(node, struct command, clist);        ❷
            ret = cmd->func(cmd->nargs, cmd->args);                 ❸
        }
    }
}
```

代码 3-31 | 由 drain_action_queue() 函数运行动作列表

代码 3-31 是 drain_action_queue()函数。

❶ action_remove_queue_head()函数用来获取全局链表 action_queue 的 head，action_queue 保存有由待执行的命令构成的动作列表。

❷ 从 action_queue 中取出动作列表，并转换成 command 结构体。

❸ command 结构体的 func 变量指定与动作列表中的命令相对应的函数，即各命令的映射函数，这点在 3.3.3 节“init.rc 文件分析函数”中已经进行了说明。比如，init.rc 文件中的“mkdir /system”将调用 do_mkdir() 函数，创建“/system”目录。

更多更新编程资源尽在 http://www.pin5i.com/

上面我们讲解了如何执行动作列表中的命令，那服务列表中的进程又如何运行呢？在代码 3-22 的“on boot”段落中，最后一行命令为 class_start，init 进程通过该命令运行“service”段落中的所有程序。在 class_start 命令中，相应的执行函数为 do_class_start()[1]。

```
static void service_start_if_not_disabled(struct service *svc)
{
    if (!(svc->flags & SVC_DISABLED)) {
          service_start(svc);
    }
}

int do_class_start(int nargs, char **args)
{
        /* Starting a class does not start services
         * which are explicitly disabled.  They must
         * be started individually.
         */
    service_for_each_class(args[1], service_start_if_not_disabled);
    return 0;
}
```

代码 3-32 | do_class_start() 函数

代码 3-32 是 class_start 命令的执行函数 do_class_start()，它通过调用 service_for_each_class() 函数，运行全局声明的 service_list 结构体中的程序，即调用 service_start() 函数，运行服务列表中的程序，service_start() 函数则通过 execve() 系统调用来运行服务列表内的程序。init 进程即是通过这一过程来运行定义在 init.rc 中的程序的。

init.rc 初始化脚本文件使用 Android 初始化语言（AIL，Android Init Language）编程而成。关于 AIL 的介绍，笔者简单地整理了一下，仅供读者参考。更详细的内容，请读者查看 init 文件夹下的 readme.txt 文件。

TIP Android Init Language（Android 初始化语言）

在 Android 平台下，init 进程通过读取 init.rc 文件来设置环境变量，运行相应程序。init.rc 文件使用 Android Init Language 编写而成。下面简单地介绍一下 Android Init Language 这种语言。

AIL 包含四种类型的声明，如下：

- Actions（动作）
- Commands（命令）

1　在 http://android.git.kernel.org/?p=platform/system/core.git;a=tree 的 init 目录下的 builtins.c 文件中。

更多更新编程资源尽在 http://www.pin5i.com/

- Services（服务）
- Options（选项）

其语法规则如下：

- 初始化语言以行为单位，由以空格间隔的语言符号组成。C 风格的反斜杠转义符可以用来在语言符号中插入空格。双引号也可以用来防止文本被空格分成多个语言符号。当反斜杠在行末时，作为折行符。
- 以#开始的行为注释行。
- Action 与 Service 隐含声明一个新的段落，所有该段落下的 Command 与 Option 的声明皆属于该段落。
- Action 与 Service 的名称是唯一的。在它们之后声明相同命名的类将被视为无效。

Actions

Actions 是一系列命令的命名，通过一个触发器（trigger）来决定何时执行。当以 on 为开头的事件发生时，若该 action 尚未添加至待执行的队列中，则将其加入到队列最后，队列中的 action 依次执行，action 中的命令也依次执行。Actions 的表现形式为：

Services

Services 由 init 进程启动，当它们退出时重启（可选）。Services 表现形式为：

Options

Options 是 Services 的修饰，它们影响 init 何时、如何运行 service。

disabled	init 进程启动的所有进程被包含在名称为“类”的运行组中。进程所属的类被启动时，若指定进程为 distabled，该进程将不会被执行，只有按照名称明确指定后才可以启动。
socket <type> <name> <perm> [<user> [<group>]]	创建一个名为“/dev/socket/<name>”的 Unix domain socket，并传递它的 fd 到已启动的进程。
user <username>	在执行服务前改变用户名。若未设定，则默认为 root。
group <groupname> [<groupname>]*	执行服务前改变组，默认组为 root。
oneshot	服务退出后不再重启。
class <name>	在执行服务前为其指定所属的类名。当一个类启动或退出时，其包含的所有服务可以一同启动或停止。若未指定，服务默认为“default”类。
onrestart	当服务重启时执行一个命令。

更多更新编程资源尽在 http://www.pin5i.com/

Triggers

Triggers（触发器）是一个字符串，与 on 关键字配用，用来匹配某种类型的事件并执行一个 action。

boot	当 init 开始执行时的触发器。
<name>=<value>	当改变属性值时触发。
device-added-<path> device-removed-<path>	添加或删除设备时触发。
service-exited-<name>	当指定的服务停止时触发。

Commands

详细内容，请参看下面表格。

exec <path> [<argument>]*	Fork 并执行一个程序，这将阻塞 init 进程直到程序执行完毕。
export <name> <name>	设定环境变量。当这个命令执行后，所有进程都可以取得设定的环境变量。
ifup <interface>	使网络接口联机。
import <filename>	读取其他设置文件，扩展当前配置文件。
hostname <name>	设置主机名。
class_start <serviceclass>	启动该类下的所有服务。
domainname <name>	设置域名。
insmod <path>	加载指定模块。
loglevel <level>	设置 log 级别。
mkdir <path>	在指定位置创建目录。
mount <type> <device> <dir> [<mountoption>]*	挂载文件系统到指定的目录下。
setprop <name> <value>	设置属性值。
setrlimit <resource> <cur> <max>	设置进程占用的资源限制。资源包括用户创建的文件大小、可打开的文件个数等。详细内容，请参考 setlimit 的 manpage。
start <service>	启动指定的服务，若已启动，则忽略。
stop <service>	停止正在运行的服务。
symlink <target> <path>	创建一个<path>的符号链接到<target>。
write <path> <string> [<string>]*	打开指定的文件，并写入字符串。

更详细的介绍，请参考 http://pdk.android.com/online-pdk/guide/bring_up.html 页面。

更多更新编程资源尽在 http://www.pin5i.com/

3.4 创建设备节点文件

与 Linux 相同，Android 中的应用程序通过设备驱动来访问硬件设备。设备节点文件是设备驱动的逻辑文件，应用程序使用设备节点文件来访问设备驱动程序。在 Linux 中，提供 mknod 实用程序来创建设备节点文件，但出于安全考虑，Android 未提供类似 mknod 的实用程序。

若想在 Android 中创建设备节点文件，需要遵从 Android 提供的创建设备节点的方法。下面我们将学习在 Android 中如何创建设备节点。

3.4.1 创建静态设备节点

在 Linux 中，运行所需要的设备节点文件都被事先定义在“/dev”目录下。应用程序无需经过其他步骤，通过事先定义好的设备节点文件即可访问设备驱动程序。但在 Android 根文件系统的映像中不存在“/dev”目录。因此在系统运行中，需要有一个进程来创建设备节点文件，担当此任务的即是 init 进程。

init 进程通过两种方式创建设备节点文件。第一种，以预先定义的设备信息为基础，当 init 进程被启动运行时，统一创建设备节点文件；第二种，在系统运行中，当有设备插入 USB 端口时，init 进程就会接收到这一事件，为插入的设备动态创建设备节点文件。

第一种方法是连接已定义的设备的方法，称为“冷拔插”（Cold Plug），而第二种方法是在系统运行的状态下连接设备，称为“热拔插”（Hot Plug）。关于“热拔插”（Hot Plug），将在下一节讲解，本节讲解与“冷拔插”（Cold Plug）相关的内容。

在 Linux 内核 2.6 版本之前，用户必须直接创建设备节点文件。创建时，必须保证设备文件的主次设备号不发生重叠，再通过 mknod 实用程序进行创建。这样做的缺点显而易见，用户必须一一记住各个设备的主设备号与次设备号，并且还要避免设备号之间发生冲突，实际操作起来相当麻烦。为了弥补这一不足，从内核 2.6x 开始引入 udev（userspace device）实用程序。udev 以守护进程的形式运行，当设备驱动被加载时，它会掌握主设备号、次设备号，以及设备类型，而后在“/dev”目录下自动创建设备节点文件。

图 3-10 展现了从加载设备驱动到 udev 创建设备节点文件的整个过程。在系统运行中，若某个设备被插入，内核就会加载与该设备相关的驱动程序。而后驱动程序会调用启动函数 probe()[1]，将主设备号、次设备号、设备类型保存到“/sys”文件系统中。

---

1 probe()函数被定义在设备驱动程序中，由内核自动调用，用来初始化设备。

更多更新编程资源尽在 http://www.pin5i.com/

然后，发出 uevent，并传递给 udev 守护进程。

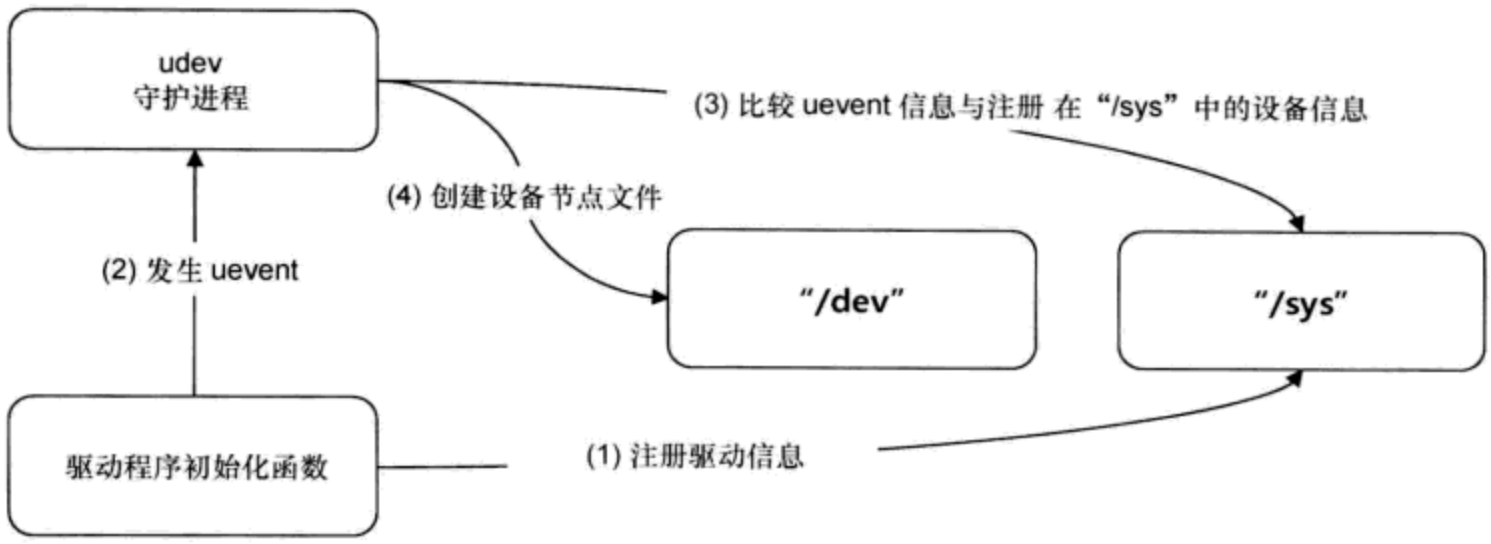


图 3-10 | udev 守护进程创建设备节点文件

TIP　uevent 是什么？

uevent 是内核向用户空间进程传递信息的信号系统，即在添加或删除设备时，内核使用 uevent 将设备信息传递到用户空间。uevent 包含设备名称、类别、主设备号、次设备号、设备节点文件创建的目录等信息，并将这些信息传递给 udev 守护进程。udev 守护进程监听 uevent，当 uevent 发生时，它接收并读取其中的设备信息，而后创建设备节点文件。

udev 以守护进程的形式运行，通过监听分析内核发出的 uevent，查看注册在/sys 目录下的设备信息，而后在/dev 目录相应位置上创建设备节点文件。系统内核启动后，udev 进程运行在用户空间内，它无法处理内核启动过程中发生的 uevent。虽然内核空间内的设备驱动程序可以正常运行，但由于未创建访问设备驱动所需的设备节点文件，将会出现应用程序无法使用相关设备的问题。

Linux 系统中，在 udev 守护进程运行前，通过提供与加载的设备驱动程序冷拔插机制，来解决设备节点文件没被创建的问题。当插入设备时，热拔插机制会立即进行处理，而冷拔插与热拔插有着不同的处理机制。当内核启动后，冷拔插机制启动 udev 守护进程，从/sys 目录下读取事先注册好的设备信息，而后引发与各设备相对应的 uevent，创建设备节点文件。Android 也采用这种处理方式来创建设备节点文件，不同的是使用 init 进程来扮演 udev 守护进程的角色。

在本书的第 7 章，我们将讲解与 Android Binder IPC 相关的内容，其中 Binder 驱动程序即是采用热拔插方式创建设备节点的。Binder 驱动程序是一个虚拟的设备，不存在物理硬件，它被用在进程间提供 RPC（Remote Procedure Call）。一个应用程序若想使用 Binder，需要通过“/dev/binder”设备节点来访问 Binder 驱动程序。

当系统内核启动时，Binder 驱动程序在初始化函数中调用 misc_register() 函数。

更多更新编程资源尽在 http://www.pin5i.com/

misc_register()函数会将创建设备节点文件所需的信息保存到/sys 目录下，如图 3-11 步骤（1）所示。

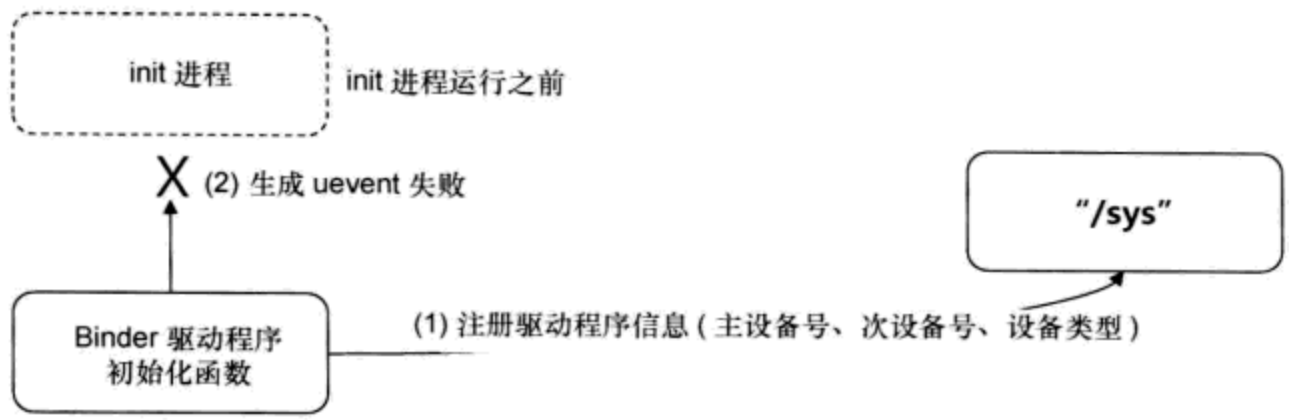


图 3-11 | 注册 Binder 驱动程序信息

Binder 驱动程序应该由用户级进程 init 创建设备节点文件，但由于系统正处于内核启动，无法发生 uevent。因此，仅向/sys 目录注册驱动程序信息，而后创建设备节点文件。

内核启动完毕后，init 进程启动，对于像 Binder 驱动程序这样无法创建设备节点文件的驱动，将采用冷插拔方式进行处理。init 进程事先获知等待冷拔插处理的驱动程序，并事先定义好各驱动程序的设备节点文件。在 Android 源代码的/system/core/init/devices.c 文件中，列出了 init 进程创建的节点文件的目录，如代码 3-33 所示。

```
static struct perms_ devperms[] = {
{ "/dev/null",          0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/zero",          0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/full",          0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/ptmx",          0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/tty",           0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/random",        0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/urandom",       0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/ashmem",        0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/binder",        0666,   AID_ROOT,       AID_ROOT,       0 },
{ "/dev/log/",          0662,   AID_ROOT,       AID_LOG,        1 },
/* these should not be world writable */
{ "/dev/android_adb",   0660,   AID_ADB,        AID_ADB,        0 },
{ "/dev/android_adb_enable",   0660,   AID_ADB,    AID_ADB,    0 },
```

代码 3-33 | 冷拔插设备的节点文件列表

在冷拔插处理时，init 进程会引用 devperms 结构体，在/dev 目录下创建设备节点文件。在代码 3-33 的 devperms 结构体中，分别列出了等待冷插拔处理的设备节点文件的名称、访问权限、用户 ID、组 ID。若想为用户定义的新设备创建设备节点文件，需要将相关驱动信息添加到 devperms 结构体中。

更多更新编程资源尽在 http://www.pin5i.com/

接下来，跟大家一起看一下 init 进程进行冷插拔处理的步骤。

首先，init 进程调用 device_ init() 函数，如代码 3-11 所示。

```
int device_init(void)
{
    fd = open_uevent_socket();
    t0 = get_usecs();
    coldboot(fd, "/sys/class");
    coldboot(fd, "/sys/block");
    coldboot(fd, "/sys/devices");
    t1 = get_usecs();
    log_event_print("coldboot %ld uS\n", ((long) (t1 - t0)));
}
```

代码 3-34 | init.c-device_init()函数

代码 3-34 是 device_init() 的函数体。device_init() 函数先创建一个套接字，用来接收 uevent。再通过 coldboot() 函数调用 do_coldboot() 函数，对内核启动时注册到/sys 目录下的驱动程序，进行冷插拔处理。代码 3-35 是 do_coldboot() 函数体。

```
static void do_coldboot(int event_fd, DIR *d)
{
    fd = openat(dfd, "uevent", O_WRONLY);
    if(fd >= 0) {
        write(fd, "add\n", 4);
        close(fd);
        handle_device_fd(event_fd);
    }
}
```

代码 3-35 | devices.c-由 do_coldboot()函数引发 uevent

do_coldboot()函数接收参数传递过来的目录路径，通过该路径查找到保存的 uevent 文件，向相关文件写入“add”信息，而后强制引起 uevent。然后在 handler_device_fd() 函数中接收相关的 uevent，获取 uevent 中的信息。

```
struct uevent {
    const char *action;
    const char *path;
    const char *subsystem;
    const char *firmware;
    int major;
    int minor;
}
void handle_device_fd(int fd)
{
```

更多更新编程资源尽在 http://www.pin5i.com/

```
        while((n = recv(fd, msg, UEVENT_MSG_LEN, 0)) > 0) {
            struct uevent uevent;
            parse_event(msg, &uevent);
            handle_device_event(&uevent);
        }
    }
```

代码 3-36 | uevent 处理

代码 3-36 描述了 uevent 的处理过程。handle_device_fd() 函数在收到 uevent 时，调用 parse_event() 函数，将 uevent 信息写入 uevent 结构体。比如，创建 Binder 驱动程序的设备节点的信息，分别保存在 uevent 结构体的各成员变量中，各变量值如下。

```
struct uevent {
    const char *action;        // "add"
    const char *path;          // "device/virtual/misc/binder"
    const char *subsystem;     // "misc"
    const char *firmware;       //
    int major;                 // 10
    int minor;                 // 62
}
```

代码 3-37 | 用于添加 Binder 驱动节点的 uevent 结构体

如代码 3-37，在 Binder 驱动程序的 uevent 信息中，包含了 init 进程写入的 add 动作，以及内核启动时 Binder 驱动写到/sys 目录下的设备信息。

向 uevent 结构体写完信息后，调用 handle_device_event() 函数，创建节点文件。

```
static void handle_device_event(struct uevent *uevent)
{
    .
    .
    if(!strncmp(uevent->subsystem, "block", 5)) {
         block = 1;
         base = "/dev/block/";
         mkdir(base, 0755);
    }
    .
    .
    else
        base = "/dev/";
    if(!strcmp(uevent->action, "add")) {
        make_device(devpath, block, uevent->major, uevent->minor);
    }
}
```

代码 3-38 | 由 handle_device_event() 函数在/dev 目录下创建子目录

更多更新编程资源尽在 http://www.pin5i.com/

代码 3-38 是 handle_device_event() 函数，此函数先检查 uevent 结构体的 subsystem 变量，而后在/dev 目录下创建子目录。subsystem 根据硬件用途的不同而表示不同的组。若硬件是存储设备，则 subsystem 是 block，创建的目录为/dev/block；若硬件是图形相关设备，则创建/dev/graphic 目录；若硬件是声音设备，则创建/dev/adsp 目录。在前面所介绍的 Binder 驱动的情形下，subsystem 即为 misc，但 handle_device_event() 函数不会创建与 misc 相关的下层目录。

在创建完所有下层目录后，调用 make_device() 函数，创建设备节点文件。

```
static void make_device(const char *path, int block, int major, int minor)
{
    mode = get_device_perm(path, &uid, &gid) | (block ? S_IFBLK : S_IFCHR);
    dev = (major << 8) | minor;
    mknod(path, mode, dev);
    chown(path, uid, gid);
}
```

代码 3-39 | device.c-make_device()函数创建设备节点

代码 3-39 是 make_device() 函数，此函数从设备节点文件列表（代码 3-33 中声明的）中获取用户 ID、组 ID 信息。而后调用 mknod() 函数，创建设备节点文件。

3.4.2　动态设备感知

init 进程支持热插拔处理，在系统运行中为新插入的设备创建设备节点文件。热插拔由 init 进程的事件处理循环来完成。

```
int main(int argc, char **argv)
{
   for(;;) {
      nr = poll(ufds, fd_count, timeout);
      .
      .
      if (ufds[0].revents == POLLIN)
               handle_device_fd(device_fd);
   }
   ...
}
```

代码 3-40 | init.c-main()-设备事件处理

代码 3-40 是 init 进程事件处理循环的一部分，通过热插拔机制来创建设备节点文

更多更新编程资源尽在 http://www.pin5i.com/

件。首先 init 进程的事件处理循环调用 poll() 函数监听来自驱动程序的 uevent，而后调用 handle_device _fd() 函数，创建设备节点文件，这个过程前面已经讲解过，请参考代码 3-36。

3.5 进程的终止与再启动

init 进程读取并分析 init.rc 文件，获得服务列表，而后从列表中依次启动服务子进程。init 进程启动的主要进程如下。

sh：搭载 Android 的机器终端，连接串口或 adbd 时，提供控制台输入输出的 shell 程序。

adbd：指 Android Debug Bridge，用来管理 QEMU 模拟器或实际机器的状态。该工具运行在目标机器（模拟器或物理机器）上，充当服务器，PC 充当连接服务器的客户端。比如，在 Eclipse 中开发应用程序时，常使用 DDMS（Dalvik Debug Monitor Service）进行调试，此时 DDMS 即是 adb 的客户端。

servicemanager：是 Android 中比较重要的一个进程，在 init 进程启动后启动，用来管理系统中的服务。详细内容，在第 7 章“Android Binder IPC”中介绍。

vold：指 Volume Dameon，用来挂载/管理 USB 存储或 SD 卡设备。

playmp3：在 Android 启动时，输出启动声音。

除以上这些进程外，init 进程还启动其他多种进程。若 init 启动的某个进程终止，则会对系统的运行产生影响。比如“服务管理器”（servicemanager），它是应用程序使用系统服务必须运行的进程。如果该进程出现意外终止，那么进程间的通信、图像输出、音频输出等功能将无法使用。因此，在 init 启动的进程中，除了一小部分外，其他大部分进程出现意外终止时，init 进程要重新启动它们。

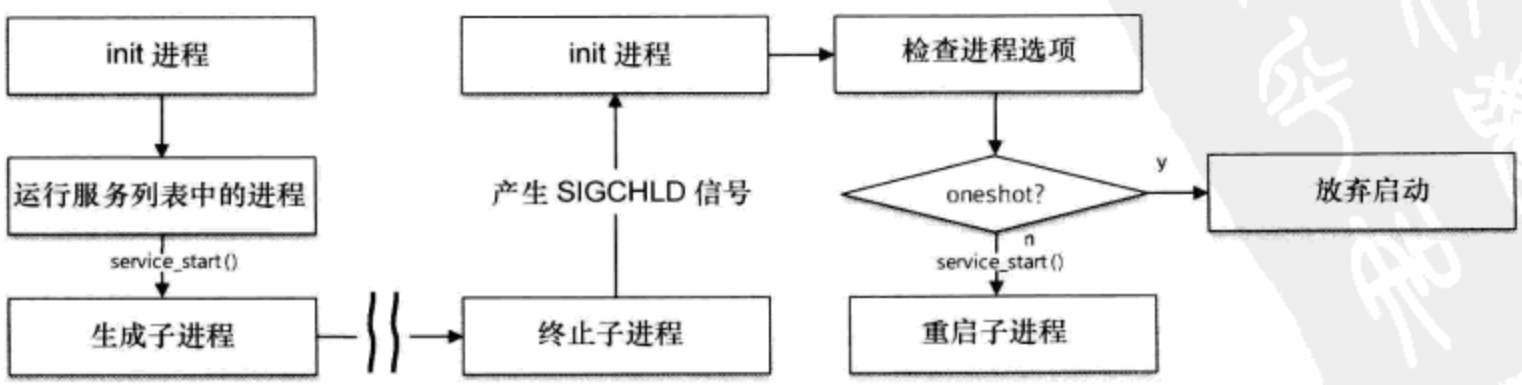


图 3-12 | init 进程处理 SIGCHLD 信号

在 3.3.2 节“服务列表”学习中，我们已经知道 init 进程会启动服务列表中的服务，

更多更新编程资源尽在 http://www.pin5i.com/

创建相应的子进程。如图 3-12 所示，当 init 的子进程意外终止时，会向父进程 init 进程传递 SIGCHLD 信号，init 进程接收该信号，检查进程选项是否设置为 oneshot，若设置为 oneshot，init 进程将放弃重启进程；否则重启进程。

关于 init 进程如何重启子进程，将在下一节进行讲解。

进程再启动代码分析

init 进程中有一个事件处理循环，如代码 3-20 所示，当其子进程终止时，init 会接收传递过来的 SIGCHLD 信号，并调用与之相对应的处理函数 sigchld_handler()，如代码 3-2 所示。

```
static void sigchld_handler(int s)
{
    write(signal_fd, &s, 1);
}
```

代码 3-41| sigchld_handler() 函数

代码 3-41 是 sigchld_handler() 函数，当发生 SIGCHLD 信号时，该函数被调用，所带参数用来接收 SIGCHLD 信号的编号，且相关编号被记录在已经创建的 socketDescriptor 中，如代码 3-17 所示。

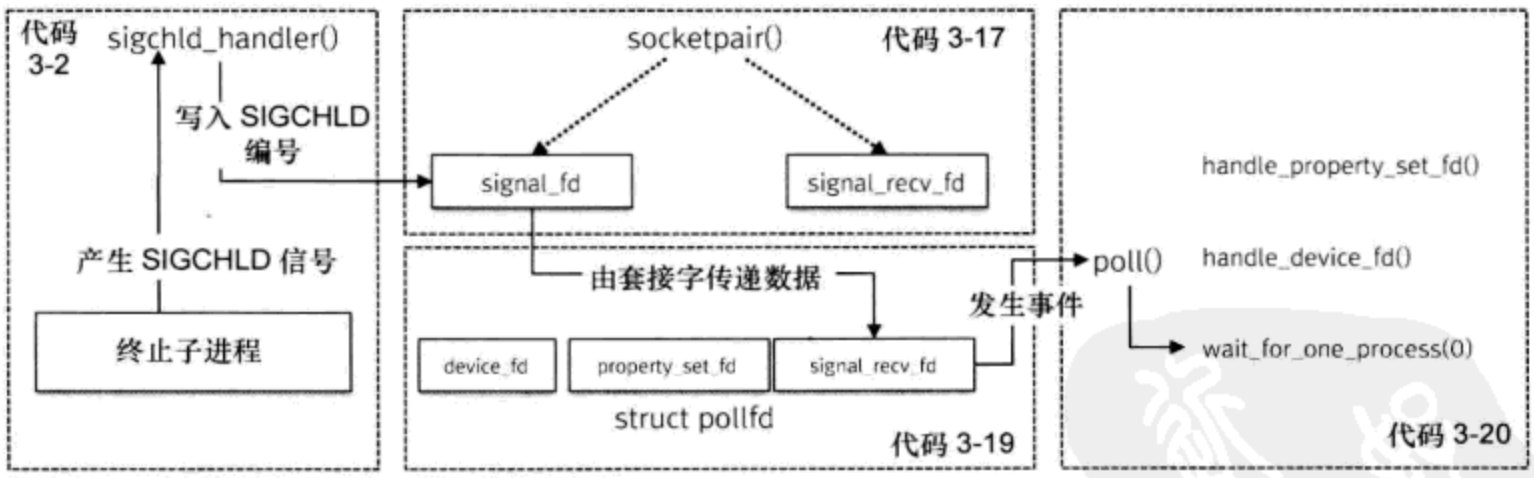


图 3-13 | 出现 SIGCHLD 信号后，调用重启处理函数的过程

图 3-13 描述了从发生 SIGCHLD 信号到调用子进程终止事件处理函数 wait_for_one_process() 整个过程。signal_fd 记录信号编号，由套接字对创建，信号编号被传递至接收端套接字描述符 signal_recv_fd 中。由于接收信号编号的 signal_recv_fd 已被注册至 POLL，如代码 3-19 所示，wait_for_one_process() 就会被调用执行。在代码 3-42 事件处理循环中，可以看到 wait_for_one_process() 函数被调用。

更多更新编程资源尽在 http://www.pin5i.com/

```
for(;;) {
   nr = poll(ufds, fd_count, timeout);                    ←❶
   if (nr <= 0)
        continue;

   if (ufds[2].revents == POLLIN) {
        /* we got a SIGCHLD - reap and restart as needed */
        read(signal_recv_fd, tmp, sizeof(tmp));
        while (!wait_for_one_process(0));                 ←❷
        continue;
   }
   ...
}
```

代码 3-42 | 在事件处理循环中，处理 SIGCHLD 信号的部分

❶ 当发生 SIGCHLD 信号时，程序就会从事件监听状态中跳出，而后执行 poll() 函数。

❷ 当 ufds[2]被注册进数据，即触发数据输入事件时，signal_recv_fd 即调用并执行 wait_for_one_process() 函数。wait_for_one_process() 函数在产生 SIGCHLD 信号的进程的服务列表中，检查进程的设置选项。若选项非 oneshot（SVC_ONE_SHOT），则添加重启选项（SVC_RESTARTING）。oneshot 选项被定义在 init.rc 文件的 service 部分中，若进程带有 oneshot 选项，进程终止时不会被重启。

代码 3-43 是 wait_for_one_process() 函数体部分。

```
static int wait_for_one_process(int block)
{
    ...
    while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1      ←❶
                                                    && errno == EINTR );

    svc = service_find_by_pid(pid);          ←❷
    if (!(svc->flags & SVC_ONESHOT)) {       ←❸
         kill(-pid, SIGKILL);
    }
    /* remove any sockets we may have created */
    for (si = svc->sockets; si; si = si->next) {     ←❹
         unlink(tmp);
    }

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

    if (svc->flags & SVC_ONESHOT) {                  ←❻
         svc->flags |= SVC_DISABLED;
    }
    if (svc->flags & SVC_DISABLED)
       return 0;
    }
```

更多更新编程资源尽在 http://www.pin5i.com/


```
        list_for_each(node, &svc->onrestart.commands) {          ←❼
            cmd = node_to_item(node, struct command, clist);
            cmd->func(cmd->nargs, cmd->args);
        }

        svc->flags |= SVC_RESTARTING;                            ←❽
    }
```

代码 3-43 | int.c- wait_for_one_process()-waitpid (-1)

❶ 当产生信号的进程被终止时，waitpid() 函数用来回收进程所占用的资源，它带有三个参数。其中，第一个参数 pid 为欲等待的子进程的识别码，设置为-1，表示查看所有子进程是否发出 SIGCHLD 信号；第二个参数 status，用于返回子进程的结束状态；第三个参数决定 waitpid() 函数是否应用阻塞处理方式。waitpid()函数返回 pid 值，返回值即是产生 SIGCHLD 信号的进程的 pid 号。

❷ service_find_by_pid() 函数用来取出与服务列表中终止进程相关的服务项目。

❸ 在取出的服务项目选项中，检查 SVC_ONESHOT 是否已设置。SVC_ONESHOT 表示进程仅运行一次，带有此选项的进程在运行一次后，不会被重新启动，由 kill（-pid，SIGKILL）函数终止。

❹ 删除进程持有的所有 socketDescriptor。

❺ SVC_RUNNING 表示在服务项持有的 pid 值与状态标记中，进程正处于驱动运行中。此段代码将删除 SVC_RUNNING。

❻ SVC_ONESHOT 选项将已设置进程标记为 SVC_DISABLED，并从 wait_for_one_process()函数中跳出，相关进程将不被重新启动。

❼ 检查待重启的进程在 init.rc 文件中是否带有 onrestart 选项。onrestart 选项是进程重启时待执行的命令。代码 3-44 是带有 onrestart 选项的进程的示例。

```
service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media
```

代码 3-44 | init.rc 文件中 servicemanager 的定义

servicemanager 重启后，调用 on_restart()[1]函数，重启 zygote 与 media。

1　在 http://android.git.kernel.org/?p=platform/system/core.git;a=tree 的 init 目录下的 builtins.c 文件中。

更多更新编程资源尽在 http://www.pin5i.com/

❽ 最后，向当前服务项的标记中添加 SVC_RESTART。此标记被应用在 restart_processes() 函数中，用来确定待重启的进程。restart_processes() 函数如代码 3-45 所示。

当 wait_for_one_process() 函数执行完毕后，事件处理循环中的 restart_processes() 函数就会被调用执行。代码 3-45 是 restart_processes() 函数。

```
static void restart_service_if_needed(struct service *svc)
{
    svc->flags &= (~SVC_RESTARTING);
    service_start(svc);
    return;
}
static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,  restart_service_if_needed);
}
```

代码 3-45|restart_processes()函数

restart_processes() 函数运行服务列表中带有 SVC_RESTART 标记的进程。当一个带有此标记的进程被终止，产生 SIGCHLD 信号时，restart_processes() 函数将重新启动它。

3.6 属性服务

属性变更请求是 init 事件处理循环处理的另一个事件。在 Android 平台中，为了让运行中的所有进程共享系统运行时所需要的各种设置值，系统开辟了属性存储区域，并提供了访问该区域的 API。属性由键（key）与值（value）构成，其表现形式为“键 = 值”。在 Linux 系统中，属性服务主要用来设置环境变量，提供各进程访问设定的环境变量值[1]。在 Android 平台中，属性服务得到更系统地应用，在访问属性值时，添加了访问权限控制，增强了访问的安全性。系统中所有运行中的进程都可以访问属性值，但仅有 init 进程才能修改属性值。其他进程修改属性值时，必须向 init 进程提出请求，最终由 init 进程负责修改属性值。在此过程中，init 进程会先检查各属性的访问权限，而后再修改属性值。当属性值更改后，若定义在 init.rc 文件中的某个特定条件得到满足，则与此条件相匹配的动作就会发生。每个动作都有一个“触发器”（trigger），它决定动作的执行时间，记录在“on property”关键字后的命令即被执行。

图 3-14 简单地描述了 init 进程与其他进程在访问并修改属性值的大致情形。

1 getenv()、setenv()等函数用来访问或设置环境变量。

更多更新编程资源尽在 http://www.pin5i.com/

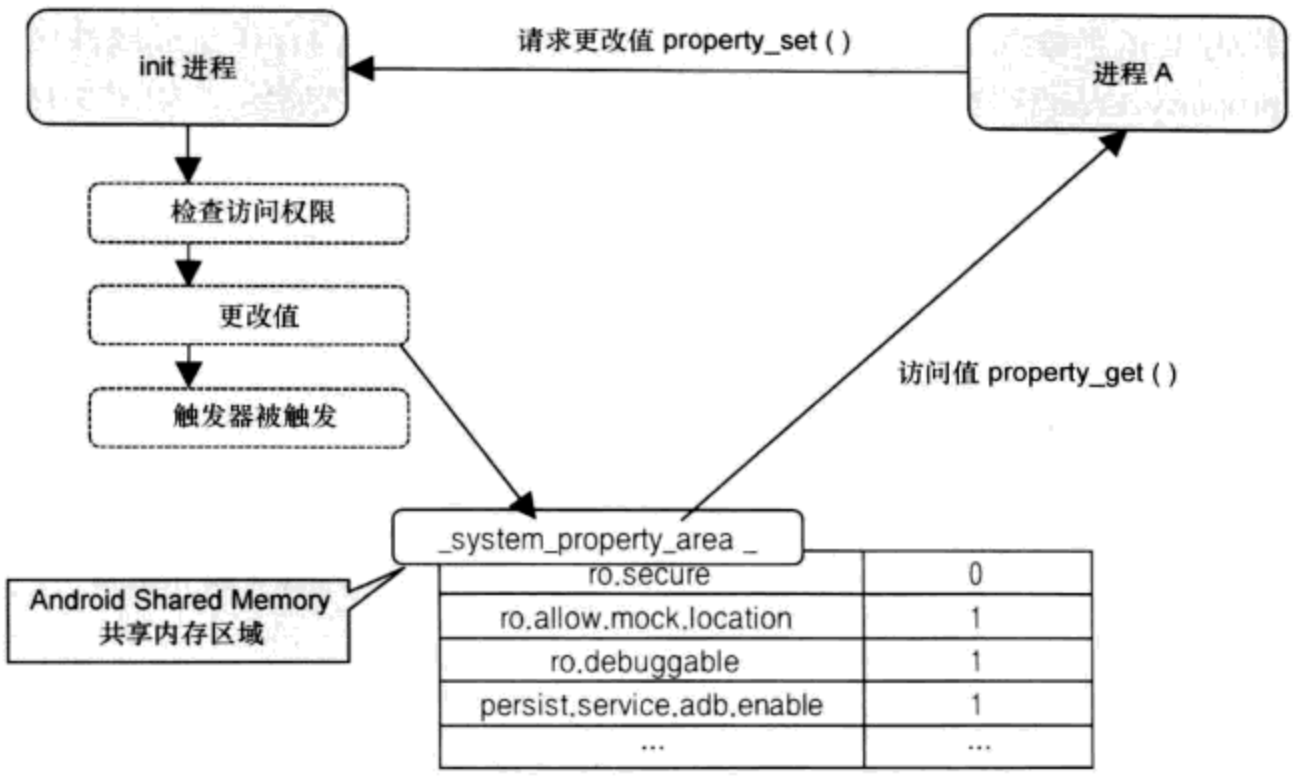


图 3-14 | 属性服务的作用

3.6.1　属性初始化

在 init 进程的 main() 函数中，调用了 property_init() 函数，用来初始化属性域，如代码 3-46❸所示。为了保存属性值，property_init()函数首先在内存中开辟一块共享区域，而后将其用作 ashmem（Android Shared Memory）[1]。外部进程可以访问这块共享内存域，获取属性值，但它们不能通过直接访问共享内存域的方式来更改属性值。一个进程若想更改属性值，必须先向 init 进程提交属性变更请求，由 init 进程更改共享内存中的属性值。

```
void property_init(void)
{
    init_property_area();                                    ❶
    load_properties_from_file(PROP_PATH_RAMDISK_DEFAULT);    ❷
}

int main(int argc, char **argv)
{
    ...
    property_init();                                         ❸
    ...
}
```

代码 3-46 | init.c-main()-属性初始化

1　Ashmen 即 named shared memory，与普通 shared memory 不同，它不使用独立的 key，而是通过字符串来获取内存的访问权限。

更多更新编程资源尽在 http://www.pin5i.com/

在代码 3-46 的❸中，调用了 property_init() 函数，在共享内存中生成属性域。当❶中的 init_property_area() 函数被调用执行后，所创建的属性域被初始化，如图 3-15 所示。

属性域的起始 1024 个字节作为属性域头，用来保存管理属性表所需要的一些数值。其余 31616 个字节空间被划分成 247 块，每块大小为 128 字节，用来保存属性值。在访问或更改属性值时，使用的函数分别为 property_get() 与 property_set()。在属性域完成初始化之后，就会从指定的文件中读取初始值，并设置为属性值，如代码 3-46❸所示。

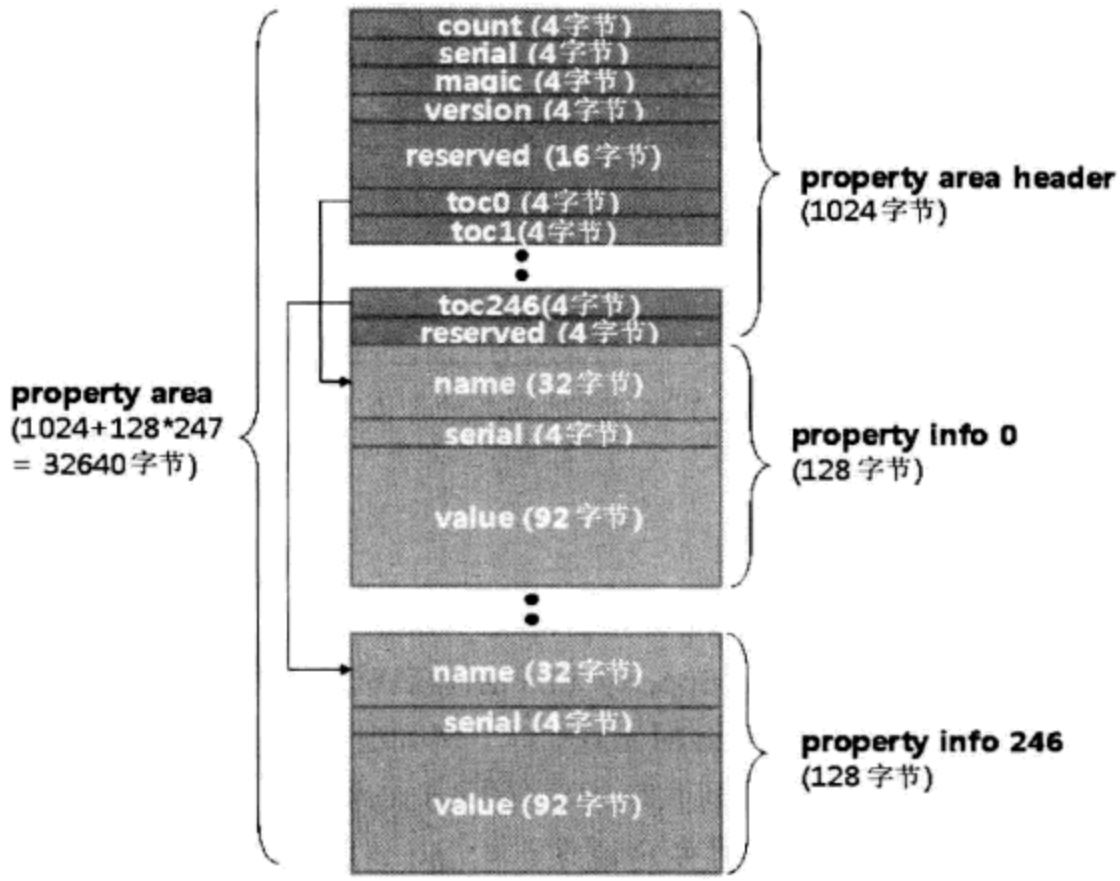


图 3-15 | 属性域的结构

```
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=0
ro.allow.mock.location=1
ro.debuggable=1
persist.service.adb.enable=1
```

图 3-16 | default.prop 文件

图 3-16 是模拟器中使用的 default.prop 文件，其中的数据以键值对（<key> = <value>）的形式存在。

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

更多更新编程资源尽在 http://www.pin5i.com/


```
    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);                    ←❶
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_properties_from_file(PROP_PATH_LOCAL_OVERRIDE);
    /* Read persistent properties after all default values have been loaded. */
    load_persistent_properties();                                         ←❷
    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);       ←❸
    if(fd < 0) return -1;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);

    listen(fd, 8);
    return fd;
}

int main(int argc, char **argv)
{
    …
    property_set_fd = start_property_service();                           ←❹
    …
}
```

代码 3-47 | init.c-创建属性服务套接字

如代码 3-47❹所示，在 init 的 main() 函数中调用了 start_property_service() 函数，用来创建启动属性服务所需要的 Unix 域套接字，并保存套接字描述符。start_property_service() 函数在创建套接字之前，先读取存储在各文件中的基本设置值，将它们设置为属性值，如代码❶。当所有系统初始值设置完毕后，开始读取保存在/data/property 目录中的属性值，如代码❷。在/data/property 目录中保存着系统运行中其他进程新生成的属性值或更改的属性值，属性的 key 被用作文件名，value 被保存在文件中。在属性初始值设置完毕后，就会创建名称为/dev/socket/property_service 的 Unix 域套接字，如代码❸。

3.6.2　属性变更请求处理

通过上面创建的 Unix 域套接字，接收到属性变更请求后，init 进程就会调用 handle_property_set_fd() 函数。

```
void handle_property_set_fd(int fd)
{
    …
    /* Check socket options here */
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {      ←❶
        close(s);
        ERROR("Unable to recieve socket options\n");
        return;
    }
```

更多更新编程资源尽在 http://www.pin5i.com/

```
    …
    switch(msg.cmd) {
        case PROP_MSG_SETPROP:
            …
                if(memcmp(msg.name,"ctl.",4) == 0) {
                    if (check_control_perms(msg.value, cr.uid)) {          ←❷
                        handle_control_message((char*) msg.name + 4, (char*) msg.value);
                    }
                    …
                } else {
                    if (check_perms(msg.name, cr.uid)) {                   ←❸
                        property_set((char*) msg.name, (char*) msg.value); ←❹
                    }
                    …
                }
                …
    }
}
```

代码 3-48 | init.c-handle_property_set_fd()-检查权限及处理信息

如代码❶所示，执行 handle_property_set_fd() 函数时，先从套接字获取 SO_PEERCRED 值，以便检查传递信息的进程的访问权限。在 struct ucred 结构体中，存储着传递信息的进程的 uid、pid 与 gid 值。通过此结构体中的值，以及消息的类型，检查进程的访问权限。在属性消息中，以“ctl”开头的消息并非请求更改系统属性值的消息，而是请求进程启动与终止的消息。在代码❷中调用 check_control_perms() 函数检查访问权限，仅有 system server、root 以及相关进程才能使用 ctl 消息，终止或启动进程。

除此之外，其他消息都被用来更改系统的属性值，如代码 3-48❸所示，调用 check_perms() 函数检查访问权限。各属性的访问权限采用 Linux 的 uid 进行区分，其定义如代码 3-49 所示。若要在系统运行中变更属性设置，应充分考虑各属性的访问权限。

```
struct {
    const char *prefix;
    unsigned int uid;
} property_perms[] = {
    { "net.rmnet0.",        AID_RADIO },
    { "net.gprs.",          AID_RADIO },
    { "ril.",               AID_RADIO },
    { "gsm.",               AID_RADIO },
    { "net.dns",            AID_RADIO },
    { "net.",               AID_SYSTEM },
    { "dev.",               AID_SYSTEM },
    { "runtime.",           AID_SYSTEM },
    { "hw.",                AID_SYSTEM },
    { "sys.",               AID_SYSTEM },
    { "service.",           AID_SYSTEM },
    { "wlan.",              AID_SYSTEM },
    { "dhcp.",              AID_SYSTEM },
    { "dhcp.",              AID_DHCP },
```

更多更新编程资源尽在 http://www.pin5i.com/


```
    { "vpn.",               AID_SYSTEM },
    { "vpn.",               AID_VPN },
    { "debug.",             AID_SHELL },
    { "log.",               AID_SHELL },
    { "service.adb.root",   AID_SHELL },
    { "persist.sys.",       AID_SYSTEM },
    { "persist.service.",   AID_SYSTEM },
    { NULL, 0 }
};
```

代码 3-49 | property_service.c-struck property_perms-访问权限

最后，在代码 3-48❹中，调用 property_set() 函数，更改属性值。若没有出现问题，接着调用 property_changed() 函数。在 init.rc 脚本文件中，记录着某个属性改变后要采取的动作，动作执行的条件以“on property:<key> = <value>”形式给出。当某个条件相关的键值被设定后，与该条件相关的触发器就会被触发。图 3-17 是 init.rc 文件中定义的属性触发器，比如当 ro.kernel.qemu 属性值设置为 1 时，adbd 服务就会启动。

```
# adbd on at boot in emulator
on property:ro.kernel.qemu=1
    start adbd

on property:persist.service.adb.enable=1
    start adbd

on property:persist.service.adb.enable=0
    stop adbd
```

图 3-17 | init.rc 文件中的属性触发器

3.7　小结

在 Android 系统之前，嵌入式系统开发者在将 Linux 移植到目标设备时，主要从事系统内核、设备驱动程序的开发。而在 Android 系统出现后，开发者不仅要考虑内核移植的问题，还要考虑到根文件系统的移植。在根文件系统的移植中，首先面临的问题是 init 进程常常无故中断执行，大多是设备节点文件创建不正常、init.rc 文件错误等原因引起的。本章详实地介绍了 init 进程的运行过程，认真学习这些内容，有利于各位掌握 Android 系统的启动过程，节省大家学习的时间，提高学习效率。

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 4 章

JNI 与 NDK

4.1 Android 与 JNI

学习 Android 必须了解 JNI 吗

从编程语言看，Android Framework 由基于 Java 语言的 Java 层与基于 C/C++语言的 C/C++层组成，每个层中的功能模块都是使用相应的语言编写的，并且两个层中的大部分模块之间保持着千丝万缕的联系。以 Android GPS 子系统为例，如图 4-1（右）所示，它提供终端的地理位置信息。在 Android 应用程序获取 GPS 信息时，需要先调用应用程序框架中 Location Manager 提供的 Java API，而后通过调用框架内的 GPS 库，连接到 GPS 设备驱动上，应用程序即可获取当前地理位置信息。在此过程中，C/C++层与 Java 层相互作用、相互配合，共同完成任务。

在 Android Framework 中，需要提供一种媒介或桥梁，将 Java 层（上层）与 C/C++层（底层）有机地联系起来，使得它们相互协调，共同完成某些任务。在两层之间充当连接桥梁这一角色的就是 Java 本地接口（JNI，Java Native Interface），它允许 Java 代码与基于 C/C++编写的应用程序和库进行交互操作，如图 4-1（左）所示。

JNI 提供了一系列接口，允许 Java 类与使用 C/C++等其他编程语言（在 JNI 中，这些语言被称为本地语言）编写的应用程序、模块、库进行交互操作。比如，在 Java 类中使用 C 语言库中的特定函数，或在 C 语言程序中使用 Java 类库，都需要借助 JNI 来完成，如图 4-2 所示。

通常在下列几种情况下使用 JNI。

- 注重处理速度

  与本地代码（C/C++等）相比，Java 代码的执行速度要慢一些。如果对某段程

更多更新编程资源尽在 http://www.pin5i.com/

序的执行速度有较高的要求，建议使用 C/C++编写代码。而后在 Java 中通过 JNI 调用基于 C/C++编写的部分，常常能够获得更快的运行速度。

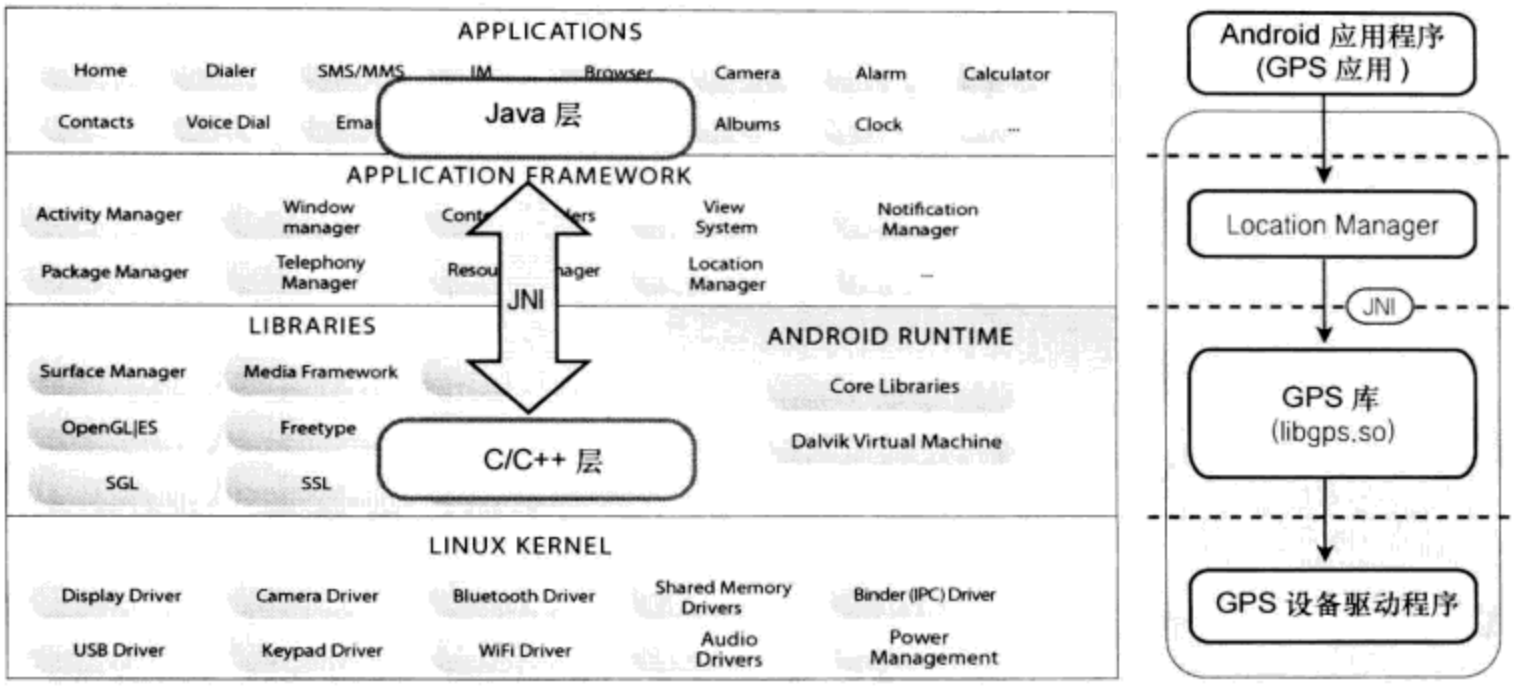


图 4-1 | 在 Android Framework 中连接 C/C++与 Java 的 JNI（左）/Android GPS 子系统（右）

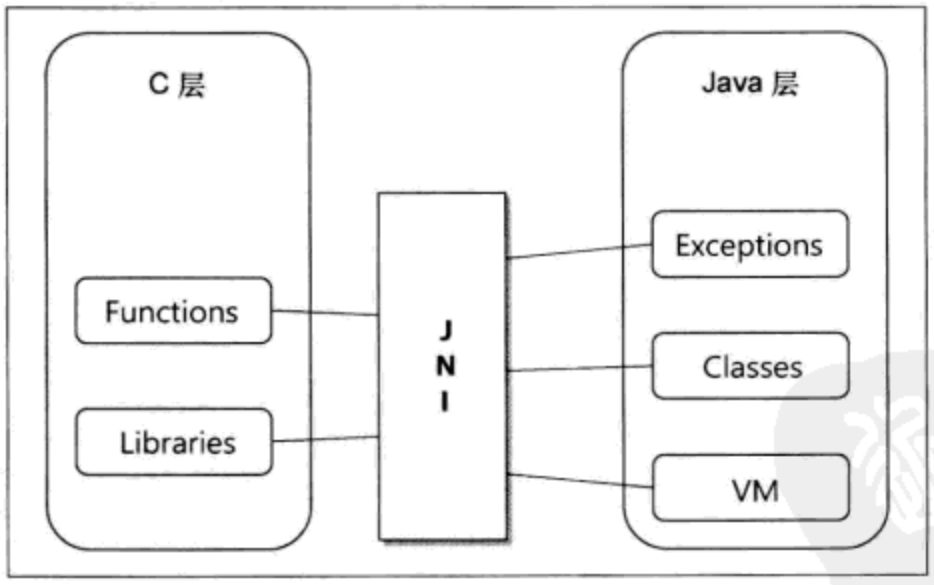


图 4-2 | JNI 连接 Java 层与 C 层

- 硬件控制

  为了更好地控制硬件，硬件控制代码通常使用 C 语言编写。而后借助 JNI 将其与 Java 层连接起来，从而实现对硬件的控制。

- 既有 C/C++代码的复用

  在程序编写过程中，常常会使用一些已经编写好的 C/C++代码，既提高了编程效率，又确保了程序的安全性与健壮性。在复用这些 C/C++代码时，就要

更多更新编程资源尽在 http://www.pin5i.com/

通过 JNI 来实现。

在 Android Framework 中，借助 JNI 综合了 Java 语言与 C/C++等本地语言的优点，使得开发者既可以利用 Java 语言跨平台、类库丰富、开发便捷等特点，又可以利用本地语言的高效，开发运行效率更高，更健壮、更安全的程序。无论你是应用程序开发者，还是平台开发者，若想充分利用 Android Framework，必须对 JNI 有相当的了解。

首先，作为 Android 应用程序开发者，必须了解 JNI。Android 应用程序是使用 Android SDK 开发的 Java 程序，它运行在一个被称为“Dalvik”的虚拟机（Dalvik Virtual Machine）上。与使用 C/C++开发的应用程序相比，它仍然带有 Java 的一些缺点，比如运行速度慢等问题。在开发图形处理或信号处理这类对 CPU 处理速度有较高要求的程序时，使用 C/C++等本地语言编写的相应模块，执行效率更高，性能也好得多。

在这种情形下，应用程序开发者可以使用 C/C++这类本地语言开发对性能要求较高的模块，再在 Java 程序中借助 JNI 将 Java 程序与 C/C++模块连接集成在一起，从而开发出一个运行速度快，执行效率高，健壮、安全的程序。

在实际 Android 应用开发中，开发者通常使用 Android SDK（Software Development Kit）开发 Java 程序。而对于性能要求较高的，常常使用 Android 提供的 NDK（Native Development Kit）开发基于 C/C++的本地库。而后再通过 JNI 将 Java 程序与 C/C++程序集成在一起。NDK 提供了一系列的工具，帮助开发者快速开发 C/C++动态库。关于 NDK 的内容，在后面部分将作更加详细的介绍。

同样，Android 平台开发者也要学习了解 JNI。比如，在搭载 Android 的设备上安装 Android Framework 不支持的硬件时，平台开发人员必须使用 C 语言实现设备的驱动程序，以便对设备进行控制。

虽然已经为硬件设备开发了驱动程序，但由于使用 C 语言开发而成，这使得使用 Java 开发 Android 应用程序的开发者，无法直接访问 C 语言实现的设备驱动程序，也就不能对硬件进行控制。Android 应用程序开发者同样需要一个桥梁，使得 Java 程序能够访问使用 C 语言编写的驱动程序，这个桥梁就是 JNI。

再次以图 4-1（右）所示的 GPS 子系统为例，为了使用 GPS 接收器的功能，Android 应用程序开发者需要一套控制硬件的 Java API，使用 JNI 将 C/C++实现的硬件设备驱动程序映射为 Java API。这样，Android 应用程序开发者就可以通过 Java 编写的 API 使用添加的新硬件了。

为了理解 Android Framework，并且灵活高效地使用它，必须认真学习作为基本知识之一的 JNI。本章将通过多个代码示例，向各位介绍 Android 中使用 JNI 的各种技术，

更多更新编程资源尽在 http://www.pin5i.com/

即以实际 Android 源代码为基础，学习在 Android 系统中如何灵活运用 JNI。

4.2 JNI 的基本原理

4.2.1 在 Java 中调用 C 库函数

作为一名使用 Java 开发 Android 应用程序的开发者，学习 JNI 时，首先要了解在 Java 代码中如何通过 JNI 调用由 C 语言编写的代码，如 C 库函数。下面通过一个简单的示例，分成几个步骤，向各位讲解 JNI 运行的原理。

示例程序是一个非常简单的 Java 程序，该程序将对 C 函数进行调用，被调用的 C 函数是一个向控制台输出字符串的简单函数。示例虽然简单，但能说明 JNI 的运行机制，了解 Java 程序如何通过 JNI 调用 C 函数。示例中所列出的步骤具有通用性，希望大家认真学习并记住这些步骤。

正式开始之前，首先向大家讲解一下开发的流程。在 Java 代码中通过 JNI 调用 C 函数的步骤如下。

- 第一步：编写 Java 代码
- 第二步：编译 Java 代码
- 第三步：生成 C 语言头文件
- 第四步：编写 C 代码
- 第五步：生成 C 共享库
- 第六步：运行 Java 程序

第一步：编写 Java 代码

首先编写调用 C 函数的 Java 源代码“HelloJNI.java”，如代码 4-1 所示。

若想在 Java 代码中通过 JNI 调用 C 函数，首先在 Java 类中声明本地方法，如代码 4-1 所示。本地方法仅在 Java 程序中进行声明，使用 C/C++等本地语言来实现。

> **TIP 本地方法 Vs 本地函数**
>
> 查看 JNI 实现，可以发现方法是在 Java 代码中声明的，而具体实现是在 C/C++代码中。在一般的 JNI 书籍中，将这两种情形下的方法都统称为本地方法，容易混淆。本书为了避免出现这种混淆，进行了明确的区分，即将仅在 Java 代码中声明的方法称为本地方法，使用 C/C++实现的函数称为本地函数。

更多更新编程资源尽在 http://www.pin5i.com/


```
class HelloJNI
{
   // 本地方法声明
   native void printHello();                                    ❶
   native void printString(String str);

   // 加载库
   static {  System.loadLibrary("hellojni");   }  ◀— ❷

   public static void main(String args[])
   {
      HelloJNI myJNI = new HelloJNI();

      // 调用本地方法(实际调用的是使用 C 语言编写的 JNI 本地函数)
      myJNI.printHello();
      myJNI.printString("Hello World from printString fun");    ❸
   }
}
```

代码 4-1 | HelloJNI.java 源代码

❶ 在 Java 类中，使用“native”关键字，声明本地方法，该方法与用 C/C++编写的 JNI 本地函数相对应。“native”关键字告知 Java 编译器，在 Java 代码中带有该关键字的方法只是声明，具体由 C/C++等其他语言编写实现。在示例的 HelloJNI 类中带有 native 关键字的 printHello()与 printString()两个方法只是声明，没有具体实现。

如果去掉方法前的 native 关键字，编译代码时，Java 编译器就会报错，抛出编译错误，告知该方法没有实现。

❷ ❶在 Java 类中仅声明了两个本地方法。接下来，调用 System.loadLibrary()方法，加载具体实现本地方法的 C 运行库。System.loadLibrary()方法加载由字符串参数指定的本地库，在不同操作系统平台下，加载的 C 运行库不同，表 4-1 列出了分别在 Windows 与 Linux 平台下加载的 C 运行库。如果当前是在 Windows 操作系统下，调用 System.loadLibrary(“hellojni”)，则 hellojni.dll 会被加载。

表 4-1　不同平台下 System.loadLibrary()方法加载的 C 运行库

操作系统平台	System.loadLibrary()方法的参数	实际加载的 C 运行库
Windows	Hellojni	hellojni.dll
Linux	Hellojni	libhellojni.so

在 Java 中加载本地运行库时通常使用“静态块”（Static Block），如示例所示。如果本地库未被正常加载，在使用本地方法调用本地库中的 C 函数时，就会发生错误。

更多更新编程资源尽在 http://www.pin5i.com/

因此在程序中使用静态块将 C 运行库在调用本地方法之前加载进来，以避免本地方法在调用 C 函数时发生错误。

❸ 在步骤❶、❷中编写好通过 JNI 调用 C 函数的 Java 类之后，在 main()函数中，创建 HelloJNI 类的对象，调用对象的本地方法，实现对 JNI 本地函数的调用。

第二步：编译 Java 代码

使用 Java 编译器（javac），编译步骤 1 中编写的 Java 源代码。编译 Java 源代码，请执行如下命令。

```
javac HelloJNI.java
```

在执行 Java 编译命令前，先查看 PC 中是否已经设置好 JDK（Java Development Kit）。若 JDK 尚未设置好，执行编译命令时就会提示命令错误，建议登录 oracle 的 Java 下载网站（http://www.oracle.com/us/sun/index.html），下载并安装最新版本的 JDK，而后将 javac 等 Java 命令添加到环境变量中。

编译好 HelloJNI.java 后，生成 HelloJNI.class 文件，如图 4-3 所示。此时，若直接执行编译好的字节码文件，就会抛出异常，如图 4-4 所示。由于尚未创建加载到 Java 代码中的 hellojni.dll 文件，无法找到 Java 虚拟机要加载的 C 运行库，所以 Java 程序运行时抛出无法找到 hellojni.dll 文件的异常。

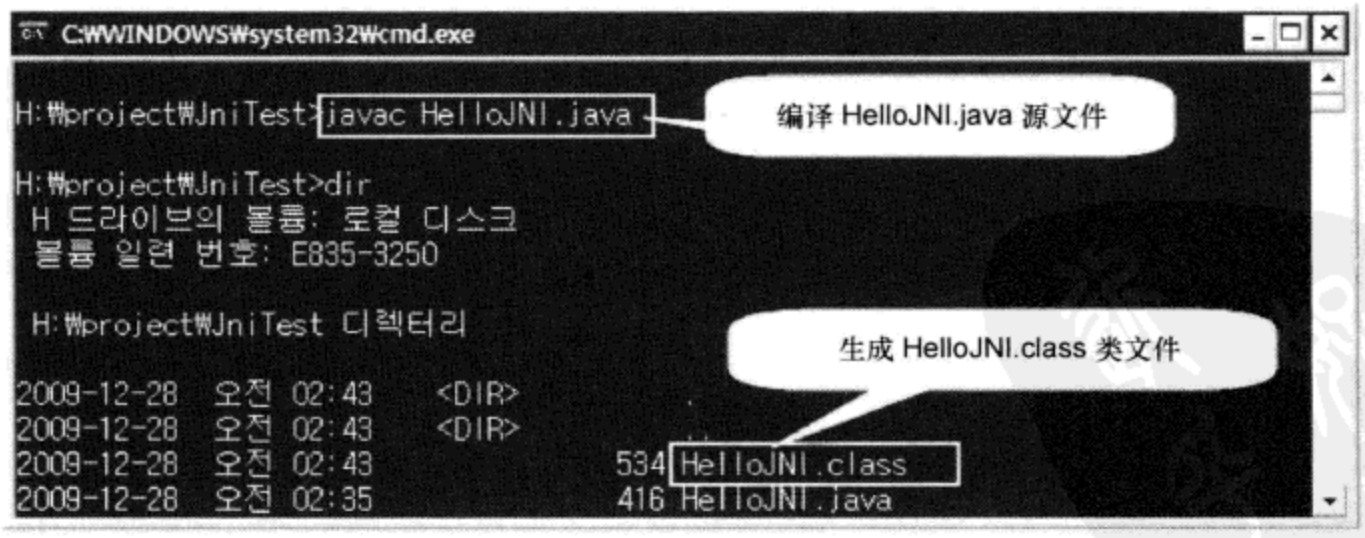


图 4-3 | 编译 HelloJNI.java

接下来，创建 hellojni.dll 运行库文件。

第三步：生成 C 语言头文件

Java 代码编写、编译完成后，在第三步到第五步中，将创建 hellojni.dll 运行库文件，具体实现 HelloJNI 类中声明的两个本地方法 printHello()与 printString()。

更多更新编程资源尽在 http://www.pin5i.com/

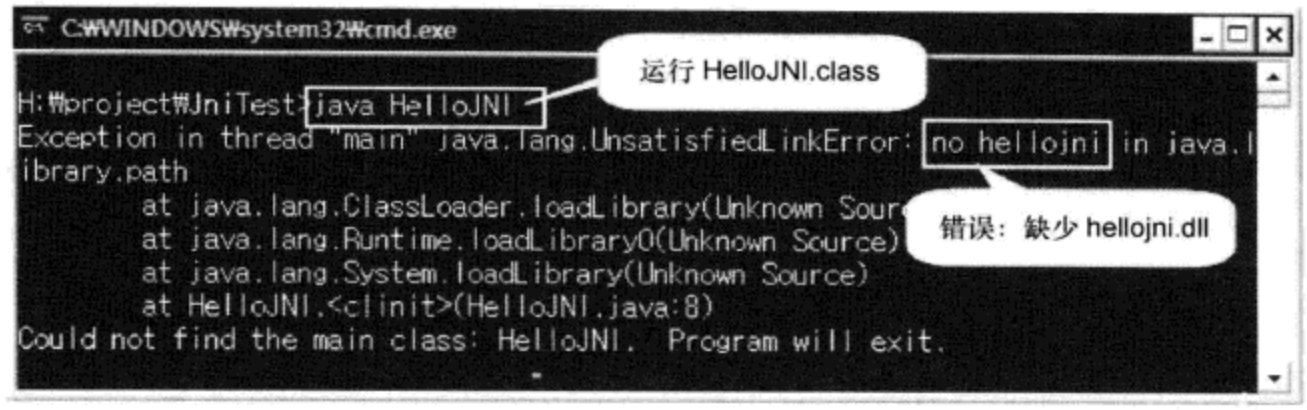


图 4-4 | 因尚未创建 hellojni.dll，运行 HelloJNI 发生错误

“如图 4-5 所示，在 HelloJNI 类中声明了 printHello()本地方法后，就要在 hellojni.dll 运行库中实现一个相同签名的 printHello()函数。并且编写好的 hellojni.dll 运行库由 System.loadLibrary()方法加载。那么，当在 Java 类中调用使用 native 关键字声明的 printHello()方法时，Java 虚拟机会正常调用 hellojni.dll 运行库中的 printHello()函数吗？”

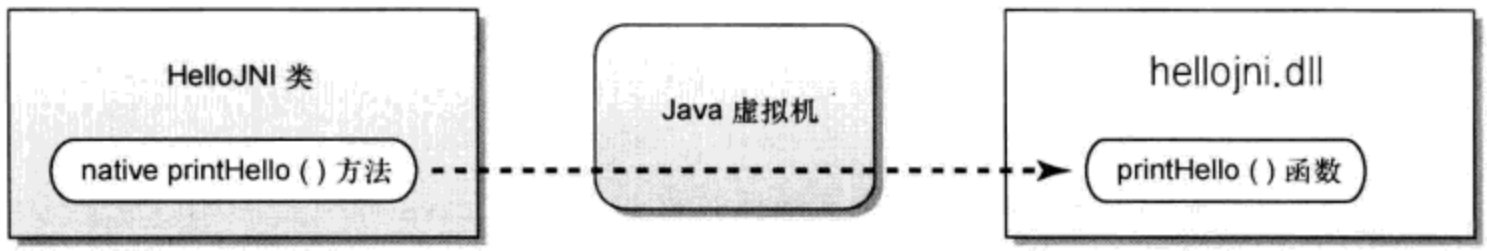


图 4-5 | 具有相同签名的本地方法与 JNI 本地函数的连接关系

为了弄清这一问题，假设已经在 hellojni.dll 中编写好 C 语言代码，如代码 4-2 所示。在此段代码中，实现了 Java 代码中声明的本地方法，本地方法与本地函数具有类似的签名。

```
// 实现本地方法 printHello()
void printHello()
{
   printf("Hello World!\n");
   return;
}

// 实现本地方法 printString(String str)
void printString(char *str)
{
   printf("%s!\n", str);
   return;
}
```

代码 4-2 | 错误的 hellojni.c 代码

更多更新编程资源尽在 http://www.pin5i.com/

编译上述代码（编译方法将在第五步进行说明），生成 hellojni.dll 运行库，而后运行 HelloJNI 类，会出现什么结果呢？

当在 HelloJNI.java 代码中，调用 printHello()与 printString()本地方法时，代码 4-2 中的 C 函数（printHello()与 printString()）会被正常调用吗？

```
//HelloJNI.java （代码 4-1）的一部分
…
myJNI.printHello();
myJNI.printString("Hello World from printString fun");
```

由代码 4-2 生成 hellojni.dll 运行库后，运行 HelloJNI 类，会产生 java.lang.UnsatisfiedLinkError 错误。当 Java 虚拟机无法找到与本地方法（如 printHello()）相链接的 JNI 本地函数时就会抛出这个错误，具体错误描述如下。

```
Exception in thread "main" java.lang.UnsatisfiedLinkError : HelloJNI.printHello()
```

图 4-6 | HelloJNI 运行抛出错误

运行时，Java 虚拟机会在加载的本地运行库中查找与 Java 本地方法相匹配的 C 函数，并生成映射表，而后将 Java 本地方法与 C 函数链接在一起。

但是 Java 虚拟机并非如代码 4-2 所示，直接将 C 函数与 Java 本地方法映射在一起。如图 4-7 所示，当 Java 虚拟机在由代码 4-2 生成的 hellojni.dll 运行库中查找不到与本地方法 printHello()相匹配的本地函数时，就会抛出如图 4-6 所示的 UnsatisfiedLinkError 错误。

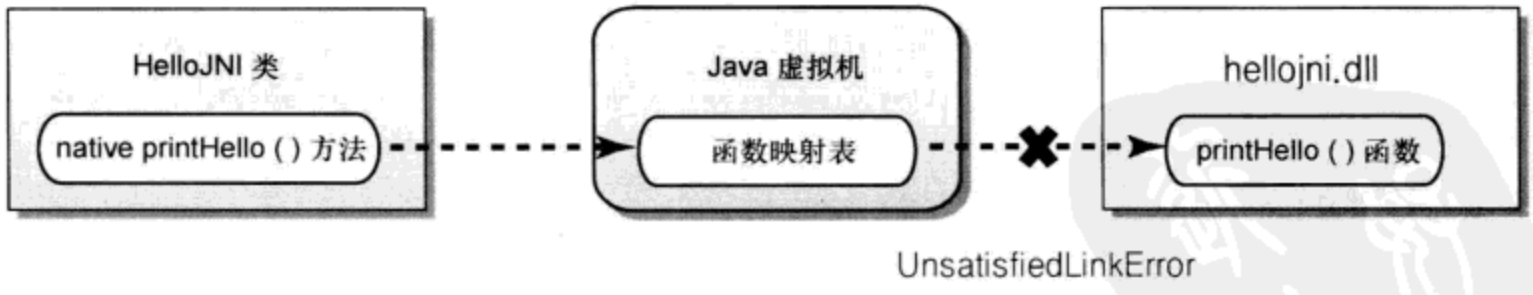


图 4-7 | Java 虚拟机查找不到与本地方法 printHello()相匹配的 JNI 本地函数

那么，Java 虚拟机在加载本地运行库时，如何把运行库中的 C 函数与 Java 代码中的本地方法映射在一起的呢？答案是使用函数原型。当生成了函数原型，Java 虚拟机即可把本地库中相应的函数与 Java 本地方法映射在一起。

若想创建本地方法的映射 C 函数，必须先生成函数原型，函数原型存在于 C/C++ 头文件中。Java 提供了 javah 工具，它位于<JDK_HOME>[1]\bin 目录下，用来生成包含

1 <JDK_HOME>是设置 JDK（Java Development Kit）的根目录。在笔者的机器上，JDK 被设置在 C:\Program Files\Java\jdk1.6.0_14。

更多更新编程资源尽在 http://www.pin5i.com/

函数原型的 C/C++头文件，其使用方法如下。

```
javah  <包含以 native 关键字声明的方法的 Java 类名称>
```

运行 javah 命令，会在当前目录下生成与 Java 类名（即 javah 命令的参数）相同名称的 C 语言头文件。在生成的 C 头文件中，定义了与 Java 本地方法相链接的 C 函数原型。

如图 4-8 所示，使用 javah 命令生成 HelloJNI.h 头文件。如打开 HelloJNI.h 头文件，在其中即可看到与 HelloJNI 类中声明的本地方法相对应的 C 函数原型。

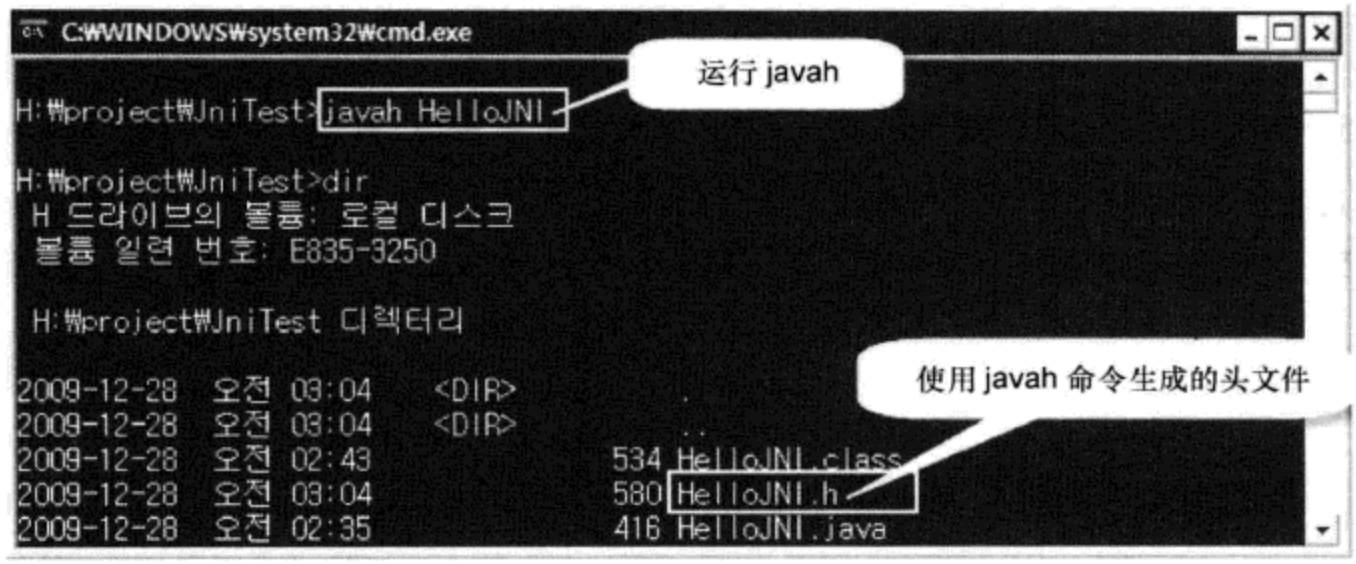


图 4-8 | 由 javah 生成 C 语言头文件

接下来，一起看一下 HelloJNI.h 头文件的内容。如果读者初次接触 JNI，可能会觉得头文件比较复杂，仔细分析一下，其实很简单。

```
/* DO NOT EDIT THIS FILE - it is machine generated */ ←❶
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    printHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_printHello
  (JNIEnv *, jobject);                                    ←❷
```

更多更新编程资源尽在 http://www.pin5i.com/

```
/*
 * Class:     HelloJNI
 * Method:    printString
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_HelloJNI_printString
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

代码 4-3 | HelloJNI.h-由 javah 命令生成头文件

❶ 该头文件由 javah 命令生成，为了保证 JNI 正常运行，请不要直接修改本文件的内容。JNI 开发者只要使用 C/C++语言实现❷中的函数即可。

❷ 这是 javah 命令生成的两个 C 函数原型，函数原型在 Java 类（在 javah 命令行参数中指定）中声明的本地方法的基础上生成。查看各函数原型的注释，可以看到与各函数原型对应的 Java 代码中的本地方法，注释中标明了三个元素：类名、本地方法名、本地方法签名。只要生成了这样的函数原型，Java 虚拟机就能把本地库函数与 Java 本地方法正常地链接在一起。

接下来，分析一下函数原型。JNIEXPORT、JNICALL 都是 JNI 关键字，表示此函数要被 JNI 调用，函数原型中必须有这两个关键字，JNI 才能正常调用函数。其实，JNIEXPORT、JNICALL 两个关键字都是宏定义，它们被定义在<JDK_HOME>/include/win32/jni_md.h 文件（在 Windows 平台下）中。

在 javah 命令生成的头文件中，观察函数原型名称，可以发现函数名遵循一定的命名规则，如图 4-9 所示，JNI 支持的函数命名形式为“Java_类名_本地方法名”。了解这些命名规则，通过函数命名即可推断出 JNI 本地函数与哪个 Java 类的哪个本地方法相对应。比如 Java_HelloJNI_printHello()函数原型，通过其名称，可以知道它与 HelloJNI 类中的 printHello()方法相对应。

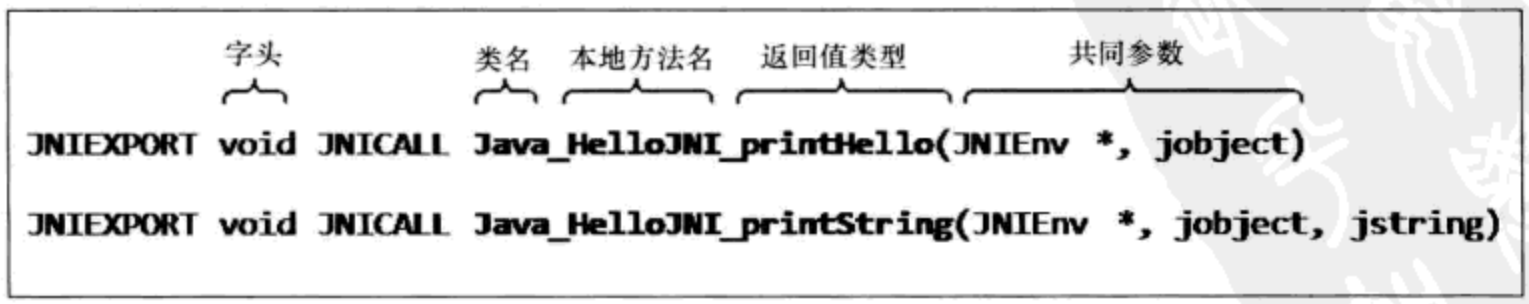


图 4-9 | javah 生成的函数原型

接下来，了解一下函数原型的参数。在生成的函数原型中，带有两个默认参数，第一

更多更新编程资源尽在 http://www.pin5i.com/

个、第二个分别为 JNIEnv *与 jobject[1]，支持 JNI 的函数必须包含这两个共同参数。第一个参数 JNIEnv *是 JNI 接口的指针，用来调用 JNI 表中的各种 JNI 函数，如图 4-10 所示。这里的 JNI 函数（非 JNI 本地函数）是指 JNI 中提供的基本函数集，用来在 JNI 本地函数中创建 Java 对象或调用相应方法，具体内容在下一节中介绍。

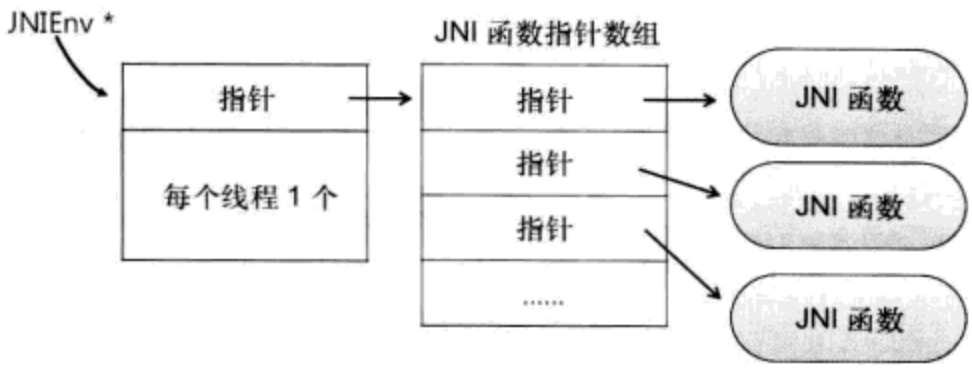


图 4-10 | JNI 接口指针 JNIEnv*

函数原型的第二个默认参数类型 jobject 也是 JNI 提供的 Java 本地类型（参考 TIP 内容），用来在 C 代码中访问 Java 对象，此参数中保存着调用本地方法的对象的一个引用。

例如，在代码 4-1 的❸中，myJNI 对象调用了本地方法，所以上面两个 JNI 本地函数的第二个参数 jobject 中保存着对 myJNI 对象的引用。

TIP Java 本地类型

Java 是一种与平台无关的语言，其数据类型在任何平台下占用相同的内存空间，比如，Java 的 int 类型在所有平台下都占用 4 个字节。但在 C/C++这类本地语言中，即使同一数据类型，在不同的平台下占用不同的内存空间。

在 JNI 编程中，Java 程序与 C/C++函数间经常进行数据交换，如果不提供一种方法消除两种语言的数据类型的差异，那么程序就无法正常运行，运行的可靠性更无法得到保障。幸运的是 JNI 提供了一套与 Java 数据类型相对应的 Java 本地类型，使得本地语言可以使用 Java 数据类型，如下表所示。C/C++开发人员在进行 JNI 编程，从 Java 代码中接收或传递数据（参数，函数返回值）时，只要使用 Java 本地类型即可。

Java 类型	Java 本地类型	占用内存大小
byte	jbyte	1
short	jshort	2
int	jint	4
long	jlong	8

1　若本地方法是静态方法（static）时，第二个参数类型为 jclass，相关内容在后面讲解。

更多更新编程资源尽在 http://www.pin5i.com/

Java 类型	Java 本地类型	占用内存大小
float	jfloat	4
double	jbouble	8
char	jchar	2
boolean	jboolean	1
void	void	

以上 Java 本地类型被定义在下面的头文件中，感兴趣的读者，可以查看一下。

```
<JDK_HOME>/include/jni.h
<JDK_HOME>/include/platform/jni_md.h
```

此外，Java 本地类型也提供了另外三种类型，分别对应于 Java 类、对象与字符串三种引用数据类型。

Java 引用类型	Java 本地类型
类	Jclass
对象	Jobject
String	Jstring

使用 javah 命令生成的函数原型的第二个参数或者是 jobject 类型，或者是 jclass 类型。在 JNI 本地函数中，使用这种类型的参数，即可操控 Java 类或对象。除此之外，还有几种引用类型，它们已超出了本书的讨论范围，感兴趣的读者，可以到 http://java.sun.com/docs/books/jni/查阅相关资料。

函数原型的第三个参数与本地方法的参数类型类似，由 javah 命令根据 Java 类本地方法参数的类型生成，如图 4-11 所示，使用 javah 命令生成 C 函数原型。在执行 javah 命令生成 C 函数原型时，根据 HelloJNI.java 的本地方法 printHello()与 printString(String str)生成两个参数类型 JNIEnv *与 jobject。由于 printString（String str）带有 String 类型的参数，在生成的 C 函数原型中添加一个 jstring 类型的参数与其相对应。

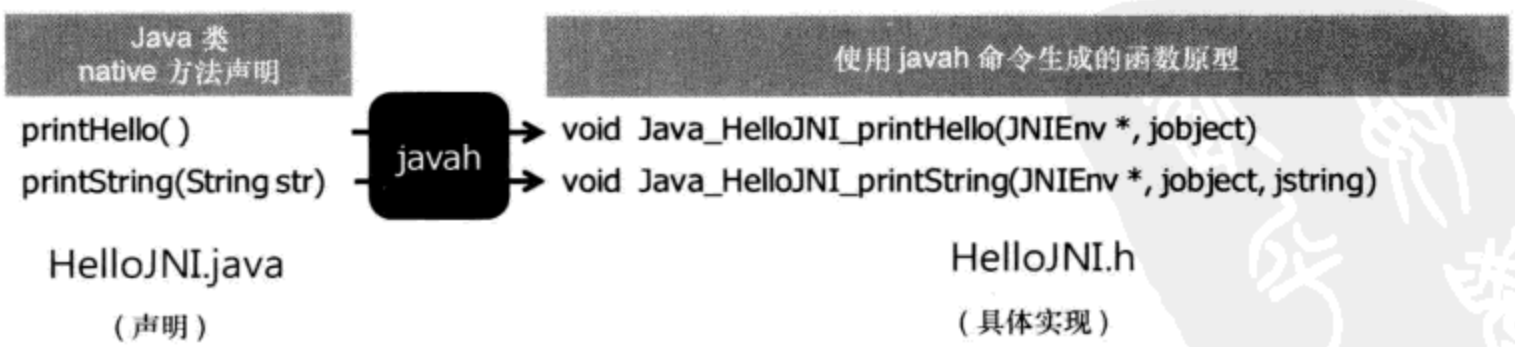


图 4-11 | 使用 javah 命令，生成与 HelloJNI.java 本地方法相对应的 JNI 本地函数原型

第四步：编写 C/C++代码

在 C 函数原型生成后，开始编写 hellojni.c 文件，具体实现 JNI 本地函数。首先

更多更新编程资源尽在 http://www.pin5i.com/

把定义在 HelloJNI.h 头文件中的函数原型复制到 hellojni.c 文件中。注意，在使用 javah 命令生成的头文件中，函数的参数仅指定了参数的类型，并未给出参数的名称。因此，复制完函数原型，开始实现 C 函数时，必须先在参数类型后指定具体的参数名称。

代码 4-4 是编写好的 hellojni.c 文件的代码清单。

```
#include "HelloJNI.h"
#include <stdio.h>

// 添加名称为 env 与 obj 的两个参数
JNIEXPORT void JNICALL Java_HelloJNI_printHello(JNIEnv *env, jobject obj)  ←❶
{
   printf("Hello World!\n");
   return;
}

JNIEXPORT void JNICALL Java_HelloJNI_printString(JNIEnv *env, jobject obj,
                                    jstring string)      ←❷
{
   // 将 Java String 转换为 C 字符串
   const char *str = (*env)->GetStringUTFChars(env, string, 0);    ←❸
   printf("%s!\n", str);

   return;
}
```

代码 4-4 | hellojni.c 源代码

分析一下主要代码。

❶ Java_HelloJNI_printHello()函数对应于 HelloJNI 类中的 printHello()本地方法，用于在控制台输出“Hello World!”。

❷ Java_HelloJNI_printString()函数接收 printString()本地方法传递过来的字符串参数，并输出到控制台上。printString()本地方法传递过来的字符串保存在 Java_HelloJNI_printString()函数的第三个参数 string（参数类型为 jstring）中。

如前所述，jstring 是 Java 本地类型，它对应于 Java 中的 String 类型，在内存中占用 16 位。而 C 语言中的字符串仅占用 8 位，在 C 语言中无法直接使用 jstring 类型的字符串，因此需要将 jstring 类型的字符串转换成 C 语言字符串，转换过程在❸中介绍。

❸ GetStringUTFChars()是 JNI 函数，用来将 Java 字符串转换成 C 语言字符串。使用此函数转换字符串之后，再使用 printf()函数输出，即可将本地方法传递过来的字符串输出到控制台上。

更多更新编程资源尽在 http://www.pin5i.com/

JNI 提供了多种 JNI 函数，用来处理 C 字符串与 Java 字符串的转换。若想查阅这些函数，请参考 http://java.sun.com/docs/books/jni/下的相关文档。GetStringUTFChars()函数通过 JNI 接口指针 JNIEnv *env 进行调用，如代码❸所示。

JNI 函数-GetStringUTFChars	
形式	const jbyte*GetStringUTFChars(JNIEnv *env, jstring string, jboolean *isCopy)
说明	将 Java 字符串对象转换成 UTF-8 字符串（C 字符串），并返回指针
参数	env-JNI 接口指针 string-Java 字符串对象 isCopy-当 String 对象中的字符串被转换成 UTF-8 字符串，被复制到内存，且指针被返回时，*isCopy 设置为 JNI_TRUE，否则设置为 JNI_FALSE。

第五步：生成 C 共享库

经过步骤 3 与 4，C 语言头文件与函数实现都已经完成，接下来，开始生成 C 共享库。由于笔者在 Windows 环境下，因而要使用微软的 Visual C++来构建代码，生成运行库。若读者的电脑中尚未设置 Visual C++，请点击以下链接，去微软公司下载页面，下载免费的 Visual C++ 2008 Express Editions，安装到你的电脑中。

http://www.microsoft.com/express/download/

安装好 Visual C++ 2008 Express Editions 之后，为了使用 Visual C++进行编译，执行 Visual Studio 2008 Command Prompt 命令，如图 4-12 所示。

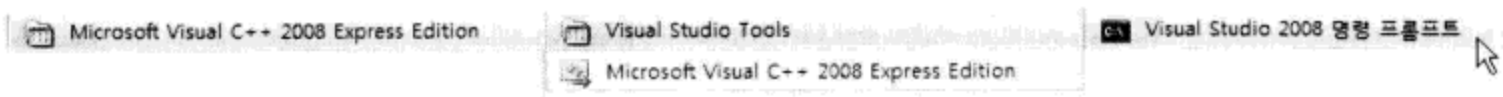


图 4-12 | 运行 Visual Studio 2008 Command Prompt 命令

在 Command Prompt 中，输入如下编译命令，生成共享运行库（<JDK_HOME>是 JDK 的设置根目录，请根据各自设置进行输入）。

```
cl -I"<JDK_HOME>\include" -I"<JDK_HOME>\include\win32" -LD hellojni.c -Fehellojni.dll

cl : visual c++ 编译命令
-I<dir> : 添加要检索头文件的目录路径 <dir>
      为了检索头文件，添加如下目录
        jni.h (<JDK_HOME>\include)
        jni_md.h (<JDK_HOME>\include\win32)
-LD : 创建 DLL
-Fe< 文件名 > : 指定编译结果文件名称
```

运行以上命令，生成 hellojni.dll 运行库，如图 4-13 所示。

更多更新编程资源尽在 http://www.pin5i.com/

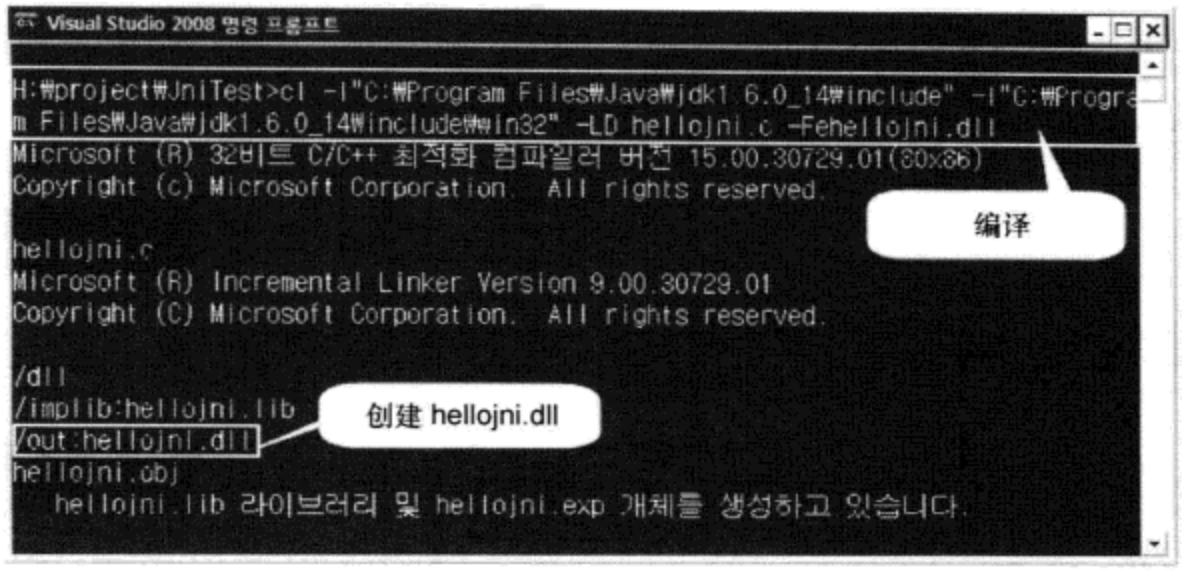


图 4-13 | 生成 hellojni.dll

第六步：运行 Java 程序

至此，所有准备步骤都完成，执行 java 命令，运行 HelloJNI 类，并查看输出结果是否正常，如图 4-14 所示。

图 4-14 | 运行 HelloJNI

4.2.2　小结

通过以上学习，我们掌握了 Java 本地方法如何通过 JNI 链接至 C 函数，以及如何具体实现等相关知识。

最后，做个小结，梳理一下，如图 4-15 所示。

（1） 在 Java 类中声明本地方法

（2） 使用 javah 命令，生成包含 JNI 本地函数原型的头文件

（3） 实现 JNI 本地函数

（4） 生成 C 共享库

（5） 通过 JNI，调用 JNI 本地函数

更多更新编程资源尽在 http://www.pin5i.com/

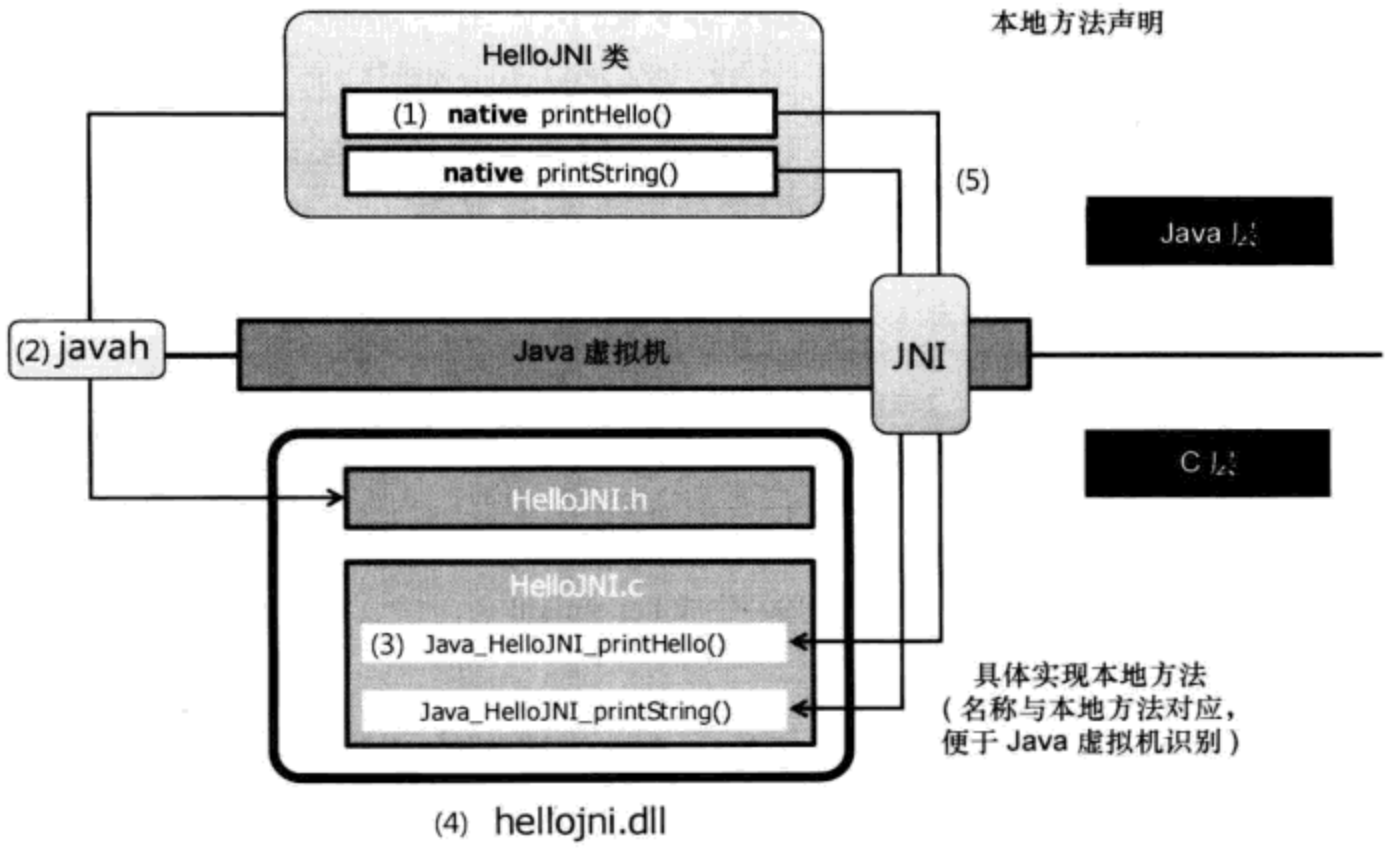


图 4-15 | 通过 JNI 调用 JNI 本地函数

4.3 调用 JNI 函数

在上一节示例中，讲解了 JNI 的运行方式，本节将讲解在由 C 语言编写的 JNI 本地函数中如何控制 Java 端的代码，主要包括以下内容。

- 创建 Java 对象
- 访问类静态成员域
- 调用类的静态方法
- 访问 Java 对象的成员变量
- 访问 Java 对象的方法

4.3.1 调用 JNI 函数的示例程序结构

在进入正文之前，先大致看一下示例程序的整体结构，如图 4-16 所示。整个示例程序由 JniFuncMain 类（包含本地方法声明）、JniTest 对象、jnifunc.dll（包含本地方法的具体实现）三部分组成。

更多更新编程资源尽在 http://www.pin5i.com/

整个示例程序从调用 JniFuncMain 类的 createJniObject()本地方法开始，如图 4-16（1）所示。该方法经由 JNI 与 jnitest.dll 中命名为 Java_JniFuncMain_createJniObject()的 C 函数链接在一起。

Java_JniFuncMain_createJniObject()函数以创建 Java 对象或调用方法的方式与 Java 代码相互作用，如图 4-16 的（2）～（6）所示。在这一过程中，JNI 机制提供了多种 JNI 函数进行支持。本节主要讲解如何通过由 C 语言实现的 JNI 本地函数来灵活使用 Java 代码。

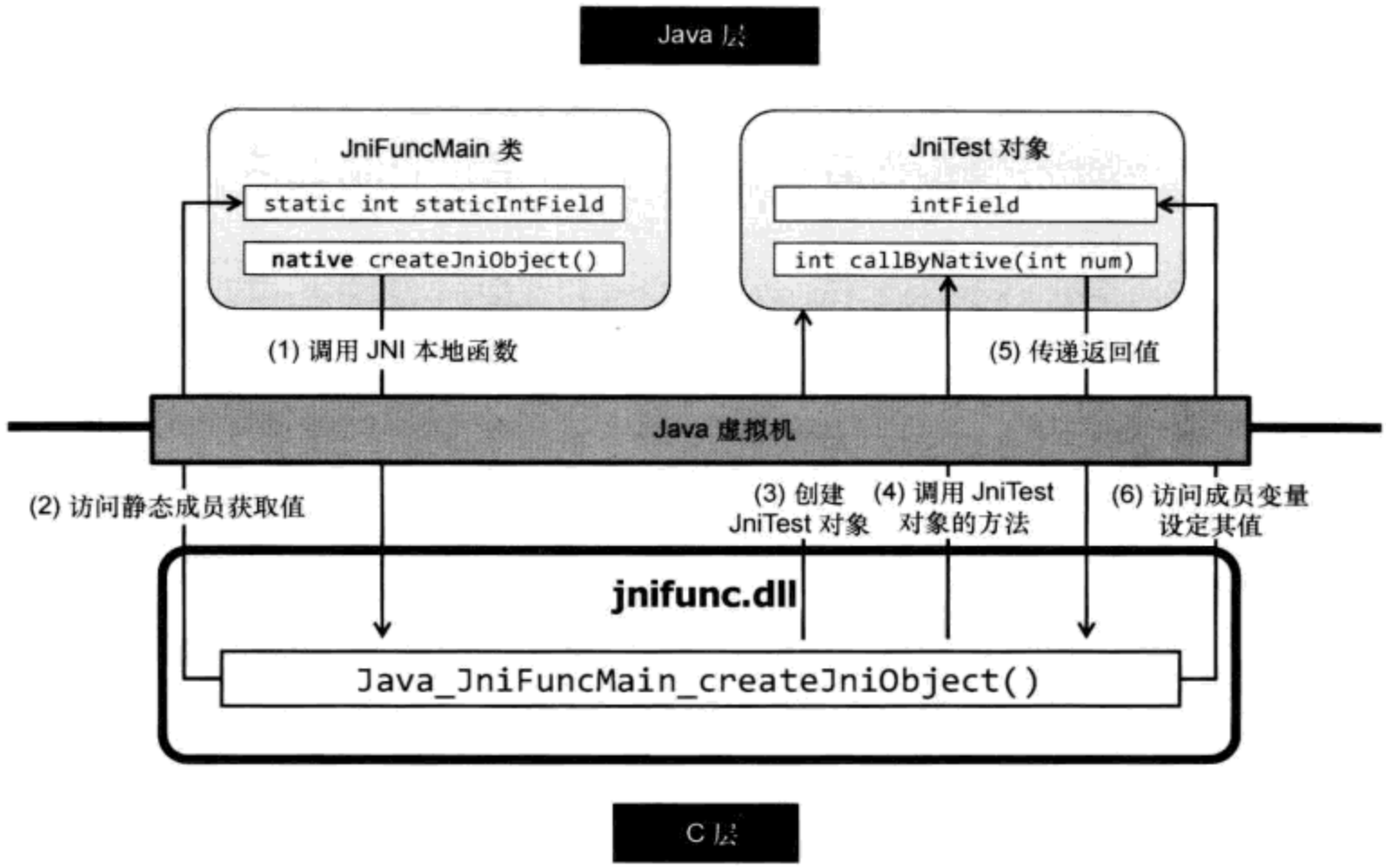


图 4-16 | 使用 JNI 函数的示例程序运行图

下面即将讲解的示例由 Java 与 C 代码混合而成，运行时两者相互协调，共同完成指定的任务。了解代码的运行情况，就会明白各模块间是如何相互作用的。

4.3.2　Java 层代码（JniFuncMain.java）

1．JniFuncMain 类

首先分析示例主程序 JniFuncMain 类的代码，如代码 4-5 所示。

更多更新编程资源尽在 http://www.pin5i.com/

```
public class JniFuncMain
{
   private static int staticIntField = 300;

   // 加载本地库 jnifunc.dll
   static { System.loadLibrary("jnifunc"); }      ←❶

   // 本地方法声明
   public static native JniTest createJniObject();   ←❷

   public static void main(String[] args)
   {
      // 从本地代码生成 JniTest 对象
      System.out.println("[Java] createJniObejct() 调用本地方法");
      JniTest jniObj = createJniObject();      ←❸

      // 调用 JniTest 对象的方法
      jniObj.callTest();          ←❹
   }
}
```

代码 4-5 | JniFuncMain.java 中的 JniFuncMain 类

❶ 通过 Java 静态块，在调用本地方法前，加载 jnifunc.dll 运行库。

❷ 与前面代码 4-1 不同，使用 static 关键字，声明本地方法 createJniObject()。在调用此方法时，不需要创建对象，直接通过 JniFuncMain 类调用即可，如代码行❸所示。

❸ 此行代码实现图 4-16（3）的功能，即不用 Java 语言的 new 运算符，调用与 createJniObject()本地方法相对应的 C 函数生成 JniTest 类的对象，再将对象的引用保存在 jniObj 引用变量中。

❹ 通过❸中生成的 jniObj 对象，调用此对象的 callTest()方法。

2．JniTest 类

请看下面代码 4-6，它是 JniTest 类的代码清单。如图 4-16 所示，程序会先调用 JNI 本地函数 Java_JniFuncMain_createJniObject()生成 JniTest 对象，再调用 callByNative()方法。JniTest 类的代码比较简单，如代码 4-6 所示，在此不再分析说明。

```
class JniTest
{
    private int intField;

    // 构造方法
```

更多更新编程资源尽在 http://www.pin5i.com/


```
    public JniTest(int num)
    {
        intField = num;
        System.out.println(`?Java] 调用 JniTest 对象的构造方法: intField = " + intField);
    }

    // 此方法由 JNI 本地函数调用
    public int callByNative(int num)
    {
        System.out.println ("[Java] JniTest 对象的 callByNative (" + num + ") 调用 ");
        return num;
    }

    public void callTest()
    {
        System.out.println ("[Java] JniTest 对象的 callTest ( ) 方法调用 : intField = "+intField);

    }
}
```

代码 4-6 | JniFuncMain.java 源代码中 JniTest 类

4.3.3　分析 JNI 本地函数代码

1．JniFuncMain.h 头文件

在 JniFuncMain.java 代码中，仅仅对本地方法 createJniObject()进行了声明，接下来，应该用 C 语言实现它。首先，使用 javah 命令，生成本地方法的函数原型，具体命令如下（JniFuncMain.java 文件在前面已经编译过）。

```
javah JniFuncMain
```

执行上述命令后，生成如下 JniFuncMain.h 头文件。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JniFuncMain */

#ifndef _Included_JniFuncMain
#define _Included_JniFuncMain
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JniFuncMain
 * Method:    createJniObject
 * Signature: ()LJniTest;
 */
```

更多更新编程资源尽在 http://www.pin5i.com/

```
JNIEXPORT jobject JNICALL Java_JniFuncMain_createJniObject
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

代码 4-7 | 使用 javah 命令，生成 JniFuncMain.h 头文件

查看 JniFuncMain.h 头文件，可以看到 createJniObject()本地方法对应的 JNI 本地函数原型，形式如下。

```
JNIEXPORT jobject JNICALL Java_JniFuncMain_createJniObject
        (JNIEnv *, jclass )
```

由 JNI 命名规则可知，此函数原型与 JniFuncMain 类的 createJniObject()本地方法相对应。在前面的学习中，我们知道使用 javah 命令生成的 JNI 本地函数原型的第二个参数默认类型为 jobject。但观察代码 4-7 中的函数原型，会发现它的第二个参数类型为 jclass，而不是 jobject。原因是什么？若想弄清楚这个问题，首先要理解函数原型第二个参数的含义。在前面的学习中，曾经提到过 jobject 类型的变量用来保存调用本地方法的对象的引用。

让我们再看看代码 4-7 中本地方法的声明，如下所示。

```
public static native JniTest createJniObject();
```

如你所见，此本地方法被声明为 static（静态）方法。在 Java 中，调用静态方法时，可以不用创建对象，通过所在的类直接调用即可，即本地静态方法通过类而非对象进行调用，因此函数原型的第二个参数类型为 jclass 类型。

在代码 4-5❸中，你可以看到调用 createJniObject()本地方法是通过 JniFuncMain 类进行的，JNI 本地函数 Java_JniFuncMain_createJniObject()的第二个参数保存的是 JniFuncMain 类的引用。

2．jnifunc.cpp 文件

使用 javah 命令生成头文件后，接下来，开始实现函数原型。代码 4-8 是实现 JNI 本地函数 Java_JniFuncMain_createJniObject()的源码。代码虽然简单，但却能准确地说明 JNI 函数如何使用 Java 的组成元素，请参考如图 4-16 所示的动作流程，分析一下代码。

```
JNIEXPORT jobject JNICALL Java_JniFuncMain_createJniObject(JNIEnv *env, jclass clazz)
{
    jclass  targetClass;
    jmethodID  mid;
    jobject  newObject;
```

更多更新编程资源尽在 http://www.pin5i.com/


```
jstring  helloStr;
jfieldID  fid;
jint  staticIntField;
jint  result;

// 获取 JniFuncMain 类的 staticIntField 变量值
fid = env->GetStaticFieldID(clazz, "staticIntField", "I");
staticIntField = env->GetStaticIntField(clazz, fid);
printf ("[CPP] 获取 JniFuncMain 类的 staticIntField 值 \n");
printf("       JniFuncMain.staticIntField = %d\n", staticIntField);

// 查找生成对象的类
targetClass = env->FindClass("JniTest");

// 查找构造方法
mid = env->GetMethodID(targetClass, "<init>", "(I)V");

// 生成 JniTest 对象 ( 返回对象的引用 )
printf ("[CPP] JniTest 对象 生成 \n");
newObject = env->NewObject(targetClass, mid, 100);

// 调用对象的方法
mid = env->GetMethodID(targetClass, "callByNative", "(I)I");
result = env->CallIntMethod(newObject, mid, 200);

// 设置 JniObject 对象的 intField 值
fid = env->GetFieldID(targetClass, "intField", "I");
printf ("[CPP] 设置 JniTest 对象的 intField 值为 200\n");
env->SetIntField(newObject, fid, result);

    // 返回对象的引用
    return newObject;
}
```

代码 4-8 | jnifunc.cpp 源码

参考图 4-16（2）～（6），逐步分析代码 4-8 中的各行代码。

3．通过 JNI，获取成员变量值

下面代码用于获取 JniFuncMain 类的 staticIntField 成员变量的值，对应于图 4-16 中的（2）。

```
// 1. 查找含有待访问成员变量的 JniFuncMain 类的 jclass 值

// 2. 获取 staticIntField 变量的 ID 值
fid = env->GetStaticFieldID(clazz, "staticIntField", "I"); ←❶

// 3. 读取 jclass 与 fieldid 指定的成员变量值
staticIntField = env->GetStaticIntField(clazz, fid);  ←❷
```

代码 4-9 | jnifunc.cpp 代码中用于获取 JniFuncMain 类的成员变量的值

更多更新编程资源尽在 http://www.pin5i.com/

程序通过 JNI 访问 Java 类/对象的成员变量按如下顺序进行。

（1）查找含待访问的成员变量的 Java 类的 jclass 值。

（2）获取此类成员变量的 jfieldID 值。若成员变量为静态变量，则调用名称为 GetStaticFieldID()的 JNI 函数；若待访问的成员变量是普通对象，则调用名称为 GetFieldID()的 JNI 函数。

（3）使用 1, 2 中获得的 jclass 与 jfieldID 值，获取或设置成员变量值。

依据以上顺序，分析代码 4-9，待读取数值的 staticIntField 成员变量在 JniFuncMain 类被声明。JniFuncMain 类的 jclass 值被传递给 JNI 本地函数 Java_JniFuncMain_createJniObject()的第二个参数中，若想获取指定类的 jclass 值，调用 JNI 函数 FindClass()即可。

❶ 若想在本地代码中访问 Java 的成员变量，必须获取相应成员变量的 ID 值。如代码 4-9 所示，成员变量的 ID 保存在 jfieldID 类型的变量中。由于待读取数值的 staticIntField 成员变量是 JniFuncMain 类的静态成员变量，在获取 staticIntField 的 ID 时，应调用名称为 GetStaticFieldID()的 JNI 函数。

若想获取普通对象中的非静态成员变量的 ID，应调用名称为 GetFieldID()的 JNI 函数。

细心的读者可能发现，在代码 4-9❶中调用的 GetStaticFieldID()函数，与下表中的 GetStaticFieldID()函数原型有些不同，函数原型中带有四个参数，而代码中仅有三个，缺少了 env 参数，部分读者可能误以为代码编写错误，其实这与所用的编程语言（C 或 C++）相关。具体内容请参考后面 Tip 中关于 JNI 函数编码风格（C 风格与 C++风格）的说明。

JNI 函数- GetStaticFieldID()	
形式	jfield GetStaticFieldID(JNIEnv *env,jclass clazz,const char *name,const char *signature)
说明	返回指定类的指定的静态成员变量的 jfieldID 的值
参数	env-JNI 接口指针 clazz-包含成员变量的类的 jclass name-成员变量名 signature-成员变量签名

JNI 函数- GetFieldID()	
形式	jfield GetFieldID(JNIEnv *env,jclass clazz,const char *name,const char *signature)
说明	返回对象中指定的成员变量的 jfieldID 的值
参数	env-JNI 接口指针 clazz-包含成员变量的类的 jclass name-成员变量名 signature-成员变量签名

更多更新编程资源尽在 http://www.pin5i.com/

以上两个函数都要求提供成员变量的签名。成员变量与成员方法都拥有签名，使用<JDK_HOME>\bin 目录的 javap 命令（Java 反编译器），即能轻松地获取成员变量或成员方法签名。

TIP 在 JNI 中获取成员变量或成员方法签名

在调用某些 JNI 函数时，要求提供指定的成员变量或成员方法的签名。当然，开发者可以根据 JNI 规范中的 Java 签名生成规则，直接创建签名。但不建议大家这么做，Java 系统会为类的成员变量或成员方法生成签名。使用时，只需使用 javap 命令（Java 反编译器），即可轻松获取指定的成员变量或成员方法的签名。

形式：javap [选项] '类名'

选项：-s 输出 Java 签名

　　-p 输出所有类及成员

请看图 4-17，它向大家展示了使用 javap 命令，获取 JniTestMain 类的 staticIntField 成员变量的签名的过程。staticIntField 成员变量是 int 类型，所以其签名是代表 int 类型的 I。关于 Java 签名的详细内容，请参考 JNI 的相关文档。

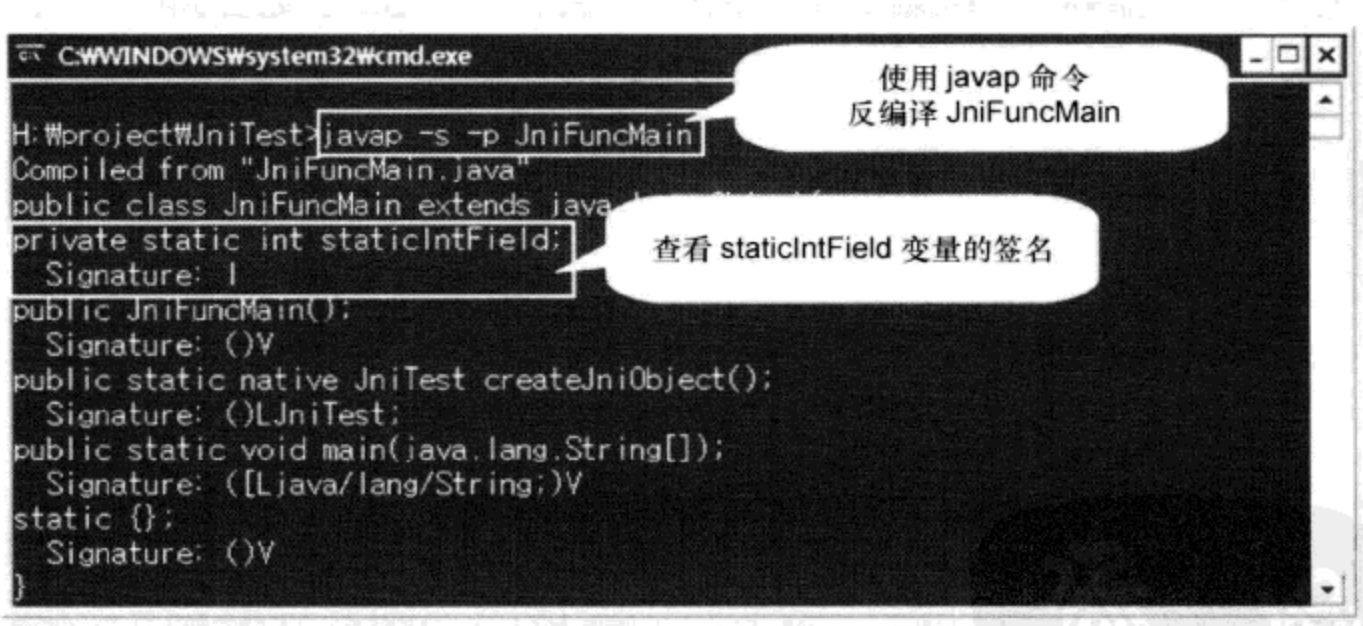


图 4-17 | 使用 javap 获取成员变量或成员方法的签名

❷ 在获取成员变量所在的类与 ID 后，根据各成员变量的类型与存储区块（static 或 non-static），调用相应的 JNI 函数读取成员变量值即可。在 JNI 中有两种函数用来获取成员变量的值，分别为 Get<type>Field 函数与 GetStatic<type>Field 函数（<type>指 Int、Char、Double 等基本数据类型，具体参考 JNI 文档）。

更多更新编程资源尽在 http://www.pin5i.com/

TIP JNI 函数编码风格（C 风格与 C++风格）

在实现 JNI 本地函数，调用 JNI 函数时，C 与 C++的调用方式略有不同。以代码 4-9❶中调用 GetStaticFieldID()函数的方式为例，说一下两种语言调用方式的不同。

查看 GetStaticFieldID()函数原型，可以看到此函数总共带有 4 个参数，如下。

```
jfield GetStaticFieldID(JNIEnv *env,jclass clazz,const char *name,const char *signature)
```

如代码 4-9❶所示，在 C++代码中，调用 GetStaticFieldID()函数时，只需要传入除第一个参数 env 之外的另外三个参数即可。

```
fid=env-> GetStaticFieldID(clazz,"staticIntField","I");  //在 C++中调用 JNI 函数
```

而在 C 语言中调用 GetStaticFieldID()函数时，第一个参数 env 仍然要传入，如下所示。

```
fid=(*env)-> GetStaticFieldID(env,clazz,"staticIntField","I");  //在 C 中调用 JNI 函数
```

这些差异在 GetStaticFieldID()等 JNI 函数中普遍存在。在实现本地代码时，要根据所用的语言的不同（C 或 C++），选用恰当的方式，调用 JNI 函数。

JNI 函数-GetStatic<type>Field	
形式	<jnitype> GetStatic<type>Field(JNIEnv *env,jclass clazz,jfieldID fieldID)
说明	返回 clazz 类中 ID 为 fieldID 的静态变量的值
参数	env-JNI 接口指针 clazz-包含成员变量的类 fieldID-成员变量的 ID
参考	<type>指 Object、Boolean、Byte、Char、Short、Int、Long、Float、Double 九种基本类型。 返回类型<jnitype>指 jobject、jboolean、jbyte、jchar、jshort、jint、jlong、jfloat、jdouble 九种基本类型。 这些类型也被应用到其他 JNI 函数的<type>中
返回值	返回静态成员变量的值

JNI 函数-Get<type>Field	
形式	<jnitype> Get<type>Field(JNIEnv *env,jobject obj,jfieldID fieldID)
说明	返回 obj 对象中 ID 为 fieldID 的成员变量的值
参数	env-JNI 接口指针 obj-包含成员变量的对象 fieldID-成员变量的 ID
返回值	返回成员变量的值

更多更新编程资源尽在 http://www.pin5i.com/

由于 staticIntField 是 int 类型的静态成员变量，所以调用 GetStaticFieldID()函数即可获取 staticIntField 的值。

生成对象

在 JNI 本地函数中如何生成 Java 类对象呢？如代码 4-10 所示，对应于图 4-16（3）。

```
// 1. 查找生成对象的类
targetClass = env->FindClass("JniTest");          ❶

// 2. 查找类的构造方法
mid = env->GetMethodID(targetClass, "<init>", "(I)V");   ❷

// 3. 生成 JniTest 类对象（返回对象引用）
newObject = env->NewObject(targetClass, mid, 100);   ❸
```

代码 4-10 | jnifunc.cpp 中调用 JNI 函数生成 Java 对象

通过 JNI 函数，生成 Java 对象的顺序如下所示。

1. 查找指定的类，并将查找到的类赋给 jclass 类型的变量。
2. 查找 Java 类构造方法的 ID 值（类型为 jmethodID）。
3. 生成 Java 类对象

乍一看有些复杂，但与编写类并调用类的构造方法，及创建对象的过程差不多。分析一下代码。

❶ 首先调用 JNI 函数 FindClass()，查找生成对象的类。由于示例中要生成 JniTest 类的对象，所以将“JniTest”类名作为 FindClass()函数参数，查找并获得 jclass 值。

JNI 函数-FindClass	
形式	jclass FindClass(JNIEnv *env,const char *name)
说明	查找 name 指定的 Java 类
参数	env-JNI 接口指针 name-待查找的类名
返回值	返回类的 jclass 值

❷ 此行代码用来获取 JniTest 类的构造方法的 ID，并保存在 jmethodID 变量中。在 JNI 本地函数中，若想使用 Java 的方法，必须先获取该方法的 ID（jmethodID 类型）。在 JNI 函数中有一个 GetMethodID()函数用来获取指定类的指定方法

更多更新编程资源尽在 http://www.pin5i.com/

ID。此函数除了可以用来获取指定类的构造方法的 ID 外，还可以获取类的其他方法的 ID。若指定的方法是静态方法，则可以调用 JNI 函数中的 GetStaticMethodID()函数，获得指定静态方法的 ID。

示例中，在生成指定类的对象之前，需要先调用 GetMethodID()函数获取该类构造方法的 ID。在调用 GetMethodID()函数时，除了提供生成对象的类，还要提供类的构造方法名称（类的构造方法名称为“<init>”，其他非构造方法，直接提供方法名即可），以及构造方法的签名（构造方法的签名为“(I)V”）。对于方法的签名，使用 javap 工具即可轻松获取，如图 4-18 所示。

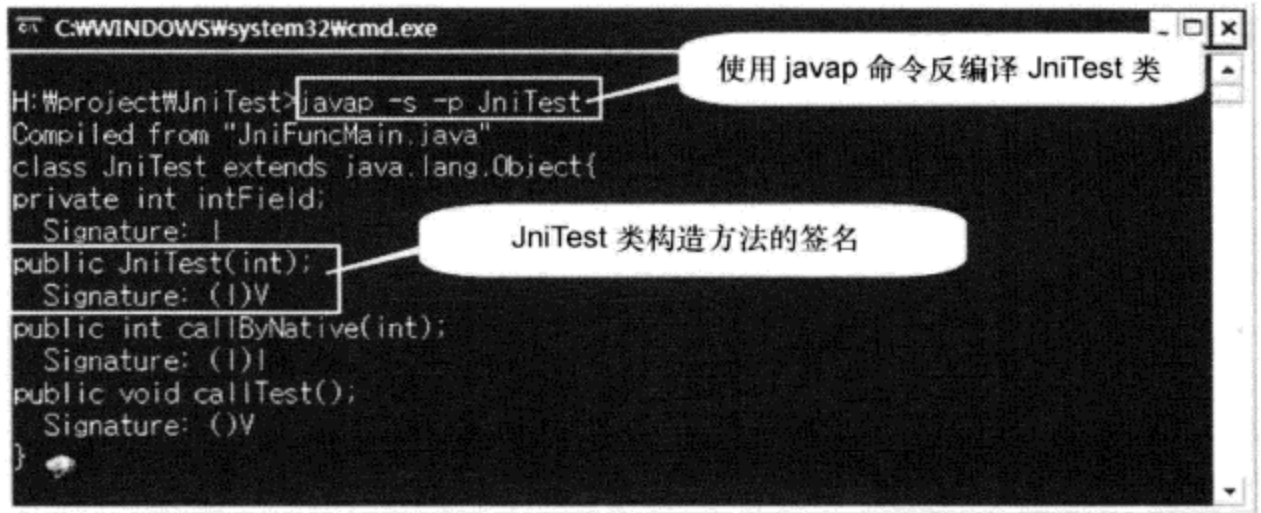


图 4-18 | 使用 javap 工具获取 JniTest 类构造方法的签名

JNI 函数-GetMethodID	
形式	jmethodID GetMethodID(JNIEnv *env,jclass clazz,const char *name,const char *signature)
说明	获取 clazz 类对象的指定方法的 ID。注意，方法名（name）与签名应当保持一致。若获取类构造方法的 ID，方法名应为“<init>”
参数	env：JNI 接口指针 clazz：Java 类 name：方法名 signature：方法签名
返回值	若方法 ID 错误，则返回 NULL

❸ 以❶、❷代码中获得的 JniTest 类的 jclass 与构造方法的 ID 为参数，调用 JNI 函数 NewObject()，生成 JniTest 类的对象。JniTest 类的构造方法 JniTest（int num）带有一个 int 类型的参数，在调用 NewObject()时，同时传入 100 这一 int 数据。在生成 JniTest 类的对象后，将对象的引用保存在 jobject 变量中。

更多更新编程资源尽在 http://www.pin5i.com/

JNI 函数-NewObject	
形式	jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, …)
说明	生成指定类的对象。methodID 指类的构造方法的 ID
参数	env：JNI 接口指针 clazz：Java 类 methodID：类的构造方法的 ID …：传递给类构造方法的参数
返回值	返回类对象的引用。若发生错误，返回 NULL

TIP 局部引用与全局引用

在实现 JNI 本地函数时，由 GetObjectClass()、FindClass()等 JNI 函数返回的 jclass、jobject 等引用都是局部引用（Local Reference）。

局部引用是 JNI 默认的，它仅在 JNI 本地函数内才有效，即当 JNI 本地函数返回后，其内部的引用就会失效。

如代码 4-8 所示，targetClass 是 jclass 类型的变量，它保存 FindClass()函数的返回值，是一个指向类的引用，并且该引用是一个局部引用，即 targetClass 中的类引用仅在 Java_JniFuncMain _createJniObject()开始调用到被返回的这一过程中才指向 JniTest 这个类。当函数执行完毕，返回值之后，targetClass 中的类引用才会失效。

在 JNI 编程中，实现 JNI 本地函数时，必须准确地理解局部引用的含义。下面再举一个例子，进一步详细地说明一下。

代码 4-11 是一段较为简单的代码。在 RefTest 类中声明了一个静态成员变量，定义了一个静态成员方法。在 RefTestMain 类中，先加载指定的运行库，再声明一个 getMember()本地方法，然后调用该本地方法，在控制台输出类中的静态成员变量的值。

```
class RefTest
{
   public static int intField;

   public static void setField(int num){
      intField = num;
   }
}

public class RefTestMain
{
   // 加载本地库
   static { System.loadLibrary("reftest"); }
```

更多更新编程资源尽在 http://www.pin5i.com/

```
    // 声明本地方法
    public static native int getMember();

    public static void main(String[] args)  {
        RefTest.setField(100);
        System.out.println("intField = " + getMember());  ←❶
        RefTest.setField(200);
        System.out.println("intField = " + getMember());  ←❷
    }
}
```

代码 4-11 | RefTestMain.java 源码

其中，本地方法getMember()的具体实现在reftest.cpp中，如代码4-12所示。观察代码4-12，可以发现它与代码 4-9 中通过 JNI 函数读取静态成员变量值的代码类似，只是为了说明误用局部引用的问题，在代码 4-12 的❸行中声明了一个静态 jclass 变量 targetClass，准备保存类的引用。

```
static jclass targetClass = 0;          ←❸
JNIEXPORT jint JNICALL Java_RefTestMain_getMember (JNIEnv *env, jclass clazz)
{
    jfieldID  fid;
    jint  intField;
    jclass targetClass;

    if(targetClass == 0)                                          ←❹
        targetClass = env->FindClass("RefTest");
    fid = env->GetStaticFieldID(targetClass, "intField", "I");    ←❺
    intField = env->GetStaticIntField(targetClass, fid);

    return intField;
}
```

代码 4-12 | reftest.cpp 源码-使用局部引用错误

分析一下代码。

❸ 与代码 4-9 不同，在 Reftest.cpp 文件中首先声明了一个静态变量 targetClass，用来保存 RefTest 类的局部引用。由于该变量声明为静态（static），所以它仅被初始化一次，并且其值一直保持到程序终止。

❶ 当在❶行中调用 getMember()本地方法时，由于 targetClass 尚未保存引用值，调用 FindClass()函数获取 RefTest 类的局部引用，而后将获得的类引用保存到静态变量 targetClass 中。当在❷行中再次调用 getMember()本地方法时，静态变量 targetClass 的值非 0，而是之前保存的 RefTest 类的局部引用。

❺ 调用 JNI 函数，获取 RefTest 类的 intField 变量的值。

更多更新编程资源尽在 http://www.pin5i.com/

分别编译以上代码，并运行，会发生如下错误。（关于编译方法，在前面已经讲解过，在此省略）

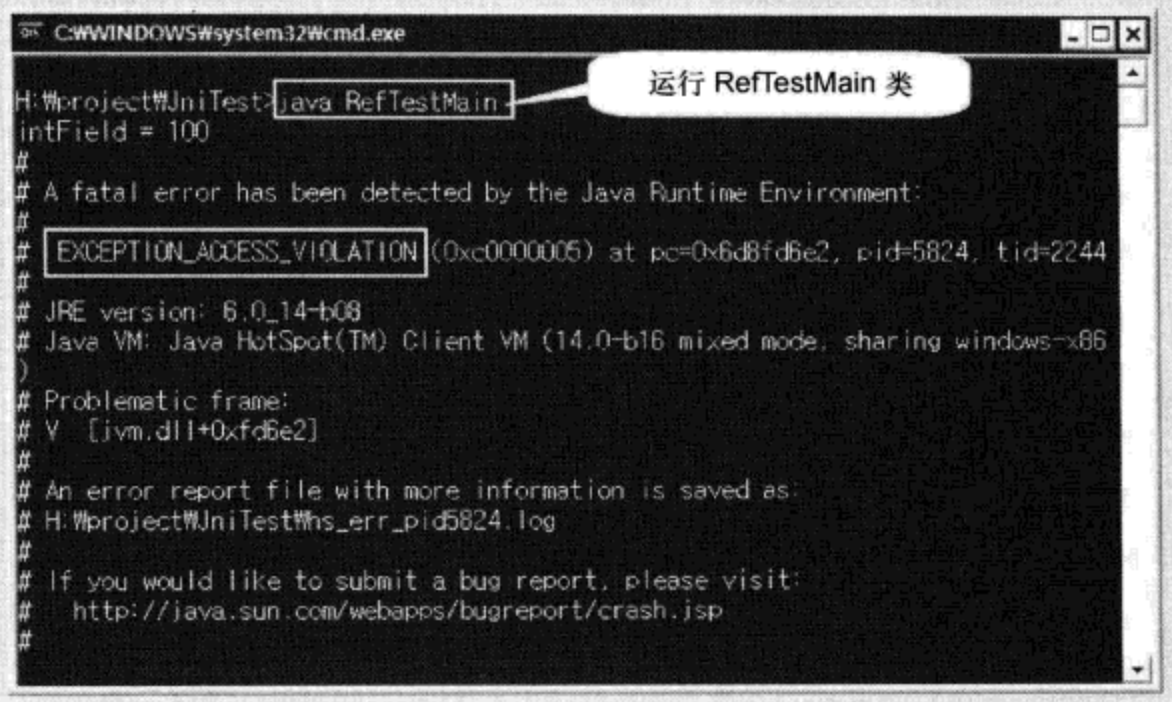


图 4-19 | 运行 RefTestMain 类-因误用局部引用，Java 虚拟机发生错误

首次调用 getMember()方法时，如代码 4-11❶所示，会正常输出“intField=100”。但当再次调用 getMember()方法时，如代码 4-11❷所示，就会发生图 4-19 中的错误。

发生错误的原因在于程序编写违反了局部引用变量的使用规则。如前所述，JNI 中的局部引用在 JNI 本地函数终止时即可无效。若想使用同一引用值，需要再次调用 FindClass()函数，重新获取引用值。但在代码 4-11❷中，却再次使用保存在 static 变量中的值（targetClass 引用在 JNI 本地函数执行完毕即已失效），试图读取类成员变量的值，因而发生图 4-19 中的错误。

为解决这一问题，JNI 提供了一个名称为 NewGlobalRef()的 JNI 函数，用来为指定的类或对象生成全局引用（Global Reference），以便在 JNI 本地函数中在全局范围内使用该引用。

JNI 函数-NewGlobalRef	
形式	jobject NewGlobalRef(JNIEnv *env, jobject obj)
说明	为 obj 指定的类或对象，生成全局引用
参数	env：JNI 接口指针 obj：待生成全局引用的引用值
返回值	返回生成的全局引用，若发生错误，则返回 NULL

请注意，当全局引用使用完毕后，应当调用名称为 DeleteGlobalRef()的 JNI 函数，显性地将全局引用销毁。具体内容，请参看 JNI 相关文档。

下面代码是使用 NewGlobalRef()函数，对代码 4-12 进行的修改。

更多更新编程资源尽在 http://www.pin5i.com/

```
#include "RefTestMain.h"

static  jclass globalTargetClass = 0;

JNIEXPORT jint JNICALL Java_RefTestMain_getMember
    (JNIEnv *env, jclass clazz)
{
   jfieldID  fid;
   jint  intField;
   jclass targetClass;

   if(globalTargetClass == 0) {
      targetClass = env->FindClass("RefTest");
      globalTargetClass = (jclass) env->NewGlobalRef(targetClass);    ❻
   }

   fid = env->GetStaticFieldID(globalTargetClass, "intField", "I");
   intField = env->GetStaticIntField(globalTargetClass, fid);

   return intField;
}
```

代码 4-13 | reftest.cpp 源码-JNI 全局引用示例

与代码 4-12 比较，可以发现在❻行中调用了 NewGlobalRef()函数，将 targetClass 中保存的 RefTest 类的局部引用（由 FindClass()函数返回）转换成全局引用。并且将生成的全局引用保存在 globalTargetClass 静态变量中，以便在整个 reftest.cpp 源码中使用。

targetClass 变量保存的是局部引用，当函数执行完成后，该局部引用即无效。而 globalTargetClass 变量中保存的是全局引用，除非调用 DeleteGlobalRef()明确将其销毁，不然这个全局引用总是有效的，可以在运行库的其他函数中使用该引用。

使用上述代码，重新生成 reftest.dll，再运行 RefTestMain 类，即可得到正常运行结果，如下图所示。

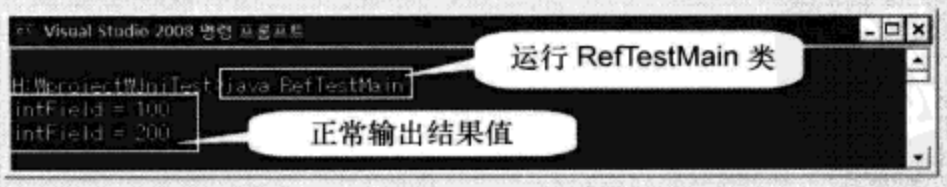


图 4-20 | 运行 RefTestMain 类-使用全局引用输出正常结果

4．调用 Java 方法

请看如下代码 4-14，该段代码描述了在 JNI 本地函数中，如何使用 JNI 函数调用 Java 方法，并将返回值保存至 JNI 本地函数的变量中的过程，对应于图 4-16 的（4）～（5）。

更多更新编程资源尽在 http://www.pin5i.com/


```
// 1. 获取含待调方法的 Java 类的 jclass
targetClass = env->GetObjectClass(newObject);          ❶

// 2. 获取待调方法的 ID
mid = env->GetMethodID(targetClass, "callByNative", "(I)I");          ❷

// 3. 调用 Java 方法 保存返回值
result = env->CallIntMethod(newObject, mid, 200);          ❸
```

代码 4-14 | jnifunc.cpp 中通过 JNI 函数调用 Java 方法并保存返回值的代码段

通过 JNI 调用 Java 方法的顺序如下。

1. 获取含待调方法的 Java 类的 jclass。
   （若待调方法属于某个 Java 类对象，则该方法用来获取 Java 类对象的 jobject）
2. 调用 GetMethodID()函数，获取待调方法的 ID（jMethodID）。
   （使用 jclass 与 GetMethodID()函数）
3. 根据返回值类型，调用相应的 JNI 函数，实现对 Java 方法的调用。
   （若待调的 Java 方法是静态方法，则调用函数的形式应为 CallStatic<type>Method()；若待调的方法属于某个类对象，则调用函数的形式应为 Call<type>Method()）

❶ 如代码 4-14 所示，向各位描述了在 JNI 本地函数中通过 JNI 调用 JniTest 对象的 callByNative()方法的过程。如上所述，程序首先获取含 callByNative()方法的 JniTest 类的 jclass。在获取 JniTest 类的 jclass 时，可以直接调用 FindClass()函数，将类引用保存在 targetClass 中。但是为了向各位介绍 GetObjectClass()这个 JNI 函数，而在此行中调用了 GetObjectClass()函数。

GetObjectClass()函数接收一个 jobject 类型的类对象引用，而后返回生成该对象的类的类引用（jclass 类型）。由于 JniTest 类对象的 jobject 值保存在 newobject 变量中（如代码 4-10❸所示），因此只需调用 GetObjectClass()函数，即可获取 JniTest 类引用。

❷ 调用名称为 GetMethodID()的 JNI 函数，获取 callByNative()方法的 ID。callByNative()方法的签名为“(I)I”，如图 4-18 所示。

❸ 由于 JniTest 对象的 callByNative()方法的返回值类型为 int，所以在 JNI 函数中需要调用 CallIntMehthod()函数。此外，callByNative()方法带有一个 int 类型的参数，在调用 CallIntMehthod()函数时，需要传入一个 int 型参数，示例代码中传入的是 200，如代码所示。callByNative()方法的返回值类型为 int，在 JNI 本地函数中，需要使用 jint 这种 Java 本地类型的变量 result 来保存返回值。

更多更新编程资源尽在 http://www.pin5i.com/

JNI 函数- CallStatic<type>Method()	
形式	<jnitype> CallStatic<type>Method(JNIEnv *env, jclass clazz, jmethodID methodID,...)
说明	调用 methodID 指定的类的静态方法
参数	env：JNI 接口指针 clazz：含待调方法的类 methodID：待调方法的 ID（由 GetStaticMethodID()函数获取） ...：传递给待调方法的参数
返回值	被调方法的返回值
参考	<type>除了前面讲解 Get<type>FieldID()时列出的 9 种类型外，又添加了 void 类型。返回值类型<jnitype>也增加了 void 类型。 待调方法的返回值不同，<type>也不同。若待调方法的返回值类型为 int，则调用函数为 CallStaticIntMethod()

JNI 函数- Call<type>Method()	
形式	<jnitype> Call<type>Method(JNIEnv *env, jobject obj, jmethodID methodID,...)
说明	调用 methodID 指定的 Java 对象的方法
参数	env：JNI 接口指针 obj：含待调方法的 Java 对象的引用 methodID：待调方法的 ID（由 GetMethodID()函数获取） ...：传递给待调方法的参数
返回值	被调方法的返回值

4．通过 JNI 设置成员变量的值

前面，已经讲解过在 JNI 本地函数中如何读取某成员变量的值。下面这部分将讲解在 JNI 中设置成员变量值的方法，所使用的方法与读取成员变量值的方法类似。但在设置成员变量值时，要调用 Set<type>Field()函数，而不是 Get<type>Field()函数。当然，若待设置的变量是静态变量，则应调用 SetStatic<type>Field()函数。

请看代码 4-15，代码描述了使用 JNI 函数如何设置 JniTest 对象的成员变量的值，对应于图 4-16（6）。代码 4-15 与读取成员变量值的代码类似，故在此省略说明。

```
// 1. 获取含 IntField 成员变量的 JniTest 类的 jclass 值
//     类引用已经被保存到 targetClass 中

// 2. 获取 JniTest 对象的 IntField 变量值
fid = env->GetFieldID(targetClass, "intField", "I");

// 3. 将 result 值设置为 IntField 值
env->SetIntField(newObject, fid, result);
```

代码 4-15 | jnifunc.cpp-调用 JNI 函数设置 Java 成员变量值

更多更新编程资源尽在 http://www.pin5i.com/

JNI 函数- SetStatic<type>Field	
形式	void SetStatic<type>Field（JNIEnv *env, jclass clazz, jfieldID fieldID, <type> value）
说明	设置 fieldID 指定的 Java 类静态成员变量的值
参数	env：JNI 接口指针 clazz：含待设置成员变量的类的引用 fieldID：待设成员变量的 ID（由 GetStaticFieldID()函数获取） value：指定设置值

JNI 函数- Set<type>Field	
形式	void Set<type>Field（JNIEnv *env, jobject obj, jfieldID fieldID, <type> value）
说明	设置 fieldID 指定的 Java 对象的成员变量的值
参数	env：JNI 接口指针 obj：包含待设成员变量的 Java 对象的引用 fieldID：待设成员变量的 ID（由 GetFieldID()函数获取） value：指定设置值

