深入理解 Android 5 源代码

李骏◆编著

基于新版 Android 5 源代码

全面讲解了 Android 5 源程序的核心技术，包括 Java Native Interface 系统、HAL 系统、IPC 通信机制、Binder 对象和 Java 接口、init 进程和 Zygote 进程、System 进程和应用程序进程、Activity 组件、应用程序管理服务、Content Provider 系统、Broadcast 系统、电源管理系统、电话系统、短信系统、传感器系统、SEAndroid 系统和 ART 系统等核心知识。

中国工信出版集团

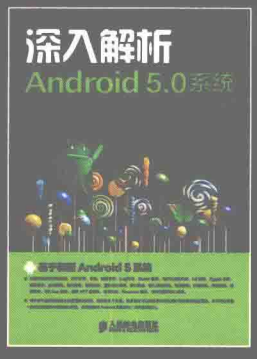
人民邮电出版社
POSTS & TELECOM PRESS

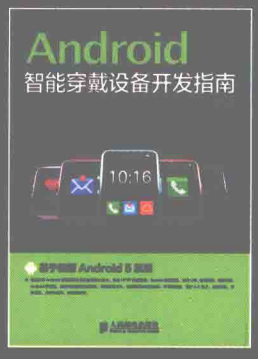
深入理解
Android 5 源代码

作者简介

李骏，清华大学电子信息工程专业学士。有多年的Android开发经验，熟悉Android底层结构和Linux驱动开发，有着丰富的Android底层和驱动层优化、移植开发经验，擅长利用JNI技术开发Android上的应用程序。曾带领团队利用NDK技术成功开发过具有库仑计电池芯片的电池管理软件，以及在Android上成功移植人脸识别程序，目前在凹凸电子担任Android架构师。

畅销书推荐





异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：董志桢

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

ISBN 978-7-115-40595-1

ISBN 978-7-115-40595-1
定价：99.00 元

深入理解 Android 5源代码

李骏◆编著

人民邮电出版社
北京

图书在版编目（CIP）数据

深入理解Android 5源代码 / 李骏编著. -- 北京 : 人民邮电出版社, 2016.1
ISBN 978-7-115-40595-1

Ⅰ. ①深… Ⅱ. ①李… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2015)第260754号

内 容 提 要

本书共分 20 章，循序渐进地分析了 Android 系统的基本源代码，依次讲解了 Android 系统介绍，获取并编译 Android 源代码，分析 Java Native Interface 系统，分析 HAL 系统，分析 IPC 通信机制，分析 Binder 对象和 Java 接口，分析 ServiceManager 和 MessageQueue，init 进程和 Zygote 进程，System 进程和应用程序进程，分析 Activity 组件，应用程序管理服务分析，Content Provider、Broadcast（广播）系统，电源管理系统分析，分析 WindowManagerService 系统、分析电话系统，分析短信系统、Sensor 传感器系统详解、分析 SEAndroid 系统和分析 ART 系统等核心知识。本书内容言简意赅，讲解方法通俗易懂，不仅适合有一定基础的读者学习，也特别有利于初学者学习。

本书适合 Android 初学者、Android 爱好者、Android 底层开发人员、Android 应用开发人员学习，也可以作为相关培训学校和大专院校相关专业师生的教学用书。

◆ 编　　著　李　骏
责任编辑　张　涛
责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：787×1092　1/16
印张：42.75
字数：1306 千字　　2016 年 1 月第 1 版
印数：1－2 500 册　　2016 年 1 月北京第 1 次印刷

定价：99.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

前　言

Android 是一款基于 Linux 平台的开源手机操作系统，该平台由操作系统、中间件、用户界面和应用软件组成，号称是首个为移动终端打造的真正开放的操作系统。

本书的内容

Android 系统是 Google 研发团队集体智慧的结晶，拥有着海量的源代码。本书受篇幅的限制，只分析了 Android 系统中的一些主要模块和类，并对主要模块的细节进行了全面分析，而相似部分并没有进行详细阐述。这样可以确保读者在有限的篇幅中了解 Android 的内部结构和运行机制，同时避免读者陷入海量代码的云雾中而不得要领的情况发生。

另外，由于 Android 系统升级较快，有些代码变动很大。虽然 Android 系统自 2008 年 9 月发布第一个版本 1.1 以来，截至 2014 年 10 月发布最新版本 5.0，一共存在十多个版本。在本书的讲解中，选择了本书写作时的最新版本 Android 5.0 系统，这样可以体验 Android 系统的最新功能。

全书共分 20 章，涵盖了 Android 系统主要的源代码，如 HAL 系统、IPC 通信机制、Binder 对象、init 进程和 Zygote 进程、System 进程和应用程序进程、Activity 组件、Content Provider、Broadcast、电源管理、电话系统、短信系统、传感器、SEAndroid、ART 等核心技术。为了帮助读者学以致用，对于重点的模块都会详细剖析其原理和实现的过程，以便读者在自己的项目开发中可以借鉴。

本书特色

本书内容丰富，分析细致，我们的目标是通过一本图书，提供多本图书的价值。在内容的编写上，本书具有以下特色。

（1）结构合理

从用户的实际需要出发，科学安排知识结构，内容由浅入深。全书详细地讲解了和 Android 应用开发有关的源代码结构。

（2）易学易懂

本书条理清晰、语言简洁，可帮助读者快速掌握每个知识点。读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行有针对性的学习。

（3）实用性强

本书彻底摒弃枯燥的理论，注重实用性和可操作性，详细讲解了各个知识点的基本知识。

读者对象

初学 Android 编程的自学者；

大中专院校的老师和学生；

毕业设计的学生；

Android 编程爱好者；

相关培训机构的老师和学员；

从事 Android 开发的程序员。

本书在编写过程中，得到了人民邮电出版社工作人员的大力支持，正是各位编辑的求实、耐心和效率，才使得本书在这么短的时间内出版。另外，也十分感谢我的家人，在我写作的时候给予的巨大支持。另外，本人水平毕竟有限，书中的纰漏和不尽如人意之处在所难免，诚请读者提出意见或建议，以便修订并使之更臻完善。另外，本书的答疑和技术交流网站为 http://www.toppr.net/，读者如有疑问可以在此提出，一定会得到满意的答复。

作　者

目　录

第 1 章　Android 系统介绍

2007 年，Google 公司推出了一款移动智能设备系统——Android，这是一种建立在 Linux 基础之上的为手机、平板电脑等移动设备提供的软件解决方案。截止到 2013 年，根据知名 IDC 公司的统计，Android 系统在世界智能手机发货量中占据 75%的份额，已经成为了当今最受欢迎的智能设备系统之一，2014 年更是达到了 84.4%。在本章将引领读者一起来了解 Android 系统的发展历程，充分了解这款操作系统的成功之处。

1.1 Android 系统成功的秘诀

Android 系统为什么能够在这么短的时间内成为移动智能设备市场占有率的老大？在本节的内容中，将从 4 个方面来为读者解答这个问题。

1.1.1　获取了业界的广泛支持

Android 系统基于 Linux 内核，是一款开源的手机操作系统。正是因为如此，在 Android 刚刚崭露头角之后，各大手机厂商和电信部门纷纷加入到了 Android 联盟当中。Android 联盟由业界内的世界级企业组成，主要成员包括中国移动、摩托罗拉、高通、T-Mobile、三星、LG、HTC 等在内的 30 多家技术和无线应用的领军企业。Android 通过与运营商、设备制造商、开发商和其他有关各方结成深层次的合作伙伴关系，希望借助建立标准化、开放式的移动电话软件平台，在移动产业内形成一个开放式的生态系统。

1.1.2　研发阵容强大

Android 的研发队伍阵容强大，包括摩托罗拉、Google、HTC（宏达电子）、PHILIPS、T-Mobile、高通、三星、LG 及中国移动在内的 34 家企业，他们都将基于该平台开发手机的新型业务，应用之间的通用性和互联性将在最大程度上得到保持。

1.1.3　为开发人员“精心定制”

Google 公司一直视程序员为前进动力的源泉，为了提高程序员们的开发积极性，不但为开发人员提供了一流的开发装备和软件服务，而且还提出了振奋人心的奖励机制。

（1）保证开发人员可以迅速转型到 Android 应用开发

Android 应用程序是通过 Java 语言开发的，只要具备 Java 开发基础，就能很快上手。作为单独的 Android 应用开发，对 Java 编程门槛的要求并不高，即使没有编程经验的初学者，也可以在突击学习 Java 之后平滑地过渡到 Android 开发上来。另外，Android 完全支持 2D、3D 和数据库，并且和浏览器实现了集成。所以，通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用。

（2）定期召开奖金丰厚的 Android 大赛

为了吸引更多的用户使用 Android 开发，Google 已经成功举办了奖金为数千万美元的开发者竞赛。鼓励开发人员创建出创意十足、十分有用的软件。这种大赛对于开发人员来说，不但能提高自己的开发水平，并且高额的奖金也是学员们学习的动力。

（3）开发人员可以利用自己的应用赚钱

为了能让 Android 平台吸引更多的关注，Google 提供了一个专门下载 Android 应用的商店：

Android Market，地址是 https://play.google.com/store。在这个商店里允许开发人员发布应用程序，也允许 Android 用户下载获取自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且可以对自己的软件进行定价。只要你的软件程序足够吸引人，你就可以获得很好的金钱回报。这样实现了程序员学习和赚钱两不误，所以吸引了更多开发人员加入到 Android 大军中来。

1.1.4　开源

Android 是一款开源的系统，开源意味着对开发人员和手机厂商来说是完全无偿免费使用的。正是因为这一原因，吸引了全世界各地无数程序员的热情。于是很多手机厂商都纷纷采用 Android 作为自己产品的系统。因为免费所以降低了成本，提高了利润。而对于开发人员来说，因为 Android 深受众多移动设备产品所采用，所以这方面的人才也变得愈发得到重视。

1.2　剖析 Android 系统架构

Android 系统是一个移动设备的开发平台，其软件层次结构包括操作系统（OS）、中间件（MiddleWare）和应用程序（Application）。根据 Android 的软件框图，其软件层次结构自下而上分为以下 4 层。

（1）操作系统层（OS）。

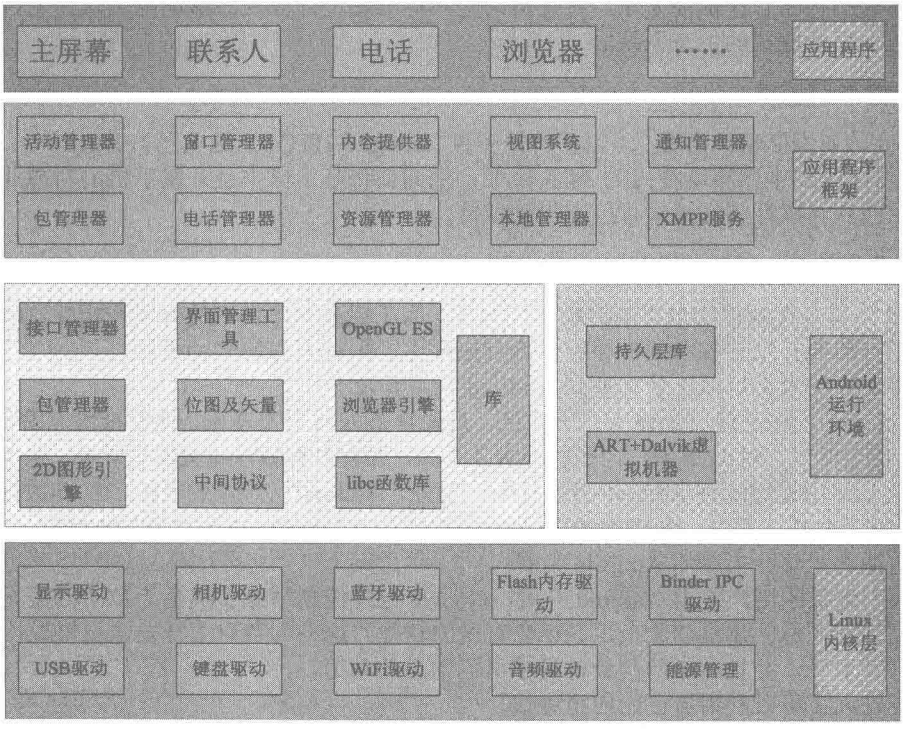
（2）各种库（Libraries）和 Android 运行环境（RunTime）。

（3）应用程序框架（Application Framework）。

（4）应用程序（Application）。

上述各个层的具体结构如图 1-1 所示。

在本节的内容中，将详细介绍 Android 操作系统的组件结构方面的知识。



▲图 1-1　Android 操作系统的组件结构图

1.2.1 底层操作系统层（OS）

因为 Android 源于 Linux，使用了 Linux 内核，所以，Android 使用 Linux 2.6 作为操作系统的基础。Android 对操作系统的使用包括核心和驱动程序两部分，Android 内核对应于 Linux 内核，Android 更多的是需要一些与移动设备相关的驱动程序。主要的驱动如下所示。

- 显示驱动（Display Driver）：是常用的基于 Linux 的帧缓冲（Frame Buffer）驱动。
- Flash 内存驱动（Flash Memory Driver）：是基于 MTD 的 Flash 驱动程序。
- 照相机驱动（Camera Driver）：常用基于 Linux 的 V4L（Video for Linux）驱动。
- 音频驱动（Audio Driver）：常用基于 ALSA（Advanced Linux Sound Architecture，高级 Linux 声音体系）驱动。
- WiFi 驱动（Camera Driver）：基于 IEEE 802.11 标准的驱动程序。
- 键盘驱动（KeyBoard Driver）：作为输入设备的键盘驱动。
- 蓝牙驱动（Bluetooth Driver）：基于 IEEE 802.15.1 标准的无线传输技术。
- Binder IPC 驱动：Andoid 一个特殊的驱动程序，具有单独的设备节点，提供进程间通信的功能。
- Power Management（能源管理）：用于管理电池电量等信息。

1.2.2 各种库（Libraries）和 Android 运行环境（RunTime）

本层次对应一般嵌入式系统，相当于中间件层次。Android 的本层次分成两个部分，一个是各种库，另一个是 Android 运行环境。本层的内容大多是使用 C 实现的，其中包含了如下所示的各种库。

- C 库：C 语言的标准库，也是系统中一个最为底层的库，C 库是通过 Linux 的系统调用来实现。
- 多媒体框架（MediaFrameword）：这部分内容是 Android 多媒体的核心部分，基于 PacketVideo（即 PV）的 OpenCORE，从功能上看本库一共分为两大部分，一部分是音频、视频的回放（PlayBack），另一部分是音、视频的记录（Recorder）。
- SGL：2D 图像引擎。
- SSL：即 Secure Socket Layer，位于 TCP/IP 协议与各种应用层协议之间，为数据通信提供安全支持。
- OpenGL ES：提供了对 3D 的支持。
- 界面管理工具（Surface Management）：提供了管理显示子系统等功能。
- SQLite：一个通用的嵌入式数据库。
- WebKit：网络浏览器的核心。
- FreeType：位图和矢量字体的功能。

在一般情况下，Android 的各种库是以系统中间件的形式提供的，它们的显著特点是与移动设备平台的应用密切相关。另外，Android 的运行环境主要是指 Dalvik（虚拟机）技术。Dalvik 和一般的 Java 虚拟机（Java VM）是有区别的。

- Java 虚拟机：执行的是 Java 标准的字节码（Bytecode）。
- ART+Dalvik：执行的是 Dalvik 可执行格式（.dex）中的执行文件。在执行的过程中，每一个应用程序即一个进程（Linux 的一个 Process）。二者最大的区别在于 Java VM 是以基于栈的虚拟机（Stack-based），而 Dalvik 是基于寄存器的虚拟机（Register-based）。显然，后者最大的好处在于可以根据硬件实现更大的优化，这更适合移动设备的特点。从 Android 5.0 版本开始，Android 的默认运行环境为 ART。ART 的机制与 Dalvik 不同。在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT，Ahead-Of-Time）。这样改进后，应用的启动（首次）和执行都会变得更加快速。

1.2.3 Application Framework（应用程序框架）

在整个 Android 系统中，和应用开发最相关的是 Application Framework，在这一层，Android

为应用程序层的开发者提供了各种功能强大的 APIs，这实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的。在本层提供了程序中所需要的各种控件，例如，视图组件（Views）、列表（List）、栅格（Grid）、文本框（Text Box）、按钮（Button），甚至还有一个嵌入式的 Web 浏览器。

一个基本的 Andoid 应用程序可以利用应用程序框架中的以下 5 个部分。

- Activity：活动。
- Broadcast Intent Receiver：广播意图接收者。
- Service：服务。
- Content Provider：内容提供者。
- Intent and Intent Filter：意图和意图过滤器。

1.2.4　顶层应用程序（Application）

Android 的应用程序主要是用户界面（User Interface）方面的，本层通常使用 Java 语言编写，其中还包含各种被放置在“res”目录中的资源文件构成。Java 程序和相关资源在经过编译后，会生成一个 APK 包。Android 本身提供了主屏幕（Home）、联系人（Contact）、电话（Phone）、浏览器（Browers）等众多的核心应用。同时应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序。这也是 Android 开源的巨大潜力的体现。

1.3　五大组件

在分析 Android 的源代码之前，很有必要先了解一下 Android 应用程序的核心组件功能。一个典型的 Android 应用程序通常由 5 个组件组成，这 5 个组件构成了 Android 的核心功能。在本节的内容中，将详细讲解这五大组件的基本知识。

1.3.1　Activity 界面

Activities 是这 5 个组件中最常用的一个组件。程序中 Activity 通常的表现形式是一个单独的界面（screen）。每个 Aactivity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。大多数程序有多个 Activity。例如，一个文本信息程序有这么几个界面：显示联系人列表界面、写信息界面、查看信息界面或者设置界面等。每个界面都是一个 Activity。切换到另一个界面就是载入一个新的 Activity。某些情况下，一个 Activity 可能会给前一个 Activity 返回值，例如，一个让用户选择相片的 Activity 会把选择到的相片返回给其调用者。

打开一个新界面后，前一个界面就被暂停，并放入历史栈中（界面切换历史栈）。使用者可以回溯前面已经打开的存放在历史栈中的界面。也可以从历史栈中删除没有价值的界面。Android 在历史栈中保留程序运行产生的所有界面：从第一个界面到最后一个。

1.3.2　Intent 和 Intent Filters 切换

Android 通过一个专门的 Intent 类来进行界面的切换。Intent 描述了程序想做什么（Intent 意为意图、目的、意向）。Intent 类还有一个相关类 IntentFilter。Intent 是一个请求来做什么事情，Intent Filters 则描述了一个 Activity（或下文的 Intent Receiver）能处理什么意图。显示某人联系信息的 Activity 使用了一个 IntentFilter，就是说它知道如何处理应用到此人数据的视图（View）操作。Activities 在文件 AndroidManifest.xml 中使用 Intent Filters。

通过解析 Intents 可以实现 Activity 的切换，可以使用 startActivity(myIntent)启用新的 Activity。系统会考察所有安装程序的 Intent Filters，然后找到与 myIntent 匹配最好的 Intent Filters 所对应的 Activity。这个新 Activity 能够接收 Intent 传来的消息，并因此被启用。解析 Intents 的过程发生在 startActivity 被实时调用时，这样做有如下两个好处。

（1）Activities 仅发出一个 Intent 请求，便能重用其他组件的功能。

（2）Activities 可以随时被替换为有等价 IntentFilter 的新 Activity。

1.3.3 Service（服务）

Service 是一个没有 UI 且长驻系统的代码，最常见的例子是媒体播放器从播放列表中播放歌曲。在媒体播放器程序中，可能有一个或多个 Activities 让用户选择播放的歌曲。然而在后台播放歌曲时无需 Activity 干涉，因为用户希望在音乐播放的同时能够切换到其他界面。既然这样，媒体播放器 Activity 需要通过 Context.startService()启动一个 Service，这个 Service 在后台运行以保持继续播放音乐。在媒体播放器被关闭之前，系统会保持音乐在后台播放。可以用 Context.bindService()方法连接到一个 Service 上（如果 Service 未运行的话，连接后还会启动它），连接后就可以通过一个 Service 提供的接口与 Service 进行通话。对音乐 Service 来说，提供了暂停和重放等功能。

1. 如何使用服务

在 Android 系统中，有如下两种使用 Service 的方法。

（1）通过调用 Context.startServece()启动服务，调用 Context.stoptService()结束服务，startService()可以传递参数给 Service。

（2）通过调用 Context.bindService()启动，调用 Context.unbindService()结束，还可以通过 ServiceConnection 访问 Service。二者可以混合使用，例如，可以先 startServece()再 unbindService()。

2. Service 的生命周期

在使用 startService()方法启动服务后，即使调用 startService()的进程结束了，Service 还仍然存在，一直到有进程调用 stoptService()或 Service 自己灭亡（stopSelf()）为止。

在 bindService()后，Service 就和调用 bindService()的进程同生共死，也就是说当调用 bindService()的进程死了，那么它绑定的 Service 也要跟着被结束，当然期间也可以调用 unbindService()让 Service 结束。

当混合使用上述两种方式时，例如，你用了 startService()，我用了 bindService()，那么只有你 stoptService()了而且我也 unbindService()了，这个 Service 才会被结束。

3. 进程生命周期

在 Android 系统中，会尝试保留那些启动了的或者绑定了的服务进程，具体规则如下所示。

（1）如果该服务正在进程的 onCreate()、onStart()或者 onDestroy()这些方法中执行时，那么主进程将会成为一个前台进程，以确保此代码不会被停止。

（2）如果服务已经开始，那么它的主进程的重要性会低于所有的可见进程，但是会高于不可见进程。由于只有少数几个进程是用户可见的，所以只要不是内存特别少，该服务就不会停止。

（3）如果有多个客户端绑定了服务，只要客户端中的一个对于用户是可见的，就可以认为该服务可见。

1.3.4 Broadcast Receiver 发送广播

在 Android 系统中，Broadcast Receiver 是一个广播接收器组件。广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的，例如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播，例如，通知其他应用程序一些数据下载完成并处于可用状态。应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的通知信息予以响应，所有的接收器均继承自 BroadcastReceiver 基类。

在 Android 系统中，广播接收器 Broadcast Receiver 没有用户界面。然而，它们可以启动一个 Activity 来响应它们收到的信息，或者用 NotificationManager 来通知用户。通知可以用很多种方式来吸引用户的注意力——闪动背灯、振动、播放声音等。一般来说是在状态栏上放一个持久的图标，用户可以打开它并获取消息。

Android 中的广播事件有两种，一种是系统广播事件，例如：ACTION_BOOT_COMPLETED（系统启动完成后触发）、ACTION_TIME_CHANGED（系统时间改变时触发）、ACTION_

BATTERY_LOW（电量低时触发）等。另一种是我们自定义的广播事件。

在 Android 系统中，广播事件的基本流程如下所示。

（1）注册广播事件：注册方式有两种，一种是静态注册，即在 AndroidManifest.xml 文件中定义，注册的广播接收器必须要继承 BroadcastReceiver；另一种是动态注册，是在程序中使用 Context.registerReceiver 注册，注册的广播接收器相当于一个匿名类。两种方式都需要 IntentFIlter。

（2）发送广播事件：通过 Context.sendBroadcast 来发送，由 Intent 来传递注册时用到的 Action。

（3）接收广播事件：当发送的广播被接收器监听到后，会调用它的 onReceive()方法，并将包含消息的 Intent 对象传给它。onReceive 中代码的执行时间不要超过 5s，否则 Android 会弹出超时对话框。

1.3.5　用 Content Provider 存储数据

在 Android 系统中，应用程序会把数据存放在一个 SQLite 数据库格式文件里，或者存放在其他有效设备里。如果想让其他程序能够使用我们程序中的数据，此时 Content Provider 就很有用了。Content Provider 是一个实现了一系列标准方法的类，这个类使得其他程序能存储、读取某种 Content Provider 可处理的数据。

1.4　进程和线程

Android 系统中也有进程和线程，代表当前系统中正在运行的程序。当第一次运行某个组件的时候，Android 会启动一个进程。在默认情况下，所有的组件和程序运行在这个进程和线程中，也可以安排组件在其他的进程或者线程中运行。在本节的内容中，简要讲解 Android 进程和线程的基本知识。

1.4.1　什么是进程

组件运行的进程由 manifest file 控制的。组件的节点一般都包含一个 process 属性，例如<activity>、<service>、<receiver>和<provider>节点。属性 process 可以设置组件运行的进程，可以配置组件在一个独立进程中运行，或者多个组件在同一个进程中运行，甚至可以多个程序在一个进程中运行，当然，前提是这些程序共享一个 User ID 并给定同样的权限。另外，<application>节点也包含了 process 属性，用来设置程序中所有组件的默认进程。

当更加常用的进程无法获取足够内存时，Android 会智能地关闭不常用的进程。当下次启动程序时会重新启动这些进程。当决定哪个进程需要被关闭时，Android 会考虑哪个对用户更加有用。例如 Android 会倾向于关闭一个长期不显示在界面的进程来支持一个经常显示在界面的进程。是否关闭一个进程决定于组件在进程中的状态。

1.4.2　什么是线程

当用户界面需要很快对用户进行响应，就需要将一些费时的操作，如网络连接、下载或者非常占用服务器时间的操作等放到其他线程。也就是说，即使为组件分配了不同的进程，有时候也需要再分配线程。

线程是通过 Java 的标准对象 Thread 来创建的，在 Android 中提供了如下管理线程的方法。

（1）Looper 在线程中运行一个消息循环。

（2）Handler 传递一个消息。

（3）HandlerThread 创建一个带有消息循环的线程。

（4）Android 让一个应用程序在单独的线程中，指导它创建自己的线程。

（5）应用程序组件（Activity、Service、Broadcast Receiver）所有都在理想的主线程中实例化。

（6）没有一个组件应该执行长时间或是阻塞操作（如网络呼叫或是计算循环）当被系统调用时，这将中断所有在该进程的其他组件。

（7）可以创建一个新的线程来执行长期操作。

第 2 章　获取并编译 Android 源代码

在分析 Android 源代码之前，需要先获取 Android 系统的源代码，并在自己的机器上进行编译。在本章的内容中，将详细讲解获取并编译 Android 5.0 源代码的基本知识。另外，因为 Android 系统源代码的文件数量巨大，目录结构层次复杂，所以，将在本章对 Android 5.0 源代码的目录结构进行整体分析，并详细介绍从 SDK 中生成 SDK 的方法。

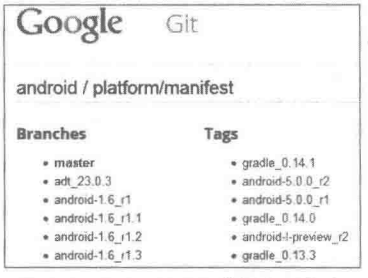
2.1 获取 Android 源代码

要想研究 Android 系统的源代码，需要先获取源代码。目前市面中的主流操作系统是 Windows、Linux 和 Mac OS。因为 Mac OS 属于类 Linux 系统，所以本书将讲解在 Windows 系统和 Linux 系统中获取 Android 源代码的知识。

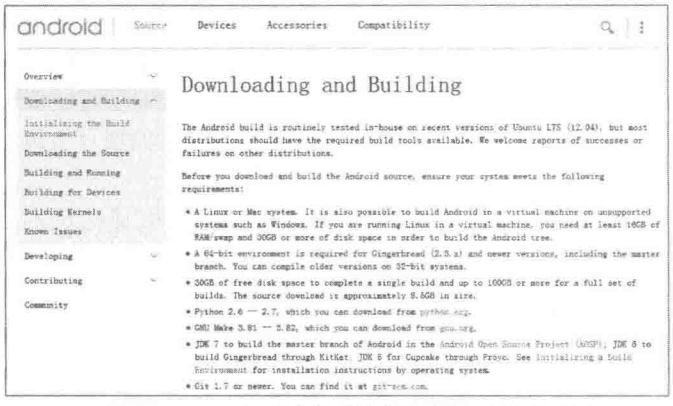
2.1.1　在 Linux 系统中获取 Android 源代码

北京时间 2014 年 11 月 5 日，Google 在 https://android. googlesource.com/上正式公布了 Android 5.0 的源代码，如图 2-1 所示。

在 Linux 系统中，通常使用 Ubuntu 来下载和编译 Android 源代码。由于 Android 的源代码内容很多，Google 采用了 Git 的版本控制工具，并对不同的模块设置不同的 Git 服务器，我们可以用 repo 自动化脚本来下载 Android 源代码。在 Android 官方站点 source.android.com/source/building.html 中，提供了获取并编译 Android 源代码的具体过程，如图 2-2 所示。



▲图 2-1　Android 5.0 的源代码分支



▲图 2-2　Android 官方提供的获取并编译源代码教程

在接下来的内容中，将详细讲解获取 Android 源代码的过程。

（1）下载 repo

在用户目录下，创建 bin 文件夹，用于存放 repo，并把该路径设置到环境变量中去，命令如下：

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

下载 repo 的脚本，用于执行 repo，命令如下：

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
```

设置可执行权限，命令如下：

```
$ chmod a+x ~/bin/repo
```

（2）初始化一个 repo 的客户端

在用户目录下，创建一个空目录，用于存放 Android 源代码，命令如下：

```
$ mkdir AndroidCode
$ cd AndroidCode
```

进入到 AndroidCode 目录，并运行 repo 下载源代码，下载主线分支的代码，主线分支包括最新修改的 bug，以及并未正式发布的版本的最新源代码，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

下载其他分支正式发布的版本，可以通过添加-b 参数来下载，命令如下：

```
$ repo init -u https://android.googlesource.com/platform/manifest -b
android-5.0_0_r1
```

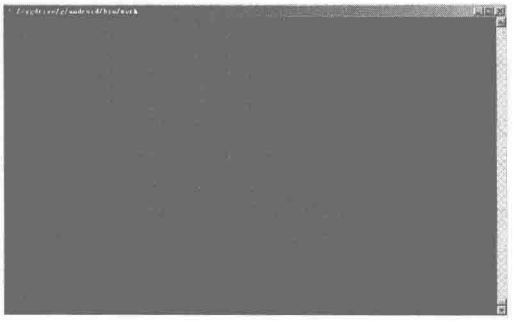
例如可以使用如下命令来初始化最新的 Android 源代码：

```
./repo init -u https://android.googlesource.com/platform/manifest -b android-5.0_0_r1
```

在下载过程中会需要填写 Name 和 E-mail，填写完毕之后，选择 Y 进行确认，最后提示 repo 初始化完成，这时可以开始同步 Android 源代码了，同步过程很漫长，需要耐心等待，执行下面命令开始同步代码：

```
$ repo sync
```

经过上述步骤后，便开始下载并同步 Android 源代码了，笔者的界面效果如图 2-3 所示。



▲图 2-3　正在下载源代码

因为网络方面的原因，可能执行“./repo init-u https://android.googlesource.com/platform/manifest-b android-5.0_0_r1”初始化命令会失败，提示一些类似网络连接失败的信息，此时不用理会，只需继续执行这个命令。如果出现多次失败提示，则可以尝试使用如下方法来解决。

（1）使用如下命令删除 Android 5.0 文件中的缓存文件，然后重新执行初始化命令：

```
rm -rf * -R
```

（2）晚上、凌晨时下载，一般这个时候的网络环境容易下载 Android 源代码。

注意

（1）在源代码下载过程中，在源代码下载目录看不到任何文件，打开“显示/隐藏”，会看到一个名为“.repo”的文件夹，这个文件夹是用来保存 Android 源代码的“临时文件”。

（2）当文件最后下载接近完成时，会从“.repo”文件夹中导出 Android 源代码。

（3）当 Android 5.0 源代码下载完成后，可以看到 Android 源代码下载目录中会有 bionic、bootable、build、cts、dalvik 等文件夹目录，这些就是 Android 的源代码。

（4）如果不得不关闭电脑停止下载，那么可以在源代码下载的终端中按下 Ctrl＋C 组合键或者 Ctrl＋Z 组合键停止源代码的下载，这样不会造成源代码的丢失或损坏。

2.1.2　在 Windows 平台获取 Android 源代码

在 Windows 平台上获取 Android 源代码的方式和在 Linux 中获取原理相同，但是需要预先在 Windows 平台上面搭建一个 Linux 环境，此处需要用到 cygwin 工具。cygwin 的作用是构建一套

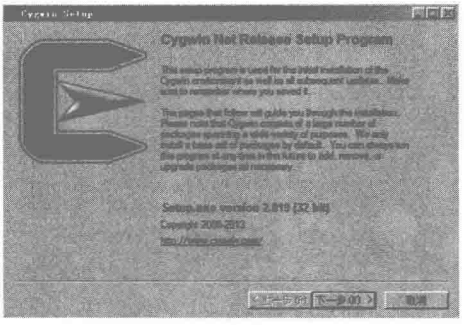
在 Windows 上的 Linux 模拟环境，下载 cygwin 工具的地址如下：

```
http://cygwin.com/install.html
```

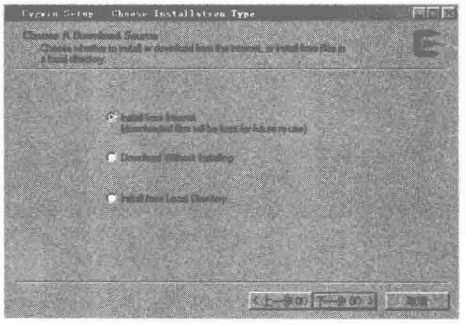
下载成功后会得到一个名为“setup.exe”可执行文件，通过此文件可以更新和下载最新的工具版本，具体流程如下所示。

（1）启动 cygwin，如图 2-4 所示。

（2）单击“下一步”按钮，选择第一个选项：从网络下载安装，如图 2-5 所示。



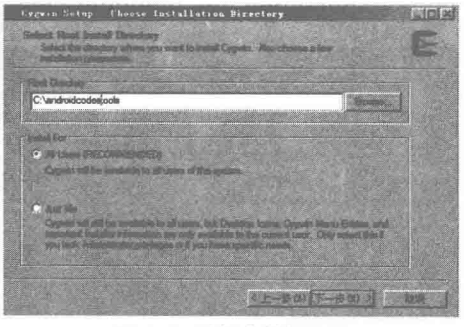
▲图 2-4 启动 cygwin



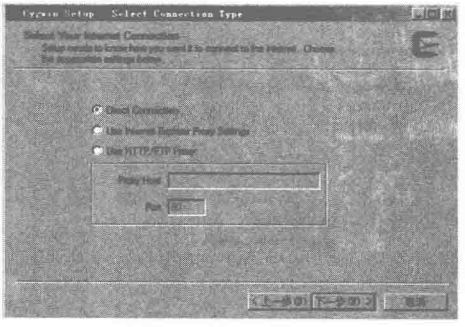
▲图 2-5 选择从网络下载安装

（3）单击“下一步”按钮，选择安装根目录，如图 2-6 所示。

（4）单击“下一步”按钮，设置网络代理。如果所在网络需要代理，则在这一步进行设置，如果不用代理，则选择直接下载，如图 2-7 所示。



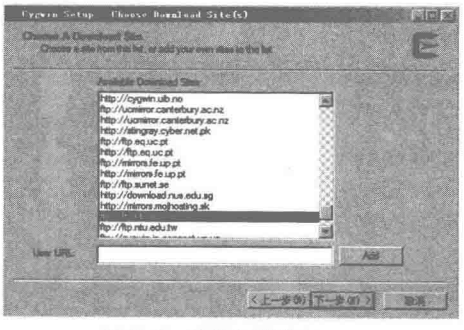
▲图 2-6 选择安装根目录



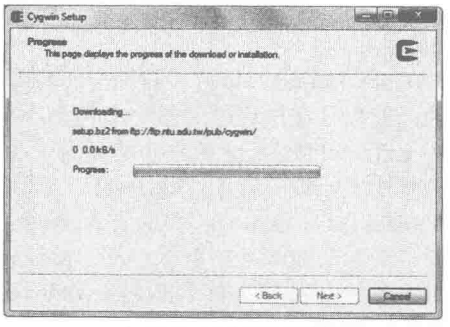
▲图 2-7 设置网络代理

（5）单击“下一步”按钮，选择下载站点。一般选择离我们比较近的站点，速度会比较快，如图 2-8 所示。

（6）单击“下一步”按钮，开始更新工具列表，如图 2-9 所示。



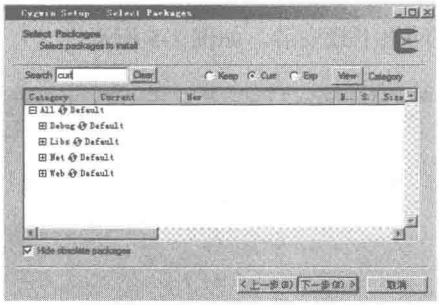
▲图 2-8 选择下载站点



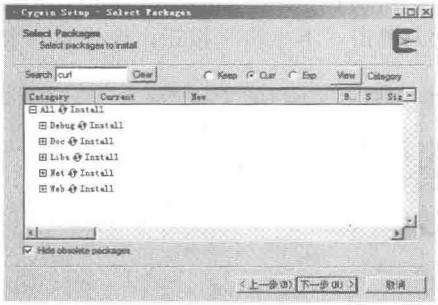
▲图 2-9 更新工具列表

（7）单击“下一步”按钮，选择需要下载的工具包。在此需要依次下载 curl、git、python 这些工具，如图 2-10 所示。

为了确保能够安装上述工具，一定要用鼠标双击图 2-10 中的那些项使其变为 Install 形式，如图 2-11 所示。



▲图 2-10　依次下载工具

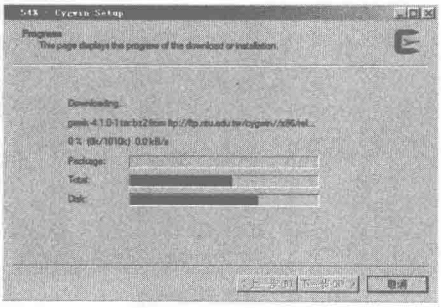


▲图 2-11　务必设置为 Install 形式

（8）单击“下一步”按钮，经过漫长的等待过程，如图 2-12 所示。

如果下载安装成功会出现提示信息，单击“完成”按钮即完成安装。当安装好 cygwin 后，打开 cygwin，会模拟出一个 Linux 的工作环境，然后按照 Linux 平台的源代码下载方法就可以下载 Android 源代码了。

建议读者在下载 Android 源代码时，严格按照官方提供的步骤进行，地址是：http://source.android.com/source/downloading.html，这一点对初学者来说尤为重要。另外，整个下载过程比较漫长，需要大家耐心等待。图 2-13 是下载 Android 源代码时的截图。



▲图 2-12　下载进度条

▲图 2-13　在 Windows 中用 cygwin 工具下载 Android 源代码

2.2　分析 Android 源代码结构

获得 Android 5.0 源代码后，源代码的全部工程分为以下 3 个部分。

- Core Project：核心工程部分，这是建立 Android 系统的基础，被保存在根目录的各个文件夹中。
- External Project：扩展工程部分，可以使其他开源项目具有扩展功能，被保存在“external”文件夹中。
- Package：包部分，提供了 Android 的应用程序、内容提供者、输入法和服务，被保存在“package”文件夹中。

在本节的内容中，将详细讲解 Android 5.0 源代码的目录结构。

▲图 2-14　下载的 Android 5.0 源代码

2.2.1 总体结构

无论是 Android 1.5 还是 Android 5.0，各个版本的源代码目录基本类似。在里面包含了原始 Android 的目标机代码、主机编译工具和仿真环境。解压缩下载的 Android 5.0 源代码包后，可以看到第一级目录有多个文件夹和一个 Makefile 文件，如图 2-14 所示。

如果是编译后的源代码目录，则会增加一个 out 文件夹，用来存放编译产生的文件。Android 5.0 第一级别目录结构的具体说明如表 2-1 所示。

表 2-1　　Android 5.0 源代码的根目录

Android 源代码根目录	描　述
abi	abi 相关代码，abi:application binary interface，应用程序二进制接口
art	全新的运行环境，需要和 Dalvik VM 区分开来
bionic	bionic C 库
bootable	启动引导相关代码
build	存放系统编译规则及 generic 等基础开发配置包
cts	Android 兼容性测试套件标准
dalvik	Dalvik 虚拟机
development	应用程序开发相关
device	设备相关代码
docs	介绍开源的相关文档
external	Android 使用的一些开源的模组
frameworks	核心框架——Java 及 C++语言，是 Android 应用程序的框架
gdk	即时通信模块
hardware	主要是硬件适配层 HAL 代码
kernel	Linux 的内核文件
libcore	核心库相关
libnativehelper	是 Support functions for Android's class libraries 的缩写，表示动态库，是实现实现的 JNI 库的基础
ndk	ndk 相关代码。Android NDK（Android Native Development Kit）是一系列的开发工具，允许程序开发人员在 Android 应用程序中嵌入 C/C++语言编写的非托管代码
out	编译完成后的代码输出在此目录
packages	应用程序包
pdk	Plug Development Kit 的缩写，是本地开发套件
prebuilts	x86 和 arm 架构下预编译的一些资源
sdk	sdk 及模拟器
system	文件系统和应用及组件，是用 C 语言实现的
tools	工具文件夹
vendor	厂商定制代码
Makefile	全局的 Makefile

由此可见，通过对源代码中根目录的每个文件夹的功能的介绍，可以看出源代码按功能分类还是非常清晰的，可以分为系统代码、工具、文档、开发环境、虚拟机、配置脚本和编译脚本等类别。并且也可以看出涉及的内容比较庞大和复杂，源代码分析工作需要多方面的理论和实践知识。

2.2.2　应用程序部分

应用程序主要是 UI 界面的实现，广大开发者基于 SDK 上开发的一个个独立的 APK 包，都是属于应用程序这一层的，应用程序在 Android 系统中处于最上层的位置。源代码结构中的 packages 目录用来实现系统的应用程序，packages 的目录结构如下所示。

```
packages /
├── apps //应用程序库
│   ├── BasicSmsReceiver          //基础短信接收
│   ├── Bluetooth                 //蓝牙
│   ├── Browser                   //浏览器
│   ├── Calculator                //计算器
│   ├── Calendar                  //日历
│   ├── Camera                    //照相机
│   ├── CellBroadcastReceiver     //单元广播接收
│   ├── CertInstaller             //被调用的包，在 Android 中安装数字签名
│   ├── Contacts                  //联系人
│   ├── DeskClock                 //桌面时钟
│   ├── Email                     //电子邮件
│   ├── Exchange                  //Exchange 服务
│   ├── Gallery                   //图库
│   ├── Gallery2                  //图库 2
│   ├── HTMLViewer                //HTML 查看器
│   ├── KeyChain                  //密码管理
│   ├── Launcher2                 //启动器 2
│   ├── Mms                       //彩信
│   ├── Music                     //音乐
│   ├── MusicFX                   //音频增强
│   ├── Nfc                       //近场通信
│   ├── PackageInstaller          //包安装器
│   ├── Phone                     //电话
│   ├── Protips                   //主屏幕提示
│   ├── Provision                 //引导设置
│   ├── QuickSearchBox            //快速搜索框
│   ├── Settings                  //设置
│   ├── SoundRecorder             //录音机
│   ├── SpareParts                //系统设置
│   ├── SpeechRecorder            //录音程序
│   ├── Stk                       //sim 卡相关
│   ├── Tag                       //标签
│   ├── VideoEditor               //视频编辑
│   └── VoiceDialer               //语音编号
├── experimental                  //非官方的应用程序
│   ├── BugReportSender           //Bug 的报告程序
│   ├── Bummer
│   ├── CameraPreviewTest         //照相机预览测试程序
│   ├── DreamTheater
│   ├── ExampleImsFramework
│   ├── LoaderApp
│   ├── NotificationLog
│   ├── NotificationShowcase
│   ├── procstatlog
│   ├── RpcPerformance
│   └── StrictModeTest
├── inputmethods                  //输入法
│   ├── LatinIME                  //拉丁文输入法
│   ├── OpenWnn                   //OpenWnn 输入法
│   └── PinyinIME                 //拼音输入法
├── providers                     //提供器
│   ├── ApplicationsProvider      //应用程序提供器，提供应用程序所需的界面
│   ├── CalendarProvider          //日历提供器
│   ├── ContactsProvider          //联系人提供器
│   ├── DownloadProvider          //下载管理提供器
│   ├── DrmProvider               //数据库相关
│   ├── GoogleContactsProvider    //Google 联系人提供器
│   ├── MediaProvider             //媒体提供器
│   ├── TelephonyProvider         //彩信提供器
│   └── UserDictionaryProvider    //用户字典提供器
├── screensavers                  //屏幕保护
│   ├── Basic                     //基本屏幕保护
```

```
    │   ├── PhotoTable          //照片方格
    │   ├── WebView             //网页
└── wallpapers                  //墙纸
    ├── Basic                   //系统内置墙纸
    ├── Galaxy4                 //S4 内置墙纸
    ├── HoloSpiral              //手枪皮套墙纸
    ├── LivePicker
    ├── MagicSmoke
    ├── MusicVisualization
    ├── NoiseField
    └── PhaseBeam
```

通过上面的目录结构可以看出，package 目录主要存放的是与 Android 系统应用层相关的内容，包括应用程序相关的包或者资源文件，其中包括系统自带的应用程序，及第三方开发的应用程序，还有屏幕保护和墙纸等应用，所以源代码中 package 目录对应着系统的应用层。

2.2.3 应用程序框架部分

应用程序框架是 Android 系统中的核心部分，也就是 SDK 部分，它会提供接口给应用程序使用，同时应用程序框架又会和系统服务、系统程序库、硬件抽象层有关联，所以其作用十分重大，应用程序框架的实现代码大部分都在/frameworks/base 和/framwworks/av 目录下，/frameworks/base 的目录结构如下所示。

```
frameworks/base
├── api                 //全是 XML 文件，定义了 API
├── cmds                //Android 中的重要命令（am、app_proce 等）
├── core                //核心库
├── data                //声音字体等数据文件
├── docs                //文档
├── drm                 //数字版权管理
├── graphics            //图形图像
├── icu4j               //用于解决国际化问题
├── include             //头文件
├── keystore            //数字签名证书相关
├── libs                //库
├── location            //地理位置
├── media               //多媒体
├── native              //本地库
├── nfc-extras          //NFC 相关
├── obex                //蓝牙传输
├── opengl              //opengl 相关
├── packages            //设置、TTS、VPN 程序
├── policy              //锁屏界面相关
├── sax                 //XML 解析器
├── services            //Android 的服务
├── telephony           //电话相关
├── test-runner         //测试相关
├── tests               //测试相关
├── tools               //工具
├── voip                //可视通话
└── wifi                //无线网络
```

以上这些文件夹包含了应用程序框架层的大部分代码，正是这些目录下的文件构成了 Android 的应用程序框架层，暴露出接口给应用程序调用，同时衔接系统程序库和硬件抽象层，形成一个由上至下的调用过程。在/frameworks/base 目录下也涉及系统服务，程序库中的一些代码，我们将在后面的两个小节中再详细分析。

2.2.4 系统服务部分

在 2.2.2 中介绍了应用程序框架层的内容，了解到大部分的实现代码保存在“/frameworks/base”目录下。其实在这个目录中还有一个名为“service”的目录，里面的代码是用于实现 Android 系统服务的。接下来将详细介绍 service 目录下的内容，其目录结构如下所示。

```
frameworks/base/services
├── common_time             //时间、日期相关的服务
```

```
├── input                    //输入系统服务
├── java                     //其他重要服务的 Java 层
├── jni                      //其他重要服务的 JNI 层
└── tests                    //测试相关
```

其中 java 和 jni 两个目录分别是一些其他的服务的 Java 层和 JNI 层实现，Java 目录下更详细的目录结构以及其他 Android 系统服务的说明如下所示。

```
frameworks/base/services/java/com/android/server
├── accessibility
├── am
├── connectivity
├── display
├── dreams
├── drm
├── input
├── location
├── net
├── pm
├── power
├── updates
├── usb
├── wm
├── AlarmManagerService.java              //闹钟服务
├── AppWidgetService.java                 //应用程序小工具服务
├── AppWidgetServiceImpl.java
├── AttributeCache.java
├── BackupManagerService.java             //备份服务
├── BatteryService.java                   //电池相关服务
├── BluetoothManagerService.java          //蓝牙
├── BootReceiver.java
├── BrickReceiver.java
├── CertBlacklister.java
├── ClipboardService.java
├── CommonTimeManagementService.java      //时间管理服务
├── ConnectivityService.java
├── CountryDetectorService.java
├── DevicePolicyManagerService.java
├── DeviceStorageMonitorService.java      //设备存储器监听服务
├── DiskStatsService.java                 //磁盘状态服务
├── DockObserver.java                     //底座监视服务
├── DropBoxManagerService.java
├── EntropyMixer.java
├── EventLogTags.logtags
├── INativeDaemonConnectorCallbacks.java
├── InputMethodManagerService.java        //输入法管理服务
├── IntentResolver.java
├── IntentResolverOld.java
├── LightsService.java
├── LocationManagerService.java           //地理位置服务
├── MasterClearReceiver.java
├── MountService.java                     //挂载服务
├── NativeDaemonConnector.java
├── NativeDaemonConnectorException.java
├── NativeDaemonEvent.java
├── NetworkManagementService.java         //网络管理服务
├── NetworkTimeUpdateService.java
├── NotificationManagerService.java       //通知服务
├── NsdService.java
├── PackageManagerBackupAgent.java
├── PreferredComponent.java
├── ProcessMap.java
├── RandomBlock.java
├── RecognitionManagerService.java
├── SamplingProfilerService.java
├── SerialService.java                    //NFC 相关
├── ServiceWatcher.java
├── ShutdownActivity.java
├── StatusBarManagerService.java          //状态栏管理服务
├── SystemBackupAgent.java
└── SystemServer.java
```

```
├── TelephonyRegistry.java
├── TextServicesManagerService.java
├── ThrottleService.java
├── TwilightCalculator.java
├── TwilightService.java
├── UiModeManagerService.java
├── UpdateLockService.java                    //锁屏更新服务
├── VibratorService.java                      //振动服务
├── WallpaperManagerService.java              //壁纸服务
├── Watchdog.java                             //看门狗
├── WifiService.java                          //无线网络服务
└── WiredAccessoryManager.java                //无线设备管理服务
```

从上面的文件夹和文件可以看出，Android 中涉及的服务种类非常多，包括界面、网络、电话等核心模块基本上都有其专属的服务，这些是属于系统级别的服务，这些系统服务一般都会在 Android 系统启动的时候加载，在系统关闭的时候结束，受到系统的管理，应用程序并没有权力去打开或者关闭，它们会随着系统的运行一直在后台运行，供应用程序和其他的组件使用。

另外，在 frameworks/av/下面也有一个 services 目录，这个目录下存放的是音频和照相机的服务的实现代码，目录结构如下所示。

```
frameworks/av/services
├── audioflinger          //音频管理服务
└── camera                //照相机的管理服务
```

这个 av/services 目录下的文件主要是用来支持 Android 系统中的音频和照相机服务的，这是两个非常重要的系统服务，开发应用程序时会经常依赖这两个服务。

2.2.5 系统程序库部分

Android 的系统程序库类型非常多，功能也非常强大，正是有了这些程序库，Android 系统才能运行多种多样的应用程序。在接下来的内容中，笔者挑了一些很常用也是很重要的系统程序库来分析它们在源代码中所处的位置。

（1）系统 C 库

Android 系统采用的是一个从 BSD 继承而来的标准的系统函数库 bionic，在源代码根目录下有这个文件夹，其目录结构如下所示。

```
bionic/
├── libc              //C 库
├── libdl             //动态链接库相关
├── libm              //数学库
├── libstdc++         //C++实现库
├── libthread_db      //线程库
├── linker            //连接器相关
└── test              //测试相关
```

（2）媒体库

Anroid 2.3 之后的媒体库由 Stragefright 实现，同时 Android 也自带了一些音视频的管理库，用于管理多媒体的录制、播放、编码和解码等功能。Android 多媒体程序库的实现代码主要在/frameworks/av/media 目录下，其目录结构如下所示。

```
frameworks/av/media/
├── common_time                 //时间相关
├── libeffects                  //多媒体效果
├── libmedia                    //多媒体录制，播放
├── libmedia_native             //里面只有一个 Android.mk，用来编译 native 文件
├── libmediaplayerservice       //多媒体播放服务的实现库
├── libstagefright              //stagefright 的实现库
├── mediaserver                 //跨进程多媒体服务
└── mtp                         //mtp 协议的实现（媒体传输协议）
```

（3）图层显示库

Android 中的图层显示库主要负责对显示子系统的管理，负责图层的渲染、叠加、绘制等功能，提供了 2D 和 3D 图层的无缝融合，是整个 Android 系统显示的“大脑中枢”，其代码在

/frameworks/native/services/surfaceflinger/目录下，其目录结构如下所示。

```
frameworks/native/services/surfaceflinger/
├── DisplayHardware           //显示底层相关
├── tests                     //测试
├── Android.mk                //MakeFile 文件
├── Barrier.h
├── Client.cpp                //显示的客户端实现文件
├── Client.h
├── clz.cpp
├── clz.h
├── DdmConnection.cpp
├── DdmConnection.h
├── DisplayDevice.cpp         //显示设备相关
├── DisplayDevice.h
├── EventThread.cpp           //消息线程
├── EventThread.h
├── GLExtensions.cpp          //opengl 扩展
├── GLExtensions.h
├── Layer.cpp                 //图层相关
├── Layer.h
├── LayerBase.cpp             //图层基类
├── LayerBase.h
├── LayerDim.cpp              //图层相关
├── LayerDim.h
├── LayerScreenshot.cpp       //图层相关
├── LayerScreenshot.h
├── MessageQueue.cpp          //消息队列
├── GLExtensions.h
├── MessageQueue.h
├── MODULE_LICENSE_APACHE2    //证书
├── SurfaceFlinger.cpp        //图层管理者，图层管理的核心类
├── SurfaceFlinger.h
├── SurfaceTextureLayer.cpp   //文字图层
├── SurfaceTextureLayer.h
├── Transform.cpp
└── Transform.h
```

（4）网络引擎库

网络引擎库主要是用来实现 Web 浏览器的引擎，支持 Android 的 Web 浏览器和一个可嵌入的 Web 视图，这个是采用第三方开发的浏览器引擎 Webkit 实现的，Webkit 的代码在/external/webkit/目录下，其目录结构如下所示。

```
external/webkit/
├── Examples                  //Webkit 例子
├── LayoutTests               //布局测试
├── PerformanceTests          //表现测试
├── Source                    //Webkit 源代码
├── Tools                     //工具
├── WebKitLibraries           //Webkit 用到的库
├── Android.mk                //Makefile
├── bison_check.mk
├── CleanSpec.mk
├── MODULE_LICENSE_LGPL       //证书
├── NOTICE
└── WEBKIT_MERGE_REVISION     //版本信息
```

（5）3D 图形库

Android 中的 3D 图形渲染是采用 Opengl 来实现的，Opengl 是开源的第三方图形渲染库，使用该库可以实现 Android 中的 3D 图形硬件加速或者 3D 图形软件加速功能，是一个非常重要的功能库。从 Android 5.0 开始，支持最新、最强大的 OpenGL ES 3.1。其实现代码在/frameworks/native/opengl 中，其目录结构如下所示。

```
frameworks/native/opengl/
├── include              //Opengl 中的头文件
├── libagl               //在 mac os 上的库
├── libs                 //Opengl 的接口和实现库
├── specs                //Opengl 的文档
```

```
├── tests                    //测试相关
└── tools                    //工具库
```

（6）SQLite

SQLite 是 Android 系统自带的一个轻量级关系数据库，其实现源代码已经在网上开源。SQLite 的优点是操作简单方便，运行速度较快，占用资源较少等特点，比较适合在嵌入式设备上使用。SQLite 是 Android 系统自带的实现数据库功能的核心库，其代码实现分为 Java 和 C 两个部分，Java 部分的代码在/frameworks/base/core/java/android/database，目录结构如下所示。

```
frameworks/base/core/java/android/database/
├── sqlite                                    //SQLite 的框架文件
├── AbstractCursor.java                       //游标的抽象类
├── AbstractWindowedCursor.java
├── BulkCursorDescriptor.java
├── BulkCursorNative.java
├── BulkCursorToCursorAdaptor.java            //游标适配器
├── CharArrayBuffer.java
├── ContentObservable.java
├── ContentObserver.java                      //内容观察者
├── CrossProcessCursor.java
├── CrossProcessCursorWrapper.java            //CrossProcessCursor 的封装类
├── Cursor.java                               //游标实现类
├── CursorIndexOutOfBoundsException.java      //游标出界异常
├── CursorJoiner.java
├── CursorToBulkCursorAdaptor.java            //适配器
├── CursorWindow.java                         //游标窗口
├── CursorWindowAllocationException.java      //游标窗口异常
├── CursorWrapper.java                        //游标封装类
├── DatabaseErrorHandler.java                 //数据库错误句柄
├── DatabaseUtils.java                        //数据库工具类
├── DataSetObservable.java
├── DataSetObserver.java
├── DefaultDatabaseErrorHandler.java          //默认数据库错误句柄
├── IBulkCursor.java
├── IContentObserver.aidl                     //aidl 用于跨进程通信
├── MatrixCursor.java
├── MergeCursor.java
├── Observable.java
├── package.html
├── SQLException.java                         //数据库异常
└── StaleDataException.java
```

Java 层的代码主要是实现 SQLite 的框架和接口的实现，方便用户开发应用程序时能很简单地操作数据库，并且捕获数据库异常。

C++层的代码在/external/sqlite 路径下，其目录结构如下所示。

```
external/sqlite/
├── android          //Android 数据库的一些工具包
└── dist             //Android 数据库底层实现
```

从上面 Java 和 C 部分的代码目录结构可以看出，SQLite 在 Android 中还是有很重要的地位的，并且在 SDK 中会有开放的接口让应用程序可以很简单方便地操作数据库，对数据进行存储和删除。

2.2.6 硬件抽象层部分

Android 的硬件抽象是各种功能的底层实现，理论上，不同的硬件平台会有不同的硬件抽象层实现，这一个层次也是与驱动层和硬件层有紧密的联系的，起着承上启下的作用，对上要实现应用程序框架层的接口，对下要实现一些硬件基本功能，以及调用驱动层的接口。需要注意的是，这一层也是广大 OEM 厂商改动最大的一层，因为这一层的代码与终端采用什么样的硬件平台有很大关系。源代码中存放的是硬件抽象层框架的实现代码和一些平台无关性的接口的实现。硬件抽象层代码在源代码根目录下的 hardware 文件夹中，其目录结构如下所示。

```
hardware/
├── libhardware              //新机制硬件库
├── libhardware_legacy       //旧机制硬件库
└── ril                      //ril 模块相关的底层实现
```

从上面的目录结构我们可以看出，硬件抽象层中主要是实现了一些底层的硬件库，用来实现应用层框架层中的功能，具体硬件库中有哪些内容，我们可以继续细分其目录结构，例如，libhardware 目录下的结构如下：

```
hardware/libhardware/
├── include                    //入口目录
├── modules                    //dex 反汇编
│   ├── audio                  //音频相关底层库
│   ├── audio_remote_submix    //音频混合相关
│   ├── gralloc                //帧缓冲
│   ├── hwcomposer             //音频相关
│   ├── local_time             //本地时间
│   ├── nfc                    //nfc 功能
│   ├── nfc-nci                //nfc 的接口
│   ├── power                  //电源
│   ├── usbaudio               //USB 音频设备
│   ├── Android.mk             //Makefile
│   └── README.android
├── tests                      //dex 生成相关
├── dexlist                    //dex 列表
├── dexopt                     //与验证和优化
└── docs                       //文档
```

从上面的目录结构我们可以分析出，libhardware 目录主要是 Android 系统的某些功能的底层实现，包括 audio、nfc 和 power。

而 libhardware_legacy 的目录与 libhardware 大同小异，只是针对旧的实现方式做的一套硬件库，其目录下还有 uevent、WiFi 以及虚拟机的底层实现。这两个目录下的代码一般会由设备厂家根据自身的硬件平台来实现符合 Android 机制的硬件库。

而 ril 目录下存放的是无线硬件设备与电话的实现，其目录结构如下所示。

```
hardware/ril/
├── include              //头文件
├── libril               //libril 库
├── mock-ril
├── reference-ril        //reference ril 库
├── rild                 //ril 守护进程
└── CleanSpec.mk
```

2.3 Android 源代码提供的接口

我们知道，Android 源代码当中提供了很多资源、工具或者文档供开发者使用，当然，其中也包括应用程序开发接口的实现，也就是我们开发应用程序所使用的 SDK 的 API。正是由于有了这些种类丰富、功能强大、抽象程度高的接口，才让我们开发应用程序变得简单方便。在本节的内容中，将详细讲解 Android 系统中这些接口的基本知识。

2.3.1　暴露接口和隐藏接口

我们可以将 Android 源代码编译生成一个 SDK，这个 SDK 的功能等同于官方网站上单独下载的 SDK 开发包。这说明在 Android 源代码中存在 SDK 的实现代码，不仅可以提供与独立 SDK 相同的 API 接口，而且会有一些 SDK 开发包中不具备的 API 接口。当然，这部分隐藏的接口在基于 SDK 开发的时候是看不到的，只有在基于源代码开发或者往独立的 SDK 中“增加”隐藏接口的时候才能调用到。

究竟源代码中的哪些接口是暴露接口，哪些接口是隐藏接口呢？比如要做一个统计电量消耗信息的应用程序，以及 WiFi 或者蓝牙的打开时间，在 SDK 中是没有直接相关的接口来调用的。当然通过其他途径可以找到很多种方法来满足这个需求，这里我们讲解怎么用源代码中的隐藏接口来实现这些功能。

在源代码路径/frameworks/base/core/java/android/os 目录下，存在两个电池相关的文件：

BatteryStats.java 和 BatteryStatsImpl.java，其中前者声明了一个与电池相关的抽象类 BatteryStats，后者继承了 BatteryStats，并实现了里面的方法。下面来看文件 BatteryStats.java 中的抽象类，在这个类中定义了很多变量来记录系统功能的状态变化，例如：

- WiFi 开关；
- 蓝牙开关；
- 音频打开；
- 视频打开；
- 上次充电时刻。

上述信息用来计算各个模块的电量消耗情况，同时在里面也定义了其他的抽象类，里面的抽象接口都可以用来计算电量消耗，文件 BatteryStats.java 的代码如下所示：

```
public abstract class BatteryStats implements Parcelable {
    /*省略部分代码*/
    //WIFI 开启时间
    public static final int WIFI_RUNNING = 4;
    //WIFI 完全锁定时间
    public static final int FULL_WIFI_LOCK = 5;
    //WIFI 扫描时间
    public static final int WIFI_SCAN = 6;
    //WIFI 其他功能开启时间
    public static final int WIFI_MULTICAST_ENABLED = 7;
    //音频开启时间
    public static final int AUDIO_TURNED_ON = 7;
     //视频开启时间
    public static final int VIDEO_TURNED_ON = 8;
    //系统状态自从上次变化到现在
    public static final int STATS_SINCE_CHARGED = 0;
    //上一次的系统状态
    public static final int STATS_LAST = 1;
    //现在的系统状态
    public static final int STATS_CURRENT = 2;
        //从上次拔下设备到现在的状态
    public static final int STATS_SINCE_UNPLUGGED = 3;
    /*省略部分代码*/
    public static abstract class Uid {
        //得到相关联 UID 锁屏状态
        public abstract Map<String, ? extends Wakelock> getWakelockStats();
        public static abstract class Wakelock {
            public abstract Timer getWakeTime(int type);
        }
        //得到相关联 UID 的传感器状态
        public abstract Map<Integer, ? extends Sensor> getSensorStats();
        //得到 Pid 状态
        public abstract SparseArray<? extends Pid> getPidStats();
        //得到进程状态
        public abstract Map<String, ? extends Proc> getProcessStats();
        //得到包状态
        public abstract Map<String, ? extends Pkg> getPackageStats();
        /**
* 得到 Uid
         * {@hide}
         */
        public abstract int getUid();
        /**
* 得到 Tcp 接收到的字节数
         * {@hide}
         */
        public abstract long getTcpBytesReceived(int which);
        /**
* 得到 Tcp 发出的字节数
         * {@hide}
         */
        public abstract long getTcpBytesSent(int which);
        //记录 WiFi 运行时刻
        public abstract void noteWifiRunningLocked();
        //记录 WiFi 停止时刻
```

```
        public abstract void noteWifiStoppedLocked();
        public abstract void noteFullWifiLockAcquiredLocked();
        public abstract void noteFullWifiLockReleasedLocked();
        public abstract void noteWifiScanStartedLocked();
        public abstract void noteWifiScanStoppedLocked();
        public abstract void noteWifiMulticastEnabledLocked();
        public abstract void noteWifiMulticastDisabledLocked();
        public abstract void noteAudioTurnedOnLocked();
        public abstract void noteAudioTurnedOffLocked();
        public abstract void noteVideoTurnedOnLocked();
        public abstract void noteVideoTurnedOffLocked();
        //得到 WiFi 运行时间
        public abstract long getWifiRunningTime(long batteryRealtime, int which);
        //得到 WiFi 锁定时间
        public abstract long getFullWifiLockTime(long batteryRealtime, int which);
        //得到 WiFi 扫描时间
        public abstract long getWifiScanTime(long batteryRealtime, int which);
        //得到 WiFi 其他功能开启时间
        public abstract long getWifiMulticastTime(long batteryRealtime,
                                                  int which);
        //得到音频开启时间
        public abstract long getAudioTurnedOnTime(long batteryRealtime, int which);
            //得到视频开启时间
        public abstract long getVideoTurnedOnTime(long batteryRealtime, int which);
        static final String[] USER_ACTIVITY_TYPES = {
            "other", "button", "touch"
        };
        public static final int NUM_USER_ACTIVITY_TYPES = 3;
        public abstract void noteUserActivityLocked(int type);
        public abstract boolean hasUserActivity();
        public abstract int getUserActivityCount(int type, int which);
        //传感器抽象类
        public static abstract class Sensor {
        public static final int GPS = -10000;
            public abstract int getHandle();
            public abstract Timer getSensorTime();
        }
        //Pid 类
        public class Pid {
            public long mWakeSum;
            public long mWakeStart;
        }
        //进程相关类
        public static abstract class Proc {
            public static class ExcessivePower {
                //唤醒方式
                public static final int TYPE_WAKE = 1;
                //CPU 类型
                public static final int TYPE_CPU = 2;
                public int type;
                public long overTime;
                public long usedTime;
            }
            //得到用户时间
            public abstract long getUserTime(int which);
            //得到系统时间
            public abstract long getSystemTime(int which);
                //得到状态
            public abstract int getStarts(int which);
            //得到 CPU 在前台运行的时间
            public abstract long getForegroundTime(int which);
            //得到 CPU 的速度等级
            public abstract long getTimeAtCpuSpeedStep(int speedStep, int which);
            //得到剩余的电量
            public abstract int countExcessivePowers();
            public abstract ExcessivePower getExcessivePower(int i);
        }
    }
/*省略部分代码*/
/** 得到屏幕点亮的时间
     * {@hide}
```

```
     */
    public abstract long getScreenOnTime(long batteryRealtime, int which);
/** 根据屏幕点亮的等级，得到相应时间
     * {@hide}
     */
    public abstract long getScreenBrightnessTime(int brightnessBin,
                                        long batteryRealtime, int which);
/**
     * 得到用电池的时候电话运行时的时间
     * {@hide}
     */
    public abstract long getPhoneOnTime(long batteryRealtime, int which);
/**
* 得到手机处于不同信号强度的时间
     * {@hide}
     */
    public abstract long getPhoneSignalStrengthTime(int strengthBin,
            long batteryRealtime, int which);
/**
     * 得到手机扫描信号用掉的时间
     * {@hide}
     */
    public abstract long getPhoneSignalScanningTime(
            long batteryRealtime, int which);
/**
     * 得到手机扫描到不同信号强度用掉的时间
     * {@hide}
     */
    public abstract int getPhoneSignalStrengthCount(int strengthBin, int which);
    /**
     * 得到手机不同数据连接消耗的时间
     * {@hide}
     */
    public abstract long getPhoneDataConnectionTime(int dataType,
            long batteryRealtime, int which);
/**
     * 得到手机进入到不同数据连接所消耗的时间
     * {@hide}
     */
    public abstract int getPhoneDataConnectionCount(int dataType, int which);
     /**
     * 得到手机处于 WiFi 打开的状态的时间
     * {@hide}
     */
    public abstract long getWifiOnTime(long batteryRealtime, int which);
    /**
     * 得到手机处于 WiFi 打开的状态并且驱动层的 WiFi 也处于打开状态时的时间
     * {@hide}
     */
    public abstract long getGlobalWifiRunningTime(long batteryRealtime, int which);
    /**
     * 得到手机蓝牙处于打开状态时的时间
     * {@hide}
     */
    public abstract long getBluetoothOnTime(long batteryRealtime, int which);
}
```

从上面的源代码可以看出，在文件 BatteryStats.java 中定义了很多与电池电量和系统状态相关的函数和变量，有一些函数在声明时加上了@hide 字样，说明这些接口因为不稳定或者其他方面的原因暂时被 Google 隐藏了，不能通过 SDK 进行访问，可能会在以后的版本中开放。

对于那些没有标记@hide 接口的函数，则不属于 Google 官方声明的隐藏接口，但是同样可以将其看成是隐藏的，只不过 Google 短期内或者一直都不会将其开放，所以不会特别进行维护，其形式与有@hide 的隐藏接口一致。

看完文件 BatteryStats.java 中定义的隐藏接口后，接下来看看文件 BatteryStatsImpl.java，在此文件中继承了 BatteryStats 抽象类，对 BatteryStats 中的很多隐藏方法进行了实现，其具体代码如下所示：

```
public final class BatteryStatsImpl extends BatteryStats {
/*省略了部分代码*/
        @Override
        public int getUid() {
            return mUid;
        }
        @Override
        public long getTcpBytesReceived(int which) {
            if (which == STATS_LAST) {
                return mLoadedTcpBytesReceived;
            } else {
                long current = computeCurrentTcpBytesReceived();
                if (which == STATS_SINCE_UNPLUGGED) {
                    current -= mTcpBytesReceivedAtLastUnplug;
                } else if (which == STATS_SINCE_CHARGED) {
                    current += mLoadedTcpBytesReceived;
                }
                return current;
            }
        }
        @Override
        public long getTcpBytesSent(int which) {
            if (which == STATS_LAST) {
                return mLoadedTcpBytesSent;
            } else {
                long current = computeCurrentTcpBytesSent();
                if (which == STATS_SINCE_UNPLUGGED) {
                    current -= mTcpBytesSentAtLastUnplug;
                } else if (which == STATS_SINCE_CHARGED) {
                    current += mLoadedTcpBytesSent;
                }
                return current;
            }
        }
    @Override public long getScreenOnTime(long batteryRealtime, int which) {
        return mScreenOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
    @Override public long getScreenBrightnessTime(int brightnessBin,
            long batteryRealtime, int which) {
        return mScreenBrightnessTimer[brightnessBin].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public long getPhoneOnTime(long batteryRealtime, int which) {
        return mPhoneOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
@Override public long getPhoneSignalStrengthTime(int strengthBin,
            long batteryRealtime, int which) {
        return mPhoneSignalStrengthsTimer[strengthBin].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public long getPhoneSignalScanningTime(
            long batteryRealtime, int which) {
        return mPhoneSignalScanningTimer.getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public int getPhoneSignalStrengthCount(int strengthBin, int which){
        return mPhoneSignalStrengthsTimer[strengthBin].getCountLocked(which);
    }
    @Override public long getPhoneDataConnectionTime(int dataType,
            long batteryRealtime, int which) {
        return mPhoneDataConnectionsTimer[dataType].getTotalTimeLocked(
                batteryRealtime, which);
    }
    @Override public int getPhoneDataConnectionCount(int dataType, int which) {
        return mPhoneDataConnectionsTimer[dataType].getCountLocked(which);
    }
    @Override public long getWifiOnTime(long batteryRealtime, int which) {
        return mWifiOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
    @Override public long getGlobalWifiRunningTime(long batteryRealtime, int
which) {
```

```
        return mGlobalWifiRunningTimer.getTotalTimeLocked(batteryRealtime,
which);
    }
    @Override public long getBluetoothOnTime(long batteryRealtime, int which) {
        return mBluetoothOnTimer.getTotalTimeLocked(batteryRealtime, which);
    }
}
```

在上述代码中，BatteryStatsImpl 继承了类 BatteryStats，然后实现了其中的隐藏接口，这样当进行应用程序开发时就可以使用这些还未开放的隐藏接口了，具体使用隐藏接口的方法将在 2.3.2 小节中详细介绍。

2.3.2 调用隐藏接口

在前面 2.3.1 的内容中，以 BatteryStats 这个类为例详细分析了 Android 中存在的隐藏接口。在接下来的内容中，将分析在源代码中使用这些隐藏接口的方法，然后分析在应用程序开发过程中调用隐藏接口的方法。

Android 系统中在 Settings 程序中使用类 BatteryStats，主要用来统计一些系统功能模块的工作时间和耗电情况。Settings 中使用 BatteryStats 类的是文件 PowerUsageSummary.java，其存放路径为：

```
/packages/apps/settings/src/com/android/settings/fuelgauge
```

文件 PowerUsageSummary.java 的主要功能是统计系统中的电量信息，在此用到了一些 BatteryStats 类中的隐藏接口，下面具体分析其实现代码，其中部分典型代码如下所示：

```
public class PowerUsageSummary extends PreferenceFragment implements Runnable {
    /*省略部分代码*/
    //定义了 BatteryStats 相关的 3 个对象，涉及到跨进程通信
    private static BatteryStatsImpl sStatsXfer;
    IBatteryStats mBatteryInfo;
    BatteryStatsImpl mStats;
    //在 onCreate 中初始化 mBatteryInfo 对象
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        if (icicle != null) {
            //对 mStats 对象赋值
            mStats = sStatsXfer;
        }
        addPreferencesFromResource(R.xml.power_usage_summary);
        //初始化 mBatteryInfo 对象
        mBatteryInfo = IBatteryStats.Stub.asInterface(
                ServiceManager.getService("batteryinfo"));
        mUm =(UserManager)getActivity().getSystemService(Context.USER_SERVICE);
        mAppListGroup = (PreferenceGroup) findPreference(KEY_APP_LIST);
        mBatteryStatusPref = mAppListGroup.findPreference(KEY_BATTERY_STATUS);
        mPowerProfile = new PowerProfile(getActivity());
        setHasOptionsMenu(true);
    }

        private void addPhoneUsage(long uSecNow) {
        long phoneOnTimeMs = mStats.getPhoneOnTime(uSecNow, mStatsType) / 1000;
        double phoneOnPower = mPowerProfile.getAveragePower(PowerProfile.POWER_RADIO_
ACTIVE)
                                                    * phoneOnTimeMs / 1000;
        addEntry(getActivity().getString(R.string.power_phone), DrainType.PHONE,
  phoneOnTimeMs, R.drawable.ic_settings_voice_calls, phoneOnPower);
    }
}
    //当 mStats 没有初始化时则通过 Parcel 接口继续初始化
    private void load() {
        try {
            byte[] data = mBatteryInfo.getStatistics();
            Parcel parcel = Parcel.obtain();
            parcel.unmarshall(data, 0, data.length);
            parcel.setDataPosition(0);
            mStats = com.android.internal.os.BatteryStatsImpl.CREATOR
```

```
                .createFromParcel(parcel);
            mStats.distributeWorkLocked(BatteryStats.STATS_SINCE_CHARGED);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException:", e);
        }
    }

    //得到屏幕的使用时间和耗电情况
private void addScreenUsage(long uSecNow) {
        double power = 0;
        // getScreenOnTime()为 BatteryStats 中的隐藏接口
        long screenOnTimeMs = mStats.getScreenOnTime(uSecNow, mStatsType) / 1000;
        power += screenOnTimeMs * mPowerProfile.getAveragePower(PowerProfile.POWER_
SCREEN_ON);
        final double screenFullPower =
                mPowerProfile.getAveragePower(PowerProfile.POWER_SCREEN_FULL);
        for (int i = 0; i < BatteryStats.NUM_SCREEN_BRIGHTNESS_BINS; i++) {
            double screenBinPower = screenFullPower * (i + 0.5f)
                    / BatteryStats.NUM_SCREEN_BRIGHTNESS_BINS;
            // getScreenBrightnessTime 为 BatteryStats 中的隐藏接口
            long brightnessTime = mStats.getScreenBrightnessTime(i, uSecNow,
            mStatsType) / 1000;
            power += screenBinPower * brightnessTime;
            if (DEBUG) {
                Log.i(TAG, "Screen bin power = " + (int) screenBinPower +
                ", time = " + brightnessTime);
            }
        }
        power /= 1000;
        addEntry(getActivity().getString(R.string.power_screen),
 DrainType.SCREEN, screenOnTimeMs, R.drawable.ic_settings_display, power);
    }
    //得到 Wifi 的使用时间和耗电情况
    private void addWiFiUsage(long uSecNow) {
        // getWifiOnTime 为 BatteryStats 中的隐藏接口
        long onTimeMs = mStats.getWifiOnTime(uSecNow, mStatsType) / 1000;
        // getGlobalWifiRunningTime 为 BatteryStats 中的隐藏接口
        long runningTimeMs = mStats.getGlobalWifiRunningTime(uSecNow, mStatsType) / 1000;
        if (DEBUG) Log.i(TAG, "WIFI runningTime=" + runningTimeMs
                + " app runningTime=" + mAppWifiRunning);
        runningTimeMs -= mAppWifiRunning;
        if (runningTimeMs < 0) runningTimeMs = 0;
        double wifiPower = (onTimeMs * 0 /* TODO */
                * mPowerProfile.getAveragePower(PowerProfile.POWER_WIFI_ON)
                                                        + runningTimeMs *
 mPowerProfile.getAveragePower(PowerProfile.POWER_WIFI_ON)) / 1000;
        if (DEBUG) Log.i(TAG, "WIFI power=" + wifiPower + " from procs=" +
 mWifiPower);
        BatterySipper bs=addEntry(getActivity().getString(R.string.power_wifi),
                                                        DrainType.WIFI,
                runningTimeMs, R.drawable.ic_settings_wifi, wifiPower + mWifiPower);
        aggregateSippers(bs, mWifiSippers, "WIFI");
    }
    //得到蓝牙的使用时间和耗电情况
    private void addBluetoothUsage(long uSecNow) {
        // getBluetoothOnTime 为 BatteryStats 中的隐藏接口
        long btOnTimeMs = mStats.getBluetoothOnTime(uSecNow, mStatsType) / 1000;
                                                double btPower = btOnTimeMs *
            mPowerProfile.getAveragePower(PowerProfile.POWER_BLUETOOTH_ON)
                                                                / 1000;
       // getBluetoothPingCount 为 BatteryStats 中的隐藏接口
        int btPingCount = mStats.getBluetoothPingCount();
        btPower += (btPingCount *
 mPowerProfile.getAveragePower(PowerProfile.POWER_BLUETOOTH_AT_CMD)) / 1000;
        BatterySipper bs = addEntry(getActivity().getString(R.string.power_bluetooth),
                                            DrainType.BLUETOOTH, btOnTimeMs,
R.drawable.ic_settings_bluetooth, btPower + mBluetoothPower);
        aggregateSippers(bs, mBluetoothSippers, "Bluetooth");
    }
```

从上述部分代码可以看出，在一些函数中广泛用到了 BatteryStats 类中的隐藏接口。首先调用隐藏接口来获取模块使用时间，然后再得到平均用电时间，经过一定的算法即可得出该模块的耗电情况。是如果没有这样的隐藏接口，计算耗电时间将是很麻烦的一件事，而且可能会用到源代码中更多我们没有权限去调用的代码，开发难度将会增加。

2.4 编译源代码

编译 Android 源代码的方法非常简单，只需使用 Android 源代码根目录下的 Makefile，执行 make 命令即可实现。当然在编译 Android 源代码之前，首先要确定已经完成同步工作。进入 Android 源代码目录使用 make 命令进行编译，使用此命令的格式如下所示：

```
$: cd ~/Android5.0（这里的 "Android5.0" 就是我们下载源代码的保存目录）
$: make
```

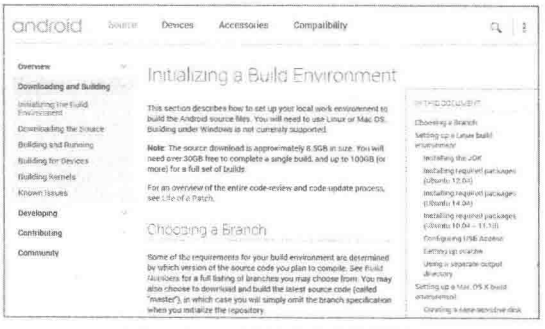
编译 Android 源代码可以得到“~/project/android/cupcake/out”目录，笔者的截图界面如图 2-15 所示。

整个编译过程也是非常漫长，需要读者耐心等待。在本节的内容中，将详细讲解编译并在模拟器中运行 Android 5.0 源代码的方法。

2.4.1 搭建编译环境

在编译 Android 源代码之前，需要先进行环境搭建工作。在 Android 官方网站 http://source.android.com/source/initializing.html 提供了编译源代码的教程，如图 2-16 所示。

▲图 2-15 编译过程的界面截图



▲图 2-16 Android 官方的编译教程

在接下来的内容中，以 Ubuntu 系统为例讲解搭建编译环境以及编译 Android 源代码的方法。具体流程如下所示。

（1）安装 JDK，编译 Android 5.0 的源代码需要 JDK 1.7，下载 jdk-7u22-linux-i586.bin 后进行安装，对应命令如下：

```
$ cd /usr
$ mkdir java
$ cd java
$ sudo cp jdk-7u22-linux-i586.bin 所在目录 ./
$ sudo chmod 755 jdk-7u22-linux-i586.bin
$ sudo sh jdk-7u22-linux-i586.bin
```

（2）设置 JDK 环境变量，将如下环境变量添加到主文件夹目录下的.bashrc 文件中，然后用 source 命令使其生效，加入的环境变量代码如下：

```
export JAVA_HOME=/usr/java/jdk1.7.0_27
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$PATH:$JAVA_HOME/bin:$JAVA_HOME/bin/tools.jar:$JRE_HOME/bin
export ANDROID_JAVA_HOME=$JAVA_HOME
```

对于安装好的 JDK，并且在添加环境变量之后，可以输入并执行命令“java –version”来查看 JDK 的版本。

（3）然后安装需要的编译工具，对于 Linux 10.04 系统来说，只需要安装如下所示的软件工具即可，在安装前保持电脑正常连接网络：

```
sudo apt-get install git-core gnupg flex bison gperf build-essential \
zip curl zlib1g-dev libc6-dev lib32ncurses5-dev ia32-libs \
x11proto-core-dev libx11-dev lib32readline5-dev lib32z-dev \
libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
```

然后使用下面的命令实现一个软链接文件：

```
sudo ln -s /usr/lib32/mesa/libGL.so.1 /usr/lib32/mesa/libGL.so
```

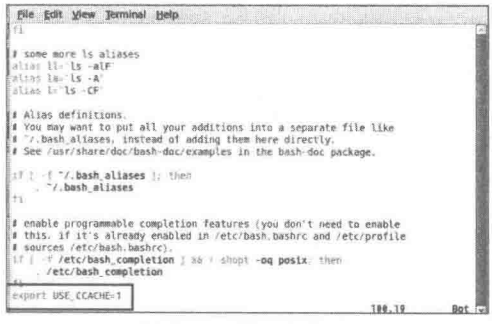
然后安装 Linux 11.10 系统需要的特别工具：

```
sudo apt-get install libx11-dev:i386
```

（4）开始设置高速缓存，目的是加快编译速度。对于配置不是很高的电脑来说，最好进行这个设置，这样可以节约很多时间。设置方法是先用 vi 或者 gedit 软件打开宿主目录下的“.bashrc”文件，然后在文件的最后添加如下值：

```
export USE_CCACHE=1
```

然后在保存后退出，重新登录系统以使设置生效，如图 2-17 所示。



▲图 2-17　设置高速缓存

在终端中切换到源代码根目录中，然后执行下面的命令设置 ccache 的大小为 50GB。

```
prebuilts/misc/linux-x86/ccache/ccache -M 50G
```

其实 ccache 就是一个执行文件，后面的-M 和 50GB 是传递给 ccache 的参数，表示设置 50GB 的缓存空间，这个大小可以根据我们的时间需要来修改。

（5）运行如下所示的命令，导入编译 Android 源代码所需的环境变量和其他参数：

```
source build/envsetup.sh
```

要想了解具体添加了哪些环境变量等，可以打开图 2-18 中方框中对应的文件。

（6）运行 lunch 命令选择编译目标，运行 lunch 命令后会出现一些已经预置好的项目。在此输入对应的数字，然后回车选择编译目标对象，如图 2-19 所示。

```
including device/samsung/manta/vendorsetup.sh
including device/generic/mips/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/lge/hammerhead/vendorsetup.sh
including device/lge/mako/vendorsetup.sh
including device/asus/grouper/vendorsetup.sh
including device/asus/deb/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
including device/asus/flo/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

▲图 2-18　打开方框中对应的文件

```
Lunch menu... pick a combo:
     1. aosp_arm-eng
     2. aosp_x86-eng
     3. aosp_mips-eng
     4. vbox_x86-eng
     5. aosp_manta-userdebug
     6. mini_mips-userdebug
     7. mini_armv7a_neon-userdebug
     8. mini_x86-userdebug
     9. aosp_hammerhead-userdebug
     10. aosp_mako-userdebug
     11. aosp_grouper-userdebug
     12. aosp_deb-userdebug
     13. aosp_tilapia-userdebug
     14. aosp_flo-userdebug

Which would you like? [aosp_arm-eng] 1
```

▲图 2-19　选择编译目标

运行 lunch 命令并选择好编译目标后，会出现如图 2-20 所示的提示信息。

（7）接下来开始步入真正的编译阶段了，运行如下所示的命令进行编译：

```
make -j16
```

因为笔者电脑的 CPU 是 i7 4770，所以，在此使用 16，如图 2-21 所示。

```
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-2.6.32-52-generic-x86_64-with-Ubuntu-10.04-lucid
HOST_BUILD_TYPE=release
BUILD_ID=KRT16M
OUT_DIR=out
```

▲图 2-20 出现的提示信息

```
TARGET_PRODUCT=aosp_arm
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
TARGET_CPU_VARIANT=generic
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-2.6.32-52-generic-x86_64-with-Ubuntu-10.04-lucid
HOST_BUILD_TYPE=release
BUILD_ID=KRT16M
OUT_DIR=out
============================================
```

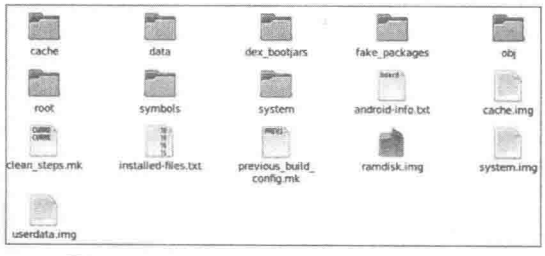
▲图 2-21 开始编译

编译过程比较慢，因为电脑配置的问题可能需要几个小时的漫长等待。编译成功后会弹出如图 2-22 所示的提示信息。

这样在编译完成后，可以在源代码中的“out/target/product/generic/”目录生成对应固件等文件，如图 2-23 所示。

```
File Edit View Terminal Help
+ MAKE_EXT4FS_CMD='make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a
system out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/p
roduct/generic/system'
+ echo make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/ta
rget/product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generi
c/system
make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/pr
oduct/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/syste
m
+ make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/
product/generic/obj/PACKAGING/systemimage_intermediates/system.img out/target/product/generic/sys
tem
Creating filesystem with parameters:
    Size: 576716800
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7040
    Inode size: 256
    Journal blocks: 2200
    Label:
    Blocks: 140800
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1262/35200 inodes and 8165/140800 blocks
+ '[' 0 -ne 0 ']'
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800 reserve=5
8472
```

▲图 2-22 编译成功时的提示信息



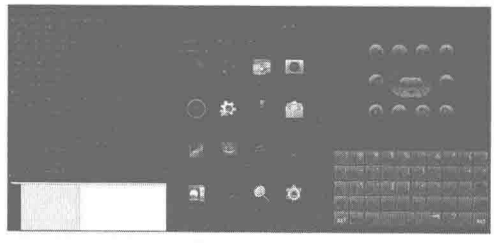
▲图 2-23 “out/target/product/generic/”目录

2.4.2 在模拟器中运行

在模拟器中运行的步骤就比较简单了，只需在终端中执行下面的命令即可：

```
emulator
```

运行成功后的效果如图 2-24 所示。



▲图 2-24 在模拟器中的编译执行效果

2.5 编译源代码生成 SDK

在 Android 5.0 的源代码的根目录下有一个 SDK 目录，所有和 SDK 相关的代码都放在这个目录中，包括镜像文件、模拟器和 ADB 等常用工具，以及 SDK 中的开发包的文档，可以通过编译的方式来生成开发需要的 SDK，编译命令如下所示：

```
$ Make SDK
```

当编译完成后，会在/out/host/linux-x86/sdk/目录下生成 SDK，这个 SDK 是完全与源代码同步的，与官方网站上下载的 SDK 功能完全相同，包含了开发用的 JAR 包、模拟器管理工具和 ADB 调试工具，可以使用这个编译生成的 SDK 来开发应用程序。

对于 Android 系统开发来说，可以分为以下两种开发方式。

- 基于 SDK 的开发。
- 基于源代码的开发。

在一般情况下，都是基于 SDK 开发应用程序的，比较方便且兼容性好。基于源代码的开发相对于基于 SDK 的开发要求对源代码的架构认识更深刻，一般用于需要修改系统层面的场合。两种方式应用场景不同，各有优点和缺点，在本节将主要介绍基于 SDK 的开发。

如果想基于 SDK 开发 Android 的应用程序，我们需要 JDK、SDK 和一个开发环境，JDK 和 SDK 在不同的平台下有不同的版本，本章主要讨论 Windows 7 平台下的开发环境搭建。

（1）安装 JDK

由于 Android 的应用程序使用 Java 语言开发的，所以首先需要安装 Java 的 JDK，下载链接：http://java.sun.com/javase/downloads/index.jsp，进入后选择合适的平台以及下载最新版本的 JDK，安装成功后，命令行下可以查看 JDK 版本。

（2）安装 Eclipse

Eclipse 是开发 Android 应用程序的 IDE 环境，有非常丰富的插件可以使用，单击 http://www.eclipse.org/downloads/可以下载合适平台的最新版本 Eclipse。

（3）安装 Android SDK

Android SDK 是 Google 对外发布的专门用于 Android 开发的工具包，里面有各种版本的开发框架和工具，以及丰富的文档，打开 http://developer.android.com/sdk/index.html 可以下载最新版本的针对 Window 7 平台的 SDK。

▲图 2-25　"Install New SoftWare" 界面

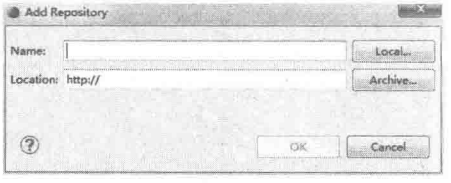
当下载完成上述 3 个工具之后，需要对开发环境进行如下所示的配置。

（1）配置 Eclipse

第 1 步：打开 Eclipse，在菜单栏上选择 help→Install New SoftWare，出现图 2-25 所示的界面。

第 2 步：单击“Add”按钮，会出现如图 2-26 所示的界面。

第 3 步：在 Name 栏里面输入 Android 或者自定义任何名字，在 Location 里面输入 https://dl-ssl.google.com/android/eclipse/，输入后的效果如图 2-27 所示。



▲图 2-26　"Add Repository" 界面

▲图 2-27　"Add Site" 界面

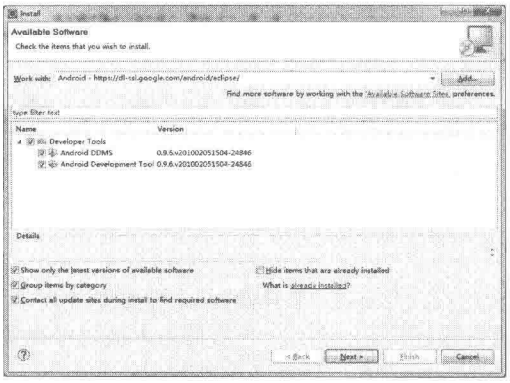
第 4 步：如果发现 https://无法使用，可以改成 http://尝试下，当输入好名字和地址之后，单击“OK”按钮，会出现如图 2-28 所示的界面。

图 2-28 中的两个插件都是开发 Android 必不可少的工具包，Android DDMS 是用来调试和管理 Android 进程、存储器、查看日志的工具，Android Development Tool 简称 ADT，是开发 Android 的插件，只有装了 ADT 才能创建 Android 工程。

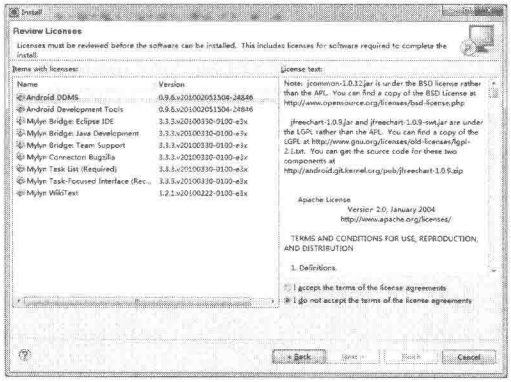
第 5 步：单击“Next”按钮，出现如图 2-29 所示的界面。

在图 2-29 中列出了将会安装的工具包，勾选“I accept...”选项，单击“Next”按钮会开始安装插件，界面如图 2-30 所示。

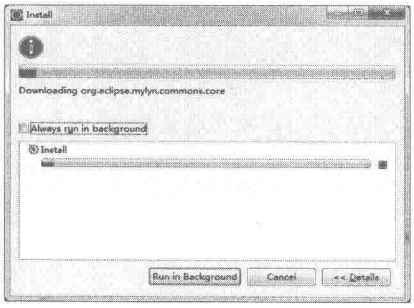
第 6 步：当所有插件安装成功后，会弹出如图 2-31 所示的提示界面。



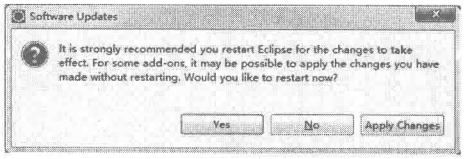
▲图 2-28 “Instal”界面



▲图 2-29 选择安装



▲图 2-30 开始安装



▲图 2-31 成功

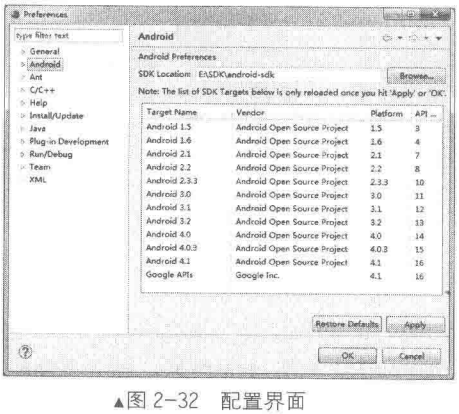
这时我们需要单击“Yes”按钮重启 Eclipse 让所有插件生效。

（2）配置 Android SDK

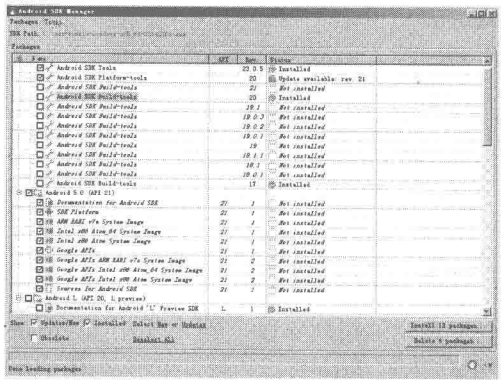
打开 Eclipse，单击 Window→preferences，来到如图 2-32 所示的界面。

这样我们就可以从 Eclipse 中新创建 Android 工程，要想新创建工程是基于什么版本的 Android 系统，可以打开 SDK 与目录下的 SDK 管理工具 SDK Manager.exe，双击后会进入到 SDK 工具包管理界面，如图 2-33 所示。

在图 2-33 中可以看到，很清晰地列出了当前版本 SDK 中包含的工具包，以及已经安装了的和没有安装的版本。可以继续单击“Install 11 Packages”或者“Delete 8 Packages”按钮安装和删除 SDK 中的工具包。如果是安装，则过程会比较慢，与网速的关系比较大。当我们将 SDK 中的工具包安装完毕，同时也完成了 Eclipse 和 SDK 的配置工作，至此 Windows 7 平台下基于 SDK 的 Android 的开发环境搭建全部完成。



▲图 2-32 配置界面



▲图 2-33 Android SDK 管理

第3章　分析 Java Native Interface 系统

Java Native Interface 缩写为 JNI，表示 Java 本地接口。从 Java 1.1 开始，JNI 标准便成为了 Java 平台的一部分，它允许 Java 代码和其他语言写的代码进行交互。JNI 一开始是为本地已编译语言，尤其是 C 和 C++而设计的，但是，它并不妨碍你使用其他语言，只要调用约定受支持就可以了。在本章的内容中，将详细分析 Android 5.0 中 JNI 的基本知识。

3.1 JNI 基础

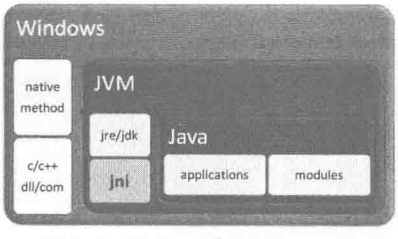
在 Android 系统中，JNI 是连接 Java 部分和 C/C++部分的桥梁。要想完整地使用 JNI，需要仔细分析 Java 代码和 C/C++代码。在 Android 中通过提供 JNI 的方式，让 Java 程序可以调用 C 语言程序。Android 中的很多 Java 类都具有 Native（本地）接口，这些接口由本地实现，然后注册到系统中。在本节的内容中，将详细讲解 JNI 的基础知识。

3.1.1　JNI 的功能结构

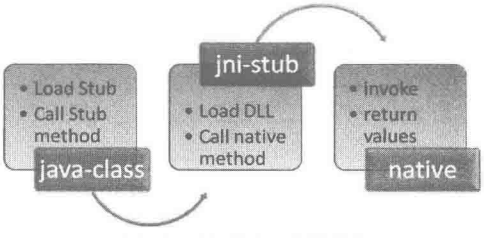
JNI 最初是由 Sun 提供的 Java 与系统中的原生方法交互的技术，用于在 Windows\Linux 系统中实现 Java 与 Native Method（本地方法）的相互调用。JVM（Java 虚拟机）在封装各种操作系统实际的差异性的同时提供了 JNI 技术，使得开发者可以通过 Java 程序（代码）调用到操作系统相关的技术实现的库函数，从而与其他技术和系统交互，使用其他技术实现的系统的功能。同时，其他技术和系统也可以通过 JNI 提供的相应原生接口调用 Java 应用系统内部实现的功能。

在 Windows 系统上，一般可执行的应用程序都是基于 Native（本地）的 PE 结构，Windows 上的 JVM 也是基于 Native 结构实现的，Java 应用体系都是构建于 JVM 之上。由此可见，Windows 系统上的 Java 体系如图 3-1 所示。

JNI 对于应用本身来说，可以将其看做是一个代理模式。对于开发者来说，需要使用 C/C++来实现一个代理程序（JNIi 程序）来实际操作目标原生函数，Java 程序中则是 JVM 通过加载并调用此 JNI 程序来间接地调用目标原生函数。JNI 的调用过程如图 3-2 所示。



▲图 3-1　Windows 系统上的 Java 体系



▲图 3-2　JNI 的调用过程图

3.1.2　JNI 的调用层次

JNI 调用的层次主要分为 3 层，在 Android 系统中这 3 层从上到下依次为：Java→JNI→C/C++(SO 库)，Java 可以访问 C/C++中的方法，同样 C/C++可以修改 Java 对象，图 3-3 描述了这 3 者之间的调用关系。

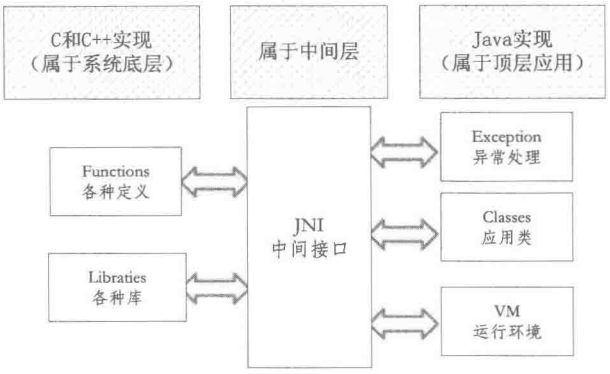
由图 3-3 可知，JNI 的调用关系为：

```
Java------------------------JNI------------------------Native
```

在 Android 5.0 的源代码中，主要的 JNI 代码放在以下的路径中：

```
frameworks/base/core/jni/
```

上述路径中的内容被编译成库 libandroid_runtime.so，这是一个普通的动态库，被放置在目标系统的“/system/lib”目录下。另外，Android 中还存在了其他的 JNI 库，其实 JNI 中的各个文件就是普通的 C++ 源文件；在 Android 中实现的 JNI 库，需要连接动态库 libnativehelper.so。



▲图 3-3 JNI 调用的层次关系

3.1.3 分析 JNI 的本质

要想弄明白 JNI 的本质，还要从 Java 的本质说起。从本质上来说，Java 语言的运行完全依赖于脚本引擎对 Java 的代码进行解释和执行。因为现代的 Java 可以从源代码编译成.class 之类的中间格式的二进制文件，所以这种处理会加快 Java 脚本的运行速度。尽管如此，基本的执行方式仍然不变，由脚本引擎（被称之为 JVM）来执行。与 Python、Perl 之类的纯脚本相比，只是把脚本变成了二进制格式而已。另外，Java 本身就是一门面向对象语言，可以调用完善的功能库。当把这个脚本引擎移植到所有平台上之后，那么这个脚本就很自然地实现“跨平台”了。绝大多数的脚本引擎都支持一个很显著的特性，就是可以通过 C/C++编写模块，并在脚本中调用这些模块。同样 Java 也是如此，Java 一定要提供一种在脚本中调用 C/C++编写的模块的机制，才能称得上是一个完善的脚本引擎。

从本质上来看，Android 平台是由 arm-linux 操作系统和一个叫做 Dalvik 虚拟机组成的。所有在 Android 模拟器上面看到的界面效果都是用 Java 语言编写的，具体请看源代码中的“frameworks/base”目录。Dalvik 虚拟机只是提供了一个标准的支持 JNI 调用的 Java 虚拟机环境。

在 Android 平台中，使用 JNI 技术封装了所有的和硬件相关的操作，通过 Java 去调用 JNI 模块，而 JNI 模块使用 C/C++调用 Android 本身的 arm-linux 底层驱动，这样便实现了对硬件的调用。

在 Android 5.0 的源代码中，和 JNI 相关的文件如下所示：

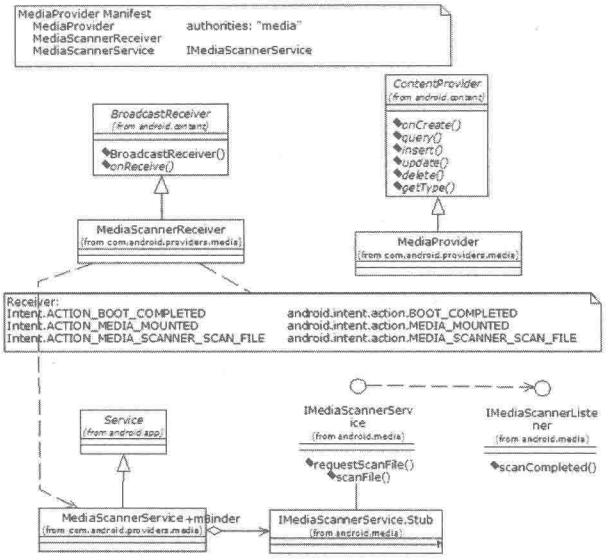
```
./frameworks/base/media/java/android/media/MediaScanner.java
./frameworks/base/media/jni/android_media_MediaScanner.cpp
./frameworks/base/media/jni/android_media_MediaPlayer.cpp
./frameworks/base/media/jni/AndroidRuntime.cpp
./libnativehelper/JNIHelp.cpp
```

由此可见，和 JNI 密切相关的是 Media 系统，而 Media 系统的架构基础是 MediaScanner。在启动 Android 系统之初，就会扫描出系统中的 Media 文件供后续应用使用，既有新加入的媒体，也有几微秒钟前删除的媒体文件，并且还需要自动更新相应的媒体库。在 Android 系统中，和用户体验模块相关的 Music、Gallery 播放等应用，也是基于 MediaScanner 的扫描媒体文件功能的。MediaScanner 位于 Android 5.0 源代码的如下路径中：

```
packages/providers/MediaProvider
```

一共包含了 3 个主要部分：MediaScannerReceiver、MediaScannerService 和 MediaProvider。在“MediaProvider”目录下的 AndroidManifest 中可以查看 MediaProvider 的基本架构，如图 3-4 所示。

- MediaScannerReceiver：是一个 BroadcastReceiver（接收广播），功能是进行媒体扫描，这也是 MediaScanner 提供给外界的接口之一。收到广播之后启动 MediaScannerService 具体执行扫描工作。



▲图 3-4　MediaProvider 的基本架构

- MediaScannerService：是一个 Service，负责媒体扫描，它还要用到 Framework 中的 MediaScanner 来共同完成具体扫描工作，扫描的结果在 MediaProvider 提供的数据库中。
- MediaProvider：是一个 ContentProvider，媒体库（Images/Audio/Video/Playlist 等）的数据提供者。负责操作数据库，并提供给别的程序 insert、query、delete、update 等操作。

3.2　分析 MediaScanner

在 Android 5.0 中，下面是 MediaScanner 系统的 JNI 的调用关系：

```
MediaScanner -------------libmedia_jni.so -------------libmedia.so
```

在 Android 系统中，MediaScanner 的功能是扫描媒体文件，得到诸如歌曲时长、歌曲作者等信息，并将这些信息存放到媒体数据库中，以供其他应用程序使用。在本节的内容中，以 MediaScanner 源代码分析作为基础，将详细分析 JNI 在 Android 系统中的作用。

3.2.1　分析 Java 层

在 MediaScanner 系统中，Java 层的实现文件为：

```
frameworks/base/media/java/android/media/MediaScanner.java。
```

在接下来的内容中，将详细讲解 MediaScanner 系统中 Java 层的具体实现源码。

（1）加载 JNI 库

在文件 MediaScanner.java 中，首先定义类 MediaScanner 并加载 JNI 库，然后定义 JNI 的 Native（本地）函数。主要代码如下所示：

```
public class MediaScanner
{
    static {
        System.loadLibrary("media_jni");
        native_init();
    }

    private final static String TAG = "MediaScanner";

    private static final String[] FILES_PRESCAN_PROJECTION = new String[] {
            Files.FileColumns._ID, // 0
```

```
            Files.FileColumns.DATA, // 1
            Files.FileColumns.FORMAT, // 2
            Files.FileColumns.DATE_MODIFIED, // 3
    };

    private static final String[] ID_PROJECTION = new String[] {
            Files.FileColumns._ID,
    };

    private static final int FILES_PRESCAN_ID_COLUMN_INDEX = 0;
    private static final int FILES_PRESCAN_PATH_COLUMN_INDEX = 1;
    private static final int FILES_PRESCAN_FORMAT_COLUMN_INDEX = 2;
    private static final int FILES_PRESCAN_DATE_MODIFIED_COLUMN_INDEX = 3;

    private static final String[] PLAYLIST_MEMBERS_PROJECTION = new String[] {
            Audio.Playlists.Members.PLAYLIST_ID, // 0
     };

    private static final int ID_PLAYLISTS_COLUMN_INDEX = 0;
    private static final int PATH_PLAYLISTS_COLUMN_INDEX = 1;
    private static final int DATE_MODIFIED_PLAYLISTS_COLUMN_INDEX = 2;

    private static final String RINGTONES_DIR = "/ringtones/";
    private static final String NOTIFICATIONS_DIR = "/notifications/";
    private static final String ALARMS_DIR = "/alarms/";
    private static final String MUSIC_DIR = "/music/";
    private static final String PODCAST_DIR = "/podcasts/";
……
    private static native final void native_init();//声明一个 native 函数，native 为关键字
    private native final void native_setup();
    ……
}
```

函数 native_init 位于包 android.media 中，其完整路径名为：

```
android.media.MediaScanner.nantive_init
```

根据规则其对应的 JNI 层函数名称为：

```
android_media_MediaScanner_native_init
```

在调用函数native之前需要先加载JNI库，一般在类的static中加载调用函数System.loadLibrary()。在加载了相应的 JNI 库之后，如果要使用相应的 native 函数，只需使用 native 声明需要被调用的函数即可：

```
private native void processDirectory(String path, String extensions, MediaScannerClient
client);
private native void processFile(String path, String mimeType, MediaScannerClient client);
public native void setLocale(String locale);
```

（2）实现扫描工作

在文件 MediaScanner.java 中，通过函数 scanDirectories 实现扫描工作，具体代码如下所示：

```
  public void scanDirectories(String[] directories, String volumeName) {
      try {
          long start = System.currentTimeMillis();
          initialize(volumeName);//初始化
          prescan(null, true);//扫描前的预处理
          long prescan = System.currentTimeMillis();

          if (ENABLE_BULK_INSERTS) {
              // create MediaInserter for bulk inserts
              mMediaInserter = new MediaInserter(mMediaProvider, 500);
          }
//函数 processDirectory 是一个 Native 函数，功能是对目标文件夹进行扫描
          for (int i = 0; i < directories.length; i++) {
              processDirectory(directories[i], mClient);
          }

          if (ENABLE_BULK_INSERTS) {
              // flush remaining inserts
```

```
                mMediaInserter.flushAll();
                mMediaInserter = null;
            }

            long scan = System.currentTimeMillis();
            postscan(directories);//扫描后的处理
            long end = System.currentTimeMillis();

            if (false) {
                Log.d(TAG, " prescan time: " + (prescan - start) + "ms\n");
                Log.d(TAG, "    scan time: " + (scan - prescan) + "ms\n");
                Log.d(TAG, "postscan time: " + (end - scan) + "ms\n");
                Log.d(TAG, "   total time: " + (end - start) + "ms\n");
            }
        } catch (SQLException e) {
            Log.e(TAG, "SQLException in MediaScanner.scan()", e);
        } catch (UnsupportedOperationException e) {
            Log.e(TAG, "UnsupportedOperationException in MediaScanner.scan()", e);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in MediaScanner.scan()", e);
        }
    }
```

在上述代码中使用函数 initialize 实现初始化操作，此函数的具体实现代码如下所示：

```
private void initialize(String volumeName) {
    //打开 MediaProvider，获得它的一个实例
    mMediaProvider = mContext.getContentResolver().acquireProvider("media");
    //得到一些 uri
    mAudioUri = Audio.Media.getContentUri(volumeName);
    mVideoUri = Video.Media.getContentUri(volumeName);
    mImagesUri = Images.Media.getContentUri(volumeName);
    mThumbsUri = Images.Thumbnails.getContentUri(volumeName);
    mFilesUri = Files.getContentUri(volumeName);
    //如果需要外部存储的话，则可以支持播放列表，用缓存池实现，如 mGenreCache 等
    if (!volumeName.equals("internal")) {
        // we only support playlists on external media
        mProcessPlaylists = true;
        mProcessGenres = true;
        mPlaylistsUri = Playlists.getContentUri(volumeName);
        mCaseInsensitivePaths = true;
    }
}
```

（3）读取并保存信息

在文件 MediaScanner.java 中，通过函数 prescan 读取之前扫描的数据库中和文件相关的信息并保存起来。函数 prescan 创建了一个用于缓存扫描文件信息的对象 FileCache，如 last_modified 等。这个 FileCache 是从 MediaProvider 中已有信息构建出来的历史信息，并且根据扫描得到的新信息来对应更新历史信息。函数 prescan 的具体实现代码如下所示：

```
    private void prescan(String filePath, boolean prescanFiles) throws RemoteException
{
        Cursor c = null;
        String where = null;
        String[] selectionArgs = null;
        // mPlayLists 保存从数据库中获取的信息
        if (mPlayLists == null) {
            mPlayLists = new ArrayList<FileEntry>();
        } else {
            mPlayLists.clear();
        }

        if (filePath != null) {
            //只有一个文件查询
            where = MediaStore.Files.FileColumns._ID + ">?" +
                " AND " + Files.FileColumns.DATA + "=?";
            selectionArgs = new String[] { "", filePath };
        } else {
            where = MediaStore.Files.FileColumns._ID + ">?";
            selectionArgs = new String[] { "" };
```

```
}

//告诉提供者不删除文件，如果不需要删除文件，则需要避免意外删除这个文件的机制
//可能在系统未被安装和未安装在扫描仪之前发生
Uri.Builder builder = mFilesUri.buildUpon();
builder.appendQueryParameter(MediaStore.PARAM_DELETE_DATA, "false");
MediaBulkDeleter deleter = new MediaBulkDeleter(mMediaProvider, builder.build());

//根据内容提供者建立文件列表
try {
    if (prescanFiles) {
        //首先从文件表读到现有文件
        // 因为可能存在删除不存在文件的情况,所以要小批量地实现数据库查询用以避免这个问题。
        long lastId = Long.MIN_VALUE;
        Uri limitUri=mFilesUri.buildUpon().appendQueryParameter("limit","1000").build();
        mWasEmptyPriorToScan = true;

        while (true) {
            selectionArgs[0] = "" + lastId;
            if (c != null) {
                c.close();
                c = null;
            }
            c = mMediaProvider.query(limitUri, FILES_PRESCAN_PROJECTION,
                    where, selectionArgs, MediaStore.Files.FileColumns._ID, null);
            if (c == null) {
                break;
            }

            int num = c.getCount();

            if (num == 0) {
                break;
            }
            mWasEmptyPriorToScan = false;
            while (c.moveToNext()) {
                long rowId = c.getLong(FILES_PRESCAN_ID_COLUMN_INDEX);
                String path = c.getString(FILES_PRESCAN_PATH_COLUMN_INDEX);
                int format = c.getInt(FILES_PRESCAN_FORMAT_COLUMN_INDEX);
                long lastModified = c.getLong(FILES_PRESCAN_DATE_MODIFIED_COLUMN_
                INDEX);
                lastId = rowId;
                if (path != null && path.startsWith("/")) {
                    boolean exists = false;
                    try {
                        exists = Libcore.os.access(path, libcore.io.OsConstants.F_OK);
                    } catch (ErrnoException e1) {
                    }
                    if (!exists && !MtpConstants.isAbstractObject(format)) {
                        MediaFile.MediaFileType mediaFileType = MediaFile.
                        getFileType(path);
                        int fileType = (mediaFileType == null ? 0 : mediaFileType.
                        fileType);

                        if (!MediaFile.isPlayListFileType(fileType)) {
                            deleter.delete(rowId);
                            if(path.toLowerCase(Locale.US).endsWith("/.nomedia")) {
                                deleter.flush();
                                String parent = new File(path).getParent();
                                mMediaProvider.call(MediaStore.UNHIDE_CALL, parent,
                                null);
                            }
                        }
                    }
                }
            }
        }
    }
}
finally {
    if (c != null) {
```

```
            c.close();
        }
        deleter.flush();
    }

    //计算图像的原始尺寸
    mOriginalCount = 0;
    c = mMediaProvider.query(mImagesUri, ID_PROJECTION, null, null, null, null);
    if (c != null) {
        mOriginalCount = c.getCount();
        c.close();
    }
}
```

（4）删除不是 SD 卡中的文件信息

在文件 MediaScanner.java 中，函数 postscan 的功能是删除不存在于 SD 卡中的文件信息，具体实现代码如下所示：

```
private void postscan(String[] directories) throws RemoteException {

    //触发播放列表后能够知道对应存储的媒体文件
    if (mProcessPlaylists) {
        processPlayLists();
    }

    if (mOriginalCount == 0 && mImagesUri.equals(Images.Media.getContentUri("external")))
            pruneDeadThumbnailFiles();

    //允许 GC 清理
    mPlayLists = null;
    mMediaProvider = null;
}
```

（5）processDirectory

在文件 MediaScanner.java 中，本地方法 processDirectory 能够直接转向 JNI，具体实现代码如下所示：

```
static void android_media_MediaScanner_processDirectory(JNIEnv *env, jobject thiz,
jstring path, jstring extensions, jobject client)
{   //获取 MediaScanner
    MediaScanner *mp = (MediaScanner *)env->GetIntField(thiz, fields.context);
    //参数判断，并抛出异常
    if (path == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }
    if (extensions == NULL) {
        jniThrowException(env, "java/lang/IllegalArgumentException", NULL);
        return;
    }

    const char *pathStr = env->GetStringUTFChars(path, NULL);
    if (pathStr == NULL) {
        jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
        return;
    }
    const char *extensionsStr = env->GetStringUTFChars(extensions, NULL);
    if (extensionsStr == NULL) {
        env->ReleaseStringUTFChars(path, pathStr);
        jniThrowException(env, "java/lang/RuntimeException", "Out of memory");
        return;
    }
    //初始化 client 实例
    MyMediaScannerClient myClient(env, client);
    //mp 调用 processDirectory
    mp->processDirectory(pathStr, extensionsStr, myClient, ExceptionCheck, env);
    //gc
    env->ReleaseStringUTFChars(path, pathStr);
    env->ReleaseStringUTFChars(extensions, extensionsStr);
}
```

（6）扫描函数 scanFile

函数 scanFile 的功能是调用函数 doScanFile 对指定的文件进行扫描，具体实现代码如下所示：

```
public void scanFile(String path, long lastModified, long fileSize,
        boolean isDirectory, boolean noMedia) {
    //这是来自本地代码的回调函数
    // Log.v(TAG, "scanFile: "+path);
    doScanFile(path, null, lastModified, fileSize, isDirectory, false, noMedia);
}
```

（7）异常处理

在 Android 5.0 中，为了处理 Java 实现的方法中和 C/C++实现方法中抛出的 Java 异常，JNI 特意提供了一套异常处理机制函数集，以专门用于检查、分析和处理异常情况。例如，在文件 jni.h 中定义了主要的异常函数，具体代码如下所示：

```
//抛出异常
jint (*Throw)(JNIEnv*, jthrowable);
//抛出新的异常
jint (*ThrowNew)(JNIEnv *, jclass, const char *);
//异常产生
jthrowable  (*ExceptionOccurred)(JNIEnv*);
void (*ExceptionDescribe)(JNIEnv*);
//清除异常
void (*ExceptionClear)(JNIEnv*);
void (*FatalError)(JNIEnv*, const char*);
```

例如，在 Camera 模块中也用到了异常处理，在文件 android_hardware_Camera.cpp 中，也涉及了异常操作，具体代码例如下所示：

```
void JNICameraContext::copyAndPost(JNIEnv* env, const sp<IMemory>& dataPtr, int msgType)
{
  ......
  if (obj == NULL) {
            LOGE("Couldn't allocate byte array for JPEG data");
            env->ExceptionClear();
        } else {
            env->SetByteArrayRegion(obj, 0, size, data);
        }
    } else {
        LOGE("image heap is NULL");
    }
  }
  ......
}
```

在文件 android_hardware_Camera.cpp 中，函数 android_hardware_Camera_startPreview()也同样用到了异常处理机制，具体代码如下所示：

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述代码中，android_hardware_Camera_startPreview()如果发现 startPreview()函数返回错误，则会抛出异常并返回。这里的异常与 Java 中的异常机制很相似，读者可以对比分析它们的原理。

3.2.2 分析 JNI 层

由于 Android 的应用层的类都是用 Java 写的，当这些 Java 类被编译为 Dex 形式的 Bytecode（位元码，是一个程序处理的电脑目标代码，通常是指虚拟机，而不是真的电脑机或硬件处理器）后，必须借助 Dalvik 虚拟机来执行并实现。虚拟机在 Android 系统中扮演了一个很重要的角色，

并且在执行 Java 类的过程中，如果 Java 类需要与 C 组件沟通时，VM 就会去载入 C 组件，然后让 Java 的函数顺利地调用到 C 组件的函数。此时，VM 扮演着桥梁的角色，让 Java 与 C 组件能够通过标准的 JNI 媒介相互沟通。

应用层的 Java 类是在虚拟机上执行的，而 C 组件不在 Android 虚拟机上执行。如果 Java 程序要求 Android 虚拟机载入（Load）所指定的 C 组件，可以使用如下所示的指令实现这一功能：

```
System.loadLibrary(*.so 的档案名);
```

例如，在 Android 5.0 的框架中，文件 MediaPlayer.java 包含了如下所示的指令：

```
public class MediaPlayer{
        static {
                System.loadLibrary("media_jni");
            }
    }
```

这要求 Android 虚拟机载入 Android 的/system/lib/libmedia_jni.so 库。载入*.so 后，Java 类与*.so 档案就汇合起来一起执行。

在 JNI 层中，MediaScanner 的对应文件是：

```
./frameworks/base/media/jni/android_media_MediaScanner.cpp
```

在接下来的内容中，将详细讲解 MediaScanner 系统 JNI 层的基本源代码。

（1）将指针保存到 Java 对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_init 的功能是将 Native 对象的指针保存到 Java 对象中。函数 android_media_MediaScanner_native_init 的具体实现代码如下所示：

```
static const char* const kClassMediaScanner =
        "android/media/MediaScanner";
    ........

/*native_init 函数的 JNI 层实现*/
static void android_media_MediaScanner_native_init(JNIEnv *env)
{
    ALOGV("native_init");
    jclass clazz = env->FindClass(kClassMediaScanner);
    if (clazz == NULL) {
        return;
    }
    fields.context = env->GetFieldID(clazz, "mNativeContext", "I");
    if (fields.context == NULL) {
        return;
    }
}
```

（2）创建 Native 层的 MediaScanner 对象

在文件 android_media_MediaScanner.cpp 中，函数 android_media_MediaScanner_native_setup 功能是创建一个 Native 层的 MediaScanner 对象，但是，此函数使用的是 Opencore 提供的 PVMediaScanner。函数 android_media_MediaScanner_native_setup 的具体实现代码如下所示：

```
static void android_media_MediaScanner_native_setup(JNIEnv *env, jobject thiz)
{
    ALOGV("native_setup");
    MediaScanner *mp = new StagefrightMediaScanner;
    if (mp == NULL) {
        jniThrowException(env, kRunTimeException, "Out of memory");
        return;
    }
    env->SetIntField(thiz, fields.context, (int)mp);
}
```

3.2.3　分析 Native（本地）层

在现实应用中，Java 的 Native 函数与 JNI 函数是一一对应的关系。在 Android 5.0 中，使用

JNI NativeMethod 的结构体来记录这种对应关系。在接下来的内容中，将详细分析 Mediascanner 系统中的 Native 层的实现源代码。

（1）注册 JNI 函数

在 Andoird 系统中，使用了一种“特定”的方式来定义 Native 函数，这与传统定义 Java JNI 的方式有所差别。其中很重要的区别是在 Andorid 中使用了一种 Java 和 C 函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型是 JNINativeMethod，具体定义如下所示：

```
typedef struct {
    const char* name;           /*Java 中函数的名字*/
    const char* signature;      /*描述了函数的参数和返回值*/
    void* fnPtr;                /*函数指针，指向 C 函数*/
} JNINativeMethod;
```

在上述代码中，比较难以理解的是第二个参数，例如：

```
"()V"
"(II)V"
"(Ljava/lang/String;Ljava/lang/String;)V"
```

实际上这些字符是与函数的参数类型一一对应的，具体说明如下所示。

- "()" 中的字符表示参数，后面的则代表返回值。例如"()V" 就表示 void Func()。
- "(II)V" 表示 void Func(int, int)。

具体的每一个字符的对应关系如下所示。

```
字符      Java 类型      C 类型
V          void          void
Z         jboolean       boolean
I           jint          int
J          jlong          long
D         jdouble         double
F          jfloat         float
B         jbyte           byte
C         jchar           char
S         jshort          short
```

而数组则以“[”开始，用两个字符表示。

```
[I        jintArray        int[]
[F       jfloatArray       float[]
[B       jbyteArray        byte[]
[C       jcharArray        char[]
[S       jshortArray       short[]
[D       jdoubleArray      double[]
[J        jlongArray       long[]
[Z       jbooleanArray     boolean[]
```

上面的都是基本类型，如果 Java 函数的参数是 class，则以"L"开头，以";"结尾，中间部分是用"/"隔开的包及类名。而其对应的 C 函数名的参数则为 jobject. 一个例外是 String 类，其对应的类为 jstring，即：

- Ljava/lang/String 中的 String jstring；
- Ljava/net/Socket 中的 Socket jobject。

如果 Java 函数位于一个嵌入类，则使用$作为类名间的分隔符，例如：

```
(Ljava/lang/String;Landroid/os/FileUtils$FileStatus;)Z"
```

定义并注册 JNINativeMethod 数组，对应的实现代码如下所示：

```
/*定义一个 JNINativeMethod 数组*/
static JNINativeMethod gMethods[] = {
    {
        "processDirectory",
        "(Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
        (void *)android_media_MediaScanner_processDirectory
    },

    {
        "processFile",
```

```
        "(Ljava/lang/String;Ljava/lang/String;Landroid/media/MediaScannerClient;)V",
        (void *)android_media_MediaScanner_processFile
    },

    {
        "setLocale",
        "(Ljava/lang/String;)V",
        (void *)android_media_MediaScanner_setLocale
    },

    {
        "extractAlbumArt",
        "(Ljava/io/FileDescriptor;)[B",
        (void *)android_media_MediaScanner_extractAlbumArt
    },

    {
        "native_init",
        "()V",
        (void *)android_media_MediaScanner_native_init
    },

    {
        "native_setup",
        "()V",
        (void *)android_media_MediaScanner_native_setup
    },

    {
        "native_finalize",
        "()V",
        (void *)android_media_MediaScanner_native_finalize
    },
};
/*注册 JNINativeMethod 数组*/
int register_android_media_MediaScanner(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
                kClassMediaScanner, gMethods, NELEM(gMethods));
}
```

（2）实现注册工作

定义并注册数组 JNINativeMethod 后，接着需要在在文件 AndroidRuntime.cpp 中调用函数 registerNativeMethods 来完成调用工作，具体实现代码如下所示：

```
int AndroidRuntime::registerNativeMethods(JNIEnv* env,
    const char* className, const JNINativeMethod* gMethods, int numMethods)
{
    return jniRegisterNativeMethods(env, className, gMethods, numMethods);
}
```

在上述代码中，函数 jniRegisterNativeMethods 在文件 JNIHelp.cpp 中实现，这是 Android 为方便 JNI 使用而提供的一个帮助函数，具体实现代码如下所示：

```
extern "C" int jniRegisterNativeMethods(C_JNIEnv* env, const char* className,
    const JNINativeMethod* gMethods, int numMethods)
{
    JNIEnv* e = reinterpret_cast<JNIEnv*>(env);

    ALOGV("Registering %s natives", className);
    scoped_local_ref<jclass> c(env, findClass(env, className));
    if (c.get() == NULL) {
        ALOGE("Native registration unable to find class '%s', aborting", className);
        abort();
    }
    if ((*env)->RegisterNatives(e, c.get(), gMethods, numMethods) < 0) {
        ALOGE("RegisterNatives failed for '%s', aborting", className);
        abort();
    }
    return 0;
}
```

通过上述代码可以了解函数 registerNativeMethods 的作用。应用层级的 Java 类别透过 Android 虚拟机呼叫到本地函数，这个过程通常是通过 Android 虚拟机去寻找“*.so”格式库文件中的本地函数。如果需要连续呼叫很多次，则需要每次都寻找一遍，这会花费很多时间。此时，组件开发人员可以自行向 Android 虚拟机登记本地函数。例如，在 Android 的/system/lib/libmedia_jni.so 档案里的代码片段如下所示：

```
#define LOG_TAG "MediaPlayer-JNI"
static JNINativeMethod gMethods[] = {
    {"setDataSource","(Ljava/lang/String;)V",
     (void *)android_media_MediaPlayer_setDataSource},
    {"setDataSource","(Ljava/io/FileDescriptor;JJ)V",
(void *)android_media_MediaPlayer_setDataSourceFD},

    {"prepare","()V",(void *)android_media_MediaPlayer_prepare},
    {"prepareAsync","()V",(void *)android_media_MediaPlayer_prepareAsync},
    {"_start","()V",(void *)android_media_MediaPlayer_start},
    {"_stop","()V",(void *)android_media_MediaPlayer_stop},
    {"getVideoWidth","()I",(void *)android_media_MediaPlayer_getVideoWidth},
    {"getVideoHeight","()I",(void *)android_media_MediaPlayer_getVideoHeight},
    {"seekTo","(I)V",(void *)android_media_MediaPlayer_seekTo},
    {"_pause","()V",(void *)android_media_MediaPlayer_pause},
    {"isPlaying","()Z",(void *)android_media_MediaPlayer_isPlaying},
    {"getCurrentPosition","()I",(void *)android_media_MediaPlayer_getCurrentPosition},
    {"getDuration","()I",(void *)android_media_MediaPlayer_getDuration},
    {"_release","()V",(void *)android_media_MediaPlayer_release},
    {"_reset","()V",(void *)android_media_MediaPlayer_reset},
    {"setAudioStreamType","(I)V", (void *)android_media_MediaPlayer_setAudioStreamType},
    {"setLooping","(Z)V",(void *)android_media_MediaPlayer_setLooping},
    {"setVolume","(FF)V",(void *)android_media_MediaPlayer_setVolume},
    {"getFrameAt","(I)Landroid/graphics/Bitmap;",
        (void *)android_media_MediaPlayer_getFrameAt},
    {"native_setup","(Ljava/lang/Object;)V",
          (void *)android_media_MediaPlayer_native_setup},
    {"native_finalize","()V",(void *)android_media_MediaPlayer_native_finalize},};
static int register_android_media_MediaPlayer(JNIEnv *env){
    return AndroidRuntime::registerNativeMethods(env,
              "android/media/MediaPlayer", gMethods, NELEM(gMethods));
}
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
}
```

这样当 Android 虚拟机载入 libmedia_jni.so 档案时，就会呼叫函数 JNI_OnLoad()，然后 JNI_OnLoad()呼叫函数 register_android_media_MediaPlayer()。此时，就呼叫到函数 AndroidRuntime::registerNativeMethods()，并向 Android 虚拟机（即 AndroidRuntime）登记在数组 gMethods[]中所包含的本地函数。由此可见，函数 registerNativeMethods 具备如下所示的两个功能。

- 更有效率地找到函数。
- 可以在执行期间进行抽换。因为 gMethods[]是一个“<名称，函数指针>”格式的对照表，所以在执行程序时，可以通过多次呼叫函数 registerNativeMethods()的方式来更换本地函数的指针。

（3）实现动态注册

当 Java 层通过 System.loadLibrary 加载完 JNI 动态库后，接着会查找函数 JNI_OnLoad，通过调用文件 android_media_MediaPlayer.cpp 中的函数 JNI_OnLoad 来完成动态注册工作，具体实现代码如下所示：

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

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        ALOGE("ERROR: GetEnv failed\n");
```

```
        goto bail;
    }
    assert(env != NULL);
     ......
    if (register_android_media_MediaScanner(env) < 0) {
        ALOGE("ERROR: MediaScanner native registration failed\n");
        goto bail;
    }
    .......
    /*成功——则返回有效的版本号*/
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

在上述代码中，函数 JNI_OnLoad 会回传 JNI_VERSION_1_4 的值给 Android 虚拟机，这时 Android 虚拟机能够知道所使用 JNI 的版本是什么。此外，它也做了一些初期的动作（可呼叫任何本地函数），例如下面的指令：

```
if (register_android_media_MediaPlayer(env) < 0) {
        LOGE("ERROR: MediaPlayer native registration failed\n");
        goto bail;
    }
```

这样就将此组件提供的各个本地函数(Native Function)登记到 Android 虚拟机里，以便能加快后续呼叫本地函数的效率。

函数 JNI_OnUnload()与 JNI_OnLoad()是相对应的。在载入 C 组件时会立即呼叫 JNI_OnLoad()进行组件内的初期动作。当 Android 虚拟机释放该 C 组件时，则会呼叫 JNI_OnUnload()函数来进行善后清除动作。当 VM 呼叫 JNI_OnLoad()或 JNI_Unload()函数时，都会将 Android 虚拟机的指针（Pointer）传递给它们，其具体参数如下所示：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved) {     }
jint JNI_OnUnload(JavaVM* vm, void* reserved){     }
```

在 JNI_OnLoad()函数里，通过 Android 取得 JNIEnv 指针的值，并保存到 env 指针变量里，如下述指令：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
}
```

因为 Android 虚拟机通常是多线程（Multi-threading）的执行环境。每一个执行绪在呼叫 JNI_OnLoad()时，所传递进来的 JNIEnv 指标值都是不同的。为了配合这种多执行绪的环境，C 组件开发者在撰写本地函数时，可借用由 JNIEnv 指标值的不同而避免执行绪的资料冲突问题，才能确保所写的本地函数能安全地在 Android 虚拟机里安全地执行。基于这个原因，在呼叫 C 组件的函数时会将 JNIEnv 指标值传递给它，对应的实现代码如下所示：

```
jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
JNIEnv* env = NULL;
    if (register_android_media_MediaPlayer(env) < 0) {
   }
}
```

这样当 JNI_OnLoad()呼叫函数 register_android_media_MediaPlayer(env)时，就将 env 指标值传递过去。这样函数 register_android_media_MediaPlayer()就能借用该标识值来区别不同的执行，以便解决资料冲突的问题。

例如，在 register_android_media_MediaPlayer()函数里可以编写如下所示的指令。

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
}
```

此时可以查看是否有其他执行程序进入此物件，如果没有，则此执行就进入该物件里执行了。并且也可以编写如下所示的指令：

```
if ((*env)->MonitorExit(env, obj) != JNI_OK) {
}
```

这样便可以查看是否此执行正在此物件内执行，如果是，此执行就会立即离开。

（4）处理路径参数

在文件 frameworks/base/media/libmedia/MediaScanner.cpp 中，函数 processDirectory 的功能是调用 doProcessDirectory 处理路径参数。里面的参数“@extensions”可能包含多个扩展名，在扩展名之间用“,”分隔开。函数 processDirectory 的具体实现代码如下所示：

```
status_t MediaScanner::processDirectory(
        const char *path, const char *extensions,
        MediaScannerClient &client,
        ExceptionCheck exceptionCheck, void *exceptionEnv) {
    int pathLength = strlen(path);
    if (pathLength >= PATH_MAX) {
        return UNKNOWN_ERROR;
    }
    char* pathBuffer = (char *)malloc(PATH_MAX + 1);
    if (!pathBuffer) {
        return UNKNOWN_ERROR;
    }

    int pathRemaining = PATH_MAX - pathLength;
    strcpy(pathBuffer, path);
    if (pathLength > 0 && pathBuffer[pathLength - 1] != '/') {
        pathBuffer[pathLength] = '/';
        pathBuffer[pathLength + 1] = 0;
        --pathRemaining;
    }

    client.setLocale(locale());

    status_t result =
        doProcessDirectory(
                pathBuffer, pathRemaining, extensions, client,
                exceptionCheck, exceptionEnv);

    free(pathBuffer);

    return result;
}
```

（5）扫描文件

当收到扫描某个文件的请求时，会调用函数 scanFile 来扫描这个文件。函数 scanFile 的具体实现代码如下所示：

```
virtual bool scanFile(const char* path, long long lastModified, long long fileSize)
 {
    jstring pathStr;
    if ((pathStr = mEnv->NewStringUTF(path)) == NULL) return false;
    //调用 Java 里面 mClient 中的 scanFile 方法
    mEnv->CallVoidMethod(mClient, mScanFileMethodID, pathStr, lastModified, fileSize);
    mEnv->DeleteLocalRef(pathStr);
    return (!mEnv->ExceptionCheck());
 }
```

（6）添加 TAG 信息

在文件\frameworks\av\media\libmedia\MediaScannerClient.cpp 中，通过函数 addStringTag 添加 TAG 信息。这个 MediaScannerClient 是在 opencore 中的文件 MediaScanner.cpp 实现的，而文件 android_media_MediaScanner.cpp 中的 MyMediaScannerClient 是从 MediaScannerClient 派生下来的。函数 addStringTag 的具体实现代码如下所示：

```
status_t MediaScannerClient::addStringTag(const char* name, const char* value)
```

```
{
    if (mLocaleEncoding != kEncodingNone) {
        //不要缓存都是 ASCII 字符串
        //呼叫 handlestringtag 直接代替
        //查看值中是否有非 ASCII 字符，应该是 UTF8 )
        bool nonAscii = false;
        const char* chp = value;
        char ch;
        while ((ch = *chp++)) {
            if (ch & 0x80) {
                nonAscii = true;
                break;
            }
        }
        //判断 name 和 value 的编码是不是 ASCII，不是则保存到 mNames 和 mValues 中
            // save the strings for later so they can be used for native encoding detection
            mNames->push_back(name);
            mValues->push_back(value);
            return OK;
        }
        //其他的失败情形
    }
   //如果字符编码是 ASCII，则调用函数 handleStringTag
    return handleStringTag(name, value);
}
```

（7）JNI 中的环境变量

在 Android 的所有模块的 JNI 层的代码中，会看到很多函数中都有 JNIEnv*类型的参数，例如文件/frameworks/base/core/jni/android_hardware_Camera.c 中的如下代码：

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    LOGV("startPreview");
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
```

在上述的函数中，第一个参数为 JNIEnv *Env，此处，JNIEnv *类型是一个指向 JNI 环境的指针，JNIEnv *类型在文件 Jni.h 中定义，在此结构体中包含了一些 JNI 中常用到的函数和一组函数指针，C/C++正是通过这些函数指针来操作 Java 函数的。JNIEnv 结构体在文件 jni.h 中定义，具体实现代码如下所示：

```
struct _JNIEnv {
    ……
    jint GetVersion()
    { return functions->GetVersion(this); }
    ……
    jclass FindClass(const char* name)
   { return functions->FindClass(this, name); }
   ……
    void CallVoidMethodA(jobject obj, jmethodID methodID, jvalue* args)
    { functions->CallVoidMethodA(this, obj, methodID, args); }
   ……
   jmethodID GetStaticMethodID(jclass clazz, const char* name, const char* sig)
    { return functions->GetStaticMethodID(this, clazz, name, sig); }
   ……
}
```

通过上述代码可以发现，正是通过 JNIEnv 这个指针，才能够调用一些 JNI 环境中的方法。

3.3 分析 Camera 系统的 JNI

在本节的内容中，将以 Camera 系统中的预览功能作为素材，在 Android 源代码中详细分析

JNI 机制衔接 Java 层和 C/C++层的方法，剖析 Java 层调用底层代码实现预览功能的具体流程。

3.3.1 Java 层预览接口

在本小节的主要内容中，将详细介绍 Camera 模块中预览功能的 Java 层的文件路径，以及其中预览函数的功能作用。Camera 中的 Java 层代码在 Camera.java 文件中实现，其详细路径为：

```
/Package/apps/camera/src/com/android/camemra/Camera.java
```

在文件 Camera.java 中定义了与预览相关的函数 startPreview()和 stopPreview()，这是图像预览的入口函数。在文件 Camera.java 中，函数 startPreview()和函数 stopPreview()的具体实现代码如下所示：

```
    //开始预览
    private void startPreview() {
    if (mPausing || isFinishing()) return;
    mFocusManager.resetTouchFocus();
    mCameraDevice.setErrorCallback(mErrorCallback);
    // If we're previewing already, stop the preview first (this will blank
    // the screen).
    if (mCameraState != PREVIEW_STOPPED) stopPreview();
    setPreviewDisplay(mSurfaceHolder);
    setDisplayOrientation();
    if (!mSnapshotOnIdle) {
        // If the focus mode is continuous autofocus, call cancelAutoFocus to
        // resume it because it may have been paused by autoFocus call.
        if (Parameters.FOCUS_MODE_CONTINUOUS_PICTURE.equals(mFocusManager.getFocusMode())) {
            mCameraDevice.cancelAutoFocus();
        }
        mFocusManager.setAeAwbLock(false); // Unlock AE and AWB.
    }
    //设置 Camera 的参数
    setCameraParameters(UPDATE_PARAM_ALL);
    if (mCameraPreviewThread != null) {
        synchronized (mCameraPreviewThread) {
            mCameraPreviewThread.notify();
        }
    }
    try {
        Log.v(TAG, "startPreview");
         //调用框架层的 Camera 类来实现预览功能
         mCameraDevice.startPreview();
    } catch (Throwable ex) {
        closeCamera();
        throw new RuntimeException("startPreview failed", ex);
    }
    mZoomState = ZOOM_STOPPED;
    setCameraState(IDLE);
    mFocusManager.onPreviewStarted();
    if (mSnapshotOnIdle) {
        mHandler.post(mDoSnapRunnable);
    }
}
//停止预览
private void stopPreview() {
    //判断 Camera 的状态
    if (mCameraDevice != null && mCameraState != PREVIEW_STOPPED) {
        Log.v(TAG, "stopPreview");
        mCameraDevice.cancelAutoFocus(); // Reset the focus.
        mCameraDevice.stopPreview();
        mFaceDetectionStarted = false;
    }
    //设置 Camera 的状态
    setCameraState(PREVIEW_STOPPED);
    mFocusManager.onPreviewStopped();
}
```

上述代码演示了 Java 层的函数功能，在 Android 的框架层封装了 Camera 的框架层类 Camera，此类的具体路径为：

```
frameworks/base/code/java/android/hardware/Camera.java
```

在类 Camera 中声明了很多 native 的方法，如 startPreview()和 stopPreview()，具体声明代码如下所示：

```
public native final void startPreview();
public native final void stopPreview();
```

上述声明的函数 native 会直接注册到 JNI 中，然后调用 C/C++层的 startPreview()和 stopPrevire()函数。

在文件 android_hardware_Camera.cpp 中实现注册 Camera 预览函数的功能，此文件的具体文件路径为：

```
frameworks/base/core/jni/android_hardware_Camera.cpp
```

3.3.2　注册预览的 JNI 函数

在本小节的主要内容中，将详细介绍将 Camera 模块的预览功能注册到 JNI 系统的方法。在文件 android_hardware_Camera.cpp 里，会将 Camera 模块中的所有接口函数注册到 JNI 系统中，文件 android_hardware_Camera.cpp 中的具体注册代码为：

```
//初始化 JNI 中 Java 对象并且注册 Camera 模块的 JNI 函数
int register_android_hardware_Camera(JNIEnv *env)
{
    field fields_to_find[] = {
        { "android/hardware/Camera", "mNativeContext",   "I", &fields.context },
        { "android/view/Surface",    ANDROID_VIEW_SURFACE_JNI_ID,
          "I",&fields.surface },
        { "android/graphics/SurfaceTexture",
          ANDROID_GRAPHICS_SURFACETEXTURE_JNI_ID, "I", &fields.surfaceTexture },
        { "android/hardware/Camera$CameraInfo", "facing",   "I",
          &fields.facing },
        { "android/hardware/Camera$CameraInfo", "orientation",   "I",
          &fields.orientation },
        { "android/hardware/Camera$Face", "rect", "Landroid/graphics/Rect;",
          &fields.face_rect },
        { "android/hardware/Camera$Face", "score", "I", &fields.face_score },
        { "android/graphics/Rect", "left", "I", &fields.rect_left },
        { "android/graphics/Rect", "top", "I", &fields.rect_top },
        { "android/graphics/Rect", "right", "I", &fields.rect_right },
        { "android/graphics/Rect", "bottom", "I", &fields.rect_bottom },
    };
    if (find_fields(env, fields_to_find, NELEM(fields_to_find)) < 0)
        return -1;
    jclass clazz = env->FindClass("android/hardware/Camera");
    fields.post_event = env->GetStaticMethodID(clazz, "postEventFromNative",
      "(Ljava/lang/Object;IIILjava/lang/Object;)V");
    if (fields.post_event == NULL) {
        LOGE("Can't find android/hardware/Camera.postEventFromNative");
        return -1;
    }
    clazz = env->FindClass("android/graphics/Rect");
    fields.rect_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.rect_constructor == NULL) {
        LOGE("Can't find android/graphics/Rect.Rect()");
        return -1;
    }
    clazz = env->FindClass("android/hardware/Camera$Face");
    fields.face_constructor = env->GetMethodID(clazz, "<init>", "()V");
    if (fields.face_constructor == NULL) {
        LOGE("Can't find android/hardware/Camera$Face.Face()");
        return -1;
    }
    // Register native functions
//注册接口函数到 JNI 中
    return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
                                                 camMethods, NELEM(camMethods));
```

在上述代码中，函数 register_android_hardware_Camera()的功能是初始化 Java 的 Camera 相关的类对象，并且将接口函数注册到 JNI 中，在文件 android_hardware_Camera.cpp 中，Camera 的函数映射表如下所示：

```
static JNINativeMethod camMethods[] = {
  { "getNumberOfCameras",
    "()I",
    (void *)android_hardware_Camera_getNumberOfCameras },
  { "getCameraInfo",
    "(ILandroid/hardware/Camera$CameraInfo;)V",
    (void*)android_hardware_Camera_getCameraInfo },
  { "native_setup",
    "(Ljava/lang/Object;I)V",
    (void*)android_hardware_Camera_native_setup },
  { "native_release",
    "()V",
    (void*)android_hardware_Camera_release },
  { "setPreviewDisplay",
    "(Landroid/view/Surface;)V",
    (void *)android_hardware_Camera_setPreviewDisplay },
  { "setPreviewTexture",
    "(Landroid/graphics/SurfaceTexture;)V",
    (void *)android_hardware_Camera_setPreviewTexture },
  //开始预览
{ "startPreview",
    "()V",
    (void *)android_hardware_Camera_startPreview },
  //停止预览
{ "_stopPreview",
    "()V",
    (void *)android_hardware_Camera_stopPreview },
  { "previewEnabled",
    "()Z",
    (void *)android_hardware_Camera_previewEnabled },
  { "setHasPreviewCallback",
    "(ZZ)V",
    (void *)android_hardware_Camera_setHasPreviewCallback },
  { "_addCallbackBuffer",
    "([BI)V",
    (void *)android_hardware_Camera_addCallbackBuffer },
  { "native_autoFocus",
    "()V",
    (void *)android_hardware_Camera_autoFocus },
  { "native_cancelAutoFocus",
    "()V",
    (void *)android_hardware_Camera_cancelAutoFocus },
  { "native_takePicture",
    "(I)V",
    (void *)android_hardware_Camera_takePicture },
  { "native_setParameters",
    "(Ljava/lang/String;)V",
    (void *)android_hardware_Camera_setParameters },
  { "native_getParameters",
    "()Ljava/lang/String;",
    (void *)android_hardware_Camera_getParameters },
  { "reconnect",
    "()V",
    (void*)android_hardware_Camera_reconnect },
  { "lock",
    "()V",
    (void*)android_hardware_Camera_lock },
  { "unlock",
    "()V",
    (void*)android_hardware_Camera_unlock },
  { "startSmoothZoom",
    "(I)V",
    (void *)android_hardware_Camera_startSmoothZoom },
  { "stopSmoothZoom",
    "()V",
    (void *)android_hardware_Camera_stopSmoothZoom },
```

```
    { "setDisplayOrientation",
      "(I)V",
      (void *)android_hardware_Camera_setDisplayOrientation },
    { "_startFaceDetection",
      "(I)V",
      (void *)android_hardware_Camera_startFaceDetection },
    { "_stopFaceDetection",
      "()V",
      (void *)android_hardware_Camera_stopFaceDetection},
};
```

根据上述代码可以看到，有了这个函数映射表，则 Camera 的 Java 层接口可以调用到 C/C++层的接口函数，C/C++层的预览函数指针名为 android_hardware_Camera_startPreview()和 android_hardware_Camera_stopPreview()，这两个函数会调用到 C/C++层的函数，在文件 android_hardware_Camera.cpp 中，其具体代码如下所示：

```
static void android_hardware_Camera_startPreview(JNIEnv *env, jobject thiz)
{
    ALOGV("startPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> camera = get_native_camera(env, thiz, NULL);
    if (camera == 0) return;
    //调用 C/C++层的 startPreview()
    if (camera->startPreview() != NO_ERROR) {
        jniThrowRuntimeException(env, "startPreview failed");
        return;
    }
}
static void android_hardware_Camera_stopPreview(JNIEnv *env, jobject thiz)
{
    ALOGV("stopPreview");
    //获得 C/C++层的 Camera 指针
    sp<Camera> c = get_native_camera(env, thiz, NULL);
    if (c == 0) return;
    //调用 C/C++层的 stopPreview()
    c->stopPreview();
}
```

3.3.3　C/C++层的预览函数

Camera 模块的 C/C++层文件路径为/frameworks/av/camera/Camera.cpp，具体的实现代码如下所示：

```
//开始预览
status_t Camera::startPreview()
{
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    //调用其他 so 库的 startPreview()
    return c->startPreview();
}
//停止预览
void Camera::stopPreview()
{
    ALOGV("stopPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return;
    //调用其他 so 库的 stopPreview()
    c->stopPreview();
}
```

通过上述代码发现，文件 Camere.cpp 的功能是实现 Camera 模块中的 C/C++层，然后继续调用更加底层的预览的实现代码。

第4章　分析HAL系统

在Android系统中，硬件抽象层（Hardware Abstract Layer，HAL）在用户空间中运行。HAL能够向下屏蔽硬件驱动模块的实现细节，向上提供硬件访问服务。通过硬件抽象层，Android系统通过如下两层来支持硬件设备：

- 第一层在用户空间中实现；
- 第二层在内核空间中实现。

在本章的内容中，将详细讲解Android 5.0中HAL源代码的基本知识，为读者步入本书后面高级知识的学习打下基础。

4.1 HAL基础

HAL层（硬件抽象层）是位于操作系统内核与硬件电路之间的接口层，其目的在于将硬件抽象化。它隐藏了特定平台的硬件接口细节，为操作系统提供虚拟硬件平台，使其具有硬件无关性，这样就可以在多种平台上进行移植。从软硬件测试的角度来看，软硬件的测试工作都可分别基于硬件抽象层来完成，从此使软硬件测试工作的并行进行成为可能。在本节的内容中，将简要介绍HAL的基础知识。

4.1.1　推出HAL的背景

在Android系统中，推出HAL的目的是为了保护一些硬件提供商的知识产权，为了避开Linux的GPL束缚。谷歌架构师的思路是把控制硬件的动作都放到了Android HAL中，而Linux Driver（驱动）仅负责完成一些简单的数据交互作用，甚至把硬件寄存器空间直接映射到User Space（用户空间）。而Android系统是基于Aparch的License，因此，硬件厂商可以只提供二进制代码，所以，Android只是一个开放的平台，并不是一个开源的平台。也许正是因为Android不遵从GPL，所以，Greg Kroah-Hartman才在2.6.33内核里将Andorid驱动从Linux中删除，当然，从后来Linux 3.3版本开始又将Android重新纳入进来。GPL和硬件厂商目前还是有着无法弥合的裂痕。Android想要把这个问题处理好也是不容易的。

Android系统为什么要把对硬件的支持划分为两层来实现呢？具体来说有如下所示的两个原因。

（1）Linux内核源代码是遵循GPL1协议的，即如果我们在Android系统所使用的Linux内核中添加或者修改了代码，那么就必须将它们公开。因此，如果Android系统像其他的Linux系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能会损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。

（2）Android系统源代码是遵循Apache License2协议的，它允许移动设备厂商添加或者修改Android系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在Android系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为，只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，它封装了硬件的实现细节和参数。这样就可以保护移动设备厂商的利益了。

在Android系统中可以分为如下所示的6种HAL：

- 上层软件；
- 内部以太网；
- 内部通信 CLIENT；
- 用户接入口；
- 虚拟驱动，设置管理模块；
- 内部通信 SERVER。

在 Android 系统中，定义硬件抽象层接口的代码具有以下 5 个特点：

- 硬件抽象层具有与硬件的密切相关性；
- 硬件抽象层具有与操作系统无关性；
- 接口定义的功能应包含硬件或系统所需硬件支持的所有功能；
- 接口定义简单明了，太多接口函数会增加软件模拟的复杂性；
- 具有可测性的接口设计有利于系统的软硬件测试和集成。

在 Android 源代码中，HAL 主要被保存在下面的目录中。

- libhardware_legacy：过去的目录，采取了链接库模块观念来架构。
- libhardware：新版的目录，被调整为用 HAL stub 观念来架构。
- ril：是 Radio 接口层。
- msm7k：和 QUAL 平台相关的信息。

4.1.2　HAL 的基本结构

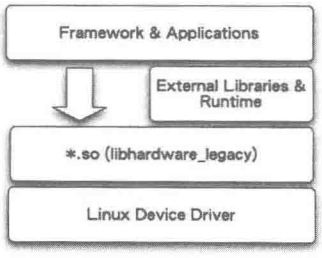
在 Android 系统中，HAL 的位置结构如图 4-1 所示。

▲图 4-1　HAL 层结构

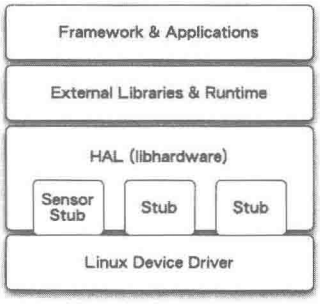
从图 4-1 所示的结构图可以看出，HAL 的功能是把 Android Framework（Android 框架）与 Linux 内核（Linux 内核）隔离。这样 Android 可以不过度依赖 Linux Kernel，从而以在不考虑驱动程序的前提下进行 Framework 层的应用开发工作。在 HAL 层主要包含了 GPS、Vibrator、Wi-Fi、Copybit、Audio、Camera、Lights、Ril、Overlay 等模块。

目前 Android 的 HAL 层仍然分布在不同的位置，所以，诸如 Camera、Wi-Fi 等目录并不包含所有的 HAL 程序代码。在 HAL 架构成熟前的结构如图 4-2 所示，现在 HAL 层的结构如图 4-3 所示。

从现在 HAL 层的结构可以看出，当前的“HAL stub”模式是一种代理人（proxy）的概念，虽然 stub 仍以“*.so”档的形式存在，但是 HAL 已经将“*.so”档隐藏了。Stub 向 HAL 提供了功能强大的操作函数（Operations），而 runtime 则从 HAL 获取特定模块（stub）的函数，然后再回调这些操作函数。这种以“Indirect Function Call”模式的架构，让 HAL stub 变成了一种“包含”关系，也就说在 HAL 里包含了许多 stub（代理人）。Runtime 只要说明 module ID（类型）就可以取得操作函数。在当前的 HAL 模式中，Android 定义了 HAL 层结构框架，这样通过接口访问硬件时就形成了统一的调用方式。



▲图 4-2　成熟前的 HAL 架构



▲图 4-3　现在的 HAL 架构

注意

HAL_legacy 和 HAL 的对比

为了使读者明白过去结构和现在结构的差别，接下来将对 HAL_legacy 和 HAL 做一个对比。

（1）HAL_legacy

这是过去 HAL 的模块，采用共享库形式，在编译时会调用到。由于采用 function call 形式来调用，因此可被多个进程使用，但会被 mapping 到多个进程空间中造成浪费，同时需要考虑代码能否安全重入的问题（thread safe）。

（2）HAL

这是新式的 HAL，采用了 HAL module 和 HAL stub 结合形式。HAL stub 不是一个共享库，在编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。在上层通过 HAL module 提供的统一接口获取并操作 HAL stub，所以文件只会被映射到一个进程，而不会存在重复映射和重入问题。

在 Android 系统中，HAL 层的源码结构如下所示。

（1）/hardware/libhardware_legacy/：旧的 HAL 架构、采取链接库模块的方式。

（2）/hardware/libhardware：新的 HAL 架构，调整为 HAL stub，具体目录的结构如下所示。

- /hardware/libhardware/hardware.c：编译成 libhardware.s，置于/system/lib。
- /hardware/libhardware/include/hardware 目录下包含如下头文件。
  - hardware.h：通用硬件模块头文件。
  - copybit.h：copybit 模块头文件。
  - gralloc.h：gralloc 模块头文件。
  - lights.h：背光模块头文件。
  - overlay.h：overlay 模块头文件。
  - qemud.h：qemud 模块头文件。
  - sensors.h：传感器模块头文件。
  - /hardware/libhardware/modules：在此目录下定义了很多硬件模块，例如，/hardware/msm7k、/hardware/qcom、/hardware/ti、/device/Samsung、/device/moto，这些是各个厂商平台相关的 HAL。

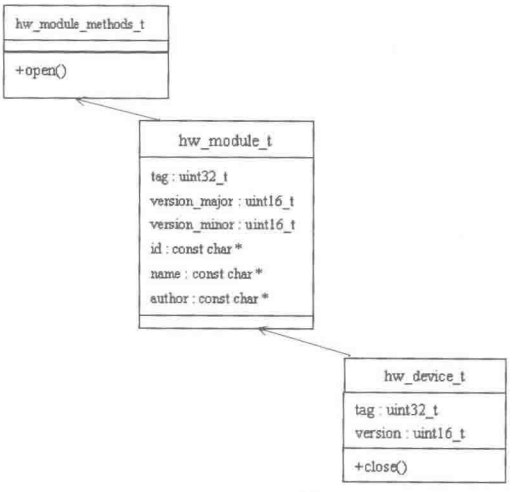
4.2 分析 HAL module 架构

Android 5.0 的 HAL 采用 HAL module 和 HAL stub 结合的形式进行架构，HAL stub 不是一个 Share Library（共享程序），在编译时上层只拥有访问 HAL stub 的函数指针，并不需要 HAL stub。上层通过 HAL module 提供的统一接口获取并操作 HAL stub，so 文件只会被 mapping 到一个进程，也不存在重复 mapping 和重入问题。

在 Android 5.0 系统中，HAL module 架构主要分为如下 3 个结构体：

- struct hw_module_t;
- struct hw_module_methods_t;
- struct hw_device_t。

上述 3 个结构的继承关系如图 4-4 所示。



▲图 4-4 Android HAL 结构的继承关系

以上 3 个抽象概念在文件 hardware.c 中进行了具体描述，而 HAL 模块的源代码保存在

“harware”目录中。对于不同的 hardware 的 HAL，对应的 lib 命名规则是“id.variant.so”，如 gralloc.msm7k.so 表示其 id 是 gralloc，msm7k 是 variant。variant 的取值范围是在该文件中定义的 variant_keys 对应的值。

4.2.1　hw_module_t

结构 hw_module_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示：

```
typedef struct hw_module_t {
    uint32_t tag;
    uint16_t module_api_version;
#define version_major module_api_version
    uint16_t hal_api_version;
#define version_minor hal_api_version
    const char *id;
    const char *name;
    const char *author;
    struct hw_module_methods_t* methods;
    void* dso;
    uint32_t reserved[32-7];

} hw_module_t;
```

在结构体 hw_module_t 中，读者需要注意以下 5 点。

（1）在结构体 hw_module_t 的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为 hw_module_t。

（2）硬件抽象层中的每一个模块都必须存在一个导出符号 HAL_MODULE_IFNO_SYM，即“HMI”，它指向一个自定义的硬件抽象层模块结构体。后面我们在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

（3）结构体 hw_module_t 的成员变量 tag 的值必须设置为 HARDWARE_MODULE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'M'<<8 | 'T'），用来标志这是一个硬件抽象层模块结构体。

（4）结构体 hw_module_t 的成员变量 dso 用来保存加载硬件抽象层模块后得到的句柄值。前面提到，每一个硬件抽象层模块都对应有一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用 dlopen 函数来加载与其对应的动态链接库文件的过程。在调用 dlclose 函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，我们在加载时需要将它保存起来。

（5）结构体 hw_module_t 的成员变量 methods 定义了一个硬件抽象层模块的操作方法列表，它的类型为 hw_module_methods_t，接下来我们就介绍它的定义。hw_module_methods_t 的定义代码如下所示：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);
} hw_module_methods_t;
```

4.2.2　结构 hw_module_methods_t 的定义

结构 hw_module_methods_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示：

```
typedef struct hw_module_methods_t {
    /** Open a specific device */
    int (*open)(const struct hw_module_t* module, const char* id,
            struct hw_device_t** device);

} hw_module_methods_t;
```

在结构体 hw_module_methods_t 中只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数 module 表示要打开的硬件设备所在的模块；参数 id 表示要

打开的硬件设备的 ID；参数 device 是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此在调用结构体 hw_module_methods_t 的成员变量 open，打开一个硬件设备时需要指定它的 ID。

4.2.3 hw_device_t 结构

结构 hw_device_t 在文件 hardware/libhardware/include/hardware/hardware.h 中定义，具体实现代码如下所示：

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);

} hw_device_t;
```

在 Android 系统中，硬件抽象层中的硬件设备使用结构体 hw_device_t 来描述，接下来介绍它的定义。定义 hw_device_t 的代码如下所示：

```
typedef struct hw_device_t {
    uint32_t tag;
    uint32_t version;
    struct hw_module_t* module;
    uint32_t reserved[12];
    int (*close)(struct hw_device_t* device);
} hw_device_t;
```

在结构体 hw_device_t 中，需要注意以下所示的 3 点。

（1）硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为 hw_device_t。

（2）结构体 hw_device_t 的成员变量 tag 的值必须设置为 HARDWARE_DEVICE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'D'<<8 | 'T'），用来标志这是一个硬件抽象层中的硬件设备结构体。

（3）结构体 hw_device_t 的成员变量 close 是一个函数指针，它用来关闭一个硬件设备。

4.3 分析文件 hardware.c

文件 hardware.c 是文件 hardware.h 的具体实现。在本节的内容中，将详细分析 Android 5.0 HAL 模块中文件 hardware.c 的基本源代码。

4.3.1 寻找动态链接库的地址

函数 hw_get_module()能够根据模块 ID 寻找硬件模块动态链接库的地址，然后调用函数 load 打开动态链接库，并从中获取硬件模块结构体地址。执行后首先得到的是根据固定的符号 HAL_MODULE_INFO_SYM 寻找到结构体 hw_module_t，然后是在 hw_moule_t 中 hw_module_methods_t 结构体成员函数提供的结构 open 打开相应的模块，并同时进行初始化操作。因为用户在调用 open()时通常都会传入一个指向 hw_device_t 指针的指针。这样函数 open()将对模块的操作函数结构保存到结构体 hw_device_t 中，用户通过它可以和模块进行交互。

函数 hw_get_module()的实现代码如下所示：

```
int hw_get_module(const char *id, const struct hw_module_t **module)
120 {
121     int status;
122     int i;
123     const struct hw_module_t *hmi = NULL;
124     char prop[PATH_MAX];
```

```
125     char path[PATH_MAX];
    /* Loop through the configuration variants looking for a module */
135     for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
/*
```

4.3.2　数组 variant_keys

在函数 hw_get_module()中需要用到数组 variant_keys，因为 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。定义数组 variant_keys 的代码如下所示：

```
*44 static const char *variant_keys[] = {
* 45     "ro.hardware",  /* This goes first so that it can pick up a different
* 46                        file on the emulator. */
* 47     "ro.product.board",
* 48     "ro.board.platform",
*  49     "ro.arch"
* 50 };
```

然后通过此数组，并使用如下代码得到操作权限：

```
136         if (i < HAL_VARIANT_KEYS_COUNT) {
137             if (property_get(variant_keys[i], prop, NULL) == 0) {
138                 continue;
139             }
```

此处的 variant_keys[i]对应有 3 个值，分别是 trout、msm7k 和 ARMV6。

接下来通过如下代码将路径和文件名保存到 path：

```
140   snprintf(path, sizeof(path), "%s/%s.%s.so",
141     HAL_LIBRARY_PATH, id, prop);
```

通过上述代码，把“HAL_LIBRARY_PATH/id.***.so”保存到 path 中，其中“***”就是上面 variant_keys 中各个元素所对应的值。

4.3.3　载入相应的库

载入相应的库，并把它们的 HMI 保存到 module 中。具体代码如下所示：

```
142         } else {
143             snprintf(path, sizeof(path), "%s/%s.default.so",
144                     HAL_LIBRARY_PATH, id);
145         }
146         if (access(path, R_OK)) {
147             continue;
148         }
149         /* we found a library matching this id/variant */
150         break;
151     }
152
153     status = -ENOENT;
154     if (i < HAL_VARIANT_KEYS_COUNT+1) {
155         /* load the module, if this fails, we're doomed, and we should not try
156          * to load a different variant. */
157         status = load(id, path, module);//load 相应库。并把它们的 HMI 保存到 module 中
158     }
159
  return status;
161 }
```

4.3.4　获得 hw_module_t 结构体

通过函数 load()打开相应的库并获得 hw_module_t 结构体，具体实现代码如下所示：

```
60 static int load(const char *id,
61         const char *path,
62         const struct hw_module_t **pHmi)
63 {
64     int status;
65     void *handle;
```

```
66      struct hw_module_t *hmi;
  handle = dlopen(path, RTLD_NOW);//打开相应的库
74      if (handle == NULL) {
75          char const *err_str = dlerror();
76          LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
77          status = -EINVAL;
78          goto done;
79      }

82      const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
83      hmi = (struct hw_module_t *)dlsym(handle, sym);//获得 hw_module_t 结构体
84      if (hmi == NULL) {
85          LOGE("load: couldn't find symbol %s", sym);
86          status = -EINVAL;
87          goto done;
88      }
89
90      /* Check that the id matches */
91      if (strcmp(id, hmi->id) != 0) {//只是一个 check
92          LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
93          status = -EINVAL;
94          goto done;
95      }
96
97      hmi->dso = handle;
98
99      /* success */
100     status = 0;
    done:
103     if (status != 0) {
104         hmi = NULL;
105         if (handle != NULL) {
106             dlclose(handle);
107             handle = NULL;
108         }
109     } else {
110         LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
111                 id, path, *pHmi, handle);
112     }
113
114     *pHmi = hmi;//得到 hw_module_t
115
116     return status;
117 }
```

4.4 分析硬件抽象层的加载过程

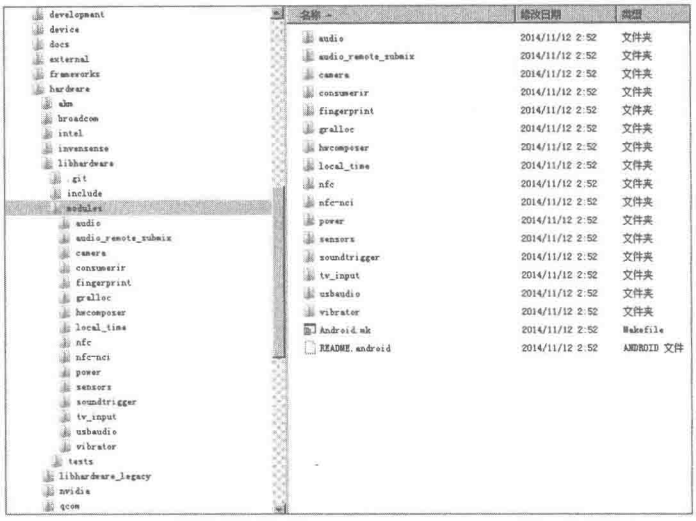
每一个硬件抽象层模块在内核中都对应有一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备，它们是通过读写设备文件来进行通信。硬件抽象层中的模块接口源文件一般保存在“hardware/libhardware”目录中，其目录结构如图 4-5 所示。

Android 系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的 ID 值就可以了。在 Android 硬件抽象层中，负责加载硬件抽象层模块的函数是 hw_get_module，此函数在如下文件中定义：

```
hardware/libhardware/include/hardware/hardware.h
```

函数 hw_get_module 有 id 和 module 两个参数。其中，id 是输入参数，表示要加载的硬件抽象层模块 ID；module 是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于 0，则表示加载成功；如果小于 0，则表示加载失败。函数 hw_get_module 的具体实现代码如下所示：

```
int hw_get_module(const char *id, const struct hw_module_t **module)
{
    return hw_get_module_by_class(id, NULL, module);
}
```



▲图 4-5　libhardware 目录

函数 hw_get_module 在文件 hardware/libhardware/hardware.c 中实现，其中数组 variant_keys 用来组装要加载的硬件抽象层模块的文件名称。常量 HAL_VARIANT_KEYS_COUNT 表示数组 variant_keys 的大小。宏 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 用来定义要加载的硬件抽象层模块文件所在的目录。第 32 行到第 50 行的 for 循环根据数组 variant_keys 在 HAL_LIBRARY_PATH1 和 HAL_LIBRARY_PATH2 目录中检查对应的硬件抽象层模块文件是否存在，如果存在则结束 for 循环；第 56 行调用 load 函数来执行加载硬件抽象层模块的操作。函数 hw_get_module 的具体实现代码如下所示：

```
16 int hw_get_module(const char *id, const struct hw_module_t **module)
17 {
18 int status;
19 int i;
20 const struct hw_module_t *hmi = NULL;
21 char prop[PATH_MAX];
22 char path[PATH_MAX];
23
24 /*
25 * Here we rely on the fact that calling dlopen multiple times on
26 * the same .so will simply increment a refcount (and not load
27 * a new copy of the library).
28 * We also assume that dlopen() is thread-safe.
29 */
30
31 /* Loop through the configuration variants looking for a module */
32 for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
33 if (i < HAL_VARIANT_KEYS_COUNT) {
34 if (property_get(variant_keys[i], prop, NULL) == 0) {
35 continue;
36 }
37
38 snprintf(path, sizeof(path), "%s/%s.%s.so",
39 HAL_LIBRARY_PATH1, id, prop);
40 if (access(path, R_OK) == 0) break;
41
42 snprintf(path, sizeof(path), "%s/%s.%s.so",
43 HAL_LIBRARY_PATH2, id, prop);
44 if (access(path, R_OK) == 0) break;
45 } else {
46 snprintf(path, sizeof(path), "%s/%s.default.so",
47 HAL_LIBRARY_PATH1, id);
48 if (access(path, R_OK) == 0) break;
49 }
50 }
```

```
51
52 status = -ENOENT;
53 if (i < HAL_VARIANT_KEYS_COUNT+1) {
54 /* load the module, if this fails, we're doomed, and we should not try
55 * to load a different variant. */
56 status = load(id, path, module);
57 }
58
59 return status;
60 }
```

编译好的模块文件位于“out/target/product/generic/system/lib/hw”目录中，而这个目录经过打包后，就对应于设备上的“/system/lib/hw”目录。宏 HAL_LIBRARY_PATH2 所定义的目录为“/vendor/lib/hw”，用来保存设备厂商所提供的硬件抽象层模块接口文件。

在上述第 56 行代码中，调用函数 load 执行硬件抽象层模块的加载操作，此函数的具体实现代码如下所示：

```
01 static int load(const char *id,
02 const char *path,
03 const struct hw_module_t **pHmi)
04 {
05 int status;
06 void *handle;
07 struct hw_module_t *hmi;
08
09 /*
10 * load the symbols resolving undefined symbols before
11 * dlopen returns. Since RTLD_GLOBAL is not or'd in with
12 * RTLD_NOW the external symbols will not be global
13 */
14 handle = dlopen(path, RTLD_NOW);
15 if (handle == NULL) {
16 char const *err_str = dlerror();
17 LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
18 status = -EINVAL;
19 goto done;
20 }
21
22 /* Get the address of the struct hal_module_info. */
23 const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
24 hmi = (struct hw_module_t *)dlsym(handle, sym);
25 if (hmi == NULL) {
26 LOGE("load: couldn't find symbol %s", sym);
27 status = -EINVAL;
28 goto done;
29 }
30
31 /* Check that the id matches */
32 if (strcmp(id, hmi->id) != 0) {
33 LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
34 status = -EINVAL;
35 goto done;
36 }
37
38 hmi->dso = handle;
39
40 /* success */
41 status = 0;
42
43 done:
44 if (status != 0) {
45 hmi = NULL;
46 if (handle != NULL) {
47 dlclose(handle);
48 handle = NULL;
49 }
50 } else {
51 LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
52 id, path, *pHmi, handle);
```

```
53 }
54
55 *pHmi = hmi;
56
57 return status;
58 }
```

在上述代码中，第 14 行调用函数 dlopen 将它加载到内存中。加载完成这个动态链接库文件之后，第 24 行就调用函数 dlsym 来获得里面名称为 HAL_MODULE_INFO_SYM_AS_STR 的符号。这个 HAL_MODULE_INFO_SYM_AS_STR 符号指向的是一个自定义的硬件抽象层模块结构体，它包含了对应的硬件抽象层模块的所有信息。HAL_MODULE_INFO_SYM_AS_STR 是一个宏，它的值定义为“HMI”。根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为“HMI”的符号，而且这个符号的第一个成员变量的类型必须定义为 hw_module_t，因此，第 24 行可以安全地将模块中的 HMI 符号转换为一个 hw_module_t 结构体指针。获得了这个 hw_module_t 结构体指针之后，第 32 行调用 strcmp 函数来验证加载得到的硬件抽象层模块 ID 是否与所要求加载的硬件抽象层模块 ID 一致。如果不一致，就说明出错了，函数返回一个错误值 -EINVAL。最后，第 38 行将成功加载后得到的模块句柄值 handle 保存在 hw_module_t 结构体指针 hmi 的成员变量 dso 中，然后将它返回给调用者。

4.5 分析硬件访问服务

当开发好硬件抽象层模块之后，我们通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用 C++语言开发的，而应用程序框架层中的硬件访问服务是使用 Java 语言开发的，因此，硬件访问服务必须通过 Java 本地接口（Java Native Interface，JNI）来调用硬件抽象层模块的接口。在本节的内容中，将详细分析硬件访问服务的基本源代码。

4.5.1　定义硬件访问服务接口

Android 系统的硬件访问服务通常运行在系统进程 System5 中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间通信机制来访问这些硬件访问服务。Android 系统提供了一种高效的进程间通信机制——Binder 进程间通信机制 6，应用程序就是通过它来访问运行在系统进程 System 中的硬件访问服务的。Binder 进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，我们首先要定义它的服务接口。

在 Android 5.0 系统中，提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为 Android 接口描述语言（Android Interface Definition Language，AIDL）。以 AIDL 定义的服务接口文件是以 aidl 为后缀名的，在编译时，编译系统会将它们转换成一个 Java 文件，然后再对它们进行编译。在本节中，我们将使用 AIDL 来定义硬件访问服务接口 IFregService。

在 Android 系统中，通常在 frameworks/base/core/java/android/os 目录中定义硬件访问服务接口，所以把定义了硬件访问服务接口 IFregService 的文件 IFregService.aidl 也保存在这个目录中，其具体代码如下所示：

```
package android.os;
interface IFregService {
void setVal(int val);
int getVal();
}
```

服务接口 IFregService 只定义了两个成员函数，它们分别是 setVal 和 getVal。其中，成员函数 setVal 用来往虚拟硬件设备 freg 的寄存器 val 中写入一个整数，而成员函数 getVal 用来从虚拟硬件设备 freg 的寄存器 val 中读出一个整数。

由于服务接口 IFregService 是使用 AIDL 语言描述的，因此需要将其添加到编译脚本文件中，这样编译系统才能将其转换为 Java 文件，然后再对它进行编译。进入到 frameworks/base 目录中，打开里面的 Andıoid.mk 文件，修改 LOCAL_SRC_FILES 变量的值：

```
LOCAL_SRC_FILES += \
......
voip/java/android/net/sip/ISipService.aidl \
core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，我们就可以使用 mmm 命令对硬件访问服务接口 IFregService 进行编译了：

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的 framework.jar 文件就包含有 IFregService 接口，它继承了 android.os.IInterface 接口。在 IFregService 接口内部，定义了一个 Binder 本地对象类 Stub，它实现了 IFregService 接口，并且继承了 android.os.Binder 类。此外，在 IFregService.Stub 类内部，还定义了一个 Binder 代理对象类 Proxy，它同样也实现了 IFregService 接口。

用 AIDL 定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为 Server 进程，而使用服务的进程称为 Client 进程。在 Server 进程中，每一个服务都对应有一个 Binder 本地对象，它通过一个桩（Stub）来等待 Client 进程发送进程间通信请求。Client 进程在访问运行 Server 进程中的服务之前，首先要获得它的一个 Binder 代理对象接口（Proxy），然后通过这个 Binder 代理对象接口向它发送进程间通信请求。

4.5.2 具体实现

在 Android 系统中，通常通过 frameworks/base/services/java/com/android/server 目录中的文件实现硬件访问服务。因此，把实现了硬件访问服务 FregService 的文件 FregService.java 也保存在这个目录中，其具体内容如下所示：

```
01 package com.android.server;
02
03 import android.content.Context;
04 import android.os.IFregService;
05 import android.util.Slog;
06
07 public class FregService extends IFregService.Stub {
08 private static final String TAG = "FregService";
09
10 private int mPtr = 0;
11
12 FregService() {
13 mPtr = init_native();
14
15 if(mPtr == 0) {
16 Slog.e(TAG, "Failed to initialize freg service.");
17 }
18 }
19
20 public void setVal(int val) {
21 if(mPtr == 0) {
22 Slog.e(TAG, "Freg service is not initialized.");
23 return;
24 }
25
26 setVal_native(mPtr, val);
27 }
28
29 public int getVal() {
30 if(mPtr == 0) {
31 Slog.e(TAG, "Freg service is not initialized.");
32 return 0;
33 }
34
```

```
35 return getVal_native(mPtr);
36 }
37
38 private static native int init_native();
39 private static native void setVal_native(int ptr, int val);
40 private static native int getVal_native(int ptr);
41 };
```

在上述代码中，硬件访问服务 FregService 继承了类 IFregService.Stub，并且实现了 IFregService 接口的成员函数 setVal 和 getVal。其中，成员函数 setVal 通过调用 JNI 方法 setVal_native 来写虚拟硬件设备 freg 的寄存器 val，而成员函数 getVal 调用 JNI 方法 getVal_native 来读虚拟硬件设备 freg 的寄存器 val。在启动硬件访问服务 FregService 时，会通过调用 JNI 函数 init_native 来打开虚拟硬件设备 freg，并且获得它的一个句柄值，保存在成员变量 mPtr 中。如果硬件访问服务 FregService 打开虚拟硬件设备 freg 失败，那么它的成员变量 mPtr 的值就等于 0；否则，就得到一个大于 0 的句柄值。这个句柄值实际上是指向虚拟硬件设备 freg 在硬件抽象层中的一个设备对象，硬件访问服务 FregService 的成员函数 setVal 和 getVal 在访问虚拟硬件设备 freg 的寄存器 val 时，必须要指定这个句柄值，以便硬件访问服务 FregService 的 JNI 实现可以知道它所要访问的是哪一个硬件设备。

4.6 分析 Android 官方实例

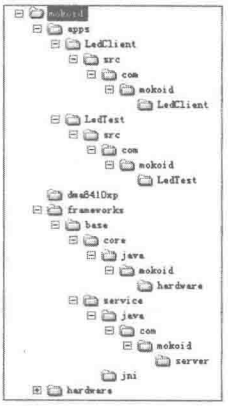
谷歌针对 HAL 提供了一个官方实例工程：Mokoid。在此工程中提供了一个 LedTest 演示程序，此程序实例完整演示了 Android 层次结构和 HAL 架构编程的方法和流程。在本节的内容中，将详细分析 Mokoid 工程的基本源代码。

4.6.1　获取实例工程源代码

读者可以从网络中获取“LedTest”示例程序的源代码，方法是，在 Linux 中使用下面的下载命令：

```
#svn checkout http://mokoid.googlecode.com/svn/trunk/mokoid-read-only
```

下载 Mokoid 工程文件后，其目录结构如图 4-6 所示。



▲图 4-6　Mokoid 工程的目录结构

Mokoid 工程代码树的具体说明如下所示。

```
.
|-- apps  -- 测试应用程序
|   |-- LedClient -- 直接调用 Service 控制硬件
|   |   |-- AndroidManifest.xml
|   |   `-- src
|   |       `-- com
|   |           `-- mokoid
|   |               `-- LedClient
|   |                   `-- LedClient.java
|   `-- LedTest -- 通过 manager 来控制硬件
|       |-- AndroidManifest.xml
|       `-- src
|           `-- com
|               `-- mokoid
|                   `-- LedTest
|                       |-- LedSystemServer.java
|                       `-- LedTest.java
|-- frameworks -- 框架代码
|   `-- base
|       |-- core
|       |   `-- java
|       |       `-- mokoid
|       |           `-- hardware
|       |               |-- ILedService.aidl -- Android Interface Definition Language 代码，提供 LedService 的接口
|       |               `-- LedManager.java -- LedManager 实现代码
```

```
|       `-- service
|           |-- com.mokoid.server.xml
|           |-- java
|           |   `-- com
|           |       `-- mokoid
|           |           `-- server
|           |               `-- LedService.java -- LedService 的 java 实现代码
|           `-- jni
|               `-- com_mokoid_server_LedService.cpp -- LedService 的 jni 实现代码
|-- hardware
|   `-- modules
|       |-- include
|       |   `-- mokoid
|       |       `-- led.h
|       `-- led
|           `-- led.c -- led 实际控制硬件的代码
```

在 Android 系统中需要通过 JNI（Java Native Interface）实现 HAL，因为 JNI 是 Java 程序可以调用 C/C++编写的动态链接库，所以，可以使用 C/C++语言编写 HAL，这样做的好处是，拥有更高的效率。在 Android 系统中有如下两种访问 HAL 的方式。

第一种：Android APP 直接通过 Service 调用“.so”格式的 JNI，虽然这种方式比较简单高效，但是不正规。

第二种：经过 Manager 调用 Service，虽然此方式实现起来比较复杂，但是，更符合目前的 Android 框架。在此方法中，在进程 LegManager 和 LedService（Java）中需要通过进程通信的方式实现通信。

在 Mokoid 工程中分别实现了上述两种方法，在接下来的内容中，将详细介绍这两种方法的具体实现原理。

4.6.2 直接调用 Service 方法的实现代码

（1）HAL 层的实现代码

文件“hardware/modules/led/led.c”的实现代码如下所示：

```
#define LOG_TAG "MokoidLedStub"
#include <hardware/hardware.h>
#include <fcntl.h>
#include <errno.h>
#include <cutils/log.h>
#include <cutils/atomic.h>
#include <mokoid/led.h>
/*****************************************************************************/
int led_device_close(struct hw_device_t* device)
{
    struct led_control_device_t* ctx = (struct led_control_device_t*)device;
    if (ctx) {
        free(ctx);
    }
    return 0;
}
int led_on(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d on.", led);
    return 0;
}
int led_off(struct led_control_device_t *dev, int32_t led)
{
    LOGI("LED Stub: set %d off.", led);
    return 0;
}
static int led_device_open(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    struct led_control_device_t *dev;
    dev = (struct led_control_device_t *)malloc(sizeof(*dev));
    memset(dev, 0, sizeof(*dev));
```

```
    dev->common.tag = HARDWARE_DEVICE_TAG;
    dev->common.version = 0;
    dev->common.module = module;
    dev->common.close = led_device_close;
    dev->set_on = led_on;//实例化支持的操作
    dev->set_off = led_off;
    *device = &dev->common;//将实例化的 led_control_device_t 地址返回给 JNI 层
success:
    return 0;
}
static struct hw_module_methods_t led_module_methods = {
    open: led_device_open
};
const struct led_module_t HAL_MODULE_INFO_SYM = {
  //定义此对象相当于向系统注册了一个 ID 为 LED_HARDWARE_MODULE_ID 的 stub
 common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: LED_HARDWARE_MODULE_ID,
        name: "Sample LED Stub",
        author: "The Mokoid Open Source Project",
        methods: &led_module_methods,//实现了一个 open 的方法供 JNI 层调用
    }
    /* supporting APIs go here */
};
```

（2）JNI 层的实现代码

文件“frameworks/base/service/jni/com_mokoid_server_LedService.cpp”的实现代码如下所示：

```
struct led_control_device_t *sLedDevice = NULL;
static jboolean mokoid_setOn(JNIEnv* env, jobject thiz, jint led) {
    LOGI("LedService JNI: mokoid_setOn() is invoked.");
    if (sLedDevice == NULL) {
        LOGI("LedService JNI: sLedDevice was not fetched correctly.");
        return -1;
    } else {
        return sLedDevice->set_on(sLedDevice, led);//调用 HAL 层的注册方法
    }
}
static jboolean mokoid_setOff(JNIEnv* env, jobject thiz, jint led) {
    LOGI("LedService JNI: mokoid_setOff() is invoked.");
    if (sLedDevice == NULL) {
        LOGI("LedService JNI: sLedDevice was not fetched correctly.");
        return -1;
    } else {
        return sLedDevice->set_on(sLedDevice, led); //调用 HAL 层的注册方法
    }
}
/** helper APIs——JNI 通过 LED_HARDSOFTWARE_MODULE_ID 找到对应的 stub*/
static inline int led_control_open(const struct hw_module_t* module,
        struct led_control_device_t** device) {
    return module->methods->open(module,
            LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
}
static jboolean
mokoid_init(JNIEnv *env, jclass clazz)
{
    led_module_t* module;
    if (hw_get_module(LED_HARDWARE_MODULE_ID, (const hw_module_t**)&module) == 0) {
        LOGI("LedService JNI: LED Stub found.");
        if (led_control_open(&module->common, &sLedDevice) == 0) {
            LOGI("LedService JNI: Got Stub operations.");
            return 0;
        }
    }
    LOGE("LedService JNI: Get Stub operations failed.");
    return -1;
}
// JNINativeMethod 是 JNI 层的注册方法
static const JNINativeMethod gMethods[] = {
```

```
    {"_init","()Z",//Framework 层调用_init 时促发
            (void*)mokoid_init},
    { "_set_on","(I)Z",
                    (void*)mokoid_setOn },
    { "_set_off","(I)Z",
                    (void*)mokoid_setOff },
};
static int registerMethods(JNIEnv* env) {
    static const char* const kClassName =
        "com/mokoid/server/LedService";
    jclass clazz;
    /* 寻找类 class */
    clazz = env->FindClass(kClassName);
    if (clazz == NULL) {
        LOGE("Can't find class %s\n", kClassName);
        return -1;
    }
    /* register all the methods */
    if (env->RegisterNatives(clazz, gMethods,
            sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK)
    {
        LOGE("Failed registering methods for %s\n", kClassName);
        return -1;
    }
    return 0;
}
// ----------------------------------------------------------------------------
////Framework 层加载 JNI 库时调用
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);
    if (registerMethods(env) != 0) {
        LOGE("ERROR: PlatformLibrary native registration failed\n");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

（3）Service 的实现代码

这里的 Service 属于 Framework 层，实现文件是 LedService.java，保存在如下所示的目录：

```
frameworks\base\service\java\com\mokoid\server
```

LedService.java 的具体实现代码如下所示：

```
package com.mokoid.server;

import android.util.Config;
import android.util.Log;
import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.RemoteException;
import android.os.IBinder;
import mokoid.hardware.ILedService;

public final class LedService extends ILedService.Stub {

    static {
        System.load("/system/lib/libmokoid_runtime.so");//加载 JNI 动态库
    }

    public LedService() {
```

```
        Log.i("LedService", "Go to get LED Stub...");
    _init();
    }

    /*
     * Mokoid LED 本地方法
     */
    public boolean setOn(int led) {
        Log.i("MokoidPlatform", "LED On");
    return _set_on(led);
    }
    public boolean setOff(int led) {
        Log.i("MokoidPlatform", "LED Off");
    return _set_off(led);
    }
    private static native boolean _init();//声明 JNI 库可以提供的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
}
```

（4）编写测试应用程序

测试应用程序属于 APP 层，文件“apps/LedClient/src/com/mokoid/LedClient /LedClient.java”的实现代码如下所示：

```
package com.mokoid.LedClient;
import com.mokoid.server.LedService;//导入 Framework 层的 LedService

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class LedClient extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Call an API on the library.
    LedService ls = new LedService();//实例化 LedService
    ls.setOn(1);//通过 LedService 提供的方法控制底层硬件

        TextView tv = new TextView(this);
        tv.setText("LED 0 is on.");
        setContentView(tv);
    }
}
```

4.6.3　通过 Manager 调用 Service 的实现代码

（1）实现 Manager

App 通过此 Manager 和 Service 实现通信功能，实现文件“frameworks/base/core/java/mokoid/hardware/LedManager.java”的实现代码如下所示：

```
package mokoid.hardware;

import android.content.Context;
import android.os.Binder;
import android.os.Bundle;
import android.os.Parcelable;
import android.os.ParcelFileDescriptor;
import android.os.Process;
import android.os.RemoteException;
import android.os.Handler;
import android.os.Message;
import android.os.ServiceManager;
import android.util.Log;
import mokoid.hardware.ILedService;

public class LedManager
{
```

```
    private static final String TAG = "LedManager";
    private ILedService mLedService;

    public LedManager() {

        mLedService = ILedService.Stub.asInterface(
                        ServiceManager.getService("led"));

    if (mLedService != null) {
            Log.i(TAG, "The LedManager object is ready.");
    }
    }

    public boolean LedOn(int n) {
        boolean result = false;

        try {
            result = mLedService.setOn(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOn:", e);
        }
        return result;
    }

    public boolean LedOff(int n) {
        boolean result = false;

        try {
            result = mLedService.setOff(n);
        } catch (RemoteException e) {
            Log.e(TAG, "RemoteException in LedManager.LedOff:", e);
        }
        return result;
    }
}
```

因为 LedService 和 LedManager 分别属于不同的进程，所以，在此需要考虑不同进程之间的通信问题。此时在 Manager 中可以增加一个 aidl 文件来描述通信接口，文件“frameworks/base/core/java/mokoid/hardware/ILedService.aidl”的实现代码如下所示：

```
package mokoid.hardware;

interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
}
```

（2）SystemServer 的实现代码

SystemServer 属于 App 层，文件“apps/LedTest/src/com /mokoid/LedTest/LedSystemServer.java”的主要实现代码如下所示：

```
package com.mokoid.LedTest;

import com.mokoid.server.LedService;

import android.os.IBinder;
import android.os.ServiceManager;
import android.util.Log;
import android.app.Service;
import android.content.Context;
import android.content.Intent;

public class LedSystemServer extends Service {
//代表一个后台进程
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    public void onStart(Intent intent, int startId) {
```

```
        Log.i("LedSystemServer", "Start LedService...");
    /* Please also see SystemServer.java for your interests. */
    LedService ls = new LedService();
        try {
            ServiceManager.addService("led", ls);
        } catch (RuntimeException e) {
            Log.e("LedSystemServer", "Start LedService failed.");
        }
    }
}
```

（3）App 测试程序

此处的测试程序属于 App 层，文件“mokoid-read-only/apps/LedTest/src/com /mokoid/LedTest/LedTest.java”的实现代码如下所示：

```
package com.mokoid.LedTest;
import mokoid.hardware.LedManager;
import com.mokoid.server.LedService;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
import android.widget.Button;
import android.content.Intent;
import android.view.View;

public class LedTest extends Activity implements View.OnClickListener {
    private LedManager mLedManager = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        startService(new Intent("com.mokoid.systemserver"));
        Button btn = new Button(this);
        btn.setText("Click to turn LED 1 On");
    btn.setOnClickListener(this);
        setContentView(btn);
    }
    public void onClick(View v) {
        if (mLedManager == null) {
        Log.i("LedTest", "Creat a new LedManager object.");
        mLedManager = new LedManager();
        }
        if (mLedManager != null) {
        Log.i("LedTest", "Got LedManager object.");
    }
        mLedManager.LedOn(1);
        TextView tv = new TextView(this);
        tv.setText("LED 1 is On.");
        setContentView(tv);
    }
}
```

4.7 HAL 和系统移植

Android 的硬件抽象层和系统移植密切相关，在本节的内容中，将详细讲解移植 Android 5.0 HAL 的基本知识，为读者步入本书后面知识的学习打下基础。

4.7.1　移植各个 Android 部件的方式

在 Android 系统中，不同子系统的移植方法不同。不同部件的移植方式如下所示。

- 显示系统：使用 Framebuffer 标准或其他驱动程序，对应的硬件抽象层是 Gralloc。
- 用户输入系统：使用 Event 设备的驱动程序，对应的硬件抽象层是 EventHub。
- 3D 加速系统：使用非标准的驱动程序，对应的硬件抽象层是 OpenGL。

- 音频系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。
- 视频输出系统：使用非标准的驱动程序，对应的硬件抽象层是 overlay 模块。
- 摄像头系统：使用非标准的驱动程序，对应的是 C++继承的硬件抽象层。
- 多媒体解码系统：使用非标准的驱动程序，对应的硬件抽象层是 Skia 和 OpenMax 插件。
- 电话系统：使用非标准的驱动程序，对应的硬件抽象层是动态开发插件库。
- GPS 定位系统：使用非标准的驱动程序，对应的硬件抽象层通常是直接接口。
- 无线局域网：使用 Wlan 驱动程序，对应的硬件抽象层分别是 Linux 下的 Wpa 和 Android 下的 WiFi。
- 蓝牙系统：使用 Bluetooth 驱动程序，对应的硬件抽象层分别是 Linux 下的 Bluez 和 Android 下的 Bluedroid。
- 传感器系统：使用非标准的驱动程序，对应的硬件抽象层是 Sensor 硬件模块。
- 振动器系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。
- 背光和指示灯系统：使用非标准的驱动程序，对应的硬件抽象层是 Light 硬件模块。
- 警告器系统：使用 Misc 驱动程序，对应的硬件抽象层是 Android 标准的 JNI 层。
- 电池管理系统：使用 Sys 文件系统中固定位置的驱动程序，对应的硬件抽象层是 Android 标准的直接接口。

4.7.2 设置设备权限

当 Android 系统启动时，在内核引导参数上一般都会设置“init=/init”，此时如果内核成功挂载了这个文件系统之后，首先运行的就是这个根目录下的 init 程序。这个 init 程序是 Android 系统运行后的第一个用户空间的程序，它以守护进程的方式运行。

当我们需要增加驱动程序的设备节点时，需要随之更改这些设备节点的属性，这些更改内容被保存在文件“system/core/init/devices.c”中。此文件代码比较冗长，接下来将只对和权限有关的代码进行讲解。

- 定义 perms_表示设备的类型，具体代码如下所示：

```
struct perms_ {
    char *name;
    mode_t perm;
    unsigned int uid;
    unsigned int gid;
    unsigned short prefix;
};
```

- 定义数组 devperms 表示系统中的设备，具体代码如下所示：

```
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 },

        /* logger should be world writable (for logging) but not readable */
    { "/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 },
    { "/dev/ttyMSM0",0600,AID_BLUETOOTH,AID_BLUETOOTH,0 },
    { "/dev/ttyHS0",0600,AID_BLUETOOTH,AID_BLUETOOTH,0 },
    { "/dev/uinput",0600,AID_BLUETOOTH,AID_BLUETOOTH,0 },
    { "/dev/alarm",0664,AID_SYSTEM,AID_RADIO,0 },
```

```
    { "/dev/tty0",0660,AID_ROOT,AID_SYSTEM,0 },
    { "/dev/graphics/",0660,AID_ROOT,AID_GRAPHICS,1 },
    { "/dev/hw3d",0660,AID_SYSTEM,AID_GRAPHICS,0 },
    { "/dev/input/",0660,AID_ROOT,AID_INPUT,1 },
    { "/dev/eac",0660,AID_ROOT,AID_AUDIO,0 },
    { "/dev/cam",0660,AID_ROOT,AID_CAMERA,0 },
    { "/dev/pmem",0660,AID_SYSTEM,AID_GRAPHICS,0 },
    { "/dev/pmem_gpu",0660,AID_SYSTEM,AID_GRAPHICS,1 },
    { "/dev/pmem_adsp",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/pmem_camera",0660,AID_SYSTEM,AID_CAMERA,1 },
    { "/dev/oncrpc/",0660,AID_ROOT,AID_SYSTEM,1 },
    { "/dev/adsp/",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/mt9t013",0660,AID_SYSTEM,AID_SYSTEM,0 },
    { "/dev/akm8976_daemon",0640,AID_COMPASS,AID_SYSTEM,0 },
    { "/dev/akm8976_aot",0640,AID_COMPASS,AID_SYSTEM,0 },
    { "/dev/akm8976_pffd",0640,AID_COMPASS,AID_SYSTEM,0 },
    { "/dev/msm_pcm_out",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/msm_pcm_in",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/msm_pcm_ctl",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/msm_snd",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/msm_mp3",0660,AID_SYSTEM,AID_AUDIO,1 },
    { "/dev/msm_audpre",0660,AID_SYSTEM,AID_AUDIO,0 },
    { "/dev/htc-acoustic",0660,AID_SYSTEM,AID_AUDIO,0 },
    { "/dev/smd0",0640,AID_RADIO,AID_RADIO,0 },
    { "/dev/qmi",0640,AID_RADIO,AID_RADIO,0 },
    { "/dev/qmi0",0640,AID_RADIO,AID_RADIO,0 },
    { "/dev/qmi1",0640,AID_RADIO,AID_RADIO,0 },
    { "/dev/qmi2",0640,AID_RADIO,AID_RADIO,0 },
    { NULL, 0, 0, 0, 0 },
};
```

在上述数组中分别设置了设备的权限、所属用户和所属组，各个权限值的含义和 Linux 中的完全一致。3 个数组分别表示所属用户、所属组和其他人的权限，其中 4 表示可读，2 表示可写，1 表示可执行。例如数组内的首行代码如下所示：

```
{ "/dev/null", 0666, AID_ROOT, AID_ROOT, 0 },
```

“/dev/null”是一个标准的设备，其权限是 0666，表示任何用户可以对其进行读写操作。如果需要增加一个新的设备节点文件，需要在数组 devperms 中新增加一行内容。

- 两个函数

在文件中有两个比较重要的函数：handle_device_event()和 make_device()，具体代码如下所示：

```
static void handle_device_event(struct uevent *uevent)
{
    ...
    /* are we block or char? where should we live? */
    if(!strncmp(uevent->path, "/block", 6)) {
        block = 1;
        base = "/dev/block/";//根据 uevent 路径改变该节点路径
        mkdir(base, 0755);
    } else {
        block = 0;
            /* this should probably be configurable somehow */
        if(!strncmp(uevent->path, "/class/graphics/", 16)) {
            base = "/dev/graphics/";//根据 uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if (!strncmp(uevent->path, "/class/oncrpc/", 14)) {
            base = "/dev/oncrpc/";
            mkdir(base, 0755);
        } else if (!strncmp(uevent->path, "/class/adsp/", 12)) {
            base = "/dev/adsp/";
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/input/", 13)) {
            base = "/dev/input/";//根据 uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/sensors/", 15)) {
            base = "/dev/sensors/";
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/mtd/", 11)) {
```

```
            base = "/dev/mtd/";//根据 uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
        } else if(!strncmp(uevent->path, "/class/misc/", 12) &&
                  !strncmp(name, "log_", 4)) {
            base = "/dev/log/";//根据 uevent 路径改变该 uevent 需要创建节点的路径
            mkdir(base, 0755);
            name += 4;
        } else if(!strncmp(uevent->path, "/class/sound/", 13)) {
            base = "/dev/snd/";
            mkdir(base, 0755);
        } else
            base = "/dev/";
    }

    snprintf(devpath, sizeof(devpath), "%s%s", base, name);

    if(!strcmp(uevent->action, "add")) {
        make_device(devpath, block, uevent->major, uevent->minor);//创建节点文件 devpath
        return;
    }
    ...
}
static void make_device(const char *path, int block, int major, int minor)
{
    unsigned uid;
    unsigned gid;
    mode_t mode;
    dev_t dev;
    if(major > 255 || minor > 255)
        return;
    mode = get_device_perm(path, &uid, &gid) | (block ? S_IFBLK : S_IFCHR);//获取将要创
    //建的节点是否需要重设它的 mode 数值
    dev = (major << 8) | minor;
    mknod(path, mode, dev);
    chown(path, uid, gid);
}
```

函数 get_device_perm()的功能是验证路径 path 是否和 devperms[]数组中的 inode 路径相同，如果相同则返回 devperms[]数组中指定的 uid、gid 和 mode 数值。这样 make_device 会向“/dev”创建 inode 节点，并同时改变该 inode 的 uid 和 gid.[luther.gliethtt]。

- 和用户名相关的名称

在文件“system/core/include/private/android_filesystem_config.h”中定义了和用户名相关的名称，其中 android_id_info 表示用户名 id 的属性，android_id_info 的定义代码如下所示：

```
struct android_id_info {
    const char *name;
    unsigned aid;
};
```

各个用户名 id 被定义在数组 android_ids[]中，此数组表示了一个映射关系，能够将字符串和整数值对应起来，android_ids[]的定义代码如下所示：

```
static struct android_id_info android_ids[] = {
    { "root",      AID_ROOT, },
    { "system",    AID_SYSTEM, },
    { "radio",     AID_RADIO, },
    { "bluetooth", AID_BLUETOOTH, },
    { "graphics",  AID_GRAPHICS, },
    { "input",     AID_INPUT, },
    { "audio",     AID_AUDIO, },
    { "camera",    AID_CAMERA, },
    { "log",       AID_LOG, },
    { "compass",   AID_COMPASS, },
    { "mount",     AID_MOUNT, },
    { "wifi",      AID_WIFI, },
    { "dhcp",      AID_DHCP, },
    { "adb",       AID_ADB, },
    { "install",   AID_INSTALL, },
    { "media",     AID_MEDIA, },
```

```
    { "shell",     AID_SHELL, },
    { "cache",     AID_CACHE, },
    { "diag",      AID_DIAG, },
    { "net_bt_admin", AID_NET_BT_ADMIN, },
    { "net_bt",    AID_NET_BT, },
    { "inet",      AID_INET, },
    { "net_raw",   AID_NET_RAW, },
    { "misc",      AID_MISC, },
    { "nobody",    AID_NOBODY, },
};
```

4.7.3　init.rc 初始化

文件“system/core/rootdir/init.rc”可以实现一些简单的初始化操作，Android 5.0 中的启动脚本文件是 init.rc。init.rc 脚本被直接安装到目标系统的根目录下，并被 init 可执行的程序解析。

在 Android 5.0 中，init.rc 是在 init 启动后被执行的启动脚本，主要包含如下所示的内容。

- Commands：命令。
- Actions：动作。
- Triggers：触发条件。
- Services：服务。
- Options：选项。
- Propertise：属性。

4.7.4　文件系统的属性

在文件“system/core/include/private/android_filesystem_config.h”中定义了各个文件的属性，其中 fs_path_config 表示文件系统路径的属性，具体定义代码如下所示：

```
struct fs_path_config {
    unsigned mode;//模式
    unsigned uid;//用户 id
    unsigned gid;//组 id
    const char *prefix;//目录前缀
};
```

在数组 android_dirs[]中定义了子目录的属性，定义代码如下所示：

```
static struct fs_path_config android_dirs[] = {
    { 00770, AID_SYSTEM, AID_CACHE,  "cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app-private" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/dalvik-cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/data" },
    { 00771, AID_SHELL,  AID_SHELL,  "data/local/tmp" },
    { 00771, AID_SHELL,  AID_SHELL,  "data/local" },
    { 01771, AID_SYSTEM, AID_MISC,   "data/misc" },
    { 00770, AID_DHCP,   AID_DHCP,   "data/misc/dhcp" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data" },
    { 00750, AID_ROOT,   AID_SHELL,  "sbin" },
    { 00755, AID_ROOT,   AID_SHELL,  "system/bin" },
    { 00755, AID_ROOT,   AID_SHELL,  "system/xbin" },
    { 00777, AID_ROOT,   AID_ROOT,   "system/etc/ppp" }, /* REMOVE */
    { 00777, AID_ROOT,   AID_ROOT,   "sdcard" },
    { 00755, AID_ROOT,   AID_ROOT,   0 },
};
```

然后在数组 android_files[]中定义了默认文件的属性，定义代码如下所示：

```
static struct fs_path_config android_files[] = {
    { 00555, AID_ROOT,      AID_ROOT,      "system/etc/ppp/ip-up" },
    { 00555, AID_ROOT,      AID_ROOT,      "system/etc/ppp/ip-down" },
    { 00440, AID_ROOT,      AID_SHELL,     "system/etc/init.goldfish.rc" },
    { 00550, AID_ROOT,      AID_SHELL,     "system/etc/init.goldfish.sh" },
    { 00440, AID_ROOT,      AID_SHELL,     "system/etc/init.trout.rc" },
    { 00550, AID_ROOT,      AID_SHELL,     "system/etc/init.ril" },
    { 00550, AID_ROOT,      AID_SHELL,     "system/etc/init.testmenu" },
```

```
    { 00550, AID_ROOT,      AID_SHELL,     "system/etc/init.gprs-pppd" },
    { 00550, AID_DHCP,      AID_SHELL,     "system/etc/dhcpcd/dhcpcd-run-hooks" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/dbus.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/hcid.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/input.conf" },
    { 00440, AID_BLUETOOTH, AID_BLUETOOTH, "system/etc/bluez/audio.conf" },
    { 00440, AID_RADIO,     AID_AUDIO,     "/system/etc/AudioPara4.csv" },
    { 00644, AID_SYSTEM,    AID_SYSTEM,    "data/app/*" },
    { 00644, AID_SYSTEM,    AID_SYSTEM,    "data/app-private/*" },
    { 00644, AID_APP,       AID_APP,       "data/data/*" },
       /* the following two files are INTENTIONALLY set-gid and not set-uid.
        * Do not change. */
    { 02755, AID_ROOT,      AID_NET_RAW,   "system/bin/ping" },
    { 02755, AID_ROOT,      AID_INET,      "system/bin/netcfg" },
       /* the following four files are INTENTIONALLY set-uid, but they
     * are NOT included on user builds. */
    { 06755, AID_ROOT,      AID_ROOT,      "system/xbin/su" },
    { 06755, AID_ROOT,      AID_ROOT,      "system/xbin/librank" },
    { 06755, AID_ROOT,      AID_ROOT,      "system/xbin/procrank" },
    { 06755, AID_ROOT,      AID_ROOT,      "system/xbin/procmem" },
    { 00755, AID_ROOT,      AID_SHELL,     "system/bin/*" },
    { 00755, AID_ROOT,      AID_SHELL,     "system/xbin/*" },
    { 00750, AID_ROOT,      AID_SHELL,     "sbin/*" },
    { 00755, AID_ROOT,      AID_ROOT,      "bin/*" },
    { 00750, AID_ROOT,      AID_SHELL,     "init*" },
    { 00644, AID_ROOT,      AID_ROOT,      0 },
};
```

第5章　分析IPC通信机制

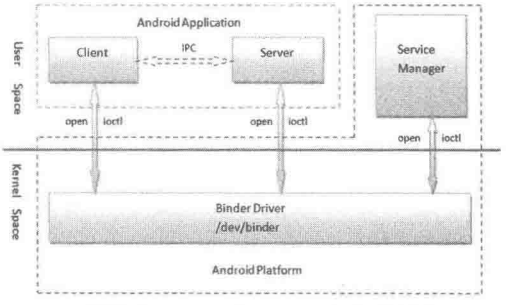
在Android系统中，应用程序都是由Activity和Service组成的。Service通常运行在独立的进程中，而Activity既可能运行在同一个进程中，也可能运行在不同的进程中。问题是：不在同一个进程中的Activity或Service是如何实现通信功能的呢？答案是Binder进程间通信机制。在本章的内容中，将详细分析Android的进程通信机制Binder的实现源代码。

5.1 Binder机制概述

Binder是Android系统提供的一种IPC（进程间通信）机制。由于Android是基于Linux内核的，因此，除了Binder以外，还存在其他的IPC机制，例如，管道和Socket等。Binder相对于其他IPC机制来说，更加灵活和方便。Binder的驱动代码在kernel/drivers/staing/android/binder.c中，另外，该目录下还有一个binder.h头文件。Binder是一个虚拟设备，所以，它的代码相比而言还算简单，读者只要有基本的Linux驱动开发方面的知识就能读懂它。/proc/binder目录下的内容可用来查看Binder设备的运行状况。

对于初学Android的读者来说，最难掌握的可能就是Binder机制了，因为，Android系统基本上可以看作是一个基于Binder通信的C/S架构。Binder就像网络一样，把系统的各个部分连接在了一起，因此，它是非常重要的。在基于Binder通信的C/S架构体系中，除了C/S架构所包括的Client端和Server端外，Android还有一个全局的ServiceManager端，它的作用是管理系统中的各种服务（Service）。

在Android系统的Binder机制中，由4个组件Client、Server、Service Manager组成，如图5-1所示。



▲图5-1　Binder机制中的组件关系图

在Android系统中，Client、Server和ServiceManager三者之间的交互关系如下所示。

- Client、Server和Service Manager在用户空间中实现，Binder驱动程序在内核空间中实现。
- Server进程要先注册一些Service到ServiceManager中，所以，Server是ServiceManager的客户端，而ServiceManager就是服务端了。Service Manager是一个守护进程，能够管理Server，并向Client提供查询Server接口。
- 如果某个Client进程要使用某个Service，必须先到ServiceManager中获取该Service的相关信息，所以，Client是ServiceManager的客户端。另外，Client根据得到的Service信息与Service所在的Server进程建立通信的通路，然后就可以直接与Service交互了，所以，Client也是Server的客户端。
- Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和Service Manager通过open和ioctl文件操作函数与Binder驱动程序进行通信。
- 在Android平台中已经实现了Binder驱动程序和Service Manager，开发者只需要在用户空间实现自己的Client和Server即可。
- 三者的交互都是基于Binder通信的，所以，通过任意两者之间的关系，都可以揭示Binder

的奥秘。

在此需要重点强调 Binder 通信与 C/S 架构之间的关系，Binder 只是为 C/S 架构提供的一种通信的方式，其实完全可以采用其他 IPC 方式进行通信，例如，在系统中有很多其他的程序就是采用 Socket 或 Pipe 方法进行进程间通信的。很多初学者可能会觉得 Binder 比较复杂，尤其是看到诸如 BpXXX、BnXXX 之类的定义便感到头晕，这很有可能是把 Binder 通信层结构和应用的业务层结构搞混了。如果能搞清楚这二者的关系，完全可以自己实现一个不使用 BpXXX 和 BnXXX 的 Service，因为 ServiceManager 并没有使用它们。

5.2 分析 Binder 驱动程序

Binder 采用 AIDL（android interface description language）来描述进程间通信的接口。Binder 作为一个特殊的字符设备，其设备节点是/dev/binder。在三星和高通产品的驱动程序中，Binder 驱动通常在如下文件中实现：

```
kernel/drivers/staging/binder.h
kernel/drivers/staging/binder.c
```

在本节的内容中，将详细分析上述文件的实现源代码。

5.2.1 分析数据结构

在 Binder 驱动程序中，主要包含了如下所示的数据结构。

（1）binder_work

binder_work 表示在 Binder 驱动中进程所要处理的工作项，定义代码如下所示：

```
struct binder_work {
   struct list_head entry;
   enum {
      BINDER_WORK_TRANSACTION = 1,
      BINDER_WORK_TRANSACTION_COMPLETE,
      BINDER_WORK_NODE,
      BINDER_WORK_DEAD_BINDER,
      BINDER_WORK_DEAD_BINDER_AND_CLEAR,
      BINDER_WORK_CLEAR_DEATH_NOTIFICATION,
   } type;
};
```

在上述结构体定义中，entry 被定义为 list_head 类型，用于实现一个双向链表，能够存储所有 binder_work 的队列；并且还包含了一个 enum 类型的 type：binder_work。

（2）binder_node

结构体 binder_node 用来定义 Binder 实体对象。在 Android 系统中，每一个 Service 组件在 Binder 驱动程序中都有一个 Binder 实体对象。定义 binder_node 的代码如下所示：

```
struct binder_node {
      int debug_id;
      struct binder_work work;
      union {
            struct rb_node rb_node;
            struct hlist_node dead_node;
      };
      struct binder_proc *proc;
      struct hlist_head refs;
      int internal_strong_refs;
      int local_weak_refs;
      int local_strong_refs;
      void __user *ptr;
      void __user *cookie;
      unsigned has_strong_ref:1;
      unsigned pending_strong_ref:1;
      unsigned has_weak_ref:1;
      unsigned pending_weak_ref:1;
      unsigned has_async_transaction:1;
```

```
        unsigned accept_fds:1;
        unsigned min_priority:8;
        struct list_head async_todo;
};
```

驱动中的 Binder 实体也叫做“节点”，隶属于提供实体的进程。结构体 binder_node 中各个成员的具体说明如表 5-1 所示。

表 5-1　　结构体 binder_node 中的成员说明信息

成　员	含　义
int debug_id	用于调试
struct binder_work work	当本节点引用计数发生改变，需要通知所属进程时，通过该成员挂入所属进程的 to-do 队列里，唤醒所属进程执行 Binder 实体引用计数的修改

union {
struct rb_node rb_node;
struct hlist_node dead_node;
}
	每个进程都维护一棵红黑树，以 Binder 绑定在用户空间的指针，即本结构的 ptr 成员为索引存放该进程所有的 Binder 实体。这样驱动可以根据 Binder 实体在用户空间的指针很快找到其位于内核的节点。rb_node 用于将本节点链入该红黑树中。
	销毁节点时须将 rb_node 从红黑树中摘除，但如果本节点还有引用没有切断，就用 dead_node 将节点隔离到另一个链表中，直到通知所有进程切断与该节点的引用后，该节点才可能被销毁

struct binder_proc *proc	本成员指向节点所属的进程，即提供该节点的进程
struct hlist_head refs	本成员是队列头，所有指向本节点的引用都链接在该队列里。这些引用可能隶属于不同的进程。通过该队列可以遍历指向该节点的所有引用
int internal_strong_refs	用以实现强指针的计数器：产生一个指向本节点的强引用该计数就会加 1
int local_weak_refs	驱动为传输中的 Binder 设置的弱引用计数。如果一个 Binder 打包在数据包中从一个进程发送到另一个进程，驱动会为该 Binder 增加引用计数，直到接收进程通过 BC_FREE_BUFFER 通知驱动释放该数据包的数据区为止
int local_strong_refs	驱动为传输中的 Binder 设置的强引用计数。同上
void __user *ptr	指向用户空间 Binder 实体的指针，来自于 flat_binder_object 的 binder 成员
void __user *cookie	指向用户空间的附加指针，来自于 flat_binder_object 的 cookie 成员
unsigned has_strong_ref unsigned pending_strong_ref unsigned has_weak_ref unsigned pending_weak_ref	这一组标志用于控制驱动与 Binder 实体所在进程交互式修改引用计数
unsigned has_async_transaction	该成员表明该节点在 to-do 队列中有异步交互尚未完成。驱动将所有发送往接收端的数据包暂存在接收进程或线程开辟的 to-do 队列里。对于异 步交互，驱动做了适当流控：如果 to-do 队列里有异步交互尚待处理则该成员置 1，这将导致新到的异步交互存放在本结构成员– asynch_todo 队列中，而不直接送到 to-do 队列里。目的是为同步交互让路，避免长时间阻塞发送端
unsigned accept_fds	表明节点是否同意接收文件方式的 Binder，来自 flat_binder_object 中 flags 成员的 FLAT_BINDER_FLAG_ACCEPTS_FDS 位。由于接收文件 Binder 会为进程自动打开一个文件，占用有限的文件描述符，节点可以设置该位拒绝这种行为
int min_priority	设置处理 Binder 请求的线程的最低优先级。发送线程将数据提交给接收线程处理时，驱动会将发送线程的优先级也赋予接收线程，使得数据即使跨了进程也能以同样优先级得到处理。不过如果发送线程优先级过低，接收线程将以预设的最小值运行。 该域的值来自于 flat_binder_object 中 flags 成员。
struct list_head async_todo	异步交互等待队列，用于分流发往本节点的异步交互包

（3）binder_ref

结构体 binder_ref 用来描述一个 Binder 引用对象，在 Android 系统中，每一个 Client 组件在 Binder 驱动程序中都有一个 Binder 引用对象。定义 binder_ref 的代码如下所示：

```
struct binder_ref {
        /* Lookups needed: */
        /*   node + proc => ref (transaction) */
        /*   desc + proc => ref (transaction, inc/dec ref) */
```

```
        /*   node => refs + procs (proc exit) */
        int debug_id;
        struct rb_node rb_node_desc;
        struct rb_node rb_node_node;
        struct hlist_node node_entry;
        struct binder_proc *proc;
        struct binder_node *node;
        uint32_t desc;
        int strong;
        int weak;
        struct binder_ref_death *death;
};
```

结构体 binder_ref 中各个成员的具体说明如表 5-2 所示。

表 5-2　　结构体 binder_ref 中的成员说明信息

成　员	含　义
int debug_id;	调试用
struct rb_node rb_node_desc;	每个进程有一棵红黑树，进程所有引用以引用号（即本结构的 desc 域）为索引添入该树中。本成员用作链接到该树的一个节点
struct rb_node rb_node_node;	每个进程又有一棵红黑树，进程所有引用以节点实体在驱动中的内存地址（即本结构的 node 域）为所引添入该树中。本成员用作链接到该树的一个节点
struct hlist_node node_entry;	该域将本引用作为节点链入所指向的 Binder 实体结构 binder_node 中的 refs 队列
struct binder_proc *proc;	本引用所属的进程
struct binder_node *node;	本引用所指向的节点（Binder 实体）
uint32_t desc;	本结构的引用号
int strong;	强引用计数
int weak;	弱引用计数
struct binder_ref_death *death;	应用程序向驱动发送 BC_REQUEST_DEATH_NOTIFICATION 或 BC_CLEAR_DEATH_NOTIFICATION 命令从而当 Binder 实体销毁时能够收到来自驱动的提醒。该域不为空，表明用户订阅了对应实体销毁的“信息”

（4）binder_ref_death

binder_ref_death 是一个通知结构体，只要某进程对某 binder 引用订阅了其实体的死亡通知，那么 binder 驱动将会为该 binder 引用建立一个 binder_ref_death 通知结构体，将其保存在当前进程的对应 binder 引用结构体的 death 域中。定义 binder_ref_death 的代码如下所示：

```
struct binder_ref_death {
        struct binder_work work;
        void __user *cookie;
};
```

（5）binder_buffer

结构体 binder_buffer 用来描述一个内核缓冲区，能够在进程之间传输数据。定义 binder_buffer 的代码如下所示：

```
struct binder_buffer {
        struct list_head entry;
        struct rb_node rb_node;
                        /* by address */
        unsigned free:1;
        unsigned allow_user_free:1;
        unsigned async_transaction:1;
        unsigned debug_id:29;

        struct binder_transaction *transaction;

        struct binder_node *target_node;
        size_t data_size;
        size_t offsets_size;
```

```
        uint8_t data[0];
};
```

结构体 binder_buffer 能够储存 Binder 的相关信息，成员的具体说明如下所示。

- entry：构建一个双向链表。
- rb_node：表示一个红黑树节点。
- transaction：用于中转请求和返回结果。
- target_node：是一个目标节点。
- data_size：表示数据的大小。
- offsets_size：是一个偏移量。
- data[0]：用于存储实际数据。

（6）binder_proc

结构体 binder_proc 表示正在使用 Binder 进程通信机制的进程，能够保存调用 Binder 的各个进程或线程的信息，例如线程 ID、进程 ID、Binder 状态信息等。定义 binder_proc 的具体实现代码如下所示：

```
struct binder_proc {
    //实现双向链表
    struct hlist_node proc_node;
    //线程队列、双向链表、所有的线程信息
    struct rb_root threads;
    struct rb_root nodes;
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    //进程 ID
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;

    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;

    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo;
    //等待队列
    wait_queue_head_t wait;
    //Binder 状态
    struct binder_stats stats;
    struct list_head delivered_death;
    //最大线程
    int max_threads;
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    //默认优先级
    long default_priority;
};
```

在上述代码中，成员 proc_node 用于实现双向链表，成员 threads 用于储存所有的线程信息。

（7）binder_thread

结构体 binder_thread 用于存储每一个单独的线程的信息，表示 Binder 线程池中的一个线程。定义 binder_thread 的具体实现代码如下所示：

```
struct binder_thread {
        struct binder_proc *proc;
        struct rb_node rb_node;
```

```
        int pid;
        int looper;
        struct binder_transaction *transaction_stack;
        struct list_head todo;
        uint32_t return_error;
        uint32_t return_error2;
        wait_queue_head_t wait;
        struct binder_stats stats;
};
```

各个成员的具体说明如下所示。

- proc：表示当前线程属于哪一个 Binder 进程（binder_proc 指针）。
- rb_node：是一个红黑树节点。
- pid：表示线程的 pid。
- looper：表示线程的状态信息。
- transaction_stack：定义了要接收和发送的进程和线程信息，其结构体为 binder_transaction。
- todo：用于创建一个双向链表。
- return_error 和 return_error2：表示返回的错误信息代码。
- wait：是一个等待队列头结构，具体的定义代码如下所示。

```
struct binder_stats {
        int br[_IOC_NR(BR_FAILED_REPLY) + 1];
        int bc[_IOC_NR(BC_DEAD_BINDER_DONE) + 1];
        int obj_created[BINDER_STAT_COUNT];
        int obj_deleted[BINDER_STAT_COUNT];
};
```

各个成员的具体说明如下所示。

- br：用来存储 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Return Protocol）。
- bc：存储着 BINDER_WRITE_READ 的写操作命令协议（Binder Driver Command Protocol）。
- obj_created：保存 BINDER_STAT_COUNT 的对象计数，当创建一个对象时需要同时调用该成员来增加相应的对象计数，而 obj_deleted 则正好与之相反。

looper 表示的线程状态信息在如下枚举中定义：

```
enum {
        BINDER_LOOPER_STATE_REGISTERED  = 0x01,
        BINDER_LOOPER_STATE_ENTERED     = 0x02,
        BINDER_LOOPER_STATE_EXITED      = 0x04,
        BINDER_LOOPER_STATE_INVALID     = 0x08,
        BINDER_LOOPER_STATE_WAITING     = 0x10,
        BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

上述枚举主要包括的状态信息有：注册、进入、退出、销毁、等待、需要返回。

（8）binder_transaction

结构体 binder_transaction 的功能是中转请求和返回结果，并保存接收和要发送的进程信息。定义结构体 binder_transaction 的具体实现代码如下所示：

```
struct binder_transaction {
        int debug_id;//调试相关
        struct binder_work work;
        struct binder_thread *from;
        struct binder_transaction *from_parent;
        struct binder_proc *to_proc;
        struct binder_thread *to_thread;
        struct binder_transaction *to_parent;
        unsigned need_reply : 1;
        struct binder_buffer *buffer;
        unsigned int    code;
        unsigned int    flags;
        long priority;
        long saved_priority;
        uid_t   sender_euid;
};
```

上述成员的具体说明如下所示。

- work：是一个 binder_work。
- from 和 to_thread：都是一个 binder_thread 对象，用于表示接收和要发送的进程信息。
- from_parent 和 to_thread：接收和发送进程信息的父节点。
- to_proc：是一个 binder_proc 类型的结构体，还包括 flags、need_reply、优先级（priority）等数据。
- sender_euid：Linux 系统中的每个进程都有两个 ID：用户 ID 和有效用户 ID，UID 一般表示进程的创建者（属于哪个用户创建），EUID 表示进程对于文件和资源的访问权限。sender_euid 表示要发送进程对文件和资源的操作权限。

另外，在结构体 binder_transaction 中，还包含了类型类 inder_buffer 的一个 buffer，用来表示 Binder 的缓冲区信息。inder_buffer 在前面已经进行了讲解。

（9）binder_write_read

结构体 binder_write_read 功能是表示在进程之间的通信过程中传输的数据，数据包中有一个 cmd 域用于区分不同的请求。定义结构体 binder_write_read 的实现代码如下所示：

```
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;
};
```

各个成员的具体说明如下所示。

- write_size 和 read_size：分别表示写入和读取的数据的大小。
- write_consumed 和 read_consumed：分别表示被消耗的写数据和读数据的大小。

当 Binder 驱动找到处理此事件的进程之后，Binder 驱动就会把需要处理的事件的任务放在读缓冲（binder_write_read）里，返回给这个服务线程，该服务线程则会执行指定命令的操作；处理请求的线程把数据交给合适的对象来执行预定操作，然后把返回结果同样用结构 binder_transaction_data 进行封装，以写命令的方式传回给 Binder 驱动，并将此数据放在一个读缓冲（binder_write_read）里，返回给正在等待结果的原进程（线程），这样就完成了一次通信。

（10）BinderDriverCommandProtocol

结构体 binder_write_read 包含的命令在 BinderDriverCommandProtocol 中定义，具体代码如下所示：

