



经典畅销书《深入理解Android：卷I》的姊妹卷，51CTO移动开发频道
和开源中国社区一致鼎力推荐！

从系统设计者的角度对Java Framework包含的重要模块和服务的源代码
进行细致剖析，深刻揭示其实现原理和工作机制



邓凡平◎著

Understanding Android Internals: Volume II

深入理解Android

卷 II



机械工业出版社
China Machine Press

深入理解Android

——卷II

邓凡平 著

ISBN : 978-7-111-38918-7

本书纸版由机械工业出版社于2012年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目 录

前言

第1章 搭建Android源码工作环境

1.1 Android系统架构

1.2 搭建开发环境

1.2.1 下载源码

1.2.2 编译源码

1.2.3 利用Eclipse调试system_process

1.3 本章小结

第2章 深入理解Java Binder和MessageQueue

2.1 概述

2.2 Java层中的Binder架构分析

2.2.1 Binder架构总览

2.2.2 初始化Java层Binder框架

2.2.3 addService实例分析

2.2.4 Java层Binder架构总结

2.3 心系两界的MessageQueue

2.3.1 MessageQueue的创建

2.3.2 提取消息

2.3.3 nativePollOnce函数分析

2.3.4 MessageQueue总结

2.4 本章小结

第3章 深入理解SystemServer

3.1 概述

3.2 SystemServer分析

3.2.1 main函数分析

3.2.2 Service群英会

3.3 EntropyService分析

3.4 DropBoxManagerService分析

3.4.1 DBMS构造函数分析

3.4.2 dropbox日志文件的添加

3.4.3 DBMS和settings数据库

3.5 DiskStatsService

和

DeviceStorageMonitorService分析

3.5.1 DiskStatsService分析

3.5.2 DeviceStorageManagerService分析

3.6 SamplingProfilerService分析

3.6.1 SamplingProfilerService 构造函数分析

3.6.2 SamplingProfilerIntegration分析

3.7 ClipboardService分析

3.7.1 复制数据到剪贴板

3.7.2 从剪切板粘贴数据

3.7.3 CBS中的权限管理

3.8 本章小结

第4章 深入理解PackageManagerService

4.1 概述

4.2 初识PackageManagerService

4.3 PKMS的main函数分析

4.3.1 构造函数分析之前期准备工作

4.3.2 构造函数分析之扫描Package

4.3.3 构造函数分析之扫尾工作

4.3.4 PKMS构造函数总结

4.4 APK Installation分析

4.4.1 adb install分析

4.4.2 pm分析

4.4.3 installPackageWithVerification函数分析

4.4.4 APK安装流程总结

4.4.5 Verification介绍

4.5 queryIntentActivities分析

4.5.1 Intent及IntentFilter介绍

4.5.2 Activity信息的管理

4.5.3 Intent匹配查询分析

4.5.4 queryIntentActivities总结

4.6 installd及UserManager介绍

4.6.1 installd介绍

4.6.2 UserManager介绍

4.7 本章学习指导

4.8 本章小结

第5章 深入理解PowerManagerService

5.1 概述

5.2 初识PowerManagerService

5.2.1 PMS构造函数分析

5.2.2 init分析

- 5.2.3 systemReady分析
- 5.2.4 BootComplete处理
- 5.2.5 初识PowerManagerService总结

5.3 PMS WakeLock分析

- 5.3.1 WakeLock客户端分析
- 5.3.2 PMS acquireWakeLock分析
- 5.3.3 Power类及LightService类介绍
- 5.3.4 WakeLock总结

5.4 userActivity及Power按键处理分析

- 5.4.1 userActivity分析
- 5.4.2 Power按键处理分析

5.5 BatteryService及BatteryStatsService分析

- 5.5.1 BatteryService分析
- 5.5.2 BatteryStatsService分析
- 5.5.3 BatteryService 及 BatteryStatsService总结

5.6 本章学习指导

5.7 本章小结

第6章 深入理解ActivityManagerService

- 6.1 概述
- 6.2 初识ActivityManagerService
 - 6.2.1 ActivityManagerService 的 main 函数分析
 - 6.2.2 AMS的setSystemProcess分析

- 6.2.3 AMS的installSystemProviders函数分析
- 6.2.4 AMS的systemReady分析
- 6.2.5 初识ActivityManagerService总结
- 6.3 startActivity分析
 - 6.3.1 从am说起
 - 6.3.2 AMS 的 startActivityAndWait 函数分析
 - 6.3.3 startActivityLocked分析
- 6.4 Broadcast和BroadcastReceiver分析
 - 6.4.1 registerReceiver流程分析
 - 6.4.2 sendBroadcast流程分析
 - 6.4.3 BROADCAST_INTENT_MSG 消息处理函数
 - 6.4.4 应用进程处理广播分析
 - 6.4.5 广播处理总结
- 6.5 startService之按图索骥
 - 6.5.1 Service知识介绍
 - 6.5.2 startService流程图
- 6.6 AMS中的进程管理
 - 6.6.1 Linux进程管理介绍
 - 6.6.2 关于Android中的进程管理的介绍
 - 6.6.3 AMS进程管理函数分析
 - 6.6.4 AMS进程管理总结
- 6.7 App的Crash处理

- 6.7.1 应用进程的Crash处理
- 6.7.2 AMS 的 handleApplicationCrash 分析
- 6.7.3 AppDeathRecipient binderDied 分析
- 6.7.4 App的Crash处理总结

6.8 本章学习指导

6.9 本章小结

第7章 深入理解ContentProvider

7.1 概述

7.2 MediaProvider的启动及创建

- 7.2.1 Context的getContentResolver函数分析
- 7.2.2 MediaStore.Image.Media的query函数分析
- 7.2.3 MediaProvider的启动及创建总结

7.3 SQLite创建数据库分析

- 7.3.1 SQLite及SQLiteDatabase家族
- 7.3.2 MediaProvider创建数据库分析
- 7.3.3 SQLiteDatabase创建数据库的分析总结

7.4 Cursor的query函数的实现分析

- 7.4.1 提取query关键点
- 7.4.2 MediaProvider的query分析
- 7.4.3 query关键点分析

7.4.4 Cursor query实现分析总结

7.5 Cursor close函数实现分析

7.5.1 客户端close的分析

7.5.2 服务端close的分析

7.5.3 finalize函数分析

7.5.4 Cursor close函数总结

7.6 ContentResolver openAssetFileDescriptor函数分析

7.6.1 openAssetFileDescriptor之客户端调用分析

7.6.2 ContentProvider 的 openTypedAssetFile函数分析

7.6.3 跨进程传递文件描述符的探讨

7.6.4 openAssetFileDescriptor 函数分析总结

7.7 本章学习指导

7.8 本章小结

第 8 章 深 入 理 解 ContentService 和 AccountManagerService

8.1 概述

8.2 数据更新通知机制分析

8.2.1 初识ContentService

8.2.2 ContentResovler 的 registerContentObserver分析

8.2.3 ContentResolver的notifyChange分析

8.2.4 数据更新通知机制总结和深入探讨

8.3 AccountManagerService分析

8.3.1 初识AccountManagerService

8.3.2 AccountManager addAccount分析

8.3.3 AccountManagerService 的分析总结

8.4 数据同步管理SyncManager分析

8.4.1 初识SyncManager

8.4.2 ContentResolver 的 requestSync 分析

8.4.3 数据同步管理SyncManager分析总结

8.5 本章学习指导

8.6 本章小结

“深入理解Android”系列书籍的规划路线图

前言

本书主要内容及特色

本书是笔者“深入理解Android”系列的第二本，这一本将关注重点放在了Android Framework的Java层。在众多可供分析的知识点中，笔者另辟蹊径，选择了SystemServer中的服务（Service）作为主人公。这些Service大体可由图1来表示。

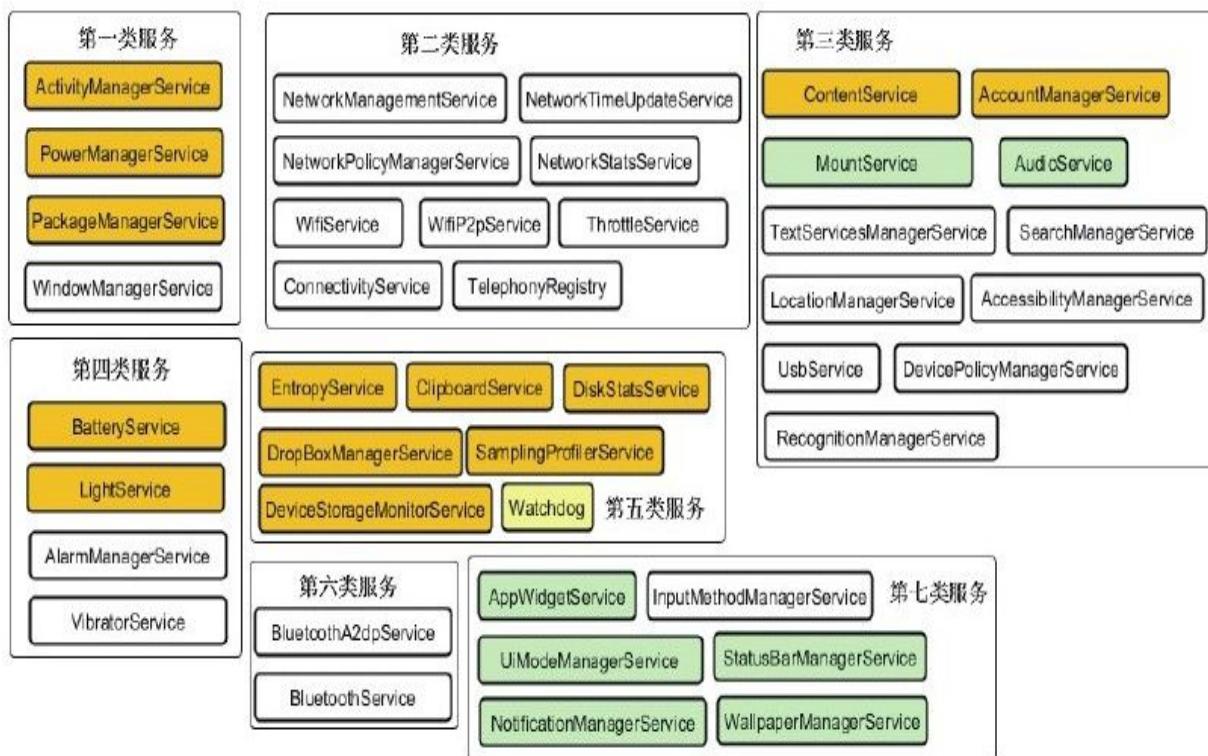


图 1 SystemServer中的服务

由图1可知，SystemServer中的服务可划分为7大类共43项（包括Watchdog在内）：

位于第一大类的是Android的核心服务，如ActivityManagerService、WindowManager-Service等。

位于第二大类的是和通信相关的服务，如Wifi的相关服务、Telephone的相关服务。

位于第三大类的是和系统功能相关的服务，如AudioService、MountService、UsbService等。

位于第四大类的是BatteryService、VibratorService等服务。

位于第五大类的是EntropyService、DiskStatsService、Watchdog等相对独立的服务。

位于第六大类的是蓝牙服务。

位于第七大类的是UI方面的服务，如状态栏服务、通知管理服务等。

以上这些服务就是Android Framework Java层的核心。毫不夸张地说，它们也是Android系统的基石。另外，这些服务的内容远非一本书所能囊括。作为Android Java层Framework分析的先头部队，本书涵盖了以下内容：

第1章，介绍了阅读本书需要做的一些准备工作，包括Android 4.0源码的下载和编译、Eclipse开发环境的搭建，以及Android系统进程(system_process)的调试等。

第2章，介绍了Java Binder和MessageQueue的实现。

第3章，介绍了SystemServer，并分析了图1中第五类包含的服务的工作原理。这些服务包括EntropyService、DropBoxManagerService、DiskStatsService、DeviceStorageMonitorService、SamplingProfilerService和ClipboardService。

第4章，分析了PackageManagerService，该服务负责Android系统中的Package信息查询和APK安装、卸载、更新等方面的工作。

第5章，讲解了PowerManagerService，它是Android中电源管理的核心服务。本章对其中的WakeLock、Power按键处理、BatteryStatsService和BatteryService都做了一番较为深入的分析。

第6章，以ActivityManagerService为分析重点，该服务是Android的核心服务。本章对ActivityManagerService的启动、Activity的创建和

启动、BroadcastReceiver的工作原理、Android中的进程管理等内容进行了较为深入的研究。

第7章，对ContentProvider的创建和启动、SQLite相关知识、Cursor query和close的实现等进行了较为深入的分析。

第8章，以ContentService和AccountManagerService为分析对象，介绍了数据更新通知机制的实现、账户管理和数据同步等方面的知识。

图1中的其他服务将会在“深入理解Android”系列的其他书中详细分析。该系列书的规划请见本书最后面的“深入理解Android系列图书路线图”。

本书以直接剖析源码的方式进行讲解，旨在引领读者一步步深入于Android系统中相关模块的内部原理，去理解它们是如何实现、如何工作的。在分析过程中，笔者根据个人研究Android代码的心得，采用了精简流程和逐个击破的方法。同时，笔者还提出了一些难度不大的知识点、相关的补充阅读资料，甚至笔者在实际项目中遇到的开放式问题，留给读者自行研究和探讨。总之，笔者希望读者在阅读完本书后，至少能有以下两个收获：

能从“基于Android并高于Android”的角度来看待和分析Android。

能初步具有大型复杂代码的分析能力。

读者对象

适合阅读本书的读者包括：

(1) Android应用开发工程师

虽然应用开发工程师平常接触的多是Android SDK，但是只有更深入地理解了Android系统运行原理，才能写出更健壮、更高效的模块。

(2) Android系统开发工程师

系统开发工程师常常需要深入理解系统的运转过程，而本书所涉及的内容正是他们在工作和学习中最想了解的。那些对具体服务（如ActivityManagerService、PackageManagerService）感兴趣的读者，也可以单刀直入，阅读本书相关章节。

(3) 对Android系统运行原理感兴趣的读者

这部分读者需要具有基本的Android开发知识基础。

如何阅读本书

本书是针对Android源码进行分析的，而源码文件所在的路径一般都很长，例如，文件AndroidRuntime.cpp 的真实路径是frameworks/base/core/jni/AndroidRuntime.cpp。为了行文方便，在各章节开头，均把本章涉及的源码路径全部列出，而在具体分析源码时，则只列出该源码的文件名。例如：

[-->AndroidRuntime.cpp]

//这里是源码和一些注释

另外，本书在描述类之间的关系及函数调用流程上，使用了UML的静态类图及序列图。UML是一个强大的工具，但它的建模规范过于烦琐，为更简单清晰地描述事情的本质，本书并未完全遵循UML的建模规范。这里仅举一例，如图2所示。



图 2 UML示例图

在图2中：

外部类内部的方框用于表示内部类。另外，“外部类A.内部类B”也用于表示内部类。接口和普通类用同一种框图表示。

本书所使用的UML图都比较简单，读者不必花费大量时间专门学习UML。

这里有必要提醒一下，要阅读此书，应具有Java基本知识。

另外，本书和《深入理解Android卷I》^[1]（简称“卷I”）部分章节有一定联系，主要集中在Binder和MessageQueue部分。读者可将“卷I中”这部分内容作为补充阅读资料来学习。卷I部分内容的电子版下载地址为：
<http://download.csdn.net/detail/hzbooks/3677793>。

本书涉及的Android 4.0源码以及一些开发工具的下载地址为：
<http://115.com/folder/fauqpj0t#Android-ICS-SOURCE-CODE>。

勘误和支持

由于作者的水平有限，加之写作时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评和指正。若有问题，可通过邮箱或在博客上留言与笔者共同讨论。笔者的联系方式是：

邮箱：fanping.deng@gmail.com

博客：blog.csdn.net/innost、cnblogs.net/innost
和http://my.oschina.net/innost/blog

致谢

本书即将付梓！首先要感谢杨福川编辑的大力支持。另外，要感谢本书的审稿编辑姜影。

再一次感谢我所在的中科创达(ThunderSoft)公司。有幸工作在这样一个互相信任、互相鼓励、平等和开放式的环境中，我才能完成本书。公司领导所给予的机会和挑战，时时鞭策着我保持虚心学习的心态。此外，我所在团队的各位同仁也给予了我不少支持和帮助。本书出版之日，将是我们团队为之努力奋斗的Android系统高效、稳定运行于客户手机之时！

一如既往地感谢妻子和家人，他们是我奋斗的动力。

谢谢在人生和职业道路上曾给予我指导的诸位师长。

当然，最应感谢的还是肯花费宝贵时间和精力关注本书的读者，你们的意见和建议，将会使我获得巨大的精神财富！

邓凡平 于北京，中科创达（ThunderSoft）
公司

[1]该书已由机械工业出版社于2011年出版，书号为9787-111-357629。

第1章 搭建Android源码工作环境

本章主要内容：

简单介绍系统架构、编译环境的搭建。

简单介绍利用Eclipse调试system_process进程的方法。

1.1 Android系统架构

到目前为止（2012年），Android系统的最新版本是4.0.3。而就在本书即将完稿之时，业界有传闻说Android 4.0.4版本已经对大厂商发布。Android系统推出速度之快让很多开发人员都很惊讶。当然，推出的速度如此之快有一些是出于商业上的考虑。不管怎样，Android系统在进化过程中，大体架构是相对稳定的。图1-1为Android的系统架构图。

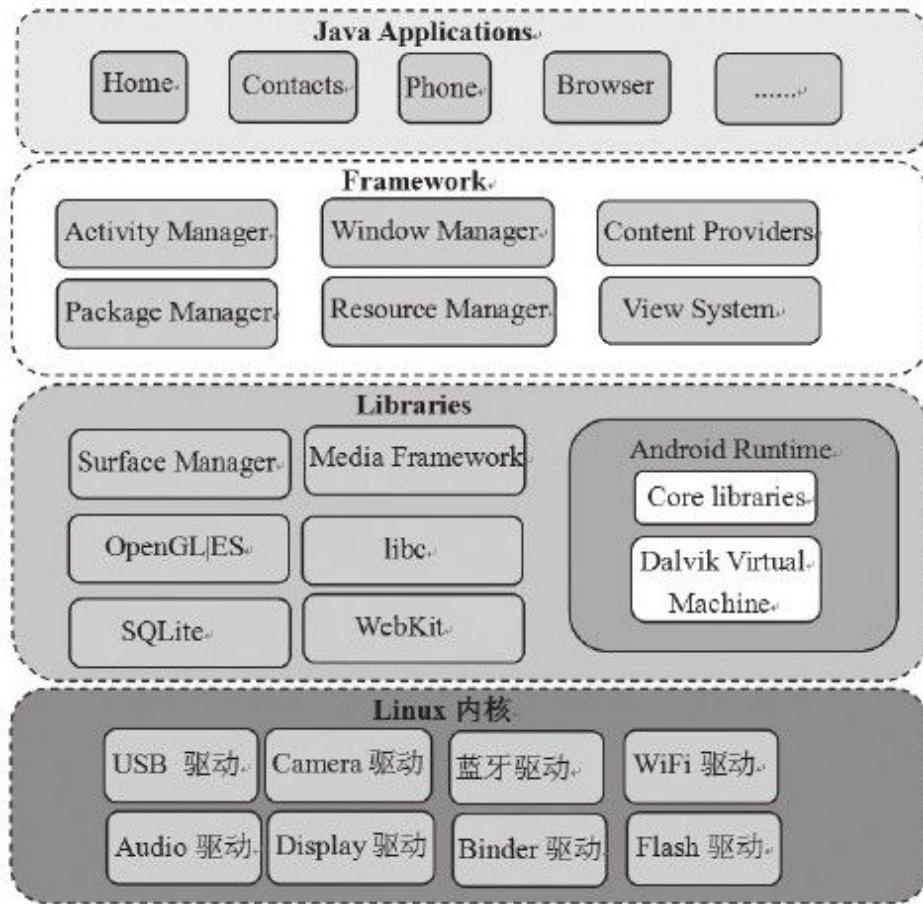


图 1-1 Android系统架构

相信绝大多数读者对图1-1或类似图表的内容已经非常熟悉了。此处，我们就不在赘述。

本书作为“深入理解Android”系列的第二卷，从内容上将承接《深入理解Android：卷I》（本书以后简称“卷I”），但是本书关注的焦点将从Native层Framework转移到Java层Framework。本书涵盖的内容如图1-2所示。

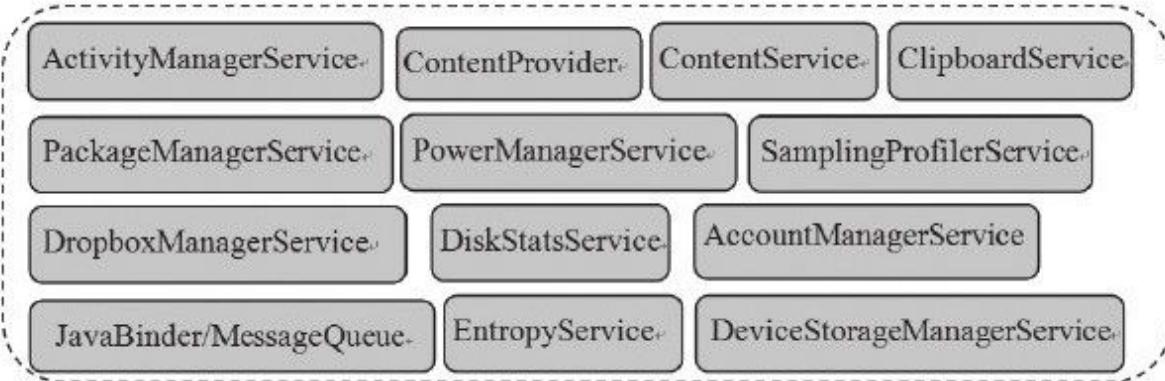


图 1-2 本书涵盖的内容

从图1-2中读者可发现，本书的大部分内容都在讨论Service（服务）。这是因为Android Java层Framework的核心就是这些Service。毫不夸张地说，正是这些Service支撑了整个Java层的运转。读者可参考第3章的图3-1来了解Framework中Service的概貌。

1.2 搭建开发环境

本节将讨论Android 4.0源码下载和编译的方法、Eclipse开发环境的搭建方法，以及system_process进程的调试方法等相关知识。

1.2.1 下载源码

Android 2.3以后，Google官方推荐使用64位的操作系统来编译源码。所以，读者要先安装64位的操作系统（Operating System, OS）。笔者推荐的操作系统是Ubuntu 10.04 X86-64版。另外，要提醒读者的是，不要随意升级Ubuntu，因为高版本Ubuntu中自带的GCC版本过高，会导致编译Android源码时出现问题。

提示 其实32位的Ubuntu也可以编译Android 2.3以后的Android系统源码，但操作颇为麻烦，而且也没什么技术含量。建议读者遵照官方要求去做。

本处不过多讨论Android源码下载的步骤，原因是：这是一个需要读者动手操作的过程，看着

电脑屏幕操作比看书后输入大串的字符串的效率要高很多。

基于上面这个原因，这里笔者向读者提供一个官方说明文档，地址是 <http://source.android.com/source/downloading.html>。该网页中有详细的代码下载步骤，读者只要执行简单的复制和粘贴操作即可下载到源代码。建议读者亲自动手实验。图1-3为该网页的截图。

The screenshot shows the 'Installing Repo' section of the 'Downloading the Source Tree' page. It includes instructions for installing Repo, initializing a client, and running repo init. The page features a light gray background with white text and several code snippets in a dark gray box.

Installing Repo

Repo is a tool that makes it easier to work with Git in the context of Android. For more information about Repo, see [Version Control](#).

To install, initialize, and configure Repo, follow these steps:

- Make sure you have a bin/ directory in your home directory, and that it is included in your path:

```
$ mkdir ~/bin  
$ PATH=~/bin:$PATH
```
- Download the Repo script and ensure it is executable:

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo  
$ chmod a+x ~/bin/repo
```
- The SHA-1 checksum for repo is 29ba4221d4fcdfa8d87931cd73466fd24040b6

Initializing a Repo client

After installing Repo, set up your client to access the android source repository

- Create an empty directory to hold your working files. If you're using MacOS, this has to be on a case-sensitive filesystem. Give it any name you like:

```
$ mkdir WORKING_DIRECTORY  
$ cd WORKING_DIRECTORY
```
- Run repo init to bring down the latest version of Repo with all its most recent bug fixes. You must specify a URL for the manifest, which specifies where the various repositories included in the Android source will be placed within your working directory.

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

To check out a branch other than "master", specify it with -b:

```
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.0.1_r1
```

图 1-3 Android源码下载网页示意图

注意 读者应选择下载Android 4.0.1的源码。虽然最新的Android 4.0.3从版本号上看变化不大，但实际代码却有较大变化。

另外，如果读者发现本章提供的下载地址无法连接，可从笔者博客上的链接去下载源码和工具，笔者博客地址是：
<http://blog.csdn.net/innost/article/details/7525205>。

1.2.2 编译源码

1. 部署JDK

Android 2.3及以后版本的代码编译都需要使用JDK1.6，所以首先要做的就是下载JDK1.6。下载地址是
<http://www.oracle.com/technetwork/java/javasebusiness/downloads/index.html>。笔者下载的文件是jdk-6u27-linux-x64.bin。把它放到一个目录中，比如将其放到/mnt/hgfs/E目录下，然后在这个目录中执行下面这个文件：

```
./jdk-6u27-linux-x64.bin
```

这个命令的作用其实就是解压。解压后的结果在/mnt/hgfs/E/jdk1.6.0_27目录中。有了JDK后，还需要设置~/.bashrc文件。在该文件末尾添加如图1-4所示的几行语句。

```
export JAVA_HOME=/mnt/hgfs/E/jdk1.6.0_27
export JRE_HOME=$JAVA_HOME/jre
export CLASSPATH=$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

图 1-4 Java环境部署示意图

重新登录系统后，Java环境就添加到系统中了。

2. 编译源码

编译源码的步骤非常简单。我们在卷I也详细介绍了编译方法。不过本书要求读者必须先编译整个系统，步骤如下：

执行source build/envsetup.sh命令。该命令将导入Android编译环境。

输入choosecombo并执行，它是在envsetup.sh中定义的一个函数。在执行过程中，分别选择release、generic、eng即可。最终屏幕输出如图1-5所示。

执行make命令以编译整个系统。编译时间由机器配置决定。笔者的4核4GB内存的机器的编译时间大概为2小时。

```
root@innost:/home/.../work-branches/Android-4.0# choosecombo
Build type choices are:
  1. release
  2. debug

Which would you like? [1]

Which product would you like? [full] generic

Variant choices are:
  1. user
  2. userdebug
  3. eng
Which would you like? [eng]

=====
PLATFORM_VERSION CODENAME=REL
PLATFORM_VERSION=4.0.1
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=ITL41D
=====
```

图 1-5 编译设置效果图

1.2.3 利用Eclipse调试system_process

本节将介绍如何利用Eclipse来调试Android Java Framework的核心进程system_process。

1.配置Eclipse

首先要下载Android SDK，下载地址为<http://developer.android.com/sdk/index.html>。在Linux环境下，该网站截图如图1-6所示。

Platform	Package	Size	MD5 Checksum
Windows	android-sdk_r18-windows.zip	37448775 bytes	bfbfdf8b2d0fdecc2a621544d706fa98
	installer_r18-windows.exe (Recommended)	37456234 bytes	48b1fe7b431afe6b9c8a992bf75dd898
Mac OS X (intel)	android-sdk_r18-macosx.zip	33903758 bytes	8328e8a5531c9d6f6f1a0261cb97af36
Linux (i386)	android-sdk_r18-linux.tgz	29731463 bytes	6cd716d0e04624b865ffed3c25b3485c

图 1-6 Android SDK下载网页截图

笔者下载的是Linux系统上的SDK。解压后的位置在 /thunderst/disk/android/android-sdk-linux_86 |下。

然后要为Eclipse安装ADT插件（Android Development Tools），步骤如下：

单击 Eclipse 菜单栏 Help 下的 Install New Software , 输入 Android ADT 下载地址 : <https://dl-ssl.google.com/android/eclipse/> , 然后安装其中的所有组件，并重启 Eclipse。

单击 Eclipse 菜单栏 Preferences 下的 Android 一栏，在右边的 SDK Location 文本框中输入刚才解压 SDK 后得到的目录（笔者设置的为 /thunderst/disk/android/android-sdk-linux_86 ） , 最终结果如图 1-7 所示。

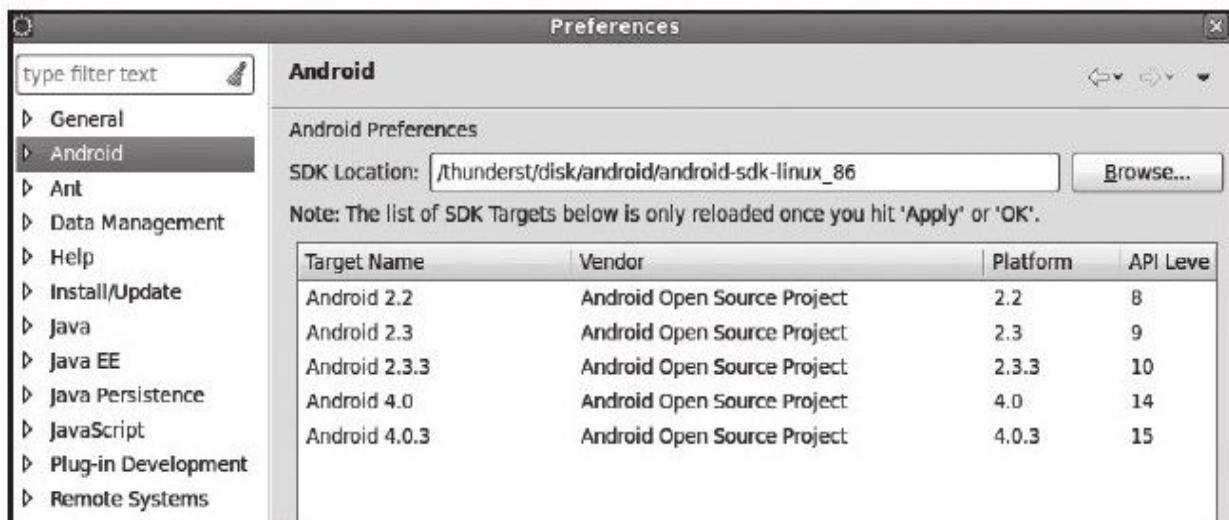


图 1-7 SDK 安装示意图

单击 Eclipse 菜单栏 Window 下的 Android SDK Manager , 弹出一个对话框 , 如图 1-8 所示。

在图 1-8 中选择下载其中的 Tools 和对应版本的 SDK 文档、镜像文件、开发包等。有条件的读者

可以将Android 4.0.3和Android 4.0对应的内容及Tools全部下载下来。

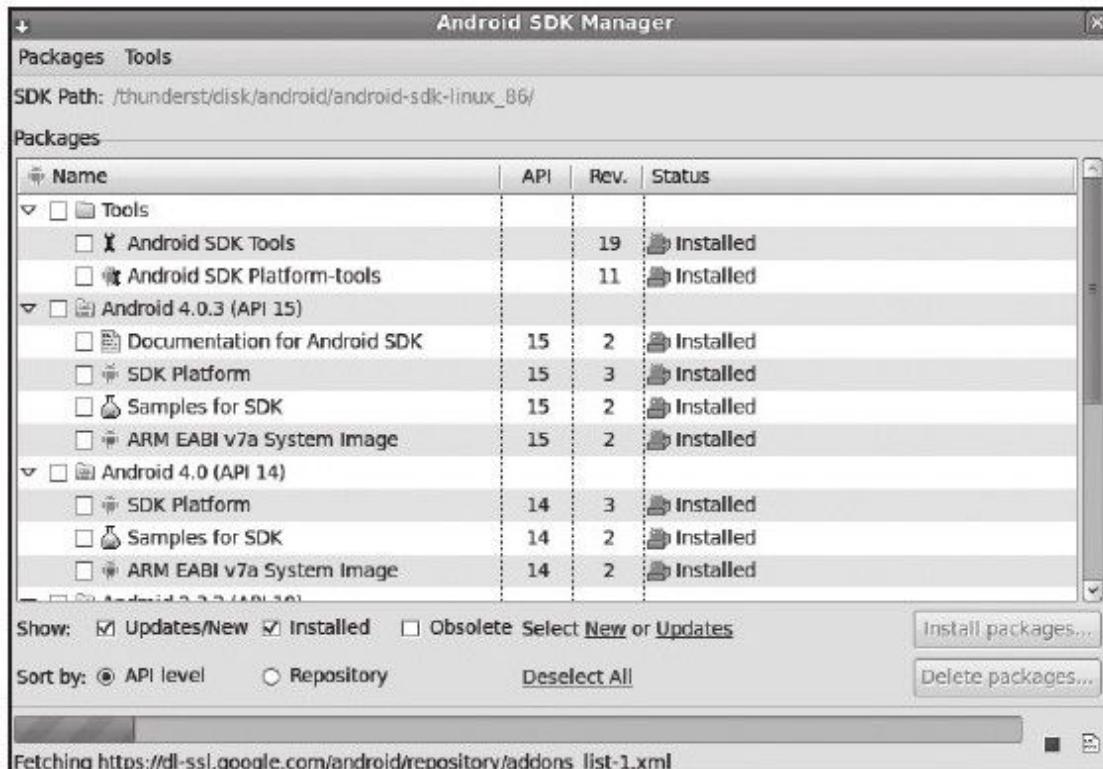


图 1-8 Android SDK Manager对话框

2. 使用Coffee Bytes Java插件

Coffee Bytes Java是Eclipse的一个插件，用于对代码进行折叠，其功能比Eclipse自带的代码折叠功能强大很多。在研究大段代码时，该插件的作用非常明显。此插件的安装和配置步骤如下：

单击 Eclipse 菜单栏 Help 下的 Install New Software，在弹出的对话框中输入

<http://eclipse.realjenius.com/update-site>，选择安装这个插件即可。

单击Eclipse菜单栏Window下的Preference，在左上角输入Folding进行搜索，结果如图1-9所示。

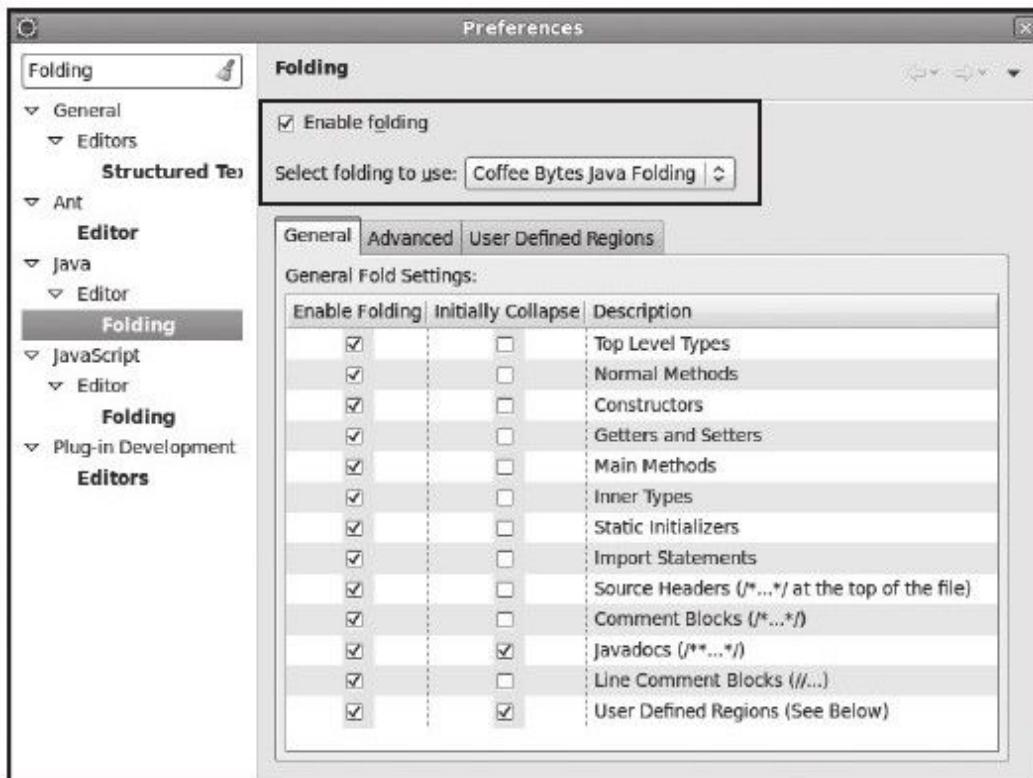


图 1-9 Coffee Bytes Java插件配置

在图1-9所示的对话框中，要先勾选Enable folding选项，然后从Select folding to use框中选择Coffee Bytes Java Folding（见图1-9中的黑框）。图1-9右下部分的复选框用来配置此插件的代码折叠功能。读者不妨按照图1-9所示来配置它。使用该插件后的示意图如图1-10所示。

```
boolean onlyCore=false;
if (ENCRYPTING_STATE.equals(cryptState)){
    Slog.w(TAG, "Detected encryption in progress...only parsing core apps");
    onlyCore=true;
} else if (ENCRYPTED_STATE.equals(cryptState)){
}

pm=PackageManagerService.main(context,
    factoryTest!=SystemServer.FACTORY_TEST_OFF,
    onlyCore);
boolean firstBoot=false;
try {
    firstBoot=pm.isFirstBoot();
} catch (RemoteException e) {
}

```

如果不关心 else if 分支，就可以把这段代码折叠起来

图 1-10 Coffee Bytes Java插件的使用示例

从图1-10中可看到，使用该插件后，代码中所有分支基本上都可以折叠起来，该功能将帮助开发人员集中精力关注自己所关心的分支。

3.导入Android源码

注意，导入源码必须在编译完整个Android源码后才可以实施，其步骤如下：

复 制 Android 源 码 目
录 /development/ide/eclipse/.classpath 到 Android 源 码
根 目 录。

打开Android源码根目录下的.classpath文件。该文件是供Eclipse使用的，其中保存的是源码目录中各个模块的路径。由于我们只关心Framework相关的模块，因此可以把一些不是Framework的目录从该文件中注释掉。同时，去掉不必要的模块

也可加快Android源码导入速度。图1-11所示为该文件的部分内容。

```
→<classpathentry.kind="src".path="frameworks/base/cmds/am/src"/>
→<classpathentry.kind="src".path="frameworks/base/cmds/input/src"/>
→<classpathentry.kind="src".path="frameworks/base/cmds/pm/src"/>
→<classpathentry.kind="src".path="frameworks/base/cmds/svc/src"/>
→<classpathentry.kind="src".path="frameworks/base/core/java"/>
→<classpathentry.kind="src".path="frameworks/base/drm/java"/>
→<classpathentry.kind="src".path="frameworks/base/graphics/java"/>
→<classpathentry.kind="src".path="frameworks/base/icu4j/java"/>
→<classpathentry.kind="src".path="frameworks/base/keystore/java"/>
→<classpathentry.kind="src".path="frameworks/base/location/java"/>
→<classpathentry.kind="src".path="frameworks/base/media/java"/>
→<classpathentry.kind="src".path="frameworks/base/obex"/>
→<classpathentry.kind="src".path="frameworks/base/opengl/java"/>
→<classpathentry.kind="src".path="frameworks/base/packages/SettingsProvider/src"/>
→<classpathentry.kind="src".path="frameworks/base/packages/SystemUI/src"/>
→<classpathentry.kind="src".path="frameworks/base/policy/src"/>
→<classpathentry.kind="src".path="frameworks/base/sax/java"/>
→<classpathentry.kind="src".path="frameworks/base/services/java"/>
→<classpathentry.kind="src".path="frameworks/base/telephony/java"/>
→<classpathentry.kind="src".path="frameworks/base/test-runner/src"/>
→<classpathentry.kind="src".path="frameworks/base/voip/java"/>
→<classpathentry.kind="src".path="frameworks/base/wifi/java"/>
→<classpathentry.kind="src".path="frameworks/ex/carousel/java"/>
→<classpathentry.kind="src".path="frameworks/ex/chips/src"/>
→<classpathentry.kind="src".path="frameworks/ex/common/java"/>
→<classpathentry.kind="src".path="frameworks/ex/variablespeed/src"/>
→<classpathentry.kind="src".path="frameworks/opt/calendar/src"/>
→<classpathentry.kind="src".path="frameworks/opt/vcard/java"/>
```

图 1-11 .classpath文件的部分内容

另外，在Eclipse环境中，一些不必要的模块会导致后续Android源码编译失败。笔者共享了一个已经配置好的.classpath文件，读者可从<http://download.csdn.net/detail/innost/4247578>处下载并直接使用。

单击Eclipse菜单栏New下的Java Project，弹出如图1-12所示的对话框。设置Location为Android4.0源码所在的路径。

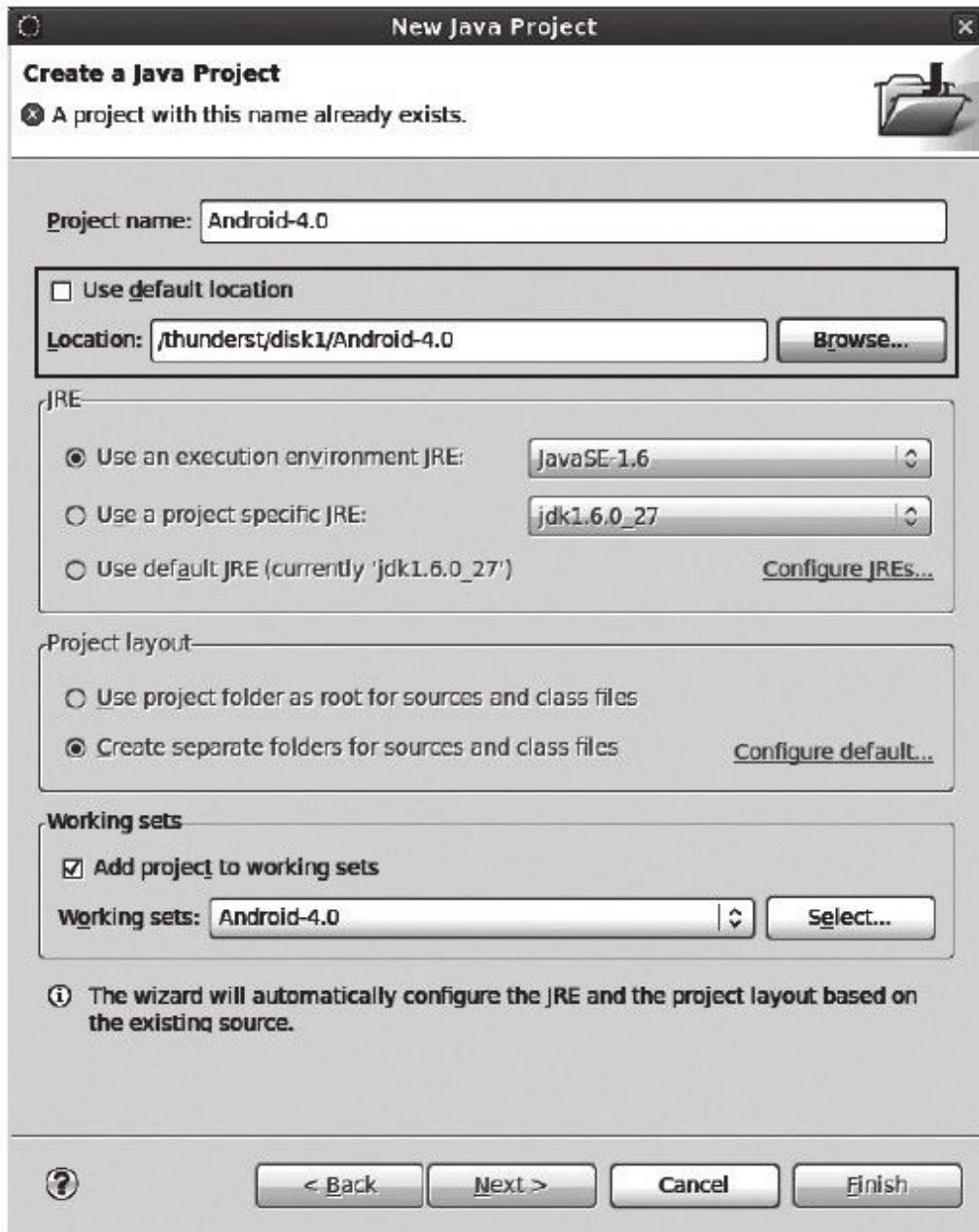


图 1-12 导入Android源码示意图

由于Android 4.0源码文件较多，导入过程会持续较长一段时间，大概十几分钟。

注意 导入源码前一定要取消Eclipse的自动编译选项（通过菜单栏Project下的Build Project

Automatically设置)。另外，源码导入完毕后，读者千万不要清理(clean)这个工程。清理会删除之前源码编译生成的文件，导致后续又得重新编译Android系统。

4. 创建并运行模拟器

单击Eclipse菜单栏Window下的AVD Manager，创建模拟器，如图1-13所示。

模拟器创建完毕后即可启动它。

5. 调试system_process

调试system_process的步骤如下：

首先编译Android源码工程。编译过程中会有很多警告。如果有错误，大部分原因是.classpath文件将不需要的模块包含了进来。读者可根据Eclipse的提示做相应处理。

在Android源码工程上右击，在弹出的菜单中依次单击Debug As->Debug Configurations，弹出如图1-14所示的对话框，然后从左边找到Remote Java Application一栏。

单击图1-14左上角黑框中的新建按钮，然后按图1-15所示的黑框中的内容来设置该对话框。

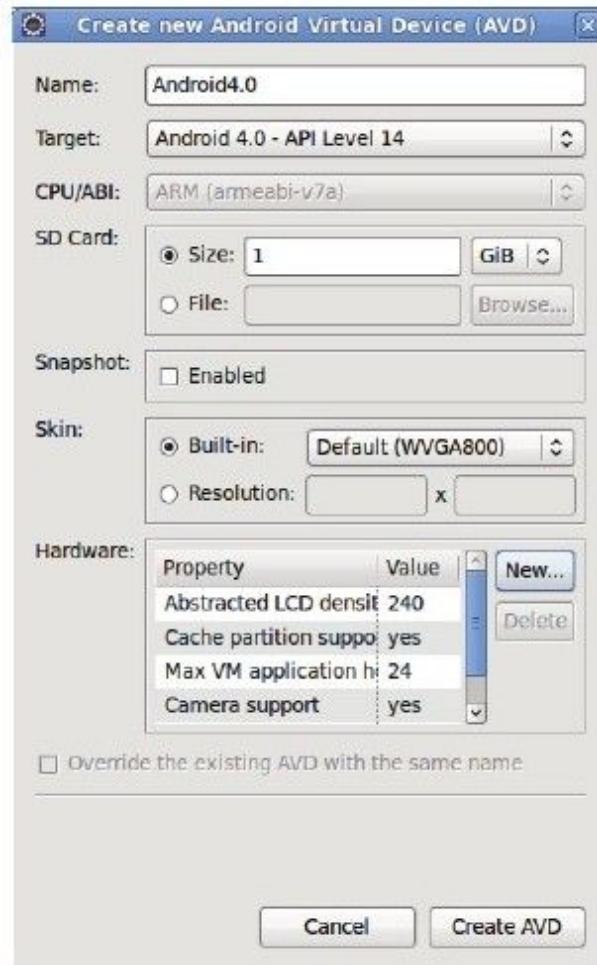


图 1-13 模拟器创建示意图

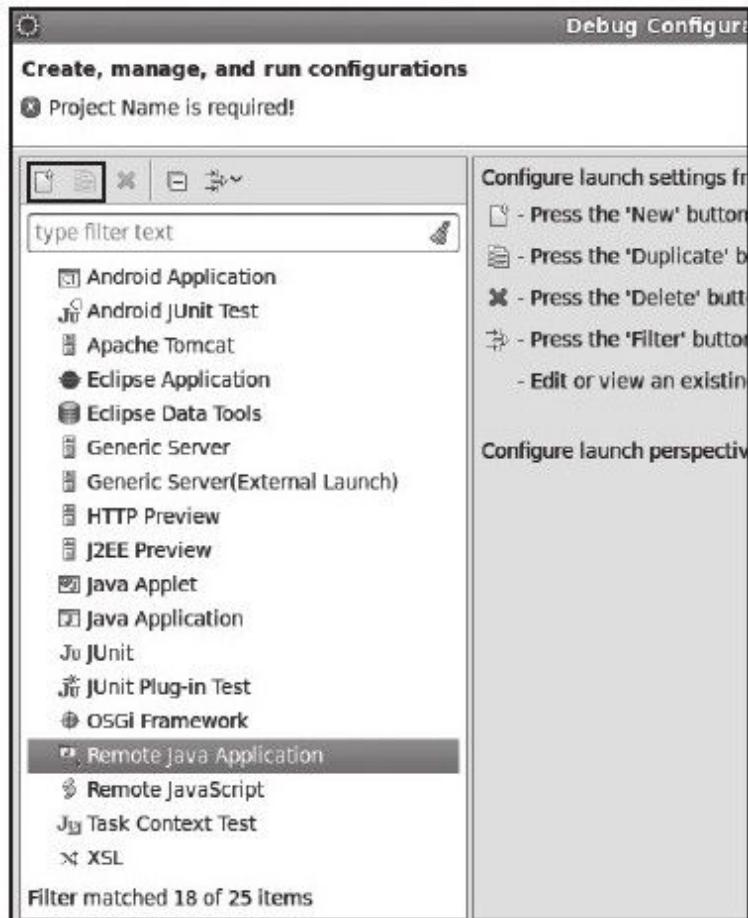


图 1-14 Debug配置框示意图

由图1-15可知，需要选择Remote调试端口号为8600，Host类型为localhost。8600是system_process进程的调试端口号。Eclipse一旦连接到该端口，即可通过JDWP协议来调试system_process。

配置完毕后，单击图1-15所示对话框右下角的Debug按钮，即可启动system_process的调试。

图1-16所示为笔者调试startActivity流程的示意图。

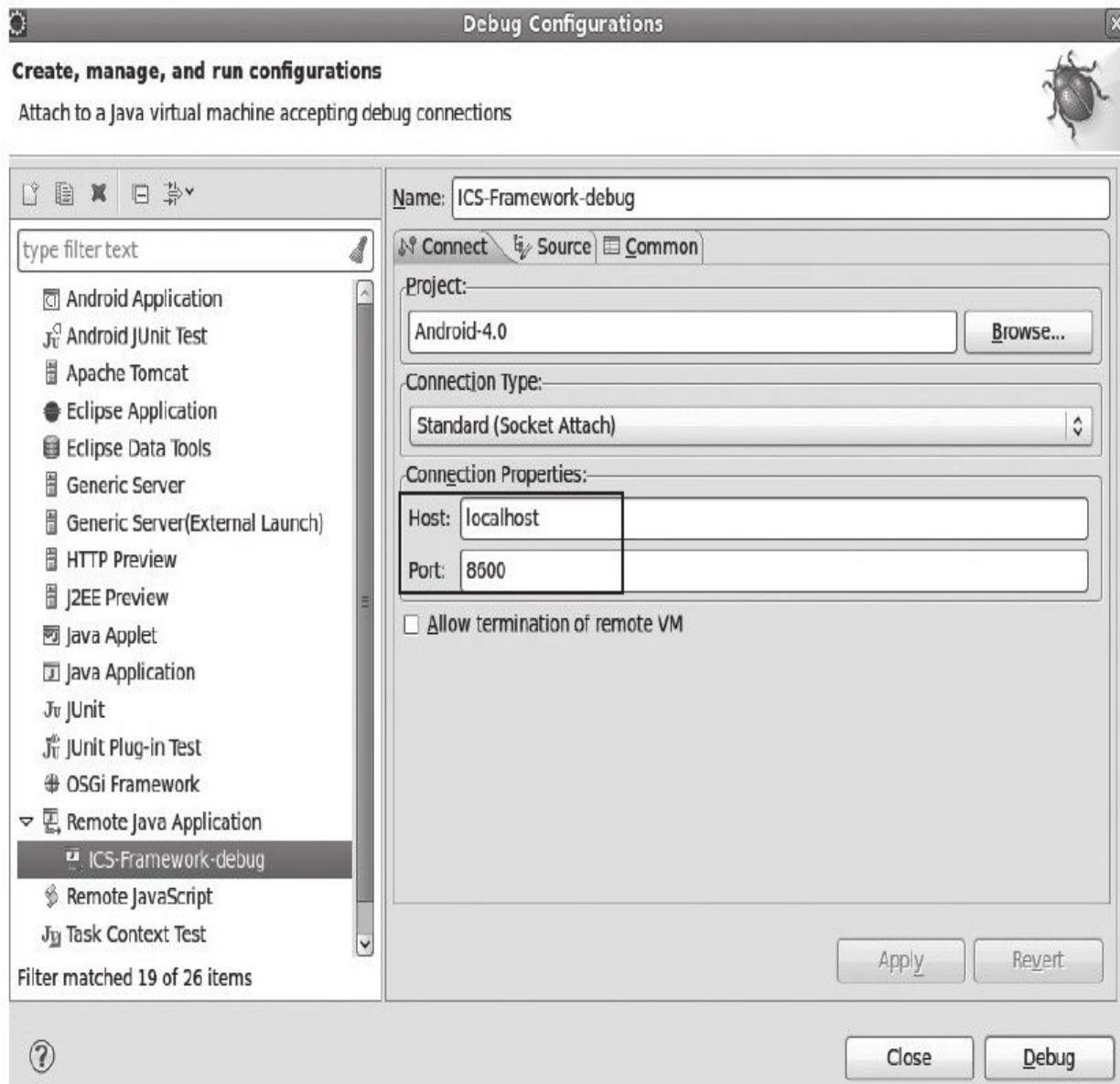


图 1-15 Remote Java Application配置示意图

```
ActivityManagerService.java | ActivityStack.java | 
778     ...
779     return aInfo;
780   }
781   ...
782   final int startActivityMayWait(IApplicationThread caller, int callingUid, 
783       Intent intent, String resolvedType, Uri[] grantedUriPermissions, 
784       int grantedMode, IBinder resultTo, 
785       String resultWho, int requestCode, boolean onlyIfNeeded, 
786       boolean debug, String profileFile, ParcelFileDescriptor profileFd, 
787       boolean autoStopProfiler, WaitResult outResult, Configuration config
788       // Refuse possible leaked file descriptors
789       if (intent != null && intent.hasFileDescriptors()) {
790           throw new IllegalArgumentException("File descriptors passed in Intent");
791       }
792   }
793   boolean componentSpecified = intent.getComponent() != null;
794   ...
795   // Don't modify the client's object!
796   intent = new Intent(intent);

```

The screenshot shows the Eclipse IDE with the DDMS perspective open. The left pane displays the Java code for `ActivityManagerService.java`. A breakpoint is set at line 279. The right pane shows the `Threads` tab of the DDMS tool, listing various threads. One thread is highlighted and marked as suspended, indicating it is stopped at the breakpoint.

图 1-16 system_process 调试效果图

另外，system_process 的调试端口号可由 DDMS查询，如图1-17所示。

由图1-17可知，在笔者测试的这台机器上，system_process的调试端口号为8611，由于系统有可能因为意外而重启，故system_process的端口号并不固定为8600。读者在调试它的时候最好先通过DDMS确认当前system_process的调试端口号。

The screenshot shows the Android Device Monitor (DDMS) interface. The top bar has tabs for Devices, Log, CPU, Memory, Network, and Storage. The Devices tab is active, displaying a table of running processes. The columns are labeled Name, PID, CPU %, and Port. The table lists various apps and system processes, with the row for 'system_process' highlighted in dark gray.

Name	PID	CPU %	Port
com.google.android.gsf.login	13612		8605
com.bel.android.dspmanager	31025		8606
com.google.android.gallery3d	22170		8607
com.sina.weibo	14149		8608
com.android.phone	1612		8609
com.koushikdutta.rommanager	1613		8610
system_process	1463		8611 / 8700
com.anddoes.fancywidgets	13718		8612
com.qihoo360.mobilesafe	14669		8613
com.android.systemui	1529		8614
com.jiasoft.swreader	543		8615
com.google.android.partnersetup	24013		8616
com.cyanogenmod.trebuchet	1626		8617
com.android.settings	16809		8618
com.google.android.calendar	1562		8619

图 1-17 查询system_process调试端口号

1.3 本章小结

本章对Android系统和源码搭建，以及如何利用Eclipse调试system_process等做了相关介绍，相信读者现在已经迫不及待了吧？马上开始我们的源码征程吧！

第2章 深入理解Java Binder和MessageQueue

本章主要内容：

分析Binder系统的Java层框架。

分析MessageQueue。

本章所涉及的源代码文件名及位置：

IBinder. java
(frameworks/base/core/java/android/os/IBinder.java
)

Binder. java
(frameworks/base/core/java/android/os/Binder.java
)

BinderInternal. java
(frameworks/base/core/java/com/android/internal/os/
/BinderInternal.java)

android_util_Binder. cpp
(frameworks/base/core/jni/android_util_Binder.cpp
)

SystemServer. java
(frameworks/base/services/java/com/android/server/
SystemServer.java)

ActivityManagerService. java
(frameworks/base/services/java/com/android/server/
am/ActivityManagerService.java)

ServiceManager. java
(frameworks/base/core/java/android/os/ServiceMan
ager.java)

ServiceManagerNative. java
(frameworks/base/core/java/android/os/ServiceMan
agerNative.java)

MessageQueue. java
(frameworks/base/core/java/android/os/MessageQue
ue.java)

android_os_MessageQueue. cpp
(frameworks/base/core/jni/android_os_MessageQue
ue.cpp)

Looper. cpp
(frameworks/base/libs/utils/android/Looper.cpp)

Looper.h
(frameworks/base/include/utils/Looper.h)

android_app_NativeActivity.cpp
(frameworks/base/core/jni/android_app_NativeActivity.cpp)

2.1 概述

本章作为本书Android分析之旅的开篇，将重点关注两个基础知识点：

Binder系统在Java世界是如何布局和工作的。

MessageQueue的新职责。

先来分析Java层中的Binder架构。

建议 读者先阅读卷I的第6章“深入理解Binder”。网上有样章可供下载。下载地址为：
<http://download.csdn.net/detail/hzbooks/3677793>。

2.2 Java层中的Binder架构分析

2.2.1 Binder架构总览

如果读者读过卷I第6章“深入理解Binder”，相信就不会对Binder架构中代表Client的Bp端及代表Server的Bn端感到陌生。Java层中Binder实际上也是一个C/S架构，而且其在类的命名上尽量保持与Native层一致，因此可认为，Java层的Binder架构是Native层Binder架构的一个镜像。Java层的Binder架构中的成员如图2-1所示。

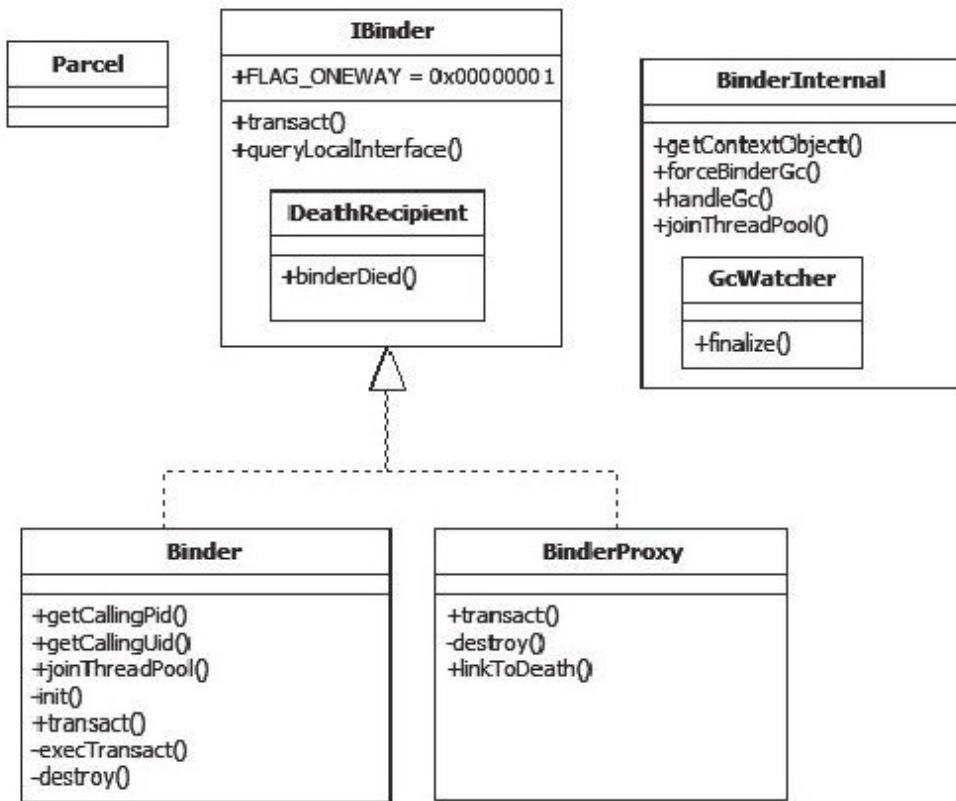


图 2-1 Java层中的Binder家族

由图2-1可知：

系统定义了一个 `IBinder` 接口类以及 `DeathRecipient` 接口。

`Binder`类和`BinderProxy`类分别实现了`IBinder`接口。其中，`Binder`类作为服务端的Bn的代表，而`BinderProxy`作为客户端的Bp的代表。

系统中还定义了一个`BinderInternal`类。该类是一个仅供`Binder`架构使用的类。它内部有一个

GcWatcher类，该类专门用于处理和Binder架构相关的垃圾回收。

Java层同样提供一个用于承载通信数据的Parcel类。

注意，IBinder接口类中定义了一个名为FLAG_ONEWAY的整型变量，该变量的意义非常重要。当客户端利用Binder机制发起一个跨进程的函数调用时，调用方（即客户端）一般会阻塞，直到服务端返回结果。这种方式和普通的函数调用是一样的。但是在调用Binder函数时，如果指明了FLAG_ONEWAY标志，则调用方只要把请求发送到Binder驱动即可返回，而不用等待服务端的结果，这就是一种所谓的非阻塞方式。在Native层中，涉及的Binder调用基本都是阻塞的，但是在Java层的framework中，使用FLAG_ONEWAY进行Binder调用的情况非常多，以后经常会碰到。

思考 使用FLAG_ONEWAY进行函数调用的程序在设计上有什么特点？这里简单分析一下：对于使用FLAG_ONEWAY的函数来说，客户端仅向服务端发出了请求，但是并不能确定服务端是否处理了该请求。所以，客户端一般会向服务端注册一个回调（同样是跨进程的Binder调用），一旦服务端处理了该请求，就会调用此回调函数来

通知客户端处理结果。当然，这种回调函数也大多采用FLAG_ONEWAY的方式。

2.2.2 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，在Java世界初创时期，系统会提前注册一些JNI函数，其中有一个函数专门负责搭建Java Binder和Native Binder的交互关系，该函数是register_android_os_Binder，代码如下：

[-->android_util_Binder.cpp :
register_android_os_Binder]

```
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，register_android_os_Binder函数完成了Java层Binder架构中最重要的4个类的初始化工作。我们重点关注前3个。

1.Binder类的初始化

int_register_android_os_Binder函数完成了Binder类的初始化工作，代码如下：

[-->android_util_Binder.cpp :
int_register_android_os_Binder]

```
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。

建议 如果读者对JNI不是很清楚，可参阅卷I第2章“深入理解JNI”。

2.BinderInternal类的初始化

下一个初始化的类是BinderInternal，其代码在int_register_android_os_BinderInternal函数中。

[-->android_util_Binder.cpp :
int_register_android_os_BinderInternal]

```
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) ) ;  
}
```

int_register_android_os_BinderInternal 的工作内容和int_register_android_os_Binder的工作内容类似，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。

3.BinderProxy类的初始化

int_register_android_os_BinderProxy 完成了 BinderProxy类的初始化工作，代码稍显复杂，如下所示：

```
[-->android_util_Binder.cpp :  
int_register_android_os_BinderProxy]]
```

```
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。

至此，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。

这几个对象的命名中都有一个Offsets，我觉得这非常别扭，不知道读者是否有同感。

框架的初始化其实就是提前获取一些JNI层的使用信息，如类成员函数的MethodID、类成员变量的FieldID等。这项工作是必需的，因为它能节省每次使用时获取这些信息的时间。当Binder调用频繁时，这些时间累积起来还是不容小觑的。

下面我们通过一个例子来分析Java层Binder的工作流程。

2.2.3 addService实例分析

这个例子源自ActivityManagerService(AMS)，我们通过它揭示Java层Binder的工作原理。该例子的分析步骤如下：

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。

本例的起点是setSystemProcess函数，其代码如下所示：

[-->ActivityManagerService.java :
setSystemProcess]

```
public static void setSystemProcess () {  
    try{  
        ActivityManagerService m=mSelf ;  
        //将ActivityManagerService服务注册到ServiceManager中  
        ServiceManager.addService ("activity", m) ;  
        .....//省略后续内容  
    }  
    .....  
    return ;  
}
```

上面所示代码行的目的是将AMS服务注册到ServiceManager中。AMS是Android核心服务中的核心，以后我们会经常和它打交道。

大家知道，整个Android系统中有一个Native的ServiceManager（以后简称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。

1. 在SM中注册服务

(1) 创建ServiceManagerProxy

在SM中注册服务的函数为addService，其代码如下：

[-->ServiceManager.java : addService]

```
public static void addService ( String name, IBinder service) {  
    try{  
        //getIServiceManager返回什么  
        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 ;  
}
```

asInterface 的参数为
BinderInternal.getContextObject的返回值。这是一个native的函数，其实现的代码为：

[-->android_util_Binder.cpp :
 android_os_BinderInternal_getContextObject]

```
static jobject android_os_BinderInternal_getContextObject (   
    JNIEnv*env, jobject clazz) {  
/*  
下面这句代码，我们在卷I第6章详细分析过，它将返回一个BpProxy对象，其中  
NULL（即0，用于标识目的端）指定Proxy通信的目的端是ServiceManager  
*/  
    sp<IBinder>b=ProcessState : self ( ) ->getContextObject  
(NULL) ;  
    //由Native对象创建一个Java对象，下面分析该函数  
    return javaObjectForIBinder (env, b) ;  
}
```

[-->android_util_Binder.cpp :
 javaObjectForIBinder]

```
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，读者可自行分析 BpBinder.cpp 中的 findObject 函数。

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 ;
}
```

BinderInternal. getContextObject该函数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。

大家还记得Native层Binder中那个著名的interface_cast宏吗？在Java层中，虽然没有这样的宏，但是定义了一个类似的函数asInterface。下面来分析ServiceManagerNative类的asInterface函数，其代码如下：

[-->ServiceManagerNative.java : asInterface]

```
static public IServiceManager asInterface (IBinder obj)
{
.....//以obj为参数，创建一个ServiceManagerProxy对象
return new ServiceManagerProxy (obj) ;
}
```

上面的代码和Native层interface_cast非常类似，都是以一个BpProxy对象为参数构造一个和业务相关的Proxy对象，例如这里的ServiceManagerProxy对象。Service-ManagerProxy对象的各个业务函数会将相应请求打包后交给BpProxy对象，最终由BpProxy对象发送给Binder驱动以完成一次通信。

提示 实际上BpProxy也不会和Binder驱动交互，真正和Binder驱动交互的是IPCThreadState。

(2) addService函数分析

现在来分析ServiceManagerProxy的addService函数，其代码如下：

[-->ServiceManagerNative.java : addService]

```
public void addService (String name, IBinder service)
throws RemoteException{
Parcel data=Parcel.obtain () ;
Parcel reply=Parcel.obtain () ;
data.writeInterfaceToken (IServiceManager.descriptor) ;
data.writeString (name) ;
```

```
//注意下面这个writeStrongBinder函数，后面我们会详细分析它  
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，其实现代码如下：

[-->android_util_Binder.cpp :
android_os_Binder Proxy_transact]

```
static jboolean android_os_BinderProxy_transact (JNIEnv*env,  
jobject obj,  
jint code, jobject dataObj,  
jobject replyObj, jint flags)  
{  
....  
//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) ;
.....
//通过Native的BpBinder对象，将请求发送给ServiceManager
status_t err=target->transact (code, *data, reply, flags) ;
.....
signalExceptionForError (env, obj, err) ;
return JNI_FALSE ;
}
```

看了上面的代码你会发现，Java层的Binder架构最终还是要借助Native的Binder架构进行通信。

关于Binder架构，笔者有一个体会愿和读者一起讨论分析。

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，架构的编写不论采用的是Native语言还是Java语言，只要把请求传递到Binder驱动就可以了，所以通信的目的是向binder发送请求和接收回复。在这个目的之上，考虑到软件的灵活性和可扩展性，于是编写了一个架构。反过来说，也可以不使用架构（即没有使用任何接口、派生之类的东西）而直接和binder交互，例如，ServiceManager作为Binder架构的一个核心程序，就是直接读取/dev/binder设备，获取并处理请求。从这一点上看，Binder架构的目的虽是简单的（即打开binder设备，然后读请求和写回

复），但是架构是复杂的（编写各种接口类和封装类等）。我们在研究源码时，一定要先搞清楚目的。实现只不过是达到该目的的一种手段和方式。脱离目的而去研究实现，如缘木求鱼，很容易偏离事物本质。

在对 addService 进行分析时，我们曾提示 writeStrongBinder 是一个特别的函数。那么它特别在哪里呢？我们后边会进行介绍。

(3) Binder 、 JavaBBinderHolder 和 JavaBBinder

ActivityManagerService 从 ActivityManagerNative 类派生，并实现了一些接口，其中和 Binder 架构相关的只有这个 ActivityManagerNative 类，其原型如下：

[-->ActivityManagerNative.java]

```
public abstract class ActivityManagerNative  
extends Binder  
implements IActivityManager
```

ActivityManagerNative 从 Binder 派生，并实现了 IActivityManager 接口。下面来看 ActivityManagerNative 的构造函数：

[-->ActivityManagerNative.java]

```
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 () ;  
}
```

Binder构造函数中会调用native的init函数，其实现的代码如下：

[-->android_util_Binder.cpp : android_os_Binder_init]

```
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) ;  
}
```

从上面代码可知，Java的Binder对象将和一个Native 的 JavaBBinderHolder 对象相关联。JavaBBinderHolder的定义如下：

[-->android_util_Binder.cpp]

```
class JavaBBinderHolder : public RefBase
{
public:
    sp<JavaBBinder> get (JNIEnv*env, jobject obj)
    {
        AutoMutex_l (mLock) ;
        sp<JavaBBinder> b=mBinder.promote () ;
        if (b==NULL) {
            //创建一个JavaBBinder, obj实际上是Java层中的Binder对象
            b=new JavaBBinder (env, obj) ;
            mBinder=b ;
        }
        return b ;
    }
    .....
private:
    Mutex mLock ;
    wp<JavaBBinder>mBinder ;
};
```

从派生关系上可以发现，JavaBBinderHolder 仅从 RefBase 派生，所以它不属于 Binder 家族。Java 层的 Binder 对象为什么 会 和 Native 层的一个与 Binder 家族 无 关 的 对象 绑定 呢？仔细 观察 JavaBBinderHolder 的 定义 可 知：JavaBBinderHolder 类的 get 函数 中 创建 了一个 JavaBBinder 对象，这个 对象 就是 从 BnBinder 派生 的。

那么，这个get函数是在哪里调用的？答案在下面这句代码中：

```
//其中，data是Parcel对象，service此时还是ActivityManagerService  
data.writeStrongBinder (service) ;
```

writeStrongBinder会做一个替换工作，下面是它的native代码实现：

[-->android_util_Binder.cpp :
android_os_Parcel_WriteStrong Binder]

```
static void android_os_Parcel_writeStrongBinder  
(JNIEnv*env,  
 jobject clazz, jobject object)  
{  
 //parcel是一个Native的对象，writeStrongBinder的真正参数是  
 //ibinderForJavaObject的返回值  
 const status_t err=parcel->writeStrongBinder (  
 ibinderForJavaObject (env, object) ) ;  
}
```

[-->android_util_Binder.cpp :
ibinderForJavaObject]

```
sp<IBinder>ibinderForJavaObject (JNIEnv*env, jobject obj)  
{  
 //如果Java的obj是Binder类，则首先获得JavaBBinderHolder对象，然后  
 //调用  
 //它的get函数。而这个get函数将返回一个JavaBBinder  
 if (env->IsInstanceOf (obj, gBinderOffsets.mClass) ) {
```

```
JavaBBinderHolder* jbh= (JavaBBinderHolder*) env->GetIntField  
(obj,  
    gBinderOffsets.mObject) ;  
return jbh !=NULL? jbh->get (env, obj) :NULL ;  
}  
//如果obj是BinderProxy类，则返回Native的BpBinder对象  
if (env->IsInstanceOf (obj, gBinderProxyOffsets.mClass) ) {  
    return (IBinder*)  
env->GetIntField (obj, gBinderProxyOffsets.mObject) ;  
}  
return NULL ;  
}
```

通过上面的介绍你会发现，addService实际添加到Parcel的并不是AMS本身，而是一个名为JavaBBinder的对象。addService正是该JavaBBinder对象最终传递到Binder驱动。

读者此时容易想到，Java层中所有的Binder对应的都是这个JavaBBinder。当然，不同的Binder对象对应不同的JavaBBinder对象。

图2-2展示了Binder、JavaBBinderHolder和Java-BBINDER的关系。

由图2-2可知：

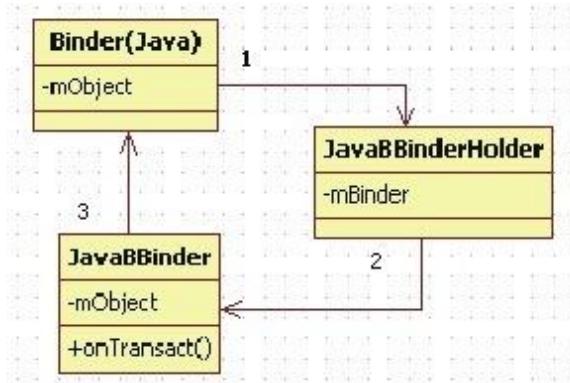


图 2-2 Binder、JavaBBinderHolder和JavaBBinder三者的关系

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。

为什么不直接让Java 层的 Binder 对象指向 Native层的JavaBBinder对象呢？由于缺乏设计文 档，这里不便妄加揣测，但从JavaBBinderHolder 的实现上来分析，估计和垃圾回收（内存管理）有关，因为JavaBBinderHolder中的mBinder对象的 类型被定义成弱引用wp了。

建议 对此，如果读者有更好的解释，不妨登录笔者的博客与大家分享一下。另外，读者务必阅读“卷I”第6章以搞清楚ServiceManager最终是如何处理addService请求的。

2.ActivityManagerService响应请求

初见JavaBBinder时，你可能多少有些吃惊。回想一下Native层的Binder架构：虽然在代码中调用的是Binder类提供的接口，但其对象却是一个实际的服务端对象，例如MediaPlayerService对象、AudioFlinger对象。

而Java层的Binder架构中，JavaBBinder却是一个和业务完全无关的对象。那么，这个对象如何实现不同业务呢？为回答此问题，我们必须看它的onTransact函数。当收到请求时，系统会调用这个函数。

关于这个问题，建议读者阅读卷I“第6章深入理解Binder”。

[-->android_util_Binder.cpp : onTransact]

```
virtual status_t onTransact (
    uint32_t code, const Parcel & data, Parcel* reply, uint32_t
flags=0)
{
    JNIEnv* env=javavm_to_jnienv (mVM) ;
```

```
IPCThreadState*thread_state=IPCThreadState::self() ;  
.....  
//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) ;  
.....  
return res !=JNI_FALSE?NO_ERROR:UNKNOWN_TRANSACTION ;  
}
```

就本例而言，上面代码中的mObject就是ActivityManagerService，现在调用它的execTransact函数，该函数在Binder类中实现，具体代码如下：

[-->Binder.java : execTransact]

```
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) ;  
    }.....  
    reply.recycle () ;  
    data.recycle () ;  
    return res ;  
}
```

ActivityManagerNative类实现了onTransact函数，代码如下：

[-->ActivityManagerNative.java : onTransact]

```
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 () ;
        .....
        //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，JavaBBinder仅是一个传声筒，它本身不实现任何业务函数，其工作是：

当它收到请求时，只是简单地调用它所绑定的Java层Binder对象的execTransact。

该Binder对象的execTransact调用其子类实现的onTransact函数。

子类的onTransact函数将业务又派发给其子类来完成。请读者务必注意其中的多层继承关系。

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。图2-3展示了AMS响应请求的整个流程。

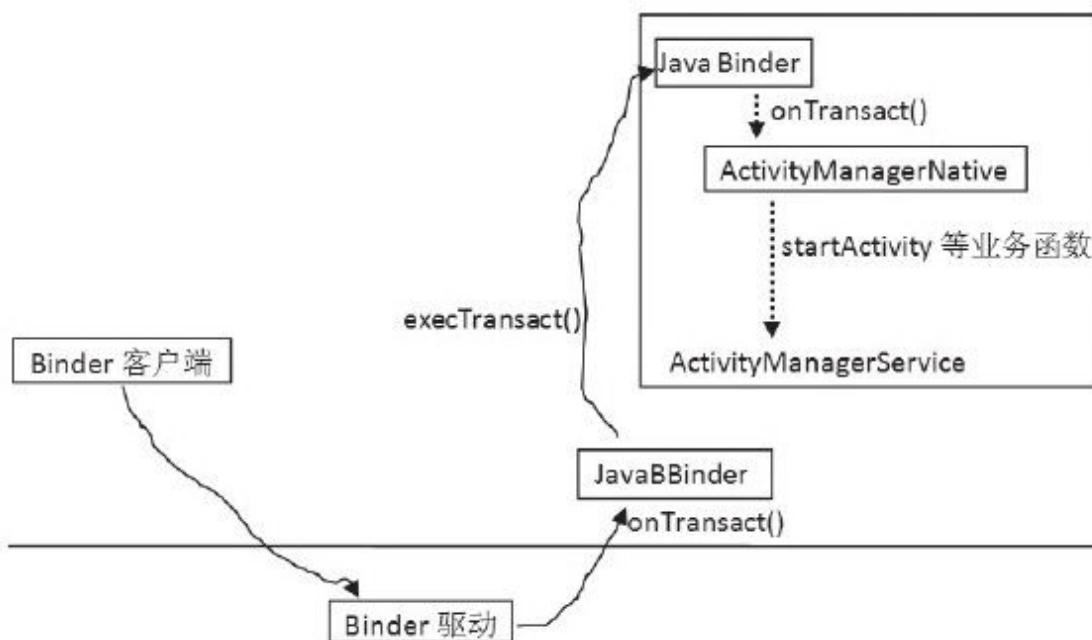


图 2-3 AMS响应请求的流程

图2-3中，右上角的大方框表示AMS这个对象，其间的虚线箭头表示调用子类重载的函数。

2.2.4 Java层Binder架构总结

图2-4展示了Java层的Binder架构。由图2-4可知：

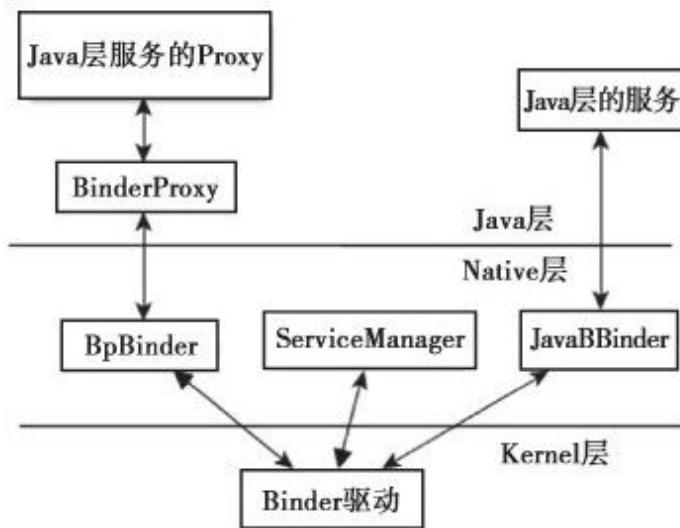


图 2-4 Java层Binder架构

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。凡是从Java层发出的请求，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。

对于代表服务端的Service来说，Java层的Binder在Native层有一个JavaBBinder对象。前面介绍过，所有Java层的Binder在Native层都对应为

JavaBBinder，而JavaBBinder仅起到中转作用，即把来自客户端的请求从Native层传递到Java层。

系统中依然只有一个Native的ServiceManager。

至此，Java层的Binder架构已介绍完毕。从前面的分析可以看出，Java层的Binder非常依赖Native层的Binder。建议想进一步了解Binder的读者阅读卷I的“第6章深入理解Binder”。

2.3 心系两界的MessageQueue

卷I第5章介绍过，MessageQueue类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世界的居民有资格向MessageQueue中添加消息以驱动Java世界的正常运转，但从Android 2.3开始，MessageQueue的核心部分下移至Native层，让Native世界的居民也能利用消息循环来处理他们所在世界的事情。因此现在的MessageQueue心系Native和Java两个世界。

2.3.1 MessageQueue的创建

现在来分析MessageQueue是如何跨界工作的，其代码如下：

[-->MessageQueue.java : MessageQueue]

```
MessageQueue () {  
    nativeInit () ;//构造函数调用nativeInit，该函数由Native层实现  
}
```

nativeInit 函数的真正实现为 android_os_MessageQueue_nativeInit，其代码如下：

[-->android_os_MessageQueue.cpp :
android_os_MessageQueue_nativeInit]

```
static void android_os_MessageQueue_nativeInit (JNIEnv*env,  
jobject obj) {  
    //NativeMessageQueue是MessageQueue在Native层的代表  
    NativeMessageQueue*nativeMessageQueue=new NativeMessageQueue  
    () ;  
    ....  
    //将这个NativeMessageQueue对象设置到Java层保存  
    android_os_MessageQueue_setNativeMessageQueue ( env,   obj,  
nativeMessageQueue) ;  
}
```

nativeInit 函数在 Native 层创建了一个与 MessageQueue 对应的 NativeMessageQueue 对象，其构造函数如下：

[-->android_os_MessageQueue.cpp :
NativeMessageQueue]

```
NativeMessageQueue : NativeMessageQueue () {  
/*  
代表消息循环的Looper也在Native层中出现了。根据消息驱动的知识，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) {  
/*
```

如为第一次进来，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，这是很常见的一种以线程为单位的单例模式

```
*/  
mLooper=new Looper (false) ;  
Looper : setForThread (mLooper) ;  
}  
}
```

Native的Looper是Native世界中参与消息循环
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。这一点以
后还将详细分析。

2.3.2 提取消息

当一切准备就绪后，Java层的消息循环处理，也就是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next函数。当消息队列为空时，next函数就会阻塞。MessageQueue同时支持Java层和Native层的事件，其next函数的实现代码如下：

[-->MessageQueue.java : next]

```
final Message next () {
    int pendingIdleHandlerCount=-1 ;
    int nextPollTimeoutMillis=0 ;
    for (;;) {
        .....
        //mPtr保存了NativeMessageQueue的指针，调用nativePollOnce进行等待
        nativePollOnce (mPtr, nextPollTimeoutMillis) ;
        synchronized (this) {
            final long now=SystemClock.uptimeMillis () ;
            //mMessages用来存储消息，这里从其中取一个消息进行处理final Message
            msg=mMessages ;
            if (msg !=null) {
                final long when=msg.when ;
                if (now>=when) {
                    mBlocked=false ;
                    mMessages=msg.next ;
                    msg.next=null ;
                    msg.markInUse () ;
                    return msg ;//返回一个Message给Looper进行派发和处理
                }
            }
        }
    }
}
```

```
        }else{
            nextPollTimeoutMillis= (int) Math.min (when-now,
                Integer.MAX_VALUE) ;
        }
    }else{
        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，例如做垃圾回收等。请读者务必记住  

此处的处置逻辑，以后分析Activity启动时会用到这里的知识
*/
.....
pendingIdleHandlerCount=0 ;
nextPollTimeoutMillis=0 ;
}
}
```

看到这里，可能有人会觉得这个MessageQueue很简单，不就是从以前在Java层的wait变成现在Native层的wait了吗？但是事情本质比表象要复杂得多，来思考下面的情况：

nativePollOnce返回后，next函数将从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不过是做了一次无用功罢了。

如果nativePollOnce在Native层等待，就表明Native层也可以投递Message（消息，为了适应业

界的习惯本书沿用英文，必要时才用中文，其他词同此），但是从Message类的实现代码上看，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）。那么nativePollOnce在等待什么呢？

对于上面的问题，相信有些读者心中已有了答案：nativePollOnce除了等待Java层来的Message，还在Native层做了大量的工作。

下面我们来分析Java层投递Message并触发nativePollOnce工作的正常流程。

1. 在Java层投递Message

MessageQueue的enqueueMessage函数完成将一个Message投递到MessageQueue中的工作，其代码如下：

[-->MessageQueue.java : enqueueMessage]

```
final boolean enqueueMessage (Message msg, long when) {  
    ....  
    final boolean needWake ;  
    synchronized (this) {  
        if (mQuiting) {  
            return false;  
        } else if (msg.target==null) {  
            mQuiting=true ;  
        }  
    }  
}
```

```

}

msg.when=when ;
Message p=mMessages ;
if (p==null||when==0||when<p.when) {
/*
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.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
Message prev=null ;
while (p !=null & & p.when<=when) {
prev=p ;
p=p.next ;
}
msg.next=prev.next ;
prev.next=msg ;
//因为消息队列之前还有剩余消息，所以这里不用调用nativeWakeup
needWake=false ;
}
}
if (needWake) {
//调用nativeWake，以触发nativePollOnce函数结束等待nativeWake
(mPtr) ;
}
return true ;
}

```

上面的代码比较简单，主要功能是：

将Message按执行时间排序，并加入消息队列。

根据情况调用nativeWake函数，以触发nativePollOnce函数，结束等待。

建议 虽然代码简单，但是对于那些不熟悉多线程的读者，还是要细细品味一下mBlocked值的作用的。我们常说细节体现美，代码也一样，这个小小的mBlocked正是如此。

2.nativeWake函数分析

nativeWake函数的代码如下所示：

[-->android_os_MessageQueue.cpp :
 android_os_MessageQueue_nativeWake]

```
static void android_os_MessageQueue_nativeWake (JNIEnv*env,  
 jobject obj,  
 jint ptr)  
{  
 NativeMessageQueue*nativeMessageQueue=// 取出  
 NativeMessageQueue对象  
 reinterpret_cast<NativeMessageQueue*> (ptr) ;  
 return nativeMessageQueue->wake () ;//调用它的wake函数  
 }  
 void NativeMessageQueue::wake () {  
 mLooper->wake () ;//层层调用，现在转到mLooper的wake函数  
 }
```

Native层Looper的wake函数代码如下：

[-->Looper.cpp : wake]

```
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) ;
        }
        }
    }
```

wake函数则更为简单，仅仅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。

2.3.3 nativePollOnce函数分析

nativePollOnce 的实现函数是 android_os_MessageQueue_nativePollOnce，代码如下：

[-->android_os_MessageQueue.cpp : android_os_MessageQueue_native Pollonce]

```
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) ;// 重任传递到Looper的
pollOnce函数
}
```

Looper的pollOnce函数如下：

[-->Looper.cpp : pollOnce]

```
inline int pollOnce (int timeoutMillis) {
return pollOnce (timeoutMillis, NULL, NULL, NULL) ;
```

}

上面的函数将调用另外一个有4个参数的pollOnce函数，这个函数的原型如下：

```
int pollOnce (int timeoutMillis, int*outFd, int*outEvents,  
void**outData)
```

其中：

timeoutMillis参数为超时等待时间。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。

outFd用来存储发生事件的那个文件描述符^[1]。

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。

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，pollOnce函数的返回值也具有特殊的意义，具体如下：

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。

返回值为ALOOPER_POLL_TIMEOUT表示等待超时。

返回值为ALOOPER_POLL_ERROR表示等待过程中发生错误。

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。上面这些知识是和epoll息息相关的。

提示 查看Looper的代码会发现，Looper采用了编译选项（即#if和#else）来控制是否使用epoll作为I/O复用的控制中枢。鉴于现在大多数系统都支持epoll，这里仅讨论使用epoll的情况。

1.epoll基础知识

epoll机制提供了Linux平台上最高效的I/O复用机制，因此有必要介绍一下它的基础知识。

从调用方法上看，epoll的用法和select/poll非常类似，其主要作用就是I/O复用，即在一个地方

等待多个文件句柄的I/O事件。

下面通过一个简单的例子来分析epoll的工作流程。

[--> epoll工作流程分析案例]

```
/*
```

使用epoll前，需要先通过epoll_create函数创建一个epoll句柄。

下面一行代码中的10表示该epoll句柄初次创建的时候分配能容纳10个fd相关信息的缓存。

对于2.6.8版本以后的内核，该值没有实际作用，这里可以忽略。其实这个值的主要目的是

确定分配一块多大的缓存。现在的内核都支持动态拓展这块缓存，所以该值就没有意义了

```
*/
```

```
int epollHandle=epoll_create (10) ;
```

```
/*
```

得到epoll句柄后，下一步就是通过epoll_ctl把需要监听的文件句柄加入到epoll句柄中。

除了指定文件句柄本身的fd值外，同时还需要指定在该fd上等待什么事件。epoll支持4类事件，

分别是EPOLLIN（句柄可读）、EPOLLOUT（句柄可写）、EPOLLERR（句柄错误）、EPOLLHUP（句柄中断）。

epoll定义了一个结构体struct epoll_event来表达监听句柄的诉求。

假设现在有一个监听端的socket句柄listener，要把它加入到epoll句柄中

```
/*
```

```
struct epoll_event listenEvent ;//先定义一个event
```

```
/*
```

EPOLLIN表示可读事件，EPOLLOUT表示可写事件，另外还有EPOLLERR，EPOLLHUP

系统默认会将EPOLLERR加入到事件集合中

```
*/
```

```
listenEvent.events=EPOLLIN ;//指定该句柄的可读事件
```

//epoll_event中有一个联合体称为data，用来存储上下文数据，本例的上下文数据就是句柄自己

```
listenEvent.data.fd=listenEvent ;  
/*
```

EPOLL_CTL_ADD将监听fd和监听事件加入到epoll句柄的等待队列中；

EPOLL_CTL_DEL将监听fd从epoll句柄中移除；

EPOLL_CTL_MOD修改监听fd的监听事件，例如本来只等待可读事件，现在需要同时等待

可写事件，那么修改listenEvent.events为EPOLLIN|EPOLLOUT后，再传给epoll句柄

```
*/
```

```
epoll_ctl  ( epollHandle,  EPOLL_CTL_ADD,  listener ,  &  
listenEvent) ;
```

```
/*
```

当把所有感兴趣的fd都加入到epoll句柄后，就可以开始坐等感兴趣的事情发生了。

为了接收所发生的事情，先定义一个epoll_event数组

```
*/
```

```
struct epoll_event resultEvents[10] ;
```

```
int timeout=-1 ;
```

```
while (1) {
```

```
/*
```

调用epoll_wait用于等待事件，其中timeout可以指定一个超时时间，resultEvents用于接收发生的事件，10为该数组的大小。

epoll_wait函数的返回值有如下含义：

nfds大于0表示所监听的句柄上有事件发生；

nfds等于0表示等待超时；

nfds小于0表示等待过程中发生了错误

```
*/
```

```
int  nfds=epoll_wait  ( epollHandle,  resultEvents ,  10 ,  
timeout) ;
```

```
if (nfds==-1)
```

```
{
```

//epoll_wait发生了错误

```
}
```

```
else if (nfds==0)
```

```
{  
//发生超时，期间没有发生任何事件  
}  
else  
{  
//resultEvents用于返回那些发生了事件的信息  
for (int i=0 ; i<nfds ; i++)  
{  
struct epoll_event& event=resultEvents[i] ;  
if (event&EPOLLIN)  
{  
/*  
收到可读事件。到底是哪个文件句柄发生该事件呢？可通过event.data这个联合  
体取得  
之前传递给epoll的上下文数据，该上下文数据可用于判断到底是谁发生了事件  
*/  
}  
.....//其他处理  
}  
}  
}
```

epoll整体使用流程如上面代码所示，基本和select/poll类似，作为Linux平台最高效的I/O复用机制，这里有些内容供读者参考。

epoll的效率为什么会比select高？其中一个原因是调用方法。每次调用select时，需要把感兴趣的事件都复制到内核中，而epoll只在epoll_ctl进行加入操作的时候才复制一次。另外，epoll内部用于保存事件的数据结构使用的是红黑树，查找速度很快。而select采用数组保存信息，不但一次能

等待的句柄个数有限，并且查找起来速度很慢（等待的事件较多时会这样）。当然，在只等待少量文件句柄时，select和epoll效率相差不是很多，但笔者还是推荐使用epoll。

epoll等待的事件有两种触发条件，一个是水平触发（EPOLLLEVEL），另外一个是边缘触发（EPOLLET, ET为Edge Trigger之意），这两种触发条件的区别非常重要。读者可通过man epoll查阅系统提供的更为详细的epoll机制去了解。

最后，关于pipe，笔者还想提出一个小问题供读者思考讨论：为什么Android中使用pipe作为线程间通信的方式？对于pipe的写端写入的数据，读端都不感兴趣，只是为了简单的唤醒。POSIX不是也有线程间同步函数吗？为什么要用pipe呢？关于这个问题的答案，可参见笔者一篇博文“随笔之如何实现一个线程池”，笔者的博客地址是<http://blog.csdn.net/innost>。

2.pollOnce函数分析

下面分析带4个参数的pollOnce函数，代码如下：

[-->Looper.cpp : pollOnce]

```
int Looper : pollOnce ( int timeoutMillis, int*outFd,
int*outEvents,
void**outData) {
int result=0;
for (;) {//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++) ;
ALooper_callbackFunc callback=response.request.callback ;
if (!callback) {//首先处理那些没有callback的response
int ident=response.request.ident ;//ident是这个response的id
int fd=response.request.fd ;
int events=response.events ;
void*data=response.request.data ;
.....
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 (outFd !=NULL) *outFd=0 ;
if (outEvents !=NULL) *outEvents=NULL ;
if (outData !=NULL) *outData=NULL ;
return result ;
}
//调用pollInner函数。注意，它在for循环内部
result=pollInner (timeoutMillis) ;
}
}
```

初看上面的代码，可能会给人有些丈二和尚摸不着头脑的感觉。但是把pollInner函数分析完毕，大家就会明白了。pollInner函数的实现代码非常长，这里把用于调试和统计的代码去掉，结果如下：

[-->Looper.cpp : pollInner]

```
int Looper::pollInner (int timeoutMillis) {
    if (timeoutMillis !=0 & & mNextMessageUptime !=LLONG_MAX) {
        nsecs_t now=systemTime (SYSTEM_TIME_MONOTONIC) ;
        .....//根据Native Message的信息计算此次需要等待的时间
        timeoutMillis=messageTimeoutMillis ;
    }
    int result=ALOOPER_POLL_WAKE ;
    mResponses.clear () ;
    mResponseIndex=0 ;
    #ifdef LOOPER_USES_EPOLL//我们只讨论使用epoll进行I/O复用的方式
    struct epoll_event eventItems[EPOLL_MAX_EVENTS] ;
    //调用epoll_wait，等待感兴趣的事件或超时发生
    int eventCount=epoll_wait      (   mEpollFd,       eventItems,
    EPOLL_MAX_EVENTS,
    timeoutMillis) ;
    #else
    .....//使用别的方式进行I/O复用
    #endif
    //从epoll_wait返回，这时候一定发生了什么事情
    mLock.lock () ;
    if (eventCount<0) { //返回值小于零，表示发生错误
        if (errno==EINTR) {
            goto Done ;
        }
        //设置result为ALOOPER_POLL_ERROR，并跳转到Done
    }
}
```

```
result=ALOOPER_POLL_ERROR ;
goto Done ;
}
//eventCount为零，表示发生超时，因此直接跳转到Done
if (eventCount==0) {
result=ALOOPER_POLL_TIMEOUT ;
goto Done ;
}
#ifndef LOOPER_USES_EPOLL
//根据epoll的用法，此时的eventCount表示发生事件的个数
for (int i=0 ; i<eventCount ; i++) {
int fd=eventItems[i].data.fd ;
uint32_t epollEvents=eventItems[i].events ;
/*
```

之前通过pipe函数创建过两个fd，这里根据fd知道是管道读端有可读事件。

读者还记得对nativeWake函数的分析吗？在那里我们向管道写端写了一个“W”字符，这样

```
就能触发管道读端从epoll_wait函数返回了
*/
if (fd==mWakeReadPipeFd) {
if (epollEvents & EPOLLIN) {
//awoken函数直接读取并清空管道数据，读者可自行研究该函数
awoken () ;
}
.....
}else{
/*
```

mRequests和前面的mResponse相对应，它也是一个KeyedVector，其中存储了

fd和对应的Request结构体，该结构体封装了和监控文件句柄相关的一些上下文信息，

例如回调函数等。我们在后面的小节会再次介绍该结构体

```
*/
ssize_t requestIndex=mRequests.indexOfKey (fd) ;
if (requestIndex>=0) {
int events=0 ;
```

```

//将epoll返回的事件转换成上层LOOPER使用的事件
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 ;
//每处理一个Request，就相应构造一个Response
pushResponse (events, mRequests.valueAt (requestIndex) ) ;
}

.....
}

}

Done : ;
#else
.....
#endif

//除了处理Request外，还处理Native的Message，注意，Native的Message
和Java层的Message
无任何关系

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 () ;
//调用Native的handler处理Native的Message
//从这里也可看出Native Message和Java层的Message没有什么关系
handler->handleMessage (message) ;
}
mLock.lock () ;
mSendingMessage=false ;

```

```
result=ALOOPER_POLL_CALLBACK ;
}else{
mNextMessageUptime=messageEnvelope.uptime ;
break ;
}
}
mLock.unlock () ;
//处理那些带回调函数的Response
for (size_t i=0 ; i<mResponses.size () ; i++) {
const Response&response=mResponses.itemAt (i) ;
ALooper_callbackFunc callback=response.request.callback ;
if (callback) {//有了回调函数，就能知道如何处理所发生的事情了
int fd=response.request.fd ;
int events=response.events ;
void*data=response.request.data ;
//调用回调函数处理所发生的事件
int callbackResult=callback (fd, events, data) ;
if (callbackResult==0) {
//callback函数的返回值很重要，如果为0，表明不需要再次监视该文件句柄
removeFd (fd) ;
}
result=ALOOPER_POLL_CALLBACK ;
}
}
return result ;
}
```

看完代码了，是否还有点模糊？那么，回顾一下pollInner函数的几个关键点：

- 1) 首先计算一下真正需要等待的时间。
- 2) 调用epoll_wait函数等待。

3) epoll_wait函数返回，这时候可能有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：

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) 真正开始处理事件是在有Done标志的位置。

首先处理Native的Message。调用Native Handler的handleMessage处理该Message。

处理Response数组中那些带有callback的事件。

上面的处理流程还是比较清晰的，但还是有一个拦路虎，那就是mRequests，下面就来清剿这个拦路虎。

3.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。

[-->android_app_NativeActivity.cpp :
loadNativeCode_native]

```
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)  
{
```

```
....
```

```
/*
```

调用Looper的addFd函数。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

```
*/
```

```
code->looper->addFd (code->mainWorkRead, 0,  
ALOOPER_EVENT_INPUT, mainWorkCallback, code) ;
```

```
....
```

```
}
```

Looper的addFd函数的实现代码如下：

[-->Looper.cpp : addFd]

```
int Looper::addFd (int fd, int ident, int events,
ALooper_callbackFunc callback, void*data) {
if (!callback) {
//判断该Looper是否支持不带回调函数的文件句柄添加。一般不支持，因为没有
回调函数
//Looper也不知道如何处理该文件句柄上发生的事情
if (!mAllowNonCallbacks) {
return -1;
}
.....
}
#ifndef LOOPER_USES_EPOLL
int epollEvents=0 ;
//将用户的事件转换成epoll使用的值
if (events&ALOOPER_EVENT_INPUT) epollEvents|=EPOLLIN ;
if (events&ALOOPER_EVENT_OUTPUT) epollEvents|=EPOLLOUT ;
{
AutoMutex_l (mLock) ;
Request request ; //创建一个Request对象
request.fd=fd ; //保存fd
request.ident=ident ; //保存id
request.callback=callback ; //保存callback
request.data=data ; //保存用户自定义数据
struct epoll_event eventItem ;
memset (&eventItem, 0, sizeof (epoll_event) ) ;
eventItem.events=epollEvents ;
eventItem.data.fd=fd ;
//判断该Request是否已经存在， mRequests以fd作为key值
ssize_t requestIndex=mRequests.indexOfKey (fd) ;
if (requestIndex<0) {
//如果是新的文件句柄，则需要为epoll增加该fd
int epollResult=epoll_ctl (mEpollFd, EPOLL_CTL_ADD, fd , &
eventItem) ;
```

```
....  
//保存request到mRequests键值数组  
mRequests.add (fd, request) ;  
}else{  
//如果之前加过，那么就修改该监听句柄的一些信息  
int epollResult=epoll_ctl (mEpollFd, EPOLL_CTL_MOD, fd, &  
eventItem) ;  
....  
mRequests.replaceValueAt (requestIndex, request) ;  
}  
}  
}  
#else  
....  
#endif  
return 1;  
}
```

4. 处理监控请求

我们发现在pollInner函数中，当某个监控fd上发生事件后，就会把对应的Request取出来调用，相应代码如下：

```
pushResponse (events, mRequests.itemAt (i) ) ;
```

PushResponse函数实现代码如下：

[-->Looper.cpp : pushResponse]

```
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  
}
```

根据前面的知识可知，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。这表明，在处理逻辑上，Native Message的优先级高于监控fd的优先级。

下面介绍如何添加Native的Message。

5.Native的sendMessage

Android 2.2中只有Java层才可以通过sendMessage往MessageQueue中添加消息，从Android 4.0开始，Native层也支持sendMessage了[2]。sendMessage的代码如下：

[-->Looper.cpp : sendMessage]

```
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) ;  
sendMessageAtTime ( now, handler, message ) ;// 调用  
sendMessageAtTime  
}
```

```
void Looper::sendMessageAtTime (nsecs_t uptime,
const sp<MessageHandler>& handler,
const Message& message) {
size_t i=0 ;
{
AutoMutex_l (mLock) ;
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 ;
}
}
//唤醒epoll_wait，让它处理消息
if (i==0) {
wake () ;
}
}
```

[1]注意，以后文件描述符也会简称为句柄。

[2]我们这里略过了Android 2.2到Android 4.0之间几个版本中的代码变化。

2.3.4 MessageQueue总结

想不到，一个小小的MessageQueue竟然有如此多的内容。在即将出版的“卷 III”中分析Android输入系统时，我们会再次在Native层与MessageQueue碰面，这里仅是为后面的相会打下一定的基础。

现在，我们将站在一个比具体代码更高的层次来认识一下MessageQueue和它的“伙伴们”。

1.消息处理的大家族合照

MessageQueue只是消息处理大家族中的一员，该家族的成员“合照”如图2-5所示。

结合前述内容可从图2-5中得到：

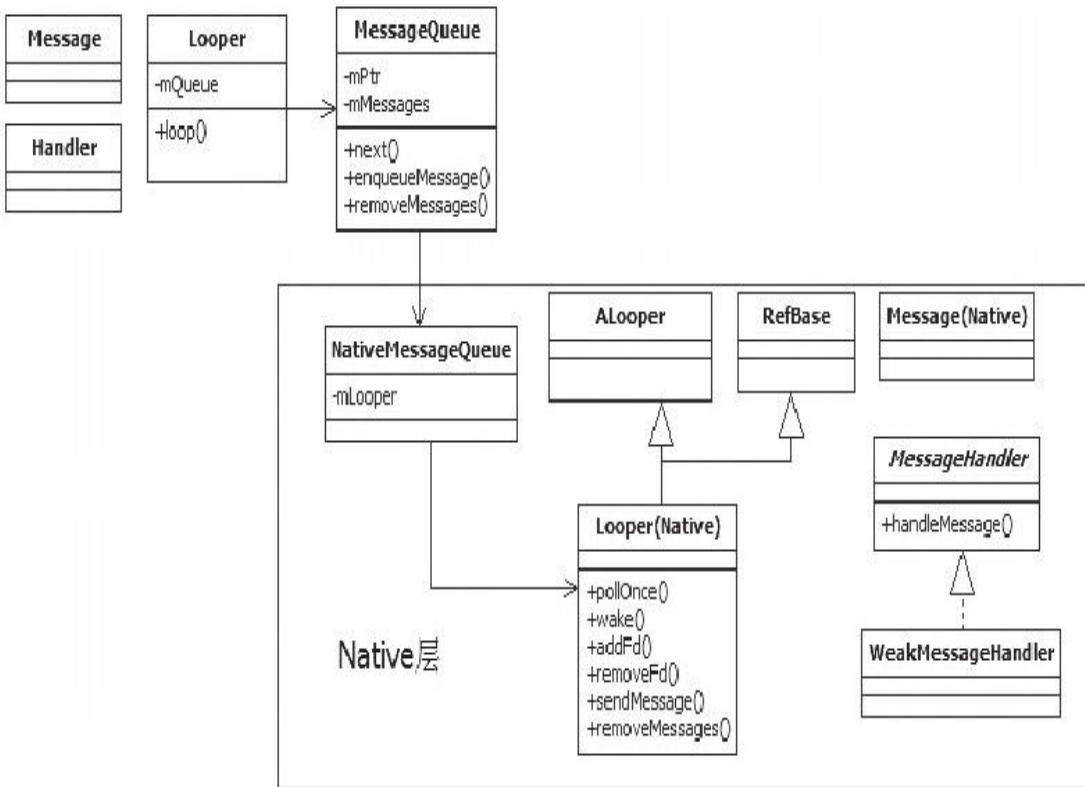


图 2-5 消息处理的家族“合照”

Java层提供了Looper类和MessageQueue类，其中Looper类提供循环处理消息的机制，MessageQueue类提供一个消息队列，以及插入、删除和提取消息的函数接口。另外，Handler也是在Java层常用的与消息处理相关的类。

MessageQueue内部通过mPtr变量保存一个Native层的NativeMessageQueue对象，mMessages保存来自Java层的Message消息。

NativeMessageQueue保存一个native的Looper对象，该Looper从ALooper派生，提供pollOnce和addFd等函数。

Java层有Message类和Handler类，而Native层对应也有Message类和Message-Handler抽象类。在编码时，一般使用的是MessageHandler的派生类WeakMessage-Handler类。

注意 在include/media/stagfright/foundation目录下也定义了一个ALooper类，它是供stagefright使用的类似Java消息循环的一套基础类。这种同名类的产生，估计是两个事先未做交流的Team中的人写的。

2.MessageQueue处理流程总结

MessageQueue核心逻辑下移到Native层后，极大地拓展了消息处理的范围，总结一下有以下几点：

MessageQueue继续支持来自Java层的Message消息，也就是早期的Message加Handler的处理方式。

MessageQueue 在 Native 层 的 代 表 NativeMessageQueue 支 持 来 自 Native 层 的

Message，是通过Native的Message和MessageHandler来处理的。

NativeMessageQueue还处理通过addFd添加的Request。在后面分析输入系统时，还会大量碰到这种方式。

从处理逻辑上看，先是处理Native的Message，然后是处理Native的Request，最后才是处理Java的Message。

对Java程序员来说，以前单纯的MessageQueue开始变得复杂。有同事经常与笔者讨论，CPU并不是很忙，为什么sendMessage的消息很久后才执行？是的，对于只了解MessageQueue Java层的工作人员，这个问题是没办法回答的。因为MessageQueue在Native层的“兄弟”NativeMessageQueue可能正在处理一个Native层的Message，而Java的调用堆栈信息又不能打印Native层的活动。所以，笔者担心，如果Java程序员只沉迷于Java语言的话，可能不能真正理解MessageQueue了。

2.4 本章小结

本章先对Java层的Binder架构做了一次较为深入的分析。Java层的Binder架构和Native层Binder架构类似，但是Java层Binder架构在通信上还是依赖Native层的Binder架构。建议想进一步了解Native层Binder架构工作原理的读者，阅读卷I“第6章 深入理解 Binder”。另外，本章还对MessageQueue进行了较为深入的分析。Android 2.2中那个功能简单的MessageQueue现在变得复杂了，原因是该类的核心逻辑下移到Native层，导致现在的MessageQueue除了支持Java层的Message派发外，还新增了支持Native层Message派发以及处理来自所监控的文件句柄的事件。另外，卷I“第5章 深入理解常见类”对Android 2.2中的MessageQueue和Looper有详细介绍，读者不妨下载电子版阅读一下。

第3章 深入理解SystemServer

本章主要内容：

分析SystemServer。

分析EntropyService、DropBoxManagerService、DiskStatsService。

分析DeviceStorageMonitorService、SamplingProfilerService以及ClipboardService。

本章所涉及的源代码文件名及位置：

SystemServer.java
(frameworks/base/services/java/com/android/server/
SystemServer.java)

com_android_server_SystemServer.cpp
(frameworks/base/services/jni/com_android_server_
SystemServer.cpp)

System_init.cpp
(frameworks/base/cmds/system_server/library/Syste
m_init.cpp)

EntropyService.java
(frameworks/base/services/java/com/android/server/

EntropyService.java)

DropBoxManagerService. java
(frameworks/base/services/java/com/android/server/
DropBox-ManagerService.java)

ActivityManagerService. java
(frameworks/base/services/java/com/android/server/
am/ActivityManagerService.java)

DiskStatsService. java
(frameworks/base/services/java/com/android/server/
DiskStatsService.java)

dumpsys. cpp
(frameworks/base/cmds/dumpsys/dumpsys.cpp)

DeviceStorageMonitorService. java
(frameworks/base/services/java/com/android/server/
Device-StorageMonitorService.java)

SamplingProfilerService. java
(frameworks/base/services/java/com/android/server/
Sampling-ProfilerService.java)

SamplingProfilerIntegration. java
(frameworks/base/core/java/com/android/internal/os/
/Sampling-ProfilerIntegration.java)

SamplingProfiler. java
(libcore/dalvik/src/main/java/dalvik/system/profiler/
SamplingProfiler.java)

ClipboardService. java
(frameworks/base/services/java/com/android/server/
ClipboardService.java)

ClipboardManagerService. java
(android.content.ClipboardManager)
(frameworks/base/core/java/android/content/Clipboa
rd-Manager.java)

ClipboardManagerService. java
(android.text.ClipboardManager)
(frameworks/base/core/java/android/text/Clipboard-
Manager.java)

ClipData. java
(frameworks/base/core/java/android/content/ClipDat
a.java)

3.1 概述

SystemServer是什么？它是Android Java的两大支柱之一。另外一个支柱是专门负责孵化Java进程的Zygote。这两大支柱倒了任何一个，都会导致Android Java的崩溃（所有由Zygote孵化的Java进程都会被销毁，而SystemServer就是由Zygote孵化而来）。若Android Java真的崩溃了，则Linux系统中的进程init会重新启动“两大支柱”以重建Android Java^[1]。

SystemServer和系统服务有着重要关系。Android系统中几乎所有的核心服务都在这个进程中，如ActivityManagerService、PowerManagerService和WindowManagerService等。那么，作为这些服务的大本营，SystemServer会是什么样的呢？

[1]关于Zygote及它和Systemserver的关系，建议读者阅读卷I第4章“深入理解Zygote”。

3.2 SystemServer分析

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。

注意 system_server进程在DDMS中看到的进程名为system_process。

3.2.1 main函数分析

SystemServer核心逻辑的入口是main函数，其代码如下：

[-->SystemServer.java : main]

```
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) ;  
    }  
    //判断性能统计功能是否开启  
    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
}
```

main函数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 init1 函数。该函数在 libandroid_servers.so 库中实现，其代码如下：

[-->com_android_server_SystemServer.cpp :
system_init]

```
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 system_init 函数又在另外一个库 libsystem_server.so 中实现，代码如下：

[-->System_init.cpp : system_init]

```
extern"C"status_t system_init ()  
{  
LOGI ("Entered system_init () " ) ;  
//初始化Binder系统  
sp<ProcessState>proc (ProcessState : self () ) ;  
//获取ServiceManager的客户端对象BpServiceManager  
sp<IServiceManager>sm=defaultServiceManager () ;  
//GrimReaper俗称死神  
sp<GrimReaper>grim=new GrimReaper () ;  
/*
```

下面这行代码的作用就是注册grim对象为ServiceManager死亡信息的接收者。
一旦SM死亡，

Binder系统就会发送讣告信息，这样grim对象的binderDied函数就会被调用。
该函数内部

将杀死 (kill) 自己 (即SystemServer) 。

笔者觉得，对于这种因挚爱离世而自杀的物体，叫死神不太合适

*/

```
sm->asBinder () ->linkToDeath (grim, grim.get () , 0) ;  
char propBuf[PROPERTY_VALUE_MAX] ;  
//判断SystemServer是否需要启动SurfaceFlinger服务，该值由init.rc  
//脚本设置，默认为零，即不启动SF服务
```

```
property_get ( "system_init.startsurfaceflinger" ,  
propBuf, "1") ;  
/*
```

从Android4.0开始，和显示相关的核心服务surfaceflinger可独立到另外一
个进程中。

笔者认为，这可能和目前SystemServer的负担过重有关。另外，随着智能终端
上HDMI的普及，

未来和显示相关的工作将会越来越繁重。将SF放在单独进程中，不仅可加强集中
管理，也可充分

利用未来智能终端上多核CPU的资源

*/

```
if (strcmp (propBuf, "1") ==0) {
```

```
SurfaceFlinger : instantiate () ;  
}  
//判断SystemServer是否启动传感器服务， 默认将启动传感器服务  
property_get ( "system_init.startsensorservice" ,  
propBuf, "1") ;  
if (strcmp (propBuf, "1") ==0) {  
//和SF相同， 传感器服务也支持在独立进程中实现  
SensorService : instantiate () ;  
}  
//获得AndroidRuntime对象  
AndroidRuntime*runtime=AndroidRuntime : getRuntime () ;  
JNIEnv*env=runtime->getJNIEnv () ;  
.....//查找Java层的SystemServer类， 获取init2函数的methodID  
jclass clazz=env->FindClass  
("com/android/server/SystemServer") ;  
.....  
jmethodID methodId=env->GetStaticMethodID  
(clazz, "init2", " () V") ;  
.....//通过JNI调用Java层的init2函数  
env->CallStaticVoidMethod (clazz, methodId) ;  
//主线程加入Binder线程池  
ProcessState : self () ->startThreadPool () ;  
IPCThreadState : self () ->joinThreadPool () ;  
return NO_ERROR ;  
}
```

SystemServer的main函数先通过init1函数， 从Java层穿越到Native层， 做了一些初始化工作后， 又通过JNI从Native层穿越到Java层去调用init2函数。

init2函数返回后， 最终又回归到Native层。

是不是感觉init1和init2这两个函数的命名和我们初学编程时自定义的函数名非常像呢？其实代码中有一段“扭捏”的注释，解释了编写这种“初级”代码的原因，也就是在对AndroidRuntime初始化前必须先初始化一些核心服务。

3.2.2 Service群英会

init1函数较简单，其实重点内容都在init2函数中。init2函数的实现代码如下：

[-->SystemServer.java : init2]

```
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  
}
```

上面的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。下面我们先来看Service的集体亮相，如图3-1所示。