4.3.4　编译及运行结果

1．编译 JniFuncMain.java

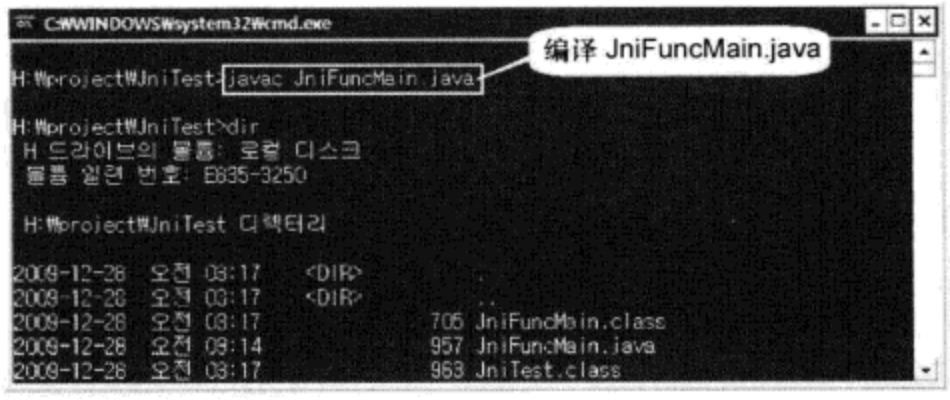


图 4-21 | 编译 JniFuncMain.java 源码

2．编译 jnifunc.cpp

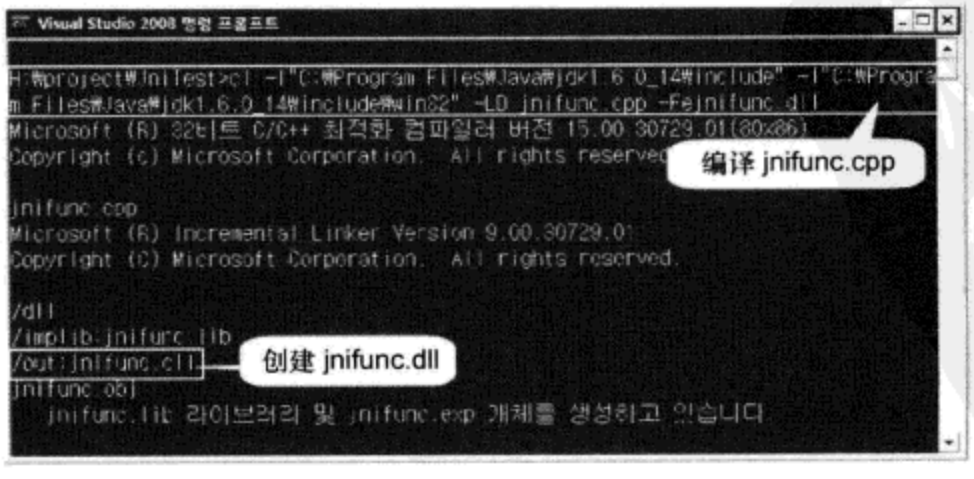


图 4-22 | 编译 jnifunc.cpp 源码

更多更新编程资源尽在 http://www.pin5i.com/

编译完源代码，运行代码，运行结果如图 4-23 所示。观察运行结果，可以看到在 JNI 本地函数中生成 Java 对象、调用 Java 方法等操作都得到正常实现。

图 4-23 | 运行 JniFuncMain 类

4.3.5 在 Android 中的应用举例

在 Android 系统中，也有大量代码使用了 JNI 函数，这些代码存在于 Android 源码的相关目录中。读者在如下目录下，可以查看到这些代码。

```
frameworks\base\core\jni
frameworks\base\services\jni
frameworks\base\media\jni
```

这些源代码已经超出本书的范围，在此就不再讲解了。但希望读者一定要认真阅读这些代码，理解它们，有助于我们了解在 Android 中如何使用 JNI。

4.4 在 C 程序中运行 Java 类

前面示例代码的主程序都是使用 Java 语言编写的，通过这些示例，我们学习了在 Java 代码中如何通过本地方法调用 C 函数，即使用 JNI 的方式。在本节中，我们将一起学习在由 C/C++编写的主程序中如何运行 Java 类，这也是使用 JNI 的重要方式。此时，一些读者可能会产生如下疑问。

由 Java 类编译生成的字节码必须运行在 Java 虚拟机上，那么在 C/C++编写的程序中运行 Java 类或对象还需要 Java 虚拟机吗？

当然，在 C/C++程序中运行 Java 类也必须使用 Java 虚拟机。为此，JNI 提供了一套 Invocation API，它允许本地代码在自身内存区域内加载 Java 虚拟机。在 C 代码中，如何使用 Invocation API，装载 Java 类，并运行指定的方法呢？这就是本节要讲解的主要内容。

下面列出的可能是你决定使用 Invocation API 在 C/C++代码中调用 Java 代码的几种典型情况。

更多更新编程资源尽在 http://www.pin5i.com/

- 需要在 C/C++编写的本地应用程序中访问用 Java 语言编写的代码或代码库。
- 希望在 C/C++编写的本地应用程序中使用标准 Java 类库。
- 当需要把已有的 C/C++程序与 Java 程序组织链接在一起时，使用 Invocation API，可以将它们组织成一个完整的程序。

除此之外，在编写并运行 Java 程序时使用的 java.exe 命令是由 C 语言编写的，该命令通过 Invocation API 接收命令参数，并执行命令参数指定的 Java 类。实际上，Android 系统的 dalvikvm 虚拟机的 Launcher 程序[1]也是通过 Invocation API 来工作的。

4.4.1 Invocation API 应用示例

下面通过一个示例来帮助大家理解 Invocation API，示例程序由 InvokeJava.cpp（代码 4-16）与 InvocationTest.java（代码 4-17）两个文件构成，如图 4-24 所示。

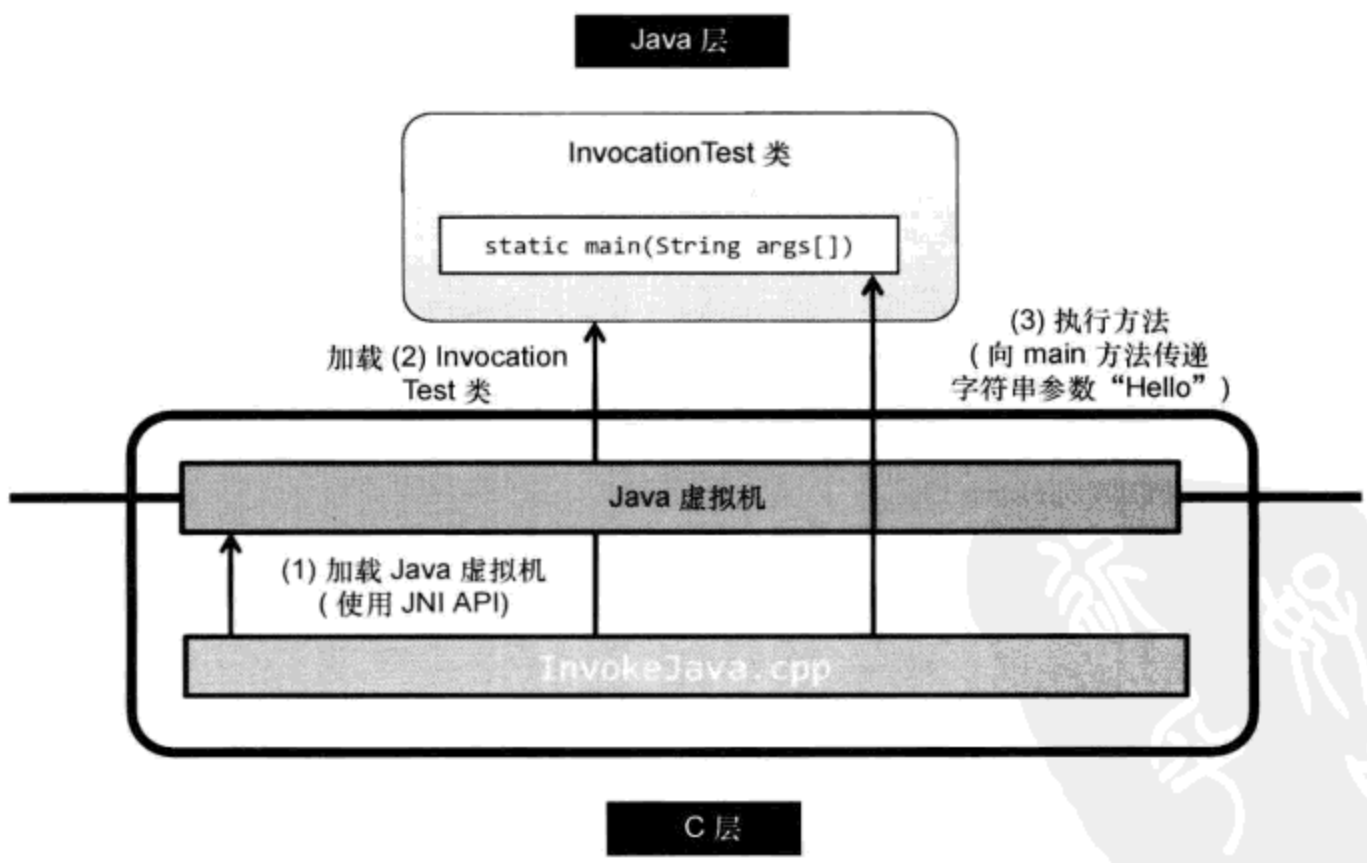


图 4-24 | Invocation API 应用示例

如图所示，示例程序将按如下顺序执行：（1）主程序 InvokeJava.cpp 使用 Invocation API 加载 Java 虚拟机。（2）通过 4.3 节中学习的 JNI 函数，加载 InvocationTest 类至内

1　dalvikvm-dalvik/dalvikvm/main.c

更多更新编程资源尽在 http://www.pin5i.com/

存中。（3）执行被加载的 InvocationTest 类的 main()方法。

C 程序如何加载 Java 虚拟机，如何通过 JNI 调用 Java 方法，下面我们将随着示例一起学习。

分析 Java 代码（InvocationApiTest.java）

如代码 4-16 所示，InvocationApiTest 类非常简单，仅含有一个 main()方法，该 main()方法是一个静态方法，带有一个字符串对象数组，在方法体中仅有一条输出语句，用来将第一个数组元素 args[0]中的字符串输出到控制台上。

```
public class InvocationApiTest
{
   public static void main(String[] args)
   {
      System.out.println(args[0]);
   }
}
```

代码 4-16 | InvocationApiTest.java 源码

分析 C 代码（invocationApi.c）

```
#include <jni.h>                                          ←❶

int main()
{
   JNIEnv *env;
   JavaVM *vm;
   JavaVMInitArgs vm_args;
   JavaVMOption options[1];
   jint res;
   jclass cls;
   jmethodID mid;
   jstring jstr;
   jclass stringClass;
   jobjectArray args;

   // 1. 生成 Java 虚拟机选项
   options[0].optionString = "-Djava.class.path="                 ❷
   vm_args.version = 0x00010002;
   vm_args.options = options;
   vm_args.nOptions = 1;
   vm_args.ignoreUnrecognized = JNI_TRUE;

   // 2. 生成 Java 虚拟机
   res = JNI_CreateJavaVM(&vm, (void**)&env, &vm_args);        ←❸
```

更多更新编程资源尽在 http://www.pin5i.com/


```
    // 3. 查找并加载类
    cls = (*env)->FindClass(env, "InvocationApiTest");

    // 4. 获取 main() 方法的 ID
    mid = (*env)->GetStaticMethodID(env, cls, "main", "([Ljava/lang/String;)V");

    // 5. 生成字符串对象，用作 main ( ) 方法的参数
    jstr = (*env)->NewStringUTF(env, "Hello Invocation API!!");
    stringClass = (*env)->FindClass(env, "java/lang/String");
    args = (*env)->NewObjectArray(env, 1, stringClass, jstr);

    // 6. 调用 main ( ) 方法
    (*env)->CallStaticVoidMethod(env, cls, mid, args);

    // 7. 销毁 Java 虚拟机
    (*vm)->DestroyJavaVM(vm);
}
```

❹ ❺ ❻ ❼

代码 4-17 | invocationApi.c 的源代码

下面开始分析代码的主要部分。

❶ #include 命令用来将 jni.h 头文件包含到本文件中。jni.h 头文件包含 C 代码使用 JNI 必需的各种变量类型或 JNI 函数的定义，在本地代码中使用 JNI 时，必须将此头文件包含到本地代码中。

❷ 生成一些参数或选项值，这些值在加载 Java 虚拟机时被引用，用来设置 Java 虚拟机的运行环境或控制 Java 虚拟机的运行，如设置 CLASSPATH 或输出调试信息等。与 Java 虚拟机参数选项相关的内容，请参考如下页面。

http://java.sun.com/javase/6/docs/technotes/tools/windows/java.html

在运行 Java 类时使用的 java.exe（Java Launcher）命令也支持这类 VM 选项。

在生成 Java 虚拟机选项时，使用代码 4-17 所示的 JavaVMInitArgs 与 JavaVMOption 结构体，它们定义在 jni.h 头文件中，定义如下：

```
typedef struct JavaVMInitArgs {
    jint version;
    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
} JavaVMInitArgs;

typedef struct JavaVMOption {
    char *optionString;
    void *extraInfo;
} JavaVMOption;
```

观察 JavaVMInitArgs 结构体定义代码，可以发现 JavaVMInitArgs 结构体内包含

更多更新编程资源尽在 http://www.pin5i.com/

JavaVMOption 结构体的指针。JavaVMOption 结构体包含 Java 虚拟机的各个参数，JavaVMInitArgs 结构体用来将这些参数选项传递给 Java 虚拟机。

接下来，看一下结构体中各个成员的含义。

JavaVMInitArgs 结构体的 version 成员用来指定传递给虚拟机的选项的变量的形式，设定在 jni.h 头文件中定义的 JNI_VERSION_1_2 的值。nOptions 与 options 用来指定 JavaVMInitArgs 所指的 JavaVMOption 结构体数组值。nOptions 指定 JavaVMOption 结构体数组元素的个数，options 用来指向 JavaVMOption 结构体的地址。在代码 4-17 中，只设置了一个 Java 虚拟机选项，即 JavaVMOption 结构体数组仅有一个元素，声明如下：

```
JavaVMOption options[1];
```

为了指定以上 JavaVMOption 结构体数组，需要指定 JavaVMInitArgs 的 options 与 nOptions，如下所示。

```
vm_args.options = options;  // JavaVMOption   结构体的地址
vm_args.nOptions = 1;       // JavaVMOption   结构体数组元素个数
```

ignoreUnrecognized 是 JavaVMInitArgs 结构体 jboolean 类型的成员，当 Java 虚拟机读到设置错误的选项值时，该成员用来决定 Java 虚拟机是忽略错误后继续执行，还是返回错误后终止执行。若 ignoreUnrecognized 被设置为 JNI_TRUE，Java 虚拟机遇到错误选项时，忽略错误后继续执行；若被设置为 JNI_FALSE，当遇到错误选项，Java 虚拟机将错误返回后终止执行。

接下来，分析一下 JavaVMOption 结构体，它用来指定 Java 虚拟机的选项值。若想创建选项值，只要向结构体的 optionString 成员指定一个字符串，用作 Java 虚拟机选项的形式。比如示例中，选项字符串为“-Djava.class.path=.”，用来设置标准选项，即将 Java 虚拟机要加载的类的默认目录设置为当前目录（.），其形式为-Dproperty=value，如下所示。

```
-Dproperty=value
设置系统属性(property)为指定的值(value)。
```

❸ 本行代码是整个程序的核心部分，即 C 应用程序调用 JNI_CreateJavaVM()函数，生成并装载 Java 虚拟机。JNI_CreateJavaVM()函数的第一个参数类型为 JavaVM，它表示 Java 虚拟机接口，用来生成或销毁 Java 虚拟机。DestroyJavaVM()是接口函数之一，该函数用来销毁 Java 虚拟机，如代码行❼所示，更具体的内容，请参看 JNI 相关文档。

在 JNI_CreateJavaVM()函数的第二个参数 env 中，保存着 JNI 接口指针的地址。通过 env 所指的 JNI 接口指针，可以使用各种 JNI 函数，即在 C/C++代码中，通过 env，可以生成 Java 对象，调用相应方法等。

更多更新编程资源尽在 http://www.pin5i.com/

JNI Invocation API-JNI_CreateJavaVM	
形式	jint JNI_CreateJavaVM(JavaVM **vm, JNIEnv **env, void *vm_args)
说明	装载并初始化 Java 虚拟机
参数	vm：JavaVM 指针的地址 env：JNI 接口指针的地址 vm_args：传递给 Java 虚拟机的参数
返回值	成功，返回 0；失败，返回负值

❹ 为了实现图 4-24 中的（2）（3）功能，首先调用 4.3 节中介绍过的 FindClass()函数，装载 InvocationApiTest 类。而后调用 GetStaticMethodID()函数，获取 main()方法的 ID，准备调用 main()方法。

❺ 在使用 CallStaticVoidMethod()函数调用 main()方法之前，首先构造出传递给 main()方法的参数。Java 的 main()方法的参数是 String[]数组，如下所示。

```
public static void main(String[] args)
```

示例中将“Hello Invocation API!!”字符串传递给 main()方法。首先调用 NewStringUTF()函数，将 UTF-8 形式的字符串，转换成 Java 字符串对象 String。然后调用 NewObjectArray()函数，创建 String 对象数组，使用创建的 String 对象将其初始化。如代码 4-17 所示，先创建含有一个元素的 String[]数组，而后将“Hello Invocation API!!”字符串赋值给数组的第一个元素。

在❺部分中，调用 JNI 本地函数处理 String 对象的方法有些复杂。如果你对此仍迷惑不解，我们不妨将这部分代码转换成与其等价的 Java 代码，帮助大家理解一下，如下所示，首先创建包含一个元素的字符串数组，而后将“Hello Invocation API!!”字符串赋值给数组的首个元素。

```
String[] args = new String[1];
args[0] = "Hello Invocation API!"
```

JNI 函数-NewStringUTF	
形式	jstring NewStringUTF(JNIEnv *env,const char *bytes)
说明	将 UTF-8 形式的 C 字符串转换成 java.lang.String 对象
参数	env：JNI 接口指针 bytes：待生成 String 对象的 C 字符串的地址
返回值	成功，返回 String 对象的 jstring 类型的引用；失败，返回 NULL

更多更新编程资源尽在 http://www.pin5i.com/

JNI 函数-NewObjectArray	
形式	jarray NewObjectArray(JNIEnv *env, jsize length, jclass elementClass, jobject initalElement)
说明	生成由 elementClass 对象组成的数组。数组元素个数由 length 指定，initalElement 参数用来初始化对象数组
参数	env：JNI 接口指针 length：数组元素个数 elementClass：数组元素对象的类型 initialElement：数组初始化值
返回值	若成功，则返回数组引用；失败，则返回 NULL

❻ 本行代码通过 CallStaticVoidMethod()函数调用 InvocationApiTest 类的 main()方法。在❺中创建的 String[]数组是 CallStaticVoidMethod()函数的第四个参数，该参数会被传递给 InvocationApiTest 类的 main()方法。当 InvocationApiTest 类的 main()方法被调用执行时，它会向控制台输出 args 字符串数组的 args[0]元素中的字符串。

4.4.2 编译及运行

1．编译 Java 代码

执行 Java 编译命令，编译 InvocationApiTest.java。

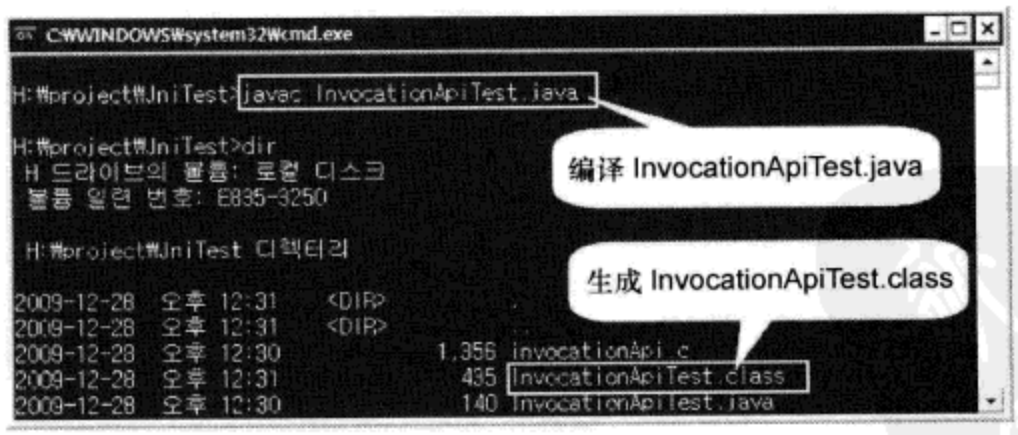


图 4-25 | 编译 InvocationApiTest.java

2．编译 C 代码

使用 Visual Studio 2008 的 Command Prompt 工具，编译 invocationApi.c 文件，所用的编译命令如下。

更多更新编程资源尽在 http://www.pin5i.com/


```
cl -I"<JDK_HOME>\include" -I"<JDK_HOME>\include\win32" invocationApi.c
        -link "<JDK_HOME>\lib\jvm.lib"
```

cl : Visual C++ 编译器

-I : 添加头文件检索目录

需要包含的头文件如下

jni.h (<JDK_HOME>\include)

jni_md.h (<JDK_HOME>\include\win32)

-link : 链接库

用于链接 Java 虚拟机的库文件 (jvm.lib 文件)

如前所述，笔者的<JDK_HOME>目录设置为 C:\Program Files\Java\jdk1.6.0_14，编译命令如图 4-26 所示。编译完成后，运行 invocationApi.exe 可执行文件。

图 4-26 | 编译 invocationApi.c

3．运行示例程序

编译完成后，运行 invocationApi.exe，在控制台上输出“Hello World Invocation API!!”字符串，执行结果如图 4-27 所示。

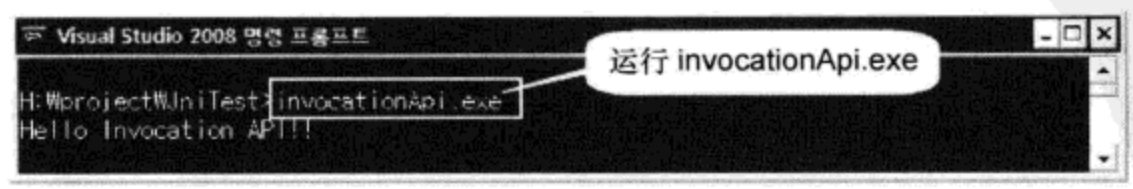


图 4-27 | 执行 invocationApi.exe

更多更新编程资源尽在 http://www.pin5i.com/

4.4.3 Invocation API 在 Android 中的应用举例：Zygote 进程

Zygote 进程是 Android 系统的核心，本身是一个应用层程序，由名称为 app_process[1] 的 C++本地应用程序调用 JNI Invocation API 启动运行。

当 Android Framework 启动时，app_process 会先初始化 Android runtime，而后再运行 Zygote 进程。Zygote 是个特殊的进程，它能提升 Android Framework 的性能，所有 Android 应用进程由 Zygote 进程 fork 出来。Zygote 进程由称为 Android zygoteInit 类[2]的 Java 程序组成，在 Android 启动时，app_process 调用 JNI invocation API 在自身程序域内加载 dalvikvm 虚拟机，而后调用 ZygoteInit 类的 main()方法，从而运行 Zygote 进程。

以上相关源码、Android runtime，以及 Zygote 进程的详细讲解将在下一章中进行，这里只是简单地提一下。

4.5 直接注册 JNI 本地函数

在前面“JNI 的基本原理”一节的学习中，我们已经知道 Java 虚拟机在运行包含本地方法的 Java 应用程序时，要经过以下两个步骤。

1. 调用 System.loadLibrary()方法，将包含本地方法具体实现的 C/C++运行库加载到内存中。
2. Java 虚拟机检索加载进来的库函数符号，在其中查找与 Java 本地方法拥有相同签名的 JNI 本地函数符号。若找到一致的，则将本地方法映射到具体的 JNI 本地函数。

像 4.2 节中生成的 hellojni.dll 一样，若 JNI 支持的功能函数仅有一个，Java 虚拟机在将本地方法与 C 运行库中的 JNI 本地函数映射在一起时，不会耗费很长时间。但在 Android Framework 这类复杂的系统下，拥有大量的包含本地方法的 Java 类，Java 虚拟机加载相应运行库，再逐一检索，将各个本地方法与相应的函数映射起来，这显然会增加运行时间，降低运行的效率。

为了解决这一问题，JNI 机制提供了名称为 RegisterNatives()的 JNI 函数，该函数允许 C/C++开发者将 JNI 本地函数与 Java 类的本地方法直接映射在一起。当不调用 RegisterNatives()函数时，Java 虚拟机会自动检索并将 JNI 本地函数与相应的 Java 本地方法链接在一起。但当开发者直接调用 RegisterNatives()函数进行映射时，Java 虚拟机

---

1 frameworks/base/cmds/app_process/app_main.cpp

2 frameworks/base/core/java/com/android/internal/os/ZygoteInit.java

更多更新编程资源尽在 http://www.pin5i.com/

就不必进行映射处理，这会极大提高运行速度，提升运行效率。

由于程序员直接将 JNI 本地函数与 Java 本地方法链接在一起，在加载运行库时，Java 虚拟机不必为了识别 JNI 本地函数而将 JNI 本地函数的名称与 JNI 支持的命名规则进行比对，即任何名称的函数都能直接链接到 Java 本地方法上。

下面通过示例，来具体了解一下详细内容。

4.5.1　加载本地库时，注册 JNI 本地函数

在代码 4-4 hellojni.c 的基础上，经过添加、修改某些代码，并调用 RegisterNatives()函数，将本地方法与 C 函数映射在一起，代替 Java 虚拟机进行映射，最终代码，请阅读代码 4-18 hellojnimap.cpp。

首先分析 System.loadLibrary()方法的执行过程。在 Java 代码中，调用 System.loadLibrary()方法时，Java 虚拟机会加载其参数指定的共享库。然后，Java 虚拟机检索共享库内的函数符号，检查 JNI_OnLoad()函数是否被实现，若共享库中含有相关函数，则 JNI_OnLoad()函数就会被自动调用。假若像前面生成的 hellojni.dll 一样，库中的 JNI_OnLoad()函数未被实现，则 Java 虚拟机会自动将本地方法与库内的 JNI 本地函数符号进行比较匹配。

在加载指定的库文件时，JNI_OnLoad()函数会被自动调用执行，程序开发者若想手工映射本地方法与 JNI 本地函数，需要在 JNI_OnLoad()函数内调用 RegisterNatives()函数进行映射匹配，如代码 4-18 的 JNI_OnLoad()函数体内的代码。JNI_OnLoad()函数形式、参数、返回值如下表所示。

JNI Invocation API-JNI_OnLoad	
形式	jint JNI_OnLoad(JavaVM *vm, void *reserved)
说明	Java 虚拟机加载本地库时（System.loadLibrary()等方法被调用时）会调用 JNI_OnLoad()函数。在使用加载库的过程中，JNI_OnLoad()函数会向 Java 虚拟机确认 JNI 的版本。若库中不包含 JNI_OnLoad()函数，Java 虚拟机会认为相关库要求 JNI 1.1 版本支持
参数	vm：JavaVM 接口指针 reserved：预定参数
返回值	若执行成功，则返回所生成的数组引用；若失败，则返回 NULL

如上所示，JNI_OnLoad()函数最基本的功能是确定 Java 虚拟机支持的 JNI 的版本。因此，JNI_OnLoad()函数必须返回有关 JNI 版本的信息。此外，通过 JNI_OnLoad()函数，Java 虚拟机可以在加载本地库时对 JNI 进行初始化。

请看代码 4-18 hellojnimap.cpp 代码，首先向包含 JNI 本地函数的库中添加 JNI_OnLoad()函数，再在该函数中调用 RegisterNatives()函数，将 Java 类的本地方法与 JNI

更多更新编程资源尽在 http://www.pin5i.com/

本地函数映射在一起。

```
#include "jni.h"                                                    ❶
#include <stdio.h>

// JNI 本地函数原型
void printHelloNative (JNIEnv *env, jobject obj);                   ❷
void printStringNative(JNIEnv *env, jobject obj, jstring string);

JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM *vm, void *reserved)
{
    JNIEnv* env = NULL;
    JNINativeMethod nm[2];
    jclass cls;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {     ❸
        printf("Error");
        return JNI_ERR;
    }

    cls = env->FindClass("HelloJNI");                               ❹

    nm[0].name = "printHello";
    nm[0].signature = "()V";
    nm[0].fnPtr = (void *)printHelloNative;                         ❺

    nm[1].name = "printString";
    nm[1].signature = "(Ljava/lang/String;)V";
    nm[1].fnPtr = (void *)printStringNative;

    env->RegisterNatives(cls, nm, 2);

    return JNI_VERSION_1_4;
}

// 实现 JNI 本地函数
void printHelloNative (JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");

    return;
}

void printStringNative(JNIEnv *env, jobject obj, jstring string)
{
    const char *str = env->GetStringUTFChars(string, 0);
    printf("%s!\n", str);

    return;
}
```

代码 4-18 | hellojnimap.cpp 源码

更多更新编程资源尽在 http://www.pin5i.com/

❶ 在实现 JNI 本地函数之前，必须先在文件开头将 jni.h 头文件包含进来。jni.h 头文件中包含各种 JNI 数据结构、JNI 函数、宏定义等。

细心的读者可能会发现在代码 4-4 hellojni.c 代码中，并没有包含 jni.h 头文件，这是为什么呢？事实上，jni.h 头文件已经被包含进去，因为 jni.h 头文件被包含在 HelloJni.h 文件中，而 HelloJni.h 被包含在 hellojni.c 之中。

❷ 此两行代码用来声明 JNI 本地函数原型。如前所述，在使用 RegisterNatives()函数进行映射时，不需要将 JNI 本地函数原型与 JNI 命名规则进行比对，所以使用的函数名比较简单。但函数中的两个公共参数，必须指定为“JNIEnv *env, jobject obj”。

❸ 在 JNI_OnLoad()函数中首先判断 JNI 的版本，即调用 GetEnv()函数，判断 Java 虚拟机是否支持 JNI 1.4。若 Java 虚拟机支持 JNI 1.4，JNI_OnLoad()函数就会返回 JNI_VERSION_1_4；若不支持，JNI_OnLoad()函数就会返回 JNI_ERR，并终止装载库的行为。

当 GetEnv()函数调用完毕后，JNI 接口指针被保存到 env 变量中，在调用 FindClass()、RegisterNatives()等 JNI 函数时，可以使用该变量。

JNI Invocation API-GetEnv

形式	jint GetEnv(JavaVM *vm,void **env, jint version)
说明	判断 Java 虚拟机是否支持 version 指定的 JNI 版本，而后将 JNI 接口指针设置到*env 中
参数	vm：JavaVM 接口指针的地址 env：JNI 接口指针地址 version：JNI 版本
返回值	若执行成功，返回 0；失败，返回负数

❹ 为了把❷行中声明的 JNI 本地函数与 HelloJNI 类的本地方法链接到一起，本行先调用 FindClass()函数加载 HelloJNI 类，并将类引用保存到 jclass 变量 cls 中。

❺ 该部分代码用来将 Java 类的本地方法与 JNI 本地函数映射在一起。首先使用 JNINativeMethod 结构体数组，将待映射的本地方法与 JNI 本地函数的相关信息保存在数组中，而后调用 RegisterNatives()函数进行映射。JNINativeMethod 结构体定义如下。

```
typedef struct {
    char *name;          // 本地方法名称
    char *signature;     // 本地方法签名
    void *fnPtr;         // 与本地方法相对应的 JNI 本地函数指针
} JNINativeMethod
```

如代码 4-18 所示，nm 是 JNINativeMethod 结构体数组，它保存着 printHello()、printString()（此两个本地方法在 HelloJNI 类中被声明，参见代码 4-1）与 printHelloNative()、printStringNative()函数（此两个本地函数在代码 4-18 中被声明）的链接信息。

更多更新编程资源尽在 http://www.pin5i.com/

```
nm[0].name = "printHello";
nm[0].signature = "()V";
nm[0].fnPtr = (void *)printHelloNative;

nm[1].name = "printString";
nm[1].signature = "(Ljava/lang/String;)V";
nm[1].fnPtr = (void *)printStringNative;
```

保存好映射信息后，将它们传递给 RegisterNatives()函数，最后由 RegisterNatives()函数完成映射。

```
env->RegisterNatives(cls, nm, 2);
```

JNI 函数- RegisterNatives	
形式	jarray RegisterNatives(JNIEnv *env, jclass clazz, const JNINativeMethod *methods, jint nMethods)
说明	将 clazz 指定类中的本地方法与 JNI 本地函数链接在一起，链接信息保存在 JNINativeMethod 结构体数组中
参数	env：JNI 接口指针 clazz：Java 类 methods：包含本地方法与 JNI 本地函数的链接信息 nMethods：methods 数组元素的个数
返回值	若执行成功，返回数组引用；否则，返回 NULL

总结一下以上内容，如图 4-28 所示。在 4.2 节的示例中，Java 本地方法与 JNI 本地函数是由 Java 虚拟机自动完成映射的，而本节中则通过 JNI_Onload()函数，由开发者直接编写代码调用相关函数来实现映射。

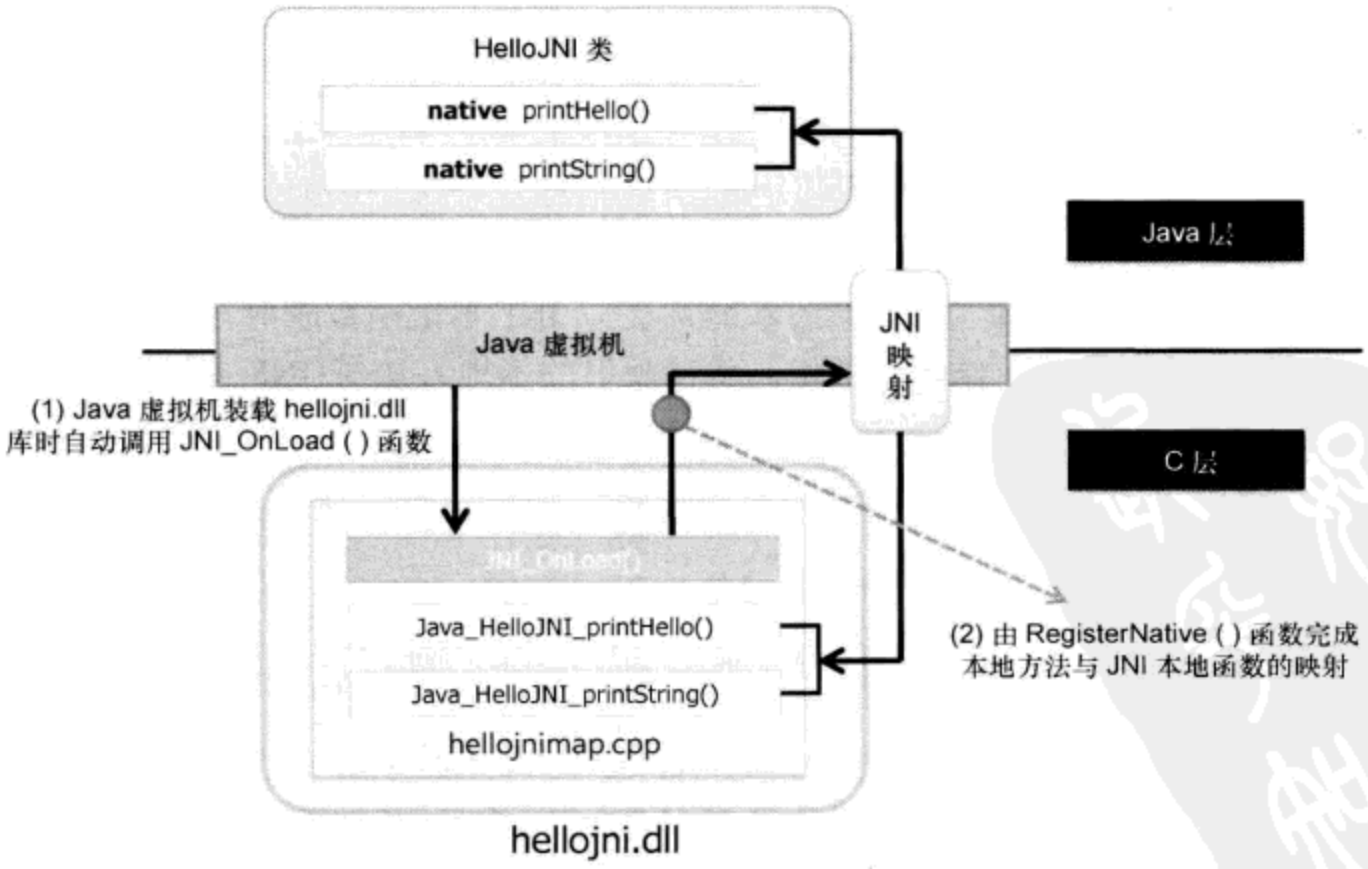


图 4-28 | 调用 RegisterNatives()函数完成本地方法与 JNI 本地函数的映射

更多更新编程资源尽在 http://www.pin5i.com/

编译 C 源码（hellojnimap.cpp）

采用 4.2 节中介绍的编译方法，编译 hellojnimap.cpp 源码，编译过程如下图所示。

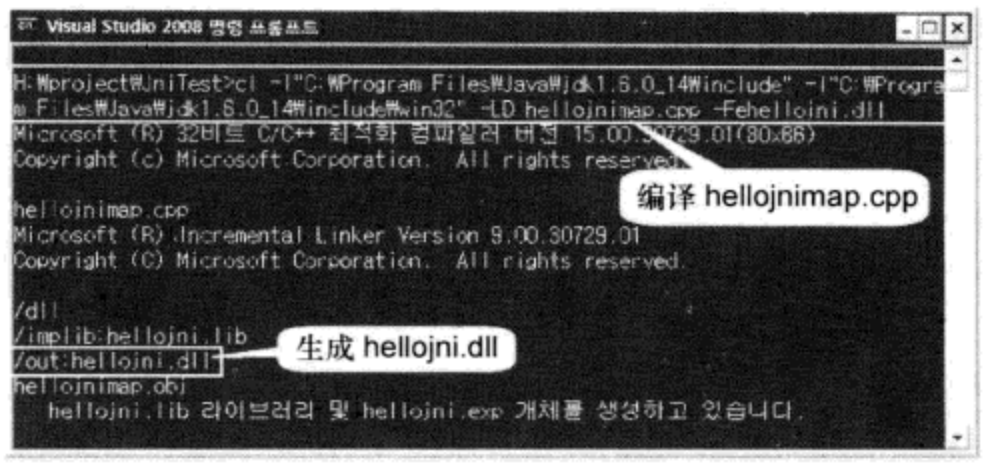


图 4-29 | 编译 hellojnimap.cpp 源码

运行示例程序

执行 4.2 节图 4-3 中编译的 HelloJNI.class 字节码文件，运行结果如下图，可以发现运行结果与图 4-14 中 HelloJNI.class 的运行结果相同。

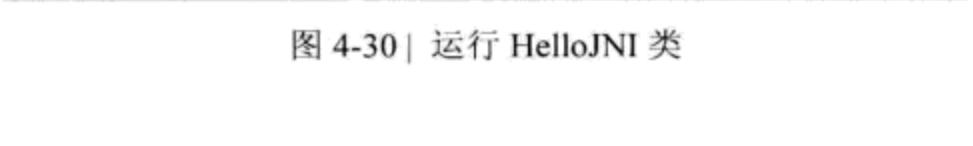


图 4-30 | 运行 HelloJNI 类

4.5.2　Android 中的应用举例

System Server：在 JNI_OnLoad 中注册 JNI 本地函数

在 Android 系统中也使用 JNI_OnLoad()函数直接映射本地方法。下面这段代码是系统服务器(System Server)代码的一部分，在启动 Android 时，系统服务器(System Server）用来运行各种服务，并加载多种本地库。在示例代码中将加载一个非常简单的 android_servers 库，并通过这一过程来分析 Android 是如何直接进行 JNI 映射的。

更多更新编程资源尽在 http://www.pin5i.com/

```
native public static void init1(String[] args); // 本地方法

public static void main(String[] args) {
    ...
    System.loadLibrary("android_servers");    ←❶
    init1(args);
}
```

代码 4-19 | Android Framework 的 System Server 程序中加载 android_servers 库的代码段[1]

❶ 加载名称为 android_servers 的库（在 Linux 系统下，加载库的名称为 libandroid_servers.so）。

略微分析一下 libandroid_servers.so 库。代码 4-20 是编译 libandroid_servers.so 库的 make 文件的一部分。在 LOCAL_SRC_FILES 变量中各个 cpp 源文件共同组成了 LOCAL_MODULE 变量中的 libandroid_servers.so 库。

```
LOCAL_SRC_FILES:= \
    com_android_server_AlarmManagerService.cpp \
    com_android_server_BatteryService.cpp \
    com_android_server_HardwareService.cpp \
    com_android_server_KeyInputQueue.cpp \
    com_android_server_SensorService.cpp \
    com_android_server_SystemServer.cpp \
    onload.cpp

    ...

LOCAL_MODULE:= libandroid_servers
```

代码 4-20 | 编译 android_servers 库的 make 文件[2]

请看代码 4-21 onload.cpp 源代码，它是组成 android_servers 库的源代码文件之一。在 onload.cpp 源代码中，可以看到 JNI_OnLoad()函数已经实现，即 JNI_OnLoad()函数被包含到 libandroid_servers.so 库之中。

因此，当执行代码 4-19 中的❶行代码 System.loadLibrary(“android_servers”)，加载 android_servers 库时，会执行代码 4-21 中的 JNI_OnLoad()函数。

1 framework/base/services/java/com/android/server/SystemServer.java

2 framework/base/services/jni/Android.mk

更多更新编程资源尽在 http://www.pin5i.com/


```
extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {   ❷
        LOGE("GetEnv failed!");
        return result;
    }

    register_android_server_KeyInputQueue(env);
    register_android_server_HardwareService(env);
    register_android_server_AlarmManagerService(env);   ❸
    register_android_server_BatteryService(env);
    register_android_server_SensorService(env);
    register_android_server_SystemServer(env);

    ...

    return JNI_VERSION_1_4;
}
```

代码 4-21 | libandroid_servers.so 库的 JNI_OnLoad()函数-onload.cpp[1]

❷ GetEnv()函数在前面已经提及过，它用来确定 JNI 的版本，并获取 JNI 接口函数的指针。

❸ 在这部分调用了一系列的注册函数，将组成 Android Framework 的 AlarmManagerService[2]、KeyInputQueue[3]等类的本地方法映射到 libandroid_servers.so 库相应的 JNI 本地函数上，注册函数的一般形式为 register_android_server_xxx()。其中，详细分析一下 register_android_ server_SystemServer()函数，如代码 4-22 所示，该函数用来映射 SystemServer 类中的本地方法。

```
static JNINativeMethod gMethods[] = {
    { "init1",                                                    ❹
    "([Ljava/lang/String;)V",
    (void*) android_server_SystemServer_init1 },
};

int register_android_server_SystemServer(JNIEnv* env)
{
    return jniRegisterNativeMethods(env, "com/android/server/SystemServer",   ❺
        ➥   gMethods, NELEM(gMethods));
```

1 framework/base/services/jni/onload.cpp

2 framework/base/services/java/com/android/server/AlarmManagerService.java

3 framework/base/services/java/com/android/server/KeyInputQueue.java

更多更新编程资源尽在 http://www.pin5i.com/

```
}

static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
    system_init();
}
```

代码 4-22 | com_android_server_SystemServer.cpp[1]-此段代码用于将 SystemServer 类中的本地方法与 libandroid_servers.so 库中的 JNI 本地函数映射在一起

❺ 在 register_android_server_SystemServer()函数中调用了 jniRegisterNativeMethods()函数，jniRegisterNativeMethods()函数体如代码 4-23 所示。JniRegisterNativeMethods()函数是 Android 系统提供的 JNI Helper 函数，它负责将指定类的本地方法映射到 JNI 本地函数上。

```
int jniRegisterNativeMethods(JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    jclass clazz;
    clazz = (*env)->FindClass(env, className);
    if (clazz == NULL) {
        LOGE("Native registration unable to find class '%s'\n", className);   ❻
        return -1;
    }
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        LOGE("RegisterNatives failed for '%s'\n", className);                  ❼
        return -1;
    }
    return 0;
}
```

代码 4-23 | JNIHelp.h[2]-jniRegisterNativeMethods()函数

分析一下 jniRegisterNativeMethods()函数代码。

❻ 调用 FindClass()函数，查找并加载 className 参数指定的类（此类含有待映射的本地方法），返回类引用，并将其保存到 jclass 变量 clazz 中。

❼ 本行调用 RegisterNatives()函数，以第❹部分中保存类引用的 jclass 变量 clazz，以及第❻部分中 JNINativeMethod 结构体类型的变量 gMethods 为参数，完成 Java 类本地方法到 JNI 本地函数的映射。

1 framework/base/services/jni/com_android_server_SystemServer.cpp

2 dalvik/libnativehelper/include/nativehelper/JNIHelp.h

更多更新编程资源尽在 http://www.pin5i.com/

由上可知，register_android_server_SystemServer()函数的功能是将 SystemServer 类（位于 com.android.server 包中）的本地方法 init1()映射到 android_server_SystemServer_init1()上。

app_process：在 C 程序中注册 JNI 本地函数

如 4.4 节所示，在 C 程序中通过 JIN Invocation API 调用 Java 代码的过程中，开发者在直接映射本地方法时，可以不使用 JNI_Onload()函数，而在 C 程序中直接调用 RegisterNatives()函数，完成 JNI 本地函数与 Java 类本地方法间的链接映射。

Android 的 app_process 系统进程即采用上述方式，将实现 Android Framework 的各种 Java 类的本地方法与 C 函数映射在一起。

```
static const RegJNIRec gRegJNI[] = {                    ←❶
    REG_JNI(register_android_debug_JNITest),
    REG_JNI(register_com_android_internal_os_RuntimeInit),    ←❷
    REG_JNI(register_android_os_SystemClock),
    REG_JNI(register_android_util_EventLog),
    REG_JNI(register_android_util_Log),
    REG_JNI(register_android_util_FloatMath),
    ...
    REG_JNI(register_android_util_Base64),
    REG_JNI(register_android_location_GpsLocationProvider),
    REG_JNI(register_android_backup_BackupDataInput),
    REG_JNI(register_android_backup_BackupDataOutput),
    REG_JNI(register_android_backup_FileBackupHelperBase),
    REG_JNI(register_android_backup_BackupHelperDispatcher),
};

int AndroidRuntime::startReg(JNIEnv* env)
{                                                       ←❸
    ...
    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    ...

    return 0;
}

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) {  // 调用函数
            return -1;
        }
    }
```

更多更新编程资源尽在 http://www.pin5i.com/

```
        return 0;
    }
```

代码 4-24 | app_process 部分源码[1]-映射 JNI 本地函数

在代码 4-24 app_process 的部分源码中，调用了 startReg()函数，该函数用来注册 Android Framework 中使用的各种 JNI 本地函数。在 startReg()函数中调用 register_jni_procs()函数，接收❶行中定义的 gRegJNI 数组，并依次执行其中用于注册 JNI 本地函数的函数❸。gRegJNI 数组是一个结构体数组，定义中使用了 REG_JNI 宏，该宏用于代码调试，阅读代码时可以将其忽略掉，gRegJNI 数组的成员是注册 JNI 本地函数的函数指针❶。

在代码 4-24 中有多个注册 JNI 本地函数的函数，其中有一个❷register_com_android_internal_os_RuntimeInit()函数，接下来一起分析一下该函数（在 gRegJNI 数组中的其他函数运行方式都与该函数类似）。register_com_android_internal_os_RuntimeInit()函数体如代码 4-25❺所示，该函数调用名称为 jniRegisterNativeMethods()的 JNI Helper 函数，将 Java 本地方法与 JNI 本地函数映射在一起。在调用 jniRegisterNativeMethods()函数时，需要以包含待映射的 Java 本地方法的 Java 类，以及包含映射信息的结构体作为参数。

如代码 4-25❺所示，调用 jniRegisterNativeMethods()函数，借助❹中的映射信息，将 RuntimeInit 类中的本地方法与❻中的 C 函数映射在一起。

```
// 本地方法与 JNI 本地函数映射信息
static JNINativeMethod gMethods[] = {
    { "finishInit", "()V",
        (void*) com_android_internal_os_RuntimeInit_finishInit },                    ❹
    { "zygoteInitNative", "()V",
        (void*) com_android_internal_os_RuntimeInit_zygoteInit },
    { "isComputerOn", "()I",
        (void*) com_android_internal_os_RuntimeInit_isComputerOn },
    { "turnComputerOn", "()V",
        (void*) com_android_internal_os_RuntimeInit_turnComputerOn },
    { "getQwertyKeyboard", "()I",
        (void*) com_android_internal_os_RuntimeInit_getQwertyKeyboard },
};

// 调用 RegisterNatives 函数，映射本地方法
int register_com_android_internal_os_RuntimeInit(JNIEnv* env)
{
    return jniRegisterNativeMethods(env,                                             ❺
                    ➥    "com/android/internal/os/RuntimeInit",
```

1 frameworks/base/core/jni/AndroidRuntime.cpp

更多更新编程资源尽在 http://www.pin5i.com/


```
                    ➥   gMethods, NELEM(gMethods));
}

// 被映射到本地方法上的 C 函数
static void com_android_internal_os_RuntimeInit_finishInit(JNIEnv* env,
     ➥  jobject clazz)
{
   gCurRuntime->onStarted();
}

static void com_android_internal_os_RuntimeInit_zygoteInit
                                ➥  (JNIEnv* env, jobject clazz)
{
   gCurRuntime->onZygoteInit();
}
static jint com_android_internal_os_RuntimeInit_is                    ❻
                                ➥  ComputerOn(JNIEnv* env, jobject clazz)
{
   return 1;
}
static void com_android_internal_os_RuntimeInit_turnComputerOn
                                ➥   (JNIEnv* env, jobject clazz)
{
}
static jint com_android_internal_os_RuntimeInit_getQwertyKeyboard
                                        ➥   (JNIEnv* env, jobject clazz)
{
   char* value = getenv("qwerty");
   if (value != NULL && strcmp(value, "true") == 0) {
      return 1;
   }

   return 0;
}
```

代码 4-25 | AndroidRuntime.cpp 部分源码[1]-Android RuntimeInit 类的本地方法与待映射的 JNI 本地函数

图 4-31 是 register_com_android_internal_os_RuntimeInit()函数的执行过程。

以上讲解了在 Android Framework 中如何把各个类的本地方法直接映射到 C 函数上，这些映射方法读者要了解学习。

1 frameworks/base/core/jni/AndroidRuntime.cpp

更多更新编程资源尽在 http://www.pin5i.com/

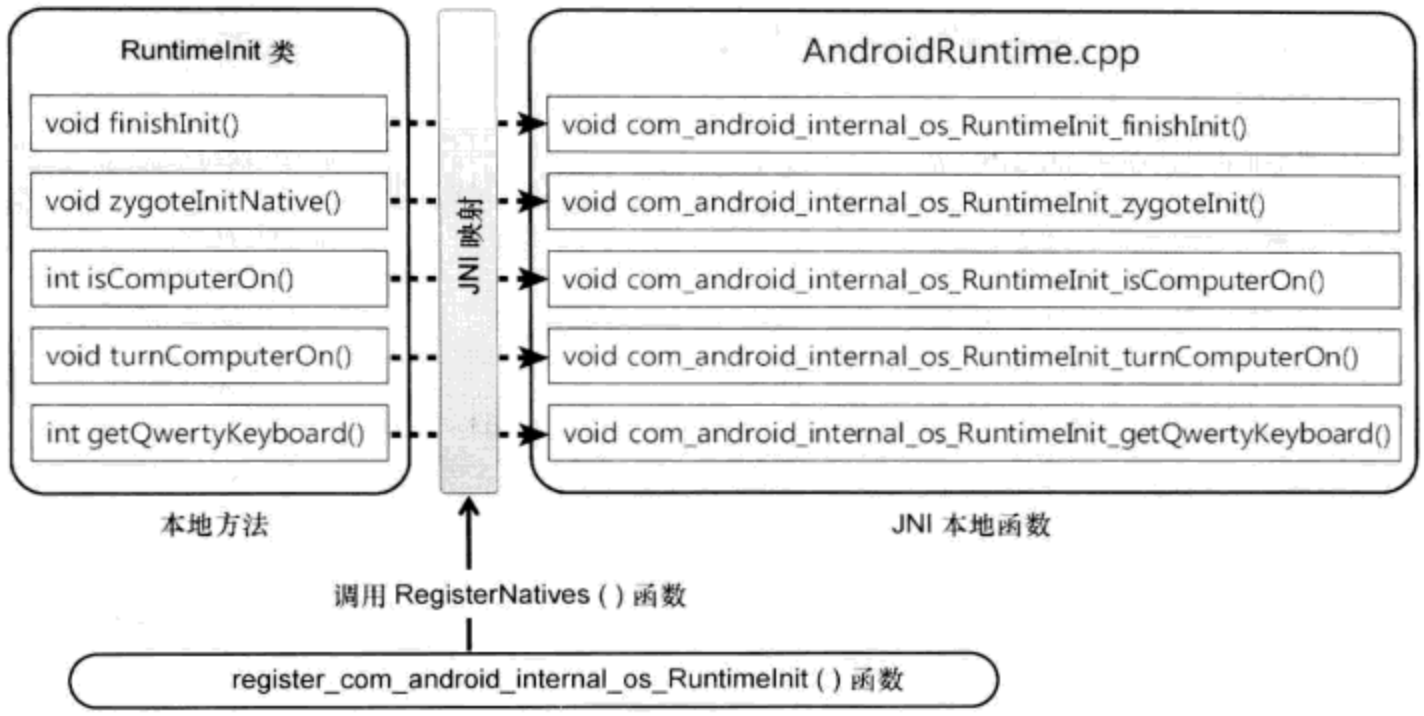


图 4-31 | Android RuntimeInit 类的本地方法与 AndroidRuntime.cpp 中的 JNI 本地函数

4.6 使用 Android NDK 开发

在 Android 平台下，应用程序的开发大部分基于 Java 语言实现的，利用谷歌提供的 Android SDK 开发工具集，开发者可以快速地开发出运行在 Dalvik 虚拟机上的应用程序。但是，有时在 Android 应用程序开发中，开发某些对性能要求较高的模块时，仍然需要使用 C/C++语言开发本地代码；或者在引用某些已经开发好的 C/C++库时，都需要使用 JNI 机制。

Android NDK（Native Development Kit）是谷歌提供的一个开发工具集，开发者利用该工具集能够轻松地开发出基于 JNI 机制的应用。在本书写作之时，Android NDK 的最新版本为 Revision 4（下称 r4），该开发工具集提供如下工具与功能。

- 包含将 C/C++源代码编译成本地库的工具（编译器、连接器等）。
- 提供将编译好的本地库插入 Android 包文件（.apk）中的功能。
- 在生成本地库时，Android 平台可支持的系统头文件与库。
- NDK 开发相关的文档、示例、规范。

TIP 使用 NDK 工具集，开发 Android 本地应用程序？

在开发者中，有些开发者使用 Android NDK 工具集，开发基于 C/C++的本地应用程序。当然，这样是完全可以的，但不建议这样做，因为 Android 系统的主要应用都是运行在 dalvik 虚拟机上的 Java 程序，希望读者多开发运行在 Android 中的 Java 程序。

更多更新编程资源尽在 http://www.pin5i.com/

如图 4-32 所示，Android NDK 先编译 C/C++本地代码，生成本地库后，将其插入到 Android 应用程序包中。但调用 JNI 编写本地代码还是由开发者来做的。

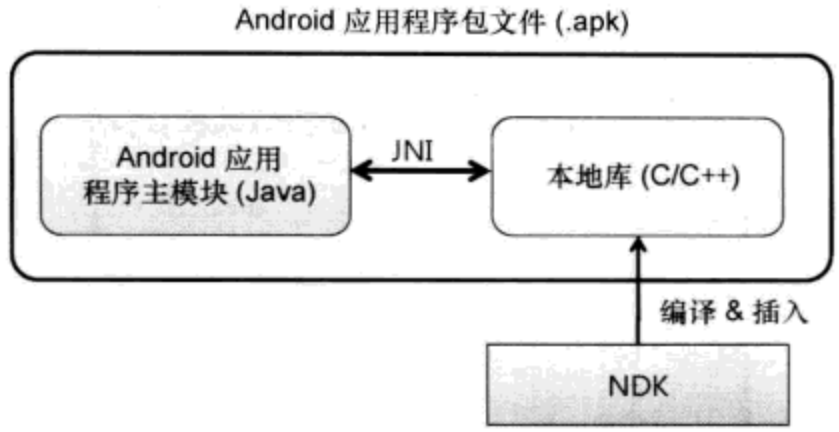


图 4-32 | Android NDK 的功能

Android NDK 提供了一些稳定可靠的系统头文件与库，如下所示。这些库在新版本的 Android 平台中会一直被支持，每次 NDK 版本更新时一些新的库会被添加进来，如 OpenGL ES 2.0 库是 NDK r3 时添加进来的，在 r4 中新添加了 libjnigraphics 库，提供 API 支持在 Android 2.2 中访问 Bitmap 对象的像素缓冲区。

libc (C library) headers

libm (math library) headers

JNI interface headers

libz (Zlib compression) headers

liblog (Android logging) header

OpenGL ES 1.1 and OpenGL ES 2.0(3D graphics library) headers

libjnigraphics (Pixel buffer access) header (for Android 2.2 and above).

A Minimal set of headers for C++ support

若应用程序使用了 NDK 不支持的系统库，即上述所列库之外的库，在升级 Android 平台时，所引用的库可能被删除或更新，导致应用程序无法启动，这点一定要注意。

接下来，将讲解如何安装 Android NDK，并通过示例讲解使用 NDK 开发应用程序的方法。由于前面已经讲解过有关 JNI 的内容，所以在学习理解 NDK 时应该不会太难。

4.6.1　安装 Android NDK

安装 Android NDK

登录以下网站，下载 NDK 压缩包，而后将其解压缩到指定的目录下。

http://developer.android.com/sdk/ndk/index.html

（目前最新版本是 r4，在 http://developer.android.com/中，可以查找到最新版本的

更多更新编程资源尽在 http://www.pin5i.com/

NDK，下载即可）

如图 4-33 所示，笔者将 NDK 安装至 C:\google\android-ndk-r4 目录下，并把此安装目录称为<NDK_HOME>，即<NDK_HOME>就是安装 NDK 的目录。

图 4-33 | NDK 的安装目录-<NDK_HOME>

安装 Cygwin

Android 是基于 Linux 的 Framework，其中运行的本地库应该是 Linux 的库文件形式（*.so）。事实上，大多数开发者都在 Windows 系统下从事 Android 开发，为了把开发的库转换编译成基于 Linux 的库，需要在 Windows 环境下安装 Cygwin。当然，如果读者本身就在 Linux 环境下从事开发，就完全没有必要安装 Cygwin。

首先从 http://www.cygwin.com 网站下载 Cywin 安装文件，下载完成后，运行其中的 setup.exe 可执行程序，开始安装 Cygwin。当出现如下画面时，请点击“下一步”。

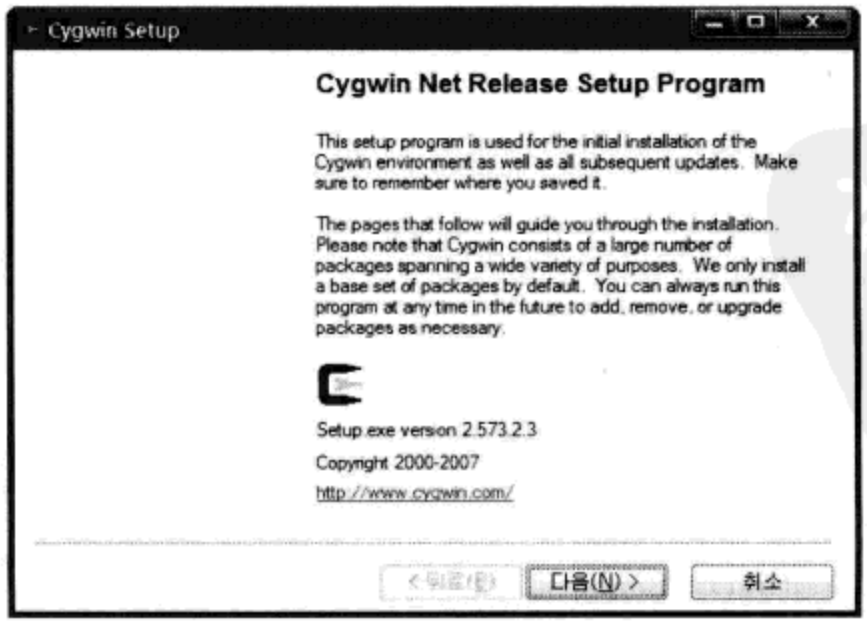


图 4-34 | 开始安装 Cywin

更多更新编程资源尽在 http://www.pin5i.com/

在选择 Cygwin 安装方法窗口中，选择“Install from Internet”（直接从网络安装，即从网络上下载完安装包后，直接安装在计算机上），单击“下一步”。

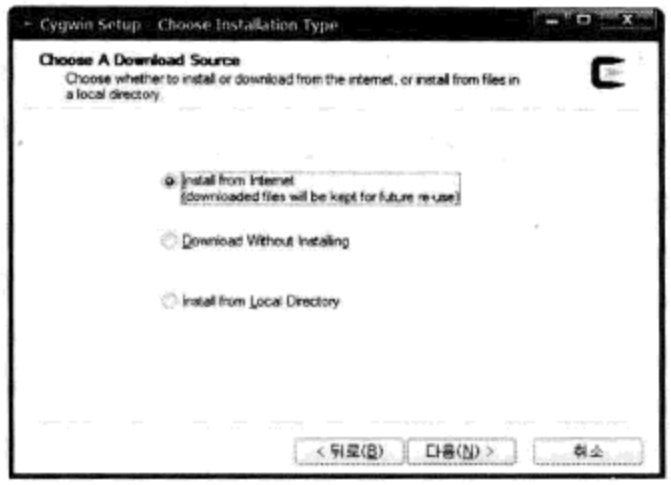


图 4-35 | 选择“Install from Internet”

在弹出的选择安装目录窗口中，保持默认安装目录（C:\cygwin）不变，该目录即是 Cygwin Shell 的根目录。其他选项也保持不变，单击“下一步”。

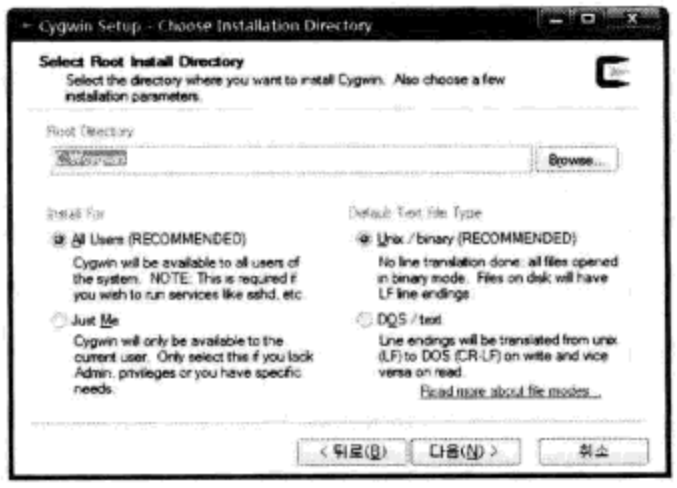


图 4-36 | 保持默认安装目录

在选择本地包目录窗口中，设置本地包保存目录，用来保存从网络上下载的 Cygwin 文件。然后单击“下一步”。

图 4-37 | 设置本地保存目录

更多更新编程资源尽在 http://www.pin5i.com/

接下来，弹出选择网络连接类型窗口。由于我们所用的网络连接大部分都是直连的，所以在窗口中，选择“Direct Connection”。

如果你使用代理服务器连接网络，请选择“Use HTTP/FTP Proxy”，而后输入代理服务器的信息。单击“下一步”。

图 4-38 | 使用代理服务器的设置

在弹出的下载站点选择窗口中，选择一个下载站点后，单击“下一步”。

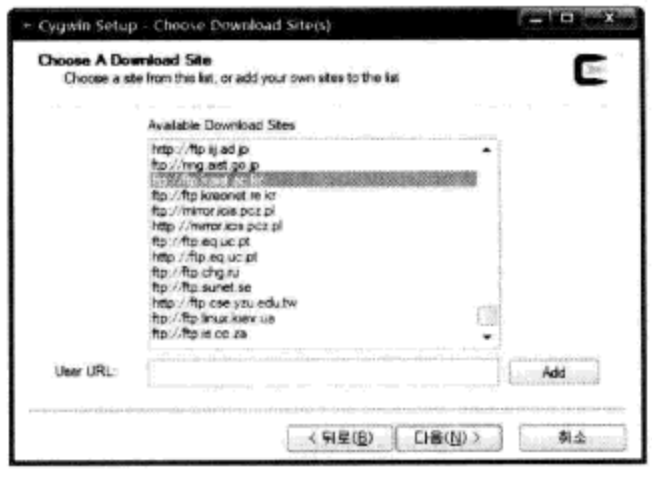


图 4-39 | 选择下载站点

接下来，在选择安装包窗口中，选择相应的包安装即可。为了使用 NDK 开发，请选择 Devel 下的 gcc-core，g++，make 三个包，进行下载安装。

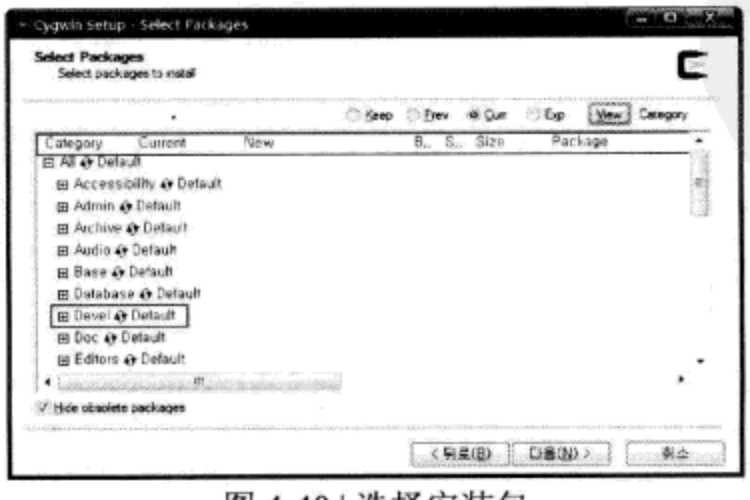


图 4-40 | 选择安装包

更多更新编程资源尽在 http://www.pin5i.com/

设置 NDK 环境

接下来，设置 NDK 使用环境。在 NDK revision 4 版本之前，设置 NDK 环境变量都非常复杂，但从 NDK revision 4 版本之后，设置起来非常简单，只要将<NDK_HOME>添加到 Windows 的 PATH 变量中即可。

图 4-41 | 向 Windows 的 PATH 变量添加<NDK_HOME>

运行 cygwin，敲入命令 ndk-build，如果 ndk-build 命令得到执行，则表示 NDK 设置正常，如下图所示（图中输出的错误是因为未指定项目，在此不必理会）。

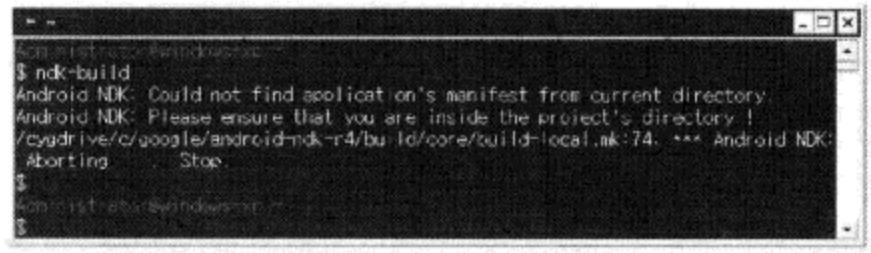


图 4-42 | 设置好 NDK 环境后，运行 ndk-build 命令

4.6.2　使用 Android NDK 开发步骤

在设置完 Android NDK 之后，在<NDK_HOME>\apps 目录下，会看到如下一些 NDK 使用示例程序。

- hello-jni：调用本地库，接收“Hello from JNI”字符串，并通过 TextView 将其输出。
- two-libs：调用本地库，返回两数之和，并通过 TextView 输出。
- san-angeles：调用本地 OpenGL ES API，渲染 3D 图片。
- hello-gl2：调用 OpenGL ES 2.0，渲染三角形。

更多更新编程资源尽在 http://www.pin5i.com/

- bitmap-plasma：一个使用本地代码访问 Android Bitmap 对象的像素缓存区的示例程序。

如果你想灵活运用 NDK，必须认真研究以上示例程序。本节将直接编写一个 Android 应用程序，用来实现与 two-libs 示例类似的功能，通过编写示例代码，向各位介绍使用 NDK 进行开发的整个过程。

在编写代码前，先分析一下整个程序的结构。整个程序大致分为两部分，一部分是使用 Java 代码编写的 Android 应用程序，另一部分是使用 C 语言编写的求两数之和的本地库。Java 代码使用 SDK 编写，代码调用 add()本地方法，求得两数之和后，通过 TextView（Android 提供的一个基本的 GUI 组件，类似于控制台）输出出来。

libndk-exam.so 共享库具体实现 add()本地方法，它由 first.c 与 second.c 两个源文件生成。add()本地方法通过 JNI 与 second.c 文件中的 Java_org_example_ndk_NDKExam_add()函数映射在一起，如图 4-43 所示。

更详细的内容，在编写代码时，将向读者介绍。

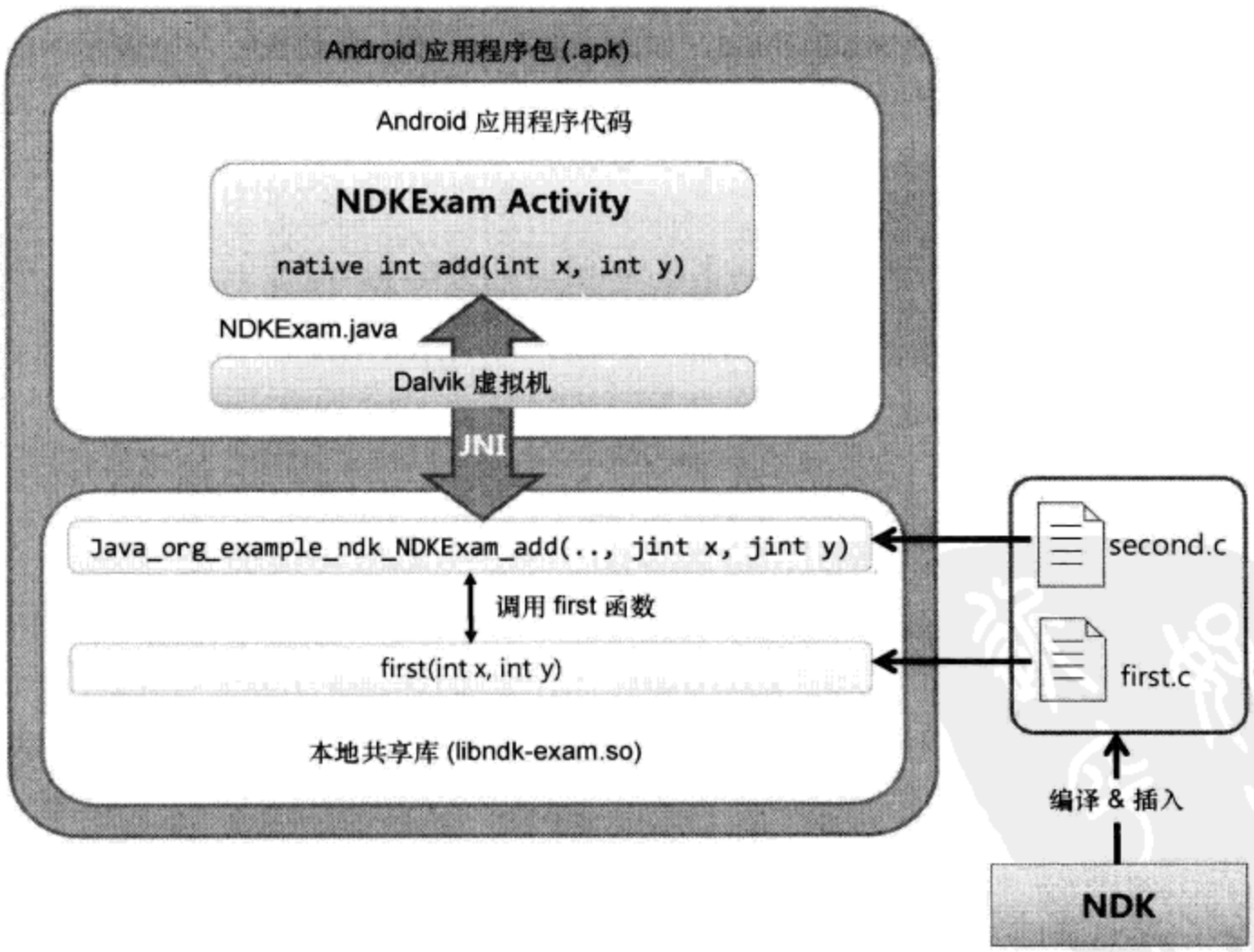


图 4-43 | NDK 示例代码组成结构

更多更新编程资源尽在 http://www.pin5i.com/

创建 Android 工程

前面，我们已经讲解过在 Java 应用程序中通过 JNI 调用 JNI 本地函数，进行程序开发的方法。同样，使用 NDK 开发 Android 应用程序有着类似的开发流程。

首先编写 Java 代码。运行已经安装好 ADT（Android Development Tools）的 Eclipse 软件，创建名称为 NDKExam 的 Android 工程，如图 4-44 所示。在本书编写时，SDK 的最新版本是 2.2，故在 Build Target 中选择“Android 2.2”。

图 4-44 | 创建 NDKExam 工程

修改 Android Activity 代码

创建完 NDKExam 工程之后，在浏览窗口中，会看到一个 NDKExam.java 文件，该文件包含程序最初运行的 Activity，双击 NDKExam.java 源文件，将其打开后，进行修改，如代码 4-26 所示。

更多更新编程资源尽在 http://www.pin5i.com/

```
package org.example.ndk;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class NDKExam extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);

   TextView  tv = new TextView(this);
   int       x  = 1000;
   int       y  = 42;

   // 调用本地方法前，装载本地库。
   System.loadLibrary("ndk-exam");

   int  z = add(x, y);

   tv.setText( "The sum of " + x + " and " + y + " is " + z );
   setContentView(tv);
}

// 本地方法声明
public native int add(int  x, int  y);
}
```

代码 4-26 | NDKExam.java 源码

在 NDKExam 代码中，调用 add()本地方法，求得两个数之和，而后将其输出到 TextView 之中。add()本地方法在 ndk-exam 库中具体实现。

如果前面讲解的 JNI，读者已经理解，那么上面这段代码读者会非常容易地理解。在以上代码中，虽然 System.loadLibrary()未被定义在静态块中，但在调用 add()本地方法之前，程序仍然会通过 System.loadLibrary()加载参数指定的库，以保证程序正常运行。

生成 JNI 本地函数原型

Java 代码编写完成后，下面开始编写 second.c 文件，该文件包含具体实现 add()本地方法的 C 函数。如前所述，映射本地方法与 JNI 本地函数的方法大致有以下两种，一种是根据 JNI 命名规则，创建 JNI 本地函数原型，以供 Java 虚拟机识别可进行 JNI 映射的函数；另一种是开发者调用名称为 RegisterNatives()的 JNI 函数，直接进行映射。

在此，采用第一种方法，即使用 javah 实用工具，快速生成与本地方法相映射的 C 函数原型。

编译 NDKExam.java 源文件，生成 NDKExam.class 字节码文件（Eclipse 默认会自

更多更新编程资源尽在 http://www.pin5i.com/

动编译 Java 源码，不需要手工编译），该文件位于<PROJECT_HOME>[1]\ bin\org\example\ndk\目录下。

在<PROJECT_HOME>目录下，执行 javah 命令，如下图所示。

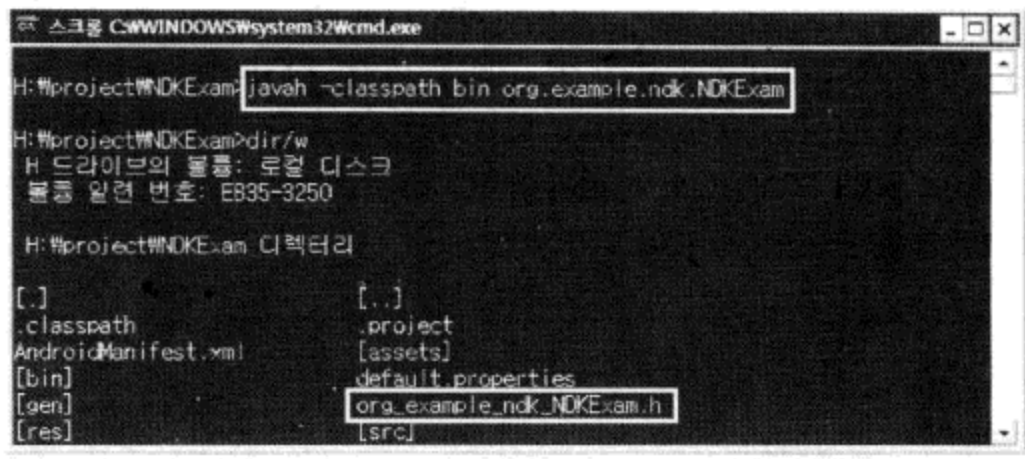


图 4-45 | 在 NDKExam 工程目录下，执行 javah 命令

执行 javah 命令后，即可查看到在<PROJECT_HOME>目录下生成了 org_example_ndk_NDKExample.h 文件。该文件中包含着与 add()本地方法相映射的 C 函数原型，打开 org_example _ndk_NDKExample.h 文件，内容如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class org_example_ndk_NDKExam */

#ifndef _Included_org_example_ndk_NDKExam
#define _Included_org_example_ndk_NDKExam
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     org_example_ndk_NDKExam
 * Method:    add
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_org_example_ndk_NDKExam_add
  (JNIEnv *, jobject, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

代码 4-27 | org_example_ndk_NDKExample.h 文件

1 <PROJECT_HOME>是 NDKExam 工程的顶层目录，笔者的<PROJECT_HOME>是指 H:\project\NDKExam 目录。

更多更新编程资源尽在 http://www.pin5i.com/

实现 JNI 本地函数

编写 second.c 文件

在 Eclipse 的菜单栏中，依次选择“File-New-Folder”菜单，在 NDKExam 工程下，新建 jni 文件夹。而后在 Eclipse 菜单栏中，依次选择“File-New-File”菜单，在 jni 文件夹下，创建 second.c 文件。在 second.c 文件中具体实现与 add()本地方法映射的名称为 Java_org_example_ndk_NDKExam_add()的 C 函数，具体实现代码如代码 4-28 所示，代码非常简单，仅调用了名称为 first()的函数。

```
#include "first.h"
#include <jni.h>

jint Java_org_example_ndk_NDKExam_add ( JNIEnv*  env,
                                        jobject  this,
                                        jint     x,
                                        jint     y )
{
   return first(x, y);
}
```

代码 4-28 | second.c 源码

编写 first.h 与 first.c

使用创建 second.c 文件的方法，在 jni 目录下，创建 first.h 与 first.c 两个文件。事实上，second.c 文件并未实现任何功能，它只是被用在处理 Android 应用程序调用 JNI 的过程中，实际的求和运算在 first.c 的 first()函数中实现。first()函数也非常简单，它接收两个整型数，计算两数之和后，将结果返回，如代码 4-29 所示。

```
// first.c

#include "first.h"

int  first(int  x, int  y)
{
    return x + y;
}
```

代码 4-29 | second.c 源码

更多更新编程资源尽在 http://www.pin5i.com/


```
// first.h

#ifndef FIRST_H
#define FIRST_H

extern int first(int  x, int  y);

#endif /* FIRST_H */
```

代码 4-30 | second.c 源码

编译本地库并插入包中

创建并编写好各个文件后，接下来使用 NDK 进行编译。NDK 自身带有编译系统，在编译与装包时，要先编写 Android.mk 文件，而后执行 ndk-bulid 脚本。Android.mk 文件拥有一套编写语法规则，在如下目录中，可以查看相关的帮助文件。

<NDK_HOME>\doc\ANDROID-MK.TXT

此外，与 ndk-build 脚本相关的内容，在如下目录中，可以查看到相关文档。

<NDK_HOME>\docs\NDK-BUILD.TXT

编写 Android.mk 文件

Android.mk 文件是一个脚本文件，它向 NDK 编译系统提供创建本地库所需用的各种信息（源代码的位置、本地库的名称等）。在 Eclipse 菜单栏中，依次选择“File-New-File”菜单，在<PROJECT_HOME>\jni 目录下，创建 Android.mk 文件，而后编写代码，如代码 4-31 所示。

```
# 指定源文件的位置
LOCAL_PATH:= $(call my-dir)    ←❶

 # 初始化与 Make 相关的环境变量
include $(CLEAR_VARS)    ←❷

# 库编译相关信息（库名、源码等）
LOCAL_MODULE    := ndk-exam    ←❸
LOCAL_SRC_FILES := first.c second.c    ←❹

# 生成共享库
include $(BUILD_SHARED_LIBRARY)  ←❺
```

代码 4-31 | Android.mk 文件

NDK 默认在<PROJECT_HOME>\jni 目录下存在 Android.mk 文件，并获取其中的

更多更新编程资源尽在 http://www.pin5i.com/

信息，所以开发者必须在<PROJECT_HOME>\jni 目录下创建 Android.mk 文件。如果你想更改 NDK 这一默认设置，请参考<NDK_HOME>\docs 目录下的 HOWTO.TXT 文件，该文件中有较为详细的说明。

下面分析一下 Android.mk 文件中的代码。

❶ 一个 Android.mk 文件首先必须定义好 LOCAL_PATH 变量，用于在开发树中查找源文件。LOCAL_PATH 变量在 Android.mk 文件的最开始被定义，若无特殊情况，一般采用如下编写形式：

LOCAL_PATH:= $(call my-dir)

$(call my-dir)用来保存 my-dir 宏函数的返回值。my-dir 是一个宏函数，由编译系统提供，用于返回包含 Android.mk 文件的目录（有关 my-dir 等宏函数的内容，请参考 ANDROID-MK.TXT 帮助文件），即将 Android.mk 文件所在的目录设置为基本目录。

一般来说，本地库的源代码与 Android.mk 文件在同一目录下，即在<PROJECT_HOME>\jni 目录下。若将$(call my-dir)返回值保存到 LOCAL_PATH 变量中，即可准确指定 NDK 编译的基本文件目录。

❷ include $(CLEAR_VARS)用来初始化 Android.mk 文件中“LOCAL_XXX”（即以 LOCAL_开头的变量，如 LOCAL_MODULE、LOCAL_SRC_FILES 等）变量，但❶中的 LOCAL_PATH 变量除外。由于 Android 编译系统会将 LOCAL_XXX 变量用作全局变量，所以需要使用该命令初始化这些变量。

❸ LOCAL_MODULE 变量必须被定义，以标识在 Android.mk 文件中描述的每个模块，即要生成的库的名称。该名称必须唯一，且不含空格，编译系统会自动产生合适的前缀和后缀，比如设置 LOCAL_MODULE 变量为 ndk-exam，编译后会生成名称为 libndk-exam.so 的共享库。

❹ LOCAL_SRC_FILES 变量必须包含将要编译打包进模块中的各个源文件。这些源文件所在的目录即是 LOCAL_PATH 变量指定的目录（即<PROJECT_HOME>\jni 目录）。例如示例中将该变量设置为 first.c，那么源文件的实际位置是<PROJECT_HOME>\jni\first.c。

❺ include $(BUILD_SHARED_LIBRARY)使用 LOCAL_MODULE、LOCAL_SRC_FILES 等变量值，创建名称为 lib$(LOCAL_MODULE).so 的共享库。

简言之，代码 4-31 中的 Android.mk 是一个脚本文件，用于将<PROJECT_HOME>\jni 下的 first.c 与 second.c 源文件编译成名称为 libndk-exam.so 的共享库。

使用 ndk-build 命令，编译生成 libndk-exam.so 库

利用 NDK 编译生成库的任务具体由<NDK_HOME>下的名称为 ndk-build 的 Shell

更多更新编程资源尽在 http://www.pin5i.com/

脚本处理执行，它是 NDK r4 新增加的命令。

ndk-build 的使用方法非常简单。首先运行 Cygwin，进入 jni 文件夹所在的工程根目录下，即进入 NDKExam 工程的根目录<PROJECT_HOME>中。而后执行 ndk-build 命令，自动编译，生成本地库。

执行 ndk-build 命令时，编译系统会在当前目录下查找 AndroidManifest.xml 文件，若该文件存在，则将当前目录看作 Android 工程目录。而后转到 jni 目录下，根据 Android.mk 进行 NDK 编译。

如图 4-46 所示，是使用 ndk-build 命令编译 NDKExam 工程的画面，编译完成后，生成 libndk-exam.so 库文件。

在 Eclipse 的包浏览窗口中，可以查看 NDKExam 整个工程的组织结构，如图 4-40 所示。观察工程的组织结构，可以看到生成的 libndk-exam.so 库文件已经被插入到 libs 目录下，如图 4-47 所示。

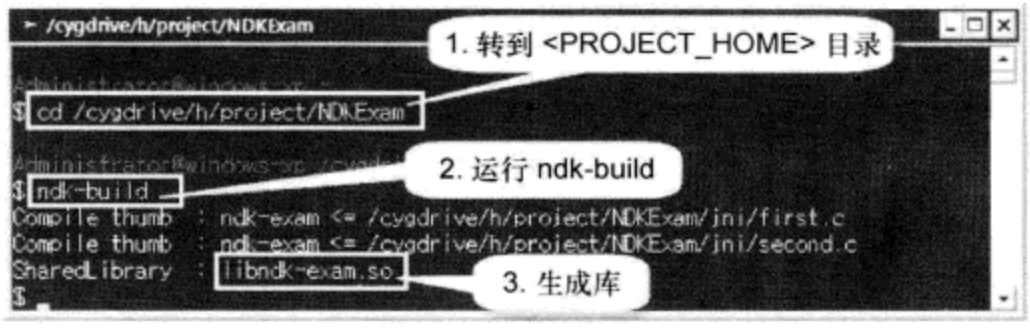


图 4-46 | 使用 ndk-build 命令，编译 NDKExam 工程

图 4-47 | NDKExam 编译后的组织结构

运行 NDKExam 应用程序

编译完 NDKExam 工程之后，在 Eclipse 中运行它，可以看到在模拟器中输出了 1000 与 42 的和，程序运行正常，如图 4-48 所示。

更多更新编程资源尽在 http://www.pin5i.com/

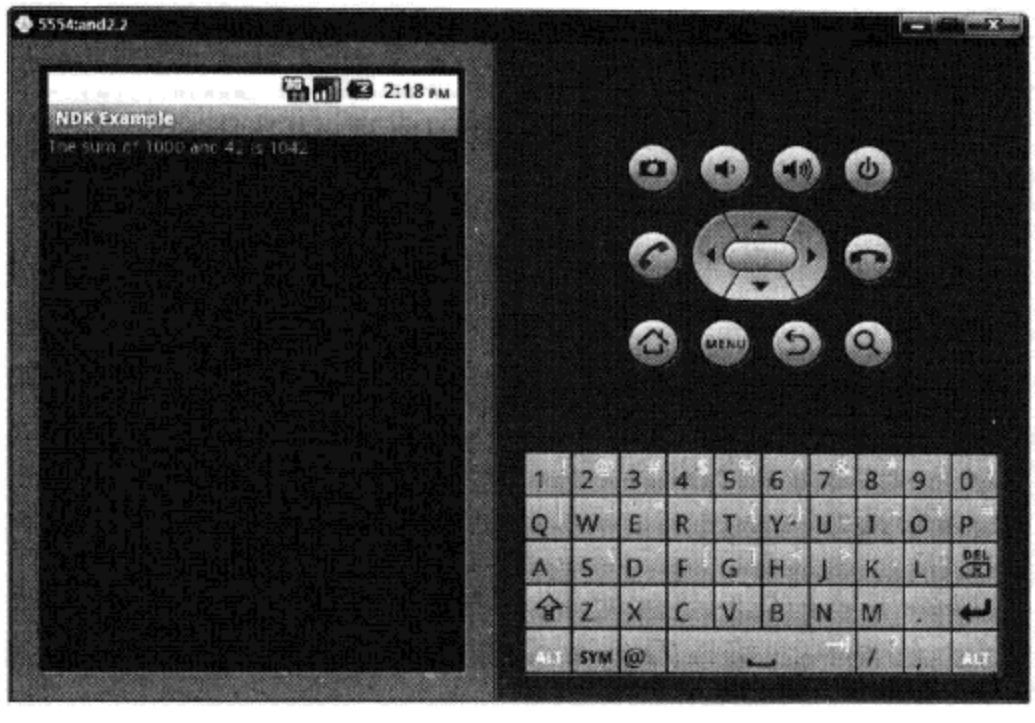


图 4-48 | 在模拟器中运行 NDKExam 应用程序

4.6.3 小结

以上我们学习了在 Android 中使用 NDK 进行开发的方法，示例程序比较简单，但结合使用前面学习的 JNI 函数，就能开发出更复杂的 NDK 应用程序来。

在 Android NDK r4 新版本中，新增了 ndk-gdb 命令，用于调试程序，此外还提供了 OpenGL ES 2.0 API 等功能。像这样，每次 NDK 版本更新都会添加一些新功能，NDK 功能不断增强，应用程度也越来越高。若想更深地了解 NDK，请认真学习安装 NDK 时附带的帮助与示例，这些示例简单但非常有用。

更多更新编程资源尽在 http://www.pin5i.com/

第 5 章

Zygote

5.1 Zygote 是什么?

从字面上看，Zygote 是“受精卵、接合子、接合体”的意思。在生物学中，受精卵是这样形成的：精子与卵子在输卵管会合后，即形成受精卵，而后受精卵开始不断分裂，在子宫内着床后继续发育，一个新生的胎儿孕育成长，并最终诞生。Zygote 是 Android 系统应用中一个相当重要的进程，它的主要功能就是执行 Android 应用程序。在 Android 系统中运行新的应用，如同卵子受精分裂一样，需要跟 Zygote 进程（拥有应用程序运行时所需要的各种元素与条件）结合后才能执行。

Zygote 进程运行时，会初始化 Dalvik 虚拟机，并启动它。Android 的应用程序是由 Java 编写的，它们不能直接以本地进程的形态运行在 Linux 上，只能运行在 Dalvik 虚拟机中。并且，每个应用程序都运行在各自的虚拟机中，应用程序每次运行都要重新初始化并启动虚拟机，这个过程会耗费相当长时间，是拖慢应用程序的原因之一。因此，在 Android 中，应用程序运行前，Zygote 进程通过共享已运行的虚拟机的代码与内存信息，缩短应用程序运行所耗费的时间。并且，它会事先将应用程序要使用的 Android Framework 中的类与资源加载到内存中，并组织形成所用资源的链接信息。新运行的 Android 应用程序在使用所需资源时不必每次重新形成资源的链接信息，这会节省大量时间，提高程序运行速度。

在 Android 平台中，使用 Zygote 进程的目的何在？下面来解答这个问题。在前面的学习中，我们知道 Android 平台是为手持内嵌设备而设计的，并且大部分手持设备都使用电池工作，其本身所拥有的资源非常有限。所以，手持设备要求内嵌系统平台必须提供一个高效的运行环境，使得应用程序在有限的资源下有更快的运行响应速度，一方面提高设备资源的利用率，另一方面尽可能地加长设备的使用时间。如前所述，Zygote 进程在“孵化”新进程时，能有效地减少系统负担，提高进程“孵化”速度，所以，Android 系统采用了 Zygote 进程。除了 Zygote 之外，Maemo 平台的 Launcher、Qt extended 的 quick launcher 等都是基于相同的目的而被使用的。

Zygote 进程是 Android 应用运行必需的进程，Zygote 进程如何运行、如何初始化，以及如何通过它来提高应用的运行速度等，这些问题是本章将讨论的主要内容。在

更多更新编程资源尽在 http://www.pin5i.com/

Android 平台下，Zygote 进程有助于快速运行应用程序，如果能理解这点，读者就能明白如何高效地向 Android 平台添加资源，快速加载相关类，并且知道在开发应用程序的过程中，运行新进程时，如何减小系统开销。

由 Zygote 孵化进程

在前面第三章中，我们学习过 init 进程，它是系统启动后运行在用户空间中的首个进程。init 进程启动完系统运行所需要的各种 Daemon 后，启动 Zygote 进程，如图 5-1 所示。Zygote 进程启动之后，Android 的服务与应用程序都由 Zygote 进程启动运行，这点可以通过查看实际设备的进程列表来确认。

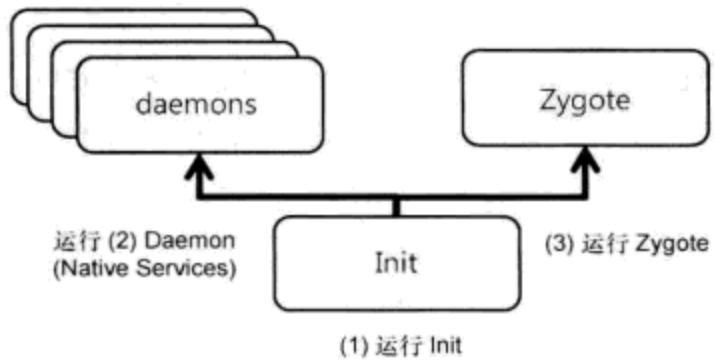


图 5-1 | Android 初始启动过程-运行 Zygote

TIP 与 Zygote 类似的 Maemo Launcher

在其他内嵌系统平台中，也使用了类似 Zygote 概念。

图 5-2 是诺基亚智能手机 Maemo 平台的架构图。与 Android 平台一样，Maemo 平台也是基于 Linux 内核的，其一大特色是包含 GNOME、GTK 等组件。其中 Maemo Launcher 组件与 Android 的 Zygote 功能类似。

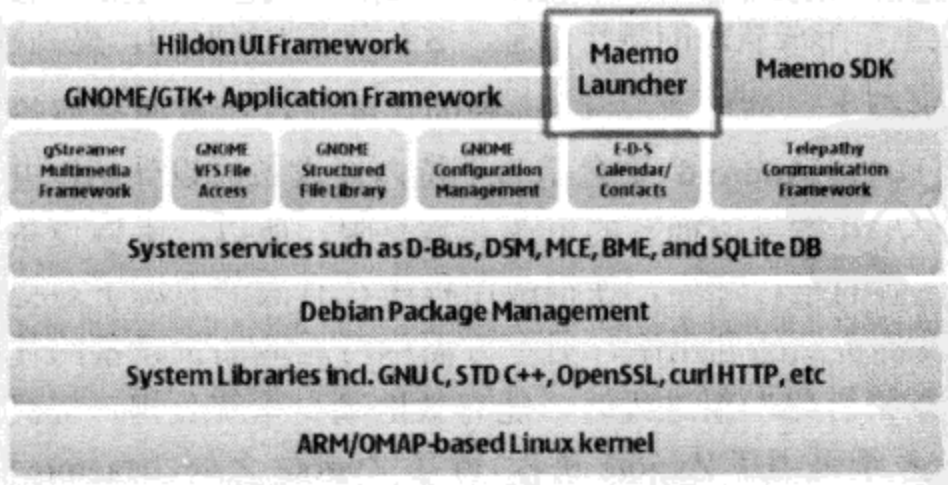


图 5-2 | 诺基亚 Maemo 平台[1]

1 诺基亚 Maemo 平台（Maemo Software，2008/6/23，http://wiki.maemo.org/User:Peterschneider/maemon_software）

更多更新编程资源尽在 http://www.pin5i.com/

Maemo Launcher 由 Maemo-invoker 与 Maemo-launcher 两部分组成，Maemo-launcher 含有应用程序运行所需要的各种数据，Maemo-invoker 快速生成并运行应用程序。

在 Android SDK 中含有一个名称为 adb（Android Debug Bridge）tool 的工具，使用它，能通过 Shell 连接到 AVD（Android Virtual Device）或实际的 Android 设备上[1]。在连接的设备上，运行 ps 命令，查看系统中的进程，如图 5-3 所示。

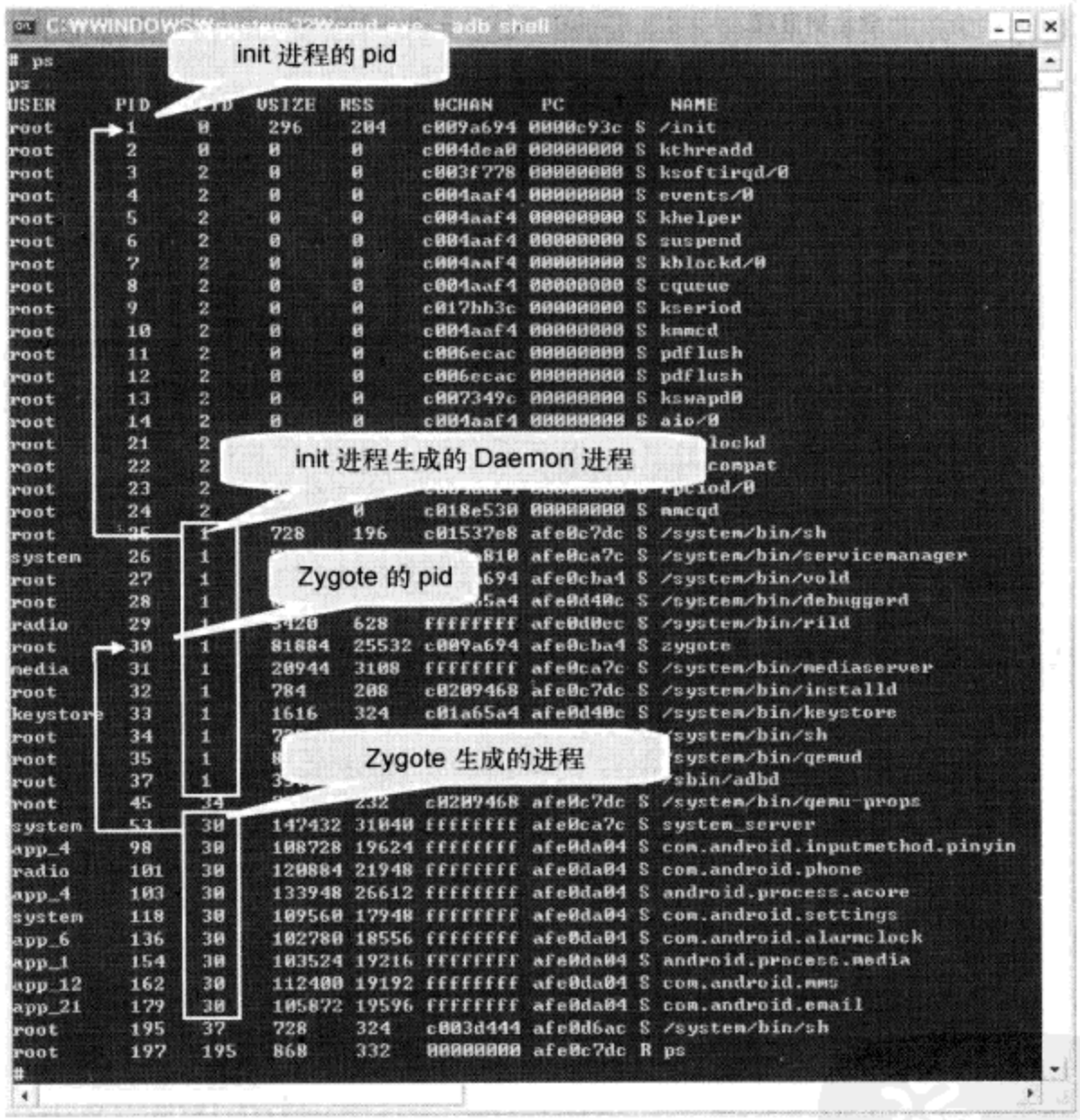


图 5-3 | 查看 Android 系统中的进程

根据父进程的 PID，Android 设备中运行的进程大致有 Daemon 进程以及在 Dalvik 虚拟机中运行的 Android 应用程序两大类。如图 5-3 所示，PID 为 1 的进程是 init 进程，其中还有 PPID（Parent PID）为 1 的进程，它们就是 init 进程启动的 Daemon 进程，Zygote 进程就是其中之一，Zygote 进程的 PID 为 30。在图 5-3 的下半部分中也可以看到一些

1　adb shell：连接到虚拟 Android 模拟器或实际设备上。

更多更新编程资源尽在 http://www.pin5i.com/

PPID 为 30 的进程，它们是 Zygote 进程的子进程，由 Zygote 进程创建并启动，大都是 Android 应用程序。

那么，在 Linux 系统中创建并运行一个进程[1]，与在 Android 系统中通过 Zygote 来创建并运行一个进程，两者究竟有何不同呢？首先，分析一下在 Linux 系统中是如何创建并运行一个新进程的，其过程如图 5-4 所示。

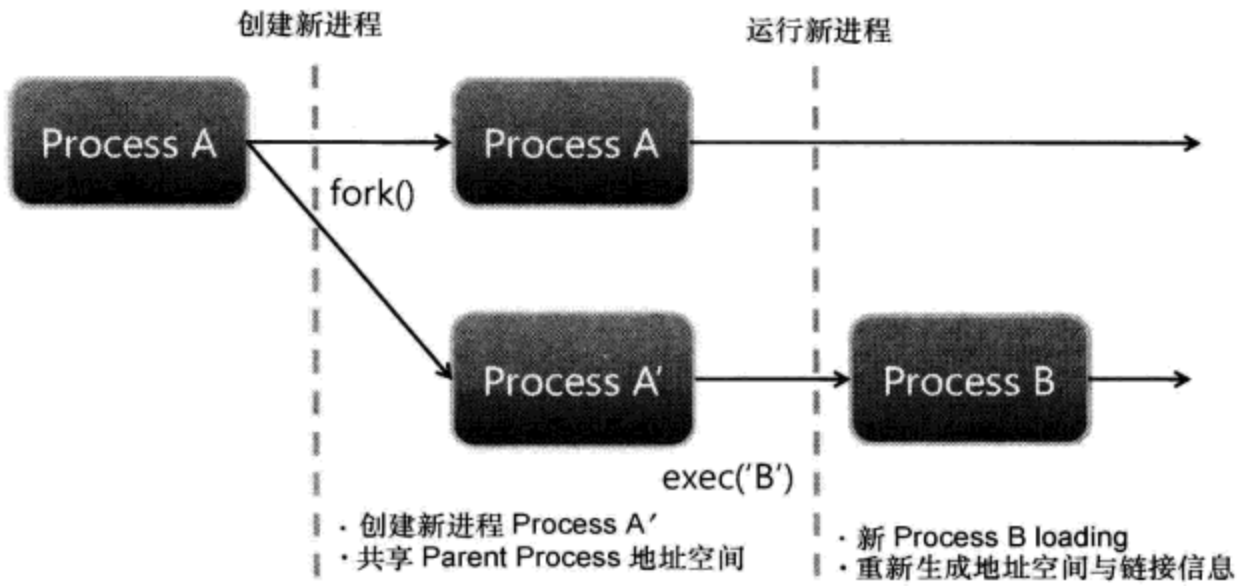


图 5-4 | 创建并运行新进程

如上图，父进程 A 调用 fork()函数创建新的子进程 A`。新创建的进程 A`共享父进程的内存结构信息与库连接信息。而后子进程 A`调用 exec(`B`)，将新进程 B 的代码加载到内存中。此时，父进程 A 的内存信息被清除，并重新分配内存，以便运行被装载的 B 进程，接着形成新的库连接信息，以供进程 B 使用。若进程 B 使用的共享库已被装载至内存，则只需更新连接信息。不然，还要添加一个步骤，即将存储器中的相关库装入内存之中。每当运行新进程时，就会重复以上过程。

那么，在 Android 中，一个新的应用进程是如何创建并运行的呢？谷歌在发布 Android 平台讲述平台特性时曾讲到，Zygote 是 Android 系统的一个主要特征，它通过 COW（Copy on Write）方式对运行在内存中的进程实现了最大程度的复用，并通过库共享有效地降低了内存的使用量（foot print）。以上说明虽然简短，但包含了大量的信息。下面通过 Zygote 进程创建新进程的整个过程来做进一步的了解。

如图 5-5 所示，Zygote 进程调用 fork()函数创建出 Zygote`子进程，子进程 Zygote`共享父进程 Zygote 的代码区与连接信息。注意，新的 Android 应用程序 A 并非通过 fork()来重新装载已有进程的代码区，而是被动态地加载到复制出的 Dalvik 虚拟机上[2]。而后，

---

1 与在 Android 平台下驱动本地进程类似。

2 应用程序 A 是要实际运行的应用程序类。Android 应用程序就是运行在 Dalvik 虚拟机上的类。

更多更新编程资源尽在 http://www.pin5i.com/

Zygote`进程将执行流程交给应用程序 A 类的方法，Android 应用程序开始运行。新生成的应用程序 A 会使用已有 Zygote 进程的库与资源的连接信息，所以运行速度很快。如图 5-6 所示，该图描述了 Zygote 运行后，新的 Android 应用程序 A 的运行过程。

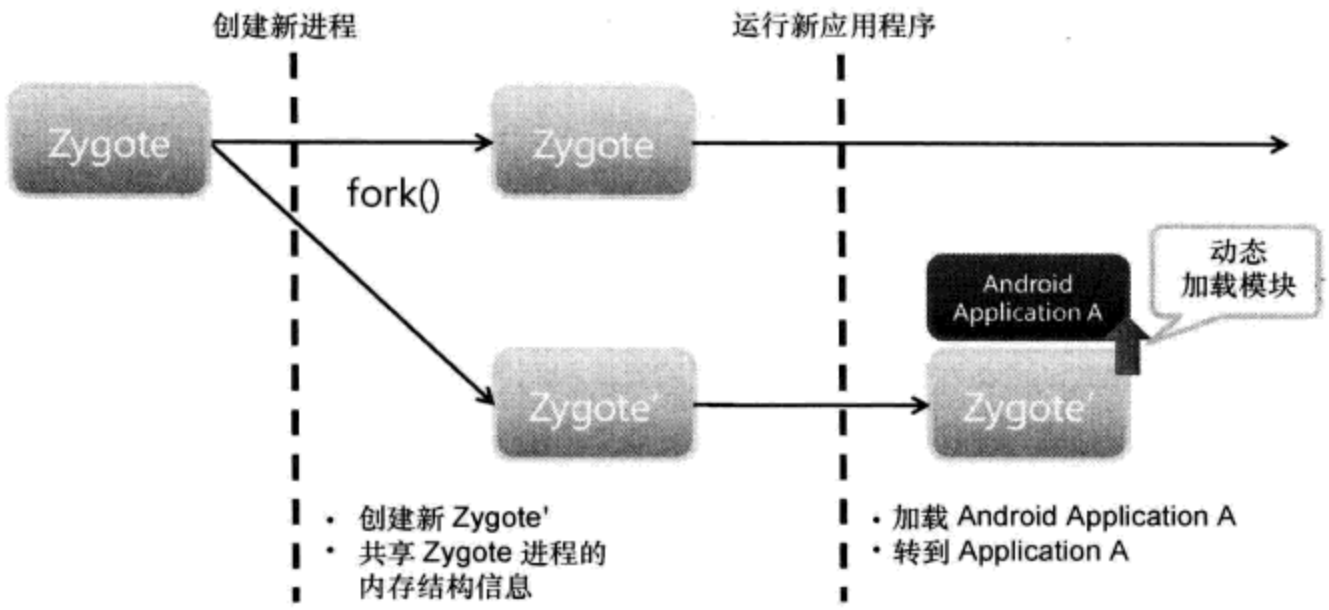


图 5-5 | 在 Android 平台上运行新应用程序的过程

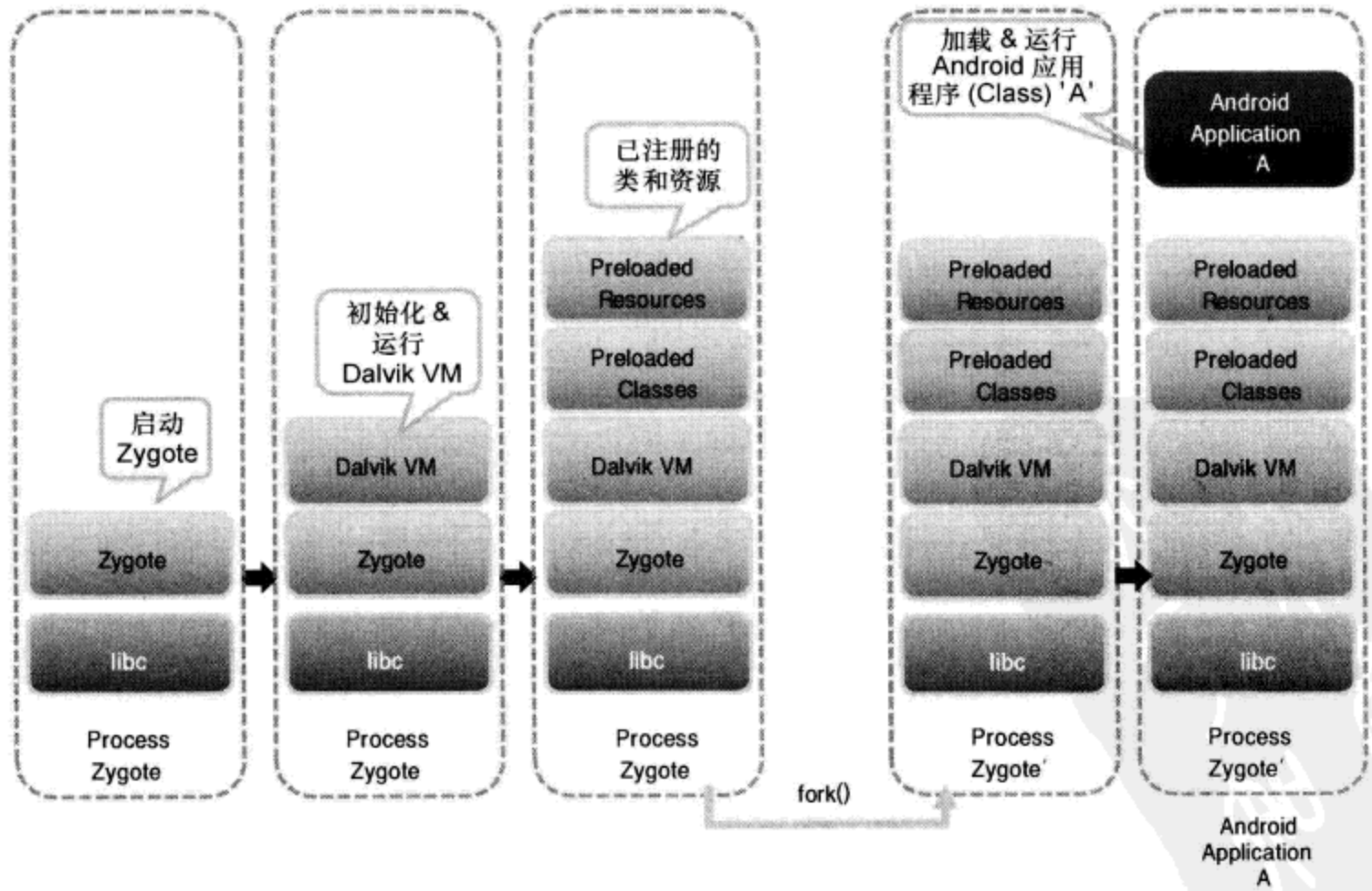


图 5-6 | Android 应用程序的运行过程

如图所示，Zygote 启动后，初始并运行 Dalvik 虚拟机，而后将需要的类与资源加载到内存中。随后调用 fork()创建出 Zygote`子进程，接着 Zygote`子进程动态加载并运

 更多更新编程资源尽在 http://www.pin5i.com/

行 Android 应用程序 A。运行的应用程序 A 会使用 Zygote 已经初始化并启动运行的 Dalvik 虚拟机代码，通过使用已加载至内存中的类与资源来加快运行速度。

以上分析了 Zygote 进程做了哪些事，应用程序运行要经历的过程。在下一章中，我们将通过实际代码来进一步学习 Zygote 运行的过程。

> **TIP** COW（Copy on Write）
>
> 在创建新进程后，新进程会共享父进程的内存空间，即新的子进程会复制所有与父进程内存空间相关的信息并使用它。COW 就是针对内存复制的一种技术。一般来说，复制内存的开销非常大，因此创建的子进程在引用父进程的内存空间时，不要进行复制，而要直接共享父进程的内存空间。而当需要修改共享内存中的信息时，子进程才会将父进程中相关的内存信息复制到自身的内存空间，并进行修改，这就是 COW（Copy on Write）技术。
>
> 要注意的是当调用 fork()直接运行 exec()时，新进程的内存空间与父进程的内存空间内容不同，此时复制父进程内存空间的做法就变得毫无意义，并且会增加新进程运行的系统开销。

5.2 由 app_process 运行 ZygoteInit class

与其他本地服务或 Daemon 不同的是，Zygote 由 Java 编写而成，不能直接由 init 进程启动运行。若想运行 Zygote 类，必须先生成 Dalvik 虚拟机，再在 Dalvik 虚拟机上装载运行 ZygoteInit 类，执行这一任务的就是 app_process 进程。

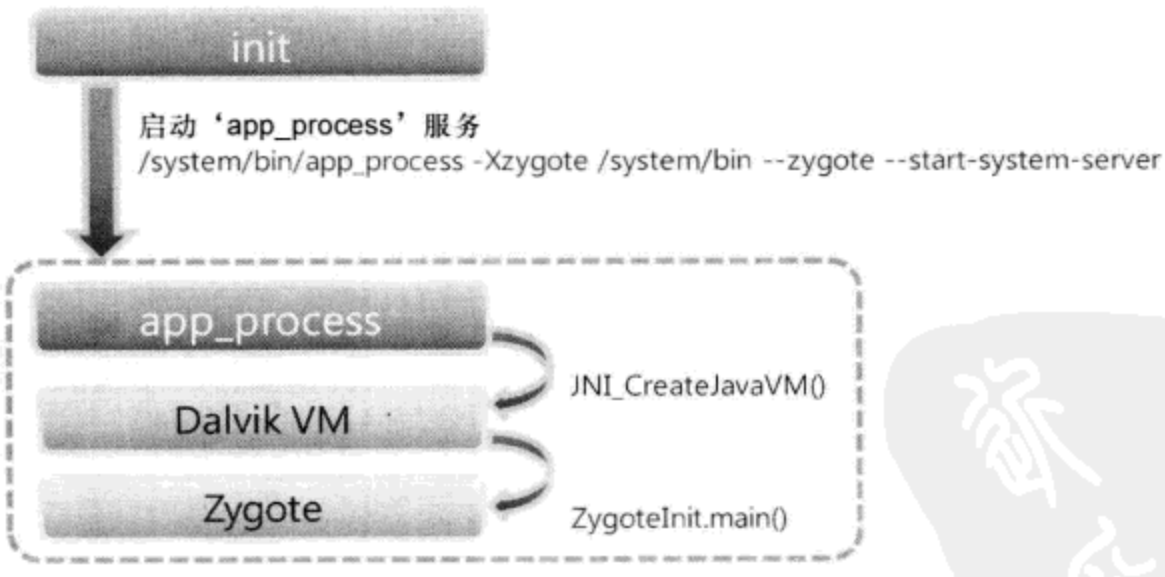


图 5-7 | 运行 app_process 进程

分析/system/bin/app_process 源码[1]中的 main()函数，可以发现 app_process 进程首先生成了一个 AppRuntime 对象，而后分析 main()函数传递进来的参数，并传递给 AppRuntime

1 frameworks/base/cmds/app_process/app_main.cpp

更多更新编程资源尽在 http://www.pin5i.com/

对象。然后生成并初始化 Dalvik 虚拟机，再调用执行 ZygoteInit 类的 main 方法。具体代码分析在下一节中进行。

5.2.1 生成 AppRuntime 对象

下面代码 5-1 是 app_process 进程的 main() 函数，执行该函数时，首先创建 AppRuntime 对象，如代码 5-1❶所示。

```
int main(int argc, const char* const argv[])
{
    …
    AppRuntime runtime;                    ❶
    …
    // Everything up to '--' or first non '-' arg goes to the vm
    int i = runtime.addVmArguments(argc, argv);    ❷

    // Next arg is parent directory
    if (i < argc) {
        runtime.mParentDir = argv[i++];    ❸
    }
    …
}
```

代码 5-1 | app_main.cpp[1]-创建 AppRuntime 对象

❶ AppRuntime 类继承自 AndroidRuntime 类，AndroidRuntime 类用于初始化并运行 Dalvik 虚拟机，为运行 Android 应用程序做好准备。❷在运行 Dalvik 虚拟机之前，通过 AppRuntime 对象，分析环境变量以及运行的参数，并以此生成虚拟机选项。

分析 init.rc 文件，可以发现 init 进程在运行 app_process 时根据如下规则传递参数，app_process 参数形式如下：

app_process [java-options] cmd-dir start-class-name [options][2]

- [java-options]：传递给虚拟机的选项，必须以“-”开始。
- cmd-dir：所要运行的进程所在的目录。
- start-class-name：要在虚拟机中运行的类的名称。app_process 会将指定的类加载到虚拟机中，而后调用类的 main()方法。

1 frameworks/base/cmds/app_process/app_main.cpp

2 带有[]，表示该项可省略。

更多更新编程资源尽在 http://www.pin5i.com/

- [options]：要传递给类的选项。

app_process 服务运行时，init.rc 文件中的运行命令如下：

/system/bin/app_process -Xzygote /system/bin --zygote --start-system-server

根据参数规则，可以知道-Xzygote 是指要传递给 VM 的选项。“-Xzygote”选项用来区分要在虚拟机中运行的类是 Zygote，还是在 Zygote 中运行的其他 Android 应用程序[1]，它会被保存在 AppRuntime 的 mOption 变量中，如代码❷行所示。在代码❸行中，运行目录参数“/system/bin”被保存到 AppRuntime 的 mParentDir 变量中。第三个参数用来指定加载到虚拟机中的类的名称，“--zygote”表示加载 com.android.internal.os.ZygoteInit 类。最后一个参数“--start-system-server”作为选项传递给生成的类，用于启动运行系统服务器。

5.2.2 调用 AppRuntime 对象

分析完要传递给虚拟机的参数，并保存到 AppRuntime 类的对象中，而后加载对象，调用对象的 main()方法。

```
int main(int argc, const char* const argv[])
{
    …
    if (i < argc) {
        arg = argv[i++];
        if (0 == strcmp("--zygote", arg)) {                    ←❶
            bool startSystemServer = (i < argc) ?
                    ➥ strcmp(argv[i], "--start-system-server") == 0 : false;
            setArgv0(argv0, "zygote");
            set_process_name("zygote");                        ←❷
            runtime.start("com.android.internal.os.ZygoteInit",
                                        ➥ startSystemServer);  ←❸
        } else {
            …
        }
        …
    }
}
```

代码 5-2 | app_main.cpp-生成 runtime 对象

在代码 5-2 的❶行中，检查类名称，根据类名称是否为“--zygote”，处理过程略微不同，但最后都是将给定的类加载至虚拟机中。假如参数传递过来的是“--zygote”，

1 在 dalvik/vm/init.c 的 dvmUsage(const char* progName)函数中可以确认实例。

更多更新编程资源尽在 http://www.pin5i.com/

程序将继续执行第❶行中的代码。

在第❸行代码中，调用 AppRuntime 的 start()成员函数，生成并初始化虚拟机，而后将 ZygoteInit 类加载至虚拟机中，执行其中的 main()方法。

如代码所示，传递给 runtime.start()函数的第一个参数“com.android.internal.os.ZygoteInit”是“完全限定名”（Full Qualified Name），简称为 FQN。在 FQN 中包含类所在的包以及类的名称，当 FQN 被传递给类加载器时，类加载器就会将包名称解析为相应路径，而后在该路径下查找并加载名称为 ZygoteInit 的类。“--start-system-server”作为最后一个参数被传递给 app_process，而后 AppRuntime 的 start()成员函数的第二个参数被设置为“true”。

5.2.3 创建 Dalvik 虚拟机

在运行 Dalvik 虚拟机之前，除了接收从 app_process 传递过来的虚拟机选项外，AppRuntime 的 start()函数还要获取与虚拟机运行相关的各种系统属性和环境变量。而后更改虚拟机的运行选项。

```
int property_get(const char *key, char *value, const char *default_value)
```

代码 5-3 | 在 libcutils 中定义的 property_get()函数

为了设置虚拟机的运行选项，需要调用 property_get()函数来访问系统中设置的相关值。代码 5-3 是 property_get()函数函数原型，用来访问系统的属性域。如前 3.6 节“属性服务”所述，系统属性域通过 init.rc 的 setprop 语句由 init 进程或其他进程进行设置。若想变更 Dalvik 虚拟机的运行选项，只需在运行虚拟机之前，参照调用 property_get()函数的代码，设置 init.rc 相关属性或通过 app_process 参数传递虚拟机的运行选项即可[1]。

```
void AndroidRuntime::start(const char* className, const bool startSystemServer)
{
    …
    if (JNI_CreateJavaVM(&mJavaVM, &env, &initArgs) < 0) {
        LOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }
    …
}
```

代码 5-4 | AndroidRuntime.cpp[2]- AndroidRuntime::start()-创建 Dalvik 虚拟机

1 与 Dalvik 虚拟机选项相关的详细内容，请参考“source\dalvik\docs\embedded-vm-control.htm”中的内容。

2 frameworks/base/core/jni/AndroidRuntime.cpp

更多更新编程资源尽在 http://www.pin5i.com/

AppRuntime 的 start()函数通过调用 JNI_CreateJavaVM()函数[1]来创建并运行虚拟机，JNI_CreateJavaVM()函数原型如下：

jint JNI_CreateJavaVM（JavaVM** p_vm，JNIEnv** p_env，void* vm_args）

- JavaVM** p_vm：生成的 JavaVM 类的接口指针
- JNIEnv** p_env：JNIEnv 类的接口指针，方便访问虚拟机
- void* vm_args：已设置的虚拟机选项

接下来，注册要在虚拟机中使用的 JNI 函数。

调用代码 5-5❷中的 startReg()函数，将调用代码❶中 static const RegJNIRec gRegJNI[]数组中的函数。注册虚拟机要使用的 JNI 函数，而后运行在虚拟机中的 Java 类就可以调用本地函数了。在 frameworks/base/core/jni 目录下的源代码中，可以查看到相关函数的定义。关于 JNI 的详细说明，请参考第 4 章 JNI 与 NDK 中的相关内容。

```
static const RegJNIRec gRegJNI[] = {                    ←❶
    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),
    …
};

void AndroidRuntime::start(const char* className, const bool startSystemServer)
{
    …
    if (startReg(env) < 0) {                            ←❷
        LOGE("Unable to register all android natives\n");
        goto bail;
    }
    …
}
```

代码 5-5 | AndroidRuntime.cpp- AndroidRuntime::start()-注册 JNI 本地函数

5.2.4 运行 ZygoteInit 类

在创建完 VM 之后，接着加载要运行的类。如前所述，根据参数的不同，app_process 也会加载执行 Zygote 以外的其他类。如代码 5-6 所示，AppRuntime 的 start()函数会查

1 dalvik/vm/Jni.c- jint JNI_CreateJavaVM(JavaVM** p_vm,JNIEnv** p_env,void* vm_args)

更多更新编程资源尽在 http://www.pin5i.com/

找指定的类，并调用指定类的 main()方法。

```
…
jclass startClass;
jmethodID startMeth;

slashClassName = strdup(className);
for (cp = slashClassName; *cp != '\0'; cp++)          ←❶
   if (*cp == '.')
      *cp = '/';
startClass = env->FindClass(slashClassName);          ←❷
if (startClass == NULL) {
   LOGE("JavaVM unable to locate class '%s'\n", slashClassName);
   /* keep going */
} else {
   startMeth = env->GetStaticMethodID(startClass, "main",          ←❸
         "([Ljava/lang/String;)V");
   if (startMeth == NULL) {
      LOGE("JavaVM unable to find main() in '%s'\n", className);
      /* keep going */
   } else {
      env->CallStaticVoidMethod(startClass, startMeth, strArray);  ←❹
   }
}
…
```

代码 5-6 | AndroidRuntime.cpp- AndroidRuntime::start()-加载 ZygoteInit 类

为了观察 Zygote 的运行情况，传递给 className 的值为 com.android.internal.so. ZygoteInit。

在❶行中，首先将类名称中的“.”替换成“/”。而后在❷行中调用 FindClass()函数，在由类名解析出的路径下查找指定的类。若指定的类存在，则会加载它，而后如❸所示调用 GetStaticMethodID()函数在类中查找形式参数为 String 数组且返回值为 Void 的 main()静态方法[1]。❹若找到指定类的 main()方法，则调用 CallStaticVoidMethod()函数，此时程序的执行转到虚拟机中运行的 Java 应用程序上（此处是 ZygoteInit 类），后面本地域中的 C++代码执行流会一直等待，直到虚拟机终止运行。

5.3 ZygoteInit 类的功能

至此，我们已经创建好虚拟机，并将 ZygoteInit 类装载到虚拟机中。接下来，ZygoteInit 类会被运行，那么 ZygoteInit 类具体拥有哪些功能呢？概括如图 5-8 所示。

1 请参考第 4 章中有关 JNI 签名的内容。

更多更新编程资源尽在 http://www.pin5i.com/

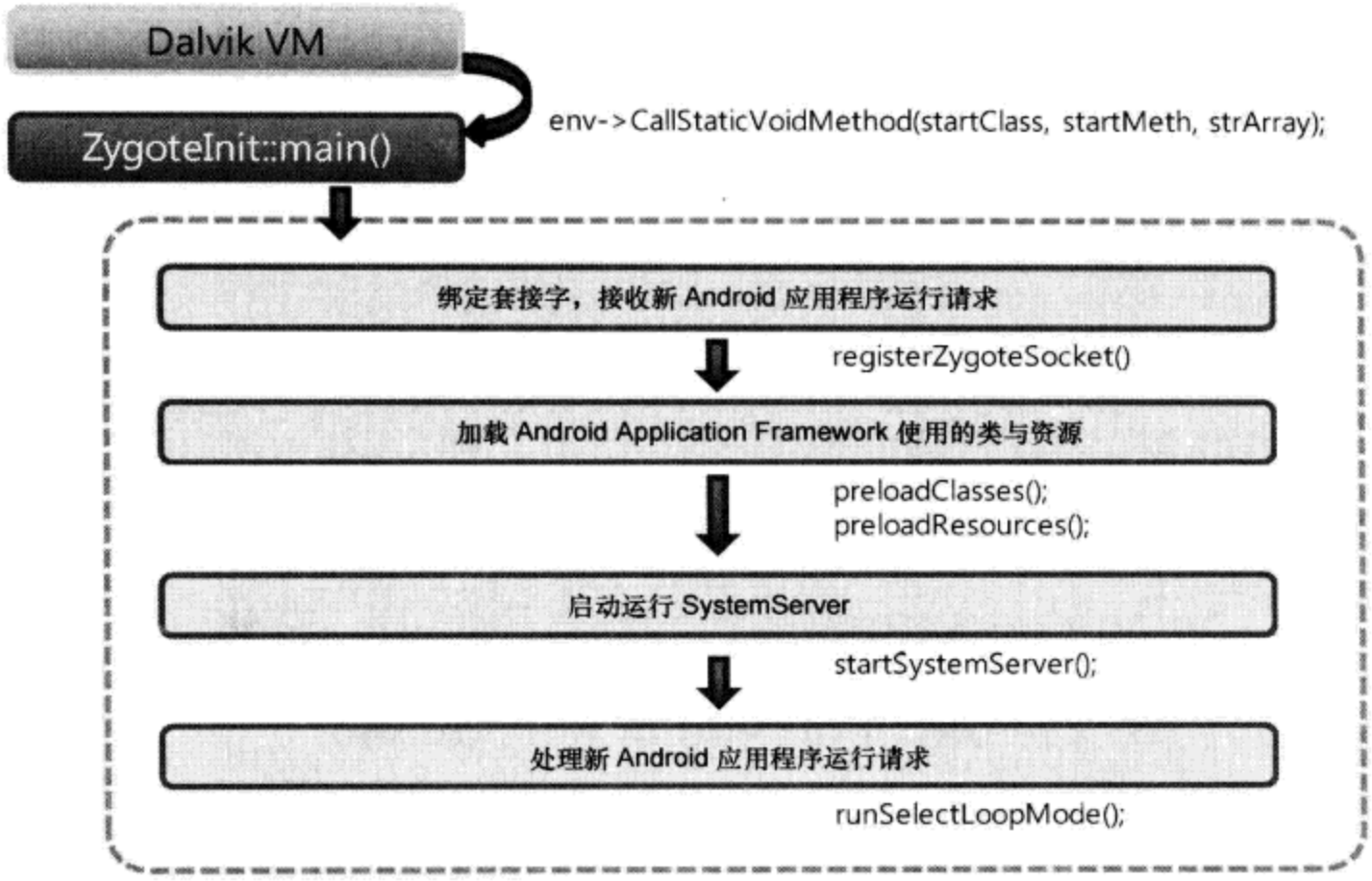


图 5-8 | ZygoteInit::main()方法的功能

下面通过 ZygoteInit 类的源代码，分析其功能。

```
public static void main(String argv[]) {
    try {
        // 绑定套接字，接收新 Android 应用程序运行请求
        registerZygoteSocket();          ❶
        …
        // 加载 Android Application Framework 使用的类与资源
        preloadClasses();                ❷
        preloadResources();
        …
        // 运行 SystemServer
        if (argv[1].equals("true")) {
            startSystemServer();         ❸
        }
        …
        if (ZYGOTE_FORK_MODE) {
            runForkMode();
        } else {
            // 处理新 Android 应用程序运行请求
            runSelectLoopMode();         ❹
        }
        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
```

更多更新编程资源尽在 http://www.pin5i.com/


```
            closeServerSocket();
            throw ex;
        }
    }
```

代码 5-7 | ZygoteInit.java-ZygoteInit::main()方法

❶ 此行语句用来绑定套接字，以便接收新 Android 应用程序的运行请求。为了从 ActivityManager 接收新 Android 应用程序的运行请求，Zygote 使用 UDS（Unix Domain Socket），init 进程在运行 app_process 时，使用 init.rc 文件中以“/dev/zygote”形式注册的套接字。

❷ 该行用于将应用程序框架中的类、平台资源（图像、XML 信息、字符串等）预先加载到内存中。新进程直接使用这些类与资源，而不需要重新加载它们，这大大加快了程序的执行速度。

❸ 通过 app_process 运行 zygote 时，参数“--start-system-server”会调用 startSystemServer()方法启动系统服务器，系统服务器用来运行 Android 平台需要的一些主要的本地服务[1]。

❹ 最后，监视 UDS，若收到新 Android 应用程序生成请求，则进入处理循环。

以上通过代码，分析了 ZygoteInit 类的 main()方法实现的四种功能。接下来，将详细地分析各个功能。

5.3.1 绑定/dev/socket/zygote 套接字

ZygoteInit 类使用由/dev/socket/zygote 生成的 UDS 套接字，从 ActivityManager 接收新 Android 应用程序的生成请求。该套接字在系统启动过程中由 init 进程生成，在 init.rc 文件中有生成该套接字的相关内容。代码 5-8 是 init.rc 文件中有关 zygote service 定义的部分，套接字的名称、种类、访问权限在第二行中标出。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
```

代码 5-8 | init.rc-定义 zygote service

1 SystemServer 主要运行 SurfaceFlinger、AudioFlinger、MediaPlayerService、CameraService 等服务，并调用 com.android.server.SystemServer 类的 init2() 方法。

更多更新编程资源尽在 http://www.pin5i.com/

ZygoteInit 类的 main()方法首先调用 registerZygoteSocket()方法，代码如下：

```
public class ZygoteInit {
    …
    private static final String ANDROID_SOCKET_ENV = "ANDROID_SOCKET_zygote";
    private static LocalServerSocket sServerSocket;
    …
    private static void registerZygoteSocket() {
        …
        if (sServerSocket == null) {
            int fileDesc;
            String env = System.getenv(ANDROID_SOCKET_ENV);          ❶
            fileDesc = Integer.parseInt(env);
            sServerSocket = new LocalServerSocket(
                    createFileDescriptor(fileDesc));                 ❷
        }
    }
    …
}
```

代码 5-9 | ZygoteInit::registerZygoteSocket()

在代码 5-9 的 registerZygoteSocket()方法中，第❷行代码会创建一个 LocalServerSocket 类的对象并将其赋值给 sServerSocket 静态变量中。

代码的第❶行调用 System.getenv()方法，获取套接字的文件描述符，该套接字由 init 进程记录在 ANDROID_SOCKET_zygote 环境变量中。应用程序 Framework 使用套接字文件描述符创建 LocalServerSocket 类的实例，并将其与/dev/socket/zygote 绑定在一起。

当创建一个新 Android 进程的请求到达 ZygoteInit 对象时，创建出 LocalServerSocket 实例接收生成新 Android 进程的信息，并在最后的循环语句中进行处理。

5.3.2 加载应用程序 Framework 中的类与平台资源

ZygoteInit 类会调用 preloadClasses()与 preloadResources()两个方法，这两个方法分别用于将应用程序 Framework 中的类，以及图标、图像、字符串等资源加载到内存之中，并对装载的类与资源生成链接信息。新生成的 Android 应用程序在使用这些已经装载的类或资源时，直接使用即可，不需要重新生成链接信息[1]。

此外，还加载许多其他的类，比如包含图形相关类的 android.graphics，以及包含

1 预加载（preload）的类目录被定义在 frameworks/base/preloaded-classes 文件中，有关各个类的信息，请参考 Android 开发者网站。

更多更新编程资源尽在 http://www.pin5i.com/

通信相关类的 android.net。在 Android 开发者网站上公开了大部分的包与类，感兴趣的读者可以参考一下该网站。

preloadClasses()方法

preloadClasses()方法的主要代码如下：

```
…
private static final String PRELOADED_CLASSES = "preloaded-classes";
…
private static void preloadClasses() {
    …
    InputStream is =
        ZygoteInit.class.getClassLoader().getResourceAsStream(          ←❶
              PRELOADED_CLASSES);
    …
    BufferedReader br                    ←❷
        = new BufferedReader(new InputStreamReader(is), 256);

    int count = 0;
    String line;
    while ((line = br.readLine()) != null) {
        line = line.trim();              ←❸
        if (line.startsWith("#") || line.equals("")) {
            continue; }
        …
        Class.forName(line);             ←❹
        …
    }
}
```

代码 5-10 | ZygoteInit.java- ZygoteInit::preloadClasses()方法

代码 5-10 选取的仅是 preloadClasses()方法代码的主要部分，下面分析一下。

在第❶行中，获取一个输入流，以便读取“preloaded-classes”文件中记录的类。“preloaded-classes”文件的部分内容如代码 5-11 所示。

```
# Classes which are preloaded by com.android.internal.os.ZygoteInit.
# Automatically generated by frameworks/base/tools/preload/WritePreloadedClassFile.java.
# MIN_LOAD_TIME_MICROS=1250
SQLite.Blob
SQLite.Database
SQLite.FunctionContext
SQLite.Stmt
SQLite.Vm
```

更多更新编程资源尽在 http://www.pin5i.com/

```
android.R$styleable
android.accounts.IAccountsService$Stub
android.app.Activity

…
```

代码 5-11 | frameworks/base/preloaded-classes 文件

在第❷部分中，在获取的输入流的基础上，创建 BufferedReader 对象，并读取“preloaded-classes”文件的内容。

第❸部分语句用来忽略所读内容中的注释与空行，而后开始读取下一行。

在代码的第❹行中，调用 Class.forName()方法，将读到的类动态地加载到内存中。Class.forName()方法并非真在内存中生成指定类的实例，它只是把类的信息加载到内存中，并初始化静态变量。

新应用程序运行时，由于使用的类已经被加载到内存中，所以程序的启动运行速度会相当快。那么，预先把要使用的类加载到内存中，究竟有多大的效果呢？如果应用程序生成时，所使用的类未被加载到内存之中，虚拟机就会将需要的类加载到内存中。此时，虚拟机都需要在存储设备中查找相关类的信息，而后将其加载到内存中，并形成已加载类的链接信息。如果所需要的类一个也没有被加载到内存中，加载类时所耗费的时间，要比使用 preloadClasses()加载类耗费的时间要多。使用 logcat 命令，可以查看在加载 preloaded-classes 文件中的类时所耗费的时间。

```
# logcat -d
…
I/Zygote  (   30): Preloading classes...

…
D/dalvikvm(   30): Trying to load lib /system/lib/libmedia_jni.so 0x0
D/dalvikvm(   30): Added shared lib /system/lib/libmedia_jni.so 0x0

…
I/Zygote  (   30): ...preloaded 1942 classes in 18202ms.