```
enum BinderDriverCommandProtocol {
        BC_TRANSACTION = _IOW('c', 0, struct binder_transaction_data),
        BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
        /*
         * binder_transaction_data: the sent command.
         */
        BC_ACQUIRE_RESULT = _IOW('c', 2, int),
        /*
         * not currently supported
         * int:  0 if the last BR_ATTEMPT_ACQUIRE was not successful.
         * Else you have acquired a primary reference on the object.
         */
        BC_FREE_BUFFER = _IOW('c', 3, int),
        /*
         * void *: ptr to transaction data received on a read
         */
        BC_INCREFS = _IOW('c', 4, int),
        BC_ACQUIRE = _IOW('c', 5, int),
        BC_RELEASE = _IOW('c', 6, int),
        BC_DECREFS = _IOW('c', 7, int),
        /*
         * int: descriptor
         */
```

```
        BC_INCREFS_DONE = _IOW('c', 8, struct binder_ptr_cookie),
        BC_ACQUIRE_DONE = _IOW('c', 9, struct binder_ptr_cookie),
        /*
         * void *: ptr to binder
         * void *: cookie for binder
         */
        BC_ATTEMPT_ACQUIRE = _IOW('c', 10, struct binder_pri_desc),
        /*
         * not currently supported
         * int: priority
         * int: descriptor
         */
        BC_REGISTER_LOOPER = _IO('c', 11),
        /*
         * No parameters.
         * Register a spawned looper thread with the device.
         */
        BC_ENTER_LOOPER = _IO('c', 12),
        BC_EXIT_LOOPER = _IO('c', 13),
        /*
         * No parameters.
         * These two commands are sent as an application-level thread
         * enters and exits the binder loop, respectively.  They are
         * used so the binder can have an accurate count of the number
         * of looping threads it has available.
         */
        BC_REQUEST_DEATH_NOTIFICATION = _IOW('c', 14, struct binder_ptr_cookie),
        /*
         * void *: ptr to binder
         * void *: cookie
         */
        BC_CLEAR_DEATH_NOTIFICATION = _IOW('c', 15, struct binder_ptr_cookie),
        /*
         * void *: ptr to binder
         * void *: cookie
         */
        BC_DEAD_BINDER_DONE = _IOW('c', 16, void *),
        /*
         * void *: cookie
         */
};
```

在上述枚举命令成员中，重要的是 BC_TRANSACTION 和 BC_REPLY 命令，被作为发送操作的命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

（11）BinderDriverReturnProtocol

在枚举 BinderDriverReturnProtocol 中定义了读操作命令协议，具体实现代码如下所示：

```
enum BinderDriverReturnProtocol {
        BR_ERROR = _IOR('r', 0, int),
        /*
         * int: error code
         */
        BR_OK = _IO('r', 1),
        /* No parameters! */
        BR_TRANSACTION = _IOR('r', 2, struct binder_transaction_data),
        BR_REPLY = _IOR('r', 3, struct binder_transaction_data),
        /*
         * binder_transaction_data: the received command.
         */
        BR_ACQUIRE_RESULT = _IOR('r', 4, int),
        /*
         * not currently supported
         * int: 0 if the last bcATTEMPT_ACQUIRE was not successful.
         * Else the remote object has acquired a primary reference.
         */
        BR_DEAD_REPLY = _IO('r', 5),
        /*
         * The target of the last transaction (either a bcTRANSACTION or
```

```
         * a bcATTEMPT_ACQUIRE) is no longer with us.  No parameters.
         */
        BR_TRANSACTION_COMPLETE = _IO('r', 6),
        /*
         * No parameters... always refers to the last transaction requested
         * (including replies).  Note that this will be sent even for
         * asynchronous transactions.
         */
        BR_INCREFS = _IOR('r', 7, struct binder_ptr_cookie),
        BR_ACQUIRE = _IOR('r', 8, struct binder_ptr_cookie),
        BR_RELEASE = _IOR('r', 9, struct binder_ptr_cookie),
        BR_DECREFS = _IOR('r', 10, struct binder_ptr_cookie),
        /*
         * void *:     ptr to binder
         * void *: cookie for binder
         */
        BR_ATTEMPT_ACQUIRE = _IOR('r', 11, struct binder_pri_ptr_cookie),
        /*
         * not currently supported
         * int: priority
         * void *: ptr to binder
         * void *: cookie for binder
         */
        BR_NOOP = _IO('r', 12),
        /*
         * No parameters.  Do nothing and examine the next command.  It exists
         * primarily so that we can replace it with a BR_SPAWN_LOOPER command.
         */
        BR_SPAWN_LOOPER = _IO('r', 13),
        /*
         * No parameters.  The driver has determined that a process has no
         * threads waiting to service incomming transactions.  When a process
         * receives this command, it must spawn a new service thread and
         * register it via bcENTER_LOOPER.
         */
        BR_FINISHED = _IO('r', 14),
        /*
         * not currently supported
         * stop threadpool thread
         */
        BR_DEAD_BINDER = _IOR('r', 15, void *),
        /*
         * void *: cookie
         */
        BR_CLEAR_DEATH_NOTIFICATION_DONE = _IOR('r', 16, void *),
        /*
         * void *: cookie
         */
        BR_FAILED_REPLY = _IO('r', 17),
        /*
         * The the last transaction (either a bcTRANSACTION or
         * a bcATTEMPT_ACQUIRE) failed (e.g. out of memory).  No parameters.
         */
};
```

在上述命令中，BC_TRANSACTION 和 BC_REPLY 命令被作为发送操作命令，其数据参数都是 binder_transaction_data 结构体。其中前者用于翻译和解析将要被处理的事件数据，而后者则是事件处理完成之后对返回“结果数据”的操作命令。

（12）binder_ptr_cookie 和 binder_transaction_data

binder_ptr_cookie 和 binder_transaction_data 是两个比较重要的结构体，其中 binder_ptr_cookie 表示一个 Binder 实体对象或 Service 组件的死亡接收通知，具体定义代码如下所示：

```
struct binder_ptr_cookie {
        void *ptr;
        void *cookie;
};
```

而 binder_transaction_data 表示在通信过程中传递的数据，具体定义代码如下所示：

```
struct binder_transaction_data {
        /* The first two are only used for bcTRANSACTION and brTRANSACTION,
         * identifying the target and contents of the transaction.
         */
        union {
                size_t  handle; /* target descriptor of command transaction */
                void    *ptr;   /* target descriptor of return transaction */
        } target;
        void            *cookie;        /* target object cookie */
        unsigned int    code;           /* transaction command */
        /* General information about the transaction. */
        unsigned int    flags;
        pid_t           sender_pid;
        uid_t           sender_euid;
        size_t          data_size;      /* number of bytes of data */
        size_t          offsets_size;   /* number of bytes of offsets */
        /* If this transaction is inline, the data immediately
         * follows here; otherwise, it ends with a pointer to
         * the data buffer.
         */
        union {
                struct {
                        /* transaction data */
                        const void      *buffer;
                        /* offsets from buffer to flat_binder_object structs */
                        const void      *offsets;
                } ptr;
                uint8_t buf[8];
        } data;
};
```

（13）flat_binder_object

在 Android 系统中，把在进程之间传递的数据称为 Binder 对象，即 Binder Object。Binder 对象在对应源代码中使用结构体 flat_binder_object 来表示，具体代码如下所示：

```
struct flat_binder_object {
        /* 8 bytes for large_flat_header. */
        unsigned long           type;
        unsigned long           flags;
        /* 8 bytes of data. */
        union {
                void            *binder;        /* local object */
                signed long     handle;         /* remote object */
        };
        /* extra data associated with local object */
        void                    *cookie;
};
```

各个成员的具体说明如下所示。

- type：描述了 Binder 的类型，传输的数据是一个复用数据联合体。对于 Binder 类型来说，数据是一个 Binder 本地对象。
- handle：是一个远程的 handle 句柄。假如 A 有个对象 O，对于 A 来说，O 就是一个本地的 Binder 对象；如果 B 想访问 A 的 O 对象，对于 B 来说，O 就是一个 handle。所以 handle 和 Binder 都指向 O。
- cookie：如果是本地对象，Binder 还可以带有额外的数据，这些数据将被保存到 cookie 字段中。
- flags：表示传输方式，比如同步和异步等，其值同样使用一个 enum 来表示，具体定义代码如下所示：

```
enum transaction_flags {
        TF_ONE_WAY      = 0x01, /* this is a one-way call: async, no return */
        TF_ROOT_OBJECT  = 0x04, /* contents are the component's root object */
        TF_STATUS_CODE  = 0x08, /* contents are a 32-bit status code */
        TF_ACCEPT_FDS   = 0x10, /* allow replies with file descriptors */
};
```

5.2.2　分析设备初始化

我们可以在文件 binder.c 中找到该初始化函数 binder_init，具体定义代码如下所示：

```
static int __init binder_init(void)
{
        int ret;

        binder_deferred_workqueue = create_singlethread_workqueue("binder");
        if (!binder_deferred_workqueue)
                return -ENOMEM;

        binder_debugfs_dir_entry_root = debugfs_create_dir("binder", NULL);
        if (binder_debugfs_dir_entry_root)
                binder_debugfs_dir_entry_proc = debugfs_create_dir("proc",
                                                 binder_debugfs_dir_entry_root);
        ret = misc_register(&binder_miscdev);
        if (binder_debugfs_dir_entry_root) {
                debugfs_create_file("state",
                                    S_IRUGO,
                                    binder_debugfs_dir_entry_root,
                                    NULL,
                                    &binder_state_fops);
                debugfs_create_file("stats",
                                    S_IRUGO,
                                    binder_debugfs_dir_entry_root,
                                    NULL,
                                    &binder_stats_fops);
                debugfs_create_file("transactions",
                                    S_IRUGO,
                                    binder_debugfs_dir_entry_root,
                                    NULL,
                                    &binder_transactions_fops);
                debugfs_create_file("transaction_log",
                                    S_IRUGO,
                                    binder_debugfs_dir_entry_root,
                                    &binder_transaction_log,
                                    &binder_transaction_log_fops);
                debugfs_create_file("failed_transaction_log",
                                    S_IRUGO,
                                    binder_debugfs_dir_entry_root,
                                    &binder_transaction_log_failed,
                                    &binder_transaction_log_fops);
        }
        return ret;
}
```

binder_init 是 Binder 驱动的初始化函数，在实现时需要调用设备驱动。Android Binder 设备驱动接口函数是 device_initcall，使用 module_init 和 module_exit 是为了同时兼容支持静态编译的驱动模块（buildin）和动态编译的驱动模块（module）。Binder 使用 device_initcall 的目的就是不让 Binder 驱动支持动态编译，而且需要在内核（Kernel）做镜像。initcall 用于注册进行初始化的函数，如果的确需要将 Binder 驱动修改为动态的内核模块，可以直接将 device_initcall 修改为 module_init，并增加 module_exit 的驱动卸载接口函数。

在注册 Binder 驱动为 Misc 设备时，指定了 Binder 驱动的 miscdevice，具体实现代码如下所示：

```
static struct miscdevice binder_miscdev = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "binder",
        .fops = &binder_fops
};
```

Binder 设备的主设备号为 10，此设备号是动态获得的，各个参数的具体说明如下所示。

- MISC_DYNAMIC_MINOR：.minor 被设置为动态获得设备号 MISC_DYNAMIC_MINOR。
- .name：表示设备名称。
- file_operations：指定了该设备的 file_operations 结构体，定义代码如下所示：

```
static struct file_operations binder_fops = {
```

```
    .owner = THIS_MODULE,
    .poll = binder_poll,
    .unlocked_ioctl = binder_ioctl,
    .mmap = binder_mmap,
    .open = binder_open,
    .flush = binder_flush,
    .release = binder_release,
};
```

在 Android 系统中，任何驱动程序都具备向用户空间的程序提供操作接口的功能。这个接口是一个标准接口，Android Binder 驱动提供了操作设备文件（/dev/binder）的接口。正如 binder_fops 所描述的 file_operations 结构体一样，其中主要包括了 binder_poll、binder_ioctl、 binder_mmap、binder_open、binder_flush、binder_release 等标准操作接口。

5.2.3 打开 Binder 设备文件

在 Android 系统的 Binder 机制中，函数 binder_open 的功能是打开 Binder 设备文件/dev/binder。在 Android 系统中，底层驱动的任何一个进程及线程都可以打开一个 Binder 设备，其打开过程的实现代码如下所示：

```
static int binder_open(struct inode *nodp, struct file *filp)
{
        struct binder_proc *proc;
        binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
                     current->group_leader->pid, current->pid);
        proc = kzalloc(sizeof(*proc), GFP_KERNEL);
        if (proc == NULL)
                return -ENOMEM;
        get_task_struct(current);
        proc->tsk = current;
        INIT_LIST_HEAD(&proc->todo);
        init_waitqueue_head(&proc->wait);
        proc->default_priority = task_nice(current);
        binder_lock(__func__);
        binder_stats_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;
        binder_unlock(__func__);
        if (binder_debugfs_dir_entry_proc) {
                char strbuf[11];
                snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
                proc->debugfs_entry = debugfs_create_file(strbuf, S_IRUGO,
                        binder_debugfs_dir_entry_proc, proc, &binder_proc_fops);
        }
        return 0;
}
```

函数 binder_open 的具体实现流程如下所示。

（1）创建并分配一个 binder_proc 空间来保存 Binder 数据。

（2）增加当前线程/进程的引用计数，给 binder_proc 的 tsk 字段赋值。

（3）实现 binder_proc 队列的初始化，主要包括：

- 使用 INIT_LIST_HEAD 初始化链表头 todo；
- 使用 init_waitqueue_head 初始化等待队列 wait；
- 设置默认优先级（default_priority）为当前进程的 nice 值（通过 task_nice 得到当前进程的 nice 值）。

（4）增加 BINDER_STAT_PROC 的对象计数，并通过 hlist_add_head 把创建的 binder_proc 对象添加到全局的 binder_proc 哈希表中，这样任何一个进程就都可以访问到其他进程的 binder_proc 对象。

（5）把当前进程（或线程）的线程组的 pid（pid 指向线程 id）赋值给 proc 的 pid 字段，同时

把创建的 binder_proc 对象指针赋值给 filp 的 private_data 对象并保存起来。

（6）在 binder proc 目录中创建只读文件/proc/binder/proc/$pid，功能是输出当前 binder proc 对象的状态。文件名以 pid 命名，但是该 pid 字段并不是当前进程/线程的 id，而是线程组的 pid，表示是线程组中第一个线程的 pid（因为上面是将 current→group_leader→pid 赋值给该 pid 字段的）。并且在创建该文件时也指定了操作该文件的函数接口为 binder_read_proc_proc，此函数的参数表示创建的 binder_proc 对象 proc。

在文件 frameworks\native\cmds\servicemanager\binder.c 中，对应函数 binder_open 的功能是打开 binder 设备文件并映射到用户空间，具体实现代码如下所示：

```
struct binder_state *binder_open(size_t mapsize)
{
    struct binder_state *bs;
    struct binder_version vers;

    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return NULL;
    }

    bs->fd = open("/dev/binder", O_RDWR);
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open device (%s)\n",
                strerror(errno));
        goto fail_open;
    }

    if ((ioctl(bs->fd, BINDER_VERSION, &vers) == -1) ||
        (vers.protocol_version != BINDER_CURRENT_PROTOCOL_VERSION)) {
        fprintf(stderr, "binder: driver version differs from user space\n");
        goto fail_open;
    }

    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
        goto fail_map;
    }

    return bs;

fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return NULL;
}
```

在上述代码中，mapsize 表示映射内存区域的大小。

再看函数 binder_release，此函数与函数 binder_open 的功能相反。当 Binder 驱动退出时，通过函数 binder_release 来释放在打开以及其他操作过程中分配的空间，并且同时清理相关的数据信息。函数 binder_release 的具体实现代码如下所示。

```
static int binder_release(struct inode *nodp, struct file *filp)
{
        struct binder_proc *proc = filp->private_data;
        debugfs_remove(proc->debugfs_entry);
        binder_defer_work(proc, BINDER_DEFERRED_RELEASE);
        return 0;
}
```

在上述代码中，首先获取使用 private_data 数据的权限，找到当前进程、线程的 pid，这样可以得到在 open 过程中创建的以 pid 命名的用来输出当前 binder proc 对象的状态的只读文件；然

后调用函数 remove_proc_entry 实现删除操作；最后通过函数 binder_defer_work 和其参数 BINDER_DEFERRED_RELEASE 释放整个 binder_proc 对象的数据和分配的空间。

在文件 frameworks\native\cmds\servicemanager\binder.c 中，对应函数 binder_release 的具体实现代码如下所示：

```
void binder_release(struct binder_state *bs, uint32_t target)
{
    uint32_t cmd[2];
    cmd[0] = BC_RELEASE;
    cmd[1] = target;
    binder_write(bs, cmd, sizeof(cmd));
}
```

在上述代码中，调用函数 binder_write 写入 Binder 设备文件，具体实现代码如下所示：

```
int binder_write(struct binder_state *bs, void *data, size_t len)
{
    struct binder_write_read bwr;
    int res;

    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (uintptr_t) data;
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
                strerror(errno));
    }
    return res;
}
```

5.2.4 内存映射

在 Android 系统中，当打开 Binder 设备文件/dev/binder 后，需要调用函数 mmap 把设备内存映射到用户进程地址空间中，这样就可以像操作用户内存那样操作设备内存。在 Binder 设备中，对内存的映射操作是有限制的，比如 Binder 不能映射具有写权限的内存区域，最大能映射 4MB 的内存区域等。在 Android 系统中，大多数设备本身具有设备映射的设备内存，或者是在驱动初始化时由 vmalloc 或 kmalloc 等内核内存函数分配的，在 mmap 操作时分配 Binder 的设备内存。

函数 mmap 实现分配功能的实现流程如下所示：

（1）在内核虚拟映射表上获取一个可以使用的区域；

（2）分配物理页，并把物理页映射到获取的虚拟空间上；

（3）每个进程/线程只能执行一次映射操作，后面的操作都会返回错误。

函数 mmap 的具体实现流程如下所示：

（1）检查内存映射条件，包括映射内存大小（4MB）、flags、是否是第一次 mmap 等；

（2）获得地址空间，并把此空间的地址记录在进程信息（buffer）中；

（3）分配物理页面（pages）并记录下来；

（4）将 buffer 插入到进程信息的 buffer 列表中；

（5）调用函数 binder_update_page_range 将分配的物理页面和 vm 空间对应起来；

（6）调用函数 binder_insert_free_buffer 把进程中的 buffer 插入到进程信息中。

函数 mmap 的具体实现代码如下所示：

```
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;
        const char *failure_string;
        struct binder_buffer *buffer;
```

```
        if ((vma->vm_end - vma->vm_start) > SZ_4M)
                vma->vm_end = vma->vm_start + SZ_4M;

        binder_debug(BINDER_DEBUG_OPEN_CLOSE,
                     "binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
                     proc->pid, vma->vm_start, vma->vm_end,
                     (vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
                     (unsigned long)pgprot_val(vma->vm_page_prot));

        if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
                ret = -EPERM;
                failure_string = "bad vm_flags";
                goto err_bad_arg;
        }
        vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

        mutex_lock(&binder_mmap_lock);
        if (proc->buffer) {
                ret = -EBUSY;
                failure_string = "already mapped";
                goto err_already_mapped;
        }

        area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
        if (area == NULL) {
                ret = -ENOMEM;
                failure_string = "get_vm_area";
                goto err_get_vm_area_failed;
        }
        proc->buffer = area->addr;
        proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;
        mutex_unlock(&binder_mmap_lock);

#ifdef CONFIG_CPU_CACHE_VIPT
        if (cache_is_vipt_aliasing()) {
                while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
                        printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n",
proc->pid, vma->vm_start, vma->vm_end, proc->buffer);
                        vma->vm_start += PAGE_SIZE;
                }
        }
#endif
        proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) /
        PAGE_SIZE), GFP_KERNEL);
        if (proc->pages == NULL) {
                ret = -ENOMEM;
                failure_string = "alloc page array";
                goto err_alloc_pages_failed;
        }
        proc->buffer_size = vma->vm_end - vma->vm_start;

        vma->vm_ops = &binder_vm_ops;
        vma->vm_private_data = proc;

        if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
                ret = -ENOMEM;
                failure_string = "alloc small buf";
                goto err_alloc_small_buf_failed;
        }
        buffer = proc->buffer;
        INIT_LIST_HEAD(&proc->buffers);
        list_add(&buffer->entry, &proc->buffers);
        buffer->free = 1;
        binder_insert_free_buffer(proc, buffer);
        proc->free_async_space = proc->buffer_size / 2;
        barrier();
        proc->files = get_files_struct(proc->tsk);
        proc->vma = vma;
        proc->vma_vm_mm = vma->vm_mm;
```

```
        /*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n",
                proc->pid, vma->vm_start, vma->vm_end, proc->buffer);*/
        return 0;

err_alloc_small_buf_failed:
        kfree(proc->pages);
        proc->pages = NULL;
err_alloc_pages_failed:
        mutex_lock(&binder_mmap_lock);
        vfree(proc->buffer);
        proc->buffer = NULL;
err_get_vm_area_failed:
err_already_mapped:
        mutex_unlock(&binder_mmap_lock);
err_bad_arg:
        printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n",
               proc->pid, vma->vm_start, vma->vm_end, failure_string, ret);
        return ret;
}
```

在上述代码中，参数 vm_area_struct 是一个结构体，在 mmp 的具体实现中会非常有用。为了优化查找方法，内核专门维护了 VMA 的链表和树形结构。在结构 vm_area_struct 中，很多成员函数都是用来维护这个树形结构的。VMA 的功能是管理进程地址空间中不同区域的数据结构。该函数首先对内存映射进行检查，主要包括映射内存的大小、flags，以及是否已经映射过了，并判断其映射条件是否合法；然后，通过内核函数 get_vm_area 从系统中申请可用的虚拟内存空间，在内核中申请并保留一块连续的内核虚拟内存空间区域。接着，将 binder_proc 的用户地址偏移（即用户进程的 VMA 地址与 Binder 申请的 VMA 地址的偏差）存放到 proc→user_buffer_offset 中；再接着，使用 kzalloc 函数根据请求映射的内存空间大小，分配 Binder 的核心数据结构 binder_proc 的 pages 成员，它主要用来保存指向申请的物理页的指针；最后，为 VMA 指定了 vm_operations_struct 操作，并且将 vma→vm_private_data 指向了核心数据 proc。

到目前为止，就可以真正地开始分配物理内存（page）了。物理内存的分配工作是通过函数 binder_update_page_range 实现的，该函数主要完成如下所示的工作。

- alloc_page：分配页面。
- map_vm_area：为分配的内存做映射关系。
- vm_insert_page：把分配的物理页插入到用户 VMA 区域。

函数 binder_update_page_range 的具体实现代码如下所示：

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
                                    void *start, void *end,
                                    struct vm_area_struct *vma)
{
        void *page_addr;
        unsigned long user_page_addr;
        struct vm_struct tmp_area;
        struct page **page;
        struct mm_struct *mm;

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: %s pages %p-%p\n", proc->pid,
                     allocate ? "allocate" : "free", start, end);

        if (end <= start)
                return 0;

        trace_binder_update_page_range(proc, allocate, start, end);

        if (vma)
                mm = NULL;
        else
                mm = get_task_mm(proc->tsk);

        if (mm) {
                down_write(&mm->mmap_sem);
                vma = proc->vma;
```

```
            if (vma && mm != proc->vma_vm_mm) {
                pr_err("binder: %d: vma mm and task mm mismatch\n",
                       proc->pid);
                vma = NULL;
            }
        }

        if (allocate == 0)
            goto free_range;

        if (vma == NULL) {
            printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
                   "map pages in userspace, no vma\n", proc->pid);
            goto err_no_vma;
        }

        for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
            int ret;
            struct page **page_array_ptr;
            page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];

            BUG_ON(*page);
            *page = alloc_page(GFP_KERNEL | __GFP_HIGHMEM | __GFP_ZERO);
            if (*page == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                       "for page at %p\n", proc->pid, page_addr);
                goto err_alloc_page_failed;
            }
            tmp_area.addr = page_addr;
            tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
            page_array_ptr = page;
            ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
            if (ret) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                       "to map page at %p in kernel\n",
                       proc->pid, page_addr);
                goto err_map_kernel_failed;
            }
            user_page_addr =
                (uintptr_t)page_addr + proc->user_buffer_offset;
            ret = vm_insert_page(vma, user_page_addr, page[0]);
            if (ret) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                       "to map page at %lx in userspace\n",
                       proc->pid, user_page_addr);
                goto err_vm_insert_page_failed;
            }
            /* vm_insert_page does not seem to increment the refcount */
        }
        if (mm) {
            up_write(&mm->mmap_sem);
            mmput(mm);
        }
        return 0;

free_range:
        for (page_addr = end - PAGE_SIZE; page_addr >= start;
             page_addr -= PAGE_SIZE) {
            page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
            if (vma)
                zap_page_range(vma, (uintptr_t)page_addr +
                               proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
            unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
err_map_kernel_failed:
            __free_page(*page);
            *page = NULL;
err_alloc_page_failed:
            ;
        }
err_no_vma:
```

```
        if (mm) {
                up_write(&mm->mmap_sem);
                mmput(mm);
        }
        return -ENOMEM;
}
```

其中 vm_operations_struct 只包括了一个打开操作和一个关闭操作，具体的定义代码如下所示：

```
static struct vm_operations_struct binder_vm_ops = {
        .open = binder_vma_open,
        .close = binder_vma_close,
};
```

5.2.5 释放物理页面

在 Android 系统的 Binder 机制中，函数 binder_insert_free_buffer 的功能是进程中的 buffer 插入到进程信息中。也就是说，通过此函数能够将一个空闲内核缓冲区加入到进程中的空闲内核缓冲区的红黑树中。函数 binder_insert_free_buffer 的具体实现代码如下所示：

```
static void binder_insert_free_buffer(struct binder_proc *proc,
                                      struct binder_buffer *new_buffer)
{
        struct rb_node **p = &proc->free_buffers.rb_node;
        struct rb_node *parent = NULL;
        struct binder_buffer *buffer;
        size_t buffer_size;
        size_t new_buffer_size;
        BUG_ON(!new_buffer->free);
        new_buffer_size = binder_buffer_size(proc, new_buffer);
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: add free buffer, size %zd, "
                     "at %p\n", proc->pid, new_buffer_size, new_buffer);
        while (*p) {
                parent = *p;
                buffer = rb_entry(parent, struct binder_buffer, rb_node);
                BUG_ON(!buffer->free);
                buffer_size = binder_buffer_size(proc, buffer);
                if (new_buffer_size < buffer_size)
                        p = &parent->rb_left;
                else
                        p = &parent->rb_right;
        }
        rb_link_node(&new_buffer->rb_node, parent, p);
        rb_insert_color(&new_buffer->rb_node, &proc->free_buffers);
}
```

5.2.6 分配内核缓冲区

在 Android 系统中，Binder 在使用 Buffer 时一次声明一个 proc（对应一个进程）的 Buffer 总大小，然后分配一页并做好映射。当在使用时，如果发现空间不足，会接着映射并把这个 Buffer 拆成两个，并把剩余的继续放到 free_buffers 里面。在 Binder 驱动程序中，函数*binder_alloc_buf 的功能是分配内核缓冲区，具体代码如下所示：

```
static struct binder_buffer *binder_alloc_buf(struct binder_proc *proc,
                                              size_t data_size,
                                              size_t offsets_size, int is_async)
{
        struct rb_node *n = proc->free_buffers.rb_node;
        struct binder_buffer *buffer;
        size_t buffer_size;
        struct rb_node *best_fit = NULL;
        void *has_page_addr;
        void *end_page_addr;
        size_t size;
        if (proc->vma == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf, no vma\n",
                       proc->pid);
```

```
        return NULL;
    }
    size = ALIGN(data_size, sizeof(void *)) +
           ALIGN(offsets_size, sizeof(void *));
    if (size < data_size || size < offsets_size) {
        binder_user_error("binder: %d: got transaction with invalid "
                "size %zd-%zd\n", proc->pid, data_size, offsets_size);
        return NULL;
    }
    if (is_async &&
       proc->free_async_space < size + sizeof(struct binder_buffer)) {
        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: binder_alloc_buf size %zd"
                     "failed, no async space left\n", proc->pid, size);
        return NULL;
    }
    while (n) {
        buffer = rb_entry(n, struct binder_buffer, rb_node);
        BUG_ON(!buffer->free);
        buffer_size = binder_buffer_size(proc, buffer);
        if (size < buffer_size) {
                best_fit = n;
                n = n->rb_left;
        } else if (size > buffer_size)
                n = n->rb_right;
        else {
                best_fit = n;
                break;
        }
    }
    if (best_fit == NULL) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf size %zd failed, "
              "no address space\n", proc->pid, size);
        return NULL;
    }
    if (n == NULL) {
        buffer = rb_entry(best_fit, struct binder_buffer, rb_node);
        buffer_size = binder_buffer_size(proc, buffer);
    }
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
               "binder: %d: binder_alloc_buf size %zd got buff"
               "er %p size %zd\n", proc->pid, size, buffer, buffer_size);
    has_page_addr =
        (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK);
    if (n == NULL) {
        if (size + sizeof(struct binder_buffer) + 4 >= buffer_size)
                buffer_size = size; /* no room for other buffers */
        else
                buffer_size = size + sizeof(struct binder_buffer);
    }
    end_page_addr =
        (void *)PAGE_ALIGN((uintptr_t)buffer->data + buffer_size);
    if (end_page_addr > has_page_addr)
        end_page_addr = has_page_addr;
    if (binder_update_page_range(proc, 1,
        (void *)PAGE_ALIGN((uintptr_t)buffer->data), end_page_addr, NULL))
        return NULL;
    rb_erase(best_fit, &proc->free_buffers);
    buffer->free = 0;
    binder_insert_allocated_buffer(proc, buffer);
    if (buffer_size != size) {
        struct binder_buffer *new_buffer = (void *)buffer->data + size;
        list_add(&new_buffer->entry, &buffer->entry);
        new_buffer->free = 1;
        binder_insert_free_buffer(proc, new_buffer);
    }
    binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
               "binder: %d: binder_alloc_buf size %zd got "
               "%p\n", proc->pid, size, buffer);
    buffer->data_size = data_size;
    buffer->offsets_size = offsets_size;
```

```
        buffer->async_transaction = is_async;
        if (is_async) {
                proc->free_async_space -= size + sizeof(struct binder_buffer);
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                             "binder: %d: binder_alloc_buf size %zd "
                             "async free %zd\n", proc->pid, size,
                             proc->free_async_space);
        }
        return buffer;
}
```

再看函数 binder_insert_allocated_buffer，功能是将分配的内核缓冲区添加到目标进程的已分配物理页面的内核缓冲区红黑树中。函数 binder_insert_allocated_buffer 的具体实现代码如下所示：

```
static void binder_insert_allocated_buffer(struct binder_proc *proc,
                                           struct binder_buffer *new_buffer)
{
        struct rb_node **p = &proc->allocated_buffers.rb_node;
        struct rb_node *parent = NULL;
        struct binder_buffer *buffer;
        BUG_ON(new_buffer->free);
        while (*p) {
                parent = *p;
                buffer = rb_entry(parent, struct binder_buffer, rb_node);
                BUG_ON(buffer->free);
                if (new_buffer < buffer)
                        p = &parent->rb_left;
                else if (new_buffer > buffer)
                        p = &parent->rb_right;
                else
                        BUG();
        }
        rb_link_node(&new_buffer->rb_node, parent, p);
        rb_insert_color(&new_buffer->rb_node, &proc->allocated_buffers);
}
```

5.2.7 释放内核缓冲区

在 Android 系统中，函数 binder_free_buf 的功能是释放内核缓冲区的操作，具体实现代码如下所示：

```
static void binder_free_buf(struct binder_proc *proc,
                            struct binder_buffer *buffer)
{
        size_t size, buffer_size;
        //计算要释放的内核缓冲区 Bufffer 的大小，保存在 buffer_size 中
        buffer_size = binder_buffer_size(proc, buffer);
        //计算数据缓冲区和偏移数组缓冲区的大小，并保存在 size 中
        size = ALIGN(buffer->data_size, sizeof(void *)) +
                ALIGN(buffer->offsets_size, sizeof(void *));

        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                     "binder: %d: binder_free_buf %p size %zd buffer"
                     "_size %zd\n", proc->pid, buffer, size, buffer_size);

        BUG_ON(buffer->free);
        BUG_ON(size > buffer_size);
        BUG_ON(buffer->transaction != NULL);
        BUG_ON((void *)buffer < proc->buffer);
        BUG_ON((void *)buffer > proc->buffer + proc->buffer_size);
        //检查要释放的内核缓冲区 Buffer 是否用于异步事物
        if (buffer->async_transaction) {
                proc->free_async_space += size + sizeof(struct binder_buffer);

                binder_debug(BINDER_DEBUG_BUFFER_ALLOC_ASYNC,
                             "binder: %d: binder_free_buf size %zd "
                             "async free %zd\n", proc->pid, size,
                             proc->free_async_space);
        }
        //释放内核缓冲区
```

```
        binder_update_page_range(proc, 0,
                (void *)PAGE_ALIGN((uintptr_t)buffer->data),
                (void *)(((uintptr_t)buffer->data + buffer_size) & PAGE_MASK),
                NULL);
        rb_erase(&buffer->rb_node, &proc->allocated_buffers);
        buffer->free = 1;
        if (!list_is_last(&buffer->entry, &proc->buffers)) {
                struct binder_buffer *next = list_entry(buffer->entry.next,
                                                struct binder_buffer, entry);
                if (next->free) {
                        rb_erase(&next->rb_node, &proc->free_buffers);
                        binder_delete_free_buffer(proc, next);
                }
        }
        if (proc->buffers.next != &buffer->entry) {
                struct binder_buffer *prev = list_entry(buffer->entry.prev,
                                                struct binder_buffer, entry);
                if (prev->free) {
                        binder_delete_free_buffer(proc, buffer);
                        rb_erase(&prev->rb_node, &proc->free_buffers);
                        buffer = prev;
                }
        }
        binder_insert_free_buffer(proc, buffer);
}
```

再看函数*buffer_start_page 和*buffer_end_page，用于计算结构体 binder_buffer 所占用的虚拟页面的地址。具体实现代码如下所示：

```
static void *buffer_start_page(struct binder_buffer *buffer)
{
        return (void *)((uintptr_t)buffer & PAGE_MASK);
}
static void *buffer_end_page(struct binder_buffer *buffer)
{
        return (void *)(((uintptr_t)(buffer + 1) - 1) & PAGE_MASK);
}
```

再看函数 binder_delete_free_buffer，功能是删除结构体 binder_buffer，具体实现代码如下所示：

```
static void binder_delete_free_buffer(struct binder_proc *proc,
                                      struct binder_buffer *buffer)
{
        struct binder_buffer *prev, *next = NULL;
        int free_page_end = 1;
        int free_page_start = 1;

        BUG_ON(proc->buffers.next == &buffer->entry);
        prev = list_entry(buffer->entry.prev, struct binder_buffer, entry);
        BUG_ON(!prev->free);
        if (buffer_end_page(prev) == buffer_start_page(buffer)) {
                free_page_start = 0;
                if (buffer_end_page(prev) == buffer_end_page(buffer))
                        free_page_end = 0;
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                             "binder: %d: merge free, buffer %p "
                             "share page with %p\n", proc->pid, buffer, prev);
        }

        if (!list_is_last(&buffer->entry, &proc->buffers)) {
                next = list_entry(buffer->entry.next,
                                  struct binder_buffer, entry);
                if (buffer_start_page(next) == buffer_end_page(buffer)) {
                        free_page_end = 0;
                        if (buffer_start_page(next) ==
                            buffer_start_page(buffer))
                                free_page_start = 0;
                        binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                                     "binder: %d: merge free, buffer"
                                     " %p share page with %p\n", proc->pid,
                                     buffer, prev);
```

```
                }
        }
        list_del(&buffer->entry);
        if (free_page_start || free_page_end) {
                binder_debug(BINDER_DEBUG_BUFFER_ALLOC,
                             "binder: %d: merge free, buffer %p do "
                             "not share page%s%s with with %p or %p\n",
                             proc->pid, buffer, free_page_start ? "" : " end",
                             free_page_end ? "" : " start", prev, next);
                binder_update_page_range(proc, 0, free_page_start ?
                        buffer_start_page(buffer) : buffer_end_page(buffer),
                        (free_page_end ? buffer_end_page(buffer) :
                        buffer_start_page(buffer)) + PAGE_SIZE, NULL);
        }
}
```

5.2.8 查询内核缓冲区

在 Android 系统中，函数*binder_buffer_lookup 的功能是根据一个用户空间地址查询一个内核缓冲区，具体实现代码如下所示：

```
static struct binder_buffer *binder_buffer_lookup(struct binder_proc *proc,
                                                   void __user *user_ptr)
{
        struct rb_node *n = proc->allocated_buffers.rb_node;
//对于已经分配的 Buffer 空间，以内存地址为索引加入红黑树 allocated_buffers 中
        struct binder_buffer *buffer;
        struct binder_buffer *kern_ptr;

        kern_ptr = user_ptr - proc->user_buffer_offset
                - offsetof(struct binder_buffer, data);
/* 进程 ioctl 传下来的指针并不是 binder_buffer 的地址，而直接是 binder_buffer.data 的地址。
user_buffer_offset 用户空间和内核空间，被映射区域起始地址之间的偏移。*/
        while (n) {
                buffer = rb_entry(n, struct binder_buffer, rb_node);
                BUG_ON(buffer->free);

                if (kern_ptr < buffer)
                        n = n->rb_left;
                else if (kern_ptr > buffer)
                        n = n->rb_right;
                else
                        return buffer;
        }
        return NULL;
}
```

5.3 Binder 封装库

在 Android 5.0 源代码中，在各层都存在和 Binder 机制有关的实现。其中主要的 Binder 库由本地原生代码实现，Java 和 C++层都定义有同样功能的供应用程序使用的 Binder 接口，它们实际上都是调用原生 Binder 库的实现。各个实现层次的具体说明如下所示。

（1）Binder 驱动部分

驱动部分位于 Binder 结构的最底层（即 Linux 内核层），有关这部分的分析已经在本章前面进行介绍了。这部分用于实现 Binder 的设备驱动，主要实现如下所示的功能：

- 组织 Binder 的服务节点；
- 调用 Binder 相关的处理线程；
- 完成实际的 Bainder 传输。

（2）Binder Adapter 层

Binder Adapter 层是对 Binder 驱动的封装，主要功能是操作 Binder 驱动。应用程序无需直接和 Binder 驱动程序关联，关联文件包括 IPCThreadState.cpp、ProcessState.cpp 和 Parcel.cpp 中的一

些内容。

Binder 核心库是 Binder 框架的核心实现，主要包括 IBinder、Binder（服务器端）和 BpBinder（客户端）。

（3）顶层

顶层的 Binder 框架和具体的客户端/服务端都分别有 Java 和 C++两种实现方案，主要供应用程序使用，如摄像头和多媒体，这部分通过调用 Binder 的核心库来实现。

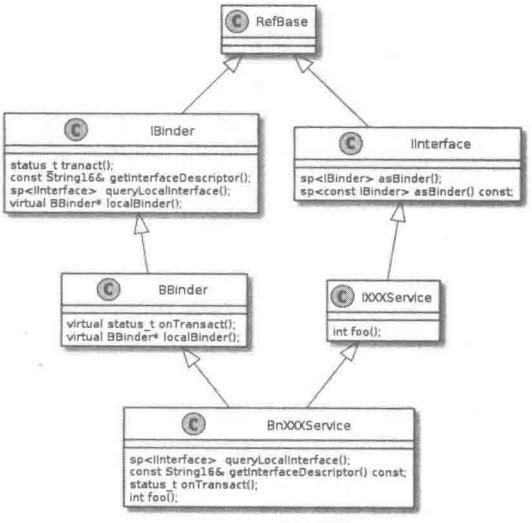
在文件 frameworks\native\include\binder\IInterface.h 中，分别定义了定义类 IInterface、类模板 BnInterface 和类模板 BpInterface。其中类模板 BnInterface 和 BpInterface 用于实现 Service 组件和 Client 组件，具体定义代码如下所示：

```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16& _descriptor);
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    virtual IBinder*onAsBinder();
};

// ----------------------------------------------------------------------

template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
                            BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder*onAsBinder();
};
```

在使用这两个模版的时候，起到了双继承的作用。使用者定义一个接口 INTERFACE，然后使用 BnInterface 和 BpInterface 两个模板结合自己的接口，构建自己的 BnXXX 和 BpXXX 两个类。

5.3.1　类 BBinder

类模板 BnInterface 继承于类 BBinder，BBinder 是服务的载体，和 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。从 Binder 驱动的角度，每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类。然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact()又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。整个 BnXXXService 的类关系如图 5-2 所示。



▲图 5-2　类关系图

由图 5-2 可以看出，BnXXXService 包含如下两部分。

- IXXXService：服务的主体的接口。
- BBinder：是服务的载体，和 Binder 驱动共同工作，保证客户的请求最终是对一个 Binder 对象（BBinder 类）的调用。

每一个服务就是一个 BBinder 类，Binder 驱动负责找出服务对应的 BBinder 类。然后把这个 BBinder 类返回给 IPCThreadState，IPCThreadState 调用 BBinder 的 transact()。BBinder 的 transact() 又会调用 onTransact()。BBinder::onTransact()是虚函数，所以实际是调用 BnXXXService::onTransact()，这样就可在 BnXXXService::onTransact()中完成具体的服务函数的调用。

在文件 frameworks\native\include\binder\Binder.h 中，定义类 BBinder 的代码如下所示：

```
class BBinder : public IBinder
{
public:
                    BBinder();
    virtual const String16& getInterfaceDescriptor() const;
    virtual bool isBinderAlive() const;
    virtual status_t pingBinder();
    virtual status_t dump(int fd, const Vector<String16>& args);
    virtual status_t transact(  uint32_t code,
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
    virtual status_t linkToDeath(const sp<DeathRecipient>& recipient,
                                void* cookie = NULL,
                                uint32_t flags = 0);
    virtual status_t unlinkToDeath( const wp<DeathRecipient>& recipient,
                                    void* cookie = NULL,
                                    uint32_t flags = 0,
                                    wp<DeathRecipient>* outRecipient = NULL);
    virtual void attachObject(  const void* objectID,
                                    void* object,
                                    void* cleanupCookie,
                                    object_cleanup_func func);
    virtual void*findObject(const void* objectID) const;
    virtual void detachObject(const void* objectID);
    virtual BBinder*localBinder();
protected:
    virtual           ~BBinder();
    virtual status_t onTransact( uint32_t code,
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
private:
                    BBinder(const BBinder& o);
          BBinder&operator=(const BBinder& o);
    class Extras;
          Extras*mExtras;
          void*mReserved0;
};
```

在类 BBinder 中，当一个 Binder 代理对象通过 Binder 驱动程序向一个 Binder 本地对象发出一个进程通信请求时，Binder 驱动程序会调用该 Binder 本地对象的成员函数 transact 来处理这个请求。函数 transact 在文件 frameworks\native\libs\binder\Binder.cpp 中实现，具体代码如下所示：

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
    }
    return err;
}
```

在上述代码中，PING_TRANSACTION 请求用来检查对象是否还存在，此处只是简单地把 pingBinder 的返回值返回给调用者，将其他的请求交给 onTransact 来处理。onTransact 是在 Bbinder 中声明的一个 protected 类型的虚函数，此功能在它的子类中实现。

再看另外一个重要的成员函数 onTransact，功能是分发和业务相关的进程间通信请求。函数 onTransact 也是在文件 frameworks\native\libs\binder\Binder.cpp 中定义，具体实现代码如下所示：

```
status_t BBinder::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch (code) {
        case INTERFACE_TRANSACTION:
            reply->writeString16(getInterfaceDescriptor());
            return NO_ERROR;

        case DUMP_TRANSACTION: {
            int fd = data.readFileDescriptor();
            int argc = data.readInt32();
            Vector<String16> args;
            for (int i = 0; i < argc && data.dataAvail() > 0; i++) {
               args.add(data.readString16());
            }
            return dump(fd, args);
        }
        case SYSPROPS_TRANSACTION: {
            report_sysprop_change();
            return NO_ERROR;
        }
        default:
            return UNKNOWN_TRANSACTION;
    }
}
```

5.3.2　类 BpRefBase

类模板 BpInterface 继承于类 BpRefBase，起了一个代理作用。BpRefBase 以上是业务逻辑（要实现什么功能），BpRefBase 以下是数据传输（通过 binder 如何将功能实现）。在文件 frameworks\native\include\binder\Binder.h 中，定义类 BpRefBase 的代码如下所示：

```
class BpRefBase : public virtual RefBase
{
protected:
                        BpRefBase(const sp<IBinder>& o);
    virtual              ~BpRefBase();
    virtual void          onFirstRef();
    virtual void          onLastStrongRef(const void* id);
    virtual bool          onIncStrongAttempted(uint32_t flags, const void* id);

    inline  IBinder*remote(){ return mRemote; }
    inline  IBinder*remote()const{ return mRemote; }

private:
                        BpRefBase(const BpRefBase& o);
    BpRefBase&           operator=(const BpRefBase& o);

    IBinder* const        mRemote;
    RefBase::weakref_type* mRefs;
    volatile int32_t      mState;
};

}; // namespace android
```

类 BpRefBase 中的成员函数 transact 用于向运行在 Server 进程中的 Service 组件发送进程之间的通信请求，这是通过 Binder 驱动程序间接实现的。函数 transact 的具体实现代码如下所示：

```
status_t BpBinder::transact(
uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
```

```
// Once a binder has died, it will never come back to life.
if (mAlive) {
status_t status = IPCThreadState::self()->transact(
mHandle, code, data, reply, flags);
if (status == DEAD_OBJECT) mAlive = 0;
return status;
}
return DEAD_OBJECT;
}
```

各个参数的具体说明如下所示。

- code：表示请求的 ID 号。
- data：表示请求的参数。
- reply：表示返回的结果。
- flags：是一些额外的标识，如 FLAG_ONEWAY，通常为 0。

5.3.3 类 IPCThreadState

前面介绍的类 BBinder 和类 BpRefBase 都是通过类 IPCThreadState 和 Binder 的驱动程序交互实现的。类 IPCThreadState 在文件 frameworks\native\include\binder\IPCThreadState.h 中实现，具体实现代码如下所示：

```
class IPCThreadState
{
public:
    static  IPCThreadState*     self();
    static  IPCThreadState*     selfOrNull();  // self(), but won't instantiate

            sp<ProcessState>    process();

            status_t clearLastError();

            int getCallingPid();
            int getCallingUid();

            void setStrictModePolicy(int32_t policy);
            int32_t getStrictModePolicy() const;

            void setLastTransactionBinderFlags(int32_t flags);
            int32_t getLastTransactionBinderFlags() const;

            int64_t clearCallingIdentity();
            void restoreCallingIdentity(int64_t token);

            void flushCommands();

            void joinThreadPool(bool isMain = true);

            // Stop the local process.
            void stopProcess(bool immediate = true);

            status_t transact(int32_t handle,
                              uint32_t code, const Parcel& data,
                              Parcel* reply, uint32_t flags);

            void incStrongHandle(int32_t handle);
            void decStrongHandle(int32_t handle);
            void incWeakHandle(int32_t handle);
            void decWeakHandle(int32_t handle);
            status_t attemptIncStrongHandle(int32_t handle);
            static  void expungeHandle(int32_t handle, IBinder* binder);
            status_t requestDeathNotification(   int32_t handle,
                                                 BpBinder* proxy);
            status_t clearDeathNotification( int32_t handle,
                                             BpBinder* proxy);

            static void shutdown();
```

```
    // Call this to disable switching threads to background scheduling when
    // receiving incoming IPC calls.  This is specifically here for the
    // Android system process, since it expects to have background apps calling
    // in to it but doesn't want to acquire locks in its services while in
    // the background.
    static  void disableBackgroundScheduling(bool disable);

private:
                            IPCThreadState();
                            ~IPCThreadState();

        status_t sendReply(const Parcel& reply, uint32_t flags);
        status_t waitForResponse(Parcel *reply,
                                 status_t *acquireResult=NULL);
        status_t talkWithDriver(bool doReceive=true);
        status_t writeTransactionData(int32_t cmd,
                                      uint32_t binderFlags,
                                      int32_t handle,
                                      uint32_t code,
                                      const Parcel& data,
                                      status_t* statusBuffer);
        status_t executeCommand(int32_t command);

        void clearCaller();

    static  void threadDestructor(void *st);
    static  void freeBuffer(Parcel* parcel,
                            const uint8_t* data, size_t dataSize,
                            const size_t* objects, size_t objectsSize,
                            void* cookie);

    const   sp<ProcessState> mProcess;
    const   pid_t mMyThreadId;
            Vector<BBinder*> mPendingStrongDerefs;
            Vector<RefBase::weakref_type*> mPendingWeakDerefs;

            Parcel mIn;
            Parcel mOut;
            status_t mLastError;
            pid_t mCallingPid;
            uid_t mCallingUid;
            int32_t mStrictModePolicy;
            int32_t mLastTransactionBinderFlags;
};

}; // namespace android
```

在类 IPCThreadState 中，成员函数用于实现数据处理。在 transact 请求中将请求的数据经过 Binder 设备发送给了 Service，Service 处理完请求后，又将结果原路返回给客户端。函数 transact 在文件 frameworks\native\libs\binder\IPCThreadState.cpp 中定义，具体实现代码如下所示：

```
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;
    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }
```

```
    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }
    return err;
}
```

5.4 初始化 Java 层 Binder 框架

虽然 Java 层 Binder 系统是 Native 层 Binder 系统的一个 Mirror，但这个 Mirror 终归还需借助 Native 层 Binder 系统来开展工作，即它和 Native 层 Binder 有着千丝万缕的关系，故一定要在 Java 层 Binder 正式工作之前建立这种关系。下面分析 Java 层 Binder 框架是如何初始化的。

在 Android 系统中，其中函数 register_android_os_Binder 专门负责搭建 Java Binder 和 Native Binder 的交互关系。此函数在如下所示的文件中实现：

```
\frameworks\base\core\jni\android_util_Binder.cpp
```

函数 register_android_os_Binder 的具体实现代码如下所示：

```
int register_android_os_Binder(JNIEnv* env)
{
  //初始化 Java Binder 类和 Native 层的关系
  if (int_register_android_os_Binder(env) < 0)
      return -1;
  //初始化 Java BinderInternal 类和 Native 层的关系
  if (int_register_android_os_BinderInternal(env) < 0)
      return -1;
 //初始化 Java BinderProxy 类和 Native 层的关系
  if (int_register_android_os_BinderProxy(env) < 0)
      return -1;
 //初始化 Java Parcel 类和 Native 层的关系
  if (int_register_android_os_Parcel(env) < 0)
      return -1;
```

```
    return 0;
}
```

根据上面的代码可知，函数 register_android_os_Binder 完成了 Java 层 Binder 架构中最重要的 4 个类的初始化工作。在接下来的内容中，将详细分析 Binder 类的初始化进程。

函数 int_register_android_os_Binder 实现了 Binder 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_Binder(JNIEnv* env)
{
jclass clazz;
//kBinderPathName 为 Java 层中 Binder 类的全路径名，"android/os/Binder"
clazz = env->FindClass(kBinderPathName);
/*
gBinderOffsets 是一个静态类对象，它专门保存 Binder 类的一些在 JNI 层中使用的信息，
如成员函数 execTransact 的 methodID,Binder 类中成员 mObject 的 fieldID

*/
 gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 gBinderOffsets.mExecTransact
              = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
 gBinderOffsets.mObject
              = env->GetFieldID(clazz, "mObject", "I");
//注册 Binder 类中 native 函数的实现
 return AndroidRuntime::registerNativeMethods(
                    env, kBinderPathName,
                    gBinderMethods, NELEM(gBinderMethods));
}
```

从上面的代码可知，gBinderOffsets 对象保存了和 Binder 类相关的某些在 JNI 层中使用的信息。

下一个初始化的类是 BinderInternal，其代码在 int_register_android_os_BinderInternal 函数中。此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_BinderInternal(JNIEnv* env)
{
 jclass clazz;
 //根据 BinderInternal 的全路径名找到代表该类的 jclass 对象。全路径名为
 // "com/android/internal/os/BinderInternal"
 clazz = env->FindClass(kBinderInternalPathName);
 //gBinderInternalOffsets 也是一个静态对象，用来保存 BinderInternal 类的一些信息
 gBinderInternalOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 forceBinderGc 的 methodID
 gBinderInternalOffsets.mForceGc
            = env->GetStaticMethodID(clazz, "forceBinderGc", "()V");
   //注册 BinderInternal 类中 native 函数的实现
  return AndroidRuntime::registerNativeMethods(
                  env, kBinderInternalPathName,
                  gBinderInternalMethods, NELEM(gBinderInternalMethods));
}
```

由此可见，int_register_android_os_BinderInternal 的功能和 int_register_android_os_Binder 的功能类似，主要包括以下两方面：

- 获取一些有用的 methodID 和 fieldID，这表明 JNI 层一定会向上调用 Java 层的函数；
- 注册相关类中 native 函数的实现。

函数 int_register_android_os_BinderProxy 完成了 BinderProxy 类的初始化工作，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
  jclass clazz;

 clazz = env->FindClass("java/lang/ref/WeakReference");
 //gWeakReferenceOffsets 用来和 WeakReference 类打交道
 gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
 //获取 WeakReference 类 get 函数的 methodID
  gWeakReferenceOffsets.mGet= env->GetMethodID(clazz, "get",
                            "()Ljava/lang/Object;");
```

```
    clazz = env->FindClass("java/lang/Error");
    //gErrorOffsets 用来和 Error 类打交道
    gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);

    clazz = env->FindClass(kBinderProxyPathName);
    //gBinderProxyOffsets 用来和 BinderProxy 类打交道
    gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    gBinderProxyOffsets.mConstructor= env->GetMethodID(clazz, "<init>", "()V");
    ..... //获取 BinderProxy 的一些信息
    clazz = env->FindClass("java/lang/Class");
    //gClassOffsets 用来和 Class 类打交道
    gClassOffsets.mGetName =env->GetMethodID(clazz,
                          "getName", "()Ljava/lang/String;");
    //注册 BinderProxy native 函数的实现
    return AndroidRuntime::registerNativeMethods(env,
         kBinderProxyPathName,gBinderProxyMethods,
                           NELEM(gBinderProxyMethods));
}
```

根据上面的代码可知，int_register_android_os_BinderProxy 函数除了初始化 BinderProxy 类外，还获取了 WeakReference 类和 Error 类的一些信息。由此可见，BinderProxy 对象的生命周期会委托 WeakReference 来管理，故 JNI 层会获取该类 get 函数的 MethodID。

到此为止，Java Binder 几个重要成员的初始化已完成，同时在代码中定义了几个全局静态对象，分别是 gBinderOffsets、gBinderInternalOffsets 和 gBinderProxyOffsets。

5.5 分析 MediaServer 的通信机制

MediaServer（缩写为 MS）是一个可执行程序，Server 是系统中很多 Service 的“沃土”和“家园”。MediaServer 主要包括如下所示的服务。

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

在本节的内容中，将以 MediaServer 系统为例讲解整个 Binder 系统的实现流程。

5.5.1 MediaServer 的入口函数

MediaServer 是一个可执行程序，在如下文件中定义：

```
\frameworks\av\media\mediaserver\main_mediaserver.cpp
```

文件 main_mediaserver.cpp 的入口函数是 main，代码如下所示：

```
int main(int argc, char** argv)
{
    signal(SIGPIPE, SIG_IGN);
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

上述代码的实现流程如下所示：

- 获得一个 ProcessState 实例；
- 调用 defaultServiceManager，得到一个 IServiceManager；
- 初始化音频系统的 AudioFlinger 服务；

- 实现多媒体系统的 MediaPlayer 服务，将以它作为主切入点；
- 实现音频系统的 AudioPolicy 服务；
- 创建一个线程池；
- 将自己加入到刚才的线程池。

5.5.2 ProcessState

在 main 函数的开始处便调用了 ProcessState，由于每个进程只有一个 ProcessState，所以它是独一无二的。调用函数 ProcessState 的代码如下所示：

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

函数 self()在如下文件中定义：

```
\frameworks\native\libs\binder\ProcessState.cpp
```

函数 self()的实现代码如下所示：

```
sp<ProcessState> ProcessState::self()
{
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState;
    return gProcess;
}
```

在上述代码中，self 函数采用了单例模式，根据这个以及 Process State 的名字这很明确地传递了一个信息：每个进程只有一个 ProcessState 对象。

接下来看 ProcessState 的构造函数，功能是打开了 Binder 设备。此函数也是在文件 ProcessState.cpp 中实现，具体实现代码如下所示：

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())
    , mVMStart(MAP_FAILED)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        // XXX Ideally, there should be a specific define for whether we
        // have mmap (or whether we could possibly have the kernel module
        // availabla).
#if !defined(HAVE_WIN32_IPC)
        // mmap the binder, providing a chunk of virtual address space to receive transactions.
        mVMStart = mmap(0,BINDER_VM_SIZE,PROT_READ,MAP_PRIVATE|MAP_NORESERVE,mDriverFD,0);
        if (mVMStart == MAP_FAILED) {
            // *sigh*
            ALOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");
            close(mDriverFD);
            mDriverFD = -1;
        }
#else
        mDriverFD = -1;
#endif
    }

    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened. Terminating.");
}
```

在上述代码中，open_driver 的作用就是打开设备“/dev/binder”，这是 Android 在内核中为完成进程间通信而专门设置的一个虚拟设备，具体实现代码如下所示：

```
static int open_driver()
```

```
{
    int fd = open("/dev/binder", O_RDWR);
    if (fd >= 0) {
        fcntl(fd, F_SETFD, FD_CLOEXEC);
        int vers;
        status_t result = ioctl(fd, BINDER_VERSION, &vers);
        if (result == -1) {
            ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));
            close(fd);
            fd = -1;
        }
        if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
            ALOGE("Binder driver protocol does not match user space protocol!");
            close(fd);
            fd = -1;
        }
        size_t maxThreads = 15;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
        if (result == -1) {
            ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
        }
    } else {
        ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
    }
    return fd;
}
```

由此可见，函数 Process::self 具备如下所示的功能：

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

5.5.3 defaultServiceManager

函数 defaultServiceManager 在如下所示的文件中实现：

```
frameworks\native\libs\binder\IServiceManager.cpp
```

函数 defaultServiceManager 的功能是返回一个 IServiceManager 对象，通过此对象可以与另一个进程 ServiceManager 进行交互，其具体实现代码如下所示：

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }

    return gDefaultServiceManager;
}
```

在上述代码中，调用了类 ProcessState 中的函数 getContextObject。注意传给它的参数是 NULL，即 0。

下面再看函数 getContextObject，此函数在文件 ProcessState.cpp 中定义，具体实现代码如下所示：

```
sp<IBinder> ProcessState::getContextObject(const String16& name, const sp<IBinder>&
caller)
{
    mLock.lock();
    sp<IBinder> object(
```

```
        mContexts.indexOfKey(name) >= 0 ? mContexts.valueFor(name) : NULL);
    mLock.unlock();
    if (object != NULL) return object;
    if (mManagesContexts) {
        ALOGE("getContextObject(%s) failed, but we manage the contexts!\n",
            String8(name).string());
        return NULL;
    }

    IPCThreadState* ipc = IPCThreadState::self();
    {
        Parcel data, reply;
        // no interface token on this magic transaction
        data.writeString16(name);
        data.writeStrongBinder(caller);
        status_t result = ipc->transact(0 /*magic*/, 0, data, &reply, 0);
        if (result == NO_ERROR) {
            object = reply.readStrongBinder();
        }
    }

    ipc->flushCommands();

    if (object != NULL) setContextObject(object, name);
    return object;
}
```

上述代码调用了函数 getStrongProxyForHandle，它的调用参数名为 handle。函数 getStrongProxyForHandle 在文件 ProcessState.cpp 中实现，具体实现代码如下所示：

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);
    if (e != NULL) {
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

BpBinder 和 BBinder 都是 Android 中与 Binder 通信相关的代表，它们都是从 IBinder 类中派生而来。BpBinder 是客户端用来与 Server 交互的代理类，p 即 Proxy 的意思。

BBinder 则是与 proxy 相对的一端，它是 proxy 交互的目的端。如果说 Proxy 代表客户端，那么 BBinder 则代表服务端。这里的 BpBinder 和 BBinder 是一一对应，即某个 BpBinder 只能和对应的 BBinder 交互。我们当然不希望通过 BpBinderA 发送的请求，却由 BBinderB 来处理。

在函数 defaultServiceManager()中创建了 BpBinder，给 BpBinder 构造函数传的参数 handle 的值是 0，0 代表了 ServiceManager 所对应的 BBinder。BpBinder 在如下所示的文件中实现：

```
frameworks\native\libs\binder\BpBinder.cpp
```

BpBinder 的具体实现代码如下所示：

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);
```

```
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    IPCThreadState::self()->incWeakHandle(handle);
}
```

通过上述代码可知，BpBinder、BBinder 这两个类没有任何地方操作 ProcessState 打开的那个/dev/binder 设备。也就是说，这两个 Binder 类没有和 Binder 设备直接交互。先看 interface_cast 的具体实现，在文件 IInterface.h 中定义，其代码如下所示：

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

由此可见，interface_cast 仅是一个模板函数，所以，interface_cast<IServiceManager>()等价于下面的代码：

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

这样又转移到 IServiceManager 对象中去了。

（1）定义业务逻辑

IServiceManager 定义了 ServiceManager 所提供的服务，在如下所示的文件中定义：

```
frameworks\native\include\binder\IServiceManager.h
```

IServiceManager 的具体实现代码如下所示：

```
class IServiceManager : public IInterface
{
public:
    DECLARE_META_INTERFACE(ServiceManager);
    virtual sp<IBinder> getService( const String16& name) const = 0;
    virtual sp<IBinder> heckService( const String16& name) const = 0;
    virtual status_t addService( const String16& name,
                                 const sp<IBinder>& service,
                                 bool allowIsolated = false) = 0;
    virtual Vector<String16>    listServices() = 0;

    enum {
        GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
        CHECK_SERVICE_TRANSACTION,
        ADD_SERVICE_TRANSACTION,
        LIST_SERVICES_TRANSACTION,
    };
};
sp<IServiceManager> defaultServiceManager();

template<typename INTERFACE>
status_t getService(const String16& name, sp<INTERFACE>* outService)
{
    const sp<IServiceManager> sm = defaultServiceManager();
    if (sm != NULL) {
        *outService = interface_cast<INTERFACE>(sm->getService(name));
        if ((*outService) != NULL) return NO_ERROR;
    }
    return NAME_NOT_FOUND;
}

bool checkCallingPermission(const String16& permission);
bool checkCallingPermission(const String16& permission,
                            int32_t* outPid, int32_t* outUid);
bool checkPermission(const String16& permission, pid_t pid, uid_t uid);
class BnServiceManager : public BnInterface<IServiceManager>
{
public:
    virtual status_t    onTransact( uint32_t code,
```

```
                                const Parcel& data,
                                Parcel* reply,
                                uint32_t flags = 0);
};
}; // namespace android

#endif // ANDROID_ISERVICE_MANAGER_H
```

由此可见，Android 通过 DECLARE_META_INTERFACE 和 IMPLENT 宏，将业务和通信牢牢地联系在了一起。

（2）业务与通信的挂钩

DECLARE_META_INTERFACE 和 IMPLEMENT_META_INTERFACE 这两个宏都定义在刚才的 IInterface.h 中。先看 DECLARE_META_INTERFACE 这个宏，具体代码如下所示：

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

将 IServiceManager 的 DELCARE 宏进行相应的替换后，得到如下所示的代码：

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

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

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

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

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

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

可以直接将 IServiceManager 中 IMPLEMENT 宏的定义展开，具体代码如下所示：

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

```
}
//实现构造函数和析构函数
IServiceManager::IServiceManager () { }
IServiceManager::~ IServiceManager() { }
```

interface_cast 是如何把 BpBinder 指针转换成一个 IServiceManager 指针的呢？是通过 asInterface 函数的如下代码实现的：

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

由此可见，interface_cast 不是指针的转换，而是利用 BpBinder 对象作为参数新建了一个 BpServiceManager 对象。我们已经知道 BpBinder 和 BBinder 与通信有关系，这里怎么突然冒出来一个 BpServiceManager？它们之间又有什么关系呢？要搞清这个问题，必须先了解 IServiceManager 家族之间的关系，具体说明如下所示。

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

以上这些关系很复杂，但 ServiceManager 并没有使用错综复杂的派生关系，它直接打开 Binder 设备并与之交互。BpServiceManager 不像 BnServiceManager 那样与 Binder 有直接的血缘关系，那么，它又是如何与 Binder 交互的呢？简而言之，BpRefBase 中 mRemote 值就是 BpBinder。请看 BpServiceManager 左边派生分支树上的一系列代码，它们都在文件 IServiceManager.cpp 中实现，具体代码如下所示：

```
public:
    BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }

    virtual sp<IBinder> getService(const String16& name) const
    {
        unsigned n;
        for (n = 0; n < 5; n++){
            sp<IBinder> svc = checkService(name);
            if (svc != NULL) return svc;
            ALOGI("Waiting for service %s...\n", String8(name).string());
            sleep(1);
        }
        return NULL;
    }

    virtual sp<IBinder> checkService( const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        return reply.readStrongBinder();
    }

    virtual status_t addService(const String16& name, const sp<IBinder>& service,
            bool allowIsolated)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        data.writeStrongBinder(service);
        data.writeInt32(allowIsolated ? 1 : 0);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readExceptionCode() : err;
    }

    virtual Vector<String16> listServices()
    {
```

```
        Vector<String16> res;
        int n = 0;

        for (;;) {
            Parcel data, reply;
            data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
            data.writeInt32(n++);
            status_t err = remote()->transact(LIST_SERVICES_TRANSACTION, data, &reply);
            if (err != NO_ERROR)
                break;
            res.add(reply.readString16());
        }
        return res;
    }
};
```

BpInterface 在文件 IInterface.h 中定义，具体的实现代码如下所示：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
                            BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder*onAsBinder();
};
```

BpRefBase 在文件\frameworks\native\libs\binder\Binder.cpp 中定义，具体实现代码如下所示：

```
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);           // Removed on first IncStrong().
        mRefs = mRemote->createWeak(this);  // Held for our entire lifetime.
    }
}

BpRefBase::~BpRefBase()
{
    if (mRemote) {
        if (!(mState&kRemoteAcquired)) {
            mRemote->decStrong(this);
        }
        mRefs->decWeak(this);
    }
}
```

由此可见，是 BpServiceManager 的一个变量 mRemote 指向了 BpBinder。在函数 defaultService Manager 中有以下两个关键对象：

- BpBinder 对象，它的 handle 值是 0；
- BpServiceManager 对象，它的 mRemote 值是 BpBinder。

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

5.5.4　注册 MediaPlayerService

再回到 MediaServer 的 main 函数，下一个要分析的是 MediaPlayerService，此函数在如下所示的文件中实现：

```
frameworks\av\media\libmediaplayerservice\MediaPlayerService.cpp
```

MediaPlayerService 的具体实现代码如下所示：

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

根据前面的分析可知，defaultServiceManager()实际返回的对象是 BpServiceManager，它是 IServiceManager 的后代。函数 addService()在文件 IServiceManager.cpp 中实现，具体实现代码如下所示：

```
virtual status_t addService(const String16& name, const sp<IBinder>& service,
        bool allowIsolated)
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    data.writeStrongBinder(service);
    data.writeInt32(allowIsolated ? 1 : 0);
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

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

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

    return DEAD_OBJECT;
}
```

IPCThreadState 是进程中真正干活的“伙计”，在如下所示的文件中实现：

```
frameworks\native\libs\binder\IPCThreadState.cpp
```

IPCThreadState 的具体实现代码如下所示：

```
IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) {
restart:
        const pthread_key_t k = gTLS;
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
        if (st) return st;
        return new IPCThreadState;
    }

    if (gShutdown) return NULL;

    pthread_mutex_lock(&gTLSMutex);
    if (!gHaveTLS) {
        if (pthread_key_create(&gTLS, threadDestructor) != 0) {
            pthread_mutex_unlock(&gTLSMutex);
            return NULL;
        }
        gHaveTLS = true;
    }
    pthread_mutex_unlock(&gTLSMutex);
    goto restart;
}
```

接下来分析它的构造函数 IPCThreadState()，也在文件 IPCThreadState.cpp 中实现，具体实现

代码如下所示：

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(androidGetTid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}

IPCThreadState::~IPCThreadState()
{
}
```

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

传输工作是很辛苦的，BpBinder 的 transact 调用了 IPCThreadState 的 transact 函数，这个函数实际完成了与 Binder 通信的工作。函数 transact 在文件 IPCThreadState.cpp 中实现，具体实现代码如下所示：

```
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;

    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
```

```
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

接下来，看函数 writeTransactionData，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示：

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;
    tr.cookie = 0;
    tr.sender_pid = 0;
    tr.sender_euid = 0;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = statusBuffer;
        tr.offsets_size = 0;
        tr.data.ptr.offsets = NULL;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

由此可见，此函数的功能是把命令写到 mOut 中，而不是直接发出去。

现在，已经把 addService 的请求信息写到 mOut 中了。接下来再看发送请求和接收回复部分的实现，实现函数 waitForResponse 在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示：

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

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;
```

```
        cmd = mIn.readInt32();

        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }

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

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;

        case BR_ACQUIRE_RESULT:
            {
                ALOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
                const int32_t result = mIn.readInt32();
                if (!acquireResult) continue;
                *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
            }
            goto finish;

        case BR_REPLY:
            {
                binder_transaction_data tr;
                err = mIn.read(&tr, sizeof(tr));
                ALOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
                if (err != NO_ERROR) goto finish;

                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);
                    } else {
                        err = *static_cast<const status_t*>(tr.data.ptr.buffer);
                        freeBuffer(NULL,
                            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), this);
                    }
                } else {
                    freeBuffer(NULL,
                        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), this);
                    continue;
                }
            }
            goto finish;

        default:
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }
```

```
finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

这样便发送了请求数据，假设马上就收到了回复。再看函数 executeCommand，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示：

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

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                   "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
                   refs, obj, refs->refBase());
        obj->incStrong(mProcess.get());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
            obj->printRefs();
        }
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                   "BR_RELEASE: object %p does not match cookie %p (expected %p)",
                   refs, obj, refs->refBase());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
            obj->printRefs();
        }
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        mPendingWeakDerefs.push(refs);
        break;
```

```
case BR_ATTEMPT_ACQUIRE:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();

    {
        const bool success = refs->attemptIncStrong(mProcess.get());
        ALOG_ASSERT(success && refs->refBase() == obj,
                "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                refs, obj, refs->refBase());

        mOut.writeInt32(BC_ACQUIRE_RESULT);
        mOut.writeInt32((int32_t)success);
    }
    break;

case BR_TRANSACTION:
    {
        binder_transaction_data tr;
        result = mIn.read(&tr, sizeof(tr));
        ALOG_ASSERT(result == NO_ERROR,
            "Not enough command data for brTRANSACTION");
        if (result != NO_ERROR) break;

        Parcel buffer;
        buffer.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);

        const pid_t origPid = mCallingPid;
        const uid_t origUid = mCallingUid;

        mCallingPid = tr.sender_pid;
        mCallingUid = tr.sender_euid;

        int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
        if (gDisableBackgroundScheduling) {
            if (curPrio > ANDROID_PRIORITY_NORMAL) {
                setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
            }
        } else {
            if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                set_sched_policy(mMyThreadId, SP_BACKGROUND);
            }
        }
        Parcel reply;
        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_TRANSACTION thr " << (void*)pthread_self()
                << " / obj " << tr.target.ptr << " / code "
                << TypeCode(tr.code) << ": " << indent << buffer
                << dedent << endl
                << "Data addr = "
                << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                << ", offsets addr="
                << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
        }
        if (tr.target.ptr) {
            sp<BBinder> b((BBinder*)tr.cookie);
            const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
            if (error < NO_ERROR) reply.setError(error);

        } else {
            const status_t error = the_context_object->transact(tr.code, buffer,
            &reply, tr.flags);
            if (error < NO_ERROR) reply.setError(error);
        }

        if ((tr.flags & TF_ONE_WAY) == 0) {
            LOG_ONEWAY("Sending reply to %d!", mCallingPid);
```

```
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->getWeakRefs()->decWeak(proxy);
        } break;

    case BR_FINISHED:
        result = TIMED_OUT;
        break;

    case BR_NOOP:
        break;

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);
        break;

    default:
        printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
        result = UNKNOWN_ERROR;
        break;
    }

    if (result != NO_ERROR) {
        mLastError = result;
    }

    return result;
}
```

在和 Binder 设备进行交互时，是通过函数 write 和函数 read 来发送和接收请求实现的。接下来看函数 talkwithDriver，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示：

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }

    binder_write_read bwr;

    // Is the read buffer empty?
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();

    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;
```

```
    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();

    // This is what we'll read.
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
        bwr.read_buffer = (long unsigned int)mIn.data();
    } else {
        bwr.read_size = 0;
        bwr.read_buffer = 0;
    }

    IF_LOG_COMMANDS() {
        TextOutput::Bundle _b(alog);
        if (outAvail != 0) {
            alog << "Sending commands to driver: " << indent;
            const void* cmds = (const void*)bwr.write_buffer;
            const void* end = ((const uint8_t*)cmds)+bwr.write_size;
            alog << HexDump(cmds, bwr.write_size) << endl;
            while (cmds < end) cmds = printCommand(alog, cmds);
            alog << dedent;
        }
        alog << "Size of receive buffer: " << bwr.read_size
            << ", needRead: " << needRead << ", doReceive: " << doReceive << endl;
    }

    // Return immediately if there is nothing to do.
    if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

    bwr.write_consumed = 0;
    bwr.read_consumed = 0;
    status_t err;
    do {
        IF_LOG_COMMANDS() {
            alog << "About to read/write, write size = " << mOut.dataSize() << endl;
        }
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
#else
        err = INVALID_OPERATION;
#endif
        if (mProcess->mDriverFD <= 0) {
            err = -EBADF;
        }
        IF_LOG_COMMANDS() {
            alog << "Finished read/write, write size = " << mOut.dataSize() << endl;
        }
    } while (err == -EINTR);

    IF_LOG_COMMANDS() {
        alog << "Our err: " << (void*)err << ", write consumed: "
            << bwr.write_consumed << " (of " << mOut.dataSize()
            << "), read consumed: " << bwr.read_consumed << endl;
    }

    if (err >= NO_ERROR) {
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        IF_LOG_COMMANDS() {
            TextOutput::Bundle _b(alog);
```

```
            alog << "Remaining data size: " << mOut.dataSize() << endl;
            alog << "Received commands from driver: " << indent;
            const void* cmds = mIn.data();
            const void* end = mIn.data() + mIn.dataSize();
            alog << HexDump(cmds, mIn.dataSize()) << endl;
            while (cmds < end) cmds = printReturnCommand(alog, cmds);
            alog << dedent;
        }
        return NO_ERROR;
    }

    return err;
}
```

5.5.5 分析 StartThread Pool 和 join Thread Pool

接下来看最后两个函数 startThreadPool()和 joinThreadPool，其中函数 startThreadPool 在文件 ProcessState.cpp 中定义，具体实现代码如下所示：

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
```

在上述代码中，函数 spawnPooledThread()的实现如下所示：

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

PoolThread 是在 IPCThreadState 中定义的一个 Thread 子类，此类在文件 IPCThreadState.h 定义，具体实现代码如下所示：

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

接下来看 IPCThreadState 的 joinThreadPool 的实现，因为新创建的线程也会调用这个函数。此函数在文件 ProcessState.cpp 中定义，具体实现代码如下所示：

```
void IPCThreadState::joinThreadPool(bool isMain)
{
//如果 isMain 为 true 则需要循环处理。把请求信息写到 mOut 中，待会儿一起发出去
    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (void*)
pthread_self(), getpid());
```

```
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

    set_sched_policy(mMyThreadId, SP_FOREGROUND);

    status_t result;
    do {
        int32_t cmd;

        // When we've cleared the incoming command queue, process any pending derefs
        if (mIn.dataPosition() >= mIn.dataSize()) {
            size_t numPending = mPendingWeakDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                    refs->decWeak(mProcess.get());
                }
                mPendingWeakDerefs.clear();
            }
            //处理已经死亡的 BBinder 对象
            numPending = mPendingStrongDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    BBinder* obj = mPendingStrongDerefs[i];
                    obj->decStrong(mProcess.get());
                }
                mPendingStrongDerefs.clear();
            }
        }

        // now get the next command to be processed, waiting if necessary
        // 发送命令，读取请求
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
            IF_LOG_COMMANDS() {
                alog << "Processing top-level Command: "
                    << getReturnString(cmd) << endl;
            }


            result = executeCommand(cmd); //处理消息
        }
        set_sched_policy(mMyThreadId, SP_FOREGROUND);

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREAD POOL err=%p\n",
        (void*)pthread_self(), getpid(), (void*)result);

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

由此可见，这两个函数在 talkWithDriver 中被引用，它们希望能从 Binder 设备那里找到点可做的事情。在整个过程中到底有多少个线程在为 Service 服务呢？目前看来是两个：

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

在进程 MediaServer 中一共注册了 4 个服务，在业务繁忙的时候，两个线程会显得有点“力不从心”。另外，如果实现的服务负担不是很重，则完全可以不调用 startThreadPool 来创建新的线程，此时使用主线程即可完全胜任。

第 6 章　分析 Binder 对象和 Java 接口

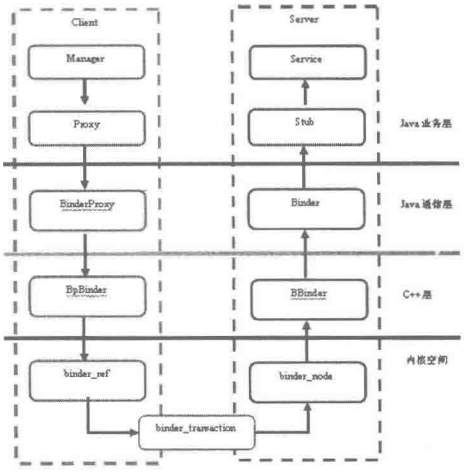
在 Android 5.0 系统源码中，Binder 机制在服务器端和客户端的通信过程中一共涉及了以下 4 类对象。

- 实体对象（binder_node）：位于驱动程序中。
- 引用对象（binder_ref）：位于驱动程序中。
- 本地对象（BBinder）：位于 Binder 库中。
- 代理对象（BpBinder）：位于 Binder 库中。

在本章的内容中，将详细分析 Android 中各种 Binder 对象的基本知识，分析各个对象的运作流程，并详细分析 Binder 机制中 Java 接口的基本知识。

6.1 分析实体对象（binder_node）

在 Android 系统中，binder_node 用来描述一个 Binder 实体对象。每个 Service 组件在 Binder 驱动程序中都对应有一个 Binder 实体对象，用来描述它在内核中的状态。Android 系统的 Binder 通信框架如图 6-1 所示。



▲图 6-1　Binder 通信框架图

Binder 实体对象 binder_node 的定义代码如下所示：

```
struct binder_node {
    //调试 id
    int debug_id;
    //描述一个待处理的工作项
    struct binder_work work;
    union {
        //挂载到宿主进程 binder_proc 的成员变量 nodes 红黑树的节点
        struct rb_node rb_node;
        //当宿主进程死亡，该 Binder 实体对象将挂载到全局 binder_dead_nodes 链表中
        struct hlist_node dead_node;
    };
    //指向该 Binder 线程的宿主进程
    struct binder_proc *proc;
    //保存所有引用该 Binder 实体对象的 Binder 引用对象
    struct hlist_head refs;
    //Binder 实体对象的强引用计数
    int internal_strong_refs;
    int local_strong_refs;
    unsigned has_strong_ref:1;
    unsigned pending_strong_ref:1;
    unsigned has_weak_ref:1;
    unsigned pending_weak_ref:1;
    //Binder 实体对象的弱引用计数
    int local_weak_refs;
    //指向用户空间 Service 组件内部的引用计数对象 wekref_impl 的地址
    void __user *ptr;
    //保存用户空间的 Service 组件地址
    void __user *cookie;
    //标示该 Binder 实体对象是否正在处理一个异步事务
    unsigned has_async_transaction:1;
```

```
    //设置该 Binder 实体对象是否可以接收包含有文件描述符的 IPC 数据
    unsigned accept_fds:1;
    //Binder 实体对象要求处理线程应具备的最小线程优先级
    unsigned min_priority:8;
    //异步事务队列
    struct list_head async_todo;
};
```

通过上述代码可知，在 Binder 驱动中，用户空间中的每一个 Binder 本地对象都对应有一个 Binder 实体对象。各个成员的具体说明如下所示。

- proc：指向 Binder 实体对象的宿主进程，宿主进程使用红黑树来维护它内部的所有 Binder 实体对象。
- rb_node：用来挂载到宿主进程 proc 的 Binder 实体对象红黑树中的节点。
- dead_node：如果该 Binder 实体对象的宿主进程已经死亡，该 Binder 实体就通过成员变量 dead_node 保存到全局链表 binder_dead_nodes。
- refs：一个 Binder 实体对象可以被多个 client 引用，成员变量 refs 用来保存所有引用该 Binder 实体的 Binder 引用对象。
- internal_strong_refs 和 local_strong_refs：都是用来描述 Binder 实体对象的强引用计数。
- local_weak_refs：用来描述 Binder 实体对象的弱引用计数。
- ptr 和 cookie：分别指向用户空间地址，cookie 指向 BBinder 的地址，ptr 指向 BBinder 对象的引用计数地址。
- has_async_transaction：描述一个 Binder 实体对象是否正在处理一个异步事务，当 Binder 驱动指定某个线程来处理某一事务时，首先将该事务保存到指定线程的 todo 队列中，表示要由该线程来处理该事务。如果是异步事务，Binder 驱动程序就会将它保存在目标 Binder 实体对象的一个异步事务 async_todo 队列中。
- min_priority：表示一个 Binder 实体对象在处理来自 client 进程请求时，要求处理线程的最小线程优先级。

在 Binder 驱动程序中，使用函数 binder_inc_node 来增加一个 Binder 实体对象的引用计数。函数 binder_inc_node 在文件\drivers\staging\android\binder.c 中定义，具体实现代码如下所示：

```
static int binder_inc_node(struct binder_node *node, int strong, int internal,
                           struct list_head *target_list)
{
        if (strong) {
                if (internal) {
                        if (target_list == NULL &&
                            node->internal_strong_refs == 0 &&
                            !(node == binder_context_mgr_node &&
                            node->has_strong_ref)) {
                                printk(KERN_ERR "binder: invalid inc strong "
                                        "node for %d\n", node->debug_id);
                                return -EINVAL;
                        }
                        node->internal_strong_refs++;
                } else
                        node->local_strong_refs++;
                if (!node->has_strong_ref && target_list) {
                        list_del_init(&node->work.entry);
                        list_add_tail(&node->work.entry, target_list);
                }
        } else {
                if (!internal)
                        node->local_weak_refs++;
                if (!node->has_weak_ref && list_empty(&node->work.entry)) {
                        if (target_list == NULL) {
                                printk(KERN_ERR "binder: invalid inc weak node "
                                        "for %d\n", node->debug_id);
                                return -EINVAL;
                        }
                        list_add_tail(&node->work.entry, target_list);
                }
```

```
        }
        return 0;
}
```

函数 binder_inc_node 中各个参数的具体说明如下所示。

- node：表示要增加引用计数的 Binder 实体对象。
- strong：表示要增加强引用计数还是要增加弱用计数。
- internal：用于区分增加的是内部引用计数还是外部引用计数。
- target_list：用于指向一个目标进程或目标线程的 todo 队列，当不是 null 值时，表示增加了 Binder 实体对象的引用计数后，需要对应增加它所引用的 Binder 本地对象的引用计数。

与函数 binder_inc_node 相反，在 Binder 驱动程序中，使用函数 binder_dec_node 来减少一个 Binder 实体对象的引用计数。函数 binder_dec_node 会减少 internal_strong_refs、local_strong_refs 或 local_weak_refs 的使用计数，并删除节点的 work.entry 链表。函数 binder_dec_node 也是在文件 \drivers\staging\android\binder.c 中定义，具体实现代码如下所示：

```
static int binder_dec_node(struct binder_node *node, int strong, int internal)
{
        if (strong) {
                if (internal)
                        node->internal_strong_refs--;
                else
                        node->local_strong_refs--;
                if (node->local_strong_refs || node->internal_strong_refs)
                        return 0;
        } else {
                if (!internal)
                        node->local_weak_refs--;
                if (node->local_weak_refs || !hlist_empty(&node->refs))
                        return 0;
        }
        if (node->proc && (node->has_strong_ref || node->has_weak_ref)) {
                if (list_empty(&node->work.entry)) {
                        list_add_tail(&node->work.entry, &node->proc->todo);
                        wake_up_interruptible(&node->proc->wait);
                }
        } else {
                if (hlist_empty(&node->refs) && !node->local_strong_refs &&
                    !node->local_weak_refs) {
                        list_del_init(&node->work.entry);
                        if (node->proc) {
                                rb_erase(&node->rb_node, &node->proc->nodes);
                                binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                                             "binder: refless node %d deleted\n",
                                             node->debug_id);
                        } else {
                                hlist_del(&node->dead_node);
                                binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                                             "binder: dead node %d deleted\n",
                                             node->debug_id);
                        }
                        kfree(node);
                        binder_stats_deleted(BINDER_STAT_NODE);
                }
        }
        return 0;
}
```

6.2 分析本地对象（BBinder）

因为 Binder 的功能就是在本地执行其他进程的功能，所以对于 Binder 机制来说，不但是 Android 系统中的一个完美的 IPC 机制，其实也是 Android 的一种远程过程调用（RPC）机制。当进程通过 Binder 获取将要调用的进程服务时，不但可以是一个本地对象，而且也可以是一个远程

服务的引用。也就是说，Binder 不但可以与本地进程通信，而且还可以与远程进程通信。此处的本地进程就是本章所讲解的本地对象，而远程进程就是远程服务的一个引用。

由此可见，无论这个对象是本地的还是远程的，Binder 的实质是将对象从一个进程映射到另一个进程中。如果是远程对象，将远程对象的引用从一个进程映射到另一个进程中。当使用这个远程对象时，实际上就是使用远程对象在本地的一个引用，类似于把这个远程对象当作一个本地对象在使用。这也就是 Binder 与其他 IPC 机制不同的地方。

在 Android 系统中，Binder 驱动程序通过函数 binder_thread_read 引用了运行在 Server 进程中的 Binder 本地对象，此函数在文件 drivers\staging\android\binder.c 中定义。当 service_manager 运行时此函数会一直等待，直到有请求到达为止。函数 binder_thread_read 的具体实现代码如下所示：

```
static int binder_thread_read(struct binder_proc *proc,
                              struct binder_thread *thread,
                              void __user *buffer, int size,
                              signed long *consumed, int non_block)
{
        void __user *ptr = buffer + *consumed;
        void __user *end = buffer + size;
        int ret = 0;
        int wait_for_proc_work;
        if (*consumed == 0) {
                if (put_user(BR_NOOP, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
        }
retry:
        wait_for_proc_work = thread->transaction_stack == NULL &&
                             list_empty(&thread->todo);
        if (thread->return_error != BR_OK && ptr < end) {
                if (thread->return_error2 != BR_OK) {
                        if (put_user(thread->return_error2, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, thread->return_error2);
                        if (ptr == end)
                                goto done;
                        thread->return_error2 = BR_OK;
                }
                if (put_user(thread->return_error, (uint32_t __user *)ptr))
                        return -EFAULT;
                ptr += sizeof(uint32_t);
                binder_stat_br(proc, thread, thread->return_error);
                thread->return_error = BR_OK;
                goto done;
        }
        thread->looper |= BINDER_LOOPER_STATE_WAITING;
        if (wait_for_proc_work)
                proc->ready_threads++;
        binder_unlock(__func__);
        trace_binder_wait_for_work(wait_for_proc_work,
                                   !!thread->transaction_stack,
                                   !list_empty(&thread->todo));
        if (wait_for_proc_work) {
                if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
                                        BINDER_LOOPER_STATE_ENTERED))) {
                        binder_user_error("binder: %d:%d ERROR: Thread waiting "
                                "for process work before calling BC_REGISTER_"
                                "LOOPER or BC_ENTER_LOOPER (state %x)\n",
                                proc->pid, thread->pid, thread->looper);
                        wait_event_interruptible(binder_user_error_wait,
                                                 binder_stop_on_user_error < 2);
                }
                binder_set_nice(proc->default_priority);
                if (non_block) {
                        if (!binder_has_proc_work(proc, thread))
                                ret = -EAGAIN;
                } else
                        ret = wait_event_freezable_exclusive(proc->wait, binder_has_proc_
```

```
                work(proc, thread));
        } else {
                if (non_block) {
                        if (!binder_has_thread_work(thread))
                                ret = -EAGAIN;
                } else
                        ret=wait_event_freezable(thread->wait,binder_has_thread_work (thread));
        }
        binder_lock(__func__);
        if (wait_for_proc_work)
                proc->ready_threads--;
        thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
        if (ret)
                return ret;
        while (1) {
                uint32_t cmd;
//将用户传进来的transact参数复制在本地变量struct binder_transaction_data tr中
                struct binder_transaction_data tr;
                struct binder_work *w;
                struct binder_transaction *t = NULL;
                //由于thread→todo不为空，执行下列语句
                if (!list_empty(&thread->todo))
                        w = list_first_entry(&thread->todo, struct binder_work, entry);
                else if (!list_empty(&proc->todo) && wait_for_proc_work)
//Service Manager被唤醒之后，就进入while循环开始处理事务了
//此处wait_for_proc_work等于1，并且proc→todo不为空，所以从proc→todo列表中得到第一个工作项
                        w = list_first_entry(&proc->todo, struct binder_work, entry);
                else {
                        if (ptr - buffer == 4 && !(thread->looper & BINDER_LOOPER_STATE_
NEED_RETURN)) /* no data added */
                                goto retry;
                        break;
                }
                if (end - ptr < sizeof(tr) + 4)
                        break;
                switch (w->type) {
//函数调用binder_transaction进一步处理
                case BINDER_WORK_TRANSACTION: {
//因为这个工作项的类型为BINDER_WORK_TRANSACTION，所以通过下面语句得到事务项
                        t = container_of(w, struct binder_transaction, work);
                } break;
//因为w→type为BINDER_WORK_TRANSACTION_COMPLETE
//这是在上面的binder_transaction函数设置的，于是执行下面的代码
                case BINDER_WORK_TRANSACTION_COMPLETE: {
                        cmd = BR_TRANSACTION_COMPLETE;
                        if (put_user(cmd, (uint32_t __user *)ptr))
                                return -EFAULT;
                        ptr += sizeof(uint32_t);
                        binder_stat_br(proc, thread, cmd);
                        binder_debug(BINDER_DEBUG_TRANSACTION_COMPLETE,
                                     "binder: %d:%d BR_TRANSACTION_COMPLETE\n",
                                     proc->pid, thread->pid);
                                     //将w从thread→todo删除
                        list_del(&w->entry);
                        kfree(w);
                        binder_stats_deleted(BINDER_STAT_TRANSACTION_COMPLETE);
                } break;
                case BINDER_WORK_NODE: {
                        struct binder_node *node = container_of(w, struct binder_node, work);
                        uint32_t cmd = BR_NOOP;
                        const char *cmd_name;
                        int strong = node->internal_strong_refs || node->local_strong_refs;
                        int weak=!hlist_empty(&node->refs)|| node->local_weak_refs|| strong;
                        if (weak && !node->has_weak_ref) {
                                cmd = BR_INCREFS;
                                cmd_name = "BR_INCREFS";
                                node->has_weak_ref = 1;
                                node->pending_weak_ref = 1;
                                node->local_weak_refs++;
                        } else if (strong && !node->has_strong_ref) {
                                cmd = BR_ACQUIRE;
```

```
                cmd_name = "BR_ACQUIRE";
                node->has_strong_ref = 1;
                node->pending_strong_ref = 1;
                node->local_strong_refs++;
            } else if (!strong && node->has_strong_ref) {
                cmd = BR_RELEASE;
                cmd_name = "BR_RELEASE";
                node->has_strong_ref = 0;
            } else if (!weak && node->has_weak_ref) {
                cmd = BR_DECREFS;
                cmd_name = "BR_DECREFS";
                node->has_weak_ref = 0;
            }
            if (cmd != BR_NOOP) {
                if (put_user(cmd, (uint32_t __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(uint32_t);
                if (put_user(node->ptr, (void * __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(void *);
                if (put_user(node->cookie, (void * __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(void *);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_USER_REFS,
                     "binder: %d:%d %s %d u%p c%p\n",
                     proc->pid, thread->pid, cmd_name, node->debug_id,
                     node->ptr, node->cookie);
            } else {
                list_del_init(&w->entry);
                if (!weak && !strong) {
                    binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                           "binder: %d:%d node %d u%p c%p deleted\n",
                           proc->pid, thread->pid, node->debug_id,
                           node->ptr, node->cookie);
                    rb_erase(&node->rb_node, &proc->nodes);
                    kfree(node);
                    binder_stats_deleted(BINDER_STAT_NODE);
                } else {
                    binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                          "binder: %d:%d node %d u%p c%p state unchanged\n",
                          proc->pid, thread->pid, node->debug_id, node->ptr,
                          node->cookie);
                }
            }
        } break;
        case BINDER_WORK_DEAD_BINDER:
        case BINDER_WORK_DEAD_BINDER_AND_CLEAR:
        case BINDER_WORK_CLEAR_DEATH_NOTIFICATION: {
            struct binder_ref_death *death;
            uint32_t cmd;
            death = container_of(w, struct binder_ref_death, work);
            if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION)
                cmd = BR_CLEAR_DEATH_NOTIFICATION_DONE;
            else
                cmd = BR_DEAD_BINDER;
            if (put_user(cmd, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            if (put_user(death->cookie, (void * __user *)ptr))
                return -EFAULT;
            ptr += sizeof(void *);
            binder_stat_br(proc, thread, cmd);
            binder_debug(BINDER_DEBUG_DEATH_NOTIFICATION,
                         "binder: %d:%d %s %p\n",
                         proc->pid, thread->pid,
                         cmd == BR_DEAD_BINDER ?
                         "BR_DEAD_BINDER" :
                         "BR_CLEAR_DEATH_NOTIFICATION_DONE",
                         death->cookie);
            if (w->type == BINDER_WORK_CLEAR_DEATH_NOTIFICATION) {
```

```
                        list_del(&w->entry);
                        kfree(death);
                        binder_stats_deleted(BINDER_STAT_DEATH);
                    } else
                        list_move(&w->entry, &proc->delivered_death);
                    if (cmd == BR_DEAD_BINDER)
                        goto done;/*DEAD_BINDER notifications can cause transactions */
                } break;
                }
                if (!t)
                    continue;
                BUG_ON(t->buffer == NULL);
//把事务项 t 中的数据复制到本地局部变量 struct binder_transaction_data tr 中
                if (t->buffer->target_node) {
                    struct binder_node *target_node = t->buffer->target_node;
                    tr.target.ptr = target_node->ptr;
                    tr.cookie =  target_node->cookie;
                    t->saved_priority = task_nice(current);
                    if (t->priority < target_node->min_priority &&
                        !(t->flags & TF_ONE_WAY))
                        binder_set_nice(t->priority);
                    else if (!(t->flags & TF_ONE_WAY) ||
                             t->saved_priority > target_node->min_priority)
                        binder_set_nice(target_node->min_priority);
                    cmd = BR_TRANSACTION;
                } else {
                    tr.target.ptr = NULL;
                    tr.cookie = NULL;
                    cmd = BR_REPLY;
                }
                tr.code = t->code;
                tr.flags = t->flags;
                tr.sender_euid = t->sender_euid;
                if (t->from) {
                    struct task_struct *sender = t->from->proc->tsk;
                    tr.sender_pid = task_tgid_nr_ns(sender,
                                                    current->nsproxy->pid_ns);
                } else {
                    tr.sender_pid = 0;
                }
                tr.data_size = t->buffer->data_size;
                tr.offsets_size = t->buffer->offsets_size;
//t→buffer→data 所指向的地址是内核空间的，如果要把数据返回给 Service Manager 进程的用户空间
//而 Service Manager 进程的用户空间是不能访问内核空间的数据的，所以需要进一步处理
//在具体处理时，Binder 机制使用类似浅拷贝的方法，通过在用户空间分配一个虚拟地址
//然后让这个用户空间虚拟地址与 t→buffer→data 这个内核空间虚拟地址指向同一个物理地址
//在此只需将 t→buffer→data 加上一个偏移值 proc→user_buffer_offset
//就可以得到 t→buffer→data 对应的用户空间虚拟地址了
//在调整了 tr.data.ptr.buffer 值后，需要一起调整 tr.data.ptr.offsets 的值
                tr.data.ptr.buffer = (void *)t→buffer→data +
                                     proc→user_buffer_offset;
                tr.data.ptr.offsets = tr.data.ptr.buffer +
                                      ALIGN(t→buffer→data_size,
                                        sizeof(void *));
//把 tr 的内容复制到用户传进来的缓冲区去，指针 ptr 指向这个用户缓冲区的地址
                if (put_user(cmd, (uint32_t __user *)ptr))
                    return -EFAULT;
                ptr += sizeof(uint32_t);
                if (copy_to_user(ptr, &tr, sizeof(tr)))
                    return -EFAULT;
                ptr += sizeof(tr);
//上述代码只是对 tr.data.ptr.bufferr 和 tr.data.ptr.offsets 的内容进行了浅拷贝工作
                trace_binder_transaction_received(t);
                binder_stat_br(proc, thread, cmd);
                binder_debug(BINDER_DEBUG_TRANSACTION,
                             "binder: %d:%d %s %d %d:%d, cmd %d"
                             "size %zd-%zd ptr %p-%p\n",
                             proc->pid, thread->pid,
                             (cmd == BR_TRANSACTION) ? "BR_TRANSACTION" :
                             "BR_REPLY",
                             t->debug_id, t->from ? t->from->proc->pid : 0,
```

```
                        t->from ? t->from->pid : 0, cmd,
                        t->buffer->data_size, t->buffer->offsets_size,
                        tr.data.ptr.buffer, tr.data.ptr.offsets);
//从 todo 列表中删除，因为已经处理了这个事务
                list_del(&t->work.entry);
                t->buffer->allow_user_free = 1;
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 true
//说明虽然在驱动程序中已经处理完了这个事务，但是仍然要在 Service Manager 完成之后需要等待回复确认
                if (cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)) {
//把当前事务 t 放在 thread->transaction_stack 队列的头部
                    t->to_parent = thread->transaction_stack;
                    t->to_thread = thread;
                    thread->transaction_stack = t;
                }
//如果 cmd == BR_TRANSACTION && !(t->flags & TF_ONE_WAY)为 false
//则不需要等待回复，而是直接删掉事务 t
                else {
                    t->buffer->transaction = NULL;
                    kfree(t);
                    binder_stats_deleted(BINDER_STAT_TRANSACTION);
                }
                break;
        }
done:
        *consumed = ptr - buffer;
        if (proc->requested_threads + proc->ready_threads == 0 &&
            proc->requested_threads_started < proc->max_threads &&
            (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
             BINDER_LOOPER_STATE_ENTERED)) /* the user-space code fails to */
             /*spawn a new thread if we leave this out */) {
                proc->requested_threads++;
                binder_debug(BINDER_DEBUG_THREADS,
                             "binder: %d:%d BR_SPAWN_LOOPER\n",
                             proc->pid, thread->pid);
                if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer))
                        return -EFAULT;
                binder_stat_br(proc, thread, BR_SPAWN_LOOPER);
        }
        return 0;
}
```

由此可见，Binder 驱动程序在引用运行在 Server 进程中的 Binder 本地对象时，是通过以下 4 个协议实现的：

- BR_INCREFS；
- BR_ACQUIRE；
- BR_DECREFS；
- BR_RELEASE。

在文件 frameworks\native\libs\binder\IPCThreadState.cpp 中，通过使用类成员函数 executeCommand 来处理上述 4 个接口协议。函数 executeCommand 的具体实现代码如下所示：

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

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
```

```
            "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
            refs, obj, refs->refBase());
    obj->incStrong(mProcess.get());
    IF_LOG_REMOTEREFS() {
        LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
        obj->printRefs();
    }
    mOut.writeInt32(BC_ACQUIRE_DONE);
    mOut.writeInt32((int32_t)refs);
    mOut.writeInt32((int32_t)obj);
    break;

case BR_RELEASE:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();
    ALOG_ASSERT(refs->refBase() == obj,
            "BR_RELEASE: object %p does not match cookie %p (expected %p)",
            refs, obj, refs->refBase());
    IF_LOG_REMOTEREFS() {
        LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
        obj->printRefs();
    }
    mPendingStrongDerefs.push(obj);
    break;

case BR_INCREFS:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();
    refs->incWeak(mProcess.get());
    mOut.writeInt32(BC_INCREFS_DONE);
    mOut.writeInt32((int32_t)refs);
    mOut.writeInt32((int32_t)obj);
    break;

case BR_DECREFS:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();
    mPendingWeakDerefs.push(refs);
    break;

case BR_ATTEMPT_ACQUIRE:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();

    {
        const bool success = refs->attemptIncStrong(mProcess.get());
        ALOG_ASSERT(success && refs->refBase() == obj,
                "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                refs, obj, refs->refBase());

        mOut.writeInt32(BC_ACQUIRE_RESULT);
        mOut.writeInt32((int32_t)success);
    }
    break;

case BR_TRANSACTION:
    {
        binder_transaction_data tr;
        result = mIn.read(&tr, sizeof(tr));
        ALOG_ASSERT(result == NO_ERROR,
            "Not enough command data for brTRANSACTION");
        if (result != NO_ERROR) break;

        Parcel buffer;
        buffer.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);

        const pid_t origPid = mCallingPid;
```

```
            const uid_t origUid = mCallingUid;

            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;

            int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
            if (gDisableBackgroundScheduling) {
                if (curPrio > ANDROID_PRIORITY_NORMAL) {
                    setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
                }
            } else {
                if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                    set_sched_policy(mMyThreadId, SP_BACKGROUND);
                }
            }
            Parcel reply;
            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BR_TRANSACTION thr " << (void*)pthread_self()
                    << " / obj " << tr.target.ptr << " / code "
                    << TypeCode(tr.code) << ": " << indent << buffer
                    << dedent << endl
                    << "Data addr = "
                    << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                    << ", offsets addr="
                    << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
            }
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer,
                &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);
            }

            //ALOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
            //     mCallingPid, origPid, origUid);

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
```

```
            proxy->getWeakRefs()->decWeak(proxy);
        } break;

    case BR_FINISHED:
        result = TIMED_OUT;
        break;

    case BR_NOOP:
        break;

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);
        break;

    default:
        printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
        result = UNKNOWN_ERROR;
        break;
    }
    if (result != NO_ERROR) {
        mLastError = result;
    }
    return result;
}
```

通过上述代码可知，在函数 executeCommand 中会调用 BBinder::transact 来处理 Client 端的请求。当需要多个线程提供服务时，驱动会请求创建新线程。具体创建某线程的过程，可以通过上述函数处理 BR_SPAWN_LOOPER 协议的过程获得。

6.3 分析引用对象（binder_ref）

在 Android 5.0 系统中，引用对象的类型为 binder_ref，定义结构体 binder_ref 的代码如下所示：

```
struct binder_ref {
    //调试 id
    int debug_id;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_desc 中的节点
    struct rb_node rb_node_desc;
    //挂载到宿主对象 binder_proc 的红黑树 refs_by_node 中的节点
    struct rb_node rb_node_node;
    //挂载到 Binder 实体对象的 refs 链表中的节点
    struct hlist_node node_entry;
    //Binder 引用对象的宿主进程 binder_proc
    struct binder_proc *proc;
    //Binder 引用对象所引用的 Binder 实体对象
    struct binder_node *node;
    //Binder 引用对象的句柄值
    uint32_t desc;
    //强引用计数
    int strong;
    //弱引用计数
    int weak;
    //注册死亡接收通知
    struct binder_ref_death *death;
};
```

在 Binder 机制中，binder_ref 用来描述一个 Binder 引用对象，每一个 Client 在 Binder 驱动中都有一个 Binder 引用对象。各个成员变量的具体说明如下所示。

- 成员变量 node：保存了该 Binder 引用对象所引用的 Binder 实体对象，Binder 实体对象使用链表保存了所有引用该实体对象的 Binder 引用对象。
- node_entry：是该 Binder 引用对象所引用的 Binder 实体对象的成员变量 refs 链表中的节点。
- desc：是一个句柄值，用来描述一个 Binder 引用对象。
- node：当 Client 进程通过句柄值来访问某个 Service 服务时，Binder 驱动程序可以通过该句柄值找到对应的 Binder 引用对象，然后根据该 Binder 引用对象的成员变量 node 找到对应的

Binder 实体对象，最后通过该 Binder 实体对象找到要访问的 Service。

- proc：执行该 Binder 引用对象的宿主进程。
- rb_node_desc 和 rb_node_node：是 binder_proc 中红黑树 refs_by_desc 和 refs_by_node 的节点。

Binder 驱动程序存在如下 4 个重要的协议，用于增加和减少 Binder 引用对象的强引用计数和弱引用计数：

- BR_INCREFS；
- BR_ACQUIRE；
- BR_DECREFS；
- BR_RELEASE。

上述计数处理功能通过函数 binder_thread_write 实现，此函数在文件\drivers\staging\android\binder.c 中定义，此函数已经在本章前面的内容中进行了详细分析。

协议 BR_INCREFS 和 BR_ACQUIRE 用于增加一个 Binder 引用对象的强引用计数和弱引用计数，此功能是通过调用函数 binder_inc_ref 实现的，具体实现代码如下所示：

```
static int binder_inc_ref(struct binder_ref *ref, int strong,
                          struct list_head *target_list)
{
        int ret;
        if (strong) {
                if (ref->strong == 0) {
                        ret = binder_inc_node(ref->node, 1, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->strong++;
        } else {
                if (ref->weak == 0) {
                        ret = binder_inc_node(ref->node, 0, 1, target_list);
                        if (ret)
                                return ret;
                }
                ref->weak++;
        }
        return 0;
}
```

而协议 BR_RELEASE 和 BR_DECREFS 用于减少一个 Binder 引用对象的强引用计数和弱引用计数，此功能是通过调用函数 binder_dec_ref 定义的，具体实现代码如下所示：

```
static int binder_dec_ref(struct binder_ref *ref, int strong)
{
        if (strong) {
                if (ref->strong == 0) {
                        binder_user_error("binder: %d invalid dec strong, "
                                          "ref %d desc %d s %d w %d\n",
                                          ref->proc->pid, ref->debug_id,
                                          ref->desc, ref->strong, ref->weak);
                        return -EINVAL;
                }
                ref->strong--;
                if (ref->strong == 0) {
                        int ret;
                        ret = binder_dec_node(ref->node, strong, 1);
                        if (ret)
                                return ret;
                }
        } else {
                if (ref->weak == 0) {
                        binder_user_error("binder: %d invalid dec weak, "
                                          "ref %d desc %d s %d w %d\n",
                                          ref->proc->pid, ref->debug_id,
                                          ref->desc, ref->strong, ref->weak);
                        return -EINVAL;
                }
                ref->weak--;
```

```
        }
        if (ref->strong == 0 && ref->weak == 0)
                binder_delete_ref(ref);
        return 0;
}
```

再看函数 binder_delete_ref，功能是销毁 binder_ref 对象，具体实现代码如下所示：

```
static void binder_delete_ref(struct binder_ref *ref)
{
        binder_debug(BINDER_DEBUG_INTERNAL_REFS,
                     "binder: %d delete ref %d desc %d for "
                     "node %d\n", ref->proc->pid, ref->debug_id,
                     ref->desc, ref->node->debug_id);

        rb_erase(&ref->rb_node_desc, &ref->proc->refs_by_desc);
        rb_erase(&ref->rb_node_node, &ref->proc->refs_by_node);
        if (ref->strong)
                binder_dec_node(ref->node, 1, 1);
        hlist_del(&ref->node_entry);
        binder_dec_node(ref->node, 0, 1);
        if (ref->death) {
                binder_debug(BINDER_DEBUG_DEAD_BINDER,
                             "binder: %d delete ref %d desc %d "
                             "has death notification\n", ref->proc->pid,
                             ref->debug_id, ref->desc);
                list_del(&ref->death->work.entry);
                kfree(ref->death);
                binder_stats_deleted(BINDER_STAT_DEATH);
        }
        kfree(ref);
        binder_stats_deleted(BINDER_STAT_REF);
}
```

6.4 分析代理对象（BpBinder）

在 Android 系统中，代理对象 BpBinder 是远程对象在当前进程的代理，它实现了 IBinder 接口。对于初学者来说，BBinder 与 BpBinder 这两者容易混淆。其实这两者很好区分，对于 Service 来说继承了 BBinder（BnInterface），因为 BBinder 有 onTransact 消息处理函数。而对于与 Service 通信的 Client 来说，需要继承 BpBinder(BpInterface)，因为 BpBinder 有消息传递函数 transcat。以 cameraService 的 Client 为例，文件 Camera.cpp 中的函数 getCameraService 能够获取远程 CameraService 的 IBinder 对象，然后通过如下代码进行了重构，得到了 BpCameraService 对象：

```
mCameraService = interface_cast<ICameraService>(binder);
```

而 BpCameraService 继承了 BpInterface，并传入了 BBinder：

```
cameraService:
   defaultServiceManager()->addService(
          String16("media.camera"), new CameraService());
```

在 IPC 传递的过程中，IBinder 指针不可缺少。这个指针对一个进程来说就像是 Socket 的 ID 一样，是唯一的。无论这个 IBinder 是 BBinder 还是 BpBinder，它们都是在重构 BpBinder 或者 BBinder 时把 IBinder 作为参数传入。

在 Android 系统中，创建 Binder 代理对象在文件 frameworks\native\libs\binder\BpBinder.cpp 中定义，具体实现代码如下所示：

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);
//设置 Binder 代理对象的生命周期，接收弱引用计数响应
```

```
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
//调用当前线程内部的 IPCThreadState 的成员函数 incWeakHandle 增加相应的 Binder 引用对象的弱引用计数
    IPCThreadState::self()->incWeakHandle(handle);
}
```

在文件 frameworks\native\libs\binder\IPCThreadState.cpp 中，定义成员函数 incWeakHandle 的代码如下所示：

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_INCREFS);
    mOut.writeInt32(handle);
}
```

当销毁一个 Binder 代理对象时，线程会调用内部的 IPCThreadState 对象的成员函数 decWeakHandle 来减少相应的 Binder 引用对象的弱引用计数。函数 decWeakHandle 的具体实现代码如下所示：

```
void IPCThreadState::decWeakHandle(int32_t handle)
{
    LOG_REMOTEREFS("IPCThreadState::decWeakHandle(%d)\n", handle);
    mOut.writeInt32(BC_DECREFS);
    mOut.writeInt32(handle);
}
```

在 Binder 代理对象中，函数 transact 的实现代码如下所示：

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }
    return DEAD_OBJECT;
}
```

函数 transact 各个参数的具体说明如下所示。

- code：表示请求的 ID 号。
- data：表示请求的参数。
- reply：表示返回的结果。
- flags：表示一些额外的标识，如 FLAG_ONEWAY，通常为 0。

上述函数 transact 只是简单地调用了 IPCThreadState::self()的 transact，在 IPCThreadState::transact 中的定义代码如下所示：

```
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;

    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
```

```
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}

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

    while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;
        err = mIn.errorCheck();
        if (err < NO_ERROR) break;
        if (mIn.dataAvail() == 0) continue;

        cmd = mIn.readInt32();

        IF_LOG_COMMANDS() {
            alog << "Processing waitForResponse Command: "
                << getReturnString(cmd) << endl;
        }

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

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;

        case BR_ACQUIRE_RESULT:
            {
                LOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
                const int32_t result = mIn.readInt32();
                if (!acquireResult) continue;
                *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
            }
            goto finish;

        case BR_REPLY:
            {
```

```
                binder_transaction_data tr;
                err = mIn.read(&tr, sizeof(tr));
                LOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
                if (err != NO_ERROR) goto finish;

                if (reply) {
                    if ((tr.flags & TF_STATUS_CODE) == 0) {
                        reply->ipcSetDataReference(
                            reinterpret_cast(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t),
                            freeBuffer, this);
                    } else {
                        err = *static_cast(tr.data.ptr.buffer);
                        freeBuffer(NULL,
                            reinterpret_cast(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t), this);
                    }
                } else {
                    freeBuffer(NULL,
                        reinterpret_cast(tr.data.ptr.buffer),
                        tr.data_size,
                        reinterpret_cast(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t), this);
                    continue;
                }
            }
            goto finish;

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

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

在上述代码中，transact 将请求经过内核模块发送给了服务端。当服务端处理完请求之后，会沿原路返回结果给调用者。在此可以看出请求是同步操作，它会等待直到结果返回为止。这样在 BpBinder 之上进行简单包装之后，就可以得到与服务对象相同的接口，调用者不用关心调用的对象是远程的还是本地的。

6.5 分析 Java 接口

在本书前面的内中，已经详细分析了 Android 系统中 Binder 通信机制的 Binder 运行库和驱动程序的源代码，这些源代码都是基于 C/C++语言实现的。在本节的内容中，将详细讲解在 Android 系统应用程序框架层中，使用 Java 语言实现的 Binder 接口的基本知识。

6.5.1 获取 Service Manager

在 Binder 通信机制中需要通过 Java 远程接口获取 Service Manager，这个 Java 接口是一个 ServiceManagerProxy 对象的 IServiceManager 接口。类 ServiceManagerProxy 实现了 IServiceManager

接口。在类 IServiceManager 中提供了如下两个成员函数来管理系统：

- ServicegetService；
- addService。

因为类 ServiceManagerProxy 需要一个 BinderProxy 对象的 IBinder 接口来作为参数，所以要想获取 Service Manager 的 Java 远程接口 ServiceManagerProxy，前提是拥有一个 BinderProxy 对象。在类 ServiceManager 中有一个名为 getIServiceManager 的静态成员函数，此函数的功能是获取 Service Manager 的 Java 远程接口，并通过 ServiceManagerNative 来获取 Service Manager 的 Java 远程接口。

函数 getIServiceManager 在文件 frameworks\base\core\java\android\os\ServiceManager.java 中定义，具体实现代码如下所示：

```
private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }
    // Find the service manager
    sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
    return sServiceManager;
}
```

在调用函数 ServiceManagerNative.asInterface 之前，先通过函数 BinderInternal.getContextObject 来获得一个 BinderProxy 对象。在上述代码中，如果尚未创建静态成员变量 sServiceManager，那么就调用函数 ServiceManagerNative.asInterface 来创建。

函数 getContextObject 在文件 frameworks\base\core\java\com\android\internal\os\BinderInternal.java 中定义，具体实现代码如下所示：

```
public static final native IBinder getContextObject();
```

由上述实现代码可知，函数 BinderInternal.getContextObject 只是一个 JNI 方法，在文件 frameworks\base\core\jni\android_util_Binder.cpp 中定义，具体实现代码如下所示：

```
static jobject android_os_BinderInternal_getContextObject(JNIEnv* env, jobject clazz)
{
//返回一个 BpBinder 对象，句柄值是 0
    sp<IBinder> b = ProcessState::self()->getContextObject(NULL);
//调用 javaObjectForIBinder 将此 BpBinder 对象转换为 BinderProxy 对象
    return javaObjectForIBinder(env, b);
}
```

在上述代码中，调用了 BpBinder 对象 javaObjectForIBinder，javaObjectForIBinder 在文件 frameworks\base\core\jni\ android_util_Binder.cpp 中定义，具体实现代码如下所示：

```
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
    if (val == NULL) return NULL;
//传进来的参数是一个 BpBinder 的指针，而 BpBinder::checkSubclass 继承于父类 IBinder::check
Subclass,
//它什么也不做就返回 false
    if (val->checkSubclass(&gBinderOffsets)) {
        jobject object = static_cast<JavaBBinder*>(val.get())->object();
        LOGDEATH("objectForBinder %p: it's our own %p!\n", val.get(), object);
        return object;
    }

    // For the rest of the function we will hold this lock, to serialize
    // looking/creation of Java proxies for native Binder proxies.
    AutoMutex _l(mProxyLock);

    //因为这个 BpBinder 对象是第一创建，所以里面什么对象也没有，此处返回的 object 为 NULL
    jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
    if (object != NULL) {
        jobject res = jniGetReferent(env, object);
        if (res != NULL) {
            ALOGV("objectForBinder %p: found existing %p!\n", val.get(), res);
```

```
            return res;
        }
        LOGDEATH("Proxy object %p of IBinder %p no longer in working set!!!", object,
val.get());
        android_atomic_dec(&gNumProxyRefs);
        val->detachObject(&gBinderProxyOffsets);
        env->DeleteGlobalRef(object);
    }
//创建一个 BinderProxy 对象
    object = env->NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor);
    if (object != NULL) {
        LOGDEATH("objectForBinder %p: created new proxy %p !\n", val.get(), object);
        // The proxy holds a reference to the native object.
//通过 BinderProxy.mObject 成员变量关联 BpBinder 对象和这个 BinderProxy 对象
        env->SetIntField(object, gBinderProxyOffsets.mObject, (int)val.get());
        val->incStrong((void*)javaObjectForIBinder);

//放到 BpBinder 里面，当下次就要使用时
//可以在上一步调用 BpBinder::findObj 找回来
        jobject refObject = env->NewGlobalRef(
                env->GetObjectField(object, gBinderProxyOffsets.mSelf));
        val->attachObject(&gBinderProxyOffsets, refObject,
                jnienv_to_javavm(env), proxy_cleanup);

        // Also remember the death recipients registered on this proxy
        sp<DeathRecipientList> drl = new DeathRecipientList;
        drl->incStrong((void*)javaObjectForIBinder);
        env->SetIntField(object, gBinderProxyOffsets.mOrgue, reinterpret_cast<jint>
        (drl.get()));

        // Note that a new object reference has been created.
        android_atomic_inc(&gNumProxyRefs);
        incRefsCreated(env);
    }

    return object;
}
```

在上述代码中，涉及了如下 3 个重要的成员变量：

- gBinderOffsets；
- gBinderProxyOffsets；
- gWeakReferenceOffsets。

gBinderOffsets 是一个全局成员变量，是一个 bindernative_offsets_t 结构体类型，定义代码如下所示：

```
static struct bindernative_offsets_t
{
    // Class state.
    jclass mClass;
    jmethodID mExecTransact;
    // Object state.
    jfieldID mObject;
} gBinderOffsets;
```

成员变量 gBinderOffsets 用于记录类 Binder 的相关信息，它是在注册类 Binder 的 JNI 方法中的函数 int_register_android_os_Binder 实现初始化工作的，函数 int_register_android_os_Binder 的定义代码如下所示：

```
const char* const kBinderPathName = "android/os/Binder";
static int int_register_android_os_Binder(JNIEnv* env)
{
    jclass clazz;
    clazz = env->FindClass(kBinderPathName);
    LOG_FATAL_IF(clazz == NULL, "Unable to find class android.os.Binder");
    gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    gBinderOffsets.mExecTransact
        = env->GetMethodID(clazz, "execTransact", "(IIII)Z");
    assert(gBinderOffsets.mExecTransact);
```

```
    gBinderOffsets.mObject
        = env->GetFieldID(clazz, "mObject", "I");
    assert(gBinderOffsets.mObject);
    return AndroidRuntime::registerNativeMethods(
        env, kBinderPathName,
        gBinderMethods, NELEM(gBinderMethods));
}
```

gBinderProxyOffsets 是一个全局变量，用于记录类 BinderProxy 的相关信息。变量 gBinderProxyOffsets 是一个 binderproxy_offsets_t 结构体类型，定义代码如下所示：

```
static struct binderproxy_offsets_t
{
    // Class state.
    jclass mClass;
    jmethodID mConstructor;
    jmethodID mSendDeathNotice;
    // Object state.
    jfieldID mObject;
    jfieldID mSelf;
} gBinderProxyOffsets;
```

变量 gBinderProxyOffsets 在注册类 BinderProxy 的 JNI 方法中，通过函数 int_register_android_os_BinderProxy 实现初始化工作的。函数 int_register_android_os_BinderProxy 的具体实现代码如下所示：

```
const char* const kBinderProxyPathName = "android/os/BinderProxy";
static int int_register_android_os_BinderProxy(JNIEnv* env)
{
    jclass clazz;
    clazz = env->FindClass("java/lang/ref/WeakReference");
    LOG_FATAL_IF(clazz == NULL, "Unable to find class java.lang.ref.WeakReference");
    gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    gWeakReferenceOffsets.mGet
        = env->GetMethodID(clazz, "get", "()Ljava/lang/Object;");
    assert(gWeakReferenceOffsets.mGet);

    clazz = env->FindClass("java/lang/Error");
    LOG_FATAL_IF(clazz == NULL, "Unable to find class java.lang.Error");
    gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz);

    clazz = env->FindClass(kBinderProxyPathName);
    LOG_FATAL_IF(clazz == NULL, "Unable to find class android.os.BinderProxy");

    gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
    gBinderProxyOffsets.mConstructor
        = env->GetMethodID(clazz, "<init>", "()V");
    assert(gBinderProxyOffsets.mConstructor);
    gBinderProxyOffsets.mSendDeathNotice
        = env->GetStaticMethodID(clazz, "sendDeathNotice", "(Landroid/os/IBinder$Death
Recipient;)V");
    assert(gBinderProxyOffsets.mSendDeathNotice);

    gBinderProxyOffsets.mObject
        = env->GetFieldID(clazz, "mObject", "I");
    assert(gBinderProxyOffsets.mObject);
    gBinderProxyOffsets.mSelf
        = env->GetFieldID(clazz, "mSelf", "Ljava/lang/ref/WeakReference;");
    assert(gBinderProxyOffsets.mSelf);

    return AndroidRuntime::registerNativeMethods(
        env, kBinderProxyPathName,
        gBinderProxyMethods, NELEM(gBinderProxyMethods));
}
```

gWeakReferenceOffsets 是一个全局变量，用来和 WeakReference 类打交道。gWeakReference Offsets 是一个 weakreference_offsets_t 结构体类型，定义代码如下所示：

```
static struct weakreference_offsets_t
{
```

```
    // Class state.
    jclass mClass;
    jmethodID mGet;
} gWeakReferenceOffsets;
```

变量 gWeakReferenceOffsets 通过函数 int_register_android_os_BinderProxy 实现初始化工作的。

在 javaObjectForIBinder 实现代码的最后，调用了 ServiceManagerNative 中的函数 asInterface，此函数在文件 frameworks\base\core\java\android\os\ServiceManagerNative.java 中定义，具体实现代码如下所示：

```
public abstract class ServiceManagerNative extends Binder implements IServiceManager
{
    /**
     * Cast a Binder object into a service manager interface, generating
     * a proxy if needed.
     */
    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);
    }
```

在上述代码中，参数 obj 是一个 BinderProxy 对象，函数 queryLocalInterface 会返回 null。这样以这个 BinderProxy 对象为参数创建了一个 ServiceManagerProxy 对象。

6.5.2　分析 ActivityManagerService 的 Java 层

在接下来的内容中，将以 ActivityManagerService 中的开源代码为例，详细讲解 Binder 通信机制中的 Java 层的工作原理。该演示实例的分析步骤如下所示：

- 首先分析 AMS 如何将自己注册到 ServiceManager；
- 然后分析 AMS 如何响应客户端的 Binder 调用请求。

ActivityManagerService 的源代码在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\am\ActivityManagerService.java
```

在接下来的内容中，将详细分析 ActivityManagerService 实例中 Java 层的具体实现流程。

（1）本例的起点是函数 setSystemProcess，其具体实现代码如下所示：

```
public static void setSystemProcess() {
    try {
        ActivityManagerService m = mSelf;

        ServiceManager.addService("activity", m, true);
        ServiceManager.addService("meminfo", new MemBinder(m));
        ServiceManager.addService("gfxinfo", new GraphicsBinder(m));
        ServiceManager.addService("dbinfo", new DbBinder(m));
        if (MONITOR_CPU_USAGE) {
            ServiceManager.addService("cpuinfo", new CpuBinder(m));
        }
        ServiceManager.addService("permission", new PermissionController(m));

        ApplicationInfo info =
            mSelf.mContext.getPackageManager().getApplicationInfo(
                    "android", STOCK_PM_FLAGS);
        mSystemThread.installSystemApplicationInfo(info);

        synchronized (mSelf) {
            ProcessRecord app = mSelf.newProcessRecordLocked(
                    mSystemThread.getApplicationThread(), info,
```

```
                info.processName, false);
            app.persistent = true;
            app.pid = MY_PID;
            app.maxAdj = ProcessList.SYSTEM_ADJ;
            mSelf.mProcessNames.put(app.processName, app.uid, app);
            synchronized (mSelf.mPidsSelfLocked) {
                mSelf.mPidsSelfLocked.put(app.pid, app);
            }
            mSelf.updateLruProcessLocked(app, true);
        }
    } catch (PackageManager.NameNotFoundException e) {
        throw new RuntimeException(
                "Unable to find android system package", e);
    }
}
```

上述代码行的目的是将 AMS 服务注册到 ServiceManager 中。

（2）整个 Android 系统中有一个 Native 的 ServiceManager（以下简称 SM）进程，它统筹管理 Android 系统上的所有 Service。成为一个 Service 的必要条件是在 SM 中注册。下面来看 Java 层的 Service 是如何在 SM 中注册的。

- 在 SM 中注册服务

在 ServiceManager 中注册服务的函数为 addService，在如下所示的文件中实现：

```
frameworks\base\tools\layoutlib\bridge\src\android\os\ServiceManager.java
```

函数 addService 的具体实现代码如下所示：

```
public static void addService(String name, IBinder service) {
try {
      getIServiceManager().addService(name, service);
   }
   ......
 }
//分析 getIServiceManager 函数
private static IServiceManager getIServiceManager() {
  ......
  //调用 asInterface，传递的参数类型为 IBinder
  sServiceManager = ServiceManagerNative.asInterface(
                    BinderInternal.getContextObject());
  return sServiceManager;
}
```

在上述代码中，函数 asInterface 的参数为 BinderInternal.getContextObject 的返回值，这是一个 native 的函数。

（3）再看函数 javaObjectForIBinder，此函数在如下所示的文件中实现：

```
frameworks\base\core\jni\android_util_Binder.cpp
```

函数 javaObjectForIBinder 的具体实现代码如下所示：

```
jobject javaObjectForIBinder(JNIEnv* env, const sp<IBinder>& val)
{
 //mProxyLock 是一个全局的静态 CMutex 对象
  AutoMutex _l(mProxyLock);

  /*
  val 对象实际类型是 BpBinder，事实上，在 Native 层的 BpBinder 中有一个 ObjectManager，
 它用来管理在 Native BpBinder 上创建的 Java BpBinder 对象。
  下面这个 findObject 用来判断 gBinderProxyOffsets
  是否已经保存在 ObjectManager 中。如果是，那就需要删除旧的 object
  */
 jobject object = (jobject)val->findObject(&gBinderProxyOffsets);
  if (object != NULL) {
     jobject res = env->CallObjectMethod(object, gWeakReferenceOffsets.mGet);
     android_atomic_dec(&gNumProxyRefs);
     val->detachObject(&gBinderProxyOffsets);
     env->DeleteGlobalRef(object);
  }
```

```
    //创建一个新的 BinderProxy 对象，并注册到 Native BpBinder 对象的 ObjectManager 中
      object = env->NewObject(gBinderProxyOffsets.mClass,
                              gBinderProxyOffsets.mConstructor);
    if (object != NULL) {
      env->SetIntField(object, gBinderProxyOffsets.mObject, (int)val.get());
      val->incStrong(object);
      jobject refObject = env->NewGlobalRef(
            env->GetObjectField(object, gBinderProxyOffsets.mSelf));
      /*
      将这个新创建的 BinderProxy 对象注册（attach）到 BpBinder 的 ObjectManager 中，
      同时注册一个回收函数 proxy_cleanup。当 BinderProxy 对象撤销（detach）的时候，
      该函数会被调用，以释放一些资源。读者可自行研究 proxy_cleanup 函数
      */
      val->attachObject(&gBinderProxyOffsets, refObject,
                        jnienv_to_javavm(env), proxy_cleanup);

      // DeathRecipientList 保存了一个用于死亡通知的 list
      sp<DeathRecipientList>drl = new DeathRecipientList;
      drl->incStrong((void*)javaObjectForIBinder);
      //将死亡通知 list 和 BinderProxy 对象联系起来
      env->SetIntField(object, gBinderProxyOffsets.mOrgue,
                       reinterpret_cast<jint>(drl.get()));
      //增加该 Proxy 对象的引用计数
      android_atomic_inc(&gNumProxyRefs);
      //下面这个函数用于垃圾回收。创建的 Proxy 对象一旦超过 200 个，该函数
      //将调用 BinderInter 类的 ForceGc 做一次垃圾回收
      incRefsCreated(env);
    }

    return object;
}
```

函数 BinderInternal.getContextObject 完成了以下两个工作：

- 创建了一个 Java 层的 BinderProxy 对象；
- 通过 JNI，该 BinderProxy 对象和一个 Native 的 BpProxy 对象挂钩，而该 BpProxy 对象的通信目标就是 ServiceManager。

（4）现在来分析 ServiceManagerProxy 中的函数 addService，此函数在如下所示的文件中实现：

```
frameworks\base\core\java\android\os\ServiceManagerNative.java
```

函数 addService 的具体实现代码如下所示：

```
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);
     //mRemote 实际上就是 BinderProxy 对象，调用它的 transact，将封装好的请求数据
    //发送出去
     mRemote.transact(ADD_SERVICE_TRANSACTION, data, reply, 0);
     reply.recycle();
     data.recycle();
}
```

BinderProxy 的 transact 是一个 native 函数，此函数在文件 android_util_Binder.cpp 中实现，具体实现代码如下所示：

```
static jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj,
                                    jint code, jobject dataObj,
                                    jobject replyObj, jint flags)
{
if (dataObj == NULL) {
   jniThrowNullPointerException(env, NULL);
   return JNI_FALSE;
}
  //从 Java 的 Parcel 对象中得到 Native 的 Parcel 对象
  Parcel* data = parcelForJavaObject(env, dataObj);
  if (data == NULL) {
```

```
        return JNI_FALSE;
    }
    //得到一个用于接收回复的 Parcel 对象
    Parcel* reply = parcelForJavaObject(env, replyObj);
    if (reply == NULL && replyObj != NULL) {
        return JNI_FALSE;
    }
    //从 Java 的 BinderProxy 对象中得到之前已经创建好的那个 Native 的 BpBinder 对象
    IBinder* target = (IBinder*)
        env->GetIntField(obj, gBinderProxyOffsets.mObject);
       if (target == NULL) {
      jniThrowException(env, "java/lang/IllegalStateException", "Binder has been finalized!");
      return JNI_FALSE;
}

ALOGV("Java code calling transact on %p in Java object %p with code %d\n",
        target, obj, code);
const bool time_binder_calls = should_time_binder_calls();

int64_t start_millis;
if (time_binder_calls) {
    start_millis = uptimeMillis();
}
    //通过 Native 的 BpBinder 对象，将请求发送给 ServiceManager
    status_t err = target->transact(code, *data, reply, flags);
        if (time_binder_calls) {
      conditionally_log_binder_call(start_millis, target, code);
}

if (err == NO_ERROR) {
    return JNI_TRUE;
} else if (err == UNKNOWN_TRANSACTION) {
    return JNI_FALSE;
}
    signalExceptionForError(env, obj, err);
    return JNI_FALSE;
}
```

通过上述代码可知，Java 层的 Binder 架构最终还是要借助 Native 的 Binder 架构进行通信。从架构的角度看，在 Java 中搭建了一整套框架，如 IBinder 接口、Binder 类和 BinderProxy 类。但是从通信角度看，无论采用的是 Native 语言还是 Java 语言，只要把请求传递到 Binder 驱动即可。所以，通信的目的是向 Binder 发送请求和接收回复。

（5）ActivityManagerService 从类 ActivityManagerNative 派生，其中和 Binder 架构相关的类是 ActivityManagerNative，此类在文件 ActivityManagerNative.java 中实现，具体代码如下所示：

```
public abstract class ActivityManagerNative
                        extends Binder
                        implements IActivityManager
```

ActivityManagerNative 从 Binder 派生，并且实现了 IActivityManager 接口。下面来看 ActivityManagerNative 的构造函数，此构造函数在文件 ActivityManagerNative.java 中实现，具体实现代码如下所示：

```
public ActivityManagerNative() {
      attachInterface(this, descriptor);//该函数很简单，读者可自行分析
    }
//这是 ActivityManagerNative 父类的构造函数，即 Binder 的构造函数
public Binder() {
      init();
}
```

在 Binder 构造函数中会调用 Native 的 init 函数 android_os_Binder_init，此函数在文件 android_util_Binder.cpp 中定义，具体实现的代码如下所示：

```
static void android_os_Binder_init(JNIEnv* env, jobject obj)
{
    //创建一个 JavaBBinderHolder 对象
    JavaBBinderHolder* jbh = new JavaBBinderHolder();
```

```
    bh->incStrong((void*)android_os_Binder_init);
    //将这个 JavaBBinderHolder 对象保存到 Java Binder 对象的 mObject 成员中
    env->SetIntField(obj, gBinderOffsets.mObject, (int)jbh);
}
```

从上述实现代码可知，Java 的 Binder 对象和 Native 的 JavaBBinderHolder 对象相关联。JavaBBinderHolder 在文件 android_util_Binder.cpp 中定义，具体实现代码如下所示：

```
class JavaBBinderHolder : public RefBase
{
public:
    sp<JavaBBinder> get(JNIEnv* env, jobject obj)
    {
        AutoMutex _l(mLock);
        sp<JavaBBinder> b = mBinder.promote();
        if (b == NULL) {
            b = new JavaBBinder(env, obj);
            mBinder = b;
            ALOGV("Creating JavaBinder %p (refs %p) for Object %p, weakCount=%d\n",
                b.get(), b->getWeakRefs(), obj, b->getWeakRefs()->getWeakCount());
        }

        return b;
    }

    sp<JavaBBinder> getExisting()
    {
        AutoMutex _l(mLock);
        return mBinder.promote();
    }

private:
    Mutex          mLock;
    wp<JavaBBinder> mBinder;
};
```

在上述代码中，从派生关系上可以发现，JavaBBinderHolder 仅从 RefBase 派生，所以它不属于 Binder 体系。类 JavaBBinderHolder 的函数 get 中创建了一个 JavaBBinder 对象，这个对象就是从 BnBinder 派生的。

addService 实际添加到 Parcel 的是一个名为 JavaBBinder 的对象，addService 正是该 JavaBBinder 对象最终传递到 Binder 驱动。不同的 Binder 对象对应不同的 JavaBBinder 对象。Binder、JavaBBinderHolder 和 Java-BBinder 的关系如下所示：

- Java 层的 Binder 通过 mObject 指向一个 Native 层的 JavaBBinderHolder 对象；
- Native 层的 JavaBBinderHolder 对象通过 mBinder 成员变量指向一个 Native 的 Java-BBinder 对象；
- Native 的 JavaBBinder 对象又通过 mObject 变量指向一个 Java 层的 Binder 对象。

（6）在 Java 层的 Binder 架构中，JavaBBinder 是一个和业务完全无关的对象。当收到请求时，系统会调用函数 onTransact。函数 onTransact 在文件 android_util_Binder.cpp 中定义，具体实现代码如下所示：

```
    virtual status_t onTransact(
        uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags = 0)
    {
        JNIEnv* env = javavm_to_jnienv(mVM);

        ALOGV("onTransact() on %p calling object %p in env %p vm %p\n", this, mObject,
        env, mVM);

        IPCThreadState* thread_state = IPCThreadState::self();
        const int strict_policy_before = thread_state->getStrictModePolicy();
        thread_state->setLastTransactionBinderFlags(flags);

        //printf("Transact from %p to Java code sending: ", this);
        //data.print();
        //printf("\n");
```

```
        //调用 Java 层 Binder 对象的 execTransact 函数
        jboolean res = env->CallBooleanMethod(mObject, gBinderOffsets.mExecTransact,
            code, (int32_t)&data, (int32_t)reply, flags);
        jthrowable excep = env->ExceptionOccurred();

        if (excep) {
            report_exception(env, excep,
                "*** Uncaught remote exception!  "
                "(Exceptions are not yet supported across processes.)");
            res = JNI_FALSE;

            /*清理本地 JNI 引用 */
            env->DeleteLocalRef(excep);
        }
        const int strict_policy_after = thread_state->getStrictModePolicy();
        if (strict_policy_after != strict_policy_before) {
            thread_state->setStrictModePolicy(strict_policy_before);
            set_dalvik_blockguard_policy(env, strict_policy_before);
        }

        jthrowable excep2 = env->ExceptionOccurred();
        if (excep2) {
            report_exception(env, excep2,
                "*** Uncaught exception in onBinderStrictModePolicyChange");
            /*清理本地 JNI 引用 */
            env->DeleteLocalRef(excep2);
        }

        // Need to always call through the native implementation of
        // SYSPROPS_TRANSACTION.
        if (code == SYSPROPS_TRANSACTION) {
            BBinder::onTransact(code, data, reply, flags);
        }

        return res != JNI_FALSE ? NO_ERROR : UNKNOWN_TRANSACTION;
    }

    virtual status_t dump(int fd, const Vector<String16>& args)
    {
        return 0;
    }

private:
    JavaVM* const  mVM;
    jobject const  mObject;
};
```

在上述代码中，mObject 就是 ActivityManagerService，开始调用 execTransact 函数。函数 execTransact 在类 Binder 中实现，在文件 Binder.java 中定义，具体实现代码如下所示：

```
private boolean execTransact(int code, int dataObj, int replyObj,int flags) {
     Parcel data = Parcel.obtain(dataObj);
     Parcel reply = Parcel.obtain(replyObj);
     boolean res;
     try {
        //调用 onTransact 函数，派生类可以重新实现这个函数，以完成业务功能
        res = onTransact(code, data, reply, flags);
           } catch (RemoteException e) {
       reply.setDataPosition(0);
       reply.writeException(e);
       res = true;
    } catch (RuntimeException e) {
       reply.setDataPosition(0);
       reply.writeException(e);
       res = true;
    } catch (OutOfMemoryError e) {
       RuntimeException re = new RuntimeException("Out of memory", e);
       reply.setDataPosition(0);
       reply.writeException(re);
       res = true;
    }
```

```
        reply.recycle();
        data.recycle();
        return res;
    }
}
```

（7）在类 ActivityManagerNative 中实现了 onTransact 函数，此函数在如下所示的文件中实现：

```
frameworks\base\core\java\android\app\ActivityManagerNative.java
```

函数 onTransact 的具体实现代码如下所示：

```
    public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
            throws RemoteException {
        switch (code) {
        case START_ACTIVITY_TRANSACTION:
        {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder b = data.readStrongBinder();
            IApplicationThread app = ApplicationThreadNative.asInterface(b);
            Intent intent = Intent.CREATOR.createFromParcel(data);
            String resolvedType = data.readString();
            IBinder resultTo = data.readStrongBinder();
            String resultWho = data.readString();
            int requestCode = data.readInt();
            int startFlags = data.readInt();
            String profileFile = data.readString();
            ParcelFileDescriptor profileFd = data.readInt() != 0
                    ? data.readFileDescriptor() : null;
            Bundle options = data.readInt() != 0
                    ? Bundle.CREATOR.createFromParcel(data) : null;
            int result = startActivity(app, intent, resolvedType,
                    resultTo, resultWho, requestCode, startFlags,
                    profileFile, profileFd, options);
            reply.writeNoException();
            reply.writeInt(result);
            return true;
        }

        case START_ACTIVITY_AS_USER_TRANSACTION:
        {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder b = data.readStrongBinder();
            IApplicationThread app = ApplicationThreadNative.asInterface(b);
            Intent intent = Intent.CREATOR.createFromParcel(data);
            String resolvedType = data.readString();
            IBinder resultTo = data.readStrongBinder();
            String resultWho = data.readString();
            int requestCode = data.readInt();
            int startFlags = data.readInt();
            String profileFile = data.readString();
            ParcelFileDescriptor profileFd = data.readInt() != 0
                    ? data.readFileDescriptor() : null;
            Bundle options = data.readInt() != 0
                    ? Bundle.CREATOR.createFromParcel(data) : null;
            int userId = data.readInt();
            int result = startActivityAsUser(app, intent, resolvedType,
                    resultTo, resultWho, requestCode, startFlags,
                    profileFile, profileFd, options, userId);
            reply.writeNoException();
            reply.writeInt(result);
            return true;
        }
        case START_ACTIVITY_AND_WAIT_TRANSACTION:
        {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder b = data.readStrongBinder();
            IApplicationThread app = ApplicationThreadNative.asInterface(b);
            String callingPackage = data.readString();
            Intent intent = Intent.CREATOR.createFromParcel(data);
            String resolvedType = data.readString();
            IBinder resultTo = data.readStrongBinder();
            String resultWho = data.readString();
```

```
        int requestCode = data.readInt();
        int startFlags = data.readInt();
        String profileFile = data.readString();
        ParcelFileDescriptor profileFd = data.readInt() != 0
                ? data.readFileDescriptor() : null;
        Bundle options = data.readInt() != 0
                ? Bundle.CREATOR.createFromParcel(data) : null;
        int userId = data.readInt();
        WaitResult result = startActivityAndWait(app, callingPackage, intent,
        resolvedType,
                resultTo, resultWho, requestCode, startFlags,
                profileFile, profileFd, options, userId);
        reply.writeNoException();
        result.writeToParcel(reply, 0);
        return true;
    }

    case START_ACTIVITY_WITH_CONFIG_TRANSACTION:
    {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder b = data.readStrongBinder();
        IApplicationThread app = ApplicationThreadNative.asInterface(b);
        String callingPackage = data.readString();
        Intent intent = Intent.CREATOR.createFromParcel(data);
        String resolvedType = data.readString();
        IBinder resultTo = data.readStrongBinder();
        String resultWho = data.readString();
        int requestCode = data.readInt();
        int startFlags = data.readInt();
        Configuration config = Configuration.CREATOR.createFromParcel(data);
        Bundle options = data.readInt() != 0
                ? Bundle.CREATOR.createFromParcel(data) : null;
        int userId = data.readInt();
        int result = startActivityWithConfig(app, callingPackage, intent,
        resolvedType,
                resultTo, resultWho, requestCode, startFlags, config, options, userId);
        reply.writeNoException();
        reply.writeInt(result);
        return true;
    }

    case START_ACTIVITY_INTENT_SENDER_TRANSACTION:
    {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder b = data.readStrongBinder();
        IApplicationThread app = ApplicationThreadNative.asInterface(b);
        IntentSender intent = IntentSender.CREATOR.createFromParcel(data);
        Intent fillInIntent = null;
        if (data.readInt() != 0) {
            fillInIntent = Intent.CREATOR.createFromParcel(data);
        }
        String resolvedType = data.readString();
        IBinder resultTo = data.readStrongBinder();
        String resultWho = data.readString();
        int requestCode = data.readInt();
        int flagsMask = data.readInt();
        int flagsValues = data.readInt();
        Bundle options = data.readInt() != 0
                ? Bundle.CREATOR.createFromParcel(data) : null;
        int result = startActivityIntentSender(app, intent,
                fillInIntent, resolvedType, resultTo, resultWho,
                requestCode, flagsMask, flagsValues, options);
        reply.writeNoException();
        reply.writeInt(result);
        return true;
    }

    case START_NEXT_MATCHING_ACTIVITY_TRANSACTION:
    {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder callingActivity = data.readStrongBinder();
        Intent intent = Intent.CREATOR.createFromParcel(data);
```

```
        Bundle options = data.readInt() != 0
                ? Bundle.CREATOR.createFromParcel(data) : null;
        boolean result = startNextMatchingActivity(callingActivity, intent, options);
        reply.writeNoException();
        reply.writeInt(result ? 1 : 0);
        return true;
    }

    case FINISH_ACTIVITY_TRANSACTION: {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder token = data.readStrongBinder();
        Intent resultData = null;
        int resultCode = data.readInt();
        if (data.readInt() != 0) {
            resultData = Intent.CREATOR.createFromParcel(data);
        }
        boolean res = finishActivity(token, resultCode, resultData);
        reply.writeNoException();
        reply.writeInt(res ? 1 : 0);
        return true;
    }

    case FINISH_SUB_ACTIVITY_TRANSACTION: {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder token = data.readStrongBinder();
        String resultWho = data.readString();
        int requestCode = data.readInt();
        finishSubActivity(token, resultWho, requestCode);
        reply.writeNoException();
        return true;
    }

    case FINISH_ACTIVITY_AFFINITY_TRANSACTION: {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder token = data.readStrongBinder();
        boolean res = finishActivityAffinity(token);
        reply.writeNoException();
        reply.writeInt(res ? 1 : 0);
        return true;
    }

    case WILL_ACTIVITY_BE_VISIBLE_TRANSACTION: {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder token = data.readStrongBinder();
        boolean res = willActivityBeVisible(token);
        reply.writeNoException();
        reply.writeInt(res ? 1 : 0);
        return true;
    }

    case REGISTER_RECEIVER_TRANSACTION:
    {
        data.enforceInterface(IActivityManager.descriptor);
        IBinder b = data.readStrongBinder();
        IApplicationThread app =
            b != null ? ApplicationThreadNative.asInterface(b) : null;
        String packageName = data.readString();
        b = data.readStrongBinder();
        IIntentReceiver rec
            = b != null ? IIntentReceiver.Stub.asInterface(b) : null;
        IntentFilter filter = IntentFilter.CREATOR.createFromParcel(data);
        String perm = data.readString();
        int userId = data.readInt();
        Intent intent = registerReceiver(app, packageName, rec, filter, perm, userId);
        reply.writeNoException();
        if (intent != null) {
            reply.writeInt(1);
            intent.writeToParcel(reply, 0);
        } else {
            reply.writeInt(0);
        }
        return true;
```

```
}

case UNREGISTER_RECEIVER_TRANSACTION:
{
    data.enforceInterface(IActivityManager.descriptor);
    IBinder b = data.readStrongBinder();
    if (b == null) {
        return true;
    }
    IIntentReceiver rec = IIntentReceiver.Stub.asInterface(b);
    unregisterReceiver(rec);
    reply.writeNoException();
    return true;
}

case BROADCAST_INTENT_TRANSACTION:
{
    data.enforceInterface(IActivityManager.descriptor);
    IBinder b = data.readStrongBinder();
    IApplicationThread app =
        b != null ? ApplicationThreadNative.asInterface(b) : null;
    Intent intent = Intent.CREATOR.createFromParcel(data);
    String resolvedType = data.readString();
    b = data.readStrongBinder();
    IIntentReceiver resultTo =
        b != null ? IIntentReceiver.Stub.asInterface(b) : null;
    int resultCode = data.readInt();
    String resultData = data.readString();
    Bundle resultExtras = data.readBundle();
    String perm = data.readString();
    int appOp = data.readInt();
    boolean serialized = data.readInt() != 0;
    boolean sticky = data.readInt() != 0;
    int userId = data.readInt();
    int res = broadcastIntent(app, intent, resolvedType, resultTo,
            resultCode, resultData, resultExtras, perm, appOp,
            serialized, sticky, userId);
    reply.writeNoException();
    reply.writeInt(res);
    return true;
}

case UNBROADCAST_INTENT_TRANSACTION:
{
    data.enforceInterface(IActivityManager.descriptor);
    IBinder b = data.readStrongBinder();
    IApplicationThread app=b!= null ? ApplicationThreadNative.asInterface(b) : null;
    Intent intent = Intent.CREATOR.createFromParcel(data);
    int userId = data.readInt();
    unbroadcastIntent(app, intent, userId);
    reply.writeNoException();
    return true;
}

case FINISH_RECEIVER_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    IBinder who = data.readStrongBinder();
    int resultCode = data.readInt();
    String resultData = data.readString();
    Bundle resultExtras = data.readBundle();
    boolean resultAbort = data.readInt() != 0;
    if (who != null) {
        finishReceiver(who, resultCode, resultData, resultExtras, resultAbort);
    }
    reply.writeNoException();
    return true;
}

case ATTACH_APPLICATION_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    IApplicationThread app = ApplicationThreadNative.asInterface(
            data.readStrongBinder());
```

```
            if (app != null) {
                attachApplication(app);
            }
            reply.writeNoException();
            return true;
        }
        case ACTIVITY_IDLE_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            Configuration config = null;
            if (data.readInt() != 0) {
                config = Configuration.CREATOR.createFromParcel(data);
            }
            boolean stopProfiling = data.readInt() != 0;
            if (token != null) {
                activityIdle(token, config, stopProfiling);
            }
            reply.writeNoException();
            return true;
        }

        case ACTIVITY_RESUMED_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            activityResumed(token);
            reply.writeNoException();
            return true;
        }

        case ACTIVITY_PAUSED_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            activityPaused(token);
            reply.writeNoException();
            return true;
        }

        case ACTIVITY_STOPPED_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            Bundle map = data.readBundle();
            Bitmap thumbnail = data.readInt() != 0
                ? Bitmap.CREATOR.createFromParcel(data) : null;
            CharSequence description=TextUtils.CHAR_SEQUENCE_CREATOR.createFromParcel(data);
            activityStopped(token, map, thumbnail, description);
            reply.writeNoException();
            return true;
        }

        case ACTIVITY_SLEPT_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            activitySlept(token);
            reply.writeNoException();
            return true;
        }

        case ACTIVITY_DESTROYED_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            activityDestroyed(token);
            reply.writeNoException();
            return true;
        }

        case GET_CALLING_PACKAGE_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            IBinder token = data.readStrongBinder();
            String res = token != null ? getCallingPackage(token) : null;
            reply.writeNoException();
            reply.writeString(res);
            return true;
```

```
}

case GET_CALLING_ACTIVITY_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    IBinder token = data.readStrongBinder();
    ComponentName cn = getCallingActivity(token);
    reply.writeNoException();
    ComponentName.writeToParcel(cn, reply);
    return true;
}

case GET_TASKS_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    int maxNum = data.readInt();
    int fl = data.readInt();
    IBinder receiverBinder = data.readStrongBinder();
    IThumbnailReceiver receiver = receiverBinder != null
        ? IThumbnailReceiver.Stub.asInterface(receiverBinder)
        : null;
    List list = getTasks(maxNum, fl, receiver);
    reply.writeNoException();
    int N = list != null ? list.size() : -1;
    reply.writeInt(N);
    int i;
    for (i=0; i<N; i++) {
        ActivityManager.RunningTaskInfo info =
                (ActivityManager.RunningTaskInfo)list.get(i);
        info.writeToParcel(reply, 0);
    }
    return true;
}

case GET_RECENT_TASKS_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    int maxNum = data.readInt();
    int fl = data.readInt();
    int userId = data.readInt();
    List<ActivityManager.RecentTaskInfo> list = getRecentTasks(maxNum,
            fl, userId);
    reply.writeNoException();
    reply.writeTypedList(list);
    return true;
}

case GET_TASK_THUMBNAILS_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    int id = data.readInt();
    ActivityManager.TaskThumbnails bm = getTaskThumbnails(id);
    reply.writeNoException();
    if (bm != null) {
        reply.writeInt(1);
        bm.writeToParcel(reply, 0);
    } else {
        reply.writeInt(0);
    }
    return true;
}

case GET_TASK_TOP_THUMBNAIL_TRANSACTION: {
    data.enforceInterface(IActivityManager.descriptor);
    int id = data.readInt();
    Bitmap bm = getTaskTopThumbnail(id);
    reply.writeNoException();
    if (bm != null) {
        reply.writeInt(1);
        bm.writeToParcel(reply, 0);
    } else {
        reply.writeInt(0);
    }
    return true;
}
```

```
        case GET_SERVICES_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            int maxNum = data.readInt();
            int fl = data.readInt();
            List list = getServices(maxNum, fl);
            reply.writeNoException();
            int N = list != null ? list.size() : -1;
            reply.writeInt(N);
            int i;
            for (i=0; i<N; i++) {
                ActivityManager.RunningServiceInfo info =
                        (ActivityManager.RunningServiceInfo)list.get(i);
                info.writeToParcel(reply, 0);
            }
            return true;
        }

        case GET_PROCESSES_IN_ERROR_STATE_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            List<ActivityManager.ProcessErrorStateInfo> list = getProcessesInErrorState();
            reply.writeNoException();
            reply.writeTypedList(list);
            return true;
        }
        case GET_RUNNING_APP_PROCESSES_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            List<ActivityManager.RunningAppProcessInfo> list = getRunningAppProcesses();
            reply.writeNoException();
            reply.writeTypedList(list);
            return true;
        }

        case GET_RUNNING_EXTERNAL_APPLICATIONS_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            List<ApplicationInfo> list = getRunningExternalApplications();
            reply.writeNoException();
            reply.writeTypedList(list);
            return true;
        }
        case MOVE_TASK_TO_FRONT_TRANSACTION: {
            data.enforceInterface(IActivityManager.descriptor);
            int task = data.readInt();
            int fl = data.readInt();
            Bundle options = data.readInt() != 0
                    ? Bundle.CREATOR.createFromParcel(data) : null;
            moveTaskToFront(task, fl, options);
            reply.writeNoException();
            return true;
        }
    ......
         //再由 ActivityManagerService 实现业务函数 startActivity
         int result = startActivity(app, intent, resolvedType,
                 grantedUriPermissions, grantedMode, resultTo, resultWho,
                 requestCode, onlyIfNeeded, debug, profileFile,
                 profileFd, autoStopProfiler);
          reply.writeNoException();
          reply.writeInt(result);
          return true;
    }
```

由此可以看出，函数 JavaBBinder 本身不实现任何业务，其工作过程如下所示：

- 当收到请求时调用它所绑定的 Java 层 Binder 对象的 execTransact；
- 该 Binder 对象的 execTransact 调用其子类实现的函数 onTransact；
- 子类的函数 onTransact 将业务又派发给其子类来完成。

通过这种方式，来自客户端的请求就能传递到正确的 Java 层 Binder 对象了。由此可见，对于代表客户端的 BinderProxy 来说，Java 层的 BinderProxy 在 Native 层对应一个 BpBinder 对象。只要从 Java 层发出请求，首先从 Java 层的 BinderProxy 传递到 Native 层的 BpBinder，继而由 BpBinder 将请求发送到 Binder 驱动。

第 7 章　分析 ServiceManager 和 MessageQueue

在 Anroid 系统中，ServiceManager 是控制服务的管家。因为 Service Manager 组件用来管理 Server，并且向 Client 提供了查询 Server 远程接口的功能，所以 Service Manager 就必然要和 Server 以及 Client 进行通信。在本章的内容中，将详细分析 ServiceManager 系统的实现源代码，并以 MessageQueue 为例讲解整个通信应用的具体过程。

7.1 分析 ServiceManager

在 Android 系统中，Service Manger、Client 和 Server 分别运行在独立的进程中，它们之间的通信属于进程之间的通信。在具体通信时采用了 Binder 机制，当 Service Manager 作为 Binder 机制的总管时，同时也充当了 Server 的角色。在本节的内容中，将详细分析 ServiceManager 实现总管和通信的源代码。

7.1.1　分析主入口函数

在 Anroid 系统中，Service Manager 在用户空间的源代码位于如下目录中：

```
frameworks\native\cmds\servicemanager
```

其核心功能主要是由文件 binder.h、binder.c 和 service_manager.c 组成。

Service Manager 的入口是文件 service_manager.c 中的函数 main，功能是：

- 打开 Binder 设备文件；
- 告诉 Binder 驱动程序自己是 Binder 上下文管理者；
- 进入到无穷循环中充当 Server 的角色，并等待 Client 的请求。

函数 main 的具体实现代码如下所示：

```
int main(int argc, char **argv)
{
    struct binder_state *bs;
    void *svcmgr = BINDER_SERVICE_MANAGER;
//调用函数 binder_open
    bs = binder_open(128*1024);
    if (binder_become_context_manager(bs)) {
        ALOGE("cannot become context manager (%s)\n", strerror(errno));
        return -1;
    }
    svcmgr_handle = svcmgr;
    binder_loop(bs, svcmgr_handler);
    return 0;
}
```

在上述实现代码中，用到了结构体 binder_state 和宏 BINDER_SERVICE_MANAGER。其中结构体 struct binder_state 在文件 binder.c 中定义，具体实现代码如下所示：

```
struct binder_state
{
    int fd;// 表示打开的/dev/binder 设备文件描述符
    void *mapped;// 把设备文件/dev/binder 映射到进程空间的起始地址
    unsigned mapsize;// 内存映射空间的大小
};
```

宏 BINDER_SERVICE_MANAGER 在文件 binder.h 中定义，具体实现代码如下所示：

```
/*表示 Service Manager 的句柄为 0 */
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

7.1.2　打开 Binder 设备文件

在函数 main 中调用函数 binder_open 打开 Binder 设备文件，此函数在文件 binder.c 中定义，具体实现代码如下所示：

```
struct binder_state *binder_open(unsigned mapsize)
{
    struct binder_state *bs;

    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return 0;
    }
//进入到 Binder 驱动程序的 binder_open 函数
    bs->fd = open("/dev/binder", O_RDWR);
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open device (%s)\n",
                strerror(errno));
        goto fail_open;
    }

    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
        goto fail_map;
    }
        /* TODO: check version */
    return bs;
fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return 0;
}
```

函数 binder_open 打开的 Binder 设备文件是/dev/binder，在初始化 Binder 驱动程序模块的时候创建此文件。在文件/drivers/staging/android/binder.c 中可以看到模块初始化入口函数 binder_init，此函数的实现代码如下所示：

```
static int __init binder_init(void)
{
    int ret;

    binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
    if (binder_proc_dir_entry_root)
        binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
    ret = misc_register(&binder_miscdev);
    if (binder_proc_dir_entry_root) {
        create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root, binder_
read_proc_state, NULL);
        create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root, binder_
read_proc_stats, NULL);
        create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_root,
binder_read_proc_transactions, NULL);
        create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_root,
binder_read_proc_transaction_log, &binder_transaction_log);
        create_proc_read_entry("failed_transaction_log", S_IRUGO, binder_proc_dir_
entry_root, binder_read_proc_transaction_log, &binder_transaction_log_failed);
    }
    return ret;
}
```

再进入到 Binder 驱动程序的函数 binder_open 中，功能是创建数据结构 struct binder_proc 来保存打开设备文件/dev/binder 的进程中的上下文信息，并将进程上下文信息保存在打开文件结构 struct file 的私有数据成员变量 private_data 中。当在执行其他文件操作时，可以通过打开文件结构 struct file 获取这个进程的上下文信息。函数 binder_open 的具体实现代码如下所示：

```
static int binder_open(struct inode *nodp, struct file *filp)
{
    struct binder_proc *proc;
    if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
        printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    if (proc == NULL)
        return -ENOMEM;
    get_task_struct(current);
    proc->tsk = current;
    INIT_LIST_HEAD(&proc->todo);
    init_waitqueue_head(&proc->wait);
    proc->default_priority = task_nice(current);
    mutex_lock(&binder_lock);
    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;
    mutex_unlock(&binder_lock);

    if (binder_proc_dir_entry_proc) {
        char strbuf[11];
        snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
        remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
        create_proc_read_entry(strbuf,  S_IRUGO,  binder_proc_dir_entry_proc,  binder_
read_proc_proc, proc);
    }
    return 0;
}
```

在上述代码中用到了结构体 binder_proc，定义代码如下所示：

```
struct binder_proc {
    struct hlist_node proc_node;
    // 用来保存 binder_proc 进程内用于处理用户请求的线程，它的最大数量由 max_threads 来决定
    struct rb_root threads;
    struct rb_root nodes;// 保存 binder_proc 进程内的 Binder 实体
//refs_by_desc 树和 refs_by_node 树用来保存 binder_proc 进程内的 Binder 引用
//引用的其他进程的 Binder 实体，它分别用两种方式来组织红黑树
//一种是以句柄作为 key 值来组织，另一种是以引用的实体节点的地址值作为 key 值来组织
    struct rb_root refs_by_desc;
    struct rb_root refs_by_node;
    int pid;
    struct vm_area_struct *vma;
    struct task_struct *tsk;
    struct files_struct *files;
    struct hlist_node deferred_work_node;
    int deferred_work;
    void *buffer;
    ptrdiff_t user_buffer_offset;

    struct list_head buffers;
    struct rb_root free_buffers;
    struct rb_root allocated_buffers;
    size_t free_async_space;

    struct page **pages;
    size_t buffer_size;
    uint32_t buffer_free;
    struct list_head todo;
    wait_queue_head_t wait;
    struct binder_stats stats;
    struct list_head delivered_death;
    int max_threads;
```

```
    int requested_threads;
    int requested_threads_started;
    int ready_threads;
    long default_priority;
};
```

7.1.3　注册处理

再看函数 main，接着调用函数 binder_become_context_manager 通知 Binder 驱动程序自己的身份是 Binder 机制的上下文管理者。函数 binder_become_context_manager 在如下文件中定义：

```
frameworks\native\cmds\servicemanager\binder.c
```

函数 binder_become_context_manager 的实现代码如下所示：

```
int binder_become_context_manager(struct binder_state *bs)
{
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

在上述代码中，调用 ioctl 文件操作函数向 Binder 驱动程序传递自己是守护进程的信息，命令号是 BINDER_SET_CONTEXT_MGR。BINDER_SET_CONTEXT_MGR 的定义代码如下所示：

```
#define BINDER_SET_CONTEXT_MGR _IOW('b', 7, int)
```

进入到 Binder 驱动程序函数 binder_ioctl，具体实现代码如下所示：

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
    void __user *ubuf = (void __user *)arg;

    /*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/

    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        return ret;

    mutex_lock(&binder_lock);
    thread = binder_get_thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {
    case BINDER_SET_CONTEXT_MGR:
        if (binder_context_mgr_node != NULL) {
            printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
            ret = -EBUSY;
            goto err;
        }
        if (binder_context_mgr_uid != -1) {
            if (binder_context_mgr_uid != current->cred->euid) {
                printk(KERN_ERR "binder: BINDER_SET_"
                    "CONTEXT_MGR bad uid %d != %d\n",
                    current->cred->euid,
                    binder_context_mgr_uid);
                ret = -EPERM;
                goto err;
            }
        } else
            binder_context_mgr_uid = current->cred->euid;
        binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
        if (binder_context_mgr_node == NULL) {
            ret = -ENOMEM;
            goto err;
        }
```

```
        binder_context_mgr_node->local_weak_refs++;
        binder_context_mgr_node->local_strong_refs++;
        binder_context_mgr_node->has_strong_ref = 1;
        binder_context_mgr_node->has_weak_ref = 1;
        break;
    default:
        ret = -EINVAL;
        goto err;
    }
    ret = 0;
err:
    if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
    mutex_unlock(&binder_lock);
    wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret && ret != -ERESTARTSYS)
        printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid,
        current->pid, cmd, arg, ret);
    return ret;
}
```

在上述函数的实现代码中，用到了如下所示的两个数据结构。

- struct binder_thread：表示一个线程，此处是执行函数 binder_become_context_manager 的线程。
- struct binder_node：表示一个 Binder 实体。

上述结构体的定义代码如下所示：

```
struct binder_thread {
//成员变量 threads 的类型是 rb_root，它表示一棵红黑树，把属于这个进程的所有线程都组织起来
    struct binder_proc *proc;
//成员变量 rb_node 用来链入这棵红黑树的节点
    struct rb_node rb_node;
    int pid;
    int looper;// looper 成员变量表示线程的状态
    struct binder_transaction *transaction_stack;// 表示线程正在处理的事务
    struct list_head todo;// 表示发往该线程的数据列表
    uint32_t return_error; /*表示操作结果返回码*/
    uint32_t return_error2; /* 表示操作结果返回码 */
        /* buffer. Used when sending a reply to a dead process that */
        /* we are also waiting on */
    wait_queue_head_t wait;// 用来阻塞线程等待某个事件的发生
    struct binder_stats stats;// 用来保存一些统计信息
};
struct binder_node {
    int debug_id;
    struct binder_work work;
    union {
        struct rb_node rb_node;
        struct hlist_node dead_node;
    };
    struct binder_proc *proc;
    struct hlist_head refs;
    int internal_strong_refs;
    int local_weak_refs;
    int local_strong_refs;
    void __user *ptr;
    void __user *cookie;
    unsigned has_strong_ref : 1;
    unsigned pending_strong_ref : 1;
    unsigned has_weak_ref : 1;
    unsigned pending_weak_ref : 1;
    unsigned has_async_transaction : 1;
    unsigned accept_fds : 1;
    int min_priority : 8;
    struct list_head async_todo;
};
```

在上述代码中，looper 成员变量表可以取下面的几个值：

```
enum {
    BINDER_LOOPER_STATE_REGISTERED  = 0x01,
```

```
    BINDER_LOOPER_STATE_ENTERED     = 0x02,
    BINDER_LOOPER_STATE_EXITED      = 0x04,
    BINDER_LOOPER_STATE_INVALID     = 0x08,
    BINDER_LOOPER_STATE_WAITING     = 0x10,
    BINDER_LOOPER_STATE_NEED_RETURN = 0x20
};
```

在函数 binder_ioctl 中，先通过 filp→private_data 获得 proc 变量，然后通过函数 binder_get_thread 获得线程信息。把当前线程 current 的 pid 作为键值，然后在进程 proc→threads 表示的红黑树中进行查找工作，查看是否已经为当前线程创建过了 binder_thread 信息。函数 binder_get_thread 的具体实现代码如下所示：

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
    struct binder_thread *thread = NULL;
    struct rb_node *parent = NULL;
    struct rb_node **p = &proc->threads.rb_node;

    while (*p) {
        parent = *p;
        thread = rb_entry(parent, struct binder_thread, rb_node);

        if (current->pid < thread->pid)
            p = &(*p)->rb_left;
        else if (current->pid > thread->pid)
            p = &(*p)->rb_right;
        else
            break;
    }
    if (*p == NULL) {
        thread = kzalloc(sizeof(*thread), GFP_KERNEL);
        if (thread == NULL)
            return NULL;
        binder_stats.obj_created[BINDER_STAT_THREAD]++;
        thread->proc = proc;
        thread->pid = current->pid;
        init_waitqueue_head(&thread->wait);
        INIT_LIST_HEAD(&thread->todo);
        rb_link_node(&thread->rb_node, parent, p);
        rb_insert_color(&thread->rb_node, &proc->threads);
        thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
        thread->return_error = BR_OK;
        thread->return_error2 = BR_OK;
    }
    return thread;
}
```

7.1.4　创建 Binder 实体对象

在 Binder 驱动程序函数 binder_ioctl 中，调用函数 binder_new_node 为 Service Manager 创建一个 Binder 实体对象。函数 binder_new_node 的具体实现代码如下所示：

```
static struct binder_node *
binder_new_node(struct binder_proc *proc, void __user *ptr, void __user *cookie)
{
    struct rb_node **p = &proc->nodes.rb_node;
    struct rb_node *parent = NULL;
    struct binder_node *node;

    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);

        if (ptr < node->ptr)
            p = &(*p)->rb_left;
        else if (ptr > node->ptr)
            p = &(*p)->rb_right;
        else
            return NULL;
```

```
    }

    node = kzalloc(sizeof(*node), GFP_KERNEL);
    if (node == NULL)
        return NULL;
    binder_stats.obj_created[BINDER_STAT_NODE]++;
    rb_link_node(&node->rb_node, parent, p);
    rb_insert_color(&node->rb_node, &proc->nodes);
    node->debug_id = ++binder_last_id;
    node->proc = proc;
    node->ptr = ptr;
    node->cookie = cookie;
    node->work.type = BINDER_WORK_NODE;
    INIT_LIST_HEAD(&node->work.entry);
    INIT_LIST_HEAD(&node->async_todo);
    if (binder_debug_mask & BINDER_DEBUG_INTERNAL_REFS)
        printk(KERN_INFO "binder: %d:%d node %d u%p c%p created\n",
              proc->pid, current->pid, node->debug_id,
              node->ptr, node->cookie);
    return node;
}
```

在上述代码中，需要先检查 proc→nodes 红黑树中是否已存在键值为 ptr 的 node，如果存在则返回 NULL。因为当前线程是第一次进入到这里，所以说是肯定不存在的，所以立即新创建一个 ptr 为 NULL 的 binder_node，并初始化其他成员变量，并将其插入到 proc→nodes 红黑树中。

7.1.5 尽职的循环

再看函数 main，接着调用函数 binder_loop 进入循环，并等待 Client 发送的请求。之所以将函数 binder_loop 称为“尽职”的循环，因为它总是围绕着 Binder 设备转，“非常敬业”。函数 binder_loop 在如下所示的文件中实现：

```
frameworks\base\cmds\servicemanager\binder.c
```

函数 binder_loop 的具体实现代码如下所示：

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
/*
  注意 binder_handler 参数，它是一个函数指针，binder_loop 读取请求后将解析
  这些请求，最后调用 binder_handler 完成最终的处理。
*/
    int res;
    struct binder_write_read bwr;
    unsigned readbuf[32];
//执行一个 ioctl 命令，这里的 bwr 参数各个成员的值：
    bwr.write_size = 0;
    bwr.write_consumed = 0;
    bwr.write_buffer = 0;
    //执行 BC_ENTER_LOOPER 命令
    readbuf[0] = BC_ENTER_LOOPER;
    //进入到函数 binder_write
    binder_write(bs, readbuf, sizeof(unsigned));
//进入 for 循环
    for (;;) {
        bwr.read_size = sizeof(readbuf);
        bwr.read_consumed = 0;
        bwr.read_buffer = (unsigned) readbuf;

        res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

        if (res < 0) {
            ALOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
            break;
        }
//接收到请求，交给 binder_parse，最终会调用 func 来处理这些请求
        res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
        if (res == 0) {
            ALOGE("binder_loop: unexpected reply?!\n");
```

```
            break;
        }
        if (res < 0) {
            ALOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

在上述代码中，往 binder_loop 中传递的函数指针是 svcmgr_handler，此指针在如下所示的文件中定义：

```
frameworks\base\cmds\servicemanager\Service_manager.c
```

svcmgr_handler 的具体实现代码如下所示：

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
    struct svcinfo *si;
    uint16_t *s;
    unsigned len;
    void *ptr;
    uint32_t strict_policy;
    int allow_isolated;
    if (txn->target != svcmgr_handle)
        return -1;
    strict_policy = bio_get_uint32(msg);
    s = bio_get_string16(msg, &len);
    if ((len != (sizeof(svcmgr_id) / 2)) ||
        memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
        fprintf(stderr,"invalid id %s\n", str8(s));
        return -1;
    }

    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = do_find_service(bs, s, len, txn->sender_euid);
        if (!ptr)
            break;
        bio_put_ref(reply, ptr);
        return 0;

    case SVC_MGR_ADD_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = bio_get_ref(msg);
        allow_isolated = bio_get_uint32(msg) ? 1 : 0;
        if (do_add_service(bs, s, len, ptr, txn->sender_euid, allow_isolated))
            return -1;
        break;

    case SVC_MGR_LIST_SERVICES: {
        unsigned n = bio_get_uint32(msg);

        si = svclist;
        while ((n-- > 0) && si)
            si = si->next;
        if (si) {
            bio_put_string16(reply, si->name);
            return 0;
        }
        return -1;
    }
    default:
        ALOGE("unknown code %d\n", txn->code);
        return -1;
    }
```

```
    bio_put_uint32(reply, 0);
    return 0;
}
```

在上述代码中，switch/case 语句将实现 IServiceManager 中定义的各个业务函数，我们重点看 do_add_service 这个函数，它最终完成了对 addService 请求的处理。

用户空间中的程序和 Binder 驱动程序交互功能，大多数都是通过 BINDER_WRITE_READ 命令实现的。write_bufffer 和 read_buffer 所指向的数据结构指定了具体要执行的操作，write_bufffer 和 read_buffer 指向的结构体是 binder_transaction_data，具体定义代码如下所示：

```
struct binder_transaction_data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
     * identifying the target and contents of the transaction.
     */
    union {
        size_t handle; * target descriptor of command transaction */
        void *ptr; /* target descriptor of return transaction */
    } target;
    void *cookie; /* target object cookie */
    unsigned int code; /* transaction command */

    /* General information about the transaction. */
    unsigned int flags;
    pid_t sender_pid;
    uid_t sender_euid;
    size_t data_size; /* number of bytes of data */
    size_t offsets_size; /* number of bytes of offsets */
    union {
        struct {
            /* transaction data */
            const void *buffer;
            /* offsets from buffer to flat_binder_object structs */
            const void *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

再来分析函数 binder_write，具体代码如下所示：

```
int binder_write(struct binder_state *bs, void *data, unsigned len)
{
    struct binder_write_read bwr;
    int res;
//write_size 大小为 4，表示 write_buffer 缓冲区大小为 4，它的内容是一个 BC_ENTER_LOOPER 命令协议号
    bwr.write_size = len;
    bwr.write_consumed = 0;
    bwr.write_buffer = (unsigned) data;// read_buffer 为空
    bwr.read_size = 0;
    bwr.read_consumed = 0;
    bwr.read_buffer = 0;
//调用 ioctl 函数进入到 Binder 驱动程序的 binder_ioctl 函数
    res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
    if (res < 0) {
        fprintf(stderr,"binder_write: ioctl failed (%s)\n",
                strerror(errno));
    }
    return res;
}
```

在上述代码中，调用函数 ioctl 进入到了 Binder 驱动程序函数 binder_ioctl，其中和 BC_ENTER_LOOPER 相关的代码如下所示：

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
```

```
    void __user *ubuf = (void __user *)arg;
    ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret)
        return ret;

    mutex_lock(&binder_lock);
    thread = binder_get_thread(proc);
    if (thread == NULL) {
        ret = -ENOMEM;
        goto err;
    }

    switch (cmd) {
    case BINDER_WRITE_READ: {
        struct binder_write_read bwr;
        if (size != sizeof(struct binder_write_read)) {
            ret = -EINVAL;
            goto err;
        }
//把用户传递进来的参数转换成 struct binder_write_read 结构体，并保存在本地变量 bwr 中
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
            printk(KERN_INFO "binder: %d:%d write %ld at %08lx, read %ld at %08lx\n",
            proc->pid, thread->pid, bwr.write_size, bwr.write_buffer, bwr.read_size,
            bwr.read_buffer);
        if (bwr.write_size > 0) {
            ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer,
            bwr.write_size, &bwr.write_consumed);
            if (ret < 0) {
                bwr.read_consumed = 0;
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (bwr.read_size > 0) {
            ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer,
            bwr.read_size, &bwr.read_consumed, filp->f_flags & O_NONBLOCK);
            if (!list_empty(&proc->todo))
                wake_up_interruptible(&proc->wait);
            if (ret < 0) {
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (binder_debug_mask & BINDER_DEBUG_READ_WRITE)
            printk(KERN_INFO "binder: %d:%d wrote %ld of %ld, read return %ld of %ld\n",
            proc->pid, thread->pid, bwr.write_consumed, bwr.write_size, bwr.read_
            consumed, bwr.read_size);
        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        break;
                        }
    ......
    default:
        ret = -EINVAL;
        goto err;
    }
    ret = 0;
err:
    if (thread)
        thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
    mutex_unlock(&binder_lock);
    wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
    if (ret && ret != -ERESTARTSYS)
```

```
        printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid,
current->pid, cmd, arg, ret);
    return ret;
}
```

在上述代码中，通过“copy_from_user(&bwr, ubuf, sizeof(bwr))”代码行将用户传递来的参数转换成 struct binder_write_read 结构，然后保存在本地变量 bwr 中，最后调用函数 binder_thread_write。其中和 BC_ENTER_LOOPER 相关的代码如下所示：

```
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;
    void __user *end = buffer + size;

    while (ptr < end && thread->return_error == BR_OK) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
            binder_stats.bc[_IOC_NR(cmd)]++;
            proc->stats.bc[_IOC_NR(cmd)]++;
            thread->stats.bc[_IOC_NR(cmd)]++;
        }
        switch (cmd) {
        case BC_ENTER_LOOPER:
            if (binder_debug_mask & BINDER_DEBUG_THREADS)
                printk(KERN_INFO "binder: %d:%d BC_ENTER_LOOPER\n",
                proc->pid, thread->pid);
            if (thread->looper & BINDER_LOOPER_STATE_REGISTERED) {
                thread->looper |= BINDER_LOOPER_STATE_INVALID;
                binder_user_error("binder: %d:%d ERROR:"
                    " BC_ENTER_LOOPER called after "
                    "BC_REGISTER_LOOPER\n",
                    proc->pid, thread->pid);
            }
            thread->looper |= BINDER_LOOPER_STATE_ENTERED;
            break;
        default:
            printk(KERN_ERR "binder: %d:%d unknown command %d\n", proc->pid, thread->pid, cmd);
            return -EINVAL;
        }
        *consumed = ptr - buffer;
    }
    return 0;
}
```

在函数 binder_ioctl 中，当 bwr.write_size 等于 0 时不会执行函数 binder_thread_write。如果 bwr.read_size 等于 32，则调用函数 binder_thread_read，相关的实现代码如下所示：

```
static int binder_thread_read(struct binder_proc *proc, struct binder_thread *thread,
                void __user *buffer, int size, signed long *consumed, int non_block)
{
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;

    int ret = 0;
    int wait_for_proc_work;

    if (*consumed == 0) {
        if (put_user(BR_NOOP, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
    }

retry:
    wait_for_proc_work = thread->transaction_stack == NULL && list_empty(&thread->todo);

    if (thread->return_error != BR_OK && ptr < end) {
```

```
        if (thread->return_error2 != BR_OK) {
            if (put_user(thread->return_error2, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            if (ptr == end)
                goto done;
            thread->return_error2 = BR_OK;
        }
        if (put_user(thread->return_error, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        thread->return_error = BR_OK;
        goto done;
    }


    thread->looper |= BINDER_LOOPER_STATE_WAITING;
    if (wait_for_proc_work)
        proc->ready_threads++;
    mutex_unlock(&binder_lock);
    if (wait_for_proc_work) {
        if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
            BINDER_LOOPER_STATE_ENTERED))) {
                binder_user_error("binder: %d:%d ERROR: Thread waiting "
                    "for process work before calling BC_REGISTER_"
                    "LOOPER or BC_ENTER_LOOPER (state %x)\n",
                    proc->pid, thread->pid, thread->looper);
                wait_event_interruptible(binder_user_error_wait,
                binder_stop_on_user_error < 2);
        }
        binder_set_nice(proc->default_priority);
        if (non_block) {
            if (!binder_has_proc_work(proc, thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work
            (proc, thread));
    } else {
        if (non_block) {
            if (!binder_has_thread_work(thread))
                ret = -EAGAIN;
        } else
            ret = wait_event_interruptible(thread->wait, binder_has_thread_work(thread));
    }
```

上述代码的运作流程如下所示。

- 因为传入的参数“*consumed == 0”，所以将值“BR_NOOP”写入到参数 ptr 指向的缓冲区中，也就是用户传进来的 bwr.read_buffer 缓冲区。
- thread->transaction_stack == NULL，并且 thread->todo 列表为空，表示当前线程没有需要处理的事务。
- wait_for_proc_work 为 true，表示查看 proc 是否有未处理的事务。
- 当前 thread->return_error == BR_OK，这是前面创建 binder_thread 时初始化设置的。
- 设置 thread 的状态为 BINDER_LOOPER_STATE_WAITING，表示线程处于等待状态。
- 调用函数 binder_set_nice 设置当前线程的优先级别为 proc->default_priority，需要将此 thread 的优先级别设置为和 proc 的一样。
- 此时 binder_has_proc_work(proc, thread)为 false，表示 proc 当前没有任何事务处理。
- 如果 non_block 为 true，表示文件打开模式为非阻塞模式，那么函数就直接返回-EAGAIN，用户需要重新执行 ioctl。

7.1.6　将信息注册到 ServiceManager

函数 do_add_service 的功能是，把自身信息注册到 ServiceManager 中。函数 do_add_service 在文件 service_manager.c 中定义，具体实现代码如下所示：

```
int do_add_service(struct binder_state *bs,
                   uint16_t *s, unsigned len,
                   void *ptr, unsigned uid, int allow_isolated)
{
    struct svcinfo *si;
    if (!ptr || (len == 0) || (len > 127))
        return -1;

    if (!svc_can_register(uid, s)) {
        ALOGE("add_service('%s',%p) uid=%d - PERMISSION DENIED\n",
             str8(s), ptr, uid);
        return -1;
    }

    si = find_svc(s, len);
    if (si) {
        if (si->ptr) {
            ALOGE("add_service('%s',%p) uid=%d - ALREADY REGISTERED, OVERRIDE\n", str8(s),
            ptr, uid);
            svcinfo_death(bs, si);
        }
        si->ptr = ptr;
    } else {
        si = malloc(sizeof(*si) + (len + 1) * sizeof(uint16_t));
        if (!si) {
            ALOGE("add_service('%s',%p) uid=%d - OUT OF MEMORY\n",
                 str8(s), ptr, uid);
            return -1;
        }
        si->ptr = ptr;
        si->len = len;
        memcpy(si->name, s, (len + 1) * sizeof(uint16_t));
        si->name[len] = '\0';
        si->death.func = svcinfo_death;
        si->death.ptr = si;
        si->allow_isolated = allow_isolated;
        si->next = svclist;
        svclist = si;
    }

    binder_acquire(bs, ptr);
    binder_link_to_death(bs, ptr, &si->death);
    return 0;
}
```

在上述函数 do_add_service 的代码中，函数 svc_can_register 用来判断注册服务的进程是否有权限的。此函数在文件 service_manager.c 中实现，具体实现代码如下所示：

```
int svc_can_register(unsigned uid, uint16_t *name)
{
    unsigned n;

    if ((uid == 0) || (uid == AID_SYSTEM))
        return 1;

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

    return 0;
}
```

其中结构数组 allowed 控制那些权限达不到 root 和 system 的进程，其定义代码如下所示：

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

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

由此可见，如果 Server 进程权限不够 root 和 system，那么必须在 allowed 中添加相应的项。ServiceManager 不过就是保存了一些服务的信息而已。

注意

为何需要一个 ServiceManager?

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

7.1.7　分析 MediaPlayerService 和 Client

在 Android 系统中，因为 ServiceManager 不是从 BnServiceManager 中派生的，所以之前没有讲请求数据是如何从通信层传递到业务层并进行处理的。在接下来的内容中，将以 MediaPlayer Service 和它的 Client 作为分析对象，讲解 ServiceManager 的管理过程。

一个 Client 想要得到某个 Service 的信息，就必须先和 ServiceManager 打交道，通过调用 getService 函数来获取对应 Service 的信息。读者请看来源于 IMediaDeathNotifier.cpp 中的例子 getMediaPlayerService()，此函数在如下所示的文件中实现：

```
frameworks\av\media\libmedia\IMediaDeathNotifier.cpp
```

函数 getMediaPlayerService()的具体实现代码如下所示：

```
IMediaDeathNotifier::getMediaPlayerService()
{
    ALOGV("getMediaPlayerService");
    Mutex::Autolock _l(sServiceLock);
    if (sMediaPlayerService == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.player"));
            if (binder != 0) {
                break;
            }
            ALOGW("Media player service not published, waiting...");
            usleep(500000); // 0.5 s
        } while (true);

        if (sDeathNotifier == NULL) {
        sDeathNotifier = new DeathNotifier();
    }
    binder->linkToDeath(sDeathNotifier);
    sMediaPlayerService = interface_cast<IMediaPlayerService>(binder);
    }
    ALOGE_IF(sMediaPlayerService == 0, "no media player service!?");
    return sMediaPlayerService;
}
```

这样有了 BpMediaPlayerService，就能够使用任何 IMediaPlayerService 提供的业务逻辑函数了。例如 createMediaRecorder 和 createMetadataRetriever 等。很显然，调用的这些函数都将把请求数据打包发送给 Binder 驱动，并由 BpBinder 中的 handle 值找到对应端的处理者来处理。这中间的过程如下所示：

（1）通信层接收到请求；

（2）递交给业务层处理。

MediaPlayerService 驻留在 MediaServer 进程中，这个进程有两个线程在 talkWithDriver 中。假设其中有一个线程收到了请求信息，它最终会通过 executeCommand 调用来处理这个请求。此函数在文件 IPCThreadState.cpp 中实现，具体实现代码如下所示：

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

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
                refs, obj, refs->refBase());
        obj->incStrong(mProcess.get());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
            obj->printRefs();
        }
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                "BR_RELEASE: object %p does not match cookie %p (expected %p)",
                refs, obj, refs->refBase());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
            obj->printRefs();
        }
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        mPendingWeakDerefs.push(refs);
        break;
```

```
case BR_ATTEMPT_ACQUIRE:
    refs = (RefBase::weakref_type*)mIn.readInt32();
    obj = (BBinder*)mIn.readInt32();

    {
        const bool success = refs->attemptIncStrong(mProcess.get());
        ALOG_ASSERT(success && refs->refBase() == obj,
                "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                refs, obj, refs->refBase());

        mOut.writeInt32(BC_ACQUIRE_RESULT);
        mOut.writeInt32((int32_t)success);
    }
    break;

case BR_TRANSACTION:
    {
        binder_transaction_data tr;
        result = mIn.read(&tr, sizeof(tr));
        ALOG_ASSERT(result == NO_ERROR,
            "Not enough command data for brTRANSACTION");
        if (result != NO_ERROR) break;

        Parcel buffer;
        buffer.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);

        const pid_t origPid = mCallingPid;
        const uid_t origUid = mCallingUid;

        mCallingPid = tr.sender_pid;
        mCallingUid = tr.sender_euid;

        int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
        if (gDisableBackgroundScheduling) {
            if (curPrio > ANDROID_PRIORITY_NORMAL) {
                setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
            }
        } else {
            if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                set_sched_policy(mMyThreadId, SP_BACKGROUND);
            }
        }
        Parcel reply;
        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_TRANSACTION thr " << (void*)pthread_self()
                << " / obj " << tr.target.ptr << " / code "
                << TypeCode(tr.code) << ": " << indent << buffer
                << dedent << endl
                << "Data addr = "
                << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                << ", offsets addr="
                << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
        }
        if (tr.target.ptr) {
            sp<BBinder> b((BBinder*)tr.cookie);
            const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
            if (error < NO_ERROR) reply.setError(error);

        } else {
            const status_t error = the_context_object->transact(tr.code, buffer,
            &reply, tr.flags);
            if (error < NO_ERROR) reply.setError(error);
        }
        if ((tr.flags & TF_ONE_WAY) == 0) {
            LOG_ONEWAY("Sending reply to %d!", mCallingPid);
```

```
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->getWeakRefs()->decWeak(proxy);
        } break;

    case BR_FINISHED:
        result = TIMED_OUT;
        break;

    case BR_NOOP:
        break;

    case BR_SPAWN_LOOPER:
        mProcess->spawnPooledThread(false);
        break;

    default:
        printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
        result = UNKNOWN_ERROR;
        break;
    }

    if (result != NO_ERROR) {
        mLastError = result;
    }

    return result;
}
```

BnMediaPlayerService 实现了 onTransact 函数，它将根据消息码调用对应的业务逻辑函数，这些业务逻辑函数由 MediaPlayerService 来实现。首先看文件 Binder.cpp 中的函数 onTransact，具体实现代码如下所示：

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

```
            break;
    }
    if (reply != NULL) {
        reply->setDataPosition(0);
    }
    return err;
}
```

接着看文件 IMediaPlayerService.cpp，里面的函数 onTransact 的具体实现代码如下所示：

```
status_t BnMediaPlayerService::onTransact(uint32_t
code, const Parcel& data,
Parcel* reply, uint32_t flags)
{
    switch(code) {
            case CREATE: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            pid_t pid = data.readInt32();
            sp<IMediaPlayerClient> client =
                interface_cast<IMediaPlayerClient>(data.readStrongBinder());
            int audioSessionId = data.readInt32();
            sp<IMediaPlayer> player = create(pid, client, audioSessionId);
            reply->writeStrongBinder(player->asBinder());
            return NO_ERROR;
        } break;
        case DECODE_URL: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            const char* url = data.readCString();
            uint32_t sampleRate;
            int numChannels;
            audio_format_t format;
            sp<IMemory> player = decode(url, &sampleRate, &numChannels, &format);
            reply->writeInt32(sampleRate);
            reply->writeInt32(numChannels);
            reply->writeInt32((int32_t) format);
            reply->writeStrongBinder(player->asBinder());
            return NO_ERROR;
        } break;
        case DECODE_FD: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            int fd = dup(data.readFileDescriptor());
            int64_t offset = data.readInt64();
            int64_t length = data.readInt64();
            uint32_t sampleRate;
            int numChannels;
            audio_format_t format;
            sp<IMemory> player = decode(fd, offset, length, &sampleRate, &numChannels,
            &format);
            reply->writeInt32(sampleRate);
            reply->writeInt32(numChannels);
            reply->writeInt32((int32_t) format);
            reply->writeStrongBinder(player->asBinder());
            return NO_ERROR;
        } break;
        case CREATE_MEDIA_RECORDER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            //从请求数据中解析对应的参数
            pid_t pid = data.readInt32();
            //子类要实现 createMediaRecorder 函数
            sp<IMediaRecorder> recorder = createMediaRecorder(pid);
            reply->writeStrongBinder(recorder->asBinder());
            return NO_ERROR;
        } break;
        case CREATE_METADATA_RETRIEVER: {
            CHECK_INTERFACE(IMediaPlayerService, data, reply);
            pid_t pid = data.readInt32();
            //子类要实现 createMetadataRetriever 函数
            sp<IMediaMetadataRetriever> retriever = createMetadataRetriever(pid);
            reply->writeStrongBinder(retriever->asBinder());
            return NO_ERROR;
        } break;
        default:
```

```
            return BBinder::onTransact(code, data, reply, flags);
        }
    }
```

7.2 获得 Service Manager 接口

在 Android 的 Binder 通信机制中，函数 defaultServiceManager 的功能是获取 Service Manager 远程接口，此函数在文件 frameworks\native\include\binder\IServiceManager.h 中定义，具体实现代码如下所示：

```
sp<IServiceManager>defaultServiceManager();
```

函数 defaultServiceManager 在文件 frameworks\native\libs\binder\IServiceManager.cpp 中定义，具体实现代码如下所示：

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

在上述代码中，gDefaultServiceManagerLock 和 gDefaultServiceManager 是全局变量，在文件 frameworks\native\libs\binder\Static.cpp 中定义，具体实现代码如下所示：

```
Mutex gDefaultServiceManagerLock;
sp<IServiceManager> gDefaultServiceManager;
```

当调用函数 defaultServiceManager 时，如果已经创建了 gDefaultServiceManager 则直接返回即可，否则通过 interface_cast<IServiceManager> (ProcessState::self()->getContextObject(NULL))创建一个，并保存在全局变量 gDefaultServiceManager 中。

在 Android 系统中，类 BpServiceManager 继承于类 BpInterface<IServiceManager>。类 BpInterface 是一个模板类，在文件 frameworks\base\include\binder\IInterface.h 中定义，具体代码如下所示：

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder* onAsBinder();
};
```

类 IServiceManager 继承于类 IInterface，而类 IInterface 和类 BpRefBase 又分别继承于类 RefBase。在类 BpRefBase 中，成员变量 mRemote 的类型为 IBinder*，表示一个 Binder 引用，引用句柄值保存在 BpBinder 类的 mHandle 成员变量中。类 BpBinder 通过类 IPCThreadState 和 Binder 驱动程序交互，而类 IPCThreadState 又通过它的成员变量 mProcess 来打开设备文件“/dev/binder”，mProcess 成员变量的类型为 ProcessState。

在创建 Service Manager 远程接口时需要先调用函数 ProcessState::self，此函数是 ProcessState 的静态成员函数，功能是返回一个全局唯一的 ProcessState 实例变量，此变量名为 gProcess。如果尚未创建 gProcess，则执行创建操作。在 ProcessState 的构造函数中，通过文件操作函数 open 打开设备文件/dev/binder，并将返回的设备文件描述符保存在成员变量 mDriverFD 中。然后调用函数 gProcess->getContextObject 获得一个句柄值为 0 的 Binder 引用（即 BpBinder）。由此可见，可

以将创建 Service Manager 远程接口的语句简化为下面的格式：

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

再看模板函数 interface_cast<IServiceManager>，此函数在文件 framework\base\include\binder\IInterface.h 中定义，具体实现代码如下所示：

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

在上述代码中调用了函数 IServiceManager::asInterface，此函数在类 IServiceManager 中通过宏 DECLARE_META_INTERFACE(ServiceManager) 声明，在文件 framework/base/include/binder/IService Manager.h 中的实现代码如下所示：

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

IServiceManager::asInterface 是通过宏 IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")定义实现的，在文件 framework\base\libs\binder\IServiceManager.cpp 中的具体实现代码如下所示：

```
#define IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")
    const android::String16 IServiceManager::descriptor("android.os.IServiceManager");
    const android::String16&
    IServiceManager::getInterfaceDescriptor() const {
    return IServiceManager::descriptor;
    }
    android::sp<IServiceManager> IServiceManager::asInterface(
    const android::sp<android::IBinder>& obj)
    {
    android::sp<IServiceManager> intr;
    if (obj != NULL) {
    intr = static_cast<IServiceManager*>(
    obj->queryLocalInterface(
    IServiceManager::descriptor).get());
    if (intr == NULL) {
    intr = new BpServiceManager(obj);
    }
    }
    return intr;
    }
    IServiceManager::IServiceManager() { }
    IServiceManager::~IServiceManager() { }
```

在上述代码中，传进来的参数 obj 就是则刚才创建的 new BpBinder(0)。类 BpBinder 中的成员函数 queryLocalInterface 继承于基类 IBinder，函数 IBinder::queryLocalInterface 在文件 framework\base\libs\binder\Binder.cpp 中定义，具体实现代码如下所示：

```
sp<IInterface>  IBinder::queryLocalInterface(const String16& descriptor)
{
    return NULL;
}
```

在函数 IServiceManager::asInterface 中会调用如下两行等价的代码：

```
intr = new BpServiceManager(obj);
intr = new BpServiceManager(new BpBinder(0));
```

最终以如下代码收尾，创建 Service Manager 远程接口工作全部完成：

```
gDefaultServiceManager = new BpServiceManager(new BpBinder(0));
```

7.3 分析 MessageQueue

类 MessageQueue 封装了与消息队列有关的操作，在一个以消息驱动的系统中，最重要的两部分就是消息队列和消息处理循环。在 Android 2.3 以前，只有 Java 部分才有资格向 MessageQueue 中添加消息以驱动 Java 世界的正常运转。从 Android 2.3 开始，MessageQueue 的核心部分被下移至 Native 层，这样让 Native 部分也能利用消息循环来处理它们的功能。Android 5.0 中的 MessageQueue 能够同时处理 Native 和 Java 两个部分的元素。

7.3.1 创建 MessageQueue

首先分析 MessageQueue 是如何同时处理 Native 和 Java 两个部分的元素的，首先看如下所示的文件：

```
frameworks\base\core\java\android\os\MessageQueue.java
```

在文件 MessageQueue.java 中，定义了如下所示的代码：

```
MessageQueue() {
    nativeInit(); //构造函数调用 nativeInit，该函数由 Native 层实现
}
```

在上述代码中，函数 nativeInit 的真正实现为函数 android_os_MessageQueue_nativeInit，此函数在如下所示的文件中实现：

```
frameworks\base\core\jni\android_os_MessageQueue.cpp
```

函数 android_os_MessageQueue_nativeInit 的具体实现代码如下所示。

```
static void android_os_MessageQueue_nativeInit(JNIEnv* env, jobject obj) {
    NativeMessageQueue* nativeMessageQueue = new NativeMessageQueue();
    if (!nativeMessageQueue) {
        jniThrowRuntimeException(env, "Unable to allocate native queue");
        return;
    }
    nativeMessageQueue->incStrong(env);
    android_os_MessageQueue_setNativeMessageQueue(env, obj, nativeMessageQueue);
}
```

函数 nativeInit 在 Native 层创建了一个与 MessageQueue 对应的 NativeMessageQueue 对象，其构造函数是 NativeMessageQueue，此函数在文件 android_os_MessageQueue.cpp 中实现，具体实现代码如下所示：

```
NativeMessageQueue::NativeMessageQueue() {
 /*
 代表消息循环的 Looper 也在 Native 层中出现了。一个线程会有一个 Looper 来循环处理消息队列中的消息。
下面一行的调用就是取得保存在线程本地存储空间（Thread Local Storage）中的 Looper 对象
 */
  mLooper = Looper::getForThread();
 if (mLooper == NULL) {
  /*
   如果第一次进来，则该线程没有设置本地存储，所以须先创建一个 Looper，然后再将其保存到 TLS 中，这是单
例模式
   */
    mLooper = new Looper(false);
    Looper::setForThread(mLooper);
  }
}
```

Native 的 Looper 是 Native 部分中参与消息循环的一位重要角色。虽然它的类名和 Java 层的 Looper 类一样，但此二者其实并无任何关系。

7.3.2 提取消息

当一切准备就绪后，在 Java 层中的 Looper 会在一个循环中提取并处理消息。消息的提取就

是调用 MessageQueue 的函数 next。当消息队列为空时，函数 next 就会阻塞。MessageQueue 同时支持 Java 层和 Native 层的事件。函数 next 在文件 MessagQueue.java 中定义，具体实现代码如下所示：

```
    final Message next() {
        int pendingIdleHandlerCount = -1;
        int nextPollTimeoutMillis = 0;
        for (;;) {
            if (nextPollTimeoutMillis != 0) {
                Binder.flushPendingCommands();
            }
//mPtr 保存了 NativeMessageQueue 的指针，调用 nativePollOnce 进行等待
            nativePollOnce(mPtr, nextPollTimeoutMillis);
            synchronized (this) {
                if (mQuiting) {
                    return null;
                }
                 //mMessages 用来存储消息，这里从其中取一个消息进行处理
                final long now = SystemClock.uptimeMillis();
                Message prevMsg = null;
                Message msg = mMessages;
                if (msg != null && msg.target == null) {
                    // Stalled by a barrier.  Find the next asynchronous message in the queue.
                    do {
                        prevMsg = msg;
                        msg = msg.next;
                    } while (msg != null && !msg.isAsynchronous());
                }
                if (msg != null) {
                    if (now < msg.when) {
                        // Next message is not ready.Set a timeout to wake up when it is ready.
                        nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_
                        VALUE);
                    } else {
                        // 获取信息.
                        mBlocked = false;
                        if (prevMsg != null) {
                            prevMsg.next = msg.next;
                        } else {
                            mMessages = msg.next;
                        }
                        msg.next = null;
                        if (false) Log.v("MessageQueue", "Returning message: " + msg);
                        msg.markInUse();
                        return msg; //返回一个 Message 给 Looper 进行派发和处理
                    }
                } else {
                    //没有信息了
                    nextPollTimeoutMillis = -1;
                }

               /*
                  处理注册的 IdleHandler，当 MessageQueue 中没有 Message 时，
                  Looper 会调用 IdleHandler 做一些工作，如做垃圾回收等
             */
                if (pendingIdleHandlerCount < 0
                        && (mMessages == null || now < mMessages.when)) {
                    pendingIdleHandlerCount = mIdleHandlers.size();
                }
                if (pendingIdleHandlerCount <= 0) {
                    // No idle handlers to run.  Loop and wait some more.
                    mBlocked = true;
                    continue;
                }

                if (mPendingIdleHandlers == null) {
                    mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandler
                    Count, 4)];
                }
                mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
```

```
            }

            // Run the idle handlers.
            // We only ever reach this code block during the first iteration.
            for (int i = 0; i < pendingIdleHandlerCount; i++) {
                final IdleHandler idler = mPendingIdleHandlers[i];
                mPendingIdleHandlers[i] = null; // release the reference to the handler

                boolean keep = false;
                try {
                    keep = idler.queueIdle();
                } catch (Throwable t) {
                    Log.wtf("MessageQueue", "IdleHandler threw exception", t);
                }

                if (!keep) {
                    synchronized (this) {
                        mIdleHandlers.remove(idler);
                    }
                }
            }

            // Reset the idle handler count to 0 so we do not run them again.
            pendingIdleHandlerCount = 0;

            // While calling an idle handler, a new message could have been delivered
            // so go back and look again for a pending message without waiting.
            nextPollTimeoutMillis = 0;
        }
    }
```

返回 nativePollOnce 后，函数 next 将从 mMessages 中提取一个消息。也就是说，要让 nativePollOnce 返回，至少要添加一个消息到消息队列，否则 nativePollOnce 只是做了一次无用功。如果 nativePollOnce 在 Native 层等待，就表明 Native 层也可以投递 Message，但是从类 Message 的实现代码上看，该类和 Native 层没有建立任何关系，即 Native 层不太可能去构造 Java 层的 Message 对象并把它插入到 Java 层的 Message 队列中。由此可见，nativePollOnce 除了等待 Java 层来的 Message，还在 Native 层做了大量的工作。

Java 层投递 Message 并触发 nativePollOnce 工作的正常流程如下所示。

（1）在 Java 层投递 Message

MessageQueue 中的函数 enqueueMessage 能够将一个 Message 投递到 MessageQueue 中的工作，此函数在文件 MesssageQueue.java 中实现，具体实现代码如下所示：

```
final boolean enqueueMessage(Message msg, long when) {
    if (msg.isInUse()) {
        throw new AndroidRuntimeException(msg + " This message is already in use.");
    }
    if (msg.target == null) {
        throw new AndroidRuntimeException("Message must have a target.");
    }

    boolean needWake;
    synchronized (this) {
        if (mQuiting) {
            RuntimeException e = new RuntimeException(
                    msg.target + " sending message to a Handler on a dead thread");
            Log.w("MessageQueue", e.getMessage(), e);
            return false;
        }

        /*
         如果 p 为空，表明消息队列中没有消息，那么 msg 将是第一个消息，needWake
         需要根据 mBlocked 的情况考虑是否触发
        */
        msg.when = when;
        Message p = mMessages;
        if (p == null || when == 0 || when < p.when) {
            // New head, wake up the event queue if blocked.
```

```
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {
            //如果 p 不为空，表明消息队列中还有剩余消息，需要将新的 msg 加到消息队列尾部
            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p;
            //因为消息队列之前还有剩余消息，所以这里不用调用 nativeWakeup
            prev.next = msg;
        }
    }
    if (needWake) {
    //调用 nativeWake，以触发 nativePollOnce 函数结束等待
        nativeWake(mPtr);
    }
    return true;
}
```

上述代码的主要功能是，将 Message 按执行的时间排序加入到消息队列中。根据情况调用 nativeWake 函数，以触发 nativePollOnce 函数，结束等待。

（2）分析函数 nativeWake

函数 nativeWake 在文件 android_os_MessageQueue.cpp 中定义，具体实现代码如下所示：

```
static void android_os_MessageQueue_nativeWake(JNIEnv* env, jobject obj, jint ptr) {
//取出 NativeMessageQueue 对象
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
//调用它的 wake 函数
    return nativeMessageQueue->wake();
}
void NativeMessageQueue::wake() {
  mLooper->wake();                          //层层调用，现在转到 mLooper 的 wake 函数
}
```

在 Native 层中，Looper 的函数 wake 在文件 Looper.cpp 中实现，具体实现代码如下所示：

```
void Looper::wake() {
  ssize_t nWrite;
     do {
         //向管道的写端写入一个字符
      nWrite = write(mWakeWritePipeFd, "W", 1);
  } while (nWrite == -1 && errno == EINTR);

  if (nWrite != 1) {
      if (errno != EAGAIN) {
          LOGW("Could not write wake signal, errno=%d", errno);
      }
  }
}
```

由此可见，函数 wake 的功能是向管道的写端写入一个字符“W”，这样管道的读端就会因为有数据可读而从等待状态中醒来。

7.3.3　分析函数 nativePollOnce

在 Android 5.0 中，nativePollOnce 的实现函数是 android_os_MessageQueue_nativePollOnce，在文件 android_os_MessageQueue.cpp 中定义，具体实现代码如下所示：

```
static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jobject obj,
```

```
        jint ptr, jint timeoutMillis)
    NativeMessageQueue* nativeMessageQueue =
                        reinterpret_cast<NativeMessageQueue*>(ptr);
    //取出 NativeMessageQueue 对象，并调用它的 pollOnce
    nativeMessageQueue->pollOnce(timeoutMillis);
}
//分析 pollOnce 函数
void NativeMessageQueue::pollOnce(int timeoutMillis) {
    mLooper->pollOnce(timeoutMillis); //将操作转发给 Mlooper 对象的 pollOnce 函数进行处理
}
```

Looper 中的函数 pollOnce 在文件 Looper.cpp 中定义，具体实现代码如下所示：

```
inline int pollOnce(int timeoutMillis) {
    return pollOnce(timeoutMillis, NULL, NULL, NULL);
}
```

在上述函数代码中，将调用另外一个有 4 个参数的函数 pollOnce，此函数的原型如下所示：

```
int pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData)
```

对上述参数的具体说明如下所示。

- 参数 timeoutMillis：表示超时等待时间。如果值为 - 1，则表示无限等待，直到有事件发生为止。如果值为 0，则无须等待立即返回。
- 参数 outFd：用来存储发生事件的那个文件描述符。
- 参数 outEvents：用来存储在该文件描述符上发生了哪些事件，目前支持可读、可写、错误和中断 4 个事件。这 4 个事件其实是从 epoll 事件转化而来的。后面我们会介绍大名鼎鼎的 epoll。
- 参数 outData：用于存储上下文数据，这个上下文数据是由用户在添加监听句柄时传递的，它的作用和 pthread_create 函数最后一个参数 param 一样，用来传递用户自定义的数据。

另外，函数 pollOnce 的返回值的具体说明如下所示。

- 当返回值为 ALOOPER_POLL_WAKE 时，表示这次返回是由 wake 函数触发的，也就是管道写端的那次写事件触发的。
- 当返回值为 ALOOPER_POLL_TIMEOUT 表示等待超时。
- 当返回值为 ALOOPER_POLL_ERROR 表示等待过程中发生错误。
- 当返回值为 ALOOPER_POLL_CALLBACK 表示某个被监听的句柄因某种原因被触发。这时，outFd 参数用于存储发生事件的文件句柄，outEvents 用于存储所发生的事件。

在文件 Looper.cpp 中定义了函数 pollOnce，具体实现代码如下所示：

```
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
    int result = 0;
    for (;;) { //这是一个无限循环
    //mResponses 是一个 Vector，这里首先需要处理 response
        while (mResponseIndex < mResponses.size()) {
            const Response& response = mResponses.itemAt(mResponseIndex++);
            int ident = response.request.ident; //ident 是这个 response 的 id
            if (ident >= 0) {
                int fd = response.request.fd;
                int events = response.events;
                void* data = response.request.data;
#if DEBUG_POLL_AND_WAKE
                ALOGD("%p ~ pollOnce - returning signalled identifier %d: "
                        "fd=%d, events=0x%x, data=%p",
                        this, ident, fd, events, data);
#endif
                if (outFd != NULL) *outFd = fd;
                if (outEvents != NULL) *outEvents = events;
                if (outData != NULL) *outData = data;
                //实际上，对于没有 callback 的 response，pollOnce 只是返回它的
                //ident，并没有实际做什么处理。因为没有 callback，所以系统也不知道如何处理
                return ident;
            }
        }

        if (result != 0) {
```

```
#if DEBUG_POLL_AND_WAKE
            ALOGD("%p ~ pollOnce - returning result %d", this, result);
#endif
            if (outFd != NULL) *outFd = 0;
            if (outEvents != NULL) *outEvents = 0;
            if (outData != NULL) *outData = NULL;
            return result;
        }
//调用 pollInner 函数。注意，它在 for 循环内部
        result = pollInner(timeoutMillis);
    }
}
```

接下来看文件 Looper.cpp 中的函数 pollInner，具体实现代码如下所示：

```
int Looper::pollInner(int timeoutMillis) {
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ pollOnce - waiting: timeoutMillis=%d", this, timeoutMillis);
#endif
    if (timeoutMillis != 0 && mNextMessageUptime != LLONG_MAX) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        int messageTimeoutMillis = toMillisecondTimeoutDelay(now, mNextMessageUptime);
        if (messageTimeoutMillis >= 0
                && (timeoutMillis < 0 || messageTimeoutMillis < timeoutMillis)) {
            timeoutMillis = messageTimeoutMillis;
        }
#if DEBUG_POLL_AND_WAKE
        ALOGD("%p ~ pollOnce - next message in %lldns, adjusted timeout: timeoutMillis=%d",
                this, mNextMessageUptime - now, timeoutMillis);
#endif
    }
    int result = ALOOPER_POLL_WAKE;
    mResponses.clear();
    mResponseIndex = 0;

    struct epoll_event eventItems[EPOLL_MAX_EVENTS];
    int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_EVENTS, timeoutMillis);

    // 获得锁
    mLock.lock();

    // Check for poll error.
    if (eventCount < 0) {
        if (errno == EINTR) {
            goto Done;
        }
        ALOGW("Poll failed with an unexpected error, errno=%d", errno);
        result = ALOOPER_POLL_ERROR;
        goto Done;
    }

    // Check for poll timeout.
    if (eventCount == 0) {
#if DEBUG_POLL_AND_WAKE
        ALOGD("%p ~ pollOnce - timeout", this);
#endif
        result = ALOOPER_POLL_TIMEOUT;
        goto Done;
    }

    // Handle all events.
#if DEBUG_POLL_AND_WAKE
    ALOGD("%p ~ pollOnce - handling events from %d fds", this, eventCount);
#endif

    for (int i = 0; i < eventCount; i++) {
        int fd = eventItems[i].data.fd;
        uint32_t epollEvents = eventItems[i].events;
        if (fd == mWakeReadPipeFd) {
            if (epollEvents & EPOLLIN) {
                awoken();
            } else {
```

```
                ALOGW("Ignoring unexpected epoll events 0x%x on wake read pipe.",
                epollEvents);
            }
        } else {
            ssize_t requestIndex = mRequests.indexOfKey(fd);
            if (requestIndex >= 0) {
                int events = 0;
                if (epollEvents & EPOLLIN) events |= ALOOPER_EVENT_INPUT;
                if (epollEvents & EPOLLOUT) events |= ALOOPER_EVENT_OUTPUT;
                if (epollEvents & EPOLLERR) events |= ALOOPER_EVENT_ERROR;
                if (epollEvents & EPOLLHUP) events |= ALOOPER_EVENT_HANGUP;
                pushResponse(events, mRequests.valueAt(requestIndex));
            } else {
                ALOGW("Ignoring unexpected epoll events 0x%x on fd %d that is "
                        "no longer registered.", epollEvents, fd);
            }
        }
    }
Done: ;

    // Invoke pending message callbacks.
    mNextMessageUptime = LLONG_MAX;
    while (mMessageEnvelopes.size() != 0) {
        nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
        const MessageEnvelope& messageEnvelope = mMessageEnvelopes.itemAt(0);
        if (messageEnvelope.uptime <= now) {
                sp<MessageHandler> handler = messageEnvelope.handler;
                Message message = messageEnvelope.message;
                mMessageEnvelopes.removeAt(0);
                mSendingMessage = true;
                mLock.unlock();

#if DEBUG_POLL_AND_WAKE || DEBUG_CALLBACKS
                ALOGD("%p ~ pollOnce - sending message: handler=%p, what=%d",
                        this, handler.get(), message.what);
#endif
                handler->handleMessage(message);
            } // release handler

            mLock.lock();
            mSendingMessage = false;
            result = ALOOPER_POLL_CALLBACK;
        } else {
            // The last message left at the head of the queue determines the next wakeup time.
            mNextMessageUptime = messageEnvelope.uptime;
            break;
        }
    }

    //释放锁
    mLock.unlock();

    // Invoke all response callbacks.
    for (size_t i = 0; i < mResponses.size(); i++) {
        Response& response = mResponses.editItemAt(i);
        if (response.request.ident == ALOOPER_POLL_CALLBACK) {
            int fd = response.request.fd;
            int events = response.events;
            void* data = response.request.data;
#if DEBUG_POLL_AND_WAKE || DEBUG_CALLBACKS
            ALOGD("%p ~ pollOnce - invoking fd event callback %p: fd=%d, events=0x%x,
            data=%p",
                    this, response.request.callback.get(), fd, events, data);
#endif
            int callbackResult = response.request.callback->handleEvent(fd, events, data);
            if (callbackResult == 0) {
                removeFd(fd);
            }
            // Clear the callback reference in the response structure promptly because we
            // will not clear the response vector itself until the next poll.
            response.request.callback.clear();
```

```
            result = ALOOPER_POLL_CALLBACK;
        }
    }
    return result;
}
```

上述代码的运作流程如下所示。

（1）计算一下真正需要等待的时间。

（2）调用函数 epoll_wait 等待。

（3）返回函数 epoll_wait，此时可能有以下 3 种情况。

- 发生错误，则跳转到 Done 处。
- 超时，也跳转到 Done 处。
- epoll_wait 监测到某些文件句柄上有事件发生。

（4）假设 epoll_wait 因为文件句柄有事件而返回，此时需要根据文件句柄来分别处理，具体情况如下所示。

- 如果是管道读端发生事件，则认为是控制命令，可以直接读取管道中的数据。
- 如果是其他 fd 发生事件，则根据 Request 构造 Response，并 push 到 Response 数组中。

（5）在有 Done 标志的位置真正开始处理事件。首先处理 Native 的 Message，然后调用 Native Handler 的 handleMessage 处理该 Message，最后处理 Response 数组中那些带有 callback 的事件。

在上面的处理流程中涉及了 mRequests，其功能是添加监控请求。添加监控请求其实就是调用 epoll_ctl 增加文件句柄。下面以 Native 的 Activity 中的一段代码为例来分析 mRequests，这段代码在如下所示的文件中定义：

```
frameworks\base\core\jni\android_app_NativeActivity.cpp
```

具体代码如下所示：

```
static jint
loadNativeCode_native(JNIEnv* env, jobject clazz, jstring path, jstring funcName,
        jobject messageQueue, jstring internalDataDir, jstring obbDir,
        jstring externalDataDir, int sdkVersion,
        jobject jAssetMgr, jbyteArray savedState)
{
    LOG_TRACE("loadNativeCode_native");

    const char* pathStr = env->GetStringUTFChars(path, NULL);
    NativeCode* code = NULL;

    void* handle = dlopen(pathStr, RTLD_LAZY);

    env->ReleaseStringUTFChars(path, pathStr);

    if (handle != NULL) {
        const char* funcStr = env->GetStringUTFChars(funcName, NULL);
        code = new NativeCode(handle, (ANativeActivity_createFunc*)
                dlsym(handle, funcStr));
        env->ReleaseStringUTFChars(funcName, funcStr);

        if (code->createActivityFunc == NULL) {
            ALOGW("ANativeActivity_onCreate not found");
            delete code;
            return 0;
        }

        code->messageQueue = android_os_MessageQueue_getMessageQueue(env, messageQueue);
        if (code->messageQueue == NULL) {
            ALOGW("Unable to retrieve native MessageQueue");
            delete code;
            return 0;
        }

        int msgpipe[2];
        if (pipe(msgpipe)) {
            ALOGW("could not create pipe: %s", strerror(errno));
```

```
            delete code;
            return 0;
        }
        code->mainWorkRead = msgpipe[0];
        code->mainWorkWrite = msgpipe[1];
        int result = fcntl(code->mainWorkRead, F_SETFL, O_NONBLOCK);
        SLOGW_IF(result != 0, "Could not make main work read pipe "
                "non-blocking: %s", strerror(errno));
        result = fcntl(code->mainWorkWrite, F_SETFL, O_NONBLOCK);
        SLOGW_IF(result != 0, "Could not make main work write pipe "
                "non-blocking: %s", strerror(errno));
        code->messageQueue->getLooper()->addFd(
                code->mainWorkRead, 0, ALOOPER_EVENT_INPUT, mainWorkCallback, code);

        code->ANativeActivity::callbacks = &code->callbacks;
        if (env->GetJavaVM(&code->vm) < 0) {
            ALOGW("NativeActivity GetJavaVM failed");
            delete code;
            return 0;
        }
        code->env = env;
        code->clazz = env->NewGlobalRef(clazz);

        const char* dirStr = env->GetStringUTFChars(internalDataDir, NULL);
        code->internalDataPathObj = dirStr;
        code->internalDataPath = code->internalDataPathObj.string();
        env->ReleaseStringUTFChars(internalDataDir, dirStr);

        dirStr = env->GetStringUTFChars(externalDataDir, NULL);
        code->externalDataPathObj = dirStr;
        code->externalDataPath = code->externalDataPathObj.string();
        env->ReleaseStringUTFChars(externalDataDir, dirStr);

        code->sdkVersion = sdkVersion;

        code->assetManager = assetManagerForJavaObject(env, jAssetMgr);

        dirStr = env->GetStringUTFChars(obbDir, NULL);
        code->obbPathObj = dirStr;
        code->obbPath = code->obbPathObj.string();
        env->ReleaseStringUTFChars(obbDir, dirStr);

        jbyte* rawSavedState = NULL;
        jsize rawSavedSize = 0;
        if (savedState != NULL) {
            rawSavedState = env->GetByteArrayElements(savedState, NULL);
            rawSavedSize = env->GetArrayLength(savedState);
        }

        code->createActivityFunc(code, rawSavedState, rawSavedSize);

        if (rawSavedState != NULL) {
            env->ReleaseByteArrayElements(savedState, rawSavedState, 0);
        }
    }

    return (jint)code;
}
```

在上述代码中调用了类 Looper 的函数 addFd。其中第一个参数表示监听的 fd；第二个参数 0 表示 ident；第三个参数表示需要监听的事件，这里只监听可读事件；第四个参数为回调函数，当该 fd 发生指定事件时，looper 将回调该函数；第五个参数 code 为回调函数的参数。

类 Looper 中的函数 addFd 在文件 Looper.cpp 中定义，具体实现代码如下所示：

```
int Looper::addFd(int fd, int ident, int events, const sp<LooperCallback>& callback, void*
data) {
#if DEBUG_CALLBACKS
    ALOGD("%p ~ addFd - fd=%d, ident=%d, events=0x%x, callback=%p, data=%p", this, fd,
ident, events, callback.get(), data);
```

```
#endif

    if (!callback.get()) {
        if (! mAllowNonCallbacks) {
            ALOGE("Invalid attempt to set NULL callback but not allowed for this looper.");
            return -1;
        }

        if (ident < 0) {
            ALOGE("Invalid attempt to set NULL callback with ident < 0.");
            return -1;
        }
    } else {
        ident = ALOOPER_POLL_CALLBACK;
    }

    int epollEvents = 0;
    if (events & ALOOPER_EVENT_INPUT) epollEvents |= EPOLLIN;
    if (events & ALOOPER_EVENT_OUTPUT) epollEvents |= EPOLLOUT;

    { // 获得锁
        AutoMutex _l(mLock);

        Request request;
        request.fd = fd;
        request.ident = ident;
        request.callback = callback;
        request.data = data;

        struct epoll_event eventItem;
        memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data
        field union
        eventItem.events = epollEvents;
        eventItem.data.fd = fd;

        ssize_t requestIndex = mRequests.indexOfKey(fd);
        if (requestIndex < 0) {
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, & eventItem);
            if (epollResult < 0) {
                ALOGE("Error adding epoll events for fd %d, errno=%d", fd, errno);
                return -1;
            }
            mRequests.add(fd, request);
        } else {
            int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_MOD, fd, & eventItem);
            if (epollResult < 0) {
                ALOGE("Error modifying epoll events for fd %d, errno=%d", fd, errno);
                return -1;
            }
            mRequests.replaceValueAt(requestIndex, request);
        }
    } //释放锁
    return 1;
}
```

在函数 pollInner 中，当某个监控 fd 上发生事件后，就会调用对应的 Request，对应的代码如下所示：

```
pushResponse(events, mRequests.itemAt(i));
```

函数 PushResponse 在文件 Looper.cpp 中实现，具体实现代码如下所示：

```
void Looper::pushResponse(int events, const Request& request) {
  Response response;
  response.events = events;
  response.request = request; //保存所发生的事情和对应的 request
  mResponses.push(response); //保存到 mResponse 数组
}
```

PoolInner 并不是单独处理 request，而是需要先收集 request，等到 Native Message 消息处理完之后再做处理，这说明 Native Message 的处理逻辑优先级高于监控 fd 的优先级。

在 Android 4.0 以前的版本中，只有 Java 层才可以通过 sendMessage 往 MessageQueue 中添加消息。而从 Android 4.0 开始，Native 层也支持 sendMessage 了。函数 sendMessage 在文件 Looper.cpp 中定义，具体实现代码如下所示：

```
void Looper::sendMessage(const sp<MessageHandler>& handler, const Message& message) {
    //Native 的 sendMessage 函数必须同时传递一个 Handler
    nsecs_t now = systemTime(SYSTEM_TIME_MONOTONIC);
    //调用 sendMessageAtTime
    sendMessageAtTime(now, handler, message);
}
```

在上述代码中，调用了文件 Looper.cpp 中的函数 sendMessageAtTime，此函数的具体实现代码如下所示：

```
void Looper::sendMessageAtTime(nsecs_t uptime, const sp<MessageHandler>& handler,
        const Message& message) {
#if DEBUG_CALLBACKS
    ALOGD("%p ~ sendMessageAtTime - uptime=%lld, handler=%p, what=%d",
            this, uptime, handler.get(), message.what);
#endif

    size_t i = 0;
    { // 获得 lock
        AutoMutex _l(mLock);
        //按时间排序，将消息插到正确的位置上
        size_t messageCount = mMessageEnvelopes.size();
        //按时间排序，将消息插到正确的位置上
        while (i < messageCount && uptime >= mMessageEnvelopes.itemAt(i).uptime) {
            i += 1;
        }

        MessageEnvelope messageEnvelope(uptime, handler, message);
        mMessageEnvelopes.insertAt(messageEnvelope, i, 1);

        //mSendingMessage 和 Java 层中的那个 mBlocked 一样，是一个小小的优化措施
        if (mSendingMessage) {
            return;
        }
    } // 释放 lock

    // //唤醒 epoll_wait，让它处理消息
    if (i == 0) {
        wake();
    }
}
```

第 8 章　init 进程和 Zygote 进程

在运行 Android 应用程序之后，首先会启动 init 进程，这个进程是 Linux 系统中用户空间的第一个进程，其进程号为 1。在 Android 系统中，Zygote 进程被称为“孵化”进程或“孕育”进程，功能和 Linux 系统的 fork 类似，用于“孕育”产生出不同的子进程。在本章的内容中，将详细讲解 Android 5.0 中 init 进程和 Zygote 进程的源代码，为读者步入本书后面高级知识的学习打下基础。

8.1 分析 init 进程

大家应该知道，Android 本质上就是一个基于 Linux 内核的操作系统。与 Linux 和 Fedora Linux 最大的区别是，Android 在应用层专门为移动设备添加了一些特有的支持。目前，Linux 有很多通信机制可以在用户空间和内核空间之间交互，例如，设备驱动文件（位于/dev 目录中）、内存文件（/proc、/sys 目录等）。Android 在加载 Linux 基本内核后，就开始运行一个初始化进程 init。从 Android 加载 Linux 内核时设置了如下所示的参数：

```
Kernelcommand line: noinitrd root=/dev/nfs console=ttySAC0 init=/initnfsroot=192.168.
1.103:/nfsbootip=192.168.1.20:192.168.1.103:192.168.1.1:255.255.255.0::eth0:on
```

在上述命令中，告诉 Linux 内核初始化完成后开始运行 init 进程，由于 init 进程就放在系统根目录下面，init 进程的代码位于源代码的目录“system/core/init”下面。在分析 init 的核心代码之前，还需要做如下所示的工作：

- 初始化属性；
- 处理配置文件的命令（主要是 init.rc 文件），包括处理各种 Action；
- 性能分析（使用 bootchart 工具）；
- 无限循环执行 command（启动其他的进程）。

init 程序并不是由一个源代码文件组成的，而是由一组源代码文件的目标文件链接而成的。这些文件位于如下所示的目录中：

```
/system/core/init
```

主要的 JNI 代码放在以下的路径中：

```
frameworks/base/core/jni/
```

另外，还涉及了其他目录中的如下几个文件：

```
\bionic\libc\bionic\libc_init_common.h
\bionic\libc\bionic\libc_init_common.c
\bionic\libc\bionic\libc_init_dynamic.c
\bionic\libc\bionic\libc_init_static.c
system\core\libcutils\properties.c
```

在本章的内容中，将仔细地分析 init 进程的启动过程，了解 Android 系统是怎么启动起来的。

8.1.1　分析入口函数

进程 init 入口函数是 main，具体实现文件的路径是：

```
system\core\init\init.c
```

函数 main 的实现非常复杂，从这个 main 函数可以看出，init 实际上就分为如下两部分功能。

（1）初始化：主要包括建立/dev、/proc 等目录，初始化属性，执行 init.rc 等初始化文件中的 action 等。

（2）使用 for 循环无限循环建立子进程。

上述两项工作是 init 的核心。

在 Linux 系统中，init 是一切应用空间进程的父进程。所以，平常在 Linux 终端执行的命令，并建立进程，实际上都是在这个无限的 for 循环中完成的。也就是说，在 Linux 终端执行 ps -e 命令后，看到的所有除了 init 外的其他进程，都是由 init 负责创建的。当然，如果 init 崩溃了，那么 Linux 系统就会崩溃。