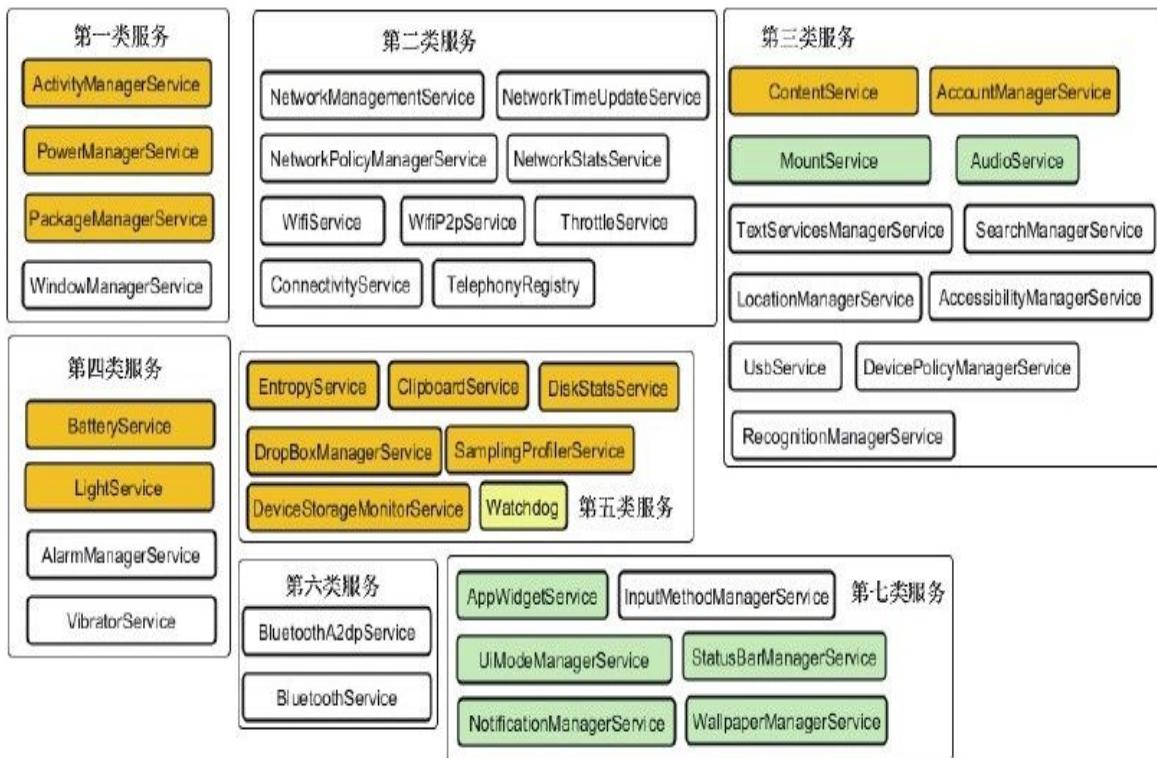


图 3-1 Service群英会

图 3-1 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，后面遇到时再做介绍。图 3-1 中的 7 大类服务主要包括：

位于第一大类的是 Android 的核心服务，如 ActivityManagerService、WindowManager-Service 等。

位于第二大类的是和通信相关的服务，如 Wifi 相关服务、Telephone 相关服务。

位于第三大类的是和系统功能相关的服务，如AudioService、MountService、Usb-Service等。

位于第四大类的是BatteryService、VibratorService等服务。

位于第五大类的是EntropyService、DiskStatsService、Watchdog等相对独立的服务。

位于第六大类的是蓝牙服务。

位于第七大类的是和UI紧密相关的服务，如状态栏服务、通知管理服务等。

注意 以上服务的分类并非官方标准，仅是笔者个人之见。

本章将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：

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Android 4.0新增的，功能非常简单。

Watchdog，即看门狗，是Android的“老员工”了。我们在卷I第4章“深入理解Zygote”中曾分析过它。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。关于Watch dog的内容，就留给读者自己分析了。

后面，将逐次分析这第五类服务的其他几项服务。

3.3 EntropyService分析

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了。

SystemServer中添加该服务的代码如下：

```
ServiceManager.addService ("entropy", new EntropyService () );
```

上边代码非常简单，从中可直接分析EntropyService的构造函数：

[-->EntropyService.java : EntropyService构造函数]

```
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 () ; //①
```

```
addDeviceSpecificEntropy () ;//②  
writeEntropy () ;//③  
scheduleEntropyWriter () ;//④  
}
```

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，这里只介绍它们的作用。感兴趣的读者可自行分析其代码。

loadInitialEntropy 函数：将entropy.dat文件的内容写到urandom设备，这样可增加系统的随机性。根据代码中的注释，系统中有一个entropy pool。在系统刚启动时，该pool中的内容为空，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。

addDeviceSpecificEntropy 函数：将一些和设备相关的信息写入urandom设备。这些信息如下：

```
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 (new Object () .hashCode () ) ;  
out.println (System.currentTimeMillis () ) ;  
out.println (System.nanoTime () ) ;
```

该函数的注释表明，即使向urandom的entropy pool中写入固定信息，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），该系统越不稳定。

writeEntropy函数：读取urandom设备的内容到entropy.dat文件。

scheduleEntropyWriter函数：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。

通过上面的分析可知，entropy.dat文件保存了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。

注意 EntropyService本身的代码很简单，但是为了尽量保证随机数的随机性，Android还是下了一番苦功的。

3.4 DropBoxManagerService分析

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。

下面来分析这项服务。其中向SystemServer添加DBMS的代码如下：

```
ServiceManager.addService (Context.DROPBOX_SERVICE, // 服务名为“dropbox”
    new DropBoxManagerService (context,
        new File ("/data/system/dropbox") ) );
```

3.4.1 DBMS构造函数分析

DBMS构造函数如下：

[-->DropBoxManagerService. java :
DropBoxManagerService构造函数]

```
public DropBoxManagerService (final Context context, File
path) {
    mDropBoxDir=path ; //path指定dropbox目录为 /data/system/dropbox
    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.Secure.CONTENT_URI, true,  
new ContentObserver(new Handler() ) {  
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) ;  
}  
}) ;  
}
```

根据上面代码可知：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。该函数在以下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。

这个函数内容较简单，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。读者可自行分析这个函数。

3.4.2 dropbox日志文件的添加

要想理清一个Service，最好从它提供的服务开始进行分析。根据前面对DBMS的介绍可知，它提供了记录系统运行时日志信息的功能，所以这里先从dropbox日志文件的生成时说起。

当某个应用程序因为发生异常而崩溃（crash）时，ActivityManagerService（AMS）的handleApplicationCrash函数被调用，其代码如下：

[-->ActivityManagerService.java :
handleApplicationCrash]

```
public void handleApplicationCrash (IBinder app,
ApplicationErrorReport.CrashInfo crashInfo) {
ProcessRecord r=findAppProcess (app, "Crash") ;
.....
//调用addErrorToDropBox函数，第一个参数是一个字符串，为"crash"
addErrorToDropBox ("crash", r, null, null, null, null, null,
crashInfo) ;
.....
}
```

下面来看addErrorToDropBox函数，具体如下：

[-->ActivityManagerService.java addErrorToDropBox]

:

```
public void addErrorToDropBox (String eventType,
ProcessRecord process, ActivityRecord activity,
ActivityRecord parent, String subject,
final String report, final File logFile,
final ApplicationErrorReport.CrashInfo crashInfo) {
/*
dropbox日志文件的命名有一定的规则，其前缀都是一个特定的tag（标签），  

tag由两部分组成，合起来是“进程类型_事件类型”。
下边代码中的processClass函数返回该进程的类型，包括“system_server”、“system_app”  

和“data_app”3种。eventType用于指定事件类型，目前也有3种类型：“crash”、“wtf”  

(what a terrible failure) 和“anr”
*/
final String dropboxTag=processClass (process )
+"_"+eventType ;
//获取DBMS Bn端的对象DropBoxManager
final DropBoxManager dbox= (DropBoxManager)
mContext.getSystemService (Context.DROPBOX_SERVICE) ;
/*
对于DBMS，不仅通过tag标识文件名，还可以根据配置的情况，允许或禁止特定
tag日志
文件的记录。isTagEnable将判断DBMS是否禁止该标签，如果该tag已被禁止，则不允许记
录日志文件
*/
if (dbox==null|| !dbox.isTagEnabled (dropboxTag) ) return ;
//创建一个StringBuilder，用于保存日志信息
final StringBuilder sb=new StringBuilder (1024) ;
appendDropBoxProcessHeaders (process, sb) ;
.....//将信息保存到字符串sb中
//单独启动一个线程用于向DBMS添加信息
```

```
Thread worker=new Thread ("Error dump : "+dropboxTag) {
@Override
public void run () {
if (report !=null) {
sb.append (report) ;
}
if (logFile !=null) {
try{//如果有log文件，那么把log文件内容读到sb中
sb.append (FileUtils.readTextFile (logFile,
128*1024, "\n\n[[TRUNCATED]]") ) ;
}.....
}
//读取crashInfo信息，一般记录的是调用堆栈信息
if (crashInfo !=null&&crashInfo.stackTrace !=null) {
sb.append (crashInfo.stackTrace) ;
}
String
setting=Settings.Secure.ERROR_LOGCAT_PREFIX+dropboxTag ;
//查询Settings数据库，判断该tag类型的日志是否对所记录的信息有行数限制，例如某些tag的日志文件只准记录1000行的信息
int lines=Settings.Secure.getInt
(mContext.getContentResolver () ,
setting, 0) ;
if (lines>0) {
sb.append ("\n") ;
InputStreamReader input=null;
try{
//创建一个新进程以运行logcat，后面的参数都是logcat常用的参数
java.lang.Process logcat=new
ProcessBuilder ("/system/bin/logcat",
"-v", "time", "-b", "events", "-b", "system", "-b",
"main", "-t", String.valueOf (lines) )
.redirectErrorStream (true) .start () ;
//由于新进程的输出已经重定向，因此这里可以获取最后lines行的信息，不熟悉
悉
```

ProcessBuilder的读者可以查看SDK中关于它的用法说明

```
....  
}  
}  
//调用DBMS的addText  
dbox.addText (dropboxTag, sb.toString () ) ;  
}  
};  
if (process==null||process.pid==MY_PID) {  
worker.run () ;//如果是SystemServer进程崩溃了，则不能在别的线程执行  
}  
}else{  
worker.start () ;  
}  
}
```

由上面代码可知，addErrorToDropBox在生成日志的内容上花了不少工夫，因为这些是最重要的，最后仅调用addText函数便将内容传给DBMS的功能。

addText函数定义在DropBoxManager类中，代码如下：

[-->DropBoxManager.java : addText]

```
public void addText (String tag, String data) {  
/*  
mService和DBMS交互。DBMS对外只提供一个add函数用于日志添加，而DBM提供了3个函数，  
分别是addText、addData、addFile，以方便使用  
*/  
try{mService.add (new Entry (tag, 0, data) ) ; }.....
```

```
}
```

DBM向DBMS传递的数据被封装在一个Entry中。下面来看DBMS的add函数，其代码如下：

[-->DropBoxManagerService.java : add]

```
public void add (DropBoxManager.Entry entry) {  
    File temp=null;  
    OutputStream output=null;  
    final String tag=entry.getTag () ;//先取出这个Entry的tag  
    try{  
        int flags=entry.getFlags () ;  
        .....  
        //做一些初始化工作，包括生成dropbox目录、统计当前已有的dropbox文件信息等  
        init () ;  
        if (!isTagEnabled (tag) ) return ;//如果该tag被禁止，则不能生成日志文件  
        long max=trimToFit () ;  
        long lastTrim=System.currentTimeMillis () ;  
        //BlockSize一般是4KB  
        byte[]buffer=new byte[mBlockSize] ;  
        //从Entry中得到一个输入流。与Java I/O相关的类比较多，且用法非常灵活，  
        建议读者阅  
        读《Java编程思想》中“Java I/O系统”一章  
        InputStream input=entry.getInputStream () ;  
        .....  
        int read=0 ;  
        while (read<buffer.length) {  
            int n=input.read (buffer, read, buffer.length-read) ;  
            if (n<=0) break ;  
            read+=n ;  
        }  
    }
```

```
//先生成一个临时文件，命名方式为“drop线程id.tmp”
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) ;
//生成GZIP压缩文件
if (read==buffer.length&&
( (flags&DropBoxManager.IS_GZIPPED) ==0) ) {
output=new GZIPOutputStream (output) ;
flags=flags|DropBoxManager.IS_GZIPPED ;
}
/*
DBMS很珍惜/data分区，若所生成文件的size大于一个BlockSize，
则一定要先压缩
*/
.....//写文件，这段代码非常烦琐，其主要目的是尽量节省存储空间
/*
生成一个EntryFile对象，并保存到DBMS内部的一个数据容器中。
DBMS除了负责生成文件外，还提供查询的功能，这个功能由getNextEntry完成。
另外，刚才生成的临时文件在createEntry函数中会重命名为带标签的名字，读者可自行分析createEntry函数
*/
long time=createEntry (temp, tag, flags) ;
temp=null ;
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) ;
}
```

```
//发送DROPBOX_ENTRY_ADDED广播。系统中目前还没有程序接收该广播  
mContext.sendBroadcast (dropboxIntent,  
    android.Manifest.permission.READ_LOGS) ;  
}.....  
}
```

上面代码中略去了DBMS写文件的部分，我们从代码注释中可获悉，DBMS非常珍惜/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。如果说细节体现功力，那么这正是一个极好的例证。

一个真实设备上/data/system/dropbox目录中的内容如图3-2所示。

```
root@innost:~# adbs /data/busybox/ls -l -h /data/system/dropbox  
-rw----- 1 1000 1000 249 Dec 23 15:51 SYSTEM_BOOT@1324655460464.txt  
-rw----- 1 1000 1000 13.4k Dec 23 15:51 SYSTEM_LAST_KMSG@1324655460520.txt.gz  
-rw----- 1 1000 1000 15.5k Dec 23 10:20 SYSTEM_TOMBSTONE@1324635608313.txt.gz  
-rw----- 1 1000 1000 3.8k Dec 23 10:26 SYSTEM_TOMBSTONE@1324635996238.txt.gz  
-rw----- 1 1000 1000 15.5k Dec 23 15:51 SYSTEM_TOMBSTONE@1324655460757.txt.gz  
-rw----- 1 1000 1000 6.1k Dec 25 18:01 data_app_anr@1324836096560.txt.gz
```

图 3-2 真实设备中dropbox目录中的内容

图 3-2 中 最 后 一 项 data_app_anr@1324836096560.txt.gz 的 大 小 是 6.1KB，该文件解压后得到的文件大小是42KB。看来，压缩确实节省了不少存储空间。

另外，我们从图3-2中还发现了其他不同的 tag ，如 SYSTEM_BOOT 、

SYSTEM_TOMBSTONE 等，这些都是由 BootReceiver 在收到 BOOT_COMPLETE 广播后收集相关信息并传递给 DBMS 而生成的日志文件。

3.4.3 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，系统就不能正常启动了。

这里总结一下和DBMS相关的配置项，具体情况如下（注意，右边双引号的内容是该配置项在数据库中的名字。这些和系统相关的配置项都在Settings数据库的Secure表内）：

```
//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，默认是三天。大于三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”

感兴趣的读者可以利用adb shell进入 /data/data/com.android.providers.settings/databases/目录，然后利用sqlite3命令操作settings.db，其中有一个Secure表，可以通过该表了解相关内容。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。

3.5 DiskStatsService和DeviceStorageMonitorService分析

DiskStatsService、DeviceStorageMonitorService与系统内部存储管理、监控有关。

3.5.1 DiskStatsService分析

DiskStatsService代码非常简单，不过也有一个很有意思的地方，例如：

[-->DiskStatsService.java]

```
public class DiskStatsService extends Binder
```

DiskStatsService从Binder派生，却没有实现任何接口，也就是说，DiskStatsService没有任何业务函数可供调用。为什么系统会存在这样的服务呢？

为了解释这个问题，有必要先介绍系统中一个很重要的命令—dumpsys。正如其名，这个命令

用于打印系统中指定服务的信息，其实现代码如下：

[-->dumpsys.cpp : main]

```
int main (int argc, char*const argv[])
{
    //先获取与ServiceManager进程通信的BpServiceManager对象
    sp<IServiceManager>sm=defaultServiceManager () ;
    fflush (stdout) ;
    Vector<String16>services ;
    Vector<String16>args ;
    if (argc==1) {//如果输入参数个数为1，则先查询在SM中注册的所有
        Service
        services=sm->listServices () ;
        //将Service排序
        services.sort (sort_func) ;
        args.add (String16 ("-a")) ;
    }else{
        //指定查询某个Service
        services.add (String16 (argv[1])) ;
        //保存剩余参数，以后可以传给Service的dump函数
        for (int i=2 ; i<argc ; i++) {
            args.add (String16 (argv[i])) ;
        }
    }
    const size_t N=services.size () ;
    .....
    for (size_t i=0 ; i<N ; i++) {
        sp<IBinder>service=sm->checkService (services[i]) ;
        .....
        //通过Binder调用该Service的dump函数，将args也传给dump函数
        int err=service->dump (STDOUT_FILENO, args) ;
        .....
    }
}
```

```
    return 0;  
}
```

从上面代码可知，dumpsys通过Binder调用某个Service的dump函数。那么“dumpsys diskstats”的输出会是什么呢？马上来试试，结果如图3-3所示。

```
# dumps sys diskstats  
Latency: 4ms [512B Data Write]  
Data-Free: 212832K / 430080K total = 49% free  
Cache-Free: 101624K / 103936K total = 97% free  
System-Free: 210064K / 409600K total = 51% free
```

图 3-3 dumps sys diskstats的结果图示

图3-3说明了执行“dumpsys diskstats”打印了系统中内部存储设备的使用情况。dumpsys是工作中常用的命令，建议读者掌握它的用法。

再来看DiskStatsService的dump函数，其实现代码如下：

[-->DiskStatsService.java : dump]

```
protected void dump ( FileDescriptor fd, PrintWriter pw,  
String[]args) {  
    byte[]junk=new byte[512] ;  
    for (int i=0 ; i<junk.length ; i++) junk[i]= (byte) i ;  
    //输出/data/system/perftest.tmp文件信息，输出后即删除该文件  
    //目前还不清楚这个文件由谁生成。从名字上看应该和性能测试有关  
    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) ;
}.....
long after=SystemClock.uptimeMillis () ;
if (tmp.exists () ) tmp.delete () ;
if (error !=null) {
.....
}else{
pw.print ("Latency : ") ;
pw.print (after-before) ;
pw.println ("ms[512B Data Write]") ;
}
//打印内部存储设备各个分区的信息
reportFreeSpace (Environment.getDataDirectory () , "Data",
pw) ;
reportFreeSpace (Environment.getDownloadCacheDirectory
() , "Cache", pw) ;
reportFreeSpace (new File ("/system") , "System", pw) ;
//有些厂商还会将/proc/yaffs信息打印出来
}
```

从前述代码中可发现，DiskStatsService没有实现任何业务接口，似乎只是为了调试而存在。所以笔者认为，DiskStatsService的功能完全可以被整合到后面即将介绍的DeviceStorageManagerService类中。总之，本节最重要的就是dumpsys这个命令了，读者一定要掌握它的用法。

3.5.2 DeviceStorageManagerService分析

DeviceStorageManagerService (DSMS) 是用来监测系统内部存储空间的状态的，添加该服务的代码如下：

```
//DSMS的服务名为devicestoragemonitor  
ServiceManager.addService  
(DeviceStorageMonitorService.SERVICE,  
    new DeviceStorageMonitorService (context) ) ;  
DSMS的构造函数的代码如下：
```

[-->DeviceStorageManagerService.java :
DeviceStorageMonitorService]

```
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%的时候，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checkMemory函数，其实现代码如下：

[-->DeviceStorageManagerService.java :
checkMemory]

```
private final void checkMemory (boolean checkCache) {  
    if (mClearingCache) {  
        .....//如果正在清理空间，则不作处理  
    }else{  
        restatDataDir () ;//重新计算3个分区的剩余空间大小  
        //如果剩余空间低于mMemLowThreshold，那么先清理一次空间  
        clearCache () ;  
        //如果空间仍不足，则发送广播，并在状态栏上设置一个警通知  
        sendNotification () ;  
        .....  
        //如果空间已满，则调用下面这个函数，以发送一次存储已满的广播  
        sendFullNotification () ;  
    }.....  
    //DEFAULT_CHECK_INTERVAL为1分钟，即每1分钟会触发一次检查  
    postCheckMemoryMsg (true, DEFAULT_CHECK_INTERVAL) ;  
}
```

当空间不足时，DSMS会先尝试clearCache函数，该函数内部会与PackageManager-Service（以下简称PKMS）交互，其代码如下：

[-->DeviceStorageManagerService.java :
clearCache]

```
private final void clearCache () {  
    if (mClearCacheObserver==null) {  
        //创建一个CachePackageDataObserver对象，当PKMS清理完空间时会回调该  
        对象的  
        //onRemoveCompleted函数
```

```
mClearCacheObserver=new CachePackageDataObserver () ;  
}  
mClearingCache=true ; //设置mClearingCache的值为true， 表示我们正在清理空间  
try{  
    //调用PKMS的freeStorageAndNotify函数以清理空间， 这个函数在分析PKMS时再介绍  
    PackageManager.Stub.asInterface (  
        ServiceManager.getService ("package") ) .  
        freeStorageAndNotify ( mMemLowThreshold,  
        mClearCacheObserver) ;  
}.....  
}
```

CachePackageDataObserver是DSMS定义的内部类，其onRemoveCompleted函数很简单，就是重新发送消息，让DSMS再检测一次存储空间。

DeviceStorageManagerService的功能单一，没有重载dump函数。而DiskStats-Service唯一有用的就是dump功能了。不知Google的工程师为什么没有把DeviceStorage-ManagerService和DiskStatsService的功能整合到一起。

3.6 SamplingProfilerService分析

添加 SamplingProfilerService 服务的代码如下：

```
ServiceManager.addService ("samplingprofiler", //服务名  
new SamplingProfilerService (context) ) ;
```

3.6.1 SamplingProfilerService构造函数分析

下面分析 SamplingProfilerService 的构造函数，其实现代码如下：

[-->SamplingProfilerService.java :
SamplingProfilerService]

```
public SamplingProfilerService (Context context) {  
//注册一个ContentObserver，用于监测Settings数据库的变化  
registerSettingObserver (context) ;  
startWorking (context) ; //①startWorking函数，见下文的分析  
}
```

先来分析上边的关键点—startWorking函数，其实现代码如下：

[-->SamplingProfilerService.java
startWorking]

:

```
private void startWorking (Context context) {  
    final DropBoxManager dropbox=//得到DropBoxManager对象  
    (DropBoxManager)  
    context.getSystemService (Context.DROPBOX_SERVICE) ;  
    //枚举/data/snapshots目录下的文件  
    File[]snapshotFiles=new File (SNAPSHOT_DIR) .listFiles () ;  
    for ( int i=0 ; snapshotFiles ! =null & & i<  
    snapshotFiles.length ;  
    i++) {  
        //将这些文件的内容转移到dropbox中，然后删除这个文件  
        handleSnapshotFile (snapshotFiles[i], dropbox) ;  
    }  
    //创建一个FileObserver对象监控shots目录，如果目录中来了新的文件，那  
   么把它们转移到  
    dropbox中  
    snapshotObserver=new FileObserver ( SNAPSHOT_DIR,  
    FileObserver.ATTRIB) {  
        @Override  
        public void onEvent (int event, String path) {  
            handleSnapshotFile ( new File ( SNAPSHOT_DIR, path ) ,  
            dropbox) ;  
        }  
    } ;  
    //启动文件夹监控，采用了Linux平台的inotify机制，感兴趣的读者可以研究  
    一下inotify  
    snapshotObserver.startWatching () ;  
}
```

看完上边的代码，不知读者是否感到有些诧异。对此，开始时笔者有两个疑惑：

其一，难道SamplingProfilerService的功能就是将/data/snapshots目录下的文件转移到dropbox中吗？该服务似乎与性能采样及统计没有任何关系。

其二，SamplingProfilerService本身并不提供性能统计的功能，那么性能采样与统计文件由谁生成呢？

对于第二个问题笔者后来找到了答案，答案就是SamplingProfilerIntegration类，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。

（可惜这个类并不是由SDK输出的，要使用该类，就只能利用源码进行编译。）至于就第一个问题，读者如有什么想法，不妨与笔者讨论一下。

3.6.2 SamplingProfilerIntegration分析

下面来看如何使用SamplingProfilerIntegration进行性能统计。系统中有很多重要进程都需要对性能进行分析，比如Zygote，其相关代码如下：

[-->zygoteInit.java : main]

```
public static void main (String argv[]) {  
    try{  
        //启动性能统计  
        SamplingProfilerIntegration.start () ;  
        .....//Zygote做自己的工作  
        //结束统计并生成结果文件  
        SamplingProfilerIntegration.writeZygoteSnapshot () ;  
        .....  
    }  
}
```

上述代码中的start函数的实现代码如下：

[-->SamplingProfilerIntegration.java : start]

```
public static void start () {  
    if (!enabled) {//判断是否开启性能统计。读者想一想，enable由谁控制？  
    return ;  
    }  
    .....  
    ThreadGroup group=Thread.currentThread () .getThreadGroup  
() ;  
    SamplingProfiler.ThreadSet threadSet=
```

```
SamplingProfiler.newThreadGroupThreadSet (group) ;  
//创建一个dalvik的SamplingProfiler，我们暂时不对它进行分析  
samplingProfiler=new  
SamplingProfiler  
(samplingProfilerDepth,  
threadSet) ;  
//启动统计  
samplingProfiler.start (samplingProfilerMilliseconds) ;  
startMillis=System.currentTimeMillis () ;  
}
```

上边代码中提出了一个问题，即用于判断是否启动性能统计的enable变量由谁控制，答案在该类的static语句中，其实现代码如下：

[-->SamplingProfilerIntegration.java]

```
static{  
samplingProfilerMilliseconds//取系统属性，默认值为0，即禁止性能  
统计  
SystemProperties.getInt ("persist.sys.profiler_ms", 0) ;  
samplingProfilerDepth=  
SystemProperties.getInt ("persist.sys.profiler_depth", 4) ;  
//如果samplingProfilerMilliseconds的值大于零，则允许性能统计  
if (samplingProfilerMilliseconds>0) {  
File dir=new File (SNAPSHOT_DIR) ;  
.....//创建/data/snapshots目录，并使其可写  
if (dir.isDirectory ()) {  
snapshotWriter=Executors.newSingleThreadExecutor (  
new ThreadFactory () {  
public Thread newThread (Runnable r) {  
return new Thread (r, TAG) ;//创建用于输出统计文件的工作线程  
}  
});  
enabled=true ;
```

```
}.....  
}else{  
snapshotWriter=null;  
enabled=false;  
Log.i (TAG, "Profiling disabled." ) ;  
}  
}  
}
```

enable的控制竟然放在static语句中，这表明要使用性能统计，就必须重新启动要统计的进程。

启动性能统计后，需要输出统计文件，这项工作由writeZygoteSnapshot函数完成，其实现代码如下：

[-->SamplingProfilerIntegration.java :
writeZygoteSnapshot]

```
public static void writeZygoteSnapshot () {  
.....  
//调用writeSnapshotFile函数，注意第一个参数为zygote，用于表示进程名  
writeSnapshotFile ("zygote", null) ;  
samplingProfiler.shutdown () ;//关闭统计  
samplingProfiler=null;  
startMillis=0 ;  
}
```

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函数在该统计文件文件头部写一些特定的信息，例如版本号、编译信息等。

注意 SamplingProfilerIntegration 的核心是SamplingProfiler，这个类定义在libcore/dalvik/

src/main/java/dalvik/system/profiler/SamplingProfiler.java文件中。感兴趣的读者可以研究一下该类的用法。在实际开发中，笔者一般使用Debug类提供的方法进行性能统计。

3.7 ClipboardService分析

ClipboardService (CBS) 是Android系统中的元老级服务了，自Android 1.0起就支持剪贴功能。在Android 4.0中再遇见它时，此功能已有了长足改进。先来看和剪贴功能有关的类的家族图谱，如图3-4所示。

由图3-4可知：

在Android 4.0中，源码中的content.ClipboardManager类继承自text.ClipboardManager类。从text.ClipboardManager类的命名中也可看出，早期的剪贴功能只支持文本。ClipboardManager由剪贴板服务的客户端使用，在SDK中有相应文档说明。

新增一个ClipData类，它就像一个容器，管理存储在其中的数据信息。具体的数据信息存储在ClipData的成员变量mItems中。该变量是一个Item类型的数组，每一个Item代表一项数据。

ClipDescription类用来描述一个ClipData中的数据类型。目前，Android系统中的剪贴板支持3种类型的数据（Text、Intent，以及URL列表）。

剪贴板的服务端由ClipboardService实现。

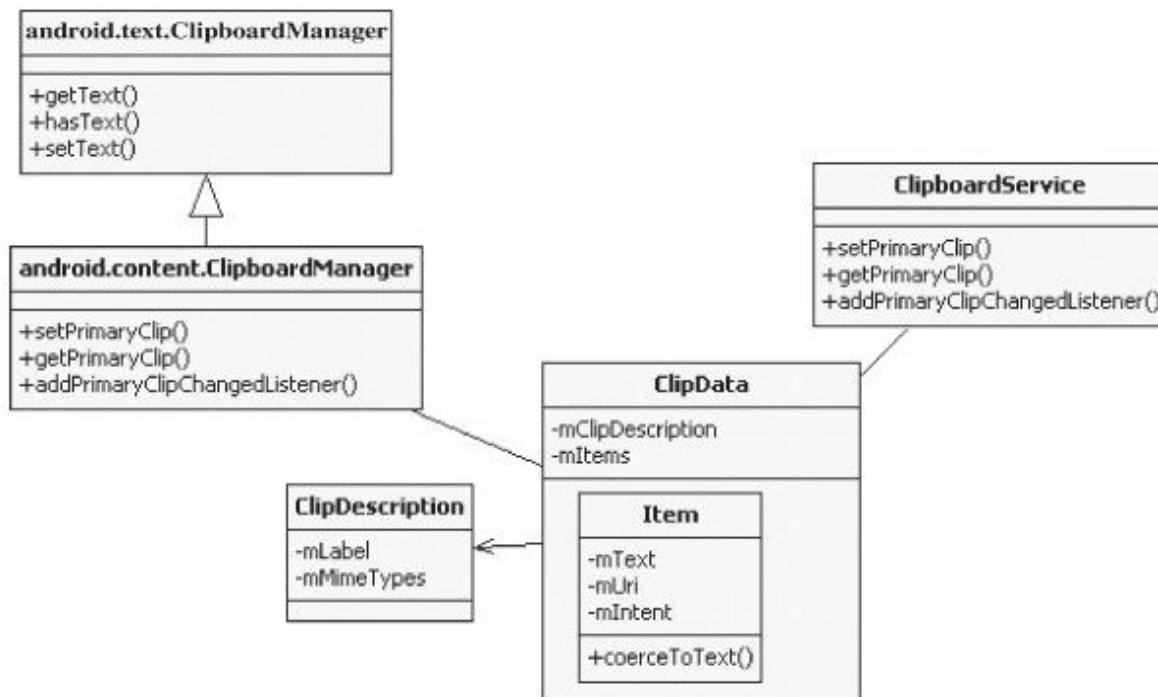


图 3-4 和剪贴功能有关的类

下边我们通过一个例子来分析CBS。该例子来源于Android SDK提供的一段示例代码（取自SDK安装目录/sample/android-14/NotePad）。

3.7.1 复制数据到剪贴板

我们截取与复制操作相关的代码，具体如下：

[-->sample]

```
//首先获取能与CBS交互的ClipboardManager对象
ClipboardManager clipboard= (ClipboardManager)
getSystemService (Context.CLIPBOARD_SERVICE) ;
//调用setPrimaryClip函数，参数是ClipData.newUri函数的返回值
clipboard.setPrimaryClip (ClipData.newUri (
getContentResolver () , "Note" , noteUri) ) ;
```

ClipData 的 newUri 是一个 static 函数，用于返回一个存储 URI 数据类型的 ClipData，代码如下。根据前文所述可知， ClipData 对象装载的就是可保存在剪贴板中的数据。

[--> ClipData.java]

```
static public ClipData newUri (ContentResolver resolver,
CharSequence label,
Uri uri) {
Item item=new Item (uri) ;//创建一个Item，将uri直接传给它的构造函数
String[]mimeTypes=null;
/*
下边代码的功能是获取这个uri代表的数据的 MIME 类型。先尝试利用 ContentResolver
从ContentProvider那查询，如果查询不到，则设置mimeTypes为
MIMETYPES_TEXT_URI_LIST，它的定义是new String["text/uri-list"]
*/
if ("content".equals (uri.getScheme ()) ) {
String realType=resolver.getType (uri) ;
//查询该uri所指向的数据的mimeTypes
mimeTypes=resolver.getStreamTypes (uri, "*/*") ;
if (mimeTypes==null) {
if (realType !=null) {
mimeTypes=new String[]{
```

```
realType, ClipDescription.MIMETYPE_TEXT_URIList} ;  
}  
}else{  
.....  
}  
if (mimeTypes==null) {  
mimeTypes=MIMETYPES_TEXT_URIList ;  
}  
//创建一个ClipData对象  
return new ClipData (label, mimeTypes, item) ;  
}  
//ClipData的构造函数  
public ClipData (CharSequence label, String[]mimeTypes, Item  
item) {  
mClipDescription=new ClipDescription (label, mimeTypes) ;  
.....  
mIcon=null ;  
mItems.add (item) ; //将item对象添加到mItems数组中  
}
```

newUri函数的主要功能在于，获得uri所指向的数据的数据类型。对于使用剪切板服务的程序来说，了解剪切板中数据的数据类型相当重要，因为这样可以判断自己能否处理这种类型的数据。

注意 uri指向数据的位置，这和PC上文件的存储位置类似，例如c：/dfp。MIME则表示该数据的数据类型。在Windows平台上是采用后缀名来表示文件类型的，前面提到的C盘下的DFP文件，后缀是.wav，表示该文件是一个WAV格式音频。

对于剪切板来说，数据源由uri指定，数据类型由MIME表示，两者缺一不可。

获得一个ClipData后，将调用setPrimaryClip函数，将数据传递到CBS。setPrimaryClip的代码如下：

[-->ClipboardManager.java : setPrimaryClip]

```
public void setPrimaryClip (ClipData clip) {  
    try{  
        //跨Binder调用，先要把参数打包。有兴趣的读者可以查看writeToParcel函数  
        getService () .setPrimaryClip (clip) ;  
    }catch (RemoteException e) {  
    }  
}
```

通过Binder发送setPrimaryClip请求后，由CBS完成实际功能，代码如下：

[-->ClipboardService.java : setPrimaryClip]

```
public void setPrimaryClip (ClipData clip) {  
    synchronized (this) {  
        .....  
        //权限检查，在3.7.3节中将单独分析  
        checkDataOwnerLocked (clip, Binder.getCallingUid () ) ;  
        /*  
        //和权限相关，后续会分析  
        clearActiveOwnersLocked () ;  
        //保存新的clipData到mPrimaryClip中  
        mPrimaryClip=clip ;
```

```
/*
mPrimaryClipListeners是一个RemoteCallbackList数组,
当CBS中的ClipData发生变化时, CBS需要向那些监控剪切板的
客户端发送通知。客户端通过addPrimaryClipChangedListener函数
注册回调
*/
final int n=mPrimaryClipListeners.beginBroadcast () ;
for (int i=0 ; i<n ; i++) {
try{
//通知客户端, 剪切板的内容发生变化
mPrimaryClipListeners.getBroadcastItem (i) .
dispatchPrimaryClipChanged () ;
}.....
}
mPrimaryClipListeners.finishBroadcast () ;
}
```

setPrimaryClip比较简单。但是由于新增支持URI和Intent这两种数据类型，因此在安全性方面还有一些需要考虑的地方。这部分内容我们放到3.7.3节中分析。

注意 RemoteCallbackList是一个比较重要的常用类，很有必要掌握它的用法。

3.7.2 从剪切板粘贴数据

我们来看一个示例，代码如下：

[-->Sample]

```
final void performPaste () {  
    //获取ClipboardManager对象  
    ClipboardManager clipboard= (ClipboardManager)  
        getSystemService (Context.CLIPBOARD_SERVICE) ;  
    //获取ContentResolver对象  
    ContentResolver cr=getContentResolver () ;  
    //从剪贴板中取出ClipData  
    ClipData clip=clipboard.getPrimaryClip () ;  
    if (clip !=null) {  
        String text=null;  
        String title=null;  
        //取剪切板ClipData中的第一项Item  
        ClipData.Item item=clip.getItemAt (0) ;  
        /*  
         * 下面这行代码取出Item中所包含的uri。看起来顺理成章，其实不然。  
         * 读者思考这样一个问题，为什么这里一定是取uri呢？原因是，在本例中，copy  
         * 方和paste方都事先了解ClipData中的数据类型。  
         * 如果paste方不了解ClipData中的数据类型，该如何处理？  
         * 一种简单的方法就是采用if/else来判断数据类型。另外还有别的方法，  
         * 下文将作分析。  
        */  
        Uri uri=item.getUri () ;  
        Cursor orig=cr.query (uri, PROJECTION, null, null, null) ;  
        .....//查询数据库并获取信息  
        orig.close () ;  
    }  
}
```

```
}

if (text==null) {
//如果paste方不了解ClipData中的数据类型，可调用coerceToText
//函数，强制得到文本类型的数据
text=item.coerceToText (this) .toString () ;//强制为文本
}

.....
}
```

下面来分析getPrimaryClip函数。

[-->ClipboardManager.java : getPrimaryClip]

```
public ClipData getPrimaryClip () {
try{
//调用CBS的getPrimaryClip，并传递自己的packageName名
return getService () .getPrimaryClip (mContext.getPackageName
() ) ;
}.....
}
```

[-->ClipboardManagerService.java :
getPrimaryClip]

```
public ClipData getPrimaryClip (String pkg) {
synchronized (this) {
//赋予该pkg相应的权限，后文再作分析
addActiveOwnerLocked (Binder.getCallingUid () , pkg) ;
return mPrimaryClip ;//返回ClipData给客户端
}
}
```

在上边的代码注释中，曾提到coerceToText函数。该函数在paste方不了解ClipData中数据类型的情况下，可以强制得到文本类型的数据。对于URI和Intent，这个功能又是如何实现的呢？来看下面的代码：

[-->ClipData.java : coerceToText]

```
public CharSequence coerceToText (Context context) {  
    //如果该Item已经有mText，则直接返回文本  
    if (mText !=null) {  
        return mText ;  
    }  
    //如果该Item中的数据是URI类型  
    if (mUri !=null) {  
        FileInputStream stream=null ;  
        try{  
            /*  
             ContentProvider需要实现openTypedAssetFileDescriptor函数，  
             以返回指定MIME（这里是text/*）类型的数据源（AssetFileDescriptor）  
            */  
            AssetFileDescriptor descr=context.getContentResolver ()  
                .openTypedAssetFileDescriptor (mUri, "text/*", null) ;  
            //创建一个输入流  
            stream=descr.createInputStream () ;  
            //创建一个InputStreamReader，读出来的数据将转换成UTF-8的文本  
            InputStreamReader reader=new InputStreamReader (stream,  
                "UTF-8") ;  
            StringBuilder builder=new StringBuilder (128) ;  
            char[]buffer=new char[8192] ;  
            int len ;  
            //从ContentProvider那读取数据，然后转换成UTF-8的字符串  
            while ( (len=reader.read (buffer)) >0) {
```

```
builder.append (buffer, 0, len) ;
}
//返回String
return builder.toString () ;
}.....
}
//如果是Intent，则调用toUri返回一个字符串
if (mIntent !=null) {
return mIntent.toUri (Intent.URI_INTENT_SCHEME) ;
}
return"" ;
}
}
```

分析上边代码可知，针对URI类型的数据，coerceToText函数还是做了不少工作的。当然，还需要提供该URI的ContentProvider实现相应的函数。

3.7.3 CBS中的权限管理

在前文的分析中，我们略去了CBS中与权限管理相关的部分，本节将集中讨论这个问题。先来回顾一下CBS中和权限管理相关的函数调用。

```
//copy方设置ClipData在CBS的setPrimaryClip函数中进行：  
checkDataOwnerLocked (clip, Binder.getCallingUid () ) ;  
clearActiveOwnersLocked () ;  
//paste方获取ClipData在CBS的getPrimaryClip函数中进行：  
addActiveOwnerLocked (Binder.getCallingUid () , pkg) ;
```

在分析这3个函数之前，先介绍一下Android系统中的URI权限管理。

1.URI权限管理介绍

Android系统的权限管理中有一类是专门针对URI的，先来看一个示例，该例来自package/providers/ContactsProvider，在它的AndroidManifest.xml中有如下声明：

[-->AndroidManifest.xml]

```
<provider android:name="ContactsProvider2"  
.....  
    android:readPermission="android.permission.READ_CONTACTS"
```

```
        android :  
        writePermission="android.permission.WRITE_CONTACTS">  
        .....  
        <grant-uri-permission android:pathPattern=".*/>  
    </provider>
```

这里声明了一个名为 ContactsProvider2 的 ContentProvider，并定义了几个权限声明，下面对其进行解释。

readPermission：要求调用query函数的客户端必须声明一个use-permission为READ_CONTACTS的权限。

writePermission：要求调用update或insert函数的客户端必须声明一个use-permission为WRITE_CONTACTS的权限。

grant-uri-permission：和授权有关。初识grant-uri-permission时，会觉得它比较难理解，下面通过举例分析帮助读者加深认识。

Contacts和ContactProvider这两个APP都是由系统提供的程序，而且二者关系紧密，所以Contacts一定会声明use_Permission为READ_CONTACTS和WRITE_CONTACT的权限。如此，Contacts就可以毫无阻碍地通过ContactsProvider来查询或更新数据库了。

假设 Contacts 新增一个功能，将 ContactsProvider 中的某条数据复制到剪切板。根据前面已介绍过的知识可以知道，Contacts 会向剪切板中复制一个 URI 类型的数据。

另外一个程序从剪切板中粘贴（paste）这条数据，由于是 URI 类型的，所以此程序会通过 ContentResolver 来查询该 uri 所指向的数据。但是这个程序却并未声明 READ_CONTACTS 的权限，所以它查询数据时必然会失败。

或许有人会问，为什么第三个程序不声明相应权限呢？原因很简单，第三个程序不知道自己该声明怎样的权限（除非这两个程序的开发者能互通信息）。本例 ContactsProvider 设置的读权限是 READ_CONTACTS，以后可能换成 READ_CONTACTS_EXTEND，第三个程序不太可能知道其中的变化。为了解决类似问题，Android 提供了一套专门针对 uri 的权限管理机制。以这套机制解决示例中权限声明问题的方法是这样的：当第三个程序从剪切板中粘贴数据时，系统会判断是否需要为这个程序授权。当然，系统不会随意授权，而是需要考虑 ContactsProvider 的情况。因为 ContactsProvider 声明了 grant-uri-permission，所以只要第三个程序所粘贴的 URI 匹配其中的 pathPattern，授权就能成功。倘若

ContactsProvider没有声明grant-uri-permission，或者uri不匹配指定的pathPattern，则授权失败。

有了前面介绍的权限管理机制，相信下面CBS中的权限管理理解起来就比较简单了。

提示 感兴趣的读者可阅读SDK安装目录下`/docs/guide/topics/security/security.html`中关于uri Permission的说明部分。

2.checkDataOwnerLocked函数分析

checkDataOwnerLocked函数的代码如下：

[-->ClipboardService.java :
checkDataDwnerLocked]

```
private final void checkDataOwnerLocked (ClipData data, int
uid) {
    //第二个参数uid为copy方进程的uid
    final int N=data.getItemCount () ;
    for (int i=0 ; i<N ; i++) {
        //为每一个item调用checkItemOwnerLocked
        checkItemOwnerLocked (data.getItemAt (i) , uid) ;
    }
}
//checkItemOwnerLocked函数分析
private final void checkItemOwnerLocked (ClipData.Item item,
int uid) {
    if (item.getUri () !=null) { //检查uri
        checkUriOwnerLocked (item.getUri () , uid) ;
    }
}
```

```
Intent intent=item.getIntent () ;  
//getData函数返回的也是一个uri，因此这里实际上检查的也是uri  
if (intent !=null&&intent.getData () !=null) {  
checkUriOwnerLocked (intent.getData () , uid) ;  
}  
}
```

权限检查就是针对uri的，因为uri所指向的数据可能是系统内部使用或私密的。例如Setting数据中的Secure表，这里的数据不能随意访问。虽然直接使用ContentResolver访问这些数据时系统会进行权限检查，但是由于目前的剪切板服务也支持URI数据类型，所以这里也需要做检查，否则恶意程序就能轻松读取私密信息。

下边来分析checkUriOwnerLocked函数，其代码如下：

[-->ClipboardService.java :
checkUriOwnerLocked]

```
private final void checkUriOwnerLocked (Uri uri, int uid) {  
....  
long ident=Binder.clearCallingIdentity () ;  
boolean allowed=false ;  
try{  
/*  
调用ActivityManagerService的checkGrantUriPermission函数，  
该函数内部将检查copy方是否能被赋予URI_READ权限。如果不允许，  
该函数会抛SecurityException异常  
*/
```

```
mAm.checkSelfPermission (uid, null, uri,  
Intent.FLAG_GRANT_READ_URI_PERMISSION) ;  
}catch (RemoteException e) {  
}finally{  
    Binder.restoreCallingIdentity (ident) ;  
}  
}
```

根据前面的知识，这里先要检查copy方是否有读取uri的权限。下面来分析paste方的权限管理。

3.clearActiveOwnersLocked函数分析

clearActiveOwnersLocked函数的代码如下：

[-->ClipboardService.java :
clearActiveOwnersLocked]

```
private final void addActiveOwnerLocked ( int uid, String  
pkg) {  
    PackageInfo pi ;  
    try{  
        /*  
         * 调用PackageManagerService的getPackageInfo函数得到相关信息  
         * 然后做一次安全检查，如果PacakgeInfo的uid信息和当前调用的uid不一致，  
         * 则抛出SecurityException。这个很好理解，因为paste方可以传递虚假的  
         * packagename，但uid是没法造假的  
        */  
        pi=mPm.getPackageInfo (pkg, 0) ;  
        if (pi.applicationInfo.uid !=uid) {  
            throw new SecurityException ("Calling uid"+uid  
+"does not own package"+pkg) ;
```

```
    }
    }.....  
    }  
    //mActivePermissionOwners用来保存已经通过安全检查的package  
    if (mPrimaryClip !=null&& !mActivePermissionOwners.contains  
(pkg) ) {  
        //针对ClipData中的每一个Item，都需要调用grantItemLocked来检查权限  
        final int N=mPrimaryClip.getItemCount () ;  
        for (int i=0 ; i<N ; i++) {  
            grantItemLocked (mPrimaryClip.getItemAt (i) , pkg) ;  
        }//保存package信息到mActivePermissionOwners  
        mActivePermissionOwners.add (pkg) ;  
    }  
}  
//grantItemLocked分析  
private final void grantItemLocked ( ClipData.Item item,  
String pkg) {  
    if (item.getUri () !=null) {  
        grantUriLocked (item.getUri () , pkg) ;  
    }//和copy方一样，这里仅检查uri的情况  
    Intent intent=item.getIntent () ;  
    if (intent !=null&&intent.getData () !=null) {  
        grantUriLocked (intent.getData () , pkg) ;  
    }  
}
```

再来看grantUriLocked的代码：

[-->ClipboardService.java : grantUriLocked]

```
private final void grantUriLocked (Uri uri, String pkg) {  
    long ident=Binder.clearCallingIdentity () ;  
    try{  
        /*  
        调用ActivityManagerService的grantUriPermissionFromOwner函数，
```

注意第二个参数传递的是CBS所在进程的uid。该函数内部也会检查权限。
该函数调用成功后，paste方就被授予了对应uri的读权限

```
/*
mAm.grantUriPermissionFromOwner (mPermissionOwner,
Process.myUid () , pkg, uri,
Intent.FLAG_GRANT_READ_URI_PERMISSION) ;
}catch (RemoteException e) {
}finally{
Binder.restoreCallingIdentity (ident) ;
}
}
```

既然有授权，那么客户端使用完毕后就需要撤销授权，这个工作是在setPrimaryClip函数的clearActiveOwnersLocked中完成的。当为剪切板设置新的ClipData时，自然需要将与旧ClipData相关的权限撤销。读者可自行分析clearActiveOwnersLocked函数。

3.8 本章小结

本章首先分析了system_server进程的启动过程，然后向读者展示了该进程中所容纳的系统核心服务。从总体上来说，这些服务可分为七大类，本章介绍了其中的第五大类服务，这是因为此类服务功能相对单一，依赖关系较为简单。这几个服务包括EntropyService、DropBoxManagerService、DiskStatsService、DeviceStorageMonitorService、SamplingProfilerService和ClipboardService。其中ClipboardService涉及uri权限管理方面的知识，相对较难，建议读者仔细阅读并深入体会。

第4章 深入理解 PackageManagerService

本章主要内容：

分析PackageManagerService。

本章所涉及的源代码文件名及位置：

SystemServer. java
(frameworks/base/services/java/com/android/server/
SystemServer.java)

IPackageManager. aidl
(frameworks/base/core/android/java/content/pm/IPa
ckageManager.aidl)

PackageManagerService. java
(frameworks/base/services/java/com/android/server/
pm/PackageManagerService.java)

Settings. java
(frameworks/base/services/java/com/android/server/
pm/Settings.java)

SystemUI 的 AndroidManifest.xml
(frameworks/base/package/SystemUI/AndroidManif

est.xml)

PackageParser.java
(frameworks/base/core/java/android/content/pm/PackageParser.java)

commandline.
(system/core/adb/commandline.c)

installd.
(frameworks/base/cmds/installd/installd.c)

commands.
(frameworks/base/cmds/installd/commands.c)

pm脚本文件 (frameworks/base/cmds/pm/pm)

Pm.
(frameworks/base/cmds/pm/src/com/android/commands/pm/Pm.java)

DefaultContainerService.java
(frameworks/base/packages/defaultcontainerservice/
src/com/android/defaultcontainerservice/DefaultContainerService.java)

UserManager.java
(frameworks/base/services/java/com/android/server/

pm/UserManager.java)

UserInfo. java
(frameworks/base/core/android/java/content/pm/Use
rInfo.java)

4.1 概述

PackageManagerService是本书分析的第一个核心服务，也是Android系统中最常用的服务之一。它负责系统中Package的管理，应用程序的安装、卸载、信息查询等。图4-1展示了PackageManagerService及客户端的类家族。

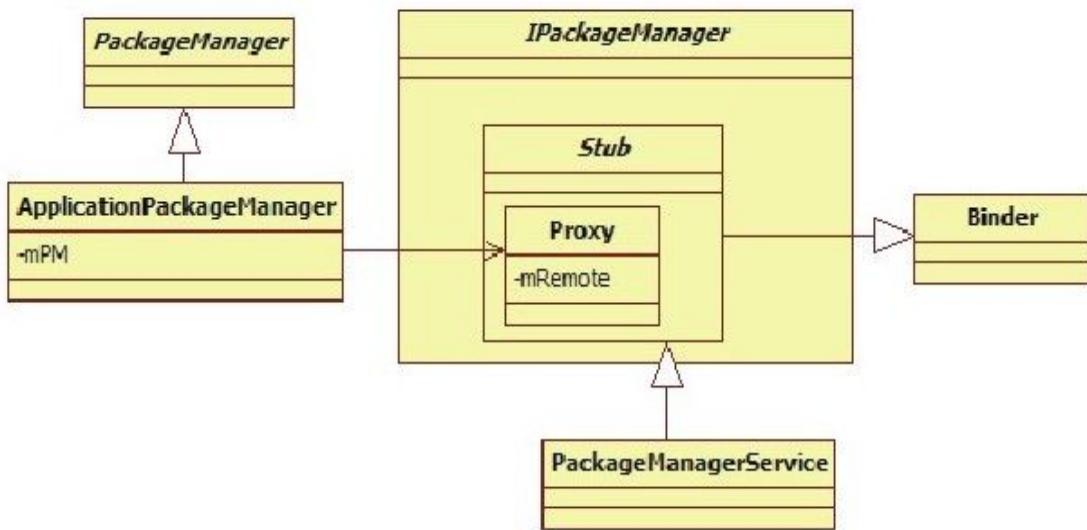


图 4-1 PackageManagerService及客户端类家族

由图4-1可知：

IPackageManager接口类中定义了服务端和客户端通信的业务函数，还定义了内部类Stub，该类从Binder派生并实现了IPackageManager接口。

PackageManagerService 继承自 IPackageManager.Stub类，由于Stub类从Binder派生，因此PackageManagerService将作为服务端参与Binder通信。

Stub类中定义了一个内部类Proxy，该类有一个IBinder类型（实际类型为BinderProxy）的成员变量mRemote，根据第2章介绍的Binder系统的知识，mRemote用于和服务端PackageManagerService通信。

IPackageManager接口类中定义了许多业务函数，但是出于安全等方面的考虑，Android对外（即SDK）提供的只是一个子集，该子集被封装在抽象类PackageManager中。客户端一般通过Context的getPackageManager函数返回一个类型为PackageManager的对象，该对象的实际类型是PackageManager 的子类 ApplicationPackageManager。这种基于接口编程的方式，虽然极大降低了模块之间的耦合性，却给代码分析带来了不小的麻烦。

ApplicationPackageManager 类继承自 PackageManager类。它并没有直接参与Binder通信，而是通过mPM成员变量指向一个IPackageManager.Stub.Proxy类型的对象。

提 示 读 者 在 源 码 中 可 能 找 不 到 IPackageManager.java文件。该文件在编译过程中是IPackage-Manager.aidl经aidl工具处理后得到的，最 终 的 文 件 位 于 Android 源 码 /out/target/common/obj/JAVA_LIBRARIES/frame work_intermediates/src/core/java/android/content/pm/ 目录中。如果读者没有整体编译过源码，也可使 用aidl工具单独处理IPackageManager.aidl。

aidl工具生成的结果文件有着相似的代码结 构。读者不妨看看下面这个笔者通过编译生成 的IPackageManager.java文件。注意，aidl工具生成 的结果文件没有格式缩进，所以看起来困难，读者 可用Eclipse中的源文件格式化命令处理它。

[-->IPackageManager.java]

```
public interface IPackageManager extends
android.os.IInterface{
    //定义内部类Stub，派生自Binder，实现IPackageManager接口
    public static abstract class Stub extends android.os.Binder
        implements android.content.pm.IPackageManager{
        private static final java.lang.String DESCRIPTOR=
            "android.content.pm.IPackageManager" ;
        public Stub () {
            this.attachInterface (this, DESCRIPTOR) ;
        }
        .....
        //定义Stub的内部类Proxy，实现IPackageManager接口
        private static class Proxy implements
```

```
android.content.pm.IPackageManager{  
    //通过mRemote变量和服务端交互  
    private android.os.IBinder mRemote ;  
    Proxy (android.os.IBinder remote) {  
        mRemote=remote ;  
    }  
    .....  
}  
.....  
}
```

接下来分析PackageManagerService，为书写方便，以后将其简称为PKMS。

4.2 初识PackageManagerService

PKMS作为系统的核心服务，由SystemServer创建，相关代码如下：

[-->SystemServer.java]

```
.....//ServerThread的run函数
/*
   Android 4.0新增的一个功能，即设备加密（encrypting the device），
该功能由
    系统属性vold.decrypt指定。这部分功能比较复杂，本书暂不讨论。
    该功能对PKMS的影响就是通过onlyCore实现的，该变量用于判断是否只扫描系
统库
    （包括APK和Jar包）
*/
String cryptState=SystemProperties.get ("vold.decrypt") ;
boolean onlyCore=false ;
//ENCRYPTING_STATE的值为trigger_restart_min_framework
if (ENCRYPTING_STATE.equals (cryptState) ) {
.....
onlyCore=true ;
} else if (ENCRYPTED_STATE.equals (cryptState) ) {
....//ENCRYPTED_STATE的值为1
onlyCore=true ;
}
//①调用PKMS的main函数，第二个参数用于判断是否为工厂测试，我们不讨论这
种情况，
//假定onlyCore的值为false
pm=PackageManagerService.main (context,
factoryTest !=SystemServer.FACTORY_TEST_OFF, onlyCore) ;
```

```
boolean firstBoot=false;
try{
    //判断本次是否为初次启动。当Zygote或SystemServer退出时，init会再次
    //启动
    //它们，所以这里的firstBoot是指开机后的第一次启动
    firstBoot=pm.isFirstBoot();
}
.....
try{
    //②做dex优化。dex是Android上针对Java字节码的一种优化技术，可提高运
    行效率
    pm.performBootDexOpt();
}
.....
try{
    pm.systemReady(); //③通知系统进入就绪状态
}
.....
}//run函数结束
```

以上代码中共有4个关键调用，分别是：

PKMS的main函数。这个函数是PKMS的核心，稍后会重点分析它。

isFirstBoot、performBootDexOpt 和 systemReady。这3个函数比较简单。学完本章后，读者可完全自行分析它们，故这里不再赘述。

首先分析PKMS的main函数，它是核心函数，此处单独用一节进行分析。

4.3 PKMS的main函数分析

PKMS的main函数代码如下：

[--> PackageManagerService.java : main]

```
public static final IPackageManager main (Context context,  
boolean factoryTest,  
boolean onlyCore) {  
    //调用PKMS的构造函数，factoryTest和onlyCore的值均为false  
    PackageManagerService m=new PackageManagerService (context,  
    factoryTest, onlyCore) ;  
    //向ServiceManager注册PKMS  
    ServiceManager.addService ("package", m) ;  
    return m;  
}
```

main函数很简单，只有短短几行代码，执行时间却较长，主要原因是PKMS在其构造函数中做了很多“重体力活”，这也是Android启动速度慢的主要原因之一。在分析该函数前，先简单介绍一下PKMS构造函数的功能。

PKMS构造函数的主要功能：扫描Android系统中几个目标文件夹中的APK，从而建立合适的数据结构以管理诸如Package信息、四大组件信息、权限信息等各种信息。抽象地看，PKMS像一

个加工厂，它解析实际的物理文件（APK文件）以生成符合自己要求的产品。例如，PKMS将解析APK包中的AndroidManifest.xml，并根据其中声明的Activity标签来创建与此对应的对象并加以保管。

PKMS的工作流程相对简单，复杂的是其中用于保存各种信息的数据结构和它们之间的关系，以及影响最终结果的策略控制（例如前面代码中的onlyCore变量，用于判断是否只扫描系统目录）。曾经阅读过PKMS的读者可能会发现，代码中大量不同的数据结构以及它们之间的关系会令人大为头疼。所以，本章除了分析PKMS的工作流程外，也将关注重要的数据结构及它们的作用。

PKMS构造函数的工作流程大体可分3个阶段：

扫描目标文件夹之前的准备工作。

扫描目标文件夹。

扫描之后的工作。

该函数涉及的知识点较多，代码段也较长，因此我们将通过分段讨论的方法，集中解决相关的重要问题。

4.3.1 构造函数分析之前期准备工作

下面开始分析构造函数第一阶段的工作，先看如下所示的代码。

[--> PackageManagerService.java : 构造函数]

```
public PackageManagerService ( Context context, boolean
factoryTest,
boolean onlyCore) {
.....
if (mSdkVersion<=0) {
/*
mSdkVersion是PKMS的成员变量，定义的时候进行赋值，其值取自系统属性
ro.build.version.sdk，即编译的SDK版本。如果没有定义，则APK
就无法知道自己运行在Android哪个版本上
*/
Slog.w (TAG, "****ro.build.version.sdk not set ! ") ;//打印一句
警告
}
mContext=context ;
mFactoryTest=factoryTest ;//假定为false，即运行在非工厂模式下
mOnlyCore=onlyCore ;//假定为false，即运行在普通模式下
//如果此系统是eng版，则扫描Package后，不对package做dex优化
mNoDexOpt="eng".equals ( SystemProperties.get
("ro.build.type") ) ;
//mMetrics用于存储与显示屏相关的一些属性，例如屏幕的宽/高尺寸，分辨率
等信息
mMetrics=new DisplayMetrics () ;
//Settings是一个非常重要的类，该类用于存储系统运行过程中的一些设置，
//下面进行详细分析
mSettings=new Settings () ;
//①addSharedUserLPw是什么？马上来分析
```

```
mSettings.addSharedUserLPw ("android.uid.system",  
    Process.SYSTEM_UID, ApplicationInfo.FLAG_SYSTEM) ;  
mSettings.addSharedUserLPw ("android.uid.phone",  
    MULTIPLE_APPLICATION_UIDS//该变量的默认值是true  
    ?RADIO_UID : FIRST_APPLICATION_UID,  
    ApplicationInfo.FLAG_SYSTEM) ;  
mSettings.addSharedUserLPw ("android.uid.log",  
    MULTIPLE_APPLICATION_UIDS  
    ?LOG_UID : FIRST_APPLICATION_UID,  
    ApplicationInfo.FLAG_SYSTEM) ;  
mSettings.addSharedUserLPw ("android.uid.nfc",  
    MULTIPLE_APPLICATION_UIDS  
    ?NFC_UID : FIRST_APPLICATION_UID,  
    ApplicationInfo.FLAG_SYSTEM) ;  
.....//第一段结束
```

刚进入构造函数，就会遇到第一个较为复杂的数据结构 Setting 及它的 addShared-User-LPw 函数。Setting 的作用是管理 Android 系统运行过程中的一些设置信息。到底是哪些信息呢？来看下面的分析。

1.初识Settings

先分析 addSharedUserLPw 函数。此处截取该函数的调用代码，如下所示：

```
mSettings.addSharedUserLPw ("android.uid.system", //字符串  
    Process.SYSTEM_UID, //系统进程使用的用户id，值为1000  
    ApplicationInfo.FLAG_SYSTEM//标志系统Package  
) ;
```

以此处的函数调用为例，我们为addSharedUserLPw传递了3个参数：第一个是字符串"android.uid.system"；第二个是SYSTEM_UID，其值为1000；第三个是FLAG_SYSTEM标志，用于标识系统Package。

在进入对addSharedUserLPw函数的分析前，先介绍一下SYSTEM_UID及相关知识。

(1) Android系统中UID/GID介绍

UID为用户ID的缩写，GID为用户组ID的缩写，这两个概念均与Linux系统中进程的权限管理有关。一般说来，每一个进程都会有一个对应的UID（即表示该进程属于哪个用户，不同用户有不同权限）。一个进程也可分属不同的用户组（每个用户组都有对应的权限）。

提示 Linux的UID/GID还可细分为几种类型，此处我们仅考虑普适意义的UID/GID。

如上所述，UID/GID和进程的权限有关。在Android平台中，系统定义的UID/GID在Process.java文件中，如下所示：

[-->Process.java]

```
//系统进程使用的UID/GID，值为1000
```

```
public static final int SYSTEM_UID=1000 ;
//Phone进程使用的UID/GID，值为1001
public static final int PHONE_UID=1001 ;
//shell进程使用的UID/GID，值为2000
public static final int SHELL_UID=2000 ;
//使用LOG的进程所在的组的UID/GID为1007
public static final int LOG_UID=1007 ;
//供WIF相关进程使用的UID/GID为1010
public static final int WIFI_UID=1010 ;
//mediaserver进程使用的UID/GID为1013
public static final int MEDIA_UID=1013 ;
//设置能读写SD卡的进程的GID为1015
public static final int SDCARD_RW_GID=1015 ;
//NFC相关的进程的UID/GID为1025
public static final int NFC_UID=1025 ;
//有权限读写内部存储的进程的GID为1023
public static final int MEDIA_RW_GID=1023 ;
//第一个应用Package的起始UID为10000
public static final int FIRST_APPLICATION_UID=10000 ;
//系统所支持的最大的应用Package的UID为99999
public static final int LAST_APPLICATION_UID=99999 ;
//和蓝牙相关的进程的GID为2000
public static final int BLUETOOTH_GID=2000 ;
```

对不同的UID/GID授予不同的权限，接下来介绍和权限设置相关的代码。

提示 读者可用adb shell登录到自己的手机，然后用busybox提供的ps命令查看进程的uid。

下面分析addSharedUserLPw函数，代码如下：

[--> Settings.java : addSharedUserLPw]

```
SharedUserSetting addSharedUserLPw ( String name, int uid,
int pkgFlags) {
/*
注意这里的参数：name为字符串 android.uid.system, uid为1000 ,
pkgFlags为
ApplicationInfo.FLAG_SYSETM (以后简写为FLAG_SYSTEM)
*/
//mSharedUsers是一个HashMap, key为字符串, 值为SharedUserSetting
对象
SharedUserSetting s=mSharedUsers.get (name) ;
if (s !=null) {
if (s.userId==uid) {
return s;
}.....
return null;
}
//创建一个新的SharedUserSetting对象, 并设置的userId为uid,
//SharedUserSetting是什么?有什么作用?
s=new SharedUserSetting (name, pkgFlags) ;
s.userId=uid;
if (addUserIdLPw (uid, s, name) ) {
mSharedUsers.put ( name, s) ; // 将 name 与 s 键值对添加到
mSharedUsers中保存
return s;
}
return null;
}
```

从以上代码可知，Settings 中有一个 mSharedUsers 成员，该成员存储的是字符串与

SharedUserSetting键值对，也就是说以字符串为key得到对应的SharedUserSetting对象。

那么SharedUserSetting是什么？它的目的是什么？来看一个例子。

(2) SharedUserSetting分析

该例子来源于SystemUI的AndroidManifest.xml，如下所示：

[-->SystemUI的AndroidManifest.xml]

```
<manifest           xmlns:android="http://schemas.android.com/apk/res/android" :  
    android="http://schemas.android.com/apk/res/android"  
    package="com.android.systemui"  
    coreApp="true"  
    android:sharedUserId="android.uid.system"  
    android:process="system">  
    ....
```

在xml中，声明了一个名为android:sharedUserId的属性，其值为"android.uid.system"。sharedUserId和UID有关，它有两个作用：

两个或多个声明了同一种sharedUserId的APK可共享彼此的数据，并且可运行在同一进程中。

更重要的是，通过声明特定的sharedUserId，该APK所在进程将被赋予指定的UID。例如，本例

中的SystemUI声明了system的uid，运行SystemUI的进程就可享有system用户所对应的权限（实际上就是将该进程的uid设置为system的uid）了。

提示 除了在AndroidManifest.xml中声明sharedUserId外，APK在编译时还必须使用对应的证书进行签名。例如本例的SystemUI，在其Android.mk中需要额外声明LOCAL_CERTIFICATE := platform，如此，才可获得指定的UID。

通过以上介绍读者能知道，如何组织一种数据结构来包括上面的内容。此处有3个关键点需注意：

XML中sharedUserId属性指定了一个字符串，它是UID的字符串描述，故对应数据结构中也应该有这样一个字符串，这样就把代码和XML中的属性联系起来了。

在Linux系统中，真正的uid是一个整数，所以该数据结构中必然有一个整型变量。

多个Package可声明同一个sharedUserId，因此该数据结构必然会保存那些声明了相同sharedUserId的Package的某些信息。

了解了上面3个关键点后，再来看Android是如何设计相应数据结构的，其中ShareUser-Setting类的关系如图4-2所示。

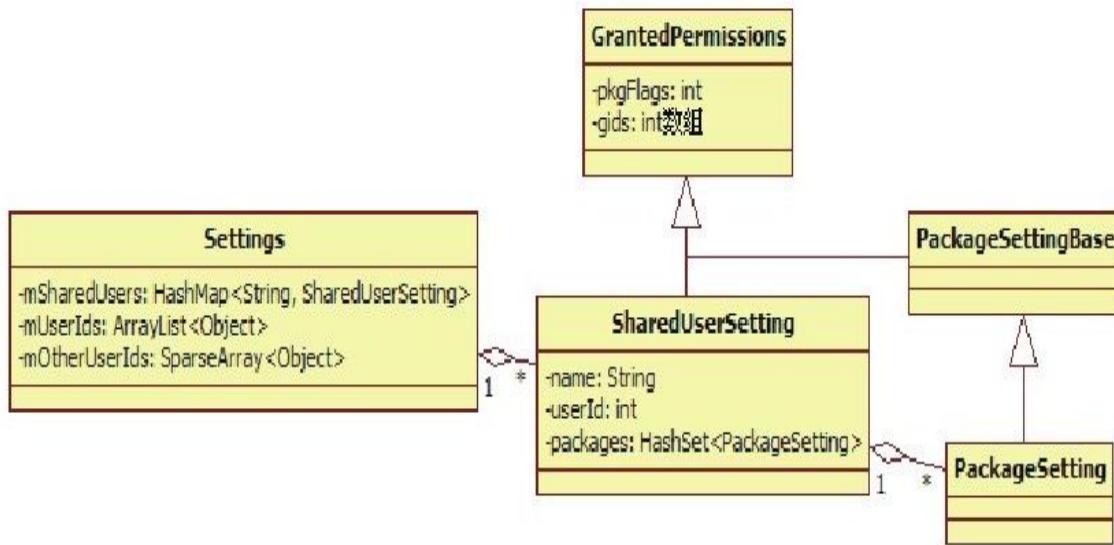


图 4-2 SharedUserSetting类的关系图

由图4-2可知：

Settings类定义了一个mSharedUsers成员，它是一个HashMap，以字符串（如`android.uid.system`）为key，对应的Value是一个SharedUserSetting对象。

SharedUserSetting派生自GrantedPermissions类，从GrantedPermissions类的命名可知，它和权限有关。SharedUserSetting定义了一个成员变量`packages`，类型为`HashSet`，用于保存声明了相同`sharedUserId`的Package的权限设置信息。

每个Package有自己的权限设置。权限的概念由 PackageSetting 类表达。该类继承自 PackagesettingBase，而 PackageSettingBase 又继承自 GrantedPermissions。

Settings 中还有两个成员，一个是 mUserIds，另一个是 mOtherUserIds，这两位成员的类型分别是 ArrayList 和 SparseArray。其目的是以 UID 为索引，得到对应的 SharedUserSettings 对象。在一般情况下，以索引获取数组元素的速度，比以 key 获取 HashMap 中元素的速度要快很多。

提示 根据以上对 mUserIds 和 mOtherUserIds 的描述，可知这是典型的以空间换时间的做法。

下边来分析 addUserLpw 函数，它的功能就是将 SharedUserSettings 对象保存到对应的数组中，代码如下：

[--> Settings.java : addUserLpw]

```
private boolean addUserLpw (int uid, Object obj, Object name) {  
    //uid不能超出限制。Android对uid进行了分类，应用APK所在进程的uid从  
    //10000开始，  
    //而系统APK所在进程的uid小于10000  
    if (uid >= PackageManagerService.FIRST_APPLICATION_UID +  
        PackageManagerService.MAX_APPLICATION_UIDS) {  
        return false;  
    }  
}
```

```
if (uid>=PackageManagerService.FIRST_APPLICATION_UID) {  
    int N=mUserIds.size () ;  
    //计算索引， 其值是uid和FIRST_APPLICATION_UID的差  
    final int index=uid-  
PackageManagerService.FIRST_APPLICATION_UID ;  
    while (index>=N) {  
        mUserIds.add (null) ;  
        N++ ;  
    }  
    .....//判断该索引位置的内容是否为空， 为空才保存  
    mUserIds.set (index, obj) ;//mUserIds保存应用Package的uid  
}else{  
    .....  
    mOtherUserIds.put ( uid, obj ) ;// 系统 Package 的 uid 由  
mOtherUserIds保存  
}  
return true ;  
}
```

至此，对Settings的分析就告一段落了。在这次“行程”中，我们重点分析了UID/GID以及SharedUserId方面的知识，并见识好几个重要的数据结构。希望读者通过SystemUI的实例能够理解这些数据结构存在的目的。

2.XML文件扫描

下面继续分析PKMS的构造函数，代码如下：

[-->PackageManagerService.java：构造函数]

```
.....//接前一段
String separateProcesses;//该值和调试有关。一般不设置该属性
SystemProperties.get ("debug.separate_processes") ;
if (separateProcesses !=null&&separateProcesses.length () >
0) {
.....
}else{
mDefParseFlags=0 ;
mSeparateProcesses=null ;
}
//创建一个Installer对象，该对象和Native进程installd交互，以后分析
installd
//时再来讨论它的作用
mInstaller=new Installer () ;
WindowManager wm;//得到一个WindowManager对象
( WindowManager ) context.getSystemService
(Context.WINDOW_SERVICE) ;
Display d=wm.getDefaultDisplay () ;
d.getMetrics (mMetrics) ;//获取当前设备的显示屏信息
synchronized (mInstallLock) {
synchronized (mPackages) {
//创建一个ThreadHandler对象，实际就是创建一个带消息循环处理的线程，该
线程
//的工作是：程序的安装和卸载等。以后分析程序安装时会和它亲密接触
mHandlerThread.start () ;
//以ThreadHandler线程的消息循环（Looper对象）为参数创建一个
PackageHandler，
//可知该Handler的handleMessage函数将运行在此线程上
mHandler=new PackageHandler (mHandlerThread.getLooper () ) ;
File dataDir=Environment.getDataDirectory () ;
//mAppDataDir指向/data/data目录
mAppDataDir=new File (dataDir, "data") ;
//mUserAppDataDir指向/data/user目录
mUserAppDataDir=new File (dataDir, "user") ;
//mDrmAppPrivateInstallDir指向/data/app-private目录
mDrmAppPrivateInstallDir=new File (dataDir, "app-private") ；
```

```
/*
创建一个UserManager对象，目前没有什么作用，但其前途将不可限量。
根据Google的设想，未来手机将支持多个User，每个User将安装自己的应用，
该功能为Android智能手机推向企业用户打下坚实基础
*/
mUserManager=new UserManager (mInstaller, mUserAppDataDir) ;
//①从文件中读权限
readPermissions () ;
//②readLPw分析
mRestoredSettings=mSettings.readLPw () ;
long startTime=SystemClock.uptimeMillis () ;
```

以上代码中创建了几个对象，此处暂可不去理会它们。另外，以上代码中还调用了两个函数，分别是readPermission和Settings的readLPw，它们有什么作用呢？下面就展开分析。

（1）readPermissions函数分析

先来分析readPermissions函数，从其函数名可猜测到它和权限有关，代码如下：

[-->PackageManagerService.java :
readPermissions]

```
void readPermissions () {
    //指向/system/etc/permission目录，该目录中存储了和设备相关的一些权限信息
    File libraryDir=new File (Environment.getRootDirectory () ,
    "etc/permissions") ;
    .....
    for (File f:libraryDir.listFiles () ) {
```

```
//先处理该目录下的非platform.xml文件
if (f.getPath () .endsWith ("etc/permissions/platform.xml") )
{
    continue ;
}
.....//调用readPermissionFromXml解析此XML文件
readPermissionsFromXml (f) ;
}
final File permFile=new File (Environment.getRootDirectory
() ,
"etc/permissions/platform.xml") ;
//解析platform.xml文件，看来该文件优先级最高
readPermissionsFromXml (permFile) ;
}
```

悬着的心终于放了下来！readPermissions函数不就是调用readPermissionFromXml函数解析/system/etc/permissions目录下的文件吗？这些文件似乎都是XML文件。该目录下都有哪些XML文件呢？这些XML文件中有些什么内容呢？来看一个实际的例子，如图4-3所示。

图4-3中列出的是笔者G7手机上/system/etc/permissions目录下的内容。在上面的代码中，虽然最后才解析到platform.xml文件，但是此处先分析该文件的内容具体如下：

```
root@android:/etc/permissions # ls
android.hardware.camera.flash-autofocus.xml
android.hardware.location.gps.xml
android.hardware.sensor.light.xml
android.hardware.sensor.proximity.xml
android.hardware.telephony.gsm.xml
android.hardware.touchscreen.multitouch.xml
android.hardware.wifi.xml
android.software.sip.voip.xml
com.android.location.provider.xml
handheld_core_hardware.xml
platform.xml
```

图 4-3/system/etc/permissions目录下的内容

[-->platform. xml]

<permissions>

<!-- 建立权限名与gid的映射关系。如下面声明的BLUETOOTH_ADMIN权限，它对应的用户组是

net_bt_admin。注意，该文件中的permission标签只对那些需要通过读写设备（蓝牙/camera）

/创建socket等进程划分了gid。因为这些权限涉及和Linux内核交互，所以需要在底层

权限（由不同的用户组界定）和Android层权限（由不同的字符串界定）之间建立映射关系

-->

<permission name="android.permission.BLUETOOTH_ADMIN">

<group gid="net_bt_admin"/>

</permission>

<permission name="android.permission.BLUETOOTH">

<group gid="net_bt"/>

</permission>

.....

<!--

赋予对应uid相应的权限。如果下面一行表示uid为shell，那么就赋予它SEND_SMS的权限，其实就是把它加到对应的用户组中-->

```
<assign-permission  
name="android.permission.SEND_SMS"uid="shell"/>  
    <assign-permission  
name="android.permission.CALL_PHONE"uid="shell"/>  
        <assign-permission  
name="android.permission.READ_CONTACTS"uid="shell"/>  
            <assign-permission  
name="android.permission.WRITE_CONTACTS"uid="shell"/>  
                <assign-permission  
name="android.permission.READ_CALENDAR"uid="shell"/>  
.....  
<!--系统提供的Java库，应用程序运行时候必须要链接这些库，该工作由系统  
自动完成-->  
    <library name="android.test.runner"  
file="/system/frameworks/android.test.runner.jar"/>  
    <library name="javax.obex"  
file="/system/frameworks/javax.obex.jar"/>
```

platform.xml文件中主要使用了如下4个标签：

permission 和 group 用于建立 Linux 层 gid 和 Android层pemission之间的映射关系。

assign-permission用于向指定的uid赋予相应的权限。这个权限由Android定义，用

字符串表示。

library用于指定系统库。当应用程序运行时，系统会自动为这些进程加载这些库。

了解了 platform.xml 后，再看其他的XML文件，这里以 handheld-core-hardware.xml 为

例进行介绍，其内容如下：

[~>handheld-core-hardware.xml]

这个XML文件包含了许多feature标签。根据该文件中的注释，这些feature标签用来描述一个手持终端（包括手机、平板电脑等）应该支持的硬件特性，例如支持camera、支持蓝牙等。

注意对于不同的硬件特性，还需要包含其他的XML文件。例如，要支持前置摄像头，还需要包含 android.hardware.camera.front.xml 文件。这些文件内容大体一样，都通过 feature 标签表明自己的硬件特性。相关说明可参考 handheld-core-hardware.xml 中的注释。

有读者可能会好奇，真实设备上 /system/etc/permission 目录中的文件是从哪里的呢？答案是，在编译阶段由不同硬件平台根据自己的配置信息复制相关文件到目标目录中得来的。

这里给出一个例子，如图4-4所示。

```
root@innost:/thunderst/work-branches/Android-4.0/device/htc# find -name *\*.mk | xargs grep \.xml
./passion-common/passion.mk:    frameworks/base/data/etc/handheld_core_hardware.xml:system/etc/permissions/hai
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.camera.flash-autofocus.xml:system/e
\
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.telephony.gsm.xml:system/etc/permis
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.location.gps.xml:system/etc/permiss
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.wifi.xml:system/etc/permissions/and
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.sensor.proximity.xml:system/etc/peri
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.sensor.light.xml:system/etc/permis
./passion-common/passion.mk:    frameworks/base/data/etc/android.software.sip.voip.xml:system/etc/permissions,
./passion-common/passion.mk:    frameworks/base/data/etc/android.hardware.touchscreen.multitouch.xml:system/e
./passion-common/passion.mk:# media config xml file
./passion-common/passion.mk:    device/htc/passion-common/media_profiles.xml:system/etc/media_profiles.xml
./passion/full_passion.mk:PRODUCT_COPY_FILES := device/sample/etc/apns-full-conf.xml:system/etc/apns-conf.xml
```

图 4-4 /system/etc/permission目录中文件的来源

由图 4-4 可知，当编译的设备目标为 htc-passion 时，就会将 Android 源码目录/frameworks/base/data/etc/下某些和该目标设备硬件特性匹配的 XML 文件复制到最终输出目录/system/etc/permissions 下。编译完成后，将生成 system 镜像。把该镜像文件烧到手机中，就成了目标设备使用的情况了。

注意 Android 4.0 源码中并没有与 htc 相关的文件，这是笔者从 Android 2.3 源码中移植过去的。读者可参考笔者一篇关于如何移植 Android 4.0 到 G7 的博文，地址为 <http://blog.csdn.net/innost/article/details/6977167>。

了解了与 XML 相关的知识后，再来分析 readPermissionFromXml 函数。相信读者已经知道它的作用了，就是将 XML 文件中的标签以及它们

之间的关系转换成代码中的相应数据结构，代码如下：

[-->PackageManagerService.java :
readPermissionsFromXml]

```
private void readPermissionsFromXml (File permFile) {  
    FileReader permReader=null;  
    try{  
        permReader=new FileReader (permFile) ;  
    }.....  
    try{  
        XmlPullParser parser=Xml.newPullParser () ;  
        parser.setInput (permReader) ;  
        XmlUtils.beginDocument (parser, "permissions") ;  
        while (true) {  
            String name=parser.getName () ;  
            //解析group标签，前面介绍的XML文件中没有单独使用该标签的地方  
            if ("group".equals (name) ) {  
                String gidStr=parser.getAttributeValue (null, "gid") ;  
                if (gidStr !=null) {  
                    int gid=Integer.parseInt (gidStr) ;  
                    //转换XML中的gid字符串为整型，并保存到mGlobalGids中  
                    mGlobalGids=appendInt (mGlobalGids, gid) ;  
                }.....  
            }else if ("permission".equals (name) ) {//解析permission标签  
                String perm=parser.getAttributeValue (null, "name") ;  
                .....  
                perm=perm.intern () ;  
                //调用readPermission处理  
                readPermission (parser, perm) ;  
                //下面解析的是assign-permission标签  
            }else if ("assign-permission".equals (name) ) {  
                String perm=parser.getAttributeValue (null, "name") ;
```

```
....  
String uidStr=parser.getAttributeValue (null, "uid") ;  
....  
//如果是assign-permission，则取出uid字符串，然后获得Linux平台上  
//的整型uid值  
int uid=Process.getUidForName (uidStr) ;  
....  
perm=perm.intern () ;  
//和assign相关的信息保存在mSystemPermissions中  
HashSet<String>perms=mSystemPermissions.get (uid) ;  
if (perms==null) {  
perms=new HashSet<String> () ;  
mSystemPermissions.put (uid, perms) ;  
}  
perms.add (perm) ;.....  
}else if ("library".equals (name) ) {//解析library标签  
String lname=parser.getAttributeValue (null, "name") ;  
String lfile=parser.getAttributeValue (null, "file") ;  
if (lname==null) {  
....  
}else if (lfile==null) {  
....  
}else{  
//将XML中的name和library属性值存储到mSharedLibraries中  
mSharedLibraries.put (lname, lfile) ;  
}....  
}else if ("feature".equals (name) ) {//解析feature标签  
String fname=parser.getAttributeValue (null, "name") ;  
....  
//在XML中定义的feature由FeatureInfo表达  
FeatureInfo fi=new FeatureInfo () ;  
fi.name=fname ;  
//存储feature名和对应的FeatureInfo到mAvailableFeatures中  
mAvailableFeatures.put (fname, fi) ;  
}....  
}....
```

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

readPermissions函数果然是将XML中的标签转换成对应的数据结构。总结相关的数据结构，如图4-5所示，此处借用了UML类图。在每个类图中，首行是数据结构名，第二行是数据结构的类型，第三行是注释。

这里必须再次强调：图4-5中各种数据结构的目的是为了保存XML中各种标签及它们之间的关系。在分析过程中，最重要的是理解各种标签的作用，而不是它们所使用的数据结构。

(2) readLPw的“佐料”

readLPw函数的功能也是解析文件，不过这些文件的内容却是在PKMS正常启动后生成的。这里仅介绍作为readLPw“佐料”的文件的信息。文件的具体位置在Settings构造函数中指明，其代码如下：

mGlobalGids +int数组 用于存储XML中group标签定义的gid	msharedlibrarie +HashMap<String, String> 通过解析library标签得来的 key为库的名字，value为库文件的位置
mSystemPermissions +SparseArray<HashSet<String>> UID为索引，存储一个字符串HashSet集合 用于描述指定UID所拥有的权限 通过解析assign-permission标签得来的	mAvailableFeatures +HashMap<String, FeatureInfo> 解析feature标签得来的。key为feature的 字符串描述，value为FeatureInfo对象
mSettings.mPermissions +HashMap<String, BasePermission> 解析permission标签得来的，key为权限名，value为BasePermission对象BasePermission 中有一个gid数组，解析permission标签中的group标签时会存储对应的gid到该gid数组中	

图 4-5 通过readPermissions函数建立的数据结构及其关系

[--> Settings.java : Settings]

```
Settings () {
    File dataDir=Environment.getDataDirectory () ;
    File systemDir=new File ( dataDir , "system" ) ; // 指
    向/data/system目录
    systemDir.mkdirs () ; //创建该目录
    ....
    /*

```

一共有5个文件， packages.xml和packages-backup.xml为一组，用于描述
系统中所安装的

Package的信息，其中backup是临时文件。PKMS先把数据写到backup中，信息
都写成功后再

改名成非backup的文件。其目的是防止在写文件过程中出错，导致信息丢失。
packages-stopped.xml

xml和packages-stopped-backup.xml为一组，用于描述系统中强制停止运行的Pakcage的

信息，backup也是临时文件。如果此处存在该临时文件，表明此前系统因为某种原因中断了正常流

程。packages.list列出当前系统中应用级（即UID大于10000）Package的信息

```
/*
mSettingsFilename=new File (systemDir, "packages.xml") ;
mBackupSettingsFilename=new File ( systemDir , "packages-
backup.xml" ) ;
mPackageListFilename=new File (systemDir, "packages.list") ;
mStoppedPackagesFilename=new File ( systemDir , "packages-
stopped.xml" ) ;
mBackupStoppedPackagesFilename=new File (systemDir,
"packages-stopped-backup.xml" ) ;
}
```

上面5个文件共分为3组，这里简单介绍一下这些文件的来历（不考虑临时的backup文件）。

packages.xml：PKMS扫描完目标文件夹后会创建该文件。当系统进行程序安装、卸载和更新等操作时，均会更新该文件。该文件保存了系统中与Package相关的一些信息。

packages.list：描述系统中存在的所有非系统自带的APK的信息。当这些程序有变动时，PKMS就会更新该文件。

packages-stopped.xml：从系统自带的设置程序中进入应用程序页面，然后在选择强制停止（Force Stop）某个应用时，系统会将该应用的相关信息记录到此文件中。也就是该文件保存系统中被用户强制停止的Package的信息。

readLPw的函数功能就是解析其中的XML文件的内容，然后建立并更新对应的数据结构，例如停止的Package重启之后依然是stopped状态。

提示 读者看完本章后，可自行分析该函数。在此之前，建议读者不必关注该函数。

3.第一阶段工作总结

在继续征程前，先总结一下PKMS构造函数在第一阶段的工作，千言万语汇成一句话：扫描并解析XML文件，将其中的信息保存到特定的数据结构中。

第一阶段扫描的XML文件与权限及上一次扫描得到的Package信息有关，它为PKMS下一阶段的工作提供了重要的参考信息。

4.3.2 构造函数分析之扫描Package

PKMS构造函数第二阶段的工作就是扫描系统中的APK了。由于需要逐个扫描文件，因此手机上装的程序越多，PKMS的工作量就越大，系统启动速度也就越慢。

1. 系统库的dex优化

接着对PKMS构造函数进行分析，代码如下：

[-->vPackageManagerService.java]

```
.....  
mRestoredSettings=mSettings.readLPw () ;//接第一段的结尾  
long startTime=SystemClock.uptimeMillis () ;//记录扫描开始的时间  
//定义扫描参数  
int scanMode=SCAN_MONITOR|SCAN_NO_PATHS|SCAN_DEFER_DEX ;  
if (mNoDexOpt) {  
    scanMode|=SCAN_NO_DEX ;//在控制扫描过程中是否对APK文件进行dex优化  
}  
final HashSet<String> libFiles=new HashSet<String> () ;  
//mFrameworkDir指向/system/frameworks目录  
mFrameworkDir=new File ( Environment.getRootDirectory()  
) , "framework") ;  
//mDalvikCacheDir指向/data/dalvik-cache目录  
mDalvikCacheDir=new File (dataDir, "dalvik-cache") ;  
boolean didDexOpt=false ;
```

```
/*
    获取Java启动类库的路径，在init.rc文件中通过BOOTCLASSPATH环境变量输出，该值如下
        /system/framework/core.jar      :      /system/frameworks/core-
junit.jar :
        /system/frameworks/bouncycastle.jar : /system/frameworks/ext.
jar :
        /system/frameworks/framework.jar : /system/frameworks/android
.policy.jar :
        /system/frameworks/services.jar : /system/frameworks/apache-
xml.jar :
        /system/frameworks/filterfw.jar
    该变量指明了framework所有核心库及文件位置
*/
String bootClassPath=System.getProperty
("java.boot.class.path") ;
if (bootClassPath !=null) {
String[]paths=splitString (bootClassPath, '：') ;
for (int i=0 ; i<paths.length ; i++) {
try{//判断该Jar包是否需要重新做dex优化
if (dalvik.system.DexFile.isDexOptNeeded (paths[i]) ) {
/*
    将该Jar包文件路径保存到libFiles中，然后通过mInstall对象发送
命令给installld，让其对该Jar包进行dex优化
*/
libFiles.add (paths[i]) ;
mInstaller.dexopt (paths[i], Process.SYSTEM_UID, true) ;
didDexOpt=true ;
}
}.....
}
}.....
/*
```

还记得mSharedLibraries的作用吗？它保存的是platform.xml中声明的系统库的信息。

这里也要判断系统库是否需要做dex优化。处理方式同上

```
*/  
if (mSharedLibraries.size () >0) {  
.....  
}  
//将framework-res.apk添加到libFiles中。framework-res.apk定义了  
系统常用的  
//资源，还有几个重要的Activity，如长按Power键后弹出的选择框  
libFiles.add ( mFrameworkDir.getPath () +"/framework-  
res.apk") ;  
//列举/system/frameworks目录中的文件  
String[]frameworkFiles=mFrameworkDir.list () ;  
if (frameworkFiles !=null) {  
.....//判断该目录下的APK或Jar文件是否需要做dex优化。处理方式同上  
}/*
```

上面代码对系统库（BOOTCLASSPATH指定，或platform.xml定义，或/system/frameworks目录下的Jar包与APK文件）进行一次仔细检查，该优化的一定要优化。

如果发现期间对任何一个文件进行了优化，则设置didDexOpt为true

```
*/  
if (didDexOpt) {  
String[]files=mDalvikCacheDir.list () ;  
if (files !=null) {  
/*
```

如果前面对任意一个系统库重新做过dex优化，就需要删除cache文件。原因和dalvik虚拟机的运行机制有关。本书暂不探讨dex及cache文件的作用。

从删除cache文件这个操作来看，这些cache文件应该使用了dex优化后的系统库，

所以当系统库重新做dex优化后，就需要删除旧的cache文件。可简单理解为缓存失效

```
/*  
for (int i=0 ;i<files.length ;i++) {  
String fn=files[i] ;  
if (fn.startsWith ("data@app@")  
||fn.startsWith ("data@app-private@") ) {  
(new File (mDalvikCacheDir, fn) ) .delete () ;  
.....
```

```
}
```

2. 扫描系统Package

清空cache文件后，PKMS终于进入重点段了。接下来看PKMS第二阶段工作的核心内容，即扫描Package，相关代码如下：

[--> PackageManagerService.java]

```
//创建文件夹监控对象，监视/system/frameworks目录。利用了Linux平台的  
inotify机制
```

```
mFrameworkInstallObserver=new AppDirObserver (  
    mFrameworkDir.getPath () , OBSERVER_EVENTS, true) ;  
mFrameworkInstallObserver.startWatching () ;  
/*
```

调用scanDirLI函数扫描/system/frameworks目录，这个函数很重要，稍后会再分析。

注意，在第三个参数中设置了SCAN_NO_DEX标志，因为该目录下的package在前面的流程

中已经过判断并根据需要做过dex优化了

```
*/  
scanDirLI (mFrameworkDir, PackageParser.PARSE_IS_SYSTEM  
|PackageParser.PARSE_IS_SYSTEM_DIR, scanMode|SCAN_NO_DEX ,  
0) ;
```

//创建文件夹监控对象，监视/system/app目录

```
mSystemAppDir=new     File     ( Environment.getRootDirectory  
() , "app") ;
```

```
mSystemInstallObserver=new AppDirObserver (
```

```
mSystemAppDir.getPath () , OBSERVER_EVENTS, true) ;
```

```
mSystemInstallObserver.startWatching () ;
```

//扫描/system/app下的package

```
scanDirLI (mSystemAppDir, PackageParser.PARSE_IS_SYSTEM
```

```
| PackageParserPARSE_IS_SYSTEM_DIR, scanMode, 0) ;  
//监视并扫描/vendor/app目录  
mVendorAppDir=new File ("/vendor/app") ;  
mVendorInstallObserver=new AppDirObserver (  
mVendorAppDir.getPath (), OBSERVER_EVENTS, true) ;  
mVendorInstallObserver.startWatching () ;  
//扫描/vendor/app下的package  
scanDirLI (mVendorAppDir, PackageParserPARSE_IS_SYSTEM  
| PackageParserPARSE_IS_SYSTEM_DIR, scanMode, 0) ;  
//和installd交互。以后单独分析installd  
mInstaller.moveFiles () ;
```

由以上代码可知，PKMS将扫描以下几个目录。

/system/frameworks：该目录中的文件都是系统库，例如framework.jar、services.jar、framework-res.apk。不过scanDirLI只扫描APK文件，所以framework-res.apk是该目录中唯一“受宠”的文件。

/system/app：该目录下全是默认的系统应用，例如Browser.apk、SettingsProvider.apk等。

/vendor/app：该目录中的文件由厂商提供，即全是厂商特定的APK文件，目前市面上的厂商都把自己的应用放在/system/app目录下。

注意 本书把这3个目录称为系统package目录，以区分后面的非系统package目录。

PKMS调用scanDirLI函数进行扫描，下面来分析此函数。

(1) scanDirLI函数分析

scanDirLI函数的代码如下：

[-->PackageManagerService.java : scanDirLI]

```
private void scanDirLI (File dir, int flags, int scanMode,
long currentTime) {
    String[]files=dir.list () ;//列举该目录下的文件
    .....
    int i ;
    for (i=0 ; i<files.length ; i++) {
        File file=new File (dir, files[i]) ;
        if (!isPackageFilename (files[i]) ) {
            continue ;//根据文件名后缀，判断是否为APK文件。这里只扫描APK文件
        }/*
        调用scanPackageLI函数扫描一个特定的文件，返回值是PackageParser的内
部类
        Package，该类的实例代表一个APK文件，所以它就是和APK文件对应的数据结构
     */
    PackageParser.Package pkg=scanPackageLI (file,
        flags|PackageParser.PARSE_MUST_BE_APK,           scanMode,
        currentTime) ;
    if (pkg==null&& (flags&PackageParser.PARSE_IS_SYSTEM) ==0 &
    &
    mLastScanError==PackageManager.INSTALL_FAILED_INVALID_APK) {
        //注意此处flags的作用，只有非系统Package扫描失败，才会删除该文件
        file.delete () ;
    }
}
```

接着来分析scanPackageLI函数。PKMS中有两个同名的scanPackageLI函数，后面会一一见到。先来看第一个也是最先碰到的scanPackageLI函数。

(2) 初会scanPackageLI函数

首次相遇的scanPackageLI函数的代码如下：

[-->PackageManagerService.java :
scanPackageLI]

```
private PackageParser.Package scanPackageLI (File scanFile,  
int parseFlags,  
int scanMode, long currentTime)  
{  
    mLastScanError=PackageManager.INSTALL_SUCCEEDED ;  
    String scanPath=scanFile.getPath () ;  
    parseFlags |=mDefParseFlags ;//默认的扫描标志，正常情况下其值为0  
    //创建一个PackageParser对象  
    PackageParser pp=new PackageParser (scanPath) ;  
    pp.setSeparateProcesses  
(mSeparateProcesses) ;//mSeparateProcesses为空  
    pp.setOnlyCoreApps (mOnlyCore) ;//mOnlyCore为false  
/*  
调用PackageParser的parsePackage函数解析APK文件。注意，这里把代表屏  
幕  
信息的mMetrics对象也传了进去  
*/  
final PackageParser.Package pkg=pp.parsePackage (scanFile,  
scanPath, mMetrics, parseFlags) ;  
.....  
PackageSetting ps=null ;
```

```
PackageSetting updatedPkg ;
```

```
.....
```

```
/*
```

这里略去一大段代码，主要是关于Package升级方面的工作。读者可能会比较好奇：既然是

升级，一定有新旧之分，如果这里刚解析后得到的Package信息是新的，那么旧Package

的信息从何得来？还记得4.3.1节中提到的package.xml文件吗？此文件中存储的就是

上一次扫描得到的Package信息。对比这两次的信息就知道是否需要做升级了。这部分代码

比较烦琐，但不影响我们正常分析。感兴趣的读者可自行研究

```
*/
```

```
//收集签名信息，这部分内容涉及signature，本书暂不讨论[1]。
```

```
if ( ! collectCertificatesLI ( pp, ps, pkg, scanFile, parseFlags) )
```

```
    return null;
```

```
//判断是否需要设置PARSE_FORWARD_LOCK标志，这个标志针对资源文件和Class文件
```

//不在同一个目录的情况。目前只有/vendor/app目录下的扫描会使用该标志。这里不讨论

```
//这种情况。
```

```
if (ps !=null&& ! ps.codePath.equals (ps.resourcePath) )
```

```
parseFlags |=PackageParser.PARSE_FORWARD_LOCK ;
```

```
String codePath=null ;
```

```
String resPath=null ;
```

```
if ( (parseFlags&PackageParser.PARSE_FORWARD_LOCK) !=0) {
```

```
.....//这里不考虑PARSE_FORWARD_LOCK的情况。
```

```
}else{
```

```
resPath=pkg.mScanPath ;
```

```
}
```

```
codePath=pkg.mScanPath ; //mScanPath指向该APK文件所在位置
```

```
//设置文件路径信息，codePath和resPath都指向APK文件所在位置
```

```
setApplicationInfoPaths (pkg, codePath, resPath) ;
```

```
//调用第二个scanPackageLI函数
```

```
    return      scanPackageLI      (      pkg,      parseFlags,
scanMode|SCAN_UPDATE_SIGNATURE,
currentTime) ;
}
```

scanPackageLI函数首先调用PackageParser对APK文件进行解析。根据前面的介绍可知，PackageParser完成了从物理文件到对应数据结构的转换。下面来分析这个PackageParser。

(3) PackageParser分析

PackageParser主要负责APK文件的解析，即解析APK文件中的AndroidManifest.xml代码如下：

[-->PackageParser.java : parsePackage]

```
public  Package  parsePackage  (  File   sourceFile,   String
destCodePath,
DisplayMetrics metrics, int flags) {
mParseError=PackageManager.INSTALL_SUCCEEDED;
mArchiveSourcePath=sourceFile.getPath() ;
.....//检查是否为APK文件
XmlResourceParser parser=null;
AssetManager assmgr=null;
Resources res=null;
boolean assetError=true ;
try{
assmgr=new AssetManager () ;
int cookie=assmgr.addAssetPath (mArchiveSourcePath) ;
if (cookie !=0) {
res=new Resources (assmgr, metrics, null) ;
```

以上代码中调用了另一个同名的 PackageParser 函数，此函数内容较长，但功能单一，就是解析 AndroidManifest.xml 中的各种标签，这里只提取其中相关的代码：

[-->PackageParser.java : parsePackage]

```
private Package parsePackage (
    Resources res,   XmlResourceParser parser,   int flags,
String[]outError)
    throws XmlPullParserException, IOException{
    AttributeSet attrs=parser ;
    mParseInstrumentationArgs=null ;
    mParseActivityArgs=null ;
    mParseServiceArgs=null ;
    mParseProviderArgs=null ;
    //得到Package的名字，其实就是得到AndroidManifest.xml中Package属性的值，
    //每个APK都必须定义该属性
    String pkgName=parsePackageName ( parser,   attrs,   flags,
outError) ;

.....
int type ;

.....
//以pkgName名字为参数，创建一个Package对象。后面的工作就是解析XML并填充
//该Package信息
final Package pkg=new Package ( pkgName) ;
boolean foundApp=false ;
.....//下面开始解析该文件中的标签，由于这段代码功能简单，所以这里仅列举相关函数
while ( ...../*如果解析未完成*/ ) {
.....
String tagName=parser.getName () ;//得到标签名
if (tagName.equals ("application") ) {
.....//解析application标签
parseApplication ( pkg,   res,   parser,   attrs,   flags,
outError) ;
}else if (tagName.equals ("permission-group") ) {
.....//解析permission-group标签
parsePermissionGroup ( pkg, res, parser, attrs, outError) ;
}else if (tagName.equals ("permission") ) {
```

```
.....//解析permission标签
parsePermission (pkg, res, parser, attrs, outError) ;
}else if (tagName.equals ("uses-permission") ) {
//从XML文件中获取uses-permission标签的属性
sa=res.obtainAttributes (attrs,
com.android.internal.R.styleable.AndroidManifestUsesPermissi
on) ;
//取出属性值，也就是对应的权限使用声明
String name=sa.getNonResourceString (com.android.internal.
R.styleable.AndroidManifestUsesPermission_name) ;
//添加到Package的requestedPermissions数组
if ( name ! =null & & ! pkg.requestedPermissions.contains
(name) ) {
pkg.requestedPermissions.add (name.intern ()) ;
}
}else if (tagName.equals ("uses-configuration") ) {
/*
该标签用于指明本Package对硬件的一些设置参数，目前主要针对输入设备（触
摸屏、键盘
等）。游戏类的应用可能对此有特殊要求。
*/
ConfigurationInfo cPref=new ConfigurationInfo () ;
.....//解析该标签所支持的各种属性
pkg.configPreferences.add ( cPref ) ;// 保 存 到 Package 的
configPreferences数组
}
.....//对其他标签解析和处理
}
```

上面代码展示了AndroidManifest.xml解析的流程，其中比较重要的函数是parser-Application，它用于解析application标签及其子标签（Android的四大组件在application标签中已声明）。

图4-6表示出了PackageParser及其内部重要成员的信息。

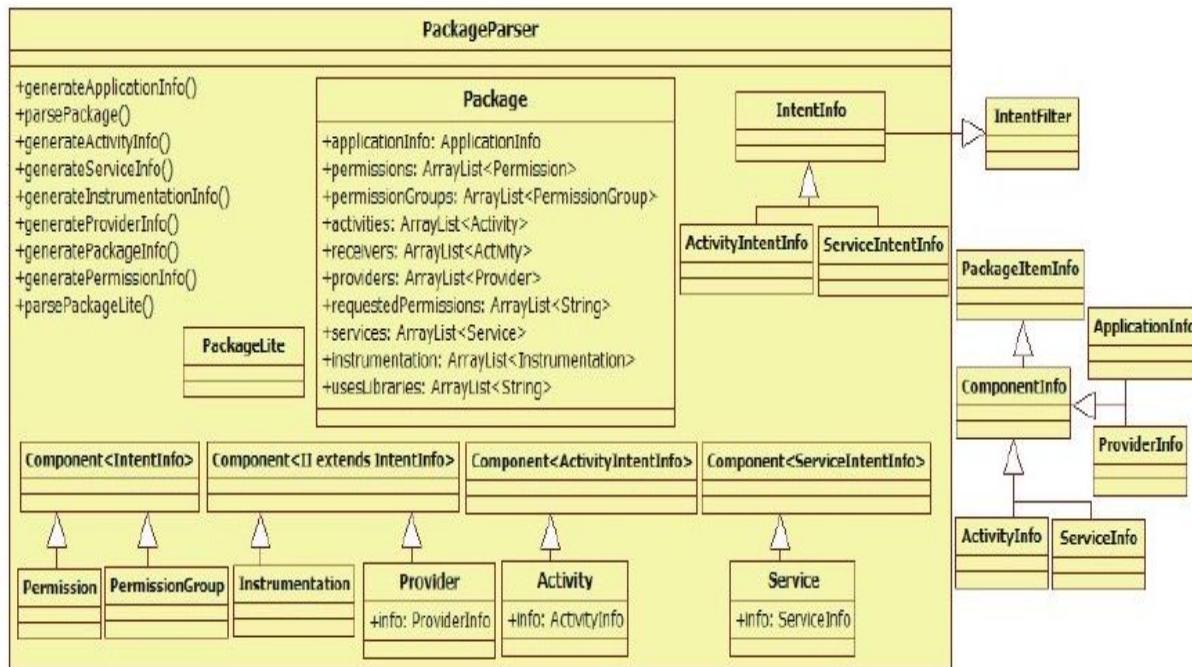


图 4-6 PackageParser家族

由图4-6可知：

PackageParser定义了相当多的内部类，这些内部类的作用就是保存对应的信息。解析AndroidManifest.xml文件得到的信息由Package保存。从该类的成员变量可看出，和Android四大组件相关的信息分别由 activities、receivers、providers、services保存。由于一个APK可声明多个组件，因此 activities 和 receivers 等均声明为 ArrayList。

以 PackageParser.Activity 为例，它从 Component<ActivityIntentInfo> 派生。Component 是一个模板类，元素类型是 ActivityIntentInfo，此类的顶层基类是 IntentFilter。PackageParser.Activity 内部有一个 ActivityInfo 类型的成员变量，该变量保存的就是四大组件中 Activity 的信息。细心的读者可能会有疑问，为什么不直接使用 ActivityInfo，而是通过 IntentFilter 构造出一个使用模板的复杂类型 PackageParser.Activity 呢？原来，Package 除了保存信息外，还需要支持 Intent 匹配查询。例如，设置 Intent 的 Action 为某个特定值，然后查找匹配该 Intent 的 Activity。由于 ActivityIntentInfo 是从 IntentFilter 派生的，因此它能判断自己是否满足该 Intent 的要求，如果满足，则返回对应的 ActivityInfo。在后续章节会详细讨论根据 Intent 查询特定 Activity 的工作流程。

PackageParser 定了一个轻量级的数据结构 PackageLite，该类仅存储 Package 的一些简单信息。我们在介绍 Package 安装的时候，会遇到 PackageLite。

注意 读者需要了解 Java 泛型类的相关知识。

(4) 与 scanPackageLI 再相遇

在PackageParser扫描完一个APK后，此时系统已经根据该APK中AndroidManifest.xml，创建了一个完整的Package对象，下一步就是将该Package加入到系统中。此时调用的函数就是另外一个scanPackageLI，其代码如下：

[--> PackageManagerService.java :
scanPackageLI]

```
private          PackageParser.Package      scanPackageLI
(PackageParser.Package pkg,
 int parseFlags, int scanMode, long currentTime) {
 File scanFile=new File (pkg.mScanPath) ;
 .....
 mScanningPath=scanFile ;
 //设置package对象中applicationInfo的flags标签，用于标示该Package
为系统
 //Package
 if ( (parseFlags&PackageParser.PARSE_IS_SYSTEM) !=0) {
 pkg.applicationInfo.flags|=ApplicationInfo.FLAG_SYSTEM ;
 }
 //①下面这句if判断极为重要，见下面的解释
 if (pkg.packageName.equals ("android") ) {
 synchronized (mPackages) {
 if (mAndroidApplication !=null) {
 .....
 mPlatformPackage=pkg ;
 pkg.mVersionCode=mSdkVersion ;
 mAndroidApplication=pkg.applicationInfo ;
 mResolveActivity.applicationInfo=mAndroidApplication ;
 mResolveActivity.name=ResolverActivity.class.getName () ;
 mResolveActivity.packageName=mAndroidApplication.packageName
 ;
```

```
mResolveActivity.processName=mAndroidApplication.processName  
;  
mResolveActivity.launchMode=ActivityInfo.LAUNCH_MULTIPLE ;  
mResolveActivity.flags=ActivityInfo.FLAG_EXCLUDE_FROM_RECENT  
S ;  
mResolveActivity.theme=  
com.android.internal.R.style.Theme_Holo_Dialog_Alert ;  
mResolveActivity.exported=true ;  
mResolveActivity.enabled=true ;  
//mResolveInfo的activityInfo成员指向mResolveActivity  
mResolveInfo.activityInfo=mResolveActivity ;  
mResolveInfo.priority=0 ;  
mResolveInfo.preferredOrder=0 ;  
mResolveInfo.match=0 ;  
mResolveComponentName=new ComponentName (  
mAndroidApplication.packageName, mResolveActivity.name) ;  
}  
}
```

刚进入scanPackageLI函数，我们就发现了一个极为重要的内容，即单独判断并处理packageName为“android”的Package。和该Package对应的APK是framework-res.apk，有图为证，如图4-7所示为该APK的AndroidManifest.xml中的相关内容。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
.... package="android".coreApp="true".android:sharedUserId="android.uid.system"  
.... android:sharedUserLabel="@string/android_system_label">
```

图 4-7 framework-res.apk的AndroidManifest.xml

实际上，framework-res.apk还包含了以下几个常用的Activity。

ChooserActivity：当多个Activity符合某个Intent的时候，系统会弹出此Activity，由用户选择合适的应用来处理。

RingtonePickerActivity：铃声选择Activity。

ShutdownActivity：关机前弹出的选择对话框。

由前述知识可知，该Package和系统息息相关，因此它得到了PKMS的特别青睐，主要体现在以下几点：

mPlatformPackage成员用于保存该Package信息。

mAndroidApplication用于保存此Package中的ApplicationInfo。

mResolveActivity指向用于表示ChooserActivity信息的ActivityInfo。

mResolveInfo为ResolveInfo类型，它用于存储系统解析Intent（经IntentFilter的过滤）后得到的结果信息，例如满足某个Intent的Activity的信息。

由前面的代码可知，`mResolveInfo`的`activityInfo`其实指向的就是`mResolveActivity`。

注意 在从PKMS中查询满足某个Intent的Activity时，返回的就是`ResolveInfo`，再根据`ResolveInfo`的信息得到具体的Activity。

此处保存这些信息，主要是为了提高运行过程中的效率。Goolge工程师可能觉得`ChooserActivity`使用的地方比较多，所以这里单独保存了此Activity的信息。

好，继续对`scanPackageLI`函数进行分析。

[-->PackageManagerService : `scanPackageLI`]

```
.....//mPackages用于保存系统内的所有Package，以packageName为key
if (mPackages.containsKey (pkg.packageName)
||mSharedLibraries.containsKey (pkg.packageName) ) {
return null;
}
File destCodeFile=new File (pkg.applicationInfo.sourceDir) ;
File destResourceFile=new File
(pkg.applicationInfo.publicSourceDir) ;
SharedUserSetting     uid=null;// 代 表 该 Package 的
SharedUserSetting对象
PackageSetting     pkgSetting=null;// 代 表 该 Package 的
PackageSetting对象
synchronized (mPackages) {
.....//此段代码有300行左右，主要做了以下几方面工作
/*
①如果该Packge声明了uses-librarie，那么系统要判断该library是否
```

在mSharedLibraries中

②如果package声明了SharedUser，则需要处理SharedUserSettings相关内容，

由Settings的getSharedUserLPw函数处理

③处理pkgSetting，通过调用Settings的getPackageLPw函数完成

④调用verifySignaturesLP函数，检查该Package的signature

*/

}

final long scanFileTime=scanFile.lastModified () ;

final boolean forceDex= (scanMode&SCAN_FORCE_DEX) !=0 ;

//确定运行该package的进程的进程名，一般用packageName作为进程名

pkg.applicationInfo.packageName=fixProcessName (

pkg.applicationInfo.packageName,

pkg.applicationInfo.packageName,

pkg.applicationInfo.uid) ;

if (mPlatformPackage==pkg) {

dataPath=new File (Environment.getDataDirectory () , "system") ;

pkg.applicationInfo.dataDir=dataPath.getPath () ;

}else{

/*

getDataPathForPackage函数返回该package的目录

一般是/data/data/packageName/

*/

dataPath=getDataPathForPackage (pkg.packageName, 0) ;

if (dataPath.exists ()) {

.....//如果该目录已经存在，则要处理uid的问题

}else{

.....//向installd发送install命令，实际上就是在/data/data目录下

//建立packageName目录。后续将分析installd相关知识

int ret=mInstaller.install (pkgName,

pkg.applicationInfo.uid,

pkg.applicationInfo.uid) ;

//为系统所有user安装此程序

mUserManager.installPackageForAllUsers (pkgName,

pkg.applicationInfo.uid) ;

```

if (dataPath.exists () ) {
    pkg.applicationInfo.dataDir=dataPath.getPath () ;
}.....
if (pkg.applicationInfo.nativeLibraryDir==null&&
    pkg.applicationInfo.dataDir !=null) {
    .....//为该Package确定native library所在目录
    //一般是/data/data/packagename/lib
}
}

//如果该APK包含了native动态库，则需要将它们从APK文件中解压并复制到对应目录中
if (pkg.applicationInfo.nativeLibraryDir !=null) {
try{
    final File nativeLibraryDir=new
    File (pkg.applicationInfo.nativeLibraryDir) ;
    final String dataPathString=dataPath.getCanonicalPath () ;
    //从Android 2.3开始，系统package的native库统一放在/system/lib
    下。所以
    //系统不会提取系统Package目录下APK包中的native库
    if (isSystemApp (pkg) && !isUpdatedSystemApp (pkg) ) {
        NativeLibraryHelper.removeNativeBinariesFromDirLI (
            nativeLibraryDir) ;
    }else if (nativeLibraryDir.getParentFile () .getCanonicalPath
    () .
        .equals (dataPathString) ) {
        boolean isSymLink ;
        try{
            isSymLink=S_ISLNK (Libcore.os.lstat (
                nativeLibraryDir.getPath () ) .st_mode) ;
        }.....//判断是否为链接，如果是，需要删除该链接
        if (isSymLink) {
            mInstaller.unlinkNativeLibraryDirectory (dataPathString) ;
        }
        //在lib下建立和CPU类型对应的目录，例如ARM平台的是arm/，MIPS平台的是
        mips/
        NativeLibraryHelper.copyNativeBinariesIfNeededLI (scanFile,

```

```

nativeLibraryDir) ;
}else{
mInstaller.linkNativeLibraryDirectory (dataPathString,
pkg.applicationInfo.nativeLibraryDir) ;
}
}.....
}

pkg.mScanPath=path ;
if ( (scanMode&SCAN_NO_DEX) ==0) {
.....//对该APK做dex优化
performDexOptLI (pkg, forceDex, (scanMode&SCAN_DEFER_DEX) ;
}
//如果该APK已经存在，要先杀掉运行该APK的进程
if ( (parseFlags & PackageManager.INSTALL_REPLACE_EXISTING) !
=0) {
killApplication (pkg.applicationInfo.packageName,
pkg.applicationInfo.uid) ;
}
.....
/*

```

在此之前，四大组件信息都属于Package的私有财产，现在需要把它们登记注册到PKMS内部的

财产管理对象中。这样，PKMS就可对外提供统一的组件信息，而不必拘泥于具体的Package

```

*/
synchronized (mPackages) {
if ( (scanMode&SCAN_MONITOR) !=0) {
mAppDirs.put (pkg.mPath, pkg) ;
}
mSettings.insertPackageSettingLPw (pkgSetting, pkg) ;
mPackages.put (pkg.applicationInfo.packageName, pkg) ;
//处理该Package中的Provider信息
int N=pkg.providers.size () ;
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) ;
//mProvidersByComponent提供基于ComponentName的Provider信息查询
mProvidersByComponent.put (new ComponentName (
    p.info.packageName, p.info.name) , p) ;
.....
}
//处理该Package中的Service信息
N=pkg.services.size () ;
r=null ;
for (i=0 ; i<N ; i++) {
    PackageParser.Service s=pkg.services.get (i) ;
    mServices.addService (s) ;
}
//处理该Package中的BroadcastReceiver信息
N=pkg.receivers.size () ;
r=null ;
for (i=0 ; i<N ; i++) {
    PackageParser.Activity a=pkg.receivers.get (i) ;
    mReceivers.addActivity (a, "receiver") ;
.....
}
//处理该Package中的Activity信息
N=pkg.activities.size () ;
r=null ;
for (i=0 ; i<N ; i++) {
    PackageParser.Activity a=pkg.activities.get (i) ;
    mActivities.addActivity (a, "activity") ;//后续将详细分析该调用
}
//处理该Package中的PermissionGroups信息
N=pkg.permissionGroups.size () ;
.....//permissionGroups处理
N=pkg.permissions.size () ;
.....//permissions处理
N=pkg.instrumentation.size () ;
```