…
```

图 5-9 | 使用 logcat 命令，查看类的加载所耗费的时间

如图 5-9 所示，执行 logcat 命令，可以看到加载 1942 个类所耗费的时间为 18202ms（在不同的测试环境下，所耗费的时间略有不同）。像这样，ZygoteInit 类将常用的类一

更多更新编程资源尽在 http://www.pin5i.com/

次性加载进内存中，每次程序运行时不必重新加载，大大缩短了程序启动时间，提高了程序的运行速度。如果不采用预加载处理，应用程序每次运行都需要额外耗费18202ms 从存储设备中加载所需要的类。

加载应用程序 Framework 中包含的资源

在 Android 应用程序 Framework 中使用的字符串、颜色、图像文件、音频文件等都被称为资源。应用程序不能直接访问这些资源，需要通过 Android 开发工具自动生成的 R 类来访问。通过 R 类可访问的资源组成信息记录在 XML 中。Android 资源大致分为两大类，如下：

- Drawable

这类资源是指背景画面、照片、图标等可在画面中绘画的资源。preloadResource 会加载按钮图片、按钮组等基本主题图像。如图 5-10 所示，Android 应用程序 Framework 中包含 CheckBox、Button、Editbox、Call 等图像文件。

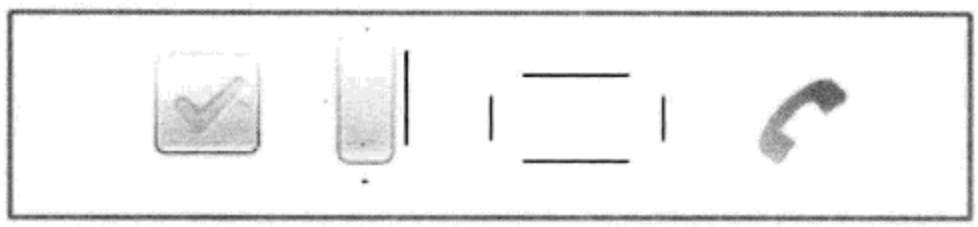


图 5-10 | Drawable 资源

- XML 管理的资源

XML 管理的资源有保存字符串的 strings.xml、保存字符串数组的 arrays.xml，以及保存颜色值的 colors.xml 等。此外，动画、布局等资源也是由 XML 文件管理的。代码 5-12 是一个保存颜色信息的 XML 文件。

```
<?xml version="1.0" encoding="utf-8"?>
<array name="preloaded_color_state_lists">
     <item>@color/hint_foreground_dark</item>
     <item>@color/hint_foreground_light</item>
     <item>@color/primary_text_dark</item>
     …
     <item>#ff000000</item>
     <item>#00000000</item>
     <item>#ffffffff</item>
</array>
```

代码 5-12 | 记录在 XML 中的 ColorState 资源

与预加载类相似，Android 应用程序也会预先加载使用的资源，这样在使用这些资源，将它们由存储设备加载到内存时，会大大缩短加载的时间，提高运行效率。

更多更新编程资源尽在 http://www.pin5i.com/

preloadResources 方法

preloadResources 方法是 ZygoteInit 类的一个方法，用于加载指定资源，其代码如 5-13 所示。

```
private static void preloadResources() {
    …
    mResources = Resources.getSystem();          ❶
    mResources.startPreloading();                ❷

    if (PRELOAD_RESOURCES) {

        TypedArray ar = mResources.obtainTypedArray(
                com.android.internal.R.array.preloaded_drawables);
        int N = preloadDrawables(runtime, ar);                          ❸
        …
        ar = mResources.obtainTypedArray(
                com.android.internal.R.array.preloaded_color_state_lists);
        N = preloadColorStateLists(runtime, ar);
    }
    mResources.finishPreloading();
}
```

代码 5-13 | ZygoteInit.java-preloadResources()

资源分为系统资源与应用程序资源。在访问系统资源时，需要先调用 Resources 类的静态方法 getSystem()，如代码第❶行，而后使用其返回的对象，预加载系统资源。

加载资源时，有时会出现因重复调用而造成资源重复加载的情形，为了避免出现这种情况，定义了一个成员变量，用来记录资源加载的状态。若指定资源已经被加载，就会抛出 IllegalStateException 异常，并终止方法的执行。

接下来，加载 Drawable 与 Color State 资源，frameworks/base/core/res/res/values/arrays.xml 文件中记录的 preloaded_drawables 与 preloaded_color_state_lists 会被加载到内存中。

与类加载相似，使用 logcat 命令，可以查看资源加载时耗费的时间，以及加载的资源个数。执行该命令，可以看到系统共加载了 63 个资源，总耗费的时间约为 1823ms。通过资源预加载，每当应用程序启动时，重复的资源不会再次被加载，这为后续应用程序的运行提供了便利，大大提高了程序的启动速度。

至此，Dalvik 虚拟机已经启动并完成初始化，还绑定了套接字，以便接收应用程序创建请求。并且，包含在应用程序 Framework 中的类与资源也被加载到内存之中。由此 ZygoteInit 类做好了接收请求创建应用程序并运行的准备。但 ZygoteInit 类在处理应用程序创建请求之前，还有一项工作就是运行 SystemServer。

更多更新编程资源尽在 http://www.pin5i.com/

5.3.3 运行 SystemServer

首先，让我们一起回顾一下 Android Framework 的启动过程，如下图所示。

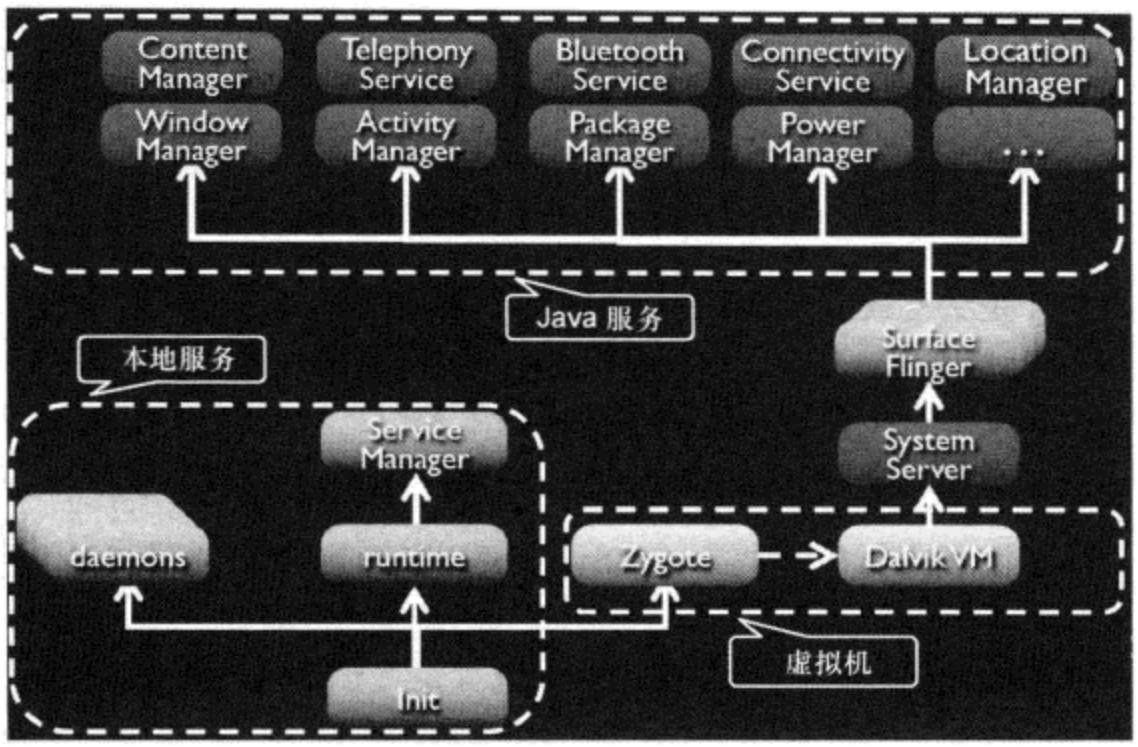


图 5-11 | Android Framework 的启动过程[1]

Zygote 启动 Dalvik 虚拟机后，会再生成一个 Dalvik 虚拟机实例，以便运行名称为 SystemServer 的 Java 服务[2]，SystemServer 用于运行 Audio Flinger 与 Surface Flinger 本地服务。在运行完所需的本地服务之后，SystemServer 开始运行 Android Framework 的服务，如 ActivityManager（管理 Android 应用程序的 Activity）、PackageManager（设置或安装应用程序）等。

```
private static boolean startSystemServer()
            throws MethodAndArgsCaller, RuntimeException {
      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",          ← ❶
      };
```

1 Android Framework 的启动过程（Google I/O session，2008，Android-Anatomy- GoogleIO.pdf，http://sites. google.com/site/io/anatomy-physiology-of-an-android）。

2 在 Google 的 Android 文档中，本地服务与库采用草绿色标识，Dalvik 虚拟机使用黄色标识，Java 服务使用蓝色标识。

更多更新编程资源尽在 http://www.pin5i.com/


```
        …
        pid = Zygote.forkSystemServer(
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids, debugFlags, null);          ←❷
        …
        if (pid == 0) {
           handleSystemServerProcess(parsedArgs);               ←❸
        }

        return true;
    }
```

代码 5-14 | ZygoteInit.java[1]-startSystemServer()

❶ 代码 5-14 是 ZygoteInit 源码中 startSystemServer()方法的一部分，startSystemServer()方法用于运行 SystemServer。代码❶定义了一个字符串数组，该数组保存 SystemServer 的启动参数，这些参数被硬编码进字符串数组。在字符串数组中，最后一个参数 com.android.server. SystemServer 用于指定 SystemServer 类。

❷ 与运行其他应用程序不同，startSystemServer()方法会调用 forkSystemServer()方法来创建新进程，并运行 SystemServer。系统在运行普通 Android 应用程序时，只负责创建应用程序进程，至于进程是否创建成功并不检查。与此不同，SystemServer 是必须运行的，因此在 forkSystemServer()方法中必须检查生成的 SystemServer 进程工作是否正常。

❸ 在生成的 SystemServer 进程中运行 com.android.server.SystemServer 类的 main()方法。执行 main()方法时会加载名称为 android_servers 的本地库，如代码 5-15 所示。

```
public static void main(String[] args) {
    // The system server has to run all of the time, so it needs to be
    // as efficient as possible with its memory usage.
    VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);
    System.loadLibrary("android_servers");
    init1(args);
}
```

代码 5-15 | SystemServer.java[2]-main()方法

本地库加载完毕后，继续调用 init1()函数，它是一个 JNI 本地方法。init1()函数调用 system_init()函数，启动 Audio Flinger、Surface Flinger、MediaPlayerService、CameraService

1 frameworks/base/core/java/com/android/internal/os

2 frameworks/base/services/java/com/android/server

更多更新编程资源尽在 http://www.pin5i.com/

等本地服务[1]，如代码 5-16 所示。

```
extern "C" status_t system_init()
{
    …
    if (strcmp(propBuf, "1") == 0) {
        SurfaceFlinger::instantiate();                    ←❶
    }

    if (!proc->supportsProcesses()) {
        AudioFlinger::instantiate();
        MediaPlayerService::instantiate();
        CameraService::instantiate();
    }

    AndroidRuntime* runtime = AndroidRuntime::getRuntime();

    runtime->callStatic("com/android/server/SystemServer", "init2");   ←❷
    if (proc->supportsProcesses()) {
        …
        ProcessState::self()->startThreadPool();
        IPCThreadState::self()->joinThreadPool();
        …
    }
    return NO_ERROR;
}
```

代码 5-16 | system_init.cpp[2]-system_init()

本地服务注册完毕后，再次调用 SystemServer 类的静态方法 init2()，如代码行❷。init2() 方法创建 android.server.ServerThread 线程，并启动它，从而运行 Android Framework 的主要服务，如代码 5-17 所示。

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

代码 5-17 | SystemServer.java[3]-init2()

当 ServerThread 线程创建好并运行后，Thread 类的 run()方法被调用，此时 ServerThread 类中重定义的 run()方法被调用。在 run()方法中包含主要服务生成与注册的代码，其中

1 这些服务将在后续章节中进行讲解，此处大致了解即可。

2 frameworks/base/cmds/system_server/library

3 frameworks/base/services/java/com/android/server

更多更新编程资源尽在 http://www.pin5i.com/

各个服务已超出本章的讨论范围，在此省略。

但在此能确定各个服务启动的时间点。当所有服务正常启动后，Zygote 才做好了运行新应用程序的准备。

5.3.4 运行新 Android 应用程序

请看代码 5-7，在 SystemServer 运行后，程序会进入一个循环，处理来自所绑定的套接字的请求。若 ZYGOTE_FORK_MODE 为 false，程序就会调用 runSelectLoopMode() 方法，直到 zygote 进程终止，该方法才会返回。

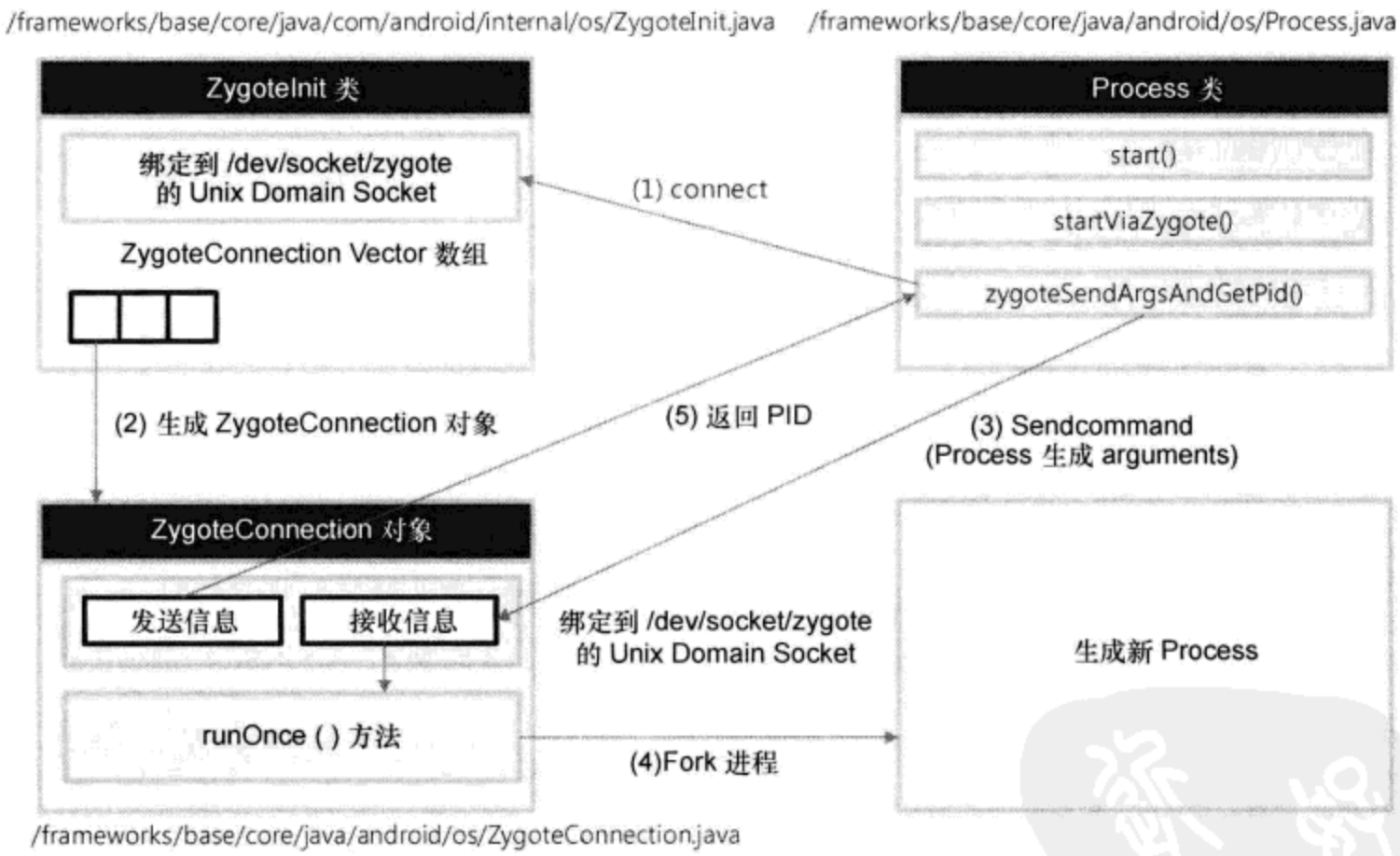


图 5-12 | ZygoteInit 类运行新应用程序的过程

图 5-12 描述的是 ZygoteInit 类运行新应用程序的过程，接下来，通过以下代码，详细分析该过程。

```
private static void runSelectLoopMode() throws MethodAndArgsCaller {
    …
    fds.add(sServerSocket.getFileDescriptor());          ←❶
    while (true) {
        …
        index = selectReadable(fdArray);                 ←❷
        …
        if (index < 0) {
            throw new RuntimeException("Error in select()");
        } else if (index == 0) {                         ←❸
```

更多更新编程资源尽在 http://www.pin5i.com/


```
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
                done = peers.get(index).runOnce();          ❹
                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
    }
```

代码 5-18 | ZygoteInit.java[1]-runSelectLoopMode()

代码 5-18 列出了 runSelectLoopMode()方法的主要部分，runSelectLoopMode()方法采用典型的异步处理方式。

❶ 在 ZygoteInit::main()方法的前半部分中，将套接字与 dev/socket/zygote 绑定在一起。本行首先将套接字的描述符添加到描述符数组中，保存在数组的第 0 个 Index 中。程序将使用该描述符处理来自外部的连接请求。

❷ selectReadable()是一个被注册为 JNI 本地方法的本地函数。该方法用来监视参数传递过来的文件描述符数组，若描述符目录中存在相关事件，则返回其在数组中的索引（Index）。

❸ 该部分代码用来处理 Index 为 0 的套接字描述符中发生的输入输出事件。为了处理传递给 dev/socket/zygote 套接字的连接请求，程序先创建出 ZygoteConnection 类的对象。在 ZygoteConnection 类的构造方法中创建输入输出流，而后生成 Credentials，检查请求连接一方的访问权限[2]。为了处理 ZygoteConnection 对象的输入输出事件，将套接字描述符添加到套接字描述符数组 fds 中。被添加的套接字描述符的输入输出事件在下一个循环中由第❷行的 selectReadable()方法进行检测。

❹ 此行代码用于处理新连接的输入输出套接字，并生成新的 Android 应用程序。代码 5-19 是 runOnce()方法的具体代码。

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
    …
    try {
        args = readArgumentList();                  ❶
        …
```

1 frameworks/base/core/java/com/android/internal/os

2 通过 getsockopt（fd，SOL_SOCKET，SO_PEERCRED，&creds，&szCreds）获取 struct ucred creds。（请参看 frameworks/base/core/jni/android_net_LocalSocketImpl.cpp 文件的 socket_get_peer_credentials()函数）

更多更新编程资源尽在 http://www.pin5i.com/

```
    }
    …

    int pid;

    try {
        parsedArgs = new Arguments(args);                    ←❷
        applyUidSecurityPolicy(parsedArgs, peer);
        applyDebuggerSecurityPolicy(parsedArgs);
        applyRlimitSecurityPolicy(parsedArgs, peer);
        applyCapabilitiesSecurityPolicy(parsedArgs, peer);

        int[][] rlimits = null;

        if (parsedArgs.rlimits != null) {
            rlimits = parsedArgs.rlimits.toArray(intArray2d);
        }
        pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,    ←❸
                ➥ parsedArgs.gids, parsedArgs.debugFlags, rlimits);
    }
    …
    if (pid == 0) {
        handleChildProc(parsedArgs, descriptors, newStderr);          ←❹
    } else { /* pid != 0 */
        return handleParentProc(pid, descriptors, parsedArgs);        ←❺
    }

    if (done) {                                                       ←❻
        peers.remove(index);
        fds.remove(index);
    }
}
```

代码 5-19 | ZygoteConnection.java-runOnce()方法

❶ 读取请求信息，请求信息包含创建新进程的参数选项，以“line count + 字符串数组”形式存在。若想运行 SystemServer，请参考代码 5-14 中的硬编码选项。

❷ 分析请求信息中的字符串数组，为运行进程设置好各个选项，具体包括设置应用程序的 gid、uid，调试标记处理，设置 rlimit，以及检查运行权限等。

❸ 创建新进程。Zygote.forkAndSpecialize()方法接收上面分析好的参数，调用 Zygote 类的本地方法 forkAndSpecialize()。

然后调用本地函数 fork()，创建新进程，并根据新创建的进程传递的选项，设置 uid、gid、rlimit 等。

❹ 该函数用来加载新进程所需要的类，并调用类的 main()方法，启动新的应用程序。直到运行的 Android 应用程序终止，该函数才会返回。

更多更新编程资源尽在 http://www.pin5i.com/

❺ Zygote 返回新进程创建是否成功，若成功，则返回进程的 pid。最后，请求完成后，断开连接，关闭套接字。

❻ 从套接字描述符数组中删除套接字描述符，防止重复处理。Zygote 重新返回到循环中，等待并处理新请求。

以上是 ZygoteInit 类加载新应用程序类并调用 main()方法执行的整个过程，类似于在 Linux 本地环境下调用 dlopen()方法使用动态库。新运行的应用程序由 ZygoteInit 类动态加载，共用装载到父进程生成的虚拟机中的代码。并且，共用应用程序 Framework 中的类以及资源的链接信息，这大大加快了应用程序创建与启动的速度。

TIP 运行 Zygote 时使用的参数类型

持版本，其桌面版本提供 3 年支持，服务器版本则提供长达 5 年的支持。

参数名	功能
--setuid	指定 user id
--setgid	指定 group id
--setgroups	指定额外 gid
--capabilities	接收以“,”分割的数字对，第一个表示权限，第二个表示 effective
--rlimit=r, c, m	控制新进程的 setrlimit 效果。 r、c、m 分别表示 resource、current、max value
--peer-wait	该参数表示不关闭命令套接字，保持开启状态
--classpath	指定 Java 类路径
--runtime-init	其余参数被传递给 com.android.internal.os.RuntimeInit 类处理

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 6 章

Android 服务概要

如果你曾开发过 Android 应用程序，那么你一定不会对“服务”（Service）[1] 这个术语感到陌生。在 Android 系统中，“服务”没有 UI 界面，它作为一个后台进程，周期性地执行某些特定任务。在开发一些用于处理无须用户干预的数据的后台应用程序时，经常会使用这类 Android 应用程序服务，例如通过网络对数据进行周期性的监控，或接受 RSS 反馈分析 XML 数据。在开发 Android 程序时，引入适当的服务，从而开发出更灵活、更健康、更敏捷的优秀程序来。

在 Android Framework 或应用程序开发中所需要的主要 API 都是以系统服务的形式存在的，例如获取终端的当前位置，读取感应器的数值，拨打电话等，这些基本的功能在 Framework 中都是以服务的形式存在的。

6.1 示例程序：理解 Android 服务的运行

下面通过一个简单的示例，了解一下 Android 服务的运行方式。在这里我们要使用的示例程序为 ApiDemo 的 Alarm Service。（关于 ApiDemo 的相关介绍，请参考 Tip 部分）

TIP ApiDemos 示例程序

下载完 Android SDK 后，在 Samples 文件夹中，有多种示例代码，认真阅读这些示例代码对编写实际应用程序非常有帮助。ApiDemos 示例程序由简单的应用程序包组成，通过该示例程序，能了解多种 Android API 的使用方法。

关于 ApiDemos 示例程序更详细的说明，请参考 Android 开发者网站中 ApiDemos 说明部分（http://developer.android.com/resources/samples/ApiDemos/index.html）。

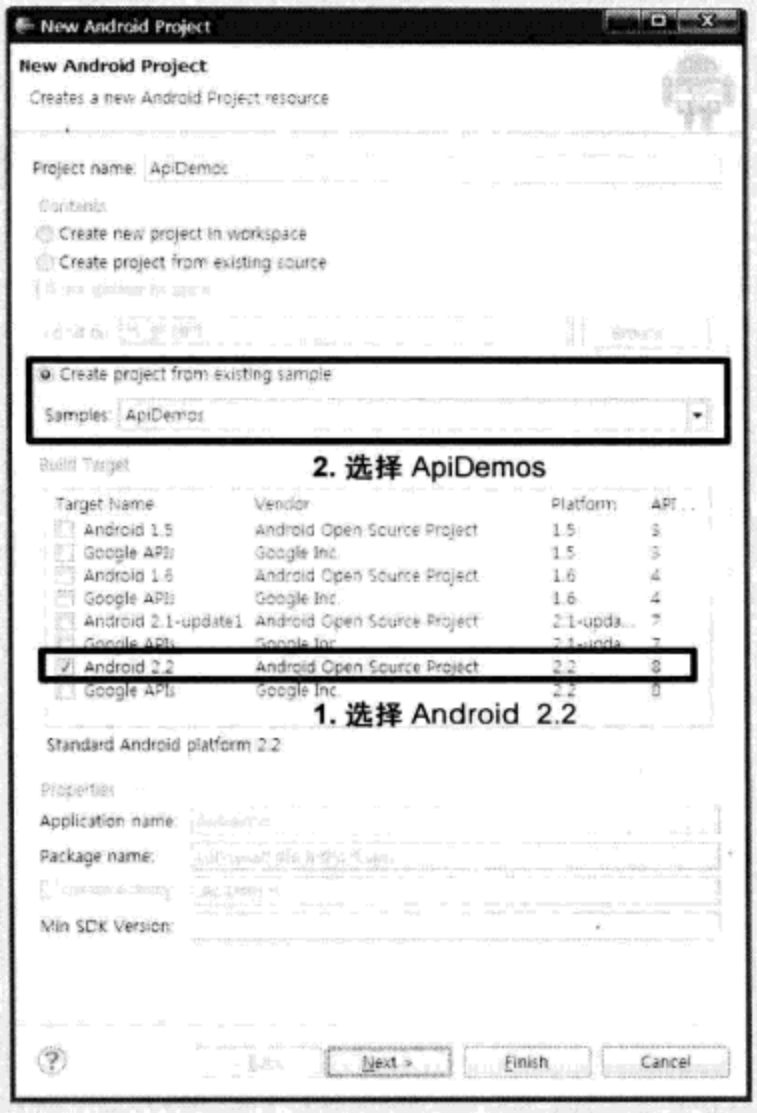
1 此处的服务是指 Android 服务之一“应用程序服务”，它继承并实现了 Service 类。有关应用程序服务的更多内容，请参考本章后半部分的相关内容。

更多更新编程资源尽在 http://www.pin5i.com/

在本书后面内容的讲解中，会多次引用 ApiDemos 示例程序。因此有必要学习 ApiDemos 编译，以及在模拟器中运行的相关方法。

编译 ApiDemos

运行 Eclipse，新建 Android 工程。在新建 Android 工程对话框中，选择 Build Target 为“Android2.2”，再点选“Create project from existing sample”，选中 ApiDemos，如下图所示。而后单击 Next 按钮编译后，ApiDemos 被安装至模拟器中。



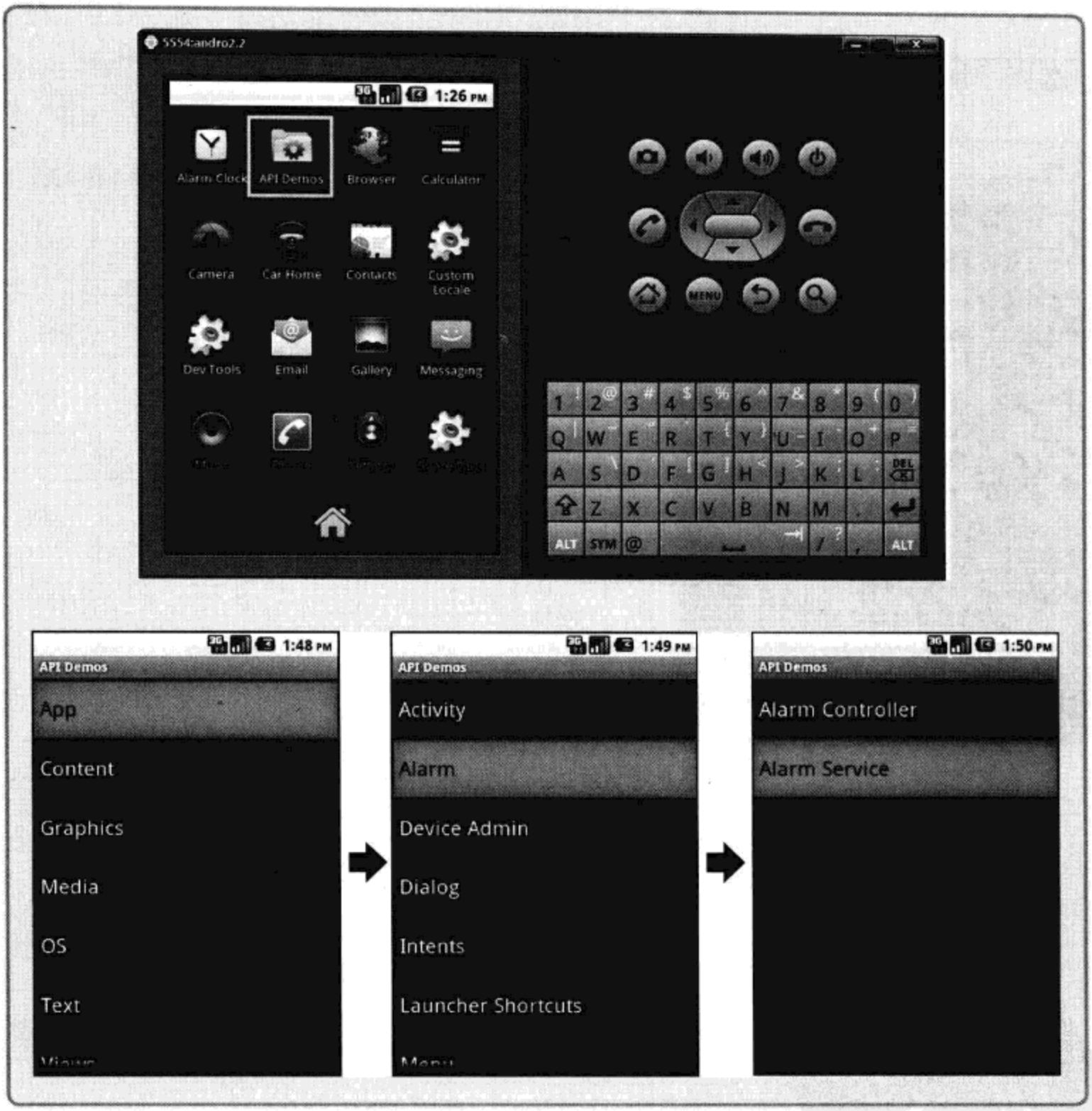
在模拟器中运行 ApiDemos

编译完 ApiDemos 后，在模拟器中运行它，即可在桌面上看到 ApiDemos 的图标，如图所示。

在 ApiDemos 程序中，依次选择 App–Alarm–Alarm Service，即可运行 Alarm Service，如图所示。

在 ApiDemos 示例程序代码中，有多个说明 Android API 使用方法的示例，从事 Android 应用程序开发的人必须认真阅读这些示例代码，这些示例代码对编写实际应用程序非常有帮助。

更多更新编程资源尽在 http://www.pin5i.com/



在 Alarm Service 的启动画面中仅有启动与停止两个按钮，画面相当简单，如图 6-1 所示。该示例程序表面上看似简单，但在其内部应用了 Android 系统服务与应用程序服务两种服务。

该程序主要由主 Activity、两个系统服务（Alarm Service，Notification Service）与应用程序服务 AlarmService_Service 组成。图 6-1 详细地描述了程序的整个运行过程，整理如下。

（1） 在主 Activity 中，点按 Start Alarm Service，启动 Alarm Service 系统服务，该系统服务每隔 30 秒请求执行一次应用程序服务 AlarmService_Service。

更多更新编程资源尽在 http://www.pin5i.com/

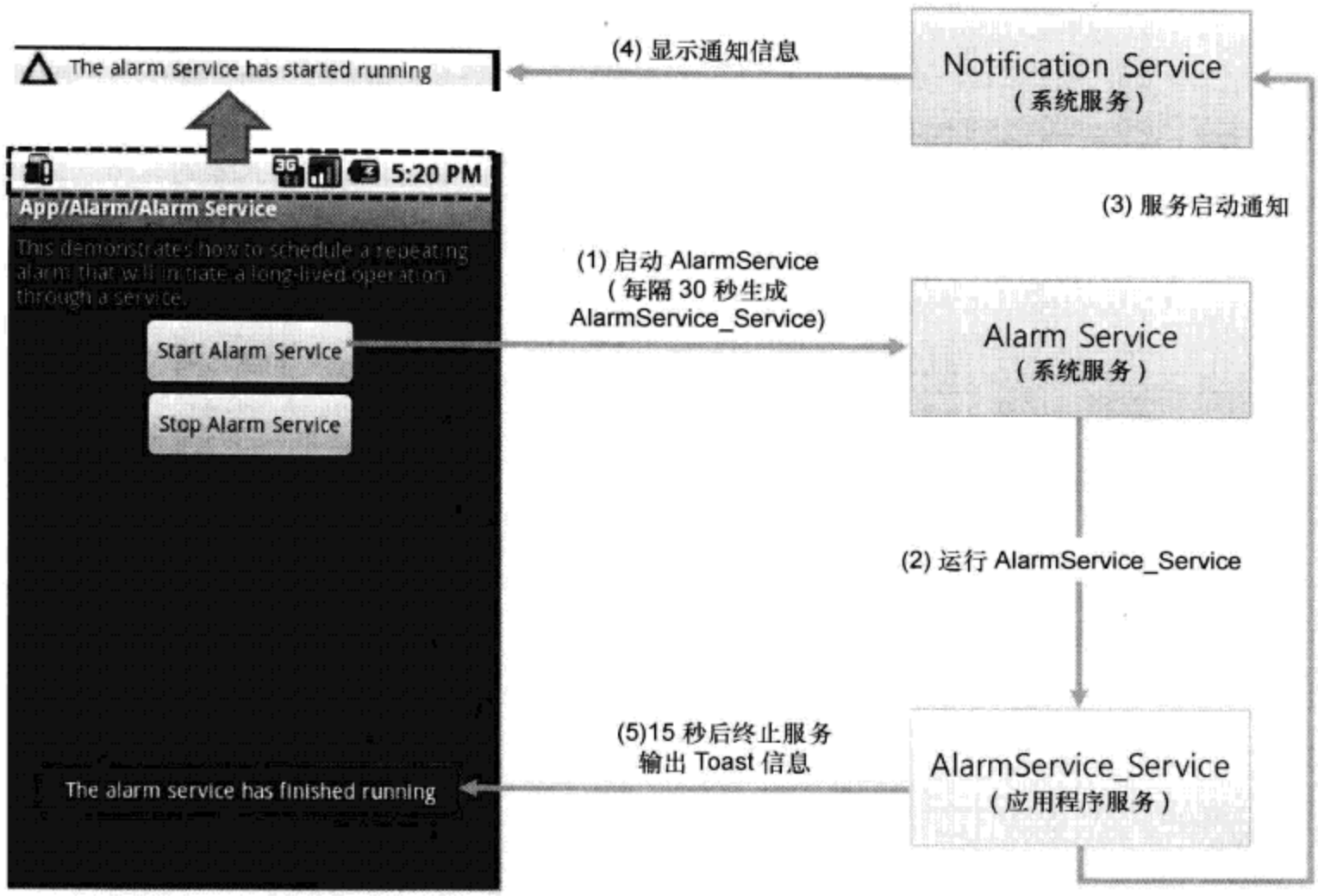


图 6-1 | 运行 API Demos 的 AlarmService

（2） 根据（1）中的用户请求，Alarm Service 每隔 30 秒运行一次应用程序服务 AlarmService_ Service。

（3） 当 AlarmService_Service 运行时，程序就会请求系统服务 Notification Service 输出字符串，告知用户 AlarmService_Service 服务已经启动。

（4） Notification Service 从 AlarmService_Service 接收字符串，并显示在画面顶部的状态栏中。

（5） AlarmService_Service 启动 15 秒后终止，同时输出 Toast 信息，告知主 Activity 服务已经终止。

以上是 ApiDemos 的 AlarmService 示例程序的运行过程，通过它，我们了解了在 Android 应用程序中如何使用系统服务与应用程序服务。虽然 AlarmService 不像 Activity 一样有 UI，仅作为后台程序运行，但是我们仍然能弄清楚程序各个部分的功能。

6.2 Android 服务的种类

如图 6-2 所示，在 Android 系统中服务主要分为两类，一类是系统服务，它们是由

更多更新编程资源尽在 http://www.pin5i.com/

Framework 提供的；另一类是应用程序服务，它们是应用程序开发者继承并实现 Service 类[1]后开发出来的。

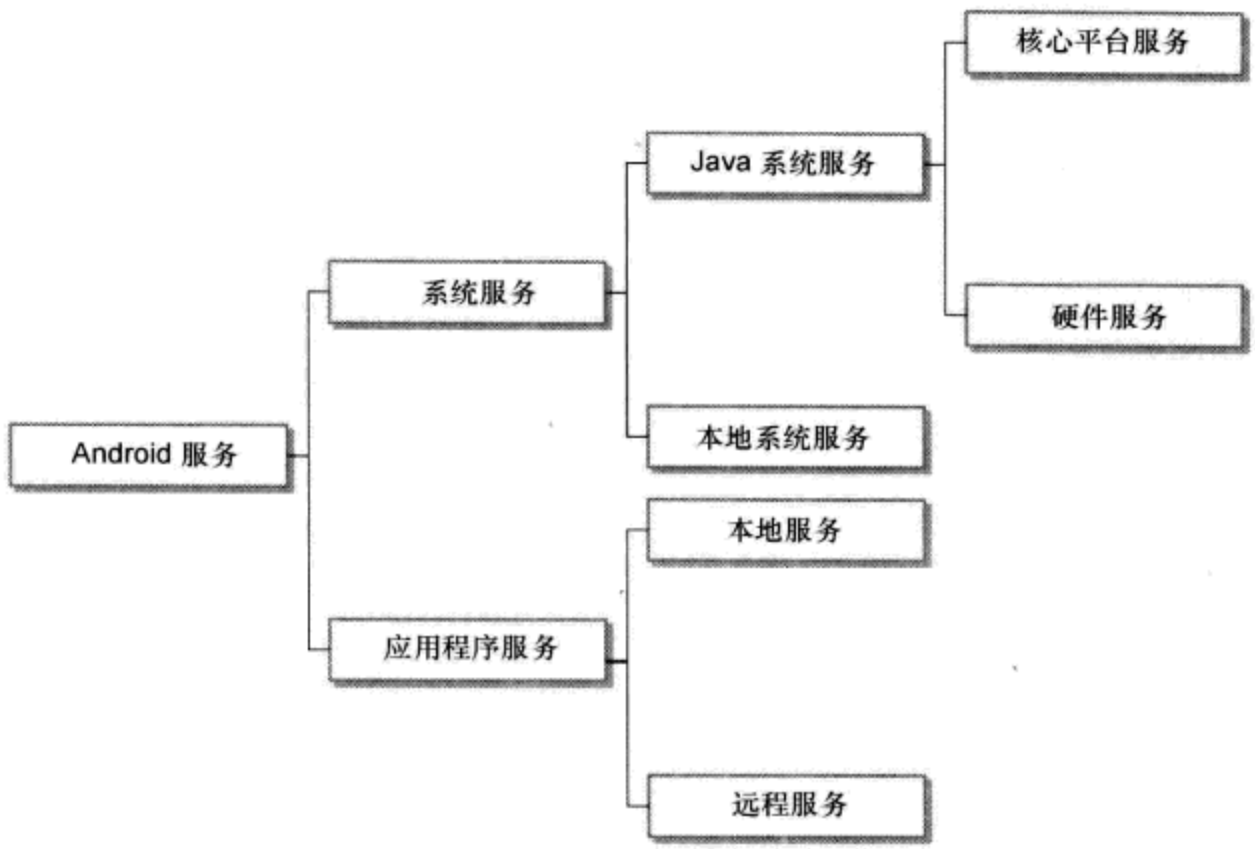


图 6-2 | Android 服务分类

从图 6-2 中，我们了解到 Android 系统中的各种服务，至于各类服务内部是如何运行的，在后面会作简略的讲解。

TIP TIP Android 服务的分类

关于 Android 系统服务的分类，请参考 2008 Google IO Anatomy and Physiology of an Android（http://sites.google.com/site/io/anatomy--physiology-of-an-android）发表资料中使用的分类。

另外，有关 Android 应用程序服务的术语，请参考如下 android framework 谷歌 groups。

http://groups.google.com/group/android-framework/browse_thread/thread/d54dd5864defd86c/cc19bfae7c792af0?lnk=gst&q=android+services#cc19bfae7c792af0

在文章中 Android Framework 的开发者 Dianne Hackborn 在提及应用程序服务时使用的术语是“基于 SDK 的应用程序服务”（SDK-based application service）。为了使用上的方便，本书把“基于 SDK 的应用程序服务”简称为“应用程序服务”。

1　frameworks/base/core/java/android/app/Service.java

更多更新编程资源尽在 http://www.pin5i.com/

6.3 Android 应用程序服务

应用程序服务是后台服务程序，它是继承了 Android SDK 的 Service 类的类实例，没有 UI 界面，定期执行某些任务。这些服务是 Android 的一种应用程序组件，例如 Activity 或 BrocastReceiver。

应用程序开发者可以通过以下两种方法来使用这些服务。

- 服务的启动与终止：启动或终止执行特定任务的后台服务。
- 通过服务绑定，实现远程控制：如同 Activity，服务的客户端在绑定到服务之后，通过服务提供的界面，即可利用服务的各种功能，实现对服务的远程控制。

为了帮助各位理解以上两种使用服务的方法，我们以 Media Player 为例形象地说明一下。假如我们想听某个音乐，只需要直接打开 Media Player 的用户界面选取指定的音乐播放即可。如果想边听音乐边做其他事情，播放器的音乐播放功能必须被做成服务的形式。当在 Media Player 中播放音乐时，后台的音乐服务就会启动，音乐就会继续播放下去。

在听音乐时，你可能需要暂停、后退、终止、重放音乐，即你需要控制音乐播放服务。为此，需要先连接到正在运行中的音乐服务上，而后利用相关功能，实现对音乐播、停、退等控制。为了实现服务的远程控制，首先要连接到服务上，这一过程被称为服务的绑定。

当服务被绑定后，客户端就可以自由地调用服务的各个方法，如 RPC，从而实现对服务的控制。

在创建应用程序服务（Application Service）时，需要根据具体情况，调用 startService()、bindService()等 API。若只是想在后台启动一个执行特定任务的服务，则调用 startService()方法；但如果想绑定服务，并通过服务提供的界面实现对服务的控制，就应调用 bindService()来创建服务。

在 Android 中，服务是应用程序组件，具有一定的生命周期，如图 6-3 所示。由 startService()与 bindService()创建的两种服务，它们的生命周期略微不同。由 startService()创建的服务目的只是服务的启动与终止，而 bindService()创建的服务是为了服务的远程控制。

不论是由 startService()，还是由 bindService()启动的服务，它们都会调用 onCreate()与 onDestroy()两个回调方法。onCreate()方法在服务初次创建时被调用，一般用于对创建的服务进行初始化。onDestroy()方法在服务终止前被调用，用于释放服务占用的所

更多更新编程资源尽在 http://www.pin5i.com/

有资源。

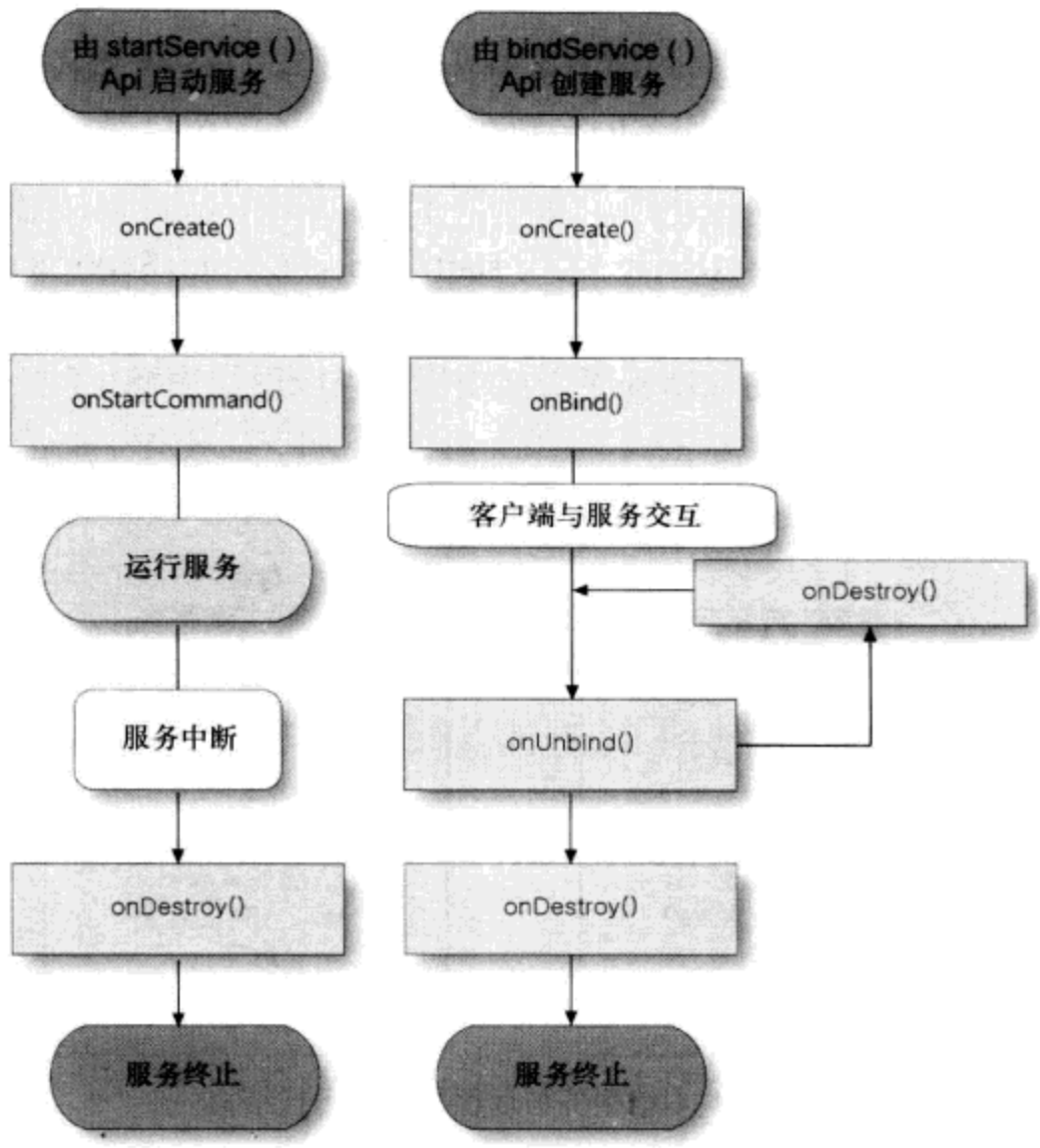


图 6-3 | Android 应用程序服务的生命周期

如图 6-3（左）所示，由 startService()启动的服务调用完 onCreate()方法后，将继续调用 onStartCommand（Intent,int,int）[1] 方法。startService()的第一个参数 Intent 将被传递给 onStartCommand()方法，作为它的第一个参数，在 Intent 中主要包含与将要运行的服务相关的信息。若从 Intent 中传递过来参数，onStartCommand()会进行处理，或运行线程处理后台任务。

在由 bindService()启动的服务中，当客户端企图绑定服务时，onBind()方法即会被调用（若服务尚未生成，会先调用 onCreate()方法）。服务的 onBind()回调会为将进行绑定操作的客户提供对象，以便将其与相应的服务绑定在一起。当绑定完成后，客户即可经由该对象对绑定的服务进行远程控制。

1　从 Android2.0 开始，与服务相关的 API 及生命周期都发生了变化。特别需要注意的是原 onStart()被弃用，取而代之的是 onStartCommand()函数，它是一个回调函数，带有一些返回值，便于系统对服务进行更精确地管理。更多详细内容，请参考 http://android-developers.blogspot.com/2010/02/service-api-changes-starting- with.htm。

更多更新编程资源尽在 http://www.pin5i.com/

应用程序服务（Application Service）的分类

如图 6-2 所示，Android 应用程序服务分为本地服务（Local Service）与远程服务（Remote Service）两类。这两类服务的区别在于创建服务的客户端（通常为 Activity）与所创建的服务是否运行在同一进程中，如图所示。

如图 6-4（左）所示，一个 Activity 调用 startService()或 bindService()，创建出服务，若该服务与创建服务的 Activity 运行在同一进程中，则把这种服务称为本地服务。并且，本地服务只能在创建该服务的应用程序内部使用，当应用程序终止时，本地服务也一同终止。

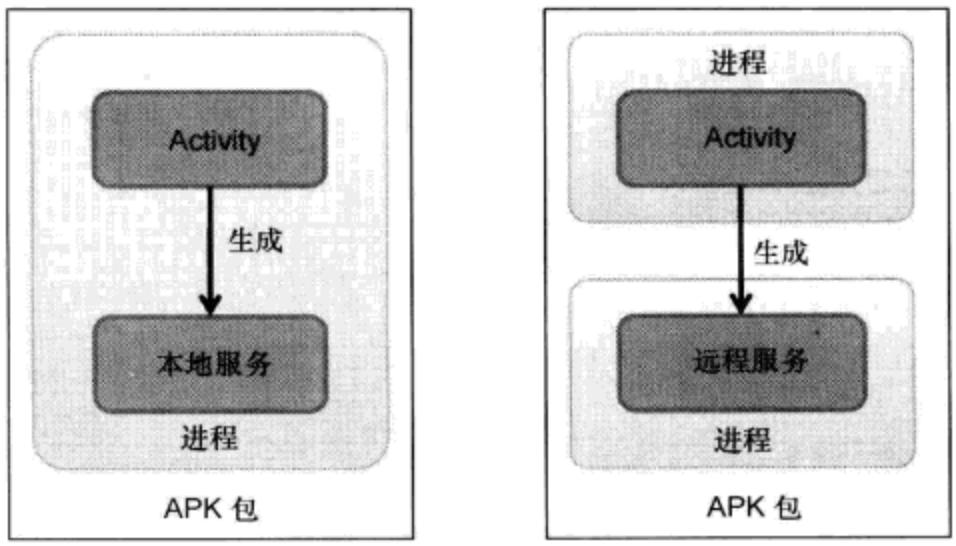


图 6-4 | 本地服务与远程服务的运行结构

与本地服务不同，远程服务并不与创建者运行在同一进程中，它运行在单独的进程中，所以当主应用程序终止时，远程服务仍然会继续运行。在编写应用程序服务时，若要求在应用程序的主要部分（如 Activity）终止时，服务仍要继续运行，执行某些特定任务，此时就该考虑使用远程服务。由于远程服务在程序退出时仍在运行，会继续消耗系统资源（如电池等），所以在设计远程服务时一定要认真考虑、慎重处理。

本地服务与远程服务最大的不同是，绑定服务时所使用的方法不同。前面在讲解绑定时，简要介绍过绑定是服务的客户程序为实现服务的远程控制而进行相互连接的一个过程。

在本地服务中，服务与使用服务的客户端程序运行在同一进程中，本地服务的绑定实际是指客户端程序获取了待绑定服务的一个引用。当绑定完成后，客户端即获取了服务的一个引用，通过该引用，客户端即可调用该服务提供的各种方法。

就远程服务而言，Activity 与远程服务运行在不同的进程中，Activity 若想控制远

更多更新编程资源尽在 http://www.pin5i.com/

程服务，必须使用 IPC 机制，在下一章中将介绍 Binder IPC 机制。远程服务绑定是指设置相关的连接设置，以便运行 Binder IPC。

在 Binder IPC 通信中，服务与 Activity 在交换数据时，需要经历 Marshalling/Unmarshalling 这一过程，为此，就要使用 AIDL（Android Interface Definition Language，Android 接口定义语言）。AIDL 是一种接口定义（IDL）语言，用于约束两个进程间的通信规则，供编译器生成代码，用来实现 Android 设备上两个进程间的通信（IPC）。有关 AIDL 更多的内容，请参考 Android 开发者网站“Designing a Remote Interface Using AIDL”（http://developer.android.com/guide/ developing/tools/aidl.html）。

对于本地服务与远程服务绑定过程的不同，可以通过分析 ApiDemos 的 Local Service Binding 与 Remote Service Binding 两个示例代码来进一步理解。在 ApiDemos 中除了这两个之外，还包含其他许多有关服务的示例代码，如图 6-5 所示。

图 6-5 | ApiDemos 应用程序中与服务有关的示例程序列表

这些示例程序能帮助我们充分理解所有服务的运行原理，请感兴趣的读者仔细阅读。

本地服务（Local Service）

在 ApiDemos 示例中包含一个名称为 Local Service Binding 的示例，它是本地服务的应用演示。下面我们将通过这个示例，了解本地服务是如何工作的。

如图 6-5 所示，选择并运行 Local Service Binding，即进入程序运行界面，如图 6-6 所示。Local Service Binding 大致由 LocalService.java 与 LocalServiceActivities.java 两个文件组成，LocalService.java 是本地服务，而 LocalServiceActivities.java 是使用服务的 Activity。在 LocalServiceActivities.java 文件中，以内部类（inner class）的形式，定

更多更新编程资源尽在 http://www.pin5i.com/

义了名称为 Controller 与 Binding 两个 Activity，其中 Binding 代表 Local Service Binding，如图 6-6（左）所示。

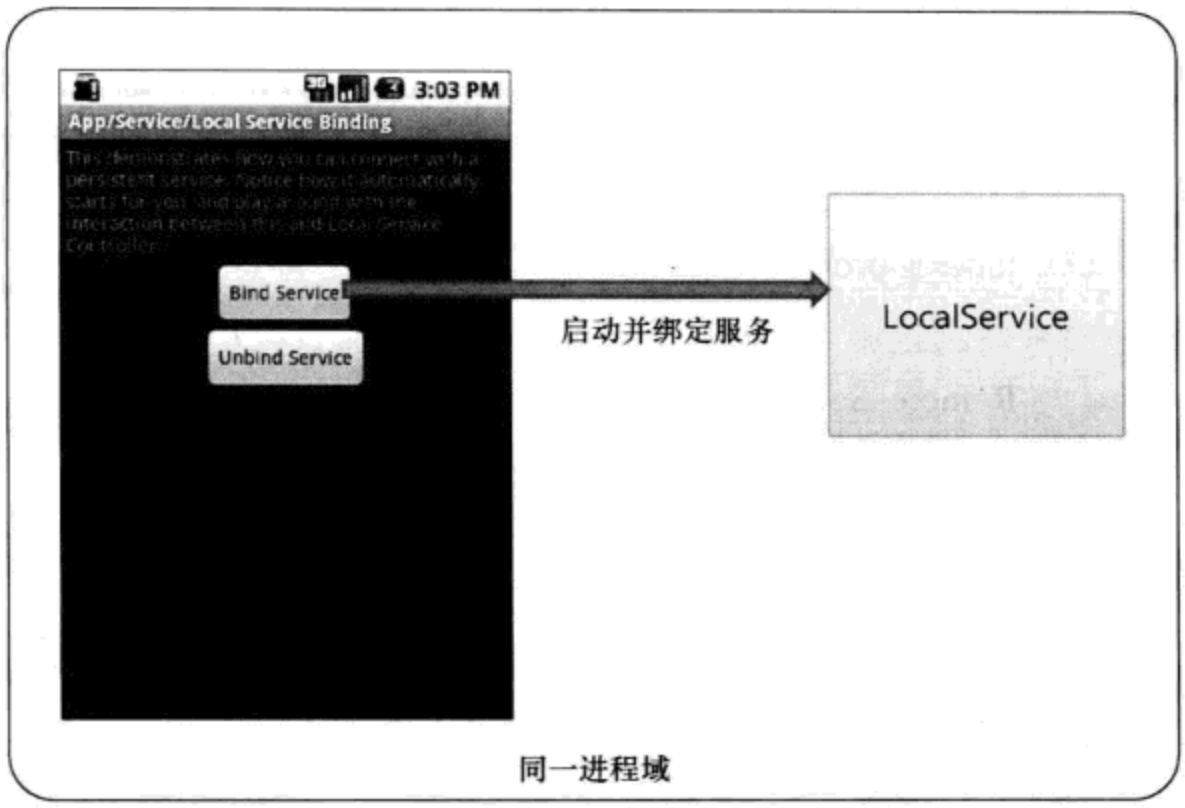


图 6-6 | 运行并绑定 Local Service Binding 程序

在 Local Service Binding 程序界面中，点击 Bind Service 按钮，即可启动并绑定 LocalService 服务。服务绑定后，Binding 就能调用 LocalService 服务提供的多种方法了。

关于 Activity 绑定本地服务的方法，对照相应代码，仔细进行分析。Local Service Binding 程序由本地服务与使用该服务的 Activity 构成，分析代码的运行方式时，应该考虑 Activity 与服务间的相互作用，交替查看两部分的代码。图 6-7 描绘了 Activity 是如何绑定本地服务，并与其进行交互的过程。

（1） 代码 6-1 是处理点击 Bind Service 按钮事件的代码。当按下 Bind Service 按钮时，doBindService()方法即会被调用，该方法会调用 bindService()，尝试绑定 LocalService。

bindService（Intent，ServiceConnection,int）API 的第一个参数是运行 LocalService 的 Intent，第二个参数是服务客户端用于处理绑定连接的对象，第三个参数 Context.BIND_AUTO_CREATE 是一个自动生成本地服务的标记，当待绑定的服务不存在时使用。若 LocalService 尚未运行，在实施绑定前，Android Framework 会先生成 LocalService。

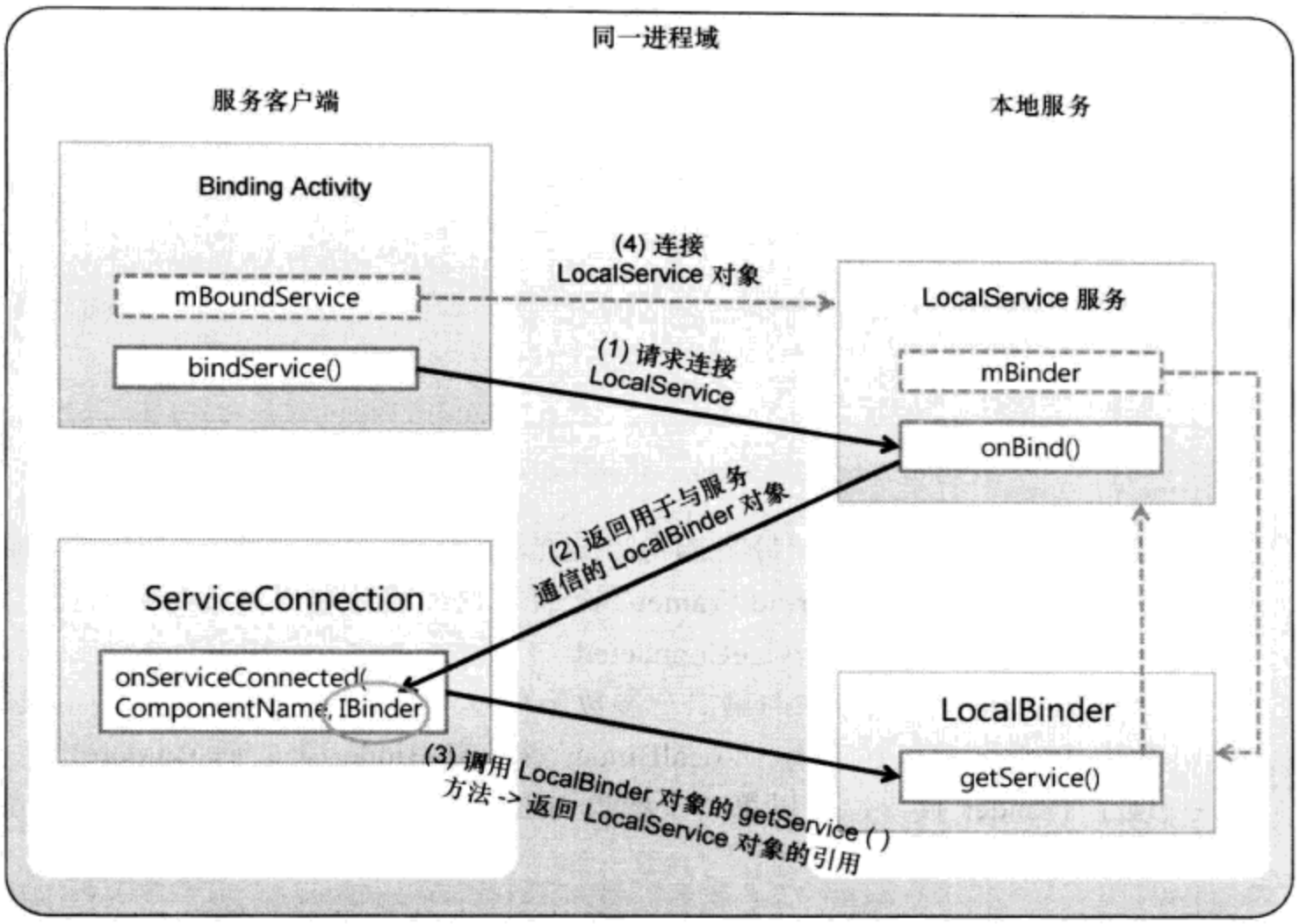


图 6-7 | 绑定本地服务

```
private OnClickListener mBindListener = new OnClickListener() {
   public void onClick(View v) {
      doBindService();
   }
};

void doBindService() {
   bindService(new Intent(Binding.this, LocalService.class),
               ➥ mConnection, Context.BIND_AUTO_CREATE);
   mIsBound = true;
}
```

代码 6-1 | LocalServiceActivities.java-Binding 中 Bind Service 按钮事件的处理代码

（2） 在待绑定的服务生成之后，Android 会调用服务的 onBind()回调方法，用来处理服务绑定。调用 onBind()方法将返回 LocalBinder 对象，继承自 Binder 类[1]，Activity 使用该对象与 LocalService 连接。

1 frameworks/base/core/java/android/os/Binder.java

更多更新编程资源尽在 http://www.pin5i.com/

```
public IBinder onBind(Intent intent) {
   return mBinder;
}

private final IBinder mBinder = new LocalBinder();

public class LocalBinder extends Binder {
   LocalService getService() {
      return LocalService.this;
   }
}
```

代码 6-2 | LocalService.java-onBind()方法的主要部分

（3） 若以上两步进展顺利，处理服务绑定的对象（在本示例中指 LocalBinder 对象）创建成功，Android Framework 就会调用服务客户端（在本示例中指 Bind Activity）的 onServiceConnected（ComponentName,IBinder）方法，如代码 6-3 所示。该方法的第二个参数保存着步骤（2）中 onBinder()生成的 LocalBinder 对象的引用。LocalBinder 继承自 Binder 类，而 Binder 类又实现了 IBinder 接口，所以第二个参数 IBinder service 可以接收 LocalBinder 对象。

而后 Binding Activity 调用 LocalBinder 对象的 getService()方法，获取 LocalService 对象的引用。

```
private ServiceConnection mConnection = new ServiceConnection() {

   public void onServiceConnected(ComponentName className, IBinder service) {
      mBoundService = ((LocalService.LocalBinder)service).getService();
   }

   public void onServiceDisconnected(ComponentName className) {
      mBoundService = null;
   }

};
```

代码 6-3 | LocalServiceActivities.java-处理绑定连接的代码

（4） 保存 LocalService 对象的引用到 Activity 的 mBoundService 成员变量中，本地服务绑定完成。

在服务绑定之后，Activity 即可通过保存在 mBoundService 成员变量中的 LocalService 对象的引用，访问服务的成员变量，以及调用服务的相关方法。

远程服务（Remote Service）

下面以 ApiDemos 的 Remote Service Binding 为例，向读者讲解远程服务的运行原理及绑定方法。Remote Service Binding 示例程序运行画面如图 6-8 所示。如图，Activity 运行在进程 A 中，点按其中的 Bind Service 按钮，RemoteService 服务就会在另一个独立的进程 B 中启动并运行，运行界面与本地服务类似。

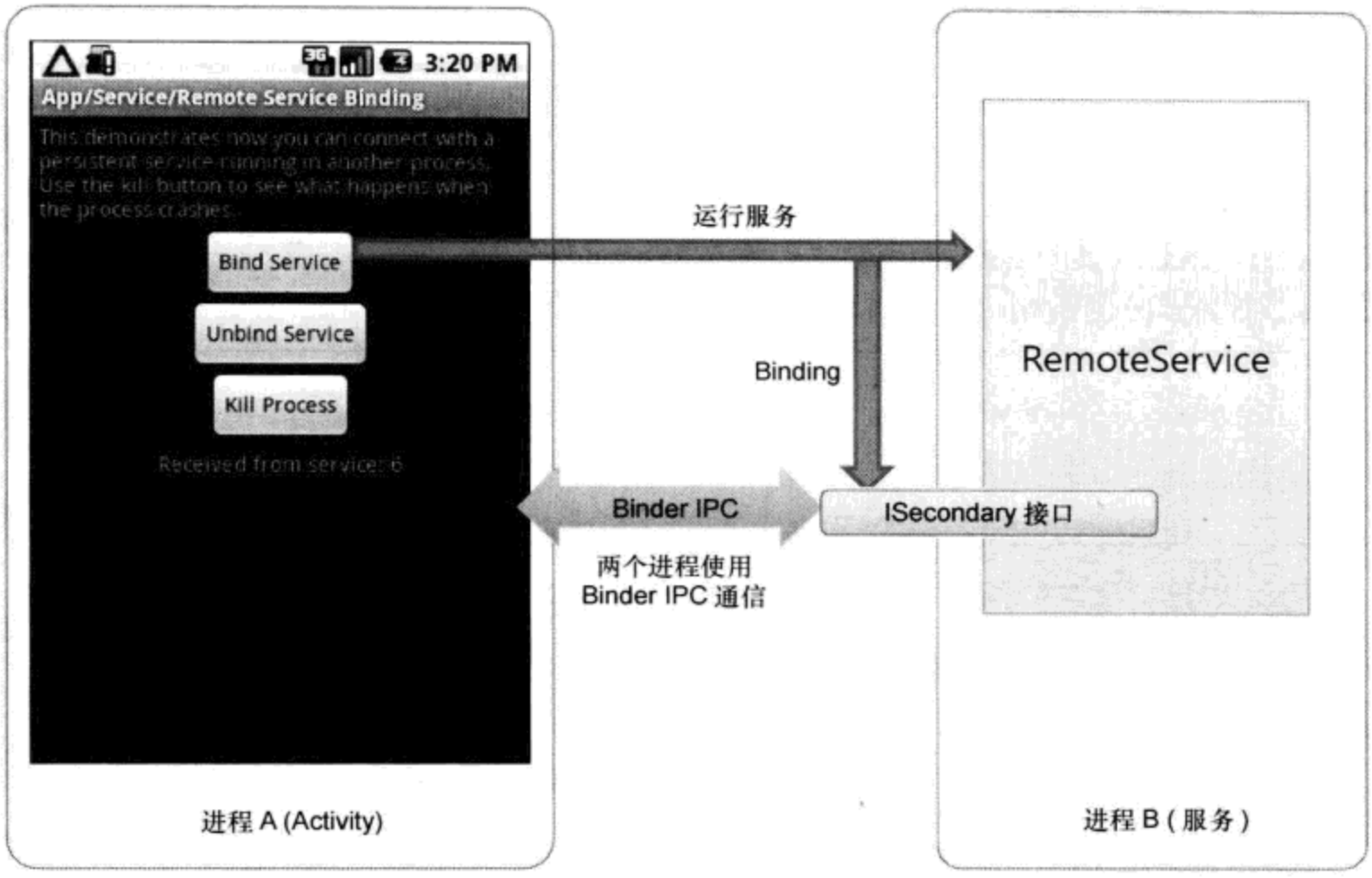


图 6-8 | Remote Service Binding 程序的运行与绑定

在分析 Remote Service Binding 程序之前，先分析其源代码的构成。与前面讲解的 Local Service Binding 不同，除了 Activity 与服务文件外，还有一个名称为 ISecondary.aidl 的 AIDL 文件，以及由该文件自动生成的 ISecondary.java 文件。（在 Remote Service Binding 源文件中，除了 ISecondary.java 外，还有 IRemoteService.aidl、IRemoteServiceCallback.aidl 文件。但是为了方便讲解相关知识，我们假定只包含一个 ISecondary.aidl 文件。若想查看更多，请直接参考示例程序的源代码。）

表 6-1 Remote Service Binding 程序的源文件

源文件名	功能
RemoteService.java*	除 RemoteService 服务外，其中还包含两个使用 RemoteService 的 Activity，分别为 Controller 与 Binding，它们是内部类
ISecondary.aidl	Activity 与服务通信接口定义

更多更新编程资源尽在 http://www.pin5i.com/

续表

源文件名	功能
ISecondary.java	Android 根据 ISecondary.aidl 自动生成的文件。执行 Marshalling/Unmarshallling 操作，以便 Activity 与服务通过 ISecondary 接口通信

*ApiDemos 的 Remote Service 示例与其他 Android 示例不同，服务与使用服务的 Activity 都在 RemoteService.java 文件中。

如前所述，Android SDK 会根据 AIDL 文件中定义的接口自动生成绑定远程服务所需要的代码。在 Remote Service Binding 示例程序中，使用了 ISecondary.aidl，如代码 6-4，在其中仅定义了一个 ISecondary 接口，并且接口中有一个 getPid()方法，用于返回进程的 ID。使用 RemoteService 的客户程序通过 ISecondary 接口，调用 RemoteService 提供的 getPid()函数（虽然在 ISecondary.aidl 文件中，还包含 basicTypes()方法，但为了方便说明，在此省略）。

```
interface ISecondary {
    int getPid();
}
```

代码 6-4 | ISecondary.aidl

当然在上面的 aidl 文件中，仅仅定义了接口，并没有具体的实现代码。若想通过 aidl 中定义的接口实现 Binder IPC，必须编写复杂的代码。不过，幸运的是这些代码并不需要我们编写，Android SDK 会根据 aidl 文件自动生成所需要的代码。代码 6-5 即是 Android SDK 根据 ISecondary.aidl 文件自动生成的 ISecondary.java 代码的主要部分。这些代码用来设定基于 ISecondary 接口的 Binder IPC 连接，以便客户端与远程服务通信。换言之，RemoteServiceBinding 通过这些代码调用 RemoteService 服务提供的 ISecondary 接口中的 getPid()方法，如同调用相关类中的方法一样。但是 ISecondary.java 只是用来设定连接，getPid()方法的实际代码仍然需要由 RemoteService 开发者直接实现。

```
public interface ISecondary extends android.os.IInterface
{
   public static abstract class Stub extends android.os.Binder
         implements com.example.android.apis.app.ISecondary
   {
      private static final java.lang.String DESCRIPTOR =
                  "com.example.android.apis.app.ISecondary";

      public Stub() {
         this.attachInterface(this, DESCRIPTOR);
      }
```

更多更新编程资源尽在 http://www.pin5i.com/

```
public static com.example.android.apis.app.ISecondary
                ➥ asInterface(android.os.IBinder obj)
{
   if ((obj==null)) {
      return null;
   }

   android.os.IInterface iin =
       ➥ (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);

   if (((iin!=null)&&(iin instanceof com.example.android.apis.app.ISecondary))) {
      return ((com.example.android.apis.app.ISecondary)iin);
   }
         return new com.example.android.apis.app.ISecondary.Stub.Proxy(obj);
}

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

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

      case TRANSACTION_getPid:
      {
         data.enforceInterface(DESCRIPTOR);
         int _result = this.getPid();
         reply.writeNoException();
         reply.writeInt(_result);
         return true;
      }
   }
   return super.onTransact(code, data, reply, ?ags);
}

private static class Proxy implements com.example.android.apis.app.ISecondary
{
   private android.os.IBinder mRemote;

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

   public android.os.IBinder asBinder() {
      return mRemote;
```

更多更新编程资源尽在 http://www.pin5i.com/

```
        }

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

        public int getPid() 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);
                mRemote.transact(Stub.TRANSACTION_getPid, _data, _reply, 0);
                _reply.readException();
                _result = _reply.readInt();
            }
            finally {
                _reply.recycle();
                _data.recycle();
            }

            return _result;
        }
    }

    static final int TRANSACTION_getPid = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
  }
}
```

代码 6-5 | 基于 ISecondary.aidl 自动生成的 ISecondary.java 代码的主要部分

那么，远程服务是如何绑定的呢？在 Remote Service Binding 程序中，RemoteService Binding 是如何绑定到 RemoteService 服务上的，又是如何使用 RemoteService 服务提供的功能（此处为 getPid()方法）的，图 6-9 对这一过程做了详尽的描述。

根据图 6-9 的顺序，并结合程序源码，逐一进行分析。

（1） Binding Activity：请求 RemoteService 连接

代码 6-6 是点击图 6-8 中的 Bind Service 按钮时事件处理代码的主要部分。与本地服务一样，Activity 必须先调用 bindService()API，绑定到远程服务上，才能实现对服务的远程控制。与本地服务类似，bindService()的第一个参数也是 Intent，用来运行远程服务。在 RemoteService 的 manifest 文件中，可以看到 RemoteService 服务处理 com.example.android.apis.app. ISecondary 动作，如代码 6-7 所示。

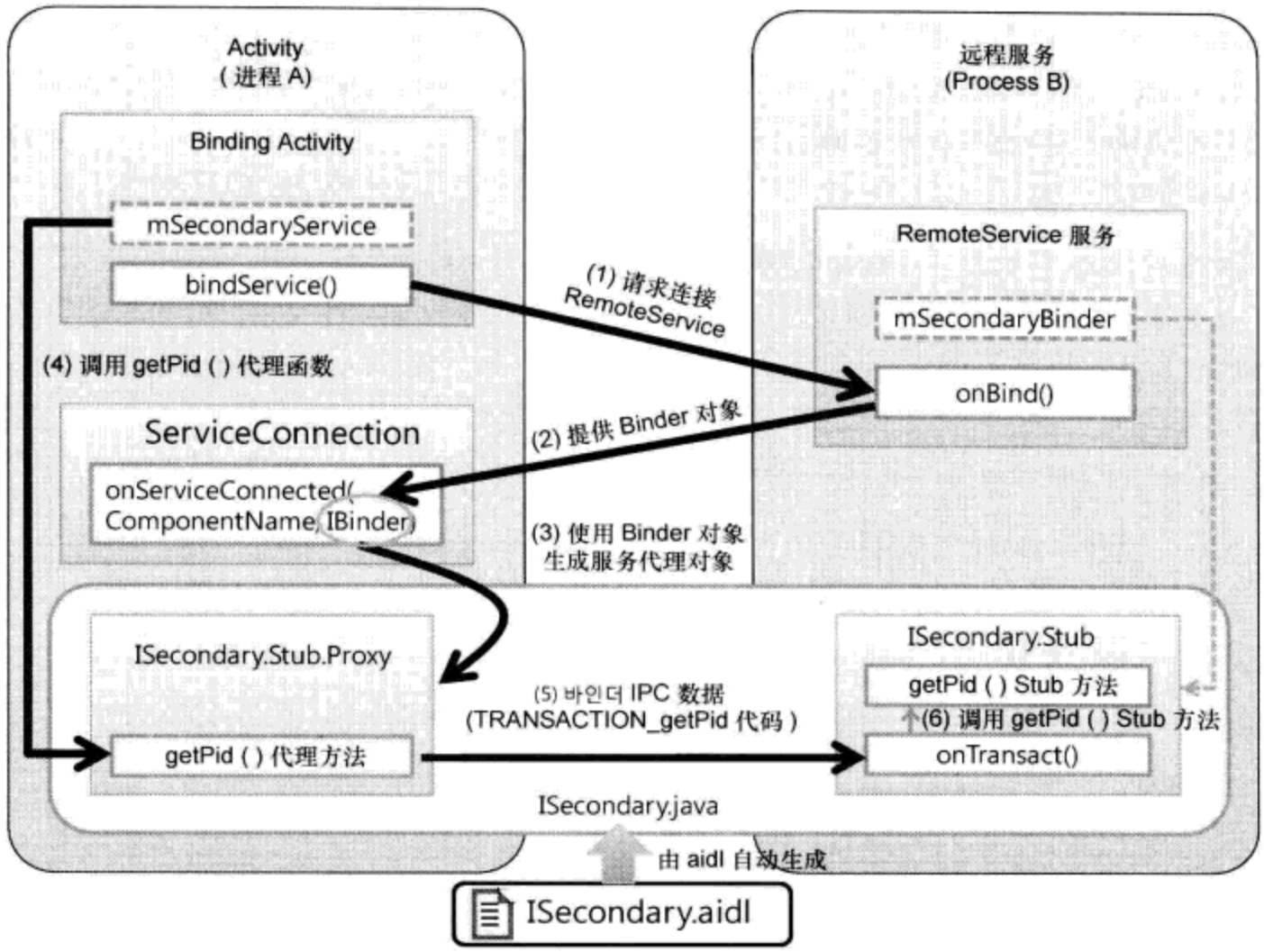


图 6-9 | 绑定远程服务

```
private OnClickListener mBindListener = new OnClickListener() {
   public void onClick(View v) {
      bindService(new Intent(ISecondary.class.getName()),
            ➥ mSecondaryConnection, Context.BIND_AUTO_CREATE);
   }
};
```

代码 6-6 | RemoteService.java-Binding 中处理 Bind Service 按钮事件的主要代码

```
<service android:name=".app.RemoteService" android:process=":remote">
    <intent-filter>
        <action android:name="com.example.android.apis.app.IRemoteService" />
        <action android:name="com.example.android.apis.app.ISecondary" />
        <action android:name="com.example.android.apis.app.REMOTE_SERVICE" />
    </intent-filter>
</service>
```

代码 6-7 | ApiDemos 程序的 manifest 文件（AndroidManifest.xml）中有关 RemoteService 的部分

（2） RemoteService 服务：具体实现服务方法，并提供用于通信的 Binder 对象

更多更新编程资源尽在 http://www.pin5i.com/

服务启动后，会根据服务的生命周期，依次调用 onCreate()与 onBind()方法。如 LocalService 一样，onCreate()方法会以通知消息的形式告知服务已经生成。onBind()方法的主要作用是生成用于处理 Binder IPC 的 Binder 对象（示例中为 mSecondaryBinder），并将其返回给系统。

服务的绑定对象是由 ISecondary.java（代码 6-5）的 ISecondary.Stub 类生成的。此时实现定义在 ISecondary 接口中的 getPid()方法的实际代码。在 getPid()方法的实现中，服务的进程 ID 将被返回。

```
private final ISecondary.Stub mSecondaryBinder = new ISecondary.Stub() {
   public int getPid() {
      return Process.myPid();
   }
};

public IBinder onBind(Intent intent) {
   if (ISecondary.class.getName().equals(intent.getAction())) {
      return mSecondaryBinder;
   }
}
```

代码 6-8 | RemoteService.java-RemoteService 的 onBind()回调方法

（3） Binding Activity：生成执行服务与 Binder IPC 的代理对象

当（1）中请求的服务生成后，将调用回调方法 onBind()，返回 Binder 对象，Android Framework 调用回调方法 onServiceConnected()，它的第二个 IBinder 类型的参数接收 Binder 对象，而后将其传递给 ISecondary.Stub. asInterface()函数，并使用它生成与 RemoteService 服务绑定在一起的服务代理对象 ISecondary.Stub.Proxy，最后将其保存到 mSecondaryService 成员变量中。

至此，RemoteService 的 ISecondary 接口的绑定就完成了。Activity 可以通过保存在 mSecondaryService 成员变量中的服务代理对象，调用 RemoteService 的 getPid()方法，就像使用自身的方法一样。

```
// ISecondary.Stub.asInterface ( ) 方法的行为，请参看代码 ISecondary.java
private ServiceConnection mSecondaryConnection = new ServiceConnection() {
   public void onServiceConnected(ComponentName className,  IBinder service) {
      mSecondaryService = ISecondary.Stub.asInterface(service);
   }
};
```

代码 6-9 | RemoteService.java-Binding Activity 生成 ISecondary.Stub.Proxy 服务代理对象

更多更新编程资源尽在 http://www.pin5i.com/

（4） Binding Activity：使用服务代理对象，调用 RemoteService 服务的 getPid()代理方法

在图 6-4 中有一个 Kill Process 按钮，点击该按钮，服务代理对象的 getPid()代理方法就会被调用，获取服务进程的 pid。而后 Activity 调用 Process.killProcess()方法，强制终止指定的服务进程。Kill Process 按钮的单击事件处理代码如下：

```
private OnClickListener mKillListener = new OnClickListener() {
    public void onClick(View v) {
        int pid = mSecondaryService.getPid();
        Process.killProcess(pid);
    }
};
```

代码 6-10 | RemoteService.java-Kill Process 按钮的单击事件处理代码

（5） Binder IPC：服务代理对象（ISecondary.Stub.Proxy）向服务 Binder 对象（ISecondary.Stub）传递 Binder IPC 数据

如图 6-9 所示，ISecondary.Stub.Proxy 远程代理对象通过 Binder IPC 向 ISecondary.Stub 服务 Binder 对象传递数据，以处理（4）中对 getPid()代理方法的调用。

（6） RemoteService 服务：调用 RemoteService 服务的 getPid() Stub 方法

ISecondary.Stub 服务 Binder 对象获取 Binder IPC 数据后，会调用（2）实现的 getPid() Stub 方法，将服务进程的 ID 返回给 Activity。

至此，我们对远程服务与 Activity 绑定的方法以及绑定后 Activity 如何调用相关接口的特定方法作了简单的介绍。远程服务通过 AIDL 定义接口，有关 AIDL 行为机制更详细的讲解将在第 10 章中进行。

TIP 关于本地服务与远程服务的生成

在创建程序服务时，开发者应如何区别本地服务与远程服务，并生成它们呢？答案在 manifest 文件中。Android 的所有应用程序服务均以<service>元素的形式保存在 manifest 文件中。

接下来，我们看一下 ApiDemos 示例的 manifest 文件。以下代码是 manifest 文件中有关 LocalService 服务的部分，可以看到只有一个具体实现了本地服务的类被记录在 android:name 属性中。

```
<service android:name=".app.LocalService" />
```

而 RemoteService 的 manifest 文件的内容不同，如下：

更多更新编程资源尽在 http://www.pin5i.com/

```
<service android:name=".app.RemoteService" android:process=":remote">
    <intent-filter>
        <action android:name="com.example.android.apis.app.IRemoteService" />
        <action android:name="com.example.android.apis.app.ISecondary" />
        <action android:name="com.example.android.apis.app.REMOTE_SERVICE" />
    </intent-filter>
</service>
```

在远程服务的<service>中还使用了 android:process 这一属性。一般地，应用程序的所有组件与应用程序运行在同一进程中，如本地服务。但是，在远程服务中通过指定 android:process 属性，服务可以运行在另一个独立的进程中。

在生成远程服务时，必须指定<service>元素的 android:process 属性。若未指定该属性，相关服务将以本地服务的形式存在并运行。

关于 manifest 更详细的内容，请参考 Android 开发者网站的相关内容。

下图是在模拟器中运行 Remote Service Binding 示例时，使用 SDK 的 DDMS（Dalvik Debug Monitor Service）查看进程的截图。观察进程情况，可以发现 RemoteService 运行在一个独立的进程中，该进程在其 manifest 文件中被指定为 com.example.android.apis:remote(:remote)。

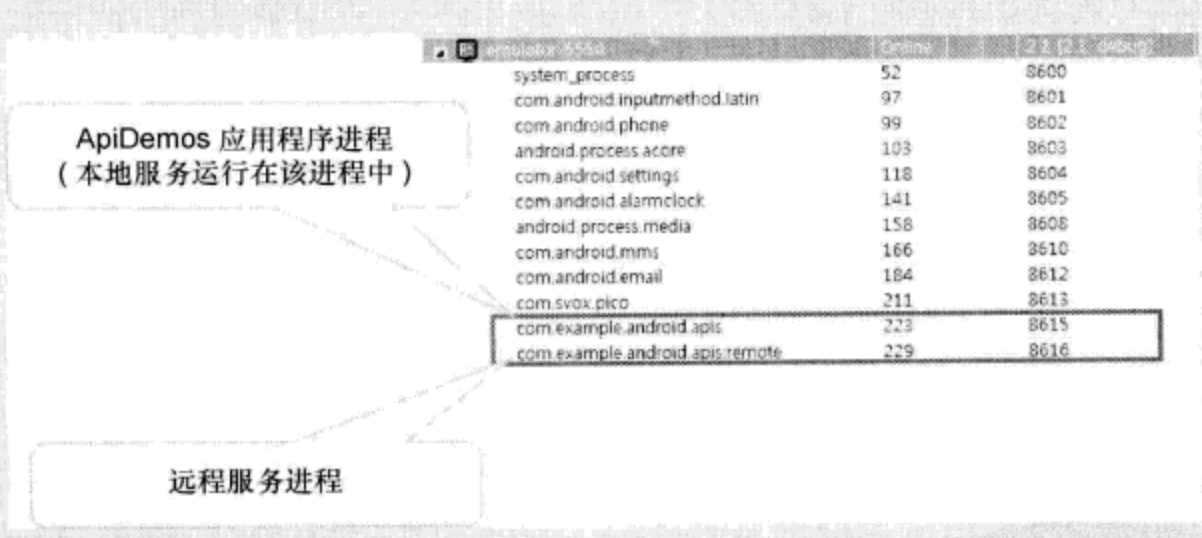


图 6-10 | Remote Service Binding 示例运行后，使用 DDMS 查看进程

6.4 Android 系统服务

Android 系统服务提供系统最基本、最核心的功能，如设备控制、位置信息、通知设定，以及消息显示等。这些服务分别存在于 Application Framework 与 Libraries 层之中，如图 6-11 所示。

该图仅仅列出了 Android Framework 中主要的系统服务，此外，还有许多其他系统服务。感兴趣的读者，请参考 Android 的源代码。

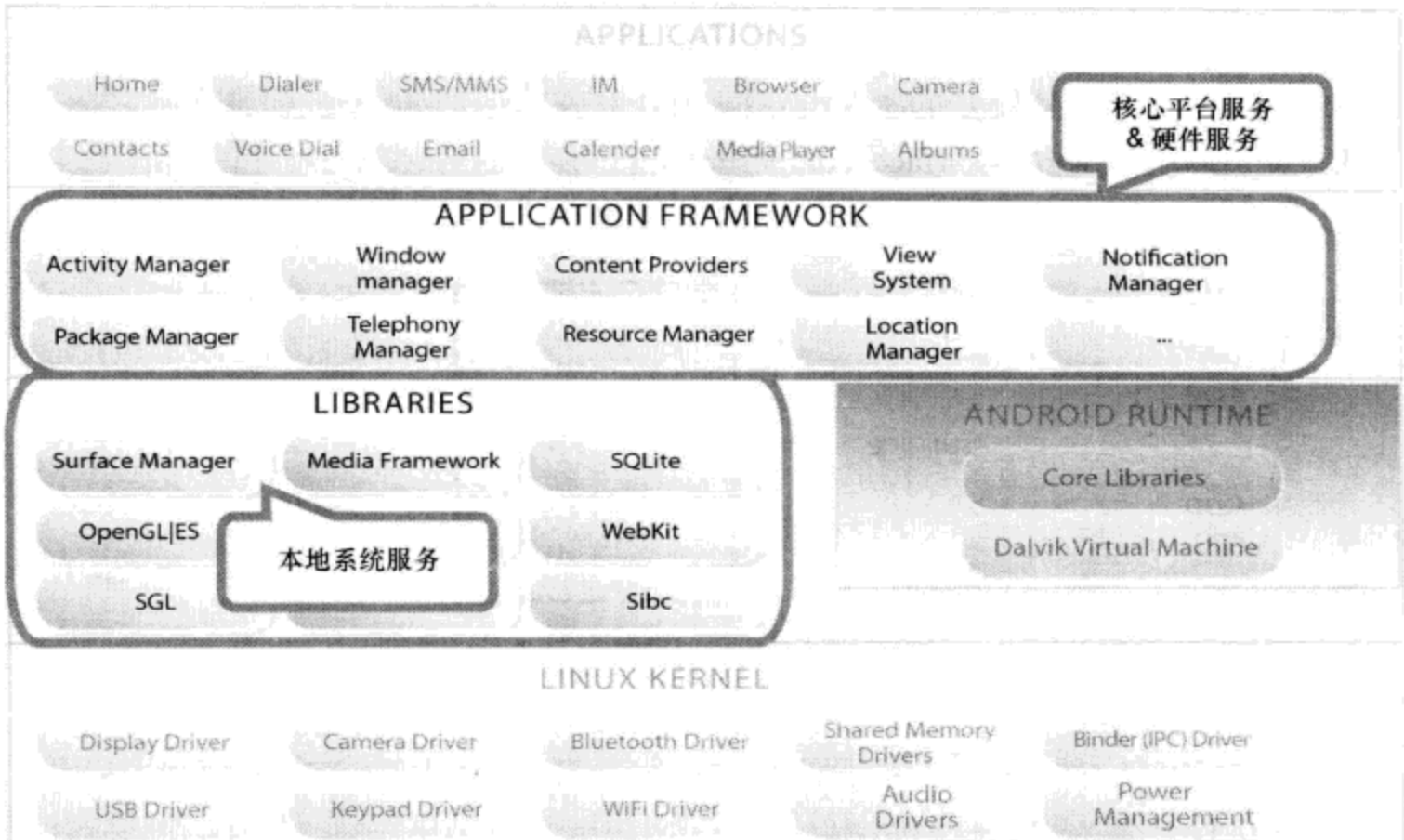


图 6-11 | Android Framework 的系统服务

系统服务分类

本地系统服务

本地系统服务使用 C++编写，运行在 Libraries 层，如图 6-11，主要包含 Audio Flinger、Surface Flinger 等。

Audio Flinger 服务

如下图所示，Audio Flinger 服务混合多种 Android 应用程序的音频数据，并发送到耳机、扬声器等音频输出设备中。在 Android 设备中，所有音频数据均经由 Audio Flinger 进行输出。

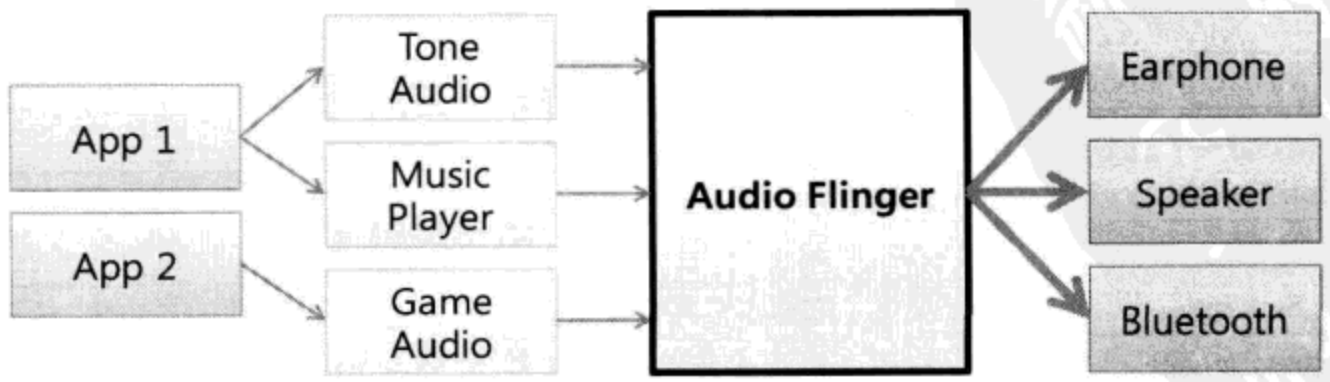


图 6-12 | Audio Flinger 服务

更多更新编程资源尽在 http://www.pin5i.com/

Surface Flinger 服务

Surface Flinger 是 Android Multimedia 的一部分，在 Android 的实现中，它是一个服务，提供系统范围内的 surface composer 功能，能够将各种应用程序的 Surface 组合后渲染到 Frame Buffer 设备中。

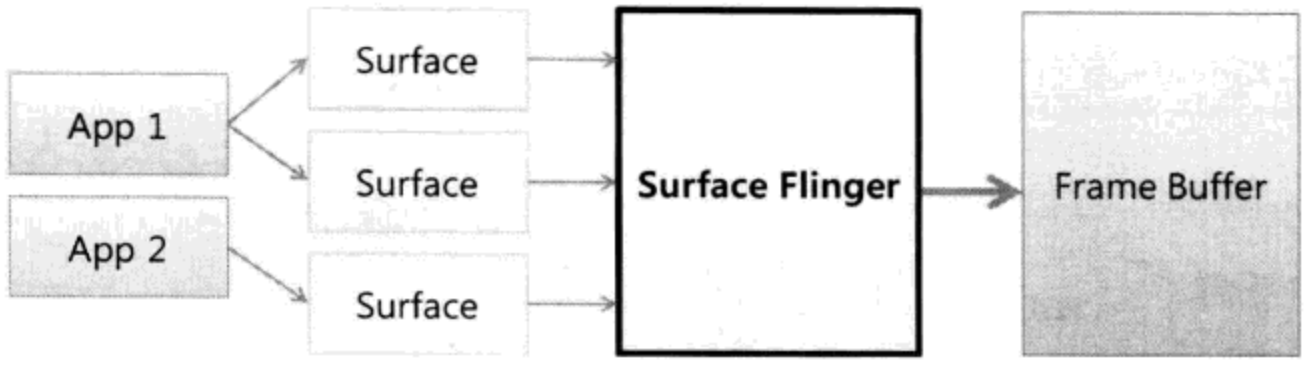


图 6-13 | Surface Flinger 服务

Java 系统服务

在 Android 启动时，Java 系统服务由 SystemServer 系统进程启动，它分为核心平台服务与硬件服务，如图 6-2 所示。

核心平台服务（Core Platform Service）

一般而言，核心平台服务不会直接与 Android 应用程序进行交互，但它们是 Android Framework 运行所必需的服务，其包含的主要服务如下表所示。

核心平台服务	功能
Activity Manager Service	管理所有 Activity 的生命周期与堆栈（Stack）
Window Manager Service	位于 Surface Flinger 之上，将要绘制到机器画面上的内容传递给 Surface Flinger
Package Manager Service	加载 apk 文件（Android 包文件）的信息，提供信息显示系统中设置了哪些包，以及加载了哪些包

硬件服务（Hardware Service）

该服务提供了一系列 API，用于控制底层硬件，主要包含如下服务。

硬件服务	功能
Alarm Manager Service	在特定时间后运行指定的应用程序，就像定时器
Connectivity Service	提供有关网络当前状态的信息
Location Service	提供终端当前的位置信息
Power Service	设备电源管理
Sensor Service	提供 Android 中各种传感器（磁力感应器、加速度传感器）的感应值
Telephony Service	提供话机状态及电话服务
Wifi Service	控制无线网络连接，如 AP 搜索、连接列表管理等

更多更新编程资源尽在 http://www.pin5i.com/

使用 Java 系统服务

无论在 Framework 内部，还是 Android 应用程序中，若想使用 Java 系统服务，必须使用能够与各服务通信的 Local Manager 对象。

如图 6-14 所示，应用程序若想使用 Location Service，获取终端设备当前的位置信息，需要先调用 getSystemService()函数，创建与 Location Service 相应的 Local Manager 对象，而后应用程序使用生成的 Local Manager 对象，调用 Location Service 提供的各种函数，执行相应的功能。

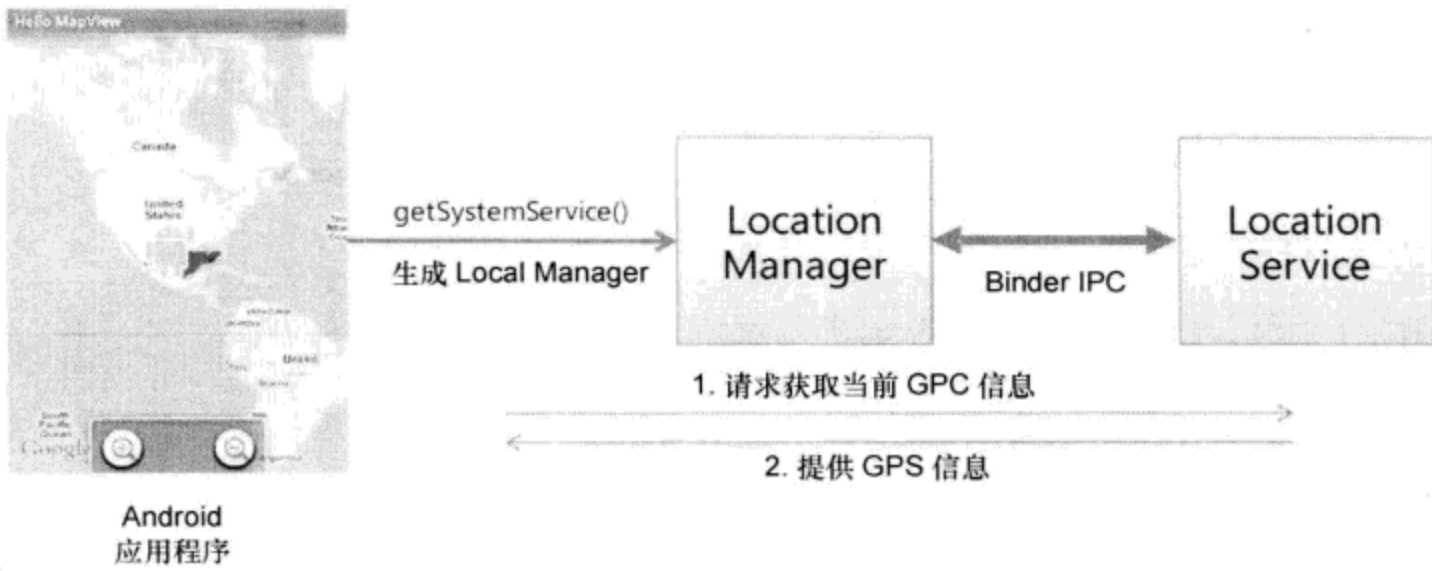


图 6-14 | Android 应用程序使用 Local Manager，访问 Location Service

6.5　运行系统服务

一般地，在使用应用程序服务前，Android 应用程序会先调用 startService()函数，启动指定的应用程序服务，而后再使用它。与之不同的是，使用系统服务时，客户端不需要先启动它，直接调用 getSystemService()使用即可。因为在 Android 系统的启动过程中，init 进程已经启动了这些系统服务。

在 Android 启动时，系统服务具体由媒体服务器（Media Server）与系统服务器（System Server）两个系统进程运行。媒体服务器进程用来启动除 Surface Flinger 之外的 Audio Flinger、Media Player Service 等本地服务。而系统服务器是 Zygote 最初生成的基于 Java 的进程，它会启动所有 Java 系统服务，例如本地系统服务 Surface Flinger。图 6-15 描述了 Android 启动时两个进程启动系统服务的过程。

下面通过代码分析，进一步了解媒体服务器与系统服务器的运行过程。

更多更新编程资源尽在 http://www.pin5i.com/

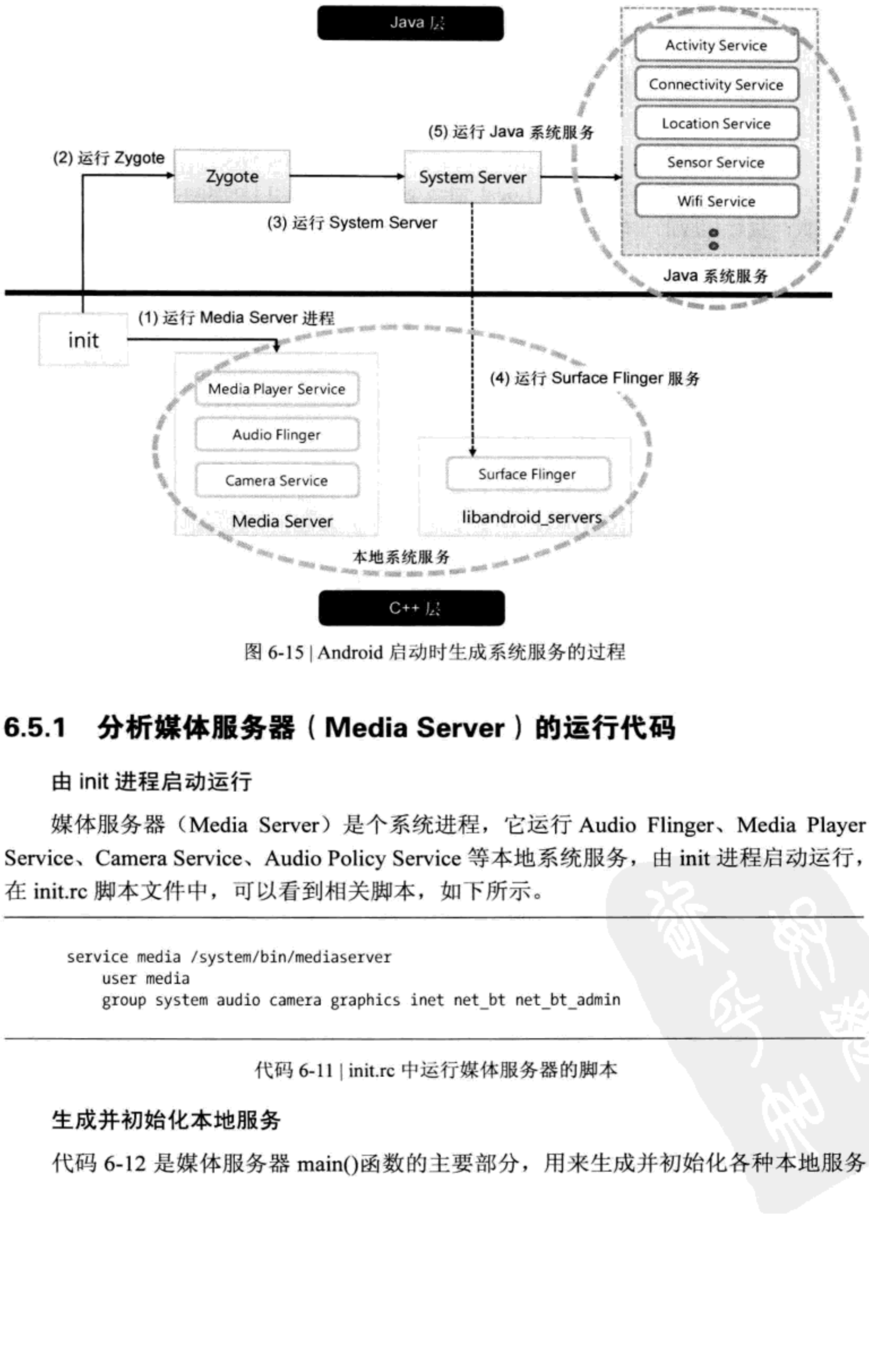


图 6-15 | Android 启动时生成系统服务的过程

6.5.1 分析媒体服务器（Media Server）的运行代码

由 init 进程启动运行

媒体服务器（Media Server）是个系统进程，它运行 Audio Flinger、Media Player Service、Camera Service、Audio Policy Service 等本地系统服务，由 init 进程启动运行，在 init.rc 脚本文件中，可以看到相关脚本，如下所示。

```
service media /system/bin/mediaserver
    user media
    group system audio camera graphics inet net_bt net_bt_admin
```

代码 6-11 | init.rc 中运行媒体服务器的脚本

生成并初始化本地服务

代码 6-12 是媒体服务器 main()函数的主要部分，用来生成并初始化各种本地服务

更多更新编程资源尽在 http://www.pin5i.com/

对象，如代码所示。

```
//frameworks/base/media/mediaserver/main_mediaserver.cpp

int main(int argc, char** argv)
{
    ...
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ...
}
```

代码 6-12 | 媒体服务器 main()函数的主要代码

分析各系统服务的初始化代码

系统服务与 Framework 中的其他模块通信时，使用 Binder IPC，系统服务这类服务提供者必须把相关信息注册到 Context Manager 中，以便 Android 应用程序这类服务使用者能够使用其提供的服务。关于 Binder IPC 机制的更多内容，将在下一章中介绍。

代码 6-13 是各本地服务的初始化代码，由代码可以看出，各个服务的初始化代码形式都相似，即首先使用 new 运算符生成服务的实例，而后调用 addService()函数将各个服务注册到 Context Manager 中。

```
// Audio Flinger 初始化代码
// frameworks/base/libs/audio?inger/AudioFlinger.cpp
void AudioFlinger::instantiate() {
    defaultServiceManager()->addService(
            ➥ String16("media.audio_?inger"), new AudioFlinger());
}

// Media Player Service 初始化代码
// frameworks/base/media/libmediaplayerservice/MediaPlayerService.cpp
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            ➥ String16("media.player"), new MediaPlayerService());
}

// Camera Service 初始化代码
// frameworks/base/camera/libcameraservice/CameraService.cpp
void CameraService::instantiate() {
    defaultServiceManager()->addService(
            ➥ String16("media.camera"), new CameraService());
}

// Audio Policy Service 初始化代码
```

更多更新编程资源尽在 http://www.pin5i.com/

```
// frameworks/base/libs/audio?inger/AudioPolicyService.cpp
void AudioPolicyService::instantiate() {
   defaultServiceManager()->addService(
          ➥ String16("media.audio_policy"), new AudioPolicyService());
}
```

代码 6-13 | 运行在 Media Server 中的系统服务初始化代码

defaultServiceManager()函数会返回 Service Manager 对象，它是一个代理对象，用来实现 Context Manager 与 Binder 之间的通信。在 Framework 中，若想使用 Context Manager 的注册或获取服务等功能，必须使用 Service Manager，更多详细内容，请阅读第 8 章中有关 Service Manager 的内容。

如代码 6-13 所示，Media Server 功能非常简单，它只负责生成各个服务的对象，并将生成的服务注册到 Context Manager 中（具体代码分析将在第 8 章中进行），这一过程如图 6-16 所示。

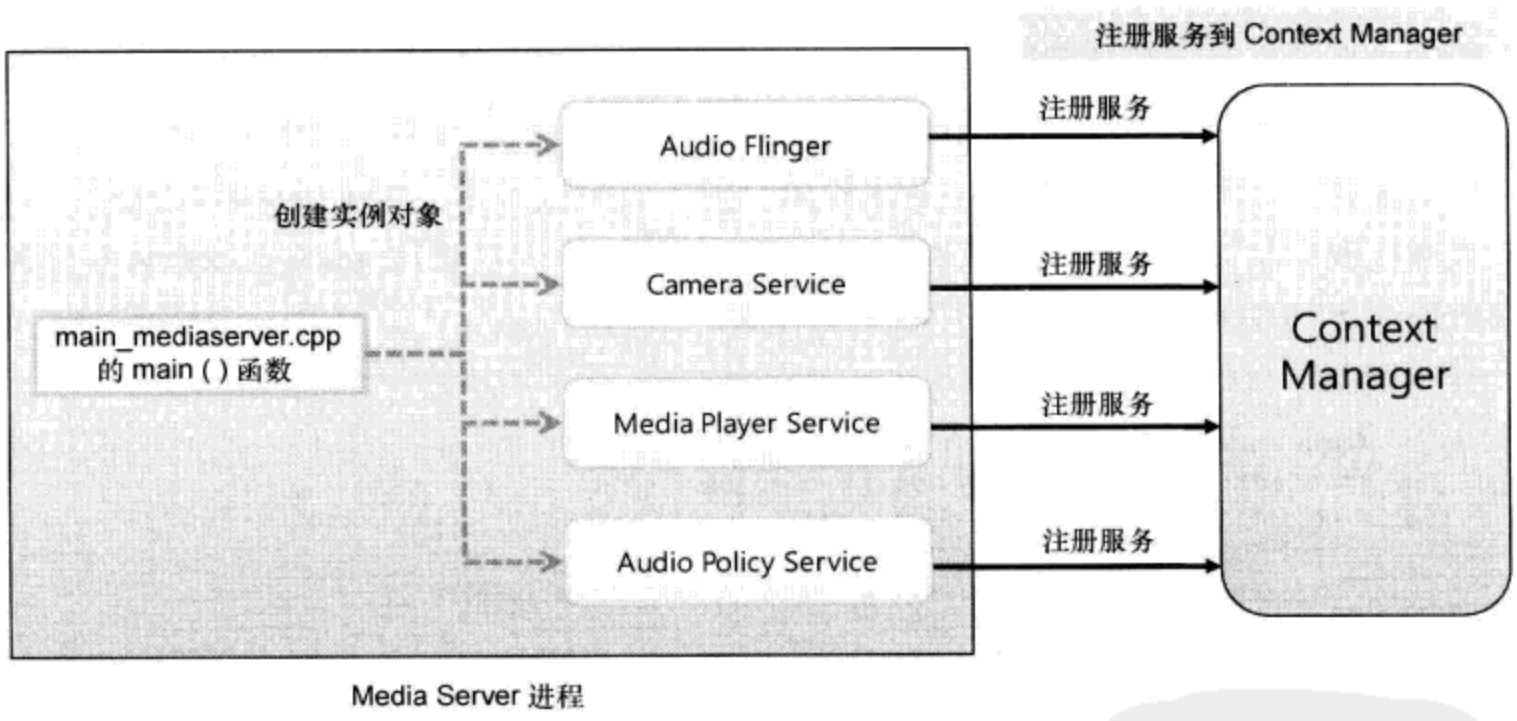


图 6-16 | Media Server[1]运行系统服务的过程

6.5.2 分析系统服务器（System Server）的运行代码

系统服务器（System Server）是一个 Java 进程，由 Zygote 进程生成。关于系统服务器是如何运行的，已经在第 5 章中讲解过，本部分从略，这里我们只分析其代码中关于创建系统服务的部分。

1 frameworks/base/media/mediaserver/main_mediaserver.cpp

更多更新编程资源尽在 http://www.pin5i.com/

由 Zygote 进程生成

在第 5 章中我们已经讲解过，System Server 是由 Zygote 进程最初生成，运行在 Dalvik 虚拟机中的 Java 进程，它用来运行多种 Java 系统服务，还有 Surface Flinger 本地系统服务。

System Server 在 Zygote 进程生成时即被运行，读者可以查看 init.rc 脚本文件中有关 Zygote 进程运行的部分，以便了解把握 System Server 是如何运行的。

代码 6-14 是从 init.rc 脚本文件中提取的有关 Zygote 运行的部分。由脚本可知，在 Zygote 运行行带有“-start-system-server”选项时，该选项请求在 Zygote 中生成 System Server。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
```

代码 6-14 | init.rc 中生成 System Server 的部分

加载 android_servers 库

代码 6-15 是 SystemServer 的 main()方法代码。关于这部分内容，在前面第 4 章讲解 JNI 时已有提及，下面结合代码，进一步讲解。

❶ SystemServer main()方法的主要功能是加载 android_servers 库（libandroid_servers.so），并调用 init1()方法。init1()方法通过 JNI 调用 system_init()本地函数。相关说明，请阅读 4.5 节中的相关内容。

```
public class SystemServer
{
   native public static void init1(String[] args);

   public static void main(String[] args) {
      System.loadLibrary("android_servers");
      init1(args);     <-- 1
   }
}
```

代码 6-15 | SystemServer main()方法的主要部分

system_init()函数的主要功能是生成并初始化本地系统服务 Surface Flinger。

更多更新编程资源尽在 http://www.pin5i.com/

❷ Surface Flinger 是基于 C++的系统服务，而 System Server 是 Java 进程，它不能直接调用 Surface Flinger 服务。System Server 必须经由 JNI 通过调用 system_init()函数来运行 Surface Flinger 服务。Surface Flinger 是本地系统服务，它采用类似 Audio Flinger 的代码来进行初始化。

```
// 初始化 SurfaceFlinger 服务

extern "C" status_t system_init()
{
    SurfaceFlinger::instantiate();                          ←❷

    AndroidRuntime* runtime = AndroidRuntime::getRuntime();
    runtime->callStatic("com/android/server/SystemServer", "init2");     ←❸

    return NO_ERROR;
}
```

代码 6-16 | system_init.cpp[1]-system_init()函数的主要部分

❸ Surface Flinger 运行后，执行 runtime->callStaic（“com/android/server/SystemServer”，“init2”）语句，调用 SystemServer 类的 init2()方法。callStaic()函数是 JNI 包装函数，它允许在 C++代码中经由 JNI 调用 Java 类的静态方法。

Java 系统服务的初始化及注册

在 SystemServer 的执行中，先初始化 Surface Flinger，而后调用 init2()方法，该方法能够生成并初始化从 Entropy 服务到 AppWidge 服务的所有 Java 系统服务。

❹ init2()方法会首先创建 ServerThread 对象，而后启动它。ServerThread 是一个 Java 线程，它可以运行 Android 的所有 Java 系统服务。

❺ 同本地系统服务一样，Java 系统服务必须先把相关服务注册到 Context Manager 中，其他模块才能使用这些服务。但是 Java 系统服务的注册方式与基于 C++的本地系统服务不同，它通过调用 ServiceManager 类的 addService()静态方法，将自身注册到 Context Manager 中。

我们可以将 ServiceManager 类[2]看作是一个存在于 Java 层并与 Context Manager 进行通信的服务管理器，更多更详细的内容将在第 10 章学习 Java Service Framework 时进行讲解。

1 frameworks/base/cmds/system_server/library/system_init.cpp

2 frameworks/base/core/java/android/os/ServiceManager.java

更多更新编程资源尽在 http://www.pin5i.com/


```
//frameworks/base/services/java/com/android/server/SystemServer.java
public static final void init2() {
    Thread thr = new ServerThread();
    thr.start();          ← ❹
}

class ServerThread extends Thread {
    ...

    public void run() {          ← ❺
        ...
        Log.i(TAG, "Entropy Service");
        ServiceManager.addService("entropy", new EntropyService());

        Log.i(TAG, "Power Manager");
        power = new PowerManagerService();
        ServiceManager.addService(Context.POWER_SERVICE, power);

        Log.i(TAG, "Activity Manager");
        context = ActivityManagerService.main(factoryTest);

        Log.i(TAG, "Telephony Registry");
        ServiceManager.addService("telephony.registry", new TelephonyRegistry(context));
        ...

        Log.i(TAG, "Clipboard Service");
        ServiceManager.addService("clipboard", new ClipboardService(context));

        Log.i(TAG, "Input Method Service");
        imm = new InputMethodManagerService(context, statusBar);
        ServiceManager.addService(Context.INPUT_METHOD_SERVICE, imm);

        Log.i(TAG, "NetStat Service");
        ServiceManager.addService("netstat", new NetStatService(context));

        Log.i(TAG, "Connectivity Service");
        connectivity = ConnectivityService.getInstance(context);
        ServiceManager.addService(Context.CONNECTIVITY_SERVICE, connectivity);
        ...

        Log.i(TAG, "Backup Service");
        ServiceManager.addService(Context.BACKUP_SERVICE,
                                  new BackupManagerService(context));