函数 main 的具体实现代码如下所示：

```
int main(int argc, char **argv)
{
    int fd_count = 0;
    struct pollfd ufds[4];
    char *tmpdev;
    char* debuggable;
    char tmp[32];
    int property_set_fd_init = 0;
    int signal_fd_init = 0;
    int keychord_fd_init = 0;
    bool is_charger = false;

    if (!strcmp(basename(argv[0]), "ueventd"))
        return ueventd_main(argc, argv);

    if (!strcmp(basename(argv[0]), "watchdogd"))
        return watchdogd_main(argc, argv);
    umask(0);
    //  下面的代码开始建立各种用户空间的目录，如/dev、/proc、/sys 等
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);

        /* 检测/dev/.booting 文件是否可读写和创建*/
    close(open("/dev/.booting", O_WRONLY | O_CREAT, 0000));

    open_devnull_stdio();
    klog_init();
    //  初始化属性
    property_init();

    get_hardware_name(hardware, &revision);
    //  处理内核命令行
    process_kernel_cmdline();
    ……

    is_charger = !strcmp(bootmode, "charger");

    INFO("property init\n");
    if (!is_charger)
        property_load_boot_defaults();

    INFO("reading config file\n");
    //  分析/init.rc 文件的内容
    init_parse_config_file("/init.rc");
    ……//  执行初始化文件中的动作
    action_for_each_trigger("init", action_add_queue_tail);
    //  在 charger 模式下略过 mount 文件系统的工作
    if (!is_charger) {
        action_for_each_trigger("early-fs", action_add_queue_tail);
```

```
        action_for_each_trigger("fs", action_add_queue_tail);
        action_for_each_trigger("post-fs", action_add_queue_tail);
        action_for_each_trigger("post-fs-data", action_add_queue_tail);
    }

    queue_builtin_action(property_service_init_action, "property_service_init");
    queue_builtin_action(signal_init_action, "signal_init");
    queue_builtin_action(check_startup_action, "check_startup");

    if (is_charger) {
        action_for_each_trigger("charger", action_add_queue_tail);
    } else {
        action_for_each_trigger("early-boot", action_add_queue_tail);
        action_for_each_trigger("boot", action_add_queue_tail);
    }

        /* run all property triggers based on current state of the properties */
    queue_builtin_action(queue_property_triggers_action, "queue_property_triggers");


#if BOOTCHART
    queue_builtin_action(bootchart_init_action, "bootchart_init");
#endif
    // 进入无限循环，建立 init 的子进程（init 是所有进程的父进程）
    for(;;) {
        int nr, i, timeout = -1;
        // 执行命令（子进程对应的命令）
        execute_one_command();
        restart_processes();

        if (!property_set_fd_init && get_property_set_fd() > 0) {
            ufds[fd_count].fd = get_property_set_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            property_set_fd_init = 1;
        }
        if (!signal_fd_init && get_signal_fd() > 0) {
            ufds[fd_count].fd = get_signal_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            signal_fd_init = 1;
        }
        if (!keychord_fd_init && get_keychord_fd() > 0) {
            ufds[fd_count].fd = get_keychord_fd();
            ufds[fd_count].events = POLLIN;
            ufds[fd_count].revents = 0;
            fd_count++;
            keychord_fd_init = 1;
        }

        if (process_needs_restart) {
            timeout = (process_needs_restart - gettime()) * 1000;
            if (timeout < 0)
                timeout = 0;
        }

        if (!action_queue_empty() || cur_action)
            timeout = 0;
// bootchart 是一个性能统计工具，用于搜集硬件和系统的信息，并将其写入磁盘，以便其
// 他程序使用
#if BOOTCHART
        if (bootchart_count > 0) {
            if (timeout < 0 || timeout > BOOTCHART_POLLING_MS)
                timeout = BOOTCHART_POLLING_MS;
            if (bootchart_step() < 0 || --bootchart_count == 0) {
                bootchart_finish();
                bootchart_count = 0;
            }
        }
```

```
#endif
        // 等待下一个命令的提交
        nr = poll(ufds, fd_count, timeout);
        if (nr <= 0)
            continue;

        for (i = 0; i < fd_count; i++) {
            if (ufds[i].revents == POLLIN) {
                if (ufds[i].fd == get_property_set_fd())
                    handle_property_set_fd();
                else if (ufds[i].fd == get_keychord_fd())
                    handle_keychord();
                else if (ufds[i].fd == get_signal_fd())
                    handle_signal();
            }
        }
    }

    return 0;
}
```

8.1.2 分析配置文件

在 init 进程中，配置文件是指文件 init.rc，其路径是：

```
system\core\rootdir\init.rc
```

在本节的内容中，将详细剖析配置文件 init.rc 的源代码。

1. init.rc 简介

文件 init.rc 是一个可配置的初始化文件，在里面定制了厂商可以配置的额外的初始化配置信息，具体格式为：

```
init.%PRODUCT%.rc
```

文件 init.rc 是在文件/system/core/init/init.c 中读取的，它基于“行”，包含一些用空格隔开的关键字（它属于特殊字符）。如果在关键字中含有空格，则使用“/”表示转义，使用“”防止关键字被断开，如果“/”在末尾则表示换行，以“#”开头的表示注释。

文件 init.rc 包含 4 种状态类别，分别是 Actions、Commands、Services 和 Options，当声明一个 Service 或者 Action 的时候，它将隐式声明一个 section，它之后跟随的 Command 或者 Option 都将属于这个 section。另外，Action 和 Service 不能重名，否则忽略为 error。

（1）Actions

Actions 就是在某种条件下触发一系列的命令，通常有一个 trigger，形式如：

```
on <trigger>
    <command>
    <command>
```

（2）Service

Service 的结构如下所示：

```
service <name> <pathname> [ <argument> ]*
   <option>
   <option>
```

（3）Option

Option 是 Service 的修饰词，主要包括如下所示的选项。

- critical：表示如果服务在 4 分钟内存在多于 4 次，则系统重启到 recovery mode。
- disabled：表示服务不会自动启动，需要手动调用名字启动。
- setEnv <name> <value>：设置启动环境变量。
- socket <name> <type> <permission> [<user> [<group>]]：开启一个 unix 域的 socket，名字为/dev/socket/<name>，<type>只能是 dgram 或者 stream，<user>和<group>默认为 0。

- user <username>：表示将用户切换为<username>，用户名已经定义好了，只能是 system/root。
- group <groupname>：表示将组切换为<groupname>。
- oneshot：表示这个 Service 只启动一次。
- class <name>：指定一个要启动的类，这个类中如果有多个 service，将会被同时启动。默认的 class 将会是“default”。
- onrestart：在重启时执行一条命令。

（4）trigger 主要包括如下所示的选项。

- boot：当/init.conf 加载完毕时。
- <name>=<value>：当<name>被设置为<value>时。
- device-added-<path>：设备<path>被添加时。
- device-removed-<path>：设备<path>被移除时。
- service-exited-<name>：服务<name>退出时。

（5）主要包含的操作命令及具体说明如下所示。

- exec <path> [<argument>]*：执行一个<path>指定的程序。
- export <name> <value>：设置一个全局变量。
- ifup <interface>：使网络接口<interface>连接。
- import <filename>：引入其他的配置文件。
- hostname <name>：设置主机名。
- chdir <directory>：切换工作目录。
- chmod <octal-mode> <path>：设置访问权限。
- chown <owner> <group> <path>：设置用户和组。
- chroot <directory>：设置根目录
- class_start <serviceclass>：启动类中的 service。
- class_stop <serviceclass>：停止类中的 service。
- domainname <name>：设置域名。
- insmod <path>：安装模块。
- mkdir <path> [mode] [owner] [group] ：创建一个目录，并可以指定权限，用户和组。
- mount <type> <device> <dir> [<mountoption>]* ：加载指定设备到目录下。
- <mountoption>：包括“ro”“rw”“remount”“noatime”。
- setprop <name> <value>：设置系统属性。
- setrlimit <resource> <cur> <max>：设置资源访问权限。
- start <service>：开启服务。
- stop <service>：停止服务。
- symlink <target> <path>：创建一个动态链接。
- sysclktz <mins_west_of_gmt>：设置系统时钟。
- trigger <event>：触发事件。
- write <path> <string> [<string>]*：向<path>路径的文件写入多个<string>。

读者可以查找 Android 的 shell，会看到根目录中有一个 init.rc 文件。启动 Android 后，会将文件 init.rc 装载到内存。而修改文件 init.rc 的内容实际上只是修改内存中的 init.rc 文件的内容。一旦重启 Android，文件 init.rc 的内容又会恢复到最初的装载。想彻底修改文件 init.rc 内容，唯一方式是修改 Android 的 ROM 中的内核镜像（boot.img）。

2. 分析 init.rc 的过程

文件 init.rc 是一个配置文件，内部有许多的语言规则，所有语言会在函数 init_parse_config_file 中进行解析。前面的主函数 main 读取完配置文件 init.rc 后，会调用函数 parse_config 进行解析。整个实现流程如下所示：

```
init_parse_config_file->read_file->parse_config
```

（1）函数 parse_config 和 init_parse_config_file 在如下文件中实现：

```
system\core\init\init_parser.c
```

函数 parse_config 和 init_parse_config_file 的具体实现代码如下所示：

```
static void parse_config(const char *fn, char *s)//s 为 init.rc 中字符串的内容
{
    struct parse_state state;
    char *args[INIT_PARSER_MAXARGS];
    int nargs;

    nargs = 0;
    state.filename = fn;
    state.line = 1;
    state.ptr = s;
    state.nexttoken = 0;
    state.parse_line = parse_line_no_op;
    for (;;) {
        switch (next_token(&state)) {
        case T_EOF:       //文件的结尾
            state.parse_line(&state, 0, 0);
            return;
        case T_NEWLINE://新的一行
            if (nargs) {
                int kw = lookup_keyword(args[0]);//读取 init.rc 返回关键字,如 service,返回 K_service
                if (kw_is(kw, SECTION)) {   //查看关键字是否为 SECTION,只有 service 和 on 满足
                    state.parse_line(&state, 0, 0);
                    parse_new_section(&state, kw, nargs, args);
                } else {
                    state.parse_line(&state, nargs, args);//on 和 service 两个段下面的内容
                }
                nargs = 0;
            }
            break;
        case T_TEXT://文本内容
            if (nargs < INIT_PARSER_MAXARGS) {
                args[nargs++] = state.text;
            }
            break;
        }
    }
}

int init_parse_config_file(const char *fn)
{
    char *data;
    data = read_file(fn, 0);
    if (!data) return -1;

    parse_config(fn, data);
    DUMP();
    return 0;
}
```

通过上述代码可以看出，在 for 的无限循环中对文件 init.rc 的内容进行了解析，以一行一行的形式进行了读取。

（2）每读取完一行内容换行到下一行时，使用函数 lookup_keyword 分析已经读取的一行的第一个参数。函数 lookup_keyword 的具体实现代码如下所示：

```
int lookup_keyword(const char *s)
{
    switch (*s++) {
    case 'c':
    if (!strcmp(s, "opy")) return K_copy;
        if (!strcmp(s, "apability")) return K_capability;
        if (!strcmp(s, "hdir")) return K_chdir;
        if (!strcmp(s, "hroot")) return K_chroot;
        if (!strcmp(s, "lass")) return K_class;
```

```
        if (!strcmp(s, "lass_start")) return K_class_start;
        if (!strcmp(s, "lass_stop")) return K_class_stop;
        if (!strcmp(s, "lass_reset")) return K_class_reset;
        if (!strcmp(s, "onsole")) return K_console;
        if (!strcmp(s, "hown")) return K_chown;
        if (!strcmp(s, "hmod")) return K_chmod;
        if (!strcmp(s, "ritical")) return K_critical;
        break;
    case 'd':
        if (!strcmp(s, "isabled")) return K_disabled;
        if (!strcmp(s, "omainname")) return K_domainname;
        break;
    case 'e':
        if (!strcmp(s, "xec")) return K_exec;
        if (!strcmp(s, "xport")) return K_export;
        break;
    case 'g':
        if (!strcmp(s, "roup")) return K_group;
        break;
    case 'h':
        if (!strcmp(s, "ostname")) return K_hostname;
        break;
    case 'i':
        if (!strcmp(s, "oprio")) return K_ioprio;
        if (!strcmp(s, "fup")) return K_ifup;
        if (!strcmp(s, "nsmod")) return K_insmod;
        if (!strcmp(s, "mport")) return K_import;
        break;
    case 'k':
        if (!strcmp(s, "eycodes")) return K_keycodes;
        break;
    case 'l':
        if (!strcmp(s, "oglevel")) return K_loglevel;
        if (!strcmp(s, "oad_persist_props")) return K_load_persist_props;
        break;
    case 'm':
        if (!strcmp(s, "kdir")) return K_mkdir;
        if (!strcmp(s, "ount_all")) return K_mount_all;
        if (!strcmp(s, "ount")) return K_mount;
        break;
    case 'o':
        if (!strcmp(s, "n")) return K_on;
        if (!strcmp(s, "neshot")) return K_oneshot;
        if (!strcmp(s, "nrestart")) return K_onrestart;
        break;
    case 'r':
        if (!strcmp(s, "estart")) return K_restart;
        if (!strcmp(s, "estorecon")) return K_restorecon;
        if (!strcmp(s, "mdir")) return K_rmdir;
        if (!strcmp(s, "m")) return K_rm;
        break;
    case 's':
        if (!strcmp(s, "eclabel")) return K_seclabel;
        if (!strcmp(s, "ervice")) return K_service;
        if (!strcmp(s, "etcon")) return K_setcon;
        if (!strcmp(s, "etenforce")) return K_setenforce;
        if (!strcmp(s, "etenv")) return K_setenv;
        if (!strcmp(s, "etkey")) return K_setkey;
        if (!strcmp(s, "etprop")) return K_setprop;
        if (!strcmp(s, "etrlimit")) return K_setrlimit;
        if (!strcmp(s, "etsebool")) return K_setsebool;
        if (!strcmp(s, "ocket")) return K_socket;
        if (!strcmp(s, "tart")) return K_start;
        if (!strcmp(s, "top")) return K_stop;
        if (!strcmp(s, "ymlink")) return K_symlink;
        if (!strcmp(s, "ysclktz")) return K_sysclktz;
        break;
    case 't':
        if (!strcmp(s, "rigger")) return K_trigger;
        break;
    case 'u':
```

```
        if (!strcmp(s, "ser")) return K_user;
        break;
    case 'w':
        if (!strcmp(s, "rite")) return K_write;
        if (!strcmp(s, "ait")) return K_wait;
        break;
    }
    return K_UNKNOWN;
}
```

由此可见，函数 lookup_keyword 主要对每一行的第一个字符做 case 判断，然后在 if 语句中调用 strcmp 命令，这些命令都是按照文件 init.rc 的格式要求进行的。比如常用的 service 和 on 等经过 lookup_keyword 后返回 K_servcie 和 K_on。随后使用 kw_is(kw, SECTION)判断返回的 kw 是不是属于 SECTION 类型。在文件 init.rc 中，只有 service 和 on 满足该类型。

（3）定义关键字

在文件 keywords.h 中定义了 init 使用的关键字，在此文件中定义了如 do_class_start、do_class_stop 之类的函数，并且还定义了枚举。文件 keywords.h 的路径如下所示：

```
system\core\init\
```

文件 keywords.h 的具体实现代码如下所示：

```
#ifndef KEYWORD
int do_chroot(int nargs, char **args);
int do_chdir(int nargs, char **args);
int do_class_start(int nargs, char **args);
int do_class_stop(int nargs, char **args);
int do_class_reset(int nargs, char **args);
……
#define __MAKE_KEYWORD_ENUM__
#define KEYWORD(symbol, flags, nargs, func) K_##symbol,
enum {
    K_UNKNOWN,
#endif
    KEYWORD(capability,  OPTION,  0, 0)
    KEYWORD(chdir,       COMMAND, 1, do_chdir)
    KEYWORD(chroot,      COMMAND, 1, do_chroot)
    KEYWORD(class,       OPTION,  0, 0)
    KEYWORD(class_start, COMMAND, 1, do_class_start)
    KEYWORD(class_stop,  COMMAND, 1, do_class_stop)
    KEYWORD(class_reset, COMMAND, 1, do_class_reset)
    KEYWORD(console,     OPTION,  0, 0)
    KEYWORD(critical,    OPTION,  0, 0)
    KEYWORD(disabled,    OPTION,  0, 0)
    KEYWORD(domainname,  COMMAND, 1, do_domainname)
    KEYWORD(exec,        COMMAND, 1, do_exec)
    KEYWORD(export,      COMMAND, 2, do_export)
……
    KEYWORD(load_persist_props,    COMMAND, 0, do_load_persist_props)
    KEYWORD(ioprio,      OPTION,  0, 0)
#ifdef __MAKE_KEYWORD_ENUM__
    KEYWORD_COUNT,
};
#undef __MAKE_KEYWORD_ENUM__
#undef KEYWORD
#endif
```

文件 keywords.h 在文件 init_parser.c 中被用到了两次，具体引用代码如下所示：

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

#include "keywords.h"

#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"
};
#undef KEYWORD

#define kw_is(kw, type) (keyword_info[kw].flags & (type))
#define kw_name(kw) (keyword_info[kw].name)
#define kw_func(kw) (keyword_info[kw].func)
#define kw_nargs(kw) (keyword_info[kw].nargs)
```

8.1.3　分析 Service

由前面的函数 lookup_keyword 可知，在调用过程中会对 on 和 service 所在的段进行解析，我们这里首先分析 Service，在分析时以文件 init.rc 中的 service zygote 为例。

1. Zygote 对应的 Service Action

在文件 init.rc 中，Zygote 对应的 Service Action 的代码如下所示：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

解析 action 的入口函数是 parse_new_section，在此函数中再分别对 service 或者 on 关键字开头的内容进行解析。函数 parse_new_section 的具体实现代码如下所示：

```
void parse_new_section(struct parse_state *state, int kw,
                       int nargs, char **args)
{
    printf("[ %s %s ]\n", args[0],
           nargs > 1 ? args[1] : "");
    switch(kw) {
    case K_service:
        state->context = parse_service(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_service;
            return;
        }
        break;
    case K_on:
        state->context = parse_action(state, nargs, args);
        if (state->context) {
            state->parse_line = parse_line_action;
            return;
        }
        break;
    case K_import:
        parse_import(state, nargs, args);
        break;
    }
    state->parse_line = parse_line_no_op;
}
```

2. init 组织 Service

在 init 进程中，使用了一个名为 Service 的结构体保存和 service action 有关的信息。此结构体是在如下文件中定义的：

system\core\init\init.h

结构体 Service 的具体实现代码如下所示：

```
struct service {
       /* list of all services */
    struct listnode slist;

    const char *name;
    const char *classname;

    unsigned flags;
    pid_t pid;
    time_t time_started;    /* time of last start */
    time_t time_crashed;    /* first crash within inspection window */
    int nr_crashed;         /* number of times crashed within window */

    uid_t uid;
    gid_t gid;
    gid_t supp_gids[NR_SVC_SUPP_GIDS];
    size_t nr_supp_gids;

#ifdef HAVE_SELINUX
    char *seclabel;
#endif

    struct socketinfo *sockets;
    struct svcenvinfo *envvars;

    struct action onrestart;  /* Actions to execute on restart. */

    /* keycodes for triggering this service via /dev/keychord */
    int *keycodes;
    int nkeycodes;
    int keychord_id;

    int ioprio_class;
    int ioprio_pri;

    int nargs;
    /* "MUST BE AT THE END OF THE STRUCT" */
    char *args[1];
}; /*     ^-------'args' MUST be at the end of this struct! */
```

另外，在文件 init.h 中还定义了结构体 action，具体实现代码如下所示：

```
struct action {
       /* node in list of all actions */
    struct listnode alist;
       /* node in the queue of pending actions */
    struct listnode qlist;
       /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};
```

这样通过上述两个结构体对 Service 进行了组织。

3. 分析 Service 用到的函数

解析 Service 会用到的两个函数，分别是 parse_service 和 parse_line_service。

（1）当解析文件 init.rc 中的 service zygote 时会执行函数 parse_service，此函数的功能是构建 Service 的骨架，对 Service 关键字开头的内容进行解析。函数 parse_service 的具体实现代码如下所示：

```
static void *parse_service(struct parse_state *state, int nargs, char **args)
{
    struct service *svc;
    if (nargs < 3) {
        parse_error(state, "services must have a name and a program\n");
        return 0;
    }
    if (!valid_name(args[1])) {
        parse_error(state, "invalid service name '%s'\n", args[1]);
        return 0;
    }

    svc = service_find_by_name(args[1]);//查找服务是否已经存在
    if (svc) {
        parse_error(state, "ignored duplicate definition of service '%s'\n", args[1]);
        return 0;
    }

    nargs -= 2;
    svc = calloc(1, sizeof(*svc) + sizeof(char*) * nargs);
    if (!svc) {
        parse_error(state, "out of memory\n");
        return 0;
    }
    svc->name = args[1];          //Sevice 的名字
    svc->classname = "default"; //svc 的类名默认是 default
    memcpy(svc->args, args + 2, sizeof(char*) * nargs);//首个参数放的是可执行文件
    svc->args[nargs] = 0;
    svc->nargs = nargs;//参数个数
    svc->onrestart.name = "onrestart";
    list_init(&svc->onrestart.commands);
    list_add_tail(&service_list, &svc->slist);
    return svc;
}
```

在上述代码中，agrs[1]就是 zygote，系统会先查找是否已经存在该服务，然后构建一个 service svc 并进行相关的填充，包括服务名、服务所属的类别名字和服务启动带入的参数个数（要减去 Service 和服务名 zygote），最后将这个 svc 加入到 service_list 全局链表中。

（2）函数 parse_line_service 的功能是，根据配置文件的内容填充 service 结构体，并解析 Service 中剩余行中的 Option，如 class、socket、onrestart 等。函数 parse_line_service 的具体实现代码如下所示：

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

    if (nargs == 0) {
        return;
    }

    svc->ioprio_class = IoSchedClass_NONE;

    kw = lookup_keyword(args[0]);
    switch (kw) {
    case K_capability:
        break;
    case K_class:
        if (nargs != 2) {
            parse_error(state, "class option requires a classname\n");
        } else {
            svc->classname = args[1];
        }
        break;
    case K_console:
        svc->flags |= SVC_CONSOLE;
        break;
    case K_disabled:
```

```
        svc->flags |= SVC_DISABLED;
        svc->flags |= SVC_RC_DISABLED;
        break;
    case K_ioprio:
        if (nargs != 3) {
            parse_error(state, "ioprio optin usage: ioprio <rt|be|idle> <ioprio 0-7>\n");
        } else {
            svc->ioprio_pri = strtoul(args[2], 0, 8);

            if (svc->ioprio_pri < 0 || svc->ioprio_pri > 7) {
                parse_error(state, "priority value must be range 0 - 7\n");
                break;
            }

            if (!strcmp(args[1], "rt")) {
                svc->ioprio_class = IoSchedClass_RT;
            } else if (!strcmp(args[1], "be")) {
                svc->ioprio_class = IoSchedClass_BE;
            } else if (!strcmp(args[1], "idle")) {
                svc->ioprio_class = IoSchedClass_IDLE;
            } else {
                parse_error(state, "ioprio option usage: ioprio <rt|be|idle> <0-7>\n");
            }
        }
        break;
    case K_group:
        if (nargs < 2) {
            parse_error(state, "group option requires a group id\n");
        } else if (nargs > NR_SVC_SUPP_GIDS + 2) {
            parse_error(state, "group option accepts at most %d supp. groups\n",
                        NR_SVC_SUPP_GIDS);
        } else {
            int n;
            svc->gid = decode_uid(args[1]);
            for (n = 2; n < nargs; n++) {
                svc->supp_gids[n-2] = decode_uid(args[n]);
            }
            svc->nr_supp_gids = n - 2;
        }
        break;
    case K_keycodes:
        if (nargs < 2) {
            parse_error(state, "keycodes option requires atleast one keycode\n");
        } else {
            svc->keycodes = malloc((nargs - 1) * sizeof(svc->keycodes[0]));
            if (!svc->keycodes) {
                parse_error(state, "could not allocate keycodes\n");
            } else {
                svc->nkeycodes = nargs - 1;
                for (i = 1; i < nargs; i++) {
                    svc->keycodes[i - 1] = atoi(args[i]);
                }
            }
        }
        break;
    case K_oneshot:
        svc->flags |= SVC_ONESHOT;
        break;
    case K_onrestart:
        nargs--;
        args++;
        kw = lookup_keyword(args[0]);
        if (!kw_is(kw, COMMAND)) {
            parse_error(state, "invalid command '%s'\n", args[0]);
            break;
        }
        kw_nargs = kw_nargs(kw);
        if (nargs < kw_nargs) {
            parse_error(state, "%s requires %d %s\n", args[0], kw_nargs - 1,
                kw_nargs > 2 ? "arguments" : "argument");
            break;
```

```
        }

        cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
        cmd->func = kw_func(kw);
        cmd->nargs = nargs;
        memcpy(cmd->args, args, sizeof(char*) * nargs);
        list_add_tail(&svc->onrestart.commands, &cmd->clist);
        break;
    case K_critical:
        svc->flags |= SVC_CRITICAL;
        break;
    case K_setenv: { /* name value */
        struct svcenvinfo *ei;
        if (nargs < 2) {
            parse_error(state, "setenv option requires name and value arguments\n");
            break;
        }
        ei = calloc(1, sizeof(*ei));
        if (!ei) {
            parse_error(state, "out of memory\n");
            break;
        }
        ei->name = args[1];
        ei->value = args[2];
        ei->next = svc->envvars;
        svc->envvars = ei;
        break;
    }
    case K_socket: {/* name type perm [ uid gid ] */
        struct socketinfo *si;
        if (nargs < 4) {
            parse_error(state, "socket option requires name, type, perm arguments\n");
            break;
        }
        if (strcmp(args[2],"dgram") && strcmp(args[2],"stream")
                && strcmp(args[2],"seqpacket")) {
            parse_error(state, "socket type must be 'dgram', 'stream' or 'seqpacket'\n");
            break;
        }
        si = calloc(1, sizeof(*si));
        if (!si) {
            parse_error(state, "out of memory\n");
            break;
        }
        si->name = args[1];
        si->type = args[2];
        si->perm = strtoul(args[3], 0, 8);
        if (nargs > 4)
            si->uid = decode_uid(args[4]);
        if (nargs > 5)
            si->gid = decode_uid(args[5]);
        si->next = svc->sockets;
        svc->sockets = si;
        break;
    }
    case K_user:
        if (nargs != 2) {
            parse_error(state, "user option requires a user id\n");
        } else {
            svc->uid = decode_uid(args[1]);
        }
        break;
    case K_seclabel:
#ifdef HAVE_SELINUX
        if (nargs != 2) {
            parse_error(state, "seclabel option requires a label string\n");
        } else {
            svc->seclabel = args[1];
        }
#endif
        break;
```

```
    default:
        parse_error(state, "invalid option '%s'\n", args[0]);
    }
}
```

8.1.4 解析 on 字段的内容

在本节的内容中将详细剖析 on 字段的内容，以 on boot 这个 section 作为列子进行分析。希望读者认真学习，为步入本书后面知识的学习打下基础。

1. zygote 对应的 on action

字段 on 的内容比较复杂，此处，将以 on boot 这个 section 作为列子进行分析。在文件 init.rc 中，zygote 对应的 on boot action 的代码如下所示：

```
on boot
# basic network init
    ifup lo
    hostname localhost
    domainname localdomain

# set RLIMIT_NICE to allow priorities from 19 to -20
    setrlimit 13 40 40

# Memory management.  Basic kernel parameters, and allow the high
# level system server to be able to adjust the kernel OOM driver
# parameters to match how it is managing things.
    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    chown root system /sys/module/lowmemorykiller/parameters/adj
    chmod 0664 /sys/module/lowmemorykiller/parameters/adj
    chown root system /sys/module/lowmemorykiller/parameters/minfree
    chmod 0664 /sys/module/lowmemorykiller/parameters/minfree

    # Tweak background writeout
    write /proc/sys/vm/dirty_expire_centisecs 200
    write /proc/sys/vm/dirty_background_ratio  5

    # Permissions for System Server and daemons.
    chown radio system /sys/android_power/state
    chown radio system /sys/android_power/request_state
    chown radio system /sys/android_power/acquire_full_wake_lock
    chown radio system /sys/android_power/acquire_partial_wake_lock
    chown radio system /sys/android_power/release_wake_lock
    chown system system /sys/power/autosleep
    chown system system /sys/power/state
    chown system system /sys/power/wakeup_count
    chown radio system /sys/power/wake_lock
    chown radio system /sys/power/wake_unlock
    chmod 0660 /sys/power/state
    chmod 0660 /sys/power/wake_lock
    chmod 0660 /sys/power/wake_unlock
……
# Set this property so surfaceflinger is not started by system_init
    setprop system_init.startsurfaceflinger 0

    class_start core
    class_start main
```

和前面对 Service 的分析类似，case 中进入 K_on 选项执行函数 parse_action。函数 parse_action 的具体实现代码如下所示：

```
static void *parse_action(struct parse_state *state, int nargs, char **args)
{
    struct action *act;
    if (nargs < 2) {
        parse_error(state, "actions must have a trigger\n");
        return 0;
```

```
    }
    if (nargs > 2) {
        parse_error(state, "actions may not have extra parameters\n");
        return 0;
    }
    act = calloc(1, sizeof(*act));
    act->name = args[1];
    list_init(&act->commands);
    list_add_tail(&action_list, &act->alist);
        /* XXX add to hash */
    return act;
}
```

2. init 组织 on

在 init 进程中可以看到一个名为 action 结构体，它类似于 Service，这个 action 的名字为 boot，最后会将这个 action 加入到全局链表 action_list 中。结构体 action 的具体实现代码如下所示：

```
struct action {
        /* node in list of all actions */
    struct listnode alist;
        /* node in the queue of pending actions */
    struct listnode qlist;
        /* node in list of actions for a trigger */
    struct listnode tlist;

    unsigned hash;
    const char *name;

    struct listnode commands;
    struct command *current;
};
```

3. 解析 on 用到的函数

在解析 on 时会用到函数 parse_line_action，功能是对 on 字段所在的 option 进行解析。函数 parse_line_action 的具体实现代码如下所示：

```
static void parse_line_action(struct parse_state* state, int nargs, char **args)
//action 所在的行
{
    struct command *cmd;
    struct action *act = state->context;//on boot 启动
    int (*func)(int nargs, char **args);
    int kw, n;

    if (nargs == 0) {
        return;
    }

    kw = lookup_keyword(args[0]);//命令的参数个数
    if (!kw_is(kw, COMMAND)) {
        parse_error(state, "invalid command '%s'\n", args[0]);
        return;
    }

    n = kw_nargs(kw);
    if (nargs < n) {
        parse_error(state, "%s requires %d %s\n", args[0], n - 1,
            n > 2 ? "arguments" : "argument");
        return;
    }
    cmd = malloc(sizeof(*cmd) + sizeof(char*) * nargs);
    cmd->func = kw_func(kw);
    cmd->nargs = nargs;
    memcpy(cmd->args, args, sizeof(char*) * nargs);
    list_add_tail(&act->commands, &cmd->clist);   //
}
```

到此为止，on 和 service 两个 section 的分析工作全部完成。

8.1.5 init 控制 Service

在 Android 5.0 系统中，当进行 init 进程初始化工作时，除了对系统做一些必要的初始化操作，还需要启动 Service。而 Service 是在 init 脚本中定义的，所以很有必要了解一下在 init 中对 Service 进行控制的知识。

1. 启动 Zygote

Android 系统是基于 Linux 内核的，而在 Linux 系统中的所有进程都是 init 进程的子进程或孙进程。也就是说，所有的进程都是直接或者间接地由 init 进程 fork 出来的。Zygote 进程也不例外，它是在系统启动的过程，由 init 进程创建的。在系统启动脚本文件 system/core /rootdir/init.rc 中，可以看到如下启动 Zygote 进程的脚本命令：

```
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
    onrestart restart media
    onrestart restart netd
```

在上述代码中，各个关键字的具体说明如下所示。

- service：用于通知 init 进程创建一个名为“zygote”的进程，这个 Zygote 进程要执行的程序是/system/bin/app_process，后面是要传给 app_process 的参数。
- Socket:表示这个 zygote 进程需要一个名称为“zygote”的 Socket 资源，这样启动系统后，就可以在/dev/socket 目录下看到有一个名为 Zygote 的文件。这里定义的 Socket 的类型为 unix domain Socket，它是用来做本地进程间通信用的。

2. 启动 Service

首先看函数 do_class_start，此函数的功能是启动 Service，此函数在如下文件中定义：

```
system\core\init\builtins.c
```

函数 do_class_start 的具体实现代码如下所示：

```
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;
}
```

在上述代码中，调用了函数 service_start_if_not_disabled 实现启动功能，此函数也在文件 builtins.c 中实现，具体实现代码如下所示：

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

在上述代码中，调用了函数 service_start 实现启动功能。函数 service_start 是整个启动功能的核心，在文件 init.c 中定义，具体实现代码如下所示：

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

```
    int n;
#ifdef HAVE_SELINUX
    char *scon = NULL;
    int rc;
#endif
        /* starting a service removes it from the disabled or reset
         * state and immediately takes it out of the restarting
         * state if it was in there
         */
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
    svc->time_started = 0;
    if (svc->flags & SVC_RUNNING) {
        return;
    }

    needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
    if (needs_console && (!have_console)) {
        ERROR("service '%s' requires console\n", svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (stat(svc->args[0], &s) != 0) {
        ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if ((!(svc->flags & SVC_ONESHOT)) && dynamic_args) {
        ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
              svc->args[0]);
        svc->flags |= SVC_DISABLED;
        return;
    }

#ifdef HAVE_SELINUX
    if (is_selinux_enabled() > 0) {
        char *mycon = NULL, *fcon = NULL;

        INFO("computing context for service '%s'\n", svc->args[0]);
        rc = getcon(&mycon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            return;
        }

        rc = getfilecon(svc->args[0], &fcon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            freecon(mycon);
            return;
        }

        rc = security_compute_create(mycon, fcon, string_to_security_class("process"), &scon);
        freecon(mycon);
        freecon(fcon);
        if (rc < 0) {
            ERROR("could not get context while starting '%s'\n", svc->name);
            return;
        }
    }
#endif

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
```

```
        int fd, sz;

        umask(077);
#ifdef __arm__
        int current = personality(0xffffFFFF);
        personality(current | ADDR_COMPAT_LAYOUT);
#endif
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);

#ifdef HAVE_SELINUX
        setsockcreatecon(scon);
#endif

        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }

#ifdef HAVE_SELINUX
        freecon(scon);
        scon = NULL;
        setsockcreatecon(NULL);
#endif

        if (svc->ioprio_class != IoSchedClass_NONE) {
            if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
                ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
                      getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
            }
        }

        if (needs_console) {
            setsid();
            open_console();
        } else {
            zap_stdio();
        }

#if 0
        for (n = 0; svc->args[n]; n++) {
            INFO("args[%d] = '%s'\n", n, svc->args[n]);
        }
        for (n = 0; ENV[n]; n++) {
            INFO("env[%d] = '%s'\n", n, ENV[n]);
        }
#endif

        setpgid(0, getpid());

    /* as requested, set our gid, supplemental gids, and uid */
        if (svc->gid) {
            if (setgid(svc->gid) != 0) {
                ERROR("setgid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->nr_supp_gids) {
            if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
```

```
                ERROR("setgroups failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->uid) {
            if (setuid(svc->uid) != 0) {
                ERROR("setuid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }

#ifdef HAVE_SELINUX
        if (svc->seclabel) {
            if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
                ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
                _exit(127);
            }
        }
#endif

        if (!dynamic_args) {
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
            int arg_idx = svc->nargs;
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
        }
        _exit(127);
    }

#ifdef HAVE_SELINUX
    freecon(scon);
#endif

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

在函数 service_start 中，参数 dynamic_args 只有当 Service 的 option 中有 oneshot 时才会用到，此时会通过替换掉启动服务的命令参数启动服务。因为 Service 的 option 会记录在 struct service 中，所以，在启动 Service 时只需考虑到这些选项即可。同时，会记录下 Service 的 pid 和状态等信息。

3. 总结 4 种启动 Service 的方式

其实在 init 进程中，可以使用如下所示的方式启动 Service。

（1）在 action 下面添加和启动服务相关的 command，在 action 中和操作服务相关的命令有。

- class_start <serviceclass> #：启动所有指定 class 的服务。
- class_stop <serviceclass> #：停止所有指定 class 的服务，后续没法通过 class_start 启动。
- class_reset <serviceclass> #：停止服务，后续可以通过 class_start 启动。
- restart <servicename> #：重启指定名称的服务，先 stop，再 start。
- start <servicename> #：启动指定名称的服务。
- stop <servicename> #：停止指定名称的服务。

在启动 command 时，在文件 init.c 的主函数 main 中，通过使用“for(;;)”循环执行函数 execute_one_command 的方式实现，此函数的具体实现代码如下所示：

```
void execute_one_command(void)
{
    int ret;

    if (!cur_action || !cur_command || is_last_command(cur_action, cur_command)) {
        cur_action = action_remove_queue_head();
        cur_command = NULL;
        if (!cur_action)
            return;
        INFO("processing action %p (%s)\n", cur_action, cur_action->name);
        cur_command = get_first_command(cur_action);
    } else {
        cur_command = get_next_command(cur_action, cur_command);
    }

    if (!cur_command)
        return;

    ret = cur_command->func(cur_command->nargs, cur_command->args);//执行 class_start 等
    INFO("command '%s' r=%d\n", cur_command->args[0], ret);
}
```

（2）使用函数 restart_processes 和 restart_service_if_needed 重启 Service，该函数位于 init 的主线程循环中，功能是查看是否有需要重新启动的 Service。在文件 init.c 中，函数 restart_processes 和 restart_service_if_needed 的具体实现代码如下所示：

```
static void restart_service_if_needed(struct service *svc)
{
    time_t next_start_time = svc->time_started + 5;

    if (next_start_time <= gettime()) {
        svc->flags &= (~SVC_RESTARTING);
        service_start(svc, NULL);
        return;
    }

    if ((next_start_time < process_needs_restart) ||
        (process_needs_restart == 0)) {
        process_needs_restart = next_start_time;
    }
}

static void restart_processes()
{
    process_needs_restart = 0;
    service_for_each_flags(SVC_RESTARTING,
                           restart_service_if_needed);
}
```

在重启过程中，会重启 flag 为 SVC_RESTARTING 的服务。这部分进程的重启其实在 init 由 handle_signal 来管理，一旦出现 Service 崩溃，函数 poll 会接收到相关文件变化的信息，并执行 handle_signal 中的函数 wait_for_one_process。函数 wait_for_one_process 的具体实现代码如下所示：

```
static int wait_for_one_process(int block)
{
    pid_t pid;
```

```
    int status;
    struct service *svc;
    struct socketinfo *si;
    time_t now;
    struct listnode *node;
    struct command *cmd;

    while ( (pid = waitpid(-1, &status, block ? 0 : WNOHANG)) == -1 && errno == EINTR );
    if (pid <= 0) return -1;
    INFO("waitpid returned pid %d, status = %08x\n", pid, status);

    svc = service_find_by_pid(pid);
    if (!svc) {
        ERROR("untracked pid %d exited\n", pid);
        return 0;
    }
.....
 svc->flags |= SVC_RESTARTING;

    /* Execute all onrestart commands for this service. */
    list_for_each(node, &svc->onrestart.commands) {
        cmd = node_to_item(node, struct command, clist);
        cmd->func(cmd->nargs, cmd->args);
    }
    notify_service_state(svc->name, "restarting");
    return 0;
}
```

在上述代码中，使用 waitpid 找到子进程退出的进程号 pid，然后查找到该 Service，对 Service 中的 onrestart 这个 commands 进行操作。同时将 Service 的 flag 设置为 SVC_RESTARTING，这样就结合前面讲到的 restart_processes 便重新启动了该服务进程。

（3）在文件 system\core\init\property_service.c 中，使用函数 handle_property_set_f 向 socket 中名称为 property_service 的属性服务发送控制的消息，这样便可以进入该函数中。函数 handle_property_set_f 的具体实现代码如下所示：

```
void handle_property_set_fd()
{
    prop_msg msg;
    int s;
    int r;
    int res;
    struct ucred cr;
    struct sockaddr_un addr;
    socklen_t addr_size = sizeof(addr);
    socklen_t cr_size = sizeof(cr);
    char * source_ctx = NULL;

    if ((s = accept(property_set_fd, (struct sockaddr *) &addr, &addr_size)) < 0) {
        return;
    }

    /* Check socket options here */
    if (getsockopt(s, SOL_SOCKET, SO_PEERCRED, &cr, &cr_size) < 0) {
        close(s);
        ERROR("Unable to recieve socket options\n");
        return;
    }

    r = TEMP_FAILURE_RETRY(recv(s, &msg, sizeof(msg), 0));
    if(r != sizeof(prop_msg)) {
        ERROR("sys_prop: mis-match msg size recieved: %d expected: %d errno: %d\n",
              r, sizeof(prop_msg), errno);
        close(s);
        return;
    }

    switch(msg.cmd) {
    case PROP_MSG_SETPROP:
        msg.name[PROP_NAME_MAX-1] = 0;
```

```
        msg.value[PROP_VALUE_MAX-1] = 0;

#ifdef HAVE_SELINUX
        getpeercon(s, &source_ctx);
#endif

        if(memcmp(msg.name,"ctl.",4) == 0) {
            // Keep the old close-socket-early behavior when handling
            // ctl.* properties.
            close(s);
            if (check_control_perms(msg.value, cr.uid, cr.gid, source_ctx)) {
                handle_control_message((char*) msg.name + 4, (char*) msg.value);
            } else {
                ERROR("sys_prop: Unable to %s service ctl [%s] uid:%d gid:%d pid:%d\n",
                      msg.name + 4, msg.value, cr.uid, cr.gid, cr.pid);
            }
        } else {
            if (check_perms(msg.name, cr.uid, cr.gid, source_ctx)) {
                property_set((char*) msg.name, (char*) msg.value);
            } else {
                ERROR("sys_prop: permission denied uid:%d  name:%s\n",
                      cr.uid, msg.name);
            }
            close(s);
        }
#ifdef HAVE_SELINUX
        freecon(source_ctx);
#endif

        break;

    default:
        close(s);
        break;
    }
}
```

（4）使用函数 handle_keychord 启动，该函数和 chorded keyboard 有关，功能是处理注册在 Service structure 上的 keychord，通常是启动 Service。函数 handle_keychord 在文件 system/core/init/keychords.c 中定义，具体实现代码如下所示：

```
void handle_keychord()
{
    struct service *svc;
    const char* debuggable;
    const char* adb_enabled;
    int ret;
    __u16 id;

    // only handle keychords if ro.debuggable is set or adb is enabled.
    // the logic here is that bugreports should be enabled in userdebug or eng builds
    // and on user builds for users that are developers.
    debuggable = property_get("ro.debuggable");
    adb_enabled = property_get("init.svc.adbd");
    ret = read(keychord_fd, &id, sizeof(id));
    if (ret != sizeof(id)) {
        ERROR("could not read keychord id\n");
        return;
    }

    if ((debuggable && !strcmp(debuggable, "1")) ||
        (adb_enabled && !strcmp(adb_enabled, "running"))) {
        svc = service_find_by_keychord(id);
        if (svc) {
            INFO("starting service %s from keychord\n", svc->name);
            service_start(svc, NULL);
        } else {
            ERROR("service for keychord %d not found\n", id);
        }
    }
}
```

8.1.6　控制属性服务

编写过 Windows 本地应用的读者应该知道，在 Windows 中有一个注册表机制，在注册表中提供了大量的属性。在 Linux 中也有类似的机制，这就是属性服务。init 在启动的过程中会启动属性服务（Socket 服务），并且在内存中建立一块存储区域，用来存储这些属性。当读取这些属性时，直接从这一内存区域读取，如果修改属性值，需要通过 Socket 连接属性服务完成。在文件 init.c 中，函数 action 调用了 函数 start_property_service 来启动属性服务，action 是 init.rc 及其类似文件中的一种执行机制。

在文件 init.c 中，可以看到和属性操作相关的代码情景，例如：

```
property_init()
property_set_fd
```

在本节的内容中，将详细讲解在 Android 5.0 源代码中 init 控制属性服务的基本知识。

1. 引入属性

在文件 init.c 的主函数 main 中，调用函数 property_init 为属性分配了一些存储空间。如果查看文件 init.rc，会发现该文件开始部分用一些 import 语句导入了其他的配置文件，例如，/init.usb.rc。大多数配置文件都直接使用了确定的文件名，只有如下的代码使用了一个变量（${ro.hardware}）执行了配置文件名的一部分：

```
import /init.${ro.hardware}.rc
```

要想了解上述变量的获得过程，首先需要了解配置文件 init.${ro.hardware}.rc 的内容，这些通常与当前的硬件有关，其中函数 get_hardware_name 用于获取硬件的名称信息，具体代码如下所示：

```
void get_hardware_name(char *hardware, unsigned int *revision)
{
    char data[1024];
    int fd, n;
    char *x, *hw, *rev;

    /* 如果 hardware 已经有值，说明 hardware 通过内核命令行提供，直接返回 */
    if (hardware[0])
        return;
    //  打开/proc/cpuinfo 文件
    fd = open("/proc/cpuinfo", O_RDONLY);
    if (fd < 0) return;
    //  读取/proc/cpuinfo 文件的内容
    n = read(fd, data, 1023);
    close(fd);
    if (n < 0) return;

    data[n] = 0;
    //  从/proc/cpuinfo 文件中获取 Hardware 字段的值
    hw = strstr(data, "\nHardware");
    rev = strstr(data, "\nRevision");
    //  成功获取 Hardware 字段的值
    if (hw) {
        x = strstr(hw, ": ");
        if (x) {
            x += 2;
            n = 0;
            while (*x && *x != '\n') {
                if (!isspace(*x))
                    //  将 Hardware 字段的值都转换为小写，并更新 hardware 参数的值
                    //  hardware 也就是在 init.c 文件中定义的 hardware 数组
                    hardware[n++] = tolower(*x);
                x++;
                if (n == 31) break;
            }
            hardware[n] = 0;
        }
    }
```

```
    if (rev) {
        x = strstr(rev, ": ");
        if (x) {
            *revision = strtoul(x + 2, 0, 16);
        }
    }
}
```

从上述代码可知，该函数主要用于确定 hardware 和 revision 变量的值。获取 hardware 的来源是从 Linux 内核命令行或文件“/proc/cpuinfo”中的内容中得到，文件“/proc/cpuinfo”是虚拟文件（内存文件），执行 cat /proc/cpuinfo 命令会看到该文件中的内容，如图 8-1 所示。

▲图 8-1 显示文件内容

在图 8-1 中，白框中的内容就是 Hardware 字段的值。由于该设备是 Nexus 7，所以值为 grouper。如果程序运行就到此为止，那么与硬件有关的配置文件名是 init.grouper.rc。有 Nexus 7 的读者会看到在根目录下确实有一个 init.grouper.rc 文件。说明 Nexus 7 的原生 ROM 并没有在其他的地方设置配置文件名，所以，配置文件名就是从/proc/cpuinfo 文件的 Hardware 字段中取的值。

接下来看在函数 get_hardware_name 后面调用的函数 process_kernel_cmdline，具体实现代码如下所示：

```
static void process_kernel_cmdline(void)
{
    /* don't expose the raw commandline to nonpriv processes */
    chmod("/proc/cmdline", 0440);

    // 导入内核命令行参数
    import_kernel_cmdline(0, import_kernel_nv);
    if (qemu[0])
        import_kernel_cmdline(1, import_kernel_nv);

    // 用属性值设置内核变量
    export_kernel_boot_props();
}
```

在上述代码中，除了使用函数 import_kernel_cmdline 导入内核变量外，其主要功能是调用函数 export_kernel_boot_props 通过属性设置内核变量。例如，通过属性 ro.boot.hardware 设置 hardware 变量。也就是说，可以通过属性值 ro.boot.hardware 修改函数 get_hardware_name 中从文件“/proc/cpuinfo”中得到的 hardware 字段值。函数 export_kernel_boot_props 的具体实现代码如下所示：

```
static void export_kernel_boot_props(void)
{
    char tmp[PROP_VALUE_MAX];
    const char *pval;
    unsigned i;
    struct {
        const char *src_prop;
        const char *dest_prop;
        const char *def_val;
    } prop_map[] = {
        { "ro.boot.serialno", "ro.serialno", "", },
        { "ro.boot.mode", "ro.bootmode", "unknown", },
        { "ro.boot.baseband", "ro.baseband", "unknown", },
        { "ro.boot.bootloader", "ro.bootloader", "unknown", },
    };
    // 通过内核的属性设置应用层配置文件的属性
    for (i = 0; i < ARRAY_SIZE(prop_map); i++) {
        pval = property_get(prop_map[i].src_prop);
        property_set(prop_map[i].dest_prop, pval ?: prop_map[i].def_val);
    }
```

```
    //  根据 ro.boot.console 属性的值设置 console 变量
    pval = property_get("ro.boot.console");
    if (pval)
        strlcpy(console, pval, sizeof(console));

    /* save a copy for init's usage during boot */
    strlcpy(bootmode, property_get("ro.bootmode"), sizeof(bootmode));

    /* if this was given on kernel command line, override what we read
     * before (e.g. from /proc/cpuinfo), if anything */
    //  获取 ro.boot.hardware 属性的值
    pval = property_get("ro.boot.hardware");
    if (pval)
        // 这里通过 ro.boot.hardware 属性再次改变 hardware 变量的值
        strlcpy(hardware, pval, sizeof(hardware));
    //  利用 hardware 变量的值设置 ro.hardware 属性
    //  这个属性就是前面提到的设置初始化文件名的属性，实际上是通过 hardware 变量设置的
    property_set("ro.hardware", hardware);

    snprintf(tmp, PROP_VALUE_MAX, "%d", revision);
    property_set("ro.revision", tmp);

    /* TODO: these are obsolete. We should delete them */
    if (!strcmp(bootmode,"factory"))
        property_set("ro.factorytest", "1");
    else if (!strcmp(bootmode,"factory2"))
        property_set("ro.factorytest", "2");
    else
        property_set("ro.factorytest", "0");
}
```

由上述代码可以看出，该函数实际上就是来回设置一些属性值，并且利用某些属性值修改 console、hardware 等变量。其中 hardware 变量（就是一个长度为 32 的字符数组）在函数 get_hardware_name 中已经从文件“/proc/cpuinfo”中获得过一次值，在函数 export_kernel_boot_props 中又通过属性 ro.boot.hardware 设置了一次值，不过，在 Nexus 7 中并没有设置该属性，所以，hardware 的值仍为 grouper。最后用变量 hardware 设置属性 ro.hardware，最后的初始化文件名为 init.grouper.rc。

2. 初始化属性服务

在文件“system\core\init\property_service.c”中，使用函数 property_init 初始化属性存储区域，具体实现代码如下所示：

```
void property_init(void)
{
    init_property_area();
}
```

在上述代码中，函数 init_property_area 也是在文件 property_service.c 中实现的，该函数用于初始化属性内存区域，也就是__system_property_area__变量。函数 init_property_area 的具体实现代码如下所示：

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

    if(pa_info_array)
        return -1;

    if(init_workspace(&pa_workspace, PA_SIZE))
        return -1;

    fcntl(pa_workspace.fd, F_SETFD, FD_CLOEXEC);

    pa_info_array = (void*) (((char*) pa_workspace.data) + PA_INFO_START);

    pa = pa_workspace.data;
```

```
    memset(pa, 0, PA_SIZE);
    pa->magic = PROP_AREA_MAGIC;
    pa->version = PROP_AREA_VERSION;

        /* plug into the lib property services */
    __system_property_area__ = pa;
    property_area_inited = 1;
    return 0;
}
```

3. 启动属性服务

在文件“system\core\init\property_service.c”中，使用函数 start_property_service 启动一个属性服务器，具体实现代码如下所示：

```
void start_property_service(void)
{
    int fd;
    //  装载不同的属性文件
    load_properties_from_file(PROP_PATH_SYSTEM_BUILD);
    load_properties_from_file(PROP_PATH_SYSTEM_DEFAULT);
    load_override_properties();
    /* Read persistent properties after all default values have been loaded. */
    load_persistent_properties();
    //  创建 Socket 服务（属性服务）
    fd = create_socket(PROP_SERVICE_NAME, SOCK_STREAM, 0666, 0, 0);
    if(fd < 0) return;
    fcntl(fd, F_SETFD, FD_CLOEXEC);
    fcntl(fd, F_SETFL, O_NONBLOCK);
    //  开始服务监听
    listen(fd, 8);
    property_set_fd = fd;
}
```

通过上述代码可以知道属性服务的启动方式，另外，在函数 start_property_service 中还涉及了如下所示的两个宏：

- PROP_PATH_SYSTEM_BUILD;
- PROP_PATH_SYSTEM_DEFAULT。

上述两个宏都是系统预定义的属性文件名的路径，为了获取这些宏的定义，需要先分析函数 property_get，该函数也是在 Property_service.c 中实现，具体实现代码如下所示：

```
const char* property_get(const char *name)
{
    prop_info *pi;
    if(strlen(name) >= PROP_NAME_MAX) return 0;
    pi = (prop_info*) __system_property_find(name);
    if(pi != 0) {
        return pi->value;
    } else {
        return 0;
    }
}
```

通过上述代码可以看到，在函数 property_get 中调用了核心函数__system_property_find，该核心函数真正实现了获取属性值的功能。函数__system_property_find 属于 bionic 的一个 library，在文件 system_properties.c 中实现，可以在如下的目录找到该文件：

```
/bionic/libc/bionic
```

函数__system_property_find 的具体实现代码如下所示：

```
const prop_info *__system_property_find(const char *name)
{
    //  获取属性存储内存区域的首地址
    prop_area *pa = __system_property_area__;
    unsigned count = pa->count;
    unsigned *toc = pa->toc;
    unsigned len = strlen(name);
    prop_info *pi;
```

```
    while(count--) {
        unsigned entry = *toc++;
        if(TOC_NAME_LEN(entry) != len) continue;

            pi = TOC_TO_INFO(pa, entry);
        if(memcmp(name, pi->name, len)) continue;
            return pi;
    }
    return 0;
}
```

从上述函数__system_property_find 的实现代码可以看出，第一行使用了一个__system_property_area__变量，该变量是全局的。

在文件 system_properties.c 对应的头文件 system_properties.h 中，定义了前面提到的两个表示属性文件路径的宏，其实，还有另外两个表示路径的宏，共 4 个属性文件。文件 system_properties.h 可以在如下所示的目录中找到：

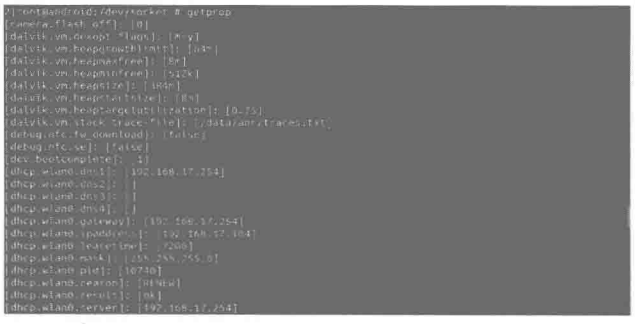
```
/bionic/libc/include/sys
```

这 4 个宏的具体定义如下所示：

```
#define PROP_PATH_RAMDISK_DEFAULT  "/default.prop"
#define PROP_PATH_SYSTEM_BUILD     "/system/build.prop"
#define PROP_PATH_SYSTEM_DEFAULT   "/system/default.prop"
#define PROP_PATH_LOCAL_OVERRIDE   "/data/local.prop"
```

此时可以进入 Android 设备的相应目录找到上述 4 个文件，一般会被保存在根目录中，通常在文件 default.prop 和 catdefault.prop 中会看到该文件的内容。而属性服务就是装载所有这 4 个属性文件中的所有属性，以及使用 property_set 设置的属性。在 Android 设备的终端可以直接使用 getprop 命令从属性服务获取所有的属性值，如图 8-2 所示。另外，getprop 命令还可以直接根据属性名获取具体的属性值，例如：

```
getprop ro.build.product
```



▲图 8-2　从属性服务获取所有的属性值

在 Android 5.0 源代码中，getprop 命令的源代码文件是 getprop.c。读者可以在如下所示的目录中找到该文件：

```
/system/core/toolbox/
```

其实 getprop 获取属性值也是通过函数 property_get 完成的，此函数实际上调用了函数__system_property_find，从__system_property_area__变量指定的内存区域获取相应的属性值。另外，在文件 system_properties.c 中还有如下两个函数用于通过属性服务修改或添加某个属性的值。

```
static int send_prop_msg(prop_msg *msg)
{
    struct pollfd pollfds[1];
    struct sockaddr_un addr;
    socklen_t alen;
    size_t namelen;
    int s;
```

```
    int r;
    int result = -1;
    //  创建用于连接属性服务的 Socket
    s = socket(AF_LOCAL, SOCK_STREAM, 0);
    if(s < 0) {
        return result;
    }

    memset(&addr, 0, sizeof(addr));
    // property_service_socket 是 Socket 设备文件名称
    namelen = strlen(property_service_socket);
    strlcpy(addr.sun_path, property_service_socket, sizeof addr.sun_path);
    addr.sun_family = AF_LOCAL;
    alen = namelen + offsetof(struct sockaddr_un, sun_path) + 1;

    if(TEMP_FAILURE_RETRY(connect(s, (struct sockaddr *) &addr, alen)) < 0) {
        close(s);
        return result;
    }

    r = TEMP_FAILURE_RETRY(send(s, msg, sizeof(prop_msg), 0));

    if(r == sizeof(prop_msg)) {
        pollfds[0].fd = s;
        pollfds[0].events = 0;
        r = TEMP_FAILURE_RETRY(poll(pollfds, 1, 250 /* ms */));
        if (r == 1 && (pollfds[0].revents & POLLHUP) != 0) {
            result = 0;
        } else {


            result = 0;
        }
    }

    close(s);
    return result;
}
// 用户可以直接调用该函数设置属性值
int __system_property_set(const char *key, const char *value)
{
    int err;
    int tries = 0;
    int update_seen = 0;
    prop_msg msg;

    if(key == 0) return -1;
    if(value == 0) value = "";
    if(strlen(key) >= PROP_NAME_MAX) return -1;
    if(strlen(value) >= PROP_VALUE_MAX) return -1;

    memset(&msg, 0, sizeof msg);
    msg.cmd = PROP_MSG_SETPROP;
    strlcpy(msg.name, key, sizeof msg.name);
    strlcpy(msg.value, value, sizeof msg.value);
    //  设置属性值
    err = send_prop_msg(&msg);
    if(err < 0) {
        return err;
    }
    return 0;
}
```

在函数 send_prop_msg 中，涉及了重要变量 property_service_socket，具体定义如下所示：

```
static const char property_service_socket[] = "/dev/socket/" PROP_SERVICE_NAME;
```

实际上，send_prop_msg 是通过这个设备文件与属性服务通信。读者可以在 Android 设备的终端进入“/dev/socket”目录，通常会看到一个名为 property_service 的文件，该文件就是属性服务映射的设备文件。

4. 处理设置属性的请求

当属性服务器收到客户端的请求时，init 会调用函数 handle_property_set_fd 进行处理。当客户端的权限满足要求时，init 就调用函数 property_set 进行相关的处理：

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

    int namelen = strlen(name);
    int valuelen = strlen(value);

    if(namelen >= PROP_NAME_MAX) return -1;
    if(valuelen >= PROP_VALUE_MAX) return -1;
    if(namelen < 1) return -1;

    pi = (prop_info*) __system_property_find(name);

    if(pi != 0) {
        /* ro.* properties may NEVER be modified once set */
        if(!strncmp(name, "ro.", 3)) return -1;

        pa = __system_property_area__;
        update_prop_info(pi, value, valuelen);
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    } else {
        pa = __system_property_area__;
        if(pa->count == PA_COUNT_MAX) return -1;

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

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

        pa->count++;
        pa->serial++;
        __futex_wake(&pa->serial, INT32_MAX);
    }
    /* If name starts with "net." treat as a DNS property. */
    if (strncmp("net.", name, strlen("net.")) == 0)  {
        if (strcmp("net.change", name) == 0) {
            return 0;
        }
        property_set("net.change", name);
    } else if (persistent_properties_loaded &&
            strncmp("persist.", name, strlen("persist.")) == 0) {
        write_persistent_property(name, value);
#ifdef HAVE_SELINUX
    } else if (strcmp("selinux.reload_policy", name) == 0 &&
               strcmp("1", value) == 0) {
        selinux_reload_policy();
#endif
    }
    property_changed(name, value);
    return 0;
}
```

到此为止，整个属性服务器的源代码知识就介绍完毕了。

8.2 分析 Zygote（孕育）进程

在 Android 5.0 系统中，在查看进程列表时会发现进程 Zygote 的父进程是 init，并且是所有应

用的父进程。另外也会发现，进程 system_server 的父进程是 Zygote。其实 Zygote 服务实际上是一种 Select 服务模型，是为启动 Java 代码而生，完成了一次 androidRuntime 的打开和关闭操作。

8.2.1 Zygote 基础

Android 系统是基于 Linux 内核的，在 Linux 系统中的所有进程都是 init 进程的子孙进程。也就是说，所有进程都是直接或者间接地由 init 进程 fork（孕育）出来的。Zygote 进程也不例外，它是在系统启动的过程，由 init 进程创建的。Zygote 是 Android 系统的核心进程之一，被认为是 Android ramework 大家族的祖先。事实上，Zygote 正是我们平常所说的 Java 运行环境（JVM）。从总体架构上看，Zygote 是一个简单的典型 C/S 结构。其他进程作为一个客服端向 Zygote 发出“孕育”请求，当 Zygote 接收到命令后就“孕育”出一个 Activity 进程。具体“孕育”过程如图 8-3 所示。

在 Android 系统中，Zygote 本身是一个应用层的程序，和驱动、内核模块等没有任何关系。Zygote 系统源代码的组成及调用结构如下所示。

（1）Zygote.java

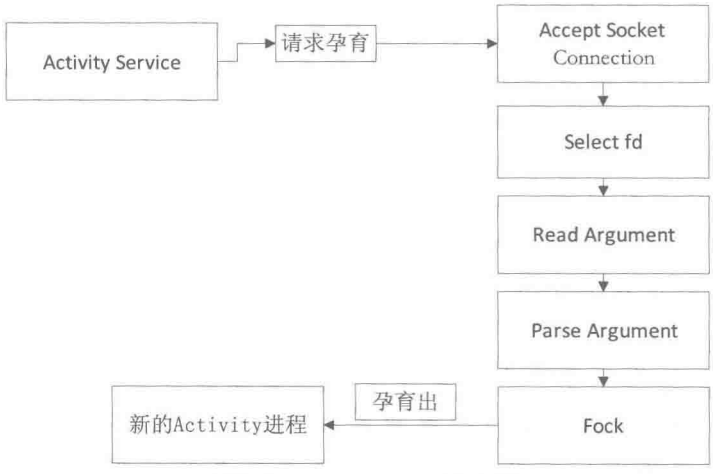
提供访问 Dalvik 的“zygote”接口，主要是包装 Linux 系统的 Fork（孕育），以建立一个新的 VM 实例进程。

（2）ZygoteConnection.java

Zygote 的套接口连接管理及其参数解析。其他 Actvitiy 建立进程请求是通过套接口发送命令参数给 Zygote。

（3）ZygoteInit.java

Zygote 系统的 main 函数入口。



▲图 8-3 Zygote 的“孕育”过程

8.2.2 分析 Zygote 的启动过程

在 Android 5.0 源代码中，在文件 system\core\rootdir\init.rc 中可以看到启动 Zygote 进程的脚本命令，具体代码如下所示：

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server
    socket zygote stream 666
```

通过上述代码，系统启动后会在“/dev/socket”目录下看到有一个名为“zygote”的文件。在上述代码中，相关键字的具体说明如下所示。

- service：通知 init 进程创建一个名为“zygote”的进程，此 zygote 进程要执行的程序是 /system/bin/app_process，后面部分需要传给 app_proces。

- socket：表示这个 zygote 进程需要一个名称为“zygote”的 socket 资源。

Zygote 最初的名字是 app_process，通过直接调用 pctrl 后把名字改成了“zygote”。Zygote 进程执行的程序是 system/bin/app_process，其对应的源代码在如下文件中定义：

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

文件 app_main.cpp 的入口函数是 main，接下来的内容中，将详细讲解启动 Zygote 进程的过程。

（1）分析启动脚本

在文件 system\core\init\init.c 中，以服务的形式来启动 Zygote 进程。在启动初始化进程 init 时，会调用函数 service_start 来启动 Zygote。函数 service_start 的具体实现代码如下所示：

```
void service_start(struct service *svc, const char *dynamic_args)
{
    struct stat s;
    pid_t pid;
    int needs_console;
    int n;
    char *scon = NULL;
    int rc;
    svc->flags &= (~(SVC_DISABLED|SVC_RESTARTING|SVC_RESET));
    svc->time_started = 0;
    if (svc->flags & SVC_RUNNING) {
        return;
    }

    needs_console = (svc->flags & SVC_CONSOLE) ? 1 : 0;
    if (needs_console && (!have_console)) {
        ERROR("service '%s' requires console\n", svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (stat(svc->args[0], &s) != 0) {
        ERROR("cannot find '%s', disabling '%s'\n", svc->args[0], svc->name);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if ((!(svc->flags & SVC_ONESHOT)) && dynamic_args) {
        ERROR("service '%s' must be one-shot to use dynamic args, disabling\n",
              svc->args[0]);
        svc->flags |= SVC_DISABLED;
        return;
    }

    if (is_selinux_enabled() > 0) {
        if (svc->seclabel) {
            scon = strdup(svc->seclabel);
            if (!scon) {
                ERROR("Out of memory while starting '%s'\n", svc->name);
                return;
            }
        } else {
            char *mycon = NULL, *fcon = NULL;

            INFO("computing context for service '%s'\n", svc->args[0]);
            rc = getcon(&mycon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                return;
            }

            rc = getfilecon(svc->args[0], &fcon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                freecon(mycon);
                return;
            }
```

```
            rc = security_compute_create(mycon, fcon, string_to_security_class
            ("process"), &scon);
            freecon(mycon);
            freecon(fcon);
            if (rc < 0) {
                ERROR("could not get context while starting '%s'\n", svc->name);
                return;
            }
        }
    }

    NOTICE("starting '%s'\n", svc->name);

    pid = fork();

    if (pid == 0) {
        struct socketinfo *si;
        struct svcenvinfo *ei;
        char tmp[32];
        int fd, sz;

        umask(077);
        if (properties_inited()) {
            get_property_workspace(&fd, &sz);
            sprintf(tmp, "%d,%d", dup(fd), sz);
            add_environment("ANDROID_PROPERTY_WORKSPACE", tmp);
        }

        for (ei = svc->envvars; ei; ei = ei->next)
            add_environment(ei->name, ei->value);

        setsockcreatecon(scon);

        for (si = svc->sockets; si; si = si->next) {
            int socket_type = (
                    !strcmp(si->type, "stream") ? SOCK_STREAM :
                        (!strcmp(si->type, "dgram") ? SOCK_DGRAM : SOCK_SEQPACKET));
            int s = create_socket(si->name, socket_type,
                                  si->perm, si->uid, si->gid);
            if (s >= 0) {
                publish_socket(si->name, s);
            }
        }

        freecon(scon);
        scon = NULL;
        setsockcreatecon(NULL);

        if (svc->ioprio_class != IoSchedClass_NONE) {
            if (android_set_ioprio(getpid(), svc->ioprio_class, svc->ioprio_pri)) {
                ERROR("Failed to set pid %d ioprio = %d,%d: %s\n",
                      getpid(), svc->ioprio_class, svc->ioprio_pri, strerror(errno));
            }
        }

        if (needs_console) {
            setsid();
            open_console();
        } else {
            zap_stdio();
        }

#if 0
        for (n = 0; svc->args[n]; n++) {
            INFO("args[%d] = '%s'\n", n, svc->args[n]);
        }
        for (n = 0; ENV[n]; n++) {
            INFO("env[%d] = '%s'\n", n, ENV[n]);
        }
#endif
```

```
        setpgid(0, getpid());

    /* as requested, set our gid, supplemental gids, and uid */
        if (svc->gid) {
            if (setgid(svc->gid) != 0) {
                ERROR("setgid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->nr_supp_gids) {
            if (setgroups(svc->nr_supp_gids, svc->supp_gids) != 0) {
                ERROR("setgroups failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->uid) {
            if (setuid(svc->uid) != 0) {
                ERROR("setuid failed: %s\n", strerror(errno));
                _exit(127);
            }
        }
        if (svc->seclabel) {
            if (is_selinux_enabled() > 0 && setexeccon(svc->seclabel) < 0) {
                ERROR("cannot setexeccon('%s'): %s\n", svc->seclabel, strerror(errno));
                _exit(127);
            }
        }

        if (!dynamic_args) {
            if (execve(svc->args[0], (char**) svc->args, (char**) ENV) < 0) {
                ERROR("cannot execve('%s'): %s\n", svc->args[0], strerror(errno));
            }
        } else {
            char *arg_ptrs[INIT_PARSER_MAXARGS+1];
            int arg_idx = svc->nargs;
            char *tmp = strdup(dynamic_args);
            char *next = tmp;
            char *bword;

            /* Copy the static arguments */
            memcpy(arg_ptrs, svc->args, (svc->nargs * sizeof(char *)));

            while((bword = strsep(&next, " "))) {
                arg_ptrs[arg_idx++] = bword;
                if (arg_idx == INIT_PARSER_MAXARGS)
                    break;
            }
            arg_ptrs[arg_idx] = '\0';
            execve(svc->args[0], (char**) arg_ptrs, (char**) ENV);
        }
        _exit(127);
    }

    freecon(scon);

    if (pid < 0) {
        ERROR("failed to start '%s'\n", svc->name);
        svc->pid = 0;
        return;
    }

    svc->time_started = gettime();
    svc->pid = pid;
    svc->flags |= SVC_RUNNING;

    if (properties_inited())
        notify_service_state(svc->name, "running");
}
```

在上述代码中，每一个 Service 命令都会促使 init 进程调用 fork 函数来创建一个新的进程，在

新的进程中会分析里面的 socket 选项。对于每一个 socket 选项来说，都会通过函数 create_socket 来在/dev/socket 目录下创建一个文件。由此可见，函数 service_start 起了一个解释文件 init.rc 中 Service 命令的作用。

再看函数 create_socket，功能是调用函数 socket 创建一个 Socket，使用文件描述符 fd 来描述此 Socket。函数 create_socket 的具体实现代码如下所示：

```
int create_socket(const char *name, int type, mode_t perm, uid_t uid, gid_t gid)
{
    struct sockaddr_un addr;
    int fd, ret;
    char *secon;
//调用函数 socket 创建一个 Socket,使用文件描述符 fd 来描述此 Socket
    fd = socket(PF_UNIX, type, 0);
    if (fd < 0) {
        ERROR("Failed to open socket '%s': %s\n", name, strerror(errno));
        return -1;
    }
//为 Socket 创建一个类型为 AF_UNIX 的 Socket 地址 addr
    memset(&addr, 0 , sizeof(addr));
    addr.sun_family = AF_UNIX;
    snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s",
             name);
    ret = unlink(addr.sun_path);
    if (ret != 0 && errno != ENOENT) {
        ERROR("Failed to unlink old socket '%s': %s\n", name, strerror(errno));
        goto out_close;
    }
    secon = NULL;
    if (sehandle) {
        ret = selabel_lookup(sehandle, &secon, addr.sun_path, S_IFSOCK);
        if (ret == 0)
            setfscreatecon(secon);
    }
    ret = bind(fd, (struct sockaddr *) &addr, sizeof (addr));
    if (ret) {
        ERROR("Failed to bind socket '%s': %s\n", name, strerror(errno));
        goto out_unlink;
    }
    setfscreatecon(NULL);
    freecon(secon);
    //设置设备文件的/dev/socket/zygote 的用户 id、用户组 id 和用户权限
    chown(addr.sun_path, uid, gid);
    chmod(addr.sun_path, perm);
    INFO("Created socket '%s' with mode '%o', user '%d', group '%d'\n",
         addr.sun_path, perm, uid, gid);
    return fd;
out_unlink:
    unlink(addr.sun_path);
out_close:
    close(fd);
    return -1;
}
```

再看函数 publish_socket，具体实现代码如下所示：

```
//参数 fd 是文件描述符，指向函数 create_socket 创建的 socket
static void publish_socket(const char *name, int fd)
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];
//将宏 ANDROID_SOCKET_ENV_PREFIX 和参数 name 描述的字符串连接在一起，并保存在字符串 key 中
    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
            sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

（2）分析入口函数

Zygote 的入口函数是 main，功能是创建 AppRuntime 变量，然后调用成员函数 start 启动进程。函数是 main 在文件 frameworks\base\cmds\app_process\app_main.cpp 中定义的，具体实现代码如下所示：

```
int main(int argc, char* const argv[])
{
#ifdef __arm__
    char value[PROPERTY_VALUE_MAX];
    property_get("ro.kernel.qemu", value, "");
    bool is_qemu = (strcmp(value, "1") == 0);
    if ((getenv("NO_ADDR_COMPAT_LAYOUT_FIXUP") == NULL) && !is_qemu) {
        int current = personality(0xFFFFFFFF);
        if ((current & ADDR_COMPAT_LAYOUT) == 0) {
            personality(current | ADDR_COMPAT_LAYOUT);
            setenv("NO_ADDR_COMPAT_LAYOUT_FIXUP", "1", 1);
            execv("/system/bin/app_process", argv);
            return -1;
        }
    }
    unsetenv("NO_ADDR_COMPAT_LAYOUT_FIXUP");
#endif

    // These are global variables in ProcessState.cpp
    mArgC = argc;
    mArgV = argv;

    mArgLen = 0;
    for (int i=0; i<argc; i++) {
        mArgLen += strlen(argv[i]) + 1;
    }
    mArgLen--;

    AppRuntime runtime;
    const char* argv0 = argv[0];
    argc--;
    argv++;

    int i = runtime.addVmArguments(argc, argv);
    bool zygote = false;
    bool startSystemServer = false;
    bool application = false;
    const char* parentDir = NULL;
    const char* niceName = NULL;
    const char* className = NULL;
    while (i < argc) {
        const char* arg = argv[i++];
        if (!parentDir) {
            parentDir = arg;
        } else if (strcmp(arg, "--zygote") == 0) {
            zygote = true;
            niceName = "zygote";
        } else if (strcmp(arg, "--start-system-server") == 0) {
            startSystemServer = true;
        } else if (strcmp(arg, "--application") == 0) {
            application = true;
        } else if (strncmp(arg, "--nice-name=", 12) == 0) {
            niceName = arg + 12;
        } else {
            className = arg;
            break;
        }
    }

    if (niceName && *niceName) {
        setArgv0(argv0, niceName);
        set_process_name(niceName);
    }
```

```
    runtime.mParentDir = parentDir;

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit",
                startSystemServer ? "start-system-server" : "");
    } else if (className) {
        // Remainder of args get passed to startup class main()
        runtime.mClassName = className;
        runtime.mArgC = argc - i;
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
                application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

（3）分析启动函数

Zygote 的启动函数是 start，功能是调用函数 startVm 在 Zygote 中创建一个虚拟机实例。函数是 start 在文件 frameworks\base\core\jni\AndroidRuntime.cpp 中定义的，具体实现代码如下所示：

```
void AndroidRuntime::start(const char* className, const char* options)
{
    ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
            className != NULL ? className : "(unknown)");

    blockSigpipe();
    if (strcmp(options, "start-system-server") == 0) {
        /* track our progress through the boot sequence */
        const int LOG_BOOT_PROGRESS_START = 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char* rootDir = getenv("ANDROID_ROOT");
    if (rootDir == NULL) {
        rootDir = "/system";
        if (!hasDir("/system")) {
            LOG_FATAL("No root directory specified, and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /*创建一个虚拟机实例 */
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * 调用函数 startReg 在虚拟机实例中注册 JNI 方法
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

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

```
    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);

    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits.
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
//调用类 com.android.internal.os.ZygoteInit 的静态成员函数 main 来启动 Zygote 进程
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
    free(slashClassName);
    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

在上述代码中，通过调用类 com.android.internal.os.ZygoteInit 中的函数 main 启动了 Zygote 进程。此成员函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
    public static void main(String argv[]) {
        try {
            // Start profiling the zygote initialization.
            SamplingProfilerIntegration.start();
            //调用函数 registerZygoteSocket 创建一个 socket 接口
            registerZygoteSocket();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
                SystemClock.uptimeMillis());
            preload();
            EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
                SystemClock.uptimeMillis());

            // Finish profiling the zygote initialization.
            SamplingProfilerIntegration.writeZygoteSnapshot();

            // Do an initial gc to clean up after startup
            gc();

            // Disable tracing so that forked processes do not inherit stale tracing tags from
            // Zygote.
            Trace.setTracingEnabled(false);

            // If requested, start system server directly from Zygote
```

```
            if (argv.length != 2) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }
            //调用函数 startSystemServer 启动 SystemServer 组件
            if (argv[1].equals("start-system-server")) {
                startSystemServer();
            } else if (!argv[1].equals("")) {
                throw new RuntimeException(argv[0] + USAGE_STRING);
            }

            Log.i(TAG, "Accepting command socket connections");
//调用函数 runSelectLoop 进入一个无限循环
//在前面创建的 socket 接口中等待 ActivityManagerService 请求，以创建新的应用程序进程
            runSelectLoop();

            closeServerSocket();
        } catch (MethodAndArgsCaller caller) {
            caller.run();
        } catch (RuntimeException ex) {
            Log.e(TAG, "Zygote died with exception", ex);
            closeServerSocket();
            throw ex;
        }
    }
```

（4）和 Zygote 进程中的 Socket 实现连接

在 Android 系统中，ActivityManagerService 通过函数 Process.start 创建一个新的进程。函数 Process.start 会先通过 Socket 连接到 Zygote 进程，并由 Zygote 进程实现创建新应用程序进程的功能。另外，而类 Process 是通过函数 openZygoteSocketIfNeeded 来连接到 Zygote 进程中的 Socket。函数 openZygoteSocketIfNeeded 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
private static void openZygoteSocketIfNeeded()
        throws ZygoteStartFailedEx {
    int retryCount;
    if (sPreviousZygoteOpenFailed) {
        /*
         * If we've failed before, expect that we'll fail again and
         * don't pause for retries.
         */
        retryCount = 0;
    } else {
        retryCount = 10;
    }
    for (int retry = 0
            ; (sZygoteSocket == null) && (retry < (retryCount + 1))
            ; retry++ ) {
        if (retry > 0) {
            try {
                Log.i("Zygote", "Zygote not up yet, sleeping...");
                Thread.sleep(ZYGOTE_RETRY_MILLIS);
            } catch (InterruptedException ex) {
                // should never happen
            }
        }
        try {
            sZygoteSocket = new LocalSocket();
            sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
                    LocalSocketAddress.Namespace.RESERVED));
            sZygoteInputStream
                    = new DataInputStream(sZygoteSocket.getInputStream());
            sZygoteWriter =
                new BufferedWriter(
                        new OutputStreamWriter(
                                sZygoteSocket.getOutputStream()),
                        256);
            Log.i("Zygote", "Process: zygote socket opened");
            sPreviousZygoteOpenFailed = false;
            break;
```

```
        } catch (IOException ex) {
            if (sZygoteSocket != null) {
                try {
                    sZygoteSocket.close();
                } catch (IOException ex2) {
                    Log.e(LOG_TAG,"I/O exception on close after exception",
                            ex2);
                }
            }
            sZygoteSocket = null;
        }
    }
    if (sZygoteSocket == null) {
        sPreviousZygoteOpenFailed = true;
        throw new ZygoteStartFailedEx("connect failed");
    }
}
```

在文件 ZygoteInit.java 中，函数 main 用到了函数 registerZygoteSocket，此函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
        try {
            String env = System.getenv(ANDROID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
        } catch (RuntimeException ex) {
            throw new RuntimeException(
                    ANDROID_SOCKET_ENV + " unset or invalid", ex);
        }

        try {
            sServerSocket = new LocalServerSocket(
                    createFileDescriptor(fileDesc));
        } catch (IOException ex) {
            throw new RuntimeException(
                    "Error binding to local socket '" + fileDesc + "'", ex);
        }
    }
}
```

在上述代码中，文件描述符创建了 socket 接口，此文件描符就是本书前面所讲的文件“/dev/socket/zygote”，而此文件描述符通过环境变量 ANDROID_SOCKET_ENV 获得。另外，由 init 进程负责解释执行系统启动脚本文件 system\core\rootdir\init.rc，而 init 进程的源代码位于文件 system/core/init/init.c 中，由函数 service_start 负责解释文件 init.rc 中的 service 命令。在 service_start 函数中，每一个 service 命令都会促使进程 init 调用函数 fork 创建一个新的进程，在新进程中会解析里面的 socket 选项。对于每一个 socket 选项来说，全部会通过 t 函数 create_socke 在“/dev/socket”目录下创建一个 zygote 文件，然后通过函数 publish_socket 将得到的文件描述符写入到环境变量中。函数 publish_socket 的具体实现代码如下所示：

```
static void publish_socket(const char *name, int fd)
{
    char key[64] = ANDROID_SOCKET_ENV_PREFIX;
    char val[64];

    strlcpy(key + sizeof(ANDROID_SOCKET_ENV_PREFIX) - 1,
            name,
            sizeof(key) - sizeof(ANDROID_SOCKET_ENV_PREFIX));
    snprintf(val, sizeof(val), "%d", fd);
    add_environment(key, val);

    /* make sure we don't close-on-exec */
    fcntl(fd, F_SETFD, 0);
}
```

在上述代码中，传进的参数 name 值为“zygote”，而 ANDROID_SOCKET_ENV_PREFIX 在

文件 system\core\include\cutils\sockets.h 中的定义代码为：

```
#define ANDROID_SOCKET_ENV_PREFIX "ANDROID_SOCKET_"
```

这样就将得到的文件描述符写入“ANDROID_SOCKET_zygote”的环境变量，这个环境变量值为 key 值。因为函数 ZygoteInit.registerZygoteSocket 和函数 create_socket 都是运行在同一个进程中，所以函数 ZygoteInit.registerZygoteSocket 可以直接使用文件描述符来创建一个 Java 层的 LocalServerSocket 对象。如果其他进程也需要打开“/dev/socket/zygote”文件用以和 Zygote 进程进行通信，则必须通过文件名作为中介来连接 LocalServerSocket。

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 startSystemServer，此函数也是在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示。

```
    private static boolean startSystemServer()
            throws MethodAndArgsCaller, RuntimeException {
        /* Hardcoded command line to start the system server */
        String args[] = {
            "--setuid=1000",
            "--setgid=1000",
"--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,3003,3
006,3007",
            "--capabilities=130104352,130104352",
            "--runtime-init",
            "--nice-name=system_server",
            "com.android.server.SystemServer",
        };
        ZygoteConnection.Arguments parsedArgs = null;

        int pid;

        try {
            parsedArgs = new ZygoteConnection.Arguments(args);
            ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
            ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);

            /* Request to fork the system server process */
            pid = Zygote.forkSystemServer(
                    parsedArgs.uid, parsedArgs.gid,
                    parsedArgs.gids,
                    parsedArgs.debugFlags,
                    null,
                    parsedArgs.permittedCapabilities,
                    parsedArgs.effectiveCapabilities);
        } catch (IllegalArgumentException ex) {
            throw new RuntimeException(ex);
        }
        /* For child process */
        if (pid == 0) {
            handleSystemServerProcess(parsedArgs);
        }
        return true;
    }
```

在文件 ZygoteInit.java 中，函数 main 的实现代码里，用到了函数 startSystemServer，此函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static boolean startSystemServer()
        throws MethodAndArgsCaller, RuntimeException {
    /* Hardcoded command line to start the system server */
    String args[] = {
        "--setuid=1000",
        "--setgid=1000",
        "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009,1010,1018,3001,3002,
3003,3006,3007", "--capabilities=130104352,130104352",
        "--runtime-init",
        "--nice-name=system_server",
```

```
        "com.android.server.SystemServer",
    };
    ZygoteConnection.Arguments parsedArgs = null;

    int pid;

    try {
        parsedArgs = new ZygoteConnection.Arguments(args);
        ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
        ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
        /* Request to fork the system server process */
        pid = Zygote.forkSystemServer(
                parsedArgs.uid, parsedArgs.gid,
                parsedArgs.gids,
                parsedArgs.debugFlags,
                null,
                parsedArgs.permittedCapabilities,
                parsedArgs.effectiveCapabilities);
    } catch (IllegalArgumentException ex) {
        throw new RuntimeException(ex);
    }
    /* For child process */
    if (pid == 0) {
        handleSystemServerProcess(parsedArgs);
    }
    return true;
}
```

在上述代码中，Zygote 进程通过函数 forkSystemServer“孕育”了一个新的进程来启动 SystemServer 组件，返回值 pid 为 0 的位置标志新进程的执行路径，即新创建进程会执行函数 handleSystemServerProcess。函数 handleSystemServerProcess 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void handleSystemServerProcess(
        ZygoteConnection.Arguments parsedArgs)
        throws ZygoteInit.MethodAndArgsCaller {
    closeServerSocket();
    // set umask to 0077 so new files and directories will default to owner-only permissions.
    Libcore.os.umask(S_IRWXG | S_IRWXO);
    if (parsedArgs.niceName != null) {
        Process.setArgV0(parsedArgs.niceName);
    }
    if (parsedArgs.invokeWith != null) {
        WrapperInit.execApplication(parsedArgs.invokeWith,
                parsedArgs.niceName, parsedArgs.targetSdkVersion,
                null, parsedArgs.remainingArgs);
    } else {
        /*
         * Pass the remaining arguments to SystemServer.
         */
        RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, parsedArgs.remainingArgs);
    }
    /* should never reach here */
}
```

在上述代码中，调用函数 closeServerSocket 关闭了子进程，然后调用函数 RuntimeInit.zygoteInit 进一步启动 SystemServer 组件。函数 RuntimeInit.zygoteInit 在文件 frameworks/base/core/java/com/android/internal/os/RuntimeInit.java 中定义，具体实现代码如下所示：

```
    public static final void zygoteInit(int targetSdkVersion, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

        redirectLogStreams();
        commonInit();
//调用函数 zygoteInitNative 来执行一个 Binder 进程间通信机制的初始化工作
//当完成这个工作后，这个进程中的 Binder 对象就可以方便地进行进程间通信
        nativeZygoteInit();
```

```
        applicationInit(targetSdkVersion, argv);
    }
```

在文件 ZygoteInit.java 中的函数 main 的实现代码中，用到了函数 runSelectLoop，功能是进入一个无限循环在前面创建的 socket 接口中，并等待 ActivityManagerService 请求创建新的应用程序进程。函数 runSelectLoop 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
    private static void runSelectLoop() throws MethodAndArgsCaller {
        ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
        ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
        FileDescriptor[] fdArray = new FileDescriptor[4];

        fds.add(sServerSocket.getFileDescriptor());
        peers.add(null);

        int loopCount = GC_LOOP_COUNT;
        while (true) {
            int index;
            if (loopCount <= 0) {
                gc();
                loopCount = GC_LOOP_COUNT;
            } else {
                loopCount--;
            }


            try {
                fdArray = fds.toArray(fdArray);
                index = selectReadable(fdArray);
            } catch (IOException ex) {
                throw new RuntimeException("Error in select()", ex);
            }

            if (index < 0) {
                throw new RuntimeException("Error in select()");
            } else if (index == 0) {
                ZygoteConnection newPeer = acceptCommandPeer();
                peers.add(newPeer);
                fds.add(newPeer.getFileDesciptor());
            } else {
                boolean done;
//将数据通过 Socket 接口发送出去后会执行下面的语句
// peers.get(index)得到的是一个 ZygoteConnection 对象，表示一个 Socket 连接
//调用 ZygoteConnection.runOnce 函数进一步处理
    done = peers.get(index).runOnce();

                if (done) {
                    peers.remove(index);
                    fds.remove(index);
                }
            }
        }
    }
```

通过上述代码，可以等待 ActivityManagerService 连接这个 Socket，然后调用函数 ZygoteConnection.runOnce 创建新的应用程序。

第 9 章　System 进程和应用程序进程

在 Android 系统中，存在如下两个十分重要的进程系统。

- System 进程：是系统进程，是整个 Android Framework 所在的进程，用于启动 Android 系统。其核心进程是 system_server，它的父进程是 zygote。
- 应用程序进程：每个 Android 应用程序运行后都会拥有自己的进程，这和 Windows 资源管理器中体现的进程是同一个含义。

在本章的内容中，将详细分析 Android 5.0 系统中上述两大进程系统的基本知识，彻底深入了解 Android 进程系统的方方面面。

9.1 分析 System 进程

在 Android 系统中，System 进程和系统服务有着重要的关系。几乎所有的 Android 核心服务都在这个进程中，如 ActivityManagerService、PowerManagerService 和 WindowManagerService 等。在本节的内容中，将详细分析 Android 5.0 中的 System 系统源代码，为读者步入本书后面知识的学习打下基础。

9.1.1　启动 System 进程前的准备工作

在 Android 系统中，通过静态类 ZygoteInit 的成员函数 handleSystemServerProcess 来启动 System 进程。具体启动过程如图 9-1 所示。

▲图 9-1　启动 System 进程前的准备工作

（1）首先在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中，获取 Zygote 进程在启动过程中创建的 Socket。其实 System 进程不需要这个 Socket，所以会调用类 ZygoteInit 的成员函数 closeServerSocket 关闭这个 Socket。对应的代码如下所示：

```
    private static void handleSystemServerProcess(
            ZygoteConnection.Arguments parsedArgs)
            throws ZygoteInit.MethodAndArgsCaller {
            //关闭这个 Socket
        closeServerSocket();
        // set umask to 0077 so new files and directories will default to owner-only permissions.
        Libcore.os.umask(S_IRWXG | S_IRWXO);
        if (parsedArgs.niceName != null) {
            Process.setArgV0(parsedArgs.niceName);
        }
        if (parsedArgs.invokeWith != null) {
            WrapperInit.execApplication(parsedArgs.invokeWith,
                    parsedArgs.niceName, parsedArgs.targetSdkVersion,
                    null, parsedArgs.remainingArgs);
        } else {
//调用类 RuntimeInit 的静态函数 zygoteInit 启动 System 进程
```

```
            RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion,
parsedArgs.remainingArgs);
        }
    }
```

（2）接下来调用类 RuntimeInit 的静态函数 zygoteInit 启动 System 进程，此函数在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
    public static final void zygoteInit(int targetSdkVersion, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application from zygote");

        redirectLogStreams();
//调用函数 commonInit 设置 Sysrem 进程的时区和键盘布局等信息
        commonInit ();
//调用函数 nativeZygoteInit 启动一个 Binder 线程池
        nativeZygoteInit();

        applicationInit(targetSdkVersion, argv);
    }
```

9.1.2 分析 SystemServer

SystemServer 是 Android Java 的两大支柱进程之一，另一个是专门负责孵化 Java 进程的 Zygote。如果这两大支柱崩溃了其中的任何一个，都会导致 Android 中 Java 层的崩溃。如果 Java 层真的崩溃了，则 Linux 系统中的进程 init 会重新启动 SystemServer 和 Zygote，以重新建立 Android 的 Java 层。在本节的内容中，将首先分析 SystemServer 的源代码。

（1）分析主函数 main

SystemServer 是由 Zygote 孵化而来的一个进程，通过 ps 命令，可知其进程名为 system_server。在 DDMS 中可以看到，进程 system_server 的进程名为 system_process。SystemServer 核心逻辑的入口是函数 main，此入口函数在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\SystemServer.java
```

文件 SystemServer.java 的入口函数是 main，具体实现代码如下所示：

```
    public static void main(String[] args) {
        if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
            //如果系统时钟早于 1970 年，则设置系统时钟从 1970 年开始
            Slog.w(TAG, "System clock is before 1970; setting to 1970.");
            SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME);
        }

        if (SamplingProfilerIntegration.isEnabled()) {
            SamplingProfilerIntegration.start();
            timer = new Timer();
            timer.schedule(new TimerTask() {
                @Override
                public void run() {
//SystemServer 性能统计，每小时统计一次，统计结果输出为文件
                    SamplingProfilerIntegration.writeSnapshot("system_server", null);
                } // SNAPSHOT_INTERVAL 定义为 1 小时
            }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
        }

        //和 Dalvik 虚拟机相关的设置，主要是内存使用方面的控制
        dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();
        VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);
        //加载动态库 libandroid_servers.so
        System.loadLibrary("android_servers");
        init1(args); //调用 native 的 init1 函数
    }

    public static final void init2() {
        Slog.i(TAG, "Entered the Android system server!");
        Thread thr = new ServerThread();
        thr.setName("android.server.ServerThread");
```

```
        thr.start();
    }
}
```

由此可见，函数 main 首先做一些初始化工作，然后加载动态库 libandroid_servers.so，最后调用 native 的函数 init1。该函数在 libandroid_servers.so 库中实现，在如下所示的文件中定义：

```
frameworks\base\services\jni\com_android_server_SystemServer.cpp
```

函数 init1 的具体实现代码如下所示：

```
extern "C" int system_init();
static void android_server_SystemServer_init1(JNIEnv* env, jobject clazz)
{
system_init();          //调用上面那个用 extern 声明的 system_init 函数
}
```

而函数 system_init 在另外一个库 libsystem_server.so 中实现，在如下所示的文件中定义：

```
frameworks\base\cmds\system_server\library\System_init.cpp
```

函数 system_init 的具体实现代码如下所示：

```
extern "C" status_t system_init()
{
    ALOGI("Entered system_init()");

    sp<ProcessState> proc(ProcessState::self());

    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p\n", sm.get());

    sp<GrimReaper> grim = new GrimReaper();
    sm->asBinder()->linkToDeath(grim, grim.get(), 0);

    char propBuf[PROPERTY_VALUE_MAX];
    property_get("system_init.startsurfaceflinger", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SurfaceFlinger::instantiate();
    }

    property_get("system_init.startsensorservice", propBuf, "1");
    if (strcmp(propBuf, "1") == 0) {
        SensorService::instantiate();
    }
    ALOGI("System server: starting Android runtime.\n");
    AndroidRuntime* runtime = AndroidRuntime::getRuntime();

    ALOGI("System server: starting Android services.\n");
    JNIEnv* env = runtime->getJNIEnv();
    if (env == NULL) {
        return UNKNOWN_ERROR;
    }
    jclass clazz = env->FindClass("com/android/server/SystemServer");
    if (clazz == NULL) {
        return UNKNOWN_ERROR;
    }
    jmethodID methodId = env->GetStaticMethodID(clazz, "init2", "()V");
    if (methodId == NULL) {
        return UNKNOWN_ERROR;
    }
    env->CallStaticVoidMethod(clazz, methodId);

    ALOGI("System server: entering thread pool.\n");
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
    ALOGI("System server: exiting thread pool.\n");

    return NO_ERROR;
}
```

通过上述代码可知，SystemServer 中的函数 main 通过函数 init1，从 Java 层穿越到 Native 层，

实现了一些初始化工作后，又通过 JNI 从 Native 层“穿越”到 Java 层去调用函数 init2。函数 init2 返回后，最终又回归到 Native 层。

（2）分析函数 init2

在文件 SystemServer.java 中，函数 init1 较简单，其重点内容都在函数 init2 中。函数 init2 的具体实现代码如下所示：

```
public static final void init2() {
    Thread thr = new ServerThread();
    thr.setName("android.server.ServerThread");
    thr.start();//启动一个线程，这个线程就像英雄大会一样，聚集了各路英雄
}
```

通过上述代码将创建一个新的线程 ServerThread，该线程的 run 函数的实现代码有 600 多行，如此之长的原因是，Android 平台中众多 Service 都汇集于此。

在 Android 5.0 系统中，共有 7 大类 43 个 Service（包括 Watchdog）。实际上，还有一些 Service 并没有在 ServerThread 的 run 函数中露面。这 7 大类服务主要包括。

- 第一大类：是 Android 的核心服务，如 ActivityManagerService、WindowManager-Service 等。
- 第二大类：是和通信相关的服务，如 Wifi 相关服务、Telephone 相关服务。
- 第三大类：是和系统功能相关的服务，如 AudioService、MountService、Usb-Service 等。
- 第四大类：是 BatteryService、VibratorService 等服务。
- 第五大类：是 EntropyService、DiskStatsService、Watchdog 等相对独立的服务。
- 第六大类：是蓝牙服务。
- 第七大类：是和 UI 紧密相关的服务，如状态栏服务、通知管理服务等。

在本章后面的内容中，将详细分析其中的第五类服务。该类中的 Service 之间关系简单，而且功能相对独立。第五大类服务包括如下所示的服务。

- EntropyService：熵服务，它和随机数的生成有关。
- ClipboardService：剪贴板服务。
- DropBoxManagerService：该服务和系统运行时日志的存储与管理有关。
- DiskStatsService 和 DeviceStorageMonitorService：这两个服务用于查看和监测系统存储空间。
- SamplingProfilerService：这个服务是从 Android 4.0 新增的，功能非常简单。
- Watchdog：即看门狗，是 Android 的“老员工”了。Android 2.3 以后其内存检测功能被去掉，所以与 Android 2.2 相比，显得更简单了。

9.1.3 分析 EntropyService

EntropyService 是 SystemServer 启动的第一个 Service，它以 3 个小时为单位，周期性地加载和保存熵池（/dev/urandom）。但是，由于/dev/urandom 本身就有的安全性相对差些，所以每隔 3 小时，Android 系统在 kernel 的熵池中增加了一些附加信息，这些信息对提高随机数的质量上是有帮助的。Android 会添加如下所示的额外的信息：

- out.println("Copyright (C) 2009 The Android Open Source Project");
- out.println("All Your Randomness Are Belong To Us");
- out.println(START_TIME);
- out.println(START_NANOTIME);
- out.println(SystemProperties.get("ro.serialno"));
- out.println(SystemProperties.get("ro.bootmode"));
- out.println(SystemProperties.get("ro.baseband"));
- out.println(SystemProperties.get("ro.carrier"));
- out.println(SystemProperties.get("ro.bootloader"));
- out.println(SystemProperties.get("ro.hardware"));
- out.println(SystemProperties.get("ro.revision"));

- out.println(System.currentTimeMillis());
- out.println(System.nanoTime())。

根据物理学基本原理，一个系统的熵越大，该系统就越不稳定。在 Android 中，目前也只有随机数常处于这种不稳定的系统中。在 Android 系统中，SystemServer 中添加该服务的代码如下所示：

```
ServiceManager.addService("entropy", new EntropyService());
```

上述代码非常简单，从中可直接分析 EntropyService 的构造函数，此函数在文件 EntropyService.java 中定义，具体实现代码如下所示：

```
public EntropyService() {
      //调用另外一个构造函数，getSystemDir 函数返回的是/data/system 目录
      this(getSystemDir() + "/entropy.dat", "/dev/urandom");
}
public EntropyService(String entropyFile, String randomDevice) {
 this.randomDevice = randomDevice;//urandom 是 Linux 系统中产生随机数的设备
 // /data/system/entropy.dat 文件保存了系统此前的熵信息
 this.entropyFile = entropyFile;
//下面有 4 个关键函数
 loadInitialEntropy();
 addDeviceSpecificEntropy();
 writeEntropy();
 scheduleEntropyWriter();
}
```

从以上代码中可以看出，EntropyService 构造函数中依次调用了 4 个关键函数，这 4 个函数比较简单，具体功能如下所示。

（1）函数 loadInitialEntropy

函数 loadInitialEntropy 的功能是，将文件 entropy.dat 中的内容写到 urandom 设备，这样可增加系统的随机性。在系统中有一个 entropy pool，在刚启动系统时，该 pool 中的内容为空，会导致早期生成的随机数变得可预测。通过将 entropy.dat 数据写到该 entropy pool（这样该 pool 中的内容就不为空）中，随机数的生成就无规律可言了。函数 loadInitialEntropy 的具体实现代码如下所示：

```
private void loadInitialEntropy() {
    try {
        RandomBlock.fromFile(entropyFile).toFile(randomDevice);
    } catch (IOException e) {
        Slog.w(TAG, "unable to load initial entropy (first boot?)", e);
    }
}
```

（2）函数 addDeviceSpecificEntropy

函数 addDeviceSpecificEntropy 的功能是，将一些和设备相关的信息写入 urandom 设备，具体实现代码如下所示：

```
private void addDeviceSpecificEntropy() {
    PrintWriter out = null;
    try {
        out = new PrintWriter(new FileOutputStream(randomDevice));
        out.println("Copyright (C) 2009 The Android Open Source Project");
        out.println("All Your Randomness Are Belong To Us");
        out.println(START_TIME);
        out.println(START_NANOTIME);
        out.println(SystemProperties.get("ro.serialno"));
        out.println(SystemProperties.get("ro.bootmode"));
        out.println(SystemProperties.get("ro.baseband"));
        out.println(SystemProperties.get("ro.carrier"));
        out.println(SystemProperties.get("ro.bootloader"));
        out.println(SystemProperties.get("ro.hardware"));
        out.println(SystemProperties.get("ro.revision"));
        out.println(System.currentTimeMillis());
        out.println(System.nanoTime());
    } catch (IOException e) {
        Slog.w(TAG, "Unable to add device specific data to the entropy pool", e);
```

```
        } finally {
            if (out != null) {
                out.close();
            }
        }
    }
```

由上述代码可知，即使向 urandom 的 entropy pool 中写入了固定信息，也能增加随机数生成的随机性。从熵的角度考虑，系统的质量越大（即 pool 中的内容越多），该系统就越不稳定。

（3）函数 writeEntropy

函数 writeEntropy 的功能是，读取 urandom 设备的内容到 entropy.dat 文件。具体实现代码如下所示：

```
private void writeEntropy() {
    try {
        RandomBlock.fromFile(randomDevice).toFile(entropyFile);
    } catch (IOException e) {
        Slog.w(TAG, "unable to write entropy", e);
    }
}
```

（4）函数 scheduleEntropyWriter

函数 scheduleEntropyWriter 的功能是，向 EntropyService 内部的 Handler 发送一个 ENTROPY_WHAT 消息。该消息每 3 小时发送一次。收到该消息后，EntropyService 会再次调用 writeEntropy 函数，将 urandom 设备的内容写到 entropy.dat 中。具体实现代码如下所示：

```
private void scheduleEntropyWriter() {
    mHandler.removeMessages(ENTROPY_WHAT);
    mHandler.sendEmptyMessageDelayed(ENTROPY_WHAT, ENTROPY_WRITE_PERIOD);
}
```

通过上面的分析可知，文件 entropy.dat 保存了 urandom 设备内容的快照（每 3 小时更新一次）。当系统重新启动时，EntropyService 又利用这个文件来增加系统的熵，通过这种方式使随机数的生成更加不可预测。

9.1.4 分析 DropBoxManagerService

在 Android 应用中，DropBoxManagerService（DBMS）用于生成和管理系统运行时的一些日志文件。这些日志文件大多记录的是系统或某个应用程序出错时的信息。其中向 SystemServer 添加 DBMS 的代码如下所示：

```
ServiceManager.addService(Context.DROPBOX_SERVICE, //服务名为 "dropbox"
                          new DropBoxManagerService(context,
                          new File("/data/system/dropbox")));
```

（1）分析 DBMS 构造函数

DBMS 构造函数在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\DropBoxManagerService.java
```

DBMS 构造函数 DropBoxManagerService 的具体实现代码如下所示：

```
    public DropBoxManagerService(final Context context, File path) {
        mDropBoxDir = path; //path 指定 dropbox 目录为/data/system/dropbox

        // Set up intent receivers
        mContext = context;
        mContentResolver = context.getContentResolver();

        IntentFilter filter = new IntentFilter();
        filter.addAction(Intent.ACTION_DEVICE_STORAGE_LOW);
        filter.addAction(Intent.ACTION_BOOT_COMPLETED);
//注册一个 Broadcast 监听对象，当系统启动完毕或者设备存储空间不足时，会收到广播
        context.registerReceiver(mReceiver, filter);
//当 Settings 数据库相应项发生变化时候，也需要告知 DBMS 进行相应处理
        mContentResolver.registerContentObserver(
```

```
            Settings.Global.CONTENT_URI, true,
            new ContentObserver(new Handler()) {
                @Override
                public void onChange(boolean selfChange) {

                    //当 Settings 数据库发生变化时候， BroadcastReceiver 的 onReceive 函数
                    //将被调用。注意第二个参数为 null
                    mReceiver.onReceive(context, (Intent) null);
                }
            });

        mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MSG_SEND_BROADCAST) {
                    mContext.sendBroadcastAsUser((Intent)msg.obj, UserHandle.OWNER,
                            android.Manifest.permission.READ_LOGS);
                }
            }
        };
    }
    public void stop() {
        mContext.unregisterReceiver(mReceiver);
    }
```

通过上述代码可知，DBMS 注册一个 BroadcastReceiver 对象，同时会监听 Settings 数据库的变动。其核心逻辑都在此 BroadcastReceiver 的 onReceive 函数中。函数 onReceive 的主要功能是，存储空间不足时，需要删除一些旧的日志文件以节省存储空间。函数 onReceive 的具体实现代码如下所示：

```
    public void onReceive(Context context, Intent intent) {
        if (intent != null && Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
            mBooted = true;
            return;
        }
        mCachedQuotaUptimeMillis = 0;  // Force a re-check of quota size
        new Thread() {
            public void run() {
                try {
                    init();
                    trimToFit();
                } catch (IOException e) {
                    Slog.e(TAG, "Can't init", e);
                }
            }
        }.start();
    }
};
```

函数 onReceive 会在以下 3 种情况发生时被调用：

- 当系统启动完毕时，由 BOOT_COMPLETED 广播触发；
- 当设备存储空间不足时，由 DEVICE_STORAGE_LOW 广播触发；
- 当 Settings 数据库相应项发生变化时候，该函数也会被触发。

（2）添加 dropbox 日志文件

在 Android 5.0 系统中，要想理清一个 Service，最好从它提供的服务开始进行分析。当某个应用程序因为发生异常而崩溃（crash）时，会调用 ActivityManagerService（AMS）的函数 handleApplicationCrash，此函数在如下所示 id 文件中定义：

```
frameworks\base\services\java\com\android\server\am\ActivityManagerService.java
```

函数 handleApplicationCrash 的具体实现代码如下所示：

```
public void handleApplicationCrash(IBinder app, ApplicationErrorReport.CrashInfo
crashInfo) {
    ProcessRecord r = findAppProcess(app, "Crash");
    final String processName = app == null ? "system_server"
            : (r == null ? "unknown" : r.processName);
```

```
        EventLog.writeEvent(EventLogTags.AM_CRASH, Binder.getCallingPid(),
                UserHandle.getUserId(Binder.getCallingUid()), processName,
                r == null ? -1 : r.info.flags,
                crashInfo.exceptionClassName,
                crashInfo.exceptionMessage,
                crashInfo.throwFileName,
                crashInfo.throwLineNumber);
        //调用 addErrorToDropBox 函数，第一个参数是一个字符串，为 "crash"
        addErrorToDropBox("crash", r, processName, null, null, null, null, null, crashInfo);

        crashApplication(r, crashInfo);
    }
```

下面查看函数 addErrorToDropBox，此函数也在文件 ActivityManagerService.java 中实现，具体实现代码如下所示：

```
public void addErrorToDropBox(String eventType,
        ProcessRecord process, String processName, ActivityRecord activity,
        ActivityRecord parent, String subject,
        final String report, final File logFile,
        final ApplicationErrorReport.CrashInfo crashInfo) {
    // NOTE -- this must never acquire the ActivityManagerService lock,
    // otherwise the watchdog may be prevented from resetting the system.

    final String dropboxTag = processClass(process) + "_" + eventType;
    final DropBoxManager dbox = (DropBoxManager)
            mContext.getSystemService(Context.DROPBOX_SERVICE);

    // Exit early if the dropbox isn't configured to accept this report type.
    if (dbox == null || !dbox.isTagEnabled(dropboxTag)) return;

    final StringBuilder sb = new StringBuilder(1024);
    appendDropBoxProcessHeaders(process, processName, sb);
    if (activity != null) {
        sb.append("Activity: ").append(activity.shortComponentName).append("\n");
    }
    if (parent != null && parent.app != null && parent.app.pid != process.pid) {
        sb.append("Parent-Process: ").append(parent.app.processName).append("\n");
    }
    if (parent != null && parent != activity) {
        sb.append("Parent-Activity: ").append(parent.shortComponentName).append("\n");
    }
    if (subject != null) {
        sb.append("Subject: ").append(subject).append("\n");
    }
    sb.append("Build: ").append(Build.FINGERPRINT).append("\n");
    if (Debug.isDebuggerConnected()) {
        sb.append("Debugger: Connected\n");
    }
    sb.append("\n");

    // Do the rest in a worker thread to avoid blocking the caller on I/O
    // (After this point, we shouldn't access AMS internal data structures.)
    Thread worker = new Thread("Error dump: " + dropboxTag) {
        @Override
        public void run() {
            if (report != null) {
                sb.append(report);
            }
            if (logFile != null) {
                try {
                    sb.append(FileUtils.readTextFile(logFile, 128 * 1024, "\n\n
                    [[TRUNCATED]]"));
                } catch (IOException e) {
                    Slog.e(TAG, "Error reading " + logFile, e);
                }
            }
            if (crashInfo != null && crashInfo.stackTrace != null) {
                sb.append(crashInfo.stackTrace);
            }
```

```
            String setting = Settings.Global.ERROR_LOGCAT_PREFIX + dropboxTag;
            int lines = Settings.Global.getInt(mContext.getContentResolver(), setting, 0);
            if (lines > 0) {
                sb.append("\n");

                // Merge several logcat streams, and take the last N lines
                InputStreamReader input = null;
                try {
                    java.lang.Process logcat = new ProcessBuilder("/system/bin/logcat",
                            "-v", "time", "-b", "events", "-b", "system", "-b", "main",
                            "-t", String.valueOf(lines)).redirectErrorStream(true).start();

                    try { logcat.getOutputStream().close(); } catch (IOException e) {}
                    try { logcat.getErrorStream().close(); } catch (IOException e) {}
                    input = new InputStreamReader(logcat.getInputStream());

                    int num;
                    char[] buf = new char[8192];
                    while ((num = input.read(buf)) > 0) sb.append(buf, 0, num);
                } catch (IOException e) {
                    Slog.e(TAG, "Error running logcat", e);
                } finally {
                    if (input != null) try { input.close(); } catch (IOException e) {}
                }
            }

            dbox.addText(dropboxTag, sb.toString());
        }
    };

    if (process == null) {
        // If process is null, we are being called from some internal code
        // and may be about to die -- run this synchronously.
        worker.run();
    } else {
        worker.start();
    }
}
```

由上述代码可知，函数 addErrorToDropBox 的核心功能是生成日志内容，并调用函数 addText 将内容传给 DBMS 的功能。函数 addText 定义在如下所示的文件中：

```
frameworks\base\core\java\android\os\DropBoxManager.java
```

在 DropBoxManager 类中，函数 addText 的实现代码如下所示：

```
public void addText(String tag, String data) {
    try { mService.add(new Entry(tag, 0, data)); } catch (RemoteException e) {}
}
```

在上述代码中实现了 mService 和 DBMS 的交互。DBMS 对外只提供一个 add 函数实现日志添加工作，而 DBM 提供了 3 个函数，分别是 addText、addData、addFile，这样方便使用。

DBM 向 DBMS 传递的数据被封装在一个 Entry 中，DBMS 中的函数 add 在文件 DropBoxManagerService.java 中定义，具体实现代码如下所示：

```
public void add(DropBoxManager.Entry entry) {
    File temp = null;
    OutputStream output = null;
    final String tag = entry.getTag();
    try {
        int flags = entry.getFlags();
        if ((flags & DropBoxManager.IS_EMPTY) != 0) throw new IllegalArgumentException();

        init();
        if (!isTagEnabled(tag)) return;
        long max = trimToFit();
        long lastTrim = System.currentTimeMillis();

        byte[] buffer = new byte[mBlockSize];
```

```
        InputStream input = entry.getInputStream();

        // First, accumulate up to one block worth of data in memory before
        // deciding whether to compress the data or not.

        int read = 0;
        while (read < buffer.length) {
            int n = input.read(buffer, read, buffer.length - read);
            if (n <= 0) break;
            read += n;
        }

        // If we have at least one block, compress it -- otherwise, just write
        // the data in uncompressed form.

        temp = new File(mDropBoxDir, "drop" + Thread.currentThread().getId() + ".tmp");
        int bufferSize = mBlockSize;
        if (bufferSize > 4096) bufferSize = 4096;
        if (bufferSize < 512) bufferSize = 512;
        FileOutputStream foutput = new FileOutputStream(temp);
        output = new BufferedOutputStream(foutput, bufferSize);
        if (read == buffer.length && ((flags & DropBoxManager.IS_GZIPPED) == 0)) {
            output = new GZIPOutputStream(output);
            flags = flags | DropBoxManager.IS_GZIPPED;
        }

        do {
            output.write(buffer, 0, read);

            long now = System.currentTimeMillis();
            if (now - lastTrim > 30 * 1000) {
                max = trimToFit();  // In case data dribbles in slowly
                lastTrim = now;
            }

            read = input.read(buffer);
            if (read <= 0) {
                FileUtils.sync(foutput);
                output.close();  // Get a final size measurement
                output = null;
            } else {
                output.flush();  // So the size measurement is pseudo-reasonable
            }

            long len = temp.length();
            if (len > max) {
                Slog.w(TAG, "Dropping: " + tag + " (" + temp.length() + " > " + max + " bytes)");
                temp.delete();
                temp = null;  // Pass temp = null to createEntry() to leave a tombstone
                break;
            }
        } while (read > 0);

        long time = createEntry(temp, tag, flags);
        temp = null;

        final Intent dropboxIntent = new Intent(DropBoxManager.ACTION_DROPBOX_ENTRY_
        ADDED);
        dropboxIntent.putExtra(DropBoxManager.EXTRA_TAG, tag);
        dropboxIntent.putExtra(DropBoxManager.EXTRA_TIME, time);
        if (!mBooted) {
            dropboxIntent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
        }
        // Call sendBroadcast after returning from this call to avoid deadlock. In particular
        // the caller may be holding the WindowManagerService lock but sendBroadcast requires a
        // lock in ActivityManagerService. ActivityManagerService has been caught holding that
        // very lock while waiting for the WindowManagerService lock.
        mHandler.sendMessage(mHandler.obtainMessage(MSG_SEND_BROADCAST, dropboxIntent));
    } catch (IOException e) {
        Slog.e(TAG, "Can't write: " + tag, e);
    } finally {
```

```
            try { if (output != null) output.close(); } catch (IOException e) {}
            entry.close();
            if (temp != null) temp.delete();
        }
    }
```

从上述代码可知，DBMS 非常爱惜“/data”分区的空间，需要考虑每一个日志文件的压缩以节省存储空间。

（3）DBMS 和 settings 数据库

DBMS 的运行依赖一些配置项。其实除了 DBMS，SystemServer 中很多服务都依赖相关的配置项。这些配置项都是通过 SettingsProvider 操作 Settings 数据库来设置和查询的。SettingsProvider 是系统中很重要的一个 APK，如果将其删除后系统就不能正常启动了。

和系统相关的配置项都在 Settings 数据库的 Secure 表内，具体说明如下所示。

```
//用来判断是否允许记录该 tag 类型的日志文件。默认是允许生成任何 tag 类型的文件
Secure.DROPBOX_TAG_PREFIX+tag: "dropbox:"+tag
//用于控制每个日志文件的存活时间，默认是 3 天。大于 3 天的日志文件就会被删除以节省空间
Secure.DROPBOX_AGE_SECONDS: "dropbox_age_seconds"
//用于控制系统保存的日志文件个数，默认是 1000 个文件
Secure.DROPBOX_MAX_FILES: "dropbox_max_files"
//用于控制 dropbox 目录最多占存储空间容量的比例，默认是 10%
Secure.DROPBOX_QUOTA_PERCENT:"dropbox_quota_percent"
//不允许 dropbox 使用的存储空间的比例，默认是 10%，也就是 dropbox 最多只能使用 90%的空间
Secure.DROPBOX_RESERVE_PERCENT:"dropbox_reserve_percent"
//dropbox 最大能使用的空间大小，默认是 5MB
Secure.DROPBOX_QUOTA_KB:"dropbox_quota_kb"
```

读者可以利用 adb shell 进入/data/data/com.android.providers.settings/databases/目录，然后利用 sqlite3 命令操作 settings.db，通过里面的表 Secure 可以了解相关内容。不过系统中的很多选项在该表中都没有相关设置，因此，实际运行时都会使用代码中设置的默认值。

9.1.5　分析 DiskStatsService

在 Android 5.0 系统中，DiskStatsService 在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\DiskStatsService.java
```

文件 DiskStatsService.java 的具体实现代码如下所示：

```
public class DiskStatsService extends Binder {
    private static final String TAG = "DiskStatsService";

    private final Context mContext;

    public DiskStatsService(Context context) {
        mContext = context;
    }

    @Override
    protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
        mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DUMP, TAG);

        // Run a quick-and-dirty performance test: write 512 bytes
        byte[] junk = new byte[512];
        for (int i = 0; i < junk.length; i++) junk[i] = (byte) i;  // Write nonzero bytes

        File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
        FileOutputStream fos = null;
        IOException error = null;

        long before = SystemClock.uptimeMillis();
        try {
            fos = new FileOutputStream(tmp);
            fos.write(junk);
        } catch (IOException e) {
            error = e;
        } finally {
```

```
            try { if (fos != null) fos.close(); } catch (IOException e) {}
        }

        long after = SystemClock.uptimeMillis();
        if (tmp.exists()) tmp.delete();

        if (error != null) {
            pw.print("Test-Error: ");
            pw.println(error.toString());
        } else {
            pw.print("Latency: ");
            pw.print(after - before);
            pw.println("ms [512B Data Write]");
        }

        reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
        reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
        reportFreeSpace(new File("/system"), "System", pw);

        // TODO: Read /proc/yaffs and report interesting values;
        // add configurable (through args) performance test parameters.
    }

    private void reportFreeSpace(File path, String name, PrintWriter pw) {
        try {
            StatFs statfs = new StatFs(path.getPath());
            long bsize = statfs.getBlockSize();
            long avail = statfs.getAvailableBlocks();
            long total = statfs.getBlockCount();
            if (bsize <= 0 || total <= 0) {
                throw new IllegalArgumentException(
                        "Invalid stat: bsize=" + bsize + " avail=" + avail + " total=" + total);
            }

            pw.print(name);
            pw.print("-Free: ");
            pw.print(avail * bsize / 1024);
            pw.print("K / ");
            pw.print(total * bsize / 1024);
            pw.print("K total = ");
            pw.print(avail * 100 / total);
            pw.println("% free");
        } catch (IllegalArgumentException e) {
            pw.print(name);
            pw.print("-Error: ");
            pw.println(e.toString());
            return;
        }
    }
}
```

从上述代码可以看出：虽然 DiskStatsService 从 Binder 中派生，但是并没有实现任何接口，也就是说，DiskStatsService 没有任何可调用的业务函数。但是，在系统中为什么会存在这样的服务呢？要想解决这个问题，需要先了解系统中的命令——dumpsys，此命令用于打印系统中指定服务的信息，在如下所示的文件中定义：

```
frameworks\native\cmds\dumpsys\dumpsys.cpp
```

文件 dumpsys.cpp 的具体实现代码如下所示：

```
#define LOG_TAG "dumpsys"

#include <utils/Log.h>
#include <binder/Parcel.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include <utils/TextOutput.h>
#include <utils/Vector.h>

#include <getopt.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

using namespace android;

static int sort_func(const String16* lhs, const String16* rhs)
{
    return lhs->compare(*rhs);
}

int main(int argc, char* const argv[])
{
    signal(SIGPIPE, SIG_IGN);
    sp<IServiceManager> sm = defaultServiceManager();
    fflush(stdout);
    if (sm == NULL) {
        ALOGE("Unable to get default service manager!");
        aerr << "dumpsys: Unable to get default service manager!" << endl;
        return 20;
    }

    Vector<String16> services;
    Vector<String16> args;
    if (argc == 1) {
        services = sm->listServices();
        services.sort(sort_func);
        args.add(String16("-a"));
    } else {
        services.add(String16(argv[1]));
        for (int i=2; i<argc; i++) {
            args.add(String16(argv[i]));
        }
    }

    const size_t N = services.size();

    if (N > 1) {
        // first print a list of the current services
        aout << "Currently running services:" << endl;

        for (size_t i=0; i<N; i++) {
            sp<IBinder> service = sm->checkService(services[i]);
            if (service != NULL) {
                aout << "  " << services[i] << endl;
            }
        }
    }

    for (size_t i=0; i<N; i++) {
        sp<IBinder> service = sm->checkService(services[i]);
        if (service != NULL) {
            if (N > 1) {
                aout << "------------------------------------------------------------"
                       "-------------------" << endl;
                aout << "DUMP OF SERVICE " << services[i] << ":" << endl;
            }
            int err = service->dump(STDOUT_FILENO, args);
            if (err != 0) {
                aerr << "Error dumping service info: (" << strerror(err)
                       << ") " << services[i] << endl;
            }
        } else {
            aerr << "Can't find service: " << services[i] << endl;
        }
    }

    return 0;
}
```

通过上述代码可知，dumpsys 通过 Binder 调用某个 Service 的 dump 函数。上述代码的具体实现流程如下所示。

（1）先获取与 ServiceManager 进程通信的 BpServiceManager 对象。

（2）如果输入参数个数为 1，则先查询在 SM 中注册的所有 Service。

（3）将 Service 排序。

（4）指定查询某个 Service。

（5）保存剩余参数，以后可以传给 Service 的 dump 函数。

（6）通过 Binder 调用该 Service 的 dump 函数，将 args 也传给 dump 函数。

接下来看文件 DiskStatsService.java 中的函数 dump，具体实现代码如下所示：

```
protected void dump(FileDescriptor fd, PrintWriter pw, String[] args) {
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.DUMP, TAG);

    // Run a quick-and-dirty performance test: write 512 bytes
    byte[] junk = new byte[512];
    for (int i = 0; i < junk.length; i++) junk[i] = (byte) i;  // Write nonzero bytes

    File tmp = new File(Environment.getDataDirectory(), "system/perftest.tmp");
    FileOutputStream fos = null;
    IOException error = null;

    long before = SystemClock.uptimeMillis();
    try {
        fos = new FileOutputStream(tmp);
        fos.write(junk);
    } catch (IOException e) {
        error = e;
    } finally {
        try { if (fos != null) fos.close(); } catch (IOException e) {}
    }

    long after = SystemClock.uptimeMillis();
    if (tmp.exists()) tmp.delete();

    if (error != null) {
        pw.print("Test-Error: ");
        pw.println(error.toString());
    } else {
        pw.print("Latency: ");
        pw.print(after - before);
        pw.println("ms [512B Data Write]");
    }

    reportFreeSpace(Environment.getDataDirectory(), "Data", pw);
    reportFreeSpace(Environment.getDownloadCacheDirectory(), "Cache", pw);
    reportFreeSpace(new File("/system"), "System", pw);
}
```

从上述代码可知，DiskStatsService 没有实现任何业务接口，只是为了调试而存在。

9.1.6 分析 DeviceStorageManagerService（监测系统内存存储空间的状态）

在 Android 5.0 系统中，DeviceStorageManagerService（DSMS）用于监测系统内部存储空间的状态，添加该服务的代码如下所示：

```
//DSMS 的服务名为 devicestoragemonitor
ServiceManager.addService(DeviceStorageMonitorService.SERVICE,
new DeviceStorageMonitorService(context));
```

DSMS 的构造函数在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\DeviceStorageMonitorService.java
```

函数 DeviceStorageMonitorService 的具体实现代码如下所示：

```
public DeviceStorageMonitorService(Context context) {
    mLastReportedFreeMemTime = 0;
```

```
        mContext = context;
        mContentResolver = mContext.getContentResolver();
        mDataFileStats = new StatFs(DATA_PATH);//获取 data 分区的信息
        mSystemFileStats = new StatFs(SYSTEM_PATH);// 获取 system 分区的信息
        mCacheFileStats = new StatFs(CACHE_PATH);// 获取 cache 分区的信息
        //获得 data 分区的总大小
        mTotalMemory = ((long)mDataFileStats.getBlockCount() *
                        mDataFileStats.getBlockSize())/100L;
        /*
        创建 3 个 Intent，分别用于通知存储空间不足、存储空间恢复正常和存储空间满。
        由于设置了 REGISTERED_ONLY_BEFORE_BOOT 标志，这 3 个 Intent 广播只能由
        系统服务接收
        */
       mStorageLowIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_LOW);
       mStorageLowIntent.addFlags(
                        Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
        mStorageOkIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_OK);
        mStorageOkIntent.addFlags(
                    Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
        mStorageFullIntent = new Intent(Intent.ACTION_DEVICE_STORAGE_FULL);
        mStorageFullIntent.addFlags(
                    Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);
        mStorageNotFullIntent = new
                    Intent(Intent.ACTION_DEVICE_STORAGE_NOT_FULL);
        mStorageNotFullIntent.addFlags(
                    Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT);

        //查询 Settings 数据库中 sys_storage_threshold_percentage 的值，默认是 10，
        //即当 data 空间只剩下 10%时，就认为空间不足
        mMemLowThreshold = getMemThreshold();
       //查询 Settings 数据库中的 sys_storage_full_threshold_bytes 的值，默认是 1MB，
       //即当 data 分区只剩 1MB 时，就认为空间已满，剩下的这 1MB 空间保留给系统自用
        mMemFullThreshold = getMemFullThreshold();
        //检查内存
        checkMemory(true);
    }
```

再来看函数 checkMemory，此函数也是在文件 DeviceStorageMonitorService.java 中定义的，具体实现代码如下所示：

```
    private final void checkMemory(boolean checkCache) {
        //if the thread that was started to clear cache is still running do nothing till its
        //finished clearing cache. Ideally this flag could be modified by clearCache
        // and should be accessed via a lock but even if it does this test will fail now and
        //hopefully the next time this flag will be set to the correct value.
        if(mClearingCache) {
            if(localLOGV) Slog.i(TAG, "Thread already running just skip");
            //make sure the thread is not hung for too long
            long diffTime = System.currentTimeMillis() - mThreadStartTime;
            if(diffTime > (10*60*1000)) {
                Slog.w(TAG, "Thread that clears cache file seems to run for ever");
            }
        } else {
            restatDataDir();
            if (localLOGV)  Slog.v(TAG, "freeMemory="+mFreeMem);

            //post intent to NotificationManager to display icon if necessary
            if (mFreeMem < mMemLowThreshold) {
                if (checkCache) {
                    if (mFreeMem < mMemCacheStartTrimThreshold) {
                        if ((mFreeMemAfterLastCacheClear-mFreeMem)
                                >= ((mMemLowThreshold-mMemCacheStartTrimThreshold)/4)) {
                            mThreadStartTime = System.currentTimeMillis();
                            mClearSucceeded = false;
                            clearCache();
                        }
                    }
                } else {
                    mFreeMemAfterLastCacheClear = mFreeMem;
                    if (!mLowMemFlag) {
                        // We tried to clear the cache, but that didn't get us
```

```
                // below the low storage limit.  Tell the user.
                Slog.i(TAG, "Running low on memory. Sending notification");
                sendNotification();
                mLowMemFlag = true;
            } else {
                if (localLOGV) Slog.v(TAG, "Running low on memory " +
                        "notification already sent. do nothing");
            }
        }
    } else {
        mFreeMemAfterLastCacheClear = mFreeMem;
        if (mLowMemFlag) {
            Slog.i(TAG, "Memory available. Cancelling notification");
            cancelNotification();
            mLowMemFlag = false;
        }
    }
    if (mFreeMem < mMemFullThreshold) {
        if (!mMemFullFlag) {
            sendFullNotification();
            mMemFullFlag = true;
        }
    } else {
        if (mMemFullFlag) {
            cancelFullNotification();
            mMemFullFlag = false;
        }
    }
}
if(localLOGV) Slog.i(TAG, "Posting Message again");
//keep posting messages to itself periodically
postCheckMemoryMsg(true, DEFAULT_CHECK_INTERVAL);
}
```

当空间不足时，DSMS 会先使用函数 clearCache 进行处理，此函数内部会与 PackageManager Service（简称 PKMS）进行交互。函数 clearCache 在文件 DeviceStorageManagerService.java 中定义，具体实现代码如下所示：

```
private final void clearCache() {
    if (mClearCacheObserver == null) {
        // Lazy instantiation
        mClearCacheObserver = new CachePackageDataObserver();
    }
    mClearingCache = true;
    try {
        if (localLOGV) Slog.i(TAG, "Clearing cache");
        IPackageManager.Stub.asInterface(ServiceManager.getService("package")).
                freeStorageAndNotify(mMemCacheTrimToThreshold, mClearCacheObserver);
    } catch (RemoteException e) {
        Slog.w(TAG, "Failed to get handle for PackageManger Exception: "+e);
        mClearingCache = false;
        mClearSucceeded = false;
    }
}
```

CachePackageDataObserver 是 DSMS 定义的内部类，其中的函数 onRemoveCompleted 用于重新发送消息，让 DSMS 再检测一次存储空间。函数 DeviceStorageManagerService 并没有重载 dump 函数。

9.1.7 分析 SamplingProfilerService

在 Android 5.0 系统的源代码中，添加 SamplingProfilerService 服务的代码如下所示：

```
ServiceManager.addService("samplingprofiler",//服务名
                          new SamplingProfilerService(context));
```

在本节的内容中，将详细分析 Android 5.0 中 SamplingProfilerService 的源代码。

（1）分析 SamplingProfilerService 构造函数

SamplingProfilerService 的构造函数在如下所示的文件中实现：

```
frameworks\base\services\java\com\android\server\SamplingProfilerService.java
```

在文件 SamplingProfilerService.java 中，函数 SamplingProfilerService 的具体实现代码如下所示：

```
public SamplingProfilerService(Context context) {
    //注册一个 CotentObserver，用于监测 Settings 数据库的变化
     registerSettingObserver(context);
     startWorking(context);//执行 startWorking 函数，见下文的分析
}
```

上述代码的核心是函数 startWorking，此函数在文件 SamplingProfilerService.java 中定义，具体实现代码如下所示：

```
private void startWorking(Context context) {
    if (LOCAL_LOGV) Slog.v(TAG, "starting SamplingProfilerService!");

    final DropBoxManager dropbox =
            (DropBoxManager) context.getSystemService(Context.DROPBOX_SERVICE);
    File[] snapshotFiles = new File(SNAPSHOT_DIR).listFiles();
    for (int i = 0; snapshotFiles != null && i < snapshotFiles.length; i++) {
        handleSnapshotFile(snapshotFiles[i], dropbox);
    }
    snapshotObserver = new FileObserver(SNAPSHOT_DIR, FileObserver.ATTRIB) {
        @Override
        public void onEvent(int event, String path) {
            handleSnapshotFile(new File(SNAPSHOT_DIR, path), dropbox);
        }
    };
    snapshotObserver.startWatching();

    if (LOCAL_LOGV) Slog.v(TAG, "SamplingProfilerService activated");
}
```

通过上述代码可知，SamplingProfilerService 本身并不提供性能统计的功能。统计功能是通过类 SamplingProfilerIntegration 实现的，这个类封装了一个 SamplingProfiler（由 Dalvik 虚拟机提供）对象，并提供了方便利用的函数进行性能统计。

（2）分析 SamplingProfilerIntegration

通过使用 SamplingProfilerIntegration 可以进行性能统计。在 Andorid 系统中有很多重要进程都需要对性能进行分析，如 Zygote，其相关代码在如下所示的文件中实现：

```
frameworks\base\core\java\com\android\internal\os\ZygoteInit.java
```

在文件 ZygoteInit.java 中，和性能分析相关的代码如下所示：

```
public static void main(String argv[]) {
    try {
        // Start profiling the zygote initialization.
        SamplingProfilerIntegration.start();

        registerZygoteSocket();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
            SystemClock.uptimeMillis());
        preload();
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
            SystemClock.uptimeMillis());

        // Finish profiling the zygote initialization.
        SamplingProfilerIntegration.writeZygoteSnapshot();

        // Do an initial gc to clean up after startup
        gc();

        // If requested, start system server directly from Zygote
        if (argv.length != 2) {
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }

        if (argv[1].equals("start-system-server")) {
            startSystemServer();
        } else if (!argv[1].equals("")) {
```

```
            throw new RuntimeException(argv[0] + USAGE_STRING);
        }

        Log.i(TAG, "Accepting command socket connections");

        if (ZYGOTE_FORK_MODE) {
            runForkMode();
        } else {
            runSelectLoopMode();
        }

        closeServerSocket();
    } catch (MethodAndArgsCaller caller) {
        caller.run();
    } catch (RuntimeException ex) {
        Log.e(TAG, "Zygote died with exception", ex);
        closeServerSocket();
        throw ex;
    }
}
```

在上述代码中，函数 start 在如下所示的文件中实现：

```
frameworks\base\core\java\com\android\internal\os\SamplingProfilerIntegration.java
```

函数 start 的具体实现代码如下所示：

```
public static void start() {
    if (!enabled) {      //判断是否开启性能统计
        return;
    }
    if (samplingProfiler != null) {
        Log.e(TAG, "SamplingProfilerIntegration already started at " + new Date
        (startMillis));
        return;
    }

    ThreadGroup group = Thread.currentThread().getThreadGroup();
    //创建一个 Dalvik 的 SamplingProfiler
    SamplingProfiler.ThreadSet threadSet = SamplingProfiler.newThreadGroupTheadSet(group);
    samplingProfiler = new SamplingProfiler(samplingProfilerDepth, threadSet);
    //启动统计
    samplingProfiler.start(samplingProfilerMilliseconds);
    startMillis = System.currentTimeMillis();
}
```

在上述代码中，使用该类的 static 语句来判断是否启动性能统计的 enable 变量由谁控制。在文件 SamplingProfilerIntegration.java 中，static 语句的实现代码如下所示：

```
static {
    samplingProfilerMilliseconds = SystemProperties.getInt("persist.sys.profiler_ms", 0);
    samplingProfilerDepth = SystemProperties.getInt("persist.sys.profiler_depth", 4);
    if (samplingProfilerMilliseconds > 0) {
        File dir = new File(SNAPSHOT_DIR);
        dir.mkdirs();
        // the directory needs to be writable to anybody to allow file writing
        dir.setWritable(true, false);
        // the directory needs to be executable to anybody to allow file creation
        dir.setExecutable(true, false);
        if (dir.isDirectory()) {
            snapshotWriter = Executors.newSingleThreadExecutor(new ThreadFactory() {
                    public Thread newThread(Runnable r) {
                        return new Thread(r, TAG);
                    }
                });
            enabled = true;
            Log.i(TAG, "Profiling enabled. Sampling interval ms: "
                 + samplingProfilerMilliseconds);
        } else {
            snapshotWriter = null;
            enabled = true;
            Log.w(TAG, "Profiling setup failed. Could not create " + SNAPSHOT_DIR);
```

```
        }
    } else {
        snapshotWriter = null;
        enabled = false;
        Log.i(TAG, "Profiling disabled.");
    }
}
```

由上述代码可知，enable 的控制在 static 语句中实现，这表明要使用性能统计，就必须重新启动要统计的进程。

当启动性能统计后，需要输出统计文件，这此功能由函数 writeZygoteSnapshot 实现。在文件 SamplingProfilerIntegration.java 中，函数 writeZygoteSnapshot 的具体实现代码如下所示：

```
public static void writeZygoteSnapshot() {
    if (!enabled) {
        return;
    }
    writeSnapshotFile("zygote", null);
    samplingProfiler.shutdown();
    samplingProfiler = null;
    startMillis = 0;
}
```

在上述代码中，调用了 writeSnapshotFile 函数，其第一个参数为 zygote，用于表示进程名。writeSnapshotFile 函数比较简单，功能就是在 shots 目录下生成一个统计文件，统计文件的名称由两部分组成，合起来就是“进程名_开始性能统计的时刻.snapshot”。另外，writeSnapshotfile 内部会调用 generateSnapshotHeader 函数在该统计文件头部写一些特定的信息，如版本号、编译信息等。在文件 SamplingProfilerIntegration.java 中，函数 writeSnapshotFile 的具体实现代码如下所示：

```
private static void writeSnapshotFile(String processName, PackageInfo packageInfo) {
    if (!enabled) {
        return;
    }
    samplingProfiler.stop();
    String name = processName.replaceAll(":", ".");
    String path = SNAPSHOT_DIR + "/" + name + "-" + startMillis + ".snapshot";
    long start = System.currentTimeMillis();
    OutputStream outputStream = null;
    try {
        outputStream = new BufferedOutputStream(new FileOutputStream(path));
        PrintStream out = new PrintStream(outputStream);
        generateSnapshotHeader(name, packageInfo, out);
        if (out.checkError()) {
            throw new IOException();
        }
        BinaryHprofWriter.write(samplingProfiler.getHprofData(), outputStream);
    } catch (IOException e) {
        Log.e(TAG, "Error writing snapshot to " + path, e);
        return;
    } finally {
        IoUtils.closeQuietly(outputStream);
    }
    // set file readable to the world so that SamplingProfilerService
    // can put it to dropbox
    new File(path).setReadable(true, false);

    long elapsed = System.currentTimeMillis() - start;
    Log.i(TAG, "Wrote snapshot " + path + " in " + elapsed + "ms.");
    samplingProfiler.start(samplingProfilerMilliseconds);
}
```

SamplingProfilerIntegration 的核心是类 SamplingProfiler，这个类定义在如下所示 id 文件中：

```
libcore/dalvik /src/main/java/dalvik/system/profiler/SamplingProfiler.java
```

文件 SamplingProfiler.java 的具体实现代码如下所示：

```
public final class SamplingProfiler {
    private final Map<HprofData.StackTrace, int[]> stackTraces
```

```
        = new HashMap<HprofData.StackTrace, int[]>();
    private final HprofData hprofData = new HprofData(stackTraces);
    private final Timer timer = new Timer("SamplingProfiler", true);
    private Sampler sampler;
    private final int depth;
    private final ThreadSet threadSet;
    private int nextThreadId = 200001;
    private int nextStackTraceId = 300001;
    private int nextObjectId = 1;
    private Thread[] currentThreads = new Thread[0];
    private final Map<Thread, Integer> threadIds = new HashMap<Thread, Integer>();
    private final HprofData.StackTrace mutableStackTrace = new HprofData.StackTrace();
    private final ThreadSampler threadSampler;
    public SamplingProfiler(int depth, ThreadSet threadSet) {
        this.depth = depth;
        this.threadSet = threadSet;
        this.threadSampler = findDefaultThreadSampler();
        threadSampler.setDepth(depth);
        hprofData.setFlags(BinaryHprof.ControlSettings.CPU_SAMPLING.bitmask);
        hprofData.setDepth(depth);
    }

    private static ThreadSampler findDefaultThreadSampler() {
        if ("Dalvik Core Library".equals(System.getProperty("java.specification.name"))) {
            String className = "dalvik.system.profiler.DalvikThreadSampler";
            try {
                return (ThreadSampler) Class.forName(className).newInstance();
            } catch (Exception e) {
                System.out.println("Problem creating " + className + ": " + e);
            }
        }
        return new PortableThreadSampler();
    }

    public static interface ThreadSet {
        public Thread[] threads();
    }
    public static ThreadSet newArrayThreadSet(Thread... threads) {
        return new ArrayThreadSet(threads);
    }
    private static class ArrayThreadSet implements ThreadSet {
        private final Thread[] threads;
        public ArrayThreadSet(Thread... threads) {
            if (threads == null) {
                throw new NullPointerException("threads == null");
            }
            this.threads = threads;
        }
        public Thread[] threads() {
            return threads;
        }
    }
    public static ThreadSet newThreadGroupTheadSet(ThreadGroup threadGroup) {
        return new ThreadGroupThreadSet(threadGroup);
    }
    private static class ThreadGroupThreadSet implements ThreadSet {
        private final ThreadGroup threadGroup;
        private Thread[] threads;
        private int lastThread;

        public ThreadGroupThreadSet(ThreadGroup threadGroup) {
            if (threadGroup == null) {
                throw new NullPointerException("threadGroup == null");
            }
            this.threadGroup = threadGroup;
            resize();
        }

        private void resize() {
            int count = threadGroup.activeCount();
            threads = new Thread[count*2];
```

```
            lastThread = 0;
        }

        public Thread[] threads() {
            int threadCount;
            while (true) {
                threadCount = threadGroup.enumerate(threads);
                if (threadCount == threads.length) {
                    resize();
                } else {
                    break;
                }
            }
            if (threadCount < lastThread) {
                // avoid retaining pointers to threads that have ended
                Arrays.fill(threads, threadCount, lastThread, null);
            }
            lastThread = threadCount;
            return threads;
        }
    }
    public void start(int interval) {
        if (interval < 1) {
            throw new IllegalArgumentException("interval < 1");
        }
        if (sampler != null) {
            throw new IllegalStateException("profiling already started");
        }
        sampler = new Sampler();
        hprofData.setStartMillis(System.currentTimeMillis());
        timer.scheduleAtFixedRate(sampler, 0, interval);
    }
    public void stop() {
        if (sampler == null) {
            return;
        }
        synchronized(sampler) {
            sampler.stop = true;
            while (!sampler.stopped) {
                try {
                    sampler.wait();
                } catch (InterruptedException ignored) {
                }
            }
        }
        sampler = null;
    }
    public void shutdown() {
        stop();
        timer.cancel();
    }
    public HprofData getHprofData() {
        if (sampler != null) {
            throw new IllegalStateException("cannot access hprof data while sampling");
        }
        return hprofData;
    }
    private class Sampler extends TimerTask {

        private boolean stop;
        private boolean stopped;

        private Thread timerThread;

        public void run() {
            synchronized(this) {
                if (stop) {
                    cancel();
                    stopped = true;
                    notifyAll();
                    return;
```

```
            }
        }

        if (timerThread == null) {
            timerThread = Thread.currentThread();
        }

        // process thread creation and death first so that we
        // assign thread ids to any new threads before allocating
        // new stacks for them
        Thread[] newThreads = threadSet.threads();
        if (!Arrays.equals(currentThreads, newThreads)) {
            updateThreadHistory(currentThreads, newThreads);
            currentThreads = newThreads.clone();
        }

        for (Thread thread : currentThreads) {
            if (thread == null) {
                break;
            }
            if (thread == timerThread) {
                continue;
            }

            StackTraceElement[] stackFrames = threadSampler.getStackTrace(thread);
            if (stackFrames == null) {
                continue;
            }
            recordStackTrace(thread, stackFrames);
        }
    }

    private void recordStackTrace(Thread thread, StackTraceElement[] stackFrames) {
        Integer threadId = threadIds.get(thread);
        if (threadId == null) {
            throw new IllegalArgumentException("Unknown thread " + thread);
        }
        mutableStackTrace.threadId = threadId;
        mutableStackTrace.stackFrames = stackFrames;

        int[] countCell = stackTraces.get(mutableStackTrace);
        if (countCell == null) {
            countCell = new int[1];
            // cloned because the ThreadSampler may reuse the array
            StackTraceElement[] stackFramesCopy = stackFrames.clone();
            HprofData.StackTrace stackTrace
                    =new HprofData.StackTrace(nextStackTraceId++,threadId,stackFramesCopy);
            hprofData.addStackTrace(stackTrace, countCell);
        }
        countCell[0]++;
    }

    private void updateThreadHistory(Thread[] oldThreads, Thread[] newThreads) {
        Set<Thread> n = new HashSet<Thread>(Arrays.asList(newThreads));
        Set<Thread> o = new HashSet<Thread>(Arrays.asList(oldThreads));

        // added = new-old
        Set<Thread> added = new HashSet<Thread>(n);
        added.removeAll(o);

        // removed = old-new
        Set<Thread> removed = new HashSet<Thread>(o);
        removed.removeAll(n);

        for (Thread thread : added) {
            if (thread == null) {
                continue;
            }
            if (thread == timerThread) {
                continue;
            }
```

```
                addStartThread(thread);
            }
            for (Thread thread : removed) {
                if (thread == null) {
                    continue;
                }
                if (thread == timerThread) {
                    continue;
                }
                addEndThread(thread);
            }
        }
        private void addStartThread(Thread thread) {
            if (thread == null) {
                throw new NullPointerException("thread == null");
            }
            int threadId = nextThreadId++;
            Integer old = threadIds.put(thread, threadId);
            if (old != null) {
                throw new IllegalArgumentException("Thread already registered as " + old);
            }

            String threadName = thread.getName();
            // group will become null when thread is terminated
            ThreadGroup group = thread.getThreadGroup();
            String groupName = group == null ? null : group.getName();
            ThreadGroup parentGroup = group == null ? null : group.getParent();
            String parentGroupName = parentGroup == null ? null : parentGroup.getName();

            HprofData.ThreadEvent event
                    = HprofData.ThreadEvent.start(nextObjectId++, threadId,
                                                  threadName, groupName, parentGroupName);
            hprofData.addThreadEvent(event);
        }

        /**
         * Record that a thread has disappeared.
         */
        private void addEndThread(Thread thread) {
            if (thread == null) {
                throw new NullPointerException("thread == null");
            }
            Integer threadId = threadIds.remove(thread);
            if (threadId == null) {
                throw new IllegalArgumentException("Unknown thread " + thread);
            }
            HprofData.ThreadEvent event = HprofData.ThreadEvent.end(threadId);
            hprofData.addThreadEvent(event);
        }
    }
}
```

9.2 分析应用程序进程

在启动 Android 应用程序过程中，除了可以获得虚拟机实例外，还可以获得一个消息循环和一个 Binder 线程池。这样在应用程序中运行的组件，可以使用系统的信息处理机制和 Binder 通信机制实现自己的业务逻辑。在本节详细分析创建应用程序的实现源代码，为读者步入本书后面知识的学习打下基础。

9.2.1　创建应用程序

在 Android 5.0 系统中，当 ActivityManagerService 创建新进程来启动某个应用程序组件时，会调用类 ActivityManagerService 中的函数 startProcessLocked 向“孵化”进程 Zygote 发送创建应用程序进程的请求。函数 startProcessLocked 在如下所示的文件中定义：

frameworks\base\services\java\com\android\server\am\ActivityManagerService.java

函数 startProcessLocked 的具体实现代码如下所示：

```
private final void startProcessLocked(ProcessRecord app,
        String hostingType, String hostingNameStr) {
    if (app.pid > 0 && app.pid != MY_PID) {
        synchronized (mPidsSelfLocked) {
            mPidsSelfLocked.remove(app.pid);
            mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);
        }
        app.setPid(0);
    }

    if (DEBUG_PROCESSES && mProcessesOnHold.contains(app)) Slog.v(TAG,
            "startProcessLocked removing on hold: " + app);
    mProcessesOnHold.remove(app);

    updateCpuStats();

    System.arraycopy(mProcDeaths, 0, mProcDeaths, 1, mProcDeaths.length-1);
    mProcDeaths[0] = 0;
    //获取创建应用程序进程的用户 ID 和用户组 ID
    try {
        int uid = app.uid;

        int[] gids = null;
        int mountExternal = Zygote.MOUNT_EXTERNAL_NONE;
        if (!app.isolated) {
            int[] permGids = null;
            try {
                final PackageManager pm = mContext.getPackageManager();
                permGids = pm.getPackageGids(app.info.packageName);

                if (Environment.isExternalStorageEmulated()) {
                    if (pm.checkPermission(
                            android.Manifest.permission.ACCESS_ALL_EXTERNAL_STORAGE,
                            app.info.packageName) == PERMISSION_GRANTED) {
                        mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL;
                    } else {
                        mountExternal = Zygote.MOUNT_EXTERNAL_MULTIUSER;
                    }
                }
            } catch (PackageManager.NameNotFoundException e) {
                Slog.w(TAG, "Unable to retrieve gids", e);
            }
            if (permGids == null) {
                gids = new int[1];
            } else {
                gids = new int[permGids.length + 1];
                System.arraycopy(permGids, 0, gids, 1, permGids.length);
            }
            gids[0] = UserHandle.getSharedAppGid(UserHandle.getAppId(uid));
        }
        if (mFactoryTest != SystemServer.FACTORY_TEST_OFF) {
            if (mFactoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL
                    && mTopComponent != null
                    && app.processName.equals(mTopComponent.getPackageName())) {
                uid = 0;
            }
            if (mFactoryTest == SystemServer.FACTORY_TEST_HIGH_LEVEL
                    && (app.info.flags&ApplicationInfo.FLAG_FACTORY_TEST) != 0) {
                uid = 0;
            }
        }
        int debugFlags = 0;
        if ((app.info.flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
            debugFlags |= Zygote.DEBUG_ENABLE_DEBUGGER;
            debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
        }
        if ((app.info.flags & ApplicationInfo.FLAG_VM_SAFE_MODE) != 0 ||
            Zygote.systemInSafeMode == true) {
```

```
            debugFlags |= Zygote.DEBUG_ENABLE_SAFEMODE;
        }
        if ("1".equals(SystemProperties.get("debug.checkjni"))) {
            debugFlags |= Zygote.DEBUG_ENABLE_CHECKJNI;
        }
        if ("1".equals(SystemProperties.get("debug.jni.logging"))) {
            debugFlags |= Zygote.DEBUG_ENABLE_JNI_LOGGING;
        }
        if ("1".equals(SystemProperties.get("debug.assert"))) {
            debugFlags |= Zygote.DEBUG_ENABLE_ASSERT;
        }
        //调用函数 start 创建应用程序进程
        Process.ProcessStartResult startResult = Process.start("android.app.Activity
        hread",
                app.processName, uid, uid, gids, debugFlags, mountExternal,
                app.info.targetSdkVersion, app.info.seinfo, null);
        BatteryStatsImpl bs = app.batteryStats.getBatteryStats();
        synchronized (bs) {
            if (bs.isOnBattery()) {
                app.batteryStats.incStartsLocked();
            }
        }

        EventLog.writeEvent(EventLogTags.AM_PROC_START,
                UserHandle.getUserId(uid), startResult.pid, uid,
                app.processName, hostingType,
                hostingNameStr != null ? hostingNameStr : "");
        if (app.persistent) {
            Watchdog.getInstance().processStarted(app.processName,
            startResult.pid);
        }
        StringBuilder buf = mStringBuilder;
        buf.setLength(0);
        buf.append("Start proc ");
        buf.append(app.processName);
        buf.append(" for ");
        buf.append(hostingType);
        if (hostingNameStr != null) {
            buf.append(" ");
            buf.append(hostingNameStr);
        }
        buf.append(": pid=");
        buf.append(startResult.pid);
        buf.append(" uid=");
        buf.append(uid);
        buf.append(" gids={");
        if (gids != null) {
            for (int gi=0; gi<gids.length; gi++) {
                if (gi != 0) buf.append(", ");
                buf.append(gids[gi]);
            }
        }
        buf.append("}");
        Slog.i(TAG, buf.toString());
        app.setPid(startResult.pid);
        app.usingWrapper = startResult.usingWrapper;
        app.removed = false;
        synchronized (mPidsSelfLocked) {
            this.mPidsSelfLocked.put(startResult.pid, app);
            Message msg = mHandler.obtainMessage(PROC_START_TIMEOUT_MSG);
            msg.obj = app;
            mHandler.sendMessageDelayed(msg, startResult.usingWrapper
                    ? PROC_START_TIMEOUT_WITH_WRAPPER : PROC_START_TIMEOUT);
        }
    } catch (RuntimeException e) {
        // XXX do better error recovery.
        app.setPid(0);
        Slog.e(TAG, "Failure starting process " + app.processName, e);
    }
}
```

类 Process 中的函数 start 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
    public static final ProcessStartResult start(final String processClass,
                                final String niceName,
                                int uid, int gid, int[] gids,
                                int debugFlags, int mountExternal,
                                int targetSdkVersion,
                                String seInfo,
                                String[] zygoteArgs) {
        try {
//调用函数 startViaZygote 让 Zygote 进程创建一个应用程序进程
            return startViaZygote(processClass, niceName, uid, gid, gids,
                    debugFlags, mountExternal, targetSdkVersion, seInfo, zygoteArgs);
        } catch (ZygoteStartFailedEx ex) {
            Log.e(LOG_TAG,
                    "Starting VM process through Zygote failed");
            throw new RuntimeException(
                    "Starting VM process through Zygote failed", ex);
        }
    }
```

在上述代码中用到了函数 startViaZygote，功能是将要创建的应用程序进程的启动参数保存在字符串列表 argsForZygote 中，并调用函数 zygoteSendArgsAndGetResult 请求进程 Zygote 创建应用程序。函数 startViaZygote 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
private static ProcessStartResult startViaZygote(final String processClass,
                                final String niceName,
                                final int uid, final int gid,
                                final int[] gids,
                                int debugFlags, int mountExternal,
                                int targetSdkVersion,
                                String seInfo,
                                String[] extraArgs)
                                throws ZygoteStartFailedEx {
    synchronized(Process.class) {
        ArrayList<String> argsForZygote = new ArrayList<String>();
        // --runtime-init, --setuid=, --setgid=,
        // and --setgroups= must go first
        argsForZygote.add("--runtime-init");
        argsForZygote.add("--setuid=" + uid);
        argsForZygote.add("--setgid=" + gid);
        if ((debugFlags & Zygote.DEBUG_ENABLE_JNI_LOGGING) != 0) {
            argsForZygote.add("--enable-jni-logging");
        }
        if ((debugFlags & Zygote.DEBUG_ENABLE_SAFEMODE) != 0) {
            argsForZygote.add("--enable-safemode");
        }
        if ((debugFlags & Zygote.DEBUG_ENABLE_DEBUGGER) != 0) {
            argsForZygote.add("--enable-debugger");
        }
        if ((debugFlags & Zygote.DEBUG_ENABLE_CHECKJNI) != 0) {
            argsForZygote.add("--enable-checkjni");
        }
        if ((debugFlags & Zygote.DEBUG_ENABLE_ASSERT) != 0) {
            argsForZygote.add("--enable-assert");
        }
        if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER) {
            argsForZygote.add("--mount-external-multiuser");
        } else if (mountExternal == Zygote.MOUNT_EXTERNAL_MULTIUSER_ALL) {
            argsForZygote.add("--mount-external-multiuser-all");
        }
        argsForZygote.add("--target-sdk-version=" + targetSdkVersion);
        //TODO optionally enable debuger
        //argsForZygote.add("--enable-debugger");
        // --setgroups is a comma-separated list
        if (gids != null && gids.length > 0) {
            StringBuilder sb = new StringBuilder();
```

```
            sb.append("--setgroups=");
            int sz = gids.length;
            for (int i = 0; i < sz; i++) {
                if (i != 0) {
                    sb.append(',');
                }
                sb.append(gids[i]);
            }
            argsForZygote.add(sb.toString());
        }
        if (niceName != null) {
            argsForZygote.add("--nice-name=" + niceName);
        }
        if (seInfo != null) {
            argsForZygote.add("--seinfo=" + seInfo);
        }
        argsForZygote.add(processClass);
        if (extraArgs != null) {
            for (String arg : extraArgs) {
                argsForZygote.add(arg);
            }
        }
        //请求进程 Zygote 创建应用程序
        return zygoteSendArgsAndGetResult(argsForZygote);
    }
}
```

在上述代码中，通过函数 zygoteSendArgsAndGetResult 调用 Zygote 进程创建了一个指定的应用程序。函数 zygoteSendArgsAndGetResult 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
    private static ProcessStartResult zygoteSendArgsAndGetResult(ArrayList<String>
args)
            throws ZygoteStartFailedEx {
//调用函数 openZygoteSocketIfNeeded 创建一个连接到 Zygote 进程的本地对象 LocalSocket
        openZygoteSocketIfNeeded();
        try {
            /**
             * 将要创建的应用程序进程启动参数列表写入到本地对象 LocalSocket 中
             * Zygote 进程接收到数据之后会创建一个新的应用程序进程
             *将创建的进程 pid 返回给 ActivityManagerService
             */
            sZygoteWriter.write(Integer.toString(args.size()));
            sZygoteWriter.newLine();
            int sz = args.size();
            for (int i = 0; i < sz; i++) {
                String arg = args.get(i);
                if (arg.indexOf('\n') >= 0) {
                    throw new ZygoteStartFailedEx(
                            "embedded newlines not allowed");
                }
                sZygoteWriter.write(arg);
                sZygoteWriter.newLine();
            }
            sZygoteWriter.flush();
            // 在此应该超时
            ProcessStartResult result = new ProcessStartResult();
            result.pid = sZygoteInputStream.readInt();
            if (result.pid < 0) {
                throw new ZygoteStartFailedEx("fork() failed");
            }
            result.usingWrapper = sZygoteInputStream.readBoolean();
            return result;
        } catch (IOException ex) {
            try {
                if (sZygoteSocket != null) {
                    sZygoteSocket.close();
                }
            } catch (IOException ex2) {
                // 失败
```

```
                Log.e(LOG_TAG,"I/O exception on routine close", ex2);
            }
            sZygoteSocket = null;
            throw new ZygoteStartFailedEx(ex);
        }
    }
```

在上述代码中用到了函数 openZygoteSocketIfNeeded，功能是创建一个连接到 Zygote 进程的本地对象 LocalSocket。函数 openZygoteSocketIfNeeded 在文件 frameworks\base\core\java\android\os\Process.java 中定义，具体实现代码如下所示：

```
    private static void openZygoteSocketIfNeeded()
            throws ZygoteStartFailedEx {
        int retryCount;
        if (sPreviousZygoteOpenFailed) {
            retryCount = 0;
        } else {
            retryCount = 10;
        }
        for (int retry = 0
                ; (sZygoteSocket == null) && (retry < (retryCount + 1))
                ; retry++ ) {
            if (retry > 0) {
                try {
                    Log.i("Zygote", "Zygote not up yet, sleeping...");
                    Thread.sleep(ZYGOTE_RETRY_MILLIS);
                } catch (InterruptedException ex) {
                    // should never happen
                }
            }
            try {
//创建一个保存在 sZygoteSocket 中的 LocalSocket 对象
                sZygoteSocket = new LocalSocket();
//将创建的 LocalSocket 对象和名为 ZYGOTE_SOCKET 的 zygote 进程建立连接
                sZygoteSocket.connect(new LocalSocketAddress(ZYGOTE_SOCKET,
                        LocalSocketAddress.Namespace.RESERVED));
//将获得的 LocalSocket 对象 sZygoteSocket 的输入流保存在变量 sZygoteInputStream 中
                sZygoteInputStream
                        = new DataInputStream(sZygoteSocket.getInputStream());
//将获得的 LocalSocket 对象 sZygoteSocket 的输出流保存在变量 sZygoteWriter 中
                sZygoteWriter =
                    new BufferedWriter(
                            new OutputStreamWriter(
                                    sZygoteSocket.getOutputStream()),
                            256);
                Log.i("Zygote", "Process: zygote socket opened");
                sPreviousZygoteOpenFailed = false;
                break;
            } catch (IOException ex) {
                if (sZygoteSocket != null) {
                    try {
                        sZygoteSocket.close();
                    } catch (IOException ex2) {
                        Log.e(LOG_TAG,"I/O exception on close after exception",
                                ex2);
                    }
                }
                sZygoteSocket = null;
            }
        }
        if (sZygoteSocket == null) {
            sPreviousZygoteOpenFailed = true;
            throw new ZygoteStartFailedEx("connect failed");
        }
    }
```

在上述代码中，sZygoteSocket 是一个 LocalSocket 类型的成员变量，能够连接 Zygote 进程中的名为“zygote”的 Socket，这个 Socket 和设备文件/dev/socket/zygote 相对应。

接下来 Zygote 进程会在函数 runSelectLoop 中接收一个创建新应用程序的要求。函数

runSelectLoop 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
private static void runSelectLoop() throws MethodAndArgsCaller {
    ArrayList<FileDescriptor> fds = new ArrayList<FileDescriptor>();
    ArrayList<ZygoteConnection> peers = new ArrayList<ZygoteConnection>();
    FileDescriptor[] fdArray = new FileDescriptor[4];

    fds.add(sServerSocket.getFileDescriptor());
    peers.add(null);

    int loopCount = GC_LOOP_COUNT;
    while (true) {
        int index;
        if (loopCount <= 0) {
            gc();
            loopCount = GC_LOOP_COUNT;
        } else {
            loopCount--;
        }


        try {
            fdArray = fds.toArray(fdArray);
            index = selectReadable(fdArray);
        } catch (IOException ex) {
            throw new RuntimeException("Error in select()", ex);
        }

        if (index < 0) {
            throw new RuntimeException("Error in select()");
        } else if (index == 0) {
            ZygoteConnection newPeer = acceptCommandPeer();
            peers.add(newPeer);
            fds.add(newPeer.getFileDesciptor());
        } else {
            boolean done;
            done = peers.get(index).runOnce();
            if (done) {
                peers.remove(index);
                fds.remove(index);
            }
        }
    }
}
```

在上述代码中，会调用函数 runOnce 处理接收到创建新应用程序的要求。函数 runOnce 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java 中定义，具体实现代码如下所示：

```
boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {

    String args[];
    Arguments parsedArgs = null;
    FileDescriptor[] descriptors;

    try {
        args = readArgumentList();//获得启动要创建应用程序进程的参数
        descriptors = mSocket.getAncillaryFileDescriptors();
    } catch (IOException ex) {
        Log.w(TAG, "IOException on command socket " + ex.getMessage());
        closeSocket();
        return true;
    }

    if (args == null) {
        // EOF reached.
        closeSocket();
        return true;
    }
```

```
        /** 用最新标准是错误的 */
        PrintStream newStderr = null;

        if (descriptors != null && descriptors.length >= 3) {
            newStderr = new PrintStream(
                    new FileOutputStream(descriptors[2]));
        }

        int pid = -1;
        FileDescriptor childPipeFd = null;
        FileDescriptor serverPipeFd = null;

        try {
            parsedArgs = new Arguments(args);

            applyUidSecurityPolicy(parsedArgs, peer, peerSecurityContext);
            applyRlimitSecurityPolicy(parsedArgs, peer, peerSecurityContext);
            applyCapabilitiesSecurityPolicy(parsedArgs, peer, peerSecurityContext);
            applyInvokeWithSecurityPolicy(parsedArgs, peer, peerSecurityContext);
            applyseInfoSecurityPolicy(parsedArgs, peer, peerSecurityContext);

            applyDebuggerSystemProperty(parsedArgs);
            applyInvokeWithSystemProperty(parsedArgs);

            int[][] rlimits = null;

            if (parsedArgs.rlimits != null) {
                rlimits = parsedArgs.rlimits.toArray(intArray2d);
            }

            if (parsedArgs.runtimeInit && parsedArgs.invokeWith != null) {
                FileDescriptor[] pipeFds = Libcore.os.pipe();
                childPipeFd = pipeFds[1];
                serverPipeFd = pipeFds[0];
                ZygoteInit.setCloseOnExec(serverPipeFd, true);
            }
//调用函数 forkAndSpecialize 创建应用程序进程
            pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs. gids,
                    parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal, parsedArgs.
                    seInfo, parsedArgs.niceName);
        } catch (IOException ex) {
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (ErrnoException ex) {
            logAndPrintError(newStderr, "Exception creating pipe", ex);
        } catch (IllegalArgumentException ex) {
            logAndPrintError(newStderr, "Invalid zygote arguments", ex);
        } catch (ZygoteSecurityException ex) {
            logAndPrintError(newStderr,
                    "Zygote security policy prevents request: ", ex);
        }

        try {
            if (pid == 0) {
                // 在子进程
                IoUtils.closeQuietly(serverPipeFd);
                serverPipeFd = null;
                handleChildProc(parsedArgs, descriptors, childPipeFd, newStderr);

                // should never get here, the child is expected to either
                // throw ZygoteInit.MethodAndArgsCaller or exec().
                return true;
            } else {
                // 在父进程
                IoUtils.closeQuietly(childPipeFd);
                childPipeFd = null;
                return handleParentProc(pid, descriptors, serverPipeFd, parsedArgs);
            }
        } finally {
            IoUtils.closeQuietly(childPipeFd);
            IoUtils.closeQuietly(serverPipeFd);
```

```
        }
    }
```

在上述代码中，通过函数 readArgumentList 获得启动要创建应用程序进程的参数，并通过函数 forkAndSpecialize 创建了这个要启动应用程序的进程。其中函数 readArgumentList 在文件 frameworks\base\core\java\com\android\internal\os\ZygoteConnection.java 中定义，具体实现代码如下所示：

```
private String[] readArgumentList()
        throws IOException {

    int argc;

    try {
        String s = mSocketReader.readLine();

        if (s == null) {
            // EOF reached.
            return null;
        }
        argc = Integer.parseInt(s);
    } catch (NumberFormatException ex) {
        Log.e(TAG, "invalid Zygote wire format: non-int at argc");
        throw new IOException("invalid wire format");
    }
    if (argc > MAX_ZYGOTE_ARGC) {
        throw new IOException("max arg count exceeded");
    }

    String[] result = new String[argc];
    for (int i = 0; i < argc; i++) {
        result[i] = mSocketReader.readLine();
        if (result[i] == null) {
            // We got an unexpected EOF.
            throw new IOException("truncated request");
        }
    }

    return result;
}
```

函数 forkAndSpecialize 在文件 libcore\dalvik\src\main\java\dalvik\system\Zygote.java 中定义，具体实现代码如下所示：

```
public static int forkAndSpecialize(int uid, int gid, int[] gids, int debugFlags,
        int[][] rlimits, int mountExternal, String seInfo, String niceName) {
    preFork();
    int pid = nativeForkAndSpecialize(
            uid, gid, gids, debugFlags, rlimits, mountExternal, seInfo, niceName);
    postFork();
    return pid;
}
```

在上述代码中，当创建一个进程的子进程时，如果返回值为 0，则表示在新创建的进程中执行。此时需要调用函数 handleChildProc 来启动这个子进程，并在 handleChildProc 中调用函数 zygoteInit 在新创建的应用程序进程中初始化运行库，这样便可以启动一个 Binder 线程池。

9.2.2　启动线程池

在创建新应用程序完毕之前，需要调用类 RuntimeInit 中的函数 nativeZygoteInit 启动一个新的 Binder 线程池，具体启动流程如下所示。

（1）调用类 RuntimeInit 中的函数 nativeZygoteInit，此函数在文件 frameworks\base\core\java\com\android\internal\os\RuntimeInit.java 中定义，具体实现代码如下所示：

```
public class RuntimeInit {
    private final static String TAG = "AndroidRuntime";
    private final static boolean DEBUG = false;
```

```
    /** true if commonInit() has been called */
    private static boolean initialized;

    private static IBinder mApplicationObject;

    private static volatile boolean mCrashing = false;

    private static final native void nativeZygoteInit();
    private static final native void nativeFinishInit();
```

函数 nativeZygoteInit 是一个 JNI 函数，在文件 frameworks\base\core\jni\AndroidRuntime.cpp 中定义实现，对应代码如下所示：

```
static void com_android_internal_os_RuntimeInit_nativeZygoteInit(JNIEnv* env, jobject
clazz)
{
    gCurRuntime->onZygoteInit();
}
```

在上述实现代码中，gCurRuntime 是一个全局变量，上述代码用到了 gCurRuntime 的成员函数 onZygoteInit 启动了一个 Binder 线程池。函数 onZygoteInit 在文件 frameworks\base\cmds\app_process\app_main.cpp 中定义，具体实现代码如下所示：

```
    virtual void onZygoteInit()
    {
        // Re-enable tracing now that we're no longer in Zygote.
        atrace_set_tracing_enabled(true);

        sp<ProcessState> proc = ProcessState::self();
        ALOGV("App process: starting thread pool.\n");
//调用函数 startThreadPool 启动一个 Binder 线程池
        proc->startThreadPool();
    }
```

在上述代码中，当调用函数 startThreadPool 启动一个 Binder 线程池后，当前应用程序进程就可以通过 Binder 机制和其他进程实现通信。函数 startThreadPool 在文件 frameworks\native\libs\binder\ProcessState.cpp 中定义，具体实现代码如下所示：

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;//默认值为 false
        spawnPooledThread(true);
    }
}
```

在上述代码中，mThreadPoolStarted 的默认值为 false。当第一次调用函数 startThreadPool 时，会在当前进程中启动 Binder 线程池，并将 mThreadPoolStarted 设置为 true，这样做的目的是防止在以后重复启动 Binder 线程池。

9.2.3 创建信息循环

当创建新应用程序进程完毕以后，会调用函数 invokeStaticMain 将类 ActivityThread 的函数 main 设置为新程序的入口函数。当使用函数 main 时，会在当前程序的进程中建立一个信息循环。

接下来首先看函数 invokeStaticMain 的具体实现，此函数在文件 frameworks\base\core\java\com\android\internal\os\RuntimeInit.java 中定义，具体实现代码如下所示：

```
    private static void invokeStaticMain(String className, String[] argv)
            throws ZygoteInit.MethodAndArgsCaller {
        Class<?> cl;

        try {
            cl = Class.forName(className);
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(
```

```
                "Missing class when invoking static main " + className,
                ex);
        }

        Method m;
        try {
//获得静态成员函数 main，并保存在 Method 对象中
            m = cl.getMethod("main", new Class[] { String[].class });
        } catch (NoSuchMethodException ex) {
            throw new RuntimeException(
                    "Missing static main on " + className, ex);
        } catch (SecurityException ex) {
            throw new RuntimeException(
                    "Problem getting static main on " + className, ex);
        }

        int modifiers = m.getModifiers();
        if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))) {
            throw new RuntimeException(
                    "Main method is not public and static on " + className);
        }

        /*
         *将 method 对象封装在静态成员函数 main 中，并保存在一个 Method 对象中
         * 将 MethodAndArgsCaller 对象作为异常抛给当前程序进程来处理
         */
        throw new ZygoteInit.MethodAndArgsCaller(m, argv);
    }
```

静态成员函数 main 在文件 frameworks\base\core\java\com\android\internal\os\RuntimeInit.java 中定义，具体实现代码如下所示：

```
public static final void main(String[] argv) {
    if (argv.length == 2 && argv[1].equals("application")) {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application");
        redirectLogStreams();
    } else {
        if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting tool");
    }

    commonInit();

    nativeFinishInit();

    if (DEBUG) Slog.d(TAG, "Leaving RuntimeInit!");
}
```

在上述代码中，当函数 main 捕获到 MethodAndArgsCaller 异常后，会调用 MethodAndArgsCaller 成员函数 run 进行后面处理。接下来看函数 MethodAndArgsCaller 和 run，这两个函数都是在文件 frameworks\base\core\java\com\android\internal\os\ZygoteInit.java 中定义，具体实现代码如下所示：

```
    public static class MethodAndArgsCaller extends Exception
            implements Runnable {
        private final Method mMethod;

        private final String[] mArgs;

        public MethodAndArgsCaller(Method method, String[] args) {
            mMethod = method;
            mArgs = args;
        }

        public void run() {
            try {
//执行函数 invoke，这样就执行了类 android.app.ActivityThread 中的函数 main
                mMethod.invoke(null, new Object[] { mArgs });
            } catch (IllegalAccessException ex) {
                throw new RuntimeException(ex);
            } catch (InvocationTargetException ex) {
```

```
                Throwable cause = ex.getCause();
                if (cause instanceof RuntimeException) {
                    throw (RuntimeException) cause;
                } else if (cause instanceof Error) {
                    throw (Error) cause;
                }
                throw new RuntimeException(ex);
            }
        }
    }
}
```

在上述代码中，变量 mMethod 和 mArgs 是在构造异常对象时传递进来的，其中变量 mMethod 和类 android.app.ActivityThread 中的函数 main 相对应。

第 10 章　分析 Activity 组件

在 Android 系统中，Activity、Service、Broadcast 和 ContentProvider 是最重要的核心组件。其中 Activity 通常的表现形式是一个单独的界面（Screen）。每个 Activity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面，并响应事件。大多数程序有多个 Activity。例如，一个文本信息程序有这样几个界面：显示联系人列表界面、写信息界面、查看信息界面或者设置界面等。每个界面都是一个 Activity，切换到另一个界面就是载入一个新的 Activity。在本章的内容中，将详细分析 Activity 组件的源代码知识。

10.1 Activity 基础

在 Android 程序中，一个 Activity 可能会给前一个 Activity 返回值，例如，一个让用户选择相片的 Activity 会把选择到的相片返回给其调用者。当打开一个新界面后，前一个界面就被暂停，并放入历史栈中（界面切换历史栈）。使用者可以回溯前面已经打开的存放在历史栈中的界面。也可以从历史栈中删除没有界面价值的界面。Android 在历史栈中保留程序运行产生的所有界面：从第一个界面到最后一个。

10.1.1　Activity 状态

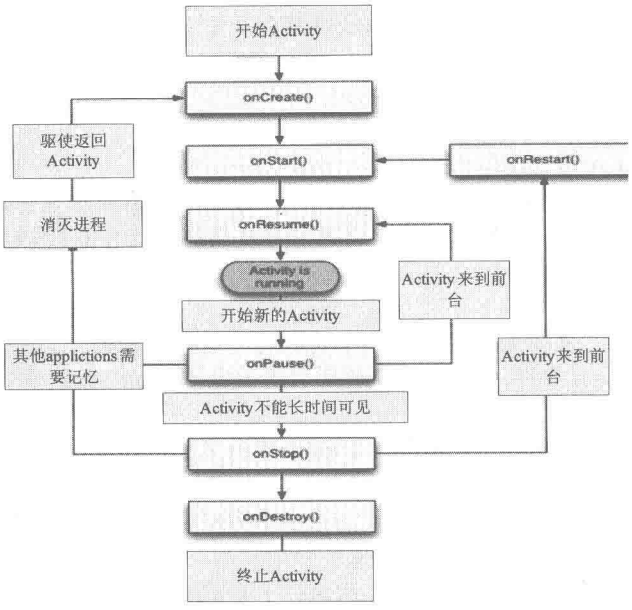
当 Activity 被创建或销毁时，它们进入或退出 Activity 栈。当它们做这些动作时，它们就会在 4 种可能的状态间迁移。

- Active：当 Activity 在栈的顶端时，它是可见的，有焦点的前台 Activity，用来响应用户的输入。Android 会不惜一切代价来尝试保证它的活跃性，需要的话它会消灭栈中更靠下的 Activity，以保证 Active Activity 需要的资源。当另一个 Activity 变成 Active 状态时，这个就会变成 Paused。
- Paused：在一些情况下，应用中的 Activity 可见但不拥有焦点；这时它就是暂停的。当最前面的 Activity 是全透明或非全屏的 Activity 时，下面的 Activity 就会到达这个状态。当暂停时，这个 Activity 还是被看作是 Active 的，但不接收用户的输入事件。在极端的情况下，Android 会杀死一个 paused 的 Activity 来恢复资源给 Active Activity。当一个 Activity 完全不可见时，它就变成 stopped。
- Stopped：当一个 Activity 不可见，它就“停止”了。这个 Activity 仍然留在内存里来保存所有的状态和成员信息；但是，当系统需要内存时，它就被消灭了。当一个 Activity 停止时，保存数据和当前 UI 状态是很重要的。一旦 Activity 退出或关闭，它就变成 Inactive 状态。
- Inactive：当一个曾经被启动过的 Activity 被杀死时，它就变成 Inactive。Inactive Activity 会从 Activity 栈中移除，当它重新显示和使用时需要再次启动。

Activity 状态转换如图 10-1 所示。

状态的变化是人为的，完全由 Android 内存管理器掌握着。Android 会首先关闭那些包含 Inactive Activity 的应用程序，其次关闭那些 stopped 的程序，在极端情况时会移除那些 paused 的程序。

为了保证无瑕疵的用户体验，这些状态的迁移对用户来说必须是不可见的。当 Activity 从 paused、stopped 或者杀死的状态返回到 Active 时，UI 必须是无差别的。所以，当 Activity 暂停或停止时，保存所有的 UI 状态和数据是很重要的。一旦 Activity 变成 Active，它需要从保存的值中恢复。



▲图 10-1 Activity 状态转换图

10.1.2 剖析 Activity 中的主要函数

（1）void onCreate(Bundle savedInstanceState)

当 Activity 被第一次加载时执行。我们新启动一个程序时其主窗体的 onCreate 事件就会被执行。如果 Activity 被销毁后（onDestroy 后），再重新加载进 Task 时，其 onCreate 事件也会被重新执行。注意，这里的参数 savedInstanceState（Bundle 类型是一个键值对集合，大家可以看成是.Net 中的 Dictionary）是一个很有用的设计，由于前面已经说到的手机应用的特殊性，一个 Activity 很可能被强制交换到后台（交换到后台就是指该窗体不再对用户可见，但实际上又还是存在于某个 Task 中的，比如一个新的 Activity 进入了当前的 Task 从而“遮盖”住了当前的 Activity，或者用户按了 Home 键返回到桌面，又或者其他重要事件发生导致新的 Activity 出现在当前 Activity 之上，如来电界面），而如果此后用户在一段时间内没有重新查看该窗体（Android 通过长按 Home 键可以选择最近运行的 6 个程序，或者用户直接再次点击程序的运行图标，如果窗体所在的 Task 和进程没有被系统销毁，则不用重新加载 Process，Task 和 Task 中的 Activity 直接重新显示 Task 顶部的 Activity，这就称为重新查看某个程序的窗体），该窗体连同其所在的 Task 和 Process 则可能已经被系统自动销毁了，此时如果再次查看该窗体，则要重新执行 onCreate 事件初始化窗体。而这个时候我们可能希望用户继续上次打开该窗体时的操作状态进行操作，而不是一切从头开始。例如，用户在编辑短信时突然来电，接完电话后用户又去做了一些其他的事情，比如保存来电号码到联系人，而没有立即返回到短信编辑界面，导致了短信编辑界面被销毁，当用户重新进入短信程序时他可能希望继续上次的编辑。此时就可以覆写 Activity 的 void onSaveInstanceState(Bundle outState)事件，通过向 outState 中写入一些我们需要在窗体销毁前保存的状态或信息，这样在窗体重新执行 onCreate 时，则会通过 savedInstanceState 将之前保存的信息传递进来，此时就可以有选择地利用这些信息来初始化窗体，而不是一切从头开始。

（2）void onStart()

onCreate 事件之后执行。或者当前窗体被交换到后台后，在用户重新查看窗体前已经过去了一段时间，窗体已经执行了 onStop 事件，但是窗体和其所在进程并没有被销毁，用户再次重新查看窗体时会执行 onRestart 事件，之后会跳过 onCreate 事件，直接执行窗体的 onStart 事件。

（3）void onResume()

onStart 事件之后执行。或者当前窗体被交换到后台后，在用户重新查看窗体时，窗体还没有

被销毁，也没有执行过 onStop 事件（窗体还继续存在于 Task 中），则会跳过窗体的 onCreate 和 onStart 事件，直接执行 onResume 事件。

（4）void onPause()

窗体被交换到后台时执行。

（5）void onStop()

onPause 事件之后执行。如果一段时间内用户还没有重新查看该窗体，则该窗体的 onStop 事件将会被执行；或者用户直接按了 Back 键，将该窗体从当前 Task 中移除，也会执行该窗体的 onStop 事件。

（6）void onRestart()

onStop 事件执行后，如果窗体和其所在的进程没有被系统销毁，此时用户又重新查看该窗体，则会执行窗体的 onRestart 事件，onRestart 事件后会跳过窗体的 onCreate 事件直接执行 onStart 事件。

（7）void onDestroy()

Activity 被销毁的时候执行。在窗体的 onStop 事件之后，如果没有再次查看该窗体，Activity 则会被销毁。

最后用一个实际的例子来说明 Activity 的各个生命周期。假设有一个程序由两个 Activity，分别是 A 和 B 组成，A 是这个程序的启动界面。当用户启动程序时，Process 和默认的 Task 分别被创建，接着 A 被压入到当前的 Task 中，依次执行了 onCreate、onStart、onResume 事件被呈现给了用户；此时用户选择 A 中的某个功能开启界面 B，界面 B 被压入当前 Task 遮盖住了 A，A 的 onPause 事件执行，B 的 onCreate、onStart、onResume 事件执行，呈现了界面 B 给用户；用户在界面 B 操作完成后，使用 Back 键返回到界面 A，界面 B 不再可见，界面 B 的 onPause、onStop、onDestroy 执行，A 的 onResume 事件被执行，呈现界面 A 给用户。此时突然来电，界面 A 的 onPause 事件被执行，电话接听界面被呈现给用户，用户接听完电话后，又按了 Home 键返回到桌面，打开另一个程序“联系人”，添加了联系人信息又做了一些其他的操作，此时界面 A 不再可见，其 onStop 事件被执行，但并没有被销毁。此后用户重新从菜单中点击了我们的程序，由于 A 和其所在的进程和 Task 并没有被销毁，A 的 onRestart 和 onStart 事件被执行，接着 A 的 onResume 事件被执行，A 又被呈现给了用户。用户这次使用完后，按 Back 键返回到桌面，A 的 onPause, onStop 被执行，随后 A 的 onDestroy 被执行，由于当前 Task 中已经没有任何 Activity，A 所在的 Process 的重要程度被降到很低，很快 A 所在的 Process 被系统结束。

10.2 分析 Activity 的启动源代码

在 Android 系统的应用程序框架层中，ActivityManagerService 负责启动 Activity。在整个启动过程中，ActivityManagerService 用于管理 Activity 的生命周期，ActivityManagerService 可以通过 ActivityStack 将所有的 Activity 按照后进先出的顺序放在一个堆栈中。对于每一个应用程序来说，都拥有一个专门的 ActivityThread 来表示应用程序的主进程。在每个 ActivityThread 中，都包含了一个 Binder 对象类型的 ApplicationThread 实例，其功能是和其他进程实现通信。

具体来说，在 Android 系统中启动 Activity 的流程如下所示。

（1）通过 Binder 进程间通信进入到 ActivityManagerService 进程中，并且调用 ActivityManagerService.startActivity 接口。

（2）ActivityManagerService 调用 ActivityStack.startActivityMayWait 做好启动 Activity 前的准备。

（3）ActivityStack 通知 ApplicationThread 即将启动 Activity，ApplicationThread 表示调用 ActivityManagerService.startActivity 接口的进程。

（4）ApplicationThread 并不执行真正的启动操作，它通过调用 ActivityManagerService.activityPaused 接口进入到 ActivityManagerService 进程中，看看是否需要创建新的进程来启动 Activity。

（5）对于通过点击应用程序图标来启动 Activity 的情形来说，ActivityManagerService 会在本

步调用 startProcessLocked 创建一个新进程。而对于通过在 Activity 内部调用 startActivity 来启动新的 Activity 来说并不需要执行这一步，因为新的 Activity 就在原来的 Activity 所在的进程中进行启动。

（6）ActivityManagerServic 调用 ApplicationThread.scheduleLaunchActivity 接口，通知相应的进程执行启动 Activity 的操作。

（7）ApplicationThread 把这个启动 Activity 的操作转发给 ActivityThread，ActivityThread 通过 ClassLoader 导入相应的 Activity 类，然后把它启动起来。

在本节的内容中，将以 Activity 组件中的 MainActivity 为例，讲解启动 MainActivity 的具体流程。

10.2.1 Launcher 启动应用程序

在 Android 系统中，Launcher 负责启动应用程序。当在 Android 中安装应用程序后，会在 Launcher 界面出现一个相应的图标，点击这个图标后 Launcher 会启动这个图标对应的应用程序。其实 Launcher 也是一个应用程序，在 Android 的源代码中，Launcher 应用程序的源代码被保存在“\packages\apps\Launcher2”目录中。在文件 packages\apps\Launcher2\src\com\android\launcher2\Launcher.java 中，用于启动 Android 应用程序的实现源代码如下所示：

```
public void onClick(View v) {
    // Make sure that rogue clicks don't get through while allapps is launching, or after
the
    // view has detached (it's possible for this to happen if the view is removed mid touch).
    if (v.getWindowToken() == null) {
        return;
    }

    if (!mWorkspace.isFinishedSwitchingState()) {
        return;
    }

    Object tag = v.getTag();
    if (tag instanceof ShortcutInfo) {
        // Open shortcut
        final Intent intent = ((ShortcutInfo) tag).intent;
        int[] pos = new int[2];
        v.getLocationOnScreen(pos);
        intent.setSourceBounds(new Rect(pos[0], pos[1],
                pos[0] + v.getWidth(), pos[1] + v.getHeight()));

        boolean success = startActivitySafely(v, intent, tag);

        if (success && v instanceof BubbleTextView) {
            mWaitingForResume = (BubbleTextView) v;
            mWaitingForResume.setStayPressed(true);
        }
    } else if (tag instanceof FolderInfo) {
        if (v instanceof FolderIcon) {
            FolderIcon fi = (FolderIcon) v;
            handleFolderClick(fi);
        }
    } else if (v == mAllAppsButton) {
        if (isAllAppsVisible()) {
            showWorkspace(true);
        } else {
            onClickAllAppsButton(v);
        }
    }
}
boolean startActivitySafely(View v, Intent intent, Object tag) {
    boolean success = false;
    try {
        success = startActivity(v, intent, tag);
    } catch (ActivityNotFoundException e) {
        Toast.makeText(this, R.string.activity_not_found, Toast.LENGTH_SHORT).show();
```

```
            Log.e(TAG, "Unable to launch. tag=" + tag + " intent=" + intent, e);
        }
        return success;
    }
```

另外，在类 Activity 中定义了函数 startActivity 的具体实现，此函数的功能是调用函数 startActivityForResult 进一步实现启动应用程序处理。函数 startActivity 在文件 frameworks/base/core/java/android/app/Activity.java 中定义，具体实现代码如下所示：

```
public void startActivity(Intent intent) {
    startActivity(intent, null);
}
```

在上述代码中用到了函数 startActivity，其第二个参数传入 null 表示不需要这个 Actvity 结束后的返回结果。函数 startActivity 在文件 frameworks/base/core/java/android/app/Activity.java 中定义，具体实现代码如下所示：

```
public void startActivityForResult(Intent intent, int requestCode, Bundle options) {
    if (mParent == null) {
        Instrumentation.ActivityResult ar =
            mInstrumentation.execStartActivity(
                this, mMainThread.getApplicationThread(), mToken, this,
                intent, requestCode, options);
        if (ar != null) {
            mMainThread.sendActivityResult(
                mToken, mEmbeddedID, requestCode, ar.getResultCode(),
                ar.getResultData());
        }
        if (requestCode >= 0) {
            mStartedActivity = true;
        }
    } else {
        if (options != null) {
            mParent.startActivityFromChild(this, intent, requestCode, options);
        } else {
            mParent.startActivityFromChild(this, intent, requestCode);
        }
    }
}
```

在上述代码中，mInstrumentation 是类 Activity 中类型为 Intrumentation 的成员，在文件 frameworks/base/core/java/android/app/Instrumentation.java 中定义，功能是监控应用程序和系统的交互。

10.2.2　返回 ActivityManagerService 的远程接口

函数 execStartActivity 的功能是返回 ActivityManagerService 的远程接口，即 ActivityManager Proxy 接口。函数 execStartActivity 在文件 frameworks/base/core/java/android/app/Instrumentation.java 中定义，具体实现代码如下所示：

```
public ActivityResult execStartActivity(
        Context who, IBinder contextThread, IBinder token, Activity target,
        Intent intent, int requestCode, Bundle options) {
    IApplicationThread whoThread = (IApplicationThread) contextThread;
    if (mActivityMonitors != null) {
        synchronized (mSync) {
            final int N = mActivityMonitors.size();
            for (int i=0; i<N; i++) {
                final ActivityMonitor am = mActivityMonitors.get(i);
                if (am.match(who, null, intent)) {
                    am.mHits++;
                    if (am.isBlocking()) {
                        return requestCode >= 0 ? am.getResult() : null;
                    }
                    break;
                }
            }
        }
    }
```

```
        try {
            intent.migrateExtraStreamToClipData();
            intent.prepareToLeaveProcess();
            int result = ActivityManagerNative.getDefault()
                .startActivity(whoThread, who.getBasePackageName(), intent,
                        intent.resolveTypeIfNeeded(who.getContentResolver()),
                        token, target != null ? target.mEmbeddedID : null,
                        requestCode, 0, null, null, options);
            checkStartActivityResult(result, intent);
        } catch (RemoteException e) {
        }
        return null;
    }
```

再看类 ActivityManagerService 中的函数 startActivity，功能是将我们的操作转发给成员变量 mMainStack 的 startActivityMayWait 函数，此处 mMainStack 的类型为 ActivityStack。函数 startActivity 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
    public final int startActivity(IApplicationThread caller, String callingPackage,
            Intent intent, String resolvedType, IBinder resultTo,
            String resultWho, int requestCode, int startFlags,
            String profileFile, ParcelFileDescriptor profileFd, Bundle options) {
        return startActivityAsUser(caller, callingPackage, intent, resolvedType, resultTo,
                resultWho, requestCode,
                startFlags, profileFile, profileFd, options, UserHandle.getCallingUserId());
    }
```

10.2.3 解析 intent 的内容

如果前面步骤中的参数 outResult 和 config 都是 null，并且如下表达式的结果为 false：

```
(aInfo.applicationInfo.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0
```

则调用函数 startActivityMayWait 解析参数 intent 的内容，并将得到的 MainActivity 信息保存在变量 aInfo 中。函数 startActivityMayWait 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
    final int startActivityMayWait(IApplicationThread caller, int callingUid,
            String callingPackage, Intent intent, String resolvedType, IBinder resultTo,
            String resultWho, int requestCode, int startFlags, String profileFile,
            ParcelFileDescriptor profileFd, WaitResult outResult, Configuration config,
            Bundle options, int userId) {
        // Refuse possible leaked file descriptors
        if (intent != null && intent.hasFileDescriptors()) {
            throw new IllegalArgumentException("File descriptors passed in Intent");
        }
        boolean componentSpecified = intent.getComponent() != null;

        // Don't modify the client's object!
        intent = new Intent(intent);

        // Collect information about the target of the Intent.
        ActivityInfo aInfo = resolveActivity(intent, resolvedType, startFlags,
                profileFile, profileFd, userId);

        synchronized (mService) {
            int callingPid;
            if (callingUid >= 0) {
                callingPid = -1;
            } else if (caller == null) {
                callingPid = Binder.getCallingPid();
                callingUid = Binder.getCallingUid();
            } else {
                callingPid = callingUid = -1;
            }

            mConfigWillChange = config != null
                    && mService.mConfiguration.diff(config) != 0;
```

```
if (DEBUG_CONFIGURATION) Slog.v(TAG,
        "Starting activity when config will change = " + mConfigWillChange);

final long origId = Binder.clearCallingIdentity();

if (mMainStack && aInfo != null &&
        (aInfo.applicationInfo.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0) {
    // This may be a heavy-weight process!  Check to see if we already
    // have another, different heavy-weight process running.
    if (aInfo.processName.equals(aInfo.applicationInfo.packageName)) {
        if (mService.mHeavyWeightProcess != null &&
                (mService.mHeavyWeightProcess.info.uid != aInfo.applicationInfo.
                uid ||
                !mService.mHeavyWeightProcess.processName.equals(aInfo.processNa
                me))) {
            int realCallingPid = callingPid;
            int realCallingUid = callingUid;
            if (caller != null) {
                ProcessRecord callerApp = mService.getRecordForAppLocked(caller);
                if (callerApp != null) {
                    realCallingPid = callerApp.pid;
                    realCallingUid = callerApp.info.uid;
                } else {
                    Slog.w(TAG, "Unable to find app for caller " + caller
                          + " (pid=" + realCallingPid + ") when starting: "
                          + intent.toString());
                    ActivityOptions.abort(options);
                    return ActivityManager.START_PERMISSION_DENIED;
                }
            }

            IIntentSender target = mService.getIntentSenderLocked(
                    ActivityManager.INTENT_SENDER_ACTIVITY, "android",
                    realCallingUid, userId, null, null, 0, new Intent[] { intent },
                    new String[] { resolvedType }, PendingIntent.FLAG_CANCEL_CURRENT
                    | PendingIntent.FLAG_ONE_SHOT, null);

            Intent newIntent = new Intent();
            if (requestCode >= 0) {
                // Caller is requesting a result.
                newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_HAS_RESULT,true);
            }
            newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_INTENT,
                    new IntentSender(target));
            if (mService.mHeavyWeightProcess.activities.size() > 0) {
                ActivityRecord hist = mService.mHeavyWeightProcess.activities. get(0);
                newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_CUR_APP,
                        hist.packageName);
                newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_CUR_TASK,
                        hist.task.taskId);
            }
            newIntent.putExtra(HeavyWeightSwitcherActivity.KEY_NEW_APP,
                    aInfo.packageName);
            newIntent.setFlags(intent.getFlags());
            newIntent.setClassName("android",
                    HeavyWeightSwitcherActivity.class.getName());
            intent = newIntent;
            resolvedType = null;
            caller = null;
            callingUid = Binder.getCallingUid();
            callingPid = Binder.getCallingPid();
            componentSpecified = true;
            try {
                ResolveInfo rInfo =
                    AppGlobals.getPackageManager().resolveIntent(
                            intent, null,
                            PackageManager.MATCH_DEFAULT_ONLY
                            | ActivityManagerService.STOCK_PM_FLAGS, userId);
                aInfo = rInfo != null ? rInfo.activityInfo : null;
                aInfo = mService.getActivityInfoForUser(aInfo, userId);
            } catch (RemoteException e) {
```

```
                    aInfo = null;
                }
            }
        }
    }

    int res = startActivityLocked(caller, intent, resolvedType,
            aInfo, resultTo, resultWho, requestCode, callingPid, callingUid,
            callingPackage, startFlags, options, componentSpecified, null);

    if (mConfigWillChange && mMainStack) {
        mService.enforceCallingPermission(android.Manifest.permission.CHANGE_
        CONFIGURATION, "updateConfiguration()");
        mConfigWillChange = false;
        if (DEBUG_CONFIGURATION) Slog.v(TAG,
                "Updating to new configuration after starting activity.");
        mService.updateConfigurationLocked(config, null, false, false);
    }

    Binder.restoreCallingIdentity(origId);

    if (outResult != null) {
        outResult.result = res;
        if (res == ActivityManager.START_SUCCESS) {
            mWaitingActivityLaunched.add(outResult);
            do {
                try {
                    mService.wait();
                } catch (InterruptedException e) {
                }
            } while (!outResult.timeout && outResult.who == null);
        } else if (res == ActivityManager.START_TASK_TO_FRONT) {
            ActivityRecord r = this.topRunningActivityLocked(null);
            if (r.nowVisible) {
                outResult.timeout = false;
                outResult.who = new ComponentName(r.info.packageName, r.info.name);
                outResult.totalTime = 0;
                outResult.thisTime = 0;
            } else {
                outResult.thisTime = SystemClock.uptimeMillis();
                mWaitingActivityVisible.add(outResult);
                do {
                    try {
                        mService.wait();
                    } catch (InterruptedException e) {
                    }
                } while (!outResult.timeout && outResult.who == null);
            }
        }
    }

    return res;
  }
}
```

10.2.4 分析检查机制

再看函数 startActivityLocked，后缀 Locked 表示这个函数是线程安全的，即该函数体只能由一个线程调用。如果另一个线程也需要调用该函数，必须等待前一个线程执行完毕。函数 startActivityLocked 的主要功能如下所示。

- 检查当前正在运行的 Activity 是否和目标 Activity 一致，如果一致则直接返回。也就是说，程序员在 Activity 使用 startActivity 启动自己，不会达到重启当前 Activity 的目的。
- 处理 INTENT_FLAG_FORWARD_RESULT 标志，在代码中此标志能够跨 Activity 传递 Result。
- 检查是否存在 Caller 的 App，此功能出现在发出 startActivity 命令后，Caller 所在的进程被意外杀死，此时 AmS 拒绝继续往下执行。
- 检查 Caller 是否具备启动指定 Activity 的权限。

函数 startActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final int startActivityLocked(IApplicationThread caller,
        Intent intent, String resolvedType, ActivityInfo aInfo, IBinder resultTo,
        String resultWho, int requestCode,
        int callingPid, int callingUid, String callingPackage, int startFlags, Bundle
options,
        boolean componentSpecified, ActivityRecord[] outActivity) {

    int err = ActivityManager.START_SUCCESS;

    ProcessRecord callerApp = null;

    if (caller != null) {
        callerApp = mService.getRecordForAppLocked(caller);
        if (callerApp != null) {
            callingPid = callerApp.pid;
            callingUid = callerApp.info.uid;
        } else {
            Slog.w(TAG, "Unable to find app for caller " + caller
                  + " (pid=" + callingPid + ") when starting: "
                  + intent.toString());
            err = ActivityManager.START_PERMISSION_DENIED;
        }
    }

    if (err == ActivityManager.START_SUCCESS) {
        final int userId = aInfo != null ? UserHandle.getUserId(aInfo.applicationInfo.uid) : 0;
        Slog.i(TAG, "START u" + userId + " {" + intent.toShortString(true, true, true,
false)
                + "} from pid " + (callerApp != null ? callerApp.pid : callingPid));
    }

    ActivityRecord sourceRecord = null;
    ActivityRecord resultRecord = null;
    if (resultTo != null) {
        int index = indexOfTokenLocked(resultTo);
        if (DEBUG_RESULTS) Slog.v(
            TAG, "Will send result to " + resultTo + " (index " + index + ")");
        if (index >= 0) {
            sourceRecord = mHistory.get(index);
            if (requestCode >= 0 && !sourceRecord.finishing) {
                resultRecord = sourceRecord;
            }
        }
    }

    int launchFlags = intent.getFlags();

    if ((launchFlags&Intent.FLAG_ACTIVITY_FORWARD_RESULT) != 0
            && sourceRecord != null) {
        // Transfer the result target from the source activity to the new
        // one being started, including any failures.
        if (requestCode >= 0) {
            ActivityOptions.abort(options);
            return ActivityManager.START_FORWARD_AND_REQUEST_CONFLICT;
        }
        resultRecord = sourceRecord.resultTo;
        resultWho = sourceRecord.resultWho;
        requestCode = sourceRecord.requestCode;
        sourceRecord.resultTo = null;
        if (resultRecord != null) {
            resultRecord.removeResultsLocked(
                sourceRecord, resultWho, requestCode);
        }
    }

    if (err == ActivityManager.START_SUCCESS && intent.getComponent() == null) {
        // We couldn't find a class that can handle the given Intent.
```

```
        // That's the end of that!
        err = ActivityManager.START_INTENT_NOT_RESOLVED;
    }

    if (err == ActivityManager.START_SUCCESS && aInfo == null) {
        // We couldn't find the specific class specified in the Intent.
        // Also the end of the line.
        err = ActivityManager.START_CLASS_NOT_FOUND;
    }

    if (err != ActivityManager.START_SUCCESS) {
        if (resultRecord != null) {
            sendActivityResultLocked(-1,
                resultRecord, resultWho, requestCode,
                Activity.RESULT_CANCELED, null);
        }
        mDismissKeyguardOnNextActivity = false;
        ActivityOptions.abort(options);
        return err;
    }

    final int startAnyPerm = mService.checkPermission(
            START_ANY_ACTIVITY, callingPid, callingUid);
    final int componentPerm = mService.checkComponentPermission(aInfo.permission,
callingPid, callingUid, aInfo.applicationInfo.uid, aInfo.exported);
    if (startAnyPerm != PERMISSION_GRANTED && componentPerm != PERMISSION_GRANTED) {
        if (resultRecord != null) {
            sendActivityResultLocked(-1,
                resultRecord, resultWho, requestCode,
                Activity.RESULT_CANCELED, null);
        }
        mDismissKeyguardOnNextActivity = false;
        String msg;
        if (!aInfo.exported) {
            msg = "Permission Denial: starting " + intent.toString()
                    + " from " + callerApp + " (pid=" + callingPid
                    + ", uid=" + callingUid + ")"
                    + " not exported from uid " + aInfo.applicationInfo.uid;
        } else {
            msg = "Permission Denial: starting " + intent.toString()
                    + " from " + callerApp + " (pid=" + callingPid
                    + ", uid=" + callingUid + ")"
                    + " requires " + aInfo.permission;
        }
        Slog.w(TAG, msg);
        throw new SecurityException(msg);
    }

    boolean abort = !mService.mIntentFirewall.checkStartActivity(intent,
            callerApp==null?null:callerApp.info, callingUid, callingPid, resolvedType,
            aInfo);

    if (mMainStack) {
        if (mService.mController != null) {
            try {
                // The Intent we give to the watcher has the extra data
                // stripped off, since it can contain private information.
                Intent watchIntent = intent.cloneFilter();
                abort |= !mService.mController.activityStarting(watchIntent,
                        aInfo.applicationInfo.packageName);
            } catch (RemoteException e) {
                mService.mController = null;
            }
        }
    }

    if (abort) {
        if (resultRecord != null) {
            sendActivityResultLocked(-1,
                resultRecord, resultWho, requestCode,
                Activity.RESULT_CANCELED, null);
```

```
            }
            // We pretend to the caller that it was really started, but
            // they will just get a cancel result
            mDismissKeyguardOnNextActivity = false;
            ActivityOptions.abort(options);
            return ActivityManager.START_SUCCESS;
        }
    //创建即将要启动的 Activity 的相关信息，并保存在 r 变量中
        ActivityRecord r = new ActivityRecord(mService, this, callerApp, callingUid,
    callingPackage, intent, resolvedType, aInfo, mService.mConfiguration,
                resultRecord, resultWho, requestCode, componentSpecified);
        if (outActivity != null) {
            outActivity[0] = r;
        }

        if (mMainStack) {
            if (mResumedActivity == null
                    || mResumedActivity.info.applicationInfo.uid != callingUid) {
                if (!mService.checkAppSwitchAllowedLocked(callingPid, callingUid, "Activity
                start")) {
                    PendingActivityLaunch pal = new PendingActivityLaunch();
                    pal.r = r;
                    pal.sourceRecord = sourceRecord;
                    pal.startFlags = startFlags;
                    mService.mPendingActivityLaunches.add(pal);
                    mDismissKeyguardOnNextActivity = false;
                    ActivityOptions.abort(options);
                    return ActivityManager.START_SWITCHES_CANCELED;
                }
            }

            if (mService.mDidAppSwitch) {
                mService.mAppSwitchesAllowedTime = 0;
            } else {
                mService.mDidAppSwitch = true;
            }

            mService.doPendingActivityLaunchesLocked(false);
        }

        err = startActivityUncheckedLocked(r, sourceRecord,
                startFlags, true, options);
        if (mDismissKeyguardOnNextActivity && mPausingActivity == null) {
            mDismissKeyguardOnNextActivity = false;
            mService.mWindowManager.dismissKeyguard();
        }
        return err;
    }
```

如果等待序列为空，则调用函数 startActivityUncheckedLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，此时要启动的 Activity 已经通过检查机制的检验，并被“诊断”为是一个“健康合法”的启动请求。函数 startActivityUncheckedLocked 的具体实现代码如下所示：

```
    final int startActivityUncheckedLocked(ActivityRecord r,
            ActivityRecord sourceRecord, int startFlags, boolean doResume,
            Bundle options) {
        final Intent intent = r.intent;
        final int callingUid = r.launchedFromUid;

        int launchFlags = intent.getFlags();
        mUserLeaving = (launchFlags&Intent.FLAG_ACTIVITY_NO_USER_ACTION) == 0;
        if (DEBUG_USER_LEAVING) Slog.v(TAG,
                "startActivity() => mUserLeaving=" + mUserLeaving);
        if (!doResume) {
            r.delayedResume = true;
        }

        ActivityRecord notTop = (launchFlags&Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP)
                != 0 ? r : null;
```

```
        if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED) != 0) {
            ActivityRecord checkedCaller = sourceRecord;
            if (checkedCaller == null) {
                checkedCaller = topRunningNonDelayedActivityLocked(notTop);
            }
            if (!checkedCaller.realActivity.equals(r.realActivity)) {
                // Caller is not the same as launcher, so always needed.
                startFlags &= ~ActivityManager.START_FLAG_ONLY_IF_NEEDED;
            }
        }

        if (sourceRecord == null) {
            if ((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
                Slog.w(TAG, "startActivity called from non-Activity context; forcing
                Intent.FLAG_ACTIVITY_NEW_TASK for: "
                      + intent);
                launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
            }
        } else if (sourceRecord.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
            launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
        } else if (r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
            launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
        }

        if (r.resultTo != null && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
            Slog.w(TAG, "Activity is launching as a new task, so cancelling activity
            result.");
            sendActivityResultLocked(-1,
                  r.resultTo, r.resultWho, r.requestCode,
                Activity.RESULT_CANCELED, null);
            r.resultTo = null;
        }

        boolean addingToTask = false;
        boolean movedHome = false;
        TaskRecord reuseTask = null;
//如果此 intent 的标志值的位 Intent.FLAG_ACTIVITY_NEW_TASK 被置位,
//而且 Intent.FLAG_ACTIVITY_MULTIPLE_TASK 没有置位，则执行下面的 if 语句
//下面代码的功能是查看当前有没有 Task 可以用来执行这个 Activity
//因为 r.launchMode 的值不为 ActivityInfo.LAUNCH_SINGLE_INSTANCE,
//所以它通过函数 findTaskLocked 来查找是否存在这样的 Task,
//此处返回的结果是 null，即 taskTop 为 null,
//所以需要创建一个新的 Task 来启动这个 Activity
        if (((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0 &&
                (launchFlags&Intent.FLAG_ACTIVITY_MULTIPLE_TASK) == 0)
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK
                || r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
            if (r.resultTo == null) {
                ActivityRecord taskTop = r.launchMode != ActivityInfo.LAUNCH_SINGLE_ INSTANCE
                      ? findTaskLocked(intent, r.info)
                      : findActivityLocked(intent, r.info);
                if (taskTop != null) {
                    if (taskTop.task.intent == null) {
                        taskTop.task.setIntent(intent, r.info);
                    }
                    ActivityRecord curTop = topRunningNonDelayedActivityLocked(notTop);
                    if (curTop != null && curTop.task != taskTop.task) {
                        r.intent.addFlags(Intent.FLAG_ACTIVITY_BROUGHT_TO_FRONT);
                        boolean callerAtFront = sourceRecord == null
                              || curTop.task == sourceRecord.task;
                        if (callerAtFront) {
                            // We really do want to push this one into the
                            // user's face, right now.
                            movedHome = true;
                            moveHomeToFrontFromLaunchLocked(launchFlags);
                            moveTaskToFrontLocked(taskTop.task, r, options);
                            options = null;
                        }
                    }
                    if ((launchFlags&Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) != 0) {
```

```
                    taskTop = resetTaskIfNeededLocked(taskTop, r);
                }
                if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED) != 0) {
                    if (doResume) {
                        resumeTopActivityLocked(null, options);
                    } else {
                        ActivityOptions.abort(options);
                    }
                    return ActivityManager.START_RETURN_INTENT_TO_CALLER;
                }
                if ((launchFlags &
                        (Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_
                        TASK))
                        == (Intent.FLAG_ACTIVITY_NEW_TASK|Intent.FLAG_ACTIVITY_CLEAR_
                        TASK)) {
                    reuseTask = taskTop.task;
                    performClearTaskLocked(taskTop.task.taskId);
                    reuseTask.setIntent(r.intent, r.info);
                } else if ((launchFlags&Intent.FLAG_ACTIVITY_CLEAR_TOP) != 0
                        || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK
                        || r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
                    ActivityRecord top = performClearTaskLocked(
                            taskTop.task.taskId, r, launchFlags);
                    if (top != null) {
                        if (top.frontOfTask) {
                            // Activity aliases may mean we use different
                            // intents for the top activity, so make sure
                            // the task now has the identity of the new
                            // intent.
                            top.task.setIntent(r.intent, r.info);
                        }
                        logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
                        top.deliverNewIntentLocked(callingUid, r.intent);
                    } else {
                        addingToTask = true;
                        sourceRecord = taskTop;
                    }
                } else if (r.realActivity.equals(taskTop.task.realActivity)) {
                    if (((launchFlags&Intent.FLAG_ACTIVITY_SINGLE_TOP) != 0
                            || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP)
                            && taskTop.realActivity.equals(r.realActivity)) {
                        logStartActivity(EventLogTags.AM_NEW_INTENT, r, taskTop.task);
                        if (taskTop.frontOfTask) {
                            taskTop.task.setIntent(r.intent, r.info);
                        }
                        taskTop.deliverNewIntentLocked(callingUid, r.intent);
                    } else if (!r.intent.filterEquals(taskTop.task.intent)) {
                        addingToTask = true;
                        sourceRecord = taskTop;
                    }
                } else if ((launchFlags&Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) == 0) {
                    addingToTask = true;
                    sourceRecord = taskTop;
                } else if (!taskTop.task.rootWasReset) {
                    taskTop.task.setIntent(r.intent, r.info);
                }
                if (!addingToTask && reuseTask == null) {
                    if (doResume) {
                        resumeTopActivityLocked(null, options);
                    } else {
                        ActivityOptions.abort(options);
                    }
                    return ActivityManager.START_TASK_TO_FRONT;
                }
            }
        }
    }
    if (r.packageName != null) {
        // 查看当前在堆栈顶端的 Activity 是否就是即将要启动的 Activity，
        // 在有些情况下，如果即将要启动的 Activity 就在堆栈的顶端，
        // 那么就不会重新启动这个 Activity 的另外一个实例
```

```
    ActivityRecord top = topRunningNonDelayedActivityLocked(notTop);
    if (top != null && r.resultTo == null) {
        if (top.realActivity.equals(r.realActivity) && top.userId == r.userId) {
            if (top.app != null && top.app.thread != null) {
                if ((launchFlags&Intent.FLAG_ACTIVITY_SINGLE_TOP) != 0
                    || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP
                    || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
                    logStartActivity(EventLogTags.AM_NEW_INTENT, top, top.task);
                    // For paranoia, make sure we have correctly
                    // resumed the top activity.
                    if (doResume) {
                        resumeTopActivityLocked(null);
                    }
                    ActivityOptions.abort(options);
                    if ((startFlags&ActivityManager.START_FLAG_ONLY_IF_NEEDED) != 0) {
                        // We don't need to start a new activity, and
                        // the client said not to do anything if that
                        // is the case, so this is it!
                        return ActivityManager.START_RETURN_INTENT_TO_CALLER;
                    }
                    top.deliverNewIntentLocked(callingUid, r.intent);
                    return ActivityManager.START_DELIVERED_TO_TOP;
                }
            }
        }
    }

} else {
    if (r.resultTo != null) {
        sendActivityResultLocked(-1,
                r.resultTo, r.resultWho, r.requestCode,
            Activity.RESULT_CANCELED, null);
    }
    ActivityOptions.abort(options);
    return ActivityManager.START_CLASS_NOT_FOUND;
}

boolean newTask = false;
boolean keepCurTransition = false;

// 因为要在一个新的 Task 里面来启动这个 Activity，所以下面新创建一个 Task
if (r.resultTo == null && !addingToTask
        && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
    if (reuseTask == null) {
        // todo: should do better management of integers.
        mService.mCurTask++;
        if (mService.mCurTask <= 0) {
            mService.mCurTask = 1;
        }
        r.setTask(new TaskRecord(mService.mCurTask, r.info, intent), null, true);
        if (DEBUG_TASKS) Slog.v(TAG, "Starting new activity " + r
                + " in new task " + r.task);
    } else {
        r.setTask(reuseTask, reuseTask, true);
    }
    newTask = true;
    if (!movedHome) {
        moveHomeToFrontFromLaunchLocked(launchFlags);
    }

} else if (sourceRecord != null) {
    if (!addingToTask &&
            (launchFlags&Intent.FLAG_ACTIVITY_CLEAR_TOP) != 0) {
        // In this case, we are adding the activity to an existing
        // task, but the caller has asked to clear that task if the
        // activity is already running.
        ActivityRecord top = performClearTaskLocked(
                sourceRecord.task.taskId, r, launchFlags);
        keepCurTransition = true;
        if (top != null) {
            logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
```

```
                top.deliverNewIntentLocked(callingUid, r.intent);
                if (doResume) {
                    resumeTopActivityLocked(null);
                }
                ActivityOptions.abort(options);
                return ActivityManager.START_DELIVERED_TO_TOP;
            }
        } else if (!addingToTask &&
                (launchFlags&Intent.FLAG_ACTIVITY_REORDER_TO_FRONT) != 0) {
            int where = findActivityInHistoryLocked(r, sourceRecord.task.taskId);
            if (where >= 0) {
                ActivityRecord top = moveActivityToFrontLocked(where);
                logStartActivity(EventLogTags.AM_NEW_INTENT, r, top.task);
                top.updateOptionsLocked(options);
                top.deliverNewIntentLocked(callingUid, r.intent);
                if (doResume) {
                    resumeTopActivityLocked(null);
                }
                return ActivityManager.START_DELIVERED_TO_TOP;
            }
        }
        r.setTask(sourceRecord.task, sourceRecord.thumbHolder, false);
        if (DEBUG_TASKS) Slog.v(TAG, "Starting new activity " + r
                + " in existing task " + r.task);

    } else {
        final int N = mHistory.size();
        ActivityRecord prev =
            N > 0 ? mHistory.get(N-1) : null;
        r.setTask(prev != null
                ? prev.task
                : new TaskRecord(mService.mCurTask, r.info, intent), null, true);
        if (DEBUG_TASKS) Slog.v(TAG, "Starting new activity " + r
                + " in new guessed " + r.task);
    }

    mService.grantUriPermissionFromIntentLocked(callingUid, r.packageName,
            intent, r.getUriPermissionsLocked());

    if (newTask) {
        EventLog.writeEvent(EventLogTags.AM_CREATE_TASK, r.userId, r.task.taskId);
    }
    logStartActivity(EventLogTags.AM_CREATE_ACTIVITY, r, r.task);
    startActivityLocked(r, newTask, doResume, keepCurTransition, options);
    return ActivityManager.START_SUCCESS;
}
```

通过上述代码得了 intent 的标志值，并保存在变量 launchFlags 中。

再看函数 startActivityLocked(r, newTask, doResume)，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void startActivityLocked(ActivityRecord r, boolean newTask,
        boolean doResume, boolean keepCurTransition, Bundle options) {
    final int NH = mHistory.size();

    int addPos = -1;

    if (!newTask) {
        // If starting in an existing task, find where that is...
        boolean startIt = true;
        for (int i = NH-1; i >= 0; i--) {
            ActivityRecord p = mHistory.get(i);
            if (p.finishing) {
                continue;
            }
            if (p.task == r.task) {
                // Here it is!  Now, if this is not yet visible to the
                // user, then just add it without starting; it will
                // get started when the user navigates back to it.
                addPos = i+1;
```

```
            if (!startIt) {
                if (DEBUG_ADD_REMOVE) {
                    RuntimeException here = new RuntimeException("here");
                    here.fillInStackTrace();
                    Slog.i(TAG, "Adding activity " + r + " to stack at " + addPos,
                            here);
                }
                mHistory.add(addPos, r);
                r.putInHistory();
                mService.mWindowManager.addAppToken(addPos, r.appToken, r.task.taskId,
                        r.info.screenOrientation, r.fullscreen,
                        (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0);
                if (VALIDATE_TOKENS) {
                    validateAppTokensLocked();
                }
                ActivityOptions.abort(options);
                return;
            }
            break;
        }
        if (p.fullscreen) {
            startIt = false;
        }
    }
}

// Place a new activity at top of stack, so it is next to interact
// with the user.
if (addPos < 0) {
    addPos = NH;
}
if (addPos < NH) {
    mUserLeaving = false;
    if (DEBUG_USER_LEAVING) Slog.v(TAG, "startActivity() behind front, mUserLeaving=
    false");
}

// Slot the activity into the history stack and proceed
if (DEBUG_ADD_REMOVE) {
    RuntimeException here = new RuntimeException("here");
    here.fillInStackTrace();
    Slog.i(TAG, "Adding activity " + r + " to stack at " + addPos, here);
}
mHistory.add(addPos, r);
r.putInHistory();
r.frontOfTask = newTask;
if (NH > 0) {
    boolean showStartingIcon = newTask;
    ProcessRecord proc = r.app;
    if (proc == null) {
        proc = mService.mProcessNames.get(r.processName, r.info.applicationInfo.uid);
    }
    if (proc == null || proc.thread == null) {
        showStartingIcon = true;
    }
    if (DEBUG_TRANSITION) Slog.v(TAG,
            "Prepare open transition: starting " + r);
    if ((r.intent.getFlags()&Intent.FLAG_ACTIVITY_NO_ANIMATION) != 0) {
        mService.mWindowManager.prepareAppTransition(
                AppTransition.TRANSIT_NONE, keepCurTransition);
        mNoAnimActivities.add(r);
    } else {
        mService.mWindowManager.prepareAppTransition(newTask
                ? AppTransition.TRANSIT_TASK_OPEN
                : AppTransition.TRANSIT_ACTIVITY_OPEN, keepCurTransition);
        mNoAnimActivities.remove(r);
    }
    r.updateOptionsLocked(options);
    mService.mWindowManager.addAppToken(
            addPos, r.appToken, r.task.taskId, r.info.screenOrientation, r.fullscreen,
            (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0);
```

```
            boolean doShow = true;
            if (newTask) {
                if ((r.intent.getFlags()
                        &Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) != 0) {
                    resetTaskIfNeededLocked(r, r);
                    doShow = topRunningNonDelayedActivityLocked(null) == r;
                }
            }
            if (SHOW_APP_STARTING_PREVIEW && doShow) {
                ActivityRecord prev = mResumedActivity;
                if (prev != null) {
                    // We don't want to reuse the previous starting preview if:
                    // (1) The current activity is in a different task.
                    if (prev.task != r.task) prev = null;
                    // (2) The current activity is already displayed.
                    else if (prev.nowVisible) prev = null;
                }
                mService.mWindowManager.setAppStartingWindow(
                        r.appToken, r.packageName, r.theme,
                        mService.compatibilityInfoForPackageLocked(
                                r.info.applicationInfo), r.nonLocalizedLabel,
                        r.labelRes, r.icon, r.windowFlags,
                        prev != null ? prev.appToken : null, showStartingIcon);
            }
        } else {
            // If this is the first activity, don't do any fancy animations,
            // because there is nothing for it to animate on top of.
            mService.mWindowManager.addAppToken(addPos, r.appToken, r.task.taskId,
                    r.info.screenOrientation, r.fullscreen,
                    (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0);
            ActivityOptions.abort(options);
        }
        if (VALIDATE_TOKENS) {
            validateAppTokensLocked();
        }

        if (doResume) {
            resumeTopActivityLocked(null);
        }
    }
```

在上述代码中，NH 表示当前系统中历史任务的个数，因为 Launcher 已经运行起来了，所以，此处 NH 的值大于 0。并且当参数 doResume 为 true 时，调用函数 resumeTopActivityLocked 实现进一步的操作。