```
.....//instrumentation处理
if (pkg.protectedBroadcasts !=null) {
N=pkg.protectedBroadcasts.size () ;
for (i=0 ; i<N ; i++) {
mProtectedBroadcasts.add      (    pkg.protectedBroadcasts.get
(i) ) ;
}
}
.....//Package的私有财产终于完成了公有化改造
return pkg ;
}
```

到此这个长达800行的代码就分析完了，下面总结一下Package扫描的流程。（5）scanDirLI函数总结

scanDirLI用于对指定目录下的APK文件进行扫描，如图4-8所示为该函数的调用流程。

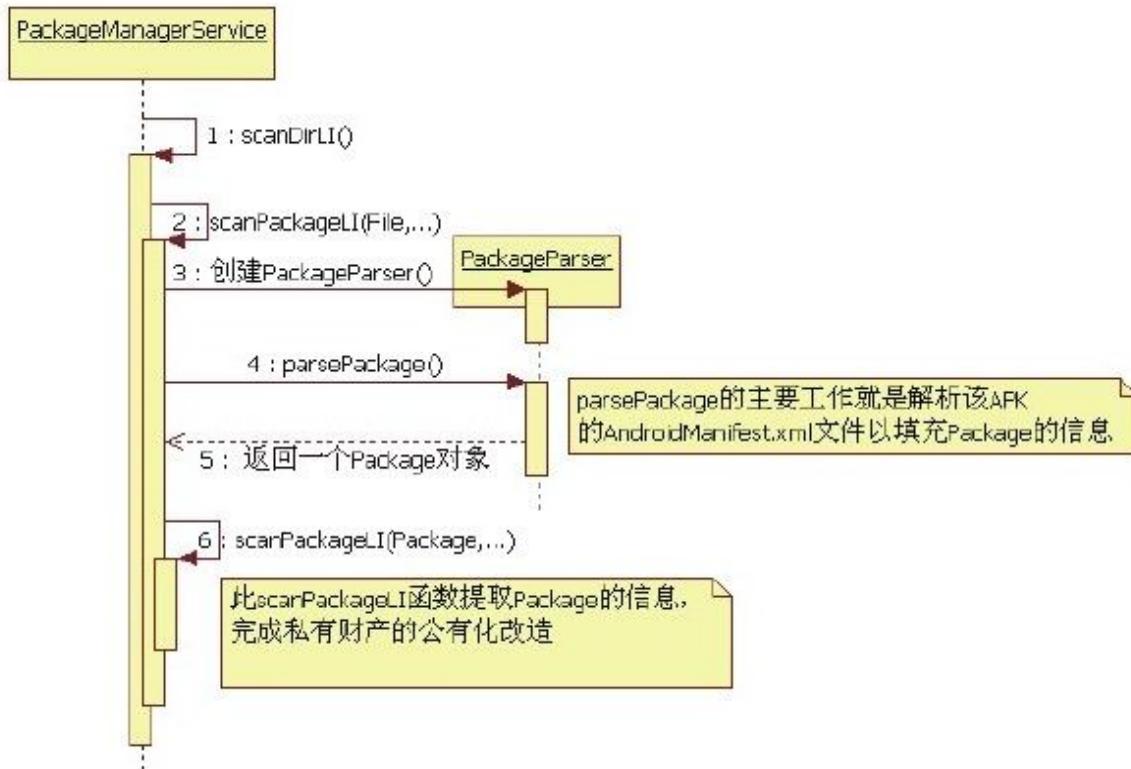


图 4-8 scanDirLI工作流程总结

图4-8比较简单，相关知识无须赘述。读者在自行分析代码时，只要注意区分两个scanPackageLI函数即可。

扫描完APK文件后，Package的私有财产就充公了。PKMS提供了好几个重要数据结构来保存这些财产，这些数据结构的相关信息如图4-9所示。

mActivities	mInstrumentation
+ActivityIntentResolver	+HashMap<ComponentName, PackageParser.Instrumentation>
用于保存所有Activity的信息	
mServices	mProviders
+ServiceIntentResolver	+HashMap<String, PackageParser.Provider>
用于保存所有Service的信息	以Provider的路径为key, 保存Provider信息
mReceivers	+ActivityIntentResolver
用于保存所有BroadcastReceiver的信息	
mProvidersByComponent	mPackages
+HashMap<ComponentName, PackageParser.Provider>	+HashMap<String, PackageParser.Package>
以ComponentName为key, 保存Provider信息	以PackageName为Key, 保存系统中所有Package信息

图 4-9 PKMS中重要的数据结构

图4-9借用UML的类图来表示PKMS中重要的数据结构。每个类图的第一行为成员变量名，第二行为数据类型，第三行为注释说明。

3.扫描非系统Package

非系统Package就是指那些不存储在系统目录下的APK文件，这部分代码如下：

[-->PackageManagerService.java : 构造函数
第三部分]

```

if (!mOnlyCore) { //mOnlyCore用于控制是否扫描非系统Package
    Iterator<PackageSetting> psit=
        mSettings.mPackages.values () .iterator () ;
    while (psit.hasNext ()) {
        ....//删除系统package中那些不存在的APK
    }
}

```

```
mAppInstallDir=new File (dataDir, "app") ;
.....//删除安装不成功的文件及临时文件
if (!mOnlyCore) {
//在普通模式下，还需要扫描/data/app以及/data/app_private目录
mAppInstallObserver=new AppDirObserver (
mAppInstallDir.getPath () , OBSERVER_EVENTS, false) ;
mAppInstallObserver.startWatching () ;
scanDirLI (mAppInstallDir, 0, scanMode, 0) ;
mDrmAppInstallObserver=new AppDirObserver (
mDrmAppPrivateInstallDir.getPath () , OBSERVER_EVENTS,
false) ;
mDrmAppInstallObserver.startWatching () ;
scanDirLI (mDrmAppPrivateInstallDir,
PackageParser.PARSE_FORWARD_LOCK, scanMode, 0) ;
}else{
mAppInstallObserver=null ;
mDrmAppInstallObserver=null ;
}
结合前述代码，这里总结几个存放APK文件的目录。
```

系 统 Package 目 录 包
括 : /system/frameworks 、 /system/app
和/vendor/app。

非 系 统 Package 目 录 包
括 : /data/app、 /data/app-private。

4.第二阶段工作总结

PKMS构造函数第二阶段的工作任务非常繁重，要创建比较多的对象，所以它是一个耗时耗内存的操作。在工作中，我们一直想优化该流程

以加快启动速度，例如延时扫描不重要的APK，或者保存Package信息到文件中，然后在启动时从文件中恢复这些信息以减少APK文件读取并解析XML的工作量。但是一直没有一个比较完满的解决方案，原因有很多。比如APK之间有着比较微妙的依赖关系，因此到底延时扫描哪些APK，尚不能确定。另外，笔者感到比较疑惑的一个问题是：对于多核CPU架构，PKMS可以启动多个线程以扫描不同的目录，但是目前代码中还没有寻找到了相关的蛛丝马迹。难道此处真的就不能优化了吗？读者如果有更好的解决方案，不妨和大家分享一下。

[1]Signature和Android安全机制有关。本系列书后续拟考虑编写有关Android安全方面的专题卷。感兴趣的读者也可先阅读《Application Security for the Android Platform》一书。

4.3.3 构造函数分析之扫尾工作

下面分析PKMS第三阶段的工作，这部分任务比较简单，就是将第二阶段收集的信息再集中整理一次，比如将有些信息保存到文件中，相关代码如下：

[--> PackageManagerService.java : 构造函数]

```
....  
mSettings.mInternalSdkPlatform=mSdkVersion ;  
//汇总并更新和Permission相关的信息  
updatePermissionsLPw (null, null, true,  
regrantPermissions, regrantPermissions) ;  
//将信息写到package.xml、package.list及package-stopped.xml文件  
中  
mSettings.writeLPr () ;  
Runtime.getRuntime () .gc () ;  
mRequiredVerifierPackage=getRequiredVerifierLPr () ;  
.....//PKMS构造函数返回  
}
```

读者可自行研究以上代码中涉及的几个函数，这里不再赘述。

4.3.4 PKMS构造函数总结

从流程角度看，PKMS构造函数的功能还算清晰，无非是扫描XML或APK文件，但是其中涉及的数据结构及它们之间的关系却较为复杂。这里有一些建议供读者参考：

理解PKMS构造函数工作的3个阶段及其各阶段的工作职责。

了解PKMS第二阶段工作中解析APK文件的几个关键步骤，可参考图4-7。

了解重点数据结构的名字和大体功能。

如果对PKMS的分析到此为止，则未免有些太小视它了。下面将分析几个重量级的知识点，期望能带领读者全方位认识PKMS。

4.4 APK Installation分析

本节将分析APK的安装及相关处理流程，它可能比读者想象得要复杂。

马上开始我们的行程，我们的行程从adb install开始。

4.4.1 adb install分析

adb install有多个参数，这里仅考虑最简单的，如adb install frameworktest.apk。adb是一个命令，install是它的参数。此处直接跳到处理install参数的代码：

[-->commandline.c : adb_commandline]

```
int adb_commandline (int argc, char**argv) {  
    ....  
    if (!strcmp (argv[0], "install") ) {  
        ....//调用install_app函数处理  
        return install_app (ttype, serial, argc, argv) ;  
    }  
    ....  
}
```

install_app函数也在commandline.c中定义，代码如下：

[-->commandline.c : install_app]

```
int install_app (transport_type transport, char*serial, int
argc, char**argv)
{
    //要安装的APK现在还在Host机器上，要先把APK复制到手机中。
    //这里需要设置复制目标的目录，如果安装在内部存储中，则目标目录
    //为/data/local/tmp ;
    //如果安装在SD卡上，则目标目录为/sdcard/tmp。
    static const char*const DATA_DEST="/data/local/tmp/%s" ;
    static const char*const SD_DEST="/sdcard/tmp/%s" ;
    const char*where=DATA_DEST ;
    char apk_dest[PATH_MAX] ;
    char verification_dest[PATH_MAX] ;
    char*apk_file ;
    char*verification_file=NULL ;
    int file_arg=-1 ;
    int err ;
    int i ;
    for (i=1 ; i<argc ; i++) {
        if (*argv[i] != '-') {
            file_arg=i ;
            break ;
        }else if (!strcmp (argv[i], "-i") ) {
            i++ ;
        }else if (!strcmp (argv[i], "-s") ) {
            where=SD_DEST ; // -s参数指明该APK安装到SD卡上
        }
    }
    .....
    apk_file=argv[file_arg] ;
    .....
```

```
// 获取目标文件的全路径，如果安装在内部存储中，则目标全路径  
为 /data/local/tmp/ 安装包名，  
// 调用 do_sync_push 将此 APK 传送到手机的目标路径  
err = do_sync_push (apk_file, apk_dest, 1 /* verify APK */);  
..... // ① Android 4.0 新增了一个安装过程中的 Verification 功能，相关知识稍  
后分析  
// ② 执行 pm 命令，这个函数很有意思  
pm_command (transport, serial, argc, argv);  
.....  
cleanup_apk;  
// ③ 在手机中执行 shell rm 命令，删除刚才传送过去的目标 APK 文件。为什么要  
删除呢  
delete_file (transport, serial, apk_dest);  
return err;  
}
```

以上代码中共有3个关键点，分别是：

Android 4.0 新增了 APK 安装过程中的 Verification 的功能。其实就是在安装时，把相关信息发送给指定的 Verification 程序（另外一个是 APK），由它对要安装的 APK 进行检查（Verify）。这部分内容在后面分析 APK 安装时会介绍。目前，标准代码中还没有从事 Verification 工作的 APK。

调用 pm_command 进行安装，这是一个比较有意思的函数，稍后对其进行分析。

安装完后，执行 shell rm 删除刚才传送给手机的 APK 文件。为什么会删除呢？因为 PKMS 在安装

过程中会将该APK复制一份到/data/app目录下，所以/data/local/tmp目录下的对应文件就可以删除了。这部分代码在后面也能见到。

先来分析pm_command命令。为什么说它有意思呢？

4.4.2 pm分析

pm_command代码如下：

[-->commandline.c : pm_command]

```
static int pm_command ( transport_type transport,
char*serial,
int argc, char**argv)
{
char buf[4096];
snprintf (buf, sizeof (buf) , "shell:pm") ;
.....//准备参数
//发送"shell:pm install参数"给手机端的adbd
send_shellcommand (transport, serial, buf) ;
return 0;
}
```

手机端的adbd在收到客户端发来的shell pm命令时会启动一个shell，然后在其中执行pm。pm是什么？为什么可以在shell下执行？

提示 读者可以通过adb shell登录到自己手机，然后执行pm，看看会发现什么。

pm实际上是一个脚本，其内容如下：

[-->pm]

```
#Script to start "pm" on the device, which has a very  
rudimentary  
#shell.  
#  
base=/system  
export CLASSPATH=$base/frameworks/pm.jar  
exec app_process$base/bin com.android.commands.pm.Pm"$@"
```

在编译system.image时，Android.mk中会将该脚本复制到system/bin目录下。从pm脚本的内容来看，它就是通过app_process执行pm.jar包的main函数。在卷I第4章分析Zygote时，已经介绍了app_process是一个Native进程，它通过创建虚拟机启动了Zygote，从而转变为一个Java进程。实际上，app_process还可以通过类似的方法（即先创建Dalvik虚拟机，然后执行某个类的main函数）来转变成其他Java程序。

注意 Android系统中常用的monkeytest、pm、am等（这些都是脚本文件）都是以这种方式启动的，所以严格地说，app_process才是Android Java进程的老祖宗。

下面来分析pm.java, app_process执行的就是它定义的main函数，它相当于Java进程的入口函数，其代码如下：

[-->pm.java]

```
public static void main (String[]args) {  
    new Pm () .run (args) ;//创建一个Pm对象，并执行它的run函数  
}  
//直接分析run函数  
public void run (String[]args) {  
    boolean validCommand=false ;  
    .....  
    //获取PKMS的binder客户端  
    mPm=IPackageManager.Stub.asInterface (  
        ServiceManager.getService ("package") ) ;  
    .....  
    mArgs=args ;  
    String op=args[0] ;  
    mNextArg=1 ;  
    .....//处理其他命令，这里仅考虑install的处理  
    if ("install".equals (op) ) {  
        runInstall () ;  
        return ;  
    }  
    .....  
}
```

接下来分析pm.java的runInstall函数，代码如下：

[-->pm.java : runInstall]

```
private void runInstall () {  
    int installFlags=0 ;  
    String installerPackageName=null ;  
    String opt ;  
    while ( (opt=nextOption () ) !=null) {  
        if (opt.equals ("-l") ) {  
            installFlags|=PackageManager.INSTALL_FORWARD_LOCK ;
```

```
        }else if (opt.equals ("-r") ) {
            installFlags|=PackageManager.INSTALL_REPLACE_EXISTING ;
        }else if (opt.equals ("-i") ) {
            installerPackageName=nextOptionData () ;
            .....//参数解析
        }.....
    }
    final Uri apkURI ;
    final Uri verificationURI ;
    final String apkFilePath=nextArg () ;
    System.err.println ("/tpkg :" +apkFilePath) ;
    if (apkFilePath !=null) {
        apkURI=Uri.fromFile (new File (apkFilePath) ) ;
    }.....
    //获取Verification Package的文件位置
    final String verificationFilePath=nextArg () ;
    if (verificationFilePath !=null) {
        verificationURI=Uri.fromFile (new File (verificationFilePath) ) ;
    }else{
        verificationURI=null ;
    }
    //创建PackageInstallObserver，用于接收PKMS的安装结果
    PackageInstallObserver obs=new PackageInstallObserver () ;
    try{
        //①调用PKMS的installPackageWithVerification完成安装
        mPm.installPackageWithVerification (apkURI, obs,
            installFlags, installerPackageName,
            verificationURI, null) ;
        synchronized (obs) {
            while (! obs.finished) {
                try{
                    obs.wait () ;//等待安装结果
                }.....
            }
            if (obs.result==PackageManager.INSTALL_SUCCEEDED) {
```

```
System.out.println ("Success") ; //安装成功，打印Success  
}.....//安装失败，打印失败原因  
}.....  
}
```

Pm解析参数后，最终通过PKMS的Binder客户端调用installPackageWithVerification以完成后续的安装工作，所以，下面进入PKMS看看安装到底是怎么一回事。

4.4.3 installPackageWithVerification函数分析

installPackageWithVerification的代码如下：

[-->PackageManagerService.java :
installPackageWithVerification]

```
public void installPackageWithVerification (Uri packageURI,  
IPackageInstallObserver observer,  
int flags, String installerPackageName, Uri  
verificationURI,  
ManifestDigest manifestDigest) {  
    //检查客户端进程是否具有安装Package的权限。在本例中，该客户端进程是  
shell  
    mContext.enforceCallingOrSelfPermission (  
        android.Manifest.permission.INSTALL_PACKAGES, null) ;  
    final int uid=Binder.getCallingUid () ;  
    final int filteredFlags ;  
    if (uid==Process.SHELL_UID||uid==0) {  
        ....//如果通过shell pm的方式安装，则增加INSTALL_FROM_ADB标志  
        filteredFlags=flags|PackageManager.INSTALL_FROM_ADB ;  
    }else{  
        filteredFlags=flags&~PackageManager.INSTALL_FROM_ADB ;  
    }  
    //创建一个Message， code为INIT_COPY，将该消息发送给之前在PKMS构造函  
数中  
    //创建的mHandler对象，将在另外一个线程中处理此消息  
    final Message msg=mHandler.obtainMessage (INIT_COPY) ;  
    //创建一个InstallParams，其基类是HandlerParams  
    msg.obj=new InstallParams (packageURI, observer,
```

```
filteredFlags, installerPackageName,  
verificationURI, manifestDigest) ;  
mHandler.sendMessage (msg) ;  
}
```

installPackageWithVerification 函数倒是蛮清闲，简简单单创建几个对象，然后发送 INIT_COPY 消息给 mHandler，就甩手退出了。根据之前在 PKMS 构造函数中介绍的知识可知，mHandler 被绑定到另外一个工作线程（借助 ThreadHandler 对象的 Looper）中，所以该 INIT_COPY 消息也将在那个工作线程中进行处理。我们马上转战到那。

1. INIT_COPY 处理

INIT_COPY 只是安装流程的第一步。先来看相关代码：

[--> PackageManagerService.java :
handleMessage]

```
public void handleMessage (Message msg) {  
try{  
doHandleMessage (msg) ; //调用doHandleMessage函数  
}.....  
}  
void doHandleMessage (Message msg) {  
switch (msg.what) {  
case INIT_COPY : {
```

```
// ① 这里记录的是 params 的基类类型 HandlerParams，实际类型为  
InstallParams
```

```
HandlerParams params= (HandlerParams) msg.obj ;
```

```
//idx为当前等待处理的安装请求的个数
```

```
int idx=mPendingInstalls.size () ;
```

```
if ( ! mBound) {
```

```
/*
```

很多读者可能想不到，APK的安装居然需要使用另外一个APK提供的服务，该服务就是

```
DefaultContainerService，由DefaultContainerService.apk提供，
```

```
下面的connectToService函数将调用bindService来启动该服务
```

```
*/
```

```
if ( ! connectToService ( ) ) {
```

```
return ;
```

```
}else{// 如果已经连上，则以 idx 为索引，将 params 保存到  
mPendingInstalls中
```

```
mPendingInstalls.add (idx, params) ;
```

```
}
```

```
}else{
```

```
mPendingInstalls.add (idx, params) ;
```

```
if (idx==0) {
```

```
//如果安装请求队列之前的状态为空，则表明要启动安装
```

```
mHandler.sendEmptyMessage (MCS_BOUND) ;
```

```
}
```

```
}
```

```
break ;
```

```
}
```

```
.....//后续再分析
```

这里假设之前已经成功启动了 DefaultContainerService（以后简称DCS），并且 idx 为零，所以这是PKMS首次处理安装请求，也就是说，下一个将要处理的是MCS_BOUND消息。

注意 connectToService在调用bindService时会传递一个DefaultContainerConnection类型的对象，以接收服务启动的结果。当该服务成功启动后，此对象的onServiceConnected被调用，其内部也将发送MCS_BOUND消息给mHandler。

2.MCS_BOUND处理

现在，安装请求的状态从INIT_COPY变成MCS_BOUND了，此时的处理流程是怎样的呢？依然在doHandleMessage函数中，直接从对应的case开始，代码如下：

[-->PackageManagerService.java
doHandleMessage]

```
.....//接doHandleMesage中的switch/case
case MCS_BOUND :{
    if (msg.obj !=null) {
        mContainerService= (IMediaContainerService) msg.obj ;
    }
    if (mContainerService==null) {
        .....//如果没法启动该service，则不能安装程序
        mPendingInstalls.clear () ;
    }else if (mPendingInstalls.size () >0) {
        HandlerParams params=mPendingInstalls.get (0) ;
        if (params !=null) {
            //调用params对象的startCopy函数，该函数由基类HandlerParams定义
            if (params.startCopy () ) {
                .....
            }
            if (mPendingInstalls.size () >0) {
```

```
mPendingInstalls.remove (0) ; //删除队列头
}
if (mPendingInstalls.size () ==0) {
if (mBound) {
.....//如果安装请求都处理完了，则需要和Service断绝联系，
//通过发送MSC_UNB消息处理断交请求。读者可自行研究此情况的处理流程
removeMessages (MCS_UNBIND) ;
Message ubmsg=obtainMessage (MCS_UNBIND) ;
sendMessageDelayed (ubmsg, 10000) ;
}
}else{
//如果还有未处理的请求，则继续发送MCS_BOUND消息。
//为什么不通过一个循环来处理所有请求呢
mHandler.sendEmptyMessage (MCS_BOUND) ;
}
}
}
}.....
break ;
```

MCS_BOUND的处理还算简单，就是调用HandlerParams的startCopy函数。在深入分析前，应先认识一下HandlerParams及相关的对象。

(1) HandlerParams和InstallArgs介绍

除了HandlerParams家族外，这里提前请出另外—个家族—InstallArgs家庭，如图4-10所示。

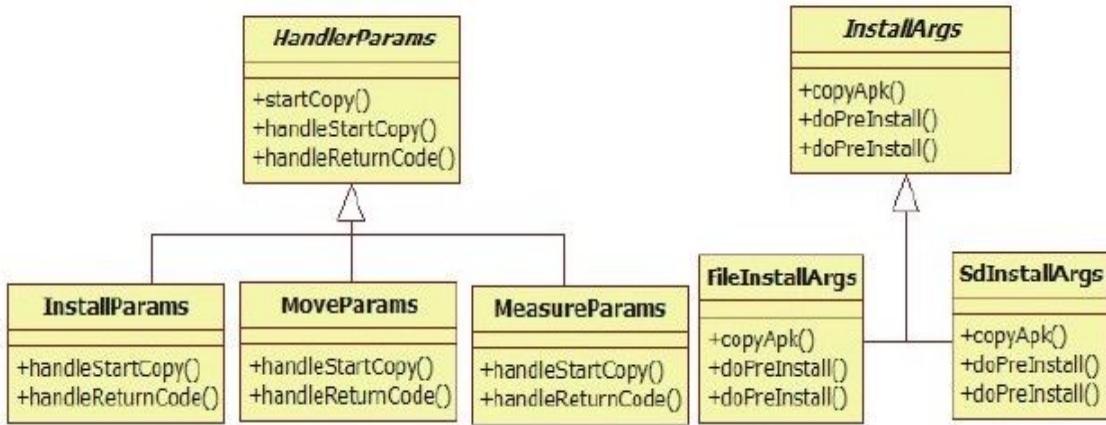


图 4-10 HandlerParams 及 InstallArgs 家族成员

由图4-10可知：

HandlerParams和InstallArgs均为抽象类。

HandlerParams 有 3 个子类，分别是 `InstallParams`、`MoveParams` 和 `MeasureParams`。其中，`InstallParams` 用于处理 APK 的安装，`MoveParams` 用于处理某个已安装APK的“搬家”请求（例如从内部存储移动到 SD 卡上），`MeasureParams` 用于查询某个已安装的APK占据存储空间的大小（例如在设置程序中得到的某个APK使用的缓存文件的大小）。

对于`InstallParams`来说，它还有两个伴儿，即 `InstallArgs` 的 派 生 类 `FileInstallArgs` 和 `SdInstallArgs`。其中，`FileInstallArgs`针对的是安装在内部存储中的APK，而`SdInstallArgs`针对的是那些安装在SD卡上的APK。

本节将讨论用于内部存储安装的FileInstallArgs。

提示 即将出版的“卷 III”将详细介绍MountService和SdInstallArgs的处理流程。

在前面MCS_BOUND的处理中，首先调用InstallParams的startCopy函数，该函数由其基类HandlerParams实现，代码如下：

[-->PackageManagerService.java
HandlerParams.startCopy]

```
final boolean startCopy () {  
    boolean res ;  
    try{  
        //MAX_RETRIES目前为4，表示尝试4次安装，如果还不成功，则认为安装失败  
        if (++mRetries>MAX_RETRIES) {  
            mHandler.sendEmptyMessage (MCS_GIVE_UP) ;  
            handleServiceError () ;  
            return false ;  
        }else{  
            handleStartCopy () ;//①调用派生类的handleStartCopy函数  
            res=true ;  
        }  
    }.....  
    handleReturnCode () ;//②调用派生类的handleReturnCode，返回处理  
    结果  
    return res ;  
}
```

在上述代码中，基类的startCopy将调用子类实现的handleStartCopy和handleReturn-Code函数。下面来看InstallParams是如何实现这两个函数的。

(2) InstallParams分析

先来看派生类InstallParams的handleStartCopy函数，代码如下：

[-->PackageManagerService :
InstallParams.handleStartCopy]

```
public void handleStartCopy () throws RemoteException{
    int ret=PackageManager.INSTALL_SUCCEEDED ;
    final boolean fwdLocked;//本书不考虑fwdLocked的情况
        (flags&PackageManager.INSTALL_FORWARD_LOCK) !=0 ;
    //根据adb install的参数，判断安装位置
    final boolean onSd= (flags &
PackageManager.INSTALL_EXTERNAL) !=0 ;
    final boolean onInt= (flags &
PackageManager.INSTALL_INTERNAL) !=0 ;
    PackageInfoLite pkgLite=null ;
    if (onInt & &onSd) {
        //APK不能同时安装在内部存储空间和SD卡上
        ret=PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION ;
    }else if (fwdLocked & &onSd) {
        //fwdLocked的应用不能安装在SD卡上
        ret=PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION ;
    }else{
        final long lowThreshold ;
        //获取DeviceStorageMonitorService的binder客户端
        final DeviceStorageMonitorService dsm=
            (DeviceStorageMonitorService) ServiceManager.getService (
```

```
DeviceStorageMonitorService.SERVICE) ;  
if (dsm==null) {  
    lowThreshold=0L ;  
}else{  
    //从DSMS查询内部存储空间最小余量， 默认是总存储空间的10%  
    lowThreshold=dsm.getMemoryLowThreshold () ;  
}  
try{  
    //授权DefContainerService URI读权限  
    mContext.grantUriPermission (DEFAULT_CONTAINER_PACKAGE,  
        packageURI, Intent.FLAG_GRANT_READ_URI_PERMISSION) ;  
    //①调用DCS的getMinimalPackageInfo函数， 得到一个PackageInfoLite  
    对象  
    pkgLite=mContainerService.getMinimalPackageInfo  
    (packageURI,  
        flags, lowThreshold) ;  
    }finally.....//撤销URI授权  
    //pkgLite的recommendedInstallLocation成员变量保存该APK推荐的安装  
    路径  
    int loc=pkgLite.recommendedInstallLocation ;  
    if (loc==PackageHelper.RECOMMEND_FAILED_INVALID_LOCATION) {  
        ret=PackageManager.INSTALL_FAILED_INVALID_INSTALL_LOCATION ;  
    }else if.....  
        ....//略去相关内容  
    }else{  
        //②根据DCS返回的安装路径， 还需要调用installLocationPolicy进行检查  
        loc=installLocationPolicy (pkgLite, flags) ;  
        if (!onSd && !onInt) {  
            if (loc==PackageHelper.RECOMMEND_INSTALL_EXTERNAL) {  
                flags|=PackageManager.INSTALL_EXTERNAL ;  
                flags&=PackageManager.INSTALL_INTERNAL ;  
            }.....//处理安装位置为内部存储空间的情况  
        }  
    }  
}
```

```
//③创建一个安装参数对象，对于安装位置为内部存储空间的情况，args的真实  
类型为FileInstallArgs  
    final InstallArgs args=createInstallArgs (this) ;  
    mArgs=args ;  
    if (ret==PackageManager.INSTALL_SUCCEEDED) {  
        final int requiredUid=mRequiredVerifierPackage==null?-1  
        :getPackageUid (mRequiredVerifierPackage) ;  
        if (requiredUid != -1 & & isVerificationEnabled ()) {  
            .....//④待会再讨论verification的处理  
        }else{  
            //⑤调用args的copyApk函数  
            ret=args.copyApk (mContainerService, true) ;  
        }  
    }  
    mRet=ret ;//确定返回值  
}
```

在以上代码中，一共列出了5个关键点，总结如下：

调用DCS的getMinimalPackageInfo函数，将得到一个PackageInfoLite对象，该对象是一个轻量级的用于描述APK的结构（注意区别前面提到的PackageLite，它包含的成员变量和PackageInfoLite类似，只不过PackageLiteInfo继承了Parcelable接口，故它可通过Binder在进程间传递）。在这段代码逻辑中，主要想取得其recommendedInstallLocation的值。此值表示该APK推荐的安装路径。

调用installLocationPolicy检查推荐的安装路径。例如系统Package不允许安装在SD卡上。

createInstallArgs将根据安装位置创建不同的InstallArgs。如果是内部存储空间，则返回FileInstallArgs，否则为SdInstallArgs。

在正式安装前，应先对该APK进行必要的检查。这部分代码后续再介绍。

调用InstallArgs的copyApk。对本例来说，将调用FileInstallArgs的copyApk函数。

下面围绕这5个关键点对handleStartCopy函数展开分析，其中installLocationPolicy和createInstallArgs比较简单，读者可自行研究。

3.handleStartCopy分析

(1) DefaultContainerService分析

首先分析DCS的getMinimalPackageInfo函数，其代码如下：

[-->DefaultContainerService.java :
getMinimalPackageInfo]

```
public PackageInfoLite getMinimalPackageInfo ( final Uri  
fileUri, int flags,
```

```
long threshold) {
    // 注意该函数的参数：fileUri 指向该APK的文件路径（此时还在/data/local/tmp目录下）
    PackageInfoLite ret=new PackageInfoLite () ;
    .....
    String scheme=fileUri.getScheme () ;
    .....
    String archiveFilePath=fileUri.getPath () ;
    DisplayMetrics metrics=new DisplayMetrics () ;
    metrics.setToDefaults () ;
    //调用PackageParser的parsePackageLite解析该APK文件
    PackageParser.PackageLite pkg=
    PackageParser.parsePackageLite (archiveFilePath, 0) ;
    if (pkg==null) {//解析失败
        ....//设置错误值
    }
    return ret ;
}
ret.packageName=pkg.packageName ;
ret.installLocation=pkg.installLocation ;
ret.verifiers=pkg.verifiers ;
//调用recommendAppInstallLocation，取得一个合理的安装位置
ret.recommendedInstallLocation=
    recommendAppInstallLocation      (      pkg.installLocation,
archiveFilePath,
flags, threshold) ;
return ret ;
}
```

APK可在AndroidManifest.xml中声明一个安装位置，不过DCS除了解析该位置外，还需要做进一步检查，这个工作由recommendAppInstallLocation函数完成，代码如下：

[-->DefaultContainerService.java
recommendAppInstallLocation]

```
private      int      recommendAppInstallLocation      (      int
installLocation,
    String archiveFilePath, int flags, long threshold) {
    int prefer;
    boolean checkBoth=false ;
    check_inner :{
        if ( (flags&PackageManager .INSTALL_FORWARD_LOCK) !=0) {
            prefer=PREFER_INTERNAL ;
            break check_inner ; //根据FORWARD_LOCK的情况，只能安装在内部存储
        }else if ( (flags&PackageManager .INSTALL_INTERNAL) !=0) {
            prefer=PREFER_INTERNAL ;
            break check_inner ;
        }
        .....//检查各种情况
    }else
        if
(installLocation==PackageManager .INSTALL_LOCATION_AUTO) {
        prefer=PREFER_INTERNAL ; //一般设定的位置为AUTO， 默认是内部空间
        checkBoth=true ; //设置checkBoth为true
        break check_inner ;
    }
    //查询settings数据库中的secure表， 获取用户设置的安装路径
    int installPreference=
        Settings.System.getInt (getApplicationContext ()
        .getContentResolver () ,
        Settings.Secure.DEFAULT_INSTALL_LOCATION,
        PackageHelper.APP_INSTALL_AUTO) ;
    if (installPreference==PackageHelper.APP_INSTALL_INTERNAL) {
        prefer=PREFER_INTERNAL ;
        break check_inner ;
    }else
        if
(installPreference==PackageHelper.APP_INSTALL_EXTERNAL) {
        prefer=PREFER_EXTERNAL ;
```

```
break check_inner ;
}
prefer=PREFER_INTERNAL ;
}
//判断外部存储空间是否为模拟的，这部分内容我们以后再介绍
final boolean emulated=Environment.getExternalStorageEmulated
() ;
final File apkFile=new File (archiveFilePath) ;
boolean fitsOnInternal=false ;
if (checkBoth||prefer==PREFER_INTERNAL) {
try{//检查内部存储空间是否足够大
fitsOnInternal=isUnderInternalThreshold (apkFile,
threshold) ;
}.....
}
boolean fitsOnSd=false ;
if (!emulated&&(checkBoth||prefer==PREFER_EXTERNAL) ) {
try{//检查外部存储空间是否足够大
fitsOnSd=isUnderExternalThreshold (apkFile) ;
}.....
}
if (prefer==PREFER_INTERNAL) {
if (fitsOnInternal) {//返回推荐安装路径为内部存储空间
return PackageHelper.RECOMMEND_INSTALL_INTERNAL ;
}
}else if (!emulated&&prefer==PREFER_EXTERNAL) {
if (fitsOnSd) {//返回推荐安装路径为外部存储空间
return PackageHelper.RECOMMEND_INSTALL_EXTERNAL ;
}
}
if (checkBoth) {
if (fitsOnInternal) {//如果内部存储空间满足条件，先返回内部存储空间
return PackageHelper.RECOMMEND_INSTALL_INTERNAL ;
}else if (!emulated&&fitsOnSd) {
return PackageHelper.RECOMMEND_INSTALL_EXTERNAL ;
}
}
```

```
        }  
        .....//到此，前几个条件都不满足，此处将根据情况返回一个明确的错误值  
        return PackageHelper.RECOMMEND_FAILED_INSUFFICIENT_STORAGE;  
    }  
}
```

DCS的getMinimalPackageInfo函数为了得到一个推荐的安装路径做了不少工作，其中，各种安装策略交叉影响。这里总结一下相关的知识点：

APK在AndroidManifest.xml中设置的安装点默认为AUTO，在具体对应时倾向内部存储空间。

用户在Settings数据库中设置的安装位置。

检查外部存储或内部存储是否有足够空间。

(2) InstallArgs的copyApk函数分析

至此，我们已经得到了一个合适的安装位置（先略过Verification这一步）。下一步工作就由copyApk来完成。根据函数名可知，该函数将完成APK文件的复制工作，此中会有蹊跷吗？来看下面的代码。

[-->PackageManagerService.java :
InstallArgs.copyApk]

```
int copyApk ( IMediaContainerService imcs, boolean temp )  
throws RemoteException{
```

```
if (temp) {  
/*
```

本例中temp参数为true，createCopyFile将在/data/app目录下创建一个临时文件。

临时文件名为vmdl-随机数.tmp。为什么会用这样的文件名呢？

因为PKMS通过Linux的inotify机制监控了/data/app目录，如果新复制生成的文件名后缀

为.apk，将触发PKMS扫描。为了防止发生这种情况，这里复制生成的文件才有了如此奇怪的名字

```
*/  
createCopyFile () ;  
}  
File codeFile=new File (codeFileName) ;  
.....  
ParcelFileDescriptor out=null;  
try{  
out=ParcelFileDescriptor.open (codeFile,  
ParcelFileDescriptor.MODE_READ_WRITE) ;  
}.....  
int ret=PackageManager.INSTALL_FAILED_INSUFFICIENT_STORAGE ;  
try{  
mContext.grantUriPermission (DEFAULT_CONTAINER_PACKAGE,  
packageURI, Intent.FLAG_GRANT_READ_URI_PERMISSION) ;  
// 调用DCS的copyResource，该函数将执行复制操作，最终结果  
是/data/local/tmp  
//下的APK文件被复制到/data/app目录下，文件名也被换成vmdl-随机数.tmp  
ret=imcs.copyResource (packageURI, out) ;  
}finally{  
....//关闭out，撤销URI授权  
}  
return ret ;  
}
```

关于临时文件，这里提供一个示例，如图4-11所示。

```
root@innost:/thunderst/disk1/Android-4.0# adbs ls /data/app  
com.thundersoft.test-2.apk  
vmdl-524884471.tmp
```

图 4-11 createCopyFile生成的临时文件

由图4-11可知，/data/app目录下有两个文件，第一个是正常的APK文件，第二个是createCopyFile生成的临时文件。

4.handleReturnCode分析

在 HandlerParams 的 startCopy 函数中，handleStartCopy 执行完之后，将调用 handleReturnCode开展后续工作，代码如下：

[-->PackageManagerService.java :
InstallParams.HandleParams]

```
void handleReturnCode () {  
    if (mArgs !=null) {  
        // 调用 processPendingInstall 函数， mArgs 指向之前创建的  
        FileInstallArgs对象  
        processPendingInstall (mArgs, mRet) ;  
    }  
}
```

[-->PackageManagerService.java :
processPendingInstall]

```
private void processPendingInstall (final InstallArgs args,
```

```
final int currentStatus) {
    //向mHandler中抛一个Runnable对象
    mHandler.post (new Runnable () {
        public void run () {
            mHandler.removeCallbacks (this) ;
            //创建一个PackageInstalledInfo对象,
            PackageInstalledInfo res=new PackageInstalledInfo () ;
            res.returnCode=currentStatus ;
            res.uid=-1 ;
            res.pkg=null ;
            res.removedInfo=new PackageRemovedInfo () ;
            if (res.returnCode==PackageManager.INSTALL_SUCCEEDED) {
                //①调用FileInstallArgs的doPreInstall
                args.doPreInstall (res.returnCode) ;
                synchronized (mInstallLock) {
                    //②调用installPackageLI进行安装
                    installPackageLI (args, true, res) ;
                }
                //③调用FileInstallArgs的doPostInstall
                args.doPostInstall (res.returnCode) ;
            }
            final boolean update=res.removedInfo.removedPackage !=null ;
            boolean doRestore= (! update&&res.pkg !=null &&
            res.pkg.applicationInfo.backupAgentName !=null) ;
            int token;//计算一个ID号
            if (mNextInstallToken<0) mNextInstallToken=1 ;
            token=mNextInstallToken++ ;
            //创建一个PostInstallData对象
            PostInstallData data=new PostInstallData (args, res) ;
            //保存到mRunningInstalls结构中, 以token为key
            mRunningInstalls.put (token, data) ;
            if ( res.returnCode==PackageManager.INSTALL_SUCCEEDED & &
            doRestore)
            {
                .....//备份恢复的情况暂时不考虑
            }
        }
    });
}
```

```
if (!doRestore) {  
    //④抛一个POST_INSTALL消息给mHandler进行处理  
    Message msg=mHandler.obtainMessage ( POST_INSTALL, token ,  
0) ;  
    mHandler.sendMessage (msg) ;  
}  
}  
}  
}
```

由上面代码可知，handleReturnCode主要做了4件事情：

调用InstallArgs的doPreInstall函数，在本例中是FileInstallArgs的doPreInstall函数。

调用PKMS的installPackageLI函数进行APK安装，该函数内部将调用InstallArgs的doRename对临时文件进行改名。另外，还需要扫描此APK文件。此过程和4.3.2节中的内容类似。至此，该APK中的私有财产就全部被登记到PKMS内部进行保存了。

调用InstallArgs的doPostInstall函数，在本例中是FileInstallArgs的doPostInstall函数。

此时，该APK已经安装完成（不论失败还是成功），继续向mHandler抛送一个POST_INSTALL消息，该消息携带一个token，通

过它可从mRunningInstalls数组中取得一个PostInstallData对象。

提示 对于FileInstallArgs来说，其doPreInstall和doPostInstall都比较简单，读者可自行阅读相关代码。另外，读者也可自行研究PKMS的installPackageLI函数。

这里介绍一下FileInstallArgs的doRename函数，它的功能是为临时文件改名，最终的文件的名称一般为“包名-数字.apk”。其中，数字是一个index，从1开始。读者可参考图4-11中/data/app目录下第一个文件的文件名。

5.POST_INSTALL处理

现在需要处理POST_INSTALL消息，因为adb install还等着安装结果呢。相关代码如下：

[-->PackageManagerService.java :
doHandleMessage]

```
.....//接前面的switch/case
case POST_INSTALL : {
    PostInstallData data=mRunningInstalls.get (msg.arg1) ;
    mRunningInstalls.delete (msg.arg1) ;
    boolean deleteOld=false ;
    if (data !=null) {
        InstallArgs args=data.args ;
        PackageInstalledInfo res=data.res ;
```

```
if (res.returnCode==PackageManager.INSTALL_SUCCEEDED) {  
    res.removedInfo.sendBroadcast (false, true) ;  
    Bundle extras=new Bundle (1) ;  
    extras.putInt (Intent.EXTRA_UID, res.uid) ;  
    final boolean update=res.removedInfo.removedPackage !=null ;  
    if (update) {  
        extras.putBoolean (Intent.EXTRA_REPLACE, true) ;  
    }  
    //发送PACKAGE_ADDED广播  
    sendPackageBroadcast (Intent.ACTION_PACKAGE_ADDED,  
        res.pkg.applicationInfo.packageName, extras, null, null) ;  
    if (update) {  
        /*  
        如果是APK升级，那么发送PACKAGE_REPLACE和MY_PACKAGE_REPLACED广播。  
        二者不同之处在于PACKAGE_REPLACE将携带一个extra信息  
        */  
        Runtime.getRuntime () .gc () ;  
        if (deleteOld) {  
            synchronized (mInstallLock) {  
                //调用FileInstallArgs的doPostDeleteLI进行资源清理  
                res.removedInfo.args.doPostDeleteLI (true) ;  
            }  
        }  
        if (args.observer !=null) {  
            try{  
                //通知pm安装的结果  
                args.observer.packageInstalled (res.name, res.returnCode) ;  
            }.....  
        }break ;  
    }  
}
```

4.4.4 APK安装流程总结

没想到APK的安装流程竟然如此复杂，其目的无非是让APK中的“私人财产”公有化。相比之下，在PKMS构造函数中进行公有化改造就非常简单。另外，如果考虑安装到SD卡的处理流程，那么APK的安装将会更加复杂。

这里要总结APK安装过程中的几个重要步骤，如图4-12所示。

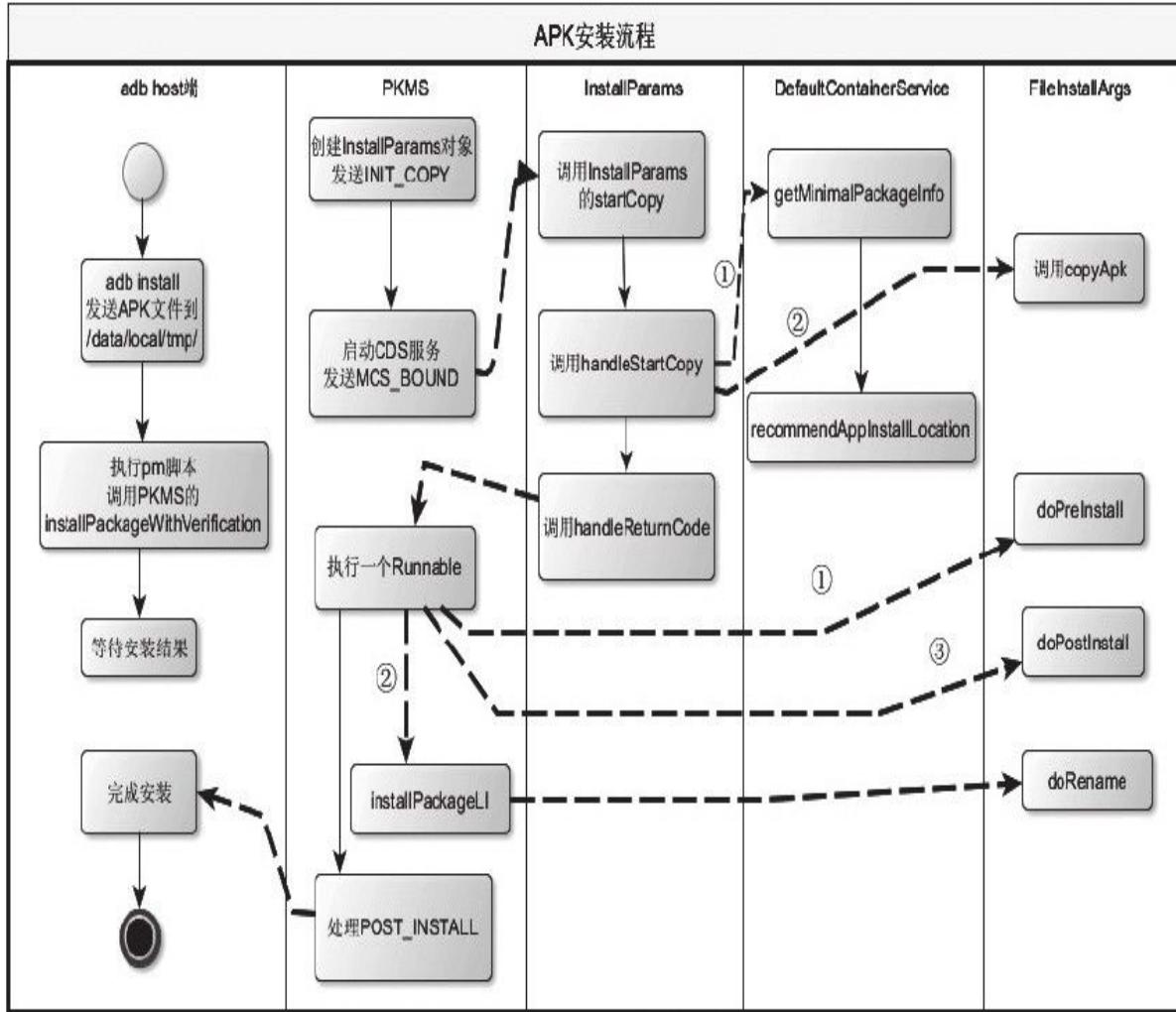


图 4-12 APK安装流程

图4-12中列出以下内容：

安装APK到内部存储空间这一工作流程涉及的主要对象包括：PKMS、Default-ContainerService、InstallParams 和 FileInstallArgs。

此工作流程中每个对象涉及的关键函数。

对象之间的调用通过虚线表达，调用顺序通过①②③等标明。

4.4.5 Verification介绍

Verification功能的出现将打乱图4-12所示的工作流程，所以这部分内容要放在最后来介绍。其代码在InstallParams的handleStartCopy中，如下所示：

[-->PackageManagerService.java :
InstallParams.handleStartCopy]

```
.....//此处已经获得了合适的安装位置
final InstallArgs args=createInstallArgs (this) ;
mArgs=args ;
if (ret==PackageManager.INSTALL_SUCCEEDED) {
final int requiredUid=mRequiredVerifierPackage==null?-1
:getId (mRequiredVerifierPackage) ;
if (requiredUid != -1 & & isVerificationEnabled ()) {
//创建一个Intent，用于查找满足条件的广播接收者
final Intent verification=new
Intent (Intent.ACTION_PACKAGE_NEEDS_VERIFICATION) ;
verification.setDataAndType (packageURI,
PACKAGE_MIME_TYPE) ;
verification.addFlags
(Intent.FLAG_GRANT_READ_URI_PERMISSION) ;
//查找满足Intent条件的广播接收者
final List<ResolveInfo>receivers=queryIntentReceivers (
verification, null,
PackageManager.GET_DISABLED_COMPONENTS) ;
//verificationId为当前等待Verification的安装包个数
final int verificationId=mPendingVerificationToken++ ;
```

```
//设置Intent的参数，例如要校验的包名
verification.putExtra
(PackageManager.EXTRA_VERIFICATION_ID,
VerificationId) ;
verification.putExtra (
PackageManager.EXTRA_VERIFICATION_INSTALLER_PACKAGE,
installerPackageName) ;
verification.putExtra (
PackageManager.EXTRA_VERIFICATION_INSTALL_FLAGS, flags) ;
if (verificationURI !=null) {
verification.putExtra
(PackageManager.EXTRA_VERIFICATION_URI,
verificationURI) ;
}
final PackageVerificationState verificationState=new
PackageVerificationState (requiredUid, args) ;
// 将 上 面 创 建 的 PackageVerificationState 保 存 到
mPendingVerification中
mPendingVerification.append ( verificationId,
verificationState) ;
//筛选符合条件的广播接收者
final List<ComponentName>sufficientVerifiers=
matchVerifiers (pkgLite, receivers, verificationState) ;
if (sufficientVerifiers !=null) {
final int N=sufficientVerifiers.size () ;
.....
for (int i=0 ;i<N ;i++) {
final ComponentName
verifierComponent=sufficientVerifiers.get (i) ;
final Intent sufficientIntent=new Intent (verification) ;
sufficientIntent.setComponent (verifierComponent) ;
//向校验包发送广播
mContext.sendBroadcast (sufficientIntent) ;
}
}}
```

```
//除此之外，如果在执行adb install的时候指定了校验包，则需要向其单独发送校验广播
final ComponentName requiredVerifierComponent=
matchComponentForVerifier (mRequiredVerifierPackage,
receivers) ;
if (ret==PackageManager.INSTALL_SUCCEEDED
&&mRequiredVerifierPackage !=null) {
verification.setComponent (requiredVerifierComponent) ;
mContext.sendOrderedBroadcast (verification,
android.Manifest.permission.PACKAGE_VERIFICATION_AGENT,
new BroadcastReceiver () {
//调用sendOrderdBroadcast，并传递一个BroadcastReceiver，该对象将在
//广播发送的最后被调用。读者可参考sendOrderdBroadcast的文档说明
public void onReceive (Context context, Intent intent) {
final Message msg=mHandler.obtainMessage (
CHECK_PENDING_VERIFICATION) ;
msg.arg1=verificationId ;
//设置一个超时执行时间，该值来自Settings数据库的secure表，默认为60秒
mHandler.sendMessageDelayed ( msg, getVerificationTimeout
() ) ;
}
}, null, 0, null, null) ;
mArgs=null;
}
}.....//不用做Verification的流程
```

PKMS的Verification工作其实就是收集安装包的信息，然后向对应的校验者发送广播。但遗憾的是，当前Android中还没有能处理Verification的组件。

另外，该组件处理完Verification后，需要调用PKMS的verifyPendingInstall函数，以通知校验结果。

提示 Verification的目的主要是在APK安装时先触发校验程序对该APK进行检查，只有检查通过才能进行真正的安装。这部分代码比较简单。不过目前源码中还没有可进行Verification的程序，读者不妨自己写一个小例子测试一番。

4.5 queryIntentActivities分析

PKMS除了负责Android系统中Package的安装、升级、卸载外，还有一项很重要的职责，就是对外提供统一的信息查询功能，其中包括查询系统中匹配某Intent的Activities、BroadCastReceivers或Services等。本节将以查询匹配某Intent的Activities为例，介绍PKMS在这方面提供的服务。

正式分析queryIntentActivities之前，先来认识一下Intent及IntentFilter。

4.5.1 Intent及IntentFilter介绍

1.Intent介绍

Intent中文是“意图”的意思，它是Android系统中一个很重要的概念，其基本思想来源于对日常生活及行为的高度抽象。我们结合用人单位招聘的例子介绍Intent背后的思想。

假设某用人单位现需招聘人员完成某项工作。该单位首先应将其需求发给猎头公司。

猎头公司从其内部的信息库中查找合适的人选。猎头公司除了考虑用人单位的需求外，还需要考虑求职者本身的要求，例如有些求职者对工作地点、加班等有要求。

二者匹配后，就会得到满足要求的求职者。之后用人单位将工作交给满足条件的人员来完成。

在现实生活中，用人单位还需和求职者进行一系列其他交互工作，例如面试、签订合同之类。但是从完成工作的角度来看，只要把工作任务交给满足要求的求职者去做即可，中间的一系列行为和工作任务本身没有太大关系。因此，Android并未将这部分内容抽象化。

意图是一个非常抽象的概念，在编码设计中，如何将它实例化呢？Android系统明确指定的一个Intent可由两方面属性来衡量。

主要属性：包括Action和Data。其中Action用于表示该Intent所表达的动作意图、Data用于表示该Action所操作的数据。

次要属性：包括Category、Type、Component和Extras。其中Category表示类别，Type表示数据的MIME类型，Component可用于指定特定的Intent

响应者（例如指定广播接收者为某Package的某个BroadcastReceiver），Extras用于承载其他的信息。

如果Intent是一份用工需求表，那么上述信息就是该表的全部可填项。在实际使用中，可根据需要填写该表的内容。

当这份需求表传给猎头公司后，猎头公司就根据该表所填写的内容，进一步对Intent进行分类。

Explicit Intent：这类Intent明确指明了要找哪些人。在代码中通过setComponent或setClass来锁定目标对象。处理这种Intent，工作很轻松。

Implicit Intent：这一类Intent只标明了作品内容，而没有指定具体人名。对于这类意图，猎头公司不得不做一系列复杂的工作才能找到满足用人单位需求的人才。

Intent就先介绍到这里。下面来看在这次“招聘”过程中“求职者”填写的信息。

2.IntentFilter介绍

“求职方”需要填写IntentFilter来表达自己的诉求。Android规定了3项内容。

Action：“求职方”支持的Intent动作（和Intent中的Action对应）。

Category：“求职方”支持的Intent种类（和Intent的Category对应）。

Data：求职方支持的Intent数据（和Intent的Data对应，包括URI和MIME类型）。

至此，猎头公司已有了需求，现在又有了求职者的信息，马上要做的工作就是匹配查询。在Android中，该工作被称为Intent Resolution。由于现在及未来人才都是最宝贵的资源，因此猎头公司在做匹配工作时，将以Intent Filter列出的3项内容为参考标准，具体步骤如下：

首先匹配IntentFilter的Action，如果Intent设置的Action不满足IntentFilter的Action，则匹配失败。如果IntentFilter未设定Action，则匹配成功。

然后检查IntentFilter的Category，匹配方法同Action的匹配，唯一有些例外的是Category为CATEGORY_DEFAULT的情况。

最后检查Data。Data的匹配过程比较烦琐，因为它和IntentFilter设置的Data内容有关，见接下来的介绍。

IntentFilter中的Data可以包括两个内容。

URI : 完整格式为“scheme : //host : port/path”，包含4个部分，scheme、host、port和path。其中host和port合起来标识URI authority，用于指明服务器的网络地址（IP加端口号）。由于uri最多可包含4个部分，因此要根据情况相应部分做匹配检查。

Date type : 指定数据的MIME类型。

要特别注意的是，uri中也可以携带数据的类型信息，所以在匹配过程中，还需要考虑uri中指定的数据类型。

提示 关于具体的匹配流程，请读者务必阅读 [SDK docs/guide/topics/intents/intents-filters.html](#) 中的说明。

4.5.2 Activity信息的管理

前面在介绍PKMS扫描APK时提到，PKMS将解析得到的package私有的Activity信息加入到自己的数据结构mActivities中保存。下面先来看以下相关代码：

[--> PackageManagerService.java :
scanPackageLI]

```
.....//此时APK文件已经解析完成  
N=pkg.activities.size () ; //取出该APK中包含的Activities信息  
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") ; //①加到mActivities中  
    保存  
}
```

上面的代码中有两个比较重要的数据结构，如图4-13所示。

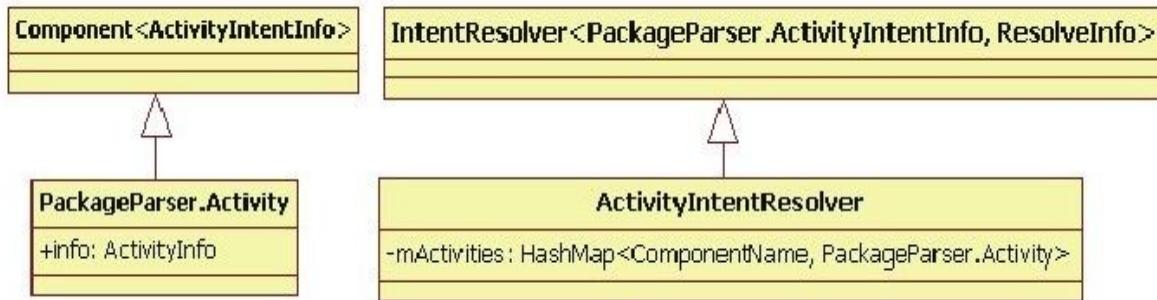


图 4-13 相关数据结构示意图

结合代码，由图4-13可知：

mActivities 为 ActivityIntentResolver 类型，是 PKMS 的成员变量，用于保存系统中所有与 Activity 相关的信息。此数据结构内部有一个 mActivities 变量，它以 Componet-Name 为 key，保存 PackageParser.Activity 对象。

从 APK 中解析得到的所有和 Activity 相关的信息（包括在 XML 中声明的 IntentFilter 标签）都由 PackageParser.Activity 来保存。

前面代码中调用 addActivity 函数完成了私有信息的公有化。 addActivity 函数的代码如下：

[--> PackageManagerService.java :
ActivityIntentResolver.addActivity]

```

public final void addActivity ( PackageParser.Activity a,
String type) {

```

```
    final boolean systemApp=isSystemApp
(a.info.applicationInfo) ;
    //将Component和Activity保存到mActivities中
    mActivities.put (a.getComponentName () , a) ;
    final int NI=a.intents.size () ;
    for (int j=0 ; j<NI ; j++) {
        //ActivityIntentInfo存储的就是XML中声明的IntentFilter信息
        PackageParser.ActivityIntentInfo intent=a.intents.get (j) ;
        if ( ! systemApp & & intent.getPriority () >0 &
&"activity".equals (type) ) {
            //非系统APK的priority必须为0。后续分析中将介绍priority的作用
            intent.setPriority (0) ;
        }
        addFilter (intent) ;//接下来将分析这个函数
    }
}
```

下面来分析addFilter函数，这里涉及较多的复杂的数据结构，代码如下：

[-->IntentResolver.java
IntentResolver.addFilter]

```
public void addFilter (F f) {
    .....
    mFilters.add (f) ;//mFilters保存所有IntentFilter信息
    //除此之外，为了加快匹配工作的速度，还需要分类保存IntentFilter信息
    //下边register_xxx函数的最后一个参数用于打印信息
    int numS=register_intent_filter (f, f.schemesIterator () ,
    mSchemeToFilter, "Scheme :") ;
    int numT=register_mime_types (f, "Type :") ;
    if (numS==0 & & numT==0) {
        register_intent_filter (f, f.actionsIterator () ,
        mActionToFilter, "Action :") ;
```

```
        }
        if (numT !=0) {
            register_intent_filter (f, f.actionsIterator () ,
mTypedActionToFilter, "TypedAction :") ;
        }
    }
```

正如代码注释中所说，为了加快匹配工作的速度，这里使用了泛型编程并定义了较多的成员变量。下面总结一下这些变量的作用（注意，除mFilters为HashSet<F>类型外，其他成员变量的类型都是HashMap<String, ArrayList<F>>，其中F为模板参数）。

mSchemeToFilter：用于保存uri中与scheme相关的IntentFilter信息。

mActionToFilter：用于保存仅设置Action条件的IntentFilter信息。

mTypedActionToFilter：用于保存既设置了Action又设置了Data的MIME类型的IntentFilter信息。

mFilters：用于保存所有IntentFilter信息。

mWildTypeToFilter：用于保存设置了Data类型类似“image/*”的IntentFilter，但是设置MIME类型类似“Image/jpeg”的不算在此类。

mTypeToFilter：除了包含mWildTypeToFilter外，还包含那些指明了Data类型为确定参数的IntentFilter信息，例如“image/*”和“image/jpeg”等都包含在mTypeToFilter中。

mBaseTypeToFilter：包含MIME中Base类型的IntentFilter信息，但不包括Sub type为“*”的IntentFilter。

不妨举个例子来说明这些变量的用法。

假设，在XML中声明一个IntentFilter，代码如下：

```
<intent-filter android:label="test">
<action android:name="android.intent.action.VIEW"/>
<data android:mimeType="audio/*" android:scheme="http"
</intent-filter>
```

那么：

在mTypedActionToFilter中能够以“android.intent.action.VIEW”为key找到该IntentFilter。

在mWildTypeToFilter和mTypeToFilter中能够以“audio”为key找到该IntentFilter。

在mSchemeToFilter中能够以“http”为key找到该IntentFilter。

下面来分析Intent匹配查询工作。

4.5.3 Intent匹配查询分析

1.客户端查询

客户端通过ApplicationPackageManager输出的queryIntentActivities函数向PKMS发起一次查询请求，代码如下：

[-->ApplicationPackageManager.java :
queryIntentActivities]

```
public List<ResolveInfo>queryIntentActivities ( Intent  
intent, int flags) {  
    try{  
        return mPM.queryIntentActivities (  
            intent, //下面这句话很重要  
            intent.resolveTypeIfNeeded ( mContext.getContentResolver  
( ) ,  
            flags) ;  
    }.....  
}
```

如果Intent的Data包含一个uri，那么就需要查询该uri的提供者（即Content-Provider）以取得该数据的数据类型。读者可自行阅读resolveTypeIfNeeded函数的代码。

另外，flags参数目前有3个可选值，分别是MATCH_DEFAULT_ONLY、GET_INTENT_FILTERS和GET_RESOLVED_FILTER。详细信息读者可查询SDK相关文档。

下面来看PKMS对匹配查询的处理。

2.queryIntentActivities分析

该函数代码如下：

[-->PackageManagerService.java :
queryIntentActivities]

```
public List<ResolveInfo>queryIntentActivities ( Intent  
intent,  
String resolvedType, int flags) {  
final ComponentName comp=intent.getComponent () ;  
if (comp !=null) {  
//Explicit的Intent，直接根据component得到对应的ActivityInfo  
final List<ResolveInfo>list=new ArrayList<ResolveInfo>  
(1) ;  
final ActivityInfo ai=getActivityInfo (comp, flags) ;  
if (ai !=null) {  
final ResolveInfo ri=new ResolveInfo () ;  
//ResovlerInfo的activityInfo指向查询得到的ActivityInfo  
ri.activityInfo=ai ;  
list.add (ri) ;  
}  
return list ;  
}
```

```
synchronized (mPackages) {  
    final String pkgName=intent.getPackage () ;  
    if (pkgName==null) {  
        //Implicit Intents, 我们重点分析此情况  
        return mActivities.queryIntent ( intent, resolvedType,  
            flags) ;  
    }  
    //Intent指明了PackageName, 比Explicit Intents情况差一点  
    final PackageParser.Package pkg=mPackages.get (pkgName) ;  
    if (pkg !=null) {  
        //其实是从该Package包含的Activities中进行匹配查询  
        return mActivities.queryIntentForPackage ( intent,  
            resolvedType,  
            flags, pkg.activities) ;  
    }  
    return new ArrayList<ResolveInfo> () ;  
}
```

上边代码分3种情况：

如果Intent指明了Component，则直接查询该Component对应的ActivityInfo。

如果Intent指明了Package名，则根据Package名找到该Package，然后再从该Package包含的Activities中进行匹配查询。

如果上面条件都不满足，则需要在全系统范围内进行匹配查询，这就是queryIntent的工作。

queryIntent函数的代码如下：

```
public List<ResolveInfo>queryIntent ( Intent intent, String resolvedType,
int flags) {
mFlags=flags ;
//调用基类的queryIntent函数
return super.queryIntent (intent, resolvedType,
(flags&PackageManager.MATCH_DEFAULT_ONLY) !=0) ;
}
```

[-->IntentResolver.java : queryIntent]

```
public List<R>queryIntent ( Intent intent, String resolvedType,
boolean defaultOnly) {
String scheme=intent.getScheme () ;
ArrayList<R>finalList=new ArrayList<R> () ;
//最多有4轮匹配工作要做
ArrayList<F>firstTypeCut=null ;
ArrayList<F>secondTypeCut=null ;
ArrayList<F>thirdTypeCut=null ;
ArrayList<F>schemeCut=null ;
//下面将设置各轮校验者
if (resolvedType !=null) {
int slashpos=resolvedType.indexOf ('/') ;
if (slashpos>0) {
final String baseType=resolvedType.substring (0, slashpos) ;
if (!baseType.equals ("*") ) {
if (resolvedType.length () !=slashpos+2
||resolvedType.charAt (slashpos+1) !='*') {
firstTypeCut=mTypeToFilter.get (resolvedType) ;
secondTypeCut=mWildTypeToFilter.get (baseType) ;
}.....//略去一部分内容
}
}
if (scheme !=null) {
```

```
schemeCut=mSchemeToFilter.get (scheme) ;
}
if (resolvedType==null & & scheme==null & & intent.getAction () !=null)
{
//看来action的filter优先级最低
firstTypeCut=mActionToFilter.get (intent.getAction () ) ;
}
//FastImmutableArraySet是一种特殊的数据结构，用于保存该Intent中携带的
//Category相关的信息
FastImmutableArraySet<String>
categories=getFastIntentCategories (intent) ;
if (firstTypeCut !=null) {
//第一轮匹配查询
buildResolveList (intent, categories, debug, defaultOnly,
resolvedType, scheme, firstTypeCut, finalList) ;
}
if (secondTypeCut !=null) {
buildResolveList (intent, categories, debug, defaultOnly,
resolvedType, scheme, secondTypeCut, finalList) ;
}
if (thirdTypeCut !=null) {
buildResolveList (intent, categories, debug, defaultOnly,
resolvedType, scheme, thirdTypeCut, finalList) ;
}
if (schemeCut !=null) {
//生成符合schemeCut条件的finalList
buildResolveList (intent, categories, debug, defaultOnly,
resolvedType, scheme, schemeCut, finalList) ;
}
//将匹配结果按Priority的大小排序
sortResults (finalList) ;
return finalList ;
}
```

在以上代码中设置了最多4轮匹配关卡，然后逐一执行匹配工作。具体的匹配工作由buildResolveList完成。匹配无非是一项查找而已，此处就不再深究细节了，建议读者在研究代码时以目的为导向，不宜深究其中的数据结构。

4.5.4 queryIntentActivities总结

本节分析了queryIntentActivities函数的实现，其功能很简单，就是进行Intent匹配查询。一路走来，相信读者也感觉到旅程并不轻松，主要原因是涉及的数据结构较多，让人有些头晕。这里，再次建议读者不要在数据结构上花太多时间，最好结合SDK中的文档说明来分析相关代码。

4.6 installd及UserManager介绍

4.6.1 installd介绍

在前面对PKMS构造函数分析时介绍过一个Installer类型的对象mInstaller，它通过socket和后台服务installd交互，以完成一些重要操作。这里先回顾一下PKMS中mInstaller的调用方法：

```
mInstaller=new Installer () ;//创建一个Installer对象  
//对某个APK文件进行dex优化  
mInstaller.dexopt (paths[i], Process.SYSTEM_UID, true) ;  
//扫描完系统Package后，调用moveFiles函数  
mInstaller.moveFiles () ;  
//当存储空间不足时，调用该函数清理存储空间  
mInstaller.freeCache (freeStorageSize) ;
```

Installer的种种行为都和其背后的installd有关。下面来分析installd。

1.installd概貌

installd是一个native进程，代码非常简单，其功能就是启动一个socket，然后处理来自Installer的命令，其代码如下：

[-->installd.c : main]

```
int main (const int argc, const char*argv[]) {
char buf[BUFFER_MAX] ;
struct sockaddr addr ;
socklen_t alen ;
int lsocket, s, count ;
//初始化全局变量，如果失败则退出
if (initialize_globals () <0) {
LOGE ("Could not initialize globals ; exiting.\n") ;
exit (1) ;
}
if (initialize_directories () <0) {
LOGE ("Could not create directories ; exiting.\n") ;
exit (1) ;
}
.....
lsocket=android_get_control_socket (SOCKET_PATH) ;
listen (lsocket, 5) ;
fcntl (lsocket, F_SETFD, FD_CLOEXEC) ;
for (;;) {
alen=sizeof (addr) ;
s=accept (lsocket, &addr, &alen) ;
fcntl (s, F_SETFD, FD_CLOEXEC) ;
for (;;) {
unsigned short count ;
readx (s, &count, sizeof (count) ) ;
//执行installer发出的命令，具体解释见下文
execute (s, buf) ;
}
close (s) ;
}
return 0 ;
}
```

installd支持的命令及参数信息都保存在数据结构cmds中，代码如下：[-->installd.c]

```
struct cmdinfo cmds[]={{//第二个变量是参数个数，第三个参数是命令响应
    函数
    {"ping", 0, do_ping},
    {"install", 3, do_install},
    {"dexopt", 3, do_dexopt},
    {"movedex", 2, do_move_dex},
    {"rmdex", 1, do_rm_dex},
    {"remove", 2, do_remove},
    {"rename", 2, do_rename},
    {"freecache", 1, do_free_cache},
    {"rmcache", 1, do_rm_cache},
    {"protect", 2, do_protect},
    {"getsize", 4, do_get_size},
    {"rmuserdata", 2, do_rm_user_data},
    {"movefiles", 0, do_movefiles},
    {"linklib", 2, do_linklib},
    {"unlinklib", 1, do_unlinklib},
    {"mkuserdata", 3, do_mk_user_data},
    {"rmuser", 1, do_rm_user},
}};
```

下面来分析相关的几个命令。

2.dexOpt命令分析

PKMS在需要对一个APK或Jar包做dex优化时，会发送dexopt命令给installd，相应的处理函数为do_dexopt，代码如下：

[-->installd.c : do_dexopt]

```
static int do_dexopt (char**arg, char reply[REPLY_MAX])
{
    return dexopt (arg[0], atoi (arg[1]) , atoi (arg[2]) ) ;
}
```

[-->commands.c : dexopt]

```
int dexopt (const char*apk_path, uid_t uid, int is_public)
{
    struct utimbuf ut;
    struct stat apk_stat, dex_stat;
    char dex_path[PKG_PATH_MAX];
    char dexopt_flags[PROPERTY_VALUE_MAX];
    char*end;
    int res, zip_fd=-1, odex_fd=-1;
    .....
    //取出系统级的dexopt_flags参数
    property_get ("dalvik.vm.dexopt-flags", dexopt_flags, "") ;
    strcpy (dex_path, apk_path) ;
    end=strrchr (dex_path, '.') ;
    if (end !=NULL) {
        strcpy (end, ".odex") ;
        if (stat (dex_path, &dex_stat) ==0) {
            return 0;
        }
    }
    //得到一个字符串，用于描述dex文件名，位于/data/dalvik-cache目录下
    if (create_cache_path (dex_path, apk_path) ) {
        return-1;
    }
    memset (&apk_stat, 0, sizeof (apk_stat) ) ;
    stat (apk_path, &apk_stat) ;
```

```
zip_fd=open (apk_path, O_RDONLY, 0) ;  
.....  
unlink (dex_path) ;  
odex_fd=open (dex_path, O_RDWR|O_CREAT|O_EXCL, 0644) ;  
.....  
pid_t pid ;  
pid=fork () ;  
if (pid==0) {  
....//uid设置  
//创建一个新进程，然后exec dexopt进程进行dex优化  
run_dexopt (zip_fd, odex_fd, apk_path, dexopt_flags) ;  
exit (67) ;  
}else{  
//installd将等待dexopt完成优化工作  
res=wait_dexopt (pid, apk_path) ;  
.....  
}  
....//资源清理  
return-1 ;  
}
```

让人大跌眼镜的是，dex优化工作竟然由installd委派给dexopt进程来实现。dex优化后会生成一个dex文件，一般位于/data/dalvik-cache目录中。这里给出一个示例，如图4-14所示。



```
system@app@trackid.apk@classes.dex  
system@framework@core.jar@classes.dex  
system@app@ReadyToRun.apk@classes.dex
```

图 4-14 dex文件示例

提 示 dexopt 进 程 由 android 源码/dalvik/dexopt/OptMain.cpp 定义。感兴趣的读者

可深入研究dex优化的工作原理。

3.movefiles命令分析

PKMS扫描完系统Package后，将发送该命令给installd，相应处理函数的代码如下：

[-->installd.c : do_movefiles]

```
static int do_movefiles (char**arg, char reply[REPLY_MAX])
{
    return movefiles () ;
}
```

[-->commands.c : movefiles]

```
int movefiles ()
{
    DIR*d ;
    int dfd, subfd ;
    struct dirent*de ;
    struct stat s ;
    char buf[PKG_PATH_MAX+1] ;
    int bufp, bufe, bufi, readlen ;
    char srcpkg[PKG_NAME_MAX] ;
    char dstpkg[PKG_NAME_MAX] ;
    char srcpath[PKG_PATH_MAX] ;
    char dstpath[PKG_PATH_MAX] ;
    int dstuid=-1, dstgid=-1 ;
    int hasspace ;
    //打开/system/etc/updatecmds目录
    d=opendir (UPDATE_COMMANDS_DIR_PREFIX) ;
    if (d==NULL) {
```

```
goto done ;
}
dfd=dirfd (d) ;
while ( (de=readdir (d) ) ) {
.....//解析该目录下的文件，然后执行对应操作
}
closedir (d) ;
done :
return 0 ;
}
```

先来看/system/etc/updatecmds目录下到底是什么文件，这里给出一个示例，如图4-15所示。

以图4-15中最后两行为例，movefiles将把com.google.andorid.gsf下的databases目录转移到com.andorid.providers.im下。从文件中的注释可知，movefiles的功能和系统升级有关。

```
# ls /system/etc/updatecmds/
google_generic_update.txt
# cat google_generic_update.txt
# Commands to move files associated with base Google packages.

# Merge all data into the new com.google.android.gsf package.
com.google.android.gsf:com.google.android.providers.talk
    databases
com.google.android.gsf:com.google.android.googleapps
    databases
com.google.android.gsf:com.google.android.apps.gtalkservice
    files
com.google.android.gsf:com.google.android.providers.settings
    databases

# Merge the old IM database in Donut to the new com.google.android.gsf package.
com.google.android.gsf:com.android.providers.im
    databases
```

图 4-15 movefiles示例

4.doFreeCache

第3章介绍了DeviceStorageMonitorService，当系统空间不足时，DSMS会调用PKMS的freeStorageAndNotify函数进行空间清理。该工作真正的实施者是installd，相应的处理命令为do_free_cache，其代码如下：

[-->installd.c : do_free_cache]

```
static int do_free_cache (char**arg, char reply[REPLY_MAX])
{
    return free_cache ( (int64_t) atol (arg[0]) ) ;
}
```

[-->commands.c : free_cache]

```
int free_cache (int64_t free_size)
{
    const char*name ;
    int dfd, subfd ;
    DIR*d ;
    struct dirent*de ;
    int64_t avail ;
    avail=disk_free () ;//获取当前系统的剩余空间大小
    if (avail<0) return -1 ;
    if (avail>=free_size) return 0 ;
    d=opendir (android_data_dir.path) ;//打开/data目录
    dfd=dirfd (d) ;
    while ( (de=readdir (d)) ) {
        if (de->d_type != DT_DIR) continue ;
        name=de->d_name ;
```

```
.....//略过..和..文件
subfd=openat (dfd, name, O_RDONLY|O_DIRECTORY) ;
//删除/data及各级子目录中的cache文件夹
delete_dir_contents_fd (subfd, "cache") ;
close (subfd) ;
.....//如果剩余空间恢复正常，则返回
}
closedir (d) ;
return -1 ; //清理空间后，仍然不满足要求
}
```

installd的介绍就到此为止，这部分内容比较简单，读者完全可自行深入研究。

4.6.2 UserManager介绍

UserManager是Android 4.0新增的一个功能，其作用是管理手机上的不同用户。这一点和PC上的Windows系统比较相似，例如，在Windows上安装程序时，都会提示是安装给本人使用还是安装给系统所有用户使用。非常遗憾的是，在目前的Android版本中，该功能尚未完全实现，在SDK中也没有相关说明。不过从现有代码中，也能发现一些蛛丝马迹。

提示 小米手机的访客模式和UserManager比较相似。

1. UserManager构造函数分析

在PKMS中，创建UserManager调用的代码如下：

```
//mUserDataDir指向/data/app目录。该目录中包含的是非系统APK文件  
mUserManager=new UserManager (mInstaller, mUserDataDir) ;  
  
[-->UserManager.java : UserManager]  
  
public UserManager (Installer installer, File baseUserPath)  
{  
    this (Environment.getDataDirectory () , baseUserPath) ;  
    mInstaller=installer ;
```

```
}

UserManager (File dataDir, File baseUserPath) {
    //mUsersDir指向/data/system/users目录
    mUsersDir=new File (dataDir, USER_INFO_DIR) ;
    mUsersDir.mkdirs () ;//创建该目录
    mBaseUserPath=baseUserPath ;
    FileUtils.setPermissions (mUsersDir.toString () ,
    FileUtils.S_IRWXU|FileUtils.S_IRWXG
    |FileUtils.S_IROTH|FileUtils.S_IXOTH,
    -1, -1) ;
    //mUserListFile指向/data/system/user/userlist.xml
    mUserListFile=new File (mUsersDir, USER_LIST_FILENAME) ;
    readUserList () ;//解析userlist.xml文件
}
```

此处不深入分析readUserList代码了，只介绍其内部工作流程。

userlist. xml保存每个用户的ID。

readUserList 到 /data/system/users 目录下解析 id.xml，将最终得到的信息保存在UserInfo对象中。

原来用户信息由 UserInfo 表达，下面是 UserInfo的定义。

[-->UserInfo.java]

```
public class UserInfo implements Parcelable{
    //主用户，全系统只能有一个这样的用户
    public static final int FLAG_PRIMARY=0x00000001 ;
```

```
//管理员，可以创建、删除其他用户信息
public static final int FLAG_ADMIN=0x00000002 ;
//访客用户
public static final int FLAG_GUEST=0x00000004 ;
public int id;//ID
public String name;//用户名
public int flags;//属性标志
.....//其他函数
}
```

UserInfo 信息比较简单，笔者觉得 UserManager 的功能暂时还不能满足企业用户的需求。感兴趣的读者不妨关注 Android 未来版本在此方面的变化。

2.installPackageForAllUsers 分析

PKMS 在扫描非系统 APK 的时候，每扫描完一个 APK 都会调用 installPackage-ForAllUsers，调用代码如下：

```
mUserManager.installPackageForAllUsers (      pkgName,
pkg.applicationInfo.uid) ;
```

[--> UserManager.java :
installPackageForAllUsers]

```
public void installPackageForAllUsers (String packageName,
int uid) {
    for (int userId:mUserIds) {
```

```
if (userId==0)
continue ;
//向installd发送命令，其中getUid将userId和uid组合为一个整型值
//installd将在/data对应的user/目录下创建相应的package子目录
mInstaller.createUserData ( packageName,
PackageManager.getUid (userId, uid) ,
userId) ;
}
}
```

4.7 本章学习指导

PKMS是本书分析的第一个重要核心服务，其中的代码量、关联的知识点、涉及的数据结构都比较多。这里提出一些学习建议供读者参考。

从工作流程上看，PKMS包含几条重要的主线。一条是PKMS自身启动时构造函数的工作流程，另外几条和APK安装、卸载相关。每一条主线的难度都比较大，读者可结合日常工作的需求进行单独研究。例如研究如何加快构造函数的执行时间等。

从数据结构上看，PKMS涉及非常多的数据类型。如果对每个数据结构进行孤立分析，很容易陷入不可自拔的状态。笔者建议，不妨跳出各种数据结构的具体形态，只从目的及功能角度去考虑。这里需要读者仔细查看前面的重要数据结构及说明示意图。

另外，由于篇幅所限，本章还有一些内容并没有涉及，需要读者在学习完本章内容的基础上自行研究。这些内容包括：

APK安装在SD卡，以及APK从内部存储空间转移到SD卡的流程。

和Package相关的内容，例如签名管理、dex优化等。

权限管理相关的内容，读者可自行阅读《Application Security for the Android Platform》一书。

4.8 本章小结

本章对 PackageManagerService 进行了较深入的分析。首先分析了PKMS创建时构造函数的工作流程；接着以APK安装为例，较详细地讲解了这个复杂的处理流程；然后又介绍了PKMS另外一项功能，即根据Intent查找匹配的Activities；最后介绍了与installId和UserManager有关的知识。

第5章 深入理解PowerManagerService

本章主要内容：

深入分析PowerManagerService。

深入 分析 BatteryService 和
BatteryStatsService。

本章所涉及的源代码文件名及位置：

PowerManagerService.java
(frameworks/base/services/java/com/android/server/
PowerManagerService.java)

com_android_server_PowerManagerService.cpp
(frameworks/base/services/jni/com_android_server_
PowerManagerService.cpp)

PowerManager.java
(frameworks/base/core/java/android/os/PowerManager.java)

WorkSoure.java
(frameworks/base/core/java/android/os/WorkSoure.java)

Power. java
 (frameworks/base/core/java/android/os/Power.java

)

 android_os_Power. cpp
(frameworks/base/core/jni/android_os_Power.cpp)

 com_android_server_InputManager. cpp
(frameworks/base/services/jni/com_android_server_ InputManager.cpp)

 LightService. java
(frameworks/base/services/java/com/android/server/ LightService.java)

 com_android_server_LightService. cpp
(frameworks/base/services/jni/com_android_server_ LightService.cpp)

 BatteryService. java
(frameworks/base/services/java/com/android/server/ BatteryService.java)

 com_android_server_BatteryService. cpp
(frameworks/base/services/jni/com_android_server_ BatteryService.cpp)

ActivityManagerService. java
(frameworks/base/services/java/com/android/server/am/Activity-ManagerService.java)

BatteryStatsService. java
(frameworks/base/services/java/com/android/server/am/Battery-StatsService.java)

BatteryStatsImpl. java
(frameworks/base/core/java/com/android/internal/os/BatteryStatsImpl.java)

LocalPowerManager. java
(frameworks/base/core/java/android/os/LocalPowerManager.java)

5.1 概述

PowerManagerService负责Android系统中电源管理方面的工作。作为系统核心服务之一，PowerManagerService与其他服务及HAL层等都有交互关系，所以PowerManagerService相对PackageManagerService来说，其社会关系更复杂，分析难度也会更大一些。

先来看直接与PowerManagerService有关的类家族成员，如图5-1所示由图5-1可知：

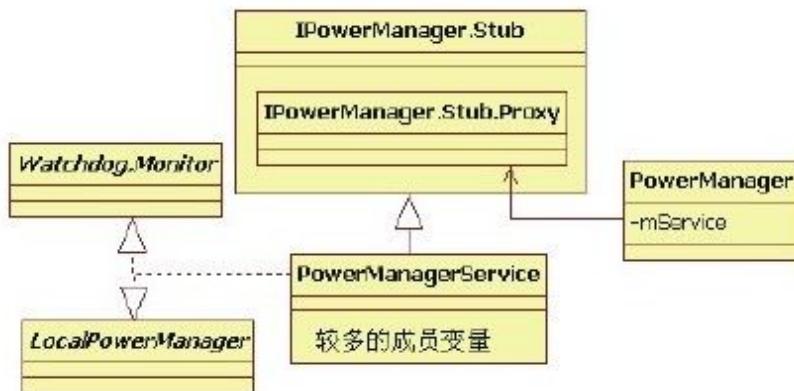


图 5-1 PowerManagerService及相关类家族

PowerManagerService从IPowerManager.Stub类派生，并实现了Watchdog.Monitor及LocalPowerManager接口。PowerManagerService

内部定义了较多的成员变量，在后续分析中，我们会对其中比较重要的成员逐一进行介绍。

根据第4章介绍的知识，IPowerManager.Stub及内部类Proxy均由aidl工具处理IPowerManager.aidl后得到。

客户端使用PowerManager类，其内部通过代表BinderProxy端的mService成员变量与PowerManagerService进行跨Binder通信。

现在开始PowerManagerService（以后简写为PMS）的分析之旅，先从它的调用流程入手。

提示 PMS 和 BatteryService、BatteryStatsService均有交互关系，这些内容放在后面分析。

5.2 初识PowerManagerService

PMS由SystemServer在ServerThread线程中创建。这里从中提取了4个关键调用点，如下所示：

[-->SystemServer.java : run]

```
.....//ServerThread的run函数
power=new PowerManagerService () ;//①创建PMS对象
ServiceManager.addService (Context.POWER_SERVICE, power) ;//注册到SM中
.....
//②调用PMS的init函数
power.init ( context, lights, ActivityManagerService.self ()
(), battery) ;
....//其他服务
power.systemReady () ;//③调用PMS的systemReady
.....//系统启动完毕，会收到ACTION_BOOT_COMPLETED广播
//④PMS处理ACTION_BOOT_COMPLETED广播
```

先从第一个关键点即PMS的构造函数开始分析。

5.2.1 PMS构造函数分析

PMS构造函数的代码如下：

[-->PowerManagerService.java PowerManagerService]

:

```
PowerManagerService () {  
    long token=Binder.clearCallingIdentity () ;  
    MY_UID=Process.myUid () ;//取本进程（即SystemServer）的uid及pid  
    MY_PID=Process.myPid () ;  
    Binder.restoreCallingIdentity (token) ;  
    //设置超时时间为1周。Power类封装了同Linux内核交互的接口。本章最后再来  
    分析它  
    Power.setLastUserActivityTimeout (7*24*3600*1000) ;  
    //初始化两个状态变量，它们非常有意义。其具体作用后续再分析  
    mUserState=mPowerState=0 ;  
    //将自己添加到看门狗（Watchdog）的监控管理队列中  
    Watchdog.getInstance () .addMonitor (this) ;  
}
```

PMS的构造函数比较简单。值得注意的是mUserState和mPowerState两个成员，至于它们的具体作用，后续分析会介绍。

下面分析第二个关键点。

5.2.2 init分析

第二个关键点是init函数，该函数将初始化PMS内部的一些重要成员变量，由于此函数代码较长，此处将分段讨论。

从流程角度看，init大体可分为3段。

1.init分析之一

这部分的代码如下：

[-->PowerManagerService.java : init]

```
void init ( Context context, LightsService lights,
IActivityManager activity,
BatteryService battery) {
//①保存几个成员变量
mLightsService=lights ; //保存LightService
mContext=context ;
mActivityService=activity ; //保存ActivityManagerService
//保存BatteryStatsService
mBatteryStats=BatteryStatsService.getService () ; //
mBatteryService=battery ; //保存BatteryService
//从LightService中获取代表不同硬件Light的Light对象
mLcdLight=lights.getLight
(LightsService.LIGHT_ID_BACKLIGHT) ;
mButtonLight=lights.getLight
(LightsService.LIGHT_ID_BUTTONS) ;
```

```
mKeyboardLight=lights.getLight  
(LightsService.LIGHT_ID_KEYBOARD) ;  
mAttentionLight=lights.getLight  
(LightsService.LIGHT_ID_ATTENTION) ;  
//②调用nativeInit函数  
nativeInit () ;  
synchronized (mLocks) {  
    updateNativePowerStateLocked () ; //③更新Native层的电源状态  
}
```

第一阶段的工作可分为3步：

对一些成员变量进行赋值。

调用nativeInit函数初始化Native层相关资源。

调用updateNativePowerStateLocked更新Native层的电源状态。这个函数的调用次数较为频繁，后续分析时会讨论。

先来看第一阶段出现的各类成员变量，如表5-1所示。

表 5-1 成员变量说明

成员变量名	数据类型	作用
mLightsService	LightsService	和 LightsService 交互用
mActivityService	IActivityManager	和 ActivityManagerService 交互
mBatteryStats	IBatteryStats	和 BatteryStatsService 交互，用于系统耗电量统计
mBatteryService	BatteryService	用于获取电源状态，例如判断是否为低电状态、查询电池电量等
mLcdLight、mButtonLight、mKeyboardLight、mAttentionLight	LightsService.Light	由 PMS 控制，在不同状态下点亮或熄灭它们

下面来看nativeInit函数，其JNI层实现代码如下：

[-->
com_android_server_PowerManagerService.cpp]

```
static void android_server_PowerManagerService_nativeInit
(JNIEnv* env,
 jobject obj) {
    //非常简单，就是创建一个全局引用对象gPowerManagerServiceObj
    gPowerManagerServiceObj=env->NewGlobalRef (obj) ;
}
```

init第一阶段工作比较简单，下面进入第二阶段的分析。

2.init分析之二

init第二阶段工作将创建两个HandlerThread对象，即创建两个带消息循环的工作线程。PMS本身由ServerThread线程创建，但它会将自己的工作委托给两个线程，它们分别是：

mScreenOffThread：按Power键关闭屏幕时，屏幕不是突然变黑的，而是一个渐暗的过程。mScreenOffThread线程就用于控制关闭屏幕过程中的亮度调节。

mHandlerThread：该线程是PMS的主要工作线程。

先来看这两个线程的创建。

(1) mScreenOffThread和mHandlerThread分析这部分的代码如下：

[-->PowerManagerService.java : init]

```
.....  
mScreenOffThread=new HandlerThread  
("PowerManagerService.mScreenOffThread") {  
    protected void onLooperPrepared () {  
        mScreenOffHandler=new Handler () ;//向这个handler发送的消息，将  
由此线程处理  
        synchronized (mScreenOffThread) {  
            mInitComplete=true ;  
            mScreenOffThread.notifyAll () ;  
        }  
    }  
}
```

```
} ;  
mScreenOffThread.start () ;//创建对应的工作线程  
synchronized (mScreenOffThread) {  
    while (!mInitComplete) {  
        try{//等待mScreenOffThread线程创建完成  
            mScreenOffThread.wait () ;  
        }.....  
    }  
}
```

注意，在Android代码中经常出现“线程A创建线程B，然后线程A等待线程B创建完成”的情况，读者了解它们的作用即可。接着看以下代码。

[-->PowerManagerService.java : init]

```
mInitComplete=false ;  
//创建mHandlerThread  
mHandlerThread=new HandlerThread ("PowerManagerService") {  
    protected void onLooperPrepared () {  
        super.onLooperPrepared () ;  
        initInThread () ;//①初始化另外一些成员变量  
    }  
};  
mHandlerThread.start () ;  
.....//等待mHandlerThread创建完成
```

由于 mHandlerThread 承担了 PMS 的主要工作，因此需要先做一些初始化工作，相关的代码在initInThread中，我们会将这部分内容放在单独一节中进行讨论。

(2) initInThread分析

initInThread本身比较简单，涉及3个方面的工作，总结如下：

PMS需要了解外面的世界，所以它会注册一些广播接收对象，接收诸如启动完毕、电池状态变化等广播。

PMS所从事的电源管理工作需要遵守一定的规则，而这些规则在代码中就是一些配置参数，这些配置参数的值可以是固定的（编译完后就无法改动），也可以是经由Settings数据库动态设定的。

PMS需要对外发出一些通知，例如屏幕关闭/开启。

了解initInThread的概貌后，再来看如下代码：

[-->PowerManagerService.java : initInThread]

```
void initInThread () {  
    mHandler=new Handler () ;  
    //PMS 内 部 也 需 要 使 用 WakeLock ， 此 处 定 义 了 几 种 不 同 的  
    //UnsynchronizedWakeLock。它 们 的  
    //作 用 见 后 文 分 析  
    mBroadcastWakeLock=new UnsynchronizedWakeLock (
```

```
PowerManager.PARTIAL_WAKE_LOCK, "sleep_broadcast", true) ;  
//创建广播通知的Intent，用于通知SCREEN_ON和SCREEN_OFF消息  
mScreenOnIntent=new Intent (Intent.ACTION_SCREEN_ON) ;  
mScreenOnIntent.addFlags  
(Intent.FLAG_RECEIVER_REGISTERED_ONLY) ;  
mScreenOffIntent=new Intent (Intent.ACTION_SCREEN_OFF) ;  
mScreenOffIntent.addFlags  
(Intent.FLAG_RECEIVER_REGISTERED_ONLY) ;  
//取配置参数，这些参数是编译时确定的，运行过程中无法修改  
Resources resources=mContext.getResources () ;  
mAnimateScreenLights=resources.getBoolean  
(com.android.internal.R.bool.config_animateScreenLights) ;  
.....//见下文的配置参数汇总  
//通过数据库设置的配置参数  
ContentResolver resolver=mContext.getContentResolver () ;  
Cursor settingsCursor=resolver.query  
(Settings.System.CONTENT_URI, null,  
.....//设置查询条件和查询项的名字，见后文的配置参数汇总  
null) ;  
//ContentQueryMap是一个常用类，简化了数据库查询工作。读者可参考SDK中  
该类的说明文档  
mSettings=new ContentQueryMap (settingsCursor,  
Settings.System.NAME,  
true, mHandler) ;  
//监视上边创建的ContentQueryMap中内容的变化  
SettingsObserver settingsObserver=new SettingsObserver () ;  
mSettings.addObserver (settingsObserver) ;  
settingsObserver.update (mSettings, null) ;  
//注册接收通知的BroadcastReceiver  
IntentFilter filter=new IntentFilter () ;  
filter.addAction (Intent.ACTION_BATTERY_CHANGED) ;  
mContext.registerReceiver (new BatteryReceiver () , filter) ;  
filter=new IntentFilter () ;  
filter.addAction (Intent.ACTION_BOOT_COMPLETED) ;  
mContext.registerReceiver ( new BootCompletedReceiver () ,  
filter) ;
```

```
filter=new IntentFilter () ;
filter.addAction (Intent.ACTION_DOCK_EVENT) ;
mContext.registerReceiver (new DockReceiver () , filter) ;
//监视Settings数据中secure表的变化
mContext.getContentResolver () .registerContentObserver (
Settings.Secure.CONTENT_URI, true,
new ContentObserver (new Handler () ) {
public void onChange (boolean selfChange) {
updateSettingsValues () ;
}
}) ;
updateSettingsValues () ;
.....//通知其他线程
}
```

在上述代码中，很大一部分用于获取配置参数。同时，对于数据库中的配置值，还需要建立监测机制，细节部分请读者自己阅读相关代码，这里总结一下常用的配置参数，如表5-2所示。

表 5-2 PMS 使用的配置参数

参数名 : 类型	来源	备注
mAnimateScreenLights:bool	config.xml ^①	关闭屏幕时屏幕光是否渐暗, 默认为 true
mUnplugTurnsOnScreen:bool	config.xml	拔掉 USB 线, 是否点亮屏幕
mScreenBrightnessDim:int	config.xml	PMS 可设置的屏幕亮度的最小值, 默认为 20 (单位 lx)
mUseSoftwareAutoBrightness:bool	config.xml	是否启用 Setting 中的亮度自动调节, 如果硬件不支持该功能, 则可由软件控制。默认为 false
mAutoBrightnessLevels:int[] mLcdBacklightValues:int[]	config.xml, 具体值由硬件厂商定义	当使用软件自动亮度调节时, 需配置不同亮度时对应的参数
STAY_ON_WHILE_PLUGGED_IN:int	Settings.db	插入 USB 时是否保持唤醒状态
SCREEN_OFF_TIMEOUT:int	Settings.db	屏幕超时时间
DIM_SCREEN:int	Settings.db	是否变暗 (dim) 屏幕
SCREEN_BRIGHTNESS_MODE:int	Settings.db	屏幕亮度模式 (自动还是手动调节)

除了获取配置参数外, initInThread还创建了好几个UnsynchronizedWakeLock对象, 它的作用是: 在Android系统中, 为了抢占电力资源, 客户端要使用WakeLock对象。PMS自己也不例外, 所以为了保证在工作中不至于突然掉电(当其他客户端都不使用WakeLock的时候, 这种情况理论上是有可能发生的), PMS需要定义供自己使用的WakeLock。由于线程同步方面的原因, PMS封装了一个UnsynchronizedWakeLock结构, 它的调用已经处于锁保护下, 所以在内部无须再做同步处

理。UnsynchronizedWakeLock比较简单，因此不再赘述。

下面来看init第三阶段的工作。

3.init分析之三

这部分内容的代码如下：

[-->PowerManagerService.java : init]

```
nativeInit () ;//不知道此处为何还要调用一次nativeInit，笔者怀疑此  
处为bug  
    synchronized (mLocks) {  
        updateNativePowerStateLocked () ;//更新native层power状态，以后  
分析  
        forceUserActivityLocked () ;//强制触发一次用户事件  
        mInitialized=true ;  
    } //init函数完毕
```

forceUserActivityLocked表示强制触发一次用户事件。这个解释是否会让读者丈二和尚摸不着头？先来看它的代码：

[-->PowerManagerService.java :
forceUserActivityLocked]

```
private void forceUserActivityLocked () {  
    if (isScreenTurningOffLocked () ) {  
        mScreenBrightness.animating=false ;  
    }
```

```
boolean savedActivityAllowed=mUserActivityAllowed ;  
mUserActivityAllowed=true ;  
//下面这个函数以后会分析，SDK中有对应的API  
userActivity (SystemClock.uptimeMillis () , false) ;  
mUserActivityAllowed=savedActivityAllowed ;  
}
```

forceUserActivityLocked 内部为调用 userActivity 扫清了一切障碍。SDK 中 PowerManager.userActivity 的说明文档 “User activity happened.Turns the device from whatever state it's in to full on, and resets the auto-off timer.” 简单翻译过来是：调用此函数后，手机将被唤醒，屏幕超时时间也将重新计算。

userActivity 是 PMS 中很重要的一个函数，本章后面将对其进行详细分析。

4.init 函数总结

PMS 的 init 函数比较简单，但是其众多的成员变量让人感到有点头晕。读者自行阅读代码时，不妨参考表 5-1 和表 5-2。

[1] config.xml 文件的全路径是 Android 4.0 源码中的 /frameworks/base/core/res/values/config.xml。

5.2.3 systemReady分析

下面来分析PMS第三阶段的工作。此时系统中大部分服务都已创建好，即将进入就绪阶段。就绪阶段的工作在systemReady中完成，代码如下：

[-->PowerManagerService.java :
systemReady]

```
void systemReady () {  
    /*  
     * 创建一个SensorManager，用于和系统中的传感器系统交互，由于该部分涉及较多的native层  
     * 代码，因此将相关内容放到本丛书后续系列进行讨论  
     */  
    mSensorManager=new SensorManager ( mHandlerThread.getLooper  
() ) ;  
    mProximitySensor=mSensorManager.getDefaultSensor  
(Sensor.TYPE_PROXIMITY) ;  
    if (mUseSoftwareAutoBrightness) {  
        mLIGHTSensor=mSensorManager.getDefaultSensor  
(Sensor.TYPE_LIGHT) ;  
    }  
    if (mUseSoftwareAutoBrightness) {  
        setPowerState (SCREEN_BRIGHT) ;  
    }else{//不考虑软件自动亮度调节，所以执行下面这个分支  
        setPowerState (ALL_BRIGHT) ;//设置手机电源状态为ALL_BRIGHT，即屏幕、按键灯都打开  
    }  
}
```

```
synchronized (mLocks) {  
    mDoneBooting=true ;  
    //根据情况启用LightSensor  
    enableLightSensorLocked ( mUseSoftwareAutoBrightness & &  
    mAutoBrightnessEnabled) ;  
    long identity=Binder.clearCallingIdentity () ;  
    try{//通知BatteryStatsService，它将统计相关的电量使用情况，后续再分  
析它  
        mBatteryStats.noteScreenBrightness ( getPreferredBrightness  
        () ) ;  
        mBatteryStats.noteScreenOn () ;  
    }.....  
}
```

systemReady主要工作为：

PMS创建SensorManager，通过它可与对应的传感器交互。关于Android传感器系统，将放到本书后续章节讨论。PMS仅仅启用或禁止特定的传感器，而来自传感器的数据将通过回调的方式通知PMS，PMS根据接收到的传感器事件做相应处理。

通过setPowerState函数设置电源状态为ALL_BRIGHT（不考虑UseSoftwareAutoBrightness的情况）。此时屏幕及键盘灯都会点亮。关于setPowerState函数，后文再作详细分析。

调用BatteryStatsService提供的函数，以通知屏幕打开事件，在BatteryStatsService内部将处理该事件。稍后，本章将详细讨论BatteryStatsService的功能。

当系统中的服务都在systemReady中进行处理后，系统会广播一次ACTION_BOOT_COMPLETED消息，而PMS也将处理该广播，下面来具体分析。

5.2.4 BootComplete处理

这部分的代码如下：

[-->PowerManagerService.java
BootCompletedReceiver]

```
private final class BootCompletedReceiver extends  
BroadcastReceiver{  
    public void onReceive (Context context, Intent intent) {  
        bootCompleted () ; //调用PMS的bootCompleted函数  
    }  
}
```

[-->PowerManagerService.java
bootCompleted]

```
void bootCompleted () {  
    synchronized (mLocks) {  
        mBootCompleted=true ;  
        //再次碰见userActivity，根据前面的描述，此时将重新计算屏幕超时时间  
        userActivity ( SystemClock.uptimeMillis () , false,  
        BUTTON_EVENT, true) ;  
        updateWakeLockLocked () ; //此处先分析这个函数  
        mLocks.notifyAll () ;  
    }  
}
```

在以上代码中，再一次遇见了userActivity，暂且将其放到一边，我们先来分析updateWakeLockLocked函数，其代码如下：

```
private void updateWakeLockLocked () {  
    /*  
     * mStayOnConditions用于控制当插上USB时，手机是否保持唤醒状态。  
     * mBatteryService的isPowered用于判断当前是否处于USB充电状态。  
     * 如果满足下面的if条件，则PMS需要使用WakeLock来确保系统不会掉电  
     */  
    if ( mStayOnConditions !=0 & & mBatteryService.isPowered  
        (mStayOnConditions) ) {  
        mStayOnWhilePluggedInScreenDimLock.acquire () ;  
        mStayOnWhilePluggedInPartialLock.acquire () ;  
    }else{  
        //如果不满足if条件，则释放对应的WakeLock，这样系统就可以进入休眠状态  
        mStayOnWhilePluggedInScreenDimLock.release () ;  
        mStayOnWhilePluggedInPartialLock.release () ;  
    }  
}
```

mStayOnWhilePluggedInScreenDimLock 和 mStayOnWhilePluggedInPartialLock 都为 UnsynchronizedWakeLock 类型，它们封装了 WakeLock，可帮助PMS在使用它们时免遭线程同步之苦。

5.2.5 初识PowerManagerService总结

这一节向读者展示了PMS的大体面貌，包括：

主要的成员变量及它们的作用和来历。如有需要，可查阅表5-1和表5-2。

见识了PMS中几个主要的函数，其中有一些将留到后文进行深入分析，现在只需要了解其大概作用即可。

5.3 PMS WakeLock分析

WakeLock是Android提供给应用程序获取电力资源的唯一方法。只要还有地方在使用WakeLock，系统就不会进入休眠状态。

WakeLock的一般使用方法如下：

```
PowerManager pm= ( PowerManager ) getSystemService  
(Context.POWER_SERVICE) ;  
//①创建一个WakeLock，注意它的参数  
PowerManager.WakeLock wl=pm.newWakeLock  
(PowerManager.SCREEN_DIM_WAKE_LOCK,  
"My Tag") ;  
wl.acquire () ;//②获取该锁  
.....//完成其他工作  
wl.release () ;//③释放该锁
```

以上代码中共列出3个关键点，本章将分析前两个（在此基础上，读者可自行分析release函数）。

这3个函数都由PMS的Binder客户端的PowerManager使用，所以将本次分析划分为客户端和服务端两大部分。

5.3.1 WakeLock客户端分析

1.newWakeLock分析

通过 PowerManager (以后简称 PM) 的 newWakeLock 将创建一个 WakeLock，代码如下：

[-->PowerManager.java : newWakeLock]

```
public WakeLock newWakeLock (int flags, String tag)
{
    //tag不能为null, 否则抛出异常
    return new WakeLock (flags, tag) ; //WakeLock为PM的内部类, 第一个参数flags很关键
}
```

WakeLock的第一个参数flags很关键，它用于控制CPU/Screen/Keyboard的休眠状态。flags的可选值如表5-3所示。

表 5-3 WakeLock 的 flags 参数说明

flags 值	CPU	Screen	Keyboard	备注
PARTIAL_WAKE_LOCK	On	Off	Off	不受电源键影响
SCREEN_DIM_WAKE_LOCK	On	Dim	Off	按下电源键后, 系统还是会进入休 眠状态
SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off	
FULL_WAKE_LOCK	On	Bright	On	
ACQUIRE_CAUSES_WAKEUP				说明：在正常情况下，获取 WakeLock 并不会唤醒机器（例如 acquire 之前机器处于关屏状态，则无法唤醒）。加上该标志后，acquire WakeLock 同时也能唤醒机器（即点亮屏幕等）。该标志常用于提示框、来电提醒等应用场景
ON_AFTER_RELEASE				说明：和用户体验有关，当 WakeLock 释放后，如没有该标志，则系统会立即黑屏。有了该标志，系统会延时一段时间再黑屏

由表5-3可知：

WakeLock只控制CPU、屏幕和键盘三大部分。

表中最后两项是附加标志，和前面的其他WAKE_LOCK标志组合使用。注意，PARTIAL_WAKE_LOCK比较特殊，附加标志不能影响它。

PARTIAL_WAKE_LOCK不受电源键控制，即按电源键不能使PARTIAL_WAKE_LOCK系统进入休眠状态（屏幕可以关闭，但CPU不会休眠）。了解了上述知识后，再来看如下代码：

[-->PowerManager.java : WakeLock]

```
WakeLock (int flags, String tag)
{
    //检查flags参数是否非法
    mFlags=flags ;
    mTag=tag ;
    //创建一个Binder对象，除了做Token外，PMS需要监视客户端的生死状况，否则有可能导致
    //WakeLock不能被释放
    mToken=new Binder () ;
}
```

客户端创建了WakeLock后，需要调用acquire以确保电力资源供应正常。下面对acquire代码进

行分析。

2.acquire分析

这部分的代码如下：

[-->PowerManager.java : WakeLock.acquire]

```
public void acquire ()  
{  
    synchronized (mToken) {  
        acquireLocked () ; //调用acquireLocked函数  
    }  
}  
  
//acquireLoced函数  
private void acquireLocked () {  
    if (! mRefCounted||mCount++==0) {  
        mHandler.removeCallbacks (mReleaser) ; //引用计数控制  
        try{  
            //调用PMS的acquirewakeLock，注意这里传递的参数，其中mWorkSource为  
            空  
            mService.acquireWakeLock (      mFlags,      mToken,      mTag,  
            mWorkSource) ;  
        }.....  
        mHeld=true ;  
    }  
}
```

上边代码中调用PMS的acquireWakeLock函数与PMS交互，该函数最后一个参数为WorkSource类。这个类从Android 2.2开始就存在，但一直没有明确的作用，下面是关于它的一段说明。

```
/**见WorkSoure.java
 *Describes the source of some work that may be done by
someone else.
 *Currently the public representation of what a work source
is is not
 *defined; this is an opaque container.
 */
```

由以上注释可知，WorkSource本意用来描述某些任务的Source。传递此Source给其他人，这些人就可以执行该Source对应的工作。目前使用WorkSource的地方仅是ContentService中的SyncManager。读者暂时可不理会WorkSource。

客户端的功能比较简单，和PMS仅通过acquireWakeLock函数交互。下面来分析服务端的工作。

5.3.2 PMS acquireWakeLock分析

这部分的代码如下：

[-->PowerManagerService.java
acquireWakeLock]

```
public void acquireWakeLock (int flags, IBinder lock, String tag, WorkSource ws) {  
    int uid=Binder.getCallingUid () ;  
    int pid=Binder.getCallingPid () ;  
    if (uid !=Process.myUid () ) {  
        mContext.enforceCallingOrSelfPermission (//检查WAKE_LOCK权限  
            android.Manifest.permission.WAKE_LOCK, null) ;  
    }  
    if (ws !=null) {  
        //如果ws不为空，需要检查调用进程是否有UPDATE_DEVICE_STATS的权限  
        enforceWakeSourcePermission (uid, pid) ;  
    }  
    long ident=Binder.clearCallingIdentity () ;  
    try{  
        synchronized (mLocks) {//调用acquireWakeLockLocked函数  
            acquireWakeLockLocked (flags, lock, uid, pid, tag, ws) ;  
        }  
    }.....  
}
```

接下来分析acquireWakeLockLocked函数。由于此段代码较长，故分段来分析。

1.acquireWakeLockLocked分析之一

开始分析之前，有必要先介绍另外一个数据结构，它为PowerManagerService的内部类，名字也为WakeLock。其定义如下：

[-->PowerManagerService.java]

```
class WakeLock implements IBinder.DeathRecipient
```

PMS的WakeLock实现了DeathRecipient接口。根据前面Binder系统的知识可知，当Binder服务端死亡后，Binder系统会向注册了讣告接收的Binder客户端发送讣告通知，因此客户端可以做一些资源清理工作。在本例中，PM.WakeLock是Binder服务端，而PMS.WakeLock是Binder客户端。假如PM.WakeLock所在进程在调用release WakeLock函数之前死亡，PMS.WakeLock的binderDied函数就会被调用，这样，PMS也能及时进行释放（release）工作。对于系统的重要资源来说，采用这种安全保护措施很有必要。

回到acquireWakeLockLocked函数，先看第一段代码：

[-->PowerManagerService.java :
acquireWakeLockLocked]

```
public void acquireWakeLockLocked (int flags, IBinder lock,
int uid,
int pid, String tag, WorkSource ws) {
.....
//mLocks是一个ArrayList，保存PMS.WakeLock对象
int index=mLocks.getIndex (lock) ;
WakeLock wl;
boolean newlock;
boolean diffsource;
WorkSource oldsource;
if (index<0) {
//创建一个PMS.WakeLock对象，保存客户端acquire传来的参数
wl=new WakeLock (flags, lock, tag, uid, pid) ;
switch (wl.flags&LOCK_MASK)
{//将flags转换成对应的minState
case PowerManager.FULL_WAKE_LOCK:
if (mUseSoftwareAutoBrightness) {
wl.minState=SCREEN_BRIGHT;
}else{
wl.    minState=      (      mKeyboardVisible?ALL_BRIGHT      :
SCREEN_BUTTON_BRIGHT) ;
}
break;
case PowerManager.SCREEN_BRIGHT_WAKE_LOCK:
wl.minState=SCREEN_BRIGHT;
break;
case PowerManager.SCREEN_DIM_WAKE_LOCK:
wl.minState=SCREEN_DIM;
break;
case PowerManager.PARTIAL_WAKE_LOCK:
//PROXIMITY_SCREEN_OFF_WAKE_LOCK在SDK中并未输出，原因是有些手机
并没有接近
//传感器
case PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK:
break;
default:
```

```
    return ;
}
mLocks.addLock (wl) ; //将PMS.WakeLock对象保存到mLocks中
if (ws !=null) {
wl.ws=new WorkSource (ws) ;
}
newlock=true ; // 设置几个参数信息， newlock 表示新创建了一个
PMS.WakeLock对象
diffsource=false ;
oldsource=null ;
}else{
//如果之前保存有PMS.WakeLock，则要判断新传入的workSource和之前保存
的WorkSource
//是否一样。此处不讨论这种情况
.....
}
```

在上面代码中，很重要的一部分就是将前面 flags 信息 转 成 PMS.WakeLock 的 成 员 变 量 minState，下面是对转换关系的总结。

FULL_WAKE_LOCK : 当 启 用 mUseSoftwareAutoBrightness 时 ， minState 为 SCREEN_BRIGHT (表示屏幕全亮) ，否则为 ALL_BRIGHT (屏幕、键盘、按键全亮。注意，只有在 打 开 键 盘 时 才 能 选 择 此 项) 或 SCREEN_BUTTON_BRIGHT (屏 幕 、 按 键 全 亮) 。

SCREEN_BRIGHT_WAKE_LOCK : minState 为SCREEN_BRIGHT，表示屏幕全亮。

SCREEN_DIM_WAKE_LOCK : minState 为 SCREEN_DIM，表示屏幕Dim。

对 PARTIAL_WAKE_LOCK 和 PROXIMITY_SCREEN_OFF_WAKE_LOCK 情况不做处理。

该做的准备工作都做了，下面来看第二阶段的工作。

2.acquireWakeLockLocked分析之二

这部分的代码如下：

[-->PowerManagerService.java :
acqurieWakeLockLocked]

```
//isScreenLock用于判断flags是否和屏幕有关，除PARTIAL_WAKE_LOCK  
外，其他WAKE_LOCK  
//都和屏幕有关  
if (isScreenLock (flags) ) {  
    if      (      (      flags      &      LOCK_MASK      )  
==PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK) {  
        mProximityWakeLockCount++ ; //引用计数控制  
        if (mProximityWakeLockCount==1) {  
            enableProximityLockLocked () ; //使能Proximity传感器  
        }  
    }else{  
        if ( (wl.flags&PowerManager.ACQUIRE_CAUSES_WAKEUP) !=0) {  
            .....//ACQUIRE_CAUSES_WAKEUP标志处理  
        }else{  
            //①gatherState返回一个状态，稍后分析该函数
```

```
mWakeLockState=      (      mUserState|mWakeLockState      )      &
mLocks.gatherState () ;
}
//②设置电源状态
setPowerState (mWakeLockState|mUserState) ;
}
}
```

以上代码列出了两个关键函数，一个是gatherState，另外一个是setPowerState，下面来分析它们。

(1) gatherState分析

gatherState函数的代码如下：

[-->PowerManagerService.java : gatherState]

```
int gatherState ()
{
int result=0;
int N=this.size () ;
for (int i=0 ; i<N ; i++) {
WakeLock wl=this.get (i) ;
if (wl.activated)
if (isScreenLock (wl.flags) )
result|=wl.minState; //对系统中所有活跃的PMS.WakeLock的状态进
行“或”操作
}
return result;
}
```

由以上代码可知，gatherState将统计当前系统内部活跃的WakeLock的minState。这里为什么要使用或操作呢？举个例子，假如WakeLock A的minState 为 SCREEN_DIM，而 WakeLock B 的minState为SCREEN_BRIGHT，二者共同作用，最终的屏幕状态显然应该是SCREEN_BRIGHT。

下面来分析setPowerState函数。

(2) setPowerState分析

setPowerState用于设置电源状态，先来看其在代码中的调用：

```
setPowerState (mWakeLockState|mUserState) ;
```

在以上代码中除了mWakeLockState外，还有一个mUserState。根据前面对gatherState函数的介绍可知，mWakeLockState的值来源于系统当前活跃的WakeLock的minState。那么mUserState代表什么呢？

mUserState代表用户触发事件导致的电源状态。例如，按 Home 键后，将该值设置为SCREEN_BUTTON_BRIGHT（假设手机没有键盘）。很显然，此时系统的电源状态应该是mUserState和mWakeLockState的组合。

提示“一个小小的变量背后代表了一个很重要的case”，读者能体会到吗？

下面来看setPowerState的代码，这段代码较长，也适合分段来看。第一段代码如下：

[-->PowerManagerService.java
setPowerState]

```
private void setPowerState (int state)
{ //调用另外一个同名函数
    setPowerState (state, false,
WindowManagerPolicy.OFF_BECAUSE_OF_TIMEOUT) ;
}

//setPowerState
private void setPowerState ( int newState, boolean
noChangeLights, int reason)
{
    synchronized (mLocks) {
        int err;
        if (noChangeLights) //在这种情况下，noChangeLights为false
            newState= (newState & ~LIGHTS_MASK) | (mPowerState &
LIGHTS_MASK) ;
        if (mProximitySensorActive) //如果打开了接近感应器，就不需要在这里
点亮屏幕了
            newState= (newState&~SCREEN_BRIGHT) ;
        if (batteryIsLow ()) //判断是否处于低电状态
            newState|=BATTERY_LOW_BIT ;
        else
            newState&=~BATTERY_LOW_BIT ;
        .....
        //如果还没启动完成，则需要将newState置为ALL_BRIGHT。细心的读者有没有
发现，在手机开机过程中键
盘、屏幕、按键等都会全部点亮一会儿呢？
    }
}
```

```
if (!mBootCompleted && !mUseSoftwareAutoBrightness)
newState |= ALL_BRIGHT ;
boolean oldScreenOn= (mPowerState&SCREEN_ON_BIT) !=0 ;
boolean newScreenOn= (newState&SCREEN_ON_BIT) !=0 ;
final boolean stateChanged=mPowerState !=newState ;
```

第一段代码主要用于得到一些状态值，例如在新状态下屏幕是否需要点亮 (new-ScreenOn) 等。再来看第二段代码，它将根据第一段的状态值完成对应的工作。

[-->PowerManagerService : setPowerState]