        Log.i(TAG, "AppWidget Service");
        appWidget = new AppWidgetService(context);
        ServiceManager.addService(Context.APPWIDGET_SERVICE, appWidget);
    }
        ...
}
```

代码 6-17 | ServerThread 创建并初始化 Java 系统服务

以上简单地介绍了 Media Server 与 System Server 启动运行 Android 系统服务的过

更多更新编程资源尽在 http://www.pin5i.com/

程，如图 6-17 所示。

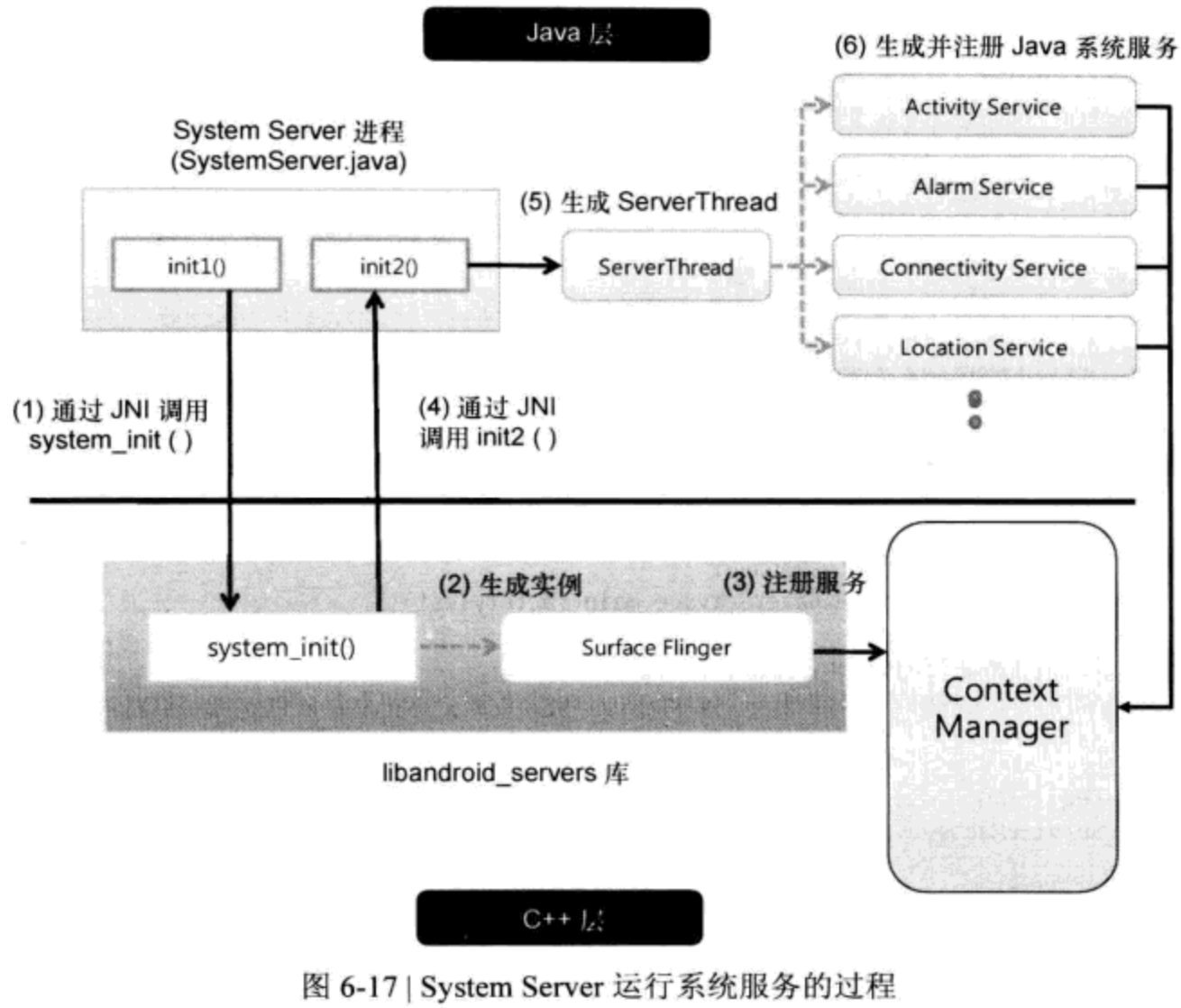


图 6-17 | System Server 运行系统服务的过程

代码看似十分简单，但若想理解整个代码内部的运行方式，就必须认真学习 Android Framework（第 8 章）与 Binder Driver（第 7 章）的相关内容。

6.6 Android Service Framework、Binder Driver 概要及相关术语

Android 系统服务体系庞大且复杂，仅凭一本书来分析完所有的系统服务是不可能的。（与应用程序服务相关的内容，请参考 Android 应用程序编程相关的书籍。）并且，各个系统服务本身又十分复杂、庞大，学习时，由于篇幅有限，无法把所有相关的背景知识都呈现给大家。

因此，本书在讲解 Android 系统服务时，将着力点放在对系统服务运行原理的理解上。在本书的第 7 章到第 11 章，将向大家介绍一些与系统服务运行原理相关的概念（Binder Driver 与 Service Framework），通过一些实例来具体地讲解系统服务（Camera

更多更新编程资源尽在 http://www.pin5i.com/

Service、Activity Manager Service）相关的知识。

从概念来看，Android 系统服务的运行原理十分简单，将其想象成服务使用者与系统服务间的 RPC 即可，即服务使用者根据事先约定好的接口远程调用系统服务提供的函数，从而使用其提供的功能，这就是 Android 系统服务基本的运行原理。

原理虽然简单，但内部实现相当复杂。因此 Android 提供了设计精良的 Service Framework 与 Binder Driver，以便开发者能容易地使用复杂的 RPC 机制。如图 6-18，它描述了服务使用者以 RPC 的方式调用 Foo 系统服务提供的 foo()方法的过程，服务运行原理形象、具体，一目了然。

首先介绍一下主要术语。（部分术语会在后续章节中作详细的说明，此处仅简略提及。其中，粗体术语是为了说明的方便本书新定义的。）

- 服务管理器（Service Server）：指运行系统服务的进程，相当于前面提到的 System Server 或 Media Server。
- 服务客户端（Server Client）：指使用系统服务的进程。
- 上下文管理器（Context Manager）：是一个管理系统服务的系统进程，它管理安装在系统中的各种系统服务的位置信息 Handle，这些 Handle 用来指定 Binder IPC 的目的地址。
- 服务框架（Service Framework）：包含前面提到的 Service Manager，其中定义了一系列类，用于服务使用者与系统服务间的 RPC 操作。
- 服务接口（Service Interface）：它是一个预先定义的接口，用在服务使用者与系统服务间。系统服务应该根据相关接口实现 Stub 函数，并提供相关服务。而服务使用者也必须根据相关接口调用服务。
- 服务使用者：在服务客户进程中实际使用服务的模块。
- 服务（Service）：由服务 Stub 函数实现定义在服务接口中的功能，是提供实际服务功能的模块。
- 服务代理（Service Proxy）：执行 RPC 时用来对数据进行 Marshalling 处理的对象，不同的服务接口对应不同的服务代理。它提供服务代码函数，根据服务接口中定义的函数，对数据分别进行不同的 Marshalling 处理。
- 服务 Stub：RPC 执行时用来对数据进行 UnMarshalling 处理的对象，该对象随接口不同而不同。它对接收到的数据进行 UnMarshalling 处理后，调用相关的服务 Stub 函数。
- Binder Driver：Binder 是 Android 中为支持 IPC 而采用的机制，它以 Android Linux 内核的 Device Driver 形态存在。

更多更新编程资源尽在 http://www.pin5i.com/

- Binder IPC：它是 Android 中进程间通过 Binder Driver 交换数据的方式。
- Binder IPC 数据：一种用在 Service Framework 与 Binder Driver 间的数据格式。
- Binder RPC：服务会向使用者提供基于特定服务接口的函数，服务使用者通过 Binder IPC 调用这些函数，就像调用自身函数一样。Binder IPC 内部是基于 Binder IPC 机制的。
- Binder RPC 数据：服务使用者与服务间进行 Binder IPC 时使用的数据。

下面举一个例子，描述服务使用者如何通过 Binder RPC 调用系统服务，如图 6-18 所示，服务使用者调用 foo()服务代理函数，而后 foo()服务代理函数通过 Binder RPC 调用 Foo 服务的 foo()服务 Stub 函数。

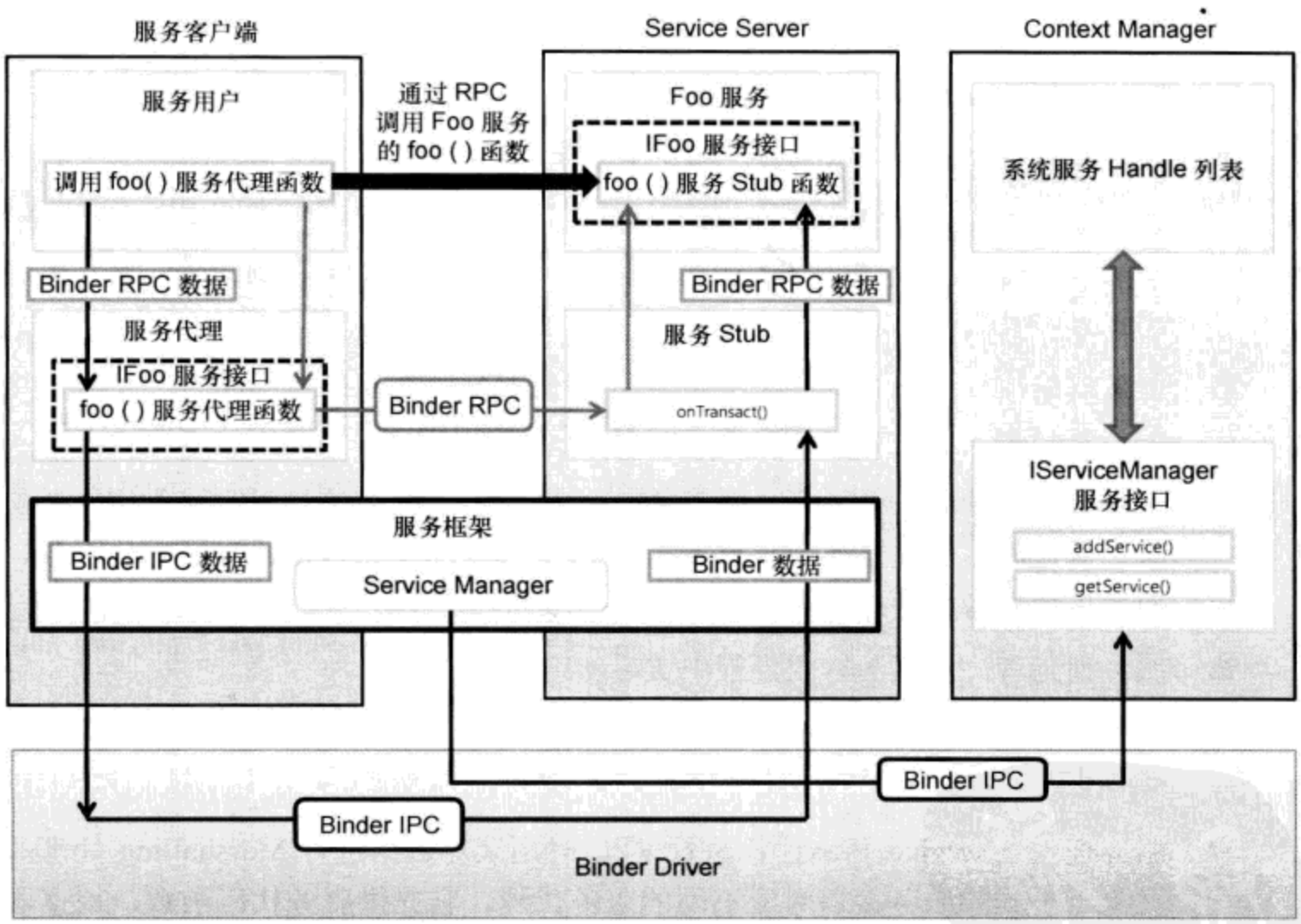


图 6-18 | Android 系统服务调用流程

首先，服务使用者调用 foo()代理函数，传递 Binder RPC 数据，该数据包含引用 Foo 服务的请求。Binder RPC 数据经过 Marshalling 处理后，由 Service Framework 生成 Binder IPC 数据，然后通过 Binder Driver 传递给 Service Server 端。

Service Server 端接收到 Binder IPC 数据后，由 Service Framework 对数据进行

更多更新编程资源尽在 http://www.pin5i.com/

UnMarshalling 处理，然后传递给 Service Stub 的 onTransact()函数，Service Stub 根据 Binder IPC 数据中的 RPC 代码判断它是一个针对 Foo 服务的 foo()服务 Stub 函数的 Binder RPC。最后，以 Binder IPC 数据中包含的 Binder RPC 数据为参数，调用 foo()服务 Stub 函数。

以上我们简单地学习了 Android 中有关系统服务的基本运行原理与术语的知识。在接下来的第 7 章与第 8 章中，我们将继续学习相关内容。在第 7 章“Android Binder IPC”中，将学习有关 Binder Driver 与 Context Manager 的内容；而第 8 章主要学习 Service Framework、Service Proxy、Servcie Stub 等知识。

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 7 章

Android Binder IPC

BeOS 是 Be 公司在 1991 年开发的运行在 BeBOX 硬件上的一款操作系统，与同期的其他操作系统不同，它是一款基于 GUI 设计的操作系统。

乔治·霍夫曼（George Hoffman）当时任 Be 公司的工程师，他启动了一个名为“OpenBinder”[1]的项目，该项目的宗旨是研究一个高效的信号传递工具，允许软件的各个构成元素相互传递信号，从而使多个软件相互协作，共同形成一个软件系统。在 Be 公司被 ParmSource 公司收购后，OpenBinder[2]由 Dinnie Hackborn 继续开发，后来成为管理 ParmOS6 Cobalt OS 的进程的基础。在 Hackborn 加入谷歌后，他在 OpenBinder 的基础上开发出了 Android Binder（以下称“Binder”），用来管理 Android 的进程。

Binder 原本是 IPC（Inter Process Communication）工具，但在 Android 中它主要用于支持 RPC（Remote Procedure Call），使得当前进程调用另一个进程的函数时就像调用自身的函数一样轻松、简单。

7.1 Linux 内存空间与 Binder Driver

Android 是基于 Linux 内核的操作平台，在学习 Binder 之前，需要先了解有关 Linux 内核内存空间的知识。Android 进程与 Linux 进程一样，它们只运行在进程固有的虚拟地址空间中。一个 4GB 的虚拟地址空间，其中 3GB 是用户空间，剩余的 1G 是内核空间（可通过内核设定修改）。用户代码与相关库分别运行在用户空间的代码区域、数据区域，以及堆栈区域中，而内核空间中运行的代码则运行在内核空间的各个区域中。并且，进程具有各自独立的地址空间，单独运行，如图 7-1 所示。

那么，一个拥有独立空间的进程如何向另一个进程传递数据呢？显然要通过两个进程共享的内核空间。虽然各个进程拥有各自独立的用户空间，但内核空间是共享的。从内核的角度看，进程不过是一个作业单位，虽然各进程的用户空间相互独立，但运行在内核空间中的任务数据、代码都是彼此共享的。

---

1 http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html

2 现在 OpenBinder 是运行在 Linux 上的 OpenSource。

更多更新编程资源尽在 http://www.pin5i.com/

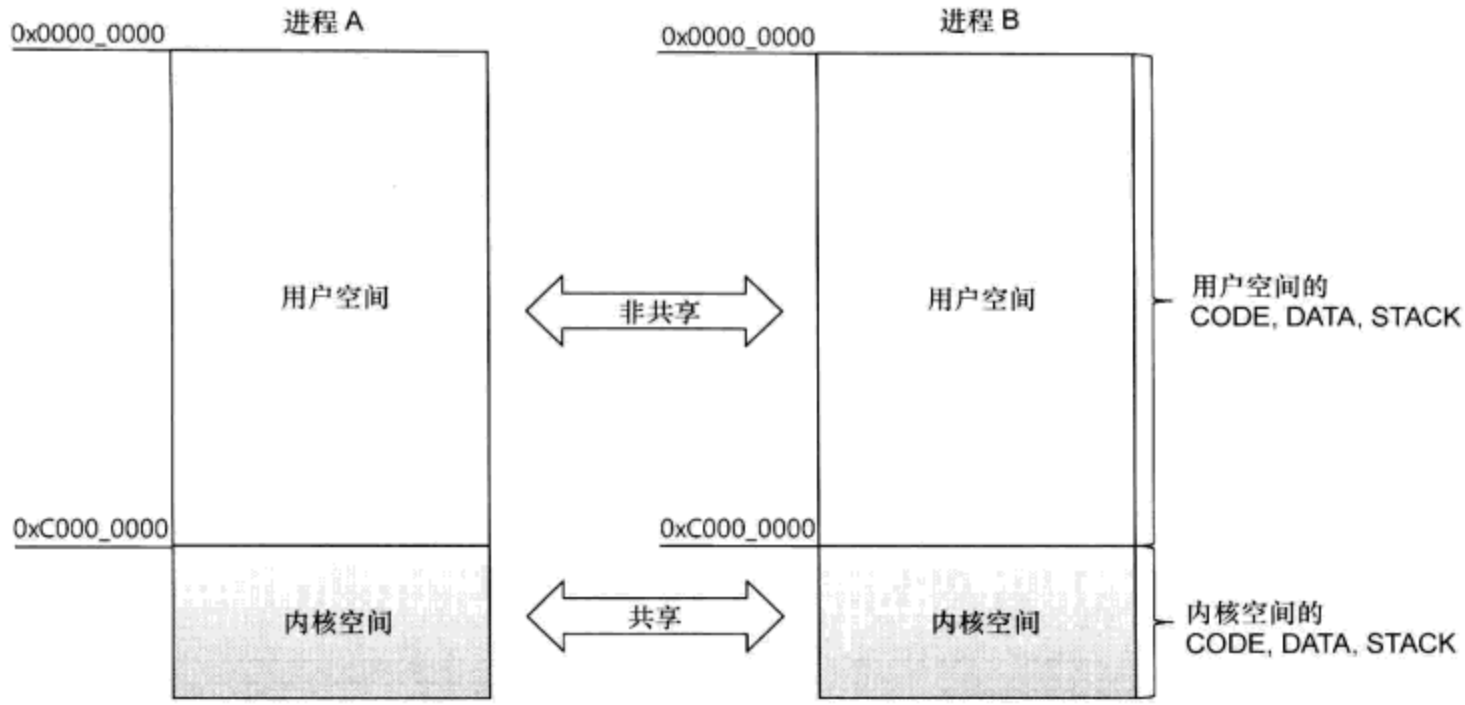


图 7-1 | 内核与用户空间

例如，在搭载了 Android 系统的手机中，使用相机拍照后，将其显示在桌面上。在这一过程中主要涉及两个进程，一个进程用来驱动相机，另一个进程用来将照片显示在桌面上。相机驱动进程在拍完照之后，必然要通过负责画面输出的进程，将拍摄的照片显示到手机桌面上。那么驱动相机的进程是如何向负责画面输出的进程提出请求，把照片显示到桌面上的呢？

经过前面的学习，我们知道两个进程共享内核空间，所以相机驱动进程可以通过内核空间把显示请求传递给负责画面输出的进程，即两个进程通过内核空间，互相交换信息，实现 IPC 通信。

事实上，Linux 本身就提供 IPC 工具，用于两个进程通过内核空间进行通信。但是 Android 中的 Binder 功能更丰富，不仅可以用来实现进程间的 IPC 通信，还可以用来调用另一个进程的函数，即支持进程间的 RPC 操作。在上面的例子中，相机驱动进程可以通过 Binder 调用负责画面输出的进程中的函数，将照片输出显示在手机桌面上。

各个进程的用户空间是无法共享的，为实现进程间的通信，Binder 使用运行在内核空间中的抽象驱动程序 Binder(IPC) Driver（以下称 Binder Driver），如图 7-2 所示。

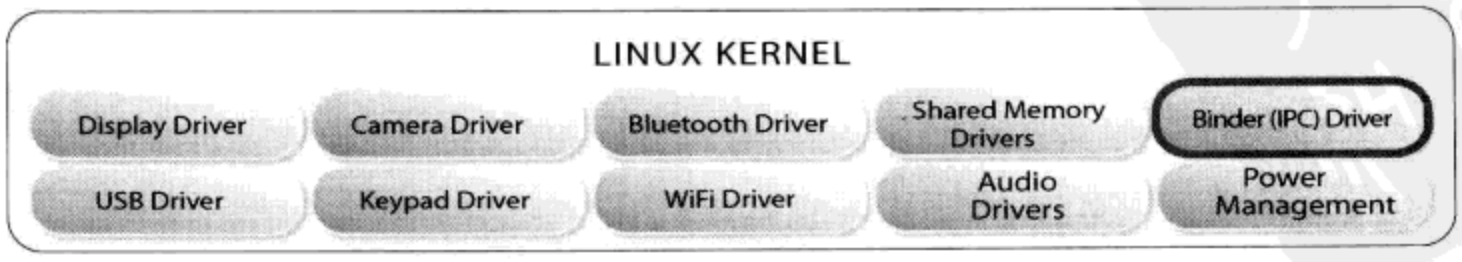


图 7-2 | Android 内核[1]的 Binder Driver

1 Android 内核在 Linux 内核基础上添加了 Android 独有的 Binder、Low Memory Killer、Ashmem 等功能。

更多更新编程资源尽在 http://www.pin5i.com/

在 Android 中通过 Binder Driver 实现进程间通信的原因如下。首先 Binder 采用 Linux 中优秀的内存管理技术，在通过内核空间传递数据时能确保数据的可靠性。其次，由于使用用户空间无法访问的内核空间来交换数据，所以 IPC 间的安全问题得到解决。

在 7.2 节“Android Binder Model”中，将讲述 Binder 的运行原理。在 7.3 节“Android Binder Driver 分析”中，将分析 Binder Driver 代码，一点点揭开 Binder 的神秘面纱。

7.2 Android Binder Model

在介绍 Android Binder Model 之前，先通过对 Surface Flinger[1]运行的分析，观察 Binder 在输出画面时起了哪些作用。

如图 7-3 所示，进程 A 负责显示 Home 桌面与状态栏，而进程 B 负责为机器的待机模式生成锁定画面。进程 C 是一个 Service Server，提供 Surface Flinger 服务，组合 Surface(画面)，并将数据传送到 Frame Buffer(LCD 画面的 Buffer)中，即 Surface Flinger 会组合进程 A 与 B 的画面，并将组合后的结果输出到 LCD 画面中。Binder 用于各进程间进行 RPC 操作，例如进程 A 与 B 通过 Binder 调用 Surface Flinger 的 LCD 输出函数，输出图像画面。在本示例中，进程 A 与 B 是服务的客户端（Service Client），它们使用 Surface Flinger 服务，而进程 C 是 Service Server，它提供 Surface Flinger 服务。那么，服务客户端如何通过 Binder IPC 调用运行在其他进程中的服务的函数呢？

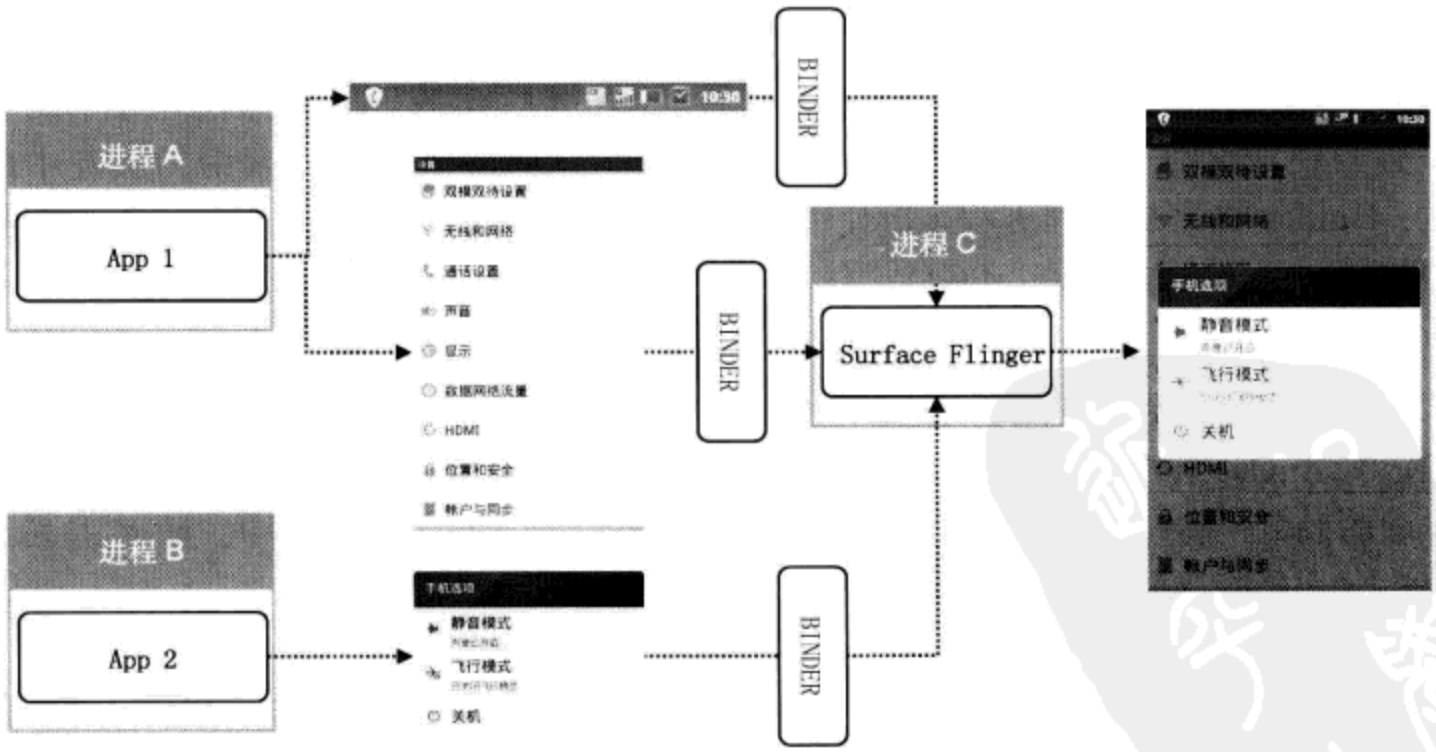


图 7-3 | 通过 Android 应用程序与 Surface Flinger 间的 Binder 执行 RPC 操作

1 在第 6 章“Android 服务概要”中已作说明。

更多更新编程资源尽在 http://www.pin5i.com/

如图 7-4 所示，服务客户端通过 Binder 调用 Service Server 的 foo()函数，整个调用过程如图所示。服务客户端要通过 Binder IPC 实现对 Service Server 的 foo()函数的调用，就需要将 Binder IPC 数据（或 IPC 数据）传递给 Service Server。如何传递呢？这就需要 Binder Driver，它充当中间人的角色，接收来自服务客户端的调用 foo()函数的 Binder IPC 数据，而后将其传递给 Service Server，即 Binder Driver 是通信媒介，用来实现进程间的通信。图 7-4 中的 Binder IPC 数据包含调用其他进程函数的信息，是 Binder Driver 操作的 IPC 数据单位。

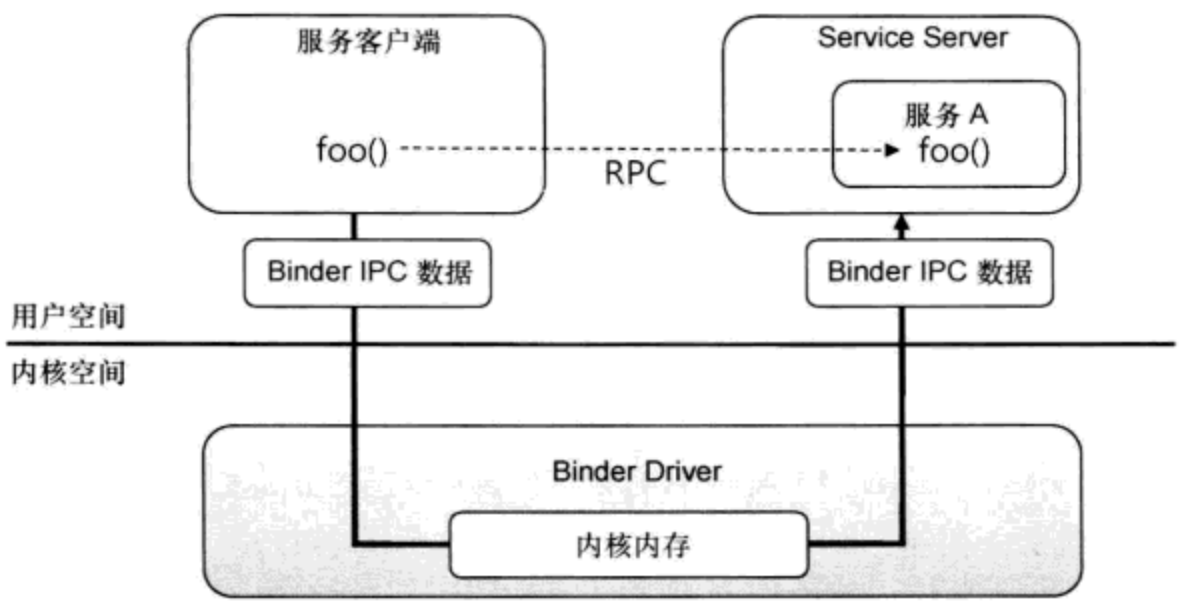


图 7-4 | Binder Driver 的作用

IPC 数据包含函数调用相关的内容，它由待调服务号、待调函数名、Binder 协议（BINDER PROTOCOL[1]）构成。在 Android 中服务号用来标记运行中的各个服务，便于区分不同服务；待调函数名用来指定服务客户端将调用的服务中的函数；Binder 协议是处理进程（指使用 Binder Driver 与 Binder 的进程）间 IPC 数据的约定。这些数据保存在 IPC 数据的各个变量[2]中，IPC 数据的组成结构，如图 7-5 所示。

在图 7-5 中，Handle[3]指服务号，用来区分不同服务。Binder Driver 通过 Handle 值确定将 Binder IPC 数据传递到哪个服务中。RPC 代码与 RPC 数据分别表示待调函数与传递的参数，即 RPC 代码确定数据要传递给服务中的哪个函数，RPC 数据[4]是传递给 RPC 代码所指函数的参数。Binder 协议表示 IPC 数据的处理方法。

---

1 在 7.2.3“Binder Protocol”中讲解。

2 IPC 数据以结构体形态存在，结构体内的各变量分别保存各种数据。关于 IPC 数据，将在 7.3.3“Binder Driver 函数分析”一节中讲解。

3 在 7.2.5“Binder Addressing”一节中讲解。

4 在 7.2.4“RPC 代码与 RPC 数据”一节中讲解。

更多更新编程资源尽在 http://www.pin5i.com/

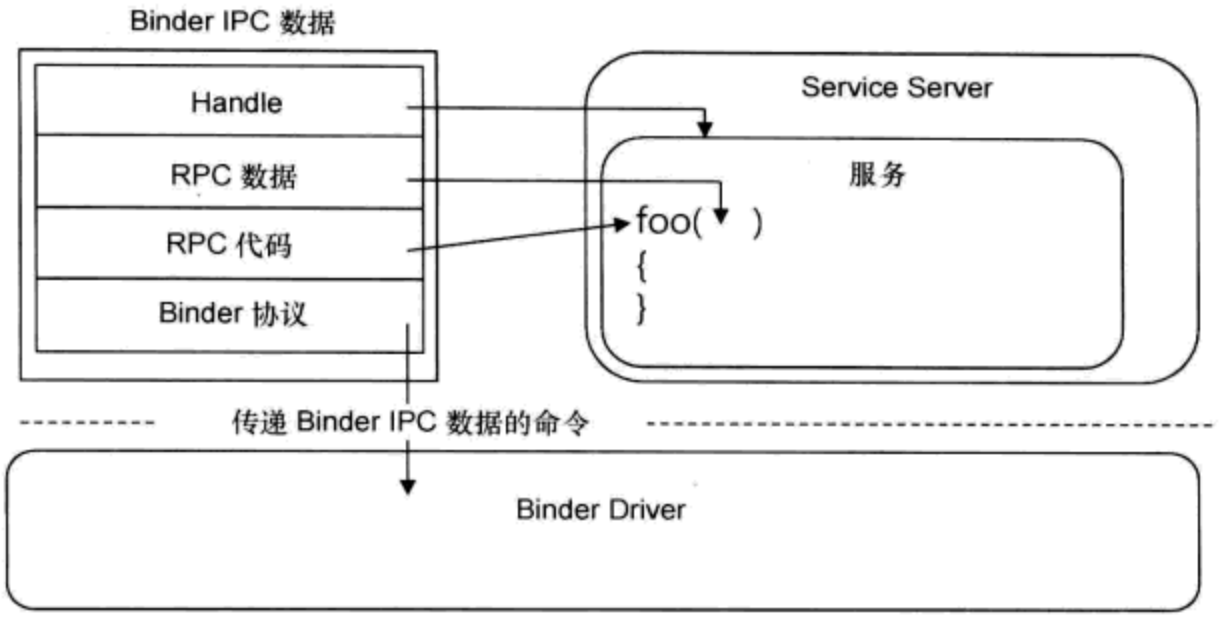


图 7-5 | Binder IPC 数据构成及作用

7.2.1 Binder IPC 数据传递

Binder Driver 是字符设备驱动程序，通过调用 open()或 ioctl()函数即可访问。如图 7-6 所示，显示了系统调用与 Binder Driver 中文件运算函数之间的连接关系。例如，系统调用 open()函数与 Binder Driver 的 binder_open()函数连接在一起。[1]

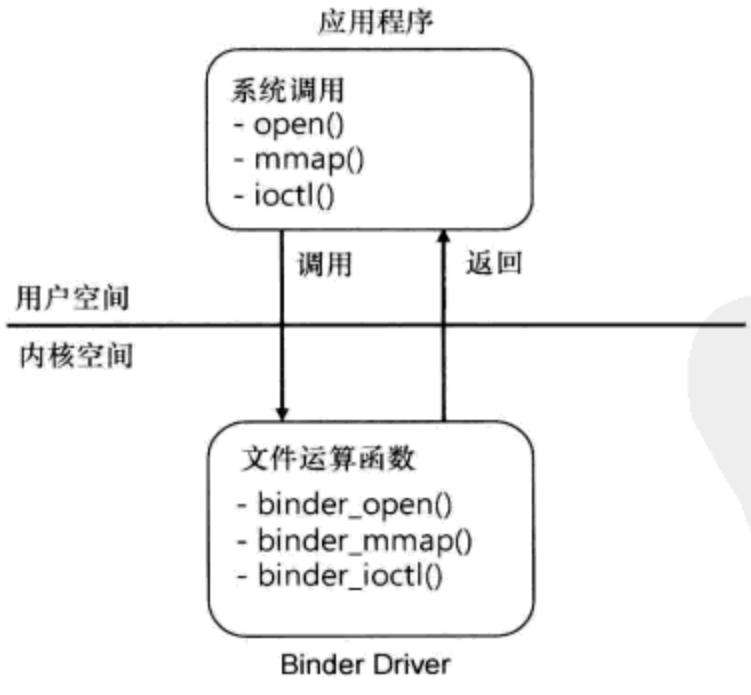


图 7-6 | 系统调用[2]与 Binder Driver 文件运算函数间的连接关系

1 文件输入输出函数与 Binder Driver 的文件运算函数在 binder_init()函数注册 MISC 设备驱动程序时被连接起来。该过程与字符设备驱动程序将自身文件处理函数与文件输入输出函数连接的过程是类似的，不同是前者不会自动生成设备节点。

2 系统调用是用户空间调用内核空间函数的接口。应用程序通过系统调用包装函数进行系统调用。例如，在图 7-6 中系统调用顺序是：open()-> __open()->binder_open()，其中__open()是系统调用。但本章不会对内核运行做过多探讨，可以认为 open()就是系统调用。

更多更新编程资源尽在 http://www.pin5i.com/

应用程序在通过 Binder 尝试 RPC 操作时，会进行 open()系统调用，获取 Binder Driver 的文件描述符。[1]而后，通过 mmap()系统调用，在内核中开辟一块区域，以便存放接收的 IPC 数据。最后，调用 ioctl()函数，将 IPC 数据作为参数，传递给 Binder Driver。Binder Driver 的 ioctl()函数形式如下。

- ioctl（文件描述符，ioctl 命令，数据类型）

ioctl()函数的第一个参数是文件描述符，它在调用 open()函数打开 Binder Driver 时返回；第二个参数是 ioctl 命令，有多种不同类型的命令，数据类型也不相同。表 7-1 列出了 ioctl 命令、Binder Driver 动作，以及数据类型之间的对应关系。

表 7-1 Binder ioctl 命令的种类

ioctl 命令	Binder Driver 动作	数据类型
BINDER_WRITE_READ	在进程间接收发送 Binder IPC 数据	struct binder_write_read
BINDER_SET_IDLE_TIMEOUT	未使用	int64_t
BINDER_SET_MAX_THREADS	设定注册在 Binder Driver 中的 Binder 线程的（BINDER THREAD）*最大个数	size_t
BINDER_SET_IDLE_PRIORITY	未使用	int
BINDER_SET_CONTEXT_MGR	设定 Binder Driver 的特殊节点（Specical Node）**	int
BINDER_THREAD_EXIT	删除 Binder 线程	int
BINDER_VERSION	提供 Binder 协议的版本	struct binder_version

*服务客户端在请求使用服务时，Binder Driver 会将 IPC DATA 中名为 BR_SWAN_LOOPER 的 RPC 代码传递给 Service Server。Service Server 接收 BR_SWAN_LOOPER 后，生成相应的服务线程。

**特殊节点（Special Node）是一个 Context Manager 进程的 Binder 节点。该节点被注册到 Binder Driver 内的全局变量 binder_context_mgr_node 中。Context Manager 进程通过 BINDER_SET_CONTEXT_MGR ioctl 命令，将自身的 Binder 节点注册到 Binder Driver 中。关于 Context Manager，请参考 7.4 节“Context Manager”的内容。

本章讨论的重点是 IPC 数据，它由 BINDER_WRITE_READ ioctl 命令传递给 Binder Driver。除了 BINDER_WRITE_READ 之外，其他 ioctl 命令也与 Binder IPC 有关。但由于重点在 Binder IPC 数据的传递上，所以后面只讨论由 BINDER_ WRITE_READ 引起的 Binder Driver 行为。

7.2.2 Binder IPC 数据流

在前面，我们已经学习过 Binder IPC 数据的作用，还了解了 IPC 数据是如何传递到 Binder Driver 中的。下面我们将进一步了解 Binder IPC 数据从服务客户端传递到

1 该方法主要在应用程序使用设备驱动程序时使用，如果不了解这一过程，请先参考驱动程序相关的书籍。

更多更新编程资源尽在 http://www.pin5i.com/

Service Server 的过程。图 7-7 描述了服务客户端调用 Servcie Server 的服务函数的过程。如图所示，服务客户端在尝试 RPC 调用 Servcie Server 的 foo()函数时，Android 系统采用何种方式传递 Binder IPC 数据，一目了然。

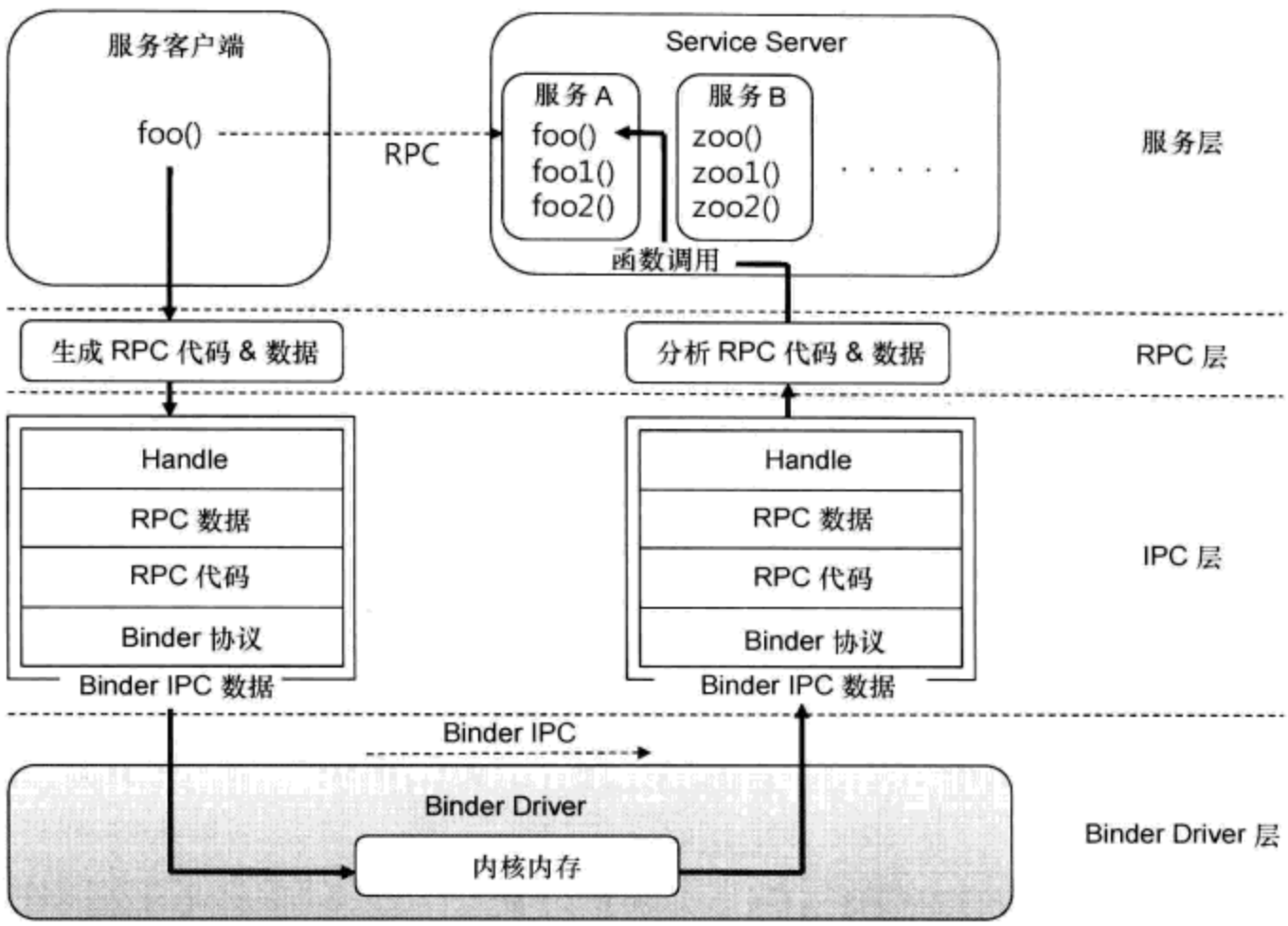


图 7-7 | Binder IPC 抽象层

Android 系统提供了几种抽象层，以便服务客户端使用指定的服务。服务客户端通过底层提供的行为，在服务层中进行 RPC 操作。服务层之下的层会生成 Binder IPC 数据，用于支持服务层的 RPC。

- 服务层：该层包含一系列提供特定功能的服务函数。服务客户端虚拟调用特定的服务函数，而实际调用是由 Service Server 进行的，即 Service Server 实际调用服务客户端请求的服务函数。
- RPC 层：服务客户端在该层生成用于调用服务函数的 RPC 代码与 RPC 数据。Service Server 会根据传递过来的 RPC 代码查找相应的函数，并将 RPC 数据传递给查找到的函数。
- IPC 层：该层将 RPC 层生成的 RPC 代码与 RPC 数据封装成 Binder IPC 数据，以便将它们传递给 Binder Driver。
- Binder Driver 层：接收来自 IPC 层的 Binder IPC 数据，查找包含指定服务的 Service Server，并将 IPC 数据传递给查找到的 Service Server。

更多更新编程资源尽在 http://www.pin5i.com/

如图 7-7 所示，服务客户端通过调用 ioctl()将 Binder IPC 数据传递给 Binder Driver，而后 Binder Driver 将其传递给指定的 Service Server。在这一过程中，Binder Driver 会根据 IPC 数据中的 Binder 协议，决定是否传递数据。有关内容将在下一节“Binder 协议”中进行讲解。

7.2.3 Binder 协议（Binder Protocol）

Binder 协议包含在 Binder IPC 数据中，它从 IPC 层传递到 Binder Driver，或从 Binder Driver 传递到 IPC 层。根据传递方向，Binder 协议分为两种，一种是从 IPC 层传递给 Binder Driver 的“BINDER COMMAND PROTOCOL”，另一种是从 Binder Driver 传递给 IPC 层的“BINDER RETURN PROTOCOL”。两种协议的区别在于 BINDER COMMAND PROTOCOL 以“BC_”打头，而 BINDER RETURN PROTOCOL 以“BR_”打头。如图 7-8 所示，它描述了 Binder 的两种协议。

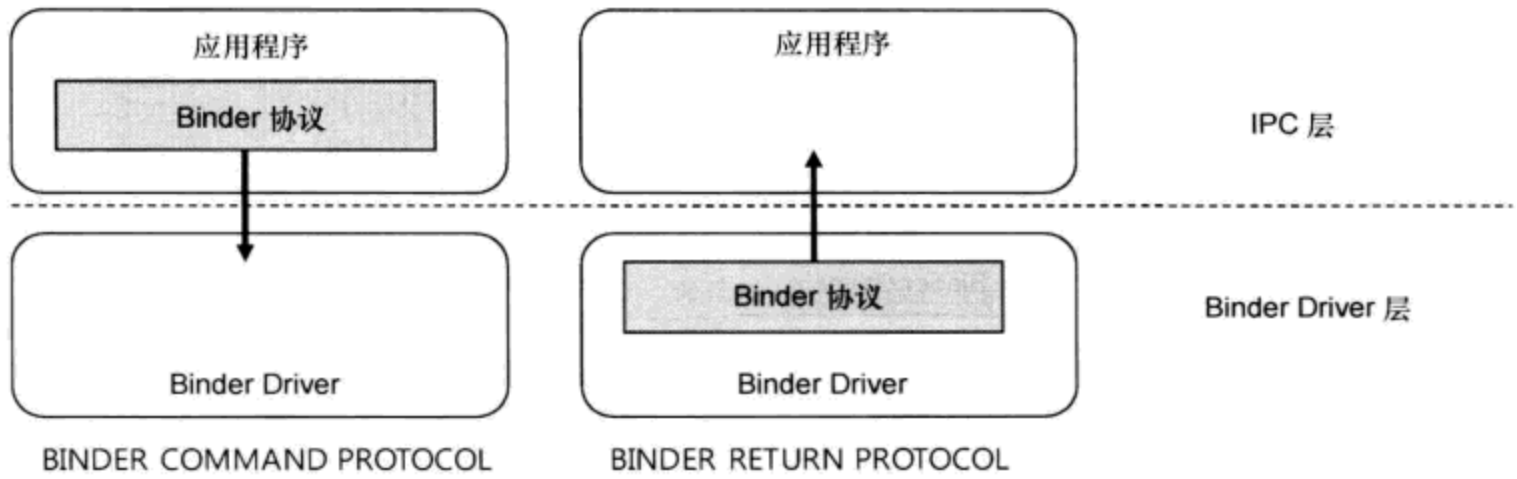


图 7-8 | Binder 协议的分类

类似网络协议，Binder 协议是数据接收方与发送方共同遵守的规定或规则。因此使用 Binder IPC 的进程与 Binder Driver 在头文件[1]中定义了 Binder 协议。表 7-2 中列出了传递 IPC 数据时使用的 Binder 协议。

表 7-2 Binder 协议

分类	Binder 协议	含义
BINDER COMMAND PROTOCOL	BC_TRANSACTION	在 Binder IPC 数据发送端通过 Binder Driver 向接收端发送 IPC 数据时使用
BINDER RETURN PROTOCOL	BR_TRANSACTION	用于 Binder IPC 数据接收端分析处理 IPC 数据

1 在内核 2.6.32 中，Binder Driver 的头文件在 kernel/drivers/staging/android/binder.h 目录下。

更多更新编程资源尽在 http://www.pin5i.com/

如图 7-9 所示，它描述了在 BC_TRANSACTION 与 BR_TRANSACTION 两种协议下 Binder Driver 与 IPC 数据接收端的不同行为动作。首先服务客户端使用 BC_TRANSACTION 将 Binder IPC 数据传递给 Binder Driver。Binder Driver 接收 IPC 数据后，分析其中所使用的协议，若为 BC_TRANSACTION，则根据 IPC 数据中的 Handle 信息查找相应的 Service Server[1]。然后 Binder Driver 会将协议更改为 BR_TRANSACTION，并将其重新插入 IPC 数据中，把包含 BR_TRANSACTION 协议的 IPC 数据传递到 Service Server 中。Service Server 接收到来自 Binder Driver 的 IPC 数据后，分析其中的协议，若为 BR_TRANSACTION，则进一步分析 IPC 数据，调用服务客户端请求的函数。如前所述，Service Server 具体调用服务的哪个函数是由 IPC 数据中的 RPC 代码决定的，在下一节中，我们将探讨 RPC 代码与服务函数之间的关系。

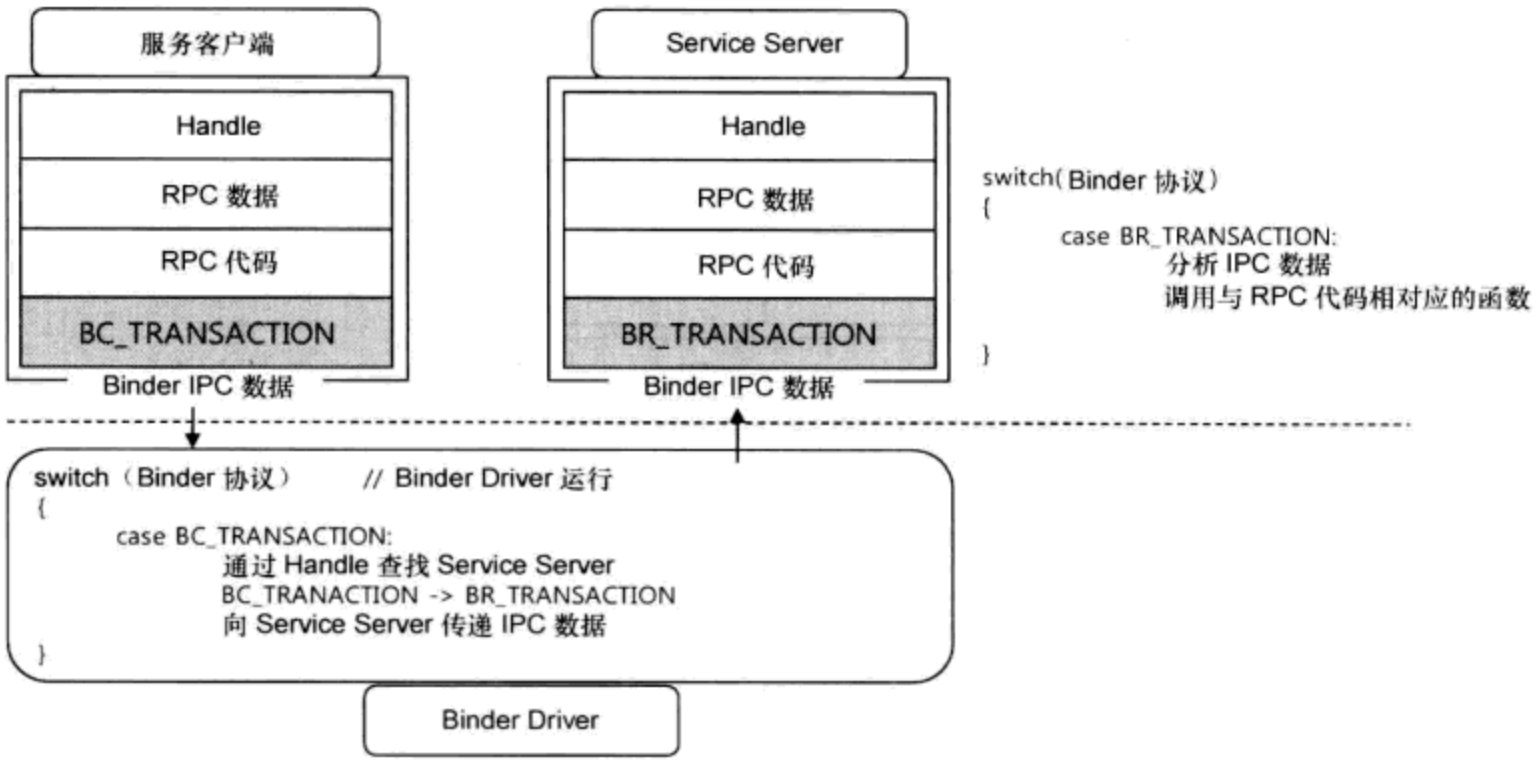


图 7-9 | 根据 Binder 协议进行不同处理

TIP IPC 应答数据与 REPLY Binder 协议

BR_TRANSACTION 与 BC_TRANSACTION 是 Binder IPC 数据发送端向接收端发送 IPC 数据时使用的两种协议。

IPC 数据接收端在接收到 IPC 数据后，会向 IPC 数据的发送端发送一些与 IPC 数据相关的应答数据，这些数据称为 IPC 应答数据。IPC 应答数据发送端在向接收端发送应答数据时使用 BR_REPLY 与 BC_REPLY 两种协议。关于应答处理的内容，将在 7.3 节“Android Binder Driver 分析”中进行详细讲解。

1 该部分内容将在 Binder Addressing 中进行讲解。

更多更新编程资源尽在 http://www.pin5i.com/

7.2.4 RPC 代码与 RPC 数据

一般地，当调用一个已经定义好的函数时，通常需要指出函数的名称，并传递相应的参数。同样，在通过 Binder 调用另一端的函数时，也必须指出要调用的函数名称，以及给出相应的参数。服务客户端在调用 Service Server 端某个服务中的函数时，它会把与要调用的函数相关的信息插入到 IPC 数据中，IPC 数据中的 RPC 代码用于指定待调用的函数，RPC 数据是要传入函数的参数。当服务客户端希望调用 Service Server 某个服务的函数时，必须向 Service Server 明确告知 RPC 代码。例如，有一个 MP3 应用程序希望调用 Audio Flinger 服务，调用过程如图 7-10 所示。

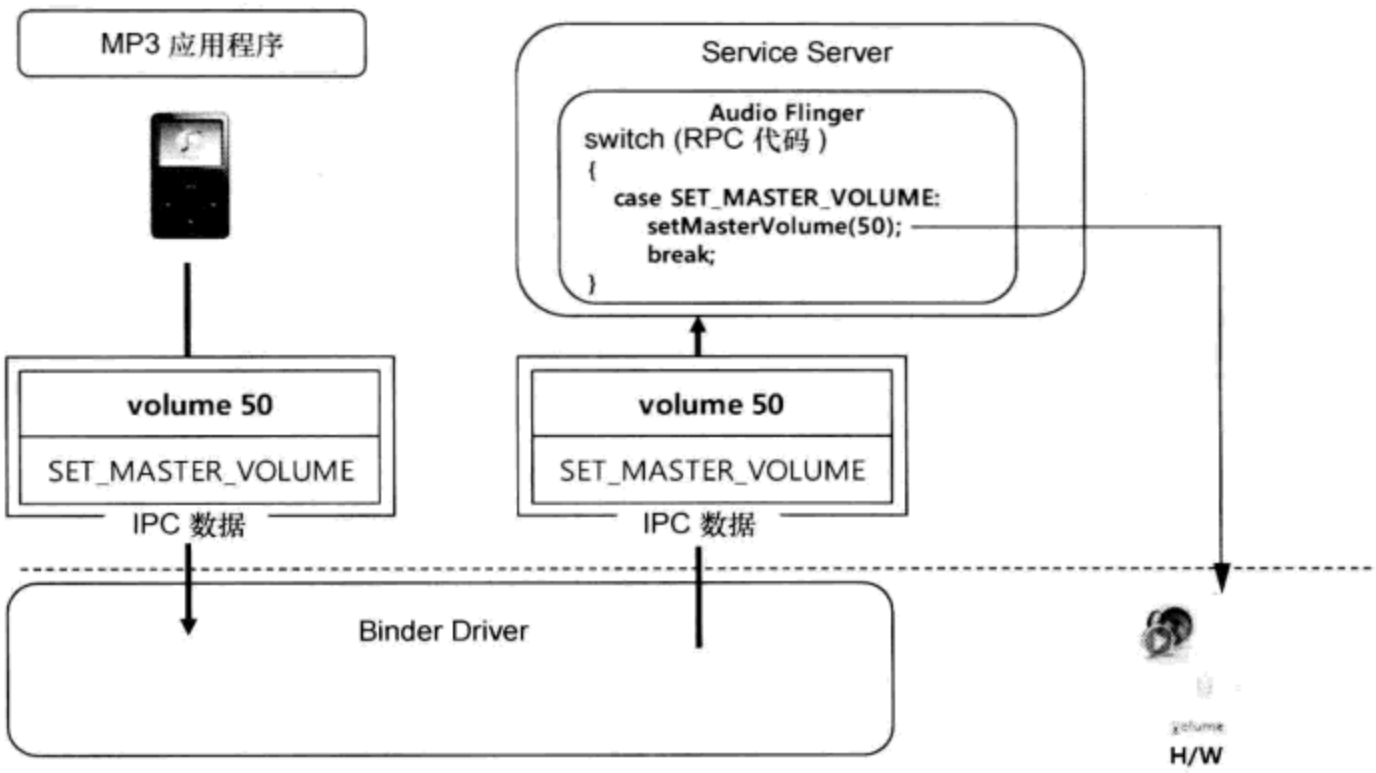


图 7-10 | RPC 代码与 RPC 数据应用实例

如图 7-10 所示，MP3 应用程序向提供 Audio Flinger 服务（用于调整硬件音频设备的音量）的 Service Server 提出请求，在播放音乐时进行音量调整。因此，它先把包含 RPC 代码（SET_MASTER_VOLUME 指定音量调整函数）与 RPC 数据（希望调整的音量大小）的 IPC 数据发送至 Service Server。Service Server 从接收到的 IPC 数据中解析出 RPC 代码为 SET_MASTER_VOLUME，然后调用 Audio Flinger 服务的 setMasterVolume()函数。经过这一系列的处理，MP3 应用程序调整音量的工作才全部完成。

7.2.5 Binder 寻址（Binder Addressing）

在 Android 系统中，有一个名称为 Context Manager 的特殊进程，它为每个服务分

更多更新编程资源尽在 http://www.pin5i.com/

配一个称为 Handle（用作 Binder IPC 的目的地址）的编号，并提供服务的添加、检索等管理功能（Context Manager 自身的 Handle 值被设置为 0）。关于 Context Manager 更多的内容，在 7.4 节“Context Manager”中将仔细进行讲解，这里只需要了解它是一种管理服务的系统进程即可。

如前所述，Binder Driver 会根据 IPC 数据中的 Handle 查找 Service Server，我们称这一过程为 Binder 寻址（Binder Addressing）。为了顺利实现 Binder 寻址，Service Server 必须先把自身服务的访问信息注册到 Context Manager。在服务注册过程中，Service Server 会向 Binder Driver 传送 IPC 数据，其中包含 RPC 代码（ADD_SERVICE）、RPC 数据（注册服务的名称），并且 Handle 值设置为 0。

如图 7-11 所示，当 Service Server 向 Context Manager 注册自身的服务时，Binder Driver 就会进行 Binder Addressing。Binder Driver 首先查找 Handle 值为 0 的 Binder 节点（Node）[1]，Handle 值为 0 的 Binder 节点是 Context Manager，Service Server 会将 IPC 数据传递给 Context Manager[2]。然后 Binder Driver 会生成一个 Binder 节点，用来表示 Service Server 中的服务 A，接下来生成 Binder 节点引用数据，以便 Context Manager 识别所生成的 Binder 节点，并将相关节点连接起来。根据生成的顺序，引用数据会被编号，这种编号将插入到 IPC 数据中，传递给 Context Manager。Context Manager 会根据 IPC 数据中的服务名称与 Binder 节点编号，将服务注册到自身的服务目录列表中。

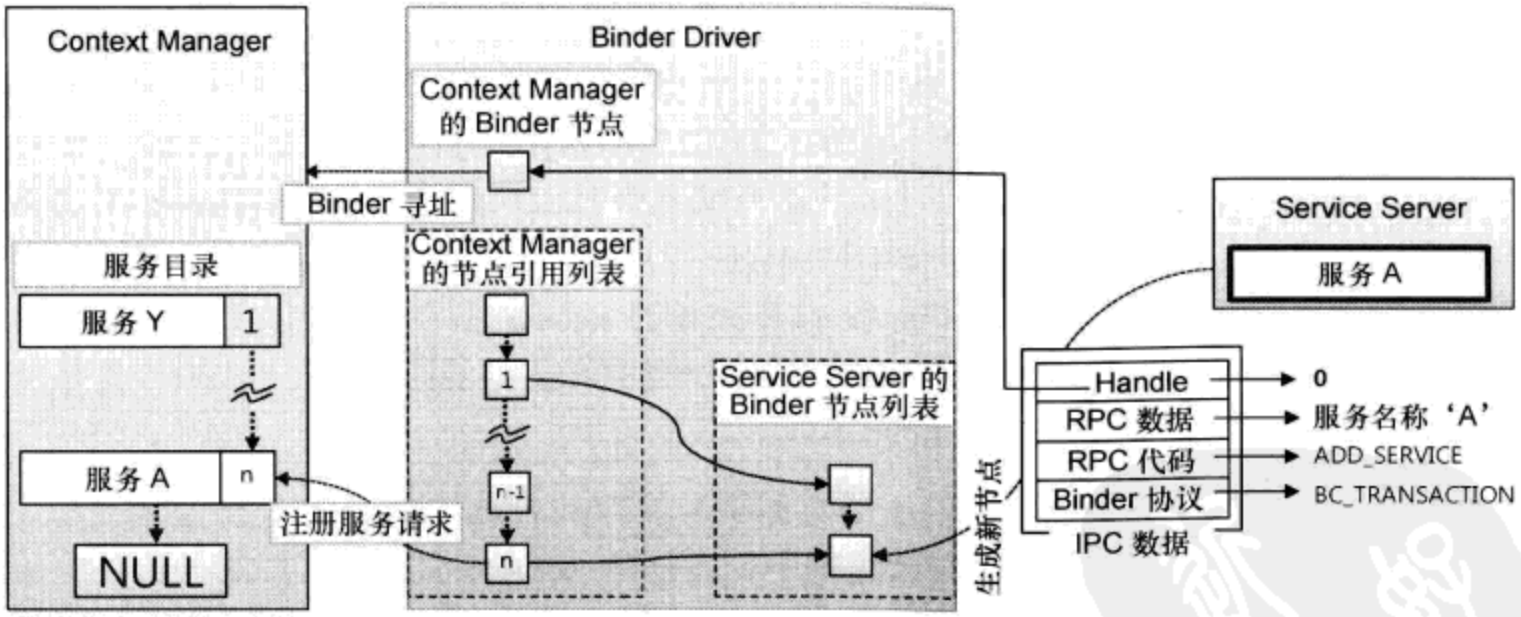


图 7-11 | 服务注册时 Binder 寻址

当服务注册在 Android 的启动阶段完成后，Context Manager 的服务目录列表、

1 Binder 节点（Node）是 Binder Driver 内部的数据结构，每个服务都对应一个 Binder 节点。并且每个进程都拥有一个保存 Binder 节点的列表。可以把 Binder 节点想象成用来查找 IPC 对象进程的标识符，更详细的内容将在 7.3 节“Android Binder Driver 分析”中讲解。

2 该过程也是 Binder Addressing。

更多更新编程资源尽在 http://www.pin5i.com/

Binder Driver 的 Binder 节点，以及 Service Server 中的服务就连接在一起了，注册在 Context Manager 中的服务就可以被其他进程使用了。

接下来，看一下服务客户端是如何进行服务检索的。如图 7-12 所示，它描述的是服务客户端在检索服务时进行 Binder 寻址的整个过程。首先，服务客户端会将包含 RPC 代码（GET_SERVICE）、RPC 数据（请求的服务名称）、值为 0 的 Handle 的 IPC 数据经由 Binder Driver 传递至 Context Manager 中。

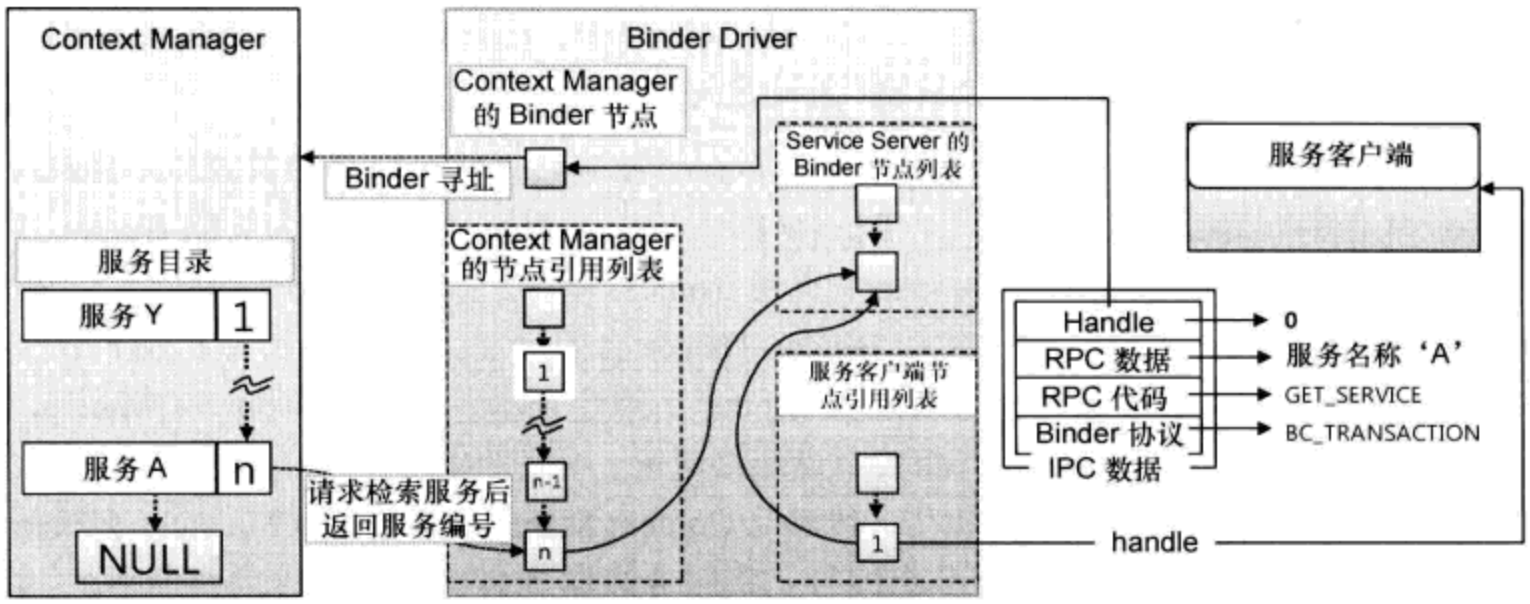


图 7-12 | 检索服务后获取服务的 Binder 节点

Context Manager 接收到 IPC 数据后，根据 IPC 数据中包含的所请求的服务名称，在自身持有的服务列表中查找相应服务的编号，将查找到的服务编号发送给 Binder Driver[1]。Binder Driver 将根据传递过来的服务编号查找对应的引用数据，然后在服务客户端生成引用数据，并将其与 Context Manager 引用数据所指的 Binder 节点连接起来。连接后的 Binder 节点与 Service Server 自身服务的 Binder 节点相对应。Binder Driver 将根据生成顺序为引用数据编号[2]，并传递给服务客户端。服务客户端将编号指定为 Handle，查找 Service Server 的 Binder 节点。

最后，服务客户端将接收到的引用数据编号保存到 Handle 中，把与服务函数相关的 RPC 代码与 RPC 数据包含进 IPC 数据中，发送给 Service Server，并调用服务 A 中的函数。

图 7-13 展现的是服务客户端通过 Binder 寻址调用服务的过程。Service Server 接收到 IPC 数据之后，会根据 IPC 数据中包含的 RPC 代码、RPC 数据进行相应的处理。

---

1 该过程包含接收 IPC 数据的进程应答，即使用名为 BC_REPLY 的 Binder 协议，传递 IPC 应答数据。

2 今后为了便于说明，将 Binder Driver 内引用数据的编号称为 Binder 节点编号。在 Service Server 与服务客户端，Binder 节点编号被用作 Handle。

更多更新编程资源尽在 http://www.pin5i.com/

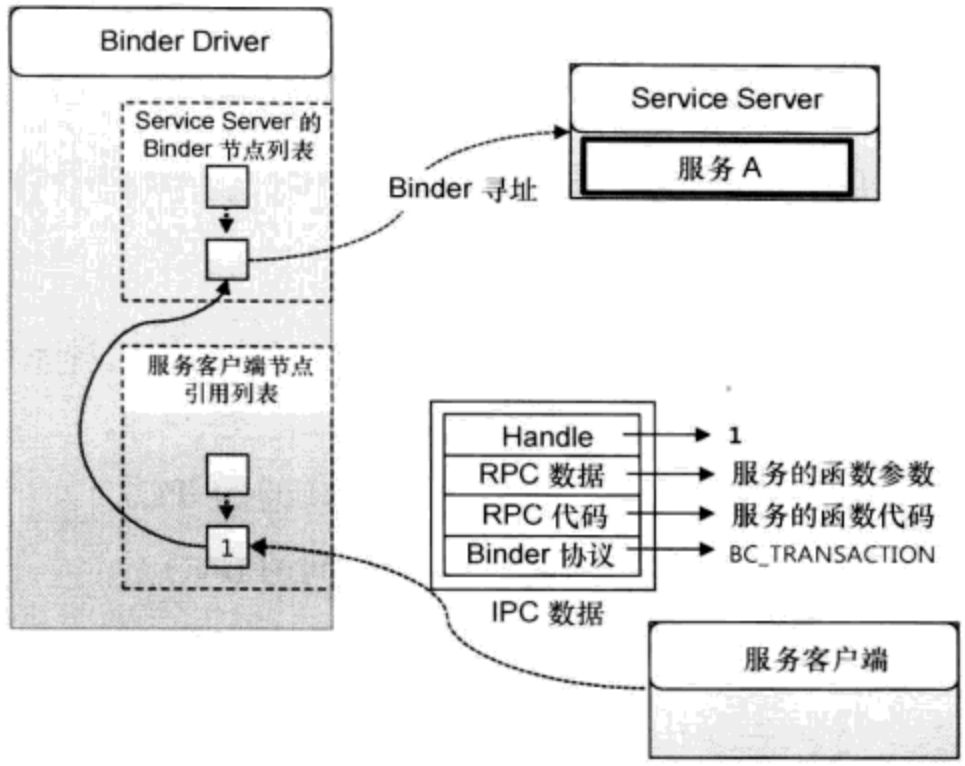


图 7-13 | 调用服务时传送 IPC 数据

本节在讲解 Binder 寻址时使用的示意图都比较简单，但实际的实现代码是相当复杂的，从 Binder Driver 的具体代码可以证实这一点。更多相关内容，将在 7.3.3“Binder Driver 函数分析”一节中讲解。

如前所述，Android 的 Binder 机制较为复杂，它不是单纯的 IPC 工具，讲解时还涉及到使用 Binder 的进程、内核级别的行为等与整个系统相关的内容。如果读者已经阅读完 7.2 节“Android Binder 模型”，那么相信读者对在 Android 系统中如何调用服务有了一定程度的了解。但仍然会有一些读者还是很迷惑，例如 Binder 寻址究竟是如何实现的，IPC 数据是如何复制到指定对象的，诸如此类的问题仍然有待进一步说明。要弄清这些问题，就要认真学习 Binder 的核心部分，即 Binder Driver。在接下来的一节中，将具体分析 Binder Driver 的代码，进一步了解其运行机制。

7.3 Android Binder Driver 分析

Binder Driver 是整个 Binder 的核心，本节将分析 Binder Driver，以便读者进一步了解它。我们把内容的重点放在 IPC 上，对远程调用 RPC 不作过多的讲解。服务客户端在调用 Service Server 的服务时，整个 IPC 过程可以分为服务注册、服务检索、服务使用 3 个阶段，以下将通过 3 个小节来说明这一过程。

在 7.3.1“从进程的角度看服务的使用”一节中，我们将会学习 Service Server、服务客户端（Service Client）、Context Manager 是如何访问 Binder Driver 的，以及服务使用的整个过程是怎样的。在 7.3.2“从 Binder Driver 的角度看服务的使用”一节中，我

更多更新编程资源尽在 http://www.pin5i.com/

们将学习在以上各进程的请求下 Binder Driver 是如何动作的。在 7.3.3“Binder Driver 函数分析”一节中，我们将分析 Binder Driver 的源代码，进一步解释 7.3.2“从 Binder Driver 的角度看服务的使用”一节中的 Binder Driver 的行为。

7.3.1 从进程的角度看服务的使用

服务客户端在使用服务时需经历以下三个阶段，无论使用何种系统服务都要经历这三个阶段。

1. 服务注册（Service Server 与 Context Manager 间的 IPC）
2. 服务检索（服务客户端与 Context Manager 间的 IPC）
3. 服务使用（服务客户端与 Service Server 间的 IPC）

服务注册

服务注册是指将 Service Server 中的服务注册到 Context Manager 的服务目录中，其中涉及到 Service Server 与 Context Manager 两个进程，它们的作用如表 7-3 所示。

表 7-3 在 Binder IPC 过程中各进程的作用（服务注册阶段）

IPC 主体	作用	
	IPC 数据	IPC 应答数据
Context Manager	接收	发送
Service Server	发送	接收

Service Server 生成用于调用 Context Manager 服务注册函数的 IPC 数据，而后将其传递给 Binder Driver。Binder Driver 将 IPC 数据传递给 Context Manager，Context Manager 接收 IPC 数据后，调用 RPC 代码指定的函数。图 7-14 描述了 Context Manager 与 Service Server 相互作用的过程。在这一过程中，Service Server 是调用 Context Manager 服务函数的客户端，而 Context Manager 则是提供服务的一端（Context Manager 是提供注册服务的 Server）。

（1）～（3） 首先 Context Manager 调用 open()函数，打开 Binder Driver，而后调用 mmap()函数，在内核空间中开辟一块用于接收 IPC 数据的 Buffer[1]，再调用 ioctl()函数进入待机状态，等待接收 IPC 数据。

（4）～（5） 为了注册服务，Service Server 先打开 Binder Driver，而后调用 mmap()函数，确定一块 Buffer，用于接收 IPC 应答数据。

---

1 为了方便说明，我们把经由 mmap 空间移动的数据称为 IPC 数据，事实上，移动的只是 IPC 数据中的 RPC 数据。IPC 数据中的 RPC 代码及 Binder 协议由 copy_to_user()函数复制，并发送到用户空间。

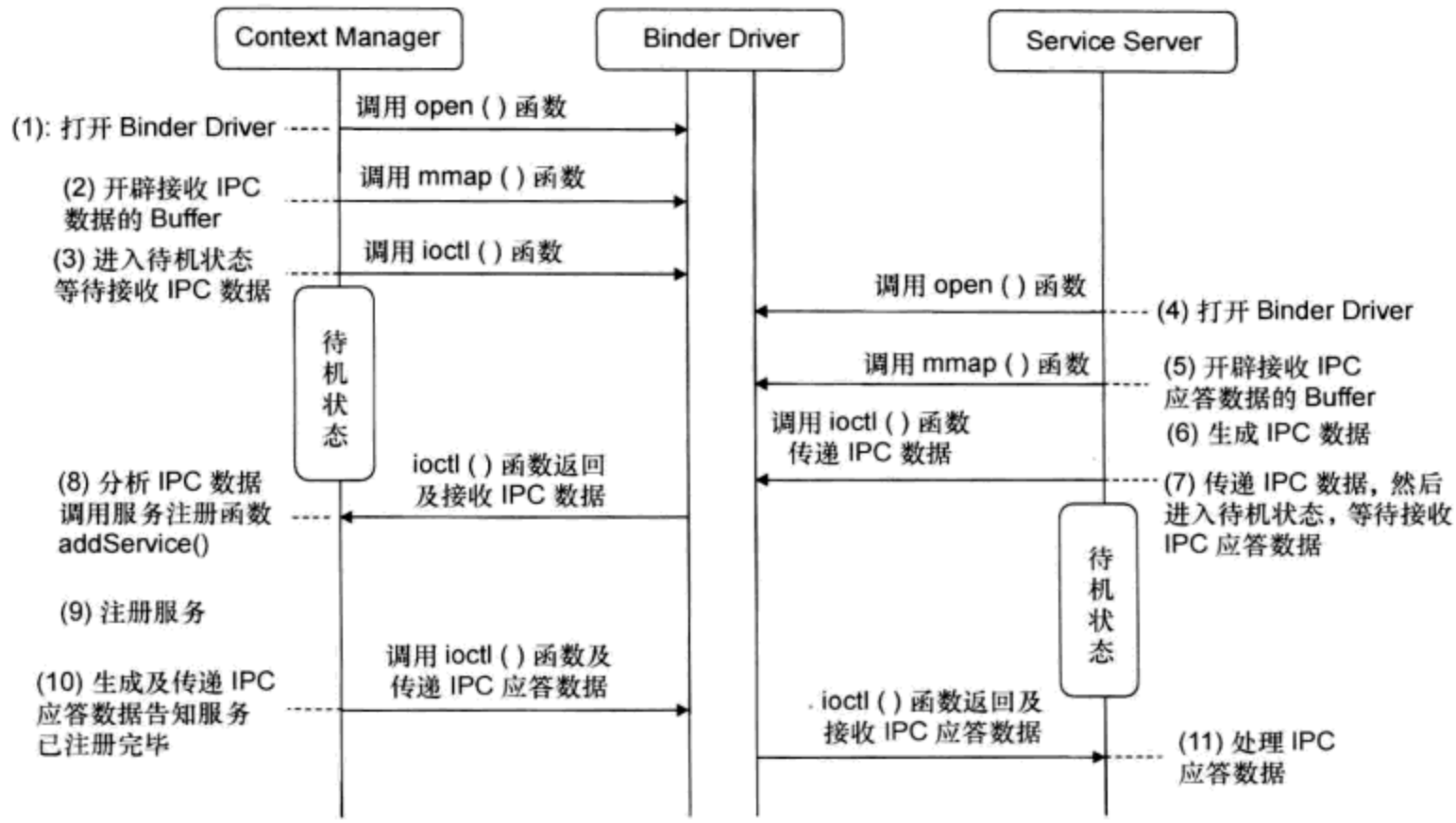


图 7-14 | 服务注册（从进程的角度看）

（6） Service Server 生成 IPC 数据，IPC 数据包含 RPC 数据（保存要注册的服务名称）、RPC 代码（Context Manager 的注册函数 ADD_SERVICE）、Handle（Context Manager 的 Handle 值为 0）三部分。

（7） Service Server 调用 ioctl()函数向 Binder Driver 传递 IPC 数据。而后 Binder Driver 将来自 Service Server 的 IPC 数据传递给 Context Manager（Handle 值为 0 的进程）。

（8）～（9） Context Manager 分析 IPC 数据中的 RPC 代码，并根据 RPC 代码调用相应的服务注册函数，而后使用 RPC 数据中的服务名称，将指定的服务注册到服务目录中。

（10） 服务注册完成后，Context Manager 会生成 IPC 应答数据，并传递给 Service Server，告知服务已经正常注册，完成 Binder IPC 循环。

当服务注册完成后，服务就处于可使用状态，服务客户端就可以使用该服务了。在下一节中，我们将学习服务客户端如何检索已注册的服务。

服务检索

服务客户端在使用 Service Server 的服务时，会向 Context Manager 请求服务的编号，这一过程称为服务检索。在服务检索阶段，涉及两个进程，它们的作用如表 7-4 所示。

更多更新编程资源尽在 http://www.pin5i.com/

表 7-4　在 Binder IPC 过程中各进程的作用（服务检索阶段）

IPC 主体	作用	
	IPC 数据	IPC 应答数据
Context Manager	接收	发送
服务客户端	发送	接收

在服务检索阶段，服务客户端会调用 Context Manager 的服务检索函数。服务客户端将首先生成调用 Context Manager 函数的 IPC 数据，将其传递给 Binder Driver。然后 Binder Driver 将 IPC 数据传递给 Context Manager，Context Manager 根据接收的 IPC 数据中的 RPC 代码调用相应的函数。

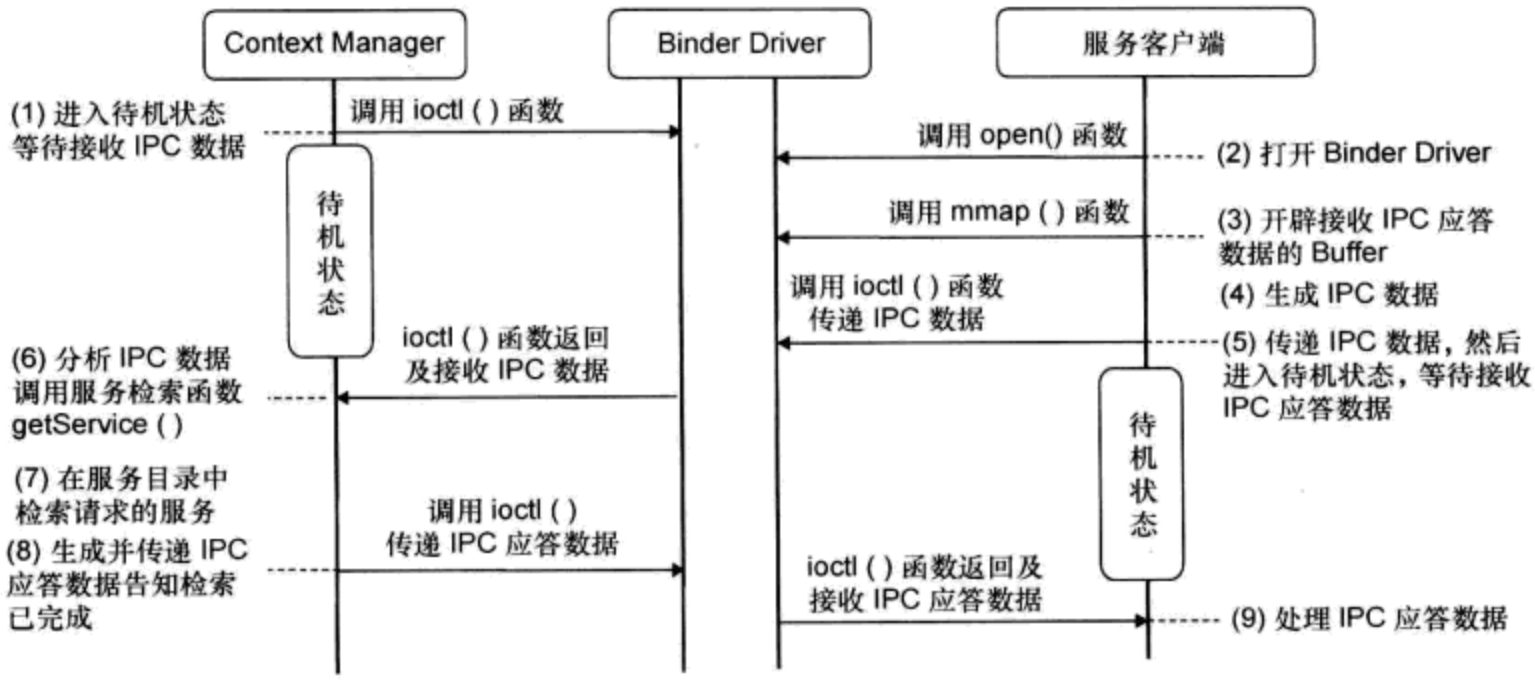


图 7-15 | 服务检索（从进程的角度看）

（1） 服务注册完成后，Context Manager 进入待机状态，等待接收 IPC 数据。

（2）～（4） 为了向 Context Manager 传递 IPC 数据，服务客户端打开 Binder Driver，而后调用 mmap()函数准备一块 Buffer，用于接收 IPC 应答数据。服务客户端生成 IPC 数据，IPC 数据包含 RPC 数据（保存要使用的服务名称）、RPC 代码（Context Manager 的服务检索函数 GET_SERVICE）、Handle（Context Manager 的 Handle 值为 0）三部分。

（5） 服务客户端调用 ioctl()函数向 Binder Driver 传递 IPC 数据。Binder Driver 根据 IPC 数据中的 Handle（Handle 值为 0），将 IPC 数据传递给 Context Manager。

（6）～（8） Context Manager 分析接收到的 IPC 数据，根据 RPC 代码，调用服务检索函数。相关函数根据 RPC 数据中的服务名称在服务目录中检索服务，

更多更新编程资源尽在 http://www.pin5i.com/

查找到服务 Binder 节点编号。Context Manager 将查找到的 Binder 节点编号插入到 IPC 应答数据中，传递给服务客户端。

（9） 服务客户端接收 IPC 应答数据，获得指定服务的 Binder 节点编号。

以上过程结束后，服务客户端即做好了使用服务的准备。在下一节中，我们将分析服务客户端实际调用服务函数的过程。

服务使用

在服务的使用阶段，服务客户端将实际使用 Service Server 的服务。在此过程中，涉及到两个进程，它们的作用如表 7-5 所示。

表 7-5　在 Binder IPC 过程中各进程的作用（服务使用阶段）

IPC 主体	作用	
	IPC 数据	IPC 应答数据
Service Server	接收	发送
服务客户端	发送	接收

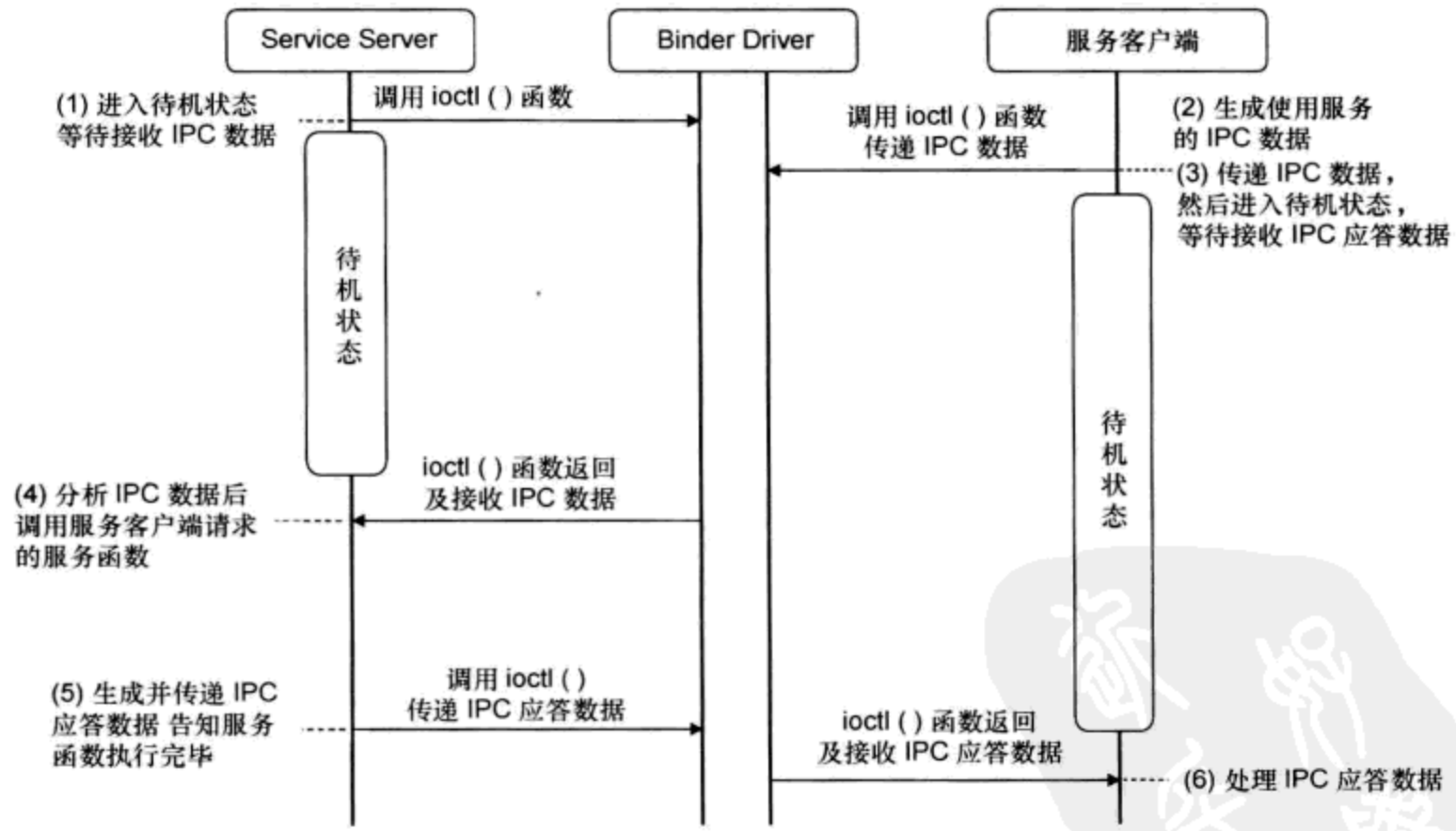


图 7-16 | 服务使用（从进程的角度看）

（1） 如 Context Manager 一样，Service Server 也会进入待机状态，等待接收 IPC 数据。并且当处理完接收的 IPC 数据后，它将再次进入待机状态，等待接收新的 IPC 数据。

更多更新编程资源尽在 http://www.pin5i.com/

（2）～（3） 为了使用服务，服务客户端首先生成 IPC 数据，IPC 数据由 RPC 代码（指定服务函数）、RPC 数据（服务函数的参数）、Handle（指定哪个服务）组成。然后调用 ioctl()函数，将 IPC 数据传递给 Binder Driver。最后 Binder Driver 根据 IPC 数据中的 Handle 把 IPC 数据传递给 Service Server。

（4）～（5） Service Server 分析 IPC 数据中的 RPC 代码，根据 RPC 代码调用服务函数。IPC 数据中包含的 RPC 数据类似于函数的参数，在函数内部使用。当指定的服务函数执行完毕后，Service Server 会生成 IPC 应答数据，其中包含着 Binder 节点编号，而后将 IPC 应答数据发送到服务客户端，告知指定的服务函数已经执行完毕。

（6） 服务客户端接收 IPC 应答数据，并进行处理。

以上从进程的角度分析了在使用 Android 服务时，服务客户端、Context Manager、Service Server 之间是如何相互作用的。如果从进程的角度上理解了各进程间的 IPC 机制，那么从 Binder Driver 角度理解 IPC 就会容易得多。下一节，将从 Binder Driver 的角度分析使用服务的过程。

7.3.2 从 Binder Driver 角度看服务的使用

服务客户端在使用服务时需要经过 Binder Driver，在这一过程中，Binder Driver 是如何运行的呢？本节将分析 Binder Driver 的整个运行过程，在 7.3.3 “Binder Driver 函数分析”一节中，将对 Binder Driver 的源代码进行分析。

服务注册

Context Manager 先于其他所有 Service Server 运行，它最先使用 Binder Driver，进入待机状态，等待 Service Server 的服务注册请求或服务客户端的服务检索请求。

在服务注册阶段，Binder Driver 将执行注册 Service Server 服务所需要的动作，图 7-17 展示了注册服务时 Binder Driver 的行为动作。

（1）～（3） 描述了 Context Manager 在进入待机状态（等待接收 IPC 数据）前使用 Binder Driver 的整个过程。Binder Driver 拥有许多结构体，这些结构体是执行 Binder IPC 所需要的。binder_proc 是其中一个结构体，该结构体中包含一系列的指针，以便访问 IPC 动作所需要的其他结构体。并且每个进程都有一个 binder_proc 结构体，通过该结构体，可以查找到另一个进程，也可以查找到进程接收 IPC 数据的 Buffer。

（1） Context Manager 通过 open()函数调用 binder_open()函数，Binder Driver 使用该函数为 Context Manager 生成并初始化 binder_proc 结构体。

更多更新编程资源尽在 http://www.pin5i.com/

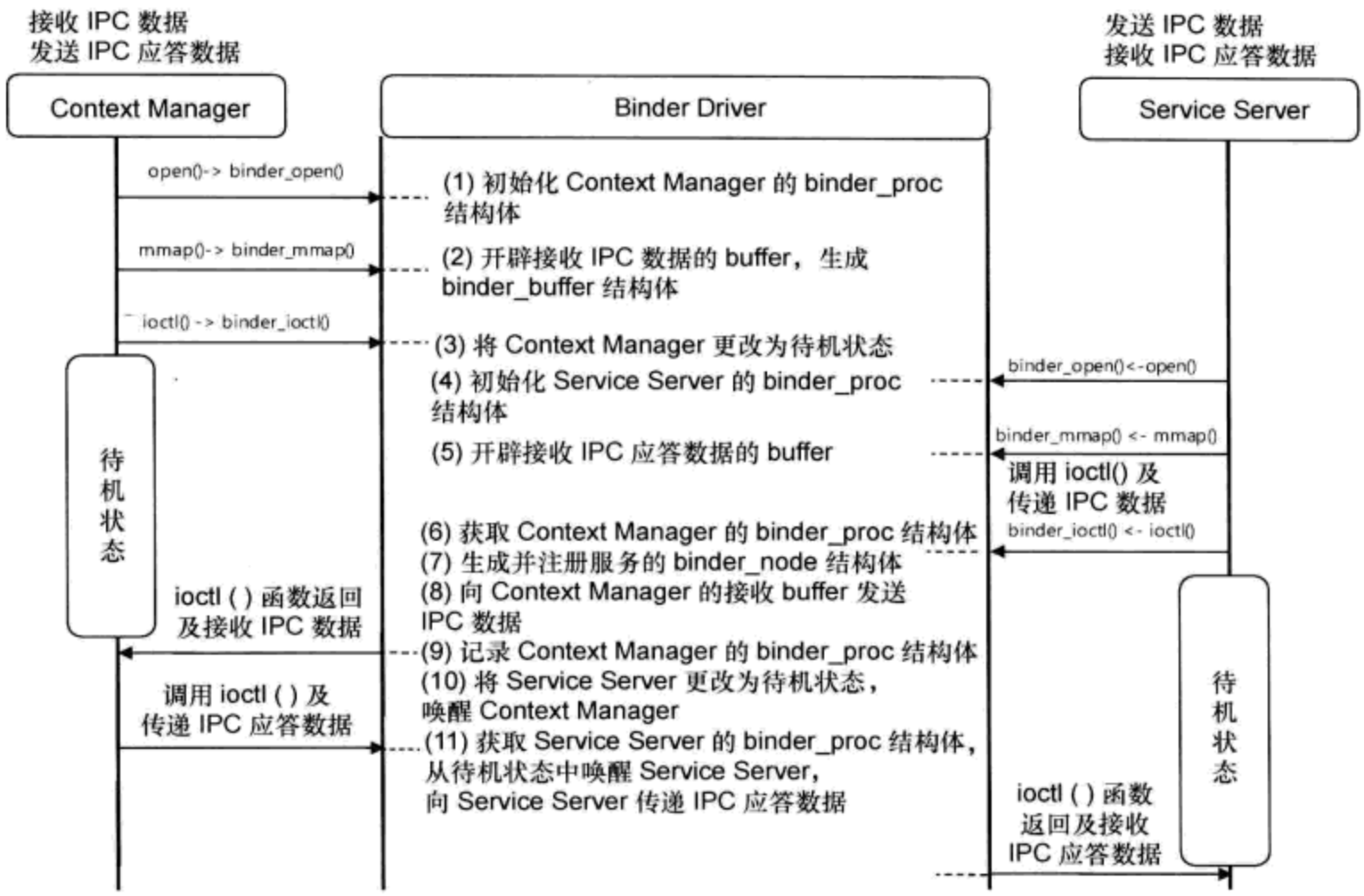


图 7-17 | 注册服务时 Binder Driver 的动作行为

（2） Context Manager 在内核空间中开辟一块 Buffer，用于接收 IPC 数据。Context Manager 通过 mmap()函数调用 Binder Driver 的 binder_mmap()函数。Binder Driver 使用 binder_mmap()函数在内核空间中分配一块用于接收 IPC 数据的 Buffer，其大小为调用 mmap()函数时指定的大小，并被保存到 binder_buffer 结构体中。而后 binder_buffer 结构体被注册到 binder_proc 结构体中。

（3） Context Manager 通过 ioctl()调用 Binder Driver 的 binder_ioctl()函数。在 binder_ioctl()函数中，Context Manager 将处于待机状态，直到另一个进程（在服务注册阶段是指 Service Server）向其发送 IPC 数据。

（4） Service Server 通过 open()调用 Binder Driver 的 binder_open()函数，Binder Driver 为 Service Server 生成 binder_proc 结构体，并将其初始化。

（5） Service Server 通过 mmap()调用 Binder Driver 的 binder_map()函数。Binder Driver 开辟一块用于接收 IPC 应答数据的 Buffer，并将其保存到 binder_buffer 结构体中。而后 Service Server 生成 IPC 数据，用来调用 Context Manager 的服务注册函数。

（6） 为了注册服务，Service Server 将 IPC 数据传递给 Binder Driver。Binder Driver

更多更新编程资源尽在 http://www.pin5i.com/

将根据 IPC 数据中的 Handle 查找到 Context Manager 的 binder_node 结构体[1]，以及 binder_proc 结构体[2]。

（7） 为 Service Server 要注册的服务，生成 binder_node 结构体。Binder Driver 将 binder_node 分别注册到 Service Server 的 binder_proc 结构体以及 Context Manager 的 binder_proc 结构体之中。

（8） Binder Driver 将在（6）中查找到的 Context Manager 的 binder_proc 结构体中查找注册在 binder_buffer 结构体中的 Buffer，并传送 IPC 数据。

（9） Binder Driver 会记下 IPC 数据发送进程（Service Server）的 binder_proc 结构体，以便在 Context Manager 向 Service Server 发送应答数据时查找 Service Server 的 binder_proc 结构体。

（10）Binder Driver 让 Service Server 处于待机状态，并将 Context Manager 从待机状态中唤醒，传递 IPC 数据。从待机状态中唤醒的 Context Manager 经由 Binder Driver 接收来自 Service Server 的 IPC 数据。Context Manager 根据接收的 IPC 数据注册指定的服务，而后生成 IPC 应答数据（通知服务注册完成），并将其传递给 Binder Driver。

（11）Binder Driver 会在（9）中的 Service Server 的 binder_proc 结构体中查找注册在 binder_buffer 结构体中的 Buffer，并传送 IPC 应答数据，唤醒接收端的 Service Server。Service Server 被唤醒后，接收 IPC 应答数据，并根据接收的 IPC 应答数据进行相应处理。

服务检索

服务检索是指服务客户端向 Context Manager 请求指定服务的 Handle 的过程。服务客户端把包含服务请求信息的 IPC 数据发送给 Context Manager，而后 Context Manager 通过 IPC 应答数据将指定服务的 Handle 发送给服务客户端。Binder Driver 将把服务客户端请求的服务的 binder_node 结构体注册到服务客户端的 binder_proc 结构体中。图 7-18 描述了服务检索时 Binder Driver 的行为动作。

（1） Context Manager 进入待机状态，等待接收 IPC 数据。

（2）～（3） 服务客户端生成并初始化 binder_proc 结构体，而后开辟一块 Buffer

---

1 binder_node 结构体是用来代表服务的数据结构。Service Server 拥有多个 binder_node 结构体，数目等同于 binder_proc 中服务的数量。binder_node 结构体拥有 binder_proc 的指针，用于查找 binder_proc 结构体。

2 与拥有多个服务的 Service Server 不同，Context Manager 自身即是一种服务，它拥有一个指向自身的 binder_node 结构体。一般地，Service Server 在向 Context Manager 请求服务注册时才会生成 binder_node 结构体，但 Context Manager 会直接访问 Binder Driver，生成 binder_node，并且该 binder_node 的编号被设置为 0。客户端通过值为 0 的 Handle，即可查找到 Context Manager。

更多更新编程资源尽在 http://www.pin5i.com/

用于接收 IPC 应答数据。

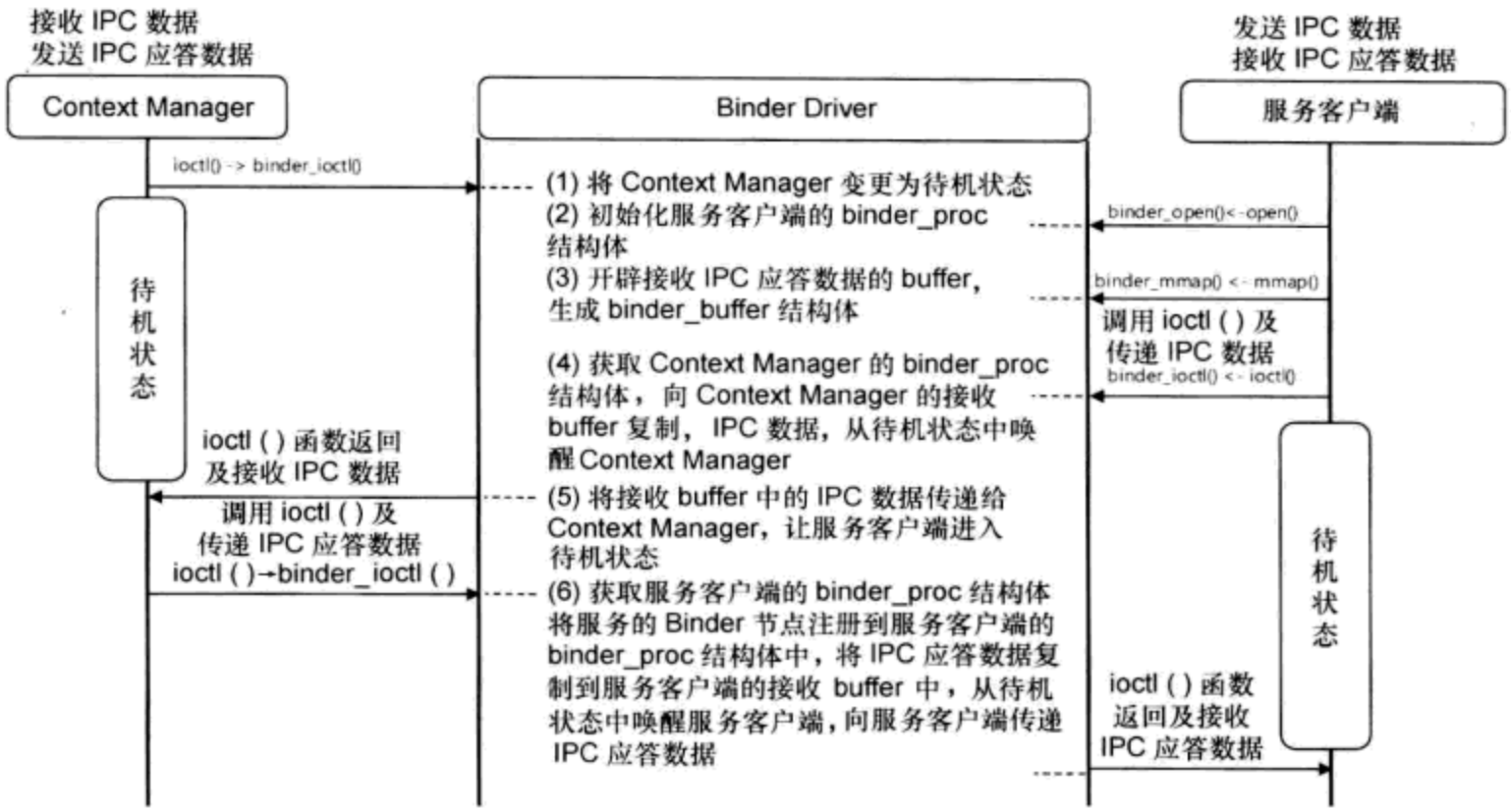


图 7-18 | 服务检索时 Binder Driver 的行为

（4） 服务客户端通过 ioctl()调用 Binder Driver 的 binder_ioctl()函数。与注册服务时一样，Binder Driver 会查找 Context Manager 的 binder_proc 结构体。

但与注册服务不同的是，它并不生成 binder_node 结构体，只是将 IPC 数据拷贝到 Context Manager 的接收 Buffer 中。并且记下服务客户端的 binder_proc 结构体，以便查找 IPC 应答数据的接收端。

（5） Binder Driver 让服务客户端处于待机状态，并唤醒处于待机状态的 Context Manager，而后接收 IPC 数据。从待机状态苏醒的 Context Manager 在服务目录中查找请求的服务，把服务的编号插入 IPC 应答数据中，并将 IPC 应答数据传递给 Binder Driver。

（6） Binder Driver 根据接收到的服务编号查找相应的 binder_node 结构体，而后将查找到的 binder_node 结构体注册到服务客户端的 binder_proc（在（4）中已被记下）中。binder_node 结构体用于在服务使用阶段查找 Service Server 的 binder_proc 结构体。

（7） Binder Driver 将（6）中注册的 binder_node 结构体的编号插入到 IPC 应答数据中，传递给服务客户端，唤醒服务客户端。在使用服务时，服务客户端会将编号作为 Handle 使用。

更多更新编程资源尽在 http://www.pin5i.com/

服务使用

在服务检索阶段，Binder Driver 的行为与服务注册阶段的行为类似，但不同之处在于它将 IPC 数据传递给了拥有实际服务的 Service Server，而不是传递给 Context Manager。经过服务检索，服务客户端就获得了 Service Server 所拥有服务的 Binder 节点编号，在生成 IPC 数据时，它就可以把要使用的服务的 Binder 节点编号设置到 IPC 数据的 Handle 中，取代原来值为 0 的 Handle。

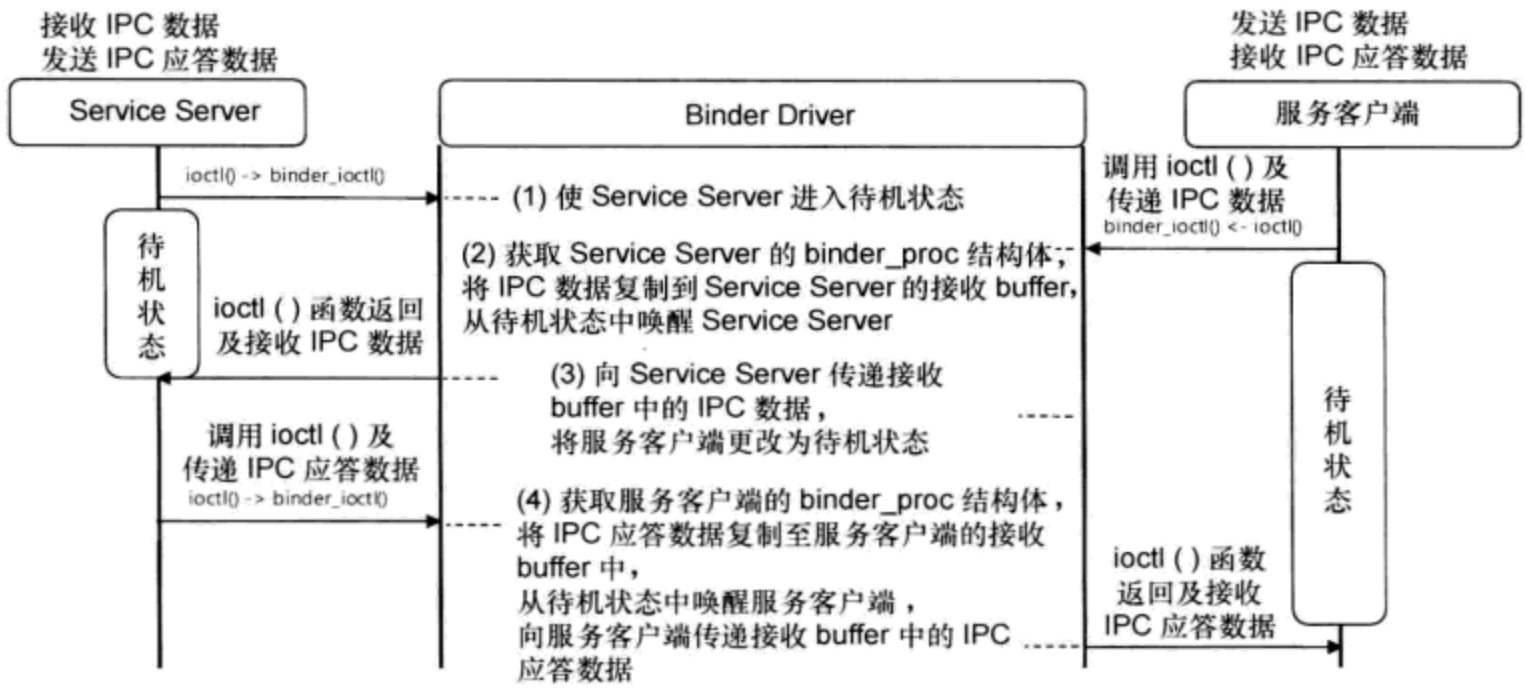


图 7-19 | 服务使用时 Binder Driver 的行为

（1） Service Server 处于待机状态，等待接收 IPC 数据。

（2） 服务客户端将从服务检索获取的 Binder 节点编号保存到 IPC 数据的 Handle 中，生成 IPC 数据后传递给 Binder Driver。Binder Driver 根据 IPC 数据中的 Handle 查找相应的 binder_node 结构体，并据此查找到注册有 binder_node 结构体的 Service Server 的 binder_proc 结构体。然后，通过注册在 binder_proc 结构体中的 binder_buffer 结构体，将 IPC 数据拷贝到 Buffer 中。

（3） Binder Driver 让服务客户端进入待机状态，唤醒处于待机状态的 Service Server。Service Server 从待机状态苏醒后接收 IPC 数据，并通过 IPC 数据中的 RPC 代码、RPC 数据调用相应的服务函数。

（4） 在服务函数执行完毕后，Service Server 会生成 IPC 应答数据，并将其传递给 Binder Driver。Binder Driver 执行的动作与服务注册阶段的（10）相同，将 IPC 应答数据传递给服务客户端。至此，服务客户端调用服务的整个 Binder IPC 过程就完成了。

更多更新编程资源尽在 http://www.pin5i.com/

7.3.3 Binder Driver 函数分析

从本节开始，将分析 Binder Driver 的主要函数，通过分析源代码，进一步了解 Binder Driver 的动作行为，加深对 7.3.2“从 Binder Driver 角度看服务的使用”一节内容的理解。笔者使用的 Android 内核版本为 2.6.32，对 Binder Driver 的分析也基于该版本。Binder Driver 的源代码也可以直接在网页的 Android 项目[1]中进行查看。由于源代码非常多，本节无法列出所有代码，仅列出对理解 Binder Driver 动作行为最重要的部分。为了便于理解，希望各位在阅读本节内容时，参考 Binder Driver 的整个源代码。

binder_open()函数

进程在使用 Binder Driver 时首先调用 binder_open()函数，binder_open()函数与系统函数 open()函数连接在一起。当进程调用 open()系统函数时，Binder Driver 的节点文件“/dev/binder”将作为参数传给所调用的函数，Binder Driver 的文件描述符作为返回值返回给进程。在调用 mmap()与 ioctl()函数时，Binder Driver 的文件描述符将作为参数传递给函数。

> **TIP 生成 Binder Driver 节点文件**
>
> “/dev/binder”设备节点由 Binder 设备驱动源代码 binder.c 的 binder_init()函数中的 misc_register()函数创建，相关内容在第 3 章“init 进程”中讲解过。

binder_open()函数将为打开 Driver 的进程生成并初始化 binder_proc 结构体。并且初始化待机队列，用来将进程切换到待机状态下。图 7-20 描述了 binder_open()函数的功能，如图 7-20 所示。

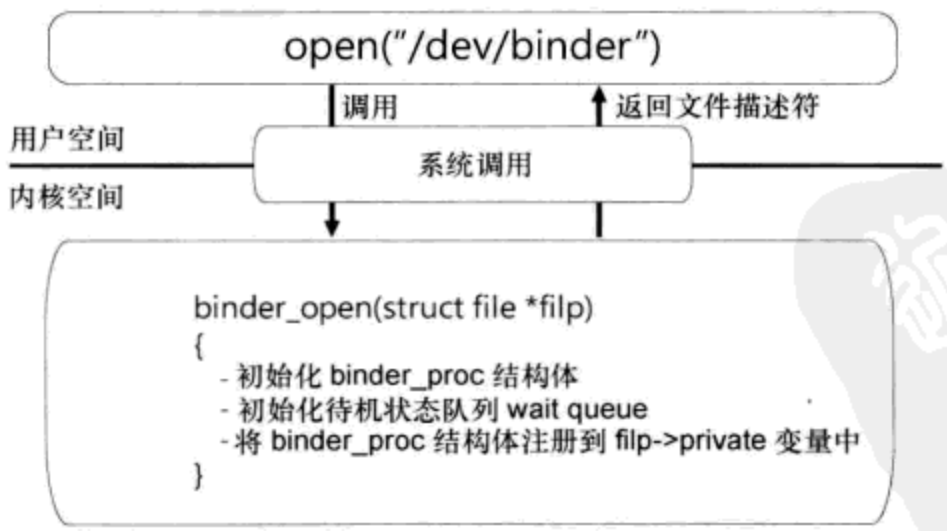


图 7-20 | binder_open()函数的功能

1 在 http://android.git.kernel.org/?p=kernel/common.git;a=summary 网站中，通过 tag 可以按照不同的内核版本查看，Binder Driver 源码在 drivers/staging/android/binder.c 文件中。

更多更新编程资源尽在 http://www.pin5i.com/

在分析 binder_open()函数之前，先分析 binder_proc 结构体，它在 Binder IPC 中起着非常重要的作用。binder_proc 结构体用来管理 Binder IPC 所需要的各种信息，包括打开 Driver 的进程的信息、接收 IPC 数据的 Buffer 信息、Binder IPC 的状态信息等。此外，binder_proc 结构体拥有 Binder 中其他结构体的指针，为访问其他结构体提供了方便，被称为 Binder Driver 的根结构体。binder_proc 结构体中主要的成员变量及其意义如表 7-6 所示。

表 7-6 binder_proc 结构体

成员变量	意义
struct rb_root threads	拥有 binder_thread 结构体的红黑树的根
struct rb_root nodes	带有 binder_node 结构体的红黑树的根
struct rb_root refs_by_desc	带有 binder_ref 结构体*的红黑树的根，使用 desc 号码区分各个 binder_ref 结构体。
struct rb_root ref_by_node	带有 binder_ref 结构体的红黑树的根，使用 binder_node 结构体区分 binder_ref 结构体。
int pid	创建 binder_proc 结构体的进程的 pid
struct vm_area_struct *vma	调用 binder_mmap()函数的进程的用户空间信息（用来保证内核空间内存）
struct task_struct *tsk	指生成 binder_proc 结构体的进程的 task_struct 结构体
void *buffer	接收 IPC 数据的 binder_buffer 结构体的指针
size_t user_buffer_offset	映射为接收 IPC 数据的 Buffer 的内核空间与用户空间的地址偏移量，用来将接收到的 IPC 数据传递到用户空间
struct list_head buffers	为接收 IPC 数据而分配的 binder_buffer 结构体列表
struct rb_root free_buffers	接收 IPC 数据后，要释放的 binder_buffer 结构体的列表
size_t buffer_size	进程在内核空间开辟的 Buffer 大小
struct list_head todo	进程从待机状态唤醒后要做的事情
wait_queue_head_t wait	用来让进程进入待机状态

*该结构体对应于 7.2.5“Binder 寻址”一节中讲解的 Binder 节点引用数据。

下面通过代码 7-1 分析一下 binder_open()函数。

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);
    proc->tsk = current;                    ❶
    INIT_LIST_HEAD(&proc->todo);            ❷
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    mutex_lock(&binder_lock);
```

更多更新编程资源尽在 http://www.pin5i.com/


```
    binder_stats.obj_created[BINDER_STAT_PROC]++;
    hlist_add_head(&proc->proc_node, &binder_procs);
    proc->pid = current->group_leader->pid;
    INIT_LIST_HEAD(&proc->delivered_death);
    filp->private_data = proc;                          ❸
    mutex_unlock(&binder_lock);
    if (binder_proc_dir_entry_proc) {;                  ❹
    snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
    create_proc_read_entry(strbuf,
      ➥ S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc);
    return 0;
}
```

代码 7-1 | binder_open()函数

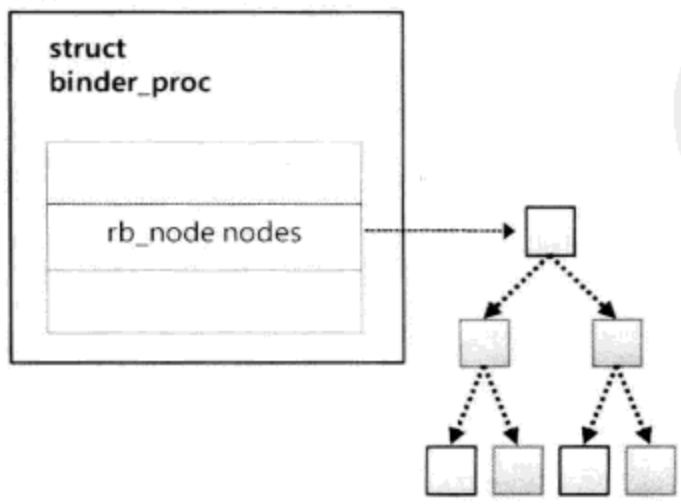
❶ task_struct 结构体中包含打开 Binder Driver 的进程的信息，该语句将 task_struct 结构体变量 current 注册到 binder_proc 结构体的 tsk 成员变量中。该变量用来输出当前正在进行 Binder IPC 的进程。

❷ Binder Driver 在接收 Binder IPC 数据后，将要执行的任务保存到 proc->todo 列表中，第一条语句用来初始化 proc->todo 列表，第二条语句初始化待机队列，以便将打开 Binder Driver 的进程切换到待机状态。

❸ 该语句将 binder_proc 结构体变量 proc 注册到 file 结构体的 private_data 成员变量中。filp 是 file 结构体变量，在调用 binder_open()时将作为参数传入。该变量也会用在 binder_mmap()与 binder_ioctl()函数中，引用前只需用“struct binder_proc proc=filp->private_data”语句，即可获取 binder_proc 结构体。

TIP 红黑树（Red-Black Tree）

在 Binder Driver 中存在多种结构体，它们大部分都是由红黑树来管理的，在进行 Binder IPC 时可以有效地减少检索结构体的时间。红黑树的定义与 API 在内核的 include/linux/rbtree.h 文件中。



在讲解结构体时，将按照上图所示的形态讲解由红黑树管理的结构体，请注意这点。

更多更新编程资源尽在 http://www.pin5i.com/

❹ binder_open()函数将在/proc/binder/proc 目录下生成文件，以便初始化 binder_proc 结构体或显示与 Binder IPC 相关的信息。当 Android 启动后，查看系统/proc/binder 目录以及子目录，将看到使用 Binder IPC 的进程的信息。

binder_mmap()函数分析

在 7.1“Linux 内存空间与 Binder Driver”一节中，已经讲解过进程的用户空间是不能共享的。但进程之间可以通过内核空间来相互交换数据，并且执行 Binder IPC 的进程通过 Binder Driver 在内核空间中开辟共享空间。进程将调用 binder_mmap()函数在内核空间中开辟一块 Buffer，用来接收 IPC 数据与 IPC 应答数据。

在分析 binder_mmap()函数之前，先看一个调用 mmap()函数的实例。如代码 7-2 所示，使用 Binder Driver 的进程调用 mmap()函数在内核空间中开辟一块 Buffer，用来接收 IPC 数据。

```
struct binder_state *binder_open(unsigned mapsize)
{
    bs->fd = open("/dev/binder", O_RDWR);
    bs->mapsize = mapsize;                          // 此处 mapsize 为 128KB
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
}
```

代码 7-2 | 调用 mmap()函数[1]

如代码所示，进程调用 mmap()函数在内核空间中映射出一块用来接收 IPC 数据的 Buffer，其大小为 mapsize。mmap()函数是一个映射函数，它将用户空间的特定区域映射到内核空间的特定区域中。由于用户空间中的进程不能直接访问内核空间，所以只能通过内核空间的特定映射区域来访问内核空间。当调用 mmap()函数时，将从 0x40000000 地址开始开辟一块指定大小的空间，而后调用内核的 binder_mmap()函数。

在 Android 系统中，内核空间以及由 mmap()函数映射出的区域都事先被定义好，Android 系统采用 Prelinked[2]方式预先确定各个库将被连接的地址。这些连接信息在 Android 平台代码的/build/core/prelink-linux-arm.map 中可以查看到。

```
# 0xC0000000 - 0xFFFFFFFF Kernel
# 0xB0100000 - 0xBFFFFFFF Thread 0 Stack
# 0xB0000000 - 0xB00FFFFF Linker
# 0xA0000000 - 0xBFFFFFFF Prelinked System Libraries
```

1 在本实例中 Context Manager 在内核空间开辟了一块 Buffer，用于接收 IPC 数据。

2 在编译连接时，代码与数据被预先确定下来。

更多更新编程资源尽在 http://www.pin5i.com/


```
# 0x90000000 - 0x9FFFFFFF Prelinked App Libraries
# 0x80000000 - 0x8FFFFFFF Non-prelinked Libraries
# 0x40000000 - 0x7FFFFFFF mmap'd stuff
# 0x10000000 - 0x3FFFFFFF Thread Stacks
# 0x00000000 - 0x0FFFFFFF .text / .data / heap
```

代码 7-3 | prelinked 内存区域

下面是一个显示用户空间（属于使用 Binder 的进程）虚拟内存映射信息的示例。在 Shell 中输入“cat/proc/PID/maps”命令，可以看到如下信息。

```
# cat maps
00008000-0000a000 r-xp 00000000 00:0d 13288118   /system/bin/servicemanager
0000a000-0000b000 rwxp 00002000 00:0d 13288118   /system/bin/servicemanager
0000b000-0000c000 rwxp 0000b000 00:00 0          [heap]
40000000-40008000 r-xs 00000000 00:08 239       /dev/ashmem/system_properties (deleted)
40008000-40028000 r-xp 00000000 00:0e 191       /dev/binder
afbc0000-afbc3000 r-xp 00000000 00:0d 13093317   /system/lib/liblog.so
afbc3000-afbc4000 rwxp 00003000 00:0d 13093317   /system/lib/liblog.so
afc00000-afc21000 r-xp 00000000 00:0d 13093348    /system/lib/libm.so
afc21000-afc22000 rwxp 00021000 00:0d 13093348   /system/lib/libm.so
afd00000-afd01000 r-xp 00000000 00:0d 13093269   /system/lib/libstdc++.so
afd01000-afd02000 rwxp 00001000 00:0d 13093269   /system/lib/libstdc++.so
afe00000-afe39000 r-xp 00000000 00:0d 13093331   /system/lib/libc.so
afe39000-afe3c000 rwxp 00039000 00:0d 13093331   /system/lib/libc.so
afe3c000-afe47000 rwxp afe3c000 00:00 0
b0000000-b0013000 r-xp 00000000 00:0d 13288218   /system/bin/linker
b0013000-b0014000 rwxp 00013000 00:0d 13288218   /system/bin/linker
b0014000-b001f000 rwxp b0014000 00:00 0
bef94000-befa9000 rwxp befeb000 00:00 0          [stack]
```

代码 7-4 | 进程用户空间信息

如代码 7-4 中，0x00008000-0x0000a000 这块区域是当前进程的用户代码区，并且这块区域是只读的。0x0000a000-0x0000b000 这块区域是当前进程的用户数据区，保存着初始化的变量，是可读写区域。0x40008000-0x40028000 这块区域是调用 mmap()函数而产生的区域，这块连续区域的大小即为之前指定的 128KB。

一般地，在将数据从内核空间传递到用户空间时，通常使用 copy_to_user()函数，该函数能够把数据复制到用户空间中。而使用 Binder 的进程则调用 Binder Driver 的 mmap()函数创建与用户空间映射的 Binder mmap 区域。并且，Binder Driver 只在内核空间的数据接收区保存数据，同时将 mmap 区域的信息告知用户空间。因此，即使进程只知道用户空间的 mmap 区域的地址，也可以引用内核空间中保存的数据。图 7-21 是接收 IPC 数据的进程的虚拟内存空间。用户空间的接收区域与内核空间 IPC 数据的接收区域由 binder_mmap()函数映射在一起。当接收 IPC 数据的进程向 Binder Driver 映

更多更新编程资源尽在 http://www.pin5i.com/

射区域保存数据时，即使不调用 copy_to_user()函数，也可以将 IPC 数据传递给用户空间。

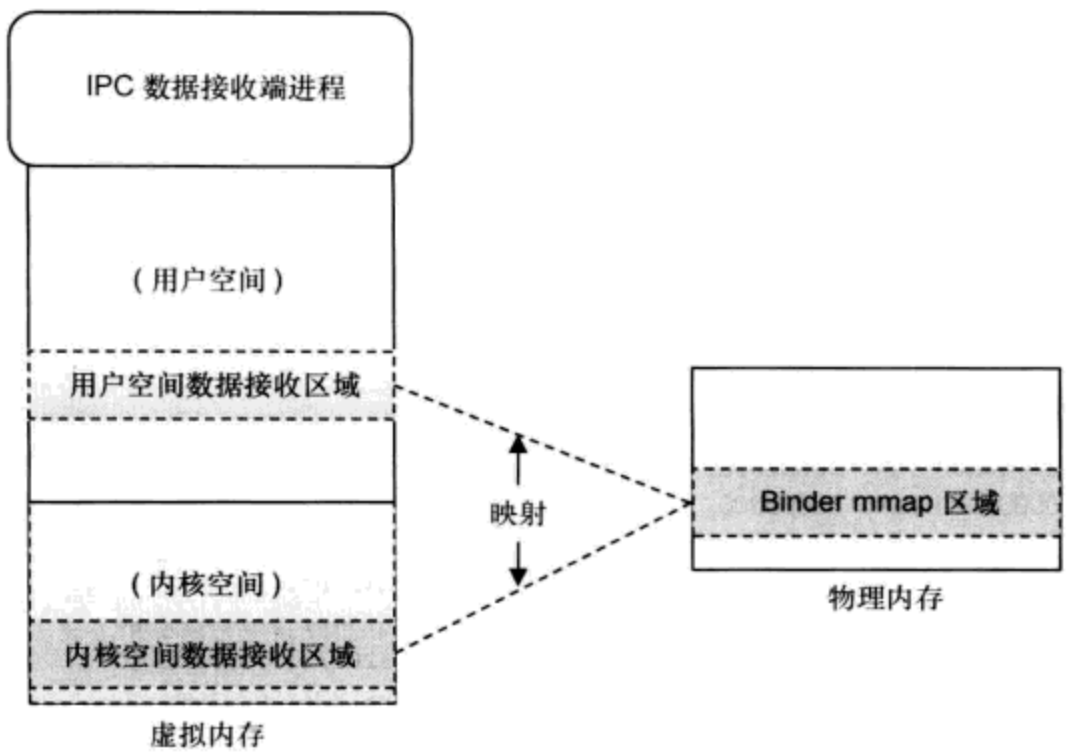


图 7-21 | 调用 binder_mmap()函数映射 binder mmap 区域

Binder Driver 的 binder_mmap()函数可以经由 mmap()系统函数进行调用，它将根据用户进程指定的大小在内核空间中开辟一块 Buffer，用来接收 IPC 数据。图 7-22 描述了 binder_mmap()函数的功能。

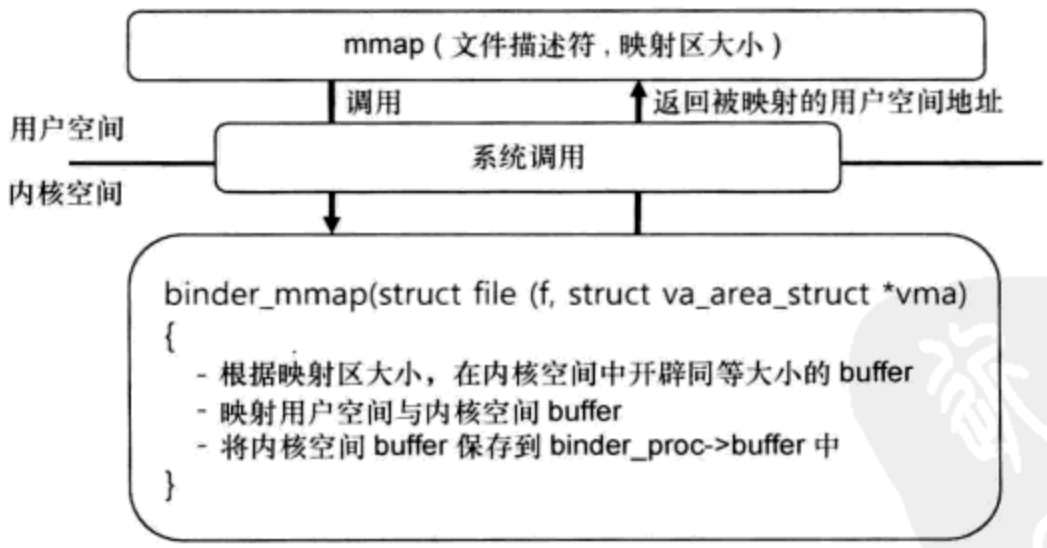


图 7-22 | binder_mmap()函数的功能

如图所示，Binder Driver 首先调用 binder_mmap()函数，根据进程需求的映射区大小在内核空间中开辟一块 Buffer，然后将其与用户空间的 Buffer 映射起来。并且，将内核空间的 Buffer 注册到 binder_proc 结构体之中。

更多更新编程资源尽在 http://www.pin5i.com/


```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
    int ret;
    struct vm_struct *area;
    struct binder_proc *proc = filp->private_data;          ←❶
    const char *failure_string;
    struct binder_buffer *buffer;
```

代码 7-5 | 初始化 binder_mmap()函数的结构体

❶ binder_mmap()函数的第一个参数是一个 file 结构体指针，它指向 Binder Driver 的文件描述符。file 结构体的 private_data 成员变量是当前进程的 binder_proc 结构体，在 binder_open()函数中已经被赋值。binder_mmap()的第二个参数是用户空间的 Buffer 信息，由 mmap()函数与内核空间映射在一起。如代码 7-4 所示，在 prelinked 内存区域中，调用 mmap()函数按指定的大小映射一块虚拟空间，vma->vm_start 与 vm->vm_end 分别指空间的起始地址与结束地址。binder_mmap()函数根据 vma 结构体的信息在内核空间开辟一块 Buffer，并把用户空间与内核空间的 Buffer 映射起来。

```
    area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
    proc->buffer = area->addr;                              ←❶
    proc->user_buffer_offset = vma->vm_start - (size_t)proc->buffer;    ←❷
```

代码 7-6 | binder_mmap()开辟内核 Buffer

如代码 7-6 所示，get_vm_area()是一个内核函数，通过该函数可以向系统申请一块可用的虚拟内存空间，即在内核中申请并保留一块连续的内核虚拟内存空间区域。若内核的 VMALLOC[1]区域有一块符合指定尺寸的空间，则生成一个 vm_struct 结构体，在把空间的起始地址赋给 vm_struct 结构体的 addr 变量后返回。

❶ 调用 get_vm_area()函数将返回 vm_struct 结构体，并将其赋值给 area 变量，此时 area->addr 中存有内核空间的起始地址。当使用 proc->buffer=area->addr 语句后，proc->buffer 即拥有了内核空间中接收 IPC 数据的 Buffer 的起始地址。

❷ 该语句用于获取用户空间 Buffer 地址与内核空间 Buffer 地址的偏移。proc->user_buffer_offset 是 int 类型的变量，它是一个负值，是用户空间映射区域的起始地址与当前分配的内核 Buffer 起始地址的差值。偏移信息用来告知进程用户空间的 Buffer 地址，在 Binder Driver 接收完 IPC 数据后，用户空间的

1 VMALLOC 区域由 vmalloc()函数使用，在不同平台下它的大小是不同的。比如，在 ARM 公司的 Versatile 平台下，VMALLOC 区域的相关信息可以在 kernel/arch/arm/mach-versatile/include/mach/vmalloc.h 文件中查看到。

更多更新编程资源尽在 http://www.pin5i.com/

Buffer 与保存 IPC 数据的内核空间映射在一起。

在代码 7-6 中，我们在内核空间中确定了一块用于接收数据的 Buffer，接下来，应该将用户空间的 Buffer 与内核空间的 Buffer 映射到实际的物理内存上。通过代码 7-7，实现了用户空间的 Buffer 与内核空间的 Buffer 的同步化。

```
proc->pages = kzalloc(sizeof(proc->pages[0])
((vma->vm_end - vma->vm_start) / PAGE_SIZE), GFP_KERNEL);
      ➥ proc->buffer_size = vma->vm_end - vma->vm_start;
if (binder_update_page_range(proc, 1, proc->buffer      ←❶
      ➥ proc->buffer + PAGE_SIZE, vma))
{
   .
   .
}
buffer = proc->buffer;      ←❷
.
binder_insert_free_buffer(proc, buffer);      ←❸
.
```

代码 7-7 | 映射物理内存

❶ 调用 binder_update_page_range()函数，分配物理页，并将存在于物理内存与虚拟内存中的内核空间的接收 Buffer 与用户空间的接收 Buffer 映射起来。

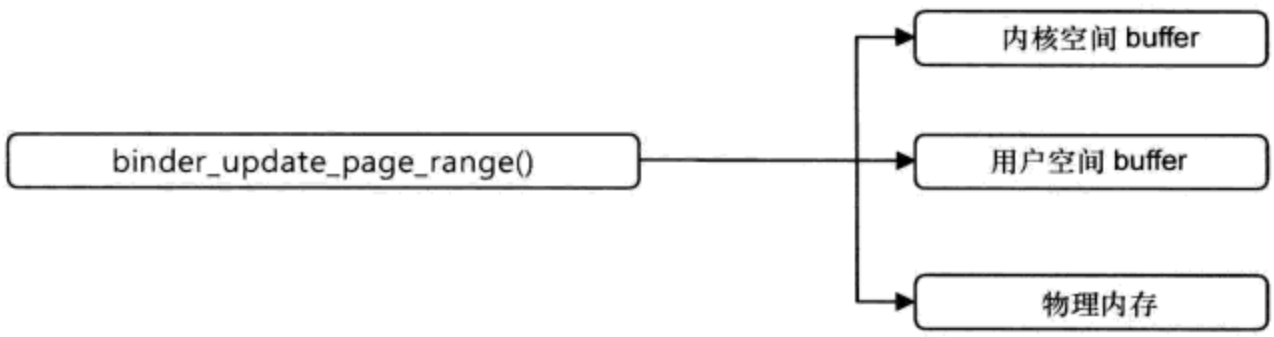


图 7-23 | binder_update_page_range()函数的功能

❷ 当指定的内核空间区域注册到 binder_buffer 结构体，进行 Binder IPC 时，该语句用来将 IPC 数据保存到 binder_buffer 结构体中。

❸ 调用 binder_insert_free_buffer()函数，将❷中确定的 binder_buffer 结构体注册到当前进程的 binder_proc 结构体的 free_buffers 变量中。free_buffers 变量是指接收 IPC 数据的 Buffer，根据 IPC 数据的大小，分配不同大小的 free_buffers，以便接收 IPC 数据。

binder_ioctl()函数分析

在控制 Binder Driver 时，进程将通过 ioctl()函数来调用 binder_ioctl()函数。Binder

更多更新编程资源尽在 http://www.pin5i.com/

Driver 分析随 ioctl()函数调用传递而来的 ioctl 命令，而后作出相应的动作。ioctl 命令在 7.2.1“传递 Binder IPC 数据”一节中已经做了讲解。ioctl 的 BINDER_WRITE_READ 命令用来请求 Binder Driver 发送或接收 IPC 数据以及 IPC 应答数据，本节将分析该命令下的 binder_ioctl()函数。

binder_ioctl()函数负责在两个进程间收发 IPC 数据以及 IPC 应答数据。因此，在分析 binder_ioctl()函数时，必然要涉及到发送 IPC 数据和接收 IPC 数据的两个进程。图 7-24 显示了使用 binder_ioctl()函数的两个进程，图中依次罗列出了 Binder Driver 内部 IPC 数据与 IPC 应答数据的收发过程。

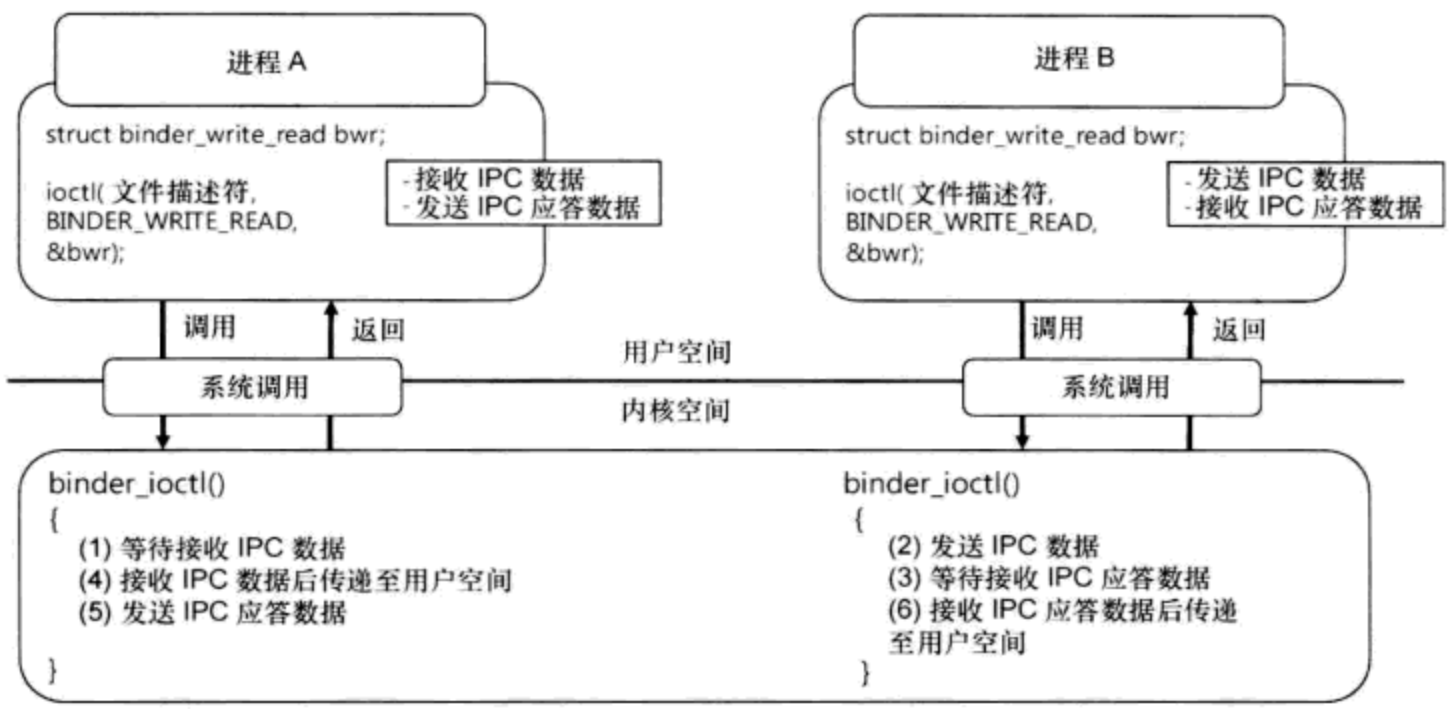


图 7-24 | binder_ioctl()的功能与收发 IPC 数据的 6 个阶段

如图 7-24 所示，进程 B 在调用 binder_ioctl()函数时传入了 BINDER_WRITE_READ 命令，在该函数内部首先发送 IPC 数据（2），而后等待接收 IPC 应答数据（3），再把 IPC 应答数据传递到用户空间中（6）。而在进程 A 的 binder_ioctl()中，首先等待接收 IPC 数据（1），而后接收 IPC 数据并发送到用户空间（4），再发送 IPC 应答数据（5）。

图 7-25 是 binder_write_read 结构体，在调用 ioctl()函数，使用 BINDER_WRITE_READ 命令时，作为参数传入函数中。在图 7-24 中的 ioctl()函数的第三个参数&bwr 即是该结构体的变量。在 binder_write_read 结构体中有两个 Buffer，分别为 write_buffer 与 read_buffer。

```
struct binder_write_read {
        signed long     write_size;     /* bytes to write */
        signed long     write_consumed;         /* bytes consumed by driver */
        unsigned long   write_buffer;
        signed long     read_size;      /* bytes to read */
        signed long     read_consumed;          /* bytes consumed by driver */
        unsigned long   read_buffer;
};
```

发送 IPC 数据 /IPC 应答数据

接收 IPC 数据 /IPC 应答数据

图 7-25 | struct binder_write_read 结构体

更多更新编程资源尽在 http://www.pin5i.com/

binder_write_read 结构体的 write_buffer 成员变量在传递经由 Binder Driver 的数据时使用，即用来发送 IPC 数据或 IPC 应答数据。binder_write_read 结构体的 read_buffer 成员变量用来接收来自 Binder Driver 的数据，即 Binder Driver 在接收 IPC 数据或 IPC 应答数据后，将它们保存到 read_buffer 中，而后再传递到用户空间中。write_size 与 read_size 分别用来指定 write_buffer 与 read_buffer 的数据大小。write_ consumed 与 read_consumed 分别用来设定 write_buffer 与 read_buffer 中被处理的数据大小。binder_write_read 结构体的定义包含在执行 Binder IPC 的进程与 Binder Driver 的源代码的头文件中。如同执行 Binder IPC 时使用的 Binder 协议或 ioctl 命令，binder_write_read 结构体在进程与 Binder Driver 中是通用的。

那么，IPC 数据与 IPC 应答数据以何种形态存在于 binder_write_read 结构体的 write_buffer 与 read_buffer 中呢？在前面的学习中，我们知道 IPC 数据与 IPC 应答数据由 Handle、RPC 代码、RPC 数据、Binder 协议构成。其中，Handle、RPC 代码、RPC 数据保存在名称为 binder_transaction_data 的结构体中。

图 7-26 描述了 binder_transaction_data 结构体的成员变量与 IPC 数据组成要素的关系（事实上，binder_transaction_data 结构体的成员变量远远多于图 7-26 中标注的，但为了便于说明问题，省略了其他无关成员）。如图 7-27 所示，Binder 协议被设置在 binder_transaction_data 结构体的前面，存在于 binder_write_read 结构体的 write_buffer 所指的空间中。

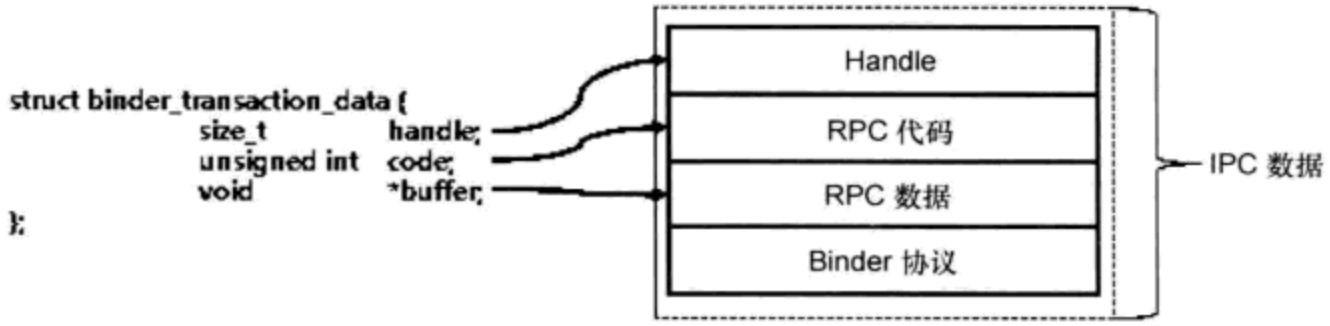


图 7-26 | binder_transaction_data 结构体与 IPC 数据的关系

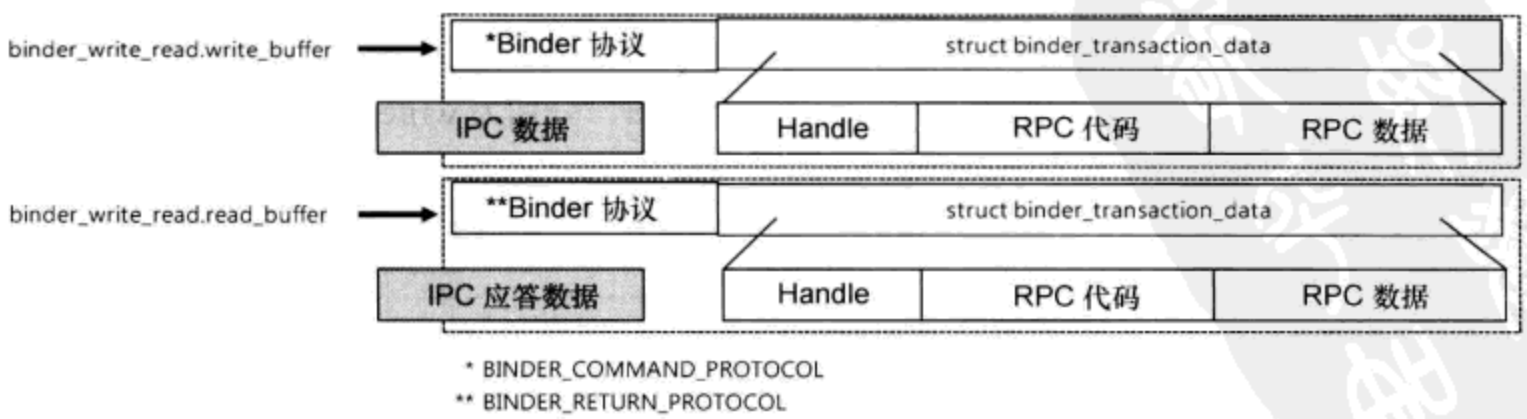


图 7-27 | 发送 IPC 数据的进程的 binder_write_read 结构体

更多更新编程资源尽在 http://www.pin5i.com/

binder_write_read 的 write_buffer 含有 binder_transaction_data 结构体与 Binder 协议，在 Driver 处理 IPC 数据时它们被作为一个处理单元。通常这种处理单元能被连续加载到 write_buffer 中，但为了说明的方便，只考虑有一个 IPC 数据的情形。同样，read_buffer 也是由 Binder 协议与 binder_transaction_data 结构体组成。

binder_write_read 结构体中包含着用户空间中生成的 IPC 数据，Binder Driver 也拥有一个相同的结构体。用户空间设置完 binder_write_read 结构体的数据后，调用 ioctl()函数传递给 Binder Driver，Binder Driver 调用 copy_from_user()函数将用户空间中的数据拷贝到自身的 binder_write_read 结构体中。相反，在传递 IPC 应答数据时，Binder Driver 将调用 copy_to_user()函数，将自身 binder_write_read 结构体中的数据拷贝至用户空间。

Binder Driver 的 binder_ioctl()函数根据服务使用的不同阶段采取不同的 IPC 行为动作。如图所示，它描述了在服务使用的不同阶段，IPC 数据与 IPC 应答数据是如何传递的。

在 7.3.2“从 Binder Driver 角度看服务的使用”一节中已经讲解过 Binder Driver 的动作顺序，下面对 binder_ioctl()函数源码的分析也按照同样的顺序进行。为了将代码与相应的动作对应起来，分析时请随时对照图 7-17、图 7-18、图 7-19 三个插图。IPC 数据与 IPC 应答数据的传递也被称为 Binder 数据传递，我们将这一过程划分为 1～6 六个阶段，如图 7-28 所示。

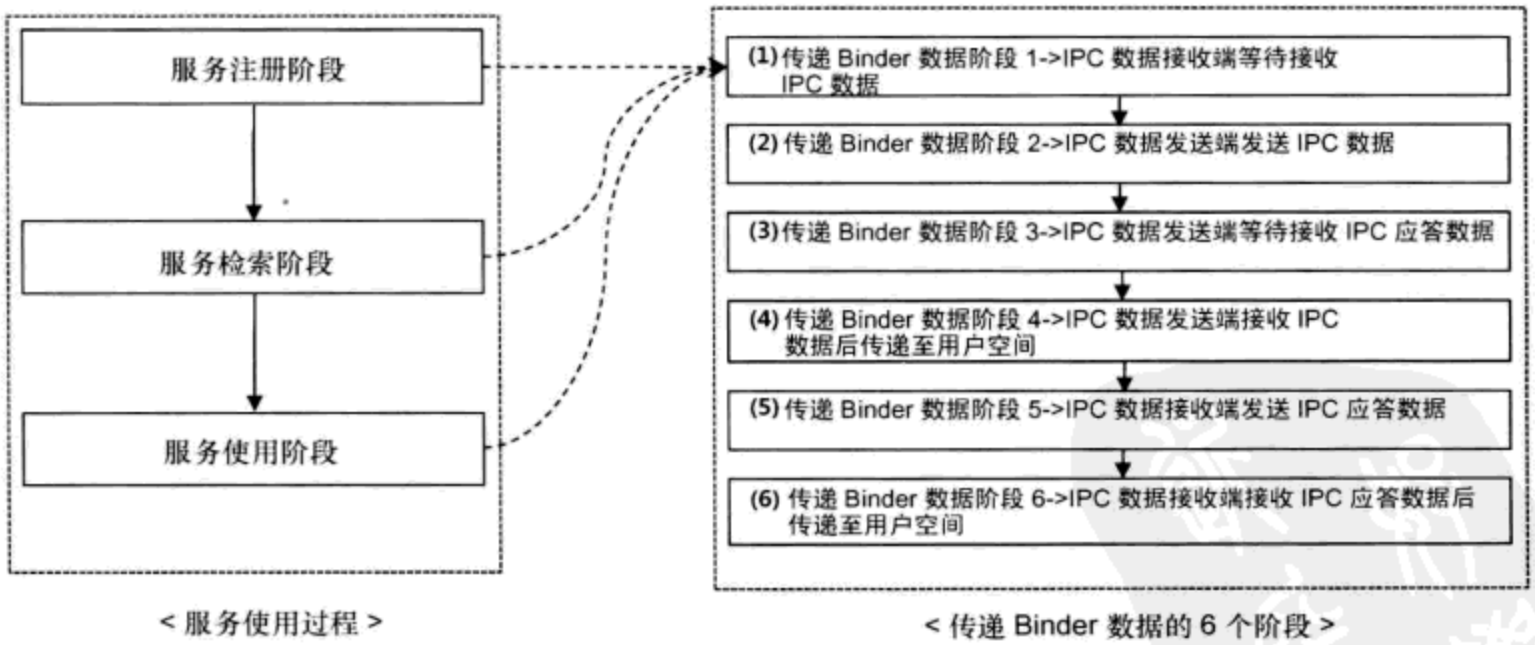


图 7-28 | 在服务使用的不同阶段传递 Binder 数据

服务注册阶段

在服务注册阶段，调用 binder_ioctl()函数首先将 Service Server 的 IPC 数据传递到 Context Manager，然后将 Context Manager 的 IPC 应答数据传递给 Service Server。

更多更新编程资源尽在 http://www.pin5i.com/

（1）传递 Binder 数据的阶段 1

Context Manager 调用 binder_open()与 binder_mmap()函数，初始化 binder_proc 结构体，创建用于接收 IPC 数据的 binder_buffer 结构体。

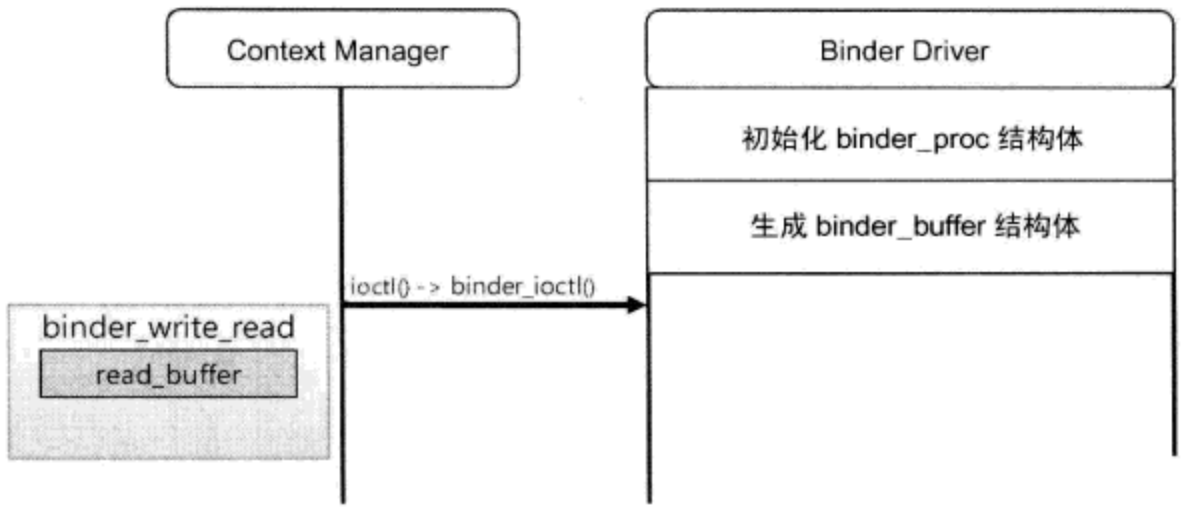


图 7-29 | 传递 Binder 数据的阶段 1

在用户空间中生成一块 Buffer，将 read_buffer 注册到 binder_write_read 结构体中[1]，而后调用 ioctl()函数。此时，binder_write_read 结构体的 read_size 指的是用户空间的 Buffer 的大小。

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
struct binder_proc *proc = filp->private_data                    ❶
void __user *ubuf = (void __user *)arg;
thread = binder_get_thread(proc);                                ❷
switch (cmd) {                                                   ❸
   case BINDER_WRITE_READ: {
      struct binder_write_read bwr;
      if (copy_from_user(&bwr, ubuf, sizeof(bwr)))               ❹
      if (bwr.write_size > 0) {
         ret = binder_thread_write(proc, thread,
         ➥ (void __user *)bwr.write_buffer, bwr.write_size,
         ➥ &bwr.write_consumed);
      if (bwr.read_size > 0) {                                   ❺
         ret = binder_thread_read(proc, thread,
         ➥ (void __user *)bwr.read_buffer, bwr.read_size, &bwr.read_consumed,
         ➥ filp->f_flags & O_NONBLOCK);
```

代码 7-8 | binder_ioctl()函数准备接收 IPC 数据

如代码 7-8 所示，binder_ioctl()函数的第二个参数 cmd 是 ioctl()的命令，第三个参

1 关于 read_buffer 的注册过程，请参看代码 7-32。

更多更新编程资源尽在 http://www.pin5i.com/

数 arg 是 binder_write_read 结构体，binder_write_read 结构体是调用 ioctl()函数时使用的参数。

❶ 获取 binder_open()函数创建的 binder_proc 结构体。

❷ 调用 binder_get_thread()函数，新建一个 binder_thread 结构体，在发送 IPC 应答数据时，它被用来查找接收进程的 binder_proc 结构体（在 7.3.2 一节的服务注册阶段（9）中已作出说明）。

❸ 分析 ioctl()命令。在传递 Binder 数据时，使用的命令是 BINDER_WRITE_READ。从 ioctl()函数的源码可以看出，在 switch 语句中，除了 BINDER_WRITE_READ 之外，还有其他的 ioctl 命令。

❹ 先将 Context Manager 持有的用户空间的 binder_write_read 结构体复制到内核空间中。而后根据 read_size 与 write_size 变量判断是发送 IPC 数据，还是接收 IPC 应答数据。Context Manager 在接收 IPC 数据时，将生成相应的 read_buffer，并且 read_size 的值将大于 0。

❺ 当 read_size 的值大于 0 时，调用 binder_thread_read()函数。

在代码 7-9 中，可以看到 binder_thread_read()函数的代码。

```
static int binder_thread_read(struct binder_proc *proc,
    struct binder_thread *thread,void  __user *buffer, int size,
    signed long *consumed, int non_block)
{
   ret=wait_event_interruptible_exclusive(proc->wait,          ❶
      binder_has_proc_work(proc, thread));
```

代码 7-9 | 调用 binder_thread_read()函数，等待接收数据

在服务注册阶段，调用 binder_thread_read()函数让当前进程进入待机状态。

❶ Context Manager 在 binder_thread_read()函数中调用 wait_event_interruptible_exclusive()函数。wait_event_interruptible_exclusive()函数通过当前进程的 binder_proc 结构体的 wait 变量将当前任务注册到待机队列中。而后将 task_struct[1]结构体内的 state 变量由 TASK_RUNNING 更改为 TASK_INTERRUPTIBLE，再调用 schedule()函数。当 Context Manager 的任务状态为 TASK_INTERRUPTIBLE 时，Context Manager 就不再继续运行，它将进入待机状态，直至接收到 IPC 数据。

1 struct task_struct 是内核中用于进程管理的数据结构。

更多更新编程资源尽在 http://www.pin5i.com/

TIP 进程的状态

Android 是基于 Linux 内核的系统，其进程遵从 Linux 进程调度策略。在 Linux 系统中，进程状态信息保存在 task_struct 结构体的 state 变量中，分为运行状态与待机状态两种。若某进程的 state 值为 TASK_RUNNING，则系统会赋予该进程运行权限并运行之。但当进程的 state 值为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 时，该进程没有运行权限，并处于待机状态，直到其 state 值发生改变。

当一个进程处于待机状态时，调用 wake_up()或 wake_up_interruptible()函数可以将进程转入运行状态。而调用 sleep_on()或 wait_event_interruptible()函数则可以将运行中的进程转入到待机状态中。调用相关函数，转换进程状态，如下图。

（2）传递 Binder 数据的阶段 2

Service Server 调用 binder_open()与 binder_mmap()函数，初始化 binder_proc 结构体，并生成 binder_buffer 结构体，用于接收 IPC 应答数据，如图 7-30 所示。

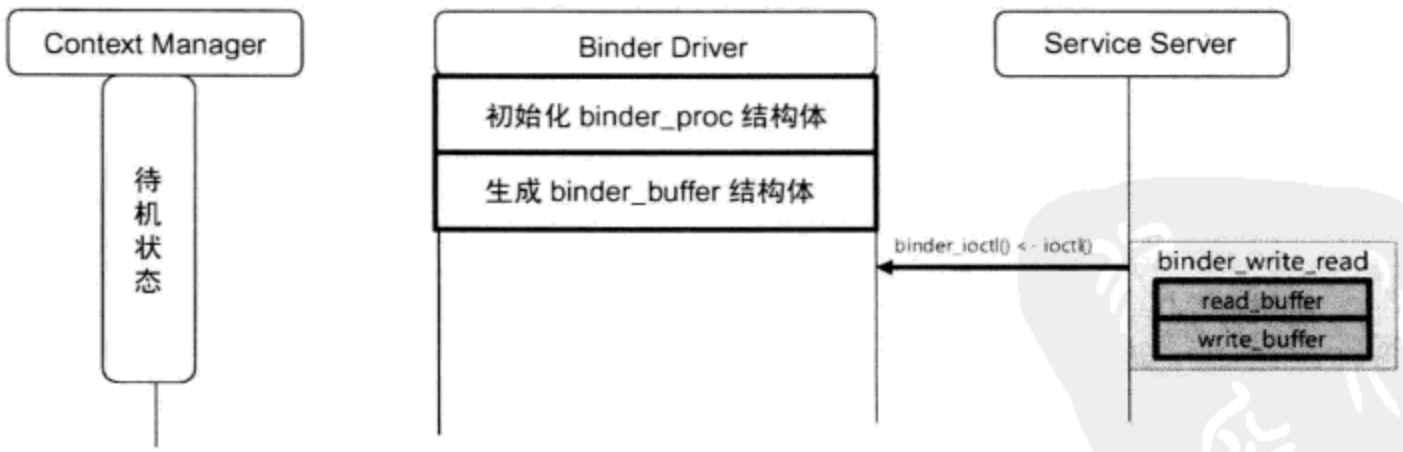


图 7-30 | 传递 Binder 数据的阶段 2

Service Server 也会生成 binder_write_read 结构体，包含 write_buffer 与 read_buffer 两个成员变量。其中，read_buffer 变量用于接收 IPC 应答数据，write_buffer 变量用于发送 IPC 数据。binder_write_read 结构体的形态结构如图 7-31 所示。在传递 Binder

更多更新编程资源尽在 http://www.pin5i.com/

数据的阶段 2 中，服务名称与 flat_binder_object 结构体会被注册到 IPC 数据的 RPC 数据中。

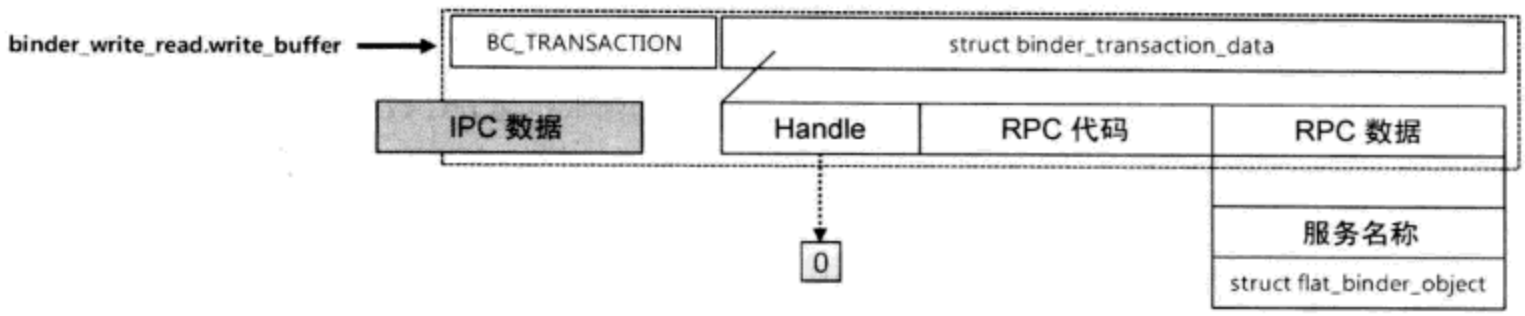


图 7-31 | 用于注册服务的 IPC 数据

在注册或检索服务时，flat_binder_object 结构体[1]用于在 Binder Driver 中生成或查找 Binder 节点（Node）。

```
struct flat_binder_object {
   unsigned long    type;
};
```

代码 7-10 | flat_binder_object 结构体

如代码 7-10 所示，flat_binder_object 结构体中包含一个 type 成员变量，该变量值可以是 BINDER_TYPE_BINDER，也可以是 BINDER_TYPE_HANDLE。若 type 值为 BINDER_TYPE_BINDER，则用来生成 Binder 节点；若 type 值为 BINDER_TYPE_HANDLE，则用来检索 Binder 节点。

```
static long binder_ioctl(struct flle *filp, unsigned int cmd, unsigned long arg)
{
   struct binder_proc *proc = fllp->private_data
   void __user *ubuf = (void __user *)arg;
   thread = binder_get_thread(proc);                          ❶
   switch (cmd) {
      case BINDER_WRITE_READ: {
      struct binder_write_read bwr;
      if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {          ❷
         ret = -EFAULT;
         goto err;
      }
      if (bwr.write_size > 0) {
      ret = binder_thread_write(proc, thread,                 ❸
          (void __user *)bwr.write_buffer, bwr.write_size, &bwr.write_consumed);
      }
```

1 flat_binder_object 结构体可以用来区别各种不同的服务，关于 flat_binder_object 结构体的更多内容，请阅读第 8 章“Android Service Framework”中的相关内容。

更多更新编程资源尽在 http://www.pin5i.com/

```
if (bwr.read_size > 0)
ret = binder_thread_read(proc, thread,
    ➥ (void __user *)bwr.read_buffer, bwr.read_size, &bwr.read_consumed,
    ➥ fllp->f_flags & O_NONBLOCK);
```

代码 7-11 | 调用 binder_thread_read()函数传递 IPC 数据

❶ Service Server 调用 binder_get_thread()函数生成 binder_thread 结构体。

❷ 调用 copy_from_user()函数，将用户空间的 binder_write_read 结构体拷贝至内核空间中。为了传递 IPC 数据，Service Server 保存了 binder_write_read 结构体的 write_buffer 数据，故 write_size 大于 0。

❸ 若 write_size 大于 0，则调用 binder_thread_write()函数，其中 proc 参数是 Service Server 的 binder_proc 类型的结构体变量。

binder_thread_write()函数如代码 7-12 所示，它根据所使用的 Binder 协议而采取不同的动作。

```
int binder_thread_write(struct binder_proc *proc, struct binder_thread *thread,
  ➥ void __user *buffer, int size, signed long *consumed)
{
uint32_t cmd;
void __user *ptr = buffer + *consumed;                    ←❶
void __user *end = buffer + size;
while (ptr < end && thread->return_error == BR_OK) {
   if (get_user(cmd, (uint32_t __user *)ptr))             ←❷
       return -EFAULT;
   ptr += sizeof(uint32_t);
```

代码 7-12 | binder_thread_write()函数的初始化

❶ 该变量指当前 write_buffer 中要处理的 IPC 数据。如前所述，多个 IPC 数据能被加载到 write_buffer 中，consumed 指 IPC 数据的个数。若当前没有 IPC 数据被处理，则*ptr 指向首个 IPC 数据。

❷ 调用 get_user()函数，将 IPC 数据的 Binder 协议拷贝至内核空间中。在服务注册时，所使用的 Binder 协议是 BC_TRANSACTION。

```
switch (cmd) {
   case BC_TRANSACTION:
      case BC_REPLY:
   struct binder_transaction_data tr;
      if (copy_from_user(&tr, ptr, sizeof(tr)))    ←❶
```

更多更新编程资源尽在 http://www.pin5i.com/


```
        return -EFAULT;
    ptr += sizeof(tr);
    binder_transaction(proc, thread, &tr, cmd == BC_REPLY);      ←❷
    break;
    }
```

代码 7-13 | binder_thread_write()函数中对 Binder 协议的处理

❶ 调用 copy_from_user()函数，将 IPC 数据中不包含 Binder 协议的 binder_transaction_data 结构体复制到内核空间中。在代码 7-11❶中已经复制过 binder_write_read 结构体了，在此再次复制 binder_transaction_data 结构体的原因在于 binder_write_read 结构体的 write_buffer 变量只是一个用户空间中 binder_transaction_data 结构体的指针，必须将指针所指的内容重新拷贝到内核空间中。

❷ 调用 binder_transaction()函数，binder_transaction()函数使用 binder_transaction_data 结构体中的数据来执行 Binder 寻址、复制 Binder IPC 数据、生成及检索 Binder 节点等操作。

```
static void binder_transaction(struct binder_proc *proc,
    struct binder_thread *thread, struct binder_transaction_data *tr, int reply)
{
    struct binder_transaction *t;
    struct binder_proc *target_proc;
    struct binder_node *target_node = NULL;
    wait_queue_head_t *target_wait;
```

代码 7-14 | binder_transaction()函数使用的结构体

代码 7-14 是 binder_transaction()函数所使用的结构体。

当 IPC 数据接收端从待机状态中苏醒后，IPC 数据发送端会使用 binder_transaction 结构体来查找等待发送的 IPC 数据。IPC 数据发送端与接收端通过该结构体来收发 IPC 数据。

target_proc 是一个指向 binder_proc 结构体的指针，它指向 Binder IPC 数据接收端的 binder_proc 结构体。target_node 是 binder_node 指针，它指向一个 binder_node 结构体，且该结构体已注册到接收端的 binder_proc 结构体中。target_wait 用来帮助接收端脱离待机状态。

```
if (tr->target.handle)
{
    struct binder_ref *ref;
    ref = binder_get_ref(proc, tr->target.handle);
    target_node = ref->node;
}
else
{
```

更多更新编程资源尽在 http://www.pin5i.com/

```
    target_node = binder_context_mgr_node;          ←❶
}
target_proc = target_node->proc;                    ←❷
```

代码 7-15 | 获取接收端的 Binder 节点

在服务注册阶段，Binder IPC 数据的接收端是 Context Manager，因此 Handle 值为 0。

❶ 该语句将全局变量 binder_context_mgr_node 赋值给 target_node 变量。

> **TIP binder_context_mgr_node**
>
> Context Manager 调用 ioctl（“/dev/binder”，BINDER_SET_CONTEXT_MGR）函数，生成 binder_node，而后将其赋给全局变量 binder_context_mgr_node。使用时只要将 Handle 设置为 0，即可使用 Context Manager 的服务节点。

❷ target_node 指向 Context Manager 的 Binder 节点，该行通过 target_node 获取 Context Manager 的 binder_proc 结构体。binder_proc 结构体的 proc 变量指向其自身所属的 binder_proc 结构体。binder_node 结构体与 binder_proc 结构体的关系见图 7-32。

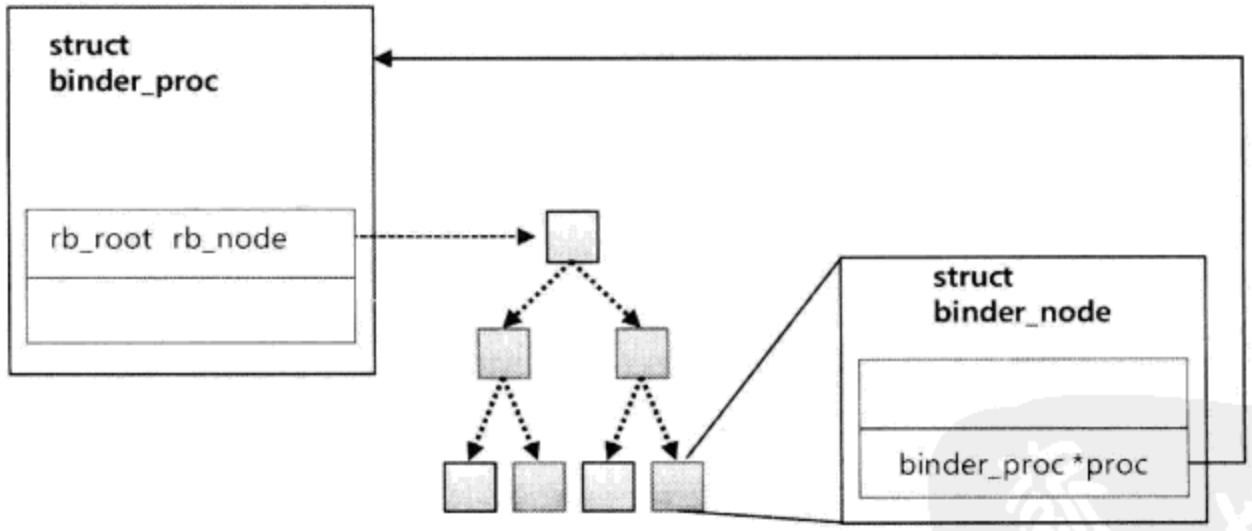


图 7-32 | 通过 binder_node 结构体引用 binder_proc 结构体

```
if (target_thread) {
    e->to_thread = target_thread->pid;
    target_list = &target_thread->todo;
    target_wait = &target_thread->wait;
} else {
    target_list = &target_proc->todo;           ←❶
    target_wait = &target_proc->wait;           ←❷
}
t = kzalloc(sizeof(*t), GFP_KERNEL);            ←❸
```

代码 7-16 | 获取接收端的 wait_queue

更多更新编程资源尽在 http://www.pin5i.com/

在收发 Binder 数据的第一个阶段，Context Manager 进入等待接收 IPC 数据的状态。而后通过代码 7-16，查找待机队列（wait_queue），以便使 Context Manager 脱离等待状态。

❶ 在 Context Manager 从等待状态苏醒后，将 binder_proc 数据结构的 todo 变量赋给 target_list，以指定要执行的任务。

❷ 获取 Context Manager 的 wait_queue。

❸ 变量 t 是一个指向 binder_transaction 结构体的指针。该语句用来生成 binder_transaction 结构体，用来向接收端发送 IPC 数据。

```
if (!reply && !(tr->flags & TF_ONE_WAY))
    t->from = thread;                                    ← ❶
t->to_proc = target_proc;
t->code = tr->code;                                      ← ❷
t->buffer = binder_alloc_buf(target_proc, tr->data_size,    ← ❸
    ➥ tr->offsets_size, !reply && (t->flags & TF_ONE_WAY));
t->buffer->transaction = t;
t->buffer->target_node = target_node;
if (copy_from_user(t->buffer->data, tr->data.ptr.buffer, tr->data_size)) {    ← ❹
}
```

代码 7-17 | 拷贝 IPC 数据到 binder_transaction()

❶ 查找发送 IPC 数据的进程，以便发送 IPC 应答数据，在收发 Binder 数据的第 5 个阶段中会讲到。该行语句把发送 IPC 数据的 Service Server 的 binder_thread 结构体赋值给 binder_transaction 结构体的 from 变量中。关于通过 binder_thread 结构体查找接收端 binder_proc 数据结构的过程，将在收发 Binder 数据的第 5 个阶段讲解。

❷ 将 IPC 数据的 RPC 代码赋给 binder_transaction 结构体的 code 变量。

❸ 调用 binder_alloc_buf()函数，分配一块空间，用来从接收端的 IPC 数据接收缓冲区中复制 IPC 数据。该接收缓冲空间由接收端在 binder_mmap()函数中确定，binder_proc 结构体的 free_buffers 指向该区域。binder_alloc_buf()函数在 free_buffers 中根据 RPC 数据的大小分配 binder_buffer 结构体的 data，而后将 binder_buffer 结构体返回。

❹ 拷贝 RPC 数据到 binder_buffer 结构体的 data 变量中。图 7-33 描绘了内核空间的 IPC 数据接收 Buffer 与用户空间数据接收区域的映射，以及拷贝 IPC 数据的过程。

在 IPC 数据中，Handle、RPC 代码、Binder 协议三种数据形态都是一定的，其尺寸固定不变，但 RPC 数据根据用处的不同，其形态会有所不同，且它的尺寸是可变的。因此在 Binder IPC 过程中，Android 系统将根据 RPC 数据的具体尺寸动态分配不同大

更多更新编程资源尽在 http://www.pin5i.com/

小的空间。并且通过将数据复制到映射空间的方式，防止因 RPT 数据过大而造成性能降低。

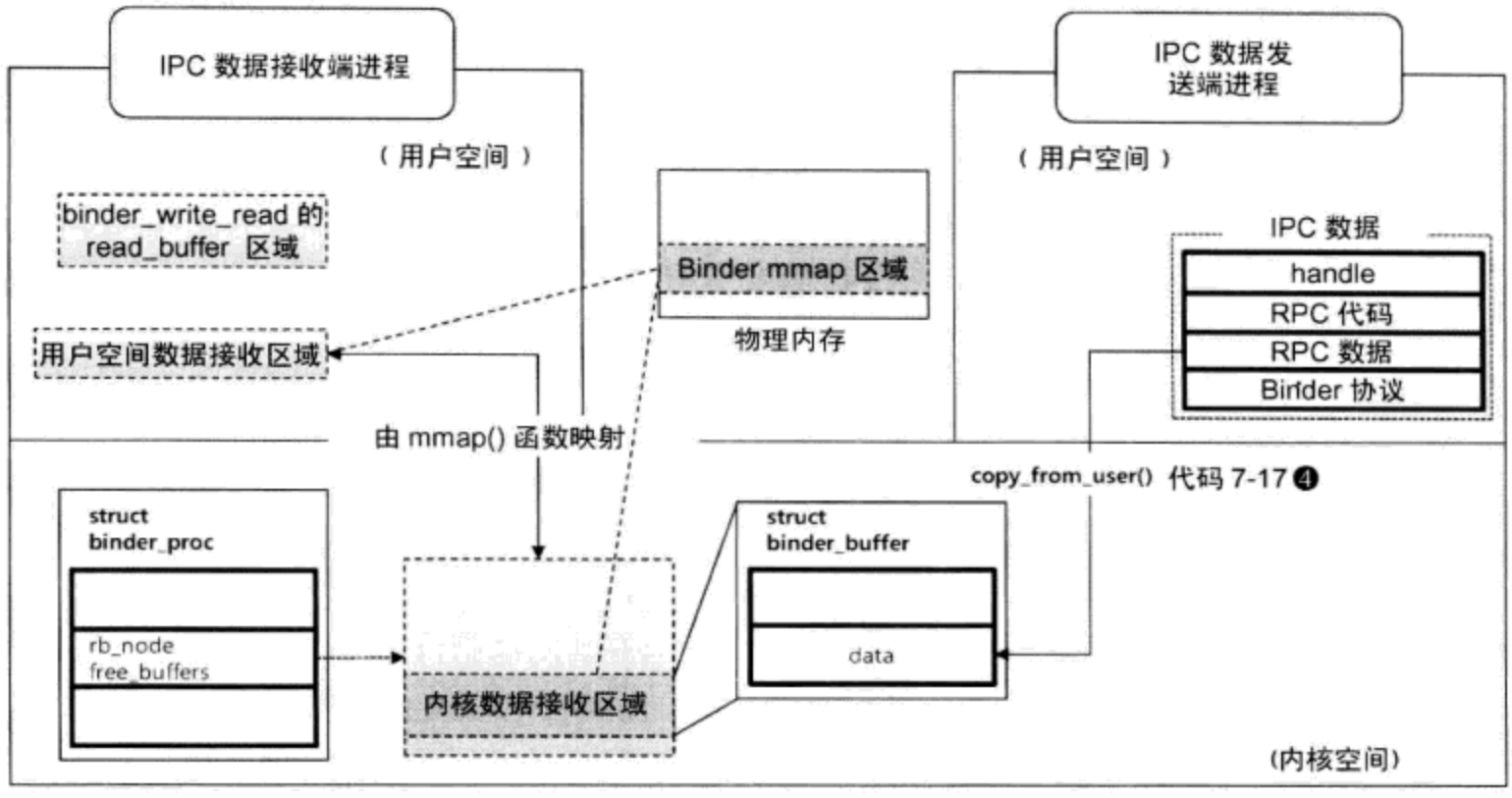


图 7-33 | 向接收端 Buffer 拷贝 RPC 数据

至此，Binder 数据发送端设置 binder_transaction 结构体的过程结束了。接下来，接收端将从待机状态中苏醒，并从该结构体中获取数据。下面我们来分析为各服务创建 Binder 节点（Node）的过程。

```
fp = (struct flat_binder_object *)(t->buffer->data + *offp);
switch (fp->type) {
   case BINDER_TYPE_BINDER:
   case BINDER_TYPE_WEAK_BINDER: {
      struct binder_ref *ref;
      struct binder_node *node = binder_get_node(proc, fp->binder);   ❶
      if (node == NULL) {
         node = binder_new_node(proc,                                 ❷
                                fp->binder,
                                fp->cookie);
      }
      ref = binder_get_ref_for_node(target_proc, node);               ❸
      fp->handle = ref->desc;
}
```

代码 7-18 | 创建新的 Binder 节点（binder_node 结构体）

代码 7-18 是 IPC 的 RPC 数据包含 flat_binder_object 结构体时执行的代码。在服务注册阶段，flat_binder_object 结构体中的 type 变量值为 BINDER_TYPE_BINDER。

更多更新编程资源尽在 http://www.pin5i.com/

❶ 调用 binder_get_node()函数，在注册至 binder_proc 结构体中的 Binder 节点中，确认当前的 Binder 节点是否已经存在。若不存在，返回 NULL，继续执行代码行❷。

❷ 在服务注册过程中会生成新的 Binder 节点，该行调用 binder_new_node()函数，创建新的 binder_node 结构体后返回。并且，在该函数体内新生成的 binder_node 结构体将被注册到当前进程的 binder_proc 结构体之中。

❸ 该行调用 binder_get_ref_for_node()函数，检查指定的 Binder 节点（函数的第二个参数）是否已经存在于指定的 binder_proc 结构体（函数的第一个参数）中。若存在，则返回 Binder 节点所在的 binder_ref 结构体，否则新建 binder_ref 结构体并将指定的 Binder 节点注册其中，而后将其返回。并且，binder_get_ref_for_node()函数也将 binder_ref 结构体注册到 binder_proc 结构体（函数的第一个参数）中的 refs_by_node 变量中。binder_ref 结构体既是 Binder 节点索引，也是参考数据[1]。