10.2.5　执行 Activity 组件的操作

在类 ActivityStack 中，当函数 startActivityLocked 通知 WindowManagerService 服务在准备好一个 Activity 组件切换操作后，如果参数 doResume 的值为 true，则继续调用另外一个函数 resumeTopActivityLocked，以继续执行启动参数 r 所描述的一个 Activity 组件的操作。函数 resumeTopActivityLocked 的核心功能是继续执行启动参数 r 所描述的一个 Activity 组件的操作，具体实现代码如下所示：

```
final boolean resumeTopActivityLocked(ActivityRecord prev, Bundle options) {
    // Find the first activity that is not finishing.
    ActivityRecord next = topRunningActivityLocked(null);
    final boolean userLeaving = mUserLeaving;
    mUserLeaving = false;

    if (next == null) {
        // There are no more activities!  Let's just start up the
        // Launcher...
        if (mMainStack) {
            ActivityOptions.abort(options);
            return mService.startHomeActivityLocked(mCurrentUser);
        }
    }
```

```
next.delayedResume = false;

//看要启动的 Activity 是否就是当前处理 Resumed 状态的 Activity，如果是，那就什么都不用做，
//直接返回就可以；否则再看一下系统当前是否是休眠状态，
//如果是，则再看看要启动的 Activity 是否就是当前处于堆栈顶端的 Activity，
//如果是，则什么都不用做
//如果上面两个条件都不满足，则继续往下执行
if (mResumedActivity == next && next.state == ActivityState.RESUMED) {
    // Make sure we have executed any pending transitions, since there
    // should be nothing left to do at this point.
    mService.mWindowManager.executeAppTransition();
    mNoAnimActivities.clear();
    ActivityOptions.abort(options);
    return false;
}
if ((mService.mSleeping || mService.mShuttingDown)
        && mLastPausedActivity == next
        && (next.state == ActivityState.PAUSED
            || next.state == ActivityState.STOPPED
            || next.state == ActivityState.STOPPING)) {
    // Make sure we have executed any pending transitions, since there
    // should be nothing left to do at this point.
    mService.mWindowManager.executeAppTransition();
    mNoAnimActivities.clear();
    ActivityOptions.abort(options);
    return false;
}
if (mService.mStartedUsers.get(next.userId) == null) {
    Slog.w(TAG, "Skipping resume of top activity " + next
            + ": user " + next.userId + " is stopped");
    return false;
}

mStoppingActivities.remove(next);
mGoingToSleepActivities.remove(next);
next.sleeping = false;
mWaitingVisibleActivities.remove(next);

next.updateOptionsLocked(options);

if (DEBUG_SWITCH) Slog.v(TAG, "Resuming " + next);

// If we are currently pausing an activity, then don't do anything
// until that is done.
if (mPausingActivity != null) {
    if (DEBUG_SWITCH || DEBUG_PAUSE) Slog.v(TAG,
            "Skip resume: pausing=" + mPausingActivity);
    return false;
}
if (false) {
    if (mLastStartedActivity != null && !mLastStartedActivity.finishing) {
        long now = SystemClock.uptimeMillis();
        final boolean inTime = mLastStartedActivity.startTime != 0
                && (mLastStartedActivity.startTime + START_WARN_TIME) >= now;
        final int lastUid = mLastStartedActivity.info.applicationInfo.uid;
        final int nextUid = next.info.applicationInfo.uid;
        if (inTime && lastUid != nextUid
                && lastUid != next.launchedFromUid
                && mService.checkPermission(
                        android.Manifest.permission.STOP_APP_SWITCHES,
                        -1, next.launchedFromUid)
                != PackageManager.PERMISSION_GRANTED) {
            mService.showLaunchWarningLocked(mLastStartedActivity, next);
        } else {
            next.startTime = now;
            mLastStartedActivity = next;
        }
    } else {
        next.startTime = SystemClock.uptimeMillis();
        mLastStartedActivity = next;
```

```
        }
    }
    if (mResumedActivity != null) {
        if (DEBUG_SWITCH) Slog.v(TAG, "Skip resume: need to start pausing");
        if (next.app != null && next.app.thread != null) {
            mService.updateLruProcessLocked(next.app, false);
        }
        startPausingLocked(userLeaving, false);
        return true;
    }
    final ActivityRecord last = mLastPausedActivity;
    if (mService.mSleeping && last != null && !last.finishing) {
        if ((last.intent.getFlags()&Intent.FLAG_ACTIVITY_NO_HISTORY) != 0
                || (last.info.flags&ActivityInfo.FLAG_NO_HISTORY) != 0) {
            if (DEBUG_STATES) {
                Slog.d(TAG, "no-history finish of " + last + " on new resume");
            }
            requestFinishActivityLocked(last.appToken, Activity.RESULT_CANCELED, null,
                    "no-history", false);
        }
    }

    if (prev != null && prev != next) {
        if (!prev.waitingVisible && next != null && !next.nowVisible) {
            prev.waitingVisible = true;
            mWaitingVisibleActivities.add(prev);
            if (DEBUG_SWITCH) Slog.v(
                    TAG, "Resuming top, waiting visible to hide: " + prev);
        } else {
            if (prev.finishing) {
                mService.mWindowManager.setAppVisibility(prev.appToken, false);
                if (DEBUG_SWITCH) Slog.v(TAG, "Not waiting for visible to hide: "
                        + prev + ", waitingVisible="
                        + (prev != null ? prev.waitingVisible : null)
                        + ", nowVisible=" + next.nowVisible);
            } else {
                if(DEBUG_SWITCH)Slog.v(TAG,"Previous already visible but still waiting to hide:"
                    + prev + ", waitingVisible="
                    + (prev != null ? prev.waitingVisible : null)
                    + ", nowVisible=" + next.nowVisible);
            }
        }
    }

    // Launching this app's activity, make sure the app is no longer
    // considered stopped.
    try {
        AppGlobals.getPackageManager().setPackageStoppedState(
                next.packageName, false, next.userId); /* TODO: Verify if correct userid */
    } catch (RemoteException e1) {
    } catch (IllegalArgumentException e) {
        Slog.w(TAG, "Failed trying to unstop package "
                + next.packageName + ": " + e);
    }

    boolean noAnim = false;
    if (prev != null) {
        if (prev.finishing) {
            if (DEBUG_TRANSITION) Slog.v(TAG,
                    "Prepare close transition: prev=" + prev);
            if (mNoAnimActivities.contains(prev)) {
                mService.mWindowManager.prepareAppTransition(
                        AppTransition.TRANSIT_NONE, false);
            } else {
                mService.mWindowManager.prepareAppTransition(prev.task == next.task
                        ? AppTransition.TRANSIT_ACTIVITY_CLOSE
                        : AppTransition.TRANSIT_TASK_CLOSE, false);
            }
            mService.mWindowManager.setAppWillBeHidden(prev.appToken);
            mService.mWindowManager.setAppVisibility(prev.appToken, false);
        } else {
```

```
            if (DEBUG_TRANSITION) Slog.v(TAG,
                    "Prepare open transition: prev=" + prev);
            if (mNoAnimActivities.contains(next)) {
                noAnim = true;
                mService.mWindowManager.prepareAppTransition(
                        AppTransition.TRANSIT_NONE, false);
            } else {
                mService.mWindowManager.prepareAppTransition(prev.task == next.task
                        ? AppTransition.TRANSIT_ACTIVITY_OPEN
                        : AppTransition.TRANSIT_TASK_OPEN, false);
            }
        }
        if (false) {
            mService.mWindowManager.setAppWillBeHidden(prev.appToken);
            mService.mWindowManager.setAppVisibility(prev.appToken, false);
        }
    } else if (mHistory.size() > 1) {
        if (DEBUG_TRANSITION) Slog.v(TAG,
                "Prepare open transition: no previous");
        if (mNoAnimActivities.contains(next)) {
            noAnim = true;
            mService.mWindowManager.prepareAppTransition(
                    AppTransition.TRANSIT_NONE, false);
        } else {
            mService.mWindowManager.prepareAppTransition(
                    AppTransition.TRANSIT_ACTIVITY_OPEN, false);
        }
    }
    if (!noAnim) {
        next.applyOptionsLocked();
    } else {
        next.clearOptionsLocked();
    }

    if (next.app != null && next.app.thread != null) {
        if (DEBUG_SWITCH) Slog.v(TAG, "Resume running: " + next);

        // This activity is now becoming visible.
        mService.mWindowManager.setAppVisibility(next.appToken, true);

        // schedule launch ticks to collect information about slow apps.
        next.startLaunchTickingLocked();

        ActivityRecord lastResumedActivity = mResumedActivity;
        ActivityState lastState = next.state;

        mService.updateCpuStats();

        if (DEBUG_STATES) Slog.v(TAG, "Moving to RESUMED: " + next + " (in existing)");
        next.state = ActivityState.RESUMED;
        mResumedActivity = next;
        next.task.touchActiveTime();
        if (mMainStack) {
            mService.addRecentTaskLocked(next.task);
        }
        mService.updateLruProcessLocked(next.app, true);
        updateLRUListLocked(next);

        // Have the window manager re-evaluate the orientation of
        // the screen based on the new activity order.
        boolean updated = false;
        if (mMainStack) {
            synchronized (mService) {
                Configuration config = mService.mWindowManager.updateOrientationFromApp
Tokens(
                        mService.mConfiguration,
                        next.mayFreezeScreenLocked(next.app) ? next.appToken : null);
                if (config != null) {
                    next.frozenBeforeDestroy = true;
                }
                updated = mService.updateConfigurationLocked(config, next, false, false);
```

```
            }
        }
        if (!updated) {
            ActivityRecord nextNext = topRunningActivityLocked(null);
            if (DEBUG_SWITCH) Slog.i(TAG,
                    "Activity config changed during resume: " + next
                    + ", new next: " + nextNext);
            if (nextNext != next) {
                // Do over!
                mHandler.sendEmptyMessage(RESUME_TOP_ACTIVITY_MSG);
            }
            if (mMainStack) {
                mService.setFocusedActivityLocked(next);
            }
            ensureActivitiesVisibleLocked(null, 0);
            mService.mWindowManager.executeAppTransition();
            mNoAnimActivities.clear();
            return true;
        }

        try {
            // Deliver all pending results.
            ArrayList a = next.results;
            if (a != null) {
                final int N = a.size();
                if (!next.finishing && N > 0) {
                    if (DEBUG_RESULTS) Slog.v(
                            TAG, "Delivering results to " + next
                            + ": " + a);
                    next.app.thread.scheduleSendResult(next.appToken, a);
                }
            }

            if (next.newIntents != null) {
                next.app.thread.scheduleNewIntent(next.newIntents, next.appToken);
            }

            EventLog.writeEvent(EventLogTags.AM_RESUME_ACTIVITY,
                    next.userId, System.identityHashCode(next),
                    next.task.taskId, next.shortComponentName);

            next.sleeping = false;
            showAskCompatModeDialogLocked(next);
            next.app.pendingUiClean = true;
            next.app.thread.scheduleResumeActivity(next.appToken,
                    mService.isNextTransitionForward());

            checkReadyForSleepLocked();

        } catch (Exception e) {
            // Whoops, need to restart this activity!
            if (DEBUG_STATES) Slog.v(TAG, "Resume failed; resetting state to "
                    + lastState + ": " + next);
            next.state = lastState;
            mResumedActivity = lastResumedActivity;
            Slog.i(TAG, "Restarting because process died: " + next);
            if (!next.hasBeenLaunched) {
                next.hasBeenLaunched = true;
            } else {
                if (SHOW_APP_STARTING_PREVIEW && mMainStack) {
                    mService.mWindowManager.setAppStartingWindow(
                            next.appToken, next.packageName, next.theme,
                            mService.compatibilityInfoForPackageLocked(
                                    next.info.applicationInfo),
                            next.nonLocalizedLabel,
                            next.labelRes, next.icon, next.windowFlags,
                            null, true);
                }
            }
            startSpecificActivityLocked(next, true, false);
            return true;
```

```
        }

        // From this point on, if something goes wrong there is no way
        // to recover the activity.
        try {
            next.visible = true;
            completeResumeLocked(next);
        } catch (Exception e) {
            // If any exception gets thrown, toss away this
            // activity and try the next one.
            Slog.w(TAG, "Exception thrown during resume of " + next, e);
            requestFinishActivityLocked(next.appToken, Activity.RESULT_CANCELED, null,
                    "resume-exception", true);
            return true;
        }
        next.stopped = false;

    } else {
        // Whoops, need to restart this activity!
        if (!next.hasBeenLaunched) {
            next.hasBeenLaunched = true;
        } else {
            if (SHOW_APP_STARTING_PREVIEW) {
                mService.mWindowManager.setAppStartingWindow(
                        next.appToken, next.packageName, next.theme,
                        mService.compatibilityInfoForPackageLocked(
                                next.info.applicationInfo),
                        next.nonLocalizedLabel,
                        next.labelRes, next.icon, next.windowFlags,
                        null, true);
            }
            if (DEBUG_SWITCH) Slog.v(TAG, "Restarting: " + next);
        }
        startSpecificActivityLocked(next, true, true);
    }
    return true;
}
```

在上述代码中，首先调用函数 topRunningActivityLocked 获得堆栈顶端的 Activity：MainActivity，并保存在变量 next 中。然后把 mUserLeaving 值保存在本地变量 userLeaving 中，并重新设置为 false。此处因为 Launcher 是当前正被执行的 Activity，所以，mResumedActivity 为 Launcher。当处理休眠状态时，mLastPausedActivity 会保存堆栈顶端的 Activity，并且因为当前不是休眠状态，所以，mLastPausedActivity 为 null。

10.2.6 将 Launcher 推入 Paused 状态

函数 startPausingLocked 的功能是将 Launcher 推入 Paused 状态，此函数在文件 frameworks/base/core/java/android/app/ApplicationThreadNative.java 中定义，具体实现代码如下所示：

```
private final void startPausingLocked(boolean userLeaving, boolean uiSleeping) {
    if (mPausingActivity != null) {
        RuntimeException e = new RuntimeException();
        Slog.e(TAG, "Trying to pause when pause is already pending for "
              + mPausingActivity, e);
    }
    ActivityRecord prev = mResumedActivity;
    if (prev == null) {
        RuntimeException e = new RuntimeException();
        Slog.e(TAG, "Trying to pause when nothing is resumed", e);
        resumeTopActivityLocked(null);
        return;
    }
    if (DEBUG_STATES) Slog.v(TAG, "Moving to PAUSING: " + prev);
    else if (DEBUG_PAUSE) Slog.v(TAG, "Start pausing: " + prev);
    mResumedActivity = null;
    mPausingActivity = prev;
    mLastPausedActivity = prev;
    prev.state = ActivityState.PAUSING;
```

```
        prev.task.touchActiveTime();
        prev.updateThumbnail(screenshotActivities(prev), null);

        mService.updateCpuStats();

        if (prev.app != null && prev.app.thread != null) {
            if (DEBUG_PAUSE) Slog.v(TAG, "Enqueueing pending pause: " + prev);
            try {
                EventLog.writeEvent(EventLogTags.AM_PAUSE_ACTIVITY,
                        prev.userId, System.identityHashCode(prev),
                        prev.shortComponentName);
//参数 prev.finishing 表示 prev 所代表的 Activity 是否正在等待结束的 Activity 列表中，
//由于 Laucher 这个 Activity 还没结束，所以此处为 false
//参数 prev.configChangeFlags 表示哪些 config 发生了变化
                prev.app.thread.schedulePauseActivity(prev.appToken, prev.finishing,
                        userLeaving, prev.configChangeFlags);
                if (mMainStack) {
                    mService.updateUsageStats(prev, false);
                }
            } catch (Exception e) {
                // Ignore exception, if process died other code will cleanup.
                Slog.w(TAG, "Exception thrown during pause", e);
                mPausingActivity = null;
                mLastPausedActivity = null;
            }
        } else {
            mPausingActivity = null;
            mLastPausedActivity = null;
        }

        // If we are not going to sleep, we want to ensure the device is
        // awake until the next activity is started.
        if (!mService.mSleeping && !mService.mShuttingDown) {
            mLaunchingActivity.acquire();
            if (!mHandler.hasMessages(LAUNCH_TIMEOUT_MSG)) {
                // To be safe, don't allow the wake lock to be held for too long.
                Message msg = mHandler.obtainMessage(LAUNCH_TIMEOUT_MSG);
                mHandler.sendMessageDelayed(msg, LAUNCH_TIMEOUT);
            }
        }
        if (mPausingActivity != null) {
            if (!uiSleeping) {
                prev.pauseKeyDispatchingLocked();
            } else {
                if (DEBUG_PAUSE) Slog.v(TAG, "Key dispatch not paused for screen off");
            }
            Message msg = mHandler.obtainMessage(PAUSE_TIMEOUT_MSG);
            msg.obj = prev;
            prev.pauseTime = SystemClock.uptimeMillis();
            mHandler.sendMessageDelayed(msg, PAUSE_TIMEOUT);
            if (DEBUG_PAUSE) Slog.v(TAG, "Waiting for pause to complete...");
        } else {
            if (DEBUG_PAUSE) Slog.v(TAG, "Activity not running, resuming next.");
            resumeTopActivityLocked(null);
        }
    }
```

在上述代码中，首先将 mResumedActivity 保存在本地变量 prev 中，并取出 Launcher 进程中的 ApplicationThread 对象，通过这个对象来通知 Launcher 此 Activity 即将进入 Paused 状态。

再看函数 schedulePauseActivity，功能是通过 Binder 进程间通信机制进入到函数 ApplicationThread.schedulePauseActivity 中。函数 schedulePauseActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
        public final void schedulePauseActivity(IBinder token, boolean finished,
                boolean userLeaving, int configChanges) {
            queueOrSendMessage(
//此处 finished 值为 false，因此，queueOrSendMessage 的第一个参数值为 H.PAUSE_ACTIVITY，
//表示要暂停 token 所代表的 Activity，即 Launcher
                    finished ? H.PAUSE_ACTIVITY_FINISHING : H.PAUSE_ACTIVITY,
```

```
                token,
                (userLeaving ? 1 : 0),
                configChanges);
        }
```

在上述代码中，调用了类 ActivityThread 中的成员函数 queueOrSendMessage。

10.2.7 处理消息

函数 queueOrSendMessage 的功能是先将相关信息组装成一个 msg，然后通过成员变量 mH 发送出去。因为此处 mH 的类型是 H，是一个继承于类 Handler 的 ActivityThread 的内部类，所以，最后由类 H.handleMessage 来处理这个消息。函数 queueOrSendMessage 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final class ActivityThread {
    ......
    private final class H extends Handler {
        ......
        public void handleMessage(Message msg) {
            ......
            switch (msg.what) {
            ......
            case PAUSE_ACTIVITY:
                handlePauseActivity((IBinder)msg.obj, false, msg.arg1 != 0, msg.arg2);
                maybeSnapshot();
                break;
            ......
            }
```

在上述代码中，msg.obj 是一个 ActivityRecord 对象的引用，它代表 Launcher 这个 Activity。

再看具体处理函数 handlePauseActivity，功能是将 Binder 引用的 token 转换成 ActivityRecord 的远程接口 ActivityClientRecord，然后实现如下所示的 3 个功能。

- 如果 userLeaving 为 true，则通过调用函数 performUserLeavingActivity 的方式来调用 Activity.onUserLeaveHint 通知 Activity 用户这就要离开它。
- 通过调用函数 performPauseActivity 的方式来调用函数 Activity.onPause，在 Activity 的生命周期中，当它要让位于其他的 Activity 时，系统就会调用它本身的 onPause 函数。
- 通知 ActivityManagerService 当前的 Activity 已经进入到 Paused 状态。

函数 handlePauseActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private void handlePauseActivity(IBinder token, boolean finished,
        boolean userLeaving, int configChanges) {
    ActivityClientRecord r = mActivities.get(token);
    if (r != null) {
        //Slog.v(TAG, "userLeaving=" + userLeaving + " handling pause of " + r);
        if (userLeaving) {
            performUserLeavingActivity(r);
        }

        r.activity.mConfigChangeFlags |= configChanges;
        performPauseActivity(token, finished, r.isPreHoneycomb());

        // Make sure any pending writes are now committed.
        if (r.isPreHoneycomb()) {
            QueuedWork.waitToFinish();
        }

        // Tell the activity manager we have paused.
        try {
            ActivityManagerNative.getDefault().activityPaused(token);
        } catch (RemoteException ex) {
        }
    }
}
```

接下来看函数 activityPaused，此函数在文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中定义，功能是通过 Binder 进程间通信机制就进入到函数 ActivityManagerService.activityPaused 中。函数 activityPaused 的具体实现代码如下所示：

```
public final void activityPaused(IBinder token, Bundle icicle) {
    ......
    final long origId = Binder.clearCallingIdentity();
    mMainStack.activityPaused(token, icicle, false);
    ......
}
```

10.2.8　暂停完毕

函数 activityPaused 的功能是向 AmS 报告自己已经暂停完毕，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
final void activityPaused(IBinder token, boolean timeout) {
    if (DEBUG_PAUSE) Slog.v(
        TAG, "Activity paused: token=" + token + ", timeout=" + timeout);

    ActivityRecord r = null;

    synchronized (mService) {
        int index = indexOfTokenLocked(token);
        if (index >= 0) {
            r = mHistory.get(index);
            mHandler.removeMessages(PAUSE_TIMEOUT_MSG, r);
            if (mPausingActivity == r) {
                if (DEBUG_STATES) Slog.v(TAG, "Moving to PAUSED: " + r
                        + (timeout ? " (due to timeout)" : " (pause complete)"));
                r.state = ActivityState.PAUSED;
                completePauseLocked();
            } else {
                EventLog.writeEvent(EventLogTags.AM_FAILED_TO_PAUSE,
                        r.userId, System.identityHashCode(r), r.shortComponentName,
                        mPausingActivity != null
                            ? mPausingActivity.shortComponentName : "(none)");
            }
        }
    }
}
```

在上述实现代码中，调用函数 indexOfTokenLocked 找到指定的 token 在 mHistory 中对应的 HistoryRecord。如果的确存在 token，那么可以直接将 token 转换为 HistoryRecord。如果函数 actvityPaused()的参数 timeout 为 false，则说明是非超时的、正常时间内暂停的 Activity，此时需要从 mHandler 中移除还没有被处理的超时消息。

再看函数 completePauseLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void completePauseLocked() {
    ActivityRecord prev = mPausingActivity;
    if (DEBUG_PAUSE) Slog.v(TAG, "Complete pause: " + prev);

    if (prev != null) {
        if (prev.finishing) {
            if (DEBUG_PAUSE) Slog.v(TAG, "Executing finish of activity: " + prev);
            prev = finishCurrentActivityLocked(prev, FINISH_AFTER_VISIBLE, false);
        } else if (prev.app != null) {
            if (DEBUG_PAUSE) Slog.v(TAG, "Enqueueing pending stop: " + prev);
            if (prev.waitingVisible) {
                prev.waitingVisible = false;
                mWaitingVisibleActivities.remove(prev);
                if (DEBUG_SWITCH || DEBUG_PAUSE) Slog.v(
                        TAG, "Complete pause, no longer waiting: " + prev);
            }
            if (prev.configDestroy) {
                if (DEBUG_PAUSE) Slog.v(TAG, "Destroying after pause: " + prev);
```

```
                destroyActivityLocked(prev, true, false, "pause-config");
            } else {
                mStoppingActivities.add(prev);
                if (mStoppingActivities.size() > 3) {
                    if (DEBUG_PAUSE) Slog.v(TAG, "To many pending stops, forcing idle");
                    scheduleIdleLocked();
                } else {
                    checkReadyForSleepLocked();
                }
            }
        } else {
            if (DEBUG_PAUSE) Slog.v(TAG, "App died during pause, not stopping: " + prev);
            prev = null;
        }
        mPausingActivity = null;
    }

    if (!mService.isSleeping()) {
        resumeTopActivityLocked(prev);
    } else {
        checkReadyForSleepLocked();
        ActivityRecord top = topRunningActivityLocked(null);
        if (top == null || (prev != null && top != prev)) {
            resumeTopActivityLocked(null);
        }
    }

    if (prev != null) {
        prev.resumeKeyDispatchingLocked();
    }

    if (prev.app != null && prev.cpuTimeAtResume > 0
            && mService.mBatteryStatsService.isOnBattery()) {
        long diff = 0;
        synchronized (mService.mProcessStatsThread) {
            diff = mService.mProcessStats.getCpuTimeForPid(prev.app.pid)
                    - prev.cpuTimeAtResume;
        }
        if (diff > 0) {
            BatteryStatsImpl bsi = mService.mBatteryStatsService.getActiveStatistics();
            synchronized (bsi) {
                BatteryStatsImpl.Uid.Proc ps =
                        bsi.getProcessStatsLocked(prev.info.applicationInfo.uid,
                        prev.info.packageName);
                if (ps != null) {
                    ps.addForegroundTimeLocked(diff);
                }
            }
        }
    }
    prev.cpuTimeAtResume = 0; // reset it
}
```

在上述代码中，首先清空变量 mPausingActivity，因为现在不需要它了，然后调用函数 resumeTopActivityLokced 继续操作。在函数 resumeTopActivityLokced 中，最终会调用函数 startSpecificActivityLocked 来实现下一步操作。

函数 startSpecificActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
private final void startSpecificActivityLocked(ActivityRecord r,
        boolean andResume, boolean checkConfig) {
    // Is this activity's application already running?
    ProcessRecord app = mService.getProcessRecordLocked(r.processName,
            r.info.applicationInfo.uid);

    if (r.launchTime == 0) {
        r.launchTime = SystemClock.uptimeMillis();
        if (mInitialStartTime == 0) {
            mInitialStartTime = r.launchTime;
```

```
            }
        } else if (mInitialStartTime == 0) {
            mInitialStartTime = SystemClock.uptimeMillis();
        }

        if (app != null && app.thread != null) {
            try {
                app.addPackage(r.info.packageName);
                realStartActivityLocked(r, app, andResume, checkConfig);
                return;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when starting activity "
                        + r.intent.getComponent().flattenToShortString(), e);
            }
        }
        mService.startProcessLocked(r.processName, r.info.applicationInfo, true, 0,
                "activity", r.intent.getComponent(), false, false);
    }
```

在上述代码中，调用函数 startProcessLocked 检查是否存在以“process + uid”命名的进程。函数 startProcessLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
    final ProcessRecord startProcessLocked(String processName,
            ApplicationInfo info, boolean knownToBeDead, int intentFlags,
            String hostingType, ComponentName hostingName, boolean allowWhileBooting,
            boolean isolated) {
        ProcessRecord app;
        if (!isolated) {
            app = getProcessRecordLocked(processName, info.uid);
        } else {
            // If this is an isolated process, it can't re-use an existing process.
            app = null;
        }
        if (DEBUG_PROCESSES) Slog.v(TAG, "startProcess: name=" + processName
                + " app=" + app + " knownToBeDead=" + knownToBeDead
                + " thread=" + (app != null ? app.thread : null)
                + " pid=" + (app != null ? app.pid : -1));
        if (app != null && app.pid > 0) {
            if (!knownToBeDead || app.thread == null) {
                // We already have the app running, or are waiting for it to
                // come up (we have a pid but not yet its thread), so keep it.
                if (DEBUG_PROCESSES) Slog.v(TAG, "App already running: " + app);
                // If this is a new package in the process, add the package to the list
                app.addPackage(info.packageName);
                return app;
            } else {
                // An application record is attached to a previous process,
                // clean it up now.
                if (DEBUG_PROCESSES || DEBUG_CLEANUP) Slog.v(TAG, "App died: " + app);
                handleAppDiedLocked(app, true, true);
            }
        }

        String hostingNameStr = hostingName != null
                ? hostingName.flattenToShortString() : null;

        if (!isolated) {
            if ((intentFlags&Intent.FLAG_FROM_BACKGROUND) != 0) {
                // If we are in the background, then check to see if this process
                // is bad.  If so, we will just silently fail.
                if (mBadProcesses.get(info.processName, info.uid) != null) {
                    if (DEBUG_PROCESSES) Slog.v(TAG, "Bad process: " + info.uid
                            + "/" + info.processName);
                    return null;
                }
            } else {
                if (DEBUG_PROCESSES) Slog.v(TAG, "Clearing bad process: " + info.uid
                        + "/" + info.processName);
                mProcessCrashTimes.remove(info.processName, info.uid);
```

```
            if (mBadProcesses.get(info.processName, info.uid) != null) {
                EventLog.writeEvent(EventLogTags.AM_PROC_GOOD,
                        UserHandle.getUserId(info.uid), info.uid,
                        info.processName);
                mBadProcesses.remove(info.processName, info.uid);
                if (app != null) {
                    app.bad = false;
                }
            }
        }
    }

    if (app == null) {
        app = newProcessRecordLocked(null, info, processName, isolated);
        if (app == null) {
            Slog.w(TAG, "Failed making new process record for "
                    + processName + "/" + info.uid + " isolated=" + isolated);
            return null;
        }
        mProcessNames.put(processName, app.uid, app);
        if (isolated) {
            mIsolatedProcesses.put(app.uid, app);
        }
    } else {
        // If this is a new package in the process, add the package to the list
        app.addPackage(info.packageName);
    }

    // If the system is not ready yet, then hold off on starting this
    // process until it is.
    if (!mProcessesReady
            && !isAllowedWhileBooting(info)
            && !allowWhileBooting) {
        if (!mProcessesOnHold.contains(app)) {
            mProcessesOnHold.add(app);
        }
        if (DEBUG_PROCESSES) Slog.v(TAG, "System not ready, putting on hold: " + app);
        return app;
    }

    startProcessLocked(app, hostingType, hostingNameStr);
    return (app.pid != 0) ? app : null;
}
```

在上述代码中调用了函数 startProcessLocked，而函数 startProcessLocked 调用接口 Process.start 创建了一个新的进程，新的进程会被导入类 android.app.ActivityThread，并且执行其主函数 main。并通过函数 attach 调用了 ActivityManagerService 的远程接口 ActivityManagerProxy 中的函数 attachApplication，传入的参数是 mAppThread，这是一个 ApplicationThread 类型的 Binder 对象，其功能是实现进程之间通信。

10.2.9 建立双向连接

在 Android 系统中，ActivityThread.java 用 thread.attach(false)来调用函数 AttachApplication Locked。AttachApplicationLocked 将本进程与 Activity Manager 建立双向连接，从而使本进程得到 Activity Manager 的服务从 attatch 到 attachApplicationLocked 的调用过程为：

```
attatch-> attachApplication (ActivityManagerService.java 中) -> attachApplicationLocked
```

下面看函数 attachApplication，功能是通过 Binder 驱动程序调用 ActivityManagerService 中的函数 attachApplication。函数 attachApplication 在文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中定义，具体实现代码如下所示：

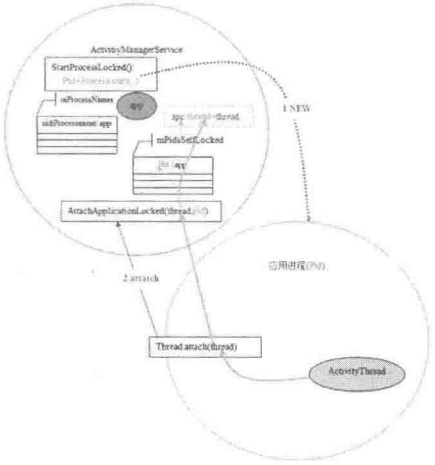
```
public void attachApplication(IApplicationThread app) throws RemoteException
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
```

```
        data.writeInterfaceToken(IActivityManager.descriptor);
        data.writeStrongBinder(app.asBinder());
        mRemote.transact(ATTACH_APPLICATION_TRANSACTION, data, reply, 0);
        reply.readException();
        data.recycle();
        reply.recycle();
    }
```

再看 ActivityManagerService 中的函数 attachApplication，功能是将操作转交给函数 attachApplicationLocked。此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public final void attachApplication(IApplicationThread thread) {
    synchronized (this) {
        int callingPid = Binder.getCallingPid();
        final long origId = Binder.clearCallingIdentity();
        attachApplicationLocked(thread, callingPid);
        Binder.restoreCallingIdentity(origId);
    }
}
```

接下来看函数 attachApplicationLocked，其功能是根据 CallingPid 在 mPidsSelfLocked 找到对应的 ProcessRecord 实例 App，并将 ActvitiyThread 放置在 app.thread 中。这样应用进程和 AMS 建立起双向连接。AM 可以使用 AIDL 接口，通过 app.thread 可以访问应用进程的对象。具体执行过程如图 10-2 所示。



▲图 10-2　函数 attachApplicationLocked 的执行过程

函数 attachApplicationLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final boolean attachApplicationLocked
(IApplicationThread thread,
        int pid) {
    ProcessRecord app;
    if (pid != MY_PID && pid >= 0) {
        synchronized (mPidsSelfLocked) {
            app = mPidsSelfLocked.get(pid);
        }
    } else {
        app = null;
    }

    if (app == null) {
        Slog.w(TAG, "No pending application record for pid " + pid
                + " (IApplicationThread " + thread + "); dropping process");
        EventLog.writeEvent(EventLogTags.AM_DROP_PROCESS, pid);
        if (pid > 0 && pid != MY_PID) {
            Process.killProcessQuiet(pid);
        } else {
            try {
                thread.scheduleExit();
            } catch (Exception e) {
                // Ignore exceptions.
            }
        }
        return false;
    }
    if (app.thread != null) {
        handleAppDiedLocked(app, true, true);
    }
    if (localLOGV) Slog.v(
            TAG, "Binding process pid " + pid + " to record " + app);

    String processName = app.processName;
    try {
        AppDeathRecipient adr = new AppDeathRecipient(
```

```
            app, pid, thread);
    thread.asBinder().linkToDeath(adr, 0);
    app.deathRecipient = adr;
} catch (RemoteException e) {
    app.resetPackageList();
    startProcessLocked(app, "link fail", processName);
    return false;
}

EventLog.writeEvent(EventLogTags.AM_PROC_BOUND, app.userId, app.pid, app.processName);

app.thread = thread;
app.curAdj = app.setAdj = -100;
app.curSchedGroup = Process.THREAD_GROUP_DEFAULT;
app.setSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
app.forcingToForeground = null;
app.foregroundServices = false;
app.hasShownUi = false;
app.debugging = false;

mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);

boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
List providers = normalMode ? generateApplicationProvidersLocked(app) : null;

if (!normalMode) {
    Slog.i(TAG, "Launching preboot mode app: " + app);
}

if (localLOGV) Slog.v(
    TAG, "New app record " + app
    + " thread=" + thread.asBinder() + " pid=" + pid);
try {
    int testMode = IApplicationThread.DEBUG_OFF;
    if (mDebugApp != null && mDebugApp.equals(processName)) {
        testMode = mWaitForDebugger
            ? IApplicationThread.DEBUG_WAIT
            : IApplicationThread.DEBUG_ON;
        app.debugging = true;
        if (mDebugTransient) {
            mDebugApp = mOrigDebugApp;
            mWaitForDebugger = mOrigWaitForDebugger;
        }
    }
    String profileFile = app.instrumentationProfileFile;
    ParcelFileDescriptor profileFd = null;
    boolean profileAutoStop = false;
    if (mProfileApp != null && mProfileApp.equals(processName)) {
        mProfileProc = app;
        profileFile = mProfileFile;
        profileFd = mProfileFd;
        profileAutoStop = mAutoStopProfiler;
    }
    boolean enableOpenGlTrace = false;
    if (mOpenGlTraceApp != null && mOpenGlTraceApp.equals(processName)) {
        enableOpenGlTrace = true;
        mOpenGlTraceApp = null;
    }

    // If the app is being launched for restore or full backup, set it up specially
    boolean isRestrictedBackupMode = false;
    if (mBackupTarget != null && mBackupAppName.equals(processName)) {
        isRestrictedBackupMode = (mBackupTarget.backupMode == BackupRecord.RESTORE)
                || (mBackupTarget.backupMode == BackupRecord.RESTORE_FULL)
                || (mBackupTarget.backupMode == BackupRecord.BACKUP_FULL);
    }

    ensurePackageDexOpt(app.instrumentationInfo != null
            ? app.instrumentationInfo.packageName
            : app.info.packageName);
    if (app.instrumentationClass != null) {
```

```
            ensurePackageDexOpt(app.instrumentationClass.getPackageName());
        }
        if (DEBUG_CONFIGURATION) Slog.v(TAG, "Binding proc "
                + processName + " with config " + mConfiguration);
        ApplicationInfo appInfo = app.instrumentationInfo != null
                ? app.instrumentationInfo : app.info;
        app.compat = compatibilityInfoForPackageLocked(appInfo);
        if (profileFd != null) {
            profileFd = profileFd.dup();
        }
        thread.bindApplication(processName, appInfo, providers,
                app.instrumentationClass, profileFile, profileFd, profileAutoStop,
                app.instrumentationArguments, app.instrumentationWatcher,
                app.instrumentationUiAutomationConnection, testMode, enableOpenGlTrace,
                isRestrictedBackupMode || !normalMode, app.persistent,
                new Configuration(mConfiguration), app.compat, getCommonServicesLocked(),
                mCoreSettingsObserver.getCoreSettingsLocked());
        updateLruProcessLocked(app, false);
        app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
    } catch (Exception e) {
        Slog.w(TAG, "Exception thrown during bind!", e);

        app.resetPackageList();
        app.unlinkDeathRecipient();
        startProcessLocked(app, "bind fail", processName);
        return false;
    }

    // Remove this record from the list of starting applications.
    mPersistentStartingProcesses.remove(app);
    if (DEBUG_PROCESSES && mProcessesOnHold.contains(app)) Slog.v(TAG,
            "Attach application locked removing on hold: " + app);
    mProcessesOnHold.remove(app);

    boolean badApp = false;
    boolean didSomething = false;

    // See if the top visible activity is waiting to run in this process...
    ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
    if (hr != null && normalMode) {
        if (hr.app == null && app.uid == hr.info.applicationInfo.uid
                && processName.equals(hr.processName)) {
            try {
                if (mHeadless) {
                    Slog.e(TAG, "Starting activities not supported on headless device: " + hr);
                } else if (mMainStack.realStartActivityLocked(hr, app, true, true)) {
                    didSomething = true;
                }
            } catch (Exception e) {
                Slog.w(TAG, "Exception in new application when starting activity "
                        + hr.intent.getComponent().flattenToShortString(), e);
                badApp = true;
            }
        } else {
            mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);
        }
    }

    // Find any services that should be running in this process...
    if (!badApp) {
        try {
            didSomething |= mServices.attachApplicationLocked(app, processName);
        } catch (Exception e) {
            badApp = true;
        }
    }

    // Check if a next-broadcast receiver is in this process...
    if (!badApp && isPendingBroadcastProcessLocked(pid)) {
        try {
            didSomething = sendPendingBroadcastsLocked(app);
```

```
            } catch (Exception e) {
                // If the app died trying to launch the receiver we declare it 'bad'
                badApp = true;
            }
        }

        // Check whether the next backup agent is in this process...
        if (!badApp && mBackupTarget != null && mBackupTarget.appInfo.uid == app.uid) {
            if (DEBUG_BACKUP) Slog.v(TAG, "New app is backup target, launching agent for " + app);
            ensurePackageDexOpt(mBackupTarget.appInfo.packageName);
            try {
                thread.scheduleCreateBackupAgent(mBackupTarget.appInfo,
                        compatibilityInfoForPackageLocked(mBackupTarget.appInfo),
                        mBackupTarget.backupMode);
            } catch (Exception e) {
                Slog.w(TAG, "Exception scheduling backup agent creation: ");
                e.printStackTrace();
            }
        }

        if (badApp) {
            // todo: Also need to kill application to deal with all
            // kinds of exceptions.
            handleAppDiedLocked(app, false, true);
            return false;
        }

        if (!didSomething) {
            updateOomAdjLocked();
        }

        return true;
    }
```

在上述代码中，先通过 pid 将取回已经创建的 ProcessRecord，并放在 App 变量中；然后初始化 App 中的其他成员，最后调用函数 mMainStack.realStartActivityLocked 执行真正的 Activity 启动操作。在此处启动 Activity 的过程，是通过调用函数 mMainStack.topRunningActivityLocked(null) 的方式从堆栈顶端取回来的，此时在堆栈顶端的 Activity 就是 MainActivity。

10.2.10 启动新的 Activity

函数 realStartActivityLocked 的功能是启动新的 Activity，并将新的 Activity 的相关的信息保存在参数 r 中。函数 realStartActivityLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityStack.java 中定义，具体实现代码如下所示：

```
    final boolean realStartActivityLocked(ActivityRecord r,
            ProcessRecord app, boolean andResume, boolean checkConfig)
            throws RemoteException {

        r.startFreezingScreenLocked(app, 0);
        mService.mWindowManager.setAppVisibility(r.appToken, true);

        // schedule launch ticks to collect information about slow apps.
        r.startLaunchTickingLocked();
        if (checkConfig) {
            Configuration config = mService.mWindowManager.updateOrientationFromAppTokens(
                    mService.mConfiguration,
                    r.mayFreezeScreenLocked(app) ? r.appToken : null);
            mService.updateConfigurationLocked(config, r, false, false);
        }

        r.app = app;
        app.waitingToKill = null;
        r.launchCount++;
        r.lastLaunchTime = SystemClock.uptimeMillis();

        if (localLOGV) Slog.v(TAG, "Launching: " + r);
```

```
int idx = app.activities.indexOf(r);
if (idx < 0) {
    app.activities.add(r);
}
mService.updateLruProcessLocked(app, true);

try {
    if (app.thread == null) {
        throw new RemoteException();
    }
    List<ResultInfo> results = null;
    List<Intent> newIntents = null;
    if (andResume) {
        results = r.results;
        newIntents = r.newIntents;
    }
    if (DEBUG_SWITCH) Slog.v(TAG, "Launching: " + r
            + " icicle=" + r.icicle
            + " with results=" + results + " newIntents=" + newIntents
            + " andResume=" + andResume);
    if (andResume) {
        EventLog.writeEvent(EventLogTags.AM_RESTART_ACTIVITY,
                r.userId, System.identityHashCode(r),
                r.task.taskId, r.shortComponentName);
    }
    if (r.isHomeActivity) {
        mService.mHomeProcess = app;
    }
    mService.ensurePackageDexOpt(r.intent.getComponent().getPackageName());
    r.sleeping = false;
    r.forceNewConfig = false;
    showAskCompatModeDialogLocked(r);
    r.compat = mService.compatibilityInfoForPackageLocked(r.info.applicationInfo);
    String profileFile = null;
    ParcelFileDescriptor profileFd = null;
    boolean profileAutoStop = false;
    if (mService.mProfileApp != null && mService.mProfileApp.equals(app.processName)) {
        if (mService.mProfileProc == null || mService.mProfileProc == app) {
            mService.mProfileProc = app;
            profileFile = mService.mProfileFile;
            profileFd = mService.mProfileFd;
            profileAutoStop = mService.mAutoStopProfiler;
        }
    }
    app.hasShownUi = true;
    app.pendingUiClean = true;
    if (profileFd != null) {
        try {
            profileFd = profileFd.dup();
        } catch (IOException e) {
            profileFd = null;
        }
    }
    app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
            System.identityHashCode(r), r.info,
            new Configuration(mService.mConfiguration),
            r.compat, r.icicle, results, newIntents, !andResume,
            mService.isNextTransitionForward(), profileFile, profileFd,
            profileAutoStop);

    if ((app.info.flags&ApplicationInfo.FLAG_CANT_SAVE_STATE) != 0) {
        if (app.processName.equals(app.info.packageName)) {
            if (mService.mHeavyWeightProcess != null
                    && mService.mHeavyWeightProcess != app) {
                Log.w(TAG, "Starting new heavy weight process " + app
                        + " when already running "
                        + mService.mHeavyWeightProcess);
            }
            mService.mHeavyWeightProcess = app;
            Message msg = mService.mHandler.obtainMessage(
                    ActivityManagerService.POST_HEAVY_NOTIFICATION_MSG);
```

```
                msg.obj = r;
                mService.mHandler.sendMessage(msg);
            }
        }

    } catch (RemoteException e) {
        if (r.launchFailed) {
            // This is the second time we failed -- finish activity
            // and give up.
            Slog.e(TAG, "Second failure launching "
                  + r.intent.getComponent().flattenToShortString()
                  + ", giving up", e);
            mService.appDiedLocked(app, app.pid, app.thread);
            requestFinishActivityLocked(r.appToken, Activity.RESULT_CANCELED, null,
                  "2nd-crash", false);
            return false;
        }
        app.activities.remove(r);
        throw e;
    }

    r.launchFailed = false;
    if (updateLRUListLocked(r)) {
        Slog.w(TAG, "Activity " + r
              + " being launched, but already in LRU list");
    }

    if (andResume) {
        // As part of the process of launching, ActivityThread also performs
        // a resume.
        r.state = ActivityState.RESUMED;
        if (DEBUG_STATES) Slog.v(TAG, "Moving to RESUMED: " + r
                + " (starting new instance)");
        r.stopped = false;
        mResumedActivity = r;
        r.task.touchActiveTime();
        if (mMainStack) {
            mService.addRecentTaskLocked(r.task);
        }
        completeResumeLocked(r);
        checkReadyForSleepLocked();
        if (DEBUG_SAVED_STATE) Slog.i(TAG, "Launch completed; removing icicle of " + r.icicle);
    } else {
        if (DEBUG_STATES) Slog.v(TAG, "Moving to STOPPED: " + r
                + " (starting in stopped state)");
        r.state = ActivityState.STOPPED;
        r.stopped = true;
    }
    if (mMainStack) {
        mService.startSetupActivityLocked();
    }

    return true;
}
```

在上述代码中，参数 r 是一个 ActivityRecord 类型的 Binder 对象，用于当做这个 Activity 的 token 值。通过上述实现代码，调用 app.thread 进入到了 ApplicationThreadProxy 中的函数 scheduleLaunchActivity 中。

10.2.11 通知机制

接下来看函数 scheduleLaunchActivity，功能是通知应用程序它可以加载应用程序中的默认 Activity。此函数最终会把这个请求封装成一个消息，然后通过类 ActivityThread 的成员变量 mH 将这个消息加入到应用程序的消息队列中去。函数 scheduleLaunchActivity 在文件 frameworks/base/core/java/android/app/ApplicationThreadNative.java 中定义，具体实现代码如下所示：

```
public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
        ActivityInfo info, Bundle state, List<ResultInfo> pendingResults,
```

```
        List<Intent> pendingNewIntents, boolean notResumed, boolean isForward)
        throws RemoteException {
    Parcel data = Parcel.obtain();
    data.writeInterfaceToken(IApplicationThread.descriptor);
    intent.writeToParcel(data, 0);
    data.writeStrongBinder(token);
    data.writeInt(ident);
    info.writeToParcel(data, 0);
    data.writeBundle(state);
    data.writeTypedList(pendingResults);
    data.writeTypedList(pendingNewIntents);
    data.writeInt(notResumed ? 1 : 0);
    data.writeInt(isForward ? 1 : 0);
    mRemote.transact(SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION, data, null,
        IBinder.FLAG_ONEWAY);
    data.recycle();
}
```

通过上述实现代码可知，函数 scheduleLaunchActivity 最终通过 Binder 驱动程序进入到类 ApplicationThread 中的函数 scheduleLaunchActivity 中。类 ApplicationThread 中的函数 schedule LaunchActivity 的功能是创建一个 ActivityClientRecord 实例，并且初始化它的成员变量，然后调用类 ActivityThread 的成员函数 queueOrSendMessage 实现进一步处理。函数 scheduleLaunchActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final void scheduleLaunchActivity(Intent intent, IBinder token, int ident,
        ActivityInfo info, Configuration curConfig, CompatibilityInfo compatInfo,
        Bundle state, List<ResultInfo> pendingResults,
        List<Intent> pendingNewIntents, boolean notResumed, boolean isForward,
        String profileName, ParcelFileDescriptor profileFd, boolean autoStopProfiler) {
    ActivityClientRecord r = new ActivityClientRecord();
    r.token = token;
    r.ident = ident;
    r.intent = intent;
    r.activityInfo = info;
    r.compatInfo = compatInfo;
    r.state = state;
    r.pendingResults = pendingResults;
    r.pendingIntents = pendingNewIntents;

    r.startsNotResumed = notResumed;
    r.isForward = isForward;
    r.profileFile = profileName;
    r.profileFd = profileFd;
    r.autoStopProfiler = autoStopProfiler;
    updatePendingConfiguration(curConfig);
    queueOrSendMessage(H.LAUNCH_ACTIVITY, r);
}
```

10.2.12　发送消息

再看类 ActivityThread 中的函数 queueOrSendMessage，功能是将消息内容放在 msg 中，然后通过 mH 把消息分发出去。当发出消息之后调用函数 handleMessage。函数 queueOrSendMessage 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，主要实现代码如下所示：

```
private final void queueOrSendMessage(int what, Object obj) {
    queueOrSendMessage(what, obj, 0, 0);
}
private final void queueOrSendMessage(int what, Object obj, int arg1, int arg2) {
    synchronized (this) {
        ......
        Message msg = Message.obtain();
        msg.what = what;
        msg.obj = obj;
        msg.arg1 = arg1;
        msg.arg2 = arg2;
        mH.sendMessage(msg);
    }
}
```

在函数 handleMessage 中，会调用类 ActivityThread 中的函数 handleLaunchActivity 实现进一步处理。函数 handleLaunchActivity 会首先调用函数 performLaunchActivity 来加载 Activity 类，然后返回到 handleLaunchActivity 函数并再调用函数 handleResumeActivity 来使这个 Activity 进入 Resumed 状态。由此可见，整个过程完全遵循 Activity 的生命周期。函数 performLaunchActivity 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示。

```
    private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent)
{
        // System.out.println("##### [" + System.currentTimeMillis() + "] ActivityThread.
performLaunchActivity(" + r + ")");
//收集要启动的 Activity 的相关信息，主要是 package 和 component 中的信息
        ActivityInfo aInfo = r.activityInfo;
        if (r.packageInfo == null) {
            r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo,
                    Context.CONTEXT_INCLUDE_CODE);
        }

        ComponentName component = r.intent.getComponent();
        if (component == null) {
            component = r.intent.resolveActivity(
                mInitialApplication.getPackageManager());
            r.intent.setComponent(component);
        }
//通过 ClassLoader 将类 MainActivity 加载进来
        if (r.activityInfo.targetActivity != null) {
            component = new ComponentName(r.activityInfo.packageName,
                    r.activityInfo.targetActivity);
        }

        Activity activity = null;
        try {
            java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
            activity = mInstrumentation.newActivity(
                    cl, component.getClassName(), r.intent);
            StrictMode.incrementExpectedActivityCount(activity.getClass());
            r.intent.setExtrasClassLoader(cl);
            if (r.state != null) {
                r.state.setClassLoader(cl);
            }
        } catch (Exception e) {
            if (!mInstrumentation.onException(activity, e)) {
                throw new RuntimeException(
                    "Unable to instantiate activity " + component
                    + ": " + e.toString(), e);
            }
        }

        try {
//创建 Application 对象，这是根据 AndroidManifest.xml 配置文件中的 Application 标签的信息来创建的
            Application app = r.packageInfo.makeApplication(false, mInstrumentation);

            if (localLOGV) Slog.v(TAG, "Performing launch of " + r);
            if (localLOGV) Slog.v(
                    TAG, r + ": app=" + app
                    + ", appName=" + app.getPackageName()
                    + ", pkg=" + r.packageInfo.getPackageName()
                    + ", comp=" + r.intent.getComponent().toShortString()
                    + ", dir=" + r.packageInfo.getAppDir());

            if (activity != null) {
                Context appContext = createBaseContextForActivity(r, activity);
                CharSequence title=r.activityInfo.loadLabel(appContext.getPackage Manager());
                Configuration config = new Configuration(mCompatConfiguration);
                if (DEBUG_CONFIGURATION) Slog.v(TAG, "Launching activity "
                        + r.activityInfo.name + " with config " + config);
//创建 Activity 的上下文信息，并通过方法 attach 将上下文信息设置到 MainActivity 中
                activity.attach(appContext, this, getInstrumentation(), r.token,
                        r.ident, app, r.intent, r.activityInfo, title, r.parent,
                        r.embeddedID, r.lastNonConfigurationInstances, config);
```

```
            if (customIntent != null) {
                activity.mIntent = customIntent;
            }
            r.lastNonConfigurationInstances = null;
            activity.mStartedActivity = false;
            int theme = r.activityInfo.getThemeResource();
            if (theme != 0) {
                activity.setTheme(theme);
            }

            activity.mCalled = false;
            //调用 MainActivity 中的 onCreate 函数
            mInstrumentation.callActivityOnCreate(activity, r.state);
            if (!activity.mCalled) {
                throw new SuperNotCalledException(
                    "Activity " + r.intent.getComponent().toShortString() +
                    " did not call through to super.onCreate()");
            }
            r.activity = activity;
            r.stopped = true;
            if (!r.activity.mFinished) {
                activity.performStart();
                r.stopped = false;
            }
            if (!r.activity.mFinished) {
                if (r.state != null) {
                    mInstrumentation.callActivityOnRestoreInstanceState(activity,
                    r.state);
                }
            }
            if (!r.activity.mFinished) {
                activity.mCalled = false;
                mInstrumentation.callActivityOnPostCreate(activity, r.state);
                if (!activity.mCalled) {
                    throw new SuperNotCalledException(
                        "Activity " + r.intent.getComponent().toShortString() +
                        " did not call through to super.onPostCreate()");
                }
            }
        }
        r.paused = true;

        mActivities.put(r.token, r);

    } catch (SuperNotCalledException e) {
        throw e;

    } catch (Exception e) {
        if (!mInstrumentation.onException(activity, e)) {
            throw new RuntimeException(
                "Unable to start activity " + component
                + ": " + e.toString(), e);
        }
    }

    return activity;
}
```

在上述代码中，并不是直接调用了 MainActivity 中的 onCreate 函数，而是通过 mInstrumentation 中的函数 callActivityOnCreate 来间接调用的。mInstrumentation 在此处的作用是监控 Activity 与系统的交互操作，其实就是系统的运行日志。

在整个过程的最后，可以自定义建立一个 App 工程文件来测试启动 Activity 的效果。例如，可以用如下所示的代码进行测试：

```
public void onCreate(Bundle savedInstanceState) {
    ……
    Log.i(LOG_TAG, "Main Activity Created.");
```

第 11 章　应用程序管理服务——PackageManagerService 分析

在启动 Android 系统的过程中，会随之启动一个应用程序管理服务 PackageManagerService，其功能是扫描系统中的特定目录，寻找里面以“.apk”为后缀的应用程序文件，并对这些文件进行解析以得到应用程序的相关信息，最后完成对这些应用程序的安装过程。在本章的内容中，将详细分析 PackageManagerService 服务的源代码知识。

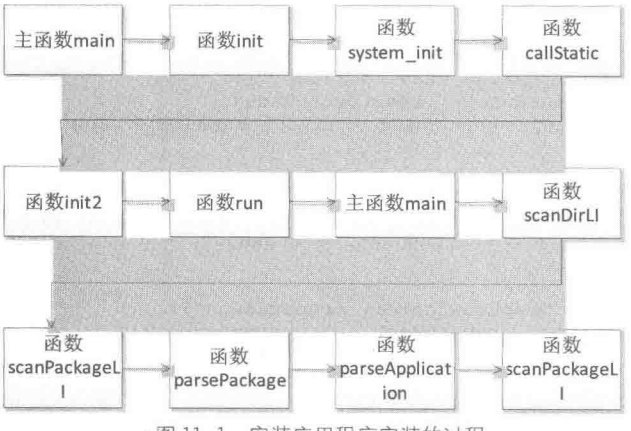
11.1 PackageManagerService 概述

在 Android 系统中，PackageManagerService 是一个应用程序管理服务。安装 Android 应用程序的过程就是解析应用程序配置文件 AndroidManifest.xml 的过程，并从里面得到应用程序的相关信息，例如，得到应用程序的 Activity、Service、Broadcast Receiver 和 Content Provider 等组件信息。当获取这些相关信息后，通过 ActivityManagerService 服务就可以在系统中正常地使用这些应用程序。

在 Android 5.0 系统中，在系统启动时通过 SystemServer 组件启动应用程序管理服务 PackageManagerService，启动后会执行应用程序安装的过程。Android 启动时会把已有的 App 安装一遍，整个过程主要分 3 部分：读取安装信息、扫描安装和写回安装信息。读取和写回主要是针对于安装信息文件，这些信息保证了启动后 App 与上一次的一致。关键步骤是扫描指定目录下的 APK 并安装。Android 中的 APK 主要分布在以下几个目录，意味着启动时要扫描的主要也是这几个目录。

- 系统核心应用：/system/priv-app。
- 系统 App：/system/app。
- 非系统 App：/data/app（安装于手机存储的一般 app）或/mnt/asec//pkg.apk（sdcard 或 forward-locked）。
- 受 DRM 保护 App：/data/app-private。
- vendor-specific 的 App：/vendor/app。
- 资源型 App：/system/framework。

从 SystemServer 进程启动 PackageManagerService 服务开始，在 Android 系统中安装应用程序的过程如图 11-1 所示。



▲图 11-1　安装应用程序安装的过程

启动 PackageManagerService 服务需要 4 个参数，context 上下文环境信息是由 ActivityManager Service 获取的，Installer 是一个安装器，是对 install 程序的一个封装，在新创建一个 Installer 后会调用 ping 命令测试是否能连接上 install 的服务端。

11.2 系统进程启动

在 Android 5.0 源代码中，打开文件 frameworks/base/services/java/com/android/server/SystemServer.java 中，在函数 startBootstrapServices 中调用 PackageManagerService 的函数 main()，具体实现代码如下所示：

```
……  mSystemServiceManager.startBootPhase(SystemService.PHASE_WAIT_FOR_DEFAULT_DISPLAY);

        // Only run "core" apps if we're encrypting the device.
        String cryptState = SystemProperties.get("vold.decrypt");
        if (ENCRYPTING_STATE.equals(cryptState)) {
            Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
            mOnlyCore = true;
        } else if (ENCRYPTED_STATE.equals(cryptState)) {
            Slog.w(TAG, "Device encrypted - only parsing core apps");
            mOnlyCore = true;
        }

        // Start the package manager.
        Slog.i(TAG, "Package Manager");
        mPackageManagerService = PackageManagerService.main(mSystemContext, mInstaller,
                mFactoryTestMode != FactoryTest.FACTORY_TEST_OFF, mOnlyCore);
        mFirstBoot = mPackageManagerService.isFirstBoot();
        mPackageManager = mSystemContext.getPackageManager();

        Slog.i(TAG, "User Service");
        ServiceManager.addService(Context.USER_SERVICE, UserManagerService.getInstance());

        // Initialize attribute cache used to cache resources from packages.
        AttributeCache.init(mSystemContext);

        // Set up the Application instance for the system process and get started.
        mActivityManagerService.setSystemProcess();
    }
……
```

11.3 开始运行

在线程 ServerThread 中启动 PackageManagerService 服务后，开始执行 ServerThread 实例 thr 的 run 函数。函数 run 在 frameworks/base/services/java/com/android/server/SystemServer.java 文件中定义，具体实现代码如下所示：

```
    public void run() {
        Slog.i(TAG, "Making services ready");
        mSystemServiceManager.startBootPhase(
                SystemService.PHASE_ACTIVITY_MANAGER_READY);

        try {
            mActivityManagerService.startObservingNativeCrashes();
        } catch (Throwable e) {
            reportWtf("observing native crashes", e);
        }

        Slog.i(TAG, "WebViewFactory preparation");
        WebViewFactory.prepareWebViewInSystemServer();

        try {
            startSystemUi(context);
        } catch (Throwable e) {
```

```
            reportWtf("starting System UI", e);
        }
        try {
            if (mountServiceF != null) mountServiceF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Mount Service ready", e);
        }
        try {
            if (networkScoreF != null) networkScoreF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Network Score Service ready", e);
        }
        try {
            if (networkManagementF != null) networkManagementF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Network Managment Service ready", e);
        }
        try {
            if (networkStatsF != null) networkStatsF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Network Stats Service ready", e);
        }
        try {
            if (networkPolicyF != null) networkPolicyF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Network Policy Service ready", e);
        }
        try {
            if (connectivityF != null) connectivityF.systemReady();
        } catch (Throwable e) {
            reportWtf("making Connectivity Service ready", e);
        }
        try {
            if (audioServiceF != null) audioServiceF.systemReady();
        } catch (Throwable e) {
            reportWtf("Notifying AudioService running", e);
        }
        Watchdog.getInstance().start();

        // It is now okay to let the various system services start their
        // third party code...
        mSystemServiceManager.startBootPhase(
                SystemService.PHASE_THIRD_PARTY_APPS_CAN_START);

        try {
            if (wallpaperF != null) wallpaperF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying WallpaperService running", e);
        }
        try {
            if (immF != null) immF.systemRunning(statusBarF);
        } catch (Throwable e) {
            reportWtf("Notifying InputMethodService running", e);
        }
        try {
            if (locationF != null) locationF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying Location Service running", e);
        }
        try {
            if (countryDetectorF != null) countryDetectorF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying CountryDetectorService running", e);
        }
        try {
            if (networkTimeUpdaterF != null) networkTimeUpdaterF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying NetworkTimeService running", e);
        }
        try {
            if (commonTimeMgmtServiceF != null) {
```

```
                commonTimeMgmtServiceF.systemRunning();
            }
        } catch (Throwable e) {
            reportWtf("Notifying CommonTimeManagementService running", e);
        }
        try {
            if (textServiceManagerServiceF != null)
                textServiceManagerServiceF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying TextServicesManagerService running", e);
        }
        try {
            if (atlasF != null) atlasF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying AssetAtlasService running", e);
        }
        try {
            // TODO(BT) Pass parameter to input manager
            if (inputManagerF != null) inputManagerF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying InputManagerService running", e);
        }
        try {
            if (telephonyRegistryF != null) telephonyRegistryF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying TelephonyRegistry running", e);
        }
        try {
            if (mediaRouterF != null) mediaRouterF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying MediaRouterService running", e);
        }

        try {
            if (mmsServiceF != null) mmsServiceF.systemRunning();
        } catch (Throwable e) {
            reportWtf("Notifying MmsService running", e);
        }
    }
});
}
```

通过上述代码可知，函数 run 除了启动 PackageManagerService 服务之外，还启动了其他的很多服务。

在文件 frameworks\base\services\core\java\com\android\server\pm\PackageManagerService.java 中，定义了 PackageManagerService 的主函数 main()，具体实现代码如下所示：

```
public static final PackageManagerService main(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
    PackageManagerService m = new PackageManagerService(context, installer,
            factoryTest, onlyCore);
    ServiceManager.addService("package", m);
    return m;
}
```

在上述代码中构造了 PackageManagerService，并将其加到 ServiceManager 服务中，这样其他的组件就可以用该服务了。在 PackageManagerService 的构造函数中可以看到，除了主线程外还会有一个叫 PackageManager 的工作线程。它主要是用在其他安装方式中，因为启动时没有用户交互，基本上不需要把工作交给后台。构造函数 PackageManagerService 的具体实现代码如下所示：

```
public PackageManagerService(Context context, Installer installer,
        boolean factoryTest, boolean onlyCore) {
    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_START,
            SystemClock.uptimeMillis());

    if (mSdkVersion <= 0) {
        Slog.w(TAG, "**** ro.build.version.sdk not set!");
    }
```

```
mContext = context;
mFactoryTest = factoryTest;
mOnlyCore = onlyCore;
mLazyDexOpt = "eng".equals(SystemProperties.get("ro.build.type"));
mMetrics = new DisplayMetrics();
mSettings = new Settings(context);
mSettings.addSharedUserLPw("android.uid.system", Process.SYSTEM_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.phone", RADIO_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.log", LOG_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.nfc", NFC_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.bluetooth", BLUETOOTH_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);
mSettings.addSharedUserLPw("android.uid.shell", SHELL_UID,
        ApplicationInfo.FLAG_SYSTEM|ApplicationInfo.FLAG_PRIVILEGED);

String separateProcesses = SystemProperties.get("debug.separate_processes");
if (separateProcesses != null && separateProcesses.length() > 0) {
    if ("*".equals(separateProcesses)) {
        mDefParseFlags = PackageParser.PARSE_IGNORE_PROCESSES;
        mSeparateProcesses = null;
        Slog.w(TAG, "Running with debug.separate_processes: * (ALL)");
    } else {
        mDefParseFlags = 0;
        mSeparateProcesses = separateProcesses.split(",");
        Slog.w(TAG, "Running with debug.separate_processes: "
                + separateProcesses);
    }
} else {
    mDefParseFlags = 0;
    mSeparateProcesses = null;
}

mInstaller = installer;

getDefaultDisplayMetrics(context, mMetrics);

SystemConfig systemConfig = SystemConfig.getInstance();
mGlobalGids = systemConfig.getGlobalGids();
mSystemPermissions = systemConfig.getSystemPermissions();
mAvailableFeatures = systemConfig.getAvailableFeatures();

synchronized (mInstallLock) {
// writer
synchronized (mPackages) {
    mHandlerThread = new ServiceThread(TAG,
            Process.THREAD_PRIORITY_BACKGROUND, true /*allowIo*/);
    mHandlerThread.start();
    mHandler = new PackageHandler(mHandlerThread.getLooper());
    Watchdog.getInstance().addThread(mHandler, WATCHDOG_TIMEOUT);

    File dataDir = Environment.getDataDirectory();
    mAppDataDir = new File(dataDir, "data");
    mAppInstallDir = new File(dataDir, "app");
    mAppLib32InstallDir = new File(dataDir, "app-lib");
    mAsecInternalPath = new File(dataDir, "app-asec").getPath();
    mUserAppDataDir = new File(dataDir, "user");
    mDrmAppPrivateInstallDir = new File(dataDir, "app-private");

    sUserManager = new UserManagerService(context, this,
            mInstallLock, mPackages);

    // Propagate permission configuration in to package manager.
    ArrayMap<String, SystemConfig.PermissionEntry> permConfig
            = systemConfig.getPermissions();
    for (int i=0; i<permConfig.size(); i++) {
        SystemConfig.PermissionEntry perm = permConfig.valueAt(i);
        BasePermission bp = mSettings.mPermissions.get(perm.name);
```

```
            if (bp == null) {
                bp = new BasePermission(perm.name, "android", BasePermission.TYPE_BUILTIN);
                mSettings.mPermissions.put(perm.name, bp);
            }
            if (perm.gids != null) {
                bp.gids = appendInts(bp.gids, perm.gids);
            }
        }

        ArrayMap<String, String> libConfig = systemConfig.getSharedLibraries();
        for (int i=0; i<libConfig.size(); i++) {
            mSharedLibraries.put(libConfig.keyAt(i),
                    new SharedLibraryEntry(libConfig.valueAt(i), null));
        }

        mFoundPolicyFile = SELinuxMMAC.readInstallPolicy();

        mRestoredSettings = mSettings.readLPw(this, sUserManager.getUsers(false),
                mSdkVersion, mOnlyCore);

        String customResolverActivity = Resources.getSystem().getString(
                R.string.config_customResolverActivity);
        if (TextUtils.isEmpty(customResolverActivity)) {
            customResolverActivity = null;
        } else {
            mCustomResolverComponentName = ComponentName.unflattenFromString(
                    customResolverActivity);
        }

        long startTime = SystemClock.uptimeMillis();

        EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SYSTEM_SCAN_START,
                startTime);

        // Set flag to monitor and not change apk file paths when
        // scanning install directories.
        final int scanFlags = SCAN_NO_PATHS | SCAN_DEFER_DEX | SCAN_BOOTING;

        final HashSet<String> alreadyDexOpted = new HashSet<String>();

        /**
         * Add everything in the in the boot class path to the
         * list of process files because dexopt will have been run
         * if necessary during zygote startup.
         */
        final String bootClassPath = System.getenv("BOOTCLASSPATH");
        final String systemServerClassPath = System.getenv("SYSTEMSERVERCLASSPATH");

        if (bootClassPath != null) {
            String[] bootClassPathElements = splitString(bootClassPath, ':');
            for (String element : bootClassPathElements) {
                alreadyDexOpted.add(element);
            }
        } else {
            Slog.w(TAG, "No BOOTCLASSPATH found!");
        }

        if (systemServerClassPath != null) {
            String[] systemServerClassPathElements = splitString(systemServerClassPath, ':');
            for (String element : systemServerClassPathElements) {
                alreadyDexOpted.add(element);
            }
        } else {
            Slog.w(TAG, "No SYSTEMSERVERCLASSPATH found!");
        }

        boolean didDexOptLibraryOrTool = false;

        final List<String> allInstructionSets = getAllInstructionSets();
        final String[] dexCodeInstructionSets =
            getDexCodeInstructionSets(allInstructionSets.toArray(new
```

```
        String[allInstructionSets.size()]));

    /**
     * Ensure all external libraries have had dexopt run on them.
     */
    if (mSharedLibraries.size() > 0) {
        // NOTE: For now, we're compiling these system "shared libraries"
        // (and framework jars) into all available architectures. It's possible
        // to compile them only when we come across an app that uses them (there's
        // already logic for that in scanPackageLI) but that adds some complexity.
        for (String dexCodeInstructionSet : dexCodeInstructionSets) {
            for (SharedLibraryEntry libEntry : mSharedLibraries.values()) {
                final String lib = libEntry.path;
                if (lib == null) {
                    continue;
                }

                try {
                    byte dexoptRequired = DexFile.isDexOptNeededInternal(lib, null,
                    dexCodeInstructionSet, false);
                    if (dexoptRequired != DexFile.UP_TO_DATE) {
                        alreadyDexOpted.add(lib);

                        // The list of "shared libraries" we have at this point is
                        if (dexoptRequired == DexFile.DEXOPT_NEEDED) {
                            mInstaller.dexopt(lib, Process.SYSTEM_UID, true, dexCode
                            InstructionSet);
                        } else {
                            mInstaller.patchoat(lib, Process.SYSTEM_UID, true, dexCode
                            InstructionSet);
                        }
                        didDexOptLibraryOrTool = true;
                    }
                } catch (FileNotFoundException e) {
                    Slog.w(TAG, "Library not found: " + lib);
                } catch (IOException e) {
                    Slog.w(TAG, "Cannot dexopt " + lib + "; is it an APK or JAR? "
                            + e.getMessage());
                }
            }
        }
    }

    File frameworkDir = new File(Environment.getRootDirectory(), "framework");

    // Gross hack for now: we know this file doesn't contain any
    // code, so don't dexopt it to avoid the resulting log spew.
    alreadyDexOpted.add(frameworkDir.getPath() + "/framework-res.apk");

    // Gross hack for now: we know this file is only part of
    // the boot class path for art, so don't dexopt it to
    // avoid the resulting log spew.
    alreadyDexOpted.add(frameworkDir.getPath() + "/core-libart.jar");

    /**
     * And there are a number of commands implemented in Java, which
     * we currently need to do the dexopt on so that they can be
     * run from a non-root shell.
     */
    String[] frameworkFiles = frameworkDir.list();
    if (frameworkFiles != null) {
        // TODO: We could compile these only for the most preferred ABI. We should
        // by other system apps.
        for (String dexCodeInstructionSet : dexCodeInstructionSets) {
            for (int i=0; i<frameworkFiles.length; i++) {
                File libPath = new File(frameworkDir, frameworkFiles[i]);
                String path = libPath.getPath();
                // Skip the file if we already did it.
                if (alreadyDexOpted.contains(path)) {
                    continue;
                }
```

```
                // Skip the file if it is not a type we want to dexopt.
                if (!path.endsWith(".apk") && !path.endsWith(".jar")) {
                    continue;
                }
                try {
                    byte dexoptRequired = DexFile.isDexOptNeededInternal(path, null,
                    dexCodeInstructionSet, false);
                    if (dexoptRequired == DexFile.DEXOPT_NEEDED) {
                        mInstaller.dexopt(path, Process.SYSTEM_UID, true,
                        dexCodeInstructionSet);
                        didDexOptLibraryOrTool = true;
                    } else if (dexoptRequired == DexFile.PATCHOAT_NEEDED) {
                        mInstaller.patchoat(path, Process.SYSTEM_UID, true,
                        dexCodeInstructionSet);
                        didDexOptLibraryOrTool = true;
                    }
                } catch (FileNotFoundException e) {
                    Slog.w(TAG, "Jar not found: " + path);
                } catch (IOException e) {
                    Slog.w(TAG, "Exception reading jar: " + path, e);
                }
            }
        }
    }

    // Collect vendor overlay packages.
    // (Do this before scanning any apps.)
    // For security and version matching reason, only consider
    // overlay packages if they reside in VENDOR_OVERLAY_DIR.
    File vendorOverlayDir = new File(VENDOR_OVERLAY_DIR);
    scanDirLI(vendorOverlayDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags | SCAN_TRUSTED_OVERLAY, 0);

    // Find base frameworks (resource packages without code).
    scanDirLI(frameworkDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR
            | PackageParser.PARSE_IS_PRIVILEGED,
            scanFlags | SCAN_NO_DEX, 0);

    // Collected privileged system packages.
    final File privilegedAppDir = new File(Environment.getRootDirectory(),
    "priv-app");
    scanDirLI(privilegedAppDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR
            | PackageParser.PARSE_IS_PRIVILEGED, scanFlags, 0);

    // Collect ordinary system packages.
    final File systemAppDir = new File(Environment.getRootDirectory(), "app");
    scanDirLI(systemAppDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

    // Collect all vendor packages.
    File vendorAppDir = new File("/vendor/app");
    try {
        vendorAppDir = vendorAppDir.getCanonicalFile();
    } catch (IOException e) {
        // failed to look up canonical path, continue with original one
    }
    scanDirLI(vendorAppDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

    // Collect all OEM packages.
    final File oemAppDir = new File(Environment.getOemDirectory(), "app");
    scanDirLI(oemAppDir, PackageParser.PARSE_IS_SYSTEM
            | PackageParser.PARSE_IS_SYSTEM_DIR, scanFlags, 0);

    if (DEBUG_UPGRADE) Log.v(TAG, "Running installd update commands");
    mInstaller.moveFiles();

    // Prune any system packages that no longer exist.
    final List<String> possiblyDeletedUpdatedSystemApps = new ArrayList<String>();
```

```
        final ArrayMap<String, File> expectingBetter = new ArrayMap<>();
        if (!mOnlyCore) {
            Iterator<PackageSetting> psit = mSettings.mPackages.values().iterator();
            while (psit.hasNext()) {
                PackageSetting ps = psit.next();

                /*
                 * If this is not a system app, it can't be a
                 * disable system app.
                 */
                if ((ps.pkgFlags & ApplicationInfo.FLAG_SYSTEM) == 0) {
                    continue;
                }

                /*
                 * If the package is scanned, it's not erased.
                 */
                final PackageParser.Package scannedPkg = mPackages.get(ps.name);
                if (scannedPkg != null) {
                    /*
                     * If the system app is both scanned and in the
                     * disabled packages list, then it must have been
                     * added via OTA. Remove it from the currently
                     * scanned package so the previously user-installed
                     * application can be scanned.
                     */
                    if (mSettings.isDisabledSystemPackageLPr(ps.name)) {
                        logCriticalInfo(Log.WARN, "Expecting better updated system app for
"+ ps.name + "; removing system app.  Last known codePath="+ ps.codePathString + ",
installStatus=" + ps.installStatus+ ", versionCode=" + ps.versionCode + "; scanned
versionCode="+ scannedPkg.mVersionCode);
                        removePackageLI(ps, true);
                        expectingBetter.put(ps.name, ps.codePath);
                    }

                    continue;
                }

                if (!mSettings.isDisabledSystemPackageLPr(ps.name)) {
                    psit.remove();
                    logCriticalInfo(Log.WARN, "System package " + ps.name
                            + " no longer exists; wiping its data");
                    removeDataDirsLI(ps.name);
                } else {
                    final PackageSetting disabledPs = mSettings.getDisabledSystemPkgLPr
                    (ps.name);
                    if (disabledPs.codePath == null || !disabledPs.codePath.exists()) {
                        possiblyDeletedUpdatedSystemApps.add(ps.name);
                    }
                }
            }
        }

        //look for any incomplete package installations
        ArrayList<PackageSetting> deletePkgsList = mSettings.getListOfIncomplete
        InstallPackagesLPr();
        //clean up list
        for(int i = 0; i < deletePkgsList.size(); i++) {
            //clean up here
            cleanupInstallFailedPackage(deletePkgsList.get(i));
        }
        //delete tmp files
        deleteTempPackageFiles();

        // Remove any shared userIDs that have no associated packages
        mSettings.pruneSharedUsersLPw();

        if (!mOnlyCore) {
            EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_DATA_SCAN_START,
                    SystemClock.uptimeMillis());
            scanDirLI(mAppInstallDir, 0, scanFlags, 0);
```

```
scanDirLI(mDrmAppPrivateInstallDir, PackageParser.PARSE_FORWARD_LOCK,
scanFlags, 0);

/**
 * Remove disable package settings for any updated system
 * apps that were removed via an OTA. If they're not a
 * previously-updated app, remove them completely.
 * Otherwise, just revoke their system-level permissions.
 */
for (String deletedAppName : possiblyDeletedUpdatedSystemApps) {
    PackageParser.Package deletedPkg = mPackages.get(deletedAppName);
    mSettings.removeDisabledSystemPackageLPw(deletedAppName);

    String msg;
    if (deletedPkg == null) {
        msg = "Updated system package " + deletedAppName
                + " no longer exists; wiping its data";
        removeDataDirsLI(deletedAppName);
    } else {
        msg = "Updated system app + " + deletedAppName
                + " no longer present; removing system privileges for "
                + deletedAppName;

        deletedPkg.applicationInfo.flags &= ~ApplicationInfo.FLAG_SYSTEM;

        PackageSetting deletedPs = mSettings.mPackages.get(deletedAppName);
        deletedPs.pkgFlags &= ~ApplicationInfo.FLAG_SYSTEM;
    }
    logCriticalInfo(Log.WARN, msg);
}

/**
 * Make sure all system apps that we expected to appear on
 * the userdata partition actually showed up. If they never
 * appeared, crawl back and revive the system version.
 */
for (int i = 0; i < expectingBetter.size(); i++) {
    final String packageName = expectingBetter.keyAt(i);
    if (!mPackages.containsKey(packageName)) {
        final File scanFile = expectingBetter.valueAt(i);

        logCriticalInfo(Log.WARN, "Expected better " + packageName
                + " but never showed up; reverting to system");

        final int reparseFlags;
        if (FileUtils.contains(privilegedAppDir, scanFile)) {
            reparseFlags = PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR
                    | PackageParser.PARSE_IS_PRIVILEGED;
        } else if (FileUtils.contains(systemAppDir, scanFile)) {
            reparseFlags = PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR;
        } else if (FileUtils.contains(vendorAppDir, scanFile)) {
            reparseFlags = PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR;
        } else if (FileUtils.contains(oemAppDir, scanFile)) {
            reparseFlags = PackageParser.PARSE_IS_SYSTEM
                    | PackageParser.PARSE_IS_SYSTEM_DIR;
        } else {
            Slog.e(TAG, "Ignoring unexpected fallback path " + scanFile);
            continue;
        }

        mSettings.enableSystemPackageLPw(packageName);

        try {
            scanPackageLI(scanFile, reparseFlags, scanFlags, 0, null);
        } catch (PackageManagerException e) {
            Slog.e(TAG, "Failed to parse original system package: "
                    + e.getMessage());
```

```
                }
            }
        }
    }

    // Now that we know all of the shared libraries, update all clients to have
    // the correct library paths.
    updateAllSharedLibrariesLPw();

    for (SharedUserSetting setting : mSettings.getAllSharedUsersLPw()) {
        // NOTE: We ignore potential failures here during a system scan (like
        // the rest of the commands above) because there's precious little we
        // can do about it. A settings error is reported, though.
        adjustCpuAbisForSharedUserLPw(setting.packages, null /* scanned package */,
        false /* force dexopt */, false /* defer dexopt */);
    }

    // Now that we know all the packages we are keeping,
    // read and update their last usage times.
    mPackageUsage.readLP();

    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_SCAN_END,
            SystemClock.uptimeMillis());
    Slog.i(TAG, "Time to scan packages: "
            + ((SystemClock.uptimeMillis()-startTime)/1000f)
            + " seconds");

    // If the platform SDK has changed since the last time we booted,
    // we need to re-grant app permission to catch any new ones that
    // appear.  This is really a hack, and means that apps can in some
    // cases get permissions that the user didn't initially explicitly
    // allow...  it would be nice to have some better way to handle
    // this situation.
    final boolean regrantPermissions = mSettings.mInternalSdkPlatform
            != mSdkVersion;
    if (regrantPermissions) Slog.i(TAG, "Platform changed from "
            + mSettings.mInternalSdkPlatform + " to " + mSdkVersion
            + "; regranting permissions for internal storage");
    mSettings.mInternalSdkPlatform = mSdkVersion;

    updatePermissionsLPw(null, null, UPDATE_PERMISSIONS_ALL
            | (regrantPermissions
                    ? (UPDATE_PERMISSIONS_REPLACE_PKG|UPDATE_PERMISSIONS_REPLACE_ALL)
                    : 0));

    // If this is the first boot, and it is a normal boot, then
    // we need to initialize the default preferred apps.
    if (!mRestoredSettings && !onlyCore) {
        mSettings.readDefaultPreferredAppsLPw(this, 0);
    }

    // If this is first boot after an OTA, and a normal boot, then
    // we need to clear code cache directories.
    if (!Build.FINGERPRINT.equals(mSettings.mFingerprint) && !onlyCore) {
        Slog.i(TAG, "Build fingerprint changed; clearing code caches");
        for (String pkgName : mSettings.mPackages.keySet()) {
            deleteCodeCacheDirsLI(pkgName);
        }
        mSettings.mFingerprint = Build.FINGERPRINT;
    }

    // All the changes are done during package scanning.
    mSettings.updateInternalDatabaseVersion();

    // can downgrade to reader
    mSettings.writeLPr();

    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_PMS_READY,
            SystemClock.uptimeMillis());
```

```
            mRequiredVerifierPackage = getRequiredVerifierLPr();
        } // synchronized (mPackages)
        } // synchronized (mInstallLock)

        mInstallerService = new PackageInstallerService(context, this, mAppInstallDir);

        // Now after opening every single application zip, make sure they
        // are all flushed.  Not really needed, but keeps things nice and
        // tidy.
        Runtime.getRuntime().gc();
    }
```

对上述构造函数的具体说明如下所示。

（1）上述代码构造了 PMS 并加到 ServiceManager 中，这样其他的组件就可以用该服务了。

（2）接下来在 PMS 构造函数中调用 readLPw()来解析文件 packages.xml 以得到上次的安装信息。

（3）除了主线程外，PMS 还有 PackageManager 工作线程，主要被用在其他安装方式中，因为启动时没什么用户交互，所以，基本上不需要把工作交给后台。

（4）安装应用程序的权限信息被保存在“/etc/permissions”目录中，通过函数 readPermissions()进行读取，并将结果保存在 Settings 中的 mPermission 中。

（5）在启动 Android 时要扫描和安装现有 App。或许是为了防止 corrupted 信息的存在，或许是为了能在升级系统或更新系统属性后保持 App 也是 valid，也或许是因为有些信息（如 uid）必须是动态生成和调整的。总之，为了要还原现有 App 的安装信息，这些信息被放在/data/system/packages.xml 里，由 Settings 管理。另外/data/system/packages.list 记录了 App 的 uid 和数据路径等信息。readLPw()就是用于恢复这些信息，实现程序位于/frameworks/base/services/java/com/android/server/pm/Settings.java。readLPw()先打开 packages.xml，再通过 XmlPullParser 类来解析其内容，它会根据不同的 tag 调用相应的函数来读取信息。

11.4 扫描 APK 文件

接下来调用函数 scanDirLI 来扫描设备中如下目录中的 APK 文件：

- /system/framework；
- /system/app；
- /vendor/app；
- /data/app；
- /data/app-private。

函数在 frameworks/base/services/java/com/android/server/PackageManagerService.java 文件中定义，具体实现代码如下所示：

```
private void scanDirLI(File dir, int parseFlags, int scanFlags, long currentTime) {
    final File[] files = dir.listFiles();
    if (ArrayUtils.isEmpty(files)) {
        Log.d(TAG, "No files in app dir " + dir);
        return;
    }

    if (DEBUG_PACKAGE_SCANNING) {
        Log.d(TAG, "Scanning app dir " + dir + " scanFlags=" + scanFlags
                + " flags=0x" + Integer.toHexString(parseFlags));
    }

    for (File file : files) {
        final boolean isPackage = (isApkFile(file) || file.isDirectory())
                && !PackageInstallerService.isStageName(file.getName());
        if (!isPackage) {
            // Ignore entries which are not packages
            continue;
        }
        try {
```

```
            scanPackageLI(file, parseFlags | PackageParser.PARSE_MUST_BE_APK,
                    scanFlags, currentTime, null);
        } catch (PackageManagerException e) {
            Slog.w(TAG, "Failed to parse " + file + ": " + e.getMessage());

            // Delete invalid userdata apps
            if ((parseFlags & PackageParser.PARSE_IS_SYSTEM) == 0 &&
                    e.error == PackageManager.INSTALL_FAILED_INVALID_APK) {
                logCriticalInfo(Log.WARN, "Deleting invalid package at " + file);
                if (file.isDirectory()) {
                    FileUtils.deleteContents(file);
                }
                file.delete();
            }
        }
    }
}
```

11.5 解析并安装文件

在扫描设备目录中的每一个文件时，如果是以“.apk”作为后缀名，那么就调用函数scanPackageLI来对它进行解析和安装。函数scanPackageLI在frameworks/base/services/java/com/android/server/PackageManagerService.java文件中定义，具体实现代码如下所示：

```
private PackageParser.Package scanPackageLI(File scanFile, int parseFlags, int scanFlags,
long currentTime, UserHandle user) throws PackageManagerException {
    if (DEBUG_INSTALL) Slog.d(TAG, "Parsing: " + scanFile);
    parseFlags |= mDefParseFlags;
    PackageParser pp = new PackageParser();
    pp.setSeparateProcesses(mSeparateProcesses);
    pp.setOnlyCoreApps(mOnlyCore);
    pp.setDisplayMetrics(mMetrics);

    if ((scanFlags & SCAN_TRUSTED_OVERLAY) != 0) {
        parseFlags |= PackageParser.PARSE_TRUSTED_OVERLAY;
    }

    final PackageParser.Package pkg;
    try {
        pkg = pp.parsePackage(scanFile, parseFlags);
    } catch (PackageParserException e) {
        throw PackageManagerException.from(e);
    }

    PackageSetting ps = null;
    PackageSetting updatedPkg;
    // reader
    synchronized (mPackages) {
        // Look to see if we already know about this package.
        String oldName = mSettings.mRenamedPackages.get(pkg.packageName);
        if (pkg.mOriginalPackages != null && pkg.mOriginalPackages.contains(oldName)) {
            // This package has been renamed to its original name.  Let's
            // use that.
            ps = mSettings.peekPackageLPr(oldName);
        }
        // If there was no original package, see one for the real package name.
        if (ps == null) {
            ps = mSettings.peekPackageLPr(pkg.packageName);
        }
        // Check to see if this package could be hiding/updating a system
        // package.  Must look for it either under the original or real
        // package name depending on our state.
        updatedPkg = mSettings.getDisabledSystemPkgLPr(ps != null ? ps.name :
        pkg.packageName);
        if (DEBUG_INSTALL && updatedPkg != null) Slog.d(TAG, "updatedPkg = " + updatedPkg);
    }
    boolean updatedPkgBetter = false;
    // First check if this is a system package that may involve an update
```

```
    if (updatedPkg != null && (parseFlags&PackageParser.PARSE_IS_SYSTEM) != 0) {
        if (ps != null && !ps.codePath.equals(scanFile)) {
            // The path has changed from what was last scanned... check the
            // version of the new path against what we have stored to determine
            // what to do.
            if (DEBUG_INSTALL) Slog.d(TAG, "Path changing from " + ps.codePath);
            if (pkg.mVersionCode < ps.versionCode) {
                // The system package has been updated and the code path does not match
                // Ignore entry. Skip it.
                logCriticalInfo(Log.INFO, "Package " + ps.name + " at " + scanFile
                        + " ignored: updated version " + ps.versionCode
                        + " better than this " + pkg.mVersionCode);
                if (!updatedPkg.codePath.equals(scanFile)) {
                    Slog.w(PackageManagerService.TAG, "Code path for hidden system pkg :
"+ ps.name + " changing from " + updatedPkg.codePathString+ " to " + scanFile);
                    updatedPkg.codePath = scanFile;
                    updatedPkg.codePathString = scanFile.toString();
                    // This is the point at which we know that the system-disk APK
                    // for this package has moved during a reboot (e.g. due to an OTA),
                    // so we need to reevaluate it for privilege policy.
                    if (locationIsPrivileged(scanFile)) {
                        updatedPkg.pkgFlags |= ApplicationInfo.FLAG_PRIVILEGED;
                    }
                }
                updatedPkg.pkg = pkg;
                throw new PackageManagerException(INSTALL_FAILED_DUPLICATE_PACKAGE, null);
            } else {
                // The current app on the system partition is better than
                // what we have updated to on the data partition; switch
                // back to the system partition version.
                // At this point, its safely assumed that package installation for
                // apps in system partition will go through. If not there won't be a working
                // version of the app
                // writer
                synchronized (mPackages) {
                    // Just remove the loaded entries from package lists.
                    mPackages.remove(ps.name);
                }

                logCriticalInfo(Log.WARN, "Package " + ps.name + " at " + scanFile
                        + " reverting from " + ps.codePathString
                        + ": new version " + pkg.mVersionCode
                        + " better than installed " + ps.versionCode);

                InstallArgs args = createInstallArgsForExisting(packageFlagsToInstall
                Flags(ps),
                        ps.codePathString, ps.resourcePathString, ps.legacyNativeLibrary
                        PathString,
                        getAppDexInstructionSets(ps));
                synchronized (mInstallLock) {
                    args.cleanUpResourcesLI();
                }
                synchronized (mPackages) {
                    mSettings.enableSystemPackageLPw(ps.name);
                }
                updatedPkgBetter = true;
            }
        }
    }

    if (updatedPkg != null) {
        // An updated system app will not have the PARSE_IS_SYSTEM flag set
        // initially
        parseFlags |= PackageParser.PARSE_IS_SYSTEM;

        // An updated privileged app will not have the PARSE_IS_PRIVILEGED
        // flag set initially
        if ((updatedPkg.pkgFlags & ApplicationInfo.FLAG_PRIVILEGED) != 0) {
            parseFlags |= PackageParser.PARSE_IS_PRIVILEGED;
        }
    }
```

```
    // Verify certificates against what was last scanned
    collectCertificatesLI(pp, ps, pkg, scanFile, parseFlags);

    /*
     * A new system app appeared, but we already had a non-system one of the
     * same name installed earlier.
     */
    boolean shouldHideSystemApp = false;
    if (updatedPkg == null && ps != null
            &&(parseFlags & PackageParser.PARSE_IS_SYSTEM_DIR)!=0 && !isSystemApp(ps)) {
        /*
         * Check to make sure the signatures match first. If they don't,
         * wipe the installed application and its data.
         */
        if (compareSignatures(ps.signatures.mSignatures, pkg.mSignatures)
                != PackageManager.SIGNATURE_MATCH) {
            logCriticalInfo(Log.WARN, "Package " + ps.name + " appeared on system, but"+
            " signatures don't match existing userdata copy; removing");
            deletePackageLI(pkg.packageName, null, true, null, null, 0, null, false);
            ps = null;
        } else {
            /*
             * If the newly-added system app is an older version than the
             * already installed version, hide it. It will be scanned later
             * and re-added like an update.
             */
            if (pkg.mVersionCode < ps.versionCode) {
                shouldHideSystemApp = true;
                logCriticalInfo(Log.INFO, "Package " + ps.name + " appeared at " + scanFile+
" but new version " + pkg.mVersionCode + " better than installed "+ ps.versionCode + ";
hiding system");
            } else {
                /*
                 * The newly found system app is a newer version that the
                 * one previously installed. Simply remove the
                 * already-installed application and replace it with our own
                 * while keeping the application data.
                 */
                logCriticalInfo(Log.WARN, "Package " + ps.name + " at " + scanFile
                        + " reverting from " + ps.codePathString + ": new version "
                        + pkg.mVersionCode + " better than installed " + ps.versionCode);
                InstallArgs args = createInstallArgsForExisting(packageFlagsToInstall
                Flags(ps),
                        ps.codePathString, ps.resourcePathString, ps.legacyNativeLibrary
                        PathString,
                        getAppDexInstructionSets(ps));
                synchronized (mInstallLock) {
                    args.cleanUpResourcesLI();
                }
            }
        }
    }

    // The apk is forward locked (not public) if its code and resources
    // are kept in different files. (except for app in either system or
    // vendor path).
    // TODO grab this value from PackageSettings
    if ((parseFlags & PackageParser.PARSE_IS_SYSTEM_DIR) == 0) {
        if (ps != null && !ps.codePath.equals(ps.resourcePath)) {
            parseFlags |= PackageParser.PARSE_FORWARD_LOCK;
        }
    }

    // TODO: extend to support forward-locked splits
    String resourcePath = null;
    String baseResourcePath = null;
    if ((parseFlags & PackageParser.PARSE_FORWARD_LOCK) != 0 && !updatedPkgBetter) {
        if (ps != null && ps.resourcePathString != null) {
            resourcePath = ps.resourcePathString;
            baseResourcePath = ps.resourcePathString;
```

```
            } else {
                // Should not happen at all. Just log an error.
                Slog.e(TAG, "Resource path not set for pkg : " + pkg.packageName);
            }
        } else {
            resourcePath = pkg.codePath;
            baseResourcePath = pkg.baseCodePath;
        }

        // Set application objects path explicitly.
        pkg.applicationInfo.setCodePath(pkg.codePath);
        pkg.applicationInfo.setBaseCodePath(pkg.baseCodePath);
        pkg.applicationInfo.setSplitCodePaths(pkg.splitCodePaths);
        pkg.applicationInfo.setResourcePath(resourcePath);
        pkg.applicationInfo.setBaseResourcePath(baseResourcePath);
        pkg.applicationInfo.setSplitResourcePaths(pkg.splitCodePaths);

        // Note that we invoke the following method only if we are about to unpack an application
        PackageParser.Package scannedPkg = scanPackageLI(pkg, parseFlags, scanFlags
                | SCAN_UPDATE_SIGNATURE, currentTime, user);

        /*
         * If the system app should be overridden by a previously installed
         * data, hide the system app now and let the /data/app scan pick it up
         * again.
         */
        if (shouldHideSystemApp) {
            synchronized (mPackages) {
                /*
                 * We have to grant systems permissions before we hide, because
                 * grantPermissions will assume the package update is trying to
                 * expand its permissions.
                 */
                grantPermissionsLPw(pkg, true, pkg.packageName);
                mSettings.disableSystemPackageLPw(pkg.packageName);
            }
        }

        return scannedPkg;
    }
```

在上述代码中，首先会为安装的 APK 文件创建一个 PackageParser 实例，然后调用这个实例的 parsePackage 函数来对这个 APK 文件进行解析。在最后还会调用另外一个版本的 scanPackageLI 函数，将解析后得到的应用程序信息保存在 PackageManagerService 中。

函数 parsePackage 在 frameworks/base/core/java/android/content/pm/PackageParser.java 文件中定义，具体实现代码如下所示：

```
public Package parsePackage(File packageFile, int flags) throws PackageParserException {
    if (packageFile.isDirectory()) {
        return parseClusterPackage(packageFile, flags);
    } else {
        return parseMonolithicPackage(packageFile, flags);
    }
}
```

在上述代码中，调用函数 parseClusterPackage 和 parseMonolithicPackage 进行解析，其中前者实现单包分析，分析所有的应用程序包含在给定的目录，把它们当作一个单包进行处理。这也执行完整性检查，如要求相同的包的名称和版本的代码，则解析为一个单一的基础 APK 和独特的分裂的名字。函数 parseClusterPackage 的具体实现代码如下所示：

```
private Package parseClusterPackage(File packageDir, int flags) throws PackageParser
Exception {
    final PackageLite lite = parseClusterPackageLite(packageDir, 0);

    if (mOnlyCoreApps && !lite.coreApp) {
        throw new PackageParserException(INSTALL_PARSE_FAILED_MANIFEST_MALFORMED,
                "Not a coreApp: " + packageDir);
    }
```

```
    final AssetManager assets = new AssetManager();
    try {
        // Load the base and all splits into the AssetManager
        // so that resources can be overriden when parsing the manifests.
        loadApkIntoAssetManager(assets, lite.baseCodePath, flags);

        if (!ArrayUtils.isEmpty(lite.splitCodePaths)) {
            for (String path : lite.splitCodePaths) {
                loadApkIntoAssetManager(assets, path, flags);
            }
        }

        final File baseApk = new File(lite.baseCodePath);
        final Package pkg = parseBaseApk(baseApk, assets, flags);
        if (pkg == null) {
            throw new PackageParserException(INSTALL_PARSE_FAILED_NOT_APK,
                    "Failed to parse base APK: " + baseApk);
        }

        if (!ArrayUtils.isEmpty(lite.splitNames)) {
            final int num = lite.splitNames.length;
            pkg.splitNames = lite.splitNames;
            pkg.splitCodePaths = lite.splitCodePaths;
            pkg.splitFlags = new int[num];

            for (int i = 0; i < num; i++) {
                parseSplitApk(pkg, i, assets, flags);
            }
        }

        pkg.codePath = packageDir.getAbsolutePath();
        return pkg;
    } finally {
        IoUtils.closeQuietly(assets);
    }
}
```

而函数 parseMonolithicPackage 的功能是解析给出的 APK 文件，把它作为一个单一的整体包装。具体实现代码如下所示：

```
public Package parseMonolithicPackage(File apkFile, int flags) throws PackageParser
Exception {
    if (mOnlyCoreApps) {
        final PackageLite lite = parseMonolithicPackageLite(apkFile, flags);
        if (!lite.coreApp) {
            throw new PackageParserException(INSTALL_PARSE_FAILED_MANIFEST_MALFORMED,
                    "Not a coreApp: " + apkFile);
        }
    }

    final AssetManager assets = new AssetManager();
    try {
        final Package pkg = parseBaseApk(apkFile, assets, flags);
        pkg.codePath = apkFile.getAbsolutePath();
        return pkg;
    } finally {
        IoUtils.closeQuietly(assets);
    }
}
```

在函数 parseClusterPackage 和 parseMonolithicPackage 中都调用了函数 parseBaseApk，因为每一个 APK 文件都是一个归档文件，里面包含了 Android 应用程序的配置文件 AndroidManifest.xml，所以，通过函数 parseBaseApk 对这个配置文件就行解析。函数 parseBaseApk 的具体实现代码如下所示：

```
private Package parseBaseApk(File apkFile, AssetManager assets, int flags)
        throws PackageParserException {
    final String apkPath = apkFile.getAbsolutePath();
```

```
        mParseError = PackageManager.INSTALL_SUCCEEDED;
        mArchiveSourcePath = apkFile.getAbsolutePath();

        if (DEBUG_JAR) Slog.d(TAG, "Scanning base APK: " + apkPath);

        final int cookie = loadApkIntoAssetManager(assets, apkPath, flags);

        Resources res = null;
        XmlResourceParser parser = null;
        try {
            res = new Resources(assets, mMetrics, null);
            assets.setConfiguration(0, 0, null, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    Build.VERSION.RESOURCES_SDK_INT);
            parser = assets.openXmlResourceParser(cookie, ANDROID_MANIFEST_FILENAME);

            final String[] outError = new String[1];
            final Package pkg = parseBaseApk(res, parser, flags, outError);
            if (pkg == null) {
                throw new PackageParserException(mParseError,
                        apkPath + " (at " + parser.getPositionDescription() + "): " +
                        outError[0]);
            }

            pkg.baseCodePath = apkPath;
            pkg.mSignatures = null;

            return pkg;

        } catch (PackageParserException e) {
            throw e;
        } catch (Exception e) {
            throw new PackageParserException(INSTALL_PARSE_FAILED_UNEXPECTED_EXCEPTION,
                    "Failed to read manifest from " + apkPath, e);
        } finally {
            IoUtils.closeQuietly(parser);
        }
    }
```

在上述加粗代码中，ANDROID_MANIFEST_FILENAME 表示 AndroidManifest.xml。当从 APK 归档文件中得到这个配置文件后，则调用上述函数对这个应用程序进行解析。通过上述函数代码，实现了对 AndroidManifest.xml 文件中各个标签的解析工作。

在 Android 5.0 系统中，通过函数 parseBaseApplication 对 application 标签进行解析。函数 parseBaseApplication 在 frameworks/base/core/java/android/content/pm/PackageParser.java 文件中定义，具体实现代码如下所示：

```
    private boolean parseBaseApplication(Package owner, Resources res,
            XmlPullParser parser, AttributeSet attrs, int flags, String[] outError)
        throws XmlPullParserException, IOException {
        final ApplicationInfo ai = owner.applicationInfo;
        final String pkgName = owner.applicationInfo.packageName;

        TypedArray sa = res.obtainAttributes(attrs,
                com.android.internal.R.styleable.AndroidManifestApplication);

        String name = sa.getNonConfigurationString(
                com.android.internal.R.styleable.AndroidManifestApplication_name, 0);
        if (name != null) {
            ai.className = buildClassName(pkgName, name, outError);
            if (ai.className == null) {
                sa.recycle();
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }
        }

        String manageSpaceActivity = sa.getNonConfigurationString(
                com.android.internal.R.styleable.AndroidManifestApplication_manage
    SpaceActivity,
                Configuration.NATIVE_CONFIG_VERSION);
```

```
if (manageSpaceActivity != null) {
    ai.manageSpaceActivityName = buildClassName(pkgName, manageSpaceActivity,
            outError);
}

boolean allowBackup = sa.getBoolean(
        com.android.internal.R.styleable.AndroidManifestApplication_allowBackup,true);
if (allowBackup) {
    ai.flags |= ApplicationInfo.FLAG_ALLOW_BACKUP;

    // backupAgent, killAfterRestore, and restoreAnyVersion are only relevant
    // if backup is possible for the given application.
    String backupAgent = sa.getNonConfigurationString(
            com.android.internal.R.styleable.AndroidManifestApplication_backupAgent,
            Configuration.NATIVE_CONFIG_VERSION);
    if (backupAgent != null) {
        ai.backupAgentName = buildClassName(pkgName, backupAgent, outError);
        if (DEBUG_BACKUP) {
            Slog.v(TAG, "android:backupAgent = " + ai.backupAgentName
                    + " from " + pkgName + "+" + backupAgent);
        }

        if (sa.getBoolean(
                com.android.internal.R.styleable.AndroidManifestApplication_
                killAfterRestore,
                true)) {
            ai.flags |= ApplicationInfo.FLAG_KILL_AFTER_RESTORE;
        }
        if (sa.getBoolean(
                com.android.internal.R.styleable.AndroidManifestApplication_
                restoreAnyVersion,
                false)) {
            ai.flags |= ApplicationInfo.FLAG_RESTORE_ANY_VERSION;
        }
        if (sa.getBoolean(
                com.android.internal.R.styleable.AndroidManifestApplication_
                fullBackupOnly,
                false)) {
            ai.flags |= ApplicationInfo.FLAG_FULL_BACKUP_ONLY;
        }
    }
}

TypedValue v = sa.peekValue(
        com.android.internal.R.styleable.AndroidManifestApplication_label);
if (v != null && (ai.labelRes=v.resourceId) == 0) {
    ai.nonLocalizedLabel = v.coerceToString();
}

ai.icon = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_icon, 0);
ai.logo = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_logo, 0);
ai.banner = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_banner, 0);
ai.theme = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_theme, 0);
ai.descriptionRes = sa.getResourceId(
        com.android.internal.R.styleable.AndroidManifestApplication_description, 0);

if ((flags&PARSE_IS_SYSTEM) != 0) {
    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_persistent,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_PERSISTENT;
    }
}

if (sa.getBoolean(
        com.android.internal.R.styleable.AndroidManifestApplication_requiredForAllUsers,
        false)) {
```

```
        owner.mRequiredForAllUsers = true;
    }

    String restrictedAccountType = sa.getString(com.android.internal.R.styleable
            .AndroidManifestApplication_restrictedAccountType);
    if (restrictedAccountType != null && restrictedAccountType.length() > 0) {
        owner.mRestrictedAccountType = restrictedAccountType;
    }

    String requiredAccountType = sa.getString(com.android.internal.R.styleable
            .AndroidManifestApplication_requiredAccountType);
    if (requiredAccountType != null && requiredAccountType.length() > 0) {
        owner.mRequiredAccountType = requiredAccountType;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_debuggable,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_DEBUGGABLE;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_vmSafeMode,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_VM_SAFE_MODE;
    }

    owner.baseHardwareAccelerated = sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_hardware
            Accelerated,
            owner.applicationInfo.targetSdkVersion >= Build.VERSION_CODES.ICE_CREAM_
            SANDWICH);

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_hasCode,
            true)) {
        ai.flags |= ApplicationInfo.FLAG_HAS_CODE;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_
            allowTaskReparenting,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_ALLOW_TASK_REPARENTING;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_
            allowClearUserData,
            true)) {
        ai.flags |= ApplicationInfo.FLAG_ALLOW_CLEAR_USER_DATA;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_testOnly,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_TEST_ONLY;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_largeHeap,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_LARGE_HEAP;
    }

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_supportsRtl,
            false /* default is no RTL support*/)) {
        ai.flags |= ApplicationInfo.FLAG_SUPPORTS_RTL;
    }
```

```
if (sa.getBoolean(
        com.android.internal.R.styleable.AndroidManifestApplication_multiArch,
        false)) {
    ai.flags |= ApplicationInfo.FLAG_MULTIARCH;
}

String str;
str = sa.getNonConfigurationString(
        com.android.internal.R.styleable.AndroidManifestApplication_permission, 0);
ai.permission = (str != null && str.length() > 0) ? str.intern() : null;

if (owner.applicationInfo.targetSdkVersion >= Build.VERSION_CODES.FROYO) {
    str = sa.getNonConfigurationString(
            com.android.internal.R.styleable.AndroidManifestApplication_
            taskAffinity,
            Configuration.NATIVE_CONFIG_VERSION);
} else {
    // Some older apps have been seen to use a resource reference
    // here that on older builds was ignored (with a warning).  We
    // need to continue to do this for them so they don't break.
    str = sa.getNonResourceString(
            com.android.internal.R.styleable.AndroidManifestApplication_
            taskAffinity);
}
ai.taskAffinity = buildTaskAffinityName(ai.packageName, ai.packageName,
        str, outError);

if (outError[0] == null) {
    CharSequence pname;
    if (owner.applicationInfo.targetSdkVersion >= Build.VERSION_CODES.FROYO) {
        pname = sa.getNonConfigurationString(
                com.android.internal.R.styleable.AndroidManifestApplication_
                process,
                Configuration.NATIVE_CONFIG_VERSION);
    } else {
        // Some older apps have been seen to use a resource reference
        // here that on older builds was ignored (with a warning).  We
        // need to continue to do this for them so they don't break.
        pname = sa.getNonResourceString(
                com.android.internal.R.styleable.AndroidManifestApplication_
                process);
    }
    ai.processName = buildProcessName(ai.packageName, null, pname,
            flags, mSeparateProcesses, outError);

    ai.enabled = sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_enabled,true);

    if (sa.getBoolean(
            com.android.internal.R.styleable.AndroidManifestApplication_isGame,
            false)) {
        ai.flags |= ApplicationInfo.FLAG_IS_GAME;
    }

    if (false) {
        if (sa.getBoolean(
                com.android.internal.R.styleable.AndroidManifestApplication_
                cantSaveState,
                false)) {
            ai.flags |= ApplicationInfo.FLAG_CANT_SAVE_STATE;

            // A heavy-weight application can not be in a custom process.
            // We can do direct compare because we intern all strings.
            if (ai.processName != null && ai.processName != ai.packageName) {
                outError[0] = "cantSaveState applications can not use custom processes";
            }
        }
    }
}

ai.uiOptions = sa.getInt(
```

```
            com.android.internal.R.styleable.AndroidManifestApplication_uiOptions, 0);

    sa.recycle();

    if (outError[0] != null) {
        mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
        return false;
    }

    final int innerDepth = parser.getDepth();
    int type;
    while ((type = parser.next()) != XmlPullParser.END_DOCUMENT
            && (type != XmlPullParser.END_TAG || parser.getDepth() > innerDepth)) {
        if (type == XmlPullParser.END_TAG || type == XmlPullParser.TEXT) {
            continue;
        }

        String tagName = parser.getName();
        if (tagName.equals("activity")) {
            Activity a = parseActivity(owner, res, parser, attrs, flags, outError, false,
            owner.baseHardwareAccelerated);
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.activities.add(a);

        } else if (tagName.equals("receiver")) {
            Activity a = parseActivity(owner, res, parser, attrs, flags, outError, true,
            false);
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.receivers.add(a);

        } else if (tagName.equals("service")) {
            Service s = parseService(owner, res, parser, attrs, flags, outError);
            if (s == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.services.add(s);

        } else if (tagName.equals("provider")) {
            Provider p = parseProvider(owner, res, parser, attrs, flags, outError);
            if (p == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.providers.add(p);

        } else if (tagName.equals("activity-alias")) {
            Activity a = parseActivityAlias(owner, res, parser, attrs, flags, outError);
            if (a == null) {
                mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
                return false;
            }

            owner.activities.add(a);

        } else if (parser.getName().equals("meta-data")) {
            // note: application meta-data is stored off to the side, so it can
            // remain null in the primary copy (we like to avoid extra copies because
            // it can be large)
            if ((owner.mAppMetaData = parseMetaData(res, parser, attrs, owner.
            mAppMetaData, outError)) == null) {
```

```
            mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
            return false;
        }

    } else if (tagName.equals("library")) {
        sa = res.obtainAttributes(attrs,
                com.android.internal.R.styleable.AndroidManifestLibrary);

        // Note: don't allow this value to be a reference to a resource
        // that may change.
        String lname = sa.getNonResourceString(
                com.android.internal.R.styleable.AndroidManifestLibrary_name);

        sa.recycle();

        if (lname != null) {
            lname = lname.intern();
            if (!ArrayUtils.contains(owner.libraryNames, lname)) {
                owner.libraryNames = ArrayUtils.add(owner.libraryNames, lname);
            }
        }

        XmlUtils.skipCurrentTag(parser);

    } else if (tagName.equals("uses-library")) {
        sa = res.obtainAttributes(attrs,
                com.android.internal.R.styleable.AndroidManifestUsesLibrary);

        // Note: don't allow this value to be a reference to a resource
        // that may change.
        String lname = sa.getNonResourceString(
                com.android.internal.R.styleable.AndroidManifestUsesLibrary_name);
        boolean req = sa.getBoolean(
                com.android.internal.R.styleable.AndroidManifestUsesLibrary_
                required,
                true);

        sa.recycle();

        if (lname != null) {
            lname = lname.intern();
            if (req) {
                owner.usesLibraries = ArrayUtils.add(owner.usesLibraries, lname);
            } else {
                owner.usesOptionalLibraries = ArrayUtils.add(
                        owner.usesOptionalLibraries, lname);
            }
        }

        XmlUtils.skipCurrentTag(parser);

    } else if (tagName.equals("uses-package")) {
        // Dependencies for app installers; we don't currently try to
        // enforce this.
        XmlUtils.skipCurrentTag(parser);

    } else {
        if (!RIGID_PARSER) {
            Slog.w(TAG, "Unknown element under <application>: " + tagName
                    + " at " + mArchiveSourcePath + " "
                    + parser.getPositionDescription());
            XmlUtils.skipCurrentTag(parser);
            continue;
        } else {
            outError[0] = "Bad element under <application>: " + tagName;
            mParseError = PackageManager.INSTALL_PARSE_FAILED_MANIFEST_MALFORMED;
            return false;
        }
    }
}
```

```
        return true;
    }
```

通过上述函数代码，对文件 AndroidManifest.xml 中的 application 标签进行了解析。在现实中常用到的标签就有 activity、service、receiver 和 provider，各个标签的含义可以参考官方文档 http://developer.android.com/guide/topics/manifest/manifest-intro.html。

最后调用函数 scanPackageLI 把解析后得到的应用程序信息保存下来，此函数在 frameworks/base/services/java/com/android/server/PackageManagerService.java 文件中定义，主要实现代码如下所示：

```
    private PackageParser.Package scanPackageDirtyLI(PackageParser.Package pkg, int
parseFlags, int scanFlags, long currentTime, UserHandle user) throws PackageManagerException {
        final File scanFile = new File(pkg.codePath);
        if (pkg.applicationInfo.getCodePath() == null ||
                pkg.applicationInfo.getResourcePath() == null) {
            // Bail out. The resource and code paths haven't been set.
            throw new PackageManagerException(INSTALL_FAILED_INVALID_APK,
                    "Code and resource paths haven't been set correctly");
        }

        if ((parseFlags&PackageParser.PARSE_IS_SYSTEM) != 0) {
            pkg.applicationInfo.flags |= ApplicationInfo.FLAG_SYSTEM;
        } else {
            // Only allow system apps to be flagged as core apps.
            pkg.coreApp = false;
        }

        if ((parseFlags&PackageParser.PARSE_IS_PRIVILEGED) != 0) {
            pkg.applicationInfo.flags |= ApplicationInfo.FLAG_PRIVILEGED;
        }

        if (mCustomResolverComponentName != null &&
                mCustomResolverComponentName.getPackageName().equals(pkg.packageName))
{
            setUpCustomResolverActivity(pkg);
        }

        if (pkg.packageName.equals("android")) {
            synchronized (mPackages) {
                if (mAndroidApplication != null) {
                    Slog.w(TAG, "*************************************************");
                    Slog.w(TAG, "Core android package being redefined.  Skipping.");
                    Slog.w(TAG, " file=" + scanFile);
                    Slog.w(TAG, "*************************************************");
                    throw new PackageManagerException(INSTALL_FAILED_DUPLICATE_PACKAGE,
                            "Core android package being redefined.  Skipping.");
                }
            ......
        // writer
        synchronized (mPackages) {
            // We don't expect installation to fail beyond this point

            // Add the new setting to mSettings
            mSettings.insertPackageSettingLPw(pkgSetting, pkg);
            // Add the new setting to mPackages
            mPackages.put(pkg.applicationInfo.packageName, pkg);
            // Make sure we don't accidentally delete its data.
            final Iterator<PackageCleanItem> iter = mSettings.mPackagesToBeCleaned.
            iterator();
            while (iter.hasNext()) {
                PackageCleanItem item = iter.next();
                if (pkgName.equals(item.packageName)) {
                    iter.remove();
                }
            }

            // Take care of first install / last update times.
            if (currentTime != 0) {
                if (pkgSetting.firstInstallTime == 0) {
```

```
            pkgSetting.firstInstallTime = pkgSetting.lastUpdateTime = currentTime;
        } else if ((scanFlags&SCAN_UPDATE_TIME) != 0) {
            pkgSetting.lastUpdateTime = currentTime;
        }
    } else if (pkgSetting.firstInstallTime == 0) {
        // We need *something*.  Take time time stamp of the file.
        pkgSetting.firstInstallTime = pkgSetting.lastUpdateTime = scanFileTime;
    } else if ((parseFlags&PackageParser.PARSE_IS_SYSTEM_DIR) != 0) {
        if (scanFileTime != pkgSetting.timeStamp) {
            // A package on the system image has changed; consider this
            // to be an update.
            pkgSetting.lastUpdateTime = scanFileTime;
        }
    }
    ......
    int N = pkg.providers.size();
    StringBuilder r = null;
    int i;
    for (i=0; i<N; i++) {
        PackageParser.Provider p = pkg.providers.get(i);
        p.info.processName = fixProcessName(pkg.applicationInfo.processName,
                p.info.processName, pkg.applicationInfo.uid);
        mProviders.addProvider(p);
        p.syncable = p.info.isSyncable;
        if (p.info.authority != null) {
            String names[] = p.info.authority.split(";");
            p.info.authority = null;
            for (int j = 0; j < names.length; j++) {
                if (j == 1 && p.syncable) {
                    // We only want the first authority for a provider to possibly be
                    // syncable, so if we already added this provider using a different
                    // authority clear the syncable flag. We copy the provider before
                    // changing it because the mProviders object contains a reference
                    // to a provider that we don't want to change.
                    // Only do this for the second authority since the resulting provider
                    // object can be the same for all future authorities for this provider.
                    p = new PackageParser.Provider(p);
                    p.syncable = false;
                }
                if (!mProvidersByAuthority.containsKey(names[j])) {
                    mProvidersByAuthority.put(names[j], p);
                    if (p.info.authority == null) {
                        p.info.authority = names[j];
                    } else {
                        p.info.authority = p.info.authority + ";" + names[j];
                    }
                    if (DEBUG_PACKAGE_SCANNING) {
                        if ((parseFlags & PackageParser.PARSE_CHATTY) != 0)
                            Log.d(TAG, "Registered content provider: " + names[j]
                                    + ", className = " + p.info.name + ", isSyncable
                                    = "+ p.info.isSyncable);
                    }
                } else {
                    PackageParser.Provider other = mProvidersByAuthority.get(names
                    [j]);
                    Slog.w(TAG, "Skipping provider name " + names[j] +
                            " (in package " + pkg.applicationInfo.packageName +
                            "): name already used by "
                            + ((other != null && other.getComponentName() != null)
                                ? other.getComponentName().getPackageName() : "?"));
                }
            }
        }
        if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
            if (r == null) {
                r = new StringBuilder(256);
            } else {
                r.append(' ');
            }
            r.append(p.info.name);
        }
```

```
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Providers: " + r);
        }

        N = pkg.services.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.Service s = pkg.services.get(i);
            s.info.processName = fixProcessName(pkg.applicationInfo.processName,
                    s.info.processName, pkg.applicationInfo.uid);
            mServices.addService(s);
            if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(s.info.name);
            }
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Services: " + r);
        }

        N = pkg.receivers.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.Activity a = pkg.receivers.get(i);
            a.info.processName = fixProcessName(pkg.applicationInfo.processName,
                    a.info.processName, pkg.applicationInfo.uid);
            mReceivers.addActivity(a, "receiver");
            if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(a.info.name);
            }
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Receivers: " + r);
        }

        N = pkg.activities.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.Activity a = pkg.activities.get(i);
            a.info.processName = fixProcessName(pkg.applicationInfo.processName,
                    a.info.processName, pkg.applicationInfo.uid);
            mActivities.addActivity(a, "activity");
            if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(a.info.name);
            }
        }
        if (r != null) {
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Activities: " + r);
        }

        N = pkg.permissionGroups.size();
        r = null;
        for (i=0; i<N; i++) {
            PackageParser.PermissionGroup pg = pkg.permissionGroups.get(i);
            PackageParser.PermissionGroup cur = mPermissionGroups.get(pg.info.name);
            if (cur == null) {
```

```
            mPermissionGroups.put(pg.info.name, pg);
            if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append(pg.info.name);
            }
        } else {
            Slog.w(TAG, "Permission group " + pg.info.name + " from package "
                    + pg.info.packageName + " ignored: original from "
                    + cur.info.packageName);
            if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                if (r == null) {
                    r = new StringBuilder(256);
                } else {
                    r.append(' ');
                }
                r.append("DUP:");
                r.append(pg.info.name);
            }
        }
    }
    if (r != null) {
        if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Permission Groups: " + r);
    }

    N = pkg.permissions.size();
    r = null;
    for (i=0; i<N; i++) {
        PackageParser.Permission p = pkg.permissions.get(i);
        HashMap<String, BasePermission> permissionMap =
                p.tree ? mSettings.mPermissionTrees
                : mSettings.mPermissions;
        p.group = mPermissionGroups.get(p.info.group);
        if (p.info.group == null || p.group != null) {
            BasePermission bp = permissionMap.get(p.info.name);

            // Allow system apps to redefine non-system permissions
            if (bp != null && !Objects.equals(bp.sourcePackage, p.info.packageName)) {
                final boolean currentOwnerIsSystem = (bp.perm != null
                        && isSystemApp(bp.perm.owner));
                if (isSystemApp(p.owner)) {
                    if (bp.type == BasePermission.TYPE_BUILTIN && bp.perm == null) {
                        // It's a built-in permission and no owner, take ownership now
                        bp.packageSetting = pkgSetting;
                        bp.perm = p;
                        bp.uid = pkg.applicationInfo.uid;
                        bp.sourcePackage = p.info.packageName;
                    } else if (!currentOwnerIsSystem) {
                        String msg = "New decl " + p.owner + " of permission  "
                                + p.info.name + " is system; overriding " +
                                bp.sourcePackage;
                        reportSettingsProblem(Log.WARN, msg);
                        bp = null;
                    }
                }
            }

            if (bp == null) {
                bp = new BasePermission(p.info.name, p.info.packageName,
                        BasePermission.TYPE_NORMAL);
                permissionMap.put(p.info.name, bp);
            }

            if (bp.perm == null) {
                if (bp.sourcePackage == null
                        || bp.sourcePackage.equals(p.info.packageName)) {
                    BasePermission tree = findPermissionTreeLP(p.info.name);
                    if (tree == null
```

```
                    || tree.sourcePackage.equals(p.info.packageName)) {
                bp.packageSetting = pkgSetting;
                bp.perm = p;
                bp.uid = pkg.applicationInfo.uid;
                bp.sourcePackage = p.info.packageName;
                if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
                    if (r == null) {
                        r = new StringBuilder(256);
                    } else {
                        r.append(' ');
                    }
                    r.append(p.info.name);
                }
            } else {
                Slog.w(TAG, "Permission " + p.info.name + " from package "
                        + p.info.packageName + " ignored: base tree "
                        + tree.name + " is from package "
                        + tree.sourcePackage);
            }
        } else {
            Slog.w(TAG, "Permission " + p.info.name + " from package "
                    + p.info.packageName + " ignored: original from "
                    + bp.sourcePackage);
        }
    } else if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
        if (r == null) {
            r = new StringBuilder(256);
        } else {
            r.append(' ');
        }
        r.append("DUP:");
        r.append(p.info.name);
    }
    if (bp.perm == p) {
        bp.protectionLevel = p.info.protectionLevel;
    }
} else {
    Slog.w(TAG, "Permission " + p.info.name + " from package "
            + p.info.packageName + " ignored: no group "
            + p.group);
}
}
if (r != null) {
    if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Permissions: " + r);
}

N = pkg.instrumentation.size();
r = null;
for (i=0; i<N; i++) {
    PackageParser.Instrumentation a = pkg.instrumentation.get(i);
    a.info.packageName = pkg.applicationInfo.packageName;
    a.info.sourceDir = pkg.applicationInfo.sourceDir;
    a.info.publicSourceDir = pkg.applicationInfo.publicSourceDir;
    a.info.splitSourceDirs = pkg.applicationInfo.splitSourceDirs;
    a.info.splitPublicSourceDirs = pkg.applicationInfo.splitPublicSourceDirs;
    a.info.dataDir = pkg.applicationInfo.dataDir;

    // TODO: Update instrumentation.nativeLibraryDir as well ? Does it
    // need other information about the application, like the ABI and what not ?
    a.info.nativeLibraryDir = pkg.applicationInfo.nativeLibraryDir;
    mInstrumentation.put(a.getComponentName(), a);
    if ((parseFlags&PackageParser.PARSE_CHATTY) != 0) {
        if (r == null) {
            r = new StringBuilder(256);
        } else {
            r.append(' ');
        }
        r.append(a.info.name);
    }
}
if (r != null) {
```

```
            if (DEBUG_PACKAGE_SCANNING) Log.d(TAG, "  Instrumentation: " + r);
        }

        if (pkg.protectedBroadcasts != null) {
            N = pkg.protectedBroadcasts.size();
            for (i=0; i<N; i++) {
                mProtectedBroadcasts.add(pkg.protectedBroadcasts.get(i));
            }
        }

        pkgSetting.setTimeStamp(scanFileTime);

        // Create idmap files for pairs of (packages, overlay packages).
        // Note: "android", ie framework-res.apk, is handled by native layers.
        if (pkg.mOverlayTarget != null) {
            // This is an overlay package.
            if (pkg.mOverlayTarget != null && !pkg.mOverlayTarget.equals("android")) {
                if (!mOverlays.containsKey(pkg.mOverlayTarget)) {
                    mOverlays.put(pkg.mOverlayTarget,
                            new HashMap<String, PackageParser.Package>());
                }
                HashMap<String, PackageParser.Package> map = mOverlays.get
                (pkg.mOverlayTarget);
                map.put(pkg.packageName, pkg);
                PackageParser.Package orig = mPackages.get(pkg.mOverlayTarget);
                if (orig != null && !createIdmapForPackagePairLI(orig, pkg)) {
                    throw new PackageManagerException(INSTALL_FAILED_UPDATE_
                    INCOMPATIBLE,
                            "scanPackageLI failed to createIdmap");
                }
            }
        } else if (mOverlays.containsKey(pkg.packageName) &&
                !pkg.packageName.equals("android")) {
            // This is a regular package, with one or more known overlay packages.
            createIdmapsForPackageLI(pkg);
        }
    }

    return pkg;
}
```

通过上述函数代码，把前面解析应用程序得到的 package、provider、service、receiver 和 activity 等信息保存在 PackageManagerService 服务中。这样，在启动 Android 系统时，安装应用程序的过程全部介绍完毕。但是，这些应用程序只是相当于在 PackageManagerService 服务注册好了，如果我们想要在 Android 桌面上看到这些应用程序，还需要有一个 Home 应用程序，负责从 PackageManagerService 服务中把这些安装好的应用程序取出来，并以友好的方式在桌面上展现出来，例如，以快捷图标的形式呈现出来。

11.6 启动系统默认 Home 应用程序 Launcher

在 Android 系统中，Home 应用程序 Launcher 由 ActivityManagerService 启动，而 ActivityManagerService 和 PackageManagerService 都是在开机时由 SystemServer 组件启动的。SystemServer 组件首先启动 PackageManagerServic 来安装系统的应用程序，当在系统中安装应用程序后，SystemServer 组件接下来会通过 ActivityManagerService 来启动 Home 应用程序 Launcher。在启动 Launcher 时会通过 PackageManagerServic 把系统中已经安装好的应用程序以快捷图标的形式展示在桌面上，这样用户就可以使用这些应用程序了。

11.6.1 设置系统进程

当将系统中应用程序的所有信息都保存在 PackageManagerService 中，然后 Home 应用程序 Launcher 启动起来后，就会把 PackageManagerService 中的应用程序信息取出来，然后以快捷图标

的形式展示在桌面上。在上述过程中会首先调用函数 setSystemProcess 设置系统进程，此函数在文件 frameworks\base\services\core\java\com\android\server\am\ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public void setSystemProcess() {
    try {
        ServiceManager.addService(Context.ACTIVITY_SERVICE, this, true);
        ServiceManager.addService(ProcessStats.SERVICE_NAME, mProcessStats);
        ServiceManager.addService("meminfo", new MemBinder(this));
        ServiceManager.addService("gfxinfo", new GraphicsBinder(this));
        ServiceManager.addService("dbinfo", new DbBinder(this));
        if (MONITOR_CPU_USAGE) {
            ServiceManager.addService("cpuinfo", new CpuBinder(this));
        }
        ServiceManager.addService("permission", new PermissionController(this));

        ApplicationInfo info = mContext.getPackageManager().getApplicationInfo(
                "android", STOCK_PM_FLAGS);
        mSystemThread.installSystemApplicationInfo(info, getClass().getClassLoader());

        synchronized (this) {
            ProcessRecord app = newProcessRecordLocked(info, info.processName, false, 0);
            app.persistent = true;
            app.pid = MY_PID;
            app.maxAdj = ProcessList.SYSTEM_ADJ;
            app.makeActive(mSystemThread.getApplicationThread(), mProcessStats);
            mProcessNames.put(app.processName, app.uid, app);
            synchronized (mPidsSelfLocked) {
                mPidsSelfLocked.put(app.pid, app);
            }
            updateLruProcessLocked(app, false, null);
            updateOomAdjLocked();
        }
    } catch (PackageManager.NameNotFoundException e) {
        throw new RuntimeException(
                "Unable to find android system package", e);
    }
}
```

在上述代码中，首先将 ActivityManagerService 实例添加到 ServiceManager 中去托管，这样在其他地方就可以通过 ServiceManager.getService 接口来访问这个全局唯一的 ActivityManagerService 实例了。然后通过调用函数 mSystemThread.installSystemApplicationInfo 来把应用程序框架层下面的 Android 包加载进来，这里的 mSystemThread 是一个 ActivityThread 类型的实例变量。

11.6.2　启动 Home 应用程序

函数 systemReady 是在 ServerThread.run 函数将系统中的一系列服务都初始化完之后才被调用，此函数在文件 frameworks\base\services\core\java\com\android\server\am\ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public void systemReady(final Runnable goingCallback) {
    synchronized(this) {
        if (mSystemReady) {
            // If we're done calling all the receivers, run the next "boot phase" passed in
            // by the SystemServer
            if (goingCallback != null) {
                goingCallback.run();
            }
            return;
        }

        // Make sure we have the current profile info, since it is needed for
        // security checks.
        updateCurrentProfileIdsLocked();

        if (mRecentTasks == null) {
            mRecentTasks = mTaskPersister.restoreTasksLocked();
```

```
            if (!mRecentTasks.isEmpty()) {
                mStackSupervisor.createStackForRestoredTaskHistory(mRecentTasks);
            }
            cleanupRecentTasksLocked(UserHandle.USER_ALL);
            mTaskPersister.startPersisting();
        }

        // Check to see if there are any update receivers to run.
        if (!mDidUpdate) {
            if (mWaitingUpdate) {
                return;
            }
            final ArrayList<ComponentName> doneReceivers = new ArrayList<ComponentName>();
            mWaitingUpdate = deliverPreBootCompleted(new Runnable() {
                public void run() {
                    synchronized (ActivityManagerService.this) {
                        mDidUpdate = true;
                    }
                    writeLastDonePreBootReceivers(doneReceivers);
                    showBootMessage(mContext.getText(
                            R.string.android_upgrading_complete),
                            false);
                    systemReady(goingCallback);
                }
            }, doneReceivers, UserHandle.USER_OWNER);

            if (mWaitingUpdate) {
                return;
            }
            mDidUpdate = true;
        }

        mAppOpsService.systemReady();
        mSystemReady = true;
    }

    ArrayList<ProcessRecord> procsToKill = null;
    synchronized(mPidsSelfLocked) {
        for (int i=mPidsSelfLocked.size()-1; i>=0; i--) {
            ProcessRecord proc = mPidsSelfLocked.valueAt(i);
            if (!isAllowedWhileBooting(proc.info)){
                if (procsToKill == null) {
                    procsToKill = new ArrayList<ProcessRecord>();
                }
                procsToKill.add(proc);
            }
        }
    }

    synchronized(this) {
        if (procsToKill != null) {
            for (int i=procsToKill.size()-1; i>=0; i--) {
                ProcessRecord proc = procsToKill.get(i);
                Slog.i(TAG, "Removing system update proc: " + proc);
                removeProcessLocked(proc, true, false, "system update done");
            }
        }

        // Now that we have cleaned up any update processes, we
        // are ready to start launching real processes and know that
        // we won't trample on them any more.
        mProcessesReady = true;
    }

    Slog.i(TAG, "System now ready");
    EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_AMS_READY,
        SystemClock.uptimeMillis());

    synchronized(this) {
        // Make sure we have no pre-ready processes sitting around.
```

```
    if (mFactoryTest == FactoryTest.FACTORY_TEST_LOW_LEVEL) {
        ResolveInfo ri = mContext.getPackageManager()
                .resolveActivity(new Intent(Intent.ACTION_FACTORY_TEST),
                        STOCK_PM_FLAGS);
        CharSequence errorMsg = null;
        if (ri != null) {
            ActivityInfo ai = ri.activityInfo;
            ApplicationInfo app = ai.applicationInfo;
            if ((app.flags&ApplicationInfo.FLAG_SYSTEM) != 0) {
                mTopAction = Intent.ACTION_FACTORY_TEST;
                mTopData = null;
                mTopComponent = new ComponentName(app.packageName,
                        ai.name);
            } else {
                errorMsg = mContext.getResources().getText(
                        com.android.internal.R.string.factorytest_not_system);
            }
        } else {
            errorMsg = mContext.getResources().getText(
                    com.android.internal.R.string.factorytest_no_action);
        }
        if (errorMsg != null) {
            mTopAction = null;
            mTopData = null;
            mTopComponent = null;
            Message msg = Message.obtain();
            msg.what = SHOW_FACTORY_ERROR_MSG;
            msg.getData().putCharSequence("msg", errorMsg);
            mHandler.sendMessage(msg);
        }
    }
}

retrieveSettings();
loadResourcesOnSystemReady();

synchronized (this) {
    readGrantedUriPermissionsLocked();
}

if (goingCallback != null) goingCallback.run();

mBatteryStatsService.noteEvent(BatteryStats.HistoryItem.EVENT_USER_RUNNING_START,
        Integer.toString(mCurrentUserId), mCurrentUserId);
mBatteryStatsService.noteEvent(BatteryStats.HistoryItem.EVENT_USER_FOREGROUND_START,
        Integer.toString(mCurrentUserId), mCurrentUserId);
mSystemServiceManager.startUser(mCurrentUserId);

synchronized (this) {
    if (mFactoryTest != FactoryTest.FACTORY_TEST_LOW_LEVEL) {
        try {
            List apps = AppGlobals.getPackageManager().
                getPersistentApplications(STOCK_PM_FLAGS);
            if (apps != null) {
                int N = apps.size();
                int i;
                for (i=0; i<N; i++) {
                    ApplicationInfo info
                        = (ApplicationInfo)apps.get(i);
                    if (info != null &&
                            !info.packageName.equals("android")) {
                        addAppLocked(info, false, null /* ABI override */);
                    }
                }
            }
        } catch (RemoteException ex) {
            // pm is in same process, this will never happen.
        }
    }

    // Start up initial activity.
```

```
        mBooting = true;
        startHomeActivityLocked(mCurrentUserId);

        try {
            if (AppGlobals.getPackageManager().hasSystemUidErrors()) {
                Message msg = Message.obtain();
                msg.what = SHOW_UID_ERROR_MSG;
                mHandler.sendMessage(msg);
            }
        } catch (RemoteException e) {
        }

        long ident = Binder.clearCallingIdentity();
        try {
            Intent intent = new Intent(Intent.ACTION_USER_STARTED);
            intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY
                    | Intent.FLAG_RECEIVER_FOREGROUND);
            intent.putExtra(Intent.EXTRA_USER_HANDLE, mCurrentUserId);
            broadcastIntentLocked(null, null, intent,
                    null, null, 0, null, null, null, AppOpsManager.OP_NONE,
                    false, false, MY_PID, Process.SYSTEM_UID, mCurrentUserId);
            intent = new Intent(Intent.ACTION_USER_STARTING);
            intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
            intent.putExtra(Intent.EXTRA_USER_HANDLE, mCurrentUserId);
            broadcastIntentLocked(null, null, intent,
                    null, new IIntentReceiver.Stub() {
                        @Override
                        public void performReceive(Intent intent, int resultCode, String
data, Bundle extras, boolean ordered, boolean sticky, int sendingUser)
                                throws RemoteException {
                        }
                    }, 0, null, null,
                    INTERACT_ACROSS_USERS, AppOpsManager.OP_NONE,
                    true, false, MY_PID, Process.SYSTEM_UID, UserHandle.USER_ALL);
        } catch (Throwable t) {
            Slog.wtf(TAG, "Failed sending first user broadcasts", t);
        } finally {
            Binder.restoreCallingIdentity(ident);
        }
        mStackSupervisor.resumeTopActivitiesLocked();
        sendUserSwitchBroadcastsLocked(-1, mCurrentUserId);
    }
}
```

在上述代码中，调用函数 startHomeActivityLocked 启动了 Home 应用程序。函数 startHome ActivityLocked 在文件 frameworks\base\services\core\java\com\android\server\am\ActivityManager Service.java 中定义，具体实现代码如下所示：

```
boolean startHomeActivityLocked(int userId) {
    if (mFactoryTest == FactoryTest.FACTORY_TEST_LOW_LEVEL
            && mTopAction == null) {
        // We are running in factory test mode, but unable to find
        // the factory test app, so just sit around displaying the
        // error message and don't try to start anything.
        return false;
    }
    Intent intent = getHomeIntent();
    ActivityInfo aInfo =
        resolveActivityInfo(intent, STOCK_PM_FLAGS, userId);
    if (aInfo != null) {
        intent.setComponent(new ComponentName(
                aInfo.applicationInfo.packageName, aInfo.name));
        // Don't do this if the home app is currently being
        // instrumented.
        aInfo = new ActivityInfo(aInfo);
        aInfo.applicationInfo = getAppInfoForUser(aInfo.applicationInfo, userId);
        ProcessRecord app = getProcessRecordLocked(aInfo.processName,
                aInfo.applicationInfo.uid, true);
        if (app == null || app.instrumentationClass == null) {
            intent.setFlags(intent.getFlags() | Intent.FLAG_ACTIVITY_NEW_TASK);
```

```
            mStackSupervisor.startHomeActivity(intent, aInfo);
        }
    }

    return true;
}
```

在上述代码中，调用了函数 getHomeIntent 来创建 Intent，此函数的具体实现代码如下所示：

```
Intent getHomeIntent() {
    Intent intent = new Intent(mTopAction, mTopData != null ? Uri.parse(mTopData) : null);
    intent.setComponent(mTopComponent);
    if (mFactoryTest != FactoryTest.FACTORY_TEST_LOW_LEVEL) {
        intent.addCategory(Intent.CATEGORY_HOME);
    }
    return intent;
}
```

在上述代码中，首先创建了一个 CATEGORY_HOME 类型的 Intent，然后向中 PackageManager Service 查询 Category 类型为 HOME 的 Activity。在此假设只有系统自带的 Launcher 应用程序注册了 HOME 类型的 Activity，文件 packages/apps/Launcher2/AndroidManifest.xml 的代码如下所示：

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.launcher">

    <original-package android:name="com.android.launcher2" />

    <permission
        android:name="com.android.launcher.permission.PRELOAD_WORKSPACE"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="signatureOrSystem" />
    <permission
        android:name="com.android.launcher.permission.INSTALL_SHORTCUT"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="dangerous"
        android:label="@string/permlab_install_shortcut"
        android:description="@string/permdesc_install_shortcut" />
    <permission
        android:name="com.android.launcher.permission.UNINSTALL_SHORTCUT"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="dangerous"
        android:label="@string/permlab_uninstall_shortcut"
        android:description="@string/permdesc_uninstall_shortcut"/>
    <permission
        android:name="com.android.launcher.permission.READ_SETTINGS"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="normal"
        android:label="@string/permlab_read_settings"
        android:description="@string/permdesc_read_settings"/>
    <permission
        android:name="com.android.launcher.permission.WRITE_SETTINGS"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="signatureOrSystem"
        android:label="@string/permlab_write_settings"
        android:description="@string/permdesc_write_settings"/>

    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.SET_WALLPAPER" />
    <uses-permission android:name="android.permission.SET_WALLPAPER_HINTS" />
    <uses-permission android:name="android.permission.VIBRATE" />
    <uses-permission android:name="android.permission.BIND_APPWIDGET" />
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <uses-permission android:name="com.android.launcher.permission.READ_SETTINGS" />
    <uses-permission android:name="com.android.launcher.permission.WRITE_SETTINGS" />

    <application
        android:name="com.android.launcher2.LauncherApplication"
        android:label="@string/application_name"
        android:icon="@mipmap/ic_launcher_home"
        android:hardwareAccelerated="true"
```

```
android:largeHeap="@bool/config_largeHeap"
android:supportsRtl="true">
<activity
    android:name="com.android.launcher2.Launcher"
    android:launchMode="singleTask"
    android:clearTaskOnLaunch="true"
    android:stateNotNeeded="true"
    android:resumeWhilePausing="true"
    android:theme="@style/Theme"
    android:windowSoftInputMode="adjustPan"
    android:screenOrientation="nosensor">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.HOME" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.MONKEY"/>
    </intent-filter>
</activity>

<activity
    android:name="com.android.launcher2.WallpaperChooser"
    android:theme="@style/Theme.WallpaperPicker"
    android:label="@string/pick_wallpaper"
    android:icon="@mipmap/ic_launcher_wallpaper"
    android:finishOnCloseSystemDialogs="true"
    android:process=":wallpaper_chooser">
    <intent-filter>
        <action android:name="android.intent.action.SET_WALLPAPER" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
    <meta-data android:name="android.wallpaper.preview"
        android:resource="@xml/wallpaper_picker_preview" />
</activity>

<!-- Intent received used to prepopulate the default workspace. -->
<receiver
    android:name="com.android.launcher2.PreloadReceiver"
    android:permission="com.android.launcher.permission.PRELOAD_WORKSPACE">
    <intent-filter>
        <action android:name="com.android.launcher.action.PRELOAD_WORKSPACE" />
    </intent-filter>
</receiver>

<!-- Intent received used to install shortcuts from other applications -->
<receiver
    android:name="com.android.launcher2.InstallShortcutReceiver"
    android:permission="com.android.launcher.permission.INSTALL_SHORTCUT">
    <intent-filter>
        <action android:name="com.android.launcher.action.INSTALL_SHORTCUT" />
    </intent-filter>
</receiver>

<!-- Intent received used to uninstall shortcuts from other applications -->
<receiver
    android:name="com.android.launcher2.UninstallShortcutReceiver"
    android:permission="com.android.launcher.permission.UNINSTALL_SHORTCUT">
    <intent-filter>
        <action android:name="com.android.launcher.action.UNINSTALL_SHORTCUT" />
    </intent-filter>
</receiver>

<!-- New user initialization; set up initial wallpaper -->
<receiver
    android:name="com.android.launcher2.UserInitializeReceiver"
    android:exported="false">
    <intent-filter>
        <action android:name="android.intent.action.USER_INITIALIZE" />
    </intent-filter>
</receiver>

<receiver android:name="com.android.launcher2.PackageChangedReceiver" >
```

```
            <intent-filter>
                <action android:name="android.intent.action.PACKAGE_CHANGED"/>
                <action android:name="android.intent.action.PACKAGE_REPLACED"/>
                <action android:name="android.intent.action.PACKAGE_REMOVED"/>
                <data android:scheme="package"></data>
            </intent-filter>
        </receiver>

        <!-- The settings provider contains Home's data, like the workspace favorites -->
        <provider
            android:name="com.android.launcher2.LauncherProvider"
            android:authorities="com.android.launcher2.settings"
            android:exported="true"
            android:writePermission="com.android.launcher.permission.WRITE_SETTINGS"
            android:readPermission="com.android.launcher.permission.READ_SETTINGS" />

        <meta-data android:name="android.nfc.disable_beam_default"
                   android:value="true" />
    </application>
</manifest>
```

此时会返回 com.android.launcher2.Launcher 这个 Activity。因为是第一次启动这个 Activity，所以，接下来调用函数 getProcessRecordLocked 返回的 ProcessRecord 值是 null，于是调用函数 startActivityLocked 启动 com.android.launcher2.Launcher 这个 Activity，这里的 mMainStack 是一个 ActivityStack 类型的成员变量。

11.6.3　启动 com.android.launcher2.Launcher

通过函数 startActivityLocked 把 com.android.launcher2.Launcher 启动起来，此函数在文件 frameworks\base\services\core\java\com\android\server\am\ActivityStack.java 中定义，具体实现代码如下所示：

```
final void startActivityLocked(ActivityRecord r, boolean newTask,
        boolean doResume, boolean keepCurTransition, Bundle options) {
    TaskRecord rTask = r.task;
    final int taskId = rTask.taskId;
    // mLaunchTaskBehind tasks get placed at the back of the task stack.
    if (!r.mLaunchTaskBehind && (taskForIdLocked(taskId) == null || newTask)) {
        // Last activity in task had been removed or ActivityManagerService is reusing task.
        // Insert or replace.
        // Might not even be in.
        insertTaskAtTop(rTask);
        mWindowManager.moveTaskToTop(taskId);
    }
    TaskRecord task = null;
    if (!newTask) {
        // If starting in an existing task, find where that is...
        boolean startIt = true;
        for (int taskNdx = mTaskHistory.size() - 1; taskNdx >= 0; --taskNdx) {
            task = mTaskHistory.get(taskNdx);
            if (task.getTopActivity() == null) {
                // All activities in task are finishing.
                continue;
            }
            if (task == r.task) {
                // Here it is!  Now, if this is not yet visible to the
                // user, then just add it without starting; it will
                // get started when the user navigates back to it.
                if (!startIt) {
                    if (DEBUG_ADD_REMOVE) Slog.i(TAG, "Adding activity " + r + " to task
                    "+ task, new RuntimeException("here").fillInStackTrace());
                    task.addActivityToTop(r);
                    r.putInHistory();
                    mWindowManager.addAppToken(task.mActivities.indexOf(r), r.appToken,
                    r.task.taskId, mStackId, r.info.screenOrientation, r.fullscreen,
                            (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0,
                            r.userId, r.info.configChanges, task.voiceSession != null,
                            r.mLaunchTaskBehind);
```

```
                if (VALIDATE_TOKENS) {
                    validateAppTokensLocked();
                }
                ActivityOptions.abort(options);
                return;
            }
            break;
        } else if (task.numFullscreen > 0) {
            startIt = false;
        }
    }
}

// Place a new activity at top of stack, so it is next to interact
// with the user.

// If we are not placing the new activity frontmost, we do not want
// to deliver the onUserLeaving callback to the actual frontmost
// activity
if (task == r.task && mTaskHistory.indexOf(task) != (mTaskHistory.size() - 1)) {
    mStackSupervisor.mUserLeaving = false;
    if (DEBUG_USER_LEAVING) Slog.v(TAG,
            "startActivity() behind front, mUserLeaving=false");
}

task = r.task;

// Slot the activity into the history stack and proceed
if (DEBUG_ADD_REMOVE) Slog.i(TAG, "Adding activity " + r + " to stack to task " + task,
new RuntimeException("here").fillInStackTrace());
task.addActivityToTop(r);
task.setFrontOfTask();

r.putInHistory();
if (!isHomeStack() || numActivities() > 0) {
    // We want to show the starting preview window if we are
    // switching to a new task, or the next activity's process is
    // not currently running.
    boolean showStartingIcon = newTask;
    ProcessRecord proc = r.app;
    if (proc == null) {
        proc = mService.mProcessNames.get(r.processName, r.info.applicationInfo.uid);
    }
    if (proc == null || proc.thread == null) {
        showStartingIcon = true;
    }
    if (DEBUG_TRANSITION) Slog.v(TAG,
            "Prepare open transition: starting " + r);
    if ((r.intent.getFlags()&Intent.FLAG_ACTIVITY_NO_ANIMATION) != 0) {
        mWindowManager.prepareAppTransition(AppTransition.TRANSIT_NONE,
        keepCurTransition);
        mNoAnimActivities.add(r);
    } else {
        mWindowManager.prepareAppTransition(newTask
                ? r.mLaunchTaskBehind
                        ? AppTransition.TRANSIT_TASK_OPEN_BEHIND
                        : AppTransition.TRANSIT_TASK_OPEN
                : AppTransition.TRANSIT_ACTIVITY_OPEN, keepCurTransition);
        mNoAnimActivities.remove(r);
    }
    mWindowManager.addAppToken(task.mActivities.indexOf(r),
            r.appToken, r.task.taskId, mStackId, r.info.screenOrientation, r.fullscreen,
            (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0, r.userId,
            r.info.configChanges, task.voiceSession != null, r.mLaunchTaskBehind);
    boolean doShow = true;
    if (newTask) {
        // Even though this activity is starting fresh, we still need
        // to reset it to make sure we apply affinities to move any
        // existing activities from other tasks in to it.
        // If the caller has requested that the target task be
        // reset, then do so.
```

```
            if ((r.intent.getFlags() & Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED) != 0) {
                resetTaskIfNeededLocked(r, r);
                doShow = topRunningNonDelayedActivityLocked(null) == r;
            }
        } else if (options != null && new ActivityOptions(options).getAnimationType()
                == ActivityOptions.ANIM_SCENE_TRANSITION) {
            doShow = false;
        }
        if (r.mLaunchTaskBehind) {
            // Don't do a starting window for mLaunchTaskBehind. More importantly make sure we
            // tell WindowManager that r is visible even though it is at the back of the stack.
            mWindowManager.setAppVisibility(r.appToken, true);
            ensureActivitiesVisibleLocked(null, 0);
        } else if (SHOW_APP_STARTING_PREVIEW && doShow) {
            // Figure out if we are transitioning from another activity that is
            // "has the same starting icon" as the next one.  This allows the
            // window manager to keep the previous window it had previously
            // created, if it still had one.
            ActivityRecord prev = mResumedActivity;
            if (prev != null) {
                // We don't want to reuse the previous starting preview if:
                // (1) The current activity is in a different task.
                if (prev.task != r.task) {
                    prev = null;
                }
                // (2) The current activity is already displayed.
                else if (prev.nowVisible) {
                    prev = null;
                }
            }
            mWindowManager.setAppStartingWindow(
                    r.appToken, r.packageName, r.theme,
                    mService.compatibilityInfoForPackageLocked(
                            r.info.applicationInfo), r.nonLocalizedLabel,
                    r.labelRes, r.icon, r.logo, r.windowFlags,
                    prev != null ? prev.appToken : null, showStartingIcon);
            r.mStartingWindowShown = true;
        }
    } else {
        // If this is the first activity, don't do any fancy animations,
        // because there is nothing for it to animate on top of.
        mWindowManager.addAppToken(task.mActivities.indexOf(r), r.appToken,
                r.task.taskId, mStackId, r.info.screenOrientation, r.fullscreen,
                (r.info.flags & ActivityInfo.FLAG_SHOW_ON_LOCK_SCREEN) != 0, r.userId,
r.info.configChanges, task.voiceSession != null, r.mLaunchTaskBehind);
        ActivityOptions.abort(options);
        options = null;
    }
    if (VALIDATE_TOKENS) {
        validateAppTokensLocked();
    }

    if (doResume) {
        mStackSupervisor.resumeTopActivitiesLocked(this, r, options);
    }
}
```

接下来会调用 com.android.launcher2.Launcher 的 onCreate 函数，此函数在 packages/apps/Launcher2/src/com/android/launcher2/Launcher.java 文件中定义，主要实现代码如下所示：

```
    protected void onCreate(Bundle savedInstanceState) {
        if (DEBUG_STRICT_MODE) {
            StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                    .detectDiskReads()
                    .detectDiskWrites()
                    .detectNetwork()   // or .detectAll() for all detectable problems
                    .penaltyLog()
                    .build());
......
        if (!mRestoring) {
            if (sPausedFromUserAction) {
```

```
            // If the user leaves launcher, then we should just load items asynchronously
when
            // they return.
            mModel.startLoader(true, -1);
......
```

在上述代码中，mModel 是一个 LauncherModel 类型的成员变量，此处通过调用它的 startLoader 成员函数来执行加载应用程序的操作。

11.6.4 加载应用程序

在文件 packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java 中定义，通过函数 startLoader 加载应用程序，具体实现代码如下所示：

```
public void startLoader(boolean isLaunching, int synchronousBindPage) {
    synchronized (mLock) {
        if (DEBUG_LOADERS) {
            Log.d(TAG, "startLoader isLaunching=" + isLaunching);
        }

        // Clear any deferred bind-runnables from the synchronized load process
        // We must do this before any loading/binding is scheduled below.
        mDeferredBindRunnables.clear();

        // Don't bother to start the thread if we know it's not going to do anything
        if (mCallbacks != null && mCallbacks.get() != null) {
            // If there is already one running, tell it to stop.
            // also, don't downgrade isLaunching if we're already running
            isLaunching = isLaunching || stopLoaderLocked();
            mLoaderTask = new LoaderTask(mApp, isLaunching);
            if (synchronousBindPage > -1 && mAllAppsLoaded && mWorkspaceLoaded) {
                mLoaderTask.runBindSynchronousPage(synchronousBindPage);
            } else {
                sWorkerThread.setPriority(Thread.NORM_PRIORITY);
                sWorker.post(mLoaderTask);
            }
        }
    }
}
```

此处不是直接加载应用程序，而是把加载应用程序的操作作为一个消息来处理。这里的 sWorker 是一个 Handler，通过它的 post 方式把一个消息放在消息队列中，然后系统就会调用传进去的参数 mLoaderTask 的 run 函数来处理这个消息，这个 mLoaderTask 是 LoaderTask 类型的实例。所以接下来会执行类 LoaderTask 的 run 函数。函数 run 在文件 packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java 中定义，主要实现代码如下所示：

```
public void run() {
    synchronized (mLock) {
        mIsLoaderTaskRunning = true;
    }

    keep_running: {
        // Elevate priority when Home launches for the first time to avoid
        // starving at boot time. Staring at a blank home is not cool.
        synchronized (mLock) {
            if (DEBUG_LOADERS) Log.d(TAG, "Setting thread priority to " +
                    (mIsLaunching ? "DEFAULT" : "BACKGROUND"));
            Process.setThreadPriority(mIsLaunching
                    ? Process.THREAD_PRIORITY_DEFAULT : Process.THREAD_PRIORITY_BACKGROUND);
        }

        // First step. Load workspace first, this is necessary since adding of apps from
        // managed profile in all apps is deferred until onResume. See http://b/17336902.
        if (DEBUG_LOADERS) Log.d(TAG, "step 1: loading workspace");
        loadAndBindWorkspace();

        if (mStopped) {
            break keep_running;
```

```
        }

        // Whew! Hard work done.  Slow us down, and wait until the UI thread has
        // settled down.
        synchronized (mLock) {
            if (mIsLaunching) {
                if (DEBUG_LOADERS) Log.d(TAG, "Setting thread priority to BACKGROUND");
                Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
            }
        }
        waitForIdle();

        // Second step. Load all apps.
        if (DEBUG_LOADERS) Log.d(TAG, "step 2: loading all apps");
        loadAndBindAllApps();

        // Restore the default thread priority after we are done loading items
        synchronized (mLock) {
            Process.setThreadPriority(Process.THREAD_PRIORITY_DEFAULT);
        }
    }
```

在上述代码中，调用成员函数 loadAndBindAllApps 实现了进一步的操作。函数 loadAndBindAllApps 在文件 packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java 中定义，主要实现代码如下所示：

```
private void loadAndBindAllApps() {
    if (DEBUG_LOADERS) {
        Log.d(TAG, "loadAndBindAllApps mAllAppsLoaded=" + mAllAppsLoaded);
    }
    if (!mAllAppsLoaded) {
        loadAllAppsByBatch();
        synchronized (LoaderTask.this) {
            if (mStopped) {
                return;
            }
            mAllAppsLoaded = true;
        }
    } else {
        onlyBindAllApps();
    }
}
```

在上述代码中，因为还没有加载过应用程序，所以，这里的 mAllAppsLoaded 为 false，接下来需要继续调用函数 loadAllAppsByBatch 实现进一步的操作。函数 loadAllAppsByBatch 在文件 packages/apps/Launcher2/src/com/android/launcher2/LauncherModel.java 中定义，主要实现代码如下所示：

```
private void loadAllAppsByBatch() {
    final long t = DEBUG_LOADERS ? SystemClock.uptimeMillis() : 0;

    // Don't use these two variables in any of the callback runnables.
    // Otherwise we hold a reference to them.
    final Callbacks oldCallbacks = mCallbacks.get();
    if (oldCallbacks == null) {
        // This launcher has exited and nobody bothered to tell us.  Just bail.
        Log.w(TAG, "LoaderTask running with no launcher (loadAllAppsByBatch)");
        return;
    }

    final Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
    mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);

    final List<UserHandle> profiles = mUserManager.getUserProfiles();

    mBgAllAppsList.clear();
    final int profileCount = profiles.size();
    for (int p = 0; p < profileCount; p++) {
        UserHandle user = profiles.get(p);
```

```
List<LauncherActivityInfo> apps = null;
int N = Integer.MAX_VALUE;

int startIndex;
int i = 0;
int batchSize = -1;
while (i < N && !mStopped) {
    if (i == 0) {
        final long qiaTime = DEBUG_LOADERS ? SystemClock.uptimeMillis() : 0;
        apps = mLauncherApps.getActivityList(null, user);
        if (DEBUG_LOADERS) {
            Log.d(TAG, "queryIntentActivities took "
                    + (SystemClock.uptimeMillis()-qiaTime) + "ms");
        }
        if (apps == null) {
            return;
        }
        N = apps.size();
        if (DEBUG_LOADERS) {
            Log.d(TAG, "queryIntentActivities got " + N + " apps");
        }
        if (N == 0) {
            // There are no apps?!?
            return;
        }
        if (mBatchSize == 0) {
            batchSize = N;
        } else {
            batchSize = mBatchSize;
        }

        final long sortTime = DEBUG_LOADERS ? SystemClock.uptimeMillis() : 0;
        Collections.sort(apps,
                new LauncherModel.ShortcutNameComparator(mLabelCache));
        if (DEBUG_LOADERS) {
            Log.d(TAG, "sort took "
                    + (SystemClock.uptimeMillis()-sortTime) + "ms");
        }
    }

    final long t2 = DEBUG_LOADERS ? SystemClock.uptimeMillis() : 0;

    startIndex = i;
    for (int j=0; i<N && j<batchSize; j++) {
        // This builds the icon bitmaps.
        mBgAllAppsList.add(new ApplicationInfo(apps.get(i), user,
                mIconCache, mLabelCache));
        i++;
    }

    final Callbacks callbacks = tryGetCallbacks(oldCallbacks);
    final ArrayList<ApplicationInfo> added = mBgAllAppsList.added;
    final boolean firstProfile = p == 0;
    mBgAllAppsList.added = new ArrayList<ApplicationInfo>();
    mHandler.post(new Runnable() {
        public void run() {
            final long t = SystemClock.uptimeMillis();
            if (callbacks != null) {
                if (firstProfile) {
                    callbacks.bindAllApplications(added);
                } else {
                    callbacks.bindAppsAdded(added);
                }
                if (DEBUG_LOADERS) {
                    Log.d(TAG, "bound " + added.size() + " apps in "
                        + (SystemClock.uptimeMillis() - t) + "ms");
                }
            } else {
                Log.i(TAG, "not binding apps: no Launcher activity");
            }
        }
```

```
            });

            if (DEBUG_LOADERS) {
                Log.d(TAG, "batch of " + (i-startIndex) + " icons processed in "
                        + (SystemClock.uptimeMillis()-t2) + "ms");
            }

            if (mAllAppsLoadDelay > 0 && i < N) {
                try {
                    if (DEBUG_LOADERS) {
                        Log.d(TAG, "sleeping for " + mAllAppsLoadDelay + "ms");
                    }
                    Thread.sleep(mAllAppsLoadDelay);
                } catch (InterruptedException exc) { }
            }
        }

        if (DEBUG_LOADERS) {
            Log.d(TAG, "cached all " + N + " apps in "
                    + (SystemClock.uptimeMillis()-t) + "ms"
                    + (mAllAppsLoadDelay > 0 ? " (including delay)" : ""));
        }
    }
}
```

对上述代码的具体说明如下所示。

（1）首先构造一个 CATEGORY_LAUNCHER 类型的 Intent。

（2）从 mContext 变量中获得 PackageManagerService 的接口。

（3）通过接口 queryIntentActivities 取回所有 Action 类型为 Intent.ACTION_MAIN，且 Category 类型为 Intent.CATEGORY_LAUNCHER 的 Activity。

11.6.5　获得 Activity

在文件 frameworks/base/services/java/com/android/server/PackageManagerService.java 中，通过接口 queryIntentActivities 取回所有 Action 类型为 Intent.ACTION_MAIN，且 Category 类型为 Intent.CATEGORY_LAUNCHER 的 Activity。接口 queryIntentActivities 的具体实现代码如下所示：

```
public List<ResolveInfo> queryIntentActivities(Intent intent,
        String resolvedType, int flags, int userId) {
    if (!sUserManager.exists(userId)) return Collections.emptyList();
    enforceCrossUserPermission(Binder.getCallingUid(), userId, false, false, "query
intent activities");
    ComponentName comp = intent.getComponent();
    if (comp == null) {
        if (intent.getSelector() != null) {
            intent = intent.getSelector();
            comp = intent.getComponent();
        }
    }

    if (comp != null) {
        final List<ResolveInfo> list = new ArrayList<ResolveInfo>(1);
        final ActivityInfo ai = getActivityInfo(comp, flags, userId);
        if (ai != null) {
            final ResolveInfo ri = new ResolveInfo();
            ri.activityInfo = ai;
            list.add(ri);
        }
        return list;
    }

    // reader
    synchronized (mPackages) {
        final String pkgName = intent.getPackage();
        if (pkgName == null) {
            List<CrossProfileIntentFilter> matchingFilters =
                    getMatchingCrossProfileIntentFilters(intent, resolvedType, userId);
```

```
            // Check for results that need to skip the current profile.
            ResolveInfo resolveInfo=querySkipCurrentProfileIntents(matchingFilters,intent,
                    resolvedType, flags, userId);
            if (resolveInfo != null) {
                List<ResolveInfo> result = new ArrayList<ResolveInfo>(1);
                result.add(resolveInfo);
                return result;
            }
            // Check for cross profile results.
            resolveInfo = queryCrossProfileIntents(
                    matchingFilters, intent, resolvedType, flags, userId);

            // Check for results in the current profile.
            List<ResolveInfo> result = mActivities.queryIntent(
                    intent, resolvedType, flags, userId);
            if (resolveInfo != null) {
                result.add(resolveInfo);
                Collections.sort(result, mResolvePrioritySorter);
            }
            return result;
        }
        final PackageParser.Package pkg = mPackages.get(pkgName);
        if (pkg != null) {
            return mActivities.queryIntentForPackage(intent, resolvedType, flags,
                    pkg.activities, userId);
        }
        return new ArrayList<ResolveInfo>();
    }
}
```

在启动 PackageManagerService 时，会把系统中的应用程序都解析一遍，然后把解析得到的 Activity 保存在 mActivities 变量中。在上述代码中，通过 mActivities 变量的 queryIntent 函数返回符合条件 intent 的 Activity，此处要返回的便是 Action 类型为 Intent.ACTION_MAIN，并且 Category 类型为 Intent.CATEGORY_LAUNCHER 的 Activity。当从函数 queryIntentActivities 返回所要求的 Activity 后，接下来调用函数 tryGetCallbacks(oldCallbacks)得到一个返回的 CallBack 接口，这个接口是由 Launcher 类实现的，接着调用这个接口中的函数 bindAllApplications 来进一步操作。注意，此处是通过消息来处理加载应用程序的操作的。函数 bindAllApplications 在文件 packages/apps/Launcher2/src/com/android/launcher2/Launcher.java 中定义，具体实现代码如下所示：

```
public void bindAllApplications(final ArrayList<ApplicationInfo> apps) {
    Runnable setAllAppsRunnable = new Runnable() {
        public void run() {
            if (mAppsCustomizeContent != null) {
                mAppsCustomizeContent.setApps(apps);
            }
        }
    };

    // Remove the progress bar entirely; we could also make it GONE
    // but better to remove it since we know it's not going to be used
    View progressBar = mAppsCustomizeTabHost.
        findViewById(R.id.apps_customize_progress_bar);
    if (progressBar != null) {
        ((ViewGroup)progressBar.getParent()).removeView(progressBar);

        // We just post the call to setApps so the user sees the progress bar
        // disappear-- otherwise, it just looks like the progress bar froze
        // which doesn't look great
        mAppsCustomizeTabHost.post(setAllAppsRunnable);
    } else {
        // If we did not initialize the spinner in onCreate, then we can directly set the
        // list of applications without waiting for any progress bars views to be hidden.
        setAllAppsRunnable.run();
    }
}
```

在上述代码中，mAllAppsGrid 是一个 AllAppsView 类型的变量，其实际类型一般是 AllApps2D。

在文件 packages/apps/Launcher2/src/com/android/launcher2/AllAppsList.java 中，通过函数 addPackage 为 APK 程序添加启动图标，通过函数 removePackage 删除给定的应用程序，通过函数 updatePackage 添加或删除已经更新包的图标，主要实现代码如下所示：

```
public void addPackage(Context context, String packageName, UserHandle user) {
    LauncherApps launcherApps = (LauncherApps)
            context.getSystemService(Context.LAUNCHER_APPS_SERVICE);
    final List<LauncherActivityInfo> matches = launcherApps.getActivityList(packageName,
user);

    for (LauncherActivityInfo info : matches) {
        add(new ApplicationInfo(info, user, mIconCache, null));
    }
}

/**
 * Remove the apps for the given apk identified by packageName.
 */
public void removePackage(String packageName, UserHandle user) {
    final List<ApplicationInfo> data = this.data;
    for (int i = data.size() - 1; i >= 0; i--) {
        ApplicationInfo info = data.get(i);
        final ComponentName component = info.intent.getComponent();
        if (info.user.equals(user) && packageName.equals(component.getPackageName())) {
            removed.add(info);
            data.remove(i);
        }
    }
    // This is more aggressive than it needs to be.
    mIconCache.flush();
}

/**
 * Add and remove icons for this package which has been updated.
 */
public void updatePackage(Context context, String packageName, UserHandle user) {
    LauncherApps launcherApps = (LauncherApps)
            context.getSystemService(Context.LAUNCHER_APPS_SERVICE);
    final List<LauncherActivityInfo>matches=launcherApps.getActivityList(packageName,user);
    if (matches.size() > 0) {
        // Find disabled/removed activities and remove them from data and add them
        // to the removed list.
        for (int i = data.size() - 1; i >= 0; i--) {
            final ApplicationInfo applicationInfo = data.get(i);
            final ComponentName component = applicationInfo.intent.getComponent();
            if (user.equals(applicationInfo.user)
                    && packageName.equals(component.getPackageName())) {
                if (!findActivity(matches, component, user)) {
                    removed.add(applicationInfo);
                    mIconCache.remove(component);
                    data.remove(i);
                }
            }
        }

        // Find enabled activities and add them to the adapter
        // Also updates existing activities with new labels/icons
        int count = matches.size();
        for (int i = 0; i < count; i++) {
            final LauncherActivityInfo info = matches.get(i);
            ApplicationInfo applicationInfo = findApplicationInfoLocked(
                    info.getComponentName().getPackageName(),
                    info.getComponentName().getShortClassName(),
                    user);
            if (applicationInfo == null) {
                add(new ApplicationInfo(info, user,
                        mIconCache, null));
            } else {
                mIconCache.remove(applicationInfo.componentName);
                mIconCache.getTitleAndIcon(applicationInfo, info, null);
```

```
                modified.add(applicationInfo);
            }
        }
    } else {
        // Remove all data for this package.
        for (int i = data.size() - 1; i >= 0; i--) {
            final ApplicationInfo applicationInfo = data.get(i);
            final ComponentName component = applicationInfo.intent.getComponent();
            if (user.equals(applicationInfo.user)
                    && packageName.equals(component.getPackageName())) {
                removed.add(applicationInfo);
                mIconCache.remove(component);
                data.remove(i);
            }
        }
    }
}
```

此时系统默认的 Home 应用程序 Launcher 会把 PackageManagerService 中的应用程序加载进来，当在图 11-2 所示的屏幕中点击图标▦时，会把刚才加载好的应用程序以图标的形式展示出来。

▲图 11-2 系统启动界面

当点击图标▦时会执行函数 Launcher.onClick，具体实现代码如下所示：

```
public void onClick(View v) {
    // Make sure that rogue clicks don't get through while allapps is launching, or after the
    // view has detached (it's possible for this to happen if the view is removed mid touch).
    if (v.getWindowToken() == null) {
        return;
    }

    if (!mWorkspace.isFinishedSwitchingState()) {
        return;
    }

    Object tag = v.getTag();
    if (tag instanceof ShortcutInfo) {
        // Open shortcut
        final Intent intent = ((ShortcutInfo) tag).intent;
        int[] pos = new int[2];
        v.getLocationOnScreen(pos);
        intent.setSourceBounds(new Rect(pos[0], pos[1],
                pos[0] + v.getWidth(), pos[1] + v.getHeight()));

        boolean success = startActivitySafely(v, intent, tag);

        if (success && v instanceof BubbleTextView) {
            mWaitingForResume = (BubbleTextView) v;
            mWaitingForResume.setStayPressed(true);
        }
    } else if (tag instanceof FolderInfo) {
```

```
        if (v instanceof FolderIcon) {
            FolderIcon fi = (FolderIcon) v;
            handleFolderClick(fi);
        }
    } else if (v == mAllAppsButton) {
        if (isAllAppsVisible()) {
            showWorkspace(true);
        } else {
            onClickAllAppsButton(v);
        }
    }
}
```

在上述代码中调用函数 showAllApps 显示了系统应用程序的图标，具体实现代码如下所示：

```
void showAllApps(boolean animated) {
    if (mState != State.WORKSPACE) return;

    showAppsCustomizeHelper(animated, false);
    mAppsCustomizeTabHost.requestFocus();

    // Change the state *after* we've called all the transition code
    mState = State.APPS_CUSTOMIZE;

    // Pause the auto-advance of widgets until we are out of AllApps
    mUserPresent = false;
    updateRunning();
    closeFolder();

    // Send an accessibility event to announce the context change
    getWindow().getDecorView()
            .sendAccessibilityEvent(AccessibilityEvent.TYPE_WINDOW_STATE_CHANGED);
}
```

此时就可以看到系统中的应用程序了，如图 11-3 所示。当点击这些应用程序图标时，就会打开这些应用程序。

▲图 11-3　系统中的应用程序

第 12 章　Content Provider 存储机制

在 Android 系统中，没有一个公共的内存区域供多个应用共享存储数据，通过 Content Provider 模块可以支持在多个应用中存储和读取数据，这是跨应用共享数据的唯一方式。在本章的内容中，将详细讲解 Android 5.0 中 Content Provider 模块的源代码。

12.1 Content Provider 基础

在 Android 系统中，Content Provider 作为应用程序 4 大组件之一，它起到在应用程序之间共享数据的作用，同时，它还是标准的数据访问接口。在本节的内容中，将简要介绍 Content Provider 组件的基本知识。

12.1.1　Content Provider 在应用程序中的架构

如果使用命令 adb shell 连接 Android 模拟器，在/data/data 目录下会看到很多以应用程序包（package）命名的文件夹，在这些文件夹中存放的是各个应用程序的数据文件。例如，进入到 Android 系统日历应用程序数据目录 com.android.providers.calendar 下的 databases 文件中，会看到一个用来保存日历数据的数据库文件 calendar.db，其权限设置代码如下所示：

```
root@android:/data/data/com.android.providers.calendar/databases # ls -l
-rw-rw---- app_17   app_17      33792 2013-09-07 15:50 calendar.db
```

在上述代码“-rw-rw----”中，各个字符的具体说明如下所示。

- 最前面的符号“-”：表示这是一个普通文件。
- 接下来的 3 个字符“rw-”：表示此文件的所有者对这个文件可读、可写、不可执行。
- 接下来的 3 个字符“rw-”：表示这个文件的所有者所在的用户组的用户对这个文件可读、可写、不可执行。
- 最后的 3 个字符“---”：表示其他的用户对这个文件不可读写、也不可执行。因为这是一个数据文件，所以，所有用户都不可以执行它是正确的。
- 接下来的两个“app_17”字符串：表示这个文件的所有者和这个所有者所在的用户组的名称均为“app_17”，这是在安装应用程序时由系统分配的。在不同的系统上这个字符串可能会不同，但是，所表示的意义是一样的。表示只有用户 ID 为 app_17，或者用户组 ID 为 app_17 的进程才可以对文件“calendar.db”进行读写操作。通过执行终端上的 ps 命令可以查看哪个进程的用户 ID 为 app_17。

Android 系统对应用程序的数据文件有着严格的保护措施。当我们在开发自己的应用程序时，有时会希望读取通讯录里面某个联系人的手机号码或者电子邮件，以便可以和这个联系人打电话或者发送电子邮件，这时就需要读取通讯录里面的联系人数据文件。在 Android 应用程序中，有很多第三方公司都推出了专业性的 API。这些 API 的目标是要将开放用户数据给第三方来使用，例如，Android 系统中的通讯录，它需要把自己联系人数据开放出来给其他应用程序使用。但是，这些数据都是各个平台自己的核心数据和核心竞争力，它们需要有保护地进行开放。为此，Android 系统特意推出了 Content Provider，它秉承了有保护地开放自己的数据给其他应用程序使用的理念。

Content Provider 机制在 Android 应用项目开发中的地位十分高，这是现实需求所决定的。例如，在设计大型的复杂的软件中，需要分模块和分层次来实现各个子功能组件，使得各个模块功能以松耦合的方式组织在一起完成整个应用程序功能。这样做的好处如下所示。

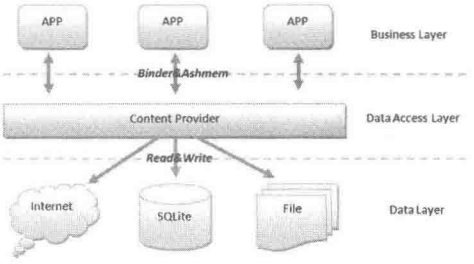
- 便于维护和扩展应用程序的代码和功能。
- 更加适应复杂的业务环境。

如果从垂直的方向来看一个大型的应用程序软件架构，通常会分为如下所示的层次结构。

- 数据层：用来保存数据，这些数据可以用文件的方式来组织，也可以用数据库的方式来组织，甚至可以保存在网络中。
- 数据访问接口层：负责向上面的业务层提供数据，而向下管理好数据层的数据。
- 业务层：通过数据访问层来获取一些业务相关的数据来实现自己的业务逻辑。

根据上述层次结构的描述，可以得出一个通用 Android 应用程序的架构图，如图 12-1 所示。

在图 12-1 所示的架构中，SQLite 数据库、文件或网络来保存数据，数据访问层使用 Content Provider 来实现，而业务层就通过一些 App 来实现。为了降低各个功能模块间耦合性，可以把业务层的各个 App 和数据访问层中的 Content Provider 放在不同的应用程序进程中来实现。而数据库中的数据统一由 Content Provider 来管理，Content Provider 拥有对这些文件直接进行读写的权限，并且可以根据需要来有保护地把这些数据开放出来供上层 App 使用。



▲图 12-1　通用 Android 应用程序的架构图

12.1.2　Content Provider 的常用接口

在 Android 系统中的数据是私有的，当然这些数据包括文件数据和数据库数据，以及一些其他类型的数据。Android 中的两个程序之间可以进行数据交换，此功能就是通过 Content Provider 实现的。一个 Content Provider 类实现了一组标准的方法接口，从而能够让其他的应用保存或读取此 Content Provider 的各种数据类型。也就是说，一个程序可以通过实现一个 Content Provider 的抽象接口的方式将自己的数据暴露出去，而外界根本看不到，并且也不用看到这个应用暴露的数据在应用当中是如何存储的，或者是用数据库存储还是用文件存储，还是通过网上获得。这些一切都不重要，重要的是外界可以通过这一套标准及统一的接口和程序里的数据打交道，可以读取程序的数据，也可以删除程序的数据。当然，中间也可能会涉及一些权限的问题。现实中比较常见的接口如下所示。

- query（Uri uri, String[] projection, String selection, String[] selectionArgs,String sortOrder）：通过 Uri 进行查询，返回一个 Cursor。
- insert（Uri url, ContentValues values）：将一组数据插入到 Uri 指定的地方。
- update（Uri uri, ContentValues values, String where, String[] selectionArgs）：更新 Uri 指定位置的数据。
- delete（Uri url, String where, String[] selectionArgs）：删除指定 Uri 并且符合一定条件的数据。

（1）ContentResolver 接口

外界的程序通过 ContentResolver 接口可以访问 ContentProvider 提供的数据，在 Activity 当中通过 getContentResolver()可以得到当前应用的 ContentResolver 实例。ContentResolver 提供的接口和 ContentProvider 中需要实现的接口对应，具体来说主要有以下几个。

- query（Uri uri, String[] projection, String selection, String[] selectionArgs,String sortOrder）：通过 Uri 进行查询，返回一个 Cursor。
- insert（Uri url, ContentValues values）：将一组数据插入到 Uri 指定的地方。
- update（Uri uri, ContentValues values, String where, String[] selectionArgs）：更新 Uri 指定位置的数据。
- delete（Uri url, String where, String[] selectionArgs）：删除指定 Uri 并且符合一定条件的数据。

（2）ContentProvider 和 ContentResolver 中的 Uri

在 ContentProvider 和 ContentResolver 中，使用的 Uri 的形式通常有两种，一种是指定全部数据，另一种是指定某个 ID 的数据。我们看下面的例子：

```
content://contacts/people/      //此 Uri 指定的就是全部的联系人数据
content://contacts/people/1     //此 Uri 指定的是 ID 为 1 的联系人的数据
```

在上边两个类中用到的 Uri 一般由 3 部分组成，具体说明如下：

- 第一部分是："content://"。
- 第二部分是要获得数据的一个字符串片段。
- 第三部分是 ID（如果没有指定 ID，那么表示返回全部）。

因为 URI 通常比较长，而且有时候容易出错，且难以理解。所以，在 Android 当中定义了一些辅助类，并且定义了一些常量来代替这些长字符串的使用，例如下面的代码：

```
Contacts.People.CONTENT_URI （联系人的 URI）
```

12.2 启动 Content Provider

在 Android 系统中，Content Provider 能够为不同的应用程序访问相同的数据提供统一的入口。通常 Content Provider 运行在独立的进程中，在系统中的每个 Content Provider 只存在一个实例，其他应用程序需要在找到这个实例后才能访问它的数据。

12.2.1 获得对象接口

在启动 Content Provider 之前，首先需要一个启动参数，如下面的代码：

```
public int getDataCount() {
    int count = 0;

    try {
        IContentProvider provider = resolver.acquireProvider(Datas.CONTENT_URI);
        Bundle bundle = provider.call(Datas.METHOD_GET_ITEM_COUNT, null, null);
        count = bundle.getInt(Datas.KEY_ITEM_COUNT, 0);
    } catch(RemoteException e) {
        e.printStackTrace();
    }
```

这样通过调用 ContentResolver 接口 resolver 中的函数 acquireProvider，获得与 Data.CONTENT_URI 对应的 Content Provider 对象的 IContentProvider 接口。函数 acquireProvider 在文件 frameworks/base/core/java/android/content/ContentResolver.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireProvider(Uri uri) {
    if (!SCHEME_CONTENT.equals(uri.getScheme())) {
        return null;
    }
    final String auth = uri.getAuthority();
    if (auth != null) {
        return acquireProvider(mContext, auth);
    }
    return null;
}
```

在上述代码中，首先验证参数 uri 的 scheme 是否正确，验证此参数是否是以“content://”开头，并取出它的 authority 部分，然后调用函数 acquireProvider 获取 ContentProvider 接口的操作。

再看另外一个成员函数 acquireProvider，此函数在文件 frameworks/base/core/java/android/app/ContextImpl.java 中定义，具体实现代码如下所示：

```
protected IContentProvider acquireProvider(Context context, String name) {
    return mMainThread.acquireProvider(context, name);
}
```

在上述代码中，调用类 ActivityThread 中的函数 acquireProvider 来获取 Content Provider 接口。

下面看类 ActivityThread 中的函数 acquireProvider，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireProvider(
```

```
        Context c, String auth, int userId, boolean stable) {
    final IContentProvider provider = acquireExistingProvider(c, auth, userId, stable);
    if (provider != null) {
        return provider;
    }
    IActivityManager.ContentProviderHolder holder = null;
    try {
        holder = ActivityManagerNative.getDefault().getContentProvider(
                getApplicationThread(), auth, userId, stable);
    } catch (RemoteException ex) {
    }
    if (holder == null) {
        Slog.e(TAG, "Failed to find provider info for " + auth);
        return null;
    }
    holder = installProvider(c, holder, holder.info,
            true /*noisy*/, holder.noReleaseNeeded, stable);
    return holder.provider;
}
```

在上述代码中，调用函数 getProvider 来进一步获取 Content Provider 接口。

12.2.2　存在校验

接下来看函数 acquireExistingProvider，功能是校验本地是否已经存在此要获取的 ContentProvider 接口，如果存在则调用函数 getProvider 直接获取。函数 acquireExistingProvider 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
public final IContentProvider acquireExistingProvider(
        Context c, String auth, int userId, boolean stable) {
    synchronized (mProviderMap) {
        final ProviderKey key = new ProviderKey(auth, userId);
        final ProviderClientRecord pr = mProviderMap.get(key);
        if (pr == null) {
            return null;
        }

        IContentProvider provider = pr.mProvider;
        IBinder jBinder = provider.asBinder();
        if (!jBinder.isBinderAlive()) {
            // The hosting process of the provider has died; we can't
            // use this one.
            Log.i(TAG, "Acquiring provider " + auth + " for user " + userId
                    + ": existing object's process dead");
            handleUnstableProviderDiedLocked(jBinder, true);
            return null;
        }

        // Only increment the ref count if we have one.  If we don't then the
        // provider is not reference counted and never needs to be released.
        ProviderRefCount prc = mProviderRefCountMap.get(jBinder);
        if (prc != null) {
            incProviderRefLocked(prc, stable);
        }
        return provider;
    }
}
```

接下来看函数 getContentProvider，功能是处理类型为 GET_CONTENT_PROVIDER_TRANSACTION 的进程通信请求，并调用函数 getContentProviderImpl 实现进一步处理。函数 getContentProvider 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
public final ContentProviderHolder getContentProvider(
        IApplicationThread caller, String name, int userId, boolean stable) {
    enforceNotIsolatedCaller("getContentProvider");
    if (caller == null) {
        String msg = "null IApplicationThread when getting content provider "
```

```
                + name;
        Slog.w(TAG, msg);
        throw new SecurityException(msg);
    }

    userId = handleIncomingUser(Binder.getCallingPid(), Binder.getCallingUid(), userId,
            false, true, "getContentProvider", null);
    return getContentProviderImpl(caller, name, null, stable, userId);
}
```

再看函数 getContentProviderImpl，功能是或一个 ContentProviderHolder 对象，此对象用来描述 Content Provider 的代理对象。函数 getContentProviderImpl 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
    private final ContentProviderHolder getContentProviderImpl(IApplicationThread caller,
            String name, IBinder token, boolean stable, int userId) {
        ContentProviderRecord cpr;
        ContentProviderConnection conn = null;
        ProviderInfo cpi = null;

        synchronized(this) {
            ProcessRecord r = null;
            if (caller != null) {
//获取 ActivityManagerService 返回的和参数 name 对应的 Content Provider 代理对象的程序进程信息
                r = getRecordForAppLocked(caller);
                if (r == null) {
                    throw new SecurityException(
                            "Unable to find app for caller " + caller
                          + " (pid=" + Binder.getCallingPid()
                          + ") when getting content provider " + name);
                }
            }

            // First check if this content provider has been published...
            cpr = mProviderMap.getProviderByName(name, userId);
            boolean providerRunning = cpr != null;
            if (providerRunning) {
                cpi = cpr.info;
                String msg;
                if ((msg=checkContentProviderPermissionLocked(cpi, r)) != null) {
                    throw new SecurityException(msg);
                }
//查看这个 Content Provider 是否设置 multiprocess 属性为 true，如果是，则允许在客户进程中被加载
                if (r != null && cpr.canRunHere(r)) {
                    // This provider has been published or is in the process
                    // of being published...  but it is also allowed to run
                    // in the caller's process, so don't make a connection
                    // and just let the caller instantiate its own instance.
                    ContentProviderHolder holder = cpr.newHolder(null);
                    holder.provider = null;
                    return holder;
                }

                final long origId = Binder.clearCallingIdentity();
                conn = incProviderCountLocked(r, cpr, token, stable);
                if (conn != null && (conn.stableCount+conn.unstableCount) == 1) {
                    if (cpr.proc != null && r.setAdj <= ProcessList.PERCEPTIBLE_APP_ADJ) {
                        updateLruProcessLocked(cpr.proc, false);
                    }
                }

                if (cpr.proc != null) {
                    if (false) {
                        if (cpr.name.flattenToShortString().equals(
                                "com.android.providers.calendar/.CalendarProvider2")) {
                            Slog.v(TAG, "****************** KILLING "
                                + cpr.name.flattenToShortString());
                            Process.killProcess(cpr.proc.pid);
                        }
                    }
```

```
                boolean success = updateOomAdjLocked(cpr.proc);
                if (DEBUG_PROVIDER) Slog.i(TAG, "Adjust success: " + success);
                if (!success) {
                    Slog.i(TAG,
                            "Existing provider " + cpr.name.flattenToShortString()
                            + " is crashing; detaching " + r);
                    boolean lastRef = decProviderCountLocked(conn, cpr, token, stable);
                    appDiedLocked(cpr.proc, cpr.proc.pid, cpr.proc.thread);
                    if (!lastRef) {
                        return null;
                    }
                    providerRunning = false;
                    conn = null;
                }
            }

            Binder.restoreCallingIdentity(origId);
        }

        boolean singleton;
        if (!providerRunning) {
            try {
//获得 PackageManagerService 服务接口，分别获取 ArticlesProvider 应用程序的相关信息，
//分别保存在 cpi 和 cpr 这两个本地变量中
                cpi = AppGlobals.getPackageManager().
                    resolveContentProvider(name,
                      STOCK_PM_FLAGS | PackageManager.GET_URI_PERMISSION_PATTERNS, userId);
            } catch (RemoteException ex) {
            }
            if (cpi == null) {
                return null;
            }
            singleton = isSingleton(cpi.processName, cpi.applicationInfo,
                    cpi.name, cpi.flags);
            if (singleton) {
                userId = 0;
            }
            cpi.applicationInfo = getAppInfoForUser(cpi.applicationInfo, userId);

            String msg;
            if ((msg=checkContentProviderPermissionLocked(cpi, r)) != null) {
                throw new SecurityException(msg);
            }

            if (!mProcessesReady && !mDidUpdate && !mWaitingUpdate
                    && !cpi.processName.equals("system")) {
                throw new IllegalArgumentException(
                        "Attempt to launch content provider before system ready");
            }
            if (mStartedUsers.get(userId) == null) {
                Slog.w(TAG, "Unable to launch app "
                        + cpi.applicationInfo.packageName + "/"
                        + cpi.applicationInfo.uid + " for provider "
                        + name + ": user " + userId + " is stopped");
                return null;
            }

            ComponentName comp = new ComponentName(cpi.packageName, cpi.name);
            cpr = mProviderMap.getProviderByClass(comp, userId);
            final boolean firstClass = cpr == null;
            if (firstClass) {
                try {
                    ApplicationInfo ai =
                        AppGlobals.getPackageManager().
                            getApplicationInfo(
                                    cpi.applicationInfo.packageName,
                                    STOCK_PM_FLAGS, userId);
                    if (ai == null) {
                        Slog.w(TAG, "No package info for content provider "
                                + cpi.name);
                        return null;
```

```
                    }
                    ai = getAppInfoForUser(ai, userId);
                    cpr = new ContentProviderRecord(this, cpi, ai, comp, singleton);
                } catch (RemoteException ex) {
                    // pm is in same process, this will never happen.
                }
            }

            if (r != null && cpr.canRunHere(r)) {
                return cpr.newHolder(null);
            }

            if (DEBUG_PROVIDER) {
                RuntimeException e = new RuntimeException("here");
                Slog.w(TAG, "LAUNCHING REMOTE PROVIDER (myuid " + r.uid
                      + " pruid " + cpr.appInfo.uid + "): " + cpr.info.name, e);
            }

        // 系统中所有正在加载的 Content Provider 都保存在 mLaunchingProviders 成员变量中。
        //在加载相应的 Content Provider 之前，首先要判断一下它是否正在被其他应用程序加载，
        //如果是就不用重复加载了
            // See if we are already in the process of launching this
            // provider.
            final int N = mLaunchingProviders.size();
            int i;
            for (i=0; i<N; i++) {
                if (mLaunchingProviders.get(i) == cpr) {
                    break;
                }
            }

            // 条件 i >= N 为 true，表明没有其他应用程序正在加载这个 Content Provider
            if (i >= N) {
                final long origId = Binder.clearCallingIdentity();

                try {
                    // Content provider is now in use, its package can't be stopped.
                    try {
                        AppGlobals.getPackageManager().setPackageStoppedState(
                                cpr.appInfo.packageName, false, userId);
                    } catch (RemoteException e) {
                    } catch (IllegalArgumentException e) {
                        Slog.w(TAG, "Failed trying to unstop package "
                                + cpr.appInfo.packageName + ": " + e);
                    }
//调用 startProcessLocked 函数来启动一个新的进程来加载这个 Content Provider 对应的类，
//然后把这个正在加载的信息增加到 mLaunchingProviders 中去
                    ProcessRecord proc = startProcessLocked(cpi.processName,
                            cpr.appInfo, false, 0, "content provider",
                            new ComponentName(cpi.applicationInfo.packageName,
                                    cpi.name), false, false);
                    if (proc == null) {
                        Slog.w(TAG, "Unable to launch app "
                                + cpi.applicationInfo.packageName + "/"
                                + cpi.applicationInfo.uid + " for provider "
                                + name + ": process is bad");
                        return null;
                    }
                    cpr.launchingApp = proc;
                    mLaunchingProviders.add(cpr);
                } finally {
                    Binder.restoreCallingIdentity(origId);
                }
            }
            if (firstClass) {
                mProviderMap.putProviderByClass(comp, cpr);
            }

            mProviderMap.putProviderByName(name, cpr);
            conn = incProviderCountLocked(r, cpr, token, stable);
            if (conn != null) {
```

```
                conn.waiting = true;
            }
        }
    }

    // Wait for the provider to be published...
    synchronized (cpr) {
        while (cpr.provider == null) {
            if (cpr.launchingApp == null) {
                Slog.w(TAG, "Unable to launch app "
                        + cpi.applicationInfo.packageName + "/"
                        + cpi.applicationInfo.uid + " for provider "
                        + name + ": launching app became null");
                EventLog.writeEvent(EventLogTags.AM_PROVIDER_LOST_PROCESS,
                        UserHandle.getUserId(cpi.applicationInfo.uid),
                        cpi.applicationInfo.packageName,
                        cpi.applicationInfo.uid, name);
                return null;
            }
            try {
                if (DEBUG_MU) {
                    Slog.v(TAG_MU, "Waiting to start provider " + cpr + " launchingApp="+
                    cpr.launchingApp);
                }
                if (conn != null) {
                    conn.waiting = true;
                }
                cpr.wait();
            } catch (InterruptedException ex) {
            } finally {
                if (conn != null) {
                    conn.waiting = false;
                }
            }
        }
    }
    return cpr != null ? cpr.newHolder(conn) : null;
}
```

12.2.3　启动 Android 应用程序

接下来按照如下所示的步骤依次启动如下所示的函数：

- ActivityManagerService.startProcessLocked;
- Process.start;
- ActivityThread.main;
- ActivityThread.attach;
- ActivityManagerService.attachApplication。

上述启动函数的过程就是依次启动 Android 应用程序的过程。

12.2.4　根据进程启动 Content Provider

接下来看函数 attachApplicationLocked，功能是处理类型为 GET_CONTENT_PROVIDER_TRANSACTION 进程通信请求，并启动指定 ID 的 Content Provider。函数 attachApplicationLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
ContentProvider
    private final boolean attachApplicationLocked(IApplicationThread thread,
            int pid) {

        // Find the application record that is being attached...  either via
        // the pid if we are running in multiple processes, or just pull the
        // next app record if we are emulating process with anonymous threads.
        ProcessRecord app;
        if (pid != MY_PID && pid >= 0) {
```

```
            synchronized (mPidsSelfLocked) {
                app = mPidsSelfLocked.get(pid);
            }
        } else {
            app = null;
        }

        if (app == null) {
            Slog.w(TAG, "No pending application record for pid " + pid
                    + " (IApplicationThread " + thread + "); dropping process");
            EventLog.writeEvent(EventLogTags.AM_DROP_PROCESS, pid);
            if (pid > 0 && pid != MY_PID) {
                Process.killProcessQuiet(pid);
            } else {
                try {
                    thread.scheduleExit();
                } catch (Exception e) {
                    // Ignore exceptions.
                }
            }
            return false;
        }

        // If this application record is still attached to a previous
        // process, clean it up now.
        if (app.thread != null) {
            handleAppDiedLocked(app, true, true);
        }
        if (localLOGV) Slog.v(
                TAG, "Binding process pid " + pid + " to record " + app);

        String processName = app.processName;
        try {
            AppDeathRecipient adr = new AppDeathRecipient(
                    app, pid, thread);
            thread.asBinder().linkToDeath(adr, 0);
            app.deathRecipient = adr;
        } catch (RemoteException e) {
            app.resetPackageList();
            startProcessLocked(app, "link fail", processName);
            return false;
        }

        EventLog.writeEvent(EventLogTags.AM_PROC_BOUND, app.userId, app.pid,
        app.processName);

        app.thread = thread;
        app.curAdj = app.setAdj = -100;
        app.curSchedGroup = Process.THREAD_GROUP_DEFAULT;
        app.setSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
        app.forcingToForeground = null;
        app.foregroundServices = false;
        app.hasShownUi = false;
        app.debugging = false;

        mHandler.removeMessages(PROC_START_TIMEOUT_MSG, app);

        boolean normalMode = mProcessesReady || isAllowedWhileBooting(app.info);
        List providers = normalMode ? generateApplicationProvidersLocked(app) : null;

        if (!normalMode) {
            Slog.i(TAG, "Launching preboot mode app: " + app);
        }

        if (localLOGV) Slog.v(
            TAG, "New app record " + app
            + " thread=" + thread.asBinder() + " pid=" + pid);
        try {
            int testMode = IApplicationThread.DEBUG_OFF;
            if (mDebugApp != null && mDebugApp.equals(processName)) {
                testMode = mWaitForDebugger
```

```
                ? IApplicationThread.DEBUG_WAIT
                : IApplicationThread.DEBUG_ON;
            app.debugging = true;
            if (mDebugTransient) {
                mDebugApp = mOrigDebugApp;
                mWaitForDebugger = mOrigWaitForDebugger;
            }
        }
        String profileFile = app.instrumentationProfileFile;
        ParcelFileDescriptor profileFd = null;
        boolean profileAutoStop = false;
        if (mProfileApp != null && mProfileApp.equals(processName)) {
            mProfileProc = app;
            profileFile = mProfileFile;
            profileFd = mProfileFd;
            profileAutoStop = mAutoStopProfiler;
        }
        boolean enableOpenGlTrace = false;
        if (mOpenGlTraceApp != null && mOpenGlTraceApp.equals(processName)) {
            enableOpenGlTrace = true;
            mOpenGlTraceApp = null;
        }

        // If the app is being launched for restore or full backup, set it up specially
        boolean isRestrictedBackupMode = false;
        if (mBackupTarget != null && mBackupAppName.equals(processName)) {
            isRestrictedBackupMode = (mBackupTarget.backupMode == BackupRecord.
            RESTORE)
                    || (mBackupTarget.backupMode == BackupRecord.RESTORE_FULL)
                    || (mBackupTarget.backupMode == BackupRecord.BACKUP_FULL);
        }

        ensurePackageDexOpt(app.instrumentationInfo != null
                ? app.instrumentationInfo.packageName
                : app.info.packageName);
        if (app.instrumentationClass != null) {
            ensurePackageDexOpt(app.instrumentationClass.getPackageName());
        }
        if (DEBUG_CONFIGURATION) Slog.v(TAG, "Binding proc "
                + processName + " with config " + mConfiguration);
        ApplicationInfo appInfo = app.instrumentationInfo != null
                ? app.instrumentationInfo : app.info;
        app.compat = compatibilityInfoForPackageLocked(appInfo);
        if (profileFd != null) {
            profileFd = profileFd.dup();
        }
        thread.bindApplication(processName, appInfo, providers,
                app.instrumentationClass, profileFile, profileFd, profileAutoStop,
                app.instrumentationArguments, app.instrumentationWatcher,
                app.instrumentationUiAutomationConnection, testMode,
                enableOpenGlTrace,
                isRestrictedBackupMode || !normalMode, app.persistent,
                new Configuration(mConfiguration),app.compat,getCommonServices Locked(),
                mCoreSettingsObserver.getCoreSettingsLocked());
        updateLruProcessLocked(app, false);
        app.lastRequestedGc = app.lastLowMemory = SystemClock.uptimeMillis();
    } catch (Exception e) {
        // todo: Yikes!  What should we do?  For now we will try to
        // start another process, but that could easily get us in
        // an infinite loop of restarting processes...
        Slog.w(TAG, "Exception thrown during bind!", e);

        app.resetPackageList();
        app.unlinkDeathRecipient();
        startProcessLocked(app, "bind fail", processName);
        return false;
    }

    // Remove this record from the list of starting applications.
    mPersistentStartingProcesses.remove(app);
    if (DEBUG_PROCESSES && mProcessesOnHold.contains(app)) Slog.v(TAG,
```

```
            "Attach application locked removing on hold: " + app);
        mProcessesOnHold.remove(app);

        boolean badApp = false;
        boolean didSomething = false;

        // See if the top visible activity is waiting to run in this process...
        ActivityRecord hr = mMainStack.topRunningActivityLocked(null);
        if (hr != null && normalMode) {
            if (hr.app == null && app.uid == hr.info.applicationInfo.uid
                    && processName.equals(hr.processName)) {
                try {
                    if (mHeadless) {
                        Slog.e(TAG, "Starting activities not supported on headless device:
                        " + hr);
                    } else if (mMainStack.realStartActivityLocked(hr, app, true, true)) {
                        didSomething = true;
                    }
                } catch (Exception e) {
                    Slog.w(TAG, "Exception in new application when starting activity "
                          + hr.intent.getComponent().flattenToShortString(), e);
                    badApp = true;
                }
            } else {
                mMainStack.ensureActivitiesVisibleLocked(hr, null, processName, 0);
            }
        }

        // Find any services that should be running in this process...
        if (!badApp) {
            try {
                didSomething |= mServices.attachApplicationLocked(app, processName);
            } catch (Exception e) {
                badApp = true;
            }
        }

        // Check if a next-broadcast receiver is in this process...
        if (!badApp && isPendingBroadcastProcessLocked(pid)) {
            try {
                didSomething = sendPendingBroadcastsLocked(app);
            } catch (Exception e) {
                // If the app died trying to launch the receiver we declare it 'bad'
                badApp = true;
            }
        }

        // Check whether the next backup agent is in this process...
        if (!badApp && mBackupTarget != null && mBackupTarget.appInfo.uid == app.uid) {
            if (DEBUG_BACKUP) Slog.v(TAG, "New app is backup target, launching agent for
            " + app);
            ensurePackageDexOpt(mBackupTarget.appInfo.packageName);
            try {
                thread.scheduleCreateBackupAgent(mBackupTarget.appInfo,
                        compatibilityInfoForPackageLocked(mBackupTarget.appInfo),
                        mBackupTarget.backupMode);
            } catch (Exception e) {
                Slog.w(TAG, "Exception scheduling backup agent creation: ");
                e.printStackTrace();
            }
        }

        if (badApp) {
            // todo: Also need to kill application to deal with all
            // kinds of exceptions.
            handleAppDiedLocked(app, false, true);
            return false;
        }

        if (!didSomething) {
            updateOomAdjLocked();
```

```
        }

        return true;
    }
```

在上述代码中，调用函数 generateApplicationProvidersLocked 获取需要在 ProcessRecord 对象 app 描述的程序进程中启动的 Content Provider 组件。函数 generateApplicationProvidersLocked 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示：

```
private final List<ProviderInfo> generateApplicationProvidersLocked(ProcessRecord app)
{
    List<ProviderInfo> providers = null;
    try {
        providers = AppGlobals.getPackageManager().
            queryContentProviders(app.processName, app.uid,
                    STOCK_PM_FLAGS | PackageManager.GET_URI_PERMISSION_PATTERNS);
    } catch (RemoteException ex) {
    }
    if (DEBUG_MU)
        Slog.v(TAG_MU, "generateApplicationProvidersLocked, app.info.uid = " + app.uid);
    int userId = app.userId;
    if (providers != null) {
        int N = providers.size();
        for (int i=0; i<N; i++) {
            ProviderInfo cpi =
                (ProviderInfo)providers.get(i);
            boolean singleton = isSingleton(cpi.processName, cpi.applicationInfo,
                    cpi.name, cpi.flags);
            if (singleton && UserHandle.getUserId(app.uid) != 0) {
                providers.remove(i);
                N--;
                continue;
            }

            ComponentName comp = new ComponentName(cpi.packageName, cpi.name);
            ContentProviderRecord cpr = mProviderMap.getProviderByClass(comp, userId);
            if (cpr == null) {
                cpr = new ContentProviderRecord(this, cpi, app.info, comp, singleton);
                mProviderMap.putProviderByClass(comp, cpr);
            }
            if (DEBUG_MU)
                Slog.v(TAG_MU,"generateApplicationProvidersLocked, cpi.uid = " + cpr.uid);
            app.pubProviders.put(cpi.name, cpr);
            app.addPackage(cpi.applicationInfo.packageName);
            ensurePackageDexOpt(cpi.applicationInfo.packageName);
        }
    }
    return providers;
}
```

12.2.5　处理消息

再看函数 bindApplication，功能是把相关的信息都封装成一个 AppBindData 对象，然后以一个消息的形式发送到主线程的消息队列中去等待处理。这个消息最终在类 ActivityThread 的函数 handleBindApplication 中进行处理。函数 bindApplication 在文件 frameworks\base\core\java\android\app\ApplicationThreadNative.java 中定义，具体实现代码如下所示：

```
public final void bindApplication(String packageName, ApplicationInfo info,
        List<ProviderInfo> providers, ComponentName testName, String profileName,
        ParcelFileDescriptor profileFd, boolean autoStopProfiler, Bundle testArgs,
        IInstrumentationWatcher testWatcher,
        IUiAutomationConnection uiAutomationConnection, int debugMode,
        boolean openGlTrace, boolean restrictedBackupMode, boolean persistent,
        Configuration config, CompatibilityInfo compatInfo, Map<String, IBinder>
services,
        Bundle coreSettings) throws RemoteException {
    Parcel data = Parcel.obtain();
```

```
        data.writeInterfaceToken(IApplicationThread.descriptor);
        data.writeString(packageName);
        info.writeToParcel(data, 0);
        data.writeTypedList(providers);
        if (testName == null) {
            data.writeInt(0);
        } else {
            data.writeInt(1);
            testName.writeToParcel(data, 0);
        }
        data.writeString(profileName);
        if (profileFd != null) {
            data.writeInt(1);
            profileFd.writeToParcel(data, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            data.writeInt(0);
        }
        data.writeInt(autoStopProfiler ? 1 : 0);
        data.writeBundle(testArgs);
        data.writeStrongInterface(testWatcher);
        data.writeStrongInterface(uiAutomationConnection);
        data.writeInt(debugMode);
        data.writeInt(openGlTrace ? 1 : 0);
        data.writeInt(restrictedBackupMode ? 1 : 0);
        data.writeInt(persistent ? 1 : 0);
        config.writeToParcel(data, 0);
        compatInfo.writeToParcel(data, 0);
        data.writeMap(services);
        data.writeBundle(coreSettings);
        mRemote.transact(BIND_APPLICATION_TRANSACTION, data, null,
                IBinder.FLAG_ONEWAY);
        data.recycle();
    }
```

再看文件 frameworks\base\core\java\android\app\ApplicationThreadNative.java 中的函数 bindApplication，具体实现代码如下所示：

```
    public final void bindApplication(String processName,
            ApplicationInfo appInfo, List<ProviderInfo> providers,
            ComponentName instrumentationName, String profileFile,
            ParcelFileDescriptor profileFd, boolean autoStopProfiler,
            Bundle instrumentationArgs, IInstrumentationWatcher instrumentationWatcher,
            IUiAutomationConnection instrumentationUiConnection, int debugMode,
            boolean enableOpenGlTrace, boolean isRestrictedBackupMode, boolean persistent,
            Configuration config, CompatibilityInfo compatInfo, Map<String, IBinder> services,
            Bundle coreSettings) {

        if (services != null) {
            // Setup the service cache in the ServiceManager
            ServiceManager.initServiceCache(services);
        }

        setCoreSettings(coreSettings);

        AppBindData data = new AppBindData();
        data.processName = processName;
        data.appInfo = appInfo;
        data.providers = providers;
        data.instrumentationName = instrumentationName;
        data.instrumentationArgs = instrumentationArgs;
        data.instrumentationWatcher = instrumentationWatcher;
        data.instrumentationUiAutomationConnection = instrumentationUiConnection;
        data.debugMode = debugMode;
        data.enableOpenGlTrace = enableOpenGlTrace;
        data.restrictedBackupMode = isRestrictedBackupMode;
        data.persistent = persistent;
        data.config = config;
        data.compatInfo = compatInfo;
        data.initProfileFile = profileFile;
        data.initProfileFd = profileFd;
        data.initAutoStopProfiler = false;
```

```
        queueOrSendMessage(H.BIND_APPLICATION, data);
    }
```

12.2.6　具体启动

接下来开始步入具体启动步骤。首先看函数 handleBindApplication，功能是将消息中的 Content Provider 组件启动起来。函数 handleBindApplication 在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private final void handleBindApplication(AppBindData data) {
    ......

    List<ProviderInfo> providers = data.providers;
    if (providers != null) {
        installContentProviders(app, providers);
        ......
    }
```

再看函数 installContentProviders，功能是启动这些 Content Provider 组件，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，具体实现代码如下所示：

```
private void installContentProviders(
        Context context, List<ProviderInfo> providers) {
    final ArrayList<IActivityManager.ContentProviderHolder> results =
        new ArrayList<IActivityManager.ContentProviderHolder>();

    for (ProviderInfo cpi : providers) {
        if (DEBUG_PROVIDER) {
            StringBuilder buf = new StringBuilder(128);
            buf.append("Pub ");
            buf.append(cpi.authority);
            buf.append(": ");
            buf.append(cpi.name);
            Log.i(TAG, buf.toString());
        }
        IActivityManager.ContentProviderHolder cph = installProvider(context, null, cpi,
                false /*noisy*/, true /*noReleaseNeeded*/, true /*stable*/);
        if (cph != null) {
            cph.noReleaseNeeded = true;
            results.add(cph);
        }
    }

    try {
        ActivityManagerNative.getDefault().publishContentProviders(
            getApplicationThread(), results);
    } catch (RemoteException ex) {
    }
}
```

由此可见，函数 installContentProviders 调用 installProvider 在本地安装了每一个 Content Proivder 的信息，并且为每一个 Content Provider 创建了一个 ContentProviderHolder 对象来保存相关的信息。ContentProviderHolder 对象是一个 Binder 对象，功能是把 Content Provider 的信息传递给 ActivityManagerService 服务。当处理完 Content Provider 后，还需要调用 ActivityManagerService 服务中的函数 publishContentProviders 来通知 ActivityManagerService 服务在这个进程中需要加载的 Content Provider。

下面看函数 attachInfo，功能是初始化前面创建的 Content Provider 组件。此函数在文件 frameworks/base/core/java/android/content/ContentProvider.java 中定义，具体实现代码如下所示：

```
private void attachInfo(Context context, ProviderInfo info, boolean testing) {
    /*
     * We may be using AsyncTask from binder threads.  Make it init here
     * so its static handler is on the main thread.
     */
    AsyncTask.init();
```

```
        mNoPerms = testing;

        /*
         * Only allow it to be set once, so after the content service gives
         * this to us clients can't change it.
         */
        if (mContext == null) {
            mContext = context;
            mMyUid = Process.myUid();
            if (info != null) {
                setReadPermission(info.readPermission);
                setWritePermission(info.writePermission);
                setPathPermissions(info.pathPermissions);
                mExported = info.exported;
            }
            ContentProvider.this.onCreate();
        }
    }
```

在上述代码中，根据 Content Provider 的信息 info 来设置相应的读写权限，然后调用其子类中的函数 onCreate 让子类执行初始化工作。

接下来需要自定义编写函数 onCreate 实现业务初始化操作，例如下面的代码：

```
public boolean onCreate() {
    Context context = getContext();
    resolver = context.getContentResolver();
    dbHelper = new DBHelper(context, DB_NAME, null, DB_VERSION);
    return true;
}
```

最后看函数 publishContentProviders，功能是将刚启动的 Content Provider 的接口发布到 ActivityManagerService 服务中。函数 publishContentProviders 在文件 frameworks/base/core/java/android/app/ActivityManageNative.java 中定义，具体实现代码如下所示：

```
public void publishContentProviders(IApplicationThread caller,
        List<ContentProviderHolder> providers) throws RemoteException
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    data.writeTypedList(providers);
    mRemote.transact(PUBLISH_CONTENT_PROVIDERS_TRANSACTION, data, reply, 0);
    reply.readException();
    data.recycle();
    reply.recycle();
}
```

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 publishContentProviders，具体实现代码如下所示：

```
    public final void publishContentProviders(IApplicationThread caller,
            List<ContentProviderHolder> providers) {
        if (providers == null) {
            return;
        }

        enforceNotIsolatedCaller("publishContentProviders");
        synchronized (this) {
            final ProcessRecord r = getRecordForAppLocked(caller);
            if (DEBUG_MU)
                Slog.v(TAG_MU, "ProcessRecord uid = " + r.uid);
            if (r == null) {
                throw new SecurityException(
                        "Unable to find app for caller " + caller
                      + " (pid=" + Binder.getCallingPid()
                      + ") when publishing content providers");
            }

            final long origId = Binder.clearCallingIdentity();
```

```
            final int N = providers.size();
            for (int i=0; i<N; i++) {
                ContentProviderHolder src = providers.get(i);
                if (src == null || src.info == null || src.provider == null) {
                    continue;
                }
                ContentProviderRecord dst = r.pubProviders.get(src.info.name);
                if (DEBUG_MU)
                    Slog.v(TAG_MU, "ContentProviderRecord uid = " + dst.uid);
                if (dst != null) {
                    ComponentName comp = new ComponentName(dst.info.packageName,
                    dst.info.name);
//把这个 Content Provider 信息保存在 mProvidersByClass 和 mProvidersByName 中
                    mProviderMap.putProviderByClass(comp, dst);
                    String names[] = dst.info.authority.split(";");
                    for (int j = 0; j < names.length; j++) {
                        mProviderMap.putProviderByName(names[j], dst);
                    }
//因为这个 Content Provider 已经加载好了，因此，把它从 mLaunchingProviders 列表中删除
                    int NL = mLaunchingProviders.size();
                    int j;
                    for (j=0; j<NL; j++) {
                        if (mLaunchingProviders.get(j) == dst) {
                            mLaunchingProviders.remove(j);
                            j--;
                            NL--;
                        }
                    }
//设置这个 ContentProviderRecord 对象 dst 的 provider 域为从参数传进来的 Content Provider 远程接口
                    synchronized (dst) {
                        dst.provider = src.provider;
                        dst.proc = r;
                        dst.notifyAll();
                    }
                    updateOomAdjLocked(r);
                }
            }
            Binder.restoreCallingIdentity(origId);
        }
    }
```

12.3 Content Provider 数据共享

在 Android 系统中，Content Provider 的最核心功能便是数据共享，提供了一个保存数据的作用。Content Provider 组件可以在不同的应用程序之间传输数据，这一功能基于匿名共享内存机制来实现。在 Android 应用程序中，通常将共享数据保存在 SQLite 中，Content Provider 借助于 SQLite 数据库游标（SQLiteCursor）来实现最终的数据共享。在本节的内容中，将详细分析 Content Provider 实现数据共享的源代码。

12.3.1　获取接口

首先看函数 query，功能是调用函数 acquireProvider 获得与参数 uri 对应的 Content Provider 接口，并通过这个接口中的函数 query 来获取相应的数据。函数 query 在文件 frameworks/base/core/java/android/content/ContentResolver.java 中定义，具体实现代码如下所示：

```
    public final Cursor query(final Uri uri, String[] projection,
            String selection, String[] selectionArgs, String sortOrder,
            CancellationSignal cancellationSignal) {
        IContentProvider unstableProvider = acquireUnstableProvider(uri);
        if (unstableProvider == null) {
            return null;
        }
        IContentProvider stableProvider = null;
        Cursor qCursor = null;
```

```
        try {
            long startTime = SystemClock.uptimeMillis();

            ICancellationSignal remoteCancellationSignal = null;
            if (cancellationSignal != null) {
                cancellationSignal.throwIfCanceled();
                remoteCancellationSignal = unstableProvider.createCancellationSignal();
                cancellationSignal.setRemote(remoteCancellationSignal);
            }
            try {
                qCursor = unstableProvider.query(mPackageName, uri, projection,
                        selection, selectionArgs, sortOrder, remoteCancellationSignal);
            } catch (DeadObjectException e) {
                // The remote process has died...  but we only hold an unstable
                // reference though, so we might recover!!!  Let's try!!!!
                // This is exciting!!1!!1!!!!1
                unstableProviderDied(unstableProvider);
//获得与参数 uri 对应的 Content Provider 接口
                stableProvider = acquireProvider(uri);
                if (stableProvider == null) {
                    return null;
                }
                qCursor = stableProvider.query(mPackageName, uri, projection,
                        selection, selectionArgs, sortOrder, remoteCancellationSignal);
            }
            if (qCursor == null) {
                return null;
            }

            // Force query execution.  Might fail and throw a runtime exception here.
            qCursor.getCount();
            long durationMillis = SystemClock.uptimeMillis() - startTime;
            maybeLogQueryToEventLog(durationMillis, uri, projection, selection, sortOrder);

            // Wrap the cursor object into CursorWrapperInner object.
            CursorWrapperInner wrapper = new CursorWrapperInner(qCursor,
                    stableProvider != null ? stableProvider : acquireProvider(uri));
            stableProvider = null;
            qCursor = null;
            return wrapper;
        } catch (RemoteException e) {
            // Arbitrary and not worth documenting, as Activity
            // Manager will kill this process shortly anyway.
            return null;
        } finally {
            if (qCursor != null) {
                qCursor.close();
            }
            if (unstableProvider != null) {
                releaseUnstableProvider(unstableProvider);
            }
            if (stableProvider != null) {
                releaseProvider(stableProvider);
            }
        }
    }
```

在上述代码中，会调用返回来的 Content Provider 接口来获取数据。这个 Content Provider 接口实际上是在类 ContentProvider 内部所创建的一个 Transport 对象的远程接口。类 Transport 继承于类 ContentProviderNative，是一个 Binder 对象的 Stub 类。

接下来需要依次调用如下所示的函数：

- ContentResolver.acquireProvider;
- ApplicationContentResolver.acquireProvider;
- ActivityThread.acquireProvider;
- ActivityThread.getProvider。

有关上述函数的具体实现过程和具体分析，请读者参阅本章上一节中的内容。接下来进入到

这个 Binder 对象的 Proxy 类 ContentProviderProxy 中执行函数 query。在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中，函数 query 的具体实现代码如下所示：

```
public Cursor query(String callingPkg, Uri url, String[] projection, String selection,
        String[] selectionArgs, String sortOrder, ICancellationSignal cancellation Signal)
                throws RemoteException {
    BulkCursorToCursorAdaptor adaptor = new BulkCursorToCursorAdaptor();
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    try {
        data.writeInterfaceToken(IContentProvider.descriptor);

        data.writeString(callingPkg);
        url.writeToParcel(data, 0);
        int length = 0;
        if (projection != null) {
            length = projection.length;
        }
        data.writeInt(length);
        for (int i = 0; i < length; i++) {
            data.writeString(projection[i]);
        }
        data.writeString(selection);
        if (selectionArgs != null) {
            length = selectionArgs.length;
        } else {
            length = 0;
        }
        data.writeInt(length);
        for (int i = 0; i < length; i++) {
            data.writeString(selectionArgs[i]);
        }
        data.writeString(sortOrder);
        data.writeStrongBinder(adaptor.getObserver().asBinder());
        data.writeStrongBinder(cancellationSignal != null ? cancellationSignal.
        asBinder() : null);

        mRemote.transact(IContentProvider.QUERY_TRANSACTION, data, reply, 0);

        DatabaseUtils.readExceptionFromParcel(reply);

        if (reply.readInt() != 0) {
            BulkCursorDescriptor d = BulkCursorDescriptor.CREATOR.createFromParcel(reply);
            adaptor.initialize(d);
        } else {
            adaptor.close();
            adaptor = null;
        }
        return adaptor;
    } catch (RemoteException ex) {
        adaptor.close();
        throw ex;
    } catch (RuntimeException ex) {
        adaptor.close();
        throw ex;
    } finally {
        data.recycle();
        reply.recycle();
    }
}
```

在上述代码中，先创建了一个 CursorWindow 对象，在此 CursorWindow 对象中包含了一块匿名共享内存，其作用是把这块匿名共享内存通过 Binder 进程间通信机制传给 Content Proivder，这样可以让 Content Proivder 在里面返回所请求的数据。

12.3.2　创建 CursorWindow 对象

首先看类 CursorWindow，在文件 frameworks/base/core/java/android/database/CursorWindow.java

中定义，主要实现代码如下所示：

```
public class CursorWindow extends SQLiteClosable implements Parcelable {
    private static final String STATS_TAG = "CursorWindowStats";
    private static final int sCursorWindowSize =
        Resources.getSystem().getInteger(
                com.android.internal.R.integer.config_cursorWindowSize) * 1024;
    public int mWindowPtr;

    private int mStartPos;
    private final String mName;

    private final CloseGuard mCloseGuard = CloseGuard.get();

    private static native int nativeCreate(String name, int cursorWindowSize);
    private static native int nativeCreateFromParcel(Parcel parcel);
    private static native void nativeDispose(int windowPtr);
    private static native void nativeWriteToParcel(int windowPtr, Parcel parcel);

    private static native void nativeClear(int windowPtr);

    private static native int nativeGetNumRows(int windowPtr);
    private static native boolean nativeSetNumColumns(int windowPtr, int columnNum);
    private static native boolean nativeAllocRow(int windowPtr);
    private static native void nativeFreeLastRow(int windowPtr);

    private static native int nativeGetType(int windowPtr, int row, int column);
    private static native byte[] nativeGetBlob(int windowPtr, int row, int column);
    private static native String nativeGetString(int windowPtr, int row, int column);
    private static native long nativeGetLong(int windowPtr, int row, int column);
    private static native double nativeGetDouble(int windowPtr, int row, int column);
    private static native void nativeCopyStringToBuffer(int windowPtr, int row, int
    column, CharArrayBuffer buffer);

    private static native boolean nativePutBlob(int windowPtr, byte[] value, int row, int column);
    private static native boolean nativePutString(int windowPtr, String value, int row,
    int column);
    private static native boolean nativePutLong(int windowPtr, long value, int row, int column);
    private static native boolean nativePutDouble(int windowPtr, double value, int row,
    int column);
    private static native boolean nativePutNull(int windowPtr, int row, int column);

    private static native String nativeGetName(int windowPtr);
    public CursorWindow(String name)
```

在上述代码中，调用了本地函数 native_init 来实现初始化的工作，初始化的过程就是创建匿名共享内存的过程。在这过程中传进来的参数 localWindow 的值为 false，表示这个匿名共享内存只能通过远程调用来访问，可以通过 Content Proivder 返回的接口 Cursor 来访问这块匿名共享内存里面的数据。

本地函数 native_init 和文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中的函数 JNINativeMethod 相对应，具体实现代码如下所示：

```
static JNINativeMethod sMethods[] =
{
    /* name, signature, funcPtr */
    { "nativeCreate", "(Ljava/lang/String;I)I",
          (void*)nativeCreate },
    { "nativeCreateFromParcel", "(Landroid/os/Parcel;)I",
          (void*)nativeCreateFromParcel },
    { "nativeDispose", "(I)V",
          (void*)nativeDispose },
    { "nativeWriteToParcel", "(ILandroid/os/Parcel;)V",
          (void*)nativeWriteToParcel },
    { "nativeGetName", "(I)Ljava/lang/String;",
          (void*)nativeGetName },
    { "nativeClear", "(I)V",
          (void*)nativeClear },
    { "nativeGetNumRows", "(I)I",
          (void*)nativeGetNumRows },
```

```
    { "nativeSetNumColumns", "(II)Z",
          (void*)nativeSetNumColumns },
    { "nativeAllocRow", "(I)Z",
          (void*)nativeAllocRow },
    { "nativeFreeLastRow", "(I)V",
          (void*)nativeFreeLastRow },
    { "nativeGetType", "(III)I",
          (void*)nativeGetType },
    { "nativeGetBlob", "(III)[B",
          (void*)nativeGetBlob },
    { "nativeGetString", "(III)Ljava/lang/String;",
          (void*)nativeGetString },
    { "nativeGetLong", "(III)J",
          (void*)nativeGetLong },
    { "nativeGetDouble", "(III)D",
          (void*)nativeGetDouble },
    { "nativeCopyStringToBuffer", "(IIILandroid/database/CharArrayBuffer;)V",
          (void*)nativeCopyStringToBuffer },
    { "nativePutBlob", "(I[BII)Z",
          (void*)nativePutBlob },
    { "nativePutString", "(ILjava/lang/String;II)Z",
          (void*)nativePutString },
    { "nativePutLong", "(IJII)Z",
          (void*)nativePutLong },
    { "nativePutDouble", "(IDII)Z",
          (void*)nativePutDouble },
    { "nativePutNull", "(III)Z",
          (void*)nativePutNull },
};
```

在上述代码中调用了函数 nativeCreate，此函数在 C++层创建了一个 CursorWindow 对象，并通过调用宏 LOG_WINDOW 将这个 C++层的 CursorWindow 对象与 Java 层的 CursorWindow 对象关联起来。

函数 nativeCreate 也是在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static jint nativeCreate(JNIEnv* env, jclass clazz, jstring nameObj, jint
cursorWindowSize) {
    String8 name;
    const char* nameStr = env->GetStringUTFChars(nameObj, NULL);
    name.setTo(nameStr);
    env->ReleaseStringUTFChars(nameObj, nameStr);

    CursorWindow* window;
    status_t status = CursorWindow::create(name, cursorWindowSize, &window);
    if (status || !window) {
        ALOGE("Could not allocate CursorWindow '%s' of size %d due to error %d.",
              name.string(), cursorWindowSize, status);
        return 0;
    }

    LOG_WINDOW("nativeInitializeEmpty: window = %p", window);
    return reinterpret_cast<jint>(window);
}
```

再看函数 register_android_database_CursorWindow，其功能是初始化用于描述 Java 层的类 CursorWindowc 成员变量 nWindow 在类内部的偏移量。函数 register_android_database_CursorWindow 也是在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
int register_android_database_CursorWindow(JNIEnv * env)
{
    jclass clazz;
    FIND_CLASS(clazz, "android/database/CharArrayBuffer");

    GET_FIELD_ID(gCharArrayBufferClassInfo.data, clazz,
          "data", "[C");
    GET_FIELD_ID(gCharArrayBufferClassInfo.sizeCopied, clazz,
```

```
            "sizeCopied", "I");

    gEmptyString = jstring(env->NewGlobalRef(env->NewStringUTF("")));
    LOG_FATAL_IF(!gEmptyString, "Unable to create empty string");

    return AndroidRuntime::registerNativeMethods(env, "android/database/CursorWindow",
            sMethods, NELEM(sMethods));
}
```

12.3.3 数据传递

在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中的函数 query 中，通过如下代码将创建的匿名共享内存传递给了我们的应用组件，这样应用组件可以和参数 url 对应的信息保存在里面：

```
BulkCursorDescriptor d = BulkCursorDescriptor.CREATOR.createFromParcel(reply);
```

类 BulkCursorDescriptor 在文件 frameworks\base\core\java\android\database\BulkCursor Descriptor.java 中定义，具体实现代码如下所示：

```
public final class BulkCursorDescriptor implements Parcelable {
    public static final Parcelable.Creator<BulkCursorDescriptor> CREATOR =
            new Parcelable.Creator<BulkCursorDescriptor>() {
        @Override
        public BulkCursorDescriptor createFromParcel(Parcel in) {
            BulkCursorDescriptor d = new BulkCursorDescriptor();
            d.readFromParcel(in);
            return d;
        }

        @Override
        public BulkCursorDescriptor[] newArray(int size) {
            return new BulkCursorDescriptor[size];
        }
    };

    public IBulkCursor cursor;
    public String[] columnNames;
    public boolean wantsAllOnMoveCalls;
    public int count;
    public CursorWindow window;

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeStrongBinder(cursor.asBinder());
        out.writeStringArray(columnNames);
        out.writeInt(wantsAllOnMoveCalls ? 1 : 0);
        out.writeInt(count);
        if (window != null) {
            out.writeInt(1);
            window.writeToParcel(out, flags);
        } else {
            out.writeInt(0);
        }
    }

    public void readFromParcel(Parcel in) {
        cursor = BulkCursorNative.asInterface(in.readStrongBinder());
        columnNames = in.readStringArray();
        wantsAllOnMoveCalls = in.readInt() != 0;
        count = in.readInt();
        if (in.readInt() != 0) {
            window = CursorWindow.CREATOR.createFromParcel(in);
        }
```

```
    }
}
```

在上述代码中，调用 window 中的函数 writeToParcel，把 window 对象内部的匿名共享内存块通过 Binder 进程间通信机制传输给 Content Provider 使用。当传进来的参数 adaptor 不为 null 时，会向 data 中写入整数 1，表示让 Content Provider 返回查询得到数据的元信息。

在文件 frameworks/base/core/java/android/database/CursorWindow.java 中，函数 writeToParcel 的具体实现代码如下所示：

```
public void writeToParcel(Parcel dest, int flags) {
    acquireReference();
    try {
        dest.writeInt(mStartPos);
        nativeWriteToParcel(mWindowPtr, dest);
    } finally {
        releaseReference();
    }
    if ((flags & Parcelable.PARCELABLE_WRITE_RETURN_VALUE) != 0) {
        releaseReference();
    }
}
```

在上述代码中，调用了函数 nativeWriteToParcel 来获取一个 Binder 本地对象，然后将这个对象写到 dest 对象中。函数 nativeWriteToParcel 在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static void nativeWriteToParcel(JNIEnv * env, jclass clazz, jint windowPtr,
        jobject parcelObj) {
// 获得关联的 C++层的 CursorWindow 对象 window
    CursorWindow* window = reinterpret_cast<CursorWindow*>(windowPtr);
    Parcel* parcel = parcelForJavaObject(env, parcelObj);

    status_t status = window->writeToParcel(parcel);
    if (status) {
        String8 msg;
        msg.appendFormat("Could not write CursorWindow to Parcel due to error %d.",
status);
        jniThrowRuntimeException(env, msg.string());
    }
}
```

在上述代码中，参数 parcelObj 指向 Java 层的 CursorWindow 对象。

12.3.4　处理进程通信的请求

函数 onTransact 的功能是处理类型为 IContentProvider.QUERY_TRANSACTION 进程间的通信请求。函数 onTransact 在文件 frameworks/base/core/java/android/content/ContentProviderNative.java 中定义，具体实现代码如下所示：

```
public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
    try {
        switch (code) {
            case QUERY_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);

                String callingPkg = data.readString();
                Uri url = Uri.CREATOR.createFromParcel(data);

                // String[] projection
                int num = data.readInt();
                String[] projection = null;
                if (num > 0) {
                    projection = new String[num];
                    for (int i = 0; i < num; i++) {
                        projection[i] = data.readString();
```

```
            }
        }

        // String selection, String[] selectionArgs...
        String selection = data.readString();
        num = data.readInt();
        String[] selectionArgs = null;
        if (num > 0) {
            selectionArgs = new String[num];
            for (int i = 0; i < num; i++) {
                selectionArgs[i] = data.readString();
            }
        }

        String sortOrder = data.readString();
        IContentObserver observer = IContentObserver.Stub.asInterface(
                data.readStrongBinder());
        ICancellationSignal cancellationSignal = ICancellationSignal.Stub. asInterface(
                data.readStrongBinder());

        Cursor cursor = query(callingPkg, url, projection, selection, selectionArgs,
                sortOrder, cancellationSignal);
        if (cursor != null) {
            try {
                CursorToBulkCursorAdaptor adaptor = new CursorToBulkCursorAdaptor(
                        cursor, observer, getProviderName());
                BulkCursorDescriptor d = adaptor.getBulkCursorDescriptor();
                cursor = null;

                reply.writeNoException();
                reply.writeInt(1);
                d.writeToParcel(reply, Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
            } finally {
                // Close cursor if an exception was thrown while constructing the adaptor.
                if (cursor != null) {
                    cursor.close();
                }
            }
        } else {
            reply.writeNoException();
            reply.writeInt(0);
        }

        return true;
    }

    case GET_TYPE_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        Uri url = Uri.CREATOR.createFromParcel(data);
        String type = getType(url);
        reply.writeNoException();
        reply.writeString(type);

        return true;
    }

    case INSERT_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        String callingPkg = data.readString();
        Uri url = Uri.CREATOR.createFromParcel(data);
        ContentValues values = ContentValues.CREATOR.createFromParcel(data);

        Uri out = insert(callingPkg, url, values);
        reply.writeNoException();
        Uri.writeToParcel(reply, out);
        return true;
    }

    case BULK_INSERT_TRANSACTION:
```

```
        {
            data.enforceInterface(IContentProvider.descriptor);
            String callingPkg = data.readString();
            Uri url = Uri.CREATOR.createFromParcel(data);
            ContentValues[] values = data.createTypedArray(ContentValues.CREATOR);

            int count = bulkInsert(callingPkg, url, values);
            reply.writeNoException();
            reply.writeInt(count);
            return true;
        }

        case APPLY_BATCH_TRANSACTION:
        {
            data.enforceInterface(IContentProvider.descriptor);
            String callingPkg = data.readString();
            final int numOperations = data.readInt();
            final ArrayList<ContentProviderOperation> operations =
                    new ArrayList<ContentProviderOperation>(numOperations);
            for (int i = 0; i < numOperations; i++) {
                operations.add(i, ContentProviderOperation.CREATOR.createFromParcel
                (data));
            }
            final ContentProviderResult[] results = applyBatch(callingPkg, operations);
            reply.writeNoException();
            reply.writeTypedArray(results, 0);
            return true;
        }

        case DELETE_TRANSACTION:
        {
            data.enforceInterface(IContentProvider.descriptor);
            String callingPkg = data.readString();
            Uri url = Uri.CREATOR.createFromParcel(data);
            String selection = data.readString();
            String[] selectionArgs = data.readStringArray();

            int count = delete(callingPkg, url, selection, selectionArgs);

            reply.writeNoException();
            reply.writeInt(count);
            return true;
        }

        case UPDATE_TRANSACTION:
        {
            data.enforceInterface(IContentProvider.descriptor);
            String callingPkg = data.readString();
            Uri url = Uri.CREATOR.createFromParcel(data);
            ContentValues values = ContentValues.CREATOR.createFromParcel(data);
            String selection = data.readString();
            String[] selectionArgs = data.readStringArray();

            int count = update(callingPkg, url, values, selection, selectionArgs);

            reply.writeNoException();
            reply.writeInt(count);
            return true;
        }

        case OPEN_FILE_TRANSACTION:
        {
            data.enforceInterface(IContentProvider.descriptor);
            String callingPkg = data.readString();
            Uri url = Uri.CREATOR.createFromParcel(data);
            String mode = data.readString();