```
if (oldScreenOn !=newScreenOn) {
if (newScreenOn) {
if (mStillNeedSleepNotification) {
//对sendNotificationLocked函数的分析见后文
sendNotificationLocked (false,
 WindowManagerPolicy.OFF_BECAUSE_OF_USER) ;
}//if (mStillNeedSleepNotification) 判断结束
boolean reallyTurnScreenOn=true ;
if (mPreventScreenOn) //mPreventScreenOn是何方神圣？见下分的分析
reallyTurnScreenOn=false ;
if (reallyTurnScreenOn) {
err=setScreenStateLocked (true) ;//点亮屏幕
.....//通知mBatteryStats做电量统计
mBatteryStats.noteScreenBrightness ( getPreferredBrightness
() ) ;
mBatteryStats.noteScreenOn () ;
}else{//reallyTurnScreenOn为false
setScreenStateLocked (false) ;//关闭屏幕
err=0 ;
}
```

```
if (err==0) {  
    sendNotificationLocked (true, -1) ;  
    if (stateChanged)  
        updateLightsLocked (newState, 0) ;//点亮按键灯或者键盘灯  
    mPowerState|=SCREEN_ON_BIT ;  
}  
}
```

以上代码看起来比较简单，就是根据情况点亮或关闭屏幕。事实果真的如此吗？还记得前面说的“一个小小的变量背后代表一个很重要的case”这句话吗？是的，这里也有一个很重要的case，由mPreventScreenOn表达。这是什么意思呢？

PMS提供了一个函数叫preventScreenOn，该函数（在SDK中未公开）使应用程序可以阻止屏幕点亮。为什么会有这种操作呢？根据该函数的解释，在两个应用之间进行切换时（尤其是正在启动一个Activity却又接到来电通知时），很容易出现闪屏现象，会严重影响用户体验。因此提供了此函数，由应用来调用并处理它。

注意 闪屏的问题似乎解决了，但事情还没完，这个解决方案还引入了另外一个问题：假设应用忘记重新使屏幕点亮，手机岂不是一直就黑屏了？为此，在代码中增加了一段处理逻

辑，即如果5秒后应用还没有使屏幕点亮，PMS将设置mPreventScreenOn为false。

继续看setPowerState最后的代码：

```
else{//newScreenOn为false的情况
.....//更新键盘灯、按键灯的状态
//从mHandler中移除mAutoBrightnessTask，这和光传感器有关。此处不讨论
    mHandler.removeCallbacks (mAutoBrightnessTask) ;
    mBatteryStats.noteScreenOff () ;//通知BatteryStatsService，屏幕已关
    mPowerState= ( mPowerState & ~ LIGHTS_MASK ) | ( newState &
LIGHTS_MASK) ;
    updateNativePowerStateLocked () ;
}
}//if (oldScreenOn !=newScreenOn) 判断结束
else if (stateChanged) {//屏幕的状态不变，但是light的状态有可能变化，所以单独更新light的状态
    updateLightsLocked (newState, 0) ;
}
mPowerState= ( mPowerState & ~ LIGHTS_MASK ) | ( newState &
LIGHTS_MASK) ;
updateNativePowerStateLocked () ;
}//setPowerState完毕
```

setPowerState函数是在PMS中真正设置屏幕及Light状态的地方，其内部将通过Power类与这些硬件交互。相关内容见5.3.3节。

(3) sendNotificationLocked函数分析

sendNotificationLocked 函数用于触发 SCREEN_ON/OFF 广播的发送，来看以下代码：

[-->PowerManagerService.java :
sendNotificationLocked]

```
private void sendNotificationLocked (boolean on, int why) {  
    .....  
    if (!on) {  
        mStillNeedSleepNotification=false ;  
    }  
    int index=0 ;  
    while (mBroadcastQueue[index] != -1) {  
        index++ ;  
    }  
    //mBroadcastQueue和mBroadcastWhy均定义为int数组，成员个数为3，它们有什么作用呢？
```

```
    mBroadcastQueue[index]=on?1:0 ;  
    mBroadcastWhy[index]=why ;  
    /*mBroadcastQueue数组一共有3个元素，根据代码中的注释，其作用如下：  
    当取得的index为2时，即0，1元素已经有值，由于屏幕ON/OFF请求是配对的，  
    所以在这种情况下只需要处理最后一次的请求。例如0元素为ON，1元素为OFF，2元素为ON，则可以去掉0，
```

1的请求，直接处理2的请求，即屏幕ON。对于那种频繁按Power键的操作，通过这种方式可以

节省一次切换操作

*/

```
if (index==2) {  
    if (!on & & mBroadcastWhy[0]>why) mBroadcastWhy[0]=why ;  
    //处理index为2的情况，见上文的说明  
    mBroadcastQueue[0]=on?1:0 ;  
    mBroadcastQueue[1]=-1 ;  
    mBroadcastQueue[2]=-1 ;
```

```
mBroadcastWakeLock.release() ;  
index=0 ;  
}  
/*
```

如果index为1，on为false，即屏幕发出关闭请求，则无须处理。根据注释中的说明，

在此种情况，屏幕已经处于OFF状态，所以无须处理。为什么在此种情况下屏幕已经关闭了呢？

```
*/  
if (index==1 & & !on) {  
mBroadcastQueue[0]=-1 ;  
mBroadcastQueue[1]=-1 ;  
index=-1 ;  
mBroadcastWakeLock.release() ;  
}  
if (mSkippedScreenOn) {  
updateLightsLocked (mPowerState, SCREEN_ON_BIT) ;  
}  
//如果index不为负数，则抛送mNotificationTask给mHandler处理  
if (index>=0) {  
mBroadcastWakeLock.acquire() ;  
mHandler.post (mNotificationTask) ;  
}  
}
```

sendNotificationLocked函数相当“诡异”，主要是mBroadcastQueue数组的使用让人感到困惑。其目的在于减少不必要的屏幕切换和广播发送，但是为什么index为1时，屏幕处于OFF状态呢？下面来分析mNotificationTask，希望它能回答这个问题。

[-->PowerManagerService.java
mNotificationTask]

```
private Runnable mNotificationTask=new Runnable ()  
{  
    public void run ()  
    {  
        while (true) { //此处是一个while循环  
            int value;  
            int why;  
            WindowManagerPolicy policy;  
            synchronized (mLocks) {  
                value=mBroadcastQueue[0]; //取mBroadcastQueue第一个元素  
                why=mBroadcastWhy[0];  
                for (int i=0 ; i<2 ; i++) { //将后面的元素往前挪一位  
                    mBroadcastQueue[i]=mBroadcastQueue[i+1];  
                    mBroadcastWhy[i]=mBroadcastWhy[i+1];  
                }  
                policy=getPolicyLocked (); //policy指向PhoneWindowManager  
                if (value==1&&!mPreparingForScreenOn) {  
                    mPreparingForScreenOn=true;  
                    mBroadcastWakeLock.acquire ();  
                }  
            } //synchronized结束  
            if (value==1) { //value为1， 表示发出屏幕ON请求  
                mScreenOnStart=SystemClock.uptimeMillis ();  
                //和WindowManagerService交互， 和锁屏界面有关  
                //mScreenOnListener为回调通知对象  
                policy.screenTurningOn (mScreenOnListener);  
                ActivityManagerNative.getDefault ().wakingUp (); //和AMS交互  
                if (mContext !=null && ActivityManagerNative.isSystemReady()  
                () ) {  
                    //发送SCREEN_ON广播  
                    mContext.sendOrderedBroadcast (mScreenOnIntent, null,  
                    mScreenOnBroadcastDone, mHandler, 0, null, null);  
                }  
            }  
        }  
    }  
}
```

```
}.....  
}else if (value==0) {  
    mScreenOffStart=SystemClock.uptimeMillis () ;  
    policy.screenTurnedOff (why) ;//通知WindowManagerService  
    ActivityManagerNative.getDefault () .goingToSleep () ;//和AMS  
交互  
    if (mContext !=null & & ActivityManagerNative.isSystemReady  
) {  
        //发送屏幕OFF广播  
        mContext.sendOrderedBroadcast (mScreenOffIntent, null,  
        mScreenOffBroadcastDone, mHandler, 0, null, null) ;  
    }  
}else break ;  
}  
};
```

mNotificationTask 比较复杂，但是它对 mBroadcastQueue 的处理比较有意思，每次取出第一个元素值后，将后续元素往前挪一位。这种处理方式能解决之前提出的那个问题吗？

说实话，目前笔者也没找到能解释 index 为 1 时，屏幕一定处于 OFF 的证据。如果有哪位读者找到证据，不妨分享一下。

另外，mNotificationTask 和 ActivityManagerService 及 WindowManagerService 都有交互。因为这两个服务内部也使用了 WakeLock，所以需要通知它们释放 WakeLock，否则会导致不必要的电力资源消耗。具体内容只能留待以后分析相关服务时再来讨论了。

(4) acquireWakeLocked第二阶段工作总结

acquireWakeLocked第二阶段工作是处理和屏幕相关的WAKE_LOCK方面的工作（isScreenLock返回为true的情况）。其中一个重要的函数就是setPowerState，该函数将根据不同的状态设置屏幕光、键盘灯等硬件设备。注意，和硬件交互相关的工作是通过Power类提供的接口完成的。

3.acquireWakeLockLocked分析之三

这部分 acquireWakeLocked 主要处理 WAKE_LOCK 为 PARTIAL_WAKE_LOCK 的情况，来看以下代码：

[-->PowerManagerService.java :
acquireWakeLockLocked]

```
else if ((flags & LOCK_MASK) == PowerManager.PARTIAL_WAKE_LOCK) {
    if (newlock) {
        mPartialCount++;
    }
    //获取kernel层的PARTIAL_WAKE_LOCK，该函数后续再分析
    Power.acquireWakeLock (Power.PARTIAL_WAKE_LOCK,
    PARTIAL_NAME);
} //else if判断结束
if (diffsource) {
    noteStopWakeLocked (wl, oldsource);
}
if (newlock||diffsource) {
```

```
    noteStartWakeLocked (wl, ws) ; //通知BatteryStatsService做电量  
统计  
}
```

当客户端使用PARTIAL_WAKE_LOCK时，PMS会调用Power.acquireWakeLock申请一个内核的WakeLock。

4.acquireWakeLock总结

acquireWakeLock有3个阶段的工作，总结如下：

如果对应的WakeLock不存在，则创建一个WakeLock对象，同时将WAKE_LOCK标志转换成对应的minState；否则，从mLocks中查找对应的WakeLock对象，然后更新其中的信息。

当WAKE_LOCK标志和屏幕有关时，需要做相应的处理，例如点亮屏幕、打开按键灯等。实际上这些工作不仅影响电源管理，还会影响到用户感受，所以其中还穿插了一些和用户体验有关的处理逻辑（如上面注释的mPreventScreenOn变量）。

当 WAKE_LOCK 和 PARTIAL_WAKE_LOCK 有关时，仅简单调用Power的acquire-WakeLock即

可， 其中涉及和Linux Kernel电源管理系统的交互。

5.3.3 Power类及LightService类介绍

根据前面的分析，PMS有时需要进行点亮屏幕、打开键盘灯等操作，为此Android提供了Power类及LightService满足PMS的要求。这两个类比较简单，但是其背后的Kernel层相对复杂一些。本章仅分析用户空间的内容，有兴趣的读者不妨以此为入口，深入研究Kernel层的实现。

1.Power类介绍

Power类提供了6个函数，如下所示：

[-->Power.java]

```
int setScreenState (boolean on) ;//打开或关闭屏幕灯  
int setLastUserActivityTimeout (long ms) ;//设置超时时间  
void reboot (String reason) ;//用于手机重启，内部调用  
rebootNative  
void shutdown () ;//已作废，建议不要调用  
void acquireWakeLock (int lock, String id) ;//获取Kernel层的  
WakeLock  
void releaseWakeLock (String id) ;//释放Kernel层的WakeLock
```

这些函数固有的实现代码如下：

[-->android_os_Power.cpp :
acqurieWakeLock]

```
static void acquireWakeLock (JNIEnv*env, jobject clazz, jint lock, jstring idObj)
{
    .....
    const char*id=env->GetStringUTFChars (idObj, NULL) ;
    acquire_wake_lock (lock, id) ;//调用此函数和Kernel层交互
    env->ReleaseStringUTFChars (idObj, id) ;
}

static void releaseWakeLock ( JNIEnv*env, jobject clazz,
jstring idObj)
{
    const char*id=env->GetStringUTFChars (idObj, NULL) ;
    release_wake_lock (id) ;//释放Kernel层的WakeLock
    env->ReleaseStringUTFChars (idObj, id) ;
}

static int setLastUserActivityTimeout ( JNIEnv*env, jobject clazz, jlong timeMS)
{
    return set_last_user_activity_timeout (timeMS/1000) ;//设置超时时间
}

static int setScreenState ( JNIEnv*env, jobject clazz,
jboolean on)
{
    return set_screen_state (on) ;//开启或关闭屏幕灯
}

static void android_os_Power_shutdown ( JNIEnv*env, jobject clazz)
{
    android_reboot (ANDROID_RB_POWEROFF, 0, 0) ;//关机
}

static void android_os_Power_reboot ( JNIEnv*env, jobject clazz, jstring reason)
{
    if (reason==NULL) {
```

```
    android_reboot (ANDROID_RB_RESTART, 0, 0) ; //重启
} else{
    const char*chars=env->GetStringUTFChars (reason, NULL) ;
    android_reboot (ANDROID_RB_RESTART2, 0, (char*) chars) ;//重启
    env->ReleaseStringUTFChars (reason, chars) ;
}
jniThrowIOException (env, errno) ;
}
```

Power类提供了和内核交互的通道，读者仅作了解即可。

2.LightService介绍

LightService.java比较简单，这里直接介绍Native层的实现，主要关注HAL层的初始化函数init_native及操作函数setLight_native。

首先来看初始化函数init_native，其代码如下：

```
[com_android_server_LightService.cpp : init_native]
static jint init_native (JNIEnv*env, jobject clazz)
{
    int err ;
    hw_module_t*module ;
    Devices*devices ;
    devices= (Devices*) malloc (sizeof (Devices) ) ;
    //初始化硬件相关的模块，模块名为lights
    err=hw_get_module (LIGHTS_HARDWARE_MODULE_ID,
        (hw_module_t const**) &module) ;
```

```
if (err==0) {  
    devices->lights[LIGHT_INDEX_BACKLIGHT]//背光  
    =get_device (module, LIGHT_ID_BACKLIGHT) ;  
    devices->lights[LIGHT_INDEX_KEYBOARD]//键盘灯  
    =get_device (module, LIGHT_ID_KEYBOARD) ;  
    devices->lights[LIGHT_INDEX_BUTTONS]//按键灯  
    =get_device (module, LIGHT_ID_BUTTONS) ;  
    devices->lights[LIGHT_INDEX_BATTERY]//电源指示灯  
    =get_device (module, LIGHT_ID_BATTERY) ;  
    devices->lights[LIGHT_INDEX_NOTIFICATIONS]//通知灯  
    =get_device (module, LIGHT_ID_NOTIFICATIONS) ;  
    devices->lights[LIGHT_INDEX_ATTENTION]//警示灯  
    =get_device (module, LIGHT_ID_ATTENTION) ;  
    devices->lights[LIGHT_INDEX_BLUETOOTH]//蓝牙提示灯  
    =get_device (module, LIGHT_ID_BLUETOOTH) ;  
    devices->lights[LIGHT_INDEX_WIFI]//WIFI提示灯  
    =get_device (module, LIGHT_ID_WIFI) ;  
}  
else{  
    memset (devices, 0, sizeof (Devices) ) ;  
}  
return (jint) devices;  
}
```

Android系统想得很周到，提供了多达8种不同类型的灯。可是有多少手机包含了所有的灯呢？

PMS点亮或关闭灯时，将调用setLight_native函数，其代码如下：

```
[com_android_server_LightService.cpp : setLight_native]  
static void setLight_native (JNIEnv*env, jobject clazz, int  
ptr,
```

```
    int light, int colorARGB, int flashMode, int onMS, int
offMS,
        int brightnessMode)
{
Devices*devices= (Devices*) ptr ;
light_state_t state;
.....
memset (&state, 0, sizeof (light_state_t) ) ;
state.color=colorARGB ; //设置颜色
state.flashMode=flashMode ; //设置闪光模式
state.flashOnMS=onMS ; //和闪光模式有关，例如亮2秒，灭2秒
state.flashOffMS=offMS ;
state.brightnessMode=brightnessMode ; //
//传递给HAL层模块进行处理
devices->lights[light]->set_light (           devices->
lights[light], &state) ;
}
```

5.3.4 WakeLock总结

相信读者此时已经对WakeLock机制有了比较清晰的认识，此处以flags标签为出发点，对WakeLock的知识点进行总结。

如果flags和屏幕有关（除PARTIAL_WAKE_LOCK外），则需要更新屏幕、灯光状态。其中，屏幕操作通过Power类来完成，灯光操作则通过LightService类来完成。

如果FLAGS是PARTIAL_WAKE_LOCK，则需要通过Power提供的接口获取Kernel层的WakeLock。

在WakeLock工作流程中还混杂了用户体验、光传感器、接近传感器等方面的处理逻辑。这部分代码集中体现在setPowerState函数中。感兴趣的读者可进行深入研究。

WakeLock还要通知BatteryStatsService，以帮助其统计电量使用情况。这方面内容放到本章最后分析。

另外，PMS在JNI层也保存了当前屏幕状态信息，这是通过updateNativePowerState-Locked完成

的，其代码如下：

[-->PowerManagerService.java
updateNativePowerStateLocked]

```
private void updateNativePowerStateLocked () {  
    nativeSetPowerState //调用native函数，传入两个参数  
    (mPowerState&SCREEN_ON_BIT) !=0,  
    (mPowerState&SCREEN_BRIGHT) ==SCREEN_BRIGHT) ;  
}  
//JNI层实现代码如下  
static void  
android_server_PowerManagerService_nativeSetPowerState (  
    JNIEnv*env, jobject serviceObj, jboolean screenOn, jboolean  
    screenBright) {  
    AutoMutex_l (gPowerManagerLock) ;  
    gScreenOn=screenOn ; //屏幕是否开启  
    gScreenBright=screenBright ; //屏幕灯是否全亮  
}
```

PMS 的 updateNativePowerStateLocked 函数曾一度让笔者感到非常困惑，主要原因是初看此函数名，感觉它极可能会和Kernel层的电源管理系统交互。等深入JNI层代码后发现，其功能仅是保存两个全局变量，和Kernel压根儿没有关系。其实，和Kernel层电源管理系统交互的主要Power类。此处的两个变量是为了方便Native层代码查询当前屏幕状态而设置的，以后分析Android输入系统时就会搞清楚它们的作用了。

5.4 userActivity及Power按键处理分析

本节介绍userActivity函数及PMS对Power按键的处理流程。

5.4.1 userActivity分析

前面曾经提到过userActivity的作用，此处举一个例子以加深读者对它的印象。

打开手机，并解锁进入桌面。如果在规定时间内不操作手机，那么屏幕将变暗，最后关闭。在此过程中，如果触动屏幕，屏幕又会重新变亮。这个触动屏幕的操作将导致userActivity函数被调用。

在上述例子中实际上包含了两方面的内容：

不操作手机，屏幕将变暗，最后关闭。在PMS中，这是一个状态切换的过程。

操作手机，将触发userActivity，此后屏幕的状态将重置。

来看以下代码：

[-->PowerManagerService.java : userActivity]

```
public void userActivity ( long time, boolean  
noChangeLights) {  
    ....//检查调用进程是否有设置DEVICE_POWER的权限  
    userActivity ( time , -1 , noChangeLights, OTHER_EVENT,  
false) ;  
}
```

此处将调用另外一个同名函数。注意第三个参数的值OTHER_EVENT。系统一共定义了3种事件，分别是OTHER_EVENT（除按键、触摸屏外的事件）、BUTTON_EVENT（按键事件）和TOUCH_EVENT（触摸屏事件）。它们主要为BatteryStatsService进行电量统计时使用，例如触摸屏事件的耗电量和按键事件的耗电量等。

[-->PowerManagerService.java : userActivity]

```
private void userActivity (long time, long timeoutOverride,  
boolean noChangeLights, int eventType, boolean force) {  
if ( ( (mPokey&POKE_LOCK_IGNORE_TOUCH_EVENTS) !=0) &&  
(eventType==TOUCH_EVENT) ) {  
//mPokey和输入事件的处理策略有关。如果此处的if判断得到满足，则表示忽略  
TOUCH_EVENT  
    return ;  
}  
    synchronized (mLocks) {  
if (isScreenTurningOffLocked () ) {  
    return ;  
}
```

```
if (mProximitySensorActive & & mProximityWakeLockCount==0)
    mProximitySensorActive=false ; //控制接近传感器
    if (mLastEventTime<=time||force) {
        mLastEventTime=time ;
        if      (      (      mUserActivityAllowed      &      &      !
mProximitySensorActive) ||force) {
            if (eventType==BUTTON_EVENT & & ! mUseSoftwareAutoBrightness)
{
            mUserState= (mKeyboardVisible?ALL_BRIGHT :
SCREEN_BUTTON_BRIGHT) ;
        }else{
            mUserState|=SCREEN_BRIGHT ; //设置用户事件导致的mUserState
        }
        .....//通知BatteryStatsService进行电量统计
        mBatteryStats.noteUserActivity (uid, eventType) ;
        //重新计算WakeLock状态
        mWakeLockState=mLocks.reactivateScreenLocksLocked () ;
        setPowerState (mUserState|mWakeLockState, noChangeLights,
WindowManagerPolicy.OFF_BECAUSE_OF_USER) ;
        //重新开始屏幕计时
        setTimeoutLocked (time, timeoutOverride, SCREEN_BRIGHT) ;
    }
}
}
}
//mPolicy指向PhoneWindowManager， 用于和WindowManagerService交
互
if (mPolicy !=null) {
    mPolicy.userActivity () ;
}
}
```

有了前面分析的基础，相信很多读者都会觉得userActivity函数很简单。在前面的代码中，通过setPowerState点亮了屏幕，那么经过一段时间后

发生的屏幕状态切换在哪儿进行呢？来看
setTimeoutLocked函数的代码：

[-->PowerManagerService.java :
setTimeoutLocked]

```
private void setTimeoutLocked ( long now, final long originalTimeoutOverride,
    int nextState) {
    //在本例中，nextState为SCREEN_BRIGHT，originalTimeoutOverride
    为-1
    long timeoutOverride=originalTimeoutOverride;
    if (mBootCompleted) {
        synchronized (mLocks) {
            long when=0 ;
            if (timeoutOverride<=0) {
                switch (nextState)
                {
                    case SCREEN_BRIGHT :
                        when=now+mKeylightDelay ; //得到一个超时时间
                        break ;
                    case SCREEN_DIM :
                        if (mDimDelay>=0) {
                            when=now+mDimDelay ;
                            break ;
                        }.....}
                    case SCREEN_OFF :
                        synchronized (mLocks) {
                            when=now+mScreenOffDelay ;
                        }
                    break ;
                    default :
                        when=now ;
                    break ;
                }
            }
        }
    }
}
```

```
}

}.....//处理timeoutOverride大于零的情况，无非就是设置状态和超时时间
mHandler.removeCallbacks (mTimeoutTask) ;
mTimeoutTask.nextState=nextState ;
mTimeoutTask.remainingTimeoutOverride=timeoutOverride>0
? (originalTimeoutOverride-timeoutOverride)
: -1 ;
//抛送一个mTimeoutTask交给mHandler执行，执行时间为when秒后
mHandler.postAtTime (mTimeoutTask, when) ;
mNextTimeout=when ; //调试用
}
}
}
```

接下来看mTimeOutTask的代码：

[-->PowerManagerService.java : TimeoutTask]

```
private class TimeoutTask implements Runnable
{
int nextState;
long remainingTimeoutOverride;
public void run ()
{
synchronized (mLocks) {
if (nextState== -1) return ;
mUserState=this.nextState ;
//调用setPowerState去真正改变屏幕状态
setPowerState (this.nextState|mWakeLockState) ;
long now=SystemClock.uptimeMillis () ;
switch (this.nextState)
{
case SCREEN_BRIGHT :
if (mDimDelay>=0) { //设置下一个状态为SCREEN_DIM
```

```
    setTimeoutLocked      (      now,      remainingTimeoutOverride,
SCREEN_DIM) ;
    break ;
}
case SCREEN_DIM://设置下一个状态为SCREEN_OFF
    setTimeoutLocked      (      now,      remainingTimeoutOverride,
SCREEN_OFF) ;
    break ;
}.....//省略花括号
}
```

TimeoutTask就是用来切换屏幕状态的，相信不少读者已经在网络上见过一个和PMS屏幕状态切换相关的图（其实就是TimeoutTask的工作流程解释），对此，本章就不再介绍了，希望读者能通过直接阅读源码加深理解。

5.4.2 Power按键处理分析

按键处理属于本书后续将会分析的输入系统的范围，此处仅对其中和Power键相关的代码进行分析，代码如下：

[-->com_android_server_InputManager.cpp : handleInterceptActions]

```
void NativeInputManager : handleInterceptActions ( jint
wmActions, nsecs_t when,
        uint32_t& policyFlags) {
    //按下Power键并松开后，将设置wmActions为WM_ACTION_GO_TO_SLEEP，
    表示需要休眠
    if (wmActions&WM_ACTION_GO_TO_SLEEP) {
        //利用JNI调用PMS的goToSleep函数
        android_server_PowerManagerService_goToSleep (when) ;
    }
    //一般的输入事件将触发userActivity函数被调用，此时将唤醒手机
    if (wmActions&WM_ACTION_POKE_USER_ACTIVITY) {
        //利用JNI调用PMS的userActivity函数。相关内容在前一节已经分析过了
        android_server_PowerManagerService_userActivity (when,
POWER_MANAGER_BUTTON_EVENT) ;
    }
    ....//其他处理
}
```

由以上代码中的注释可知，当按下Power键并松开时[1]，将触发PMS的goToSleep函数被调用。

下面来看goToSleep函数的代码：

[-->PowerManagerService.java : goToSleep]

```
public void goToSleep (long time)
{
    goToSleepWithReason (time,
WindowManagerPolicy.OFF_BECAUSE_OF_USER) ;
}
public void goToSleepWithReason (long time, int reason)
{
    mContext.enforceCallingOrSelfPermission (//检查调用进程是否有
DEVICE_POWER权限
        android.Manifest.permission.DEVICE_POWER, null) ;
    synchronized (mLocks) {
        goToSleepLocked (time, reason) ; //调用goToSleepLocked函数
    }
}
```

[-->PowerManagerService.java :
goToSleepLocked]

```
private void goToSleepLocked (long time, int reason) {
    if (mLastEventTime<=time) {
        mLastEventTime=time ;
        mWakeLockState=SCREEN_OFF ;
        int N=mLocks.size () ;
        int numCleared=0 ;
        boolean proxLock=false ;
        for (int i=0 ; i<N ; i++) {
            WakeLock wl=mLocks.get (i) ;
            if (isScreenLock (wl.flags) ) {
                if ( (wl.flags&LOCK_MASK) ==
```

```
PowerManager.PROXIMITY_SCREEN_OFF_WAKE_LOCK)
&& reason==WindowManagerPolicy.OFF_BECAUSE_OF_PROX_SENSOR) {
proxLock=true ; //判断goToSleep的原因是否与接近传感器有关
}else{
mLocks.get (i) .activated=false ; //禁止和屏幕相关的WakeLock
numCleared++ ;
}
}//if (isScreenLock (wl.f (ags) ) 判断结束
}//for循环结束
if (! proxLock) {
mProxIgnoredBecauseScreenTurnedOff=true ;
}
mStillNeedSleepNotification=true ;
mUserState=SCREEN_OFF ;
setPowerState (SCREEN_OFF, false, reason) ; //关闭屏幕
cancelTimerLocked () ; //从mHandler中撤销mTimeoutTask任务
}
}
```

掌握了前面的基础知识就会感到Power键的处理流程真的是很简单，读者是否也有同感呢？

[1]必须在一定时间内完成按下和松开Power键的操作，否则系统会认为是关机操作。详情将在即将出版的“卷 III”输入系统一章进行分析。

5.5 BatteryService及BatteryStatsService分析

从前面介绍PMS的代码中发现，PMS和系统中其他两个服务（BatteryService及BatteryStatsService）均有交互，其中：

BatteryService提供接口用于获取电池信息，充电状态等。

BatteryStatsService主要用于用电统计，通过它可知谁是系统中的“耗电大户”。

下面先来介绍稍简单的BatteryService。

5.5.1 BatteryService分析

BatteryService由SystemServer创建，代码如下：

```
battery=new BatteryService (context, lights) ;  
ServiceManager.addService ("battery", battery) ;
```

下面来看BatteryService的构造函数：

[-->BatteryService.java : Battery Service]

```
public BatteryService ( Context context, LightsService
lights) {
    mContext=context;
    mLed=new Led (context, lights) ;//提示灯控制，感兴趣的读者可自行
阅读相关代码
    //BatteryService也需要和BatteryStatsService交互
    mBatteryStats=BatteryStatsService.getService () ;
    //获取一些配置参数
    mCriticalBatteryLevel=mContext.getResources () .getInteger (
        com.android.internal.R.integer.config_criticalBatteryWarning
Level) ;
    mLowBatteryWarningLevel=mContext.getResources () .getInteger
(
    com.android.internal.R.integer.config_lowBatteryWarningLevel
) ;
    mLowBatteryCloseWarningLevel=mContext.getResources
() .getInteger (
    com.android.internal.R.integer.config_lowBatteryCloseWarning
Level) ;
    //启动uevent监听对象，监视power_supply信息
    mPowerSupplyObserver.startObserving
("SUBSYSTEM=power_supply") ;
    //如果下列文件存在，那么启动另一个uevent监听对象。该uevent事件来自
invalid charger
    //switch设备（即不匹配的充电设备）
    if ( new File
( "/sys/devices/virtual/switch/invalid_charger/state" ) .exists
() ) {
        mInvalidChargerObserver.startObserving (
        "DEVPATH=/devices/virtual/switch/invalid_charger" ) ;
    }
    update () ;//①查询HAL层，获取此时的电池信息
}
```

BatteryService定义了3个非常重要的阈值，分别是：

mCriticalBatteryLevel表示严重低电，其值为4。当电量低于该值时会强制关机。该值由config.xml中的config_criticalBatteryWarningLevel控制。

mLowBatteryWarningLevel表示低电，值为15，当电量低于该值时，系统会报警，例如闪烁LED灯。该值由config.xml中的config_lowBatteryWarningLevel控制。

mLowBatteryCloseWarningLevel表示一旦电量大于此值，就脱离低电状态，即可停止警示灯。该值为20，由config.xml中的config_lowBatteryCloseWarningLevel控制。在BatteryService构造函数的最后调用了update函数，该函数将查询系统电池信息，以

更新BatteryService内部的成员变量。此函数代码如下：

[-->BatteryService.java：update]

```
private synchronized final void update () {  
    native_update () ; //到Native层查询并更新内部变量的值  
    processValues () ; //处理更新后的状态
```

}

1.native_update函数分析

native_update的实现代码如下：

[-->com_android_server_BatteryService.cpp :
android_server_BatteryService_update]

```
static void android_server_BatteryService_update
(JNIEnv*env, jobject obj)
{
    setBooleanField ( env, obj, gPaths.acOnlinePath,
gFieldIds.mAcOnline) ;
.....//获取电池信息，并通过JNI设置到Java层对应的变量中
    setIntField (env, obj, gPaths.batteryTemperaturePath,
gFieldIds.mBatteryTemperature) ;
    const int SIZE=128 ;
    char buf[SIZE] ;
    //获取信息，以下参数并不是所有手机都支持的
    if (readFromFile (gPaths.batteryStatusPath, buf, SIZE) >0)
        env->SetIntField (obj, gFieldIds.mBatteryStatus,
getBatteryStatus (buf) ) ;
    else
        env->SetIntField (obj, gFieldIds.mBatteryStatus,
gConstants.statusUnknown) ;
.....
}
```

Android系统中的电池信息如表5-4所示。

表 5-4 Android 系统中的电池信息

变量名	功 能	备 注
mAcOnline	是否用外接充电器充电	即用交流电充电
mUsbOnline	是否用 USB 供电	—
mBatteryStatus	电池状态	共有 5 个状态，详细内容可参考 com_android_server_BatteryService.cpp 中 BatteryManagerConstants 的定义
mBatteryHealth	电池健康状态	共 7 个状态，详细内容可参考 com_android_server_BatteryService.cpp 中 BatteryManagerConstants 的定义
mBatteryPresent	是否使用电池	有些手机在没有电池的情况下可直接利用 USB/ 交流供电
mBatteryLevel	电池电量	—
mBatteryVoltage	电池电压	—
mBatteryTemperature	电池温度	—
mBatteryTechnology	电池制造技术	一般为 Li-poly，即锂电池技术

mBatteryStatus 和 mBatteryHealth 均有几种不同状态，详细信息可查看 getBatteryStatus 和 getBatteryHealth 函数的实现。

上述信息均通过从 /sys/class/power_supply 目录读取对应文件得到。和以往使用固定路径（可能是 Android 2.2 版本之前）不同的是，本处先读取 power_supply 目录中各个子目录中的 type 文件，然后根据 type 文件的内容，再做对应处理：

如果 type 文件的内容为 Mains，则读取对应子目录中的 online 文件，可判断是否为 AC 充电。

如果 type 文件的内容为 Battery，则从对应子目录中其他的文件中读取电池相关的信息，例如从

temp文件获取电池温度，从technology文件读取电池制造技术等。

如果type文件的内容为USB，则读取该子目录中的online文件内容，可判断是否为USB充电。

提示 读者可通过dumpsys battery查看自己手机的电池信息。

2.processValues分析

获取了电池信息后，BatteryService就要做一些处理，此项工作通过processValues完成，其代码如下：

[-->BatteryService.java : processValues]

```
private void processValues () {  
    long dischargeDuration=0 ;  
    mBatteryLevelCritical=mBatteryLevel<  
    =mCriticalBatteryLevel ;  
    if (mAcOnline) {  
        mPlugType=BatteryManager .BATTERY_PLUGGED_AC ;  
    }else if (mUsbOnline) {  
        mPlugType=BatteryManager .BATTERY_PLUGGED_USB ;  
    }else{  
        mPlugType=BATTERY_PLUGGED_NONE ;  
    }  
    //通知BatteryStatsService，该函数以后再分析  
    mBatteryStats.setBatteryState ( mBatteryStatus,  
    mBatteryHealth,
```

```
mPlugType,           mBatteryLevel,           mBatteryTemperature,  
mBatteryVoltage  
) ;  
shutdownIfNoPower () ;//如果电量不够，则弹出关机对话框  
shutdownIfOverTemp () ;//如果电池过热，则弹出关机对话框  
.....//根据当前电池信息与上次电池信息比较，判断是否需要发送广播等  
if (...../*比较前后两次电池信息是否发生变化*/) {  
.....//记录信息到日志文件  
Intent statusIntent=new Intent () ;  
statusIntent.setFlags (  
Intent.FLAG_RECEIVER_REGISTERED_ONLY_BEFORE_BOOT) ;  
if (mPlugType !=0 & &mLastPlugType==0) {  
statusIntent.setAction (Intent.ACTION_POWER_CONNECTED) ;  
mContext.sendBroadcast (statusIntent) ;  
}.....  
if (sendBatteryLow) {  
mSentLowBatteryBroadcast=true ;//发送低电提醒  
statusIntent.setAction (Intent.ACTION_BATTERY_LOW) ;  
mContext.sendBroadcast (statusIntent) ;  
}.....  
mLed.updateLightsLocked () ;//更新LED灯状态  
mLastBatteryStatus=mBatteryStatus ;//保存新的电池信息  
.....  
}
```

processValues函数非常简单，此处不再详述。
另外，当电池信息发生改变时，系统会发送uevent事件给BatteryService，此时BatteryService只要重新调用update即可完成工作。

5.5.2 BatteryStatsService分析

BatteryStatsService（以后简称BSS）主要功能是收集系统中各模块和应用进程用电量情况。抽象地说，BSS就是一块“电表”，不过这块“电表”不只是显示总的耗电量，而是分门别类地显示耗电量，力图做到更为精准。

和其他服务不太一样的是，BSS的创建和注册是在ActivityManagerService中进行的，相关代码如下：

[-->ActivityManagerService.java :
ActivityManagerService]

```
private ActivityManagerService () {  
    .....// 创建 BSS 对象，传递一个 File 对象，指向 /data/system/batterystats.bin  
    mBatteryStatsService=new BatteryStatsService (new File (  
        systemDir, "batterystats.bin") .toString () ) ;  
}
```

BBS发布的代码如下：

[-->ActivityManagerService.java : main]

```
// 调用 BSS 的 publish 函数，在内部将其注册到 ServiceManager
```

```
m.mBatteryStatsService.publish (context) ;
```

下面来分析BSS的构造函数。

1.BatteryStatsService介绍

让人大跌眼镜的是，BSS其实只是一个壳，具体功能委托 BatteryStatsImpl（以后简称 BSImpl）来实现。BatteryStatsService 的代码如下：

[-->BatteryStatsService.java :
BatteryStatsService]

```
BatteryStatsService (String filename) {  
    mStats=new BatteryStatsImpl (filename) ;  
}
```

图5-2展示了BSS及BSImpl的家族图谱。

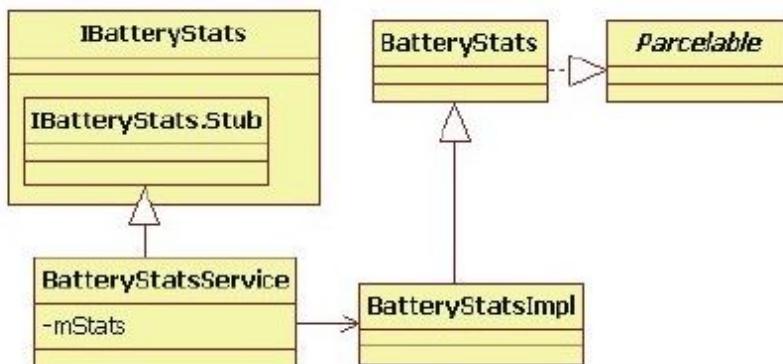


图 5-2 BSS及BSImpl家族图谱

由图5-2可知：

BSS通过成员变量mStats指向一个BSImpl类型的对象。

BSImpl从BatteryStats类派生。更重要的是，该类实现了Parcelable接口，由此可知，BSImpl对象的信息可以写到Parcel包中，从而可通过Binder在进程间传递。实际上，在Android手机的设置中查到的用电信息就是来自BSImpl的。

BSS的getStatistics函数提供了查询系统用电信息的接口，该函数的代码如下：

[-->BatteryStatsService : getStatistics]

```
public byte[]getStatistics () {  
    mContext.enforceCallingPermission ( // 检查调用进程是否有  
    BATTERY_STATS权限  
    android.Manifest.permission.BATTERY_STATS, null) ;  
    Parcel out=Parcel.obtain () ;  
    mStats.writeToParcel (out, 0) ;//将BSImpl信息写到数据包中  
    byte[]data=out.marshall () ;//序列化为一个buffer，然后通过  
    Binder传递  
    out.recycle () ;  
    return data;  
}
```

由此可以看出，电量统计的核心类是BSImpl，下面就来分析它。

2.初识BSImpl

BSImpl功能是进行电量统计，那么是否存在计量工具呢？答案是肯定的，并且BSImpl使用了不止一种计量工具。

(1) 计量工具和统计对象介绍

BSImpl一共使用了4种计量工具，如图5-3所示。

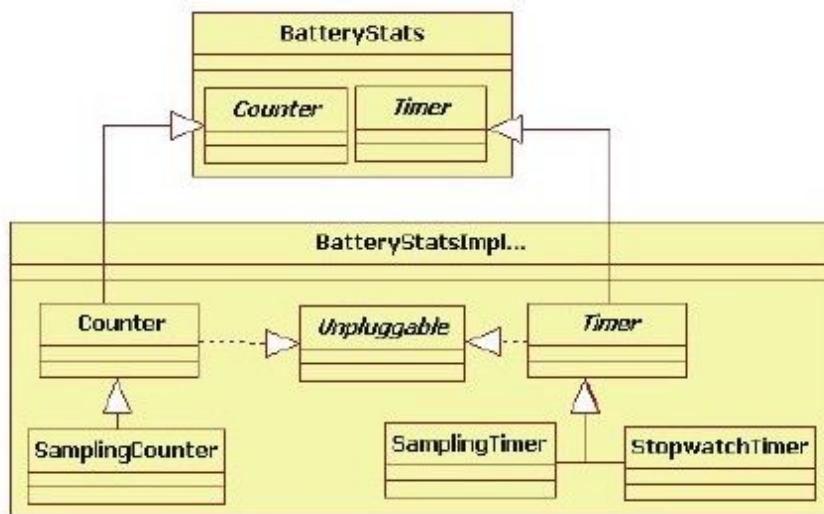


图 5-3 计量工具图例

由图5-3可知：

一共有两大类计量工具，`Counter`用于计数，`Timer`用于计时。

BSImpl实现了`StopwatchTimer`（秒表）、`SamplingTimer`（抽样计时）、`Counter`（计数器）

和SamplingCounter（抽样计数）等4个具体的计量工具。

BSImpl中定义了一个Unpluggable接口。当手机插上USB线充电（不论是由交流电还是由USB供电）时，该接口的plug函数都被调用。反之，当拔去USB线时，该接口的unplug函数被调用。设置这个接口的目的是为了满足BSImpl对各种情况下系统用电量的统计要求。关于Unpluggable接口的作用，在后续内容中能见到。

虽然只有4种计量工具（笔者觉得已经相当多了），但是可以在很多地方使用它们。下面先来认识部分被挂牌要求统计用电量的对象，如表5-5所示。

表5-5中的电量统计项已经够多了吧？还不止这些，为了做到更精确，Android还希望能统计每个进程在各种情况下的耗电量。这是一项庞大的工程，怎么做到的呢？来看下一节的内容。

（2）BatteryStats.Uid介绍

在Android 4.0中，和进程相关的用电量统计并非以单个PID为划分单元，而是以Uid为组，相关类结构如图5-4所示。

表 5-5 用电量统计项

成员变量名	类 型	备 注
mScreenOnTimer	StopwatchTimer	统计屏幕开启耗电量
mScreenBrightnessTimer[]	StopwatchTimer	统计各级屏幕亮度（共 5 级）的耗电量
mInputEventCounter	Counter	统计输入事件耗电量
mPhoneOnTimer	StopwatchTimer	统计通话耗电量
mPhoneSignalStrengthsTimer[]	StopwatchTimer	统计手机信号各级强度耗电量，共 5 级
mPhoneSignalScanningTimer	StopwatchTimer	统计搜索手机信号耗电量
mPhoneDataConnectionsTimer[]	StopwatchTimer	统计手机使用各种数据通信方式（如 GPRS、CDMA 等）的用电量，一共 15 级
mWifiOnTimer	StopwatchTimer	Wifi 用电量（包括使用网络和开启 Wifi 功能却没有使用网络的情况）
mGlobalWifiRunningTimer	StopwatchTimer	使用 Wifi 的用电量
mAudioOnTimer	StopwatchTimer	使用 Audio 的耗电量
mVideoOnTimer	StopwatchTimer	使用 Video 的耗电量

由图5-4可知：

Wakelock 用于统计该Uid 对应进程 使用 WakeLock的用电情况。

Proc 用于统计Uid中某个进程的电量使用情况。

Pkg 用于统计某个特定Package的使用情况，其内部类Serv 用于统计该Pkg中Service的用电情况。

Sensor 用于统计传感器用电情况。

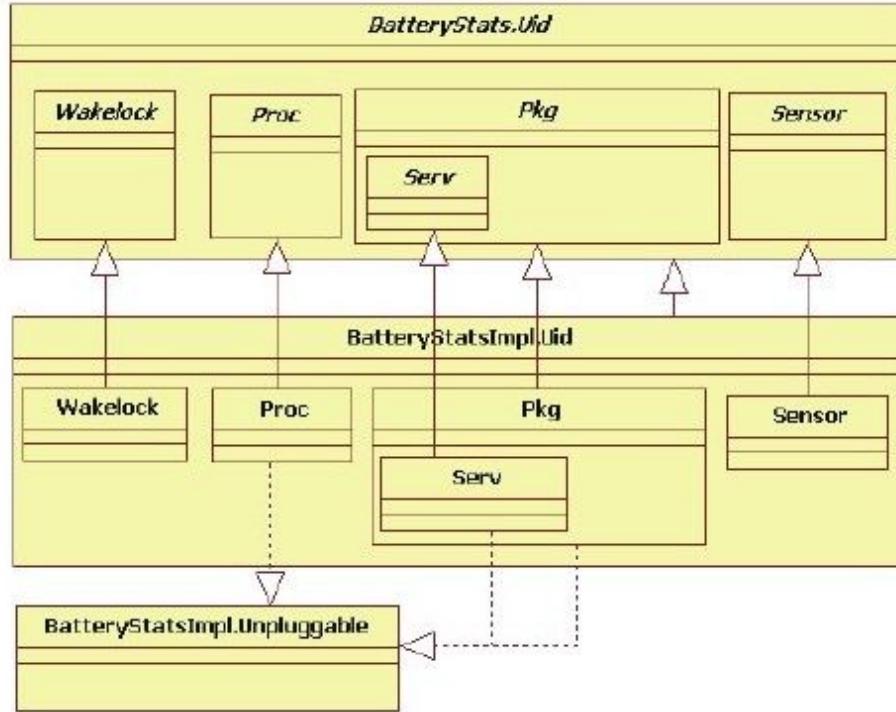


图 5-4 BatteryStats.Uid家族

基于以上的了解，以后分析将会轻松很多，下面来分析它的代码。

3.BSImpl流程分析

(1) 构造函数分析

先分析构造函数，代码如下：

[-->BatteryStatsImpl.java : BatteryStatsImpl]

```

public BatteryStatsImpl (String filename) {
    //JounaledFile为日志文件对象，内部包含两个文件，原始文件和临时文件。
    //目的是双备份，
    //以防止在读写过程中文件信息丢失或出错
}

```

```
mFile=new JournaledFile ( new File ( filename ) , new File  
(filename+".tmp" ) ) ;  
mHandler=new MyHandler () ;//创建一个Handler对象  
mStartCount++ ;  
//创建表5-5中的用电统计项对象  
mScreenOnTimer=new StopwatchTimer ( null , -1 , null,  
mUnpluggables) ;  
for (int i=0 ; i<NUM_SCREEN_BRIGHTNESS_BINS ; i++) {  
    mScreenBrightnessTimer[i]=new StopwatchTimer (null , -100-i,  
null,  
mUnpluggables) ;  
}  
mInputEventCounter=new Counter (mUnpluggables) ;  
.....  
mOnBattery=mOnBatteryInternal=false ;// 设置这两位成员变量为  
false  
initTimes () ;//①初始化统计时间  
mTrackBatteryPastUptime=0 ;  
mTrackBatteryPastRealtime=0 ;  
mUptimeStart=mTrackBatteryUptimeStart=  
SystemClock.uptimeMillis () *1000 ;  
mRealtimeStart=mTrackBatteryRealtimeStart=  
SystemClock.elapsedRealtime () *1000 ;  
mUnpluggedBatteryUptime=getBatteryUptimeLocked  
(mUptimeStart) ;  
mUnpluggedBatteryRealtime=getBatteryRealtimeLocked  
(mRealtimeStart) ;  
mDischargeStartLevel=0 ;  
mDischargeUnplugLevel=0 ;  
mDischargeCurrentLevel=0 ;  
initDischarge () ;//②初始化和电池level有关的成员变量  
clearHistoryLocked () ;//③删除用电统计的历史记录  
}
```

要看懂这段代码比较困难，主要原因是变量太多，并且没有注释说明。只能根据名字来推测了。在以上代码中除了计量工具外，还出现了三大类成员变量：

用于统计时间的成员变量，例如mUptimeStart、mTrackBatteryPastUptime等。这些参数的初始化函数为initTimes。注意，系统时间为uptime和realtime。uptime和realtime的时间起点都从系统启动开始算（since the system was booted），但是uptime不包括系统休眠时间，而realtime包括系统休眠时间^[1]。

用于记录各种情况下用于表电池电量的成员变量，如mDischargeStartLevel、mDischargeCurrentLevel等，这些成员变量的初始化函数为initDischarge。

用于保存历史记录的HistoryItem，在clearHistoryLocked函数中初始化，主要有mHistory、mHistoryEnd等成员变量（这些成员变量在clearHistoryLocked函数中出现）。

上述这些成员变量的具体作用，只有通过后文的分析才能弄清楚。这里先介绍StopwatcherTimer。

```
//调用方式
mPhoneSignalScanningTimer=new StopwatchTimer (null, -200+1,
null, mUnpluggables) ;
//mUnpluggables类型为ArrayList<Unpluggable>, 用于保存插拔USB线
时需要对应更新用电
//信息的统计对象
//StopwatchTimer的构造函数
StopwatchTimer (Uid uid, int type, ArrayList<StopwatchTimer
>timerPool,
ArrayList<Unpluggable>unpluggables) {
//在本例中, uid为0, type为负数, timerPool为空, unpluggables为
mUnpluggables
super (type, unpluggables) ;
mUid=uid ;
mTimerPool=timerPool ;
}
//Timer的构造函数
Timer (int type, ArrayList<Unpluggable>unpluggables) {
mType=type ;
mUnpluggables=unpluggables ;
unpluggables.add (this) ;
}
```

在 StopwatchTimer 中 比 较 难 理 解 的 就 是 unpluggables, 根据注释说明, 当插拔USB线时, 需要更新用电统计的对象, 应该将其加入到 mUnpluggables 数组中。

在启动秒表时, 调用它的 startRunningLocked 函数, 并传入 BSImpl 实例, 代码如下:

：

[--> BatteryStatsImpl.java
StopwatchTimer.startRuningLocked]

```
void startRunningLocked (BatteryStatsImpl stats) {  
    if (mNesting++==0) {//嵌套调用控制  
        //getBatteryRealtimeLocked函数返回总的电池使用时间  
        mUpdateTime=stats.getBatteryRealtimeLocked (  
            SystemClock.elapsedRealtime () *1000) ;  
        if (mTimerPool !=null) {//不讨论这种情况  
        }  
        mCount++ ;  
        mAcquireTime=mTotalTime ;//计数控制, 请读者阅读相关注释说明  
    }  
}
```

当停用秒表时，调用它的stopRunningLocked函数，代码如下：

[-->BatteryStatsImpl.java :
StopwatchTimer.stopRunningLocked]

```
void stopRunningLocked (BatteryStatsImpl stats) {  
    if (mNesting==0) {  
        return ;//嵌套控制  
    }  
    if (--mNesting==0) {  
        if (mTimerPool !=null) {//不讨论这种情况  
        }else{  
            final long realtime=SystemClock.elapsedRealtime () *1000 ;  
            //计算此次启动/停止周期的时间  
            final long batteryRealtime=stats.getBatteryRealtimeLocked  
(realtime) ;  
            mNesting=1 ;  
            //mTotalTime代表从启动开始该秒停表一共记录的时间  
            mTotalTime=computeRunTimeLocked (batteryRealtime) ;  
            mNesting=0 ;
```

```
        }
        if (mTotalTime==mAcquireTime) mCount-- ;
    }
}
```

在StopwatchTimer中定义了很多的时间参数，这些参数用于记录各种时间，例如总耗时、最近一次工作周期的耗时等。如果不是工作需要（例如研究Settings应用中和BatteryInfo相关的内 容），读者仅需了解它的作用即可。

(2) ActivityManagerService和BSS交互

ActivityManagerService创建BSS后，还要进行几项操作，具体代码分别如下：

[-->ActivityManagerService.java :
ActivityManagerService构造函数]

```
mBatteryStatsService=new BatteryStatsService (new File (
    systemDir, "batterystats.bin") .toString () ) ;
//操作通过BSImpl创建JournalizedFile文件
mBatteryStatsService.getActiveStatistics () .readLocked () ;
mBatteryStatsService.getActiveStatistics
() .writeAsyncLocked () ;
//BSImpl的getIsOnBattery返回mOnBattery变量，初始化值为false
mOnBattery=DEBUG_POWER?true
:mBatteryStatsService.getActiveStatistics () .getIsOnBattery
() ;
//设置回调，该回调也用于信息统计，留到介绍ActivityManagerService时
再来分析
```

```
mBatteryStatsService.getActiveStatistics() .setCallback  
(this) ;
```

[-->ActivityManagerService.java : main]

```
m.mBatteryStatsService.publish(context) ;
```

[-->BatteryStatsService.java : publish]

```
public void publish(Context context) {  
    mContext=context;  
    //注意，BSS服务称为batteryinfo，而BatteryService服务称为battery  
    ServiceManager.addService("batteryinfo", asBinder());  
    //PowerProfile见下文解释  
    mStats.setNumSpeedSteps((new PowerProfile  
(mContext).getNumSpeedSteps()));  
    //设置通信信号扫描超时时间  
    mStats.setRadioScanningTimeout((mContext.getResources()  
().getInteger(  
        com.android.internal.R.integer.config_radioScanningTimeout)  
        *1000L));  
}
```

在以上代码中，比较有意思的是PowerProfile类，它将解析Android 4.0源代码/frameworks/base/core/res/res/xml/power_profile.xml文件。此XML文件存储的是各种操作（和硬件相关）的耗电情况，如图5-5所示。

```
<?xml version="1.0" encoding="utf-8"?>
<!-- All values are in mA h except as noted -->
<device name="Android">
    <item name="none">0</item>
    <item name="screen.on">0.1</item>
    <item name="bluetooth.active">0.1</item>
    <item name="bluetooth.on">0.1</item>
    <item name="screen.full">0.1</item>
    <item name="wifi.on">0.1</item>
    <item name="wifi.active">0.1</item>
    <item name="wifi.scan">0.1</item>
    <item name="dsp.audio">0.1</item>
    <item name="dsp.video">0.1</item>
    <item name="radio.active">1</item>
</device>
```

图 5-5 PowerProfile文件示例

由图5-5可知，该文件保存了各种操作的耗电情况，以mA h（毫安时）为单位。Power-Profile的getNumSpeedSteps将返回CPU支持的频率值，目前在该XML中只定义了一个值，即400MHz。

注意 在编译时，各厂家会将特定硬件平台的power_profile.xml复制到输出目录。此处展示的power_profile.xml和硬件平台无关。

(3) BatteryService和BSS交互

BatteryService在它的processValues函数中和BSS交互，processValues函数的代码如下：

[-->BatteryService.java : processValues]

```
private void processValues () {
    .....
    mBatteryStats.setBatteryState (mBatteryStatus,
        mBatteryHealth, mPlugType,
```

```
mBatteryLevel, mBatteryTemperature, mBatteryVoltage) ;  
}
```

BSS 的工作由 BSImpl 来完成，BsImpl 的 setBatteryState 函数的代码如下：

[--> BatteryStatsImpl.java : setBatteryState]

```
public void setBatteryState ( int status, int health, int  
plugType, int level,  
    int temp, int volt) {  
    synchronized (this) {  
        boolean onBattery=plugType==BATTERY_PLUGGED_NONE ; //判断是否为  
电池供电  
        int oldStatus=mHistoryCur.batteryStatus ;  
        .....  
        if (onBattery) {  
            //mDischargeCurrentLevel记录当前使用电池供电时的电池电量  
            mDischargeCurrentLevel=level ;  
            mRecordingHistory=true ; //mRecordingHistory表示需要记录一次历史  
值  
        }  
        //此时，onBattery为当前状态，mOnBattery为历史状态  
        if (onBattery !=mOnBattery) {  
            mHistoryCur.batteryLevel= (byte) level ;  
            mHistoryCur.batteryStatus= (byte) status ;  
            mHistoryCur.batteryHealth= (byte) health ;  
            .....//更新mHistoryCur中的电池信息  
            setOnBatteryLocked (onBattery, oldStatus, level) ;  
        }else{  
            boolean changed=false ;  
            if (mHistoryCur.batteryLevel !=level) {  
                mHistoryCur.batteryLevel= (byte) level ;  
                changed=true ;
```

```
}

.....//判断电池信息是否发生变化
if (changed) { //如果发生变化，则需要增加一次历史记录
addHistoryRecordLocked (SystemClock.elapsedRealtime () ) ;
}
}
if ( ! onBattery & &
status==BatteryManager.BATTERY_STATUS_FULL) {
mRecordingHistory=false ;
}
}
}
```

setBatteryState函数的工作如下：判断当前供电状态是否发生变化，由onBattery和mOnBattery进行比较决定。其中onBattery用于判断当前是否为电池供电，mOnBattery为上次调用该函数时得到的判断值。如果供电状态发生变化（其实就是经历一次USB拔插过程），则调用setOnBatteryLocked函数。如果供电状态未发生变化，则需要判断电池信息是否发生变化，例如电量和电压等。如果发生变化，则调用addHistoryRecordLocked。该函数用于添加一次历史记录。

接下来看setOnBatteryLocked函数的代码：

[-->BatteryStatsImpl.java
setOnBatteryLocked]

```
void setOnBatteryLocked ( boolean onBattery, int oldStatus,
int level) {
    boolean dowrite=false ;
    //发送一个消息给mHandler，将在内部调用ActivityManagerService设置
    的回调函数
    Message m=mHandler.obtainMessage ( MSG_REPORT_POWER_CHANGE ) ;
    m.arg1=onBattery?1:0 ;
    mHandler.sendMessage ( m ) ;
    mOnBattery=mOnBatteryInternal=onBattery ;
    long uptime=SystemClock.uptimeMillis () *1000 ;
    long mSecRealtime=SystemClock.elapsedRealtime () ;
    long realtime=mSecRealtime*1000 ;
    if ( onBattery ) {
        //关于电量信息统计，有一个值得注意的地方：当oldStatus为满电状态，或当
        前电量
        //大于90，或mDischargeCurrentLevel小于20并且当前电量大于80时，要清
        空统计
        //信息，以开始新的统计。也就是说在满足特定条件的情况下，电量使用统计信
        息会清零并重
        //新开始。读者不妨用自己手机试一试
        if ( oldStatus==BatteryManager.BATTERY_STATUS_FULL||level>
=90
            || (mDischargeCurrentLevel<20 && level>=80) ) {
            dowrite=true ;
            resetAllStatsLocked () ;
            mDischargeStartLevel=level ;
        }
        //读取/proc/wakelock文件，该文件反映了系统Wakelock的使用状态，
        //感兴趣的读者可自行研究
        updateKernelWakelocksLocked () ;
        mHistoryCur.batteryLevel= (byte) level ;
        mHistoryCur.states           &           =           ~
HistoryItem.STATE_BATTERY_PLUGGED_FLAG ;
        //添加一条历史记录
        addHistoryRecordLocked (mSecRealtime) ;
        //mTrackBatteryUptimeStart表示使用电池的开始时间，由uptime表示
```

```
mTrackBatteryUptimeStart=uptime ;
//mTrackBatteryRealtimeStart表示使用电池的开始时间，由realtime表示
mTrackBatteryRealtimeStart=realtime ;
//mUnpluggedBatteryUptime记录总的电池使用时间（不论中间插拔多少次）
mUnpluggedBatteryUptime=getBatteryUptimeLocked (uptime) ;
//mUnpluggedBatteryRealtime记录总的电池使用时间
mUnpluggedBatteryRealtime=getBatteryRealtimeLocked
(realtime) ;
//记录电量
mDischargeCurrentLevel=mDischargeUnplugLevel=level ;
if (mScreenOn) {
    mDischargeScreenOnUnplugLevel=level ;
    mDischargeScreenOffUnplugLevel=0 ;
}else{
    mDischargeScreenOnUnplugLevel=0 ;
    mDischargeScreenOffUnplugLevel=level ;
}
mDischargeAmountScreenOn=0 ;
mDischargeAmountScreenOff=0 ;
//调用doUnplugLocked函数
doUnplugLocked (mUnpluggedBatteryUptime,
mUnpluggedBatteryRealtime) ;
}else{
.....//处理使用USB充电的情况，请读者在上面讨论的基础上自行分析
}
.....//记录信息到文件
}
```

doUnplugLocked函数将更新对应信息，该函数比较简单，无须赘述。另外，addHistoryRecordLocked函数用于增加一条历史记录（由HistoryItem表示），读者也可自行研究。

从本节的分析可知，Android将电量统计分得非常细，例如由电池供电的情况要统计，由USB/AC充电的情况也要统计，因此有了setBatteryState函数。

(4) PowerManagerService和BSS交互

PMS 和 BSS 交互是最多的，此处以noteScreenOn和noteUserActivity为例，来介绍BSS到底是如何统计电量的。

先来看noteScreenOn函数。当开启屏幕时，PMS会调用BSS的noteScreenOn以通知屏幕开启，该函数在内部调用BSImpl的noteScreenOnLocked，其代码如下：

[-->BatteryStatsImpl.java :
noteScreenOnLocked]

```
public void noteScreenOnLocked () {
    if (!mScreenOn) {
        mHistoryCur.states|=HistoryItem.STATE_SCREEN_ON_FLAG ;
        //增加一条历史记录
        addHistoryRecordLocked (SystemClock.elapsedRealtime () ) ;
        mScreenOn=true ;
        //启动mScreenOnTime秒停表，其工作就是记录时间，读者可自行研究其内部实现
        mScreenOnTimer.startRunningLocked (this) ;
        if (mScreenBrightnessBin>=0) //启动对应屏幕亮度的秒停表（参考表5-5）
```

```
mScreenBrightnessTimer[mScreenBrightnessBin].startRunningLocked (this) ;  
    //屏幕开启也和内核WakeLock有关，所以这里一样要更新WakeLock的用电统计  
    noteStartWakeLocked (-1, -1, "dummy", WAKE_TYPE_PARTIAL) ;  
    if (mOnBatteryInternal)  
        updateDischargeScreenLevelsLocked (false, true) ;  
    }  
}
```

再来看noteUserActivity，当有输入事件触发PMS的userActivity时，该函数被调用，代码如下：

[-->BatteryStatsImpl.java :
noteUserActivityLocked]

```
//BSS的noteUserActivity将调用BSImpl的noteUserActivityLocked  
public void noteUserActivityLocked (int uid, int event) {  
    getUidStatsLocked (uid) .noteUserActivityLocked (event) ;  
}
```

先是调用getUidStatsLocked以获取一个Uid对象，如果该Uid是首次出现的，则要在内部创建一个Uid对象。下面直接介绍Uid的noteUserActivityLocked函数：

[-->BatteryStatsImpl.java : Uid :
noteUserActivityLocked]

```
public void noteUserActivityLocked (int type) {
```

```
if (mUserActivityCounters==null) {  
    initUserActivityLocked () ;  
}  
if (type<0) type=0 ;  
else if (type>=NUM_USER_ACTIVITY_TYPES)  
    type=NUM_USER_ACTIVITY_TYPES-1 ;  
//noteUserActivityLocked只是调用对应type的Counter的stepAtomic函  
数  
//每个Counter内部都有个计数器，stepAtomic使该计数器增1  
mUserActivityCounters[type].stepAtomic () ;  
}
```

mUserActivityCounters 为一个 7 元 Counter 数组，该数组对应 7 种不同的输入事件类型，在代码中，由 BSImpl 的 成 员 变 量 USER_ACTIVITY_TYPES 表示，如下所示：

```
static final String[]USER_ACTIVITY_TYPES={  
    "other", "cheek", "touch", "long_touch", "touch_up", "button"  
, "unknown"  
};
```

另外，在 LocalPowerManager 中，也定义了相关的 type 值，如下所示：

[-->LocalPowerManager.java]

```
public interface LocalPowerManager{  
    public static final int OTHER_EVENT=0 ;  
    public static final int BUTTON_EVENT=1 ;  
    public static final int TOUCH_EVENT=2 ; //目前只使用这3种事件  
    ....
```

}

[1]读者可阅读SDK文档中关于SystemClock类的说明。

5.5.3 BatteryService及BatteryStatsService 总结

本节重点讨论了BatteryService和BatteryStatsService。其中，BatteryService和系统中的供电系统交互，通过它可获取电池状态等信息。而BatteryStatsService用于统计系统用电量的情况。就难度而言，BSS较为复杂，原因是Android试图对系统耗电量作非常详细的统计，导致统计项非常繁杂。另外，电量统计大多采用被动通知的方式（即需要其他服务主动调用BSS提供的noteXXXOn/noteXXXOff函数），这种实现方法一方面加重了其他服务的负担，另一方面影响了这些服务未来功能的扩展。

注意 虽然Google费尽心血来完善电量统计，但这并不是解决耗电量大的根本途径。读者可分析Settings程序中电量统计图的绘制以加深对各种统计对象的理解。Settings中和电量相关的文件在Android 4.0源代码的/packages/apps/Settings/src/com/android/settings/fuelgauge/目录中。

5.6 本章学习指导

本章最难的部分其实在BSS中，PMS和BatteryService相对比较简单。在这3项服务中，PMS是核心。读者在研究PMS时，要注意把握以下几个方面：

PMS的初期工作流程，即构造函数、init函数、systemReady函数和BootCompleted函数等。

PMS功能为根据当前系统状态（包括mUserState和mWakeLockState）去操作屏幕和灯光。而触发状态改变的有WakeLock的获取和释放、userActivity函数的调用，因此读者也要搞清楚PMS在这两个方面的工作原理。

PMS还有一部分功能和传感器有关，其功能无非还是根据状态操作屏幕和灯光。除非工作需要，否则只需要简单了解这部分的工作流程即可。

对BSS来说，复杂之处在于它定义了很多成员变量和数据类型，并且没有一份电量统计标准的说明文档，因此笔者认为，读者只要搞清楚那几个计量工具和各个统计项的作用即可，如果在

其他服务的代码中看到和BSS交互的函数，那么只需知道原因和目的即可。

另外，电源管理需要HAL层和Linux内核提供支持，感兴趣的读者不妨以本章知识为切入点，对底层技术进行一番深入剖析。

5.7 本章小结

电源管理系统的核 心 是 PowerManagerService，还包 括 BatteryService 和 BatteryStats-Service。本章对 Android 平台中的电源管理系统进行了较详细的分析，其中：

对于 PMS，本章分析了它的初始化流程、WakeLock 获取流程、userActivity 函数的工作流程及 Power 按键处理流程。

BatteryService 功能较为简单，读者大概了解即可。

对于 BatteryStatsService，本章对它内部的数据结构、统计对象等进行了较详细的介绍，并对其工作流程展开了分析。建议读者结合 Settings 应用中的相关代码，加深对其中各种计量工具及统计对象的理解。

第6章 深入理解 ActivityManagerService

本章主要内容：

详细分析ActivityManagerService。

本章所涉及的源代码文件名及位置：

SystemServer. java
(frameworks/base/services/java/com/android/server/
SystemServer.java)

ActivityManagerService. java
(frameworks/base/services/java/com/android/server/
am/ActivityManagerService.java)

ContextImpl. java
(frameworks/base/core/java/android/app/ContextIm
pl.java)

ActivityThread. java
(frameworks/base/core/java/android/app/ActivityTh
read.java)

ActivityStack. java
(frameworks/base/services/java/com/android/server/

am/ActivityStack.java)

Am. java
(frameworks/base/cmds/am/src/com/android/comma
nds/am/Am.java)

ProcessRecord. java
(frameworks/base/services/java/com/android/server/
am/ProcessRecord.java)

ProcessList. java
(frameworks/base/services/java/com/android/server/
am/ProcessList.java)

RuntimeInit. java
(frameworks/base/core/java/com/android/internal/os/
/RuntimeInit.java)

6.1 概述

相信绝大部分读者对本书提到的ActivityManagerService（以下简称AMS）都有所耳闻。AMS是Android中最核心的服务，主要负责系统中四大组件的启动、切换、调度及应用进程的管理和调度等工作，其职责与操作系统中的进程管理和调度模块类似，因此它在Android中非常重要。

AMS是本书碰到的第一块“难啃的骨头”^[1]，涉及的知识点较多。为了帮助读者更好地理解AMS，本章将带领读者按5条不同的线来分析它。

第一条线：同其他服务一样，将分析system_server中AMS的调用轨迹。

第二条线：以am命令启动一个Activity为例，分析应用进程的创建、Activity的启动，以及它们和AMS之间的交互等知识。

第三条线和第四条线：分别以Broadcast和Service为例，分析AMS中Broadcast和Service的相关处理流程。其中，Service的分析将只给出流程

图，希望读者能在前面学习的基础上自己分析并掌握相关知识。

第五条线：以一个Crash的应用进程为出发点，分析AMS如何打理该应用进程的“身后事”。

除了这5条线外，还将统一分析在这5条线中频繁出现的与AMS中应用进程的调度、内存管理等相关知识。

提示 ContentProvider将放到下一章分析，但是本章将涉及和ContentProvider有关的知识点。

先来看AMS的家族图谱，如图6-1所示。

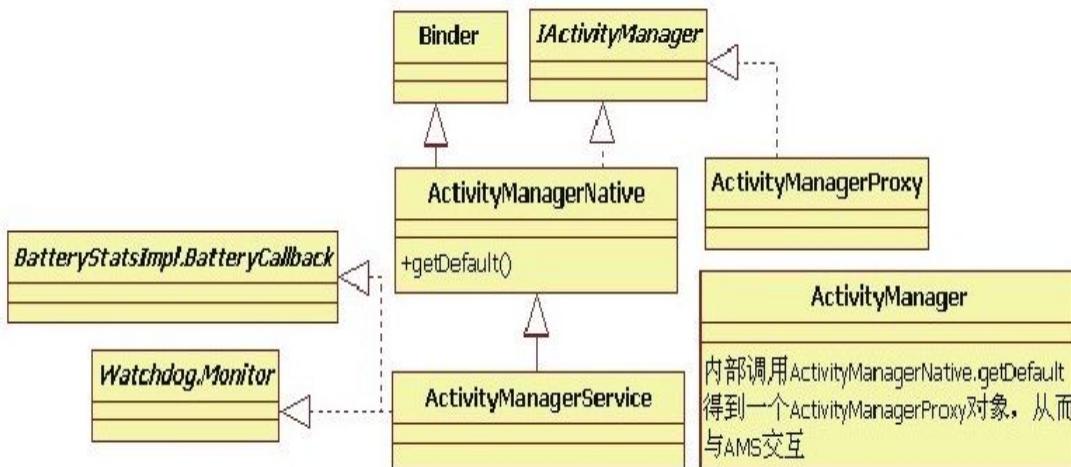


图 6-1 AMS家族图谱

由图6-1可知：

AMS 由 ActivityManagerNative（以后简称 AMN）类派生，并实现 Watchdog.Monitor 和 BatteryStatsImpl.BatteryCallback 接口。而 AMN 由 Binder 派生，实现了 IActivityManager 接口。

客户端使用 ActivityManager 类。由于 AMS 是系统核心服务，很多 API 不能开放供客户端使用，因此设计者没有让 ActivityManager 直接加入 AMS 家族。ActivityManager 类内部通过调用 AMN 的 getDefault 函数得到一个 ActivityManagerProxy 对象，通过它可与 AMS 通信。

下面具体分析 AMS。相信不少读者已经磨拳擦掌，跃跃欲试了。

提示 读者最好在桌上放一杯清茶，以保持 AMS 分析旅途中头脑清醒。

[1]AMS 较复杂，其中一个原因是其功能较多，另一个原因是它的代码质量及结构并不出彩。

6.2 初识ActivityManagerService

AMS 由 system_server 的 ServerThread 线程创建，提取它的调用轨迹，代码如下：

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

```
//①调用main函数，得到一个Context对象
context=ActivityManagerService.main (factoryTest) ;
//②setSystemProcess：这样system_server进程可加到AMS中，并被它管理
ActivityManagerService.setSystemProcess () ;
// ③ installSystemProviders：将 SettingsProvider 放 到
system_server进程中来运行
ActivityManagerService.installSystemProviders () ;
//④在内部保存WindowManagerService（以后简称WMS）
ActivityManagerService.self () .set WindowManager (wm) ;
//⑤和WMS交互，弹出“启动进度”对话框
ActivityManagerNative.getDefault () .showBootMessage (
context.getResources () .getText (
//该字符串中文意思为：正在启动应用程序
com.android.internal.R.string.android_upgrading_starting_app
s) ,
false) ;
//⑥AMS是系统的核心，只有它准备好后，才能调用其他服务的systemReady
//注意，有少量服务在AMS systemReady之前就绪，它们不影响此处的分析
ActivityManagerService.self () .systemReady (new Runnable ()
{
    public void run () {
        startSystemUi (contextF) ; //启动systemUi。如此，状态栏就准备好了
    }
}) ;
```

```
if (batteryF !=null) batteryF.systemReady () ;  
if (networkManagementF !=null) networkManagementF.systemReady  
() ;  
.....  
watchdog.getInstance () .start () ;//启动watchdog  
.....//调用其他服务的systemReady函数  
}
```

在以上代码中，一共列出了6个重要调用及这些调用的简单说明，本节将分析除与WindowManagerService（以后简称WMS）交互的④、⑤外的其余四项调用。

先来分析①处调用。

6.2.1 ActivityManagerService的main函数分析

AMS的main函数将返回一个Context类型的对象，该对象在system_server中被其他服务大量使用。Context，顾名思义，代表了一种上下文环境（笔者觉得其意义和JNIEnv类似），有了这个环境，我们可以做很多事情（例如获取该环境中的资源、Java类信息等）。那么AMS的main将返回一个怎样的上下文环境呢？来看以下代码：

[-->ActivityManagerService.java : main]

```
public static final Context main (int factoryTest) {  
    AThread thr=new AThread () ;//①创建一个AThread线程对象  
    thr.start () ;  
    .....//等待thr创建成功  
    ActivityManagerService m=thr.mService ;  
    mSelf=m ;  
    //②调用ActivityThread的systemMain函数  
    ActivityThread at=ActivityThread.systemMain () ;  
    mSystemThread=at ;  
    //③得到一个Context对象，注意调用的函数名为getSystemContext，何为  
    System Context  
    Context context=at.getSystemContext () ;  
    context.setTheme (android.R.style.Theme_Holo) ;  
    m.mContext=context ;  
    m.mFactoryTest=factoryTest ;  
    //ActivityStack是AMS中用来管理Activity的启动和调度的核心类，以后再  
    分析它  
    m.mMainStack=new ActivityStack (m, context, true) ;  
    //调用BSS的publish函数，这个知识点我们在第5章中介绍过了  
    m.mBatteryStatsService.publish (context) ;  
    //另外一个service：UsageStatsService。读者阅读完本章后自行分析它  
    m.mUsageStatsService.publish (context) ;  
    synchronized (thr) {  
        thr.mReady=true ;  
        thr.notifyAll () ;//通知thr线程，本线程工作完成  
    }  
    //④调用AMS的startRunning函数  
    m.startRunning (null, null, null, null) ;  
    return context ;  
}
```

在main函数中，我们又列出了4个关键函数，分别是：

创建 AThread 线程。虽然 AMS 的 main 函数由 ServerThread 线程调用，但是 AMS 自己的工作并没有放在 ServerThread 中去做，而是新创建了一个线程，即 AThread 线程。

ActivityThread. systemMain 函数。初始化 ActivityThread 对象。

ActivityThread. getSystemService 函数。用于获取一个 Context 对象，从函数名上看，该 Context 代表了 System 的上下文环境。

AMS 的 startRunning 函数。

注意，main 函数中有一处等待（wait）、一处通知（notifyAll），这是因为：

main 函数首先需要等待 AThread 所在线程启动并完成一部分工作。

AThread 完成那一部分工作后，将等待 main 函数完成后续的工作。

这种双线程互相等待的情况，在 Android 代码中比较少见，读者只需了解它们的目的即可。下边来分析以上代码中的第一个关键点。

1. AThread 分析

(1) AThread分析

AThread的代码如下：

[-->ActivityManagerService.java : AThread]

```
static class AThread extends Thread{//AThread从Thread类派生
ActivityManagerService mService ;
boolean mReady=false ;
public AThread () {
super ("ActivityManager") ;//线程名就叫ActivityManager
}
public void run () {
Looper.prepare () ;//看来，AThread线程将支持消息循环
android.os.Process.setThreadPriority （//设置线程优先级
android.os.Process.THREAD_PRIORITY_FOREGROUND） ;
android.os.Process.setCanSelfBackground (false) ;
//创建AMS对象
ActivityManagerService m=new ActivityManagerService () ;
synchronized (this) {
mService=m；//为AThread内部成员变量mService赋值，指向AMS
notifyAll () ;//通知main函数所在线程
}
synchronized (this) {
while (! mReady) {
try{
wait () ;//等待main函数所在线程的notifyAll
}.....
}
}
Looper.loop () ;//进入消息循环
}
}
```

从本质上说，AThread是一个支持消息循环及处理的线程，其主要工作就是创建AMS对象，然后通知AMS的main函数。这样看来，main函数等待的就是这个AMS对象。

(2) AMS的构造函数分析

AMS的构造函数的代码如下：

[-->ActivityManagerService.java :
ActivityManagerService]

```
private ActivityManagerService () {  
    File dataDir=Environment.getDataDirectory () ;//指向/data/目录  
    File systemDir=new File ( dataDir , "system" ) ;// 指向/data/system/目录  
    systemDir.mkdirs () ;//创建/data/system/目录  
    //创建BatteryStatsService (以后简称BSS) 和UsageStatsService (以后简称USS)  
    //我们在分析PowerManageService时已经见过BSS了  
    mBatteryStatsService=new BatteryStatsService (new File (  
        systemDir, "batterystats.bin" ) .toString () ) ;  
    mBatteryStatsService.getActiveStatistics () .readLocked () ;  
    mBatteryStatsService.getActiveStatistics  
() .writeAsyncLocked () ;  
    mOnBattery=DEBUG_POWER?true  
    :mBatteryStatsService.getActiveStatistics () .getIsOnBattery  
() ;  
    mBatteryStatsService.getActiveStatistics ( ) .setCallback  
(this) ;  
    //创建USS  
    mUsageStatsService=new UsageStatsService (new File (
```

```
    systemDir, "usagestats") .toString () ) ;  
    //获取OpenGL版本  
    GL_ES_VERSION=SystemProperties.getInt  
    ("ro.opengles.version",  
     ConfigurationInfo.GL_ES_VERSION_UNDEFINED) ;  
    //mConfiguration类型为Configuration，用于描述资源文件的配置属性，  
    例如  
    //字体、语言等  
    mConfiguration.setToDefaults () ;  
    mConfiguration.locale=Locale.getDefault () ;  
    //mProcessStats为ProcessStats类型，用于统计CPU、内存等信息。其内部  
    工作原理就是  
    //读取并解析/proc/stat文件的内容。该文件由内核生成，用于记录kernel及  
    system  
    //一些运行时的统计信息。读者可在Linux系统上通过man proc命令查询详细信  
    息  
    mProcessStats.init () ;  
    //解析/data/system/packages-compat.xml文件，该文件用于存储那些需  
    要考虑屏幕尺寸  
    //的APK的一些信息。读者可参考AndroidManifest.xml中compatible-  
    screens相关说明。  
    //当APK所运行的设备不满足要求时，AMS会根据设置的参数以采用屏幕兼容的方  
    式去运行它  
    mCompatModePackages=new CompatModePackages ( this,  
    systemDir ) ;  
    Watchdog.getInstance () .addMonitor (this) ;  
    //创建一个新线程，用于定时更新系统信息（和mProcessStats交互）  
    mProcessStatsThread=new Thread ("ProcessStats") {.....//先略去该  
    段代码}  
    mProcessStatsThread.start () ;  
}
```

AMS的构造函数比想象得要简单些，下面回
顾一下它的工作：

创建BSS、USS、mProcessStats（ProcessStats类型）、mProcessStatsThread线程，这些都与系统运行状况统计相关。

创建 /data/system 目录，为 mCompatModePackages（CompatModePackages 类型）和 mConfiguration（Configuration 类型）等成员变量赋值。

AMS main函数的第一个关键点就分析到此，接下来来分析它的第二个关键点。

2.ActivityThread.systemMain函数分析

ActivityThread是Android Framework中一个非常重要的类，它代表一个应用进程的主线程（对于应用进程来说，ActivityThread的main函数确实是该进程的主线程执行），其职责就是调度及执行在该线程中运行的四大组件。

注意 应用进程指那些运行APK的进程，它们由Zyote派生（fork）而来，上面运行了dalvik虚拟机。与应用进程相对的就是系统进程（包括Zygote和system_server）。

另外，读者须将“应用进程和系统进程”与“应用APK和系统APK”的概念区分开来。对APK的判

别依赖其文件所在位置（如果APK文件在/data/app目录下，则为应用APK）。

ActivityThread. systemMain函数代码如下：

[-->ActivityThread.java : systemMain]

```
public static final ActivityThread systemMain () {  
    HardwareRenderer.disable (true) ;//禁止硬件渲染加速  
    //创建一个ActivityThread对象，其构造函数非常简单  
    ActivityThread thread=new ActivityThread () ;  
    thread.attach (true) ;//调用它的attach函数，注意传递的参数为true  
    return thread ;  
}
```

在分析ActivityThread的attach函数之前，先提一个问题供读者思考：前面所说的ActivityThread代表应用进程（其上运行了APK）的主线程，而system_server并非一个应用进程，那么为什么此处也需要ActivityThread呢？

还记得在PackageManagerService分析中提到的framework-res.apk吗？这个APK除了包含资源文件外，还包含一些Activity（如关机对话框），这些Activity实际上运行在system_server进程中^[1]。从这个角度看，system_server是一个特殊的应用进程。

通过ActivityThread可以把Android系统提供的组件之间的交互机制和交互接口（如利用Context提供的API）也拓展到system_server中使用。

提示 解答这个问题，对于理解system_server中各服务的交互方式是尤其重要的。

下面来看ActivityThread的attach函数。

(1) attach函数分析

[-->ActivityThread.java : attach]

```
private void attach (boolean system) {  
    sThreadLocal.set (this) ;  
    mSystemThread=system ; //判断是否为系统进程  
    if (!system) {  
        .....//应用进程的处理流程  
    }else{//系统进程的处理流程，该情况只在system_server中处理  
        //设置DDMS时看到的system_server进程名为system_process  
        android.ddm.DdmHandleAppName.setAppName ("system_process") ;  
        try{  
            //ActivityThread的几员“大将”出场，见后文的分析  
            mInstrumentation=new Instrumentation () ;  
            ContextImpl context=new ContextImpl () ;  
            //初始化context，注意第一个参数值为getSystemContext  
            context.init ( getSystemContext ( ) .mPackageName, null,  
this) ;  
            Application app;//利用Instrumentation创建一个Application对象  
            Instrumentation.newApplication ( Application.class,  
context) ;  
            //一个进程支持多个Application，mAllApplications用于保存该进程中的  
            //Application对象
```

```
mAllApplications.add (app) ;  
mInitialApplication=app ;//设置mInitialApplication  
app.onCreate () ;//调用Application的onCreate函数  
}.....//try/catch结束  
}//if (! system) 判断结束  
//注册Configuration变化的回调通知  
ViewRootImpl.addConfigCallback (new ComponentCallbacks2 () {  
    public void onConfigurationChanged ( Configuration  
newConfig) {  
        .....//当系统配置发生变化（如语言切换等）时，需要调用该回调  
    }  
    public void onLowMemory () {}  
    public void onTrimMemory (int level) {}  
}) ;  
}
```

attach函数中出现了几个重要成员，其类型分别是 Instrumentation 类、 Application 类及 Context 类，它们的作用如下（为了保证准确，这里先引用Android的官方说明）。

Instrumentation : Base class for implementing application instrumentation code. When running with instrumentation turned on, this class will be instantiated for you before any of the application code, allowing you to monitor all of the interaction the system has with the application. An Instrumentation implementation is described to the system through an AndroidManifest.xml's < instrumentation > tag. 大意是：Instrumentation 是一

个工具类。当它被启用时，系统先创建它，再通过它来创建其他组件。另外，系统和组件之间的交互也将通过 Instrumentation 来传递，这样，Instrumentation 就能监测系统和这些组件的交互情况了。在实际使用中，我们可以创建 Instrumentation 的派生类来进行相应的处理。读者可查询 Android 中 Junit 的使用来了解 Instrumentation 的作用。本书不讨论 Instrumentation 方面的内容。

Application : Base class for those who need to maintain global application state. You can provide your own implementation by specifying its name in your `AndroidManifest.xml`'s `<application>` tag, which will cause that class to be instantiated for you when the process for your application/package is created. 大意是：Application 类保存了一个全局的 application 状态。Application 由 `AndroidManifest.xml` 中的 `<application>` 标签声明。在实际使用时需定义 Application 的派生类。

Context : Interface to global information about an application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching

activities, broadcasting and receiving intents, etc. 大意是：Context是一个接口，通过它可以获取并操作Application 对应的资源、类，甚至包含于Application中的四大组件。

提示 此处的Application是Android中的一个概念，可理解为一种容器，其内部包含四大组件。另外，一个进程可以运行多个Application。

Context是一个抽象类，而由AMS创建的是它的子类ContextImpl。如前所述，Context提供了Application的上下文信息，这些信息是如何传递给Context的呢？此问题包括两个方面：

Context本身是什么？

Context背后所包含的上下文信息又是什么？

下面来关注上边代码中调用的getSystemContext函数。

(2) getSystemContext函数分析

getSystemContext函数的实现代码如下：

[-->ActivityThread.java : getSystemContext]

```
public ContextImpl getSystemContext () {  
    synchronized (this) {
```

```
if (mSystemContext==null) { //单例模式
    ContextImpl context=ContextImpl.createSystemContext (this) ;
    //LoadedApk是Android2.3引入的一个新类，代表一个加载到系统中的APK
    LoadedApk info=new LoadedApk (this , "android" , context,
null,
    CompatibilityInfo.DEFAULT_COMPATIBILITY_INFO) ;
    //初始化该ContextImpl对象
    context.init (info, null, this) ;
    //初始化资源信息
    context.getResources () .updateConfiguration (
        getConfiguration () , getDisplayMetricsLocked (
        CompatibilityInfo.DEFAULT_COMPATIBILITY_INFO, false) ) ;
    mSystemContext=context; //保存这个特殊的ContextImpl对象
}
}
return mSystemContext ;
}
```

以上代码无非是先创建一个ContextImpl，然后再将其初始化（调用init函数）。为什么函数名是getSystemContext呢？因为在初始化ContextImpl时使用了一个LoadedApk对象。如注释中所说，LoadedApk是Android 2.3引入的一个类，该类用于保存一些和APK相关的信息（如资源文件位置、JNI库位置等）。在getSystemContext函数中初始化ContextImpl的这个LoadedApk所代表的package，名为“android”，也就是framework-res.apk，由于该APK仅供system_server进程使用，所以此处称为getSystemContext。

上面这些类的关系比较复杂，可通过图6-2展示它们之间的关系。

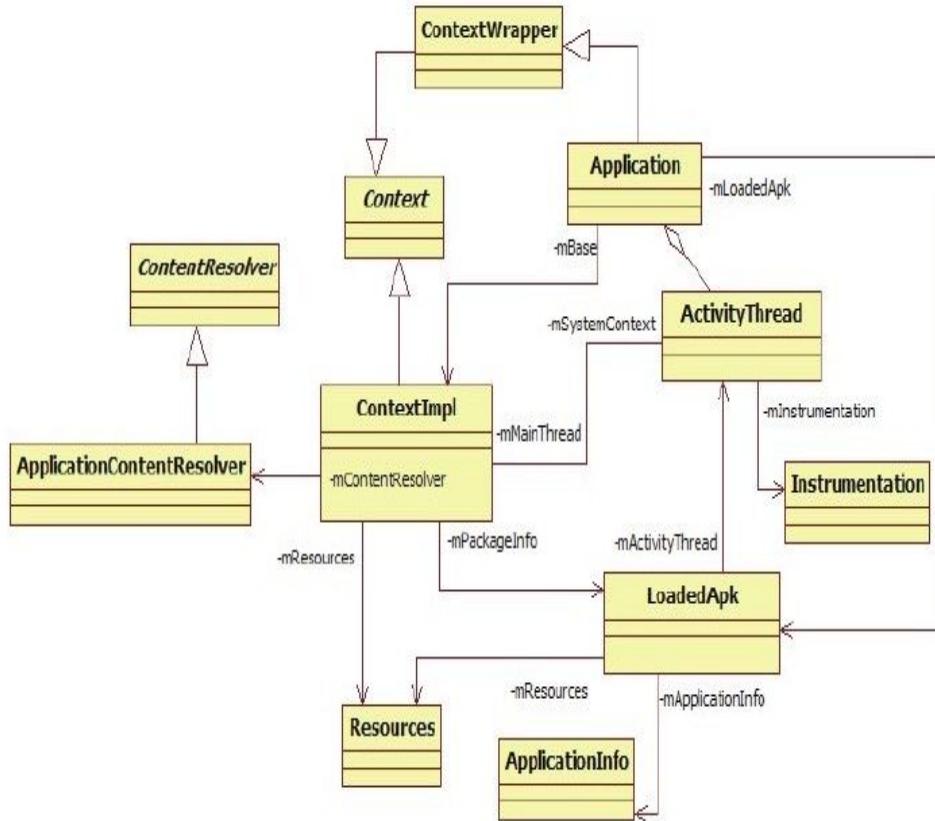


图 6-2 ContextImpl和它的“兄弟”们

由图6-2可知：

先来看派生关系，**ApplicationContentResolver**从**ContentResolver**派生，它主要用于和**ContentProvider**打交道。**ContextImpl**和**ContextWrapper**均从**Context**继承，而**Application**则从**ContextWrapper**派生。

从涉及的面来看，ContextImpl涉及的面最广。它通过mResources指向Resources，通过mPackageInfo指向LoadedApk，通过mMainThread指向ActivityThread，通过mContentResolver指向ApplicationContentResolver。

ActivityThread代表主线程，它通过mInstrumentation指向Instrumentation。另外，它还保存多个Application对象。

注意 在函数中有些成员变量的类型为其基类的类型，而在图6-2中直接指向了实际类型。

(3) systemMain函数总结

systemMain函数调用结束后，我们得到了什么？

得到一个ActivityThread对象，它代表应用进程的主线程。

得到一个Context对象，它背后所指向的Application环境与framework-res.apk有关。费了如此大的工夫，systemMain函数的目的到底是什么？一针见血地说：systemMain函数将为system_server进程搭建一个和应用进程一样的Android运行环境。这句话涉及两个概念。

进程：来源于操作系统，是在OS中看到的运行体。我们编写的代码一定要运行在一个进程中。

Android运行环境：Android努力构筑了一个自己的运行环境。在这个环境中，进程的概念被模糊化了。组件的运行及它们之间的交互均在该环境中实现。

Android运行环境是构建在进程之上的。有Android开发经验的读者可能会发现，应用程序一般只和Android运行环境交互。基于同样的道理，system_server希望它内部的那些service也通过Android运行环境交互，因此也需为它创建一个运行环境。由于system_server的特殊性，此处调用了systemMain函数，而普通的应用进程将在主线程中调用ActivityThread的main函数来创建Android运行环境。

另外，ActivityThread虽然本意是代表进程的主线程，但是作为一个Java类，它的实例到底由什么线程创建，恐怕不是ActivityThread自己能做主的，所以在system_server中可以发现，ActivityThread对象由其他线程创建，而在应用进程中，ActivityThread将由主线程来创建。

3.ActivityThread.getSystemContext函数分析

ActivityThread.getSystemContext函数在上一节已经见过了。调用该函数后，将得到一个代表系统进程的Context对象。到底什么是Context？先来看图6-3所示的Context家族图谱。

注意 该族谱成员并未完全列出。另外，Activity、Service和Application所实现的接口也未画出。

由图6-3可知：

ContextWrapper比较有意思，其在SDK中的说明为 Proxying implementation of Context that simply delegates all of its calls to another Context. Can be subclassed to modify behavior without changing the original Context. 大概意思是：ContextWrapper是一个代理类，被代理的对象是另外一个Context。在图6-3中，被代理的类其实是ContextImpl，由ContextWrapper通过mBase成员变量指定。读者可查看Context-Wrapper.java，其内部函数功能的实现最终都由mBase完成。这样设计的目的是想把ContextImpl隐藏起来。

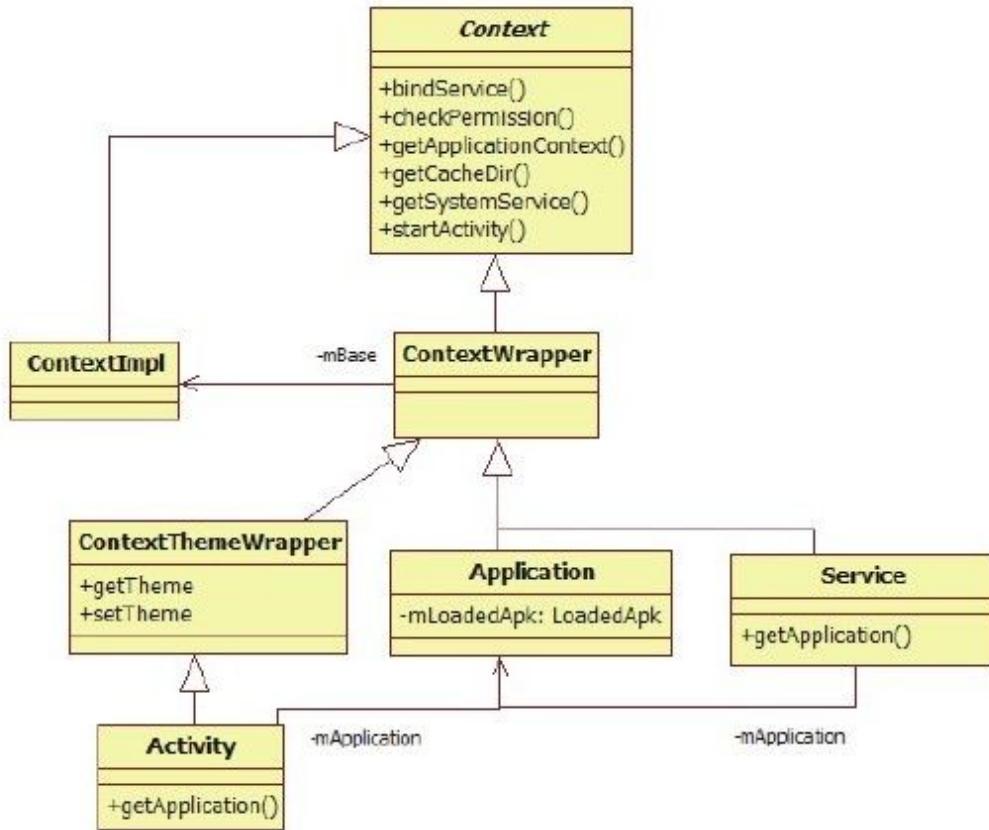


图 6-3 Context家族图谱

Application从ContextWrapper派生，并实现了ComponentCallbacks2接口。Application中有一个LoadedApk类型的成员变量mLoadedApk。LoadedApk代表一个APK文件。由于一个AndroidManifest.xml文件只能声明一个Application标签，所以一个Application必然会和一个LoadedApk绑定。

Service从ContextWrapper派生，其中Service内部成员变量mApplication指向Application（在AndroidManifest.xml中，Service只能作为

Application的子标签，所以在代码中Service必然会和一个Application绑定）。

ContextThemeWrapper 重载了 和 Theme（主题）相关的两个函数。这些和界面有关，所以Activity作为Android系统中的UI容器，必然也会从ContextThemeWrapper 派生。与 Service 一样，Activity 内部也通过 mApplication 成员变量指向 Application。对 Context 的分析先到这里，再来分析第三个关键函数 startRunning。

4.AMS的startRunning函数分析

startRunning函数的实现代码如下：

[-->ActivityManagerService.java :
startRunning]

```
//注意调用该函数时所传递的4个参数全为null
public final void startRunning ( String pkg, String cls,
String action,
String data) {
    synchronized (this) {
        if (mStartRunning) return ; //如果已经调用过该函数，则直接返回
        mStartRunning=true ;
        //mTopComponent最终赋值为null
        mTopComponent=pkg !=null &&cls !=null
        ?new ComponentName (pkg, cls) :null ;
        mTopAction=action !=null?action :Intent.ACTION_MAIN ;
        mTopData=data ; //mTopData最终为null
```

```
    if ( ! mSystemReady) return ; //此时mSystemReady为false，所以直接  
    返回  
    }  
    systemReady (null) ; //这个函数很重要，可惜不在本次startRunning中  
    调用  
}
```

startRunning函数很简单，此处不赘述。

至此，AMS的main函数所涉及的4个知识点已全部分析完。下面回顾一下AMS的main函数的工作。

5.ActivityManagerService的main函数总结

AMS的main函数的目的有两个：

第一个也是最容易想到的目的是创建AMS对象。

另外一个目的比较隐晦，但是非常重要，那就是创建一个供system_server进程使用的Android运行环境。

根据目前所分析的代码，Android运行环境将包括两个类成员：ActivityThread和Context-Impl（一般用它的基类Context）。

图6-4展示了在这两个类中定义的一些成员变量，通过它们可看出ActivityThread及Context-Impl

的作用。

ActivityThread	ContextImpl
-mSystemContext: ContextImpl -mLooper: Looper -mServices: HashMap<IBinder, Service> -mInitialApplication: Application -mAllApplications: ArrayList<Application> -mActivities: HashMap<IBinder, ActivityClientRecord>	-mResources: Resources -mPackageInfo: LoadedApk -mMainThread: ActivityThread -mDatabasesDir: File -mFilesDir: File

图 6-4 ActivityThread和ContextImpl中的部分成员变量

由图6-4可知：

ActivityThread 中有一个 mLooper 成员，它代表一个消息循环。这恐怕是 ActivityThread 被称做“Thread”的一个直接证据。另外，mServices 用于保存 Service，Activities 用于保存 ActivityClientRecord，mAllApplications 用于保存 Application。关于这些变量的具体作用，以后遇到时再介绍。

对于 ContextImpl，其成员变量表明它和资源、APK文件有关。

AMS 的 main 函数先分析到此，其创建的 Android 运行环境将在后面的分析中用到。接下来分析 AMS 的第三个调用函数 setSystemProcess。

[1] 实际上，SettingsProvider.apk 也运行于 system_server进程中。

6.2.2 AMS的setSystemProcess分析

AMS的setSystemProcess的代码如下：

[-->ActivityManagerService.java :
setSystemProcess]

```
public static void setSystemProcess () {  
    try{  
        ActivityManagerService m=mSelf ;  
        //向ServiceManager注册几个服务  
        ServiceManager.addService ("activity", m) ;  
        //用于打印内存信息  
        ServiceManager.addService ("meminfo", new MemBinder (m) ) ;  
        /*  
         * Android 4.0新增的服务，用于输出应用进程使用硬件进行显示加速方面的信息  
(Applications  
         Graphics Acceleration Info)。读者可通过adb shell dumpsys  
gfxinfo查看具体的  
输出  
        */  
        ServiceManager.addService ( "gfxinfo" , new GraphicsBinder  
(m) ) ;  
        if (MONITOR_CPU_USAGE) //该值默认为true，添加cpuinfo服务  
        ServiceManager.addService ("cpuinfo", new CpuBinder (m) ) ;  
        //向SM注册权限管理服务PermissionController  
        ServiceManager.addService ( "permission" , new  
PermissionController (m) ) ;  
        /*  
         * 重要说明：  
         向 PackageManagerService 查询 package 名 为 android 的  
ApplicationInfo。
```

注意这句调用，虽然PKMS和AMS同属一个进程，但是二者交互仍然借助Context。

其实，此处完全可以直接调用PKMS的函数。为什么要费如此周折呢？后面具体说明

```
/*
ApplicationInfo info;//使用AMS的mContext对象
mSelf.mContext.getPackageManager () .getApplicationInfo (
"android", STOCK_PM_FLAGS) ;
//①调用ActivityThread的installSystemApplicationInfo函数
mSystemThread.installSystemApplicationInfo (info) ;
synchronized (mSelf) {
//②此处涉及AMS对进程的管理，见下文分析
ProcessRecord app=mSelf.newProcessRecordLocked (
mSystemThread.getApplicationThread () , info,
info.processName) ;//注意，最后一个参数为字符串，值为system
app.persistent=true ;
app.pid=MY_PID ;
app.maxAdj=ProcessList.SYSTEM_ADJ ;
//③保存该ProcessRecord对象
mSelf.mProcessNames.put ( app.processName, app.info.uid,
app) ;
synchronized (mSelf.mPidsSelfLocked) {
mSelf.mPidsSelfLocked.put (app.pid, app) ;
}
//根据系统当前状态，调整进程的调度优先级和OOM_Adj，后面将详细分析该函
数
mSelf.updateLruProcessLocked (app, true, true) ;
}
}.....//抛出异常
}
```

在以上代码中列出了一个重要说明和两个关键点。

重要说明：AMS向PKMS查询名称为android的ApplicationInfo。此处AMS和PKMS的交互是通过Context来完成的，查看这一系列函数调用的代码，最终发现AMS将通过Binder发送请求给PKMS来完成查询功能。AMS和PKMS同属一个进程，它们完全可以不通过Context来交互。此处为何要如此大费周章呢？原因很简单，Android希望system_server中的服务也通过Android运行环境来交互。这更多是从设计上来考虑的，比如组件之间交互接口的统一及未来系统的可扩展性。

关键点一：ActivityThread 的installSystemApplicationInfo函数。

关键点二：ProcessRecord类，它和AMS对进程的管理有关。

通过重要说明，相信读者已能真正理解AMS的main函数中第二个目的的作用了。

现在来看第一个关键点，即ActivityThread的installSystemApplicationInfo函数。

1.ActivityThread的installSystemApplicationInfo函数

installSystemApplicationInfo函数的参数为一个ApplicationInfo对象，该对象由AMS通过Context

查询PKMS中一个名为android的package得来（根据前面介绍的知识，目前只有framework-res.apk声明其package名为android）。

再来看installSystemApplicationInfo的代码，如下所示：

[-->ActivityThread.java :
installSystemApplicationInfo]

```
public void installSystemApplicationInfo ( ApplicationInfo  
info) {  
    synchronized (this) {  
        //返回的ContextImpl对象即之前在AMS的main函数一节中创建的那个对象  
        ContextImpl context=getSystemContext () ;  
        //又调用init初始化该Context，是不是重复调用init呢  
        context.init (new LoadedApk (this, "android", context, info,  
            CompatibilityInfo.DEFAULT_COMPATIBILITY_INFO ) , null,  
        this) ;  
        //创建一个Profiler对象，用于性能统计  
        mProfiler=new Profiler () ;  
    }  
}
```

在以上代码中看到调用context.init的地方，读者可能会有疑惑，getSystemContext函数将返回mSystemContext，而此mSystemContext在AMS的main函数中已经初始化过了，此处为何再次初始化呢？

仔细查看看代码便会发现：

第一次执行init时，在LoadedApk构造函数中第四个表示ApplicationInfo的参数为null。

第二次执行init时，LoadedApk构造函数的第四个参数不为空，即该参数将真正指向一个实际的ApplicationInfo，该ApplicationInfo来源于framework-res.apk。

基于上面的信息，某些读者可能马上能想到：Context第一次执行init的目的仅仅是创建一个Android运行环境，而该Context并没有和实际的ApplicationInfo绑定。而第二次执行init前，先利用Context 和 PKMS 交互 得到一个实际的ApplicationInfo，然后再通过init将此Context和ApplicationInfo绑定。

是否觉得前面的疑惑已豁然而解？且慢，此处又抛出了一个更难的问题：第一次执行init后得到的Context虽然没有绑定ApplicationInfo，不是也能使用吗？此处为何非要和一个ApplicationInfo绑定？答案很简单，因为framework-res.apk（包括后面将介绍的SettingsProvider.apk）运行在system_server中。和其他所有APK一样，它的运行需要一个正确初始化的Android运行环境。

长嘘一口气，这个大难题终于弄明白了！在此基础上，AMS下一步的工作就顺理成章了。

由于framework-res.apk是一个APK文件，和其他APK文件一样，它应该运行在一个进程中。而AMS是专门用于进程管理和调度的，所以运行APK的进程应该在AMS中有对应的管理结构。因此AMS下一步工作就是将这个运行环境和一个进程管理结构对应起来并交由AMS统一管理。

AMS中的进程管理结构是ProcessRecord。

2.ProcessRecord和IAplicationThread

分析ProcessRecord之前，先来思考一个问题：AMS如何与应用进程交互？例如AMS启动一个位于其他进程的Activity，由于该Activity运行在另外一进程中，因此AMS势必要和该进程进行跨进程通信。

答案自然是通过Binder进行通信。为此，Android提供了一个IAplicationThread接口，该接口定义了AMS和应用进程之间的交互函数，图6-5所示为该接口的家族图谱。

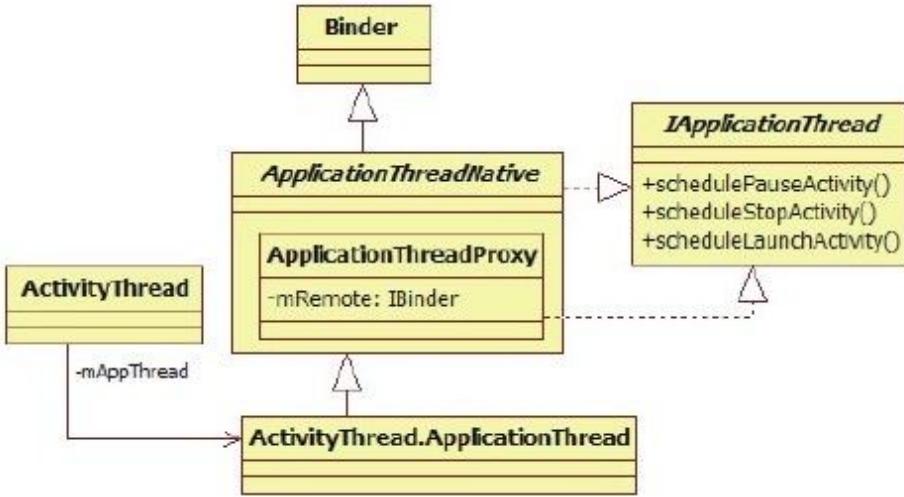


图 6-5 ApplicationThread接口的家庭图谱

由图6-5可知：

ApplicationThreadNative 实现了 IApplicationThread 接口。从该接口定义的函数可知，AMS 通过它可以和应用进程进行交互，例如，AMS 启动一个 Activity 的时候会调用该接口的 scheduleLaunchActivity 函数。

ActivityThread 通过成员变量 mAppThread 指向它的内部类 ApplicationThread，而 ApplicationThread 从 ApplicationThreadNative 派生。

基于以上知识，你能快速得出，IApplicationThread 的 Binder 服务端在应用进程中还是在 AMS 中，这个问题的答案吗？

提示 如果读者知道Binder系统支持客户端监听服务端的死亡消息，那么这个问题的答案就简单了：服务端自然在应用进程中，因为AMS需要监听应用进程的死亡通知。

有了IApplicationThread接口，AMS就可以和应用进程交互了。例如，对于下面一个简单的函数：

[-->ActivityThread.java :
scheduleStopActivity]

```
public final void scheduleStopActivity ( IBinder token,  
boolean showWindow,  
int configChanges) {  
queueOrSendMessage ( //该函数内部将给一个Handler发送对应的消息  
showWindow?H.STOP_ACTIVITY_SHOW : H.STOP_ACTIVITY_HIDE,  
token, 0, configChanges) ;  
}
```

当AMS想要停止（stop）一个Activity时，会调用对应进程IApplicationThread Binder客户端的scheduleStopActivity函数。该函数服务端实现的就是向ActivityThread所在线程发送一个消息。在应用进程中，ActivityThread运行在主线程中，所以这个消息最终在主线程被处理。

提示 Activity的onStop函数也将在主线程中被调用。

IAplicationThread仅仅是AMS和另外一个进程交互的接口，除此之外，AMS还需要更多的有关该进程的信息。在AMS中，进程的信息都保存在 ProcessRecord 数据结构中。那么，ProcessRecord是什么呢？要回答这个问题，需要先来看 setSystemProcess 的第二个关键点，即 newProcessRecordLocked函数，其代码如下：

[-->ActivityManagerService.java :
newProcessRecordLocked]

```
final          ProcessRecord      newProcessRecordLocked
(IAplicationThread thread,
 ApplicationInfo info, String customProcess) {
    String proc=customProcess !=null?customProcess : info.processName ;
    BatteryStatsImpl.Uid.Proc ps=null ;
    BatteryStatsImpl
    stats=mBatteryStatsService.getActiveStatistics () ;
    synchronized (stats) {
        //BSImpl将为该进程创建一个耗电量统计项
        ps=stats.getProcessStatsLocked (info.uid, proc) ;
    }
    //创建一个ProcessRecord对象，代表对应的进程。AMS和该进程的对象是第二个参数thread
    return new ProcessRecord (ps, thread, info, proc) ;
}
```

ProcessRecord的成员变量较多，先来看看在其构造函数中都初始化了哪些成员变量。

[-->ProcessRecord.java : ProcessRecord]

```
ProcessRecord (BatteryStatsImpl.Uid.Proc_batteryStats,
IAplicationThread_thread, ApplicationInfo_info,
String_processName) {
    batteryStats=_batteryStats ;//用于电量统计
    info=_info ;//保存ApplicationInfo
    processName=_processName ;//保存进程名
    //一个进程能运行多个Package, pkgList用于保存package名
    pkgList.add (_info.packageName) ;
    thread=_thread ;//保存IAplicationThread, 通过它可以和应用进程交互
    //下面这些xxxAdj成员变量和进程调度优先级、OOM_adj有关。以后再分析它们
    //的作用
    maxAdj=ProcessList.EMPTY_APP_ADJ ;
    hiddenAdj=ProcessList.HIDDEN_APP_MIN_ADJ ;
    curRawAdj=setRawAdj=-100 ;
    curAdj=setAdj=-100 ;
    //用于控制该进程是否常驻内存（即使被杀掉，系统也会重启它），只有重要的
    //进程才会有此待遇
    persistent=false ;
    removed=false ;
}
```

ProcessRecord 除保存和应用进程通信的 IAplicationThread 对象外，还保存了进程名、不同状态对应的 Oom_adj 值及一个 ApplicationInfo。一个进程虽然可运行多个 Application，但是 ProcessRecord 一般保存该进程中先运行的那个 Application 的 ApplicationInfo。

至此，已经创建了一个ProcessRecord对象，和其他应用进程不同的是，该对象对应的进程为system_server。为了彰显其特殊性，AMS为其中的一些成员变量设置了特定的值：

[-->ActivityManagerService
setSystemProcess]

```
app.persistent=true ; //设置该值为true  
app.pid=MY_PID ; //设置pid为system_server的进程号  
app.maxAdj=ProcessList.SYSTEM_ADJ ; //设置最大OOM_Adj，系统进程  
默认值为-16  
//另外，app的processName被设置成“system”
```

这时，一个针对system_server的ProcessRecord对象就创建完成了。此后AMS将把它并入自己的“势力”范围内。

AMS中有两个成员变量用于保存ProcessRecord，一个是mProcessNames，另一个是mPidsSelfLocked。图6-6为这两个成员变量的数据结构示意图。

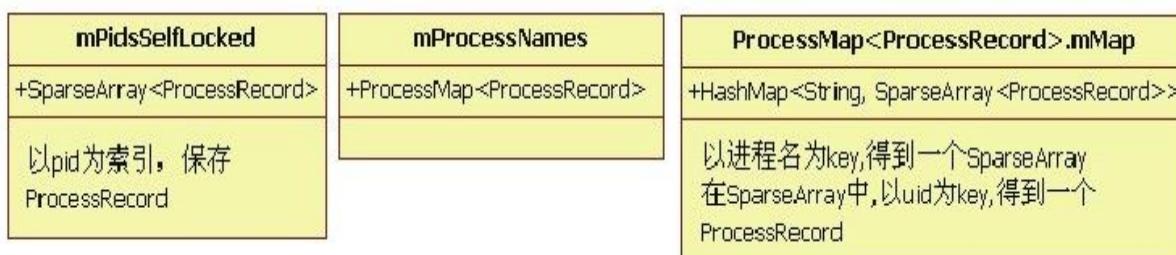


图 6-6 mPidsSelfLocked和mProcessNames数据结构示意图

3.AMS的setSystemProcess总结

现在来总结回顾setSystemProcess的工作：

注册 AMS、meminfo、gfxinfo 等服务到 ServiceManager中。

根据PKMS返回的ApplicationInfo初始化Android运行环境，并创建一个代表system_server进程的ProcessRecord，从此，system_server进程也并入AMS的管理范围内。

6.2.3 AMS的installSystemProviders函数分析

还记得Settings数据库吗？system_server中很多Service都需要向它查询配置信息。为此，Android提供了一个SettingsProvider来帮助开发者。该Provider在SettingsProvider.apk中，installSystemProviders就会加载该APK并把SettingsProvider放到system_server进程中来运行。

此时的system_server进程已经加载了framework-res.apk，现在又要加载另外一个APK文件，这就是多个APK运行在同一进程的典型案例。另外，通过installSystemProviders函数还能见识到ContentProvider的安装过程。下面就来分析installSystemProviders函数，其实现代码如下：

提示 读者在定制自己的Android系统时，千万不可去掉/system/app/SettingsProvider.apk，否则系统将无法正常启动。

[-->ActivityManagerService.java :
installSystemProviders]

```
public static final void installSystemProviders () {
```

```
List<ProviderInfo>providers ;  
synchronized (mSelf) {  
/*  
从 mProcessNames 找到 进程 名 为 “system” 且 uid 为 SYSTEM_UID 的  
ProcessRecord，  
返 回 值 就 是 前 面 在 installSystemApplication 中 创 建 的 那 个  
ProcessRecord，它代表  
SystemServer进 程  
*/  
ProcessRecord app=mSelf.mProcessNames.get ("system" ,  
Process.SYSTEM_UID) ;  
//①关键调用，见下文分析  
providers=mSelf.generateApplicationProvidersLocked (app) ;  
if (providers !=null) {  
.....//将非系统APK（即未设ApplicationInfo.FLAG_SYSTEM标志）提供的  
Provider  
//从providers列表中去掉  
}  
if (providers !=null) { //②为SystemServer进程安装Provider  
mSystemThread.installSystemProviders (providers) ;  
}  
// 监 视 Settings 数 据 库 中 Secure 表 的 变 化 ， 目 前 只 关 注  
long_press_timeout配置的变化  
mSelf.mCoreSettingsObserver=new CoreSettingsObserver  
(mSelf) ;  
//UsageStatsService的工作，以后再讨论  
mSelf.mUsageStatsService.monitorPackages () ;  
}
```

上述代码中列出了两个关键调用，分别是：

调 用 generateApplicationProvidersLocked 函
数，该函数返回一个ProviderInfo List。

调用ActivityThread的installSystemProviders函数。ActivityThread可以看做是进程的Android运行环境，那么installSystemProviders表示为该进程安装ContentProvider。

注意 此处不再区分系统进程还是应用进程。由于只和ActivityThread交互，因此它运行在什么进程中无关紧要。

下面来看第一个关键点generateApplicationProvidersLocked函数。

1.AMS 的 generateApplicationProvidersLocked 函数分析

generateApplicationProvidersLocked 函数的实现代码如下：

[-->ActivityManagerService.java :
generateApplicationProvidersLocked]

```
private final List<ProviderInfo>
generateApplicationProvidersLocked (
    ProcessRecord app) {
    List<ProviderInfo> providers=null;
    try{
        //①向PKMS查询满足要求的ProviderInfo，最重要的查询条件包括：进程名和
        //进程uid
        providers=AppGlobals.getPackageManager () .
        queryContentProviders (app.processName, app.info.uid,
```

```
STOCK_PM_FLAGS|PackageManager.GET_URI_PERMISSION_PATTERNs )  
;  
}.....  
if (providers !=null) {  
final int N=providers.size () ;  
for (int i=0 ; i<N ; i++) {  
//@AMS对ContentProvider的管理，见下文解释  
ProviderInfo cpi= (ProviderInfo) providers.get (i) ;  
ComponentName comp=new ComponentName ( cpi.packageName,  
cpi.name) ;  
ContentProviderRecord cpr=mProvidersByClass.get (comp) ;  
if (cpr==null) {  
cpr=new ContentProviderRecord (cpi, app.info, comp) ;  
//ContentProvider在AMS中用ContentProviderRecord来表示  
mProvidersByClass.put ( comp, cpr ) ;// 保 存 到 AMS 的  
mProvidersByClass中  
}  
//将信息保存到ProcessRecord中  
app.pubProviders.put (cpi.name, cpr) ;  
//保存PackageName到ProcessRecord中  
app.addPackage (cpi.applicationInfo.packageName) ;  
//对该APK进行dex优化  
ensurePackageDexOpt (cpi.applicationInfo.packageName) ;  
}  
}  
return providers ;  
}
```

由以上代码可知：
generateApplicationProvidersLocked先从PKMS那里查询满足条件的ProviderInfo信息，而后将它们分别保存到AMS和ProcessRecord中的对应的数据结构中。

下面先来看查询函数queryContentProviders。

(1) PMS中queryContentProviders函数分析
queryContentProviders函数的实现代码如下：

[-->PackageManagerService.java :
queryContentProviders]