```
struct binder_ref {
    struct binder_proc *proc;
    struct binder_node *node;
    uint32_t desc;
};
```

代码 7-19 | binder_ref 结构体

代码 7-19 是 binder_ref 结构体的定义。proc 是 binder_proc 类型的指针，它指向 binder_ref 结构体所在的 binder_proc 结构体；node 是 binder_node 类型指针，指向 binder_ref 结构体所持有的 binder_node 结构体。desc 是一个编号，用来区分当前进程的 binder 节点，当有 binder 节点生成时该编号就会递增。例如，在 Service Server 首次注册服务第一次生成 Binder 节点时，binder_ref 结构体的 desc 值为 1。由于 Context Manager 的 Binder 节点编号约定为 0，所以其他 desc 的值从 1 开始编号。

如上所述，binder_get_ref_for_node()函数的第一个参数 target_proc 是 Context Manager 的 binder_proc 结构体，第二个参数是 binder 节点。调用 binder_get_ref_for_node()函数，创建 binder_ref 结构体，并注册第二个参数传递过来的 binder 节点。图 7-34 描述了调用 binder_get_ref_for_node()函数后，Service Server 的 binder_node 结构体被注册到 Context Manager 的 binder_proc 结构体的过程。

当前创建的 binder_ref 结构体的 desc 编号用来在 Context Manager 管理的服务列表中区分不同的服务。至此，Service Server 创建了待注册服务的 Binder 节点，并将其注

1 在 7.2.5“Binder 选址”一节中提到的参考数据即是 binder_ref 结构体。

更多更新编程资源尽在 http://www.pin5i.com/

册到 Context Manager 之中。Service Server 完成了在服务注册阶段传递 IPC 数据的所有准备工作，接下来将 Context Manager 从待机状态中唤醒，使之获取 binder_transaction 结构体中的 IPC 数据，如代码 7-17 所示。代码 7-20 的功能是从待机状态中唤醒 Context Manager。

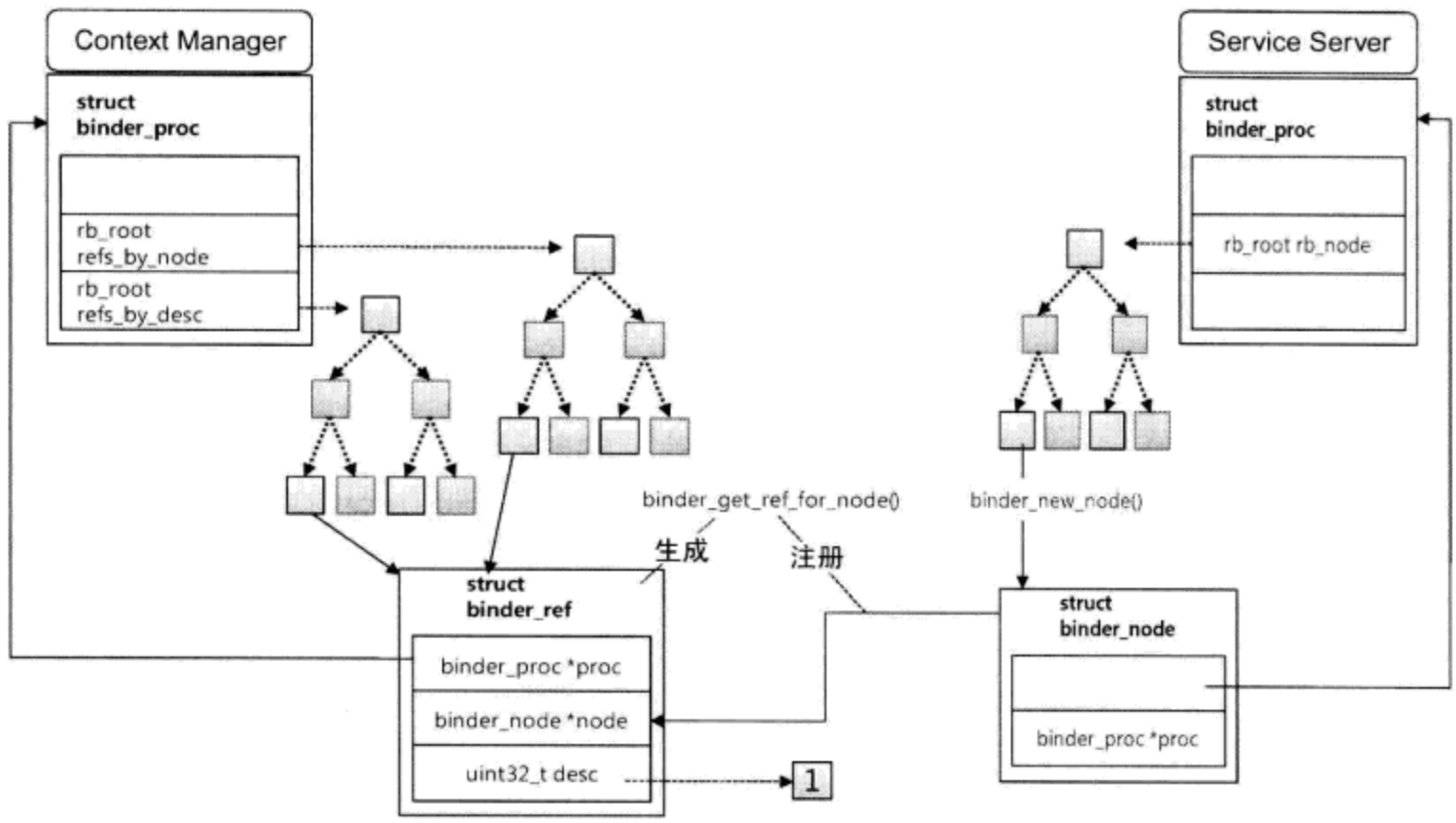


图 7-34 | 创建 Binder 节点并注册到 Context Manager 中

```
t->work.type = BINDER_WORK_TRANSACTION;          ❶
list_add_tail(&t->work.entry, target_list);      ❷
if (target_wait)
   wake_up_interruptible(target_wait);           ❸
```

代码 7-20 | 唤醒接收端进程

❶ Context Manager 从待机状态脱离，以 binder_transaction 结构体中的 binder_work 结构体的 type 变量为基础，执行动作。

```
struct binder_work {
   struct list_head entry;
   enum {
      BINDER_WORK_TRANSACTION = 1,
   } type;
```

代码 7-21 | binder_work 结构体

更多更新编程资源尽在 http://www.pin5i.com/

当 binder_work 结构体的 type 变量值为 BINDER_WORK_TRANSACTION 时，Context Manager 从 binder_transaction 结构体中获取 Service Server 传递而来的 IPC 数据。详细内容将在 Binder 数据传递第 4 个阶段中讲解。

❷ Context Manager 确认自身 binder_proc 结构体中的 todo 变量，该变量在 Binder 数据收发的第❹个阶段中会用到。在代码 7-16❶中，已经将 Context Manager 的 binder_proc 结构体的 todo 变量值赋给 target_list 变量中，该行将注册❶中设置的 binder_work 结构体。

❸ 调用 wake_up_interruptible()函数，并使用代码 7-16❷中的 target_wait 作为参数，将 Context Manager 从待机状态中唤醒。在接下来的进程调度中，Context Manager 将执行 Binder 数据收发第 4 个阶段的相应任务。

（3）传递 Binder 数据的阶段 3

在传递 Binder 数据的第 1 个阶段中，Context Manager 通过 binder_write_read 结构体的 read_buffer 进入到待机状态中。类似地，Service Server 完成代码 7-11 的过程❷后，执行第❸条语句，进入待机状态。Service Server 将在收发 Binder 数据的第 6 个阶段中苏醒，接收 IPC 应答数据。Service Server 进入待机状态的过程与 Context Manager 进入待机状态的过程是一样的，在此不再赘言。

（4）传递 Binder 数据的阶段 4

Context Manager 在传递 Binder 数据的第 2 个阶段中将从待机状态中被唤醒，即从代码 7-9❶中的待机状态中苏醒，开始执行接下来的动作。Context Manager 首先查找 binder_transaction 结构体，该结构体是 Service Server 在代码 7-17 中生成的。

```
while (1) {
   uint32_t cmd;
   struct binder_transaction_data tr;
   struct binder_work *w;
   struct binder_transaction *t = NULL;
   if (!list_empty(&proc->todo) && wait_for_proc_work)
      w = list_first_entry(&proc->todo, struct binder_work, entry);          ←❶
   switch (w->type) {
      case BINDER_WORK_TRANSACTION: {
         t = container_of(w, struct binder_transaction, work);               ←❷
      } break;
};
```

代码 7-22 | 查找 binder_transaction 结构体

在代码 7-22 中，binder_transaction_data 结构体用来向用户空间传递 IPC 数据；binder_work 结构体用于查找 Service Server 在代码 7-17 中生成的 binder_transaction 结

更多更新编程资源尽在 http://www.pin5i.com/

构体，查找到的 binder_transaction 结构体被保存到指针变量 t 中。

❶ 调用 list_first_entry()函数，查找 Service Server 在代码 7-20 中注册的 binder_work 结构体，并将查找到的 binder_work 结构体返回。list_first_entry()函数是 container_of()函数的引用函数，container_of()函数将根据首个参数（该参数是 binder_work 结构体）查找到包含该参数的结构体的首地址。

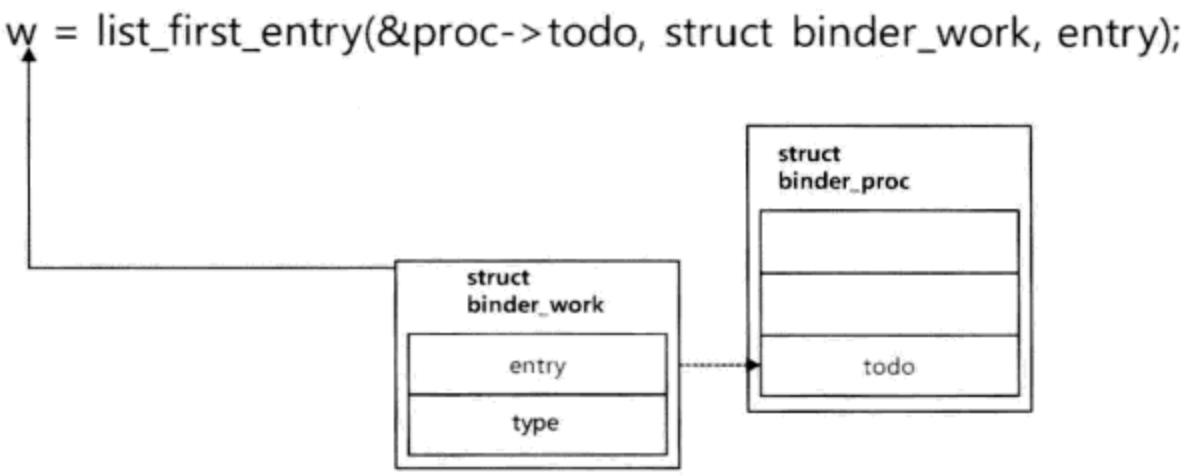


图 7-35 | 查找并获取 Service Server 注册的 binder_work 结构体

❷ 若 binder_work 结构体的 type 成员变量值为 BINDER_WORK_TRANSACTION，则继续调用 container_of()函数，查找持有 binder_work 结构体的 binder_transaction 结构体。

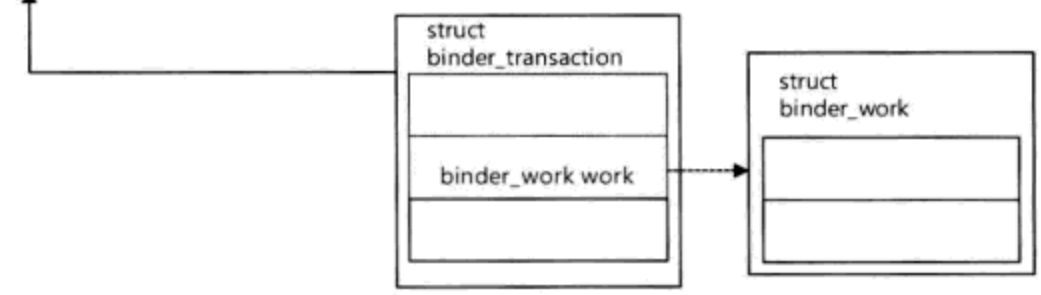


图 7-36 | 查找获取 Service Server 注册的 binder_transaction 结构体

TIP container_of()函数

container_of()函数通过某结构体内的成员变量的地址来查找结构体的首地址，这是因为结构体内的成员变量所在的空间都是连续的。container_of()函数的第一个参数 ptr 是结构体成员变量的地址，第二个参数 type 是待查找的 struct 定义，第三个参数是第一个参数在结构体内的名称。

```
/**
 * container_of - cast a member of a structure out to the
 * containing structure
```

更多更新编程资源尽在 http://www.pin5i.com/


```
 * @ptr:    the pointer to the member.
 * @type:   the type of the container struct this is embedded in.
 * @member: the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) ({          \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

在调用该函数时，必须提供第一个参数，即结构体内成员变量的地址。除此之外，还必须给出包含该成员变量的结构体的定义，当然成员变量在结构体内的名称也是必须的。在调用container_of()函数时，只有给出这三个参数，才能查找到特定结构体的起始地址。

Context Manager通过Service Server创建的binder_transaction结构体获取IPC数据，并将 IPC 数据保存到 binder_transaction_data 结构体，以便将 IPC 数据传递到用户空间中。

```
tr.code = t->code;                                          ←❶
tr.data_size = t->buffer->data_size;
tr.data.ptr.buffer = (void *)t->buffer->data + proc->user_buffer_offset;    ←❷
```

代码 7-23 | 保存 IPC 数据到 binder_transaction_data 结构体

❶ 保存 Service Server 传递的 RPC 代码。

❷ 保存 Service Server 传递的 RPC 数据。如图 7-33 所示，内核空间接收 Buffer 的地址为 t->buffer->data，而 proc->user_buffer_offset 则指用户空间的接收 Buffer 与内核空间的接收 Buffer 间的地址偏移。因此 tr.data.ptr.buffer 保存的是用户空间中接收 Buffer 的地址。binder_write_read 结构体存在于用户空间中，read_buffer 是 binder_write_read 结构体的成员变量，它可以引用保存在 tr.data.ptr.buffer 所指的用户空间中的 IPC 数据。在向接收端传递数据时，Binder IPC 也采用相同的方法，将内核空间中的数据传递到用户空间中，而不需要调用 copy_to_user() 函数。通过这种方式，可以避免在内核空间与用户空间之间拷贝数据，从而提升 IPC 运作的效率。

经过❶与❷两个操作后，即完成了设置 binder_transaction_data 结构体的任务。接下来，就要将 binder_transaction_data 结构体中的数据传递到用户空间中。从该阶段开始，数据将被保存到 Context Manager 的 read_buffer 中。

```
cmd = BR_TRANSACTION;                                       ←❶
if (put_user(cmd, (uint32_t __user *)ptr))                  ←❷
   return -EFAULT;
```

更多更新编程资源尽在 http://www.pin5i.com/

```
ptr += sizeof(uint32_t);
if (copy_to_user(ptr, &tr, sizeof(tr)))          ←❸
   return -EFAULT;
if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
   t->to_parent = thread->transaction_stack;
   t->to_thread = thread;
   thread->transaction_stack = t;                ←❹
```

代码 7-24 | 拷贝 IPC 数据到用户空间

❶ 设置 Binder 协议为 BR_TRANSACTION，准备处理 Context Manager 的 IPC 数据。

❷ *ptr 指向 binder_write_read 结构体的 read_buffer 成员变量，在传递 Binder 数据的第 1 个阶段中，Context Manager 在进入待机状态前调用 ioctl()函数时，它作为一个参数被传入函数中。

❸ 将代码 7-23 与代码 7-24 中生成的 binder_transaction_data 结构体拷贝到 read_buffer 中。经过❶～❸过程后，read_buffer 中的数据如下图所示。

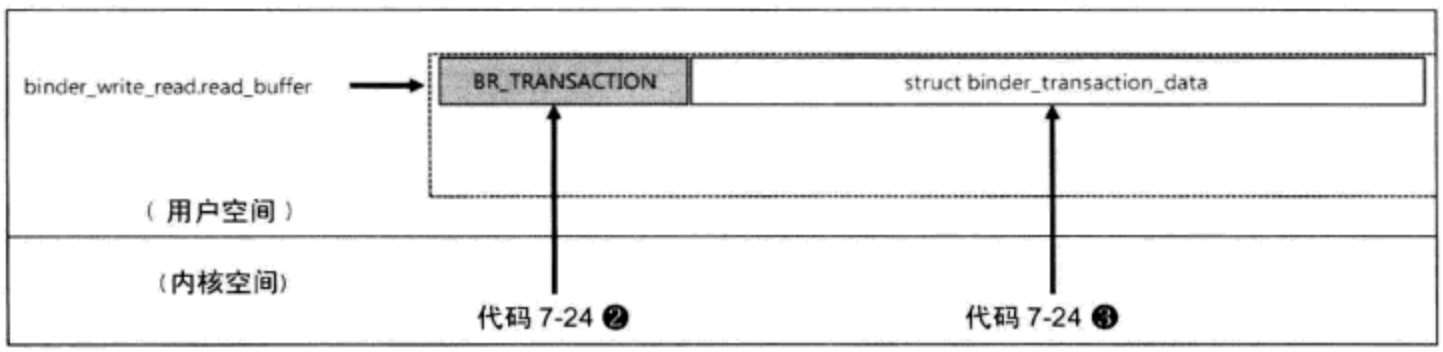


图 7-37 | 拷贝内核空间中的 IPC 数据到用户空间中

❹ 在代码 7-8 的❷中，调用 binder_get_thread()函数，将当前的 binder_transaction 结构体注册到 binder_thread 结构体的 transaction_stack 中。binder_transaction 结构体在传递 Binder IPC 数据的第 5 个阶段中用于查找 IPC 应答数据的接收端。

图 7-38 描述了通过 binder_thread 结构体查找另一端进程的过程。在各结构体的连接部分标注出了源代码所在的位置，阅读时可以随时翻到相应的位置查看这些代码。随后，Binder Driver 的 binder_ioctl()函数执行结束，Context Manager 调用的 ioctl()函数返回。Context Manager 分析 binder_write_read 结构体的 read_buffer 中的 Binder 协议，而后调用 binder_transaction_data 结构体的 RPC 代码所指的函数，并将 RPC 数据作为参数传递给调用的函数。经过这一系列的操作，Service Server 的服务即被注册到 Context Manager 之中。

更多更新编程资源尽在 http://www.pin5i.com/

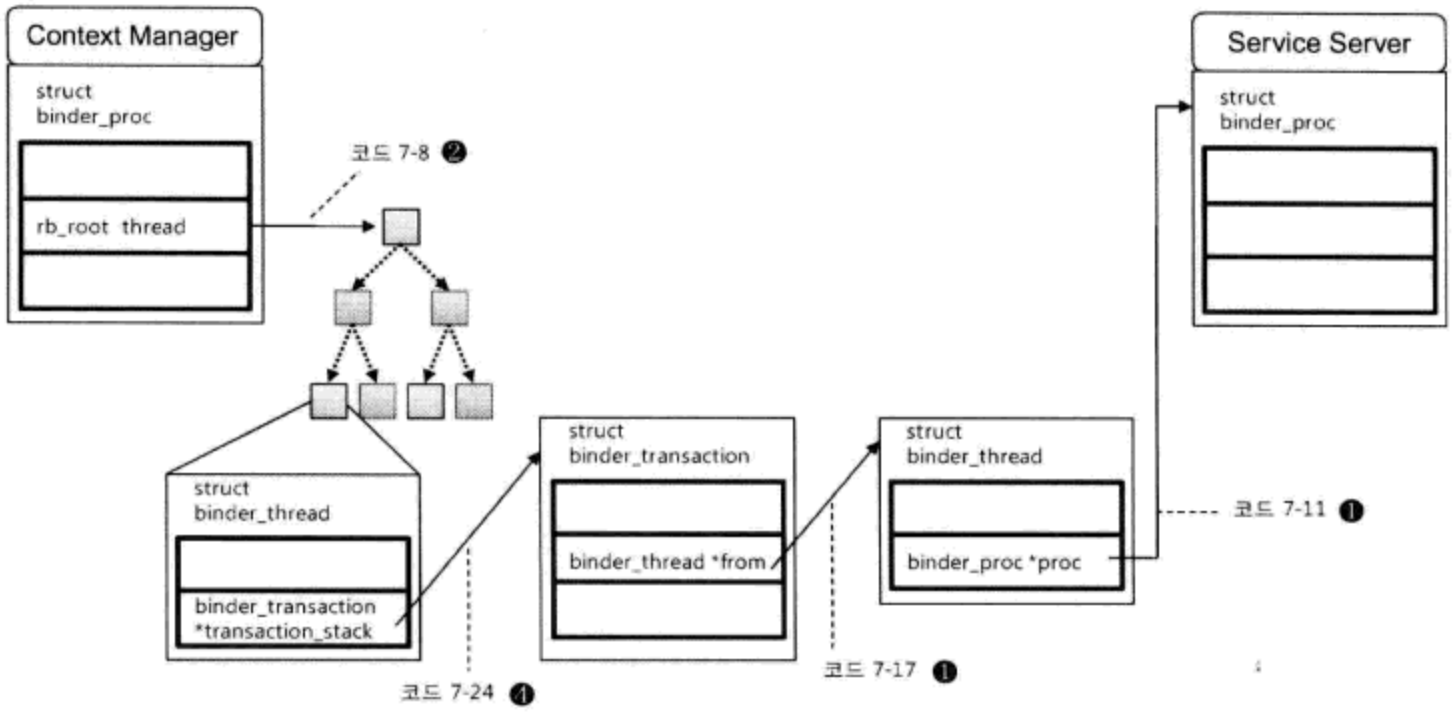


图 7-38 | 由 binder_thread 结构体查找另一端进程

（5）传递 Binder 数据的阶段 5

在此阶段中，Context Manager 将处理接收到的 IPC 数据，而后向 Service Server 发送 IPC 应答数据，告知处理已经完成。该阶段的数据传递过程与传递 Binder 数据的第 2 个阶段类似，涉及到的 Binder 协议分别是 BC_REPLY（Context Manager→Binder Driver）与 BR_REPLY（Binder Driver→Context Manager）。

如代码 7-13 所示，在 binder_thread_write()函数中根据 Binder 协议的不同进行不同的处理，当 Binder 协议为 BC_REPLY 时，调用 binder_transaction()函数。代码 7-25 描述了传递 IPC 应答数据时查找接收端 binder_proc 结构体的过程。

```
if (reply) {
   in_reply_to = thread->transaction_stack;          ←❶
   thread->transaction_stack = in_reply_to->to_parent;
   target_thread = in_reply_to->from;                ←❷
   target_proc  = target_thread->proc;               ←❸
}
```

代码 7-25 | binder_transaction()函数查找 IPC 应答数据的接收进程

❶ 获取代码 7-24❹中注册的 binder_transaction 结构体。

❷ 获取代码 7-17❶中注册的 Service Server 的 binder_thread 结构体。

❸ 通过 binder_thread 结构体，获取 Service Server 的 binder_proc 结构体。在获得了 Service Server 的 binder_proc 结构体之后，即可获得 Service Server 的接收

更多更新编程资源尽在 http://www.pin5i.com/

Buffer。通过代码 7-17，将 IPC 应答数据保存到 binder_transaction 结构体中。而后 Service Server 从待机状态中苏醒，查找相应的结构体，将 IPC 应答数据传递到用户空间中。该过程与传递 Binder 数据的第 2 个阶段类似，在此不再赘述。需要注意的是，在服务注册阶段 IPC 应答数据中不存在 flat_binder_ object 结构体，所以创建 Binder 节点的过程被省略掉了。

（6）传递 Binder 数据的阶段 6

Service Server 从待机状态中苏醒后，其行为动作与传递 Binder 数据第 4 个阶段类似，不同的是所使用的 Binder 协议为 BR_REPLY。Service Server 接收完 IPC 应答数据后将在用户空间中执行一系列与 IPC 应答数据相关的动作。

服务检索阶段

在服务检索阶段，调用 binder_ioctl()函数，将服务客户端的 IPC 数据发送到 Context Manager，并将 Context Manager 的 IPC 应答数据发送到服务客户端。在下面的讨论中，假定服务的客户端请求已经在服务注册阶段注册过。

（1）传递 Binder 数据的第 1 个阶段

该阶段类似注册服务时传递 Binder 数据的第 1 个阶段，Context Manager 进入待机状态，等待接收 IPC 数据。

（2）传递 Binder 数据的第 2 个阶段

在该阶段中 Binder Driver 的行为动作类似于注册服务时传递 Binder 数据的第 2 个阶段。但是发送 IPC 数据的进程不是 Service Server，而是服务客户端。

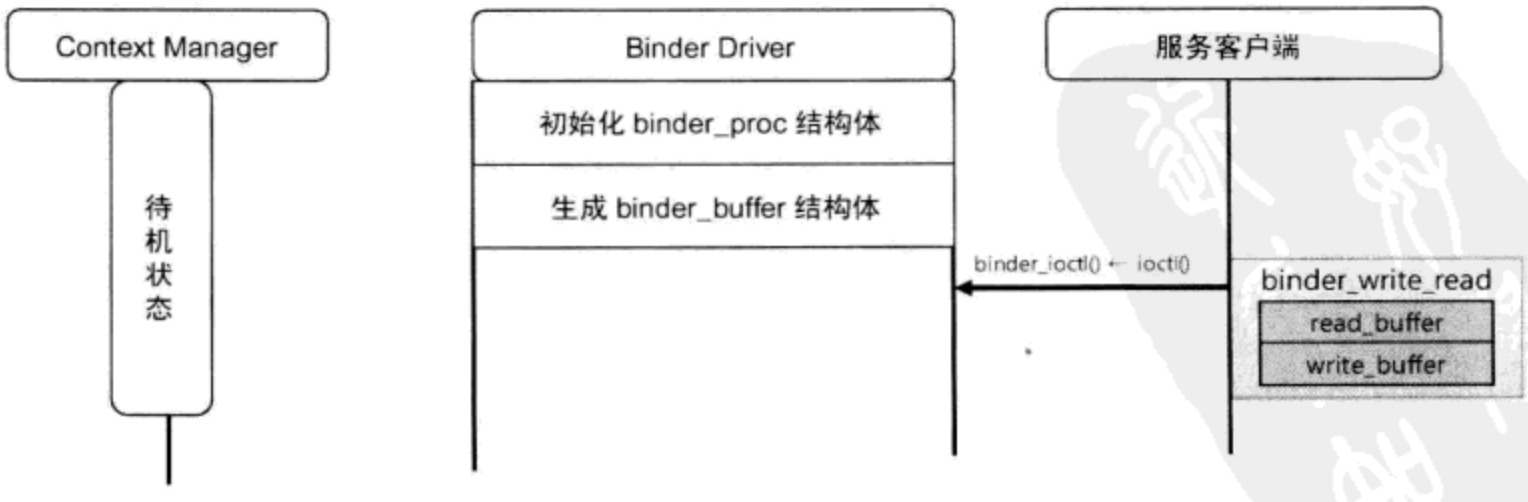


图 7-39 | 在服务检索中传递 Binder 数据的第 2 个阶段

服务客户端并不注册服务，所以在其 RPC 数据中不会存在 flat_binder_object 结构体。在服务客户端生成的 IPC 数据中包含 Context Manager 的 Handle（0）、RPC 代码（服务检索函数），以及请求的服务名称，如图 7-40 所示。

更多更新编程资源尽在 http://www.pin5i.com/

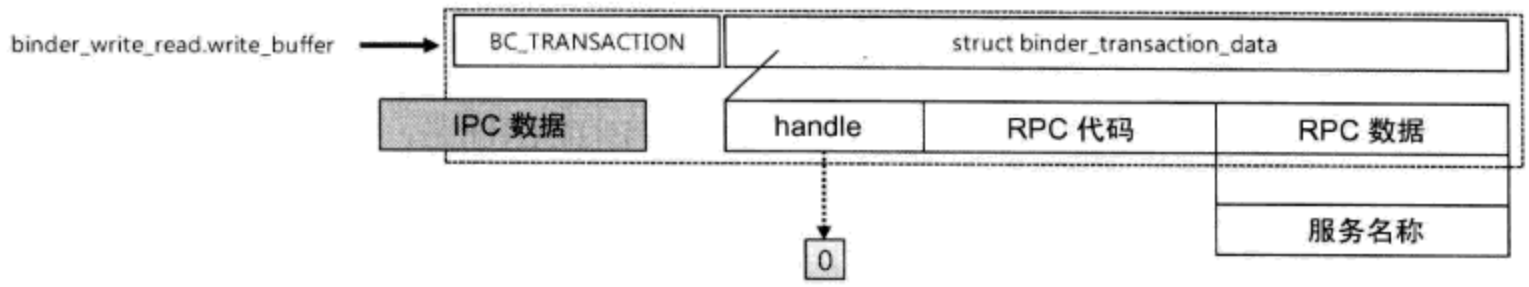


图 7-40 | 服务检索阶段的 IPC 数据结构

由于 IPC 数据中不包含 flat_binder_object 结构体，所以不会创建 binder_node 结构体，如代码 7-18 所示。除此之外的行为动作与服务注册阶段传递 Binder 数据的第 2 个阶段是一样的。

（3）传递 Binder 数据的第 3 个阶段

与 Service Server 类似，服务客户端既发送 IPC 数据，又接收 IPC 应答数据，它会使用 binder_write_read 结构体的 write_buffer 与 read_buffer 两个成员变量。同服务注册阶段的传递 IPC 数据的第 3 个阶段一样，服务客户端将进入待机状态中，等待接收 IPC 应答数据。

（4）传递 Binder 数据的第 4 个阶段

该阶段与服务注册阶段的传递 IPC 数据的第 4 个阶段类似。

Context Manager 接收 IPC 数据后，会在服务目录中生成 binder_object 结构体，该结构体中包含着与服务名称相对应的服务目录编号。并且，将结构体中的 type 变量设置为 BINDER_TYPE_HANDLE，生成 IPC 应答数据。在图 7-41 中，binder_object 结构体会被复制到 Binder Driver 的 flat_binder_object 中，用来查找 Binder 节点。在下一个阶段中，通过 binder_object 数据，将 Binder 节点传递给服务客户端。

图 7-41 | 服务检索阶段中的 IPC 应答数据结构

（5）传递 Binder 数据的第 5 个阶段

在该阶段中，服务客户端将获取所用服务的 Binder 节点。在服务注册阶段的代

更多更新编程资源尽在 http://www.pin5i.com/

码 7-18 中，flat_binder_object 结构体的 type 变量为 BINDER_TYPE_BINDER，继而生成 Binder 节点。

在服务检索阶段时，type 值变为 BINDER_TYPE_HANDLE，Binder 节点被传递给服务客户端，相关源代码如代码 7-26 所示。

```
fp = (struct flat_binder_object *)(t->buffer->data + *offp);
switch (fp->type) {
   case BINDER_TYPE_HANDLE:
   case BINDER_TYPE_WEAK_HANDLE: {
      struct binder_ref *ref = binder_get_ref(proc, fp->handle);          ❶
      if (ref->node->proc == target_proc) {
      }
   else {
      struct binder_ref *new_ref;
      new_ref = binder_get_ref_for_node(target_proc, ref->node);          ❷
      fp->handle = new_ref->desc;                                         ❸
   } break;
```

代码 7-26 | 传递 Binder 节点

❶ 调用 binder_get_ref()函数，其第一个参数 proc 为 Context Manager 的 binder_proc 结构体，第二个参数 fp->handle 是服务客户端要使用的服务编号。服务编号即 binder_ref 结构体中 desc 变量的值。将 desc 的值传入 binder_get_ref()函数后，即可获取与服务编号相对应的 binder_ref 结构体，此结构体由 Context Manager 持有。

❷ 调用 binder_get_ref_for_node()函数，其第一个参数 target_proc 是服务客户端的 binder_proc 结构体，第二个参数 ref->node 是❶中 binder_ref 结构体的一个 binder_node 结构体。在该函数内部，将为接收的 binder_node 结构体创建新的 binder_ref 结构体，并将其指向 binder_node 结构体。

过程❶与❷如图 7-42 所示。

每当 binder_get_ref_for_node()函数向服务客户端注册 Binder 节点时都会生成一个 binder_ref 结构体，同时 desc 的值递增。如图 7-42 所示，binder_ref 结构体是初次注册 Binder 节点时生成的，其 desc 值为 1。当再次注册新 Binder 节点时，binder_ref 结构体的 desc 值会变为 2。

❸ 修改 RPC 数据中 flat_binder_object 结构体内的服务编号，将其设置为当前 binder_ref 结构体的 desc 编号。而后在下一个阶段中，将包含修改后的 RPC 数据的 IPC 应答数据发送到服务客户端。

（6）传递 Binder 数据的第 6 个阶段

更多更新编程资源尽在 http://www.pin5i.com/

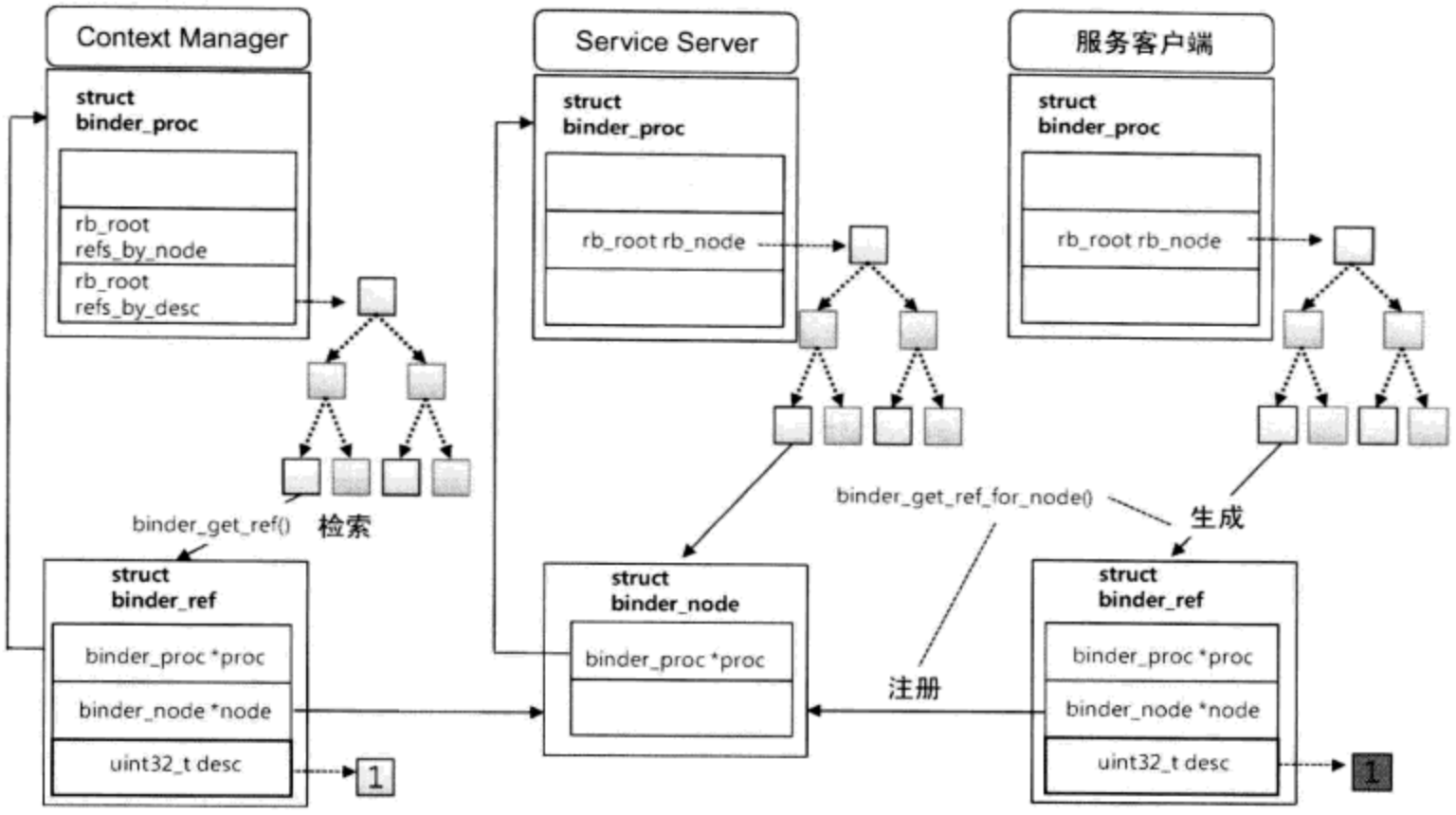


图 7-42 | Binder 节点检索及服务客户端注册

该阶段与服务注册阶段中传递 Binder 数据的第 6 个阶段类似。服务客户端使用 IPC 应答数据中的 Binder 节点编号，进入服务使用阶段。

服务使用阶段

在服务使用阶段，服务客户端与拥有服务的 Service Server 进行 IPC 通信。通过在服务检索阶段获得的 Binder 节点编号，向 Service Server 发送 IPC 数据。在服务使用阶段中，发送 IPC 数据的进程是服务客户端，而接收端是 Service Server。

下面分六个阶段来分析服务使用的整个过程。

（1）传递 Binder 数据的第 1 个阶段

在服务使用阶段，Service Server 进入待机状态，等待接收 IPC 数据。

（2）传递 Binder 数据的第 2 个阶段

在服务使用阶段中，IPC 数据包含与所用服务函数相关的 RPC 代码与 RPC 数据。如前所述，在服务检索的传递 Binder 数据的第 6 个阶段中已经获取了服务的 Binder 节点编号，IPC 数据中的 Handle 即是该 Binder 节点编号。IPC 数据结构如下图所示。

该阶段与服务注册的传递 Binder 数据的第 2 个阶段类似，但是与 binder_transaction() 函数查找接收端 binder_proc 结构体的过程有所不同。

更多更新编程资源尽在 http://www.pin5i.com/

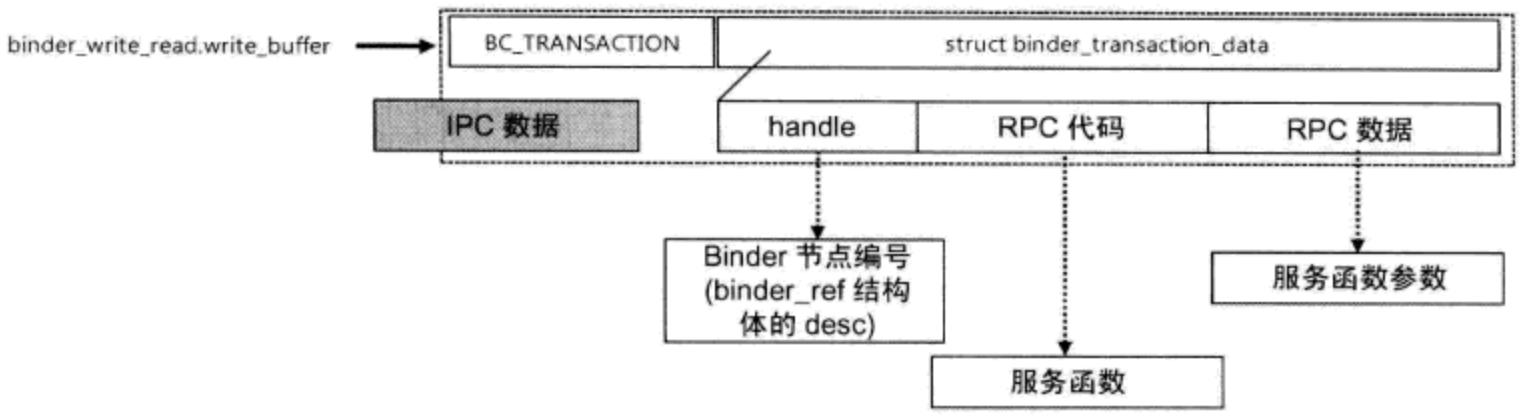


图 7-43 | IPC 数据结构

```
if (tr->target.handle) {
   struct binder_ref *ref;
   ref = binder_get_ref(proc, tr->target.handle);          ←❶
   target_node = ref->node;
   } else {
      target_node = binder_context_mgr_node;
   }
   target_proc = target_node->proc;                        ←❷
```

代码 7-27 | binder_transaction()函数的 Binder 寻址

在服务注册阶段，Handle 值为 0，查找到的是 Context Manager 的 binder_proc 结构体；而在服务使用阶段则使用 target.handle，查找对应进程的 binder_proc 结构体，如代码 7-27 所示。

❶ 调用 binder_get_ref()函数，查找 desc 值为指定 Handle 的 binder_ref 结构体。查找到指定的结构体后，将 binder_ref 结构体的 node 指向 Service Server 的 binder_node 结构体。

❷ binder_node 结构体的 proc 指向 Service Server 的 binder_proc 结构体，即接收 IPC 数据的进程。

从服务的使用阶段开始，RPC 数据中不再存在 flat_binder_object 结构体，不需要生成或检索 Binder 节点。其余的过程与服务注册或服务检索的传递 Binder 数据的第 2 个阶段相同。

（3）传递 Binder 数据的第 3 个阶段

该阶段与服务注册阶段或服务使用阶段类似，但接收 IPC 数据的不是 Context Manager，而是 Service Server。Service Server 调用 IPC 数据中 RPC 代码所指的函数，并将 RPC 数据作为参数传入该函数中。此后的过程与 Binder 服务注册的传递 Binder 数据的第 4、5、6 阶段相同，在此不再赘言。

更多更新编程资源尽在 http://www.pin5i.com/

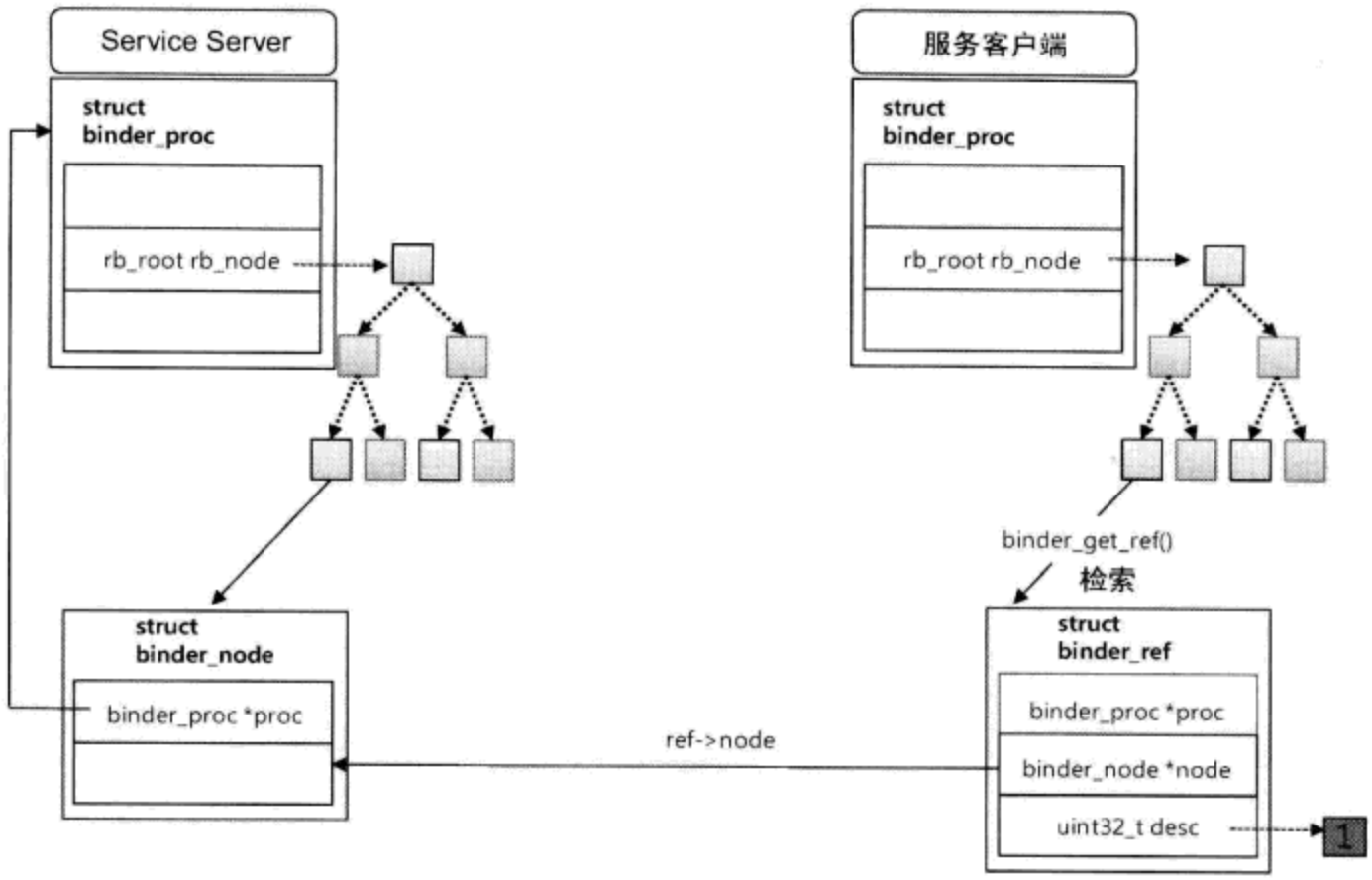


图 7-44 | 使用 Binder 节点编号，获取 Service Server 的 binder_proc 结构体

以上我们根据 Binder Driver 的动作构成，分析了服务客户端使用 Service Server 的服务的过程。下一节我们将学习 Context Manager 处理 IPC 数据的过程。

7.4 Context Manager

Context Manager 对应的进程为 servicemanager，它先于 Service Server 与服务客户端运行，首先进入接收 IPC 数据的待机状态，处理来自 Service Server 或服务客户端的请求。查看 init.rc 脚本文件，可以看到 Context Manager 在 mediaserver 与 system_ server 之前先运行了，如图 7-45 所示。

图 7-45 显示的是 Android 启动后运行中的各种进程，其中矩形框内标出的即是 Context Manager，它对应于/system/bin/servicemanager 进程。

每当 Service Server 注册服务时，Context Manager 都会把服务的名称与 Binder 节点编号注册到自身的服务目录中，该服务目录通过根文件系统下的/system/service 程序即可查看。

service 程序以 IPC 应答数据的形式接收 Context Manager 服务目录中的服务名称，并将接收到的服务名称输出到画面中。

更多更新编程资源尽在 http://www.pin5i.com/

```
PID USER      VSZ STAT COMMAND
  1 0         276 S    /init
  2 0           0 SW<  [kthreadd]
  3 0           0 SW<  [ksoftirqd/0]
  4 0           0 SW<  [watchdog/0]
  5 0           0 SW<  [events/0]
  6 0           0 SW<  [khelper]
 12 0           0 SW<  [suspend]
100 0           0 SW<  [kblockd/0]
107 0           0 SW<  [kseriod]
111 0           0 SW<  [kmmcd]
135 0           0 SW   [pdflush]
136 0           0 SW   [pdflush]
137 0           0 SW<  [kswapd0]
139 0           0 SW<  [aio/0]
140 0           0 SW<  [nfsiod]
273 0           0 SW<  [mtdblockd]
274 0           0 SW<  [spi1.0]
314 0           0 SW<  [rpciod/0]
513 0        2592 S    /busybox/bin/sh
514 0         680 S    /system/bin/logcat -c
515 1000      808 S    /system/bin/servicemanager
516 0         840 S    /system/bin/vold
518 0         668 S    /system/bin/debuggerd
519 1001     4372 S    /system/bin/rild
520 0       73932 S    zygote /bin/app_process -Xzygote /system/bin --zygote
521 1013    17440 S    /system/bin/mediaserver
523 1002     1168 S    /system/bin/dbus-daemon --system --nofork
525 0         808 S    /system/bin/installd
526 0        1268 S    /sbin/adbd
540 1000      187m S    system_server
586 1001      113m S    com.android.phone
592 10000     122m S    android.process.acore
610 10010     103m S    com.android.mms
650 10002     106m S    android.process.media
671 10004     103m S    com.android.alarmclock
694 10008   98496 S    com.android.calendar
713 0        2592 R    ps
```

图 7-45 | Context Manager（servicemanager 进程）

```
# service list
service list
Found 44 services:
0       phone: [com.android.internal.telephony.ITelephony]
1       iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
2       simphonebook: [com.android.internal.telephony.IIccPhoneBook]
.
.
40      SurfaceFlinger: [android.ui.ISurfaceComposer]
41      media.camera: [android.hardware.ICameraService]
42      media.player: [android.hardware.IMediaPlayerService]
43      media.audio_flinger: [android.media.IAudioFlinger]
#
```

代码 7-28 | Context Manager（servicemanager）中的服务目录

Context Manager 的运行

Context Manager 与其他 Android 服务不同，它采用 C 语言编写，以便使其与 Binder Driver 紧密衔接。Context Manager 的源码在/frameworks/base/cmds/servicemanager 目录

更多更新编程资源尽在 http://www.pin5i.com/

下的 service_manager.c 文件中。Context Manager 的 main()函数大致可以分为三部分，分别为 binder_open()函数（用来打开 Binder Driver 并创建 IPC 数据接收 Buffer）、binder_become_context_manager()函数（注册特殊节点，即 0 号 Binder 节点），以及 binder_loop()函数（用来不断接收 IPC 数据）。

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
    bs = binder_open(128*1024);                    ←❶
    if (binder_become_context_manager(bs)){}       ←❷
    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);               ←❸
    return 0;
}
```

代码 7-29 | Context Manager 的 main()函数

❶ 调用 binder_open()函数，将引起 open()与 mmap()函数调用，调用 open()函数打开 Binder Driver，而调用 mmap()函数则生成接收 IPC 数据的 Buffer。Context Manager 使用大小为 128KB 的 Buffer 来接收 IPC 数据。

❷ 与 Service Server 和服务客户端不同，这是 Context Manager 特有的语句，用于访问 Binder Driver，并将自身的 Binder 节点设置为 0 号节点。在 binder_become_context_manager()函数中，仅有一条调用 ioctl()函数的语句，如代码 7-30 所示。

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

代码 7-30 | binder_become_context_manager()函数

当向 Binder Driver 传递 BINDER_SER_CONTEXT_MGR 的 ioctl 命令后，Binder Driver 将通过在 binder_ioctl()函数中调用 binder_new_node()函数，生成编号为 0 的 Binder 节点，并将其注册到 binder_context_mgr_node 全局变量中，该变量将在代码 7-15❶中使用。

```
case BINDER_SET_CONTEXT_MGR:
    binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
```

代码 7-31 | 生成编号为 0 的 Binder 节点

更多更新编程资源尽在 http://www.pin5i.com/

❸ Context Manager 在 binder_loop()函数中使用 for 语句不断等待接收 IPC 数据，当接收到 IPC 数据时，binder_parse()函数就会被调用，如代码 7-32 所示。

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
    struct binder_write_read bwr;
    unsigned readbuf[32];
    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;          ❶
        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);  ❷
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);  ❸
    }
}
```

代码 7-32 | binder_loop()函数

❶ 在 7.3.3 “Binder Driver 函数分析”一节中，我们已经看过 binder_write_read 结构体。该语句用来注册 binder_write_read 结构体的用户空间 Buffer，与图 7-17 的 read_buffer 相对应。

❷ 调用 ioctl()函数，进入待机状态，等待接收 IPC 数据，与服务注册或服务检索中传递 Binder 数据的第 1 个阶段类似。

❸ 当返回 Binder Driver 的 binder_ioctl()函数时，执行该语句。Context Manager 通过 Binder Driver 接收 IPC 数据，而后调用 binder_parse()函数分析 IPC 数据。

```
int binder_parse(struct binder_state *bs, struct binder_io *bio,
    ➥ uint32_t *ptr, uint32_t size, binder_handler func)
{
    while (ptr < end) {
        uint32_t cmd = *ptr++;
        switch(cmd) {
            case BR_TRANSACTION: {
                struct binder_txn *txn = (void *) ptr;
                binder_dump_txn(txn);
                if (func) {
                    res = func(bs, txn, &msg, &reply);      ❶
                    binder_send_reply(bs, &reply, txn->data, res);
                }
                break;
    }
```

代码 7-33 | binder_parse()函数

更多更新编程资源尽在 http://www.pin5i.com/

Context Manager 首先解析 IPC 数据中的 Binder 协议，在服务注册过程中，所使用的 Binder 协议为 BR_TRANSACTION，因而继续向下执行❶。

❶ func()函数是在代码 7-29 的❸中注册过的 svcmgr_handler()函数。在 Binder Driver 传递的 IPC 数据中，Binder 协议之后即是 binder_transaction_data 结构体。func()函数使用 binder_txn 结构体，该结构体与 binder_transaction_data 结构体是一样的。而后再以该结构体作为参数调用 svcmgr_handler()函数，如代码 7-34 所示。

```
int svcmgr_handler(struct binder_state *bs, struct binder_txn *txn,
    ➥ struct binder_io *msg, struct binder_io *reply)
{
   struct svcinfo *si;
   switch(txn->code) {
      case SVC_MGR_GET_SERVICE:                    ←❶
      case SVC_MGR_CHECK_SERVICE:
         s = bio_get_string16(msg, &len);
         ptr = do_find_service(bs, s, len);
         if (!ptr)
            break;
         bio_put_ref(reply, ptr);
         return 0;
      case SVC_MGR_ADD_SERVICE:                    ←❷
         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;
   }
   return 0;
}
```

代码 7-34 | svcmgr_handler()函数

Context Manager 的主要任务包括服务注册与服务检索两大任务，具体由 svcmgr_handler()函数根据 RPC 代码执行。

❶ 服务客户端在检索服务时执行该部分代码。服务客户端首先通过 RPC 数据传递服务名称，从 do_find_service()函数自身的服务列表中获取带有指定名称的服务编号，而后 bio_put_ref()函数生成 binder_object 结构体，该结构体将被包含到 IPC 应答数据的 RPC 数据中。这个过程与服务检索的传递 Binder 数据的第 4 个阶段类似。然后通过代码 7-33 中❶的 binder_send_reply()函数，将 IPC 应答数据传递给 Binder Driver。

❷ 当 Service Server 注册服务时执行该部分代码。Context Manager 调用 do_add_service()函数将 IPC 数据的 RPC 数据包含的服务名称与 Binder 节点编号注册到自身的服务目录中。而后调用 binder_send_reply()函数，将 IPC 应答数据传递

更多更新编程资源尽在 http://www.pin5i.com/

给 Binder Driver。

7.5 小结

Binder 由 Service Server（含有 Android 的服务）、Service Client（使用服务的客户端）、Context Manager（确定服务的位置），以及 Binder Driver 四大部分组成。就像组件一样，Android 提供的各种功能被细分成一些具有特定功能的服务。并且，使用服务的进程通过 IPC 数据与服务进行相关作用。

Binder 就像连接 Android 各种服务的环扣，处于中心的是 Binder Driver。在使用框架服务或应用程序服务的过程中，Android 提供了很多接口，隐藏了 Binder Driver 的复杂的行为动作。Android 使用更高层次的抽象隐藏了处于底层的 Binder Driver，并且提供了丰富的接口，让我们使用起来更加方便。

在下一章中，将学习 Service Manager 与 Service Framework，它们都在更高的层次上对底层的 Binder Driver 进行了抽象。若想进一步理解 Binder，必须学习与这些抽象层相关的知识。迄今为止，我们学习的有关 Binder Driver 行为动作的知识都是为了帮助大家掌握抽象层中的 IPC 机制而准备的，为进一步理解 Android Framework 中的内部动作打下基础。

更多更新编程资源尽在 http://www.pin5i.com/

第 8 章

Android Service Framework

Android 系统服务提供系统最基本、最核心的功能，比如设备控制、地理位置信息提供、定时设置等。这些系统服务都是使用 Android 的 Service Framework 实现的，Android Service Framework 是 Android 系统的重要组成部分。探索 Service Framework 内部原理，理解并掌握 Service Framework 的运作方式，有助于我们开发出各种新功能，例如视频通话、DMB 服务等非基本的系统服务。在本部分，将重点学习 Android Service Framework，分析各个构成元素，理解其原理与运行机制，以便开发出各种服务[1]。

8.1 服务框架（Service Framework）

Android Service Framework 是一系列类的集合，它用来开发运行在 Android 平台上的各种服务。Service Framework 提供了设计精良、复用度高的服务设计与实现，开发者可以使用 Service Framework 提供的各种类与接口，快速开发出可靠优良的服务。

在使用 Service Framework 开发服务时，开发者只要把主要精力集中在开发服务的核心功能上即可，服务核心功能之外的部分 Service Framework 会帮我们处理。例如，在开发服务时，服务注册、服务 Binder IPC 等功能都由系统提供，开发者不需要实现这些功能。总之，开发者在使用 Service Framework 开发服务时，只要集中精力开发服务要提供的主要功能即可。

之前，读者可能听说过 Android Application Framework 这个词，但什么是 Service Framework，有些读者可能不太明白。在 Android 平台中，Service Framework 大致由两大部分构成，一部分是使用 C++语言编写的本地服务框架[2]，另一部分是使用 Java 语言编写的 Java 服务框架。既然已经存在“应用程序框架”这一术语了，为什么还要使用“服务框架”（Service Framework）这个术语呢？

首先在说明由 C++语言编写的系统服务以及由 Java 编写的系统服务时，有必要将两

1 若无特殊说明，服务默认指系统服务。在指应用程序服务的时候，均以“应用程序服务”字样注明。

2 在第 6 章“Android 服务概要”中已作出说明。

更多更新编程资源尽在 http://www.pin5i.com/

者合在一起来说明。在使用应用程序框架（Application Framework）[1]时，由 Java 实现的服务将被包含进去，而由 C++实现的服务则不会被包含进去，所以在开发服务时需要一个通用的术语来指代所使用的框架（Framework）。并且，应用程序框架中构成要素十分庞大，为了说明的方便，也需要一个术语将实现服务所需要的所有框架元素表达出来。

其次，在“服务框架”（Service Framework）中包含了所有框架（Framework）的特征。框架（Framework）有两个代表性的特征，一是通过扩展而非修改框架添加新的功能；二是使用框架实现的应用程序控制流由框架而非程序自身控制。在 Android 平台中，许多服务通过扩展而非修改服务框架来实现，并且服务的动作流也是由服务框架所决定，所实现的服务只是按照内部定义的方式进行运作。若开发者采用任意的方法开发服务，那么开发出的服务将无法在系统中正常运行，即只有使用 Android 提供的类与已经定义好的方法开发，才能够开发出可以在系统中正常运行的服务。

图 8-1 描述了“服务框架”（Service Framework）在 Android 平台堆栈各部分相对应的情形。图中处于中间粗矩形边框内的部分即是“服务框架”，其中在粗矩形框的下半部分，位于本地库中的部分是本地服务框架（Native Service Framework）。而矩形的上半部分，位于“应用程序框架（Application Framework）”中的部分则是 Java 服务框架（Java Service Framework），它通过中间的 JNI 本地函数与本地服务框架（Native Service Framework）进行交互。

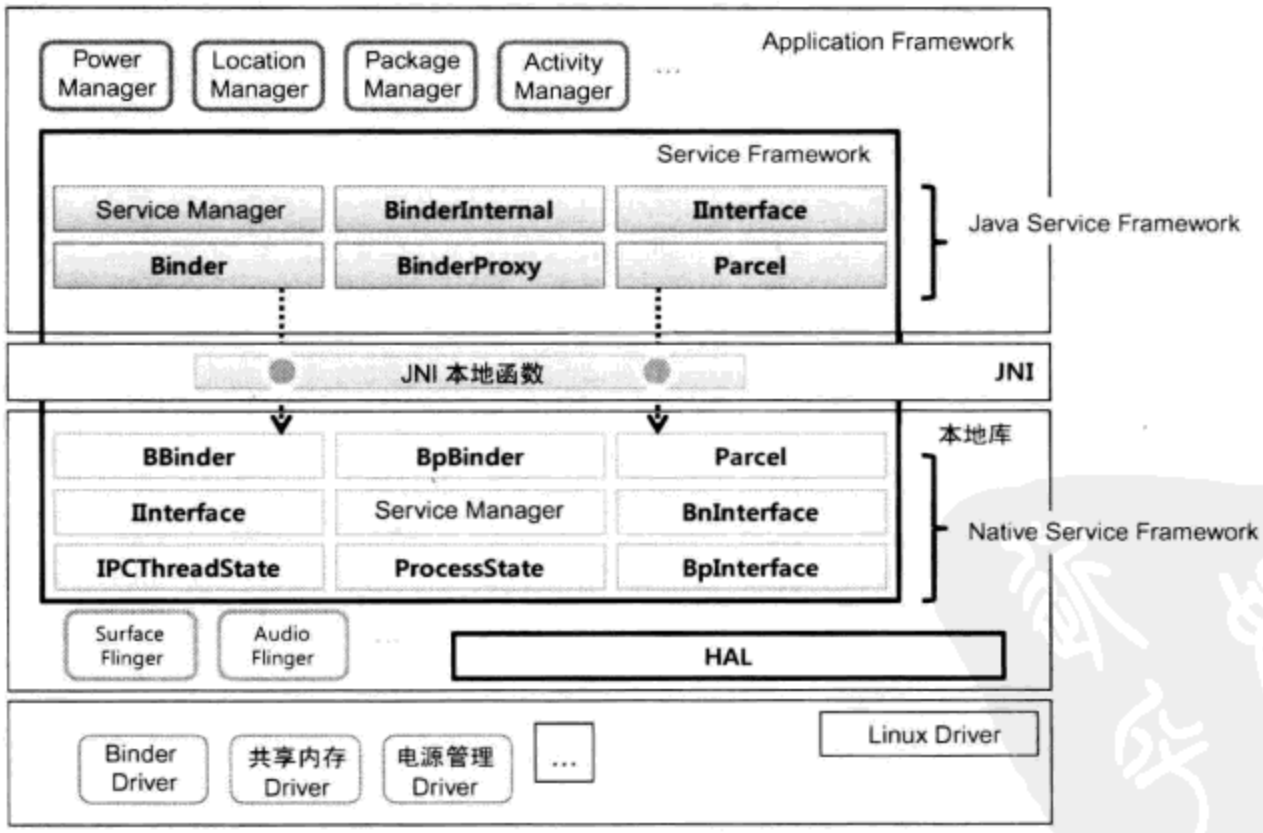


图 8-1 | 服务框架（Service Framework）

1 指位于 Android 平台堆栈中的应用程序框架。Android 平台堆栈大致由 Linux 内核、本地库、应用程序框架，以及应用程序等构成。

更多更新编程资源尽在 http://www.pin5i.com/

应用程序框架（Application Framework）中的电源管理（Power Manager）等 Java 系统服务都是使用 Java 服务框架实现的，而 Surface Flinger 这类本地系统服务都是使用本地服务框架实现的。但是 HAL（Hardware Abstraction Layer）、Linux 设备驱动、Dalvik 虚拟机等都不包含在“服务框架”（Service Framework）中。

图 8-2 显示了 Android Service Framework 提供的 4 种主要功能。

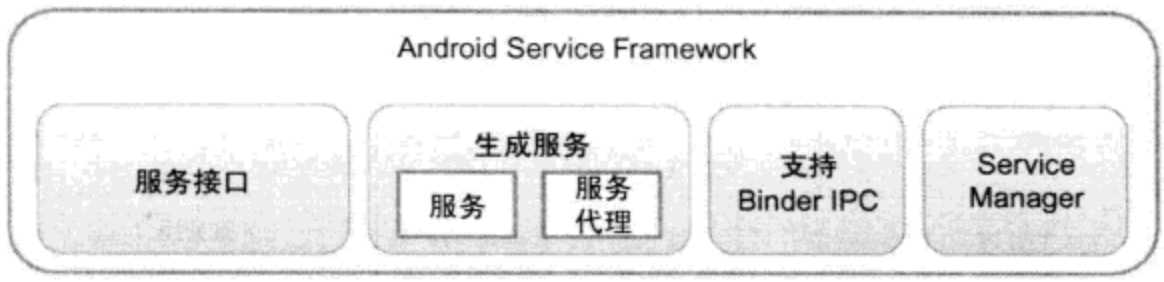


图 8-2 | Android Service Framework 提供主要功能

- 服务接口：在服务接口中以函数的形式声明服务要提供的功能，而后服务或服务代理实现该接口中的服务函数，提供接口中定义的一系列服务。
- 服务生成：服务生成功能支持服务与服务代理的生成。服务实现服务接口中定义的函数（服务 Stub 函数），服务代理帮助实现远程调用（服务代理函数）服务 Stub 函数。Service Framework 提供服务的位置信息（Service Handle），以便服务代理请求服务，也提供一系列的方法以帮助服务接收服务请求。
- Binder IPC 处理：Binder IPC 处理功能提供 Binder IPC 的生成以及与 Binder Driver 通信的功能，以便支持服务与服务使用者之间的 Binder IPC 通信。
- 服务管理（Service Manager）：提供向系统注册或检索服务的功能。

由于 C++层的“服务框架”（Service Framework）与 Java 层的“服务框架”（Service Framework）的运行方式与实现方法都不相同，所以本书在讲解“服务框架”（Service Framework）时将分开讲解。本章先讲解“本地服务框架”（Native Service Framework）[1]部分，关于 Java 服务框架的讲解将在本书第 10 章中进行。

8.2 服务框架（Service Framework）的构成

在 8.1 节中，已经简单地介绍过 Android“服务框架”相关的内容。本节将继续讲解组成“服务框架”（Service Framework）的各种类，以及了解它们之间是如何交互作用的。

---

1 若无特殊说明，本章中的“服务框架”（Service Framework）均指“本地服务框架”（Native Service Framework）。在涉及 Java 服务框架部分，都将使用“Java 服务框架”字样明确标明。

更多更新编程资源尽在 http://www.pin5i.com/

8.2.1 各层构成元素的配置

在分析“服务框架”（Service Framework）时，将 Binder IPC 各抽象层的作用以及相应的类联系起来，有助于我们更容易地理解“服务框架”中的各个类。表 8-1 描述了“服务框架”各个层的组成元素以及它们各自的作用。为了便于说明，假设有一个名称为“FooService”的服务，它是使用“服务框架”实现的。

表 8-1 服务框架各抽象层的构成元素

		服务客户端（Service Client）	Service Server
服务层	配置	服务接口 IFooService foo()	服务接口 IFooService foo() 服务 FooService foo()
	功能	■ FooService 服务的共用接口 ■ 提供 FooService 服务的 foo()函数接口	■ FooService 服务的共用接口 ■ 运行 FooService 服务的 foo()函数
RPC 层	配置	服务协议 BpFooService foo()	服务 Stub BnFooService onTransact()
	功能	■ 支持 Binder RPC ■ 调用 foo()函数，转换成 RPC 数据*	■ 支持 Binder RPC ■ 分析 RPC 代码与数据，调用 foo()服务的 Stub 函数
IPC 层	配置	Binder IPC 处理 BpBinder IPCThreadState	Binder IPC 处理 BpBinder IPCThreadState
	功能	■ 支持 Binder IPC ■ 将 RPC 代码与数据转换成 IPC 数据**，传递给 Binder Driver	■ 支持 Binder IPC ■ 接收 Binder IPC 数据，转换成 RPC 代码与数据

*Binder RPC 数据由 RPC 代码（包含调用 foo()函数的信息）与 RPC 数据（所调函数的参数）组成。

**向 Binder RPC 数据中添加目的地信息（BpBinder，Service Handle）、Binder Driver 信息（IPCThreadState，Binder 协议），即转换成 Binder IPC 数据。

首先服务客户端与 Service Server 在 Service Framework 的构成元素配置上存在不同。在服务层的服务客户端有服务接口，而在 Service Server 中不仅有服务接口，还有相应的服务。在 RPC 层的服务客户端只有名称为“BpFooService”的服务代理，而在 Service Server 中有 BnFooService 服务 Stub。在 IPC 层，服务客户端与 Service Server 的构成元素都是一样的。

更多更新编程资源尽在 http://www.pin5i.com/

其次，服务客户端与 Service Server 中构成元素的功能有所不同。在服务层的服务客户端服务使用者调用 IFooService foo()函数，而在 Service Server 端运行的是 FooService 的 foo()服务的 Stub 函数。在 RPC 层的服务客户端存在有 BpFooService 服务代理，并且将调用 foo()代理函数的信息转换成 RPC 代码与数据。但在 Service Server 端有 BnFooService 服务的 Stub，分析接收到的 RPC 代码与数据，并调用 foo()服务的 Stub 函数。在 IPC 层，服务客户端与 Service Server 的构成元素都是一样的，但是服务客户端会将 RPC 代码与数据转换成 IPC 数据，而 Service Server 则将 IPC 数据转换成 RPC 代码与数据。

8.2.2 各层构成元素间的相互作用

“服务框架”（Service Framework）中的各个类之间是如何相互作用的呢？首先服务客户端与 Service Server 中使用的 Service Framework 的类与位于同一层的对等类产生相互作用（同层相互作用）。比如，在服务客户端与 Service Server 中，位于 RPC 层的服务代理与服务 Stub 类相互对应，并相互作用；位于 IPC 层的 BpBinder 与 BBinder 类相互对应并发生作用。各层相互作用的类如下。

第一，位于服务层中的服务接口使服务使用者与服务使用统一的接口进行相互作用。比如，在图 8-3 的 FooService 服务中，IFooService 担当接口角色，向两端提供统一的 foo()接口。

图 8-3 | IFooService 服务接口

第二，处于 RPC 层的服务代理与服务 Stub 支持 Binder RPC 操作。从 Binder RPC 观点来看，服务代理将函数调用信息转换为 RPC 代码与数据，服务 Stub 分析服务代理发送来的 RPC 代码与数据，而后调用服务 Stub 函数。例如，在 8-4 中服务使用者调用 BpFooService 服务代理的 foo()代理函数，TRANSACTION_FOO 代码被发送到 BnFooService 服务 Stub 中，而后服务 Stub 调用 TRANSACTION_FOO 代码指定的 foo()Stub 函数。

第三，位于 IPC 层的 BpBinder 类与 BBinder 类支持 Binder IPC 操作。在 Binder IPC 中，当调用 BpBinder 的 transact()函数时，BBinder 的 transact()函数即被调用，同时传递 RPC 代码与数据。BpBinder 与 BBinder 两个类都提供支持 Binder RPC 的服务函数。例如，在图 8-5 中，调用服务客户端的 BpBinder 的 pingBinder()函数，通过 transact()函数，BBinder 的 pingBinder()函数将被调用。

更多更新编程资源尽在 http://www.pin5i.com/

图 8-4 | BpFooService 服务代理与 BnFooService 服务 Stub 间的相互作用

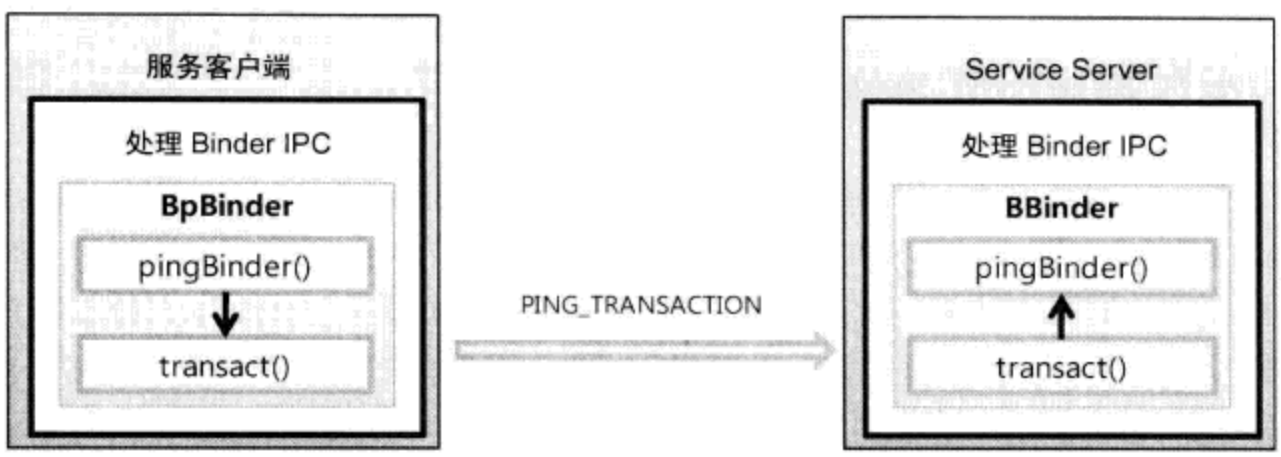


图 8-5 | BpBinder 与 BBinder 的相互作用

若服务函数不是“服务框架”（Service Framework）提供的基本函数，则需要重新定义 BBinder 的 onTransact()函数，具体实现 Service Stub 类，以便调用 Service Stub 函数。例如，在图 8-6 中，BnFooService 是一个 Service Stub 类，它继承于 BBinder 类并重新定义了 onTransact()函数，实现对服务的 foo()函数的调用。

请看图 8-6，该图描述了 BnFooService 类的作用。

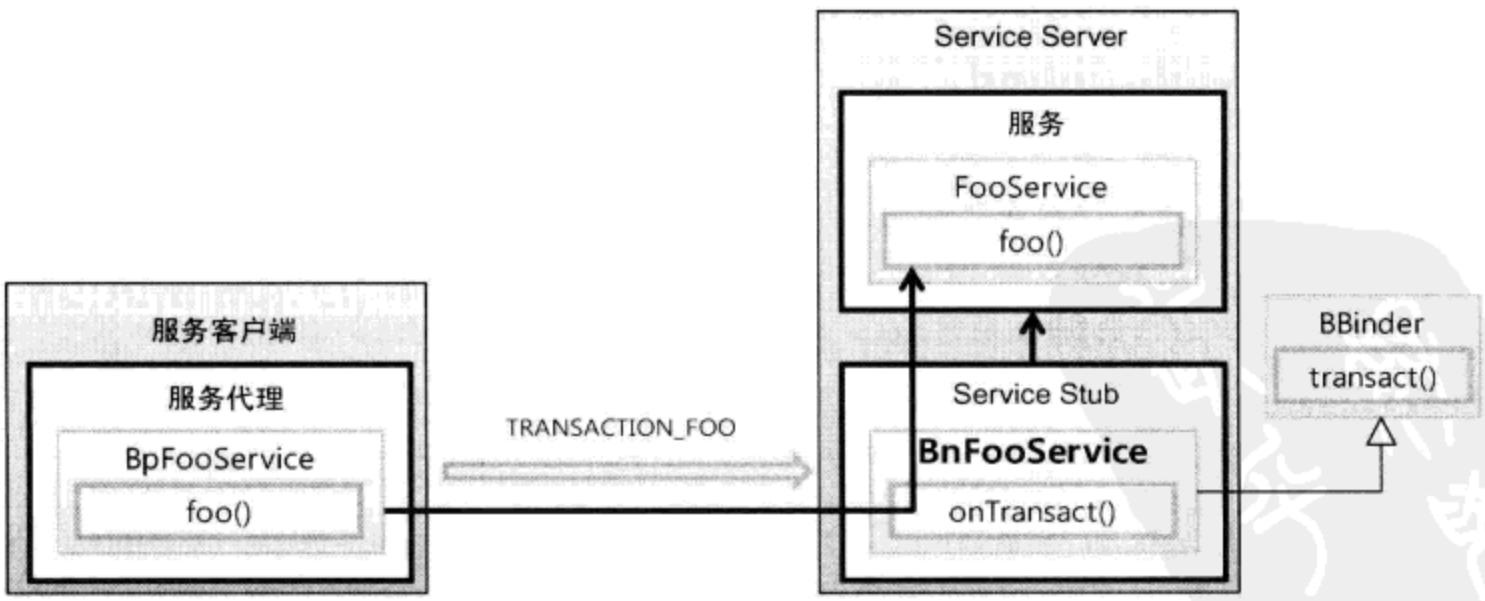


图 8-6 | BnFooService 类的作用

除了 BpBinder 与 BBinder 类之外，Service Framework 还提供了 IPCThreadState 类，用来支持 Binder IPC 操作。如图 8-7 所示，IPCThreadState 类在服务客户端与 Service

更多更新编程资源尽在 http://www.pin5i.com/

Server 都位于 IPC 层。服务客户端与 Service Server 两个进程在传递 Binder IPC 数据时实际是通过 Binder Driver 进行的，IPCThreadState 使用 Binder 协议命令与 Binder Driver 进行通信。

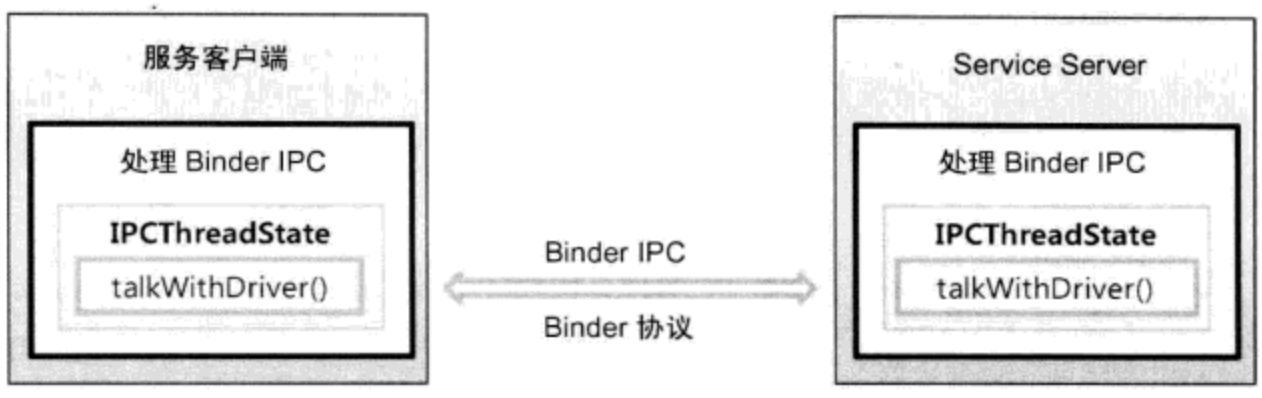


图 8-7 | IPCThreadState 相互作用

服务客户端与 Service Server 在使用 Service Framework 中的类时，这些类不仅与对等层上的对应的类相互作用，在服务客户端与 Service Server 内部也会与其他类发生作用（在垂直方向上）。例如，在服务客户端中位于 RPC 层的 BpFooService 服务代理类将与 IPC 层的 BpBinder 类发生相互作用。

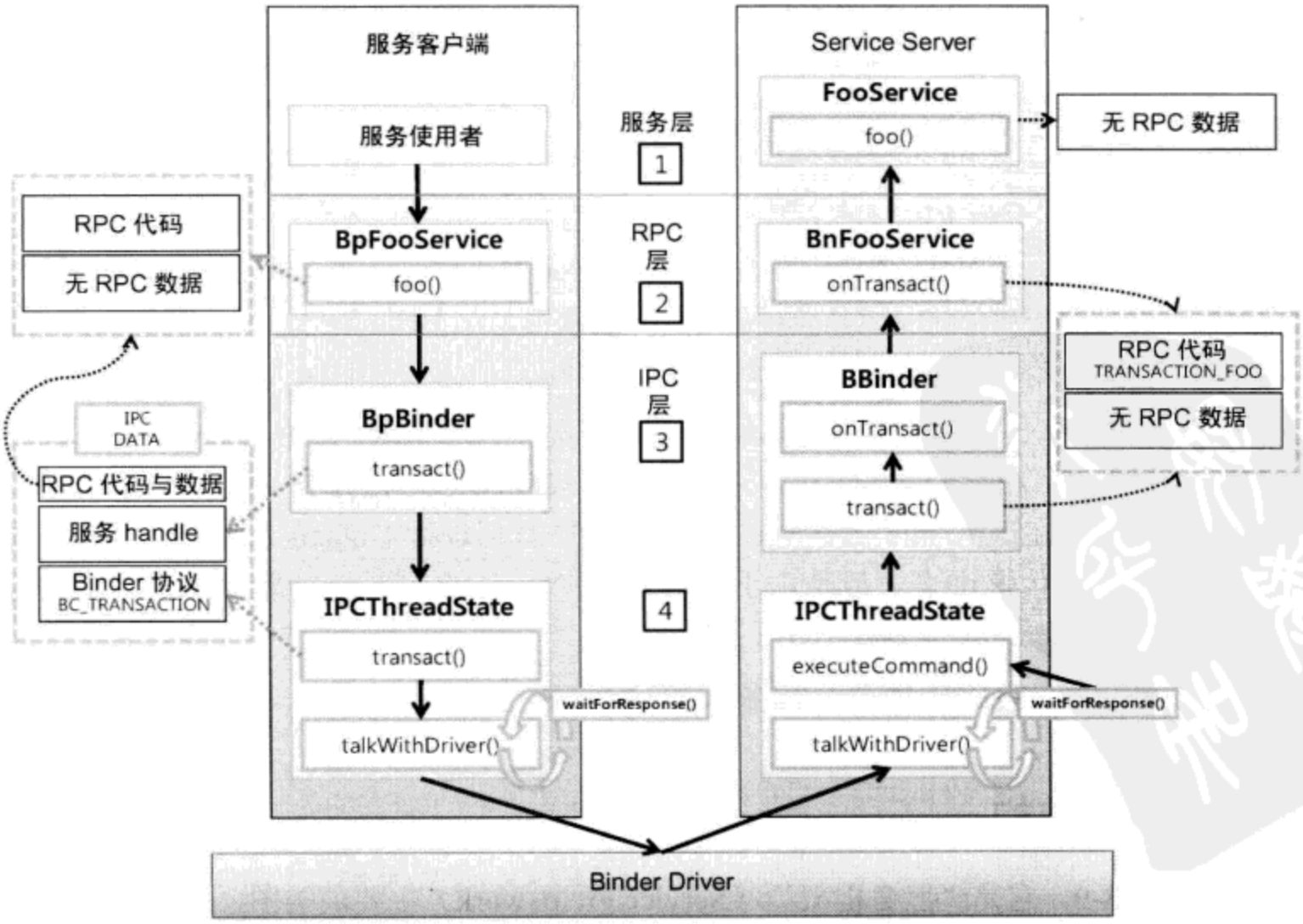


图 8-8 | 服务客户端与 Service Server 内部元素间的相互作用

更多更新编程资源尽在 http://www.pin5i.com/

在调用 FooService 服务的 foo()函数时，服务客户端与 Service Server 内部 Service Framework 组成元素将发生什么样的作用呢？图 8-8 描述了 Service Framework 组成元素相互作用的过程。首先，我们看一下在服务客户端中发生的行为。

1. 服务使用者调用 foo()函数，引起 BpFooService 的 foo()代理函数的调用。
2. BpFooService 的 foo()代理函数将 RPC 代码（TRANSACTION_FOO）与参数转换成 RPC 数据，而后调用 BpBinder 类的 transact()函数，传递 RPC 代码与数据。
3. BpBinder 的 transact()函数向 RPC 代码与数据中添加 FooService 服务的 Service Handle 信息后，以参数的形式传递给 IPCThreadState 类的 transact()函数。
4. IPCThreadState 类的 transact()函数向接收到的数据中添加 Binder 协议（BC_TRANSACTION），生成 IPC 数据后，将其传递给 Binder Driver。

而在 Service Server 端将按照相反的顺序调用相关函数，函数调用顺序如下。

4. Binder Driver 向 IPCThreadState 传递 Binder IPC 数据。IPCThreadState 在接收到数据后，会调用自身的 executeCommand()函数分析数据。若数据中包含的 Binder 协议为 BR_TRANSACTION，则调用 BBinder 类的 transact()函数，并且将来自服务客户端的 RPC 代码与数据作为参数传入 transact()函数。
3. BBinder 分析 RPC 代码，若所调用的服务函数非基本函数，则调用 onTransact()函数。由于 BnFooService 继承于 BBinder 并重定义了 onTransact()函数，所以 BnFooService 的 onTransact()函数将被调用。从 BBinder 传递过来的数据将再次以参数的形式传递给 onTransact()函数。
2. BnFooService 的 onTransact()函数分析接收到的 RPC 数据，调用 RPC 代码（TRANSACTION_FOO）所指向的 FooService 的名称为 foo()的 stub 函数。一般地，在调用 stub 函数时，将先对 RPC 数据进行 Unmarshalling 处理，再作为参数传递进去。但是由于 foo()这个 Stub 函数不带参数，所以不用进行 Unmarshalling 处理。
1. 执行 FooService 的 foo()服务 Stub 函数。

8.2.3 类的结构

请看图 8-9，它是“服务框架”（Service Framework）主要的类图，在类图中简单地标出了各个类的主要成员函数与成员变量，这些成员函数与变量都是用来实现系统

更多更新编程资源尽在 http://www.pin5i.com/

服务的。其中虚线框中的类都被定义在同一个头文件中，其他的类名称与头文件名称[1]一致。主要类的作用如图 8-9 所示。

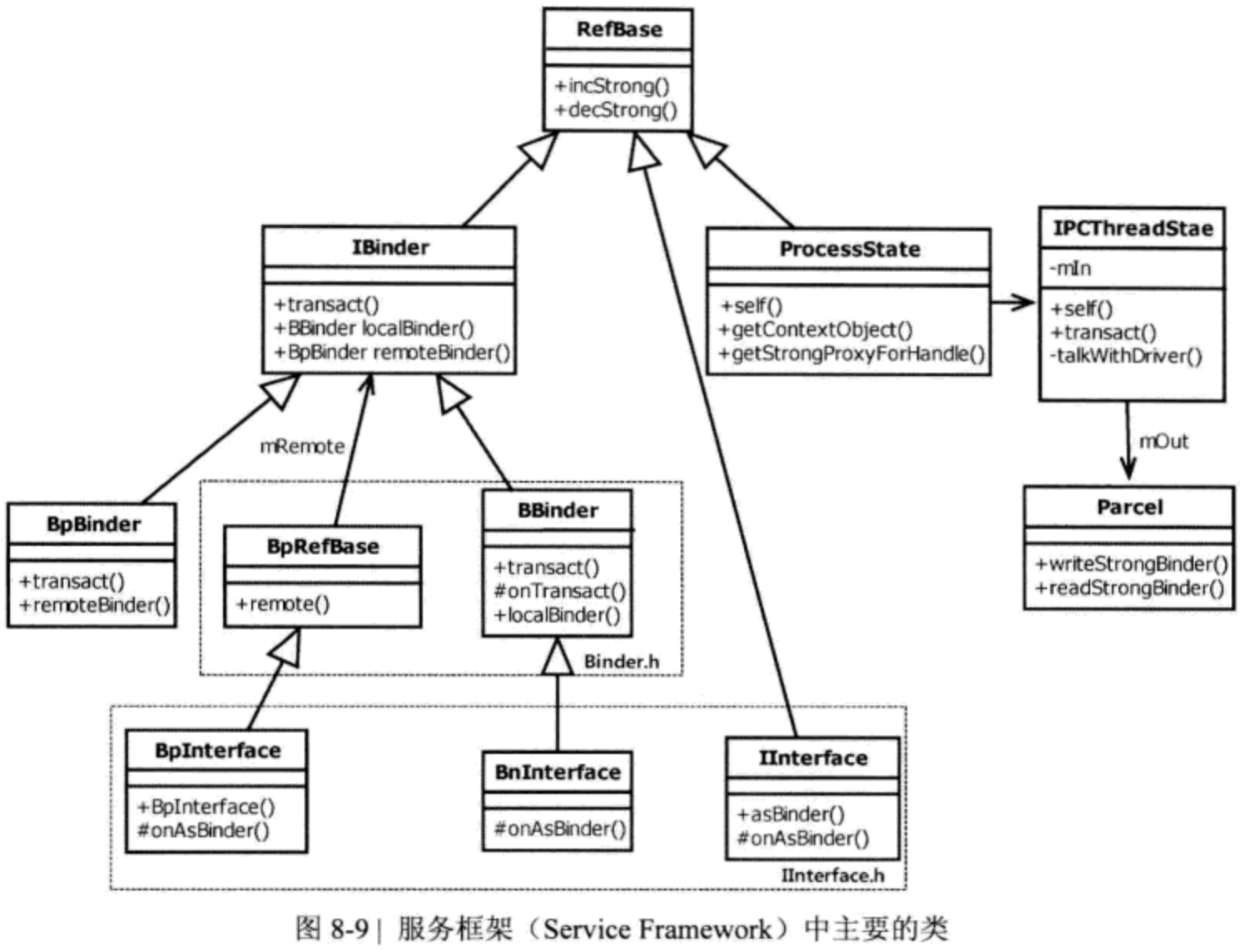


图 8-9 | 服务框架（Service Framework）中主要的类

- IBinder、BBinder、BpBinder：IBinder 类是对 Android Binder 的抽象，它有 BBinder、BpBinder 两个子类。BBinder 类负责接收 RPC 代码与数据，并在 Binder Driver 内部生成 Binder 节点。BpBinder 类保存着目标服务的 Handle 信息，用于 Binder Driver 查找 Service Server 的 Binder 节点的过程中。
- IInterface、BnInterface、BpInterface：IInterface 类提供类型变换功能，将服务或服务代理类转换为 IBinder 类型。实际的类型转换是由 BnInterface、BpInterface 两个类完成，BnInterface 将服务类转换成 IBinder 类型，而 BpInterface 则将服务代理类转换成 IBinder 类型。在通过 Binder Driver 传递 Binder 对象时，必须进行类型转换，比如在向系统注册服务时，需要先将服务类转换成 IBinder，再将其传递给 Context Manager。

1 相关类的头文件在/frameworks/base/include/binder/目录中，源文件在/framework/base/libs/binder 目录中。

更多更新编程资源尽在 http://www.pin5i.com/

- ProcessState、IPCThreadState：ProcessState 类用来管理 Binder Driver，IPCThreadState 类用来支持服务客户端、Service Server 与 Binder Driver 间的 Binder IPC 通信。
- Parcel：在服务与服务代理间进行 Binder IPC 时，Parcel 类负责保存 Binder IPC 数据。Parcel 类可以处理的数据有 C 语言的基本数据结构、基本数据结构数组、Binder 对象、文件描述符等。

在开发服务时需要实现的有服务层中的服务接口、服务类，以及 RPC 层中的服务代理、服务 stub。“服务框架”（Service Framework）各个层包含的类，以及开发者需要实现的类如下所示，其中用中文标出的部分是服务开发者使用“服务框架”（Service Framework）中的类直接实现的。

- 服务层：IInterface[1]、BnInterface、BpInterface、BpRefBase、服务接口、服务类
- RPC 层：服务代理类、服务 Stub 类
- IPC 层：BBinder[2]、BpBinder、IPCThreadState、ProcessState、Parcel

至此，我们已经了解过 Android Service Framework 有关的内容，但仅止于一些概念的讲解说明，仅学习这些内容很难准确地理解 Service Framework。因此，下面将通过对 Audio Flinger 系统服务源码的分析，进一步学习有关 Service Framework 运行方式的内容。Audio Flinger 是 Android 的一个系统服务，它负责音频输出。

8.3 运行机制

开发运行在系统中的服务时，若想提高开发水平，开发出类似内置于 Android 的系统服务，就必须对组成 Service Framework 的各种类有充分的掌握与理解。幸运的是，Android 是一个开源项目，我们通过分析系统服务的源码就能了解 Service Framework 中的各个类是如何使用、如何运作的。下面以 Audio Flinger 系统服务为例来分析类的构成以及各个类是如何协作的，Audio Flinger 是 Android 系统典型的系统服务之一，它负责 Android 系统中的声音处理。

图 8-10 是组成 Audio Flinger 系统服务的类图。Audio Flinger 系统服务由 IAudioFlinger 服务接口、BnAudioFlinger 服务 stub、AudioFlinger 服务、BpAudioFlinger 服务代理构成。如图所示，IAudioFlinger 服务接口处于类图的中间，其左侧是实现服务所需要的

1 IInterface.h，BnInterface、BpInterface 类一同被定义在头文件中。

2 Binder.h，BpRefBase 类一同被定义在头文件中。

更多更新编程资源尽在 http://www.pin5i.com/

各种类，右侧则是实现服务代理所需要的各种类。

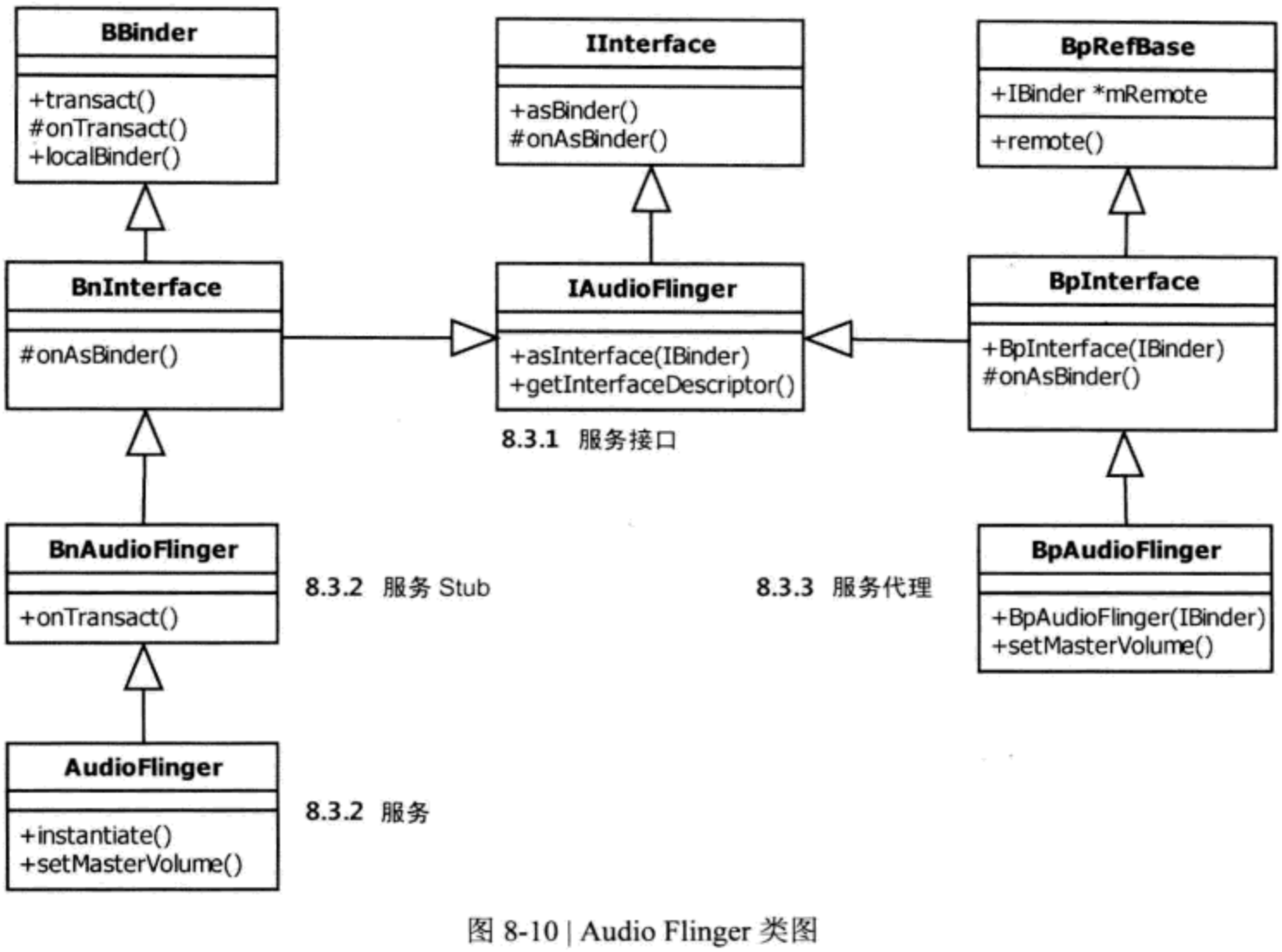


图 8-10 | Audio Flinger 类图

8.3.1 服务接口

服务函数在服务接口中声明，并分别在服务与服务代理中实现。例如，在 IAudioFlinger 服务接口中有一个 setMasterVolume()服务函数的声明，该服务函数分别在 AudioFlinger 服务与 BpAudioFlinger 服务代理中实现。在服务接口中还存在一些函数，这些函数能够把服务接口转换成 IBinder 类型，或把 IBinder 类型转换成服务接口。

例如，在 IInterface 类中，asBinder()函数用来将服务接口类型转换为 IBinder 类型，而 asInterface()函数则用来将 IBinder 类型转换为服务接口类型。

在 Audio Flinger 服务中存在名称为 IAudioFlinger 的服务接口，如代码 8-1[1]所示，并且服务提供的功能由 IAudioFlinger 的成员函数声明。

---

1 为了帮助大家理解 Service Framework 的构成元素是如何运作的，源码中记载了一个在 IAudioFlinger 中声明的 setMaster Volume()服务函数。

更多更新编程资源尽在 http://www.pin5i.com/

```
class IAudioFlinger : public IInterface                    ←❶
{
public:
    DECLARE_META_INTERFACE(AudioFlinger);                  ←❷

    /* set the audio hardware state. This will probably be used by
     * the preference panel, mostly.
    */
    virtual     status_t    setMasterVolume(float value) = 0;  ←❸

}
```

代码 8-1 | IAudioFlinger.h[1]-IAudioFlinger 类的主要源码

IAudioFlinger 类的主要特征如下：

- IAudioFlinger 类继承自 IInterface 类。
- 类的内部使用 DECLARE_META_INTERFACE 宏。
- 声明 IAudioFlinger 的 setMasterVolume()成员函数（服务函数）。

代码 8-1❶：IInterface 类的作用

IInterface 类是 IAudioFlinger 类的父类，它的主要功能之一就是支持服务与 IBinder 类型间的显式转换。即 IInterface 类通过 asBinder()函数将 IAudioFlinger 类型转换为 IBinder 类型。在进行 Binder IPC 通信时，IBinder 对象要被保存到 RPC 数据中并传递给 Binder Driver，所以在实现服务时需要进行类型转换。例如，在向系统注册 Audio Flinger 服务时，Service Manager 首先要将 BBinder 类型的服务对象包含到 RPC 数据中，再将其传递给 Context Manager，在这一过程中需要先把服务转换成 IBinder 类型后，再传递给 Binder Driver。

IAudioFlinger 类型是如何被转换成 IBinder 类型的呢？下面看一下源代码。如代码 8-2 所示，IInterface 类中仅声明了一个名称为 asBinder()的 public 成员函数，该函数的返回值为 IBinder 指针。

```
class IInterface : public virtual RefBase
{
public:
    IInterface();
    sp<IBinder>         asBinder();
    sp<const IBinder>   asBinder() const;

protected:
    virtual                     ~IInterface();
```

1 /frameworks/base/include/media/IAudioFlinger.h

更多更新编程资源尽在 http://www.pin5i.com/


```
    virtual IBinder*            onAsBinder() = 0;
};
```

代码 8-2 | IInterface.h[1]-IInterface 类的源码

代码 8-3 是 asBinder()函数的实现代码。若存在 IInterface 类型的接口，就会调用 onAsBinder()函数，该成员函数是一个虚拟函数，子类中必须实现该虚拟函数。

```
sp<IBinder> IInterface::asBinder()
{
    return this ? onAsBinder() : NULL;
}
```

代码 8-3 | IInterface.cpp[2]-asBinder()成员函数

图 8-11 是 IInterface、BnInterface、BpInterface 等类的关系图。由类图可知，IAudio Flinger 类继承了 IInterface 类，BnInterface 与 BpInterface 类继承了 IAudioFlinger 类，所以当调用 IInterface 的 onAsBinder()函数时，BnInterface 的 onAsBinder()函数或 BpInterface 的 onAsBinder()函数即会被调用。

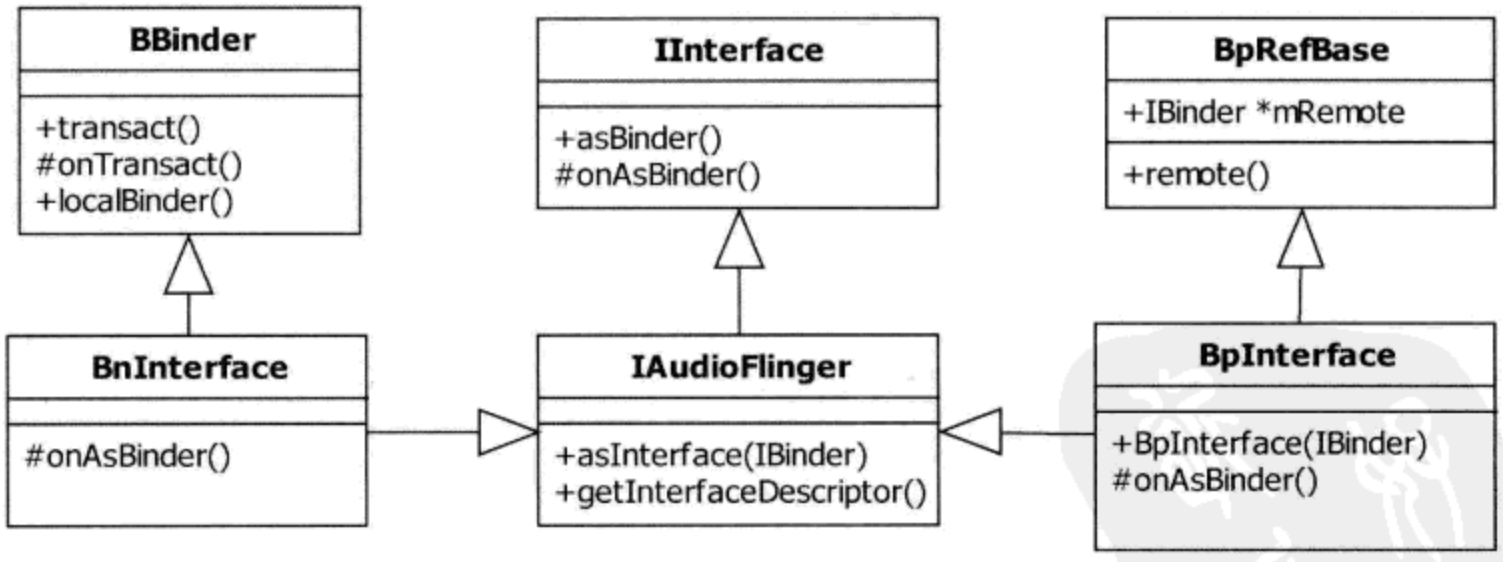


图 8-11 | BnInterface 类与 BpInterface 类的继承关系

表 8-2 描述了 BnInterface 与 BpInterface 两个类中 onAsBinder()成员函数代码的不同。BnInterface 类的 onAsBinder()函数的返回值为 IBinder 类型，即返回类自身的地址。如图 8-11 所示，BnInterface 类继承了 BBinder 类，而 BBinder 类又继承了 IBinder 类，所以 BnInterface 类是 IBinder 类[3]的子类。与此不同，BpInterface 类的 onAsBinder()函

1 /frameworks/base/include/binder/IInterface.h

2 /frameworks/base/libs/binder/IInterface.cpp

3 BBinder 与 BpBinder 两个类都是 IBinder 类的子类，请参考图 8-9 的类图关系。

更多更新编程资源尽在 http://www.pin5i.com/

数将调用 remote()函数，而调用 remote()函数又会引起 BpRefBase 类的 remote()函数的调用，返回 mRemote 变量，mRemote 变量是 IBinder 类型的。

表 8-2　比较 BnInterface 与 BpInterface 两个类中的 onAsBinder()函数

类名	onAsBinder()函数的源码

BnInterface	template<typename INTERFACE>
	IBinder* BnInterface<INTERFACE>::onAsBinder()
	{
	　　return this;
	}

BpInterface	template<typename INTERFACE>
	inline IBinder* BpInterface<INTERFACE>::onAsBinder()
	{
	　　return remote();
	}

代码 8-1❷：DECLARE_META_INTERFACE 宏的作用

代码 8-4 是 DECLARE_META_INTERFACE（AudioFlinger）宏声明部分。代码 8-5 是宏预处理代码，在代码中有 descriptor 数据成员、asInterface()、getInterfaceDescriptor()成员函数的声明。其中，asInterface()是类型变换函数，它接收一个 IBinder 类型的参数，将其转换为 IAudioFlinger 类型，功能与 IInterface 的 asBinder()函数正好相反。

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

代码 8-4 | IInterface.h-DECLARE_META_INTERFACE 宏

```
static const String16 descriptor;
static sp< IAudioFlinger > asInterface(const sp<IBinder>& obj);
virtual const String16& getInterfaceDescriptor() const;
IAudioFlinger ();
virtual ~IAudioFlinger ();
```

代码 8-5 |IInterface.h-DECLARE_META_INTERFACE 宏预处理代码

在 DECLARE_META_INTERFACE 宏预处理后，我们可以查看 IAudioFlinger 的成员函数 asInterface()的实现。代码 8-6 是预处理后的代码，asInterface()的参数是 BBinder 或 BpBinder，它们都是 IBinder 类型的。若参数传递过来的对象不是 NULL，则将返回 AudioFlinger 服务的对象或 BpAudioFlinger 服务代理对象。

更多更新编程资源尽在 http://www.pin5i.com/

若传递过来的参数类型是 BpBinder，则 IBinder 的 queryLocalInterface()函数将被调用。由于 IBinder 的 queryLocalInterface()函数会返回 NULL 值，因此 asInterface()函数将创建 BpAudioFlinger 实例，并将创建好的实例返回。

```
sp<IAudioFlinger> IAudioFlinger::asInterface(const sp<IBinder>& obj)
{
    sp<IAudioFlinger> intr;
    if (obj != NULL) {
        intr = static_cast<IAudioFlinger*>(
            obj->queryLocalInterface(IAudioFlinger::descriptor).get());
        if (intr == NULL) {
            intr = new BpAudioFlinger(obj);
        }
    }
    return intr;
}
```

代码 8-6 | IAudioFlinger.cpp[1]中的 asInterface()函数

若参数值为 BBinder 类型，则 BnInterface 的 queryLocalInterface()函数被调用。BnInterface 的 queryLocalInterface()函数重新定义并实现了 IBinder 的 queryLocalInterface()函数，如代码 8-7 所示。若参数值为 IAudioFlinger 接口的名称（android.media.IAudioFlinger），则返回 BnInterface 的实例。在代码 8-6 中已经以参数的形式传递了 IAudioFlinger 的 descriptor 变量的值，所以 queryLocalInterface()函数将返回 AudioFlinger 服务的实例。

```
inline sp<IInterface> BnInterface<IAudioFlinger>::queryLocalInterface(
        const String16& _descriptor)
{
    if (_descriptor == IAudioFlinger::descriptor) return this;
    return NULL;
}
```

代码 8-7 | IInterface.h-BnInterface 类的 queryLocalInterface()函数

在代码 8-6 中使用 static_cast 运算符将 queryLocalInterface()函数返回的服务对象强制转换成 IAudioFlinger 类型，而后由 asInterface()函数返回。

简言之，asInterface()函数由 DECLARE_META_INTERFACE（AudioFlinger）与 IMPLEMENT_META_ INTERFACE（AudioFlinger）宏生成，该函数接收 IBinder 类型的参数，通过判断参数的具体类型（BBinder 类型或 BpBinder 类型），调用相应函数，返回 IAudioFlinger 类型的服务或服务代理，即将 IBinder 类型转换为 IAudioFlinger 类

1 frameworks/base/media/libmedia/IAudioFlinger.cpp

更多更新编程资源尽在 http://www.pin5i.com/

型后返回。

代码 8-1❸：服务函数声明

在服务接口中，服务与服务代理声明了所使用的服务函数。在 IAudioFlinger 中声明了许多虚拟函数，这些函数对应 Audio Flinger 提供的各种功能，例如 setMasterVolume()函数，它提供系统音量设置功能。

小结

前面我们分析了 Audio Flinger 服务的 IAudioFlinger 类，概括起来，IAudioFlinger 服务接口提供如下三种功能。

第一，在 IAudioFlinger 与 IBinder 类型间进行类型转换。在将 IAudioFlinger 类型转换为 IBinder 类型时，需要调用 IInterface 类的 asBinder()函数，而后调用 BnInterface 或 BpInterface 的 onAsBinder()函数进行处理。

第二，调用 IAudioFlinger 类的 asInterface()函数，可以将 IBinder 类型转换为 IAudioFlinger 类型。该函数由 Service Framework 提供的 DECLARE_META_INTERFACE 与 IMPLEMENT_ META_ INTERFACE 宏进行声明与实现。

第三，在 IAudioFlinger 类中声明了多种纯虚函数，对应 Audio Flinger 服务提供的各项功能。并且，继承了该类的服务与服务用户全部通过相同的接口来提供或使用服务。

图 8-12 描述了 IAudioFlinger 与 IBinder 类间进行类型转换的过程。

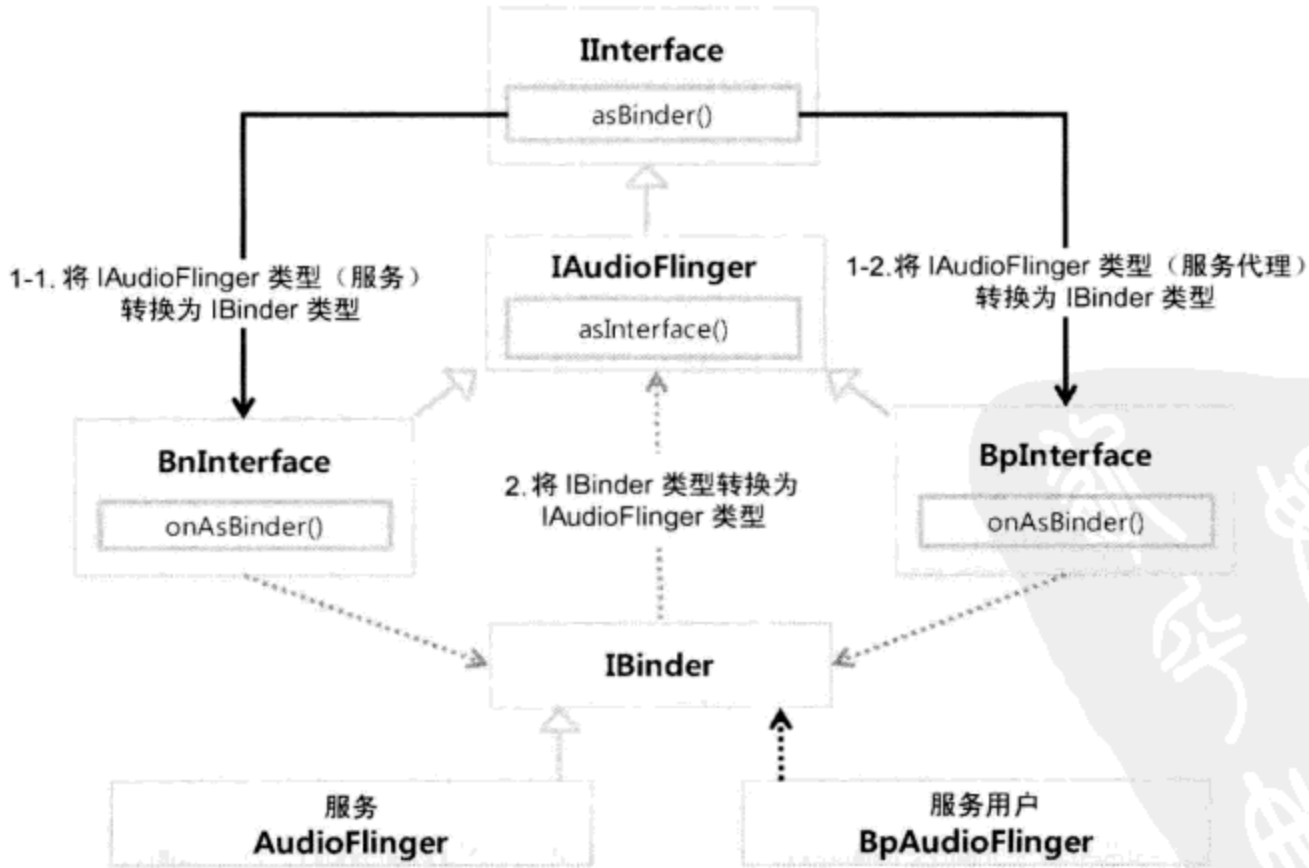


图 8-12 | IAudioFlinger 与 IBinder 类间的类型转换

更多更新编程资源尽在 http://www.pidc.com/

1. IAudioFlinger→IBinder

   调用 IInterface 类的 asBinder()函数，将 IAudioFlinger 类型转换为 IBinder 类型。若具体类型为服务，则调用 BnInterface 类的 onAsBinder()函数，将 IAudioFlinger 类型转换为 BBinder 类型，如图 8-12(1-1)所示，若具体类型为服务代理，则调用 BpInterface 类的 onAsBinder()函数，将 IAudioFlinger 类型转换为 BpBinder 类型，如图 8-12(1-2)所示。

2. IBinder→IAudioFlinger

   调用 IAudioFlinger 类的 asInterface()函数，将 IBinder 类型转换为 IAudioFlinger 类型。若参数传递过来的 IBinder 类型为服务，则返回 AudioFlinger 实例，否则创建并返回 BpAudioFlinger 实例。

8.3.2 服务

服务函数在服务类中被实现，例如用于控制系统声音的 setMasterVolume()函数，它在 AudioFlinger 类中被具体实现。并且，还需要服务 stub 类，以便通过 Binder RPC 运行服务类的服务 stub 函数。下面分析一下 Audio Flinger 服务的 AudioFlinger 类，以及 BnAudioFlinger 服务 stub 类。

AudioFlinger 类基继承了 IAudioFlinger 类，它具体实现 Audio Flinger 的服务功能。代码 8-8 是 AudioFlinger 类的主要代码。

```
class AudioFlinger :  public BnAudioFlinger          ←❶
{
public:
   // IAudioFlinger interface
   virtual     status_t    setMasterVolume(float value);     ←❷
   static void instantiate();      ←❸
};
```

代码 8-8 | AudioFlinger.h[1]-AudioFlinger 类的主要代码

代码 8-8❶：BnAudioFlinger 服务 stub 的功能

BnAudioFlinger 服务 stub 类负责分析 Audio Flinger 服务中使用的 RPC 代码，并调用 AudioFlinger 类的相应函数。请看代码 8-9，在 BnAudioFlinger 类中仅声明了一个 onTransact()函数，该函数的第一个参数接收 RPC 代码，第二个参数接收 RPC 数据。

1 /frameworks/base/libs/audioflinger/AudioFlinger.h

更多更新编程资源尽在 http://www.pin5i.com/

```
class BnAudioFlinger : public BnInterface<IAudioFlinger>
{
public:
   virtual status_t    onTransact( uint32_t code,
                                   const Parcel& data,
                                   Parcel* reply,
                                   uint32_t flags = 0);
};
```

代码 8-9 | BnAudioFlinger 类[1]

BnInterface 类是 BnAudioFlinger 类的父类，它被声明为类的模板，继承了 IAudioFlinger 与 BBinder 两个类。图 8-13 是一个类图，描述了 BnAudioFlinger、AudioFlinger、BnInterface 等各个类之间的继承关系。

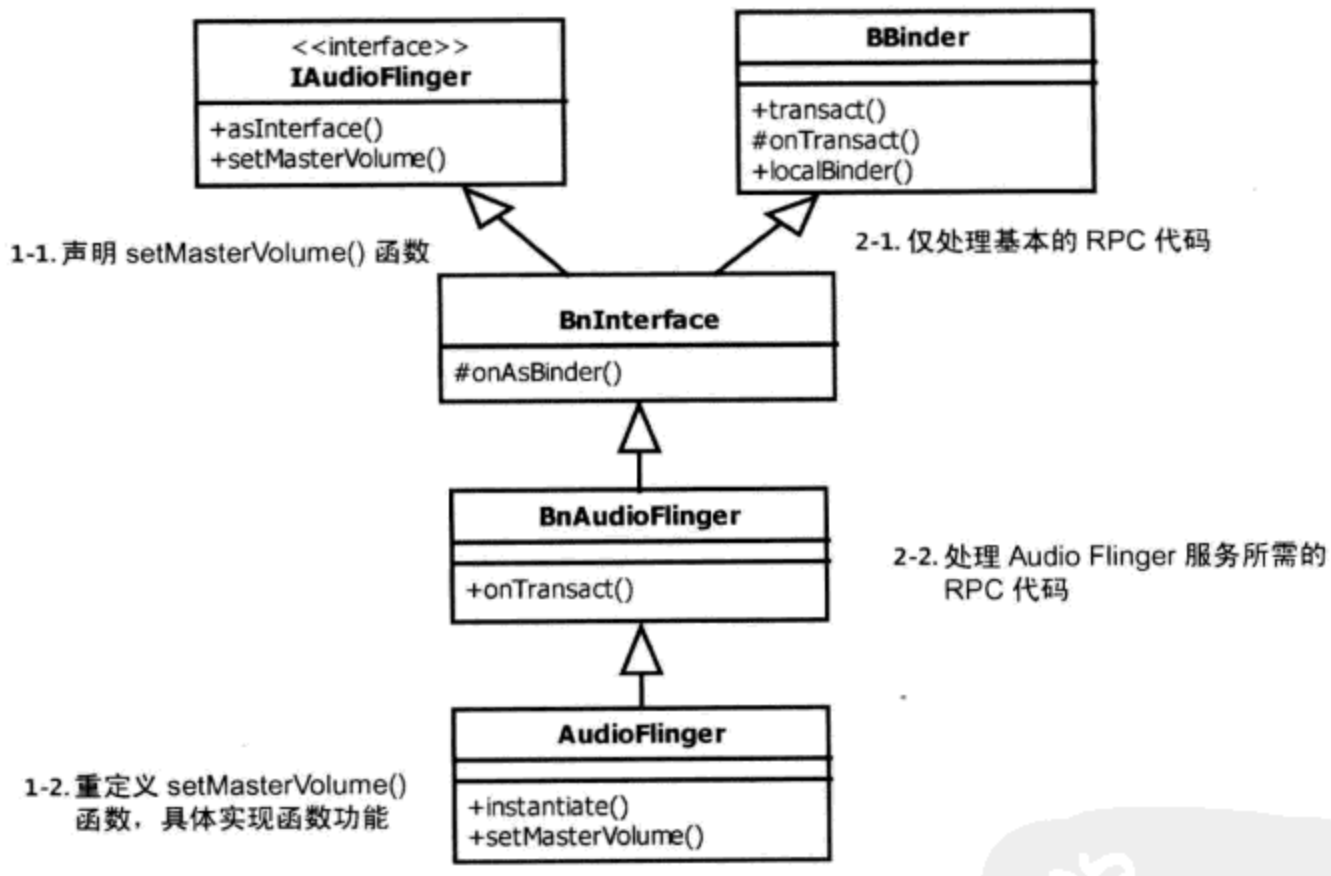


图 8-13 | AudioFlinger 服务的类图

从图 8-13 的类图中，可以看出 setMasterVolume()函数在 IAudioFlinger 接口中被声明，如图 8-13（1-1）所示。setMasterVolume()函数是纯虚函数，在子类中必须被重新定义。在处于类图最下端的 AudioFlinger 类中，setMasterVolume()函数被重新定义，并且实现了音量调整的功能，如图 8-13（1-2）所示。

如图 8-13（2-1）所示，在处于类图顶端的 BBinder 类中有一个 transact()函数，该函数负

1 /frameworks/base/include/media/IAudioFlinger.h

责接收 RPC 代码与数据，并调用 onTransact()函数，onTransact()函数在子类 BnAudioFlinger 中被重新定义，如图 8-13（2-2）所示。由于 onTransact()函数只对 INTERFACE_TRANSACTION 与 DUMP_TRANSACTION RPC 代码作基本的处理，所以需要添加处理 RPC 代码的功能，AudioFlinger 服务使用这些 RPC 代码。BnAudioFlinger 的 onTransact()函数分析 Audio Flinger 服务中使用的 RPC 代码，调用 AudioFlinger 类中相应的服务 stub 函数。

比如，在图 8-14 中当服务用户调用 setMasterVolume()服务代理函数将系统音量调整到中央时，BnAudioFlinger 服务 stub 就会分析包含在 RPC 代码与数据中的 SET_MASTER_VOLUME RPC 代码，并调用 setMasterVolume()服务 stub 函数。

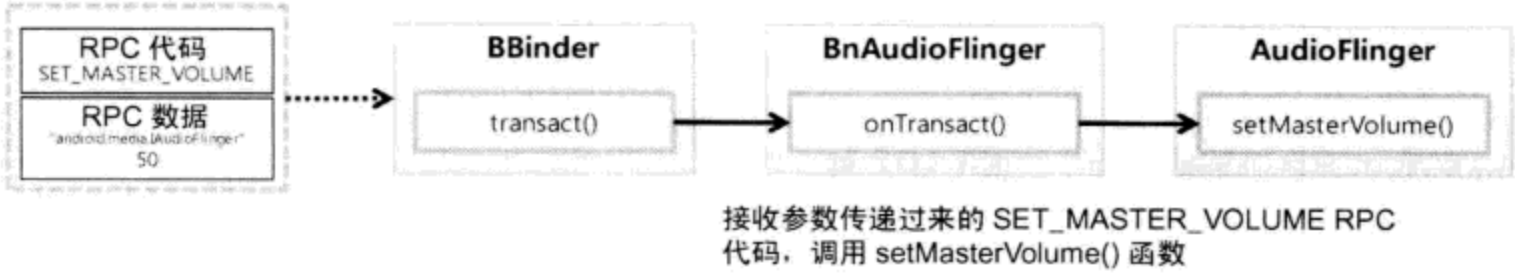


图 8-14 | BnAudioFlinger 的 onTransact()函数处理 RPC 代码

代码 8-10 是 BnAudioFlinger 的 onTransact()函数的代码。若函数的第一个参数是 SET_MASTER_VOLUME RPC 代码，onTransact()函数就会调用 setMasterVolume()函数，最后被调用的函数是 AudioFlinger 类的 setMasterVolume()函数，如图 8-13（1-2）所示。函数的第二个参数是 RPC 数据，该数据中包含 IAudioFlinger 的接口名称“android.media.IAudioFlinger”以及音量大小值。在 onTransact()函数中首先调用 CHECK_INTERFACE()宏函数[1]，检查被调用的服务接口的名称与 AudioFlinger 服务接口名称是否一致，而后将音量大小的值以参数的形式传递给 setMasterVolume()函数。

```
status_t BnAudioFlinger::onTransact(
    ➥ uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
   switch(code) {
   case SET_MASTER_VOLUME: {
      CHECK_INTERFACE(IAudioFlinger, data, reply);
      reply->writeInt32( setMasterVolume(data.readFloat()) );
      return NO_ERROR;
   } break;
   }
}
```

代码 8-10 | IAudioFlinger.cpp[2]-BnAudioFlinger 的 onTransact()函数

1 #defing CHECK_INTERFACE(interface, data, reply)\if(!data checkInterface(this)){return PERMISSION_DENIED;}

2 /frameworks/base/media/libmedia/IAudioFlinger.cpp

更多更新编程资源尽在 http://www.pin5i.com/

代码 8-10 是 BnAudioFlinger 的 onTransact()函数的代码，请仔细阅读代码。

代码 8-8❷：服务功能的实现

setMasterVolume()函数提供系统音量调节功能。由于对 setMasterVolume()函数源码的分析已经超出本书讨论的范围，感兴趣的读者，请参看 Android 系统源码[1]。

代码 8-8❸：注册 Audio Flinger 服务

一个系统服务必须先将自身注册到 Android 系统中才能提供给服务用户使用。一般地，服务注册将在服务生成的同时进行，调用 AudioFlinger 类的 instantiate()函数会生成 AudioFlinger 实例，并通过 Service Manager 将其注册到系统之中。代码 8-11 是 instantiate()函数的代码，它调用 Service Manager 的 addService()函数，将 AudioFlinger 服务对象注册到系统中。关于通过 Service Manager 向系统注册服务的过程，请参看 8.4 “Native Service Manager”一节中的内容。

```
void AudioFlinger::instantiate() {
   defaultServiceManager()->addService(
       ➥ String16("media.audio_flinger"), new AudioFlinger());
}
```

代码 8-11 | AudioFlinger.cpp-instantiate()函数

8.3.3 服务代理（Service Proxy）

服务代理函数用来传递 Binder RPC 所需要的 RPC 代码与数据，这些函数具体在服务代理类中实现。比如 BpAudioFlinger 服务代理的 setMasterVolume()函数会将 SET_MASTER_VOLUME（RPC 代码）与音量值（RPC 数据）发送到 BpBinder 中。下面分析一下 Audio Flinger 服务的 BpAudioFlinger 服务代理类。

BpAudioFlinger 服务代理类继承了 IAudioFlinger 类，服务用户通过它来使用 Audio Flinger 服务。代码 8-12 是 BpAudioFlinger 类的代码。

```
class BpAudioFlinger : public BpInterface<IAudioFlinger>          ❶
{
public:

   virtual status_t setMasterVolume(float value)                  ❷
   {
      Parcel data, reply;
```

1 /frameworks/base/libs/audioflinger/AudioFlinger.cpp

更多更新编程资源尽在 http://www.pin5i.com/


```
        data.writeInterfaceToken(IAudioFlinger::getInterfaceDescriptor());
        data.writeFloat(value);
        remote()->transact(SET_MASTER_VOLUME, data, &reply);
        return reply.readInt32();
    }

    BpAudioFlinger(const sp<IBinder>& impl)          ❸
        : BpInterface<IAudioFlinger>(impl){}
};
```

代码 8-12 | IAudioFlinger.cpp-BpAudioFlinger 类的主要代码

代码 8-12❶：BpAudioFlinger 的继承关系

由代码 8-12 可以看出，BpAudioFlinger 类继承了 BpInterface 类。BpInterface 类被声明为类模板（Class Template），它继承了 IAudioFlinger 接口。图 8-15 是 BpAudioFlinger 与 BpInterface 类的继承关系图，通过该图可以大致了解各个类之间的继承关系。

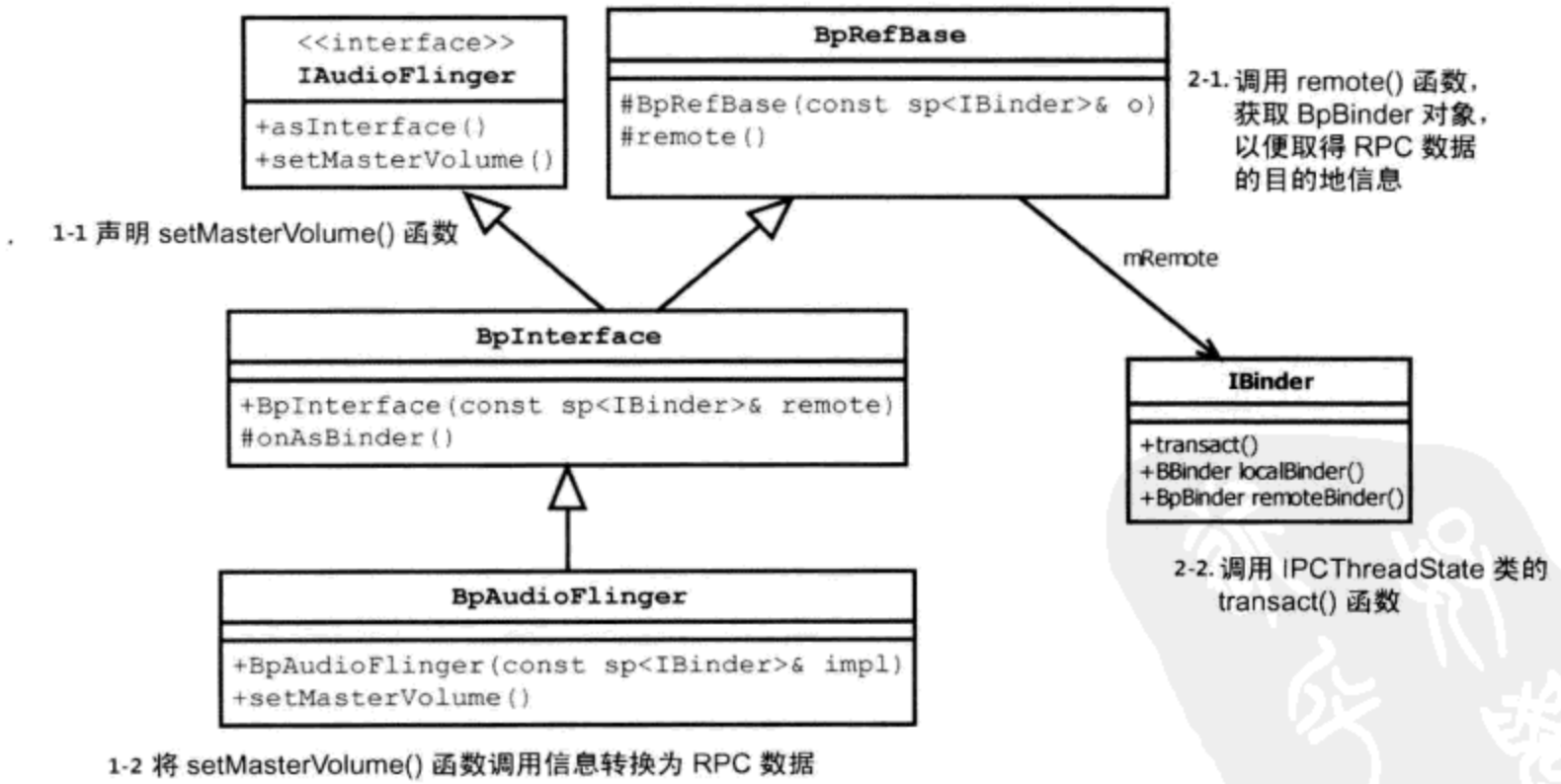


图 8-15 | BpAudioFlinger 类的继承关系图

从图 8-15 的类图，可以看出 setMasterVolume()函数在 IAudioFlinger 接口中被声明，如图 8-15（1-1）所示。如图 8-15（1-2）所示，BpAudioFlinger 类重新定义了 setMasterVolume()函数，并提供了将调用 setMasterVolume()函数相关的信息转换为 RPC 代码与数据的功能。

如图 8-15（2-1）所示，remote()函数在 RefBpBase 类中被声明，它返回 IBinder 类型

更多更新编程资源尽在 http://www.pin5i.com/

的对象。在图 8-15(2-2)中 IBinder 类处理 setMasterVolume()函数的调用，若为 BpBinder，则将 RPC 代码、RPC 数据，以及服务 Handle 信息一并传递给 IPCThreadState 的 transact()函数。

代码 8-12❷：setMasterVolume()函数

setMasterVolume()是 BpAudioFlinger 服务代理类的一个函数，它将自身的调用信息以 Binder RPC 代码、RPC 数据的形式返回。代码 8-13 是 setMasterVolume()函数的代码。setMasterVolume ()函数首先将 IAudioFlinger 接口名称与音量值写入 RPC 数据中，而后调用 IBinder 的 transact()函数，并将 SET_MASTER_VOLUME RPC 代码（作为 transact()函数的第一个参数）与 RPC 数据（作为 transact()函数的第二个参数）传入到函数中。

```
virtual status_t setMasterVolume(float value)
{
    Parcel data, reply;
    data.writeInterfaceToken(IAudioFlinger::getInterfaceDescriptor());
    data.writeFloat(value);
    remote()->transact(SET_MASTER_VOLUME, data, &reply);
    return reply.readInt32();
}
```

代码 8-13 | IAudioFlinger.cpp- setMasterVolume()函数

图 8-16 描述了调用 setMasterVolume()函数生成 RPC 代码、RPC 数据，而后将它们传递给 IBinder 的 transact()函数的过程。表示 setMasterVolume()函数被调用的 SET_MASTER_VOLUME 以 RPC 代码的形式被传递，而音量值与接口名称字符串（android.media.AudioFlinger）则以 RPC 数据的形式被传递。

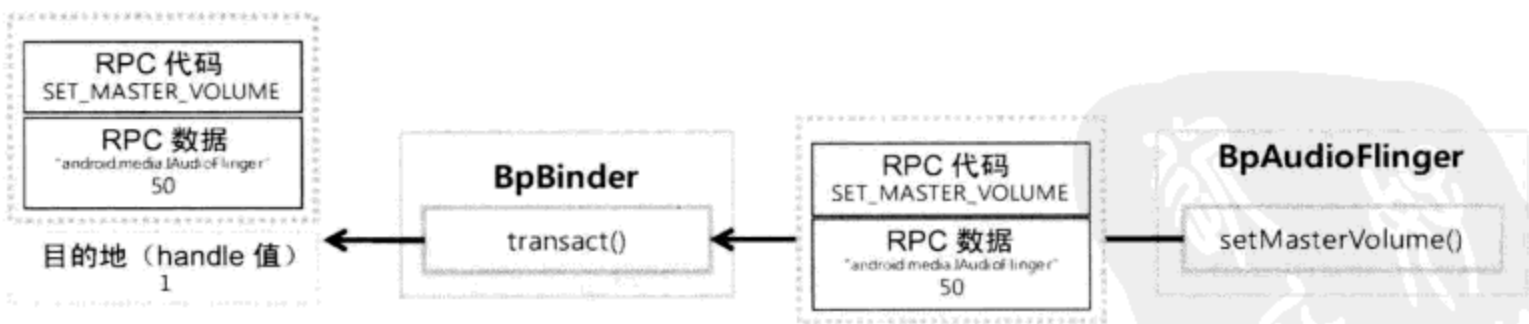


图 8-16 | 调用 setMasterVolume()函数时生成的 RPC 代码与数据

代码 8-12❸：获取 BpBinder 对象

BpBinder 对象是以 BpAudioFlinger 构造函数的参数形式获取的，下面来分析创建 BpAudioFlinger 的代码。代码 8-14 是 AudioSystem 类的 get_audio_flinger 成员函数的主要源代码。

更多更新编程资源尽在 http://www.pin5i.com/


```
const sp<IAudioFlinger>& AudioSystem::get_audio_flinger()
{
    sp<IServiceManager> sm = defaultServiceManager();
    sp<IBinder> binder;
    do {
        binder = sm->getService(String16("media.audio_flinger"));
        if (binder != 0)
            break;
        usleep(500000); // 0.5 s
    } while(true);

    gAudioFlinger = interface_cast<IAudioFlinger>(binder);
}
```

代码 8-14 | AudioSystem.cpp[1]-get_audio_flinger()函数的主要代码

首先，AudioSystem 通过 Service Manager 获取 Audio Flinger 服务的服务代理对象。而后调用 interface_cast()函数[2]，将 IBinder 类型转换为 IAudioFlinger 类型。

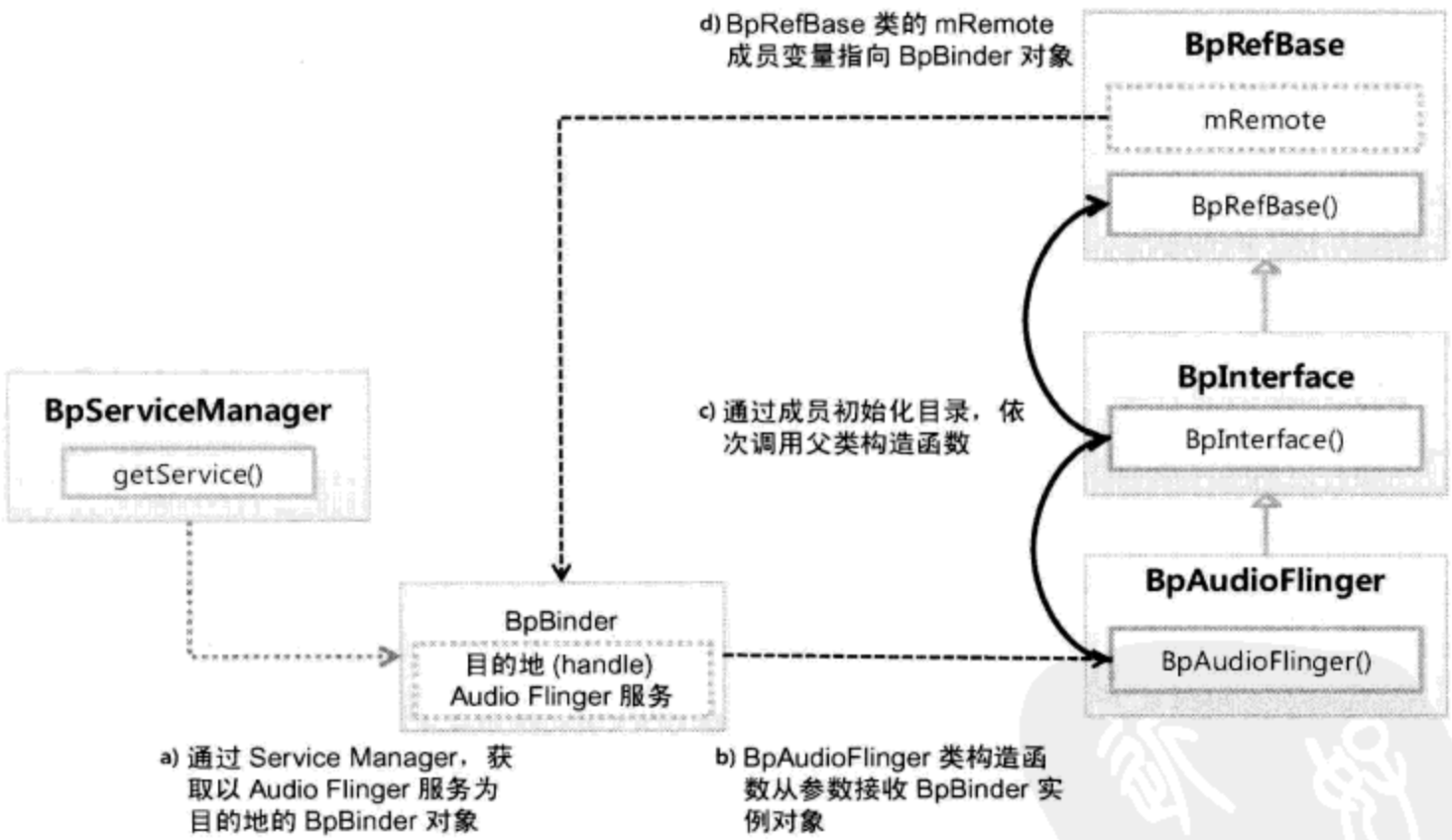


图 8-17 | 生成 BpAudioFlinger 类

[1] /frameworks/base/media/libmedia/AudioSystem.cpp

[2] interface_cast()函数是一个函数模板，在函数体内调用 IAudioFlinger 的 asInterface()函数。

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

更多更新编程资源尽在 http://www.pin5i.com/

图 8-17 描述了在 asInterface()函数[1]中 BpAudioFlinger 实例的生成过程。[2] a）调用 Service Manager 的 getService()函数，获取 BpBinder 对象；b）以参数的形式将 BpBinder 对象传递给 BpAudioFlinger 的构造函数。c）通过成员初始化列表（member initialization list），依次调用 BpAudioFlinger 父类 BpInterface、RefBpBase 的构造函数。d）最后，BpBinder 对象被保存到 RefBpBase 的 mRemote 变量中。

8.3.4 Binder IPC 处理

ProcessState

ProcessState 类提供的主要功能概括如下：

- ProcessState 类生成后，将打开 Binder Driver，以便在 Binder IPC 中使用。
- 创建 BpBinder 类的实例。也可以生成目的地为 Context Manager 的 BpBinder 实例。

关于 ProcessState 类的行为，将在 8.4“Native Service Manager”一节中以 Audio Flinger 服务为例进行详细说明。

IPCThreadState

在与 Binder Driver 进行交互，执行 Binder IPC 通信时，IPCThreadState 类负责传递 Binder IPC 数据。同样，关于 IPCThreadState 类的行为，将在 8.4“Native Service Manager”一节中以 Audio Flinger 服务为例进行详细说明。

Parcel

Parcel 是 Binder IPC 数据的容器，这些数据由发送端传送到接收端。Parcel 专为传递 IPC 设计，设计精良而高效。Parcel 中有多种读写各种类型数据的成员函数，其中最基本的函数用来读写基本数据类型的数据，支持的数据类型有 int、double、float、string。int 类型支持 32 位与 64 位，string 支持 C 风格的字符串、UTF-8 以及 UTF-16 字符串。此外，Parcel 还可以读写文件描述符（File Descriptor）与 IBinder 类对象的引用。

图 8-18 依次显示了在调用 BpAudioFlinger 类的 setMasterVolume()成员函数时保存在 Parcel 实例中的数据。图中 Parcel 的实例指 data 与 mOut 两个变量，data 变量作为函数参数传递，mOut 变量则相当于 IPCThreadState 的数据成员。

❶ 调用 BpAudioFlinger 的 setMasterVolume()函数，将 SET_MASTER_VOLUME（RPC 代码）作为第一个参数传入 BpBinder 的 transact()函数中，而后将“android.media.IAudioFlinger”字符串与音量值（50.0）保存到 Parcel 实例（data）中，

---

[1] 在 8.3.1“服务接口”一节已作出说明。

[2] 假定 Audio Flinger 服务用户与 AudioFlinger 服务位于不同的进程中。

更多更新编程资源尽在 http://www.pin5i.com/

再将其作为第二个参数传入函数中。

❷ 在 BpAudioFlinger 的 setMasterVolume()函数中调用 BpBinder 的 transact()函数时，目的地信息将被作为首个参数传入 IPCThreadState 的 transact()函数中，并且 Parcel 实例将被作为第二个参数传入函数中。

❸ 在 BpBinder 的 transact()函数中，调用 IPCThreadState 的 transact()函数时，BC_TRANSACTION（Binder 协议）与❷-❸中传递过来的所有数据都将被作为参数传入 writeTransactionData()函数中。该函数将创建 binder_transaction_data 结构体，以供 Binder Driver 使用，并且将接收到的 RPC 代码、RPC 数据，以及目的地信息保存到结构体之中。