            ParcelFileDescriptor fd;
            fd = openFile(callingPkg, url, mode);
            reply.writeNoException();
            if (fd != null) {
```

```
            reply.writeInt(1);
            fd.writeToParcel(reply,
                    Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            reply.writeInt(0);
        }
        return true;
    }

    case OPEN_ASSET_FILE_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        String callingPkg = data.readString();
        Uri url = Uri.CREATOR.createFromParcel(data);
        String mode = data.readString();

        AssetFileDescriptor fd;
        fd = openAssetFile(callingPkg, url, mode);
        reply.writeNoException();
        if (fd != null) {
            reply.writeInt(1);
            fd.writeToParcel(reply,
                    Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
        } else {
            reply.writeInt(0);
        }
        return true;
    }

    case CALL_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);

        String callingPkg = data.readString();
        String method = data.readString();
        String stringArg = data.readString();
        Bundle args = data.readBundle();

        Bundle responseBundle = call(callingPkg, method, stringArg, args);

        reply.writeNoException();
        reply.writeBundle(responseBundle);
        return true;
    }

    case GET_STREAM_TYPES_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        Uri url = Uri.CREATOR.createFromParcel(data);
        String mimeTypeFilter = data.readString();
        String[] types = getStreamTypes(url, mimeTypeFilter);
        reply.writeNoException();
        reply.writeStringArray(types);

        return true;
    }

    case OPEN_TYPED_ASSET_FILE_TRANSACTION:
    {
        data.enforceInterface(IContentProvider.descriptor);
        String callingPkg = data.readString();
        Uri url = Uri.CREATOR.createFromParcel(data);
        String mimeType = data.readString();
        Bundle opts = data.readBundle();

        AssetFileDescriptor fd;
        fd = openTypedAssetFile(callingPkg, url, mimeType, opts);
        reply.writeNoException();
        if (fd != null) {
            reply.writeInt(1);
            fd.writeToParcel(reply,
```

```
                        Parcelable.PARCELABLE_WRITE_RETURN_VALUE);
                } else {
                    reply.writeInt(0);
                }
                return true;
            }

            case CREATE_CANCELATION_SIGNAL_TRANSACTION:
            {
                data.enforceInterface(IContentProvider.descriptor);

                ICancellationSignal cancellationSignal = createCancellationSignal();
                reply.writeNoException();
                reply.writeStrongBinder(cancellationSignal.asBinder());
                return true;
            }
        }
    } catch (Exception e) {
        DatabaseUtils.writeExceptionToParcel(reply, e);
        return true;
    }

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

在上述代码中，函数 createFromParcel 从数据流 data 中重建一个本地的 CursorWindow 对象，然后将数据流 data 的下一个整数值读取出来。如果这个整数值不为 0，则变量 wantsCursorMetadata 的值为 true，表示 Content Provider 在返回 IBulkCursor 接口给第三方应用程序之前要先执行一次数据库查询操作，这样可以把结果数据的元信息返回给第三方应用程序。

下面看函数 createFromParcel，其功能是将倒数第二个进程间的通信数据封装成一个 CursorWindow 对象。函数 createFromParcel 在文件 frameworks/base/core/java/android/content/CursorWindow.java 中定义，具体实现代码如下所示：

```
public static final Parcelable.Creator<CursorWindow> CREATOR
        = new Parcelable.Creator<CursorWindow>() {
    public CursorWindow createFromParcel(Parcel source) {
        return new CursorWindow(source);
    }

    public CursorWindow[] newArray(int size) {
        return new CursorWindow[size];
    }
};
```

在上述代码中，使用参数 source 创建了一个 CursorWindow。创建 CursorWindow 对象功能是通过函数 CursorWindow 实现的，此函数在文件 frameworks/base/core/java/android/content/CursorWindow.java 中定义，具体实现代码如下所示：

```
private CursorWindow(Parcel source) {
    mStartPos = source.readInt();
    mWindowPtr = nativeCreateFromParcel(source);
    if (mWindowPtr == 0) {
        throw new CursorWindowAllocationException("Cursor window could not be "
                + "created from binder.");
    }
    mName = nativeGetName(mWindowPtr);
    mCloseGuard.open("close");
}
```

在上述创建 CursorWindow 对象的过程中，首先是从数据流 source 中将写入的 Binder 接口读取出来，然后使用这个 Binder 接口初始化这个 CursorWindow 对象。其实，这个 Binder 接口的实际类型为 IMemory，在里面封装了对匿名共享内存的访问操作。初始化这个匿名共享内存对象的操作是由本地函数 nativeGetName 实现的，此函数在文件 frameworks/base/core/jni/android_database_CursorWindow.cpp 中定义，具体实现代码如下所示：

```
static jstring nativeGetName(JNIEnv* env, jclass clazz, jint windowPtr) {
```

```
    CursorWindow* window = reinterpret_cast<CursorWindow*>(windowPtr);
    return env->NewStringUTF(window->name().string());
}
```

这样便分别在 Java 层和 C++层将 CursorWindow 对象实现了关联。

12.3.5 数据操作

再看文件 frameworks/base/core/java/android/database/sqlite/SQLiteQueryBuilder.java 中的函数 query，功能是准备一个数据库查询计划，此函数的具体实现代码如下所示：

```
public Cursor query(SQLiteDatabase db, String[] projectionIn,
        String selection, String[] selectionArgs, String groupBy,
        String having, String sortOrder, String limit, CancellationSignal
        cancellationSignal) {
    if (mTables == null) {
        return null;
    }

    if (mStrict && selection != null && selection.length() > 0) {
        String sqlForValidation = buildQuery(projectionIn, "(" + selection + ")", groupBy,
                having, sortOrder, limit);
        validateQuerySql(db, sqlForValidation,
                cancellationSignal); // will throw if query is invalid
    }
    String sql = buildQuery(
            projectionIn, selection, groupBy, having,
            sortOrder, limit);
    if (Log.isLoggable(TAG, Log.DEBUG)) {
        Log.d(TAG, "Performing query: " + sql);
    }
    return db.rawQueryWithFactory(
            mFactory, sql, selectionArgs,
            SQLiteDatabase.findEditTable(mTables),
            cancellationSignal); // will throw if query is invalid
}
```

在上述代码中，调用函数 buildQuery 构造了一个 SQL 语句，能够根据从参数传来的“列名”子句、select 子句、where 子句、group by 子句、having 子句、order 子句以及 limit 子句构造一个完整的 SQL 语句。构造完这个 SQL 查询语句后，调用从参数传下来的数据库对象 db 的函数 rawQueryWith Factory 来实现进一步操作。函数 rawQueryWithFactory 在文件 frameworks/base/core/java/android/ database/sqlite/SQLiteDatabase.java 中定义，具体实现代码如下所示：

```
public Cursor rawQueryWithFactory(
        CursorFactory cursorFactory, String sql, String[] selectionArgs,
        String editTable, CancellationSignal cancellationSignal) {
    acquireReference();
    try {
        SQLiteCursorDriver driver = new SQLiteDirectCursorDriver(this, sql, editTable,
                cancellationSignal);
        return driver.query(cursorFactory != null ? cursorFactory : mCursorFactory,
                selectionArgs);
    } finally {
        releaseReference();
    }
}
```

在上述代码中创建了一个 SQLiteCursorDriver 对象 driver，然后调用它的成员函数 query 创建一个 Cursor 对象，此 Cursor 对象的类型是 SQLiteCursor：

```
public Cursor query(CursorFactory factory, String[] selectionArgs) {
    final SQLiteQuery query = new SQLiteQuery(mDatabase, mSql, mCancellationSignal);
    final Cursor cursor;
    try {
        query.bindAllArgsAsStrings(selectionArgs);

        if (factory == null) {
            cursor = new SQLiteCursor(this, mEditTable, query);
```

```
        } else {
            cursor = factory.newCursor(mDatabase, this, mEditTable, query);
        }
    } catch (RuntimeException ex) {
        query.close();
        throw ex;
    }

    mQuery = query;
    return cursor;
}
```

在上述代码中，会先根据数据库对象 mDatabase 和原生 SQL 语句构造一个 SQLiteQuery 对象。在创建这个对象的过程中会解析这个原生 SQL 语句，并创建数据库查询计划，这样可以等到真正查询时就可以立即从数据库中获取数据，无需分析和理解这个 SQL 字符串语句，上述整个过程被称为 SQL 语句编译。有了这个 SQLiteQuery 对象之后，再将其和数据库对象 mDatabase 等待信息一起来创建一个 SQLiteCursor 对象，这样，这个 SQLiteCursor 对象就可以圈定将来要从数据库中获取的数据了。当执行完这一步骤后，就可以把这个 SQLiteCursor 对象返回给上层，最终返回到类 Transport 中的函数 bulkQuery 中。有了这个 SQLiteCursor 对象后，就可以通过创建一个 CursorToBulkCursorAdaptor 对象的方式把它和匿名共享内存关联起来，这样就为将来从数据库中查询得到的数据找到了归宿。CursorToBulkCursorAdaptor 对象的实现文件是 frameworks/base/core/java/android/database/CursorToBulkCursorAdaptor.java，其成员函数 CursorToBulkCursorAdaptor 的功能是将 SQLiteCurosr 对象转换为一个 AbstractWindowedCursor 对象，目的是为了调用函数 setWindow 把传进来的 CursorWindow 对象 window 保存起来，以便后面用来保存数据。函数 CursorToBulkCursorAdaptor 的具体实现代码如下所示：

```
public CursorToBulkCursorAdaptor(Cursor cursor, IContentObserver observer,
        String providerName) {
    if (cursor instanceof CrossProcessCursor) {
        mCursor = (CrossProcessCursor)cursor;
    } else {
        mCursor = new CrossProcessCursorWrapper(cursor);
    }
    mProviderName = providerName;
    synchronized (mLock) {
        createAndRegisterObserverProxyLocked(observer);
    }
}
```

然后执行类 SQLiteCursor 中的成员函数 getCount，功能是获取 MainActivity 组件获取信息的请求。函数 getCount 在文件 frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java 中定义，具体实现代码如下所示：

```
public int getCount() {
    if (mCount == NO_COUNT) {
        fillWindow(0);
    }
    return mCount;
}
```

在上述代码中，变量 mCount 的初始化值为 NO_COUNT，表示还没有去执行数据库查询操作，所以不知道它的值是多少，这需要通过调用函数 fillWindow 从数据库中查询获得。函数 fillWindow 在文件 frameworks/base/core/java/android/database/sqlite/SQLiteCursor.java 中定义，具体实现代码如下所示：

```
private void fillWindow(int requiredPos) {
    clearOrCreateWindow(getDatabase().getPath());

    try {
        if (mCount == NO_COUNT) {
            int startPos = DatabaseUtils.cursorPickFillWindowStartPosition(requiredPos, 0);
            mCount = mQuery.fillWindow(mWindow, startPos, requiredPos, true);
            mCursorWindowCapacity = mWindow.getNumRows();
```

```
            if (Log.isLoggable(TAG, Log.DEBUG)) {
                Log.d(TAG, "received count(*) from native_fill_window: " + mCount);
            }
        } else {
            int startPos = DatabaseUtils.cursorPickFillWindowStartPosition(requiredPos,
                    mCursorWindowCapacity);
            mQuery.fillWindow(mWindow, startPos, requiredPos, false);
        }
    } catch (RuntimeException ex) {
        closeWindow();
        throw ex;
    }
}
```

到此为止，会最终把从 Content Provider 中查询得到的数据通过匿名共享内存返回给第三方应用程序。这样分析完成了 Android 应用程序组件 Content Provider 在应用程序之间共享数据的原理，由此可见，整个过程是通过 Binder 进程间通信机制和匿名共享内存来实现的。

第 13 章　分析广播机制源代码

在 Android 中，Broadcast 是一种广泛运用的、在应用程序之间传输信息的机制。其中 BroadcastReceiver 用于接收广播信息，是对发送出来的 Broadcast 进行过滤接收并响应的一类组件。而 BroadcastReceiver 是一个广播接收器，能够专注于接收广播通知信息，并做出对应处理的组件。在本章的内容中，将详细讲解 Android 5.0 中 Broadcast（广播）系统的源代码。

13.1 Broadcast 基础

在 Android 系统中，Broadcast 是一种广泛运用的、在应用程序之间传输信息的机制。其中 BroadcastReceiver 负责对发送出来的 Broadcast 进行过滤接收并响应，而 BroadcastReceiver 负责接收广播通知信息并做出对应的处理。

在 Android 系统中，发送 Broadcast 和使用 BroadcastReceiver 过滤接收的过程如下所示。

（1）首先在需要发送信息的地方，把要发送的信息和用于过滤的信息（如 Action、Category）装入一个 Intent 对象。

（2）通过调用 Context.sendBroadcast()、sendOrderBroadcast()或 sendStickyBroadcast()方法，把 Intent 对象以广播方式发送出去。

（3）当发送 Intent 以后，所有已经注册的 BroadcastReceiver 会检查注册时的 IntentFilter 是否与发送的 Intent 相匹配，如果匹配则就会调用 BroadcastReceiver 的 onReceive()方法。所以，当我们定义一个 BroadcastReceiver 时，都需要实现 onReceive()方法。

在现实开发应用中，有如下两种注册 BroadcastReceiver 的方式。

（1）静态方式：在文件 AndroidManifest.xml 中用<receiver>标签生命注册，并在标签内用<intent-filter>标签设置过滤器。

（2）动态方式：在代码中先定义并设置好一个 IntentFilter 对象，然后在需要注册的地方调用 Context.registerReceiver()方法，如果取消时就调用 Context.unregisterReceiver()方法。如果用动态方式注册的 BroadcastReceiver 的 Context 对象被销毁时，BroadcastReceiver 也就自动取消注册了（特别注意，有些可能需要后台监听的，如短信消息）。

如果在使用 sendBroadcast()方法时指定了接收权限，则只有在 AndroidManifest.xml 中用<uses-permission>标签声明了拥有此权限的 BroascastReceiver 才会有可能接收到发送来的 Broadcast。同样，若在注册 BroadcastReceiver 时指定了可接收的 Broadcast 的权限，则只有在包内的 Android Manifest.xml 中用<uses-permission>标签声明了，拥有此权限的 Context 对象所发送的 Broadcast 才能被这个 BroadcastReceiver 所接收。

在 Android 开发应用中，广播事件的基本流程如下所示。

（1）注册广播事件：注册方式有两种，一种是静态注册，就是在 AndroidManifest.xml 文件中定义，注册的广播接收器必须要继承 BroadcastReceiver；另一种是动态注册，是在程序中使用 Context.registerReceiver 注册，注册的广播接收器相当于一个匿名类。两种方式都需要 IntentFIlter。

（2）发送广播事件：通过 Context.sendBroadcast 来发送，由 Intent 来传递注册时用到的 Action。

（3）接收广播事件：当发送的广播被接收器监听到后，会调用它的 onReceive()方法，并将包含消息的 Intent 对象传给它。onReceive 中代码的执行时间不要超过 5s，否则 Android 会弹出超时对话框（Dialog）。

13.2 发送广播信息

在 Android 系统中，发送广播功能是通过 sendBroadcast 实现的，整个过程以 ActivityManager Service 为中心。广播的发送者将广播发送到 ActivityManagerService，当 ActivityManagerService 接收到这个广播后，会在自己的注册中心查看有哪些广播接收器订阅了这个广播，然后将此广播逐一发送到这些广播接收器中。上述广播过程中的发送和处理是异步实现的，ActivityManager Service 并不等待广播接收器处理完这些广播就会返回。由此可见，广播的发送路径就是从发送者到 ActivityManagerService，再从 ActivityManagerService 到接收者，这中间的两个过程都是通过 Binder 进程间通信机制来完成的。

在本节的内容中，将详细分析 Android 5.0 中发送广播信息的源代码。

13.2.1 intent 描述指示

在 Android 应用开发过程中，当在某个 Service 中想要发送广播时，通常会调用如下代码来实现：

```
Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
          intent.putExtra(COUNTER_VALUE, counter);
          sendBroadcast(intent);
```

Android 中的广播使用 Intent 来描述，BROADCAST_COUNTER_ACTION 名称就是用来和广播接收者的类型进行匹配的。在类 Intent 中的定义代码如下所示：

```
public class Intent implements Parcelable, Cloneable {
    // ---------------------------------------------------------------------
    private String mAction;
    private Uri mData;
    private String mType;
    private String mPackage;
    private ComponentName mComponent;
    private int mFlags;
    private HashSet<String> mCategories;
    private Bundle mExtras;
private Rect mSourceBounds;
}
```

在上述代码中，变量 mAction 和 mExtras 不为空，其余为空。

13.2.2 传递广播信息

在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义函数 sendBroadcast，功能是调用 ContextImpl.sendBroadcast 实现进一步的操作，具体实现代码如下所示：

```
public class ContextWrapper extends Context {
    Context mBase;

    public ContextWrapper(Context base) {
        mBase = base;
}

    @Override
    public void sendBroadcast(Intent intent) {
        mBase.sendBroadcast(intent);
    }
...
}
```

在上述代码中，变量 mBase 是一个 ContextImpl 实例。

再看文件 frameworks/base/core/java/android/app/ContextImpl.java 中的函数 sendBroadcast，功能是调用类 ActivityManagerService 中的远程接口 ActivityManagerProxy，将这个广播信息发送给 ActivityManagerService。函数 sendBroadcast 的具体实现代码如下所示：

```
public void sendBroadcast(Intent intent) {
```

```
        warnIfCallingFromSystemProcess();
        String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
        try {
            intent.prepareToLeaveProcess();
            ActivityManagerNative.getDefault().broadcastIntent(
                mMainThread.getApplicationThread(), intent, resolvedType, null,
                Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE, false, false,
                getUserId());
        } catch (RemoteException e) {
        }
    }
```

在上述代码中，resolvedType 表示这个 Intent 的 MIME 类型。

13.2.3　封装传递

再看文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中的函数 broadcastIntent，功能是封装传递的参数，并通过 Binder 驱动程序进入到类 ActivityManagerService 中的函数 broadcastIntent 中。函数 broadcastIntent 的具体实现代码如下所示：

```
public int broadcastIntent(IApplicationThread caller,
        Intent intent, String resolvedType,  IIntentReceiver resultTo,
        int resultCode, String resultData, Bundle map,
        String requiredPermission, int appOp, boolean serialized,
        boolean sticky, int userId) throws RemoteException
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    //将传进来的参数写入 data 对象中
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    intent.writeToParcel(data, 0);
    data.writeString(resolvedType);
    data.writeStrongBinder(resultTo != null ? resultTo.asBinder() : null);
    data.writeInt(resultCode);
    data.writeString(resultData);
    data.writeBundle(map);
    data.writeString(requiredPermission);
    data.writeInt(appOp);
    data.writeInt(serialized ? 1 : 0);
    data.writeInt(sticky ? 1 : 0);
    data.writeInt(userId);
    mRemote.transact(BROADCAST_INTENT_TRANSACTION, data, reply, 0);
    reply.readException();
    int res = reply.readInt();
    reply.recycle();
    data.recycle();
    return res;
}
```

13.2.4　处理发送请求

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 broadcastIntent，功能是处理类型为 BROADCAST_INTENT_TRANSACTION 的进程通信请求。函数 broadcastIntent 的具体实现代码如下所示：

```
public final int broadcastIntent(IApplicationThread caller,
        Intent intent, String resolvedType, IIntentReceiver resultTo,
        int resultCode, String resultData, Bundle map,
        String requiredPermission, int appOp, boolean serialized, boolean sticky, int
        userId) {
    enforceNotIsolatedCaller("broadcastIntent");
    synchronized(this) {
    //验证 intent 描述的广播内容是否合法
        intent = verifyBroadcastLocked(intent);
        //获取发送广播进程的身份
        final ProcessRecord callerApp = getRecordForAppLocked(caller);
        final int callingPid = Binder.getCallingPid();
        final int callingUid = Binder.getCallingUid();
```

```
        final long origId = Binder.clearCallingIdentity();
        //调用函数 broadcastIntentLocked 处理参数 intent 描述的广播
        int res = broadcastIntentLocked(callerApp,
                callerApp != null ? callerApp.info.packageName : null,
                intent, resolvedType, resultTo,
                resultCode, resultData, map, requiredPermission, appOp, serialized, sticky,
                callingPid, callingUid, userId);
        Binder.restoreCallingIdentity(origId);
        return res;
    }
}
```

13.2.5 查找广播接收者

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 broadcastIntentLocked，功能是查找目标广播的接收者。此函数会先根据 intent 找出相应的广播接收器，然后验证是否设置这个 intent 的 Intent.FLAG_RECEIVER_REPLACE_PENDING 位。如果没有，则 ActivityManagerService 会在当前的系统中查看有没有相同的 intent 还未被处理；如果有，则用当前新的 intent 来替换旧的 intent。函数 broadcastIntentLocked 的具体实现代码如下所示：

```
    private final int broadcastIntentLocked(ProcessRecord callerApp,
            String callerPackage, Intent intent, String resolvedType,
            IIntentReceiver resultTo, int resultCode, String resultData,
            Bundle map, String requiredPermission, int appOp,
            boolean ordered, boolean sticky, int callingPid, int callingUid,
            int userId) {
        intent = new Intent(intent);

        // By default broadcasts do not go to stopped apps.
        intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);

        if (DEBUG_BROADCAST_LIGHT) Slog.v(
            TAG, (sticky ? "Broadcast sticky: ": "Broadcast: ") + intent
            + " ordered=" + ordered + " userid=" + userId);
        if ((resultTo != null) && !ordered) {
            Slog.w(TAG, "Broadcast " + intent + " not ordered but result callback
            requested!");
        }
        ......
        // 根据 intent 找出相应的广播接收器
        List receivers = null;
        List<BroadcastFilter> registeredReceivers = null;
        // Need to resolve the intent to interested receivers...
        if ((intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY)
                == 0) {
            receivers = collectReceiverComponents(intent, resolvedType, users);
        }
        if (intent.getComponent() == null) {
            registeredReceivers = mReceiverResolver.queryIntent(intent,
                    resolvedType, false, userId);
        }
    //验证是否设置 intent 的 Intent.FLAG_RECEIVER_REPLACE_PENDING 位
        final boolean replacePending =
                (intent.getFlags()&Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0;

        if (DEBUG_BROADCAST) Slog.v(TAG, "Enqueing broadcast: " + intent.getAction()
                + " replacePending=" + replacePending);

        int NR = registeredReceivers != null ? registeredReceivers.size() : 0;
        if (!ordered && NR > 0) {
            // If we are not serializing this broadcast, then send the
            // registered receivers separately so they don't wait for the
            // components to be launched.
            final BroadcastQueue queue = broadcastQueueForIntent(intent);
            BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
                    callerPackage, callingPid, callingUid, requiredPermission, appOp,
                    registeredReceivers, resultTo, resultCode, resultData, map,
                    ordered, sticky, false, userId);
            if (DEBUG_BROADCAST) Slog.v(
```

```
                TAG, "Enqueueing parallel broadcast " + r);
        final boolean replaced = replacePending && queue.replaceParallel
        BroadcastLocked(r);
        if (!replaced) {
            queue.enqueueParallelBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();
        }
        registeredReceivers = null;
        NR = 0;
    }

    // Merge into one list.
    int ir = 0;
    if (receivers != null) {
        // A special case for PACKAGE_ADDED: do not allow the package
        // being added to see this broadcast.  This prevents them from
        // using this as a back door to get run as soon as they are
        // installed.  Maybe in the future we want to have a special install
        // broadcast or such for apps, but we'd like to deliberately make
        // this decision.
        String skipPackages[] = null;
        if (Intent.ACTION_PACKAGE_ADDED.equals(intent.getAction())
                || Intent.ACTION_PACKAGE_RESTARTED.equals(intent.getAction())
                || Intent.ACTION_PACKAGE_DATA_CLEARED.equals(intent.getAction())) {
            Uri data = intent.getData();
            if (data != null) {
                String pkgName = data.getSchemeSpecificPart();
                if (pkgName != null) {
                    skipPackages = new String[] { pkgName };
                }
            }
        } else if (Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE.equals(intent.
        getAction())) {
            skipPackages = intent.getStringArrayExtra(Intent.EXTRA_CHANGED_PACKAGE_
            LIST);
        }
        if (skipPackages != null && (skipPackages.length > 0)) {
        //循环验证是否存在和参数 intent 一样的广播
            for (String skipPackage : skipPackages) {
                if (skipPackage != null) {
                    int NT = receivers.size();
                    for (int it=0; it<NT; it++) {
                        ResolveInfo curt = (ResolveInfo)receivers.get(it);
                        if (curt.activityInfo.packageName.equals(skipPackage)) {
                            receivers.remove(it);
                            it--;
                            NT--;
                        }
                    }
                }
            }
        }

        int NT = receivers != null ? receivers.size() : 0;
        int it = 0;
        ResolveInfo curt = null;
        BroadcastFilter curr = null;
        while (it < NT && ir < NR) {
            if (curt == null) {
                curt = (ResolveInfo)receivers.get(it);
            }
            if (curr == null) {
                curr = registeredReceivers.get(ir);
            }
            if (curr.getPriority() >= curt.priority) {
                // Insert this broadcast record into the final list.
                receivers.add(it, curr);
                ir++;
                curr = null;
                it++;
                NT++;
```

```
                } else {
                    // Skip to the next ResolveInfo in the final list.
                    it++;
                    curt = null;
                }
            }
        }
        while (ir < NR) {
            if (receivers == null) {
                receivers = new ArrayList();
            }
            receivers.add(registeredReceivers.get(ir));
            ir++;
        }

        if ((receivers != null && receivers.size() > 0)
                || resultTo != null) {
            BroadcastQueue queue = broadcastQueueForIntent(intent);
            BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
                    callerPackage, callingPid, callingUid, requiredPermission, appOp,
                    receivers, resultTo, resultCode, resultData, map, ordered,
                    sticky, false, userId);
            if (DEBUG_BROADCAST) Slog.v(
                    TAG, "Enqueueing ordered broadcast " + r
                    + ": prev had " + queue.mOrderedBroadcasts.size());
            if (DEBUG_BROADCAST) {
                int seq = r.intent.getIntExtra("seq", -1);
                Slog.i(TAG, "Enqueueing broadcast " + r.intent.getAction() + " seq=" + seq);
            }
            boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);
            if (!replaced) {
                queue.enqueueOrderedBroadcastLocked(r);
                queue.scheduleBroadcastsLocked();
            }
        }

        return ActivityManager.BROADCAST_SUCCESS;
    }
```

在上述代码中，成员变量 mHandler 在类 ActivityManagerService 的内部被定义，是一个 Handler 类变量，通过此类中的函数 sendEmptyMessage，可以将一个类型为 BROADCAST_INTENT_MSG 的空消息放进 ActivityManagerService 的消息队列中。此处的空消息是指这个消息除了有类型信息之外，没有任何其他额外的信息。

13.2.6 处理广播信息

再看函数 scheduleBroadcastsLocked，具体实现代码如下所示：

```
private final void scheduleBroadcastsLocked() {
        if (DEBUG_BROADCAST) Slog.v(TAG, "Schedule broadcasts: current="
                + mBroadcastsScheduled);
        if (mBroadcastsScheduled) {
            return;
        }
        mHandler.sendEmptyMessage(BROADCAST_INTENT_MSG);
        mBroadcastsScheduled = true;
}
```

在上述代码中，mBroadcastsScheduled 表示是否已经向所有运行的线程发送了一个类型为 BROADCAST_INTENT_MSG 的消息。

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 handleMessage，功能是处理类型为 BROADCAST_INTENT_MSG 的广播信息。主要实现代码如下所示：

```
    public void handleMessage(Message msg) {
        switch (msg.what) {
        case SHOW_ERROR_MSG: {
            HashMap data = (HashMap) msg.obj;
```

```
            boolean showBackground = Settings.Secure.getInt(mContext.getContentResolver(),
                    Settings.Secure.ANR_SHOW_BACKGROUND, 0) != 0;
            synchronized (ActivityManagerService.this) {
                ProcessRecord proc = (ProcessRecord)data.get("app");
                AppErrorResult res = (AppErrorResult) data.get("result");
                if (proc != null && proc.crashDialog != null) {
                    Slog.e(TAG, "App already has crash dialog: " + proc);
                    if (res != null) {
                        res.set(0);
                    }
                    return;
                }
                if (!showBackground && UserHandle.getAppId(proc.uid)
                        >= Process.FIRST_APPLICATION_UID && proc.userId != mCurrentUserId
                        && proc.pid != MY_PID) {
                    Slog.w(TAG, "Skipping crash dialog of " + proc + ": background");
                    if (res != null) {
                        res.set(0);
                    }
                    return;
                }
                if (mShowDialogs && !mSleeping && !mShuttingDown) {
                    Dialog d = new AppErrorDialog(mContext,
                            ActivityManagerService.this, res, proc);
                    d.show();
                    proc.crashDialog = d;
                } else {
                    // The device is asleep, so just pretend that the user
                    // saw a crash dialog and hit "force quit".
                    if (res != null) {
                        res.set(0);
                    }
                }
            }

            ensureBootCompleted();
        } break;
        case BROADCAST_INTENT_MSG: {
            ……
        //调用函数 processNextBroadcast 来处理下一个未处理的广播
            processNextBroadcast(true);
        } break;
        ......
        }
    }
}
```

在上述代码中，调用类 ActivityManagerService 中的函数 processNextBroadcast 来处理下一个未处理的广播。函数 processNextBroadcast 在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，具体实现代码如下所示：

```
final void processNextBroadcast(boolean fromMsg) {
    synchronized(mService) {
        BroadcastRecord r;

        if (DEBUG_BROADCAST) Slog.v(TAG, "processNextBroadcast ["
                + mQueueName + "]: "
                + mParallelBroadcasts.size() + " broadcasts, "
                + mOrderedBroadcasts.size() + " ordered broadcasts");

        mService.updateCpuStats();

        if (fromMsg) {
            mBroadcastsScheduled = false;
        }

        // First, deliver any non-serialized broadcasts right away.
        while (mParallelBroadcasts.size() > 0) {
            r = mParallelBroadcasts.remove(0);
            r.dispatchTime = SystemClock.uptimeMillis();
```

```
            r.dispatchClockTime = System.currentTimeMillis();
            final int N = r.receivers.size();
            if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Processing parallel broadcast ["
                    + mQueueName + "] " + r);
            for (int i=0; i<N; i++) {
                Object target = r.receivers.get(i);
                if (DEBUG_BROADCAST)  Slog.v(TAG,
                        "Delivering non-ordered on [" + mQueueName + "] to registered "
                        + target + ": " + r);
                deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, false);
            }
            addBroadcastToHistoryLocked(r);
            if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Done with parallel broadcast ["
                    + mQueueName + "] " + r);
        }

        // Now take care of the next serialized one...

        // If we are waiting for a process to come up to handle the next
        // broadcast, then do nothing at this point.  Just in case, we
        // check that the process we're waiting for still exists.
        if (mPendingBroadcast != null) {
            if (DEBUG_BROADCAST_LIGHT) {
                Slog.v(TAG, "processNextBroadcast ["
                        + mQueueName + "]: waiting for "
                        + mPendingBroadcast.curApp);
            }

            boolean isDead;
            synchronized (mService.mPidsSelfLocked) {
                isDead = (mService.mPidsSelfLocked.get(
                        mPendingBroadcast.curApp.pid) == null);
            }
            if (!isDead) {
                // It's still alive, so keep waiting
                return;
            } else {
                Slog.w(TAG, "pending app  ["
                        + mQueueName + "]" + mPendingBroadcast.curApp
                        + " died before responding to broadcast");
                mPendingBroadcast.state = BroadcastRecord.IDLE;
                mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
                mPendingBroadcast = null;
            }
        }

        boolean looped = false;

        do {
            if (mOrderedBroadcasts.size() == 0) {
                // No more broadcasts pending, so all done!
                mService.scheduleAppGcsLocked();
                if (looped) {
                    // If we had finished the last ordered broadcast, then
                    // make sure all processes have correct oom and sched
                    // adjustments.
                    mService.updateOomAdjLocked();
                }
                return;
            }
            r = mOrderedBroadcasts.get(0);
            boolean forceReceive = false;

            // Ensure that even if something goes awry with the timeout
            // detection, we catch "hung" broadcasts here, discard them,
            // and continue to make progress.
            //
            // This is only done if the system is ready so that PRE_BOOT_COMPLETED
            // receivers don't get executed with timeouts. They're intended for
            // one time heavy lifting after system upgrades and can take
            // significant amounts of time.
```

```
            int numReceivers = (r.receivers != null) ? r.receivers.size() : 0;
            if (mService.mProcessesReady && r.dispatchTime > 0) {
                long now = SystemClock.uptimeMillis();
                if ((numReceivers > 0) &&
                        (now > r.dispatchTime + (2*mTimeoutPeriod*numReceivers))) {
                    Slog.w(TAG, "Hung broadcast ["
                            + mQueueName + "] discarded after timeout failure:"
                            + " now=" + now
                            + " dispatchTime=" + r.dispatchTime
                            + " startTime=" + r.receiverTime
                            + " intent=" + r.intent
                            + " numReceivers=" + numReceivers
                            + " nextReceiver=" + r.nextReceiver
                            + " state=" + r.state);
                    broadcastTimeoutLocked(false); // forcibly finish this broadcast
                    forceReceive = true;
                    r.state = BroadcastRecord.IDLE;
                }
            }

            if (r.state != BroadcastRecord.IDLE) {
                if (DEBUG_BROADCAST) Slog.d(TAG,
                        "processNextBroadcast("
                        + mQueueName + ") called when not idle (state="
                        + r.state + ")");
                return;
            }

            if (r.receivers == null || r.nextReceiver >= numReceivers
                    || r.resultAbort || forceReceive) {
                // No more receivers for this broadcast!  Send the final
                // result if requested...
                if (r.resultTo != null) {
                    try {
                        if (DEBUG_BROADCAST) {
                            int seq = r.intent.getIntExtra("seq", -1);
                            Slog.i(TAG, "Finishing broadcast ["
                                    + mQueueName + "] " + r.intent.getAction()
                                    + " seq=" + seq + " app=" + r.callerApp);
                        }
                        performReceiveLocked(r.callerApp, r.resultTo,
                            new Intent(r.intent), r.resultCode,
                            r.resultData, r.resultExtras, false, false, r.userId);
                        // Set this to null so that the reference
                        // (local and remote) isnt kept in the mBroadcastHistory.
                        r.resultTo = null;
                    } catch (RemoteException e) {
                        Slog.w(TAG, "Failure ["
                                + mQueueName + "] sending broadcast result of "
                                + r.intent, e);
                    }
                }

                if (DEBUG_BROADCAST) Slog.v(TAG, "Cancelling BROADCAST_TIMEOUT_MSG");
                cancelBroadcastTimeoutLocked();

                if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Finished with ordered broadcast "
                        + r);

                // ... and on to the next...
                addBroadcastToHistoryLocked(r);
                mOrderedBroadcasts.remove(0);
                r = null;
                looped = true;
                continue;
            }
        } while (r == null);

        // Get the next receiver...
        int recIdx = r.nextReceiver++;
```

```
// Keep track of when this receiver started, and make sure there
// is a timeout message pending to kill it if need be.
r.receiverTime = SystemClock.uptimeMillis();
if (recIdx == 0) {
    r.dispatchTime = r.receiverTime;
    r.dispatchClockTime = System.currentTimeMillis();
    if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Processing ordered broadcast ["
            + mQueueName + "] " + r);
}
if (! mPendingBroadcastTimeoutMessage) {
    long timeoutTime = r.receiverTime + mTimeoutPeriod;
    if (DEBUG_BROADCAST) Slog.v(TAG,
            "Submitting BROADCAST_TIMEOUT_MSG ["
            + mQueueName + "] for " + r + " at " + timeoutTime);
    setBroadcastTimeoutLocked(timeoutTime);
}

Object nextReceiver = r.receivers.get(recIdx);
if (nextReceiver instanceof BroadcastFilter) {
    // Simple case: this is a registered receiver who gets
    // a direct call.
    BroadcastFilter filter = (BroadcastFilter)nextReceiver;
    if (DEBUG_BROADCAST)  Slog.v(TAG,
            "Delivering ordered ["
            + mQueueName + "] to registered "
            + filter + ": " + r);
    deliverToRegisteredReceiverLocked(r, filter, r.ordered);
    if (r.receiver == null || !r.ordered) {
        // The receiver has already finished, so schedule to
        // process the next one.
        if (DEBUG_BROADCAST) Slog.v(TAG, "Quick finishing ["
                + mQueueName + "]: ordered="
                + r.ordered + " receiver=" + r.receiver);
        r.state = BroadcastRecord.IDLE;
        scheduleBroadcastsLocked();
    }
    return;
}

// Hard case: need to instantiate the receiver, possibly
// starting its application process to host it.

ResolveInfo info =
    (ResolveInfo)nextReceiver;
ComponentName component = new ComponentName(
        info.activityInfo.applicationInfo.packageName,
        info.activityInfo.name);

boolean skip = false;
int perm = mService.checkComponentPermission(info.activityInfo.permission,
        r.callingPid, r.callingUid, info.activityInfo.applicationInfo.uid,
        info.activityInfo.exported);
if (perm != PackageManager.PERMISSION_GRANTED) {
    if (!info.activityInfo.exported) {
        Slog.w(TAG, "Permission Denial: broadcasting "
                + r.intent.toString()
                + " from " + r.callerPackage + " (pid=" + r.callingPid
                + ", uid=" + r.callingUid + ")"
                + " is not exported from uid " + info.activityInfo.applicationInfo.uid
                + " due to receiver " + component.flattenToShortString());
    } else {
        Slog.w(TAG, "Permission Denial: broadcasting "
                + r.intent.toString()
                + " from " + r.callerPackage + " (pid=" + r.callingPid
                + ", uid=" + r.callingUid + ")"
                + " requires " + info.activityInfo.permission
                + " due to receiver " + component.flattenToShortString());
    }
    skip = true;
}
if (info.activityInfo.applicationInfo.uid != Process.SYSTEM_UID &&
```

```
            r.requiredPermission != null) {
            try {
                perm = AppGlobals.getPackageManager().
                        checkPermission(r.requiredPermission,
                                info.activityInfo.applicationInfo.packageName);
            } catch (RemoteException e) {
                perm = PackageManager.PERMISSION_DENIED;
            }
            if (perm != PackageManager.PERMISSION_GRANTED) {
                Slog.w(TAG, "Permission Denial: receiving "
                        + r.intent + " to "
                        + component.flattenToShortString()
                        + " requires " + r.requiredPermission
                        + " due to sender " + r.callerPackage
                        + " (uid " + r.callingUid + ")");
                skip = true;
            }
        }
        if (r.appOp != AppOpsManager.OP_NONE) {
            int mode = mService.mAppOpsService.checkOperation(r.appOp,
                    info.activityInfo.applicationInfo.uid, info.activityInfo.packageName);
            if (mode != AppOpsManager.MODE_ALLOWED) {
                if (DEBUG_BROADCAST)  Slog.v(TAG,
                        "App op " + r.appOp + " not allowed for broadcast to uid "
                        + info.activityInfo.applicationInfo.uid + " pkg "
                        + info.activityInfo.packageName);
                skip = true;
            }
        }
        boolean isSingleton = false;
        try {
            isSingleton = mService.isSingleton(info.activityInfo.processName,
                    info.activityInfo.applicationInfo,
                    info.activityInfo.name, info.activityInfo.flags);
        } catch (SecurityException e) {
            Slog.w(TAG, e.getMessage());
            skip = true;
        }
        if ((info.activityInfo.flags&ActivityInfo.FLAG_SINGLE_USER) != 0) {
            if (ActivityManager.checkUidPermission(
                    android.Manifest.permission.INTERACT_ACROSS_USERS,
                    info.activityInfo.applicationInfo.uid)
                            != PackageManager.PERMISSION_GRANTED) {
                Slog.w(TAG, "Permission Denial: Receiver " + component.flattenToShortString()
                        + " requests FLAG_SINGLE_USER, but app does not hold "
                        + android.Manifest.permission.INTERACT_ACROSS_USERS);
                skip = true;
            }
        }
        if (r.curApp != null && r.curApp.crashing) {
            // If the target process is crashing, just skip it.
            if (DEBUG_BROADCAST)  Slog.v(TAG,
                    "Skipping deliver ordered ["
                    + mQueueName + "] " + r + " to " + r.curApp
                    + ": process crashing");
            skip = true;
        }

        if (skip) {
            if (DEBUG_BROADCAST)  Slog.v(TAG,
                    "Skipping delivery of ordered ["
                    + mQueueName + "] " + r + " for whatever reason");
            r.receiver = null;
            r.curFilter = null;
            r.state = BroadcastRecord.IDLE;
            scheduleBroadcastsLocked();
            return;
        }

        r.state = BroadcastRecord.APP_RECEIVE;
        String targetProcess = info.activityInfo.processName;
```

```
r.curComponent = component;
if (r.callingUid != Process.SYSTEM_UID && isSingleton) {
    info.activityInfo = mService.getActivityInfoForUser(info.activityInfo, 0);
}
r.curReceiver = info.activityInfo;
if (DEBUG_MU && r.callingUid > UserHandle.PER_USER_RANGE) {
    Slog.v(TAG_MU, "Updated broadcast record activity info for secondary user, "
            + info.activityInfo + ", callingUid = " + r.callingUid + ", uid = "
            + info.activityInfo.applicationInfo.uid);
}

// Broadcast is being executed, its package can't be stopped.
try {
    AppGlobals.getPackageManager().setPackageStoppedState(
            r.curComponent.getPackageName(), false, UserHandle.getUserId
            (r.callingUid));
} catch (RemoteException e) {
} catch (IllegalArgumentException e) {
    Slog.w(TAG, "Failed trying to unstop package "
            + r.curComponent.getPackageName() + ": " + e);
}

// Is this receiver's application already running?
ProcessRecord app = mService.getProcessRecordLocked(targetProcess,
        info.activityInfo.applicationInfo.uid);
if (app != null && app.thread != null) {
    try {
        app.addPackage(info.activityInfo.packageName);
        processCurBroadcastLocked(r, app);
        return;
    } catch (RemoteException e) {
        Slog.w(TAG, "Exception when sending broadcast to "
              + r.curComponent, e);
    } catch (RuntimeException e) {
        Log.wtf(TAG, "Failed sending broadcast to "
                + r.curComponent + " with " + r.intent, e);
        // If some unexpected exception happened, just skip
        // this broadcast.  At this point we are not in the call
        // from a client, so throwing an exception out from here
        // will crash the entire system instead of just whoever
        // sent the broadcast.
        logBroadcastReceiverDiscardLocked(r);
        finishReceiverLocked(r, r.resultCode, r.resultData,
                r.resultExtras, r.resultAbort, true);
        scheduleBroadcastsLocked();
        // We need to reset the state if we failed to start the receiver.
        r.state = BroadcastRecord.IDLE;
        return;
    }

    // If a dead object exception was thrown -- fall through to
    // restart the application.
}

// Not running -- get it started, to be executed when the app comes up.
if (DEBUG_BROADCAST)  Slog.v(TAG,
        "Need to start app ["
        + mQueueName + "] " + targetProcess + " for broadcast " + r);
if ((r.curApp=mService.startProcessLocked(targetProcess,
        info.activityInfo.applicationInfo, true,
        r.intent.getFlags() | Intent.FLAG_FROM_BACKGROUND,
        "broadcast", r.curComponent,
        (r.intent.getFlags()&Intent.FLAG_RECEIVER_BOOT_UPGRADE) != 0, false))
                == null) {
    // Ah, this recipient is unavailable.  Finish it if necessary,
    // and mark the broadcast record as ready for the next.
    Slog.w(TAG, "Unable to launch app "
            + info.activityInfo.applicationInfo.packageName + "/"
            + info.activityInfo.applicationInfo.uid + " for broadcast "
            + r.intent + ": process is bad");
    logBroadcastReceiverDiscardLocked(r);
```

```
                finishReceiverLocked(r, r.resultCode, r.resultData,
                        r.resultExtras, r.resultAbort, true);
                scheduleBroadcastsLocked();
                r.state = BroadcastRecord.IDLE;
                return;
            }

            mPendingBroadcast = r;
            mPendingBroadcastRecvIndex = recIdx;
        }
    }
```

函数 processNextBroadcast 的具体实现流程如下所示。

（1）判断 fromMsg，如果是通过消息发送过来的就为真，否则为假；如果为真，mBroadcasts Scheduled = false。这样的话，在函数 scheduleBroadcastsLocked 里面就可以再次发送 BROADCAST_INTENT_MSG 的消息，从而触发 processNextBroadcast 函数被再次调用。

（2）判断 mParallelBroadcasts 是否为空，不为空就开始调用这个列表里面的 receivers 来接收消息，这个过程后面在串行 Intent 的时候也会碰到，我们留到后面讨论，这里只需要知道它通过一个 while 循环把 Intent 发送给关注这个 Intent 的所有的 receivers。

（3）判断 mPendingBroadcast 是否为空，如果不为空，就表示先前发送的串行的 Intent 还没有处理完毕，一般出现这种可能是因为我们要发送到的 receiver 还没有启动，所以，需要先启动这个 Activity，然后等待启动的这个 Activity 处理，这时候，这个 mPendingBroadcast 就为 true；如果发送这种情况需要判断这个 Activity 是否死了，如果死了，那么就把 mPendingBroadcast 设为 false，否则就直接返回，继续等待。

（4）顺序从 mOrderedBroadcasts 中取出 BroadCastRecord 消息，然后对这个消息的 receiver 一个一个地调用其接收流程。在处理这个消息的过程中，先判断其接收者是不是 BroadFilter，如果是，则调用 deliverToRegisteredReceiver 来接收。

（5）如果不是，BroadCast Filter 则需要找出这个 receiver 所在的进程，这时通常是一个 IntentFilter 所在的进程。如果这个进程活着则调用 processCurBroadcastLocked(r, app)来处理，否则需要用 startProcessLocked 先启动这个进程，然后设置 mPendingBroadcast = r，这样等应用起来它会处理这个消息。

13.2.7　检查权限

在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中，通过函数 deliverToRegisteredReceiverLocked 来检查广播发送和接收的权限，然后调用函数 performReceive Locked 来进一步执行广播发送的操作。函数 deliverToRegisteredReceiverLocked 的具体实现代码如下所示：

```
private final void deliverToRegisteredReceiverLocked(BroadcastRecord r,
        BroadcastFilter filter, boolean ordered) {
    boolean skip = false;
    if (filter.requiredPermission != null) {
        int perm = mService.checkComponentPermission(filter.requiredPermission,
                r.callingPid, r.callingUid, -1, true);
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: broadcasting "
                    + r.intent.toString()
                    + " from " + r.callerPackage + " (pid="
                    + r.callingPid + ", uid=" + r.callingUid + ")"
                    + " requires " + filter.requiredPermission
                    + " due to registered receiver " + filter);
            skip = true;
        }
    }
    if (!skip && r.requiredPermission != null) {
        int perm = mService.checkComponentPermission(r.requiredPermission,
                filter.receiverList.pid, filter.receiverList.uid, -1, true);
        if (perm != PackageManager.PERMISSION_GRANTED) {
```

```
            Slog.w(TAG, "Permission Denial: receiving "
                    + r.intent.toString()
                    + " to " + filter.receiverList.app
                    + " (pid=" + filter.receiverList.pid
                    + ", uid=" + filter.receiverList.uid + ")"
                    + " requires " + r.requiredPermission
                    + " due to sender " + r.callerPackage
                    + " (uid " + r.callingUid + ")");
            skip = true;
        }
    }
    if (r.appOp != AppOpsManager.OP_NONE) {
        int mode = mService.mAppOpsService.checkOperation(r.appOp,
                filter.receiverList.uid, filter.packageName);
        if (mode != AppOpsManager.MODE_ALLOWED) {
            if (DEBUG_BROADCAST)  Slog.v(TAG,
                    "App op " + r.appOp + " not allowed for broadcast to uid "
                    + filter.receiverList.uid + " pkg " + filter.packageName);
            skip = true;
        }
    }

    if (!skip) {
        // If this is not being sent as an ordered broadcast, then we
        // don't want to touch the fields that keep track of the current
        // state of ordered broadcasts.
        if (ordered) {
            r.receiver = filter.receiverList.receiver.asBinder();
            r.curFilter = filter;
            filter.receiverList.curBroadcast = r;
            r.state = BroadcastRecord.CALL_IN_RECEIVE;
            if (filter.receiverList.app != null) {
                // Bump hosting application to no longer be in background
                // scheduling class.  Note that we can't do that if there
                // isn't an app...  but we can only be in that case for
                // things that directly call the IActivityManager API, which
                // are already core system stuff so don't matter for this.
                r.curApp = filter.receiverList.app;
                filter.receiverList.app.curReceiver = r;
                mService.updateOomAdjLocked();
            }
        }
        try {
            if (DEBUG_BROADCAST_LIGHT) {
                int seq = r.intent.getIntExtra("seq", -1);
                Slog.i(TAG, "Delivering to " + filter
                        + " (seq=" + seq + "): " + r);
            }
            performReceiveLocked(filter.receiverList.app, filter.receiverList.receiver,
                new Intent(r.intent), r.resultCode, r.resultData,
                r.resultExtras, r.ordered, r.initialSticky, r.userId);
            if (ordered) {
                r.state = BroadcastRecord.CALL_DONE_RECEIVE;
            }
        } catch (RemoteException e) {
            Slog.w(TAG, "Failure sending broadcast " + r.intent, e);
            if (ordered) {
                r.receiver = null;
                r.curFilter = null;
                filter.receiverList.curBroadcast = null;
                if (filter.receiverList.app != null) {
                    filter.receiverList.app.curReceiver = null;
                }
            }
        }
    }
}
```

再看函数 performReceiveLocked，此函数在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，具体实现代码如下所示：

```
private static void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
        Intent intent, int resultCode, String data, Bundle extras,
        boolean ordered, boolean sticky, int sendingUser) throws RemoteException {
    // Send the intent to the receiver asynchronously using one-way binder calls.
    if (app != null && app.thread != null) {
        // If we have an app thread, do the call through that so it is
        // correctly ordered with other one-way calls.
        app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
                data, extras, ordered, sticky, sendingUser);
    } else {
        receiver.performReceive(intent, resultCode, data, extras, ordered,
                sticky, sendingUser);
    }
}
```

各个参数的具体说明如下所示。

- app：表示注册广播接收器的 Activity 所在的进程记录块。
- receiver：指向一个实现了 IIntentReceiver 接口的 Binder 代理对象，表示目标广播接收者。
- intent：表示即将要发送给目标广播接收者的一个广播。

13.2.8　处理的进程通信请求

首先看函数 scheduleRegisteredReceiver，功能是通过 Binder 驱动程序来到类 ApplicationThread 中的函数 scheduleRegisteredReceiver 中。函数 scheduleRegisteredReceiver 在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，功能是把这个广播分发给 MainActivity，具体实现代码如下所示：

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
        int resultCode, String dataStr, Bundle extras, boolean ordered,
        boolean sticky, int sendingUser) throws RemoteException {
    Parcel data = Parcel.obtain();
    data.writeInterfaceToken(IApplicationThread.descriptor);
    data.writeStrongBinder(receiver.asBinder());
    intent.writeToParcel(data, 0);
    data.writeInt(resultCode);
    data.writeString(dataStr);
    data.writeBundle(extras);
    data.writeInt(ordered ? 1 : 0);
    data.writeInt(sticky ? 1 : 0);
    data.writeInt(sendingUser);
    mRemote.transact(SCHEDULE_REGISTERED_RECEIVER_TRANSACTION, data, null,
            IBinder.FLAG_ONEWAY);
    data.recycle();
}
```

再看文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中的函数 scheduleRegisteredReceiver，功能是通过 Binder 驱动程序进入到类 ApplicationThread 中的函数 scheduleRegisteredReceiver 中去，ApplicationThread 是 ActivityThread 的一个内部类。函数 scheduleRegisteredReceiver 的具体实现代码如下所示：

```
public void scheduleRegisteredReceiver(IIntentReceiver receiver, Intent intent,
        int resultCode, String dataStr, Bundle extras, boolean ordered,
        boolean sticky, int sendingUser) throws RemoteException {
    receiver.performReceive(intent, resultCode, dataStr, extras, ordered,
            sticky, sendingUser);
}
```

再看函数 performReceive，功能是接收来自参数 intent 描述的广播。函数 performReceive 在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void performReceive(Intent intent, int resultCode, String data,
                Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
            LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
            if (ActivityThread.DEBUG_BROADCAST) {
                int seq = intent.getIntExtra("seq", -1);
                Slog.i(ActivityThread.TAG, "Receiving broadcast " + intent.getAction()
```

```
                + " seq=" + seq + " to " + (rd != null ? rd.mReceiver : null));
            }
            if (rd != null) {
                rd.performReceive(intent, resultCode, data, extras,
                        ordered, sticky, sendingUser);
            } else {
                // The activity manager dispatched a broadcast to a registered
                // receiver in this process, but before it could be delivered the
                // receiver was unregistered.  Acknowledge the broadcast on its
                // behalf so that the system's broadcast sequence can continue.
                if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                        "Finishing broadcast to unregistered receiver");
                IActivityManager mgr = ActivityManagerNative.getDefault();
                try {
                    if (extras != null) {
                        extras.setAllowFds(false);
                    }
                    mgr.finishReceiver(this, resultCode, data, extras, false);
                } catch (RemoteException e) {
                    Slog.w(ActivityThread.TAG, "Couldn't finish broadcast to
                    unregistered receiver");
                }
            }
        }
    }
```

在上述代码中，调用类 ReceiverDispatcher 中的函数 performReceive 实现了进一步处理。下面看类 ReceiverDispatcher 中的函数 performReceive，此函数也是在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void performReceive(Intent intent, int resultCode, String data,
            Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
        if (ActivityThread.DEBUG_BROADCAST) {
            int seq = intent.getIntExtra("seq", -1);
            Slog.i(ActivityThread.TAG, "Enqueueing broadcast " + intent.getAction() +
            " seq=" + seq + " to " + mReceiver);
        }
        Args args = new Args(intent, resultCode, data, extras, ordered,
                sticky, sendingUser);
        if (!mActivityThread.post(args)) {
            if (mRegistered && ordered) {
                IActivityManager mgr = ActivityManagerNative.getDefault();
                if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                        "Finishing sync broadcast to " + mReceiver);
                args.sendFinished(mgr);
            }
        }
    }
}
```

这样 ReceiverDispatcher 的内部类 Args，会在 MainActivity 所在的线程消息循环中处理这个广播，并最终将这个广播分发给所注册的 BroadcastReceiver 实例的函数 onReceive 进行处理。

接下来看类 Args 中的函数 run，此函数在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public void run() {
        final BroadcastReceiver receiver = mReceiver;
        final boolean ordered = mOrdered;

        if (ActivityThread.DEBUG_BROADCAST) {
            int seq = mCurIntent.getIntExtra("seq", -1);
            Slog.i(ActivityThread.TAG, "Dispatching broadcast " + mCurIntent.
            getAction()
                    + " seq=" + seq + " to " + mReceiver);
            Slog.i(ActivityThread.TAG, "  mRegistered=" + mRegistered
                    + " mOrderedHint=" + ordered);
        }
        final IActivityManager mgr = ActivityManagerNative.getDefault();
        final Intent intent = mCurIntent;
```

```
            mCurIntent = null;

            if (receiver == null || mForgotten) {
                if (mRegistered && ordered) {
                    if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                            "Finishing null broadcast to " + mReceiver);
                    sendFinished(mgr);
                }
                return;
            }
            Trace.traceBegin(Trace.TRACE_TAG_ACTIVITY_MANAGER, "broadcastReceiveReg");
            try {
                ClassLoader cl =  mReceiver.getClass().getClassLoader();
                intent.setExtrasClassLoader(cl);
                setExtrasClassLoader(cl);
                receiver.setPendingResult(this);
                receiver.onReceive(mContext, intent);
            } catch (Exception e) {
                if (mRegistered && ordered) {
                    if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                            "Finishing failed broadcast to " + mReceiver);
                    sendFinished(mgr);
                }
                if (mInstrumentation == null ||
                        !mInstrumentation.onException(mReceiver, e)) {
                    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
                    throw new RuntimeException(
                        "Error receiving broadcast " + intent
                        + " in " + mReceiver, e);
                }
            }
            if (receiver.getPendingResult() != null) {
                finish();
            }
            Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
        }
    }
```

在上述代码中，mReceiver 是类 ReceiverDispatcher 的成员变量，其类型是 BroadcastReceiver。此时通过这个 ReceiverDispatcher 实例，即可调用它的 onReceive 函数分发处理这个广播。

13.3 分析 BroadCastReceiver

在 Android 的广播系统中，ActivityManagerService 是整个广播的中心，负责系统中所有广播的注册和发布操作。其中，Android 应用程序注册广播接收器的过程，就是将广播接收器注册到 ActivityManagerService 的过程。在 Android 应用程序中，通过调用类 ContextWrapper 中的函数 registerReceiver 将广播接收器 BroadcastReceiver 注册到 ActivityManagerService 中，而类 ContextWrapper 本身可以使用类 ContextImpl 来注册广播接收器。

13.3.1 MainActivity 的调用

在 Android 的广播系统中，从 MainActivity 开始发起注册广播接收器的操作。在类 MainActivity 中，调用函数 registerReceiver 来注册广播接收器，对应的代码如下所示：

```
public class MainActivity extends Activity implements OnClickListener {
……
    public void onResume() {
        super.onResume();
        IntentFilter counterActionFilter = new IntentFilter(CounterService.BROADCAST_
COUNTER_ACTION);
        registerReceiver(counterActionReceiver, counterActionFilter);
    }
```

在上述代码中，函数 onResume 通过其父类 ContextWrapper 中的函数 registerReceiver 注册一

个 BroadcastReceiver 实例 counterActionReceiver，并且通过 IntentFilter 实例 counterActionFilter 通知 ActivityManagerService 要订阅的广播是 CounterService.BROADCAST_COUNTER_ACTION 类型。这样当 ActivityManagerService 接收 CounterService.BROADCAST_COUNTER_ACTION 类型的广播时，就会分发给 counterActionReceiver 实例的 onReceive 函数。

13.3.2 注册广播接收者

再看类 ContextWrapper 中的函数 registerReceiver，功能是在 ActivityManagerService 中注册广播接收者。函数 registerReceiver 在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义，具体实现代码如下所示：

```
public Intent registerReceiver(
    BroadcastReceiver receiver, IntentFilter filter) {
    return mBase.registerReceiver(receiver, filter);
}
```

在上述代码中，其实是调用了类 ContextImpl 中的函数 registerReceiver 来注册了广播接收者。函数 registerReceiver 在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义，具体实现代码如下所示：

```
public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter) {
    return registerReceiver(receiver, filter, null, null);
}

@Override
public Intent registerReceiver(BroadcastReceiver receiver, IntentFilter filter,
        String broadcastPermission, Handler scheduler) {
    if (receiver == null) {
        // Allow retrieving current sticky broadcast; this is safe since we
        // aren't actually registering a receiver.
        return super.registerReceiver(null, filter, broadcastPermission, scheduler);
    } else {
        throw new ReceiverCallNotAllowedException(
                "BroadcastReceiver components are not allowed to register to receive
                intents");
    }
}
private Intent registerReceiverInternal(BroadcastReceiver receiver, int userId,
        IntentFilter filter, String broadcastPermission,
        Handler scheduler, Context context) {
    IIntentReceiver rd = null;
    if (receiver != null) {
        if (mPackageInfo != null && context != null) {
            if (scheduler == null) {
                scheduler = mMainThread.getHandler();
            }
            rd = mPackageInfo.getReceiverDispatcher(
                receiver, context, scheduler,
                mMainThread.getInstrumentation(), true);
        } else {
            if (scheduler == null) {
                scheduler = mMainThread.getHandler();
            }
            rd = new LoadedApk.ReceiverDispatcher(
                    receiver, context, scheduler, null, true).getIIntentReceiver();
        }
    }
    try {
        return ActivityManagerNative.getDefault().registerReceiver(
                mMainThread.getApplicationThread(), mBasePackageName,
                rd, filter, broadcastPermission, userId);
    } catch (RemoteException e) {
        return null;
    }
}
```

在上述代码中，变量 mPackageInfo 是一个 LoadedApk 实例，功能是处理广播接收者的信息。

参数 broadcastPermission 和 scheduler 都是 null，而参数 context 是通过调用函数 getOuterContext 得到的，在此处指向了 MainActivity。

13.3.3　获取接口对象

函数 getReceiverDispatcher 的功能是获得一个 IIntentReceiver 接口对象 rd，这是一个 Binder 对象。然后将这个传给 ActivityManagerService，ActivityManagerService 在收到相应的广播时会通过这个 Binder 对象通知 MainActivity 来接收。函数 getReceiverDispatcher 在文件 frameworks/base/core/java/android/app/LoadedApk.java 中定义，具体实现代码如下所示：

```
public IIntentReceiver getReceiverDispatcher(BroadcastReceiver r,
        Context context, Handler handler,
        Instrumentation instrumentation, boolean registered) {
    synchronized (mReceivers) {
        LoadedApk.ReceiverDispatcher rd = null;
        HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher> map = null;
        if (registered) {
            map = mReceivers.get(context);
            if (map != null) {
                rd = map.get(r);
            }
        }
        if (rd == null) {
            rd = new ReceiverDispatcher(r, context, handler,
                    instrumentation, registered);
            if (registered) {
                if (map == null) {
                    map = new HashMap<BroadcastReceiver, LoadedApk.ReceiverDispatcher>();
                    mReceivers.put(context, map);
                }
                map.put(r, rd);
            }
        } else {
            rd.validate(context, handler);
        }
        rd.mForgotten = false;
        return rd.getIIntentReceiver();
    }
}
static final class ReceiverDispatcher {

    final static class InnerReceiver extends IIntentReceiver.Stub {
        final WeakReference<LoadedApk.ReceiverDispatcher> mDispatcher;
        final LoadedApk.ReceiverDispatcher mStrongRef;

        InnerReceiver(LoadedApk.ReceiverDispatcher rd, boolean strong) {
            mDispatcher = new WeakReference<LoadedApk.ReceiverDispatcher>(rd);
            mStrongRef = strong ? rd : null;
        }
        public void performReceive(Intent intent, int resultCode, String data,
                Bundle extras, boolean ordered, boolean sticky, int sendingUser) {
            LoadedApk.ReceiverDispatcher rd = mDispatcher.get();
            if (ActivityThread.DEBUG_BROADCAST) {
                int seq = intent.getIntExtra("seq", -1);
                Slog.i(ActivityThread.TAG, "Receiving broadcast " + intent.getAction() +
                " seq=" + seq + " to " + (rd != null ? rd.mReceiver : null));
            }
            if (rd != null) {
                rd.performReceive(intent, resultCode, data, extras,
                        ordered, sticky, sendingUser);
            } else {
                // The activity manager dispatched a broadcast to a registered
                // receiver in this process, but before it could be delivered the
                // receiver was unregistered.  Acknowledge the broadcast on its
                // behalf so that the system's broadcast sequence can continue.
                if (ActivityThread.DEBUG_BROADCAST) Slog.i(ActivityThread.TAG,
                        "Finishing broadcast to unregistered receiver");
                IActivityManager mgr = ActivityManagerNative.getDefault();
```

```
                try {
                    if (extras != null) {
                        extras.setAllowFds(false);
                    }
                    mgr.finishReceiver(this, resultCode, data, extras, false);
                } catch (RemoteException e) {
                    Slog.w(ActivityThread.TAG, "Couldn't finish broadcast to unregistered
                    receiver");
                }
            }
        }
    }

    final IIntentReceiver.Stub mIIntentReceiver;
    final BroadcastReceiver mReceiver;
    final Context mContext;
    final Handler mActivityThread;
    final Instrumentation mInstrumentation;
    final boolean mRegistered;
    final IntentReceiverLeaked mLocation;
    RuntimeException mUnregisterLocation;
    boolean mForgotten;
.....
    ReceiverDispatcher(BroadcastReceiver receiver, Context context,
            Handler activityThread, Instrumentation instrumentation,
            boolean registered) {
        if (activityThread == null) {
            throw new NullPointerException("Handler must not be null");
        }

        mIIntentReceiver = new InnerReceiver(this, !registered);
        mReceiver = receiver;
        mContext = context;
        mActivityThread = activityThread;
        mInstrumentation = instrumentation;
        mRegistered = registered;
        mLocation = new IntentReceiverLeaked(null);
        mLocation.fillInStackTrace();
    }
```

在上述函数 getReceiverDispatcher 中，首先检查参数 r 是否已经存在相应的 ReceiverDispatcher，如果有，则就直接返回；否则就创建一个 ReceiverDispatcher，并且以 r 为 Key 值保在一个 HashMap 中。只要给定一个 Activity 和 BroadcastReceiver，就可以查看在 LoadedApk 中是否已经存在相应的广播接收发布器 ReceiverDispatcher。

接下来看文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中的函数 registerReceiver，具体实现代码如下所示：

```
public Intent registerReceiver(IApplicationThread caller, String packageName,
        IIntentReceiver receiver,
        IntentFilter filter, String perm, int userId) throws RemoteException
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    data.writeString(packageName);
    data.writeStrongBinder(receiver != null ? receiver.asBinder() : null);
    filter.writeToParcel(data, 0);
    data.writeString(perm);
    data.writeInt(userId);
    mRemote.transact(REGISTER_RECEIVER_TRANSACTION, data, reply, 0);
    reply.readException();
    Intent intent = null;
    int haveIntent = reply.readInt();
    if (haveIntent != 0) {
        intent = Intent.CREATOR.createFromParcel(reply);
    }
    reply.recycle();
    data.recycle();
```

```
        return intent;
    }
```

在上述代码中，通过 Binder 驱动程序来到 ActivityManagerService 中的函数 registerReceiver 中。

13.3.4　处理进程间的通信请求

接下来看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 registerReceiver，功能是处理组件 Broadcounter 发出的类型为 REGISTER_RECEIVER_TRANSACTION 的进程间的通信请求。函数 registerReceiver 的具体实现代码如下所示：

```
    public Intent registerReceiver(IApplicationThread caller, String callerPackage,
            IIntentReceiver receiver, IntentFilter filter, String permission, int userId)
{
        enforceNotIsolatedCaller("registerReceiver");
        int callingUid;
        int callingPid;
        synchronized(this) {
        //调用函数 registerReceiver 的应用程序进程记录块，MainActivity 就是在里面启动
            ProcessRecord callerApp = null;
            if (caller != null) {
                callerApp = getRecordForAppLocked(caller);
                if (callerApp == null) {
                    throw new SecurityException(
                            "Unable to find app for caller " + caller
                            + " (pid=" + Binder.getCallingPid()
                            + ") when registering receiver " + receiver);
                }
                if (callerApp.info.uid != Process.SYSTEM_UID &&
                        !callerApp.pkgList.contains(callerPackage)) {
                    throw new SecurityException("Given caller package " + callerPackage
                            + " is not running in process " + callerApp);
                }
                callingUid = callerApp.info.uid;
                callingPid = callerApp.pid;
            } else {
                callerPackage = null;
                callingUid = Binder.getCallingUid();
                callingPid = Binder.getCallingPid();
            }

            userId = this.handleIncomingUser(callingPid, callingUid, userId,
                    true, true, "registerReceiver", callerPackage);
//传进来的 filter 只有一个 action，就是前面描述的 CounterService.BROADCAST_COUNTER_ACTION
            List allSticky = null;

            // Look for any matching sticky broadcasts...
            Iterator actions = filter.actionsIterator();
            if (actions != null) {
                while (actions.hasNext()) {
                    String action = (String)actions.next();
//通过函数 getStickiesLocked 查找是否存在对应的 sticky intent 列表
                    allSticky = getStickiesLocked(action, filter, allSticky,
                            UserHandle.USER_ALL);
                    allSticky = getStickiesLocked(action, filter, allSticky,
                            UserHandle.getUserId(callingUid));
                }
            } else {
                allSticky = getStickiesLocked(null, filter, allSticky,
                        UserHandle.USER_ALL);
                allSticky = getStickiesLocked(null, filter, allSticky,
                        UserHandle.getUserId(callingUid));
            }

            // The first sticky in the list is returned directly back to
            // the client.
            Intent sticky = allSticky != null ? (Intent)allSticky.get(0) : null;

            if (DEBUG_BROADCAST) Slog.v(TAG, "Register receiver " + filter
                    + ": " + sticky);
```

```
            if (receiver == null) {
                return sticky;
            }
//如果传进来的 receiver 不为 null，则继续往下执行
            ReceiverList rl
                = (ReceiverList)mRegisteredReceivers.get(receiver.asBinder());
            if (rl == null) {
                rl = new ReceiverList(this, callerApp, callingPid, callingUid,
                        userId, receiver);
                if (rl.app != null) {
                    rl.app.receivers.add(rl);
                } else {
                    try {
                        receiver.asBinder().linkToDeath(rl, 0);
                    } catch (RemoteException e) {
                        return sticky;
                    }
                    rl.linkedToDeath = true;
                }
                mRegisteredReceivers.put(receiver.asBinder(), rl);
            } else if (rl.uid != callingUid) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for uid " + callingUid
                        + " was previously registered for uid " + rl.uid);
            } else if (rl.pid != callingPid) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for pid " + callingPid
                        + " was previously registered for pid " + rl.pid);
            } else if (rl.userId != userId) {
                throw new IllegalArgumentException(
                        "Receiver requested to register for user " + userId
                        + " was previously registered for user " + rl.userId);
            }
//将广播接收器 receiver 保存起来，并没有把它和 filter 关联起来
            BroadcastFilter bf = new BroadcastFilter(filter, rl, callerPackage,
                    permission, callingUid, userId);
            rl.add(bf);
            if (!bf.debugCheck()) {
                Slog.w(TAG, "==> For Dynamic broadast");
            }
            mReceiverResolver.addFilter(bf);

            // Enqueue broadcasts for all existing stickies that match
            // this filter.
            if (allSticky != null) {
                ArrayList receivers = new ArrayList();
                receivers.add(bf);

                int N = allSticky.size();
                for (int i=0; i<N; i++) {
                    Intent intent = (Intent)allSticky.get(i);
                    BroadcastQueue queue = broadcastQueueForIntent(intent);
                    BroadcastRecord r = new BroadcastRecord(queue, intent, null,
                            null, -1, -1, null, AppOpsManager.OP_NONE, receivers, null, 0,
                            null, null, false, true, true, -1);
                    queue.enqueueParallelBroadcastLocked(r);
                    queue.scheduleBroadcastsLocked();
                }
            }
            return sticky;
        }
    }
```

在上述代码中，使用创建的 BroadcastFilter 把广播接收器列表 rl 和 filter 关联起来，然后保存在 ActivityManagerService 中的成员变量 mReceiverResolver 中。

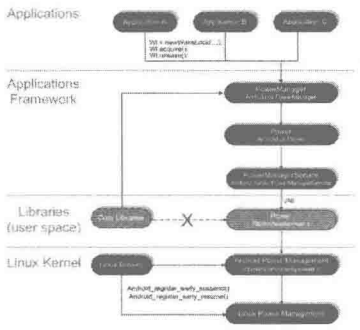
第 14 章　分析电源管理系统

Android 系统中的电源管理系统是 Android Power Management，这是一个基于标准 Linux 电源管理的轻量级 Android 电源管理系统。整个电源管理系统分为 4 大部分，分别是应用层、框架层、HAL 层和 Kernel 层。在本章的内容中，将详细分析 Android 系统中 Power Management 模块的基本架构知识。

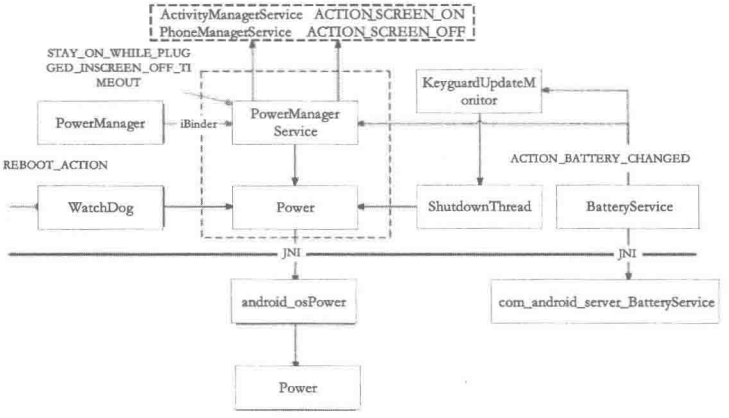
14.1 Power Management 架构基础

在 Android 系统中，电源管理模块 Android Power Management 的整体架构如图 14-1 所示。

从图 14-1 所示的架构可以看出，电源管理主要是通过锁和定时器来切换系统的状态，使系统的功耗降至最低。为了使读者了解得更加清晰，请看图 14-2 所示的电源管理详细架构图。



▲图 14-1　Android Power Management（电源管理系统）的整体架构



▲图 14-2　电源管理系统的详细架构图

由此可见，整个电源管理模块分为 4 大部分，分别是应用层、框架层、HAL 层和 Kernel 层。整个运作流程如下所示：

wake_lock → setScreenState(off) → request_suspend_state → early_suspend → wake_unlock → suspend→late suspend→sleep→wakeup→early resume→resume→late resume

在本章后面的内容中，将详细分析上述 4 个层次的具体实现过程。

14.2 分析 Framework 层

我们知道，Android 系统的 Framework 层是接口层，为上面的应用层提供了拿来即用的接口。在 Android Power Management 系统中，Framework 层涉及如下所示的文件：

- frameworks/base/core/java/android/os/PowerManager.java；
- frameworks/base/services/java/com/android/server/power/PowerManagerService.java。

在本节的内容中，将详细介绍 Power Management 系统中 Framework 层文件的实现过程。

14.2.1 文件 PowerManager.java

文件 PowerManager.java 是提供给应用层调用的，最终的核心实现在文件 PowerManager Service.java 中实现。在此文件 PowerManager.java 中定义了类 android.os.PowerManager，功能是控制设备的电源状态的切换。当通过函数 getSystemService(Context.POWER_SERVICE) 获取 PowerManager 对象的时候，是通过构造函数 PowerManager(IPowerManagerservice, Handler handler){} 来创建并获取的，而此处 IPowerManager 则是创建 PowerManager 实例的核心，而 IPowerManager 则是由 PowerManagerService 实现，所以，从本质上说 PowerManager 的大部分方法是由 PowerManager Service 实现的。

在接下来的内容中，将详细分析文件 PowerManager.java 的具体实现流程。

（1）定义类 PowerManager 和接口函数，具体代码如下所示：

```
public final class PowerManager {
    private static final String TAG = "PowerManager";
    public static final int PARTIAL_WAKE_LOCK = 0x00000001;
    @Deprecated
    public static final int SCREEN_DIM_WAKE_LOCK = 0x00000006;
    @Deprecated
    public static final int SCREEN_BRIGHT_WAKE_LOCK = 0x0000000a;
    @Deprecated
    public static final int FULL_WAKE_LOCK = 0x0000001a;
    public static final int PROXIMITY_SCREEN_OFF_WAKE_LOCK = 0x00000020;
    public static final int WAKE_LOCK_LEVEL_MASK = 0x0000ffff;
    public static final int ACQUIRE_CAUSES_WAKEUP = 0x10000000;
    public static final int ON_AFTER_RELEASE = 0x20000000;
    public static final int WAIT_FOR_PROXIMITY_NEGATIVE = 1;
    public static final int BRIGHTNESS_ON = 255;
```

（2）定义对外接口函数，以实现对电源状态的控制管理。其中函数 goToSleep 的功能是强制设备进入 Sleep 状态，函数 wakeUp 的功能是强制设备进入 wakeUp 状态：

```
public void goToSleep(long time) {
    try {
        mService.goToSleep(time, GO_TO_SLEEP_REASON_USER);
    } catch (RemoteException e) {
    }
}
public void wakeUp(long time) {
    try {
        mService.wakeUp(time);
    } catch (RemoteException e) {
    }
}
```

（3）定义函数 userActivity，功能是当发生 User activity 事件时，电源设备会被切换到 Full on 的状态，并同时 Reset Screen off timer，具体代码如下所示：

```
public void userActivity(long when, boolean noChangeLights) {
    try {
        mService.userActivity(when, USER_ACTIVITY_EVENT_OTHER,
                noChangeLights ? USER_ACTIVITY_FLAG_NO_CHANGE_LIGHTS : 0);
    } catch (RemoteException e) {
    }
}
```

14.2.2 提供 PowerManager 功能

文件 PowerManagerService.java 是 Power Management 系统中整个 Framework 层文件的核心，这个类的作用就是提供 PowerManager 的功能，以及整个电源管理状态机的运行。PowerManager Service 服务是 Android 系统的上层的电源管理服务，主要负责系统待机、屏幕背光、按键背光、键盘背光以及用户事件的处理。通过锁的申请与释放，以及默认的待机时间来控制系统的待机状态；通过系统默认关闭屏的时间，以及用户操作的事件状态控制背光的亮和暗。另外，PowerManagerService 服务还包括了光线、距离传感器上层查询与控制，LCD 亮度的调节最终也

是由该服务完成。在本章前面介绍的文件 PowerManager.java 只是定义了各个对外接口函数，而文件 PowerManagerService.java 则定义了各个接口函数的具体实现。

在接下来的内容中，将详细分析文件 PowerManagerService.java 的具体实现过程。

1. 定义常量和变量

在文件的开始定义服务类 PowerManagerService，并定义需要的变量和常量，主要实现代码如下所示：

```
private static final int LOCK_MASK = PowerManager.PARTIAL_WAKE_LOCK
                                   | PowerManager.SCREEN_DIM_WAKE_LOCK
                                   | PowerManager.SCREEN_BRIGHT_WAKE_LOCK
                                   | PowerManager.FULL_WAKE_LOCK
                                   | PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK;

// time since last state: time since last event:
// The short keylight delay comes from secure settings; this is the default.
private static final int SHORT_KEYLIGHT_DELAY_DEFAULT = 6000; // t+6 sec
private static final int MEDIUM_KEYLIGHT_DELAY = 15000;       // t+15 sec
private static final int LONG_KEYLIGHT_DELAY = 6000;        // t+6 sec
private static final int LONG_DIM_TIME = 7000;             // t+N-5 sec

// How long to wait to debounce light sensor changes in milliseconds
private static final int LIGHT_SENSOR_DELAY = 2000;//光线传感器时延

// light sensor events rate in microseconds
private static final int LIGHT_SENSOR_RATE = 1000000;//光线传感器频率

// For debouncing the proximity sensor in milliseconds
private static final int PROXIMITY_SENSOR_DELAY = 1000;//距离传感器时延

// trigger proximity if distance is less than 5 cm
private static final float PROXIMITY_THRESHOLD = 5.0f;//距离传感器距离范围

// Cached secure settings; see updateSettingsValues()
private int mShortKeylightDelay = SHORT_KEYLIGHT_DELAY_DEFAULT;//键盘灯短暂时延

// Default timeout for screen off, if not found in settings database = 15 seconds.
private static final int DEFAULT_SCREEN_OFF_TIMEOUT = 15000;
//默认屏幕超时时间,从 Settings 中获取

// flags for setPowerState
private static final int SCREEN_ON_BIT         = 0x00000001;
private static final int SCREEN_BRIGHT_BIT     = 0x00000002;
private static final int BUTTON_BRIGHT_BIT     = 0x00000004;
private static final int KEYBOARD_BRIGHT_BIT   = 0x00000008;
private static final int BATTERY_LOW_BIT       = 0x00000010;

// values for setPowerState

// SCREEN_OFF == everything off
private static final int SCREEN_OFF = 0x00000000;//屏幕灭掉，进入睡眠状态

// SCREEN_DIM == screen on, screen backlight dim
private static final int SCREEN_DIM = SCREEN_ON_BIT;//屏幕灭掉，依然在工作状态

// SCREEN_BRIGHT == screen on, screen backlight bright
private static final int SCREEN_BRIGHT = SCREEN_ON_BIT | SCREEN_BRIGHT_BIT;
//屏幕亮，处于工作状态

// SCREEN_BUTTON_BRIGHT == screen on, screen and button backlights bright
private static final int SCREEN_BUTTON_BRIGHT = SCREEN_BRIGHT | BUTTON_BRIGHT_BIT;
//屏幕亮，按键灯亮

// SCREEN_BUTTON_BRIGHT == screen on, screen, button and keyboard backlights bright
private static final int ALL_BRIGHT = SCREEN_BUTTON_BRIGHT | KEYBOARD_BRIGHT_BIT;
//按键灯亮，键盘灯亮

// used for noChangeLights in setPowerState()
```

```
private static final int LIGHTS_MASK = SCREEN_BRIGHT_BIT | BUTTON_BRIGHT_BIT |
KEYBOARD_BRIGHT_BIT;//屏幕亮，按键灯亮，键盘灯亮

boolean mAnimateScreenLights = true;

static final int ANIM_STEPS = 60/4;
// Slower animation for autobrightness changes
static final int AUTOBRIGHTNESS_ANIM_STEPS = 60;

// These magic numbers are the initial state of the LEDs at boot.  Ideally
// we should read them from the driver, but our current hardware returns 0
// for the initial value.  Oops!
static final int INITIAL_SCREEN_BRIGHTNESS = 255;//屏幕初始状态：亮
static final int INITIAL_BUTTON_BRIGHTNESS = Power.BRIGHTNESS_OFF;//按键灯初始状态：灭
static final int INITIAL_KEYBOARD_BRIGHTNESS = Power.BRIGHTNESS_OFF;//键盘灯初始状态：灭

private final int MY_UID;
private final int MY_PID;

private boolean mDoneBooting = false;
private boolean mBootCompleted = false;//开机完成标志位
private int mStayOnConditions = 0;
private final int[] mBroadcastQueue = new int[] { -1, -1, -1 };
private final int[] mBroadcastWhy = new int[3];
private boolean mPreparingForScreenOn = false;
private boolean mSkippedScreenOn = false;
private boolean mInitialized = false;
private int mPartialCount = 0;
private int mPowerState;
// mScreenOffReason can be WindowManagerPolicy.OFF_BECAUSE_OF_USER,
// WindowManagerPolicy.OFF_BECAUSE_OF_TIMEOUT or WindowManagerPolicy.OFF_BECAUSE_OF_
PROX_SENSOR
private int mScreenOffReason;
private int mUserState;
private boolean mKeyboardVisible = false;
private int mStartKeyThreshold = 0;
private boolean mUserActivityAllowed = true;
private int mProximityWakeLockCount = 0;
private boolean mProximitySensorEnabled = false;//距离传感器是否可用
private boolean mProximitySensorActive = false;//当前距离传感器是否工作
private int mProximityPendingValue = -1; // -1 == nothing, 0 == inactive, 1 == active
private long mLastProximityEventTime;
private int mScreenOffTimeoutSetting;//屏幕超时设置
private int mMaximumScreenOffTimeout = Integer.MAX_VALUE;
private int mKeylightDelay;
private int mDimDelay;
private int mScreenOffDelay;
private int mWakeLockState;
private long mLastEventTime = 0;
private long mScreenOffTime;
private volatile WindowManagerPolicy mPolicy;
private final LockList mLocks = new LockList();
private Intent mScreenOffIntent;
private Intent mScreenOnIntent;
private LightsService mLightsService;//系统 LightsService
private Context mContext;
private LightsService.Light mLcdLight;//屏
private LightsService.Light mButtonLight;//按键灯
private LightsService.Light mKeyboardLight;//键盘灯(若有实体输入法按键)
private LightsService.Light mAttentionLight;//通知等(若有信号灯)
private UnsynchronizedWakeLock mBroadcastWakeLock;//广播同步锁
private UnsynchronizedWakeLock mStayOnWhilePluggedInScreenDimLock;
private UnsynchronizedWakeLock mStayOnWhilePluggedInPartialLock;
private UnsynchronizedWakeLock mPreventScreenOnPartialLock;
private UnsynchronizedWakeLock mProximityPartialLock;
private HandlerThread mHandlerThread;
private HandlerThread mScreenOffThread;
private Handler mScreenOffHandler;
private Handler mHandler;

//计时器线程，主要完成管理屏幕超时操作，如当有用户点击屏幕时
```

```
//该计时器重新开始计时，直到无任何操作，且到屏幕时延最大时间，将屏幕灭掉
private final TimeoutTask mTimeoutTask = new TimeoutTask();
private final BrightnessState mScreenBrightness
      = new BrightnessState(SCREEN_BRIGHT_BIT);//亮度管理
private boolean mStillNeedSleepNotification;
private boolean mIsPowered = false;
private IActivityManager mActivityService;
private IBatteryStats mBatteryStats;
private BatteryService mBatteryService;//电池服务
private SensorManager mSensorManager;//Sensor 管理器
private Sensor mProximitySensor;//距离传感器
private Sensor mLightSensor;//光线传感器
private Sensor mLightSensorKB;//光线传感器
private boolean mLightSensorEnabled;//光线传感器是否可用
private float mLightSensorValue = -1;
private boolean mProxIgnoredBecauseScreenTurnedOff = false;
private int mHighestLightSensorValue = -1;
private boolean mLightSensorPendingDecrease = false;
private boolean mLightSensorPendingIncrease = false;
private float mLightSensorPendingValue = -1;
private int mLightSensorScreenBrightness = -1;
private int mLightSensorButtonBrightness = -1;
private int mLightSensorKeyboardBrightness = -1;
private boolean mDimScreen = true;
private boolean mIsDocked = false;
private long mNextTimeout;
private volatile int mPokey = 0;
private volatile boolean mPokeAwakeOnSet = false;
private volatile boolean mInitComplete = false;
private final HashMap<IBinder,PokeLock> mPokeLocks = new HashMap<IBinder,PokeLock>();
// mLastScreenOnTime is the time the screen was last turned on
private long mLastScreenOnTime;
private boolean mPreventScreenOn;
private int mScreenBrightnessOverride = -1;
private int mButtonBrightnessOverride = -1;
private int mScreenBrightnessDim;
private boolean mUseSoftwareAutoBrightness;
private boolean mAutoBrightessEnabled;
private int[] mAutoBrightnessLevels;
private int[] mLcdBacklightValues;
private int[] mButtonBacklightValues;
private int[] mKeyboardBacklightValues;
private int mLightSensorWarmupTime;
boolean mUnplugTurnsOnScreen;
private int mWarningSpewThrottleCount;
private long mWarningSpewThrottleTime;
private int mAnimationSetting = ANIM_SETTING_OFF;

// Must match with the ISurfaceComposer constants in C++.
private static final int ANIM_SETTING_ON = 0x01;
private static final int ANIM_SETTING_OFF = 0x10;

// Used when logging number and duration of touch-down cycles
private long mTotalTouchDownTime;
private long mLastTouchDown;
private int mTouchCycles;

// could be either static or controllable at runtime
private static final boolean mSpew = false;
private static final boolean mDebugProximitySensor = (false || mSpew);
private static final boolean mDebugLightSensor = (false || mSpew);

private native void nativeInit();
private native void nativeSetPowerState(boolean screenOn, boolean screenBright);
private native void nativeStartSurfaceFlingerAnimation(int mode);
```

在文件 PowerManagerService 中，其中需要重点说明的变量如下所示。

- mDirty：功能是表示 power state 的变化，在系统中一共定义了 12 个与之类似的变化，每一个 state 对应一个固定的数字，都是 2 的倍数。这样，当有若干个状态一起变化时就会按位取或，

这样不但会得到一个唯一的结果，而且可以准确地标示出各个状态的变化。

- mWakefulness：功能是标示 device 处于是醒着的、还是睡眠中的状态，或者处于两者之间的一种状态。这个状态是和 display 的电源状态不同的，display 的电源状态是独立管理的。这个变量用来标示 DIRTY_WAKEFULNESS 这个 power state 下的一个具体的内容。例如，当系统进入 Draaming 时，首先变化的是 mDirty，在 mDirty 中监听 DIRTY_WAKEFULNESS 的位置是否发生了变化，此时只是知道，DIRTY_WAKEFULNESS 发生了变化，并不知道 DIRTY_WAKEFULNESS 发生了怎样的变化，并不知道 wakefulness 发生了怎样的变化。如果需要进一步了解系统 wakefulness 变成了什么，则需要查看 mWakefulness 的内容。

2. 开机启动及处理

（1）当 Android 系统启动时，会调用文件 SystemServer.java 中的接口函数 run，在此函数中将 power 服务加入到系统服务中，具体代码如下所示：

```
power = new PowerManagerService();
ServiceManager.addService(Context.POWER_SERVICE, power);
```

其实在文件 SystemServer.java 中还定义了多种类型的服务加入到了系统服务中，具体代码如下所示：

```
try {
    // Wait for installd to finished starting up so that it has a chance to
    // create critical directories such as /data/user with the appropriate
    // permissions.  We need this to complete before we initialize other services.
    Slog.i(TAG, "Waiting for installd to be ready.");
    installer = new Installer();
    installer.ping();
    Slog.i(TAG, "Power Manager");
    power = new PowerManagerService();
    ServiceManager.addService(Context.POWER_SERVICE, power);
    Slog.i(TAG, "Activity Manager");
    context = ActivityManagerService.main(factoryTest);
    Slog.i(TAG, "Display Manager");
    display = new DisplayManagerService(context, wmHandler, uiHandler);
    ServiceManager.addService(Context.DISPLAY_SERVICE, display, true);
    Slog.i(TAG, "Telephony Registry");
    telephonyRegistry = new TelephonyRegistry(context);
    ServiceManager.addService("telephony.registry", telephonyRegistry);
    Slog.i(TAG, "Scheduling Policy");
    ServiceManager.addService(Context.SCHEDULING_POLICY_SERVICE,
            new SchedulingPolicyService());
    AttributeCache.init(context);
    if (!display.waitForDefaultDisplay()) {
        reportWtf("Timeout waiting for default display to be initialized.",
                new Throwable());
    }
    Slog.i(TAG, "Package Manager");
    // Only run "core" apps if we're encrypting the device.
    String cryptState = SystemProperties.get("vold.decrypt");
    if (ENCRYPTING_STATE.equals(cryptState)) {
        Slog.w(TAG, "Detected encryption in progress - only parsing core apps");
        onlyCore = true;
    } else if (ENCRYPTED_STATE.equals(cryptState)) {
        Slog.w(TAG, "Device encrypted - only parsing core apps");
        onlyCore = true;
    }
    pm = PackageManagerService.main(context, installer,
            factoryTest != SystemServer.FACTORY_TEST_OFF,
            onlyCore);
    boolean firstBoot = false;
    try {
        firstBoot = pm.isFirstBoot();
    } catch (RemoteException e) {
    }
    ActivityManagerService.setSystemProcess();
    Slog.i(TAG, "Entropy Mixer");
```

```
ServiceManager.addService("entropy", new EntropyMixer(context));
Slog.i(TAG, "User Service");
ServiceManager.addService(Context.USER_SERVICE,
        UserManagerService.getInstance());
mContentResolver = context.getContentResolver();
// The AccountManager must come before the ContentService
try {
    Slog.i(TAG, "Account Manager");
    accountManager = new AccountManagerService(context);
    ServiceManager.addService(Context.ACCOUNT_SERVICE, accountManager);
} catch (Throwable e) {
    Slog.e(TAG, "Failure starting Account Manager", e);
}
Slog.i(TAG, "Content Manager");
contentService = ContentService.main(context,
        factoryTest == SystemServer.FACTORY_TEST_LOW_LEVEL);
Slog.i(TAG, "System Content Providers");
ActivityManagerService.installSystemProviders();
Slog.i(TAG, "Lights Service");
lights = new LightsService(context);
Slog.i(TAG, "Battery Service");
battery = new BatteryService(context, lights);
ServiceManager.addService("battery", battery);
Slog.i(TAG, "Vibrator Service");
vibrator = new VibratorService(context);
ServiceManager.addService("vibrator", vibrator);
```

（2）当光感服务与电池管理服务都开始启动后，开始进行初始化 power 服务的工作。初始化工作是通过调用函数 init 实现的，具体实现如下加粗代码所示：

```
// only initialize the power service after we have started the
// lights service, content providers and the battery service.
power.init(context, lights, ActivityManagerService.self(), battery,
        BatteryStatsService.getService(), display);
Slog.i(TAG, "Alarm Manager");
alarm = new AlarmManagerService(context);
ServiceManager.addService(Context.ALARM_SERVICE, alarm);
Slog.i(TAG, "Init Watchdog");
Watchdog.getInstance().init(context, battery, power, alarm,
        ActivityManagerService.self());
```

在函数 init()中实现了一些基本的初始化工作，包括将 lights 和 battery 两个服务实例传入到 power 服务中，这两个服务将与 power 进行交互。除此之外，还开启了如下所示的两线程。

- 开启处理亮度动画线程函数 mScreenBrightnessAnimator.start()

mScreenBrightnessAnimator 是 PowerManagerService 子类 ScreenBrightnessAnimator 的实例，演示代码如下所示：

```
mHandlerThread = new HandlerThread("PowerManagerService")
mHandlerThread.start( );
```

- 初始化线程 initThread

当使用 start 启动电源管理服务后会调用到 run 接口，并在其中回调到子类中的 protected void onLooperPrepared()，该接口又调用到 initInThread()，功能是实现一些值的初始化工作，并标识为“mInitComplete = true;”，发现为“true”的标识后，mHandlerThread.notifyAll()会通知创建 mHandler ThreadLooper 实例，该实例在函数 systemReady()中被 SystemSensorManager(mHandlerThread.get Looper())所使用。另外，在 initThread 线程中还实现了对 mSettings.addObserver(settingsObserver) 的监听，如果用户在系统中变更了关于背光时间或是否启用光感等服务，PowerManagerService 可以获取到最新的状态值，这一功能需要使用函数 update()进行更新。

（3）继续看函数 init()，最后调用函数 forceUserActivityLocked()关闭服务，并标志初始化工作完成。

3. 与系统其他模块之间的交互

在 Android 系统中，PowerManagerService 作为 Framework 中重要的能源管理模块，除了与应

用程序交互之外，还需要与系统中其他模块进行配合，在提供良好的能源管理同时提供友好的用户体验。Android 系统除了提供公共接口与其他模块交互外，还提供了 BroadCast 机制以对系统中发生的重要变化做出反应。在表 14-1 中列出了在 PowerManagerService 中注册的 Receiver，以及这些 Receiver 监听的事件和处理方法。

表 14-1　　PowerManagerService 中注册的 Receiver 说明

BatteryReceiver	ACTION_BATTERY_CHANGED	handleBatterStateChangeLocked()
BootCompleteReceiver	ACTION_BOOT_COMPLETED	startWatchingForBootAnimationFinished()
userSwitchReceiver	ACTION_USER_SWITCHED	handleSettingsChangedLocked
DockReceiver	ACTION_DOCK_EVENT	updatePowerStateLocked
DreamReceiver	ACTION_DREAMING_STARTED ACTION_DREAMING_STOPPED	scheduleSandmanLocked

在文件 PowerManagerService.java 中除了注册上述 5 个 Receiver 之外，还定义了一个 SettingsObserver 以监视系统中以下属性的变化。

- SCREENSAVER_ENABLE：屏保的功能开启。
- SCREENSAVER_ACTIVE_ON_SLEEP：在睡眠时屏保启动。
- SCREENSAVER_ACTIVE_ON_DOCK：连接底座并且屏保启动。
- SCREEN_OFF_TIMEOUT：休眠时间。
- STAY_ON_PLUGGED_IN：有硬件插入并且屏幕开启。
- SCREEN_BRIGHTNESS：屏幕的亮度。
- SCREEN_BRIGHTNESS_MODE：屏幕亮度的模式。

SettingObserver 会监视到上述属性发生的变化，并且会调用 SettingObserver 中的 onChange 方法：

```
public void onChange(boolean selfChange, Uri uri) {
          synchronized (mLock) {
             handleSettingsChangedLocked();
          }
       }
```

由此可见，PowerManagerService 不但能够接收用户的请求，被动地去做一些操作，而且还要主动地监视系统中一些重要属性的变化和重要事件的发生。无论是处理主动还是被动的操作，在上面都一一列出了对应的处理函数。

4. 分析核心函数

（1）进入休眠状态

函数 goToSleep 的功能是进入休眠状态，具体实现代码如下所示：

```
@Override // Binder call
   public void goToSleep(long eventTime, int reason) {
      if (eventTime > SystemClock.uptimeMillis()) {
         throw new IllegalArgumentException("event time must not be in the future");
      }
//权限检查
         mContext.enforceCallingOrSelfPermission(android.Manifest.permission.
DEVICE_POWER, null);

      final long ident = Binder.clearCallingIdentity();
      try {
         goToSleepInternal(eventTime, reason);//这里会调用函数的实现，在 PowerManager
Service 中有很多类似的使用方式，之后的代码中我会直接列出对应方法的实现
      } finally {
         Binder.restoreCallingIdentity(ident);
      }
   }
   private void goToSleepInternal(long eventTime, int reason) {
      synchronized (mLock) {
         if (goToSleepNoUpdateLocked(eventTime, reason)) {
```

```
                updatePowerStateLocked();
            }
        }
    }
```

（2）确定是否需要休眠

函数 goToSleepNoUpdateLocked 的功能是确定是否要休眠，具体实现代码如下所示：

```
private boolean goToSleepNoUpdateLocked(long eventTime, int reason) {
    if (DEBUG_SPEW) {
        Slog.d(TAG, "goToSleepNoUpdateLocked: eventTime=" + eventTime + ", reason=" +
reason);
    }

    if (eventTime < mLastWakeTime || mWakefulness == WAKEFULNESS_ASLEEP
            || !mBootCompleted || !mSystemReady) {
        return false;
    }

    switch (reason) {
        case PowerManager.GO_TO_SLEEP_REASON_DEVICE_ADMIN:
            Slog.i(TAG, "Going to sleep due to device administration policy...");
            break;
        case PowerManager.GO_TO_SLEEP_REASON_TIMEOUT:
            Slog.i(TAG, "Going to sleep due to screen timeout...");
            break;
        default:
            Slog.i(TAG, "Going to sleep by user request...");
            reason = PowerManager.GO_TO_SLEEP_REASON_USER;
            break;
    }

    sendPendingNotificationsLocked();
    mNotifier.onGoToSleepStarted(reason);
    mSendGoToSleepFinishedNotificationWhenReady = true;

    mLastSleepTime = eventTime;
    mDirty |= DIRTY_WAKEFULNESS;
    mWakefulness = WAKEFULNESS_ASLEEP;

    // Report the number of wake locks that will be cleared by going to sleep.
    int numWakeLocksCleared = 0;
    final int numWakeLocks = mWakeLocks.size();
    for (int i = 0; i < numWakeLocks; i++) {
        final WakeLock wakeLock = mWakeLocks.get(i);
        switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
            case PowerManager.FULL_WAKE_LOCK:
            case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
            case PowerManager.SCREEN_DIM_WAKE_LOCK:
                numWakeLocksCleared += 1;
                break;
        }
    }
    EventLog.writeEvent(EventLogTags.POWER_SLEEP_REQUESTED, numWakeLocksCleared);
    return true;
}
```

通过上述代码可知，在此并没有真正地让设备进入到 sleep 休眠状态，而仅仅只是把 PowerManagerService 中一些必要的属性进行了赋值处理。正因为如此，所以，在 Android 系统中可以把多个 power state 属性的多个变化放在一起共同执行，而真正的功能执行者是 updatePowerStateLocked。

注意　在 PowerManagerService 的具体实现代码中，有很多含有“xxxNoUpdateLocked”格式后缀的函数名字，其实现原理都类似于函数 goToSleepNoUpdateLocked。

（3）更新电源状态的锁定

函数 updatePowerStateLocked 的功能是更新电源状态的锁定，也就是把影响到 Power

Management 发生变化的放在一起进行更新，让电源管理机制能够真正地起到作用。函数 updatePowerStateLocked 的具体实现代码如下所示：

```
private void updatePowerStateLocked() {
    if (!mSystemReady || mDirty == 0) {    //如果系统没有准备好，或者 power state 没有发生任
                                           //何变化，这个方法可以不用执行
        return;
    }
    // Phase 0: Basic state updates.
    updateIsPoweredLocked(mDirty);
    updateStayOnLocked(mDirty);
    // Phase 1: Update wakefulness.
    // Loop because the wake lock and user activity computations are influenced
    // by changes in wakefulness.
    final long now = SystemClock.uptimeMillis();
    int dirtyPhase2 = 0;
    for (;;) {
        int dirtyPhase1 = mDirty;
        dirtyPhase2 |= dirtyPhase1;
        mDirty = 0;
        updateWakeLockSummaryLocked(dirtyPhase1);  //在前面解释几个变量的时候，就已经提到了
                                                   //WakeLockSummary 和 UserActivitySummary
        updateUserActivitySummaryLocked(now, dirtyPhase1);  //在这里的两个方法中已经开始用
                                                   //到了。想必通过方法名，大概也已经有所了解其功能了
        if (!updateWakefulnessLocked(dirtyPhase1)) {
            break;
        }
    }
    // Phase 2: Update dreams and display power state.
    updateDreamLocked(dirtyPhase2);
    updateDisplayPowerStateLocked(dirtyPhase2);
    // Phase 3: Send notifications, if needed.
    if (mDisplayReady) {
        sendPendingNotificationsLocked();
    }
    // Phase 4: Update suspend blocker.
    // Because we might release the last suspend blocker here, we need to make sure
    // we finished everything else first!
    updateSuspendBlockerLocked();
}
```

通过上述实现代码可知，函数 updatePowerStateLocked 按照如下 4 个阶段对 Power State（电源状态）进行更新。

- 第 1 阶段：基本状态的更新。

首先执行函数 updateIsPoweredLocked，功能是判断设备是否处于充电状态，如果 DIRTY_BATTERY_STATE 发生了变化，说明设备的电池状态有改变。函数 updateIsPoweredLocked 的具体实现代码如下所示：

```
/**
 * Updates the value of mIsPowered.
 * Sets DIRTY_IS_POWERED if a change occurred.
 */
private void updateIsPoweredLocked(int dirty) {
    if ((dirty & DIRTY_BATTERY_STATE) != 0) {
        final boolean wasPowered = mIsPowered;
        final int oldPlugType = mPlugType;
        mIsPowered = mBatteryService.isPowered(BatteryManager.BATTERY_PLUGGED_ANY);
        mPlugType = mBatteryService.getPlugType();
        mBatteryLevel = mBatteryService.getBatteryLevel();

        if (DEBUG) {
            Slog.d(TAG, "updateIsPoweredLocked: wasPowered=" + wasPowered
                    + ", mIsPowered=" + mIsPowered
                    + ", oldPlugType=" + oldPlugType
                    + ", mPlugType=" + mPlugType
                    + ", mBatteryLevel=" + mBatteryLevel);
        }
```

```
        if (wasPowered != mIsPowered || oldPlugType != mPlugType) {
            mDirty |= DIRTY_IS_POWERED;

            // Update wireless dock detection state.
            final boolean dockedOnWirelessCharger = mWirelessChargerDetector.update(
                    mIsPowered, mPlugType, mBatteryLevel);

            // Treat plugging and unplugging the devices as a user activity.
            // Users find it disconcerting when they plug or unplug the device
            // and it shuts off right away.
            // Some devices also wake the device when plugged or unplugged because
            // they don't have a charging LED.
            final long now = SystemClock.uptimeMillis();
            if (shouldWakeUpWhenPluggedOrUnpluggedLocked(wasPowered, oldPlugType,
                    dockedOnWirelessCharger)) {
                wakeUpNoUpdateLocked(now);
            }
            userActivityNoUpdateLocked(
                    now, PowerManager.USER_ACTIVITY_EVENT_OTHER, 0, Process.SYSTEM_UID);

            // Tell the notifier whether wireless charging has started so that
            // it can provide feedback to the user.
            if (dockedOnWirelessCharger) {
                mNotifier.onWirelessChargingStarted();
            }
        }
    }
}
```

然后通过对比判断（通过电池状态前后的变化和充电状态的变化来判断）确定是否处于充电状态，会在 mDirty 中标记出充电方式的改变，并同时根据充电状态的变化进行一些相应的处理，同时在处理是否在充电或者充电方式的改变时，都会认为是发生了一次用户事件或者称为用户活动的发生。

接着执行函数 updateStayOnLocked，功能是更新 device 是否处于开启状态。此函数也是通过 mStayOn 发生的前后变化作为判断依据，如果 device 的属性 Settings.Global.STAY_ON_WHILE_PLUGGED_IN 为置位，并且没有达到电池充电时持续开屏时间的最大值（也就是说，在插入电源后的一段时间内保持开屏状态），那么 mStayOn 为真。函数 updateStayOnLocked 的具体实现代码如下所示：

```
private void updateStayOnLocked(int dirty) {
    if ((dirty & (DIRTY_BATTERY_STATE | DIRTY_SETTINGS)) != 0) {
        final boolean wasStayOn = mStayOn;
        if (mStayOnWhilePluggedInSetting != 0
                && !isMaximumScreenOffTimeoutFromDeviceAdminEnforcedLocked()) {
            mStayOn = mBatteryService.isPowered(mStayOnWhilePluggedInSetting);
        } else {
            mStayOn = false;
        }

        if (mStayOn != wasStayOn) {
            mDirty |= DIRTY_STAY_ON;
        }
    }
}
```

由此可见，在第一阶段的更新过程中主要是进行了充电状态的判断工作，然后根据充电的状态，更新了一些必要属性的变化，同时更新了 mDirty。

- 第 2 阶段：显示内容的更新。

mWakefuless 表示 device 处于醒着、睡眠，或两者之间的一种状态，这种状态会影响到 wake lock 和 user activity 的计算，所以要进行更新。在第 2 阶段先通过一个死循环进行处理，只有当 updateWakefulnessLocked 返回为 false 时才能跳出这个循环。在刚进入这个循环时，对 mDirty 进行了重置，这能够说明在这次 updatePowerState 后会执行前面所有发生的 power state，而不会让其影响到下一次的变化。同时也在为下一次的 power state 从头开始更新做好准备。其中函数

updateWakeLockSummaryLocked 的实现代码如下所示：

```
private void updateWakeLockSummaryLocked(int dirty) {
    if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_WAKEFULNESS)) != 0) {
        mWakeLockSummary = 0;

        final int numWakeLocks = mWakeLocks.size();
        for (int i = 0; i < numWakeLocks; i++) {
            final WakeLock wakeLock = mWakeLocks.get(i);
            switch (wakeLock.mFlags & PowerManager.WAKE_LOCK_LEVEL_MASK) {
                case PowerManager.PARTIAL_WAKE_LOCK:
                    mWakeLockSummary |= WAKE_LOCK_CPU;
                    break;
                case PowerManager.FULL_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU
                                | WAKE_LOCK_SCREEN_BRIGHT | WAKE_LOCK_BUTTON_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_BRIGHT;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.SCREEN_DIM_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_SCREEN_DIM;
                        if (mWakefulness == WAKEFULNESS_AWAKE) {
                            mWakeLockSummary |= WAKE_LOCK_STAY_AWAKE;
                        }
                    }
                    break;
                case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:
                    if (mWakefulness != WAKEFULNESS_ASLEEP) {
                        mWakeLockSummary |= WAKE_LOCK_CPU | WAKE_LOCK_PROXIMITY_SCREEN_OFF;
                    }
                    break;
            }
        }

        if (DEBUG_SPEW) {
            Slog.d(TAG, "updateWakeLockSummaryLocked: mWakefulness="
                    + wakefulnessToString(mWakefulness)
                    + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary));
        }
    }
}
```

函数 updateUserActivitySummaryLocked 对锁屏时间和变暗时间进行比较。假设在系统中设置的睡眠时间是 30s，而在 PowerManagerService 中默认的 SCREEN_DIM_DURATION 是 7s，则说明，如果没有用户活动，则设备的屏幕在第 23s 开始变换，持续 7s 时间，然后才开始关闭屏幕。函数 updateUserActivitySummaryLocked 的具体实现代码如下所示：

```
private void updateUserActivitySummaryLocked(long now, int dirty) {
    // Update the status of the user activity timeout timer.
    if ((dirty & (DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS | DIRTY_SETTINGS)) != 0) {
        mHandler.removeMessages(MSG_USER_ACTIVITY_TIMEOUT);
        long nextTimeout = 0;
        if (mWakefulness != WAKEFULNESS_ASLEEP) {
            final int screenOffTimeout = getScreenOffTimeoutLocked();
            final int screenDimDuration = getScreenDimDurationLocked(screenOffTimeout);
            mUserActivitySummary = 0;
            if (mLastUserActivityTime >= mLastWakeTime) {
                nextTimeout = mLastUserActivityTime
```

```
                    + screenOffTimeout - screenDimDuration;
            if (now < nextTimeout) {
                mUserActivitySummary |= USER_ACTIVITY_SCREEN_BRIGHT;
            } else {
                nextTimeout = mLastUserActivityTime + screenOffTimeout;
                if (now < nextTimeout) {
                    mUserActivitySummary |= USER_ACTIVITY_SCREEN_DIM;
                }
            }
        }
        if (mUserActivitySummary == 0
                && mLastUserActivityTimeNoChangeLights >= mLastWakeTime) {
            nextTimeout = mLastUserActivityTimeNoChangeLights + screenOffTimeout;
            if (now < nextTimeout
                    && mDisplayPowerRequest.screenState
                            != DisplayPowerRequest.SCREEN_STATE_OFF) {
                mUserActivitySummary = mDisplayPowerRequest.screenState
                        == DisplayPowerRequest.SCREEN_STATE_BRIGHT ?
                        USER_ACTIVITY_SCREEN_BRIGHT : USER_ACTIVITY_SCREEN_DIM;
            }
        }
        if (mUserActivitySummary != 0) {
            Message msg = mHandler.obtainMessage(MSG_USER_ACTIVITY_TIMEOUT);
            msg.setAsynchronous(true);
            mHandler.sendMessageAtTime(msg, nextTimeout);
        }
    } else {
        mUserActivitySummary = 0;
    }
    if (DEBUG_SPEW) {
        Slog.d(TAG, "updateUserActivitySummaryLocked: mWakefulness="
                + wakefulnessToString(mWakefulness)
                + ", mUserActivitySummary=0x" + Integer.toHexString(mUserActivitySummary)
                + ", nextTimeout=" + TimeUtils.formatUptime(nextTimeout));
    }
}
}
```

具体在什么时候才可以跳出这个循环，需要视函数 updateWakefulnessLocked 的返回值而定。函数 updateWakefulnessLocked 的具体实现代码如下所示。

```
/**
     * 这个方法的功能是：根据当前的 wakeLocks 和用户的活动情况，来决定设备是否需要休眠
     *
        *///当 wakefulness 发生变化的时，返回 true，同时也需要重新计算 power state
    private boolean updateWakefulnessLocked(int dirty) {
        boolean changed = false;
        if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_BOOT_COMPLETED
                | DIRTY_WAKEFULNESS | DIRTY_STAY_ON | DIRTY_PROXIMITY_POSITIVE
                | DIRTY_DOCK_STATE)) != 0) {
            if (mWakefulness == WAKEFULNESS_AWAKE && isItBedTimeYetLocked()) {
                if (DEBUG_SPEW) {
                    Slog.d(TAG, "updateWakefulnessLocked: Bed time...");
                }
                final long time = SystemClock.uptimeMillis();
                if (shouldNapAtBedTimeLocked()) {
                    changed = napNoUpdateLocked(time);
                } else {
                    changed = goToSleepNoUpdateLocked(time,
                            PowerManager.GO_TO_SLEEP_REASON_TIMEOUT);
                }
            }
        }
        return changed;
    }
```

在上述代码中用到了函数 isItBedTimeYetLocked，功能是询问是否到了应该休眠的时间了，如果现在设备处于醒着的状态，立即改变为睡眠时间。函数 isItBedTimeYetLocked 的具体实现代码如下所示：

```
private boolean isItBedTimeYetLocked() {
    return mBootCompleted && !isBeingKeptAwakeLocked();
}
```

在上述代码中，如果有应用程序持有 wakelock 或产生了用户活动，或者处于充电状态，那么 isBeingKeptAwakeLocked 的返回值就是 true，相应地 isItBedTimeYetLocked 返回值为 false，因为还有没释放的 wakelock，或者有用户活动或者是在充电等，这说明还没有到睡眠的时间。但是，如果 wakelock 都释放了，并且也没有了用户活动了，那么就可以进入睡眠状态。这时的设备由醒着状态转换为睡眠的处理过程，此过程需要调用函数 updateWakefulnessLocked 实现，此函数的具体实现代码如下所示：

```
private boolean shouldNapAtBedTimeLocked() {
    return mDreamsActivateOnSleepSetting
           || (mDreamsActivateOnDockSetting
                 && mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED);
}
```

在上述代码中，mDreamsActivateOnSleepSetting 的默认值为 false，mDreamsActivateOnDockSetting 的默认值为 true。因为一般的用户没有接入 Dock，所以，“mDockState != Intent.EXTRA_DOCK_STATE_UNDOCKED”的值为 false，上述函数的返回值是 false。这样接下来需要执行函数 goToSleepNoUpdateLocked，此函数已经在前面进行了讲解，此函数只是更新了 power state 中一些必要的属性，并没有真正地执行能够让 Device 进入 sleep 的代码，真正的执行代码在函数 updatePowerStateLocked 中。

- 第 3 阶段：dream 和 display 状态的更新。

在这一阶段，函数 updateDreamLocked 会根据 mDirty 的变化，并结合其他的属性共同判断是否要开始屏保（Dreaming）处理。如果需要开始进行屏保的话，需要通过 DreamManagerService 开启 Dreaming。函数 updateDreamLocked 的具体实现代码如下所示：

```
private void updateDreamLocked(int dirty) {
    if ((dirty & (DIRTY_WAKEFULNESS
            | DIRTY_USER_ACTIVITY
            | DIRTY_WAKE_LOCKS
            | DIRTY_BOOT_COMPLETED
            | DIRTY_SETTINGS
            | DIRTY_IS_POWERED
            | DIRTY_STAY_ON
            | DIRTY_PROXIMITY_POSITIVE
            | DIRTY_BATTERY_STATE)) != 0) {
        scheduleSandmanLocked();
    }
}
```

函数 updateDisplayPowerStateLocked 主要功能是每次重新计算 Display Power State 的值，即 SCREEN_STATE_OFF、SCREEN_STATE_DIM 和 SCREEN_STATE_BRIGHT 这 3 个值之一。如果在 DisplayController 中更新了 Display Power State 的值，那么 DisplayController 会发送通知消息，所以还需要返回来重新检查一次。函数 updateDisplayPowerStateLocked 的具体实现代码如下所示：

```
private void updateDisplayPowerStateLocked(int dirty) {
    if ((dirty & (DIRTY_WAKE_LOCKS | DIRTY_USER_ACTIVITY | DIRTY_WAKEFULNESS
            | DIRTY_ACTUAL_DISPLAY_POWER_STATE_UPDATED | DIRTY_BOOT_COMPLETED
            | DIRTY_SETTINGS | DIRTY_SCREEN_ON_BLOCKER_RELEASED)) != 0) {
        int newScreenState = getDesiredScreenPowerStateLocked();
        //获取 display 的 power state 将要变成的状态
        if (newScreenState != mDisplayPowerRequest.screenState) {
        //mDisplayPower Request.screenState 是目前 display 所处的 power state
            if (newScreenState == DisplayPowerRequest.SCREEN_STATE_OFF
                    && mDisplayPowerRequest.screenState
                            != DisplayPowerRequest.SCREEN_STATE_OFF) {
                            //这个判断意味着：目前 display 的电源状态不是 OFF，但是想要变为 OFF
                mLastScreenOffEventElapsedRealTime = SystemClock.elapsedRealtime();
            }
            mDisplayPowerRequest.screenState = newScreenState;
            nativeSetPowerState(
```

```
                    newScreenState != DisplayPowerRequest.SCREEN_STATE_OFF,
                    newScreenState == DisplayPowerRequest.SCREEN_STATE_BRIGHT);
            }
            int screenBrightness = mScreenBrightnessSettingDefault;
            float screenAutoBrightnessAdjustment = 0.0f;
            boolean autoBrightness = (mScreenBrightnessModeSetting ==
                    Settings.System.SCREEN_BRIGHTNESS_MODE_AUTOMATIC);
                   //获取屏幕亮度模式是否为自动变化
            if          (isValidBrightness(mScreenBrightnessOverrideFromWindowManager))
{//mScreenBrightnessOverideFromeWindowManager 是 WindowManager 设置的亮度大小，默认值为-1
                screenBrightness = mScreenBrightnessOverrideFromWindowManager;
                autoBrightness = false;
            } else if (isValidBrightness(mTemporaryScreenBrightnessSettingOverride))
{//mTemporaryScreenBrightnessSettingOverride 在 widget 中设置的临时亮度大小，默认为-1
                screenBrightness = mTemporaryScreenBrightnessSettingOverride;
            } else if (isValidBrightness(mScreenBrightnessSetting)) {
            //在 Settings 中设置的默认亮度，在 android4.2 中其值为 102
                screenBrightness = mScreenBrightnessSetting;
            }
            if (autoBrightness) {//如果亮度是自动调节的话
                screenBrightness = mScreenBrightnessSettingDefault;
                if (isValidAutoBrightnessAdjustment(
                        mTemporaryScreenAutoBrightnessAdjustmentSettingOverride)) {
                    screenAutoBrightnessAdjustment =
                            mTemporaryScreenAutoBrightnessAdjustmentSettingOverride;
                } else if (isValidAutoBrightnessAdjustment(
                        mScreenAutoBrightnessAdjustmentSetting)) {
                    screenAutoBrightnessAdjustment = mScreenAutoBrightnessAdjustmentSetting;
                }
            }
            screenBrightness = Math.max(Math.min(screenBrightness,
                    mScreenBrightnessSettingMaximum), mScreenBrightnessSettingMinimum);
            screenAutoBrightnessAdjustment = Math.max(Math.min(
                    screenAutoBrightnessAdjustment, 1.0f), -1.0f);
            mDisplayPowerRequest.screenBrightness = screenBrightness;//从这行向下开始就是
            //配置完成 DisplayPowerRequest，然后以此为参数通过 requestPowerSTate 方法进行设置
            mDisplayPowerRequest.screenAutoBrightnessAdjustment =
                    screenAutoBrightnessAdjustment;
            mDisplayPowerRequest.useAutoBrightness = autoBrightness;
            mDisplayPowerRequest.useProximitySensor = shouldUseProximitySensorLocked();
            mDisplayPowerRequest.blockScreenOn = mScreenOnBlocker.isHeld();
            mDisplayReady = mDisplayPowerController.requestPowerState(mDisplayPowerRequest,
                    mRequestWaitForNegativeProximity);
            mRequestWaitForNegativeProximity = false;
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateScreenStateLocked: mDisplayReady=" + mDisplayReady
                        + ", newScreenState=" + newScreenState
                        + ", mWakefulness=" + mWakefulness
                        + ", mWakeLockSummary=0x" + Integer.toHexString(mWakeLockSummary)
                        + ", mUserActivitySummary=0x" + Integer.toHexString(mUserActivitySummary)
                        + ", mBootCompleted=" + mBootCompleted);
            }
        }
    }
```

在上述代码中实现了对屏幕亮度的请求，并改变了屏幕的亮度。在调用的函数 goToSleepNoUpdateLocked 中，Display State 会因为 requestPowerState 函数的调用而起作用。

- 第 4 阶段：suspend blocker 的更新。

之所以放在最后一步才进行 Suspend Blocker（暂停阻滞）的更新，是因为，在这里可能会释放 Suspend Blocker。本阶段 Suspend Blocker 的更新很简单，仅需要判断 Device 是否需要持有 CPU 或者是否需要 CPU 继续运行，如果有没有释放的 WakeLock，或者还有用户活动的话，或者屏幕没有关闭等，则肯定需要持有 CPU。所以，这里就是根据需求去申请或者释放 SuspendBlocker。

到此为止，整个 PowerManagerService 的工作过程介绍完毕，已经基本上讲解了 PowerManagerService 在 Framework 中的实现过程。

（4）更新状态

如果要在屏幕中点击或者滑动一次就会调用一次 userActivity 函数，该函数会更新一些状态，

其中比较重要的是 mUserState 值的状态，此值决定了系统对 LCD、按键以及键盘的亮和灭，例如，“mUserState = 0X7”表示 LCD 和按键背光亮。函数 userActivity 的具体实现代码如下所示：

```
public void userActivity(long eventTime, int event, int flags) {
    final long now = SystemClock.uptimeMillis();
    if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.DEVICE_POWER)
            != PackageManager.PERMISSION_GRANTED) {
        // Once upon a time applications could call userActivity().
        // Now we require the DEVICE_POWER permission.  Log a warning and ignore the
        // request instead of throwing a SecurityException so we don't break old apps.
        synchronized (mLock) {
            if (now >= mLastWarningAboutUserActivityPermission + (5 * 60 * 1000)) {
                mLastWarningAboutUserActivityPermission = now;
                Slog.w(TAG, "Ignoring call to PowerManager.userActivity() because the "
                        + "caller does not have DEVICE_POWER permission.  "
                        + "Please fix your app!  "
                        + " pid=" + Binder.getCallingPid()
                        + " uid=" + Binder.getCallingUid());
            }
        }
        return;
    }

    if (eventTime > SystemClock.uptimeMillis()) {
        throw new IllegalArgumentException("event time must not be in the future");
    }

    final int uid = Binder.getCallingUid();
    final long ident = Binder.clearCallingIdentity();
    try {
        userActivityInternal(eventTime, event, flags, uid);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

（5）获得唤醒锁

函数 acquireWakeLock 的功能是获得唤醒锁，文件 PowerManagerService.java 的最基本功能是管理所有的应用程序申请的 wakelock。函数 acquireWakeLock 的具体实现代码如下所示：

```
public void acquireWakeLock(IBinder lock, int flags, String tag, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    PowerManager.validateWakeLockParameters(flags, tag);

    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK, null);
    if (ws != null && ws.size() != 0) {
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final int uid = Binder.getCallingUid();
    final int pid = Binder.getCallingPid();
    final long ident = Binder.clearCallingIdentity();
    try {
        acquireWakeLockInternal(lock, flags, tag, ws, uid, pid);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

在 Android 系统中，音频播放器、视频播放器、Camera 等申请的 wakelock 都是通过这个类来管理的。请读者看下面的代码：

```
static final String PARTIAL_NAME = "PowerManagerService"
Power.acquireWakeLock(Power.PARTIAL_WAKE_LOCK,
        PARTIAL_NAME);
```

在上述代码中调用了类 Power 中的函数 acquireWakeLock()，此时的 PARTIAL_NAME 作为参数传递到底层去。

（6）获得唤醒的内部锁

函数 acquireWakeLockInternal 的功能是获得唤醒的内部锁，具体实现代码如下所示：

```
private void acquireWakeLockInternal(IBinder lock, int flags, String tag, WorkSource ws,
        int uid, int pid) {
    synchronized (mLock) {
        if (DEBUG_SPEW) {
            Slog.d(TAG, "acquireWakeLockInternal: lock=" + Objects.hashCode(lock)
                    + ", flags=0x" + Integer.toHexString(flags)
                    + ", tag=\"" + tag + "\", ws=" + ws + ", uid=" + uid + ", pid=" + pid);
        }

        WakeLock wakeLock;
        int index = findWakeLockIndexLocked(lock);
        if (index >= 0) {
            wakeLock = mWakeLocks.get(index);
            if (!wakeLock.hasSameProperties(flags, tag, ws, uid, pid)) {
                // Update existing wake lock.  This shouldn't happen but is harmless.
                notifyWakeLockReleasedLocked(wakeLock);
                wakeLock.updateProperties(flags, tag, ws, uid, pid);
                notifyWakeLockAcquiredLocked(wakeLock);
            }
        } else {
            wakeLock = new WakeLock(lock, flags, tag, ws, uid, pid);
            try {
                lock.linkToDeath(wakeLock, 0);
            } catch (RemoteException ex) {
                throw new IllegalArgumentException("Wake lock is already dead.");
            }
            notifyWakeLockAcquiredLocked(wakeLock);
            mWakeLocks.add(wakeLock);
        }

        applyWakeLockFlagsOnAcquireLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;
        updatePowerStateLocked();
    }
}
```

（7）获取锁中标志的唤醒锁

函数 applyWakeLockFlagsOnAcquireLocked 的功能是在获取的锁中标志某个唤醒锁，具体实现代码如下所示：

```
private void applyWakeLockFlagsOnAcquireLocked(WakeLock wakeLock) {
    if ((wakeLock.mFlags & PowerManager.ACQUIRE_CAUSES_WAKEUP) != 0 &&
            isScreenLock(wakeLock)) {
        wakeUpNoUpdateLocked(SystemClock.uptimeMillis());
    }
}
```

（8）释放唤醒锁

函数 releaseWakeLock 的功能是释放唤醒锁，具体实现代码如下所示：

```
public void releaseWakeLock(IBinder lock, int flags) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK,
null);
    final long ident = Binder.clearCallingIdentity();
    try {
        releaseWakeLockInternal(lock, flags);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}
```

（9）释放唤醒内部锁

函数 releaseWakeLockInternal 的功能是释放唤醒内部锁，具体实现代码如下所示：

```
private void releaseWakeLockInternal(IBinder lock, int flags) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                        + " [not found], flags=0x" + Integer.toHexString(flags));
            }
            return;
        }

        WakeLock wakeLock = mWakeLocks.get(index);
        if (DEBUG_SPEW) {
            Slog.d(TAG, "releaseWakeLockInternal: lock=" + Objects.hashCode(lock)
                    + " [" + wakeLock.mTag + "], flags=0x" + Integer.toHexString(flags));
        }

        mWakeLocks.remove(index);
        notifyWakeLockReleasedLocked(wakeLock);
        wakeLock.mLock.unlinkToDeath(wakeLock, 0);

        if ((flags & PowerManager.WAIT_FOR_PROXIMITY_NEGATIVE) != 0) {
            mRequestWaitForNegativeProximity = true;
        }

        applyWakeLockFlagsOnReleaseLocked(wakeLock);
        mDirty |= DIRTY_WAKE_LOCKS;
        updatePowerStateLocked();
    }
}
```

（10）更新唤醒锁资源和内部唤醒锁的资源

函数 updateWakeLockWorkSource 和 updateWakeLockWorkSourceInternal 被绑定机制 Binder 所调用，功能是分别更新唤醒锁资源和内部唤醒锁的资源，这两个函数的具体实现代码如下所示：

```
public void updateWakeLockWorkSource(IBinder lock, WorkSource ws) {
    if (lock == null) {
        throw new IllegalArgumentException("lock must not be null");
    }

    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.WAKE_LOCK, null);
    if (ws != null && ws.size() != 0) {
        mContext.enforceCallingOrSelfPermission(
                android.Manifest.permission.UPDATE_DEVICE_STATS, null);
    } else {
        ws = null;
    }

    final long ident = Binder.clearCallingIdentity();
    try {
        updateWakeLockWorkSourceInternal(lock, ws);
    } finally {
        Binder.restoreCallingIdentity(ident);
    }
}

private void updateWakeLockWorkSourceInternal(IBinder lock, WorkSource ws) {
    synchronized (mLock) {
        int index = findWakeLockIndexLocked(lock);
        if (index < 0) {
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakeLockWorkSourceInternal: lock=" + Objects.hashCode
                (lock) + " [not found], ws=" + ws);
            }
            throw new IllegalArgumentException("Wake lock not active");
        }
```

```
            WakeLock wakeLock = mWakeLocks.get(index);
            if (DEBUG_SPEW) {
                Slog.d(TAG, "updateWakeLockWorkSourceInternal: lock=" + Objects.hashCode(lock)
                        + " [" + wakeLock.mTag + "], ws=" + ws);
            }

            if (!wakeLock.hasSameWorkSource(ws)) {
                notifyWakeLockReleasedLocked(wakeLock);
                wakeLock.updateWorkSource(ws);
                notifyWakeLockAcquiredLocked(wakeLock);
            }
        }
    }
```

14.3 JNI 层架构分析

在 Android 的电源管理系统中，在 Power 类中并没有实现 Native 申明的方法，而是在文件 frameworks/base/core/jni/android_os_Power.cpp 中实现的。也就是说，文件 android_os_Power.cpp 就是 Power Management 系统的 JNI 层。在本节的内容中，将详细分析 Android 电源管理系统中的 JNI 层的基本架构知识。

14.3.1　定义了两层之间的接口函数

文件 android_os_Power.cpp 比较简单，定义了连接应用层和内核层之间的桥梁作用的接口函数，这些函数已经在本章前面的 Framework 层部分的内容中调用过。文件 android_os_Power.cpp 的主要实现代码如下所示：

```
static void
acquireWakeLock(JNIEnv *env, jobject clazz, jint lock, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
    acquire_wake_lock(lock, id);
    env->ReleaseStringUTFChars(idObj, id);
}
static void
releaseWakeLock(JNIEnv *env, jobject clazz, jstring idObj)
{
    if (idObj == NULL) {
        jniThrowNullPointerException(env, "id is null");
        return ;
    }
    const char *id = env->GetStringUTFChars(idObj, NULL);
    release_wake_lock(id);
    env->ReleaseStringUTFChars(idObj, id);
}
static int
setLastUserActivityTimeout(JNIEnv *env, jobject clazz, jlong timeMS)
{
    return set_last_user_activity_timeout(timeMS/1000);
}
static int
setScreenState(JNIEnv *env, jobject clazz, jboolean on)
{
    return set_screen_state(on);
}
static void android_os_Power_shutdown(JNIEnv *env, jobject clazz)
{
    android_reboot(ANDROID_RB_POWEROFF, 0, 0);
}
static void android_os_Power_reboot(JNIEnv *env, jobject clazz, jstring reason)
{
    if (reason == NULL) {
```

```
        android_reboot(ANDROID_RB_RESTART, 0, 0);
    } else {
        const char *chars = env->GetStringUTFChars(reason, NULL);
        android_reboot(ANDROID_RB_RESTART2, 0, (char *) chars);
        env->ReleaseStringUTFChars(reason, chars);  // In case it fails.
    }
    jniThrowIOException(env, errno);
}
static JNINativeMethod method_table[] = {
    { "acquireWakeLock", "(ILjava/lang/String;)V", (void*)acquireWakeLock },
    { "releaseWakeLock", "(Ljava/lang/String;)V", (void*)releaseWakeLock },
    { "setLastUserActivityTimeout", "(J)I", (void*)setLastUserActivityTimeout },
    { "setScreenState", "(Z)I", (void*)setScreenState },
    { "shutdown", "()V", (void*)android_os_Power_shutdown },
    { "rebootNative", "(Ljava/lang/String;)V", (void*)android_os_Power_reboot },
};
int register_android_os_Power(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(
        env, "android/os/Power",
        method_table, NELEM(method_table));
}
};
```

14.3.2 与 Linux Kernel 层进行交互

与 Linux Kernel 的交互工作是通过文件 hardware\libhardware\power\power.c 实现的，Andriod 与 Kernel 的交互主要是通过 sys 文件的方式来实现。文件 power.c 的具体实现代码如下所示。

```
static void power_init(struct power_module *module)
{
}
static void power_set_interactive(struct power_module *module, int on)
{
}
static void power_hint(struct power_module *module, power_hint_t hint,
                       void *data) {
    switch (hint) {
    default:
        break;
    }
}
static struct hw_module_methods_t power_module_methods = {
    .open = NULL,
};
struct power_module HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .module_api_version = POWER_MODULE_API_VERSION_0_2,
        .hal_api_version = HARDWARE_HAL_API_VERSION,
        .id = POWER_HARDWARE_MODULE_ID,
        .name = "Default Power HAL",
        .author = "The Android Open Source Project",
        .methods = &power_module_methods,
    },
    .init = power_init,
    .setInteractive = power_set_interactive,
    .powerHint = power_hint,
};
```

14.4 Kernel（内核）层架构分析

在 Android 系统中，Power Management 系统内核层的实现文件是：

- drivers/android/power.c；
- drivers/base/main.c；
- drivers/base/power/suspend.c；

- drivers/base/power/resume.c；
- kernel/power/main.c；
- kernel/power/wakelock.c；
- kernel/power/unwakelock.c；
- kernel/power/earlysuspend.c。

在本节的内容中，将对上述内核层的具体实现进行详细讲解。为了节省本书的篇幅，只讲解主要的实现文件的核心代码。

14.4.1　文件 power.c

在 Power Management 系统的内核层中，实现文件 drivers/android/power.c 对 Kernel 提供了如下所示的接口函数。

（1）初始化 Suspend lock

接口函数 EXPORT_SYMBOL(android_init_suspend_lock)的功能是初始化 Suspend lock，在使用 Power Management 内核之前必须做初始化工作，函数 android_init_suspend_lock 的具体实现代码如下所示：

```
int android_init_suspend_lock(android_suspend_lock_t *lock)
{
    return android_init_suspend_lock_internal(lock, 0);
}
```

（2）释放 suspend lock 相关的资源

接口函数 EXPORT_SYMBOL(android_uninit_suspend_lock)的功能是释放 suspend lock 相关的资源，具体实现代码如下所示：

```
void android_uninit_suspend_lock(android_suspend_lock_t *lock)
{
    unsigned long irqflags;
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_uninit_suspend_lock name=%s\n",
            lock->name);
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(lock->stat.count) {
        if(g_deleted_wake_locks.stat.count == 0) {
            g_deleted_wake_locks.name = "deleted_wake_locks";
            android_init_suspend_lock_internal(
                &g_deleted_wake_locks, 1);
        }
        g_deleted_wake_locks.stat.count += lock->stat.count;
        g_deleted_wake_locks.stat.expire_count += lock->stat.expire_count;
        g_deleted_wake_locks.stat.total_time  =  ktime_add(g_deleted_wake_locks.stat.
total_time, lock->stat.total_time);
        g_deleted_wake_locks.stat.max_time = ktime_add(g_deleted_wake_locks.stat.max_
time, lock->stat.max_time);
    }
#endif
    list_del(&lock->link);
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（3）申请 lock，并调用相应的 unlock

接口函数 EXPORT_SYMBOL(android_lock_suspend)的功能是申请 lock，并调用相应的 unlock 来释放 lock。函数 android_lock_suspend 的具体实现代码如下所示：

```
void android_lock_suspend(android_suspend_lock_t *lock)
{
    unsigned long irqflags;
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
        lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
```

```
        lock->stat.last_time = ktime_get();
    }
#endif
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_power: acquire wake lock: %s\n",
            lock->name);
    lock->expires = INT_MAX;
    lock->flags &= ~ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
    list_del(&lock->link);
    list_add(&lock->link, &g_active_partial_wake_locks);
    g_current_event_num++;
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（4）申请 Partial Wakelock

接口函数 EXPORT_SYMBOL(android_lock_suspend_auto_expire)的功能是申请 Partial Wakelock，当定时时间到后会自动释放。函数 android_lock_suspend_auto_expire 的具体实现代码如下所示：

```
void android_lock_suspend_auto_expire(android_suspend_lock_t *lock, int timeout)
{
    unsigned long irqflags;
    spin_lock_irqsave(&g_list_lock, irqflags);
#ifdef CONFIG_ANDROID_POWER_STAT
    if(!(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE)) {
        lock->flags |= ANDROID_SUSPEND_LOCK_ACTIVE;
        lock->stat.last_time = ktime_get();
    }
#endif
    if (android_power_debug_mask & ANDROID_POWER_DEBUG_WAKE_LOCK)
        printk(KERN_INFO "android_power: acquire wake lock: %s, "
            "timeout %d.%03lu\n", lock->name, timeout / HZ,
            (timeout % HZ) * MSEC_PER_SEC / HZ);
    lock->expires = jiffies + timeout;
    lock->flags |= ANDROID_SUSPEND_LOCK_AUTO_EXPIRE;
    list_del(&lock->link);
    list_add(&lock->link, &g_active_partial_wake_locks);
    g_current_event_num++;
    wake_up(&g_wait_queue);
    spin_unlock_irqrestore(&g_list_lock, irqflags);
}
```

（5）释放 lock

接口函数 EXPORT_SYMBOL(android_unlock_suspend)的功能是释放 lock，具体实现代码如下所示：

```
static void android_unlock_suspend_stat_locked(android_suspend_lock_t *lock)
{
    if(lock->flags & ANDROID_SUSPEND_LOCK_ACTIVE) {
        ktime_t duration;
        lock->flags &= ~ANDROID_SUSPEND_LOCK_ACTIVE;
        lock->stat.count++;
        duration = ktime_sub(ktime_get(), lock->stat.last_time);
        lock->stat.total_time = ktime_add(lock->stat.total_time, duration);
        if(ktime_to_ns(duration) > ktime_to_ns(lock->stat.max_time))
            lock->stat.max_time = duration;
        lock->stat.last_time = ktime_get();
    }
}
#endif
```

（6）注册 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(android_register_early_suspend)的功能是注册 Early Suspend 的驱动，具体实现代码如下所示：

```
void android_register_early_suspend(android_early_suspend_t *handler)
{
    struct list_head *pos;

    mutex_lock(&g_early_suspend_lock);
```

```
    list_for_each(pos, &g_early_suspend_handlers) {
        android_early_suspend_t *e = list_entry(pos, android_early_suspend_t, link);
        if(e->level > handler->level)
            break;
    }
    list_add_tail(&handler->link, pos);
    mutex_unlock(&g_early_suspend_lock);
}
```

（7）取消已经注册的 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(android_unregister_early_suspend)的功能是取消已经注册的 Early Suspend 的驱动，具体实现代码如下所示：

```
void android_unregister_early_suspend(android_early_suspend_t *handler)
{
    mutex_lock(&g_early_suspend_lock);
    list_del(&handler->link);
    mutex_unlock(&g_early_suspend_lock);
}
```

14.4.2　文件 earlysuspend.c

在 Power Management 系统的内核层中，实现文件 kernel/power/earlysuspend.c 对 Kernel 提供了如下所示的接口函数。

（1）注册 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(register_early_suspend)的功能是注册 Early Suspend 的驱动，具体实现代码如下所示：

```
void register_early_suspend(struct early_suspend *handler)
{
        struct list_head *pos;

        mutex_lock(&early_suspend_lock);
        list_for_each(pos, &early_suspend_handlers) {
                struct early_suspend *e;
                e = list_entry(pos, struct early_suspend, link);
                if (e->level > handler->level)
                        break;
        }
        list_add_tail(&handler->link, pos);
        if ((state & SUSPENDED) && handler->suspend)
                handler->suspend(handler);
        mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(register_early_suspend);
```

（2）取消已经注册的 Early Suspend 驱动

接口函数 EXPORT_SYMBOL(unregister_early_suspend)的功能是取消已经注册的 Early Suspend 的驱动，具体实现代码如下所示：

```
void unregister_early_suspend(struct early_suspend *handler)
{
        mutex_lock(&early_suspend_lock);
        list_del(&handler->link);
        mutex_unlock(&early_suspend_lock);
}
EXPORT_SYMBOL(unregister_early_suspend);
```

14.4.3　文件 wakelock.c

在 Power Management 系统的内核层中，实现文件 kernel/power/wakelock.c 对 Kernel 提供了如下所示的接口函数。

（1）EXPORT_SYMBOL(wake_unlock)

此接口函数的功能是释放 lock，函数 wake_unlock 的具体实现代码如下所示：

```
void wake_unlock(struct wake_lock *lock)
```

```
{
        int type;
        unsigned long irqflags;
        spin_lock_irqsave(&list_lock, irqflags);
        type = lock->flags & WAKE_LOCK_TYPE_MASK;
#ifdef CONFIG_WAKELOCK_STAT
        wake_unlock_stat_locked(lock, 0);
#endif
        if (debug_mask & DEBUG_WAKE_LOCK)
                pr_info("wake_unlock: %s\n", lock->name);
        lock->flags &= ~(WAKE_LOCK_ACTIVE | WAKE_LOCK_AUTO_EXPIRE);
        list_del(&lock->link);
        list_add(&lock->link, &inactive_locks);
        if (type == WAKE_LOCK_SUSPEND) {
                long has_lock = has_wake_lock_locked(type);
                if (has_lock > 0) {
                        if (debug_mask & DEBUG_EXPIRE)
                                pr_info("wake_unlock: %s, start expire timer, "
                                        "%ld\n", lock->name, has_lock);
                        mod_timer(&expire_timer, jiffies + has_lock);
                } else {
                        if (del_timer(&expire_timer))
                                if (debug_mask & DEBUG_EXPIRE)
                                        pr_info("wake_unlock: %s, stop expire "
                                                "timer\n", lock->name);
                        if (has_lock == 0)
                                queue_work(suspend_work_queue, &suspend_work);
                }
                if (lock == &main_wake_lock) {
                        if (debug_mask & DEBUG_SUSPEND)
                                print_active_locks(WAKE_LOCK_SUSPEND);
#ifdef CONFIG_WAKELOCK_STAT
                        update_sleep_wait_stats_locked(0);
#endif
                }
        }
        spin_unlock_irqrestore(&list_lock, irqflags);
}
EXPORT_SYMBOL(wake_unlock);
```

（2）EXPORT_SYMBOL(wake_lock)

此接口函数的功能是申请 lock，必须调用相应的 unlock 来释放 lock。函数 wake_lock 的具体实现代码如下所示：

```
void wake_lock(struct wake_lock *lock)
{
        wake_lock_internal(lock, 0, 0);
}
EXPORT_SYMBOL(wake_lock);
```

（3）static DEFINE_TIMER(expire_timer, expire_wake_locks, 0, 0)

此接口函数的功能是如果定时时间到，则加入到 suspend 队列中。函数 expire_wake_locks 的具体实现代码如下所示：

```
static void expire_wake_locks(unsigned long data)
{
        long has_lock;
        unsigned long irqflags;
        if (debug_mask & DEBUG_EXPIRE)
                pr_info("expire_wake_locks: start\n");
        spin_lock_irqsave(&list_lock, irqflags);
        if (debug_mask & DEBUG_SUSPEND)
                print_active_locks(WAKE_LOCK_SUSPEND);
        has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
        if (debug_mask & DEBUG_EXPIRE)
                pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
        if (has_lock == 0)
                queue_work(suspend_work_queue, &suspend_work);
        spin_unlock_irqrestore(&list_lock, irqflags);
}
```

14.4.4　文件 resume.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/resume.c 对 Kernel 提供了如下所示的接口函数。

（1）EXPORT_SYMBOL_GPL(device_power_up)

此接口函数的功能是打开特殊的设备，函数 device_power_up 的具体实现代码如下所示：

```
void device_power_up(void)
{
    sysdev_resume();
    dpm_power_up();
}
EXPORT_SYMBOL_GPL(device_power_up);
```

（2）EXPORT_SYMBOL_GPL(device_resume)

此接口函数的功能是重新存储设备的状态，函数 device_resume 的具体实现代码如下所示：

```
void device_resume(void)
{
    down(&dpm_sem);
    dpm_resume();
    up(&dpm_sem);
}
EXPORT_SYMBOL_GPL(device_resume);
```

14.4.5　文件 suspend.c

在 Power Management 系统的内核层中，实现文件 drivers/base/power/suspend.c 对 Kernel 提供了如下所示的接口函数。

（1）EXPORT_SYMBOL_GPL(device_suspend)

此接口函数的功能是保存系统状态，并结束系统中设备的运行。函数 device_suspend 的具体实现代码如下所示：

```
int device_suspend(pm_message_t state)
{
    int error = 0;
    down(&dpm_sem);
    down(&dpm_list_sem);
    while (!list_empty(&dpm_active) && error == 0) {
        struct list_head * entry = dpm_active.prev;
        struct device * dev = to_device(entry);
        get_device(dev);
        up(&dpm_list_sem);
        error = suspend_device(dev, state);
        down(&dpm_list_sem);
        /* Check if the device got removed */
        if (!list_empty(&dev->power.entry)) {
            /* Move it to the dpm_off or dpm_off_irq list */
            if (!error) {
                list_del(&dev->power.entry);
                list_add(&dev->power.entry, &dpm_off);
            } else if (error == -EAGAIN) {
                list_del(&dev->power.entry);
                list_add(&dev->power.entry, &dpm_off_irq);
                error = 0;
            }
        }
        if (error)
            printk(KERN_ERR "Could not suspend device %s: "
                "error %d\n", kobject_name(&dev->kobj), error);
        put_device(dev);
    }
    up(&dpm_list_sem);
    if (error)
        dpm_resume();
    up(&dpm_sem);
    return error;
```

```
}
EXPORT_SYMBOL_GPL(device_suspend);
```

（2）EXPORT_SYMBOL_GPL(device_power_down)

此接口函数的功能是关闭特殊设备，函数 device_power_down 的具体实现代码如下所示：

```
int device_power_down(pm_message_t state)
{
    int error = 0;
    struct device * dev;
    list_for_each_entry_reverse(dev, &dpm_off_irq, power.entry) {
        if ((error = suspend_device(dev, state)))
            break;
    }
    if (error)
        goto Error;
    if ((error = sysdev_suspend(state)))
        goto Error;
 Done:
    return error;
 Error:
    dpm_power_up();
    goto Done;
}
EXPORT_SYMBOL_GPL(device_power_down);
```

14.4.6 文件 main.c

在 Power Management 系统的内核层中，内核文件 kernel/power/main.c 的主要实现代码如下所示：

```
static int __init pm_init(void)
{
    int error = pm_start_workqueue();
    if (error)
        return error;
    hibernate_image_size_init();
    hibernate_reserved_size_init();
    power_kobj = kobject_create_and_add("power", NULL);
    if (!power_kobj)
        return -ENOMEM;
    return sysfs_create_group(power_kobj, &attr_group);
}
core_initcall(pm_init);
```

在上述代码中，函数 pm_init(void)的返回值为 sysfs_create_group(power_kobj, &attr_group，表示当我们对“sysfs/”目录下相对的节点进行操作时，会调用与 attr_group 中相关函数。

14.4.7 proc 文件

在 Power Management 系统的内核层中，给 Framework 层提供了如下所示的 proc 文件。

- "/sys/android_power/acquire_partial_wake_lock"：申请 partial wake lock。
- "/sys/android_power/acquire_full_wake_lock"：申请 full wakelock。
- "/sys/android_power/release_wake_lock"：释放相应的 wake lock。
- "/sys/android_power/request_state"：请求改变系统状态，进入 standby 和回到 wakeup 状态。
- "/sys/android_power/state"：指示当前系统的状态。

Android 电源管理系统的主要功能是通过 Wake lock 来实现的，在最底层主要是通过如下 3 个队列来实现其管理功能。

- static LIST_HEAD(g_inactive_locks)。
- static LIST_HEAD(g_active_partial_wake_locks)。
- static LIST_HEAD(g_active_full_wake_locks)。

在处理过程中实现如下所示的插入和移动操作。

- 所有被初始化的 lock 都会被插入到队列 g_inactive_locks 中。

- 当前活动的 Partial Wake Lock 被插入到 g_active_partial_wake_locks 队列中。
- 活动的 Full Wake Lock 被插入到 g_active_full_wake_locks 队列中。
- 所有的 Partial Wake Lock 和 Full Wake Lock，在过期后或 unlock 后都会被移到 inactive 的队列，以等待下次被调用。

14.5 wakelock 和 early_suspend

在 Android 系统中，wakelock 和 early_suspend 是一种特殊机制，能够实现系统的“唤醒”和“休眠”功能，获取系统资源的信息，例如，电源信息和 CPU 信息等。在本节的内容中，将详细讲解 wakelock 和 early_suspend 机制的基本知识。

14.5.1 wakelock 的原理

wakelock 在 Android 的电源管理系统中扮演一个核心的角色。wakelock 是一种“锁”机制，只要有人拿着这个锁，系统就无法进入休眠状态。这个锁可以是有超时的或者是没有超时的，超时的锁会在时间过去以后自动解锁。如果没有锁了或者超时了，内核就会启动休眠机制来进入休眠。

当系统在启动完毕后，会自己去加一把名为“main”的锁，而当系统有意愿去睡眠时则会先去释放这把“main”锁。在 Android 中，在 early_suspend 的最后一步会去释放“main”锁（wake_unlock: main）。释放完后则会去检查是否还有其他存在的锁，如果没有则直接进入睡眠过程。

wakelock 的缺点是，如果有某一应用获锁而不释放，或者因一直在执行某种操作而没时间来释放的话，则会导致系统一直进入不了睡眠状态而造成功耗过大的问题。

在 wakelock 中有 3 种类型，最常用的是 WAKE_LOCK_SUSPEND，作用是防止系统进入睡眠。wakelock 的接口定义在文件 wakelock.c 中，定义代码如下所示。

```
enum {
    WAKE_LOCK_SUSPEND, /* Prevent suspend */
    WAKE_LOCK_IDLE,    /* Prevent low power idle */
    WAKE_LOCK_TYPE_COUNT
};
```

在 wakelock 中，有如下两个地方可以让系统从 early_suspend 进入 suspend 状态。

- 在 wake_unlock 中，当解锁之，没有其他的 wakelock，则进入 suspend。
- 当超时锁的定时器超时后，定时器的回调函数会判断有没有其他的 wakelock，若没有则进入 suspend。

在 Android 系统中，在 Kernel 层使用 wake lock 的基本步骤如下所示。

（1）调用函数 android_init_suspend_lock 初始化一个 wake lock。

（2）调用相关申请 lock 的函数 android_lock_suspend 或 android_lock_suspend_auto_expire 请求 lock，此处只能申请 Partial Wake Lock，如果要申请 Full Wake Lock，则需要调用函数 android_lock_partial_suspend_auto_expire（该函数没有 EXPORT 出来）。

（3）如果是 auto expire 的 wake lock 则可忽略，否则必须及时把相关的 wake lock 释放掉，否则会造成系统长期运行在高功耗的状态。

（4）在驱动卸载或不再使用 Wake lock 时，需要及时调用 android_uninit_suspend_lock 以释放资源。

由此可以总结出，Kernel 的 wake lock 唤醒操作的基本顺序依次是。

- 框架层函数的 acquireWakeLock()，此函数的具体实现代码如下所示：

```
int acquire_wake_lock(int lock, const char* id)
{
initialize_fds();
// LOGI("acquire_wake_lock lock=%d id='%s'\n", lock, id);
if (g_error) return g_error;
int fd;
if (lock == PARTIAL_WAKE_LOCK) {
```

```
fd = g_fds[ACQUIRE_PARTIAL_WAKE_LOCK];
}
else {
return EINVAL;
}
return write(fd, id, strlen(id));
}
```

- android_os_Power.cpp 的 acquireWakeLock()。
- power.c 的 acquire_wake_lock()。

14.5.2 early_suspend 的原理

early_suspend 在 Linux 内核的睡眠过程前被调用。因为背光需要的能耗过大，所以，常采用此类方法在手机系统的设计中操作背光。如一些在内核中预先进行处理的事件可以先注册上 early_suspend 函数，这样当系统要进入睡眠之前会首先调用这些注册的函数。

和 Android 休眠唤醒相关的实现文件如下所示：

```
linux_source/kernel/power/main.c
linux_source/kernel/power/earlysuspend.c
linux_source/kernel/power/wakelock.c
linux_source/kernel/power/process.c
linux_source/driver/base/power/main.c
linux_source/arch/xxx/mach-xxx/pm.c 或 linux_source/arch/xxx/plat-xxx/pm.c
```

14.5.3 Android 休眠

当用户读写“/sys/power/state”时，文件“linux_source/kernel/power/main.c”中的 state_store()函数会被调用。其中，Android 的 early_suspend 会执行：

```
request_suspend_state(state);
```

标准的 Linux 休眠会执行：

```
error = enter_state(state);
```

函数 state_store()的原型如下所示：

```
static ssize_t state_store(struct kobject *kobj, struct kobj_attribute *attr,
            const char *buf, size_t n)
```

在函数 request_suspend_state()中，会调用 early_suspend_work 的工作队列以进入 early_suspend()函数中。函数 request_suspend_state()的原型如下所示：

```
void request_suspend_state(suspend_state_t new_state)
```

在函数 early_suspend()中，首先要判断当前请求的状态是否还是 suspend，如果不是则直接退出；如果是，则函数会调用已经注册的 early_suspend 的函数。然后同步文件系统，最后释放 main_wake_lock。函数 early_suspend()的原型如下所示：

```
static void early_suspend(struct work_struct *work)
```

在函数 wake_unlock()中删除链表中的 wake_lock 节点，目的是判断当前是否存在 wake_lock。如果 wake_lock 的数目为 0，则调用工作队列 suspend_work，然后进入 suspend 状态。函数 wake_unlock()的原型如下所示：

```
void wake_unlock(struct wake_lock *lock)
```

在函数 suspend()中，首先判断当前是否有 wake_lock，如果有则退出；然后同步文件系统，最后调用 pm_suspend()函数。函数 suspend()的原型如下所示：

```
static void suspend(struct work_struct *work)
```

在函数 pm_suspend()中调用 enter_state()函数，这样就进入了标准 linux 的休眠过程。函数 pm_suspend()的原型如下所示：

```
int pm_suspend(suspend_state_t state)
```

在函数 enter_state()中，首先检查一些状态参数，再同步文件系统，然后调用 suspend_prepare()来冻结进程，最后调用 suspend_devices_and_enter()让外设进入休眠。函数 enter_state()的原型如下所示：

```
static int enter_state(suspend_state_t state)
```

在函数 suspend_prepare()中，先通过“pm_prepare_console()”给 suspend 分配一个虚拟终端来输出信息，再广播一个系统进入 suspend 的通报，关闭用户态的 helper 进程，然后调用函数 suspend_freeze_processes()来冻结进程，最后会尝试释放一些内存。函数 suspend_prepare()的原型如下所示：

```
static int suspend_prepare(void)
```

在函数 suspend_freeze_processes()中调用 freeze_processes()函数，而在 freeze_processes()函数中又调用了 try_to_freeze_tasks()函数来完成冻结任务。在冻结过程中，会判断当前进程是否有 wake_lock，如果有则冻结失败，函数会放弃冻结。函数 freeze_processes()的原型如下所示：

```
static int try_to_freeze_tasks(bool sig_only)
```

到此为止，所有的进程都已经停止了，内核进程有可能在停止的时候握有一些信号量，如果这时候在外设里面去解锁这个信号量有可能会发生死锁，所以建议不要在外设的 suspend()里面等待锁。而且在 suspend 的过程中，很多 log 是无法输出的，所以一旦出现问题就非常难以调试。

接下来回到 enter_state()函数中，当冻结进程完成后调用 suspend_devices_and_enter()函数，目的是让外设进入休眠。在该函数中，首先休眠串口，然后通过 device_suspend()函数调用各驱动的 suspend 函数。

当外设进入休眠后调用“suspend_ops->prepare()”，suspend_ops 是板级的 PM 操作，假如是 s3c6410，则被注册在文件“linux_source/arch/arm/plat-s3c64xx/pm.c”中，在里面只定义了 suspend_ops->enter()函数：

```
static struct platform_suspend_ops s3c6410_pm_ops = {
    .enter = s3c6410_pm_enter,
    .valid = suspend_valid_only_mem,
};
```

然后在多 CPU 中关闭非启动 CPU，具体代码如下所示：

```
int suspend_devices_and_enter(suspend_state_t state)
{
    int error;
    if (!suspend_ops)
        return -ENOSYS;
    if (suspend_ops->begin) {
        error = suspend_ops->begin(state);
        if (error)
            goto Close;
    }
    suspend_console();
    suspend_test_start();
    error = device_suspend(PMSG_SUSPEND);
    if (error) {
        printk(KERN_ERR "PM: Some devices failed to suspend\n");
        goto Recover_platform;
    }
    suspend_test_finish("suspend devices");
    if (suspend_test(TEST_DEVICES))
        goto Recover_platform;
    if (suspend_ops->prepare) {
        error = suspend_ops->prepare();
        if (error)
            goto Resume_devices;
    }
    if (suspend_test(TEST_PLATFORM))
        goto Finish;
    error = disable_nonboot_cpus();
    if (!error && !suspend_test(TEST_CPUS))
```

```
        suspend_enter(state);
    enable_nonboot_cpus();
Finish:
    if (suspend_ops->finish)
        suspend_ops->finish();
Resume_devices:
    suspend_test_start();
    device_resume(PMSG_RESUME);
    suspend_test_finish("resume devices");
    resume_console();
Close:
    if (suspend_ops->end)
        suspend_ops->end();
    return error;
Recover_platform:
    if (suspend_ops->recover)
        suspend_ops->recover();
    goto Resume_devices;
}
```

接下来调用函数 suspend_enter()，该函数将首先关闭 IRQ，然后调用 device_power_down()，此函数会调用 suspend_late()函数。这个函数是系统真正进入休眠最后调用的函数，通常会在这个函数中作最后的检查，接下来休眠所有的系统设备和总线。最后调用 suspend_pos→enter()使 CPU 进入省电状态。此时整个休眠过程完成了。函数 suspend_enter()的原型如下所示：

```
static int suspend_enter(suspend_state_t state)
```

14.5.4 Android 唤醒

如果在休眠中系统被中断或者其他事件唤醒，接下来的代码就从 suspend 完成的地方开始执行，以 s3c6410 为例，即在文件 pm.c 中的 s3c6410_pm_enter()中的 cpu_init()，然后执行 suspend_enter()的 sysdev_resume()函数，唤醒系统设备和总线，使能系统中断。然后回到 suspend_devices_and_enter()函数中，使能休眠时候停止掉的非启动 CPU，并且继续唤醒每个设备。当函数 suspend_devices_and_enter()被执行完成后，系统外设已经唤醒，但进程依然处于冻结的状态，返回到 enter_state 函数中，调用 suspend_finish()函数。在函数 suspend_finish()中解冻进程和任务，这样可以使用户空间绑定进程，从而广播一个系统从 suspend 状态退出的 notify（提醒）。

当所有的唤醒已经结束后，用户进程都已经开始运行了，但没点亮屏幕，唤醒通常会是以下的几种原因。

（1）如果是来电，那么 Modem 会发送命令给 rild，这样可以让 rild 通知 WindowManager 有来电响应，这样就会远程调用 PowerManagerService 来写“on”到“/sys/power/state”来调用 late resume()，执行点亮屏幕等操作。

（2）用户按键事件会送到 WindowManager 中，WindowManager 会处理这些按键事件，按键分为几种情况，如果按键不是唤醒键，那么 WindowManager 会主动放弃 wakeLock 来使系统进入再次休眠；如果按键是唤醒键，那么 WindowManger 就会调用 PowerManagerService 中的接口来执行 late Resume。

（3）当“on”被写入到“/sys/power/state“之后，同 early_suspend 过程一样，request_suspend_state()会被调用，只是执行的工作队列变为 late_resume_work。在 late_resume()函数中，唤醒调用了 early_suspend 的设备。

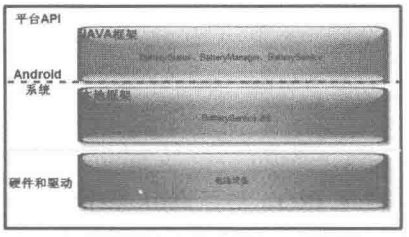
14.6 Battery 电池系统架构和管理

Android 中的电池使用方式主要有 3 种不同的模式，分别是 AC、USB 和 Battery。因为在应用程序层次中通常包括了电池状态显示的功能，所以，从 Android 系统的软件方面（包括驱动程序和用户空间内容）需要在一定程度上获得电池的状态，电池系统主要负责电池信息统计、显示。Android 电池系统的架构如图 14-3 所示。

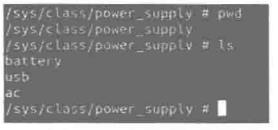
自下而上，Android 的电池系统分成以下所示的几个部分。

14.6.1　实现驱动程序

特定硬件平台电池的驱动程序，用 Linux 的 Power Supply 驱动程序，实现向用户空间提供信息。Battery 驱动程序需要通过 sys 文件系统向用户空间提供接口，sys 文件系统的路径是由上层的程序指定的。Linux 标准的 Power Supply 驱动程序所使用的文件系统路径为“/sys/class/power_supply”，其中的每个子目录表示一种能源供应设备的名称，如图 14-4 所示。



▲图 14-3　Android 电池系统的架构图



▲图 14-4　3 种不同的模式

Power Supply 驱动程序的头文件在 include/linux/power_supply.h 中定义，实现注册和注销驱动程序的函数如下所示：

```
int power_supply_register(struct device *parent,struct power_supply *psy);
void power_supply_unregister(struct power_supply *psy);
struct power_supply {
const char *name;
/* 设备名称 */
enum power_supply_type type;
/* 类型 */
enum power_supply_property *properties;
/* 属性指针 */
size_t num_properties;
/* 属性的数目 */
char **supplied_to;
size_t num_supplicants;
int (*get_property)(struct power_supply *psy, /* 获得属性 */
enum power_supply_property psp,
union power_supply_propval *val);
void (*external_power_changed)(struct power_supply *psy);
/* ..... 省略部分内容 */
};
```

Linux 中的驱动程序为 power_supply，如图 14-5 所示。

▲图 14-5　Linux 中的驱动程序 power_supply

14.6.2　实现 JNI 本地代码

Android 电池系统的代码路径为：

```
frameworks/base/services/jni/com_android_server_BatteryService.cpp
```

类 com_android_server_BatteryService 调用 sys 文件系统访问驱动程序，也同时提供了 JNI 的接口，这个文件提供的方法列表如下所示：

```
static JNINativeMethod sMethods[] = {
{"native_update", "()V", (void*)android_server_BatteryService_update},
};
```

本地 JNI 的处理的流程如下所示。

（1）根据设备类型判定设备后，得到各个设备的相关属性。

（2）如果是交流电或者 USB 设备，则只需要得到它们是否在线（onLine）。

（3）如果是电池设备，则需要得到更多的信息。例如，状态（status）、健康程度（health）、容量（capacity）和电压（voltage_now）等。

Linux 驱动 Driver 维护着保存电池信息的一组文件 sysfs，功能是供应用程序获取电源的相关状态，具体说明如下所示。

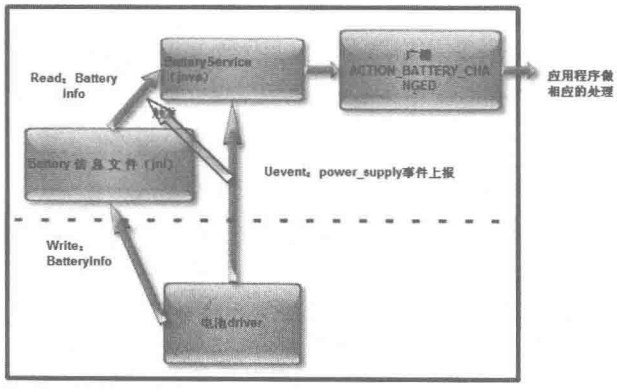
- #define AC_ONLINE_PATH “/sys/class/power_supply/ac/online” AC 电源连接状态。
- #define USB_ONLINE_PATH “/sys/class/power_supply/usb/online” USB 电源连接状态。
- #define BATTERY_STATUS_PATH “/sys/class/power_supply/battery/status” 充电状态。
- #define BATTERY_HEALTH_PATH “/sys/class/power_supply/battery/health” 电池状态。
- #define BATTERY_PRESENT_PATH “/sys/class/power_supply/battery/present” 使用状态。
- #define BATTERY_CAPACITY_PATH“/sys/class/power_supply/battery/capacity”电池 level。
- #define BATTERY_VOLTAGE_PATH “/sys/class/power_supply/battery/batt_vol” 电池电压。
- #define BATTERY_TEMPERATURE_PATH “/sys/class/power_supply/battery/batt_temp” 电池温度。
- #define BATTERY_TECHNOLOGY_PATH “/sys/class/power_supply/battery/technology” 电池技术。当电池状态发生变化时，driver 会更新这些文件传送信息到 Java 层。

14.6.3 Java 层代码

Android 电池系统的 Java 层代码路径为。

- frameworks/base/services/java/com/android/server/BatteryService.java：电池服务文件。
- frameworks/base/core/java/android/os/ : android.os：包中和 Battery 相关的部分。
- frameworks/base/core/java/com/android/internal/os/：和 Battery 相关的内部部分 Battery Service.java。通过调用 BatteryService JNI 来实现 com.android.server 包中的 BatteryService 类。BatteryManager.java 中定义了一些 Java 应用程序层可以使用的常量。

上述文件的调用运行关系如图 14-6 所示。



▲图 14-6 Java 层的调用运行关系

Android 电池系统在驱动程序层以上的部分都是 Android 系统中默认的内容。在移植的过程中基本不需要改动，电池系统需要移植的部分仅有 Battery 驱动程序。Battery 驱动程序用 Linux 标准的 Power Supply 驱动程序与上层的接口是 sys 文件系统，主要用于读取 sys 文件系统中的文件来获取电池相关的信息。

BatteryService 作为电池及充电相关的服务，功能是监听 Uevent、读取 sysfs 中的状态、广播 Intent.ACTION_BATTERY_CHANGED 。BatteryService 实现了一个 UevenObserver mUEvent

Observer。uevent 是 Linux 内核用来向用户空间主动上报事件的机制，对于 Java 程序来说，只实现 UEventObserver 的虚函数 onUEvent，然后注册即可。BatteryService 只关注 power_supply 的事件，所以，在构造函数中实现注册。在文件 BatteryService.java 中，实现 UEventObserver 的代码如下所示：

```
private final UEventObserver mInvalidChargerObserver = new UEventObserver() {
    @Override
    public void onUEvent(UEventObserver.UEvent event) {
        final int invalidCharger = "1".equals(event.get("SWITCH_STATE")) ? 1 : 0;
        synchronized (mLock) {
            if (mInvalidCharger != invalidCharger) {
                mInvalidCharger = invalidCharger;
            }
        }
    }
};
```

在文件 BatteryService.java 中，函数 update 用于读取 sysfs 文件做到同步取得电池信息，然后根据读到的状态更新 BatteryService 的成员变量，并广播一个 Intent 来通知其他关注电源状态的组件。

当 Kernel 有 power_supply 事件上报时，mUEventObserver 调用 update()函数，然后 update 调用 native_update 从 sysfs 中读取相关状态(com_android_server_BatteryService.cpp)。update()函数的具体实现代码如下所示：

```
private void update(BatteryProperties props) {
    synchronized (mLock) {
        if (!mUpdatesStopped) {
            mBatteryProps = props;
            // Process the new values.
            processValuesLocked();
        }
    }
}
```

14.6.4　实现 Uevent 部分

Uevent 是内核通知 Android 有状态变化的一种方法，比如 USB 线插入、拔出，电池电量变化等。其本质是内核发送（可以通过 socket）一个字符串，应用层（Android）接收并解释该字符串，获取相应信息，如图 14-7 所示，如果其中有信息变化，uevent 触发，做出相应的数据更新。

```
root@android:/sys/class/power_supply/ac # cd ../
root@android:/sys/class/power_supply # ls
ac
battery
root@android:/sys/class/power_supply # cd battery/
root@android:/sys/class/power_supply/battery # ls
capacity
device
health
power
present
status
subsystem
technology
temp
type
uevent
voltage_now
root@android:/sys/class/power_supply/battery # cat capacity
90
root@android:/sys/class/power_supply/battery # cat health
Good
root@android:/sys/class/power_supply/battery #
```

▲图 14-7　Uevent 获取信息

Android 中的 BatteryService 及相关组件

（1）Androiduevent 架构

Android 系统中的很多事件都是通过 uevent 与 kernel 进行异步通信的，其中类 UEventObserver 是核心。UEventObserver 接收 kernel 的 uevent 信息的抽象类。

Server 层代码的实现文件如下所示。

- frameworks/frameworks/base/services/java/com/android/server/SystemServer.java。
- frameworks/frameworks/base/services/java/com/android/server/BatteryService.java。

Java 层代码的实现文件是：frameworks/base/core/java/android/os/UEventObserver.java。
JNI 层代码的实现文件是：frameworks/base/core/jni/android_os_UEventObserver.cpp。
底层代码的实现文件是：hardware/libhardware_legacy/uevent/uevent.c。
读写 Kernel 的接口是：socket(PF_NETLINK,SOCK_DGRAM, NETLINK_KOBJECT_UEVENT)。
（2）UEventObserver 类的使用
在文件 UEventObserver.java 中，提供了如下所示的 3 个接口给子类进行调用。

- onUEvent(UEvent event)：子类必须重写这个 onUEvent 来处理 uevent，具体实现代码如下所示：

```
public static final class UEvent {
    // collection of key=value pairs parsed from the uevent message
    private final HashMap<String,String> mMap = new HashMap<String,String>();

    public UEvent(String message) {
        int offset = 0;
        int length = message.length();

        while (offset < length) {
            int equals = message.indexOf('=', offset);
            int at = message.indexOf('\0', offset);
            if (at < 0) break;

            if (equals > offset && equals < at) {
                // key is before the equals sign, and value is after
                mMap.put(message.substring(offset, equals),
                        message.substring(equals + 1, at));
            }

            offset = at + 1;
        }
    }

    public String get(String key) {
        return mMap.get(key);
    }

    public String get(String key, String defaultValue) {
        String result = mMap.get(key);
        return (result == null ? defaultValue : result);
    }

    public String toString() {
        return mMap.toString();
    }
}
```

- startObserving(Stringmatch)：启动进程，要提供一个字符串参数，具体实现代码如下所示：

```
public final void startObserving(String match) {
    if (match == null || match.isEmpty()) {
        throw new IllegalArgumentException("match substring must be non-empty");
    }

    final UEventThread t = getThread();
    t.addObserver(match, this);
}
```

- stopObserving()：停止进程，具体实现代码如下所示：

```
public final void stopObserving() {
    final UEventThread t = getThread();
    if (t != null) {
        t.removeObserver(this);
    }
}
```

在 UEvent thread 中会不停调用 update()方法，来更新电池的信息数据。

（3）vold server 分析

在文件 system/vold/NetlinkManager.cpp 中的实现代码如下所示：

```
if ((mSock = socket(PF_NETLINK,SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
            SLOGE("Unable to create uevent socket: %s", strerror(errno));
            return -1;
    }
    if (setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz)) < 0) {
            SLOGE("Unable to set uevent socket options: %s", strerror(errno));
            return -1;
    }
    if (bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
            SLOGE("Unable to bind uevent socket: %s", strerror(errno));
            return -1;
    }
```

在文件 system/vold/NetlinkHandler.cpp 的 NetlinkHandler::onEvent 中的处理代码如下所示：

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
      VolumeManager *vm = VolumeManager::Instance();
      const char *subsys = evt->getSubsystem();
      if (!subsys) {
          SLOGW("No subsystem found in netlink event");
          return;
      }
      if (!strcmp(subsys, "block")) {
          vm->handleBlockEvent(evt);
      } else if (!strcmp(subsys, "switch")) {
          vm->handleSwitchEvent(evt);
      } else if (!strcmp(subsys, "battery")) {
      } else if (!strcmp(subsys, "power_supply")) {
      }
}
```

最后在文件 system/core/libsysutils/src/NetlinkListener.cpp 中实现监听。

（4）Batteryserver 分析

Java 层的实现代码位于：

```
frameworks/frameworks/base/services/java/com/android/server/BatteryService.java
```

NI 层的实现代码位于：

```
frameworks/base/services/jni/com_android_server_BatteryService.cpp
```

BatteryService 运行在 system_process 当中，在系统初始化时启动，例如在文件 BatteryService.java 中的对应代码如下所示：

```
Log.i(TAG, "Starting Battery Service.");
BatteryService battery = new BatteryService(context);
ServiceManager.addService("battery", battery);
```

BatteryService 通过 JNI（com_android_server_BatteryService.cpp）来读取数据。BatteryService 通过 JNI 不仅注册函数，而且还注册变量。BatteryService 是运行在 system_process 当中，在系统初始化时启动，例如在文件 BatteryService.java 中声明变量的代码如下所示：

```
//###############在 BatteryService.java 中声明的变量################
private boolean mAcOnline;
private boolean mUsbOnline;
private int mBatteryStatus;
private int mBatteryHealth;
private boolean mBatteryPresent;
private int mBatteryLevel;
private int mBatteryVoltage;
private int mBatteryTemperature;
private String mBatteryTechnology;
//在 BatteryService.java 中声明的变量，在 com_android_server_BatteryService.cpp 中共用，即在
com_android_server_BatteryService.cpp 中其实操作的也是 BatteryService.java 中声明的变量
gFieldIds.mAcOnline = env->GetFieldID(clazz, "mAcOnline", "Z");
gFieldIds.mUsbOnline = env->GetFieldID(clazz, "mUsbOnline", "Z");
```

```
gFieldIds.mBatteryStatus = env->GetFieldID(clazz, "mBatteryStatus", "I");
gFieldIds.mBatteryHealth = env->GetFieldID(clazz, "mBatteryHealth", "I");
gFieldIds.mBatteryPresent = env->GetFieldID(clazz, "mBatteryPresent", "Z");
gFieldIds.mBatteryLevel = env->GetFieldID(clazz, "mBatteryLevel", "I");
gFieldIds.mBatteryTechnology = env->GetFieldID(clazz, "mBatteryTechnology", Ljava/
lang/String; ");
gFieldIds.mBatteryVoltage = env->GetFieldID(clazz, "mBatteryVoltage", "I");
gFieldIds.mBatteryTemperature = env->GetFieldID(clazz, "mBatteryTemperature", "I");
//上面这些变量的值，对应是从下面的文件中读取的，一个文件存储一个数值
#define AC_ONLINE_PATH "/sys/class/power_supply/ac/online"
#define USB_ONLINE_PATH "/sys/class/power_supply/usb/online"
#define BATTERY_STATUS_PATH "/sys/class/power_supply/battery/status"
#define BATTERY_HEALTH_PATH "/sys/class/power_supply/battery/health"
#define BATTERY_PRESENT_PATH "/sys/class/power_supply/battery/present"
#define BATTERY_CAPACITY_PATH "/sys/class/power_supply/battery/capacity"
#define BATTERY_VOLTAGE_PATH "/sys/class/power_supply/battery/batt_vol"
#define BATTERY_TEMPERATURE_PATH "/sys/class/power_supply/battery/batt_temp"
#define BATTERY_TECHNOLOGY_PATH "/sys/class/power_supply/battery/technology"
```

BatteryService 主动把数据传送给所关心的应用程序，所有的电池的信息数据是通过 Intent 传送出去的。在文件 BatteryService.java 中，实现数据传送功能的实现代码如下所示：

```
Intent intent = new Intent(Intent.ACTION_BATTERY_CHANGED);
intent.addFlags(Intent.FLAG_RECEIVER_REGISTERED_ONLY);
intent.putExtra("status", mBatteryStatus);
intent.putExtra("health", mBatteryHealth);
intent.putExtra("present", mBatteryPresent);
intent.putExtra("level", mBatteryLevel);
intent.putExtra("scale", BATTERY_SCALE);
intent.putExtra("icon-small", icon);
intent.putExtra("plugged", mPlugType);
intent.putExtra("voltage", mBatteryVoltage);
intent.putExtra("temperature", mBatteryTemperature);
intent.putExtra("technology", mBatteryTechnology);
ActivityManagerNative.broadcastStickyIntent(intent, null);
```

应用程序如果想要接收到 BatteryService 发送出来的电池信息，则需要注册一个 Intent 为 Intent.ACTION_BATTERY_CHANGED 的 BroadcastReceiver。实现数据接收功能的注册方法如下所示：

```
IntentFilter mIntentFilter = new IntentFilter();
mIntentFilter.addAction(Intent.ACTION_BATTERY_CHANGED);
registerReceiver(mIntentReceiver, mIntentFilter);
private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub
        String action = intent.getAction();
        if (action.equals(Intent.ACTION_BATTERY_CHANGED)) {
            int nVoltage = intent.getIntExtra("voltage", 0);
            if(nVoltage!=0){
                mVoltage.setText("V: " + nVoltage + "mV - Success…");
            }
            else{
                mVoltage.setText("V: " + nVoltage + "mV - fail…");
            }
        }
    }
};
```

电池的信息会随着时间不停变化，自然地就需要考虑如何实时地更新电池的数据信息。在 BatteryService 启动时，会同时通过 UEventObserver 启动一个 onUEvent Thread。每一个 Process 最多只能有一个 onUEvent Thread，即使这个 Process 中有多个 UEventObserver 实例。当在一个 Process 中，第一次 Call startObserving()方法后，这个 UEvent thread 就启动。而一旦这个 UEvent thread 启动后，就不会停止。在 BatteryService.java 中实现数据更新功能的代码如下所示：

```
mUEventObserver.startObserving("SUBSYSTEM=power_supply");
private UEventObserver mUEventObserver = new UEventObserver() {
    @Override
```

```
    public void onUEvent(UEventObserver.UEvent event) {
        update();
    }
};
```

在 UEvent thread 中会不停调用 update()方法来更新电池的信息数据。

14.7 JobScheduler 节能调度机制

耗电量大一直是 Android 智能设备的痛处，由于系统原理的问题，这方面一直处理得不够好，大多数设备只能使用一天，如果不充电很少可以连续使用两天。从 Android 5.0 版本开始，将通过 Volta 中的 JobScheduler 机制对这个问题作出一些改进，它通过改善第三方程序的工作序列来降低程序的耗电量。在本节的内容中，将详细讲解 JobScheduler 机制的基本知识。

14.7.1 JobScheduler 机制的推出背景

提高电池续航，也就意味着减少系统和程序的电量消耗。为此，Google 工程师们经过测试发现，每次唤醒设备 1～2 秒时，都会消耗 2 分钟的待机电量。由此可见，每次唤醒设备时，不仅是点亮了屏幕，系统也在后台处理很多事情。而 Android 5.0 版本为了解决这个问题，使用了一个新的 API JobScheduler，这个东西可以让系统批处理一些不重要的 App 请求，例如数据库清理和日志上传等。研发人员也可以使用这个 API 减少自己 App 的不必要操作。

过去，如果开发人员想通过后台调取服务器数据，或完成某些处理工作，应用程序必须先监听是否有事件正在发生，并为自己设定一个唤醒时间。这样每当应用程序开始运行时，开发人员需要检查各种环境条件，以确定是否具备条件让它完成工作，还是需要稍后再试。上述检查方式不仅复杂，而且容易出错，会不断浪费资源。比如当一个应用程序被唤醒后，发现条件不符合就只能去睡觉并为下次唤醒再次设定时间，这是一个反复的过程。

从 Android 5.0 版本开始，上述问题将引用 JobScheduler 机制来修复，它作为一个调度应用程序，负责当应用程序被唤醒时，提供适当的运行环境，所以开发者不用再让程序检测环境是否符合需求，开发人员只需要按照标准的流程来，调度程序会自动为唤醒的程序，准备好运行环境。

应用程序可以使用这个调度程序，来唤醒它们，比如当设备连接到充电器后，调度程序将唤醒那些需要处理器工作的程序，让它们进行工作，或者在设备连接至 WiFi 时上传下载照片、更新内容等。该调度程序还支持一个时间窗口，以便它可以唤醒一组应用程序，这将使那些不需要精确唤醒时间，但每隔一两小时需要运行一次的程序能在同一时间点运行，这样就能让处理器保持更长时间的休眠。

JobScheduler 的优势相当巨大，它不仅可以帮助手机节省电量，实际由于不再需要监听、更改和设置报警，还可以帮助开发人员减少代码书写量。

14.7.2 JobScheduler 的实现

类 JobScheduler 的实现文件是 platform/frameworks/base/core/java/android/app/job/JobScheduler.java，这是从 Android 5.0 开始提供的一种全新 API，可以通过定义 Job 系统以异步方式在稍后的时间或在特定条件（例如，当设备正在充电）下运行，以达到优化电池寿命的目的。类 JobScheduler 的具体功能如下所示。

- 推迟非面向用户的工作。
- 设置有一个需要访问网络或 WiFi 连接的任务。
- 设置应用程序都有一个编号，可以作为一个批次上定期运行的任务。

文件 platform/frameworks/base/core/java/android/app/job/JobScheduler.java 的主要实现代码如下所示：

```
public abstract class JobScheduler {
    /**从 schedule(JobInfo)返回的一个无效参数。*/
    public static final int RESULT_FAILURE = 0;
    /**
     从 schedule(JobInfo)中返回的值，如果此应用程序已对超过时间太短的工作太多的请求
     */
    public static final int RESULT_SUCCESS = 1;

    public abstract int schedule(JobInfo job);

    /**
     * 取消一个待决的 jobscheduler 工作
     */
    public abstract void cancel(int jobId);

    /**
     * 取消所有已注册的 jobscheduler 包的工作
     */
    public abstract void cancelAll();

    /**
     *定义一个所有通过这种包还没有被执行登记工作的列表
     */
    public abstract List<JobInfo> getAllPendingJobs();

}
```

14.7.3 实现操作调度

在 Android 5.0 系统中，JobScheduler API 将把要做的事情全部由类 JobService 来控制。类 JobService 在文件 platform/frameworks/base/core/java/android/app/job/JobService.java 中定义，JobService 作为一个大容器封装了 JobScheduler 传递的数据，这些数据能够设置对操控应用程序的工作调度参数。文件 JobService.java 的具体实现代码如下所示：

```
public abstract class JobService extends Service {
    private static final String TAG = "JobService";
    public static final String PERMISSION_BIND =
            "android.permission.BIND_JOB_SERVICE";
    private final int MSG_EXECUTE_JOB = 0;
    private final int MSG_STOP_JOB = 1;
    private final int MSG_JOB_FINISHED = 2;

    private final Object mHandlerLock = new Object();

    @GuardedBy("mHandlerLock")
    JobHandler mHandler;

    /** Binder for this service. */
    IJobService mBinder = new IJobService.Stub() {
        @Override
        public void startJob(JobParameters jobParams) {
            ensureHandler();
            Message m = Message.obtain(mHandler, MSG_EXECUTE_JOB, jobParams);
            m.sendToTarget();
        }
        @Override
        public void stopJob(JobParameters jobParams) {
            ensureHandler();
            Message m = Message.obtain(mHandler, MSG_STOP_JOB, jobParams);
            m.sendToTarget();
        }
    };

    /** @hide */
    void ensureHandler() {
        synchronized (mHandlerLock) {
            if (mHandler == null) {
                mHandler = new JobHandler(getMainLooper());
            }
```

```
        }
    }
    class JobHandler extends Handler {
        JobHandler(Looper looper) {
            super(looper);
        }

        @Override
        public void handleMessage(Message msg) {
            final JobParameters params = (JobParameters) msg.obj;
            switch (msg.what) {
                case MSG_EXECUTE_JOB:
                    try {
                        boolean workOngoing = JobService.this.onStartJob(params);
                        ackStartMessage(params, workOngoing);
                    } catch (Exception e) {
                        Log.e(TAG, "Error while executing job: " + params.getJobId());
                        throw new RuntimeException(e);
                    }
                    break;
                case MSG_STOP_JOB:
                    try {
                        boolean ret = JobService.this.onStopJob(params);
                        ackStopMessage(params, ret);
                    } catch (Exception e) {
                        Log.e(TAG, "Application unable to handle onStopJob.", e);
                        throw new RuntimeException(e);
                    }
                    break;
                case MSG_JOB_FINISHED:
                    final boolean needsReschedule = (msg.arg2 == 1);
                    IJobCallback callback = params.getCallback();
                    if (callback != null) {
                        try {
                            callback.jobFinished(params.getJobId(), needsReschedule);
                        } catch (RemoteException e) {
                            Log.e(TAG, "Error reporting job finish to system: binder has gone" +
                                    "away.");
                        }
                    } else {
                        Log.e(TAG, "finishJob() called for a nonexistent job id.");
                    }
                    break;
                default:
                    Log.e(TAG, "Unrecognised message received.");
                    break;
            }
        }

        private void ackStartMessage(JobParameters params, boolean workOngoing) {
            final IJobCallback callback = params.getCallback();
            final int jobId = params.getJobId();
            if (callback != null) {
                try {
                    callback.acknowledgeStartMessage(jobId, workOngoing);
                } catch(RemoteException e) {
                    Log.e(TAG, "System unreachable for starting job.");
                }
            } else {
                if (Log.isLoggable(TAG, Log.DEBUG)) {
                    Log.d(TAG, "Attempting to ack a job that has already been processed.");
                }
            }
        }

        private void ackStopMessage(JobParameters params, boolean reschedule) {
            final IJobCallback callback = params.getCallback();
            final int jobId = params.getJobId();
            if (callback != null) {
                try {
                    callback.acknowledgeStopMessage(jobId, reschedule);
```

```
                } catch(RemoteException e) {
                    Log.e(TAG, "System unreachable for stopping job.");
                }
            } else {
                if (Log.isLoggable(TAG, Log.DEBUG)) {
                    Log.d(TAG, "Attempting to ack a job that has already been processed.");
                }
            }
        }
    }
    public abstract boolean onStartJob(JobParameters params);

    public abstract boolean onStopJob(JobParameters params);
    public final void jobFinished(JobParameters params, boolean needsReschedule) {
        ensureHandler();
        Message m = Message.obtain(mHandler, MSG_JOB_FINISHED, params);
        m.arg2 = needsReschedule ? 1 : 0;
        m.sendToTarget();
    }
}
```

由此可见，JobService 服务在执行每个在程序的主线程 Handler 中运行的工作时，意味着必须卸载执行逻辑，从另一个“线程/处理器/AsyncTask”进行选择。如果不这样做，会导致阻止从 JobManager 发出的任何未来回调任务。

14.7.4 封装调度任务

在 Android 5.0 系统中，类 JobInfo 功能封装全部的调度任务。类 JobInfo 在文件 platform/frameworks/base/core/java/android/app/job/JobInfo.java 中定义，定义调度参数的代码如下所示：

```
public class JobInfo implements Parcelable {
    /**默认值 */
    public static final int NETWORK_TYPE_NONE = 0;
    /** 需要网络连接的工作 */
    public static final int NETWORK_TYPE_ANY = 1;
    /**需要网络连接的工作，无需计量*/
    public static final int NETWORK_TYPE_UNMETERED = 2;

    /**
     *补偿工作已在默认情况下，以毫秒为单位
     */
    public static final long DEFAULT_INITIAL_BACKOFF_MILLIS = 30000L;  // 30 seconds.

    /**
     *可以工作时的最大回退值，单位为毫秒
     */
    public static final long MAX_BACKOFF_DELAY_MILLIS = 5 * 60 * 60 * 1000;  // 5 hours.

    /**
    线性回退失败的作业
     */
    public static final int BACKOFF_POLICY_LINEAR = 0;

    /**
     *指数回退失败的作业
     */
    public static final int BACKOFF_POLICY_EXPONENTIAL = 1;
    public static final int DEFAULT_BACKOFF_POLICY = BACKOFF_POLICY_EXPONENTIAL;

    private final int jobId;
    private final PersistableBundle extras;
    private final ComponentName service;
    private final boolean requireCharging;
    private final boolean requireDeviceIdle;
    private final boolean hasEarlyConstraint;
    private final boolean hasLateConstraint;
    private final int networkType;
    private final long minLatencyMillis;
    private final long maxExecutionDelayMillis;
```

```
    private final boolean isPeriodic;
    private final boolean isPersisted;
    private final long intervalMillis;
    private final long initialBackoffMillis;
    private final int backoffPolicy;
```

在文件 JobInfo.java 中，通过类 Builder 来配置应该运行的计划任务，期间可以安排执行任务的具体条件，例如在运行下列应用情形时：

- 当该装置在充电开始；
- 开始时，该装置被连接到一个未计量的网络；
- 开始时，该设备处于空闲状态；
- 往前一段指定的时间段，或以最小的延迟结束。

类 Builder 的具体实现代码如下所示：

```
public static final class Builder {
    private int mJobId;
    private PersistableBundle mExtras = PersistableBundle.EMPTY;
    private ComponentName mJobService;
    // Requirements.
    private boolean mRequiresCharging;
    private boolean mRequiresDeviceIdle;
    private int mNetworkType;
    private boolean mIsPersisted;
    // One-off parameters.
    private long mMinLatencyMillis;
    private long mMaxExecutionDelayMillis;
    // Periodic parameters.
    private boolean mIsPeriodic;
    private boolean mHasEarlyConstraint;
    private boolean mHasLateConstraint;
    private long mIntervalMillis;
    // Back-off parameters.
    private long mInitialBackoffMillis = DEFAULT_INITIAL_BACKOFF_MILLIS;
    private int mBackoffPolicy = DEFAULT_BACKOFF_POLICY;
    /** Easy way to track whether the client has tried to set a back-off policy. */
    private boolean mBackoffPolicySet = false;
    public Builder(int jobId, ComponentName jobService) {
        mJobService = jobService;
        mJobId = jobId;
    }
    public Builder setExtras(PersistableBundle extras) {
        mExtras = extras;
        return this;
    }
    public Builder setRequiredNetworkType(int networkType) {
        mNetworkType = networkType;
        return this;
    }
    public Builder setRequiresCharging(boolean requiresCharging) {
        mRequiresCharging = requiresCharging;
        return this;
    }
    public Builder setRequiresDeviceIdle(boolean requiresDeviceIdle) {
        mRequiresDeviceIdle = requiresDeviceIdle;
        return this;
    }
    public Builder setPeriodic(long intervalMillis) {
        mIsPeriodic = true;
        mIntervalMillis = intervalMillis;
        mHasEarlyConstraint = mHasLateConstraint = true;
        return this;
    }
    public Builder setMinimumLatency(long minLatencyMillis) {
        mMinLatencyMillis = minLatencyMillis;
        mHasEarlyConstraint = true;
        return this;
    }
    public Builder setOverrideDeadline(long maxExecutionDelayMillis) {
```

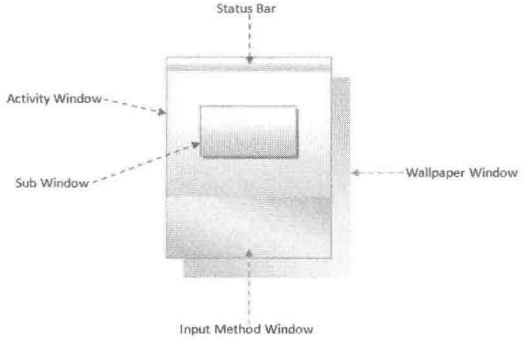
```
            mMaxExecutionDelayMillis = maxExecutionDelayMillis;
            mHasLateConstraint = true;
            return this;
        }
        public Builder setBackoffCriteria(long initialBackoffMillis, int backoffPolicy) {
            mBackoffPolicySet = true;
            mInitialBackoffMillis = initialBackoffMillis;
            mBackoffPolicy = backoffPolicy;
            return this;
        }
        public Builder setPersisted(boolean isPersisted) {
            mIsPersisted = isPersisted;
            return this;
        }
        public JobInfo build() {
            // Allow jobs with no constraints - What am I, a database?
            if (!mHasEarlyConstraint && !mHasLateConstraint && !mRequiresCharging &&
                    !mRequiresDeviceIdle && mNetworkType == NETWORK_TYPE_NONE) {
                throw new IllegalArgumentException("You're trying to build a job with no " +
                        "constraints, this is not allowed.");
            }
            mExtras = new PersistableBundle(mExtras);  // Make our own copy.
            // Check that a deadline was not set on a periodic job.
            if (mIsPeriodic && (mMaxExecutionDelayMillis != 0L)) {
                throw new IllegalArgumentException("Can't call setOverrideDeadline() on a " +
                        "periodic job.");
            }
            if (mIsPeriodic && (mMinLatencyMillis != 0L)) {
                throw new IllegalArgumentException("Can't call setMinimumLatency() on a " +
                        "periodic job");
            }
            if (mBackoffPolicySet && mRequiresDeviceIdle) {
                throw new IllegalArgumentException("An idle mode job will not respect any" +
                        " back-off policy, so calling setBackoffCriteria with" +
                        " setRequiresDeviceIdle is an error.");
            }
            return new JobInfo(this);
        }
    }

}
```

第 15 章　分析 WindowManagerService 系统

如果从整体的角度来看，Android 应用程序窗口的实现十分复杂，因为它们的类型和作用不同，并且会相互影响。在 Android 系统中，对系统中的所有窗口进行管理是通过窗口管理服务 WindowManagerService 实现的。在本章的内容中，将详细讲解 WindowManagerService 系统的基本架构知识。

15.1 WindowManagerService 基础

在 Android 系统中，虽然在同一时刻只有一个 Activity 窗口处于激活状态，但是，这对于 WindowManagerService 服务来说，并不意味着每次只需管理一个 Activity 窗口即可。举个例子，在两个不同的 Activity 窗口的切换过程中，这两个 Activity 窗口实际上都是可见的。即使在只有一个 Activity 窗口是可见的时候，WindowManagerService 服务仍然需要同时管理着多个窗口，这是因为可见的 Activity 窗口可能还会被设置了壁纸窗口（Wallpaper Winodw）或者弹出子窗口（Sub Window），以及可能会出现状态栏（Status Bar）、输入法窗口（Input Method Window）。Activity 窗口及其子窗口、壁纸窗口、输入法窗口和状态栏的位置结构如图 15-1 所示。



▲图 15-1　Activity 窗口及其子窗口、壁纸窗口、输入法窗口和状态栏的位置结构

在 Android 系统中，WindowManagerService 不能假设同一时刻它只需要管理一个窗口，它需要通过各个窗口在屏幕上的位置和大小来决定哪些窗口需要显示以及要显示在哪里，这就需要计算出各个窗口的可见区域。

在 Android 系统中，WindowManagerService 通常用以下方式来控制一个窗口是否显示以及显位置。

（1）因为每一个 Activity 窗口的大小等于屏幕的大小，所以只要对每一个 Activity 窗口设置一个不同的 Z 轴位置，然后就可以使得位于最上面的，即当前被激活的 Activity 窗口是可见的。

（2）每一个子窗口的 *Z* 轴位置都比它的父窗口大，但是大小要比父窗口小，这时候 Activity 窗口及其所弹出的子窗口都可以同时显示出来。

（3）如果是非全屏 Activity 窗口，它会在屏幕的上方留出一块区域以显示状态栏。对于整个屏幕来说，这块留出来的区域被称为装饰区（decoration），而对于 Activity 窗口来说被称为内容周边区（Content Inset）。

（4）只有在需要时才会出现输入法窗口，它同样是出现在屏幕的装饰区或者说 Activity 窗口的内容周边区。

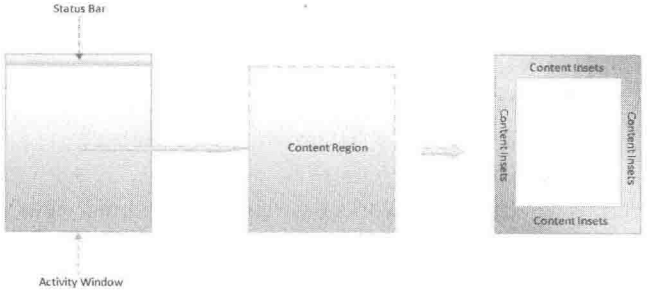
（5）壁纸窗口出现在需要壁纸的 Activity 窗口的下方，这时候要求 Activity 窗口是半透明的，这样就可以将它后面的壁纸窗口一同显示出来。

（6）在切换两个不同 Activity 窗口的过程中，实际上是前一个窗口显示退出动画，而后一个窗口显示开始动画的过程。在动画显示的过程中，窗口的大小会有一个变化的过程，这样就导致前后两个 Activity 窗口的大小不再都等于屏幕的大小，因而它们就有可能同时都处于可见的状态。其实 Activity 窗口的切换过程十分复杂，因为即将要显示的 Activity 窗口可能还会被设置一个启动窗口（Starting Window）。一个被设置了启动窗口的 Activity 窗口，要等到它的启动窗口显示了之后才可以显示出来。

综上所述，窗口在 *X*、*Y* 和 *Z* 轴的位置及其大小的计算非常重要，它们共同决定了一个窗口是否是整体可见，部分可见，还是整体不可见。在 Android 系统中，WindowManagerService 通过一个实现了 WindowManagerPolicy 接口的策略类来计算一个窗口的位置和大小。

15.2 计算 Activity 窗口的大小

在 Android 5.0 系统中，Activity 窗口的大小是由 WindowManagerService 服务来计算的。WindowManagerService 会根据屏幕及其修饰区的大小决定 Activity 窗口的大小。只有知道 Activity 窗口的大小之后，才能对它里面的 UI 元素进行测量、布局并绘制。通常 Activity 窗口的大小等于整个屏幕的大小，但是它并不占据着整块屏幕。Activity 窗口区域的划分如图 15-2 所示。



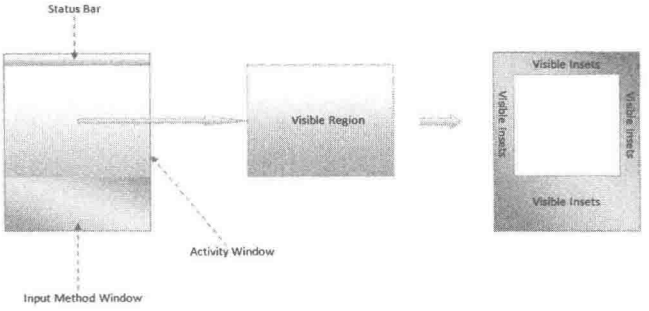
▲图 15-2 Activity 窗口区域的划分

在一个 Activity 窗口中，扣除状态栏所占用的区域之后得到的区域被称为内容区域（Content Region），这个内容区域就是用来显示 Activity 窗口的内容的。假设 Activity 窗口的四周都有一块类似状态栏的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为内容区域，而被剔除出来的区域所组成的区域就称为周边区域（Content Insets）。Activity 窗口的内容周边区域可以用四元组 content-left、content-top、content-right 和 content-bottom 进行描述，其中 content-left、content-right、content-top 和 content-bottom 分别用来描述内容区域与窗口区域的左、右、上、下边界距离。

另外，有时 Activity 窗口还需要显示输入法窗口，如图 15-3 所示。

在图 15-3 的示意图中，Activity 窗口的内容区域的大小可能没发生变化，这取决于它的 Soft Input Mode。假设 Activity 窗口的内容区域没有发生变化，但是在底部的一些区域被输入法窗口遮挡了，即在底部的一些内容是不可见的。当从 Activity 窗口扣除状态栏和输入法窗口所占用的区域之后，所得到的区域就称为可见区域（Visible Region）。同样，假设 Activity 窗口的四周都有一块类似状态栏和输入法窗口的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为可见区域，而被扣除出来的区域所组成的区域就称为可见周边区域（Visible Insets）。Activity 窗口的可见周边区域可以用一个四元组 visible-left、visible-top、visible-right 和 visible-bottom 来描述，

其中，visible-left、visible-right、visible-top 和 visible-bottom 分别用来描述可见区域与窗口区域的左、右、上、下边界距离。在通常情况下，Activity 窗口的内容区域和可见区域的大小是一致的，状态栏和输入法窗口所占用的区域又称为屏幕修饰区。



▲图 15-3　Activity 窗口的 Visible 区域

由此可见，WindowManagerService 需要根据屏幕以及可能出现的状态栏和输入法窗口的大小，来计算出 Activity 窗口的整体大小及其内容区域周边和可见区域周边的大小。有了上述这 3 个数据之后，Activity 窗口就可以对它里面的 UI 元素进行测量、布局以及绘制等操作。在本节的内容中，将详细讲解计算 Activity 窗口大小的具体过程。

15.2.1　实现 View 遍历

在 Android 5.0 源代码中，函数 performTraversals 是进行 View 遍历的核心，该函数的实现过程主要分为如下 4 大步骤：

- 创建 Surface，并打通 Native 层（relayoutWindow）；
- 计算视图大小（performMeasure）；
- 实现布局功能，将视图放置在合适位置（performLayout）；
- 实现绘制功能（performDraw）。

函数 performTraversals 在文件 frameworks/base/core/java/android/view/ViewRootImplImplImplImpl.java 中定义，具体实现过程如下所示。

（1）定义函数 performTraversals，获得 Activity 窗口的当前宽度 desiredWindowWidth 和当前高度 desiredWindowHeight，具体实现代码如下所示：

```
private void performTraversals() {
    // cache mView since it is used so much below...
    final View host = mView;

    if (DBG) {
        System.out.println("======================================");
        System.out.println("performTraversals");
        host.debug();
    }

    if (host == null || !mAdded)
        return;

    mIsInTraversal = true;
    mWillDrawSoon = true;
    boolean windowSizeMayChange = false;
    boolean newSurface = false;
    boolean surfaceChanged = false;
    WindowManager.LayoutParams lp = mWindowAttributes;

    int desiredWindowWidth;
    int desiredWindowHeight;

    final int viewVisibility = getHostVisibility();
```

```
boolean viewVisibilityChanged = mViewVisibility != viewVisibility
        || mNewSurfaceNeeded;

WindowManager.LayoutParams params = null;
if (mWindowAttributesChanged) {
    mWindowAttributesChanged = false;
    surfaceChanged = true;
    params = lp;
}
CompatibilityInfo compatibilityInfo = mDisplayAdjustments.getCompatibilityInfo();
if (compatibilityInfo.supportsScreen() == mLastInCompatMode) {
    params = lp;
    mFullRedrawNeeded = true;
    mLayoutRequested = true;
    if (mLastInCompatMode) {
        params.privateFlags &= ~WindowManager.LayoutParams.PRIVATE_FLAG_COMPATIBLE_
        WINDOW;
        mLastInCompatMode = false;
    } else {
        params.privateFlags |= WindowManager.LayoutParams.PRIVATE_FLAG_COMPATIBLE_
        WINDOW;
        mLastInCompatMode = true;
    }
}

mWindowAttributesChangesFlag = 0;

Rect frame = mWinFrame;
if (mFirst) {
    mFullRedrawNeeded = true;
    mLayoutRequested = true;

    if (lp.type == WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL
            || lp.type == WindowManager.LayoutParams.TYPE_INPUT_METHOD) {
        // NOTE -- system code, won't try to do compat mode.
        Point size = new Point();
        mDisplay.getRealSize(size);
        desiredWindowWidth = size.x;
        desiredWindowHeight = size.y;
    } else {
        DisplayMetrics packageMetrics =
            mView.getContext().getResources().getDisplayMetrics();
        desiredWindowWidth = packageMetrics.widthPixels;
        desiredWindowHeight = packageMetrics.heightPixels;
    }

    // We used to use the following condition to choose 32 bits drawing caches:
    // PixelFormat.hasAlpha(lp.format) || lp.format == PixelFormat.RGBX_8888
    // However, windows are now always 32 bits by default, so choose 32 bits
    mAttachInfo.mUse32BitDrawingCache = true;
    mAttachInfo.mHasWindowFocus = false;
    mAttachInfo.mWindowVisibility = viewVisibility;
    mAttachInfo.mRecomputeGlobalAttributes = false;
    viewVisibilityChanged = false;
    mLastConfiguration.setTo(host.getResources().getConfiguration());
    mLastSystemUiVisibility = mAttachInfo.mSystemUiVisibility;
    // Set the layout direction if it has not been set before (inherit is the default)
    if (mViewLayoutDirectionInitial == View.LAYOUT_DIRECTION_INHERIT) {
        host.setLayoutDirection(mLastConfiguration.getLayoutDirection());
    }
    host.dispatchAttachedToWindow(mAttachInfo, 0);
    mAttachInfo.mTreeObserver.dispatchOnWindowAttachedChange(true);
    dispatchApplyInsets(host);
    //Log.i(TAG, "Screen on initialized: " + attachInfo.mKeepScreenOn);

} else {
    desiredWindowWidth = frame.width();
    desiredWindowHeight = frame.height();
    if (desiredWindowWidth != mWidth || desiredWindowHeight != mHeight) {
        if (DEBUG_ORIENTATION) Log.v(TAG,
                "View " + host + " resized to: " + frame);
```

```
                mFullRedrawNeeded = true;
                mLayoutRequested = true;
                windowSizeMayChange = true;
            }
        }
```

Activity 窗口当前的宽度和高度是保存在 ViewRootImplImplImplImpl 类的成员变量 mWinFrame 中的。类 ViewRootImplImplImplImpl 的另外两个成员变量 mWidth 和 mHeight 也是用来描述 Activity 窗口当前的宽度和高度，但是，它们的值是由应用程序进程上一次主动请求 WindowManagerService 服务计算得到的，并且会一直保持不变，直到应用程序进程下一次再请求 WindowManagerService 服务来重新计算为止。Activity 窗口的当前宽度和高度有时候是被 Window ManagerService 服务主动请求应用程序进程修改的，修改后的值就会保存在类 ViewRootImplImpl ImplImpl 的成员变量 mWinFrame 中，它们可能会与类 ViewRootImplImplImplImpl 的成员变量 mWidth 和 mHeight 的值不同。

如果 Activity 窗口是第一次被请求执行测量、布局和绘制操作，即类 ViewRootImplImplImpl Impl 的成员变量 mFirst 的值等于 true，那么它的当前宽度 desiredWindowWidth 和当前高度 desiredWindowHeight 就等于屏幕的宽度和高度，否则的话，它的当前宽度 desiredWindowWidth 和当前高度 desiredWindowHeight 就等于保存在 ViewRootImplImplImpl 类的成员变量 mWinFrame 中的宽度和高度值。如果 Activity 窗口不是第一次被请求执行测量、布局和绘制操作，并且 Activity 窗口主动上一次请求 WindowManagerService 服务计算得到的宽度 mWidth 和高度 mHeight 不等于 Activity 窗口的当前宽度 desiredWindowWidth 和当前高度 desiredWindowHeight，那么就说明 Activity 窗口的大小发生了变化，这时候变量 windowSizeMayChange 的值就会被标记为 true，以便接下来可以对 Activity 窗口的大小变化进行处理。

类 ViewRootImplImplImpl 的成员变量 mAttachInfo 和 mPendingContentInsets、mPendingVisible Insets。ViewRootImplImplImpl 类的成员变量 mAttachInfo 指向的一个 AttachInfo 对象，这个 AttachInfo 对象用来描述 Activity 窗口的属性，例如，这个 AttachInfo 对象的成员变量 mContentInsets 和 mVisibleInsets 分别用来描述 Activity 窗口上一次主动请求 WindowManagerService 服务计算得到的内容周边大小和可见周边大小，即 Activity 窗口的当前内容周边大小和可见周边大小。类 ViewRootImplImplImpl 的成员变量 mPendingContentInsets 和 mPendingVisibleInsets 也是用来描述 Activity 窗口的内容周边大小和可见周边大小，不过它们是由 WindowManagerService 服务主动请求 Activity 窗口设置的，但是尚未生效。

接下来将分两种情况来分析上述代码。

- 第一种：Activity 窗口是第一次被请求执行测量、布局和绘制操作，即 ViewRootImplImpl Impl 类的成员变量 mFirst 的值等于 true，那么这段代码在测量 Activity 窗口的顶层视图 host 的大小之前，首先会调用这个顶层视图 host 的成员函数 fitSystemWindows 来设置它的 4 个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小，即设置为 Activity 窗口的初始化内容周边大小。这样做的目的是可以在 Activity 窗口的四周留下足够的区域来放置可能会出现的系统窗口，也就是状态栏和输入法窗口。
- 第二种：Activity 窗口不是第一次被请求执行测量、布局和绘制操作，即 ViewRootImplImpl Impl 类的成员变量 mFirst 的值等于 false，那么这段代码就会检查 Activity 窗口是否被 Window ManagerService 服务主动请求设置了一个新的内容周边大小 mPendingContentInsets 和一个新的可见周边大小 mPendingVisibleInsets。如果是的话，那么就会分别将它们保存在 ViewRootImplImpl Impl 类的成员变量 mAttachInfo 所指向的一个 AttachInfo 对象的成员变量 mContentInsets 和成员变量 mVisibleInsets 中。注意，如果 Activity 窗口被 WindowManagerService 服务主动请求设置了一个新的内容周边大小 mPendingContentInsets，那么这段代码同时还需要同步调用 Activity 窗口的顶层视图 host 的成员函数 fitSystemWindows 来将它的 4 个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小设置为新的内容周边大小，并且将变量 insetsChanged 的值设置为 true，表明 Activity 窗口的内容周边大小发生了变化。

在上述第二种情况下，如果 Activity 窗口的宽度被设置为 ViewGroup.LayoutParams.WRAP_

CONTENT，或者高度被设置为 ViewGroup.LayoutParams.WRAP_CONTENT，这说明 Activity 窗口的大小要等于内容区域的大小。但是，由于 Activity 窗口的大小是需要覆盖整个屏幕的，所以，这时会将 Activity 窗口的当前宽度 desiredWindowWidth 和当前高度 desiredWindowHeight 设置为屏幕的宽度和高度。也就是说，如果将 Activity 窗口的宽度和高度设置为 ViewGroup.LayoutParams.WRAP_CONTENT，实际上就意味着它的宽度和高度等于屏幕的宽度和高度。这种情况也意味着 Acitivity 窗口的大小发生了变化，因此，就将变量 windowResizesToFitContent 的值设置为 true。

经过上述一系列处理之后，这段代码就会调用 ViewRootImplImplImpl 类的成员函数 getRootMeasureSpec 来根据 Activity 窗口的当前宽度和宽度测量规范，以及高度和高度测量规范来计算得到它的顶层视图 host 的宽度测量规范 childWidthMeasureSpec 和高度测量规范 childHeightMeasureSpec。有了这两个规范之后，就可以调用 Activity 窗口的顶层视图 host 的成员函数 measure 来执行窗口大小的测量工作。

（2）再看下面的代码：

```
getRunQueue().executeActions(mAttachInfo.mHandler);

boolean insetsChanged = false;

boolean layoutRequested = mLayoutRequested && !mStopped;
if (layoutRequested) {

    final Resources res = mView.getContext().getResources();

    if (mFirst) {
        // make sure touch mode code executes by setting cached value
        // to opposite of the added touch mode.
        mAttachInfo.mInTouchMode = !mAddedTouchMode;
        ensureTouchModeLocally(mAddedTouchMode);
    } else {
        if (!mPendingOverscanInsets.equals(mAttachInfo.mOverscanInsets)) {
            insetsChanged = true;
        }
        if (!mPendingContentInsets.equals(mAttachInfo.mContentInsets)) {
            insetsChanged = true;
        }
        if (!mPendingStableInsets.equals(mAttachInfo.mStableInsets)) {
            insetsChanged = true;
        }
        if (!mPendingVisibleInsets.equals(mAttachInfo.mVisibleInsets)) {
            mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);
            if (DEBUG_LAYOUT) Log.v(TAG, "Visible insets changing to: "
                    + mAttachInfo.mVisibleInsets);
        }
        if (lp.width == ViewGroup.LayoutParams.WRAP_CONTENT
                || lp.height == ViewGroup.LayoutParams.WRAP_CONTENT) {
            windowSizeMayChange = true;

            if (lp.type == WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL
                    || lp.type == WindowManager.LayoutParams.TYPE_INPUT_METHOD) {
                // NOTE -- system code, won't try to do compat mode.
                Point size = new Point();
                mDisplay.getRealSize(size);
                desiredWindowWidth = size.x;
                desiredWindowHeight = size.y;
            } else {
                DisplayMetrics packageMetrics = res.getDisplayMetrics();
                desiredWindowWidth = packageMetrics.widthPixels;
                desiredWindowHeight = packageMetrics.heightPixels;
            }
        }
    }

    // Ask host how big it wants to be
    windowSizeMayChange |= measureHierarchy(host, lp, res,
            desiredWindowWidth, desiredWindowHeight);
}
```

```
if (collectViewAttributes()) {
    params = lp;
}
if (mAttachInfo.mForceReportNewAttributes) {
    mAttachInfo.mForceReportNewAttributes = false;
    params = lp;
}

if (mFirst || mAttachInfo.mViewVisibilityChanged) {
    mAttachInfo.mViewVisibilityChanged = false;
    int resizeMode = mSoftInputMode &
            WindowManager.LayoutParams.SOFT_INPUT_MASK_ADJUST;
    // If we are in auto resize mode, then we need to determine
    // what mode to use now.
    if (resizeMode == WindowManager.LayoutParams.SOFT_INPUT_ADJUST_UNSPECIFIED) {
        final int N = mAttachInfo.mScrollContainers.size();
        for (int i=0; i<N; i++) {
            if (mAttachInfo.mScrollContainers.get(i).isShown()) {
                resizeMode = WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE;
            }
        }
        if (resizeMode == 0) {
            resizeMode = WindowManager.LayoutParams.SOFT_INPUT_ADJUST_PAN;
        }
        if ((lp.softInputMode &
                WindowManager.LayoutParams.SOFT_INPUT_MASK_ADJUST) != resizeMode) {
            lp.softInputMode = (lp.softInputMode &
                    ~WindowManager.LayoutParams.SOFT_INPUT_MASK_ADJUST) |
                    resizeMode;
            params = lp;
        }
    }
}

if (params != null) {
    if ((host.mPrivateFlags & View.PFLAG_REQUEST_TRANSPARENT_REGIONS) != 0) {
        if (!PixelFormat.formatHasAlpha(params.format)) {
            params.format = PixelFormat.TRANSLUCENT;
        }
    }
    mAttachInfo.mOverscanRequested = (params.flags
            & WindowManager.LayoutParams.FLAG_LAYOUT_IN_OVERSCAN) != 0;
}

if (mApplyInsetsRequested) {
    mApplyInsetsRequested = false;
    mLastOverscanRequested = mAttachInfo.mOverscanRequested;
    dispatchApplyInsets(host);
    if (mLayoutRequested) {
        // Short-circuit catching a new layout request here, so
        // we don't need to go through two layout passes when things
        // change due to fitting system windows, which can happen a lot.
        windowSizeMayChange |= measureHierarchy(host, lp,
                mView.getContext().getResources(),
                desiredWindowWidth, desiredWindowHeight);
    }
}

if (layoutRequested) {
    // Clear this now, so that if anything requests a layout in the
    // rest of this function we will catch it and re-run a full
    // layout pass.
    mLayoutRequested = false;
}

boolean windowShouldResize = layoutRequested && windowSizeMayChange
    && ((mWidth != host.getMeasuredWidth() || mHeight != host.getMeasuredHeight())
        || (lp.width == ViewGroup.LayoutParams.WRAP_CONTENT &&
                frame.width() < desiredWindowWidth && frame.width() != mWidth)
        || (lp.height == ViewGroup.LayoutParams.WRAP_CONTENT &&
```

```
                frame.height() < desiredWindowHeight && frame.height() != mHeight));

// Determine whether to compute insets.
// If there are no inset listeners remaining then we may still need to compute
// insets in case the old insets were non-empty and must be reset.
final boolean computesInternalInsets =
        mAttachInfo.mTreeObserver.hasComputeInternalInsetsListeners()
        || mAttachInfo.mHasNonEmptyGivenInternalInsets;
```

上述代码主要完成如下所示的两个功能。

- 检查是否需要处理 Activity 窗口的大小变化事件。如果满足以下条件，那么就需要处理，即将变量 windowShouldResize 的值设置为 true。

➢ ViewRootImplImpl 类的成员变量 mLayoutRequest 的值等于 true，这说明应用程序进程正在请求对 Activity 窗口执行一次测量、布局和绘制操作。

➢ 变量 windowResizesToFitContent 的值等于 true，这说明前面检测到了 Activity 窗口的大小发生了变化。

➢ 前面已经对 Activity 窗口的顶层视图 host 的大小重新进行了测量。如果测量出来的宽度 host.mMeasuredWidth、高度 host.mMeasuredHeight 和 Activity 窗口的当前宽度 mWidth 和高度 mHeight 一样，即使条件 1 和条件 2 能满足，那么也可以认为 Activity 窗口的大小没有发生变化。换句话说，只有当测量出来的大小和当前大小不一致时，才认为 Activity 窗口大小发生了变化。另一方面，如果测量出来的大小和当前大小一致，但是 Activity 窗口的大小被要求设置成 WRAP_CONTENT，即设置成和屏幕的宽度 desiredWindowWidth 和高度 desiredWindowHeight 一致，但是 WindowManagerService 服务请求 Activity 窗口设置的宽度 frame.width()和高度 frame.height()与它们不一致，而且与 Activity 窗口上一次请求 WindowManagerService 服务计算的宽度 mWidth 和高度 mHeight 也不一致，那么也是认为 Activity 窗口大小发生了变化的。

- 检查 Activity 窗口是否需要指定有额外的内容周边区域和可见周边区域。如果有的话，那么变量 attachInfo 所指向的一个 AttachInfo 对象的成员变量 mTreeObserver 所描述的一个 TreeObserver 对象的成员函数 hasComputeInternalInsetsListerner 的返回值 ComputeInternalInsets 就会等于 true。Activity 窗口指定额外的内容周边区域和可见周边区域是为了放置一些额外的内容。

（3）再看下面的代码：

```
boolean insetsPending = false;
int relayoutResult = 0;

if (mFirst || windowShouldResize || insetsChanged ||
        viewVisibilityChanged || params != null) {

    if (viewVisibility == View.VISIBLE) {
        // If this window is giving internal insets to the window
        // manager, and it is being added or changing its visibility,
        // then we want to first give the window manager "fake"
        // insets to cause it to effectively ignore the content of
        // the window during layout.  This avoids it briefly causing
        // other windows to resize/move based on the raw frame of the
        // window, waiting until we can finish laying out this window
        // and get back to the window manager with the ultimately
        // computed insets.
        insetsPending = computesInternalInsets && (mFirst || viewVisibilityChanged);
    }
```

上述代码需要在满足如下的条件之一的前提下才能执行。

- Activity 窗口是第一次执行测量、布局和绘制操作，即 ViewRootImpl 类的成员变量 mFirst 的值等于 true。
- 前面得到的变量 windowShouldResize 的值等于 true，即 Activity 窗口大小的确发生了变化。
- 前面得到的变量 insetsChanged 的值等于 true，即 Activity 窗口的内容区域周边发生了变化。
- Activity 窗口的可见性发生了变化，即变量 viewVisibilityChanged 的值等于 true。
- Activity 窗口的属性发生了变化，即变量 params 指向了一个 WindowManager.LayoutParams

对象。

在满足上述条件之一且 Activity 窗口处于可见状态时，即变量 viewVisibility 的值等于 View.VISIBLE，那么就需要检查接下来请求 WindowManagerService 服务计算大小时，是否要告诉 WindowManagerService 服务它指定了额外的内容区域周边和可见区域周边，但是，这些额外的内容区域周边和可见区域周边又还有确定。这种情况发生在 Activity 窗口第一次执行测量、布局和绘制操作或者由不可见变化可见时。因此，当前面得到的变量 computesInternalInsets 等于 true 时，即 Activity 窗口指定了额外的内容区域周边和可见区域周边，那么就需要检查 ViewRootImpl 类的成员变量 mFirst 或者变量 viewVisibilityChanged 的值是否等于 true。如果这些条件能满足，那么变量 insetsPending 的值就会等于 true，表示 Activity 窗口有额外的内容区域周边和可见区域周边等待指定。

（4）再看下面的代码，这段代码也需要在满足（3）中的条件之一的前提下才执行：

```
boolean contentInsetsChanged = false;
boolean hadSurface = mSurface.isValid();

try {
    if (DEBUG_LAYOUT) {
        Log.i(TAG, "host=w:" + host.getMeasuredWidth() + ", h:" +
                host.getMeasuredHeight() + ", params=" + params);
    }

    if (mAttachInfo.mHardwareRenderer != null) {
        // relayoutWindow may decide to destroy mSurface. As that decision
        // happens in WindowManager service, we need to be defensive here
        // and stop using the surface in case it gets destroyed.
        mAttachInfo.mHardwareRenderer.pauseSurface(mSurface);
    }
    final int surfaceGenerationId = mSurface.getGenerationId();
    relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);
    if (!mDrawDuringWindowsAnimating &&
            (relayoutResult & WindowManagerGlobal.RELAYOUT_RES_ANIMATING) != 0) {
        mWindowsAnimating = true;
    }

    if (DEBUG_LAYOUT) Log.v(TAG, "relayout: frame=" + frame.toShortString()
            + " overscan=" + mPendingOverscanInsets.toShortString()
            + " content=" + mPendingContentInsets.toShortString()
            + " visible=" + mPendingVisibleInsets.toShortString()
            + " visible=" + mPendingStableInsets.toShortString()
            + " surface=" + mSurface);

    if (mPendingConfiguration.seq != 0) {
        if (DEBUG_CONFIGURATION) Log.v(TAG, "Visible with new config: "
                + mPendingConfiguration);
        updateConfiguration(mPendingConfiguration, !mFirst);
        mPendingConfiguration.seq = 0;
    }

    final boolean overscanInsetsChanged = !mPendingOverscanInsets.equals(
            mAttachInfo.mOverscanInsets);
    contentInsetsChanged = !mPendingContentInsets.equals(
            mAttachInfo.mContentInsets);
    final boolean visibleInsetsChanged = !mPendingVisibleInsets.equals(
            mAttachInfo.mVisibleInsets);
    final boolean stableInsetsChanged = !mPendingStableInsets.equals(
            mAttachInfo.mStableInsets);
    if (contentInsetsChanged) {
        if (mWidth > 0 && mHeight > 0 && lp != null &&
                ((lp.systemUiVisibility|lp.subtreeSystemUiVisibility)
                        & View.SYSTEM_UI_LAYOUT_FLAGS) == 0 &&
                mSurface != null && mSurface.isValid() &&
                !mAttachInfo.mTurnOffWindowResizeAnim &&
                mAttachInfo.mHardwareRenderer != null &&
                mAttachInfo.mHardwareRenderer.isEnabled() &&
                lp != null && !PixelFormat.formatHasAlpha(lp.format)
                && !mBlockResizeBuffer) {
```

```
            disposeResizeBuffer();
        }
        mAttachInfo.mContentInsets.set(mPendingContentInsets);
        if (DEBUG_LAYOUT) Log.v(TAG, "Content insets changing to: "
                + mAttachInfo.mContentInsets);
    }
    if (overscanInsetsChanged) {
        mAttachInfo.mOverscanInsets.set(mPendingOverscanInsets);
        if (DEBUG_LAYOUT) Log.v(TAG, "Overscan insets changing to: "
                + mAttachInfo.mOverscanInsets);
        // Need to relayout with content insets.
        contentInsetsChanged = true;
    }
    if (stableInsetsChanged) {
        mAttachInfo.mStableInsets.set(mPendingStableInsets);
        if (DEBUG_LAYOUT) Log.v(TAG, "Decor insets changing to: "
                + mAttachInfo.mStableInsets);
        // Need to relayout with content insets.
        contentInsetsChanged = true;
    }
    if (contentInsetsChanged || mLastSystemUiVisibility !=
            mAttachInfo.mSystemUiVisibility || mApplyInsetsRequested
            || mLastOverscanRequested != mAttachInfo.mOverscanRequested) {
        mLastSystemUiVisibility = mAttachInfo.mSystemUiVisibility;
        mLastOverscanRequested = mAttachInfo.mOverscanRequested;
        mApplyInsetsRequested = false;
        dispatchApplyInsets(host);
    }
    if (visibleInsetsChanged) {
        mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);
        if (DEBUG_LAYOUT) Log.v(TAG, "Visible insets changing to: "
                + mAttachInfo.mVisibleInsets);
    }

    if (!hadSurface) {
        if (mSurface.isValid()) {
            // If we are creating a new surface, then we need to
            // completely redraw it.  Also, when we get to the
            // point of drawing it we will hold off and schedule
            // a new traversal instead.  This is so we can tell the
            // window manager about all of the windows being displayed
            // before actually drawing them, so it can display then
            // all at once.
            newSurface = true;
            mFullRedrawNeeded = true;
            mPreviousTransparentRegion.setEmpty();

            if (mAttachInfo.mHardwareRenderer != null) {
                try {
                    hwInitialized = mAttachInfo.mHardwareRenderer.initialize(
                            mSurface);
                } catch (OutOfResourcesException e) {
                    handleOutOfResourcesException(e);
                    return;
                }
            }
        }
    } else if (!mSurface.isValid()) {
        // If the surface has been removed, then reset the scroll
        // positions.
        if (mLastScrolledFocus != null) {
            mLastScrolledFocus.clear();
        }
        mScrollY = mCurScrollY = 0;
        if (mScroller != null) {
            mScroller.abortAnimation();
        }
        disposeResizeBuffer();
        // Our surface is gone
        if (mAttachInfo.mHardwareRenderer != null &&
```

```
                mAttachInfo.mHardwareRenderer.isEnabled()) {
            mAttachInfo.mHardwareRenderer.destroy();
        }
    } else if (surfaceGenerationId != mSurface.getGenerationId() &&
            mSurfaceHolder == null && mAttachInfo.mHardwareRenderer != null) {
        mFullRedrawNeeded = true;
        try {
            mAttachInfo.mHardwareRenderer.updateSurface(mSurface);
        } catch (OutOfResourcesException e) {
            handleOutOfResourcesException(e);
            return;
        }
    }
} catch (RemoteException e) {
}

if (DEBUG_ORIENTATION) Log.v(
        TAG, "Relayout returned: frame=" + frame + ", surface=" + mSurface);

mAttachInfo.mWindowLeft = frame.left;
mAttachInfo.mWindowTop = frame.top;