```
public List<ProviderInfo>queryContentProviders ( String
processName,
    int uid, int flags) {
    ArrayList<ProviderInfo>finalList=null;
    synchronized (mPackages) {
        //还记得mProvidersByComponent的作用吗？它以ComponentName为key,
        //保存了
        //PKMS扫描APK得到的PackageParser.Provider信息。读者可参考图4-9
        final Iterator<PackageParser.Provider>i=
            mProvidersByComponent.values () .iterator () ;
        while (i.hasNext ()) {
            final PackageParser.Provider p=i.next () ;
            //下面的if语句将从这些Provider中搜索本例设置的processName为
            //system,
            //uid为SYSTEM_UID, flags为FLAG_SYSTEM的Provider
            if (p.info.authority !=null
                && (processName==null
                    || (p.info.processName.equals (processName)
                        &&p.info.applicationInfo.uid==uid) )
                &&mSettings.isEnabledLPr (p.info, flags)
                && (!mSafeMode|| (p.info.applicationInfo.flags
                    &ApplicationInfo.FLAG_SYSTEM) !=0) ) {
                if (finalList==null) {
                    finalList=new ArrayList<ProviderInfo> (3) ;
                }
                //由PackageParser.Provider得到ProviderInfo，并添加到finalList中
```

```
//关于Provider类及ProviderInfo类，可参考图4-6
finalList.add ( PackageParser.generateProviderInfo ( p,
flags ) ) ;
}
}
}
if (finalList !=null)
// 最终结果按 provider 的 initOrder 排序，该值用于表示初始化
ContentProvider的顺序
Collections.sort (finalList, mProviderInitOrderSorter) ;
return finalList ;//返回最终结果
}
```

queryContentProviders 函数很简单，就是从 PKMS 那里查找满足条件的 Provider，然后生成 AMS 使用的 ProviderInfo 信息。为何偏偏能找到 SettingsProvider 呢？来看它的 AndroidManifest.xml 文件，如图 6-7 所示。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    package="com.android.providers.settings"
    ...
    coreApp="true"
    ...
    android:sharedUserId="android.uid.system">

    ...
    <application android:allowClearUserData="false"
        ...
        android:label="@string/app_label"
        ...
        android:process="system"
        ...
        android:backupAgent="SettingsBackupAgent"
        ...
        android:killAfterRestore="false"
        ...
        android:icon="@drawable/ic_launcher_settings">
        ...
        <!-- todo.add: android:neverEncrypt="true" -->

        ...
        <provider android:name="SettingsProvider" android:authorities="settings"
            ...
            android:multiprocess="false"
            ...
            android:writePermission="android.permission.WRITE_SETTINGS"
            ...
            android:initOrder="100" />
        ...
    </application>
</manifest>
```

图 6-7 SettingsProvider的AndroidManifest.xml文件示意

由图 6-7 可知，SettingsProvider 设置其 uid 为 android.uid.system，同时在 application 中设置了 process 名为 system。而在 framework-res.apk 中也做了相同的设置。所以，现在可以确认 SettingsProvider 将和 framework-res.apk 运行在同一个进程，即 system_server 中。

提示 从运行效率角度来说，这样做也是合情合理的。因为 system_server 进程中的很多 Service 都依赖 Settings 数据库，把它们放在同一个进程中，可以降低由于进程间通信带来的效率损失。

(2) 关于ContentProvider的介绍

前面介绍的从 PKMS 那里查询到的 ProviderInfo 还属于公有财产，现在我们要将它与 AMS 及 ProcessRecord 联系起来。

AMS 保存 ProviderInfo 的原因是它要管理 ContentProvider。

ProcessRecord 保存 ProviderInfo 的原因是 ContentProvider 最终要落实到一个进程中。其实也是为了方便 AMS 管理，例如该进程一旦退出，

AMS需要把其中的ContentProvider信息从系统中去除。

AMS及ProcessRecord均使用了一个新的数据结构ContentProviderRecord来管理ContentProvider信息。图6-8展示了ContentProviderRecord相应的数据结构。

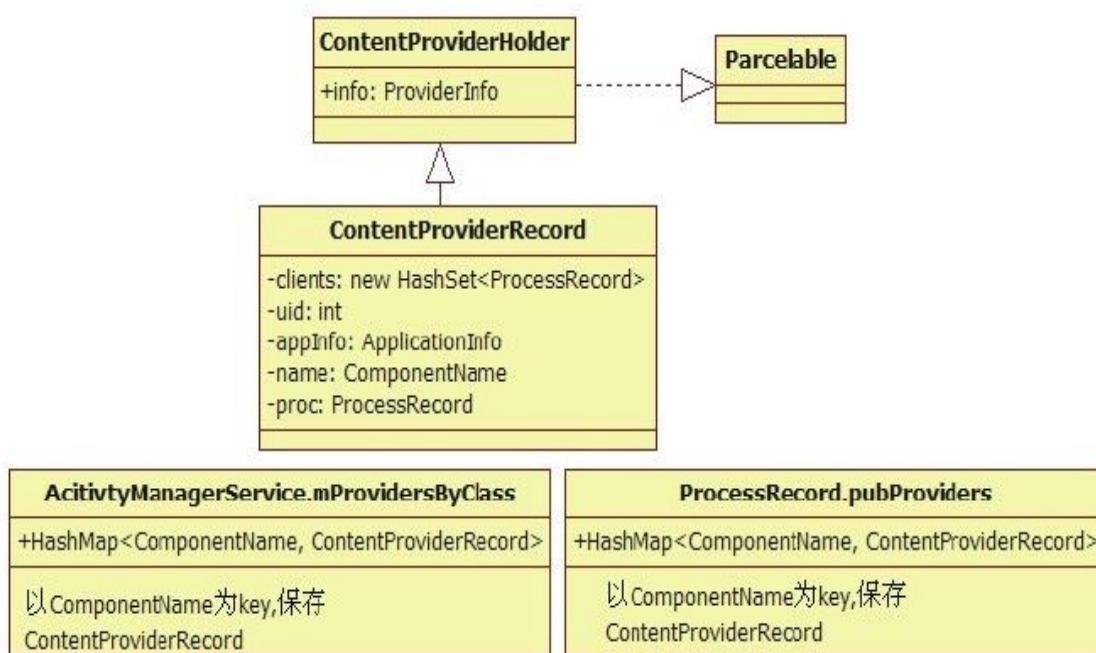


图 6-8 ContentProvicerRecord及相应的“管理团队”

由图6-8可知：

ContentProviderRecord
ContentProviderHolder 派生，内部保存了
ProviderInfo、该 Provider 所驻留的进程

ProcessRecord，以及使用该ContentProvider的客户端进程ProcessRecord（即clients成员变量）。

AMS 的 mProviderByClass 成员变量及 ProcessRecord 的 pubProviders 成员变量均以 ComponentName 为 key 来保存对应的 ContentProviderRecord 对象。

至此，Provider 信息已经保存到 AMS 及 ProcessRecord 中了。那么，下一步的工作是什么呢？

2. ActivityThread 的 installSystemProviders 函数分析

在 AMS 和 ProcessRecord 中都保存了 Provider 信息，但这些都仅是一些信息，并不是 ContentProvider，因此下面要创建一个 ContentProvider 实例（即 SettingsProvider 对象）。该工作由 ActivityThread 的 installSystemProviders 函数来完成，其实现代码如下：

[-->ActivityThread.java :
installSystemProviders]

```
public final void installSystemProviders (List<ProviderInfo> providers) {  
    if (providers != null)
```

```
//调用installContentProviders, 第一个参数真实类型是Application  
installContentProviders (mInitialApplication, providers) ;  
}
```

installContentProviders 这个函数是所有 ContentProvider 产生的必经之路，其实现代码如下：

[-->ActivityThread.java :
installContentProviders]

```
private void installContentProviders (  
    Context context, List<ProviderInfo>providers) {  
    final     ArrayList<IActivityManager.ContentProviderHolder>  
results=  
    new ArrayList<IActivityManager.ContentProviderHolder> () ;  
    Iterator<ProviderInfo>i=providers.iterator () ;  
    while (i.hasNext ()) {  
        ProviderInfo cpi=i.next () ;  
        //①调用installProvider函数, 得到一个IContentProvider对象  
        IContentProvider cp=installProvider ( context, null, cpi,  
false) ;  
        if (cp !=null) {  
            IActivityManager.ContentProviderHolder cph=  
            new IActivityManager.ContentProviderHolder (cpi) ;  
            cph.provider=cp ;  
            //将返回的cp保存到results数组中  
            results.add (cph) ;  
            synchronized (mProviderMap) {  
                //mProviderRefCountMap , 类型 为 HashMap<IBinder,  
                ProviderRefCount>,  
                //主要通过ProviderRefCount对ContentProvider进行引用计数控制, 一旦  
                引用计数
```

```
//降为零，则表示系统中没有地方使用该ContentProvider，要考虑从系统中  
注销它  
    mProviderRefCountMap.put ( cp.asBinder () , new  
ProviderRefCount (10000) );  
}  
}  
}  
}  
try{  
    //②调用AMS的publishContentProviders注册这些ContentProvider，第一个参数  
    //为ApplicationThread  
    ActivityManagerNative.getDefault () .publishContentProviders  
(  
        getApplicationThread () , results) ;  
    }  
....  
}
```

installContentProviders 实际上是标准的 ContentProvider 安装时调用的程序。安装 ContentProvider 包括两方面的工作：

先在 ActivityThread 中通过 installProvider 得到一个 ContentProvider 实例。

向 AMS 发布这个 ContentProvider 实例。如此这般，一个 APK 中声明的 Content-Provider 才能发挥其该有的作用。

提示 上述工作其实和 Binder Service 类似，一个 Binder Service 也需要先创建，然后注册到 ServiceManager 中。

下面来看 ActivityThread 的 installProvider 函数。

(1) ActivityThread的installProvider函数分析

[-->ActivityThread.java : installProvider]

```
private IContentProvider installProvider (Context context,
    IContentProvider provider, ProviderInfo info, boolean
noisy) {
    //注意本例所传的参数：context为mInitialApplication, provider为
    null, info不为null,
    //noisy为false
    ContentProvider localProvider=null;
    if (provider==null) {
        Context c=null;
        ApplicationInfo ai=info.applicationInfo ;
        /*
         * 下面这个if判断的作用就是为该ContentProvider找到对应的Application。
         * 在AndroidManifest.xml中，ContentProvider是Application的子标签，
         * 所以
         */
        ContentProvider 和 Application有一种对应关系。在本例中，传入的
        context (
            其实是 mInitialApplication ) 代表的是 framework-res.apk ， 而
            Provider代表的
            是SettingsProvider。而SettingsProvider.apk所对应的Application
            还未创建，
            所以下面的判断语句最终会进入最后的else分支
        */
        if (context.getPackageName () .equals (ai.packageName) ) {
            c=context;
        }else if (mInitialApplication !=null&&
            mInitialApplication.getPackageName ( ) .equals
            (ai.packageName) ) {
```

```
c=mInitialApplication ;  
}else{  
try{  
//ai.packageName应该是SettingsProvider.apk的Package,  
//名为com.android.providers.settings  
//下面将创建一个context, 指向该APK  
c=context.createPackageContext (ai.packageName,  
Context.CONTEXT_INCLUDE_CODE) ;  
}  
}//if (context.getPackageName () .equals (ai.packageName) ) 判断结束
```

```
if (c==null) return null;
```

```
try{  
/*
```

为什么一定要找到对应的Context呢？除了ContentProvider和Application的

对应关系外，还有一个决定性原因：即只有对应的Context才能加载对应APK的Java字节码，

从而可通过反射机制生成ContentProvider实例

```
*/
```

```
final java.lang.ClassLoader cl=c.getClassLoader () ;
```

```
//通过Java反射机制得到真正的ContentProvider,
```

```
//此处将得到一个SettingsProvider对象
```

```
localProvider= ( ContentProvider ) cl.loadClass  
(info.name) .newInstance () ;
```

```
//从ContentProvider中取出其mTransport成员 (见下文分析)
```

```
provider=localProvider.getIContentProvider () ;
```

```
if (provider==null) return null;
```

```
//初始化该ContentProvider, 内部会调用其onCreate函数
```

```
localProvider.attachInfo (c, info) ;
```

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

```
}//if (provider==null) 判断结束
```

```
synchronized (mProviderMap) {
```

```
/*
```

ContentProvider必须指明一个或多个authority，在第4章曾经提到过，在URL中host : port的组合表示一个authority。这个单词不太好理解，可简单

```
认为它用于指定ContentProvider的位置（类似网站的域名）
*/
String names[] = PATTERN_SEMICOLON.split (info.authority) ;
for (int i=0 ; i<names.length ; i++) {
    ProviderClientRecord pr=new ProviderClientRecord (names[i],
    provider, localProvider) ;
    try{
        //下面这条语句对linkToDeath的调用颇让人费解，见下文分析
        provider.asBinder () .linkToDeath (pr, 0) ;
        mProviderMap.put (names[i], pr) ;
    }.....
} //for循环结束
if (localProvider !=null) {
    //mLocalProviders用于存储由本进程创建的ContentProvider信息
    mLocalProviders.put (provider.asBinder () ,
    new ProviderClientRecord (null, provider, localProvider) ) ;
}
} //synchronized结束
return provider ;
}
```

以上代码不算复杂，但是涉及一些数据结构和一条令人费解的语句，即对linkToDeath函数的调用语句。先来说说那句令人费解的调用。

在本例中，provider变量并非通过函数参数传入，而是在本进程内部创建的。provider在本例中是Bn端（后面分析ContentProvider的getIContentProvider时即可知道），Bn端进程为Bn端设置死亡通知本身就比较奇怪。如果Bn端进程死亡，它设置的死亡通知也无法发送给自己。幸好源代码中有句注释：“Cache the pointer for the

remote provider”。意思是如果provider参数是通过installProvider传递过来的（即该Provider代表远端进程的ContentProvider，此时它应为Bp端），那么这种处理是合适的。不管怎样，这仅仅是为了保存pointer，所以也无关紧要。

至于代码中涉及的数据结构如图6-9所示。

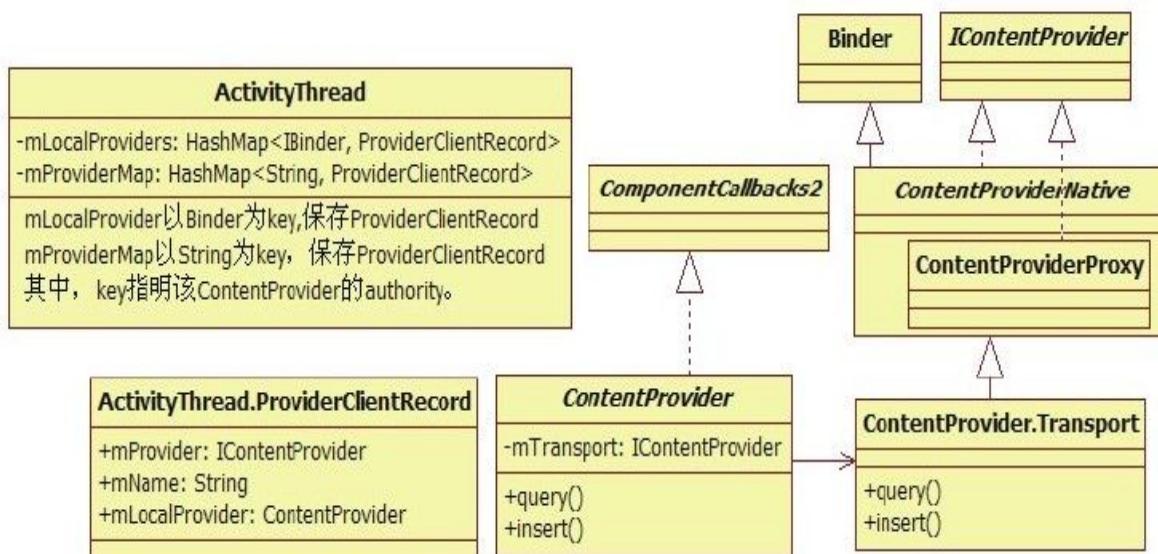


图 6-9 ActivityThread中ContentProvider涉及的数据结构

由图6-9可知：

ContentProvider类本身只是一个容器，而跨进程调用的支持是通过内部类Transport实现的。Transport从ContentProviderNative派生，而ContentProvider的成员变量mTransport指向该Transport对象。ContentProvider的

getIContentProvider函数即返回mTransport成员变量。

ContentProviderNative从Binder派生，并实现了IContentProvider接口。其内部类ContentProviderProxy是供客户端使用的。

ProviderClientRecord是ActivityThread提供的用于保存ContentProvider信息的一个数据结构。它的mLocalProvider用于保存ContentProvider对象，mProvider用于保存IContentProvider对象。另外一个成员mName用于保存该ContentProvider的一个authority。注意，ContentProvider可以定义多个authority，就好像一个网站有多个域名一样。

至此，本例中的SettingProvider已经创建完毕，接下来的工作就是把它推向历史舞台—即发布该Provider了。

(2) AMS的publishContentProviders分析

publishContentProviders函数用于向AMS注册ContentProviders，其实现代码如下：

[-->ActivityManagerService.java :
publishContentProviders]

```
public final void publishContentProviders
(IApplicationThread caller,
 List<ContentProviderHolder>providers) {
.....
synchronized (this) {
//找到调用者所在的ProcessRecord对象
final ProcessRecord r=getRecordForAppLocked (caller) ;
.....
final long origId=Binder.clearCallingIdentity () ;
final int N=providers.size () ;
for (int i=0 ;i<N ;i++) {
ContentProviderHolder src=providers.get (i) ;
.....
//①注意：先从该ProcessRecord中找对应的ContentProviderRecord
ContentProviderRecord dst=r.pubProviders.get
(src.info.name) ;
if (dst !=null) {
ComponentName comp=new ComponentName (dst.info.packageName,
dst.info.name) ;
//以ComponentName为key，保存到mProvidersByClass中
mProvidersByClass.put (comp, dst) ;
String names[]=dst.info.authority.split (" ; ") ;
for (int j=0 ;j<names.length ;j++)
mProvidersByName.put (names[j], dst) ;//以authority为key,
//mLaunchingProviders用于保存处于启动状态的Provider
int NL=mLaunchingProviders.size () ;
int j ;
for (j=0 ;j<NL ;j++) {
if (mLaunchingProviders.get (j) ==dst) {
mLaunchingProviders.remove (j) ;
j-- ;
NL-- ;
}//
}//for (j=0 ;j<NL ;j++) 结束
synchronized (dst) {
dst.provider=src.provider ;
```

```
dst.proc=r ;
dst.notifyAll () ;
}//synchronized结束
updateOomAdjLocked (r) ;//每发布一个Provider，需要调整对应进程的
oom_adj
}//if (dst !=null) 结束
}//for (int i=0 ;j<N ;j++) 结束
Binder.restoreCallingIdentity (origId) ;
}//synchronized (this) 结束
}
```

这里应解释一下publishContentProviders的工作流程：

先根据调用者的PID找到对应的ProcessRecord对象。

该 ProcessRecord 的 pubProviders 中 保 存 了 ContentProviderRecord信息。该信息由前面介绍的 AMS 的 generateApplicationProvidersLocked 函数根据Package本身的信息生成。此处将判断要发布的 ContentProvider是否由该Package声明。

如果判断返回成功，则将该ContentProvider及其对应的authority加到mProviders-ByName中。注意，AMS中还有一个mProvidersByClass变量，该变量以Content-Provider的ComponentName为key，即 系 统 提 供 两 种 方 式 找 到 某 一 个 Content-

Provider，一种是通过authority，另一种方式就是指明ComponentName。

mLaunchingProviders和最后的notifyAll函数用于通知那些等待ContentProvider所在进程启动的客户端进程。例如，进程A要查询一个数据库，需要通过进程B中的某个ContentProvider来实施。如果B还未启动，那么AMS就需要先启动B。在这段时间内，A需要等待B启动并注册对应的ContentProvider。B一旦完成注册，就需要告知A退出等待以继续后续的查询工作。

现在，一个SettingsProvider就算正式在系统中挂牌并注册了，此后，和Settings数据库相关的操作均由它来管理。

3.AMS的installSystemProviders总结

AMS的installSystemProviders函数其实就是用于启动SettingsProvider，其中比较复杂的是ContentProvider相关的数据结构，读者可参考图6-9。

6.2.4 AMS的systemReady分析

作为核心服务，AMS的systemReady会做什么呢？由于该函数内容较多，我们将它的工作分为三个阶段。首先看第一阶段的工作。

1.systemReady第一阶段的工作

该阶段的具体代码如下：

[-->ActivityManagerService.java :
systemReady]

```
public void systemReady (final Runnable goingCallback) {  
    synchronized (this) {  
        .....  
        if (!mDidUpdate) { //判断是否为升级操作  
            if (mWaitingUpdate) return; //升级未完成，直接返回  
            //准备PRE_BOOT_COMPLETED广播  
            Intent intent=new Intent  
                (Intent.ACTION_PRE_BOOT_COMPLETED) ;  
            List<ResolveInfo>ris=null;  
            //向PKMS查询该广播的接收者  
            ris=AppGlobals.getPackageManager () .queryIntentReceivers (  
                intent, null, 0) ;  
            .....//从返回的结果中删除那些非系统APK的广播接收者  
            intent.addFlags (Intent.FLAG_RECEIVER_BOOT_UPGRADE) ;  
            //读取/data/system/called_pre_boots.dat文件，这里存储了上次启动时  
            候已经
```

```
//接收并处理了PRE_BOOT_COMPLETED广播的组件。鉴于该广播的特殊性，系统希望
//该广播仅被这些接收者处理一次
ArrayList<ComponentName>lastDoneReceivers=
readLastDonePreBootReceivers () ;
final ArrayList<ComponentName>doneReceivers=
new ArrayList<ComponentName> () ;
.....//从PKMS返回的接收者中删除那些已经处理过该广播的对象
for (int i=0 ; i<ris.size () ; i++) {
ActivityInfo ai=ris.get (i) .activityInfo ;
ComponentName comp=new ComponentName ( ai.packageName,
ai.name) ;
doneReceivers.add (comp) ;
intent.setComponent (comp) ;
IIntentReceiver finisher=null ;
if (i==ris.size () -1) {
//为最后一个广播接收者注册一个回调通知，当该接收者处理完广播后，将调用
该
//回调
finisher=new IIntentReceiver.Stub () {
public void performReceive (Intent intent, int resultCode,
String data, Bundle extras, boolean ordered,
boolean sticky) {
mHandler.post (new Runnable () {
public void run () {
synchronized (ActivityManagerService.this) {
mDidUpdate=true ;
}
//保存那些处理过该广播的接收者信息
writeLastDonePreBootReceivers (doneReceivers) ;
showBootMessage (mContext.getText (
R.string.android_upgrading_complete) ,
false) ;
systemReady (goingCallback) ;
} //run结束
}) ; //new Runnable结束
```

```
 } //performReceive结束
 } ; //finisher创建结束
 } //if (i==ris.size () -1) 判断结束
 //发送广播给指定的接收者
 broadcastIntentLocked (null, null, intent, null, finisher,
 0 , null, null, null, true, false, MY_PID,
 Process.SYSTEM_UID) ;
 if (finisher !=null) mWaitingUpdate=true ;
 }
 if (mWaitingUpdate) return ;
 mDidUpdate=true ;
 }
 mSystemReady=true ;
 if (!mStartRunning) return ;
 } //synchronized (this) 结束
```

由以上代码可知，systemReady第一阶段的工作并不轻松，其主要职责是发送并处理与PRE_BOOT_COMPLETED广播相关的事情。目前代码中还没有接收该广播的地方，不过从代码中的注释中可猜测到，该广播接收者的工作似乎和系统升级有关。

建议 如有哪位读者了解与此相关的知识，不妨和大家分享。

下面来介绍systemReady第二阶段的工作。

2.systemReady第二阶段的工作

该阶段的具体工作如下：

[-->ActivityManagerService.java systemReady]

:

```
ArrayList<ProcessRecord>procsToKill=null;
synchronized (mPidSelfLocked) {
    for (int i=mPidsSelfLocked.size () -1;i>=0;i--) {
        ProcessRecord proc=mPidsSelfLocked.valueAt (i) ;
        //从mPidSelfLocked中找到那些先于AMS启动的进程，哪些进程有如此能耐，
        //在AMS还未启动完毕就启动完了呢？对，那些声明了persistent为true的进
        程有可能
        if (!isAllowedWhileBooting (proc.info) ) {
            if (procsToKill==null)
                procsToKill=new ArrayList<ProcessRecord> () ;
            procsToKill.add (proc) ;
        }
    }
}
//for结束
//synchronized结束
synchronized (this) {
    if (procsToKill!=null) {
        for (int i=procsToKill.size () -1;i>=0;i--) {
            ProcessRecord proc=procsToKill.get (i) ;
            //把这些进程关闭，removeProcessLocked函数比较复杂，以后再分析
            removeProcessLocked (proc, true, false) ;
        }
    }
}
//至此，系统已经准备完毕
mProcessesReady=true ;
}
synchronized (this) {
    if (mFactoryTest==SystemServer.FACTORY_TEST_LOW_LEVEL) {
        //和工厂测试有关，不对此进行讨论
    }
}
//查询Settings数据，获取一些配置参数
retrieveSettings () ;
```

systemReady第二阶段的工作包括：

杀死那些在AMS还未启动完毕就先启动的应用进程。注意，这些应用进程一定是APK所在的Java进程，因为只有应用进程才会向AMS注册，而一般Native（例如mediaserver）进程是不会向AMS注册的。

从Settings数据库中获取配置信息，目前只取4个配置参数，分别是：debug_app（设置需要debug的app的名称）、wait_for_debugger（如果为1，则等待调试器，否则正常启动debug_app）、always_finish_activities（当一个activity不再有地方使用时，是否立即对它执行destroy）、font_scale（用于控制字体放大倍数，这是Android 4.0新增的功能）。以上配置项由Settings数据库的System表提供。

3.systemReady第三阶段的工作

该阶段的具体工作如下：

[-->ActivityManagerService.java :
systemReady]

```
//调用systemReady传入的参数，它是一个Runnable对象，下节将分析此函数
if (goingCallback != null) goingCallback.run () ;
synchronized (this) {
```

```
if (mFactoryTest !=SystemServer.FACTORY_TEST_LOW_LEVEL) {  
try{  
//从PKMS中查询那些persistent为1的ApplicationInfo  
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) ;  
//由于framework-res.apk已经由系统启动，所以这里需要把它去除  
//framework-res.apk的packageName为android  
if (info !=null& & !info.packageName.equals ("android") )  
addAppLocked (info) ;//启动该Application所在的进程  
}  
}  
}  
}.....  
}  
mBooting=true ;//设置mBooting变量为true，其作用后面会介绍  
try{  
if (AppGlobals.getPackageManager () .hasSystemUidErrors () ) {  
.....//处理那些Uid有错误的Application  
}.....  
//启动全系统第一个Activity，即Home  
mMainStack.resumeTopActivityLocked (null) ;  
}  
}//synchronized结束
```

systemReady第三阶段的工作有3项：

调用systemReady设置的回调对象goingCallback的run函数。

启动那些声明了persistent的APK。

启动桌面。

先看回调对象goingCallback的run函数的工作。

(1) goingCallback的run函数分析

run函数的实现代码如下：

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

```
ActivityManagerService.self () .systemReady (new Runnable ()  
{  
    public void run () {  
        startSystemUi (contextF) ; //启动SystemUi  
        //调用其他服务的systemReady函数  
        if (batteryF !=null) batteryF.systemReady () ;  
        if (networkManagementF !=null) networkManagementF.systemReady  
() ;  
        .....  
        Watchdog.getInstance () .start () ; //启动Watchdog  
        .....//调用其他服务的systemReady函数  
    }  
}
```

run函数比较简单，执行的工作如下：

执行startSystemUi，在该函数内部启动SystemUIService，该Service和状态栏有关。

调用一些服务的systemReady函数。

启动Watchdog。

startSystemUi的实现代码如下：

[-->SystemServer.java : startSystemUi]

```
static final void startSystemUi (Context context) {  
    Intent intent=new Intent () ;  
    intent.setComponent ( new ComponentName  
("com.android.systemui",  
 "com.android.systemui.SystemUIService") ) ;  
    context.startService (intent) ;  
}
```

SystemUIService由SystemUi.apk提供，它实现了系统的状态栏。

注意 在精简 ROM 时，也不能删除 SystemUi.apk。

(2) 启动Home界面

如前所述，resumeTopActivityLocked将启动Home界面，此函数非常重要也比较复杂，故以后再详细分析。我们提取了resumeTopActivityLocked启动Home界面时的相关代码，如下所示：

[-->ActivityStack.java :
resumeTopActivityLocked]

```
final boolean resumeTopActivityLocked (ActivityRecord prev)  
{
```

```
//找到下一个要启动的Activity
ActivityRecord next=topRunningActivityLocked (null) ;
final boolean userLeaving=mUserLeaving;
mUserLeaving=false;
if (next==null) {
//如果下一个要启动的ActivityRecord为空，则启动Home
if (mMainStack) {//全系统就一个ActivityStack，所以mMainStack永远
为true
//mService指向AMS
return mService.startHomeActivityLocked () ;//mService 指向
AMS
}
}
.....//以后再详细分析
}
```

下面来看 AMS 的 startHomeActivityLocked 函数，其实现代码如下：

[-->ActivityManagerService.java :
startHomeActivityLocked]

```
boolean startHomeActivityLocked () {
Intent intent=new Intent (mTopAction,
mTopData !=null?Uri.parse (mTopData) :null) ;
intent.setComponent (mTopComponent) ;
if (mFactoryTest !=SystemServer.FACTORY_TEST_LOW_LEVEL)
intent.addCategory (Intent.CATEGORY_HOME) ;//添加Category为
HOME类别
//向PKMS查询满足条件的ActivityInfo
ActivityInfo aInfo=
intent.resolveActivityInfo (mContext.getPackageManager () ,
STOCK_PM_FLAGS) ;
if (aInfo !=null) {
```

```
intent.setComponent (new ComponentName (
    aInfo.applicationInfo.packageName, aInfo.name) ) ;
ProcessRecord           app=getProcessRecordLocked
(aInfo.processName,
 aInfo.applicationInfo.uid) ;
//在正常情况下，app应该为null，因为刚开机，Home进程肯定还没启动
if (app==null||app.instrumentationClass==null) {
    intent.setFlags (intent.getFlags () |
Intent.FLAG_ACTIVITY_NEW_TASK) ;
//启动Home
mMainStack.startActivityLocked (null, intent, null, null, 0,
aInfo,
null, null, 0, 0, false, false, null) ;
}
}//if (aInfo !=null) 判断结束
return true ;
}
```

至此，AMS中的Service都启动完毕，Home也靓丽登场，整个系统准备完毕，只等待用户的检验了。不过在分析逻辑上还有一点没涉及，那会是什么呢？

(3) 发送ACTION_BOOT_COMPLETED广播

由前面的代码可知，AMS发送了ACTION_PRE_BOOT_COMPLETED广播，可系统中没有地方处理它。在前面的章节中，还碰到一个ACTION_BOOT_COMPLETED广播，该广播广很受欢迎，却不知道它是在哪里发送的。

当 Home Activity 启动后，ActivityStack 的 activityIdleInternal 函数将被调用，其中有一句代码颇值得注意：

[-->ActivityStack.java : activityIdleInternal]

```
final ActivityRecord activityIdleInternal ( IBinder token,
boolean fromTimeout,
Configuration config) {
boolean booting=false;
.....
if (mMainStack) {
booting=mService.mBooting ; //在systemReady的第三阶段工作中设置该
值为true
mService.mBooting=false ;
}
.....
if ( booting ) mService.finishBootning ( ) ;// 调 用 AMS 的
finishBootning函数
}
```

[-->ActivityManagerService.java :
finishBootning]

```
final void finishBootning () {
IntentFilter pkgFilter=new IntentFilter () ;
pkgFilter.addAction (Intent.ACTION_QUERY_PACKAGE_RESTART) ;
pkgFilter.addDataScheme ("package") ;
mContext.registerReceiver (new BroadcastReceiver () {
public void onReceive (Context context, Intent intent) {
.....//处理Package重启的广播
},
pkgFilter) ;
```

```
synchronized (this) {  
    final int NP=mProcessesOnHold.size () ;  
    if (NP>0) {  
        ArrayList<ProcessRecord> procs=  
            new ArrayList<ProcessRecord> (mProcessesOnHold) ;  
        for (int ip=0 ; ip<NP ; ip++) //启动那些等待启动的进程  
            startProcessLocked (procs.get (ip) , "on-hold", null) ;  
    }  
    if (mFactoryTest !=SystemServer.FACTORY_TEST_LOW_LEVEL) {  
        //每15分钟检查一次系统各应用进程使用电量的情况，如果某进程使用  
        WakeLock时间  
        //过长，AMS将关闭该进程  
        Message nmsg=  
            mHandler.obtainMessage (CHECK_EXCESSIVE_WAKE_LOCKS_MSG) ;  
        mHandler.sendMessageDelayed (nmsg, POWER_CHECK_DELAY) ;  
        //设置系统属性sys.boot_completed的值为1  
        SystemProperties.set ("sys.boot_completed", "1") ;  
        //发送ACTION_BOOT_COMPLETED广播  
        broadcastIntentLocked (null, null,  
            new Intent (Intent.ACTION_BOOT_COMPLETED, null) ,  
            null, null, 0, null, null,  
            android.Manifest.permission.RECEIVE_BOOT_COMPLETED,  
            false, false, MY_PID, Process.SYSTEM_UID) ;  
    }  
}
```

原来，在Home启动成功后，AMS才发送ACTION_BOOT_COMPLETED广播。

4.AMS的systemReady总结

systemReady函数完成了系统就绪的必要工作，然后它将启动Home Activity。至此，Android

系统就全部启动了。

6.2.5 初识ActivityManagerService总结

本节所分析的4个关键函数均较复杂，与之相关知识点总结如下：

AMS的main函数：创建AMS实例，其中最重要的工作是创建Android运行环境，得到一个ActivityThread和一个Context对象。

AMS的setSystemProcess函数：该函数注册AMS和meminfo等服务到ServiceManager中。另外，它为system_server创建了一个ProcessRecord对象。由于AMS是Java世界的进程管理及调度中心，要做到对Java进程一视同仁，尽管system_server贵为系统进

程，此时也不得不将其并入AMS的管理范围内。

AMS的installSystemProviders函数：为system_server加载SettingsProvider。

AMS的systemReady函数：做系统启动完毕前最后一些扫尾工作。该函数调用完毕后，Home Activity将呈现在用户面前。

对AMS调用轨迹分析是我们破解AMS的第一条线，希望读者反复阅读，以真正理解其中涉及的知识点，尤其是和Android运行环境及Context相关知识。

6.3 startActivity分析

本节将重点分析Activity的启动过程，它是五条线中最难分析的一条，只要用心，相信读者能啃动这块“硬骨头”。

6.3.1 从am说起

am和pm（见4.4.2节）一样，也是一个脚本，它用来和AMS交互，如启动Activity、启动Service、发送广播等。其核心文件在Am.java中，代码如下：

[-->Am.java : main]

```
public static void main (String[]args) {  
try{  
    (new Am () ) .run (args) ;//构造一个am对象，并调用run函数  
}.....  
}
```

am的用法很多，读者可通过adb shell登录手机，然后执行am，这样就能获取它的用法。

利用am启动一个activity的方法如下：

```
am start[-D][-W][-P<FILE>][--start-profiler<FILE>][-S]<INTENT>
```

其中：

方括号中的参数为可选项，尖括号中的参数为必选项。

<INTENT>参数有很多，主要是用于设置Intent的各项参数。

假设已知某个Activity的ComponentName（package名和Activity的Class名），启动这个Activity的相应命令如下：

```
am start-W-n com.dfp.test/.TestActivity
```

其中，-W选项表示am将会等目标Activity启动后才返回，-n表示后面的参数用于设置Intent的Component。就本例而言，com.dfp.test为Package名，.TestActivity为该Package下对应的Activity类名，所以将要启动的Activity的全路径名为com.dfp.test.TestActivity。

现在就以上面的命令为例来分析am的run函数，代码如下：

[-->Am.java : run]

```
private void run (String[]args) throws Exception{  
    mAm=ActivityManagerNative.getDefault () ;  
    mArgs=args ;  
    String op=args[0] ;mNextArg=1 ;  
    if (op.equals ("start") ) runStart () ;//用于启动Activity  
    else if.....//处理其他参数  
}
```

runStart函数用于处理Activity启动请求，其代码如下：

[-->Am.java : runStart]

```
private void runStart () throws Exception{  
    Intent intent=makeIntent () ;  
    String mimeType=intent.getType () ;  
    //获取mimeType,  
    if (mimeType==null&&intent.getData () !=null  
        && "content".equals (intent.getData () .getScheme () ) ) {  
        mimeType=mAm.getProviderMimeType (intent.getData () ) ;  
    }  
    if (mStopOption) {  
        ....//处理-S选项，即先停止对应的Activity，再启动它  
    }  
    //FLAG_ACTIVITY_NEW_TASK这个标签很重要  
    intent.addFlags (Intent.FLAG_ACTIVITY_NEW_TASK) ;  
    ParcelFileDescriptor fd=null ;  
    if (mProfileFile !=null) {  
        try{.....//处理-P选项，用于性能统计  
            fd=ParcelFileDescriptor.open (.....)  
        }.....  
    }  
    IActivityManager.WaitResult result=null ;  
    int res ;
```

```
if (mWaitOption) { //mWaitOption控制是否等待启动结果，如果有 -W 选  
项，则该值为true  
    //调用AMS的startActivityAndWait函数  
    result=mAm.startActivityAndWait (null, intent, mimeType,  
        null, 0, null, null, 0, false, mDebugOption,  
        mProfileFile, fd, mProfileAutoStop) ;  
    res=result.result ;  
}  
.....  
.....//打印结果  
}
```

am最终将调用AMS的startActivityAndWait函数来处理这次启动请求。下面将深入到AMS内部去继续这次旅程。

提示 为什么选择从am来分析Activity的启动呢？这是因为如果选择从一个Activity来分析如何启动另一个Activity，则将给人一种鸡生蛋、蛋孵鸡的感觉，故此处选择从am入手。除此之外，从am来分析Activity的启动也是Activity启动分析中相对简单的一条路线。

6.3.2 AMS的startActivityAndWait函数分析

startActivityAndWait函数有很多参数，下面先来认识一下它们。

[-->ActivityManagerService.java :
startActivityAndWait原型]

```
public final WaitResult startActivityAndWait (
    /*
        在绝大多数情况下，一个Activity的启动是由一个应用进程发起的，  

        IApplicationThread是  

        应用进程和AMS交互的通道，也可算是调用进程的标识，在本例中，am并非一个应  

        用进程，所以  

        传递的caller为null
    */
    IApplicationThread caller,
    //Intent及resolvedType，在本例中，resolvedType为null
    Intent intent, String resolvedType,
    //grantedUriPermissions和grantedMode两个参数和授权有关，读者可参考  

    第3章对Clipboard
    Service分析时介绍的授权知识
    Uri[] grantedUriPermissions, //在本例中为null
    int grantedMode, //在本例中为0
    IBinder resultTo, // 在本例中为 null，用于接收  

    startActivityForResult的结果
    String resultWho, //在本例中为null
    //requestCode在本例中为0，该值的具体意义由调用者解释。如果该值大于等  

    于0，则AMS内部保存该值，
```

```
//并通过onActivityResult返回给调用者
int requestCode,
boolean onlyIfNeeded, //本例为false
boolean debug, //是否调试目标进程
//下面3个参数和性能统计有关
String profileFile,
ParcelFileDescriptor profileFd, boolean autoStopProfiler)
```

关于以上代码中一些参数的具体作用，以后碰到时会再作分析。建议读者先阅读SDK文档中关于Activity类定义的几个函数，如startActivity、startActivityForResult及onActivity-Result等。

startActivityAndWait的代码如下：

[-->ActivityManagerService.java :
startActivityAndWait]

```
public final WaitResult startActivityAndWait
(IApplicationThread caller,
Intent intent, String resolvedType,
Uri[] grantedUriPermissions,
int grantedMode, IBinder resultTo, String resultWho, int
requestCode,
boolean onlyIfNeeded, boolean debug, String profileFile,
ParcelFileDescriptor profileFd, boolean autoStopProfiler) {
//创建WaitResult对象用于存储处理结果
WaitResult res=new WaitResult () ;
//mMainStack为ActivityStack类型，调用它的startActivityMayWait函
数
mMainStack.startActivityMayWait ( caller , -1 , intent,
resolvedType,
grantedUriPermissions, grantedMode, resultTo, resultWho,
```

```
requestCode, onlyIfNeeded, debug, profileFile, profileFd,  
autoStopProfiler, res, null) ;//最后一个参数为Configuration,  
//在本例中为null  
return res;  
}
```

mMainStack 为 AMS 的成员变量，类型为 ActivityStack，该类是Activity调度的核心角色。正式分析它之前，有必要先介绍一下相关的基础知识。

1.Task、 Back Stack、 ActivityStack 及 Launch mode

(1) 关于Task及Back Stack的介绍

先来看图 6-10 所示，其中列出了用户在 Android 系统上想干的三件事情，分别用 A、B、C 表示，将每一件事情称为一个 Task。一个 Task 还可细分为多个子步骤，即 Activity。

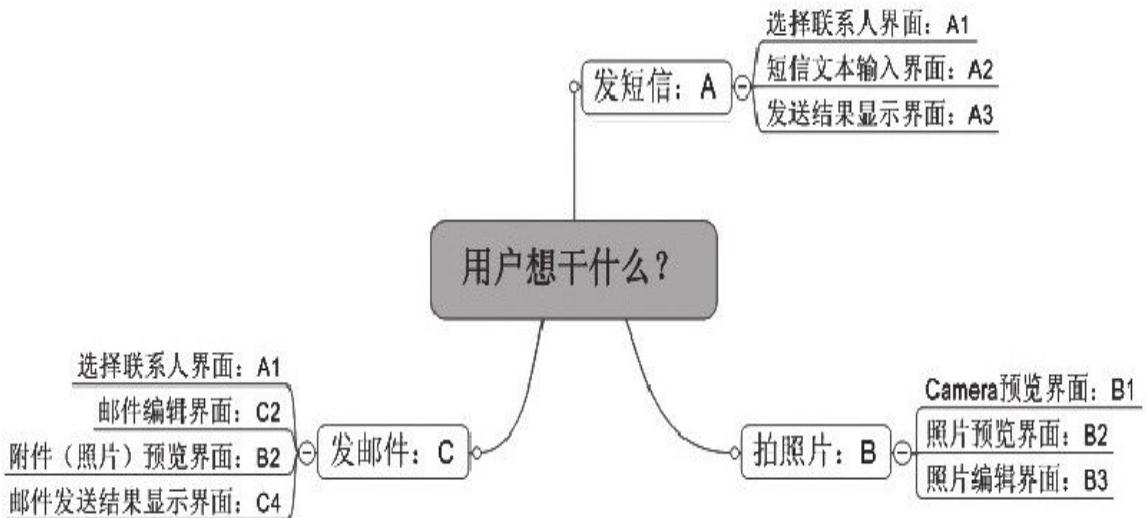


图 6-10 用户想干的事

提示 为什么叫 Activity ? 读者可参考 Merrian-Webster 词典对 Activity 的解释 [1] : “an organizational unit for performing a specific function”, 也就是说, 它是一个有组织的单元, 用于完成某项指定功能。

由图6-10可知, A、B两个Task使用了不同的Activity来完成相应的任务。注意, A、B这两个Task的Activity之间没有复用。

再来看C这个Task, 它可细分为4个Activity, 其中有两个Activity分别使用了A Task的A1、B Task的B2。C Task为什么不新建自己的Activity, 而用其他Task的呢? 这是因为用户想做的事情(即Task)即使完全不同, 但是当细分Task为Activity时, 也可能出现Activity功能类似的情况。

既然A1、B2已能满足要求，自然也就不用重复创建Activity了。另外，通过重用Activity，也可为用户提供一致的界面和体验。

了解了Android设计理念后，我们来看看Android是如何组织Task及它所包含的Activity的。此处有一个简单的例子，如图6-11所示。

由图6-11可知：

本例中的Task包含4个Activity。用户可单击按钮跳转到下一个Activity。同时，通过返回键可回到上一个Activity。

虚线下方是这些Activity的组织方式。Android采用了Stack的方法管理这3个Activity。例如在Activity 1启动Activity 2后，Activity 2入栈成为栈顶成员，Activity 1成为栈底成员，而界面显示的是栈顶成员的内容。当按返回键时，Activity 3出栈，这时候Activity 2成为栈顶，界面显示也相应变成了Activity 2。

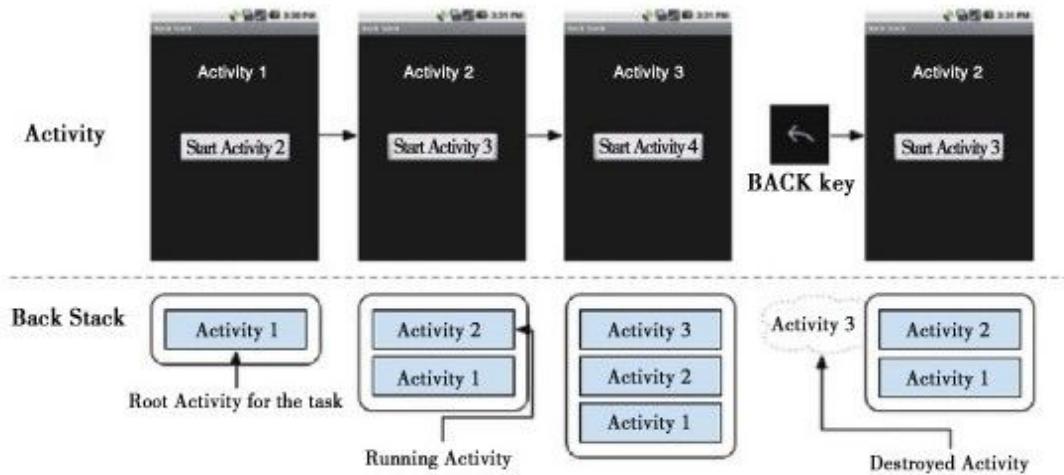


图 6-11 Task及Back Stack示例

以上是一个Task的情况。那么，多个Task又会是何种情况呢？如图6-12所示。

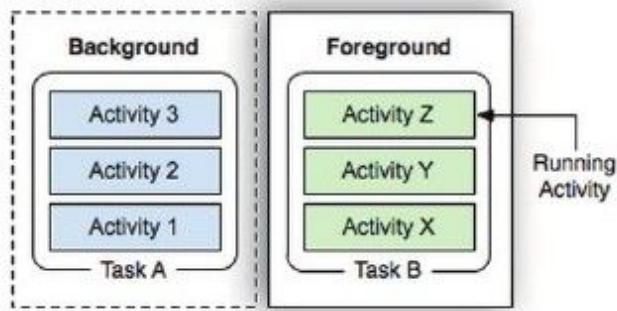


图 6-12 多个Task的情况

由图6-12可知：对多Task的情况来说，系统只支持一个处于前台的Task，即用户当前看到的Activity所属的Task的，其余的Task均处于后台，这些后台Task内部的Activity保持顺序不变。用户可以一次将整个Task挪到后台或者置为前台。

提示 用过Android手机的读者应该知道，长按Home键，系统会弹出近期Task列表，使用户能快速在多个Task间切换。

以上内容从抽象角度介绍了什么是Task，以及Android如何分解Task和管理Activity，那么在实际代码中，是如何考虑并设计的呢？

(2) 关于ActivityStack的介绍

通过上述分析，我们对Android的设计有了一定了解，那么如何用代码来实现这一设计呢？此处有两点需要考虑：

Task内部Activity的组织方式。由图6-11可知，Android通过先入后出的方式来组织Activity。数据结构中的Stack即以这种方式工作。

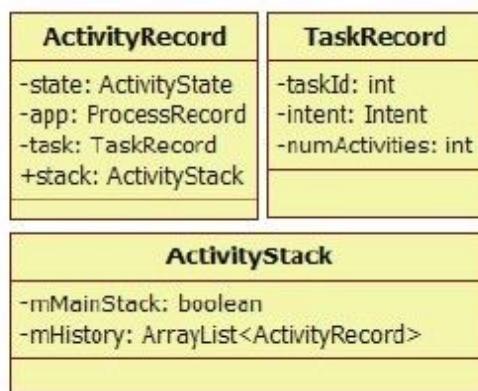


图 6-13 ActivityStack及相关成员
多个Task的组织及管理方式。

Android设计了一个ActivityStack类来负责上述工作，它的组成如图6-13所示。

由图6-13可知：

Activity由ActivityRecord表示，Task由TaskRecord表示。ActivityRecord的task成员指向该Activity所在的Task。state变量用于表示该Activity所处的状态（包括INITIALIZING、RESUMED、PAUSED等状态）。

ActivityStack用mHistory这个ArrayList保存ActivityRecord。令人大跌眼镜的是，该mHistory保存了系统中所有Task的ActivityRecord，而不是针对某个Task进行保存。

ActivityStack的mMainStack成员比较有意思，它代表此ActivityStack是否为主ActivityStack。有主必然有从，但是目前系统中只有一个ActivityStack，并且它的mMainStack为true。从ActivityStack的命名可推测，Android在开发之初也想用ActivityStack来管理单个Task中的ActivityRecord（在ActivityStack.java的注释中说过，该类为“State and management of a single stack of activities”），但不知何故，现在的代码实现中将所有Task的ActivityRecord都放到mHistory中了，并且依然保留mMainStack。

ActivityStack 中 没 有 成 员 用 于 保 存 TaskRecord。

由上述内容可知，ActivityStack采用数组的方式保存所有Task的ActivityRecord，并且没有成员保存TaskRecord。这种实现方式有优点亦有缺点：

优点是少了TaskRecord一级的管理，直接以ActivityRecord为管理单元。这种做法能降低管理方面的开销。

缺点是弱化了Task的概念，结构不够清晰。

下面来看ActivityStack中几个常用的搜索ActivityRecord的函数，代码如下：

[-->ActivityStack.java
topRunningActivityLocked]

/*topRunningActivityLocked：
找到栈中第一个与 notTop 不同的，并且不处于finishing状态的
ActivityRecord。当notTop为

null时，该函数即返回栈中第一个需要显示的ActivityRecord。提醒读者，栈的出入口只能是栈顶。

虽然mHistory是一个数组，但是查找均从数组末端开始，所以其行为也大体符合
Stack的定义

```
*/  
final ActivityRecord topRunningActivityLocked  
(ActivityRecord notTop) {  
    int i=mHistory.size () -1;  
    while (i>=0) {
```

```
ActivityRecord r=mHistory.get (i) ;
if (!r.finishing&&r !=notTop) return r ;
i-- ;
}
return null ;
}
```

类似的函数还有：

[-->ActivityStack.java :
topRunningNonDelayedActivityLocked]

```
/*topRunningNonDelayedActivityLocked
与topRunningActivityLocked类似，但ActivityRecord要求增加一项，即
delayeResume为
false
*/
final ActivityRecord topRunningNonDelayedActivityLocked
(ActivityRecord notTop) {
    int i=mHistory.size () -1 ;
    while (i>=0) {
        ActivityRecord r=mHistory.get (i) ;
        //delayedResume变量控制是否暂缓resume Activity
        if (!r.finishing & & !r.delayedResume & & r !=notTop) return
r ;
        i-- ;
    }
    return null ;
}
```

ActivityStack还提供findActivityLocked函数以
根 据 Intent 及 ActivityInfo 来 查 找 匹 配 的

ActivityRecord，同样，查找也是从mHistory尾端开始，相关代码如下：

[-->ActivityStack.java : findActivityLocked]

```
private ActivityRecord findActivityLocked ( Intent intent,
ActivityInfo info) {
    ComponentName cls=intent.getComponent () ;
    if (info.targetActivity !=null)
        cls=new ComponentName ( info.packageName,
info.targetActivity) ;
    final int N=mHistory.size () ;
    for (int i= (N-1) ;i>=0 ;i--) {
        ActivityRecord r=mHistory.get (i) ;
        if (!r.finishing)
            if (r.intent.getComponent () .equals (cls) ) return r ;
    }
    return null ;
}
```

另一个findTaskLocked函数的返回值是ActivityRecord，其代码如下：

[ActivityStack.java : findTaskLocked]

```
private ActivityRecord findTaskLocked ( Intent intent,
ActivityInfo info) {
    ComponentName cls=intent.getComponent () ;
    if (info.targetActivity !=null)
        cls=new ComponentName ( info.packageName,
info.targetActivity) ;
    TaskRecord cp=null ;
    final int N=mHistory.size () ;
    for (int i= (N-1) ;i>=0 ;i--) {
```

```
ActivityRecord r=mHistory.get (i) ;
//r.task !=cp, 表示不搜索属于同一个Task的ActivityRecord
if (!r.finishing&&r.task !=cp
&&r.launchMode !=ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
cp=r.task ;
//如果Task的affinity相同，则返回这条ActivityRecord
if (r.task.affinity !=null) {
if (r.task.affinity.equals (info.taskAffinity) ) return r ;
}else if (r.task.intent !=null
&&r.task.intent.getComponent () .equals (cls) ) {
//如果Task Intent的ComponentName相同
return r ;
}else if (r.task.affinityIntent !=null
&&r.task.affinityIntent.getComponent () .equals (cls) ) {
return r ;
}//if (r.task.affinity !=null) 判断结束
}//if (!r.finishing&&r.task !=cp.....) 判断结束
}//for循环结束
return null ;
}
```

其实，`findTaskLocked`是根据`mHistory`中`ActivityRecord`所属的Task的情况来进行相应的查找工作。

以上这4个函数均是`ActivityStack`中常用的函数，如果不需逐项（case by case）地研究AMS，那么读者仅需了解这几个函数的作用即可。

(3) 关于Launch Mode的介绍

Launch Mode用于描述Activity的启动模式，目前一共有4种模式，分别是standard、singleTop、singleTask和singleInstance。初看它们，较难理解，实际上不过是Android玩的一个“小把戏”而已。启动模式就是用于控制Activity和Task关系的。

standard：一个Task中可以有多个相同类型的Activity。注意，此处是相同类型的Activity，而不是同一个Activity对象。例如在Task中有A、B、C、D等4个Activity，如果再启动A类Activity，Task就会变成A、B、C、D、A。最后一个A和第一个A是同一类型，却并非同一对象。另外，多个Task中也可以有同类型的Activity。

singleTop：当某Task中有A、B、C、D等4个Activity时，如果D想再启动一个D类型的Activity，那么Task将是什么样子呢？在singleTop模式下，Task中仍然是A、B、C、D，只不过D的onNewIntent函数将被调用，而在standard模式下，Task将变成A、B、C、D、D，最后的D为新创建的D类型Activity对象。在singleTop模式下，只有目标Activity当前正好在栈顶时才有效，例如只有启动处于栈顶的D时才有用，如果启动不处于栈顶的A、B、C等，则无效。

`singleTask`：在这种启动模式下，该Activity只存在一个实例，并且将和一个Task绑定。当需要此Activity时，系统会以`onNewIntent`方式启动它，而不会新建Task和Activity。注意，该Activity虽只有一个实例，但是在Task中除了它之外，还可以有其他的Activity。

`singleInstance`：它是`singleTask`的加强版，即一个Task只能有这么一个设置了`singleInstance`的Activity，不能再有别的Activity。而在`singleTask`模式中，Task还可以有其他的Activity。

注意，Android建议一般的应用开发者不要轻易使用最后两种启动模式。因为这些模式虽然名义上为Launch Mode，但是它们也会影响Activity出栈的顺序，导致用户按返回键返回时不一致的用户体验。

除了启动模式外，Android还有其他一些标志用于控制Activity及Task之间的关系。这里只列举其中一部分，详细信息请参阅SDK文档中Intent的相关说明。

`FLAG_ACTIVITY_NEW_TASK`：将目标Activity放到一个新的Task中。

FLAG_ACTIVITY_CLEAR_TASK：当启动一个Activity时，先把和目标Activity有关联的Task“干掉”，然后启动一个新的Task，并把目标Activity放到新的Task中。该标志必须和FLAG_ACTIVITY_NEW_TASK标志一起使用。

FLAG_ACTIVITY_CLEAR_TOP：当启动一个不处于栈顶的Activity时，先把排在它前面的Activity“干掉”。例如Task有A、B、C、D等4个Activity，要启动B，应直接把C、D“干掉”，而不是新建一个B。

提示 这些启动模式和标志，在笔者看来很像洗扑克牌时的手法，因此可以称之为小把戏。虽是小把戏，但是相关代码的逻辑及分支却异常繁杂，我们应从更高的角度来看待它们。

介绍完上面的知识后，下面来分析ActivityStack的startActivityMayWait函数。

2.ActivityStack 的 startActivityMayWait 函数分析

startActivityMayWait 函数的目标是启动com.dfp.test.TestActivity，假设系统之前没有启动过该Activity，本例最终的结果将是：

由于在 am 中设置了 FLAG_ACTIVITY_NEW_TASK 标志，因此除了会创建一个新的 ActivityRecord 外，还会新创建一个 TaskRecord。

还需要启动一个新的应用进程以加载并运行 com.dfp.test.TestActivity 的一个实例。

如果 TestActivity 不是 Home，还需要停止当前正在显示的 Activity。

我们可将这个函数分三部分进行介绍，下面先来分析第一部分。

(1) startActivityMayWait 分析之一

这部分的代码具体如下：

[-->ActivityStack.java : startActivityMayWait]

```
final int startActivityMayWait ( IApplicationThread caller,
int callingUid,
Intent intent, String resolvedType,
Uri[] grantedUriPermissions,
int grantedMode, IBinder resultTo,
String resultWho, int requestCode, boolean onlyIfNeeded,
boolean debug, String profileFile, ParcelFileDescriptor
profileFd,
boolean autoStopProfiler, WaitResult outResult,
Configuration config) {
....
```

```
//在本例中，已经指明了Component，这样可省去为Intent匹配搜索之苦
boolean componentSpecified=intent.getComponent () !=null;
//创建一个新的Intent，防止客户传入的Intent被修改
intent=new Intent (intent) ;
//查询满足条件的ActivityInfo，在resolveActivity内部和PKMS交互，读者不妨自己
//尝试分析该函数
ActivityInfo aInfo=resolveActivity ( intent, resolvedType,
debug,
profileFile, profileFd, autoStopProfiler) ;
synchronized (mService) {
int callingPid;
if (callingUid>=0) {
callingPid=-1;
}else if (caller==null) {//本例中，caller为null
callingPid=Binder.getCallingPid () ;//取出调用进程的Pid
//取出调用进程的Uid。在本例中，调用进程是am，它由shell启动
callingUid=Binder.getCallingUid () ;
}else{
callingPid=callingUid=-1;
}//if (callingUid>=0) 判断结束
//在本例中config为null
mConfigWillChange=config !=null
&&mService.mConfiguration.diff (config) !=0;
final long origId=Binder.clearCallingIdentity () ;
if ( mMainStack & & aInfo !=null & &
(aInfo.applicationInfo.flags&
ApplicationInfo.FLAG_CANT_SAVE_STATE) !=0) {
/*
AndroidManifest.xml中的Application标签可以声明一个cantSaveState
属性，设置了该属性的Application将不享受系统提供的状态保存/恢复功能。
当一个Application退到后台时，系统会为它保存状态，当调度其到前台运行
时，
将恢复它之前的状态，以保证用户体验的连续性。声明了该属性的Application
被称为
```

“heavy weight process”。可惜系统目前不支持该属性，因为
PackageParser

```
将不解析该属性。详情请见PackageParser.java parseApplication函数
*/
}

.....//待续
```

startActivityMayWait第一阶段的工作内容相对比较简单，主要包括以下几方面：

首先需要通过PKMS查找匹配该Intent的ActivityInfo。

处理FLAG_CANT_SAVE_STATE的情况，但系统目前不支持此情况。

另外，获取调用者的pid和uid。由于本例的caller为null，故所得到的pid和uid均为am所在进程的uid和pid。

下面介绍startActivityMayWait第二阶段的工作。

(2) startActivityMayWait分析之二

这部分的代码具体如下：

[-->ActivityStack.java : startActivityMayWait]

```
//调用此函数启动Activity，将返回值保存到res
int res=startActivityLocked (caller, intent, resolvedType,
```

```
grantedUriPermissions, grantedMode, aInfo,
resultTo, resultWho, requestCode, callingPid, callingUid,
onlyIfNeeded, componentSpecified, null) ;
// 如 果 configuration 发 生 变 化 , 则 调 用 AMS 的
updateConfigurationLocked
//进行处理。关于这部分内容，读者学完本章后可自行分析
if (mConfigWillChange & & mMainStack) {
    mService.enforceCallingPermission (
        android.Manifest.permission.CHANGE_CONFIGURATION,
        "updateConfiguration () " );
    mConfigWillChange=false ;
    mService.updateConfigurationLocked (config, null, false) ;
}
```

此 处 , 启 动 Activity 的 核 心 函 数 是 startActivityLocked , 该 函 数 异 常 复 杂 , 后 面 将 用 一 节 专 门 分 析 。 下 面 先 继 续 分 析 startActivityMayWait 第 三 阶 段 的 工 作。

(3) startActivityMayWait 分 析 之 三

这 部 分 的 代 码 具 体 如 下 :

[-->ActivityStack.java : startActivityMayWait]

```
if (outResult !=null) {
    outResult.result=res ; //设置启动结果
    if (res==IActivityManager.START_SUCCESS) {
        //将该结果加到mWaitingActivityLaunched中保存
        mWaitingActivityLaunched.add (outResult) ;
        do{
            try{
                mService.wait () ; //等待启动结果
            }
```

```
    }
    }while (!outResult.timeout && outResult.who==null) ;
    }else if (res==IActivityManager.START_TASK_TO_FRONT) {
        ....//处理START_TASK_TO_FRONT结果，读者可自行分析
    }
    }//if (outResult !=null) 结束
    return res;
}
}
```

第三阶段的工作就是根据返回值做一些处理，那么 res 返回成功（即 res==IActivityManager.START_SUCCESS 的时候）后为何还需要等待呢？

这是因为目标Activity要运行在一个新的应用进程中，就必须等待那个应用进程正常启动并处理相关请求。注意，只有am设置了-W选项，才会进入wait状态。

[1]<http://www.merriam-webster.com/dictionary/activity>中第六条解释。

6.3.3 startActivityLocked分析

startActivityLocked 是 startActivityMayWait 第二阶段的工作重点，该函数有点长，请读者耐心看代码。

[-->ActivityStack.java : startActivityLocked]

```
final int startActivityLocked (IApplicationThread caller,
Intent intent, String resolvedType,
Uri[] grantedUriPermissions,
int grantedMode, ActivityInfo aInfo, IBinder resultTo,
String resultWho, int requestCode,
int callingPid, int callingUid, boolean onlyIfNeeded,
boolean componentSpecified, ActivityRecord[] outActivity) {
int err=START_SUCCESS ;
ProcessRecord callerApp=null ;
//如果caller不为空，则需要从AMS中找到它的ProcessRecord。本例的
caller为null
if (caller !=null) {
callerApp=mService.getRecordForAppLocked (caller) ;
//其实就是想得到调用进程的pid和uid
if (callerApp !=null) {
callingPid=callerApp.pid ;//一定要保证调用进程的pid和uid正确
callingUid=callerApp.info.uid ;
}else{//如调用进程没有在AMS中注册，则认为其是非法的
err=START_PERMISSION_DENIED ;
}
}//if (caller !=null) 判断结束
//下面两个变量意义很重要。sourceRecord用于描述启动目标Activity的那
个Activity,
```

```
//resultRecord用于描述接收启动结果的Activity，即该Activity的
onActivityResult
    //将被调用以通知启动结果，读者可先阅读SDK中startActivityForResult
    函数的说明
    ActivityRecord sourceRecord=null;
    ActivityRecord resultRecord=null;
    if (resultTo !=null) {
        //本例resultTo为null,
    }
    //获取Intent设置的启动标志，它们是和Launch Mode类似的“小把戏”，
    //所以，读者务必理解本书介绍的关于Launch Mode的知识点
    int launchFlags=intent.getFlags () ;
    if ( (launchFlags& Intent.FLAG_ACTIVITY_FORWARD_RESULT) !=0
        &&sourceRecord !=null) {
        .....
    /*

```

前面介绍的Launch Mode和Activity的启动有关，实际上还有一部分标志用于控制

Activity启动结果的通知。有关FLAG_ACTIVITY_FORWARD_RESULT的作用，读者可

参考SDK中的说明。使用这个标签有个前提，即Activity必须先存在，正如上边代码的if中sourceRecord

不为 null 的判断所示。另外，读者自己可尝试编写例子，以测试 FLAG_ACTIVITY_FORWARD_RESULT

标志的作用

```
    */
    //检查err值及Intent的情况
    if (err==START_SUCCESS&&intent.getComponent () ==null)
        err=START_INTENT_NOT_RESOLVED ;
    .....
    //如果err不为0，则调用sendActivityResultLocked返回错误
    if (err !=START_SUCCESS) {
        if (resultRecord !=null) {//resultRecord接收启动结果
            sendActivityResultLocked ( -1 ,  resultRecord,  resultWho,
requestCode,
            Activity.RESULT_CANCELED,  null) ;

```

```

}

.....
return err ;
}

//检查权限
final      int      perm=mService.checkComponentPermission
(aInfo.permission,
    callingPid,      callingUid,      aInfo.applicationInfo.uid,
aInfo.exported) ;
.....//权限检查失败的处理，不必理会
if (mMainStack) {
    //可为AMS设置一个IActivityController类型的监听者，AMS有任何动静都
会回调该
    //监听者。不过谁又有如此本领去监听AMS呢？在进行Monkey测试的时候，
Monkey会
    //设置该回调对象。这样，Monkey就能根据AMS放映的情况进行相应处理了
    if (mService.mController !=null) {
        boolean abort=false ;
        try{
            Intent watchIntent=intent.cloneFilter () ;
            //交给回调对象处理，由它判断是否能继续后面的行程
            abort= ! mService.mController.activityStarting (watchIntent,
aInfo.applicationInfo.packageName) ;
        }.....
        //回调对象决定不启动该Activity。在进行Monkey测试时，可设置黑名单，位
于
        //黑名单中的Activity将不能启动
        if (abort) {
            ....//通知resultRecord
            return START_SUCCESS ;
        }
    }
} //if (mMainStack) 判断结束
//创建一个ActivityRecord对象
ActivityRecord r=new ActivityRecord ( mService, this,
callerApp, callingUid,

```

```
intent, resolvedType, aInfo, mService.mConfiguration,
resultRecord, resultWho, requestCode, componentSpecified) ;
if (outActivity !=null)
outActivity[0]=r ; //保存到输入参数outActivity数组中
if (mMainStack) {
//mResumedActivity代表当前界面显示的Activity
if (mResumedActivity==null
||mResumedActivity.info.applicationInfo.uid !=callingUid) {
//检查调用进程是否有权限切换Application, 相关知识见下文的解释
if ( ! mService.checkAppSwitchAllowedLocked ( callingPid,
callingUid,
"Activity start") ) {
PendingActivityLaunch pal=new PendingActivityLaunch () ;
//如果调用进程没有权限切换Activity, 则只能把这次Activity启动请求保存
起来,
//后续有机会时再启动它
pal.r=r ;
pal.sourceRecord=sourceRecord ;
.....
//所有Pending的请求均保存到AMS mPendingActivityLaunches变量中
mService.mPendingActivityLaunches.add (pal) ;
mDismissKeyguardOnNextActivity=false ;
return START_SWITCHES_CANCELED ;
}
} //if (mResumedActivity==null.....) 判断结束
if (mService.mDidAppSwitch) { //用于控制app switch, 见下文解释
mService.mAppSwitchesAllowedTime=0 ;
} else{
mService.mDidAppSwitch=true ;
}
//启动处于Pending状态的Activity
mService.doPendingActivityLaunchesLocked (false) ;
} //if (mMainStack) 判断结束
//调用startActivityUncheckedLocked函数
err=startActivityUncheckedLocked (r, sourceRecord,
grantedUriPermissions, grantedMode, onlyIfNeeded, true) ;
```

```
.....  
    return err;  
}
```

startActivityLocked函数的主要工作包括：

处理 sourceRecord 及 resultRecord。其中，sourceRecord 表示发起本次请求的 Activity，resultRecord 表示接收处理结果的 Activity（启动一个 Activity 肯定需要它完成某项事情，当目标 Activity 将事情完成后，就需要告知请求者该事情的处理结果）。在一般情况下，sourceRecord 和 resultRecord 应指向同一个 Activity。

处理 app switch。如果 AMS 当前禁止 app switch，则只能把本次启动请求保存起来，以待允许 app switch 时再处理。从代码中可知，AMS 在处理本次请求前，会先调用 doPendingActivityLaunchesLocked 函数，在该函数内部将启动之前因系统禁止 app switch 而保存的 Pending 请求。

调用 startActivityUncheckedLocked 处理本次 Activity 启动请求。

先来看 app switch，它虽然是一个小变量，但是意义重大。

1. 关于resume/stopAppSwitches的介绍

AMS提供了两个函数，用于暂时（注意，是暂时）禁止App切换。为什么会有这种需求呢？因为当某些重要（例如设置账号等）Activity处于前台（即用户当前所见的Activity）时，不希望系统因用户操作之外的原因而切换Activity（例如恰好此时收到来电信号而弹出来电界面）。

先来看stopAppSwitches，其代码如下：

[-->ActivityManagerService.java :
stopAppSwitches]

```
public void stopAppSwitches () {  
    .....//检查调用进程是否有STOP_APP_SWITCHES权限  
    synchronized (this) {  
        //设置一个超时时间，过了该时间，AMS可以重新切换App (switch app) 了  
        mAppSwitchesAllowedTime=SystemClock.uptimeMillis ()  
        +APP_SWITCH_DELAY_TIME ;  
        mDidAppSwitch=false ; //设置mDidAppSwitch为false  
        mHandler.removeMessages (DO_PENDING_ACTIVITY_LAUNCHES_MSG) ;  
        Message msg;//防止应用进程调用了stop却没调用resume，5秒后处理该消息  
        mHandler.obtainMessage (DO_PENDING_ACTIVITY_LAUNCHES_MSG) ;  
        mHandler.sendMessageDelayed (msg, APP_SWITCH_DELAY_TIME) ;  
    }  
}
```

在以上代码中有两点需要注意：

此处控制机制名为app switch，而不是activity switch。为什么呢？因为如果从受保护的Activity中启动另一个Activity，那么这个新Activity的目的应该是针对同一任务，这次启动就不应该受app switch的制约，反而应该对其大开绿灯。目前，在执行Settings中设置设备策略（DevicePolicy）时就会stopAppSwitch。

执行stopAppSwitch后，应用程序应该调用resumeAppSwitches以允许app switch，但是为了防止应用程序有意或无意忘记resume app switch，系统设置了一个超时时间

（5秒），过了此超时时间，系统将处理相应的消息，其内部会resume app switch。再来看resumeAppSwitches函数，其实现代码如下：

[-->ActivityManagerService :
resumeAppSwitches]

```
public void resumeAppSwitches () {  
    ....//检查调用进程是否有STOP_APP_SWITCHES权限  
    synchronized (this) {  
        mAppSwitchesAllowedTime=0 ;  
    }  
    //注意，系统并不在此函数内启动那些被阻止的Activity  
}
```

在 resumeAppSwitches 中 只 设 置 mAppSwitchesAllowedTime 的 值 为 0， 它 并 不 处 理 在 stop 和 resume 这 段 时 间 内 积 攒 起 的 Pending 请 求， 那 么 这 些 请 求 是 在 何 时 被 处 理 的 呢？

从 前 面 代 码 可 知， 如 果 在 执 行 resume app switch 后， 又 有 新 的 请 求 需 要 处 理， 则 先 处 理 那 些 pending 的 请 求（调 用 doPendingActivityLaunchesLocked）。

在 resumeAppSwitches 中 并 未 撤 销 stopAppSwitches 函数 中 设置 的 超 时 消 息， 所 以 在 处 理 那 条 超 时 消 息 的 过 程 中， 也 会 处 理 pending 的 请 求。

在 本 例 中， 由 于 不 考 虑 app switch 的 情 况， 那 么 接 下 来 的 工 作 就 是 调 用 startActivity-UncheckedLocked 函数 来 处 理 本 次 Activity 的 启 动 请 求。 此 时， 我 们 已 经 创建 了 一 个 ActivityRecord 用 于 保 存 目 标 Activity 的 相 关 信 息。

2.startActivityUncheckedLocked函数分析

startActivityUncheckedLocked 函数 很 长， 但 是 目 的 比 较 简 单， 即 为 新 创建 的 Activity-Record 找 到 一 个 合 适 的 Task。 本 例 最 终 的 结 果 是 创建 一 个 新 的 Task， 其 中 startActivity-UncheckedLocked 函数 的

处理逻辑却比较复杂，对它的分析可分为三个阶段，下面先看第一阶段。

(1) startActivityUncheckedLocked分析之一

这部分的代码如下：

[-->ActivityStack.java :
startActivityUncheckedLocked]

```
final int startActivityUncheckedLocked (ActivityRecord r,
ActivityRecord sourceRecord, Uri[] grantedUriPermissions,
int grantedMode, boolean onlyIfNeeded, boolean doResume) {
    //在本例中, sourceRecord为null, onlyIfNeeded为false, doResume
    为true
    final Intent intent=r.intent ;
    final int callingUid=r.launchedFromUid ;
    int launchFlags=intent.getFlags () ;
    //判断是否需要调用因本次Activity启动而被系统移到后台的当前Activity的
    //onUserLeaveHint 函数。可阅读 SDK 文档中关于 Activity
    onUserLeaveHint函数的说明
    mUserLeaving=          (           launchFlags         &
Intent.FLAG_ACTIVITY_NO_USER_ACTION) ==0 ;
    //设置ActivityRecord的delayedResume为true, 本例中的doResume为
    true
    if (!doResume) r.delayedResume=true ;
    //在本例中, notTop为null
    ActivityRecord      notTop=      (           launchFlags         &
Intent.FLAG_ACTIVITY_PREVIOUS_IS_TOP)
    !=0?r : null ;
    if (onlyIfNeeded) {.....//在本例中, 该变量为false, 故略去相关代码
    }
    //根据sourceRecord的情况进行对应处理。你能理解下面这段if/else的判断
    语句吗
```

```
if (sourceRecord==null) {  
    //如果请求的发起者为空，则需要新建一个Task  
    if ((launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) ==0)  
        launchFlags|=Intent.FLAG_ACTIVITY_NEW_TASK ;  
    }else  
        if (sourceRecord.launchMode==ActivityInfo.LAUNCH_SINGLE_INSTANCE) {  
            //如果sourceRecord单独占一个Instance，则新的Activity必然处于另一个  
            Task中  
            launchFlags|=Intent.FLAG_ACTIVITY_NEW_TASK ;  
        }else if (r.launchMode==ActivityInfo.LAUNCH_SINGLE_INSTANCE  
        || r.launchMode==ActivityInfo.LAUNCH_SINGLE_TASK) {  
            //如果启动模式设置了singleTask或singleInstance，则也要创建新Task  
            launchFlags|=Intent.FLAG_ACTIVITY_NEW_TASK ;  
        } //if (sourceRecord==null) 判断结束  
    //如果新Activity和接收结果的Activity不在一个Task中，则不能启动新的  
    Activity  
    if ( r.resultTo !=null && ( launchFlags &  
    Intent.FLAG_ACTIVITY_NEW_TASK) !=0) {  
        sendActivityResultLocked (-1 , r.resultTo, r.resultWho,  
        r.requestCode,  
        Activity.RESULT_CANCELED, null) ;  
        r.resultTo=null ;  
    }  
}
```

startActivityUncheckedLocked第一阶段的工作还算简单，主要确定是否需要为新的Activity创建一个Task，即是否设置FLAG_ACTIVITY_NEW_TASK标志。

接下来看下一阶段的工作。

(2) startActivityUncheckedLocked分析之二

这部分的代码如下：

[-->ActivityStack.java
startActivityUncheckedLocked]

```
boolean addingToTask=false;
TaskRecord reuseTask=null;
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) {
//搜索mHistory, 得到一个ActivityRecord
ActivityRecord taskTop=r.launchMode !=
ActivityInfo.LAUNCH_SINGLE_INSTANCE
?findTaskLocked (intent, r.info)
:findActivityLocked (intent, r.info) ;
if (taskTop !=null) {
.....//这么多复杂的逻辑处理, 无非就是要找到一个合适的Task, 然后对应做一些
//处理。此处不讨论这段代码, 读者可根据工作中的具体情况进行研究
}
} //if (r.resultTo==null) 判断结束
}
```

在本例中，目标Activity首次登场，所以前面的逻辑处理都没有起作用，建议读者根据具体情况分析该段代码。

下面来看startActivityUncheckedLocked第三阶段的工作。

(3) startActivityUncheckedLocked分析之三

这部分的代码如下：

[-->ActivityStack.java
startActivityUncheckLocked]

```
if (r.packageName !=null) {  
    //判断目标Activity是否已经在栈顶，如果是，需要判断是创建一个新的  
    Activity  
    //还是调用onNewIntent (singleTop模式的处理)  
    ActivityRecord          top=topRunningNonDelayedActivityLocked  
(notTop) ;  
    if (top !=null&&r.resultTo==null) {  
        ....//不讨论此段代码  
        } //if (top !=null.....) 结束  
    } else{  
        ....//通知错误  
        return START_CLASS_NOT_FOUND ;  
    }  
    //在本例中，需要创建一个Task  
    boolean newTask=false ;  
    boolean keepCurTransition=false ;  
    if (r.resultTo==null&& ! addingToTask  
&& (launchFlags& Intent.FLAG_ACTIVITY_NEW_TASK) !=0) {  
        if (reuseTask==null) {  
            mService.mCurTask++ ;//AMS中保存了当前Task的数量  
            if (mService.mCurTask<=0) mService.mCurTask=1 ;  
            //为该ActivityRecord设置一个新的TaskRecord  
            r.setTask ( new TaskRecord ( mService.mCurTask, r.info,  
                intent) ,  
                null, true) ;  
        } else r.setTask (reuseTask, reuseTask, true) ;  
        newTask=true ;  
        //下面这个函数为Android 4.0新增的，用于处理  
        FLAG_ACTIVITY_TASK_ON_HOME的情况，  
        //请阅读SDK文档对Intent的相关说明
```

```
moveHomeToFrontFromLaunchLocked (launchFlags) ;  
}else if.....//其他处理情况  
//授权控制。在SDK中启动Activity的函数没有授权设置方面的参数。在实际工  
作中，笔者曾碰  
//到过一个有趣的情况：在开发的一款定制系统中，用浏览器下载了受DRM保护的  
图片，  
//此时要启动Gallery3D来查看该图片，但是由于为DRM目录设置了读写权限，  
而Gallery3D  
//并未声明相关权限，结果抛出异常，导致不能浏览该图片。由于  
startActivity等函数不能设置  
//授权，最终只能修改Gallery3D并为其添加use-permissions项  
if (grantedUriPermissions !=null & & callingUid>0) {  
for (int i=0 ; i<grantedUriPermissions.length ; i++) {  
mService.grantUriPermissionLocked ( callingUid,  
r.packageName,  
grantedUriPermissions[i], grantedMode,  
r.getUriPermissionsLocked () ) ;  
}  
mService.grantUriPermissionFromIntentLocked ( callingUid,  
r.packageName,  
intent, r.getUriPermissionsLocked () ) ;  
//调用startActivityLocked，此时ActivityRecord和TaskRecord均创建  
完毕  
startActivityLocked ( r, newTask, doResume,  
keepCurTransition) ;  
return START_SUCCESS;  
}//startActivityUncheckLocked函数结束
```

startActivityUncheckLocked的第三阶段工作也比较复杂，不过针对本例，它将创建一个新的TaskRecord，并调用startActivityLocked函数进行处理。

下面我们分析startActivityLocked函数。

(4) startActivityLocked函数分析

该函数的实现代码如下：

[-->ActivityStack.java : startActivityLocked]

```
private final void startActivityLocked ( ActivityRecord r,
boolean newTask,
    boolean doResume, boolean keepCurTransition) {
    final int NH=mHistory.size () ;
    int addPos=-1;
    if ( ! newTask ) { // 如果不是新 Task , 则从 mHistory 中找到对应的
ActivityRecord的位置
        .....
    }
    if ( addPos<0) addPos=NH ;
    //否则加到mHistory数组的最后
    mHistory.add (addPos, r) ;
    //设置ActivityRecord的inHistory变量为true , 表示已经加到mHistory
数组中了
    r.putInHistory () ;
    r.frontOfTask=newTask ;
    if ( NH>0) {
        // 判断是否显示 Activity 切换动画之类的事情 , 需要与
WindowManagerService交互
    }
    //最终调用resumeTopActivityLocked
    if (doResume) resumeTopActivityLocked (null) ; //重点分析这个函
数
}
```

在以上列出的startActivityLocked函数中，略去了一部分逻辑处理，这部分内容和Activity之间

的切换动画有关（通过这些动画，使切换过程看起来更加平滑和美观，需和WMS交互）。

提示 笔者认为，此处将Activity切换和动画处理这两个逻辑揉到一起并不合适，但是似乎也没有更合适的地方来进行该工作了。读者不妨自行研读一下该段代码以加深体会。

（5）startActivityUncheckedLocked总结

说实话，startActivityUncheckedLocked函数的复杂度超乎笔者的梦想，只这些函数名就够让人头疼的。但是针对本例而言，相关逻辑的难度还算适中，毕竟这是Activity启动流程中最简单的情况。可用一句话总结本例中startActivityUncheckedLocked函数的功能：创建ActivityRecord和TaskRecord，并将ActivityRecord添加到mHistory末尾，然后调用resumeTopActivityLocked启动它。

下面用一节来分析resumeTopActivityLocked函数。

3.resumeTopActivityLocked函数分析

resumeTopActivityLocked函数的实现代码如下：

[-->ActivityStack.java resumeTopActivityLocked]

:

```
final boolean resumeTopActivityLocked (ActivityRecord prev)
{
    //从mHistory中找到第一个需要启动的ActivityRecord
    ActivityRecord next=topRunningActivityLocked (null) ;
    final boolean userLeaving=mUserLeaving;
    mUserLeaving=false;
    if (next==null) {
        //如果mHistory中没有要启动的Activity，则启动Home
        if (mMainStack) return mService.startHomeActivityLocked () ;
    }
    //在本例中，next将是目标Activity
    next.delayedResume=false;
    .....//和WMS交互，略去
    //将该ActivityRecord从下面几个队列中移除
    mStoppingActivities.remove (next) ;
    mGoingToSleepActivities.remove (next) ;
    next.sleeping=false;
    mWaitingVisibleActivities.remove (next) ;
    //如果当前正在中断一个Activity，需先等待那个Activity pause完毕，然后系统会重新
    //调用resumeTopActivityLocked函数以找到下一个要启动的Activity
    if (mPausingActivity !=null) return false;
    /*****      请      读      者      注      意      *****/
    //①mResumedActivity指向上次启动的Activity，也就是当前界面显示的这个Activity
    //在本例中，当前Activity就是Home界面
    if (mResumedActivity !=null) {
        //先中断Home。这种情况放到最后进行分析
        startPausingLocked (userLeaving, false) ;
        return true;
    }
```

```
//②如果mResumedActivity为空，则一定是系统第一个启动的Activity，读者应能猜测到它就
//是Home
.....//如果prev不为空，则需要通知WMS进行与Activity切换相关的工作
try{
//通知PKMS修改该Package stop状态，详细信息参考4.3.1节的说明
AppGlobals.getPackageManager () .setPackageStoppedState (
next.packageName, false) ;
}.....
if (prev !=null) {
.....//还是和WMS有关，通知它停止绘画
}
if (next.app !=null&&next.app.thread !=null) {
//如果该ActivityRecord已有对应的进程存在，则只需要重启Activity。就
本例而言，
//此进程还不存在，所以要先创建一个应用进程
}else{
//第一次启动
if (!next.hasBeenLaunched) {
next.hasBeenLaunched=true ;
}else{
.....//通知WMS显示启动界面
}
//调用另外一个startSpecificActivityLocked函数
startSpecificActivityLocked (next, true, true) ;
}
return true ;
}
```

resumeTopActivityLocked函数中有两个非常重要的关键点：

如果mResumedActivity不为空，则需要先暂停（pause）这个Activity。由代码中的注释可知，

mResumedActivity代表上一次启动的（即当前正显示的）Activity。现在要启动一个新的Activity，须先停止当前Activity，这部分工作由startPausingLocked函数完成。

mResumedActivity什么时候为空呢？当然是在启动全系统第一个Activity时，即启动Home界面的时候。除此之外，该值都不会为空。

上面代码中先分析了第二个关键点，即mResumedActivity为空的情况。选择先分析此种情况的原因是：如果先分析startPausingLocked，则后续分析会涉及3个进程，即当前Activity所在进程、AMS所在进程及目标进程，分析的难度相当大。

好了，继续我们的分析。resumeTopActivityLocked最后将调用另外一个startSpecific-ActivityLocked，该函数将真正创建一个应用进程。

(1) startSpecificActivityLocked分析

这部分的代码如下：

```
[-->ActivityStack.java :  
startSpecificActivityLocked]
```

```
private final void startSpecificActivityLocked  
(ActivityRecord r,  
    boolean andResume, boolean checkConfig) {  
    //从AMS中查询是否已经存在满足要求的进程（根据processName和uid来查  
    找）  
    //在本例中，查询结果应该为null  
    ProcessRecord app=mService.getProcessRecordLocked  
(r.processName,  
        r.info.applicationInfo.uid) ;  
    //设置启动时间等信息  
    if (r.launchTime==0) {  
        r.launchTime=SystemClock.uptimeMillis () ;  
        if (mInitialStartTime==0) mInitialStartTime=r.launchTime ;  
    }else if (mInitialStartTime==0) {  
        mInitialStartTime=SystemClock.uptimeMillis () ;  
    }  
    //如果该进程存在并已经向AMS注册（例如之前在该进程中启动了其他  
    Activity）  
    if (app !=null&&app.thread !=null) {  
        try{  
            app.addPackage (r.info.packageName) ;  
            //通知该进程中的启动目标Activity  
            realStartActivityLocked (r, app, andResume, checkConfig) ;  
            return ;  
        }.....  
    }  
    //如果该进程不存在，则需要调用AMS的startProcessLocked创建一个应用进  
    程  
    mService.startProcessLocked ( r.processName,  
        r.info.applicationInfo,  
        true, 0, "activity", r.intent.getComponent () , false) ;  
}
```

AMS的startProcessLocked函数将创建一个新的应用进程，下面将分析这个函数。

(2) startProcessLocked分析

这部分内容的代码如下：

[-->ActivityManagerService.java :
startProcessLocked]

```
final ProcessRecord startProcessLocked (String processName,
    ApplicationInfo     info,     boolean     knownToBeDead,     int
intentFlags,
    String hostingType, ComponentName hostingName,
    boolean allowWhileBooting) {
    //根据processName和uid寻找是否已经存在ProcessRecord
    ProcessRecord     app=getProcessRecordLocked     ( processName,
info.uid) ;
    if (app !=null&&app.pid>0) {
        ....//处理相关情况
    }
    String hostingNameStr=hostingName !=null
?hostingName.flattenToString () :null;
    //①处理FLAG_FROM_BACKGROUND标志，见下文解释
    if ( (intentFlags&Intent.FLAG_FROM_BACKGROUND) !=0) {
        if (mBadProcesses.get (info.processName, info.uid) !=null)
return null;
    }else{
        mProcessCrashTimes.remove (info.processName, info.uid) ;
        if (mBadProcesses.get (info.processName, info.uid) !=null) {
            mBadProcesses.remove (info.processName, info.uid) ;
            if (app !=null) app.bad=false ;
        }
    }
    if (app==null) {
        //创建一个ProcessRecord，并保存到mProcessNames中。注意，此时还没有
        创建实际进程
```

```
app=newProcessRecordLocked (null, info, processName) ;  
mProcessNames.put (processName, info.uid, app) ;  
}else app.addPackage (info.packageName) ;  
.....  
//②调用另外一个startProcessLocked函数  
startProcessLocked (app, hostingType, hostingNameStr) ;  
return (app.pid !=0) ?app : null;  
}
```

在以上代码中列出两个关键点，其中第一个和FLAG_FROM_BACKGROUND有关，相关知识点如下：

FLAG_FROM_BACKGROUND标志发起这次启动的Task属于后台任务。很显然，手机中没有界面供用户操作位于后台Task中的Activity。如果没有设置该标志，那么这次启动请求就是由前台Task因某种原因而触发的（例如，用户单击某个按钮）。

如果一个应用进程在1分钟内连续崩溃超过2次，则AMS会将其ProcessRecord加入所谓的mBadProcesses中。一个应用崩溃后，系统会弹出一个警告框以提醒用户。但是，如果一个后台Task启动了一个“Bad Process”，然后该Process崩溃，结果弹出一个警告框，那么用户就会觉得很奇怪：“为什么突然弹出一个框？”因此，此处将禁止后台Task启动“Bad Process”。如果用户主动选

择启动（例如，单击一个按钮），则不能禁止该操作，并且要把应用进程从mBadProcesses中移除，以给它们“重新做人”的机会。当然，若作为测试工作者，要是该应用每次启动后都会崩溃，就需要其不停地去启动该应用以达到测试的目的。

提示 这其实是一种安全机制，防止不健全的程序不断启动可能会崩溃的组件，但是这种机制并不限制用户的行为。

下面来看第二个关键点，即另一个startProcessLocked函数，其代码如下：

[-->ActivityManagerService.java :
startProcessLocked]

```
private final void startProcessLocked (ProcessRecord app,  
String hostingType, String hostingNameStr) {  
if (app.pid>0&&app.pid !=MY_PID) {  
synchronized (mPidssSelfLocked) {  
mPidssSelfLocked.remove (app.pid) ;  
mHandler.removeMessages (PROC_START_TIMEOUT_MSG, app) ;  
}  
app.pid=0 ;  
}  
//mProcessesOnHold用于保存那些在系统还没有准备好就提前请求启动的  
ProcessRecord  
mPidssOnHold.remove (app) ;  
updateCpuStats () ;
```

```
System.arraycopy ( mProcDeaths , 0 , mProcDeaths , 1 ,
mProcDeaths.length-1) ;
mProcDeaths[0]=0 ;
try{
int uid=app.info.uid ;
int[]gids=null ;
try{//从PKMS中查询该进程所属的gid
gids=mContext.getPackageManager () .getPackageGids (
app.info.packageName) ;
}.....
.....//工厂测试
int debugFlags=0 ;
if ( (app.info.flags&ApplicationInfo.FLAG_DEBUGGABLE) !=0) {
debugFlags|=Zygote.DEBUG_ENABLE_DEBUGGER ;
debugFlags|=Zygote.DEBUG_ENABLE_CHECKJNl ;
}....//设置其他一些debugFlags
//发送消息给Zygote，它将派生一个子进程，该子进程执行ActivityThread
的main函数
//注意，我们传递给Zygote的参数并没有包含任何与Activity相关的信息。现
在仅仅启动
//一个应用进程
Process.ProcessStartResult startResult=
Process.start ("android.app.ActivityThread",
app.processName, uid, uid, gids, debugFlags,
app.info.targetSdkVersion, null) ;
//电量统计项
BatteryStatsImpl bs=app.batteryStats.getBatteryStats () ;
synchronized (bs) {
if (bs.isOnBattery ()) app.batteryStats.incStartsLocked () ;
}
//如果该进程为persistent，则需要通知Watchdog，实际上
processStarted内部只
//关心刚才创建的进程是不是com.android.phone
if (app.persistent) {
Watchdog.getInstance () .processStarted (app.processName,
startResult.pid) ;
```

```
        }
        app.pid=startResult.pid ;
        app.usingWrapper=startResult.usingWrapper ;
        app.removed=false ;
        synchronized (mPidSelfLocked) {
            // 以 pid 为 key , 将代表该进程的 ProcessRecord 对象加入到
            // mPidSelfLocked中保管
            thismPidSelfLocked.put (startResult.pid, app) ;
            //发送一个超时消息，如果这个新创建的应用进程10秒内没有和AMS交互，则可
            //断定
            //该应用进程启动失败
            Message msg=mHandler.obtainMessage
            (PROC_START_TIMEOUT_MSG) ;
            msg.obj=app ;
            //正常的超时时间为10秒。不过如果该应用进程通过valgrind加载，则延长到
            //300秒
            //valgrind是Linux平台上一款检查内存泄露的程序，被加载的应用将在它的环
            //境中工作，
            //这项工作需耗费较长时间。读者可查询valgrind的用法
            mHandler.sendMessageDelayed (msg, startResult.usingWrapper
            ?PROC_START_TIMEOUT_WITH_WRAPPER : PROC_START_TIMEOUT) ;
        }.....
    }
}
```

startProcessLocked通过发送消息给Zygote以派生一个应用进程[1]，读者仔细研究所发消息的内容，会发现此处并未设置和Activity相关的信息，也就是说，该进程启动后，将完全不知道自己要干什么，怎么办？下面就此进行分析。

4.startActivity分析之半程总结

至此，我们已经走完了startActivity分析之旅的一半路程，一路走来，我们越过了很多“险滩恶途”。此处用图6-14来记录前半程中的各个关键点。

图6-14列出了针对本例的调用顺序，其中对每个函数的大体功能也做了简单描述。

注意 图6-14中的调用顺序及功能说明只是针对本例而言的。读者以后可结合具体情况再深入研究其中的内容。

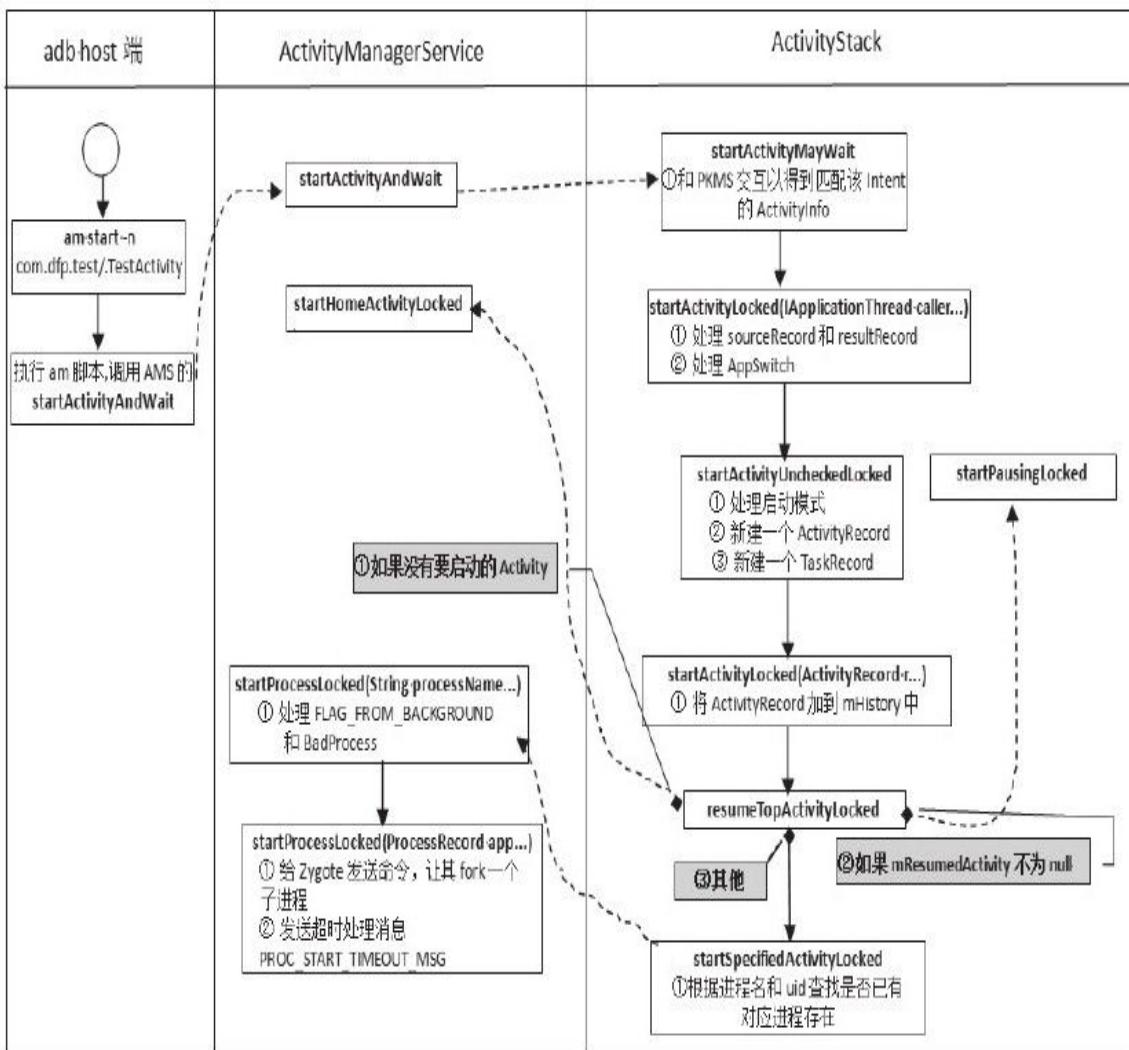


图 6-14 startActivity 半程总结

5. 应用进程的创建及初始化

如前所述，应用进程的入口是ActivityThread的main函数，它是在主线程中执行的，其代码如下：

[-->ActivityThread.java : main]

```
public static void main (String[]args) {  
    SamplingProfilerIntegration.start () ;  
    //和调试及strictMode有关  
    CloseGuard.setEnabled (false) ;  
    //设置进程名为"<pre-initialized>"  
    Process.setArgV0 ("<pre-initialized>") ;  
    //准备主线程消息循环  
    Looper.prepareMainLooper () ;  
    if (sMainThreadHandler==null)  
        sMainThreadHandler=new Handler () ;  
    //创建一个ActivityThread对象  
    ActivityThread thread=new ActivityThread () ;  
    //①调用attach函数，注意其参数值为false  
    thread.attach (false) ;  
    Looper.loop () ;//进入主线程消息循环  
    throw new RuntimeException ("Main thread loop unexpectedly  
    exited") ;  
}
```

在main函数内部将创建一个消息循环Loop，接着调用ActivityThread的attach函数，最终将主线程加入消息循环。

我们在分析AMS的setSystemProcess时曾分析过ActivityThread的attach函数，那时传入的参数值为true。现在来看设置其为false的情况。

[-->ActivityThread.java : attach]

```
private void attach (boolean system) {  
    sThreadLocal.set (this) ;  
    mSystemThread=system ;  
    if (!system) {
```

```
ViewRootImpl.addFirstDrawHandler (new Runnable () {
    public void run () {
        ensureJitEnabled () ;
    }
}) ;
//设置在DDMS中看到的本进程的名字为"<pre-initialized>"  

    android.ddm.DdmHandleAppName.setAppName ( "<pre-initialized  

>" ) ;
// 设置 RuntimeInit 的 mApplicationObject 参数 , 后续会介绍  

RuntimeInit类
RuntimeInit.setApplicationObject (mAppThread.asBinder () ) ;
//获取和AMS交互的Binder客户端
IActivityManager mgr=ActivityManagerNative.getDefault () ;
try{
    //①调用AMS的attachApplication, mAppThread为ApplicationThread
    //它是应用进程和AMS交互的接口
    mgr.attachApplication (mAppThread) ;
}.....
}else.....//system process的处理
ViewRootImpl.addConfigCallback (new ComponentCallbacks2 ()
{....//添加回调函数} ) ;
}
```

我们知道，AMS创建一个应用进程后，会设置一个超时时间（一般是10秒）。如果超过这个时间，应用进程还没有和AMS交互，则断定该进程创建失败。所以，应用进程启动后，需要尽快和AMS交互，即调用AMS的attachApplication函数。在该函数内部将调用attachApplicationLocked，所以此处直接分析

attachApplicationLocked，先看其第一阶段的工作。

(1) attachApplicationLocked分析之一

这部分的代码如下：

[-->ActivityManagerService.java :
attachApplicationLocked]

```
private final boolean attachApplicationLocked  
(IApplicationThread thread,  
    int pid) { //此pid代表调用进程的pid  
    ProcessRecord app;  
    if (pid != MY_PID & & pid >= 0) {  
        synchronized (mPidSelfLocked) {  
            app = mPidSelfLocked.get (pid); // 根据 pid 查找 对应 的  
            ProcessRecord 对象  
        }  
    } else  
        app = null;  
    /*
```

如果该应用进程由AMS启动，则它一定在AMS中有对应的ProcessRecord，读者可回顾前面创建

应用进程的代码：AMS先创建了一个ProcessRecord对象，然后才发命令给Zygote。

如果此处app为null，则表示AMS没有该进程的记录，故需要“杀死”它

```
*/  
if (app == null) {  
    if (pid > 0 & & pid != MY_PID) //如果pid大于零，且不是system_server  
    进程，则  
        //Quietly (即不打印任何输出) “杀死”process  
        Process.killProcessQuiet (pid);  
    else{
```

```
//调用ApplicationThread的scheduleExit函数。应用进程完成处理工作后  
//将退出运行  
//为何不像上面一样直接“杀死”它呢？读者可查阅Linux pid相关的知识自行解  
答
```

```
    thread.scheduleExit () ;  
}  
return false ;  
}  
/*
```

判断app的thread是否为空，如果不为空，则表示该ProcessRecord对象还未和一个

应用进程绑定。注意，app是根据pid查找到的，如果旧进程没有被“杀死”，则系统不会重用

该pid。为什么此处会出现ProcessRecord thread不为空的情况呢？见下面代码的注释说明

```
*/  
if ( app.thread != null ) handleAppDiedLocked ( app, true,  
true) ;  
String processName=app.processName ;  
try{  
/*
```

创建一个应用进程“讣告”接收对象。当应用进程退出时，该对象的binderDied将被调

用。这样，AMS就能做相应处理。binderDied函数将在另外一个线程中执行，其内部也会

调用handleAppDiedLocked。假如用户在binderDied被调用之前又启动一个进程，

那么就会出现以上代码中app.thread不为null的情况。这是多线程环境中常出现的

情况，不熟悉多线程编程的读者要仔细体会。

```
*/  
AppDeathRecipient adr=new AppDeathRecipient ( pp, pid,  
thread) ;  
thread.asBinder () .linkToDeath (adr, 0) ;  
app.deathRecipient=adr ;  
}.....
```

```
//设置该进程的调度优先级和oom_adj相关的成员  
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 ;  
//启动成功，从消息队列中撤销PROC_START_TIMEOUT_MSG消息  
mHandler.removeMessages (PROC_START_TIMEOUT_MSG, app) ;
```

attachApplicationLocked第一阶段的工作比较简单：

设置代表该应用进程的ProcessRecord对象的一些成员变量，如用于和应用进程交互的thread对象、进程调度优先级及oom_adj的值等。

从消息队列中撤销PROC_START_TIMEOUT_MSG。至此，该进程启动成功，但是这一阶段的工作仅针对进程本身（如设置调度优先级，oom_adj等），还没有涉及和Activity启动相关的内容，这部分工作将在第二阶段完成。

(2) attachApplicationLocked分析之二

这部分的代码如下：

[-->ActivityManagerService.java attachApplicationLocked]

:

```
.....  
//system_server早就启动完毕，所以normalMode为true  
boolean    normalMode=mProcessesReady||isAllowedWhileBooting  
(app.info) ;  
/*  
我们在6.2.3节中分析过generateApplicationProvidersLocked函数，  
在该函数内部将查询（根据进程名，uid确定）PKMS以获取需运行在该进程中的  
ContentProvider  
*/  
List providers=normalMode?generateApplicationProvidersLocked  
(app) :null;  
try{  
int testMode=IApplicationThread.DEBUG_OFF ;  
if (mDebugApp !=null&&mDebugApp.equals (processName) ) {  
.....//处理debug选项  
}  
.....//处理Profile  
boolean isRestrictedBackupMode=false ;  
.....//  
//dex化对应的APK包  
ensurePackageDexOpt (app.instrumentationInfo !=null?  
app.instrumentationInfo.packageName : app.info.packageName) ;  
//如果设置了Instrumentation类，该类所在的Package也需要dex化  
if (app.instrumentationClass !=null)  
ensurePackageDexOpt (app.instrumentationClass.getPackageName  
() ) ;  
ApplicationInfo appInfo=app.instrumentationInfo !=null  
?app.instrumentationInfo : app.info ;  
//查询该Application使用的CompatibilityInfo  
app.compat=compatibilityInfoForPackageLocked (appInfo) ;  
if (profileFd !=null) //用于记录性能文件  
profileFd=profileFd.dup () ;
```

```
//①通过ApplicationThread和应用进程交互，调用其bindApplication函数
    thread.bindApplication (processName, appInfo, providers,
    app.instrumentationClass, profileFile, profileFd,
    profileAutoStop, app.instrumentationArguments,
    app.instrumentationWatcher, testMode,
    isRestrictedBackupMode|| ! normalMode, app.persistent,
    mConfiguration, app.compat, getCommonServicesLocked () ,
    mCoreSettingsObserver.getCoreSettingsLocked () ) ;
    //updateLruProcessLocked函数以后再作分析
    updateLruProcessLocked (app, false, true) ;
    //记录两个时间
    app.lastRequestedGc=app.lastLowMemory=SystemClock.uptimeMillis
is () ;
}.....//try结束
...//从mProcessesOnHold和mPersistentStartingProcesses中删除相关信息
mPersistentStartingProcesses.remove (app) ;
mProcessesOnHold.remove (app) ;
```

由以上代码可知，第二阶段的工作主要是为调用ApplicationThread的bindApplication做准备，我们将在后面的章节中分析该函数的具体内容。此处先来看它的原型。

```
/*
正如我们在前面分析时提到的，刚创建的这个进程并不知道自己的历史使命是什么，甚至连自己的
进程名都不知道，只能设为<pre-initialized>。其实，Android应用进程的历史使命是
AMS在其启动后才赋予它的，这一点和我们理解的一般意义上的进程不太一样。根据之前的介绍，
```

Android的组件应该运行在Android运行环境中。从OS角度来说，该运行环境需要和一个进程绑定。

所以，创建应用进程这一步只是创建了一个能运行Android运行环境的容器，而我们的工作实际上

还远未结束。

bindApplication的功能就是创建并初始化位于该进程中的Android运行环境
*/
public final void bindApplication (
String processName, //进程名，一般是package名
ApplicationInfo appInfo, //该进程对应的ApplicationInfo
List<ProviderInfo>providers, //在该Package中声明的Provider信息

ComponentName instrumentationName, //和instrumentation有关
//下面3个参数和性能统计有关
String profileFile,
ParcelFileDescriptor profileFd, boolean autoStopProfiler,
//这两个和Instrumentation有关，在本例中，这几个参数暂时都没有用
Bundle instrumentationArgs,
IInstrumentationWatcher instrumentationWatcher,
int debugMode, //调试模式
boolean isRestrictedBackupMode,
boolean persistent, //该进程是否是为persistent进程
Configuration config, //当前的配置信息，如屏幕大小和语言等
CompatibilityInfo compatInfo, //兼容信息
//AMS将常用的Service信息传递给应用进程，目前传递的Service信息只有
PKMS、
//WMS 及 AlarmManagerService 。 读者可参看 AMS
getCommonServicesLocked函数
Map<String, IBinder>services,
Bundle coreSettings) //核心配置参数，目前仅有“long_press”值

对bindApplication的原型分析就到此为止，再来看attachApplicationLocked最后一阶段的工作。

(3) attachApplicationLocked分析之三

这部分的代码如下：

[-->ActivityManagerService.java :
attachApplicationLocked]

```
boolean badApp=false;
boolean didSomething=false;
/*
至此，应用进程已经准备好了Android运行环境，下面这条调用代码将返回
ActivityStack中
第一个需要运行的ActivityRecord。由于多线程的原因，能保证得到的hr就是
我们的目标
Activity吗？
*/
ActivityRecord           hr=mMainStack.topRunningActivityLocked
(null) ;
if (hr !=null&&normalMode) {
//需要根据processName和uid等确定该Activity是否与目标进程有关
if (hr.app==null&&app.info.uid==hr.info.applicationInfo.uid
&&processName.equals (hr.processName) ) {
try{
//调用AS的realStartActivityLocked启动该Activity，最后两个参数为
true
if ( mMainStack.realStartActivityLocked ( hr,   app,   true,
true) ) {
didSomething=true ;
}
}catch (Exception e) {
badApp=true ; //设置badApp为true
}
}else{
```

```
//如果hr和目标进程无关，则调用ensureActivitiesVisibleLocked函数处理它
    mMainStack.ensureActivitiesVisibleLocked(    hr,    null,
processName, 0) ;
}
}//if (hr !=null & &normalMode) 判断结束
//mPendingServices 存储那些因目标进程还未启动而处于等待状态的
ServiceRecord
if (!badApp&&mPendingServices.size () >0) {
ServiceRecord sr=null;
try{
for (int i=0 ; i<mPendingServices.size () ; i++) {
sr=mPendingServices.get (i) ;
//和Activity不一样的是，如果Service不属于目标进程，则暂不处理
if (app.info.uid !=sr.appInfo.uid
|| !processName.equals (sr.processName) ) continue; //继续循环
//该Service将运行在目标进程中，所以从mPendingService中移除它
mPendingServices.remove (i) ;
i-- ;
//处理此Service的启动，以后再作分析
realStartServiceLocked (sr, app) ;
didSomething=true; //设置该值为true
}
}
}
}.....
.....//启动等待的BroadcastReceiver
.....//启动等待的BackupAgent，相关代码类似Service的启动
if (badApp) {
//如果以上几个组件启动有错误，则设置badApp为true。此处将调用
handleAppDiedLocked
//进行处理。该函数我们以后再作分析
handleAppDiedLocked (app, false, true) ;
return false ;
}
/*

```

调整进程的oom_adj值。didSomething表示在以上流程中是否启动了Activity或其他组件。

如果启动了任一组件，则didSomething为true。读者以后会知道，这里的启动只是向

应用进程发出对应的指令，客户端进程是否成功处理还是未知数。基于这种考虑，此处不宜

马上调节进程的oom_adj。

读者可简单地把oom_adj看做一种优先级。如果一个应用进程没有运行任何组件，那么当内存

出现不足时，该进程是最先被系统“杀死”的。反之，如果一个进程运行的组件越多，那么它就

越不易被系统“杀死”以回收内存。updateOomAdjLocked就是根据该进程中组件的情况对应

调节进程的oom_adj值的。

```
*/  
if (!didSomething) updateOomAdjLocked ();  
return true;  
}
```

attachApplicationLocked第三阶段的工作就是通知应用进程启动Activity和Service等组件，其中用于启动Activity的函数是ActivityStack realStartActivityLocked。

此处先分析应用进程的bindApplication，该函数为应用进程绑定一个Application。

提示 还记得AMS中System Context执行的两次init吗？第二次init的功能就是将Context和对应的Application绑定在一起。

(4) ApplicationThread的bindApplication分析

bindApplication 在 ApplicationThread 中的实现，其代码如下：

[-->ActivityThread.java : bindApplication]

```
public final void bindApplication (...) {  
    if (services != null) //保存AMS传递过来的系统Service信息  
        ServiceManager.initServiceCache (services) ;  
    //向主线程消息队列添加SET_CORE_SETTINGS消息  
    setCoreSettings (coreSettings) ;  
    //创建一个AppBindData对象，其实就是用来存储一些参数  
    AppBindData data=new AppBindData () ;  
    data.processName=processName ;  
    data.appInfo=appInfo ;  
    data.providers=providers ;  
    data.instrumentationName=instrumentationName ;  
    .....//将AMS传过来的参数保存到AppBindData中  
    //向主线程发送H.BIND_APPLICATION消息  
    queueOrSendMessage (H.BIND_APPLICATION, data) ;  
}
```

由以上代码可知，ApplicationThread接收到来自AMS的指令后，会将指令中的参数封装到一个数据结构中，然后通过发送消息的方式转交给主线程去处理。BIND_APPLICATION 最终将由 handleBindApplication 函数处理。该函数并不复杂，但是其中有些点是值得关注的，这些点主要是初始化应用进程的一些参数。
handleBindApplication函数的代码如下：

[-->ActivityThread.java
handleBindApplication]

```
private void handleBindApplication (AppBindData data) {  
    mBoundApplication=data ;  
    mConfiguration=new Configuration (data.config) ;  
    mCompatConfiguration=new Configuration (data.config) ;  
    //初始化性能统计对象  
    mProfiler=new Profiler () ;  
    mProfiler.profileFile=data.initProfileFile ;  
    mProfiler.profileFd=data.initProfileFd ;  
    mProfiler.autoStopProfiler=data.initAutoStopProfiler ;  
    //设置进程名。从此，之前那个名为“<pre_initialized>”的进程终于有了  
正式的名字  
    Process.setArgV0 (data.processName) ;  
    android.ddm.DdmHandleAppName.setAppName (data.processName) ;  
    if (data.persistent) {  
        //对于persistent的进程，在低内存设备上，不允许其使用硬件加速显示  
        Display display=  
        WindowManagerImpl.getDefault () .getDefaultDisplay () ;  
        //当内存大于512MB，或者屏幕尺寸大于1024像素*600像素时，可以使用硬件加  
速  
        if (!ActivityManager.isHighEndGfx (display) )  
            HardwareRenderer.disable (false) ;  
    }  
    //启动性能统计  
    if (mProfiler.profileFd !=null)  
        mProfiler.startProfiling () ;  
    //如果目标SDK版本小于12，则设置AsyncTask使用pool executor，否则使  
用  
    //serialized executor。这些executor涉及Java Concurrent类，对此  
不熟悉的读者  
    //请自行学习和研究  
    if (data.appInfo.targetSdkVersion<=12)
```

```
    AsyncTask.setDefaultExecutor  
    (AsyncTask.THREAD_POOL_EXECUTOR) ;  
    //设置timezone  
    TimeZone.setDefault (null) ;  
    //设置语言  
    Locale.setDefault (data.config.locale) ;  
    //设置资源及兼容模式  
    applyConfigurationToResourcesLocked (data.config,  
    data.compatInfo) ;  
    applyCompatConfiguration () ;  
    //根据传递过来的ApplicationInfo创建一个对应的LoadApk对象  
    data.info=getPackageInfoNoCheck (data.appInfo,  
    data.compatInfo) ;  
    //对于系统APK，如果当前系统为userdebug/eng版，则需要使能dropbox的日  
    志记录  
    if ( (data.appInfo.flags &  
        (ApplicationInfo.FLAG_SYSTEM |  
        ApplicationInfo.FLAG_UPDATED_SYSTEM_APP) ) !=0) {  
        StrictModeconditionallyEnableDebugLogging () ;  
    }  
    /*  
     * 如目标SDK版本大于9，则不允许在主线程使用网络操作（如Socket connect  
     * 等），否则抛出  
     * NetworkOnMainThreadException，这么做的目的是防止应用程序在主线程中  
     * 因网络操作执行  
     * 时间过长而造成用户体验下降。说实话，没有必要进行这种限制，在主线程中是否  
     * 进行网络操作  
     * 是应用的事情。再说，Socket也可作为进程间通信的手段，在这种情况下，网络  
     * 操作耗时很短。  
     * 作为系统，不应该设置这种限制。另外，Goolge可以提供一些开发指南或规范来  
     * 指导开发者，  
     * 而不应如此限制。  
    */  
    if (data.appInfo.targetSdkVersion>9)  
        StrictMode.enableDeathOnNetwork () ;  
    //如果没有设置屏幕密度，则为Bitmap设置默认的屏幕密度
```

```
if ( (data.appInfo.flags & ApplicationInfo.FLAG_SUPPORTS_SCREEN_DENSITIES) ==0)
Bitmap.setDefaultDensity (DisplayMetrics.DENSITY_DEFAULT) ;
if (data.debugMode != IApplicationThread.DEBUG_OFF) {
.....//调试模式相关处理
}
IBinder b=ServiceManager.getService
(Context.CONNECTIVITY_SERVICE) ;
IConnectivityManager service=
IConnectivityManager.Stub.asInterface (b) ;
try{
//设置HTTP代理信息
ProxyProperties proxyProperties=service.getProxy () ;
Proxy.setHttpProxySystemProperty (proxyProperties) ;
}catch (RemoteException e) {}
if (data.instrumentationName !=null) {
//在正常情况下，此条件不满足
}else{
//创建Instrumentation对象，在正常情况都在这个条件下执行
mInstrumentation=new Instrumentation () ;
}
//如果Package中声明了FLAG_LARGE_HEAP，则可跳过虚拟机的内存限制，放心使用内存
if ( (data.appInfo.flags & ApplicationInfo.FLAG_LARGE_HEAP) !=
=0)
dalvik.system.VMRuntime.getRuntime () .clearGrowthLimit () ;
//创建一个Application，data.info为LoadedApk类型，在其内部会通过Java反射机制
//创建一个在该APK AndroidManifest.xml中声明的Application对象
Application app=data.info.makeApplication (
data.restrictedBackupMode, null) ;
//mInitialApplication保存该进程中第一个创建的Application
mInitialApplication=app ;
//安装本Package中携带的ContentProvider
if (! data.restrictedBackupMode) {
List<ProviderInfo>providers=data.providers ;
```

```
if (providers !=null) {  
    //installContentProviders我们已经分析过了  
    installContentProviders (app, providers) ;  
    mH.sendEmptyMessageDelayed (H.ENABLE_JIT, 10*1000) ;  
}  
}  
//调用Application的onCreate函数，做一些初始工作  
mInstrumentation.callApplicationOnCreate (app) ;  
}
```