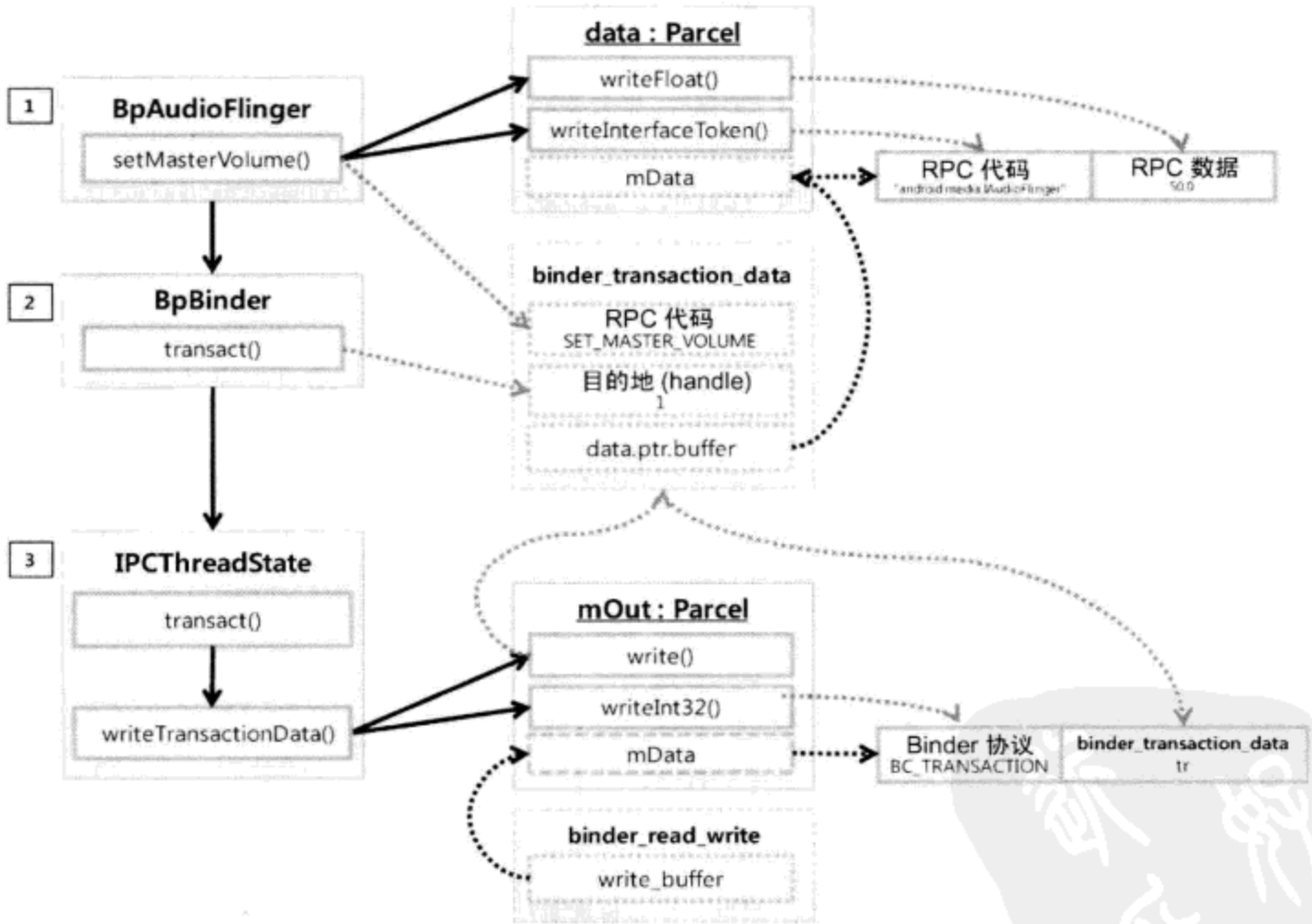


图 8-18 | 调用 setMasterVolume()函数，保存数据到 Parcel 对象

接着，生成 Binder IPC 数据，这些数据是要被传递给 Binder Driver 的。如在第 7 章“Android Binder IPC”中所讲解的一样，Binder IPC 数据由 Binder 协议与 binder_transaction_data 结构体构成。因此 writeTransactionData()函数将把接收的 BC_TRANSACTION（Binder 协议）以及 binder_transaction_data 结构体保存到 Parcel 实例（mOut）中。

那么，保存在 Parcel 实例（mOut）中的 Binder IPC 数据会怎样呢？在第 7 章“Android Binder IPC”中，已经讲解过 IPCThreadState 类与 Binder Driver 使用 binder_write_read 结构体来传递数据。在传送 Binder IPC 数据时，Binder IPC 数据将被保存到 binder_write_read 结

更多更新编程资源尽在 http://www.pin5i.com/

构体的 write_buffer 中。最终，Parcel 实例（mOut）将被保存到 write_buffer 中，并传递给 Binder Driver。

我们曾经提到过在使用 Parcel 类传递信息时，信息中不仅包含数据，还有对象的引用。以上只是讲解了发送数据的情形，关于传递对象引用的情形，将在 8.4“Native Service Manager”一节中详细讲解。

8.4 本地服务管理（Native Service Manager）

8.4.1 Service Manager 概要

在 Android 系统中存在着各种各样的服务，如管理音频设备的 Audio Flinger 服务、管理帧缓冲设备的 Surface Flinger 服务，还有管理相机设备的 Camera 服务等。这些服务的信息与目录都是由“上下文管理器”（Context Manager）负责管理的。

如图 8-19 所示，服务[1]向 Context Manager 提出服务注册请求，服务用户通过服务名称从 Context Manager 获取服务信息。在这一过程中，Context Manager 充当连接服务与服务用户的链条。

图 8-19 | Context Manager、服务及服务用户间的关系

服务用户从 Context Manager 获取服务信息后，即可在没有 Context Manager 的介入下直接访问服务。服务用户直接访问服务不行吗？为什么还需要 Context Manager 这个管理器呢？若想知道这个问题的答案，请继续阅读下面的内容。

[1] 这里指注册在 Context Manager 中的服务，如本地系统服务。注意并不是所有的服务都注册到 Context Manager 中。

更多更新编程资源尽在 http://www.pin5i.com/

我们知道，服务与服务用户是不同的进程，它们分别运行在各自独立的内存空间中，在两个进程中若想实现信息共享，进程间必须进行 IPC 通信。如第 7 章所述，Android 系统通过 Binder Driver 来提供并管理服务与服务用户进程间的通信。

在服务被注册到 Context Manager 的过程中，Binder Driver 将生成 Binder 节点，该节点中包含服务连接的信息。并且 Service Framework 会以服务 Handle 值的形式引用 Binder 节点。服务用户若想访问服务，必须首先从 Context Manager 中获取服务的 Handle 值，以便连接到要使用的服务上。

Service Framework 提供了用于处理 Binder RPC 的服务代理（Service Proxy）[1]，这些服务代理帮助服务用户使用 Audio Flinger、Surface Flinger、相机服务等各种服务提供的功能。服务用户通过服务代理使用相关服务，服务代理与服务间复杂的 Binder RPC 由底层的 Service Framework 完成，在上层服务用户看来，服务代理与服务就像同一个进程，如同在同一个进程中使用服务一样。事实上，我们完全可以把服务代理想象成服务。

Service Manager 相当于 Context Manager 的服务代理。下面来分析使用 Service Manager 注册服务，以及获取服务信息的过程。

首先分析服务（Service）与服务管理器（Service Manager）的关系。在 Android 启动过程中，System Server 初始化并启动多种服务，这些服务通过 Service Manager 与 Context Manager 进行 Binder RPC 调用，图 8-20 描述了这一过程。

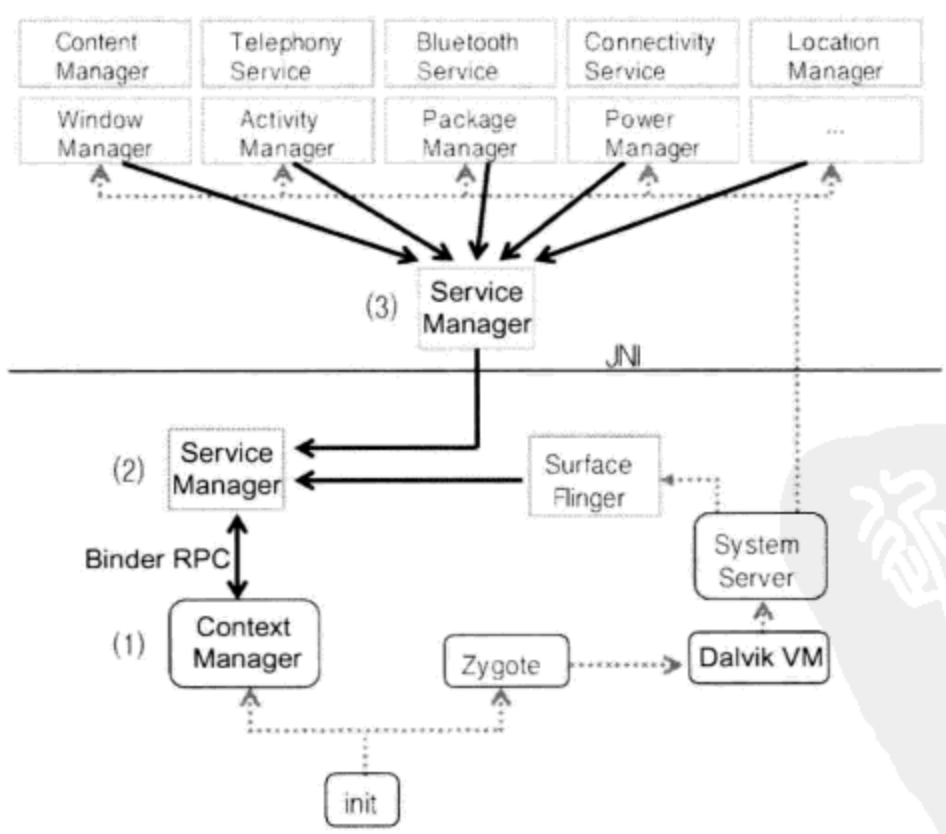


图 8-20 | System Server 中的服务与 Service Manager 的关系

[1] 服务代理负责在服务与服务用户间传递 Binder RPC 数据。Audio Flinger 的服务代理是 BpAudioFlinger。

更多更新编程资源尽在 http://www.pin5i.com/

Java 系统服务通过(3)的 Java 层的 Service Manager 注册服务，本地系统服务通过(2)的 C/C++层的 Service Manager 注册服务。(3)Java 层的 Service Manager 通过 JNI 与(2)的 C/C++层的 Service Manager 连接在一起，C/C++层的 Service Manager 通过 Binder Driver 与(1)的 Context Manager 进行 Binder RPC。关于 Context Manager 的内容在第 7 章已经讲解过，本部分将讲解 C/C++层的 Service Manager，另外关于 Java 层的 Service Manager，我们将在第 10 章“Android Java Service Framework”中进行讲解。

8.4.2 Service Manager 类

前面曾经讲到过，Context Manager 的服务代理即是 Service Manager，并且 Service Manager 分为 C/C++层与 Java 层两种。本节将讲解位于 C/C++层的 Service Manager 由哪些类构成，以及它有什么功能，都做了哪些事情。

如图 8-21 所示，左下侧的 BpServiceManager 继承了两个类。首先它继承了 Iservice Manager 服务接口，实现了创建 Binder RPC 数据的方法，这些 Binder RPC 数据用于向 Context Manager 提出服务注册或获取服务信息请求。其次，BpServiceManager 还继承了 BpInterface，以便通过指向 Context Manager 的 BpBinder，向 Context Manager 传递 Binder RPC 数据。

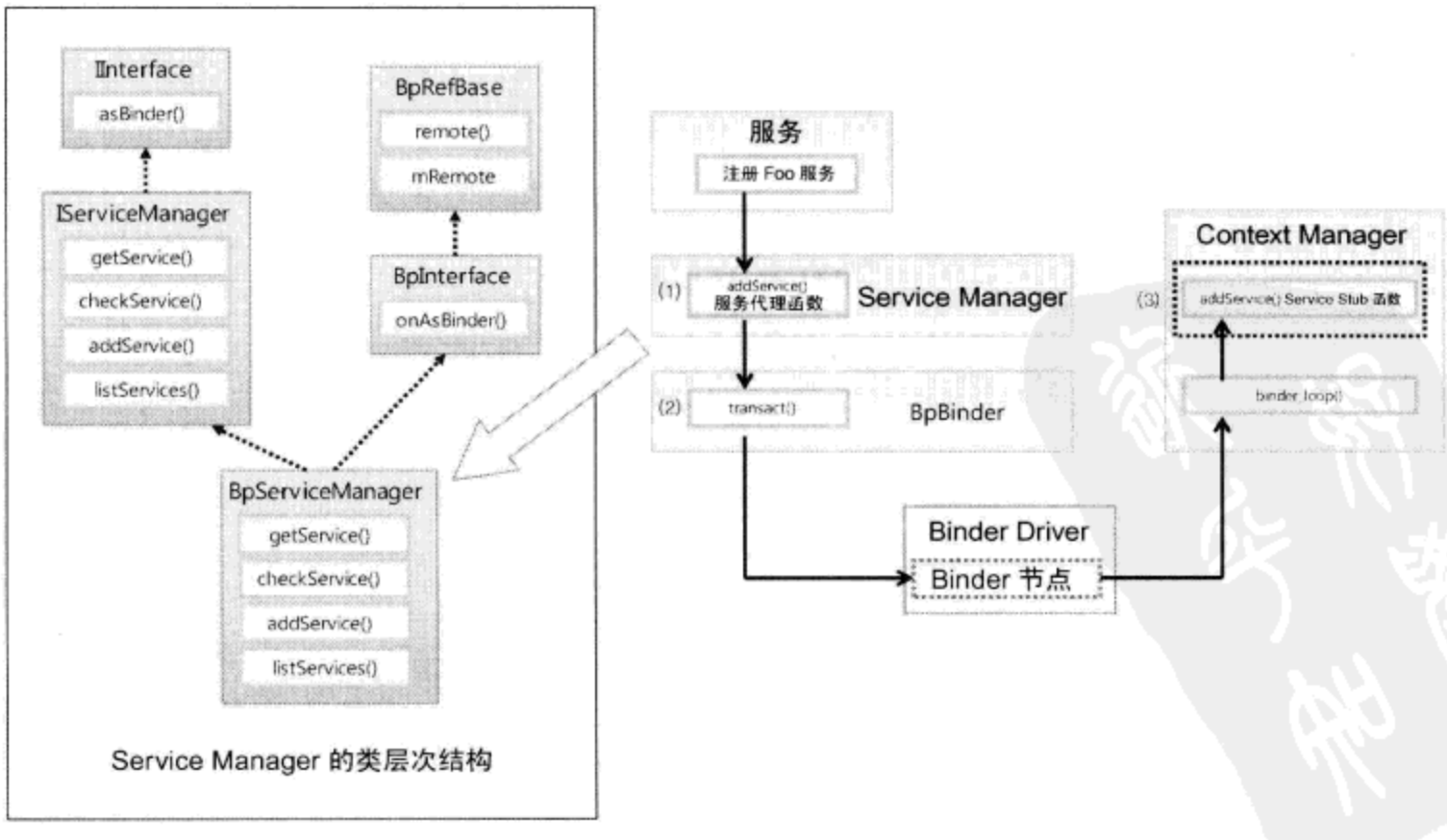


图 8-21 | Service Manager 的类层次结构以及 Binder RPC 的过程

以服务注册为例，分析一下它们之间是如何相互作用的。如图 8-21 右图所示，Service Manager 向 Context Manager 提出服务注册请求，在(1)中生成用于注册服务的 Binder

更多更新编程资源尽在 http://www.pidi.com/

RPC 数据，而后引用(2)中 BpBinder 的服务 Handle，最后通过(3)的 Binder Driver 向 Context Manager 传递 Binder RPC 数据。

如图 8-22(左)所示，在 Android 平台的普通服务运作过程中，BpFoo 是服务代理，BnFoo 是服务 stub，它们分别继承并实现了服务接口 IFoo 类，IFoo 类是抽象类，其成员函数是纯虚函数。在 Android 基本的 RPC 机制中，服务 stub 将处理来自服务代理的 Binder RPC 请求。

当服务用户请求 BpFoo.foo()函数调用时，BnFoo.foo()即对该请求进行处理，这种方式是 Android 使用的基本 RPC 机制。但是，当服务接口为 Service Manager 时，服务调用方式略有不同，如图 8-22（右）所示。如图所示，BpServiceManager 继承并实现了 IServiceManager 服务接口，它相当于服务代理，但服务 stub 不是由 BnService Manager 来处理，而是由一个单独的进程 Context Manager（servicemanager [1]）进行处理，即在 Service Manager 的 Binder RPC 中不会涉及到 BnServiceManager 类。

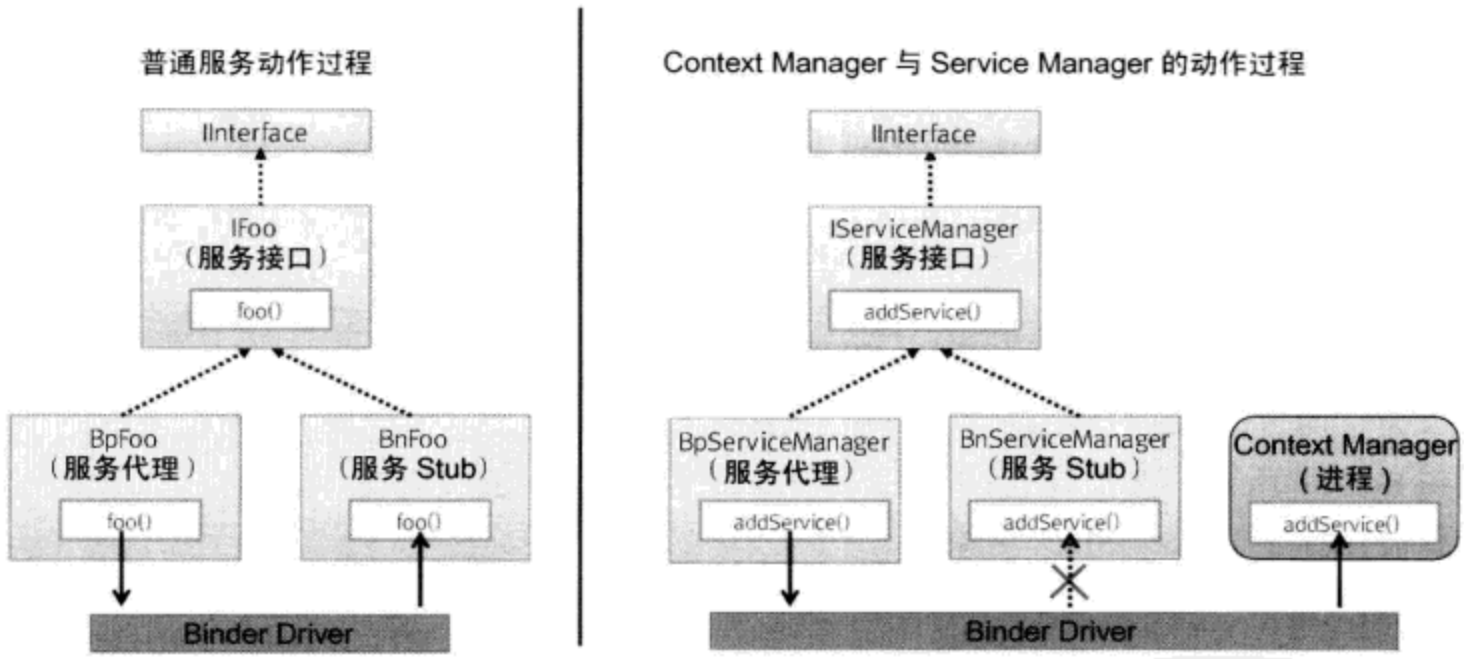


图 8-22 | Android 基本的 Binder RPC 与 Service Manager 的 Binder RPC

那么，作为一项服务，Context Manager 提供哪些功能呢？首先，Service Manager（服务代理）与 Context Manager（服务）通过基于 IServiceManager 服务接口的 Binder RPC 进行交互操作。因此我们可以通过分析 IServiceManager 来了解 Context Manager 提供的功能。IServiceManager 是如何定义 Context Manager 提供的各种功能的呢？请参看表 8-3，表中列出了 IServiceManager 中主要的成员函数。

表 8-3 IServiceManager 的成员函数

sp<IBinder> getService(const String 16&name)	此函数以服务名称（name）作为参数，接收到服务名称后，调用 checkService()成员函数。当 Binder IPC 失败时，重新尝试。该函数的返回值是一个指向 BpBinder 实例的指针。

[1] Context Manager 的进程名称为 servicemanager，与 Service Manager 完全不同，希望读者不要将两者混淆了。

更多更新编程资源尽在 http://www.pin5i.com/

续表

sp<IBinder> checkService(const String16&name)	接收服务名称，并从 Context Manager 获取相关服务的信息，而后生成 BpBinder 实例，返回指向该实例的指针。
status_t addService(const Sting16& name,const sp<IBinder>&service)	以服务名称与服务实例的指针为参数，向 Context Manager 请求注册服务，其返回值表示服务是否注册成功。
Vector<String16> listService	该函数是无参函数，获取注册在 Context Manager 中的服务的列表，并将其返回。

如同上表所列的一样，Context Manager 提供以上四种方法的功能，BpService Manager 中有相应的服务代理函数来实现这四种功能，每个函数的功能通过函数的名称即可知道，例如 addService()函数用来注册服务，getService()函数用来获取服务信息等。

8.4.3 Service Manager 的运行

前面已经了解了 Service Manager 的大概。在本节中，将通过一些示意图与源代码的分析来具体地了解 Service Manager 主要的行为动作。

Service Manager 初始化

服务或服务用户在使用 Service Manager 注册或检索服务之前，需要先创建 Process State 实例，该实例中拥有 Binder Driver 的信息。

为了通过 Binder IPC 使用 Context Manager 的功能，首先生成 ProcessState 实例，如图 8-23(1)所示。在生成 ProcessState 实例的同时，Binder Driver 即被打开，并且相关信息被保存在 ProcessState 对象中。

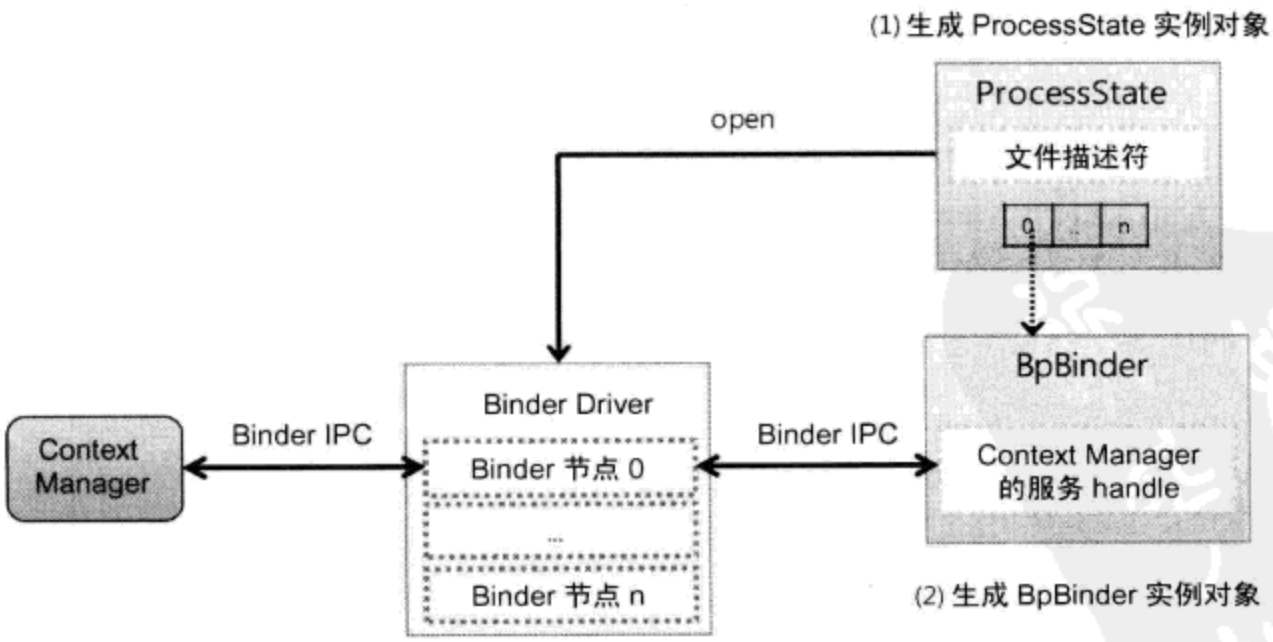


图 8-23 | Service Manager 初始化-生成 ProcessState 与 BpBinder 实例

然后，如图 8-23(2)所示，创建 BpBinder 对象，该对象拥有 Context Manager 的服

务 Handle 值。

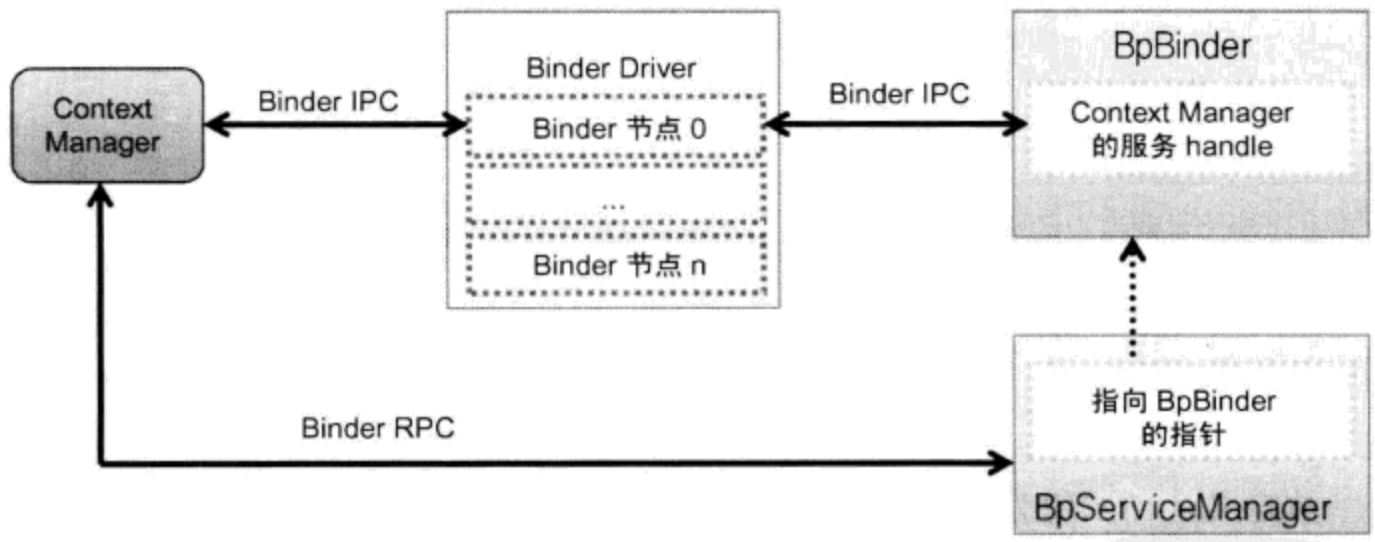
由于 Context Manager 的服务 Handle 值为 0，所以直接生成 BpBinder。对于其他服务，由于不知道这些服务的 Handle 值，需要先从 Context Manager 获取服务信息后，再生成相应的服务。关于这一过程，将在获取服务信息的内容中进行详细说明。

在通过服务 Handle 向特定服务发送 Binder IPC 数据时，将会用到 BpBinder，并且通过 ProcessState 可以防止在同一进程中重复生成 BpBinder 对象。因此，当应用程序访问某个使用中的服务时，不必创建新的 BpBinder 对象，直接使用已有的 BpBinder 对象即可。

我们知道 BpBinder 拥有指向 Context Manager 的 Binder 节点的服务 Handle，在创建 BpBinder 实例的过程中，它们之间通过 Context Manager 与 BpBinder 间的 Binder IPC 进行相互作用，如图 8-23 所示。

接下来，生成 BpServiceManager 实例，它用来与 Context Manager 进行 Binder RPC 操作。

如图 8-24(3)所示，BpServiceManager 拥有指向 BpBinder 的指针，而 BpBinder 则拥有 Context Manager 的服务 Handle。



(3) 生成 BpServiceManager 实例对象

图 8-24 | Service Manager 初始化-生成 BpServiceManager 实例

至此，BpServiceManager 与 Context Manager 之间就可以进行 Binder RPC 操作了。

服务 Handle 与 BpBinder，以及与 BpServiceManager 的连接就像拼插积木一样，在连接其他服务与服务代理时也采用这样的结构形式。从图 8-21 的类结构示意图可以看出，BpServiceManager 继承了 BpInterface，所有服务代理也继承了 BpInterface。

Service Manager 初始化分析

以 Media Server 的 main()函数代码（代码 8-15）为例，进一步分析 Service Manager

 更多更新编程资源尽在 http://www.pin5i.com/

的初始化过程。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());          ❶
    sp<IServiceManager> sm = defaultServiceManager();     ❷
    ...
}
```

代码 8-15 | Media Server 的 main()函数[1]

❶ 由于 ProcessState 的构造函数是私有函数（关键字 private），在创建 ProcessState 实例时，无法使用 new 运算符，只能调用它的 self()成员函数来创建实例对象。

❷ 创建 BpServiceManager 实例对象，相关内容将在“（3）生成 Service Manager（BpServiceManager）”中详细讲解。

（1）生成 ProcessState

```
sp<ProcessState> ProcessState::self()
{
    if (gProcess != NULL) return gProcess;     ❸
    if (gProcess == NULL) gProcess = new ProcessState();     ❹
        return gProcess;
}
```

代码 8-16 | ProcessState 的 self()函数[2]

如代码 8-16 所示，self()成员函数在创建 ProcessState 的实例时采用了“单例模式”（Singleton pattern）。

❸ gProcess 是一个全局变量，用来保存 ProcessState 对象，该变量在 libbinder 中被定义，初始值为 NULL。

❹ 使用 New 关键字，创建 ProcessState 类的实例，而后将生成的实例保存到全局变量 gProcess 中。

TIP 单例模式（Singleton pattern）

单例模式保证一个类只有一个实例，并提供一个访问它的全局访问点。该模式可以严格地控制客户怎样访问它以及何时访问它。

[1] frameworks/base/media/mediaserver/main_mediaserver.cpp

[2] frameworks/base/libs/binder/ProcessState.cpp

更多更新编程资源尽在 http://www.pan5i.com/

在 self()成员函数中，通过调用 ProcessState 的私有构造函数，创建 ProcessState 的实例。ProcessState 的私有构造函数主要代码如下：

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())  ❺
    , mVMStart(MAP_FAILED)
{
    mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE |
        ➥ MAP_NORESERVE, mDriverFD, 0);  ❻
}
```

代码 8-17 | ProcessState 构造函数

在 ProcessState 的构造函数中，主要做两件与 Binder 相关的事情，这一过程相当于进程使用 Binder IPC 的准备过程。

❺ 调用 ProcessState 的构造函数时，先调用 open_driver()函数打开 Binder Driver，并将返回的文件描述符保存到 mDriverFD 成员变量中。

❻ 从 Binder Driver 获取一块 Binder mmp 区域，当 Binder Driver 向进程传递 Binder RPC 数据时，该块区域被用作保存区域。并且，该块区域的首址被保存到 mVMStart 成员变量中。

这样，ProcessState 的实例就创建好了，整个创建过程可以用一个简单的示意图表现出来，如图 8-25 所示。

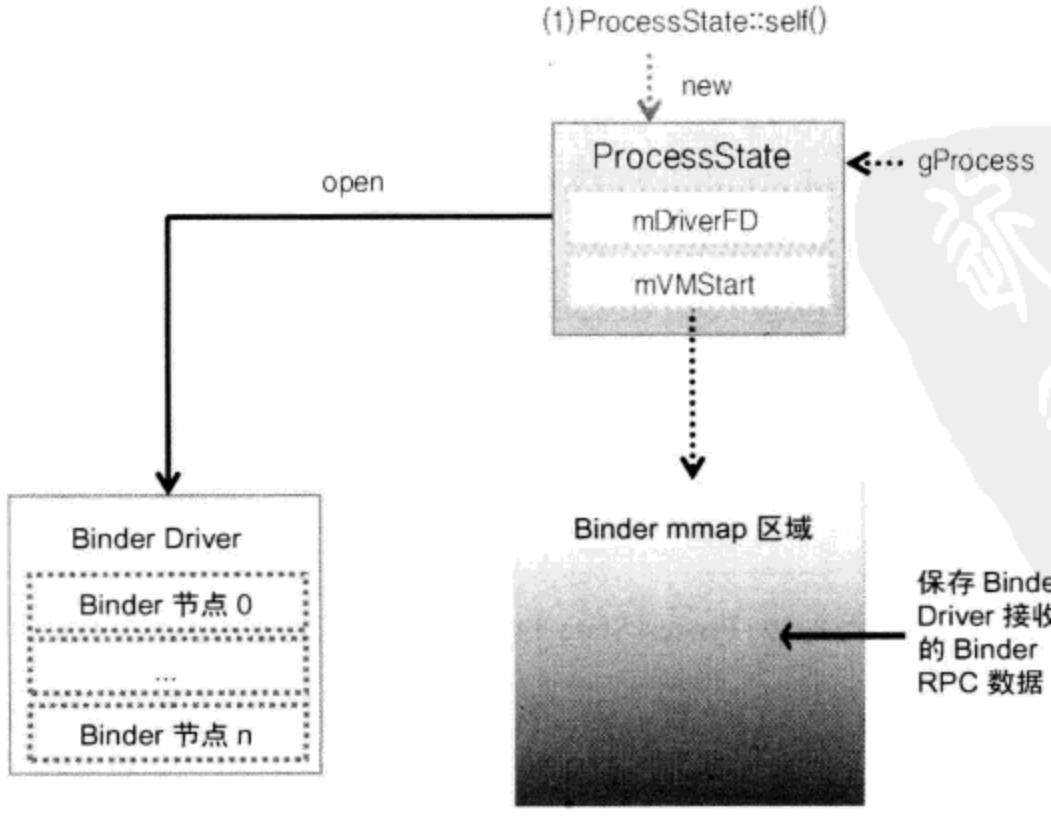


图 8-25 | 创建 ProcessState 的实例

更多更新编程资源尽在 http://www.pin5i.com/

（2）创建 BpBinder

在代码 8-15 中，第❷行代码调用 defaultServiceManager()函数，创建 BpService Manager 实例对象。下面来分析 defaultServiceManager()函数的代码。与创建 Process State 对象一样，在创建 BpServiceManager 实例对象时也使用了“单例模式”（Singleton pattern），如代码 8-18 所示。

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL)          ←❼
        return gDefaultServiceManager;

    if (gDefaultServiceManager == NULL) {
        gDefaultServiceManager = interface_cast<IServiceManager>(
            ProcessState::self()->getContextObject(NULL));          ←❽
    }
    return gDefaultServiceManager;
}
```

代码 8-18 | defaultServiceManager()函数[1]

❼ gDefaultServiceManager 是一个全局变量，用来保存 Service Manager，它在 libbinder 定义，初始值为 NULL。

❽ 该行语句较为复杂，首先调用 ProcessState 的 getContextObject()函数，再使用 interface_cast 将 getContextObject()函数的返回的对象转换成 IServiceManager 类型，最后将生成的 BpServiceManager 对象保存到全局变量 gDefaultService Manager 中。

首先来分析 ProcessState 的 getContextObject()函数的主要代码，代码如下所示。

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    if (supportsProcesses()) {          ←❾
        return getStrongProxyForHandle(0);          ←❿
    }
}
```

代码 8-19 | ProcessState 的 getContextObject()函数

❾ 调用 supportsProcesses()函数，若 ProcessState 的成员变量 mDriverFD 值为 0，

[1] frameworks/base/libs/binder/IServiceManager.cpp

更多更新编程资源尽在 http://www.pin5i.com/

则返回 true。当 ProcessState 的构造函数打开 Binder Driver 时将返回文件描述符，此文件描述符被保存到成员变量 mDriverFD 中。若 mDriverFD 值大于 0，则表示操作正常。

⑩ 调用 getStrongProxyForHandle()函数，该函数的参数为服务 Handle 值。若参数为 0，则说明它是 Context Manager 的服务 Handle。调用该函数将生成持有 Context Manager 的服务 Handle 的 BpBinder 对象，并返回指向该对象的指针。

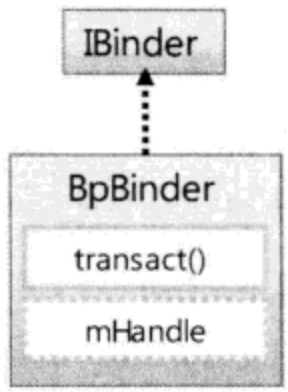


图 8-26 | BpBinder 的成员变量 mHandle

如图 8-26 所示，BpBinder 拥有一个名称为 mHandle 的成员变量，该变量保存着服务的 Handle。当调用 getStrongProxyForHandle(0)时，即可生成 mHandle 值为 0 的 BpBinder 对象。

请看图 8-27，该图描述了 8-18 的语句❽执行的情况。从示意图可以看出，代码执行后，最终生成了一个指向 Context Manager 的 BpBinder 对象。

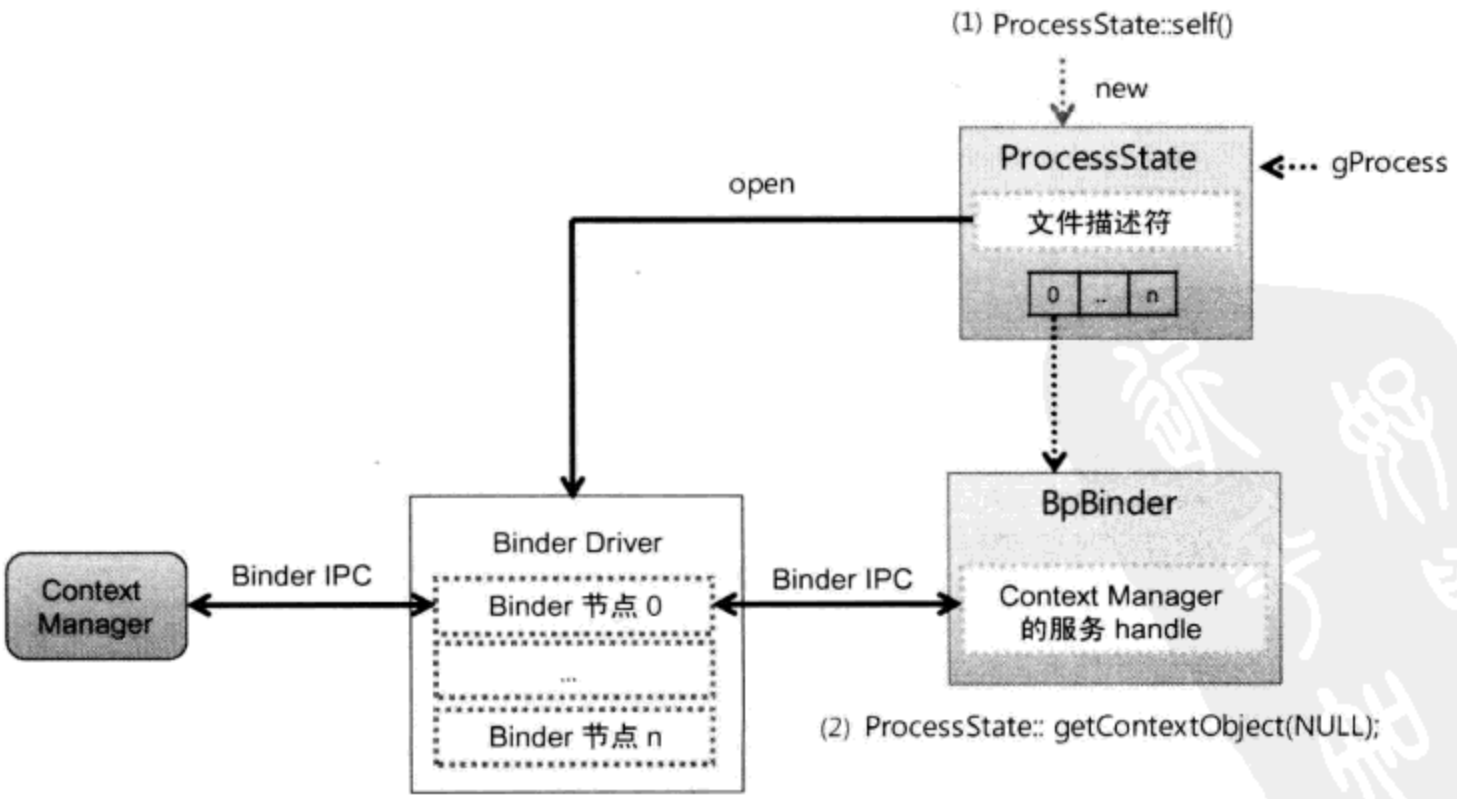


图 8-27 | 生成持有 Context Manager 的 Service Handle 的 BpBinder 对象

更多更新编程资源尽在 http://www.pin5i.com/

(3) 生成 Service Manager（BpServiceManager）

现在，再次回到代码 8-18□中，一起分析一下 interface_cast<Iservice Manager>部分。若想弄清 interface_cast<IServiceManager>中实际的处理函数，必须先了解 inter face_cast 模板函数（代码 8-20）与宏（代码 8-21）。

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

代码 8-20 | interface_cast 模板[1]

调用 interface_cast<IServiceManager>时，根据代码 8-20 中的模板代码，可以知道 INTERFACE 将被 IServiceManager 代替，最终调用 IServiceManager 的 asInterface()成员函数。

实际上，函数 asInterface()的代码是不存在的。请看下面一段宏定义，DECLARE_META_INTERFACE 是函数的声明部分，IMPLEMENT_META_INTERFACE 是函数的实现部分。

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

代码 8-21 | Service Manager 的接口宏[2]

当代码 8-21 中的 IMPLEMENT_META_INTERFACE 宏扩展后，形成如下所示的 asInter face()的代码。

```
sp<IServiceManager> IServiceManager::asInterface(const sp<IBinder>& obj)
{
   sp<IServiceManager> intr;
   if (obj != NULL) {
      intr = static_cast<IServiceManager*>(
       ➥ obj->queryLocalInterface(IServiceManager::descriptor).get());  ← ⓫
      if (intr == NULL) {
         intr = new BpServiceManager(obj);  ← ⓬
      }
   }
```

[1] frameworks/base/include/binder/IInterface.h

[2] frameworks/base/include/binder/IServiceManager.h
frameworks/base/libs/binder/IServiceManager.cpp

更多更新编程资源尽在 http://www.pin5i.com/


```
        return intr;
    }
```

代码 8-22 | 扩展 IMPLEMENT_META_INTERFACE 宏

⓫ IBinder 的 queryLocalInterface()函数将根据参数是 BBinder 或 BpBinder（它们都继承并实现了 IBinder）而采取不同的行为动作。当参数 obj 是 BBinder[1]时，转换类型为服务对象；当参数 obj 是 BpBinder 时，则返回 NULL。若传入与 Context Manager 相对应的 BpBinder，则 intr 的值为 NULL。

⓬ 生成新的 BpServiceManager 实例对象。通过构造函数的参数传入 BpBinder 实例对象的指针，该实例持有 Context Manager 的服务 Handle，并且 BpBinder 实例的指针被保存到 mRemote 成员变量中，mRemote 是 BpServiceManager 的父类 BpRefBase 类的成员变量，如图 8-28 所示。

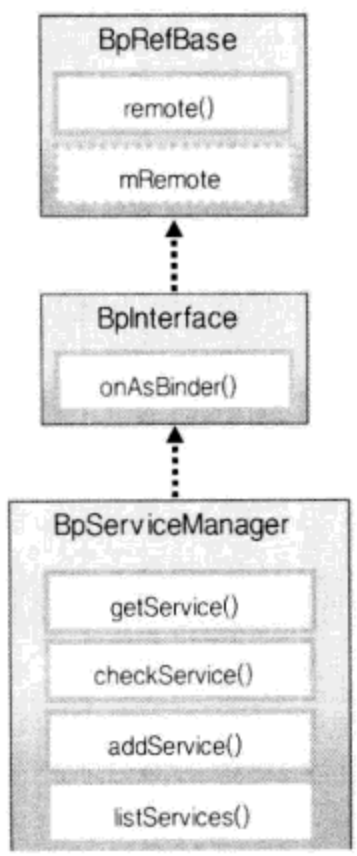


图 8-28 | BpRefBase 的 mRemote 成员变量与 remote()成员函数

如前所述，BpBinder 实例的指针被保存在 mRemote 变量中，它可以引用 BpRefBase 的 remote()成员函数。由于 BpServiceManager 继承了 BpRefBase 类，所以可以使用 remote()这个成员函数。

到此为止，已经学习了 Service Manager（BpServiceManager）的生成，以及通过 Binder Driver 连接 Context Manager 的过程，整个过程如图 8-29 所示。

[1] 采用 Binder IPC 的服务都继承了 BBinder 类。

更多更新编程资源尽在 http://www.pin5i.com/

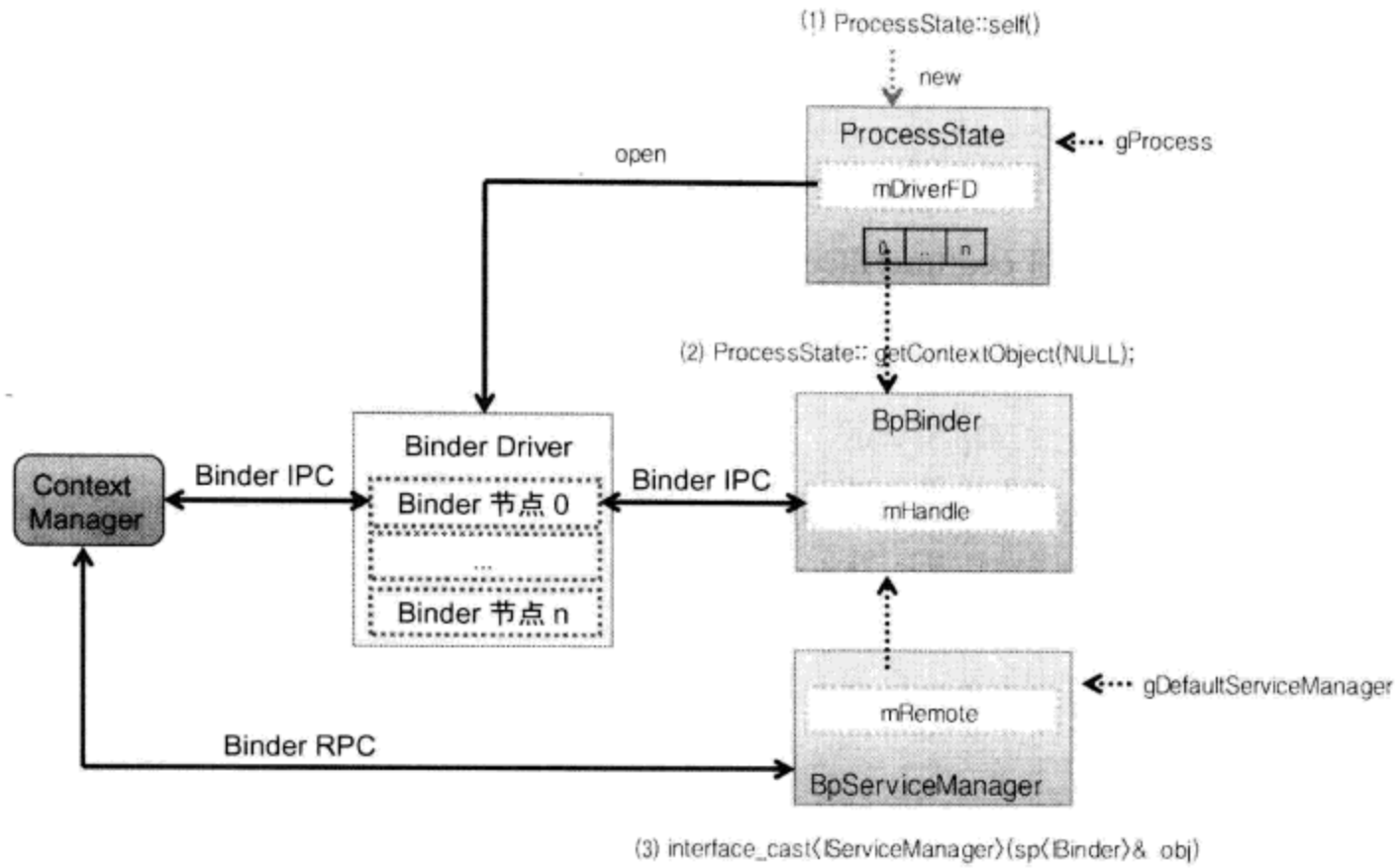


图 8-29 | 生成 BpServiceManager 的实例

上面通过分析代码学习了 Service Manager 的初始化过程，即生成 Service Manager 实例的过程。接下来，将学习如何使用 Context Manager 注册服务或获取服务信息的相关内容。

服务注册

为了让其他进程使用自身提供的功能，服务必须先注册到 Context Manager 之中。本节通过分析 Audio Flinger 的 instantiate()函数代码来学习服务注册的过程。

```
void AudioFlinger::instantiate() {
    defaultServiceManager()->addService(
        ➥ String16("media.audio_flinger"), new AudioFlinger());      ❶
}
```

代码 8-23 | Audio Flinger 的 instantiate()函数[1]

❶ 调用 defaultServiceManager()函数，获取 BpServiceManager 的实例指针。而后调用 BpServiceManager 的 addService()成员函数，并将 Audio Flinger 的服务名称以及新生成的 Audio Flinger 的实例指针传入函数中。

如图 8-30 所示，调用 BpServiceManager 的 addService()函数时，首先生成 Binder RPC

[1] frameworks/base/libs/audioflinger/AudioFlinger.cpp

更多更新编程资源尽在 http://www.pin5i.com/

数据，Binder RPC 数据用于发送到 Context Manager 以注册 Audio Flinger 服务。而后引用 BpBinder（该对象持有 Context Manager 的服务 Handle），通过 Binder Driver 传递给 Context Manager。

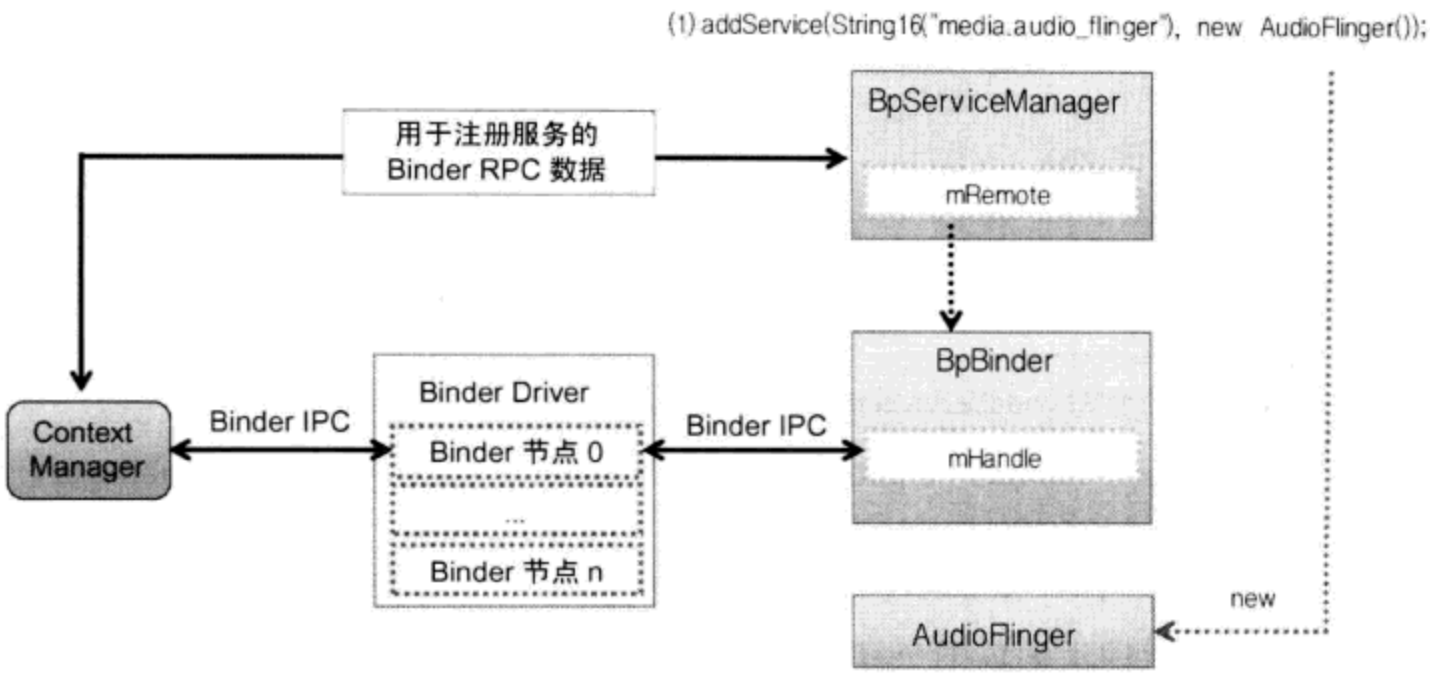


图 8-30 | 注册服务

下面来分析 BpServiceManager 的 addService()成员函数，函数源代码如下。

```
status_t addService(const String16& name, const sp<IBinder>& service)
{
    Parcel data, reply;                                    ←❷
    data.writeInterfaceToken(
        ➥ IServiceManager::getInterfaceDescriptor());     ←(a)
    data.writeString16(name);                              ←(b)
    data.writeStrongBinder(service);                       ←(c)
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION,
        ➥ data, &reply);                                  ←❸
    return err == NO_ERROR ? reply.readInt32() : err;      ←❹
}
```

代码 8-24 | BpServiceManager 的 addService()函数

❷ Parcel 类是信息的容器，该语句声明了两个 Parcel 类型的变量，以便存放数据。Parcel 类提供了多个函数，用来按照顺序保存数据，或者依次读取保存的数据。

addService()函数按照(a)、(b)、(c)顺序保存并生成要发送的 RPC 数据，保存的数据如图 8-31 所示。

❸ 该语句先调用 remote()函数，返回持有 Context Manager 的 BpBinder 对象，再传递生成的 Binder RPC 数据与 Binder RPC 代码（ADD_SERVICE_TRANSAC

更多更新编程资源尽在 http://www.pin5i.com/

TION），执行 Binder IPC 处理。

❹ 从 Context Manager 发送的 Binder RPC 应答数据中，读取大小为 int32 的数据，并将其返回。

关于代码 8-24❸❹的具体执行过程，将在后面分别进行详细讲解，此处从略。

变量 data（送信 $_{parcel}$）[1]是 Parcel 类型的变量，用于保存相关数据，存储的数据内容如图 8-31(a)(b)(c)所示。

26	"android.os.IServiceManager"	19	"media.audio_flinger"	flat_binder_object
(a) ServiceManager 的名称		(b) AudioFlinger 服务的名称		(c) AudioFlinger 对象序列化

图 8-31 | addService()的 data（送信 $_{parcel}$）变量中保存的内容

(a) 调用 IServiceManager 的 getInterfaceDescriptor()函数返回 Service Manager 的名称（“android.os.IServiceManager”），而后调用 Parcel 的 writeInterfaceToken()函数，将 Service Manager 名称保存到 data 中。并且保存时，首先保存字符串的长度，再保存字符串本身，接收 Binder RPC 数据的一方将从 data 中以指定的长度取出字符串并进行处理。

(b) 调用 Parcel 的 writeString16()函数，将 Audio Flinger 的服务名称（“media.audio_flinger”）保存到 data 中，存储格式如上图所示。

(c) 调用 Parcel 的 writeStrongBinder()函数，向 data 中保存表示 Audio Flinger 对象的 flat_binder_object 数据结构。

接着，分析图 8-31 的(c)部分。在将 Audio Flinger 的实例发送到 Binder Driver 时，需要先序列化 Audio Flinger 对象。因此，调用 Parcel 的 writeStrong Binder()函数，将 Audio Flinger 对象转换成 flat_binder_object 数据结构，执行对象序列化操作。

Parcel 的 writeStrongBinder()函数接收 Audio Flinger 对象参数，其代码如下：

```
status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
{
   return flatten_binder(ProcessState::self(), val, this);      ← ❺
}
```

代码 8-25 | Parcel 的 writeStrongBinder()函数[2]

❺ 调用 flatten_binder()函数，该函数是全局函数，带有三个参数，分别为 Process

[1] 下标 Parcel 表示 data 变量所属的数据类型，“送信”表示 data 是用来发送数据的。

[2] frameworks/base/libs/binder/Parcel.cpp

更多更新编程资源尽在 http://www.pin5i.com/

State 对象、AudioFlinger 对象，以及保存 Binder RPC 数据的 data（送信 parcel）对象的指针。

代码 8-26 是全局函数 flatten_binder()的主要代码。

```
status_t flatten_binder(const sp<ProcessState>& proc,
const sp<IBinder>& binder, Parcel* out)
{
   flat_binder_object obj;
   obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
   if (binder != NULL) {
      IBinder *local = binder->localBinder();  ←❻
      if (!local) {
         ...
      } else {    ←❼
         obj.type = BINDER_TYPE_BINDER;    ←(a)
         obj.binder = local->getWeakRefs();
         obj.cookie = local;    ←(b)
      }
   }
   return finish_flatten_binder(binder, obj, out);    ←❽
}
```

代码 8-26 | flatten_binder()函数

在 flatten_binder()函数中使用了 flat_binder_object 数据结构，其定义如下：

```
struct flat_binder_object {
   unsigned long type;
   unsigned long flags;
   union {
      void *binder;
      signed long handle;
   };
   void *cookie;
};
```

代码 8-27 | flat_binder_object 数据结构[1]

代码 8-26 的语句❻用来判断参数传递过来的 binder（当前为 AudioFlinger 实例）是否是 BBinder。由于 Audio Flinger 继承了 BBinder，所以它是 BBinder。代码继续执行❼的(a)(b)部分，将相关数据保存到 flat_binder_object 数据结构中，如图 8-32 所示。

图 8-32 的 flat_binder_object 数据结构中保存的数据如下：

[1] bionic/libc/kernel/common/linux/binder.h

更多更新编程资源尽在 http://www.pin5i.com/

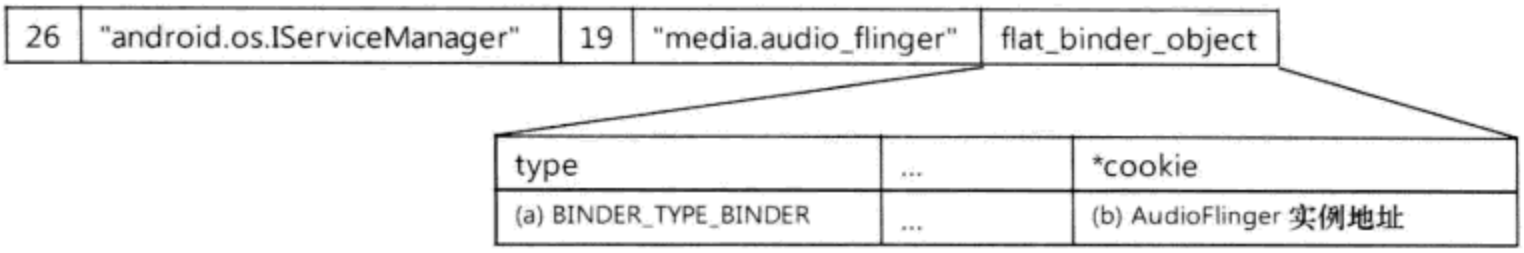


图 8-32 | flat_binder_object 数据结构

(a) 保存 BINDER_TYPE_BINDER 到 type 中，以便 Binder Driver 生成新的 Binder 节点。

(b) 在 cookie 中保存 Audio Flinger 实例对象的指针。

在代码 8-26 的语句❽中，调用 finish_flatten_binder()函数，将 flat_binder_object 数据结构保存到 data（送信 parcel）[1]中。

到此，调用 addService()函数生成 Binder RPC 数据（用于注册服务）的过程就讲解完了。返回到代码 8-24 中，仔细分析一下语句❸。在该行中，BpServiceManager 的 addService()函数将调用 BpBinder 的 transact()函数，同时把 Binder RPC 代码与 Binder RPC 数据作为参数传入函数中，以供函数生成 Binder IPC 数据。

transact()是 BpBinder 类中的一个函数，其主要代码如下所示。

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    status_t status = IPCThreadState::self()->transact(
    mHandle, code, data, reply, flags);      ❾
    return status;
}
```

代码 8-28 | BpBinder 的 transact()函数[2]

❾ 在该行中，从 BpServiceManager 的 addService()传递而来的参数（Binder RPC 代码、Binder RPC 数据、保存 Binder RPC 数据的 Parcel 类的指针）被原封不动地传递给 IPCThreadState 的 transact()函数。

除此之外，transact()函数还额外增添了一个参数，即第一个参数，它是自身所持有的服务 Handle（mHandle）。在本示例中，使用持有 Context Manager 服务 Handle 值的 BpBinder，所以 mHandle 的值为 0。

[1] 下标 Parcel 表示 data 变量所属的数据类型，“送信”表示 data 是用来发送数据的。

[2] frameworks/base/libs/binder/BpBinder.cpp

更多更新编程资源尽在 http://www.pin5i.com/

实际上，生成 Binder IPC 数据的任务是由 IPCThreadState 类负责的。如前所述，与 BpBinder 的 transact()函数相比，IPCThreadState 的 transact()函数额外增加了一个服务 Handle 参数，其代码如下：

```
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;
    err = writeTransactionData(BC_TRANSACTION, flags,
      ➥ handle, code, data, NULL);     ⓾
    ...
    err = waitForResponse(reply);     ⓫
    ...
    return err;
}
```

代码 8-29 | IPCThreadState 的 transact()函数[1]

⓾ 调用 writeTransactionData()函数，创建 binder_transaction_data 数据结构。该数据结构与 Binder 协议一起构成 Binder IPC 数据。

⓫ 调用 waitForResponse()函数，将 Binder IPC 数据传递给 Binder Driver。

writeTransactionData()是 IPCThreadState 类的一个函数，其主要代码如下，让我们一起分析函数代码，了解它是如何创建 Binder IPC 数据的。

```
status_t IPCThreadState::writeTransactionData(int32_t cmd,
    ➥ uint32_t binderFlags, int32_thandle, uint32_t code, const Parcel& data,
    ➥ status_t* statusBuffer)
{
    binder_transaction_data tr;
    tr.target.handle = handle;      ←(a)
    tr.code = code;      ←(b)
    tr.flags = binderFlags;
    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {      ⓬
        tr.data_size = data.ipcDataSize();      ←(c)
        tr.data.ptr.buffer = data.ipcData();      ←(d)
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);      ←(e)
        tr.data.ptr.offsets = data.ipcObjects();      ←(f)
    }
```

[1] frameworks/base/libs/binder/IPCThreadState.cpp

更多更新编程资源尽在 http://www.pin5i.com/

```
    ...
    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));                    ❸
    return NO_ERROR;
}
```

代码 8-30 | IPCThreadState 的 writeTransactionData()函数

IPCThreadState 的 writeTransactionData()函数中使用了 binder_transaction_data 数据结构，该数据结构定义如下：

```
struct binder_transaction_data {
    union {
        size_t handle;
        void *ptr;
    } target;
    void *cookie;
    unsigned int code;
    unsigned int flags;
    pid_t sender_pid;
    uid_t sender_euid;
    size_t data_size;
    size_t offsets_size;
    union {
        struct {
            const void *buffer;
            const void *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

代码 8-31 | binder_transaction_data 数据结构

在代码 8-30❷部分中，将 BpBinder 传递而来的参数（服务 Handle、Binder RPC 代码、Binder RPC 数据）保存到 binder_transaction_data 数据结构的各个成员变量中，如图 8-33 所示。

❸ 保存 Binder 协议（BC_TRANSACTION）与 binder_transaction_data 数据结构到 mOut 中。为了与 Binder Driver 通信，IPCThreadState 持有 mOut 与 mIn 两个成员变量，它们都是 Parcel 类型，其中 mOut 用来发送数据，mIn 用来接收数据。

binder_transaction_data 数据结构中保存的内容如下：

(a) handle 中保存 Context Manager 的服务 Handle 0。

更多更新编程资源尽在 http://www.pin5i.com/

〈 binder_transaction_data 数据结构 〉

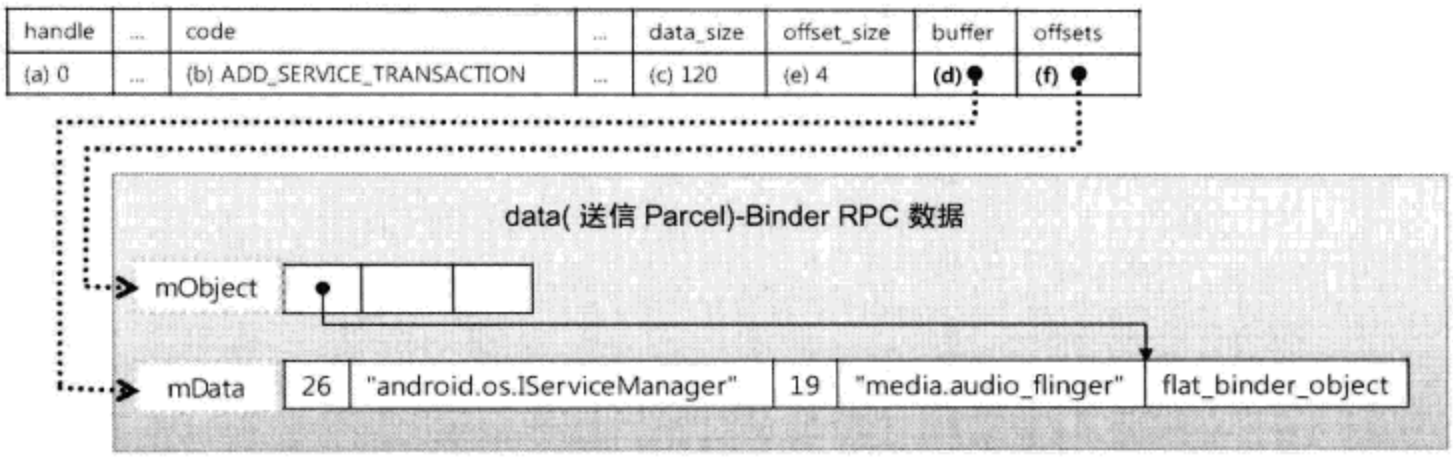


图 8-33 | binder_transaction_data 数据结构（用于注册服务）

(b) code 中保存 ADD_SERVICE_TRANSACTION。Context Manager 根据该数据判断要处理的是服务注册。

(c) data_size 保存数据大小，此处为 120，它与 data（送信 Parcel）的 mData 变量中数据的大小一致。（被保存的数据包括：4（字符串的大小）+26*2（2 位的字符串）+4（字符串大小）+19*2（2 位的字符串）+16（flat_binder_object 数据结构的大小））。

(d) buffer 保存着 data（送信 Parcel）成员变量 mData 的指针。

(e) offset_size 保存着数据 4，它是 data（送信 Parcel）成员变量 mObject 中的数据大小。

(f) offsets 保存着 data（送信 Parcel）成员变量 mObject 的指针。

图 8-33 中的 data（送信 Parcel）保存着 BpServiceManager 的 addService()函数生成的 Binder RPC 数据。如图 8-34 所示，Parcel 类有两个成员变量，分别是 mData 与 mObject，其中 mData 是一块用来保存数据的 buffer，所包含的数据有 Service Manager 的名称、Audio Flinger 的服务名称、Audio Flinger 对象的序列化数据结构 flat_binder_ object。

mObject 是数组形式的 buffer，它包含着 flat_binder_object 数据结构在 mData（mData 所指的 buffer）中的存储位置。元素 mObject[0]保存着第一个 flat_binder_object 数据结构的位置，元素 mObject[1]保存着第二个 flat_binder_object 数据结构的位置。示例中，mObject 仅含有一个数组元素，指向 Audio Flinger 的 flat_binder_object 数据结构。

代码 8-30⓬语句执行完成后，mOut（送信 Parcel）中就保存着 Binder 协议与 binder_transaction_data 数据结构，如图 8-34 所示。

如图所示，在 binder_transaction_data 结构体的 buffer 与 offsets 两个成员变量中分别保存着 data（送信 Parcel）的 mData 与 mObject 两个成员变量的地址。该地址位于发

更多更新编程资源尽在 http://www.pin5i.com/

送端进程（Service Manager）的用户地址空间中。Binder Driver 将其拷贝到接收端进程（Context Manager）的 Binder mmap 区域中。

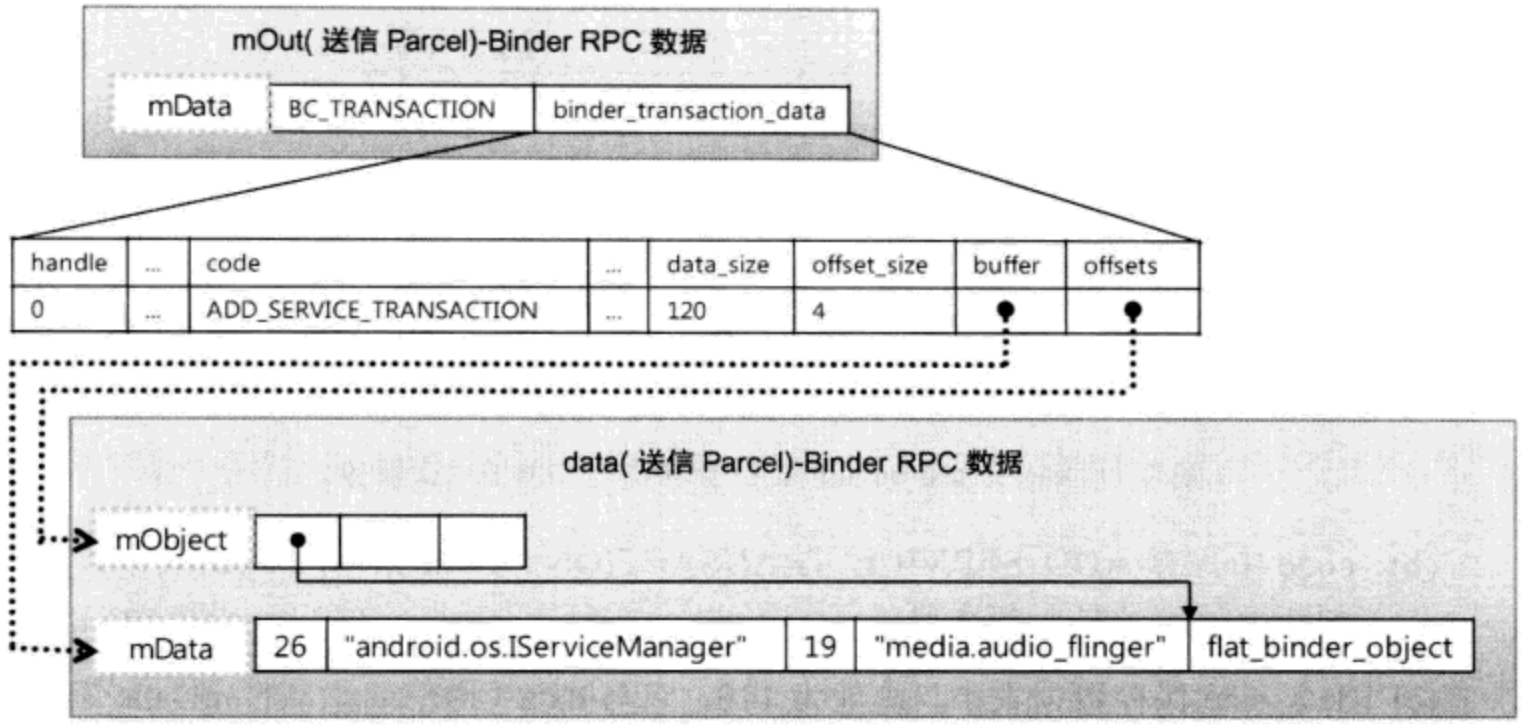


图 8-34 | Binder IPC 数据（被传递到 Binder Driver）

在代码 8-29⓫中，调用 waitForResponse()函数，执行与 Binder Driver 间的 Binder IPC 数据的收发操作。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
   int32_t cmd;
   int32_t err;
   while (1) {
      if ((err=talkWithDriver()) < NO_ERROR) break;      ⓮
      cmd = mIn.readInt32();
      switch (cmd) {
         case BR_REPLY:      ⓯
         {
            binder_transaction_data tr;
            err = mIn.read(&tr, sizeof(tr));
            if (reply) {
               if ((tr.flags & TF_STATUS_CODE) == 0) {
                  reply->ipcSetDataReference(
                     ➥ reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                     ➥ tr.data_size,
                     ➥ reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                     ➥ tr.offsets_size/sizeof(size_t), freeBuffer, this);
               }
               ...
```

代码 8-32 | IPCThreadState 的 waitForResponse()函数

更多更新编程资源尽在 http://www.pingsi.com/

⓮ 调用 talkWithDriver()函数，将保存在 mOut 中的 Binder IPC 数据传递给 Binder Driver，并将来自 Binder Driver 的 Binder IPC 保存至 mIn（收信 Parcel）中。

⓯ 该段代码用来处理接收到的 Binder IPC 数据。调用 mIn.readInt32()函数，读取 Binder 协议，在从 Binder Driver 接收的 Binder 协议中保存着 BR_REPLY，所以继续执行 switch 语句中与 BR_REPLY 相匹配的部分，调用 mIn.read(&tr, sizeof(tr))函数，读取 binder_transaction_data 数据结构。

IPCThreadState 从 Binder Driver 接收 Binder IPC 数据后，保存在 mIn（收信 Parcel）中，所保存的数据内容如图 8-35 所示。

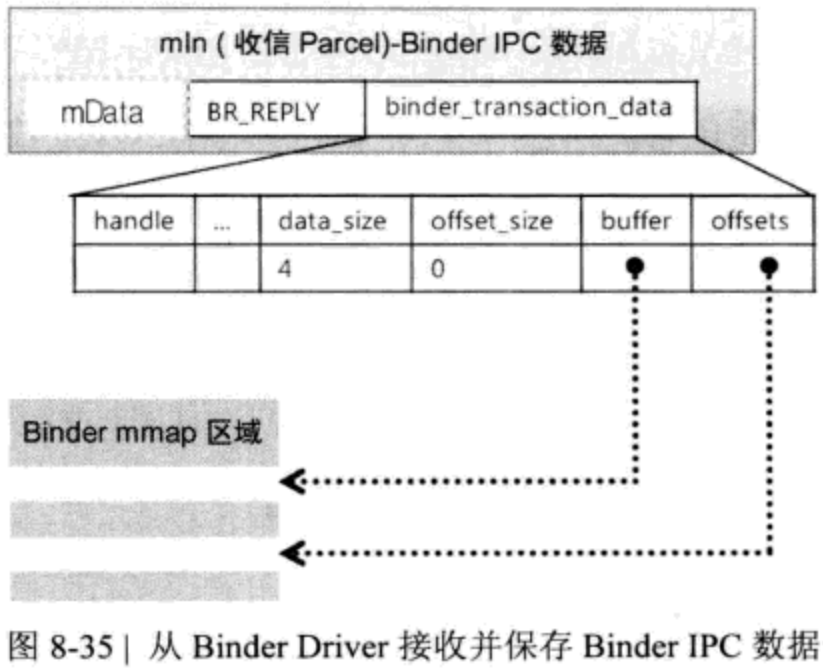


图 8-35 | 从 Binder Driver 接收并保存 Binder IPC 数据

如图所示，binder_transaction_data 的 buffer 与 offsets 指向 Binder mmap 区域中的 Binder RPC 数据。data_size 表示 buffer 中存储的有效数据的大小。在 Context Manager 处理服务注册时，若成功，则返回 0；失败，则返回-1。无论是 0 或-1 都是 int32 类型的，所以 data_size 保存的数据大小为 4。offsets_size 是 offsets 所指的 flat_binder_object 数据结构的大小，若为 0，则表示 RPC 数据中不含有 flat_binder_object 数据结构。

在处理接收到的 Binder IPC 数据时，将调用 Parcel 的 ipcSetDataReference()函数，将接收到的 binder_transaction_data 数据结构设置为 reply（收信 Parcel），相当于接收的 Binder RPC 数据。

```
void Parcel::ipcSetDataReference(const uint8_t* data, size_t dataSize,
    const size_t* objects, size_t objectsCount, release_func relFunc, void* relCookie)
{
    mError = NO_ERROR;
    mData = const_cast<uint8_t*>(data);          ←(a)
    mDataSize = mDataCapacity = dataSize;        ←(b)
    mDataPos = 0;
    mObjects = const_cast<size_t*>(objects);             ←(c)
```

更多更新编程资源尽在 http://www.pin5i.com/

```
        mObjectsSize = mObjectsCapacity = objectsCount;  ←(d)
        ...
    }
```

代码 8-33 | Parcel 的 ipcSetDataReference()函数[1]

在 ipcSetDataReference()函数中，以接收到的 binder_transaction_data 数据结构为基础，设置 reply（收信 Parcel）主要的成员变量，如代码 8-33 所示。

(a) buffer 保存在 mData 中，且该 buffer 持有接收的 Binder RPC 数据的起始地址。

(b) mDataSize 保存着 data_size，data_size 指接收的 Binder RPC 数据的大小。

(c) mObjects 保存着 flat_binder_object 结构体在 Binder RPC 中的存储位置 offsets。

(d) mObjectsSize 保存 Binder RPC 中 flat_binder_object 结构体的个数。

如图 8-36 所示，reply(收信 Parcel)的 mData 变量指向 Binder RPC 数据在 Binder mmap 区域中的起始位置，它是 Context Manager 处理服务注册的结果。

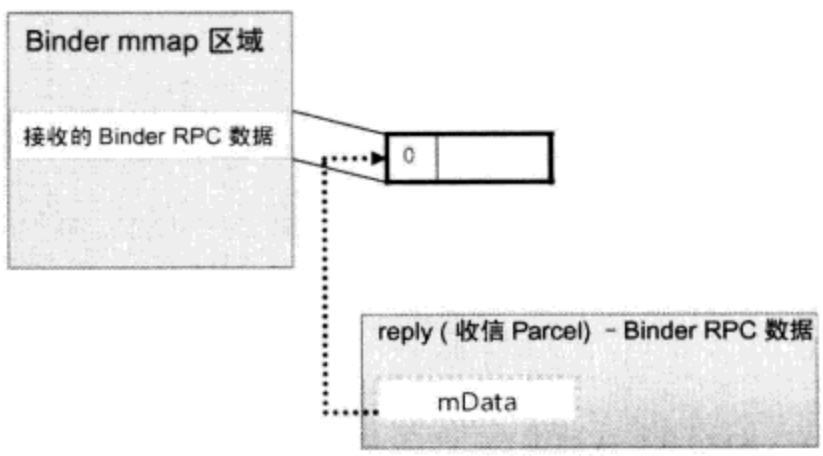


图 8-36 | 接收的 Binder RPC 数据与 reply（收信 Parcel）的 mData 变量

从 reply(收信 Parcel)中以 4 字节(int32 大小)读取数据后，如前所述，Context Manager 执行服务注册后，生成应答数据，即 addService()函数的返回值，如代码 8-24❹所示。

当服务注册完成后，Binder Driver 将为 Audio Flinger 生成新的 Binder 节点，而后 Context Manager 将 Audio Flinger 添加到自身的服务目录中，如图 8-37（2）所示。

通过服务注册的整个过程，我们了解了 Context Manager 与 Service Manager 交换 Binder RPC 数据的过程。

在 Service Framework 中，传递给 Binder Driver 的 Binder IPC 数据包含 Binder 协议与 binder_ transaction_data 两部分。要传递给 Context Manager 的 Binder RPC 数据保存在位于发送端进程（Service Manager）用户内存中的 data（送信 Parcel）中，Binder Driver

[1] frameworks/base/libs/binder/Parcel.cpp

更多更新编程资源尽在 http://www.pudn.com/

将其拷贝到接收端进程（Context Manager）的 Binder mmap 区域中。并且，在 Service Framework 中，从 Binder Driver 接收的 Binder IPC 数据也由 Binder 协议与 binder_transaction_data 两部分构成，Context Manager 的 Binder RPC 数据被保存到接收端进程（Service Manager）的 Binder mmap 区域中，reply（收信 Parcel）将引用它。Binder RPC 数据、Binder RPC 代码以及 Binder IPC 数据的处理过程如图 8-38 所示。

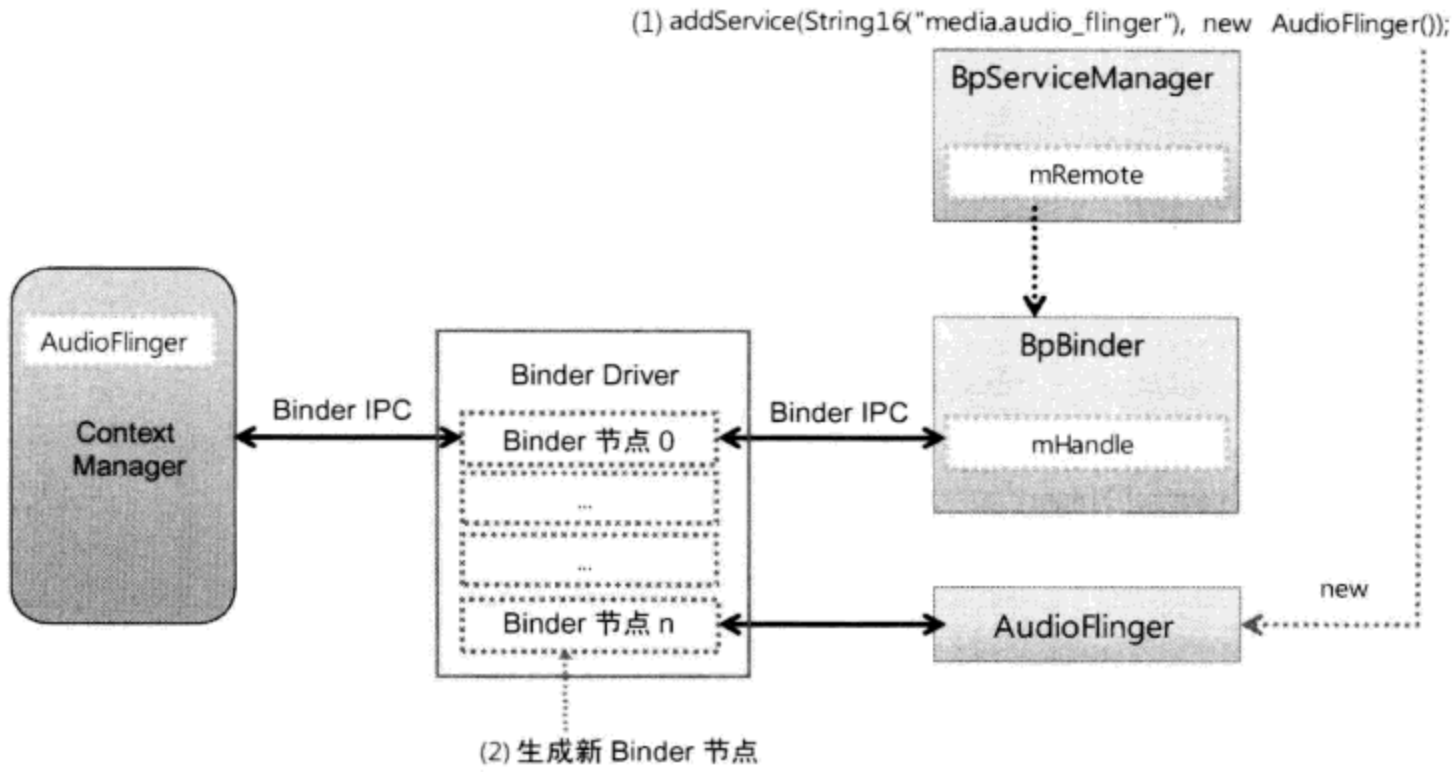


图 8-37 | 完成服务注册

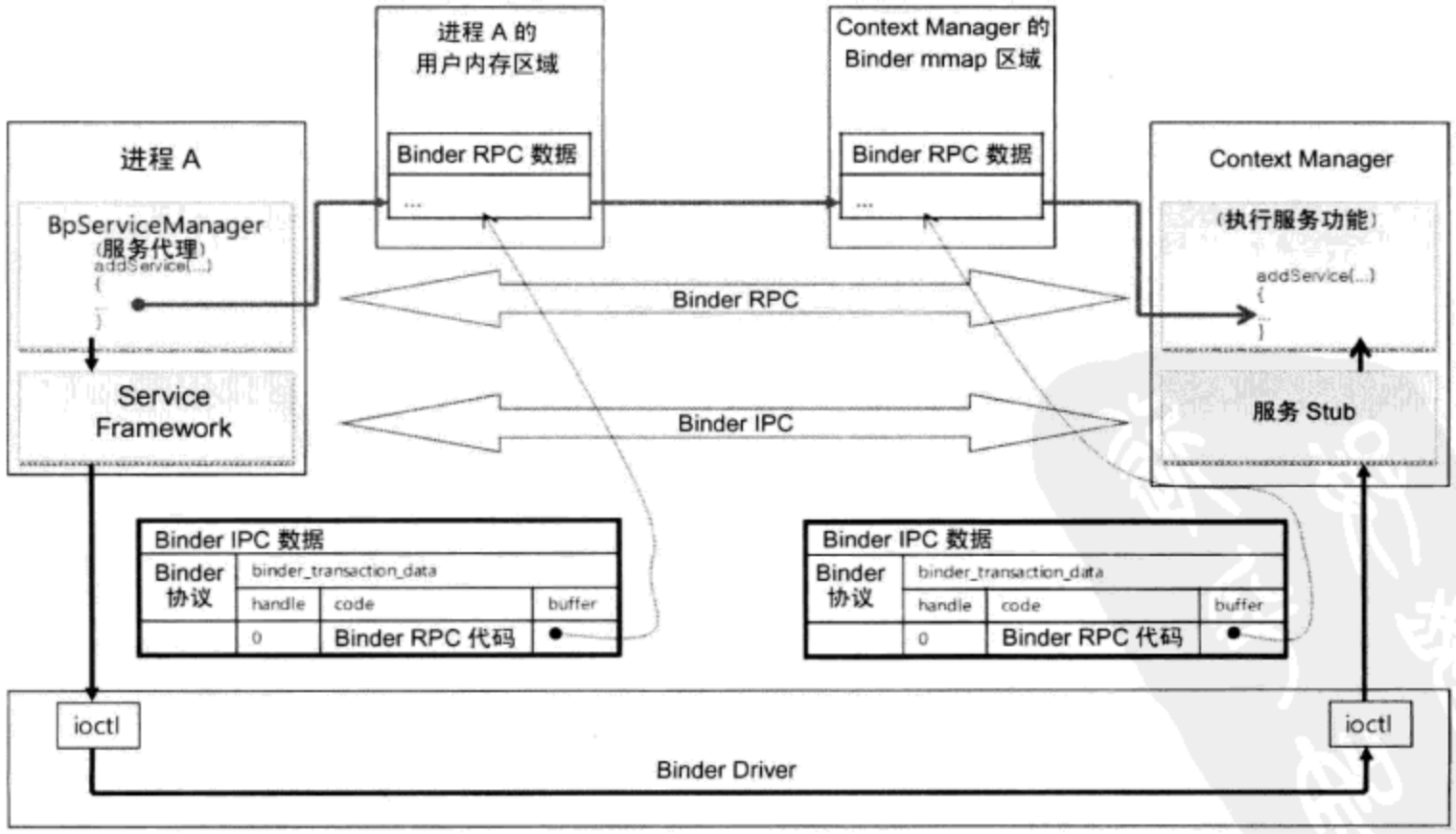


图 8-38 | Binder RPC 数据、代码以及 Binder IPC 数据的处理过程

更多更新编程资源尽在 http://www.pin5i.com/

以上，通过分析代码了解了服务注册的过程。接下来分析服务用户端是如何获取服务信息的，其实获取服务信息的过程与服务注册的过程有些类似。

获取服务信息

服务用户若想访问某个服务，必须先从 Context Manager 获取指定服务的信息。本节以 AudioSystem 的 get_audio_flinger()函数从 Context Manager 获取 Audio Flinger 信息为例，来分析服务用户是如何从 Context Manager 获取服务信息的。

```
const sp<IAudioFlinger>& AudioSystem::get_audio_flinger()
{
    if (gAudioFlinger.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();          ❶
        sp<IBinder> binder;
        binder = sm->getService(String16("media.audio_flinger"));  ❷
        gAudioFlinger = interface_cast<IAudioFlinger>(binder);     ❸
        ...
    }
    return gAudioFlinger;
}
```

代码 8-34 | AudioSystem 的 get_audio_flinger()函数[1]

❶ 调用 defaultServiceManager()函数，获取 BpServiceManager 实例对象的指针。

❷ 调用 Service Manager 的 getService()函数，获取持有 Audio Flinger 服务 Handle 的 BpBinder 对象。getService()函数的参数是 Audio Flinger 服务的名称，用作在 Context Manager 的服务目录中查找指定服务的关键字。

❸ 调用 interface_cast<IAudioFlinger>()，将❷中获取的 BpBinder 对象，转换成用作 Audio Flinger 服务代理的 BpAudioFlinger 对象。

接下来分析代码 8-34❷的内部处理过程。对于语句❸的详细分析说明，将在本节的后半部分进行。

如图 8-39 所示，调用 BpServiceManager 的 getService()函数，根据从 AudioSystem 的 get_audio_ flinger()函数传递而来的参数，生成要传递给 Context Manager 的 Binder RPC 数据（用于获取 Audio Flinger 的服务信息），并引用持有 Context Manager 服务 Handle 的 BpBinder 对象，通过 Binder Driver，将 Binder RPC 数据传送至 Context Manager 中。

[1] frameworks/base/media/libmedia/AudioSystem.cpp

更多更新编程资源尽在 http://www.pi58.com/

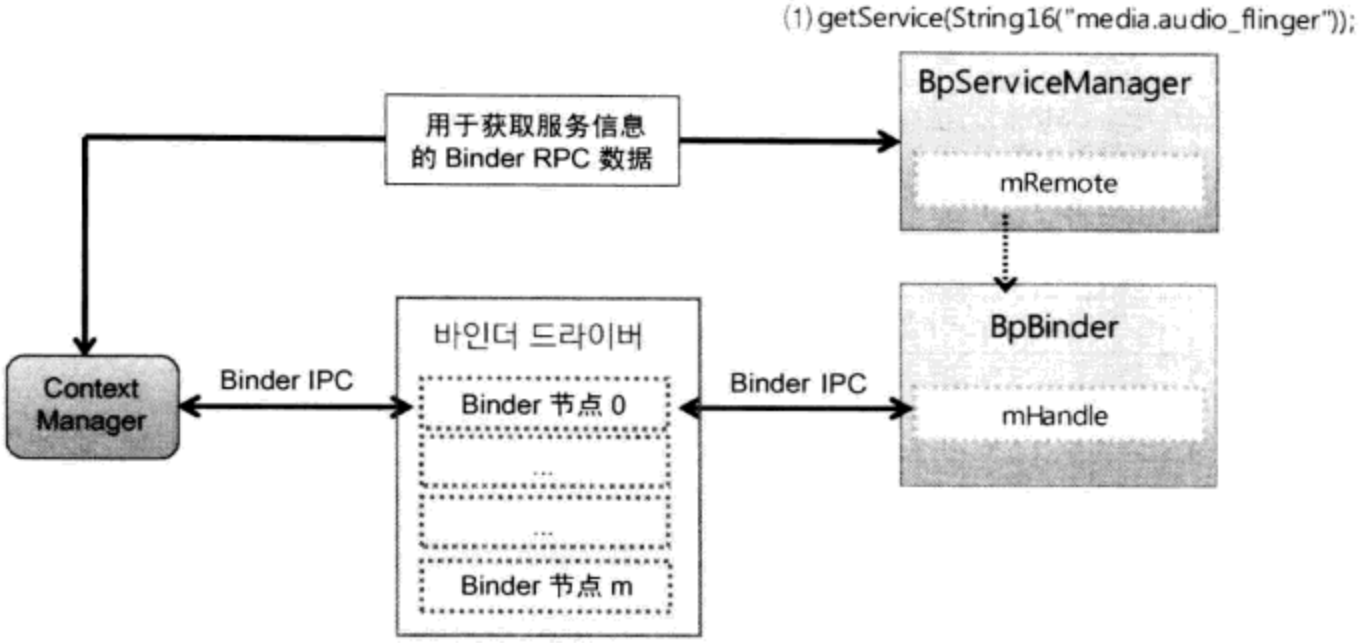


图 8-39 | 获取服务

首先来分析 BpServiceManager 的 getService()函数，函数代码如下：

```
sp<IBinder> getService(const String16& name) const
{
    unsigned n;
    for (n = 0; n < 5; n++){
        sp<IBinder> svc = checkService(name);          ❹
        if (svc != NULL) return svc;
        sleep(1);
    }
    return NULL;
}
```

代码 8-35 | BpServiceManager 的 getService()函数

如代码 8-35❹所示，在 getService()函数中调用了 checkService()函数，并且为了防止 Binder IPC 失败，采用了循环处理语句。

checkService()是 BpServiceManager 类的成员函数，其主要代码如下：

```
sp<IBinder> checkService( const String16& name) const
{
    Parcel data, reply;                                ❺
    data.writeInterfaceToken(
        IServiceManager::getInterfaceDescriptor());    (a)
    data.writeString16(name);                          (b)
    remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);   ❻
    return reply.readStrongBinder();                   ❼
}
```

代码 8-36 | BpServiceManager 的 checkService()函数

更多更新编程资源尽在 http://www.pin5i.com/

❺ 首先定义两个 Parcel 类型的变量，用于发送或接收数据。

如图 8-40 所示，data（发信 $_{Parcel}$）用于存储数据，包含 Service Manager 与 Audio Flinger 的名称，保存的数据与服务注册时的数据相似，但区别在于它不含有 flat_binder_ object 数据结构。

26	"android.os.IServiceManager"	19	"media.audio_flinger"
(a) ServiceManager 的名称		(b) AudioFlinger 的服务名称	

图 8-40 | checkService()的 data（发信 $_{Parcel}$）中存储的数据

❻ 调用 BpBinder 的 transact()函数，传递生成的 Binder RPC 数据与 Binder RPC 代码（CHECK_ SERVICE_TRANSACTION），处理 Binder IPC。

❼ 调用 BpServiceManager 的 checkService()函数将返回 Audio Flinger 的 BpBinder 实例的指针。调用 Parcel 的 readStrongBinder()函数，以 flat_binder_object 数据结构为基础，创建 BpBinder 对象。

这与在服务注册过程中将 Audio Flinger 对象序列化并转换成 flat_binder_object 数据结构的过程完全相反。

下面来分析 BpBinder 的 transact()函数，其中处理 Binder IPC 的代码如下：

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    status_t status = IPCThreadState::self()->transact(
        mHandle, code, data, reply, flags);     ←❽
    if (status == DEAD_OBJECT) mAlive = 0;
    return status;
}
```

代码 8-37 | BpBinder 的 transact()函数

如代码 8-37❽所示，在 BpBinder 的 transact()函数中，继续调用 IPCThreadState 的 transact()函数。并且，IPCThreadState 的 transact()函数的第一个参数 mHandle 是 Context Manager 的服务 Handle。

那么，传递给 Binder Driver 的 Binder IPC 数据是如何生成的呢？Binder IPC 数据的生成过程类似 IPCThreadState 在服务注册时所作的动作。下面让我们来学习这一过程。

binder_transaction_data 数据结构用于获取服务信息，其成员变量如图 8-41 所示。

更多更新编程资源尽在 http://www.pin5i.com/

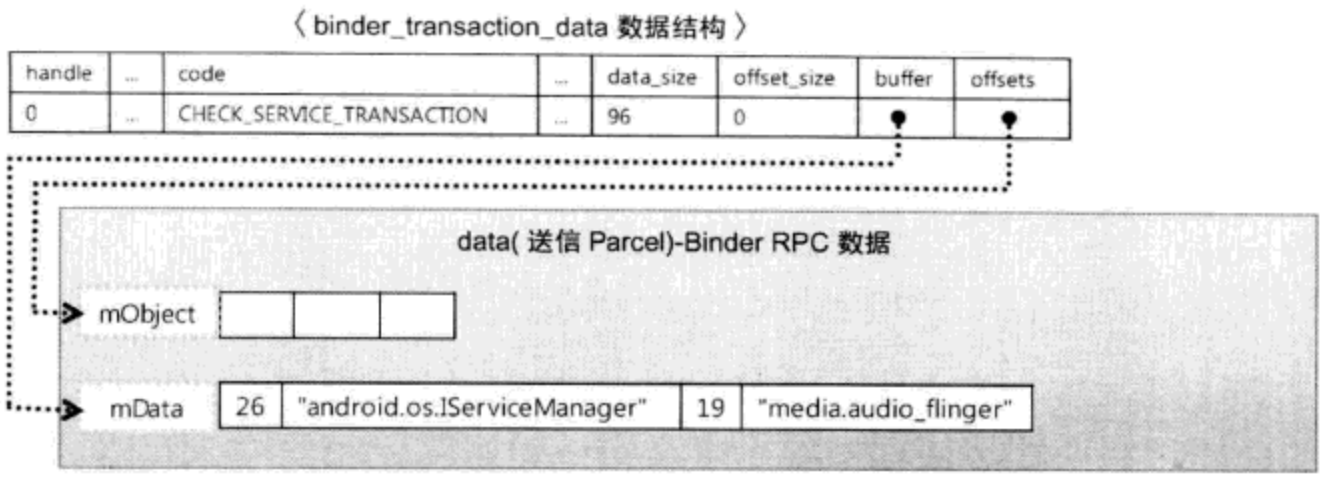


图 8-41 | 用于获取服务的 Binder RPC 数据

data(送信 Parcel)由 BpServiceManager 的 checkService()函数填充数据并传递过来，在其 mData 成员变量中保存着 Service Manager 的名称与 Audio Flinger 的服务名称。并且由于未传递 flat_binder_object 数据结构，mObject 成员变量中是空的，没有数据。（请参看图 8-40）

如图所示，binder_transaction_data 数据结构中保存的数据与服务注册时的数据类似，但是 data_size 与 offset_size 的大小随 Binder RPC 数据而有所不同。并且在服务注册中变量 code 保存的是 ADD_SERVICE_TRANSACTION，而非 CHECK_SERVICE_TRANSACTION，如图 8-41 所示。

然后，调用 IPCThreadState 的 waitForResponse()函数，处理与 Binder Driver 的数据交互过程。

要传送到 Binder Driver 中的 Binder IPC 数据（IPCThreadState 的 mOut(送信 Parcel)）的组成元素如图 8-42 所示。

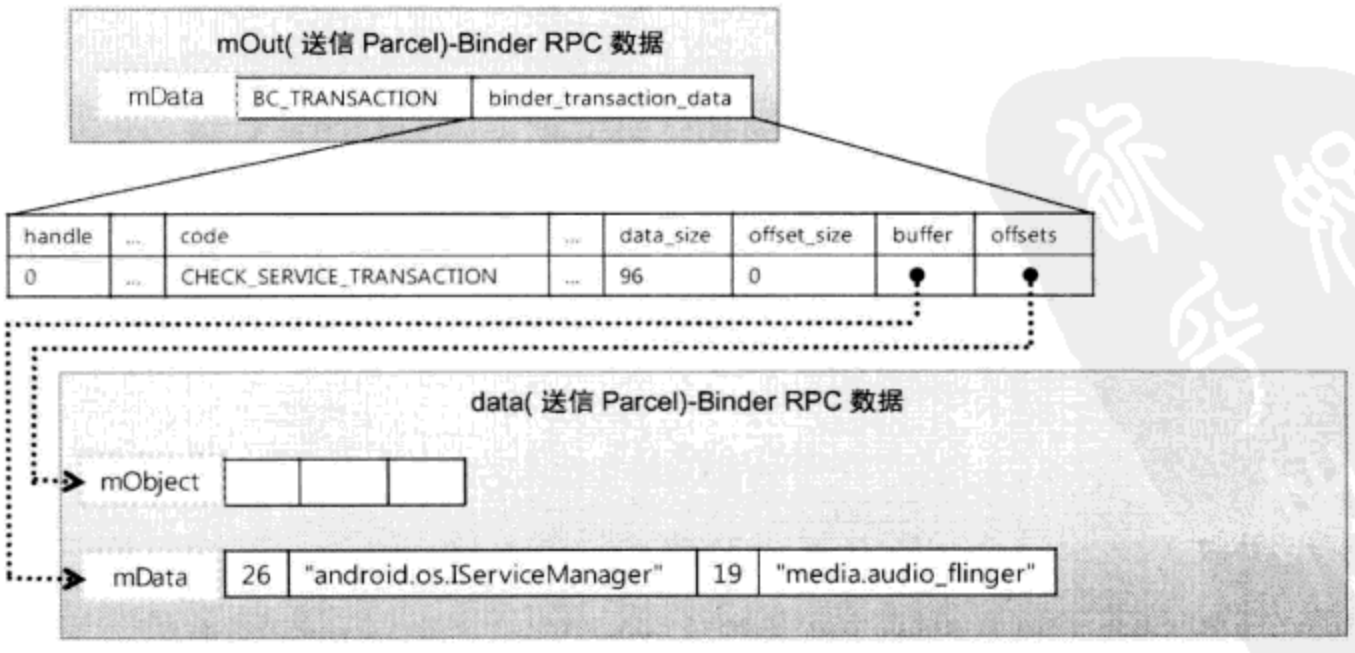


图 8-42 | 向 Binder Driver 传递的 Binder IPC 数据

当获取服务信息后，将从 Binder Driver 收到 Binder IPC 数据（IPCThreadState 的 mIn(收信 Parcel)），该数据的组成结构如图 8-43 所示。

更多更新编程资源尽在 http://www.pin5i.com/

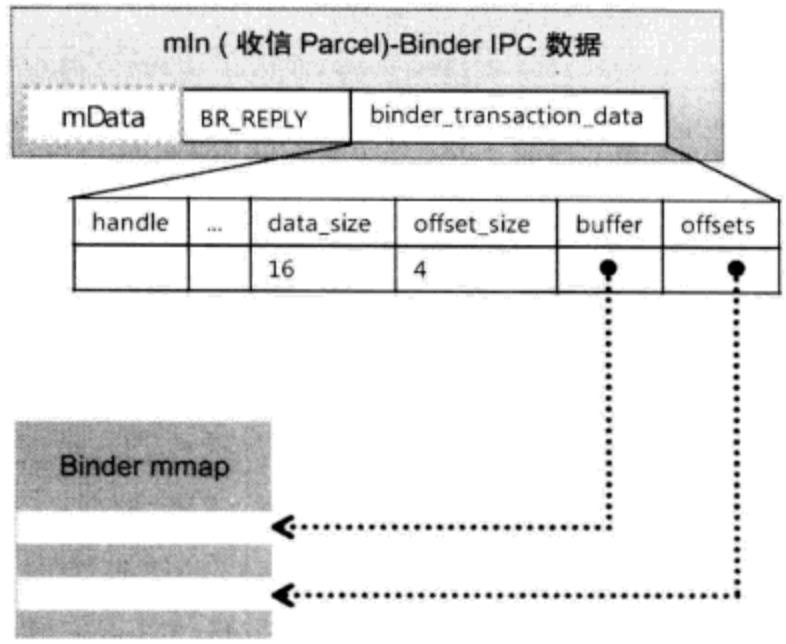


图 8-43 | 从 Binder Driver 收到 Binder IPC 数据

binder_transaction_data 数据结构中的 buffer 与 offsets 分别指向接收的 Binder RPC 数据在 Binder mmap 区域中的位置。data_size 表示 buffer 的有效数据大小，offset_size 表示 flat_binder_object 数据结构的大小，该数据结构保存在 offsets 所指的位置上。与服务注册时不同，offset_size 为 4，表示仅有一个 flat_binder_object 数据结构[1]。作为获取服务的结果，Context Manager 会返回 flat_binder_object 数据结构，该数据结构中包含着指定服务的 Handle。

调用 Parcel 的 ipcSetDataReference()函数，将使用接收到的 binder_transaction_data 数据结构设置 reply(送信 Parcel)，该过程同服务注册过程类似。（请参考代码 8-33）

如图 8-44 所示，位于 Binder mmap 区域中的 Binder RPC 数据是在获取服务信息的过程中 Context Manager 返回的结果，reply(送信 Parcel)的 mData 成员变量指向 Binder RPC 数据在内存中的起始地址。

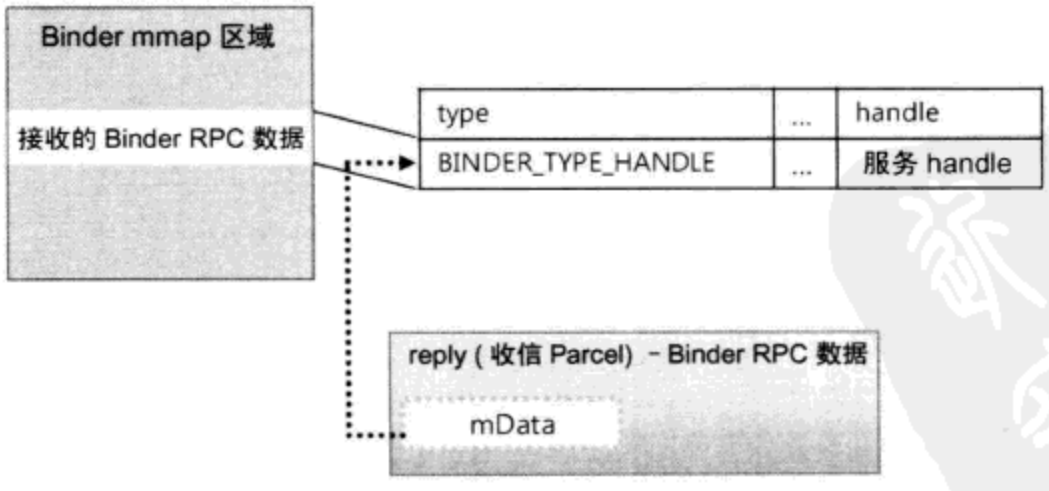


图 8-44 | 接收的 Binder RPC 数据与 reply(送信 Parcel)的 mData 成员变量

[1] offset_size 是指针的大小，该指针指向 flat_binder_object 数据结构所保存的位置。由于指针是四位的，所以正文中指的是一个指向 flat_binder_object 数据结构的指针。

更多更新编程资源尽在 http://www.pin5i.com/

接下来，如图 8-45(2)所示，以从 Binder Driver 接收到的 Context Manager 的 Binder RPC 数据为基础，调用 Parcel 的 readStrongBinder()函数，创建 BpBinder 对象，该对象持有 Audio Flinger 的服务 Handle。

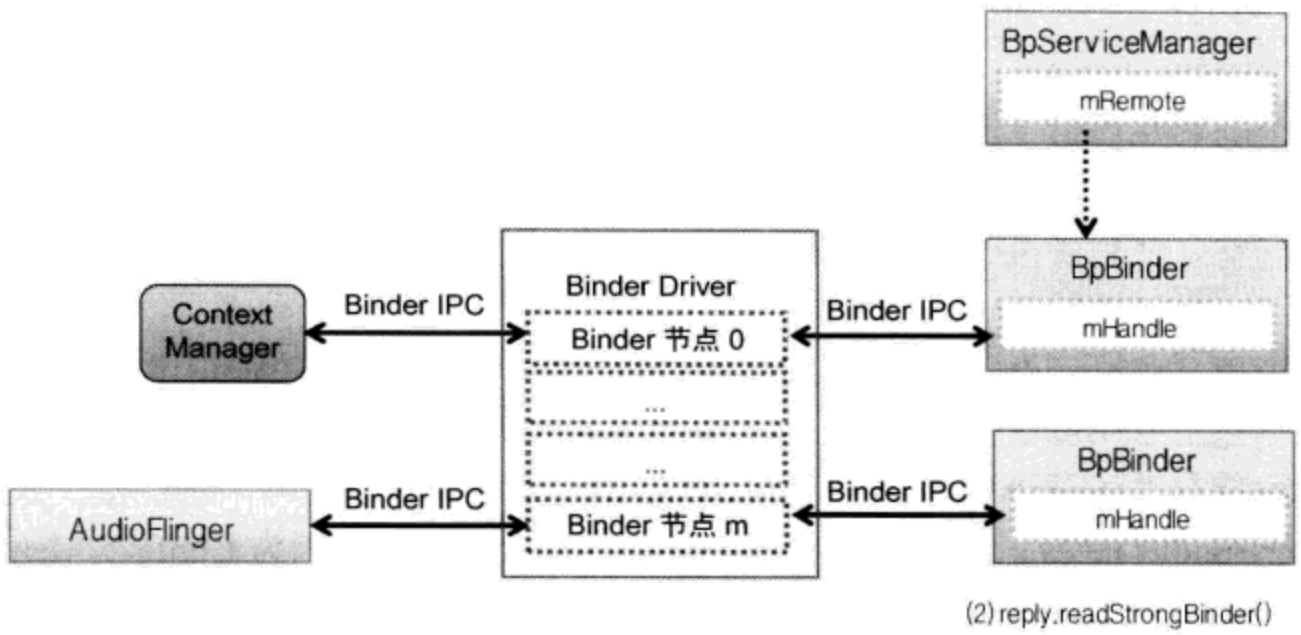


图 8-45 | 获取服务 2

让我们来分析 Parcel 类的 readStrongBinder()函数的代码，主要代码如下：

```
sp<IBinder> Parcel::readStrongBinder() const
{
    sp<IBinder> val;
    unflatten_binder(ProcessState::self(), *this, &val);
    return val;
}
```

代码 8-38 | Parcel 类的 readStrongBinder()函数

如上述代码所示，在 Parcel 的 readStrongBinder()函数中又调用了 unflatten_binder()函数，该函数代码如下：

```
status_t unflatten_binder(const sp<ProcessState>& proc,
const Parcel& in, sp<IBinder>* out)
{
    const flat_binder_object* flat = in.readObject(false);
    if (flat) {
        switch (flat->type) {
            case BINDER_TYPE_BINDER:
                *out = static_cast<IBinder*>(flat->cookie);
                return finish_unflatten_binder(NULL, *flat, in);
            case BINDER_TYPE_HANDLE:        ❾
                *out = proc->getStrongProxyForHandle(flat->handle);
```

更多更新编程资源尽在 http://www.pin5i.com/

```
                return finish_unflatten_binder(
                    static_cast<BpBinder*>(out->get()), *flat, in);
            }
        }
        return BAD_TYPE;
    }
```

代码 8-39 | unflatten_binder()函数

如代码 8-39❾所示，当 flat->type 值为 BINDER_TYPE_HANDLE（请参考图 8-44 中的 Binder RPC 数据）时，继续调用 ProcessState 的 getStrongProxyForHandle()函数。

在调用 getStrongProxyForHandle()函数时，将相关服务的 Handle 作为参数传入函数中，创建出 BpBinder 对象。在 Service Manager 初始化过程中（请参看代码 8-18），讲解创建持有 Context Manager 的服务 Handle 的 BpBinder 对象时，提到过该 getStrongProxyForHandle()函数。当时传入函数的服务 Handle 值为 0，因为 Context Manager 的服务 Handle 为 0。若是其他服务，就要先从 Context Manager 获取指定服务的 Handle，再传入到函数中。

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    handle_entry* e = lookupHandleLocked(handle);          ❿
    if (e != NULL) {
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {      ⓫
            b = new BpBinder(handle);          ⓬
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        }
        ...
    }
    return result;
}
```

代码 8-40 | ProcessState 的 getStrongProxyForHandle()函数

代码 8-40❿保证 ProcessState 不会在同一进程中针对指定的服务生成多个重复的 BpBinder 对象。lookupHandleLocked()函数从参数接收服务 Handle，查找与其相对应的 List Entry 并返回之，若查找不到，则创建 Entry，更新 List 后，将其返回。

⓫中若进程第一次访问 Audio Flinger，b 值为 NULL，继续执行语句⓬，生成持有 Audio Flinger 服务 Handle 的 BpBinder 对象。

更多更新编程资源尽在 http://www.pin5i.com/

接下来，如图 8-46(3)所示，生成 BpAudioFlinger 实例对象，该对象能够引用 BpBinder 实例的指针，并且 BpBinder 实例持有 Audio Flinger 的服务 Handle。

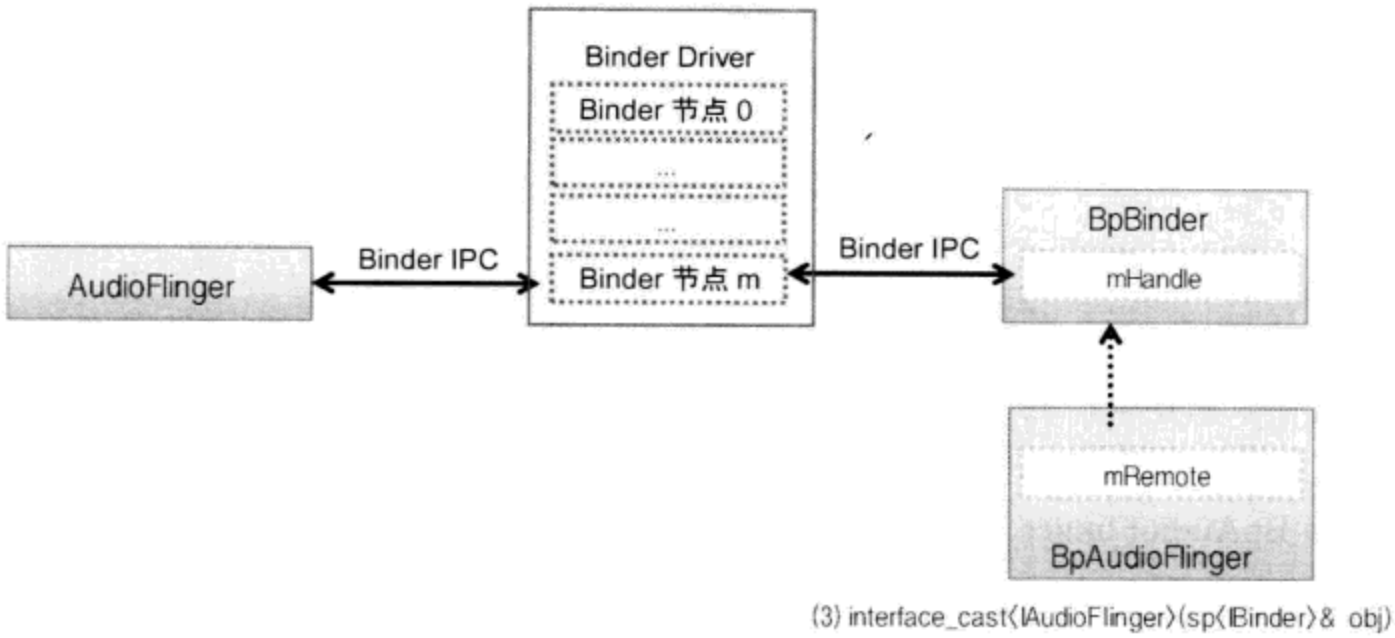


图 8-46 | 生成 BpAudioFlinger 实例对象

在代码 8-34❸的 interface_cast<IAudioFlinger>(binder)中，interface_cast 模板中的 INTERFACE 会被 IAudioFlinger 替换（请参看代码 8-20），所以最终调用的是 Iaudio Flinger 的 asInterface()函数。

并且，IAudioFlinger 类在头文件与实现文件中，分别定义了宏，如下所示。

```
DECLARE_META_INTERFACE(AudioFlinger);
IMPLEMENT_META_INTERFACE(AudioFlinger, "android.media.IAudioFlinger");
```

代码 8-41 | AudioFlinger 的接口宏[1]

当代码 8-41 中的 IMPLEMENT_META_INTERFACE 宏被扩展时，asInterface()代码发生改变，如代码 8-42 所示。观察代码 8-42，可以发现代码与初始化 Service Manager 过程中创建 BpServiceManager 实例对象部分类似。由于定义为同一个宏，通过 Binder IPC 交互的所有对象都定义了这个宏。（请参考代码 8-20～代码 8-22）

下面是 IAudioFlinger 类的 asInterface()函数的主要代码。

```
sp<IAudioFlinger> IAudioFlinger::asInterface(const sp<IBinder>& obj)
```

1 frameworks/base/include/media/IAudioFlinger.h
frameworks/base/media/libmedia/IAudioFlinger.cpp

更多更新编程资源尽在 http://www.pin5i.com/


```
{
    sp<IAudioFlinger> intr;
    if (obj != NULL) {
        intr = static_cast<IAudioFlinger*>(
            obj->queryLocalInterface(IAudioFlinger::descriptor).get());
        if (intr == NULL) {
            intr = new BpAudioFlinger (obj);          ←⓭
        }
    }
    return intr;
}
```

代码 8-42 | IAudioFlinger 类的 asInterface()函数

在代码 8-42⓭中，调用 BpAudioFlinger()构造函数，创建了 BpAudioFlinger 对象。注意该构造函数的参数为前面创建的 BpBinder 实例对象的指针。这样 Audio Flinger 的服务代理 BpAudioFlinger 对象就创建好了。

8.5 编写本地服务

前面我们已经学习了 Android 的 Service Framework，本节将在 Service Framework 的基础上尝试编写 Android 本地服务 HelloWorld。HelloWorld 服务功能很简单，它仅仅输出“hello,world”这一字符串[1]。

8.5.1 设计 HelloWorld 系统服务

请看图 8-47，它是 HelloWorld 系统服务结构的类图。从类图中可以看出，HelloWorld 系统服务由 IHelloWorldService 服务接口、BnHelloWorldService 服务 Stub、HelloWorld Service 服务、BpHelloWorldService 服务代理构成。

源代码路径如下：

- frameworks/base/cmds/helloworld- Service Server、客户端源码
- frameworks/base/include/helloworld- HelloWorldService 头文件(.h)
- frameworks/base/libs/helloworld- HelloWorldService 源文件(.cpp)

[1] 我们在编写 HelloWorld 本地服务时参考了 Michael Richarddson 公布在 github 上的 Android- HelloWorldService 的代码。大家可以去http://github.com/mcr/Android-HelloWorldService页面查看源代码。

更多更新编程资源尽在 http://www.pin5i.com/

图 8-47 | HelloWorld 服务类图

8.5.2 HelloWorld 服务接口

首先，来分析代码 8-43。在 IHelloWorldService.h 头文件中，定义了服务接口类 IhelloWorldService 类，它继承了 IInterface 类。在 IHelloWorldService 类中定义了 DECLARE_META_INTERFACE 宏，也定义了服务函数 helloWorld()。

```
#include <binder/IInterface.h>

namespace android {

//定义 HelloWorldService 使用的 RPC 代码
enum {
HW_HELLOWORLD = IBinder::FIRST_CALL_TRANSACTION,
};

//定义服务接口类 IHelloWorldService
class IHelloWorldService: public IInterface {

   public:
   //服务接口宏
   DECLARE_META_INTERFACE(HelloWorldService);
   //服务函数
   virtual status_t helloWorld(const char *str)=0;
```

更多更新编程资源尽在 http://www.pin5i.com/


```
};
}; // namespace android
```

代码 8-43 | IHelloWorldService.h- IHelloWorldService 类定义

我们可以通过 IInterface 提供的 IMPLEMENT_META_INTERFACE 宏实现在 Ihello WorldService 类的 DECLARE_META_INTERFACE 宏中声明的函数。宏声明代码在 IHelloWorldService.cpp 文件中。

```
#include <helloworld/IHelloWorldService.h>
#include <helloworld/BpHelloWorldService.h>

namespace android {

//接口宏
IMPLEMENT_META_INTERFACE(HelloWorldService, "android.apps.IHelloWorldService");

};// namespace android
```

代码 8-44 | IHelloWorldService.cpp –IhelloWorldService 服务接口的实现代码

8.5.3 HelloWorld 服务

BnHelloWorldService 服务 Stub

BnHelloWorldService 服务 Stub 类用来处理从服务用户接收到的 RPC 代码，它继承了 BnInterface 类，并且重新定义了 BBinder 类的 onTransact()方法。BnHello World Service 服务 Stub 类的定义在 BnHelloWorldService.h 文件中，如代码 8-45 所示。

```
#include <binder/Parcel.h>
#include <helloworld/IHelloWorldService.h>

namespace android {
class BnHelloWorldService : public BnInterface<IHelloWorldService>
{
   public:
   //重定义 BBinder 类的 onTransact() 方法
   virtual status_t onTransact( uint32_t code,
                ➥ const Parcel& data,
                ➥ Parcel* reply,
                ➥  uint32_t flags = 0);
};
}; // namespace android
```

代码 8-45 | BnHelloWorldService.h -BnHelloWorldService 服务 Stub 类的定义

更多更新编程资源尽在 http://www.pidb.com/

代码 8-46 是 onTransact()方法的具体实现。当传入 onTransact()方法的 RPC 代码为 HW_HELLOWORLD 时，通过 CHECK_INTERFACE 宏，确认请求的是否为 Hello World 服务接口，而后读取来自服务代理的输出字符串。然后，调用 helloWorld()方法，将读取的字符串传入函数中。

```
#include <helloworld/BnHelloWorldService.h>
#include <binder/Parcel.h>

namespace android {

//重定义处理 RPC 代码的 onTransact() 方法
status_t BnHelloWorldService::onTransact(uint32_t code,
            ➥    const Parcel &data,
            ➥    Parcel *reply,
            ➥    uint32_t flags)
{
   switch(code) {

   //接收的 HW_HELLOWORLD RPC 代码
   case HW_HELLOWORLD: {
      //确认为正确的服务请求
      CHECK_INTERFACE(IHelloWorldService, data, reply);
      const char *str;
      //读取来自服务用户端的字符串
      str = data.readCString();
      //调用输出方法，向标准输出设备输出接收到的字符串
      reply->writeInt32(helloWorld(str));
      return NO_ERROR;
   } break;
   default:
      return BBinder::onTransact(code, data, reply, flags);
   }
}
};// namespace android
```

代码 8-46 | BnHelloWorldService.cpp –HelloWorld 服务 Stub 源码

HelloWorldService 服务类

请看代码 8-47，在代码中定义了 HelloWorldService 类，它继承了 BnHelloWorld Service 类。在 HelloWorldService 类中首先声明了 instantiate()函数，该函数用来生成 HelloWorld 服务。并且 HelloWorldService 的构造函数被声明为私有的（private），防止使用 new 关键字随意生成 HelloWorld 服务。在 instantiate()函数内部，调用 Service Manager 的 addService()方法，在生成 HelloWorld 服务的同时将其注册到系统中。紧接着，声明虚函数 helloWorld()，该函数用来将接收到的字符串输出到标准设备上。

```
#include <binder/Parcel.h>
```

更多更新编程资源尽在 http://www.pin5i.com/

```
#include <helloworld/BnHelloWorldService.h>
#include <utils/Log.h>

namespace android {
class HelloWorldService : public BnHelloWorldService
{
public:
   //初始化 HelloWorldService. 向系统注册服务
   static void instantiate();
   //具体实现服务接口的方法
   virtual status_t helloWorld(const char *str);
   //调用 BnHelloWorldService() 的 onTransact() 方法
   virtual status_t onTransact(
            ➥  uint32_t code,
            ➥  const Parcel& data,
            ➥  Parcel* reply,
            ➥  uint32_t flags);

private:
   //将构造函数设置为私有函数，防止使用 new 运算符直接创建对象
   HelloWorldService();
   virtual ~HelloWorldService();
};
};// namespace android
```

代码 8-47 | HelloWorldService.h –HelloWorldService 类定义

具体实现 HelloWorldService 类的代码位于 HelloWorldService.cpp 文件中，如代码 8-48 所示。在源代码中，可以看到 instantiate()函数调用了 Service Manager 的 addService()函数，将生成的 HelloWorld 服务注册到系统中；在 helloWorld()方法中使用 printf()函数，将接收到的字符串输出到标准设备中。

```
#include <binder/IServiceManager.h>
#include <binder/IPCThreadState.h>
#include <helloworld/BnHelloWorldService.h>
#include <helloworld/HelloWorldService.h>
#include <utils/Log.h>

namespace android {

void HelloWorldService::instantiate() {
   defaultServiceManager()->addService(
   String16("android.apps.IHelloWorldService"), new HelloWorldService());
}

status_t HelloWorldService::helloWorld(const char* str) {
   LOGI("%s\n", str);
   printf("%s\n", str);
   return NO_ERROR;
}
```

更多更新编程资源尽在 http://www.pidi8.com/


```
HelloWorldService::HelloWorldService(){
   LOGI("HelloWorldService is created");
}

HelloWorldService::~HelloWorldService(){
   LOGI("HelloWorldService is destroyed");
}
// ----------------------------------------------------------

status_t HelloWorldService::onTransact(
      uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
   return BnHelloWorldService::onTransact(code, data, reply, flags);
}
// ----------------------------------------------------------

}; // namespace android
```

代码 8-48 | HelloWorldService.cpp –实现 HelloWorldService 服务类

8.5.4 HelloWorld 服务代理

在使用 HelloWorld 服务的过程中，需要使用 BpHelloWorldService 服务代理。BpHelloWorld Service 类继承了 BpInterface 类，在 BpHelloWorldService.h 头文件中进行声明，如代码 8-49 所示。并且，使用 BpInterface<IHelloWorldService>模板语句，同时继承了 IhelloWorldService 类。

```
#include <binder/Parcel.h>
#include <helloworld/IHelloWorldService.h>

namespace android{
class BpHelloWorldService: public BpInterface<IHelloWorldService>
{
public:
   BpHelloWorldService (const sp<IBinder>& impl);
   virtual status_t helloWorld(const char *str);
};

}; // namespace android
```

代码 8-49 | BpHelloWorldService.h –声明 HelloWorld 服务代理

请看代码 8-50，它是 HelloWorld 服务代理的实现代码。观察源代码，可以看到 helloWorld()服务代理函数首先调用 IHelloWorldService 的 getInterfaceDescriptor()函数获取服务接口名称，将其保存到发送数据中，然后把要输出的字符串保存到待发送的数据中。最后通过 remote()方法获取 IBinder 对象，以 HW_HELLOWORLD RPC 代码

更多更新编程资源尽在 http://www.pin5i.com/

与待发送的数据为参数调用 transact()方法。

```
#include <binder/Parcel.h>
#include <helloworld/BpHelloWorldService.h>

namespace android{

status_t BpHelloWorldService::helloWorld(const char *str) {
   Parcel data, reply;
   //将服务接口名称保存到发送数据中
   data.writeInterfaceToken(
      IHelloWorldService::getInterfaceDescriptor());
   //将输出字符串保存到发送数据中
   data.writeCString(str);
   //调用 BpBinder 类的 transact() 方法
   status_t status = remote()->transact(HW_HELLOWORLD, data, &reply);
   if (status != NO_ERROR) {
      LOGE("print helloworld error: %s", strerror(-status));
   } else {
      //读取 helloWorld() 方法的调用结果
      status = reply.readInt32();
   }
   return status;
}

BpHelloWorldService::BpHelloWorldService (const sp<IBinder>& impl)
   : BpInterface<IHelloWorldService>(impl)
{}

}; // namespace android
```

代码 8-50 | BpHelloWorldService.cpp –实现 HelloWorld 服务代理

最后，使用 reply 变量处理 helloWorld()函数调用的结果。注意 transact()函数的返回值表示服务端的 transact()函数被正常调用，保存在 reply 数据中返回来的值实际就是 helloWorld()函数的返回值。

8.5.5 运行 HelloWorld 服务

在创建好 HelloWorldService 服务后，如何运行它呢？运行 HelloWorldService 服务，需要有服务客户端与 Service Server 进程，服务代理运行在服务客户端中，服务运行在 Service Server 进程中，如图 8-48 所示。

首先生成 Service Server 进程，运行 HelloWorldService 服务。请看代码 8-51，它是 main_helloworldservice.cpp 中 main()函数的代码。

在 main()函数中，首先调用 HelloWorldService 的 instantiate()函数，创建 Hello

更多更新编程资源尽在 http://www.p5i.com/

WorldService 的实例对象，并将 HelloWorld 服务注册到系统中。而后调用 ProcessState 的 startThreadPool()方法创建线程池，再调用 IPCThreadState 的 joinThreadPool()方法，在线程池中等待接收服务用户请求。

图 8-48 | 服务客户端与 Service Server 进程

```
#define LOG_TAG "main_helloworldservice"

#include <binder/IPCThreadState.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include <utils/Log.h>

#include <helloworld/HelloWorldService.h>

using namespace android;

int main(int argc, char *argv[])
{
    HelloWorldService::instantiate();
    ProcessState::self()->startThreadPool();
    LOGI("HelloWorldService is starting now");
    IPCThreadState::self()->joinThreadPool();
    return 0;
}
```

代码 8-51 | main_ helloworldservice.cpp 的 main()函数

服务客户端使用 HelloWorld 服务的过程略微有些复杂。如代码 8-52 所示，首先调用 defaultServiceManager()函数，获取 Service Manager 后，再调用 Service Manager 的 getService()函数，获取指向 HelloWorld 服务的 BpBinder 对象。由于在系统启动时无法提前预知 HelloWorld 服务准确的启动时间，所以在服务用户端应该进行相应处理，直至 getService()函数返回 BpBinder 对象。在获取 BpBinder 对象后，使用 interface_cast 宏，将其转换为 BpHelloWorldService 服务代理。最后调用服务代理的 helloWorld()函数，传入字符串“hello, world”后，将其输出到标准设备中。

```
#define LOG_TAG "main_helloworldclient"

#include <binder/IPCThreadState.h>
```

更多更新编程资源尽在 http://www.pin5i.com/

```
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>

#include <utils/Log.h>
#include <utils/RefBase.h>
#include <helloworld/IHelloWorldService.h>

using namespace android;

int main(int argc, char *argv[])
{
    LOGI("HelloWorldService client is now starting");

    sp<IServiceManager> sm = defaultServiceManager();
    sp<IBinder> b;
    sp<IHelloWorldService> sHelloWorldService;

    do {
        b = sm->getService(String16("android.apps.IHelloWorldService"));
        if (b != 0)
        break;
        LOGI("HelloWorldSerivce is not working, waiting...");
        usleep(500000);
    } while(true);

    sHelloWorldService = interface_cast<IHelloWorldService>(b);
    sHelloWorldService -> helloWorld("hello, world");

    return(0);
}
```

代码 8-52 | main_helloworldservice.cpp-服务客户端调用 HelloWorld 服务的代码

编译 HelloWorld 系统服务

若想编译 HelloWorld 服务代码，需要将 helloworld 源代码包含进 Android 平台源码中，再进行整体编译。首先将 HelloWorld 服务的头文件放到/frameworks/base/include/helloworld 目录下，源代码文件放到/frameworks/base/libs/helloworld 目录下；而后编写 Android.mk 文件，如代码 8-53 所示，编写完成后，将其放到源代码目录中，以便将源代码与 Android 平台代码一起编译。

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    IHelloWorldService.cpp \
    BnHelloWorldService.cpp \
    BpHelloWorldService.cpp \
    HelloWorldService.cpp

LOCAL_SHARED_LIBRARIES := \
```

更多更新编程资源尽在 http://www.ptmind.com/


```
    libcutils \
    libutils \
    libbinder \

LOCAL_PRELINK_MODULE := false

LOCAL_MODULE:= libhelloworld

include $(BUILD_SHARED_LIBRARY)
```

代码 8-53 | 编写用于编译 HelloWorld 服务的 Android.mk 文件

接着，将 Service Server 与客户端源代码文件放到/frameworks/base/cmds/helloworld/目录下，编写如代码 8-54 所示的 Android.mk 文件，将它也放入相同的目录下。

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
  main_helloworldservice.cpp

LOCAL_SHARED_LIBRARIES := \
    libutils \
    libbinder \
    libhelloworld
    base := $(LOCAL_PATH)/../..