// !!FIXME!! This next section handles the case where we did not get the
// window size we asked for. We should avoid this by getting a maximum size from
// the window session beforehand.
if (mWidth != frame.width() || mHeight != frame.height()) {
    mWidth = frame.width();
    mHeight = frame.height();
}
```

上述代码调用了 ViewRootImpl 类的成员函数 relayoutWindow，用于请求 WindowManagerService 服务计算 Activity 窗口的大小以及内容区域周边大小和可见区域周边大小。计算完毕之后，Activity 窗口的大小就会保存在 ViewRootImpl 类的成员变量 mWinFrame 中，而 Activity 窗口的内容区域周边大小和可见区域周边大小分别保存在 ViewRootImpl 类的成员变量 mPendingContentInsets 和 mPendingVisibleInsets 中。如果这次计算得到的 Activity 窗口的内容区域周边大小 mPendingContentInsets 和可见区域周边大小 mPendingVisibleInsets 与上一次计算得到的不一致，即与 ViewRootImpl 类的成员变量 mAttachInfo 所指向的一个 AttachInfo 对象的成员变量 mContentInsets 和 mVisibleInsets 所描述的大小不一致，那么变量 contentInsetsChanged 和 visibleInsetsChanged 的值就会等于 true，表示 Activity 窗口的内容区域周边大小和可见区域周边大小发生了变化。

因为变量 frame 和 ViewRootImpl 类的成员变量 mWinFrame 引用的是同一个 Rect 对象，因此，这时候变量 frame 描述的也是 Activity 窗口请求 WindowManagerService 服务计算之后得到的大小。这段代码分别将计算得到的 Activity 窗口的左上角坐标保存在变量 attachInfo 所指向的一个 AttachInfo 对象的成员变量 mWindowLeft 和 mWindowTop 中，并且将计算得到的 Activity 窗口的宽度和高度保存在 ViewRootImpl 类的成员变量 mWidth 和 mHeight 中。

（5）再看下面的代码，这段代码也需要在满足（3）中的条件之一的前提下才执行：

```
if (!mStopped) {
    boolean focusChangedDueToTouchMode = ensureTouchModeLocally(
            (relayoutResult&WindowManagerGlobal.RELAYOUT_RES_IN_TOUCH_MODE) != 0);
    if (focusChangedDueToTouchMode || mWidth != host.getMeasuredWidth()
            || mHeight != host.getMeasuredHeight() || contentInsetsChanged) {
        int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
        int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);

        if (DEBUG_LAYOUT) Log.v(TAG, "Ooops, something changed!  mWidth="
                + mWidth + " measuredWidth=" + host.getMeasuredWidth()
                + " mHeight=" + mHeight
                + " measuredHeight=" + host.getMeasuredHeight()
                + " coveredInsetsChanged=" + contentInsetsChanged);

         // Ask host how big it wants to be
        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);

        // Implementation of weights from WindowManager.LayoutParams
```

```
        // We just grow the dimensions as needed and re-measure if
        // needs be
        int width = host.getMeasuredWidth();
        int height = host.getMeasuredHeight();
        boolean measureAgain = false;

        if (lp.horizontalWeight > 0.0f) {
            width += (int) ((mWidth - width) * lp.horizontalWeight);
            childWidthMeasureSpec = MeasureSpec.makeMeasureSpec(width,
                    MeasureSpec.EXACTLY);
            measureAgain = true;
        }
        if (lp.verticalWeight > 0.0f) {
            height += (int) ((mHeight - height) * lp.verticalWeight);
            childHeightMeasureSpec = MeasureSpec.makeMeasureSpec(height,
                    MeasureSpec.EXACTLY);
            measureAgain = true;
        }

        if (measureAgain) {
            if (DEBUG_LAYOUT) Log.v(TAG,
                    "And hey let's measure once more: width=" + width
                    + " height=" + height);
            performMeasure(childWidthMeasureSpec, childHeightMeasureSpec);
        }

        layoutRequested = true;
    }
}
```

上述代码用于检查是否需要重新测量 Activity 窗口的大小，如果满足如下条件之一就需要重新进行测量工作。

- Activity 窗口的触摸模式发生了变化，并且由此引发了 Activity 窗口当前获得焦点的控件发生了变化，即变量 focusChangedDueToTouchMode 的值等于 true。这个检查是通过调用 ViewRootImpl 类的成员函数 ensureTouchModeLocally 来实现的。
- Activity 窗口前面测量出来的宽度 host.mMeasuredWidth 和高度 host.mMeasuredHeight 不等于 WindowManagerService 服务计算出来的宽度 mWidth 和高度 mHeight。
- Activity 窗口的内容区域周边大小和可见区域周边大小发生了变化，即前面得到的变量 contentInsetsChanged 的值等于 true。

这样在重新计算了一次后，如果 Activity 窗口的属性 lp 表明需要对测量出来的宽度 width 和高度 height 进行扩展，即变量 lp 所指向的一个 WindowManager.LayoutParams 对象的成员变量 horizontalWeight 和 verticalWeight 的值大于 0.0，那么就需要对 Activity 窗口的顶层视图 host 的最大可用空间进行扩展后再进行一次测量工作。

（6）再看下面的代码，Activity 窗口的大小测量工作全部结束，此时就可以对 Activity 窗口的内容进行布局，但前提是 ViewRoot 类的成员变量 mLayoutRequest 的值等于 true。对 Activity 窗口的内容进行布局是通过调用它的顶层视图 host 的成员函数 layout 来实现的。

```
if (computesInternalInsets) {
    // Clear the original insets.
    final ViewTreeObserver.InternalInsetsInfo insets = mAttachInfo.mGivenInternalInsets;
    insets.reset();

    // Compute new insets in place.
    mAttachInfo.mTreeObserver.dispatchOnComputeInternalInsets(insets);
    mAttachInfo.mHasNonEmptyGivenInternalInsets = !insets.isEmpty();

    // Tell the window manager.
    if (insetsPending || !mLastGivenInsets.equals(insets)) {
        mLastGivenInsets.set(insets);

        // Translate insets to screen coordinates if needed.
        final Rect contentInsets;
        final Rect visibleInsets;
        final Region touchableRegion;
```

```
        if (mTranslator != null) {
            contentInsets = mTranslator.getTranslatedContentInsets(insets.contentInsets);
            visibleInsets = mTranslator.getTranslatedVisibleInsets(insets.visibleInsets);
            touchableRegion = mTranslator.getTranslatedTouchableArea(insets.touchable Region);
        } else {
            contentInsets = insets.contentInsets;
            visibleInsets = insets.visibleInsets;
            touchableRegion = insets.touchableRegion;
        }

        try {
            mWindowSession.setInsets(mWindow, insets.mTouchableInsets,
                    contentInsets, visibleInsets, touchableRegion);
        } catch (RemoteException e) {
        }
    }
}
```

当变量 computesInternalInsets 的值等于 true 时，就表示 Activity 窗口指定有额外的内容区域周边和可见区域周边，这时把它们告诉给 WindowManagerService 服务，以便 WindowManager Service 服务下次可以知道 Activity 窗口的真实布局。Activity 窗口额外指定的内容区域周边大小和可见区域周边大小是通过调用变量 attachInfo 所指向的一个 AttachInfo 对象的成员变量 mTreeObserver 所描述的一个 TreeObserver 对象的成员函数 dispatchOnComputeInternalInsets 来计算的。计算完成之后，就会保存在变量 attachInfo 所指向的一个 AttachInfo 对象的成员变量 mGivenInternalInsets 中，并且会通过 ViewRootImpl 类的静态成员变量 sWindowSession 所指向一个 Binder 代理对象来设置到 WindowManagerService 服务中去。

15.2.2　函数 relayoutWindow

函数 relayoutWindow 的功能是请求 WindowManagerService 服务计算 Activity 窗口的大小，此函数在文件 frameworks/base/core/java/android/view/ViewRootImplImplImplImpl.java 中定义，具体实现代码如下所示：

```
private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,
        boolean insetsPending) throws RemoteException {

    float appScale = mAttachInfo.mApplicationScale;
    boolean restore = false;
    if (params != null && mTranslator != null) {
        restore = true;
        params.backup();
        mTranslator.translateWindowLayout(params);
    }
    if (params != null) {
        if (DBG) Log.d(TAG, "WindowLayout in layoutWindow:" + params);
    }
    mPendingConfiguration.seq = 0;
    //Log.d(TAG, ">>>>>> CALLING relayout");
    if (params != null && mOrigWindowType != params.type) {
        // For compatibility with old apps, don't crash here.
        if (mTargetSdkVersion < android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
            Slog.w(TAG, "Window type can not be changed after "
                    + "the window is added; ignoring change of " + mView);
            params.type = mOrigWindowType;
        }
    }
    int relayoutResult = mWindowSession.relayout(
            mWindow, mSeq, params,
            (int) (mView.getMeasuredWidth() * appScale + 0.5f),
            (int) (mView.getMeasuredHeight() * appScale + 0.5f),
            viewVisibility, insetsPending ? WindowManagerGlobal.RELAYOUT_INSETS_PENDING : 0,
            mWinFrame, mPendingOverscanInsets, mPendingContentInsets, mPendingVisible
            Insets, mPendingStableInsets, mPendingConfiguration, mSurface);
    //Log.d(TAG, "<<<<<< BACK FROM relayout");
    if (restore) {
        params.restore();
```

```
        }

        if (mTranslator != null) {
            mTranslator.translateRectInScreenToAppWinFrame(mWinFrame);
            mTranslator.translateRectInScreenToAppWindow(mPendingOverscanInsets);
            mTranslator.translateRectInScreenToAppWindow(mPendingContentInsets);
            mTranslator.translateRectInScreenToAppWindow(mPendingVisibleInsets);
            mTranslator.translateRectInScreenToAppWindow(mPendingStableInsets);
        }
        return relayoutResult;
    }
```

因为 sWindowSession 是一个 Binder 代理对象，引用了运行在 WindowManagerService 服务这一侧的一个 Session 对象，所以，成员函数 relayoutWindow 通过调用这个 Session 对象的成员函数 relayout 来请求 WindowManagerService 服务计算 Activity 窗口的大小，其中，传递给 WindowManagerService 服务的参数包括。

- 成员变量 mWindow 用来标志要计算的是哪一个 Activity 窗口的大小。
- Activity 窗口的顶层视图经过测量后得到宽度和高度。
- Activity 窗口的可见状态，即参数 viewVisibility。
- Activity 窗口是否有额外的内容区域周边和可见区域周边等待告诉给 WindowManagerService 服务，即参数 insetsPending。
- 成员变量 mWinFrame，这是一个输出参数，用来保存 WindowManagerService 服务计算后得到的 Activity 窗口的大小。
- 成员变量 mPendingContentInsets，这是一个输出参数，用来保存 WindowManagerService 服务计算后得到的 Activity 窗口的内容区域周边大小。
- 成员变量 mPendingVisibleInsets，这是一个输出参数，用来保存 WindowManagerService 服务计算后得到的 Activity 窗口的可见区域周边大小。
- 成员变量 mPendingConfiguration，这是一个输出参数，用来保存 WindowManagerService 服务返回来的 Activity 窗口的配置信息。
- 成员变量 mSurface，这是一个输出参数，用来保存 WindowManagerService 服务返回来的 Activity 窗口的绘图表面。

当得到了 Activity 窗口的大小和内容区域周边大小和可见区域周边大小之后，如果 Activity 窗口是运行在兼容模式中，即类 ViewRootImpl 的成员变量 mTranslator 指向了一个 Translator 对象，那么就需要调用成员函数 translateRectInScreenToAppWindow 对它们进行转换。

15.2.3 函数 relayoutWindow

函数 relayoutWindow 的功能是计算参数 window 所描述的一个 Activity 窗口的大小，此函数在文件 frameworks\base\services\core\java\com\android\server\wm\WindowManagerService.java 中定义，具体实现代码如下所示：

```
public int relayoutWindow(Session session, IWindow client, int seq,
        WindowManager.LayoutParams attrs, int requestedWidth,
        int requestedHeight, int viewVisibility, int flags,
        Rect outFrame, Rect outOverscanInsets, Rect outContentInsets,
        Rect outVisibleInsets, Rect outStableInsets, Configuration outConfig,
        Surface outSurface) {
    boolean toBeDisplayed = false;
    boolean inTouchMode;
    boolean configChanged;
    boolean surfaceChanged = false;
    boolean animating;
    boolean hasStatusBarPermission =
            mContext.checkCallingOrSelfPermission(android.Manifest.permission.
            STATUS_BAR)
                    == PackageManager.PERMISSION_GRANTED;

    long origId = Binder.clearCallingIdentity();
```

```
synchronized(mWindowMap) {
    WindowState win = windowForClientLocked(session, client, false);
    if (win == null) {
        return 0;
    }
    WindowStateAnimator winAnimator = win.mWinAnimator;
    if (win.mRequestedWidth != requestedWidth
            || win.mRequestedHeight != requestedHeight) {
        win.mLayoutNeeded = true;
        win.mRequestedWidth = requestedWidth;
        win.mRequestedHeight = requestedHeight;
    }

    if (attrs != null) {
        mPolicy.adjustWindowParamsLw(attrs);
    }

    // if they don't have the permission, mask out the status bar bits
    int systemUiVisibility = 0;
    if (attrs != null) {
        systemUiVisibility = (attrs.systemUiVisibility|attrs.subtreeSystemUi
        Visibility);
        if ((systemUiVisibility & StatusBarManager.DISABLE_MASK) != 0) {
            if (!hasStatusBarPermission) {
                systemUiVisibility &= ~StatusBarManager.DISABLE_MASK;
            }
        }
    }

    if (attrs != null && seq == win.mSeq) {
        win.mSystemUiVisibility = systemUiVisibility;
    }

    winAnimator.mSurfaceDestroyDeferred =
            (flags&WindowManagerGlobal.RELAYOUT_DEFER_SURFACE_DESTROY) != 0;

    int attrChanges = 0;
    int flagChanges = 0;
    if (attrs != null) {
        if (win.mAttrs.type != attrs.type) {
            throw new IllegalArgumentException(
                    "Window type can not be changed after the window is added.");
        }
        flagChanges = win.mAttrs.flags ^= attrs.flags;
        attrChanges = win.mAttrs.copyFrom(attrs);
        if ((attrChanges & (WindowManager.LayoutParams.LAYOUT_CHANGED
                | WindowManager.LayoutParams.SYSTEM_UI_VISIBILITY_CHANGED)) != 0) {
            win.mLayoutNeeded = true;
        }
    }

    if (DEBUG_LAYOUT) Slog.v(TAG, "Relayout " + win + ": viewVisibility=" +
    viewVisibility
            + " req=" + requestedWidth + "x" + requestedHeight + " " + win.mAttrs);

    win.mEnforceSizeCompat =
            (win.mAttrs.privateFlags & PRIVATE_FLAG_COMPATIBLE_WINDOW) != 0;

    if ((attrChanges & WindowManager.LayoutParams.ALPHA_CHANGED) != 0) {
        winAnimator.mAlpha = attrs.alpha;
    }

    final boolean scaledWindow =
        ((win.mAttrs.flags & WindowManager.LayoutParams.FLAG_SCALED) != 0);

    if (scaledWindow) {
        // requested{Width|Height} Surface's physical size
        // attrs.{width|height} Size on screen
        win.mHScale = (attrs.width  != requestedWidth)  ?
                (attrs.width  / (float)requestedWidth) : 1.0f;
```

```
            win.mVScale = (attrs.height != requestedHeight) ?
                    (attrs.height / (float)requestedHeight) : 1.0f;
        } else {
            win.mHScale = win.mVScale = 1;
        }

        boolean imMayMove = (flagChanges & (FLAG_ALT_FOCUSABLE_IM | FLAG_NOT_
        FOCUSABLE)) != 0;

        final boolean isDefaultDisplay = win.isDefaultDisplay();
        boolean focusMayChange = isDefaultDisplay && (win.mViewVisibility !=
        viewVisibility
                || ((flagChanges & FLAG_NOT_FOCUSABLE) != 0)
                || (!win.mRelayoutCalled));

        boolean wallpaperMayMove = win.mViewVisibility != viewVisibility
                && (win.mAttrs.flags & FLAG_SHOW_WALLPAPER) != 0;
        wallpaperMayMove |= (flagChanges & FLAG_SHOW_WALLPAPER) != 0;

        win.mRelayoutCalled = true;
        final int oldVisibility = win.mViewVisibility;
        win.mViewVisibility = viewVisibility;
        if (DEBUG_SCREEN_ON) {
            RuntimeException stack = new RuntimeException();
            stack.fillInStackTrace();
            Slog.i(TAG, "Relayout " + win + ": oldVis=" + oldVisibility
                    + " newVis=" + viewVisibility, stack);
        }
        if (viewVisibility == View.VISIBLE &&
                (win.mAppToken == null || !win.mAppToken.clientHidden)) {
            toBeDisplayed = !win.isVisibleLw();
            if (win.mExiting) {
                winAnimator.cancelExitAnimationForNextAnimationLocked();
                win.mExiting = false;
            }
            if (win.mDestroying) {
                win.mDestroying = false;
                mDestroySurface.remove(win);
            }
            if (oldVisibility == View.GONE) {
                winAnimator.mEnterAnimationPending = true;
            }
            if (toBeDisplayed) {
                if (win.isDrawnLw() && okToDisplay()) {
                    winAnimator.applyEnterAnimationLocked();
                }
                if ((win.mAttrs.flags
                        & WindowManager.LayoutParams.FLAG_TURN_SCREEN_ON) != 0) {
                    if (DEBUG_VISIBILITY) Slog.v(TAG,
                            "Relayout window turning screen on: " + win);
                    win.mTurnOnScreen = true;
                }
                if (win.isConfigChanged()) {
                    if (DEBUG_CONFIGURATION) Slog.i(TAG, "Window " + win
                            + " visible with new config: " + mCurConfiguration);
                    outConfig.setTo(mCurConfiguration);
                }
            }
            if ((attrChanges&WindowManager.LayoutParams.FORMAT_CHANGED) != 0) {
                // To change the format, we need to re-build the surface.
                winAnimator.destroySurfaceLocked();
                toBeDisplayed = true;
                surfaceChanged = true;
            }
            try {
                if (!win.mHasSurface) {
                    surfaceChanged = true;
                }
                SurfaceControl surfaceControl = winAnimator.createSurfaceLocked();
                if (surfaceControl != null) {
                    outSurface.copyFrom(surfaceControl);
```

```
                if (SHOW_TRANSACTIONS) Slog.i(TAG,
                        "  OUT SURFACE " + outSurface + ": copied");
            } else {
                // For some reason there isn't a surface.  Clear the
                // caller's object so they see the same state.
                outSurface.release();
            }
        } catch (Exception e) {
            mInputMonitor.updateInputWindowsLw(true /*force*/);

            Slog.w(TAG, "Exception thrown when creating surface for client "
                    + client + " (" + win.mAttrs.getTitle() + ")",
                    e);
            Binder.restoreCallingIdentity(origId);
            return 0;
        }
        if (toBeDisplayed) {
            focusMayChange = isDefaultDisplay;
        }
        if (win.mAttrs.type == TYPE_INPUT_METHOD
                && mInputMethodWindow == null) {
            mInputMethodWindow = win;
            imMayMove = true;
        }
        if (win.mAttrs.type == TYPE_BASE_APPLICATION
                && win.mAppToken != null
                && win.mAppToken.startingWindow != null) {
            // Special handling of starting window over the base
            // window of the app: propagate lock screen flags to it,
            // to provide the correct semantics while starting.
            final int mask =
                WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED
                | WindowManager.LayoutParams.FLAG_DISMISS_KEYGUARD
                | WindowManager.LayoutParams.FLAG_ALLOW_LOCK_WHILE_SCREEN_ON;
            WindowManager.LayoutParams sa = win.mAppToken.startingWindow.mAttrs;
            sa.flags = (sa.flags&~mask) | (win.mAttrs.flags&mask);
        }
    } else {
        winAnimator.mEnterAnimationPending = false;
        if (winAnimator.mSurfaceControl != null) {
            if (DEBUG_VISIBILITY) Slog.i(TAG, "Relayout invis " + win
                    + ": mExiting=" + win.mExiting);
            // If we are not currently running the exit animation, we
            // need to see about starting one.
            if (!win.mExiting) {
                surfaceChanged = true;
                // Try starting an animation; if there isn't one, we
                // can destroy the surface right away.
                int transit = WindowManagerPolicy.TRANSIT_EXIT;
                if (win.mAttrs.type == TYPE_APPLICATION_STARTING) {
                    transit = WindowManagerPolicy.TRANSIT_PREVIEW_DONE;
                }
                if (win.isWinVisibleLw() &&
                        winAnimator.applyAnimationLocked(transit, false)) {
                    focusMayChange = isDefaultDisplay;
                    win.mExiting = true;
                } else if (win.mWinAnimator.isAnimating()) {
                    // Currently in a hide animation... turn this into
                    // an exit.
                    win.mExiting = true;
                } else if (win == mWallpaperTarget) {
                    // If the wallpaper is currently behind this
                    // window, we need to change both of them inside
                    // of a transaction to avoid artifacts.
                    win.mExiting = true;
                    win.mWinAnimator.mAnimating = true;
                } else {
                    if (mInputMethodWindow == win) {
                        mInputMethodWindow = null;
                    }
                    winAnimator.destroySurfaceLocked();
```

```
            }
            //TODO (multidisplay): Magnification is supported only for the default
            if (mAccessibilityController != null
                    && win.getDisplayId() == Display.DEFAULT_DISPLAY) {
                mAccessibilityController.onWindowTransitionLocked(win, transit);
            }
        }
    }

    outSurface.release();
    if (DEBUG_VISIBILITY) Slog.i(TAG, "Releasing surface in: " + win);
}

if (focusMayChange) {
    //System.out.println("Focus may change: " + win.mAttrs.getTitle());
    if (updateFocusedWindowLocked(UPDATE_FOCUS_WILL_PLACE_SURFACES,
            false /*updateInputWindows*/)) {
        imMayMove = false;
    }
    //System.out.println("Relayout " + win + ": focus=" + mCurrentFocus);
}

// updateFocusedWindowLocked() already assigned layers so we only need to
// reassign them at this point if the IM window state gets shuffled
if (imMayMove && (moveInputMethodWindowsIfNeededLocked(false) || toBeDisplayed)) {
    // Little hack here -- we -should- be able to rely on the
    // function to return true if the IME has moved and needs
    // its layer recomputed.  However, if the IME was hidden
    // and isn't actually moved in the list, its layer may be
    // out of data so we make sure to recompute it.
    assignLayersLocked(win.getWindowList());
}

if (wallpaperMayMove) {
    getDefaultDisplayContentLocked().pendingLayoutChanges |=
            WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER;
}

final DisplayContent displayContent = win.getDisplayContent();
if (displayContent != null) {
    displayContent.layoutNeeded = true;
}
win.mGivenInsetsPending = (flags&WindowManagerGlobal.RELAYOUT_INSETS_PENDING) != 0;
configChanged = updateOrientationFromAppTokensLocked(false);
performLayoutAndPlaceSurfacesLocked();
if (toBeDisplayed && win.mIsWallpaper) {
    DisplayInfo displayInfo = getDefaultDisplayInfoLocked();
    updateWallpaperOffsetLocked(win,
            displayInfo.logicalWidth, displayInfo.logicalHeight, false);
}
if (win.mAppToken != null) {
    win.mAppToken.updateReportedVisibilityLocked();
}
outFrame.set(win.mCompatFrame);
outOverscanInsets.set(win.mOverscanInsets);
outContentInsets.set(win.mContentInsets);
outVisibleInsets.set(win.mVisibleInsets);
outStableInsets.set(win.mStableInsets);
if (localLOGV) Slog.v(
    TAG, "Relayout given client " + client.asBinder()
    + ", requestedWidth=" + requestedWidth
    + ", requestedHeight=" + requestedHeight
    + ", viewVisibility=" + viewVisibility
    + "\nRelayout returning frame=" + outFrame
    + ", surface=" + outSurface);

if (localLOGV || DEBUG_FOCUS) Slog.v(
    TAG, "Relayout of " + win + ": focusMayChange=" + focusMayChange);

inTouchMode = mInTouchMode;
animating = mAnimator.mAnimating && win.mWinAnimator.isAnimating();
```

```
            if (animating && !mRelayoutWhileAnimating.contains(win)) {
                mRelayoutWhileAnimating.add(win);
            }

            mInputMonitor.updateInputWindowsLw(true /*force*/);

            if (DEBUG_LAYOUT) {
                Slog.v(TAG, "Relayout complete " + win + ": outFrame=" + outFrame.
                toShortString());
            }
        }

        if (configChanged) {
            sendNewConfiguration();
        }

        Binder.restoreCallingIdentity(origId);

        return (inTouchMode ? WindowManagerGlobal.RELAYOUT_RES_IN_TOUCH_MODE : 0)
                | (toBeDisplayed ? WindowManagerGlobal.RELAYOUT_RES_FIRST_TIME : 0)
                | (surfaceChanged ? WindowManagerGlobal.RELAYOUT_RES_SURFACE_CHANGED : 0)
                | (animating ? WindowManagerGlobal.RELAYOUT_RES_ANIMATING : 0);
    }
```

在上述代码中，参数 client 是一个 Binder 代理对象，引用了运行在应用程序进程这一侧中的一个 W 对象，用来标志一个 Activity 窗口。从前面 Android 应用程序窗口（Activity）与 WindowManagerService 服务的连接过程分析文中可以知道，在应用程序进程这一侧的每一个 W 对象，在 WindowManagerService 服务这一侧都有一个对应的 WindowState 对象，用来描述一个 Activity 窗口的状态。因此，WindowManagerService 类的成员函数 relayoutWindow 首先通过调用另外一个成员函数 windowForClientLocked 来获得与参数 client 所对应的一个 WindowState 对象 win，以便接下来可以对它进行操作。

在类 WindowManagerService 中，成员函数 relayoutWindow 实现窗口大小计算的过程如下所示。

- 参数 requestedWidth 和 requestedHeight 描述的是应用程序进程请求设置 Activity 窗口中的宽度和高度，它们会被记录在 WindowState 对象 win 的成员变量 mRequestedWidth 和 mRequestedHeight 中。
- WindowState 对象 win 的成员变量 mAttr，它指向的是一个 WindowManager.LayoutParams 对象，用来描述 Activity 窗口的布局参数。其中，这个 WindowManager.LayoutParams 对象的成员变量 width 和 height 是用来描述 Activity 窗口的宽度和高度。当这个 WindowManager.LayoutParams 对象的成员变量 flags 的 WindowManager.LayoutParams.FLAG_SCALED 位不等于 0 的时候，就说明需要给 Activity 窗口的大小设置缩放因子。缩放因子分为两个维度，分别是宽度缩放因子和高度缩放因子，保存在 WindowState 对象 win 的成员变量 HScale 和 VScale 中，计算方法分别是用应用程序进程请求设置 Activity 窗口中的宽度和高度除以 Activity 窗口在布局参数中所设置的宽度和高度。
- 参数 insetsPending 用来描述 Activity 窗口是否有额外的内容区域周边和可见区域周边未设置，它被记录在 WindowState 对象 win 的成员变量 mGivenInsetsPending 中。
- 调用 WindowManagerService 类的成员函数 performLayoutAndPlaceSurfacesLocked 来计算 Activity 窗口的大小。计算完成之后，参数 client 所描述的 Activity 窗口的大小、内容区域周边大小和可见区域边周边大小就会分别保存在 WindowState 对象 win 的成员变量 mFrame、mContentInsets 和 mVisibleInsets 中。
- 将 WindowState 对象 win 的成员变量 mFrame、mContentInsets 和 mVisibleInsets 的值分别复制到参数 outFrame、outContentInsets 和 outVisibleInsets 中，以便可以返回给应用程序进程。

接下来通过类 WindowManagerService 的成员函数 performLayoutAndPlaceSurfacesLocked、performLayoutAndPlaceSurfacesLockedLoop 和 performLayoutAndPlaceSurfacesLockedInner 进行进一步计算操作，具体实现代码如下所示：

```
    private final void performLayoutAndPlaceSurfacesLocked() {
```

```
    int loopCount = 6;
    do {
        mTraversalScheduled = false;
        performLayoutAndPlaceSurfacesLockedLoop();
        mH.removeMessages(H.DO_TRAVERSAL);
        loopCount--;
    } while (mTraversalScheduled && loopCount > 0);
    mInnerFields.mWallpaperActionPending = false;
}
private final void performLayoutAndPlaceSurfacesLockedLoop() {
    if (mInLayout) {
        if (DEBUG) {
            throw new RuntimeException("Recursive call!");
        }
        Slog.w(TAG, "performLayoutAndPlaceSurfacesLocked called while in layout.
Callers=" + Debug.getCallers(3));
        return;
    }

    if (mWaitingForConfig) {
        // Our configuration has changed (most likely rotation), but we
        // don't yet have the complete configuration to report to
        // applications.  Don't do any window layout until we have it.
        return;
    }

    if (!mDisplayReady) {
        // Not yet initialized, nothing to do.
        return;
    }

    Trace.traceBegin(Trace.TRACE_TAG_WINDOW_MANAGER, "wmLayout");
    mInLayout = true;
    boolean recoveringMemory = false;

    try {
        if (mForceRemoves != null) {
            recoveringMemory = true;
            // Wait a little bit for things to settle down, and off we go.
            for (int i=0; i<mForceRemoves.size(); i++) {
                WindowState ws = mForceRemoves.get(i);
                Slog.i(TAG, "Force removing: " + ws);
                removeWindowInnerLocked(ws.mSession, ws);
            }
            mForceRemoves = null;
            Slog.w(TAG, "Due to memory failure, waiting a bit for next layout");
            Object tmp = new Object();
            synchronized (tmp) {
                try {
                    tmp.wait(250);
                } catch (InterruptedException e) {
                }
            }
        }
    } catch (RuntimeException e) {
        Slog.wtf(TAG, "Unhandled exception while force removing for memory", e);
    }

    try {
        performLayoutAndPlaceSurfacesLockedInner(recoveringMemory);

        mInLayout = false;

        if (needsLayout()) {
            if (++mLayoutRepeatCount < 6) {
                requestTraversalLocked();
            } else {
                Slog.e(TAG, "Performed 6 layouts in a row. Skipping");
                mLayoutRepeatCount = 0;
            }
        } else {
```

```
            mLayoutRepeatCount = 0;
        }

        if (mWindowsChanged && !mWindowChangeListeners.isEmpty()) {
            mH.removeMessages(H.REPORT_WINDOWS_CHANGE);
            mH.sendEmptyMessage(H.REPORT_WINDOWS_CHANGE);
        }
    } catch (RuntimeException e) {
        mInLayout = false;
        Slog.wtf(TAG, "Unhandled exception while laying out windows", e);
    }

    Trace.traceEnd(Trace.TRACE_TAG_WINDOW_MANAGER);
}
private final void performLayoutAndPlaceSurfacesLockedInner(boolean recoveringMemory) {
    if (DEBUG_WINDOW_TRACE) {
        Slog.v(TAG, "performLayoutAndPlaceSurfacesLockedInner: entry. Called by "
                + Debug.getCallers(3));
    }

    final long currentTime = SystemClock.uptimeMillis();

    int i;
    boolean updateInputWindowsNeeded = false;

    if (mFocusMayChange) {
        mFocusMayChange = false;
        updateInputWindowsNeeded = updateFocusedWindowLocked(UPDATE_FOCUS_WILL_PLACE_
SURFACES, false /*updateInputWindows*/);
    }

    // Initialize state of exiting tokens.
    final int numDisplays = mDisplayContents.size();
    for (int displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {
        final DisplayContent displayContent = mDisplayContents.valueAt(displayNdx);
        for (i=displayContent.mExitingTokens.size()-1; i>=0; i--) {
            displayContent.mExitingTokens.get(i).hasVisible = false;
        }
    }

    for (int stackNdx = mStackIdToStack.size() - 1; stackNdx >= 0; --stackNdx) {
        // Initialize state of exiting applications.
        final AppTokenList exitingAppTokens =
                mStackIdToStack.valueAt(stackNdx).mExitingAppTokens;
        for (int tokenNdx = exitingAppTokens.size() - 1; tokenNdx >= 0; --tokenNdx) {
            exitingAppTokens.get(tokenNdx).hasVisible = false;
        }
    }

    mInnerFields.mHoldScreen = null;
    mInnerFields.mScreenBrightness = -1;
    mInnerFields.mButtonBrightness = -1;
    mInnerFields.mUserActivityTimeout = -1;
    mInnerFields.mObscureApplicationContentOnSecondaryDisplays = false;

    mTransactionSequence++;

    final DisplayContent defaultDisplay = getDefaultDisplayContentLocked();
    final DisplayInfo defaultInfo = defaultDisplay.getDisplayInfo();
    final int defaultDw = defaultInfo.logicalWidth;
    final int defaultDh = defaultInfo.logicalHeight;

    if (SHOW_LIGHT_TRANSACTIONS) Slog.i(TAG,
            ">>> OPEN TRANSACTION performLayoutAndPlaceSurfaces");
    SurfaceControl.openTransaction();
    try {

        if (mWatermark != null) {
            mWatermark.positionSurface(defaultDw, defaultDh);
        }
        if (mStrictModeFlash != null) {
```

```
            mStrictModeFlash.positionSurface(defaultDw, defaultDh);
        }
        if (mCircularDisplayMask != null) {
            mCircularDisplayMask.positionSurface(defaultDw, defaultDh, mRotation);
        }
        if (mEmulatorDisplayOverlay != null) {
            mEmulatorDisplayOverlay.positionSurface(defaultDw, defaultDh, mRotation);
        }

        boolean focusDisplayed = false;

        for (int displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {
            final DisplayContent displayContent = mDisplayContents.valueAt(displayNdx);
            boolean updateAllDrawn = false;
            WindowList windows = displayContent.getWindowList();
            DisplayInfo displayInfo = displayContent.getDisplayInfo();
            final int displayId = displayContent.getDisplayId();
            final int dw = displayInfo.logicalWidth;
            final int dh = displayInfo.logicalHeight;
            final int innerDw = displayInfo.appWidth;
            final int innerDh = displayInfo.appHeight;
            final boolean isDefaultDisplay = (displayId == Display.DEFAULT_DISPLAY);

            // Reset for each display.
            mInnerFields.mDisplayHasContent = false;
            mInnerFields.mPreferredRefreshRate = 0;

            int repeats = 0;
            do {
                repeats++;
                if (repeats > 6) {
                    Slog.w(TAG, "Animation repeat aborted after too many iterations");
                    displayContent.layoutNeeded = false;
                    break;
                }

                if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("On entry to LockedInner",
                    displayContent.pendingLayoutChanges);

                if ((displayContent.pendingLayoutChanges &
                        WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER) != 0 &&
                        (adjustWallpaperWindowsLocked() &
                                ADJUST_WALLPAPER_LAYERS_CHANGED) != 0) {
                    assignLayersLocked(windows);
                    displayContent.layoutNeeded = true;
                }

                if (isDefaultDisplay && (displayContent.pendingLayoutChanges
                        & WindowManagerPolicy.FINISH_LAYOUT_REDO_CONFIG) != 0) {
                    if (DEBUG_LAYOUT) Slog.v(TAG, "Computing new config from layout");
                    if (updateOrientationFromAppTokensLocked(true)) {
                        displayContent.layoutNeeded = true;
                        mH.sendEmptyMessage(H.SEND_NEW_CONFIGURATION);
                    }
                }

                if ((displayContent.pendingLayoutChanges
                        & WindowManagerPolicy.FINISH_LAYOUT_REDO_LAYOUT) != 0) {
                    displayContent.layoutNeeded = true;
                }

                // FIRST LOOP: Perform a layout, if needed.
                if (repeats < 4) {
                    performLayoutLockedInner(displayContent, repeats == 1,
                            false /*updateInputWindows*/);
                } else {
                    Slog.w(TAG, "Layout repeat skipped after too many iterations");
                }

                // FIRST AND ONE HALF LOOP: Make WindowManagerPolicy think
                // it is animating.
```

```
            displayContent.pendingLayoutChanges = 0;

            if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("loop number "
                    + mLayoutRepeatCount, displayContent.pendingLayoutChanges);

            if (isDefaultDisplay) {
                mPolicy.beginPostLayoutPolicyLw(dw, dh);
                for (i = windows.size() - 1; i >= 0; i--) {
                    WindowState w = windows.get(i);
                    if (w.mHasSurface) {
                        mPolicy.applyPostLayoutPolicyLw(w, w.mAttrs);
                    }
                }
                displayContent.pendingLayoutChanges |= mPolicy.finishPostLayout
                PolicyLw();
                if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats(
                    "after finishPostLayoutPolicyLw", displayContent.pendingLayout
                    Changes);
            }
        } while (displayContent.pendingLayoutChanges != 0);

        mInnerFields.mObscured = false;
        mInnerFields.mSyswin = false;
        displayContent.resetDimming();

        // Only used if default window
        final boolean someoneLosingFocus = !mLosingFocus.isEmpty();

        final int N = windows.size();
        for (i=N-1; i>=0; i--) {
            WindowState w = windows.get(i);
            final TaskStack stack = w.getStack();
            if (stack == null && w.getAttrs().type != TYPE_PRIVATE_PRESENTATION) {
                continue;
            }

            final boolean obscuredChanged = w.mObscured != mInnerFields.mObscured;

            // Update effect.
            w.mObscured = mInnerFields.mObscured;
            if (!mInnerFields.mObscured) {
                handleNotObscuredLocked(w, currentTime, innerDw, innerDh);
            }

            if (stack != null && !stack.testDimmingTag()) {
                handleFlagDimBehind(w);
            }

            if (isDefaultDisplay && obscuredChanged && (mWallpaperTarget == w)
                    && w.isVisibleLw()) {
                // This is the wallpaper target and its obscured state
                // changed... make sure the current wallaper's visibility
                // has been updated accordingly.
                updateWallpaperVisibilityLocked();
            }

            final WindowStateAnimator winAnimator = w.mWinAnimator;

            // If the window has moved due to its containing
            // content frame changing, then we'd like to animate
            // it.
            if (w.mHasSurface && w.shouldAnimateMove()) {
                // Frame has moved, containing content frame
                // has also moved, and we're not currently animating...
                // let's do something.
                Animation a = AnimationUtils.loadAnimation(mContext,
                        com.android.internal.R.anim.window_move_from_decor);
                winAnimator.setAnimation(a);
                winAnimator.mAnimDw = w.mLastFrame.left - w.mFrame.left;
                winAnimator.mAnimDh = w.mLastFrame.top - w.mFrame.top;
```

```
        //TODO (multidisplay): Accessibility supported only for the default display.
        if (mAccessibilityController != null
                && displayId == Display.DEFAULT_DISPLAY) {
            mAccessibilityController.onSomeWindowResizedOrMovedLocked();
        }

        try {
            w.mClient.moved(w.mFrame.left, w.mFrame.top);
        } catch (RemoteException e) {
        }
    }

    //Slog.i(TAG, "Window " + this + " clearing mContentChanged - done placing");
    w.mContentChanged = false;

    // Moved from updateWindowsAndWallpaperLocked().
    if (w.mHasSurface) {
        // Take care of the window being ready to display.
        final boolean committed =
                winAnimator.commitFinishDrawingLocked(currentTime);
        if (isDefaultDisplay && committed) {
            if (w.mAttrs.type == TYPE_DREAM) {
                // HACK: When a dream is shown, it may at that
                // point hide the lock screen.  So we need to
                // redo the layout to let the phone window manager
                // make this happen.
                displayContent.pendingLayoutChanges |=
                        WindowManagerPolicy.FINISH_LAYOUT_REDO_LAYOUT;
                if (DEBUG_LAYOUT_REPEATS) {
                    debugLayoutRepeats(
                        "dream and commitFinishDrawingLocked true",
                        displayContent.pendingLayoutChanges);
                }
            }
            if ((w.mAttrs.flags & FLAG_SHOW_WALLPAPER) != 0) {
                if (DEBUG_WALLPAPER_LIGHT) Slog.v(TAG,
                        "First draw done in potential wallpaper target " + w);
                mInnerFields.mWallpaperMayChange = true;
                displayContent.pendingLayoutChanges |=
                        WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER;
                if (DEBUG_LAYOUT_REPEATS) {
                    debugLayoutRepeats(
                        "wallpaper and commitFinishDrawingLocked true",
                        displayContent.pendingLayoutChanges);
                }
            }
        }

        winAnimator.setSurfaceBoundariesLocked(recoveringMemory);

        final AppWindowToken atoken = w.mAppToken;
        if (DEBUG_STARTING_WINDOW && atoken != null
                && w == atoken.startingWindow) {
            Slog.d(TAG, "updateWindows: starting " + w + " isOnScreen="
                + w.isOnScreen() + " allDrawn=" + atoken.allDrawn
                + " freezingScreen=" + atoken.mAppAnimator.freezingScreen);
        }
        if (atoken != null
                && (!atoken.allDrawn || atoken.mAppAnimator.freezingScreen)) {
            if (atoken.lastTransactionSequence != mTransactionSequence) {
                atoken.lastTransactionSequence = mTransactionSequence;
                atoken.numInterestingWindows = atoken.numDrawnWindows = 0;
                atoken.startingDisplayed = false;
            }
            if ((w.isOnScreen() || winAnimator.mAttrType == TYPE_BASE_
            APPLICATION)
                    && !w.mExiting && !w.mDestroying) {
                if (DEBUG_VISIBILITY || DEBUG_ORIENTATION) {
                    Slog.v(TAG, "Eval win " + w + ": isDrawn=" + w.isDrawnLw()
                            + ", isAnimating=" + winAnimator.isAnimating());
                    if (!w.isDrawnLw()) {
```

```
                                Slog.v(TAG, "Not displayed: s=" + winAnimator.mSurface Control
                                        + " pv=" + w.mPolicyVisibility
                                        + " mDrawState=" + winAnimator.mDrawState
                                        + " ah=" + w.mAttachedHidden
                                        + " th=" + atoken.hiddenRequested
                                        + " a=" + winAnimator.mAnimating);
                            }
                        }
                        if (w != atoken.startingWindow) {
                            if (!atoken.mAppAnimator.freezingScreen || !w.mAppFreezing) {
                                atoken.numInterestingWindows++;
                                if (w.isDrawnLw()) {
                                    atoken.numDrawnWindows++;
                                    if (DEBUG_VISIBILITY || DEBUG_ORIENTATION) Slog.v(TAG,
                                            "tokenMayBeDrawn: " + atoken
                                            + " freezingScreen=" + atoken.mAppAnimator.
                                            freezingScreen
                                            + " mAppFreezing=" + w.mAppFreezing);
                                    updateAllDrawn = true;
                                }
                            }
                        } else if (w.isDrawnLw()) {
                            atoken.startingDisplayed = true;
                        }
                    }
                }
            }

            if (isDefaultDisplay && someoneLosingFocus && (w == mCurrentFocus)
                    && w.isDisplayedLw()) {
                focusDisplayed = true;
            }

            updateResizingWindows(w);
        }

        mDisplayManagerInternal.setDisplayProperties(displayId,
                mInnerFields.mDisplayHasContent, mInnerFields.mPreferredRefreshRate,
                true /* inTraversal, must call performTraversalInTrans... below */);

        getDisplayContentLocked(displayId).stopDimmingIfNeeded();

        if (updateAllDrawn) {
            updateAllDrawnLocked(displayContent);
        }
    }

    if (focusDisplayed) {
        mH.sendEmptyMessage(H.REPORT_LOSING_FOCUS);
    }

    // Give the display manager a chance to adjust properties
    // like display rotation if it needs to.
    mDisplayManagerInternal.performTraversalInTransactionFromWindowManager();

} catch (RuntimeException e) {
    Slog.wtf(TAG, "Unhandled exception in Window Manager", e);
} finally {
    SurfaceControl.closeTransaction();
    if (SHOW_LIGHT_TRANSACTIONS) Slog.i(TAG,
            "<<< CLOSE TRANSACTION performLayoutAndPlaceSurfaces");
}

final WindowList defaultWindows = defaultDisplay.getWindowList();

// If we are ready to perform an app transition, check through
// all of the app tokens to be shown and see if they are ready
// to go.
if (mAppTransition.isReady()) {
    defaultDisplay.pendingLayoutChanges |= handleAppTransitionReadyLocked
    (defaultWindows);
```

```
        if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("after handleAppTransitionReady
        Locked",
                defaultDisplay.pendingLayoutChanges);
    }

    if (!mAnimator.mAnimating && mAppTransition.isRunning()) {
        // We have finished the animation of an app transition.  To do
        // this, we have delayed a lot of operations like showing and
        // hiding apps, moving apps in Z-order, etc.  The app token list
        // reflects the correct Z-order, but the window list may now
        // be out of sync with it.  So here we will just rebuild the
        // entire app window list.  Fun!
        defaultDisplay.pendingLayoutChanges |= handleAnimatingStoppedAndTransitionLocked();
        if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("after handleAnimStopAnd
        XitionLock",
            defaultDisplay.pendingLayoutChanges);
    }

    if (mInnerFields.mWallpaperForceHidingChanged && defaultDisplay.pending
    LayoutChanges == 0
            && !mAppTransition.isReady()) {
        // At this point, there was a window with a wallpaper that
        // was force hiding other windows behind it, but now it
        // is going away.  This may be simple -- just animate
        // away the wallpaper and its window -- or it may be
        // hard -- the wallpaper now needs to be shown behind
        // something that was hidden.
        defaultDisplay.pendingLayoutChanges |= WindowManagerPolicy.FINISH_LAYOUT_REDO_
        LAYOUT;
        if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("after animateAwayWallpaperLocked",
            defaultDisplay.pendingLayoutChanges);
    }
    mInnerFields.mWallpaperForceHidingChanged = false;

    if (mInnerFields.mWallpaperMayChange) {
        if (DEBUG_WALLPAPER_LIGHT) Slog.v(TAG, "Wallpaper may change!  Adjusting");
        defaultDisplay.pendingLayoutChanges |=
                WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER;
        if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("WallpaperMayChange",
                defaultDisplay.pendingLayoutChanges);
    }

    if (mFocusMayChange) {
        mFocusMayChange = false;
        if (updateFocusedWindowLocked(UPDATE_FOCUS_PLACING_SURFACES,
                false /*updateInputWindows*/)) {
            updateInputWindowsNeeded = true;
            defaultDisplay.pendingLayoutChanges |= WindowManagerPolicy.FINISH_LAYOUT_
            REDO_ANIM;
        }
    }

    if (needsLayout()) {
        defaultDisplay.pendingLayoutChanges |= WindowManagerPolicy.FINISH_LAYOUT_REDO_
        LAYOUT;
        if (DEBUG_LAYOUT_REPEATS) debugLayoutRepeats("mLayoutNeeded",
                defaultDisplay.pendingLayoutChanges);
    }

    for (i = mResizingWindows.size() - 1; i >= 0; i--) {
        WindowState win = mResizingWindows.get(i);
        if (win.mAppFreezing) {
            // Don't remove this window until rotation has completed.
            continue;
        }
        win.reportResized();
        mResizingWindows.remove(i);
    }

    if (DEBUG_ORIENTATION && mDisplayFrozen) Slog.v(TAG,
            "With display frozen, orientationChangeComplete="
```

```
            + mInnerFields.mOrientationChangeComplete);
    if (mInnerFields.mOrientationChangeComplete) {
        if (mWindowsFreezingScreen) {
            mWindowsFreezingScreen = false;
            mLastFinishedFreezeSource = mInnerFields.mLastWindowFreezeSource;
            mH.removeMessages(H.WINDOW_FREEZE_TIMEOUT);
        }
        stopFreezingDisplayLocked();
    }

    // Destroy the surface of any windows that are no longer visible.
    boolean wallpaperDestroyed = false;
    i = mDestroySurface.size();
    if (i > 0) {
        do {
            i--;
            WindowState win = mDestroySurface.get(i);
            win.mDestroying = false;
            if (mInputMethodWindow == win) {
                mInputMethodWindow = null;
            }
            if (win == mWallpaperTarget) {
                wallpaperDestroyed = true;
            }
            win.mWinAnimator.destroySurfaceLocked();
        } while (i > 0);
        mDestroySurface.clear();
    }

    // Time to remove any exiting tokens?
    for (int displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {
        final DisplayContent displayContent = mDisplayContents.valueAt(displayNdx);
        ArrayList<WindowToken> exitingTokens = displayContent.mExitingTokens;
        for (i = exitingTokens.size() - 1; i >= 0; i--) {
            WindowToken token = exitingTokens.get(i);
            if (!token.hasVisible) {
                exitingTokens.remove(i);
                if (token.windowType == TYPE_WALLPAPER) {
                    mWallpaperTokens.remove(token);
                }
            }
        }
    }

    // Time to remove any exiting applications?
    for (int stackNdx = mStackIdToStack.size() - 1; stackNdx >= 0; --stackNdx) {
        // Initialize state of exiting applications.
        final AppTokenList exitingAppTokens =
                mStackIdToStack.valueAt(stackNdx).mExitingAppTokens;
        for (i = exitingAppTokens.size() - 1; i >= 0; i--) {
            AppWindowToken token = exitingAppTokens.get(i);
            if (!token.hasVisible && !mClosingApps.contains(token) && !token.mDefer
            Removal) {
                // Make sure there is no animation running on this token,
                // so any windows associated with it will be removed as
                // soon as their animations are complete
                token.mAppAnimator.clearAnimation();
                token.mAppAnimator.animating = false;
                if (DEBUG_ADD_REMOVE || DEBUG_TOKEN_MOVEMENT) Slog.v(TAG,
                        "performLayout: App token exiting now removed" + token);
                removeAppFromTaskLocked(token);
                exitingAppTokens.remove(i);
            }
        }
    }

    if (!mAnimator.mAnimating && mRelayoutWhileAnimating.size() > 0) {
        for (int j=mRelayoutWhileAnimating.size()-1; j>=0; j--) {
            try {
                mRelayoutWhileAnimating.get(j).mClient.doneAnimating();
            } catch (RemoteException e) {
```

```
            }
        }
        mRelayoutWhileAnimating.clear();
    }

    if (wallpaperDestroyed) {
        defaultDisplay.pendingLayoutChanges |=
                WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER;
        defaultDisplay.layoutNeeded = true;
    }

    for (int displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {
        final DisplayContent displayContent = mDisplayContents.valueAt(displayNdx);
        if (displayContent.pendingLayoutChanges != 0) {
            displayContent.layoutNeeded = true;
        }
    }

    // Finally update all input windows now that the window changes have stabilized.
    mInputMonitor.updateInputWindowsLw(true /*force*/);

    setHoldScreenLocked(mInnerFields.mHoldScreen);
    if (!mDisplayFrozen) {
        if (mInnerFields.mScreenBrightness < 0 || mInnerFields.mScreenBrightness > 1.0f) {
            mPowerManagerInternal.setScreenBrightnessOverrideFromWindowManager(-1);
        } else {
            mPowerManagerInternal.setScreenBrightnessOverrideFromWindowManager(
                    toBrightnessOverride(mInnerFields.mScreenBrightness));
        }
        if (mInnerFields.mButtonBrightness < 0 || mInnerFields.mButtonBrightness > 1.0f) {
            mPowerManagerInternal.setButtonBrightnessOverrideFromWindowManager(-1);
        } else {
            mPowerManagerInternal.setButtonBrightnessOverrideFromWindowManager(
                    toBrightnessOverride(mInnerFields.mButtonBrightness));
        }
        mPowerManagerInternal.setUserActivityTimeoutOverrideFromWindowManager(
                mInnerFields.mUserActivityTimeout);
    }

    if (mTurnOnScreen) {
        if (DEBUG_VISIBILITY) Slog.v(TAG, "Turning screen on after layout!");
        mPowerManager.wakeUp(SystemClock.uptimeMillis());
        mTurnOnScreen = false;
    }

    if (mInnerFields.mUpdateRotation) {
        if (DEBUG_ORIENTATION) Slog.d(TAG, "Performing post-rotate rotation");
        if (updateRotationUncheckedLocked(false)) {
            mH.sendEmptyMessage(H.SEND_NEW_CONFIGURATION);
        } else {
            mInnerFields.mUpdateRotation = false;
        }
    }

    if (mWaitingForDrawnCallback != null ||
            (mInnerFields.mOrientationChangeComplete && !defaultDisplay.layoutNeeded &&
                    !mInnerFields.mUpdateRotation)) {
        checkDrawnWindowsLocked();
    }

    final int N = mPendingRemove.size();
    if (N.> 0) {
        if (mPendingRemoveTmp.length < N) {
            mPendingRemoveTmp = new WindowState[N+10];
        }
        mPendingRemove.toArray(mPendingRemoveTmp);
        mPendingRemove.clear();
        DisplayContentList displayList = new DisplayContentList();
        for (i = 0; i < N; i++) {
            WindowState w = mPendingRemoveTmp[i];
            removeWindowInnerLocked(w.mSession, w);
```

```
            final DisplayContent displayContent = w.getDisplayContent();
            if (displayContent != null && !displayList.contains(displayContent)) {
                displayList.add(displayContent);
            }
        }

        for (DisplayContent displayContent : displayList) {
            assignLayersLocked(displayContent.getWindowList());
            displayContent.layoutNeeded = true;
        }
    }

    // Remove all deferred displays stacks, tasks, and activities.
    for (int displayNdx = mDisplayContents.size() - 1; displayNdx >= 0; --displayNdx) {
        mDisplayContents.valueAt(displayNdx).checkForDeferredActions();
    }

    if (updateInputWindowsNeeded) {
        mInputMonitor.updateInputWindowsLw(false /*force*/);
    }
    setFocusedStackFrame();

    // Check to see if we are now in a state where the screen should
    // be enabled, because the window obscured flags have changed.
    enableScreenIfNeededLocked();

    scheduleAnimationLocked();

    if (DEBUG_WINDOW_TRACE) {
        Slog.e(TAG, "performLayoutAndPlaceSurfacesLockedInner exit: animating="
                + mAnimator.mAnimating);
    }
}
```

上述 3 个函数是前者调用后者的关系，在调用成员函数 performLayoutAndPlaceSurfacesLocked Inner 来刷新系统 UI 操作前后，函数 performLayoutAndPlaceSurfacesLocked 和 performLayoutAnd PlaceSurfacesLockedLoop 会执行如下所示的两个操作。

- 在调用前检查系统中是否存在强制删除的窗口。有内存不足的情况下，有一些窗口就会被回收，即要从系统中删除，这些窗口会保存在 WindowManagerService 类的成员变量 mForce Removes 所描述的一个 ArrayList 中。如果存在这些窗口，那么 WindowManagerService 类的成员函数 performLayoutAndPlaceSurfacesLocked 就会调用另外一个成员函数 removeWindowInnerLocked 来删除它们，以便可以回收它们所占用的内存。
- 在调用后检查系统中是否有窗口需要移除。如果有的话，那么 WindowManagerService 类的成员变量 mPendingRemove 所描述的一个 ArrayList 的大小就会大于 0。这种情况下，Window ManagerService 类的成员函数 performLayoutAndPlaceSurfacesLocked 就会调用另外一个成员函数 removeWindowInnerLocked 来移除这些窗口。注意，WindowManagerService 类的成员函数 removeWindowInnerLocked 只是用来移除窗口，但是并没有回收这些窗口所占用的内存。等到合适的时候，例如，内存不足时，才会考虑回收这些窗口所占用的内存。移除一个窗口的操作也是很复杂的，除了要将窗口从 WindowManagerService 类的相关成员变量中移除之外，还要考虑重新调整输入法窗口和壁纸窗口，因为被移除的窗口可能要求显示壁纸和输入法窗口，当它被移除之后，就要将壁纸窗口和输入法窗口调整到合适的 Z 轴位置上去，以便可以交给下一个需要显示壁纸和输入法窗口的窗口使用。此外，在移除了窗口之后，WindowManagerService 服务还需要重新计算现存的其他窗口的 Z 轴位置，以便可以正确地反映系统当前的 UI 状态，这是通过调用 WindowManagerService 类的成员函数 assignLayersLocked 来实现的。重新计算了现存的其他窗口的 Z 轴位置之后，又需要再次刷新系统的 UI，即要对 WindowManagerService 类的成员函数 performLayoutAndPlaceSurfacesLocked 进行递归调用，并且在调用前，将 WindowManagerService 类的成员变量 mLayoutNeeded 的值设置为 true。由此就可见，系统 UI 的刷新过程是非常复杂的。

成员函数 performLayoutAndPlaceSurfacesLockedInner 的代码非常巨大，实现了 Window

ManagerService 的核心功能，其基本的实现框架关键点如下所示。

- 在一个最多执行 7 次的 while 循环中，做两件事情：第一件事情是计算各个窗品的大小，这是通过调用另外一个成员函数 performLayoutLockedInner 来实现的；第二件事情是执行窗口的动画，主要是处理窗口的启动窗口显示动画和窗口切换过程中的动画，以及更新各个窗口的可见性。注意，每一次 while 循环执行之后，如果发现系统中的各个窗口的相应布局属性不再发生变化，那么就不行执行下一次的 while 循环，即该 while 循环可能不用执行 7 次就结束了。窗口的动画显示过程和窗口的可见性更新过程是相当复杂的，它们也是 WindowManagerService 服务最为核的地方，在后面将详细分析。
- 经过第 1 点的操作之后，接下来就可以将各个窗口的属性，例如，大小、位置等，通知 SurfaceFlinger 服务了，也就是让 SurfaceFlinger 服务更新它里面的各个 Layer 的属性值，以便可以对这些 Layer 执行可见性计算、合成等操作，最后渲染到硬件帧缓冲区中去。SurfaceFlinger 服务计算系统中各个窗口，即各个 Layer 的可见性，将它们合成、渲染到硬件帧缓冲区的过程可以参考前面 Android 系统 Surface 机制的 SurfaceFlinger 服务渲染应用程序 UI 的过程分析一文。注意，各个窗口的属性更新操作是被包含在 SurfaceFlinger 服务的一个事务中的，即一个 Transaction 中，这样做是为了避免每更新一个窗口的一个属性就触发 SurfaceFlinger 服务重新计算各个 Layer 的可见性，以及对各个 Layer 进行合并和渲染的操作。启动 SurfaceFlinger 服务的一个事务可以通过调用 Surface 类的静态成员函数 openTransaction 来实现，而关闭 SurfaceFlinger 服务的一个事务可以通过调用 Surface 类的静态成员函数 closeTransaction 来实现。
- 经过第 1 点和第 2 点的操作之后，一次系统 UI 的刷新过程就完成了，这时候就会将系统中的那些不会再显示的窗口的绘图表面销毁掉，并且将那些已经完成退出了的窗口令牌，即将我们在前面 Android 应用程序窗口（Activity）与 WindowManagerService 服务的连接过程分析中所提到的 WindowToken 移除掉，以及将那些已经退出了的 Activity 窗口令牌，即将我们在前面 Android 应用程序窗口（Activity）与 WindowManagerService 服务的连接过程分析中所提到的 AppWindowToken 也移除掉。这一步实际执行的是窗口清理操作。

上述 3 个操作是函数 performLayoutAndPlaceSurfacesLockedInner 的核心，理解上述 3 点后，基本上也就可以理解 WindowManagerService 服务刷新系统 UI 的具体过程了。

再看函数 performLayoutLockedInner，具体执行过程如下所示。

- 准备阶段：调用 PhoneWindowManager 类的成员函数 beginLayoutLw 来设置屏幕的大小。屏幕的大小可以通过调用 WindowManagerService 类的成员变量 mDisplay 所描述的一个 Display 对象的成员函数 getWidth 和 getHeight 来获得。
- 计算阶段：调用 PhoneWindowManager 类的成员函数 layoutWindowLw 来计算各个窗口的大小、内容区域周边大小以及可见区域周边大小。
- 结束阶段：调用 PhoneWindowManager 类的成员函数 finishLayoutLw 来执行一些清理工作。

函数 performLayoutLockedInner 的具体实现代码如下所示：

```
private final void performLayoutLockedInner(final DisplayContent displayContent,
                        boolean initial, boolean updateInputWindows) {
    if (!displayContent.layoutNeeded) {
        return;
    }
    displayContent.layoutNeeded = false;
    WindowList windows = displayContent.getWindowList();
    boolean isDefaultDisplay = displayContent.isDefaultDisplay;

    DisplayInfo displayInfo = displayContent.getDisplayInfo();
    final int dw = displayInfo.logicalWidth;
    final int dh = displayInfo.logicalHeight;

    final int NFW = mFakeWindows.size();
    for (int i=0; i<NFW; i++) {
        mFakeWindows.get(i).layout(dw, dh);
    }
```

```
final int N = windows.size();
int i;

if (DEBUG_LAYOUT) {
    Slog.v(TAG, "-------------------------------------");
    Slog.v(TAG, "performLayout: needed="
            + displayContent.layoutNeeded + " dw=" + dw + " dh=" + dh);
}

WindowStateAnimator universeBackground = null;

mPolicy.beginLayoutLw(isDefaultDisplay, dw, dh, mRotation);
if (isDefaultDisplay) {
    // Not needed on non-default displays.
    mSystemDecorLayer = mPolicy.getSystemDecorLayerLw();
    mScreenRect.set(0, 0, dw, dh);
}

mPolicy.getContentRectLw(mTmpContentRect);
displayContent.resize(mTmpContentRect);

int seq = mLayoutSeq+1;
if (seq < 0) seq = 0;
mLayoutSeq = seq;

boolean behindDream = false;

// First perform layout of any root windows (not attached
// to another window).
int topAttached = -1;
for (i = N-1; i >= 0; i--) {
    final WindowState win = windows.get(i);

    // Don't do layout of a window if it is not visible, or
    // soon won't be visible, to avoid wasting time and funky
    // changes while a window is animating away.
    final boolean gone = (behindDream && mPolicy.canBeForceHidden(win, win.mAttrs))
            || win.isGoneForLayoutLw();

    if (DEBUG_LAYOUT && !win.mLayoutAttached) {
        Slog.v(TAG, "1ST PASS " + win
                + ": gone=" + gone + " mHaveFrame=" + win.mHaveFrame
                + " mLayoutAttached=" + win.mLayoutAttached
                + " screen changed=" + win.isConfigChanged());
        final AppWindowToken atoken = win.mAppToken;
        if (gone) Slog.v(TAG, "  GONE: mViewVisibility="
                + win.mViewVisibility + " mRelayoutCalled="
                + win.mRelayoutCalled + " hidden="
                + win.mRootToken.hidden + " hiddenRequested="
                + (atoken != null && atoken.hiddenRequested)
                + " mAttachedHidden=" + win.mAttachedHidden);
        else Slog.v(TAG, "  VIS: mViewVisibility="
                + win.mViewVisibility + " mRelayoutCalled="
                + win.mRelayoutCalled + " hidden="
                + win.mRootToken.hidden + " hiddenRequested="
                + (atoken != null && atoken.hiddenRequested)
                + " mAttachedHidden=" + win.mAttachedHidden);
    }

    // If this view is GONE, then skip it -- keep the current
    // frame, and let the caller know so they can ignore it
    // if they want.  (We do the normal layout for INVISIBLE
    // windows, since that means "perform layout as normal,
    // just don't display").
    if (!gone || !win.mHaveFrame || win.mLayoutNeeded
            || ((win.isConfigChanged() || win.setInsetsChanged()) &&
                    ((win.mAttrs.privateFlags & PRIVATE_FLAG_KEYGUARD) != 0 ||
                    win.mAppToken != null && win.mAppToken.layoutConfigChanges))
            || win.mAttrs.type == TYPE_UNIVERSE_BACKGROUND) {
        if (!win.mLayoutAttached) {
            if (initial) {
```

```
                //Slog.i(TAG, "Window " + this + " clearing mContentChanged - initial");
                win.mContentChanged = false;
            }
            if (win.mAttrs.type == TYPE_DREAM) {
                // Don't layout windows behind a dream, so that if it
                // does stuff like hide the status bar we won't get a
                // bad transition when it goes away.
                behindDream = true;
            }
            win.mLayoutNeeded = false;
            win.prelayout();
            mPolicy.layoutWindowLw(win, null);
            win.mLayoutSeq = seq;
            if (DEBUG_LAYOUT) Slog.v(TAG, "  LAYOUT: mFrame="
                    + win.mFrame + " mContainingFrame="
                    + win.mContainingFrame + " mDisplayFrame="
                    + win.mDisplayFrame);
        } else {
            if (topAttached < 0) topAttached = i;
        }
    }
    if (win.mViewVisibility == View.VISIBLE
            && win.mAttrs.type == TYPE_UNIVERSE_BACKGROUND
            && universeBackground == null) {
        universeBackground = win.mWinAnimator;
    }
}

if (mAnimator.mUniverseBackground  != universeBackground) {
    mFocusMayChange = true;
    mAnimator.mUniverseBackground = universeBackground;
}

boolean attachedBehindDream = false;

// Now perform layout of attached windows, which usually
// depend on the position of the window they are attached to.
// XXX does not deal with windows that are attached to windows
// that are themselves attached.
for (i = topAttached; i >= 0; i--) {
    final WindowState win = windows.get(i);

    if (win.mLayoutAttached) {
        if (DEBUG_LAYOUT) Slog.v(TAG, "2ND PASS " + win
                + " mHaveFrame=" + win.mHaveFrame
                + " mViewVisibility=" + win.mViewVisibility
                + " mRelayoutCalled=" + win.mRelayoutCalled);
        // If this view is GONE, then skip it -- keep the current
        // frame, and let the caller know so they can ignore it
        // if they want.  (We do the normal layout for INVISIBLE
        // windows, since that means "perform layout as normal,
        // just don't display").
        if (attachedBehindDream && mPolicy.canBeForceHidden(win, win.mAttrs)) {
            continue;
        }
        if ((win.mViewVisibility != View.GONE && win.mRelayoutCalled)
                || !win.mHaveFrame || win.mLayoutNeeded) {
            if (initial) {
                //Slog.i(TAG, "Window " + this + " clearing mContentChanged - initial");
                    win.mContentChanged = false;
            }
            win.mLayoutNeeded = false;
            win.prelayout();
            mPolicy.layoutWindowLw(win, win.mAttachedWindow);
            win.mLayoutSeq = seq;
            if (DEBUG_LAYOUT) Slog.v(TAG, "  LAYOUT: mFrame="
                    + win.mFrame + " mContainingFrame="
                    + win.mContainingFrame + " mDisplayFrame="
                    + win.mDisplayFrame);
        }
    } else if (win.mAttrs.type == TYPE_DREAM) {
```

```
            // Don't layout windows behind a dream, so that if it
            // does stuff like hide the status bar we won't get a
            // bad transition when it goes away.
            attachedBehindDream = behindDream;
        }
    }

    // Window frames may have changed.  Tell the input dispatcher about it.
    mInputMonitor.setUpdateInputWindowsNeededLw();
    if (updateInputWindows) {
        mInputMonitor.updateInputWindowsLw(false /*force*/);
    }

    mPolicy.finishLayoutLw();
}
```

15.2.4　拦截消息的处理类

拦截消息处理类的实现文件是 frameworks\base\policy\src\com\android\internal\policy\impl\PhoneWindowManager.java，其成员函数 beginLayoutLw、layoutWindowLw 和 finishLayoutLw 实现了 Activity 窗口的大小计算操作。定义拦截消息类的代码如下所示：

```
public class PhoneWindowManager implements WindowManagerPolicy {
    static final String TAG = "WindowManager";
    static final boolean DEBUG = false;
    static final boolean localLOGV = false;
    static final boolean DEBUG_LAYOUT = false;
    static final boolean DEBUG_INPUT = false;
    static final boolean DEBUG_STARTING_WINDOW = false;
    static final boolean DEBUG_WAKEUP = false;
    static final boolean SHOW_STARTING_ANIMATIONS = true;
    static final boolean SHOW_PROCESSES_ON_ALT_MENU = false;
……
    static final Rect mTmpParentFrame = new Rect();
    static final Rect mTmpDisplayFrame = new Rect();
    static final Rect mTmpOverscanFrame = new Rect();
    static final Rect mTmpContentFrame = new Rect();
    static final Rect mTmpVisibleFrame = new Rect();
    static final Rect mTmpDecorFrame = new Rect();
    static final Rect mTmpStableFrame = new Rect();
    static final Rect mTmpNavigationFrame = new Rect();
……
```

在类 PhoneWindowManager 中定义了如下 5 组成员变量。

- 第一组成员变量：mW 和 mH，分别用来描述当前这轮窗口大小计算过程的屏幕宽度和高度。
- 第二组成员变量：mCurLeft、mCurTop、mCurRight 和 mCurBottom，组成了一个四元组（mCurLeft, mCurTop, mCurRight, mCurBottom），用来描述当前这轮窗口大小计算过程的屏幕装饰区，它对应于前面所提到的 Activity 窗口的可见区域周边。
- 第三组成员变量：mContentLeft、mContentTop、mContentRight 和 mContentBottom，组成了一个四元组（mContentLeft, mContentTop, mContentRight, mContentBottom），也是用来描述当前这轮窗口大小计算过程的屏幕装饰区，不过它对应的是前面所提到的 Activity 窗口的内容区域周边。
- 第四组成员变量：mDockLeft、mDockTop、mDockRight、mDockBottom 和 mDockLayer，其中，前 4 个成员变量组成一个四元组（mDockLeft, mDockTop, mDockRight, mDockBottom），用来描述当前这轮窗口大小计算过程中的输入法窗口所占据的位置，后一个成员变量 mDockLayer 用来描述输入法窗品的 Z 轴位置。
- 第五组成员变量：mTmpParentFrame、mTmpDisplayFrame、mTmpContentFrame 和 mTmpVisibleFrame，这是一组临时 Rect 区域，用来作为参数传递给具体的窗口计算大小的，避免每次都创建一组新的 Rect 区域来作来参数传递窗口。

除了上述 5 组成员变量之外，类 PhoneWindowManager 还有一个类型为 WindowState 的成员变量 mStatusBar，其功能是描述系统的状态栏。

在类 PhoneWindowManager 中，函数 beginLayoutLw 的功能如下所示。

- 初始化上面提到的 4 组成员变量，其中，mW 和 mH 设置为参数 displayWidth 和 displayHeight 所指定的屏幕宽度和高度，并且使得（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这 3 个区域的大小等于屏幕的大小。
- 计算状态栏的大小。状态栏的大小一经确定，并且它是可见的，那么就会修改成员变量 mCurLeft、mContentLeft 和 mDockLeft 的值为状态栏的所占据的区域的下边界位置，这样就可以将（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这 3 个区域限制为剔除状态栏区域之后所得到的屏幕区域。

函数 beginLayoutLw 的具体实现代码如下所示：

```
public void beginLayoutLw(boolean isDefaultDisplay, int displayWidth, int displayHeight,
                    int displayRotation) {
    final int overscanLeft, overscanTop, overscanRight, overscanBottom;
    if (isDefaultDisplay) {
        switch (displayRotation) {
            case Surface.ROTATION_90:
                overscanLeft = mOverscanTop;
                overscanTop = mOverscanRight;
                overscanRight = mOverscanBottom;
                overscanBottom = mOverscanLeft;
                break;
            case Surface.ROTATION_180:
                overscanLeft = mOverscanRight;
                overscanTop = mOverscanBottom;
                overscanRight = mOverscanLeft;
                overscanBottom = mOverscanTop;
                break;
            case Surface.ROTATION_270:
                overscanLeft = mOverscanBottom;
                overscanTop = mOverscanLeft;
                overscanRight = mOverscanTop;
                overscanBottom = mOverscanRight;
                break;
            default:
                overscanLeft = mOverscanLeft;
                overscanTop = mOverscanTop;
                overscanRight = mOverscanRight;
                overscanBottom = mOverscanBottom;
                break;
        }
    } else {
        overscanLeft = 0;
        overscanTop = 0;
        overscanRight = 0;
        overscanBottom = 0;
    }
    mOverscanScreenLeft = mRestrictedOverscanScreenLeft = 0;
    mOverscanScreenTop = mRestrictedOverscanScreenTop = 0;
    mOverscanScreenWidth = mRestrictedOverscanScreenWidth = displayWidth;
    mOverscanScreenHeight = mRestrictedOverscanScreenHeight = displayHeight;
    mSystemLeft = 0;
    mSystemTop = 0;
    mSystemRight = displayWidth;
    mSystemBottom = displayHeight;
    mUnrestrictedScreenLeft = overscanLeft;
    mUnrestrictedScreenTop = overscanTop;
    mUnrestrictedScreenWidth = displayWidth - overscanLeft - overscanRight;
    mUnrestrictedScreenHeight = displayHeight - overscanTop - overscanBottom;
    mRestrictedScreenLeft = mUnrestrictedScreenLeft;
    mRestrictedScreenTop = mUnrestrictedScreenTop;
    mRestrictedScreenWidth = mSystemGestures.screenWidth = mUnrestrictedScreenWidth;
    mRestrictedScreenHeight = mSystemGestures.screenHeight = mUnrestrictedScreenHeight;
    mDockLeft = mContentLeft = mVoiceContentLeft = mStableLeft = mStableFullscreenLeft
            = mCurLeft = mUnrestrictedScreenLeft;
```

```
    mDockTop = mContentTop = mVoiceContentTop = mStableTop = mStableFullscreenTop
            = mCurTop = mUnrestrictedScreenTop;
    mDockRight = mContentRight = mVoiceContentRight = mStableRight = mStable
FullscreenRight
            = mCurRight = displayWidth - overscanRight;
    mDockBottom = mContentBottom = mVoiceContentBottom = mStableBottom = mStable
FullscreenBottom
            = mCurBottom = displayHeight - overscanBottom;
    mDockLayer = 0x10000000;
    mStatusBarLayer = -1;

    // start with the current dock rect, which will be (0,0,displayWidth,displayHeight)
    final Rect pf = mTmpParentFrame;
    final Rect df = mTmpDisplayFrame;
    final Rect of = mTmpOverscanFrame;
    final Rect vf = mTmpVisibleFrame;
    final Rect dcf = mTmpDecorFrame;
    pf.left = df.left = of.left = vf.left = mDockLeft;
    pf.top = df.top = of.top = vf.top = mDockTop;
    pf.right = df.right = of.right = vf.right = mDockRight;
    pf.bottom = df.bottom = of.bottom = vf.bottom = mDockBottom;
    dcf.setEmpty();  // Decor frame N/A for system bars.

    if (isDefaultDisplay) {
        // For purposes of putting out fake window up to steal focus, we will
        // drive nav being hidden only by whether it is requested.
        final int sysui = mLastSystemUiFlags;
        boolean navVisible = (sysui & View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) == 0;
        boolean navTranslucent = (sysui
                & (View.NAVIGATION_BAR_TRANSLUCENT | View.SYSTEM_UI_TRANSPARENT)) != 0;
        boolean immersive = (sysui & View.SYSTEM_UI_FLAG_IMMERSIVE) != 0;
        boolean immersiveSticky = (sysui & View.SYSTEM_UI_FLAG_IMMERSIVE_STICKY) != 0;
        boolean navAllowedHidden = immersive || immersiveSticky;
        navTranslucent &= !immersiveSticky;  // transient trumps translucent
        boolean isKeyguardShowing = isStatusBarKeyguard() && !mHideLockScreen;
        if (!isKeyguardShowing) {
            navTranslucent &= areTranslucentBarsAllowed();
        }

        // When the navigation bar isn't visible, we put up a fake
        // input window to catch all touch events.  This way we can
        // detect when the user presses anywhere to bring back the nav
        // bar and ensure the application doesn't see the event.
        if (navVisible || navAllowedHidden) {
            if (mHideNavFakeWindow != null) {
                mHideNavFakeWindow.dismiss();
                mHideNavFakeWindow = null;
            }
        } else if (mHideNavFakeWindow == null) {
            mHideNavFakeWindow = mWindowManagerFuncs.addFakeWindow(
                    mHandler.getLooper(), mHideNavInputEventReceiverFactory,
                    "hidden nav", WindowManager.LayoutParams.TYPE_HIDDEN_NAV_CONSUMER, 0,
                    0, false, false, true);
        }

        // For purposes of positioning and showing the nav bar, if we have
        // decided that it can't be hidden (because of the screen aspect ratio),
        // then take that into account.
        navVisible |= !canHideNavigationBar();

        boolean updateSysUiVisibility = false;
        if (mNavigationBar != null) {
            boolean transientNavBarShowing = mNavigationBarController.isTransientShowing();
            // Force the navigation bar to its appropriate place and
            // size.  We need to do this directly, instead of relying on
            // it to bubble up from the nav bar, because this needs to
            // change atomically with screen rotations.
            mNavigationBarOnBottom = (!mNavigationBarCanMove || displayWidth <
            displayHeight);
            if (mNavigationBarOnBottom) {
                // It's a system nav bar or a portrait screen; nav bar goes on bottom.
```

```
            int top = displayHeight - overscanBottom
                    - mNavigationBarHeightForRotation[displayRotation];
            mTmpNavigationFrame.set(0, top, displayWidth, displayHeight -
            overscanBottom);
            mStableBottom = mStableFullscreenBottom = mTmpNavigationFrame.top;
            if (transientNavBarShowing) {
                mNavigationBarController.setBarShowingLw(true);
            } else if (navVisible) {
                mNavigationBarController.setBarShowingLw(true);
                mDockBottom = mTmpNavigationFrame.top;
                mRestrictedScreenHeight = mDockBottom - mRestrictedScreenTop;
                mRestrictedOverscanScreenHeight = mDockBottom - mRestrictedOverscan
                ScreenTop;
            } else {
                // We currently want to hide the navigation UI.
                mNavigationBarController.setBarShowingLw(false);
            }
            if (navVisible && !navTranslucent && !navAllowedHidden
                    && !mNavigationBar.isAnimatingLw()
                    && !mNavigationBarController.wasRecentlyTranslucent()) {
                // If the opaque nav bar is currently requested to be visible,
                // and not in the process of animating on or off, then
                // we can tell the app that it is covered by it.
                mSystemBottom = mTmpNavigationFrame.top;
            }
        } else {
            // Landscape screen; nav bar goes to the right.
            int left = displayWidth - overscanRight
                    - mNavigationBarWidthForRotation[displayRotation];
            mTmpNavigationFrame.set(left, 0, displayWidth - overscanRight,
            displayHeight);
            mStableRight = mStableFullscreenRight = mTmpNavigationFrame.left;
            if (transientNavBarShowing) {
                mNavigationBarController.setBarShowingLw(true);
            } else if (navVisible) {
                mNavigationBarController.setBarShowingLw(true);
                mDockRight = mTmpNavigationFrame.left;
                mRestrictedScreenWidth = mDockRight - mRestrictedScreenLeft;
                mRestrictedOverscanScreenWidth = mDockRight - mRestrictedOverscan
                ScreenLeft;
            } else {
                // We currently want to hide the navigation UI.
                mNavigationBarController.setBarShowingLw(false);
            }
            if (navVisible && !navTranslucent && !mNavigationBar.isAnimatingLw()
                    && !mNavigationBarController.wasRecentlyTranslucent()) {
                // If the nav bar is currently requested to be visible,
                // and not in the process of animating on or off, then
                // we can tell the app that it is covered by it.
                mSystemRight = mTmpNavigationFrame.left;
            }
        }
        // Make sure the content and current rectangles are updated to
        // account for the restrictions from the navigation bar.
        mContentTop = mVoiceContentTop = mCurTop = mDockTop;
        mContentBottom = mVoiceContentBottom = mCurBottom = mDockBottom;
        mContentLeft = mVoiceContentLeft = mCurLeft = mDockLeft;
        mContentRight = mVoiceContentRight = mCurRight = mDockRight;
        mStatusBarLayer = mNavigationBar.getSurfaceLayer();
        // And compute the final frame.
        mNavigationBar.computeFrameLw(mTmpNavigationFrame, mTmpNavigationFrame,
                mTmpNavigationFrame, mTmpNavigationFrame, mTmpNavigationFrame, dcf,
                mTmpNavigationFrame);
        if (DEBUG_LAYOUT) Slog.i(TAG, "mNavigationBar frame: " + mTmpNavigationFrame);
        if (mNavigationBarController.checkHiddenLw()) {
            updateSysUiVisibility = true;
        }
    }
    if (DEBUG_LAYOUT) Slog.i(TAG, String.format("mDock rect: (%d,%d - %d,%d)",
            mDockLeft, mDockTop, mDockRight, mDockBottom));
```

```
        // decide where the status bar goes ahead of time
        if (mStatusBar != null) {
            // apply any navigation bar insets
            pf.left = df.left = of.left = mUnrestrictedScreenLeft;
            pf.top = df.top = of.top = mUnrestrictedScreenTop;
            pf.right = df.right = of.right = mUnrestrictedScreenWidth + mUnrestricted
            ScreenLeft;
            pf.bottom = df.bottom = of.bottom = mUnrestrictedScreenHeight
                    + mUnrestrictedScreenTop;
            vf.left = mStableLeft;
            vf.top = mStableTop;
            vf.right = mStableRight;
            vf.bottom = mStableBottom;

            mStatusBarLayer = mStatusBar.getSurfaceLayer();

            // Let the status bar determine its size.
            mStatusBar.computeFrameLw(pf, df, vf, vf, vf, dcf, vf);

            // For layout, the status bar is always at the top with our fixed height.
           mStableTop = mUnrestrictedScreenTop + mStatusBarHeight;

           boolean statusBarTransient = (sysui & View.STATUS_BAR_TRANSIENT) != 0;
           boolean statusBarTranslucent = (sysui
                   & (View.STATUS_BAR_TRANSLUCENT | View.SYSTEM_UI_TRANSPARENT)) != 0;
           if (!isKeyguardShowing) {
               statusBarTranslucent &= areTranslucentBarsAllowed();
           }

           // If the status bar is hidden, we don't want to cause
           // windows behind it to scroll.
           if (mStatusBar.isVisibleLw() && !statusBarTransient) {
               // Status bar may go away, so the screen area it occupies
               // is available to apps but just covering them when the
               // status bar is visible.
               mDockTop = mUnrestrictedScreenTop + mStatusBarHeight;

               mContentTop = mVoiceContentTop = mCurTop = mDockTop;
               mContentBottom = mVoiceContentBottom = mCurBottom = mDockBottom;
               mContentLeft = mVoiceContentLeft = mCurLeft = mDockLeft;
               mContentRight = mVoiceContentRight = mCurRight = mDockRight;

               if (DEBUG_LAYOUT) Slog.v(TAG, "Status bar: " +
                   String.format(
                       "dock=[%d,%d][%d,%d] content=[%d,%d][%d,%d] cur=[%d,%d][%d,%d]",
                       mDockLeft, mDockTop, mDockRight, mDockBottom,
                       mContentLeft, mContentTop, mContentRight, mContentBottom,
                       mCurLeft, mCurTop, mCurRight, mCurBottom));
           }
           if (mStatusBar.isVisibleLw() && !mStatusBar.isAnimatingLw()
                   && !statusBarTransient && !statusBarTranslucent
                   && !mStatusBarController.wasRecentlyTranslucent()) {
               // If the opaque status bar is currently requested to be visible,
               // and not in the process of animating on or off, then
               // we can tell the app that it is covered by it.
               mSystemTop = mUnrestrictedScreenTop + mStatusBarHeight;
           }
           if (mStatusBarController.checkHiddenLw()) {
               updateSysUiVisibility = true;
           }
       }
       if (updateSysUiVisibility) {
           updateSystemUiVisibilityLw();
       }
   }
}
```

执行完上述函数 beginLayoutLw 后，会返回到类 WindowManagerService 的成员函数 performLayoutLockedInner 中，然后会调用类 PhoneWindowManager 的成员函数 layoutWindowLw 来计算系统中各个可见窗口的大小。函数 layoutWindowLw 的具体实现代码如下所示：

```
public void layoutWindowLw(WindowState win, WindowState attached) {
    // we've already done the status bar
    final WindowManager.LayoutParams attrs = win.getAttrs();
    if ((win == mStatusBar && (attrs.privateFlags & PRIVATE_FLAG_KEYGUARD) == 0) ||
            win == mNavigationBar) {
        return;
    }
    final boolean isDefaultDisplay = win.isDefaultDisplay();
    final boolean needsToOffsetInputMethodTarget = isDefaultDisplay &&
            (win == mLastInputMethodTargetWindow && mLastInputMethodWindow != null);
    if (needsToOffsetInputMethodTarget) {
        if (DEBUG_LAYOUT) Slog.i(TAG, "Offset ime target window by the last ime window
state");
        offsetInputMethodWindowLw(mLastInputMethodWindow);
    }

    final int fl = PolicyControl.getWindowFlags(win, attrs);
    final int sim = attrs.softInputMode;
    final int sysUiFl = PolicyControl.getSystemUiVisibility(win, null);

    final Rect pf = mTmpParentFrame;
    final Rect df = mTmpDisplayFrame;
    final Rect of = mTmpOverscanFrame;
    final Rect cf = mTmpContentFrame;
    final Rect vf = mTmpVisibleFrame;
    final Rect dcf = mTmpDecorFrame;
    final Rect sf = mTmpStableFrame;
    dcf.setEmpty();

    final boolean hasNavBar = (isDefaultDisplay && mHasNavigationBar
            && mNavigationBar != null && mNavigationBar.isVisibleLw());

    final int adjust = sim & SOFT_INPUT_MASK_ADJUST;

    if (isDefaultDisplay) {
        sf.set(mStableLeft, mStableTop, mStableRight, mStableBottom);
    } else {
        sf.set(mOverscanLeft, mOverscanTop, mOverscanRight, mOverscanBottom);
    }

    if (!isDefaultDisplay) {
        if (attached != null) {
            // If this window is attached to another, our display
            // frame is the same as the one we are attached to.
            setAttachedWindowFrames(win, fl, adjust, attached, true, pf, df, of, cf, vf);
        } else {
            // Give the window full screen.
            pf.left = df.left = of.left = cf.left = mOverscanScreenLeft;
            pf.top = df.top = of.top = cf.top = mOverscanScreenTop;
            pf.right = df.right = of.right = cf.right
                    = mOverscanScreenLeft + mOverscanScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom
                    = mOverscanScreenTop + mOverscanScreenHeight;
        }
    } else  if (attrs.type == TYPE_INPUT_METHOD) {
        pf.left = df.left = of.left = cf.left = vf.left = mDockLeft;
        pf.top = df.top = of.top = cf.top = vf.top = mDockTop;
        pf.right = df.right = of.right = cf.right = vf.right = mDockRight;
        // IM dock windows layout below the nav bar...
        pf.bottom = df.bottom = of.bottom = mUnrestrictedScreenTop +
        mUnrestrictedScreenHeight;
        // ...with content insets above the nav bar
        cf.bottom = vf.bottom = mStableBottom;
        // IM dock windows always go to the bottom of the screen.
        attrs.gravity = Gravity.BOTTOM;
        mDockLayer = win.getSurfaceLayer();
    } else if (win == mStatusBar && (attrs.privateFlags & PRIVATE_FLAG_KEYGUARD) != 0)
{
        pf.left = df.left = of.left = mUnrestrictedScreenLeft;
        pf.top = df.top = of.top = mUnrestrictedScreenTop;
        pf.right = df.right = of.right = mUnrestrictedScreenWidth +
```

```
        mUnrestrictedScreenLeft;
        pf.bottom = df.bottom = of.bottom = mUnrestrictedScreenHeight +
        mUnrestrictedScreenTop;
        cf.left = vf.left = mStableLeft;
        cf.top = vf.top = mStableTop;
        cf.right = vf.right = mStableRight;
        vf.bottom = mStableBottom;
        cf.bottom = mContentBottom;
    } else {

        // Default policy decor for the default display
        dcf.left = mSystemLeft;
        dcf.top = mSystemTop;
        dcf.right = mSystemRight;
        dcf.bottom = mSystemBottom;
        final boolean inheritTranslucentDecor = (attrs.privateFlags
                & WindowManager.LayoutParams.PRIVATE_FLAG_INHERIT_TRANSLUCENT_DECOR) != 0;
        final boolean isAppWindow =
                attrs.type >= WindowManager.LayoutParams.FIRST_APPLICATION_WINDOW &&
                attrs.type <= WindowManager.LayoutParams.LAST_APPLICATION_WINDOW;
        final boolean topAtRest =
                win == mTopFullscreenOpaqueWindowState && !win.isAnimatingLw();
        if (isAppWindow && !inheritTranslucentDecor && !topAtRest) {
            if ((sysUiFl & View.SYSTEM_UI_FLAG_FULLSCREEN) == 0
                    && (fl & WindowManager.LayoutParams.FLAG_FULLSCREEN) == 0
                    && (fl & WindowManager.LayoutParams.FLAG_TRANSLUCENT_STATUS) == 0
                    && (fl & WindowManager.LayoutParams.
                            FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS) == 0) {
                // Ensure policy decor includes status bar
                dcf.top = mStableTop;
            }
            if ((fl & WindowManager.LayoutParams.FLAG_TRANSLUCENT_NAVIGATION) == 0
                    && (sysUiFl & View.SYSTEM_UI_FLAG_HIDE_NAVIGATION) == 0
                    && (fl & WindowManager.LayoutParams.
                            FLAG_DRAWS_SYSTEM_BAR_BACKGROUNDS) == 0) {
                // Ensure policy decor includes navigation bar
                dcf.bottom = mStableBottom;
                dcf.right = mStableRight;
            }
        }

        if ((fl & (FLAG_LAYOUT_IN_SCREEN | FLAG_LAYOUT_INSET_DECOR))
                == (FLAG_LAYOUT_IN_SCREEN | FLAG_LAYOUT_INSET_DECOR)) {
            if (DEBUG_LAYOUT) Slog.v(TAG, "layoutWindowLw(" + attrs.getTitle()
                    + "): IN_SCREEN, INSET_DECOR");
            // This is the case for a normal activity window: we want it
            // to cover all of the screen space, and it can take care of
            // moving its contents to account for screen decorations that
            // intrude into that space.
            if (attached != null) {
                // If this window is attached to another, our display
                // frame is the same as the one we are attached to.
                setAttachedWindowFrames(win, fl, adjust, attached, true, pf, df, of, cf, vf);
            } else {
                if (attrs.type == TYPE_STATUS_BAR_PANEL
                        || attrs.type == TYPE_STATUS_BAR_SUB_PANEL) {
                    // Status bar panels are the only windows who can go on top of
                    // the status bar.  They are protected by the STATUS_BAR_SERVICE
                    // permission, so they have the same privileges as the status
                    // bar itself.
                    //
                    // However, they should still dodge the navigation bar if it exists.

                    pf.left = df.left = of.left = hasNavBar
                            ? mDockLeft : mUnrestrictedScreenLeft;
                    pf.top = df.top = of.top = mUnrestrictedScreenTop;
                    pf.right = df.right = of.right = hasNavBar
                            ? mRestrictedScreenLeft+mRestrictedScreenWidth
                            : mUnrestrictedScreenLeft + mUnrestrictedScreenWidth;
                    pf.bottom = df.bottom = of.bottom = hasNavBar
                            ? mRestrictedScreenTop+mRestrictedScreenHeight
```

```
                : mUnrestrictedScreenTop + mUnrestrictedScreenHeight;

        if (DEBUG_LAYOUT) Slog.v(TAG, String.format(
                    "Laying out status bar window: (%d,%d - %d,%d)",
                    pf.left, pf.top, pf.right, pf.bottom));
    } else if ((fl & FLAG_LAYOUT_IN_OVERSCAN) != 0
            && attrs.type >= WindowManager.LayoutParams.FIRST_APPLICATION_WINDOW
            && attrs.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
        // Asking to layout into the overscan region, so give it that pure
        // unrestricted area.
        pf.left = df.left = of.left = mOverscanScreenLeft;
        pf.top = df.top = of.top = mOverscanScreenTop;
        pf.right = df.right = of.right = mOverscanScreenLeft +
        mOverscanScreenWidth;
        pf.bottom = df.bottom = of.bottom = mOverscanScreenTop
                + mOverscanScreenHeight;
    } else if (canHideNavigationBar()
            && (sysUiFl & View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION) != 0
            && attrs.type >= WindowManager.LayoutParams.FIRST_APPLICATION_
            WINDOW
            && attrs.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
        // Asking for layout as if the nav bar is hidden, lets the
        // application extend into the unrestricted overscan screen area.  We
        // only do this for application windows to ensure no window that
        // can be above the nav bar can do this.
        pf.left = df.left = mOverscanScreenLeft;
        pf.top = df.top = mOverscanScreenTop;
        pf.right = df.right = mOverscanScreenLeft + mOverscanScreenWidth;
        pf.bottom = df.bottom = mOverscanScreenTop + mOverscanScreenHeight;
        // We need to tell the app about where the frame inside the overscan
        // is, so it can inset its content by that amount -- it didn't ask
        // to actually extend itself into the overscan region.
        of.left = mUnrestrictedScreenLeft;
        of.top = mUnrestrictedScreenTop;
        of.right = mUnrestrictedScreenLeft + mUnrestrictedScreenWidth;
        of.bottom = mUnrestrictedScreenTop + mUnrestrictedScreenHeight;
    } else {
        pf.left = df.left = mRestrictedOverscanScreenLeft;
        pf.top = df.top = mRestrictedOverscanScreenTop;
        pf.right = df.right = mRestrictedOverscanScreenLeft
                + mRestrictedOverscanScreenWidth;
        pf.bottom = df.bottom = mRestrictedOverscanScreenTop
                + mRestrictedOverscanScreenHeight;
        // We need to tell the app about where the frame inside the overscan
        // is, so it can inset its content by that amount -- it didn't ask
        // to actually extend itself into the overscan region.
        of.left = mUnrestrictedScreenLeft;
        of.top = mUnrestrictedScreenTop;
        of.right = mUnrestrictedScreenLeft + mUnrestrictedScreenWidth;
        of.bottom = mUnrestrictedScreenTop + mUnrestrictedScreenHeight;
    }

    if ((fl & FLAG_FULLSCREEN) == 0) {
        if (win.isVoiceInteraction()) {
            cf.left = mVoiceContentLeft;
            cf.top = mVoiceContentTop;
            cf.right = mVoiceContentRight;
            cf.bottom = mVoiceContentBottom;
        } else {
            if (adjust != SOFT_INPUT_ADJUST_RESIZE) {
                cf.left = mDockLeft;
                cf.top = mDockTop;
                cf.right = mDockRight;
                cf.bottom = mDockBottom;
            } else {
                cf.left = mContentLeft;
                cf.top = mContentTop;
                cf.right = mContentRight;
                cf.bottom = mContentBottom;
            }
        }
```

```
            } else {
                // Full screen windows are always given a layout that is as if the
                // status bar and other transient decors are gone.  This is to avoid
                // bad states when moving from a window that is not hding the
                // status bar to one that is.
                cf.left = mRestrictedScreenLeft;
                cf.top = mRestrictedScreenTop;
                cf.right = mRestrictedScreenLeft + mRestrictedScreenWidth;
                 cf.bottom = mRestrictedScreenTop + mRestrictedScreenHeight;
            }
            applyStableConstraints(sysUiFl, fl, cf);
            if (adjust != SOFT_INPUT_ADJUST_NOTHING) {
                vf.left = mCurLeft;
                vf.top = mCurTop;
                vf.right = mCurRight;
                vf.bottom = mCurBottom;
            } else {
                vf.set(cf);
            }
        }
    } else if ((fl & FLAG_LAYOUT_IN_SCREEN) != 0 || (sysUiFl
            & (View.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN
                    | View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION)) != 0) {
        if (DEBUG_LAYOUT) Slog.v(TAG, "layoutWindowLw(" + attrs.getTitle() +
                "): IN_SCREEN");
        // A window that has requested to fill the entire screen just
        // gets everything, period.
        if (attrs.type == TYPE_STATUS_BAR_PANEL
                || attrs.type == TYPE_STATUS_BAR_SUB_PANEL) {
            pf.left = df.left = of.left = cf.left = hasNavBar
                    ? mDockLeft : mUnrestrictedScreenLeft;
            pf.top = df.top = of.top = cf.top = mUnrestrictedScreenTop;
            pf.right = df.right = of.right = cf.right = hasNavBar
                            ? mRestrictedScreenLeft+mRestrictedScreenWidth
                            : mUnrestrictedScreenLeft + mUnrestrictedScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom = hasNavBar
                             ? mRestrictedScreenTop+mRestrictedScreenHeight
                             : mUnrestrictedScreenTop + mUnrestrictedScreenHeight;
            if (DEBUG_LAYOUT) Slog.v(TAG, String.format(
                        "Laying out IN_SCREEN status bar window: (%d,%d - %d,%d)",
                        pf.left, pf.top, pf.right, pf.bottom));
        } else if (attrs.type == TYPE_NAVIGATION_BAR
                || attrs.type == TYPE_NAVIGATION_BAR_PANEL) {
            // The navigation bar has Real Ultimate Power.
            pf.left = df.left = of.left = mUnrestrictedScreenLeft;
            pf.top = df.top = of.top = mUnrestrictedScreenTop;
            pf.right = df.right = of.right = mUnrestrictedScreenLeft
                    + mUnrestrictedScreenWidth;
            pf.bottom = df.bottom = of.bottom = mUnrestrictedScreenTop
                    + mUnrestrictedScreenHeight;
            if (DEBUG_LAYOUT) Slog.v(TAG, String.format(
                        "Laying out navigation bar window: (%d,%d - %d,%d)",
                        pf.left, pf.top, pf.right, pf.bottom));
        } else if ((attrs.type == TYPE_SECURE_SYSTEM_OVERLAY
                        || attrs.type == TYPE_BOOT_PROGRESS)
                && ((fl & FLAG_FULLSCREEN) != 0)) {
            // Fullscreen secure system overlays get what they ask for.
            pf.left = df.left = of.left = cf.left = mOverscanScreenLeft;
            pf.top = df.top = of.top = cf.top = mOverscanScreenTop;
            pf.right = df.right = of.right = cf.right = mOverscanScreenLeft
                    + mOverscanScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom = mOverscanScreenTop
                    + mOverscanScreenHeight;
        } else if (attrs.type == TYPE_BOOT_PROGRESS
                || attrs.type == TYPE_UNIVERSE_BACKGROUND) {
            // Boot progress screen always covers entire display.
            pf.left = df.left = of.left = cf.left = mOverscanScreenLeft;
            pf.top = df.top = of.top = cf.top = mOverscanScreenTop;
            pf.right = df.right = of.right = cf.right = mOverscanScreenLeft
                    + mOverscanScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom = mOverscanScreenTop
```

```
                + mOverscanScreenHeight;
        } else if (attrs.type == TYPE_WALLPAPER) {
            // The wallpaper also has Real Ultimate Power, but we want to tell
            // it about the overscan area.
            pf.left = df.left = mOverscanScreenLeft;
            pf.top = df.top = mOverscanScreenTop;
            pf.right = df.right = mOverscanScreenLeft + mOverscanScreenWidth;
            pf.bottom = df.bottom = mOverscanScreenTop + mOverscanScreenHeight;
            of.left = cf.left = mUnrestrictedScreenLeft;
            of.top = cf.top = mUnrestrictedScreenTop;
            of.right = cf.right = mUnrestrictedScreenLeft + mUnrestrictedScreenWidth;
            of.bottom = cf.bottom = mUnrestrictedScreenTop + mUnrestrictedScreen
            Height;
        } else if ((fl & FLAG_LAYOUT_IN_OVERSCAN) != 0
                && attrs.type >= WindowManager.LayoutParams.FIRST_APPLICATION_WINDOW
                && attrs.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
            // Asking to layout into the overscan region, so give it that pure
            // unrestricted area.
            pf.left = df.left = of.left = cf.left = mOverscanScreenLeft;
            pf.top = df.top = of.top = cf.top = mOverscanScreenTop;
            pf.right = df.right = of.right = cf.right
                    = mOverscanScreenLeft + mOverscanScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom
                    = mOverscanScreenTop + mOverscanScreenHeight;
        } else if (canHideNavigationBar()
                && (sysUiFl & View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION) != 0
                && (attrs.type == TYPE_STATUS_BAR
                    || attrs.type == TYPE_TOAST
                    || (attrs.type >= WindowManager.LayoutParams.FIRST_APPLICATION_
                    WINDOW
                    && attrs.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW))) {
            // Asking for layout as if the nav bar is hidden, lets the
            // application extend into the unrestricted screen area.  We
            // only do this for application windows (or toasts) to ensure no window that
            // can be above the nav bar can do this.
            // XXX This assumes that an app asking for this will also
            // ask for layout in only content.  We can't currently figure out
            // what the screen would be if only laying out to hide the nav bar.
            pf.left = df.left = of.left = cf.left = mUnrestrictedScreenLeft;
            pf.top = df.top = of.top = cf.top = mUnrestrictedScreenTop;
            pf.right = df.right = of.right = cf.right = mUnrestrictedScreenLeft
                    + mUnrestrictedScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom = mUnrestrictedScreenTop
                    + mUnrestrictedScreenHeight;
        } else {
            pf.left = df.left = of.left = cf.left = mRestrictedScreenLeft;
            pf.top = df.top = of.top = cf.top = mRestrictedScreenTop;
            pf.right = df.right = of.right = cf.right = mRestrictedScreenLeft
                    + mRestrictedScreenWidth;
            pf.bottom = df.bottom = of.bottom = cf.bottom = mRestrictedScreenTop
                    + mRestrictedScreenHeight;
        }

        applyStableConstraints(sysUiFl, fl, cf);

        if (adjust != SOFT_INPUT_ADJUST_NOTHING) {
            vf.left = mCurLeft;
            vf.top = mCurTop;
            vf.right = mCurRight;
            vf.bottom = mCurBottom;
        } else {
            vf.set(cf);
        }
    } else if (attached != null) {
        if (DEBUG_LAYOUT) Slog.v(TAG, "layoutWindowLw(" + attrs.getTitle() +
                "): attached to " + attached);
        // A child window should be placed inside of the same visible
        // frame that its parent had.
        setAttachedWindowFrames(win, fl, adjust, attached, false, pf, df, of, cf, vf);
    } else {
        if (DEBUG_LAYOUT) Slog.v(TAG, "layoutWindowLw(" + attrs.getTitle() +
```

```
                "): normal window");
            // Otherwise, a normal window must be placed inside the content
            // of all screen decorations.
            if (attrs.type == TYPE_STATUS_BAR_PANEL) {
                // Status bar panels are the only windows who can go on top of
                // the status bar.  They are protected by the STATUS_BAR_SERVICE
                // permission, so they have the same privileges as the status
                // bar itself.
                pf.left = df.left = of.left = cf.left = mRestrictedScreenLeft;
                pf.top = df.top = of.top = cf.top = mRestrictedScreenTop;
                pf.right = df.right = of.right = cf.right = mRestrictedScreenLeft
                        + mRestrictedScreenWidth;
                pf.bottom = df.bottom = of.bottom = cf.bottom = mRestrictedScreenTop
                        + mRestrictedScreenHeight;
            } else if (attrs.type == TYPE_TOAST || attrs.type == TYPE_SYSTEM_ALERT
                    || attrs.type == TYPE_VOLUME_OVERLAY) {
                // These dialogs are stable to interim decor changes.
                pf.left = df.left = of.left = cf.left = mStableLeft;
                pf.top = df.top = of.top = cf.top = mStableTop;
                pf.right = df.right = of.right = cf.right = mStableRight;
                pf.bottom = df.bottom = of.bottom = cf.bottom = mStableBottom;
            } else {
                pf.left = mContentLeft;
                pf.top = mContentTop;
                pf.right = mContentRight;
                pf.bottom = mContentBottom;
                if (win.isVoiceInteraction()) {
                    df.left = of.left = cf.left = mVoiceContentLeft;
                    df.top = of.top = cf.top = mVoiceContentTop;
                    df.right = of.right = cf.right = mVoiceContentRight;
                    df.bottom = of.bottom = cf.bottom = mVoiceContentBottom;
                } else if (adjust != SOFT_INPUT_ADJUST_RESIZE) {
                    df.left = of.left = cf.left = mDockLeft;
                    df.top = of.top = cf.top = mDockTop;
                    df.right = of.right = cf.right = mDockRight;
                    df.bottom = of.bottom = cf.bottom = mDockBottom;
                } else {
                    df.left = of.left = cf.left = mContentLeft;
                    df.top = of.top = cf.top = mContentTop;
                    df.right = of.right = cf.right = mContentRight;
                    df.bottom = of.bottom = cf.bottom = mContentBottom;
                }
                if (adjust != SOFT_INPUT_ADJUST_NOTHING) {
                    vf.left = mCurLeft;
                    vf.top = mCurTop;
                    vf.right = mCurRight;
                    vf.bottom = mCurBottom;
                } else {
                    vf.set(cf);
                }
            }
        }
    }

    // TYPE_SYSTEM_ERROR is above the NavigationBar so it can't be allowed to extend over it.
    if ((fl & FLAG_LAYOUT_NO_LIMITS) != 0 && attrs.type != TYPE_SYSTEM_ERROR) {
        df.left = df.top = -10000;
        df.right = df.bottom = 10000;
        if (attrs.type != TYPE_WALLPAPER) {
            of.left = of.top = cf.left = cf.top = vf.left = vf.top = -10000;
            of.right = of.bottom = cf.right = cf.bottom = vf.right = vf.bottom = 10000;
        }
    }

    if (DEBUG_LAYOUT) Slog.v(TAG, "Compute frame " + attrs.getTitle()
            + ": sim=#" + Integer.toHexString(sim)
            + " attach=" + attached + " type=" + attrs.type
            + String.format(" flags=0x%08x", fl)
            + " pf=" + pf.toShortString() + " df=" + df.toShortString()
            + " of=" + of.toShortString()
            + " cf=" + cf.toShortString() + " vf=" + vf.toShortString()
```

```
            + " dcf=" + dcf.toShortString()
            + " sf=" + sf.toShortString());

    win.computeFrameLw(pf, df, of, cf, vf, dcf, sf);

    // Dock windows carve out the bottom of the screen, so normal windows
    // can't appear underneath them.
    if (attrs.type == TYPE_INPUT_METHOD && win.isVisibleOrBehindKeyguardLw()
            && !win.getGivenInsetsPendingLw()) {
        setLastInputMethodWindowLw(null, null);
        offsetInputMethodWindowLw(win);
    }
    if (attrs.type == TYPE_VOICE_INTERACTION && win.isVisibleOrBehindKeyguardLw()
            && !win.getGivenInsetsPendingLw()) {
        offsetVoiceInputWindowLw(win);
    }
}
```

通过上述实现代码可知，参数 win 表示当前要计算大小的窗口，第二个参数 attrs 表示窗口 win 的布局参数，第三个参数 attached 表示窗口 win 的父窗口，如果它的值等于 null，就表示窗口 win 没有父窗口。由此可见，函数 layoutWindowLw 会根据窗口 win 的是子窗口还是全屏窗口及其软键盘显示模式来决定如何计算它的大小。此处，只需关注输入法窗口和非全屏的 Activity 窗口的大小计算方式，其他类型的窗口大小计算方式是差不多的。

在计算一个窗口的大小时需要如下所示的 4 个参数。

- 父窗口大小 pf。
- 屏幕大小 df。
- 内容区域周边大小 cf。
- 可见区域周边大小 vf。

如果参数 win 代表的是输入法窗口，即参数 attrs 所描述的一个 WindowManager.LayoutParams 对象的成员变量 type 的值等于 TYPE_INPUT_METHOD，那么上述 4 个区域 pf、df、cf 和 vf 就等于 PhoneWindowManager 类的成员变量 mDockLeft、mDockTop、mDockRight 和 mDockBottom 所组成的区域的大小。如果参数 win 描述的是一个非全屏的 Activity 窗口，即参数 attrs 所描述的一个 WindowManager.LayoutParams 对象的成员变量 flags 的 FLAG_LAYOUT_IN_SCREEN 位和 FLAG_LAYOUT_INSET_DECOR 位等于 1，那么 PhoneWindowManager 类的成员函数 layoutWindowLw 就会继续检查参数 attached 的值是否等于 null。如果不等于 null，就说明参数 win 所描述的一个非全屏的 Activity 窗口附加在其他窗口上，即它具有一个父窗口，这时候就会调用另外一个成员函数 setAttachedWindowFrames 来计算它的大小。

当得到用于计算窗口 win 的 4 个参数 pf、df、cf 和 vf 后，就可以调用参数 win 所描述的一个 WindowState 对象的成员函数 computeFrameLw 来计算窗口 win 的具体大小。计算结果会得到窗口 win 的大小，以及它的内容区域周边大小和可见区域周边大小。窗口经过计算后得到的内容区域周边大小和可见区域周边大小并不一定等于参数 cf 和 vf 所指定的大小。在计算完成窗口 win 的大小之后，类 PhoneWindowManager 的成员函数 layoutWindowLw 会检查窗口 win 是否是一个输入法窗口，并且它是否指定了额外的内容区域周边和可见区域周边。如果这两个条件都成立，那么就需要调整类 PhoneWindowManager 的成员变量 mContentBottom 和 mCurBottom 的值，以便使类 PhoneWindowManager 的 4 个成员变量是（mContentLeft、mContentTop、mContentRight 和 mContentBottom）所围成的内容区域和 4 个成员变量（mCurLeft、mCurTop、mCurRight 和 mCurBottom）所围成的可见区域，不会覆盖到输入法窗口额外指定的内容区域周边和可见区域周边。

15.2.5 判断是否计算过

在文件 frameworks\base\services\core\java\com\android\server\wm\WindowState.java 中，通过函数 computeFrameLw 可以了解 Activity 窗口大小计算的具体过程，实现代码如下所示：

```
public void computeFrameLw(Rect pf, Rect df, Rect of, Rect cf, Rect vf, Rect dcf, Rect
sf) {
```

```
    mHaveFrame = true;

    TaskStack stack = mAppToken != null ? getStack() : null;
    if (stack != null && !stack.isFullscreen()) {
        getStackBounds(stack, mContainingFrame);
        if (mUnderStatusBar) {
            mContainingFrame.top = pf.top;
        }
    } else {
        mContainingFrame.set(pf);
    }

    mDisplayFrame.set(df);

    final int pw = mContainingFrame.width();
    final int ph = mContainingFrame.height();

    int w,h;
    if ((mAttrs.flags & WindowManager.LayoutParams.FLAG_SCALED) != 0) {
        if (mAttrs.width < 0) {
            w = pw;
        } else if (mEnforceSizeCompat) {
            w = (int)(mAttrs.width * mGlobalScale + .5f);
        } else {
            w = mAttrs.width;
        }
        if (mAttrs.height < 0) {
            h = ph;
        } else if (mEnforceSizeCompat) {
            h = (int)(mAttrs.height * mGlobalScale + .5f);
        } else {
            h = mAttrs.height;
        }
    } else {
        if (mAttrs.width == WindowManager.LayoutParams.MATCH_PARENT) {
            w = pw;
        } else if (mEnforceSizeCompat) {
            w = (int)(mRequestedWidth * mGlobalScale + .5f);
        } else {
            w = mRequestedWidth;
        }
        if (mAttrs.height == WindowManager.LayoutParams.MATCH_PARENT) {
            h = ph;
        } else if (mEnforceSizeCompat) {
            h = (int)(mRequestedHeight * mGlobalScale + .5f);
        } else {
            h = mRequestedHeight;
        }
    }

    if (!mParentFrame.equals(pf)) {
        //Slog.i(TAG, "Window " + this + " content frame from " + mParentFrame
        //        + " to " + pf);
        mParentFrame.set(pf);
        mContentChanged = true;
    }
    if (mRequestedWidth != mLastRequestedWidth || mRequestedHeight != mLastRequested
Height) {
        mLastRequestedWidth = mRequestedWidth;
        mLastRequestedHeight = mRequestedHeight;
        mContentChanged = true;
    }

    mOverscanFrame.set(of);
    mContentFrame.set(cf);
    mVisibleFrame.set(vf);
    mDecorFrame.set(dcf);
    mStableFrame.set(sf);

    final int fw = mFrame.width();
    final int fh = mFrame.height();
```

```
//System.out.println("In: w=" + w + " h=" + h + " container=" +
//container + " x=" + mAttrs.x + " y=" + mAttrs.y);

float x, y;
if (mEnforceSizeCompat) {
    x = mAttrs.x * mGlobalScale;
    y = mAttrs.y * mGlobalScale;
} else {
    x = mAttrs.x;
    y = mAttrs.y;
}

Gravity.apply(mAttrs.gravity, w, h, mContainingFrame,
        (int) (x + mAttrs.horizontalMargin * pw),
        (int) (y + mAttrs.verticalMargin * ph), mFrame);

//System.out.println("Out: " + mFrame);

// Now make sure the window fits in the overall display.
Gravity.applyDisplay(mAttrs.gravity, df, mFrame);

// Make sure the content and visible frames are inside of the
// final window frame.
mContentFrame.set(Math.max(mContentFrame.left, mFrame.left),
        Math.max(mContentFrame.top, mFrame.top),
        Math.min(mContentFrame.right, mFrame.right),
        Math.min(mContentFrame.bottom, mFrame.bottom));

mVisibleFrame.set(Math.max(mVisibleFrame.left, mFrame.left),
        Math.max(mVisibleFrame.top, mFrame.top),
        Math.min(mVisibleFrame.right, mFrame.right),
        Math.min(mVisibleFrame.bottom, mFrame.bottom));

mStableFrame.set(Math.max(mStableFrame.left, mFrame.left),
        Math.max(mStableFrame.top, mFrame.top),
        Math.min(mStableFrame.right, mFrame.right),
        Math.min(mStableFrame.bottom, mFrame.bottom));

mOverscanInsets.set(Math.max(mOverscanFrame.left - mFrame.left, 0),
        Math.max(mOverscanFrame.top - mFrame.top, 0),
        Math.max(mFrame.right - mOverscanFrame.right, 0),
        Math.max(mFrame.bottom - mOverscanFrame.bottom, 0));

mContentInsets.set(mContentFrame.left - mFrame.left,
        mContentFrame.top - mFrame.top,
        mFrame.right - mContentFrame.right,
        mFrame.bottom - mContentFrame.bottom);

mVisibleInsets.set(mVisibleFrame.left - mFrame.left,
        mVisibleFrame.top - mFrame.top,
        mFrame.right - mVisibleFrame.right,
        mFrame.bottom - mVisibleFrame.bottom);

mStableInsets.set(Math.max(mStableFrame.left - mFrame.left, 0),
        Math.max(mStableFrame.top - mFrame.top, 0),
        Math.max(mFrame.right - mStableFrame.right, 0),
        Math.max(mFrame.bottom - mStableFrame.bottom, 0));

mCompatFrame.set(mFrame);
if (mEnforceSizeCompat) {
    // If there is a size compatibility scale being applied to the
    // window, we need to apply this to its insets so that they are
    // reported to the app in its coordinate space.
    mOverscanInsets.scale(mInvGlobalScale);
    mContentInsets.scale(mInvGlobalScale);
    mVisibleInsets.scale(mInvGlobalScale);
    mStableInsets.scale(mInvGlobalScale);

    // Also the scaled frame that we report to the app needs to be
    // adjusted to be in its coordinate space.
```

```
            mCompatFrame.scale(mInvGlobalScale);
        }

        if (mIsWallpaper && (fw != mFrame.width() || fh != mFrame.height())) {
            final DisplayContent displayContent = getDisplayContent();
            if (displayContent != null) {
                final DisplayInfo displayInfo = displayContent.getDisplayInfo();
                mService.updateWallpaperOffsetLocked(this,
                        displayInfo.logicalWidth, displayInfo.logicalHeight, false);
            }
        }

        if (DEBUG_LAYOUT || WindowManagerService.localLOGV) Slog.v(TAG,
                "Resolving (mRequestedWidth="
                + mRequestedWidth + ", mRequestedheight="
                + mRequestedHeight + ") to" + " (pw=" + pw + ", ph=" + ph
                + "): frame=" + mFrame.toShortString()
                + " ci=" + mContentInsets.toShortString()
                + " vi=" + mVisibleInsets.toShortString()
                + " vi=" + mStableInsets.toShortString());
    }
```

在上述代码中，类 WindowState 的成员变量 mHaveFrame 用来描述是否计算过一个窗口的大小。当成员函数 computeFrameLw 被调用时，说明一个相应的窗口的大小得到计算了，所以，函数 computeFrameLw 一开始就会将成员变量 mHaveFrame 的值设置为 true。函数 computeFrameLw 计算一个窗口大小的过程如下所示。

- 类 WindowState 的成员变量 mContainingFrame 和 mDisplayFrame 描述了当前正在处理的窗口的父窗口和屏幕的大小，它们正好就分别等于参数 pf 和 df 的大小，因此，函数就直接将参数 pf 和 df 的值分别保存在类 WindowState 的成员变量 mContainingFrame 和 mDisplayFrame 中。
- 如果当前正在处理的窗口运行在兼容模式，即类 WindowState 的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 flags 的 FLAG_COMPATIBLE_WINDOW 位等于 1，那么，就需要将其父窗口的大小 mContainingFrame 限制在兼容模式下的屏幕区域中。兼容模式下的屏幕区域保存在 WindowManagerService 类的成员变量 mCompatibleScreenFrame 中，将父窗口的大小 mContainingFrame 与它执行一个相交操作，就可以将父窗口的大小限制在兼容模式下的屏幕区域中。
- 在当前正在处理的窗口运行在兼容模式的情况下，如果其大小被限制在了兼容模式下的屏幕区域之中，也就是成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 flags 的 FLAG_LAYOUT_NO_LIMITS 位等于 0，那么，同样需要将屏幕大小 mDisplayFrame 限制在兼容模式下的屏幕区域 mCompatibleScreenFrame，这也是通过执行一个相交操作来完成。
- 类 WindowState 的成员变量 mFrame 描述的是当前正在处理的窗品的大小，目的就是计算它的值，此时可以初步得到了窗口的宽度 w 和高度 h。一个窗口的大小是受以下因素影响的。

➢ 是否指定缩放因子。如果一个窗口的大小被指定了缩放因子，即 WindowState 类的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 flags 的 FLAG_SCALED 位等于 1，那么该窗口的大小就是在它的布局参数中指定的，即是由 WindowState 类的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 width 和 height 所指定的。但是，如果在布局参数中指定的窗口宽度或者高度小于 0，那么就会使用其父窗口的大小来作为当前窗口的大小。当前窗口的父窗口的宽度和高度分别保存在变量 pw 和 ph 中。

➢ 是否设置等于父窗口的大小。如果一个窗口的大小被指定为其父窗口的大小，即 WindowState 类的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 width 和 height 的值等于 mAttrs.MATCH_PARENT，那么该窗口的大小就会等于其父窗口的大小，即等于变量 pw 和 ph 所描述的宽度和高度。另一方面，如果一个窗口的大小没有指定为其父窗口的大小，那么它的大小就会等于应用程序进程请求 WindowManagerService 所设置的大小，即等于 WindowState 类的成员变量 mRequestedWidth 和 mRequestedHeight 所描述的宽度和高度。

- 开始确定最终的窗口大小，需要进一步根据窗口的 Gravity 属性来作调整。需要将窗口的

最终大小保存在变量 frame 中，即 WindowState 类的成员变量 mFrame 中。整个调整工作分为如下两步实现。

➢ 根据窗口的 Gravity 值，以及位置、初始大小和父窗口大小，来计算窗口的大小，并且保存在变量 frame 中，即保存在 WindowState 类的成员变量 mFrame 中，这是通过调用 Gravity 类的静态成员函数 apply 来实现的。其中，窗口的初始大小保存在变量 w 和 h 中，父窗口大小保存在变量 container 中，即 WindowState 类的成员变量 mContainingFrame 中，位置保存在 WindowState 类的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 x 和 y 中。注意，如果窗口指定了相对父窗口的 margin 值，那么还需要相应地调整其位置值，即要在指定的位置值的基础上，再加上相对父窗口的 margin 值。一个窗口相对父窗口的 margion 是通过一个百分比来表示的，用这个百分比乘以父窗口的大小就可以得到绝对值。这个百分比又分为在水平方向和垂直方向两个值，分别保存在 WindowState 类的成员变量 mAttrs 所指向的一个 WindowManager.LayoutParams 对象的成员变量 horizontalMargin 和 verticalMargin 中。

➢ 前面步骤计算得到的窗口大小没有考虑屏幕的大小，因此，接下来还需要继续调用 Gravity 类的静态成员函数 applyDisplay 来将前面计算得到的窗口大小限制在屏幕区域 df 中，即限制在 WindowState 类的成员变量 mDisplayFrame 所描述的区域中。

● 开始计算窗口的内容区域周边和可见区域周边大小。内容区域周边和可见区域周边大小的计算很简单，只要将窗口的大小 frame，即 WindowState 类的成员变量 mFrame 所描述的区域，分别减去变量 content 和 visible，即 WindowState 类的成员变量 mContentFrame 和 mVisibleFrame 所描述的区域，就可以得到窗口的内容区域周边和可见区域周边大小，它们分别保存在 WindowState 类的成员变量 mContentInsets 和 mVisibleInsets 中。注意，在计算窗口的内容区域周边和可见区域周边大小之前，首先要保证窗口的内容区域和可见区域包含在整个窗口区域中，这是由中间的 8 个 if 语句来保证的。

● 窗口上一次的大小被保存在变量 fw 和 fh 中。如果当前正在处理的窗口是一个壁纸窗口，即 WindowState 类的成员变量 mIsWallpaper 的值等于 true，并且该窗口的大小发生了变化，即变量 fw 和 fh 所描述的窗口大小不等于变量 frame 描述的窗口大小，那么就需要调用 WindowManagerService 类的成员函数 updateWallpaperOffsetLocked 来更新壁纸的位置。在后面的内容中，再详细描述系统的壁纸窗口的位置是如何计算的。

到此为止，一个窗口的大小就计算完成。从上述计算的过程可知，整个窗口大小被保存在类 WindowState 的成员变量 mFrame 中，而窗口的内容区域周边大小和可见区域周边大小分别保存在 WindowState 类的成员变量 mContentInsets 和 mVisibleInsets 中。

最后需要返回到文件 frameworks\base\policy\src\com\android\internal\policy\impl\PhoneWindowManager.java 中，通过函数 finishLayoutLw 结束当前窗口大小的计算工作，具体实现代码如下所示：

```
public void finishLayoutLw() {
        return;
    }
```

到此为止，计算 Activity 窗口大小的内容全部讲解完毕。但是，这些内容仅是 WindowManagerService 系统的一小部分，WindowManagerService 还涉及了窗口组织、输入法窗口调整、壁纸窗口调整、Z 轴位置计算与调整、启动窗口显示、窗口切换过程和窗口动画显示等内容。由于本书篇幅的限制，这些内容将不在书中呈现，读者可以参阅相关资料了解。

第 16 章　分析电话系统

对于一款 Android 手机设备来说，其最重要的功能便是实现通信处理，例如拨打/接通电话和收发短信/彩信等。在实现上述通信功能的过程中，Android 需要确保这些信息的安全性。在本章的内容中，将详细讲解 Android 5.0 源代码中电话系统的基本知识。

16.1 Android 电话系统详解

Android 系统作为一款流行的智能手机平台，电话（Telephony）部分功能自然十分重要。电话系统的主要功能是呼叫（Call）、短信（SMS）、数据连接（Data Connection）、SIM 卡和电话本等功能。本书将介绍绝大多数功能的实现框架。

16.1.1　电话系统简介

Android 的 Radio Interface Layer (RIL)提供了电话服务和 Radio 硬件之间的抽象层。Radio Interface Layer RIL(Radio Interface Layer)负责数据的可靠传输、AT 命令的发送以及 response 的解析。应用处理器通过 AT 命令集与带 GPRS 功能的无线通信模块通信。AT command 是由 Hayes 公司发明的，是一个调制解调器制造商采用的一个调制解调器命令语言，每条命令以字母“AT”开头。

在 Android 系统中，实现电话功能部分的具体说明如下所示。

（1）在“hardware/ril/include/telephony/”目录中，文件 ril.h 是 ril 部分的基础头文件，其中定义的结构体 RIL_RadioFunctions 的代码如下所示：

```
typedef struct {
int version;
RIL_RequestFunc onRequest;
RIL_RadioStateRequest onStateRequest;
RIL_Supports supports;
RIL_Cancel onCancel;
RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

在结构体 RIL_RadioFunctions 中包含了几个函数指针的结构体，这实际上是一个移植层的接口。在实现下层的库之后，由 rild 守护进程得到这些函数指针，执行对应的函数。

其中几个重要的函数指针的原型如下所示：

```
typedef void (*RIL_RequestFunc) (int request, void *data,
size_t datalen, RIL_Token t);
typedef RIL_RadioState (*RIL_RadioStateRequest)();
typedef int (*RIL_Supports)(int requestCode);
typedef void (*RIL_Cancel)(RIL_Token t);
typedef const char * (*RIL_GetVersion) (void);
```

其中最为重要的函数是 onRequest()，这是一个请求执行的函数。

（2）rild 守护进程。

在 Android 系统中，rild 守护进程的实现文件包含在“hardware/ril/rild”目录中，其中包含了文件 rild.c 和 radiooptions.c，这个目录中的文件经过编译后会生成一个可执行程序，这个程序在系统的安装路径如下：

```
/system/bin/rild
```

文件 rild.c 是这个守护进程的入口，它具有一个主函数的入口 main()，执行的过程是将请求转换成 AT 命令的字符串，给下层的硬件执行。在运行过程中，使用 dlopen 打开“/system/lib/”路径中名称为 libreference-ril.so 的动态库，然后从中取出 RIL_Init 符号来运行。

RIL_Init 符号是一个函数指针，执行这个函数后，返回的是一个 RIL_RadioFunctions 类型的指针。得到这个指针后，调用 RIL_register()函数，将这个指针注册到 libril 库之中，然后进入循环。事实上，这个守护进程提供了一个申请处理的框架，而具体的功能都是在 libril.so 和 libreference-ril.so 中完成的。

（3）libreference-ril.so 动态库。

在 Android 系统中，libreference-ril.so 动态库的路径如下所示：

```
hardware/ril/reference-ril
```

其中 Android 电话功能主要的文件是 reference-ril.c 和 atchannel.c。这个库必须实现的是一个名为 RIL_Init 的函数，这个函数执行的结果是返回一个 RIL_RadioFunctions 结构体的指针，指针指向函数指针。这个库在执行的过程中需要创建一个线程来执行实际的功能。在执行的过程中，这个库将打开一个“/dev/ttySXXX”的终端（终端的名字是从上层传入的），然后利用这个终端控制硬件执行。

（4）libril.so 动态库。

在 Android 系统中，libril.so 库的目录如下所示：

```
hardware/ril/libril
```

其中的主要文件是 ril.cpp，此库需要实现以下几个接口：

```
RIL_startEventLoop(void);
void RIL_setcallbacks (const RIL_RadioFunctions *callbacks);
RIL_register (const RIL_RadioFunctions *callbacks);
RIL_onRequestComplete(RIL_Token t, RIL_Errno e, void *response,
size_t responselen);
void RIL_onUnsolicitedResponse(int unsolResponse, void *data,
size_t datalen);
RIL_requestTimedCallback (RIL_TimedCallback callback, void *param,
const struct timeval *relativeTime);
```

上述函数也是被 rild 守护进程调用的，不同的 vendor 可以通过自己的方式实现这几个接口，这样可以保证 RIL 可以在不同系统的移植。其中函数 RIL_register()把外部的 RIL_RadioFunctions 结构体注册到这个库之中，在恰当的时候调用相应的函数。在 Android 电话功能执行的过程中，这个库处理了一些将请求转换成字符串的功能。

16.1.2 电话系统结构

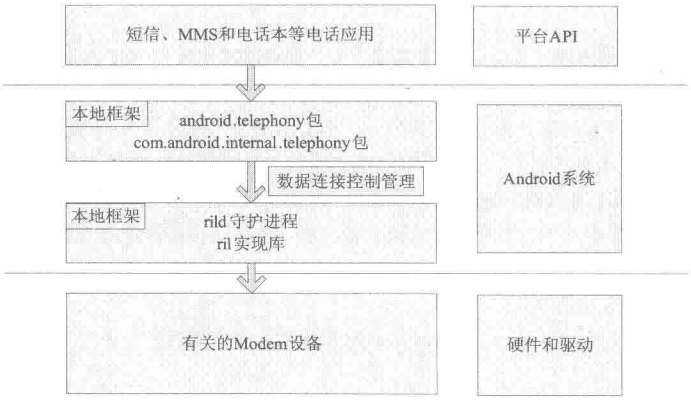
Android 电话系统主要分为 Modem 驱动、RIL（Radio Interface Layer）、电话服务框架和应用共 4 层结构，具体结构如图 16-1 所示。

Android 电话系统的代码结构如图 16-2 所示。

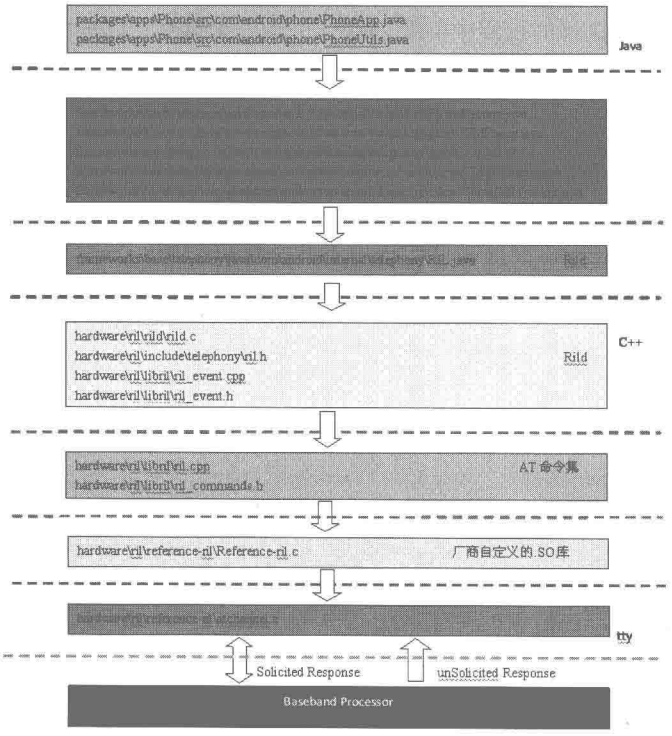
由图 16-1 可知，Android 电话系统从上到下主要包括 Java 应用、Java 框架、本地 RIL 层和 Modem 驱动，这几部分的具体说明如下所示。

（1）Modem 驱动

实现电话功能的主要硬件是通信模块（Modem），Modem 通过与通信网络进行沟通传输语音及数据，完成呼叫、短信等相关电话功能。对于大部分目前的独立通信模块而言，无论是 2G 还是 3G 都已经非常成熟，模块化相当完善，硬件接口非常简单，也有着相对统一的软件接口。一般的 Modem 模块装上 SIM 卡，直接上电即可工作，自动完成初始的找网、网络注册等工作，完成之后即可打电话、发短信等。但独立模块因为体积问题，在手机设计中较少使用，而是使用 chip-on-board 的方式。另外也有不少 Modem 基带与应用处理器共存。



▲图 16-1　电话系统的层次结构



▲图 16-2　电话系统的代码结构

（2）RIL 硬件抽象层

RIL 负责数据的可靠传输、AT 命令发送以及 Response 解析功能。应用处理器通过 AT 命令集和带 GPRS 功能的无线通信模块实现通信功能。AT command 是由 Hayes 公司发明的，是一个调制解调器制造商采用的一个调制解调器命令语言，每条命令以字母 AT 开头。

RIL 支持的本地代码包括 ril 库和守护进程，主要代码路径如下所示：

```
hardware/ril/include
hardware/ril/libril
hardware/ril/rild
hardware/ril/reference-ril
```

具体编译结果如下所示。

- /system/bin/rild：守护进程。
- /system/lib/libril.so：生成 RIL 库。

- /system/lib/libreference-ril.so：生成 RIL 参考库。

在 Android 电话系统中没有 JNI 部分，RIL 守护进程通过名为 rild 的 Socker 和 Java 框架层进行通信。

（3）Java 框架

此部分的代码路径如下所示：

```
frameworks/base/telephony/java/
```

在此目录中存在如下 Java 类。

- android/telephony：实现了 Java 类 android.telephony、android.telephony.gsm 以及 android.telephony.cdma。
- com/android/internal/telephony：实现了内部的类 com.android.internal.telephony、com.android.internal.telephony.gsm 以及 com.android.internal.telephony.cdma，带 GSM 的是 GSM 的专用协议，而不带的是通用部分。

（4）应用层

在电话系统中，通过 Service 实现 Phone 应用，并同时实现 Phone 的 UI 界面逻辑。而短信和网络选择分别在 MMS 和 Settings 应用中实现。

16.1.3 驱动程序介绍

在介绍驱动移植之前，需要先了解 rild 与 libril.so 以及 librefrence_ril.so 的关系。

（1）rild

用于实现 main()函数作为整个 ril 层的入口点，负责完成初始化。

（2）libril.so

与 rild 结合相当紧密，是其共享库，在编译时就已经建立了这一关系。这部分功能通过文件 ril.cpp 和 ril_event.cpp 实现，其中文件 libril.so 驻留在 rild 这一守护进程中，主要完成同上层通信的工作，接受 ril 请求并传递给 librefrence_ril.so，同时把来自 librefrence_ril.so 的反馈回传给调用进程。

（3）librefrence_ril.so

rild 通过手动的 dlopen 方式加载，此加载过程的结合稍微松散，这是因为 librefrence.so 主要负责与 Modem 硬件通信的缘故，这样做更方便替换或修改以适配更多的 Modem 种类。转换来自 libril.so 的请求为 AT 命令，同时监控 Modem 的反馈信息，并传递回 libril.so。在初始化时，rild 通过符号 RIL_Init 获取一组函数指针并以此与之建立联系。

（4）radiooptions

radiooptiongs 通过获取启动参数，利用 socket 和 rild 进行通信，可供调试时配置 Modem 参数。

经过前面内容的介绍，我们知道移植 Modem 驱动的主要工作是 USB 转 Serial 接口，以及实现特殊 USB/Serial。上述驱动保存在如下目录中：

```
drivers/usb/seral/
drivers/seral/
```

在 Android 电话系统的 Modem 驱动中，通常使用 USB 转 Serial 标准实现 AT 和数据通道的接口，此功能可以通过文件“drivers/usb/serial/option.c”实现的，首先需要定义下面的 option_1port_device 设备。具体代码如下所示：

```
static struct usb_serial_driver option_1port_device = {
        .driver = {
                .owner =        THIS_MODULE,
                .name =         "option1",
        },
        .description       = "GSM modem (1-port)",
        .usb_driver        = &option_driver,
        .id_table          = option_ids,
        .num_ports         = 1,
        .probe             = option_probe,
        .open              = usb_wwan_open,
```

```
        .close            = usb_wwan_close,
        .dtr_rts          = usb_wwan_dtr_rts,
        .write            = usb_wwan_write,
        .write_room       = usb_wwan_write_room,
        .chars_in_buffer  = usb_wwan_chars_in_buffer,
        .set_termios      = usb_wwan_set_termios,
        .tiocmget         = usb_wwan_tiocmget,
        .tiocmset         = usb_wwan_tiocmset,
        .ioctl            = usb_wwan_ioctl,
        .attach           = usb_wwan_startup,
        .disconnect       = usb_wwan_disconnect,
        .release          = usb_wwan_release,
        .read_int_callback = option_instat_callback,
#ifdef CONFIG_PM
        .suspend          = usb_wwan_suspend,
        .resume           = usb_wwan_resume,
#endif
};
```

上述驱动会通过 option_init 注册成为 usb_serial_driver，然后通过对 usb_driver 注册来响应 USB 设备枚举，对应的 USB Driver 如下：

```
static struct usb_driver option_driver = {
        .name       = "option",
        .probe      = usb_serial_probe,
        .disconnect = usb_serial_disconnect,
#ifdef CONFIG_PM
        .suspend    = usb_serial_suspend,
        .resume     = usb_serial_resume,
        .supports_autosuspend = 1,
#endif
        .id_table   = option_ids,
        .no_dynamic_id =        1,
};
```

为了匹配在上述枚举中定义的设备，需要定义 WENDOR ID 和 PRODUCT ID。在文件 option.c 中已经定义了很多可以支持的设备，我们只需在数据里添加设备 IDS 即可。数组 option_ids[]的代码如下所示：

```
static const struct usb_device_id option_ids[] = {
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_COLT) },
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA) },
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_LIGHT) },
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_QUAD) },
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_QUAD_LIGHT) },
        { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_NDIS) },
.....................................
```

我们可以定义两个需要的 ID，可以按照下面的格式将其添加到上述 option_ids[]数组中：

```
{ USB_DEVICE(XXX_VENDOR_ID, XXX_PRODUCT_RICOLA_LIGHT) },
```

上述代码中的加粗部分是我们定义的 ID 内容。

此时开启 Modem 电源后，会出现两个分别名为“/dev/ttyusb0”和“/dev/ttyusb1”的端口。

16.1.4　RIL 接口

1. RIL 目录分析

（1）目录“hardware/ril/libril”

目录“hardware/ril/libril”下的代码负责与上层客户进程进行交互。在接收到客户进程命令后，调用相应函数进行处理，然后将命令响应结果传回客户进程。在收到来自网络端的事件后，也传给客户进程。

- 文件 ril_commands.h：在里面列出了 telephony 可以接收的命令、每个命令对应的处理函数以及命令响应的处理函数。例如下面的代码：

```
{RIL_REQUEST_GET_SIM_STATUS, dispatchVoid, responseSimStatus},
{RIL_REQUEST_ENTER_SIM_PIN, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PUK, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PIN2, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PUK2, dispatchStrings, responseInts},
{RIL_REQUEST_CHANGE_SIM_PIN, dispatchStrings, responseInts},
... .. ..
```

- 文件 ril_unsol_commands.h：在里面列出了 telephony 可以接收的事件类型和对每个事件的处理函数。例如下面的代码：

```
{RIL_UNSOL_RESPONSE_RADIO_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NETWORK_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NEW_SMS, responseString, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NEW_SMS_STATUS_REPORT, responseString, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NEW_SMS_ON_SIM, responseInts, WAKE_PARTIAL},
.. . . . .
```

- 文件 ril_event.h/cpp：实现了处理与事件源（端口，modem 等）相关的功能。ril_event_loop 监视所有注册的事件源，当某事件源有数据到来时，相应事件源的回调函数被触发（firePending →ev→func()）。
- 文件 ril.cpp 功能较为庞大，各个部分的具体功能如下所示。

➢ RIL_register 函数：打开监听端口，接收来自客户进程的命令请求（s_fdListen = android_get_control_socket(SOCKET_NAME_RIL);），当与某客户进程连接建立时，调用 listenCallback 函数；创建一单独线程监视并处理所有事件源（通过 ril_event_loop）。

➢ listenCallback 函数：当与客户进程连接建立时，此函数被调用。此函数接着调用 processCommandsCallback 处理来自客户进程的命令请求。

➢ processCommandsCallback 函数：具体处理来自客户进程的命令请求。对每一个命令，ril_commands.h 中都规定了对应的命令处理函数（dispatchXXX），processCommandsCallback 会调用这个命令处理函数进行处理。

➢ dispatch 系列函数：此函数接收来自客户进程的命令及相应参数，并调用 onRequest 进行处理。

➢ RIL_onUnsolicitedResponse 函数：将来自网络端的事件封装（通过调用 responseXXX）后传给客户进程。

➢ RIL_onRequestComplete 函数：将命令的最终响应结构封装（通过调用 responseXXX）后传给客户进程。

➢ response 系列函数：对每一个命令，都规定了一个对应的 response 函数来处理命令的最终响应；对每一个网络端的事件，也规定了一个对应的 response 函数来处理此事件。response 函数可被 onUnsolicitedResponse 或者 onRequestComplete 调用。

（2）目录“hardware/ril/reference-ril”

在此目录下的代码主要负责与 Modem 进行交互。

- 文件 reference-ril.c：此文件的核心是函数 onRequest()和 onUnsolicited()。

➢ onRequest 函数：在这个函数里，每一个 RIL_REQUEST_XXX 请求都被转化成相应的 AT command，发送给 modem，然后睡眠等待。当收到此 AT command 的最终响应后，线程被唤醒，将响应传给客户进程（RIL_onRequestComplete -> sendResponse）。

➢ onUnsolicited 函数：这个函数处理 modem 从网络端收到的各种事件，如网络信号变化，拨入的电话，收到短信等。然后将时间传给客户进程（RIL_onUnsolicitedResponse -> sendResponse）。

- 文件 atchannel.c：负责向 modem 读写数据。其中，写数据（主要是 AT command）功能运行在主线程中，读数据功能运行在一个单独的读线程中。此文件的核心功能是函数 at_send_command_full_nolock，此函数运行在主线程里面，用于将一个 AT command 命令写入 modem 后进入睡眠状态（使用 pthread_cond_wait 或类似函数），直到 modem 读线程将其唤醒。唤醒后此函数获得了 AT command 的最终响应并返回。函数 readerLoop 运行在一个单独的读线程里面，负责从

modem 中读取数据。读到的数据可分为 3 种类型：网络端传入的事件；modem 对当前 AT command 的部分响应；modem 对当前 AT command 的全部响应。对第三种类型的数据（AT command 的全部响应），读线程唤醒（pthread_cond_signal）睡眠状态的主线程。

- 文件 at_tok.c：提供 AT 响应的解析函数。
- 文件 misc.c：只提供一个字符串匹配函数。

（3）目录“hardware/ril/rild”

在此目录下的代码主要是为了生成 rild 和 radiooptions 的可执行文件。

- 文件 radiooptions.c：用于生成 radiooptions 的可执行文件，radooptions 程序仅是把命令行参数传递给 socket｛rild-debug｝去处理而已，从而达到与 rild 通信，可供调试时配置 Modem 参数。
- 文件 rild.c：用于生成 rild 的可执行文件。

2. RIL 接口移植

实现 RIL 接口的过程比较复杂，复杂程度主要体现在维护工作、需要处理的命令和较多结构体上。在文件“hardware/ril/include/telephony/ril.h”中定义了 RIL 接口，并且同一个目录下的 ril_cdma_sms.h 作为对 CDMA 协议的有利补充而存在。

在文件 ril.h 中首先定义核心结构 RIL_Env，具体代码如下所示：

```
struct RIL_Env {
    //请求完成
    void (*OnRequestComplete)(RIL_Token t, RIL_Errno e,
                        void *response, size_t responselen);
    //上报消息响应
    void (*OnUnsolicitedResponse)(int unsolResponse, const void *data,
                                  size_t datalen);
    //请求中进行周期处理
    void (*RequestTimedCallback) (RIL_TimedCallback callback,
                                  void *param, const struct timeval *relativeTime);
};
```

上述结构体是由 libril.so 库作为标准实现的可以为硬件抽象层的实现库调用。能够在发生请求时针对不同的情况而作出具体响应，例如，有请求完成函数响应、上报信息函数响应和请求中周期处理函数 3 个响应。

在结构体 RIL_RadioFunctions 定义需要的函数指针，具体代码如下所示：

```
typedef void (*RIL_RequestFunc) (int request, void *data,
                                 size_t datalen, RIL_Token t);
typedef RIL_RadioState (*RIL_RadioStateRequest)();
typedef int (*RIL_Supports)(int requestCode);
typedef void (*RIL_Cancel)(RIL_Token t);
typedef void (*RIL_TimedCallback) (void *param);
typedef const char * (*RIL_GetVersion) (void);

typedef struct {
    int version;        /
    RIL_RequestFunc onRequest;
    RIL_RadioStateRequest onStateRequest;
    RIL_Supports supports;
    RIL_Cancel onCancel;
    RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

RIL 实现库需要实现结构体 RIL_RadioFunctions 中的内容，当通过 RIL_Init 返回 rild 后，rild 会调用下面的函数完成注册：

```
void RIL_register (const RIL_RadioFunctions *callbacks);
```

此时成功搭建了 rild 到 RIL 的实现库的请求发送路径和响应路径。

16.1.5　分析电话系统的实现流程

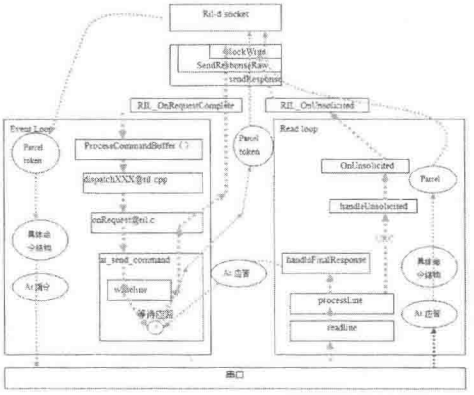
通过本章前面内容的学习，了解了 Android 电话系统的基本结构和移植方法。在本节的内容中，

将简要介绍 Android 电话系统的实现流程。在 Android 系统中，拨打电话的基本流程如图 16-3 所示。

1. 初始启动流程

主入口功能是通过文件 rild.c 中的 main()函数实现的，这个过程主要完成以下 3 个任务。

（1）开启 libril.so 中的 event 机制，这是在函数 RIL_startEventLoop()中实现的，是最核心的由多路I/O驱动的消息循环。



▲图 16-3 拨打电话的流程

此任务的核心内容是通过 RIL_startEventLoop()函数实现的，此函数在文件 ril.cpp 中实现，其主要目的是通过 pthread_create（&s_tid_dispatch, &attr, eventLoop, NULL）建立一个 dispatch 线程，入口点在 eventLoop。而在 eventLoop 中会调 ril_event.cpp 中的 ril_event_loop()函数，以建立起消息（event）队列机制。接下来看上述消息队列的机制，实现代码都在文件 ril_event.cpp 中。主要代码如下所示：

```
void ril_event_init();
void ril_event_set(struct ril_event * ev, int fd, bool persist, ril_event_cb func, void
* param);
void ril_event_add(struct ril_event * ev);
void ril_timer_add(struct ril_event * ev, struct timeval * tv);
void ril_event_del(struct ril_event * ev);
void ril_event_loop();
struct ril_event {
struct ril_event *next;
struct ril_event *prev;
int fd;
int index;
bool persist;
struct timeval timeout;
ril_event_cb func;
void *param;
};
```

每个 ril_event 结构都与一个 fd 句柄绑定（可以是文件，socket，管道等），并且带一个 func 指针去执行指定的操作。具体流程是：当完成 ril_event_init 后，通过 ril_event_set 配置一个新 ril_event，并通过 ril_event_add 加入队列之中（通常用 rilEventAddWakeup 来添加），add 会把队列里所有 ril_event 的 fd 放入一个 fd 集合 readFds 中。这样，ril_event_loop 能通过一个多路复用 I/O 的机制（select）来等待这些 fd，如果任何一个 fd 有数据写入，则进入分析流程 processTimeouts()、processReadReadies(&rfds, n)、firePending()。后文会详细分析这些流程。

在进入 ril_event_loop 之前，通过 pipe 机制实现挂入了一个 s_wakeupfd_event，这个 event 的目的是可以在某些情况下能内部唤醒 ril_event_loop 的多路复用阻塞，比如一些 timeout 命令到期的时候。

到此为止，第一个任务分析完毕，这样便建立起了基于 event 队列的消息循环，稍后便可以接收上层发来的的请求了（上层请求的 event 对象建立，在第三个任务中）。

（2）初始化 librefrence_ril.so，也就是与硬件或模拟硬件 modem 通信的部分（后面统一称硬件），通过 RIL_Init 函数完成。

此任务的入口是 RIL_Init，RIL_Init 首先通过参数获取硬件接口的设备文件或模拟硬件接口的 socket，然后新开一个线程继续初始化处理，即 mainLoop 循环处理。

mainLoop 的主要任务是建立起与硬件的通信，然后通过 read 方法阻塞等待硬件的主动上报或响应。在注册一些基础回调（timeout,readerclose）后，mainLoop 将首先打开硬件设备文件，建立起与硬件的通信，s_device_path 和 s_port 是前面获取的设备路径参数，在此将其打开。两者可以同时打开并拥有各自的 reader，由此可见很容易添加双卡双待等支持。

接下来通过 at_open()函数建立起这一设备文件上的 reader 等待循环，此功能是通过新创建一个线程的方式完成的，新创建线程的代码如下所示：

```
ret = pthread_create(&s_tid_reader, &attr, readerLoop, &attr)
```

此线程的入口点是 readerLoop。

因为 AT 命令都是以 rn 或 nr 的换行符来作为分隔符的，所以，readerLoop 是 line 驱动的，除非发生出错或超时，否则会读到一行完整的响应或主动上报时才会返回。这个循环运行起来以后，已经建立了基本的 AT 响应机制。

有了响应的机制后，就可以通过如下代码在 initializeCallback 中执行一些 Modem 的初始化命令，主要都是 AT 命令的方式：

```
RIL_requestTimedCallback(initializeCallback, NULL, &TIMEVAL_0)
```

（3）通过 RIL_Init 获取一组函数指针 RIL_RadioFunctions，并通过 RIL_register 完成注册，并打开接收上层命令的 socket 通道。

此任务是由 RIL_Init 的返回值开始的，这是一个 RIL_RadioFunctions 结构的指针。此指针的定义代码如下所示：

```
typedef struct {
int version;              /* 设置 RIL_VERSION */
RIL_RequestFunc onRequest;
RIL_RadioStateRequest onStateRequest;
RIL_Supports supports;
RIL_Cancel onCancel;
RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

上述指针中最重要的是 onRequest 域，上层来的请求都由这个函数进行映射后转换成对应的 AT 命令发送给硬件。rild 通过 RIL_register 注册这一指针。

在 RIL_register 中还需要完成另外一个任务，就是打开前面提到的与上层通信的 socket 接口（s_fdListen 是主接口，s_fdDebug 供调试时使用）。然后将这两个 socket 接口使用任务一中实现的机制进行注册（仅列出 s_fdListen）对应代码如下所示：

```
ril_event_set (&s_listen_event, s_fdListen, false,
listenCallback, NULL);
rilEventAddWakeup (&s_listen_event);
```

这样将两个 socket 加到任务一中建立起来多路复用 I/O 的检查句柄集合中，一旦有上层来的（调试）请求，event 机制便能响应处理了。到此为止，启动流程全部介绍完毕。

2. request 流程

（1）多路复用 I/O 机制的运转

request 接收是通过 ril_event_loop 中的多路复用 I/O 实现的，其中使用 ril_event_set 来配置一个 event，此处主要有如下两种 event。

- ril_event_add：添加使用多路 I/O 的 even，负责将其挂到队列，同时将 event 的通道句柄 fd 加入到 watch_table，然后通过 select 进行等待。
- ril_timer_add：添加 timer event，它将其挂在队列，同时重新计算最短超时时间。

无论是哪一种 add，最终都会调用 triggerEvLoop 来刷新队列，并更新超时值或等待对象。在刷新之后，ril_event_loop 从阻塞的位置使用 select 返回。此时只有两种可能：一是超时，二是等待到了某 I/O 操作。

- 超时处理：在 processTimeouts 中完成，需要“摘下”超时的 event 并将其加入 pending_list.。
- 检查有 I/O 操作的通道的处理：在 processReadReadies 中完成，需要将超时的 event 加入到 pending_list。

在 firePending 中检索 pending_list 的 event 并依次执行 event→func。当完成上述操作之后，计算新的超时时间，并重新 select（选择）阻塞的多路 I/O 通道。

在初始化完成以后，会在队列上挂了如下 3 个 event 对。

- s_listen_event：名为 rild 的 socket，主要使用 requeset & response 通道实现。
- s_debug_event：名为 rild-debug 的 socket，调试用 requeset & response 通道，其流程与 s_listen_event 基本相同。
- s_wakeupfd_event：是一个无名管道，用于队列主动唤醒。

（2）request 的传入和 dispatch

上层部分的核心代码保存在如下文件中：

```
frameworks/base/telephony/java/com/android/internal/telephony/gsm/RIL.java
```

此文件是 Android Java 框架处理 radio(gsm)的核心组件，首先看里面的函数 dial()，代码如下所示：

```
public void
dial (String address, int clirMode, Message result)
{
    RILRequest rr = RILRequest.obtain(RIL_REQUEST_DIAL, result);
    rr.mp.writeString(address);
    rr.mp.writeInt(clirMode);
    if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));
    send(rr);
}
```

在上述代码中，rr 是以 RIL_REQUEST_DIAL 为 request 号而申请的一个 RILRequest 对象，此 request 号在 Java 框架和 rild 库中共享。在 RILRequest 初始化的时候，会连接名为 rild 的 socket（也就是 rild 中 s_listen_event 绑定的 socket）。

在 Android 系统中，rr.mp 是一个 Parcel 对象，Parcel 是一套简单的序列化协议，用于将对象或对象的成员序列化成字节流，以供传递参数之用。在此可以看到 String address 和 int clirMode 都是将依次序列化的成员。在之前 rr 初始化的时候，request 号与 request 的序列号已经成为头两个将被序列化的成员，这为后面的 request 解析打下了基础。

send 到 handleMessage 的流程比较简单，send 会将 rr 直接传递给另一个线程的 handleMessage，handleMessage，目的是执行 data = rr.mp.marshall()序列化操作，并将 data 字节流写入到 rild socket。

如果此时返回 rild，select 会发现 rild socket 有了请求链接的信号，这会导致 s_listen_event 被挂入 pending_list，从而执行 event→func，即执行下面的代码：

```
static void listenCallback (int fd, short flags, void *param);
```

接下来运行下面的代码以获取传入的 socket 描述符：

```
s_fdCommand = accept(s_fdListen, (sockaddr *) &peeraddr, &socklen)
```

然后通过 record_stream_new 建立起一个 record_stream 将其与 s_fdCommand 绑定，在此无需关注 record_stream 的具体流程，只需关注 command event 回调和 processCommandsCallback()函数即可。从前面的 event 机制分析可以得出，一旦 s_fdCommand 上有数据，此回调函数就会被调用。

processCommandsCallback 通过 record_stream_get_next 阻塞读取 s_fdCommand 上发来的数据，一直到收到一个完整的 request（request 包的完整性由 record_stream 的机制保证）为止，然后将其送达 processCommandBuffer。

进入 processCommandBuffer 以后就说明正式进入了命令的解析部分，每个命令将以 RequestInfo 的形式存在。对应代码如下所示：

```
typedef struct RequestInfo {
int32_t token;
CommandInfo *pCI;
struct RequestInfo *p_next;
char cancelled;
char local;
 } RequestInfo;
```

此处的 pRI 是一个 RequestInfo 结构指针，在上层和 rild 之间的 request 号是统一的，在文件 ril.cpp 中定义了这个号。对应代码如下所示：

```
static CommandInfo s_commands[] = {
#include "ril_commands.h"
};
```

在定义时包含了一个 ril_commands.h 的枚举。pRI 直接访问数组 s_commands[]以获取自己的 pCI，这是一个 CommandInfo 结构，定义代码如下所示：

```
typedef struct {
int requestNumber;
void (*dispatchFunction) (Parcel &p, struct RequestInfo *pRI);
int(*responseFunction) (Parcel &p, void *response, size_t responselen);
} CommandInfo;
```

（3）request 的详细解析

对于 dial 而言，CommandInfo 结构的初始化是通过如下代码实现的：

```
{RIL_REQUEST_DIAL, dispatchDial, responseVoid},
```

在此执行了 dispatchFunction 之后，也就是 dispatchDial()函数。我们可以看到其实有很多种类的 dispatch function，比如 dispatchVoid、dispatchStrings、dispatchSIM_IO 等。这些函数的区别在于 Parcel 传入的参数形式，其中 Void 就是不带参数的，Strings 是以 string[]作参数。

当拥有 request 号和参数后，就可以进行具体的 request()函数调用了。是通过如下代码实现的：

```
s_callbacks.onRequest(pRI->pCI->requestNumber, xxx, len, pRI)
```

s_callbacks 是获取自 libreference-ril 的 RIL_RadioFunctions 结构指针，request 请求在这里转入底层的 libreference-ril 处理，handler 是 reference-ril.c 中的 onRequest。

onRequest 进行一个简单的 switch 分发，RIL_REQUEST_DIAL 的基本流程如下所示：

```
onRequest-->requestDial-->at_send_command-->at_send_command_full-->at_send_command_f
ull_nolock-->writeline
```

在 requestDial 中将命令和参数转换成对应的 AT 命令，然后调用公共 send command 接口 at_send_command。除了这个接口之外，还有如下常用的接口：

```
at_send_command_singleline
at_send_command_sms
at_send_command_multiline
```

接下来需要执行 at_send_command_full，前面的几个接口都会最终到这里为止，然后通过一个互斥的 at_send_command_full_nolock 调用来完成最终的写入操作。

3. response 流程

通过前面的 request 流程，终止了 at_send_command_full_nolock 里的 writeline 操作，因为这里完成命令写入到硬件设备的操作，接下来就是等待硬件响应，也就是 response 的过程了。

在实现 response 获取信息时，在 readerLoop 中用 readline()函数以“行”为单位接收信息。AT 主要有如下两种 response 方式。

- 主动上报：比如网络状态、短信和来电等都不需要经过请求，此方式用 unsolicited 来专门描述。
- 真正意义上的 response：即命令的响应。

此时可以看到所有的“行”都是经过 SMS 自动上报筛选的，因为短信的 AT 处理通常比较麻烦，无论收发都单独列出。这里是因为要即时处理这条短信消息（两行，标志＋pdu），而不能拆开处理。处理函数是 onUnsolicited()。

除了 SMS 特例，所有的 line 都要经过 processLine 处理，我们来看看下面的流程：

```
processLine
|----no cmd--->handleUnsolicited //主动上报
|----isFinalResponseSuccess--->handleFinalResponse //成功,标准响应
|----isFinalResponseError--->handleFinalResponse //失败，标准响应
|----get '>'--->send sms pdu //收到>符号，发送 sms 数据再继续等待响应
|----switch s_type--->具体响应  //命令有具体的响应信息需要对应分析
```

在此需要重点关注 handleUnsolicited 自动上报和 switch s_type 具体响应信息，另外，具体响应需要 handleFinalResponse 这样的标准响应来最终完成。

（1）onUnsolicite（主动上报响应），实现函数如下：

```
static void onUnsolicited (const char *s, const char *sms_pdu);
```

response 的主要的解析过程是由文件 at_tok.c 中的函数完成的，其实就是字符串按块解析，具体的解析方式由每条命令或上报信息自行决定。onUnsolicited 只解析出头部（一般是+XXXX 的形式），然后按类型决定下一步操作，操作方式有 RIL_onUnsolicitedResponse 和 RIL_requestTimed Callback 两种。

- RIL_onUnsolicitedResponse

将 unsolicited 信息直接返回给上层。通过 Parcel 传递，将 RESPONSE_UNSOLICITED，unsolResponse（request 号）写入 Parcel，然后通过 s_unsolResponses 数组，查找到对应的 response Function 完成进一步的解析，存入 Parcel 中。最终通过 sendResponse 将其传递回原进程。具体流程如下所示：

```
sendResponse-->sendResponseRaw-->blockingWrite-->write to s_fdCommand
```

- RIL_requestTimedCallback

通过 event 机制实现的 timer 机制，回调对应的内部处理函数。通过 internalRequestTimed Callback 将回调添加到 event 循环，最终完成 callback 上挂的函数的回调。例如 pollSIMState 和 onPDPContextListChanged 等回调不用返回上层，直接在内部处理就可以实现。

（2）switch s_type 命令的具体响应及 handleFinalResponse 标准响应

命令类型（s_type）在 send command 的时候设置，具体有 NO_RESULT、NUMERIC、SINGLELINE 和 MULTILINE 几种类型供不同的 AT 使用。这几个类型的解析方式类似，通过比较 AT 头标记等判断处理，如果是对应的响应，就通过 addIntermediate 挂到一个临时结果 sp_response→p_intermediates 队列里。如果不是对应响应，那应该是穿插其中的自动上报，用 onUnsolicite 来处理。具体响应只是起了一个获取响应信息到临时的结果，需要等待具体分析的作用。无论有无具体响应，最终都以标准响应 handleFinalResponse 来完成，也就是一直接收到 OK、ERROR 等标准 response 来结束，这是大多数 AT 命令的规范。

16.2 电话系统中的音频模块

在 Android 设备中进行通话时需要音频系统的支持，在建立通话模式时会调用音频系统实现无线通话功能。在本节的内容中，将详细讲解 Android 音频系统的基本知识。

16.2.1 音频系统结构

Android 音频系统对应的硬件设备有音频输入部分和音频输出两部分，手机中的输入设备通常是话筒，输出设备通常是耳机和扬声器。Android 音频系统的核心是 Audio 系统，它在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。Audio 部分作为 Android 的 Audio 系统的输入/输出层次，一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。

Audio 系统主要分成如下几个层次：

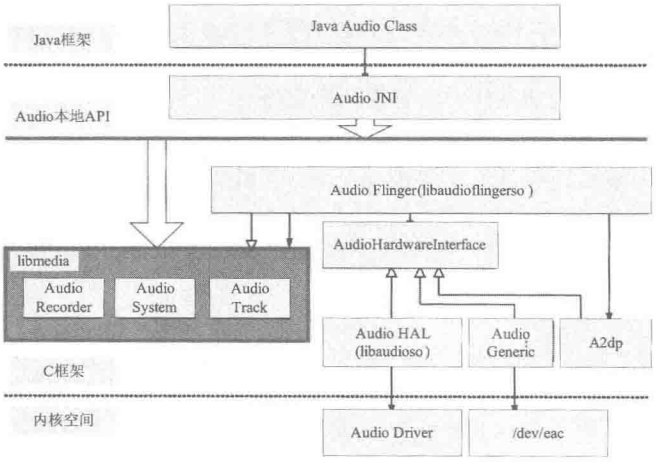
（1）Media 库提供的 Audio 系统本地部分接口；

（2）AudioFlinger 作为 Audio 系统的中间层；

（3）Audio 的硬件抽象层提供底层支持；

（4）Audio 接口通过 JNI 和 Java 框架提供给上层。

Android 音频系统的基本层次结构如图 16-4 所示。



▲图 16-4　Android 音频系统的框架结构

图 16-4 中各个构成部分的具体说明如下所示。

（1）Audio 的 Java 部分

Java 部分的代码路径是“frameworks/base/media/java/android/media”。

与 Audio 相关的 Java 包是“android.media”，在里面主要包含了和 AudioManager、Audio 等系统相关的类。

（2）Audio 的 JNI 部分

JNI 部分的代码路径是：“frameworks/base/core/jni”。

生成库是“libandroid_runtime.so”，Audio 的 JNI 是其中的一个部分。

（3）Audio 的框架部分

框架部分的头文件路径是“frameworks/base/include/media/”。

源代码路径是“frameworks/base/media/libmedia/”。

Audio 本地框架是 Media 库的一部分，本部分内容被编译成库 libmedia.so，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4）Audio Flinger

Flinger 部分的代码路径是“frameworks/base/libs/audioflinger”，此部分内容被编译成库 libaudioflinger.so，这是 Audio 系统的本地服务部分。

（5）Audio 的硬件抽象层接口

硬件抽象层接口的头文件路径是“hardware/libhardware_legacy/include/hardware/”。

Audio 硬件抽象层的实现在各个系统中可能是不同的，需要使用代码去继承相应的类并实现它们，作为 Android 系统本地框架层和驱动程序接口。

16.2.2　分析音频系统的层次

在 Android 中，Audio 系统从上到下分别由 Java 的 Audio 类、Audio 本地框架类、AudioFlinger 和 Audio 的硬件抽象层几个部分组成。

1. 层次说明

（1）Audio 本地框架类：是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

（2）AudioFlinger：继承了 libmeida 里面的接口，提供实现库 libaudiofilnger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so。

（3）JNI：在 Audio 系统中，使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接口来实现。

（4）Audio 硬件抽象层：提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要关注和独立完成的部分。

因为 Android 中的 Audio 系统不涉及编解码环节，只负责上层系统和底层 Audio 硬件的交互，所以，通常以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和输出/输入两个类来表示整个 Audio 系统，输出/输入两个类负责数据通道。在各个层次之间具有对应关系，如表 16-1 所示。

表 16-1　　Android 各个层次的对应关系

层次说明	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media. AudioSystem	android.media AudioTrack	android.media AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

2. Media 库中的 Audio 框架

在 Media 库中提供了 Android 的 Audio 系统的核心框架，在库中实现了 AudioSystem、AudioTrack 和 AudioRecorder 3 个类。另外，还提供了 IAudioFlinger 类接口，通过此类可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件被保存在“frameworks/base/include/media/”目录中，其中包含了如下头文件。

- AudioSystem.h：media 库的 Audio 部分对上层的总管接口。
- IAudioFlinger.h：需要下层实现的总管接口。
- AudioTrack.h：放音部分对上接口。
- IAudioTrack.h：放音部分需要下层实现的接口。
- AudioRecorder.h：录音部分对上接口。
- IAudioRecorder.h：录音部分需要下层实现的接口。

其中文件 IAudioFlinger.h、IAudioTrack.h 和 IAudioRecorder.h 的接口是通过下层的继承来实现的。文件 AudioFlinger.h、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，它们既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。

从具体从功能上看，AudioSystem 用于综合管理 Audio 系统，而 AudioTrack 和 AudioRecorder 分别负责输出和输入音频数据，即分别实现播放和录制功能。

在文件 AudioSystem.h 中定义了枚举值和 set/get 等一系列接口，主要代码如下所示：

```
class AudioSystem
{
public:
    enum stream_type {                    // Audio 流的类型
        SYSTEM          = 1,
        RING            = 2,
        MUSIC           = 3,
        ALARM           = 4,
        NOTIFICATION    = 5,
        BLUETOOTH_SCO   = 6,
        ENFORCED_AUDIBLE = 7,
        NUM_STREAM_TYPES
    };
```

```
    enum audio_output_type {            // Audio 数据输出类型
      AUDIO_OUTPUT_DEFAULT     =-1,
        AUDIO_OUTPUT_HARDWARE     = 0,
        AUDIO_OUTPUT_A2DP         = 1,
        NUM_AUDIO_OUTPUT_TYPES
    };
    enum audio_format {                 // Audio 数据格式
        FORMAT_DEFAULT = 0,
        PCM_16_BIT,
        PCM_8_BIT,
        INVALID_FORMAT
    };
    enum audio_mode {                       // Audio 模式
       MODE_INVALID = -2,
        MODE_CURRENT = -1,
        MODE_NORMAL = 0,
        MODE_RINGTONE,
        MODE_IN_CALL,
        NUM_MODES  // not a valid entry, denotes end-of-list
    };
    enum audio_routes {                 // Audio 路径类型
        ROUTE_EARPIECE          = (1 << 0),
        ROUTE_SPEAKER           = (1 << 1),
        ROUTE_BLUETOOTH_SCO   = (1 << 2),
        ROUTE_HEADSET           = (1 << 3),
        ROUTE_BLUETOOTH_A2DP  = (1 << 4),
        ROUTE_ALL                 = -1UL,
    };
       enum audio_in_acoustics {
        AGC_ENABLE    = 0x0001,
        AGC_DISABLE   = 0,
        NS_ENABLE     = 0x0002,
        NS_DISABLE    = 0,
        TX_IIR_ENABLE = 0x0004,
        TX_DISABLE    = 0
    };
    static status_t speakerphone(bool state);
    static status_t isSpeakerphoneOn(bool* state);
    static status_t bluetoothSco(bool state);
    static status_t isBluetoothScoOn(bool* state);
    static status_t muteMicrophone(bool state);
    static status_t isMicrophoneMuted(bool *state);
    static status_t setMasterVolume(float value);
    static status_t setMasterMute(bool mute);
    static status_t getMasterVolume(float* volume);
    static status_t getMasterMute(bool* mute);
    static status_t setStreamVolume(int stream, float value);
    static status_t setStreamMute(int stream, bool mute);
    static status_t getStreamVolume(int stream, float* volume);
    static status_t getStreamMute(int stream, bool* mute);
    static status_t setMode(int mode);
    static status_t getMode(int* mode);
    static status_t setRouting(int mode, uint32_t routes, uint32_t mask);
    static status_t getRouting(int mode, uint32_t* routes);
    static status_t isMusicActive(bool *state);
    static status_t setParameter(const char* key, const char* value);
    static void setErrorCallback(audio_error_callback cb);
    static const sp<IAudioFlinger>& get_audio_flinger();
    static float linearToLog(int volume);
    static int logToLinear(float volume);
    static status_t getOutputSamplingRate(int* samplingRate, int stream = DEFAULT);
    static status_t getOutputFrameCount(int* frameCount, int stream = DEFAULT);
    static status_t getOutputLatency(uint32_t* latency, int stream = DEFAULT);
    static bool routedToA2dpOutput(int streamType);
    static status_t getInputBufferSize(uint32_t sampleRate, int format, int channelCount,
    size_t* buffSize);
};
```

在上述枚举值中，是用单独的位来表示 audio_routes，而不是用顺序的枚举值来表示，所以，在使用这个值的过程中可以使用“或”的方式。例如，表示声音既可以从耳机（EARPIECE）输

出，也可以从扬声器（SPEAKER）输出。上述功能是否能够实现，是由下层提供支持的。在这个类中，set/get 等接口控制的也是相关的内容，例如 Audio 声音大小、Audio 模式和路径等。

AudioTrack 是 Audio 输出环节的类，在里面包含了最重要的接口 write()，主要代码如下所示：

```
class AudioTrack
{
    typedef void (*callback_t)(int event,
void* user, void *info);
    AudioTrack( int streamType,
                uint32_t sampleRate  = 0,      // 音频的采样律
                int format          = 0,       // 音频的格式（例如 8 位或者 16 位的 PCM）
                int channelCount     = 0,      // 音频的通道数
                int frameCount       = 0,      // 音频的帧数
                uint32_t flags       = 0,
                callback_t cbf       = 0,
                void* user           = 0,
                int notificationFrames = 0);
    void        start();
    void        stop();
    void        flush();
    void        pause();
    void        mute(bool);
    ssize_t     write(const void* buffer, size_t size);
```

类 AudioRecord 是用于实现和 Audio 输入相关的功能，其中最重要的功能是通过接口函数 read()实现的，主要代码如下所示：

```
class AudioRecord
{
public:
    AudioRecord(int streamType,
                uint32_t sampleRate  = 0,      // 音频的采样律
                int format          = 0,       // 音频的格式（如 8 位或者 16 位的 PCM）
                int channelCount     = 0,      // 音频的通道数
                int frameCount       = 0,      // 音频的帧数
                uint32_t flags      = 0,
                callback_t cbf = 0,
                void* user = 0,
                int notificationFrames = 0);
    status_t    start();
    status_t    stop();
    ssize_t     read(void* buffer, size_t size);
```

在类 AudioTrack 和 AudioRecord 中，函数 read()和 write()的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（PCM 数据）。这两个类还涉及 Auido 数据格式、通道数、帧数目等参数，不但可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

另外，在 libmedia 库中提供的只是一个 Audio 系统框架，其中类 AudioSystem、AudioTrack 和 AudioRecord 分别调用下层的接口 IAudioFlinger、IAudioTrack 和 IAudioRecord 来实现。另外的一个接口是 IAudioFlingerClient，它作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IAudioFlinger 运行时信息。

3. 本地代码

在 Android 系统中，AudioFlinger 是 Audio 音频系统的中间层，能够作为 libmedia 提供的 Audio 部分接口的实现。这部分本地代码的路径如下：

```
frameworks/base/libs/audioflinger
```

文件 AudioFlinger.h 和 AudioFlinger.cpp 是实现 AudioFlinger 的核心文件，在里面提供了类 AudioFlinger，此类是一个 IAudioFlinger 的实现，其接口代码如下所示：

```
class AudioFlinger : public BnAudioFlinger,
public IBinder::DeathRecipient
{
public:
```

```
    static void instantiate();
    virtual     status_t    dump(int fd, const Vector<String16>& args);
    virtual sp<IAudioTrack> createTrack(
// 获得音频输出接口（Track）
                            pid_t pid,
                            int streamType,
                            uint32_t sampleRate,
                            int format,
                            int channelCount,
                            int frameCount,
                            uint32_t flags,
                            const sp<IMemory>& sharedBuffer,
                            status_t *status);
    virtual     uint32_t    sampleRate(int output) const;
    virtual     int         channelCount(int output) const;
    virtual     int         format(int output) const;
    virtual     size_t      frameCount(int output) const;
    virtual     uint32_t    latency(int output) const;
    virtual     status_t    setMasterVolume(float value);
    virtual     status_t    setMasterMute(bool muted);
    virtual     status_t    setStreamVolume(int stream, float value);
    virtual     status_t    setStreamMute(int stream, bool muted);
    virtual     status_t    setRouting(int mode, uint32_t routes, uint32_t mask);
    virtual     uint32_t    getRouting(int mode) const;
    virtual     status_t    setMode(int mode);
    virtual     int         getMode() const;
    virtual sp<IAudioRecord> openRecord(
// 获得音频输出接口（Record）
                            pid_t pid,
                            int streamType,
                            uint32_t sampleRate,
                            int format,
                            int channelCount,
                            int frameCount,
                            uint32_t flags,
                            status_t *status);
```

由上述代码可以看出，AudioFlinger 使用函数 createTrack()来创建音频的输出设备 IAudioTrack，使用函数 openRecord()来创建音频的输入设备 IAudioRecord。并且还使用接口“get/set”来实现控制功能。

构造函数 AudioFlinger()的代码如下所示：

```
AudioFlinger::AudioFlinger()
{
    mHardwareStatus = AUDIO_HW_IDLE;
    mAudioHardware = AudioHardwareInterface::create();
    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
        mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
        status_t status;
        AudioStreamOut *hwOutput =
            mAudioHardware->openOutputStream (AudioSystem::PCM_16_BIT, 0, 0, &status);
        mHardwareStatus = AUDIO_HW_IDLE;
        if (hwOutput) {
            mHardwareMixerThread =
            new MixerThread(this, hwOutput, AudioSystem::AUDIO_OUTPUT_HARDWARE);
        } else {
            LOGE("Failed to initialize hardware  output stream, status: %d", status);
        }
    #ifdef WITH_A2DP
        mA2dpAudioInterface = new A2dpAudioInterface();
        AudioStreamOut *a2dpOutput = mA2dpAudioInterface->openOutputStream(AudioSystem::
        PCM_16_BIT, 0, 0, &status);
        if (a2dpOutput) {
            mA2dpMixerThread = new MixerThread(this, a2dpOutput, AudioSystem::AUDIO_
            OUTPUT_A2DP);
            if (hwOutput) {
                uint32_t frameCount = ((a2dpOutput->bufferSize()/a2dpOutput->frameSize())
                * hwOutput->sampleRate()) / a2dpOutput->sampleRate();
                MixerThread::OutputTrack *a2dpOutTrack = new MixerThread::OutputTrack
```

```
                (mA2dpMixerThread,
                hwOutput->sampleRate(),
                AudioSystem::PCM_16_BIT,
                hwOutput->channelCount(),
                frameCount);
                mHardwareMixerThread->setOuputTrack(a2dpOutTrack);
            }
        } else {
            LOGE("Failed to initialize A2DP output stream, status: %d", status);
        }
#endif
        setRouting(AudioSystem::MODE_NORMAL, AudioSystem::ROUTE_SPEAKER, AudioSystem::
        ROUTE_ALL);
        setRouting(AudioSystem::MODE_RINGTONE, AudioSystem::ROUTE_SPEAKER, AudioSystem::
        ROUTE_ALL);
        setRouting(AudioSystem::MODE_IN_CALL, AudioSystem::ROUTE_EARPIECE, AudioSystem::
        ROUTE_ALL);
        setMode(AudioSystem::MODE_NORMAL);
        setMasterVolume(1.0f);
        setMasterMute(false);
        mAudioRecordThread = new AudioRecordThread(mAudioHardware, this);
        if (mAudioRecordThread != 0) {
            mAudioRecordThread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);
        }
    } else {
        LOGE("Couldn't even initialize the stubbed audio hardware!");
    }
}
```

由上述代码可以看出，在初始化 AudioFlinger 之后，会首先获得放音设备，然后为混音器（Mixer）建立线程并建立放音设备线程，最后在线程中获得放音设备。

在文件 AudioResampler.h 中定义了类 AudioResampler，此类是一个音频重取样器的工具类，定义代码如下所示：

```
class AudioResampler {
public:
    enum src_quality {
        DEFAULT=0,
        LOW_QUALITY=1,              // 线性差值算法
        MED_QUALITY=2,              // 立方差值算法
        HIGH_QUALITY=3              // fixed multi-tap FIR 算法
    };
    static AudioResampler* create(int bitDepth,
    int inChannelCount, // 静态地创建函数
            int32_t sampleRate, int quality=DEFAULT);
    virtual ~AudioResampler();
    virtual void init() = 0;
    virtual void setSampleRate(int32_t inSampleRate);
// 设置重采样率
    virtual void setVolume(int16_t left, int16_t right);
// 设置音量
    virtual void resample(int32_t* out, size_t outFrameCount,
                    AudioBufferProvider* provider) = 0;
};
```

在上述音频重取样工具类中，包含了如下 3 种质量。

- 低等质量（LOW_QUALITY）：使用线性差值算法实现。
- 中等质量（MED_QUALITY）：使用立方差值算法实现。
- 高等质量（HIGH_ QUALITY）：使用 FIR（有限阶滤波器）实现。

在 AudioResampler 中，AudioResamplerOrder1 是线性实现，AudioResamplerCubic.*文件提供立方实现方式，AudioResamplerSinc.*提供 FIR 实现。

通过文件 AudioMixer.h 和 AudioMixer.cpp 实现了一个 Audio 系统混音器，它被 AudioFlinger 调用，一般用于在声音输出之前的处理，提供多通道处理、声音缩放、重取样。AudioMixer 调用了 AudioResampler。

4. JNI 代码

在 Android 中的 Audio 系统中，通过 JNI 向 Java 层提供功能强大的接口，这样就可以在 Java 层通过 JNI 接口完成 Audio 系统的大部分操作。

Audio JNI 的实现代码保存在“frameworks/base/core/jni”目录下，在目录中主要有 3 个核心文件，这 3 个它们分别对应了 Android Java 框架中的 3 个类的支持，这 3 个文件的具体说明如下所示。

- android.media.AudioSystem：负责 Audio 系统的总体控制。
- android.media.AudioTrack：负责 Audio 系统的输出环节。
- android.media.AudioRecorder：负责 Audio 系统的输入环节。

在 Android 的 Java 层中，可以对 Audio 系统进行控制和数据流操作，其中控制操作和底层的处理基本一致。对于数据流操作来说，由于 Java 不支持指针，因此，接口被封装成了另外的形式。例如，在音频输出功能中，通过文件 android_media_AudioTrack.cpp 提供了写字节和写短整型的接口类型。对应代码如下所示：

```
static jint android_media_AudioTrack_native_
write(JNIEnv *env,  jobject thiz,
jbyteArray javaAudioData,
jint offsetInBytes, jint sizeInBytes,
jint javaAudioFormat) {
    jbyte* cAudioData = NULL;
    AudioTrack *lpTrack = NULL;
    lpTrack = (AudioTrack *)env->GetIntField(
              thiz, javaAudioTrackFields. Native TrackInJavaObj);
    ssize_t written = 0;
    if (lpTrack->sharedBuffer() == 0) {
    //进行写操作
        written = lpTrack->write(cAudioData + offsetInBytes, sizeInBytes);
    } else {
        if (javaAudioFormat == javaAudioTrackFields.PCM16) {
            memcpy(lpTrack->sharedBuffer()->pointer(),
                   cAudioData+offsetInBytes, sizeInBytes);
            written = sizeInBytes;
        } else if (javaAudioFormat == javaAudioTrackFields.PCM8) {
            int count = sizeInBytes;
            int16_t *dst = (int16_t *)lpTrack->sharedBuffer()->pointer();
            const int8_t *src = (const int8_t *)(cAudioData + offsetInBytes);
            while(count--) {
                *dst++ = (int16_t)(*src++^0x80) << 8;
            }
            written = sizeInBytes;
        }
    }
    env->ReleasePrimitiveArrayCritical(javaAudioData, cAudioData, 0);
    return (int)written;
}
```

5. Java 代码

在 Android 的 Audio 系统中，和 Java 相关的类定义在包 android.media 中，Java 部分的代码保存在“frameworks/base/media/java/android/media”目录中，在里面主要实现了如下类：

- android.media.AudioSystem；
- android.media. Audio Track；
- android.media.AudioRecorder；
- android.media.AudioFormat。

其中前 3 个类和本地代码是对应的，在 AudioFormat 中提供了一些和 Audio 相关的枚举值。在此需要注意的是，在 Audio 系统的 Java 代码中，虽然可以通过 AudioTrack 和 AudioRecorder 的 write()和 read()接口在 Java 层对 Audio 的数据流进行操作。但是，更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的输入/输出，而 Java 层只进行控制类操作，不处理数据流。

16.3 分析拨号流程

在通话过程中，包括拨号处理和被动接电话处理两部分。在本节的内容中，将详细讲解 Android 5.0 源代码中实现拨号处理功能的基本流程。

16.3.1 拨号界面

在 Android 5.0 源代码中，实现拨号界面的入口类实现文件是 packages\apps\Dialer\src\com\android\dialer\DialtactsActivity.java，在其 onCreate()方法中通过 setupDalerTab()、setupCallLogTab()、setupContactsTab 和 setupFavoritestTab()函数构建了以“tabHost”形式显示 4 个部分的主界面入口。具体实现代码如下所示：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mFirstLaunch = true;

    final Resources resources = getResources();
    mActionBarHeight = resources.getDimensionPixelSize(R.dimen.action_bar_height_large);

    setContentView(R.layout.dialtacts_activity);
    getWindow().setBackgroundDrawable(null);

    final ActionBar actionBar = getActionBar();
    actionBar.setCustomView(R.layout.search_edittext);
    actionBar.setDisplayShowCustomEnabled(true);
    actionBar.setBackgroundDrawable(null);

    mActionBarController = new ActionBarController(this,
            (SearchEditTextLayout) actionBar.getCustomView());

    SearchEditTextLayout searchEditTextLayout =
            (SearchEditTextLayout) actionBar.getCustomView();
    searchEditTextLayout.setPreImeKeyListener(mSearchEditTextLayoutListener);

    mSearchView = (EditText) searchEditTextLayout.findViewById(R.id.search_view);
    mSearchView.addTextChangedListener(mPhoneSearchQueryTextListener);
    mVoiceSearchButton = searchEditTextLayout.findViewById(R.id.voice_search_button);
    searchEditTextLayout.findViewById(R.id.search_magnifying_glass)
            .setOnClickListener(mSearchViewOnClickListener);
    searchEditTextLayout.findViewById(R.id.search_box_start_search)
            .setOnClickListener(mSearchViewOnClickListener);
    searchEditTextLayout.setOnBackButtonClickedListener(new OnBackButtonClicked Listener(){
        @Override
        public void onBackButtonClicked() {
            onBackPressed();
        }
    });

    mIsLandscape = getResources().getConfiguration().orientation
            == Configuration.ORIENTATION_LANDSCAPE;

    final View floatingActionButtonContainer = findViewById(
            R.id.floating_action_button_container);
    ImageButton floatingActionButton=(ImageButton)findViewById(R.id.floating_action_button);
    floatingActionButton.setOnClickListener(this);
    mFloatingActionButtonController = new FloatingActionButtonController(this,
            floatingActionButtonContainer, floatingActionButton);

    ImageButton optionsMenuButton = (ImageButton)
            searchEditTextLayout.findViewById(R.id.dialtacts_options_menu_button);
    optionsMenuButton.setOnClickListener(this);
    mOverflowMenu = buildOptionsMenu(searchEditTextLayout);
    optionsMenuButton.setOnTouchListener(mOverflowMenu.getDragToOpenListener());
```

```
    // Add the favorites fragment, and the dialpad fragment, but only if savedInstanceState
    // is null. Otherwise the fragment manager takes care of recreating these fragments.
    if (savedInstanceState == null) {
        getFragmentManager().beginTransaction()
                .add(R.id.dialtacts_frame, new ListsFragment(), TAG_FAVORITES_FRAGMENT)
                .add(R.id.dialtacts_container, new DialpadFragment(), TAG_DIALPAD_FRAGMENT)
                .commit();
    } else {
        mSearchQuery = savedInstanceState.getString(KEY_SEARCH_QUERY);
        mInRegularSearch = savedInstanceState.getBoolean(KEY_IN_REGULAR_SEARCH_UI);
        mInDialpadSearch = savedInstanceState.getBoolean(KEY_IN_DIALPAD_SEARCH_UI);
        mFirstLaunch = savedInstanceState.getBoolean(KEY_FIRST_LAUNCH);
        mShowDialpadOnResume = savedInstanceState.getBoolean(KEY_IS_DIALPAD_SHOWN);
        mActionBarController.restoreInstanceState(savedInstanceState);
    }

    final boolean isLayoutRtl = DialerUtils.isRtl();
    if (mIsLandscape) {
        mSlideIn = AnimationUtils.loadAnimation(this,
                isLayoutRtl ? R.anim.dialpad_slide_in_left : R.anim.dialpad_slide_in_ right);
        mSlideOut = AnimationUtils.loadAnimation(this,
                isLayoutRtl ? R.anim.dialpad_slide_out_left : R.anim.dialpad_slide_out_ right);
    } else {
        mSlideIn = AnimationUtils.loadAnimation(this, R.anim.dialpad_slide_in_bottom);
        mSlideOut = AnimationUtils.loadAnimation(this, R.anim.dialpad_slide_out_bottom);
    }

    mSlideIn.setInterpolator(AnimUtils.EASE_IN);
    mSlideOut.setInterpolator(AnimUtils.EASE_OUT);

    mSlideOut.setAnimationListener(mSlideOutListener);

    mParentLayout = (FrameLayout) findViewById(R.id.dialtacts_mainlayout);
    mParentLayout.setOnDragListener(new LayoutOnDragListener());
    floatingActionButtonContainer.getViewTreeObserver().addOnGlobalLayoutListener(
            new ViewTreeObserver.OnGlobalLayoutListener() {
                @Override
                public void onGlobalLayout() {
                    final ViewTreeObserver observer =
                            floatingActionButtonContainer.getViewTreeObserver();
                    if (!observer.isAlive()) {
                        return;
                    }
                    observer.removeOnGlobalLayoutListener(this);
                    int screenWidth = mParentLayout.getWidth();
                    mFloatingActionButtonController.setScreenWidth(screenWidth);
                    updateFloatingActionButtonControllerAlignment(false /* animate */);
                }
            });

    setupActivityOverlay();

    mDialerDatabaseHelper = DatabaseHelperManager.getDatabaseHelper(this);
    SmartDialPrefix.initializeNanpSettings(this);
}
```

拨号菜单界面的实现文件是 packages\apps\Dialer\src\com\android\dialer\dialpad\DialpadFragment.java，此文件提供了一个拨号键盘人机交互界面，通过拨号按钮单击事件的响应函数实现拨号逻辑处理，具体实现代码如下所示：

```
private void handleDialButtonPressed() {
    if (isDigitsEmpty()) { // No number entered.
        handleDialButtonClickWithEmptyDigits();
    } else {
        final String number = mDigits.getText().toString();

        // "persist.radio.otaspdial" is a temporary hack needed for one carrier's automated
        // test equipment.
        // TODO: clean it up.
        if (number != null
```

```
                && !TextUtils.isEmpty(mProhibitedPhoneNumberRegexp)
                && number.matches(mProhibitedPhoneNumberRegexp)) {
            Log.i(TAG, "The phone number is prohibited explicitly by a rule.");
            if (getActivity() != null) {
                DialogFragment dialogFragment = ErrorDialogFragment.newInstance(
                        R.string.dialog_phone_call_prohibited_message);
                dialogFragment.show(getFragmentManager(), "phone_prohibited_dialog");
            }

            // Clear the digits just in case.
            clearDialpad();
        } else {
            final Intent intent = CallUtil.getCallIntent(number,
                    (getActivity() instanceof DialtactsActivity ?
                            ((DialtactsActivity) getActivity()).getCallOrigin() : null));
            DialerUtils.startActivityWithErrorToast(getActivity(), intent);
            hideAndClearDialpad(false);
        }
    }
}
```

文件 DialpadFragment.java 的布局文件是 dialpad_fragment.xml，具体实现代码如下所示：

```
<view class="com.android.dialer.dialpad.DialpadFragment$DialpadSlidingRelativeLayout"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

        <!-- spacer view -->
        <View
            android:id="@+id/spacer"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:background="#00000000" />
        <!-- Dialpad shadow -->
        <View
            android:layout_width="match_parent"
            android:layout_height="@dimen/shadow_length"
            android:background="@drawable/shadow_fade_up" />
        <include layout="@layout/dialpad_view" />
        <!-- "Dialpad chooser" UI, shown only when the user brings up the
                Dialer while a call is already in progress.
                When this UI is visible, the other Dialer elements
                (the textfield/button and the dialpad) are hidden. -->
        <ListView android:id="@+id/dialpadChooser"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:background="@color/background_dialer_light"
                android:visibility="gone" />

    </LinearLayout>

    <!-- Margin bottom and alignParentBottom don't work well together, so use a Space
instead. -->
    <Space android:id="@+id/dialpad_floating_action_button_margin_bottom"
        android:layout_width="match_parent"
        android:layout_height="@dimen/floating_action_button_margin_bottom"
        android:layout_alignParentBottom="true" />

    <FrameLayout
        android:id="@+id/dialpad_floating_action_button_container"
        android:background="@drawable/fab_green"
        android:layout_width="@dimen/floating_action_button_width"
        android:layout_height="@dimen/floating_action_button_height"
        android:layout_above="@id/dialpad_floating_action_button_margin_bottom"
```

```
        android:layout_centerHorizontal="true">

        <ImageButton
            android:id="@+id/dialpad_floating_action_button"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@drawable/floating_action_button"
            android:contentDescription="@string/description_dial_button"
            android:src="@drawable/fab_ic_call"/>

    </FrameLayout>

</view>
```

▲图 16-5　拨号界面

拨号界面的效果如图 16-5 所示。

16.3.2　实现 Phone 应用

文件 packages\services\Telephony\src\com\android\phone\OutgoingCallBroadcaster.java 是发起呼叫的唯一入口，功能是对呼叫信息进行统一验证，可以从 SMS 或 Constants 等位置发起呼叫。如果广播还未被取消，终止 NEW_OUTGOING_CALL 广播，启动 InCallScreen，并带有已经被修改的号码或其他提供的信息。文件 OutgoingCallBroadcaster.java 的具体实现过程如下所示。

（1）在函数 onCreate()中获取传入的 intent 意图，具体实现代码如下所示：

```
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.outgoing_call_broadcaster);
    mWaitingSpinner = (ProgressBar) findViewById(R.id.spinner);
    //获取传入的 intent 意图
    Intent intent = getIntent();
    if (DBG) {
        final Configuration configuration = getResources().getConfiguration();
        Log.v(TAG, "onCreate: this = " + this + ", icicle = " + icicle);
        Log.v(TAG, " - getIntent() = " + intent);
        Log.v(TAG, " - configuration = " + configuration);
    }

    if (icicle != null) {
        Log.i(TAG, "onCreate: non-null icicle!  "
              + "Bailing out, not sending NEW_OUTGOING_CALL broadcast...");

        return;
    }

    processIntent(intent);
    if (DBG) Log.v(TAG, "At the end of onCreate(). isFinishing(): " + isFinishing());
}
```

（2）在获取到 intent 后，在函数 processIntent 中通过 intent 获取 action 和电话号码，具体实现代码如下所示：

```
private void processIntent(Intent intent) {
    if (DBG) {
        Log.v(TAG, "processIntent() = " + intent + ", thread: " + Thread.currentThread());
    }
    final Configuration configuration = getResources().getConfiguration();

    // Outgoing phone calls are only allowed on "voice-capable" devices.
    if (!PhoneGlobals.sVoiceCapable) {
        Log.i(TAG, "This device is detected as non-voice-capable device.");
        handleNonVoiceCapable(intent);
        return;
    }

    String action = intent.getAction();
    String number = PhoneNumberUtils.getNumberFromIntent(intent, this);
    // Check the number, don't convert for sip uri
```

```
// TODO put uriNumber under PhoneNumberUtils
if (number != null) {
    if (!PhoneNumberUtils.isUriNumber(number)) {
        number = PhoneNumberUtils.convertKeypadLettersToDigits(number);
        number = PhoneNumberUtils.stripSeparators(number);
    }
} else {
    Log.w(TAG, "The number obtained from Intent is null.");
}

AppOpsManager appOps = (AppOpsManager)getSystemService(Context.APP_OPS_SERVICE);
int launchedFromUid;
String launchedFromPackage;
try {
    launchedFromUid = ActivityManagerNative.getDefault().getLaunchedFromUid(
            getActivityToken());
    launchedFromPackage = ActivityManagerNative.getDefault().getLaunchedFromPackage(
            getActivityToken());
} catch (RemoteException e) {
    launchedFromUid = -1;
    launchedFromPackage = null;
}
if(appOps.noteOpNoThrow(AppOpsManager.OP_CALL_PHONE,launchedFromUid,launchedFromPackage)
        != AppOpsManager.MODE_ALLOWED) {
    Log.w(TAG, "Rejecting call from uid " + launchedFromUid + " package "
            + launchedFromPackage);
    finish();
    return;
}
boolean callNow;

if (getClass().getName().equals(intent.getComponent().getClassName())) {
    if (!Intent.ACTION_CALL.equals(intent.getAction())) {
        Log.w(TAG, "Attempt to deliver non-CALL action; forcing to CALL");
        intent.setAction(Intent.ACTION_CALL);
    }
}
final boolean isExactEmergencyNumber =
        (number != null) && PhoneNumberUtils.isLocalEmergencyNumber(this, number);
final boolean isPotentialEmergencyNumber =
    (number != null) && PhoneNumberUtils.isPotentialLocalEmergencyNumber(this, number);
if (VDBG) {
    Log.v(TAG, " - Checking restrictions for number '" + number + "':");
    Log.v(TAG, "     isExactEmergencyNumber     = " + isExactEmergencyNumber);
    Log.v(TAG, "     isPotentialEmergencyNumber = " + isPotentialEmergencyNumber);
}

/* Change CALL_PRIVILEGED into CALL or CALL_EMERGENCY as needed. */
// TODO: This code is redundant with some code in InCallScreen: refactor.
if (Intent.ACTION_CALL_PRIVILEGED.equals(action)) {
    if (isPotentialEmergencyNumber) {
        Log.i(TAG, "ACTION_CALL_PRIVILEGED is used while the number is a potential"
            + " emergency number. Use ACTION_CALL_EMERGENCY as an action instead.");
        action = Intent.ACTION_CALL_EMERGENCY;
    } else {
        action = Intent.ACTION_CALL;
    }
    if (DBG) Log.v(TAG, " - updating action from CALL_PRIVILEGED to " + action);
    intent.setAction(action);
}

if (Intent.ACTION_CALL.equals(action)) {
    if (isPotentialEmergencyNumber) {
        Log.w(TAG, "Cannot call potential emergency number '" + number
                + "' with CALL Intent " + intent + ".");
        Log.i(TAG, "Launching default dialer instead...");

        Intent invokeFrameworkDialer = new Intent();
        final Resources resources = getResources();
        invokeFrameworkDialer.setClassName(
                resources.getString(R.string.ui_default_package),
```

```
                resources.getString(R.string.dialer_default_class));
            invokeFrameworkDialer.setAction(Intent.ACTION_DIAL);
            invokeFrameworkDialer.setData(intent.getData());
            if (DBG) Log.v(TAG, "onCreate(): calling startActivity for Dialer: "
                        + invokeFrameworkDialer);
            startActivity(invokeFrameworkDialer);
            finish();
            return;
        }
        callNow = false;
    } else if (Intent.ACTION_CALL_EMERGENCY.equals(action)) {
        if (!isPotentialEmergencyNumber) {
            Log.w(TAG, "Cannot call non-potential-emergency number " + number
                    + " with EMERGENCY_CALL Intent " + intent + "."
                    + " Finish the Activity immediately.");
            finish();
            return;
        }
        callNow = true;
    } else {
        Log.e(TAG, "Unhandled Intent " + intent + ". Finish the Activity immediately.");
        finish();
        return;
    }
    PhoneGlobals.getInstance().wakeUpScreen();
    if (TextUtils.isEmpty(number)) {
        if (intent.getBooleanExtra(EXTRA_SEND_EMPTY_FLASH, false)) {
            Log.i(TAG, "onCreate: SEND_EMPTY_FLASH...");
            PhoneUtils.sendEmptyFlash(PhoneGlobals.getPhone());
            finish();
            return;
        } else {
            Log.i(TAG, "onCreate: null or empty number, setting callNow=true...");
            callNow = true;
        }
    }

    if (callNow) {
    }
    Uri uri = intent.getData();
    String scheme = uri.getScheme();
    if (PhoneAccount.SCHEME_SIP.equals(scheme) || PhoneNumberUtils.isUriNumber(number)) {
        Log.i(TAG, "The requested number was detected as SIP call.");
        startSipCallOptionHandler(this, intent, uri, number);
        finish();
        return;
    }

    Intent broadcastIntent = new Intent(Intent.ACTION_NEW_OUTGOING_CALL);
    if (number != null) {
        broadcastIntent.putExtra(Intent.EXTRA_PHONE_NUMBER, number);
    }
    CallGatewayManager.checkAndCopyPhoneProviderExtras(intent, broadcastIntent);
    broadcastIntent.putExtra(EXTRA_ALREADY_CALLED, callNow);
    broadcastIntent.putExtra(EXTRA_ORIGINAL_URI, uri.toString());
    // Need to raise foreground in-call UI as soon as possible while allowing 3rd party app
    // to intercept the outgoing call.
    broadcastIntent.addFlags(Intent.FLAG_RECEIVER_FOREGROUND);
    if (DBG) Log.v(TAG, " - Broadcasting intent: " + broadcastIntent + ".");
    mHandler.sendEmptyMessageDelayed(EVENT_OUTGOING_CALL_TIMEOUT,
            OUTGOING_CALL_TIMEOUT_THRESHOLD);
    sendOrderedBroadcastAsUser(broadcastIntent, UserHandle.OWNER,
            PERMISSION, new OutgoingCallReceiver(),
            null,  // scheduler
            Activity.RESULT_OK,  // initialCode
            number,  // initialData: initial value for the result data
            null);  // initialExtras
}
```

通过上述代码可知，在获取到号码后通过“isPotentialEmergencyNumber”判断这个号码是否为紧急号码。如果为紧急号码，则直接进入 InCallScreen 界面，如果不是紧急号码，则通过另一

种方式开启 InCallScreen()。

16.3.3 Call 通话控制

Android 5.0 通过文件 packages\services\Telephony\src\com\android\phone\CallController.java 实现 Call 通话控制，这是一个自定义消息处理的 Handler 类型，通过函数 placeCall 实现了 Handler 消息处理逻辑。具体实现代码如下所示：

```
public void placeCall(Intent intent) {
    log("placeCall()...  intent = " + intent);
    if (VDBG) log("             extras = " + intent.getExtras());

    // TODO: Do we need to hold a wake lock while this method runs?
    //       Or did we already acquire one somewhere earlier
    //       in this sequence (like when we first received the CALL intent?)

    if (intent == null) {
        Log.wtf(TAG, "placeCall: called with null intent");
        throw new IllegalArgumentException("placeCall: called with null intent");
    }

    String action = intent.getAction();
    Uri uri = intent.getData();
    if (uri == null) {
        Log.wtf(TAG, "placeCall: intent had no data");
        throw new IllegalArgumentException("placeCall: intent had no data");
    }

    String scheme = uri.getScheme();
    String number = PhoneNumberUtils.getNumberFromIntent(intent, mApp);
    if (VDBG) {
        log("- action: " + action);
        log("- uri: " + uri);
        log("- scheme: " + scheme);
        log("- number: " + number);
    }

    // This method should only be used with the various flavors of CALL
    // intents.  (It doesn't make sense for any other action to trigger an
    // outgoing call!)
    if (!(Intent.ACTION_CALL.equals(action)
          || Intent.ACTION_CALL_EMERGENCY.equals(action)
          || Intent.ACTION_CALL_PRIVILEGED.equals(action))) {
        Log.wtf(TAG, "placeCall: unexpected intent action " + action);
        throw new IllegalArgumentException("Unexpected action: " + action);
    }

    // Check to see if this is an OTASP call (the "activation" call
    // used to provision CDMA devices), and if so, do some
    // OTASP-specific setup.
    Phone phone = mApp.mCM.getDefaultPhone();
    if (TelephonyCapabilities.supportsOtasp(phone)) {
        checkForOtaspCall(intent);
    }
    mApp.setRestoreMuteOnInCallResume(false);

    CallStatusCode status = placeCallInternal(intent);

    switch (status) {
        // Call was placed successfully:
        case SUCCESS:
        case EXITED_ECM:
            if (DBG) log("==> placeCall(): success from placeCallInternal(): " + status);
            break;

        default:
            // Any other status code is a failure.
            log("==> placeCall(): failure code from placeCallInternal(): " + status);
            handleOutgoingCallError(status);
```

```
            break;
        }
    }
```

在上述代码中，调用函数 placeCallInternal 处理具体的拨号请求，根据 placeCallInternal 返回的状态可以更新 inCallUiState 的状态。函数 placeCallInternal 的具体实现代码如下所示：

```
private CallStatusCode placeCallInternal(Intent intent) {
    if (DBG) log("placeCallInternal()...  intent = " + intent);

    // TODO: This method is too long.  Break it down into more
    // manageable chunks.

    final Uri uri = intent.getData();
    final String scheme = (uri != null) ? uri.getScheme() : null;
    String number;
    Phone phone = null;

    // Check the current ServiceState to make sure it's OK
    // to even try making a call.
    CallStatusCode okToCallStatus = checkIfOkToInitiateOutgoingCall(
            mCM.getServiceState());

    try {
        number = PhoneUtils.getInitialNumber(intent);
        if (VDBG) log("- actual number to dial: '" + number + "'");
        String sipPhoneUri = intent.getStringExtra(
                OutgoingCallBroadcaster.EXTRA_SIP_PHONE_URI);
        ComponentName thirdPartyCallComponent = (ComponentName) intent.getParcelable Extra(
                OutgoingCallBroadcaster.EXTRA_THIRD_PARTY_CALL_COMPONENT);
        phone = PhoneUtils.pickPhoneBasedOnNumber(mCM, scheme, number, sipPhoneUri,
                thirdPartyCallComponent);
        if (VDBG) log("- got Phone instance: " + phone + ", class = " + phone.getClass());

        // update okToCallStatus based on new phone
        okToCallStatus = checkIfOkToInitiateOutgoingCall(
                phone.getServiceState().getState());

    } catch (PhoneUtils.VoiceMailNumberMissingException ex) {
        if (okToCallStatus != CallStatusCode.SUCCESS) {
            if (DBG) log("Voicemail number not reachable in current SIM card state.");
            return okToCallStatus;
        }
        if (DBG) log("VoiceMailNumberMissingException from getInitialNumber()");
        return CallStatusCode.VOICEMAIL_NUMBER_MISSING;
    }

    if (number == null) {
        Log.w(TAG, "placeCall: couldn't get a phone number from Intent " + intent);
        return CallStatusCode.NO_PHONE_NUMBER_SUPPLIED;
    }
    boolean isEmergencyNumber = PhoneNumberUtils.isLocalEmergencyNumber(mApp, number);
    boolean isPotentialEmergencyNumber =
            PhoneNumberUtils.isPotentialLocalEmergencyNumber(mApp, number);
    boolean isEmergencyIntent = Intent.ACTION_CALL_EMERGENCY.equals(intent.getAction());

    if (isPotentialEmergencyNumber && !isEmergencyIntent) {
        Log.e(TAG, "Non-CALL_EMERGENCY Intent " + intent
                + " attempted to call potential emergency number " + number
                + ".");
        return CallStatusCode.CALL_FAILED;
    } else if (!isPotentialEmergencyNumber && isEmergencyIntent) {
        Log.e(TAG, "Received CALL_EMERGENCY Intent " + intent
                + " with non-potential-emergency number " + number
                + " -- failing call.");
        return CallStatusCode.CALL_FAILED;
    }

    if (isEmergencyNumber
        && ((okToCallStatus == CallStatusCode.EMERGENCY_ONLY)
            || (okToCallStatus == CallStatusCode.OUT_OF_SERVICE))) {
```

```
        if (DBG) log("placeCall: Emergency number detected with status = " + okToCallStatus);
        okToCallStatus = CallStatusCode.SUCCESS;
        if (DBG) log("==> UPDATING status to: " + okToCallStatus);
    }

    if (okToCallStatus != CallStatusCode.SUCCESS) {
        // If this is an emergency call, launch the EmergencyCallHelperService
        // to turn on the radio and retry the call.
        if (isEmergencyNumber && (okToCallStatus == CallStatusCode.POWER_OFF)) {
            Log.i(TAG, "placeCall: Trying to make emergency call while POWER_OFF!");

            // If needed, lazily instantiate an EmergencyCallHelper instance.
            synchronized (this) {
                if (mEmergencyCallHelper == null) {
                    mEmergencyCallHelper = new EmergencyCallHelper(this);
                }
            }

            // ...and kick off the "emergency call from airplane mode" sequence.
            mEmergencyCallHelper.startEmergencyCallFromAirplaneModeSequence(number);
            return CallStatusCode.SUCCESS;
        } else {
            // Otherwise, just return the (non-SUCCESS) status code
            // back to our caller.
            if (DBG) log("==> placeCallInternal(): non-success status: " + okToCallStatus);
            mCallLogger.logCall(null /* callerInfo */, number, 0 /* presentation */,
                    Calls.OUTGOING_TYPE, System.currentTimeMillis(), 0 /* duration */);

            return okToCallStatus;
        }
    }
    Uri contactUri = intent.getData();

    // If a gateway is used, extract the data here and pass that into placeCall.
    final RawGatewayInfo rawGatewayInfo = mCallGatewayManager.getRawGatewayInfo(intent,
    number);

    // Watch out: PhoneUtils.placeCall() returns one of the
    // CALL_STATUS_* constants, not a CallStatusCode enum value.
    int callStatus = PhoneUtils.placeCall(mApp,
                                          phone,
                                          number,
                                          contactUri,
                                          (isEmergencyNumber || isEmergencyIntent),
                                          rawGatewayInfo,
                                          mCallGatewayManager);

    switch (callStatus) {
        case PhoneUtils.CALL_STATUS_DIALED:
            if (VDBG) log("placeCall: PhoneUtils.placeCall() succeeded for regular call '"
                    + number + "'.");


            boolean voicemailUriSpecified = scheme != null &&
                scheme.equals(PhoneAccount.SCHEME_VOICEMAIL);
            boolean exitedEcm = false;
            if (PhoneUtils.isPhoneInEcm(phone) && !isEmergencyNumber) {
                Log.i(TAG, "About to exit ECM because of an outgoing non-emergency call");
                exitedEcm = true;  // this will cause us to return EXITED_ECM from this method
            }

            if (phone.getPhoneType() == PhoneConstants.PHONE_TYPE_CDMA) {
                // Start the timer for 3 Way CallerInfo
                if (mApp.cdmaPhoneCallState.getCurrentCallState()
                        == CdmaPhoneCallState.PhoneCallState.THRWAY_ACTIVE) {
                    mApp.cdmaPhoneCallState.setThreeWayCallOrigState(true);

                    // Schedule the "Dialing" indication to be taken down in 3 seconds:
                    sendEmptyMessageDelayed(THREEWAY_CALLERINFO_DISPLAY_DONE,
                                            THREEWAY_CALLERINFO_DISPLAY_TIME);
                }
```

```
            }

            // Success!
            if (exitedEcm) {
                return CallStatusCode.EXITED_ECM;
            } else {
                return CallStatusCode.SUCCESS;
            }

        case PhoneUtils.CALL_STATUS_DIALED_MMI:
            if (DBG) log("placeCall: specified number was an MMI code: '" + number + "'.");
            return CallStatusCode.DIALED_MMI;

        case PhoneUtils.CALL_STATUS_FAILED:
            Log.w(TAG, "placeCall: PhoneUtils.placeCall() FAILED for number '"
                    + number + "'.");
            // We couldn't successfully place the call; there was some
            // failure in the telephony layer.