由以上代码可知，bindApplication函数将设置一些初始化变量，其中最重要的有：

创建一个Application对象，该对象是本进程中运行的第一个Application。

如果该Application有ContentProvider，则应安装它们。

提示 从以上代码中可知，ContentProvider的创建于bindApplication函数中，其时机早于其他组件的创建。

(5) 应用进程的创建及初始化总结

本节从应用进程的入口函数main开始，分析了应用进程和AMS之间的两次重要交互，它们分别是：

在应用进程启动后，需要尽快调用AMS的attachApplication函数，该函数是这个刚创建的应用进程第一次和AMS交互。此时的它还默默“无名”，连一个确定的进程名都没有。不过没关系，attachApplication函数将根据创建该应用进程之前所保存的ProcessRecord为其准备一切“手续”。

attachApplication准备好一切后，将调用应用进程的bindApplication函数，在该函数内部将发消息给主线程，最终该消息由handleBindApplication处理。handle-BindApplication将为该进程设置进程名，初始化一些策略和参数信息等。另外，它还创建一个Application对象。同时，如果该Application声明了ContentProvider，还需要为该进程安装ContentProvider。

提示 这个流程有点类似生孩子，一般生之前需要到医院去登记，生完后又需去注册户口，如此这般，这个孩子才会在社会中有合法的身份。

6.ActivityStack realStartActivityLocked分析

如前所述，AMS调用完bindApplication后，将通过realStartActivityLocked启动Activity。在此之前，要创建完应用进程并初始化Android运行环境（除此之外，连Content-Provider都安装好了）。

[-->ActivityStack.java realStartActivityLocked]

:

```
//注意，在本例中该函数的最后两个参数的值都为true
final boolean realStartActivityLocked ( ActivityRecord r,
ProcessRecord app,
boolean andResume, boolean checkConfig ) throws
RemoteException{
    r.startFreezingScreenLocked (app, 0) ;
    mService.mWindowManager.setAppVisibility (r, true) ;
    if (checkConfig) {
        .....//处理Config发生变化的情况
        mService.updateConfigurationLocked (config, r, false) ;
    }
    r.app=app ;
    app.waitingToKill=null ;
    //将ActivityRecord加到ProcessRecord的activities中保存
    int idx=app.activities.indexOf (r) ;
    if (idx<0) app.activities.add (r) ;
    //更新进程的调度优先级等，以后再分析该函数
    mService.updateLruProcessLocked (app, true, true) ;
    try{
        List<ResultInfo>results=null ;
        List<Intent>newIntents=null ;
        if (andResume) {
            results=r.results ;
            newIntents=r.newIntents ;
        }
        if (r.isHomeActivity) mService.mHomeProcess=app ;
        //看看是否有dex对应Package的需要
        mService.ensurePackageDexOpt (
            r.intent.getComponent () .getPackageName () ) ;
        r.sleeping=false ;
        r.forceNewConfig=false ; .....
        //①通知应用进程启动Activity
```

```
    app.thread.scheduleLaunchActivity ( new Intent ( r.intent ) ,  
    r,  
        System.identityHashCode ( r ) , r.info,  
    mService.mConfiguration,  
        r.compat, r.icicle, results, newIntents, !andResume,  
        mService.isNextTransitionForward ( ) , profileFile,  
    profileFd,  
        profileAutoStop) ;  
    if ( app.info.flags &  
ApplicationInfo.FLAG_CANT_SAVE_STATE) !=0) {  
.....//处理heavy-weight的情况  
}  
}  
}  
}.....//try结束  
r.launchFailed=false;  
.....  
if (andResume) {  
    r.state=ActivityState.RESUMED ;  
    r.stopped=false ;  
    mResumedActivity=r ; //设置mResumedActivity为目标Activity  
    r.task.touchActiveTime () ;  
    //添加该任务到近期任务列表中  
    if (mMainStack) mService.addRecentTaskLocked (r.task) ;  
    //②关键函数，见下文分析  
    completeResumeLocked (r) ;  
    //如果在这些过程中，用户按了Power键，怎么办？  
    checkReadyForSleepLocked () ;  
    r.icicle=null ;  
    r.haveState=false ;  
}.....  
//启动系统设置向导Activity，当系统更新或初次使用时需要进行配置  
if (mMainStack) mService.startSetupActivityLocked () ;  
return true ;  
}
```

在以上代码中有两个关键函数，分别是：scheduleLaunchActivity 和 completeResumeLocked。其中，scheduleLaunchActivity用于和应用进程交互，通知它启动目标Activity。而completeResumeLocked将继续AMS的处理流程。先来看第一个关键函数。

(1) scheduleLaunchActivity函数分析

[-->ActivityThread.java :
scheduleLaunchActivity]

```
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 () ;  
.....//保存AMS发送过来的参数信息  
//向主线程发送消息，该消息的处理在handleLaunchActivity中进行  
queueOrSendMessage (H.LAUNCH_ACTIVITY, r) ;  
}
```

[-->ActivityThread.java : handleMessage]

```
public void handleMessage (Message msg) {  
switch (msg.what) {
```

```
case LAUNCH_ACTIVITY : {  
    ActivityClientRecord r= (ActivityClientRecord) msg.obj ;  
    //根据ApplicationInfo得到对应的PackageInfo  
    r.packageInfo=getPackageInfoNoCheck (  
        r.activityInfo.applicationInfo, r.compatInfo) ;  
    //调用handleLaunchActivity处理  
    handleLaunchActivity (r, null) ;  
    }break ;  
.....}
```

[-->ActivityThread.java :
handleLaunchActivity]

```
private void handleLaunchActivity (ActivityClientRecord r,  
Intent customIntent) {  
    unscheduleGcIdler () ;  
    if (r.profileFd !=null) {.....//略去}  
    handleConfigurationChanged (null, null) ; /*  
     ①创建Activity，通过Java反射机制创建目标Activity，将在内部完成  
     Activity生命周期  
     的前两步，即调用其onCreate和onStart函数。至此，我们的目标  
     com.dfp.test.TestActivity  
     创建完毕  
    */  
    Activity a=performLaunchActivity (r, customIntent) ;  
    if (a !=null) {  
        r.createdConfig=new Configuration (mConfiguration) ;  
        Bundle oldState=r.state ;  
        //②调用handleResumeActivity，其内部有个关键点，见下文分析  
        handleResumeActivity (r.token, false, r.isForward) ;  
        if (! r.activity.mFinished&&r.startsNotResumed) {  
            .....//  
            .  
            r.paused=true ;
```

```
    }else{
        //如果启动错误，通知AMS
        ActivityManagerNative.getDefault ()
            .finishActivity (r.token, Activity.RESULT_CANCELED, null) ;
    }
}
```

handleLaunchActivity的工作包括：

首先调用performLaunchActivity，该在函数内部通过Java反射机制创建目标Activity，然后调用它的onCreate及onStart函数。

调用handleResumeActivity，会在其内部调用目标Activity的onResume函数。除此

之外，handleResumeActivity还完成了一件很重要的事情，具体见下面的代码：

[-->ActivityThread.java :
handleResumeActivity]

```
final void handleResumeActivity ( IBinder token, boolean
clearHide,
        boolean isForward) {
    unscheduleGcIdler () ;
    //内部调用目标Activity的onResume函数
    ActivityClientRecord     r=performResumeActivity     ( token,
clearHide) ;
    if (r !=null) {
        final Activity a=r.activity ;
        final int forwardBit=isForward?
```

```
WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARD_NAVIGATION  
: 0 ;  
.....  
if (!r.onlyLocalRequest) {  
    //将上面完成onResume的Activity保存到mNewActivities中  
    r.nextIdle=mNewActivities ;  
    mNewActivities=r ;  
    //①向消息队列中添加一个Idler对象  
    Looper.myQueue () .addIdleHandler (new Idler () ) ;  
}  
r.onlyLocalRequest=false ;  
.....  
}
```

根据第2章对MessageQueue的分析，当消息队列中没有其他要处理的消息时，将处理以上代码中通过addIdleHandler添加的Idler对象，也就是说，Idler对象的优先级最低，这是不是说它的工作不重要呢？非也。至少在handleResumeActivity函数中添加的这个Idler并不简单，其代码如下：

[-->ActivityThread.java : Idler]

```
private class Idler implements MessageQueue.IdleHandler{  
    public final boolean queueIdle () {  
        ActivityClientRecord a=mNewActivities ;  
        boolean stopProfiling=false ;  
        .....  
        if (a !=null) {  
            mNewActivities=null ;  
            IActivityManager am=ActivityManagerNative.getDefault () ;  
            ActivityClientRecord prev ;  
            do{
```

```
if (a.activity !=null & & !a.activity.mFinished) {  
    //调用AMS的activityIdle  
    am.activityIdle (a.token, a.createdConfig, stopProfiling) ;  
    a.createdConfig=null;  
}  
prev=a ;  
a=a.nextIdle ;  
prev.nextIdle=null ;  
}while (a !=null) ; //do循环结束  
}//if (a !=null) 判断结束  
.....  
ensureJitEnabled () ;  
return false ;  
}//queueIdle函数结束  
}
```

由以上代码可知，Idler将为那些已经完成onResume的Activity调用AMS的activityIdle函数。该函数是Activity成功创建并启动的流程中与AMS交互的最后一步。虽然对应用进程来说，Idler处理的优先级最低，但AMS似乎不这么认为，因为它还设置了超时等待，以处理应用进程没有及时调用activityIdle的情况。这个超时等待即由realStartActivityLocked中最后一个关键点completeResumeLocked函数设置。

(2) completeResumeLocked函数分析

这部分的代码具体如下：

[-->ActivityStack.java
completeResumeLocked]

```
private final void completeResumeLocked ( ActivityRecord
next) {
    next.idle=false ;
    next.results=null ;
    next.newIntents=null ;
    //发送一个超时处理消息， 默认为10秒。 IDLE_TIMEOUT_MSG 就是针对
activityIdle函数的
    Message msg=mHandler.obtainMessage (IDLE_TIMEOUT_MSG) ;
    msg.obj=next ;
    mHandler.sendMessageDelayed (msg, IDLE_TIMEOUT) ;
    //通知AMS
    if ( mMainStack ) mService.reportResumedActivityLocked
(next) ;
    .....//略去其他逻辑的代码
}
```

由以上代码可知， AMS给了应用进程10秒的时间， 希望它在10秒内调用activityIdle函数。这个时间不算长， 和前面AMS等待应用进程启动的超时时间一样。所以， 笔者有些困惑， 为什么要把这么重要的操作放到Idler中去做。

下面来看activityIdle函数， 在其内部将调用ActivityStack activityIdleInternal。

(3) activityIdleInternal函数分析

这部分的代码具体如下：

[-->ActivityStack.java : activityIdleInternal]

```
final ActivityRecord activityIdleInternal ( IBinder token,
boolean fromTimeout,
Configuration config) {
/*
如果应用进程在超时时间内调用了activityIdleInternal函数，则
fromTimeout为false，  

否则，一旦超时，在IDLE_TIMEOUT_MSG的消息处理中也会调用该函数，并设置
fromTimeout  

为true
*/
ActivityRecord res=null;
ArrayList<ActivityRecord>stops=null;
ArrayList<ActivityRecord>finishes=null;
ArrayList<ActivityRecord>thumbnails=null;
int NS=0;
int NF=0;
int NT=0;
IApplicationThread sendThumbnail=null;
boolean booting=false;
boolean enableScreen=false;
synchronized (mService) {
//从消息队列中撤销IDLE_TIMEOUT_MSG
if (token !=null) mHandler.removeMessages (IDLE_TIMEOUT_MSG,
token) ;
int index=indexOfTokenLocked (token) ;
if (index>=0) {
ActivityRecord r=mHistory.get (index) ;
res=r;
//注意，只有fromTimeout为true，才会执行下面的条件语句
if (fromTimeout) reportActivityLaunchedLocked (fromTimeout,
r, -1, -1) ;
if (config !=null) r.configuration=config;
/*
```

`mLaunchingActivity`是一个WakeLock，它能防止在操作Activity过程中掉电，同时

这个WakeLock不能长时间使用，否则有可能耗费过多电量。所以，系统设置了一个超时

处理消息`LAUNCH_TIMEOUT_MSG`，超时时间为10秒。一旦目标Activity启动成功，

就需要释放WakeLock

`*/`

```
if (mResumedActivity==r & & mLaunchingActivity.isHeld () ) {
```

```
    mHandler.removeMessages (LAUNCH_TIMEOUT_MSG) ;
```

```
    mLaunchingActivity.release () ;
```

```
}
```

```
    r.idle=true ;
```

```
    mService.scheduleAppGcsLocked () ;
```

```
.....
```

```
    ensureActivitiesVisibleLocked (null, 0) ;
```

```
    if (mMainStack) {
```

```
        if (!mService.mBooted) {
```

```
            mService.mBooted=true ;
```

```
            enableScreen=true ;
```

```
}
```

```
} //if (mMainStack) 判断结束
```

`else if (fromTimeout) {`//注意，只有`fromTimeout`为true，才会执行下面的case

```
    reportActivityLaunchedLocked (fromTimeout, null, -1, -1) ;
```

```
}
```

```
/*
```

①`processStoppingActivitiesLocked`函数返回那些因本次Activity启动而

被暂停 (paused) 的Activity

```
*/
```

```
stops=processStoppingActivitiesLocked (true) ;
```

```
.....
```

```
for (i=0 ; i<NS ; i++) {
```

```
    ActivityRecord r= (ActivityRecord) stops.get (i) ;
```

```
    synchronized (mService) {
```

```
//如果这些Activity处于finishing状态，则通知它们执行Destroy操作，  
最终它们  
    //的onDestroy函数会被调用  
    if ( r.finishing ) finishCurrentActivityLocked ( r,  
FINISH_IMMEDIATELY ) ;  
    else//否则将通知它们执行stop操作，最终Activity的onStop被调用  
        stopActivityLocked ( r ) ;  
    }//synchronized结束  
    }//for循环结束  
.....//处理等待结束的Activities  
//发送ACTION_BOOT_COMPLETED广播  
if (booting) mService.finishBooting () ;  
.....  
return res ;  
}
```

在activityIdleInternal中有一个关键点，即处理那些因为本次Activity启动而被暂停的Activity。有两种情况需考虑：

如果被暂停的Activity处于finishing状态（如Activity在其onPause中调用了finish函数），则调用finishCurrentActivityLocked。

否则，要调用stopActivityLocked处理暂停的Activity。

此处涉及除AMS和目标进程外的第三个进程，即被切换到后台的那个进程。不过至此，我们的目标Activity终于正式登上了历史舞台。

提示 本例的分析结束了吗？没有。因为am设置了-w选项，所以其实我们还在startActivity-AndWait函数中等待结果。ActivityStack中有两个函数能够触发AMS notifyAll，一个 是 reportActivityLaunchedLocked，另一个是 reportActivityVisibleLocked。前面介绍的 activityInternal函数只在fromTimeout为true时才会调用 reportActivityLaunchedLocked，但本例中 fromTimeout为false，这该如何是好？该问题的解答非常复杂，简单点说就是：当Activity显示出来时，其在AMS中对应ActivityRecord对象的 windowVisible函数将被调用，其内部会触发 reportActivityLaunched-Locked函数，这样我们的 startActivityAndWait才能被唤醒。

7.startActivity分析之后半程总结

总结startActivity后半部分的流程，主要涉及目标进程和AMS的交互，如图6-15所示。

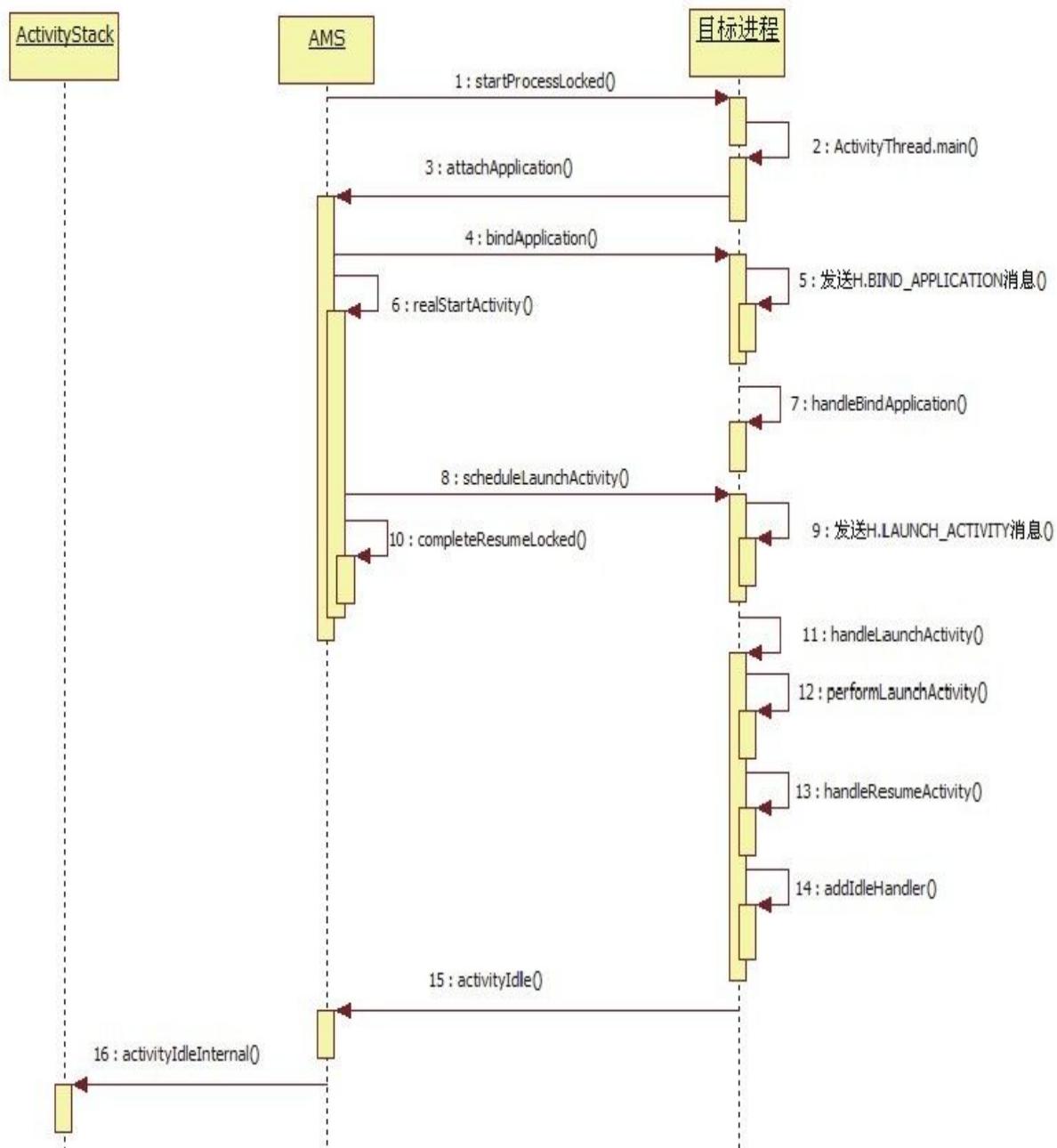


图 6-15 startActivity后半程总结

图6-15中涉及16个重要函数调用，而且这仅是startActivity后半部分的调用流程，可见整个流程有多么复杂！

8.startPausingLocked函数分析

现在我们分析图6-14中的startPausingLocked分支。根据前面的介绍，当启动一个新Activity时，系统将先行处理当前的Activity，即调用startPausingLocked函数来暂停当前Activity。

(1) startPausingLocked分析

该部分的代码如下：

[-->ActivityStack.java : startPausingLocked]

```
private final void startPausingLocked (boolean userLeaving,  
boolean uiSleeping) {  
    //mResumedActivity保存当前正显示的Activity  
    ActivityRecord prev=mResumedActivity ;  
    mResumedActivity=null ;  
    //设置mPausingActivity为当前Activity  
    mPausingActivity=prev ;  
    mLastPausedActivity=prev ;  
    prev.state=ActivityState.PAUSING ;//设置状态为PAUSING  
    prev.task.touchActiveTime () ;  
    .....  
    if (prev.app !=null & & prev.app.thread !=null) {  
        try{  
            //①调用当前Activity所在进程的schedulePauseActivity函数  
            prev.app.thread.schedulePauseActivity (           prev,  
            prev.finishing,  
            userLeaving, prev.configChangeFlags) ;  
            if (mMainStack) mService.updateUsageStats (prev, false) ;  
        }.....//catch分支  
    }.....//else分支
```

```
if (!mService.mSleeping && !mService.mShuttingDown) {  
    //获取WakeLock，以防止在Activity切换过程中掉电  
    mLaunchingActivity.acquire();  
    if (!mHandler.hasMessages(LAUNCH_TIMEOUT_MSG)) {  
        Message msg=mHandler.obtainMessage(LAUNCH_TIMEOUT_MSG);  
        mHandler.sendMessageDelayed(msg, LAUNCH_TIMEOUT);  
    }  
}  
  
if (mPausingActivity != null) {  
    //暂停输入事件派发  
    if (!uiSleeping) prev.pauseKeyDispatchingLocked();  
    //设置PAUSE超时，时间为500毫秒，这个时间相对较短  
    Message msg=mHandler.obtainMessage(PAUSE_TIMEOUT_MSG);  
    msg.obj=prev;  
    mHandler.sendMessageDelayed(msg, PAUSE_TIMEOUT);  
}.....//else分支  
}
```

startPausingLocked 将调用应用进程的 schedulePauseActivity 函数，并设置500毫秒的超时时间，所以应用进程需尽快完成相关处理。和 scheduleLaunchActivity 一样， schedulePauseActivity 将向 ActivityThread 主线程发送 PAUSE_ACTIVITY 消息，最终该消息由 handlePauseActivity 来处理。

(2) handlePauseActivity 分析

这部分的代码如下：

[-->ActivityThread.java : handlePauseActivity]

```
private void handlePauseActivity ( IBinder token, boolean finished,
    boolean userLeaving, int configChanges) {
    //当Activity处于finishing状态时, finished参数为true, 不过在本例中
    //该值为false
    ActivityClientRecord r=mActivities.get (token) ;
    if (r !=null) {
        //调用Activity的onUserLeaving函数,
        if (userLeaving) performUserLeavingActivity (r) ;
        r.activity.mConfigChangeFlags|=configChanges;
        //调用Activity的onPause函数
        performPauseActivity ( token,  finished,  r.isPreHoneycomb
() ) ;
        .....
        try{
            //调用AMS的activityPaused函数
            ActivityManagerNative.getDefault () .activityPaused
(token) ;
        }.....
    }
}
```

[-->ActivityManagerService.java :
activityPaused]

```
public final void activityPaused (IBinder token) {
    .....
    mMainStack.activityPaused (token, false) ;
}
```

[-->ActivityStack.java : activityPaused]

```
final void activityPaused (IBinder token, boolean timeout) {
```

```
ActivityRecord r=null;
synchronized (mService) {
    int index=indexOfTokenLocked (token) ;
    if (index>=0) {
        r=mHistory.get (index) ;
        //从消息队列中撤销PAUSE_TIMEOUT_MSG消息
        mHandler.removeMessages (PAUSE_TIMEOUT_MSG, r) ;
        if (mPausingActivity==r) {
            r.state=ActivityState.PAUSED ;//设置ActivityRecord的状态
            completePauseLocked () ;//完成本次Pause操作
        }.....}
    }
}
```

(3) completePauseLocked分析

这部分的代码具体如下：

[-->ActivityStack.java :
completePauseLocked]

```
private final void completePauseLocked () {
    ActivityRecord prev=mPausingActivity ;
    if (prev !=null) {
        if (prev.finishing) {
            prev=finishCurrentActivityLocked (prev,
                FINISH_AFTER_VISIBLE) ;
        }else if (prev.app !=null) {
            if (prev.configDestroy) {
                destroyActivityLocked (prev, true, false) ;
            }else{
                //①将刚才被暂停的Activity保存到mStoppingActivities中
                mStoppingActivities.add (prev) ;
                if (mStoppingActivities.size () >3) {
```

```
//如果被暂停的Activity超过3个，则发送IDLE_NOW_MSG消息，该消息最终
//由我们前面介绍的activeIdleInternal处理
scheduleIdleLocked () ;
}
}

//设置mPausingActivity为null，这是图6-14中②、③分支的分割点
mPausingActivity=null;
}

//②resumeTopActivityLocked将启动目标Activity
if ( ! mService.isSleeping ( ) ) resumeTopActivityLocked
(prev) ;
.....
}
```

就本例而言，以上代码还算简单，最后还是通过 resumeTopActivityLocked 来启动目标 Activity。当然，由于之前已经设置了 mPausingActivity 为 null，所以最终会走到图6-14中③的分支。

(4) stopActivityLocked分析

根据前面的介绍，此次目标 Activity 将走完 onCreate、onStart 和 onResume 流程，但是被暂停的 Activity 才刚走完 onPause 流程，那么它的 onStop 什么时候调用呢？答案就在 activityIdleInternal 中，它将为 mStoppingActivities 中的成员调用 stopActivityLocked 函数。

[--> ActivityStack.java : stopActivityLocked]

```
private final void stopActivityLocked (ActivityRecord r) {  
    if ((r.intent.getFlags () &  
Intent.FLAG_ACTIVITY_NO_HISTORY) !=0  
    || (r.info.flags&ActivityInfo.FLAG_NO_HISTORY) !=0) {  
        if (!r.finishing) {  
            requestFinishActivityLocked (r, Activity.RESULT_CANCELED,  
null,  
                "no-history") ;  
        }  
    }else if (r.app !=null&&r.app.thread !=null) {  
        try{  
            r.stopped=false ;  
            //设置STOPPING状态，并调用对应的scheduleStopActivity函数  
            r.state=ActivityState.STOPPING ;  
            r.app.thread.scheduleStopActivity (r, r.visible,  
                r.configChangeFlags) ;  
        }.....  
    }  
}
```

对应进程的scheduleStopActivity函数将根据visible 的情况，向主线程消息循环发送H.STOP_ACTIVITY_HIDE 或 H.STOP_ACTIVITY_SHOW 消息。不论哪种情况，最终都由handleStopActivity来处理。

[-->ActivityThread.java : handleStopActivity]

```
private void handleStopActivity (IBinder token, boolean  
show, int configChanges) {  
    ActivityClientRecord r=mActivities.get (token) ;  
    r.activity.mConfigChangeFlags|=configChanges ;  
    StopInfo info=new StopInfo () ;  
    //调用Activity的onStop函数
```

```
performStopActivityInner (r, info, show, true) ;  
.....  
try{//调用AMS的activityStopped函数  
ActivityManagerNative.getDefault () .activityStopped (  
r.token, r.state, info.thumbnail, info.description) ;  
}  
}
```

虽然AMS没有为stop设置超时消息处理，但是严格来说，还是有超时限制的，只是这个超时处理与activityIdleInternal结合起来了。

(5) startPausingLocked总结

startPausingLocked的流程如图6-16所示。

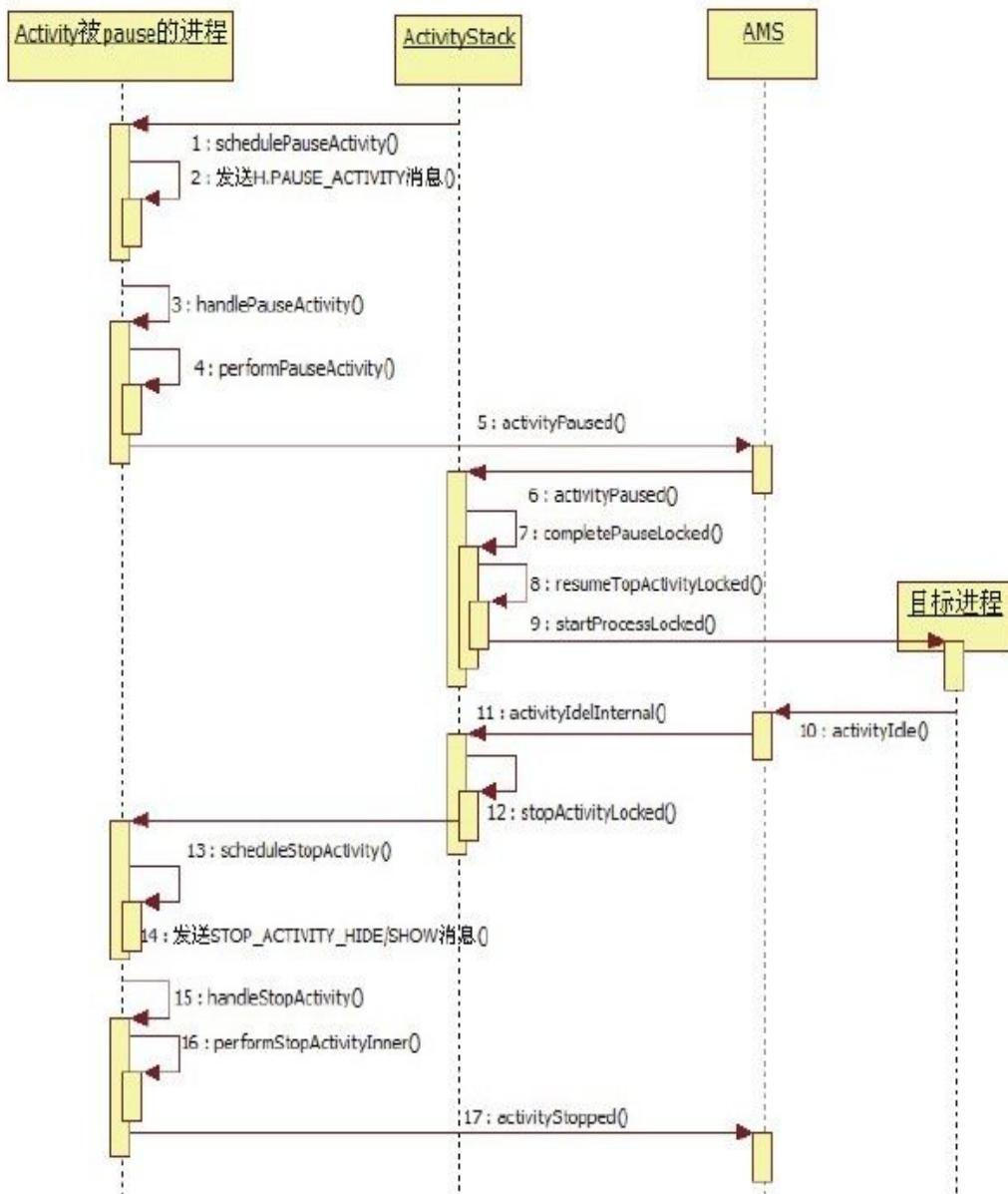


图 6-16 startPausingActivity流程总结

图6-16比较简单，读者最好结合代码再把流程走一遍，以加深理解。

9.startActivity总结

Activity的启动就介绍到这里。这一路分析下来，相信读者也和笔者一样觉得此行绝不轻松。先回顾一下此次旅程：

行程的起点是am。am是Android中很重要的程序，读者务必要掌握它的用法。我们利用am start命令，发起本次目标Activity的启动请求。

接下来进入ActivityManagerService 和 ActivityStack这两个核心类。对于启动Activity来说，这段行程又可分细分为两个阶段：第一阶段的主要工作就是根据启动模式和启动标志找到或创建ActivityRecord及对应的TaskRecord；第二阶段工作就是处理Activity启动或切换相关的工作。

然后讨论了AMS直接创建目标进程并运行Activity的流程，其中涉及目标进程的创建，目标进程中Android运行环境的初始化，目标Activity的创建以及onCreate、onStart及onResume等其生命周期中重要函数的调用等相关知识点。

接着又讨论了AMS先暂停当前Activity，然后在创建目标进程并运行Activity的流程。其中牵扯到两个应用进程和AMS的交互，其难度之大可见一斑。

读者在阅读本节时，务必要区分此旅程中两个阶段工作的重点：其一是找到合适的ActivityRecord和TaskRecord；其二是调度相关进程进行Activity切换。在SDK文档中，介绍最为详细的是第一阶段中系统的处理策略，例如启动模式、启动标志的作用等。第二阶段工作其实是与Android组件调度相关的工作。SDK文档只是针对单个Activity进行生命周期方面的介绍。

坦诚地说，这次旅程略过不少逻辑情况。原因有二，一方面受限于精力和篇幅；另一方面是作为调度核心类，和AMS相关的代码及处理逻辑非常复杂，而且其间还夹杂了与WMS的交互逻辑，使复杂度更甚。再者，笔者个人感觉这部分代码肯定谈不上高效、严谨和美观，甚至有些丑陋（在分析它们的过程中，远没有研究Audio、Surface时那种畅快淋漓的感觉）。

此处列出几个供读者深入研究的点：

各种启动模式、启动标志的处理流程。

Configuration发生变化时Activity的处理，以及在Activity中对状态保存及恢复的处理流程。

Activity生命周期各个阶段的转换及相关处理。Android 2.3以后新增的与Fragment的生命周期

相关的转换及处理。

建议 在研究代码前，先仔细阅读SDK文档相关内容，以获取必要的感性认识，否则直接看代码很容易迷失方向。

[1] 关于Zygote的工作原理，请读者阅读卷I第4章“深入理解Zygote”。

6.4 Broadcast和BroadcastReceiver分析

Broadcast，中文意思为“广播”。它是Android平台中的一种通知机制。从广义角度来说，它是一种进程间通信的手段。有广播，就对应有广播接收者。Android中四大组件之一的BroadcastReceiver即代表广播接收者。目前，系统提供两种方式来声明一个广播接收者。

在AndroidManifest.xml中声明<receiver>标签。在应用程序运行时，系统会利用Java反射机制构造一个广播接收者实例。本书将这种广播接收者称为静态注册者或静态接收者。

在应用程序运行过程中，可调用Context提供的registerReceiver函数注册一个广播接收者实例。本书将这种广播接收者称为动态注册者或动态接收者。与之相对应，当应用程序不再需要监听广播时（例如当应用程序退到后台时），则要调用 unregisterReceiver 函数撤销之前注册的 BroadcastReceiver 实例。

当系统将广播派发给对应的广播接收者时，广播接收者的onReceive函数会被调用。在此函数中，可对该广播进行相应处理。

另外，Android定义了3种不同类型的广播发送方式，它们分别是：

普通广播发送方式，由sendBroadcast及相关函数发送。以工作中的场景为例，当程序员们正埋头工作之时，如果有人大喊一声“吃午饭去”，前刻还在专心编码的人会马上站起来走向餐厅。这种方式即为普通广播发送方式，所有对“吃午饭”感兴趣的接收者都会响应。

串行广播发送方式，即ordered广播，由sendOrdedBroadcast及相关函数发送。在该类型方式下，按接收者的优先级将广播一个一个地派发给接收者。只有等上一个接收者处理完毕后，系统才将该广播派发给下一个接收者。其中，任意一个接收者都可以中止后续的派发流程。还是以工作中的场景为例：经常有项目经理（PM）深夜组织一帮人跟踪bug的状态。PM看见一个bug，问某程序员，“这个bug你能改吗？”如果得到的答案是“暂时不会”或“暂时没时间”，他会将目光转向下一个神情轻松者，直到找到一个担当者为止。这种方式即为ordered广播发送方式，很明显，它的特点是“一个一个来”。

Sticky广播发送方式，由sendStickyBroadcast及相关函数发送。Sticky的意思是“粘”，其背后有

一个很重要的考虑。我们举个例子：假设某广播发送者每5秒发送一次携带自己状态信息的广播，此时某个应用进程注册了一个动态接收者来监听该广播，那么该接收者的OnReceive函数何时被调用呢？在正常情况下需要等这一轮的5秒周期结束后才调用（因为发送者在本周期结束后会主动再发一个广播）。而在Sticky模式下，系统将马上派发该广播给刚注册的接收者。注意，这个广播是系统发送的，其中存储的是上一次广播发送者的状态信息。也就是说，在Sticky模式下，广播接收者能立即得到一个广播，而不用等到这一周期结束。其实就是系统会保存Sticky的广播^[1]，当有新广播接收者来注册时，系统就把Sticky广播发给它。

以上我们对广播及广播接收者做了一些简单介绍，读者也可参考SDK文档中的相关说明来加强理解。

下面将以动态广播接收者为例，分析Android对广播的处理流程。

6.4.1 registerReceiver流程分析

1.ContextImpl registerReceiver分析

registerReceiver函数用于注册一个动态广播接收者，该函数在Context.java中声明。根据本章前面对Context家族的介绍（参考图6-3）可知，其功能最终将通过ContextImpl类的registerReceiver函数来完成。此处，可直接去看ContextImpl是如何实现此函数的。在SDK中一共定义了两个同名的registerReceiver函数，其代码如下：

[-->ContextImpl.java : registerReceiver]

```
/*
```

在 SDK 中输出该函数，这也是最常用的函数。当广播到来时，
BroadcastReceiver对象的onReceive

函数将在主线程中被调用

```
*/
```

```
public Intent registerReceiver (BroadcastReceiver receiver,  
IntentFilter filter) {
```

```
    return registerReceiver (receiver, filter, null, null) ;
```

```
}
```

```
/*
```

功能和前面类似，但增加了两个参数，分别是broadcastPermission和
scheduler，作用有

两个：

其一：对广播者的权限增加了限制，只有拥有相应权限的广播者发出的广播才能被
此接收者接收

其二：BroadcastReceiver对象的onReceive函数可调度到scheduler所在
的线程中执行

```
*/
```

```
public Intent registerReceiver (BroadcastReceiver receiver,  
IntentFilter filter, String broadcastPermission, Handler  
scheduler) {  
    /*
```

注意，下面所调用函数的最后一个参数为getOuterContext的返回值。前面曾说过，ContextImpl为

Context家族中真正干活的对象，而它对外的代理人可以是Application和Activity等，

getOuterContext就返回这个对外代理人。一般在Activity中调用registerReceiver函数，故此处

getOuterContext返回的对外代理人的类型就是Activity。

*/

```
return registerReceiverInternal ( receiver, filter,
broadcastPermission,
scheduler, getOuterContext () ) ;
}
```

殊途同归，最终的功能由registerReceiverInternal来完成，其代码如下：

[-->ContextImpl.java
registerReceiverInternal]

```
private Intent registerReceiverInternal ( BroadcastReceiver
receiver,
IntentFilter filter, String broadcastPermission, Handler
scheduler,
Context context) {
IIntentReceiver rd=null;
if (receiver !=null) {
//①准备一个IIntentReceiver对象
if (mPackageInfo !=null&&context !=null) {
//如果没有设置scheduler，则默认使用主线程的Handler
if (scheduler==null) scheduler=mMainThread.getHandler () ;
//通过getReceiverDispatcher函数得到一个IIntentReceiver类型的对象
rd=mPackageInfo.getReceiverDispatcher (
receiver, context, scheduler, mMainThread.getInstrumentation
() ,
```

```
    true) ;
}else{
    if (scheduler==null) scheduler=mMainThread.getHandler () ;
    //直接创建LoadedApk.ReceiverDispatcher对象
    rd=new LoadedApk.ReceiverDispatcher ( receiver, context,
scheduler,
    null, true) .getIIntentReceiver () ;
} //if (mPackageInfo !=null&&context !=null) 结束
} //if (receiver !=null) 结束
try{
    //②调用AMS的registerReceiver函数
    return ActivityManagerNative.getDefault () .registerReceiver
(
    mMainThread.getApplicationThread () , mBasePackageName,
    rd, filter, broadcastPermission) ;
}.....
}
```

以上代码列出了两个关键点：其一是准备一个IIntentReceiver对象；其二是调用AMS的registerReceiver函数。

先来看IIntentReceiver，它是一个Interface，图6-17列出了和它相关的成员图谱。

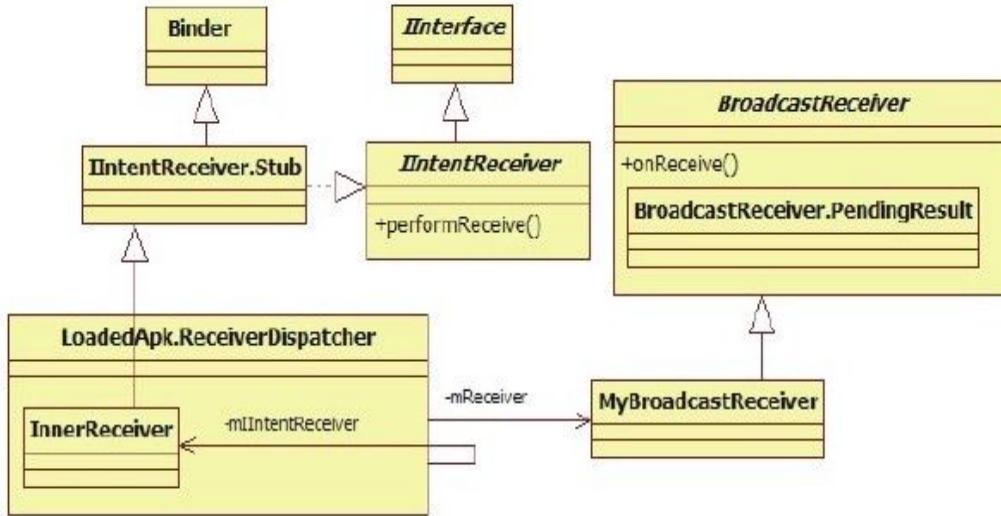


图 6-17 IIntentReceiver相关成员示意图

由图6-17可知：

`BroadcastReceiver` 内部有一个 `PendingResult` 类。该类是Android 2.3以后新增的，用于异步处理广播消息。例如，当`BroadcastReceiver`收到一个广播时，其 `onReceive` 函数将被调用。一般都是在该函数中直接处理该广播。不过，当该广播处理比较耗时，还可采用异步的方式进行处理，即先调用 `BroadcastReceiver` 的 `goAsync` 函数得到一个 `PendingResult` 对象，然后将该对象放到工作线程中去处理，这样 `onReceive` 函数就可以立即返回而不至于耽误太长时间（这一点对于 `onReceive` 函数被主线程调用的情况尤为有用）。在工作线程处理完这条广播后，需调用 `PendingResult` 的 `finish` 函数来完成整个广播的处理流程。

广播由AMS发出，而接收及处理工作却在另外一个进程中进行，整个过程一定涉及进程间通信。在图6-17中，虽然在BroadcastReceiver中定义了一个广播接收者，但是它与Binder没有有任何关系，故其并不直接参与进程间通信。与之相反，IIntentReceiver接口则和Binder有密切关系，故可推测广播的接收是由IIntentReceiver接口来完成的。确实，在整个流程中，首先接收到来自AMS的广播的将是该接口的Bn端，即LoadedApk.ReceiverDispatcher的内部类InnerReceiver。

接收广播的处理将放到本节最后再来分析，下面先来看AMS的registerReceiver函数。

2.AMS的registerReceiver分析

AMS的registerReceiver函数比较简单，但是由于其中将出现一些新的变量类型和成员，因此接下来按分两部分进行分析。

(1) registerReceiver分析之一

registerReceiver的返回值是一个Intent，它指向一个匹配过滤条件（由filter参数指明）的Sticky Intent。即使有多个符合条件的Intent，也只返回一个。

[-->ActivityManagerService.java
registerReceiver]

:

```
public Intent registerReceiver ( IApplicationThread caller,
String callerPackage,
    IIntentReceiver receiver, IntentFilter filter, String
permission) {
    synchronized (this) {
        ProcessRecord callerApp=null;
        if (caller !=null) {
            callerApp=getRecordForAppLocked (caller) ;
            .....//如果callerApp为null，则抛出异常，即系统不允许未登记照册的进程注
册
            //动态广播接收者
            //检查调用进程是否有callerPackage的信息，如果没有，也抛异常
            if (callerApp.info.uid !=Process.SYSTEM_UID & &
                !callerApp.pkgList.contains (callerPackage) ) {
                throw new SecurityException (.....) ;
            }
            .....//if (caller !=null) 判断结束
            List allSticky=null;
            //下面这段代码的功能是从系统中所有 Sticky Intent 中查询匹配
IntentFilter的Intent，
            //匹配的Intent保存在allSticky中
            Iterator actions=filter.actionsIterator () ;
            if (actions !=null) {
                while (actions.hasNext () ) {
                    String action= (String) actions.next () ;
                    allSticky=getStickiesLocked (action, filter, allSticky) ;
                }
            }.....
            //如果存在sticky的Intent，则选取第一个Intent作为本函数的返回值
            Intent sticky=allSticky !=null? (Intent) allSticky.get (0) :
null;
            //如果没有设置接收者，则直接返回sticky的Intent
```

```
if (receiver==null) return sticky;
//新的数据类型ReceiverList及成员变量mRegisteredReceivers，见下文的解释
//receiver.asBinder将返回IIntentReceiver的Bp端
ReceiverList rl
= (ReceiverList) mRegisteredReceivers.get (receiver.asBinder()
() ) ;
//如果是首次调用，则此处rl的值将为null
if (rl==null) {
rl=new ReceiverList (this, callerApp, Binder.getCallingPid
() ,
Binder.getCallingUid () , receiver) ;
if (rl.app !=null) {
rl.app.receivers.add (rl) ;
}else{
try{
//监听广播接收者所在进程的死亡消息
receiver.asBinder () .linkToDeath (rl, 0) ;
}.....
rl.linkedToDeath=true ;
}//if (rl.app !=null) 判断结束
//将rl保存到mRegisterReceivers中
mRegisteredReceivers.put (receiver.asBinder () , rl) ;
}
//新建一个BroadcastFilter对象
BroadcastFilter bf=new BroadcastFilter ( filter, rl,
callerPackage,
permission) ;
rl.add (bf) ;//将其保存到rl中
//mReceiverResolver成员变量，见下文解释
mReceiverResolver.addFilter (bf) ；
```

以上代码的流程倒是很简单，不过其中出现的几个成员变量和数据类型却严重阻碍了我们的

思维活动。先解决它们，BroadcastFilter及相关成员变量如图6-18所示。

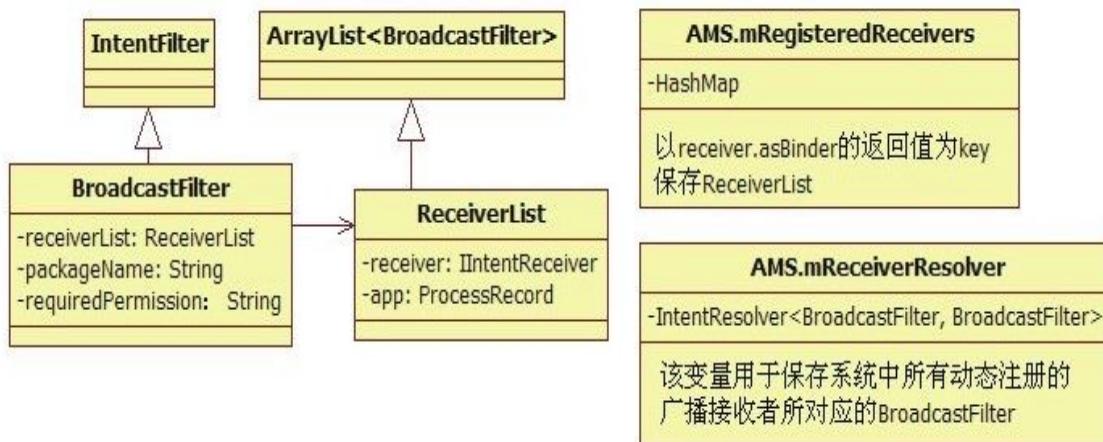


图 6-18 BroadcastFilter及相关成员变量

结合代码，对图6-18中各数据类型和成员变量的作用及关系的解释如下：

在AMS中，BroadcastReceiver的过滤条件由BroadcastFilter表示，该类从Intent-Filter派生。由于一个BroadcastReceiver可设置多个过滤条件（即多次为同一个Broadcast-Receiver对象调用registerReceiver函数以设置不同的过滤条件），故AMS使用ReceiverList（从ArrayList<BroadcastFilter>派生）这种数据类型来表达这种一对多的关系。

ReceiverList除了能存储多个BroadcastFilter外，还应该有成员指向某一个具体BroadcastReceiver。如果不这样，那么又是如何知

道到底是哪个BroadcastReceiver设置的过滤条件呢？前面说过，BroadcastReceiver接收广播是通过IIntentReceiver接口进行的，故ReceiverList中有receiver成员变量指向IIntentReceiver。

AMS 提供 mRegisterReceivers 用于保存 IIntentReceiver 和对应 ReceiverList 的关系。此外，AMS 还提供 mReceiverResolver 变量用于存储所有动态注册的 Broadcast-Receiver 所设置的过滤条件。

清楚这些成员变量和数据类型之间的关系后，接着来分析 registerReceiver 第二阶段的工作。

(2) registerReceiver 分析之二

这部分代码如下：

[--> ActivityManagerService.java :
registerReceiver]

```
//如果allSticky不为空，则表示有Sticky的Intent，需要立即调度广播发送
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) ;
        //为每一个需要发送的广播创建一个BroadcastRecord（暂称之为广播记录）
        对象
        BroadcastRecord r=new BroadcastRecord (intent, null,
```

```
null, -1, -1, null, receivers, null, 0, null, null,
false, true, true) ;
//如果mParallelBroadcasts当前没有成员，则需要触发AMS发送广播
if (mParallelBroadcasts.size () ==0)
scheduleBroadcastsLocked () ;//向AMS发送BROADCAST_INTENT_MSG
消息
//所有非ordered广播记录都保存在mParallelBroadcasts中
mParallelBroadcasts.add (r) ;
}//for循环结束
}//if (allSticky !=null) 判断结束
return sticky;
}//synchronized结束
}
```

这一阶段的工作用一句话就能说清楚：为每一个满足 IntentFilter 的 Sticky 的 Intent 创建一个 BroadcastRecord 对象，并将其保存到 mParllelBroadcasts 数组中，最后，根据情况调度 AMS 发送广播。

从上边的描述中可以看出，一旦存在满足条件的 Sticky 的 Intent，系统需要尽快调度广播发送。说到这里，想和读者分享一种笔者在实际工作中碰到的情况。

我们注册了一个 BroadcastReceiver，用于接收 USB 的连接状态。在注册完后，它的 onReceiver 函数很快就会被调用。当时笔者的一些同事认为是注册操作触发 USB 模块又发送了一次广播，却又感到有些困惑，USB 模块应该根据 USB 的状态变

化去触发广播发送，而不应理会广播接收者的注册操作，这到底是怎么一回事呢？

相信读者现在应该能轻松解决这种困惑了吧？对于Sticky的广播，一旦有接收者注册，系统会马上将该广播传递给它们。

和ProcessRecord及ActivityRecord类似，AMS定义了一个BroadcastRecord数据结构，用于存储和广播相关的信息，同时还有两个成员变量，它们作用和关系如图6-19所示。



图 6-19 BroadcastReceiver及相关变量

图6-19比较简单，读者可自行研究。

在代码中，`registerReceiver` 将调用 `scheduleBroadcastsLocked` 函数，通知AMS立即着手开展广播发送工作，在其内部就发送

BROADCAST_INTENT_MSG消息给AMS。相关的处理工作将放到后面再来讨论。

[1]读者可以想一想，由于系统内部保存了Stikcy广播，当一个恶意程序频繁发送这种广播时，是否会把内存都耗光呢？

6.4.2 sendBroadcast流程分析

在SDK中同样定义了几个函数用于发送广播。不过，根据之前的经验，最终和AMS交互的函数可能通过一个接口就能完成。下面来看最简单的广播发送函数sendBroadcast，其代码如下：

[-->ContextImpl.java : sendBroadcast]

```
public void sendBroadcast (Intent intent) {  
    String resolvedType=intent.resolveTypeIfNeeded  
(getContentResolver () ) ;  
    try{  
        intent.setAllowFds (false) ;  
        //调用AMS的brodcastIntent，在SDK中定义的广播发送函数最终都会调用它  
        ActivityManagerNative.getDefault () .broadcastIntent (  
            mMainThread.getApplicationThread () , intent, resolvedType,  
            null,  
            Activity.RESULT_OK, null, null, null, false, false) ;  
    }.....  
}
```

AMS的broadcastIntent函数的主要工作将交由AMS的broadcastIntentLocked来完成，故此处直接分析broadcastIntentLocked。

我们分阶段来分析broadcastIntentLocked的工作，先来看第一阶段工作。

1.broadcastIntentLocked分析之一

这部分的代码如下：

[-->ActivityManagerService.java :
broadcastIntentLocked]

```
private final int broadcastIntentLocked ( ProcessRecord
callerApp,
    String callerPackage, Intent intent, String resolvedType,
    IIntentReceiver resultTo, int resultCode, String
resultData,
    Bundle map, String requiredPermission,
    boolean ordered, boolean sticky, int callingPid, int
callingUid) {
    intent=new Intent (intent) ;
    //为Intent增加FLAG_EXCLUDE_STOPPED_PACKAGES标志，表示该广播不会
    //传递给被STOPPED
    //的Package
    intent.addFlags (Intent.FLAG_EXCLUDE_STOPPED_PACKAGES) ;
    // 处理一些特殊的广播，包括UID_REMOVED, PACKAGE_REMOVED 和
    PACKAGE_ADDED等
    final boolean uidRemoved=Intent.ACTION_UID_REMOVED.equals (
        intent.getAction () ) ;
    ....//处理特殊的广播，主要和PACKAGE相关，例如接收到PACKAGE_REMOVED广
    播后，AMS
    //需将该Package的组件从相关成员中删除，相关代码可自行阅读
    //处理TIME_ZONE变化广播
    if ( intent.ACTION_TIMEZONE_CHANGED.equals ( intent.getAction
() ) )
        mHandler.sendMessage (UPDATE_TIME_ZONE) ;
    //处理CLEAR_DNS_CACHE广播
    if ( intent.ACTION_CLEAR_DNS_CACHE.equals ( intent.getAction
() ) )
```

```
mHandler.sendMessage (CLEAR_DNS_CACHE) ;  
//处理PROXY_CHANGE广播  
if (Proxy.PROXY_CHANGE_ACTION.equals (intent.getAction ()) )  
{  
    ProxyProperties proxy=intent.getParcelableExtra ("proxy") ;  
    mHandler.sendMessage (mHandler.obtainMessage (  
        UPDATE_HTTP_PROXY, proxy) ) ;  
}
```

由以上代码可知，broadcastIntentLocked第一阶段的工作主要是处理一些特殊的广播消息。下面来看broadcastIntentLocked第二阶段的工作。

2.broadcastIntentLocked分析之二

这部分的代码如下：

[-->ActivityManagerService.java :
broadcastIntentLocked]

```
//处理发送sticky广播的情况  
if (sticky) {  
    ....//检查发送进程是否有BROADCAST_STICKY权限  
    if (requiredPermission !=null) {  
        //在发送Sticky广播的时候，不能携带权限信息  
        return BROADCAST_STICKY_CANT_HAVE_PERMISSION ;  
    }  
    //在发送Stikcy广播的时候，也不能指定特定的接收对象  
    if (intent.getComponent () !=null) ....//抛出异常  
    //将这个Sticky的Intent保存到mStickyBroadcasts中  
    ArrayList<Intent>list=mStickyBroadcasts.get  
(intent.getAction () ) ;  
    if (list==null) {
```

```
list=new ArrayList<Intent> () ;
mStickyBroadcasts.put (intent.getAction () , list) ;
}
.....//如果list中已经有该Intent，则直接用新的Intent替换旧的Intent
//否则将该Intent加入到list中保存
if (i>=N) list.add (new Intent (intent) ) ;
}
//定义两个变量，其中receivers用于保存匹配该Intent的所有广播接收者，包括静态或动态
//注册的广播接收者，registeredReceivers用于保存符合该Intent的所有动态注册者
List receivers=null;
List<BroadcastFilter>registeredReceivers=null;
try{
if (intent.getComponent () !=null) {
.....//如果指定了接收者，则从PKMS中查询它的信息
}else{
//FLAG_RECEIVER_REGISTERED_ONLY标签表明该广播只能发给动态注册者
if      (      (      intent.getFlags      (      )      &
Intent.FLAG_RECEIVER_REGISTERED_ONLY) ==0) {
//如果没有设置前面的标签，则需要查询PKMS，获得那些在
AndroidManifest.xml
//中声明的广播接收者，即静态广播接收者的信息，并保存到receivers中
receivers=
AppGlobals.getPackageManager () .queryIntentReceivers (
intent, resolvedType, STOCK_PM_FLAGS) ;
}
//再从AMS的mReceiverResolver中查询符合条件的动态广播接收者
registeredReceivers=mReceiverResolver.queryIntent (intent,
resolvedType, false) ;
}
}
}.....
/*
判断该广播是否设置了REPLACE_PENDING标签。如果设置了该标签，并且之前的
那个Intent还没
```

有被处理，则可以用新的Intent替换旧的Intent。这么做的目的是减少不必要的广播发送，但筆者感觉，这个标签的作用并不靠谱，因为只有旧的Intent没被处理的时候，它才能被替换。因为旧

Intent被处理的时间不能确定，所以不能保证广播发送者的一番“好意”能够实现。因此，在发送广播时，千万不要以为设置了该标志就一定能节约不必要的广播发送。

```
/*
final boolean replacePending=
(           intent.getFlags           (           )           &
Intent.FLAG_RECEIVER_REPLACE_PENDING) !=0 ;
//先处理动态注册的接收者
int NR=registeredReceivers !=null?registeredReceivers.size
():0 ;
//如果此次广播为非串行化发送，并且符合条件的动态注册接收者个数不为零
if (!ordered&&NR>0) {
//创建一个BroadcastRecord对象即可，注意，一个BroadcastRecord对象
可包括所有的
//接收者（可参考图6-19）
BroadcastRecord r=new BroadcastRecord (intent, callerApp,
callerPackage, callingPid, callingUid, requiredPermission,
registeredReceivers, resultTo, resultCode, resultData, map,
ordered, sticky, false) ;
boolean replaced=false ;
if (replacePending) {
.....//从mParallelBroadcasts中查找是否有旧的Intent，如果有就替代它，
并设置
//replaced为true
}
if (!replaced) {//如果没有被替换，则保存到mParallelBroadcasts中
mParallelBroadcasts.add (r) ;
scheduleBroadcastsLocked () ;//调度一次广播发送
}
//至此，动态注册的广播接收者已处理完毕，设置registeredReceivers为
null
registeredReceivers=null ;//
```

```
NR=0 ;  
}
```

broadcastIntentLocked 第二阶段的工作有两项：

查询满足条件的动态广播接收者及静态广播接收者。

当本次广播不为ordered时，需要尽快发送该广播。另外，非ordered的广播都被AMS保存在mParallelBroadcasts中。

3.broadcastIntentLocked分析之三

下面来看broadcastIntentLocked最后一阶段的工作，其代码如下：

[-->ActivityManagerService.java :
broadcastIntentLocked]

```
int ir=0 ;  
if (receivers !=null) {  
String skipPackages[] =null ;  
.....//处理PACKAGE_ADDED的Intent，系统不希望有些应用程序一安装就启动。  
//实现这一目的的工作原理即是在该程序内部监听PACKAGE_ADDED广播。如果系  
统  
//没有这一招防备，则PKMS安装完程序后所发送的PAKCAGE_ADDED消息将触发  
该应用的启动  
.....//处理ACTION_EXTERNAL_APPLICATIONS_AVAILABLE广播
```

```
.....//将动态注册的接收者registeredReceivers的元素合并到receivers中去
    //处理完毕后，所有的接收者（无论动态还是静态注册的）都存储到receivers
    //变量中了
    if ( (receivers !=null & & receivers.size () >0) ||resultTo !
    =null) {
        //创建一个BroadcastRecord对象，注意它的receivers中包括所有的接收者
        BroadcastRecord r=new BroadcastRecord (intent, callerApp,
        callerPackage, callingPid, callingUid, requiredPermission,
        receivers, resultTo, resultCode, resultData, map, ordered,
        sticky, false) ;
        boolean replaced=false ;
        if (replacePending) {
            .....//替换mOrderedBroadcasts中旧的Intent
        }//
        if ( ! replaced) { //如果没有替换，则添加该条广播记录到
        mOrderedBroadcasts中
            mOrderedBroadcasts.add (r) ;
            scheduleBroadcastsLocked () ; //调度AMS进行广播发送工作
        }
    }
    return BROADCAST_SUCCESS ;
}
```

由以上代码可知，AMS将动态注册者和静态注册者都合并到receivers中去了。注意，如果本次广播不是ordered，那么表明动态注册者就已经在第二阶段工作中被处理了。因此在合并时，将不会有动态注册者被加到receivers中。最终所创建的广播记录存储在mOrderedBroadcasts中，也就是说，不管是否串行化发送，静态接收者对应的广播记录都将保存在mOrderedBroadcasts中。

为什么不将它们保存在mParallelBroadcasts中呢？结合代码会发现，保存在mParallel-Broadcasts中的BroadcastRecord所包含的都是动态注册的广播接收者信息，这是因为动态接收者所在的进程是已经存在的（如果该进程不存在，如何去注册动态接收者呢？），而静态广播接收者就不能保证它已经和一个进程绑定在一起了（静态广播接收者此时可能还仅仅是在AndroidManifest.xml中声明的一个信息，相应的进程并未创建和启动）。根据前一节分析的Activity启动流程，AMS还需要进行创建应用进程，初始化Android运行环境等一系列复杂的操作。读到后面内容时你会发现，AMS将在一个循环中逐条处理mParallelBroadcasts中的成员（即派发给接收者）。如果将静态广播接收者也保存到mParallelBroadcasts中，会有什么后果呢？假设这些静态接收者所对应的进程全部未创建和启动，那么AMS将在那个循环中创建多个进程！这样，系统压力一下就上去了。所以，对于静态注册者，它们对应的广播记录都被放到mOrderedBroadcasts中保存。AMS在处理这类广播信息时，一个进程一个进程地处理，只有处理完一个接收者，才继续下一个接收者。这种做法的好处是，避免了惊群效应的出现，坏处则是延时较长。

下面进入 AMS 的
BROADCAST_INTENT_MSG 消息处理函数，看
看情况是否如上所说。

6.4.3 BROADCAST_INTENT_MSG消息处理函数

BROADCAST_INTENT_MSG 消息将触发 processNextBroadcast 函数，下面分阶段来分析该函数。

1. processNextBroadcast 分析之一

[--> ActivityManagerService.java :
processNextBroadcast]

```
private final void processNextBroadcast (boolean fromMsg) {  
    //如果是BROADCAST_INTENT_MSG消息触发该函数，则fromMsg为true  
    synchronized (this) {  
        BroadcastRecord r ;  
        updateCpuStats () ;//更新CPU使用情况  
        if (fromMsg) mBroadcastsScheduled=false ;  
        //先处理mParallelBroadcasts中的成员。如前所述，AMS在一个循环中处理  
        它们  
        while (mParallelBroadcasts.size () >0) {  
            r=mParallelBroadcasts.remove (0) ;  
            r.dispatchTime=SystemClock.uptimeMillis () ;  
            r.dispatchClockTime=System.currentTimeMillis () ;  
            final int N=r.receivers.size () ;  
            for (int i=0 ; i<N ; i++) {  
                Object target=r.receivers.get (i) ;  
                //①mParallelBroadcasts中的成员全为BroadcastFilter类型，所以下面  
                的函数  
                //将target直接转换成BroadcastFilter类型。注意，最后一个参数为false
```

```
    deliverToRegisteredReceiverLocked ( r , ( BroadcastFilter )
target,
    false) ;
}
//将这条处理过的记录保存到mHistoryBroadcast中，供调试使用
addBroadcastToHistoryLocked ( r ) ;
}
```

deliverToRegisteredReceiverLocked函数的功能就是派发广播给接收者，其代码如下：

[-->ActivityManagerService.java :
deliverToRegisteredReceiverLocked]

```
private final void deliverToRegisteredReceiverLocked
(BroadcastRecord r,
 BroadcastFilter filter, boolean ordered) {
boolean skip=false;
//检查发送进程是否有filter要求的权限
if (filter.requiredPermission !=null) {
int perm=checkComponentPermission
(filter.requiredPermission,
r.callingPid, r.callingUid, -1, true) ;
if (perm !=PackageManager.PERMISSION_GRANTED) skip=true ;
}
//检查接收者是否有发送者要求的权限
if (r.requiredPermission !=null) {
int perm=checkComponentPermission (r.requiredPermission,
filter.receiverList.pid, filter.receiverList.uid , -1 ,
true) ;
if (perm !=PackageManager.PERMISSION_GRANTED) skip=true ;
}
if (!skip) {
if (ordered) {
```

```
.....//设置一些状态、成员变量等信息，不涉及广播发送
}
try{
//发送广播
performReceiveLocked (filter.receiverList.app,
filter.receiverList.receiver, new Intent ( r.intent ) ,
r.resultCode,
r.resultData, r.resultExtras, r.ordered, r.initialSticky) ;
if (ordered) r.state=BroadcastRecord.CALL_DONE_RECEIVE ;
}.....
}
}
}
```

下面来看performReceiveLocked函数，其代码如下：

[-->ActivityManagerService.java
performReceiveLocked]

```
static void performReceiveLocked ( ProcessRecord app,
IIntentReceiver receiver,
Intent intent, int resultCode, String data, Bundle extras,
boolean ordered, boolean sticky) throws RemoteException{
if (app !=null&&app.thread !=null) {
// 如 果 app 及 app.thread 不 为 null ， 则 调 度
scheduleRegisteredReceiver,
//注意这个函数名为scheduleRegisteredReceiver，它只针对动态注册的广
播接收者
app.thread.scheduleRegisteredReceiver ( receiver, intent,
resultCode,
data, extras, ordered, sticky) ;
}else{
//否则调用IIntentReceiver的performReceive函数
```

```
    receiver.performReceive(intent, resultCode, data, extras,  
    ordered, sticky) ;  
}  
}
```

对于动态注册者而言，在大部分情况下会执行上述代码中的if分支，所以应用进程Application Thread的schedule Registered Receiver函数将被调用。稍后再分析应用进程的广播处理流程。

2.process Next Broadcast分析之二

至此，process Next Broadcast已经在[一个while循环中处理完mParallel Broadcasts的所有成员了](#)，实际上，这种处理方式也会造成惊群效应，但影响相对较少。这是因为对于动态注册者来说，它们所在的应用进程已经创建并初始化成功。此处的广播发送只是调用应用进程的一个函数而已。相比于创建进程，再初始化Android运行环境所需的工作量，调用schedule Registered Receiver的工作就比较轻松了。

下面来看process Next Broadcast第二阶段的工作，代码如下：

[-->Activity Manager Service.java : process
Next Broadcast]

```
/*
```

现在要处理mOrderedBroadcasts中的成员。如前所述，它要处理一个接一个地接收者，如果

接收者所在进程还未启动，则需要等待。mPending Broadcast变量用于标志因为应用进程还未

启动而处于等待状态的Broadcast Record。

```
*/
```

```
if (mPending Broadcast !=null) {  
    boolean isDead;  
    synchronized (mPidSelfLocked) {  
        isDead= (mPidSelfLocked.get (mPidPending  
Broadcast.curApp.pid) ==null) ;  
    }  
}
```

/*重要说明

判断要等待的进程是否为dead进程，如果没有dead进程，则继续等待。仔细思考，此处直接

返回会有什么问题。

问题不小！假设有两个ordered广播A和B，有两个接收者，AR和BR，并且BR所

在进程已经启动并完成初始化Android运行环境工作。如果process Next Broadcast先处理A，再处理B，那么此处B的处理将因为mPending Broadcast不为空而被延后。虽然B和A

之间没有任何关系（例如完全是两个不同的广播消息）。

但是事实上，大多数开发人员理解的order是针对单个广播的，例如A有5个接收者，那么对这

5个接收者的广播的处理是串行的。通过此处的代码发现，系统竟然串行处理A和B广播，即

B广播要待到A的5个接收者都处理完了才能处理

```
*/
```

```
if (!isDead) return ;  
else{  
    mPending Broadcast.state=Broadcast Record.IDLE ;  
    mPending Broadcast.next Receiver=mPending Broadcast  
RecvIndex ;  
    mPendingBroadcast=null ;  
}
```

```
boolean looped=false;
do{
    //mOrderedBroadcasts处理完毕
    if (mOrdered Broadcasts.size () ==0) {
        schedule App GcsLocked () ;
        if (looped) update OomAdj Locked () ;
        return ;
    }
    r=mOrdered Broadcasts.get (0) ;
    boolean force Receive=false;
    //下面这段代码用于判断此条广播是否处理时间过长
    //先得到该条广播的所有接收者
    int numReceivers= (r.receivers !=null) ?r.receivers.size () :
    0;
    if (mProcesses Ready && r.dispatchTime>0) {
        long now=System Clock.uptime Millis () ;
        //如果总耗时超过2倍的接收者个数*每个接收者最长处理时间（10秒），则
        //强制结束这条广播的处理
        if ( (numReceivers>0) &&
            (now>r.dispatchTime+
            (2*BROADCAST_TIMEOUT*numReceivers) ) ) {
            broadcastTimeoutLocked (false) ;//读者阅读完本节后可自行研究该函
            数
            forceReceive=true;
            r.state=Broadcast Record.IDLE;
        }
    } //if (mProcesses Ready.....) 判断结束
    if (r.state !=Broadcast Record.IDLE) return ;
    //如果下面这个if条件满足，则表示该条广播要么已经全部被处理，要么被中途
    取消
    if (r.receivers==null||r.nextReceiver>=numReceivers
        ||r.result Abort||force Receive) {
        if (r.resultTo !=null) {
            try{
                //将该广播的处理结果传给设置了resultTo的接收者
                performReceiveLocked (r.callerApp, r.resultTo,
```

```
new Intent (r.intent) , r.resultCode,
r.resultData, r.result Extras, false, false) ;
r.resultTo=null;
}.....
}
cancel Broadcast Timeout Locked () ;
addBroadcast To History Locked (r) ;
mOrdered Broadcasts.remove (0) ;
r=null;
looped=true ;
continue ;
}
}while (r==null) ;
```

processNextBroadcast第二阶段的工作比较简单：

首先根据是否处于 pending 状态 (mPendingBroadcast不为null) 进行相关操作。读者要认真体会代码中的重要说明。

处理超时的广播记录。这个超时时间是 $2*\text{BROADCAST_TIMEOUT}*\text{numReceivers}$ 。 BROADCAST_TIMEOUT默认为10秒。由于涉及创建进程，初始化Android运行环境等“重体力活”，故此处超时时间还乘以一个固定倍数2。

3.processNextBroadcast分析之三

下面来看processNextBroadcast第三阶段的工作，代码如下：

[-->ActivityManagerService.java
processNextBroadcast]

:

```
int recIdx=r.nextReceiver++ ;
r.receiverTime=SystemClock.uptimeMillis () ;
if (recIdx==0) {
    r.dispatchTime=r.receiverTime ; //记录本广播第一次处理开始的时间
    r.dispatchClockTime=System.currentTimeMillis () ;
}
//设置广播处理超时时间，BROADCAST_TIMEOUT为10秒
if (!mPendingBroadcastTimeoutMessage) {
    long timeoutTime=r.receiverTime+BROADCAST_TIMEOUT ;
    setBroadcastTimeoutLocked (timeoutTime) ;
}
//取该条广播下一个接收者
Object nextReceiver=r.receivers.get (recIdx) ;
if (nextReceiver instanceof BroadcastFilter) {
    //如果是动态接收者，则直接调用 deliver To Registered
    ReceiverLocked处理
    BroadcastFilter filter= (BroadcastFilter) next Receiver ;
    deliver To Registered Receiver Locked ( r, filter,
    r.ordered) ;
    if (r.receiver==null|| !r.ordered) {
        r.state=Broadcast Record.IDLE ;
        schedule Broadcasts Locked () ;
    }
    return ; //已经通知一个接收者去处理该广播，需要等它的处理结果，所以此处
    直接返回
}
//如果是静态接收者，则它的真实类型是ResolveInfo
ResolveInfo info= (ResolveInfo) nextReceiver ;
boolean skip=false ;
//检查权限
int perm=checkComponentPermission
(info.activityInfo.permission,
```

```
r.callingPid,                                r.callingUid,
info.activityInfo.applicationInfo.uid,
    info.activityInfo.exported) ;
if (perm != PackageManager.PERMISSION_GRANTED) skip=true;
if      (      info.activityInfo.applicationInfo.uid      !
=Process.SYSTEM_UID&&
r.requiredPermission!=null) {
.....
}
//设置skip为true
if (r.curApp !=null&&r.curApp.crashing) skip=true;
if (skip) {
r.receiver=null;
r.curFilter=null;
r.state=BroadcastRecord.IDLE;
scheduleBroadcastsLocked () ;//再调度一次广播处理
return ;
}
r.state=BroadcastRecord.APP_RECEIVE;
String targetProcess=info.activityInfo.packageName;
r.curComponent=new ComponentName (
info.activityInfo.applicationInfo.packageName,
info.activityInfo.name) ;
r.curReceiver=info.activityInfo;
try{
//设置Package stopped的状态为false
AppGlobals.getPackageManager () .setPackageStoppedState (
r.curComponent.getPackageName () , false) ;
}.....
ProcessRecord app=getProcessRecordLocked (targetProcess,
info.activityInfo.applicationInfo.uid) ;
//如果该接收者对应的进程已经存在
if (app !=null&&app.thread !=null) {
try{
app.addPackage (info.activityInfo.packageName) ;
//该函数内部调用应用进程的scheduleReceiver函数，读者可自行分析
```

```
processCurBroadcastLocked (r, app) ;  
return ; //已经触发该接收者处理本广播，需要等待处理结果  
}.....  
}  
//最糟的情况就是该进程还没有启动  
if ( (r.curApp=startProcessLocked (targetProcess,  
info.activityInfo.applicationInfo, true, .....) !=0) ) ==null) {  
.....//进程启动失败的处理  
return ;  
}  
mPendingBroadcast=r ; //设置mPendingBroadcast  
mPendingBroadcastRecvIndex=recIdx ;  
}
```

对processNextBroadcast第三阶段的工作总结如下：

如果广播接收者为动态注册对象，则直接调用deliverToRegisteredReceiverLocked处理它。

如果广播接收者为静态注册对象，并且该对象对应的进程已经存在，则调用processCurBroadcastLocked处理它。

如果广播接收者为静态注册对象，并且该对象对应的进程还不存在，则需要创建该进程。这是最糟糕的情况。

此处，不再讨论新进程创建及与Android运行环境初始化相关的逻辑，读者可返回阅

读“attachApplicationLocked分析之三”，其中有处理mPendingBroadcast的内容。

6.4.4 应用进程处理广播分析

下面来分析当应用进程收到广播后的处理流程，以动态接收者为例。

1.ApplicationThread
scheduleRegisteredReceiver函数分析

如前所述，AMS 将通过 scheduleRegisteredReceiver 函数将广播交给应用进程，该函数代码如下：

[-->ActivityThread.java :
scheduleRegisteredReceiver]

```
public void scheduleRegisteredReceiver ( IIntentReceiver
receiver, Intent intent,
int resultCode, String dataStr, Bundle extras, boolean
ordered,
boolean sticky) throws RemoteException{
//又把receiver对象传了回来。还记得注册时传递的一个IIntentReceiver类
型
//的对象吗？
receiver.performReceive ( intent, resultCode, dataStr,
extras, ordered,
sticky) ;
}
```

就本例而言，receiver对象的真实类型为LoadedApk.ReceiverDispatcher，来看它的performReceive函数，代码如下：

[-->LoadedApk.java : performReceive]

```
public void performReceive (Intent intent, int resultCode,
    String data, Bundle extras, boolean ordered, boolean
sticky) {
    //Args是一个Runnable对象
    Args args=new Args ( intent, resultCode, data, extras,
ordered, sticky) ;
    //mActivityThread是一个Handler，还记得SDK提供的两个同名的
registerReceiver
    //函数吗？如果没有传递Handler，则使用主线程的Handler
    if (!mActivityThread.post (args) ) {
        if (mRegistered&&ordered) {
            IActivityManager mgr=ActivityManagerNative.getDefault () ;
            args.sendFinished (mgr) ;
        }
    }
}
```

scheduleRegisteredReceiver最终向主线程的Handler投递了一个Args对象，这个对象的run函数将在主线程中被调用。

2.Args.run分析

这部分内容的代码如下：

[-->LoadedApk.java : Args.run]

```
public void run () {
    final BroadcastReceiver receiver=mReceiver ;
    final boolean ordered=mOrdered ;
    final IActivityManager mgr=ActivityManagerNative.getDefault
() ;
    final Intent intent=mCurIntent ;
    mCurIntent=null ;
    .....
    try{
        //获取ClassLoader对象，千万注意，此处并没有通过反射机制创建一个广播接
收者，
        //对于动态接收者来说，在注册前就已经创建完毕
        ClassLoader cl=mReceiver.getClass () .getClassLoader () ;
        intent.setExtrasClassLoader (cl) ;
        setExtrasClassLoader (cl) ;
        receiver.setPendingResult (this) ;//设置pendingResult
        //调用该动态接收者的onReceive函数
        receiver.onReceive (mContext, intent) ;
    }.....
    //调用finish完成工作
    if (receiver.getPendingResult () !=null) finish () ;
}
```

Finish的代码很简单，此处不再赘述，在其内部会通过 sendFinished 函数 调用 AMS 的 finishReceiver函数，以通知AMS。

3.AMS的finishReceiver函数分析

不论是ordered还是非ordered广播，AMS的 finishReceiver函数都会被调用，它的代码如下：

[-->ActivityManagerService.java
finishReceiver]

```
public void finishReceiver ( IBinder who, int resultCode,  
String resultData,  
Bundle resultExtras, boolean resultAbort) {  
    .....  
    boolean doNext ;  
    final long origId=Binder.clearCallingIdentity () ;  
    synchronized (this) {  
        //判断是否还需要继续调度后续的广播发送  
        doNext=finishReceiverLocked (  
            who, resultCode, resultData, resultExtras, resultAbort,  
            true) ;  
    }  
    if (doNext) { //发起下一次广播发送  
        processNextBroadcast (false) ;  
    }  
    trimApplications () ;  
    Binder.restoreCallingIdentity (origId) ;  
}
```

由以上代码可知，finishReceiver将根据情况
调度下一次广播发送。

6.4.5 广播处理总结

广播处理的流程及相关知识点还算比较简单，可以用图6-20来表示本例中广播的处理流程。

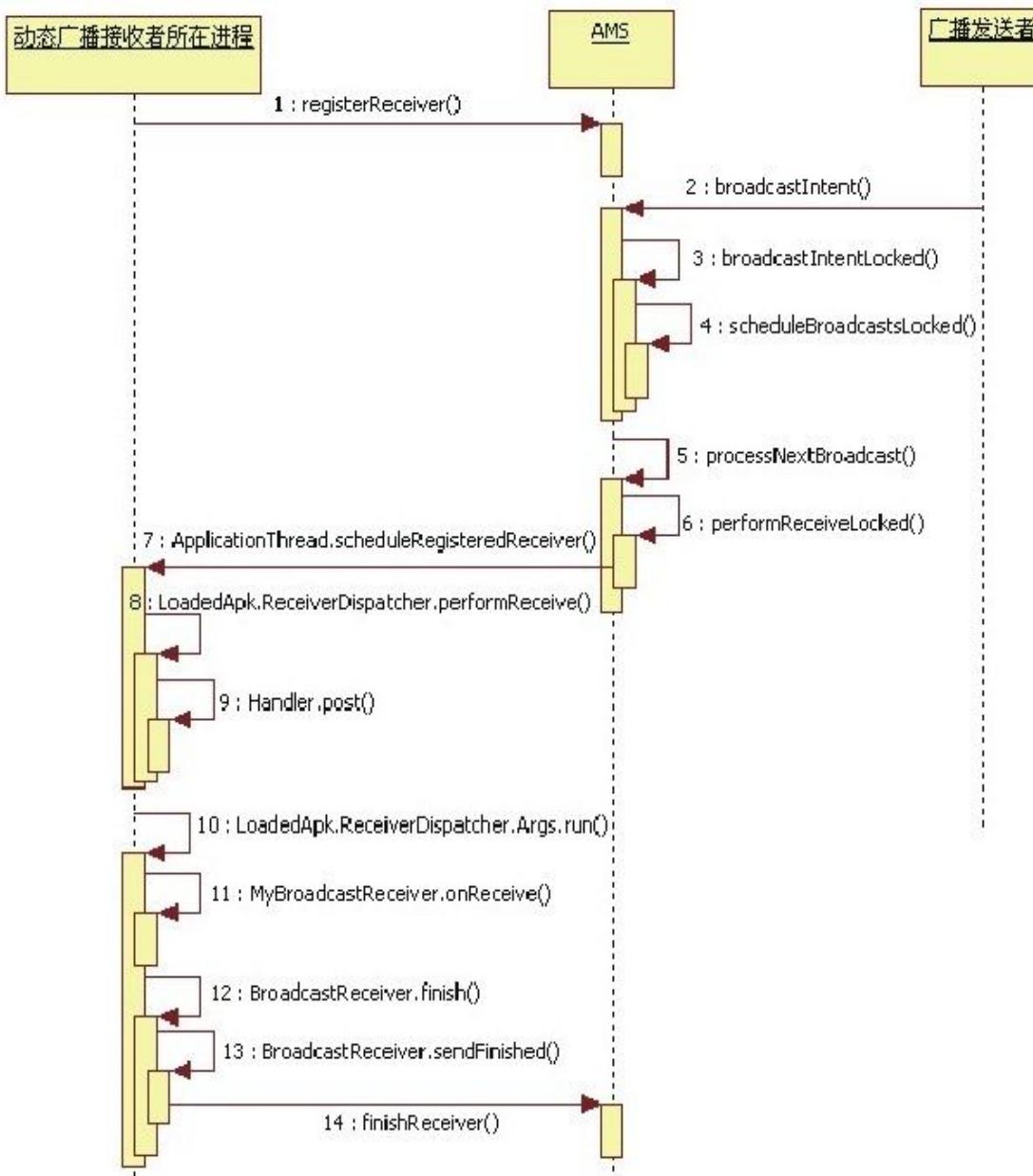


图 6-20 Broadcast处理流程

在图6-20中，将调用函数所属的实际对象类型标注了出来，其中第11步的MyBroadcastReceiver为本例中所注册的广播接收者。

注意 任何广播对于静态注册者来说，都是 ordered，而且该order是全局性的，并非只针对该广播的接收者，故从广播发出到静态注册者的 onReceive函数被调用中间经历的这段时间相对较长。

6.5 startService之按图索骥

Service 是 Android 的四大组件之一。和 Activity, BroadcastReceiver 相比，Service 定位于业务层逻辑处理，而 Activity 定位于前端 UI 层逻辑处理，BroadcastReceiver 定位于通知逻辑的处理。

做为业务服务提供者，Service 自有一套规则，先来看有关 Service 的介绍。

6.5.1 Service 知识介绍

四大组件之一的 Service，其定义非常符合 C/S 架构中 Service 的概念，即为 Client 服务，处理 Client 的请求。在 Android 中，目前接触最多的是 Binder 中的 C/S 架构。在这种架构中，Client 通过调用预先定义好的业务函数向对应的 Service 发送请求。作为四大组件之一的 Service，其响应 Client 的请求方式有两种：

Client 通过调用 startService 向 Service 端发送一个 Intent，该 Intent 携带请求信息。而 Service 的 onStartCommand 会接收该 Intent，并处理之。该方式是 Android 平台特有的，借助 Intent 来传递请求。

Client 调用 bindService 函数和一个指定的 Service 建立 Binder 关系，即绑定成功后，Client 端将得到处理业务逻辑的 Binder Bp 端。此后 Client 直接调用 Bp 端提供的业务函数向 Service 端发出请求。注意，在这种方式中，Service 的 onBind 函数被调用，如果该 Service 支持 Binder，则需返回一个 IBinder 对象给客户端。

以上介绍的是 Service 响应客户端请求的两种方式，读者务必将两者分清楚。此外，这两种方式还影响 Service 对象的生命周期，简单总结如下：

对于以 startService 方式启动的 Service 对象，其生命周期一直延续到 stopSelf 或 stopService 被调用为止。

对于以 bindService 方式启动的 Service 对象，其生命周期延续到最后一个客户端调用完 unbindService 为止。

注意 生命周期控制一般都涉及引用计数的使用。如果某 Service 对象同时支持这两种请求方式，那么当总引用计数减为零时，其生命就走向终点。

和Service相关的知识还有，当系统内存不足时，系统如何处理Service。如果Service和UI某个部分绑定（例如类似通知栏中Music播放的信息），那么此Service优先级较高（可通过调用startForeground把自己变成一个前台Service），系统不会轻易杀死这些Service来回收内存。

以上这些内容都较简单，阅读SDK文档中Service的相关说明即可了解，具体路径为SDK路径/docs/guide/topics/fundamentals/services.html。

本章不分析和Service相关的函数的原因有二：

Service的处理流程和本章重点介绍的Activity的处理流程差不多，并且Service的处理逻辑更简单。能阅读到此处的读者，想必对拿下Service信心满满。

“授人以鱼，不如授人以渔”。希望读者在经历过如此大量而又复杂的代码分析考验后，能学会和掌握分析方法。

6.5.2 startService流程图

本节将以startService为分析对象，把相关的流程图描绘出来，旨在帮读者根据该流程图自行研读与Service相关的处理逻辑。startService调用轨迹如图6-21和图6-22所示。

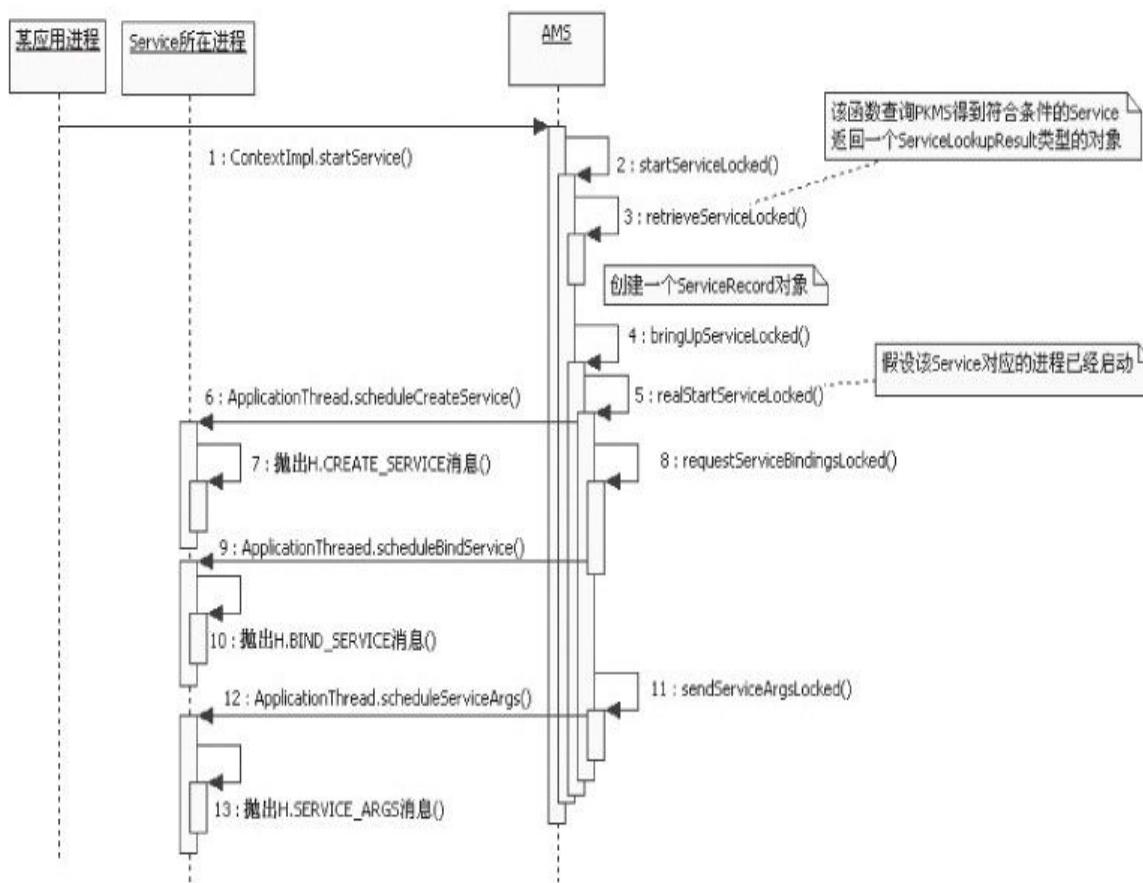


图 6-21 startService流程图之一

图6-21列出了和startService相关的调用流程。在这个流程中，可假设Service所对应的进程已经

存在。

单独提取图6-21中Service所在进程对H.CREATE_SERVICE等消息的处理流程具体如图6-22所示。

注意 图6-21和图6-22中也包含了bindService的处理流程。在实际分析时，读者可分开研究bindService和startService的处理流程。

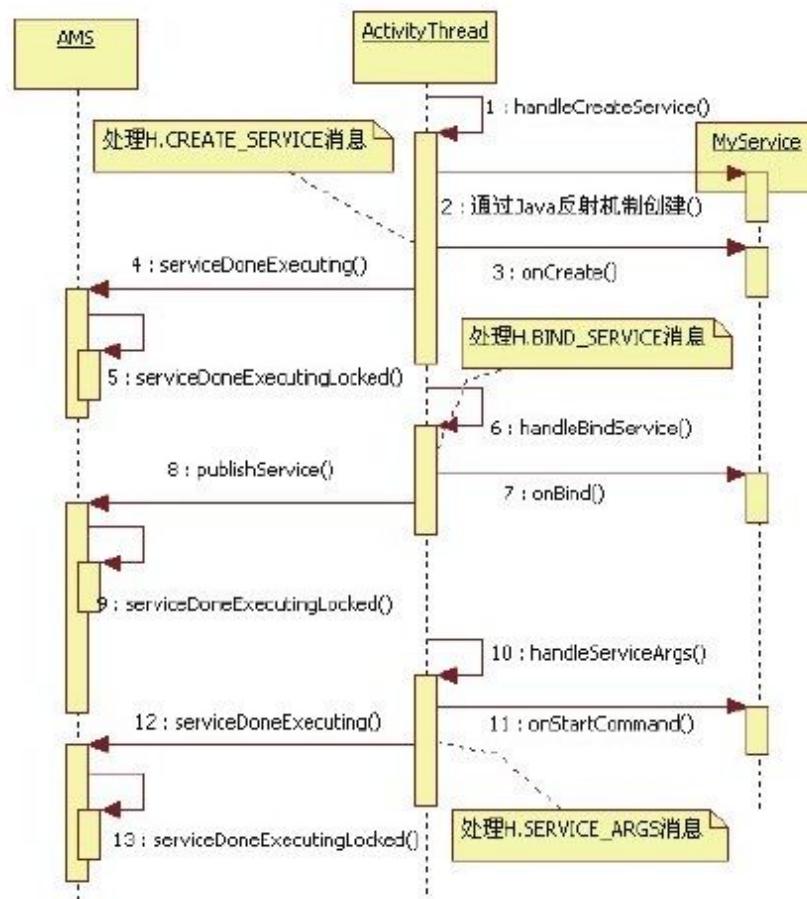


图 6-22 startService中相关Message的处理流程

6.6 AMS中的进程管理

前面曾反复提到，Android平台中很少能接触到进程的概念，取而代之的是有明确定义的四大组件。但是作为运行在Linux用户空间内的一个系统或框架，Android不仅不能脱离进程，反而要大力利用Linux操作系统提供的进程管理机制和手段，更好地为自己服务。作为Android平台中组件运行管理的核心服务，ActivityManagerService当仁不让地接手了这方面的工作。目前，AMS对进程的管理仅涉及两个方面：

调节进程的调度优先级和调度策略。

调节进程的oom值。

先来看在Linux操作系统中这两方面的进程管理和控制手段。

6.6.1 Linux进程管理介绍^[1]

1.Linux进程调度优先级和调度策略

调度优先级和调度策略是操作系统中一个很重要的概念。简而言之，它是系统中CPU资源的管理和控制手段。这又该如何理解？此处进行简单介绍。读者可自行阅读操作系统方面的书籍以加深理解。

相对于在OS（操作系统）上运行的应用进程个数来说，CPU的资源非常有限。

调度优先级是OS分配CPU资源给应用进程时（即调度应用进程运行）需要参考的一个指标。一般而言，优先级高的进程将更有机会得到CPU资源。

调度策略用于描述OS调度模块分配CPU给应用进程所遵循的规则，即当将CPU控制权交给调度模块时，系统如何选择下一个要运行的进程。此处不能仅考虑各进程的调度优先级，因为存在多个进程具有相同调度优先级的情况。在这种情况下，一个需要考虑的重要因素是，每个进程所分配的时间片及它们的使用情况。

提示 可简单认为，调度优先级及调度策略二者一起影响了系统分配CPU资源给应用进程。注意，此处的措词为“影响”，而非“控制”。因为对于用户空间可以调用的API来说，如果二者能控制

CPU资源分配，那么该系统的安全性会大打折扣。

Linux提供了两个API用于设置调度优先级及调度策略。先来看设置调度优先级的函数setpriority，其原型如下：

```
int setpriority (int which, int who, int prio) ;
```

其中：

which 和 who 参数联合使用。当 which 为 PRIO_PROGRESS 时， who 代表一个进程；当 which 为 PRIO_PGROUP 时， who 代表一个进程组；当 which 为 PRIO_USER 时， who 代表一个 uid。

第三个参数prio用于设置应用进程的nicer值，可取范围从-20到19。Linux kernel用nicer值来描述进程的调度优先级，该值越大，表明该进程越友好（nice），其被调度运行的几率越低。

下面来看设置调度策略的函数 sched_setscheduler，其原型如下：

```
int sched_setscheduler ( pid_t pid, int policy,  
const struct sched_param*param) ;
```

其中：

第一个参数为进程id。

第二个参数为调度策略。目前Android支持3种调度策略：SCHED_OTHER，标准round-robin分时共享策略（也就是默认的策略）；SCHED_BATCH，针对具有batch风格（批处理）进程的调度策略；SCHED_IDLE，针对优先级非常低的适合在后台运行的进程。除此之外，Linux还支持实时（Real-time）调度策略，包括SCHED_FIFO，先入先出调度策略；SCHED_RR，round-robin调度策略，也就是循环调度。

param参数中最重要的是该结构体中的sched_priority变量。针对Android中的3种非实时调度策略，该值必须为NULL。

以上介绍了调度优先级和调度策略的概念。建议读者做个小实验来测试调动优先级及调动策略的作用，步骤如下：

挂载SD卡到PC机并向其中复制一些媒体文件，然后重新挂载SD卡到手机。该操作就能触发MediaScanner扫描新增的这些媒体文件。

利用top命令查看CPU使用率，会发现进程android.process.media（即MediaScanner所在的进程）占用CPU较高（可达70%以上），原因是该进

程会扫描媒体文件，该工作将利用较多的CPU资源。

此时，如果启动另一个进程，然后做一些操作，会发现MediaScanner所在进程的CPU利用率会降下来（例如下降到30%~40%），这表明系统将CPU资源更多地分给了用户正在操作的这个进程。

出现这种现象的原因是，MediaScannerService的扫描线程将调度优先级设置为11，而默认的调度优先级为0。相比而言，MediaScannerService优先级就比较低。

2. 关于Linux进程oom_adj的介绍

从Linux kernel 2.6.11开始，内核提供了进程的OOM控制机制，目的是当系统出现内存不足（Out Of Memory, OOM）的情况时，Kernel可根据进程的oom_adj值的大小来选择并杀死一些进程，以回收内存。简而言之，oom_adj可表示Linux进程内存资源的优先级，其可取范围从-16到15，另外有一个特殊值-17用于禁止系统在OOM情况下杀死该进程。和nicer值一样，oom_adj的值越高，那么在OOM情况下，该进程越有可能被杀掉。每个进程的oom_adj初值为0。

Linux没有提供单独的API用于设置进程的oom_adj。目前的做法就是向/proc/进程id/oom_adj文件中写入对应的oom_adj值，通过这种方式就能调节进程的oom_adj了。

另外，有必要简单介绍一下Android为Linux Kernel新增的lowmemorykiller（以后简称LMK）模块的工作方式。LMK的职责是根据当前内存大小去杀死对应oom_adj及以上的进程以回收内存。这里有两个关键参数，即为LMK设置不同内存的阈值及oom_adj，它们分别由/sys/module/lowmemorykiller/parameters/minfree和/sys/module/lowmemorykiller/parameters/adj控制。

注意 这两个参数的典型设置为：

minfree，2048，3072，4096，6144，7168，8192用于描述不同级别的内存阈值，单位为KB。adj，0，1，2，4，7，15用于描述对应内存阈值的oom_adj值。

表示当剩余内存为2048KB时，LMK将杀死oom_adj大于等于0的进程。

网络上有一些关于Android手机内存优化的方法，其中一种就利用了LMK的工作原理。提示

lowmemorykiller 的 代 码 在
kernel/drivers/staging/android/lowmemorykiller.c
中，感兴

趣的读者可尝试自行阅读。

[1] 更 为 详 细 的 知 识 请 参 阅
<http://blog.csdn.net/innost/article/details/6940136>。

6.6.2 关于Android中的进程管理的介绍

前面介绍了Linux操作系统中进程管理（包括调度和OOM控制）方面的API，但AMS是如何利用它们的呢？这就要涉及AMS中的进程管理规则了。这里简单介绍相关规则。

Android将应用进程分为五大类，分别为Foreground类、Visible类、Service类、Background类及Empty类。这五大类的划分各有规则。

1. 进程分类

(1) Foreground类

该类中的进程重要性最高，属于该类的进程包括下面几种情况：

含一个前端Activity（即onResume函数被调用过了，或者说当前正在显示的那个Activity）。

含一个Service，并且该Service和一个前端Activity绑定（例如Music应用包括一个前端界面和一个播放Service，当我们一边听歌一边操作Music界面时，该Service即和一个前端Activity绑定）。

含一个调用了startForeground的Service，或者该进程的Service正在调用其生命周期的函数（onCreate、onStart或onDestroy）。

该进程中有一个BroadcastReceiver实例正在执行onReceive函数。

(2) Visible类

属于Visible类的进程中没有处于前端的组件，但是用户仍然能看到它们，例如位于一个对话框后的Activity界面。目前该类进程包括两种：

该进程包含一个仅onPause被调用的Activity（即它还在前台，只不过部分界面被遮住）。

包含一个Service，并且该Service和一个Visible（或Foreground）的Activity绑定（从字面上看，这种情况不太好和Foreground进程中第二种情况区分）。

(3) Service类、Background类及Empty类

这三类进程都没有可见的部分，具体情况如下。

Service进程：该类进程包含一个Service。此Service通过startService启动，并且不属于前面两类

进程。这种进程一般在后台默默地干活，例如前面介绍的Media-ScannerService。

Background进程：该类进程包含当前不可见的Activity（即它们的onStop被调用过）。系统保存这些进程到一个LRU（最近最少使用）列表。当系统需要回收内存时，该列表中那些最近最少使用的进程将被杀死。

Empty进程：这类进程中不包含任何组件。为什么会出现这种不包括任何组件的进程呢？其实很简单，假设该进程仅创建了一个Activity，它完成工作后主动调用finish函数销毁（destroy）自己，之后该进程就会成为Empty进程。系统保留Empty进程的原因是当又重新需要它们时（例如用户在别的进程中通过startActivity启动了它们），可以省去fork进程、创建Android运行环境等一系列漫长而艰苦的工作。

通过以上介绍可发现，当某个进程和前端显示有关系时，其重要性相对要高，这或许是体现Google重视用户体验的一个很直接的证据吧。

建议读者可阅读SDK/docs/guide/topics/fundamentals/processes-and-threads.html以获取更为详细的信息。

2.Process类API介绍

下面介绍Android平台中进程调度和OOM控制的API，它们统一被封装在Process.java中，其相关代码如下：

[-->Process.java]

//设置线程的调度优先级，Linux kernel并不区分线程和进程，二者对应同一个数据结构Task

```
public static final native void setThreadPriority (int tid,  
int priority)  
throws IllegalArgumentException, SecurityException;  
/*
```

设置线程的Group，实际上就是设置线程的调度策略，目前Android定义了3种Group。

由于缺乏相关资料，加之笔者感觉对应的注释也只可意会不可言传，故此处直接照搬了英文

注释，敬请读者谅解。

```
THREAD_GROUP_DEFAULT      : Default      thread      group-gets  
a 'normal' share of the CPU
```

```
THREAD_GROUP_BG_NONINTERACTIVE : Background  non-interactive  
thread group.
```

```
All threads in this group are scheduled with a reduced share  
of the CPU
```

```
THREAD_GROUP_FG_BOOST   : Foreground 'boost' thread  group-All  
threads in
```

this group are scheduled with an increased share of the CPU.