LOCAL_MODULE:= helloworldservice

include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
    main_helloworldclient.cpp

LOCAL_SHARED_LIBRARIES := \
    libutils \
    libbinder \
    libhelloworld

base := $(LOCAL_PATH)/../..
    LOCAL_MODULE:= helloworldclient

include $(BUILD_EXECUTABLE)
```

代码 8-54 | 编写用于编译 HelloWorld 的 Service Server 与客户端的 Android.mk 文件

关于编译 Android 平台代码的内容，在第 2 章“搭建 Android 开发环境”中已作介绍，在此不再赘述，若遗忘，请回顾第 2 章中的相关内容。

更多更新编程资源尽在 http://www.pin5i.com/

TIP 如何运行系统服务

我们知道系统服务运行在 Service Server 进程中，Android 启动时将自动运行记录在 init.rc 脚本文件服务段落中的 Service Server 进程，在该进程中运行着本地系统服务。若想运行系统服务，可以将运行指定服务的Service Server进程记录到init.rc文件的服务段落中，以启动指定的系统服务；也可以通过把创建服务的代码添加到已处于运行状态的 Service Server 进程中的方式运行服务。

系统服务运行结果

编译成功后，将生成一个名称为 system.img[1]的文件，将其复制到<ANDROID_SDK>[2]/platforms/android-8/images 目录下，运行模拟器。然后在 Command Prompt 中使用 adb shell 命令连接至 Shell Prompt，运行 HelloWorld 服务。注意由于 HelloWorld 服务的 Service Server 与客户端编译后的运行文件在/system/bin 文件夹中，所以首先转到相关目录下再运行服务。

首先在后台运行 helloworldservice，用它来生成 HelloWorld 服务。而后运行 helloworldclient，它将创建使用 HelloWorld 服务的服务代理。此时，HelloWorld 服务的用户将向服务请求 helloWorld()功能，服务就会调用 helloWorld()函数。

请看图 8-49，它显示的是 HelloWorld 服务程序运行的结果。根据以上说明顺序，执行相应命令，即可得到所期望的运行结果，输出字符串“hello,world”。

图 8-49 Helloworld 服务的运行界面

下面是通过 DDMS 确认的日志信息。

```
INFO/(315): HelloWorldService is created
INFO/main_helloworldservice(315): HelloWorldService is starting now
INFO/main_helloworldclient(317): HelloWorldService client is now starting
INFO/(315): hello, world
```

---

1 /out/target/product/generic/system.img

2 Android SDK 的设置目录。

更多更新编程资源尽在 http://www.pin5i.com/

8.6 小结

本章学习了与 Android 本地服务 Framework 相关的内容，下面再次梳理一下本章的全部内容，作个小结。

首先我们学习了什么是 Service Framework，它提供了哪四项功能，各层有哪些构成元素、各个类的结构以及它们之间是如何相互作用的。为了帮助大家理解 Service Framework 的运作机制，以 Audio Flinger 为示例分析了服务接口、服务、服务代理类的源代码，有助于大家进一步理解相关内容。

其次，我们也介绍了本地服务 Framework 中的 Service Manager，它用来将运行在 Android 平台中的系统服务注册到 Context Manger 之中。并且在 Service Manager 一节中，还学习了管理服务信息及目录的 Context Manager 与 C/C++层中的 Service Manager 是如何通过 Binder RPC 进行通信的。

Context Manager 是 Android 平台中最初运行的服务，它在 Service Framework 中的服务 Handle 为 0，且在服务与服务用户间提供服务注册与获取服务等本身特有的功能。

服务代理 BpServiceManager 与 Context Manager 通过 Binder RPC 进行调用，分析了服务注册与获取的整个过程，在该过程中，我们了解了服务用户在访问服务时是如何连接服务 Handle、BpBinder，以及服务代理的。

最后，我们使用 Native Service Framework 编写了一个名称为 HelloWorldService 的服务。在该服务示例中，服务客户端使用 HelloWorldService 服务的使用者调用 helloWorld()方法，Service Server 的 HelloWorldService 服务就会输出“hello,world”字符串。

虽然编写的 HelloWorldService 服务非常简单，但它却体现了使用 Service Framework 编写服务的流程。请大家认真学习示例，领悟其中的精要，以便将其应用到编程过程中，编写出运行在 Android 终端机上的系统服务，实现所需要的功能。

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 9 章

本地系统服务（Native System Service）分析

9.1 相机服务（Cameral Service）

本地系统服务是使用 C++语言编写的，它在 Android Framework 四层结构中处于本地库层。本章通过分析相机服务，来学习 Android Service Framework 是由哪些本地系统服务组成的，以及应用程序框架层与本地系统服务之间是如何协作、如何进行交互的。

近来，越来越多的移动设备中配备了相机，为用户提供拍照、摄影等服务，受到广大用户的喜爱与追捧，相机俨然已成为移动设备的必备设备之一。在搭载 Android 系统的设备中，Android 系统提供相机服务，它是典型的本地系统服务之一，专门用来支持手持设备中的相机设备。在谷歌的 CDD（Compatibility Definition Document，兼容性定义文档）[1]中，定义了一系列的 CTS（Compatibility Test Suit）要求事项，只有通过 CTS 测试的设备才有可能获得 Android 的商标和享受 Android Market 的权限。在该文档中指出移动设备中搭载的相机设备像素必须在 200 万以上。

在 Android SDK 基本包中包含相机应用程序。图 9-1 是 SDK 包中相机应用程序在模拟器环境中运行的画面，开发者在开发与相机服务相关的应用程序或分析研究相机服务的组成结构时，即使实际移动设备中未搭载相机设备，仍然可以通过模拟器来进行开发或研究学习，非常方便。

1 http://source.android.com/compatibility/downloads.html

更多更新编程资源尽在 http://www.pin5i.com/

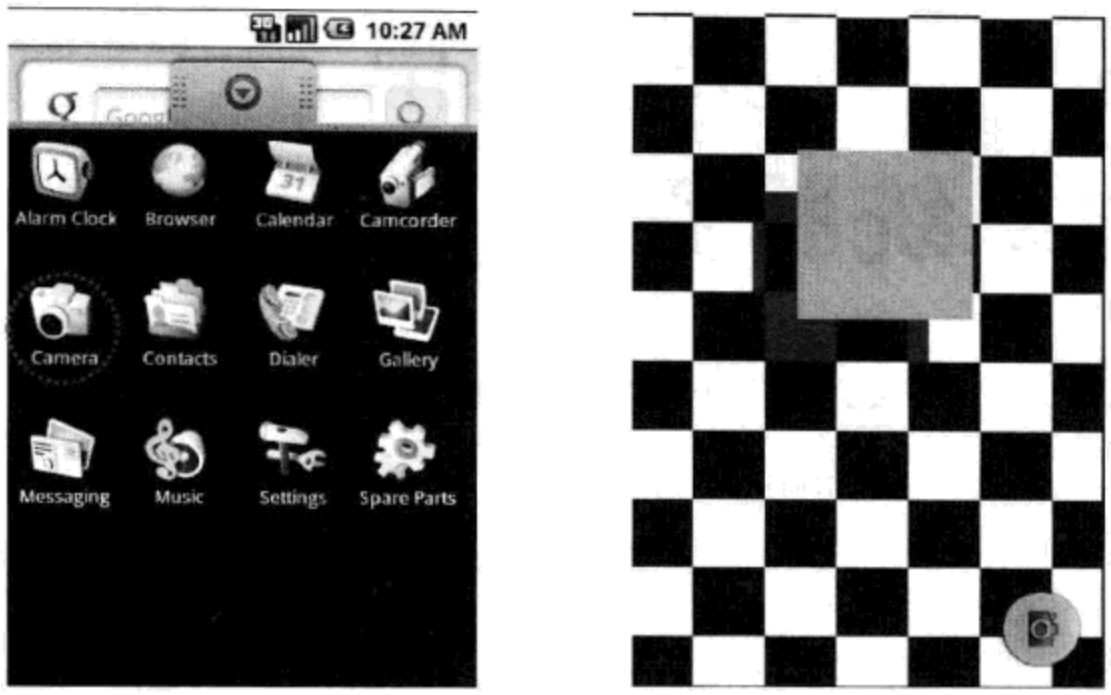


图 9-1 | 运行相机应用程序

9.2 相机应用程序

在分析相机服务框架之前，我们先研究一下 ApiDemos 中的 CameraPreview[1]示例代码，了解应用程序是如何使用相机服务的。

在应用程序访问相机服务之前，必须先在 AndroidManifest.xml 文件中设置 CAMERA 权限，如代码 9-1 所示。

```
<uses-permission android:name="android.permission.CAMERA"/>
```

代码 9-1 | 应用程序的 AndroidManifest.xml 文件

TIP Android 设备访问权限

在 Android 系统中对于设备节点的访问权限通常使用 Linux 用户 ID 或组 ID 来控制，一个应用程序在访问相关设备时，系统必须明确地向应用程序提供相关设备的访问权限。若应用程序不具有访问某个设备的权限，那么系统服务就会拒绝执行应用程序提出的各种请求。为了防止黑客恶意入侵对操作系统或应用程序造成不良影响，设置应用程序时，用户必须认真查看应用程序，赋予应用程序相应的权限。

CameraPreview 通过应用程序框架中的 android.hardware.Camera 类来访问相机服

1 development/samples/ApiDemos/src/com/example/android/apis/graphics/CameraPreview.java

更多更新编程资源尽在 http://www.pin5i.com/

务。那么，在创建应用程序的 Surface（绘画对象），更改或删除属性的过程中，相机服务如何使用呢？图 9-2 展示了使用相机服务的过程。

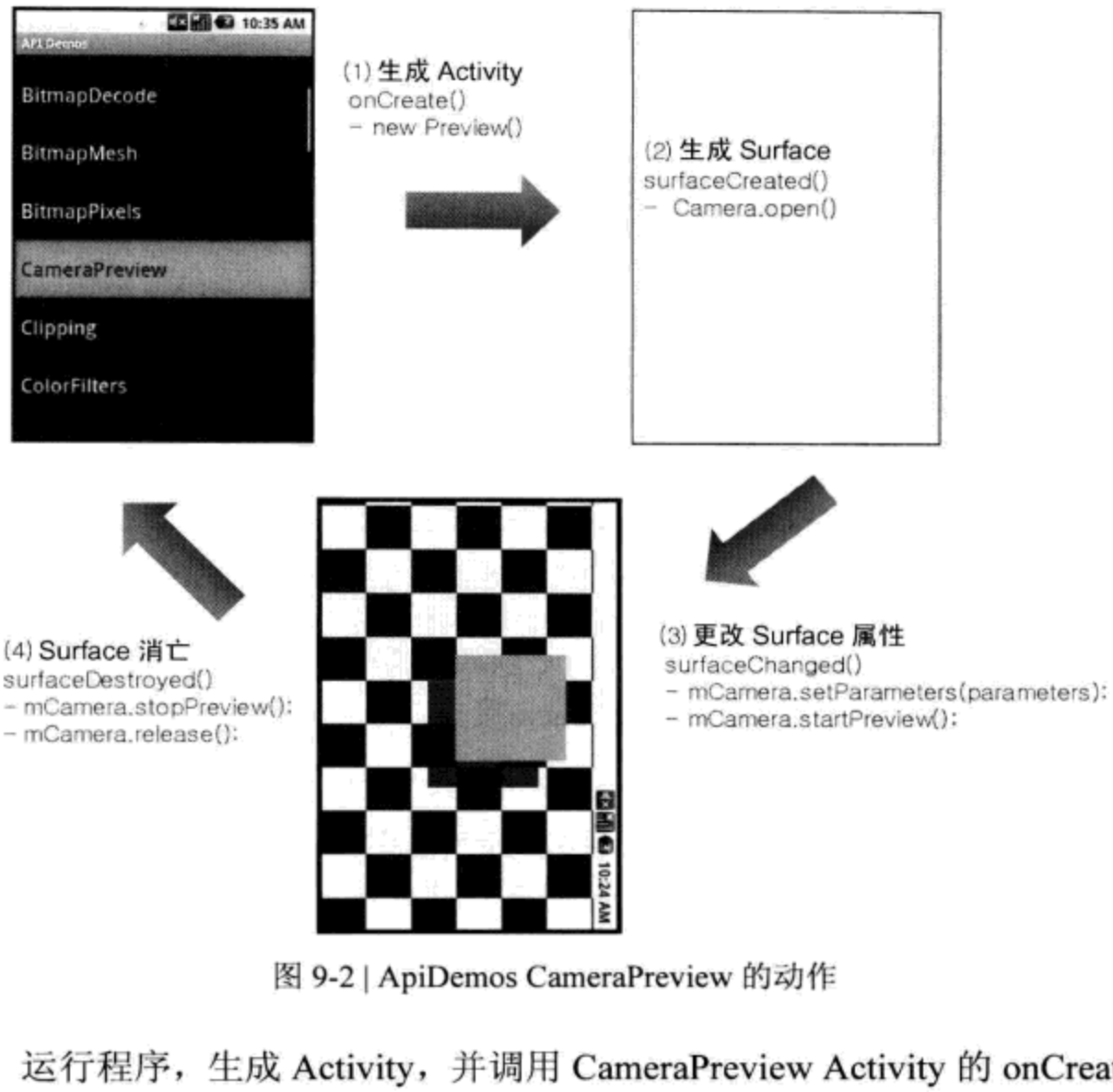


图 9-2 | ApiDemos CameraPreview 的动作

(1) 运行程序，生成 Activity，并调用 CameraPreview Activity 的 onCreate()方法，如代码 9-2 所示。

```
public class CameraPreview extends Activity {
   private Preview mPreview;
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      mPreview = new Preview(this);  ❶
      setContentView(mPreview);
   }
}
```

代码 9-2 |CameraPreview 的 onCreate()方法

❶ 为配合相机服务，该行语句生成 Preview 类的实例对象，用来负责在显示屏上执行绘画操作。

更多更新编程资源尽在 http://www.pin5i.com/

在代码 9-3 中，可以看到 Preview 类的构造方法（构造函数）的部分代码。从代码中可知，Preview 类继承了 SurfaceView 父类，并实现了 SurfaceHolder.Callback 接口。SurfaceView 在系统视图（View）层次结构中，为应用程序提供了专用的绘图空间（Surface）。在专用绘图空间中实施的所有绘图作业都在后台被处理，应用程序不必等到 Surface 生成或绘图作业完成的时候。

```
class Preview extends SurfaceView implements SurfaceHolder.Callback {
   Preview(Context context) {
      super(context);
      mHolder = getHolder();                    ❷
      mHolder.addCallback(this);                ❸
      mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
    ...
}
```

代码 9-3 | Preview 的构造方法

由于应用程序不能直接访问 Surface，必须通过 SurfaceHolder 来处理 Surface，所以 Preview 同时实现了 SurfaceHolder.Callback 接口。

如代码 9-3 所示，在调用构造方法生成 SurfaceView 后，❷继续调用 getHolder()方法，获取 SurfaceHolder 实例对象，而后❸调用 addCallback()方法，连接 Surface 与回调方法。当创建或销毁 Surface，或者更改属性时，就会调用 SurfaceHolder.Callback 接口方法，最终调用实现该接口的 Preview 类的方法。

(2) Surface 对象生成后，紧接着调用 surfaceCreated()方法，该方法的源码如代码 9-4 所示。

```
public void surfaceCreated(SurfaceHolder holder) {
   mCamera = Camera.open();    ❹
}
```

代码 9-4 | surfaceCreated()方法

❹ 获取 Camera 实例对象。surfaceCreated()方法不会传递有关 Surface 的信息（画面大小信息）。关于相机的属性，将在调用 surfaceChanged()方法时进行设置。

(3) 当 surface 属性发生改变时，将会调用 surfaceChanged()方法，并且画面的尺寸信息将作为参数传入其中。surfaceChanged()方法源码如代码 9-5 所示。

更多更新编程资源尽在 http://www.pin5i.com/


```
public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
    Camera.Parameters parameters = mCamera.getParameters();
    List<Size> sizes = parameters.getSupportedPreviewSizes();
    Size optimalSize = getOptimalPreviewSize(sizes, w, h);
    parameters.setPreviewSize(optimalSize.width, optimalSize.height);
    mCamera.setParameters(parameters);     ❺
    mCamera.startPreview();     ❻
}
```

代码 9-5 | surfaceChanged()方法

❺ 调用 Camera 的 setParameters()方法，设置预览画面尺寸，而后再调用❻startPreview()方法，执行预览操作。

(4) 当应用程序不再需要 Surface 时，就会调用 surfaceDestroyed()方法销毁它，如代码 9-6 所示。

```
public void surfaceDestroyed(SurfaceHolder holder) {
    mCamera.stopPreview();  ❼
    mCamera.release();  ❽
    mCamera = null;
}
```

代码 9-6 | surfaceDestroyed()方法

❼ 调用 stopPreview()方法，终止预览，❽调用 release()方法，取消与相机服务的连接。

CameraPreview 示例程序比较简单，它通过应用程序框架提供的 android.hardware.Camera 来控制驱动相机设备。通过该示例的学习，我们了解了框架内部的运行机制，明白了应用程序请求是如何处理的。在下一节中，将学习相机服务框架的相关知识，掌握这些基础的知识是十分必要的，希望大家认真学习。

9.3 相机服务框架（Camera Service Framework）

9.3.1 相机服务框架层次结构

相机服务框架是由哪些部分构成的，各部分间是如何联系的呢？请看图 9-3，它是相机服务框架的层次结构图。

(1) 如前所述，应用程序通过 Android Framework 提供的 android.hardware.Camera 类来访问相机服务。

更多更新编程资源尽在 http://www.pin5i.com/

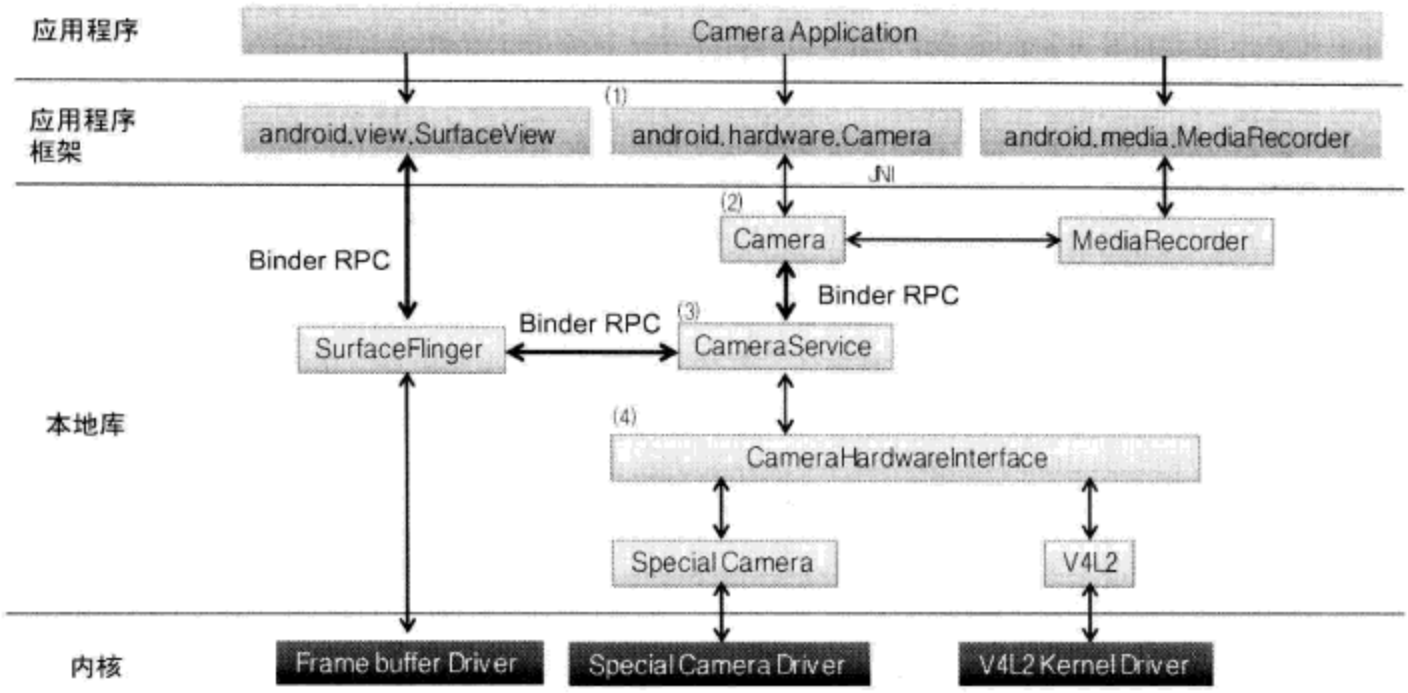


图 9-3 | 相机服务框架层次结构

(2) android.hardware.Camera 中的本地方法通过 JNI 调用本地库中 Camera 的成员函数。

(3) 为处理应用程序请求，Camera 通过 Binder RPC 与相机服务进行交互。在 Camera 连接（connect）的过程中，服务代理（BpCameraService）与服务（CameraService）之间是如何进行 Binder RPC 操作的呢？请参看图 9-4，它描述了这一过程。关于相机设备设置与请求预览的内容，将在本章后半部分进行讲解说明。

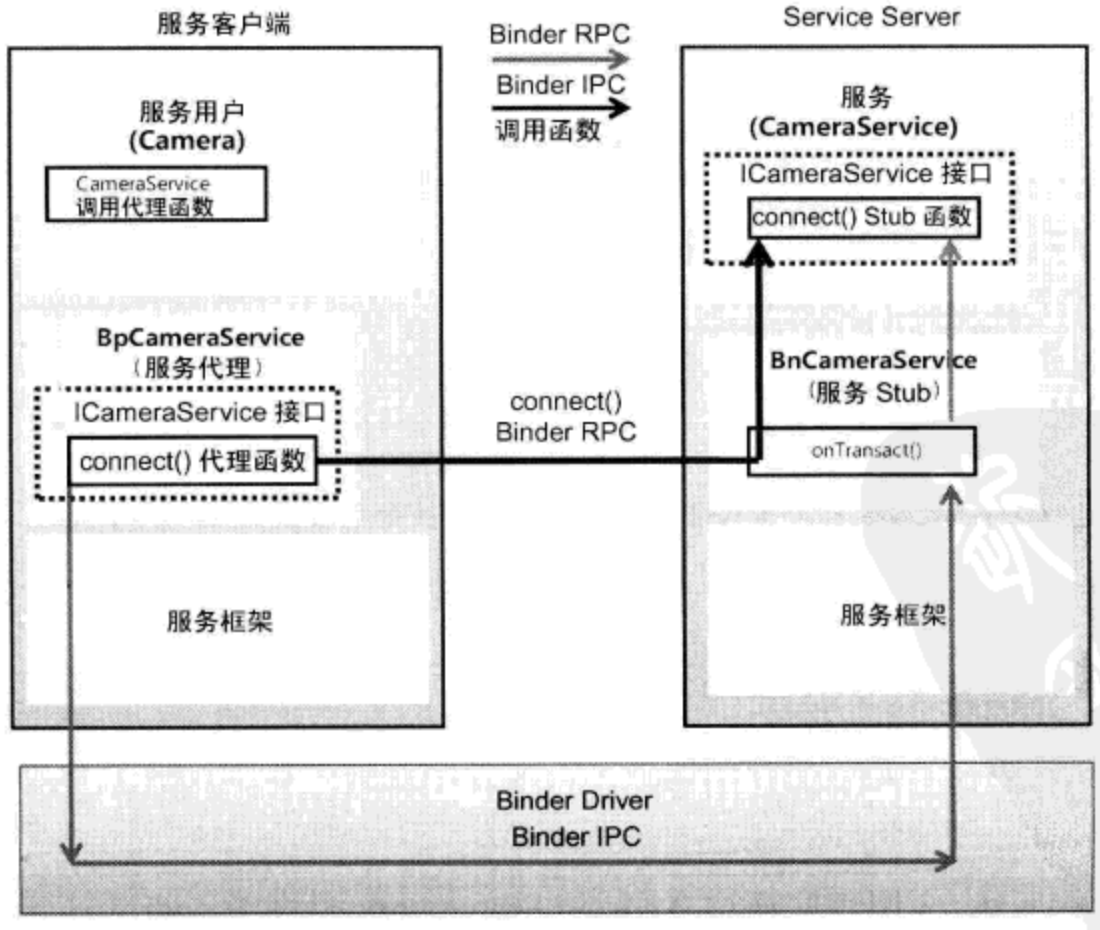


图 9-4 | 通过 Binder RPC 连接相机服务

更多更新编程资源尽在 http://www.pin5i.com/

(4) CameraHardwareInterface 是抽象类，它定义了一系列接口，允许 Service Framework 访问相机设备。每个 Device Vendor 都继承并实现了 Camera HardwareInterface 抽象类，支持相机设备功能，与 Service Framework 进行连接。

9.3.2 相机服务框架类

相机服务是如何运作的呢？若想弄清这个问题，必须先了解组成 Service Framework 的各个类与 Binder RPC 的连接关系。图 9-5 描述了在不同的三个部分中各个类与 Binder RPC 的关系。

(a) Camera 类继承自 ICameraClient 类，负责在应用程序与相机服务间传递 Binder RPC 数据。

(b) CameraService 类继承了 ICameraService 类，负责应用程序与相机服务间的连接。

(c) CameraService::Client 类继承自 ICamera 类，负责相机设备的设置、控制，以及处理来自相机设备的事件。

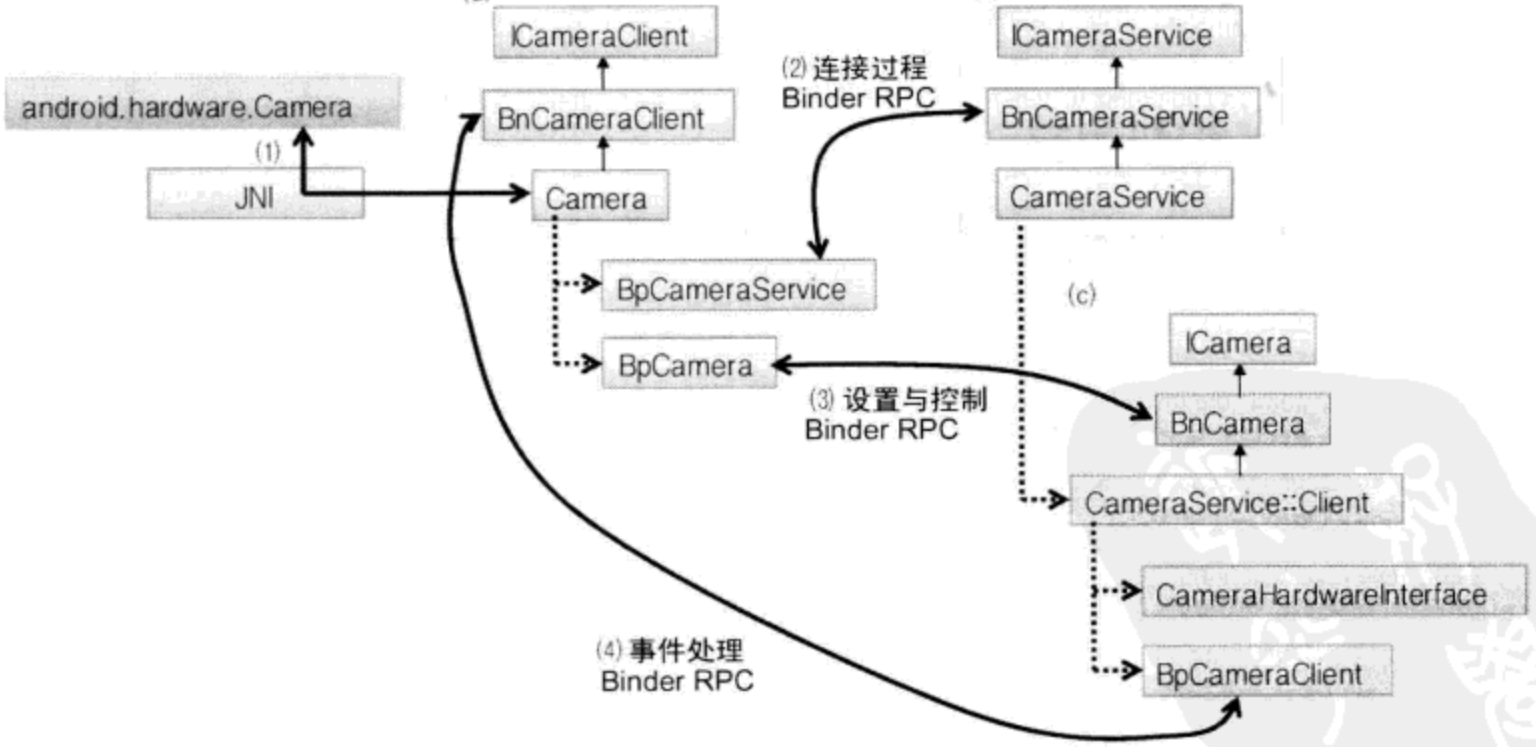


图 9-5 | 类与 Binder RPC 的关系

下面看一下各个类与 Binder RPC 的具体关系。

(1) android.hardware.Camera 中的本地方法通过 JNI 调用本地库中 Camera 的成员函数。

(2) 在应用程序连接相机服务时，Camera 通过 BpCameraService（服务代理）与

更多更新编程资源尽在 http://www.pin5i.com/

BnCamera Service（服务 stub）进行 Binder RPC 操作（经由 ICameraService 接口执行 Binder RPC 交互）。

(3) 应用程序在请求相机设备设置或预览功能时，Camera 通过 BpCamera（服务代理）与 BnCamera（服务 stub）进行 Binder RPC 操作（经由 ICamera 接口执行 Binder RPC 交互）。

(4) 当相机设备发生事件时，CameraService::Client 通过 BpCameraClient（服务代理）与 BnCameraClient（服务 stub）进行 Binder RPC 操作（经由 ICameraClient 接口执行 Binder RPC 交互）。

为了连接应用程序与 Binder RPC，CameraService 向上下文管理者（Context Manager）注册服务。注意，虽然 Camera 与 CameraService::Client 是以服务形式运作的，但它们并不向 Context Manager 服务。

应用程序在连接相机服务时，将在 Camera 与 CameraService::Client 之间生成额外的两个 Binder RPC 结构，有关内容在 9.4.3 一节中详细讲解。

9.4 相机服务框架的运行

9.4.1 初始化相机服务

相机服务在 Media Server 中初始化。代码 9-7 是 Media Server[1]的 main()函数，在该函数中相机服务被初始化。

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ...
    CameraService::instantiate();    ❶
    ...
}
```

代码 9-7 | Media Server 的 main()函数[2]

❶ 调用 CameraService 的 instantiate()成员函数，执行 CameraService 初始化。

1 Media Server 是用来初始化大部分本地系统服务的 Service Server。

2 frameworks/base/media/mediaserver/main_mediaserver.cpp

更多更新编程资源尽在 http://www.pin5i.com/

CameraService 的 instantiate()成员函数用来创建 CameraService 实例对象，并将其注册到默认服务管理器中，函数代码如代码 9-8 所示。

```
void CameraService::instantiate() {
    defaultServiceManager()->addService(
        ➥ String16("media.camera"), new CameraService());      ←❷
}
```

代码 9-8 | CameraService 的 instantiate()成员函数[1]

❷ 创建 CameraService 实例对象，并将其注册到 Context Manager 中。相机服务的注册过程与其他服务的注册过程是一样的，具体内容，请参考 8.4 “Native Service Manager”一节的相关内容。

经过上述一系列过程之后，相机服务已准备好执行用户的请求。那么，相机用户在使用相机服务时都经历哪些程序步骤，相机服务是如何处理并回应服务用户的请求的呢？下面将继续探讨这些问题，找到问题的答案，消除大家心中的疑惑。

9.4.2 连接相机服务

在使用相机服务之前，应用程序必须先与相机服务连接起来，在这一过程中，ICameraService 的 Binder RPC 扮演连接桥梁的角色。在完成连接后，相应的 Binder RPC 关系就确立了，利用它可以设置相机设备、传递相应命令、接收相应的事件。请参看图 9-6，它描述了连接相机服务的过程。

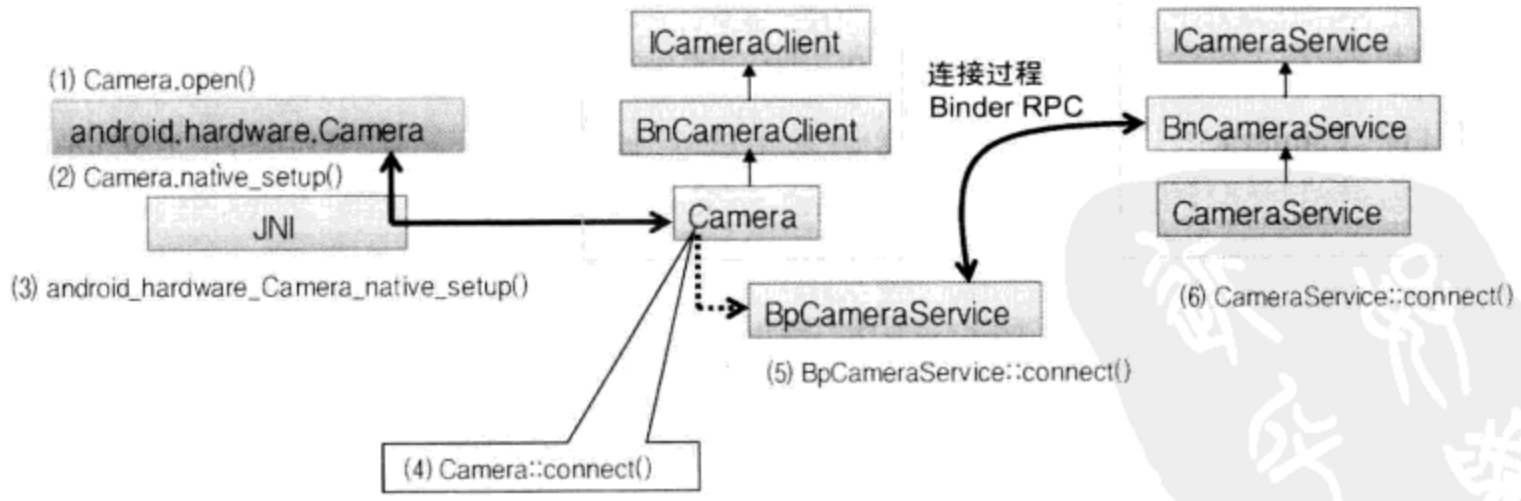


图 9-6 | 连接相机服务

(1) 应用程序调用 android.hardware.Camera 的 open()方法。

1 frameworks/base/camera/libcameraservice/CameraService.cpp

更多更新编程资源尽在 http://www.pin5i.com/

(2) open()方法调用 native_setup()本地方法。

(3) native_setup()方法通过 JNI 调用 android_hardware_Camera_native_setup()函数。

(4) android_hardware_Camera_native_setup()函数调用 Camera 的 connect()成员函数。

(5) Camera 的 connect()成员函数从 Context Manager 获取相机服务信息，生成服务代理（BpCameraService）后，通过 Binder RPC 连接到相机服务 stub（BnCameraService）。

(6) 实际连接是由 CameraService 的 connect()成员函数进行处理的。

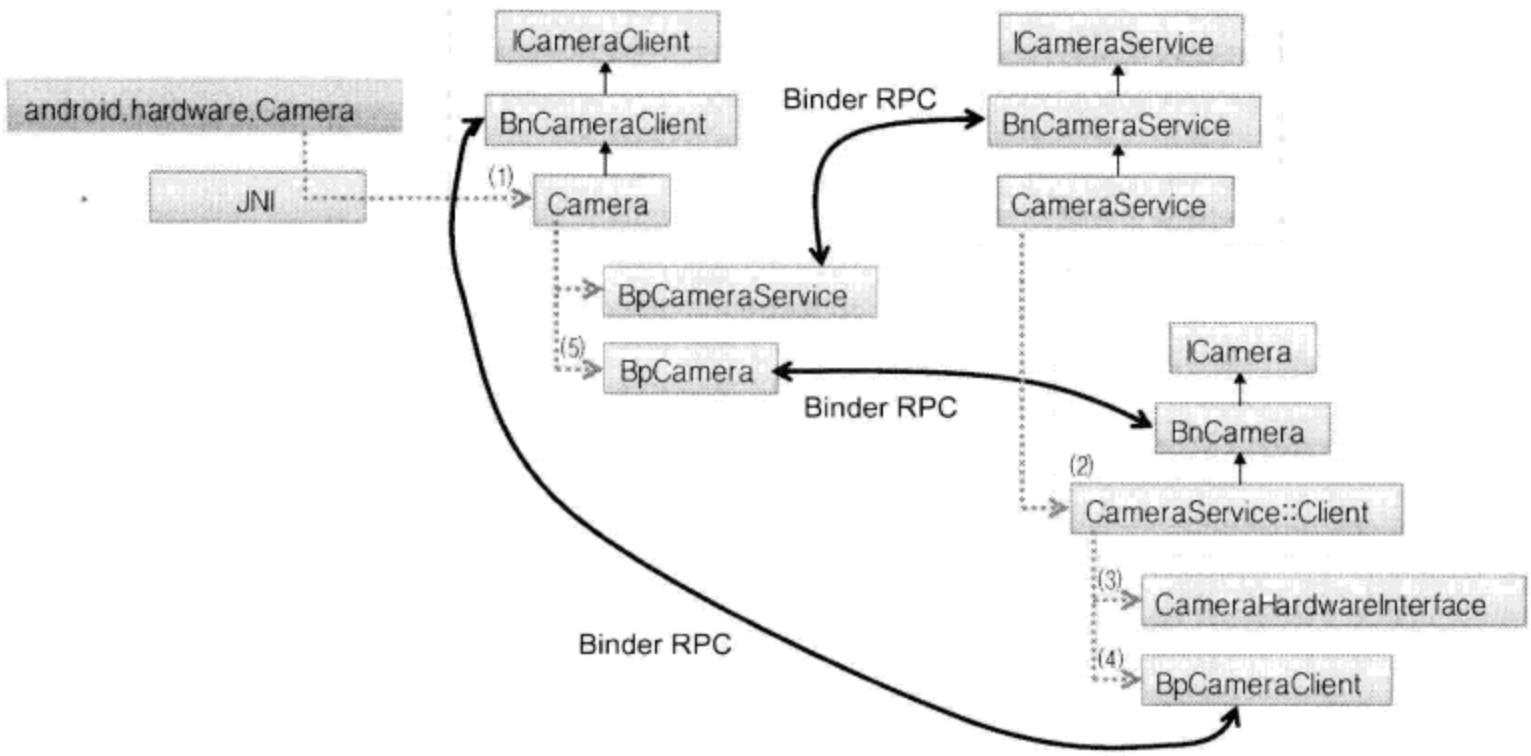


图 9-7 | 完成相机服务连接

相机服务连接完成后，形成如图 9-7 所示的结构。

(1) Camera 根据应用程序连接请求生成实例对象，而后实例对象在 BpCameraService 中生成用于请求 CameraService 连接的 Binder RPC 数据，并将其转换为 Binder 对象[1]一同传递。

(2) 以来自 Camera 的 Binder RPC 数据为基础，CameraService 创建 Camera 的服务代理 BpCameraClient 的实例，而后再创建 CameraService::Client 实例对象，用来控制相机设备、处理事件的接收。

(3) CameraService::Client 创建 CameraHardwareInterface 实例对象，初始化相机设备。

---

1 本章中的 Binder 对象都是指 flat_binder_object 数据结构，关于 flat_binder_object 数据结构存储的内容，请参考 8.4“Native Service Manager”一节的相关内容。

更多更新编程资源尽在 http://www.pan5i.com/

(4) 将由 CameraService 传递而来的 BpCameraClient 实例对象保存到 CameraService::Client，以便传递相机设备中发生的事件。

(5) 处理完连接请求后，会生成 CameraService::Client 实例的 Binder 对象，CameraService 将其保存到 Binder RPC 数据中，并进行应答。而后生成并保存 CameraService::Client 的代理对象 BpCamera，作为 Camera 返回的 Binder 对象。

当连接过程完成后，即可通过 BpCamera 控制或设置相机设备，使用 BpCameraClient 传递相机设备事件。

9.4.3 相机服务连接过程分析

下面通过源代码来学习相机服务的连接是如何处理的。对于 JNI 部分，由于它只是单纯地调用 Camera 类的成员函数，在此省略讨论。

1) 首先看一下 Camera 的 connect()函数代码，它用来传递 Binder RPC 数据。

```
sp<Camera> Camera::connect()
{
    sp<Camera> c = new Camera();
    const sp<ICameraService>& cs = getCameraService();
    if (cs != 0) {
        c->mCamera = cs->connect(c);  ←❶
    }
    return c;
}
```

代码 9-9 | Camera 的 connect()函数[1]

Camera 的 connect()成员函数在应用程序调用 JNI 方法中被调用，其代码如代码 9-9 所示。

❶ 调用 getCameraService()函数，先从 Context Manager 获取相机服务的服务 Handle，而后生成 BpCameraService 实例对象，并将其返回。注意，在调用 BpCameraService 的 connect()成员函数时，前面创建的 Camera 实例将被传入到函数中。

下面分析 BpCameraService 的 connect()成员函数，其代码如下所示：

```
class BpCameraService: public BpInterface<ICameraService>
{
```

1 frameworks/base/libs/camera/Camera.cpp

更多更新编程资源尽在 http://www.pin5i.com/

```
    virtual sp<ICamera> connect(const sp<ICameraClient>& cameraClient)
    {
        Parcel data, reply;
        data.writeInterfaceToken(ICameraService::getInterfaceDescriptor());
        data.writeStrongBinder(cameraClient->asBinder());                    ❷
        remote()->transact(BnCameraService::CONNECT, data, &reply);
        return interface_cast<ICamera>(reply.readStrongBinder());            ❸
    }
}
```

代码 9-10 | BpCameraService 的 connect()成员函数[1]

❷ 将参数传递过来的 Camera 实例转换为 Binder 对象后，保存到 Binder RPC 数据中，以便传递。在保存 Binder 对象时，Binder Driver 将为 Camera 分配新的 Binder 节点，且 Camera 以服务的形式运行。并且，在 Binder RPC 数据的接收端将使用 Camera 的 Binder 对象为 Camera 生成服务代理 BpCameraClient 实例对象。

❸ CameraService 用来处理连接，在用于应答的 Binder RPC 数据中，包含着 CameraService::Client 的 Binder 对象，并以此生成 CameraService::Client 的服务代理 BpCamera 实例对象，生成的对象实例存储在 Camera 的 mCamera 成员变量中，如代码 9-9❶所示。

2) CameraService 处理接收到的 Binder RPC 数据，并进行应答，源代码如下所示：

```
status_t BnCameraService::onTransact(
    ➥ uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CONNECT: {
            CHECK_INTERFACE(ICameraService, data, reply);
            sp<ICameraClient> cameraClient = interface_cast<ICameraClient>(
                ➥ data.readStrongBinder());                                 ❹
            sp<ICamera> camera = connect(cameraClient);                      ❺
            reply->writeStrongBinder(camera->asBinder());                    ❻
            return NO_ERROR;
        } break;
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

代码 9-11 | BnCameraService 的 onTransact()函数[2]

1 frameworks/base/libs/camera/Camera.cpp

2 frameworks/base/libs/camera/ICameraService.cpp

更多更新编程资源尽在 http://www.pin5i.com/

❹ 在相机服务接收到的 Binder RPC 数据中包含着 Camera 的 Binder 对象。

调用 Parcel 的 readStrongBinder()与 interface_cast<ICameraClient>函数，为 Camera 生成服务代理 BpCameraClient 实例对象。

❺ 调用 CameraService 的 connect()函数，并通过参数传入 BpCameraClient 实例对象。

❻ 在 CameraService 的 connect()函数中，创建 CameraService::Client 对象，并将其返回。调用 Parcel 的 writeStrongBinder()函数，将 connect()函数返回的对象以 Binder 对象的形式保存到 Binder RPC 数据中。在该过程中，CameraService::Client 对象将被分配一个 Binder 节点，且以服务的形式运行。

实际上，具体的连接是由 CameraService 的 connect()函数来负责的，connect()函数的主要代码如下：

```
sp<ICamera> CameraService::connect(const sp<ICameraClient>& cameraClient)
{
    int callingPid = getCallingPid();
    client = new Client(this, cameraClient, callingPid);        ←❼
    mClient = client;
    return client;
}
```

代码 9-12 | CameraService 的 connect()函数[1]

❼ 在 CameraService 的 connect()函数中，使用 New 运算符，创建 CameraService::Client 对象，并将 this 指针与 BpCameraClient(cameraClient)作为参数传入函数中。

3) 调用 CameraService::Client 的 Client()构造函数，创建 CameraService::Client 实例对象，构造函数代码如下：

```
CameraService::Client::Client(const sp<CameraService>& cameraService,
const sp<ICameraClient>& cameraClient, pid_t clientPid)
{
    mCameraService = cameraService;
    mCameraClient = cameraClient;          ←❽
    mHardware = openCameraHardware();      ←❾
}
```

代码 9-13 | CameraService::Client 的 Client()构造函数[2]

❽ 将 Camera 的服务用户类 BpCameraClient 的实例对象保存到成员变量 mCamera

1 frameworks/base/camera/libcameraservice/CameraService.cpp

2 frameworks/base/camera/libcameraservice/CameraService.cpp

更多更新编程资源尽在 http://www.pin5i.com/

Client 之中。

❾ 调用 openCameraHardware()函数，初始化相机设备。

在连接操作中将生成新的 Camera 与 CameraService::Client 对象，形成如图 9-8 所示的结构。

新生成的 Camera 与 CameraService::Client 以服务的形式运作，

(1) Camera 生成 CameraService::Client 服务代理 BpCamera 实例对象。

(2) CameraService::Client 生成 Camera 的服务代理 BpCameraClient 实例对象。

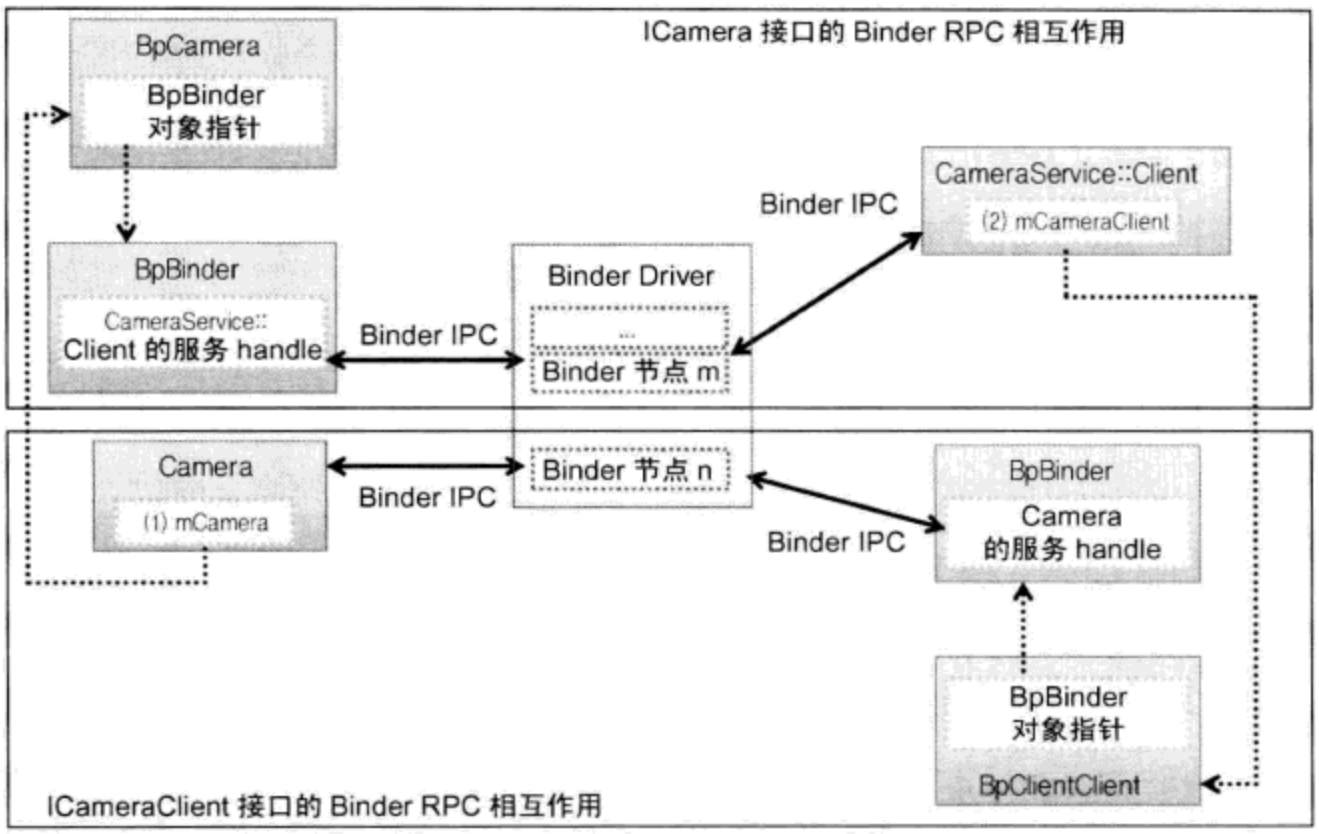


图 9-8 | 各个实例对象与 Binder RPC 的关系

经过这一系列过程，在 Camera 与 CameraService::Client 之间形成了服务或用户的相互连接结构。

9.4.4 相机设置与控制

如前所言，应用程序使用 ICamera 的 Binder RPC 请求更改相机设置或提请预览。在应用程序中，调用 setParameters()方法，将相应的设置参数以 Binder RPC 的形式传递给相机设备，整个过程如图 9-9 所示。

(1) 应用程序调用 android.hardware.Camera 的 setParameters()方法，变更相机设置。

(2) setParameters()是本地方法，调用该方法，将通过 JNI 调用 android_hardware_Camera_ setParameters()函数。

更多更新编程资源尽在 http://www.qin5i.com/

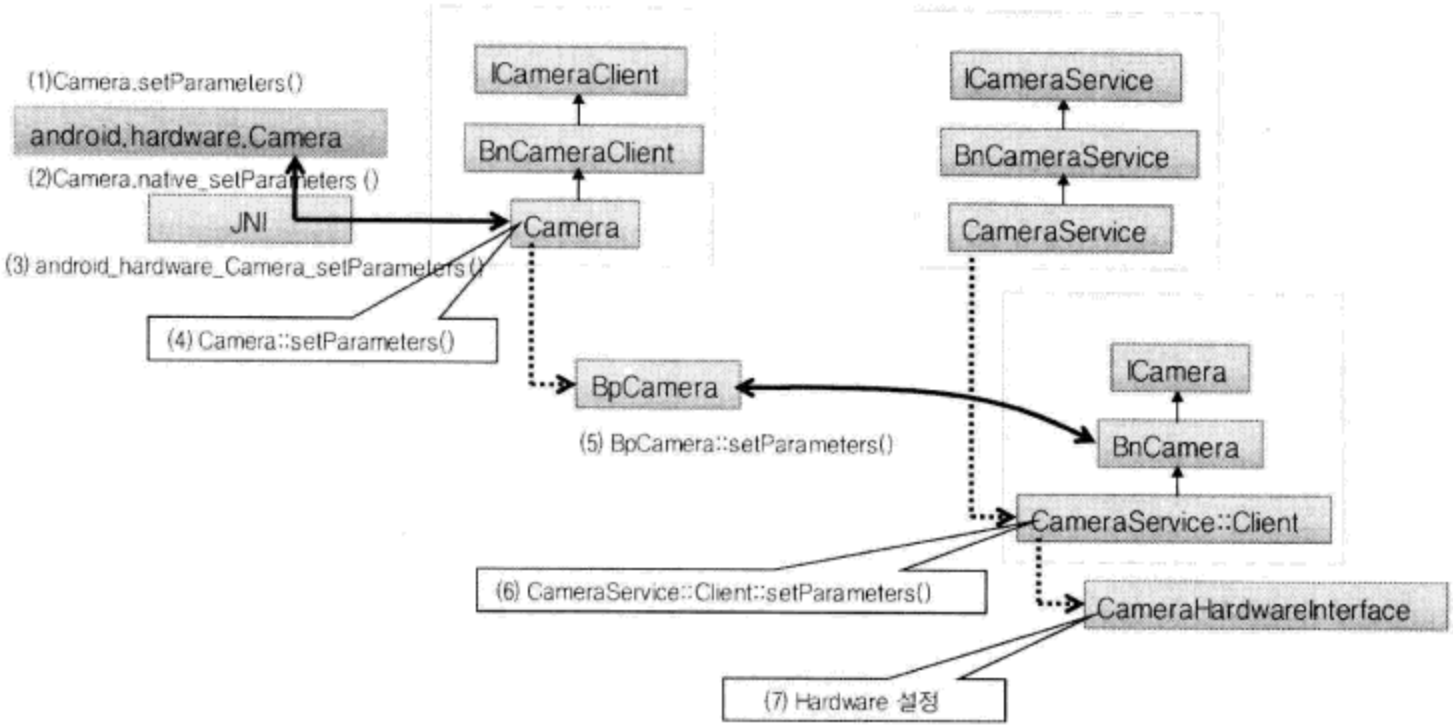


图 9-9 | 调用相机服务的 setParameters()方法

(3) 在 android_hardware_Camera_setParameters()函数中调用 Camera 的 setParameters()成员函数。

(4)～(5) Camera 的 setParameters()成员函数通过 BpCamera 以 Binder RPC 的形式向 BnCamera 请求变更相机设置。

(6) 在 CameraService::Client 的 setParameters()成员函数中，调用 CameraHardware Interface 的 setParameters()成员函数。

(7) CameraHardwareInterface 将设置更改应用到相机设备上。

9.4.5 相机设置与控制分析

下面来分析源代码，进一步了解设置相机设备的过程。首先来分析 Camera 类的 setParameters()成员函数的源代码，如下所示：

```
status_t Camera::setParameters(const String8& params)
{
   sp <ICamera> c = mCamera;
   return c->setParameters(params);          ←❶
}
```

代码 9-14 | Camera 的 setParameters()成员函数[1]

❶ mCamera 保存着 BpCamera 的实例对象。该语句调用 BpCamera 的 setParameters()

1 frameworks/base/libs/camera/Camera.cpp

更多更新编程资源尽在 http://www.pin5i.com/

函数，通过 Binder RPC 调用 CameraService::Client 的 setParameters()函数。

接着来起分析 CameraService::Client 的 setParameters()函数，其主要代码如下：

```
status_t CameraService::Client::setParameters(const String8& params)
{
    CameraParameters p(params);
    return mHardware->setParameters(p);          ❷
}
```

代码 9-15 | CameraService::Client 的 setParameters()函数[1]

❷ mHardware 存储着 CameraHardwareInterface 实例对象，该语句通过 mHardware 将设置应用到相机设备中。

9.4.6 相机事件处理

当相机设备发生事件时，相机服务使用 ICameraClient 的 Binder RPC，将其传递给应用程序。例如，应用程序在调用 takePicture()函数获取相机静态图像时，相机设备准备好静态图像后，将以异步的方式将静态图像已备好的信息通知给应用程序。在这一过程中，首先发生 Shutter 事件，随后分别发生与 RAW 图像和 JPEG 图像相关的事件。

CameraService::Client 通过 BpCameraClient 使用 Binder RPC 将事件传递给 Camera，最后再经过 JNI 传递给应用程序。

Shutter 事件发生后，事件处理过程如图 9-10 所示。

(1) 相机设备准备好静态图像后，首先通过 CameraHardwareInterface 触发 Shutter 事件。

(2) 初始化 CameraHardwareInterface 时，CameraService::Client 调用已经注册过的回调函数，通过 CameraService::Client 的 handleShutter()成员函数处理 Shutter 事件。CameraService::Client 的 handleShutter()成员函数通过 BpCameraClient 使用 Binder RPC 向 BnCamera Client 请求处理 Shutter 事件。

(3) 由 Binder RPC 传递而来的事件实际由 Camera 的 notifyCallback()函数负责处理。

(4) Camera 的 notifyCallback()函数通过 JNI 接口向应用程序传递 Shutter 事件。

(5) 应用程序调用预定义好的回调函数，处理 Shutter 事件。

1 frameworks/base/camera/libcameraservice/CameraService.cpp

更多更新编程资源尽在 http://www.pin5i.com/

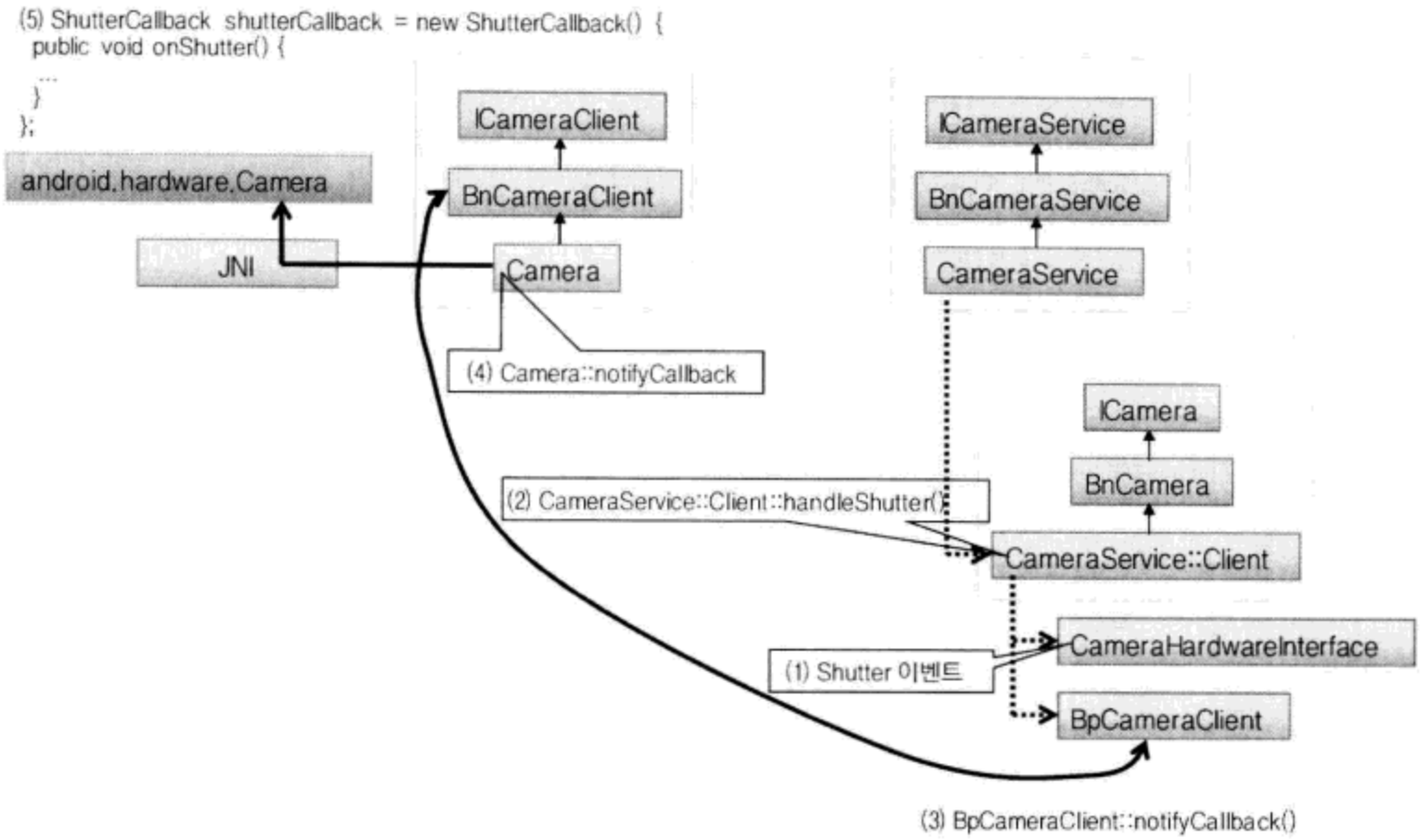


图 9-10 | 处理相机服务的 Shutter 事件

9.4.7　相机事件处理分析

下面我们来分析源码，进一步了解相机服务框架是如何处理事件的。

首先应用程序调用 takePicture()方法，请求获取静态图像，takePicture()方法代码如下：

```
status_t CameraService::Client::takePicture()
{
    mHardware->enableMsgType(CAMERA_MSG_SHUTTER |
        CAMERA_MSG_POSTVIEW_FRAME |
        CAMERA_MSG_RAW_IMAGE |
        CAMERA_MSG_COMPRESSED_IMAGE);          ❶
    return mHardware->takePicture();           ❷
}
```

代码 9-16 | CameraService::Client 的 takePicture()函数[1]

当应用程序调用 android.hardware.Camera 的 takePicture()方法请求静态图像时，CameraService::Client 的 takePicture()函数就会被调用执行，如同在相机设置与控制中讲解的一样。

1 frameworks/base/camera/libcameraservice/CameraService.cpp

更多更新编程资源尽在 http://www.pin5i.com/

❶ 首先在 CameraService::Client 的 takePicture()中启用相应的消息类型，当相机设备中发生 CAMERA_MSG_SHUTTER、CAMERA_MSG_RAW_IMAGE、CAMERA_MSG_ COMPRESSED_IMAGE 等事件时进行捕获。然后❷调用静态图像处理函数。下面是 CameraService::Client 的 handleShutter()函数的主要代码，如代码 9-17 所示。

```
void CameraService::Client::handleShutter( image_rect_type *size )
{
    sp<ICameraClient> c = mCameraClient;
    if (c != NULL) {
        c->notifyCallback(CAMERA_MSG_SHUTTER, 0, 0);        ←❸
    }
    mHardware->disableMsgType(CAMERA_MSG_SHUTTER);          ←❹
}
```

代码 9-17 | CameraService::Client 的 handleShutter()函数

相机设备准备好静态图像后，首先发送 CAMERA_MSG_SHUTTER 事件，该事件由 CameraService::Client 的 handleShutter()成员函数进行处理。

❸ mCameraClient 中保存着 BpCameraClient 的实例对象，通过 Binder RPC 调用 Camera 的 notifyCallback()函数处理发生的事件。

❹ 调用 disableMsgType()函数，阻止 CAMERA_MSG_SHUTTER 事件发生。

Camera 的 notifyCallback()函数用来接收并处理 Shutter 事件，该函数代码如下：

```
void Camera::notifyCallback(int32_t msgType, int32_t ext1, int32_t ext2)
{
    sp<CameraListener> listener;
    {
        listener = mListener;
    }
    if (listener != NULL) {
        listener->notify(msgType, ext1, ext2);          ←❺
    }
}
```

代码 9-18 | Camera 的 notifyCallback()函数

❺ 在 Camera 的 notifyCallback()函数中，调用 CameraListener 的 notify()函数，将以 Binder RPC 传递过来的事件转发给应用程序。在变量 mListener 中保存着 JNICameraContext 的实例对象，它通过 JNI 在应用程序与 Service Framework 间传递数据。

更多更新编程资源尽在 http://www.pin5i.com/

9.5　小结

本章介绍了相机服务框架，以及应用程序使用相机服务的方法，内容梳理如下。

1. 创建 Surface 同时连接相机服务
2. 更改 Surface 属性，设置相机设备，启动预览
3. 销毁 Surface 对象，终止预览，解除与相机服务的连接。

在介绍相机服务框架的层次结构与行为时，分成以下三个方面进行讲解。

1. 使用 ICameraService 接口的 Binder RPC，连接相机服务
2. 使用 ICamera 接口的 Binder RPC，设置与控制相机设备
3. 使用 ICameraClient 接口的 Binder RPC，处理相机设备事件

更多更新编程资源尽在 http://www.pin5i.com/

更多更新编程资源尽在 http://www.pin5i.com/

第 10 章

Java 服务框架（Java Service Framework）

Android 服务框架（Android Service Framework）分为 Java 服务框架（Java Service Framework）与本地服务框架（Native Service Framework）两部分。其中，本地服务框架（Native Service Framework）在第 8 章已经学习过，并且还直接编写了一个本地系统服务的示例，相信读者都已经掌握了。从本章开始，将继续学习 Java 服务框架（Java Service Framework）的相关知识。与本地服务框架一样，Java 服务框架也提供 4 种核心功能，只是系统内部服务运作机制以及服务编写的方法不同而已。

10.1 Java 服务框架（Java Service Framework）

Java 服务框架是一系列类的集合，这些类用于支持开发 Java 系统服务，这些服务运行在基于 Java 的应用程序框架中。如上所言，Android 服务框架由 Java 服务框架（Java 层）与本地服务框架（C++层）两部分组成，这两个层通过 JNI 进行交互，如图 10-1 所示。在 Java 中使用 JNI 目的就是为了重用一些由 C/C++编写的代码。同样，Java 服务框架通过 JNI 可以使用本地服务框架，这样 Java 层的服务用户不仅可以使用由 Java 语言编写的服务，还可以使用 C++语言编写的本地服务。

Java 服务框架与本地服务框架的不同有如下两点。

1. 服务生成：在 Java 服务框架中，开发 Java 服务的方法有两种。第一种是继承 Binder 类进行开发的方式，这种方式常常用于开发可以实现精确控制的服务中，开发 Java 系统服务时也采用这种方式。在 Android 开发者工具包中，为开发者提供了 AIDL 语言（用来自动生成服务接口、服务代理以及服务 stub 代码）以及编译器，这使得编写 Java 系统服务比编写本地系统服务要容易得多。

更多更新编程资源尽在 http://www.pin5i.com/

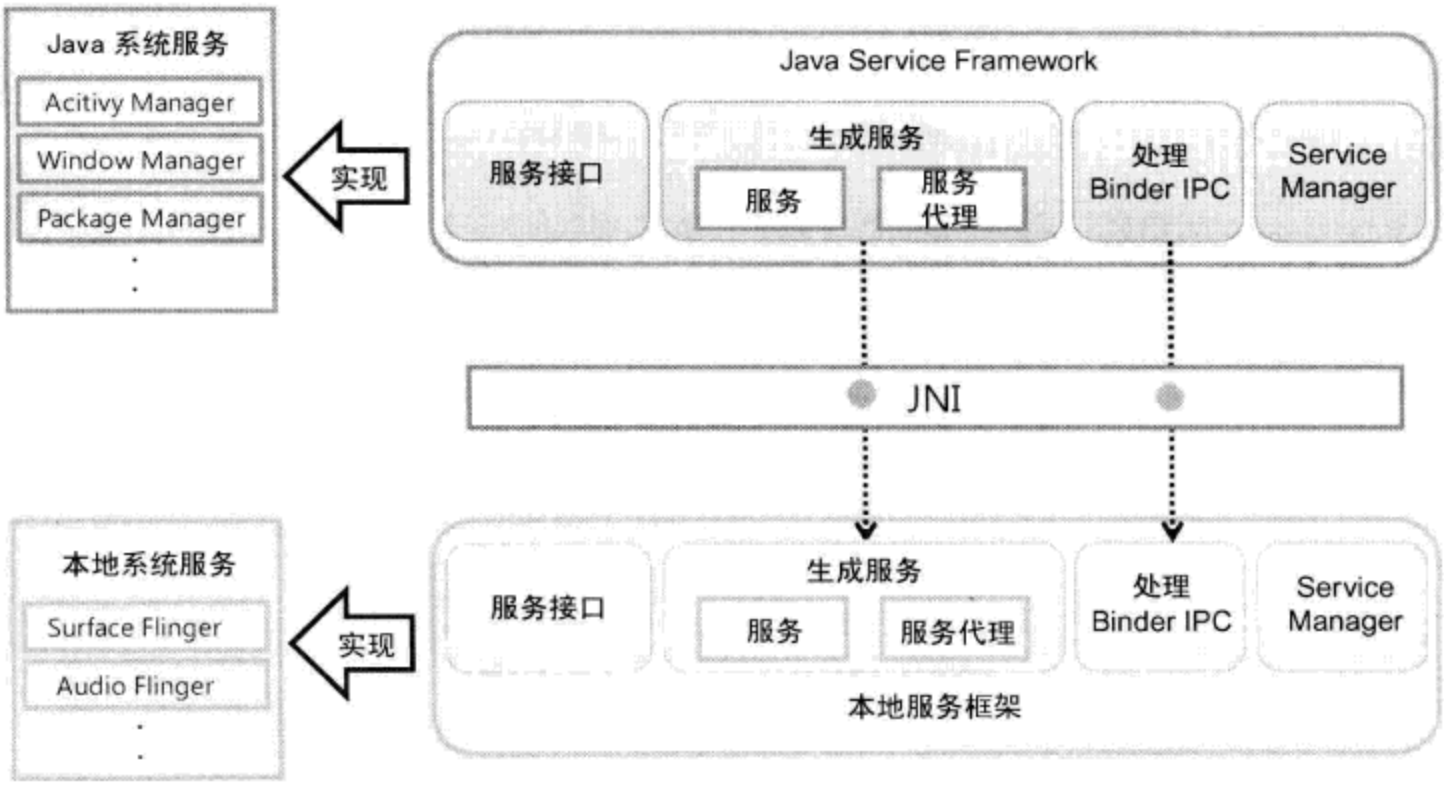


图 10-1 | Java 服务框架（Java 层）与本地服务框架（C++层）

第二种方法是继承 Service 类进行开发，常常用来开发一些周期执行某些任务的后台服务。由于采用该方法创建服务的内容已经超出本书的讨论范围，感兴趣的读者朋友，请参考 Android 开发者文档中的相关内容。

2. Binder IPC 处理：为了支持 Binder IPC，Java 服务框架将通过 JNI 使用本地服务框架中的相应组成部分。

> **TIP Java 服务管理**
>
> 在 Java 层中使用两种方法管理服务，第一种与本地系统服务一样，Java 系统服务将服务注册到 Context Manager 中，而后通过 Service Manager 使用服务；第二种，Java 应用程序服务由 Activity Manager Service 而非 Context Manager 进行管理，Activity Manager Service 是一种核心 Java 系统服务，在第 11 章中已经作过说明。

10.1.1 Java 服务框架的层次结构

如图 10-2 所示，Java 服务框架由服务层、RPC 层、IPC 层三个层构成。在第 8 章“Android Service Framework”中，我们曾举了一个名称为 FooService 的服务示例。假如 FooService 服务是使用 Java 服务框架实现的。那么，在每个层上与本地服务框架有 3 个不同点。第一，管理者类（Manager）位于服务用户的服务层；第二，由 AIDL 自动生成的 Stub 与代理（Proxy）类位于 RPC 层；第三，IPC 层中的组成部分通过 JNI 连接到本地服务框架中。

更多更新编程资源尽在 http://www.pin5i.com/

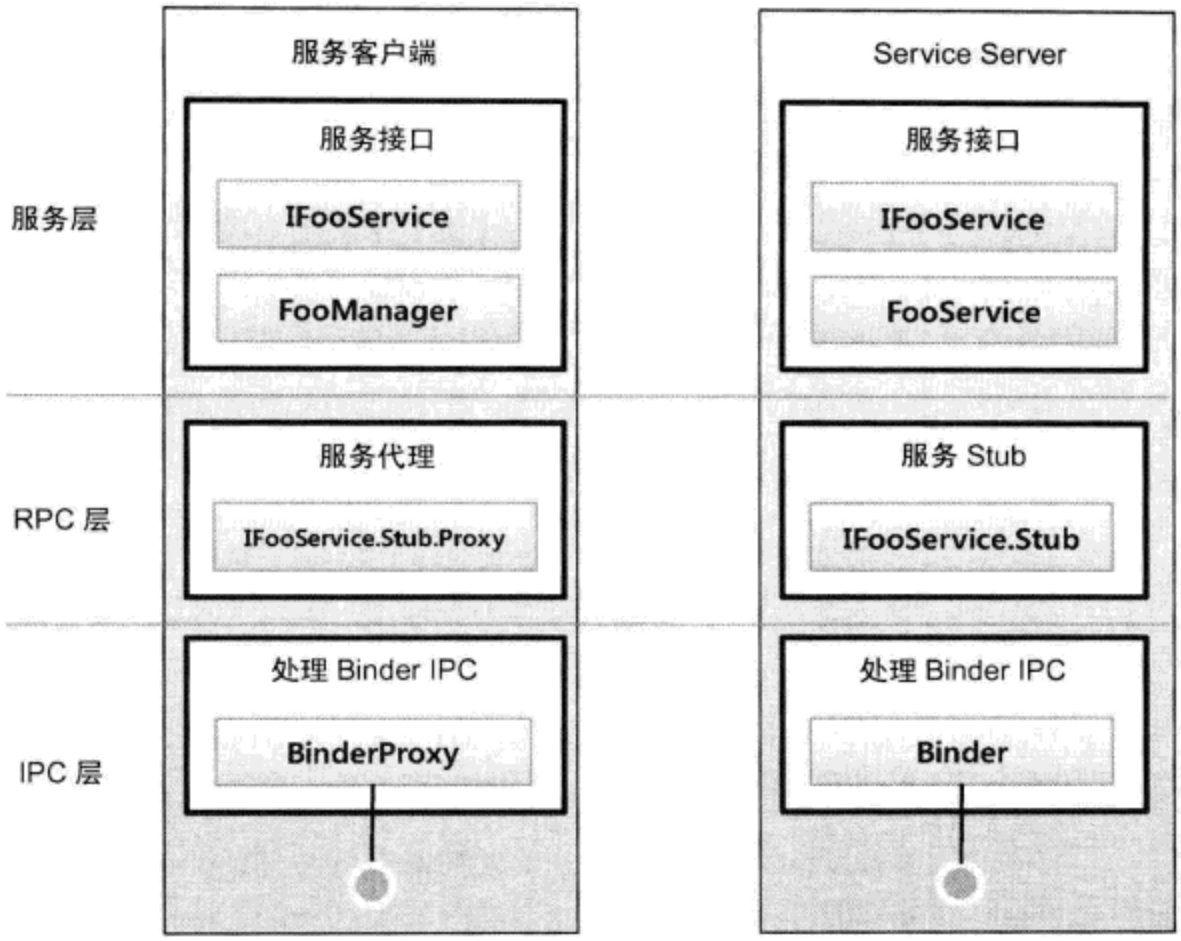


图 10-2 | Java 服务框架由服务层、RPC 层、IPC 层构成

服务层

与本地服务框架不同，之所以要实现 FooManager 类，是因为 SDK 包中并不包含 ServiceManager[1]类，应用程序无法使用 ServiceManager 类来注册或检索系统服务。

为了让应用程序开发者[2]能够使用系统服务，系统服务开发者必须将包装类包含到 SDK 中。例如，在图 10-3 中，为了让应用程序开发者能够使用 FooService，系统服务开发者就要编写 FooManager 包装类，添加使用 ServiceManager 获取 FooService 的功能，并将其编入 SDK 之中。然后应用程序开发者就可以通过包含在 SDK 中的 FooManager 类来使用 FooService 系统服务了。

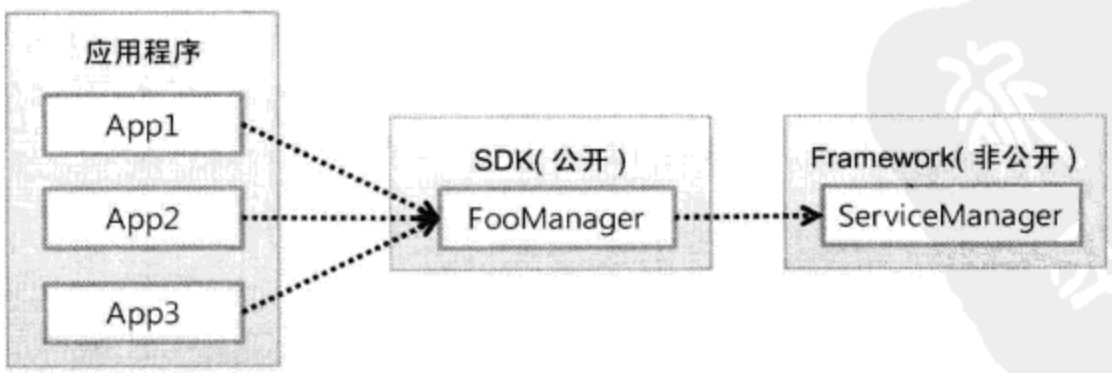


图 10-3 | FooManager 包装类的作用

1 /frameworks/base/core/java/android/os/ServiceManager.java

2 比如，某些 Android 手机厂商编写了一些独有的服务，他们就要把这些服务提供给应用程序开发者使用。

更多更新编程资源尽在 http://www.pin5i.com/

RPC 层

Java 服务框架使用 Android 平台中的 AIDL（Android Interface Definition Language）语言与编译器自动生成服务代理和服务 Stub。在 Android 系统中进程之间不能共享内存，为了使其他应用程序也可以访问本应用程序提供的服务，Android 系统采用了远程过程调用的方式来实现，并使用一种接口定义语言 AIDL 来公开服务的接口。

若 FooService 服务的服务 Stub 与代理都由 AIDL 自动生成，如使用 AIDL 语言将 IfooService 服务接口定义在 IFooService.aidl 文件中，那么 AIDL 编译器会自动生成 IFooService.Stub 与 IFooService.Stub.Proxy 两个类，如图 10-4 所示。

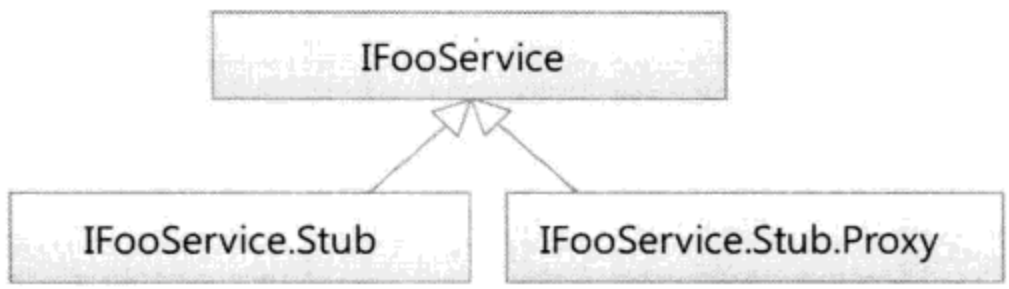


图 10-4 | AIDL 为 FooService 生成服务 Stub 与代理类

IPC 层

使用 Java 服务框架开发的服务与服务代理通过 Binder IPC 进行交互。为了支持 Binder IPC，本地服务框架提供了 BpBinder 与 BBinder 两个类，Java 服务框架则提供了 BinderProxy 与 Binder 两个类。

例如，IFooService.Stub.Proxy 服务代理在调用 foo()代理方法时，通过 Binder IPC，FooService 服务的 foo() Stub 方法就会被调用执行，此时服务代理向 BinderProxy 传递 RPC 代码与数据，服务通过 Binder 来接收传递过来的数据。

Java 服务框架也可以通过 JNI 使用本地服务框架的 Binder IPC，即 BinderProxy 与 Binder 通过 JNI 使用本地服务框架的 BpBinder 与 BBinder 类的功能。比如，在 BinderProxy 与 Binder 进行交互时，调用 BinderProxy 的 transact()方法会引起 Binder 的 execTransact()方法的调用，如图 10-5 所示。首先，BinderProxy 的 transact()方法使用 JNI 本地函数调用 BpBinder 的 transact()函数，而后通过 Binder IPC 调用 BBinder 的 transact()函数。接下来，BBinder 要调用 Binder 的 execTransact()方法，但自身却不提供这项功能。

若想向 BBinder 中添加新的功能，就要定义一个继承 BBinder 类的服务 Stub 子类，并且重写 onTransact()方法，这在第 8 章中已经讲解过，若已遗忘，请返回第 8 章参考相关内容。幸运的是 Java 服务框架提供了一个名称为 JavaBBinder 的服务 Stub 类，它继承了 BBinder 类，重写了 onTransact()函数，实现了对 Binder 的 execTransact()方法的调用。

Java 服务框架各层中包含的类如下所示。

- 服务层：IInterface[Java]

更多更新编程资源尽在：http://www.pin5i.com/

■ IPC层：Binder、BinderProxy、BinderInternal、Parcel[Java]

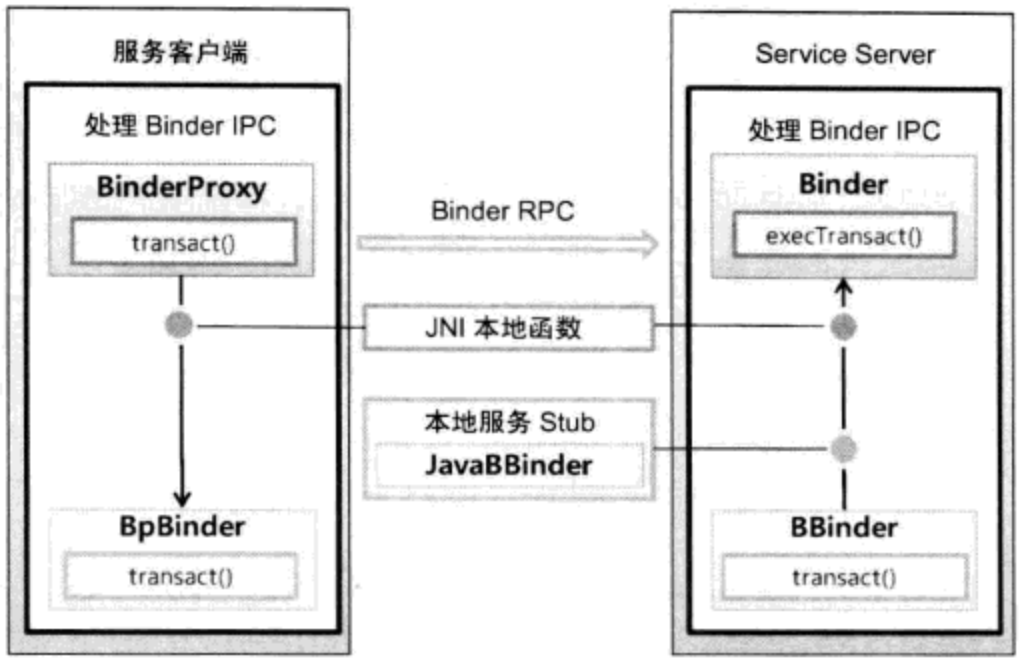


图 10-5 | BinderProxy 与 Binder 的 Binder IPC

TIP 术语区分

在介绍本地服务框架与 Java 服务框架时，有些类名称相同，为了便于区分，在类名称的右上位置使用了上标。比如 Parcel 类，若是本地服务框架中的 Parcle 类，将其标记为 Parcel[C++]；Java 服务框架中的 Parcle 类，标记为 Parcel[Java]。

10.1.2 Java 服务框架中各个类间的相互作用

如前所言，为了支持 Binder RPC，Java 服务框架通过 JNI 使用本地服务框架的功能，服务客户与 Service Server 的各个组成元素在纵向上进行相互作用，这也是两个框架的不同点之一。

首先，看一下 Java 系统服务用户所在的服务客户端，如图 10-6 所示。与本地服务框架相比，Java 服务框架在服务客户端中多了两个调用过程，分别是服务用户调用 FooManager 的 foo()方法的过程与 BinderProxy 的 transact()方法通过 JNI 本地函数 android_os_BinderProxy_transact()[1]调用 BpBinder 的 transact()函数的过程。当 BpBinder 的 transact()函数调用执行后，接下来的过程就与在本地服务框架中说明的内容一样了。

接着，再来看 Java 系统服务所在 Service Server，如图 10-7 所示，在 Java 服务框架中多了 BBinder 的 transact()函数使用 JavaBBinder 本地服务 Stub 调用 Binder 的 execTransact()方法的过程。

---

1 Java 服务框架中使用的 JNI 本地函数（android_os_BinderProxy_transact()函数）与 JavaBBinder 类将在 10.2 节中进行详细讲解。

更多更新编程资源尽在 http://www.pin5i.com/

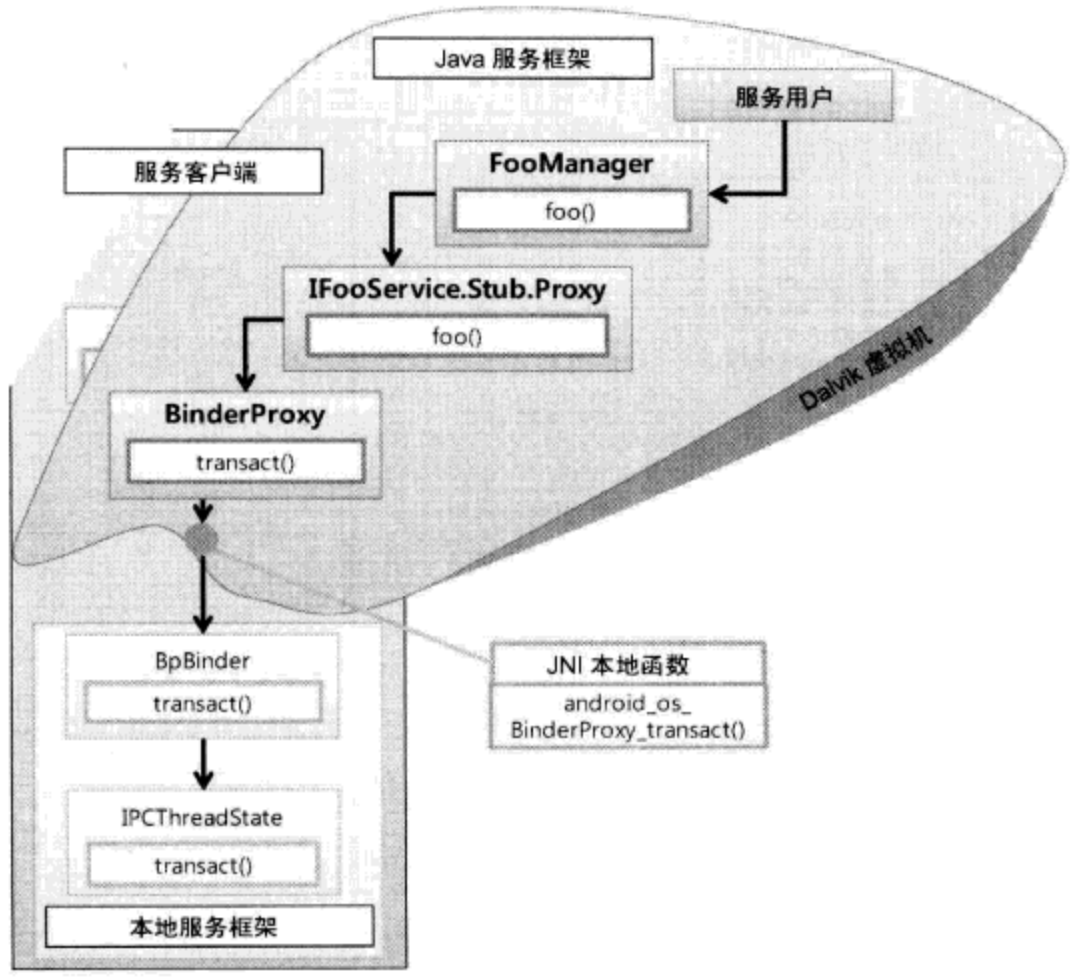


图 10-6

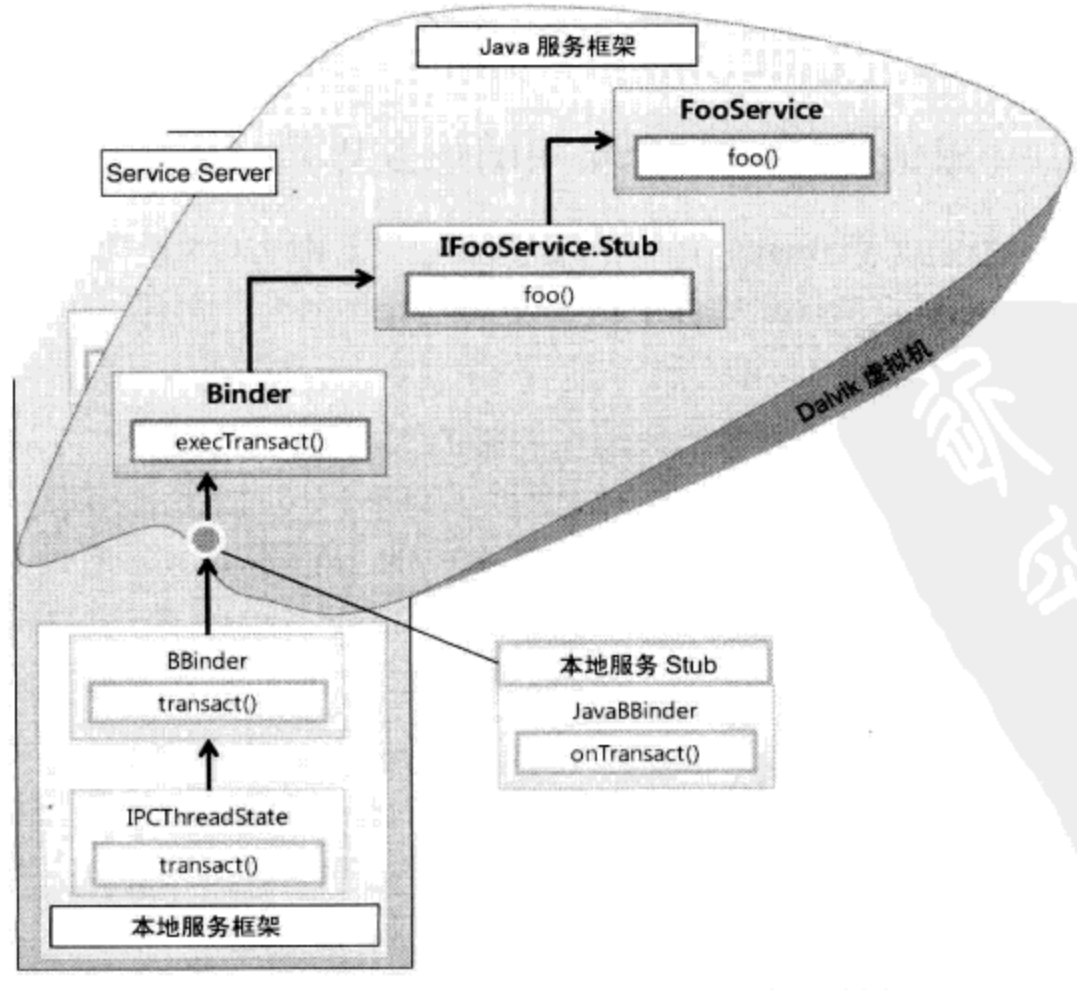


图 10-7 | Service Server 内部各部分的相互作用

更多更新编程资源尽在：http://www.pin5i.com/

为了帮助大家进一步理解 Java 服务框架各部分相互作用的过程，我们一起分析学习一下 FooService 服务注册到系统以及使用的过程。在将 Java 服务注册到系统时，Java 服务框架的各个部分之间是如何作用的呢？图 10-8 详细描述了这一过程。与本地服务框架类似，在服务注册与使用过程中涉及到的对象有 Service Server（提供服务）、服务客户端（使用服务）、Service Manager 以及 Binder Driver（用于在各个对象间提供通信支持）。

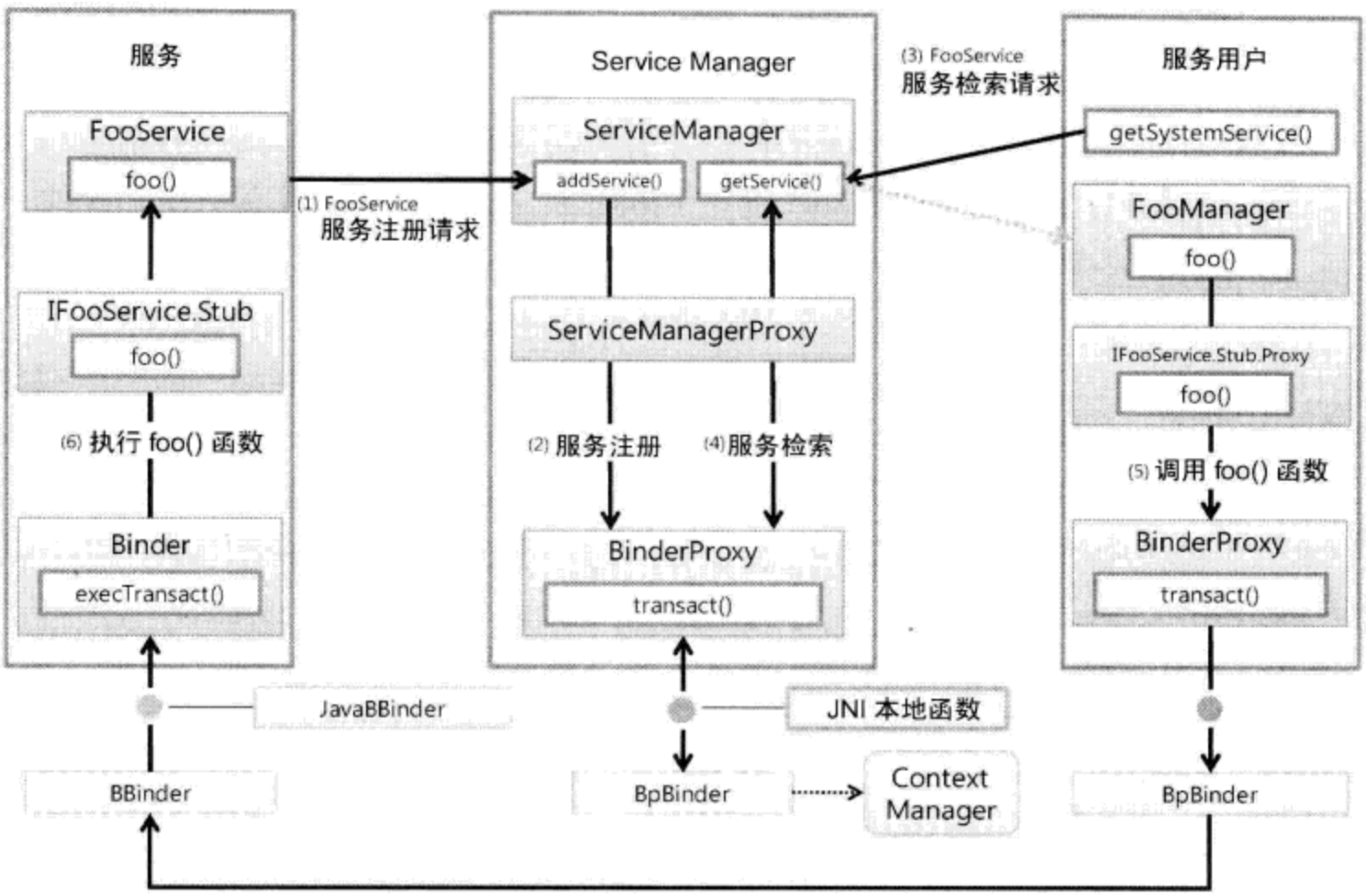


图 10-8 | 使用 FooService 时各部分间的相互作用

（1） 请求服务注册（服务）：在向系统注册服务时，本地服务框架将通过本地服务管理器 BpServiceManager 来处理服务注册，而 Java 服务框架则使用 Java 服务管理器 ServiceManager^{Java} 来处理服务注册。FooService 服务在向系统注册时将调用 ServiceManager^{Java} 的 addService()方法。在 ServiceManager^{Java} 内部存在 Binder Proxy 对象，它通过 JNI 与持有 Context Manager 指针的 BpBinder 连接在一起。

（2） 注册服务（服务管理器）：ServiceManagerProxy 服务代理将调用 addService()方法的信息转换为 RPC 数据。Binder RPC 数据被存储在 Parcel^{Java} 类中传递给 BinderProxy，而后通过 JNI 传递给 BpBinder，最后通过 Binder IPC 传递给 Context Manager，如此 FooService 服务即被注册到系统之中。

（3） 请求检索服务（服务用户）：在使用 FooService 服务时，本地服务用户将通过 BpService Manager 来检索服务，而 Java 服务用户则会调用 SDK 中的 getSystemService()方法来检索服务。

更多更新编程资源尽在 http://www.pin5i.com/

（4） 服务检索（服务管理器）：调用 getSystemService()方法将引起 ServiceManager[Java] 的 getService()方法的调用，在系统中检索 FooService 服务。若检索到 FooService，则会向服务用户返回 FooManager 对象，该对象可以引用 IFooService.Stub.Proxy 服务代理。

（5） 调用 foo()服务代理方法（服务用户）：服务用户调用 FooManager 的 foo()方法，而后 IFooService.Stub.Proxy 将 foo()方法的调用信息转换为 RPC 数据，通过 BinderProxy 传递给 BpBinder。

（6） 执行 foo()服务 Stub 方法（服务）：BBinder 从 Binder Driver 接收 Binder RPC 数据后，通过 JavaBBinder 调用 Binder 的 execTransact()方法。而后 RPC 数据被传递给 IFooService.Stub 服务 Stub 的 onTransact()方法，经过分析后，调用 FooService 的 foo()服务 Stub 方法。

以上我们学习了 Java 服务框架。Java 服务框架最重要的特征就是通过 JNI 复用本地服务框架提供的功能。特别地，在处理 IPC 层中的 Binder IPC 时，BinderProxy 与 Binder 类通过 JNI 复用 BpBinder 与 BBinder 类。为此，Java 服务框架分别提供了 JNI 本地函数 android_os_BinderProxy_transact()（用于连接 BinderProxy 与 BpBinder）与 JavaBBinder 服务 Stub 类（用于连接 Binder 与 BBinder）。

并且，使用 Java 服务框架实现的系统服务被包含在应用程序框架中，其中一部分可以使用在 SDK 中，在开发应用程序时，应当额外提供服务管理器（Service Manager）的包装类，以便使用系统服务。Android 系统也提供了一种接口描述语言 AIDL 用来自动为服务接口生成服务 Stub 与代理。

10.2 运行机制

Java 服务框架各组成部分之间是如何相互作用的，Binder、BinderProxy、Parcel 类是如何通过 JNI 使用本地服务框架的功能的，各个类是如何生成的，如何设置 JNI 本地函数。本节将通过源码分析来学习 Java 服务框架使用本地服务框架的运作机制。

TIP Java 服务框架的运作机制与 JNI

Ubuntu 一词来自南非的祖鲁语或科萨语，原意为“因为有你所以有我”，是非洲传统的一种价值观，类似于我们所说的“仁爱”，Ubuntu 操作系统将这种思想带入软件世界。Ubuntu 操作系统每隔六个月发布一次新版本，标注在名称后的数字代表发布的年份与月份。至 2010 年 7 月，发布的最新版本为 Ubuntu 10.04 LTS（Long Term Support）。LTS 版本每两年发布一次，是长期支持版本，其桌面版本提供 3 年支持，服务器版本则提供长达 5 年的支持。

更多更新编程资源尽在 http://www.ibiancheng.net/

Java 服务框架通过 JNI 使用本地服务框架提供的功能，在本章的内容讲解中将会涉及到很多与 JNI 相关的内容，在运作机制的学习过程中，请随时参考第 4 章“JNI 与 NDK”中的相关部分。

10.2.1 Java 服务框架初始化

在第 4 章我们已经提到过，当 app_process 进程启动时，AndroidRuntime 类就会调用 startReg()函数，将 JNI 本地函数加载到 Dalvik 虚拟机中。此时，使用 resgister_android_os_Binder()函数注册的 JNI 本地函数就是与 Java 服务框架有关联的本地函数。

```
int register_android_os_Binder(JNIEnv* env)
{
    if (int_register_android_os_Binder(env) < 0)
        return -1;
    if (int_register_android_os_BinderInternal(env) < 0)
        return -1;
    if (int_register_android_os_BinderProxy(env) < 0)
        return -1;
    if (int_register_android_os_Parcel(env) < 0)
        return -1;
    return 0;
}
```

代码 10-1 | android_util_binder.cpp[1]中的 resgister_android_os_Binder()函数

代码 10-1 是 resgister_android_os_Binder()函数的一部分，在函数中调用了另外四个函数，函数名称的最后一个词表示使用 JNI 本地函数的 Java 类名称。接下来，将逐一介绍各个类及其相应的 JNI 本地函数。BinderInternal 类用在 ServiceManager[Java] 中，将在 10.4“服务管理器”一节中详细介绍，另外三个类分别在 10.2.2“Binder 类”、10.2.3“BinderProxy 类”、10.2.4“Parcel[Java] 类”中进行介绍。

10.2.2 Binder

Binder 类的 JNI 设置

在使用 Binder 类之前，应当先向 Dalvik 虚拟机注册与 Binder 本地方法相对应的 JNI 本地函数。如代码 10-1 所示，调用 int_resgister_android_os_Binder()函数，Binder 类的部分信息将被保存到 gBinderOffsets 全局变量中，并且 Binder 类的本地方法将被映射到

1 /frameworks/base/core/jni/android_util_Binder.cpp

更多更新编程资源尽在 http://www.pin5i.com/

JNI 本地函数中。

```
static int int_register_android_os_Binder(JNIEnv* env)
{
    jclass clazz;

    clazz = env->FindClass(kBinderPathName);                    ❶
    LOG_FATAL_IF(clazz == NULL, "Unable to find class android.os.Binder");

    gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);  ❷
    gBinderOffsets.mExecTransact
        ➥ = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
    assert(gBinderOffsets.mExecTransact);

    gBinderOffsets.mObject
        ➥ = env->GetFieldID(clazz, "mObject", "I");
    assert(gBinderOffsets.mObject);

    return AndroidRuntime::registerNativeMethods(               ❸
        ➥ env, kBinderPathName,
        ➥ gBinderMethods, NELEM(gBinderMethods));
}
```

代码 10-2 | int_resgister_android_os_Binder()函数

代码 10-2 是 int_resgister_android_os_Binder()函数的源码。❶首先在 Dalvik 虚拟机中，查找 android.os.Binder 类，而后❷将要在 JNI 本地函数中使用的 Binder 类的主要信息保存到 bindernative_offsets_t 结构体中。

bindernative_offsets_t 结构体中保存的信息有 Binder 的类信息、execTransact()方法 ID、mOject 变量 ID。请看图 10-9，它显示了 bindernative_offsets_t 结构体中存储的 Binder 类的信息。

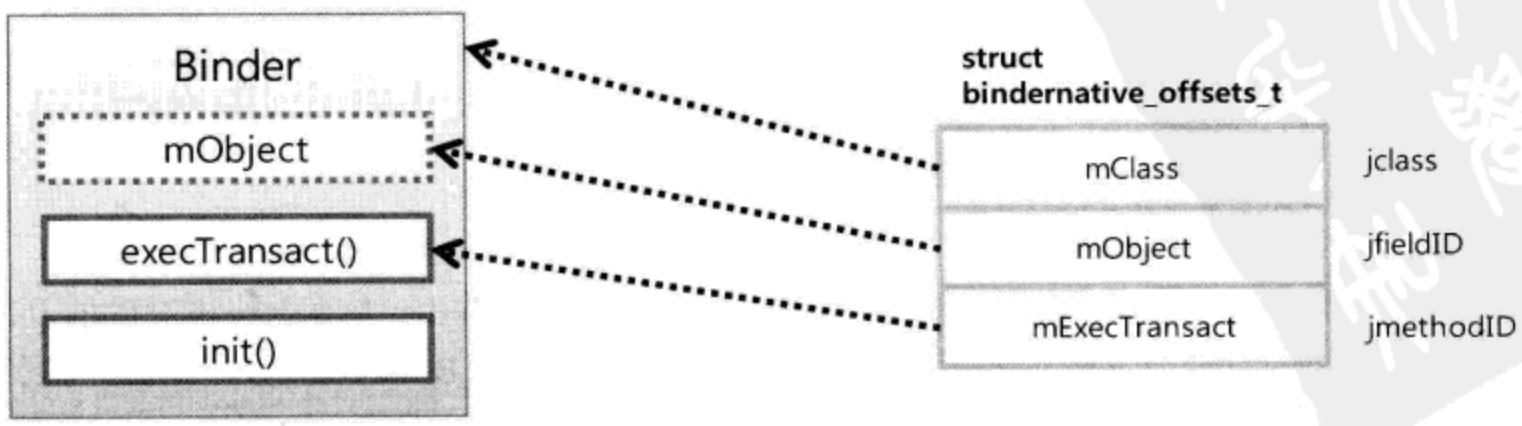


图 10-9 | 存储在 bindernative_offsets_t 结构体中的 Binder 类的信息

然后，❸调用 AndroidRuntime 类的 registerNativeMethods()函数，将与 Binder 本地

更多更新编程资源尽在 http://www.pin5i.com/

方法相对应的 JNI 本地函数注册到 Dalvik 虚拟机中。图 10-10 描述了 Binder 本地方法与 JNI 本地函数[1]的对应映射关系。

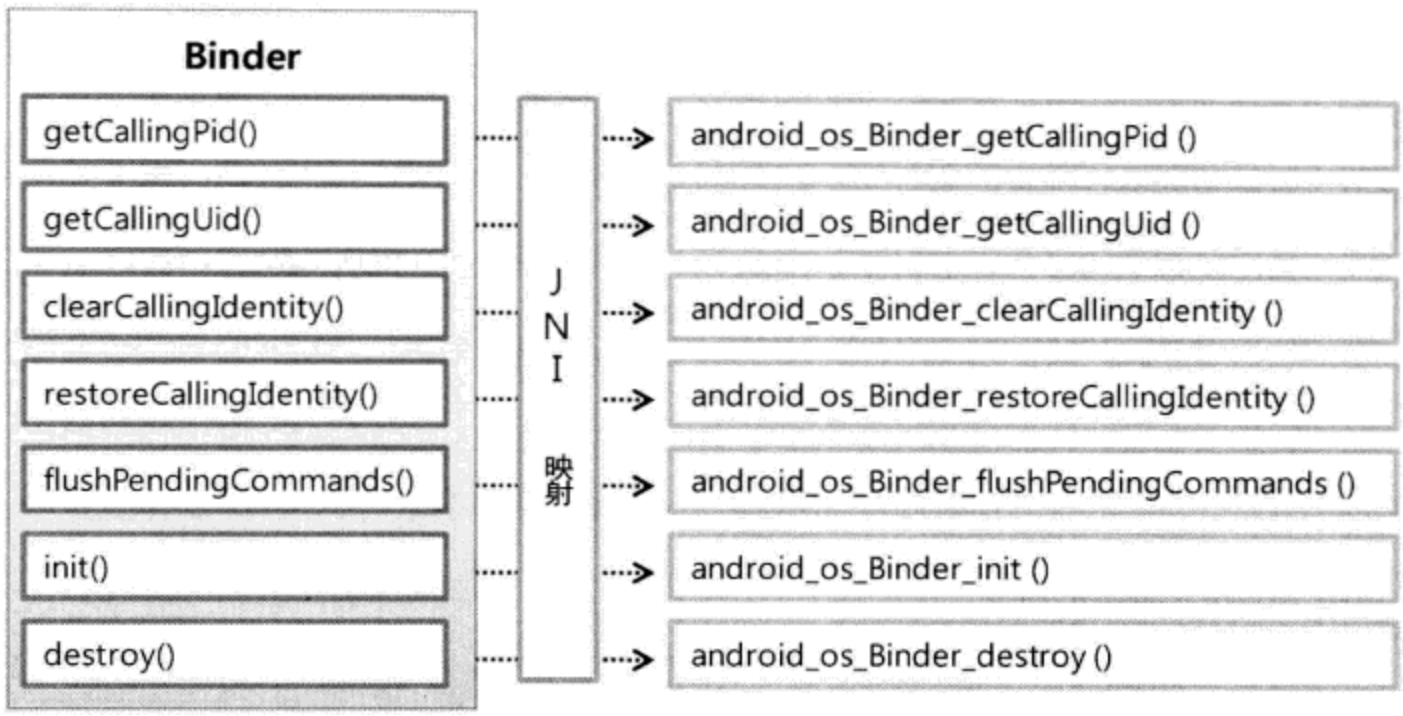


图 10-10 | Binder 类的本地方法与 JNI 本地函数的映射关系

生成 Binder 对象

为了进行 Binder IPC，使用 BBinder 提供的功能，Binder 类在生成 Binder 对象的同时将生成 BBinder 对象。请看代码 10-3，它给出了 Binder 类构造方法的源码。Binder 类在构造方法中调用 init()本地方法，由图 10-10 可知，init()方法与 JNI 的 android_ os_Binder_ init()函数映射在一起。

```
public class Binder implements IBinder {
   private native final void init();
   public Binder() {
      init();
   }
}
```

代码 10-3 | Binder.java-Binder 类构造方法

代码 10-4 是 android_os_Binder_init()函数源码，它首先生成 JavaBBinderHolder 类的对象，而后调用名称为 SetIntField()的 JNI 函数，将刚创建的 JavaBBinderHolder 对象

1 一般而言，在 Android 源码中，将 Java 类的 Fully Qualified Name（类的全名，包含所在的包）中的“.”替换为“_”，而后再在尾部添加上“_本地方法名称”，即是与 Java 本地方法相对应的 JNI 本地函数名称。例如，Binder 类的 init()本地方法，其所在的包为 android.os，先将 android.os.Binder 转换为 android_os_Binder，再添加上本地方法名称，所以它对应的 JNI 本地方法为 android_os_Binder_init()。

更多更新编程资源尽在 http://www.pin5i.com/

的地址保存到 Binder 的 mObject 变量中。

```
static void android_os_Binder_init(JNIEnv* env, jobject clazz)
{
    JavaBBinderHolder* jbh = new JavaBBinderHolder(env, clazz);
    if (jbh == NULL) {
        jniThrowException(env, "java/lang/OutOfMemoryError", NULL);
        return;
    }
    LOGV("Java Binder %p: acquiring first ref on holder %p", clazz, jbh);
    jbh->incStrong(clazz);
    env->SetIntField(clazz, gBinderOffsets.mObject, (int)jbh);
}
```

代码 10-4 | android_os_Binder_init()函数

生成 JavaBBinder 对象

如上所言，在创建 Binder 对象的同时会生成 BBinder 对象，但我们在分析代码的过程中仅发现一段创建 JavaBBinderHolder 对象的代码，并未发现创建 BBinder 对象的代码。这是怎么回事？BBinder 对象究竟在哪里被创建了呢？

代码 10-5 是 JavaBBinderHolder 类的主要代码，其构造函数非常简单，它只是将 Binder 对象的地址保存到 mObject 变量之中。

```
class JavaBBinderHolder : public RefBase
{
public:
    JavaBBinderHolder(JNIEnv* env, jobject object)
        : mObject(object)
    {
        LOGV("Creating JavaBBinderHolder for Object %p\n", object);
    }

   sp<JavaBBinder> get(JNIEnv* env){}
private:
    Mutex           mLock;
    jobject         mObject;
    wp<JavaBBinder> mBinder;
};
```

代码 10-5 | android_util_Binder.cpp[1]-JavaBBinderHolder 类的主要代码

事实上，JavaBBinder 对象是在 JavaBBinderHolder 的 get()函数中创建的，如代

1 /frameworks/base/core/jni/android_util_Binder.cpp

更多更新编程资源尽在：http://www.pin5i.com/

码 10-6 所示，在 get()函数中直接调用 JavaBBinder 类的构造函数，生成了 JavaBBinder 对象。JavaBBinder 类继承并实现了父类 BBinder，创建 JavaBBinder 对象[1]时也会生成 BBinder 的对象。

```
sp<JavaBBinder> JavaBBinderHolder::get(JNIEnv* env)
{
    AutoMutex _l(mLock);
    sp<JavaBBinder> b = mBinder.promote();
    if (b == NULL) {
        b = new JavaBBinder(env, mObject);
        mBinder = b;
    }

    return b;
}
```

代码 10-6 | JavaBBinderHolder 的 get()函数

图 10-11 简单地描述了创建 Binder 与 JavaBBinder 对象的过程。首先使用 Binder

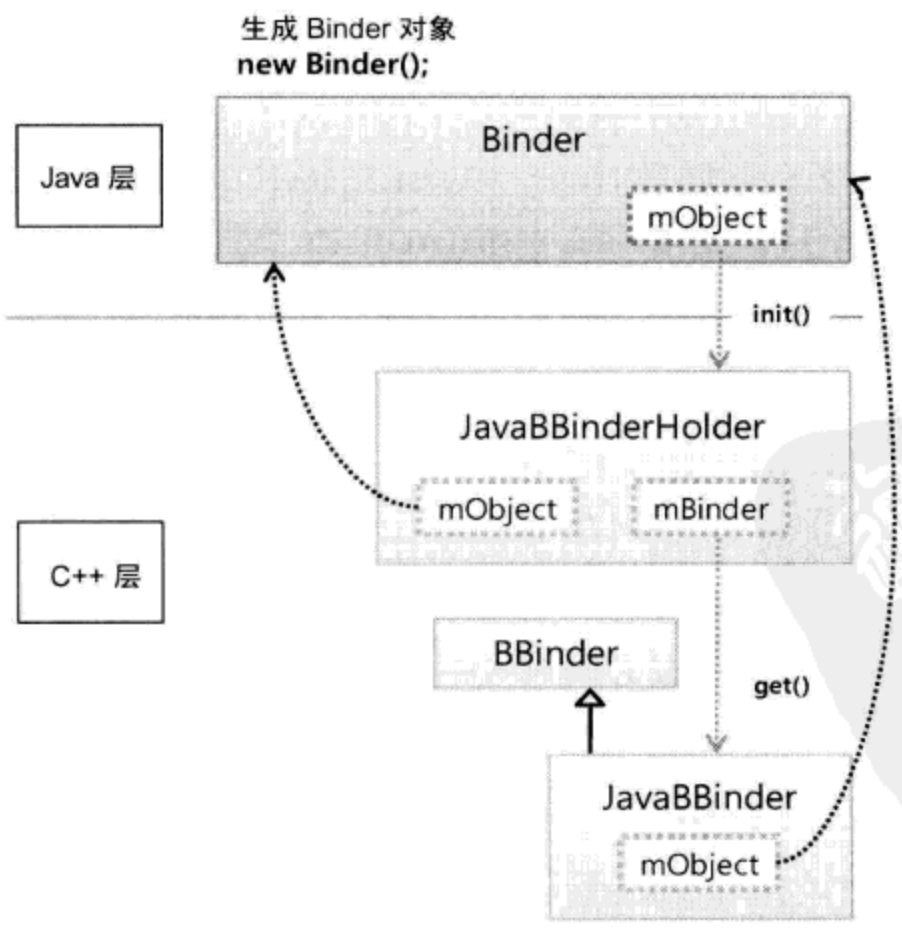


图 10-11 | 生成 BBinder 对象

1 创建 JavaBBinder 对象时，并非一创建 Binder 对象时就生成它，而是在需要的时候才生成它，设计时使用了一种称为“延迟加载”（Lazy Loading）的方法。

更多更新编程资源尽在 http://www.pin5i.com/

构造方法，调用本地方法 init()，进而调用 JNI 本地函数（android_os_Binder_init()），生成 JavaBBinderHolder 对象。JavaBBinderHolder 构造函数接收 Binder 对象（调用 init()方法的对象）的地址，将其保存到 mObject 变量中。而后调用 JavaBBinderHolder 的 get()方法调用 JavaBBinder()构造函数。JavaBBinder()构造函数以 JavaBBinderHolder 的 mObject 为参数，将其保存到自身的 mObject 变量中，最后生成 JavaBBinder 对象。

Binder 类与 JavaBBinder 服务 Stub 类的相互作用

在 10.1.2 一节中，已经讲过，调用 BBinder 的 transact()函数将引起对 JavaB Binder 的 onTransact()函数的调用。除了基本的 Binder RPC 函数外，若想扩展 BBinder 的功能，需要在继承 BBinder 的服务 Stub 类中重新定义 onTransact()方法。JavaBBinder 服务 Stub 类继承了 BBinder 类，重定义了 onTransact()方法，在其中调用了 Binder 的 exec Transact()方法。

代码 10-7 是 JavaBBinder 的 onTransact()函数源码。在 C++代码中通过 JNI 调用 Java 类方法时，要使用名称为 CallxxxMethod()的 JNI 函数。

由于 Binder 的 execTransact()方法返回 boolean 类型的值，所以应该使用名称为 Call BooleanMethod()的 JNI 函数调用 Binder 的 execTransact()方法。

在调用名称为 CallBooleanMethod()的 JNI 函数[1]时，需要提供 Binder 对象的地址以及 execTransact()方法的 ID。请看代码 10-7，由代码可以看出，CallBooleanMethod()第一个参数是 Binde 对象的地址，它存储在 JavaBBinder 的 mObject 变量中；第二个参数为 execTransact()方法的 ID，它保存在 gBinderOffsets 结构体（该结构体是在进行 Binder 类的 JNI 设置过程中生成的）的 mExecTransact 变量中。如此一来，使用 CallBoolean Method()方法即可调用 Binder 的 execTransact()方法。

```
virtual status_t JavaBBinder::onTransact(
      ➥ uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
{
   JNIEnv* env = javavm_to_jnienv(mVM);
   jboolean res = env->CallBooleanMethod(mObject,
   gBinderOffsets.mExecTransact, code, (int32_t)&data,
                                       ➥ (int32_t)reply, flags);

   return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
}
```

代码 10-7 | JavaBBinder 之 onTransact()函数的主要代码

调用 Binder 类的 execTransact()方法，将引起 onTransact()方法的调用，onTransact()

1 调用 JNI 函数时，需要先调用 javavm_to_jnienv()函数获取 JNIEnv 对象。

更多更新编程资源尽在 http://www.pin5i.com/

方法的代码如 10-8 所示。❶就像本地服务 Stub 继承 BBinder 类重新定义 onTransact()函数添加新功能一样，Java 服务 Stub 类可以通过继承 Binder 类，重新定义 onTransact()函数，添加与 RPC 代码相对应的服务 Stub 方法。

```
private boolean execTransact(int code, int dataObj, int replyObj,
    int flags) {
    Parcel data = Parcel.obtain(dataObj);
    Parcel reply = Parcel.obtain(replyObj);

    boolean res;
    try {
        res = onTransact(code, data, reply, flags);      ←❶
    }
}
```

代码 10-8 | Binder.java-execTransact()方法的主要代码

10.2.3 BinderProxy

BinderProxy 类的 JNI 设定

在使用 BinderProxy 类之前，首先必须将 BinderProxy 类本地方法对应的 JNI 本地函数注册到虚拟机中。在代码 10-1 中，调用 int_register_android_os_BinderProxy()函数时，BinderProxy 类的部分信息就被保存到 gBinderProxyOffsets 全局变量中，并且将 BinderProxy 类的本地方法与 JNI 本地函数映射在一起。

图 10-12 是保存着 BinderProxy 类信息的 binderproxy_offsets_t 结构体，图 10-13 描述了 BinderProxy 类本地方法与 JNI 本地函数的映射关系。注意 BinderProxy 类构造方法 ID 保存在 mConstructor 变量中，该 ID 用于 JNI 本地函数创建 BinderProxy 对象的过程中。

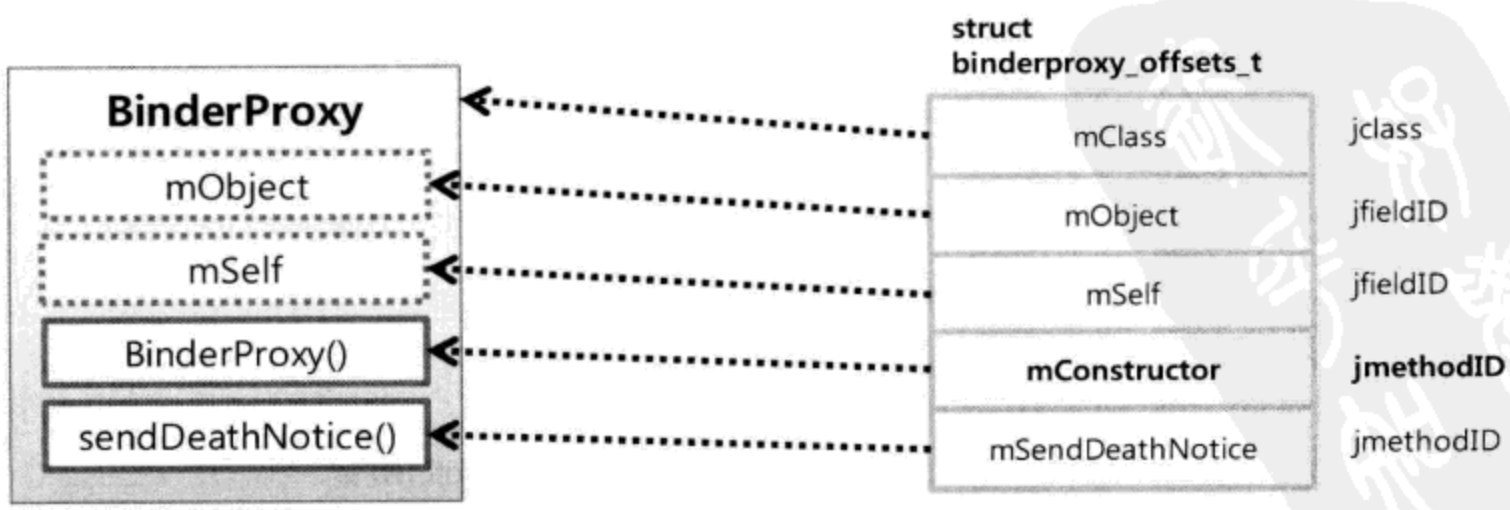


图 10-12 | 保存 BinderProxy 类信息的 binderproxy_offsets_t 结构体

更多更新编程资源尽在 http://www.pin5i.com/

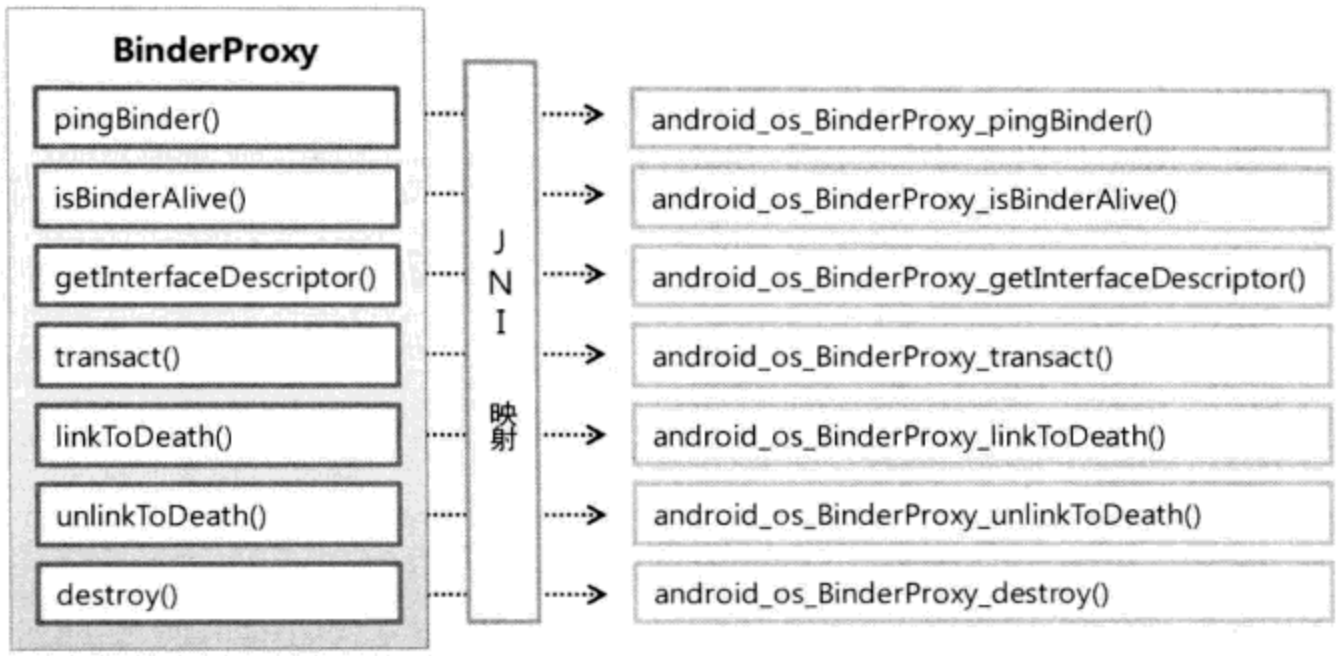


图 10-13 | BinderProxy 类本地方法与 JNI 本地函数的映射关系

生成 BinderProxy 对象

BinderProxy 类在执行 Binder IPC 时需要使用本地服务框架的 BpBinder 功能，因此在创建 BinderProxy 对象时将同时生成 BpBinder 对象。在第 8 章中已经讲过，BpBinder 对象是由 Parcel^{C++}的 readStrongBinder()函数创建的。

因此，BinderProxy 对象是在调用 Parcel^{Java} 的 readStrongBinder()方法时生成的。当调用 Parcel^{Java} 的 readStrongBinder()方法时，Parcel^{C++}的 readStrongBinder()函数以及 javaObjectForBinder()函数将依次被调用，从而生成 BpBinder 与 BinderProxy 对象，如图 10-14 所示。

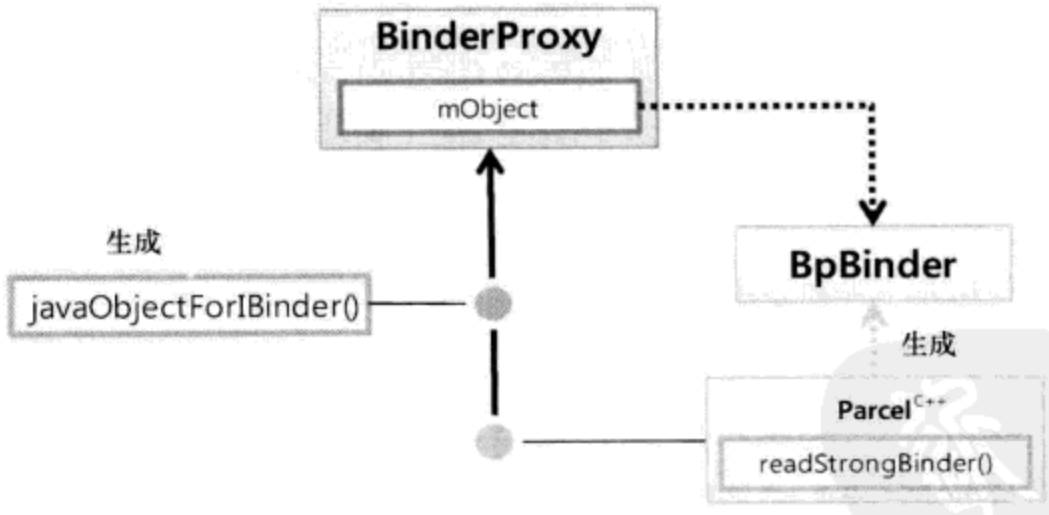


图 10-14 | 生成 BinderProxy 对象-调用 Parcel^{Java} 的 readStrongBinder()方法

C++在通过 JNI 生成 Java 层的对象时，要调用名称为 NewObject()的 JNI 函数，并且调用该函数时需要提供类的信息以及构造方法的 ID。在“BinderProxy 类的 JNI 设定”部分中，已经将 BinderProxy 类信息与构造方法的 ID 保存到 gBinderProxyOffsets 的 mConstructor 变量，因此可以使用这两个参数调用 NewObject()函数。

下面来分析 javaObjectForBinder()函数，该函数的主要代码如代码 10-9 所示。在

更多更新编程资源尽在 http://www.pin5i.com/

❶中调用名称为 NewObject()的 JNI 函数，生成 BinderProxy 对象后，将 BpBinder 对象[1]保存到 BinderProxy 的 mObject 变量中。

```
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
   if (val->checkSubclass(&gBinderOffsets)) {
   jobject object = static_cast<JavaBBinder*>(val.get())->object();
   return object;
   }

   object = env->NewObject(gBinderProxyOffsets.mClass,
                           ➥ gBinderProxyOffsets.mConstructor);          ←❶
   if (object != NULL)
      env->SetIntField(object,
                           ➥ gBinderProxyOffsets.mObject, (int)val.get());
   return object;
}
```

代码 10-9 | javaObjectForBinder()函数的主要代码

BinderProxy 类与 BpBinder 类的相互作用

如图 10-5 所示，当调用 BinderProxy 的 transact()本地方法时，BpBinder 的 transact()函数就会被调用。由于 BinderProxy 的 transact()本地方法与 JNI 本地函数 android_os_BinderProxy_transact()映射在一起，当 BinderProxy 与 BpBinder 相互作用时，JNI 本地函数即会被调用。

请看代码 10-10，它是 android_os_BinderProxy_transact()函数的主要代码。❶获取 BinderProxy 的 mObject 变量中的 BpBinder 对象地址，而后调用 transact()函数。

```
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
     ➥   jint code, jobject dataObj, jobject replyObj, jint flags)
{
   Parcel* data = parcelForJavaObject(env, dataObj);
   Parcel* reply = parcelForJavaObject(env, replyObj);

   IBinder* target = (IBinder*)
   env->GetIntField(obj, gBinderProxyOffsets.mObject);          ←❶

   status_t err = target->transact(code, *data, reply, flags);
}
```

代码 10-10 | android_os_BinderProxy_transact()函数的主要代码

1 在第 8 章讲解 Android 服务框架时，提到过 BpBinder 对象是调用 ParcelC++的 readStrongBinder()函数创建的，它被作为 javaObjectForBinder()函数的第二个参数传入函数中。

更多更新编程资源尽在 http://www.pin5i.com/

10.2.4 Parcel

在 Binder IPC 期间，Parcel^{Java} 类用来保存由发送端向接收端发送的数据。特别地，Parcel^{Java} 在内部缓冲区中持有 IBinder 对象的引用，并且在进程间移动时需要维持这个引用。为此，Java 服务框架需要通过 JNI 来使用 Parcel^{C++} 类提供的功能。

Parcel^{Java} 类的 JNI 设定

Parcel^{Java} 类的本地方法通过 JNI 与 Parcel^{C++} 类中同名称的本地成员函数映射在一起。在代码 10-1 中，调用 int_register_android_os_BinderProxy()函数时，Parcel^{Java} 类的部分信息就被保存到 gParcelOffsets 全局变量中，并且将 Parcel^{Java} 类的本地方法与 JNI 本地函数映射在一起。图 10-15 描述了 Parcel^{Java} 类的本地方法与 JNI 本地函数的映射关系。由于 Parcel^{Java} 类的本地方法很多，大约 20 个以上，在图中只标出了最具代表性的三个，分别为 init()方法（用于初始化 Parcel^{Java} 类），以及用于向 Parcel^{Java} 读写 IBinder 对象的 readStrongBidner()、WriteStrongBidner()方法。

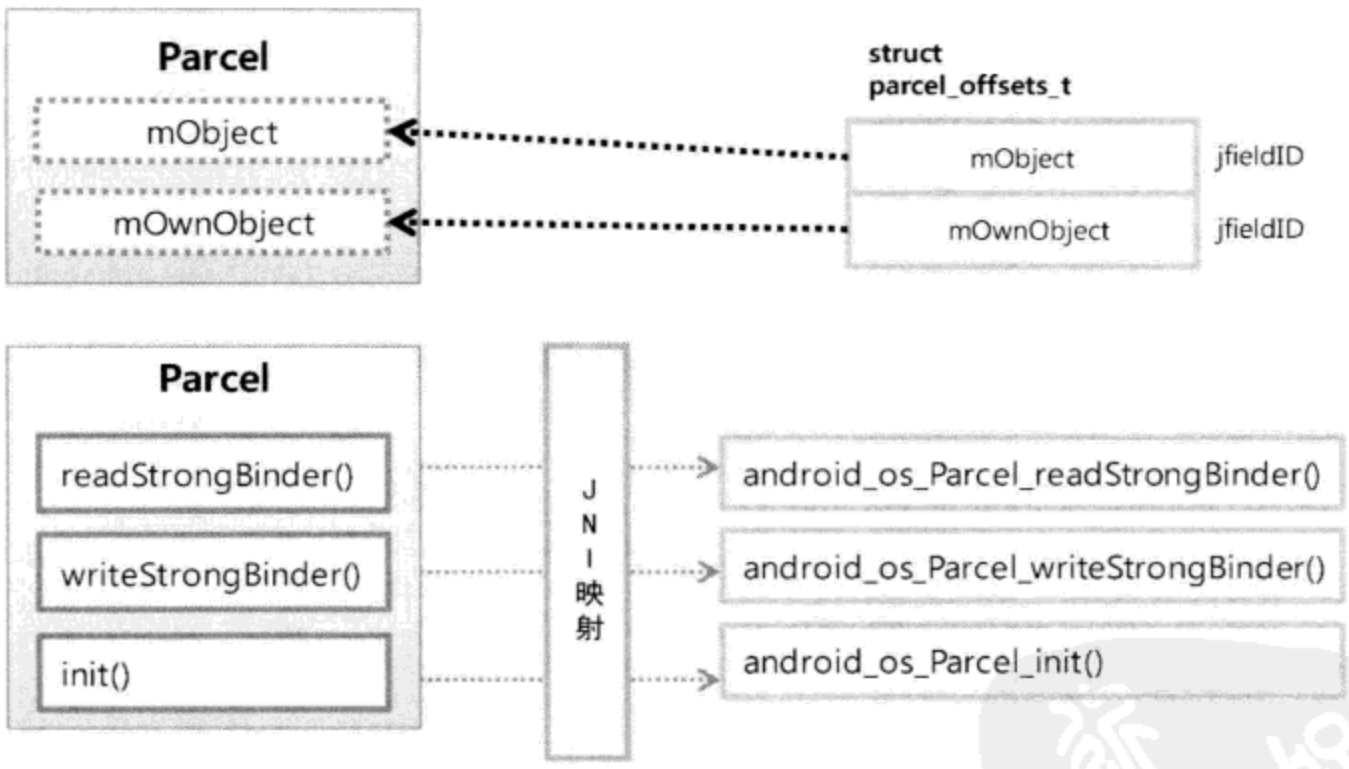


图 10-15 | Parcel^{Java} 类的本地方法与 JNI 本地函数的映射关系

生成 Parcel^{Java} 对象

生成 Parcel^{Java} 对象的过程与 Binder 和 BinderProxy 略有不同。首先来看 Parcel^{Java} 的构造方法，如代码 10-11 所示，它是一个私有方法，外部代码不能使用 new Parcel()语句创建 Parcel^{Java} 的实例对象。

但是，Parcel^{Java} 类提供了一个名称为 obtain()方法，使用该方法即可创建 Parcel^{Java}

更多更新编程资源尽在：http://www.pin5i.com/

类的实例对象。

```
public final class Parcel {
   private Parcel(int obj) {
      init(obj);
   }
   public static Parcel obtain() {}
}
```

代码 10-11 | Parcel.java[1]-Parcel 类的主要代码

obtain()方法调用 Parcel[Java] 的私有构造方法，而构造方法将调用 init()本地方法，从而引起对本地函数 android_os_Parcel_init()的调用。本地函数 android_os_Parcel_init()的主要代码如 10-12 所示，由代码❶可以看出，本地函数创建了 Parcel[C++]类的实例对象。

```
static void android_os_Parcel_init(JNIEnv* env, jobject clazz, jint parcelInt)
{
   Parcel* parcel = (Parcel*)parcelInt;
   int own = 0;
   if (!parcel) {
         own = 1;
      parcel = new Parcel;          ←❶
   }
   env->SetIntField(clazz, gParcelOffsets.mOwnObject, own);
   env->SetIntField(clazz, gParcelOffsets.mObject, (int)parcel);
}
```

代码 10-12 | 本地函数 android_os_Parcel_init()的主要代码

生成 Parcel[Java] 对象的过程如图 10-16 所示。首先通过 Parcel[Java] 类的 obtain()方法调用 Parcel[Java] 类的构造方法，进而调用本地方法 init()，在 C++层创建 Parcel[C++]类的实例对象，最后将生成的实例对象保存到 Parcel[Java] 类的 mObject 变量中。

Parcel[Java] 类与 Parcel[C++]类间的相互作用

一般地，Parcel[Java] 类用来在服务代理中保存 Binder RPC 数据。例如，FooService 服务用户在调用 IFooService.Stub.Proxy 服务代理的 foo()代理方法时，就会调用 Parcel[Java] 类的 obtain()函数创建出 Parcel[Java] 对象，并将其传入到 transact()方法中，如代码 10-13 所示。

1 /frameworks/base/core/java/android/os/Parcel.java

更多更新编程资源尽在 http://www.pin5i.com/

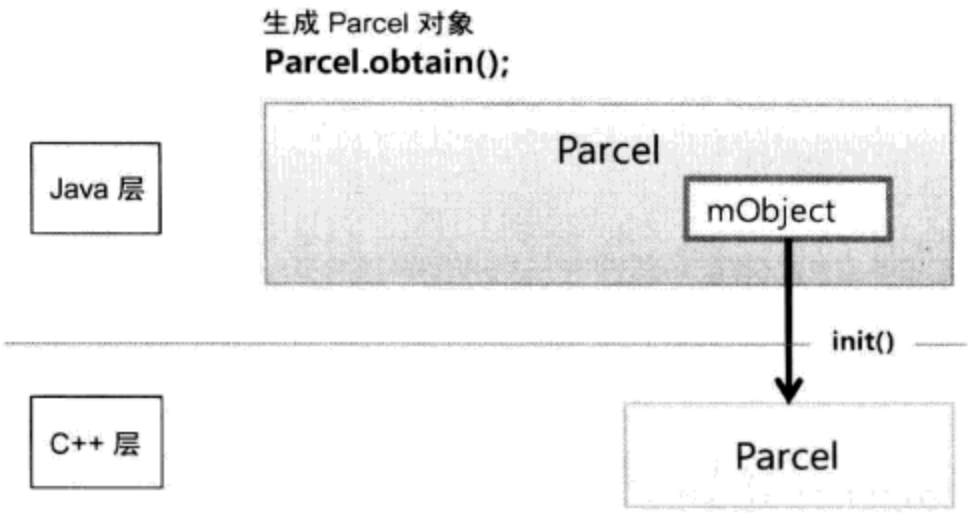


图 10-16 | 生成 Parcel[Java] 类对象

```
public void foo(){
   Parcel data = Parcel.obtain();
   transact(TRANSACTION_FOO, data, null, 0);
}
```

代码 10-13 | IFooService.Stub.Proxy 的 foo()服务代理方法

请看图 10-17，它描述了调用 BinderProxy 的 transact()方法后，Parcel[Java] 对象的传

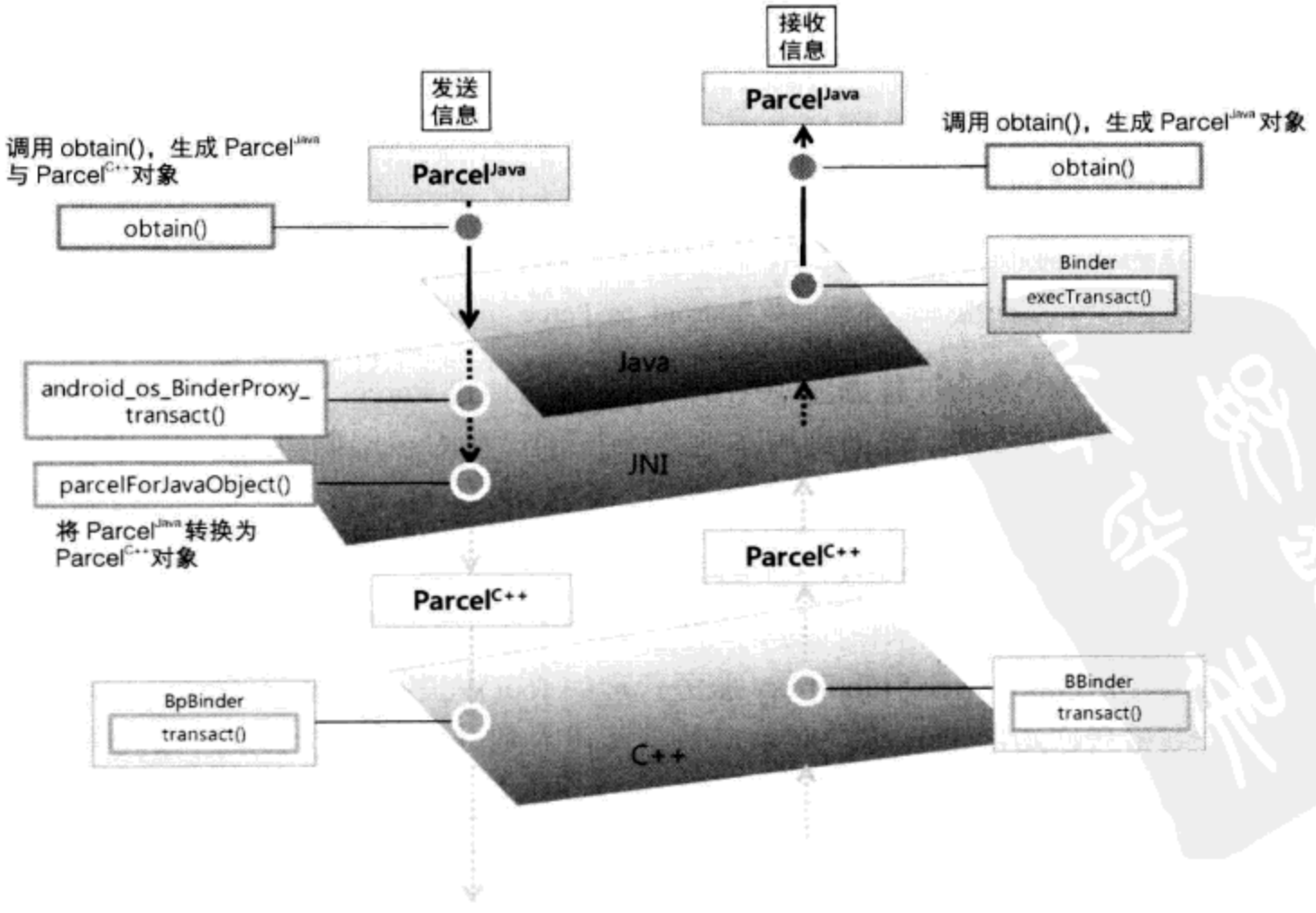


图 10-17 | Parcel[Java] 类与 Parcel[C++]类的相互作用

更多更新编程资源尽在：http://www.pinsixi.com/

递路线。服务代理进行 Binder RPC 时，在虚拟机中生成的 Parcel^{Java} 对象就要被传递到服务中，为此需要将 Parcel^{Java} 对象转换为 Parcel^{C++}对象，Java 服务框架提供了 parcelForJavaObject()函数来完成这一转换。

代码 10-14 是 parcelForJavaObject()函数的源码。parcelForJavaObject()函数将 Parcel^{Java} 对象转换为 Parcel^{C++}对象的语句如❶所示，首先调用名称为 GetIntField()的 JNI 函数，从 Parcel^{Java} 的 mObject[1]变量中获取 Parcel^{C++}对象的地址，而后将其转换为 Parcel^{C++} 指针。

```
Parcel* parcelForJavaObject(JNIEnv* env, jobject obj)
{
    Parcel* p = (Parcel*)env->GetIntField(obj,
                    gParcelOffsets.mObject);          ← ❶
    return p;
}
```

代码 10-14 | parcelForJavaObject()函数

在相反的方向上，通过 BBinder 的 transact()函数接收到 Parcel^{C++}对象后，需要将 Parcel^{C++}对象转换为 Parcel^{Java} 对象。如代码 10-8，调用 Binder 类的 execTransact()函数时，Parcel^{C++}对象被作为参数传入函数中。然后调用 Parcel^{Java} 的 obtain()方法生成 Parcel^{Java} 类对象，如代码 10-15。此时，Parcel^{Java} 的构造方法将调用 init()本地方法，若存在 Parcel^{C++} 对象地址，则不生成 Parcel^{C++}对象，并将其保存到 gParcelOffsets 的 mObject 变量中，如代码 10-12 所示。

```
static protected final Parcel obtain(int obj) {
    return new Parcel(obj);
}
```

代码 10-15 | Parcel.java-obtain()方法

10.3 Java 系统服务的实现

若想开发出运行在 Android 平台上的高效的 Java 系统服务，一种比较有效的方法是分析研究一些优秀的 Java 系统服务程序，并学习这些程序的组织结构与设计思路，进而将这些学到的知识应用到具体的 Java 系统服务开发中，以提高程序的效率，保证程序的准确性。下面以闹钟服务（AlarmManagerService）为基础，同各位一起分析系

1 关于 ParcelJava 对象的生成过程，请参考图 10-16。

更多更新编程资源尽在 http://www.pin5i.com/

统服务的结构，学习如何编写 Java 系统服务。

10.3.1 闹钟服务（Alarm Manager Service）分析

图 10-18 显示了实现闹钟服务（AlarmManagerServic[1]）的类图结构。首先，IInterface 接口与 Binder 类位于最顶层，它们是 Java 服务框架的组成部分，IAlarmManager 服务接口与服务 Stub 继承这两个接口与类；其次位于类图中间的是服务 Stub 类与服务代理类，它们由 AIDL（Android Interface Definition Language）自动生成；最后 Alarm Manager Service 与 AlarmManager 包装类位于类图的最底层，它们最终实现了闹钟服务。

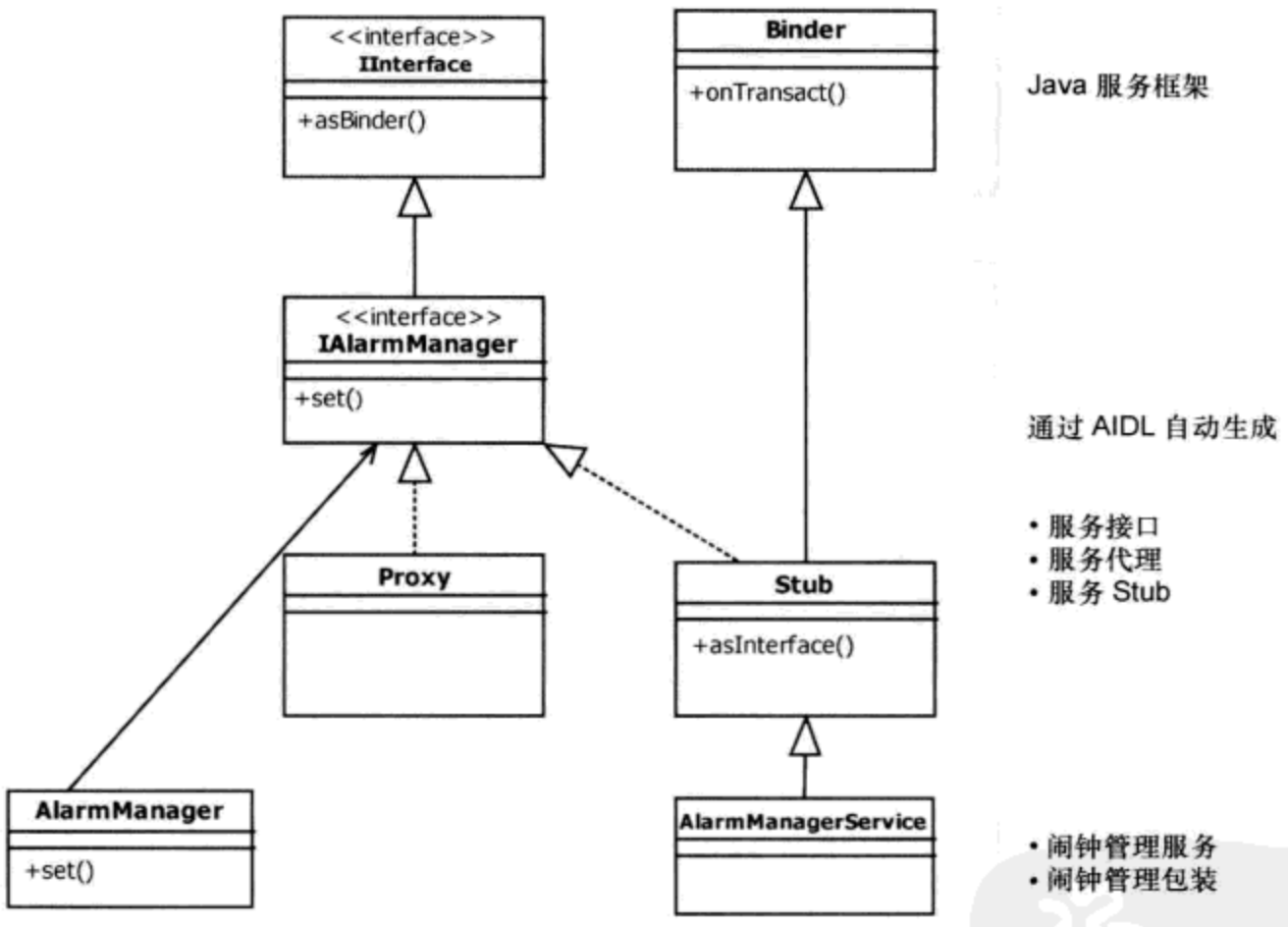


图 10-18 | 实现闹钟服务（AlarmManagerService）的类图

闹钟服务（AlarmManagerService）的实现方式

在编写本地系统服务时，开发者必须亲自实现服务接口、服务代理、服务 Stub、服务。而在编写 Java 系统服务时，开发者可以使用 AIDL 语言自动生成它们。闹钟服务（AlarmManagerService）就使用 AIDL 语言，自动生成了相关的类。

---

1 /frameworks/base/services/java/com/android/server/AlarmManagerService.java

更多更新编程资源尽在：http://www.ibiblio.com/


```
package android.app;                    // IAlarmManager 所属的包
import android.app.PendingIntent; // 导入使用的类
/**
 * System private API for talking with the alarm manager service.
 *
 * {@hide}
 */
interface IAlarmManager {              // 声明下列方法
   void set(int type, long triggerAtTime, in PendingIntent operation);
   void setRepeating(int type, long triggerAtTime, long interval, in PendingIntent
                                                                    operation);
   void setInexactRepeating(int type, long triggerAtTime, long interval, in PendingIntent
                                                                    operation);
   void setTimeZone(String zone);
   void remove(in PendingIntent operation);
}
```

代码 10-16 | IAlarmManager.aidl[1]

代码 10-16 是闹钟服务（AlarmManagerService）的 AIDL 源代码[2]，在 IAlarmManager 接口中共声明了 5 个方法。代码 10-16 是使用 AIDL 编译器自动为闹钟服务（AlarmManagerService）生成的服务接口、服务代理、服务 Stub 类。

TIP 注释中@hide 的含义

在代码 10-16 IAlarmManager.aidl 文件的注释中有两个说明，一个是指出提供闹钟服务的方法是系统私有的 API，另一个则是@hide 标记，查看一下 Android 源码，你会发现一些变量、函数、类等都被@hide 属性标记了，在编译 Android 系统时，它们不会被包含在 SDK 中，即它们不能经由 SDK 访问。

```
public static abstract class Stub extends android.os.Binder implements
      ➥ android.app.IAlarmManager {
   private static final java.lang.String DESCRIPTOR = "android.app.IAlarmManager";

   public static android.app.IAlarmManager asInterface(
      ➥ android.os.IBinder obj) {
      if ((obj == null)) {
         return null;
      }
      android.os.IInterface iin = (android.os.IInterface) obj
            ➥ .queryLocalInterface(DESCRIPTOR);
```

1 /frameworks/base/core/java/android/app/IAlarmManager.aidl

2 关于 AIDL 的内容，将在 10.5 节中详细说明。在此，只讲解与生成闹钟服务的服务 Stub 及服务代理类相关的内容。

更多更新编程资源尽在 http://www.pin5i.com/

```
            if (((iin != null) && (iin instanceof android.app.IAlarmManager))) {
                return ((android.app.IAlarmManager) iin);
            }
            return new android.app.IAlarmManager.Stub.Proxy(obj);
        }

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

        public boolean onTransact(int code, android.os.Parcel data,
                android.os.Parcel reply, int flags){
            switch (code) {
                case TRANSACTION_set: {
                    this.set(_arg0, _arg1, _arg2);
                    return true;
                }
            }
        }
    }
```

代码 10-17 | IAlarmManager.aidl 生成的服务 Stub 的主要源代码

代码 10-17 是服务 Stub 类的主要代码，它是由 AIDL 自动生成的。服务 Stub 类应当重新定义了 Binder 类的 onTransact()方法，并添加系统 Binder RPC 功能。因此，AlarmManagerService 的服务 Stub 类重定义了 onTransact()方法，并且添加了与 IAlarmManager 接口中 5 个方法相关的代码。

例如，在 onTransact()方法中，若接收到的 RPC 代码为 TRANSACTION_set，则调用 set()函数。Java 服务 Stub 与本地服务 Stub 的不同之处在于 asInterface()、asBinder()两个类型转换方法[1]都包含在服务 Stub 之中。

接下来，分析 AlarmManagerService 类的主要代码，如代码 10-18 所示。AlarmManagerService 类位于类图的最底层，它继承了 IAlarmManager.Stub 服务 Stub 类，实现了闹钟服务功能。如代码 10-18 所示，AlarmManagerService 类重新定义了服务接口的 set()方法，具体实现了闹钟功能。

```
class AlarmManagerService extends IAlarmManager.Stub {
    private final Context mContext;

    public AlarmManagerService(Context context) {
        mContext = context;
    }

    public void set(int type, long triggerAtTime, PendingIntent operation) {
```

1 asInterface()与 asBinder()两个类型转换方法，在第 8 章中已作出了说明。

```
            setRepeating(type, triggerAtTime, 0, operation);
        }
    }
```

代码 10-18 | AlarmManagerService.java-AlarmManagerService 类的主要代码

使用闹钟服务（AlarmManagerService）

应用程序开发者若要使用系统服务，需要调用 SDK 的 getSystemService()方法。因此在使用闹钟服务（AlarmManagerService）时，必须调用 getSystemService()方法，并且将 Context[1]类的 ALARM_SERVICE 变量[2]作为参数传入 getSystemService()方法中。获取 AlarmManager 具体是使用 ContextImpl 类的 getAlarmManager()方法[3]实现的，该方法如代码 10-19 所示。

```
private AlarmManager getAlarmManager() {
    synchronized (sSync) {
        if (sAlarmManager == null) {
            IBinder b = ServiceManager.getService(ALARM_SERVICE);          ←❶
            IAlarmManager service = IAlarmManager.Stub.asInterface(b);     ←❷
            sAlarmManager = new AlarmManager(service);                     ←❸
        }
    }
    return sAlarmManager;
}
```

代码 10-19 | ContextImpl.java[4]-getAlarmManager()方法

❶ 调用 ServiceManager 的 getService()方法，请求 AlarmManagerService 服务，返回一个指向 AlarmManagerService 服务的 BinderProxy 对象。

❷ 调用 IAlarmManager.Stub 服务 Stub 的 asInterface()方法，获取 IAlarmManager.Stub.Proxy 服务代理类的对象。

❸ 创建并返回 AlarmManager 的实例对象。

调用 getSystemService()方法获取 AlarmManager 对象如代码 10-20 所示，它以 Context 的 ALARM_SERVICE 变量为参数，通过强制类型转换，获得 AlarmManager 对象。

---

1 /frameworks/base/core/java/android/content/Context.java

2 ALARM_SERVICE 是 Context 类的成员变量，它持有“alarm”值。

3 AlarmManager 类采用单例模式实现，通过 getAlarmManager()方法只能生成唯一一个实例对象。

4 /frameworks/base/core/java/android/app/ContextImpl.java

更多更新编程资源尽在 http://www.pin5i.com/

```
AlarmManager alarm = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
```

代码 10-20 | 获取 AlarmManager 对象

10.3.2 编写 HelloWorldService 系统服务

设计 HelloWorldService

在上一节中，学习了闹钟服务（AlarmManagerService）的组织结构，了解了如何编写运行在 Android 平台中的系统服务。在这一节我们将参考闹钟服务的结构，设计一个名称为“HelloWorldService”的系统服务，其类图关系如图 10-19 所示。根据类图顺序，使用 AIDL 语言自动生成 HelloWorldService 服务接口、服务代理、服务 Stub 类之后，再实现 HelloWorldService 与 HelloWorldManager 类。

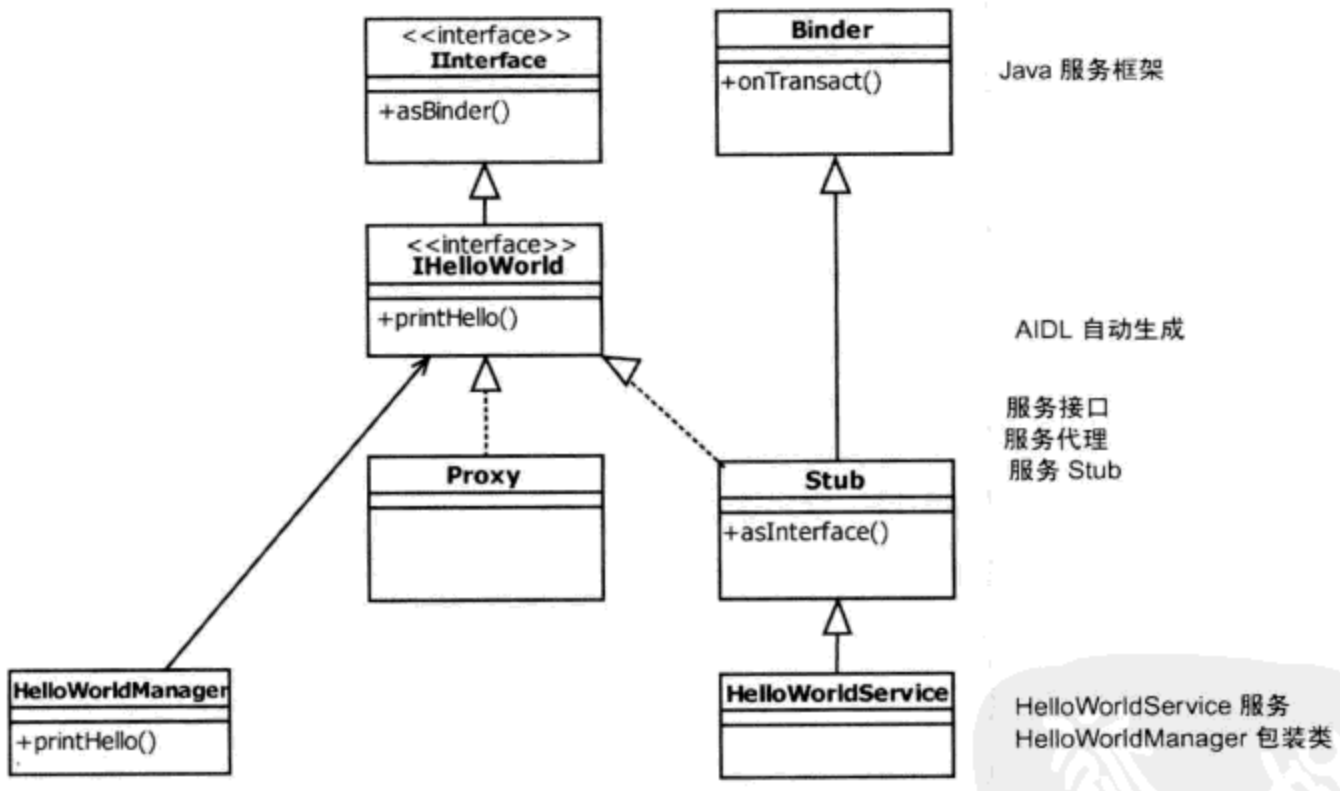


图 10-19 | HelloWorldService 系统服务类图

实现 HelloWorldService

自动生成服务接口、服务代理、服务 Stub

首先使用 AIDL 语言编写 IHelloWorld.aidl 文件，如代码 10-21 所示，在其中声明了 IHelloWorld 接口与 printHello()方法。而后将编写好的 IHelloWorld.aidl 文件移动到

更多更新编程资源尽在 http://www.pin5i.com/

与 IAlarmManager.aidl 文件相同的目录下[1]。

```
package android.app;

/**
 * System private API for talking with the helloworld service.
 *
 * {@hide}
 */

interface IHelloWorld{
  void printHello();
}
```

代码 10-21 | IHelloWorld.aidl-HelloWorld 服务的 AIDL 代码

接着，修改 Android.mk 设置文件[2]，以便 IHelloWorld.aidl 文件顺利通过 AIDL 编译器的编译。文件修改非常简单，只需要在 LOCAL_SRC_FILES 条目下，添加上 Ihello World.aidl 即可，如代码 10-22 所示。若在 aidl_files 条目下也添加了 IHelloWorld.aidl 文件，则会被包含到 SDK 中。

```
## READ ME: ######################################################
##
## When updating this list of aidl files, consider if that aidl is
## part of the SDK API.  If it is, also add it to the list below that
## is preprocessed and distributed with the SDK.
##
## READ ME: ######################################################
LOCAL_SRC_FILES += \
   ...
   core/java/android/app/IAlarmManager.aidl \
   core/java/android/app/IHelloWorld.aidl \
   ...