            // Log failed call.
            mCallLogger.logCall(null /* callerInfo */, number, 0 /* presentation */,
                    Calls.OUTGOING_TYPE, System.currentTimeMillis(), 0 /* duration */);

            return CallStatusCode.CALL_FAILED;

        default:
            Log.wtf(TAG, "placeCall: unknown callStatus " + callStatus
                    + " from PhoneUtils.placeCall() for number '" + number + "'.");
            return CallStatusCode.SUCCESS;  // Try to continue anyway...
    }
}
```

16.3.4　静态方法调用

在文件 packages\services\Telephony\src\com\android\phone\PhoneUtils.java 中，函数 placeCall 是一个静态方法调用，实现拨号调用界面处理。函数 placeCall 的具体实现代码如下所示：

```
public static int placeCall(Context context, Phone phone, String number, Uri contactRef,
        boolean isEmergencyCall, RawGatewayInfo gatewayInfo, CallGatewayManager
callGateway) {
    final Uri gatewayUri = gatewayInfo.gatewayUri;

    if (VDBG) {
        log("placeCall()... number: '" + number + "'"
                + ", GW:'" + gatewayUri + "'"
                + ", contactRef:" + contactRef
                + ", isEmergencyCall: " + isEmergencyCall);
    } else {
        log("placeCall()... number: " + toLogSafePhoneNumber(number)
                + ", GW: " + (gatewayUri != null ? "non-null" : "null")
                + ", emergency? " + isEmergencyCall);
    }
    final PhoneGlobals app = PhoneGlobals.getInstance();

    boolean useGateway = false;
    if (null != gatewayUri &&
        !isEmergencyCall &&
        PhoneUtils.isRoutableViaGateway(number)) { // Filter out MMI, OTA and other codes.
        useGateway = true;
    }

    int status = CALL_STATUS_DIALED;
    Connection connection;
    String numberToDial;
    if (useGateway) {
        // TODO: 'tel' should be a constant defined in framework base
        // somewhere (it is in webkit.)
        if (null==gatewayUri || !PhoneAccount.SCHEME_TEL.equals(gatewayUri. getScheme())) {
            Log.e(LOG_TAG, "Unsupported URL:" + gatewayUri);
            return CALL_STATUS_FAILED;
```

```
        }
        numberToDial = gatewayUri.getSchemeSpecificPart();
    } else {
        numberToDial = number;
    }

    // Remember if the phone state was in IDLE state before this call.
    // After calling CallManager#dial(), getState() will return different state.
    final boolean initiallyIdle = app.mCM.getState() == PhoneConstants.State.IDLE;

    try {
        connection=app.mCM.dial(phone,numberToDial,VideoProfile.VideoState.AUDIO_ONLY);
    } catch (CallStateException ex) {
        Log.w(LOG_TAG, "Exception from app.mCM.dial()", ex);
        return CALL_STATUS_FAILED;
    }

    int phoneType = phone.getPhoneType();

    // On GSM phones, null is returned for MMI codes
    if (null == connection) {
        status = CALL_STATUS_FAILED;
    } else {
        // Now that the call is successful, we can save the gateway info for the call
        if (callGateway != null) {
            callGateway.setGatewayInfoForConnection(connection, gatewayInfo);
        }

        if (phoneType == PhoneConstants.PHONE_TYPE_CDMA) {
            updateCdmaCallStateOnNewOutgoingCall(app, connection);
        }

        if (gatewayUri == null) {
            String content = context.getContentResolver().SCHEME_CONTENT;
            if ((contactRef != null) && (contactRef.getScheme().equals(content))) {
                Object userDataObject = connection.getUserData();
                if (userDataObject == null) {
                    connection.setUserData(contactRef);
                } else {
                    // TODO: This branch is dead code, we have
                    // just created the connection which has
                    // no user data (null) by default.
                    if (userDataObject instanceof CallerInfo) {
                    ((CallerInfo) userDataObject).contactRefUri = contactRef;
                    } else {
                    ((CallerInfoToken) userDataObject).currentInfo.contactRefUri =
                        contactRef;
                    }
                }
            }
        }

        startGetCallerInfo(context, connection, null, null, gatewayInfo);

        setAudioMode();

        if (DBG) log("about to activate speaker");
        // Check is phone in any dock, and turn on speaker accordingly
        final boolean speakerActivated = activateSpeakerIfDocked(phone);

        final BluetoothManager btManager = app.getBluetoothManager();

        // See also similar logic in answerCall().
        if (initiallyIdle && !speakerActivated && isSpeakerOn(app)
                && !btManager.isBluetoothHeadsetAudioOn()) {
            // This is not an error but might cause users' confusion. Add log just in case.
            Log.i(LOG_TAG, "Forcing speaker off when initiating a new outgoing call...");
            PhoneUtils.turnOnSpeaker(app, false, true);
        }
    }
```

```
    return status;
}

/* package */ static String toLogSafePhoneNumber(String number) {
    // For unknown number, log empty string.
    if (number == null) {
        return "";
    }

    if (VDBG) {
        // When VDBG is true we emit PII.
        return number;
    }

    // Do exactly same thing as Uri#toSafeString() does, which will enable us to compare
    // sanitized phone numbers.
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < number.length(); i++) {
        char c = number.charAt(i);
        if (c == '-' || c == '@' || c == '.') {
            builder.append(c);
        } else {
        builder.append('x');
        }
    }
    return builder.toString();
}
```

16.3.5　通话管理

文件 frameworks\opt\telephony\src\java\com\android\internal\telephony\CallManager.java 是整个通话管理的大管家，这个类是 Frameworks 层在 Call 业务中面对 App 层的最后一层封装。通过类 CallManager 可以对底层的业务做进一步归纳，实际上也是为 PhoneApp 提供了 Call 业务的控制接口。其核心代码如下所示：

```
public Phone getPhoneInCall() {
    Phone phone = null;
    if (!getFirstActiveRingingCall().isIdle()) {
        phone = getFirstActiveRingingCall().getPhone();
    } else if (!getActiveFgCall().isIdle()) {
        phone = getActiveFgCall().getPhone();
    } else {
        // If BG call is idle, we return default phone
        phone = getFirstActiveBgCall().getPhone();
    }
    return phone;
}

public Phone getPhoneInCall(long subId) {
    Phone phone = null;
    if (!getFirstActiveRingingCall(subId).isIdle()) {
        phone = getFirstActiveRingingCall(subId).getPhone();
    } else if (!getActiveFgCall(subId).isIdle()) {
        phone = getActiveFgCall(subId).getPhone();
    } else {
        // If BG call is idle, we return default phone
        phone = getFirstActiveBgCall(subId).getPhone();
    }
    return phone;
}
public Phone getDefaultPhone() {
    return mDefaultPhone;
}
public Phone getFgPhone() {
    return getActiveFgCall().getPhone();
}
public Phone getFgPhone(long subId) {
    return getActiveFgCall(subId).getPhone();
}
```

```
public Phone getBgPhone() {
    return getFirstActiveBgCall().getPhone();
}
public Phone getBgPhone(long subId) {
    return getFirstActiveBgCall(subId).getPhone();
}
public Phone getRingingPhone() {
    return getFirstActiveRingingCall().getPhone();
}
public Phone getRingingPhone(long subId) {
    return getFirstActiveRingingCall(subId).getPhone();
}
private boolean hasMoreThanOneRingingCall() {
    int count = 0;
    for (Call call : mRingingCalls) {
        if (call.getState().isRinging()) {
            if (++count > 1) return true;
        }
    }
    return false;
}
private boolean hasMoreThanOneRingingCall(long subId) {
    int count = 0;
    for (Call call : mRingingCalls) {
        if ((call.getState().isRinging()) &&
            ((call.getPhone().getSubId() == subId) ||
            (call.getPhone() instanceof SipPhone))) {
            if (++count > 1) return true;
        }
    }
    return false;
}
private boolean hasMoreThanOneHoldingCall(long subId) {
    int count = 0;
    for (Call call : mBackgroundCalls) {
        if ((call.getState() == Call.State.HOLDING) &&
            ((call.getPhone().getSubId() == subId) ||
            (call.getPhone() instanceof SipPhone))) {
            if (++count > 1) return true;
        }
    }
    return false;
}
public List<Connection> getFgCallConnections() {
    Call fgCall = getActiveFgCall();
    if ( fgCall != null) {
        return fgCall.getConnections();
    }
    return mEmptyConnections;
}
public List<Connection> getFgCallConnections(long subId) {
    Call fgCall = getActiveFgCall(subId);
    if ( fgCall != null) {
        return fgCall.getConnections();
    }
    return mEmptyConnections;
}
public List<Connection> getBgCallConnections() {
    Call bgCall = getFirstActiveBgCall();
    if ( bgCall != null) {
        return bgCall.getConnections();
    }
    return mEmptyConnections;
}
public List<Connection> getBgCallConnections(long subId) {
    Call bgCall = getFirstActiveBgCall(subId);
    if ( bgCall != null) {
        return bgCall.getConnections();
    }
    return mEmptyConnections;
}
```

```
public Connection getFgCallLatestConnection() {
    Call fgCall = getActiveFgCall();
    if ( fgCall != null) {
        return fgCall.getLatestConnection();
    }
    return null;
}
public Connection getFgCallLatestConnection(long subId) {
    Call fgCall = getActiveFgCall(subId);
    if ( fgCall != null) {
        return fgCall.getLatestConnection();
    }
    return null;
}
```

文件 CallManager.java 的功能是，进一步抽象封装后的 Call 实例，以便说明当前维护的通话状态。由此可见，上层所关注的是以下 3 种通话状态。

- 前台通话：正在拨出以及已经接通了的电话，注意是作为主叫方。
- 后台电话：如果前台正在通话中，则后面打进来的电话被作为后台电话来处理。
- 响铃电话：正在响铃而尚未被接通的电话，包括存在前台通话时的情况，注意是作为被叫方。

针对上述 3 种通话状态，文件 CallManager.java 派生出与之相关的 Call、Connection 以及 Phone 的状态函数。具体来说，在文件 CallManager.java 中派生出如下 3 类函数来处理上述 3 种通话状态。

（1）getXXCall& hasXXCall

这部分的方法比较多，剔除私有方法可以看到，在公共方法中与 ForegroundCall、BackgroundCall 和 RingingCall 3 种状态相关的大致可以分为两类：

- 第一类是判断当前 Call 是否为对应状态下的通话（has）；
- 第二类是获取处于对应状态下通话实例（get）。

对于后者来说，getForegroundCall()与 getActiveFgCall()有什么区别呢？从代码中可以看出，getForegroundCall()通过 unmodifiableList 方法返回一个前台通话列表，可以用来轮询当前所有的前台通话，而 getActiveFgCall()则返回一个具体的前台通话实例。此外，需要注意的是，由于后台通话可能有多个（比如说多方通话尚未合并时），此时调用 getFirstActiveBgCall()获取第一个后台电话，同理，对于 getFirstActiveRingingCall()来说也是如此。

（2）getXXPhone

再看对应于 ForegroundCall、BackgroundCall 和 RingingCall 3 种状态下的 getFgPhone, getBgPhone 和 getRingingPhone，都是取自于 Call.getPhone()方法获取对应对象的。归根结底，它们仍然获取的是对应 CallTracker 维护的 Phone 实例。其中私有方法 getPhoneBase()用来消掉代理，指向真正所用到的 Phone 对象，在需要真正 Phone 对象作为参数的场合会用到。

（3）getXXConnection

对应的方法会返回一个前台/后台状态下的通话链路列表，通过轮询该列表可以获取到当前处于该状态下的所有 Connections。其中 getFgCallLatestConnection()方法用来返回最新的一次前台通话 Connection，用到的场合一般是需要最新一次前台通话链路的地方，比如说“重拨电话”的时候。

16.3.6　dial 拨号

在文件 frameworks\opt\telephony\src\java\com\android\internal\telephony\gsm\GSMPhone.java 中，通过函数 dial 实现 dial 拨号处理功能。具体实现代码如下所示：

```
public Connection
dial (String dialString, UUSInfo uusInfo, int videoState) throws CallStateException {
    ImsPhone imsPhone = mImsPhone;

    boolean imsUseEnabled =
            ImsManager.isEnhanced4gLteModeSettingEnabledByPlatform(mContext) &&
            ImsManager.isEnhanced4gLteModeSettingEnabledByUser(mContext);
    if (!imsUseEnabled) {
        Rlog.w(LOG_TAG, "IMS is disabled: forced to CS");
    }
```

```
    if (imsUseEnabled && imsPhone != null
            && ((imsPhone.getServiceState().getState() == ServiceState.STATE_IN_SERVICE
            && !PhoneNumberUtils.isEmergencyNumber(dialString))
            || (PhoneNumberUtils.isEmergencyNumber(dialString)
            && mContext.getResources().getBoolean(
                    com.android.internal.R.bool.useImsAlwaysForEmergencyCall))) ) {
        try {
            if (LOCAL_DEBUG) Rlog.d(LOG_TAG, "Trying IMS PS call");
            return imsPhone.dial(dialString, videoState);
        } catch (CallStateException e) {
            if (LOCAL_DEBUG) Rlog.d(LOG_TAG, "IMS PS call exception " + e);
            if (!ImsPhone.CS_FALLBACK.equals(e.getMessage())) {
                CallStateException ce = new CallStateException(e.getMessage());
                ce.setStackTrace(e.getStackTrace());
                throw ce;
            }
        }
    }

    if (LOCAL_DEBUG) Rlog.d(LOG_TAG, "Trying (non-IMS) CS call");
    return dialInternal(dialString, null, VideoProfile.VideoState.AUDIO_ONLY);
}
```

在上述代码中，mCT 表示类 GsmCallTracker。

16.3.7 状态跟踪

在 Android 5.0 源代码中，通过文件 frameworks\opt\telephony\src\java\com\android\internal\telephony\gsm\GsmCallTracker.java 实现了对 Call 状态的跟踪处理。GSMCallTracker 在本质上是一个 Handler，是 Android 的通话管理层。GSMCallTracker 建立了 ConnectionList 来管理现行的通话连接，并向上层提供电话调用接口。在文件 GsmCallTracker.java 中，通过函数 dial 实现拨号状态跟踪，具体实现代码如下所示：

```
synchronized Connection
dial (String dialString, int clirMode, UUSInfo uusInfo) throws CallStateException {
    // note that this triggers call state changed notif
    clearDisconnected();

    if (!canDial()) {
        throw new CallStateException("cannot dial in current state");
    }

    String origNumber = dialString;
    dialString = convertNumberIfNecessary(mPhone, dialString);
    if (mForegroundCall.getState() == GsmCall.State.ACTIVE) {
        switchWaitingOrHoldingAndActive();
        fakeHoldForegroundBeforeDial();
    }

    if (mForegroundCall.getState() != GsmCall.State.IDLE) {
        //we should have failed in !canDial() above before we get here
        throw new CallStateException("cannot dial in current state");
    }

    mPendingMO = new GsmConnection(mPhone.getContext(), checkForTestEmergencyNumber
(dialString), this, mForegroundCall);
    mHangupPendingMO = false;

    if ( mPendingMO.getAddress() == null || mPendingMO.getAddress().length() == 0
            || mPendingMO.getAddress().indexOf(PhoneNumberUtils.WILD) >= 0
    ) {
        // Phone number is invalid
        mPendingMO.mCause = DisconnectCause.INVALID_NUMBER;

        // handlePollCalls() will notice this call not present
        // and will mark it as dropped.
        pollCallsWhenSafe();
```

```
        } else {
            // Always unmute when initiating a new call
            setMute(false);

            mCi.dial(mPendingMO.getAddress(), clirMode, uusInfo, obtainCompleteMessage());
        }

        if (mNumberConverted) {
            mPendingMO.setConverted(origNumber);
            mNumberConverted = false;
        }

        updatePhoneState();
        mPhone.notifyPreciseCallStateChanged();

        return mPendingMO;
    }
```

另外，文件 GSMCallTracker.java 维护了通话列表 connections，顺序记录了连接上的通话，这些通话包括 ACTIVE、DIALING、ALERTING、HOLDING、INCOMING 和 WAITING 等状态的连接。GSMCallTracker 将这些连接分为如下 3 类进行管理。

- RingingCall：INCOMING、WAITING。
- ForegourndCall：ACTIVE、DIALING、ALERTING。
- BackgroundCall：HOLDING。

这样上层函数通过 getRingCall()和 getForegrouandCall()等来获得电话系统中特定通话连接。为了管理电话状态，GSMCallTracker 在构造时就将自己登记到了电话状态变化通知表中。RIL-Java 一收到电话状态变化的通知，就会使用 EVENT_CALL_STATE_CHANGE 通知到 GSMCallTacker。

在一般的实现中，我们的通话 Call Table 是通过 AT+CLCC 查询到的，CPI 可以通知到电话的改变，但是，CPI 在各个 Modem 的实现中差别比较大，所以，参考设计都没有用到 CPI 这样的电话连接改变通知，而是使用最为传统的 CLCC 查询 CALL TABLE。在 GSMTracker 中使用 connections 来管理 Android 电话系统中的通话连接。每次电话状态发生变化时，GSMTracker 就会使用 CLCC 查询来更新 connections 内容，如果内容有发生变化，则向上层发起电话状态改变的通知。

在 RIL-JAVA 中，会涉及如下所示的 CurrentCallList 查询。

- hangup。
- dial。
- acceptCall。
- rejectCall。

当文件 GSMcallTracker.java 在发起上述调用时，都会有一个共同的 ResultMessage 构造函数：obtainCompleteMessage()，这个函数实际上调用的是：obtainCompleteMessage(EVENT_OPERATION_COMPLETE)，这就意味着在这些电话操作后，GSMCallTracker 会收到 EVENT_OPERATION_COMPLETE 消息，于是将目光转移到 handleMessage()@GSMCallTracker 的 EVENT_OPERATION_COMPLETE 事件处理：operationComplete@GSMCallTracker。函数 operationComplete 会使用 cm.getCurrentCalls(lastRelevantPoll)调用，向 RILD 发起 RIL_REQUEST_GET_CURRENT_CALLS 调用，这个最终就是向 Modem 发起 AT+CLCC，获取到真正的电话列表。

16.3.8　RIL 消息“出/入”口

到现在为止，拨号过程已经来到 RIL 层。追踪 RIL_REQUEST_DIAL 这个 TAG 标志向下走，来到 Reference-ril.c 文件，找到如下相应的 case 分支代码：

```
case RIL_REQUEST_DIAL:
        requestDial(data, datalen, t);

static void requestDial(void *data, size_t datalen, RIL_Token t){
 p_dial = (RIL_Dial *)data;

 switch (p_dial->clir) {
```

```
        case 1: clir = "I"; break;  /*invocation*/
        case 2: clir = "i"; break;  /*suppression*/
        default:
        case 0: clir = ""; break;   /*subscription default*/
    }
    asprintf(&cmd, "ATD%s%s;", p_dial->address, clir);
    ret = at_send_command(cmd, NULL);
    free(cmd);
    RIL_onRequestComplete(t, RIL_E_SUCCESS, NULL, 0);
}
```

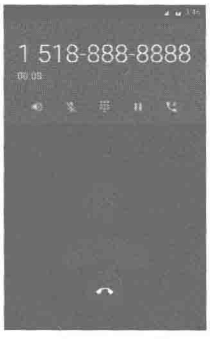
到此为止，电话基本上算是拨出去了，至于是否成功还要看返回结果，通常形式上的流程如下所示：

```
===>>[SendAT] ATD15188888888 //拨号
<<====[RecvAT] OK
<<====[RecvAT] +CLCC: 1, 0, 2, 0, 0, "15188888888", 129, //主动上报 clcc
===>>[SendAT] AT+CMUT=0, time=   //设置话筒静音关
<<====[RecvAT] OK, time=
===>>[SendAT] AT+CLCC          //主动查询，
<<====[RecvAT] +CLCC: 1, 0, 2, 0, 0, "15188888888", 129, //以这一次的 clcc 为准
```

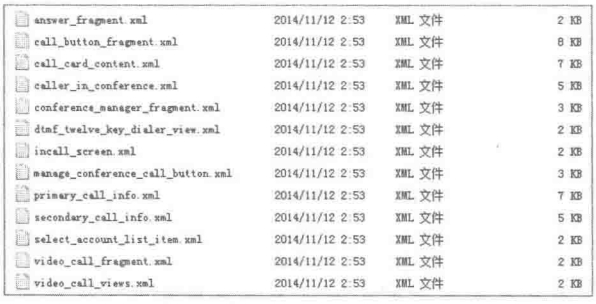
16.3.9 显示通话主界面

拨号完成后会显示通话主界面，此界面通过文件 packages\apps\phone\InCallScreen.java 实现。在电话的呼出流程中，我们最后须要按下拨号键，才能将电话拨打出去，那么在按下拨号键之后，我们可以看到会弹出一个界面，表示拨号信息以及一些其他信息，这个界面就是 InCall S creen 界面。当然，在来电时弹出的界面依然是 InCallScreen，在接通电话之后表现的那个界面仍然是 InCallScreen。也就是说在通话进程中，我们始终可见并操作的那个界面就是 InCallScreen。Android 5.0 的 InCallScreen 界面效果如图 16-6 所示。读者经过比较可以发现，InCallScreen 的拨号和接通时的界面效果基本一致，而来电界面主要由于多了一个滑动接听的控件，从而导致界面不太一样。

实现拨号界面的布局文件目录是：packages\apps\InCallUI\res\layout\，如图 16-7 所示。



▲图 16-6 Android 5.0 的拨号界面



answer_fragment.xml	2014/11/12 2:53	XML 文件	2 KB
call_button_fragment.xml	2014/11/12 2:53	XML 文件	8 KB
call_card_content.xml	2014/11/12 2:53	XML 文件	7 KB
caller_in_conference.xml	2014/11/12 2:53	XML 文件	5 KB
conference_manager_fragment.xml	2014/11/12 2:53	XML 文件	3 KB
dtmf_twelve_key_dialer_view.xml	2014/11/12 2:53	XML 文件	2 KB
incall_screen.xml	2014/11/12 2:53	XML 文件	2 KB
manage_conference_call_button.xml	2014/11/12 2:53	XML 文件	3 KB
primary_call_info.xml	2014/11/12 2:53	XML 文件	7 KB
secondary_call_info.xml	2014/11/12 2:53	XML 文件	5 KB
select_account_list_item.xml	2014/11/12 2:53	XML 文件	2 KB
video_call_fragment.xml	2014/11/12 2:53	XML 文件	2 KB
video_call_views.xml	2014/11/12 2:53	XML 文件	2 KB

▲图 16-7 拨号界面的布局文件目录

在 InCallScreen 的 UI 界面中，主要包含对以下 6 个方面的控制。

- incomingCallWidget：接通/挂断/短信回复时须要应用。
- dialpadButton：显示或隐藏拨号盘（DTMF）。
- audioButton：开启/关闭扬声器。
- muteButton：开启/关闭麦克风静音，开启之后对方没法听到你的声音。
- holdButton：开启/关闭呼叫保持。
- addButton：增长多路通话，即在通话的进程中可以暂停，用来以后通话；拨打另一路通话并接通。

注意 被动接电话和主动拨号处理类似，为节省本书篇幅，在书中不再进行详细讲解。

第 17 章　分析短信系统

对于一款 Android 手机设备来说，除了拨打电话之外，还有一种比较重要的数据通信方式：短信。在实现短信收发功能的过程中，Android 需要确保这些信息的安全性。在本章的内容中，将详细讲解 Android 5.0 源代码中短信系统的基本知识。

17.1 短信系统的主界面

在 Android 系统中，应用程序通过文件 SmsManager.java 实现发送短信功能。在发送短信时，首先来到发送短信界面，默认主界面是通过文件“packages\apps\Mms\src\com\android\mms\uiConversationList.java”实现，在此文件中首先会监听 onListItemClick 事件，具体代码如下所示。

```
protected void onListItemClick(ListView l, View v, int position, long id) {
    Cursor cursor  = (Cursor) getListView().getItemAtPosition(position);
    Conversation conv = Conversation.from(this, cursor);
    long tid = conv.getThreadId();

    if (LogTag.VERBOSE) {
        Log.d(TAG, "onListItemClick: pos=" + position + ", view=" + v + ", tid=" + tid);
    }

    openThread(tid);
}
```

在上述代码中，如果 position 为 0，则调用函数 createNewMessage 新创建一个短信，具体代码如下所示。

```
private void createNewMessage() {
    startActivity(ComposeMessageActivity.createIntent(this, 0));
}
```

在文件中，根据用户在主界面的选项可以调用对应的函数实现对应的功能，例如新创建短信选项、删除回话选项等。这个功能是通过 onOptionsItemSelected 实现的，具体代码如下所示。

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.action_compose_new:
            createNewMessage();
            break;
        case R.id.action_delete_all:
            // The invalid threadId of -1 means all threads here.
            confirmDeleteThread(-1L, mQueryHandler);
            break;
        case R.id.action_settings:
            Intent intent = new Intent(this, MessagingPreferenceActivity.class);
            startActivityIfNeeded(intent, -1);
            break;
        case R.id.action_debug_dump:
            LogTag.dumpInternalTables(this);
            break;
        case R.id.action_cell_broadcasts:
            Intent cellBroadcastIntent = new Intent(Intent.ACTION_MAIN);
            cellBroadcastIntent.setComponent(new ComponentName(
                    "com.android.cellbroadcastreceiver",
                    "com.android.cellbroadcastreceiver.CellBroadcastListActivity"));
            cellBroadcastIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

```
            try {
                startActivity(cellBroadcastIntent);
            } catch (ActivityNotFoundException ignored) {
                Log.e(TAG, "ActivityNotFoundException for CellBroadcastListActivity");
            }
            return true;
        default:
            return true;
    }
    return false;
}
```

函数 createNewMessage 的功能是设置程序来到一个新的界面 startActivity，即来到文件“packages\apps\Mms\src\com\android\mms\ComposeMessageActivity.java”，在此文件中首先会监听用户是否单击“发送”按钮，具体实现代码如下所示。

```
public void onClick(View v) {
    if ((v == mSendButtonSms || v == mSendButtonMms) && isPreparedForSending()) {
        confirmSendMessageIfNeeded();
    } else if ((v == mRecipientsPicker)) {
        launchMultiplePhonePicker();
    }
}
```

如果单击了“发送”按钮，则调用函数 confirmSendMessageIfNeeded 进行再次判断，以确定是否发送信息。具体实现代码如下所示。

```
private void confirmSendMessageIfNeeded() {
    if (!isRecipientsEditorVisible()) {
        sendMessage(true);
        return;
    }

    boolean isMms = mWorkingMessage.requiresMms();
    if (mRecipientsEditor.hasInvalidRecipient(isMms)) {
        if (mRecipientsEditor.hasValidRecipient(isMms)) {
            String title = getResourcesString(R.string.has_invalid_recipient,
                    mRecipientsEditor.formatInvalidNumbers(isMms));
            new AlertDialog.Builder(this)
                .setTitle(title)
                .setMessage(R.string.invalid_recipient_message)
                .setPositiveButton(R.string.try_to_send,
                        new SendIgnoreInvalidRecipientListener())
                .setNegativeButton(R.string.no, new CancelSendingListener())
                .show();
        } else {
            new AlertDialog.Builder(this)
                .setTitle(R.string.cannot_send_message)
                .setMessage(R.string.cannot_send_message_reason)
                .setPositiveButton(R.string.yes, new CancelSendingListener())
                .show();
        }
    } else {
        // The recipients editor is still open. Make sure we use what's showing there
        // as the destination.
        ContactList contacts = mRecipientsEditor.constructContactsFromInput(false);
        mDebugRecipients = contacts.serialize();
        sendMessage(true);
    }
}
```

经过上述再次确认后，如果确认发送，则调用函数 sendMessage 发送当前新创建的短信，具体实现代码如下所示。

```
private void sendMessage(boolean bCheckEcmMode) {
    if (bCheckEcmMode) {
        // TODO: expose this in telephony layer for SDK build
        String inEcm = SystemProperties.get(TelephonyProperties.PROPERTY_INECM_MODE);
        if (Boolean.parseBoolean(inEcm)) {
            try {
```

```
                startActivityForResult(
                        new Intent(TelephonyIntents.ACTION_SHOW_NOTICE_ECM_BLOCK_OTHERS,
                        null), REQUEST_CODE_ECM_EXIT_DIALOG);
                return;
            } catch (ActivityNotFoundException e) {
                // continue to send message
                Log.e(TAG, "Cannot find EmergencyCallbackModeExitDialog", e);
            }
        }
    }

    if (!mSendingMessage) {
        if (LogTag.SEVERE_WARNING) {
            String sendingRecipients = mConversation.getRecipients().serialize();
            if (!sendingRecipients.equals(mDebugRecipients)) {
                String workingRecipients = mWorkingMessage.getWorkingRecipients();
                if (!mDebugRecipients.equals(workingRecipients)) {
                    LogTag.warnPossibleRecipientMismatch("ComposeMessageActivity.
                    sendMessage" + " recipients in window: \"" +
                            mDebugRecipients + "\" differ from recipients from conv: \"" +
                            sendingRecipients + "\" and working recipients: " +
                            workingRecipients, this);
                }
            }
            sanityCheckConversation();
        }

        // send can change the recipients. Make sure we remove the listeners first and then add
        // them back once the recipient list has settled.
        removeRecipientsListeners();

        mWorkingMessage.send(mDebugRecipients);// mDebugRecipients 是指同步得到的全部收件
人，以分号间隔

        mSentMessage = true;
        mSendingMessage = true;
        addRecipientsListeners();//重新添加对收件人的监听

        mScrollOnSend = true;   // in the next onQueryComplete, scroll the list to the end.
    }
    // But bail out if we are supposed to exit after the message is sent.
    if (mSendDiscreetMode) {
        finish();// 信息发送完成后，退出 activity
    }
}
```

在上述代码中，如果当前默认的 SIM 卡有效，则用当前的 SIM 卡发送，否则进入选择 SIM 卡对话框。如果 SIM 卡有效，则首先设置当前的 SIM 卡，然后通过函数 removeRecipientsListeners 取消对收件人的监听。

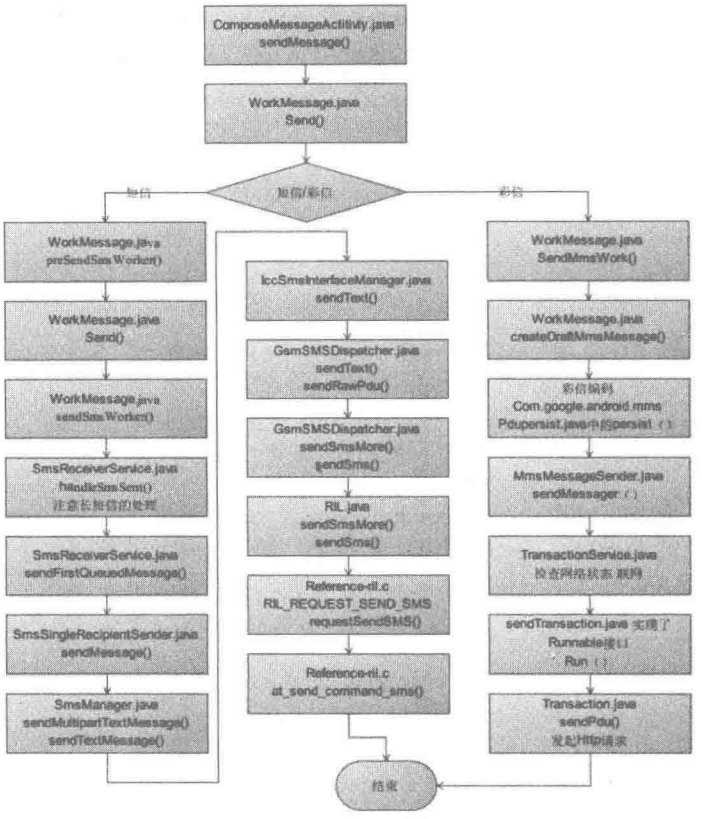
17.2 发送普通短信

这里的普通短信是相对于彩信来说的，在 Android 系统中，发送短信的流程如图 17-1 所示。

接着本章前面介绍的流程。在调用函数 sendMessage 发送当前新创建的短信之前，需要对当前短信的类型进行判断，如果是彩信，则调用 sendMmsWork 进行处理；如果是普通短信，则调用 preSendSmsWorker 进行处理。在接下来的内容中，将首先讲解在 Android 系统中发送普通短信的基本流程。

首先在文件“packages\apps\Mms\src\com\android\mms\data\WorkingMessage.java”中，通过函数 preSendSmsWorker 来发送信息，具体实现代码如下所示。

```
    private void preSendSmsWorker(Conversation conv,String msgText,String recipientsInUI) {
        // 发送一个广播，说明当前的内容是用户想要的内容
        UserHappinessSignals.userAcceptedImeText(mActivity);
//ComposeMessageActivity. onPreMessageSent-> resetMessage 用于重置一些信息，比如清空输入内
//容框、一些监听等
```



▲图 17-1 发送短信的流程

```
    mStatusListener.onPreMessageSent();

    long origThreadId = conv.getThreadId();

    // 如果当前会话 ID<=0，新创建会话 ID
    long threadId = conv.ensureThreadId();

    String semiSepRecipients = conv.getRecipients().serialize();

    // recipientsInUI can be empty when the user types in a number and hits send
    if (LogTag.SEVERE_WARNING && ((origThreadId != 0 && origThreadId != threadId) ||
            (!semiSepRecipients.equals(recipientsInUI) && !TextUtils.isEmpty
            (recipientsInUI)))) {
        String msg = origThreadId != 0 && origThreadId != threadId ?
                "WorkingMessage.preSendSmsWorker threadId changed or " +
                "recipients changed. origThreadId: " +
                origThreadId + " new threadId: " + threadId +
                " also mConversation.getThreadId(): " +
                mConversation.getThreadId()
            :
                "Recipients in window: \"" +
                recipientsInUI + "\" differ from recipients from conv: \"" +
                semiSepRecipients + "\"";

        LogTag.warnPossibleRecipientMismatch(msg, mActivity);
    }

    // just do a regular send. We're already on a non-ui thread so no need to fire
    //发送信息
    sendSmsWorker(msgText, semiSepRecipients, threadId);

    //可能此对话被存在了草稿中，所以在发送后需要删除
    deleteDraftSmsMessage(threadId);
}
```

在上述代码中，调用函数 sendSmsWorker 来发送信息，具体实现代码如下所示。

```
private void sendSmsWorker(String msgText, String semiSepRecipients, long threadId) {
    String[] dests = TextUtils.split(semiSepRecipients, ";");//通过分号，分开收件人
    if (LogTag.VERBOSE || Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.d(LogTag.TRANSACTION, "sendSmsWorker sending message: recipients=" +
                semiSepRecipients + ", threadId=" + threadId);
    }
    MessageSender sender = new SmsMessageSender(mActivity, dests, msgText, threadId);
    try {
        sender.sendMessage(threadId); //根据 ThreadID 发送短信

        // //删除旧的消息，检查限制
        Recycler.getSmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
    } catch (Exception e) {
        Log.e(TAG, "Failed to send SMS message, threadId=" + threadId, e);
    }

    mStatusListener.onMessageSent();//回调接口，在发送信息时调用此函数
    MmsWidgetProvider.notifyDatasetChanged(mActivity);
}
```

在上述代码中，调用了文件“packages\apps\Mms\src\com\android\mms\transaction\SmsMessage Sender.java”中的函数 sendMessage 发送短信并实现广播。具体实现代码如下所示。

```
public boolean sendMessage(long token) throws MmsException {
    return queueMessage(token);
}
```

在上述代码中，是通过调用函数 queueMessage 实现真正的发送功能的，具体实现代码如下所示。

```
    private boolean queueMessage(long token) throws MmsException {
        if ((mMessageText == null) || (mNumberOfDests == 0)) {
            // Don't try to send an empty message.
            throw new MmsException("Null message body or dest.");
        }
// 得到发送报告设置状态
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(mContext);
        boolean requestDeliveryReport = prefs.getBoolean(
                MessagingPreferenceActivity.SMS_DELIVERY_REPORT_MODE,
                DEFAULT_DELIVERY_REPORT_MODE);
//queueMessage 把消息按照收件人拆开成多条消息，并且都加入发送队列
        for (int i = 0; i < mNumberOfDests; i++) {
            try {
                if (LogTag.DEBUG_SEND) {
                    Log.v(TAG, "queueMessage mDests[i]: " + mDests[i] + " mThreadId: " +
                    mThreadId);
                }
                Sms.addMessageToUri(mContext.getContentResolver(),
                        Uri.parse("content://sms/queued"), mDests[i],
                        mMessageText, null, mTimestamp,
                        true /* read */,
                        requestDeliveryReport,
                        mThreadId);
            } catch (SQLiteException e) {
                if (LogTag.DEBUG_SEND) {
                    Log.e(TAG, "queueMessage SQLiteException", e);
                }
                SqliteWrapper.checkSQLiteException(mContext, e);
            }
        }
         // 发送广播给 SmsReceiver
        mContext.sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
                null,
                mContext,
                SmsReceiver.class));
        return false;
    }
```

在上述代码中，如下代码行的功能是发送广播给 SmsReceiver。

```
mContext.sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
```

```
        null,
        mContext,
        SmsReceiver.class));
```

上述广播功能是通过文件“packages\apps\Mms\src\com\android\mms\transaction\SmsReceiver.java”实现的，主要实现代码如下所示。

```
public class SmsReceiver extends BroadcastReceiver {
    static final Object mStartingServiceSync = new Object();
    static PowerManager.WakeLock mStartingService;
    private static SmsReceiver sInstance;
    public static SmsReceiver getInstance() {
        if (sInstance == null) {
            sInstance = new SmsReceiver();
        }
        return sInstance;
    }
    @Override
    public void onReceive(Context context, Intent intent) {
        onReceiveWithPrivilege(context, intent, false);
    }
    protected void onReceiveWithPrivilege(Context context, Intent intent, boolean
privileged) {
        if (!privileged && intent.getAction().equals(Intents.SMS_RECEIVED_ACTION)) {
            return;
        }
        intent.setClass(context, SmsReceiverService.class);
        intent.putExtra("result", getResultCode());
        beginStartingService(context, intent);
    }
    public static void beginStartingService(Context context, Intent intent) {
        synchronized (mStartingServiceSync) {
            if (mStartingService == null) {
                PowerManager pm =
                    (PowerManager)context.getSystemService(Context.POWER_SERVICE);
                mStartingService = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                        "StartingAlertService");
                mStartingService.setReferenceCounted(false);
            }
            mStartingService.acquire();
            context.startService(intent);
        }
    }
    public static void finishStartingService(Service service, int startId) {
        synchronized (mStartingServiceSync) {
            if (mStartingService != null) {
                if (service.stopSelfResult(startId)) {
                    mStartingService.release();
                }
            }
        }
    }
}
```

在上述代码中，当类 SmsReceiver 的 onReceive 收到此广播后会启动服务 SmsReceiverService，此服务在文件“packages\apps\Mms\src\com\android\mms\transaction\SmsReceiverService.java”中实现，此文件是 SMS 处理的 Service，负责短信的发送和接收功能。具体实现代码如下所示。

```
    public void handleMessage(Message msg) {
        int serviceId = msg.arg1;
        Intent intent = (Intent)msg.obj;
        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            Log.v(TAG, "handleMessage serviceId: " + serviceId + " intent: " + intent);
        }
        if (intent != null) {
            String action = intent.getAction();

            int error = intent.getIntExtra("errorCode", 0);

            if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
```

```
            Log.v(TAG, "handleMessage action: " + action + " error: " + error);
        }

        if (MESSAGE_SENT_ACTION.equals(intent.getAction())) {
            handleSmsSent(intent, error);
        } else if (SMS_RECEIVED_ACTION.equals(action)) {
            handleSmsReceived(intent, error);
        } else if (ACTION_BOOT_COMPLETED.equals(action)) {
            handleBootCompleted();
        } else if (TelephonyIntents.ACTION_SERVICE_STATE_CHANGED.equals(action)) {
            handleServiceStateChanged(intent);
        } else if (ACTION_SEND_MESSAGE.endsWith(action)) {
            handleSendMessage();
        } else if (ACTION_SEND_INACTIVE_MESSAGE.equals(action)) {
            handleSendInactiveMessage();
        }
    }
    // NOTE: We MUST not call stopSelf() directly, since we need to
    // make sure the wake lock acquired by AlertReceiver is released.
    SmsReceiver.finishStartingService(SmsReceiverService.this, serviceId);
  }
}
```

在上述代码中，当得到发送短信的指令 ACTION_SEND_MESSAGE 后执行函数 handleSend Message，具体实现代码如下所示。

```
private void handleSendMessage() {
    if (!mSending) {
        sendFirstQueuedMessage();
    }
}
```

在上述代码中，调用函数 sendFirstQueuedMessage 查询队列中的信息，包括上次没有发送出去存放在发送队列的信息，取出队列中的第一条消息后进行发送。函数 sendFirstQueuedMessage 的具体实现代码如下所示。

```
public synchronized void sendFirstQueuedMessage() {
    boolean success = true;
    // get all the queued messages from the database
    final Uri uri = Uri.parse("content://sms/queued");
    ContentResolver resolver = getContentResolver();
    Cursor c = SqliteWrapper.query(this, resolver, uri,
                SEND_PROJECTION, null, null, "date ASC");// date ASC so we send out in
                                                         // same order the user tried
                                                         // to send messages.
    if (c != null) {
        try {
            if (c.moveToFirst()) {
                String msgText = c.getString(SEND_COLUMN_BODY);
                String address = c.getString(SEND_COLUMN_ADDRESS);
                int threadId = c.getInt(SEND_COLUMN_THREAD_ID);
                int status = c.getInt(SEND_COLUMN_STATUS);

                int msgId = c.getInt(SEND_COLUMN_ID);
                Uri msgUri = ContentUris.withAppendedId(Sms.CONTENT_URI, msgId);

                SmsMessageSender sender = new SmsSingleRecipientSender(this,
                        address, msgText, threadId, status == Sms.STATUS_PENDING,
                        msgUri);

                if (LogTag.DEBUG_SEND ||
                        LogTag.VERBOSE ||
                        Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
                    Log.v(TAG, "sendFirstQueuedMessage " + msgUri +
                            ", address: " + address +
                            ", threadId: " + threadId);
                }

                try {
                    sender.sendMessage(SendingProgressTokenManager.NO_TOKEN);;
```

```
                mSending = true;
            } catch (MmsException e) {
                Log.e(TAG, "sendFirstQueuedMessage: failed to send message " + msgUri
                        + ", caught ", e);
                mSending = false;
                messageFailedToSend(msgUri, SmsManager.RESULT_ERROR_GENERIC_FAILURE);
                success = false;
                // Sending current message fails. Try to send more pending messages
                // if there is any.
                sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
                        null,
                        this,
                        SmsReceiver.class));
            }
        }
    } finally {
        c.close();
    }
}
if (success) {
    // We successfully sent all the messages in the queue. We don't need to
    // be notified of any service changes any longer.
    unRegisterForServiceStateChanges();
}
}
```

在上述代码中，调用了文件“packages\apps\Mms\src\com\android\mms\transaction\SmsSingleRecipientSender.java”中的函数 sendMessage，功能是实现单个联系人的消息发送功能。具体实现代码如下所示。

```
public boolean sendMessage(long token) throws MmsException {
    if (LogTag.DEBUG_SEND) {
        Log.v(TAG, "sendMessage token: " + token);
    }
    if (mMessageText == null) {
        // Don't try to send an empty message, and destination should be just
        // one.
        throw new MmsException("Null message body or have multiple destinations.");
    }
    SmsManager smsManager = SmsManager.getDefault();
    ArrayList<String> messages = null;
    if ((MmsConfig.getEmailGateway() != null) &&
            (Mms.isEmailAddress(mDest) || MessageUtils.isAlias(mDest))) {
        String msgText;
        msgText = mDest + " " + mMessageText;
        mDest = MmsConfig.getEmailGateway();
        messages=smsManager.divideMessage(msgText);//拆分长短信，每条短信最多能有 160 个字符
    } else {
        messages = smsManager.divideMessage(mMessageText);
        // remove spaces and dashes from destination number
        // (e.g. "801 555 1212" -> "8015551212")
        // (e.g. "+8211-123-4567" -> "+82111234567")
        mDest = PhoneNumberUtils.stripSeparators(mDest);
        mDest = Conversation.verifySingleRecipient(mContext, mThreadId, mDest);
    }
    int messageCount = messages.size();

    if (messageCount == 0) {
        // Don't try to send an empty message.
        throw new MmsException("SmsMessageSender.sendMessage: divideMessage returned " +
                "empty messages. Original message is \"" + mMessageText + "\"");
    }
//把消息移到 Outbox 中发送广播
    boolean moved = Sms.moveMessageToFolder(mContext, mUri, Sms.MESSAGE_TYPE_OUTBOX, 0);
    if (!moved) {
        throw new MmsException("SmsMessageSender.sendMessage: couldn't move message " +
                "to outbox: " + mUri);
    }
    if (LogTag.DEBUG_SEND) {
        Log.v(TAG, "sendMessage mDest: " + mDest + " mRequestDeliveryReport: " +
```

```
                mRequestDeliveryReport);
        }

        ArrayList<PendingIntent> deliveryIntents =  new ArrayList<PendingIntent>
        (messageCount);
        ArrayList<PendingIntent> sentIntents = new ArrayList<PendingIntent>
        (messageCount);
        for (int i = 0; i < messageCount; i++) {
            if (mRequestDeliveryReport && (i == (messageCount - 1))) {
                // TODO: Fix: It should not be necessary to
                // specify the class in this intent.  Doing that
                // unnecessarily limits customizability.
                deliveryIntents.add(PendingIntent.getBroadcast(
                        mContext, 0,
                        new Intent(
                        //标识短信已发送，发送广播
                    MessageStatusReceiver.MESSAGE_STATUS_RECEIVED_ACTION,
                                mUri,
                                mContext,
                                MessageStatusReceiver.class),
                                0));
            } else {
                deliveryIntents.add(null);
            }
            Intent intent  = new Intent(SmsReceiverService.MESSAGE_SENT_ACTION,
                    mUri,
                    mContext,
                    SmsReceiver.class);

            int requestCode = 0;
            if (i == messageCount -1) {
                // 如果是长短信的最后一部分，则执行发送下一条短信的操作
                requestCode = 1;
                intent.putExtra(SmsReceiverService.EXTRA_MESSAGE_SENT_SEND_NEXT, true);
            }
            if (LogTag.DEBUG_SEND) {
                Log.v(TAG, "sendMessage sendIntent: " + intent);
            }
            sentIntents.add(PendingIntent.getBroadcast(mContext, requestCode, intent, 0));
        }
        try {
 //短信处理完后，交由底层来发送消息
            smsManager.sendMultipartTextMessage(mDest,    mServiceCenter,    messages,
sentIntents, deliveryIntents);
        } catch (Exception ex) {
            Log.e(TAG, "SmsMessageSender.sendMessage: caught", ex);
            throw new MmsException("SmsMessageSender.sendMessage: caught " + ex +
                    " from SmsManager.sendTextMessage()");
        }
        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE) || LogTag.DEBUG_SEND) {
            log("sendMessage: address=" + mDest + ", threadId=" + mThreadId +
                    ", uri=" + mUri + ", msgs.count=" + messageCount);
        }
        return false;
    }

    private void log(String msg) {
        Log.d(LogTag.TAG, "[SmsSingleRecipientSender] " + msg);
    }
}
```

在上述代码中，调用底层函数 sendMultipartTextMessage 来发送短信，此函数在文件“frameworks\opt\telephony\src\java\android\telephony\gsm\SmsManager.java”中实现，具体实现代码如下所示。

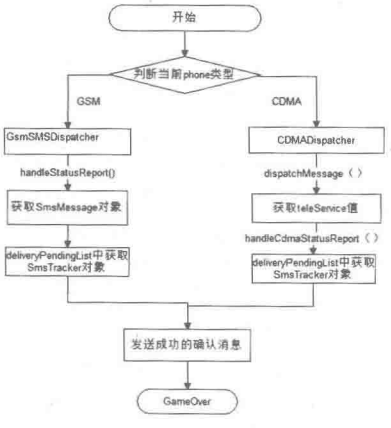
```
public final void sendMultipartTextMessage(
        String destinationAddress, String scAddress, ArrayList<String> parts,
        ArrayList<PendingIntent> sentIntents, ArrayList<PendingIntent> deliveryIntents)
{
    mSmsMgrProxy.sendMultipartTextMessage(destinationAddress, scAddress, parts,
```

```
            sentIntents, deliveryIntents);
    }
```

当 sendMultipartTextMessage 调用函数 sendMultipartText 时表示发送多文本短信信息，当 sendMultipartTextMessage 调用函数 sendText 时表示发送短的文本信息。这两个函数在文件“frameworks\opt\telephony\src\java\com\android\internal\telephony\IccSmsInterfaceManager.java”中定义，具体实现代码如下所示。

```
public void sendMultipartText(String callingPackage, String destAddr, String scAddr,
        List<String> parts, List<PendingIntent> sentIntents,
        List<PendingIntent> deliveryIntents) {
    mPhone.getContext().enforceCallingPermission(
            Manifest.permission.SEND_SMS,
            "Sending SMS message");
    if (Rlog.isLoggable("SMS", Log.VERBOSE)) {
        int i = 0;
        for (String part : parts) {
            log("sendMultipartText: destAddr=" + destAddr + ", srAddr=" + scAddr +
                    ", part[" + (i++) + "]=" + part);
        }
    }
    if (mAppOps.noteOp(AppOpsManager.OP_SEND_SMS, Binder.getCallingUid(),
            callingPackage) != AppOpsManager.MODE_ALLOWED) {
        return;
    }
    mDispatcher.sendMultipartText(destAddr, scAddr, (ArrayList<String>) parts,
            (ArrayList<PendingIntent>) sentIntents, (ArrayList<PendingIntent>) delivery
Intents);
}
public void sendText(String callingPackage, String destAddr, String scAddr,
        String text, PendingIntent sentIntent, PendingIntent deliveryIntent) {
    mPhone.getContext().enforceCallingPermission(
            Manifest.permission.SEND_SMS,
            "Sending SMS message");
    if (Rlog.isLoggable("SMS", Log.VERBOSE)) {
        log("sendText: destAddr=" + destAddr + " scAddr=" + scAddr +
            " text='"+ text + "' sentIntent=" +
            sentIntent + " deliveryIntent=" + deliveryIntent);
    }
    if (mAppOps.noteOp(AppOpsManager.OP_SEND_SMS, Binder.getCallingUid(),
            callingPackage) != AppOpsManager.MODE_ALLOWED) {
        return;
    }
    mDispatcher.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent);
}
```

到此为止，整个短信的基本发送流程介绍完毕。在接下来的内容中，将根据网络模式进行短信发送处理。在 Android 系统中，内置了 CDMA 和 GSM 两种模式，具体过程如图 17-2 所示。



▲图 17-2 CDMA 模式和 GSM 模式

CDMA 模式在文件“frameworks\opt\telephony\src\java\com\android\internal\telephony\cdma\CdmaSMSDispatcher.java”中实现，发送过程是通过依次调用函数 sendText→sendSubmitPdu→sendRawPdu 实现的，具体实现代码如下所示。

```
protected void sendText(String destAddr, String scAddr, String text,
        PendingIntent sentIntent, PendingIntent deliveryIntent) {
    SmsMessage.SubmitPdu pdu = SmsMessage.getSubmitPdu(
            scAddr, destAddr, text, (deliveryIntent != null), null);
    sendSubmitPdu(pdu, sentIntent, deliveryIntent, destAddr);
}
protected void sendSubmitPdu(SmsMessage.SubmitPdu pdu,
        PendingIntent sentIntent, PendingIntent deliveryIntent, String destAddr) {
    if (SystemProperties.getBoolean(TelephonyProperties.PROPERTY_INECM_MODE, false)) {
        if (sentIntent != null) {
            try {
                sentIntent.send(SmsManager.RESULT_ERROR_NO_SERVICE);
            } catch (CanceledException ex) {}
        }
        if (VDBG) {
            Rlog.d(TAG, "Block SMS in Emergency Callback mode");
        }
        return;
    }
    sendRawPdu(pdu.encodedScAddress, pdu.encodedMessage, sentIntent, deliveryIntent,
destAddr);
}
```

接下来将根据 sendSms 和 sendMessage 选项进行处理，在 sendSms 线路下，将调用文件“frameworks\opt\telephony\src\java\com\android\internal\telephony\RIL.java”的函数 sendCdmaSms 来发送数据信息，具体实现代码如下所示。

```
public void
sendCdmaSms(byte[] pdu, Message result) {
    int address_nbr_of_digits;
    int subaddr_nbr_of_digits;
    int bearerDataLength;
    ByteArrayInputStream bais = new ByteArrayInputStream(pdu);
    DataInputStream dis = new DataInputStream(bais);

    RILRequest rr
            = RILRequest.obtain(RIL_REQUEST_CDMA_SEND_SMS, result);

    try {
        rr.mParcel.writeInt(dis.readInt()); //teleServiceId
        rr.mParcel.writeByte((byte) dis.readInt()); //servicePresent
        rr.mParcel.writeInt(dis.readInt()); //serviceCategory
        rr.mParcel.writeInt(dis.read()); //address_digit_mode
        rr.mParcel.writeInt(dis.read()); //address_nbr_mode
        rr.mParcel.writeInt(dis.read()); //address_ton
        rr.mParcel.writeInt(dis.read()); //address_nbr_plan
        address_nbr_of_digits = (byte) dis.read();
        rr.mParcel.writeByte((byte) address_nbr_of_digits);
        for(int i=0; i < address_nbr_of_digits; i++){
            rr.mParcel.writeByte(dis.readByte()); // address_orig_bytes[i]
        }
        rr.mParcel.writeInt(dis.read()); //subaddressType
        rr.mParcel.writeByte((byte) dis.read()); //subaddr_odd
        subaddr_nbr_of_digits = (byte) dis.read();
        rr.mParcel.writeByte((byte) subaddr_nbr_of_digits);
        for(int i=0; i < subaddr_nbr_of_digits; i++){
            rr.mParcel.writeByte(dis.readByte()); //subaddr_orig_bytes[i]
        }

        bearerDataLength = dis.read();
        rr.mParcel.writeInt(bearerDataLength);
        for(int i=0; i < bearerDataLength; i++){
            rr.mParcel.writeByte(dis.readByte()); //bearerData[i]
        }
    }catch (IOException ex){
```

```
        if (RILJ_LOGD) riljLog("sendSmsCdma: conversion from input stream to object failed:
        "+ ex);
        }

        if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));

        send(rr);
    }
```

在上述代码中，通过调用文件“hardware\ril\reference-ril\reference-ril.c”中的如下 case 语句实现处理。

```
case RIL_REQUEST_CDMA_SEND_SMS:
    requestCdmaSendSMS(data, datalen, t);
    break;
```

在上述代码中，通过调用函数 requestCdmaSendSMS 实现了数据处理功能，具体实现代码如下所示。

```
static void requestCdmaSendSMS(void *data, size_t datalen, RIL_Token t)
{
    int err = 1; // Set to go to error:
    RIL_SMS_Response response;
    RIL_CDMA_SMS_Message* rcsm;

    RLOGD("requestCdmaSendSMS datalen=%d, sizeof(RIL_CDMA_SMS_Message)=%d",
           datalen, sizeof(RIL_CDMA_SMS_Message));

    // verify data content to test marshalling/unmarshalling:
    rcsm = (RIL_CDMA_SMS_Message*)data;
    RLOGD("TeleserviceID=%d, bIsServicePresent=%d, \
           uServicecategory=%d, sAddress.digit_mode=%d, \
           sAddress.Number_mode=%d, sAddress.number_type=%d, ",
           rcsm->uTeleserviceID,  rcsm->bIsServicePresent,
           rcsm->uServicecategory,rcsm->sAddress.digit_mode,
           rcsm->sAddress.number_mode,rcsm->sAddress.number_type);

    if (err != 0) goto error;

    // Cdma Send SMS implementation will go here:
    // But it is not implemented yet.

    memset(&response, 0, sizeof(response));
    RIL_onRequestComplete(t, RIL_E_SUCCESS, &response, sizeof(response));
    return;

error:
    // Cdma Send SMS will always cause send retry error.
    RIL_onRequestComplete(t, RIL_E_SMS_SEND_FAIL_RETRY, NULL, 0);
}
```

而在 sendMessage 线路下，会直接跳转到文件“hardware\ril\reference-ril\reference-ril.c”的如下 case 语句实现处理。

```
case RIL_REQUEST_CDMA_SEND_SMS:
    requestCdmaSendSMS(data, datalen, t);
    break;
```

后面的过程就和 sendSms 线路完全一致了。

GSM 模式在文件“frameworks\opt\telephony\src\java\com\android\internal\telephony\gsm\GsmSMSDispatcher.java”中实现，接下来将根据 sendSms 和 sendMessage 选项进行处理，在 sendSms 线路下，首先执行函数 sendText，具体实现代码如下所示。

```
protected void sendText(String destAddr, String scAddr, String text,
        PendingIntent sentIntent, PendingIntent deliveryIntent) {
    SmsMessage.SubmitPdu pdu = SmsMessage.getSubmitPdu(
            scAddr, destAddr, text, (deliveryIntent != null));
    if (pdu != null) {
        sendRawPdu(pdu.encodedScAddress, pdu.encodedMessage, sentIntent, deliveryIntent,
                destAddr);
```

```
    } else {
        Rlog.e(TAG, "GsmSMSDispatcher.sendText(): getSubmitPdu() returned null");
    }
}
```

然后跳转到文件“hardware\ril\reference-ril\reference-ril.c”的如下 case 语句实现处理。

```
case RIL_REQUEST_SEND_SMS:
    requestSendSMS(data, datalen, t);
    break;
```

而在 sendMessage 线路下，会直接跳转到文件“hardware\ril\reference-ril\reference-ril.c”的如下 case 语句实现处理。

```
case RIL_REQUEST_SEND_SMS:
    requestSendSMS(data, datalen, t);
    break;
```

17.3 发送彩信

在 Android 系统中，如果发送的是彩信，首先在文件“packages\apps\Mms\src\com\android\mms\data\WorkingMessage.java”中，通过函数 sendSmsWorker 创建一个 SmsMessageSender，将消息存入发送队列中并通知 SmsReceiver 发送。具体实现代码如下所示。

```
private void sendSmsWorker(String msgText, String semiSepRecipients, long threadId) {
    String[] dests = TextUtils.split(semiSepRecipients, ";");
    if (LogTag.VERBOSE || Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.d(LogTag.TRANSACTION, "sendSmsWorker sending message: recipients=" +
                semiSepRecipients + ", threadId=" + threadId);
    }
    MessageSender sender = new SmsMessageSender(mActivity, dests, msgText, threadId);
    try {
        sender.sendMessage(threadId);

        // Make sure this thread isn't over the limits in message count
        Recycler.getSmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
    } catch (Exception e) {
        Log.e(TAG, "Failed to send SMS message, threadId=" + threadId, e);
    }

    mStatusListener.onMessageSent();
    MmsWidgetProvider.notifyDatasetChanged(mActivity);
}

private void sendMmsWorker(Conversation conv, Uri mmsUri, PduPersister persister,
        SlideshowModel slideshow, SendReq sendReq, boolean textOnly) {
    long threadId = 0;
    Cursor cursor = null;
    boolean newMessage = false;
    try {
        // Put a placeholder message in the database first
        DraftCache.getInstance().setSavingDraft(true);
        mStatusListener.onPreMessageSent();

        // Make sure we are still using the correct thread ID for our
        // recipient set.
        threadId = conv.ensureThreadId();

        if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
            LogTag.debug("sendMmsWorker: update draft MMS message " + mmsUri +
                    " threadId: " + threadId);
        }

        // One last check to verify the address of the recipient.
        String[] dests = conv.getRecipients().getNumbers(true /* scrub for MMS address */);
        if (dests.length == 1) {
            // verify the single address matches what's in the database. If we get a different
            // address back, jam the new value back into the SendReq.
```

```
            String newAddress =
                Conversation.verifySingleRecipient(mActivity, conv.getThreadId(), dests[0]);

            if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
                LogTag.debug("sendMmsWorker: newAddress " + newAddress +
                        " dests[0]: " + dests[0]);
            }

            if (!newAddress.equals(dests[0])) {
                dests[0] = newAddress;
                EncodedStringValue[] encodedNumbers = EncodedStringValue.encodeStrings
                (dests);
                if (encodedNumbers != null) {
                    if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
                        LogTag.debug("sendMmsWorker: REPLACING number!!!");
                    }
                    sendReq.setTo(encodedNumbers);
                }
            }
        }
        newMessage = mmsUri == null;
        if (newMessage) {
            // Write something in the database so the new message will appear as sending
            ContentValues values = new ContentValues();
            values.put(Mms.MESSAGE_BOX, Mms.MESSAGE_BOX_OUTBOX);
            values.put(Mms.THREAD_ID, threadId);
            values.put(Mms.MESSAGE_TYPE, PduHeaders.MESSAGE_TYPE_SEND_REQ);
            if (textOnly) {
                values.put(Mms.TEXT_ONLY, 1);
            }
            mmsUri = SqliteWrapper.insert(mActivity, mContentResolver, Mms.Outbox.
            CONTENT_URI, values);
        }
        mStatusListener.onMessageSent();

        // If user tries to send the message, it's a signal the inputted text is
        // what they wanted.
        UserHappinessSignals.userAcceptedImeText(mActivity);

        // First make sure we don't have too many outstanding unsent message.
        cursor = SqliteWrapper.query(mActivity, mContentResolver,
                Mms.Outbox.CONTENT_URI, MMS_OUTBOX_PROJECTION, null, null, null);
        if (cursor != null) {
            long maxMessageSize = MmsConfig.getMaxSizeScaleForPendingMmsAllowed() *
            MmsConfig.getMaxMessageSize();
            long totalPendingSize = 0;
            while (cursor.moveToNext()) {
                totalPendingSize += cursor.getLong(MMS_MESSAGE_SIZE_INDEX);
            }
            if (totalPendingSize >= maxMessageSize) {
                unDiscard(); // it wasn't successfully sent. Allow it to be saved as a draft.
                mStatusListener.onMaxPendingMessagesReached();
                markMmsMessageWithError(mmsUri);
                return;
            }
        }
    } finally {
        if (cursor != null) {
            cursor.close();
        }
    }

    try {
        if (newMessage) {
            // Create a new MMS message if one hasn't been made yet.
            mmsUri = createDraftMmsMessage(persister, sendReq, slideshow, mmsUri,
                    mActivity, null);
        } else {
            // Otherwise, sync the MMS message in progress to disk.
            updateDraftMmsMessage(mmsUri, persister, slideshow, sendReq, null);
        }
```

```
            // Be paranoid and clean any draft SMS up.
            deleteDraftSmsMessage(threadId);
        } finally {
            DraftCache.getInstance().setSavingDraft(false);
        }

        // Resize all the resizeable attachments (e.g. pictures) to fit
        // in the remaining space in the slideshow.
        int error = 0;
        try {
            slideshow.finalResize(mmsUri);
        } catch (ExceedMessageSizeException e1) {
            error = MESSAGE_SIZE_EXCEEDED;
        } catch (MmsException e1) {
            error = UNKNOWN_ERROR;
        }
        if (error != 0) {
            markMmsMessageWithError(mmsUri);
            mStatusListener.onAttachmentError(error);
            return;
        }
        MessageSender sender = new MmsMessageSender(mActivity, mmsUri,
                slideshow.getCurrentMessageSize());
        try {
            if (!sender.sendMessage(threadId)) {
                // The message was sent through SMS protocol, we should
                // delete the copy which was previously saved in MMS drafts.
                SqliteWrapper.delete(mActivity, mContentResolver, mmsUri, null, null);
            }

            // Make sure this thread isn't over the limits in message count
            Recycler.getMmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
        } catch (Exception e) {
            Log.e(TAG, "Failed to send message: " + mmsUri + ", threadId=" + threadId, e);
        }
        MmsWidgetProvider.notifyDatasetChanged(mActivity);
    }
```

在上述代码中，调用函数 createDraftMmsMessage 把 slideshow 对象里的幻灯片信息转化成 PduPart 的字节数组，也就是把彩信里的媒体文件（图片、音频、视频和其他类型的文件）编码。函数 createDraftMmsMessage 的具体实现代码如下所示。

```
    private static Uri createDraftMmsMessage(PduPersister persister, SendReq sendReq,
            SlideshowModel slideshow, Uri preUri, Context context,
            HashMap<Uri, InputStream> preOpenedFiles) {
        if (slideshow == null) {
            return null;
        }
        try {
            PduBody pb = slideshow.toPduBody();
            sendReq.setBody(pb);
            Uri res = persister.persist(sendReq, preUri == null ? Mms.Draft.CONTENT_URI :
            preUri,
                    true, MessagingPreferenceActivity.getIsGroupMmsEnabled(context),
                    preOpenedFiles);
            slideshow.sync(pb);
            return res;
        } catch (MmsException e) {
            return null;
        }
    }
```

将彩信里的媒体文件（图片、音频、视频和其他类型的文件）进行编码的过程是通过文件“packages\apps\Mms\src\com\android\mms\model\SlideshowModel.java”实现的，具体代码如下所示。

```
    private PduBody makePduBody(SMILDocument document) {
        PduBody pb = new PduBody();

        boolean hasForwardLock = false;
```

```
    for (SlideModel slide : mSlides) {
        for (MediaModel media : slide) {
            PduPart part = new PduPart();

            if (media.isText()) {
                TextModel text = (TextModel) media;
                // Don't create empty text part.
                if (TextUtils.isEmpty(text.getText())) {
                    continue;
                }
                // Set Charset if it's a text media.
                part.setCharset(text.getCharset());
            }

            // Set Content-Type.
            part.setContentType(media.getContentType().getBytes());

            String src = media.getSrc();
            String location;
            boolean startWithContentId = src.startsWith("cid:");
            if (startWithContentId) {
                location = src.substring("cid:".length());
            } else {
                location = src;
            }

            // Set Content-Location.
            part.setContentLocation(location.getBytes());

            // Set Content-Id.
            if (startWithContentId) {
                //Keep the original Content-Id.
                part.setContentId(location.getBytes());
            }
            else {
                int index = location.lastIndexOf(".");
                String contentId = (index == -1) ? location
                        : location.substring(0, index);
                part.setContentId(contentId.getBytes());
            }

            if (media.isText()) {
                part.setData(((TextModel) media).getText().getBytes());
            } else if (media.isImage() || media.isVideo() || media.isAudio()) {
                part.setDataUri(media.getUri());
            } else {
                Log.w(TAG, "Unsupport media: " + media);
            }

            pb.addPart(part);
        }
    }

    // Create and insert SMIL part(as the first part) into the PduBody.
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    SmilXmlSerializer.serialize(document, out);
    PduPart smilPart = new PduPart();
    smilPart.setContentId("smil".getBytes());
    smilPart.setContentLocation("smil.xml".getBytes());
    smilPart.setContentType(ContentType.APP_SMIL.getBytes());
    smilPart.setData(out.toByteArray());
    pb.addPart(0, smilPart);

    return pb;
}
```

接下来会调用文件“Android 4.3\packages\apps\Mms\src\com\android\mms\transaction\MmsMessageSender.java”中的函数进行发送处理，具体实现代码如下所示。

```
public boolean sendMessage(long token) throws MmsException {
    // Load the MMS from the message uri
```

```
    if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
        LogTag.debug("sendMessage uri: " + mMessageUri);
    }
    PduPersister p = PduPersister.getPduPersister(mContext);
    GenericPdu pdu = p.load(mMessageUri);

    if (pdu.getMessageType() != PduHeaders.MESSAGE_TYPE_SEND_REQ) {
        throw new MmsException("Invalid message: " + pdu.getMessageType());
    }

    SendReq sendReq = (SendReq) pdu;

    // Update headers.
    updatePreferencesHeaders(sendReq);

    // MessageClass.
    sendReq.setMessageClass(DEFAULT_MESSAGE_CLASS.getBytes());

    // Update the 'date' field of the message before sending it.
    sendReq.setDate(System.currentTimeMillis() / 1000L);

    sendReq.setMessageSize(mMessageSize);

    p.updateHeaders(mMessageUri, sendReq);

    long messageId = ContentUris.parseId(mMessageUri);

    // Move the message into MMS Outbox.
    if (!mMessageUri.toString().startsWith(Mms.Draft.CONTENT_URI.toString())) {
        ContentValues values = new ContentValues(7);

        values.put(PendingMessages.PROTO_TYPE, MmsSms.MMS_PROTO);
        values.put(PendingMessages.MSG_ID, messageId);
        values.put(PendingMessages.MSG_TYPE, pdu.getMessageType());
        values.put(PendingMessages.ERROR_TYPE, 0);
        values.put(PendingMessages.ERROR_CODE, 0);
        values.put(PendingMessages.RETRY_INDEX, 0);
        values.put(PendingMessages.DUE_TIME, 0);

        SqliteWrapper.insert(mContext, mContext.getContentResolver(),
                PendingMessages.CONTENT_URI, values);
    } else {
        p.move(mMessageUri, Mms.Outbox.CONTENT_URI);
    }

    // Start MMS transaction service
    SendingProgressTokenManager.put(messageId, token);
    mContext.startService(new Intent(mContext, TransactionService.class));

    return true;
}
```

接下来进入核心文件“packages\apps\Mms\src\com\android\mms\transaction\TransactionService.java”，在此文件中涉及了许多判断功能，主要有网络状态判断、开启彩信网络应用判断、关闭彩信网络应用判断、发送彩信和接收彩信等，并将判断结果转给对应的文件 sendTransaction.java、NotificationTransaction.java 和 RetriveTransaction.java 进行下一步处理。其中函数 beginMmsConnectivity 用于开启彩信功能，具体实现代码如下所示。

```
protected int beginMmsConnectivity() throws IOException {
    // Take a wake lock so we don't fall asleep before the message is downloaded.
    createWakeLock();

    int result = mConnMgr.startUsingNetworkFeature(
            ConnectivityManager.TYPE_MOBILE, Phone.FEATURE_ENABLE_MMS);

    if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.v(TAG, "beginMmsConnectivity: result=" + result);
    }
```

```
        switch (result) {
            case PhoneConstants.APN_ALREADY_ACTIVE:
            case PhoneConstants.APN_REQUEST_STARTED:
                acquireWakeLock();
                return result;
        }

        throw new IOException("Cannot establish MMS connectivity");
    }
```

函数 endMmsConnectivity 用于关闭彩信功能，具体实现代码如下所示。

```
    protected void endMmsConnectivity() {
        try {
            if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
                Log.v(TAG, "endMmsConnectivity");
            }

            // cancel timer for renewal of lease
            mServiceHandler.removeMessages(EVENT_CONTINUE_MMS_CONNECTIVITY);
            if (mConnMgr != null) {
                mConnMgr.stopUsingNetworkFeature(
                        ConnectivityManager.TYPE_MOBILE,
                        Phone.FEATURE_ENABLE_MMS);
            }
        } finally {
            releaseWakeLock();
        }
    }
```

函数 onNetworkUnavailable 用于监听当前的网络是否可用，具体实现代码如下所示。

```
    private void onNetworkUnavailable(int serviceId, int transactionType) {
        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            Log.v(TAG, "onNetworkUnavailable: sid=" + serviceId + ", type=" + transactionType);
        }

        int toastType = TOAST_NONE;
        if (transactionType == Transaction.RETRIEVE_TRANSACTION) {
            toastType = TOAST_DOWNLOAD_LATER;
        } else if (transactionType == Transaction.SEND_TRANSACTION) {
            toastType = TOAST_MSG_QUEUED;
        }
        if (toastType != TOAST_NONE) {
            mToastHandler.sendEmptyMessage(toastType);
        }
        stopSelf(serviceId);
    }
```

接下来进入文件“packages\apps\Mms\src\com\android\mms\transaction\SendTransaction.java”实现线程发送处理，在此文件中实现 Runnable 接口后，在函数 run 中实现真正的发送处理功能。具体实现代码如下所示。

```
    public void process() {
        mThread = new Thread(this, "SendTransaction");
        mThread.start();
    }

    public void run() {
        try {
            RateController rateCtlr = RateController.getInstance();
            if (rateCtlr.isLimitSurpassed() && !rateCtlr.isAllowedByUser()) {
                Log.e(TAG, "Sending rate limit surpassed.");
                return;
            }

            // Load M-Send.req from outbox
            PduPersister persister = PduPersister.getPduPersister(mContext);
            SendReq sendReq = (SendReq) persister.load(mSendReqURI);

            // Update the 'date' field of the PDU right before sending it.
```

```
        long date = System.currentTimeMillis() / 1000L;
        sendReq.setDate(date);

        // Persist the new date value into database.
        ContentValues values = new ContentValues(1);
        values.put(Mms.DATE, date);
        SqliteWrapper.update(mContext, mContext.getContentResolver(),
                        mSendReqURI, values, null, null);

        // fix bug 2100169: insert the 'from' address per spec
        String lineNumber = MessageUtils.getLocalNumber();
        if (!TextUtils.isEmpty(lineNumber)) {
            sendReq.setFrom(new EncodedStringValue(lineNumber));
        }

        // Pack M-Send.req, send it, retrieve confirmation data, and parse it
        long tokenKey = ContentUris.parseId(mSendReqURI);
        byte[] response = sendPdu(SendingProgressTokenManager.get(tokenKey),
                                new PduComposer(mContext, sendReq).make());
        SendingProgressTokenManager.remove(tokenKey);

        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            String respStr = new String(response);
            Log.d(TAG, "[SendTransaction] run: send mms msg (" + mId + "), resp=" + respStr);
        }

        SendConf conf = (SendConf) new PduParser(response).parse();
        if (conf == null) {
            Log.e(TAG, "No M-Send.conf received.");
        }

        // Check whether the responding Transaction-ID is consistent
        // with the sent one.
        byte[] reqId = sendReq.getTransactionId();
        byte[] confId = conf.getTransactionId();
        if (!Arrays.equals(reqId, confId)) {
            Log.e(TAG, "Inconsistent Transaction-ID: req="
                    + new String(reqId) + ", conf=" + new String(confId));
            return;
        }

        // From now on, we won't save the whole M-Send.conf into
        // our database. Instead, we just save some interesting fields
        // into the related M-Send.req.
        values = new ContentValues(2);
        int respStatus = conf.getResponseStatus();
        values.put(Mms.RESPONSE_STATUS, respStatus);

        if (respStatus != PduHeaders.RESPONSE_STATUS_OK) {
            SqliteWrapper.update(mContext, mContext.getContentResolver(),
                            mSendReqURI, values, null, null);
            Log.e(TAG, "Server returned an error code: " + respStatus);
            return;
        }

        String messageId = PduPersister.toIsoString(conf.getMessageId());
        values.put(Mms.MESSAGE_ID, messageId);
        SqliteWrapper.update(mContext, mContext.getContentResolver(),
                        mSendReqURI, values, null, null);

        // Move M-Send.req from Outbox into Sent.
        Uri uri = persister.move(mSendReqURI, Sent.CONTENT_URI);

        mTransactionState.setState(TransactionState.SUCCESS);
        mTransactionState.setContentUri(uri);
    } catch (Throwable t) {
        Log.e(TAG, Log.getStackTraceString(t));
    } finally {
        if (mTransactionState.getState() != TransactionState.SUCCESS) {
            mTransactionState.setState(TransactionState.FAILED);
            mTransactionState.setContentUri(mSendReqURI);
```

```
            Log.e(TAG, "Delivery failed.");
        }
        notifyObservers();
    }
}
```

在上述代码中，调用了文件“packages\apps\Mms\src\com\android\mms\transaction\Transaction.java”中的函数 sendPdu，在此函数中通过 HTTP 协议来传输彩信数据。具体实现代码如下所示。

```
protected byte[] sendPdu(long token, byte[] pdu,
        String mmscUrl) throws IOException, MmsException {
    if (pdu == null) {
        throw new MmsException();
    }

    ensureRouteToHost(mmscUrl, mTransactionSettings);
    return HttpUtils.httpConnection(
            mContext, token,
            mmscUrl,
            pdu, HttpUtils.HTTP_POST_METHOD,
            mTransactionSettings.isProxySet(),
            mTransactionSettings.getProxyAddress(),
            mTransactionSettings.getProxyPort());
}
```

到此为止，整个彩信的发送过程全部结束。由此可见，彩信的发送与短信不同，是以网络的方式发送的。下面总结发送彩信的基本流程。

（1）先取出所有 due_time 在当前时间之前的待发送的彩信，然后将它的 Uri 和 transactionType 封装到 TransactionBundle 中并传给 ServiceHandler，然后将类型设置为 EVENT_TRANSACTION_REQUEST。

（2）在 ServiceHandler 中创建一个 SendTransaction 对象，然后调用 processTransaction 方法，根据当前 Transaction 是否已在队列中，以及当前的连接状态确定该把这个 SendTransaction 对象放到哪个队列中（mPending 为待发送，mProcessing 为发送中）。

（3）使用 sendMessageDelayed 方法发送一个标记为 EVENT_CONTINUE_MMS_CONNECTIVITY 的 message 来保持连接。

（4）将 TransactionService 放入该 Transaction 对象的观察者列表，以便于在后面成功发送后，继续发送待发送的彩信。

（5）使用 SendTransaction 的 Run 方法从数据库中获取指定彩信，并构造 SendReq，经由 HttpUtils 发送编码后的彩信。根据发送结果，选择是将错误状态存入数据库，还是将该彩信转到已发送箱并通知 TransactionService 处理待发送的彩信。

（6）执行 TransactionService.update 方法后，先将 Transaction 从 mProcessing 列表中移除。如果 mPending 不为空，则说明有彩信处于已基本处理但未发送状态，所以调用 mServiceHandler 设置 EVENT_HANDLE_NEXT_PENDING_TRANSACTION 进行处理。

（7）从 mPending 队列中取出第一个并交给 processTransaction 进行处理。因为调用 process Transaction 的 Transaction 都会被加入 mProcessing 队列中，而发送这个 Transaction 成功后会再次通知其观察者，进而调用 TransactionService 的 update 方法继续发送 mPending 队列中的信息，所以在 mPending 队列中的彩信会自动按顺序发完。

（8）对于成功发送的消息，使用 Notification 通知用户（包括消息未读、消息报告等）。并发送 android.intent.action.TRANSACTION_COMPLETED_ACTION 的广播。

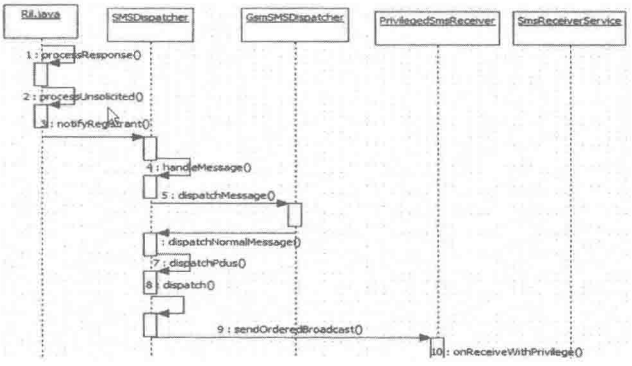
17.4 接收短信

在 Android 5.0 系统中，和发送短信相比，接收短信的过程要简单一点，整个过程涉及了如下所示的文件。

- com.android.internal.telephony/Ril.java。

- com.android.internal.telephony/SMSDispatcher。
- com.android.internal.telephony/CommandsInterface。
- com.android.internal.telephony/GsmSMSDispatcher。
- com.android.internal.telephony/CdmanSMSDispatcher。
- com.android.internal.telephony/ImsSMSDispatcher。
- hardware/ril/libril/ril.cpp。
- com.android.mms.transaction/PrivilegedSmsReceiver。

在 Android 5.0 系统中接收短信过程的流程图如图 17-3 所示。



▲图 17-3　Android 接收短信的流程图

17.4.1　Java 应用层的接收流程

在文件 Ril.java 中定义了一个完整的 receive 的框架，当用户接收到短信信息时，在底层首先通过 rild 将接收到的短信通过 socket 传输机制传送给文件 Ril.java。因为接收短信过程是一个不定时的操作，所以必须使用监听器来不间断地监视这个 socket。如果发现有短信来临，则会马上触发其相应的操作。rild 是一个守护进程，是整个 Android 系统 ril 层的入口点，定义 RILReceiver 内部类的实现代码如下所示。

```
class RILReceiver implements Runnable {
        byte[] buffer;
        RILReceiver() {
            buffer = new byte[RIL_MAX_COMMAND_BYTES];
        }
        public void
        run() {
            int retryCount = 0;
            String rilSocket = "rild";

            try {for (;;) {
                LocalSocket s = null;
                LocalSocketAddress l;

                boolean multiRild = SystemProperties.getBoolean("ro.multi.rild", false);

                if (mInstanceId == 0 || multiRild == false) {
                    rilSocket = SOCKET_NAME_RIL;
                } else {
                    rilSocket = SOCKET_NAME_RIL1;
                }
                try {
                    s = new LocalSocket();
                    l = new LocalSocketAddress(rilSocket,
                            LocalSocketAddress.Namespace.RESERVED);
                    s.connect(l);
                }
                  if (retryCount == 8) {
                      Log.e (LOG_TAG,
```

```
                        "Couldn't find '" + rilSocket
                        + "' socket after " + retryCount
                        + " times, continuing to retry silently");
                } else if (retryCount > 0 && retryCount < 8) {
                    Log.i (LOG_TAG,
                        "Couldn't find '" + rilSocket
                        + "' socket; retrying after timeout");
                }
                try {
                    Thread.sleep(SOCKET_OPEN_RETRY_MILLIS);
                } catch (InterruptedException er) {
                }
                retryCount++;
                continue;
            }
            retryCount = 0;
            mSocket = s;
            Log.i(LOG_TAG, "Connected to '" + rilSocket + "' socket");
            int length = 0;
            try {
                InputStream is = mSocket.getInputStream();
                for (;;) {
                    Parcel p;
                    length = readRilMessage(is, buffer);
                    if (length < 0) {
                        // End-of-stream reached
                        break;
                    }
                    p = Parcel.obtain();
                    p.unmarshall(buffer, 0, length);
                    p.setDataPosition(0);
                    processResponse(p);
            p.recycle();
}
```

通过上述代码可以看出上述线程会一直和守护进程 rild 实现 sorcket 通信功能，并获取了由守护进程汇报的数据。汇报工作是通过函数 processResponse 实现的，具体实现代码如下所示。

```
private void  processResponse (Parcel p) {
       int type;

       type = p.readInt();

       if (type == RESPONSE_UNSOLICITED) {
          processUnsolicited (p);
       } else if (type == RESPONSE_SOLICITED) {
          processSolicited (p);
       }
       releaseWakeLockIfDone();
    }
```

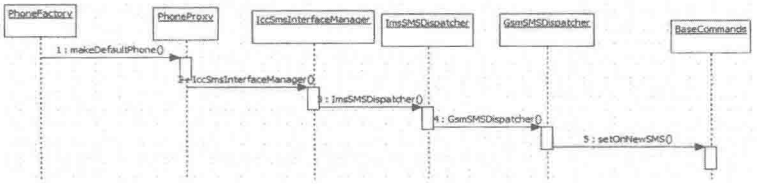
通过上述代码中对汇报的数据进行了分类处理，具体说明如下所示。

- RESPONSE_UNSOLICITED：表示接收到数据就直接汇报的类型，主动汇报的类型有网络状态、短信和来电等。
- RESPONSE_SOLICITED：表示必须先请求然后才响应的类型，此处的短信接收就是如此。
- processUnsolicited：表示该方法根据当前的请求的类型实现，如果是短信则是 RIL_UNSOL_RESPONSE_NEW_SMS，以下是其具体的调用代码。

```
case RIL_UNSOL_RESPONSE_NEW_SMS: {
            if (RILJ_LOGD) unsljLog(response);

            // FIXME this should move up a layer
            String a[] = new String[2];

            a[1] = (String)ret;

            SmsMessage sms;

            sms = SmsMessage.newFromCMT(a);
```

```
            if (mSMSRegistrant != null) {
                mSMSRegistrant
                    .notifyRegistrant(new AsyncResult(null, sms, null));
            }
            break;
            }
```

在上述代码中，创建 mSMSRegistrant 对象的过程 mSMSRegistrant 是 BaseCommands 的成员变量，并且通过调用方法 setOnNewSMS 来赋值，BaseCommands 是 Ril.java 的子类。虽然方法 setOnNewSMS 设置 handler 的源头是在 GsmSMSDispatcher 类中实现的，但是最后会调用类 SmsDispatcher 的 handMessage 方法，这是因为 GsmSMSDispatcher 是 SmsDispatcher 的子类，而且因为类 GsmSMSDispatcher 没有复写 handMessage 的方法，所以在接收到消息后肯定由父类的 handMessage 方法来处理。上述处理流程如图 17-4 所示。



▲图 17-4　处理接收信息的过程

17.4.2　Framework 层的处理过程

在 Android 系统的 Framework 层中，类 SMSDispatche 给 CommandInterface 对象设置了 handler 的处理方法，设置当接收到短信后触发函数 mSMSRegistrant.notifyRegistrant(new AsyncResult(null, sms, null))方法，然后回调之前传入的 handler，接下来在 handler 中处理短信消息。具体实现代码如下所示。

```
public void handleMessage(Message msg) {
    AsyncResult ar;
    switch (msg.what) {
    case EVENT_NEW_SMS:
        // A new SMS has been received by the device
        if (Config.LOGD) {
            Log.d(TAG, "New SMS Message Received");
        }
        SmsMessage sms;
        ar = (AsyncResult) msg.obj;
        if (ar.exception != null) {
            Log.e(TAG, "Exception processing incoming SMS. Exception:"
                    + ar.exception);
            return;
        }
        sms = (SmsMessage) ar.result;
        try {
            int result = dispatchMessage(sms.mWrappedSmsMessage);
            if (result != Activity.RESULT_OK) {
                // RESULT_OK means that message was broadcast for app(s) to
                // handle.
                // Any other result, we should ack here.
                boolean handled = (result == Intents.RESULT_SMS_HANDLED);
                notifyAndAcknowledgeLastIncomingSms(handled, result, null);
            }
        } catch (RuntimeException ex) {
            Log.e(TAG, "Exception dispatching message", ex);
            notifyAndAcknowledgeLastIncomingSms(false,
                    Intents.RESULT_SMS_GENERIC_ERROR, null);
        }
        break;
    }
}
```

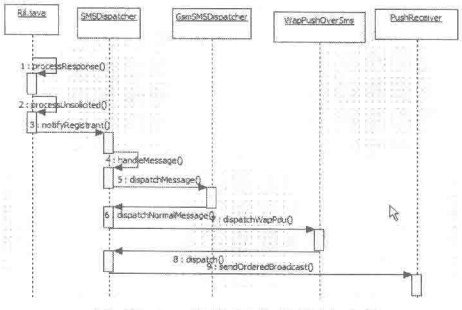
在上述代码中，“int result = dispatchMessage(sms.mWrappedSmsMessage)”代码行调用了子类（GsmSMSDispatcher）的函数 dispatchMessage 进行处理，经过判断处理将普通短信交给方法 dispatchPdus(pdus)来处理。具体实现代码如下所示。

```
protected void dispatchPdus(byte[][] pdus) {
Intent intent = new Intent(Intents.SMS_RECEIVED_ACTION);
intent.putExtra("pdus", pdus);
dispatch(intent, "android.permission.RECEIVE_SMS");
}
void dispatch(Intent intent, String permission) {
// Hold a wake lock for WAKE_LOCK_TIMEOUT seconds, enough to give any
// receivers time to take their own wake locks.
mWakeLock.acquire(WAKE_LOCK_TIMEOUT);
mContext.sendOrderedBroadcast(intent, permission, mResultReceiver,
this, Activity.RESULT_OK, null, null);
}
```

通过上述代码使用顺序广播法将短信播放出去（action 是 SMS_RECEIVED_ACTION），无论广播是否被中断，在最后都会调用 mResultReceiver 将已读或未读的状态告知给对方。如果在短信广播中间没有受到终止，则当短信应用程序中的 PrivilegedSmsReceiver 广播接收器接收到广播后会调用函数 onReceiveWithPrivilege。因为类 PrivilegedSmsReceiver 继承于类 SmsReceiver，所以会调用父类的方法 onReceiveWithPrivilege。具体实现代码如下所示。

```
protected void onReceiveWithPrivilege(Context context, Intent intent, boolean privileged)
{
        // If 'privileged' is false, it means that the intent was delivered to the base
        // no-permissions receiver class.  If we get an SMS_RECEIVED message that way, it
        // means someone has tried to spoof the message by delivering it outside the normal
        // permission-checked route, so we just ignore it.
        if (!privileged && (intent.getAction().equals(Intents.SMS_RECEIVED_ACTION)
                || intent.getAction().equals(Intents.SMS_CB_RECEIVED_ACTION))) {
            return;
        }

        intent.setClass(context, SmsReceiverService.class);
        intent.putExtra("result", getResultCode());
        beginStartingService(context, intent);
    }
```

这样接下来只需将该 Service 交给类 SmsReceiverService 去处理即可，到此，整个接收短信的过程介绍完毕。至于彩信的接收过程和上述普通短信的接收过程类似，两者的区别在于函数 dispatchMessage 会根据 smsHeader.portAddrs 来判断当前是彩信还是短信，然后调用对应的方法进行处理。具体过程如图 17-5 所示。



▲图 17-5 接收彩信的具体流程

由此可见，Android 系统的彩信接收应用层部分从 PushReceiver 开始，具体流程如下所示。

（1）当调用 onReceive 后设置让屏幕亮 5 秒，然后创建一个 ReceivePushTask，并使用它的 execute 方法。ReceivePushTask 是一个 AsyncTask，实现了 doInBack ground 方法。当传入 intent 后，会在 doInBackground 中将其中的数据转成 GenericPdu，并根据其消息类型做出不同的操作。

（2）如果是发送报告或已读报告，将其存入数据库。

（3）如果是彩信通知并且已存在，则不进行处理。否则将其存入数据库，并启动 Transaction Service 进行处理。在 TransactionService 中调用 mServiceHandler，大体过程与发送彩信时相同，只是此处创建的是 NotificationTransaction。

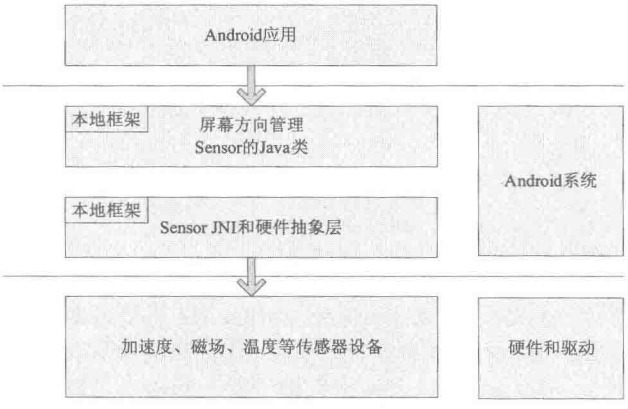
（4）如果不支持自动下载或数据传输没打开，仅通知 mmsc。否则下载相应的彩信，然后删除彩信通知，通知 mmsc 会删除超过容量限制的彩信，然后通知 TransactionService 处理其余待发送的彩信。

第 18 章　Sensor 传感器系统详解

传感器是近年来随着物联网这一概念的流行而推出的，现在人们已经逐渐认识了传感器这一概念。其实传感器在大家日常的生活中经常见到甚至是用到，比如楼宇的声控楼梯灯和马路上的路灯等。在 Android 5.0 源代码中的传感器系统是 Sensor，提供的传感器主要有：加速度传感器、磁场、方向、陀螺仪、光线、压力、温度和接近等。在本章的内容中，将详细讲解 Android 系统中传感器系统的基本知识。

18.1 Android 传感器系统概述

传感器系统会主动对上层报告传感器精度和数据的变化，并且提供了设置传感器精度的接口，这些接口可以在 Java 应用和 Java 框架中使用。Android 传感器系统的基本层次结构如图 18-1 所示。



▲图 18-1　传感器系统的层次结构

根据图 18-1 所示的结构，Android 传感器系统从上到下分别是：Java 应用层、Java 框架对传感器的应用、传感器类、传感器硬件抽象层、传感器驱动。各个层的具体说明如下所示。

（1）传感器系统的 Java 部分

代码路径是：

```
frameworks/base/include/core/java/android/hardware
```

此部分对应的实现文件是 Sensor*.java。

（2）传感器系统的 JNI 部分

代码路径是：

```
frameworks/base/core/jni/android_hardware_SensorManager.cpp
```

在此部分中提供了对类 android.hardware.Sensor.Manage 的本地支持。

（3）传感器系统 HAL 层

头文件路径是：

```
hardware/libhardware/include/hardware/sensors.h
```

在 Android 系统中，传感器系统的硬件抽象层需要特意编码实现。

（4）驱动层

驱动层的代码路径是：

```
kernel/driver/hwmon/$(PROJECT)/sensor
```

在库 sensor.so 中提供了如下所示的 8 个 API 函数。

- 控制方面：在结构体 ensors_control_device_t 中定义，包括如下所示的函数。
  - int (*open_data_source)(struct sensors_control_device_t *dev)。
  - int (*activate)(struct sensors_control_device_t *dev, int handle, int enabled)。
  - int (*set_delay)(struct sensors_control_device_t *dev, int32_t ms)。
  - int (*wake)(struct sensors_control_device_t *dev)。
- 数据方面：在结构体 sensors_data_device_t 中定义，包括如下所示的函数。
  - int (*data_open)(struct sensors_data_device_t *dev, int fd)。
  - int (*data_close)(struct sensors_data_device_t *dev)。
  - int (*poll)(struct sensors_data_device_t *dev, sensors_data_t* data)。
- 模块方面：在结构体 sensors_module_t 中定义，包括如下一个函数。

int (*get_sensors_list)(struct sensors_module_t* module, struct sensor_t const** list)。

在 Android 系统的 Java 层中，Sensor 的状态是由 SensorService 来负责控制的，其 Java 代码和 JNI 代码分别位于如下文件中。

```
frameworks/base/services/java/com/android/server/SensorService.java
frameworks/base/services/jni/com_android_server_SensorService.cpp
```

SensorManager 负责在 Java 层 Sensor 的数据控制，它的 Java 代码和 JNI 代码分别位于如下文件中。

```
frameworks/base/core/java/android/hardware/SensorManager.java
frameworks/base/core/jni/android_hardware_SensorManager.cpp
```

在 Android 的 Framework 中，是通过文件 sensorService.java 和 sensorManager.java 实现与 Sensor 传感器通信的。文件 sensorService.java 的通信功能是通过 JNI 调用 sensorService.cpp 中的方法实现的。

文件 sensorManager.java 的具体通信功能是通过 JNI 调用 sensorManager.cpp 中的方法实现的。文件 sensorService.cpp 和 sensorManager.cpp 通过文件 hardware.c 与 sensor.so 通信。其中文件 sensorService.cpp 实现对 sensor 的状态控制，文件 sensorManger.cpp 实现对 sensor 的数据控制。

库 sensor.so 通过 ioctl 控制 sensor driver 的状态，通过打开 sensor driver 对应的设备文件读取 G-sensor 采集的数据。

18.2 Java 层详解

在 Android 系统中，传感器系统的 Java 部分的实现文件是：

```
\sdk\apps\SdkController\src\com\android\tools\sdkcontroller\activities\SensorActivity.java
```

通过阅读文件 SensorActivity.java 的源代码可知，在应用程序中使用传感器需要用到 hardware 包中的 SensorManager、SensorListener 等相关的类，具体实现代码如下所示。

```
public class SensorActivity extends BaseBindingActivity
        implements android.os.Handler.Callback {

    @SuppressWarnings("hiding")
    public static String TAG = SensorActivity.class.getSimpleName();
    private static boolean DEBUG = true;

    private static final int MSG_UPDATE_ACTUAL_HZ = 0x31415;

    private TableLayout mTableLayout;
```

```
    private TextView mTextError;
    private TextView mTextStatus;
    private TextView mTextTargetHz;
    private TextView mTextActualHz;
    private SensorChannel mSensorHandler;

    private final Map<MonitoredSensor, DisplayInfo> mDisplayedSensors =
        new HashMap<SensorChannel.MonitoredSensor, SensorActivity.DisplayInfo>();
    private final android.os.Handler mUiHandler = new android.os.Handler(this);
    private int mTargetSampleRate;
    private long mLastActualUpdateMs;

    /** 第一次创建 activity 时调用 */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.sensors);
        mTableLayout = (TableLayout) findViewById(R.id.tableLayout);
        mTextError  = (TextView) findViewById(R.id.textError);
        mTextStatus = (TextView) findViewById(R.id.textStatus);
        mTextTargetHz = (TextView) findViewById(R.id.textSampleRate);
        mTextActualHz = (TextView) findViewById(R.id.textActualRate);
        updateStatus("Waiting for connection");

        mTextTargetHz.setOnKeyListener(new OnKeyListener() {
            @Override
            public boolean onKey(View v, int keyCode, KeyEvent event) {
                updateSampleRate();
                return false;
            }
        });
        mTextTargetHz.setOnFocusChangeListener(new OnFocusChangeListener() {
            @Override
            public void onFocusChange(View v, boolean hasFocus) {
                updateSampleRate();
            }
        });
    }

    @Override
    protected void onResume() {
        if (DEBUG) Log.d(TAG, "onResume");
        // BaseBindingActivity 绑定后套服务
        super.onResume();
        updateError();
    }

    @Override
    protected void onPause() {
        if (DEBUG) Log.d(TAG, "onPause");
        // BaseBindingActivity.onResume will unbind from (but not stop) the service.
        super.onPause();
    }

    @Override
    protected void onDestroy() {
        if (DEBUG) Log.d(TAG, "onDestroy");
        super.onDestroy();
        removeSensorUi();
    }

    // ----------

    @Override
    protected void onServiceConnected() {
        if (DEBUG) Log.d(TAG, "onServiceConnected");
        createSensorUi();
    }

    @Override
    protected void onServiceDisconnected() {
```

```
        if (DEBUG) Log.d(TAG, "onServiceDisconnected");
        removeSensorUi();
    }

    @Override
    protected ControllerListener createControllerListener() {
        return new SensorsControllerListener();
    }

    // ----------

    private class SensorsControllerListener implements ControllerListener {
        @Override
        public void onErrorChanged() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    updateError();
                }
            });
        }

        @Override
        public void onStatusChanged() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    ControllerBinder binder = getServiceBinder();
                    if (binder != null) {
                        boolean connected = binder.isEmuConnected();
                        mTableLayout.setEnabled(connected);
                        updateStatus(connected ? "Emulated connected" : "Emulator
                        disconnected");
                    }
                }
            });
        }
    }

    private void createSensorUi() {
        final LayoutInflater inflater = getLayoutInflater();

        if (!mDisplayedSensors.isEmpty()) {
            removeSensorUi();
        }

        mSensorHandler = (SensorChannel) getServiceBinder().getChannel(Channel.SENSOR_
        CHANNEL);
        if (mSensorHandler != null) {
            mSensorHandler.addUiHandler(mUiHandler);
            mUiHandler.sendEmptyMessage(MSG_UPDATE_ACTUAL_HZ);

            assert mDisplayedSensors.isEmpty();
            List<MonitoredSensor> sensors = mSensorHandler.getSensors();
            for (MonitoredSensor sensor : sensors) {
                final TableRow row = (TableRow) inflater.inflate(R.layout.sensor_row,
                                                                 mTableLayout,
                                                                 false);
                mTableLayout.addView(row);
                mDisplayedSensors.put(sensor, new DisplayInfo(sensor, row));
            }
        }
    }

    private void removeSensorUi() {
        if (mSensorHandler != null) {
            mSensorHandler.removeUiHandler(mUiHandler);
            mSensorHandler = null;
        }
        mTableLayout.removeAllViews();
        for (DisplayInfo info : mDisplayedSensors.values()) {
```

```
            info.release();
        }
        mDisplayedSensors.clear();
    }

    private class DisplayInfo implements CompoundButton.OnCheckedChangeListener {
        private MonitoredSensor mSensor;
        private CheckBox mChk;
        private TextView mVal;

        public DisplayInfo(MonitoredSensor sensor, TableRow row) {
            mSensor = sensor;

            // Initialize displayed checkbox for this sensor, and register
            // checked state listener for it.
            mChk = (CheckBox) row.findViewById(R.id.row_checkbox);
            mChk.setText(sensor.getUiName());
            mChk.setEnabled(sensor.isEnabledByEmulator());
            mChk.setChecked(sensor.isEnabledByUser());
            mChk.setOnCheckedChangeListener(this);

            //初始化显示该传感器的文本框
            mVal = (TextView) row.findViewById(R.id.row_textview);
            mVal.setText(sensor.getValue());
        }

        /**
         *为相关的复选框选中状态进行变化的处理。当复选框被选中时会注册传感器变化。
         *如果不加以控制会取消传感器的变化
         */
        @Override
        public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
            if (mSensor != null) {
                mSensor.onCheckedChanged(isChecked);
            }
        }

        public void release() {
            mChk = null;
            mVal = null;
            mSensor = null;

        }

        public void updateState() {
            if (mChk != null && mSensor != null) {
                mChk.setEnabled(mSensor.isEnabledByEmulator());
                mChk.setChecked(mSensor.isEnabledByUser());
            }
        }

        public void updateValue() {
            if (mVal != null && mSensor != null) {
                mVal.setText(mSensor.getValue());
            }
        }
    }

    /**实现回调处理程序*/
    @Override
    public boolean handleMessage(Message msg) {
        DisplayInfo info = null;
        switch (msg.what) {
        case SensorChannel.SENSOR_STATE_CHANGED:
            info = mDisplayedSensors.get(msg.obj);
            if (info != null) {
                info.updateState();
            }
            break;
        case SensorChannel.SENSOR_DISPLAY_MODIFIED:
            info = mDisplayedSensors.get(msg.obj);
```

```
            if (info != null) {
                info.updateValue();
            }
            if (mSensorHandler != null) {
                updateStatus(Integer.toString(mSensorHandler.getMsgSentCount()) + "
                events sent");

                //如果值已经修改则更新 "actual rate"
                long ms = mSensorHandler.getActualUpdateMs();
                if (ms != mLastActualUpdateMs) {
                    mLastActualUpdateMs = ms;
                    String hz = mLastActualUpdateMs <= 0 ? "--" :
                                Integer.toString((int) Math.ceil(1000. / ms));
                    mTextActualHz.setText(hz);
                }
            }
            break;
        case MSG_UPDATE_ACTUAL_HZ:
            if (mSensorHandler != null) {
                //如果值已经修改则更新 "actual rate"
                long ms = mSensorHandler.getActualUpdateMs();
                if (ms != mLastActualUpdateMs) {
                    mLastActualUpdateMs = ms;
                    String hz = mLastActualUpdateMs <= 0 ? "--" :
                                Integer.toString((int) Math.ceil(1000. / ms));
                    mTextActualHz.setText(hz);
                }
                mUiHandler.sendEmptyMessageDelayed(MSG_UPDATE_ACTUAL_HZ, 1000 /*1s*/);
            }
        }
        return true; // we consumed this message
    }

    private void updateStatus(String status) {
        mTextStatus.setVisibility(status == null ? View.GONE : View.VISIBLE);
        if (status != null) mTextStatus.setText(status);
    }

    private void updateError() {
        ControllerBinder binder = getServiceBinder();
        String error = binder == null ? "" : binder.getServiceError();
        if (error == null) {
            error = "";
        }

        mTextError.setVisibility(error.length() == 0 ? View.GONE : View.VISIBLE);
        mTextError.setText(error);
    }

    private void updateSampleRate() {
        String str = mTextTargetHz.getText().toString();
        try {
            int hz = Integer.parseInt(str.trim());

            // Cap the value. 50 Hz is a reasonable max value for the emulator.
            if (hz <= 0 || hz > 50) {
                hz = 50;
            }

            if (hz != mTargetSampleRate) {
                mTargetSampleRate = hz;
                if (mSensorHandler != null) {
                    mSensorHandler.setUpdateTargetMs(hz <= 0 ? 0 : (int)(1000.0f / hz));
                }
            }
        } catch (Exception ignore) {}
    }
}
```

通过上述代码可知，整个 Java 层利用了大家熟悉的观察者模式对传感器的数据进行了监听处理。

18.3 Frameworks 层详解

在 Android 系统中，传感器系统的 Frameworks 层的代码路径是。

```
frameworks/base/include/core/java/android/hardware
```

Frameworks 层是 Android 系统提供的应用程序开发接口和应用程序框架。与应用程序的调用是通过类实例化或类继承进行的。对应用程序来说，最重要的就是把 SensorListener 注册到 SensorManager 上，从而才能以观察者身份接收到数据的变化，因此，我们把目光落在 SensorManager 的构造函数、RegisterListener 函数和通知机制相关的代码上。

在本节的内容中，将详细讲解传感器系统中的 Frameworks 层的核心架构知识。

18.3.1　监听传感器的变化

在 Android 传感器系统的 Frameworks 层中，文件 SensorListener.java 用于监听从 Java 应用层中传递过来的变化。文件 SensorListener.java 比较简单，具体代码如下所示。

```
package android.hardware;
@Deprecated
public interface SensorListener {
    public void onSensorChanged(int sensor, float[] values);
    public void onAccuracyChanged(int sensor, int accuracy);
}
```

18.3.2　注册监听

当文件 SensorListener.java 监听到变化之后，会通过文件 SensorManager.java 来向服务注册监听变化，并调度 Sensor 的具体任务。例如，在开发 Android 传感器应用程序时，通用的开发流程如下所示。

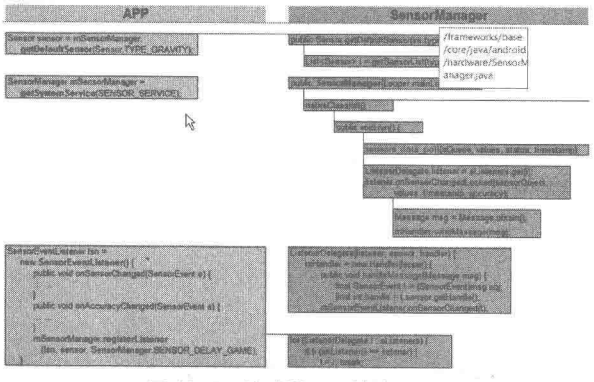
（1）通过“getSystemService(SENSOR_SERVICE)”语句得到传感器服务。这样得到一个用来管理分配调度处理 Sensor 工作的 SensorManager。SensorManager 服务并不运行于后台，真正属于 Sensor 的系统服务是 SensorService，在终端下的“#service list”中可以看到 sensorservice: [android.gui.SensorServer]。

（2）通过“getDefaultSensor(Sensor.TYPE_GRAVITY)”得到传感器类型，当然还有各种千奇百怪的传感器，具体可以查阅 Android 官方网站 API 或者源代码 Sensor.java。

（3）注册监听器 SensorEventListener。在应用程序中打开一个监听接口，专门用于处理传感器的数据。

（4）通过回调函数 onSensorChanged 和 onAccuracyChanged 实现实时监听。例如对重力感应器的 xyz 值经算法变换得到左右上下前后方向等，就由这个回调函数实现。

综上所述，传感器顶层的处理流程如图 18-2 所示。



▲图 18-2　传感器顶层的处理流程

文件 SensorManager.java 的具体实现流程如下所示。

（1）定义类 SensorManager，然后设置各种传感器的初始变量值，具体代码如下所示。

```
public abstract class SensorManager {
    protected static final String TAG = "SensorManager";
    private static final float[] mTempMatrix = new float[16];

    // Cached lists of sensors by type.  Guarded by mSensorListByType.
    private final SparseArray<List<Sensor>> mSensorListByType =
            new SparseArray<List<Sensor>>();

    // Legacy sensor manager implementation. Guarded by mSensorListByType during initialization.
    private LegacySensorManager mLegacySensorManager;
    @Deprecated
    public static final int SENSOR_ORIENTATION = 1 << 0;
    @Deprecated
    public static final int SENSOR_ACCELEROMETER = 1 << 1;
    @Deprecated
    public static final int SENSOR_TEMPERATURE = 1 << 2;
    @Deprecated
    public static final int SENSOR_MAGNETIC_FIELD = 1 << 3;
    @Deprecated
    public static final int SENSOR_LIGHT = 1 << 4;
    @Deprecated
    public static final int SENSOR_PROXIMITY = 1 << 5;
    @Deprecated
    public static final int SENSOR_TRICORDER = 1 << 6;
    @Deprecated
    public static final int SENSOR_ORIENTATION_RAW = 1 << 7;
    @Deprecated
    public static final int SENSOR_ALL = 0x7F;
    @Deprecated
    public static final int SENSOR_MIN = SENSOR_ORIENTATION;
    @Deprecated
    public static final int SENSOR_MAX = ((SENSOR_ALL + 1)>>1);

    @Deprecated
    public static final int DATA_X = 0;
    @Deprecated
    public static final int DATA_Y = 1;
    @Deprecated
    public static final int DATA_Z = 2;
    @Deprecated
    public static final int RAW_DATA_INDEX = 3;
    @Deprecated
    public static final int RAW_DATA_X = 3;
    @Deprecated
    public static final int RAW_DATA_Y = 4;
    @Deprecated
    public static final int RAW_DATA_Z = 5;

    /** Standard gravity (g) on Earth. This value is equivalent to 1G */
    public static final float STANDARD_GRAVITY = 9.80665f;

    /** Sun's gravity in SI units (m/s^2) */
    public static final float GRAVITY_SUN           = 2718.0f;
    /** Mercury's gravity in SI units (m/s^2) */
    public static final float GRAVITY_MERCURY        = 3.70f;
    /** Venus' gravity in SI units (m/s^2) */
    public static final float GRAVITY_VENUS          = 8.87f;
    /** Earth's gravity in SI units (m/s^2) */
    public static final float GRAVITY_EARTH          = 9.80665f;
    /** The Moon's gravity in SI units (m/s^2) */
    public static final float GRAVITY_MOON           = 1.6f;
    /** Mars' gravity in SI units (m/s^2) */
    public static final float GRAVITY_MARS           = 3.71f;
    /** Jupiter's gravity in SI units (m/s^2) */
    public static final float GRAVITY_JUPITER        = 23.12f;
    /** Saturn's gravity in SI units (m/s^2) */
    public static final float GRAVITY_SATURN         = 8.96f;
```

```
/** Uranus' gravity in SI units (m/s^2) */
public static final float GRAVITY_URANUS         = 8.69f;
/** Neptune's gravity in SI units (m/s^2) */
public static final float GRAVITY_NEPTUNE        = 11.0f;
/** Pluto's gravity in SI units (m/s^2) */
public static final float GRAVITY_PLUTO          = 0.6f;
/** Gravity (estimate) on the first Death Star in Empire units (m/s^2) */
public static final float GRAVITY_DEATH_STAR_I    = 0.000000353036145f;
/** Gravity on the island */
public static final float GRAVITY_THE_ISLAND     = 4.815162342f;


/** Maximum magnetic field on Earth's surface */
public static final float MAGNETIC_FIELD_EARTH_MAX = 60.0f;
/** Minimum magnetic field on Earth's surface */
public static final float MAGNETIC_FIELD_EARTH_MIN = 30.0f;


/** Standard atmosphere, or average sea-level pressure in hPa (millibar) */
public static final float PRESSURE_STANDARD_ATMOSPHERE = 1013.25f;


/** Maximum luminance of sunlight in lux */
public static final float LIGHT_SUNLIGHT_MAX = 120000.0f;
/** luminance of sunlight in lux */
public static final float LIGHT_SUNLIGHT     = 110000.0f;
/** luminance in shade in lux */
public static final float LIGHT_SHADE       = 20000.0f;
/** luminance under an overcast sky in lux */
public static final float LIGHT_OVERCAST     = 10000.0f;
/** luminance at sunrise in lux */
public static final float LIGHT_SUNRISE      = 400.0f;
/** luminance under a cloudy sky in lux */
public static final float LIGHT_CLOUDY       = 100.0f;
/** luminance at night with full moon in lux */
public static final float LIGHT_FULLMOON     = 0.25f;
/** luminance at night with no moon in lux*/
public static final float LIGHT_NO_MOON      = 0.001f;


/** get sensor data as fast as possible */
public static final int SENSOR_DELAY_FASTEST = 0;
/** rate suitable for games */
public static final int SENSOR_DELAY_GAME = 1;
/** rate suitable for the user interface  */
public static final int SENSOR_DELAY_UI = 2;
/**（默认值）适合屏幕方向的变化*/
public static final int SENSOR_DELAY_NORMAL = 3;


/**
 *返回的值，该传感器是不可信的，需要进行校准或环境不允许读数
 */
public static final int SENSOR_STATUS_UNRELIABLE = 0;

/**
 *该传感器报告的是低精度的数据，必须与环境进行校准
 */
public static final int SENSOR_STATUS_ACCURACY_LOW = 1;

/**
 * This sensor is reporting data with an average level of accuracy,
 * calibration with the environment may improve the readings
 */
public static final int SENSOR_STATUS_ACCURACY_MEDIUM = 2;

/** This sensor is reporting data with maximum accuracy */
public static final int SENSOR_STATUS_ACCURACY_HIGH = 3;

/** see {@link #remapCoordinateSystem} */
public static final int AXIS_X = 1;
```

```
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_Y = 2;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_Z = 3;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_X = AXIS_X | 0x80;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_Y = AXIS_Y | 0x80;
/** see {@link #remapCoordinateSystem} */
public static final int AXIS_MINUS_Z = AXIS_Z | 0x80;
```

（2）定义各种设备类型方法和设备数据的方法，这些方法非常重要，在我们编写的应用程序中，可以通过 AIDL 接口远程调用（RPC）的方式得到 SensorManager。这样通过在类 SensorManager 中的方法，可以得到底层的各种传感器数据。上述方法的具体实现代码如下所示。

```
public int getSensors() {
    return getLegacySensorManager().getSensors();
}
public List<Sensor> getSensorList(int type) {
    // cache the returned lists the first time
    List<Sensor> list;
    final List<Sensor> fullList = getFullSensorList();
    synchronized (mSensorListByType) {
        list = mSensorListByType.get(type);
        if (list == null) {
            if (type == Sensor.TYPE_ALL) {
                list = fullList;
            } else {
                list = new ArrayList<Sensor>();
                for (Sensor i : fullList) {
                    if (i.getType() == type)
                        list.add(i);
                }
            }
            list = Collections.unmodifiableList(list);
            mSensorListByType.append(type, list);
        }
    }
    return list;
}
public Sensor getDefaultSensor(int type) {
    // TODO: need to be smarter, for now, just return the 1st sensor
    List<Sensor> l = getSensorList(type);
    return l.isEmpty() ? null : l.get(0);
}
@Deprecated
public boolean registerListener(SensorListener listener, int sensors) {
    return registerListener(listener, sensors, SENSOR_DELAY_NORMAL);
}
@Deprecated
public boolean registerListener(SensorListener listener, int sensors, int rate) {
    return getLegacySensorManager().registerListener(listener, sensors, rate);
}
@Deprecated
public void unregisterListener(SensorListener listener) {
    unregisterListener(listener, SENSOR_ALL | SENSOR_ORIENTATION_RAW);
}
@Deprecated
public void unregisterListener(SensorListener listener, int sensors) {
    getLegacySensorManager().unregisterListener(listener, sensors);
}
public void unregisterListener(SensorEventListener listener, Sensor sensor) {
    if (listener == null || sensor == null) {
        return;
    }
    unregisterListenerImpl(listener, sensor);
}
public void unregisterListener(SensorEventListener listener) {
    if (listener == null) {
        return;
```

```
        }
        unregisterListenerImpl(listener, null);
    }
    protected abstract void unregisterListenerImpl(SensorEventListener listener, Sensor sensor);
    public boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)
{
        return registerListener(listener, sensor, rate, null);
    }
    public boolean registerListener(SensorEventListener listener, Sensor sensor, int rate,
            Handler handler) {
        if (listener == null || sensor == null) {
            return false;
        }

        int delay = -1;
        switch (rate) {
            case SENSOR_DELAY_FASTEST:
                delay = 0;
                break;
            case SENSOR_DELAY_GAME:
                delay = 20000;
                break;
            case SENSOR_DELAY_UI:
                delay = 66667;
                break;
            case SENSOR_DELAY_NORMAL:
                delay = 200000;
                break;
            default:
                delay = rate;
                break;
        }
        return registerListenerImpl(listener, sensor, delay, handler);
    }
    protected abstract boolean registerListenerImpl(SensorEventListener listener, Sensor
    sensor, int delay, Handler handler);

    public static boolean getRotationMatrix(float[] R, float[] I,
            float[] gravity, float[] geomagnetic) {
        // TODO: move this to native code for efficiency
        float Ax = gravity[0];
        float Ay = gravity[1];
        float Az = gravity[2];
        final float Ex = geomagnetic[0];
        final float Ey = geomagnetic[1];
        final float Ez = geomagnetic[2];
        float Hx = Ey*Az - Ez*Ay;
        float Hy = Ez*Ax - Ex*Az;
        float Hz = Ex*Ay - Ey*Ax;
        final float normH = (float)Math.sqrt(Hx*Hx + Hy*Hy + Hz*Hz);
        if (normH < 0.1f) {
            // device is close to free fall (or in space?), or close to
            // magnetic north pole. Typical values are  > 100.
            return false;
        }
        final float invH = 1.0f / normH;
        Hx *= invH;
        Hy *= invH;
        Hz *= invH;
        final float invA = 1.0f / (float)Math.sqrt(Ax*Ax + Ay*Ay + Az*Az);
        Ax *= invA;
        Ay *= invA;
        Az *= invA;
        final float Mx = Ay*Hz - Az*Hy;
        final float My = Az*Hx - Ax*Hz;
        final float Mz = Ax*Hy - Ay*Hx;
        if (R != null) {
            if (R.length == 9) {
                R[0] = Hx;     R[1] = Hy;     R[2] = Hz;
                R[3] = Mx;     R[4] = My;     R[5] = Mz;
                R[6] = Ax;     R[7] = Ay;     R[8] = Az;
```

```
        } else if (R.length == 16) {
            R[0]  = Hx;    R[1]  = Hy;    R[2]  = Hz;   R[3]  = 0;
            R[4]  = Mx;    R[5]  = My;    R[6]  = Mz;   R[7]  = 0;
            R[8]  = Ax;    R[9]  = Ay;    R[10] = Az;   R[11] = 0;
            R[12] = 0;     R[13] = 0;     R[14] = 0;    R[15] = 1;
        }
    }
    if (I != null) {
        // compute the inclination matrix by projecting the geomagnetic
        // vector onto the Z (gravity) and X (horizontal component
        // of geomagnetic vector) axes.
        final float invE = 1.0f / (float)Math.sqrt(Ex*Ex + Ey*Ey + Ez*Ez);
        final float c = (Ex*Mx + Ey*My + Ez*Mz) * invE;
        final float s = (Ex*Ax + Ey*Ay + Ez*Az) * invE;
        if (I.length == 9) {
            I[0] = 1;     I[1] = 0;     I[2] = 0;
            I[3] = 0;     I[4] = c;     I[5] = s;
            I[6] = 0;     I[7] =-s;     I[8] = c;
        } else if (I.length == 16) {
            I[0] = 1;     I[1] = 0;     I[2] = 0;
            I[4] = 0;     I[5] = c;     I[6] = s;
            I[8] = 0;     I[9] =-s;     I[10]= c;
            I[3] = I[7] = I[11] = I[12] = I[13] = I[14] = 0;
            I[15] = 1;
        }
    }
    return true;
}
public static float getInclination(float[] I) {
    if (I.length == 9) {
        return (float)Math.atan2(I[5], I[4]);
    } else {
        return (float)Math.atan2(I[6], I[5]);
    }
}

public static boolean remapCoordinateSystem(float[] inR, int X, int Y,
        float[] outR)
{
    if (inR == outR) {
        final float[] temp = mTempMatrix;
        synchronized(temp) {
            // we don't expect to have a lot of contention
            if (remapCoordinateSystemImpl(inR, X, Y, temp)) {
                final int size = outR.length;
                for (int i=0 ; i<size ; i++)
                    outR[i] = temp[i];
                return true;
            }
        }
    }
    return remapCoordinateSystemImpl(inR, X, Y, outR);
}

private static boolean remapCoordinateSystemImpl(float[] inR, int X, int Y,
        float[] outR)
{

    final int length = outR.length;
    if (inR.length != length)
        return false;   // invalid parameter
    if ((X & 0x7C)!=0 || (Y & 0x7C)!=0)
        return false;   // invalid parameter
    if (((X & 0x3)==0) || ((Y & 0x3)==0))
        return false;   // no axis specified
    if ((X & 0x3) == (Y & 0x3))
        return false;   // same axis specified
    int Z = X ^ Y;

    // extract the axis (remove the sign), offset in the range 0 to 2.
    final int x = (X & 0x3)-1;
```

```
        final int y = (Y & 0x3)-1;
        final int z = (Z & 0x3)-1;

        // compute the sign of Z (whether it needs to be inverted)
        final int axis_y = (z+1)%3;
        final int axis_z = (z+2)%3;
        if (((x^axis_y)|(y^axis_z)) != 0)
            Z ^= 0x80;

        final boolean sx = (X>=0x80);
        final boolean sy = (Y>=0x80);
        final boolean sz = (Z>=0x80);

        // Perform R * r, in avoiding actual muls and adds.
        final int rowLength = ((length==16)?4:3);
        for (int j=0 ; j<3 ; j++) {
            final int offset = j*rowLength;
            for (int i=0 ; i<3 ; i++) {
                if (x==i)   outR[offset+i] = sx ? -inR[offset+0] : inR[offset+0];
                if (y==i)   outR[offset+i] = sy ? -inR[offset+1] : inR[offset+1];
                if (z==i)   outR[offset+i] = sz ? -inR[offset+2] : inR[offset+2];
            }
        }
        if (length == 16) {
            outR[3] = outR[7] = outR[11] = outR[12] = outR[13] = outR[14] = 0;
            outR[15] = 1;
        }
        return true;
    }
    public static float[] getOrientation(float[] R, float values[]) {

        if (R.length == 9) {
            values[0] = (float)Math.atan2(R[1], R[4]);
            values[1] = (float)Math.asin(-R[7]);
            values[2] = (float)Math.atan2(-R[6], R[8]);
        } else {
            values[0] = (float)Math.atan2(R[1], R[5]);
            values[1] = (float)Math.asin(-R[9]);
            values[2] = (float)Math.atan2(-R[8], R[10]);
        }
        return values;
    }
    public static float getAltitude(float p0, float p) {
        final float coef = 1.0f / 18.255f;
        return 44330.0f * (1.0f - (float)Math.pow(p/p0, coef));
    }

    public static void getAngleChange( float[] angleChange, float[] R, float[] prevR) {
        float rd1=0,rd4=0, rd6=0,rd7=0, rd8=0;
        float ri0=0,ri1=0,ri2=0,ri3=0,ri4=0,ri5=0,ri6=0,ri7=0,ri8=0;
        float pri0=0, pri1=0, pri2=0, pri3=0, pri4=0, pri5=0, pri6=0, pri7=0, pri8=0;

        if(R.length == 9) {
            ri0 = R[0];
            ri1 = R[1];
            ri2 = R[2];
            ri3 = R[3];
            ri4 = R[4];
            ri5 = R[5];
            ri6 = R[6];
            ri7 = R[7];
            ri8 = R[8];
        } else if(R.length == 16) {
            ri0 = R[0];
            ri1 = R[1];
            ri2 = R[2];
            ri3 = R[4];
            ri4 = R[5];
            ri5 = R[6];
            ri6 = R[8];
            ri7 = R[9];
```

```
            ri8 = R[10];
        }

        if(prevR.length == 9) {
            pri0 = prevR[0];
            pri1 = prevR[1];
            pri2 = prevR[2];
            pri3 = prevR[3];
            pri4 = prevR[4];
            pri5 = prevR[5];
            pri6 = prevR[6];
            pri7 = prevR[7];
            pri8 = prevR[8];
        } else if(prevR.length == 16) {
            pri0 = prevR[0];
            pri1 = prevR[1];
            pri2 = prevR[2];
            pri3 = prevR[4];
            pri4 = prevR[5];
            pri5 = prevR[6];
            pri6 = prevR[8];
            pri7 = prevR[9];
            pri8 = prevR[10];
        }
        rd1 = pri0 * ri1 + pri3 * ri4 + pri6 * ri7; //rd[0][1]
        rd4 = pri1 * ri1 + pri4 * ri4 + pri7 * ri7; //rd[1][1]
        rd6 = pri2 * ri0 + pri5 * ri3 + pri8 * ri6; //rd[2][0]
        rd7 = pri2 * ri1 + pri5 * ri4 + pri8 * ri7; //rd[2][1]
        rd8 = pri2 * ri2 + pri5 * ri5 + pri8 * ri8; //rd[2][2]

        angleChange[0] = (float)Math.atan2(rd1, rd4);
        angleChange[1] = (float)Math.asin(-rd7);
        angleChange[2] = (float)Math.atan2(-rd6, rd8);
    }
    public static void getRotationMatrixFromVector(float[] R, float[] rotationVector) {

        float q0;
        float q1 = rotationVector[0];
        float q2 = rotationVector[1];
        float q3 = rotationVector[2];

        if (rotationVector.length == 4) {
            q0 = rotationVector[3];
        } else {
            q0 = 1 - q1*q1 - q2*q2 - q3*q3;
            q0 = (q0 > 0) ? (float)Math.sqrt(q0) : 0;
        }

        float sq_q1 = 2 * q1 * q1;
        float sq_q2 = 2 * q2 * q2;
        float sq_q3 = 2 * q3 * q3;
        float q1_q2 = 2 * q1 * q2;
        float q3_q0 = 2 * q3 * q0;
        float q1_q3 = 2 * q1 * q3;
        float q2_q0 = 2 * q2 * q0;
        float q2_q3 = 2 * q2 * q3;
        float q1_q0 = 2 * q1 * q0;

        if(R.length == 9) {
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
            R[2] = q1_q3 + q2_q0;

            R[3] = q1_q2 + q3_q0;
            R[4] = 1 - sq_q1 - sq_q3;
            R[5] = q2_q3 - q1_q0;

            R[6] = q1_q3 - q2_q0;
            R[7] = q2_q3 + q1_q0;
            R[8] = 1 - sq_q1 - sq_q2;
        } else if (R.length == 16) {
```

```
            R[0] = 1 - sq_q2 - sq_q3;
            R[1] = q1_q2 - q3_q0;
            R[2] = q1_q3 + q2_q0;
            R[3] = 0.0f;

            R[4] = q1_q2 + q3_q0;
            R[5] = 1 - sq_q1 - sq_q3;
            R[6] = q2_q3 - q1_q0;
            R[7] = 0.0f;

            R[8] = q1_q3 - q2_q0;
            R[9] = q2_q3 + q1_q0;
            R[10] = 1 - sq_q1 - sq_q2;
            R[11] = 0.0f;

            R[12] = R[13] = R[14] = 0.0f;
            R[15] = 1.0f;
        }
    }
    public static void getQuaternionFromVector(float[] Q, float[] rv) {
        if (rv.length == 4) {
            Q[0] = rv[3];
        } else {
            Q[0] = 1 - rv[0]*rv[0] - rv[1]*rv[1] - rv[2]*rv[2];
            Q[0] = (Q[0] > 0) ? (float)Math.sqrt(Q[0]) : 0;
        }
        Q[1] = rv[0];
        Q[2] = rv[1];
        Q[3] = rv[2];
    }
    public boolean requestTriggerSensor(TriggerEventListener listener, Sensor sensor) {
        return requestTriggerSensorImpl(listener, sensor);
    }
    protected abstract boolean requestTriggerSensorImpl(TriggerEventListener listener,
            Sensor sensor);
    public boolean cancelTriggerSensor(TriggerEventListener listener, Sensor sensor) {
        return cancelTriggerSensorImpl(listener, sensor, true);
    }
    protected abstract boolean cancelTriggerSensorImpl(TriggerEventListener listener,
            Sensor sensor, boolean disable);
    private LegacySensorManager getLegacySensorManager() {
        synchronized (mSensorListByType) {
            if (mLegacySensorManager == null) {
                Log.i(TAG, "This application is using deprecated SensorManager API which
will "
                        + "be removed someday.  Please consider switching to the new API.");
                mLegacySensorManager = new LegacySensorManager(this);
            }
            return mLegacySensorManager;
        }
    }
}
```

上述代码方法其实就是我们在开发传感器应用程序时用到的 API 接口。有关上述方法的具体说明，读者可以查阅 Android 官方网站 SDK API 中对类 android.hardware.SensorManager 的具体说明，如图 18-3 所示。

18.4 JNI 层详解

在 Android 系统中，传感器系统的 JNI 部分的代码路径是。

```
frameworks/base/core/jni/android_hardware_SensorManager.cpp
```

在此文件中提供了对类 android.hardware.Sensor.Manage 的本地支持。系统上层和 JNI 层的调用关系如图 18-4 所示。

在图 8-4 所示的调用关系中，涉及了如下所示的 API 接口方法。

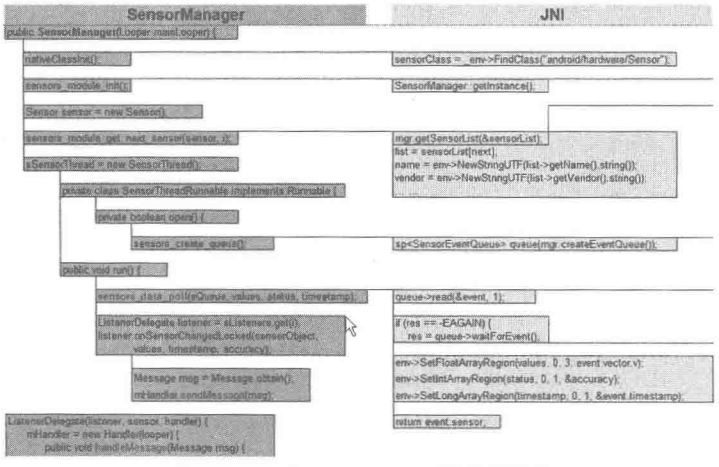
- nativeClassInit()：在 JNI 层得到 android.hardware.Sensor 的 JNI 环境指针。

Reference | Resources | Videos | Blog

Public Methods

static float	getAltitude(float p0, float p) Computes the Altitude in meters from the atmospheric pressure and the pressure at sea level.
static void	getAngleChange(float[] angleChange, float[] R, float[] prevR) Helper function to compute the angle change between two rotation matrices.
Sensor	getDefaultSensor(int type) Use this method to get the default sensor for a given type.
static float	getInclination(float[] I) Computes the geomagnetic inclination angle in radians from the inclination matrix I returned by getR
static float[]	getOrientation(float[] R, float[] values) Computes the device's orientation based on the rotation matrix.
static void	getQuaternionFromVector(float[] Q, float[] rv) Helper function to convert a rotation vector to a normalized quaternion.
static boolean	getRotationMatrix(float[] R, float[] I, float[] gravity, float[] geomagnetic) Computes the inclination matrix I as well as the rotation matrix R transforming a vector from the de • X is defined as the vector product Y.Z (it is tangential to the ground at the device's current locat
static void	getRotationMatrixFromVector(float[] R, float[] rotationVector) Helper function to convert a rotation vector to a rotation matrix.
List<Sensor>	getSensorList(int type) Use this method to get the list of available sensors of a certain type.
int	getSensors() *This method is deprecated. This method is deprecated, use `getSensorList(int)` instead*
boolean	registerListener(SensorListener listener, int sensors, int rate) *This method is deprecated. This method is deprecated, use `registerListener(SensorEventListene*`

▲图 18-3　SDK API 中对于类 android.hardware.SensorManager 的具体说明



▲图 18-4　系统上层和 JNI 层的调用关系

- sensors_module_init()：通过 JNI 调用本地框架，得到 SensorService，SensorService 初始化控制流各功能。
- new Sensor()：建立一个 Sensor 对象，具体可查阅官方网站 API android.hardware.Sensor。
- sensors_module_get_next_sensor()：上层得到设备支持的所有 Sensor，并放入 SensorList 链表。
- new SensorThread()：创建 Sensor 线程，当应用程序 registerListener()注册监听器时开启线程 run()，注意，当没有数据变化时线程会阻塞。

18.4.1　实现 Native（本地）函数

文件 android_hardware_SensorManager.cpp 的功能是实现文件 SensorManager.java 中的 Native（本地）函数，主要是通过调用文件 SenrsorManager.cpp 和文件 SensorEventQueue.cpp 中的相关类来完成相关工作的。文件 android_hardware_SensorManager.cpp 的具体实现代码如下所示。

```
static struct {
    jclass clazz;
    jmethodID dispatchSensorEvent;
} gBaseEventQueueClassInfo;
```

```
namespace android {

struct SensorOffsets
{
    jfieldID    name;
    jfieldID    vendor;
    jfieldID    version;
    jfieldID    handle;
    jfieldID    type;
    jfieldID    range;
    jfieldID    resolution;
    jfieldID    power;
    jfieldID    minDelay;
} gSensorOffsets;


/*
 * The method below are not thread-safe and not intended to be
 */

static void
nativeClassInit (JNIEnv *_env, jclass _this)
{
    jclass sensorClass = _env->FindClass("android/hardware/Sensor");
    SensorOffsets& sensorOffsets = gSensorOffsets;
    sensorOffsets.name = _env->GetFieldID(sensorClass, "mName", "Ljava/lang/String;");
    sensorOffsets.vendor = _env->GetFieldID(sensorClass, "mVendor","Ljava/lang/String;");
    sensorOffsets.version = _env->GetFieldID(sensorClass, "mVersion",   "I");
    sensorOffsets.handle = _env->GetFieldID(sensorClass, "mHandle",    "I");
    sensorOffsets.type = _env->GetFieldID(sensorClass, "mType",      "I");
    sensorOffsets.range = _env->GetFieldID(sensorClass, "mMaxRange",  "F");
    sensorOffsets.resolution = _env->GetFieldID(sensorClass, "mResolution","F");
    sensorOffsets.power = _env->GetFieldID(sensorClass, "mPower",     "F");
    sensorOffsets.minDelay = _env->GetFieldID(sensorClass, "mMinDelay",  "I");
}

static jint
nativeGetNextSensor(JNIEnv *env, jclass clazz, jobject sensor, jint next)
{
    SensorManager& mgr(SensorManager::getInstance());

    Sensor const* const* sensorList;
    size_t count = mgr.getSensorList(&sensorList);
    if (size_t(next) >= count)
        return -1;

    Sensor const* const list = sensorList[next];
    const SensorOffsets& sensorOffsets(gSensorOffsets);
    jstring name = env->NewStringUTF(list->getName().string());
    jstring vendor = env->NewStringUTF(list->getVendor().string());
    env->SetObjectField(sensor, sensorOffsets.name,     name);
    env->SetObjectField(sensor, sensorOffsets.vendor,    vendor);
    env->SetIntField(sensor, sensorOffsets.version,     list->getVersion());
    env->SetIntField(sensor, sensorOffsets.handle,      list->getHandle());
    env->SetIntField(sensor, sensorOffsets.type,        list->getType());
    env->SetFloatField(sensor, sensorOffsets.range,     list->getMaxValue());
    env->SetFloatField(sensor, sensorOffsets.resolution, list->getResolution());
    env->SetFloatField(sensor, sensorOffsets.power,     list->getPowerUsage());
    env->SetIntField(sensor, sensorOffsets.minDelay,    list->getMinDelay());

    next++;
    return size_t(next) < count ? next : 0;
}

//------------------------------------------------------------------------------

class Receiver : public LooperCallback {
    sp<SensorEventQueue> mSensorQueue;
    sp<MessageQueue> mMessageQueue;
    jobject mReceiverObject;
    jfloatArray mScratch;
```

```
public:
    Receiver(const sp<SensorEventQueue>& sensorQueue,
            const sp<MessageQueue>& messageQueue,
            jobject receiverObject, jfloatArray scratch) {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        mSensorQueue = sensorQueue;
        mMessageQueue = messageQueue;
        mReceiverObject = env->NewGlobalRef(receiverObject);
        mScratch = (jfloatArray)env->NewGlobalRef(scratch);
    }
    ~Receiver() {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        env->DeleteGlobalRef(mReceiverObject);
        env->DeleteGlobalRef(mScratch);
    }
    sp<SensorEventQueue> getSensorEventQueue() const {
        return mSensorQueue;
    }

    void destroy() {
        mMessageQueue->getLooper()->removeFd( mSensorQueue->getFd() );
    }

private:
    virtual void onFirstRef() {
        LooperCallback::onFirstRef();
        mMessageQueue->getLooper()->addFd(mSensorQueue->getFd(), 0,
                ALOOPER_EVENT_INPUT, this, mSensorQueue.get());
    }

    virtual int handleEvent(int fd, int events, void* data) {
        JNIEnv* env = AndroidRuntime::getJNIEnv();
        sp<SensorEventQueue> q = reinterpret_cast<SensorEventQueue *>(data);
        ssize_t n;
        ASensorEvent buffer[16];
        while ((n = q->read(buffer, 16)) > 0) {
            for (int i=0 ; i<n ; i++) {

                env->SetFloatArrayRegion(mScratch, 0, 16, buffer[i].data);

                env->CallVoidMethod(mReceiverObject,
                        gBaseEventQueueClassInfo.dispatchSensorEvent,
                        buffer[i].sensor,
                        mScratch,
                        buffer[i].vector.status,
                        buffer[i].timestamp);

                if (env->ExceptionCheck()) {
                    ALOGE("Exception dispatching input event.");
                    return 1;
                }
            }
        }
        if (n<0 && n != -EAGAIN) {
            // FIXME: error receiving events, what to do in this case?
        }

        return 1;
    }
};

static jint nativeInitSensorEventQueue(JNIEnv *env, jclass clazz, jobject eventQ, jobject
msgQ, jfloatArray scratch) {
    SensorManager& mgr(SensorManager::getInstance());
    sp<SensorEventQueue> queue(mgr.createEventQueue());

    sp<MessageQueue> messageQueue = android_os_MessageQueue_getMessageQueue(env, msgQ);
    if (messageQueue == NULL) {
        jniThrowRuntimeException(env, "MessageQueue is not initialized.");
        return 0;
    }
```

```
    sp<Receiver> receiver = new Receiver(queue, messageQueue, eventQ, scratch);
    receiver->incStrong((void*)nativeInitSensorEventQueue);
    return jint(receiver.get());
}

static jint nativeEnableSensor(JNIEnv *env, jclass clazz, jint eventQ, jint handle, jint
us) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    return receiver->getSensorEventQueue()->enableSensor(handle, us);
}

static jint nativeDisableSensor(JNIEnv *env, jclass clazz, jint eventQ, jint handle) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    return receiver->getSensorEventQueue()->disableSensor(handle);
}

static void nativeDestroySensorEventQueue(JNIEnv *env, jclass clazz, jint eventQ, jint
handle) {
    sp<Receiver> receiver(reinterpret_cast<Receiver *>(eventQ));
    receiver->destroy();
    receiver->decStrong((void*)nativeInitSensorEventQueue);
}


//-----------------------------------------------------------------------------

static JNINativeMethod gSystemSensorManagerMethods[] = {
    {"nativeClassInit",
            "()V",
            (void*)nativeClassInit },

    {"nativeGetNextSensor",
            "(Landroid/hardware/Sensor;I)I",
            (void*)nativeGetNextSensor },
};

static JNINativeMethod gBaseEventQueueMethods[] = {
    {"nativeInitBaseEventQueue",

"(Landroid/hardware/SystemSensorManager$BaseEventQueue;Landroid/os/MessageQueue;[F)I",
            (void*)nativeInitSensorEventQueue },

    {"nativeEnableSensor",
            "(III)I",
            (void*)nativeEnableSensor },

    {"nativeDisableSensor",
            "(II)I",
            (void*)nativeDisableSensor },

    {"nativeDestroySensorEventQueue",
            "(I)V",
            (void*)nativeDestroySensorEventQueue },
};

}; // namespace android

using namespace android;

#define FIND_CLASS(var, className) \
        var = env->FindClass(className); \
        LOG_FATAL_IF(! var, "Unable to find class " className); \
        var = jclass(env->NewGlobalRef(var));

#define GET_METHOD_ID(var, clazz, methodName, methodDescriptor) \
        var = env->GetMethodID(clazz, methodName, methodDescriptor); \
        LOG_FATAL_IF(! var, "Unable to find method " methodName);

int register_android_hardware_SensorManager(JNIEnv *env)
{
```

```
    jniRegisterNativeMethods(env, "android/hardware/SystemSensorManager",
            gSystemSensorManagerMethods, NELEM(gSystemSensorManagerMethods));

    jniRegisterNativeMethods(env, "android/hardware/SystemSensorManager$BaseEventQueue",
            gBaseEventQueueMethods, NELEM(gBaseEventQueueMethods));

    FIND_CLASS(gBaseEventQueueClassInfo.clazz,
"android/hardware/SystemSensorManager$BaseEventQueue");

    GET_METHOD_ID(gBaseEventQueueClassInfo.dispatchSensorEvent,
            gBaseEventQueueClassInfo.clazz,
            "dispatchSensorEvent", "(I[FIJ)V");

    return 0;
}
```

18.4.2 处理客户端数据

文件 frameworks\native\libs\gui\SensorManager.cpp 功能是提供了对传感器数据部分的操作，实现了“sensor_data_XXX()”格式的函数。另外，在 Native 层的客户端，文件 SensorManager.cpp 还负责与服务端 SensorService.cpp 之间的通信工作。文件 SensorManager.cpp 的具体实现代码如下所示。

```
// ----------------------------------------------------------------------------
namespace android {
// ----------------------------------------------------------------------------

ANDROID_SINGLETON_STATIC_INSTANCE(SensorManager)

SensorManager::SensorManager()
    : mSensorList(0)
{
    // okay we're not locked here, but it's not needed during construction
    assertStateLocked();
}

SensorManager::~SensorManager()
{
    free(mSensorList);
}

void SensorManager::sensorManagerDied()
{
    Mutex::Autolock _l(mLock);
    mSensorServer.clear();
    free(mSensorList);
    mSensorList = NULL;
    mSensors.clear();
}

status_t SensorManager::assertStateLocked() const {
    if (mSensorServer == NULL) {
        // try for one second
        const String16 name("sensorservice");
        for (int i=0 ; i<4 ; i++) {
            status_t err = getService(name, &mSensorServer);
            if (err == NAME_NOT_FOUND) {
                usleep(250000);
                continue;
            }
            if (err != NO_ERROR) {
                return err;
            }
            break;
        }

        class DeathObserver : public IBinder::DeathRecipient {
            SensorManager& mSensorManger;
            virtual void binderDied(const wp<IBinder>& who) {
```

```
                ALOGW("sensorservice died [%p]", who.unsafe_get());
                mSensorManger.sensorManagerDied();
            }
        public:
            DeathObserver(SensorManager& mgr) : mSensorManger(mgr) { }
        };

        mDeathObserver = new DeathObserver(*const_cast<SensorManager *>(this));
        mSensorServer->asBinder()->linkToDeath(mDeathObserver);

        mSensors = mSensorServer->getSensorList();
        size_t count = mSensors.size();
        mSensorList = (Sensor const**)malloc(count * sizeof(Sensor*));
        for (size_t i=0 ; i<count ; i++) {
            mSensorList[i] = mSensors.array() + i;
        }
    }

    return NO_ERROR;
}


ssize_t SensorManager::getSensorList(Sensor const* const** list) const
{
    Mutex::Autolock _l(mLock);
    status_t err = assertStateLocked();
    if (err < 0) {
        return ssize_t(err);
    }
    *list = mSensorList;
    return mSensors.size();
}

Sensor const* SensorManager::getDefaultSensor(int type)
{
    Mutex::Autolock _l(mLock);
    if (assertStateLocked() == NO_ERROR) {
        // For now we just return the first sensor of that type we find.
        // in the future it will make sense to let the SensorService make
        // that decision.
        for (size_t i=0 ; i<mSensors.size() ; i++) {
            if (mSensorList[i]->getType() == type)
                return mSensorList[i];
        }
    }
    return NULL;
}

sp<SensorEventQueue> SensorManager::createEventQueue()
{
    sp<SensorEventQueue> queue;

    Mutex::Autolock _l(mLock);
    while (assertStateLocked() == NO_ERROR) {
        sp<ISensorEventConnection> connection =
                mSensorServer->createSensorEventConnection();
        if (connection == NULL) {
            // SensorService just died.
            ALOGE("createEventQueue: connection is NULL. SensorService died.");
            continue;
        }
        queue = new SensorEventQueue(connection);
        break;
    }
    return queue;
}

// ----------------------------------------------------------------------------
}; // namespace android
```

18.4.3 处理服务端数据

文件 frameworks\native\services\sensorservice\SensorService.cpp 功能是实现了 Sensor 真正的后台服务，是服务端的数据处理中心。在 Android 的传感器系统中，SensorService 作为一个轻量级的 System Service，运行于 SystemServer 内，即在 system_init<system_init.cpp>中调用了 SensorService::instantiate()。SensorService 主要功能如下所示。

（1）通过 SensorService::instantiate 创建实例对象，并增加到 ServiceManager 中，然后创建并启动线程，并执行 threadLoop。

（2）threadLoop 从 sensor 驱动获取原始数据，然后通过 SensorEventConnection 把事件发送给客户端。

（3）BnSensorServer 的成员函数负责让客户端获取 sensor 列表和创建 SensorEventConnection。

文件 SensorService.cpp 的具体实现代码如下所示。

```
namespace android {

const char* SensorService::WAKE_LOCK_NAME = "SensorService";

SensorService::SensorService()
    : mInitCheck(NO_INIT)
{
}

void SensorService::onFirstRef()
{
    ALOGD("nuSensorService starting...");

    SensorDevice& dev(SensorDevice::getInstance());

    if (dev.initCheck() == NO_ERROR) {
        sensor_t const* list;
        ssize_t count = dev.getSensorList(&list);
        if (count > 0) {
            ssize_t orientationIndex = -1;
            bool hasGyro = false;
            uint32_t virtualSensorsNeeds =
                    (1<<SENSOR_TYPE_GRAVITY) |
                    (1<<SENSOR_TYPE_LINEAR_ACCELERATION) |
                    (1<<SENSOR_TYPE_ROTATION_VECTOR);

            mLastEventSeen.setCapacity(count);
            for (ssize_t i=0 ; i<count ; i++) {
                registerSensor( new HardwareSensor(list[i]) );
                switch (list[i].type) {
                    case SENSOR_TYPE_ORIENTATION:
                        orientationIndex = i;
                        break;
                    case SENSOR_TYPE_GYROSCOPE:
                        hasGyro = true;
                        break;
                    case SENSOR_TYPE_GRAVITY:
                    case SENSOR_TYPE_LINEAR_ACCELERATION:
                    case SENSOR_TYPE_ROTATION_VECTOR:
                        virtualSensorsNeeds &= ~(1<<list[i].type);
                        break;
                }
            }
            const SensorFusion& fusion(SensorFusion::getInstance());

            if (hasGyro) {
                registerVirtualSensor( new RotationVectorSensor() );
                registerVirtualSensor( new GravitySensor(list, count) );
                registerVirtualSensor( new LinearAccelerationSensor(list, count) );

                // these are optional
                registerVirtualSensor( new OrientationSensor() );
                registerVirtualSensor( new CorrectedGyroSensor(list, count) );
```

```
            }
            mUserSensorList = mSensorList;

            if (hasGyro) {
                // virtual debugging sensors are not added to mUserSensorList
                registerVirtualSensor( new GyroDriftSensor() );
            }

            if (hasGyro &&
                    (virtualSensorsNeeds & (1<<SENSOR_TYPE_ROTATION_VECTOR))) {
                if (orientationIndex >= 0) {
                    mUserSensorList.removeItemsAt(orientationIndex);
                }
            }
            for (size_t i=0 ; i<mSensorList.size() ; i++) {
                switch (mSensorList[i].getType()) {
                    case SENSOR_TYPE_GRAVITY:
                    case SENSOR_TYPE_LINEAR_ACCELERATION:
                    case SENSOR_TYPE_ROTATION_VECTOR:
                        if (strstr(mSensorList[i].getVendor().string(), "Google")) {
                            mUserSensorListDebug.add(mSensorList[i]);
                        }
                        break;
                    default:
                        mUserSensorListDebug.add(mSensorList[i]);
                        break;
                }
            }

            run("SensorService", PRIORITY_URGENT_DISPLAY);
            mInitCheck = NO_ERROR;
        }
    }
}

void SensorService::registerSensor(SensorInterface* s)
{
    sensors_event_t event;
    memset(&event, 0, sizeof(event));

    const Sensor sensor(s->getSensor());
    // add to the sensor list (returned to clients)
    mSensorList.add(sensor);
    // add to our handle->SensorInterface mapping
    mSensorMap.add(sensor.getHandle(), s);
    // create an entry in the mLastEventSeen array
    mLastEventSeen.add(sensor.getHandle(), event);
}

void SensorService::registerVirtualSensor(SensorInterface* s)
{
    registerSensor(s);
    mVirtualSensorList.add( s );
}

SensorService::~SensorService()
{
    for (size_t i=0 ; i<mSensorMap.size() ; i++)
        delete mSensorMap.valueAt(i);
}

static const String16 sDump("android.permission.DUMP");

status_t SensorService::dump(int fd, const Vector<String16>& args)
{
    const size_t SIZE = 1024;
    char buffer[SIZE];
    String8 result;
    if (!PermissionCache::checkCallingPermission(sDump)) {
        snprintf(buffer, SIZE, "Permission Denial: "
                "can't dump SurfaceFlinger from pid=%d, uid=%d\n",
```

```
                IPCThreadState::self()->getCallingPid(),
                IPCThreadState::self()->getCallingUid());
        result.append(buffer);
    } else {
        Mutex::Autolock _l(mLock);
        snprintf(buffer, SIZE, "Sensor List:\n");
        result.append(buffer);
        for (size_t i=0 ; i<mSensorList.size() ; i++) {
            const Sensor& s(mSensorList[i]);
            const sensors_event_t& e(mLastEventSeen.valueFor(s.getHandle()));
            snprintf(buffer, SIZE,
                    "%-48s| %-32s | 0x%08x | maxRate=%7.2fHz | "
                    "last=<%18.1f,%18.1f,%18.1f>\n",
                    s.getName().string(),
                    s.getVendor().string(),
                    s.getHandle(),
                    s.getMinDelay() ? (1000000.0f / s.getMinDelay()) : 0.0f,
                    e.data[0], e.data[1], e.data[2]);
            result.append(buffer);
        }
        SensorFusion::getInstance().dump(result, buffer, SIZE);
        SensorDevice::getInstance().dump(result, buffer, SIZE);

        snprintf(buffer, SIZE, "%d active connections\n",
                mActiveConnections.size());
        result.append(buffer);
        snprintf(buffer, SIZE, "Active sensors:\n");
        result.append(buffer);
        for (size_t i=0 ; i<mActiveSensors.size() ; i++) {
            int handle = mActiveSensors.keyAt(i);
            snprintf(buffer, SIZE, "%s (handle=0x%08x, connections=%d)\n",
                    getSensorName(handle).string(),
                    handle,
                    mActiveSensors.valueAt(i)->getNumConnections());
            result.append(buffer);
        }
    }
    write(fd, result.string(), result.size());
    return NO_ERROR;
}

void SensorService::cleanupAutoDisabledSensor(const sp<SensorEventConnection>& connection,
        sensors_event_t const* buffer, const int count) {
    SensorInterface* sensor;
    status_t err = NO_ERROR;
    for (int i=0 ; i<count ; i++) {
        int handle = buffer[i].sensor;
        if (getSensorType(handle) == SENSOR_TYPE_SIGNIFICANT_MOTION) {
            if (connection->hasSensor(handle)) {
                sensor = mSensorMap.valueFor(handle);
                err = sensor ?sensor->resetStateWithoutActuatingHardware(connection.
                get(), handle) : status_t(BAD_VALUE);
                if (err != NO_ERROR) {
                    ALOGE("Sensor Inteface: Resetting state failed with err: %d", err);
                }
                cleanupWithoutDisable(connection, handle);
            }
        }
    }
}

bool SensorService::threadLoop()
{
    ALOGD("nuSensorService thread starting...");

    const size_t numEventMax = 16;
    const size_t minBufferSize = numEventMax + numEventMax * mVirtualSensorList.size();
    sensors_event_t buffer[minBufferSize];
    sensors_event_t scratch[minBufferSize];
    SensorDevice& device(SensorDevice::getInstance());
    const size_t vcount = mVirtualSensorList.size();
```

```
ssize_t count;
bool wakeLockAcquired = false;
const int halVersion = device.getHalDeviceVersion();
do {
    count = device.poll(buffer, numEventMax);
    if (count<0) {
        ALOGE("sensor poll failed (%s)", strerror(-count));
        break;
    }

    // Poll has returned. Hold a wakelock.
    // Todo(): add a flag to the sensors definitions to indicate
    // the sensors which can wake up the AP
    for (int i = 0; i < count; i++) {
        if (getSensorType(buffer[i].sensor) == SENSOR_TYPE_SIGNIFICANT_MOTION) {
            acquire_wake_lock(PARTIAL_WAKE_LOCK, WAKE_LOCK_NAME);
            wakeLockAcquired = true;
            break;
        }
    }

    recordLastValue(buffer, count);

    // handle virtual sensors
    if (count && vcount) {
        sensors_event_t const * const event = buffer;
        const DefaultKeyedVector<int, SensorInterface*> virtualSensors(
                getActiveVirtualSensors());
        const size_t activeVirtualSensorCount = virtualSensors.size();
        if (activeVirtualSensorCount) {
            size_t k = 0;
            SensorFusion& fusion(SensorFusion::getInstance());
            if (fusion.isEnabled()) {
                for (size_t i=0 ; i<size_t(count) ; i++) {
                    fusion.process(event[i]);
                }
            }
            for (size_t i=0 ; i<size_t(count) && k<minBufferSize ; i++) {
                for (size_t j=0 ; j<activeVirtualSensorCount ; j++) {
                    if (count + k >= minBufferSize) {
                        ALOGE("buffer too small to hold all events: "
                                "count=%u, k=%u, size=%u",
                                count, k, minBufferSize);
                        break;
                    }
                    sensors_event_t out;
                    SensorInterface* si = virtualSensors.valueAt(j);
                    if (si->process(&out, event[i])) {
                        buffer[count + k] = out;
                        k++;
                    }
                }
            }
            if (k) {
                // record the last synthesized values
                recordLastValue(&buffer[count], k);
                count += k;
                // sort the buffer by time-stamps
                sortEventBuffer(buffer, count);
            }
        }
    }

    // handle backward compatibility for RotationVector sensor
    if (halVersion < SENSORS_DEVICE_API_VERSION_1_0) {
        for (int i = 0; i < count; i++) {
            if (getSensorType(buffer[i].sensor) == SENSOR_TYPE_ROTATION_VECTOR) {
                // All the 4 components of the quaternion should be available
                // No heading accuracy. Set it to -1
                buffer[i].data[4] = -1;
```

```
                }
            }
        }

        // send our events to clients...
        const SortedVector< wp<SensorEventConnection> > activeConnections(
                getActiveConnections());
        size_t numConnections = activeConnections.size();
        for (size_t i=0 ; i<numConnections ; i++) {
            sp<SensorEventConnection> connection(
                    activeConnections[i].promote());
            if (connection != 0) {
                connection->sendEvents(buffer, count, scratch);
                // Some sensors need to be auto disabled after the trigger
                cleanupAutoDisabledSensor(connection, buffer, count);
            }
        }

        // We have read the data, upper layers should hold the wakelock.
        if (wakeLockAcquired) release_wake_lock(WAKE_LOCK_NAME);

    } while (count >= 0 || Thread::exitPending());

    ALOGW("Exiting SensorService::threadLoop => aborting...");
    abort();
    return false;
}

void SensorService::recordLastValue(
        sensors_event_t const * buffer, size_t count)
{
    Mutex::Autolock _l(mLock);

    // record the last event for each sensor
    int32_t prev = buffer[0].sensor;
    for (size_t i=1 ; i<count ; i++) {
        // record the last event of each sensor type in this buffer
        int32_t curr = buffer[i].sensor;
        if (curr != prev) {
            mLastEventSeen.editValueFor(prev) = buffer[i-1];
            prev = curr;
        }
    }
    mLastEventSeen.editValueFor(prev) = buffer[count-1];
}

void SensorService::sortEventBuffer(sensors_event_t* buffer, size_t count)
{
    struct compar {
        static int cmp(void const* lhs, void const* rhs) {
            sensors_event_t const* l = static_cast<sensors_event_t const*>(lhs);
            sensors_event_t const* r = static_cast<sensors_event_t const*>(rhs);
            return l->timestamp - r->timestamp;
        }
    };
    qsort(buffer, count, sizeof(sensors_event_t), compar::cmp);
}

SortedVector< wp<SensorService::SensorEventConnection> >
SensorService::getActiveConnections() const
{
    Mutex::Autolock _l(mLock);
    return mActiveConnections;
}

DefaultKeyedVector<int, SensorInterface*>
SensorService::getActiveVirtualSensors() const
{
    Mutex::Autolock _l(mLock);
    return mActiveVirtualSensors;
}
```

```
String8 SensorService::getSensorName(int handle) const {
    size_t count = mUserSensorList.size();
    for (size_t i=0 ; i<count ; i++) {
        const Sensor& sensor(mUserSensorList[i]);
        if (sensor.getHandle() == handle) {
            return sensor.getName();
        }
    }
    String8 result("unknown");
    return result;
}

int SensorService::getSensorType(int handle) const {
    size_t count = mUserSensorList.size();
    for (size_t i=0 ; i<count ; i++) {
        const Sensor& sensor(mUserSensorList[i]);
        if (sensor.getHandle() == handle) {
            return sensor.getType();
        }
    }
    return -1;
}


Vector<Sensor> SensorService::getSensorList()
{
    char value[PROPERTY_VALUE_MAX];
    property_get("debug.sensors", value, "0");
    if (atoi(value)) {
        return mUserSensorListDebug;
    }
    return mUserSensorList;
}

sp<ISensorEventConnection> SensorService::createSensorEventConnection()
{
    uid_t uid = IPCThreadState::self()->getCallingUid();
    sp<SensorEventConnection> result(new SensorEventConnection(this, uid));
    return result;
}

void SensorService::cleanupConnection(SensorEventConnection* c)
{
    Mutex::Autolock _l(mLock);
    const wp<SensorEventConnection> connection(c);
    size_t size = mActiveSensors.size();
    ALOGD_IF(DEBUG_CONNECTIONS, "%d active sensors", size);
    for (size_t i=0 ; i<size ; ) {
        int handle = mActiveSensors.keyAt(i);
        if (c->hasSensor(handle)) {
            ALOGD_IF(DEBUG_CONNECTIONS, "%i: disabling handle=0x%08x", i, handle);
            SensorInterface* sensor = mSensorMap.valueFor( handle );
            ALOGE_IF(!sensor, "mSensorMap[handle=0x%08x] is null!", handle);
            if (sensor) {
                sensor->activate(c, false);
            }
        }
        SensorRecord* rec = mActiveSensors.valueAt(i);
        ALOGE_IF(!rec, "mActiveSensors[%d] is null (handle=0x%08x)!", i, handle);
        ALOGD_IF(DEBUG_CONNECTIONS,
                "removing connection %p for sensor[%d].handle=0x%08x",
                c, i, handle);

        if (rec && rec->removeConnection(connection)) {
            ALOGD_IF(DEBUG_CONNECTIONS, "... and it was the last connection");
            mActiveSensors.removeItemsAt(i, 1);
            mActiveVirtualSensors.removeItem(handle);
            delete rec;
            size--;
        } else {
```

```
            i++;
        }
    }
    mActiveConnections.remove(connection);
    BatteryService::cleanup(c->getUid());
}

status_t SensorService::enable(const sp<SensorEventConnection>& connection,
        int handle)
{
    if (mInitCheck != NO_ERROR)
        return mInitCheck;

    Mutex::Autolock _l(mLock);
    SensorInterface* sensor = mSensorMap.valueFor(handle);
    SensorRecord* rec = mActiveSensors.valueFor(handle);
    if (rec == 0) {
        rec = new SensorRecord(connection);
        mActiveSensors.add(handle, rec);
        if (sensor->isVirtual()) {
            mActiveVirtualSensors.add(handle, sensor);
        }
    } else {
        if (rec->addConnection(connection)) {
            // this sensor is already activated, but we are adding a
            // connection that uses it. Immediately send down the last
            // known value of the requested sensor if it's not a
            // "continuous" sensor.
            if (sensor->getSensor().getMinDelay() == 0) {
                sensors_event_t scratch;
                sensors_event_t& event(mLastEventSeen.editValueFor(handle));
                if (event.version == sizeof(sensors_event_t)) {
                    connection->sendEvents(&event, 1);
                }
            }
        }
    }

    if (connection->addSensor(handle)) {
        BatteryService::enableSensor(connection->getUid(), handle);
        // the sensor was added (which means it wasn't already there)
        // so, see if this connection becomes active
        if (mActiveConnections.indexOf(connection) < 0) {
            mActiveConnections.add(connection);
        }
    } else {
        ALOGW("sensor %08x already enabled in connection %p (ignoring)",
            handle, connection.get());
    }


    // we are setup, now enable the sensor.
    status_t err = sensor ? sensor->activate(connection.get(), true) : status_t(BAD_VALUE);

    if (err != NO_ERROR) {
        // enable has failed, reset state in SensorDevice.
        status_t resetErr = sensor ? sensor->resetStateWithoutActuatingHardware
        (connection.get(),
                handle) : status_t(BAD_VALUE);
        // enable has failed, reset our state.
        cleanupWithoutDisable(connection, handle);
    }
    return err;
}

status_t SensorService::disable(const sp<SensorEventConnection>& connection,
        int handle)
{
    if (mInitCheck != NO_ERROR)
        return mInitCheck;
```

```
    status_t err = cleanupWithoutDisable(connection, handle);
    if (err == NO_ERROR) {
        SensorInterface* sensor = mSensorMap.valueFor(handle);
        err = sensor ? sensor->activate(connection.get(), false) : status_t(BAD_VALUE);
    }
    return err;
}

status_t    SensorService::cleanupWithoutDisable(const    sp<SensorEventConnection>&
connection,
        int handle) {
    Mutex::Autolock _l(mLock);
    SensorRecord* rec = mActiveSensors.valueFor(handle);
    if (rec) {
        // see if this connection becomes inactive
        if (connection->removeSensor(handle)) {
            BatteryService::disableSensor(connection->getUid(), handle);
        }
        if (connection->hasAnySensor() == false) {
            mActiveConnections.remove(connection);
        }
        // see if this sensor becomes inactive
        if (rec->removeConnection(connection)) {
            mActiveSensors.removeItem(handle);
            mActiveVirtualSensors.removeItem(handle);
            delete rec;
        }
        return NO_ERROR;
    }
    return BAD_VALUE;
}

status_t SensorService::setEventRate(const sp<SensorEventConnection>& connection,
        int handle, nsecs_t ns)
{
    if (mInitCheck != NO_ERROR)
        return mInitCheck;

    SensorInterface* sensor = mSensorMap.valueFor(handle);
    if (!sensor)
        return BAD_VALUE;

    if (ns < 0)
        return BAD_VALUE;

    nsecs_t minDelayNs = sensor->getSensor().getMinDelayNs();
    if (ns < minDelayNs) {
        ns = minDelayNs;
    }

    if (ns < MINIMUM_EVENTS_PERIOD)
        ns = MINIMUM_EVENTS_PERIOD;

    return sensor->setDelay(connection.get(), handle, ns);
}

// ---------------------------------------------------------------------------

SensorService::SensorRecord::SensorRecord(
        const sp<SensorEventConnection>& connection)
{
    mConnections.add(connection);
}

bool SensorService::SensorRecord::addConnection(
        const sp<SensorEventConnection>& connection)
{
    if (mConnections.indexOf(connection) < 0) {
        mConnections.add(connection);
        return true;
    }
```

```
    return false;
}

bool SensorService::SensorRecord::removeConnection(
        const wp<SensorEventConnection>& connection)
{
    ssize_t index = mConnections.indexOf(connection);
    if (index >= 0) {
        mConnections.removeItemsAt(index, 1);
    }
    return mConnections.size() ? false : true;
}

// ---------------------------------------------------------------------------

SensorService::SensorEventConnection::SensorEventConnection(
        const sp<SensorService>& service, uid_t uid)
    : mService(service), mChannel(new BitTube()), mUid(uid)
{
}

SensorService::SensorEventConnection::~SensorEventConnection()
{
    ALOGD_IF(DEBUG_CONNECTIONS, "~SensorEventConnection(%p)", this);
    mService->cleanupConnection(this);
}

void SensorService::SensorEventConnection::onFirstRef()
{
}

bool SensorService::SensorEventConnection::addSensor(int32_t handle) {
    Mutex::Autolock _l(mConnectionLock);
    if (mSensorInfo.indexOf(handle) < 0) {
        mSensorInfo.add(handle);
        return true;
    }
    return false;
}

bool SensorService::SensorEventConnection::removeSensor(int32_t handle) {
    Mutex::Autolock _l(mConnectionLock);
    if (mSensorInfo.remove(handle) >= 0) {
        return true;
    }
    return false;
}

bool SensorService::SensorEventConnection::hasSensor(int32_t handle) const {
    Mutex::Autolock _l(mConnectionLock);
    return mSensorInfo.indexOf(handle) >= 0;
}

bool SensorService::SensorEventConnection::hasAnySensor() const {
    Mutex::Autolock _l(mConnectionLock);
    return mSensorInfo.size() ? true : false;
}

status_t SensorService::SensorEventConnection::sendEvents(
        sensors_event_t const* buffer, size_t numEvents,
        sensors_event_t* scratch)
{
    // filter out events not for this connection
    size_t count = 0;
    if (scratch) {
        Mutex::Autolock _l(mConnectionLock);
        size_t i=0;
        while (i<numEvents) {
            const int32_t curr = buffer[i].sensor;
            if (mSensorInfo.indexOf(curr) >= 0) {
                do {
```

```
                scratch[count++] = buffer[i++];
            } while ((i<numEvents) && (buffer[i].sensor == curr));
        } else {
            i++;
        }
    }
} else {
    scratch = const_cast<sensors_event_t *>(buffer);
    count = numEvents;
}

// NOTE: ASensorEvent and sensors_event_t are the same type
ssize_t size = SensorEventQueue::write(mChannel,
        reinterpret_cast<ASensorEvent const*>(scratch), count);
if (size == -EAGAIN) {
    // the destination doesn't accept events anymore, it's probably
    // full. For now, we just drop the events on the floor.
    //ALOGW("dropping %d events on the floor", count);
    return size;
}

return size < 0 ? status_t(size) : status_t(NO_ERROR);
}

sp<BitTube> SensorService::SensorEventConnection::getSensorChannel() const
{
    return mChannel;
}

status_t SensorService::SensorEventConnection::enableDisable(
        int handle, bool enabled)
{
    status_t err;
    if (enabled) {
        err = mService->enable(this, handle);
    } else {
        err = mService->disable(this, handle);
    }
    return err;
}

status_t SensorService::SensorEventConnection::setEventRate(
        int handle, nsecs_t ns)
{
    return mService->setEventRate(this, handle, ns);
}

// ---------------------------------------------------------------------------
}; // namespace android
```

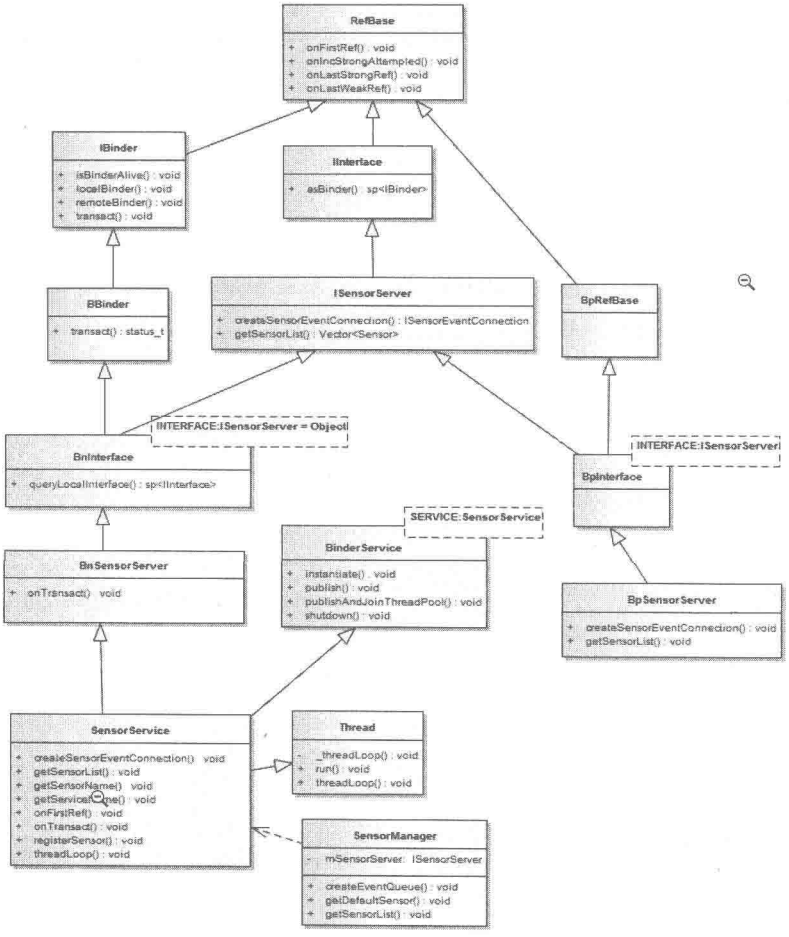
通过上述代码可以了解 SensorService 服务的创建过程和启动过程，整个过程的“C/S”通信架构如图 18-5 所示。

在此需要注意，BpSensorServer 并没有在系统中被用到，即使从 ISensorServer.cpp 中把它删除也不会对 Sensor 的工作有任何影响。这是因为它的工作已经被 SensorManager.cpp 所取代，ServiceManager 会直接获取上面 System_init 文件中添加的 SensorService 对像。

18.4.4　封装 HAL 层的代码

在 Android 系统中，通过文件 frameworks\native\services\sensorservice\SensorDevice.cpp 封装了 HAL 层的代码，主要包含的功能如下所示。

- 获取 sensor 列表（getSensorList）。
- 获取 sensor 事件（poll）。
- Enable 或 Disable sensor（activate）。
- 设置 delay 时间。



▲图 18-5 “C/S”通信架构图

文件 SensorDevice.cpp 的具体实现代码如下所示。

```
namespace android {
// ---------------------------------------------------------------------------

ANDROID_SINGLETON_STATIC_INSTANCE(SensorDevice)

SensorDevice::SensorDevice()
    :  mSensorDevice(0),
       mSensorModule(0)
{
    status_t err = hw_get_module(SENSORS_HARDWARE_MODULE_ID,
            (hw_module_t const**)&mSensorModule);

    ALOGE_IF(err, "couldn't load %s module (%s)",
            SENSORS_HARDWARE_MODULE_ID, strerror(-err));

    if (mSensorModule) {
        err = sensors_open(&mSensorModule->common, &mSensorDevice);

        ALOGE_IF(err, "couldn't open device for module %s (%s)",
                SENSORS_HARDWARE_MODULE_ID, strerror(-err));

        if (mSensorDevice) {
            sensor_t const* list;
            ssize_t count = mSensorModule->get_sensors_list(mSensorModule, &list);
            mActivationCount.setCapacity(count);
            Info model;
```

```
            for (size_t i=0 ; i<size_t(count) ; i++) {
                mActivationCount.add(list[i].handle, model);
                mSensorDevice->activate(mSensorDevice, list[i].handle, 0);
            }
        }
    }
}

void SensorDevice::dump(String8& result, char* buffer, size_t SIZE)
{
    if (!mSensorModule) return;
    sensor_t const* list;
    ssize_t count = mSensorModule->get_sensors_list(mSensorModule, &list);

    snprintf(buffer, SIZE, "%d h/w sensors:\n", int(count));
    result.append(buffer);

    Mutex::Autolock _l(mLock);
    for (size_t i=0 ; i<size_t(count) ; i++) {
        const Info& info = mActivationCount.valueFor(list[i].handle);
        snprintf(buffer, SIZE, "handle=0x%08x, active-count=%d, rates(ms)={ ",
                list[i].handle,
                info.rates.size());
        result.append(buffer);
        for (size_t j=0 ; j<info.rates.size() ; j++) {
            snprintf(buffer, SIZE, "%4.1f%s",
                    info.rates.valueAt(j) / 1e6f,
                    j<info.rates.size()-1 ? ", " : "");
            result.append(buffer);
        }
        snprintf(buffer, SIZE, " }, selected=%4.1f ms\n",  info.delay / 1e6f);
        result.append(buffer);
    }
}

ssize_t SensorDevice::getSensorList(sensor_t const** list) {
    if (!mSensorModule) return NO_INIT;
    ssize_t count = mSensorModule->get_sensors_list(mSensorModule, list);
    return count;
}

status_t SensorDevice::initCheck() const {
    return mSensorDevice && mSensorModule ? NO_ERROR : NO_INIT;
}

ssize_t SensorDevice::poll(sensors_event_t* buffer, size_t count) {
    if (!mSensorDevice) return NO_INIT;
    ssize_t c;
    do {
        c = mSensorDevice->poll(mSensorDevice, buffer, count);
    } while (c == -EINTR);
    return c;
}

status_t SensorDevice::resetStateWithoutActuatingHardware(void *ident, int handle)
{
    if (!mSensorDevice) return NO_INIT;
    Info& info( mActivationCount.editValueFor(handle));
    Mutex::Autolock _l(mLock);
    info.rates.removeItem(ident);
    return NO_ERROR;
}

status_t SensorDevice::activate(void* ident, int handle, int enabled)
{
    if (!mSensorDevice) return NO_INIT;
    status_t err(NO_ERROR);
    bool actuateHardware = false;

    Info& info( mActivationCount.editValueFor(handle) );
```

```
    ALOGD_IF(DEBUG_CONNECTIONS,
            "SensorDevice::activate: ident=%p, handle=0x%08x, enabled=%d, count=%d",
            ident, handle, enabled, info.rates.size());

    if (enabled) {
        Mutex::Autolock _l(mLock);
        ALOGD_IF(DEBUG_CONNECTIONS, "... index=%ld",
                info.rates.indexOfKey(ident));

        if (info.rates.indexOfKey(ident) < 0) {
            info.rates.add(ident, DEFAULT_EVENTS_PERIOD);
            if (info.rates.size() == 1) {
                actuateHardware = true;
            }
        } else {
            // sensor was already activated for this ident
        }
    } else {
        Mutex::Autolock _l(mLock);
        ALOGD_IF(DEBUG_CONNECTIONS, "... index=%ld",
                info.rates.indexOfKey(ident));

        ssize_t idx = info.rates.removeItem(ident);
        if (idx >= 0) {
            if (info.rates.size() == 0) {
                actuateHardware = true;
            }
        } else {
            // sensor wasn't enabled for this ident
        }
    }

    if (actuateHardware) {
        ALOGD_IF(DEBUG_CONNECTIONS, "\t>>> actuating h/w");

        err = mSensorDevice->activate(mSensorDevice, handle, enabled);
        ALOGE_IF(err, "Error %s sensor %d (%s)",
                enabled ? "activating" : "disabling",
                handle, strerror(-err));
    }

    { // scope for the lock
        Mutex::Autolock _l(mLock);
        nsecs_t ns = info.selectDelay();
        mSensorDevice->setDelay(mSensorDevice, handle, ns);
    }

    return err;
}

status_t SensorDevice::setDelay(void* ident, int handle, int64_t ns)
{
    if (!mSensorDevice) return NO_INIT;
    Mutex::Autolock _l(mLock);
    Info& info( mActivationCount.editValueFor(handle) );
    status_t err = info.setDelayForIdent(ident, ns);
    if (err < 0) return err;
    ns = info.selectDelay();
    return mSensorDevice->setDelay(mSensorDevice, handle, ns);
}

int SensorDevice::getHalDeviceVersion() const {
    if (!mSensorDevice) return -1;

    return mSensorDevice->common.version;
}

// ---------------------------------------------------------------------------

status_t SensorDevice::Info::setDelayForIdent(void* ident, int64_t ns)
```

```
{
    ssize_t index = rates.indexOfKey(ident);
    if (index < 0) {
        ALOGE("Info::setDelayForIdent(ident=%p, ns=%lld) failed (%s)",
                ident, ns, strerror(-index));
        return BAD_INDEX;
    }
    rates.editValueAt(index) = ns;
    return NO_ERROR;
}

nsecs_t SensorDevice::Info::selectDelay()
{
    nsecs_t ns = rates.valueAt(0);
    for (size_t i=1 ; i<rates.size() ; i++) {
        nsecs_t cur = rates.valueAt(i);
        if (cur < ns) {
            ns = cur;
        }
    }
    delay = ns;
    return ns;
}

// ---------------------------------------------------------------------------
}; // namespace android
```

这样 SensorSevice 会把任务交给 SensorDevice，而 SensorDevice 会调用标准的抽象层接口。由此可见，Sensor 架构的抽象层接口是最标准的一种，它很好地实现了抽象层与本地框架的分离。

18.4.5　处理消息队列

在 Android 的传感器系统中，文件 frameworks\native\libs\gui\SensorEventQueue.cpp 实现了处理消息队列的功能。此文件能够在创建其实例时传入 SensorEventConnection 的实例，SensorEvent Connection 继承于 ISensorEventConnection。SensorEventConnection 其实是客户端调用 SensorService 的 createSensorEventConnection()方法创建的，是客户端与服务端沟通的桥梁，通过这个桥梁，可以完成如下所示的任务。

- 获取管道的句柄。
- 往管道读写数据。
- 通知服务端对 Sensor 使可用。

文件 frameworks\native\libs\gui\SensorEventQueue.cpp 的具体实现代码如下所示。

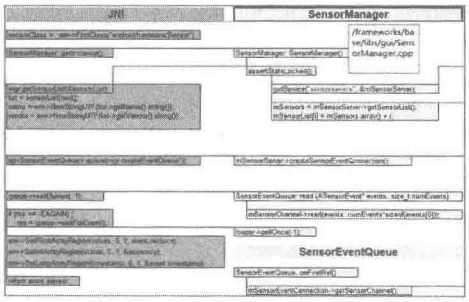
```
// ---------------------------------------------------------------------------
namespace android {
// ---------------------------------------------------------------------------

SensorEventQueue::SensorEventQueue(const sp<ISensorEventConnection>& connection)
    : mSensorEventConnection(connection)
{
}

SensorEventQueue::~SensorEventQueue()
{
}

void SensorEventQueue::onFirstRef()
{
    mSensorChannel = mSensorEventConnection->getSensorChannel();
}

int SensorEventQueue::getFd() const
{
    return mSensorChannel->getFd();
}
```

```
ssize_t SensorEventQueue::write(const sp<BitTube>& tube,
        ASensorEvent const* events, size_t numEvents) {
    return BitTube::sendObjects(tube, events, numEvents);
}

ssize_t SensorEventQueue::read(ASensorEvent* events, size_t numEvents)
{
    return BitTube::recvObjects(mSensorChannel, events, numEvents);
}

sp<Looper> SensorEventQueue::getLooper() const
{
    Mutex::Autolock _l(mLock);
    if (mLooper == 0) {
        mLooper = new Looper(true);
        mLooper->addFd(getFd(), getFd(), ALOOPER_EVENT_INPUT, NULL, NULL);
    }
    return mLooper;
}

status_t SensorEventQueue::waitForEvent() const
{
    const int fd = getFd();
    sp<Looper> looper(getLooper());

    int events;
    int32_t result;
    do {
        result = looper->pollOnce(-1, NULL, &events, NULL);
        if (result == ALOOPER_POLL_ERROR) {
            ALOGE("SensorEventQueue::waitForEvent error (errno=%d)", errno);
            result = -EPIPE; // unknown error, so we make up one
            break;
        }
        if (events & ALOOPER_EVENT_HANGUP) {
            // the other-side has died
            ALOGE("SensorEventQueue::waitForEvent error HANGUP");
            result = -EPIPE; // unknown error, so we make up one
            break;
        }
    } while (result != fd);

    return  (result == fd) ? status_t(NO_ERROR) : result;
}

status_t SensorEventQueue::wake() const
{
    sp<Looper> looper(getLooper());
    looper->wake();
    return NO_ERROR;
}

status_t SensorEventQueue::enableSensor(Sensor const* sensor) const {
    return mSensorEventConnection->enableDisable(sensor->getHandle(), true);
}

status_t SensorEventQueue::disableSensor(Sensor const* sensor) const {
    return mSensorEventConnection->enableDisable(sensor->getHandle(), false);
}

status_t SensorEventQueue::enableSensor(int32_t handle, int32_t us) const {
    status_t err = mSensorEventConnection->enableDisable(handle, true);
    if (err == NO_ERROR) {
        mSensorEventConnection->setEventRate(handle, us2ns(us));
    }
    return err;
}

status_t SensorEventQueue::disableSensor(int32_t handle) const {
    return mSensorEventConnection->enableDisable(handle, false);
}
```

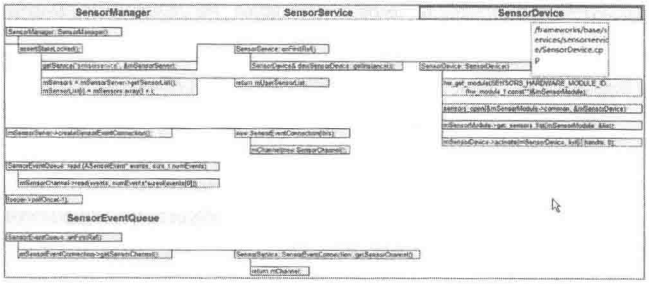
```
status_t SensorEventQueue::setEventRate(Sensor const* sensor, nsecs_t ns) const {
    return mSensorEventConnection->setEventRate(sensor->getHandle(), ns);
}

// ----------------------------------------------------------------------------
}; // namespace android
```

由此可见，SensorManager 负责控制流，通过 C/S 的 Binder 机制与 SensorService 实现通信。具体过程如图 18-6 所示。

而 SensorEventQueue 负责数据流，功能是通过管道机制来读写底层的数据。具体过程如图 18-7 所示。



▲图 18-6　SensorManager 控制流的处理流程



▲图 18-7　SensorEventQueue 数据流的处理流程

18.5 HAL 层详解

在 Android 系统中，在 HAL 层中提供了 Android 独立于具体硬件的抽象接口。其中 HAL 层的头文件路径是。

```
hardware/libhardware/include/hardware/sensors.h
```

而具体实现文件需要开发者个人编写，具体可以参考如下所示的文件。

```
hardware\invensense\libsensors_iio\sensors_mpl.cpp
```

文件 sensors.h 的主要实现代码如下所示。

```
typedef struct {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} sensors_vec_t;

typedef struct {
  union {
    float uncalib[3];
    struct {
      float x_uncalib;
      float y_uncalib;
      float z_uncalib;
    };
```

```
    };
    union {
      float bias[3];
      struct {
        float x_bias;
        float y_bias;
        float z_bias;
      };
    };
} uncalibrated_event_t;

/**
 * Union of the various types of sensor data
 * that can be returned.
 */
typedef struct sensors_event_t {
    /* must be sizeof(struct sensors_event_t) */
    int32_t version;

    /* sensor identifier */
    int32_t sensor;

    /* sensor type */
    int32_t type;

    /* reserved */
    int32_t reserved0;

    /* time is in nanosecond */
    int64_t timestamp;

    union {
        float         data[16];

        /* acceleration values are in meter per second per second (m/s^2) */
        sensors_vec_t   acceleration;

        /* magnetic vector values are in micro-Tesla (uT) */
        sensors_vec_t   magnetic;

        /* orientation values are in degrees */
        sensors_vec_t   orientation;

        /* gyroscope values are in rad/s */
        sensors_vec_t   gyro;

        /* temperature is in degrees centigrade (Celsius) */
        float         temperature;

        /* distance in centimeters */
        float         distance;

        /* light in SI lux units */
        float         light;

        /* pressure in hectopascal (hPa) */
        float         pressure;

        /* relative humidity in percent */
        float         relative_humidity;

        /* step-counter */
        uint64_t       step_counter;

        /* uncalibrated gyroscope values are in rad/s */
        uncalibrated_event_t uncalibrated_gyro;

        /* uncalibrated magnetometer values are in micro-Teslas */
        uncalibrated_event_t uncalibrated_magnetic;
    };
    uint32_t       reserved1[4];
```

```
} sensors_event_t;

struct sensor_t;
struct sensors_module_t {
    struct hw_module_t common;
    int (*get_sensors_list)(struct sensors_module_t* module,
            struct sensor_t const** list);
};

struct sensor_t {

    /* Name of this sensor.
     * All sensors of the same "type" must have a different "name".
     */
    const char*     name;

    /* vendor of the hardware part */
    const char*     vendor;
    int             version;
    int             handle;

    /* this sensor's type. */
    int             type;

    /* maximum range of this sensor's value in SI units */
    float           maxRange;

    /* smallest difference between two values reported by this sensor */
    float           resolution;

    /* 传感器的粗略值，单位毫安*/
    float           power;
    int32_t         minDelay;

    /*保留字段，必须为零*/
    void*           reserved[8];
};

struct sensors_poll_device_t {
    struct hw_device_t common;
    int (*activate)(struct sensors_poll_device_t *dev,
            int handle, int enabled);
    int (*setDelay)(struct sensors_poll_device_t *dev,
            int handle, int64_t ns);
    int (*poll)(struct sensors_poll_device_t *dev,
            sensors_event_t* data, int count);
};
typedef struct sensors_poll_device_1 {
    union {
        struct sensors_poll_device_t v0;

        struct {
            struct hw_device_t common;
            int (*activate)(struct sensors_poll_device_t *dev,
                    int handle, int enabled);

            int (*setDelay)(struct sensors_poll_device_t *dev,
                    int handle, int64_t period_ns);

            int (*poll)(struct sensors_poll_device_t *dev,
                    sensors_event_t* data, int count);
        };
    };
    int (*batch)(struct sensors_poll_device_1* dev,
            int handle, int flags, int64_t period_ns, int64_t timeout);

    void (*reserved_procs[8])(void);

} sensors_poll_device_1_t;
```

```
/**用于方便打开和关闭装置的 API */

static inline int sensors_open(const struct hw_module_t* module,
        struct sensors_poll_device_t** device) {
    return module->methods->open(module,
            SENSORS_HARDWARE_POLL, (struct hw_device_t**)device);
}

static inline int sensors_close(struct sensors_poll_device_t* device) {
    return device->common.close(&device->common);
}

static inline int sensors_open_1(const struct hw_module_t* module,
        sensors_poll_device_1_t** device) {
    return module->methods->open(module,
            SENSORS_HARDWARE_POLL, (struct hw_device_t**)device);
}

static inline int sensors_close_1(sensors_poll_device_1_t* device) {
    return device->common.close(&device->common);
}

__END_DECLS

#endif  // ANDROID_SENSORS_INTERFACE_H
```

而具体的实现文件是 Linux Kernel 层，也就是具体的硬件设备驱动程序，例如，可以将其命名为“sensors.c”，然后编写如下定义 struct sensors_poll_device_t 的代码。

```
struct sensors_poll_device_t {
    struct hw_device_t common;

    // Activate/deactivate one sensor.
    int (*activate)(struct sensors_poll_device_t *dev,
            int handle, int enabled);

    //对于一个给定的传感器,设置在事件之间的延迟,单位微秒
    int (*setDelay)(struct sensors_poll_device_t *dev,
            int handle, int64_t ns);

    //返回传感器数据的数组
    int (*poll)(struct sensors_poll_device_t *dev,
            sensors_event_t* data, int count);
};
```

也可以编写如下定义 struct sensors_module_t 的代码。

```
struct sensors_module_t {
    struct hw_module_t common;

    /**
     *枚举所有可用的传感器
     * @return number of sensors in the list
     */
    int (*get_sensors_list)(struct sensors_module_t* module,
            struct sensor_t const** list);
};
```

也可以编写如下定义 struct sensor_t 的代码。

```
struct sensor_t {
    /* 传感器名字 */
    const char*     name;
    /*硬件部分的供应商*/
    const char*     vendor;
    /* 版本硬件驱动*/
    int             version;
    /*处理标识的此传感器。这个句柄是用来激活和关闭该传感器。在这个版本的 API 的手柄值必须是 8 位的
     */
    int             handle;
    /* 这种传感器类型 */
```

```
    int             type;
    /*该传感器的 maximaum 范围值*/
    float           maxRange;
    /* smallest difference between two values reported by this sensor */
    float           resolution;
    /* rough estimate of this sensor's power consumption in mA */
    float           power;
    /* minimum delay allowed between events in microseconds. A value of zero
     * means that this sensor doesn't report events at a constant rate, but
     * rather only when a new data is available */
    int32_t         minDelay;
    /* reserved fields, must be zero */
    void*           reserved[8];
};
```

也可以编写如下定义 struct sensors_event_t 的代码。

```
typedef struct {
    union {
        float v[3];
        struct {
            float x;
            float y;
            float z;
        };
        struct {
            float azimuth;
            float pitch;
            float roll;
        };
    };
    int8_t status;
    uint8_t reserved[3];
} sensors_vec_t;

/**
 *各种类型的传感器数据可以被返回的联合类型
 */
typedef struct sensors_event_t {
    /* 必须是 struct sensors_event_t 类型 */
    int32_t version;

    /* 传感器标识 */
    int32_t sensor;

    /* 传感器类型 */
    int32_t type;

    /* 保留的 */
    int32_t reserved0;

    /* 纳秒时间 */
    int64_t timestamp;

    union {
        float           data[16];

        /* 加速度值，单位是(m/s^2) */
        sensors_vec_t   acceleration;

        /*磁矢量值，微特斯拉 (uT) */
        sensors_vec_t   magnetic;

        /* 某一度的定向值 */
        sensors_vec_t   orientation;

        /*陀螺仪值，单位为 rad/ s */
        sensors_vec_t   gyro;

        /* 温度，单位是摄氏度 */
        float           temperature;
```

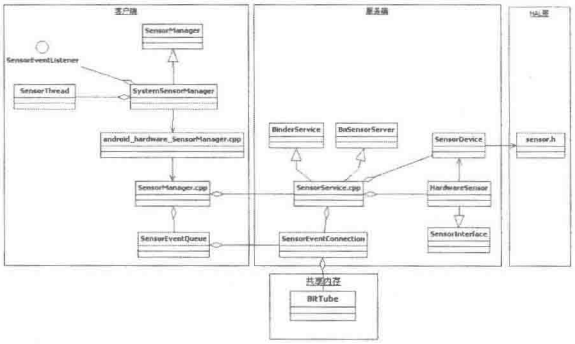
```
        /* 距离，单位厘米 */
        float           distance;

        /* 光亮度 */
        float           light;

        /* 压力，单位是 hPa*/
        float           pressure;

        /* 相对湿度*/
        float           relative_humidity;
    };
    uint32_t        reserved1[4];
} sensors_event_t;
```

也可以编写如下定义 struct sensors_module_t 的代码。

```
static const struct sensor_t sSensorList[] = {
        { "MMA8452Q 3-axis Accelerometer",
                "Freescale Semiconductor",
                1, SENSORS_HANDLE_BASE+ID_A,
                SENSOR_TYPE_ACCELEROMETER, 4.0f*9.81f, (4.0f*9.81f)/256.0f, 0.2f, 0, { } },
        { "AK8975 3-axis Magnetic field sensor",
                "Asahi Kasei",
                1, SENSORS_HANDLE_BASE+ID_M,
                SENSOR_TYPE_MAGNETIC_FIELD, 2000.0f, 1.0f/16.0f, 6.8f, 0, { } },
        { "AK8975 Orientation sensor",
                "Asahi Kasei",
                1, SENSORS_HANDLE_BASE+ID_O,
                SENSOR_TYPE_ORIENTATION, 360.0f, 1.0f, 7.0f, 0, { } },

    { "ST 3-axis Gyroscope sensor",
          "STMicroelectronics",
          1, SENSORS_HANDLE_BASE+ID_GY,
          SENSOR_TYPE_GYROSCOPE, RANGE_GYRO, CONVERT_GYRO, 6.1f, 1190, { } },

    { "AL3006Proximity sensor",
        "Dyna Image Corporation",
        1, SENSORS_HANDLE_BASE+ID_P,
        SENSOR_TYPE_PROXIMITY,
        PROXIMITY_THRESHOLD_CM, PROXIMITY_THRESHOLD_CM,
        0.5f, 0, { } },

        { "AL3006 light sensor",
                "Dyna Image Corporation",
                1, SENSORS_HANDLE_BASE+ID_L,
                SENSOR_TYPE_LIGHT, 10240.0f, 1.0f, 0.5f, 0, { } },

};

static int open_sensors(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device);

static int sensors__get_sensors_list(struct sensors_module_t* module,
        struct sensor_t const** list)
{
    *list = sSensorList;
    return ARRAY_SIZE(sSensorList);
}

static struct hw_module_methods_t sensors_module_methods = {
    .open = open_sensors
};

const struct sensors_module_t HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .version_major = 1,
        .version_minor = 0,
        .id = SENSORS_HARDWARE_MODULE_ID,
        .name = "MMA8451Q & AK8973A & gyro Sensors Module",
        .author = "The Android Project",
```

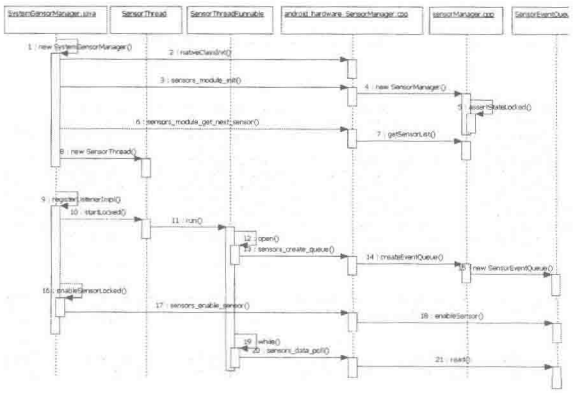
```
        .methods = &sensors_module_methods,
    },
    .get_sensors_list = sensors__get_sensors_list
};

static int open_sensors(const struct hw_module_t* module, const char* name,
        struct hw_device_t** device)
{
    return init_nusensors(module, device); //待后面讲解
}
```

到此为止，整个 Android 系统中传感器模块的源代码分析完毕。由此可见，整个传感器系统的总体调用关系如图 18-8 所示。



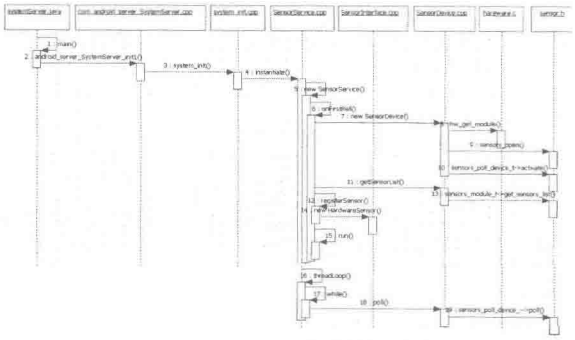
▲图 18-8　传感器系统的总体调用关系

客户端读取数据时的调用时序如图 18-9 所示。



▲图 18-9　客户端读取数据时的调用时序图

服务器端的调用时序如图 18-10 所示。



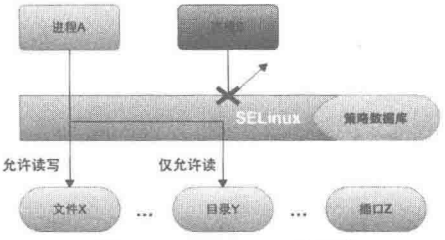
▲图 18-10　服务器端的调用时序图

第 19 章　分析 SEAndroid 系统

SEAndroid 是 Security-Enhanced Android 的缩写，是指将一直在 Linux 操作系统中使用的 MAC 强制存取控管套件 SELinux 移植到 Android 平台上，以达到强化 Android 操作系统对 App 的存取控管的目的，这样可以建立一个类似于沙箱的执行隔离效果，以确保每一个 App 之间的独立运作，并阻止恶意 App 对系统或其他应用程序的攻击。在本章的内容中，将详细讲解 Android 5.0 系统中 SEAndroid 模块的基本知识，为读者步入本书后面知识的学习打下基础。

19.1 SEAndroid 概述

从 Android 4.0 开始，谷歌便不遗余力地改善 Android 系统的流畅性。特别是在最新版本 5.0 中，使用 ART 替换了 Dalvik。从 Android 4.3 版本开始，Android 便引入了基于 SELinux 的安全机制：SEAndroid，来加强系统安全性。SEAndroid 的核心理念是，即使 root 权限被篡夺，只求阻止应用的恶意行为。其策略如图 19-1 所示。



▲图 19-1　SEAndroid 的策略

在引进 SEAndroid 安全机制之前，Android 系统的安全机制分为应用程序和内核两个级别。应用程序级别的安全机制就是我们通常说的 Permission 机制。一个应用如果需要访问一些系统敏感或者特权资源，那么就必须要在 AndroidManifest.xml 配置文件中进行申请，并且在安装的时候由用户决定是否赋予相应的权限。应用安装过后，一般是通过系统服务来间接使用系统敏感或者特权资源的。这样系统服务在代表应用使用这些资源之前，就会先检查应用之前是否已经申请过相应的权限。如果已经申请过，那么就直接放行，否则就拒绝执行。

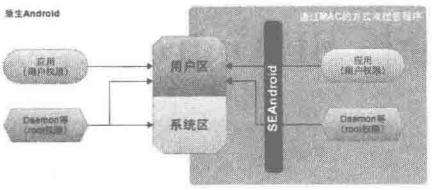
在 Android 系统中，无论是应用级别的 Permission 机制，还是内核级别的 Linux UID/GID 机制，都会受到 DAC 问题的困扰，这时迫切需要一种更为强有力的安全机制来保证系统的安全。在访问控制模型中，与 DAC 机制相对的是 MAC 机制。MAC 的全称是 Mandatory Access Control，译为强制访问控制。在 MAC 机制中，用户、进程或者文件的权限是由管理策略决定的，而不是由它们自主决定的。例如，可以设定一个如下所示的管理策略：

> 不允许用户 A 将其创建的文件 F 授予用户 B 访问

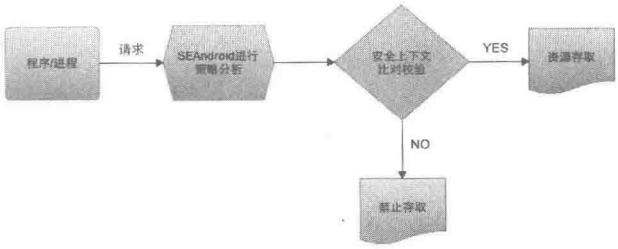
这样无论用户 A 如何修改文件 F 的权限位，用户 B 都是无法访问文件 F 的。这种安全访问模型可以有力地保护系统的安全，而 SEAndroid 就是一种 MAC 机制。在 SEAndroid 中，每一个进程和文件都会关联一个安全上下文。这个安全上下文由用户、角色、类型、安全级别 4 个部分组成，每一部分通过一个冒号来分隔。例如，u:r:t:s0 描述的就是一个 SEAndroid 安全上下文。当每一个进程和文件都关联上一个安全上下文之后，系统管理员就可以基于这些安全上下文制定一个安全访问策略，用来规定什么样的进程可以访问什么样的文件。

SEAndroid 等于“SElinux + Android”，通过 MAC 的方式管控应用程序，从而提升原生 Android 系统的安全性，如图 19-2 所示。

究竟 SEAndroid 如何管控程序呢？程序向 SEAndroid（确切的说是 SELinux）发送请求，SEAndroid 根据策略数据库进行策略分析，比对安全上下文，控制应用程序的资源存取，如图 19-3 所示。



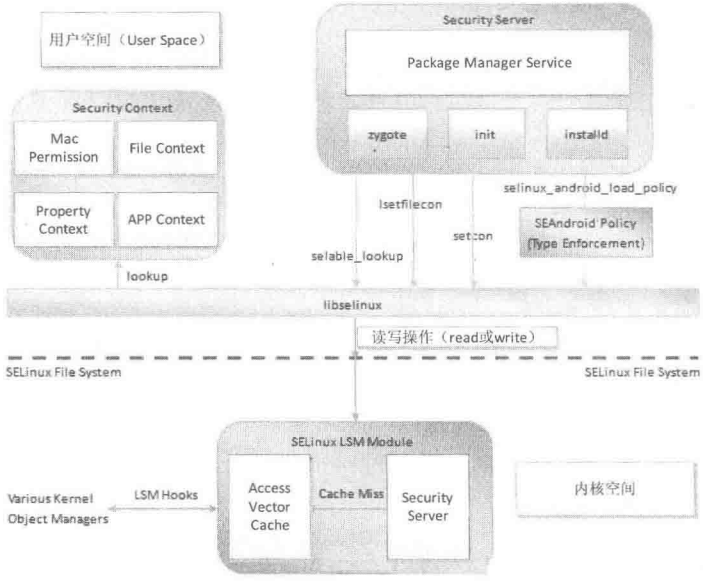
▲图 19-2　SEAndroid 的安全策略



▲图 19-3　SEAndroid 管控程序

SEAndroid 所要保护的对象是系统中的资源，这些资源分布在各个子系统中，例如分布文件子系统中的资源、进程、socket 和 ipc 等。对于 Android 系统来说，因为使用了与传统 Linux 系统不一样的用户空间运行时，所以它在用户空间有一些特有的资源是需要特别保护的，例如系统属性的设置。

SEAndroid 机制的整体框架如图 19-4 所示。



▲图 19-4　SEAndroid 机制的整体框架

SEAndroid 机制以 SELinux 文件系统接口为边界，分为内核空间和用户空间两部分。在内核空间中，主要涉及一个称为 SELinux LSM 的模块。而在用户空间中，涉及了 Security Context、Security Server 和 SEAndroid Policy 等模块。这些内核空间模块和用户空间模块之间的交互如下所示。

- 内核空间中的 SELinux LSM 模块：负责内核资源的安全访问控制。
- 用户空间中的 SEAndroid Policy：描述了资源安全访问策略。在启动系统时，用户空间的 Security Server 通过 SELinux 文件系统接口，将这些安全访问策略加载到内核空间的 SELinux LSM 模块中。
- 用户空间中的 Security Context：描述了资源安全上下文，SEAndroid 的安全访问策略就是在资源的安全上下文基础上实现的。
- 用户空间中的 Security Server：不但需要用户空间的 Security Context 去检索对象的安全上下文，而且也需要到内核空间去操作对象的安全上下文。
- 用户空间中的 SELinux 库：封装了对 SELinux 文件系统接口的读写操作。当用户空间的 Security Server 访问内核空间的 SELinux LSM 模块时，都是间接地通过 SELinux 进行的。这样做

的好处是，将对 SELinux 文件系统接口的读写操作封装成更有意义的函数调用。当用户空间中的 Security Server 到用户空间的 Security Context 去检索对象的安全上下文时，也需要通过 SELinux 库来实现。

在接下来的内容中，将从内核空间和用户空间这两个角度来分析 SEAndroid 安全机制框架。

19.1.1 内核空间

在内核空间中有一个 SELinux LSM 模块，此模块包含有一个访问向量缓冲（Access Vector Cache）和一个安全服务（Security Server）。Security Server 负责安全访问控制逻辑，即由它来决定一个主体访问一个客体是否是合法的。这里说的主体一般就是指进程，而客体就是主体要访问的资源，例如文件。

与 SELinux Security Server 相关的内核子模块是 LSM，其全称是 Linux Security Model。LSM 是为了 SELinux 而设计的，是一个通用的安全模块，不但 SELinux 可以使用，而且其他的模块也可以使用。

究竟 SELinux、LSM 和内核中的子系统是如何交互的呢？具体过程如下所示。

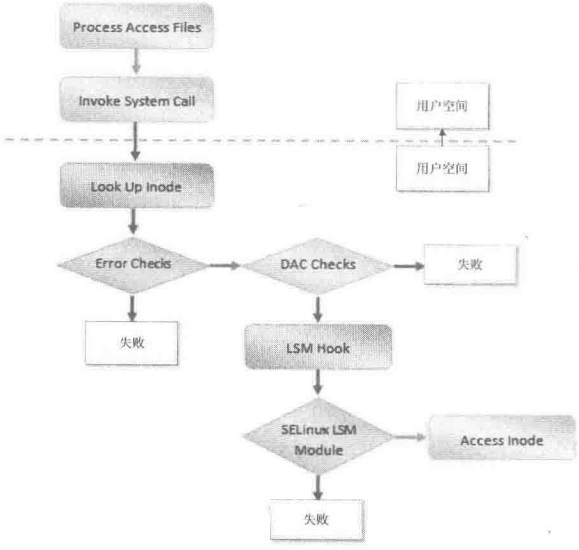
（1）首先，SELinux 会在 LSM 中注册相应的回调函数。

（2）LSM 在相应的内核对象子系统中会加入一些 Hook 代码。例如，在调用系统接口函数 read 读取一个文件时，会进入到内核的文件子系统中。在文件子系统中，读取文件函数 vfs_read 会调用 LSM 加入的 Hook 代码。这些 Hook 代码就会调用之前 SELinux 注册进来的回调函数，以便后者可以进行安全检查。

（3）当 SELinux 进行安全检查的时候，首先会检查自己的 Access Vector Cache 是否已经有结果。如果有，则直接将结果返回给相应的内核子系统就可以了；如果没有，则需要到 Security Server 中去进行检查。检查出来的结果在返回给相应的内核子系统的同时，也会保存在自己的 Access Vector Cache 中，以便下次可以快速地得到检查结果。上面描述的安全访问控制流程可以通过图 19-5 来总结。

从图 19-5 可以看到，内核中的资源在访问的过程中，一般需要经过如下 3 次检查才可以通过。

- 一般性错误检查，例如访问的对象是否存在、访问参数是否正确等。
- DAC 检查，即基于 Linux UID/GID 的安全检查。
- SELinux 检查，即基于安全上下文和安全策略的安全检查。



▲图 19-5 SELinux 安全访问控制流程

19.1.2　用户空间

在用户空间中，SEAndroid 包含有 3 个主要的模块，具体说明如下所示。

1. 安全上下文（Security Context）

SEAndroid 安全策略又是建立在对象安全上下文的基础上的，是一种基于安全策略的 MAC 安全机制。这里的对象安全分为如下两种类型。

- 主体（Subject）：通常指进程。
- 客体（Object）：指进程所要访问的资源，例如文件和系统属性等。

安全上下文实际上就是一个附加在对象上的标签（Tag），这个标签实际上是一个由如下 4 部分内容组成的字符串。

- SELinux 用户。
- SELinux 角色。
- 类型。
- 安全级别。

上述每一个部分都通过一个冒号来分隔，具体格式为。

```
user:role:type:sensitivity
```

例如，当在开启 SEAndroid 安全机制的设备上执行如下带-Z 选项的 ls 命令时，可以看到一个文件的安全上下文。

```
$ ls -Z /init.rc
-rwxr-x--- root     root     u:object_r:rootfs:s0 init.rc
```

上面的命令列出文件/init.rc 的安全上下文为“u:object_r:rootfs:s0”，这表明文件/init.rc 的 SELinux 用户、SELinux 角色、类型和安全级别分别为 u、object_r、rootfs 和 s0。

例如，在开启了 SEAndroid 安全机制的设备上执行如下带-Z 选项的 ps 命令时，可以看到一个进程的安全上下文。

```
$ ps -Z
LABEL                          USER     PID   PPID  NAME
u:r:init:s0                     root      1     0    /init
......
```

上面的命令列出进程 init 的安全上下文为“u:r:init:s0”，这表明进程 init 的 SELinux 用户、SELinux 角色、类型和安全级别分别为 u、r、init 和 s0。

因为在安全上下文中，只有类型（Type）才是最重要的，而 SELinux 用户、SELinux 角色和安全级别可以忽略不计，所以，SEAndroid 安全机制又被称为是基于 TE（Tyoe Enforcement）策略的安全机制。

对于进程来说，SELinux 用户和 SELinux 角色只是用来限制进程可以标注的类型。而对于文件来说，SELinux 用户和 SELinux 角色可以忽略不计。为了完整地描述一个文件的安全上下文，通常将这个文件进行如下设置。

- 将其 SELinux 角色固定为 object_r。
- 将其 SELinux 用户设置为创建它的进程的 SELinux 用户。

因为在 SEAndroid 机制中只定义了一个 SELinux 用户 u，所以，通过 ps -Z 和 ls -Z 命令看到的所有的进程和文件的安全上下文中的 SELinux 用户都是 u。并且，因为 SEAndroid 只定义了一个 SELinux 角色 r，所以，通过 ps -Z 命令看到的所有进程的安全上下文中的 SELinux 角色都为 r。

在文件 external/sepolicy/users 和 external/sepolicy/roles 中可以看到 SEAndroid 所定义的 SELinux 用户和 SELinux 角色。其中文件 external/sepolicy/users 中的内容如下所示。

```
user u roles { r } level s0 range s0 - mls_systemhigh;
```

上述代码声明了一个 SELinux 用户 u，它可用的 SELinux 角色为 r，它的默认安全级别为 s0，可用的安全级别范围为 s0~mls_systemhigh，其中，mls_systemhigh 为系统定义的最高安全级别。

文件 external/sepolicy/roles 的代码如下所示。

```
role r;
role r types domain;
```

在上述代码中，第一个语句声明了一个 SELinux 角色 r，第二个语句允许 SELinux 角色 r 与类型 domain 关联。

在 SEAndroid 机制中，通常将用来标注文件的安全上下文中的类型称为 file_type，而用来标注进程的安全上下文的类型称为 domain，并且每一个用来描述文件安全上下文的类型都将 file_type 设置为其属性，每一个用来进程安全上下文的类型都将 domain 设置为其属性。

将一个类型设置为另一个类型的属性可以通过 type 语句实现。例如，前面提到的用来描述进程 init 的安全策略的文件 external/sepolicy/init.te，就使用以下的 type 语句来将类型 domain 设置为类型 init 的属性。

```
type init domain;
```

这样就可以表明 init 描述的类型是用来描述进程的安全上下文的。

同样，如果查看另外一个文件 external/sepolicy/file.te，可以看到 App 数据文件的类型声明。

```
type app_data_file, file_type, data_file_type;
```

上述代码表明类型 app_data_file 具有 file_type 属性，即它是用来描述文件的安全上下文的。

到此为止，了解了 SEAndroid 安全机制的安全上下文后，就可以了解 Android 对象的安全上下文的定义过程。在此只讨论 4 种类型的对象的安全上下文，分别是 App 进程、App 数据文件、系统文件和系统属性。这 4 种类型对象的安全上下文通过 4 个文件来描述：mac_permissions.xml、seapp_contexts、file_contexts 和 property_contexts，它们均位于 external/sepolicy 目录中。其中文件 external/sepolicy/mac_permissions.xml 的内容如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<policy>

    <!-- Platform dev key in AOSP -->
    <signer signature="@PLATFORM" >
      <seinfo value="platform" />
    </signer>

    <!-- Media dev key in AOSP -->
    <signer signature="@MEDIA" >
      <seinfo value="media" />
    </signer>

    <!-- shared dev key in AOSP -->
    <signer signature="@SHARED" >
      <seinfo value="shared" />
    </signer>

    <!-- release dev key in AOSP -->
    <signer signature="@RELEASE" >
      <seinfo value="release" />
    </signer>

    <!-- All other keys -->
    <default>
      <seinfo value="default" />
    </default>

</policy>
```

文件 mac_permissions.xml 给不同签名的 App 分配不同的 seinfo 字符串，例如，在 AOSP 源代码环境下编译并且使用平台签名的 App 获得的 seinfo 为“platform”，使用第三方签名安装的 App 获得的 seinfo 签名为“default”。seinfo 描述的并不是安全上下文中的 Type，而是在另外一个文件 external/sepolicy/seapp_contexts 中查找对应的 Type 的。文件 external/sepolicy/seapp_contexts 的内容如下所示。

```
# Input selectors:
#   isSystemServer (boolean)
#   user (string)
#   seinfo (string)
#   name (string)
#   sebool (string)
# isSystemServer=true can only be used once.
# An unspecified isSystemServer defaults to false.
# An unspecified string selector will match any value.
# A user string selector that ends in * will perform a prefix match.
# user=_app will match any regular app UID.
# user=_isolated will match any isolated service UID.
# All specified input selectors in an entry must match (i.e. logical AND).
# Matching is case-insensitive.
# Precedence rules:
#     (1) isSystemServer=true before isSystemServer=false.
#     (2) Specified user= string before unspecified user= string.
#     (3) Fixed user= string before user= prefix (i.e. ending in *).
#     (4) Longer user= prefix before shorter user= prefix.
#     (5) Specified seinfo= string before unspecified seinfo= string.
#     (6) Specified name= string before unspecified name= string.
#     (7) Specified sebool= string before unspecified sebool= string.
#
# Outputs:
#   domain (string)
#   type (string)
#   levelFrom (string; one of none, all, app, or user)
#   level (string)
# Only entries that specify domain= will be used for app process labeling.
# Only entries that specify type= will be used for app directory labeling.
# levelFrom=user is only supported for _app or _isolated UIDs.
# levelFrom=app or levelFrom=all is only supported for _app UIDs.
# level may be used to specify a fixed level for any UID.
#
isSystemServer=true domain=system
user=system domain=system_app type=system_data_file
user=bluetooth domain=bluetooth type=bluetooth_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=_app domain=untrusted_app type=app_data_file levelFrom=none
user=_app seinfo=platform domain=platform_app type=platform_app_data_file
user=_app seinfo=shared domain=shared_app type=platform_app_data_file
user=_app seinfo=media domain=media_app type=platform_app_data_file
user=_app seinfo=release domain=release_app type=platform_app_data_file
user=_isolated domain=isolated_app
```

对于使用平台签名的 App 来说，它的 seinfo 为“platform”。用户空间的 Security Server 在为它查找对应的 Type 时，使用的 user 输入为“_app”。这样在 seapp_contexts 文件中，与它匹配的一行程序为：

```
user=_app seinfo=platform domain=platform_app type=platform_app_data_file
```

这样就可以知道，使用平台签名的 App 所运行的进程 domain 为“platform_app”，并且它的数据文件的 file_type 为“platform_app_data_file”。

又如，使用第三方签名的 App 的 seinfo 为“default”。用户空间的 Security Server 在为它查找对应的 Type 时，使用的 user 输入也为“_app”。我们注意到，在 seapp_contexts 文件中，没有一行对应的 user 和 seinfo 分别为“_app”和“default”。但是有一行是最匹配的，即：

```
user=_app domain=untrusted_app type=app_data_file levelFrom=none
```

这样就可以知道，使用第三方签名的 App 所运行在的进程 domain 为“unstrusted_app”，并且它的数据文件的 file_type 为“app_data_file”。

接下再来看系统文件的安全上下文是如何定义的。通过查看 external/sepolicy/file_contexts 文件，就可以看到系统文件的安全上下文描述，如下所示：

```
##############################################
# Root
/          u:object_r:rootfs:s0
```

```
# Data files
/adb_keys       u:object_r:rootfs:s0
/default.prop       u:object_r:rootfs:s0
/fstab\..*      u:object_r:rootfs:s0
/init\..*       u:object_r:rootfs:s0
/res(/.*)?      u:object_r:rootfs:s0
/ueventd\..*        u:object_r:rootfs:s0

# Executables
/charger        u:object_r:rootfs:s0
/init           u:object_r:rootfs:s0
/sbin(/.*)?     u:object_r:rootfs:s0

......

##############################
# System files
#
/system(/.*)?       u:object_r:system_file:s0
/system/bin/ash     u:object_r:shell_exec:s0
/system/bin/mksh    u:object_r:shell_exec:s0

......
```

文件 file_contexts 通过正则表达式来描述系统文件的安全上下文。例如，在上面列出的内容的最后 3 行中，倒数第三行的正则表达式表示在/system 目录下的所有文件的安全上下文均为“u:object_r:system_file:s0”，最后两行的正则表达式则表示文件/system/bin/ash 和/system/bin/mksh 的安全上下文应为“u:object_r:shell_exec:s0”。虽然倒数第三行的正则表达式描述的文件涵盖后面两个正则表达示描述的文件，但是，后面两个正则表达式描述的方式更加具体，因此，/system/bin/ash 和/system/bin/mksh 两个文件的最终安全上下文都被设置为“u:object_r:shell_exec:s0”。

在 Android 系统中，因为 App 通过读写属性能够获得相应的信息和控制系统的行为，所以 SEAndroid 需要对这些熟悉进行保护，这说明 Android 系统的属性也需要关联安全上下文。这是通过文件 external/sepolicy/property_contexts 来描述的，具体实现代码如下所示。

```
############################
# property service keys
#
#
net.rmnet0              u:object_r:radio_prop:s0
net.gprs                u:object_r:radio_prop:s0
net.ppp                 u:object_r:radio_prop:s0
net.qmi                 u:object_r:radio_prop:s0
net.lte                 u:object_r:radio_prop:s0
net.cdma                u:object_r:radio_prop:s0
gsm.                    u:object_r:radio_prop:s0
persist.radio           u:object_r:radio_prop:s0
net.dns                 u:object_r:radio_prop:s0
sys.usb.config          u:object_r:radio_prop:s0
......
```

属性的安全上下文与文件的安全上下文是类似的，它们的 SELinux 用户、SELinux 角色和安全级别均定义为 u、object_r 和 s0。从上面列出的内容可以看出，以 net.开头的几个属性，以及所有以 gsm.开头的属性、persist.radio 和 sys.usb.config 属性的安全上下文均被设置为“u:object_r:radio_prop:s0”。这意味着只有有权限访问 Type 为 radio_prop 的资源的进程才可以访问这些属性。

2. 安全策略（SEAndroid Policy）

SEAndroid 安全机制中的安全策略是在安全上下文的基础上进行描述的，也就是说，它通过主体和客体的安全上下文，定义主体是否有权限访问客体。SEAndroid 安全机制主要是使用对象安全上下文中的类型来定义安全策略，这种安全策略就称 Type Enforcement，简称 TE。在 external/sepolicy 目录中，所有以.te 为后缀的文件经过编译之后，就会生成一个 sepolicy 文件。这个 sepolicy 文件会打包在 ROM 中，并且保存在设备上的根目录下，即它在设备上的路径为/sepolicy。

在接下来的内容中，通过文件 app.te 的内容来分析在 SEAndroid 安全机制中，使用平台签名的 App 所定义的安全策略，相关代码如下所示。

```
#
# Apps signed with the platform key.
#
type platform_app, domain;
permissive platform_app;
app_domain(platform_app)
platform_app_domain(platform_app)
# Access the network.
net_domain(platform_app)
# Access bluetooth.
bluetooth_domain(platform_app)
unconfined_domain(platform_app)
......
```

从上面列出的内容可以看出，platform_app 接下来会通过 app_domain、platform_app_domain、net_domain、bluetooth_domain 和 unconfined_domain 宏分别加入到其他的 domain 中去，以便可以获得相应的权限。接下来就以 unconfined_domain 宏为例，分析 platform_app 获得了哪些权限。

宏 unconfined_domain 定义在文件 te_macros 中，具体实现代码如下所示。

```
    ......

    ####################################
    # unconfined_domain(domain)
    # Allow the specified domain to do anything.
    #
    define(`unconfined_domain', `
    typeattribute $1 mlstrustedsubject;
    typeattribute $1 unconfineddomain;
    ')
......
```

$1 引用的就是 unconfined_domain 的参数，即 platform_app。通过接下来的两个 typeattribute 语句，为 platform_app 设置了 mlstrustedsubject 和 unconfineddomain 两个属性。也就是说，mlstrustedsubject 和 unconfineddomain 这两个 Type 具有权限，platform_app 这个 Type 也具有。接下来我们主要分析 unconfineddomain 这个 Type 具有哪些权限。

文件 unconfined.te 定义了 unconfineddomain 这个 Type 所具有的权限，具体实现代码如下所示。

```
allow unconfineddomain self:capability_class_set *;
allow unconfineddomain kernel:security *;
allow unconfineddomain kernel:system *;
allow unconfineddomain self:memprotect *;
allow unconfineddomain domain:process *;
allow unconfineddomain domain:fd *;
allow unconfineddomain domain:dir r_dir_perms;
allow unconfineddomain domain:lnk_file r_file_perms;
allow unconfineddomain domain:{ fifo_file file } rw_file_perms;
allow unconfineddomain domain:socket_class_set *;
allow unconfineddomain domain:ipc_class_set *;
allow unconfineddomain domain:key *;
allow unconfineddomain fs_type:filesystem *;
allow unconfineddomain {fs_type dev_type file_type}:{ dir blk_file lnk_file sock_file
fifo_file } *;
allow unconfineddomain {fs_type dev_type file_type}:{ chr_file file } ~entrypoint;
allow unconfineddomain node_type:node *;
allow unconfineddomain node_type:{ tcp_socket udp_socket rawip_socket } node_bind;
allow unconfineddomain netif_type:netif *;
allow unconfineddomain port_type:socket_class_set name_bind;
allow unconfineddomain port_type:{ tcp_socket dccp_socket } name_connect;
allow unconfineddomain domain:peer recv;
allow unconfineddomain domain:binder { call transfer set_context_mgr };
allow unconfineddomain property_type:property_service set;
```

一个 Type 所具有的权限是通过 allow 语句来描述的，例如下面的 allow 语句表明 domain 为 unconfineddomain 的进程可以与其他进程进行 binder ipc 通信（call），并且能够向这些进程传递

Binder 对象（transfer），以及将自己设置为 Binder 上下文管理器（set_context_mgr）。

```
allow unconfineddomain domain:binder { call transfer set_context_mgr };
```

在 SEAndroid 机制中，只有通过 allow 语句声明的权限才是被允许的，而没有通过 allow 语句声明的权限都是禁止的，这样可以最大限度地保护系统中的资源。

如果继续分析文件 app.te 的内容会发现，使用第三方签名的 App 所运行在的进程同样是加入到 unconfineddomain 这个 domain 的，具体实现代码如下所示。

```
#
# Untrusted apps.
#
type untrusted_app, domain;
permissive untrusted_app;
app_domain(untrusted_app)
net_domain(untrusted_app)
bluetooth_domain(untrusted_app)
unconfined_domain(untrusted_app)
```

这是不是意味着使用平台签名和第三方签名的 App 所具有的权限都是一样的呢？答案是否定的。虽然使用平台签名和第三方签名的 App 在 SEAndroid 安全框架的约束下都具有 unconfineddomain 这个 domain 所赋予的权限，但是别忘记，在进行 SEAndroid 安全检查之前，使用平台签名和第三方签名的 App 首先要通过 DAC 检查，也就是要通过传统的 Linux UID/GID 安全检查。由于使用平台签名和第三方签名的 App 在安装的时候分配到的 Linux UID/GID 是不一样的，因此，就决定了它们所具有权限是不一样的。

系统中第一个启动的进程是 init 进程，在启动 init 进程的过程中会执行很多的系统初始化工作，其中就包括加载 SEAndroid 安全策略的工作，具体实现代码如下所示。

```
int main(int argc, char **argv)
{
    ......

    union selinux_callback cb;
    cb.func_log = klog_write;
    selinux_set_callback(SELINUX_CB_LOG, cb);

    cb.func_audit = audit_callback;
    selinux_set_callback(SELINUX_CB_AUDIT, cb);

    INFO("loading selinux policy\n");
    if (selinux_enabled) {
        if (selinux_android_load_policy() < 0) {
            selinux_enabled = 0;
            INFO("SELinux: Disabled due to failed policy load\n");
        } else {
            selinux_init_all_handles();
        }
    } else {
        INFO("SELinux:  Disabled by command line option\n");
    }

    ......
}
```

上述代码定义在文件 system/core/init/init.c 中。调用到了 3 个与 SEAndroid 相关的函数：selinux_set_callback、selinux_android_load_policy 和 selinux_init_all_handles，其中，selinux_set_callback 和 selinux_android_load_policy 来自于 libselinux，而 selinux_init_all_handles 也是定义在文件 system/core/init/init.c 中，并且它最终调用函数 libselinux 来打开文件 file_contexts 和文件 property_contexts，以便可以用来查询系统文件和系统属性的安全上下文。

函数 selinux_set_callback 的功能是向 libselinux 设置 SEAndroid 日志和审计回调函数，而函数 selinux_android_load_policy 的功能是加载安全策略到内核空间的 SELinux LSM 模块中去。函数 selinux_android_load_policy 在文件 external/libselinux/src/android.c 中定义，具体实现代码如下所示。

```
int selinux_android_load_policy(void)
{
    char *mnt = SELINUXMNT;
    int rc;
    rc = mount(SELINUXFS, mnt, SELINUXFS, 0, NULL);
    if (rc < 0) {
        if (errno == ENODEV) {
            /* SELinux not enabled in kernel */
            return -1;
        }
        if (errno == ENOENT) {
            /* Fall back to legacy mountpoint. */
            mnt = OLDSELINUXMNT;
            rc = mkdir(mnt, 0755);
            if (rc == -1 && errno != EEXIST) {
                selinux_log(SELINUX_ERROR,"SELinux:  Could not mkdir:  %s\n",
                    strerror(errno));
                return -1;
            }
            rc = mount(SELINUXFS, mnt, SELINUXFS, 0, NULL);
        }
    }
    if (rc < 0) {
        selinux_log(SELINUX_ERROR,"SELinux:  Could not mount selinuxfs:  %s\n",
                strerror(errno));
        return -1;
    }
    set_selinuxmnt(mnt);

    return selinux_android_reload_policy();
}
```

SELINUXMNT、OLDSELINUXMNT 和 SELINUXFS 是 3 个宏，它们定义在文件 external/libselinux/src/policy.h 文件中，具体如下所示。

```
/* Preferred selinuxfs mount point directory paths. */
#define SELINUXMNT "/sys/fs/selinux"
#define OLDSELINUXMNT "/selinux"

/* selinuxfs filesystem type string. */
#define SELINUXFS "selinuxfs"
```

函数 selinux_android_load_policy 的实现过程如下所示。

- 以/sys/fs/selinux 为安装点，安装一个类型为 selinuxfs 的文件系统，也就是 SELinux 文件系统，用来与内核空间的 SELinux LSM 模块通信。Android 系统以/sys/fs/selinux 为安装点。
- 如果不能在/sys/fs/selinux 这个安装点安装 SELinux 文件系统，那么再以/selinux 为安装点，安装 SELinux 文件系统。
- 在成功安装 SELinux 文件系统之后，调用函数 selinux_android_reload_policy 将 SEAndroid 安全策略加载到内核空间的 SELinux LSM 模块中去。

函数 selinux_android_reload_policy 也是在文件 external/libselinux/src/android.c 中定义，具体实现代码如下所示。

```
static const char *const sepolicy_file[] = {
        "/data/security/current/sepolicy",
        "/sepolicy",
        0 };

......

int selinux_android_reload_policy(void)
{
    int fd = -1, rc;
    struct stat sb;
    void *map = NULL;
    int i = 0;

    while (fd < 0 && sepolicy_file[i]) {
```

```
        fd = open(sepolicy_file[i], O_RDONLY | O_NOFOLLOW);
        i++;
    }
    if (fd < 0) {
        selinux_log(SELINUX_ERROR, "SELinux:  Could not open sepolicy:  %s\n",
                strerror(errno));
        return -1;
    }
    if (fstat(fd, &sb) < 0) {
        selinux_log(SELINUX_ERROR, "SELinux:  Could not stat %s:  %s\n",
                sepolicy_file[i], strerror(errno));
        close(fd);
        return -1;
    }
    map = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (map == MAP_FAILED) {
        selinux_log(SELINUX_ERROR, "SELinux:  Could not map %s:  %s\n",
            sepolicy_file[i], strerror(errno));
        close(fd);
        return -1;
    }

    rc = security_load_policy(map, sb.st_size);
    if (rc < 0) {
        selinux_log(SELINUX_ERROR, "SELinux:  Could not load policy:  %s\n",
            strerror(errno));
        munmap(map, sb.st_size);
        close(fd);
        return -1;
    }

    munmap(map, sb.st_size);
    close(fd);
    selinux_log(SELINUX_INFO, "SELinux: Loaded policy from %s\n", sepolicy_file[i]);

    return 0;
}
```

函数 selinux_android_reload_policy 的执行过程如下所示。

（1）依次从/data/security/current 和根目录寻找 sepolicy 文件，找到之后就打开，获得一个文件描述符 fd。

（2）通过文件描述符 fd 将前面打开的 sepolicy 文件的内容映射到内存中来，并且得到它的起始地址为 map。

（3）调用另外一个函数 security_load_policy 将已经映射到内存中的 sepolicy 文件内容，即 SEAndroid 安全策略，加载到内核空间的 SELinux LSM 模块中去。

（4）加载完成后，释放 sepolicy 文件占用的内存，并且关闭 sepolicy 文件。

函数 security_load_policy 在文件 external/libselinux/src/load_policy.c 中定义，具体实现过程如下所示。

- 首先打/sys/fs/selinux/load 文件。
- 然后将参数 data 所描述的安全策略写入到这个文件中去。
- 由于/sys/fs/selinux 是由内核空间的 SELinux LSM 模块导出来的文件系统接口，因此，当将安全策略写入到位于该文件系统中的 load 文件时，就相当于是将安全策略从用户空间加载到 SELinux LSM 模块中去了。这样，以后 SELinux LSM 模块中的 Security Server 就可以通过它来进行安全检查。

函数 security_load_policy 的具体实现代码如下所示。

```
int security_load_policy(void *data, size_t len)
{
    char path[PATH_MAX];
    int fd, ret;

    if (!selinux_mnt) {N
        errno = ENOENT;
```

```
        return -1;
    }

    snprintf(path, sizeof path, "%s/load", selinux_mnt);
    fd = open(path, O_RDWR);
    if (fd < 0)
        return -1;

    ret = write(fd, data, len);
    close(fd);
    if (ret < 0)
        return -1;
    return 0;
}
```

在上述代码中，selinux_mnt 是一个全局变量，描述了 SELinux 文件系统的安装点，Android 5.0 中的值等于/sys/fs/selinux。

3. Security Server

用户空间的 Security Server 的功能是保护用户空间资源，并操作内核空间对象的安全上下文。Security Server 由应用程序安装服务 PackageManagerService、应用程序安装守护进程 installd、应用程序进程孵化器 Zygote 进程以及 init 进程组成。其中，PackageManagerService 和 installd 负责创建 App 数据目录的安全上下文，Zygote 进程负责创建 App 进程的安全上下文，而 init 进程负责控制系统属性的安全访问。

应用程序安装服务 PackageManagerService 在启动时，会在/etc/security 目录中找到 mac_permissions.xml 文件，然后对它进行解析，得到 App 签名或者包名与 seinfo 的对应关系。当 PackageManagerService 安装 App 的时候，就会根据其签名或者包名查找到对应的 seinfo，并且将这个 seinfo 传递给另外一个守护进程 installed。

在 Android 系统中，守护进程 installd 负责创建 App 数据目录。在创建 App 数据目录时需要给它设置安全上下文，使 SEAndroid 安全机制可以对它进行安全访问控制。installd 根据 PackageManagerService 传递过来的 seinfo，并且调用 libselinux 库提供的函数 selabel_lookup 到 seapp_contexts 文件中查找对应的 Type。通过查找到的这个 Type，installd 就可以给正在安装的 App 的数据目录设置安全上下文，这是通过调用 libselinux 库提供的函数 lsetfilecon 来实现的。

在 Android 系统中，Zygote 进程负责创建应用程序进程。应用程序进程是 SEAndroid 安全机制中的主体，因此，它们也需要设置安全上下文，这是由 Zygote 进程来设置的。组件管理服务 ActivityManagerService 在请求 Zygote 进程创建应用程序进程之前，会到 PackageManagerService 中去查询对应的 seinfo，并且将这个 seinfo 传递到 Zygote 进程。于是，Zygote 进程在 fork 一个应用程序进程之后，就会使用 ActivityManagerService 传递过来的 seinfo，并且调用 libselinux 库提供的 selabel_lookup 函数到前面我们分析的 seapp_contexts 文件中查找到对应的 Domain。有了这个 Domain 之后，Zygote 进程就可以给刚才创建的应用程序进程设置安全上下文了，这是通过调用 libselinux 库提供的 lsetcon 函数来实现的。

19.2 文件安全上下文

SEAndroid 是一种基于安全策略的 MAC 安全机制，此类安全策略在主体和客体的安全上下文之上实施。在实施安全策略之前，SEAndroid 安全机制中的主体和客体是已经有安全上下文的。在 SEAndroid 安全机制中，主体一般就是进程，而客体一般就是文件。文件的安全上下文的关联有不同的方式。

在 SEAndroid 机制中，在创建文件的过程中设置文件的安全上下文。在 Android 系统中，有如下两种产生文件的方式。

- 一种是预置在 ROM 里面的：例如打包在 system.img 里面的文件，它们的安全上下文是在

制作 ROM 的过程中设置的。

- 一种是动态创建的：即系统在运行的过程中创建，它们的安全上下文如果没有特别指定，就与父目录的安全上下文一致。

此外，有些文件的安全上下文是不适合使用父目录的安全上下文的，例如应用程序数据文件，它们的安全上文需要根据一定的规则来特别指定。SEAndroid 安全机制根据应用程序类型的签名来给其数据文件设置不同的安全上下文，以区分系统应用程序和第三方应用程序的数据文件。无论是系统应用程序还是第三方应用程序，因为它们的数据文件都是位于 data 分区的 data 子目录中的，所以需要使用一种机制给在/data/data 目录中创建的数据文件设置不同的安全上下文。在安装应用程序时，PackageManagerService 会通过守护进程 installd 在 /data/data 目录中创建相应的数据目录。这样当以后在运行应用程序的过程中，默认创建的数据文件就位于对应的数据目录中。只要给这些数据目录设置不同 的安全上下文，就可以让不同类型的应用程序在运行的过程中创建不同安全上下文的数据文件。

19.2.1 设置打包在 ROM 里面的文件的安全上下文

接下来将以 ROM 里面的 system.img 为例，介绍打包在 ROM 中的文件的安全上下文的设置过程。在本章 19.1 节中已经分析过 system.img 的制作过程了，因此这里我们只关注与安全上下文设置相关的逻辑。

生成 system.img 的命令位于 build/core/Makefile 文件中，具体实现代码如下所示。

```
BUILT_SYSTEMIMAGE := $(systemimage_intermediates)/system.img

# $(1): output file
define build-systemimage-target
  @echo "Target system fs image: $(1)"
  @mkdir -p $(dir $(1)) $(systemimage_intermediates) && rm -rf $(systemimage_
intermediates)/system_image_info.txt
  $(call generate-userimage-prop-dictionary, $(systemimage_intermediates)/system_
image_info.txt, skip_fsck=true)
  $(hide) PATH=$(foreach p,$(INTERNAL_USERIMAGES_BINARY_PATHS),$(p):)$$PATH \
      ./build/tools/releasetools/build_image.py \
      $(TARGET_OUT) $(systemimage_intermediates)/system_image_info.txt $(1)
endef

$(BUILT_SYSTEMIMAGE): $(FULL_SYSTEMIMAGE_DEPS) $(INSTALLED_FILES_FILE)
    $(call build-systemimage-target,$@)
```

从上述代码可以看出，system.img 由命令 build-system-target 生成。在执行 build-system-target 命令的过程中会执行如下所示的两个子命令。

- generate-userimage-prop-dictionary：生成一个属性文件 system_image_info.txt。
- build_image：制作 system.img 镜像文件。在制作 system.img 镜像文件的过程中，会用到第一个命令生成的属性文件 system_image_info.txt。

上述第一个子命令 generate-userimage-prop-dictionary 在文件 build/core/Makefile 中定义，具体实现代码如下所示。

```
SELINUX_FC := $(TARGET_ROOT_OUT)/file_contexts
......

# $(1): the path of the output dictionary file
# $(2): additional "key=value" pairs to append to the dictionary file.
define generate-userimage-prop-dictionary
$(if $(INTERNAL_USERIMAGES_EXT_VARIANT),$(hide) echo "fs_type=$(INTERNAL_USERIMAGES_
EXT_VARIANT)" >> $(1))
$(if $(BOARD_SYSTEMIMAGE_PARTITION_SIZE),$(hide) echo "system_size=$(BOARD_
SYSTEMIMAGE_PARTITION_SIZE)" >> $(1))
$(if $(BOARD_USERDATAIMAGE_PARTITION_SIZE),$(hide) echo "userdata_size=$(BOARD_
USERDATAIMAGE_PARTITION_SIZE)" >> $(1))
$(if $(BOARD_CACHEIMAGE_FILE_SYSTEM_TYPE),$(hide) echo "cache_fs_type=$(BOARD_
CACHEIMAGE_FILE_SYSTEM_TYPE)" >> $(1))
$(if $(BOARD_CACHEIMAGE_PARTITION_SIZE),$(hide) echo "cache_size=$(BOARD_CACHEIMAGE_
```

```
PARTITION_SIZE)" >> $(1))
$(if $(BOARD_VENDORIMAGE_FILE_SYSTEM_TYPE),$(hide) echo "vendor_fs_type=$(BOARD_
VENDORIMAGE_FILE_SYSTEM_TYPE)" >> $(1))
$(if $(BOARD_VENDORIMAGE_PARTITION_SIZE),$(hide) echo "vendor_size=$(BOARD_
VENDORIMAGE_PARTITION_SIZE)" >> $(1))
$(if $(INTERNAL_USERIMAGES_SPARSE_EXT_FLAG),$(hide) echo "extfs_sparse_flag=$
(INTERNAL_USERIMAGES_SPARSE_EXT_FLAG)" >> $(1))
$(if $(mkyaffs2_extra_flags),$(hide) echo "mkyaffs2_extra_flags=$(mkyaffs2_
extra_flags)" >> $(1))
$(hide) echo "selinux_fc=$(SELINUX_FC)" >> $(1)
```

此处传过来的第一个参数便是指向上述的属性文件 system_image_info.txt，它的内容是通过一系列的 echo 命令生成的，每一行都是“key=value”形式。其中与文件安全上下文相关的是最后一行。

```
selinux_fc=$(SELINUX_FC)
```

变量 SELINUX_FC 指向一个 file_contexts 文件。这个 file_contexts 文件就是我们在上一节中提到的 file_contexts 文件，用来描述文件的安全上下文。我们知道，system.img 镜像文件是安装在目标设备上的/system 目录中的，因此，我们就观察一下在 file_contexts 文件中与/system 目录相关的文件的安全上下文是如何设置的。

文件 file_contexts 最开始是位于 build/external/sepolicy 目录中，经过编译后，就会保存在$OUT/root 目录中，其中$OUT 指向的是产品输出目录。打开$OUT/root/file_contexts 文件，可以看到与/system 目录相关的文件的安全上下文的设置规则，具体内容如下所示。

```
##############################
# System files
#
/system(/.*)?       u:object_r:system_file:s0
/system/bin/ash     u:object_r:shell_exec:s0
/system/bin/mksh    u:object_r:shell_exec:s0
/system/bin/sh      --  u:object_r:shell_exec:s0
/system/bin/run-as  --  u:object_r:runas_exec:s0
/system/bin/app_process u:object_r:zygote_exec:s0
/system/bin/servicemanager  u:object_r:servicemanager_exec:s0
/system/bin/surfaceflinger  u:object_r:surfaceflinger_exec:s0
/system/bin/drmserver   u:object_r:drmserver_exec:s0
/system/bin/vold    u:object_r:vold_exec:s0
/system/bin/netd    u:object_r:netd_exec:s0
/system/bin/rild    u:object_r:rild_exec:s0
/system/bin/mediaserver u:object_r:mediaserver_exec:s0
/system/bin/dbus-daemon u:object_r:dbusd_exec:s0
/system/bin/installd    u:object_r:installd_exec:s0
/system/bin/keystore    u:object_r:keystore_exec:s0
/system/bin/debuggerd   u:object_r:debuggerd_exec:s0
/system/bin/bluetoothd  u:object_r:bluetoothd_exec:s0
/system/bin/wpa_supplicant  u:object_r:wpa_exec:s0
/system/bin/qemud   u:object_r:qemud_exec:s0
/system/bin/sdcard      u:object_r:sdcardd_exec:s0
/system/bin/dhcpcd      u:object_r:dhcp_exec:s0
/system/bin/mtpd    u:object_r:mtp_exec:s0
/system/bin/pppd    u:object_r:ppp_exec:s0
/system/bin/tf_daemon   u:object_r:tee_exec:s0
/system/bin/racoon  u:object_r:racoon_exec:s0
/system/etc/ppp(/.*)?   u:object_r:ppp_system_file:s0
/system/etc/dhcpcd(/.*)? u:object_r:dhcp_system_file:s0
/system/xbin/su     u:object_r:su_exec:s0
/system/vendor/bin/gpsd u:object_r:gpsd_exec:s0
/system/bin/ping    u:object_r:ping_exec:s0
```

下面再来看第二个子命令 build_image 的实现，它是由文件 build/tools/releasetools/build_image.py 实现的，其入口函数 main 的实现代码如下所示。

```
def main(argv):
  ......

  in_dir = argv[0]
  glob_dict_file = argv[1]
```

```
  out_file = argv[2]

  glob_dict = LoadGlobalDict(glob_dict_file)
  image_filename = os.path.basename(out_file)
  mount_point = ""
  if image_filename == "system.img":
    mount_point = "system"
  elif image_filename == "userdata.img":
    mount_point = "data"
  elif image_filename == "cache.img":
    mount_point = "cache"
  elif image_filename == "vendor.img":
    mount_point = "vendor"
  else:
    print >> sys.stderr, "error: unknown image file name ", image_filename
    exit(1)

  image_properties = ImagePropFromGlobalDict(glob_dict, mount_point)
  if not BuildImage(in_dir, image_properties, out_file):
    print >> sys.stderr, "error: failed to build %s from %s" % (out_file, in_dir)
    exit(1)

if __name__ == '__main__':
  main(sys.argv[1:])
```

参数 argv[1]指向的就是上面提到的属性文件 system_image_info.txt，最终保存在本地变量 glob_dict_file 中。另外一个参数 argv[2]指向的是要输出的 system.img 文件路径，最终保存在本地变量 out_file 中。