目前代码中还没有地方使用THREAD_GROUP_FG_BOOST这种Group

```
*/
```

```
public static final native void setThreadGroup (int tid, int  
group)
```

```
throws IllegalArgumentException, SecurityException;
```

//设置进程的调度策略，包括该进程的所有线程

```
public static final native void setProcessGroup (int pid,  
int group)  
throws IllegalArgumentException, SecurityException;  
//设置线程的调度优先级  
public static final native void setThreadPriority ( int  
priority)  
throws IllegalArgumentException, SecurityException;  
//调整进程的oom_adj值  
public static final native boolean setOomAdj (int pid, int  
amt) ;
```

Process类还为不同调度优先级定义一些非常直观的名字以避免在代码中直接使用整型，例如为最低的调度优先级19定义了整型变量THREAD_PRIORITY_LOWEST。除此之外，Process还提供了fork子进程等相关的函数。

注意 Process.java中的大多数函数是由JNI层实现的，其中Android在调度策略设置这一功能上还有一些特殊的地方，感兴趣的读者不妨阅读system/core/libcutils/sched_policy.c文件。

3. 关于ProcessList类和ProcessRecord类的介绍

(1) ProcessList类的介绍

ProcessList类有两个主要功能：

定义一些成员变量，这些成员变量描述了不同状态下进程的oom_adj值。

在Android 4.0之后，LMK的配置参数由ProcessList综合考虑手机总内存大小和屏幕尺寸后再行设置（在Android 2.3中，LMK的配置参数在init.rc中由init进程设置，并且没有考虑屏幕尺寸的影响）。读者可自行阅读updateOomLevels函数，此处不再赘述。

本节主要关注ProcessList对oom_adj的定义。虽然前面介绍时将Android进程分为五大类，但是在实际代码中的划分更为细致，考虑得更为周全。

[-->ProcessList.java]

```
class ProcessList{  
    //当一个进程连续发生Crash的间隔小于60秒时，系统认为它为Bad进程  
    static final int MIN_CRASH_INTERVAL=60*1000;  
    //下面定义各种状态下进程的oom_adj  
    //不可见进程的oom_adj，最大15，最小为7。杀死这些进程一般不会带来用户  
    //能够察觉的影响  
    static final int HIDDEN_APP_MAX_ADJ=15;  
    static int HIDDEN_APP_MIN_ADJ=7;  
    /*  
     * B List中的Service所在进程的oom_adj。什么样的Service会在B List中  
     * 呢？其中的  
     * 解释只能用原文来表达“these are the old and decrepit services  
     * that aren't as  
     * shiny and interesting as the ones in the A list”  
     */  
    static final int SERVICE_B_ADJ=8;  
    /*
```

前一个进程的oom_adj，例如从A进程的Activity跳转到位于B进程的Activity，

B进程是当前进程，而A进程就是previous进程了。因为从当前进程A退回B进程是一个很简单

却很频繁的操作（例如按back键退回上一个Activity），所以previous进程的oom_adj

需要单独控制。这里不能简单按照五大类来划分previous进程，还需要综合考虑

*/

```
static final int PREVIOUS_APP_ADJ=7 ;
```

//Home进程的oom_adj为6，用户经常和Home进程交互，故它的oom_adj也需要单独控制

```
static final int HOME_APP_ADJ=6 ;
```

//Service类中进程的oom_adj为5

```
static final int SERVICE_APP_ADJ=5 ;
```

//正在执行backup操作的进程，其oom_adj为4

```
static final int BACKUP_APP_ADJ=4 ;
```

//heavy weight的概念前面曾提到过，该类进程的oom_adj为3

```
static final int HEAVY_WEIGHT_APP_ADJ=3 ;
```

//Perceptible进程指那些当前并不在前端显示而用户能感觉到它在运行的进程，例如正在

//后台播放音乐的Music

```
static final int PERCEPTIBLE_APP_ADJ=2 ;
```

//Visible类进程，oom_adj为1

```
static final int VISIBLE_APP_ADJ=1 ;
```

//Foreground类进程，oom_adj为0

```
static final int FOREGROUND_APP_ADJ=0 ;
```

//persistent类进程（即退出后，系统需要重新启动的重要进程），其oom_adj为-12

```
static final int PERSISTENT_PROC_ADJ=-12 ;
```

//核心服务所在进程，oom_adj为-12

```
static final int CORE_SERVER_APP_ADJ=-12 ;
```

//系统进程，其oom_adj为-16

```
static final int SYSTEM_APP_ADJ=-16 ;
```

//内存页大小定义为4KB

```
static final int PAGE_SIZE=4*1024 ;
```

```
//系统中能容纳的不可见进程数最少为2，最多为15。在Android 4.0系统中，  
可通过设置程序来  
//调整  
static final int MIN_HIDDEN_APPS=2 ;  
static final int MAX_HIDDEN_APPS=15 ;  
//下面两个参数用于调整进程的LRU权重。注意，它们的注释和具体代码中的用法  
不能统一起来  
static final long CONTENT_APP_IDLE_OFFSET=15*1000 ;  
static final long EMPTY_APP_IDLE_OFFSET=120*1000 ;  
//LMK设置了6个oom_adj阈值  
private final int[]mOomAdj=new int[]{  
    FOREGROUND_APP_ADJ, VISIBLE_APP_ADJ, PERCEPTIBLE_APP_ADJ,  
    BACKUP_APP_ADJ, HIDDEN_APP_MIN_ADJ, EMPTY_APP_ADJ  
};  
//用于内存较低机器（例如小于512MB）中LMK的内存阈值配置  
private final long[]mOomMinFreeLow=new long[]{  
    8192, 12288, 16384,  
    24576, 28672, 32768  
};  
//用于较高内存机器（例如大于1GB）的LMK内存阈值配置  
private final long[]mOomMinFreeHigh=new long[]{  
    32768, 40960, 49152,  
    57344, 65536, 81920  
};
```

从以上代码中定义的各种oom_adj值可知，AMS中的进程管理规则远比想象的要复杂（读者以后见识到具体的代码，更会有这样的体会）。

说明 在ProcessList中定义的大部分变量在Android 2.3 代码中定义于ActivityManagerService.java中，但这段代码的开发者仅把原代码

复制了过来，其中的注释并未随着系统升级而更新。

(2) ProcessRecord中相关成员变量的介绍

ProcessRecord定义了较多成员变量用于进程管理。笔者不打算深究其中的细节。这里仅把其中的主要变量及一些注释列举出来。下文会分析它们的作用。

[-->ProcessRecord.java]

```
//用于LRU列表控制
long lastActivityTime ; //For managing the LRU list
long lruWeight ; //Weight for ordering in LRU list
//和oom_adj有关
int maxAdj ; //Maximum OOM adjustment for this process
int hiddenAdj ; //If hidden, this is the adjustment to use
int curRawAdj ; //Current OOM unlimited adjustment for this
process
int setRawAdj ; //Last set OOM unlimited adjustment for this
process
int curAdj ; //Current OOM adjustment for this process
int setAdj ; //Last set OOM adjustment for this process
//和调度优先级有关
int curSchedGroup ; //Currently desired scheduling class
int setSchedGroup ; //Last set to background scheduling class
//回收内存级别，见后文解释
int trimMemoryLevel ; //Last selected memory trimming level
//判断该进程的状态，主要和其中运行的Activity、Service有关
boolean keeping ; //Actively running code so don't kill due
to that?
```

```
boolean setIsForeground ; //Running foreground UI when last
set?
boolean foregroundServices ; //Running any services that are
foreground?
boolean foregroundActivities ; //Running any activities that
are foreground?
boolean systemNoUi ; //This is a system process, but not
currently showing UI.
boolean hasShownUi ; //Has UI been shown in this process
since it
was started?
boolean pendingUiClean ; //Want to clean up resources from
showing UI?
boolean hasAboveClient ; //Bound using BIND_ABOVE_CLIENT, so
want to be lower
//是否处于系统BadProcess列表
boolean bad ; //True if disabled in the bad process list
//描述该进程是否因为有太多后台组件而被杀死
boolean killedBackground ; //True when proc has been killed
due to too many bg
String waitingToKill ; //Process is waiting to be killed when
in the
bg ; reason
//序号，每次调节进程优先级或者LRU列表位置时，这些序号都会递增
int adjSeq ; //Sequence id for identifying oom_adj assignment
cycles
int lruSeq ; //Sequence id for identifying LRU update cycles
```

上面注释中提到了LRU（很少使用）一词，它和AMS另外一个用于管理应用进程ProcessRecord的数据结构有关。

提示 进程管理和调度一向比较复杂，从ProcessRecord定义的这些变量中可见一斑。需要

提醒读者的是，对这部分功能的相关说明非常少，故代码读起来会感觉比较晦涩。

6.6.3 AMS进程管理函数分析

在AMS中，和进程管理有关的函数只要有两个，分别是updateLruProcessLocked和updateOomAdjLocked。这两个函数的调用点有多处，本节以attachApplication为切入点，尝试对它们进行分析。

注意 AMS 一共定义了3个updateOomAdjLocked函数，此处将其归为一类。

先回顾一下attachApplication函数被调用的情况：AMS新创建一个应用进程，该进程启动后最重要的就是调用AMS的attachApplication。

提示 不熟悉的读者可阅读6.3.3节。

//attachApplication 主要工作由
attachApplicationLocked 完成，故直接分析
attachApplicationLocked，其相关代码如下：

[-->ActivityManagerService.java :
attachApplicationLocked]

```
private final boolean attachApplicationLocked
(IApplicationThread thread,
 int pid) {
```

```
ProcessRecord app ;
//根据之前的介绍的内容，AMS在创建应用进程前已经将对应的ProcessRecord
保存到
//mPidSelfLocked中了
.....//其他一些处理
//初始化ProcessRecord中的一些成员变量
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 ;
.....
//调用应用进程的bindApplication，以初始化其内部的Android运行环境
thread.bindApplication (.....) ;
//①调用updateLruProcessLocked函数
updateLruProcessLocked (app, false, true) ;
app.lastRequestedGc=app.lastLowMemory=SystemClock.uptimeMillis
is () ;
.....//启动Activity等操作
//②didSomething为false，则调用updateOomAdjLocked函数
if (!didSomething) {
updateOomAdjLocked () ;
}
```

在以上这段代码中调用了两个重要函数，分别是 updateLruProcessLocked 函数 和 update-Oom-AdjLocked 函数。

1.updateLruProcessLocked函数分析

根据前文所述，我们知道了系统中所有应用进程（同时包括system_server）的Process-Record信息都保存在mPidsSelfLocked成员中。除此之外，AMS还有一个成员变量mLru-Processes也用于保存ProcessRecord。mLruProcesses的类型虽然是ArrayList，但其内部成员却是按照ProcessRecord的lruWeight大小排序的。在运行过程中，AMS会根据lruWeight的变化调整mLruProcesses成员的位置。

就本例而言，刚连接（attach）上的这个应用进程的ProcessRecord需要通过updateLruProcessLocked函数加入mLruProcesses数组中。下面来看updateLruProcessLocked函数的代码，如下所示：

[-->ActivityManagerService.java
updateLruProcessLocked]

```
final void updateLruProcessLocked (ProcessRecord app,
boolean oomAdj, boolean updateActivityTime) {
    mLruSeq++ ; //每一次调整LRU列表，系统都会分配一个唯一的编号
    updateLruProcessInternalLocked      (      app,      oomAdj,
updateActivityTime, 0) ;
}
```

[-->ActivityManagerService.java
updateLruProcessInternalLocked]

```
private final void updateLruProcessInternalLocked
(ProcessRecord app,
    boolean oomAdj, boolean updateActivityTime, int bestPos) {
    //获取app在mLruProcesses中的索引位置，对于本例而言，返回值lrui为-1
    int lrui=mLruProcesses.indexOf (app) ;
    //如果之前有记录，则先从数组中删掉，因为此处需要重新调整位置
    if (lrui>=0) mLruProcesses.remove (lrui) ;
    //获取mLruProcesses中数组索引的最大值，从0开始
    int i=mLruProcesses.size () -1;
    int skipTop=0;
    app.lruSeq=mLruSeq ;//将系统全局的lru调整编号赋给ProcessRecord的
    lruSeq
    //更新lastActivityTime值，其实就是获取一个时间
    if (updateActivityTime) {
        app.lastActivityTime=SystemClock.uptimeMillis () ;
    }
    if (app.activities.size () >0) {
        //如果该app含Activity，则lruWeight为当前时间
        app.lruWeight=app.lastActivityTime ;
    }else if (app.pubProviders.size () >0) {
        /*
        如果有发布的ContentProvider，则lruWeight要减去一个OFFSET。
        对此的理解需结合CONTENT_APP_IDLE_OFFSET的定义。读者暂时把它
        看做一个常数
        */
        app.lruWeight=app.lastActivityTime-
        ProcessList.CONTENT_APP_IDLE_OFFSET ;
        //设置skipTop。这个变量实际上没有用，放在此处让人很头疼
        skipTop=ProcessList.MIN_HIDDEN_APPS ;
    }else{
        app.lruWeight=app.lastActivityTime-
        ProcessList.EMPTY_APP_IDLE_OFFSET ;
        skipTop=ProcessList.MIN_HIDDEN_APPS ;
    }
    //从数组最后一个元素开始循环
```

```
while (i>=0) {
    ProcessRecord p=mLruProcesses.get (i) ;
    //下面这个if语句没有任何意义，因为skipTop除了做自减操作外，不影响其他
    //任何内容
    if (skipTop>0&&p.setAdj>=ProcessList.HIDDEN_APP_MIN_ADJ) {
        skipTop-- ;
    }
    //将app调整到合适的位置
    if (p.lruWeight<=app.lruWeight||i<bestPos) {
        mLruProcesses.add (i+1, app) ;
        break ;
    }
    i-- ;
}
//如果没有找到合适的位置，则把app加到队列头
if (i<0) mLruProcesses.add (0, app) ;
//如果该将app绑定到其他Service，则要对应调整Service所在进程的LRU
if (app.connections.size () >0) {
    for (ConnectionRecord cr : app.connections) {
        if (cr.binding !=null&&cr.binding.service !=null
            &&cr.binding.service.app !=null
            &&cr.binding.service.app.lruSeq !=mLruSeq) {
            updateLruProcessInternalLocked (cr.binding.service.app,
                oomAdj, updateActivityTime, i+1) ;
        }
    }
}
//conProviders也是一种Provider，相关信息下一章再介绍
if (app.conProviders.size () >0) {
    for (ContentProviderRecord cpr : app.conProviders.keySet () )
{
    .....//对ContentProvider所在进程做类似的调整
}
}
//在本例中，oomAdj为false，故updateOomAdjLocked不会被调用
if (oomAdj) updateOomAdjLocked () ;//以后分析
```

}

由以上代码可知，updateLruProcessLocked的主要工作是根据app的lruWeight值调整它在数组中的位置。lruWeight值越大，其在数组中的位置就越靠后。如果该app和某些Service（仅考虑通过bindService建立关系的那些Service）或ContentProvider有交互关系，那么这些Service或ContentProvider所在的进程也需要调节lruWeight值。

下面介绍第二个重要函数updateOomAdjLocked。

提示 在以上代码中，skipTop变量完全没有实际作用，却给为阅读代码带来了很大干扰。

2.updateOomAdjLocked函数分析

分段来看updateOomAdjLocked函数。

（1）updateOomAdjLocked分析之一

这部分的代码如下：

[-->ActivityManagerService.java :
updateOomAdjLocked]

```
final void updateOomAdjLocked () {
```

```
//在一般情况下，resumedAppLocked返回mResumedActivity，即当前正处于前台的Activity
final ActivityRecord TOP_ACT=resumedAppLocked () ;
//得到前台Activity所属进程的ProcessRecord信息
final ProcessRecord TOP_APP=TOP_ACT ! =null?TOP_ACT.app :
null;
mAdjSeq++ ; //oom_adj在进行调节时也会有唯一的序号
mNewNumServiceProcs=0 ;
/*
```

下面这几句代码的作用如下：

1. 根据hidden adj划分级别，一共有9个级别（即numSlots值）
2. 根据mLruProcesses的成员个数计算平均落在各个级别的进程数（即factor值）。但是这里

的魔数（magic number）4却令人头疼不已。如有清楚该内容的读者，不妨分享一下研究结果

```
/*
int numSlots=ProcessList.HIDDEN_APP_MAX_ADJ-
ProcessList.HIDDEN_APP_MIN_ADJ+1 ;
int factor= (mLruProcesses.size () -4) /numSlots ;
if (factor<1) factor=1 ;
int step=0 ;
int numHidden=0 ;
int i=mLruProcesses.size () ;
int curHiddenAdj=ProcessList.HIDDEN_APP_MIN_ADJ ;
//从mLruProcesses数组末端开始循环
while (i>0) {
i-- ;
ProcessRecord app=mLruProcesses.get (i) ;
//①调用另外一个updateOomAdjLocked函数
updateOomAdjLocked (app, curHiddenAdj, TOP_APP, true) ;
//updateOomAdjLocked函数会更新app的curAdj
if (curHiddenAdj<ProcessList.HIDDEN_APP_MAX_ADJ
&&app.curAdj==curHiddenAdj) {
/*
```

这段代码的目的其实很简单。即当某个adj级别的ProcessRecord处理个数超过均值后，

就跳到下一级别进行处理。注意，这段代码的结果会影响updateOomAdjLocked的第二个参数

```
/*
step++ ;
if (step>=factor) {
step=0 ;
curHiddenAdj++ ;
}
}//if (curHiddenAdj<ProcessList.HIDDEN_APP_MAX_ADJ.....) 判断结束
//app.killedBackground初值为false
if (! app.killedBackground) {
if (app.curAdj>=ProcessList.HIDDEN_APP_MIN_ADJ) {
numHidden++ ;
//mProcessLimit初始值为ProcessList.MAX (值为15) ,
//可通过setProcessLimit函数对其进行修改
if (numHidden>mProcessLimit) {
app.killedBackground=true ;
//如果后台进程个数超过限制，则会杀死对应的后台进程
Process.killProcessQuiet (app.pid) ;
}
}
}
}//if (! app.killedBackground) 判断结束
}//while循环结束
```

updateOomAdjLocked第一阶段的工作看起来很简单，但是其中也包含一些较难理解的内容，具体如下：

处理hidden adj，划分9个级别。

根据mLruProcesses中进程个数计算每个级别平均会存在多少进程。在这个计算过程中出现了

一个魔数4令人极度费解。

利用一个循环从mLruProcesses末端开始对每个进程执行另一个updateOomAdj-Locked函数。关于这个函数的内容，我们放到下一节再讨论。

判断处于Hidden状态的进程数是否超过限制，如果超过限制，则会杀死一些进程。接着来看updateOomAdjLocked下一阶段的工作。

(2) updateOomAdjLocked分析之二

这部分的代码如下：

[-->ActivityManagerService.java :
updateOomAdjLocked]

```
mNumServiceProcs=mNewNumServiceProcs ;  
//numHidden表示处于hidden状态的进程个数  
//当Hidden进程个数小于7时（15/2的整型值），执行if分支  
if (numHidden<= (ProcessList.MAX_HIDDEN_APPS/2) ) {  
....  
/*
```

我们不讨论这段缺乏文档及使用魔数的代码，但这里有个知识点要注意：

该知识点和Android 4.0新增接口ComponentCallbacks2有关，主要是通知应用进程进行

内存清理，ComponentCallbacks2接口定义了一个函数onTrimMemory (int level) ，

而四大组件除BroadcastReceiver外，均实现了该接口。系统定义了4个level以通知进程

做对应处理：

TRIM_MEMORY_UI_HIDDEN，提示进程当前不处于前台，故可释放一些UI资源

TRIM_MEMORY_BACKGROUND，表明该进程已加入LRU列表，此时进程可以对一些简单的资源

进行清理

TRIM_MEMORY_MODERATE，提示进程可以释放一些资源，这样其他进程的日子会好过些。

即所谓的“我为人人，人人为我”

TRIM_MEMORY_COMPLETE，该进程需尽可能释放一些资源，否则当内存不足时，它可能会被杀死

```
/*
}else{//假设hidden进程数超过7,
final int N=mLruProcesses.size () ;
for (i=0 ; i<N ; i++) {
ProcessRecord app=mLruProcesses.get (i) ;
if (app.curAdj>
ProcessList.VISIBLE_APP_ADJ||app.systemNoUi)
&&app.pendingUiClean) {
if (app.trimMemoryLevel<
ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN
&&app.thread !=null) {
try{//调用应用进程ApplicationThread的scheduleTrimMemory函数
app.thread.scheduleTrimMemory (
ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN) ;
}.....
}//if (app.trimMemoryLevel.....) 判断结束
app.trimMemoryLevel=
ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN ;
app.pendingUiClean=false ;
}else{
app.trimMemoryLevel=0 ;
}
}//for循环结束
}//else结束
//Android 4.0中设置有一个开发人员选项，其中有一项用于控制是否销毁后台的Activity
//读者可自行研究destroyActivitiesLocked函数
if (mAlwaysFinishActivities)
```

```
mMainStack.destroyActivitiesLocked ( null, false , "always-  
finish") ;  
}
```

通过上述代码，可获得两个信息：

Android 4.0 增加了新的接口类 ComponentCallbacks2，其中只定义了一个函数 onTrimMemory。从以上描述中可知，它主要通知应用进程进行一定的内存释放。

Android 4.0 Settings新增了一个开放人员选项，通过它可控制AMS对后台Activity的操作。

这里和读者探讨一下ComponentCallbacks2接口的意义。此接口的目的是通知应用程序根据情况做一些内存释放，但笔者觉得，这种设计方案的优劣尚有待考证，这主要是出于以下几种考虑：

不是所有应用程序都会实现该函数。原因有很多，主要原因是，该接口只是SDK 14才有的，之前的版本没有这个接口。另外，应用程序都会尽可能抢占资源（在不超过允许范围内）以保证运行速度，不应该考虑其他程序的事情。

无法区分在不同的level下到底要释放什么样的内存。代码中的注释也是含糊其辞。到底什么

样 的 资 源 可 以 在
TRIM_MEMORY_BACKGROUND 级别下释放，
什 么 样 的 资 源 不 可 以 在
TRIM_MEMORY_BACKGROUND 级别下释放？

既然系统加了这些接口，读者不妨参考源码中的使用案例来开发自己的程序。

建议 真诚希望 Google 能给出一个明确的文档，说明这几个函数该怎么使用。

接下来分析在以上代码中出现的针对每个 ProcessRecord 都 调 用 的 updateOomAdjLocked 函数。

3. 第二个 updateOomAdjLocked 分析 [1]

这部分的代码如下：

[--> ActivityManagerService.java :
updateOomAdjLocked]

```
private final boolean updateOomAdjLocked (ProcessRecord app,  
int hiddenAdj,  
ProcessRecord TOP_APP, boolean doingAll) {  
    //设置该app的hiddenAdj  
    app.hiddenAdj=hiddenAdj ;  
    if (app.thread==null) return false ;  
    final boolean wasKeeping=app.keeping ;  
    boolean success=true ;
```

```
//下面这个函数的调用极其关键。从名字上看，它会计算该进程的oom_adj及调度策略
    computeOomAdjLocked ( app,    hiddenAdj,    TOP_APP,    false,
doingAll) ;
    if (app.curRawAdj !=app.setRawAdj) {
        if (wasKeeping&& ! app.keeping) {
            .....//统计电量
            app.lastCpuTime=app.curCpuTime ;
        }
        app.setRawAdj=app.curRawAdj ;
    }
    //如果新旧oom_adj不同，则重新设置该进程的oom_adj
    if (app.curAdj !=app.setAdj) {
        if (Process.setOomAdj (app.pid, app.curAdj) ) //设置该进程的
        oom_adj
        app.setAdj=app.curAdj ;
        .....
    }
    //如果新旧调度策略不同，则需重新设置该进程的调度策略
    if (app.setSchedGroup !=app.curSchedGroup) {
        app.setSchedGroup=app.curSchedGroup ;
        //waitingToKill是一个字符串，用于描述杀掉该进程的原因
        if (app.waitingToKill !=null&&
        app.setSchedGroup==Process.THREAD_GROUP_BG_NONINTERACTIVE) {
            Process.killProcessQuiet (app.pid) ; //
            success=false ;
        }else{
            if (true) {//强制执行if分支
                long oldId=Binder.clearCallingIdentity () ;
                try{//设置进程调度策略
                    Process.setProcessGroup (app.pid, app.curSchedGroup) ;
                }.....
            }.....
        }
    }
    return success ;
```

}

上面的代码还算简单，主要完成两项工作：

调用computeOomAdjLocked计算获得某个进程的oom_adj和调度策略。

调整进程的调度策略和oom_adj。

建议 思考一个问题：为何AMS只设置进程的调度策略，而不设置进程的调度优先级？

看来AMS调度算法的核心就在computeOomAdjLocked中。

4.computeOomAdjLocked分析

这段代码较长，其核心思想是综合考虑各种情况以计算进程的oom_adj和调度策略。建议读者阅读代码时聚焦到AMS关注的几个因素上。computeOomAdjLocked的代码如下：

[-->ActivityManagerService.java :
computeOomAdjLocked]

```
private final int computeOomAdjLocked ( ProcessRecord app,  
int hiddenAdj,  
ProcessRecord TOP_APP, boolean recursed, boolean doingAll) {  
....
```

```
    app.adjTypeCode=ActivityManager.RunningAppProcessInfo.REASON  
_UNKNOWN ;  
    app.adjSource=null ;  
    app.adjTarget=null ;  
    app.empty=false ;  
    app.hidden=false ;  
    //该应用进程包含Activity的个数  
    final int activitiesSize=app.activities.size () ;  
    //如果maxAdj小于FOREGROUND_APP_ADJ，基本上就没什么工作可以做了。这  
类进程优先级相当高  
    if (app.maxAdj<=ProcessList.FOREGROUND_APP_ADJ) {  
.....//读者可自行阅读这块代码  
        return (app.curAdj=app.maxAdj) ;  
    }  
    final boolean  
hadForegroundActivities=app.foregroundActivities ;  
    app.foregroundActivities=false ;  
    app.keeping=false ;  
    app.systemNoUi=false ;  
    int adj ;  
    int schedGroup ;  
    //如果app为前台Activity所在的那个应用进程  
    if (app==TOP_APP) {  
        adj=ProcessList.FOREGROUND_APP_ADJ ;  
        schedGroup=Process.THREAD_GROUP_DEFAULT ;  
        app.adjType="top-activity" ;  
        app.foregroundActivities=true ;  
    }else if (app.instrumentationClass !=null) {  
.....//略过instrumentationClass不为null的情况  
    }else if (app.curReceiver !=null||  
        ( mPendingBroadcast !=null &&  
        mPendingBroadcast.curApp==app) ) {  
        //此情况对应正在执行onReceive函数的广播接收者所在进程，它的优先级也很  
高  
        adj=ProcessList.FOREGROUND_APP_ADJ ;  
        schedGroup=Process.THREAD_GROUP_DEFAULT ;
```

```
app.adjType="broadcast" ;
}else if (app.executingServices.size () >0) {
//正在执行Service生命周期函数的进程
adj=ProcessList.FOREGROUND_APP_ADJ ;
schedGroup=Process.THREAD_GROUP_DEFAULT ;
app.adjType="exec-service" ;
}else if (activitiesSize>0) {
adj=hiddenAdj ;
schedGroup=Process.THREAD_GROUP_BG_NONINTERACTIVE ;
app.hidden=true ;
app.adjType="bg-activities" ;
}else{//不含任何组件的进程，即所谓的Empty进程
adj=hiddenAdj ;
schedGroup=Process.THREAD_GROUP_BG_NONINTERACTIVE ;
app.hidden=true ;
app.empty=true ;
app.adjType="bg-empty" ;
}
```

//下面几段代码将根据情况重新调整前面计算得到的adj和schedGroup，请读者注意下面代码中对Home

进程的特殊处理

```
if (! app.foregroundActivities && activitiesSize>0) {
//对无前台Activity所在进程的处理
}
if (adj>ProcessList.PERCEPTIBLE_APP_ADJ) {
.....
}
```

//如果前面计算出来的adj大于HOME_APP_ADJ，并且该进程又是Home进程，则需要重新调整

```
if (adj>ProcessList.HOME_APP_ADJ && app==mHomeProcess) {
//重新调整adj和schedGroup的值
adj=ProcessList.HOME_APP_ADJ ;
schedGroup=Process.THREAD_GROUP_BG_NONINTERACTIVE ;
app.hidden=false ;
app.adjType="home" ; //描述调节adj的原因
}
```

```

    if      (      adj>ProcessList.PREVIOUS_APP_ADJ      &      &
app==mPreviousProcess
    &&app.activities.size () >0) {
    ....
}
app.adjSeq=mAdjSeq ;
app.curRawAdj=adj ;
.....
//下面这几段代码处理那些进程中含有Service、ContentProvider组件情况
下的adj调节
if  (  app.services.size   ( )   !   =0   &   &   (  adj>
ProcessList.FOREGROUND_APP_ADJ
||schedGroup==Process.THREAD_GROUP_BG_NONINTERACTIVE) ) {
}
if  (  s.connections.size   ( )   >0   &   &   (  adj>
ProcessList.FOREGROUND_APP_ADJ
||schedGroup==Process.THREAD_GROUP_BG_NONINTERACTIVE) ) {
}
if  (  app.pubProviders.size   ( )   !   =0   &   &   (  adj>
ProcessList.FOREGROUND_APP_ADJ
||schedGroup==Process.THREAD_GROUP_BG_NONINTERACTIVE) ) {
}
.....
}
//终于计算完毕
app.curRawAdj=adj ;
if (adj>app.maxAdj) {
adj=app.maxAdj ;
if (app.maxAdj<=ProcessList.PERCEPTIBLE_APP_ADJ)
schedGroup=Process.THREAD_GROUP_DEFAULT ;
}
if (adj<ProcessList.HIDDEN_APP_MIN_ADJ)
app.keeping=true ;
.....
app.curAdj=adj ;
app.curSchedGroup=schedGroup ;
.....

```

```
    return app.curRawAdj;
}
```

computeOomAdjLocked的工作比较琐碎，实际上也谈不上什么算法，仅仅是简单地根据各种情况来设置几个值。随着系统的改进和完善，这部分代码变动的可能性比较大。

5.updateOomAdjLocked调用点统计

updateOomAdjLocked调用点很多，这里给出其中一个updateOomAdjLocked（ProcessRecord）函数的调用点统计，如图6-23所示。



图 6-23 updateOomAdjLocked
(ProcessRecord) 函数的调用点统计图

从图6-23中可知，此函数被调用的地方较多，这也说明AMS非常关注应用进程的状况。

提示 笔者觉得，AMS中这部分代码不是特别高效，不知各位读者是否有同感？

[1]有两个名为updateOomAdjLocked的函数，这里分析的是第二个。

6.6.4 AMS进程管理总结

本节首先向读者介绍了Linux平台中和进程调度、OOM管理相关的API，然后介绍了AMS如何利用这些API完成Android平台中进程管理方面的工作，从中可以发现，AMS设置的检查点比较密集，也就是说经常会进行进程调度方面的操作。

6.7 App的Crash处理

在Android平台中，应用进程fork出来后会为虚拟机设置一个未截获异常处理器，即在程序运行时，如果有任何一个线程抛出了未被截获的异常，那么该异常最终会抛给未截获异常处理器去处理。设置未截获异常处理器的代码如下：

[-->RuntimeInit.java : commonInit]

```
private static final void commonInit () {  
    //调用完毕后，该应用中所有线程抛出的未处理异常都会由UncaughtHandler  
    来处理  
    Thread.setDefaultUncaughtExceptionHandler (new  
    UncaughtHandler () );  
    ....  
}
```

应用程序有问题是最平常不过的事情了，不过，当抛出的异常没有被截获时，系统又会做什么处理呢？来看UncaughtHandler的代码。

6.7.1 应用进程的Crash处理

UncaughtHandler的代码如下：

[-->RuntimeInit.java : UncaughtHandler]

```
private static class UncaughtHandler implements
Thread.UncaughtExceptionHandler{
public void uncaughtException (Thread t, Throwable e) {
try{
if (mCrashing) return ;
mCrashing=true ;
// 调用 AMS 的 handleApplicationCrash 函数 。 第一个参数
mApplicationObject其实
//就是前面经常见到的ApplicationThread对象
ActivityManagerNative.getDefault () .handleApplicationCrash (
mApplicationObject, new ApplicationErrorReport.CrashInfo
(e) ) ;
}.....
finally{
// 这里有一句注释很有意思：Try everything to make sure this
process goes
//away.从下面这两句调用来看，该应用进程确实想方设法要离开Java世界
Process.killProcess (Process.myPid () ) ;//把自己杀死
System.exit (10) ;//如果上面那句话不成功，则再尝试exit方法
}
}
}
```

6.7.2 AMS的handleApplicationCrash分析

AMS handleApplicationCrash 函数 的 代 码 如
下：

[-->ActivityManagerService.java :
handleApplicationCrash]

```
public void handleApplicationCrash (IBinder app,  
ApplicationErrorReport.CrashInfo crashInfo) {  
    //找到对应的ProcessRecord信息，后面那个字符串“Crash”用于打印输出  
    ProcessRecord r=findAppProcess (app, "Crash") ;  
    final String processName=app==null?"system_server"  
    : (r==null?"unknown":r.processName) ;  
    //添加crash信息到dropbox中  
    ddErrorToDropBox ("crash", r, processName, null, null, null,  
    null, null,  
    crashInfo) ;  
    //调用crashApplication函数  
    crashApplication (r, crashInfo) ;  
}
```

上述代码中的crashApplication函数的代码如
下：

[-->ActivityManagerService.java :
crashApplication]

```
private void crashApplication (ProcessRecord r,
```

```
ApplicationErrorReport.CrashInfo crashInfo) {
    long timeMillis=System.currentTimeMillis() ;
    //从应用进程传递过来的crashInfo中获取相关信息
    String shortMsg=crashInfo.exceptionClassName ;
    String longMsg=crashInfo.exceptionMessage ;
    String stackTrace=crashInfo.stackTrace ;
    if (shortMsg !=null&&longMsg !=null) {
        longMsg=shortMsg+" : "+longMsg ;
    }else if (shortMsg !=null) {
        longMsg=shortMsg ;
    }
    AppErrorHandler result=new AppErrorHandler () ;
    synchronized (this) {
        if (mController !=null) {
            //通知监视者。目前仅MonkeyTest中会为AMS设置监听者。测试过程中可设定是否一检测
            //到App Crash即停止测试。另外，Monkey测试也会将App Crash信息保存起来
            //供开发人员分析
        }
        final long origId=Binder.clearCallingIdentity () ;
        if (r !=null&&r.instrumentationClass !=null) {
            .....//instrumentationClass不为空的情况
        }
        //①调用makeAppCrashingLocked处理，如果返回false，则整个处理流程完毕
        if (r==null||!makeAppCrashingLocked (r, shortMsg, longMsg,
                stackTrace) ) {
            .....return ;
        }
        Message msg=Message.obtain () ;
        msg.what=SHOW_ERROR_MSG ;
        HashMap data=new HashMap () ;
        data.put ("result", result) ;
        data.put ("app", r) ;
        msg.obj=data ;
```

```
//发送SHOW_ERROR_MSG消息给mHandler， 默认处理是弹出一个对话框， 提示
用户某进程
    //崩溃（crash）， 用户可以选择“退出”或“退出并报告”
    mHandler.sendMessage (msg) ;
    .....
} //synchronized (this) 结束
//下面这个get函数是阻塞的， 直到用户处理了对话框为止。注意， 此处涉及两个
线程：
    //handleApplicationCrash函数是在Binder调用线程中处理的， 而对话框则
    是在mHandler所
    //在线程中处理的
    int res=result.get () ;
    Intent appErrorIntent=null;
    synchronized (this) {
        if (r !=null)
            mProcessCrashTimes.put (r.info.processName, r.info.uid,
            SystemClock.uptimeMillis () );
        if (res==AppErrorDialog.FORCE_QUIT_AND_REPORT)
            //createAppErrorIntentLocked返回一个Intent， 该Intent的Action是
            //"android.intent.action.APP_ERROR" ， 指定接收者是
            app.errorReportReceiver
        //成员， 该成员变量在关键点makeAppCrashingLocked中被设置
        appErrorIntent=createAppErrorIntentLocked (r, timeMillis,
        crashInfo) ;
    } //synchronized (this) 结束
    if (appErrorIntent !=null) {
        try{//启动能处理APP_ERROR的应用进程， 目前的源码中还没有地方处理它
            mContext.startActivity (appErrorIntent) ;
        }.....
    }
}
```

以上代码中还有一个关键函数
makeAppCrashingLocked， 其代码如下：

[--> ActivityManagerService.java makeAppCrashingLocked]

:

```
private boolean makeAppCrashingLocked (ProcessRecord app,  
String shortMsg, String longMsg, String stackTrace) {  
    app.crashing=true ;  
    //生成一个错误报告，存放在crashingReport变量中  
    app.crashingReport=generateProcessError (app,  
        ActivityManager.ProcessErrorStateInfo.CRASHED, null,  
        shortMsg,  
        longMsg, stackTrace) ;  
    /*
```

在上边代码中，我们知道系统会通过APP_ERROR Intent启动一个Activity去处理错误报告，

实际上在此之前，系统需要为它找到指定的接收者（如果有）。代码在startAppProblemLocked

中，此处简单描述该函数的处理过程如下：

1. 先查询Settings Secure表中“send_action_app_error”是否使能，如果没有使能，则

不能设置指定接收者

2. 通过PKMS查询安装此Crash应用的应用是否能接收APP_ERROR Intent。必须注意

安装此应用的应用（例如通过“安卓市场”安装了该Crash应用）。如果没有，则转下一步处理

3. 查询系统属性“ro.error.receiver.system.apps”是否指定了APP_ERROR处理器，如果

没有，则转下一步处理

4. 查询系统属性“ro.error.receiver.default”是否指定了默认的处理器

5. 处理者的信息保存在app的errorReportReceiver变量中

另外，如果该Crash应用正好是串行广播发送处理中的一员，则需要结束它的处理流程以

保证后续广播处理能正常执行。读者可参考skipCurrentReceiverLocked函数

*/

```
startAppProblemLocked (app) ;
```

```
app.stopFreezingAllLocked () ;
```

```
//调用handleAppCrashLocked做进一步处理。读者可自行阅读  
return handleAppCrashLocked (app) ;  
}
```

当App的Crash处理完后，事情并未就此结束，因为该应用进程退出后，之前AMS为它设置的讣告接收对象将被唤醒。接下来介绍AppDeathRecipient binderDied的处理流程。

6.7.3 AppDeathRecipient binderDied分析

1.binderDied函数分析

binderDied函数的代码如下：

[-->ActivityManagerService.java :
AppDeathRecipient binderDied]

```
public void binderDied () {  
    //注意，该函数也是通过Binder线程调用的，所以此处要加锁  
    synchronized (ActivityManagerService.this) {  
        appDiedLocked (mApp, mPid, mAppThread) ;  
    }  
}
```

最终的处理函数是appDiedLocked，其中所传递的3个参数保存了对应死亡进程的信息。下面来看appDiedLocked的代码：

[-->ActivityManagerService.java :
appDiedLocked]

```
final void appDiedLocked (ProcessRecord app, int pid,  
IApplicationThread thread) {  
    .....  
    if (app.pid==pid&&app.thread !=null&&  
        app.thread.asBinder () ==thread.asBinder () ) {
```

```
//判断是否需要执行内存调整操作，如果App instrumentationClass不为空，则代表该进程
//是正常启动的，否则有可能是因为进行JUnit测试而启动的，这时就无须进行
//后面的内存调整工作了
boolean doLowMem=app.instrumentationClass==null;
//下面这个函数非常重要
handleAppDiedLocked (app, false, true) ;
if (doLowMem) {
boolean haveBg=false;
//如果系统中还存在oom_adj大于HIDDEN_APP_MIN_ADJ的进程
for (int i=mLruProcesses.size () -1; i>=0 ; i--) {
ProcessRecord rec=mLruProcesses.get (i) ;
if (rec.thread !=null&&rec.setAdj>=
ProcessList.HIDDEN_APP_MIN_ADJ) {
haveBg=true ; //还有后台进程
break ;
}
}//for循环结束
//如果没有后台进程
if (! haveBg) {
long now=SystemClock.uptimeMillis () ;
for (int i=mLruProcesses.size () -1; i>=0 ; i--) {
.....//将这些进程按一定规则加到mProcessesToGc中，尽量保证
//heavy/important/visible/foreground的进程位于mProcessesToGc数
```

组

```
//的前端
};//for循环结束
/*
发送GC_BACKGROUND_PROCESSES_MSG消息给mHandler，该消息的处理过程就
是
```

调用应用进程的scheduleLowMemory或processInBackground函数。其中，
scheduleLowMemory 将触发 onLowMemory 被调用，而
processInBackground将

触发应用进程进行一次垃圾回收

读者可自行阅读该消息的处理函数performAppGcsIfAppropriateLocked

```
*/
```

```
scheduleAppGcsLocked () ;  
} //if (!haveBg) 判断结束  
} //if (doLowMem) 判断结束  
}
```

以上代码中有一个关键函数 handleAppDiedLocked，下面来看它的处理过程。

2. handleAppDiedLocked函数分析

该函数的代码如下：

[-->ActivityManagerService.java :
handleAppDiedLocked]

```
private final void handleAppDiedLocked (ProcessRecord app,  
boolean restarting, boolean allowRestart) {  
    //在本例中，传入的参数为restarting=false, allowRestart=true  
    cleanUpApplicationRecordLocked (app, restarting,  
allowRestart, -1) ;  
    if (!restarting) {  
        mLruProcesses.remove (app) ;  
    }  
    .... //下面还有一部分代码处理和Activity相关的收尾工作，读者可自行阅读  
}
```

重点看上边代码中的 cleanUpApplicationRecordLocked 函数，该函数的主要功能就是处理 Service、ContentProvider 及 BroadcastReceiver 相关的收尾工作。先来看 Service 方面的工作。

(1) cleanUpApplicationRecordLocked之处理Service

这部分的代码如下：

[-->ActivityManagerService.java :
cleanUpApplicationRecordLocked]

```
private final void cleanUpApplicationRecordLocked  
(ProcessRecord app,  
    boolean restarting, boolean allowRestart, int index) {  
    if (index >= 0) mLruProcesses.remove (index) ;  
    mProcessesToGc.remove (app) ;  
    //如果该Crash进程有对应打开的对话框，则关闭它们，这些对话框包括  
    //crash、anr和wait等  
    if (app.crashDialog != null) {  
        app.crashDialog.dismiss () ;  
        app.crashDialog=null;  
    }  
    .....//处理anrDialog、waitDialog  
    //清理app的一些参数  
    app.crashing=false ;  
    app.notResponding=false ;  
    .....  
    // 处理该进程中所驻留的Service或它和别的进程中的Service建立的  
    Connection关系  
    //该函数是AMS Service处理流程中很重要的一环，读者要仔细阅读  
    killServicesLocked (app, allowRestart) ;
```

cleanUpApplicationRecordLocked 函数首先处理 几个 对话框 (dialog) , 然后调用

killServicesLocked函数做相关处理。作为Service流程的一部分，读者需要深入研究。

(2) cleanUpApplicationRecordLocked之处理 ContentProvider

再来看cleanUpApplicationRecordLocked下一阶段的工作，这一阶段的工作主要和ContentProvider有关。

[-->ActivityManagerService.java :
cleanUpApplicationRecordLocked]

```
boolean restart=false;
int NL=mLaunchingProviders.size () ;
if (!app.pubProviders.isEmpty () ) {
//得到该进程中发布的ContentProvider信息
Iterator<ContentProviderRecord>it=
app.pubProviders.values () .iterator () ;
while (it.hasNext () ) {
ContentProviderRecord cpr=it.next () ;
cpr.provider=null;
cpr.proc=null;
int i=0;
if (!app.bad&&allowRestart) {
for (;i<NL ;i++) {
/*
如果有客户端进程在等待这个已经死亡的ContentProvider，则系统会
尝试重新启动它，即设置restart变量为true
*/
if (mLaunchingProviders.get (i) ==cpr) {
restart=true;
```

```
        break ;
    }
}//for循环结束
}else i=NL ;
if (i>=NL) {
/*

```

如果没有客户端进程等待这个ContentProvider，则调用下面这个函数处理它，我们

在卷I的第10章曾提过一个问题，即ContentProvider进程被杀死后，系统该如何处理那些使用了该ContentProvider的客户端进程。例如，Music和

MediaProvider之间有交互，如果杀死了MediaProvider，Music会怎样呢？答案是系统会杀死Music，证据就在removeDyingProviderLocked函数中，读者可自行阅读其内部处理流程

```
/*
removeDyingProviderLocked (app, cpr) ;
NL=mLaunchingProviders.size () ;
}
}//while (it.hasNext ()) 循环结束
app.pubProviders.clear () ;
}
//下面这个函数的功能是检查本进程中的ContentProvider是否存在于
//mLaunchingProviders中，如果存在，则表明有客户端在等待，故需考虑是否重启本进程或者
//杀死客户端（当死亡进程变成bad process的时，需要杀死客户端）
if ( checkAppInLaunchingProvidersLocked ( app, false ) )
restart=true ;
....
```

从以上的描述中可知，ContentProvider所在进程和其客户端进程实际上有着非常紧密而隐晦（之所以说其隐晦，是因为SDK中没有任何说明）的关系。在目前软件开发追求模块间尽量保持松耦合关系的大趋势下，Android 中的

ContentProvider和其客户端这种紧耦合的设计思路似乎不够明智。不过，这种设计是否是不得已而为之呢？读者不妨探讨一下，如果有更合适的解决方案，期待能一起分享。

(3) cleanUpApplicationRecordLocked之处理
BroadcastReceiver这部分的代码如下：

[-->ActivityManagerService.java :
cleanUpApplicationRecordLocked]

```
skipCurrentReceiverLocked (app) ;  
//从AMS中去除接收者  
if (app.receivers.size () >0) {  
Iterator<ReceiverList>it=app.receivers.iterator () ;  
while (it.hasNext ()) {  
removeReceiverLocked (it.next ()) ;  
}  
app.receivers.clear () ;  
}  
if (mBackupTarget !=null&&app.pid==mBackupTarget.app.pid) {  
//处理Backup信息  
}  
mHandler.obtainMessage (DISPATCH_PROCESS_DIED, app.pid,  
app.info.uid, null) .sendToTarget () ;  
//注意该变量名为restarting，前面设置为restart.  
if (restarting) return ;  
if (!app.persistent) {  
mProcessNames.remove (app.processName, app.info.uid) ;  
if (mHeavyWeightProcess==app) {  
.....//处理HeavyWeightProcess的情况}  
}else if (!app.removed) {  
if (mPersistentStartingProcesses.indexOf (app) <0) {
```

```
mPersistentStartingProcesses.add (app) ;
restart=true ;
}
}
mProcessesOnHold.remove (app) ;
if (app==mHomeProcess) mHomeProcess=null ;
if (restart) {//如果需要重启，则调用startProcessLocked处理它
mProcessNames.put (app.processName, app.info.uid, app) ;
startProcessLocked (app, "restart", app.processName) ;
}else if (app.pid>0&&app.pid !=MY_PID) {
synchronized (mPidSelfLocked) {
mPidSelfLocked.remove (app.pid) ;
mHandler.removeMessages (PROC_START_TIMEOUT_MSG, app) ;
}
app.setPid (0) ;
}
}
```

在这段代码中，除了处理BroadcastReceiver方面的工作外，还包括其他方面的收尾工作。最后，如果要重启该应用，则需调用startProcessLocked函数进行处理。这部分代码不再详述，读者可自行阅读。

6.7.4 App的Crash处理总结

分析完整个处理流程，有些读者或许会咋舌。应用进程的诞生是一件很麻烦的事情，没想到应用进程的善后工作居然也很费事，希望各个应用进程能“活得”更稳健点儿。

图6-24展示了应用进程Crash后的处理的流程。

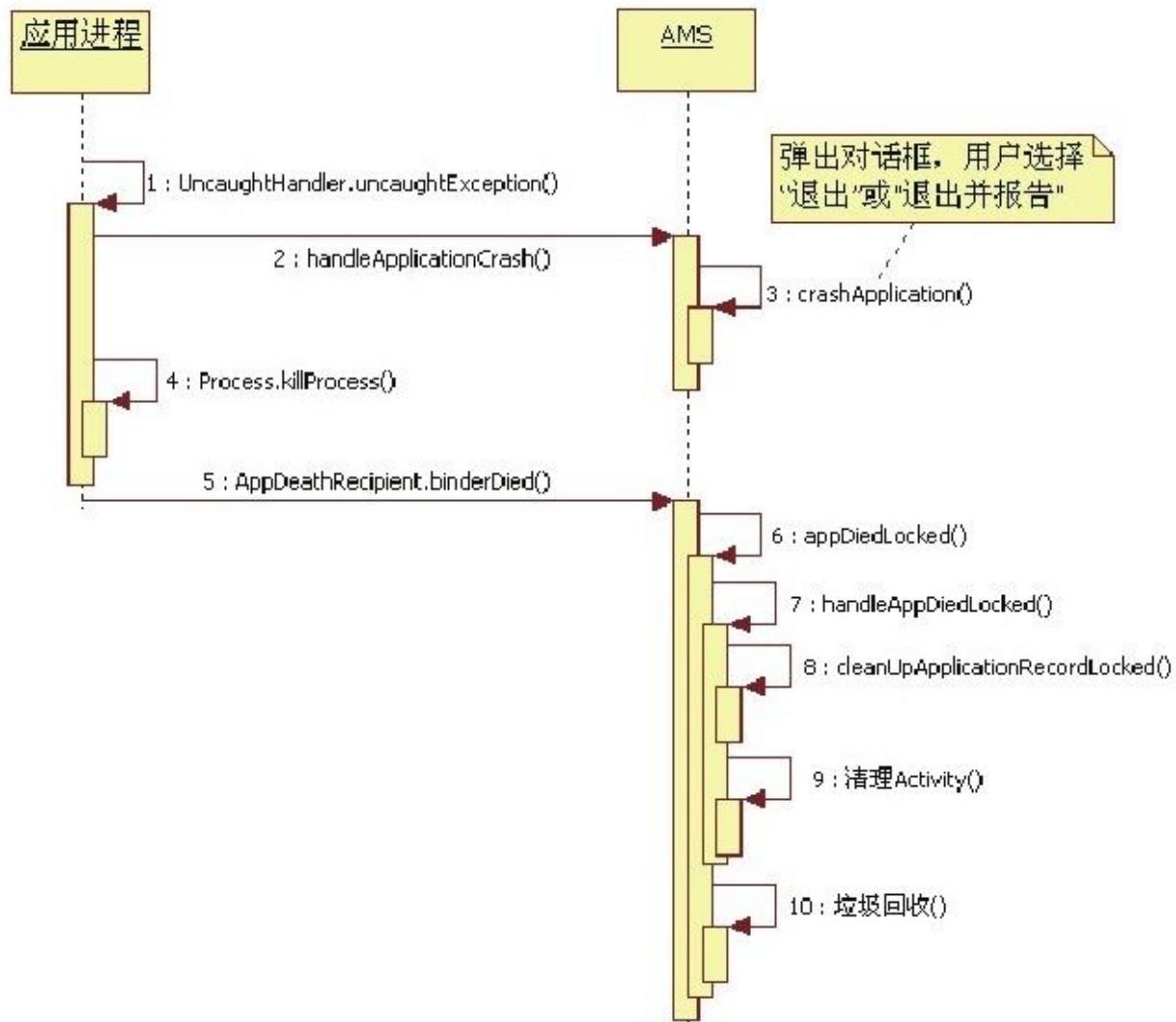


图 6-24 应用进程的Crash处理流程

6.8 本章学习指导

本章内容较为复杂，即使用了这么长的篇幅来讲解AMS，依然只能覆盖其中一部分内容。读者在阅读本章时，一定要注意文中的分析脉络，以搞清楚流程为主旨。以下是本章的思路总结：

首先搞清楚AMS初创时期的一系列流程，这对理解Android运行环境和系统启动流程等很有帮助。

搞清楚一个最简单的情形下Activity启动所历经的“磨难”。这部分流程最复杂，建议读者在搞清书中所阐述内容的前提下结合具体问题进行分析。

BroadcastReceiver的处理流程相对简单。读者务必理解AMS发送广播的处理流程，这对实际工作非常有帮助。例如最近在处理一个广播时，发现静态注册的广播接收者收到广播的时间较长，研究了AMS广播发送的流程后，将其改成了动态注册，结果响应速度就快了很多。

关于Service的处理流程希望读者根据流程图自行分析和研究。

AMS中的进程管理这一部分内容最难看懂。此处有多方面的原因，笔者觉得和缺乏相关说明有重要关系。建议读者只了解AMS进程管理的大概面貌即可。另外，建议读者不要试图通过修改这部分代码来优化Android的运行效率。进程调度规则向来比较复杂，只有在大量实验的基础上才能得到一个合适的模型。

AMS在处理应用进程的Crash及死亡的工作上也是不遗余力的。这部分工作相对比较简单，相信读者能轻松掌握。

由于精力和篇幅的原因，AMS中还有很多精彩的内容未能涉及，建议读者在本章学习基础上，根据具体情况继续深入研究。

6.9 本章小结

本章对AMS进行了有针对性的分析：

分析了AMS的创建及初始化过程。

以启动一个Activity为例，分析了Activity的启动，应用进程的创建等一系列比较复杂的处理流程。

介绍了广播接收者及广播发送的处理流程。Service的处理流程读者可根据流程图并结合代码自行开展研究。

还介绍了AMS中的进程管理及相关的知识。重点是在AMS中对LRU、oom_adj及scheduleGroup的调整。

最后介绍了应用进程Crash及死亡后，AMS的处理流程。

第7章 深入理解ContentProvider

本章主要内容：

深入分析ContentProvider的创建和启动，以及SQLite相关的知识点。

深入分析Cursor query和close函数的实现。

深入分析ContentResolver openAssetFileDescriptor函数的实现。

本章所涉及的源代码文件名及位置：

ActivityManagerService. java
(frameworks/base/services/java/com/android/server/am/ActivityManagerService.java)

ContextImpl. java
(frameworks/base/core/java/android/app/ContextImpl.java)

ActivityThread. java
(frameworks/base/core/java/android/app/ActivityThread.java)

MediaStore. java
(frameworks/base/core/java/android/provider/Media

Store.java)

ContentResolver. java
(frameworks/base/core/java/android/content/Content
Resolver.java)

ContentProvider. java
(frameworks/base/core/java/android/content/Content
Provider.java)

MediaProvider. java
(package/providers/MediaProvider/src/java/com/and
roid/MediaProvider/MediaProvider.java)

SQLiteDatabase. java
(frameworks/base/core/java/android/database/sqlite/
SQLiteDatabase.java)

SQLiteCompiledSql. java
(frameworks/base/core/java/android/database/sqlite/
SQLiteCompiledSql.java)

android_database_SQLiteDatabase. cpp
(frameworks/base/core/jni/android_database_SQLIt
eDatabase.cpp)

android_database_SQLiteCompiledSql. cpp
(frameworks/base/core/jni/android_database_SQLIt

e-Compiled-Sql.cpp)

sqlite3_android.
(external/sqlite3/android/sqlite3_android.cpp) cpp

SQLiteQueryBuilder.java
(frameworks/base/core/java/android/database/sqlite/
SQLiteQueryBuilder.java)

SQLiteCursorDriver.java
(frameworks/base/core/java/android/database/sqlite/
SQLiteCursorDriver.java)

SQLiteQuery.java
(frameworks/base/core/java/android/database/sqlite/
SQLiteQuery.java)

SQLiteCursor.java
(frameworks/base/core/java/android/database/sqlite/
SQLiteCursor.java)

SQLiteProgram.java
(frameworks/base/core/java/android/database/sqlite/
SQLiteProgram.java)

CursorToBulkCursorAdaptor.java
(frameworks/base/core/java/android/database/CursorToBulkCursorAdaptor.java)

BulkCursorToCursorAdaptor.java
(frameworks/base/core/java/android/database/BulkCursorToCursorAdaptor.java)

CursorWindow. java
 (frameworks/base/core/java/android/database/CursorWindow.java)

android_database_CursorWindow. cpp
 (frameworks/base/core/jni/android_database_Cursor
 Window.cpp)

CursorWindow.
(frameworks/base/libs/binder/CursorWindow.cpp) cpp

android_database_SQLiteQuery. cpp
 (frameworks/base/core/jni/android_database_SQLIt
 eQuery.cpp)

CursorWrapper.java
(frameworks/base/core/java/android/database/CursorWrapper.java)

AbstractCursor.java
(frameworks/base/core/java/android/database/AbstractCursor.java)

BulkCursorNative. java
(frameworks/base/core/java/android/database/BulkC
ursorNative.java)

ParcelFileDescriptor. java
(frameworks/base/core/java/android/os/ParcelFileDe
scriptor.java)

MediaProvider. java
(packages/providers/MediaProvider/src/com/android
/providers/media/MediaProvider.java)

android_util_Binder. cpp
(frameworks/base/core/jni/android_util_Binder.cpp
)

Parcel. cpp
(frameworks/base/libs/binder/Parcel.cpp)

binder. c
(kernel/drivers/staging/android/binder.c)

7.1 概述

本章重点分析ContentProvider、SQLite、Cursor query、close函数的实现及ContentResolver openAssetFileDescriptor函数。为了帮助读者进一步理解本章的知识点，笔者特意挑选了4条分析路线。

第一条：以客户端进程通过MediaStore.Images.Media类的静态函数query来查询MediaProvider中Image相关信息为入口点，分析系统如何创建和启动MediaProvider。此分析路线着重关注客户端进程、ActivityManagerService及MediaProvider所在进程间的交互。

第二条：沿袭第一条分析路径，但是将关注焦点转移到SQLiteDatabase如何创建数据库的分析上。另外，本条路线还将对SQLite进行相关介绍。

第三条：重点研究Cursor query和close函数的实现细节。

第四条：分析ContentResolver openAssetFileDescriptor函数的实现。

闲话少说，立即开始本次分析之旅。

7.2 MediaProvider的启动及创建

第一、二、三条分析路线都将以下面这段示例为参考。

[-->MediaProvider客户端示例]

```
void QueryImage (Context context) {  
    //①得到ContentResolver对象  
    ContentResolver cr=context.getContentResolver () ;  
    Uri uri=MediaStore.Images.Media.EXTERNAL_CONTENT_URI ;  
    //②查询数据库  
    Cursor cursor=MediaStore.Images.Media.query ( cr,     uri,  
null) ;  
    cursor.moveToFirst () ;//③移动游标到头部  
.....//从游标中取出数据集  
    cursor.close () ;//④关闭游标  
}
```

先介绍一下这段示例的情况：客户端（即运行本示例的进程）查询（query）的目标ContentProvider是MediaProvider，它运行于进程android.process.media中。假设目标进程此时还未启动。

本节的关注点集中在：

MediaProvider所在进程是如何创建的？
MediaProvider又是如何创建的？

客户端通过什么和位于目标进程中的
MediaProvider交互的？

先来看第一个关键函数getContentResolver。

7.2.1 Context的getContentResolver函数分析

根据第6章对Context的介绍，Context的
getContentResolver最终会调用它所代理的
ContextImpl对象的getContentResolver函数，此处
直接看ContextImpl的代码。

[-->ContextImpl.java : getContentResolver]

```
public ContentResolver getContentResolver () {  
    return mContentResolver ;  
}
```

该函数直接返回mContentResolver，此变量在
ContextImpl初始化时创建，相关代码如下：

[-->ContextImpl.java : init]

```
    final void init ( LoadedApk packageInfo, IBinder
activityToken,
        ActivityThread mainThread, Resources container, String
packageName) {
    .....
    mMainThread=mainThread ; //mainThread指向ActivityThread对象
    //mContentResolver的真实类型是ApplicationContentResolver
    mContentResolver=new ApplicationContentResolver ( this,
mainThread) ;
    .....
}
```

由以上代码可知，`mContentResolver`的真实类型是`ApplicationContentResolver`，它是`ContextImpl`定义的内部类并继承了`ContentResolver`。

`getContentResolver`函数比较简单，就分析到此。下面来看第二个关键函数。

提示 为了书写方便，后面将`ContentProvider`简称为CP，将`ContentResolver`简称为CR。

7.2.2 MediaStore.Image.Media的query函数分析

第二个关键点是在MediaProvider客户端示例中所调用的MediaStore.Image.Media的query函数。MediaStore是多媒体开发中常用的类，其

内部定义了专门针对Image、Audio、Video等不同多媒体信息的内部类来帮助客户端开发人员更好地和MediaProvider交互。这些类及相互之间的关系如图7-1所示。

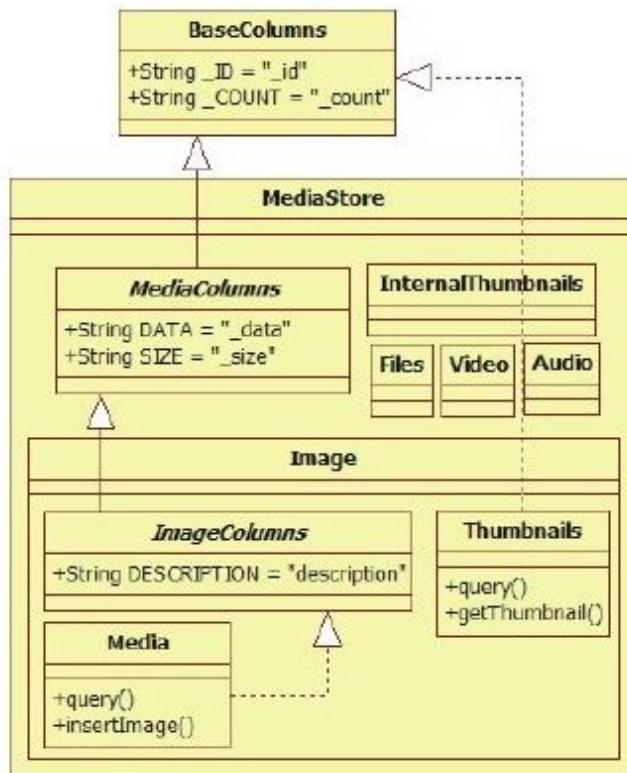


图 7-1 MediaStore类图

由图7-1可知，MediaStore定义了较多的内部类，我们重点展示作为内部类之一的Image的情况，其中：

MediaColumns定义了所有与媒体相关的数据库表都会用到的数据库字段，而ImageColumns定义了单独针对Image的数据库字段。

Image类定义了一个名为Media的内部类用于查询和Image相关的信息，同时Image类还定义了一个名为Thumbnails的内部类用于查询和Image相关的缩略图的信息（在Android平台上，缩略图的来源有两种，一种是Image，另一种是Video，故Image定义了名为Thumbnails的内部类，而Video也定义了一个名为Thumbnails的内部类）。

提示 MediaStore类较为复杂，主要原因是它定义了一些同名类。读者阅读代码时务必仔细。

下面来看Image.Media的query函数，其代码非常简单，如下所示：

[-->MediaStore.java : Image.Media.query]

```
public static final class Media implements ImageColumns{
    public static final Cursor query (ContentResolver cr, Uri
uri,
```

```
String[]projection) {  
    //直接调用ContentResolver的query函数  
    return cr.query (uri, projection, null,  
    null, DEFAULT_SORT_ORDER) ;  
}
```

Image.Media的query函数直接调用CR的query函数，虽然CR的真实类型是Application-ContentResolver，但是此函数却由其基类CR实现。

提示 追求运行效率的程序员也许会对上边这段代码的实现颇有微词，因为Image.Media的query函数基本上没做任何有意义的工作。相比客户端直接调用cr.query函数，此处的query就增加了一次函数调用和返回的开销（即Image.Media query调用和返回时参数的入栈/出栈）。但是，通过Image.Media的封装将使程序更清晰和易读（与直接使用CR的query相比，代码阅读者一看Image.Media就知道其query函数应该和Image有关，否则需要通过解析uri参数才能确定查询的信息是什么）。代码清晰易读和运行效率高，往往是软件开发中的熊掌和鱼，它们之间的对立性，将在本章中体现得淋漓尽致。笔者建议读者在实际开发中结合具体情况决定取舍，万不可钻牛角尖。

1.ContentResolver的query函数分析

这部分的代码如下：

[-->ContentResolver.java : query]

```
public final Cursor query (Uri uri, String[]projection,  
String selection, String[]selectionArgs, String sortOrder) {  
    //调用acquireProvider函数，参数为uri，函数也由CR实现  
    IContentProvider provider=acquireProvider (uri) ;  
    //注意，下面将和CP交互，相关知识留待7.4节再分析  
    .....  
}
```

CR的query将调用acquireProvider，该函数定义在CR类中，代码如下：

[-->ContentResolver.java : query]

```
public final IContentProvider acquireProvider (Uri uri) {  
    if ( ! SCHEME_CONTENT.equals ( uri.getScheme () ) ) return  
null;  
    String auth=uri.getAuthority () ;  
    if (auth !=null) {  
        //acquireProvider是一个抽象函数，由ContentResolver的子类实现。在  
        //本例中，该函数  
        //将由ApplicationContentResolver实现。uri.getAuthority将返回代  
        //表目标  
        //CP的名字  
        return acquireProvider (mContext, uri.getAuthority () ) ;  
    }  
    return null;  
}
```

如上所述，acquireProvider由CR的子类实现，在本例中该函数由ApplicationContent-Resolver定义，代码如下：

[-->ContextImpl.java : acquireProvider]

```
protected IContentProvider acquireProvider (Context context,
String name) {
    //mMainThread指向代表应用进程主线程的ActivityThread对象，每个应用
    //进程只有一个
    //ActivityThread对象
    return mMainThread.acquireProvider (context, name) ;
}
```

如以上代码所示，最终ActivityThread的acquireProvider函数将被调用，希望它不要再被层层转包了。

2. ActivityThread的acquireProvider函数分析

ActivityThread的acquireProvider函数的代码如下：

[-->ActivityThread.java : acquireProvider]

```
public final IContentProvider acquireProvider (Context c,
String name) {
    //①调用getProvider函数，它很重要，具体见下文分析
    IContentProvider provider=getProvider (c, name) ;
    .....
    IBinder jBinder=provider.asBinder () ;
```

```
synchronized (mProviderMap) {  
    //客户端进程将本进程使用的CP信息保存到mProviderRefCountMap中，  
    //其主要功能与引用计数和资源释放有关，读者暂可不理会它  
    ProviderRefCount prc=mProviderRefCountMap.get (jBinder) ;  
    if (prc==null)  
        mProviderRefCountMap.put  ( jBinder,  new  ProviderRefCount  
(1) ) ;  
    else prc.count++ ;  
}  
return provider ;  
}
```

在acquireProvider内部调用getProvider得到一个IContentProvider类型的对象，该函数非常重要，其代码为：

[-->ActivityThread.java : getProvider]

```
private  IContentProvider  getProvider  ( Context  context,  
String name) {  
    /*  
     * 查询该应用进程是否已经保存了用于和远端CP通信的对象existing。  
     * 此处，我们知道 existing 的类型是 IContentProvider，不过  
     * IContentProvider是一个interface，那么existing的真实类型是什么呢？稍后再揭示  
     */  
    IContentProvider   existing=getExistingProvider  ( context,  
name) ;  
    if (existing !=null) return existing ;//如果existing存在，则直接  
    //返回  
    IActivityManager.ContentProviderHolder holder=null ;  
    try{  
        //如果existing不存在，则需要向AMS查询，返回值的类型为  
        ContentProviderHolder
```

```
holder=ActivityManagerNative.getDefault()
() .getContentProvider (
    getApplicationThread () , name) ;
}.....
//注意：记住下面这个函数调用，此时是在客户端进程中
IContentProvider prov=installProvider (      context,
holder.provider,
holder.info, true) ;
.....
return prov ;
}
```

以上代码中让人比较头疼的是其中新出现的几种数据类型，如 IContentProvider、ContentProviderHolder。先来分析 AMS 的 getContentProvider。

3.AMS的getContentProvider函数分析

getContentProvider 的功能主要由 getContentProviderImpl 函数实现，故此处可直接对它进行分析。

(1) getContentProviderImpl启动目标进程

getContentProviderImpl 函数较长，可分段来看，先来分析下面一段代码。

[-->ActivityManagerService.java :
getContentProviderImpl]

```
private final ContentProviderHolder getContentProviderImpl ( IApplicationThread caller, String name) {  
    ContentProviderRecord cpr ;  
    ProviderInfo cpi=null ;  
    synchronized (this) {  
        ProcessRecord r=null ;  
        if (caller !=null) {  
            r=getRecordForAppLocked (caller) ;  
            if (r==null) .....//如果查询不到调用者信息，则抛出SecurityException  
        } //if (caller !=null) 判断结束  
        //name参数为调用进程指定的代表目标CP的authority  
        cpr=mProvidersByName .get (name) ;  
        //如果cpr不为空，表明该CP已经在AMS中注册  
        boolean providerRunning=cpr !=null ;  
        if (providerRunning) {  
            .....//如果该CP已经存在，则进行对应处理，相关内容可自行阅读  
        }  
        //如果目标CP对应进程还未启动  
        if (! providerRunning) {  
            try{  
                //查询PKMS，得到指定的ProviderInfo信息  
                cpi=AppGlobals.getPackageManager () .resolveContentProvider ( name, STOCK_PM_FLAGS |  
                    PackageManager .GET_URI_PERMISSION_PATTERNS) ;  
            }.....  
            String msg ;  
            //权限检查，此处不作讨论  
            if ( (msg=checkContentProviderPermissionLocked (cpi, r) ) !=  
                null)  
                throw new SecurityException (msg) ;  
            /*  
             * 如果system_server还没启动完毕，并且该CP不运行在system_server中，则  
             * 此时不允许启动CP。  
             * 读者还记得哪个CP运行在system_server进程中吗？答案是  
             * SettingsProvider  
             */
```

```
.....  
    ComponentName comp=new ComponentName ( cpi.packageName,  
cpi.name) ;  
    cpr=mProvidersByClass.get (comp) ;  
    final boolean firstClass=cpr==null;  
    //初次启动MediaProvider对应进程时，firstClass一定为true  
    if (firstClass) {  
        try{  
            //查询PKMS，得到MediaProvider所在的Application信息  
            ApplicationInfo ai=  
                AppGlobals.getPackageManager () .getApplicationInfo (  
cpi.applicationInfo.packageName, STOCK_PM_FLAGS) ;  
            if (ai==null) return null;  
            //在AMS内部通过ContentProviderRecord来保存CP的信息，类似  
//ActivityRecord、BroadcastRecord等  
            cpr=new ContentProviderRecord (cpi, ai, comp) ;  
        }.....  
    } //if (firstClass) 判断结束
```

以上代码的逻辑比较简单，主要是为目标CP（即 MediaProvider）创建一个 ContentProviderRecord对象。结合第6章的知识，AMS为四大组件都设计了对应的数据结构，如 ActivityRecord、BroadcastRecord等。

接着看 getContentViewerImpl，其下一步的工作就是启动目标进程，代码如下：

[-->ActivityManagerService.java :
getContentViewerImpl]

```
/*
```

canRunHere函数用于判断目标CP能否运行在前面定义的变量r所对应的进程（即调用者所在进程）

该函数内部做如下判断：

```
( info.multiprocess||info.processName.equals  
(app.processName) )  
&& (uid==Process.SYSTEM_UID||uid==app.info.uid)  
就本例而言，MediaProvider不能运行在客户端进程中  
*/  
if (r !=null&&cpr.canRunHere (r) ) return cpr ;  
final int N=mLaunchingProviders.size () ;  
.....//查找目标CP对应的进程是否正处于启动状态  
//如果i大于等于N，表明目标进程的信息不在mLaunchingProviders中  
if (i>=N) {  
final long origId=Binder.clearCallingIdentity () ;  
.....  
//调用startProcessLocked函数创建目标进程  
ProcessRecord proc=startProcessLocked (cpi.processName,  
cpr.appInfo, false, 0, "content provider",  
new ComponentName (cpi.applicationInfo.packageName,  
cpi.name) , false) ;  
if (proc==null) return null ;  
cpr.launchingApp=proc ;  
//将该进程信息保存到mLaunchingProviders中  
mLaunchingProviders.add (cpr) ;  
}  
if (firstClass) mProvidersByClass.put (comp, cpr) ;  
mProvidersByName.put (name, cpr) ;  
/*  
下面这个函数将为客户端进程和目标CP进程建立紧密的关系，即当目标CP进程死  
亡后，  
AMS将根据该函数建立的关系找到客户端进程并杀死（kill）它们。在7.2.3节  
有对这个函数的相关解释  
*/  
incProviderCount (r, cpr) ;  
if (cpr.launchingApp==null) return null ;  
try{
```

```
cpr.wait () ;//等待目前进程的启动  
}.....  
}//synchronized (this) 结束  
return cpr ;  
}
```

通过对以上代码的分析发现，getContentProviderImpl将等待一个事件，想必读者也能明白，此处一定是在等待目标进程启动并创建好MediaProvider。目标进程的这部分工作用专业词语来表达就是发布（publish）目标ContentProvider（即本例的MediaProvider）。

（2）MediaProvider的创建

根据第6章的介绍，目标进程启动后要做的第一件大事就是调用AMS的attachApplication函数，该函数的主要功能由attachApplicationLocked完成。我们回顾一下相关代码。

[-->ActivityManagerService.java :
attachApplicationLocked]

```
private      final      boolean      attachApplicationLocked  
(IApplicationThread thread,  
 int pid) {  
     .....  
     //通过PKMS查询运行在该进程中的CP信息，并保存到mProvidersByClass中  
     List providers=normalMode?  
     generateApplicationProvidersLocked (app) :null ;
```

```
//调用目标应用进程的bindApplication函数，此处将providers信息传递给  
目标进程  
    thread.bindApplication (processName, appInfo, providers,  
    app.instrumentationClass, profileFile,  
    ....) ;  
    ....  
}
```

再来看目标进程bindApplication的实现，其内部最终会通过handleBindApplication函数处理，我们回顾一下相关代码。

[-->ActivityThread.java :
handleBindApplication]

```
private void handleBindApplication (AppBindData data) {  
    ....  
    if (! data.restrictedBackupMode) {  
        List<ProviderInfo>providers=data.providers ;  
        if (providers !=null) {  
            //调用installContentProviders安装此ContentProvider  
            installContentProviders (app, providers) ;  
            ....  
        }  
    }  
    ....  
}
```

AMS传递过来的ProviderInfo列表将由目标进程的installContentProviders处理，其相关代码如下：

[-->ActivityThread.java
installContentProviders]

```
private void installContentProviders (Context context,  
List<ProviderInfo>providers) {  
final     ArrayList<IActivityManager.ContentProviderHolder>  
results=  
new ArrayList<IActivityManager.ContentProviderHolder> () ;  
Iterator<ProviderInfo>i=providers.iterator () ;  
while (i.hasNext ()) {  
//①调用installProvider，注意该函数传递的第二个参数为null  
IContentProvider cp=installProvider ( context, null, cpi,  
false) ;  
if (cp !=null) {  
IActivityManager.ContentProviderHolder cph=  
new IActivityManager.ContentProviderHolder (cpi) ;  
cph.provider=cp ;  
results.add (cph) ;//将信息添加到results数组中  
.....//创建引用计数  
}  
}  
}//while循环结束  
try{//②调用AMS的publishContentProviders发布ContentProviders  
ActivityManagerNative.getDefault () .publishContentProviders  
(  
getApplicationThread () , results) ;  
}.....  
}
```

以上代码列出了两个关键点，分别是：

调用 installProvider 得到一个 IContentProvider 类型的对象。

调用AMS的publishContentProviders发布本进程所运行的CP。该函数留到后面再作分析。

在继续分析之前，笔者要特别强调installProvider，该函数既在客户端进程中被调用（还记得7.2.2节ActivityThread的acquireProvider函数中那句注释吗？），又在目标进程（即此处MediaProvider所在进程）中被调用。与客户端进程的调用相比，只在一处有明显的不同：

客户端进程调用installProvider函数时，该函数的第二个参数不为null。

目标进程调用installProvider函数时，该函数的第二个参数硬编码为null。

我们曾经在6.2.3分析过installProvider函数，结合那里的介绍可知：installProvider是一个通用函数，不论客户端使用远端的CP还是目标进程安装运行在其上的CP，最终都会调用它，只不过参数不同罢了。

下面来看installProvider函数，其代码如下：

[-->ActivityThread.java : installProvider]

```
private IContentProvider installProvider (Context context,
    IContentProvider provider, ProviderInfo info, boolean
noisy) {
```

```
ContentProvider localProvider=null;
if (provider==null) {//针对目标进程的情况
Context c=null;
ApplicationInfo ai=info.applicationInfo;
if (context.getPackageName () .equals (ai.packageName) ) {
c=context;
}.....//这部分代码已经在6.2.3节分析过了，其目的就是为了得到正确的
//Context用于加载Java字节码
try{
final java.lang.ClassLoader cl=c.getClassLoader () ;
//通过Java反射机制创建MediaProvider实例
localProvider= (ContentProvider) cl.
loadClass (info.name) .newInstance () ;
//注意下面这句代码
provider=localProvider.getIContentProvider () ;
}
}else if (localLOGV) {
Slog.v (TAG , "Installing external
provider"+info.authority+" : "
+info.name) ;
}//if (provider==null) 判断结束
/*

```

由以上代码可知，对于provider不为null的情况（即客户端调用的情况），该函数没有

什么特殊的处理

```
*/
```

```
.....
```

```
/*
```

引用计数及设置DeathRecipient等相关操作。在6.2.3节中曾介绍过，目标进程为自己

进程中的CP实例设置DeathRecipient没有作用，因为二者在同一个进程中，自己怎么能

接收自己的讣告消息呢？不过，如果客户端进程为目标进程的CP设置DeathRecipient又

有作用吗？请读者仔细思考这个问题

```
*/
```

```
    return provider; //最终返回的对象是IContentProvider类型的，它到底  
    是什么呢？  
}
```

由以代码可知，installProvider最终返回的是一个IContentProvider类型的对象。对于目标进程而言，该对象是通过调用CP的实例对象的（本例就是MediaProvider）getIContentProvider函数得来的。而对于客户端进程而言，该对象是由installProvider第二个参数传递进来的，那么，这个IContentProvider到底是什么？

(3) IContentProvider的真面目

要说清楚 IContentProvider，就要先来看ContentProvider家族的类图，如图7-2所示。

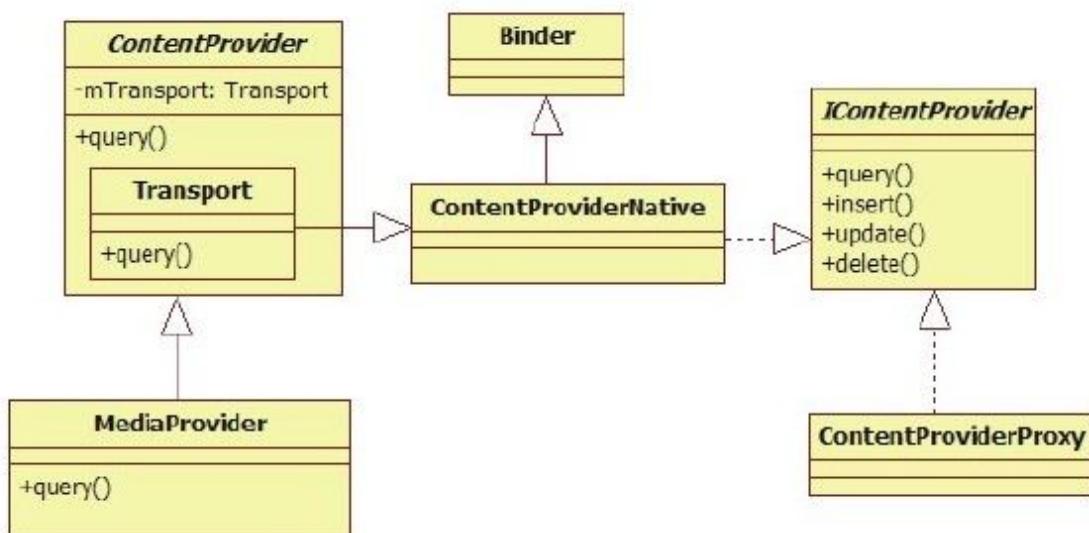


图 7-2 ContentProvider类图

图7-2揭示了IContentProvider的真面目，具体介绍如下：

每个CP实例中都有一个mTransport成员，其类型为Transport。

Transport类从ContentProviderNative派生。由图7-2可知，ContentProviderNative从Binder类派生，并实现了IContentProvider接口。结合前面的代码，IContentProvider将是客户端进程和目标进程交互的接口，即目标进程使用IContentProvider的Bn端Transport，而客户端使用IContentProvider的Bp端，其类型是ContentProviderProxy（定义在ContentProviderNative.java中）。

客户端如何通过IContentProvider query函数和目标CP进程进行交互？其流程如下：

CP客户端得到IContentProvider的Bp端（实际类型是ContentProviderProxy），并调用其query函数，在该函数内部将参数信息打包，传递给Transport（它是IContentProvider的Bn端）。

Transport的onTransact函数将调用Transport的query函数，而Transport的query函数又将调用CP子类定义的query函数（即MediaProvider的query函数）。

关于目标进程这一系列的调用函数，不妨先看看Transport的query函数，其代码为：

[-->ContentProvider.java : Transport.query]

```
public Cursor query (Uri uri, String[]projection,  
String selection, String[]selectionArgs, String sortOrder) {  
    enforceReadPermission (uri) ;  
    //Transport为ContentProvider的内部类，此处将调用ContentProvider  
    的query函数  
    //本例中，该query函数由MediaProvider实现，故最终会调用  
    MediaProvider的query  
    return ContentProvider.this.query ( uri, projection,  
    selection,  
    selectionArgs, sortOrder) ;  
}
```

读者务必要弄清楚，此处只有一个目标CP的实例，即只有一个MediaProvider对象。Transport的query函数内部虽然调用的是基类CP的query函数，但是根据面向对象的多态原理，该函数最终由其子类（本例中是MediaProvider）来实现。

认识了IContentProvider，即知道了客户端进程和目标进程的交互接口。

继续我们的分析。此时目标进程需要将MediaProvider的信息通过AMS发布出去。

(4) AMS pulishContentProviders分析

要把目标进程的CP信息发布出去，需借助AMS的publishContentProviders函数，其代码如下：

[-->ActivityManagerService.java :
publishContentProviders]

```
public final void publishContentProviders  
(IAplicationThread caller,  
List<ContentProviderHolder>providers) {  
....  
synchronized (this) {  
final ProcessRecord r=getRecordForAppLocked (caller) ;  
final long origId=Binder.clearCallingIdentity () ;  
final int N=providers.size () ;  
for (int i=0 ;i<N ;i++) {  
ContentProviderHolder src=providers.get (i) ;  
ContentProviderRecord dst=r.pubProviders.get  
(src.info.name) ;  
if (dst !=null) {  
....//将相关信息分别保存到mProviderByClass和mProvidersByName中  
int NL=mLaunchingProviders.size () ;  
....//目标进程已经启动，将其从mLaunchingProviders中移除  
synchronized (dst) {  
dst.provider=src.provider ;//将信息保存在dst中  
dst.proc=r ;  
//触发还在 (wait) getContentProvider中的那个客户端进程  
dst.notifyAll () ;  
}  
updateOomAdjLocked (r) ;//调节目标进程的oom_adj等相关参数  
} //if (dst !=null) 判断结束  
....  
}  
}
```

至此，客户端进程将从getServiceProvider中返回，并调用installProvider函数。根据前面的分析，客户端进程调用installProvider时，其第二个参数不为null，即客户端进程已经从AMS中得到了能直接和目标进程交互的IContentProvider Bp端对象。此后，客户端就可

直接使用该对象向目标进程发起请求。

7.2.3 MediaProvider的启动及创建总结

回顾一下整个MediaProvider的启动和创建流程，如图7-3所示。

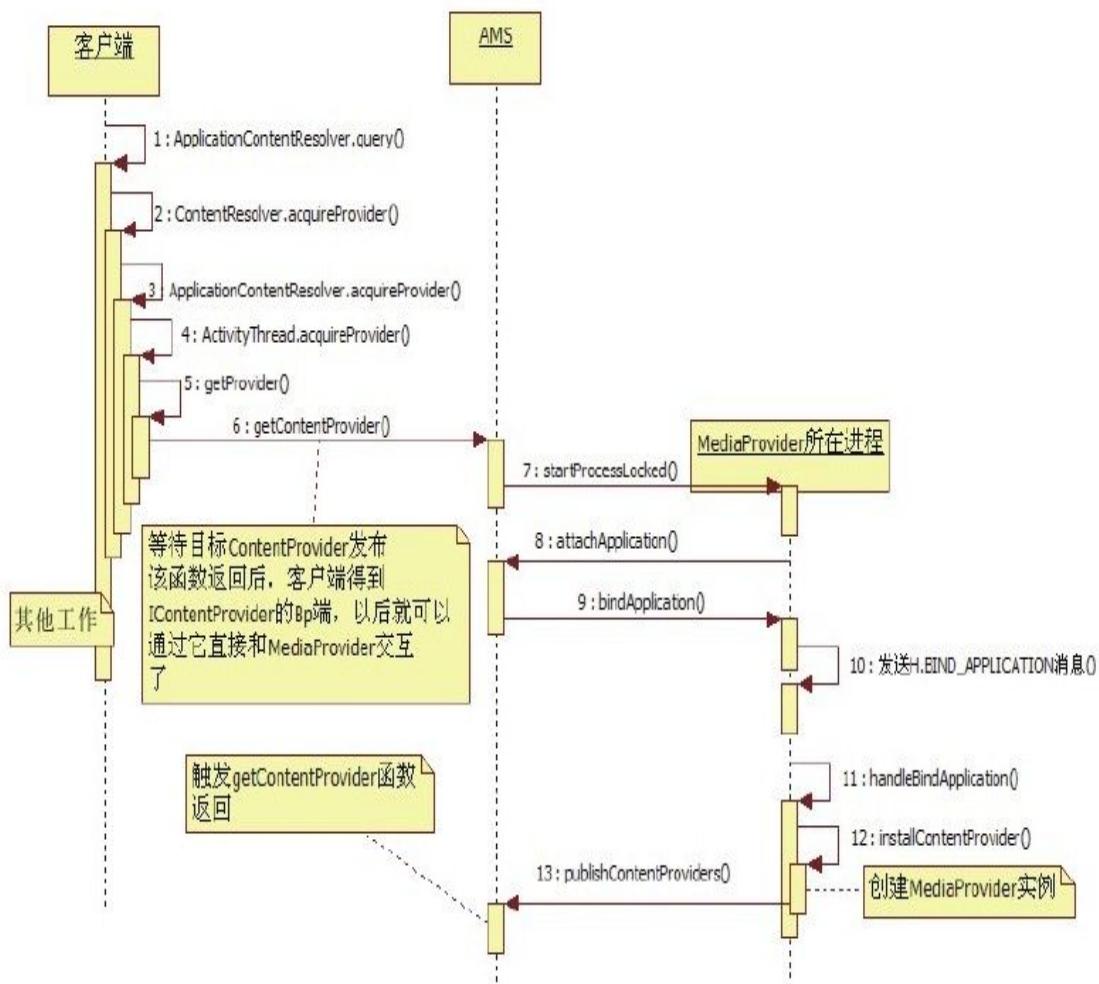


图 7-3 MediaProvider的启动和创建流程

整个流程相对比较简单。读者在分析时只要注意installProvider这个函数在目标进程和客户端

进程中被调用时的区别即可。这里再次强调：

目标进程调用installProvider时，传递的第二个参数为null，使内部通过Java反射机制真正创建目标CP实例。

客户端调用installProvider时，其第二个参数已经通过查询AMS得到。该函数真正的工作只不过是引用计数控制和设置讣告接收对象罢了。

至此，客户端进程和目标进程通信的通道IContentProvider已经登场。除此之外，客户端进程和目标CP还建立了非常紧密的关系，这种关系造成的后果就是一旦目标CP进程死亡，AMS会杀死与之有关的客户端进程。不妨回顾一下与之相关知识点：

该关系的建立是在AMS getContentTypeImpl 函数中调用incProviderCount 完成的，关系的确立以ContentProviderRecorder 保存客户端进程的ProcessRecord信息为标志。

一旦CP进程死亡，AMS能根据该ContentProviderRecorder中保存的客户端信息找到使用该CP的所有客户端进程，然后再杀死它们。

客户端能否撤销这种紧密关系呢？答案是肯定的，但这和Cursor是否关闭有关。这里先简单描述一下流程：

当 Cursor 关闭时，ContextImpl 的 releaseProvider会被调用。根据前面的介绍，它最终会调用ActivityThread的releaseProvider函数。

ActivityThread 的 releaseProvider 函数会导致 completeRemoveProvider被调用，在其内部根据该 CP 的引用计数判断是否需要调用 AMS 的 removeContentProvider。

通过 AMS 的removeContentProvider将删除对应ContentProviderRecord中此客户端进程的信息，这样一来，客户端进程和目标CP进程的紧密关系就荡然无存了。至此，本章第一条分析路线就介绍完毕了。

提示 读者可能觉得，这条路线是对第6章的补充和延续。不过，虽然目标进程由AMS创建和启动，而且CP的发布也需和AMS交互，但是对于CP来说，我们更关注客户端和目标进程中CP实例间的交互。事实上，客户端得到IContentProvider Bp端对象后，即可直接与目标进程的CP实例交互，也就无须借助AMS了，所以笔者将这条路线放到了本章进行分析。

7.3 SQLite创建数据库分析

作为Android多媒体系统中媒体信息的仓库，MediaProvider使用了SQLite数据库来管理系统中多媒体相关的数据信息。作为第二条分析路线，本节的目标是分析MediaProvider如何利用SQLite创建数据库，同时还将介绍和SQLite相关的一些知识点。

下面先来看大名鼎鼎的SQLite及Java层的SQLiteDatabase家族。

7.3.1 SQLite及SQLiteDatabase家族

1.SQLite轻装上阵

SQLite是一个轻量级的数据库，它和笔者之前接触的SQLServer或Oracle DB比起来，犹如蚂蚁和大象的区别。它“轻”到什么程度呢？笔者总结了SQLite具有的两个特点：

从代码上看，SQLite所有的功能都实现在Sqlite3.c中，而头文件Sqlite3.h定义了它所支持的

API。其中，Sqlite3.c文件包含12万行左右的代码，相当于一个中等偏小规模的程序。

从使用者角度的来说，SQLite编译完成后将生成一个libsqlite.so，大小仅为300KB左右。SQLite确实够轻，但这个“轻”还不是本节标题“轻装上阵”一词中的“轻”。为什么？此处先向读者们介绍一个直接使用SQLite API开发的Android native程序示例，该示例最终编译成的二进制可执行程序名为SqliteTest。

注意 本书后文所说的SQLite API特指libsqlite.so提供的Native层的API。

使用SQLite API开发的Android native程序示例的代码如下：

[-->SqliteTest.cpp：libsqlite示例]

```
#include<unistd.h>
#include<sqlite3.h> //包含SQLite API头文件，这里的3是SQLite的版本号
#include<stdlib.h>
#define LOG_TAG "SQLITE_TEST" //定义该进程logcat输出的标签
#include<utils/Log.h>
#ifndef NULL
#define NULL (0)
#endif
//声明数据库文件的路径
#define DB_PATH "/mnt/sdcard/sqlite3test.db"
```

```
/*
```

声明一个全局的SQLite句柄，开发者无须了解该数据结构的具体内容，只要知道它代表了使用者

和数据库的一种连接关系即可。以后凡是针对特定数据库的操作，都需要传入对应的SQLite句柄

```
*/
```

```
static sqlite3*g_pDBHandle=NULL ;
```

```
/*
```

定义一个宏，用于检测 SQLite API 调用的返回值，如果 value 不等于 expectValue，则打印警告，

并退出程序。注意，进程退出后，系统会自动回收分配的内存资源。对如此简单的例子来说，读者大可

不必苛责。其中，sqlite3_errmsg函数用于打印错误信息

```
*/
```

```
#define CHECK_DB_ERR (value, expectValue) \
do\
{\
    if (value !=expectValue) \
    {\
        LOGE ("Sqlite db fail :%s", g_pDBHandle==NULL?"db not\
connected":sqlite3_errmsg (g_pDBHandle) ) ;\
        exit (0) ;\
    }\
}
```

```
}while (0)
```

```
int main (int argc, char*argv[])
```

```
{
```

```
LOGD ("Delete old DB file") ;
```

```
unlink (DB_PATH) ;//先删除旧的数据库文件
```

```
LOGD ("Create new DB file") ;
```

```
/*
```

调用sqlite3_open创建一个数据库，并将和该数据库的连接环境保存在全局的SQLite句柄

g_pDBHandle中，以后操作g_pDBHandle就是操作DB_PATH对应的数据库

```
*/
```

```
int ret=sqlite3_open (DB_PATH, &g_pDBHandle) ;
```

```
CHECK_DB_ERR (ret, SQLITE_OK) ;
```

```
LOGD ("Create Table personal_info :") ; /*
```

定义宏TABLE_PERSONAL_INFO，用于描述为本例数据库建立一个表所用的SQL语句，

不熟悉SQL语句的读者可先学习相关知识。从该宏的定义可知，将建立一个名为personal_info的表，该表有4列，第一列是主键，类型是整型（SQLite中为INTEGER），

每加入一行数据该值会自动递增；第二列名为name，数据类型是字符串（SQLite中为TEXT）；

第三列名为age，数据类型是整型；第四列名为sex，数据类型是字符串

```
*/
```

```
#define TABLE_PERSONAL_INFO\
"CREATE TABLE personal_info" \
" (ID INTEGER primary key autoincrement, " \
"name TEXT, " \
"age INTEGER, " \
"sex TEXT" \
") "
//打印TABLE_PERSONAL_INFO所对应的SQL语句
LOGD ("\t%s\n", TABLE_PERSONAL_INFO) ;
//调用sqlite3_exec执行前面那条SQL语句
ret=sqlite3_exec ( g_pDBHandle, TABLE_PERSONAL_INFO, NULL,
NULL, NULL) ;
CHECK_DB_ERR (ret, SQLITE_OK) ;
/*
```

定义插入一行数据所使用的SQL语句，注意最后一行中的问号，它表示需要在插入数据

前和具体的值绑定。插入数据库对应的SQL语句是标准的INSERT命令

```
*/
```

```
LOGD ("Insert one personal info into personal_info table") ;
```

```
#define INSERT_PERSONAL_INFO\
"INSERT INTO personal_info" \

```

```
" (name, age, sex) " \

```

```
"VALUES" \

```

```
" (?, ?, ?) " //注意这一行语句中的问号
```

```
LOGD ("\t%s\n", INSERT_PERSONAL_INFO) ;
```

//sqlite3_stmt是SQLite中很重要的一个结构体，它代表了一条SQL语句

```

sqlite3_stmt* pstmt=NULL ;
/*
调用sqlite3_prepare初始化pstmt，并将其和INSERT_PERSONAL_INFO绑定。
也就是说，如果执行pstmt，就会执行INSERT_PERSONAL_INFO语句
*/
ret=sqlite3_prepare (g_pDBHandle, INSERT_PERSONAL_INFO, -1,
&pstmt, NULL) ;
CHECK_DB_ERR (ret, SQLITE_OK) ;
/*
调用sqlite3_bind_xxx为该pstmt中对应的问号绑定具体的值，该函数的第二个参数用于
指定第几个问号
*/
ret=sqlite3_bind_text (pstmt , 1 , "dengfanping" , -1 ,
SQLITE_STATIC) ;
ret=sqlite3_bind_int (pstmt, 2, 30) ;
ret=sqlite3_bind_text (pstmt, 3, "male", -1, SQLITE_STATIC) ;
//调用sqlite3_step执行对应的SQL语句，该函数如果执行成功，我们的
personal_info
//表中将添加一条新记录，对应值为 (1, dengfanping, 30, male)
ret=sqlite3_step (pstmt) ;
CHECK_DB_ERR (ret, SQLITE_DONE) ;
//调用sqlite3_finalize销毁该SQL语句
ret=sqlite3_finalize (pstmt) ;
//下面将从表中查询name为dengfanping的person的age值
LOGD ("select dengfanping's age from personal_info table") ;
pstmt=NULL ;
/*
重新初始化该pstmt，并将其和SQL语句“SELECT age FROM personal_info
WHERE name
=?”绑定
*/
ret=sqlite3_prepare ( g_pDBHandle , "SELECT age FROM
personal_info WHERE name
=?", -1, &pstmt, NULL) ;

```

```
CHECK_DB_ERR (ret, SQLITE_OK) ;
/*
绑定stmt中第一个问号为字符串“dengfanping”，最终该SQL语句为
SELECT age FROM personal_info WHERE name='dengfanping'
*/
ret=sqlite3_bind_text ( stmt , 1 , "dengfanping" , -1 ,
SQLITE_STATIC) ;
CHECK_DB_ERR (ret, SQLITE_OK) ;
//执行这条查询语句
while (true) //在一个循环中遍历结果集
{
    ret=sqlite3_step (stmt) ;
    if (ret==SQLITE_ROW)
    {
        //从结果集中取出第一列（由于执行SELECT时只选择了age，故最终结果只有一
列），
        //调用sqlite3_column_int返回结果集的第一列（从0开始）第一行的值
        int myage=sqlite3_column_int (stmt, 0) ;
        LOGD ("Got dengfanping's age :%d\n", myage) ;
    }
    else//如果ret为其他值，则退出循环
    break ;
}
else LOGD ("Find nothing\n") ; //SELECT执行失败
ret=sqlite3_finalize (stmt) ; //销毁stmt
if (g_pDBHandle)
{
    LOGD ("Close DB") ;
    //调用sqlite3_close关闭数据库连接并释放g_pDBHandle
    sqlite3_close (g_pDBHandle) ;
    g_pDBHandle=NULL ;
}
return 0 ;
}
```

通过上述示例代码可发现，SQLite API的使用主要集中在以下几点上：创建代表指定数据库的sqlite3实例。

创建代表一条SQL语句的sqlite3_stmt实例，在使用过程中首先要调用sqlite3_prepare将其和一条代表SQL语句的字符串绑定。如该字符串含有通配符“？”，后续就需要通过sqlite3_bind_xxx函数为通配符绑定特定的值以生成一条完整的SQL语句。最终调用sqlite3_step执行这条语句。

如果是查询（即SELECT命令）命令，则需调用 sqlite3_step 函数遍历结果集，并通过 sqlite3_column_xx等函数取出结果集中某一行指定列的值。

最后需调用sqlite3_finalize和sqlite3_close来释放sqlite3_stmt实例及sqlite3实例所占据的资源。

SQLite API的使用非常简单明了。不过很可惜，这份简单明了所带来的快捷，只供那些Native层程序开发者独享。对于Java程序员，他们只能使用Android在SQLite API之上所封装的SQLiteDatabase家族提供的类和相关API。综合考虑到架构及系统的稳定性和可扩展性等各种情况，Android在SQLite API之上进行了面向对象的封装和解耦等设计，最终呈现在大家面前的是一

个庞大而复杂的SQLiteDatabase家族，其成员有61个之多（参阅frameworks/base/core/java/android/database目录中的文件）。

现在读者应该理解本节标题“SQLite轻装上阵”中“轻”的真正含义了。在后续的分析过程中，我们主要和SQLiteDatabase家族打交道。随着分析的深入，读者能逐渐见认识SQLiteDatabase的“厚重”。

2.SQLiteDatabase家族介绍

图7-4展示了SQLiteDatabase家族中的几位重要成员。

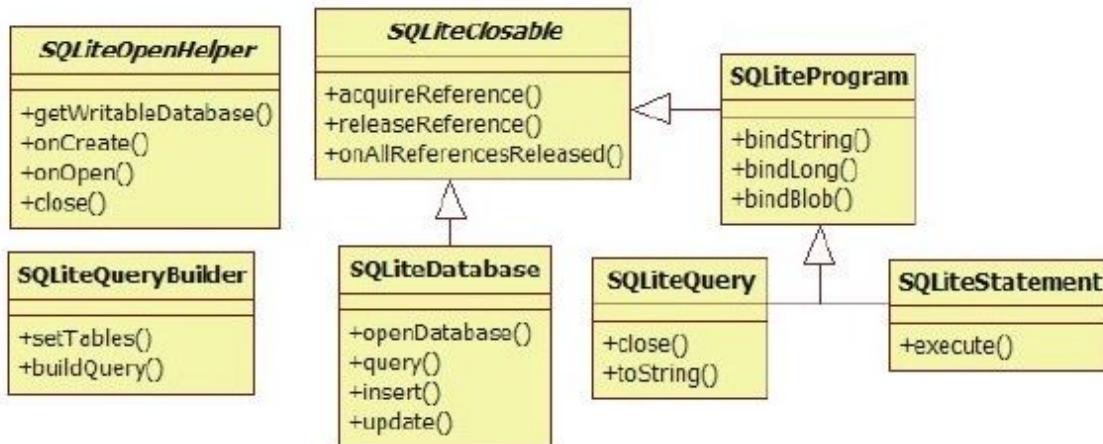


图 7-4 SQLiteDatabase家族部分成员

图7-4中相关类的说明如下：

SQLiteOpenHelper是一个帮助（Utility）类，用于方便开发者创建和管理数据库。

SQLiteQueryBuilder是一个帮助类，用于帮助开发者创建SQL语句。

SQLiteDatabase代表SQLite数据库，它内部封装了一个Native层的sqlite3实例。

Android提供了3个类，即 SQLiteProgram、SQLiteQuery和SQLiteStatement用于描述和SQL语句相关的信息。从图7-4可知，SQLiteProgram是基类，它提供了一些API用于参数绑定。SQLiteQuery主要用于query查询操作，而SQLiteStatement用于query之外的一些操作（根据SDK的说明，如果SQLiteStatement用于query查询，其返回的结果集只能是1行×1列的）。注意，在这3个类中，基类SQLiteProgram将保存一个指向Native层的sqlite3_stmt实例的变量，但是这个成员变量的赋值却和另外一个对开发者隐藏的类SQLiteCompiledSql有关。从这个角度看，可以认为Native层sqlite3_stmt实例的封装是由SQLiteCompiledSql完成的。这方面的知识在后文进行分析时即能见到。

SQLiteClosable用于控制SQLiteDatabase家族中一些类的实例的生命周期，例如SQLiteDatabase

实例和SQLiteQuery实例。每次使用这些实例对象前都需要调用acquireReference以增加引用计数，使用完毕后都需要调用releaseReferenece以减少引用计数。

提示 读者见识了SQLiteDatabase家族中这几位成员后有何感想？是否觉得要真正搞清楚它们还需要花费一番工夫呢？

下面来看MediaProvider是如何使用SQLiteDatabase的，重点关注SQLite数据库是如何创建的。

7.3.2 MediaProvider创建数据库分析

在MediaProvider中触发数据库的是attach函数，其代码如下：

[-->MediaProvider : attach]

```
private Uri attachVolume (String volume) {  
    Context context=getContext () ;  
    DatabaseHelper db ;  
    if (INTERNAL_VOLUME.equals (volume) ) {  
        .....//针对内部存储空间的数据库  
    }else if (EXTERNAL_VOLUME.equals (volume) ) {  
        .....  
        String  dbName="external-"+Integer.toHexString ( volumeID )  
        +".db" ;  
        //①构造一个DatabaseHelper对象  
        db=new DatabaseHelper (context, dbName, false,  
        false, mObjectRemovedCallback) ;  
        .....//省略不相关的内容  
    }.....  
    if (! db.mInternal) {  
        //②调用DatabaseHelper的getWritableDatabase函数，该函数返回值的类  
        型为  
        //SQLiteDatabase，即代表SQLite数据库的对象  
        createDefaultFolders (db.getWritableDatabase () ) ;  
        .....  
    }  
    .....  
}
```

以上代码中列出了两个关键点，分别是：

构造一个DatabaseHelper对象。

调用 DatabaseHelper 对象的 getWritableDatabase 函数得到一个代表 SQLite 数据库的 SQLiteDatabase 对象。

1.DatabaseHelper分析

DatabaseHelper 是 MediaProvider 的内部类，它从 SQLiteOpenHelper 派生。

(1) DatabaseHelper 构造函数分析

DatabaseHelper 构造函数的代码如下：

[--> MediaProvider.java : DatabaseHelper]

```
public DatabaseHelper (Context context, String name, boolean  
internal,  
boolean earlyUpgrade,  
SQLiteDatabase.CustomFunction objectRemovedCallback) {  
//重点关注其基类的构造函数  
super (context, name, null, DATABASE_VERSION) ;  
mContext=context ;  
 mName=name ;  
mInternal=internal ;  
mEarlyUpgrade=earlyUpgrade ;  
mObjectRemovedCallback=objectRemovedCallback ;  
}
```

SQLiteOpenHelper 作为 DatabaseHelper 的 基类，其构造函数的代码如下：

[-->SQLiteOpenHelper.java :
SQLiteOpenHelper]

```
public SQLiteOpenHelper ( Context context, String name,  
CursorFactory factory,  
int version) {  
    //调用另外一个构造函数，注意它新建了一个默认的错误处理对象  
    this ( context, name, factory, version, new  
DefaultDatabaseErrorHandler () );  
}  
public SQLiteOpenHelper ( Context context, String name,  
CursorFactory factory,  
int version, DatabaseErrorHandler errorHandler) {  
    ....  
    mContext=context;  
    mName=name;  
    //看到“factory”一词，读者要能想到设计模式中的工厂模式，在本例中该变量  
    为null  
    mFactory=factory;  
    mNewVersion=version;  
    mErrorHandler=errorHandler;  
}
```

上面这些函数都比较简单，其中却蕴含一个较为深刻的设计理念，具体如下：从 SQLiteOpenHelper 的构造函数中可知，MediaProvider 对应的数据库对象（即 SQLiteDatabase 实例）并不在该函数中创建。那

么，代表数据库的SQLiteDatabase实例是何时创建呢？此处使用了所谓的延迟创建（lazy creation）的方法，即SQLiteDatabase实例真正创建的时间是在第一次使用它的时候，也就是本例中第二个关键点函数getWritableDatabase。

在分析getWritableDatabase函数之前，先介绍一些和延迟创建相关的知识。

延迟创建或延迟初始化（lazy initialization）所谓的“重型”资源（如占内存较大或创建时间比较长的资源），是系统开发和设计中常用的一种策略^[1]。在使用这种策略时，开发人员不仅在资源创建时“斤斤计较”，在资源释放的问题上也是“慎之又慎”。资源释放的控制一般会采用引用计数技术。

结合前面对SQLiteDatabase的介绍会发现，SQLiteDatabase这个框架，在设计时不是简单地将SQLite API映射到Java层，而是有大量更为细致的考虑。例如，在这个框架中，资源创建采用了lazy creation方法，资源释放又利用SQLiteClosable来控制生命周期。

建议 Android中的SQLiteDatabase框架虽更完善、更具扩展性，但是使用它比直接使用SQLite API要复杂得多，因此在开发过程中，应当

根据实际情况综合考虑是否使用该框架。例如，笔者在开发公司的DLNA解决方案时，就直接使用了SQLite API，而没使用这个框架。

(2) getWritableDatabase函数分析

现在来看getWritableDatabase的代码，具体如下：

[-->SQLiteDatabase.java :
getWritableDatabase]

```
public synchronized SQLiteDatabase getWritableDatabase () {  
    if (mDatabase !=null) {  
        //第一次调用该函数时mDatabase还未创建。以后的调用将直接返回已经创建好的mDatabase  
    }  
    boolean success=false;  
    SQLiteDatabase db=null;  
    if (mDatabase !=null) mDatabase.lock () ;  
    try{  
        mIsInitializing=true;  
        if ( mName==null) {  
            db=SQLiteDatabase.create (null) ;  
        }else{  
            //①调用Context的openOrCreateDatabase创建数据库  
            db=mContext.openOrCreateDatabase ( mName, 0,  
                mFactory, mErrorHandler) ;  
        }  
        int version=db.getVersion () ;  
        if (version !=mNewVersion) {  
            db.beginTransaction () ;  
            try{
```

```
if (version==0) {  
/*  
如果初次创建该数据库（即对应的数据库文件不存在），则调用子类实现的  
onCreate函数。子类实现的onCreate函数将完成数据库建表等操作。读者不妨  
查看MediaProvider DatabaseHelper实现的onCreate函数  
onCreate (db) ;  
}else{  
//如果从数据库文件中读出来的版本号与MediaProvider设置的版本号不一致，  
//则调用子类实现的onDowngrade或onUpgrade做相应处理  
if (version>mNewVersion)  
onDowngrade (db, version, mNewVersion) ;  
else  
onUpgrade (db, version, mNewVersion) ;  
}  
db.setVersion (mNewVersion) ;  
db.setTransactionSuccessful () ;  
}finally{  
db.endTransaction () ;  
}  
}//if (version !=mNewVersion) 判断结束  
onOpen (db) ;//调用子类实现的onOpen函数  
success=true ;  
return db ;  
}.....
```

由以上代码可知，代表数据库的 SQLiteDatabase 对象是由 Context openOrCreateDatabase 创建的。下面单起一节具体分析此函数。

2.ContextImpl openOrCreateDatabase分析

(1) openOrCreateDatabase函数分析

相信读者已能准确定位openOrCreateDatabase函数的真正实现了，它就在ContextImpl.java中，其代码如下：

[-->ContextImpl.java : openOrCreateDatabase]

```
public SQLiteDatabase openOrCreateDatabase (String name, int mode,
                                         CursorFactory factory, DatabaseErrorHandler errorHandler) {
    File f=validateFilePath (name, true) ;
    //调用SQLiteDatabase的静态函数openOrCreateDatabase创建数据库
    SQLiteDatabase db=SQLiteDatabase.openOrCreateDatabase
    (f.getPath () ,
     factory, errorHandler) ;
    setFilePermissionsFromMode (f.getPath () , mode, 0) ;
    return db;
}
```

[-->SQLiteDatabase.java : openDatabase]

```
public static SQLiteDatabase openDatabase ( String path,
                                         CursorFactory
                                         factory, int flags, DatabaseErrorHandler errorHandler) {
    //又调用openDatabase创建SQLiteDatabase实例，真的是层层转包啊
    SQLiteDatabase sqliteDatabase=openDatabase (path, factory,
                                                flags, errorHandler, (short) 0) ;
    if      (      sBlockSize==0      )      sBlockSize=new      StatFs
    ("/data") .getBlockSize () ;
    //为该SQLiteDatabase实例设置一些参数。这些内容和SQLite本身的特性有关，本书不
    //深入讨论这方面的内容，感兴趣的读者不妨参考SQLite官网提供的资料
    sqliteDatabase.setPageSize (sBlockSize) ;
    sqliteDatabase.setJournalMode (path, "TRUNCATE") ;
```

```
synchronized (mActiveDatabases) {  
    mActiveDatabases.add (  
        new WeakReference<SQLiteDatabase> (sqliteDatabase) ) ;  
}  
return sqliteDatabase;  
}
```

openDatabase将真正创建一个SQLiteDatabase实例，其相关代码是：

[-->SqliteDatabase.java : openDatabase]

```
private static SQLiteDatabase openDatabase ( String path,  
CursorFactory factory,  
int flags, DatabaseErrorHandler errorHandler,  
short connectionNum) {  
    //构造一个SQLiteDatabase实例  
    SQLiteDatabase db=new SQLiteDatabase (path, factory, flags,  
errorHandler,  
connectionNum) ;  
    try{  
        db.dbopen (path, flags) ;//打开数据库, dbopen是一个native函数  
        db.setLocale (Locale.getDefault () ) ;//设置Locale  
        .....  
    return db ;  
}.....  
}
```

其实openDatabase主要就干了两件事情，即创建一个SQLiteDatabase实例，然后调用该实例的dbopen函数。

(2) SQLiteDatabase的构造函数及dbopen函数分析

先看SQLiteDatabase 的构造函数，代码如下：

[-->SQLiteDatabase.java : SQLiteDatabase]

```
private SQLiteDatabase (String path, CursorFactory factory,
int flags,
DatabaseErrorHandler errorHandler, short connectionNum) {
setMaxSqlCacheSize (DEFAULT_SQL_CACHE_SIZE) ;
mFlags=flags ;
mPath=path ;
mFactory=factory ;
mPrograms=new WeakHashMap<SQLiteClosable, Object> () ;
//config_cursorWindowSize值为2048，所以下面得到的limit值应该为
8MB
int limit=Resources.getSystem () .getInteger (
com.android.internal.R.integer.config_cursorWindowSize)
*1024*4 ;
native_setSqliteSoftHeapLimit (limit) ;
}
```

前面说过，Java层的SQLiteDatabase对象会和一个Native层sqlite3实例绑定，从以上代码中可发现，绑定的工作并未在构造函数中进行。实际上，该工作是由dbopen函数完成的，其相关代码如下：

[-->android_database_SQLiteDatabase.cpp :
dbopen]

```
static void dbopen ( JNIEnv*env, jobject object, jstring pathString, jint flags)
{
    int err;
    sqlite3*handle=NULL ;
    sqlite3_stmt*statement=NULL ;
    char const*path8=env->GetStringUTFChars ( pathString,
NULL) ;
    int sqliteFlags ;
    registerLoggingFunc (path8) ;
    if (flags&CREATE_IF_NECESSARY) {
        sqliteFlags=SQLITE_OPEN_READWRITE|SQLITE_OPEN_CREATE ;
    }.....
    //调用sqlite3_open_v2函数创建数据库，sqlite3_open_v2和示例中的
    sqlite3_open类似
    //handle用于存储新创建的sqlite3*类型的实例
    err=sqlite3_open_v2 (path8, &handle, sqliteFlags, NULL) ;
    .....
    sqlite3_soft_heap_limit (sSqliteSoftHeapLimit) ;
    err=sqlite3_busy_timeout (handle, 1000/*ms*/) ;
    .....
    //Android在原生SQLite之上还做了一些特殊的定制，相关内容留待本节最后分
析
    err=register_android_functions (handle, UTF16_STORAGE) ;
    // 将 handle 保存到 Java 层的 SQLiteDatabase 对象中，这样 Java 层
    SQLiteDatabase实例
    //就和一个Native层的sqlite3实例绑定到一起了
    env->SetIntField (object, offset_db_handle, (int) handle) ;
    handle=NULL ; //The caller owns the handle now.
    done :
    if (path8 !=NULL) env->ReleaseStringUTFChars (pathString,
path8) ;
```

```
if (statement !=NULL) sqlite3_finalize (statement) ;  
if (handle !=NULL) sqlite3_close (handle) ;  
}
```

从上述代码可知，使用dbopen函数其实就是为了得到Native层的一个sqlite3实例。另外，Android对SQLite还设置了一些与平台相关的函数，这部分内容将在后文进行分析。

3.SQLiteCompiledSql介绍

前文曾提到，Native层sqlite3_stmt实例的封装是由未对开发者公开的类SQLiteCompiledSql完成的。由于SQLiteCompiledSql的隐秘性，没有在图7-4中把它列出来。现在我们就来揭开它神秘的面纱，其代码如下：

[-->SQLiteCompiledSql.java :
SQLiteCompiledSql]

```
SQLiteCompiledSql (SQLiteDatabase db, String sql) {  
    db.verifyDbIsOpen () ;  
    db.verifyLockOwner () ;  
    mDatabase=db ;  
    mSqlStmt(sql ;  
    .....  
    nHandle=db.mNativeHandle ;  
    native_compile (sql) ;//调用native_compile函数，代码如下  
}
```

[-->
android_database_SQLiteCompiledSql.cpp :
native_compile]

```
static void native_compile ( JNIEnv*env, jobject object,
jstring sqlString)
{
    compile (env, object, GET_HANDLE (env, object) , sqlString) ;
}

//下面来看compile的实现
sqlite3_stmt*compile (JNIEnv*env, jobject object,
sqlite3*handle, jstring sqlString)
{
    int err;
    jchar const*sql;
    jsize sqlLen;
    sqlite3_stmt*statement=GET_STATEMENT (env, object) ;
    if (statement !=NULL) .....//释放之前的sqlite3_stmt实例
    sql=env->GetStringChars (sqlString, NULL) ;
    sqlLen=env->GetStringLength (sqlString) ;
    //调用sqlite3_prepare16_v2得到一个sqlite3_stmt实例
    err=sqlite3_prepare16_v2 (handle, sql, sqlLen*2, &statement,
NULL) ;
    env->ReleaseStringChars (sqlString, sql) ;
    if (err==SQLITE_OK) {
        //保存到Java层的SQLiteCompiledSql对象中
        env->SetIntField ( object, gStatementField , ( int )
statement) ;
        return statement;
    }.....
```

当compile函数执行完后，一个绑定了SQL语句的sqlite3_stmt实例就和Java层的SQLiteCompileSql对象绑定到一起了。

4.Android SQLite自定义函数介绍

本节将介绍Android在SQLite上自定义的一些函数。一切还得从SQL的触发器说起。

(1) 触发器介绍

触发器(Trigger)是数据库开发技术中一个常见的术语。其本质非常简单，就是在指定表上发生特定事情时，数据库需要执行的某些操作。还是有点模糊吧？再来看MediaProvider设置的一个触发器：

```
db.execSQL ( "CREATE TRIGGER IF NOT EXISTS images_cleanup
DELETE ON images"+
"BEGIN"+
"DELETE FROM thumbnails WHERE image_id=old._id ; "+
"SELECT_DELETE_FILE (old._data) ; "+
"END" ) ;
```

上面这条SQL语句是什么意思呢？

CREATE TRIGGER IF NOT EXISTS images_cleanup : 如果没有定义名为

images_cleanup 的触发器，就创建一个名为 images_cleanup的触发器。

DELETE ON images：设置该触发器的触发条件。显然，当我们对images表执行delete操作时，该触发器将被触发。

BEGIN和END之间则定义了该触发器要执行的动作。从前面的代码可知，它将执行两项操作：

删除thumbnails（缩略图）表中对应的信息。为什么要删除缩略图呢？因为原图的信息已经不存在了，留着缩略图也没用。

执行 _DELETE_FILE 函数，其参数是 old_data。从名字上来看，这个函数的功能应为删除文件。为什么要删除此文件？原因也很简单，数据库都没有该项信息了，还留着图片干什么！另外，如不删除文件，下一次媒体扫描时就又会把它们找到。

提示 _DELETE_FILE这个操作曾给笔者及同仁带来极大困扰，因为最开始并不知道有这个触发器。结果好不容易下载的测试文件全部被删除了。另外，由于MediaProvider本身的设计缺陷，频繁挂/卸载SD卡时也会错误删除数据库信息（这

个缺陷只能尽量避免，无法彻底根除），结果实体文件也被删除掉了。

有人可能会感到奇怪，这个`_DELETE_FILE`函数是谁设置的呢？答案就在`register_android_functions`中。

(2) `register_android_functions`介绍

`register_android_functions`在`dbopen`中被调用，其代码如下：

```
[-->sqlite3_android.cpp :  
register_android_functions]
```

```
//dbopen调用它时，第二个参数设置为0  
extern"C"int register_android_functions (sqlite3*handle, int  
utf16Storage)  
{  
    int err;  
    UErrorCode status=U_ZERO_ERROR;  
    UCollator*collator=ucol_open (NULL, &status) ;  
    .....  
    if (utf16Storage) {  
        err=sqlite3_exec (handle, "PRAGMA encoding='UTF-16'", 0, 0,  
        0) ;  
        .....  
    }else{  
        //sqlite3_create_collation_xx定义一个用于排序的文本比较函数，读者  
        可自行阅读  
        //SQLite官方文档以获得更详细的说明  
        err=sqlite3_create_collation_v2 (handle, "UNICODE",
```

```
SQLITE_UTF8, collator, collate8,
    (void (*) (void*) ) localized_collator_dtor) ;
}
/*
```

调用sqlite3_create_function创建一个名为"PHONE_NUMBERS_EQUAL"的函数，

第三个参数2表示该函数有两个参数，SQLITE_UTF8表示字符串编码为UTF8，
phone_numbers_equal为该函数对应的函数指针，也就是真正会执行的函数。
注意

"PHONE_NUMBERS_EQUAL" 是 SQL 语句中使用的函数名，
phone_numbers_equal是Native

层对应的函数
*/
err=sqlite3_create_function (
handle, "PHONE_NUMBERS_EQUAL", 2,
SQLITE_UTF8, NULL, phone_numbers_equal, NULL, NULL) ;
.....
//注册_DELETE_FILE对应的函数为delete_file
err=sqlite3_create_function (handle , "_DELETE_FILE" , 1 ,
SQLITE_UTF8,
NULL, delete_file, NULL, NULL) ;
if (err !=SQLITE_OK) {
return err ;
}
#if ENABLE_ANDROID_LOG
err=sqlite3_create_function (handle, "_LOG", 1, SQLITE_UTF8,
NULL, android_log, NULL, NULL) ;
.....
#endif
.....//和PHONE相关的一些函数
return SQLITE_OK ;
}

register_android_functions注册了Android平台上定制的一些函数。来看和_DELETE_FILE有关

的delete_file函数，其代码为：

[-->Sqlite3_android.cpp : delete_file]

```
static void delete_file (sqlite3_context*context, int argc,
sqlite3_value**argv)
{
if (argc !=1) {
sqlite3_result_int (context, 0) ;
return ;
}
//从argv中取出第一个参数，这个参数是触发器调用_DELETE_FILE时传递的
char const*path= ( char const* ) sqlite3_value_text
(argv[0]) ;
.....
/*
Android 4.0之后，系统支持多个存储空间（很多平板都有一块很大的内部存储
空间）。
```

为了保持兼容性，环境变量EXTERNAL_STORAGE还是指向SD卡的挂载目录，而其他存储设备的

挂载目录由SECONDARY_STORAGE表示，各个挂载目录由冒号分隔开。

下面这段代码用于判断_DELETE_FILE函数所传递的文件路径是不是正确的

*/

```
bool good_path=false ;
char const*external_storage=getenv ("EXTERNAL_STORAGE") ;
if (external_storage&&strncmp (external_storage,
path, strlen (external_storage) ) ==0) {
good_path=true ;
}else{
char const*secondary_paths=getenv ("SECONDARY_STORAGE") ;
while (secondary_paths&&secondary_paths[0]) {
const char*colon=strchr (secondary_paths, ':') ;
int length= (colon?colon-secondary_paths :
strlen (secondary_paths) ) ;
if (strncmp (secondary_paths, path, length) ==0) {
```

```
good_path=true ;
}
secondary_paths+=length ;
while (*secondary_paths=='：') secondary_paths++ ;
}
}
if (!good_path) {
sqlite3_result_null (context) ;
return ;
}
//调用unlink删除文件
int err=unlink (path) ;
if (err !=-1) {
sqlite3_result_int (context, 1) ;//设置返回值
}else{
sqlite3_result_int (context, 0) ;
}
}
```

[1]其实这是一种广义的设计模式，读者可参考《Pattern-Oriented Software Architecture Volume 3：Patterns for Resource Management》一书加深理解。

7.3.3 SQLiteDatabase创建数据库的分析总结

本节以MediaProvider创建数据库为入口，对SQLite及Java层的SQLiteDatabase进行了介绍。其中，应重点阅读SQLiteTest中的示例代码，通过它可以掌握SQLite API的用法。在此基础上，还介绍了SQLiteDatabase家族并分析了MediaProvider中数据库创建的相关代码。

本节涉及的知识点并不复杂，读者不妨重温一下7.3.1节的内容，以加深对SQLite及SQLiteDatabase的理解。

建议 先从SQLiteDatabase开始分析，学习成本会较大，因此，可自己尝试封装一个轻量级的C++SQLite库，通过这次尝试，学习并掌握如何设计一个良好的框架。

7.4 Cursor的query函数的实现分析

本节将分析CP中另一个比较复杂的知识点，即query函数的实现。下面从CR的query函数开始介绍，其代码如下：

[-->ContentResolver.java : query]

```
/*
```

注意query函数中的参数，它们组合后得到的SQL语句如下（方括号中的语句为笔者添加的注释）

SELECT projection指定的列名[如果projection为null，则使用“*”]
FROM表名[由目标CP根据uri参数设置]WHERE
(selection) [如果selection中有通配符，则具体参数由selectionArgs
指定]

```
ORDERED BY sortOrder
```

```
*/
```

```
public final Cursor query (Uri uri, String[]projection,  
String selection, String[]selectionArgs, String sortOrder) {  
/*
```

根据前面的分析，下面这个函数返回的provider的真实类型是ContentProviderProxy，

对应的Bn端对象的真实类型是CP的内部类Transport。本次执行query的目标CP是MediaProvider

```
*/
```

```
IContentProvider provider=acquireProvider (uri) ;
```

```
//来看下面的代码
```

```
try{
```

```
long startTime=SystemClock.uptimeMillis () ;
```

```
//①调用远端进程的query函数
```

```
Cursor qCursor=provider.query (uri, projection,
```

```
selection, selectionArgs, sortOrder) ;  
if (qCursor==null) {  
//如果返回的结果为空，则释放provider，这个函数的作用在7.2.3节介绍过  
releaseProvider (provider) ;  
return null;  
}  
//②计算查询结果包含的数据项条数，结果保存在qCursor的内部变量中  
qCursor.getCount () ;  
long durationMillis=SystemClock.uptimeMillis () -startTime ;  
//③最终返回给客户端一个游标对象，其真实数据类型为CursorWrapperInner  
return new CursorWrapperInner (qCursor, provider) ;  
}
```

上面这段代码揭示了CR的query函数的工作流程：

调用远程CP的query函数，返回一个Cursor类型的对象qCursor。

该函数最终返给客户端的是一个CursorWrapperInner类型的对象。

Cursor和CursorWrappperInner这两个新出现的数据类型严重干扰了我们的思考。暂且不管它们，先来分析以上代码中列出的几个关键点函数。首先要分析的是第一个关键点函数query。

7.4.1 提取query关键点

1.客户端query关键点

按以前的分析习惯，碰到Binder调用时会马上转入服务端（即Bn端）去分析，但是这个思路在query函数中行不通。为什么？来看IContentProvider Bp端的query函数，它定义在ContentProviderProxy中，代码如下：

[-->ContentProviderNative.java :
ContentProviderProxy.query]

```
public Cursor query ( Uri url, String[]projection, String  
selection,  
String[]selectionArgs, String sortOrder ) throws  
RemoteException{  
    //①构造一个BulkCursorToCursorAdaptor对象  
    BulkCursorToCursorAdaptor adaptor=new  
    BulkCursorToCursorAdaptor () ;  
    Parcel data=Parcel.obtain () ;  
    Parcel reply=Parcel.obtain () ;  
    try{  
        data.writeInterfaceToken ( IContentProvider.descriptor ) ;  
        .....//将参数打包到data请求包中  
        //②adaptor.getObserver () 返回一个IContentObserver类型的对象，把它  
        //也打包到data请求包中。ContentObserver相关的知识留到第8章再分析  
        data.writeStrongBinder ( adaptor.getObserver () .asBinder  
        () ) ;  
        //发送请求给远端的Bn端  
        mRemote.transact ( IContentProvider.QUERY_TRANSACTION, data,  
        reply, 0 ) ;  
        DatabaseUtils.readExceptionFromParcel ( reply ) ;  
        //③从回复包中得到一个IBulkCursor类型的对象
```

```
IBulkCursor bulkCursor=
BulkCursorNative.asInterface (reply.readStrongBinder () ) ;
if (bulkCursor !=null) {
int rowCount=reply.readInt () ;
int idColumnPosition=reply.readInt () ;
boolean wantsAllOnMoveCalls=reply.readInt () !=0 ;
//④调用adaptor的initialize函数
adaptor.initialize (bulkCursor, rowCount,
idColumnPosition, wantsAllOnMoveCalls) ;
}.....
return adaptor ;
}.....
finally{
data.recycle () ;
reply.recycle () ;
}
}
```

从以上代码中可发现，ContentProviderProxy query函数中还大有文章，其中一共列出了4个关键点。最令人头疼的是其中新出现的两个类 BulkCursorToCursorAdaptor和IBulkCursor。此处不必急于分析它们。

我们再到服务端去提取query函数的关键点。

2.服务端query关键点

根据第2章对Binder的介绍，客户端发来的请求先在Bn端的onTransact中得到处理，代码如下：

[-->ContentProviderNative.java : onTransact]

```
public boolean onTransact ( int code, Parcel data, Parcel
reply, int flags)
    throws RemoteException{
try{
switch (code) {
case QUERY_TRANSACTION://处理query请求
{
data.enforceInterface (IContentProvider.descriptor) ;
Uri url=Uri.CREATOR.createFromParcel (data) ;
.....//从请求包中提取参数
//⑤创建ContentObserver Binder通信的Bp端
IContentObserver observer=IContentObserver.Stub.asInterface
(
    data.readStrongBinder () ) ;
//⑥调用MediaProvider实现的query函数。Cursor是一个接口类，那么变量
//cursor的真实类型是什么？
Cursor cursor=query ( url,     projection,     selection,
selectionArgs,
sortOrder) ;
if (cursor !=null) {
//⑦创建一个CursorToBulkCursorAdaptor类型的对象
CursorToBulkCursorAdaptor adaptor=new
CursorToBulkCursorAdaptor (
cursor, observer, getProviderName () ) ;
final IBinder binder=adaptor.asBinder () ;
//⑧返回结果集所含记录项的条数，这个函数看起来极不起眼，但却非常关键
final int count=adaptor.count () ;
//返回名为"_id"的列在结果集中的索引位置，该列由数据库在创建表时自动添
加
final                                         int
index=BulkCursorToCursorAdaptor.findRowIdColumnIndex (
    adaptor.getColumnNames () ) ;
//wantsAllOnMoveCalls一般为false，读者阅读完本章后再自行分析它
final boolean wantsAllOnMoveCalls=
    adaptor.getWantsAllOnMoveCalls () ;
reply.writeNoException () ;
```

```
//将binder的信息写到reply回复包中  
reply.writeStrongBinder (binder) ;  
reply.writeInt (count) ;//将结果集包含的记录项行数返回给客户端  
reply.writeInt (index) ;  
reply.writeInt (wantsAllOnMoveCalls?1:0) ;  
}.....  
return true ;  
}.....//其他情况处理  
.....  
}
```