# AIDL files to be preprocessed and included in the SDK,
# relative to the root of the build tree.
# ============================================================
aidl_files := \          ←❶
   frameworks/base/core/java/android/accounts/IAccountManager.aidl \
```

代码 10-22 | frameworks/base/Android.mk 文件的主要内容

实现 HelloWorldService 服务

HelloWorldService 要继承并实现服务 Stub 类。代码 10-23 是 HelloWorldService 类

1　/frameworks/base/core/java/android/app/IAlarmManager.aidl

2　/frameworks/base/Android.mk

更多更新编程资源尽在 http://www.pin5i.com/

的源码，它继承并实现了 IHelloWorld.Stub 服务 Stub 类，IHelloWorld.Stub 服务 Stub 类是由 AIDL 编译器自动生成的。HelloWorldService 类中的 printHello()方法非常简单，它只是向日志中输出一个字符串“Hello，World”。

```
package com.android.server;

import android.app.IHelloWorld;
import android.content.Context;
import android.os.RemoteException;
import android.util.Slog;


class HelloWorldService extends IHelloWorld.Stub {

  private static final String TAG = "HelloWorldService";
  Context mContext;
  public HelloWorldService(Context context){
      mContext = context;
  }

  public void printHello() throws RemoteException {
      Slog.i(TAG, "Hello,World!");
  }
}
```

代码 10-23 | HelloWorldService.java[1]-HelloWorldService 类

注册 HelloWorldService 服务

Java 系统服务在 ServerThread 类的 run()方法中生成并被注册到 Android 平台中。同样，HelloWorldService 系统服务也是在 ServerThread 类的 run()方法中生成 HelloWorldService 实例对象，并通过 ServiceManager 的 addService()方法将其注册到系统之中，如代码 10-24 所示。

```
try {
   Slog.i(TAG, "HelloWorld Service");
   helloWorldService = new HelloWorldService(context);
   ServiceManager.addService(Context.HELLO_SERVICE, helloWorldService);
} catch (Throwable e) {
   Slog.e(TAG, "Failure starting HelloWorld Service", e);
}
```

代码 10-24 | SystemServer.java[2]-在 ServerThread 的 run()方法中注册 HelloWorldService 服务

1 /frameworks/base/services/java/com/android/server/HelloWorldService.java

2 /frameworks/base/services/java/com/android/server/SystemServer.java

更多更新编程资源尽在 http://www.chinaitpub.com/

Context 类的 HELLO_SERVICE 是一个字符串对象，它保存着字符串“helloworld”，如代码 10-25 所示。

```
/**
     * Use with {@link #getSystemService} to retrieve a
     * {@blink android.app.services} for accessing HelloWorld Service.
     *
     * @hide
     * @see #getSystemService
*/
public static final String HELLO_SERVICE = "helloworld";
```

代码 10-25 | Context.java[1]-Context 类中定义的 HELLO_SERVICE

10.3.3 使用 HelloWorldService 系统服务

在上一节 10.3.2 中，已经实现了 HelloWorldService 系统服务，并且将其注册到系统中。在本节中，将编写一个名称为 HelloWorldManager 的类，它使得应用程序开发者可以使用注册在系统中的 HelloWorldService 系统服务。

编写 HelloWorldManager 类

参考 AlarmManager 包装类，编写 HelloWorldManager 类，它是一个包装类，如代码 10-26 所示。

```
package android.app;

import android.os.RemoteException;

public class HelloWorldManager
{
   private final IHelloWorld mService;

   /**
   * package private on purpose
   */
   HelloWorldManager(IHelloWorld service) {          ←❶
      mService = service;
   }

   public void printHello() {                        ←❷
      try {
         mService.printHello();
```

1 /frameworks/base/core/java/android/content/Context.java

更多更新编程资源尽在 http://www.pin5i.com/

```
            } catch (RemoteException ex) {
            }
        }
    }
```

代码 10-26 | HelloWorldManager.java[1]-HelloWorldManager 类

❶ HelloWorldManager 类的构造方法接收 IHelloWorld.Stub.Proxy 类的实例对象，并将其保存到私有的 IHelloWorld 类型变量中。注意，HelloWorldManager 类的构造方法默认访问权限修饰符为 default，这意味着只有同一个包中的类才能调用它，即只有 android.app 包中的类才能创建 HelloWorldManager 类的实例对象，请使用 SDK 开发的朋友们注意这一点。

❷ 调用 HelloWorldManager 的 printHello()方法，将引起 IHelloWorld.Stub.Proxy 的 printHello()方法调用，该方法可能会抛出 RemoteException[2]异常，调用时要包裹在 try/catch 语句中。

获取 HelloWorldManager

应用程序开发者可以通过 getSystemService()方法来使用系统服务。ContextImpl 类的 getSystemService()方法接收一个字符串参数，若字符串与 HELLO_SERVICE 变量[3]中的字符串相同，则调用 getHelloWorldManager()方法，获取并返回 HelloWorldManager 对象，如代码 10-27❶所示。

```
public Object getSystemService(String name) {
    if (WINDOW_SERVICE.equals(name)) {
        return WindowManagerImpl.getDefault();
    } else if (ALARM_SERVICE.equals(name))
        return getAlarmManager();
    } else if (HELLO_SERVICE.equals(name)) {
        return getHelloWorldManager();          ←❶
    }
    return null;
}
```

代码 10-27

代码 10-28 是 getHelloWorldManager()方法的代码，与 getAlarmManager()方法的内容相同，在此不再赘述。

1 /frameworks/base/core/java/android/app/HelloWorldManager.java

2 RemoteException 是 Binder RPC 过程中产生的异常。迄今为止，Android 平台中定义的 RemoteException 仅有一个 DeadObjectException，当所调用的服务不存在时就会发生该异常。

3 HELLO_SERVICE 是 Context 类的一个成员变量，它保存着字符串“helloworld”。

更多更新编程资源尽在 http://www.pin5i.com/


```
private HelloWorldManager getHelloWorldManager() {
   synchronized (sSync) {
      if (sHelloWorldManager == null) {
         IBinder b = ServiceManager.getService(HELLO_SERVICE);          ❶
         IHelloWorld service = IHelloWorld.Stub.asInterface(b);         ❷
         sHelloWorldManager= new HelloWorldManager (service);           ❸
      }
   }
   return sHelloWorldManager;
}
```

代码 10-28 | ContextImpl.java-getHelloWorldManager()方法

请看图 10-20，它描述了 HelloWorldService 服务中获取 HelloWorldManager 对象的整个过程。

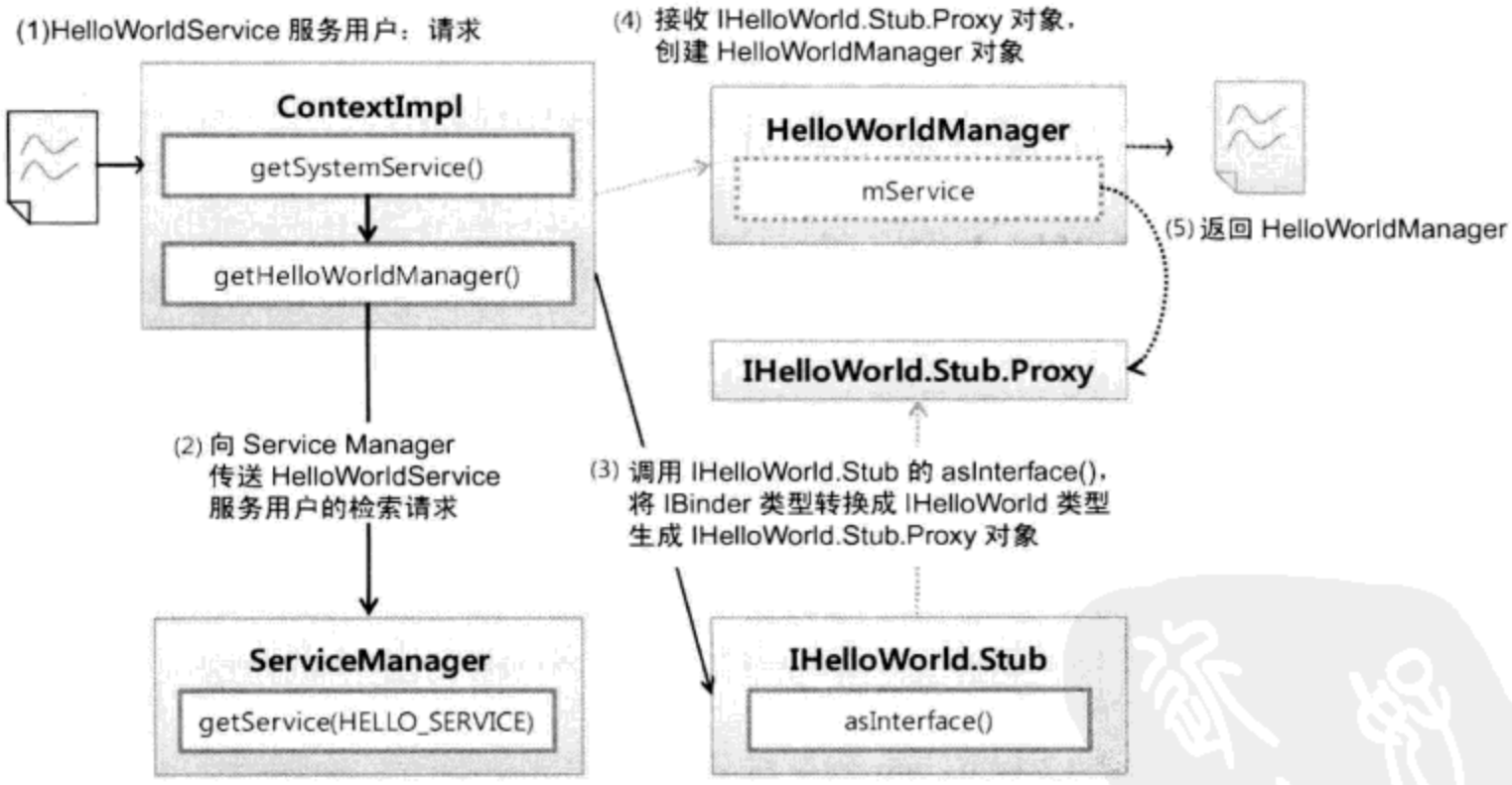


图 10-20 | 获取 HelloWorldManager

（1） 为了使用 HelloWorldService 服务，服务用户调用 getSystemService()方法，在 getSystemService()方法内部转调 getHelloWorldManager()方法。

（2） getHelloWorldManager()方法向 ServiceManager 请求检索 HelloWorldService 服务。若 ServiceManager 检索到 HelloWorldService 服务，则返回一个指向它的 BinderProxy 对象。

（3） getHelloWorldManager()方法将接收到的 BinderProxy 对象传递给 HelloWorld.Stub 服务 Stub 的 asInterface()方法，生成 IHelloWorld.Stub.Proxy 服务代理对象。

更多更新编程资源尽在 http://www.pin5i.com/

（4） getHelloWorldManager()方法将生成的 IHelloWorld.Stub.Proxy 服务代理对象传递给 HelloWorldManager 的构造方法，生成 HelloWorldManager 对象，并且其 mService 变量中保存着 IHelloWorld.Stub.Proxy 服务代理对象的引用。

（5） 最后，在为使用 HelloWorldService 服务而调用 getSystemService()方法的位置上，获取返回的 HelloWorldManager 对象。

为了使用 HelloWorldService 服务，应用程序调用 HelloWorldManager 的方法，进而调用 IHelloWorld.Stub.Proxy 服务代理的方法，使用 Binder RPC 与 HelloWorldService 服务进行交互。

最后，为了使用 HelloWorldService 服务，如代码 10-29 所示，应用程序将调用 getSystemService()方法，把 Context.HELLO_SERVICE 传入方法中，然后将返回值转换为 HelloWorldManager 类型，这样就可以使用 HelloWorldService 服务了。

```
HelloWorldManager hello = (HelloWorldManager)
        ➥ getSystemService(Context.HELLO_SERVICE);
```

代码 10-29 | 应用程序获取 HelloWorldManager

10.3.4 编译 HelloWorldService 系统服务

HelloWorldService 系统服务编写完成后，需要使用 make 命令，将其编译到 Android 平台中，如下所示。

```
$ make
```

编译时若 SDK 被改动，则会显出如图 10-21 所示的警告信息。由于新添加的 HelloWorld Manager 类改动了 SDK API，所以警告信息通知我们通过添加“@hide”注释将相应的方法注释掉或使用 make update-api 命令更新 current.xml[1]文件。

```
******************************
You have tried to change the API from what has been previously approved.

To make these errors go away, you have two choices:
   1) You can add "@hide" javadoc comments to the methods, etc. listed in the
      errors above.

   2) You can update current.xml by executing the following command:
         make update-api

      To submit the revised current.xml to the main Android repository,
      you will need approval.
******************************
```

图 10-21 | SDK API 发生改动时出现的警告信息

1 /frameworks/base/api/current.xml，该文件包含 Android SDK 中所有公开 API 的相关信息

更多更新编程资源尽在 http://www.pin5i.com/

或者向 HelloWorldManager 类添加“@hide”注释，或者使用 make update-api 命令更新 SDK 的公开 API 信息，请根据开发环境选择其一。在这里，我们选择第二种方法，即使用 make update-api 命令更新 SDK 的公开 API 信息。更新完成后，在 current.xml 文件中即可看到被改动的 API 信息，如代码 10-30 所示。

```
<class name="HelloWorldManager"
   extends="java.lang.Object"
   abstract="false"
   static="false"
   final="false"
   deprecated="not deprecated"
   visibility="public"
>
<method name="printHello"
   return="void"
   abstract="false"
   native="false"
   synchronized="false"
   static="false"
   final="false"
   deprecated="not deprecated"
   visibility="public"
>
</method>
</class>
```

代码 10-30 | HelloWorldManager 类被添加到 current.xml[1]文件中

编译顺利完成后，将生成 system.img[2]文件。我们将生成的 system.img 文件复制到 <ANDROID_SDK>[3]/platforms/android-8/images 目录下，运行模拟器，编写好经由 HelloWorldManager 使用 HelloWorldService 的应用程序后，调用 printHello()方法，即会看到“Hello，World!”日志信息，如图 10-22 所示。

	pid	tag	Message
D	181	dalvikvm	GC freed 97 objects / 3832 bytes in 21
I	96	ActivityManager	Starting activity: Intent { act=andro
I	96	SystemServer	Hello,World!
I	96	ActivityManager	Displayed activity edu.suwon.services/

图 10-22 | ServerThread 类生成 HelloWorldService

对于 HelloWorldService 系统服务是否已被正常注册到 Android 平台中，并且运行是否正常，使用 adb shell 命令来进行确认。输入 adb shell 命令，出现 shell prompt 后，

1 /frameworks/base/api/current.xml

2 30 /out/target/product/generic/system.img

3 Android SDK 的安装目录

更多更新编程资源尽在 http://www.pin5i.com/

再输入 service list 命令，就会显示出已经注册在系统中的 HelloWorldService 服务接口的名称，如图 10-23 所示。

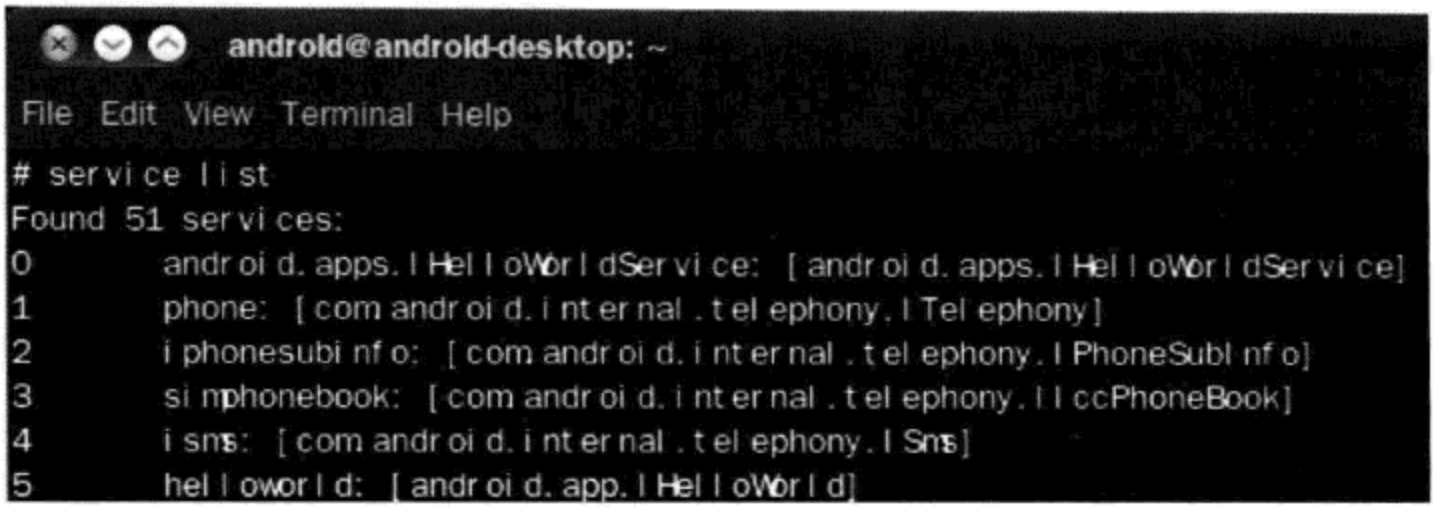


图 10-23 | HelloWorldService 服务注册信息

10.4 Java Service Manager

在 Android 平台中，Java 系统服务与本地系统服务一样都是由 Context Manager 进行管理的。通常使用 Java 服务框架编写的系统服务都是通过 Java Service Manager 注册到系统中的，本节将讲解有关 Java Service Manager 的内容。

10.4.1 Java Service Manager 简介

如图 10-24 所示，Java Service Manager 由服务层、RPC 层、IPC 层组成。IServiceManager 服务接口与 ServiceManager 包装类位于服务层中，它们由 Service Manager 用户使用。Service ManagerProxy 服务代理类位于 RPC 层，它将方法的调用信息转换为 Binder RPC 数据。Binder Proxy 类位于 IPC 层，它通过 JNI 与 BpBinder 类连接在一起。在图中可以看到 Java Service Manager 有相应的服务代理，但并未发现服务 Stub，这是由于服务 Stub 与服务并非是使用 Java 服务框架编写的类，因而未将其标注在图中。

图 10-24 | Java Service Manager 的各个层

与 Native Service Manager 一样，Java Service Manager 也提供服务注册、服务检索、服务确认以及服务目录功能。由于 Java Service Manager 的功能与 Native Service Manager 提供的大部分

更多更新编程资源尽在 http://www.bi5ai.com/

功能类似，本节将不再对这些功能另作说明，我们将讲解的重点放在 Java Service Manager 如何通过 JNI 连接本地服务框架上。

TIP 关于 Java Service Manager

一般地，Android 系统在运行应用程序时都会生成新的进程。当新生成的进程中有使用服务的请求时，就会使用 ServiceManager[Java] 向 Context Manager 请求检索指定的服务。在使用 ActivityManagerService 系统服务时，每当进程生成时都会通过 ServiceManager[Java] 反复检索使用 ActivityManagerService 的用户。

ActivityManagerService 服务通常运行在 system_server 进程中，即使仅有一个服务用户，也可以将其共享给各个进程，以供它们使用。这就是说使用 ActivityManagerService 服务时，不必每次都通过 ServiceManager[Java] 检索它。在系统运行期间，将被请求的服务列表保存到缓存中，每当新进程生成时，将列表传递给它，这样在使用 ActivityManagerService 时就无须向 ServiceManager[Java] 请求检索了。

为了提升系统速度，Java Service Manager 的 ServiceManager[Java] 类提供了 initServiceCache() 方法。在 Android 平台中，initServiceCache()方法被用于 ActivityThread 类的 bindApplication() 方法中，服务列表以参数的形式被传入 initServiceCache()方法中，并被保存到 ServiceManager[Java] 的缓存中。

10.4.2 BinderInternal

BinderInternal 类提供获取 BpBinder 对象的功能，它指向 Context Manager。

BinderInternal 的 JNI 设定

在使用 BinderInternal 类之前，需要先将与其本地方法对应的 JNI 本地函数注册到 Dalvik 虚拟机中。如代码 10-1，当调用 int_register_android_os_BinderInternal()函数时，BinderInternal 的部分信息就会被保存到 gBinderInternalOffsets 全局变量中，并完成 BinderInternal 本地方法与 JNI 本地函数的映射。图 10-25 与图 10-26 反映了 BinderInternal 的 JNI 设定完成之后的情况。

图 10-25 | 保存 BinderInternal 类信息的 binderinternal_offsets_t 结构体

更多更新编程资源尽在 http://www.pin5i.com/

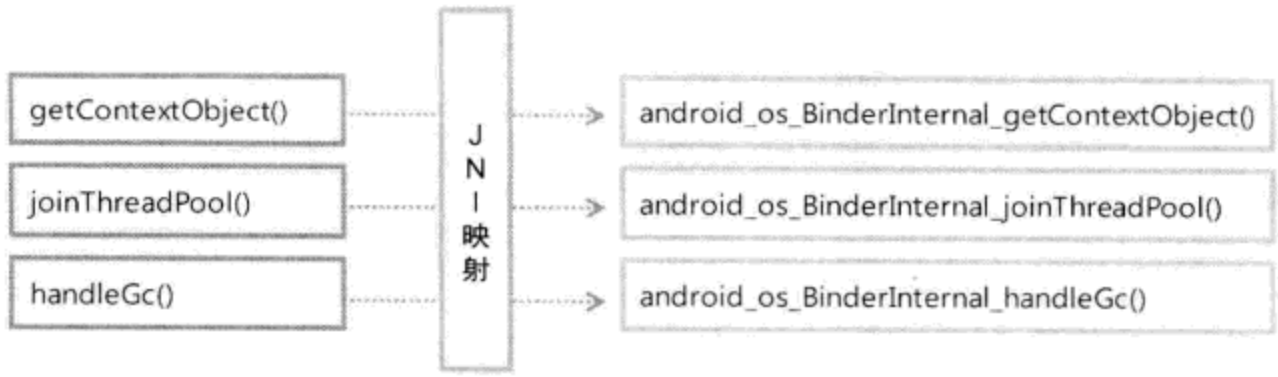


图 10-26 | BinderInternal 本地方法与 JNI 函数映射关系

生成 BinderInternal 对象[1]

BinderInternal 类的成员变量与方法都是静态的（static），使用时不必使用 new 运算符创建该类的对象，直接使用其 public 成员变量、getContextObject()、joinThreadPool()等方法即可。

ServiceManager[Java] 类与 BinderInternal 类的相互作用

若想在 Java Service Manager 所在的进程与 Context Manager 之间进行 Binder IPC，需要首先获取一个指向 Context Manager 的 BpBinder 对象。在 Native Service Manager 中使用 ProcessState 的 getContextObject()函数获取 BpBinder 对象，而在 Java Service Manager 中则使用 BinderInternal 的 getContextObject()方法来获取它。

BinderInternal 的 getContextObject()方法是一个本地方法，它通过 JNI 与本地函数 android_os_ BinderInternal_getContextObject()连接在一起。代码 10-31 是本地函数 android_os_ BinderInternal_getContextObject()的代码，其内部仍然调用 ProcessState 的 getContextObject()函数，该函数在第 8 章“Android Service Framework”讲解的 default ServiceManager()函数中也以相同的方式被调用了。也就是说，在获取指向 Context Manager 的 BpBinder 对象时，无论是 BpServiceManager 还是 ServiceManager[Java]，它们采用的方式是相同的。接着，函数调用 javaObjectForBinder()[2]函数生成 BinderProxy 对象，并将其返回。

```
static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject clazz)
{
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
    return javaObjectForIBinder(env, b);
}
```

代码 10-31 | android_os_BinderInternal_getContextObject()函数

1 执行 ZygoteInit 类时将调用 preloadClasses()方法，预加载 Android 平台所需要的类信息。BinderInternal 类被 Class 类的 forName()方法动态加载。在使用 forName()方法加载类时，不仅类的相关信息，各类的静态成员变量会被初始化，静态初始化块会被执行，ServiceManager[Java] 类也采用同一过程进行初始化。

2 javaObjectForBinder()函数将在 10.4.3 节中进行讲解，它使用 JNI 函数生成 BinderProxy 对象，并将其 mObject 变量指向 BpBinder 对象。

更多更新编程资源尽在 http://www.pin5i.com/

图 10-27 描述了 ServiceManager[Java] 使用 BinderInternal 获取指向 Context Manager 的 BpBinder 对象的过程。当在 ServiceManager[Java] 内部调用 getIServiceManager()方法时，Binder Internal 的 getContextObject 本地方法即被调用，进而调用 JNI 本地函数 android_os_BinderInternal_getContextObject()，生成指向 Context Manager 的 BpBinder 对象。而后调用 JNI 本地函数 javaObjectForIBinder()生成持有 BpBinder 对象引用的 BinderProxy 实例对象。

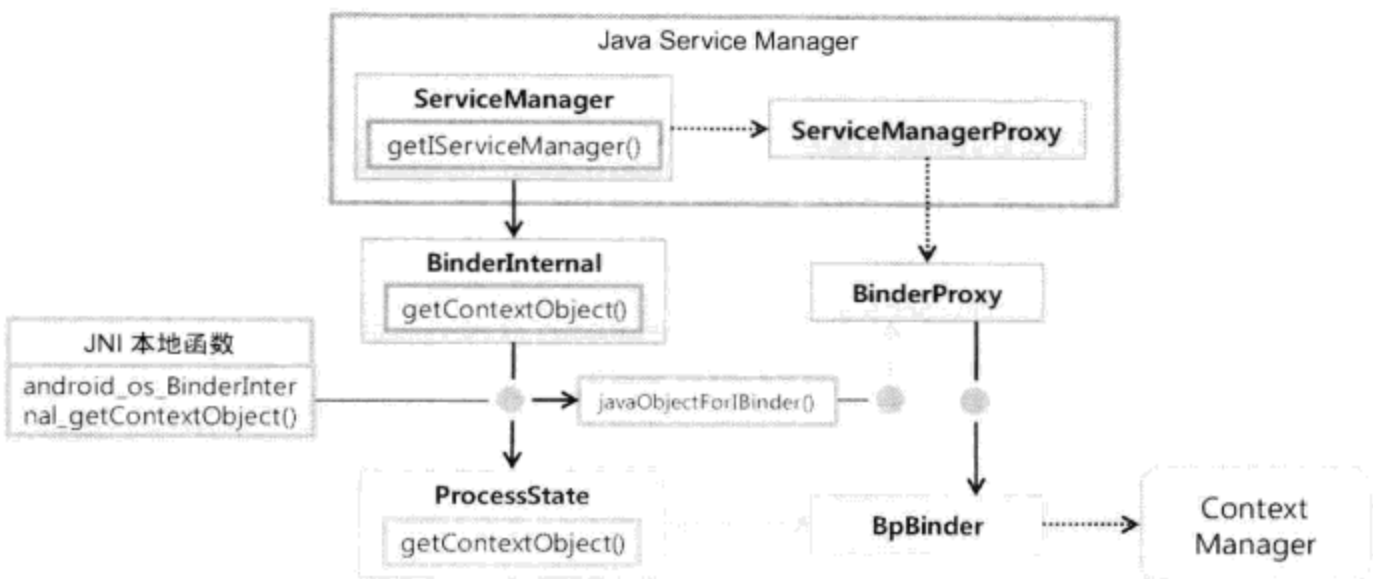


图 10-27 | 获取持有 Context Manager 的 BpBinder 对象

10.4.3 Java Service Manager 的运行实例

下面以闹钟服务注册与使用过程为例来学习 Java Service Manager 注册检索服务的过程。

服务注册

为了向系统注册 AlarmManagerService，首先创建 AlarmManagerService 实例对象，而后调用 ServiceManager[Java] 的 addService()方法，传入 AlarmManagerService 的名称与对象，向系统注册服务，如代码 10-32 所示。

```
AlarmManagerService alarm = new AlarmManagerService(context);
ServiceManager.addService(Context.ALARM_SERVICE, alarm);
```

代码 10-32 | SystemServer.java-注册 AlarmManagerService

接下来，分析 addService()方法代码，了解注册 Java 系统服务的方法。

初始化 Java Service Manager

ServiceManager[Java] 的 addService()方法如代码 10-33 所示，在其内部它又调用了 getIService Manager().addService()方法，如代码 10-33❶所示。

更多更新编程资源尽在 http://www.pin5i.com/

```
public static void addService(String name, IBinder service) {
      try {
         getIServiceManager().addService(name, service);      ←❶
      } catch (RemoteException e) {
         Log.e(TAG, "error in addService", e);
      }
}
```

代码 10-33 | ServiceManager.java[1]-addService()方法

请继续看代码 10-34，它是 getIServiceManager()方法的代码。由代码可以看到，getIService Manager()方法首先调用了 BinderInternal 的 getContextObject()方法，如前所述，调用该方法将会返回一个指向 Context Manager 的 BpBinder 实例对象。

```
private static IServiceManager getIServiceManager() {

   // Find the service manager
   sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
   return sServiceManager;
}
```

代码 10-34 | ServiceManager.java-getIServiceManager()方法

在获取指向 Context Manager 的 BpBinder 实例对象之后，它被作为参数传入 Service Manager Native 的 asInterface()方法中。请看代码 10-35，它是 asInterface()方法的代码，将生成 Service ManagerProxy 的实例对象，并将其返回。

```
static public IServiceManager asInterface(IBinder obj)
{
   if (obj == null) {
      return null;
   }
   IServiceManager in =
      (IServiceManager)obj.queryLocalInterface(descriptor);
   if (in != null) {
      return in;
   }

   return new ServiceManagerProxy(obj);
}
```

代码 10-35 | ServiceManagerNative.java[2]-asInterface()方法

1 /frameworks/base/core/java/android/os/ServiceManager.java

2 /frameworks/base/core/java/android/os/ServiceManagerNative.java

更多更新编程资源尽在 http://www.pin5i.com/

当调用 ServiceManager[Java] 的 addService()方法准备完成后，就会调用 ServiceManager Proxy 对象（该对象由 getIServiceManager()方法返回）的 addService()方法，如代码 10-33❶所示。ServiceManager[Java] 的整个初始化过程如图 10-28 所示。

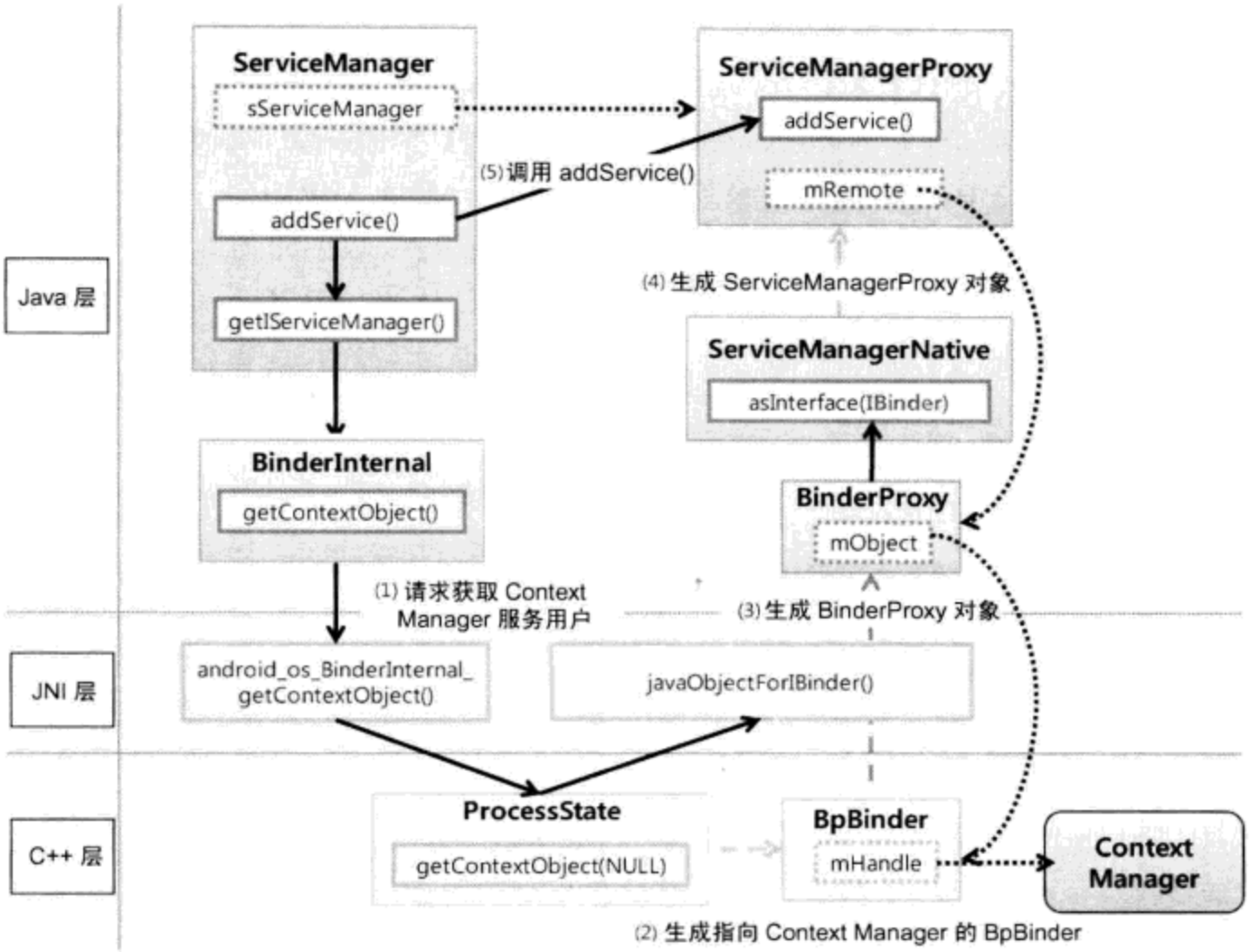


图 10-28 | ServiceManager.java-初始化 ServiceManager[Java]

（1） 调用 ServiceManager[Java] 的 addService()方法，在其内部 getIServiceManager()方法将被调用。在 getIServiceManager()方法中调用 BinderInternal 类的 getContexObject()方法，getContextObject()方法是本地方法，它通过 JNI 调用本地函数 android_os_BinderInternal_ getContextObject()。

A．在 android_os_BinderInternal_getContextObject()函数中，调用 ProcessState 的 getContextObject()成员函数，获取一个指向 Context Manager 的 BpBinder 对象。

B．javaObjectForI Binder()函数生成 BinderProxy 对象，其 mObject 变量持有 BpBinder 对象的引用。并且，生成的 BinderProxy 对象将作为参数传递给 ServiceManagerNative 的 asInterface()方法。

（2） 调用 ServiceManagerNative 的 asInterface()方法，生成 ServiceManagerProxy 类的实例对象。

（3） 最后，调用 ServiceManagerProxy 对象的 addService()方法注册服务。

更多更新编程资源尽在 http://www.pin5i.com/

调用 addService()方法注册服务

如前所言，调用 ServiceManager[Java] 的 addService()方法最终会引起 ServiceManagerProxy 的 addService()方法的调用。下面来分析 ServiceManagerProxy 的 addService()方法代码，主要如代码 10-36 所示。

该方法首先生成 RPC 数据，这些数据是要被传递给 Context Manager 的。在 Parcel[Java] 类型的 data 变量中依次保存着 IServiceManager 的描述符（“android.os.IServiceManager”）、系统服务名称，以及 Binder 类型的服务实例。而后将 ADD_SERVICE_TRANSACTION RPC 代码（它表示调用 addService()方法）与 data 中的 RPC 数据作为参数传递给 Binder Proxy 的 transact()方法。然后通过 Binder IPC 将 RPC 代码与 RPC 数据传递给 Context Manager，把相关服务注册为系统服务。

```
public void addService(String name, IBinder service)
    throws RemoteException {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IServiceManager.descriptor);
    data.writeString(name);
    data.writeStrongBinder(service);                    ❶
    mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
    reply.recycle();
    data.recycle();
}
```

代码 10-36 | ServiceManagerProxy 的 addService()方法

如代码 10-36❶，在进行 Binder IPC 时，调用 Parcel[Java] 的 writeStrongBinder()方法，将 Binder 对象 BinderProxy 与 Binder 对象转换为 BpBinder 与 JavaBBinder 对象。Write Strong Binder()是一个本地方法，它通过 JNI 调用本地函数 android_os_Parcel_writeStrong Binder()函数，如代码 10-37 所示。

```
static void android_os_Parcel_writeStrongBinder(JNIEnv* env, jobject clazz, jobject object)
{
    Parcel* parcel = parcelForJavaObject(env, clazz);          ❶
    if (parcel != NULL) {
        const status_t err =                                   ❷
            parcel->writeStrongBinder(ibinderForJavaObject(env, object));
        if (err != NO_ERROR) {
            jniThrowException(env, "java/lang/OutOfMemoryError", NULL);
        }
    }
}
```

代码 10-37 | android_os_Parcel_writeStrongBinder()函数的主要代码

更多更新编程资源尽在：http://www.pin5i.com/

下面简单地分析一下android_os_Parcel_writeStrongBinder()函数的主要部分。

❶ 调用parcelForJavaObject()函数[1]，获取Parcel[Java]的mObject变量所引用的Parcel[C++]对象的地址。

❷ 调用ibinderForJavaObject()函数，将Binder与BinderProxy对象分别转换为JavaB Binder与BpBinder对象。

接着，分析ibinderForJavaObject()函数，其主要如代码10-38所示。在代码10-38❶中，首先获取Binder的mObject变量中保存的JavaBBinderHolder对象，而后调用get()函数[2]，生成JavaBBinder类的实例，并将其返回。

```
sp<IBinder> ibinderForJavaObject(JNIEnv* env, jobject obj)
{
    if (obj == NULL) return NULL;
    if (env->IsInstanceOf(obj, gBinderOffsets.mClass)) {          ← ❶
        JavaBBinderHolder* jbh = (JavaBBinderHolder*)
            ➥ env->GetIntField(obj, gBinderOffsets.mObject);
        return jbh != NULL ? jbh->get(env) : NULL;
    }
    if (env->IsInstanceOf(obj, gBinderProxyOffsets.mClass)) {     ← ❷
        return (IBinder*)
            env->GetIntField(obj, gBinderProxyOffsets.mObject);
    }
    return NULL;
}
```

代码10-38 | ibinderForJavaObject()函数的主要代码

在代码10-38❷中，获取BinderProxy的mObject变量中保存的BpBinder对象的地址，并将其返回。

使用服务

在使用闹钟服务时，需要调用ServiceManager[Java]的getService()方法，并传入Alarm Manager Service的名称，如代码10-39所示。

```
IBinder b = ServiceManager.getService(ALARM_SERVICE);
IAlarmManager service = IAlarmManager.Stub.asInterface(b);
sAlarmManager = new AlarmManager(service);
```

代码10-39 | ApplicationContext.java-检索Alarm Service Manager

1 请参考10.2.4一节中有关ParcelJava的部分。

2 请参考10.2.2一节中有关Binder的部分。

更多更新编程资源尽在 http://www.pin5i.com/

调用 ServiceManager[Java] 的 getService()方法时，就会在 Binder 对象[1]的缓存中查找 Alarm Manager Service 的 Binder 对象。若查找不到，则会调用 ServiceManagerProxy（调用 getIServiceManager()方法[2]获取）的 getService()方法，如代码 10-40❶所示。我们假定 Alarm Manager Service 仍未被注册，继续分析 getService()方法。

```
public static IBinder getService(String name) {
   try {
      IBinder service = sCache.get(name);
      if (service != null) {
         return service;
      } else {
         return getIServiceManager().getService(name);          ❶
      }
   } catch (RemoteException e) {
      Log.e(TAG, "error in getService", e);
   }
   return null;
}
```

代码 10-40 | ServiceManager.java-getService()方法

ServiceManagerProxy 的 getService()方法如代码 10-41 所示。首先在 Parcel[Java] 类型的 data 变量中依次保存 IServiceManager 的描述符（“android.os.IServiceManager”）、系统服务名称。而后将 GET_SERVICE_TRANSACTION RPC 代码（它表示调用 getService()方法）与 Parcel[Java] 对象作为参数传递给 BinderProxy 的 transact()方法。然后通过 Binder IPC 将 RPC 数据传递给 Context Manager，以 Binder 对象形式返回相关服务。

```
public IBinder getService(String name) throws RemoteException {
   Parcel data = Parcel.obtain();
   Parcel reply = Parcel.obtain();
   data.writeInterfaceToken(IServiceManager.descriptor);
   data.writeString(name);
   mRemote.transact(GET_SERVICE_TRANSACTION, data, reply, 0);
   IBinder binder = reply.readStrongBinder();                   ❶
   reply.recycle();
   data.recycle();
   return binder;
}
```

代码 10-41 | ServiceManagerProxy 的 getService()方法

如代码 10-41❶所示，在进行 Binder IPC 时，调用 Parcel[Java] 的 readStrongBinder()

1 指 IBinder 类型的对象。

2 请参考 10.4.3.1 一节服务注册中的相关部分。

更多更新编程资源尽在 http://www.pin5i.com/

方法，而非 Parcel[Java] 的 writeStrongBinder()方法，将 BpBinder 与 JavaBBinder 对象转换为 BinderProxy 与 Binder 对象。readStrongBinder()是一个本地方法，它通过 JNI 与本地函数 android_os_Parcel_readStrongBinder()连接在一起。代码 10-42 是本地函数 android_os_Parcel_readStrongBinder()的主要代码。它通过调用 parcelForJavaObject()函数，获取 Parcel[Java] 的 mObject 变量所引用的 Parcel[C++]对象的地址。而后调用 javaObject ForIBinder()函数[1]，分别将 JavaBBinder 与 BpBinder 对象转换成 Binder 与 BinderProxy 对象。

```
static jobject android_os_Parcel_readStrongBinder(JNIEnv* env, jobject clazz)
{
    Parcel* parcel = parcelForJavaObject(env, clazz);
    if (parcel != NULL) {
        return javaObjectForIBinder(env, parcel->readStrongBinder());
    }
    return NULL;
}
```

代码 10-42 | android_os_Parcel_readStrongBinder()的主要代码

10.5 使用 AIDL 生成服务代理与服务 Stub

AIDL（Android Interface Definition Language）是一种接口定义语言，用于生成可以在 Android 设备上两个进程之间进行进程间通信（IPC）的代码。Android Java Service Framework 提供的大多数系统服务都是使用 AIDL 语言生成的。使用 AIDL 语言，可以自动生成服务接口、服务代理、服务 Stub 代码。

以 10.3 节中编写的 HelloWorldService 为例，服务接口要定义一些函数，以使服务代理与服务拥有相同的接口，并且要在服务接口与 IBinder 间实现类型转换功能。还有，服务代理必须为 Binder RPC 生成 Binder RPC 代码与数据。特别地，在编写运行在不同进程中的服务时，对传递的数据进行 Marshalling 处理时经常产生许多重复性的代码。在这种情形下，使用 AIDL 语言描述服务接口，从而自动生成服务接口、服务代理、服务 Stub 代码。

在 Android 开发者网站上，指出了使用 AIDL 实现 IPC 服务的步骤，有如下四步：

1. 使用 AIDL 语法，创建.aidl 文件。该.aidl 文件定义了服务接口的方法与属性。
2. 在 makefile 文件中加入.aidl 文件。AIDL 编译器[2]将自动生成服务接口、服务

1 请参考 10.2.3 一节有关 BinderProxy 的内容。

2 在/frameworks/base/tools/aidl 文件夹中，可以查看 aidl 编译器源码。

更多更新编程资源尽在 http://www.pin5i.com/

Stub，以及服务代理类。即在 Android.mk 文件的 LOCAL_SRC_FILES 中加入创建好的.aidl 文件。

3. 创建一个继承 Stub 的类并且实现.aidl 文件中声明的方法。
4. 向服务用户公开接口。

10.5.1 在 AIDL 文件中定义服务接口

如何使用 AIDL 语言编写 aidl 文件[1]呢？代码 10-43 是使用 AIDL 语言编写 aidl 文件的示例。如示例所示，第一句是包声明语句（Package Statement），而后是导入声明语句（import statement）。在 AIDL 中对于非内建（built-in）的数据类型，即使在同一文件夹中，也必需显式地进行导入。AIDL 支持的数据类型有如下几种：

- Java 原生类型（int，double，boolean 等）、String、CharSequence 类：不需要使用 import 语句。
- List、Map 类：容器类（container）中的元素必须是 Java 基本类型、String、CharSequence、AIDL 生成的接口类型或 Parcelable 类型。

此外，在使用实现由 AIDL 生成的接口或 Parcelable 接口的类时也必须进行导入。

接下来是名称为 IMyService 的接口声明语句（interface statement）。与 Java 语言规则相同，接口名称需与 aidl 文件名称保持一致。接口声明语句形式为：interface_ header，识别符（identier），{interface_items}，示例中声明了 test()与 getSize()两个方法。

```
package edu.android.suwon.service;

interface IMyService {
   void test();
   int getSize(in String name);
}
```

代码 10-43 | IMyService.aidl

在 AIDL 语言的接口语法中，有两个 Java 接口语法不具备的特征。第一，interface_ header 能够使用字符串“interface”或“oneway interface”进行声明。关键字 oneway 表示当服务用户请求相应功能时不需要等待应答可以直接调用返回。该关键字可以用于接口声明或方法声明语句中，若接口声明语句中使用了“oneway”关键字，则该接

1 与编写 AIDL 文件相关的详细内容，请参考 Android 开发者网站（http://developer.android.com/guide/ developing/tools/aidl.html）。

更多更新编程资源尽在 http://www.qinsi.com/

口中声明的所有方法都采用了 oneway 方式。

第二，细心观察 getSize()方法声明，可以发现在形式参数之前有一个“in”关键字。当服务用户调用服务方法时，该关键字表示相关参数传递的方向（direction）。传递方向指示符共有 in、out、inout 三种，in 表示参数要传递到服务方法内部，out 表示将值返回到服务方法的调用端，inout 表示传送相应值并接收返回值。

10.5.2 使用 AIDL 编译器，生成服务接口、服务 Stub 以及服务代理

在 Eclipse 中创建 IMyService.aidl 文件，保存文件后自动生成 IMyService.java 文件，如图 10-29 所示。在 Eclipse 的包浏览器窗口（Package Explorer）的 gen 文件夹下，可以看到生成的 IMyService.java 文件。

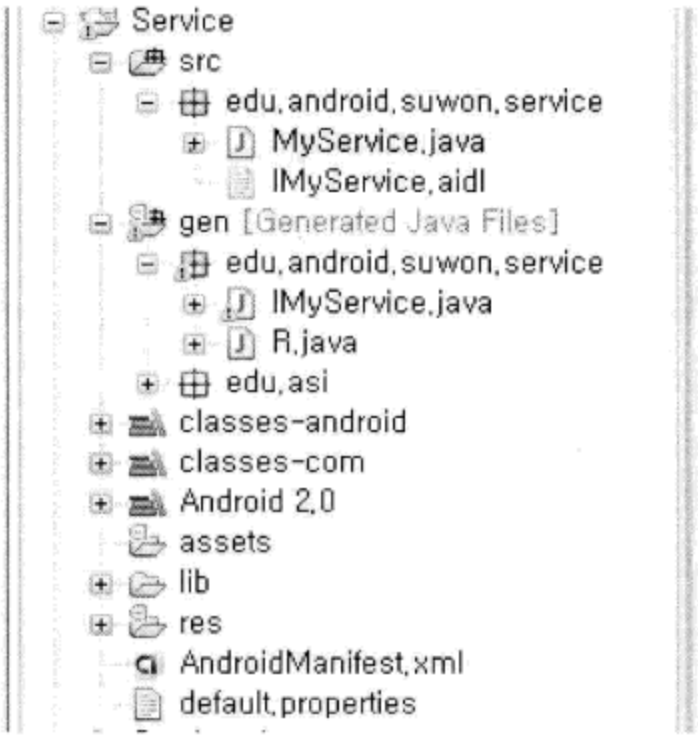


图 10-29 | 自动生成的 IMyService.java 文件

代码 10-44 是 IMyService.java 的主要代码，下面让我们分析一下。

```
/*
* This file is auto-generated.  DO NOT MODIFY.                    ←❶
*/
package edu.android.suwon.service;

public interface IMyService extends android.os.IInterface {
/** Local-side IPC implementation stub class. */                  ←❷
public static abstract class Stub extends android.os.Binder implements
    ➥ edu.android.suwon.service.IMyService {                     ←❸
   private static final java.lang.String DESCRIPTOR =
    ➥ "edu.android.suwon.service.IMyService";
```

更多更新编程资源尽在 http://www.pin5i.com/

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

    @Override
    public boolean onTransact(int code, android.os.Parcel data,          ←❹
        android.os.Parcel reply, int flags)
        throws android.os.RemoteException {
        switch (code) {
        case TRANSACTION_getSize: {
            java.lang.String _arg0;
            arg0 = data.readString();                                    ←❺
            int _result = this.getSize(_arg0);
        }
        }
    }
}
```

代码 10-44 | IMyService.java 的主要代码

❶ IMyService.java 文件自动生成，禁止修改。

❷ 由 AIDL 编译器生成的服务接口，继承于 IInterface 接口。

❸ Stub 类继承了 Binder 类，并实现了自动生成的 IMyService 接口，它是具体的服务。

❹ Stub 类的 onTransact()方法从服务用户端接收 RPC 数据，并调用相应的方法。TRANSACTION _getSize 是 getSize()方法的 RPC 代码，RPC 代码的生成规则以“TRANSACTION_方法名”形式出现。

❺ 调用 Parcel[Java] 类的 readString()方法，根据 in 方向指示符，接收服务用户传递过来的 String 类型的名称。

10.5.3 继承 Stub 类创建服务

下面要创建一个继承 Stub 的类，并实现.aidl 文件中声明的方法。代码 10-44 是 MyService 类的代码，它继承了 Stub 类，并且实现了 test()与 getSize()两个方法，方法的实现很简单，test()方法输出“test”字符串，getSize()方法返回参数 name 的长度。

```
package edu.android.suwon.service;

import android.os.RemoteException;
import edu.android.suwon.service.IMyService.Stub;

public class MyService extends Stub {
```

更多更新编程资源尽在：http://www.pin5i.com/


```
    @Override
    public void test() throws RemoteException {
        System.out.println("test");
    }

    @Override
    public int getSize(String name) throws RemoteException {
        return name.length();
    }

}
```

代码 10-45 | MyService 类

由代码可以看到，MyService 类的两个方法都可能抛出 RemoteException 异常。RemoteException 是所有远程调用异常的父类，在调用可能抛出该异常的方法时，要使用 try/catch 语句处理 RemoteException 异常，这样在 Binder IPC 期间就能够检测到发生的通信错误了。在编写本书之际，RemoteException 异常仅有一个 DeadObjectException 子异常，当生成调用对象的进程终止时即发生此异常。

编写方法时，请遵循以下规则：

- 不要将服务抛出的异常传递给服务用户。
- IPC 调用是同步的，服务用户会一直等待，直到服务返回结果值。
- 不要在接口内部声明属性。

10.5.4 服务接口的调用

经过步骤（1）～（3）后，生成了所需要的服务，那么服务用户怎样才能使用这些服务呢？为实现此目的，必须创建一个实现 IMyService 接口的类，使得服务用户可以调用相应的服务。AIDL 编译器会自动生成一个 Proxy 类，Proxy 类的主要代码如 10-46 所示。

```
private static class Proxy implements
    ➥ edu.android.suwon.service.IMyService {                    ←❶

  public void test() throws android.os.RemoteException {         ←❷
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    try {
      _data.writeInterfaceToken(DESCRIPTOR);
      mRemote.transact(Stub.TRANSACTION_test, _data, _reply, 0);
      _reply.readException();
```

更多更新编程资源尽在 http://www.pin5i.com/

```
        }
    }

    static final int TRANSACTION_test =
        ➥ (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
}
```

代码 10-46 | Proxy 类的主要代码

❶ 与服务一样，实现 IMyService 接口。

❷ 生成调用服务 test()方法的 RPC 数据。RPC 数据中表示调用 test()方法的 RPC 代码（TRANSACTION_test）以及 IMyService 接口信息将作为参数传递给 BinderProxy 的 transact()方法，而后通过 Binder IPC 将 RPC 数据传递给 MyService 服务，调用服务的 test()方法。

10.6 小结

本章我们分析了 Android 的 Java Service Framework。通过分析，发现 Java 服务框架与本地服务框架十分类似，但在系统内部服务运作机制与编写方法上两者是不同的。具体地说，与本地服务框架不同，Java 服务框架为服务用户提供了管理器类（Manager），使用 AIDL 工具自动生成服务接口、服务 Stub、服务代理类，并且 BinderProxy、Binder、Parcel 等组成元素通过 JNI 与本地服务框架的相应元素连接在一起。

不仅如此，讲解时我们还编写了 HelloWorldService 实例，以帮助读者进一步理解 Java 服务框架。这个示例提供的功能非常简单，一些读者可能有些失望，他们希望看到使用 wifi 实现的通话服务或视频通话这类复杂的示例。但在学习基础知识时，笔者建议使用尽可能简单的实例，只要把问题讲解清楚就可以了。正如 C 语言之父 Brain W.Kernighan 在《The C Programming Language》一书第 1 章所言，学习一门语言唯一的方法就是用这门语言来编写程序，并且他采用了闻名世界的“hello，world”示例。在此，笔者也向所有想学习 Android Service Framework 的朋友赠送一句忠告：学习 Service Framework 的唯一之道就是使用它来尝试编写程序。

更多更新编程资源尽在 http://www.pin5i.com/

第 11 章

Java 系统服务运行分析

在上一章中我们重点讲解了有关 Android Java 服务框架的内容，并对 Java 系统服务的运作机制进行了全面系统的学习与探讨。本章将通过对一个 Java 系统服务实例—Activity Manager Service 源代码的分析，进一步学习 Java 系统服务在 Android 框架中是如何运作的。

11.1 Activity Manager Service

Activity Manager Service 是一种 Java 系统服务，它是平台核心服务之一，用来创建 Android 应用程序组件（Activity）、服务、广播接收器（Broadcast Receiver）等，并管理它们的生命周期。关于 Android 应用程序组件的具体内容，已超出本书讨论范围，请参考相关书籍或访问 Android 开发者网站中 Application Fundamentals 部分（http://developer.android.com/ guide/topic/fundamentals.html）。

> **TIP 关于术语“服务”**
>
> 在本章中，系统服务与基于 SDK 的应用程序服务术语混用。如无特殊说明，“服务”均指应用程序服务，它是 Android 应用程序组件构成元素之一。

本章以 ApiDemos 中的 Remote Service Controller 应用程序（参考图 11-1）为例，讲解 Activity Manager Service 是如何生成 Android 应用程序服务的，以及如何管理相关服务的生命周期的。同时，还要学习请求服务的应用程序进程与 Activity Manager Service 是如何进行相互作用的。

如图 11-1 所示，Remote Service Controller 应用程序比较简单，它通过单独的进程运行 Remote Service，（1）点击“Start Service”按钮，（2）RemoteService 就被启动运行。当应用程序运行服务时，Activity Manager Service 将负责运行应用程序请求的 RemoteService。

那么，Activity Manager Service 是如何运行 RemoteService 的呢？RemoteService 是运行在应用程序与单独进程中的远程应用程序服务，若运行它，必须首先生成新的进

更多更新编程资源尽在 http://www.pin5i.com/

程。在第 5 章中已经讲过，所有基于 Java 的进程都是由 Zygote 创建的，所以 Activity Manager Service 使用 Zygote 创建运行服务的进程。然后，在新创建的进程中运行应用程序请求的 RemoteService 服务。

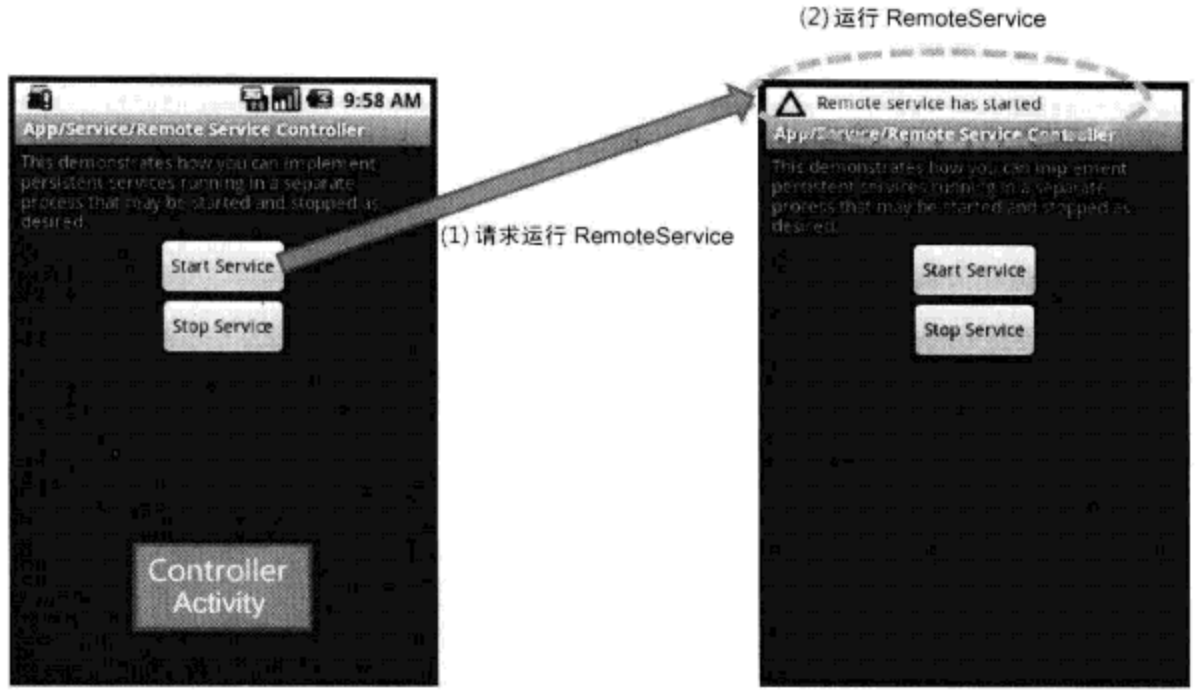


图 11-1 | 运行 ApiDemos 中的 Remote Service Controller 应用程序

如图 11-2 所示，它描述了当按下“Start Service”按钮时，Activity Manager Service 生成应用程序服务 RemoteService 的详细过程。

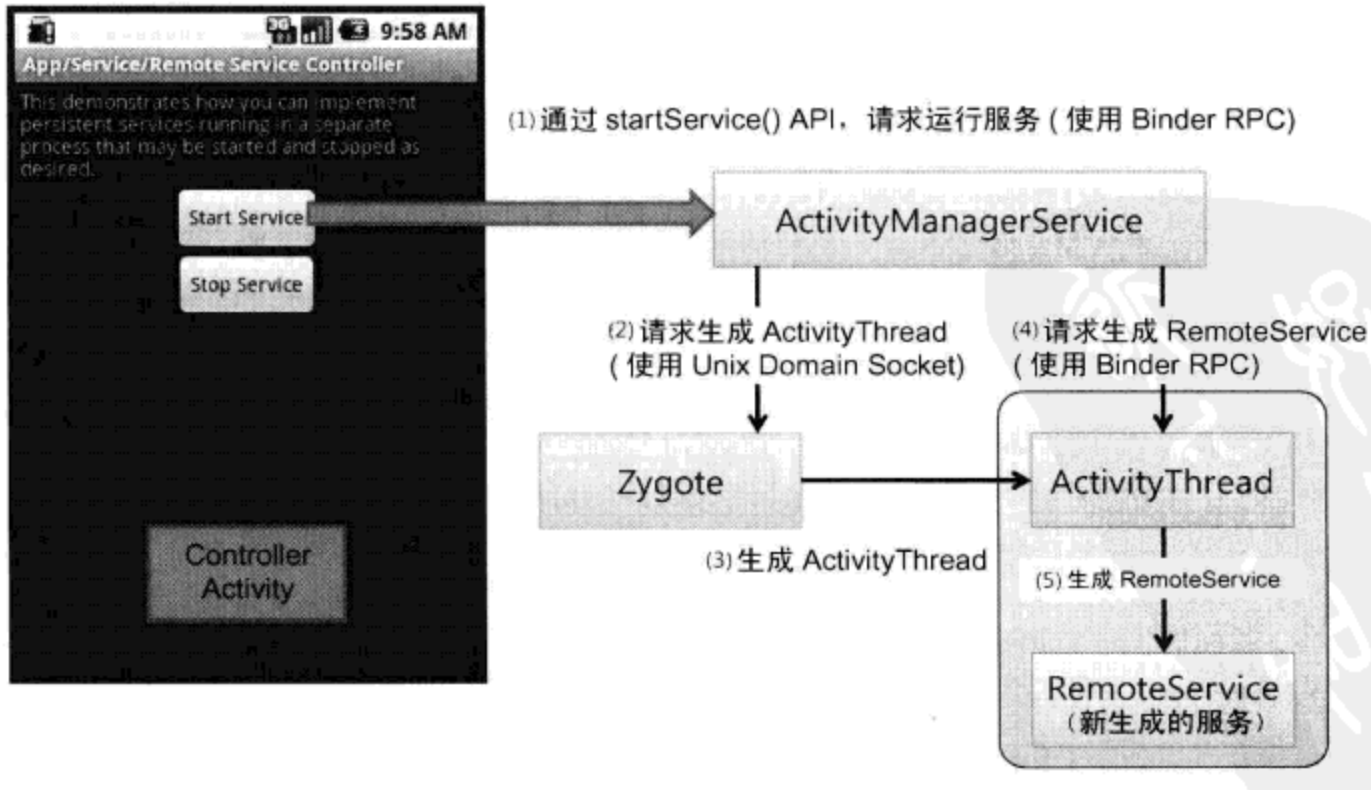


图 11-2 | 生成 Remote Service

Remote Service 生成过程如下。

更多更新编程资源尽在 http://www.pin5i.com/

（1） Android 应用程序通过 startService()或 bindService() API 生成应用程序服务。在本章中我们将通过分析 startService()方法的行为学习 Activity Manager Service 生成服务的方法。

（2） 在应用程序通过 startService()请求运行服务后，Activity Manager Service 并未直接加载所请求的服务类（RemoteService.class），而是向 Zygote 请求生成用于运行服务的 ActivityThread。这是因为 RemoteService 是一个运行在单独进程中的远程服务。ActivityThread[1]是 Android 所有应用程序的主线程，负责生成 Activity、服务，并管理它们的生命周期。

（3） Zygote 从 Activity Manager Service 接收生成 Activity Thread 请求后，新创建一个进程，并将 Activity Thread 加载到其中。

（4） Activity Manager Service 向（3）中生成的 Activity Thread 请求生成 RemoteService 服务。

（5） ActivityThread 启动运行 RemoteService 服务。

Activity Manager Service 是 Android 非常重要的系统服务，它主要用来生成应用程序组件，如应用程序服务、Activity、广播接收器（Broadcast Receiver）等，并且它也负责管理这些组件的生命周期。

正如第 6 章服务概要中提到的一样，从服务启动到服务终止即是服务的一个生命周期。如图 11-3 所示，它描述了从服务启动到服务终止的整个周期。

如图所示，使用 Activity Manager Service 生成 Activity、服务等应用程序组件，并管理这些组件的生命周期，整个过程看似非常简单。但若想具体理解内部运行方式，必须先理解 Binder（第 7 章）与 Java 服务框架（第 10 章）相关的知识。本章将以这些知识为基础，同各位一起更具体、更详细地学习 Activity Manager Service 究竟是如何运行的。

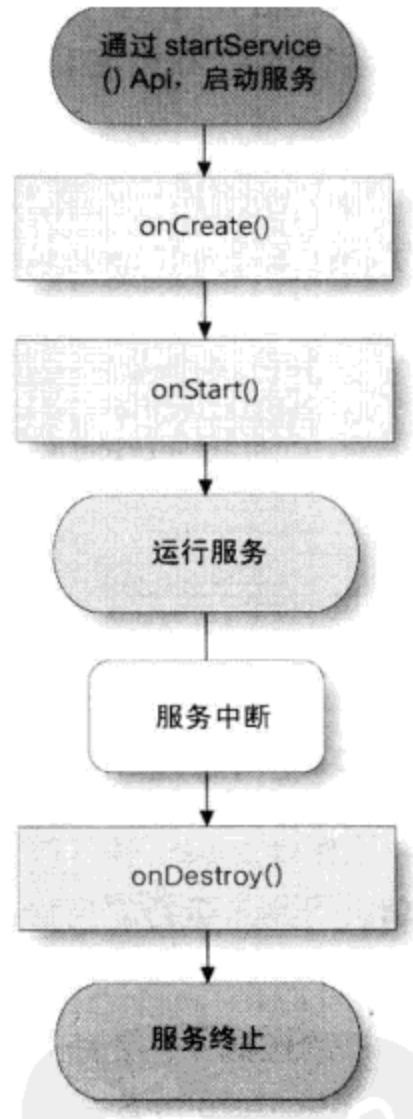


图 11-3 | Android 服务的生命周期

11.2 Activity Manager Service 创建服务分析

如图 11-2 中的（1）～（5）所示，Activity 调用 startService()方法，Activity Manager

1 frameworks/base/core/java/android/app/ActivityThread.java

更多更新编程资源尽在 http://www.pin5i.com/

Service 就会生成应用程序服务。下面通过分析代码，来学习 Activity Manager Service 生成应用程序服务的过程。

11.2.1 Controller Activity–调用 startService()方法

如图 11-2(1)所示，点击“Start Service”按钮，代码 11-1 中的单击事件处理代码就会被调用。此时，单击事件处理代码调用 startService()方法，一个 Intent 对象将作为参数传递给它。

Android 应用程序组件由 Intent 消息激活。在 Android 中使用 Intent 显式地指定要运行的服务类名称（或使用[Action]隐性指定），即可运行指定的组件（这里指服务）。

代码 11-1 是一个请求运行 RemoteService 服务的示例，它使用了隐性的 Intent，指定为 com. example.android.apis.app.REMOTE_SERVICE。

```
private OnClickListener mStartListener = new OnClickListener() {
    public void onClick(View v) {
        startService(new Intent("com.example.android.apis.app.REMOTE_SERVICE"));
    }
};
```

代码 11-1 | RemoteService.java-点击“Start Service”按钮后的事件处理代码

TIP 使用隐性 Intent 运行应用程序服务

隐性 Intent 并不包含等待运行的 Activity 或服务的名称，它只含有特定的 Action 值，可用来运行应用程序组件。Android 框架接收了包含在隐性 Intent 中的 Action 之后，它将检索最适合处理该 Action 的 Activity 或服务，并运行它。

如代码 11–2 所示，它是 ApiDemos 示例程序的 AndroidManifest.xml 文件中有关 RemoteService 的内容。代码中使用<intent–filter>标记，使得 Remote Service 能够处理 com.example.android. apis.app.REMOTE_SERVICE 动作。（在 Android 框架中已经包含这类信息。）如代码 11–1 所示，Android 框架接收隐性 Intent 之后，通过 Intent 中包含的动作信息，确定要运行的服务为 Remote Service。

关于 Intent 与 Intent Filter 的内容，请参考 Android 开发者网站中有关 Intents and Intent Filters 的部分（http://developer.android.com/guide/topics/intents/ intents–filters.html）。

更多更新编程资源尽在 http://www.pin5i.com/


```
<service android:name=".app.RemoteService" android:process=":remote">
    <intent-filter>
        <action android:name="com.example.android.apis.app.IRemoteService" />
        <action android:name="com.example.android.apis.app.ISecondary" />
        <action android:name="com.example.android.apis.app.REMOTE_SERVICE" />
    </intent-filter>
</service>
```

代码 11-2 | ApiDemos 示例程序的 AndroidManifest.xml 文件中有关 RemoteService 的部分

11.2.2 Activity Manager Service 的 startService()方法的调用过程（使用 Binder RPC）

当 Activity 调用 startService() API 后，Android 框架内部是如何处理这一调用的呢？下面让我们来分析一下。如代码 11-1 所示，Activity 调用的 startService() API 只是负责向 Activity Manager Service 提出请求，以生成服务以及运行相关的内容，具体功能由 Activity Manager Service 中的具体相同名称的 startService()[1] Stub 方法实现。

换言之，在 Activity 中调用 startService() API，将通过 Binder RPC 调用 Activity Manager Service 提供的 startService() Stub 方法。

图 11-4 更详细地描述了图 11-2（1）的过程，即它展现了当 Controller Activity 调用 startService()方法时如何通过 Binder RPC 调用 Activity Manager Service 的 startService() Stub 方法的过程。

图 11-4 描述的调用过程如下所示。

1. Controller Activity：调用 ActivityManagerProxy 对象的 startService()代理方法。
2. ActivityManagerProxy 对象：通过 Java 服务框架向 ActivityManagerNative 对象发送 START_SERVICE_TRANSACTION RPC 代码以及 Binder RPC 数据。
3. ActivityManagerNative 对象：调用 ActivityManagerService 的 startService() Stub 方法。

接下来，通过分析一些主要代码，进一步理解调用 startService()的过程。

（1） Controller Activity（调用 ActivityManagerProxy 对象的 startService()代理方法）。

1 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 类的 startService()方法。

更多更新编程资源尽在 http://www.pin5i.com/

（a） ContextWrapper 类–调用 ContextImpl 对象的 startService()方法。

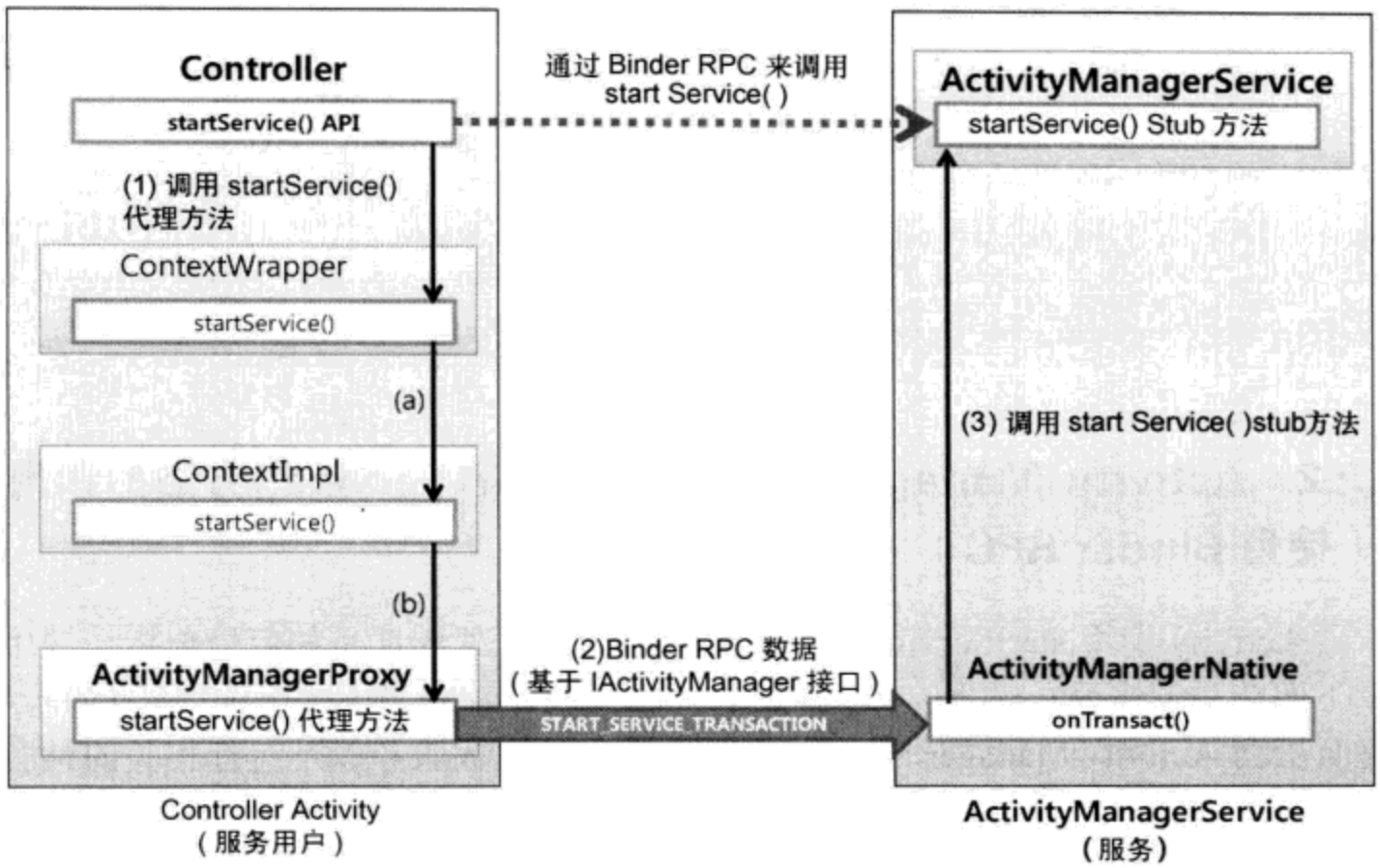


图 11-4 | Controller Activity 通过 Binder RPC 调用 ActivityManagerService 的 startService()方法

```
public class ContextWrapper extends Context {
   Context mBase;

   public ComponentName startService(Intent service) {
      return mBase.startService(service);
   }
}
```

代码 11-3 | ContextWrapper 类-startService()方法的主要代码

当 Activity 调用 startService() API 时，ContextWrapper 类[1]的 startService()方法就会被调用，如代码 11-3 所示。ContextWrapper 继承了 Context 抽象类，负责包装 mBase 成员变量中的 Context 对象。如图 11-4 所示，ContextWrapper 对象包装了 ContextImpl 对象。因此，调用 ContextWrapper 的 startService()方法会引起对 ContextImpl 对象的 start Service()方法的调用。

1 frameworks/base/core/java/android/content/ContextWrapper.java

更多更新编程资源尽在 http://www.pin5i.com/

（b） ContextImpl 类-startService()方法

```
public ComponentName startService(Intent service) {
    ComponentName cn =  ActivityManagerNative.getDefault().startService(
                        ➥ mMainThread.getApplicationThread(), service,
                        ➥ service.resolveTypeIfNeeded(getContentResolver()));

    return cn;
}
```

代码 11-4 | ContextImpl 类-startService()方法主要代码

ContextImpl[1]实现了 Context 抽象类，负责访问应用程序自身资源、运行 Activity 或应用程序服务，以及收发 Intent 等。代码 11-4 是 ContextImpl 类中 startService()方法的主要代码。

请看图 11-5，它描述的是调用 ActivityManagerNative.getDefault().startService()方法后的情形。如图所示，调用 ActivityManagerNative[2].getDefault()函数会返回 ActivityManagerProxy[3]对象，该对象用于通过 Binder RPC 调用 Activity Manager Service 提供的 IActivityManager 服务接口方法。Activity 端将通过该对象自由地调用 Activity Manager Service 的 IActivityManager 接口中的多种方法，就像调用本地函数一样。

TIP IActivityManager*服务接口

该服务接口中定义了 Activity Manager Service 向应用程序提供的多种 API。Android 应用程序使用该接口向 Activity Manager Service 请求多种功能。IActivityManager 服务接口定义了一系列的方法，如控制 Android 应用组件运行的方法，以及管理 Activity 生命周期的方法 activityIdle()、activityPaused()等。

* frameworks/base/core/java/android/app/IActivityManager.java

代码 11-4 的 ActivityManagerNative.getDefault().startService()方法将会调用 Activity Manager Proxy 的 startService()代理方法，ActivityManagerProxy 的 startService()方法原型见代码 11-6。如图 11-5 所示，随着 ActivityManagerProxy 的 startService()方法被调用，ActivityManager Service 的 startService() Stub 方法就会被远程调用，两个 startService()方法统一使用如下函数原型。

由代码 11-5 可知，startService()方法主要有两个参数，第一个参数 caller 为 Iapplication Thread 类型，负责处理由 Activity Manager Service 传递过来的基于 IApplicationThread

1 frameworks/base/core/java/android/app/ContextImpl.java
2 frameworks/base/core/java/android/app/ActivityManagerNative.java
3 frameworks/base/core/java/android/app/ActivityManagerNative.java 的 ActivityManagerProxy 类。

更多更新编程资源尽在 http://www.pin5i.com/

服务接口的 Binder RPC。

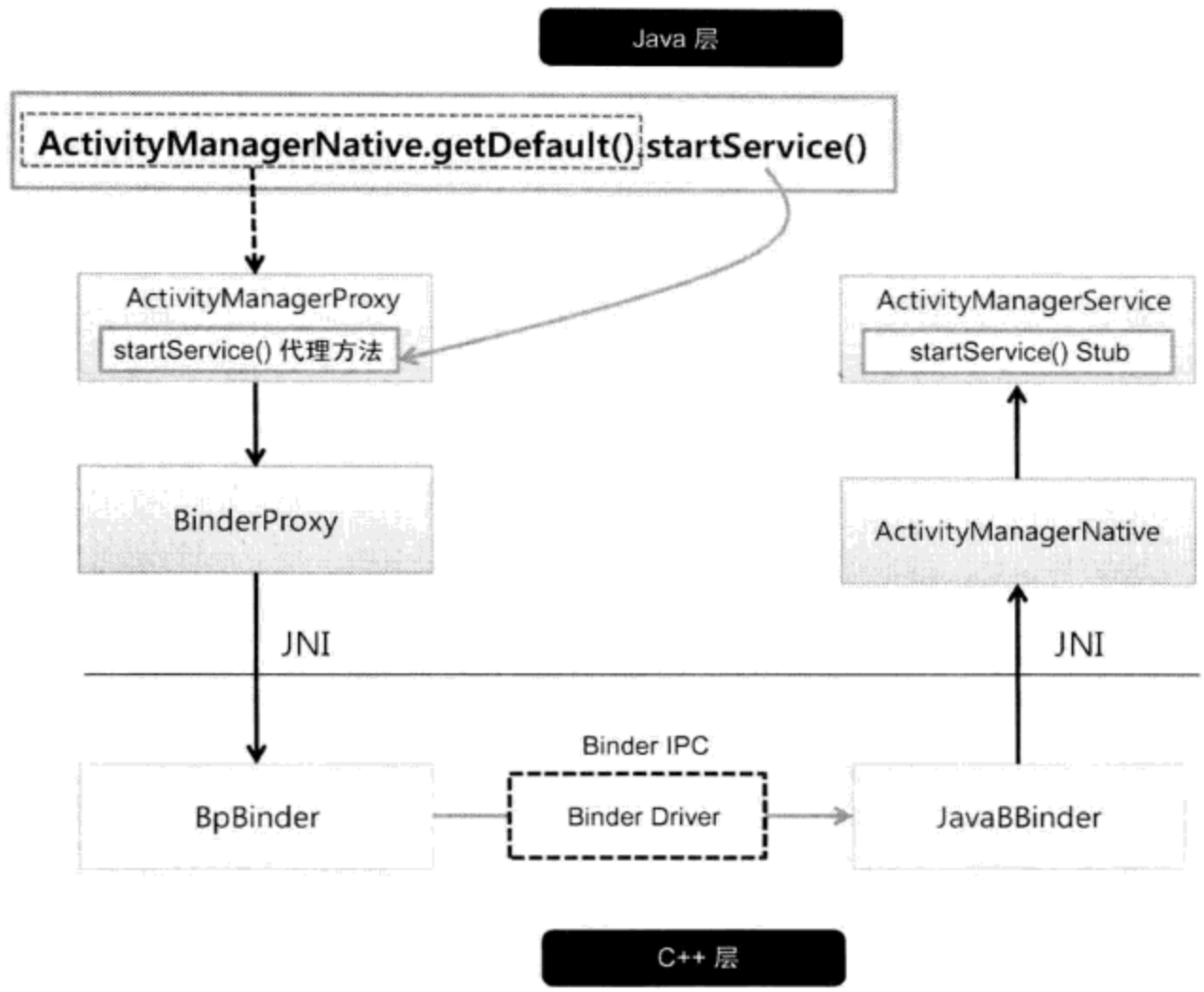


图 11-5 | 基于 IActivityManager 接口的 Binder IPC 设定

```
public ComponentName startService(IApplicationThread caller,
                                  Intent service,
                                  String resolvedType)
```

代码 11-5 | ActivityManagerProxy 类-startService()代理方法原型

第二个参数 service 是 Intent 类型，含有要运行的服务的信息，代码 11-1 中 startService()方法的 Intent 参数被原封不动地传递过来。

TIP IApplicationThread 服务接口*

Activity Manager Service 使用 IApplicationThread 接口来控制与自身相连的应用程序。该接口中提供了一些方法，如 schedulePauseActivity()或 sheduleStopActivity()方法等，以供 Activity Manager Service 管理应用程序的生命周期。

*frameworks/base/core/java/android/app/IApplicationThread.java

更多更新编程资源尽在 http://www.pin5i.com/

（2） ActivityManagerProxy 对象-startService()代理方法。

```
public ComponentName startService(IApplicationThread caller,
            ➥  Intent service,String resolvedType) {
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();

    // 将参数值保存到 data
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    service.writeToParcel(data, 0);
    data.writeString(resolvedType);

    // 传递 Binder RPC 数据
    mRemote.transact(START_SERVICE_TRANSACTION, data, reply, 0);
    ComponentName res = ComponentName.readFromParcel(reply);

    return res;
}
```

代码 11-6 | ActivityManagerProxy 对象-startService()方法主要代码

调用 ActivityManagerProxy 对象的 startService()代理方法后，startService()代理方法将进行哪些处理呢？如图 11-5 所示，ActivityManagerProxy 对象与 Activity Manager Native 对象连接在一起，ActivityManagerNative 对象用于接收并处理基于 IActivity Manager 服务接口的 Binder RPC 数据，ActivityManagerProxy 对象的主要功能是向 Activity Manager Native 对象传递 Binder RPC 数据。

代码中 mRemote.transact()方法用来在 Java 对象中传递 Binder RPC 数据。Activity ManagerProxy 对象 startService()代理方法使用 Parcel 类型的 data 变量保存来自参数 caller、service、resolvedType 的值，而后调用 mRemote.transact()，传入 START_SERVICE_TRANSACTION，将保存在 data 中的数据传递给 ActivityManagerNative。

（3） ActivityManagerNative 对象–调用 startService() Stub 方法。

通过 START_SERVICE_TRANSACTION，从 ActivityManagerProxy 对象接收到 Binder RPC 数据后，ActivityManagerNative 将调用 onTransact()方法来处理它。

ActivityManagerNative 对象分析从 ActivityManagerProxy 获取的 RPC 代码，确定 Activity Manager Service 要调用的 Stub 方法。由于 ActivityManagerProxy 传送的 RPC 代码为 START_SERVICE_TRANSACTION，所以 onTransact()将调用 startService()代理方法。

在调用 startService() Stub 方法之前，需要首先获取它的参数。为此，需要先将 on Transact()的 data 参数中包含的 Binder RPC 数据（基于 IActivityManager 服务接口）进行 Unshalling 处理，而后以获取的数据为参数调用 Activity Manager Service 的 start Service()

更多更新编程资源尽在 http://www.pin5i.com/

方法。

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
{
   switch (code) {
   ...
   case START_SERVICE_TRANSACTION: {

      data.enforceInterface(IActivityManager.descriptor);
      IBinder b = data.readStrongBinder();
      IApplicationThread app =  ApplicationThreadNative.asInterface(b);
      Intent service = Intent.CREATOR.createFromParcel(data);
      String resolvedType = data.readString();
      ComponentName cn = startService(app, service, resolvedType);

      return true;
   }
   ...
   }

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

代码 11-7 | ActivityManagerNative 的 onTransact()方法

代码 11-7 是 ActivityManagerNative 的 onTransact()方法的主要代码，该方法从 Activity Manager Proxy 处接收 RPC 代码（START_SERVICE_TRANSACTION）与 Binder RPC 数据，而后进行处理。

如图 11-6 所示，onTransact()方法通过 Binder RPC 接收 data 变量（Parcel 类对象）中的数据，即传入 ActivityManagerProxy 的 startService()代理方法的参数值（caller、service、resolvedType）。而后对 data 变量中的数据进行 Unshalling 处理，将获得的各个数据分别保存到相应的变量中。然后以保存各个数据的变量为参数，调用 Activity Manager Service 的 startService() Stub 方法。

在代码 11-7 中有一条语句如下。

```
IApplicationThread app=ApplicationThreadNative.asInterface(b);
```

ApplicationThreadNative.asInterface()方法将为 ApplicationThreadNative 生成 Application Thread Proxy 对象。而后在请求生成服务的 RemoteActivityController Activity 与 Activity Manager Service 之间形成两个 Binder 连接，如图 11-7 所示。Activity 通过 Binder RPC（基于 IActivityManager 服务接口）向 Activity Manager Service 请求运行指定的服务、收发 intent 等功能。并且，Activity Manager Service 也通过基于 Iapplication Thread 接口的 Binder RPC 控制与自身相连的应用程序。

更多更新编程资源尽在 http://www.pin5i.com

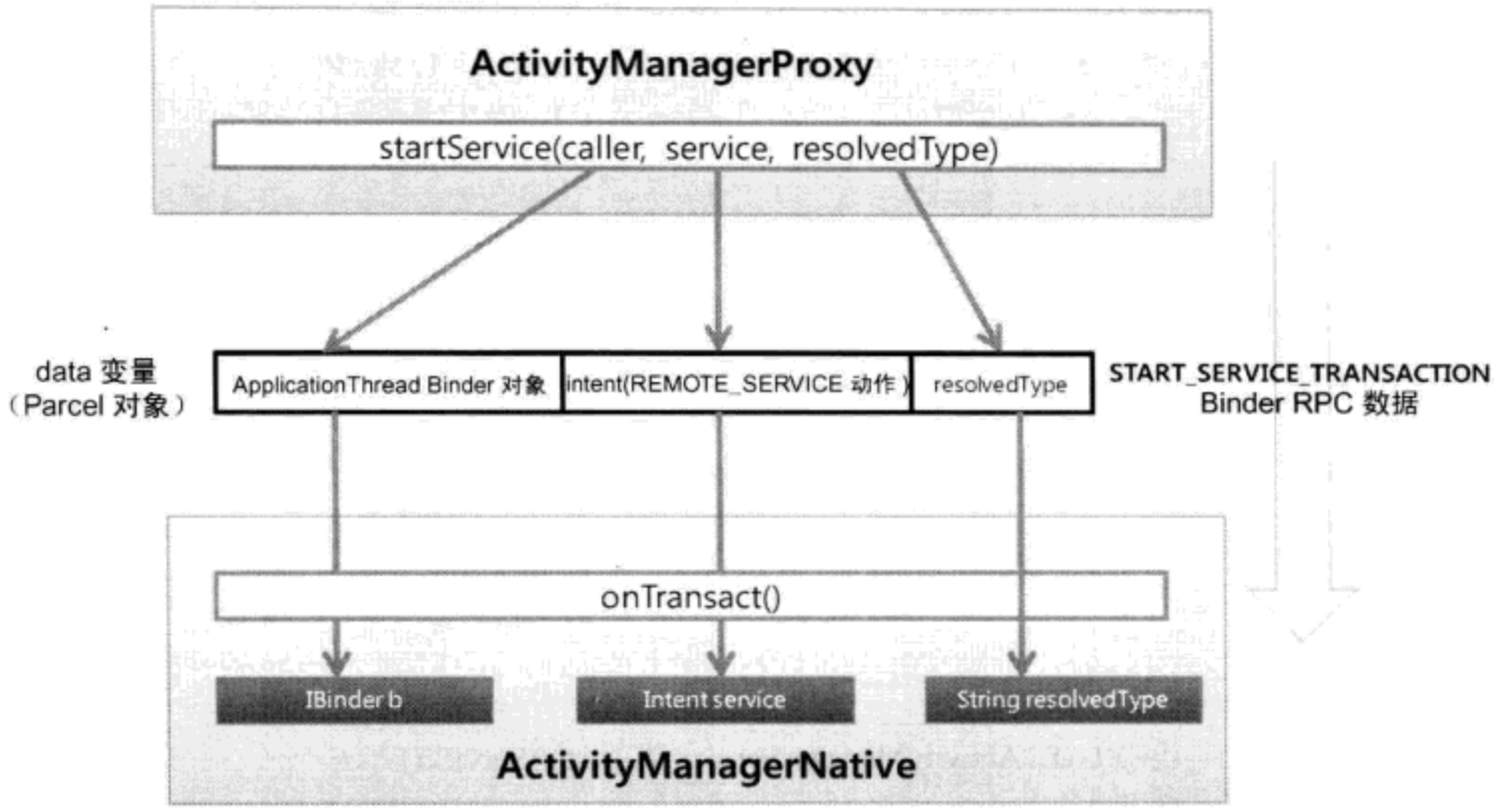


图 11-6 | 在 ActivityManagerProxy 与 ActivityManagerNative 间传递 Binder RPC 数据

图 11-7 | Activity 与 Activity Manager Service 间的 Binder 连接

以上通过代码分析了 Controller Activity 调用 startService() API 时是如何通过 Binder RPC 调用 Activity Manager Service 的 startService() Stub 方法的。在这一过程中，Controller Activity 通过 RPC 机制调用 Activity Manager Service 的 startService() Stub 方法就像调用自身的本地方法一样简单，如图 11-4 所示。

11.2.3 Activity Manager Service——运行 startService()Stub 方法

前面我们学习了 Activity 向 Activity Manager Service 传递服务信息的过程。接下来，

更多更新编程资源尽在 http://www.pin5i.com/

将学习 Activity Manager Service 是如何运行请求的服务的。本节重点分析 Activity Manager Service 的 startService()方法是如何执行的，它具体有哪些行为动作，对应于图 11-2（2）过程。

```
public ComponentName startService(IApplicationThread caller,  Intent service, String
                                                                    resolvedType)
{
   final int callingPid = Binder.getCallingPid();
   final int callingUid = Binder.getCallingUid();

   ComponentName res = startServiceLocked(caller, service, resolvedType,
                                          callingPid, callingUid);

   return res;
}
```

代码 11-8 | ActivityManagerService 类的 startService()方法

代码 11-8 是 Activity Manager Service 中 startService()方法的主要代码。该方法的主要语句是调用 startServiceLocked()方法的语句。startServiceLocked()方法的第一个参数 caller，它持有 ApplicationThreadProxy 对象的引用。如图 11-7 所示，Application Thread Proxy 对象是 Activity Manager Service 为了通过基于 IApplicationThread 服务接口的 Binder RPC 控制 Controller Activity 而生成的；第二个参数 service 持有 Intent 对象引用，它含有要生成的服务信息；最后两个参数，它们用来将请求生成服务的进程（这里是指运行 Controller Activity 的进程）的 pid 与 uid 传入方法中。

```
ComponentName startServiceLocked(IApplicationThread caller, Intent service, String
                                 resolvedType,int callingPid, int callingUid)
{
   // 从retiveServiceLocked() 方法的返回值 ServiceLookupResult 结构体变量 res 获取
   // ServiceRecord 值。
   ServiceLookupResult res = retrieveServiceLocked(service, resolvedType, callingPid,
                                                   callingUid);

   ServiceRecord r = res.record;
   bringUpServiceLocked(r, service.getFlags(), false);

   return r.name;
}
```

代码 11-9 | ActivityManagerService 类的 startServiceLocked()方法

请看代码 11-9，它是 startServiceLocked()方法的主要代码，其主要功能是调用 retrieveService Locked()方法，并从返回的 ServiceLookupResult 结构体变量 res 获取 Service

更多更新编程资源尽在 http://www.pin5i.com/

Record 值。

ServiceRecord 类[1]含有与 Android 应用程序服务相关的各种信息，比如服务包名和位置、权限、服务进程信息、运行统计信息等。

如代码 11-9 所示，startServiceLocked()方法通过调用 retrieveService Locked()方法，并向其传递 Intent 对象，以此获取服务相关信息。参数 service 中保存着 com.example.android.apis.app. REMOTE_SERVICE 动作（请参考代码 11-1），它用来指定隐性的（implicit）intent，方法 retrieveService Locked()会查找最合适的服务来处理它，并且将查找的服务信息保存到 ServiceRecord 对象中。

同时，请求服务的进程（这里是指 Controller Activity）的 pid 与 uid 也将以参数的形式传入 retrieveService Locked()方法中，以便检查它是否拥有运行指定服务的权限，这样能有效地防止不具有权限的进程运行指定的服务。

获取 ServiceRecord 对象后，它会被传递给 bringUpServiceLocked()方法，作为该方法的第一个参数。bringUpServiceLocked()方法的主要代码，如代码 11-10 所示。

```
private final boolean bringUpServiceLocked(ServiceRecord r, int intentFlags,
                                           boolean whileRestarting)
{
    final String appName = r.processName;
    ProcessRecord app = getProcessRecordLocked(appName, r.appInfo.uid);

    if (app != null && app.thread != null) {
        // 不用生成 Process，直接在已有的进程域内运行服务
        realStartServiceLocked(r, app);
        return true;
    }

    startProcessLocked(appName, r.appInfo, true, intentFlags,"service", r.name, false)
    mPendingServices.add(r);

    return true;
}
```

代码 11-10 | ActivityManagerService 类的 bringUpServiceLocked()方法

bringUpServiceLocked()首先使用 ServiceRecord 对象，调用 getProcessRecordLocked()方法，通过运行 RemoteService 服务的进程名称与 uid，检索 ProcessRecord[2]对象是否已经存在。

1 frameworks/base/services/java/com/android/server/am/ServiceRecord.java

2 frameworks/base/services/java/com/android/server/am/ProcessRecord.java

更多更新编程资源尽在 http://www.pin5i.com/

ProcessRecord 类包含当前处于运行中的特定进程的大部分信息。若 ProcessRecord 已经存在，且运行服务的进程已经处于运行状态中，通过调用 realStartServiceLocked() 方法，可以在同进程区域内启动运行服务。

AndroidManifest.xml 文件中有关 RemoteService 的部分，<service>标签的<android:process>属性值被设置为“:remote”，如代码 11-2 所示，所以 RemoteService 服务将以远程服务的形式运行。即在运行 RemoteService 服务之前，先创建新的进程，而后在新的进程域内运行服务。在代码 11-10 中，运行服务的进程尚未生成，app 变量值为 NULL，程序继续调用执行 startProcessLocked()方法，其主要代码如 11-11 所示。

为了便于向新生成的进程请求 RemoteService 服务，Activity Manager Service 将 ServiceRecord 对象保存到 mPendingServices 数组中。如前所言，Activity Manager Service 自身不会直接运行服务，而是向 Zygote 生成的 ActivityThread 请求运行指定的服务。在 Zygote 生成 ActivityThread 之后，Activity Manager Service 通过保存在 mPending Services 数组中的 Service Record 对象向 ActivityThread 请求运行指定的服务。

关于<service>标签及 android:process 属性的相关内容，请参考 Android 开发者网站中有关 AndroidManifest.xml 部分(http://developer.android.com/guide/topic/manifest/ma nifest-intro. html)。

```
    private final ProcessRecord startProcessLocked(String processName,
    ➥ ApplicationInfo info, boolean knownToBeDead, int intentFlags, String hostingType,
    ➥ ComponentName hostingName,
    ➥ boolean allowWhileBooting)
{
    // 新生成 ProcessRecord
    app = newProcessRecordLocked(null, info, processName);

    mProcessNames.put(processName, info.uid, app);
    startProcessLocked(app, hostingType, hostingNameStr);

    return (app.pid != 0) ? app : null;
}

private final void startProcessLocked(ProcessRecord app, String hostingType,
                                      ➥ String hostingNameStr)
{
    int uid = app.info.uid;
    int[] gids = null;
    int debugFlags = 0;

    gids = mContext.getPackageManager().getPackageGids(app.info.packageName);

    int pid = Process.start("android.app.ActivityThread",
```

更多更新编程资源尽在 http://www.pin5i.com/


```
                    null, uid, uid, gids, debugFlags, null);

        // 以生成的进程的pid 为键，将 ProcessRecord 对象保存到
           Activity Manage Service 的哈希表 mPidsSelfLocked 中
        this.mPidsSelfLocked.put(pid, app);
    }
```

代码 11-11 | ActivityManagerService 类的 startProcessLocked()方法

在 ActivityManagerService 类代码中，存在两个 startProcessLocked()方法，它们拥有不同的参数与返回值，如代码 11-11 所示。在代码 11-10 中，调用的是拥有 7 个参数的 startProcess Locked()方法，即代码 11-11 中的第一个 startProcessLocked()方法。为了运行远程服务，该方法首先创建包含新进程信息的 ProcessRecord 对象，将其插入 mProcess Names 队列，而后调用第二个 startProcessLocked()方法，即拥有三个参数的 startProcess Locked()方法。

第二个 startProcessLocked()方法，即拥有三个参数的 startProcessLocked()方法通过调用 Process 类[1]的 start()方法，请求 Zygote 生成 android.app.ActivityThread 进程。关于 Zygote 如何生成 Android Java 进程在第 5 章中已经讲解过，在此不再赘述。而后将 Zygote 生成的进程的 pid（键）与 ProcessRecord 对象（值）以键值对（key/value）的形式保存到 mPidsSelfLocked 哈希表中。这样，Activity Manager Service 就可以通过 pid 获取哈希表中的 ProcessRecord 对象，并通过它管理 pid 对应的进程的 ProcessRecord 信息。

11.2.4 运行 ActivityThread 类的 main()方法

以上学习了在生成 RemoteService 时（如图 11-2）Activity Manager Service 向 Zygote 请求生成 ActivityThread 的过程，它是为了运行服务而生成新进程的准备阶段。

那么，为了运行服务，Zygote 是如何在新进程中运行 Activity Manager Service 请求的 ActivityThread 类的呢？下面将分析这个过程。

在第 5 章中已经提到过，Zygote 会接收运行指定类的请求，创建新的进程，将指定的类加载到进程中，而后调用类的 main()方法。当 Zygote 接收到运行 ActivityThread 类的请求后，Zygote 会将 ActivityThread 类加载到新生成的进程中，如图 11-2（3）所示，而后调用 ActivityThread 类的 main()方法。

1 frameworks/base/core/java/android/os/Process.java

更多更新编程资源尽在 http://www.pin5i.com/

```
public final class ActivityThread {
    // 创建 ApplicationThread()对象
    final ApplicationThread mAppThread = new ApplicationThread();

    public static final void main(String[] args) {
        Looper.prepareMainLooper();  // 初始化消息队列
        ActivityThread thread = new ActivityThread();
        thread.attach(false);
        Looper.loop();  // 运行消息循环
    }

   // 消息处理
    public void handleMessage(Message msg) {
        switch (msg.what) {

            ...

            case CREATE_SERVICE:
                handleCreateService((CreateServiceData)msg.obj);
                break;

            case SERVICE_ARGS:
                handleServiceArgs((ServiceArgsData)msg.obj);
                break;

            case STOP_SERVICE:
                handleStopService((IBinder)msg.obj);
                maybeSnapshot();
                break;

            ...
        }
    }
}
```

代码 11-12 | ActivityThread 类的 main()方法

代码 11-12 是 ActivityThread 类 main()方法的主要代码。下面来分析 main()方法，首先调用 Looper.prepareMainLooper()方法，生成消息队列（详细内容，请参考 TIP），以便在同进程的各线程间传递消息。

TIP Android 消息队列

Android 消息队列用于在同进程的多个线程间传递消息。但是在 Android 系统中并不存在用于进程间通信的全局消息队列，进程间的消息传递要通过 Intent 来进行。

各个 Android 线程拥有独有的 Looper(frameworks/base/core/java/android/os/ Looper.java) 用来处理消息队列。Looper 是一个用于运行消息循环的类，使用时一般先要调用 Looper.prepare()

更多更新编程资源尽在 http://www.pin5i.com/

方法生成消息队列，而后调用 Looper.loop()方法运行消息队列。在代码 11-12 中调用了 Looper.prepareMainLooper()方法生成消息队列，这与生成应用程序的主 Looper 时使用的方法是一样的。

在 Android 开发者网站上有一个为处理消息队列而创建 Looper 线程的示例，请读者进入 http://developer.android.com/reference/android/os/Looper.html 页面查看示例源码。

```
class LooperThread extends Thread
{
   public Handler mHandler;  // 消息处理器

   public void run() {
      Looper.prepare();   // 生成消息队列

      // 创建消息处理对象
      mHandler = new Handler() {
         public void handleMessage(Message msg) {

            // 在此处理接收到的消息
         }
      };

      Looper.loop();      // 运行消息队列
   }
}
```

main()方法在生成消息队列后，紧接着创建 ActivityThread 对象。该对象通过与 Activity Manager Service 相互作用，负责运行 Android 应用程序进程的主线程以及管理 Activity 生命周期等。

如代码 11-12 所示，在 ActivityThread 对象的 mAppThread 变量中保存着 Application Thread 对象。ApplicationThread 类继承自 ApplicationThreadNative 类，它负责将从 Activity Manager Service 接收到的 Binder RPC 数据（基于 IApplicationThread 接口，且用于实现对 ActivityThread 的控制）通过消息队列传递给 ActivityThread。

当生成 ActivityThread 对象后，它的 attach()方法即被调用，图 11-8 描述了调用 attach()方法而引发的一系列的行为。如图所示，attach()方法的实际处理是由 Activity Manager Service 的 attachApplication() Stub 方法负责的。

为了处理 attach()方法调用，ActivityThread 将通过 Binder RPC 调用 Activity Manager Service 的 attachApplication()方法。为此，如图 11-8 所示，ActivityThread 对象首先生成 ActivityManager Proxy，而后通过 Binder RPC 访问 Activity Manager Service。如前所言，ActivityManagerProxy 对象用于向 Activity Manager Service 请求特定的功能。

以上过程大致分为三步，简单整理如下：

1. ActivityThread–调用 ActivityManagerProxy 对象的 attachApplication()代理方法。

更多更新编程资源尽在 http://www.pin5i.com/

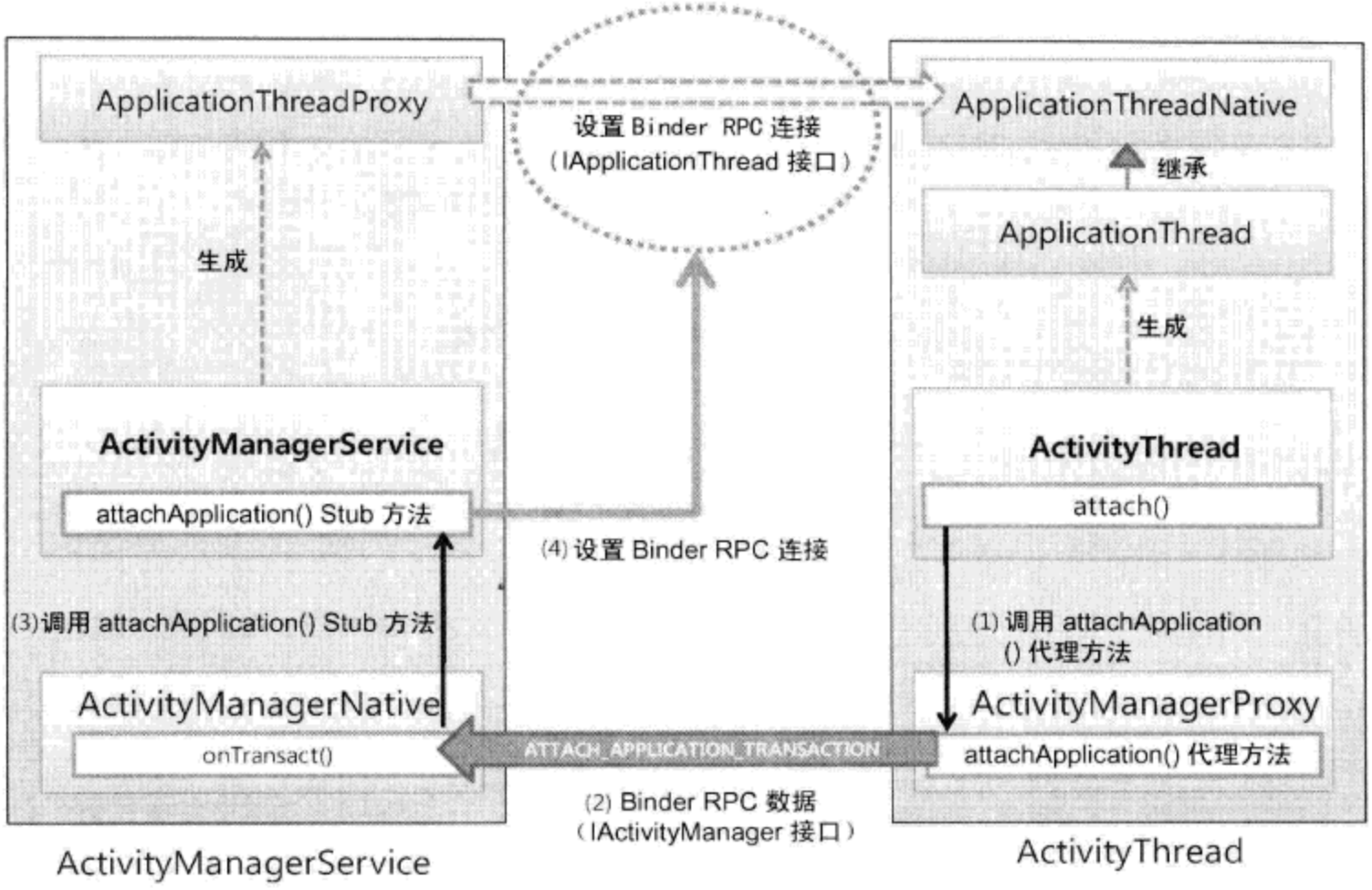


图 11-8 | ActivityThread 对象的 attach()方法的行为

2. ActivityManagerProxy 对象–向 ActivityManagerNative 对象传递 RPC 代码（ATTACH_APPLICATION_TRANSACTION）与 Binder RPC 数据。

3. ActivityManagerNative 对象–调用 ActivityManagerService 的 attachApplication()方法。

那么，调用 ActivityThread 的 attach()方法究竟是如何通过 Binder RPC 调用 ActivityManager Service 的 attachApplication()方法的呢？下面通过代码分析，同大家一起研究一番，重点关注如何在 ActivityThread 与 ActivityManagerService 之间建立用于交互的 Binder 连接，类似于图 11-7。

（1） ActivityThread 对象–调用 attachApplication()方法

```
private final void attach(boolean system)
{
    if (!system) {
        IActivityManager mgr = ActivityManagerNative.getDefault();
        mgr.attachApplication(mAppThread);
    }
}
```

代码 11-13 | ActivityThread 类的 attach()方法

更多更新编程资源尽在 http://www.pin5i.com/

代码 11-13 是 ActivityThread 类的 attach()方法的主要代码，该方法用于在 Activity Thread 与 ActivityManagerService 之间建立基于 IActivityManager 接口的 Binder RPC 连接。当 Binder RPC 连接建立后，ActivityThread 就可以通过 ActivityManagerProxy 对象向 Activity Manager Service 请求服务。

如图 11-5 所示，调用 ActivityManagerNative.getDefault()方法将创建 ActivityManager Proxy 对象。因此代码 11-13 中的 mgr 变量持有 ActivityManagerProxy 对象的引用，调用 mgr.attach Application()方法将会调用 ActivityManagerProxy 对象的 attachApplication()方法（代码 11-14）。在代码 11-12 中生成 ActivityThread 时 mAppThread 成员变量中保存的 ApplicationThread 对象将以参数的形式传入 attachApplication()方法中。

（2） ActivityManagerProxy 对象

向 ActivityManagerNative 传递 RPC 代码（ATTACH_APPLICATION_TRANSACT ION）与 Binder RPC 数据。

```
public void attachApplication(IApplicationThread app)
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(app.asBinder());
    mRemote.transact(ATTACH_APPLICATION_TRANSACTION, data, reply, 0);
}
```

代码 11-14 | ActivityManagerProxy 类的 attachApplication()方法

代码 11-14 是 ActivityManagerProxy 对象的 attachApplication()方法的主要代码，它将 app 参数传递的与 ApplicationThread 相应的 Binder 对象进行 Mashalling 处理，而后将 RPC 代码（ATTACH_APPLICATION_TRANSACTION）与 Binder RPC 数据传递给 ActivityManagerNative 对象。

（3） ActivityManagerNative 对象–调用 attachApplication()方法

RPC 代码（ATTACH_APPLICATION_TRANSACTION）与 Binder RPC 数据是由 ActivityThread 发送的，并且它们由 ActivityManagerNative 对象的 onTransact()方法进行处理，如代码 11-15 所示。如图 11-8 所示，为了实现与 ActivityThread 间的 Binder RPC（基于 IApplicationThread 接口的）通信，Activity Manager Service 调用 ApplicationThreadNative. asInterface()方法创建 Application ThreadProxy 对象。

Activity Manager Service 使用生成的 ApplicationThreadProxy 对象控制 ActivityThread。ActivityThread 对象通过 attach()方法设置 Binder RPC 连接，以便 Activity Manager Service 进行控制。

更多更新编程资源尽在 http://www.pin5i.com/

```
public boolean onTransact(int code, Parcel data, Parcel reply, int ?ags)
{
   switch (code) {
                ...
   case ATTACH_APPLICATION_TRANSACTION: {
      IApplicationThread app = ApplicationThreadNative.asInterface(
                ➥ data.readStrongBinder());
      attachApplication(app);
      return true;
   }
                ...
   }
}
```

代码 11-15 | ActivityManagerNative 类的 onTransact()方法

当 ActivityThread 与 Activity Manager Service 之间的 Binder RPC 连接建立后，Activity Manager Service 的 attachApplication()方法即被调用。

11.2.5 Activity Manager Service——attachApplication()Stub 方法

首先来看一下 Activity Manager Service 的 attachApplication()方法的整个执行过程，如图 11-9 所示。该方法用来向 ActivityThread 请求生成应用程序服务，应用程序服务即 Controller Activity 调用 startService() API 方法请求运行的服务。

Controller 请求的服务实际是由 ActivityThread 生成的，并且 Activity Manager Service 的 attach Application()方法控制着这一过程，如图 11-9 所示。Activity Manager Service 通过基于 IApplicationThread 接口的 Binder RPC 控制着 Android 应用程序组件 Activity 及服务的生成，并管理它们的生命周期。

下面来分析图 11-9 的动作流程。在前面部分，为了实现 Activity Manager Service 与用于运行远程服务的 ActivityThread 之间的交互，已经设置好 Binder RPC 连接。在图 11-9 中，Activity Manager Service 通过 Binder RPC 将要生成的服务信息传递给 ActivityThread，生成并运行 RemoteService 服务。

整个过程从调用 Activity Manager Service 的 attachApplication()方法（基于 Iapplica tion Thread）开始共分为如下五个步骤。

（1） Activity Manager Service–调用 ActivityManagerProxy 对象的 scheduleCreateService() 代理方法。

（2） ActivityManagerProxy 对象–向 ActivityThread 的 ActivityManagerNative 对象传递 RPC 代码（SCHEDULE_CREATE_SERVICE_TRANSACTION）与 Binder

更多更新编程资源尽在 http://www.pin5i.com/

RPC 数据。

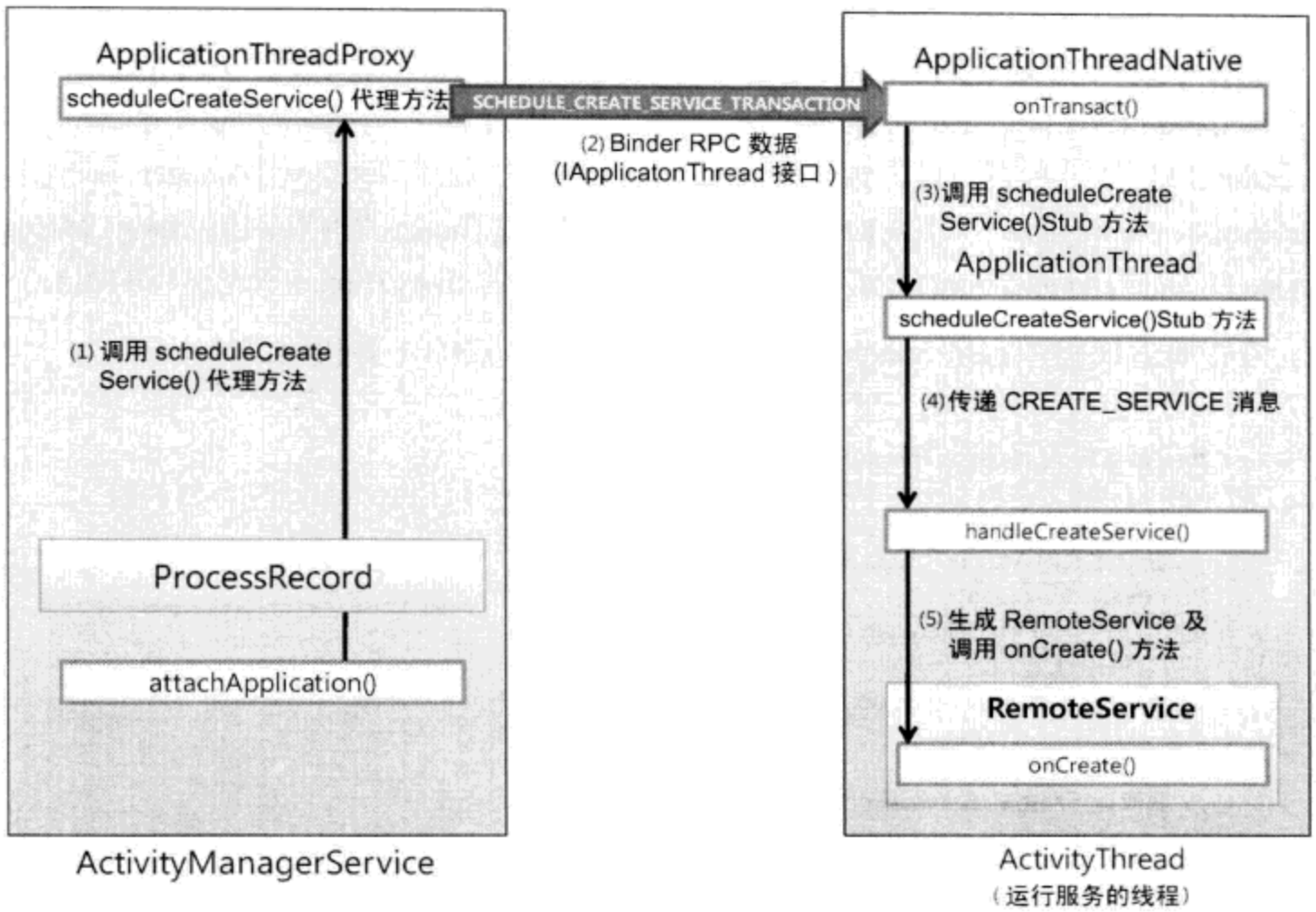


图 11-9 | ActivityManagerService 对象的 attachApplication()方法执行过程

（3） ActivityManagerNative 对象–调用 ApplicationCreateService 的 ApplicationThread 对象的 scheduleCreateService() Stub 方法。

（4） ApplicationThread 对象–通过消息队列，向 ApplicationCreateService 的 Activity Thread 传递 CREATE_SERVICE 消息。

（5） ActivityThread 对象–生成 RemoteService 服务及调用 onCreate()方法。

下面来分析一下程序代码，以便详细地了解 attachApplication() Stub 方法的动作行为，帮助读者进一步加深对图 11-9 的理解。

（1） Activity Manager Service–调用 scheduleCreateService()代理方法

```
public final void attachApplication(IApplicationThread thread)
{
   int callingPid = Binder.getCallingPid();
   attachApplicationLocked(thread, callingPid);
}
```

代码 11-16 | ActivityManagerService 类的 attachApplication()方法

更多更新编程资源尽在 http://www.pin5i.com/

代码 11-16 是 ActivityManagerService 中 attachApplication() Stub 方法的主要代码，它主要用于调用一个名称为 attachApplicationLocked()的方法（代码 11-17）。

attachApplication()在调用 attachApplicationLocked()方法时需要提供两个参数，第一个参数 thread 持有 ApplicationThreadProxy 对象（在代码 11-15 中生成）的引用，第二个参数 callingPid 是进程的 pid，该进程通过 Binder RPC 调用 Activity Manager Service 的 attachApplication()方法。在图 11-8 中 ActivityThread 通过 Binder RPC 调用 Activity Manager Service 的 attach Application()方法，此时 callingPid 即是 ActivityThread 所在进程的 pid。

请看如下代码，它是 attachApplicationLocked()方法的主要代码。

```
// thread 指向 ApplicationThreadProxy 对象
private final boolean attachApplicationLocked(IApplicationThread thread, int pid)
{
    // 获取生成的 ActivityThread 的 pid 值的 ProcessRecord
    ProcessRecord app;
    app = mPidsSelfLocked.get(pid);

    // 连接 ProcessRecord 和 ApplicationThreadProxy 对象
    app.thread = thread;

    // 获取运行服务的 ServiceRecord
    ServiceRecord sr = null;
    for (int i=0; i<mPendingServices.size(); i++) {
        sr = mPendingServices.get(i);
        mPendingServices.remove(i);
        i--;

        realStartServiceLocked(sr, app);
    }

    return true;
}
```

代码 11-17 | ActivityManagerService 的 attachApplicationLocked()方法

在 attachApplicationLocked()方法中，首先使用 ActivityThread 的 pid 值为键，调用 mPidsSelf Locked.get()哈希表函数，获取与其相对应的 ProcessRecord 对象（在代码 11-11 中，Zygote 生成进程后，将返回的 pid 与 ProcessRecord 对象以键值对的形式保存到哈希表中）。

我们需要获取与 ActivityThread 的 pid 相对应的 ProcessRecord 对象，以便将其与 Application ThreadProxy 对象映射在一起，如代码所示。Activity Manager Service 使用 pid 从 mPidsSelf Locked 哈希表中获取相应的 ProcessRecord 对象，而后通过该对象的 thread 成员变量，获取与相同 pid 的 ActivityThread 连接在一起的 ApplicationThreadProxy

更多更新编程资源尽在 http://www.pin5i.com/

对象，如图 11-10 所示。

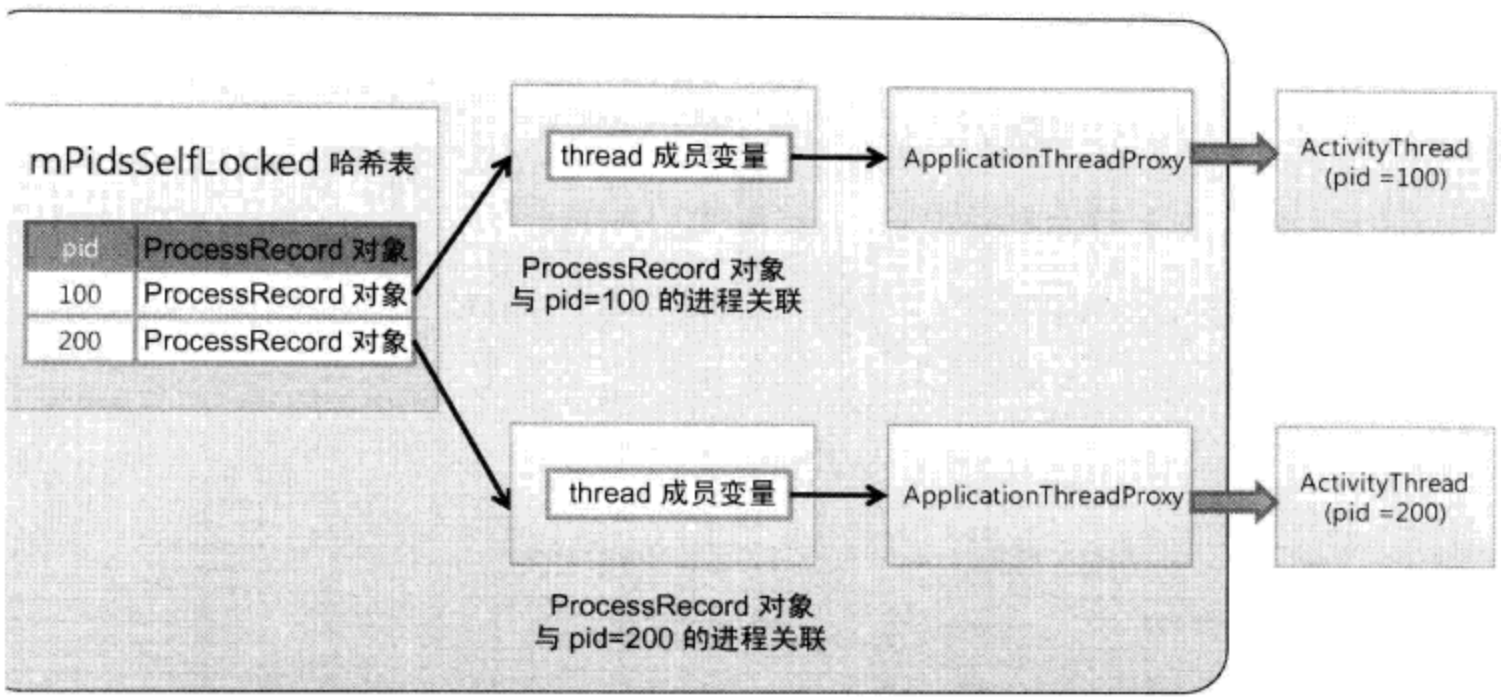


图 11-10 | 通过 pid 获取与指定进程相映射的 ApplicationThreadProxy 对象

如前所言，Activity Manager Service 使用 ApplicationThreadProxy 对象控制与其连接在一起的 ActivityThread。Activity Manager Service 是控制应用程序组件的系统服务，这些应用程序组件运行在 Android 系统的各种 ActivityThread 中，所以必须获取与想要控制的 ActivityThread 相关联的 ApplicationThreadProxy 对象。如代码所示，当将 ApplicationThread Proxy 对象连接到包含各种进程信息的 ProcessRecord 对象后，Activity Manager Service 即可获取想要控制的应用程序的 pid，而后获取与其相对应的 Application ThreadProxy 对象。

假设 Activity Manager Service 要控制 Activity1 的生命周期，如图 11-11 所示。为此，Activity Manager Service 首先获取与 ActivityThread（pid=100）关联的 ProcessRecord 对象，而后通过 thread 变量获取 ApplicationThreadProxy 对象，再向 ActivityThread 发送 Binder RPC 命令即可。

以上我们了解了 Activity Manager Service 将 ApplicationThreadProxy 连接至 Process Record 的原因。那么，在运行 Activity Manager Service 请求的应用程序服务时，如何获取保存服务相关信息的 ServiceRecord 结构体呢？在代码 11-10 中，bringUpService Locked()方法会将 Activity Manager Service 要运行的服务（这里是指 RemoteService）的 ServiceRecord 对象保存到 mPendingServices 队列中。在代码中通过调用 mPending Services.get()方法获取保存在队列中的 ServiceRecord 对象。

在获取 ProcessRecord 与 ServiceRecord 对象后，它们作为参数传入 realStartService Locked()方法中，如代码 11-18 所示。

更多更新编程资源尽在 http://www.pin5i.com/

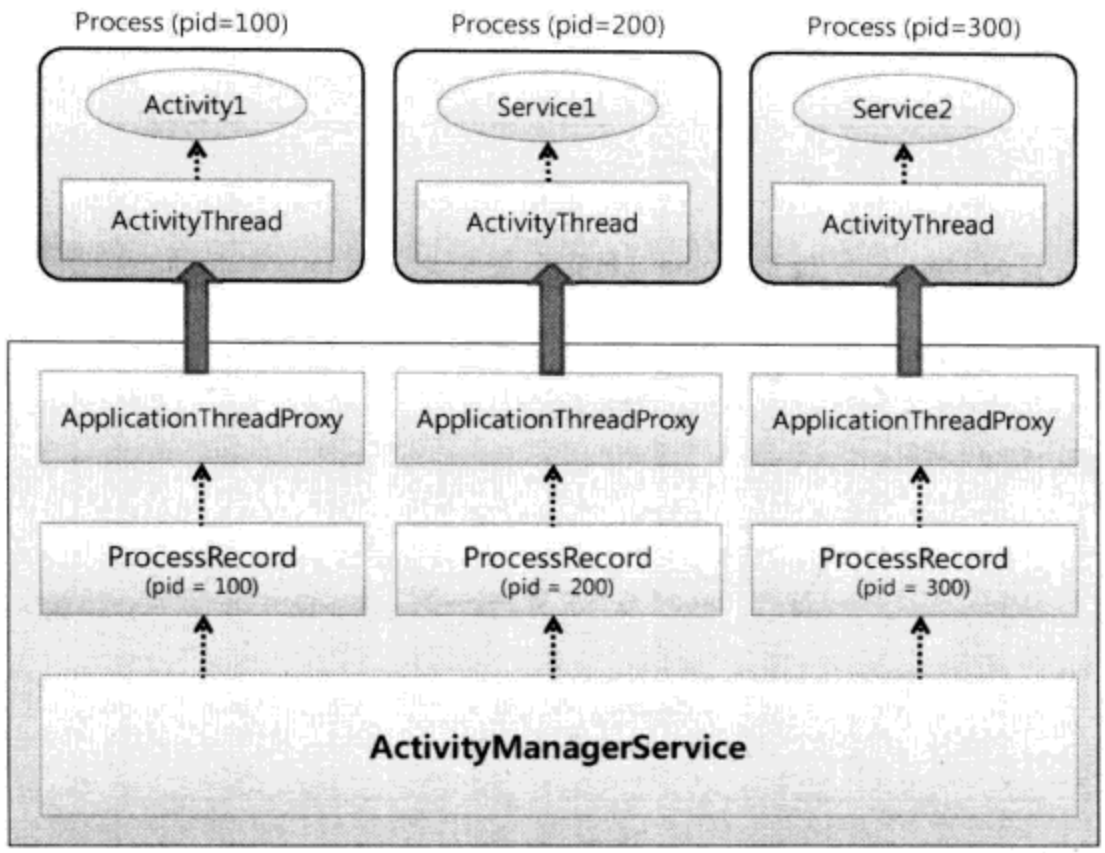


图 11-11 | ProcessRecord 与 ApplicationThreadProxy 的关系

```
private final void realStartServiceLocked(ServiceRecord r, ProcessRecord app)
{
   app.thread.scheduleCreateService(r, r.serviceInfo);
}
```

代码 11-18 | ActivityManagerService 的 realStartServiceLocked()方法

如代码 11-18 所示，在 realStartServiceLocked()方法内部仅调用了 app.thread.schedule Create Service()方法。app.thread 中保存着 ApplicationThreadProxy 对象，它用来控制请求服务运行的 ActivityThread。因此调用 app.thread.scheduleCreateService()方法，最终调用的是 Application ThreadProxy 的 scheduleCreateService()方法，此时含有服务信息的 ServiceRecord 对象将作为参数传入方法之中。

（2） ApplicationThreadProxy 对象–传递 Binder RPC 数据

```
public final void scheduleCreateService(IBinder token, ServiceInfo info)
{
   Parcel data = Parcel.obtain();
   data.writeStrongBinder(token);
   info.writeToParcel(data, 0);
   mRemote.transact(SCHEDULE_CREATE_SERVICE_TRANSACTION, data, null, IBinder.FLAG_ONEWAY);
}
```

代码 11-19 | ApplicationThreadProxy 类的 scheduleCreateService()方法

更多更新编程资源尽在 http://www.pin5i.com/

调用 ApplicationThreadProxy 对象的 scheduleCreateService()代理方法后，ServiceInfo 对象中有关 RemoteService 的信息即会通过 RPC 代码（SCHEDULE_CREATE_SERVICE_TRANSACTION）与 Binder RPC 数据传递给 ApplicationThreadNative 对象，如图 11-9 所示。

（3） ApplicationThreadNative 对象–调用 scheduleCreateService() Stub 方法

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
{
   switch (code) {
   ...
   case SCHEDULE_CREATE_SERVICE_TRANSACTION: {
      IBinder token = data.readStrongBinder();
      ServiceInfo info = ServiceInfo.CREATOR.createFromParcel(data);
      scheduleCreateService(token, info);
      return true;
   }
   ...
   }

   return super.onTransact(code, data, reply, ?ags);
}
```

代码 11-20 | ApplicationThreadProxy 类的 onTransact()方法

ApplicationThreadNative 对象接收来自 ApplicationThreadProxy 对象的 Binder RPC 数据，并调用 onTransact()方法进行处理。

接收数据后，onTransact()方法先对来自 ApplicationThreadProxy 对象的 Service Record 与 ServiceInfo 对象进行 UnMarshalling 处理，而后分别保存到 token 与 info 变量中，再将它们作为参数，调用 ActivityThread 的 scheduleCreateService()方法。

（4） ApplicationThread 对象–向 ActivityThread 传递 CREATE_SERVICE 消息

```
public final void scheduleCreateService(IBinder token, ServiceInfo info)
{
   CreateServiceData s = new CreateServiceData();
   s.token = token;
   s.info = info;
   queueOrSendMessage(H.CREATE_SERVICE, s);
}
```

代码 11-21 | ApplicationThread 类的 scheduleCreateService()方法

在 scheduleCreateService() Stub 方法内部，首先创建一个 CreateServiceData 对象，将参数传递而来的 ServiceRecord 与 ServiceInfo 对象保存到其中，而后调用 queue OrSend Message()

更多更新编程资源尽在 http://www.pin5i.com/

方法以 CREATE_SERVICE 形式将其传递到 ActivityThread 的消息队列中。

以上整个过程如图 11-12 所示。

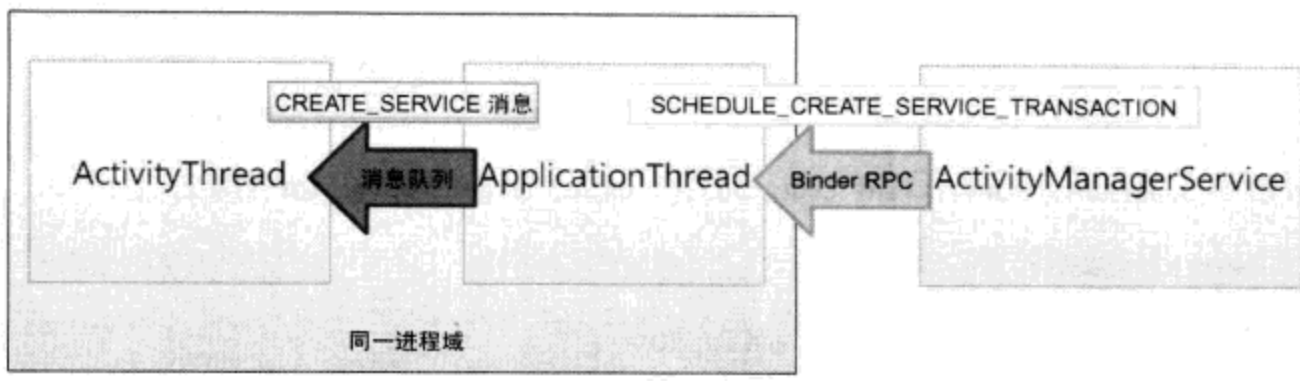


图 11-12 | 处理 SCHEDULE_CREATE_SERVICE_TRANSACTION

ApplicationThread 只是用于通过 Binder RPC 接收 Activity Manager Service 的控制命令，实际运行 Activity Manager Service 请求的服务以及生命周期的管理都是由 Activity Thread 进行的，ApplicationThread 与 ActivityThread 运行在同一进程域中，它们之间使用消息队列进行通信。

（5） ActivityThread 对象–生成服务并调用服务的 onCreate()方法

在前面代码 11-12 中已经讲解过，ActivityThread 自身拥有消息队列，并拥有一个 handle Message()方法，用来处理来自 ApplicationThread 的消息。ApplicationThread 通过 Binder RPC 数据接收来自 Activity Manager Service 的命令，而后使用消息队列将其传递给 ActivityThread。

当消息为 CREATE_SERVICE 时，handleMessage()会调用 handleCreateService()方法，创建服务实例，如代码 11-22 所示。

```
public void handleMessage(Message msg)
{
   switch (msg.what) {
   ...
   case CREATE_SERVICE:
      handleCreateService((CreateServiceData)msg.obj);
      break;
   ...

   }
}

private final void handleCreateService(CreateServiceData data)
{
   //生成服务实例
   PackageInfo packageInfo = getPackageInfoNoCheck(data.info.applicationInfo);
   Service service = null;
   java.lang.ClassLoader cl = packageInfo.getClassLoader();
```

更多更新编程资源尽在 http://www.pin5i.com/


```
        service = (Service) cl.loadClass(data.info.name).newInstance();

        //服务的生命周期开始了
        service.onCreate();
    }
```

代码 11-22 | 处理 CREATE_SERVICE 消息的主要代码

如代码所示，handleCreateService()方法拥有一个 CreateServiceData 类型的参数，该参数对象中含有应用程序服务类（这里指 RemoteService）。handleCreateService()方法首先加载参数指定的类，而后生成类的实例对象。在生成服务实例对象后，根据服务的生命周期，调用应用程序服务中实现的 onCreate()回调方法。

11.3 小结

本章以 ApiDemos 为实例分析了 Activity 通过 Activity Manager Service 创建应用程序服务的过程。同时还通过调用 onCreate()回调方法的方式学习了 Activity Manager Service 控制 Android 组件生命周期的原理。

请看图 11-13，它简单地描述了创建服务的整个过程，其中步骤（1）（3）（4）是以 Binder RPC 的形式调用的接收端的方法，在图中我们只是简单地标注出了调用的方法名。

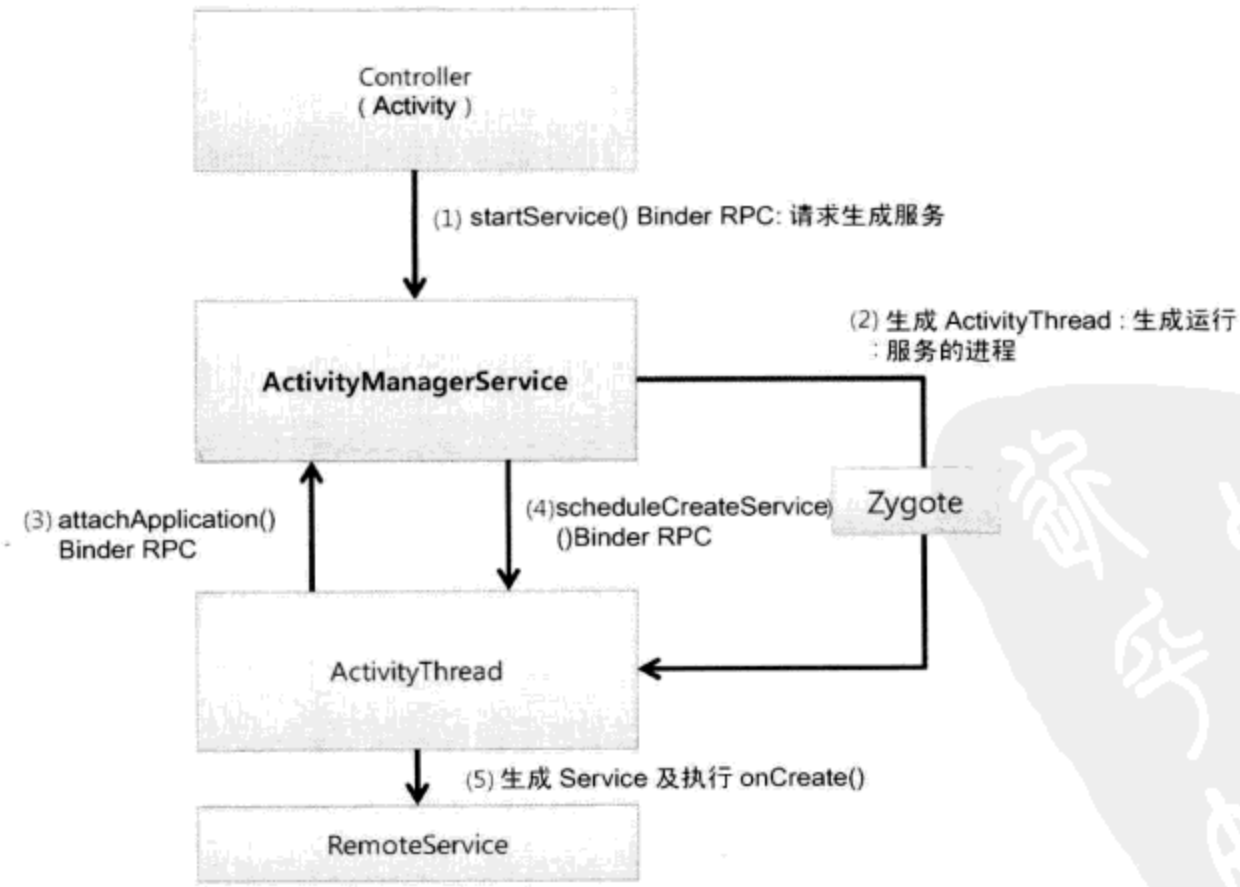


图 11-13 | Activity Manager Service 生成服务的过程

更多更新编程资源尽在 http://www.pin5i.com/

以上整个过程可以简单地分为如下五个步骤。

（1） 为了运行 RemoteService 服务，Controller Activity 通过 startService() API 向 Activity Manager Service 请求运行 RemoteService 服务。

（2） 若请求的服务是远程服务，Activity Manager Service 向 Zygote 请求生成 Activity Thread，以便服务运行在独立的进程中。

（3） ActivityThread 由 Zygote 生成后，通过 attachApplication()代码方法，向 Activity Manager Service 注册，以便 Activity Manager Service 进行控制。

（4） Activity Manager Service 向 ActivityThread 请求生成 RemoteService 服务。

（5） ActivityThread 生成请求的服务 RemoteService 实例，而后调用该服务的 onCreate()回调函数。

更多更新编程资源尽在 http://www.pin5i.com/

附录

AIDL 语法

AIDL 语言的语法被定义在 frameworks\base\tools\aidl 目录下名称为 aidl_language_y.y 的 yacc 文件中。aidl_language_y.y 文件内容如下：

```
//token
IMPORT, PACKAGE, IDENTIFIER, GENERIC, ARRAY,
parcelable, interface, in, out, inout, oneway
//AIDL Syntax
document:
        document_items
    |   headers document_items

headers:
        package
    |   imports
    |   package imports

package:
        PACKAGE

imports:
        IMPORT
    |   IMPORT imports

document_items:

    | document_items declaration
    | document_items error

declaration:
        parcelable_decl
    |   interface_decl

parcelable_decl:
        PARCELABLE IDENTIFIER ';'
    |   PARCELABLE ';'
    |   PARCELABLE error ';'

interface_header:
        INTERFACE
```

更多更新编程资源尽在 http://www.pin5i.com/

```
    |   ONEWAY INTERFACE

interface_decl:
        interface_header IDENTIFIER '{' interface_items '}'
    |   INTERFACE error '{' interface_items '}'
    |   INTERFACE error '}'

interface_items:

    |   interface_items method_decl
    |   interface_items error ';'

method_decl:
        type IDENTIFIER '(' arg_list ')' ';'
    |   ONEWAY type IDENTIFIER '(' arg_list ')' ';'

arg_list:

    |   arg
    |   arg_list ',' arg
    |   error

arg:
        direction type IDENTIFIER

type:
        IDENTIFIER
    |   IDENTIFIER ARRAY
    |   GENERIC

direction:

    |   IN
    |   OUT
    |   INOUT
```

代码-定义在 aidl_language_y.y 文件中的 AIDL 语法

更多更新编程资源尽在 http://www.pin5i.com/

Android框架揭秘
Inside the Android Framework

也许你曾想，即使不了解Android内核框架的秘密，自己依旧可以编写出Android应用程序。但是你也必须承认，如果深入了解、学习了Android的框架知识，并能熟练掌握框架内部的运行原理，就一定能够设计、编写出更加多样、执行效率更高、更健壮的应用程序。

Android是开源系统，越来越多的硬件厂商开始采用这一系统推出独具特色的手机产品。在这些个性化、特色化的产品中，定制系统是相当重要的一环。只有在深入而准确地分析、研究Android框架之后，我们才能定制开发出符合我们自身特色的系统。因此，是否深入理解Android框架，是我们在进行开发时差异化战略的关键步骤之一。

本书从分析Android系统源码着手，深入研究分析了Android框架的内部运行原理与机制。如果你想深入了解学习Android框架的有关知识，想进一步提高自身的应用程序开发水平，那么强烈建议你阅读本书内容，本书将引入你到达“幸福”的彼岸。

本书涵盖内容

- Android框架概要，了解启动过程
- 移植Android以及如何开发适合不同型号手机的应用程序
- 分析Android框架所需的基础知识 ——JNI（Java Native Interface）与Binder
- 分析Zygote、Service Manager、Service Server等核心组件
- Android Service Framework的组成与理解
- Camera Service、Activity Manager Service等服务分析

ISBN 978-7-115-27476-2

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

封面设计：任文杰

ISBN 978-7-115-27476-2
定价：69.00 元