函数 LoadGlobalDict 用来打开属性文件 system_image_info.txt，并且将它每一行的 key 和 value 提取出来，并且保存在字典 glob_dict 中。注意，这个字典 glob_dict 包含有一个 key 等于 selinux_fc、value 等于 file_contexts 文件路径的项。

接下来再通过 os.path.basename 将输出的文件路径 out_file 的最后一项提取出来，就可以得到 image_filename 的值为“system.img”，因此，再接下来就会得到本地变量 mount_point 的值为“system”，表示我们现在正在打包的是 system.img 文件。

函数 ImagePropFromGlobalDict 的功能是从字典 glob_dict 中提取与安装点 mount_point 相关的项，并且保存在另外一个字典中返回给调用者。具体实现代码如下所示。

```
def ImagePropFromGlobalDict(glob_dict, mount_point):
  """Build an image property dictionary from the global dictionary.

  Args:
    glob_dict: the global dictionary from the build system.
    mount_point: such as "system", "data" etc.
  """
  d = {}

  def copy_prop(src_p, dest_p):
    if src_p in glob_dict:
      d[dest_p] = str(glob_dict[src_p])

  common_props = (
      "extfs_sparse_flag",
      "mkyaffs2_extra_flags",
      "selinux_fc",
      "skip_fsck",
      )
  for p in common_props:
    copy_prop(p, p)

  d["mount_point"] = mount_point
  if mount_point == "system":
    copy_prop("fs_type", "fs_type")
    copy_prop("system_size", "partition_size")
  elif mount_point == "data":
    copy_prop("fs_type", "fs_type")
    copy_prop("userdata_size", "partition_size")
```

```
  elif mount_point == "cache":
    copy_prop("cache_fs_type", "fs_type")
    copy_prop("cache_size", "partition_size")
  elif mount_point == "vendor":
    copy_prop("vendor_fs_type", "fs_type")
    copy_prop("vendor_size", "partition_size")

  return d
```

通过上述代码可以看出，在函数 ImagePropFromGlobalDict 返回给调用者的字典中，包含了一个以 selinux_fc 为 key 值的项，其值指向上述分析的 fiiles_contexts 文件。

再次来到函数 main 中，最后调用函数 BuildImage 来生成最终的 system.img 文件，具体实现代码如下所示。

```
def BuildImage(in_dir, prop_dict, out_file):
  """Build an image to out_file from in_dir with property prop_dict.

  Args:
    in_dir: path of input directory.
    prop_dict: property dictionary.
    out_file: path of the output image file.

  Returns:
    True iff the image is built successfully.
  """
  build_command = []
  fs_type = prop_dict.get("fs_type", "")
  run_fsck = False
  if fs_type.startswith("ext"):
    build_command = ["mkuserimg.sh"]
    ......
    build_command.extend([in_dir, out_file, fs_type,
                          prop_dict["mount_point"]])
    ......
    if "selinux_fc" in prop_dict:
      build_command.append(prop_dict["selinux_fc"])
  else:
    build_command = ["mkyaffs2image", "-f"]
    ......
    build_command.append(in_dir)
    build_command.append(out_file)
    if "selinux_fc" in prop_dict:
      build_command.append(prop_dict["selinux_fc"])
      build_command.append(prop_dict["mount_point"])

  exit_code = RunCommand(build_command)
  ......

  return exit_code == 0
```

在上述代码中，参数 prop_dict 指向的就是前面调用 ImagePropFromGlobalDict 获得的字典，如果它里面包含有一个 key 为“fs_type”的项，并且它的 value 等于“ext”，那么就意味着将要制作 ext 格式的 system.img 镜像文件，否则的话，就意味着将要制作 yaffs2 格式的 system.img 镜像文件。前者通过命令 mkuserimg 来生成，而后者通过命令 mkyaffs2image 来生成。无论生成的是什么格式的 system.img 镜像文件，只要参数 prop_dict 包含有一个 key 为“selinux_fc”的项，那么都会将它的 value 提取出来，并且作为一个参数传递给命令 mkuserimg 或 mkyaffs2image 来使用。参数 prop_dict 描述的字典包含有 key 为“selinux_fc”的项，并且它的 value 描述的就是我们在上面提到的 file_contexts 的路径。当将这个 file_contexts 文件路径传递给命令 mkuserimg 或者 mkyaffs2image 时，后者就会根据它设置的规则给打包在 system.img 里面的文件设置对应的安全上下文。这样就获得了一个关联有安全上下文的 system.img 镜像文件。

19.2.2　设置虚拟文件系统的安全上下文

启动 Android 系统之后，会由 init 进程在/sys/fs/selinux 中安装一个 SELinux 虚拟文件系统，

然后再加载 SEAndroid 安全策略到内核空间的 SELinux LSM 模块中去。

SEAndroid 安全策略是由 external/sepolicy 模块生成的。通过分析 external/sepolicy 模块的编译脚本 Android.mk 会发现，SEAndroid 安全策略包含有文件 genfs_contexts 定义的安全上下文设置规则，具体内容如下所示。

```
###################################
include $(CLEAR_VARS)

LOCAL_MODULE := sepolicy
LOCAL_MODULE_CLASS := ETC
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_PATH := $(TARGET_ROOT_OUT)

include $(BUILD_SYSTEM)/base_rules.mk

sepolicy_policy.conf := $(intermediates)/policy.conf
$(sepolicy_policy.conf): PRIVATE_MLS_SENS := $(MLS_SENS)
$(sepolicy_policy.conf): PRIVATE_MLS_CATS := $(MLS_CATS)
$(sepolicy_policy.conf)   :   $(call   build_policy,   security_classes   initial_sids
access_vectors global_macros mls_macros mls policy_capabilities te_macros attributes
*.te roles users initial_sid_contexts fs_use genfs_contexts port_contexts)
    @mkdir -p $(dir $@)
    $(hide) m4 -D mls_num_sens=$(PRIVATE_MLS_SENS) -D mls_num_cats=$(PRIVATE_MLS_CATS)
-s $^ > $@
    $(hide) sed '/dontaudit/d' $@ > $@.dontaudit

$(LOCAL_BUILT_MODULE) : $(sepolicy_policy.conf) $(HOST_OUT_EXECUTABLES)/checkpolicy
    @mkdir -p $(dir $@)
    $(hide) $(HOST_OUT_EXECUTABLES)/checkpolicy -M -c $(POLICYVERS) -o $@ $<
    $(hide) $(HOST_OUT_EXECUTABLES)/checkpolicy -M -c $(POLICYVERS) -o $(dir $<)/$(notdir
$@).dontaudit $<.dontaudit

built_sepolicy := $(LOCAL_BUILT_MODULE)
sepolicy_policy.conf :=
```

生成的 SEAndroid 安全策略被保存在文件 policy.conf 中，在生成这个 policy.conf 文件的过程中会调用 build_policy 函数。在传递给函数 build_policy 的参数中，包含了生成 SEAndroid 安全策略所需要的源文件。其中，源文件 genfs_contexts 描述了虚拟文件系统的安全上下文设置规则，具体内容如下所示。

```
# Label inodes with the fs label.
genfscon rootfs / u:object_r:rootfs:s0
# proc labeling can be further refined (longest matching prefix).
genfscon proc / u:object_r:proc:s0
genfscon proc /net/xt_qtaguid/ctrl u:object_r:qtaguid_proc:s0
# selinuxfs booleans can be individually labeled.
genfscon selinuxfs / u:object_r:selinuxfs:s0
genfscon cgroup / u:object_r:cgroup:s0
# sysfs labels can be set by userspace.
genfscon sysfs / u:object_r:sysfs:s0
genfscon inotifyfs / u:object_r:inotify:s0
genfscon vfat / u:object_r:sdcard_external:s0
genfscon debugfs / u:object_r:debugfs:s0
genfscon fuse / u:object_r:sdcard_internal:s0
```

从这里就可以看出 SELinux 虚拟文件系统关联的安全上下文为“u:object_r:selinuxfs:s0”，这说明只有那些对 Type 为“selinuxfs”的文件有访问权限的进程才可以访问 SELinux 虚拟文件系统，即/sys/fs/selinux 目录下的文件。

19.2.3 设置应用程序数据文件的安全上下文

在 Android 系统中，每一个应用程序在/data/data 目录下都有一个用包名命名的目录，用来作为数据的保存目录，这个数据目录是在应用程序安装的时候由守护进程 installd 创建的。当守护进程 installd 创建应用程序数据目录的时候，会同时设置它的安全上下文，以便可以对它进行保护。

PackageManagerService 负责安装 Android 应用程序，它在启动时会通过类 SELinuxMMAC 的

静态成员函数 readInstallPolicy 读取本章前面 19.1 节中的 mac_permissions.xml 文件，具体实现代码如下所示。

```
public class PackageManagerService extends IPackageManager.Stub {
    ......

    public PackageManagerService(Context context, Installer installer,
            boolean factoryTest, boolean onlyCore) {
        ......

        synchronized (mInstallLock) {
        // writer
        synchronized (mPackages) {
            ......

            mFoundPolicyFile = SELinuxMMAC.readInstallPolicy();

            ......
        } // synchronized (mPackages)
        } // synchronized (mInstallLock)
    }

    ......
}
```

函数 readInstallPolicy 在文件 frameworks/base/services/java/com/android/server/pm/PackageManagerService.java 中定义，具体实现代码如下所示。

```
public final class SELinuxMMAC {
    ......

    // Signature seinfo values read from policy.
    private static final HashMap<Signature, String> sSigSeinfo =
        new HashMap<Signature, String>();

    // Package name seinfo values read from policy.
    private static final HashMap<String, String> sPackageSeinfo =
        new HashMap<String, String>();

    // Locations of potential install policy files.
    private static final File[] INSTALL_POLICY_FILE = {
        new File(Environment.getDataDirectory(), "system/mac_permissions.xml"),
        new File(Environment.getRootDirectory(), "etc/security/mac_permissions.xml"),
        null};
    ......

    public static boolean readInstallPolicy() {

        return readInstallPolicy(INSTALL_POLICY_FILE);
    }
    ......

    private static boolean readInstallPolicy(File[] policyFiles) {
        FileReader policyFile = null;
        int i = 0;
        while (policyFile == null && policyFiles != null && policyFiles[i] != null) {
            try {
                policyFile = new FileReader(policyFiles[i]);
                break;
            } catch (FileNotFoundException e) {
                Slog.d(TAG,"Couldn't find install policy " + policyFiles[i].getPath());
            }
            i++;
        }

        ......

        try {
            XmlPullParser parser = Xml.newPullParser();
            parser.setInput(policyFile);
```

```
            XmlUtils.beginDocument(parser, "policy");
            while (true) {
                XmlUtils.nextElement(parser);
                if (parser.getEventType() == XmlPullParser.END_DOCUMENT) {
                    break;
                }

                String tagName = parser.getName();
                if ("signer".equals(tagName)) {
                    String cert = parser.getAttributeValue(null, "signature");
                    ......
                    Signature signature;
                    try {
                        signature = new Signature(cert);
                    } catch (IllegalArgumentException e) {
                        ......
                    }
                    String seinfo = readSeinfoTag(parser);
                    if (seinfo != null) {
                        ......
                        sSigSeinfo.put(signature, seinfo);
                    }
                } else if ("default".equals(tagName)) {
                    String seinfo = readSeinfoTag(parser);
                    if (seinfo != null) {
                        ......
                        // The 'null' signature is the default seinfo value
                        sSigSeinfo.put(null, seinfo);
                    }
                } else if ("package".equals(tagName)) {
                    String pkgName = parser.getAttributeValue(null, "name");
                    ......
                    String seinfo = readSeinfoTag(parser);
                    if (seinfo != null) {
                        ......
                        sPackageSeinfo.put(pkgName, seinfo);
                    }
                } else {
                    XmlUtils.skipCurrentTag(parser);
                    continue;
                }
            }
        } catch (XmlPullParserException e) {
            Slog.w(TAG, "Got execption parsing ", e);
        } catch (IOException e) {
            Slog.w(TAG, "Got execption parsing ", e);
        }
        try {
            policyFile.close();
        } catch (IOException e) {
            //omit
        }
        return true;
    }

    ......
}
```

类 SELinuxMMAC 中的两个静态成员变量 sSigSeinfo 和 sPackageSeinfo 指向的都是一个 HashMap，其中 sSigSeinfo 是以应用程序签名为 Key 值，而 sPackageSeinfo 是以应用程序包名为 Key 值，并且两者的 Value 描述的都是一个 seinfo 字符串。这个 seinfo 字符串决定了应用程序数据目录的安全上下文设置。

再看 SELinuxMMAC 类的静态成员函数 readInstallPolicy，这里列出了两个重载版本的实现。其中，无参数版本的 readInstallPolicy 以静态成员 INSTALL_POLICY_FILE 描述的一个数组为参数，调用 File 数组参数版本的 readInstallPolicy。

SELinuxMMAC 类的静态成员变量 INSTALL_POLICY_FILE 指向的是一个 File 数组，它里面

包含了两个文件路径，分别是/data/system/mac_permissions.xml 和/etc/security/mac_permissions.xml。从 File 数组参数版本的 readInstallPolicy 开始实现可以知道，它会依次检查/data/system/mac_permissions.xml 和/etc/security/mac_permissions.xml 文件的存在性。只要其中的一个文件存在，那么就会停止检查过程，并且将已经存在的文件打开，以便接下来可以对它进行读取和解析。一般来说，mac_permissions.xml 文件都是保存在/etc/security 目录下的。

打开了要读取的解析的 mac_permissions.xml 文件之后，接下来就开始对它进行解析了。mac_permissions.xml 是一个 xml 文件，对 mac_permissions.xml 的解析比较简单，具体说明如下所示。

- 如果碰到一个以“signer”为名称的 xml 标签，那么就将它的 signature 属性值读取出来，并且以此为参数创建了一个 Signature 对象。接下来再通过另外一个函数 readSeinfoTag 在以“signer”为名称的 xml 标签中，寻找一个名称为“seinfo”的 xml 标签，并且将它的 value 属性值读取出来，作为一个 seinfo 字符串。最后，就以刚才创建的 Signature 对象为 Key 值，并且以通过函数 readSeinfoTag 读取出来的 seinfo 字符串为 Value 值，保存在 SELinuxMMAC 类的静态成员函数 sSigSeinfo 所描述的一个 HashMap 中。
- 如果碰到一个以“package”为名称的 xml 标签，那么就将它的 name 属性值读取出来，作为包名。接下来同样是通过函数 readSeinfoTag 在以“package”为名称的 xml 标签中，寻找一个名称为“seinfo”的 xml 标签，并且将它的 value 属性值读取出来，作为一个 seinfo 字符串。最后，就以刚才读取的包名为 Key 值，并且以通过函数 readSeinfoTag 读取出来的 seinfo 字符串为 Value 值，保存在 SELinuxMMAC 类的静态成员函数 sPackageSeinfo 所描述的一个 HashMap 中。
- 如果碰到一个以“default”为名称的 xml 标签，那么就直接通过函数 readSeinfoTag 在其子标签中，寻找一个名称为“seinfo”的 xml 标签，并且将它的 value 属性值读取出来，作为一个 seinfo 字符串。最后，以 null 为 Key 值，并且以通过函数 readSeinfoTag 读取出来的 seinfo 字符串为 Value 值，保存在 SELinuxMMAC 类的静态成员函数 sSigSeinfo 所描述的一个 HashMap 中。

也就是说，可以在文件 mac_permissions.xml 中通过签名或者包名来为某一个应用程序设置特定的 seinfo 字符串，而对于没有显示通过签名或者包名来设置 seinfo 字符串的应用程序，那么它们的 seinfo 字符串就由 mac_permissions.xml 文件中的 default 标签来确定。

PackageManagerService 通过调用 SELinuxMMAC 类的静态成员函数 readInstallPolicy，初始化安装应用程序时要用到的 seinfo 字符串。

PackageManagerService 在安装应用程序时会调用到 PackageManagerService 类的成员函数 scanPackageLI 解析应用程序的信息，这个函数会同时给正在安装的应用程序分配一个 seinfo 字符串。具体实现代码如下所示。

```
public class PackageManagerService extends IPackageManager.Stub {
    ......

    private PackageParser.Package scanPackageLI(PackageParser.Package pkg,
            int parseFlags, int scanMode, long currentTime, UserHandle user) {
        ......

        synchronized (mPackages) {
            ......

            if (mFoundPolicyFile) {
                SELinuxMMAC.assignSeinfoValue(pkg);
            }

            ......
        }

        ......
    }

    ......
}
```

这个函数定义在文件 frameworks/base/services/java/com/android/server/pm/PackageManager

Service.java 中。在 PackageManagerService 类的成员函数 scanPackageLI 中，参数 pkg 指向的是一个 Package 对象，用来描述正在安装的应用程序。如果 PackageManagerService 类的成员变量 mFoundPolicyFile 的值等于 true，那么就说明前面已经成功地读取和解析到了一个 mac_permissions.xml 文件，这时候就会调用 SELinuxMMAC 类的静态成员函数 assignSeinfoValue 给正在安装的应用程序设置一个 seinfo 字符串。

在类 SELinuxMMAC 中，静态成员函数 assignSeinfoValue 在文件 frameworks/base/services/java/com/android/server/pm/SELinuxMMAC.java 中定义，具体实现代码如下所示。

```
public final class SELinuxMMAC {
    ......

    public static void assignSeinfoValue(PackageParser.Package pkg) {
        ......

        if (((pkg.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) != 0) ||
            ((pkg.applicationInfo.flags & ApplicationInfo.FLAG_UPDATED_SYSTEM_APP) != 0))
{

            // We just want one of the signatures to match.
            for (Signature s : pkg.mSignatures) {
                ......

                if (sSigSeinfo.containsKey(s)) {
                    String seinfo = pkg.applicationInfo.seinfo = sSigSeinfo.get(s);
                    ......

                    return;
                }
            }

            // Check for seinfo labeled by package.
            if (sPackageSeinfo.containsKey(pkg.packageName)) {
                String seinfo = pkg.applicationInfo.seinfo = sPackageSeinfo.get(pkg.
                packageName);
                ......

                return;
            }
        }

        String seinfo = pkg.applicationInfo.seinfo = sSigSeinfo.get(null);
        ......
    }
}
```

这样最终获得的 seinfo 字符串都保存在用来描述应用程序安装信息的 Package 对象的成员变量 applicationInfo 所描述的一个 ApplicationInfo 对象的成员变量 seinfo 中。

当 PackageManagerService 解析完成要安装的应用程序的信息之后，接下来就会调用 PackageManagerService 类的成员函数 createDataDirsLI 来给正在安装的应用程序创建数据目录，这个函数在文件 frameworks/base/services/java/com/android/server/pm/PackageManagerService.java 中定义，具体实现代码如下所示。

```
public class PackageManagerService extends IPackageManager.Stub {
    ......

    private int createDataDirsLI(String packageName, int uid, String seinfo) {
        int[] users = sUserManager.getUserIds();
        int res = mInstaller.install(packageName, uid, uid, seinfo);
        ......
    }

    ......
}
```

参数 packageName 表示正在安装的应用程序的包名，而参数 uid 和 seinfo 表示分配给正在安

装的应用程序的 uid 和 seinfo。PackageManagerService 类的成员函数 createDataDirsLI 通过调用成员变量 mInstaller 指向的一个 Installer 对象的成员函数 install 来为正在安装的应用程序创建数据目录，并且会将包名、uid 和 seinfo 等信息传递给它。

Installer 类的成员函数 install 在 frameworks/base/services/java/com/android/server/pm/Installer.java 文件中定义，具体实现代码如下所示。

```
public final class Installer {
    ......

    public int install(String name, int uid, int gid, String seinfo) {
        StringBuilder builder = new StringBuilder("install");
        builder.append(' ');
        builder.append(name);
        builder.append(' ');
        builder.append(uid);
        builder.append(' ');
        builder.append(gid);
        builder.append(' ');
        builder.append(seinfo != null ? seinfo : "!");
        return execute(builder.toString());
    }

    ......
}
```

函数 install 通过 socket 向守护进程 installd 发送一个 install 命令，请求为正在安装的应用程序创建数据目录，并且会将包名、uid 和 seinfo 等信息传递给它。守护进程 installd 的源代码位于 frameworks/native/cmds/installd 目录中，当它收到 PackageManagerService 发送过来的 intall 命令后，就会调用函数 install 为正在安装的应用程序创建数据目录。函数 install 在文件 rameworks/native/cmds/installd/commands.c 中定义，具体实现代码如下所示。

```
int install(const char *pkgname, uid_t uid, gid_t gid, const char *seinfo)
{
    char pkgdir[PKG_PATH_MAX];
    ......

    if (create_pkg_path(pkgdir, pkgname, PKG_DIR_POSTFIX, 0)) {
        ALOGE("cannot create package path\n");
        return -1;
    }

    ......

    if (selinux_android_setfilecon2(pkgdir, pkgname, seinfo, uid) < 0) {
        ALOGE("cannot setfilecon dir '%s': %s\n", pkgdir, strerror(errno));
        ......
        return -errno;
    }

    ......

    return 0;
}
```

函数 install 首先调用函数 create_pkg_path 在/data/data 目录下创建一下名称等于包名 pkgname 的子目录，然后调用由 libselinux 库提供的函数 selinux_android_setfilecon2 为刚才创建的目录设置安全上下文。

函数 selinux_android_setfilecon2 在文件 external/libselinux/src/android.c 中定义，具体实现代码如下所示。

```
int selinux_android_setfilecon2(const char *pkgdir,
                const char *pkgname,
                const char *seinfo,
                uid_t uid)
{
    char *orig_ctx_str = NULL, *ctx_str;
```

```
    context_t ctx = NULL;
    int rc;

    if (is_selinux_enabled() <= 0)
        return 0;

    __selinux_once(once, seapp_context_init);

    rc = getfilecon(pkgdir, &ctx_str);
    if (rc < 0)
        goto err;

    ctx = context_new(ctx_str);
    orig_ctx_str = ctx_str;
    if (!ctx)
        goto oom;

    rc = seapp_context_lookup(SEAPP_TYPE, uid, 0, seinfo, pkgname, ctx);
    if (rc == -1)
        goto err;
    else if (rc == -2)
        goto oom;

    ctx_str = context_str(ctx);
    if (!ctx_str)
        goto oom;
    rc = security_check_context(ctx_str);
    if (rc < 0)
        goto err;

    if (strcmp(ctx_str, orig_ctx_str)) {
        rc = setfilecon(pkgdir, ctx_str);
        if (rc < 0)
            goto err;
    }

    rc = 0;
out:
    freecon(orig_ctx_str);
    context_free(ctx);
    return rc;
err:
    selinux_log(SELINUX_ERROR, "%s:  Error setting context for pkgdir %s, uid %d: %s\n",
            __FUNCTION__, pkgdir, uid, strerror(errno));
    rc = -1;
    goto out;
oom:
    selinux_log(SELINUX_ERROR, "%s:  Out of memory\n", __FUNCTION__);
    rc = -1;
    goto out;
}
```

函数 selinux_android_setfilecon2 的具体实现过程如下所示。

（1）调用函数 is_selinux_enabled 检查系统是否启用了 SELinux。如果没有启用，那就什么也不用做就返回；否则的话，就继续往下执行。

（2）调用函数 seapp_context_init 读取和解析我们在前面 SEAndroid 安全机制框架分析中提到的 seapp_contexts 文件。注意，__selinux_once 是一个宏，它实际上是利用了 pthread 库提供的函数 pthread_once 保证函数 seapp_context_init 在进程内有且仅有一次会被调用到，适合用来执行初始化工作。

（3）调用函数 getfilecon 获得要设置新的安全上下文的目录 pkgdir 原来已有的安全上下文，保存在变量 ctx_str。我们在 19.1 节中提到，文件或者目录在创建的时候，默认设置的是父目录的安全上下文。

（4）调用函数 context_new 在原来的安全上下文的基础上创建一个新的安全上下文 ctx，以便可以获得原来安全上下文的 SELinux 用户、角色和安全级别等信息。

（5）调用函数 seapp_context_lookup 根据传进来的参数 seinfo 在 seapp_contexts 文件中找到对

应的 Type，并且将其设置为新的安全上下文 ctr 的 Type。

（6）调用函数 context_str 获得新创建的安全上下文的字符串描述，以便可以调用函数 ecurity_check_context 来验证新创建的安全上下文的正确性。如果不正确的话，就出错返回。否则的话，继续往下执行。

（7）比较原来的安全上下文和新创建的安全上下文。如果不一致的话，那么就调用函数 setfilecon 将目录 pkgdir 的安全上下文设置为新创建的安全上下文。

在上面几个步骤中，最重要的就是（2）、（5）和（7）3 步，因此，接下来就继续分析函数 seapp_context_init、seapp_context_lookup 和 setfilecon 的实现。

函数 seapp_context_init 在 external/libselinux/src/android.c 文件中定义，通过调用另外一个函数 selinux_android_seapp_context_reload 来读取和解析 seapp_contexts 文件。具体实现代码如下所示。

```
static void seapp_context_init(void)
    {
            selinux_android_seapp_context_reload();
    }
```

函数 selinux_android_seapp_context_reload 在 external/libselinux/src/android.c 文件中定义，具体实现代码如下所示。

```
int selinux_android_seapp_context_reload(void)
    {
        FILE *fp = NULL;
        char line_buf[BUFSIZ];
        ......
        struct seapp_context *cur;
        ......

        while ((fp==NULL) && seapp_contexts_file[i])
            fp = fopen(seapp_contexts_file[i++], "r");
        ......

        seapp_contexts = calloc(nspec, sizeof(struct seapp_context *));
        ......

        while (fgets(line_buf, sizeof line_buf - 1, fp)) {
            ......

            cur = calloc(1, sizeof(struct seapp_context));
            ......

            token = strtok_r(p, " \t", &saveptr);
            ......

            while (1) {
                name = token;
                value = strchr(name, '=');
                ......

                if (!strcasecmp(name, "isSystemServer")) {
                    if (!strcasecmp(value, "true"))
                        cur->isSystemServer = 1;
                    else if (!strcasecmp(value, "false"))
                        cur->isSystemServer = 0;
                    ......
                } else if (!strcasecmp(name, "user")) {
                    cur->user = strdup(value);
                    ......
                } else if (!strcasecmp(name, "seinfo")) {
                    cur->seinfo = strdup(value);
                    ......
                } else if (!strcasecmp(name, "name")) {
                    cur->name = strdup(value);
                    ......
                } else if (!strcasecmp(name, "domain")) {
                    cur->domain = strdup(value);
                    ......
```

```
            } else if (!strcasecmp(name, "type")) {
                cur->type = strdup(value);
                ......
            } else if (!strcasecmp(name, "levelFromUid")) {
                if (!strcasecmp(value, "true"))
                    cur->levelFrom = LEVELFROM_APP;
                else if (!strcasecmp(value, "false"))
                    cur->levelFrom = LEVELFROM_NONE;
                ......
            } else if (!strcasecmp(name, "levelFrom")) {
                if (!strcasecmp(value, "none"))
                    cur->levelFrom = LEVELFROM_NONE;
                else if (!strcasecmp(value, "app"))
                    cur->levelFrom = LEVELFROM_APP;
                else if (!strcasecmp(value, "user"))
                    cur->levelFrom = LEVELFROM_USER;
                else if (!strcasecmp(value, "all"))
                    cur->levelFrom = LEVELFROM_ALL;
                .......
            } else if (!strcasecmp(name, "level")) {
                cur->level = strdup(value);
                ......
            } else if (!strcasecmp(name, "sebool")) {
                cur->sebool = strdup(value);
                ......
            }
            ......

            token = strtok_r(NULL, " \t", &saveptr);
            ......
        }

        seapp_contexts[nspec] = cur;
        ......
    }

    ......

    ret = 0;
out:
    fclose(fp);
    return ret;

    ......
}
```

seapp_contexts_file 是一个全局变量，也是定义在 external/libselinux/src/android.c 文件中，具体实现代码如下所示。

```
static char const * const seapp_contexts_file[] = {
        "/data/security/current/seapp_contexts",
        "/seapp_contexts",
        0 };
```

此时可以看到，seapp_contexts 可能保存在目录/data/security/current 中，也有可能保存在根目录中，函数 selinux_android_seapp_context_reload 依次进行检查。只要其中一个地方存在，那么就对它进行打开。打开之后，接下来就按行进行解析。

文件 seapp_contexts 的内容如下所示。

```
isSystemServer=true domain=system
    user=system domain=system_app type=system_data_file
    user=bluetooth domain=bluetooth type=bluetooth_data_file
    user=nfc domain=nfc type=nfc_data_file
    user=radio domain=radio type=radio_data_file
    user=_app domain=untrusted_app type=app_data_file levelFrom=none
    user=_app seinfo=platform domain=platform_app type=platform_app_data_file
    user=_app seinfo=shared domain=shared_app type=platform_app_data_file
    user=_app seinfo=media domain=media_app type=platform_app_data_file
    user=_app seinfo=release domain=release_app type=platform_app_data_file
    user=_isolated domain=isolated_app
```

函数 selinux_android_seapp_context_reload 将每一行的内容都用一个 seapp_context 结构体来描述，并且将这些结构体全部缓存在全局变量 seapp_contexts 中。通过这种方式，以后需要从文件 seapp_contexts 中查找文件的 Type 或者进程的 Domain 时，就可以直接在内存进行，加快了查找的速度。

函数 seapp_context_lookup 在文件 external/libselinux/src/android.c 中定义，功能是从全局变量 seapp_contexts 中查找文件的 Type 或者进程的 Domain。具体实现代码如下所示。

```
static int seapp_context_lookup(enum seapp_kind kind,
                uid_t uid,
                int isSystemServer,
                const char *seinfo,
                const char *pkgname,
                context_t ctx)
{
    const char *username = NULL;
    ......
    struct seapp_context *cur;
    ......

    userid = uid / AID_USER;
    appid = uid % AID_USER;
    if (appid < AID_APP) {
        for (n = 0; n < android_id_count; n++) {
            if (android_ids[n].aid == appid) {
                username = android_ids[n].name;
                break;
            }
        }
        ......
    } else if (appid < AID_ISOLATED_START) {
        username = "_app";
        ......
    } else {
        username = "_isolated";
        ......
    }

    ......

    for (i = 0; i < nspec; i++) {
        cur = seapp_contexts[i];

        if (cur->isSystemServer != isSystemServer)
            continue;

        if (cur->user) {
            if (cur->prefix) {
                if (strncasecmp(username, cur->user, cur->len-1))
                    continue;
            } else {
                if (strcasecmp(username, cur->user))
                    continue;
            }
        }

        if (cur->seinfo) {
            if (!seinfo || strcasecmp(seinfo, cur->seinfo))
                continue;
        }

        if (cur->name) {
            if (!pkgname || strcasecmp(pkgname, cur->name))
                continue;
        }

        if (kind == SEAPP_TYPE && !cur->type)
            continue;
        else if (kind == SEAPP_DOMAIN && !cur->domain)
```

```
            continue;

        if (cur->sebool) {
            int value = security_get_boolean_active(cur->sebool);
            if (value == 0)
                continue;
            ......
        }

        if (kind == SEAPP_TYPE) {
            if (context_type_set(ctx, cur->type))
                goto oom;
        } else if (kind == SEAPP_DOMAIN) {
            if (context_type_set(ctx, cur->domain))
                goto oom;
        }

        ......

        break;
    }

    ......

    return 0;

    ......
}
```

函数 seapp_context_lookup 首先根据参数 uid 来确定接下来查找中要用到的关键字 username，这对应于 seapp_contexts 文件中的 user 字段。

Android 从 4.2 开始支持多用户。通过支持多用户功能，在安装应用程序时分配到的 uid 由两部分组成，分别是 User Id 和 App Id。常量 AID_USER 的值定义为 100000，用 uid 除以该值获得的整数就为 User Id，而用 uid 对该值进行取模得到的就为 App Id，不同结果的具体说明如下所示。

- App Id 小于 AID_APP(10000)：这是保留给系统使用的，它们对应的 username 保存在数组 android_ids 中，具体可以参考文件 system/core/include/private/android_filesystem_config.h。
- App Id 大于等于 AID_APP(10000)并且小于 AID_ISOLATED_START(99000)：这是分配给应用程序使用的，它们对应的 username 统一设置为“_app”。
- App Id 大于等于 AID_ISOLATED_START(99000)：这是分配给运行在独立沙箱（没有任何自己的 Permission）的应用程序使用的，它们对应的 username 统一设置为“_isolated”。

接下来在全局变量 seapp_contexts 中，通过 username、isSystemServer、seinfo 和 pkgname 查找关键字寻找目标文件的 Type 或者目标进程的 Domain，这是由参数 kind 的值决定的。如果 kind 的值等于 SEAPP_TYPE，那么就表明要寻找的是文件的 Type。如果 kind 的值等于 SEAPP_DOMAIN，那么就表明要查找的是文件的 Domain。在这个场景中，我们要查找的是文件的 Type。在接下来的一篇文章分析进程的安全上下文时，我们就会通过函数 seapp_context_lookup 查找目标进程的 Domain。

通过遍历保存在数组 seapp_contexts 中的每一个 seapp_context，如果该 seapp_context 定义了相应的字段，那么就要求它与对应的查找关键字相等。如果某一个字段与对应的查找关键字不相等，就说明当前遍历的 seapp_context 不匹配，要继续下一个 seapp_context 的查找。有一点需要特别注意的是，如果当前正在遍历的 seapp_context 的某一个字段没有定义，那么它就默认匹配对应的查找关键字。

如果查找到了匹配的 seapp_context，那么就将它的 Type 或者 Domain 字段值提取出来，并且设置到参数 ctx 所描述的安全上下文中去，以便返回给调用者使用。

我们再来看函数 setfilecon，此函数在文件 external/libselinux/src/setfilecon.c 中定义，具体实现代码如下所示。

```
int setfilecon(const char *path, const security_context_t context)
    {
```

```
        return setxattr(path, XATTR_NAME_SELINUX, context, strlen(context) + 1,
                0);
    }
```

参数 path 描述的就是要设置安全上下文的目录或者文件，而参数 context 描述的就是要设置的安全上下文。

函数 setfilecon 实际上是通过调用系统接口 setxattr，将参数 context 描述的安全上下文设置为参数 path 描述的目录或者文件名称为 XATTR_NAME_SELINUX（“security.selinux”）的扩展属性中。也就是说，在 SEAndroid 中，目录或者文件的安全上下文其实是保存在名称为 security.selinux 扩展属性中的。这要求对应的文件系统支持扩展属性。

19.3 进程安全上下文

在 SEAndroid 系统中，除了要给文件关联安全上下文外，还需要给进程关联安全上下文，因为只有当进程和文件都关联安全上下文之后，SEAndroid 安全策略才能发挥作用。也就是说，当一个进程想访问一个文件时，SEAndroid 会提取出进程和文件的安全上下文，根据安全策略规则决定是否允许访问。在 Linux 系统中，当一个可执行文件加载到一个进程中执行时，该进程的安全上下文就设置为指定的值。也就是说，可以在安全策略中静态地为进程设置安全上下文。但是，这种进程安全上下文设置方式不适合于 Android 系统中的应用程序进程。在 Android 系统中，应用程序进程都是由 Zygote 进程 fork 出来。这些应用程序进程被 Zygote 进程 fork 出来之后，会通过 exec 系统调用将对应的可执行文件加载起来执行，这不像传统 Linux 的应用程序进程一样。这样就会使得 Zygote 进程及其创建的所有应用程序进程对应的可执行文件均为/system/bin/app_process。因为需要给不同的应用程序设置不同的安全上下文，以便给它们赋予不同的安全权限，所以，需要在应用程序进程创建出来之后动态地设置它的安全上下文。

19.3.1 为独立进程静态地设置安全上下文

Android 系统的第一个进程是 init，其他所有的进程都是由 init 进程直接或者间接 fork 出来的。一个新创建的文件的安全上下文在默认情况下来自于其父目录。与此类似，一个新创建的进程的安全上下文在默认情况下来自于其父进程。因此，我们就先看看系统中的第一个进程 init 的安全上下文是如何设置的。

查看 init 进程的启动脚本 system/core/rootdir/init.rc，具体内容如下所示。

```
on early-init
    ......

    # Set the security context for the init process.
    # This should occur before anything else (e.g. ueventd) is started.
    setcon u:r:init:s0

    ......
```

这段脚本的功能是，在启动 init 进程后立即调用函数 setcon，将自己的安全上下文设置为“u:r:init:s0”，即将 init 进程的 domain 指定为 init。

接下来再看看 init 这个 domain 的定义，在 external/sepolicy/init.te 文件中实现，具体内容如下所示。

```
# init switches to init domain (via init.rc).
type init, domain;
permissive init;
# init is unconfined.
unconfined_domain(init)
tmpfs_domain(init)
# add a rule to handle unlabelled mounts
allow init unlabeled:filesystem mount;
```

- 第一个 type 语句将 domain 设置为 init 的属性，这意味着 init 是用来描述进程的安全上下

文的。

- 第二个 permissive 语句指定当 domain 为 init 的进程违反 SEAndroid 安全策略访问资源时，只进行日志输出，而不是拒绝执行。由于这里列出来的内容来自 Android 5.0，而 Android 5.0 开启的是 Permissive 的 SEAndroid 模式，因此，这里会看到这样的一个 permissive 语句。
- 第三个 unconfined_domain 语句是一个宏，定义在 external/sepolicy/te_macros 文件中，用来指定 init 是一个不受限制的 domain，即它可以访问系统中的大部分资源。具体内容如下所示。

```
####################################
# unconfined_domain(domain)
# Allow the specified domain to do anything.
#
define(`unconfined_domain', `
typeattribute $1 mlstrustedsubject;
typeattribute $1 unconfineddomain;
')
```

- 第四个 tmpfs_domain 语句也是定义在 external/sepolicy/te_macros 文件中的一个宏，用来指定当 domain 为 init 的进程在 type 为 tmpfs 的目录中创建文件时，将新创建的文件的 type 设置为 init_tmpfs，并且允许 domain 为 init 的进程对它们进行读和执行。具体内容如下所示。

```
####################################
# tmpfs_domain(domain)
# Define and allow access to a unique type for
# this domain when creating tmpfs / shmem / ashmem files.
define(`tmpfs_domain', `
type $1_tmpfs, file_type;
type_transition $1 tmpfs:file $1_tmpfs;
# Map with PROT_EXEC.
allow $1 $1_tmpfs:file { read execute execmod };
')
```

上述第 5 个 allow 语句允许 domain 为 init 的进程 mount 未指定安全上下文的文件系统时，将其安全上下文设置为 unlabeled。

上面列出的脚本就指明了 init 进程的安全上下文，以及它所具有的 SEAndroid 权限。接下来就再来看看负责创建应用程序进程的 Zygote 进程的安全上下文的设置过程。

Zygote 进程是由 init 进程创建的，它的启动命令在文件 system/core/rootdir/init.rc 中定义，具体内容如下所示。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

这表示 Zygote 进程对应的可执行文件为/system/bin/app_process。

通过检查 external/sepolicy/file_contexts，可以发现文件/system/bin/app_process 的安全上下文为“u:object_r:zygote_exec:s0”，具体内容如下所示。

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-
server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

这表示 Zygote 进程对应的可执行文件为/system/bin/app_process。

通过检查 external/sepolicy/file_contexts，可以发现文件/system/bin/app_process 的安全上下文为“u:object_r:zygote_exec:s0”，具体内容如下所示。

```
# zygote
type zygote, domain;
type zygote_exec, exec_type, file_type;

permissive zygote;
init_daemon_domain(zygote)
unconfined_domain(zygote)
```

对上述内容的具体说明如下所示。

- 第一个 type 语句将 domain 设置为 zygote 的属性，表明 zygote 是用来描述进程的安全上下文的。
- 第二个 type 语句将 exec_type 和 file_type 设置为 zygote_exec 的属性，表明 zygote_exec 是用来描述可执行文件的安全上下文的。
- 第三个 permissive 语句同样是表明当 domain 为 zygote 的进程违反 SEAndroid 安全策略访问资源时，只进行日志输出，而不是拒绝执行。
- 第四个 init_daemon_domain 语句是一个宏，定义在文件 external/sepolicy/te_macros 中，用来设置 zygote 这个 domain 的权限，具体内容如下所示。

```
######################################
# init_daemon_domain(domain)
# Set up a transition from init to the daemon domain
# upon executing its binary.
define(`init_daemon_domain', `
domain_auto_trans(init, $1_exec, $1)
tmpfs_domain($1)
')
```

宏 init_daemon_domain 由另外两个宏 tmpfs_domain 和 domain_auto_trans 组成，其中宏 domain_auto_trans 在文件 external/sepolicy/te_macros 中定义，具体内容如下所示。

```
######################################
# domain_auto_trans(olddomain, type, newdomain)
# Automatically transition from olddomain to newdomain
# upon executing a file labeled with type.
#
define(`domain_auto_trans', `
# Allow the necessary permissions.
domain_trans($1,$2,$3)
# Make the transition occur by default.
type_transition $1 $2:process $3;
')
```

- 第二个 type_transition 语句指定当一个 domain 为 init 的进程创建一个子进程执行一个 type 为 zygote_exec 的文件时，将该子进程的 domain 设置为 zygote，而不是继承父进程的 domain。
- 第一个 domain_trans 语句是一个宏，也是定义在 external/sepolicy/te_macros 中，用来允许进程的 domain 从 init 修改为 zygote，它的定义如下所示。

```
######################################
# domain_auto_trans(olddomain, type, newdomain)
# Automatically transition from olddomain to newdomain
# upon executing a file labeled with type.
#
define(`domain_auto_trans', `
# Allow the necessary permissions.
domain_trans($1,$2,$3)
# Make the transition occur by default.
type_transition $1 $2:process $3;
')
```

- 第二个 type_transition 语句指定当一个 domain 为 init 的进程创建一个子进程执行一个 type 为 zygote_exec 的文件时，将该子进程的 domain 设置为 zygote，而不是继承父进程的 domain。
- 第一个 domain_trans 语句是一个宏，在 external/sepolicy/te_macros 中定义，用来允许进程的 domain 从 init 修改为 zygote，具体内容如下所示。

```
allow $1 $3:process transition;
```

```
allow $3 $2:file { entrypoint read execute };
```

- 第一个 allow 语句允许 domain 为 init 的进程将 domain 修改为 zygote。
- 第二个 allow 语句允许 type 为 zygote_exec 的可执行文件作为进入 zygote 这个 domain 的入口点。

在文件 external/sepolicy/zygote.te 中，通过 init_daemon_domain 设置了 Zygote 进程的 domain 为 zygote。通过 Zygote 进程的创建过程可以了解安全策略，具体过程如下所示。

（1）首先，zygote 进程是由 init 进程 fork 出来的。在 fork 出来的时候，zygote 进程的 domain 来自于父进程 init 的 domain，即此时 zygote 进程的 domain 为 init。

（2）然后刚刚 fork 出来的 zygote 进程，会通过系统接口 exec 将文件/system/bin/app_process 加载进来执行。由于 allow 和 type_transition 规则的存在，使得文件/system/bin/app_process 被 exec 函数刚刚 fork（孕育）出来的 zygote 进程的时候，它的 domain 自动地从 init 转换为 zygote。这样就可以给 init 进程和 zygote 进程设置不同的 domain，以便可以给它们赋予不同的 SEAndroid 安全权限。

（3）回到文件 external/sepolicy/zygote.te 中，最后一个 unconfined_domain 语句同样是将 zygote 这个 domain 设置为一个不受限的 domain，以便它可以访问系统中的大部分资源。

19.3.2 为应用程序进程设置安全上下文

应用程序进程是由 ActivityManagerService 请求 Zygote 进程创建的。ActivityManagerService 在请求 Zygote 进程创建应用程序进程时会传递很多参数，例如，应用程序在安装时分配到的 UID 和 GID。通过使用 SEAndroid 安全机制之后，ActivityManagerService 传递给 Zygote 进程的参数包含了一个 seinfo。这个 seinfo 与 19.2 节中介绍的 seinfo 是一样的，不过它的作用是用来设置应用程序进程的安全上下文，而不是设置应用程序数据文件的安全上下文。接下来就分析应用程序进程的安全上下文设置过程。

当 ActivityMangerService 需要创建应用程序进程的时候，就会调用类 ActivityMangerService 中的成员函数 startProcessLocked，这个函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，具体实现代码如下所示。

```
public final class ActivityManagerService  extends ActivityManagerNative
        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
    ......

    private final void startProcessLocked(ProcessRecord app,
            String hostingType, String hostingNameStr) {
        ......

        try {
            ......

            // Start the process.  It will either succeed and return a result containing
            // the PID of the new process, or else throw a RuntimeException.
            Process.ProcessStartResult startResult = Process.start("android.app.
            ActivityThread",
                    app.processName, uid, uid, gids, debugFlags, mountExternal,
                    app.info.targetSdkVersion, app.info.seinfo, null);

            ......
        } catch (RuntimeException e) {
            ......
        }
    }

    ......
}
```

在上述代码中，参数 app 指向的是一个 ProcessRecord 对象，用来描述正在创建的应用程序进程。其中，它的成员变量 info 指向的是一个 ApplicationInfo 对象。从前面 SEAndroid 安全机制中

的文件安全上下文关联分析一文可以知道，这个 ApplicationInfo 对象有一个类型为 String 的成员变量 seinfo，是在应用程序安装的时候通过解析文件 mac_permissions.xml 获得的。

再看类 ActivityManagerService 中的成员函数 startProcessLocked，功能是通过调用 Process 类的静态成员函数 start 来创建应用程序进程，其中就包含了要创建的应用程序进程的各种参数，这些参数会通过 Socket IPC 传递给 Zygote 进程。最后，Zygote 进程会通过调用 ZygoteConnection 类的成员函数 runOnce 来执行创建应用程序进程的工作。

ZygoteConnection 类的成员函数 runOnce 在文件 frameworks/base/core/java/com/android/internal/os/ZygoteConnection.java 中定义，具体实现过程如下所示。

- 首先通过调用函数 readArgumentList 读取 ActivityManagerService 发送过来的应用程序创建参数 args。
- 然后再创建一个 Arguments 对象来解析该参数。
- 将解析后得到的参数传递给 Zygote 类的静态成员函数 forkAndSpecialize，以便后者可以执行创建应用程序进程的工作。

函数 runOnce 的具体实现代码如下所示。

```
class ZygoteConnection {
    ......

    boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
        ......

        try {
            args = readArgumentList();
            ......
        } catch (IOException ex) {
            ......
        }

        ......

        try {
            parsedArgs = new Arguments(args);
            ......

            pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid, parsedArgs.gids,
                    parsedArgs.debugFlags,rlimits,parsedArgs.mountExternal,parsedArgs.seInfo,
                    parsedArgs.niceName);
        } catch (IOException ex) {
            ......
        } catch (ErrnoException ex) {
            ......
        } catch (IllegalArgumentException ex) {
            ......
        } catch (ZygoteSecurityException ex) {
            ......
        }

    ......
    }

    ......
}
```

类 Zygote 中的静态成员函数 forkAndSpecialize 在文件 libcore/dalvik/src/main/java/dalvik/system/Zygote.java 中定义，具体实现代码如下所示。

```
public class Zygote {
    ......

    public static int forkAndSpecialize(int uid, int gid, int[] gids, int debugFlags,
            int[][] rlimits, int mountExternal, String seInfo, String niceName) {
        preFork();
        int pid = nativeForkAndSpecialize(
                uid, gid, gids, debugFlags, rlimits, mountExternal, seInfo, niceName);
```

```
        postFork();
        return pid;
    }

    native public static int nativeForkAndSpecialize(int uid, int gid, int[] gids, int
debugFlags,
            int[][] rlimits, int mountExternal, String seInfo, String niceName);

    ......
}
```

在上述代码中，通过调用 JNI 函数 nativeForkAndSpecialize 来执行创建应用程序进程的工作。

类 Zygote 中的 JNI 函数 nativeForkAndSpecialize 由 C++层的函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 来实现，此函数在文件 dalvik/vm/native/dalvik_system_Zygote.cpp 中定义，具体实现代码如下所示。

```
static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4* args,
    JValue* pResult)
{
    pid_t pid;

    pid = forkAndSpecializeCommon(args, false);

    RETURN_INT(pid);
}
```

JNI 函数 nativeForkAndSpecialize 在调用的过程中，传递进来的参数都被保存在函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 的参数 args 指向的一块内存中。

函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 通过调用函数 forkAndSpecializeCommon 来执行创建应用程序进程的工作，此函数在文件 dalvik/vm/native/dalvik_system_Zygote.cpp 中定义，具体实现代码如下所示。

```
static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4* args,
    JValue* pResult)
{
    pid_t pid;

    pid = forkAndSpecializeCommon(args, false);

    RETURN_INT(pid);
}
```

函数 nativeForkAndSpecialize 在调用的过程中，传递进来的参数都被保存在函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 的参数 args 指向的一块内存中。

函数 Dalvik_dalvik_system_Zygote_forkAndSpecialize 通过调用函数 forkAndSpecializeCommon 来执行创建应用程序进程的工作，此函数在文件 dalvik/vm/native/dalvik_system_Zygote.cpp 中定义，具体实现代码如下所示。

```
static int setSELinuxContext(uid_t uid, bool isSystemServer,
                             const char *seInfo, const char *niceName)
{
#ifdef HAVE_ANDROID_OS
    return selinux_android_setcontext(uid, isSystemServer, seInfo, niceName);
#else
    return 0;
#endif
}
```

函数 setSELinuxContext 的功能是，通过调用 libselinux 提供的函数 selinux_android_setcontext 来设置刚刚创建出来的应用程序进程的安全上下文。

函数 selinux_android_setcontext 在文件 external/libselinux/src/android.c 中定义，具体实现代码如下所示。

```
int selinux_android_setcontext(uid_t uid,
                               int isSystemServer,
```

```
                const char *seinfo,
                const char *pkgname)
{
    char *orig_ctx_str = NULL, *ctx_str;
    context_t ctx = NULL;
    int rc;

    if (is_selinux_enabled() <= 0)
        return 0;

    __selinux_once(once, seapp_context_init);

    rc = getcon(&ctx_str);
    ......

    ctx = context_new(ctx_str);
    orig_ctx_str = ctx_str;
    ......

    rc = seapp_context_lookup(SEAPP_DOMAIN, uid, isSystemServer, seinfo, pkgname, ctx);
    ......

    ctx_str = context_str(ctx);
    ......

    rc = security_check_context(ctx_str);
    ......

    if (strcmp(ctx_str, orig_ctx_str)) {
        rc = setcon(ctx_str);
        ......
    }

    rc = 0;
out:
    ......
    return rc;

    ......
}
```

函数 selinux_android_setcontext 的执行过程如下所示。

- 调用函数 is_selinux_enabled 检查系统是否启用了 SELinux。如果没有启用的话，那就什么也不用做就返回；否则的话，就继续往下执行。
- 调用函数 seapp_context_init 读取和解析在前面 SEAndroid 安全机制框架分析中提到的 seapp_contexts 文件。注意，__selinux_once 是一个宏，它实际上是利用了 pthread 库提供的函数 pthread_once 保证函数 seapp_context_init 在进程内有且仅有一次会被调用到，适合用来执行初始化工作。
- 调用函数 getcon 获得当前进程的安全上下文，保存在变量 ctx_str。
- 调用函数 context_new 在原来的安全上下文的基础上创建一个新的安全上下文 ctx，以便可以获得原来安全上下文的 SELinux 用户、角色和安全级别等信息。
- 调用函数 seapp_context_lookup；根据传进来的参数 seinfo 在 seapp_contexts 文件中找到对应的 Domain，并且将其设置为新的安全上下文 ctr 的 Domain。
- 调用函数 context_str 获得新创建的安全上下文的字符串描述，以便可以调用函数 ecurity_check_context 来验证新创建的安全上下文的正确性。如果不正确的话，就出错返回。否则的话，继续往下执行。
- 比较原来的安全上下文和新创建的安全上下文。如果不一致的话，那么就调用函数 setcon 将当前进程的安全上下文设置为新创建的安全上下文。

函数 setcon 在文件 external/libselinux/src/procattr.c 中定义，通过如下所示的 3 个宏来定义。

```
#define setselfattr_def(fn, attr) \
    int set##fn(const security_context_t c) \
```

```
    { \
        return setprocattrcon(c, 0, #attr); \
    }

#define all_selfattr_def(fn, attr) \
    getselfattr_def(fn, attr)    \
    setselfattr_def(fn, attr)

all_selfattr_def(con, current)
```

从上述3个宏的定义中可以看出，函数setcon最终通过调用函数setprocattrcon来设置当前进程的安全上下文，其中，第一个参数c描述的是要设置的安全上下文，第三个参数的值等于“current”。

函数setprocattrcon在文件external/libselinux/src/procattr.c中定义，功能是打开proc文件系统中的/proc/self/task/<tid>/attr/current文件，并且向其写入参数context所描述的安全上下文。其中，<tid>描述的是当前线程的id。向/proc/self/task/<tid>/attr/current文件写入安全上下文实际上就是，将进程的安全上下文保存在内核中用来描述进程的结构体task_struct中。这与文件的安全上下文是保存在文件扩展属性中是不一样的。函数setprocattrcon的具体实现代码如下所示。

```
static int setprocattrcon(security_context_t context,
            pid_t pid, const char *attr)
{
    char *path;
    int fd, rc;
    pid_t tid;
    ssize_t ret;
    int errno_hold;

    if (pid > 0)
        rc = asprintf(&path, "/proc/%d/attr/%s", pid, attr);
    else {
        tid = gettid();
        rc = asprintf(&path, "/proc/self/task/%d/attr/%s", tid, attr);
    }
    if (rc < 0)
        return -1;

    fd = open(path, O_RDWR);
    free(path);
    if (fd < 0)
        return -1;
    if (context)
        do {
            ret = write(fd, context, strlen(context) + 1);
        } while (ret < 0 && errno == EINTR);
    else
        do {
            ret = write(fd, NULL, 0);   /* clear */
        } while (ret < 0 && errno == EINTR);
    errno_hold = errno;
    close(fd);
    errno = errno_hold;
    if (ret < 0)
        return -1;
    else
        return 0;
}
```

这样，新创建的应用程序进程的安全上下文就设置好了，但是，究竟如何将设备上的/system/bin/gpsd文件下载到PC上呢？通过adb pull命令无法将开启了SEAndroid安全机制的设备上的/system/bin/gpsd文件读取下来。这是因为adb pull命令是通过运行在设备上的adbd守护进程来读取/system/bin/gpsd文件，并且传回给PC的，而守护进程adbd没有权限读取文件/system/bin/gpsd。

通过如下所示的ls -Z和ps -Z命令，可以来观察/system/bin/gpsd文件和adbd守护进程的安全上下文。

```
./adb shell ls -Z /system/bin/gpsd
-rwxr-xr-x root     shell             u:object_r:gpsd_exec:s0 gpsd
$ ./adb shell ps -Z | grep 'adbd'
u:r:adbd:s0                  shell     1978  1     /sbin/adbd
```

这说明 domain 为 adbd 的进程无法读取 type 为 gpsd_exec 的文件。

其实可以在 PC 上通过执行如下所示的 adb shell cat 命令，来读取设备上的/system/bin/gpsd 文件。

```
./adb shell cat /system/bin/gpsd > ./gpsd
```

上面的命令实际上是通过 shell 启动 cat 程序，并且通过这个 cat 进程来读取文件/system/bin/gpsd 的内容。

因为 cat 进程的安全上下文来自于其父进程 shell，所以，通过在设备上执行 ps -Z 命令观察一下 shell 进程的安全上下文。

```
$ ps -Z
u:r:shell:s0                 shell     23486 1978  /system/bin/sh
```

这说明 domain 为 shell 的进程可以读取 type 为 gpsd_exec 的文件。

第 20 章　分析 ART 系统

从 Android 5.0 版本开始，Android 应用程序默认的运行环境为 ART。ART 机制与 Dalvik 不同，在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT，Ahead-Of-Time）。这样的话，应用的启动（首次）和执行都会变得更加快速。在本章的内容中，将简要介绍 Android ART 机制的基本架构知识。

20.1 对比 Dalvik VM 和 ART

传统的 Dalvik 虚拟机其实是一个 Java 虚拟机，只不过它执行的不是 class 文件，而是 dex 文件。因此，ART 运行时最理想的方式也是实现为一个 Java 虚拟机的形式，这样就可以很容易地将 Dalvik 虚拟机替换掉。ART 运行时就是真地和 Dalvik 虚拟机一样，实现了一套完全兼容 Java 虚拟机的接口。为了方便描述，接下来就将 ART 运行时称为 ART 虚拟机，它和 Dalvik 虚拟机、Java 虚拟机的关系如图 20-1 所示。

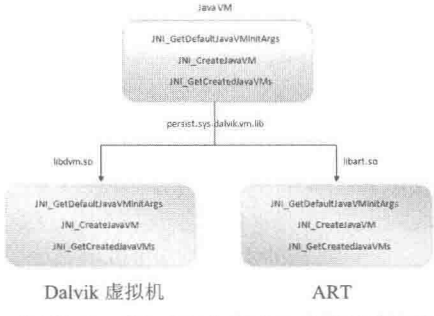
从图 20-1 可知，Dalvik 虚拟机和 ART 虚拟机都实现了如下 3 个用来抽象 Java 虚拟机的接口。

（1）JNI_GetDefaultJavaVMInitArgs：获取虚拟机的默认初始化参数。

（2）JNI_CreateJavaVM：在进程中创建虚拟机实例。

（3）JNI_GetCreatedJavaVMs：获取进程中创建的虚拟机实例。

在 Android 系统中，Davik 虚拟机实现在 libdvm.so 文件中，ART 虚拟机实现在 libart.so 文件中。也就是说，文件 libdvm.so 和文件 libart.so 导出了如下 3 个接口供外界调用。



▲图 20-1　ART、Dalvik 和 Java 虚拟机的关系

- JNI_GetDefaultJavaVMInitArgs。
- JNI_CreateJavaVM。
- JNI_GetCreatedJavaVMs。

此外，Android 系统还提供了一个系统属性 persist.sys.dalvik.vm.lib，其值等于 libdvm.s 或 libart.so，具体说明如下所示。

- 当等于 libdvm.so 时，表示当前用的是 Dalvik 虚拟机。
- 当等于 libart.so 时，表示当前用的是 ART 虚拟机。

上面介绍了 Dalvik 虚拟机和 ART 虚拟机的共同之处，当然，它们之间有显著不同。Dalvik 虚拟机执行的是 dex 字节码，ART 虚拟机执行的是本地机器码。这意味着 Dalvik 虚拟机包含有一个解释器，用来执行 dex 字节码。当然，Android 从 2.2 开始，也包含有 JIT（Just-In-Time），用来在运行时动态地将执行频率很高的 dex 字节码翻译成本地机器码，然后再执行。通过 JIT，就可以有效地提高 Dalvik 虚拟机的执行效率。但是，将 dex 字节码翻译成本地机器码是发生在应用程序的运行过程中的，并且应用程序每一次重新运行的时候，都要重做这个翻译工作。所以，即使 Dalvik 虚拟机采用了 JIT，其总体性能还是不能与直接执行本地机器码的 ART 虚拟机相比。

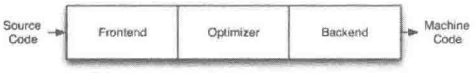
另外，ART 可以在不重新编译 APK 的基础上直接对其进行加载和运行，这是由于 APK 在安

装时被执行了 AOT。AOT（Ahead Of Time）是相对 JIT（Just In Time）而言的。也就是在 APK 运行之前，就对其包含的 dex 字节码进行翻译，得到对应的本地机器指令，于是就可以在运行时直接执行了。这种技术不但使得我们可以不对原有的 APK 作任何修改，还可以使得这些 APK 只需要在安装时翻译一次，就可以无数次以本地机器指令的形式运行。这种技术与我们用 C/C++语言编写一个程序，然后用 GCC 编译得到一个可执行程序，最后这个可执行程序就可以无数次地加载到系统执行是差不多的。

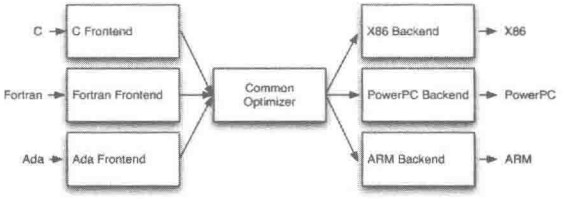
在 ART 机制中，打包在 APK 里面的 dex 字节码是通过 LLVM 翻译成本地机器指令的。LLVM 是一个用来快速开发自己的编译器的框架系统，其作者之一是 Chris Lattner（Apple 公司 Swift 语言的首席架构师）。LLVM 包含了如图 20-2 所示的 3 类组件。

其中，前端（Frontend）对输入的源代码（Source Code）进行语法分析后，生成一棵抽象语法树（Abstract Syntax Tree，AST），并且可以进一步将得到的抽象语法树转化一种称为 LLVM IR 的中间语言。LLVM IR 是一种与编程语言无关的中间语言，也就是说，不管是 C 语言，还是 Fortran、Ada 语言编写的源文件，经过语法分析后，最终都可以得到一个对应的 LLVM IR 文件。这个 LLVM IR 文件可以作为后面的优化器（Optimizer）和后端（Backend）的输入文件。优化器对 LLVM IR 文件进行优化，例如消除代码里面的冗余计算，以提高最终生成的代码的执行效率。后端负责生成最终的机器指令。

LLVM 的上述架构大大简化开发编译器的流程，因为开发者需要关注的仅仅是前端，然后就可以利用现成的优化器来进行代码优化，并且利用现成的后端生成各种体系结构相关的机器指令，如图 20-3 所示。



▲图 20-2　LLVM 组件



▲图 20-3　生成各种体系结构相关的机器指令

在图 20-3 中，利用现成的与语言无关的优化器和后端作为和语言相关的前端，生成了各种体系结构相关的机器指令。即分别为 C、Fortran 和 Ada 3 种语言开发 3 个不同的前端，然后利用现成的优化器对它们生成的 LLVM IR 语言进行优化，并且通过现成的后端生成 X86、PowerPC 和 ARM 3 种不同体系结构的机器指令。

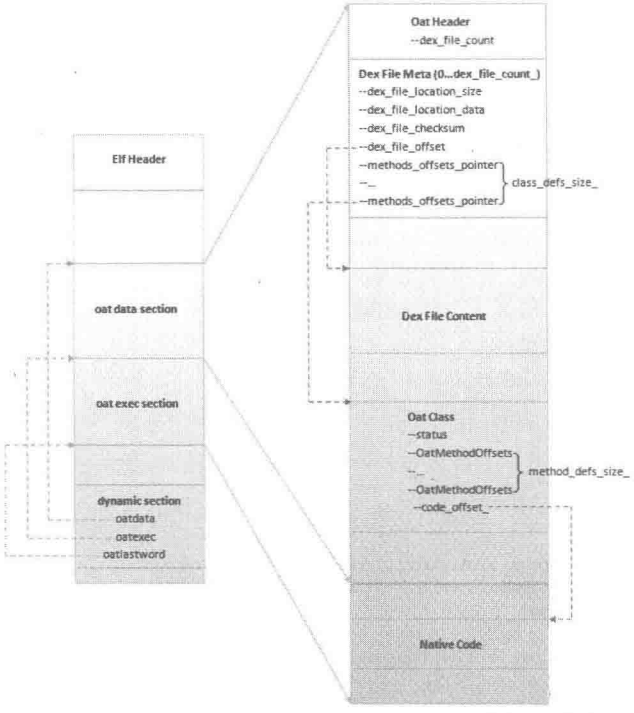
在 ART 运行过程中，安装服务 PackageManagerService 在安装 APK 时，会通过守护进程 installd 调用工具 dex2oat 翻译打包在 APK 里面的 dex 字节码。这个翻译器实际上就是基于 LLVM 架构实现的一个编译器，其前端是一个 dex 语法分析器。翻译后得到的是一个 ELF 格式的 oat 文件，这个 oat 文件同样是以“.odex”为后缀，并且也被保存在/data/dalvik-cache 目录中。ELF 是 Linux 系统使用的一种文件格式，我们平时接触的静态库、动态库和可执行文件都是以这种格式保存的，但是由 dexoat 工具生成的 oat 文件与上述 3 种文件都不一样，它有两个特殊的段 oatdata 和 oatexec，分别用来存储原来打包在 APK 里面的 dex 文件，和翻译这个 dex 文件里面的类方法得到本地机器指令，如图 20-4 所示。

在 oat 文件的动态段（dymanic section）中导出了 3 个符号，分别是 oatdata、oatexec 和 oatlastword，分别用来描述 oatdata 和 oatexec 段到内存后的起止地址。在 oatdata 段中包含了如下两个重要的信息。

- 一个信息是原来的 classes.dex 文件的完整内容。
- 一个信息引导 ART 找到 classes.dex 文件里面的类方法所对应的本地机器指令，这些本地机器指令就保存在 oatexec 段中。

假如在文件 classes.dex 中有一个类 A，当知道类 A 的名字后就可以通过保存在 oatdata 段的

dex 文件得到类 A 的所有信息，如 A 的父类、成员变量和成员函数等。另一方面，类 A 在 oatdata 段中有一个对应的 OatClass 结构体。这个 OatClass 结构体描述了类 A 的每一个方法所对应的本地机器指令在 oatexec 段的位置。也就是说，当知道一个类及其某一个方法的名字（签名）后，可以通过 oatdata 段的 dex 文件内容和 OatClass 结构体找到其在 oatexec 段的本地机器指令，这样就可以执行这个类方法了。



▲图 20-4 ART 翻译 classes.dex 后得到的 ELF 格式的 oat 文件

此时可以总结出 ART 的运行原理，具体说明如下所示。

- 在 Android 系统启动过程中创建的 Zygote 进程利用 ART 运行时导出的 Java 虚拟机接口创建 ART 虚拟机。
- APK 在安装的时候，打包在里面的 classes.dex 文件会被工具 dex2oat 翻译成本地机器指令，最终得到一个 ELF 格式的 oat 文件。
- APK 运行时，上述生成的 oat 文件会被加载到内存中，并且 ART 虚拟机可以通过里面的 oatdata 和 oatexec 段，找到任意一个类方法对应的本地机器指令来执行。

20.2 启动 ART

在启动 Android 系统时，会创建一个 Zygote 进程作为应用程序进程孵化器。在启动 Zygote 进程的过程中会创建一个 Dalvik 虚拟机，Zygote 进程是通过复制自己来创建新的应用程序进程的，这说明 Zygote 进程会将自己的 Dalvik 虚拟机复制给应用程序进程。通过这种方式可以大大地提高应用程序的启动速度，因为这种方式避免了在启动每一个应用程序进程时都要创建一个 Dalvik 的情形发生。其实 Zygote 进程通过自我复制的方式来创建应用程序进程，省去的不仅仅是应用程序进程创建 Dalvik 虚拟机的时间，还能省去应用程序进程加载各种系统库和系统资源的时间，因为它们在 Zygote 进程中已经加载过了，并且也会连同 Dalvik 虚拟机一起复制到应用程序进程中去。在本节的内容中，将详细讲解启动 ART 的具体过程。

20.2.1　运行 app_process 进程

启动过程从 init.rc 文件开始，在文件 init.rc 中由这一行表示启动 zygote。

```
service zygote/system/bin/app_process-Xzygote/system/bin--zygote--start-system-server
```

init 进程根据它执行 app_process（frameworks/base/cmds/app_process/app_main.cpp），也就是 Zygote 了。当 Android 系统启动时会创建一个 Zygote 进程，作为应用程序的进程孵化器，并且在启动 Zygote 进程的过程中会创建一个 Dalvik 虚拟机。Zygote 进程是通过复制自己来创建新的应用程序进程的，这意味着 Zygote 进程会将自己的 Dalvik 虚拟机复制给应用程序进程。上述方式可以大大地提高应用程序的启动速度，因为这种方式避免了每一个应用程序进程在启动的时候都要去创建一个 Dalvik。事实上，Zygote 进程通过自我复制的方式来创建应用程序进程，省去的不仅仅是应用程序进程创建 Dalvik 虚拟机的时间，还能省去应用程序进程加载各种系统库和系统资源的时间，因为它们在 Zygote 进程中已经加载过了，并且也会连同 Dalvik 虚拟机一起复制到应用程序进程中去。这也就是 ART 优于 Dalvik 的原因所在。

当 Android 系统启动 init 进程时会运行 app_process 进程，在文件/frameworks/base/cmds/app_process/app_main.cpp 中定义了 app_process 进程的具体实现，在主函数 main 中会启动 Zygote，对应代码如下所示的加粗部分。

```
    if (niceName && *niceName) {
        setArgv0(argv0, niceName);
        set_process_name(niceName);
    }

    runtime.mParentDir = parentDir;

    if (zygote) {
        runtime.start("com.android.internal.os.ZygoteInit",
                startSystemServer ? "start-system-server" : "");
    } else if (className) {
        // Remainder of args get passed to startup class main()
        runtime.mClassName = className;
        runtime.mArgC = argc - i;
        runtime.mArgV = argv + i;
        runtime.start("com.android.internal.os.RuntimeInit",
                application ? "application" : "tool");
    } else {
        fprintf(stderr, "Error: no class name or --zygote supplied.\n");
        app_usage();
        LOG_ALWAYS_FATAL("app_process: no class name or --zygote supplied.");
        return 10;
    }
}
```

在上述代码中，runtime 是 AppRuntime 的实例，AppRuntime 继承自 AndroidRuntime。类 AndroidRuntime 中的函数 start 在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中定义，具体实现代码如下所示。

```
void AndroidRuntime::start(const char* className, const char* options)
{
    ALOGD("\n>>>>>> AndroidRuntime START %s <<<<<<\n",
          className != NULL ? className : "(unknown)");

    /*
     * 'startSystemServer == true' means runtime is obsolete and not run from
     * init.rc anymore, so we print out the boot start event here.
     */
    if (strcmp(options, "start-system-server") == 0) {
        /* track our progress through the boot sequence */
        const int LOG_BOOT_PROGRESS_START = 3000;
        LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)));
    }

    const char* rootDir = getenv("ANDROID_ROOT");
```

```
    if (rootDir == NULL) {
        rootDir = "/system";
        if (!hasDir("/system")) {
            LOG_FATAL("No root directory specified, and /android does not exist.");
            return;
        }
        setenv("ANDROID_ROOT", rootDir, 1);
    }

    //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
    //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);

    /* start the virtual machine */
    JniInvocation jni_invocation;
    jni_invocation.Init(NULL);
    JNIEnv* env;
    if (startVm(&mJavaVM, &env) != 0) {
        return;
    }
    onVmCreated(env);

    /*
     * Register android functions.
     */
    if (startReg(env) < 0) {
        ALOGE("Unable to register all android natives\n");
        return;
    }

    /*
     * We want to call main() with a String array with arguments in it.
     * At present we have two arguments, the class name and an option string.
     * Create an array to hold them.
     */
    jclass stringClass;
    jobjectArray strArray;
    jstring classNameStr;
    jstring optionsStr;

    stringClass = env->FindClass("java/lang/String");
    assert(stringClass != NULL);
    strArray = env->NewObjectArray(2, stringClass, NULL);
    assert(strArray != NULL);
    classNameStr = env->NewStringUTF(className);
    assert(classNameStr != NULL);
    env->SetObjectArrayElement(strArray, 0, classNameStr);
    optionsStr = env->NewStringUTF(options);
    env->SetObjectArrayElement(strArray, 1, optionsStr);

    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits.
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
```

```
        }
    }
    free(slashClassName);

    ALOGD("Shutting down VM\n");
    if (mJavaVM->DetachCurrentThread() != JNI_OK)
        ALOGW("Warning: unable to detach main thread\n");
    if (mJavaVM->DestroyJavaVM() != 0)
        ALOGW("Warning: VM did not shut down cleanly\n");
}
```

在上述代码中，“JniInvocation jni_invocation”用于声明类 JniInvocation 的变量，“jni_invocation.Init(NULL)”用于调用类 JniInvocation 中的函数 init。由此可见，类 AndroidRutime 的成员函数 start 最主要实现了如下所示的 3 个功能。

- 创建一个 JniInvocation 实例，并且调用它的成员函数 init 来初始化 JNI 环境。
- 调用 AndroidRutime 类的成员函数 startVm 来创建一个虚拟机及其对应的 JNI 接口，即创建一个 JavaVM 接口和一个 JNIEnv 接口。
- 通过上述 JavaVM 接口和 JNIEnv 接口在 Zygote 进程中加载指定的 class。

其中，上述第 1 个功能和第 2 个功能又是最关键的。因此，接下来继续分析它们所对应的函数的实现。

类 JniInvocation 在文件“/libnativehelper/JniInvocation.cpp”中定义，函数 init 的具体实现代码如下所示。

```
bool JniInvocation::Init(const char* library) {
#ifdef HAVE_ANDROID_OS
  char default_library[PROPERTY_VALUE_MAX];
  property_get("persist.sys.dalvik.vm.lib", default_library, "libdvm.so");
#else
  const char* default_library = "libdvm.so";
#endif
  if (library == NULL) {
    library = default_library;
  }

  handle_ = dlopen(library, RTLD_NOW);
  if (handle_ == NULL) {
    ALOGE("Failed to dlopen %s: %s", library, dlerror());
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetDefaultJavaVMInitArgs_),
                  "JNI_GetDefaultJavaVMInitArgs")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_CreateJavaVM_),
                  "JNI_CreateJavaVM")) {
    return false;
  }
  if (!FindSymbol(reinterpret_cast<void**>(&JNI_GetCreatedJavaVMs_),
                  "JNI_GetCreatedJavaVMs")) {
    return false;
  }
  return true;
}
```

在上述代码中，函数 init 首先读取系统属性 persist.sys.dalvik.vm.lib 的值。因为系统属性 persist.sys.dalvik.vm.lib 的值要么等于 libdvm.so，要么等于 libart.so。所以，接下来通过函数 dlopen 加载到进程来是如下两者之一。

- libdvm.so。
- libart.so。

无论加载的是哪一个，都要求它导出 JNI_GetDefaultJavaVMInitArgs、JNI_CreateJavaVM 和 JNI_GetCreatedJavaVMs 这 3 个接口，并且分别保存在 JniInvocation 类的 3 个成员变量 JNI_GetDefaultJavaVMInitArgs_、JNI_CreateJavaVM_和 JNI_GetCreatedJavaVMs_中，这 3 个接口也就是

前面提到的用来抽象 Java 虚拟机的 3 个接口。

20.2.2 准备启动

回到函数 AndroidRuntime::start，“if (startVm(&mJavaVM, &env) != 0) {”用于调用函数 startVm 启动虚拟机。也就是说，类 JniInvocation 的成员函数 init 实际上就是根据系统属性 persist.sys.dalvik.vm.lib 来初始化 Dalvik 虚拟机或者 ART 虚拟机环境。类 AndroidRutime 的成员函数 AndroidRuntime::startVm 的具体实现代码如下所示。

```
int AndroidRuntime::startVm(JavaVM** pJavaVM, JNIEnv** pEnv)
{
    int result = -1;
    JavaVMInitArgs initArgs;
    JavaVMOption opt;
    char propBuf[PROPERTY_VALUE_MAX];
    char stackTraceFileBuf[PROPERTY_VALUE_MAX];
    char dexoptFlagsBuf[PROPERTY_VALUE_MAX];
    char enableAssertBuf[sizeof("-ea:")-1 + PROPERTY_VALUE_MAX];
    char jniOptsBuf[sizeof("-Xjniopts:")-1 + PROPERTY_VALUE_MAX];
    char heapstartsizeOptsBuf[sizeof("-Xms")-1 + PROPERTY_VALUE_MAX];
    char heapsizeOptsBuf[sizeof("-Xmx")-1 + PROPERTY_VALUE_MAX];
    char heapgrowthlimitOptsBuf[sizeof("-XX:HeapGrowthLimit=")-1 + PROPERTY_VALUE_MAX];
    char heapminfreeOptsBuf[sizeof("-XX:HeapMinFree=")-1 + PROPERTY_VALUE_MAX];
    char heapmaxfreeOptsBuf[sizeof("-XX:HeapMaxFree=")-1 + PROPERTY_VALUE_MAX];
    char heaptargetutilizationOptsBuf[sizeof("-XX:HeapTargetUtilization=")-1 + PROPERTY_
VALUE_MAX];
    char jitcodecachesizeOptsBuf[sizeof("-Xjitcodecachesize:")-1 + PROPERTY_VALUE_MAX];
    char extraOptsBuf[PROPERTY_VALUE_MAX];
    char* stackTraceFile = NULL;
    bool checkJni = false;
    bool checkDexSum = false;
    bool logStdio = false;
    enum {
      kEMDefault,
      kEMIntPortable,
      kEMIntFast,
      kEMJitCompiler,
    } executionMode = kEMDefault;


    property_get("dalvik.vm.checkjni", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkJni = true;
    } else if (strcmp(propBuf, "false") != 0) {
        /* property is neither true nor false; fall back on kernel parameter */
        property_get("ro.kernel.android.checkjni", propBuf, "");
        if (propBuf[0] == '1') {
            checkJni = true;
        }
    }

    property_get("dalvik.vm.execution-mode", propBuf, "");
    if (strcmp(propBuf, "int:portable") == 0) {
        executionMode = kEMIntPortable;
    } else if (strcmp(propBuf, "int:fast") == 0) {
        executionMode = kEMIntFast;
    } else if (strcmp(propBuf, "int:jit") == 0) {
        executionMode = kEMJitCompiler;
    }

    property_get("dalvik.vm.stack-trace-file", stackTraceFileBuf, "");

    property_get("dalvik.vm.check-dex-sum", propBuf, "");
    if (strcmp(propBuf, "true") == 0) {
        checkDexSum = true;
    }

    property_get("log.redirect-stdio", propBuf, "");
```

```
if (strcmp(propBuf, "true") == 0) {
    logStdio = true;
}

strcpy(enableAssertBuf, "-ea:");
property_get("dalvik.vm.enableassertions", enableAssertBuf+4, "");

strcpy(jniOptsBuf, "-Xjniopts:");
property_get("dalvik.vm.jniopts", jniOptsBuf+10, "");

/* exit()线程处理 */
opt.extraInfo = (void*) runtime_exit;
opt.optionString = "exit";
mOptions.add(opt);

/* fprintf()线程处理*/
opt.extraInfo = (void*) runtime_vfprintf;
opt.optionString = "vfprintf";
mOptions.add(opt);

/* 注册敏感线程框架 */
opt.extraInfo = (void*) runtime_isSensitiveThread;
opt.optionString = "sensitiveThread";
mOptions.add(opt);

opt.extraInfo = NULL;

/* enable verbose; standard options are { jni, gc, class } */
//options[curOpt++].optionString = "-verbose:jni";
opt.optionString = "-verbose:gc";
mOptions.add(opt);
//options[curOpt++].optionString = "-verbose:class";

/*
 *默认的启动和堆的最大尺寸
 */
strcpy(heapstartsizeOptsBuf, "-Xms");
property_get("dalvik.vm.heapstartsize", heapstartsizeOptsBuf+4, "4m");
opt.optionString = heapstartsizeOptsBuf;
mOptions.add(opt);
strcpy(heapsizeOptsBuf, "-Xmx");
property_get("dalvik.vm.heapsize", heapsizeOptsBuf+4, "16m");
opt.optionString = heapsizeOptsBuf;
mOptions.add(opt);

//增加错误主线程的解释器的堆栈大小: 6315322
opt.optionString = "-XX:mainThreadStackSize=24K";
mOptions.add(opt);

//设置最大 JIT 代码缓存大小。注: 0 表示将禁用 JIT
strcpy(jitcodecachesizeOptsBuf, "-Xjitcodecachesize:");
property_get("dalvik.vm.jit.codecachesize", jitcodecachesizeOptsBuf+19,  NULL);
if (jitcodecachesizeOptsBuf[19] != '\0') {
  opt.optionString = jitcodecachesizeOptsBuf;
  mOptions.add(opt);
}

strcpy(heapgrowthlimitOptsBuf, "-XX:HeapGrowthLimit=");
property_get("dalvik.vm.heapgrowthlimit", heapgrowthlimitOptsBuf+20, "");
if (heapgrowthlimitOptsBuf[20] != '\0') {
    opt.optionString = heapgrowthlimitOptsBuf;
    mOptions.add(opt);
}

strcpy(heapminfreeOptsBuf, "-XX:HeapMinFree=");
property_get("dalvik.vm.heapminfree", heapminfreeOptsBuf+16, "");
if (heapminfreeOptsBuf[16] != '\0') {
    opt.optionString = heapminfreeOptsBuf;
    mOptions.add(opt);
}
```

```
strcpy(heapmaxfreeOptsBuf, "-XX:HeapMaxFree=");
property_get("dalvik.vm.heapmaxfree", heapmaxfreeOptsBuf+16, "");
if (heapmaxfreeOptsBuf[16] != '\0') {
    opt.optionString = heapmaxfreeOptsBuf;
    mOptions.add(opt);
}

strcpy(heaptargetutilizationOptsBuf, "-XX:HeapTargetUtilization=");
property_get("dalvik.vm.heaptargetutilization",heaptargetutilizationOptsBuf+26,"");
if (heaptargetutilizationOptsBuf[26] != '\0') {
    opt.optionString = heaptargetutilizationOptsBuf;
    mOptions.add(opt);
}

property_get("ro.config.low_ram", propBuf, "");
if (strcmp(propBuf, "true") == 0) {
  opt.optionString = "-XX:LowMemoryMode";
  mOptions.add(opt);
}

/*
 *启用或禁用 dexopt 特征，如字节码校验和为精确计算 GC 寄存器映射
 */
property_get("dalvik.vm.dexopt-flags", dexoptFlagsBuf, "");
if (dexoptFlagsBuf[0] != '\0') {
    const char* opc;
    const char* val;

    opc = strstr(dexoptFlagsBuf, "v=");     /* verification */
    if (opc != NULL) {
        switch (*(opc+2)) {
        case 'n':   val = "-Xverify:none";     break;
        case 'r':   val = "-Xverify:remote";   break;
        case 'a':   val = "-Xverify:all";      break;
        default:    val = NULL;               break;
        }

        if (val != NULL) {
            opt.optionString = val;
            mOptions.add(opt);
        }
    }

    opc = strstr(dexoptFlagsBuf, "o=");     /* optimization */
    if (opc != NULL) {
        switch (*(opc+2)) {
        case 'n':   val = "-Xdexopt:none";     break;
        case 'v':   val = "-Xdexopt:verified"; break;
        case 'a':   val = "-Xdexopt:all";      break;
        case 'f':   val = "-Xdexopt:full";     break;
        default:    val = NULL;               break;
        }

        if (val != NULL) {
            opt.optionString = val;
            mOptions.add(opt);
        }
    }

    opc = strstr(dexoptFlagsBuf, "m=y");    /* register map */
    if (opc != NULL) {
        opt.optionString = "-Xgenregmap";
        mOptions.add(opt);

        /* turn on precise GC while we're at it */
        opt.optionString = "-Xgc:precise";
        mOptions.add(opt);
    }
}

/*启用调试；设置暂停= Y，暂停 VM 初始化*/
```

```
    /* use android ADB transport */
    opt.optionString =
        "-agentlib:jdwp=transport=dt_android_adb,suspend=n,server=y";
    mOptions.add(opt);

    ALOGD("CheckJNI is %s\n", checkJni ? "ON" : "OFF");
    if (checkJni) {
        /*扩展的 JNI 检查*/
        opt.optionString = "-Xcheck:jni";
        mOptions.add(opt);

        /* 设置 JNI 全局引用*/
        opt.optionString = "-Xjnigreflimit:2000";
        mOptions.add(opt);

        /* with -Xcheck:jni, this provides a JNI function call trace */
        //opt.optionString = "-verbose:jni";
        //mOptions.add(opt);
    }

    char lockProfThresholdBuf[sizeof("-Xlockprofthreshold:") + sizeof(propBuf)];
    property_get("dalvik.vm.lockprof.threshold", propBuf, "");
    if (strlen(propBuf) > 0) {
      strcpy(lockProfThresholdBuf, "-Xlockprofthreshold:");
      strcat(lockProfThresholdBuf, propBuf);
      opt.optionString = lockProfThresholdBuf;
      mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected opcodes. Eg "1-0a,3c,f1-ff" */
    char jitOpBuf[sizeof("-Xjitop:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.op", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitOpBuf, "-Xjitop:");
        strcat(jitOpBuf, propBuf);
        opt.optionString = jitOpBuf;
        mOptions.add(opt);
    }

    /* Force interpreter-only mode for selected methods */
    char jitMethodBuf[sizeof("-Xjitmethod:") + PROPERTY_VALUE_MAX];
    property_get("dalvik.vm.jit.method", propBuf, "");
    if (strlen(propBuf) > 0) {
        strcpy(jitMethodBuf, "-Xjitmethod:");
        strcat(jitMethodBuf, propBuf);
        opt.optionString = jitMethodBuf;
        mOptions.add(opt);
    }

    if (executionMode == kEMIntPortable) {
        opt.optionString = "-Xint:portable";
        mOptions.add(opt);
    } else if (executionMode == kEMIntFast) {
        opt.optionString = "-Xint:fast";
        mOptions.add(opt);
    } else if (executionMode == kEMJitCompiler) {
        opt.optionString = "-Xint:jit";
        mOptions.add(opt);
    }

    if (checkDexSum) {
        /* perform additional DEX checksum tests */
        opt.optionString = "-Xcheckdexsum";
        mOptions.add(opt);
    }

    if (logStdio) {
        /* convert stdout/stderr to log messages */
        opt.optionString = "-Xlog-stdio";
        mOptions.add(opt);
    }
```

```
    if (enableAssertBuf[4] != '\0') {
        /* accept "all" to mean "all classes and packages" */
        if (strcmp(enableAssertBuf+4, "all") == 0)
            enableAssertBuf[3] = '\0';
        ALOGI("Assertions enabled: '%s'\n", enableAssertBuf);
        opt.optionString = enableAssertBuf;
        mOptions.add(opt);
    } else {
        ALOGV("Assertions disabled\n");
    }

    if (jniOptsBuf[10] != '\0') {
        ALOGI("JNI options: '%s'\n", jniOptsBuf);
        opt.optionString = jniOptsBuf;
        mOptions.add(opt);
    }

    if (stackTraceFileBuf[0] != '\0') {
        static const char* stfOptName = "-Xstacktracefile:";

        stackTraceFile = (char*) malloc(strlen(stfOptName) +
            strlen(stackTraceFileBuf) +1);
        strcpy(stackTraceFile, stfOptName);
        strcat(stackTraceFile, stackTraceFileBuf);
        opt.optionString = stackTraceFile;
        mOptions.add(opt);
    }

    /* extra options; parse this late so it overrides others */
    property_get("dalvik.vm.extra-opts", extraOptsBuf, "");
    parseExtraOpts(extraOptsBuf);

    /* 设置本地属性 */
    {
        char langOption[sizeof("-Duser.language=") + 3];
        char regionOption[sizeof("-Duser.region=") + 3];
        strcpy(langOption, "-Duser.language=");
        strcpy(regionOption, "-Duser.region=");
        readLocale(langOption, regionOption);
        opt.extraInfo = NULL;
        opt.optionString = langOption;
        mOptions.add(opt);
        opt.optionString = regionOption;
        mOptions.add(opt);
    }
    opt.optionString = "-Djava.io.tmpdir=/sdcard";
    mOptions.add(opt);

    initArgs.version = JNI_VERSION_1_4;
    initArgs.options = mOptions.editArray();
    initArgs.nOptions = mOptions.size();
    initArgs.ignoreUnrecognized = JNI_FALSE;

    /*
     * 初始化 VM

     */
    if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
        ALOGE("JNI_CreateJavaVM failed\n");
        goto bail;
    }

    result = 0;

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

由上述实现代码可知，函数 AndroidRuntime::startVm 最终会调用 JNI_CreateJavaVM 函数。此

处的函数 JNI_CreateJavaVM 在文件“art/runtime/jni_internal.cc”中定义，具体实现代码如下所示。

```
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
  if (IsBadJniVersion(args->version)) {
    LOG(ERROR) << "Bad JNI version passed to CreateJavaVM: " << args->version;
    return JNI_EVERSION;
  }
  Runtime::Options options;
  for (int i = 0; i < args->nOptions; ++i) {
    JavaVMOption* option = &args->options[i];
    options.push_back(std::make_pair(std::string(option->optionString),
option->extraInfo));
  }
  bool ignore_unrecognized = args->ignoreUnrecognized;
  if (!Runtime::Create(options, ignore_unrecognized)) {
    return JNI_ERR;
  }
  Runtime* runtime = Runtime::Current();
  bool started = runtime->Start();
  if (!started) {
    delete Thread::Current()->GetJniEnv();
    delete runtime->GetJavaVM();
    LOG(WARNING) << "CreateJavaVM failed";
    return JNI_ERR;
  }
  *p_env = Thread::Current()->GetJniEnv();
  *p_vm = runtime->GetJavaVM();
  return JNI_OK;
}
```

类 JniInvocation 的静态成员函数 GetJniInvocation 返回的便是前面所创建的 JniInvocation 实例。有了这个 JniInvocation 实例之后，就继续调用它的成员函数 JNI_CreateJavaVM 来创建一个 JavaVM 接口及其对应的 JNIEnv 接口。

函数 GetJniInvocation 定义在文件 libnativehelper/JniInvocation.cpp 中，具体实现代码如下所示。

```
jint JniInvocation::JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  return JNI_CreateJavaVM_(p_vm, p_env, vm_args);
}
```

类 JniInvocation 的成员变量 JNI_CreateJavaVM_指向的就是前面所加载的 libdvm.so 或者 libart.so 所导出的函数 JNI_CreateJavaVM。类 JniInvocation 的成员函数 JNI_CreateJavaVM 返回的 JavaVM 接口指向的要么是 Dalvik 虚拟机，要么是 ART 虚拟机。

20.2.3　创建运行实例

在文件“art/runtime/jni_internal.cc”中，函数 JNI_CreateJavaVM 会调用函数 Create 创建 Runtime 的实例。函数 Create 在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
bool Runtime::Create(const Options& options, bool ignore_unrecognized) {
  // TODO: acquire a static mutex on Runtime to avoid racing.
  if (Runtime::instance_ != NULL) {
    return false;
  }
  InitLogging(NULL);  //初始化 Log 系统
  instance_ = new Runtime; //创建 Runtime 实例
  if (!instance_->Init(options, ignore_unrecognized)) {
    delete instance_;
    instance_ = NULL;
    return false;//初始化 Runtime
  }
  return true;
}
```

再次回到函数 JNI_Create JavaVM 中，“Runtime* runtime = Runtime::Current()”用于获得 Runtime 当前实例，Runtime 使用单例模式实现。“bool started = runtime->Start()”用于调用 Start 函数，该函数在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
bool Runtime::Start() {
  VLOG(startup) << "Runtime::Start entering";
  CHECK(host_prefix_.empty()) << host_prefix_;
  Thread* self = Thread::Current();//获得当前运行线程
  self->TransitionFromRunnableToSuspended(kNative);//将该线程状态从 Runnable 切换到 Suspend

  started_ = true;
  // 完成 Native 函数的初始化工作
  InitNativeMethods();

  // Initialize well known thread group values that may be accessed threads while attaching.
  InitThreadGroups(self);

  Thread::FinishStartup();

  if (is_zygote_) {
    if (!InitZygote()) {
      return false;
    }
  } else {
    DidForkFromZygote();
  }

  StartDaemonThreads();

  system_class_loader_ = CreateSystemClassLoader();

  self->GetJniEnv()->locals.AssertEmpty();

  VLOG(startup) << "Runtime::Start exiting";

  finished_starting_ = true;

  return true;
}
```

函数 Runtime::InitNativeMethods 在文件 art/runtime/runtime.cc 中定义，具体实现代码如下所示。

```
void Runtime::InitNativeMethods() {
  VLOG(startup) << "Runtime::InitNativeMethods entering";
  Thread* self = Thread::Current();
  JNIEnv* env = self->GetJniEnv();//获取 JNI 环境

  // Must be in the kNative state for calling native methods (JNI_OnLoad code).
  CHECK_EQ(self->GetState(), kNative);
  JniConstants::init(env);
  WellKnownClasses::Init(env);

  //调用 RegisterRuntimeNativeMethods 函数完成 Native 函数的注册
  RegisterRuntimeNativeMethods(env);
  {
    std::string mapped_name(StringPrintf(OS_SHARED_LIB_FORMAT_STR, "javacore"));
    std::string reason;
    self->TransitionFromSuspendedToRunnable();
    if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
      LOG(FATAL) << "LoadNativeLibrary failed for \"" << mapped_name << "\": " << reason;
    }
    self->TransitionFromRunnableToSuspended(kNative);
  }

  // Initialize well known classes that may invoke runtime native methods.
  WellKnownClasses::LateInit(env);

  VLOG(startup) << "Runtime::InitNativeMethods exiting";
}
```

20.2.4 注册本地 JNI 函数

在文件 art/runtime/runtime.cc 中的函数 Runtime::InitNativeMethods 中，通过代码行“RegisterRuntimeNativeMethods(env)”调用函数 RegisterRuntimeNativeMethods 来注册 Native 函数，函数

RegisterRuntimeNativeMethods 具体实现代码如下所示。

```
void Runtime::RegisterRuntimeNativeMethods(JNIEnv* env) {
#define REGISTER(FN) extern void FN(JNIEnv*); FN(env)
  // Register Throwable first so that registration of other native methods can throw
exceptions
  REGISTER(register_java_lang_Throwable);
  REGISTER(register_dalvik_system_DexFile);
  REGISTER(register_dalvik_system_VMDebug);
  REGISTER(register_dalvik_system_VMRuntime);
  REGISTER(register_dalvik_system_VMStack);
  REGISTER(register_dalvik_system_Zygote);
  REGISTER(register_java_lang_Class);
  REGISTER(register_java_lang_DexCache);
  REGISTER(register_java_lang_Object);
  REGISTER(register_java_lang_Runtime);
  REGISTER(register_java_lang_String);
  REGISTER(register_java_lang_System);
  REGISTER(register_java_lang_Thread);
  REGISTER(register_java_lang_VMClassLoader);
  REGISTER(register_java_lang_reflect_Array);
  REGISTER(register_java_lang_reflect_Constructor);
  REGISTER(register_java_lang_reflect_Field);
  REGISTER(register_java_lang_reflect_Method);
  REGISTER(register_java_lang_reflect_Proxy);
  REGISTER(register_java_util_concurrent_atomic_AtomicLong);
  REGISTER(register_org_apache_harmony_dalvik_ddmc_DdmServer);
  REGISTER(register_org_apache_harmony_dalvik_ddmc_DdmVmInternal);
  REGISTER(register_sun_misc_Unsafe);
#undef REGISTER
}
```

在上述代码中列出了需要注册的函数列表，有关上述函数的具体实现请读者自行分析，例如 register_java_lang_Throwable 在文件“runtime/native/java_lang_Throwable.cc”中定义，具体实现代码如下所示。

```
void register_java_lang_Throwable(JNIEnv* env) {
  REGISTER_NATIVE_METHODS("java/lang/Throwable");
}
```

20.2.5　启动守护进程

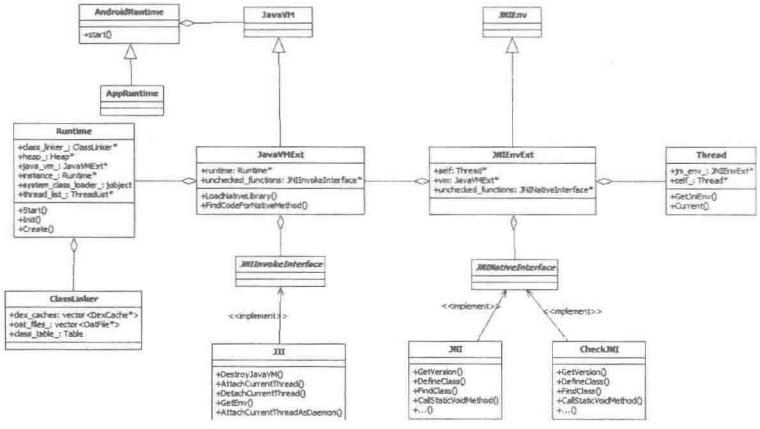
再次返回到 Runtime:: Start 函数，“if (!InitZygote()) {”代码行用于调用 InitZygote 完成一些文件系统的 mount 工作。然后通过“StartDaemonThreads()”代码行调用 java.lang.Daemons.start()函数启动守护进程。函数 StartDaemonThreads()的具体实现代码如下所示。

```
void Runtime::StartDaemonThreads() {
  VLOG(startup) << "Runtime::StartDaemonThreads entering";

  Thread* self = Thread::Current();

  // Must be in the kNative state for calling native methods.
  CHECK_EQ(self->GetState(), kNative);

  JNIEnv* env = self->GetJniEnv();
  env->CallStaticVoidMethod(WellKnownClasses::java_lang_Daemons,
                            WellKnownClasses::java_lang_Daemons_start);
  if (env->ExceptionCheck()) {
    env->ExceptionDescribe();
    LOG(FATAL) << "Error starting java.lang.Daemons";
  }

  VLOG(startup) << "Runtime::StartDaemonThreads exiting";
}
```

综上所述，Android 系统通过将 ART 运行时抽象成一个 Java 虚拟机，以及通过系统属性 persist.sys.dalvik.vm.lib 和一个适配层 JniInvocation，就可以无缝地将 Dalvik 虚拟机替换为 ART 运行时。这个替换过程设计非常巧妙，因为涉及的代码修改是非常少的。涉及类的具体关系如图 20-5 所示。

20.2.6 解析参数

在函数 JNI_CreateJavaVM()中，先调用 Create()函数创建 Runtime。Runtime 是一个单例，创建后会马上调用文件“/art/runtime/runtime.cc”中的 Init()函数。函数 Init()的功能是解析参数，初始化 Heap 和 JavaVMExt 结构，实现线程和信号处理，并创建 ClassLinker 等。函数 Init()的具体实现代码如下所示。



▲图 20-5 启动 ART 涉及的类

```
bool Runtime::Start() {
  VLOG(startup) << "Runtime::Start entering";

  CHECK(host_prefix_.empty()) << host_prefix_;

  // Restore main thread state to kNative as expected by native code.
  Thread* self = Thread::Current();
  self->TransitionFromRunnableToSuspended(kNative);

  started_ = true;

  // InitNativeMethods needs to be after started_ so that the classes
  // it touches will have methods linked to the oat file if necessary.
  InitNativeMethods();

  // Initialize well known thread group values that may be accessed threads while attaching.
  InitThreadGroups(self);

  Thread::FinishStartup();

  if (is_zygote_) {
    if (!InitZygote()) {
      return false;
    }
  } else {
    DidForkFromZygote();
  }

  StartDaemonThreads();

  system_class_loader_ = CreateSystemClassLoader();

  self->GetJniEnv()->locals.AssertEmpty();

  VLOG(startup) << "Runtime::Start exiting";

  finished_starting_ = true;

  return true;
}
```

在文件“/art/runtime/runtime.cc”中，通过函数 Runtime::ParsedOptions* Runtime::ParsedOptions::Creat 解析参数，将 raw_options 中的参数放入 parsed，如对环境变量 BOOTCLASSPATH 和 CLASSPATH 的处理。函数 Runtime::ParsedOptions* Runtime::ParsedOptions::Creat 的具体实现代码如下所示。

```
Runtime::ParsedOptions* Runtime::ParsedOptions::Create(const Options& options, bool
ignore_unrecognized) {
  UniquePtr<ParsedOptions> parsed(new ParsedOptions());
  const char* boot_class_path_string = getenv("BOOTCLASSPATH");
  if (boot_class_path_string != NULL) {
    parsed->boot_class_path_string_ = boot_class_path_string;
  }
  const char* class_path_string = getenv("CLASSPATH");
  if (class_path_string != NULL) {
    parsed->class_path_string_ = class_path_string;
  }
  // -Xcheck:jni is off by default for regular builds but on by default in debug builds.
  parsed->check_jni_ = kIsDebugBuild;

  parsed->heap_initial_size_ = gc::Heap::kDefaultInitialSize;
  parsed->heap_maximum_size_ = gc::Heap::kDefaultMaximumSize;
  parsed->heap_min_free_ = gc::Heap::kDefaultMinFree;
  parsed->heap_max_free_ = gc::Heap::kDefaultMaxFree;
  parsed->heap_target_utilization_ = gc::Heap::kDefaultTargetUtilization;
  parsed->heap_growth_limit_ = 0;  // 0 means no growth limit.
  // Default to number of processors minus one since the main GC thread also does work.
  parsed->parallel_gc_threads_ = sysconf(_SC_NPROCESSORS_CONF) - 1;
  // Only the main GC thread, no workers.
  parsed->conc_gc_threads_ = 0;
  parsed->stack_size_ = 0;  // 0 means default.
  parsed->low_memory_mode_ = false;
……
```

然后初始化 Monitor（相当于 mutex+conditional variable，可用于多个线程同步）和线程链表等，然后实现比较重要的 Heap 及 GC 的初始化工作。其中 gc::Heap 功能通过文件“art/runtime/gc/heap.cc”中的函数 Heap::Heap 实现，具体实现代码如下所示。

```
Heap::Heap(size_t initial_size, size_t growth_limit, size_t min_free, size_t max_free,
           double target_utilization, size_t capacity, const std::string& original_image_
           file_name,
           bool concurrent_gc, size_t parallel_gc_threads, size_t conc_gc_threads,
           bool low_memory_mode, size_t long_pause_log_threshold, size_t long_gc_log_
           threshold,
           bool ignore_max_footprint)
    : alloc_space_(NULL),
      card_table_(NULL),
      concurrent_gc_(concurrent_gc),
      parallel_gc_threads_(parallel_gc_threads),
      conc_gc_threads_(conc_gc_threads),
      low_memory_mode_(low_memory_mode),
      long_pause_log_threshold_(long_pause_log_threshold),
      long_gc_log_threshold_(long_gc_log_threshold),
      ignore_max_footprint_(ignore_max_footprint),
      have_zygote_space_(false),
      soft_ref_queue_lock_(NULL),
      weak_ref_queue_lock_(NULL),
      finalizer_ref_queue_lock_(NULL),
      phantom_ref_queue_lock_(NULL),
      is_gc_running_(false),
      last_gc_type_(collector::kGcTypeNone),
      next_gc_type_(collector::kGcTypePartial),
      capacity_(capacity),
      growth_limit_(growth_limit),
      max_allowed_footprint_(initial_size),
      native_footprint_gc_watermark_(initial_size),
      native_footprint_limit_(2 * initial_size),
      activity_thread_class_(NULL),
      application_thread_class_(NULL),
      activity_thread_(NULL),
```

```
    application_thread_(NULL),
    last_process_state_id_(NULL),
    // Initially care about pauses in case we never get notified of process states, or
    if the JNI
    // code becomes broken.
    care_about_pause_times_(true),
    concurrent_start_bytes_(concurrent_gc_ ? initial_size - kMinConcurrentRemainingBytes
        : std::numeric_limits<size_t>::max()),
    total_bytes_freed_ever_(0),
    total_objects_freed_ever_(0),
    large_object_threshold_(3 * kPageSize),
    num_bytes_allocated_(0),
    native_bytes_allocated_(0),
    gc_memory_overhead_(0),
    verify_missing_card_marks_(false),
    verify_system_weaks_(false),
    verify_pre_gc_heap_(false),
    verify_post_gc_heap_(false),
    verify_mod_union_table_(false),
    min_alloc_space_size_for_sticky_gc_(2 * MB),
    min_remaining_space_for_sticky_gc_(1 * MB),
    last_trim_time_ms_(0),
    allocation_rate_(0),
    /* For GC a lot mode, we limit the allocations stacks to be kGcAlotInterval allocations.
    This
     * causes a lot of GC since we do a GC for alloc whenever the stack is full. When heap
     * verification is enabled, we limit the size of allocation stacks to speed up their
     * searching.
     */
    max_allocation_stack_size_(kGCALotMode ? kGcAlotInterval
        : (kDesiredHeapVerification > kNoHeapVerification) ? KB : MB),
    reference_referent_offset_(0),
    reference_queue_offset_(0),
    reference_queueNext_offset_(0),
    reference_pendingNext_offset_(0),
    finalizer_reference_zombie_offset_(0),
    min_free_(min_free),
    max_free_(max_free),
    target_utilization_(target_utilization),
    total_wait_time_(0),
    total_allocation_time_(0),
    verify_object_mode_(kHeapVerificationNotPermitted),
    running_on_valgrind_(RUNNING_ON_VALGRIND) {
if (VLOG_IS_ON(heap) || VLOG_IS_ON(startup)) {
  LOG(INFO) << "Heap() entering";
}

live_bitmap_.reset(new accounting::HeapBitmap(this));
mark_bitmap_.reset(new accounting::HeapBitmap(this));

// Requested begin for the alloc space, to follow the mapped image and oat files
byte* requested_alloc_space_begin = NULL;
std::string image_file_name(original_image_file_name);
if (!image_file_name.empty()) {
  space::ImageSpace* image_space = space::ImageSpace::Create(image_file_name);
  CHECK(image_space != NULL) << "Failed to create space for " << image_file_name;
  AddContinuousSpace(image_space);
  // Oat files referenced by image files immediately follow them in memory, ensure alloc space
  // isn't going to get in the middle
  byte* oat_file_end_addr = image_space->GetImageHeader().GetOatFileEnd();
  CHECK_GT(oat_file_end_addr, image_space->End());
  if (oat_file_end_addr > requested_alloc_space_begin) {
    requested_alloc_space_begin =
        reinterpret_cast<byte*>(RoundUp(reinterpret_cast<uintptr_t>(oat_file_end_addr),
                                        kPageSize));
  }
}

alloc_space_ = space::DlMallocSpace::Create(Runtime::Current()->IsZygote() ? "zygote
space" : "alloc space", initial_size,
```

```
                                        growth_limit, capacity,
                                        requested_alloc_space_begin);
  CHECK(alloc_space_ != NULL) << "Failed to create alloc space";
  alloc_space_->SetFootprintLimit(alloc_space_->Capacity());
  AddContinuousSpace(alloc_space_);

  // Allocate the large object space.
  const bool kUseFreeListSpaceForLOS = false;
  if (kUseFreeListSpaceForLOS) {
    large_object_space_=space::FreeListSpace::Create("large object space",NULL,capacity);
  } else {
    large_object_space_ = space::LargeObjectMapSpace::Create("large object space");
  }
  CHECK(large_object_space_ != NULL) << "Failed to create large object space";
  AddDiscontinuousSpace(large_object_space_);

  // Compute heap capacity. Continuous spaces are sorted in order of Begin().
  byte* heap_begin = continuous_spaces_.front()->Begin();
  size_t heap_capacity = continuous_spaces_.back()->End() - continuous_spaces_
.front()->Begin();
  if (continuous_spaces_.back()->IsDlMallocSpace()) {
    heap_capacity += continuous_spaces_.back()->AsDlMallocSpace()->NonGrowthLimit Capacity();
  }

  // Allocate the card table.
  card_table_.reset(accounting::CardTable::Create(heap_begin, heap_capacity));
  CHECK(card_table_.get() != NULL) << "Failed to create card table";

  image_mod_union_table_.reset(new accounting::ModUnionTableToZygoteAllocspace(this));
  CHECK(image_mod_union_table_.get() != NULL) << "Failed to create image mod-union table";

  zygote_mod_union_table_.reset(new accounting::ModUnionTableCardCache(this));
  CHECK(zygote_mod_union_table_.get() != NULL) << "Failed to create Zygote mod-union table";

  // TODO: Count objects in the image space here.
  num_bytes_allocated_ = 0;

  // Default mark stack size in bytes.
  static const size_t default_mark_stack_size = 64 * KB;
  mark_stack_.reset(accounting::ObjectStack::Create("mark stack", default_mark_stack_ size));
  allocation_stack_.reset(accounting::ObjectStack::Create("allocation stack",
                                                          max_allocation_stack_size_));
  live_stack_.reset(accounting::ObjectStack::Create("live stack",
                                                    max_allocation_stack_size_));

  // It's still too early to take a lock because there are no threads yet, but we can create locks
  // now. We don't create it earlier to make it clear that you can't use locks during heap
  // initialization.
  gc_complete_lock_ = new Mutex("GC complete lock");
  gc_complete_cond_.reset(new ConditionVariable("GC complete condition variable",
                                                *gc_complete_lock_));

  // Create the reference queue locks, this is required so for parallel object scanning in the GC.
  soft_ref_queue_lock_ = new Mutex("Soft reference queue lock");
  weak_ref_queue_lock_ = new Mutex("Weak reference queue lock");
  finalizer_ref_queue_lock_ = new Mutex("Finalizer reference queue lock");
  phantom_ref_queue_lock_ = new Mutex("Phantom reference queue lock");

  last_gc_time_ns_ = NanoTime();
  last_gc_size_ = GetBytesAllocated();

  if (ignore_max_footprint_) {
    SetIdealFootprint(std::numeric_limits<size_t>::max());
    concurrent_start_bytes_ = max_allowed_footprint_;
  }

  // Create our garbage collectors.
  for (size_t i = 0; i < 2; ++i) {
    const bool concurrent = i != 0;
    mark_sweep_collectors_.push_back(new collector::MarkSweep(this, concurrent));
    mark_sweep_collectors_.push_back(new collector::PartialMarkSweep(this, concurrent));
```

```
    mark_sweep_collectors_.push_back(new collector::StickyMarkSweep(this, concurrent));
  }

  CHECK_NE(max_allowed_footprint_, 0U);
  if (VLOG_IS_ON(heap) || VLOG_IS_ON(startup)) {
    LOG(INFO) << "Heap() exiting";
  }
}
```

在上述代码中，函数 ImageSpace::Create()会检测 image 文件，如果没有就调用 GenerateImage()来创建。正因为上述操作过程，所以 log 中会有如下所示的信息。

```
I/art  (161):  GenerateImage:  /system/bin/dex2oat--image=/data/dalvik-cache/system@
framework@boot.art  --runtime-arg  -Xms64m--runtime-arg  -Xmx64m  --dex-file=/system/
framework/core-libart.jar ... --oat-file=/data/dalvik-cache/system@framework@boot.oat
--base=0x60000000--image-classes-zip=/system/framework/framework...
```

上述过程调用了 dex2oat，把 BOOTCLASSPATH 里的包打成 image 文件，它最后会生成两个文件 boot.art 和 boot.oat。其中前者是 image 文件，后者是 elf 文件。这个 image 会被放到创建的 Heap 中。在函数 Heap::Heap 中，接下来会为一些数据结构分配空间，创建各种互斥量及初始化 GC。其中 MarkSweep、PartialMarkSweep 和 StickyMarkSweep 都是 art::gc::collector::GarbageCollector 的继承类，和几个子类应用了 Template Method 模式。在函数 GarbageCollector::Run()中实现了主要算法，此函数在文件“art/runtime/gc/collector/garbage_collector.cc”中定义，具体实现代码如下所示。

```
void GarbageCollector::Run() {
  ThreadList* thread_list = Runtime::Current()->GetThreadList();
  uint64_t start_time = NanoTime();
  pause_times_.clear();
  duration_ns_ = 0;

  InitializePhase();

  if (!IsConcurrent()) {
    // Pause is the entire length of the GC.
    uint64_t pause_start = NanoTime();
    ATRACE_BEGIN("Application threads suspended");
    thread_list->SuspendAll();
    MarkingPhase();
    ReclaimPhase();
    thread_list->ResumeAll();
    ATRACE_END();
    uint64_t pause_end = NanoTime();
    pause_times_.push_back(pause_end - pause_start);
  } else {
    Thread* self = Thread::Current();
    {
      ReaderMutexLock mu(self, *Locks::mutator_lock_);
      MarkingPhase();
    }
    bool done = false;
    while (!done) {
      uint64_t pause_start = NanoTime();
      ATRACE_BEGIN("Suspending mutator threads");
      thread_list->SuspendAll();
      ATRACE_END();
      ATRACE_BEGIN("All mutator threads suspended");
      done = HandleDirtyObjectsPhase();
      ATRACE_END();
      uint64_t pause_end = NanoTime();
      ATRACE_BEGIN("Resuming mutator threads");
      thread_list->ResumeAll();
      ATRACE_END();
      pause_times_.push_back(pause_end - pause_start);
    }
    {
      ReaderMutexLock mu(self, *Locks::mutator_lock_);
      ReclaimPhase();
```

```
    }
  }

  uint64_t end_time = NanoTime();
  duration_ns_ = end_time - start_time;

  FinishPhase();
}
```

在上述代码中调用了 InitializePhase()、MarkingPhase()、ReclaimPhase()和 FinishPhase()等虚函数，这几个虚函数在 MarkSweep 等几个子类中有具体实现。

再次回到函数 Runtime::Init()，通过如下代码实现 ClassLinker 的初始化操作，其主要功能是调用 CreateFromImage()函数实现的。

```
ClassLinker* ClassLinker::CreateFromImage(InternTable* intern_table) {
  UniquePtr<ClassLinker> class_linker(new ClassLinker(intern_table));
  class_linker->InitFromImage();
  return class_linker.release();
}
```

在文件“art\art\runtime\class_linker.cc”中，通过函数 InitFromImage()从 Heap 中得到 image 的空间，然后得到 dex caches 数组，接着把这些 dex caches 对应的 dex file 信息注册到 BootClassPath 中去。函数 InitFromImage()的具体实现代码如下所示。

```
void ClassLinker::InitFromImage() {
  VLOG(startup) << "ClassLinker::InitFromImage entering";
  CHECK(!init_done_);

  gc::Heap* heap = Runtime::Current()->GetHeap();
  gc::space::ImageSpace* space = heap->GetImageSpace();
  dex_cache_image_class_lookup_required_ = true;
  CHECK(space != NULL);
  OatFile& oat_file = GetImageOatFile(space);
  CHECK_EQ(oat_file.GetOatHeader().GetImageFileLocationOatChecksum(), 0U);
  CHECK_EQ(oat_file.GetOatHeader().GetImageFileLocationOatDataBegin(), 0U);
  CHECK(oat_file.GetOatHeader().GetImageFileLocation().empty());
  portable_resolution_trampoline_ = oat_file.GetOatHeader().GetPortableResolution
  Trampoline();
  quick_resolution_trampoline_ = oat_file.GetOatHeader().GetQuickResolutionTrampoline();
  mirror::Object* dex_caches_object = space->GetImageHeader().GetImageRoot(Image
  Header::kDexCaches);
  mirror::ObjectArray<mirror::DexCache>* dex_caches =
      dex_caches_object->AsObjectArray<mirror::DexCache>();

  mirror::ObjectArray<mirror::Class>* class_roots =
      space->GetImageHeader().GetImageRoot(ImageHeader::kClassRoots)->AsObjectArray<
mirror::Class>();
  class_roots_ = class_roots;

  // Special case of setting up the String class early so that we can test arbitrary objects
  // as being Strings or not
  mirror::String::SetClass(GetClassRoot(kJavaLangString));

  CHECK_EQ(oat_file.GetOatHeader().GetDexFileCount(),
           static_cast<uint32_t>(dex_caches->GetLength()));
  Thread* self = Thread::Current();
  for (int32_t i = 0; i < dex_caches->GetLength(); i++) {
    SirtRef<mirror::DexCache> dex_cache(self, dex_caches->Get(i));
    const std::string& dex_file_location(dex_cache->GetLocation()->ToModifiedUtf8());
    const OatFile::OatDexFile* oat_dex_file = oat_file.GetOatDexFile(dex_file_location,
    NULL);
    CHECK(oat_dex_file != NULL) << oat_file.GetLocation() << " " << dex_file_location;
    const DexFile* dex_file = oat_dex_file->OpenDexFile();
    if (dex_file == NULL) {
      LOG(FATAL) << "Failed to open dex file " << dex_file_location
                 << " from within oat file " << oat_file.GetLocation();
    }

    CHECK_EQ(dex_file->GetLocationChecksum(), oat_dex_file->GetDexFileLocationChecksum());
```

```
    AppendToBootClassPath(*dex_file, dex_cache);
  }

  // Set classes on AbstractMethod early so that IsMethod tests can be performed during
the live
  // bitmap walk.
  mirror::ArtMethod::SetClass(GetClassRoot(kJavaLangReflectArtMethod));

  // Set entry point to interpreter if in InterpretOnly mode.
  if (Runtime::Current()->GetInstrumentation()->InterpretOnly()) {
    ReaderMutexLock mu(self, *Locks::heap_bitmap_lock_);
    heap->FlushAllocStack();
    heap->GetLiveBitmap()->Walk(InitFromImageInterpretOnlyCallback, this);
  }

  // reinit class_roots_
  mirror::Class::SetClassClass(class_roots->Get(kJavaLangClass));
  class_roots_ = class_roots;

  // reinit array_iftable_ from any array class instance, they should be ==
  array_iftable_ = GetClassRoot(kObjectArrayClass)->GetIfTable();
  DCHECK(array_iftable_ == GetClassRoot(kBooleanArrayClass)->GetIfTable());
  // String class root was set above
  mirror::ArtField::SetClass(GetClassRoot(kJavaLangReflectArtField));
  mirror::BooleanArray::SetArrayClass(GetClassRoot(kBooleanArrayClass));
  mirror::ByteArray::SetArrayClass(GetClassRoot(kByteArrayClass));
  mirror::CharArray::SetArrayClass(GetClassRoot(kCharArrayClass));
  mirror::DoubleArray::SetArrayClass(GetClassRoot(kDoubleArrayClass));
  mirror::FloatArray::SetArrayClass(GetClassRoot(kFloatArrayClass));
  mirror::IntArray::SetArrayClass(GetClassRoot(kIntArrayClass));
  mirror::LongArray::SetArrayClass(GetClassRoot(kLongArrayClass));
  mirror::ShortArray::SetArrayClass(GetClassRoot(kShortArrayClass));
  mirror::Throwable::SetClass(GetClassRoot(kJavaLangThrowable));
  mirror::StackTraceElement::SetClass(GetClassRoot(kJavaLangStackTraceElement));

  FinishInit();

  VLOG(startup) << "ClassLinker::InitFromImage exiting";
}
```

在文件“art\art\runtime\class_linker.cc”中，通过函数 AppendToBootClassPath()和 RegisterDex FileLocked()将 dex cache 和 dex file 关联起来，同时把 dex file 注册到 boot_class_path_，将 dex cache 注册到 dex_caches_。函数 AppendToBootClassPath()和 RegisterDexFileLocked()的具体实现代码如下所示。

```
void ClassLinker::AppendToBootClassPath(const DexFile& dex_file, SirtRef<mirror::
DexCache>& dex_cache) {
  CHECK(dex_cache.get() != NULL) << dex_file.GetLocation();
  boot_class_path_.push_back(&dex_file);
  RegisterDexFile(dex_file, dex_cache);
}
void ClassLinker::RegisterDexFileLocked(const DexFile& dex_file, SirtRef<mirror::
DexCache>& dex_cache) {
  dex_lock_.AssertExclusiveHeld(Thread::Current());
  CHECK(dex_cache.get() != NULL) << dex_file.GetLocation();
  CHECK(dex_cache->GetLocation()->Equals(dex_file.GetLocation()))
      << dex_cache->GetLocation()->ToModifiedUtf8() << " " << dex_file.GetLocation();
  dex_caches_.push_back(dex_cache.get());
  dex_cache->SetDexFile(&dex_file);
  dex_caches_dirty_ = true;
}
```

当注册上述信息后，在 ClassLinker 调用 FindClass()函数时会用到。执行完 Runtime 中的函数 Create()和函数 Init()后，在 JNI_CreateJavaVM 函数中 Runtime 的 Start()函数被调用。

20.2.7 初始化类、方法和域

在文件 runtime.cc 的函数 InitNativeMethods()中分别调用函数 JniConstants::init()和函数

WellKnownClasses::Init()。函数 InitNativeMethods()的具体实现代码如下所示。

```
void Runtime::InitNativeMethods() {
  VLOG(startup) << "Runtime::InitNativeMethods entering";
  Thread* self = Thread::Current();
  JNIEnv* env = self->GetJniEnv();

  // Must be in the kNative state for calling native methods (JNI_OnLoad code).
  CHECK_EQ(self->GetState(), kNative);

  // First set up JniConstants, which is used by both the runtime's built-in native
  // methods and libcore.
  JniConstants::init(env);
  WellKnownClasses::Init(env);

  // Then set up the native methods provided by the runtime itself.
  RegisterRuntimeNativeMethods(env);
  {
    std::string mapped_name(StringPrintf(OS_SHARED_LIB_FORMAT_STR, "javacore"));
    std::string reason;
    self->TransitionFromSuspendedToRunnable();
    if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
      LOG(FATAL) << "LoadNativeLibrary failed for \"" << mapped_name << "\": " << reason;
    }
    self->TransitionFromRunnableToSuspended(kNative);
  }

  // Initialize well known classes that may invoke runtime native methods.
  WellKnownClasses::LateInit(env);

  VLOG(startup) << "Runtime::InitNativeMethods exiting";
}
```

函数 JniConstants::init()在文件“libnativehelper/JniConstants.cpp”中定义，WellKnownClasses::Init()在文件“art/runtime/well_known_classes.cc”中定义，这两个函数的具体实现代码如下所示。

```
void JniConstants::init(JNIEnv* env) {
    bidiRunClass = findClass(env, "java/text/Bidi$Run");
    bigDecimalClass = findClass(env, "java/math/BigDecimal");
    booleanClass = findClass(env, "java/lang/Boolean");
    byteClass = findClass(env, "java/lang/Byte");
    byteArrayClass = findClass(env, "[B");
    calendarClass = findClass(env, "java/util/Calendar");
    characterClass = findClass(env, "java/lang/Character");
    charsetICUClass = findClass(env, "java/nio/charset/CharsetICU");
    constructorClass = findClass(env, "java/lang/reflect/Constructor");
    floatClass = findClass(env, "java/lang/Float");
    deflaterClass = findClass(env, "java/util/zip/Deflater");
    doubleClass = findClass(env, "java/lang/Double");
    errnoExceptionClass = findClass(env, "libcore/io/ErrnoException");
    fieldClass = findClass(env, "java/lang/reflect/Field");
    fieldPositionIteratorClass = findClass(env, "libcore/icu/NativeDecimalFormat$
    FieldPositionIterator");
    fileDescriptorClass = findClass(env, "java/io/FileDescriptor");
    gaiExceptionClass = findClass(env, "libcore/io/GaiException");
    inet6AddressClass = findClass(env, "java/net/Inet6Address");
    inetAddressClass = findClass(env, "java/net/InetAddress");
    inetSocketAddressClass = findClass(env, "java/net/InetSocketAddress");
    inetUnixAddressClass = findClass(env, "java/net/InetUnixAddress");
    inflaterClass = findClass(env, "java/util/zip/Inflater");
    inputStreamClass = findClass(env, "java/io/InputStream");
    integerClass = findClass(env, "java/lang/Integer");
    localeDataClass = findClass(env, "libcore/icu/LocaleData");
    longClass = findClass(env, "java/lang/Long");
    methodClass = findClass(env, "java/lang/reflect/Method");
    mutableIntClass = findClass(env, "libcore/util/MutableInt");
    mutableLongClass = findClass(env, "libcore/util/MutableLong");
    objectClass = findClass(env, "java/lang/Object");
    objectArrayClass = findClass(env, "[Ljava/lang/Object;");
    outputStreamClass = findClass(env, "java/io/OutputStream");
    parsePositionClass = findClass(env, "java/text/ParsePosition");
```

```
    patternSyntaxExceptionClass = findClass(env, "java/util/regex/PatternSyntaxException");
    realToStringClass = findClass(env, "java/lang/RealToString");
    referenceClass = findClass(env, "java/lang/ref/Reference");
    shortClass = findClass(env, "java/lang/Short");
    socketClass = findClass(env, "java/net/Socket");
    socketImplClass = findClass(env, "java/net/SocketImpl");
    stringClass = findClass(env, "java/lang/String");
    structAddrinfoClass = findClass(env, "libcore/io/StructAddrinfo");
    structFlockClass = findClass(env, "libcore/io/StructFlock");
    structGroupReqClass = findClass(env, "libcore/io/StructGroupReq");
    structLingerClass = findClass(env, "libcore/io/StructLinger");
    structPasswdClass = findClass(env, "libcore/io/StructPasswd");
    structPollfdClass = findClass(env, "libcore/io/StructPollfd");
    structStatClass = findClass(env, "libcore/io/StructStat");
    structStatVfsClass = findClass(env, "libcore/io/StructStatVfs");
    structTimevalClass = findClass(env, "libcore/io/StructTimeval");
    structUcredClass = findClass(env, "libcore/io/StructUcred");
    structUtsnameClass = findClass(env, "libcore/io/StructUtsname");
}
void WellKnownClasses::Init(JNIEnv* env) {
  com_android_dex_Dex = CacheClass(env, "com/android/dex/Dex");
  dalvik_system_PathClassLoader = CacheClass(env, "dalvik/system/PathClassLoader");
  java_lang_ClassLoader = CacheClass(env, "java/lang/ClassLoader");
  java_lang_ClassNotFoundException = CacheClass(env, "java/lang/ClassNotFoundException");
  java_lang_Daemons = CacheClass(env, "java/lang/Daemons");
  java_lang_Object = CacheClass(env, "java/lang/Object");
  java_lang_Error = CacheClass(env, "java/lang/Error");
  java_lang_reflect_AbstractMethod = CacheClass(env, "java/lang/reflect/AbstractMethod");
  java_lang_reflect_ArtMethod = CacheClass(env, "java/lang/reflect/ArtMethod");
  java_lang_reflect_Constructor = CacheClass(env, "java/lang/reflect/Constructor");
  java_lang_reflect_Field = CacheClass(env, "java/lang/reflect/Field");
  java_lang_reflect_Method = CacheClass(env, "java/lang/reflect/Method");
  java_lang_reflect_Proxy = CacheClass(env, "java/lang/reflect/Proxy");
  java_lang_RuntimeException = CacheClass(env, "java/lang/RuntimeException");
  java_lang_StackOverflowError = CacheClass(env, "java/lang/StackOverflowError");
  java_lang_System = CacheClass(env, "java/lang/System");
  java_lang_Thread = CacheClass(env, "java/lang/Thread");
  java_lang_Thread$UncaughtExceptionHandler = CacheClass(env, "java/lang/Thread$
UncaughtExceptionHandler");
  java_lang_ThreadGroup = CacheClass(env, "java/lang/ThreadGroup");
  java_lang_Throwable = CacheClass(env, "java/lang/Throwable");
  java_nio_DirectByteBuffer = CacheClass(env, "java/nio/DirectByteBuffer");
  org_apache_harmony_dalvik_ddmc_Chunk = CacheClass(env, "org/apache/harmony/dalvik/
ddmc/Chunk");
  org_apache_harmony_dalvik_ddmc_DdmServer = CacheClass(env, "org/apache/harmony/
dalvik/ddmc/DdmServer");

  com_android_dex_Dex_create = CacheMethod(env, com_android_dex_Dex, true, "create",
"(Ljava/nio/ByteBuffer;)Lcom/android/dex/Dex;");
  java_lang_ClassNotFoundException_init = CacheMethod(env, java_lang_ClassNotFound
Exception, false, "<init>", "(Ljava/lang/String;Ljava/lang/Throwable;)V");
  java_lang_ClassLoader_loadClass = CacheMethod(env, java_lang_ClassLoader, false,
"loadClass", "(Ljava/lang/String;)Ljava/lang/Class;");

  java_lang_Daemons_requestGC = CacheMethod(env, java_lang_Daemons, true, "requestGC",
"()V");
  java_lang_Daemons_requestHeapTrim = CacheMethod(env, java_lang_Daemons, true,
"requestHeapTrim", "()V");
  java_lang_Daemons_start = CacheMethod(env, java_lang_Daemons, true, "start", "()V");

  ScopedLocalRef<jclass> java_lang_ref_FinalizerReference(env, env->FindClass("java/
lang/ref/FinalizerReference"));
  java_lang_ref_FinalizerReference_add = CacheMethod(env, java_lang_ref_Finalizer
Reference.get(), true, "add", "(Ljava/lang/Object;)V");
  ScopedLocalRef<jclass> java_lang_ref_ReferenceQueue(env, env->FindClass("java/lang/
ref/ReferenceQueue"));
  java_lang_ref_ReferenceQueue_add = CacheMethod(env, java_lang_ref_ReferenceQueue.
get(), true, "add", "(Ljava/lang/ref/Reference;)V");

  java_lang_reflect_Proxy_invoke = CacheMethod(env, java_lang_reflect_Proxy, true,
"invoke", "(Ljava/lang/reflect/Proxy;Ljava/lang/reflect/ArtMethod;[Ljava/lang/Object;]
```

```
Ljava/lang/Object;");
  java_lang_Thread_init = CacheMethod(env, java_lang_Thread, false, "<init>", "(Ljava/
lang/ThreadGroup;Ljava/lang/String;IZ)V");
  java_lang_Thread_run = CacheMethod(env, java_lang_Thread, false, "run", "()V");
  java_lang_Thread$UncaughtExceptionHandler_uncaughtException = CacheMethod(env,
java_lang_Thread$UncaughtExceptionHandler, false, "uncaughtException", "(Ljava/lang/
Thread;Ljava/lang/Throwable;)V");
  java_lang_ThreadGroup_removeThread = CacheMethod(env, java_lang_ThreadGroup, false,
"removeThread", "(Ljava/lang/Thread;)V");
  java_nio_DirectByteBuffer_init = CacheMethod(env, java_nio_DirectByteBuffer, false,
"<init>", "(JI)V");
  org_apache_harmony_dalvik_ddmc_DdmServer_broadcast = CacheMethod(env, org_apache_
harmony_dalvik_ddmc_DdmServer, true, "broadcast", "(I)V");
  org_apache_harmony_dalvik_ddmc_DdmServer_dispatch = CacheMethod(env, org_apache_
harmony_dalvik_ddmc_DdmServer, true, "dispatch", "([I[BII]Lorg/apache/harmony/dalvik/
ddmc/Chunk;");

  java_lang_Thread_daemon = CacheField(env, java_lang_Thread, false, "daemon", "Z");
  java_lang_Thread_group = CacheField(env, java_lang_Thread, false, "group", "Ljava/
lang/ThreadGroup;");
  java_lang_Thread_lock = CacheField(env, java_lang_Thread, false, "lock", "Ljava/
lang/Object;");
  java_lang_Thread_name = CacheField(env, java_lang_Thread, false, "name", "Ljava/
lang/String;");
  java_lang_Thread_priority = CacheField(env, java_lang_Thread, false, "priority", "I");
  java_lang_Thread_uncaughtHandler = CacheField(env, java_lang_Thread, false,
"uncaughtHandler", "Ljava/lang/Thread$UncaughtExceptionHandler;");
  java_lang_Thread_nativePeer = CacheField(env, java_lang_Thread, false, "nativePeer",
"I");
  java_lang_ThreadGroup_mainThreadGroup = CacheField(env, java_lang_ThreadGroup, true,
"mainThreadGroup", "Ljava/lang/ThreadGroup;");
  java_lang_ThreadGroup_name = CacheField(env, java_lang_ThreadGroup, false, "name",
"Ljava/lang/String;");
  java_lang_ThreadGroup_systemThreadGroup = CacheField(env, java_lang_ThreadGroup, true,
"systemThreadGroup", "Ljava/lang/ThreadGroup;");
  java_lang_reflect_AbstractMethod_artMethod = CacheField(env, java_lang_reflect_
AbstractMethod, false, "artMethod", "Ljava/lang/reflect/ArtMethod;");
  java_lang_reflect_Field_artField = CacheField(env, java_lang_reflect_Field, false,
"artField", "Ljava/lang/reflect/ArtField;");
  java_lang_reflect_Proxy_h = CacheField(env, java_lang_reflect_Proxy, false, "h",
"Ljava/lang/reflect/InvocationHandler;");
  java_nio_DirectByteBuffer_capacity = CacheField(env, java_nio_DirectByteBuffer, false,
"capacity", "I");
  java_nio_DirectByteBuffer_effectiveDirectAddress = CacheField(env, java_nio_
DirectByteBuffer, false, "effectiveDirectAddress", "J");
  org_apache_harmony_dalvik_ddmc_Chunk_data = CacheField(env, org_apache_harmony_
dalvik_ddmc_Chunk, false, "data", "[B");
  org_apache_harmony_dalvik_ddmc_Chunk_length = CacheField(env, org_apache_harmony_
dalvik_ddmc_Chunk, false, "length", "I");
  org_apache_harmony_dalvik_ddmc_Chunk_offset = CacheField(env, org_apache_harmony_
dalvik_ddmc_Chunk, false, "offset", "I");
  org_apache_harmony_dalvik_ddmc_Chunk_type = CacheField(env, org_apache_harmony_
dalvik_ddmc_Chunk, false, "type", "I");

  java_lang_Boolean_valueOf = CachePrimitiveBoxingMethod(env, 'Z', "java/lang/Boolean");
  java_lang_Byte_valueOf = CachePrimitiveBoxingMethod(env, 'B', "java/lang/Byte");
  java_lang_Character_valueOf = CachePrimitiveBoxingMethod(env, 'C', "java/lang/
Character");
  java_lang_Double_valueOf = CachePrimitiveBoxingMethod(env, 'D', "java/lang/Double");
  java_lang_Float_valueOf = CachePrimitiveBoxingMethod(env, 'F', "java/lang/Float");
  java_lang_Integer_valueOf = CachePrimitiveBoxingMethod(env, 'I', "java/lang/Integer");
  java_lang_Long_valueOf = CachePrimitiveBoxingMethod(env, 'J', "java/lang/Long");
  java_lang_Short_valueOf = CachePrimitiveBoxingMethod(env, 'S', "java/lang/Short");
}
```

通过上述代码可知，通过 FindClass()、GetStaticFieldID()和 GetStaticMethodID()等函数分别初始化了系统基本类、方法和域，这一些都是最基本的类。

然后 RegisterRuntimeNativeMethods()函数注册了系列系统类中的 Native 函数。

```
void Runtime::RegisterRuntimeNativeMethods(JNIEnv* env) {
```

```
#define REGISTER(FN) extern void FN(JNIEnv*); FN(env)
  // Register Throwable first so that registration of other native methods can throw
exceptions
  REGISTER(register_java_lang_Throwable);
  …
```

接着函数 InitNativeMethods()会载入 libjavacore.so 这个库，单独载入是因为它本身包含了 System.loadLibrary()实现，不先载入会出现顺序紊乱问题。

```
if (!instance_->java_vm_->LoadNativeLibrary(mapped_name, NULL, reason)) {
```

再次回到 Runtime::Start()函数进行线程初始化，再判断是否为 Zygote 进程。如果是则调用 InitZygote()进行初始化（其中主要是 mount 一些文件系统）操作，否则调用 DidForkFromZygote()函数。函数 DidForkFromZygote()会创建线程池，创建 signalcatcher 线程和启动 JDWP 调试线程）。这个函数主要工作是调用 Heap 对象的 CreateThreadPool()函数来创建线程池。函数 DidForkFromZygote 在文件 Runtime:.cc 中定义，具体实现代码如下所示。

```
void Runtime::DidForkFromZygote() {
  is_zygote_ = false;

  // Create the thread pool.
  heap_->CreateThreadPool();

  StartSignalCatcher();

  // Start the JDWP thread. If the command-line debugger flags specified "suspend=y",
  // this will pause the runtime, so we probably want this to come last.
  Dbg::StartJdwp();
}
```

最后 Start()函数中调用了 StartDaemonThreads()函数，这个函数的工作是调用 Java 类 Daemons 的 start()方法来启动一些 Deamon 线程，这个过程实际上和 Dalvik 启动时完成的最后一项工作相同。函数 Runtime::在文件 Runtime.cc 中定义，具体实现代码如下所示。

```
void Runtime::StartDaemonThreads() {
  VLOG(startup) << "Runtime::StartDaemonThreads entering";

  Thread* self = Thread::Current();

  // Must be in the kNative state for calling native methods.
  CHECK_EQ(self->GetState(), kNative);

  JNIEnv* env = self->GetJniEnv();
  env->CallStaticVoidMethod(WellKnownClasses::java_lang_Daemons,
                            WellKnownClasses::java_lang_Daemons_start);
  if (env->ExceptionCheck()) {
    env->ExceptionDescribe();
    LOG(FATAL) << "Error starting java.lang.Daemons";
  }

  VLOG(startup) << "Runtime::StartDaemonThreads exiting";
}
```

然后启动后台线程，用 JNI 调用 java.lang.ClassLoader.getSystemClassLoader()得到系统的 Class Loader（由 createSystemClassLoader()创建），调用 com.android.internal.os.ZygoteInit.main()时用的就是它。

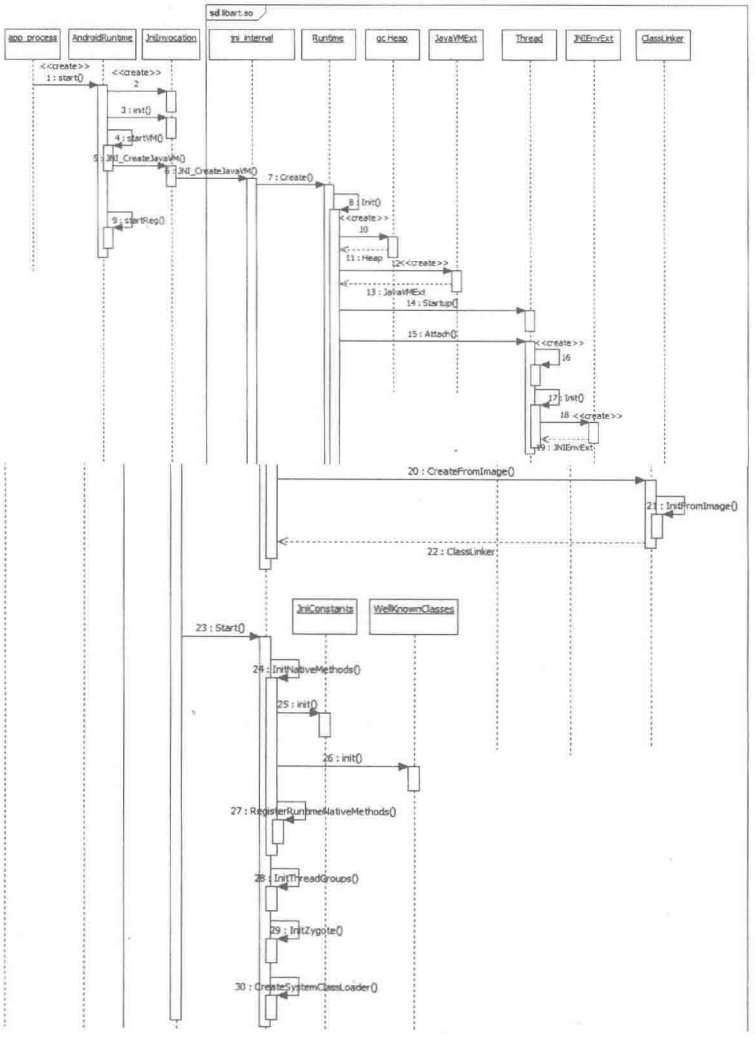
```
StartDaemonThreads();
system_class_loader_ = CreateSystemClassLoader();
```

从 StartVM()返回后，AndroidRuntime 执行 startReg()在创建线程时加一个 hook 函数，这样每个 Thread 起来时会先去执行 AndroidRuntime::javaThreadShell()，而该函数会初始化 Java 虚拟机环境，这样新创建的线程就可以调用 Java 层了。函数 startReg 在文件 AndroidRuntime.cpp 中定义，具体实现代码如下所示。

```
int AndroidRuntime::startReg(JNIEnv* env)
{
```

```
    androidSetCreateThreadFunc((android_create_thread_fn) javaCreateThreadEtc);

    ALOGV("--- registering native functions ---\n");
    env->PushLocalFrame(200);

    if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
        env->PopLocalFrame(NULL);
        return -1;
    }
    env->PopLocalFrame(NULL);

    //createJavaThread("fubar", quickTest, (void*) "hello");

    return 0;
}

AndroidRuntime* AndroidRuntime::getRuntime()
{
    return gCurRuntime;
}
```

到此为止，AndroidRuntime 中的 start()函数的执行过程全部讲解完毕。总结的执行流程如图 20-6 所示。



▲图 20-6　函数 start()的执行流程

20.3 分析主函数 main

ART 的基本初始化工作完成后，接下来开始执行主函数，具体步骤如下所示。

- 先通过 FindClass()找到相应的类。
- 然后通过 GetStaticMethodID()找到相应的方法。
- 最后调用 CallStaticVoidMethod()进入 Java 世界执行托管代码工作。

在本节的内容中，将以 Zygote 初始化操作为例进行讲解，其中类名为 com.android.internal.os.ZygoteInit，方法为 main，将详细分析执行 ART 主函数的具体过程，为读者步入本书后面知识的学习打下基础。

在文件 frameworks/base/core/jni/AndroidRuntime.cpp 中，通过 AndroidRuntime::start 中如下所示的代码调用 startVM()启动虚拟机，然后调用 startReg()注册 JNI 方法，并调用 com.android.internal.os.ZygoteInit 类的 main 函数。

```
    /*
     * Start VM.  This thread becomes the main thread of the VM, and will
     * not return until the VM exits.
     */
    char* slashClassName = toSlashClassName(className);
    jclass startClass = env->FindClass(slashClassName);
    if (startClass == NULL) {
        ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
        /* keep going */
    } else {
        jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
            "([Ljava/lang/String;)V");
        if (startMeth == NULL) {
            ALOGE("JavaVM unable to find main() in '%s'\n", className);
            /* keep going */
        } else {
            env->CallStaticVoidMethod(startClass, startMeth, strArray);

#if 0
            if (env->ExceptionCheck())
                threadExitUncaughtException(env);
#endif
        }
    }
```

在上述代码中涉及了 3 个函数：FindClass()、GetStaticMethodID()和 CallStaticVoidMethod()。函数 FindClass()在文件“/art/runtime/jni_internal.cc”中实现，具体实现代码如下所示。

```
static jclass FindClass(JNIEnv* env, const char* name) {
  CHECK_NON_NULL_ARGUMENT(FindClass, name);
  Runtime* runtime = Runtime::Current();
  ClassLinker* class_linker = runtime->GetClassLinker();
  std::string descriptor(NormalizeJniClassDescriptor(name));
  ScopedObjectAccess soa(env);
  Class* c = NULL;
  if (runtime->IsStarted()) {
    ClassLoader* cl = GetClassLoader(soa);
    c = class_linker->FindClass(descriptor.c_str(), cl);
  } else {
    c = class_linker->FindSystemClass(descriptor.c_str());
  }
  return soa.AddLocalReference<jclass>(c);
}
```

函数 GetClassLoader()也是在文件“art/runtime/jni_internal.cc”中定义，功能是调用 GetSystemClassLoader()得到前面初始化好的系统 ClassLoader，具体实现代码如下所示。

```
static ClassLoader* GetClassLoader(const ScopedObjectAccess& soa)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  ArtMethod* method = soa.Self()->GetCurrentMethod(NULL);
```

```
  // If we are running Runtime.nativeLoad, use the overriding ClassLoader it set.
  if (method == soa.DecodeMethod(WellKnownClasses::java_lang_Runtime_nativeLoad)) {
    return soa.Self()->GetClassLoaderOverride();
  }
  // If we have a method, use its ClassLoader for context.
  if (method != NULL) {
    return method->GetDeclaringClass()->GetClassLoader();
  }
  // We don't have a method, so try to use the system ClassLoader.
  ClassLoader* class_loader = soa.Decode<ClassLoader*>(Runtime::Current()->GetSystem
ClassLoader());
  if (class_loader != NULL) {
    return class_loader;
  }
  // See if the override ClassLoader is set for gtests.
  class_loader = soa.Self()->GetClassLoaderOverride();
  if (class_loader != NULL) {
    // If so, CommonTest should have set UseCompileTimeClassPath.
    CHECK(Runtime::Current()->UseCompileTimeClassPath());
    return class_loader;
  }
  // Use the BOOTCLASSPATH.
  return NULL;
}
```

20.4 查找目标类

开始调用 ClassLinker 的函数 FindClass()查找目标类，在这一过程了涉及的关键函数有：LookupClass()、DefineClass()、InsertClass()、LoadClass()和 LinkClass()，上述关键函数的具体说明如下所示。

20.4.1　函数 LookupClass()

函数 LookupClass()在文件“art/runtime/class_linker.cc”中定义，先在 ClassLinker 的成员变量 class_table_中找指定类，找到就返回，若找不到就判断是否要在 image 中找（class_loader 为 NULL 且 dex_cache_image_class_lookup_required 为 true）。如果是，就调用 LookupClassFromImage()在 Image 中进行查找，找到了就调用 InsertClass()将找到的类放入到 class_table_中方便下次查找。

函数 LookupClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::FindClass(const char* descriptor, mirror::ClassLoader*
class_loader) {
  DCHECK_NE(*descriptor, '\0') << "descriptor is empty string";
  Thread* self = Thread::Current();
  DCHECK(self != NULL);
  self->AssertNoPendingException();
  if (descriptor[1] == '\0') {
    // only the descriptors of primitive types should be 1 character long, also avoid class lookup
    // for primitive classes that aren't backed by dex files.
    return FindPrimitiveClass(descriptor[0]);
  }
  // Find the class in the loaded classes table.
  mirror::Class* klass = LookupClass(descriptor, class_loader);
  if (klass != NULL) {
    return EnsureResolved(self, klass);
  }
  // Class is not yet loaded.
  if (descriptor[0] == '[') {
    return CreateArrayClass(descriptor, class_loader);

  } else if (class_loader == NULL) {
    DexFile::ClassPathEntry pair = DexFile::FindInClassPath(descriptor, boot_class_path_);
    if (pair.second != NULL) {
      return DefineClass(descriptor, NULL, *pair.first, *pair.second);
    }
```

```
  } else if (Runtime::Current()->UseCompileTimeClassPath()) {
    // First try the boot class path, we check the descriptor first to avoid an unnecessary
    // throw of a NoClassDefFoundError.
    if (IsInBootClassPath(descriptor)) {
      mirror::Class* system_class = FindSystemClass(descriptor);
      CHECK(system_class != NULL);
      return system_class;
    }
    // Next try the compile time class path.
    const std::vector<const DexFile*>* class_path;
    {
      ScopedObjectAccessUnchecked soa(self);
      ScopedLocalRef<jobject> jclass_loader(soa.Env(), soa.AddLocalReference<jobject>
      (class_loader));
      class_path = &Runtime::Current()->GetCompileTimeClassPath(jclass_loader.get());
    }

    DexFile::ClassPathEntry pair = DexFile::FindInClassPath(descriptor, *class_path);
    if (pair.second != NULL) {
      return DefineClass(descriptor, class_loader, *pair.first, *pair.second);
    }

  } else {
    ScopedObjectAccessUnchecked soa(self->GetJniEnv());
    ScopedLocalRef<jobject> class_loader_object(soa.Env(),
                                                soa.AddLocalReference<jobject>(class_loader));
    std::string class_name_string(DescriptorToDot(descriptor));
    ScopedLocalRef<jobject> result(soa.Env(), NULL);
    {
      ScopedThreadStateChange tsc(self, kNative);
      ScopedLocalRef<jobject> class_name_object(soa.Env(),
soa.Env()->NewStringUTF(class_name_string.c_str()));
      if (class_name_object.get() == NULL) {
        return NULL;
      }
      CHECK(class_loader_object.get() != NULL);
      result.reset(soa.Env()->CallObjectMethod(class_loader_object.get(),
                           WellKnownClasses::java_lang_ClassLoader_ loadClass,
                                               class_name_object.get()));
    }
    if (soa.Self()->IsExceptionPending()) {
      // If the ClassLoader threw, pass that exception up.
      return NULL;
    } else if (result.get() == NULL) {
      // broken loader - throw NPE to be compatible with Dalvik
      ThrowNullPointerException(NULL, StringPrintf("ClassLoader.loadClass returned null
      for %s", class_name_string.c_str()).c_str());
      return NULL;
    } else {
      // success, return mirror::Class*
      return soa.Decode<mirror::Class*>(result.get());
    }
  }

  ThrowNoClassDefFoundError("Class %s not found", PrintableString(descriptor).c_str());
  return NULL;
}
```

函数 LookupClassFromImage()也在文件“art/runtime/class_linker.cc”中定义，具体实现代码如下所示。

```
mirror::Class* ClassLinker::LookupClassFromImage(const char* descriptor) {
  Thread* self = Thread::Current();
  const char* old_no_suspend_cause =
      self->StartAssertNoThreadSuspension("Image class lookup");
  mirror::ObjectArray<mirror::DexCache>* dex_caches = GetImageDexCaches();
  for (int32_t i = 0; i < dex_caches->GetLength(); ++i) {
    mirror::DexCache* dex_cache = dex_caches->Get(i);
    const DexFile* dex_file = dex_cache->GetDexFile();
    // First search using the class def map, but don't bother for non-class types.
    if (descriptor[0] == 'L') {
```

```
      const DexFile::StringId* descriptor_string_id = dex_file->FindStringId (descriptor);
      if (descriptor_string_id != NULL) {
        const DexFile::TypeId* type_id =
            dex_file->FindTypeId(dex_file->GetIndexForStringId(*descriptor_string_id));
        if (type_id != NULL) {
          mirror::Class* klass = dex_cache->GetResolvedType(dex_file->GetIndexForTypeId
          (*type_id));
          if (klass != NULL) {
            self->EndAssertNoThreadSuspension(old_no_suspend_cause);
            return klass;
          }
        }
      }
    }
    // Now try binary searching the string/type index.
    const DexFile::StringId* string_id = dex_file->FindStringId(descriptor);
    if (string_id != NULL) {
      const DexFile::TypeId* type_id =
          dex_file->FindTypeId(dex_file->GetIndexForStringId(*string_id));
      if (type_id != NULL) {
        uint16_t type_idx = dex_file->GetIndexForTypeId(*type_id);
        mirror::Class* klass = dex_cache->GetResolvedType(type_idx);
        if (klass != NULL) {
          self->EndAssertNoThreadSuspension(old_no_suspend_cause);
          return klass;
        }
      }
    }
  }
  self->EndAssertNoThreadSuspension(old_no_suspend_cause);
  return NULL;
}
```

20.4.2　函数 DefineClass()

函数 DefineClass()在文件“art/runtime/class_linker.cc”中定义，实现 LoadClass()、InsertClass()和 LinkClass()等动作。其中，LoadClass()调用 LoadField()和 LoadMethod()等函数把类中的域和方法数据从 dex 文件中读出来，填入 Class 结构。

函数 DefineClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::DefineClass(const char* descriptor,
                                        mirror::ClassLoader* class_loader,
                                        const DexFile& dex_file,
                                        const DexFile::ClassDef& dex_class_def) {
  Thread* self = Thread::Current();
  SirtRef<mirror::Class> klass(self, NULL);
  // Load the class from the dex file.
  if (UNLIKELY(!init_done_)) {
    // finish up init of hand crafted class_roots_
    if (strcmp(descriptor, "Ljava/lang/Object;") == 0) {
      klass.reset(GetClassRoot(kJavaLangObject));
    } else if (strcmp(descriptor, "Ljava/lang/Class;") == 0) {
      klass.reset(GetClassRoot(kJavaLangClass));
    } else if (strcmp(descriptor, "Ljava/lang/String;") == 0) {
      klass.reset(GetClassRoot(kJavaLangString));
    } else if (strcmp(descriptor, "Ljava/lang/DexCache;") == 0) {
      klass.reset(GetClassRoot(kJavaLangDexCache));
    } else if (strcmp(descriptor, "Ljava/lang/reflect/ArtField;") == 0) {
      klass.reset(GetClassRoot(kJavaLangReflectArtField));
    } else if (strcmp(descriptor, "Ljava/lang/reflect/ArtMethod;") == 0) {
      klass.reset(GetClassRoot(kJavaLangReflectArtMethod));
    } else {
      klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
    }
  } else {
    klass.reset(AllocClass(self, SizeOfClass(dex_file, dex_class_def)));
  }
  if (UNLIKELY(klass.get() == NULL)) {
    CHECK(self->IsExceptionPending());  // Expect an OOME.
```

```
    return NULL;
  }
  klass->SetDexCache(FindDexCache(dex_file));
  LoadClass(dex_file, dex_class_def, klass, class_loader);
  // Check for a pending exception during load
  if (self->IsExceptionPending()) {
    klass->SetStatus(mirror::Class::kStatusError, self);
    return NULL;
  }
  ObjectLock lock(self, klass.get());
  klass->SetClinitThreadId(self->GetTid());
  {
    // Add the newly loaded class to the loaded classes table.
    mirror::Class* existing = InsertClass(descriptor, klass.get(), Hash(descriptor));
    if (existing != NULL) {
      // We failed to insert because we raced with another thread. Calling EnsureResolved
      may cause
      // this thread to block.
      return EnsureResolved(self, existing);
    }
  }
  // Finish loading (if necessary) by finding parents
  CHECK(!klass->IsLoaded());
  if (!LoadSuperAndInterfaces(klass, dex_file)) {
    // Loading failed.
    klass->SetStatus(mirror::Class::kStatusError, self);
    return NULL;
  }
  CHECK(klass->IsLoaded());
  // Link the class (if necessary)
  CHECK(!klass->IsResolved());
  if (!LinkClass(klass, NULL, self)) {
    // Linking failed.
    klass->SetStatus(mirror::Class::kStatusError, self);
    return NULL;
  }
  CHECK(klass->IsResolved());
  Dbg::PostClassPrepare(klass.get());

  return klass.get();
}
```

函数 LoadClass()也是在文件“art/runtime/class_linker.cc”中定义的，具体实现代码如下所示。

```
void ClassLinker::LoadClass(const DexFile& dex_file,
                            const DexFile::ClassDef& dex_class_def,
                            SirtRef<mirror::Class>& klass,
                            mirror::ClassLoader* class_loader) {
  CHECK(klass.get() != NULL);
  CHECK(klass->GetDexCache() != NULL);
  CHECK_EQ(mirror::Class::kStatusNotReady, klass->GetStatus());
  const char* descriptor = dex_file.GetClassDescriptor(dex_class_def);
  CHECK(descriptor != NULL);

  klass->SetClass(GetClassRoot(kJavaLangClass));
  uint32_t access_flags = dex_class_def.access_flags_;
  // Make sure that none of our runtime-only flags are set.
  CHECK_EQ(access_flags & ~kAccJavaFlagsMask, 0U);
  klass->SetAccessFlags(access_flags);
  klass->SetClassLoader(class_loader);
  DCHECK_EQ(klass->GetPrimitiveType(), Primitive::kPrimNot);
  klass->SetStatus(mirror::Class::kStatusIdx, NULL);

  klass->SetDexClassDefIndex(dex_file.GetIndexForClassDef(dex_class_def));
  klass->SetDexTypeIndex(dex_class_def.class_idx_);

  // Load fields fields.
  const byte* class_data = dex_file.GetClassData(dex_class_def);
  if (class_data == NULL) {
    return;  // no fields or methods - for example a marker interface
  }
  ClassDataItemIterator it(dex_file, class_data);
```

```
  Thread* self = Thread::Current();
  if (it.NumStaticFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* statics = AllocArtFieldArray(self, it.Num
    StaticFields());
    if (UNLIKELY(statics == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetSFields(statics);
  }
  if (it.NumInstanceFields() != 0) {
    mirror::ObjectArray<mirror::ArtField>* fields =
        AllocArtFieldArray(self, it.NumInstanceFields());
    if (UNLIKELY(fields == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetIFields(fields);
  }
  for (size_t i = 0; it.HasNextStaticField(); i++, it.Next()) {
    SirtRef<mirror::ArtField> sfield(self, AllocArtField(self));
    if (UNLIKELY(sfield.get() == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetStaticField(i, sfield.get());
    LoadField(dex_file, it, klass, sfield);
  }
  for (size_t i = 0; it.HasNextInstanceField(); i++, it.Next()) {
    SirtRef<mirror::ArtField> ifield(self, AllocArtField(self));
    if (UNLIKELY(ifield.get() == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetInstanceField(i, ifield.get());
    LoadField(dex_file, it, klass, ifield);
  }

  UniquePtr<const OatFile::OatClass> oat_class;
  if (Runtime::Current()->IsStarted() && !Runtime::Current()->UseCompileTimeClassPath
()) {
    oat_class.reset(GetOatClass(dex_file, klass->GetDexClassDefIndex()));
  }

  // Load methods.
  if (it.NumDirectMethods() != 0) {
    // TODO: append direct methods to class object
    mirror::ObjectArray<mirror::ArtMethod>* directs =
        AllocArtMethodArray(self, it.NumDirectMethods());
    if (UNLIKELY(directs == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetDirectMethods(directs);
  }
  if (it.NumVirtualMethods() != 0) {
    // TODO: append direct methods to class object
    mirror::ObjectArray<mirror::ArtMethod>* virtuals =
        AllocArtMethodArray(self, it.NumVirtualMethods());
    if (UNLIKELY(virtuals == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetVirtualMethods(virtuals);
  }
  size_t class_def_method_index = 0;
  for (size_t i = 0; it.HasNextDirectMethod(); i++, it.Next()) {
    SirtRef<mirror::ArtMethod> method(self, LoadMethod(self, dex_file, it, klass));
    if (UNLIKELY(method.get() == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
```

```
    }
    klass->SetDirectMethod(i, method.get());
    if (oat_class.get() != NULL) {
      LinkCode(method, oat_class.get(), class_def_method_index);
    }
    method->SetMethodIndex(class_def_method_index);
    class_def_method_index++;
  }
  for (size_t i = 0; it.HasNextVirtualMethod(); i++, it.Next()) {
    SirtRef<mirror::ArtMethod> method(self, LoadMethod(self, dex_file, it, klass));
    if (UNLIKELY(method.get() == NULL)) {
      CHECK(self->IsExceptionPending());  // OOME.
      return;
    }
    klass->SetVirtualMethod(i, method.get());
    DCHECK_EQ(class_def_method_index, it.NumDirectMethods() + i);
    if (oat_class.get() != NULL) {
      LinkCode(method, oat_class.get(), class_def_method_index);
    }
    class_def_method_index++;
  }
  DCHECK(!it.HasNext());
}
```

20.4.3 函数 InsertClass()

函数 InsertClass()在文件“art/runtime/class_linker.cc”中定义，主要功能是把该类写入class_table_中方便下次查找。函数 InsertClass()的具体实现代码如下所示。

```
mirror::Class* ClassLinker::InsertClass(const char* descriptor, mirror::Class* klass,
                                        size_t hash) {
  if (VLOG_IS_ON(class_linker)) {
    mirror::DexCache* dex_cache = klass->GetDexCache();
    std::string source;
    if (dex_cache != NULL) {
      source += " from ";
      source += dex_cache->GetLocation()->ToModifiedUtf8();
    }
    LOG(INFO) << "Loaded class " << descriptor << source;
  }
  WriterMutexLock mu(Thread::Current(), *Locks::classlinker_classes_lock_);
  mirror::Class* existing =
      LookupClassFromTableLocked(descriptor, klass->GetClassLoader(), hash);
  if (existing != NULL) {
    return existing;
  }
  if (kIsDebugBuild && klass->GetClassLoader() == NULL && dex_cache_image_class_lookup_
required_) {
    // Check a class loaded with the system class loader matches one in the image if the class
    // is in the image.
    existing = LookupClassFromImage(descriptor);
    if (existing != NULL) {
      CHECK(klass == existing);
    }
  }
  Runtime::Current()->GetHeap()->VerifyObject(klass);
  class_table_.insert(std::make_pair(hash, klass));
  class_table_dirty_ = true;
  return NULL;
}
```

20.4.4 函数 LinkClass()

函数 LinkClass()在文件“art/runtime/class_linke.cc”中定义，功能是动态绑定虚函数和接口函数，其调用结构如下所示。

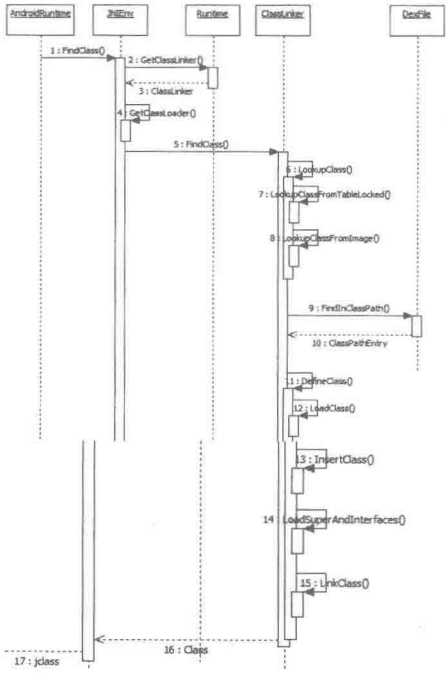
```
LinkSuperClass() // 检查父类
LinkMethods()
LinkVirtualMethods() // 结合父类进行虚函数绑定，填写 Class 中的虚函数表 vtable_
```

```
LinkInterfaceMethods() //处理接口类函数信息 iftable_。注意接口类中的虚函数也会影响虚函数表，因此会更新 vtable_
LinkInstanceFields() & LinkStaticFields() // 更新域信息，如域中的 Offset 和类的对象大小等
```

函数 LinkClass()的具体实现代码如下所示。

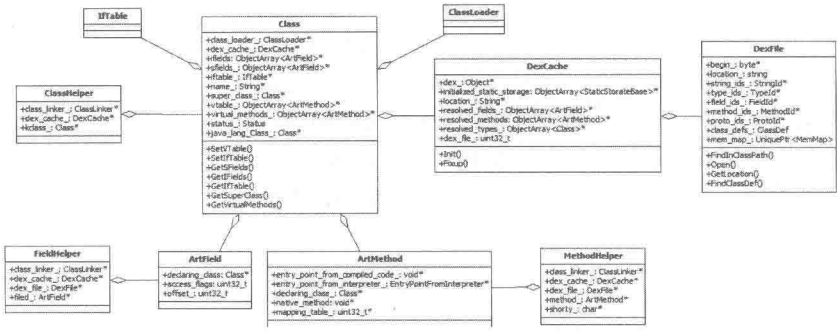
```
bool ClassLinker::LinkClass(SirtRef<mirror::Class>& klass,
                            mirror::ObjectArray<mirror::Class>* interfaces, Thread* self) {
  CHECK_EQ(mirror::Class::kStatusLoaded, klass->GetStatus());
  if (!LinkSuperClass(klass)) {
    return false;
  }
  if (!LinkMethods(klass, interfaces)) {
    return false;
  }
  if (!LinkInstanceFields(klass)) {
    return false;
  }
  if (!LinkStaticFields(klass)) {
    return false;
  }
  CreateReferenceInstanceOffsets(klass);
  CreateReferenceStaticOffsets(klass);
  CHECK_EQ(mirror::Class::kStatusLoaded, klass->GetStatus());
  klass->SetStatus(mirror::Class::kStatusResolved, self);
  return true;
}
```

对于函数 FindClass()来说，总共包含了内置类、启动类、系统类和其他类。其中内置类是很基本的类，一般是初始化时预加载好的（如 WellKnownClasses 和 JniConstants 中的程序），它们可以通过 LookupClassFromImage()函数找到。启动类是在 BOOTCLASSPATH 里的类，由于它们是启动类，所以这里还没有 ClassLoader。除掉前面的内置类，其余的通过 DexFile::FindInClassPath()查找得到。而系统类和其他类的加载过程是类似的，都是通过 ClassLoader 的 loadClass 方法，区别在于前者通过特殊的 SystemClassLoader 进行加载。例如，对于一个还没被加载过的启动类来说，一般流程如图 20-7 所示。



▲图 20-7　加载启动类的流程

整个过程涉及很多类，其中最主要的是 Class 类，具体结构如图 20-8 所示。



▲图 20-8 类结构关系

20.5 类操作

再次回到 FindClass()函数，因为在调用 ZygoteInit.main()时，所需的类在初始化时都已经装载链接好了，所以，此处不按照上面的流程进行，而是直接通过 JNI 调用 ClassLoader.loadClass()进行装载。完成后将找到的类转为 jclass 返回给 AndroidRuntime。类的查找工作结束后可以找相应的方法。GetStaticMethodID 会调用 FindMethodID()函数，它首先对该类进行验证，保证这个类是初始化好的，再调用其他函数进行目标函数的查找。

其中函数 GetStaticMethodID()在文件 jni_internal.cc 中定义，能够将未初始化的类初始化，获取静态函数 main 的 ID。具体实现代码如下所示。

```
static jmethodID GetStaticMethodID(JNIEnv* env, jclass java_class, const char* name,
                                   const char* sig) {
  CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, java_class);
  CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, name);
  CHECK_NON_NULL_ARGUMENT(GetStaticMethodID, sig);
  ScopedObjectAccess soa(env);
  return FindMethodID(soa, java_class, name, sig, true);
}
```

FindMethodID()函数也在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static jmethodID FindMethodID(ScopedObjectAccess& soa, jclass jni_class,
                              const char* name, const char* sig, bool is_static)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  Class* c = soa.Decode<Class*>(jni_class);
  if (!Runtime::Current()->GetClassLinker()->EnsureInitialized(c, true, true)) {
    return NULL;
  }

  ArtMethod* method = NULL;
  if (is_static) {
    method = c->FindDirectMethod(name, sig);
  } else {
    method = c->FindVirtualMethod(name, sig);
    if (method == NULL) {
      // No virtual method matching the signature.  Search declared
      // private methods and constructors.
      method = c->FindDeclaredDirectMethod(name, sig);
    }
  }

  if (method == NULL || method->IsStatic() != is_static) {
    ThrowNoSuchMethodError(soa, c, name, sig, is_static ? "static" : "non-static");
    return NULL;
  }
```

```
    return soa.EncodeMethod(method);
}
```

通过上述实现代码可知，会根据不同的需求执行不同的函数，具体说明如下所示。

- 如果要找的是静态函数（通过 GetStaticMethodID()过来的），则调用 FindDirectMethod()查找该类及其父类的非虚函数（通过 Class 的成员变量 direct_methods_）。
- 否则调用 FindVirtualMethod()查找该类及其父类的虚函数（通过 Class 的成员变量）virtual_methods_，如果没找到再调用 FindDeclaredDirectMethod()查找该类的非虚函数。找到的条件是函数名和函数签名相同，如这里是“main”和“([Ljava/lang/String;]V”。找到目标函数后，就可以执行了。

函数 FindDirectMethod()的具体定义代码如下所示。

```
-Method* Class::FindDirectMethod(const StringPiece& name,
-  const StringPiece& signature) {
-  for (Class* klass = this; klass != NULL; klass = klass->GetSuperClass()) {
+Method*  Class::FindDeclaredDirectMethod(const  DexCache*  dex_cache,  uint32_t  dex_
method_idx) const {
+  if (GetDexCache() == dex_cache) {
+    for (size_t i = 0; i < NumDirectMethods(); ++i) {
+      Method* method = GetDirectMethod(i);
+      if (method->GetDexMethodIndex() == dex_method_idx) {
+        return method;
+      }
+    }
+  }
+  return NULL;
+}
```

函数 FindDeclaredDirectMethod()的具体定义代码如下所示。

```
-Method* Class::FindDeclaredDirectMethod(const StringPiece& name,
-                                        const StringPiece& signature) {
+Method* Class::FindInterfaceMethod(const DexCache* dex_cache, uint32_t dex_method_idx)
const {
+  // Check the current class before checking the interfaces.
+  Method* method = FindDeclaredVirtualMethod(dex_cache, dex_method_idx);
+  if (method != NULL) {
+    return method;
+  }
+
+  int32_t iftable_count = GetIfTableCount();
+  ObjectArray<InterfaceEntry>* iftable = GetIfTable();
+  for (int32_t i = 0; i < iftable_count; i++) {
+  method = iftable->Get(i)->GetInterface()->FindVirtualMethod(dex_cache, dex_method_idx);
+    if (method != NULL) {
+      return method;
+    }
+  }
+  return NULL;
+}
```

20.6 实现托管操作

在执行托管操作时，函数 InvokeMain()会验证 Java 的 main 方法，并调用 CallStaticVoidMethod 来运行 main 方法。CallStaticVoidMethod(jni.h)在结构_JNIEnv 中实现，函数 CallStaticVoidMethod()在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static void CallStaticVoidMethod(JNIEnv* env, jclass, jmethodID mid, ...) {
    va_list ap;
    va_start(ap, mid);
    CHECK_NON_NULL_ARGUMENT(CallStaticVoidMethod, mid);
    ScopedObjectAccess soa(env);
    InvokeWithVarArgs(soa, NULL, mid, ap);
    va_end(ap);
}
```

在上述代码中，va_list 用于处理不定长度参数，function 表示 JNINativeInterface 结构指针表，用于保存 JNI 接口函数，例如调用 method 等。

函数 CallStaticVoidMethodV 在文件 JNI_interl.cc 中定义，具体实现代码如下所示。

```
static void CallStaticVoidMethodV(JNIEnv* env, jclass, jmethodID mid, va_list args) {
  CHECK_NON_NULL_ARGUMENT(CallStaticVoidMethodV, mid);
  ScopedObjectAccess soa(env);
  InvokeWithVarArgs(soa, NULL, mid, args);
}
```

函数 InvokeWithArgArray()也在文件 JNI_interl.cc 中定义，具体实现代码如下所示。

```
void InvokeWithArgArray(const ScopedObjectAccess& soa, ArtMethod* method,
                        ArgArray* arg_array, JValue* result, char result_type)
  SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  uint32_t* args = arg_array->GetArray();
  if (UNLIKELY(soa.Env()->check_jni)) {
    CheckMethodArguments(method, args);
  }
  method->Invoke(soa.Self(), args, arg_array->GetNumBytes(), result, result_type);
}
```

接下来执行文件“art/runtime/mirrorart_method.cc”中的函数 invoke，具体实现代码如下所示。

```
void ArtMethod::Invoke(Thread* self, uint32_t* args, uint32_t args_size, JValue* result,
                       char result_type) {
  if (kIsDebugBuild) {
    self->AssertThreadSuspensionIsAllowable(); //设定 Debug 时线程可以被 hold
    CHECK_EQ(kRunnable, self->GetState());
  }

  // Push a transition back into managed code onto the linked list in thread.
  ManagedStack fragment;
  self->PushManagedStackFragment(&fragment); //管理栈帧：this 放入 fragment,this 清空，保存现场

  Runtime* runtime = Runtime::Current();
  // Call the invoke stub, passing everything as arguments.
  if (UNLIKELY(!runtime->IsStarted())) {
    LOG(INFO) << "Not invoking " << PrettyMethod(this) << " for a runtime that isn't started";
    if (result != NULL) {
      result->SetJ(0);
    }
  } else {
    const bool kLogInvocationStartAndReturn = false;
    if (GetEntryPointFromCompiledCode() != NULL) {      //存在被编译的 code
      if (kLogInvocationStartAndReturn) {
        LOG(INFO) << StringPrintf("Invoking '%s' code=%p", PrettyMethod(this).c_str(),
GetEntryPointFromCompiledCode());
      }
#ifdef ART_USE_PORTABLE_COMPILER
      (*art_portable_invoke_stub)(this, args, args_size, self, result, result_type);
#else
      (*art_quick_invoke_stub)(this, args, args_size, self, result, result_type);
#endif
      if (UNLIKELY(reinterpret_cast<int32_t>(self->GetException(NULL)) == -1)) {
        // Unusual case where we were running LLVM generated code and an
        // exception was thrown to force the activations to be removed from the
        // stack. Continue execution in the interpreter.//llvm 生成代码过程中异常会进入解释器
        self->ClearException();
        ShadowFrame* shadow_frame = self->GetAndClearDeoptimizationShadowFrame(result);
        self->SetTopOfStack(NULL, 0);
        self->SetTopOfShadowStack(shadow_frame); //stack & shadow frame 设置
        interpreter::EnterInterpreterFromDeoptimize(self, shadow_frame, result);
        //解释器继续执行
      }
      if (kLogInvocationStartAndReturn) {
        LOG(INFO) << StringPrintf("Returned '%s' code=%p", PrettyMethod(this).c_str(),
        GetEntryPointFromCompiledCode());
      }
    } else {
```

```
      LOG(INFO) << "Not invoking '" << PrettyMethod(this)
          << "' code=" << reinterpret_cast<const void*>(GetEntryPointFromCompiledCode());
      if (result != NULL) {
        result->SetJ(0);
      }
    }
  }

  // Pop transition.
  self->PopManagedStackFragment(fragment); //恢复现场
}
```

在上述代码中，前后分别实现了对托管代码栈的保存和恢复工作。

接下来进入解释器的函数 EnterInterpreterFromDeoptimize()，具体实现代码如下所示。

```
void EnterInterpreterFromDeoptimize(Thread* self, ShadowFrame* shadow_frame, JValue*
ret_val)
    SHARED_LOCKS_REQUIRED(Locks::mutator_lock_) {
  JValue value;
  value.SetJ(ret_val->GetJ());  // Set value to last known result in case the shadow frame
chain is empty
  MethodHelper mh;                                          //method 操作的 class
  while (shadow_frame != NULL) {
    self->SetTopOfShadowStack(shadow_frame);
    mh.ChangeMethod(shadow_frame->GetMethod());                    /获取 method
    const DexFile::CodeItem* code_item = mh.GetCodeItem();          //code 传递
    value = Execute(self, mh, code_item, *shadow_frame, value);    //执行解释器
    ShadowFrame* old_frame = shadow_frame;
    shadow_frame = shadow_frame->GetLink();     //下一个 frame 赋值给 shadow_frame
    delete old_frame;                          //删掉前一个 frame
  }
  ret_val->SetJ(value.GetJ());
}
```

函数 Execute()在文件 interpretor.cc 中定义，具体实现代码如下所示。

```
static inline JValue Execute(Thread* self, MethodHelper& mh, const DexFile::CodeItem*
code_item,
                             ShadowFrame& shadow_frame, JValue result_register) {
  DCHECK(shadow_frame.GetMethod() == mh.GetMethod() ||
         shadow_frame.GetMethod()->GetDeclaringClass()->IsProxyClass());
  DCHECK(!shadow_frame.GetMethod()->IsAbstract());
  DCHECK(!shadow_frame.GetMethod()->IsNative());
  if (shadow_frame.GetMethod()->IsPreverified()) { //是否提前做过 method access verify
    // Enter the "without access check" interpreter.
    return ExecuteImpl<false>(self, mh, code_item, shadow_frame, result_register);
//进入具体实现函数，可以发现是个 C 语言解释器
  } else {
    // Enter the "with access check" interpreter.
    return ExecuteImpl<true>(self, mh, code_item, shadow_frame, result_register);
//进入具体实现函数，是一个 C 语言解释器
  }
}
```

再回到前面文件“art/runtime/mirrorart_method.cc”中的函数 invoke。

```
const bool kLogInvocationStartAndReturn = false;
if (GetEntryPointFromCompiledCode() != NULL) {      //存在被编译的 code
    if (kLogInvocationStartAndReturn) {
      LOG(INFO) << StringPrintf("Invoking '%s' code=%p", PrettyMethod(this).c_str(),
      GetEntryPointFromCompiledCode());
    }
#ifdef ART_USE_PORTABLE_COMPILER          //portable 编译器
    (*art_portable_invoke_stub)(this, args, args_size, self, result, result_type);
#else
    (*art_quick_invoke_stub)(this, args, args_size, self, result, result_type);
#endif
    if (UNLIKELY(reinterpret_cast<int32_t>(self->GetException(NULL)) == -1)) {
    // Unusual case where we were running LLVM generated code and an
    // exception was thrown to force the activations to be removed from the
    // stack. Continue execution in the interpreter.  //llvm 生成代码过程中异常会进入解释器
```

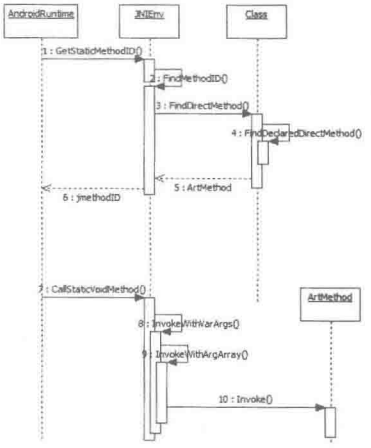
上述两个分支分别代表的函数是 art_portable_invoke_stub()和 art_quick_invoke_stub()，ART_USE_PORTABLE_COMPILER 是一个重要的宏。

在执行托管代码前，要先为其创建栈。这些栈通过 ManagedStack 的成员 link_形成一个先入后出的链表。当执行完托管代码后，只要将最近放入的托管代码栈恢复回来即可。中间是目标函数的执行，但在跳入目标函数体前还需要先执行一些 ABI 层的上下文处理代码，这段代码称为 stub。首先按 ART_USE_PORTABLE_COMPILER 来决定是用 art_quick_invok_stub 还是 art_portable_invok_stub。由于它们是由汇编写成，平台相关，所以每个体系结构（x86, arm, mps）都有其实现。以 x86 体系为例，art_portable_invoke_stub 定义在文件 portable_entrypoints_x86.S 中，具体实现代码如下所示。

```
30DEFINE_FUNCTION art_portable_invoke_stub
31    PUSH ebp                  // save ebp
32    PUSH ebx                  // save ebx
33    mov %esp, %ebp             // copy value of stack pointer into base pointer
34    .cfi_def_cfa_register ebp
35    mov 20(%ebp), %ebx          // get arg array size
36    addl LITERAL(28),%ebx // reserve space for return addr, method*, ebx, and ebp in frame
37    andl LITERAL(0xFFFFFFF0), %ebx    // align frame size to 16 bytes
38    subl LITERAL(12), %ebx       // remove space for return address, ebx, and ebp
39    subl %ebx, %esp             // reserve stack space for argument array
40    lea  4(%esp), %eax          // use stack pointer + method ptr as dest for memcpy
41    pushl 20(%ebp)              // push size of region to memcpy
42    pushl 16(%ebp)              // push arg array as source of memcpy
43    pushl %eax                 // push stack pointer as destination of memcpy
44    call SYMBOL(memcpy)          // (void*, const void*, size_t)
45    addl LITERAL(12), %esp       // pop arguments to memcpy
46    mov 12(%ebp), %eax          // move method pointer into eax
47    mov %eax, (%esp)            // push method pointer onto stack
48    call *METHOD_CODE_OFFSET(%eax) // call the method
49    mov %ebp, %esp             // restore stack pointer
50    POP ebx                   // pop ebx
51    POP ebp                   // pop ebp
52    mov 20(%esp), %ecx          // get result pointer
53    cmpl LITERAL(68), 24(%esp)    // test if result type char == 'D'
54    je return_double_portable
55    cmpl LITERAL(70), 24(%esp)    // test if result type char == 'F'
56    je return_float_portable
57    mov %eax, (%ecx)            // store the result
58    mov %edx, 4(%ecx)           // store the other half of the result
59    ret
60    return_double_portable:
61    fstpl (%ecx)               // store the floating point result as double
62    ret
63    return_float_portable:
64    fstps (%ecx)               // store the floating point result as float
65    ret
66    END_FUNCTION art_portable_invoke_stub
```

由此可见这是 x86 体系中的函数调用过程。首先保存栈帧等信息，然后把参数数组复制到栈中，再执行 call 指令跳转到要执行的目标函数。METHOD_CODE_OFFSET 指向 ArtMethod 中的成员变量 entry_point_from_compiled_code_，也就是编译好的目标函数的地址。接下来，就是等目标函数愉快地执行完，然后恢复上下文，保存返回值，最后执行 ret 指令返回。查找目标函数和执行的过程比较直观，如图 20-9 所示。

到此为止，执行完托管代码后返回到 AndroidRuntimie::start()函数，调用函数 DetachCurrent Thread()和函数 DestroyJavaVM()来做清理工作，并关闭虚拟机，完成整个工作过程。



▲图 20-9 查找目标函数和执行的过程

```
897    ALOGD("Shutting down VM\n");
898    if (mJavaVM->DetachCurrentThread() != JNI_OK)
899        ALOGW("Warning: unable to detach main thread\n");
900    if (mJavaVM->DestroyJavaVM() != 0)
901        ALOGW("Warning: VM did not shut down cleanly\n");
```

函数 DetachCurrentThread()和函数 DestroyJavaVM()在文件 jni_internal.cc 中定义，具体实现代码如下所示。

```
static jint DetachCurrentThread(JavaVM* vm) {
    if (vm == NULL || Thread::Current() == NULL) {
      return JNI_ERR;
    }
    JavaVMExt* raw_vm = reinterpret_cast<JavaVMExt*>(vm);
    Runtime* runtime = raw_vm->runtime;
    runtime->DetachCurrentThread();
    return JNI_OK;
  }
public:
static jint DestroyJavaVM(JavaVM* vm) {
    if (vm == NULL) {
      return JNI_ERR;
    }
    JavaVMExt* raw_vm = reinterpret_cast<JavaVMExt*>(vm);
    delete raw_vm->runtime;
    return JNI_OK;
}
```

在 Android 系统中，当在一个线程里面调用 AttachCurrentThread 后，如果不要用的时候一定要用 DetachCurrentThread 处理，否则线程无法正常退出。

20.7 加载 OAT 文件

ART 的核心是 OAT 文件，这是一种 Android 私有 ELF 文件格式，它不仅包含有从 dex 文件翻译而来的本地机器指令，还包含有原来的 dex 文件内容。这样无需重新编译原有的 APK，就可以让它在 ART 里面正常运行，即不需要改变原来的 APK 编程接口。作为 Android 私有的一种 ELF 文件，OAT 文件包含如下两个特殊的段。

- oatdata：包含用来生成本地机器指令的 dex 文件内容。
- oatexec：包含有生成的本地机器指令。

上述两者之间的关系通过存储在 oatdata 段前面的 OAT 头部描述。此外，在 OAT 文件的 dynamic 段，导出了 3 个符号 oatdata、oatexec 和 oatlastword，它们的值就是用来界定 oatdata 段和 oatexec 段的起止位置的。其中，[oatdata, oatexec - 4]描述的是 oatdata 段的起止位置，而[oatexec, oatlastword]描述的是 oatlastword 的起止位置。

在本节的内容中，将详细讲解 ART 加载 OAT 文件的具体过程。

20.7.1　产生 OAT

在 Android 系统安装 APK 的过程中，会通过 dex2oat 工具生成一个 OAT 文件，此功能通过文件 frameworks/native/cmds/installd/commands.c 中的函数 run_dex2oat 实现，具体实现代码如下所示。

```
static void run_dex2oat(int zip_fd, int oat_fd, const char* input_file_name,
    const char* output_file_name, const char* dexopt_flags)
{
    static const char* DEX2OAT_BIN = "/system/bin/dex2oat";
    static const int MAX_INT_LEN = 12;      // '-'+10dig+'\0' -OR- 0x+8dig
    char zip_fd_arg[strlen("--zip-fd=") + MAX_INT_LEN];
    char zip_location_arg[strlen("--zip-location=") + PKG_PATH_MAX];
    char oat_fd_arg[strlen("--oat-fd=") + MAX_INT_LEN];
    char oat_location_arg[strlen("--oat-name=") + PKG_PATH_MAX];

    sprintf(zip_fd_arg, "--zip-fd=%d", zip_fd);
```

```
    sprintf(zip_location_arg, "--zip-location=%s", input_file_name);
    sprintf(oat_fd_arg, "--oat-fd=%d", oat_fd);
    sprintf(oat_location_arg, "--oat-location=%s", output_file_name);

    ALOGV("Running %s in=%s out=%s\n", DEX2OAT_BIN, input_file_name, output_file_name);
    execl(DEX2OAT_BIN, DEX2OAT_BIN,
          zip_fd_arg, zip_location_arg,
          oat_fd_arg, oat_location_arg,
          (char*) NULL);
    ALOGE("execl(%s) failed: %s\n", DEX2OAT_BIN, strerror(errno));
}
```

在上述代码中，参数 zip_fd 和 oat_fd 是打开文件描述符，分别指向正在安装的 APK 文件和要生成的 OAT 文件。在 OAT 文件的生成过程中，主要涉及了将包含在 APK 里面的 classes.dex 文件的 dex 字节码翻译成本地机器指令。这相当于是编写一个输入文件为 dex、输出文件为 OAT 的编译器，这个编译器是基于 LLVM 开发的。

在安装 APK 过程中，生成的 OAT 文件的输入只有一个 dex 文件，即来自于打包在要安装的 APK 文件里面的 classex.dex 文件。其实一个 OAT 文件是可以由若干个 dex 生成的，这说明在生成的 OAT 文件的 oatdata 段中包含了多个 dex 文件，那么在什么情况下会生成包含多个 dex 文件的 OAT 文件呢？当启动 ART 环境时，在启动 Zygote 进程的过程中会调用 libart.so 里面的函数 JNI_CreateJavaVM 来创建一个 ART 虚拟机。函数 JNI_CreateJavaVM 在文件 art/runtime/jni_internal.cc 中定义，具体实现代码如下所示。

```
extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
  const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
  if (IsBadJniVersion(args->version)) {
    LOG(ERROR) << "Bad JNI version passed to CreateJavaVM: " << args->version;
    return JNI_EVERSION;
  }
  Runtime::Options options;
  for (int i = 0; i < args->nOptions; ++i) {
    JavaVMOption* option = &args->options[i];
    options.push_back(std::make_pair(std::string(option->optionString),
option->extraInfo));
  }
  bool ignore_unrecognized = args->ignoreUnrecognized;
  if (!Runtime::Create(options, ignore_unrecognized)) {
    return JNI_ERR;
  }
  Runtime* runtime = Runtime::Current();
  bool started = runtime->Start();
  if (!started) {
    delete Thread::Current()->GetJniEnv();
    delete runtime->GetJavaVM();
    LOG(WARNING) << "CreateJavaVM failed";
    return JNI_ERR;
  }
  *p_env = Thread::Current()->GetJniEnv();
  *p_vm = runtime->GetJavaVM();
  return JNI_OK;
}
```

在上述代码中，参数 vm_args 作为 ART 虚拟机的启动参数，被转换为一个 JavaVMInitArgs 对象后，会按照 Key-Value 的组织形式保存一个 Options 向量中，并且该向量作为参数传递给 Runtime 类的静态成员函数 Create。

20.7.2 创建 ART 虚拟机

在 Android 5.0 系统中，通过调用类 Runtime 中的静态成员函数 Create，可以在进程中创建一个 ART 虚拟机。在创建成功后，会调用类 Runtime 中的静态成员函数 Start 来启动该 ART 虚拟机，这个创建 ART 虚拟的动作只会在 Zygote 进程中执行。无论是 SystemServer 系统进程，还是 Android 应用程序进程的 ART 虚拟机，都直接从 Zygote 进程 fork 出来进行共享。由此可见，这与 Dalvik 虚拟机的创建方式是完全一样的。类 Runtime 中的静态成员函数 Create 在文件 art/runtime/

runtime.cc 中定义，此函数首先会判断当前进程是否已经创建有一个 ART 虚拟机实例了。如果有则立即返回，否则就创建一个 ART 虚拟机实例，并且保存在 Runtime 类的静态成员变量 instance_ 中，最后调用类 Runtime 的成员函数 init 对该新创建的 ART 虚拟机进行初始化。函数 Create 的具体实现代码如下所示。

```
bool Runtime::Create(const Options& options, bool ignore_unrecognized) {
  // TODO: acquire a static mutex on Runtime to avoid racing.
  if (Runtime::instance_ != NULL) {
    return false;
  }
  InitLogging(NULL);  // Calls Locks::Init() as a side effect.
  instance_ = new Runtime;
  if (!instance_->Init(options, ignore_unrecognized)) {
    delete instance_;
    instance_ = NULL;
    return false;
  }
  return true;
}
```

在上述代码中，instance_是类 Runtime 的静态成员变量，指向了进程中的一个 Runtime 单例。这个 Runtime 单例描述了当前进程的 ART 虚拟机实例。

类Runtime的成员函数init在文件art/runtime/runtime.cc中定义，功能是先调用类ParsedOptions的静态成员函数 Create 对 ART 虚拟机的启动参数 raw_options 进行解析。将解析后得到的参数保存在一个 ParsedOptions 对象中，然后根据这些参数生成一个 ART 虚拟机堆，ART 虚拟机堆使用 Heap 对象来描述。函数 init 的具体实现代码如下所示。

```
bool Runtime::Init(const Options& raw_options, bool ignore_unrecognized) {
  ......

  UniquePtr<ParsedOptions> options(ParsedOptions::Create(raw_options, ignore_
unrecognized));
  ......

  heap_ = new gc::Heap(options->heap_initial_size_,
                       options->heap_growth_limit_,
                       options->heap_min_free_,
                       options->heap_max_free_,
                       options->heap_target_utilization_,
                       options->heap_maximum_size_,
                       options->image_,
                       options->is_concurrent_gc_enabled_,
                       options->parallel_gc_threads_,
                       options->conc_gc_threads_,
                       options->low_memory_mode_,
                       options->long_pause_log_threshold_,
                       options->long_gc_log_threshold_,
                       options->ignore_max_footprint_);
  ......

  java_vm_ = new JavaVMExt(this, options.get());
  ......

  Thread* self = Thread::Attach("main", false, NULL, false);
  ......

  if (GetHeap()->GetContinuousSpaces()[0]->IsImageSpace()) {
    class_linker_ = ClassLinker::CreateFromImage(intern_table_);
  } else {
    ......
    class_linker_ = ClassLinker::CreateFromCompiler(*options->boot_class_path_,
    intern_table_);
  }
  ......

  return true;
}
```

创建好 ART 虚拟机堆后，类 Runtime 的成员函数 init 接着会创建一个 JavaVMExt 实例。这个 JavaVMExt 实例最终需要返回给调用者，使调用者可以通过该 JavaVMExt 实例来和 ART 虚拟机进行交互。然后类 Runtime 中的成员函数 init 通过调用类 Thread 的成员函数 Attach，将当前线程作为 ART 虚拟机的主线程，使得当前线程可以调用 ART 虚拟机提供的 JNI 接口。

类 Runtime 的成员函数 GetHeap 能够返回当前 ART 虚拟机的堆，也就是前面创建的 ART 虚拟机堆。通过调用类 Heap 的成员函数 GetContinuousSpaces 可以获得堆里面的连续空间列表。如果这个列表的第一个连续空间是一个 Image 空间，那么就调用 ClassLinker 类的静态成员函数 CreateFromImage 来创建一个 ClassLinker 对象。否则的话，上述 ClassLinker 对象就要通过 ClassLinker 类的另外一个静态成员函数 CreateFromCompiler 来创建。创建出来的 ClassLinker 对象是后面 ART 虚拟机加载 Java 类时要用到的。

ART 虚拟机的堆包含有 3 个连续空间和一个不连续空间，其中 3 个连续空间分别用来分配不同的对象。当第一个连续空间不是 Image 空间时，就表明当前进程不是 Zygote 进程，而是安装应用程序时启动的一个 dex2oat 进程。安装应用程序时启动的 dex2oat 进程也会在内部创建一个 ART 虚拟机，不过这个 ART 虚拟机是用来将 dex 字节码编译成本地机器指令的，而 Zygote 进程创建的 ART 虚拟机是用来运行应用程序的。

20.7.3 解析启动参数并创建堆

在 Android 系统中，通过使用类 ParsedOptions 的静态成员函数 Create 和 ART 虚拟机堆 Heap 的构造函数，实现 ART 虚拟机的启动参数解析过程和 ART 虚拟机的堆创建过程。其中在文件 art/runtime/runtime.cc 中定义类 ParsedOptions 中的静态成员函数 Create，具体实现代码如下所示。

```
Runtime::ParsedOptions* Runtime::ParsedOptions::Create(const Options& options, bool
ignore_unrecognized) {
  UniquePtr<ParsedOptions> parsed(new ParsedOptions());
  const char* boot_class_path_string = getenv("BOOTCLASSPATH");
  if (boot_class_path_string != NULL) {
    parsed->boot_class_path_string_ = boot_class_path_string;
  }
  ......

  parsed->is_compiler_ = false;
  ......

  for (size_t i = 0; i < options.size(); ++i) {
    const std::string option(options[i].first);
    ......

    if (StartsWith(option, "-Xbootclasspath:")) {
      parsed->boot_class_path_string_ = option.substr(strlen("-Xbootclasspath:")).data();
    } else if (option == "bootclasspath") {
      parsed->boot_class_path_
          = reinterpret_cast<const std::vector<const DexFile*>*>(options[i].second);
    } else if (StartsWith(option, "-Ximage:")) {
      parsed->image_ = option.substr(strlen("-Ximage:")).data();
    } else if (......) {
      ......
    } else if (option == "compiler") {
      parsed->is_compiler_ = true;
    } else {
      ......
    }
  }

  ......

  if (!parsed->is_compiler_ && parsed->image_.empty()) {
    parsed->image_ += GetAndroidRoot();
    parsed->image_ += "/framework/boot.art";
  }

  ......
```

```
  return parsed.release();
}
```

在 ART 虚拟机的启动参数中，最为常用的如下所示。

- -Xbootclasspath：用来指定启动类路径。如果没有指定启动类路径，那么默认的启动类路径就通过环境变量 BOOTCLASSPATH 来获得。
- -Ximage：用来指定 ART 虚拟机所使用的 Image 文件。这个 Image 是用来启动 ART 虚拟机的。
- compiler：用来指定当前要创建的 ART 虚拟机是用来将 dex 字节码编译成本地机器指令的。

如果没有指定 Image 文件，并且当前创建的 ART 虚拟机又不是用来编译 dex 字节码的，那么就将该 Image 文件指定为设备上的/system/framework/boot.art 文件。因为 system 分区的文件都是在制作 ROM 时打包进去的，所以如果没有指定 Image 文件，那么需要将 system 分区预先准备好的 framework/boot.art 文件作为 Image 文件来启动 ART 虚拟机。但是，可能不存在文件/system/framework/boot.art，此时就需要生成一个新的 Image 文件，这个 Image 文件是一个包含了多个 dex 文件的 OAT 文件。

类 Heap 中的构造函数 Heap 在文件 art/runtime/gc/heap.cc 中定义，具体实现代码如下所示。

```
Heap::Heap(size_t initial_size, size_t growth_limit, size_t min_free, size_t max_free,
           double target_utilization, size_t capacity, const std::string& original_image_
           file_name,
           bool concurrent_gc, size_t parallel_gc_threads, size_t conc_gc_threads,
           bool low_memory_mode, size_t long_pause_log_threshold, size_t long_gc_log_
           threshold,
           bool ignore_max_footprint)
    : ..... {
  .....

  std::string image_file_name(original_image_file_name);
  if (!image_file_name.empty()) {
    space::ImageSpace* image_space = space::ImageSpace::Create(image_file_name);
    .....
    AddContinuousSpace(image_space);
    .....
  }

  .....
}
```

在上述代码中，参数 original_image_file_name 表示 Image 文件的路径，如果其值不为空则以它为参数，然后调用 ImageSpace 类的静态成员函数 Create 创建一个 Image 空间，并且调用类 Heap 的成员函数 AddContinuousSpace 将该 Image 空间作为本进程的 ART 虚拟机堆的第一个连续空间。

类 ImageSpace 的静态成员函数 Create 在文件 art/runtime/gc/space/image_space.cc 中定义，具体实现代码如下所示。

```
ImageSpace* ImageSpace::Create(const std::string& original_image_file_name) {
  if (OS::FileExists(original_image_file_name.c_str())) {
    return space::ImageSpace::Init(original_image_file_name, false);
  }
  std::string image_file_name(GetDalvikCacheFilenameOrDie(original_image_file_name));
  if (OS::FileExists(image_file_name.c_str())) {
    space::ImageSpace* image_space = space::ImageSpace::Init(image_file_name, true);
    if (image_space != NULL) {
      return image_space;
    }
  }
  CHECK(GenerateImage(image_file_name)) << "Failed to generate image: " << image_file_name;
  return space::ImageSpace::Init(image_file_name, true);
}
```

在上述代码中，首先检查参数 original_image_file_name 指定的 Image 文件是否存在。如果存在则以它为参数，并调用类 ImageSpace 的另外一个静态成员函数 init 来创建一个 Image 空间。否

则调用函数 GetDalvikCacheFilenameOrDie 根据参数 original_image_file_name 构造另外一个在/data/dalvik-cache 目录下的文件路径，然后再检查这个文件是否存在。 如果存在的话，则以它为参数调用 ImageSpace 类的静态成员函数 init 来创建一个 Image 空间。否则调用类 ImageSpace 的静态成员函数 GenerateImage 生成一个新的 Image 文件，然后再调用类 ImageSpace 的静态成员函数 init 创建一个 Image 空间。

20.7.4 生成指定目录文件

类 ImageSpace 中的静态成员函数 GenerateImage 在文件 art/runtime/gc/space/image_space.cc 中定义，功能是调用 dex2oat 工具在/data/dalvik-cache 目录下生成两个文件：system@framework@boot.art@classes.dex 和 system@framework@boot.art@classes.oat，具体说明如下所示。

- system@framework@boot.art@classes.dex：是一个 Image 文件，通过“—image”选项传递给 dex2oat 工具，在里面包含了一些需要在 Zygote 进程启动时预加载的类。这些需要预加载的类由文件/system/framework/framework.jar 中的 preloaded-classes 文件指定。
- system@framework@boot.art@classes.oat：是一个 OAT 文件，通过--oat-file 选项传递给 dex2oat 工具，它是由系统启动路径中指定的 jar 文件生成的。每一个 jar 文件都通过一个--dex-file 选项传递给 dex2oat 工具。这样 dex2oat 工具就可以将它们所包含的 classes.dex 文件里面的 dex 字节码翻译成本地机器指令。

函数 GenerateImage 的具体实现代码如下所示。

```
static bool GenerateImage(const std::string& image_file_name) {
  const std::string boot_class_path_string(Runtime::Current()->GetBootClassPath
  String());
  std::vector<std::string> boot_class_path;
  Split(boot_class_path_string, ':', boot_class_path);
  ......

  std::vector<std::string> arg_vector;

  std::string dex2oat(GetAndroidRoot());
  dex2oat += (kIsDebugBuild ? "/bin/dex2oatd" : "/bin/dex2oat");
  arg_vector.push_back(dex2oat);

  std::string image_option_string("--image=");
  image_option_string += image_file_name;
  arg_vector.push_back(image_option_string);
  ......

  for (size_t i = 0; i < boot_class_path.size(); i++) {
    arg_vector.push_back(std::string("--dex-file=") + boot_class_path[i]);
  }

  std::string oat_file_option_string("--oat-file=");
  oat_file_option_string += image_file_name;
  oat_file_option_string.erase(oat_file_option_string.size() - 3);
  oat_file_option_string += "oat";
  arg_vector.push_back(oat_file_option_string);
  ......

  if (kIsTargetBuild) {
    arg_vector.push_back("--image-classes-zip=/system/framework/framework.jar");
    arg_vector.push_back("--image-classes=preloaded-classes");
  }
  ......

  // Convert the args to char pointers.
  std::vector<char*> char_args;
  for (std::vector<std::string>::iterator it = arg_vector.begin(); it != arg_vector. end();
      ++it) {
    char_args.push_back(const_cast<char*>(it->c_str()));
  }
  char_args.push_back(NULL);
```

```
    // fork and exec dex2oat
    pid_t pid = fork();
    if (pid == 0) {
      ......

      execv(dex2oat.c_str(), &char_args[0]);

      ......
      return false;
    } else {
      ......

      // wait for dex2oat to finish
      int status;
      pid_t got_pid = TEMP_FAILURE_RETRY(waitpid(pid, &status, 0));
      .......
    }
    return true;
  }
```

这样就得到了一个包含多个 dex 文件的 OAT 文件：system@framework@boot.art@classes.oat。由此可见，通过系统启动类路径指定的 dex 文件生成的 OAT 文件是类型为 BOOT 的 OAT 文件，即 boot.art 文件。

20.7.5　加载 OAT 文件

在运行 ART 时，可以通过类 OatFile 中的静态成员函数 Open 在本进程中加载 OAT 文件。函数 Open 在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
OatFile* OatFile::Open(const std::string& filename,
                       const std::string& location,
                       byte* requested_base,
                       bool executable) {
  CHECK(!filename.empty()) << location;
  CheckLocation(filename);
#ifdef ART_USE_PORTABLE_COMPILER
  if (executable) {
    return OpenDlopen(filename, location, requested_base);
  }
#endif
  UniquePtr<File> file(OS::OpenFileForReading(filename.c_str()));
  if (file.get() == NULL) {
    return NULL;
  }
  return OpenElfFile(file.get(), location, requested_base, false, executable);
}
```

对上述代码的具体说明如下所示。

- 参数 filename 和 location：是一样的，都指向了要加载的 OAT 文件。
- 参数 requested_base：是一个可选参数，用来描述要加载的 OAT 文件里面的 oatdata 段要加载的位置。
- 参数 executable：表示要加载的 OAT 是不是应用程序的主执行文件。一个应用程序通常只有一个 classes.dex 文件，当 classes.dex 文件经过编译后会得到一个 OAT 主执行文件，但是应用程序也可以在运行时动态加载 dex 文件。这些动态加载的 dex 文件在加载时会被翻译成 OAT 再运行，它们相应打包在应用程序的 classes.dex 文件来说，也就不属于主执行文件了。

由此可见，如果在编译时指定了 ART_USE_PORTABLE_COMPILER 宏，并且参数 executable 为 true，那么就通过类 OatFile 的静态成员函数 OpenDlopen 来加载指定的 OAT 文件。类 OatFile 的静态成员函数 OpenDlopen 直接通过动态链接器提供的 dlopen 函数来加载 OAT 文件。在其余情况下，通过类 OatFile 的静态成员函数 OpenElfFile 来手动加载指定的 OAT 文件。这种方式是按照 ELF 文件格式来解析要加载的 OAT 文件的，并且根据解析获得的信息将 OAT 里面相应的段加载到内存中来。

接下来看类 OatFile 中的静态成员函数 OpenDlopen 和 OpenElfFile，其中函数 OpenDlopen 在文件 art/runtime/oat_file.cc 中定义，功能是先创建一个 OatFile 对象，然后调用 OatFile 对象的成员函数 Dlopen 加载参数 elf_filename 指定的 OAT 文件。函数 OpenDlopen 的具体实现代码如下所示。

```
OatFile* OatFile::OpenDlopen(const std::string& elf_filename,
                             const std::string& location,
                             byte* requested_base) {
  UniquePtr<OatFile> oat_file(new OatFile(location));
  bool success = oat_file->Dlopen(elf_filename, requested_base);
  if (!success) {
    return NULL;
  }
  return oat_file.release();
}
```

类 OatFile 中的成员函数 Dlopen 在文件 art/runtime/oat_file.cc 中定义，功能是首先是通过动态链接器提供的函数 dlopen 将参数 elf_filename 指定的 OAT 文件加载到内存中，然后通过动态链接器提供的函数 dlsym 从加载进来的 OAT 文件获得两个导出符号 oatdata 和 oatlastword 的地址，分别保存在当前正在处理的 OatFile 对象的成员变量 begin_和 end_中。最后通过调用另外一个成员函数 Setup 来解析已经加载内存中的 oatdata 段，以获得 ART 运行所需要的更多信息。分析完成 OatFile 类的静态成员函数 OpenElfFile 之后，再来看 OatFile 类的成员函数 Setup 的实现。函数 Dlopen 的具体实现代码如下所示。

```
bool OatFile::Dlopen(const std::string& elf_filename, byte* requested_base) {
  char* absolute_path = realpath(elf_filename.c_str(), NULL);
  ......

  dlopen_handle_ = dlopen(absolute_path, RTLD_NOW);
  ......

  begin_ = reinterpret_cast<byte*>(dlsym(dlopen_handle_, "oatdata"));
  ......

  if (requested_base != NULL && begin_ != requested_base) {
    ......
    return false;
  }

  end_ = reinterpret_cast<byte*>(dlsym(dlopen_handle_, "oatlastword"));
  ......

  // Readjust to be non-inclusive upper bound.
  end_ += sizeof(uint32_t);
  return Setup();
}
```

类 OatFile 中的静态成员函数 OpenElfFile 在文件 art/runtime/oat_file.cc 中定义，此函数通过 ElfFile 类来手动加载参数 file 指定的 OAT 文件，也就是按照 ELF 文件格式来解析参数 file 指定的 OAT 文件，并且将文件里面的 oatdata 段和 oatexec 段加载到内存中来。为了便于理解，将类 ElfFile 看作是 ART 运行时自己实现的 OAT 文件动态链接器。当加载完成参数 file 指定的 OAT 文件后，通过两个导出符号 oatdata 和 oatlastword 来获得 oatdata 段和 oatexec 段的起止位置。如果参数 requested_base 的值不等于 0，那么就要求 oatdata 段必须要加载到 requested_base 指定的位置去。函数 OpenElfFile 的具体实现代码如下所示。

```
bool OatFile::ElfFileOpen(File* file, byte* requested_base, bool writable, bool
executable) {
  elf_file_.reset(ElfFile::Open(file, writable, true));
  ......
  bool loaded = elf_file_->Load(executable);
  ......
  begin_ = elf_file_->FindDynamicSymbolAddress("oatdata");
  ......
  if (requested_base != NULL && begin_ != requested_base) {
    ......
    return false;
```

```
    }
    end_ = elf_file_->FindDynamicSymbolAddress("oatlastword");
    ......
    // Readjust to be non-inclusive upper bound.
    end_ += sizeof(uint32_t);
    return Setup();
}
```

20.7.6　解析字段

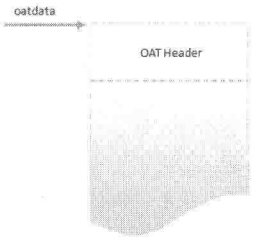
将参数 file 指定的 OAT 文件加载到内存之后，类 OatFile 中的静态成员函数 OpenElfFile 最后会调用类 OatFile 的成员函数 Setup 来解析其中的 oatdata 段。类 OatFile 中的成员函数 Setup 在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
bool OatFile::Setup() {
  if (!GetOatHeader().IsValid()) {
    LOG(WARNING) << "Invalid oat magic for " << GetLocation();
    return false;
  }
  const byte* oat = Begin();
  oat += sizeof(OatHeader);
  if (oat > End()) {
    LOG(ERROR) << "In oat file " << GetLocation() << " found truncated OatHeader";
    return false;
  }
  oat += GetOatHeader().GetImageFileLocationSize();
  if (oat > End()) {
    LOG(ERROR)<<"In oat file"<<GetLocation()<<"found truncated image file location:"
              << reinterpret_cast<const void*>(Begin())
              << "+" << sizeof(OatHeader)
              << "+" << GetOatHeader().GetImageFileLocationSize()
              << "<=" << reinterpret_cast<const void*>(End());
    return false;
  }
  for (size_t i = 0; i < GetOatHeader().GetDexFileCount(); i++) {
    size_t dex_file_location_size = *reinterpret_cast<const uint32_t*>(oat);
    ......

    oat += sizeof(dex_file_location_size);
    ......

    const char* dex_file_location_data = reinterpret_cast<const char*>(oat);
    oat += dex_file_location_size;
    ......

    std::string dex_file_location(dex_file_location_data, dex_file_location_size);

    uint32_t dex_file_checksum = *reinterpret_cast<const uint32_t*>(oat);
    oat += sizeof(dex_file_checksum);
    ......

    uint32_t dex_file_offset = *reinterpret_cast<const uint32_t*>(oat);
    ......

    oat += sizeof(dex_file_offset);
    ......

    const uint8_t* dex_file_pointer = Begin() + dex_file_offset;
    if (!DexFile::IsMagicValid(dex_file_pointer)) {
      ......
      return false;
    }
    if (!DexFile::IsVersionValid(dex_file_pointer)) {
      ......
      return false;
    }

    const DexFile::Header* header = reinterpret_cast<const DexFile::Header*>(dex_file_
    pointer);
    const uint32_t* methods_offsets_pointer = reinterpret_cast<const uint32_t*>(oat);
```

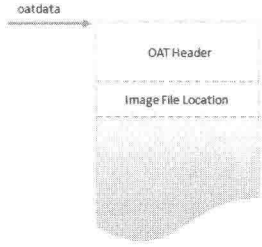
```
    oat += (sizeof(*methods_offsets_pointer) * header->class_defs_size_);
    ......

    oat_dex_files_.Put(dex_file_location, new OatDexFile(this,
                                                         dex_file_location,
                                                         dex_file_checksum,
                                                         dex_file_pointer,
                                                         methods_offsets_pointer));
  }
  return true;
}
```

通过对函数代码的分析可知，OAT 文件的 oatdata 段在最开始保存着一个 OAT 头，如图 20-10 所示。

紧接着在 OAT 头后面的是 Image 空间文件路径，如图 20-11 所示。



▲图 20-10 OAT 头部



▲图 20-11 OAT 头和 Image 空间文件路径

另外，通过上述粗斜体代码能够获得包含在 oatdata 段的 dex 文件描述信息，每一个 dex 文件记录在 oatdata 段中包括了如下所示的描述信息。

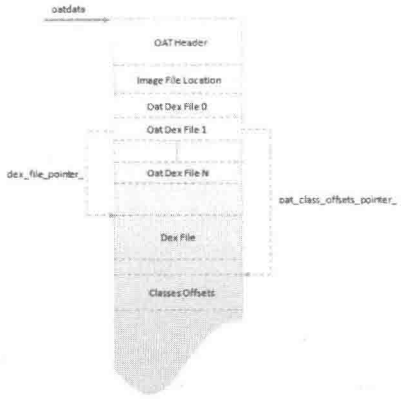
- dex 文件路径大小：保存在变量 dex_file_location_size 中。
- dex 文件路径：保存在变量 dex_file_location_data 中。
- dex 文件检验和：保存在变量 dex_file_checksum 中。
- dex 文件内容在 oatdata 段的偏移：保存在变量 dex_file_offset 中。通过这个信息可以在 oatdata 段中找到对应的 dex 文件的内容。dex 文件最开始部分是一个 dex 文件头，上述代码通过检查 dex 文件头的模数和版本号来确保变量 dex_file_offset 指向的位置确实是一个 dex 文件。
- dex 文件包含的类的本地机器指令信息偏移数组：保存在变量 methods_offsets_pointer 中。通过这个信息可以找到 dex 文件里面的每一个类方法对应的本地机器指令。这个数组的大小等于 header->class_defs_size_，即 dex 文件里面的每一个类在数组中都对应有一个偏移值。这里的 header 指向的是 dex 文件头，它的 class_defs_size_描述了 dex 文件包含的类的个数。在 DEX 文件中，每一个类都是有一个从 0 开始的编号，该编号就是用来索引到上述数组的，从而获得对应的类所有方法的本地机器指令信息。

上述得到的每一个 dex 文件的信息都被封装在一个 OatDexFile 对象中，以便以后可以直接访问。如果使用 OatDexFile 来描述每一个 dex 文件的描述信息，那么就可以通过图 20-12 看到这些描述信息在 oatdata 段的位置。

类 OatFile 中的成员函数 GetOatHeader、Begin 和 End 在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
const OatHeader& OatFile::GetOatHeader() const {
  return *reinterpret_cast<const OatHeader*>(Begin());
}

const byte* OatFile::Begin() const {
  CHECK(begin_ != NULL);
  return begin_;
}
```

```
const byte* OatFile::End() const {
  CHECK(end_ != NULL);
  return end_;
}
```



▲图 20-12　4 OAT 头、Image 空间文件路径、dex 文件描述信息

上述函数主要涉及了类 OatFile 的两个成员变量 begin_和 end_，它们分别是 OAT 文件里面的 oatdata 段开始地址和 oatexec 段的结束地址。通过上述类 OatFile 的成员函数 GetOatHeader 代码，可以看到 OAT 文件中的 oatdata 段的开始储存着一个 OAT 头，这个 OAT 头通过类 OatHeader 描述。在文件 art/runtime/oat.h 中定义，具体实现代码如下所示。

```
class PACKED(4) OatHeader {
 public:
   ......
 private:
  uint8_t magic_[4];
  uint8_t version_[4];
  uint32_t adler32_checksum_;

  InstructionSet instruction_set_;
  uint32_t dex_file_count_;
  uint32_t executable_offset_;
  uint32_t interpreter_to_interpreter_bridge_offset_;
  uint32_t interpreter_to_compiled_code_bridge_offset_;
  uint32_t jni_dlsym_lookup_offset_;
  uint32_t portable_resolution_trampoline_offset_;
  uint32_t portable_to_interpreter_bridge_offset_;
  uint32_t quick_resolution_trampoline_offset_;
  uint32_t quick_to_interpreter_bridge_offset_;

  uint32_t image_file_location_oat_checksum_;
  uint32_t image_file_location_oat_data_begin_;
  uint32_t image_file_location_size_;
  uint8_t image_file_location_data_[0];  // note variable width data at end

  ......
};
```

类 OatHeader 中各个成员变量的具体说明如下所示。

- magic：标志 OAT 文件的一个魔数，等于‘oat\n’。

注意

什么是魔数

很多类型的文件，其起始的几个字节的内容是固定的（或是有意填充，或是本就如此）。根据这几个字节的内容就可以确定文件类型，因此这几个字节的内容被称为魔数（magic number）。此外在一些程序代码中，程序员常常将在代码中出现但没有解释的数字常量或字符串称为魔数（magic number）或魔字符串。

- version：OAT 文件版本号，目前的值等于‘007、0’。
- adler32_checksum_：OAT 头部检验和。
- instruction_set_：本地机指令集，有 4 种取值，分别为 kArm(1)、kThumb2(2)、kX86(3)和 kMips(4)。
- dex_file_count_：OAT 文件包含的 dex 文件个数。
- executable_offset_：oatexec 段开始位置与 oatdata 段开始位置的偏移值。
- interpreter_to_interpreter_bridge_offset_和 interpreter_to_compiled_code_bridge_offset_：ART 运行时在启动的时候，可以通过-Xint 选项指定所有类的方法都是解释执行的，这与传统的虚拟机使用解释器来执行类方法差不多。同时，有些类方法可能没有被翻译成本地机器指令，这时候也要求对它们进行解释执行。这意味着解释执行的类方法在执行的过程中，可能会调用到另外一个也是解释执行的类方法，也可能调用到另外一个按本地机器指令执行的类方法中。OAT 文件在内部提供有两段 trampoline 代码，分别用来从解释器调用另外一个也是通过解释器来执行的类方法和从解释器调用另外一个按照本地机器执行的类方法。这两段 trampoline 代码的偏移位置就保存在成员变量 interpreter_to_interpreter_bridge_offset_和 interpreter_to_compiled_code_bridge_offset_中。
- jni_dlsym_lookup_offset_：类方法在执行的过程中，如果要调用的另外一个方法是一个 JNI 函数，那么就要通过放置在 jni_dlsym_lookup_offset_中的一段 trampoline 代码来调用。
- portable_resolution_trampoline_offset_和 quick_resolution_trampoline_offset_：用来在运行时解析还未链接的类方法的两段 trampoline 代码。其中，portable_resolution_trampoline_offset_指向的 trampoline 代码用于 Portable 类型的 Backend 生成的本地机器指令，而 quick_resolution_trampoline_offset_用于 Quick 类型的 Backend 生成的本地机器指令。
- portable_to_interpreter_bridge_offset_和 quick_to_interpreter_bridge_offset_：与 interpreter_to_interpreter_bridge_offset_和 interpreter_to_compiled_code_bridge_offset_的作用刚好相反，用来在按照本地机器指令执行的类方法中调用解释执行的类方法的两段 trampoline 代码。其中，portable_to_interpreter_bridge_offset_用于 Portable 类型的 Backend 生成的本地机器指令，而 quick_to_interpreter_bridge_offset_用于 Quick 类型的 Backend 生成的本地机器指令。

由于每一个应用程序都会依赖于 boot.art 文件，因此，为了节省由打包在应用程序里面的 classes.dex 生成的 OAT 文件的体积，其中变量 interpreter_to_interpreter_bridge_offset_、interpreter_to_compiled_code_bridge_offset_、jni_dlsym_lookup_offset_、portable_resolution_trampoline_offset_、portable_to_interpreter_bridge_offset_、quick_resolution_trampoline_offset_和 quick_to_interpreter_bridge_offset_指向的 trampoline 代码段只存在于 boot.art 文件中。换句话说，在由打包在应用程序里面的 classes.dex 生成的 OAT 文件的 oatdata 段头部中，上述 7 个成员变量的值均等于 0。

而在如下所示的 4 个成员变量中，记录了一个 OAT 文件所依赖的用来创建 Image 空间文件及创建这个 Image 空间文件所使用的 OAT 文件的相关信息。

- image_file_location_data_：用来创建 Image 空间的文件路径在内存中的地址。
- image_file_location_size_：用来创建 Image 空间的文件路径的大小。
- image_file_location_oat_data_begin_：用来创建 Image 空间的 OAT 文件的 oatdata 段在内存中的位置。
- image_file_location_oat_checksum_：用来创建 Image 空间的 OAT 文件的检验和。

再看 OatDexFile 类的构造函数的实现，此函数在文件 art/runtime/oat_file.cc 中定义，功能是将在上面得到的 dex 文件描述信息保存在相应的成员变量中。通过这些信息，可以获得包含在该 dex 文件里面的类的所有方法的本地机器指令信息。具体实现代码如下所示。

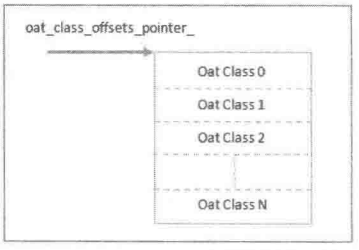
```
OatFile::OatDexFile::OatDexFile(const OatFile* oat_file,
                                const std::string& dex_file_location,
                                uint32_t dex_file_location_checksum,
                                const byte* dex_file_pointer,
                                const uint32_t* oat_class_offsets_pointer)
    : oat_file_(oat_file),
      dex_file_location_(dex_file_location),
      dex_file_location_checksum_(dex_file_location_checksum),
```

```
    dex_file_pointer_(dex_file_pointer),
    oat_class_offsets_pointer_(oat_class_offsets_pointer) {}
```

例如，通过调用类 OatDexFile 中的成员函数 GetOatClass，可以获得指定类的所有方法的本地机器指令信息。此函数在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
const OatFile::OatClass* OatFile::OatDexFile::GetOatClass(uint16_t class_def_index)
const {
  uint32_t oat_class_offset = oat_class_offsets_pointer_[class_def_index];

  const byte* oat_class_pointer = oat_file_->Begin() + oat_class_offset;
  CHECK_LT(oat_class_pointer, oat_file_->End()) << oat_file_->GetLocation();
  mirror::Class::Status status = *reinterpret_cast<const mirror::Class::Status*>
  (oat_class_pointer);

  const byte* methods_pointer = oat_class_pointer + sizeof(status);
  CHECK_LT(methods_pointer, oat_file_->End()) << oat_file_->GetLocation();

  return new OatClass(oat_file_,
                      status,
                      reinterpret_cast<const OatMethodOffsets*>(methods_pointer));
}
```

在上述代码中，参数 class_def_index 表示要查找的目标类的编号。这个编号用作数组 oat_class_offsets_pointer_（即前面描述的 methods_offsets_pointer 数组）的索引，就可以得到一个偏移位置 oat_class_offset。这个偏移位置是相对于 OAT 文 件的 oatdata 段的，因此，将该偏移值加上 OAT 文件的 oatdata 段的开始位置后，就可以得到目标类的所有方法的本地机器指令信息。这些信息的布局如图 20-13 所示。



▲图 20-13　dex 文件里面的类描述信息

在 OAT 文件中，每一个 dex 文件包含的每一个类的描述信息都通过一个 OatClass 对象来描述，这通常被称之为 OAT 类。通过类 OatClass 中的构造函数可以理解 OAT 类的作用，具体实现代码如下所示。

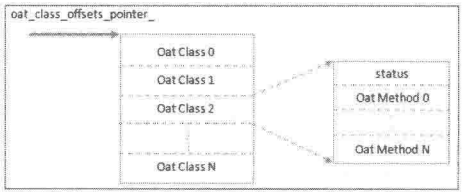
```
OatFile::OatClass::OatClass(const OatFile* oat_file,
                            mirror::Class::Status status,
                            const OatMethodOffsets* methods_pointer)
    : oat_file_(oat_file), status_(status), methods_pointer_(methods_pointer) {}
```

上述函数在文件 art/runtime/oat_file.cc 中定义，其中参数 oat_file 表示宿主 OAT 文件，参数 status 表示 OAT 类状态，数组参数 methods_pointer 表示 OAT 类的各个方法的信息，它们被分别保存在 OatClass 类的相应成员变量中。通过这些信息可以获得包含在该 dex 文件中类的所有方法的本地机器指令信息，例如，通过调用类 OatClass 的成员函数 GetOatMethod 可以获得指定类方法的本地机器指令信息，函数 GetOatMethod 在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
const OatFile::OatMethod OatFile::OatClass::GetOatMethod(uint32_t method_index) const
{
  const OatMethodOffsets& oat_method_offsets = methods_pointer_[method_index];
  return OatMethod(
      oat_file_->Begin(),
      oat_method_offsets.code_offset_,
      oat_method_offsets.frame_size_in_bytes_,
```

```
        oat_method_offsets.core_spill_mask_,
        oat_method_offsets.fp_spill_mask_,
        oat_method_offsets.mapping_table_offset_,
        oat_method_offsets.vmap_table_offset_,
        oat_method_offsets.gc_map_offset_);
}
```

在上述代码中，参数 method_index 表示目标方法在类中的编号。通过这个编号作为索引可以在 OatClass 类的成员变量 methods_pointer_指向的一个数组中，找到目标方法的本地机器指令信息。这些本地机器指令信息封装在一个 OatMethod 对象中，它们在 OAT 文件的具体布局信息如图 20-14 所示。



▲图 20-14 dex 文件里面的类（OatClass）描述信息

OatMethod 的构造函数在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

```
OatFile::OatMethod::OatMethod(const byte* base,
                              const uint32_t code_offset,
                              const size_t frame_size_in_bytes,
                              const uint32_t core_spill_mask,
                              const uint32_t fp_spill_mask,
                              const uint32_t mapping_table_offset,
                              const uint32_t vmap_table_offset,
                              const uint32_t gc_map_offset)
  : begin_(base),
    code_offset_(code_offset),
    frame_size_in_bytes_(frame_size_in_bytes),
    core_spill_mask_(core_spill_mask),
    fp_spill_mask_(fp_spill_mask),
    mapping_table_offset_(mapping_table_offset),
    vmap_table_offset_(vmap_table_offset),
    native_gc_map_offset_(gc_map_offset) {
    ......
}
```

类 OatMethod 中包含了很多对应类方法的本地机器指令执行时要用到的信息，例如，包括参数 base 描述的 OAT 文件的 OAT 头在内存的位置信息，也包括参数 code_offset 描述的类方法的本地机器指令相对 OAT 头的偏移位置信息。将这两者相加，就可以得到一个类方法的本地机器指令在内存的位置。可以通过调用类 OatMethod 的成员函数 GetCode 来获得这个结果，函数 GetCode 在文件 art/runtime/oat_file.cc 中定义，具体实现代码如下所示。

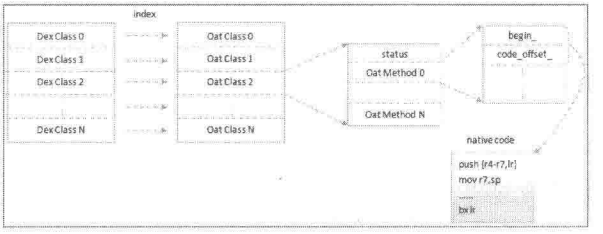
```
const void* OatFile::OatMethod::GetCode() const {
  return GetOatPointer<const void*>(code_offset_);
}
```

类 OatMethod 中的成员函数需要调用另一个成员函数 GetOatPointer，目的是获得一个类方法的本地机器指令在内存中的位置。函数 GetOatPointer 在文件 art/runtime/oat_file.h 中定义，具体实现代码如下所示。

```
class OatFile {
  ......

  class OatMethod {
   ......

   private:
    template<class T>
    T GetOatPointer(uint32_t offset) const {
```

```
      if (offset == 0) {
        return NULL;
      }
      return reinterpret_cast<T>(begin_ + offset);
    }

    ......
  };

  ......
};
```

到此为止，通过分析 OAT 文件加载过程就可以看到 OAT 文件的格式，并且了解在 OAT 文件中找到一个类方法的本地机器指令的过程。在 OAT 文件中查找类方法的本地机器指令的过程如图 20-15 所示。



▲图 20-15　查找类方法的过程

对图 20-15 中所示过程的具体说明如下所示。

（1）首先根据类签名信息从包含在 OAT 文件里面的 dex 文件中查找目标 Class 的编号，然后再根据这个编号找到在 OAT 文件中对应的 OatClass。

（2）然后根据方法签名从包含在 OAT 文件里面的 dex 文件中查找目标方法的编号。

（3）再根据这个编号在前面找到的 OatClass 中对应的 OatMethod。

有了这个 OatMethod 之后，就可以根据它的成员变量 begin_和 code_offset_找到目标类方法的本地机器指令。其中在从 dex 文件中根据签名找到类和方法的编号时，要求对 dex 文件进行解析操作。