和客户端对应，服务端的query处理也比较复杂，其中的“拦路虎”仍是新出现的几种数据类型。

在扫清这些“拦路虎”之前，应先把客户端和服务端query调用的关键点总结一下。

3.提取query关键点总结

我们提取query两端的调用关键点，如图7-5所示。

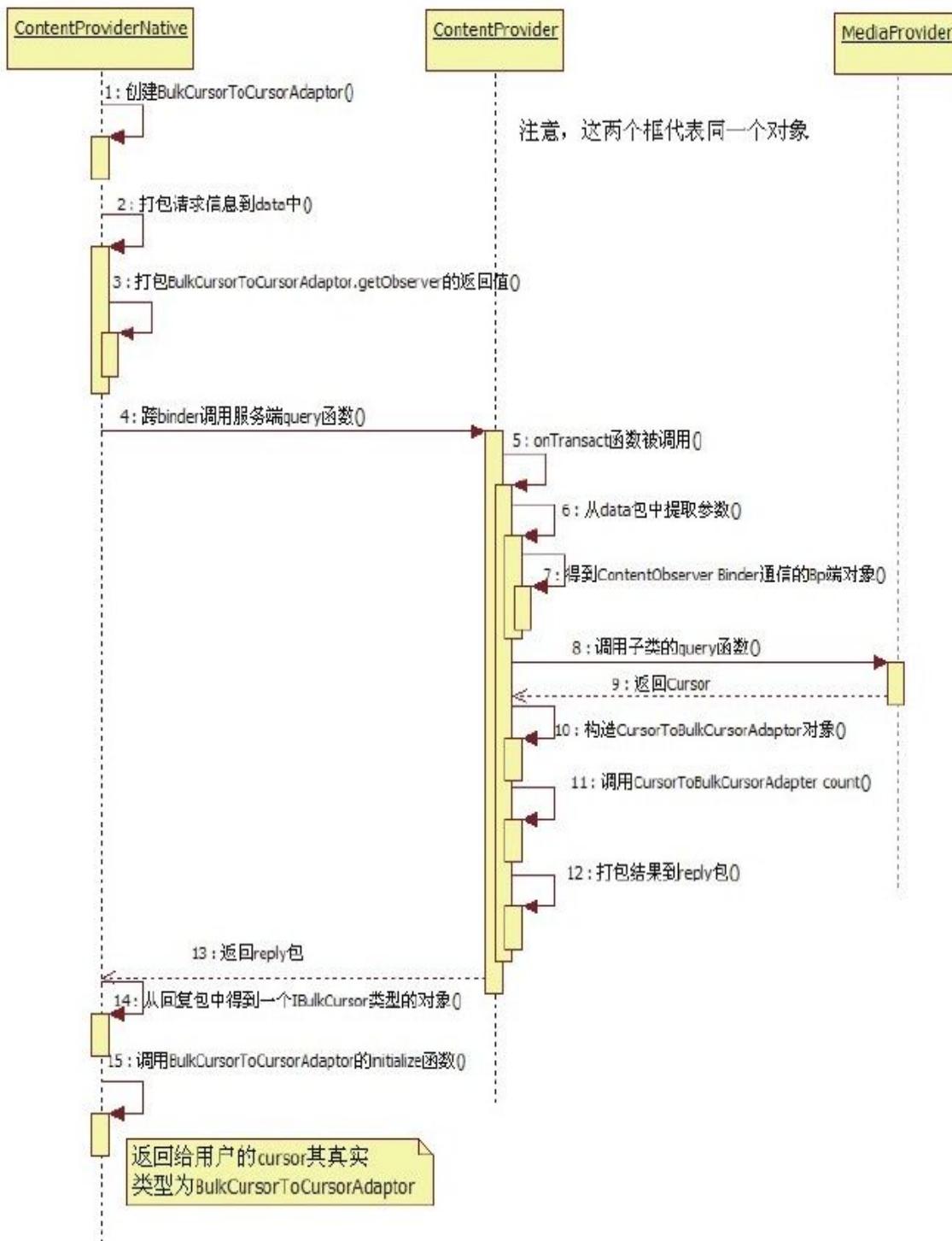


图 7-5 `query` 调用关键点示意

再来总结一下前面提到的几个“拦路虎”，它们分别是：

客户端创建的BulkCursorToCursorAdaptor，以及从服务端query后得到的IBulkCursor。

服务端创建的CursorToCursorAdaptor，以及从子类query函数返回的Cursor。

从名字上看，这几个类都和Cursor有关，所以有必要先搞清MediaProvider query返回的Cursor到底是什么。

注意 图7-5借用了UML的序列图来展示query调用顺序，其中ContentProvider（CP）框和MediaProvider框代表同一个对象。另外，图7-5中的调用函数编号并不完全对应代码中的关键函数调用编号。

7.4.2 MediaProvider的query分析

本节将分析MediaProvider的query函数。此次分析的焦点不是MediaProvider本身的业务逻辑，而是要搞清query函数返回的Cursor到底是什么，其代码如下：

[-->MediaProvider.java : query]

```
public Cursor query (Uri uri, String[]projectionIn, String  
selection,  
String[]selectionArgs, String sort) {  
int table=URI_MATCHER.match (uri) ;  
.....  
//根据uri取出对应的DatabaseHelper对象，MediaProvider针对内部存储  
中的媒体文件和  
//外部存储（即SD卡）中的媒体文件分别创建了两个数据库  
DatabaseHelper database=getDatabaseForUri (uri) ;  
//我们在 7.3.2 节 分析 过 getReadableDatabase 函数 的 兄 弟  
getWritableDatabase函数  
SQLiteDatabase db=database.getReadableDatabase () ;  
//创建一个SQLiteQueryBuilder对象，用于方便开发人员编写SQL语句  
SQLiteQueryBuilder qb=new SQLiteQueryBuilder () ;  
.....//设置qb的参数，例如调用setTables函数为本次查询设定目标Table  
//①调用SQLiteQueryBuilder的query函数  
Cursor c=qb.query (db, projectionIn, selection,  
combine (prependArgs, selectionArgs) , groupBy, null, sort,  
limit) ;  
if (c !=null) {  
//②调用该 Cursor 对象的 setNotificationUri 函数，这部分内容和  
ContentObserver
```

```
//有关，相关内容留到第8章再进行分析  
c.setNotificationUri (getContext () .getContentResolver () ,  
uri) ;  
}  
return c ;  
}
```

上边代码列出的两个关键点分别是：

调用SQLiteQueryBuilder的query函数得到一个Cursor类型的对象。

调用 Cursor 类型 对象 的 setNotificationUri 函数。从名字上看，该函数是为Cursor对象设置通知URI。和ContentObserver有关的内容留到第8章再进行分析。

下面来看SQLiteQueryBuilder的query函数。

1.SQLiteQueryBuilder的query函数分析

这部分的代码如下：

[-->SQLiteQueryBuilder.java : query]

```
public Cursor query ( SQLiteDatabase db ,  
String []projectionIn ,  
String selection , String []selectionArgs , String groupBy ,  
String having , String sortOrder , String limit ) {  
.....  
//调用buildQuery函数得到对应SQL语句的字符串  
String sql=buildQuery (
```

```
projectionIn, selection, groupBy, having,
sortOrder, limit) ;
/*
调用 SQLiteDatabase 的 rawQueryWithFactory 函数 , mFactory 是
SQLiteQueryBuilder
的成员变量，初始值为null，本例也没有设置它，故mFactory为null
*/
return db.rawQueryWithFactory (
mFactory, sql, selectionArgs,
SQLiteDatabase.findEditTable (mTables) ) ;
}
```

[-->SQLiteDatabase.java : rawQueryWithFactory]

```
public Cursor rawQueryWithFactory (
    CursorFactory cursorFactory, String sql,
    String[]selectionArgs,
    String editTable) {
    verifyDbIsOpen () ;
    BlockGuard.getThreadPolicy () .onReadFromDisk () ;
    //数据库开发中经常碰到连接池的概念，其目的也是缓存重型资源。有兴趣的读者不妨自行
    //研究一下这个getDbConnection函数
    SQLiteDatabase db=getDbConnection (sql) ;
    //创建一个SQLiteDirectCursorDriver对象
    SQLiteCursorDriver driver=new SQLiteDirectCursorDriver (
        db, sql, editTable) ;
    Cursor cursor=null ;
    try{
        //调用SQLiteCursorDriver的query函数
        cursor=driver.query (
            cursorFactory !=null?cursorFactory :mFactory,
            selectionArgs) ;
    }finally{
```

```
releaseDbConnection (db) ;  
}  
return cursor ;
```

以上代码中又出现一个新类型，即 SQLiteCursorDriver, cursor变量是其query函数的返回值。

(1) SQLiteCursorDriver的query函数分析

SQLiteCursorDriver的query函数的代码如下：

[-->SQLiteCursorDriver.java : query]

```
public Cursor query ( CursorFactory factory,  
String[]selectionArgs) {  
    //本例中，factory为空  
    SQLiteQuery query=null;  
    try{  
        mDatabase.lock (mSql) ;  
        mDatabase.closePendingStatements () ;  
        //①构造一个SQLiteQuery对象  
        query=new SQLiteQuery (mDatabase, mSql, 0, selectionArgs) ;  
        //原来，最后返回的游标对象其真实类型是SQLiteCursor  
        if (factory==null) { //②构造一个SQLiteCursor对象  
            mCursor=new SQLiteCursor (this, mEditTable, query) ;  
        }else{  
            mCursor=factory.newCursor (mDatabase, this,  
            mEditTable, query) ;  
        }  
        mQuery=query ;  
        query=null;  
        return mCursor ; //原来返回的cursor对象其真实类型是SQLiteCursor  
    }finally{
```

```
    if (query != null) query.close () ;  
    mDatabase.unlock () ;  
}  
}
```

SQLiteCursorDriver的query函数的主要功能就是创建一个SQLiteDatabase实例和一个SQLiteDatabase实例。至此，我们终于搞清楚了MediaProvider的query返回的游标对象其真实类型是SQLiteDatabase。

注意 这里的MediaProvider query指的是图7-5中的编号为8的关键调用。

下面来看SQLiteDatabase和SQLiteDatabase为何方神圣。

(2) SQLiteDatabase介绍

在图7-4中曾介绍过SQLiteDatabase，它保存了和查询（即SQL的SELECT命令）命令相关的信息，其构造函数的代码如下：

[-->SQLiteDatabase.java : 构造函数]

```
SQLiteDatabase ( SQLiteDatabase db, String query, int  
offsetIndex,  
String[] bindArgs) {  
//注意，在本例中offsetIndex为0，offsetIndex的意义到后面再解释  
super (db, query) ; //调用基类构造函数
```

```
mOffsetIndex=offsetIndex ;  
bindAllArgsAsStrings (bindArgs) ;  
}
```

SQLiteQuery 的基类是 SQLiteProgram，其代码如下：

[-->SQLiteProgram.java：构造函数]

```
SQLiteProgram (SQLiteDatabase db, String sql) {  
    this (db, sql, null, true) ;//调用另外一个构造函数，注意传递的参数  
}  
SQLiteProgram ( SQLiteDatabase db, String sql,  
Object []bindArgs,  
boolean compileFlag) {  
    //本例中bindArgs为null, compileFlag为true  
    mSql =sql.trim () ;  
    //返回该SQL语句的类型，query语句将返回STATEMENT_SELECT类型  
    int n=DatabaseUtils.getSqlStatementType (mSql) ;  
    switch (n) {  
        case DatabaseUtils.STATEMENT_UPDATE :  
            mStatementType=n|STATEMENT_CACHEABLE ;  
            break ;  
        case DatabaseUtils.STATEMENT_SELECT :  
            /*  
             * mStatementType成员变量用于标示该SQL语句的类型，如果该SQL语句  
             * 是 STATEMENT_SELECT 类型，则 mStatementType 将设置  
             * STATEMENT_CACHEABLE  
             * 标志。该标志表示此对象将被缓存起来，以避免再次执行同样的SELECT命令时重  
             * 新构造  
             * 一个对象  
             */  
            mStatementType=n|STATEMENT_CACHEABLE |  
            STATEMENT_USE_POOLED_CONN ;
```

```
break ;
.....//其他情况处理
default :
mStatementType=n ;
}
db.acquireReference () ; //增加引用计数
db.addSQLiteClosable (this) ;
mDatabase=db ;
nHandle=db.mNativeHandle ; //此mNativeHandle对应一个SQLite3实例
if (bindArgs !=null) .....//绑定参数
//complieAndBindAllArgs 将为此对象绑定一个 sqlite3_stmt 实例 ,
native层对象的指针
//保存在nStatement成员变量中
if (compileFlag) compileAndbindAllArgs () ;
}
```

来看 compileAndbindAllArgs 函数，其代码是：

[-->SQLiteProgram.java :
compileAndbindAllArgs]

```
void compileAndbindAllArgs () {
.....
//如果该对象还未和native层的sqlite3_stmt实例绑定，则调用compileSql
函数
if (nStatement==0) compileSql () ;
.....//绑定参数
for (int index : mBindArgs.keySet () ) {
Object value=mBindArgs.get (index) ;
if (value==null) {
native_bind_null (index) ;
}.....//绑定其他类型的数据
```

```
    }  
}
```

compileSql函数将绑定Java层SQLiteQuery对象到一个Native的sqlite3_stmt实例。根据前文的分析，这个绑定是通过SQLiteCompiledSql对象实施的，其相关代码如下：

[-->SQLiteProgram.java : compileSql]

```
private void compileSql () {  
    //如果mStatementType未设置STATEMENT_CACHEABLE标志，则每次都创建一个  
    //SQLiteCompiledSql对象。根据7.3.2节中的分析，该对象会真正和Native  
    //层的  
    //sqlite_stmt实例绑定  
    if ( (mStatementType&STATEMENT_CACHEABLE) ==0) {  
        mCompiledSql=new SQLiteCompiledSql (mDatabase, mSql) ;  
        nStatement=mCompiledSql.nStatement ;  
        return ;  
    }  
    //从SQLiteDatabase对象中查询是否已经缓存过符合该SQL语句要求的  
    //SQLiteCompiledSql  
    //对象  
    mCompiledSql=mDatabase.getCompiledStatementForSql (mSql) ;  
    if (mCompiledSql==null) {  
        //创建一个新的SQLiteCompiledSql对象，并把它保存到mDatabase中  
        mCompiledSql=new SQLiteCompiledSql (mDatabase, mSql) ;  
        mCompiledSql.acquire () ;  
        mDatabase.addToCompiledQueries (mSql, mCompiledSql) ;  
    }.....  
    //保存Native层sqlite3_stmt实例指针到nStatement成员变量  
    nStatement=mCompiledSql.nStatement ;
```

}

通过以上分析可以发现，SQLiteDatabase将和一个代表SELECT命令的sqlite3_stmt实例绑定。同时，为了减少创建sqlite3_stmt实例的开销，SQLiteDatabase框架还会把对应的SQL语句和对应的SQLiteCompiledSql对象缓存起来。如果下次执行同样的SELECT语句，那么系统将直接取出之前保存的SQLiteCompiledSql对象，这样就不用重新创建sqlite3_stmt实例了。

思考 与直接使用SQLite API相比，SQLiteDatabase框架明显考虑了更多问题。读者自己封装C++SQLite库时是否想到了这些问题？

(3) SQLiteCursor分析

再来看SQLiteCursor类，其构造函数的代码如下：

[-->SQLiteCursor.java：构造函数]

```
public SQLiteCursor ( SQLiteCursorDriver driver, String
editTable,
    SQLiteQuery query) {
    .....
    mDriver=driver ;
    mEditTable=editTable ;
    mColumnNameMap=null ;
    mQuery=query ; //保存此SQLiteQuery对象
```

```
query.mDatabase.lock (query.mSql) ;
try{
//得到此次执行query得到的结果集所包含的列数
int columnCount=mQuery.columnCountLocked () ;
mColumns=new String[columnCount] ;
for (int i=0 ; i<columnCount ; i++) {
String columnName=mQuery.columnNameLocked (i) ;
//保存列名
mColumns[i]=columnName ;
if ("_id".equals (columnName) ) {
mRowIdColumnIndex=i ; //保存"_id"列在结果集中的索引位置
}
}
}finally{
query.mDatabase.unlock () ;
}
```

SQLiteCursor比较简单，此处不再详述。

2.Cursor分析

至此，我们已经知道MediaProvider query返回的游标对象的真实类型了。现在，终于可以请Cursor家族登台亮相了，如图7-6所示。

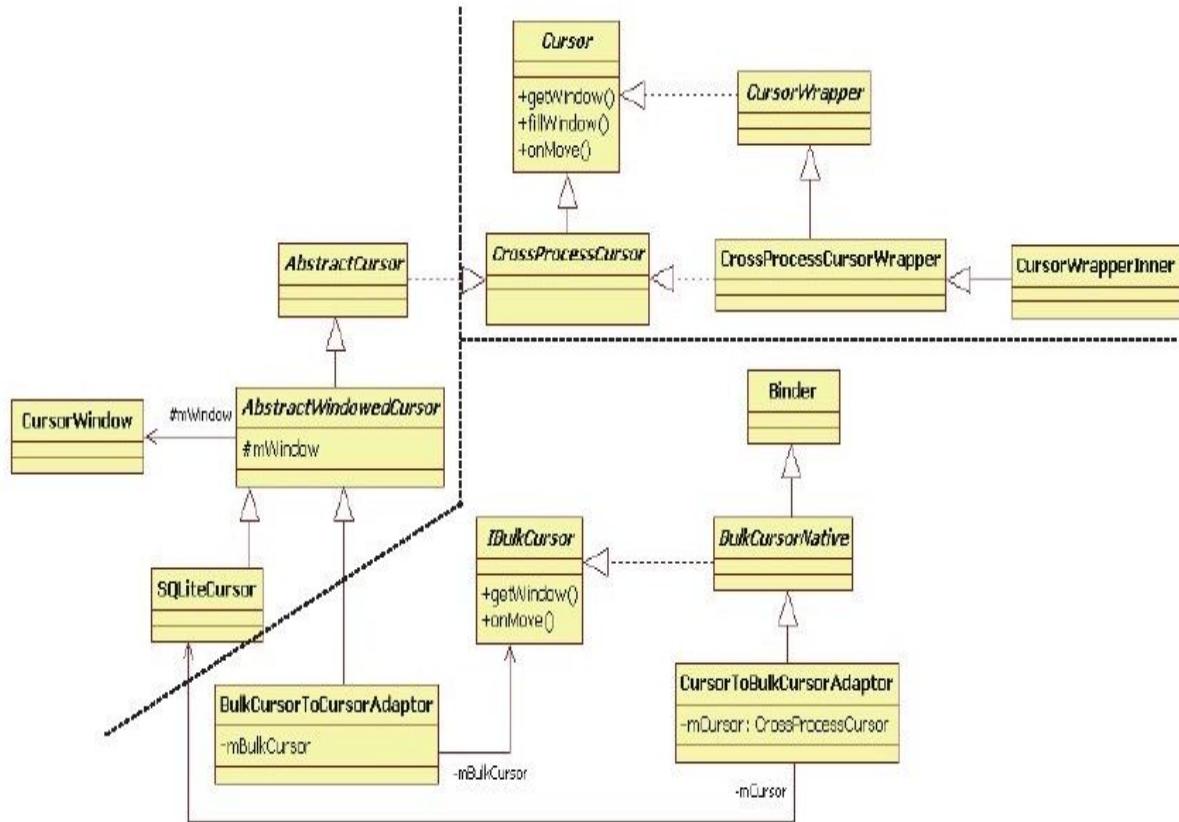


图 7-6 Cursor家族

图7-6中元素较多，包含的知识点也较为复杂，因此必须仔细阅读下文的解释。

通过7.3.1节中SqliteTest示例可知，query查询（即SELECT命令）的结果和一个Native sqlite_stmt实例绑定在一起，开发者可通过遍历该sqlite_stmt实例得到自己想要的结果（例如，调用sqlite3_step遍历结果集中的行，然后通过sqlite3_column_xxx取出指定列的值）。查询结果集可通过数据库开发技术中一个专用术语——游标（Cursor）来遍历和获取。图7-6的左上部分是

和Cursor有关的类，它们包括：接口类Cursor和CrossProcessCursor、抽象类AbstractCursor、AbstractWindowCursor，以及真正的实现类SQLiteCursor。根据前面的分析，SQLiteCursor内部保存一个已经绑定了sqlite3_stmt实例的SQLiteQuery对象，故读者可简单地把SQLiteCursor看成是一个已经包含了查询结果集的游标对象，虽然此时还未真正执行SQL语句。

如上所述，SQLiteCursor是一个已经包含了结果集的游标对象。从进程角度看，query的结果集目前还属于MediaProvider所在的进程，而本次query请求是由客户端发起的，所以一定要有一种方法将MediaProvider中的结果集传递到客户端进程。数据传递使用的技术很简单，就是大家耳熟能详的共享内存技术。SQLite API没有提供相关的功能，但是SQLiteDatabase框架对跨进程数据传递进行了封装，最终得到了图7-6左上部分的CursorWindow类。其代码中的注释明确表明了CursorWindow的作用，它是“A buffer containing multiple cursor rows”。

认识了CursorWindow类后，相信读者就能猜出query中数据传递的大致流程了。

MediaProvider 将结果集中的数据存储到 CursorWindow 的共享内存中，然后客户端将其从共享内存中取出来即可。

上述流程描述是对的，但实际过程并非如此简单，因为 SQLiteDatabase 框架希望客户端看到的不是共享内存，而是一个代表结果集的游标对象，就好像客户端查询的是本进程中的数据库一样。由于存在这种要求^[1]，Android 构造了图 7-6 右下角的类家族。其中，最重要的两个类是 CursorToBulkCursorAdaptor 和 BulkCursorToCursorAdaptor。从名字上看，它们采用了设计模式中的 Adaptor 模式；从继承关系上看，这两个类将参与跨进程的 Binder 通信（其中客户端使用的 BulkCursorToCursorAdaptor 通过 mBulkCursor 与位于 MediaProvider 所在进程的 CursorToBulkCursorAdaptor 通信）。这两个类中最重要的 onMove 函数，以后我们碰到时再作分析。

另外，图 7-6 中右上角部分展示了 CursorWrapperInner 类的派生关系。CursorWrapperInner 类是 ContentResolver query 函数最终返回给客户端的游标对象的类型。CursorWrapperInner 的目的应该是拓展 CursorToBulkCursorAdaptor 类的功能。

Cursor家族有些复杂。笔者觉得，目前对Cursor的架构设计有些过度（over-designed）。这不仅会导致我们分析时困难重重，并且会对实际代码的运行效率造成一定损失。

下面我们将焦点放到跨进程的数据传输上。

[1]此处应该还存在其他方面的设计考虑，希望读者能参与讨论。

7.4.3 query关键点分析

本节将按如下顺序分析query函数中的关键点：

介绍服务端的CursorToBulkCursorAdaptor及其count函数。

跨进程共享数据的关键类CursorWindow。

客户端的BulkCursorToCursorAdaptor及其initialize函数，以及返回给客户端使用的CursorWrapperInner类

1. CursorToBulkCursorAdaptor函数分析

(1) 构造函数分析

CursorToBulkCursorAdaptor构造函数的代码如下：

[-->CursorToBulkCursorAdaptor.java：构造函数]

```
public CursorToBulkCursorAdaptor ( Cursor cursor,
IContentObserver observer,
String providerName) {
```

```
// 传入的 cursor 变量其真实类型是 SQLiteCursor，它是  
CrossProcessCursor  
    if (cursor instanceof CrossProcessCursor) {  
        mCursor= (CrossProcessCursor) cursor ;  
    }else{  
        mCursor=new CrossProcessCursorWrapper (cursor) ;  
    }  
    mProviderName=providerName ;  
    synchronized (mLock) { //和ContentObserver有关，我们以后再作分析  
        createAndRegisterObserverProxyLocked (observer) ;  
    }  
}
```

CursorToBulkCursorAdaptor的构造函数很简单，此处不详述。来看下一个重要的函数，即CursorToBulkCursorAdaptor的count。该函数返回本次查询结果集所包含的行数。

(2) count函数分析

count函数的代码如下：

[-->CursorToBulkCursorAdaptor.java : count]

```
public int count () {  
    synchronized (mLock) {  
        throwIfCursorIsClosed () ; //如果mCursor已经关闭，则抛出异常  
        //CursorToBulkCursorAdaptor 的 mCursor 变量的真实类型是  
        SQLiteCursor  
        return mCursor.getCount () ;  
    }  
}
```

count最终将调用SQLiteDatabase的getCount函数，其代码如下：

[-->SQLiteDatabase.java : getCount]

```
public int getCount () {  
    if (mCount==NO_COUNT) { //NO_COUNT为-1，首次调用时满足if条件  
        fillWindow (0) ; //关键函数  
    }  
    return mCount ;  
}
```

getCount函数将调用一个非常重要的函数，即fillWindow。顾名思义，读者可以猜测到它的功能：将结果数据保存到CursorWindow的那块共享内存中。

下面单起一节来分析和CursorWindow相关知识点。

2.CursorWindow分析

CursorWindow的创建源于前边代码中对fillWindow的调用。fillWindow的代码如下：

[-->SQLiteDatabase.java : fillWindow]

```
private void fillWindow (int startPos) {  
    //①如果CursorWindow已经存在，则清空（clear）它，否则新创建一个  
    //CursorWindow对象
```

```
clearOrCreateLocalWindow (getDatabase () .getPath () ) ;  
mWindow.setStartPosition (startPos) ;  
//@getQuery返回一个SQLiteQuery对象，此处将调用它的fillWindow函数  
int count=getQuery () .fillWindow (mWindow) ;  
if (startPos==0) {  
mCount=count ;  
}.....  
}
```

下面先来看clearOrCreateLocalWindow函数。

(1) clearOrCreateLocalWindow函数分析

[-->SQLiteCursor. java :
clearOrCreateLocalWindow]

```
protected void clearOrCreateLocalWindow (String name) {  
if (mWindow==null) {  
mWindow=new CursorWindow (name, true) ;//创建一个CursorWindow  
对象  
}else mWindow.clear () ;//清空CursorWindow中的信息  
}
```

CursorWindow的构造函数的代码如下：

[-->CursorWindow. java : CursorWindow]

```
public CursorWindow (String name, boolean localWindow) {  
mStartPos=0 ;//本次查询的起始行位置，例如查询数据库表中第10到第100行  
的结果，  
//其起始行就是10  
/*
```

调用nativeCreate函数，注意传递的参数，其中sCursorWindowSize为2MB，localWindow

为 true 。 sCursorWindowSize 是一个静态变量，其值取自 frameworks/base/core/res/res

/values/config.xml 中定义的 config_cursorWindowSize 变量，该值是 2048KB，而

sCursorWindowSize 在此基础上扩大了 1024 倍，最终的结果就是 2MB

```
*/  
mWindowPtr=nativeCreate ( name, sCursorWindowSize,  
localWindow) ;  
mCloseGuard.open ("close") ;  
recordNewWindow (Binder.getCallingPid () , mWindowPtr) ;  
}
```

nativeCreate 是一个 native 函数，其真正实现在 android_database_CursorWindow.cpp 中，其代码如下：

[--> android_database_CursorWindow.cpp :
nativeCreate]

```
static jint nativeCreate (JNIEnv*env, jclass clazz,  
jstring nameObj, jint cursorWindowSize, jboolean localOnly)  
{  
    String8 name ;  
    if (nameObj) {  
        const char*nameStr=env->GetStringUTFChars (nameObj, NULL) ;  
        name.setTo (nameStr) ;  
        env->ReleaseStringUTFChars (nameObj, nameStr) ;  
    }  
    .....  
    CursorWindow*window ;  
    // 创建一个 Native 层的 CursorWindow 对象
```

```
    status_t      status=CursorWindow      :      create      (      name,
cursorWindowSize,
localOnly, &window) ;

.....
    return reinterpret_cast<jint> (window) ;//将指针转换成jint类型
}
```

不妨再看看 CursorWindow 的 create 函数，其代码如下：

[--> CursorWindow.cpp : create]

```
status_t CursorWindow : create (const String8 & name, size_t
size, bool localOnly,
CursorWindow**outCursorWindow) {
String8 ashmemName ("CursorWindow:") ;
ashmemName.append (name) ;
ashmemName.append (localOnly?" (local) ":" (remote) ") ;
status_t result ;
//创建共享内存，调用Android平台提供的ashmem_create_region函数
int ashmemFd=ashmem_create_region (ashmemName.string () ,
size) ;
if (ashmemFd<0) {
result=-errno ;
}else{
result=ashmem_set_prot_region (ashmemFd,
PROT_READ|PROT_WRITE) ;
if (result>=0) {
//映射共享内存以得到一块地址，data变量指向该地址的起始位置
void*data=::mmap (NULL, size, PROT_READ|PROT_WRITE,
MAP_SHARED, ashmemFd, 0) ;
.....
result=ashmem_set_prot_region (ashmemFd, PROT_READ) ;
```

```
if (result>=0) {  
    //创建一个CursorWindow对象  
    CursorWindow*window=new CursorWindow (name, ashmemFd,  
    data, size, false) ;  
    result=window->clear () ;  
    if (!result) {  
        *outCursorWindow=window ;  
        return OK ;//创建成功  
    }  
}.....//出错处理  
}  
return result ;  
}
```

由以上代码可知， CursorWindow的create函数将构造一个Native的CursorWindow对象。最终，Java 层 的 CursorWindow 对象会和此 Native 的 CursorWindow对象绑定。

提示 CursorWindow创建中涉及共享内存方面的知识，读者可上网查询或阅读卷I的7.2.2节。

至此，用于承载数据的共享内存已创建完毕，但我们还没有执行SQL的SELECT语句。这个工作由SQLiteQuery的fillWindow函数完成。

(2) SQLiteQuery fillWindow分析

前面曾说过， SQLiteQuery保存了一个Native层的sqlite3_stmt实例，那么它的fillWindow函数是否就是执行 SQL 语句后将结果信息填充到

CursorWindow中了呢？可以通过以下代码来验证。

[-->SQLiteQuery.java : fillWindow]

```
int fillWindow (CursorWindow window) {  
    mDatabase.lock (mSql) ;  
    long timeStart=SystemClock.uptimeMillis () ;  
    try{  
        acquireReference () ;//增加一次引用计数  
        try{  
            window.acquireReference () ;  
            /*  
             * 调用nativeFillWindow函数。其中，nHandle指向Native层的sqlite3实例，  
             * nStatement指向Native层的sqlite3_stmt实例，window.mWindowPtr指向  
             * Native层的CursorWindow实例，该函数最终返回这次SQL语句执行后得到的结  
             * 果  
             * 集中的记录项个数。mOffsetIndex参数的解释见下文  
            */  
            int numRows=nativeFillWindow (nHandle,  
                nStatement, window.mWindowPtr,  
                window.getStartPosition () , mOffsetIndex) ;  
            mDatabase.logTimeStat (mSql, timeStart) ;  
            return numRows ;  
        }.....finally{  
            window.releaseReference () ;  
        }  
    }finally{  
        releaseReference () ;  
        mDatabase.unlock () ;  
    }  
}
```

mOffsetIndex 和 SQL 语句的 OFFSET 参数有关，可通过一条SQL语句来认识它。

```
SELECT*FROM IMAGES LIMIT 10 OFFSET 1
```

//上面这条SQL语句的意思是从IMAGES表中查询10条记录，这10条记录的起始位置为第1条。

//也就是查询第1到第11条记录

来看 nativeFillWindow 的实现函数，其代码是：

```
[-->android_database_SQLiteQuery.cpp :  
nativeFillWindow]
```

```
static jint nativeFillWindow (JNIEnv*env, jclass clazz, jint  
databasePtr,  
    jint statementPtr, jint windowPtr, jint startPos, jint  
offsetParam) {  
    //取出Native层的实例  
    sqlite3*database=reinterpret_cast<sqlite3*>  
(databasePtr) ;  
    sqlite3_stmt*statement=reinterpret_cast<sqlite3_stmt*>  
(statementPtr) ;  
    CursorWindow*window=reinterpret_cast<CursorWindow*>  
(windowPtr) ;  
    if (offsetParam>0) {  
        //如果设置了查询的OFFSET，则需要为其绑定起始行。根据下面的设置，读者能  
        //推测出未绑定具体值的SQL语句吗？答案是：  
        //SELECT*FROM TABLE OFFSET，其中， offsetParam指明是第几个通配  
        符，  
        //startPos用于绑定到这个通配符
```

```
    int     err=sqlite3_bind_int      (  statement,      offsetParam,
startPos) ;
}.....  
//计算本次查询 (query) 返回的结果集的列数  
int numColumns=sqlite3_column_count (statement) ;  
//将SQL执行的结果保存到CursorWindow对象中  
status_t status=window->setNumColumns (numColumns) ;  
.....  
int retryCount=0;  
int totalRows=0;  
int addedRows=0;  
bool windowFull=false;  
bool gotException=false;  
//是否遍历所有结果  
const bool countAllRows= (startPos==0) ;  
//注意下面这个循环，它将遍历SQL 的结果集，并将数据取出来保存到  
CursorWindow对象中  
while (!gotException && (!windowFull||countAllRows) ) {  
    int err=sqlite3_step (statement) ;  
    if (err==SQLITE_ROW) {  
        retryCount=0;  
        totalRows+=1;  
        //windowFull变量用于标示CursorWindow是否还有足够内存。从前面的介绍  
可知，  
        //一个CursorWindow只分配了2MB的共享内存空间  
        if (startPos>=totalRows||windowFull) {  
            continue ;  
        }  
        //在共享内存中分配一行空间用于存储这一行的数据  
        status=window->allocRow () ;  
        if (status) {  
            windowFull=true ; //CursorWindow已经没有空间了  
            continue ;  
        }  
        for (int i=0 ; i<numColumns ; i++) {  
            //获取这一行记录项中各列的值
```

```
int type=sqlite3_column_type (statement, i) ;
if (type==SQLITE_TEXT) {
//如果这列中存储的是字符串，则将其取出来并通过CursorWindow的
//putString函数保存到共享内存中
const char*text=reinterpret_cast<const char*> (
sqlite3_column_text (statement, i) ) ;
size_t sizeIncludingNull=sqlite3_column_bytes ( statement,
i)
+1 ;
status=window->putString (addedRows, i, text,
sizeIncludingNull) ;
if (status) {
windowFull=true ;
break ; //CursorWindow没有足够的空间
}
}.....//处理其他数据类型
}
if (windowFull||gotException) {
window->freeLastRow () ;
}else{
addedRows+=1 ;
}
}else if (err==SQLITE_DONE) {
.....//结果集中所有行都遍历完
break ;
}else if (err==SQLITE_LOCKED||err==SQLITE_BUSY) {
//如果数据库正因为其他操作而被锁住，此处将尝试等待一段时间
if (retryCount>50) { //最多等50次，每次1秒
throw_sqlite3_exception ( env, database , "retrycount
exceeded" ) ;
gotException=true ;
}else{
usleep (1000) ;
retryCount++ ;
}
}.....
```

```
}

//重置sqlite3_stmt实例，以供下次使用
sqlite3_reset (statement) ;
.....//返回结果集中的行数
return countAllRows?totalRows : 0 ;
}
```

通过以上代码可确认，fillWindow函数实现的就是将SQL语句的执行结果填充到了CursorWindow的共享内存中。读者如感兴趣，不妨研究一下CursorWindow是如何保存结果信息的。

建议 笔者在做网络开发时常做的一件事情就是将自定义的一些类实例对象序列化到一块内存中，然后将这块内存的内容通过socket发送给一个远端进程，而远端进程再将收到的数据反序列化以得到一个实例对象。通过这种方式，远端进程就得到了一个来自发送端的实例对象。读者不妨自学序列化/反序列化相关的知识。

(3) CursorWindow分析总结

本节向读者介绍了CursorWindow相关的知识点。其实，CursorWindow就是对一块共享内存的封装。另外我们也看到了如何将执行SELECT语句后得到的结果集填充到这块共享内存中。但是这块内存现在还仅属于服务端进程，只有客户端进

程得到这块内存后，客户端才能真正获取执行SELECT后的结果。那么，客户端是何时得到这块内存的呢？让我们回到客户端进程。

3.BulkCursorToCursorAdaptor 和 CursorWrapperInner分析

客户端的工作是先创建 BulkCursorToCursorAdaptor，然后根据远端查询(query)的结果调用 BulkCursorToCursorAdaptor 的 initialize 函数。

[-->BulkCursorToCursorAdaptor.java]

```
public final class BulkCursorToCursorAdaptor extends AbstractWindowedCursor{  
    private static final String TAG="BulkCursor";  
    //mObserverBridge和ContentObserver有关，我们留到7.5节再分析  
    private SelfContentObserver mObserverBridge=new SelfContentObserver (this) ;  
    private IBulkCursor mBulkCursor;  
    private int mCount;  
    private String[]mColumns;  
    private boolean mWantsAllOnMoveCalls;  
    //initialize函数  
    public void initialize (IBulkCursor bulkCursor, int count,  
    int idIndex,  
    boolean wantsAllOnMoveCalls) {  
        mBulkCursor=bulkCursor ;  
        mColumns=null ;  
        mCount=count ;  
        mRowIdColumnIndex=idIndex ;
```

```
mWantsAllOnMoveCalls=wantsAllOnMoveCalls ; //该值为false  
}  
.....  
}
```

由以上代码可知，`BulkCursorToCursorAdaptor`仅简单保存了来自远端的信息，并没有什么特殊操作。看来客户端进程没有在上面代码的执行过程中共享内存。该工作会不会由 `CursorWrapperInner` 来完成呢？看 `ContentResolver query` 最终返回给客户端的对象的类 `CursorWrapperInner`，其代码也较简单。

[-->`ContentResolver`.java
`CursorWrapperInner`]

```
private final class CursorWrapperInner extends  
CursorWrapper{  
    private final IContentProvider mContentProvider;  
    public static final String TAG="CursorWrapperInner";  
    /*
```

`CloseGuard`类是Android dalvik虚拟机提供的一个辅助类，用于帮助开发者判断

使用它的类的实例对象是否被显式关闭（`close`）。例如，假设有一个 `CursorWrapperInner`

对象，当没有地方再引用它时，其`finalize`函数将被调用。如果之前没有调用过 `CursorWrapperInner` 的 `close` 函数，那么 `finalize` 函数 `CloseGuard` 的 `warnIsOpen`

将打印警告信息：“A resource was acquired at attached stack trace but never

released.See `java.io.Closeable` for information on avoiding resource

```
leaks."。感兴趣的读者可自行研究CloseGuard类  
*/  
private final CloseGuard mCloseGuard=CloseGuard.get () ;  
private boolean mProviderReleased ;  
CursorWrapperInner (Cursor cursor, IContentProvider icp) {  
    super (cursor) ;//调用基类的构造函数，其内部会将cursor变量保存到  
mCursor中  
    mContentProvider=icp ;  
    mCloseGuard.open ("close") ;  
}  
.....  
}
```

CursorWrapperInner的构造函数也没有去获取共享内存。别急，先看看执行query后的结果。

客户端通过Image.Media query函数，将得到一个CursorWrapperInner类型的游标对象。当然，客户端并不知道这么重要的细节，它只知道自己用的是接口类Cursor。根据前面的分析，此时客户端 通 过 这 个 游 标 对 象 可 与 服 务 端 的 CursorToBulkCursorAdaptor交互，即进程间Binder通信的通道已经打通。但是此时客户端还未拿到那块至关重要的共享内存，即进程间的数据通道还没打通。那么，数据通道是何时打通的呢？

数据通道打通的时间和lazy creation有关，即只在使用它时才打通。

4.moveToFirst函数分析

据前文的分析，客户端从Image.Media query 函数得到的游标对象，其真实类型是 CursorWrapperInner。游标对象的使用有一个特点，即必须先调用它的move家族的函数。这个家族包括moveToFirst、moveToLast等函数。为什么一定要调用它们呢？来分析最常见的moveToFirst 函数，该函数实际上由CursorWrapperInner的基类 CursorWrapper 来实现，代码如下：

[-->CursorWrapper.java : moveToFirst]

```
public boolean moveToFirst () {  
    //mCursor指向BulkCursorToCursorAdaptor  
    return mCursor.moveToFirst ();  
}
```

mCursor 成员变量的真实类型是 BulkCursorToCursorAdaptor，但其moveToFirst函数却是该类的“老祖宗”AbstractCursor实现的，代码如下：

[-->AbstractCursor.java : moveToFirst]

```
public final boolean moveToFirst () {  
    return moveToPosition (0) ; //调用moveToPosition，直接来看该函数  
}  
//moveToPosition的参数position表示将移动游标到哪一行  
public final boolean moveToPosition (int position) {
```

```
//getCount返回结果集中的行数，这个值在搭建Binder通信通道时，已经由服务端计算并返回
//给客户端了
final int getCount () ;
//mPos变量记录了当前游标的位置，该变量初值为-1
if (position>=count) {
    mPos=count ;
    return false ;
}
if (position<0) {
    mPos=-1 ;
    return false ;
}
if (position==mPos) return true ;
//onMove函数为抽象函数，由子类实现
boolean result=onMove (mPos, position) ;
if (result==false) mPos=-1 ;
else{
    mPos=position ;
    if (mRowIdColumnIndex !=-1) {
        mCurrentRowID=Long.valueOf (getLong (mRowIdColumnIndex) ) ;
    }
}
return result ;
}
```

在上边代码中，moveToPosition将调用子类实现的onMove函数。在本例中，子类就是BulkCursorToCursorAdaptor，接下来看它的onMove函数。

(1) BulkCursorToCursorAdaptor的onMove函数分析

onMove函数的代码如下：

[-->BulkCursorToCursorAdaptor.java :
onMove]

```
public boolean onMove (int oldPosition, int newPosition) {  
    throwIfCursorIsClosed () ;  
    try{  
        //mWindow的类型就是CursorWindow。第一次调用该函数， mWindow为null  
        if (mWindow==null  
            ||newPosition<mWindow.getStartPosition ()  
            ||newPosition>=mWindow.getStartPosition () +  
            mWindow.getNumRows () ) {  
            /*  
             *mBulkCurosr用于和位于服务端的IBulkCursor Bn端通信， 其getWindow函  
数  
             *将返回一个CursorWindow类型的对象。也就是说， 调用完getWindow函数后，  
             *客户端进程就得到了一个CursorWindow， 从此， 客户端和服务端之间的数据通  
道就  
             *打通了  
            */  
            setWindow (mBulkCursor.getWindow (newPosition) ) ;  
            }else if (mWantsAllOnMoveCalls) {  
                mBulkCursor.onMove (newPosition) ;  
            }  
            }.....  
            if (mWindow==null) return false ;  
            return true ;  
  
    }
```

建立数据通道的关键函数是IBulkCurosr的 getWindow。对于客户端而言， IBulkCursor Bp端

对象的类型是BulkCursorProxy，下面介绍它的getWindow函数。

(2) BulkCursorProxy的getWindow函数分析

该getWindow函数的代码如下：

[-->BulkCursorNative.java :
BulkCursorProxy : getWindow]

```
public CursorWindow getWindow ( int startPos ) throws  
RemoteException  
{  
    Parcel data=Parcel.obtain () ;  
    Parcel reply=Parcel.obtain () ;  
    try{  
        data.writeInterfaceToken (IBulkCursor.descriptor) ;  
        data.writeInt (startPos) ;  
        mRemote.transact ( GET_CURSOR_WINDOW_TRANSACTION, data,  
        reply, 0) ;  
        DatabaseUtils.readExceptionFromParcel (reply) ;  
        CursorWindow window=null ;  
        if (reply.readInt () ==1) {  
            /*  
             * 根据服务端reply包构造一个本地的CursorWindow对象，读者可自行研究  
             * newFromParcel函数，其内部会调用nativeCreateFromParcel函数以创建  
             * 一个Native的CursorWindow对象。整个过程就是笔者在前面提到的反序列化过  
             * 程  
            */  
            window=CursorWindow.newFromParcel (reply) ;  
        }  
        return window ;  
    }.....  
}
```

再来看IBulkCursor Bn端的getWindow函数，此Bn端对象的真实类型是CursorToBulk-CursorAdaptor。

(3) CursorToBulkCursorAdaptor 的 getWindow函数分析

该getWindow函数的代码如下：

[-->CursorToBulkCursorAdaptor.java :
getWindow]

```
public CursorWindow getWindow (int startPos) {  
    synchronized (mLock) {  
        throwIfCursorIsClosed () ;  
        CursorWindow window ;  
        //mCursor 是 MediaProvider query 返回的值， 其真实类型是  
        //SQLiteCursor， 满足  
        //下面的if条件  
        if (mCursor instanceof AbstractWindowedCursor) {  
            AbstractWindowedCursor windowedCursor=  
                (AbstractWindowedCursor) mCursor ;  
            //对于本例而言， SQLiteCursor已经和一个CursorWindow绑定了， 所以  
            //window的值  
            //不为空  
            window=windowedCursor.getWindow () ;  
            if (window==null) {  
                window=new CursorWindow (mProviderName, false) ;  
                windowedCursor.setWindow (window) ;  
            }  
            //调用SQLiteCursor的moveToPosition函数， 该函数前面已经分析过了，在  
            //其
```

```
//内部将触发onMove函数的调用，此处将是SQLiteDatabase的onMove函数
mCursor.moveToPosition (startPos) ;
}else{
.....
}
if (window !=null) {
window.acquireReference () ;
}
return window ;
}
}
```

服务端返回的CursorWindow对象正是之前在count函数中创建的那个CursorWindow对象，其内部已经包含了执行本次query的查询结果。

另外，在将服务端的CursorWindow传递到客户端之前，系统会调用 CursorWindow 的 writeToParcel函数进行序列化工作。读者可自行阅读 CursorWindow 的 writeToParcel 及 nativeWriteToParcel函数。

(4) SQLiteDatabase的moveToPostion函数分析

该函数由 SQLiteDatabase 的基类 AbstractCursor 实现。我们前面已经看过它的代码了，其内部的主要工作就是调用 AbstractCursor 子类（此处就是 SQLiteDatabase 自己）实现onMove函数，因此可直接看 SQLiteDatabase 的onMove函数。

[-->SQLiteDatabase.Cursor. java : onMove]

```
public boolean onMove (int oldPosition, int newPosition) {  
    if ( mWindow==null||newPosition<mWindow.getStartPosition  
( ) ||  
        newPosition>= (mWindow.getStartPosition ( ) +  
        mWindow.getNumRows ( ) ) ) {  
        fillWindow (newPosition) ;  
    }  
    return true ;  
}
```

以上代码中的if判断很重要，具体解释如下：

当 mWindow 为 空 ， 即 服 务 端 未 创 建 CursorWindow 时 （ 当 然 ， 就 本 例 而 言 ， CursorWindow早已在query时就创建好了） ， 需 调 用 fillWindow 。 该 函 数 内 部 将 调 用 clearOrCreateLocalWindow。如果CursorWindow不 存 在 ， 则 创建 一 个 CursorWindow 对 象 。 如 果 已 经 存 在 ， 则 清 空 CursorWindow 对 象 的 信 息 。

当 newPosition 小 于 上 一 次 查 询 得 到 的 CursorWindow的起始位置， 或者newPosition大于上一次查询得到的CursorWindow的最大行位置，也需调用fillWindow。由 于 此 时 CursorWindow 已 经 存 在 ， 则 clearOrCreateLocalWindow 会 调 用 它 的 clear函 数 以 清 空 之 前 保 存 的 信 息 。

调用fillWindow后将执行SQL语句，以获得正确的结果集。例如，假设上次执行query时设置了查询是从第10行开始的90条记录（即10~100行的记录），那么，当新的query若指定了从0行开始或从101行开始时，就需重新调用fillWindow，即将新的结果填充到CursorWindow中。如果新query查询的行数位于10~100之间，则无需再次调用fillWindow了。

这是服务端针对query做的一些优化处理，即当CursorWindow已经包含了所要求的数据时，就没有必要再次查询了。按理说，客户端也应该做类似的判断，以避免发起不必要的Binder请求。我们回过头来看客户端BulkCursorToCursorAdaptor的onMove函数。

[-->BulkCursorToCursorAdaptor.java :
onMove]

```
public boolean onMove (int oldPosition, int newPosition) {  
    throwIfCursorIsClosed () ;  
    try{  
        //同样，客户端也做了对应的优化处理，如果不满足if条件，客户端根本无须调用  
        //mBulkCurosr的getWindow函数，这样服务端也就不会收到对应的Binder请求了  
        if (mWindow==null  
            ||newPosition<mWindow.getStartPosition ()  
            ||newPosition>=mWindow.getStartPosition () +
```

```
mWindow.getNumRows () ) {  
    setWindow (mBulkCursor.getWindow (newPosition) ) ;  
}  
.....  
}
```

(5) moveToFirst函数分析总结

moveToFirst 及其相关的兄弟函数（如moveToLast和move等）的目的是移动游标位置到指定行。通过上面的代码分析，我们发现它的工作其实远不止移动游标位置这么简单。对于还未拥有CursorWindow的客户端来说，moveToFirst将导致客户端反序列化来自服务端的CursorWindow信息，从而使客户端和服务端之间的数据通道真正建立起来。

7.4.4 Cursor query实现分析总结

本节的知识点是除ActivityManagerService之外难度最大的。当我们回过头来看这一路分析旅程时，可能会有如下感知：

AMS的难度体现在它的游戏规则上；而query的难度体现在它所做的层层封装上（包括其创建的类和它们之间的派生关系）。从本质上说，query要做的工作其实很简单，就是将数据复制到共享内存。这份工作的内容谈不上很难，因为它既没有复杂的游戏规则，也没有较高的技术门槛。

为什么query甚至SQLiteDatabase会涉及如此多的类呢？这个问题只能由设计者来回答，但是笔者觉得其中一定还存在较大的优化余地。

7.5 Cursor close函数实现分析

虽然大部分Java程序员对主动回收资源（包括内存等）的认识普遍不如C/C++程序员，但是对于Cursor这种重型资源（不仅占用了一个文件描述符，还共享了一块2MB的内存）来说，Java程序员在编程过程中务必显式调用close函数以释放这些资源。在日常工作中，笔者已无数次碰到因为Cursor未关闭而导致Monkey测试未通过的情况了。

另外，有些同事曾问笔者，虽然没有显式调用close函数，但这个对象也没有地方再引用它了。按照Java垃圾回收机制的规矩，该对象在一定时间后会被回收。既然Cursor本身都被回收了，为什么它包含的资源（指CursorWindow）却没有被回收呢？关于这个问题，本节最后再作讨论。

建议 如果写代码时不重视资源回收，等最后出问题时再来查找泄露点的做法，会极大地增加了软件开发的成本。虽然Java提供了垃圾回收机制，但希望读者不要把资源回收的意识丢失。

下面来分析Cursor close函数，先从客户端说起。

7.5.1 客户端close的分析

客户端拿到的游标对象的真实类型是 CursorWrapperInner，其close函数的代码如下：

[-->ContentResolver.java :
CursorWrapperInner.close]

```
public void close () {  
    super.close () ;//调用基类的close函数  
    //撤销客户端进程和目标CP进程的亲密关系  
    ContentResolver.this.releaseProvider (mContentProvider) ;  
    mProviderReleased=true ;  
    if (mCloseGuard !=null) mCloseGuard.close () ;  
}
```

下面来看 CursorWrapperInner 的 基类 CursorWrapper 的 close 函数。这里必须提醒读者，后文对函数分析会频繁从基类转到子类，又从子类转到基类。造成这种局面是因为对类封装得太厉害。

[-->CursorWrapper.java : close]

```
public void close () {  
    mCursor.close () ;//mCursor指向BulkCursorToCursorAdaptor  
}
```

BulkCursorToCursorAdaptor close 的 代 码 如
下：

[-->BulkCurosrToCursorAdaptor.java : close]

```
public void close () {  
    super.close () ;//①再次调用基类close函数，该函数会释放本地创建的  
    CursorWindow资源  
    if (mBulkCursor !=null) {  
        try{  
            mBulkCursor.close () ;//②调用远端对象的close函数  
        }.....  
        finally{  
            mBulkCursor=null ;  
        }  
    }  
}
```

对于Cursor close函数来说，笔者更关注其中所包含的CursorWindow资源是如何释放的。根据以上代码中的注释可知，BulkCurosrToCursorAdaptor 的 close 调用的 基类 close函数会释放CursorWindow。

接下来来看super.close这个函数。这个close由 BulkCurosrToCursorAdaptor 的 父 类 AbstractWindowedCursor的父类AbstractCursor实现，其代码如下：

[-->AbstractCursor.java : close]

```
public void close () {  
    mClosed=true ;  
    mContentObservable.unregisterAll () ;  
    onDeactivateOrClose () ;//调用子类实现的onDeactivateOrClose函数  
}
```

onDeactivateOrClose 由 AbstractCursor 的 子类 AbstractWindowedCursor 实现，代码如下：

[-->AbstractWindowedCursor.java : onDeactivateOrClose]

```
protected void onDeactivateOrClose () {  
  
    //调用基类即AbstractCursor的onDeactivateOrClose函数  
    super.onDeactivateOrClose () ;  
    closeWindow () ;//释放CursorWindow资源  
}
```

close 函数涉及的调用居然在派生树中如此反复出现，这让人很无奈！后面还会碰到类似的做法，读者务必保持镇静。

[-->AbstractWindowedCursor.java : close]

```
protected void closeWindow () {  
    if (mWindow !=null) {  
        mWindow.close () ;//调用CurosrWindow的close函数  
        mWindow=null ;  
    }  
}
```

CursorWindow从SQLiteClosable派生，根据前面的介绍，释放SQLiteClosable代表的资源会利用引用计数来控制，这是如何实现的呢？来看代码：

[-->CursorWindow.java : close]

```
public void close () {  
    releaseReference () ;//减少一次引用计数  
}
```

releaseReference 由 CursorWindow 的 基 类 SQLiteClosable实现，其代码如下：

[-->SQLiteClosable.java : releaseReference]

```
public void releaseReference () {  
    boolean refCountIsZero=false ;  
    synchronized (this) {  
        refCountIsZero=--mReferenceCount==0 ;  
    }  
    if (refCountIsZero) {//当引用计数减为0时，就可以真正释放资源了  
        onAllReferencesReleased ( ) ;// 调 用 子 类 实 现 的  
        onAllReferencesReleased函数  
    }  
}
```

资源释放的工作由子类实现的onAllReferencesReleased完成，对CursorWindow来说，该函数的代码如下：

[-->CursorWindow.java :
onAllReferencesReleased]

```
protected void onAllReferencesReleased () {  
    dispose () ;//调用dispose  
}  
private void dispose () {  
    if (mCloseGuard !=null) {  
        mCloseGuard.close () ;  
    }  
    if (mWindowPtr !=0) {  
        recordClosingOfWindow (mWindowPtr) ;  
        //调用nativeDispose函数，该函数内部会释放Native层的CursorWindow对  
象  
        //至此，客户端获取的那块共享内存就彻底释放干净了  
        nativeDispose (mWindowPtr) ;  
        mWindowPtr=0 ;  
    }  
}
```

至此，客户端游标对象的close函数已分析完毕，不知读者看完整个流程后有何感想。

7.5.2 服务端close的分析

服务端close函数的触发是因为客户端通过IBulkCurosr close 函数发送了 Binder 请求。IBulkCurosr 的 Bn 端就是目标 CP 进程的 CursorToBulkCursorAdaptor，其close函数的代码如下：

[-->CursorToBulkCursorAdaptor.java : close]

```
public void close () {  
    synchronized (mLock) {  
        disposeLocked () ;  
    }  
}
```

[-->CursorToBulkCursorAdaptor.java :
disposeLocked]

```
private void disposeLocked () {  
    if (mCursor !=null) {  
        //注销ContentObserver，相关知识留到第8章再分析  
        unregisterObserverProxyLocked () ;  
        mCursor.close () ;//调用SQLiteCursor的close函数  
        mCursor=null;  
    }  
    closeWindowForNonWindowedCursorLocked () ;  
}
```

SQLiteCursor的close函数的代码如下：

[-->SQLiteCursor.java : close]

```
public void close () {  
    // 和客户端一样，先调用AbstractCursor的close，最后会触发  
    AbstractWindowedCursor  
    //onDeactivateOrClose函数，在那里，服务端的CursorWindow走向终结  
    super.close () ;  
    synchronized (this) {  
        mQuery.close () ;// 调用SQLiteDatabase的close，内部将释放  
        sqlite3_stmt实例  
        //调用SQLiteDirectCursorDriver的cursorClosed函数  
        mDriver.cursorClosed () ;  
    }  
}
```

至此，服务端的close函数就分析完毕。内容较简单，无须详述。

现在来回答本节最开始提出的问题，如果没有显式调用游标对象的close函数，那么该对象被垃圾回收时是否会调用close函数呢？我们将在下一节用代码来回答这个问题。

7.5.3 finalize函数分析

游标对象被回收前，其finalize函数将被调用。来看CursorWrapperInner的finalize函数，代码如下：

[-->ContentResolver. java :
CursorWrapperInner.finalize]

```
protected void finalize () throws Throwable{
try{
if (mCloseGuard !=null) {
mCloseGuard.warnIfOpen () ;//打印一句警告
}
if (!mProviderReleased&&mContentProvider !=null) {
ContentResolver.this.releaseProvider (mContentProvider) ;
}
//上边这段代码除了打印一句警告外，并没有调用close函数
}finally{
super.finalize () ;//调用基类的finalize，它会有什么特殊处理吗？
}
}
```

很可惜，我们寄予厚望的super.finalize函数也不会做出什么特殊的处理。难道Cursor-Window资源就没地方处理了？这个问题的答案如下：

客户端所持有的CursorWindow资源会在该对象执行 finalize 时被回收。读者可查看 CursorWindow 的 finalize 函数。

前面分析过，服务端的 close 函数由 BulkCurosrToCursorAdaptor 调用 IBulkCursorclose 函数触发。但 BulkCurosrToCursorAdaptor 却没有实现 finalize 函数，故 Bulk-CurosrToCursorAdaptor 被回收时，并不会触发服务端的 Cursor 释放。所以，如客户端不显式调用 close，将导致服务端进程的资源无法释放。

提示 笔者在分析 Monkey 测试失败案例时发现，导致 Monkey 失败的直接原因是 android.process.media 进程。根据上下文对 finalize 的分析可知，问题的根源其实在客户端，当客户端未显示 close 游标时，将导致 android.process.media 进程大量资源无法释放直到其最后崩溃。由于使用 MediaProvider 的客户端较多（包括 Music、Gallery3D、Video 等），所以每次出现这种问题时，都需要所有 MediaProvider 的客户端开发者协助调查。

7.5.4 Cursor close函数总结

Cursor close函数并未涉及什么较难的知识点。希望通过这一节介绍的知识点帮助读者建立及时回收资源的意识。虽然多写了数行代码，但是这几行代码对维护整个系统的稳定性是非常重要的。

提示 另外，通过对close函数的分析，读者也见识了过度封装的坏处，即在派生树中反复寻找对应函数的实现。虽然并非代码设计者故意如此，但是继承关系过多就很容易产生这种情况。目前笔者也没有找到完美的解决办法，欢迎广大读者参与讨论。

7.6 ContentResolver openAssetFileDescriptor函数分析

通过对Cursor query的分析可知，客户端进程可像查询本地数据库那样，从目标CP进程获取信息。不过，这种方法也有其局限性：

客户端只能按照结果集的组织方式来获取数据，而结果集的组织方式是行列式的，即客户端须移动游标到指定行，才能获取自己感兴趣的列的值。在实际生活中，不是所有信息都能组织成行列的格式。

query查询得到的数据的数据量很有限。通过分析可知，用于承载数据的共享内存只有2MB大小。对于较大数据量的数据，通过query方式来获取显然不合适。

考虑到query的局限性，ContentProvider还支持另外一种更直接的数据传输方式，笔者称之为“文件流方式”。因为通过这种方式客户端将得到一个类似文件描述符的对象，然后在其上创建对应的输入或输出流对象。这样，客户端就可通过它们和CP进程交互数据了。

下面来分析这个功能是如何实现的。先向读者介绍客户端使用的函数，即CR的openAssetFileDescriptor。

7.6.1 openAssetFileDescriptor之客户端调用分析

这部分的代码如下：

[-->ContentResolver.java :
openAssetFileDescriptor]

```
public final AssetFileDescriptor openAssetFileDescriptor
(Uri uri,
 String mode) throws FileNotFoundException{
 //openAssetFileDescriptor是一个通用函数，它支持3种scheme类型的
URI。见
//下文的解释
String scheme=uri.getScheme() ;
if (SCHEME_ANDROID_RESOURCE.equals (scheme) ) {
if (! "r".equals (mode) ) {
throw new FileNotFoundException ("Can't write
resources："+uri) ;
}
//创建资源文件输入流
OpenResourceIdResult r=getResourceId (uri) ;
try{
return r.r.openRawResourceFd (r.id) ;
}.....
}else if (SCHEME_FILE.equals (scheme) ) {
//创建普通文件输入流
```

```
ParcelFileDescriptor pfd=ParcelFileDescriptor.open ( new File (uri.getPath () ) , modeToMode (uri, mode) ) ;  
return new AssetFileDescriptor (pfd, 0, -1) ;  
}else{  
if ("r".equals (mode) ) {  
//我们重点分析这个函数，用于读取目标CP的数据  
return openTypedAssetFileDescriptor (uri, "*/*", null) ;  
}.....//其他模式的支持，请读者学完本章后自行研究这部分内容  
}  
}
```

如以上代码所述，openAssetFileDescriptor是一个通用函数，它支持3种sheme类型的URI。

SCHEME_ANDROID_RESOURCE：字符串表达为android.resource。通过它可以读取APK包（其实就是一个压缩文件）中封装的资源。假设在应用进程的res/raw目录下存放一个test.ogg文件，最终生成的资源ID由R.raw.tet来表达，那么如果应用进程想要读取这个资源，创建的URI就是 android.resource : //com.package.name/R.raw.test。读者不妨试一试。

SCHEME_FILE：字符串表达为file。通过它可以读取普通文件。

除上述两种scheme之外的URI：这种资源背后到底对应的是什么数据需要由目标CP来解释。

接下来分析在第3种sheme类型下调用的openTypedAssetFileDescriptor函数。

1.openTypedAssetFileDescriptor函数分析

这部分的代码如下：

[-->ContentResolver.java :
openTypedAssetFileDescriptor]

```
public final AssetFileDescriptor
openTypedAssetFileDescriptor (Uri uri,
    String mimeType, Bundle opts) throws FileNotFoundException{
    //建立和目标CP进程交互的通道。读者还记得此处provider的真实类型吗？
    //其真实类型是ContentProviderProxy
    IContentProvider provider=acquireProvider (uri) ;
    try{
        //①调用远端CP的openTypedAssetFile函数，返回值的
        //类型是AssetFileDescriptor。此处传递参数的值为：mimeType="*/*"
        //opts=null
        AssetFileDescriptor fd=provider.openTypedAssetFile (uri,
            mimeType, opts) ;
        .....
        //②创建一个ParcelFileDescriptor类型的变量
        ParcelFileDescriptor pfd=new ParcelFileDescriptorInner (
            fd.getParcelFileDescriptor () , provider) ;
        provider=null;
        //③又创建一个AssetFileDescriptor类型的变量作为返回值
        return new AssetFileDescriptor (pfd, fd.getStartOffset () ,
            fd.getDeclaredLength () ) ;
    }.....
    finally{
        if (provider !=null) {
            releaseProvider (provider) ;
```

```
    }  
}  
}
```

在以上代码中，阻碍我们思维的依然是新出现的类。先应解决它们，然后再分析代码中的关键调用。

2.FileDescriptor家族介绍

本节涉及的类家族图谱如图7-7所示。

图7-7中的内容比较简单，只需稍作介绍。

FileDescriptor类是Java的标准类，它是对文件描述符的封装。进程打开的每一个文件都有一个对应的文件描述符。在Native语言开发中，它用一个int型变量来表示。

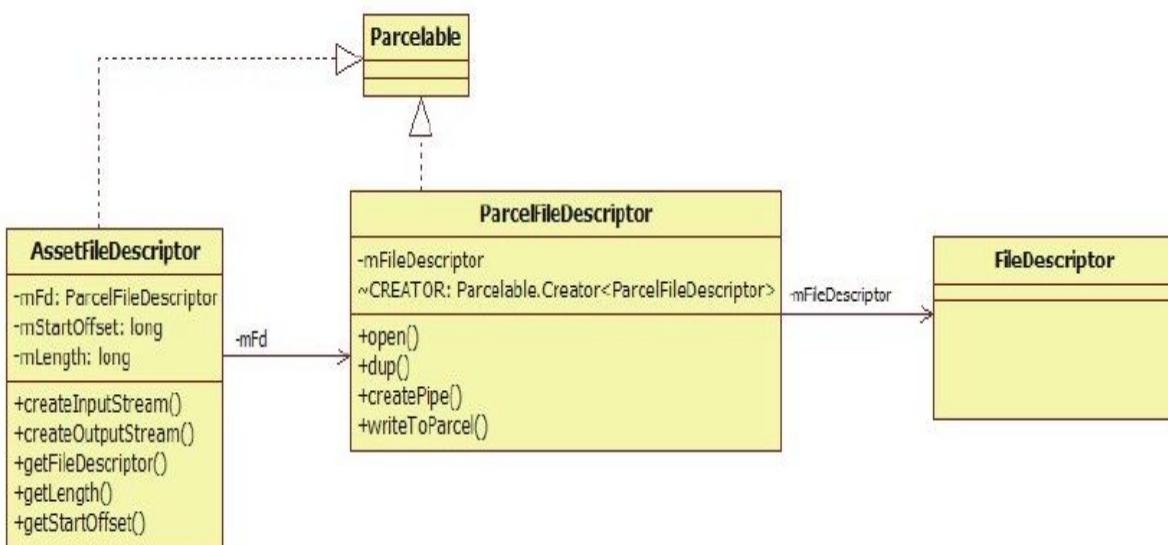


图 7-7 FileDescriptor家族图谱

文件描述符作为进程的本地资源，如想越过进程边界将其传递给其他进程，则需借助进程间共享技术。在Android平台上，设计者封装了一个ParcelFileDescriptor类。此类实现了Parcel接口，自然就支持了序列化和反序列化的功能。从图7-7可知，一个ParcelFileDescriptor通过mFileDescriptor指向一个文件描述符。

AssetFileDescriptor也实现了Parcel接口，其内部通过mFd成员变量指向一个ParcelFileDescriptor对象。从这里可看出，AssetFileDescriptor是对ParcelFileDescriptor类的进一步封装和扩展。实际上，根据SDK文档中对AssetFileDescriptor的描述可知，其作用在于从AssetManager（后续分析资源管理的时候会介绍它）中读取指定的资源数据。

提 示 简 单 向 读 者 介 绍 一 下 与 AssetFileDescriptor相关的知识。它用于读取APK包中指定的资源数据。以前面提到的test.ogg为例，如果通过AssetFileDescriptor读取它，那么其mFd成员则指向一个ParcelFileDescriptor对象。且不管这个对象是否跨越了进程边界，它毕竟代表一个文件。假设这个文件是一个APK包，AssetFileDescriptor的mStartOffset变量用于指明

test.ogg在这个APK包中的起始位置，比如100字节。而mLength用于指明test.ogg的长度，假设是1000字节。通过上面的介绍可知，该APK文件从100字节到1100字节这一段空间中存储的就是test.ogg的数据。这样，AssetFileDescriptor就能将test.ogg数据从APK包中读取出来了。

下面来看ContentProvider的openTypedAssetFile函数。

7.6.2 ContentProvider的openTypedAssetFile函数分析

这部分的代码具体如下：

[-->ContentProvider.java : openTypedAssetFile]

```
public AssetFileDescriptor openTypedAssetFile ( Uri uri,
String mimeTypeFilter,
Bundle opts) throws FileNotFoundException{
//本例满足下面的if条件
if ("/*/*".equals (mimeTypeFilter) )
return openAssetFile (uri, "r") ;//此函数的代码见下文
String baseType=getType (uri) ;
if (baseType !=null&&
ClipDescription.compareMimeTypes
( baseType,
mimeTypeFilter) ) {
return openAssetFile (uri, "r") ;
}
throw new FileNotFoundException ("Can't open"+
uri+"as type"+mimeTypeFilter) ;
}
```

[-->ContentProvider.java : openAssetFile]

```
public AssetFileDescriptor openAssetFile ( Uri uri, String
mode)
throws FileNotFoundException{
//openFile由子类实现。这里还以MediaProvider为例
```

```
ParcelFileDescriptor fd=openFile (uri, mode) ;  
//根据openFile返回的fd得到一个AssetFileDescriptor对象  
return fd !=null?new AssetFileDescriptor (fd, 0, -1) :null;  
}
```

下面分析MediaProvider实现的openFile函数。

1.MediaProvider openFile分析

这部分的代码如下：

[-->MediaProvider.java : openFile]

```
public ParcelFileDescriptor openFile (Uri uri, String mode)  
throws FileNotFoundException{  
ParcelFileDescriptor pfd=null;  
//假设URI符合下面的if条件，即客户端想读取的是某音乐文件所属专辑  
(Album) 的信息  
if (URI_MATCHER.match (uri) ==AUDIO_ALBUMART_FILE_ID) {  
DatabaseHelper database=getDatabaseForUri (uri) ;  
.....  
SQLiteDatabase db=database.getReadableDatabase () ;  
.....  
SQLiteQueryBuilder qb=new SQLiteQueryBuilder () ;  
//得到客户端指定的代表该音乐文件的_id值  
int songid=Integer.parseInt (uri.getPathSegments () .get  
(3) ) ;  
qb.setTables ("audio_meta") ;  
qb.appendWhere ("_id="+songid) ;  
Cursor c=qb.query (db,  
new String[]{  
MediaStore.Audio.Media.DATA,  
MediaStore.Audio.Media.ALBUM_ID},  
null, null, null, null, null) ;
```

```
if (c.moveToFirst () ) {  
    String audiopath=c.getString (0) ;  
    //获取该音乐所属的albumid值  
    int albumid=c.getInt (1) ;  
    //注意，下面函数中调用的ALBUMART_URI将指向album_art表  
    Uri newUri=ContentUris.withAppendedId ( ALBUMART_URI,  
    albumid) ;  
    try{  
        //调用CP实现的openFileHelper函数。注意，pfd的  
        //类型是ParcelFileDescriptor  
        pfd=openFileHelper (newUri, mode) ;  
    }.....  
    }  
    c.close () ;  
    return pfd ;  
}  
.....  
}
```

在以上代码中，MediaProvider将首先通过客户端指定的音乐文件的_id去查询它的专辑信息。此处给读者一个示例，如图7-8所示。

```
sqlite> select _id,album_id,_data from audio_meta;  
329|44|/mnt/sdcard/ringtones/愤怒的小鸟.mp3  
346|1|/mnt/sdcard/Music/Secret Garden/White Stones/01 Steps.mp3  
347|1|/mnt/sdcard/Music/Secret Garden/White Stones/02 Poeme.mp3  
348|1|/mnt/sdcard/Music/Secret Garden/White Stones/03 Hymn To Hope.mp3  
349|1|/mnt/sdcard/Music/Secret Garden/White Stones/04 Moving.mp3  
350|1|/mnt/sdcard/Music/Secret Garden/White Stones/05 First Day Of Spring.mp3  
351|1|/mnt/sdcard/Music/Secret Garden/White Stones/06 Passacaglia.mp3
```

图 7-8 audio_meta内容展示

图 7-8 中设置的 SQL 语句是 select_id, album_id, _data from audio_meta，得到的结果集

包含：第一列音乐文件的_id值，第二列返回音乐文件所属专辑的album_id值，第三列返回对应歌曲的文件存储路径。

以上代码在调用openFileHelper函数前构造了一个新的URI变量，根据代码中的注释可知，它将查询album_art表，不妨再看一个示例，如图7-9所示。

```
sqlite> select _data,album_id from album_art;
/mnt/sdcard/Android/data/com.android.providers.media/albumthumbs/1318716108346|12
/mnt/sdcard/Android/data/com.android.providers.media/albumthumbs/1318716109894|24
/mnt/sdcard/Android/data/com.android.providers.media/albumthumbs/1318716113564|35
/mnt/sdcard/Android/data/com.android.providers.media/albumthumbs/1318947766485|49
/mnt/sdcard/Android/data/com.android.providers.media/albumthumbs/1318947766649|51
```

图 7-9 album_art内容展示

在图7-9中，结果集的第一列为专辑艺术家的缩略图文件存储路径，第二列为专辑艺术家album_id值。所以，要打开的文件就是对应album_id的缩略图。再来看openFileHelper的代码。

2.ContentProvider openFileHelper函数分析

这部分的代码如下：

[-->ContentProvider.java : openFileHelper]

```
protected final ParcelFileDescriptor openFileHelper ( Uri  
uri,
```

```
String mode) throws FileNotFoundException{
    //获取缩略图的文件路径
    Cursor c=query (uri, new String[]{"_data"} , null, null,
null) ;
    int count= (c !=null) ?c.getCount () :0 ;
    if (count !=1) {
        .....//一个album_id只能对应一个缩略图文件
    }
    c.moveToFirst () ;
    int i=c.getColumnIndex ("_data") ;
    String path= (i>=0?c.getString (i) :null) ;
    c.close () ;
    if (path==null)
        throw new FileNotFoundException ("Column_data not found.") ;
    int modeBits=ContentResolver.modeToMode (uri, mode) ;
    //创建ParcelFileDescriptor对象，内部会首先打开一个文件以得到
    //一个FileDescriptor对象，然后再创建一个ParcelFileDescriptor对
象，其实
    //就是设置ParcelFileDescriptor中成员变量mFileDescriptor的值
    return  ParcelFileDescriptor.open ( new File ( path ) ,
modeBits) ;
}
```

至此，服务端已经打开指定文件了。那么，这个服务端的文件描述符是如何传递到客户端的呢？我们单起一节来回答这个问题。

7.6.3 跨进程传递文件描述符的探讨

在回答上一节的问题之前，不知读者是否思考过下面的问题：实现文件描述符跨进程传递的目的是什么？

以上节读取音乐专辑的缩略图为例，问题的答案就是，让客户端能够读取专辑的缩略图文件。为什么客户端不先获得对应专辑缩略图的文件存储路径，然后直接打开这个文件，却要如此大费周章呢？原因有二：

出于安全的考虑，MediaProvider不希望客户端绕过它去直接读取存储设备上的文件。另外，客户端须额外声明相关的存储设备读写权限，然后才能直接读取其上面的文件。

虽然本例针对的是一个实际文件，但是从可扩展性角度看，我们希望客户端使用一个更通用的接口，通过这个接口可读取实际文件的数据，也可读取来自的网络的数据，而作为该接口的使用者无须关心数据到底从何而来。

提示 实际上还有更多的原因，读者不妨尝试在以上两点原因的基础上拓展思考。

1.序列化ParcelFileDescriptor

继续讨论本例的情况。现在服务端已经打开了某个缩略图文件，并且获得了一个文件描述符对象FileDescriptor。这个文件是服务端打开的。如何让客户端也打开这个文件呢？根据前文分析，客户端不会也不应该通过文件路径自己去打开这个文件。那该如何处理？

没关系，Binder驱动支持跨进程传递文件描述符。先来看ParcelFileDescriptor的序列化函数writeToParcel，代码如下：

[-->ParcelFileDescriptor.java : writeToParcel]

```
public void writeToParcel (Parcel out, int flags) {  
    //往Parcel包中直接写入mFileDescriptor指向的FileDescriptor对象  
    out.writeFileDescriptor (mFileDescriptor) ;  
    if ( (flags&PARCELABLE_WRITE_RETURN_VALUE) !=0 && !mClosed)  
{  
    try{  
        close () ;  
    }.....  
    }  
}
```

Parcel 的 writeFileDescriptor 是一个 native 函数，代码如下：

[-->android_util_Binder.cpp :
android_os_Parcel_writeFileDescriptor]

```
static void android_os_Parcel_writeFileDescriptor
(JNIEnv*env,
 jobject clazz, jobject object)
{
Parcel*parcel=parcelForJavaObject (env, clazz) ;
if (parcel !=NULL) {
//先调用jniGetFDFromFileDescriptor从Java层FileDescriptor对象中
//取出对应的文件描述符。在Native层，文件描述符是一个int整型
//然后调用Native parcel对象的writeDupFileDescriptor函数
const status_t err=
parcel->writeDupFileDescriptor (
jniGetFDFromFileDescriptor (env, object) ) ;
if (err !=NO_ERROR) {
signalExceptionForError (env, clazz, err) ;
}
}
}
```

Native Parcel类的writeDupFileDescriptor函数
的代码如下：

[-->Parcel.cpp : writeDupFileDescriptor]

```
status_t Parcel::writeDupFileDescriptor (int fd)
{
return writeFileDescriptor (dup (fd) , true) ;
}
//直接来看writeFileDescriptor函数
status_t Parcel : writeFileDescriptor ( int fd, bool
takeOwnership)
```

```
{  
flat_binder_object obj ;  
obj.type=BINDER_TYPE_FD ;  
obj.flags=0x7f|FLAT_BINDER_FLAG_ACCEPTS_FDS ;  
obj.handle=fd ; //将MediaProvider打开的文件描述符传递给Binder协议  
obj.cookie= (void*) (takeOwnership?1:0) ;  
return writeObject (obj, true) ;  
}
```

由上边代码可知，ParcelFileDescriptor的序列化过程就是将其内部对应文件的文件描述符取出，并存储到一个由 Binder 驱动的 flat_binder_object 对象中。该对象最终会发送给 Binder 驱动。

2. 反序列化ParcelFileDescriptor

假设客户端进程收到了来自服务端的回复，客户端要做的就是根据服务端的回复包构造一个新的ParcelFileDescriptor。我们重点关注文件描述符的反序列化，其中调用的函数是 Parcel 的 readFileDescriptor，其代码如下：

[-->ParcelFileDescriptor. java :
readFileDescriptor]

```
public final ParcelFileDescriptor readFileDescriptor () {  
//从internalReadFileDescriptor中返回一个FileDescriptor对象  
FileDescriptor fd=internalReadFileDescriptor () ;
```

```
//构造一个ParcelFileDescriptor对象，该对象对应的文件就是服务端打开的那个缩略图文件
```

```
return fd != null ? new ParcelFileDescriptor(fd) : null;
```

internalReadFileDescriptor是一个native函数，其实现代码如下：

[-->android_util_Binder.cpp :
android_os_Parcel_readFileDescriptor]

```
static jobject android_os_Parcel_readFileDescriptor  
(JNIEnv* env, jobject clazz)  
{  
    Parcel* parcel=parcelForJavaObject(env, clazz) ;  
    if (parcel != NULL) {  
        //调用Parcel的readFileDescriptor得到一个文件描述符  
        int fd=parcel->readFileDescriptor() ;  
        if (fd<0) return NULL;  
        fd=dup(fd) ;//调用dup复制该文件描述符  
        if (fd<0) return NULL;  
        //调用jniCreateFileDescriptor以返回一个Java层的FileDescriptor对  
象  
        return jniCreateFileDescriptor(env, fd) ;  
    }  
    return NULL ;  
}
```

来看Parcel的readFileDescriptor函数，代码如
下：

[-->Parcel.cpp : readFileDescriptor]

```
int Parcel::readFileDescriptor () const
{
const flat_binder_object*flat=readObject (true) ;
if (flat) {
switch (flat->type) {
case BINDER_TYPE_FD :
//当服务端发送回复包的时候，handle变量指向fd。当客户端接收到回复包的时候，
//又从handle中得到fd。此fd是彼fd吗？
return flat->handle ;
}
return BAD_TYPE ;
}
```

笔者在以上代码中提到了一个较深刻的问题：此fd是彼fd吗？这个问题的真实含义是：

服务端打开了一个文件，得到了一个fd。注意，fd是一个整型。在服务端上，这个fd确实对应了一个已经打开的文件。

客户端得到的也是一个整型值，它对应的是一个文件吗？

如果说客户端得到一个整型值，就认为它得到了一个文件，这种说法未免有些草率。在以上代码中，我们发现客户端确实根据收到的那个整型值创建了一个FileDescriptor对象。那么，怎样才可知道这个整型值在客户端中一定代表一个文件呢？

这个问题的终极解答在Binder驱动的代码中。来看它的binder_transaction函数。

3.文件描述符传递之Binder驱动的处理

这部分的代码如下：

[-->binder.c : binder_transaction]

```
static void binder_transaction (struct binder_proc*proc,
    struct binder_thread*thread,
    struct binder_transaction_data*tr, int reply)
.....
switch (fp->type) {
case BINDER_TYPE_FD : {
    int target_fd;
    struct file*file;
    if (reply) {
        .....
        //Binder驱动根据服务端返回的fd找到内核中文件的代表file，其数据类型是
        //struct file
        file=fget (fp->handle) ;
        .....
        //target_proc为客户端进程，task_get_unused_fd_flags函数用于从客
        //户端
        //进程中找一个空闲的整型值，用作客户端进程的文件描述符
        target_fd=task_get_unused_fd_flags (target_proc,
        O_CLOEXEC) ;
        .....
        //将客户端进程的文件描述符和代表文件的file对象绑定
        task_fd_install (target_proc, target_fd, file) ;
        fp->handle=target_fd ;
    }break ;
    ....//其他处理
}
```

}

一切真相大白！原来，Binder驱动代替客户端打开了对应的文件，所以现在可以肯定，客户端收到的整型值确确实实代表一个文件。

4. 深入讨论

在研究这段代码时，笔者曾经向所在团队同仁问过这样一个问题：在Linux平台上，有什么办法能让两个进程共享同一文件的数据呢？曾得到下面这些回答：

两个进程打开同一文件。这种方式前面讨论过了，安全性和可扩展性都比较差，不是我们想要的方式。

通过父子进程的亲缘关系，使用文件重定向技术。由于这两个进程关系太亲近，这种实现方式拓展性较差，也不是我们想要的。

跳出两个进程打开同一个文件的限制。在两个进程间创建管道，然后由服务端读取文件数据并写入管道，再由客户端进程从管道中获取数据。这种方式和前面介绍的openAssetFileDescriptor有殊途同归之处。

在缺乏类似Binder驱动支持的情况下，要在Linux平台上做到文件描述符的跨进程传递是件比较困难的事。从上面3种回答来看，最具扩展性的是第三种方式，即进程间采用管道作为通信手段。但是对Android平台来说，这种方式的效率显然不如现有的openAssetFileDescriptor的实现。原因在于管道本身的特性。

服务端必须单独启动一个线程来不断地往管道中写入数据，即整个数据的流动是由写端驱动的（虽然当管道无空间的时候，如果读端不读取数据，写端也没法再写入数据，但是如果写端不写数据，则读端一定读不到数据。基于这种认识，笔者认为管道中数据流动的驱动力应该在写端）。

Android 3.0以后为CP提供了管道支持，我们来看相关的函数。

[-->ContentProvider.java : openPipeHelper]

```
public<T>ParcelFileDescriptor openPipeHelper ( final Uri
uri,
final String mimeType, final Bundle opts,
final T args, final PipeDataWriter<T>func)
throws FileNotFoundException{
try{
//创建管道
final ParcelFileDescriptor[] fds=ParcelFileDescriptor.
```

```
createPipe () ;  
//构造一个AsyncTask对象  
AsyncTask<Object, Object, Object>task=new  
AsyncTask<Object, Object, Object> () {  
@Override  
protected Object doInBackground (Object.....params) {  
//往管道写端写数据，如果没有这个后台线程的写操作，客户端无论如何  
//也读不到数据的  
func.writeDataToPipe (fds[1], uri, mimeType, opts, args) ;  
try{  
fds[1].close () ;  
}.....  
return null ;  
}  
};  
// AsyncTask.THREAD_POOL_EXECUTOR 是一个线程池， task 的  
doInBackground  
//函数将在线程池中的一个线程中运行  
task.executeOnExecutor (AsyncTask.THREAD_POOL_EXECUTOR,  
(Object[]) null) ;  
return fds[0] ;//返回读端给客户端  
}.....  
}
```

由以上代码可知，采用管道这种方式的开销
确实比客户端直接获取文件描述符的开销大。

7.6.4 openAssetFileDescriptor函数分析总结

本节讨论了CP提供的第二种数据共享方式，即文件流方式。通过这种方式，客户端可从目标CP中获取想要的数据。

本节所涉及的内容较杂，从Java层一直到驱动层。希望读者能认真体会，务必把其中的原理搞清楚。

7.7 本章学习指导

本章围绕CP进行了较为深入的讨论。相比而言，CP的启动和创建的难度及重要性要小，而SQLite的使用、进程间的数据共享和传递等就复杂得多。建议读者阅读完本章后，能在此基础上，对下列问题做进一步的深入研究：

客户端进程是如何撤销它和目标CP进程之间的紧密关系的。

尝试自行封装一个轻量级的、面向对象的SQLite类库。

针对序列化和反序列相关的知识，最好能编写一些简单的例子，用于实践。

Java程序员应阅读《高质量Java程序设计》一书，树立良好的资源管理和回收意识。

最好再深入阅读脚注中提到的参考书《Pattern-Oriented Software Architecture Volume 3 : Patterns for Resource Management》。

7.8 本章小结

本章围绕CP进行了较为深入细致的讨论。首先介绍了目标CP进程是如何启动并创建CP实例的；接着又介绍了SQLite及SQLiteDatabase家族；最后三节分别围绕Cursor的query、close函数的实现及文件描述符跨进程传递的内容进行了专题讨论。

第8章 深入理解ContentService和AccountManagerService

本章主要内容：

介绍ContentService

介绍AccountManagerService

本章所涉及的源代码文件名及位置：

SystemServer. java
(frameworks/base/services/java/com/android/server/
SystemServer.java)

ContentService. java
(frameworks/base/core/java/android/content/Content
Service.java)

ContentResolver. java
(frameworks/base/core/java/android/content/Content
Resolver.java)

UsbSettings. java
(packages/apps/Settings/src/com/android/settings/de
viceinfo/UsbSettings.java)

DevelopmentSettings. java
(packages/apps/Settings/src/com/android/settings/DevelopmentSettings.java)

UsbDeviceManager. java
(frameworks/base/services/java/com/android/server/usb/UsbDevice-Manager.java)

AccountManagerService. java
(frameworks/base/core/java/android/accounts/AccountManagerService.java)

AccountAuthenticatorCache. java
(frameworks/base/core/java/android/accounts/AccountAuthenticatorCache.java)

RegisteredServicesCache. java
(frameworks/base/core/java/android/content/pm/RegisteredServices-Cache.java)

AccountManager. java
(frameworks/base/core/java/android/accounts/AccountManager.java)

EasAuthenticatorService
(packages/apps>Email/src/com/android/email/service/EasAuthenticator-Service.java)

SyncManager. java
(frameworks/base/core/java/android/content/SyncM
anager.java)

SyncStorageEngine. java
(frameworks/base/core/java/android/content/SyncSt
orageEngine.java)

SyncAdapterCache. java
(frameworks/base/core/java/android/content/SyncA
dapterCache.java)

SyncQueue. java
(frameworks/base/core/java/android/content/SyncQ
ueue.java)

EmailSyncAdapterService. java
(packages/apps/Exchange/src/com/android/exchang
e/EmailSync-AdapterService.java)

AbstractThreadedSyncAdapter. java
(frameworks/base/core/java/android/content/Abstrac
t-ThreadedSyncAdapter.java)

8.1 概述

本章第一个要分析的是ContentService。ContentService包含以下两个主要功能：

它是Android平台中数据更新通知的执行者。数据更新通知与第7章分析Cursorquery函数实现时提到的ContentObserver有关。这部分内容将在8.2节中介绍。

它是Android平台中数据同步服务的管理中心。当用户通过Android手机中的Contacts应用将联系人信息同步到远端服务器时，就需要和ContentService交互。这部分内容是本章的难点，将在8.4节中介绍。

本 章 第 二 个 要 分 析 的 是 AccountManagerService，它负责管理Android手机中用户的账户，这些账户是用户的online账户，例如用户在Google、Facebook上注册的账户。

本章将先分析ContentService中数据通知机制的实现，然后分析AccountManager-Service，最后再介绍ContentService中的数据同步服务。

提示 这两个Service的难度都不大，它们在设计结构上有较大的相似性，在内容上也有一定的关联。另外，作为本书最后一章，笔者照例会留一些难度适中的问题或知识点供读者自行分析研究。

8.2 数据更新通知机制分析

何为数据更新通知？先来看日常生活中的一个例子。

笔者所在公司采用BugZilla来管理Bug。在日常工作中，笔者和同事们的一部分工作就是登录BugZilla查询各自名下的Bug并修改它们。如何跟踪自己的Bug呢？其实，以上描述中已经提到了一种方法，即登录BugZilla并查询。除此之外，BugZilla还支持另一种方法，即为每个Bug设置一个关系人列表，一旦该Bug的状态发生变化，BugZilla就会给该Bug关系人列表中的人发送邮件。

上例中提到的第二种方法就是本节要分析的数据更新通知机制。一般说来，领导和项目经理（PM）使用第一种方法的居多，因为他们需要不定时地查询和统计全局Bug的情况。而程序员使用第二种方法较多（也许是没办法的事情吧，谁会情愿主动查询自己的Bug呢？）。

类似的通知机制在日常生活中的其他地方还有使用。在操作系统中，这种通知机制同样也广泛存在。例如，在OS中，设计人员一般会安排外

部设备以中断的方式通知CPU并让其开展后续处理，而不会让CPU去轮询外设的状态。

现在回到Android平台，如果程序需要监控某数据项的变化，可以采用一个类似while循环的语句不断查询它以判断其值是否发生了变化。显而易见，这种方式的效率很低。有了通知机制以后，程序只需注册一个ContentObserver实例即可。一旦该项数据发生变化，系统就会通过ContentObserver的onChange函数来通知我们。与前面所述的轮询相比，此处的通知方式明显更高效。

通过上面的描述可以知道，通知机制的实施包括两个步骤：第一步，注册观察者；第二步，通知观察者。在Android平台中，这两步都离不开ContentService，下面来认识一下它。

提示 在设计模式中，通知机制对应的模式是Observer模式，即观察者模式。

8.2.1 初识ContentService

SystemServer创建ContentService的代码非常简单，如下所示：

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

```
public void run () {  
    ....  
    ContentService.main (context,  
    factoryTest==SystemServer.FACTORY_TEST_LOW_LEVEL) ;  
    ....  
}
```

以上代码中直接调用了ContentService的main函数。在一般情况下，该函数第二个参数为false。此main函数的代码如下：

[-->ContentService.java : main]

```
public static IContentService main (Context context, boolean  
factoryTest) {  
    //构造ContentService实例  
    ContentService service=new ContentService ( context,  
factoryTest) ;  
    //将ContentService注册到ServiceManager中，其注册名叫content  
    ServiceManager.addService  
(ContentResolver.CONTENT_SERVICE_NAME,  
    service) ;  
    return service ;  
}
```

ContentService的构造函数的代码如下：

[-->ContentService.java : ContentService]

```
ContentService (Context context, boolean factoryTest) {  
    mContext=context ;  
    mFactoryTest=factoryTest ;  
    getSyncManager () ;//获取同步服务管理对象，接下来看它的代码  
}
```

[-->ContentService.java : getSyncManager]

```
private SyncManager getSyncManager () {  
    synchronized (mSyncManagerLock) {  
        try{  
            //创建一个SyncManager实例，它是ContentService中负责数据同步服务的  
            //主力军。留待8.4节再详细分析它  
            if (mSyncManager==null) mSyncManager=new  
                SyncManager (mContext, mFactoryTest) ;  
            }.....  
            return mSyncManager ;  
        }  
    }
```

看完以上代码读者可能会觉得ContentService比较简单。其实，ContentService中最难实现的功能是数据同步服务，不过该功能的实现都封装在SyncManager及相关类中了，所以在分析通知机制时不会和数据同步服务有太多瓜葛。

下面来分析通知机制实施的第一步，即注册ContentObserver。该步骤由ContentResolver提供的registerContentObserver函数来实现。

8.2.2 ContentResolver的registerContentObserver分析

这部分的代码如下：

```
[-->ContentResolver.java :  
registerContentObserver]
```

```
public final void registerContentObserver (Uri uri,  
boolean notifyForDescendents, ContentObserver observer) {
```

注意registerContentObserver传递的参数，其中：

uri是客户端设置的它需要监听的数据项的地址，用uri来表示

notifyForDescendents：如果该值为true，则所有地址包含此uri的数据项发生变化时

都会触发通知；否则只有完全符合该uri地址的数据项发生变化时才会触发通知。
以文件夹和

其中的文件为例，若uri指向某文件夹，则需设置notifyForDescendents为true，即该文件

夹下的任何文件发生变化，都需要通知监听者。

observer是客户端设置的监听对象。当数据项发生变化时，该对象的onChange函数将被调用

```
*/
```

```
try{
```

```
/*
```

调用ContentService的registerContentObserver函数，其第三个参数是
observer.getContentObserver的返回值，它是什么呢？

```
*/
```

```
getContentService () .registerContentObserver (uri,  
notifyForDescendents,  
observer.getContentObserver () ) ;
```

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

registerContentObserver 最终将调用 ContentService 的 registerContentObserver 函数，其中第三个参数是 ContentObserver getContentObserver 的返回值。这个返回值是什么呢？要搞明白这个问题，需请出 ContentObserver 家族成员。

1.ContentObserver介绍

ContentObserver家族成员如图8-1所示。

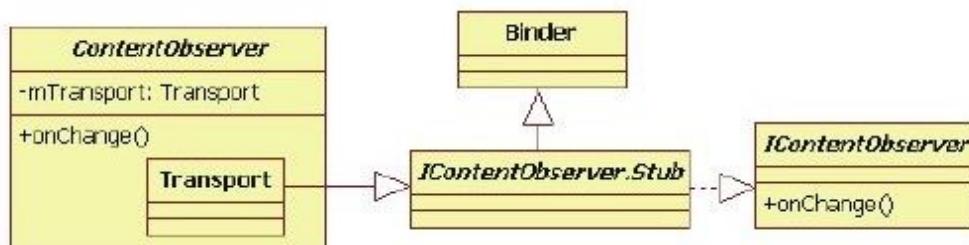


图 8-1 ContentObserver家族类图

图8-1中的ContentObserver类和第7章中介绍的 ContentProvider类非常类似，内部都定义了一个 Transport 类参与 Binder 通信。由图 8-1 可知， Transport类从IContentObserver.Stub派生。从Binder 通信角度来看，客户端进程中的Transport将是Bn 端。如此，通过 registerContentObserver 传递到 ContentService 所在进程的就是 Bp 端。

IContentObserver Bp 端对 象 的 真 实 类 型 是 IContentObserver.Stub.Proxy。

注 意 IContentObserver.java 由 aidl 处理 IContentObserver.aidl 生成，其所在位置为 out/targer/common/obj/JAVA_LIBRARIES/framework_intermediates/src/core/java/android/database/IContentObserver.java。

2.registerContentObserver函数分析

下 面 来 看 ContentService 的 registerContentObserver函数的代码。

[-->ContentService. java :
registerContentObserver]

```
public void registerContentObserver ( Uri uri, boolean
notifyForDescendents,
IContentObserver observer) {
.....
synchronized (mRootNode) {
//ContentService要做的事情其实很简单，就是保存uri和observer的对应
关系到
//其内部变量mRootNode中
mRootNode.addObserverLocked ( uri, observer,
notifyForDescendents,
mRootNode, Binder.getCallingUid () ,
Binder.getCallingPid () );
}
```

mRootNode是ContentService的成员变量，其类型为ObserverNode。ObserverNode的组织形式是数据结构中的树，其叶子节点的类型为ObserverEntry，它保存了uri和对应的IContentObserver对象。本节不关注它们的内部实现，读者若有兴趣，不妨自行研究。

至此，客户端已经为某数据项设置了ContentObserver。再来看通知机制实施的第二步，即通知观察者。

8.2.3 ContentResolver的notifyChange分析

数据更新的通知由ContentResolver的notifyChange函数触发。看MediaProvider的update函数的代码如下：

[-->MediaProvider.java : update]

```
public int update ( Uri uri, ContentValues initialValues,
String userWhere,
String[]whereArgs) {
int count ;
int match=URI_MATCHER.match (uri) ;
DatabaseHelper database=getDatabaseForUri (uri) ;
//找到对应的数据库对象
SQLiteDatabase db=database.getWritableDatabase () ;
.....
synchronized (sGetTableAndWhereParam) {
getTableAndWhere (uri, match, userWhere,
sGetTableAndWhereParam) ;
switch (match) {
.....
case VIDEO_MEDIA :
case VIDEO_MEDIA_ID : {
ContentValues values=new ContentValues (initialValues) ;
values.remove (ImageColumns.BUCKET_ID) ;
values.remove (ImageColumns.BUCKET_DISPLAY_NAME) ;
.....//调用SQLiteDatabase的update函数更新数据库
count=db.update (sGetTableAndWhereParam.table, values,
sGetTableAndWhereParam.where, whereArgs) ;
.....
```

```
.....//其他处理
} //switch语句结束
}.....//synchronized处理结束
if (count>0 && ! db.inTransaction ()) //调用notifyChange触发通知
    getContext () .getContentResolver () .notifyChange ( uri,
null) ;
return count ;
}
```

由以上代码可知，MediaProvider update函数更新完数据库后，将通过notfiyChange函数来通知观察者。notfiyChange函数的代码如下：

[-->ContentResolver.java : notifyChange]

```
public void notifyChange ( Uri uri, ContentObserver
observer) {
    //在一般情况下，observer参数为null。调用另一个notifyChange函数，直接来看它
    notifyChange (uri, observer, true) ;
}
public void notifyChange ( Uri uri, ContentObserver
observer,
boolean syncToNetwork) {
    //第三个参数syncToNetwork用于控制是否需要发起一次数据同步请求
    try{
        //调用ContentService的notifyChange函数
        getContentService () .notifyChange (
            uri, observer==null?null:observer.getContentObserver () ,
            observer !=null&&observer.deliverSelfNotifications () ,
            syncToNetwork) ;
    }.....
}
```

由以上代码可知，ContentService 的 notifyChange 函数将被调用，其代码如下：

[-->ContentServic : notifyChange]

```
public void notifyChange ( Uri uri, IContentObserver
observer,
    boolean observerWantsSelfNotifications, boolean
syncToNetwork) {
    long identityToken=clearCallingIdentity () ;
    try{
        ArrayList<ObserverCall>calls=new ArrayList<ObserverCall>
()
;
        //从根节点开始搜索需要通知的观察者，结果保存在calls数组中
        synchronized (mRootNode) {
            mRootNode.collectObserversLocked (uri, 0, observer,
                observerWantsSelfNotifications, calls) ;
        }
        final int numCalls=calls.size () ;
        for (int i=0 ; i<numCalls ; i++) {
            ObserverCall oc=calls.get (i) ;
            try{
                /*
                    调用客户端IContentObserver Bn端，即ContentObserver
                    内部类Transport的onChange函数。最后再由Transport调用
                    客户端提供的ContentObserver子类的onChange函数
                */
                oc.mObserver.onChange (oc.mSelfNotify) ;
            }.....//异常处理
        }
        if (syncToNetwork) {
            SyncManager syncManager=getSyncManager () ;
            if (syncManager !=null) {
                //发起一次同步请求，相关内容留待8.4节分析
            }
        }
    }
}
```

```
    syncManager.scheduleLocalSync (null,  
        uri.getAuthority () ) ;  
    }  
}  
}  
}finally{  
    restoreCallingIdentity (identityToken) ;  
}  
}  
}
```

8.2.4 数据更新通知机制总结和深入探讨

总结上面所描述的数据更新通知机制的流程，如图8-2所示。

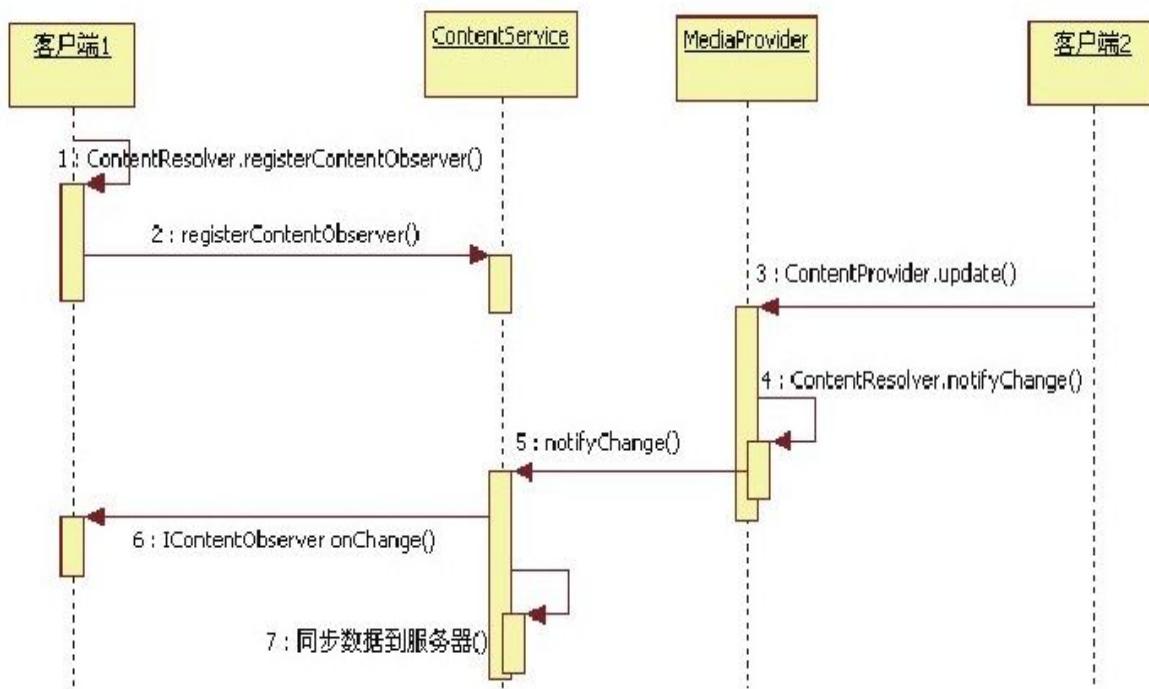


图 8-2 数据更新通知的流程图

从前面的代码介绍和图8-2所示的流程来看，Android平台中的数据更新通知机制较为简单。不过此处尚有几个问题想和读者一起探讨。

问题一：由图8-2可知，客户端2调用ContentProvider的update函数将间接触发客户端1的ContentObserver的onChange函数被调用。如果客

客户端1在onChange函数中耗时过长，会不会导致客户端2阻塞在update函数中呢？

想到这个问题的读者应该是非常细致和认真的了。确实，从前面所示的代码和流程图来看，这个情况几乎是必然会发生，但是实际上该问题并不存在，原因在下面这一段代码中：

[-->IContentObserver.java : Proxy :
onChange]

```
private static class Proxy implements android.database.IContentObserver{
    private android.os.IBinder mRemote;
    .....
    public void onChange (boolean selfUpdate)
        throws android.os.RemoteException{
        android.os.Parcel_data=android.os.Parcel.obtain () ;
        try{
            _data.writeInterfaceToken (DESCRIPTOR) ;
            _data.writeInt ( ( (selfUpdate) ? (1) : (0) ) ) ;
            //调用客户端1的ContentObserver Bn端的onChange函数
            mRemote.transact (Stub.TRANSACTION_onChange, _data, null,
                android.os.IBinder.FLAG_ONEWAY) ;
        }finally{
            _data.recycle () ;
        }
    }
    .....
}
```

以上代码告诉我们，ContentService在调用客户端注册的IContentObserver的onChange函数时，使用了FLAG_ONEWAY标志。根据第2章对该标志的介绍（参见2.2.1节），使用该标志的Binder调用只需将请求发给Binder驱动即可，无须等待客户端onChange函数的返回。因此，即使客户端1在onChange中恶意浪费时间，也不会阻塞客户端2的update函数了。

问题二：假设服务端有一项功能，需要客户端通过某种方式来控制它的开闭（即禁止或使用该功能），考虑一下有几种方式能实现这个控制机制？这是一个开放性问题，需要集思广益。

Android平台上至少有3种方法可以实现这个控制机制。

第一种：服务端实现一个API函数，客户端直接调用这个函数来控制。

第二种：客户端发送指定的广播，而服务端注册该广播的接收者，然后在这个广播接收者的onReceive函数中处理。

第三种：服务端输出一个ContentProvider，并为这个功能输出一个uri地址，然后注册一个ContentObserver。客户端可通过更新数据的方式

来触发服务端ContentObserver的onChange函数，服务端在该函数中做对应处理即可。

在Android代码中，这三种方法都有使用。下面将以Settings应用中和USB相关的功能设置为例来观察第一种和第三种方法的使用情况。

第一个实例和Android 4.0中新支持的USB MTP/PTP功能有关，相关代码如下：

[-->UsbSettings.java : onPreferenceTreeClick]

```
public boolean onPreferenceTreeClick ( PreferenceScreen  
preferenceScreen,  
Preference preference) {  
....  
if (preference==mMtp) {  
mUsbManager.setCurrentFunction (UsbManager.USB_FUNCTION_MTP,  
true) ;  
updateToggles (UsbManager.USB_FUNCTION_MTP) ;  
}else if (preference==mPtp) {  
mUsbManager.setCurrentFunction (UsbManager.USB_FUNCTION_PTP,  
true) ;  
updateToggles (UsbManager.USB_FUNCTION_PTP) ;  
}  
return true ;  
}
```

由以上代码可知，如果用户从Settings界面中选择了使能MTP，将直接调用UsbManager的

setCurrentFunction来使能MTP功能。这个函数的Bn端实现在UsbService中。

不过，同样是USB相关的功能控制，ADB的开关控制却采用了第三种方法，相关代码为：

[-->DevelopmentSettings.java : onClick]

```
public void onClick (DialogInterface dialog, int which) {  
    if (which==DialogInterface.BUTTON_POSITIVE) {  
        mOkClicked=true ;  
        //设置Settings数据库ADB对应的数据项值为1  
        Settings.Secure.putInt ( getActivity () .getContentResolver()  
        () ,  
        Settings.Secure.ADB_ENABLED, 1) ;  
    }else  
        mEnableAdb.setChecked ( false ) ;//界面更新  
}
```

上面的数据项更新操作将导致UsbDeviceManager做对应处理，其相关代码如下：

[-->UsbDeviceManager.java : onChange]

```
private class AdbSettingsObserver extends ContentObserver{  
    .....  
    public void onChange (boolean selfChange) {  
        //从数据库中取出对应项的值  
        boolean enable= (Settings.Secure.getInt (mContentResolver,  
        Settings.Secure.ADB_ENABLED, 0) >0) ;  
        //发送MSG_ENABLE_ADB消息，UsbDeviceManager将处理此消息
```

```
mHandler.sendMessage (MSG_ENABLE_ADB, enable) ;  
}  
}
```

同样是USB相关的功能，Settings应用却采用了两种截然不同的方法来处理它们。这种做法为笔者目前所从事的项目中USB扩展功能的实现带来了极大困扰，因为我们想采用统一的方法来处理USB相关功能。到底应采用哪种方法比较合适呢？第一种方法和第三种方法各自的适用场景是什么？读者不妨仔细思考并将结论与大家分享。

我们在第7章中分析Cursor query函数时曾看到过ContentObserver的身影，但是并没有对其进行详细分析。如果现在回过头去分析query函数流程中和ContentObserver相关的部分，则所涉及的流程可能比本节内容还要多。

8.3 AccountManagerService分析

本节将分析AccountManagerService。如前所述，AccountManagerService负责管理手机中用户的online账户，主要工作涉及账户的添加和删除、AuthToken（全称为authentication token。有了它，客户端就无须每次操作都向服务器发送密码了）的获取和更新等。关于AccountManagerService更详细的功能，可阅读SDK文档中AccountManager的说明。

8.3.1 初识AccountManagerService

下面看AccountManagerService创建时的代码：

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

```
....  
//注册AccountManagerService到ServiceManager，服务名为account  
ServiceManager.addService (Context.ACCOUNT_SERVICE,  
new AccountManagerService (context) ) ;
```

其构造函数的代码如下：

[-->AccountManagerService.java :
AccountManagerService]

```
public AccountManagerService (Context context) {  
    // 调用另外一个构造函数，其第三个参数将构造一个  
    AccountAuthenticatorCache对象，它是  
    //什么呢？见下文分析  
    this (context, context.getPackageManager () ,  
    new AccountAuthenticatorCache (context) ) ;  
}
```

在AccountManagerService构造函数中创建了一个AccountAuthenticatorCache对象，它是什么？来看下文。

1.AccountAuthenticatorCache分析

AccountAuthenticatorCache是Android平台中账户验证服务（Account Authenticator Service, AAS）的管理中心。AAS由应用程序通过在AndroidManifest.xml中输出符合指定要求的Service信息而来。稍后读者将看到这些要求的具体格式。

先来看AccountAuthenticatorCache的派生关系，如图8-3所示。

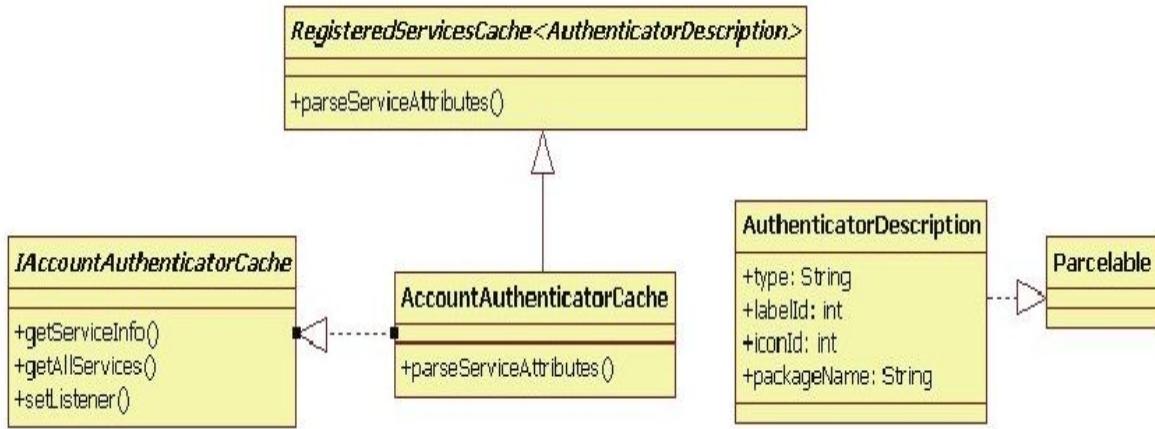


图 8-3 AccountAuthenticatorCache类图

由图8-3可知：

从

AccountAuthenticatorCache
 RegisteredServicesCache<AuthenticatorDescription>派生。RegisteredServicesCache是一个模板类，专门用于管理系统中指定Service的信息收集和更新，而具体是哪些Service，则由RegisteredServicesCache构造时的参数指定。AccountAuthenticatorCache对外输出是由RegisteredServicesCache模板参数指定的类的实例。在图8-3中就是AuthenticatorDescription。

AuthenticatorDescription继承了Parcelable接口，这代表它可以跨Binder传递。该类描述了AAS相关的信息。

AccountAuthenticatorCache实现了IAccountAuthenticatorCache接口。这个接口供外部

调用者使用以获取AAS的信息。

下面看AccountAuthenticatorCache的创建，其相关代码如下：

[-->AccountAuthenticatorCache.java :
AccountAuthenticatorCache]

```
public AccountAuthenticatorCache (Context context) {  
    /*  
     ACTION_AUTHENTICATOR_INTENT 值  
     为"android.accounts.AccountAuthenticator"  
     AUTHENTICATOR_META_DATA_NAME 值  
     为"android.accounts.AccountAuthenticator"  
     AUTHENTICATOR_ATTRIBUTES_NAME值为"account-authenticator"  
    */  
    super (context,  
        AccountManager.ACTION_AUTHENTICATOR_INTENT,  
        AccountManager.AUTHENTICATOR_META_DATA_NAME,  
        AccountManager.AUTHENTICATOR_ATTRIBUTES_NAME,  
        sSerializer) ;  
}
```

AccountAuthenticatorCache 调 用 其 基 类 RegisteredServicesCache的构造函数时，传递了3个字符串参数，这3个参数用 于 控 制 RegisteredServicesCache 从 PackageManagerService 获取哪些Service的信息。

(1) RegisteredServicesCache分析

[-->RegisteredServicesCache.java
RegisteredServicesCache]

```
public RegisteredServicesCache ( Context context, String
interfaceName,
    String metaDataName, String attributeName,
    XmlSerializerAndParser<v>serializerAndParser) {
    mContext=context ;
    //保存传递进来的参数
    mInterfaceName=interfaceName ;
    mMetaDataName=metaDataName ;
    mAttributesName=attributeName ;
    mSerializerAndParser=serializerAndParser ;
    File dataDir=Environment.getDataDirectory () ;
    File systemDir=new File (dataDir, "system") ;
    //syncDir指向/data/system/registered_service目录
    File syncDir=new File (systemDir, "registered_services") ;
    // 下 面 这 个 文 件 指 向 syncDir 目 录 下 的
    android.accounts.AccountAuthenticator.xml
    mPersistentServicesFile=new AtomicFile (new File (syncDir,
    interfaceName+".xml") ) ;
    //生成服务信息
    generateServicesMap () ;
    final BroadcastReceiver receiver=new BroadcastReceiver () {
        public void onReceive (Context context1, Intent intent) {
            generateServicesMap () ;
        }
    } ;
    //注册Package安装、卸载和更新等广播监听者
    mReceiver=new AtomicReference<BroadcastReceiver>
    (receiver) ;
    IntentFilter intentFilter=new IntentFilter () ;
    intentFilter.addAction (Intent.ACTION_PACKAGE_ADDED) ;
    intentFilter.addAction (Intent.ACTION_PACKAGE_CHANGED) ;
    intentFilter.addAction (Intent.ACTION_PACKAGE_REMOVED) ;
```

```
intentFilter.addDataScheme ("package") ;  
mContext.registerReceiver (receiver, intentFilter) ;  
IntentFilter sdFilter=new IntentFilter () ;  
sdFilter.addAction  
(Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE) ;  
sdFilter.addAction  
(Intent.ACTION_EXTERNAL_APPLICATIONS_UNAVAILABLE) ;  
mContext.registerReceiver (receiver, sdFilter) ;  
}
```

由以上代码可知：

成员变量 mPersistentServicesFile 指向 /data/system/registered_service 目录下的一个文件，该文件保存了以前获取的对应 Service 的信息。就 AccountAuthenticator 而言，mPersistentServicesFile 指向该目录的 android.accounts.AccountAuthenticator.xml 文件。

由于 RegisteredServicesCache 管理的是系统中指定 Service 的信息，当系统中有 Package 安装、卸载或更新时，RegisteredServicesCache 也需要对应更新自己的信息，因为有些 Service 可能会随着 APK 被删除而不复存在。

generateServiceMap 函数将获取指定的 Service 信息，其代码如下：

[--> RegisteredServicesCache.java :
generateServicesMap]

```
void generateServicesMap () {
    //获取PackageManager接口，用来和PackageManagerService交互
    PackageManager pm=mContext.getPackageManager () ;
    ArrayList<ServiceInfo<V>>serviceInfos=new           ArrayList<
    ServiceInfo<V>> () ;
    /*
    在 本 例 中 ，   查 询   PKMS   中   满   足   Intent   Action
    为"android.accounts.AccountAuthenticator"
    的服务信息。由以下代码可知，这些信息指的是Service中声明的MetaData信息
    */
    List<ResolveInfo>resolveInfos=pm.queryIntentServices (
        new          Intent          (          mInterfaceName          )          ,
        PackageManager.GET_META_DATA) ;
    for (ResolveInfo resolveInfo : resolveInfos) {
        try{
        /*
        调用parseServiceInfo函数解析从PKMS中获得的MetaData信息，该函数
        返回的是一个模板类对象。就本例而言，这个函数返回一个
        ServiceInfo<AccountAuthenticator>类型的对象
        */
        ServiceInfo<V>info=parseServiceInfo (resolveInfo) ;
        serviceInfos.add (info) ;
    }
    }
    synchronized (mServicesLock) {
        if (mPersistentServices==null)
            readPersistentServicesLocked () ;
        mServices=Maps.newHashMap () ;
        StringBuilder changes=new StringBuilder () ;
        .....//检查mPersistentServices保存的服务信息和当前从PKMS中取出来的
        PKMS
        //信息，判断是否有变化，如果有变化，需要通知监听者。读者可自行阅读这段
        代码，
        //注意其中uid的作用
        mPersistentServicesFileNotExist=false ;
```

```
 }  
 }
```

接下来解析Service的parseServiceInfo函数。

(2) parseServiceInfo函数分析

这部分的代码如下：

[-->RegisteredServicesCache.java :
parseServiceInfo]

```
private ServiceInfo<V>parseServiceInfo ( ResolveInfo  
service)  
throws XmlPullParserException, IOException{  
    android.content.pm.ServiceInfo si=service.serviceInfo ;  
    ComponentName componentName=new ComponentName  
(si.packageName, si.name) ;  
    PackageManager pm=mContext.getPackageManager () ;  
    XmlResourceParser parser=null ;  
    try{  
        //解析MetaData信息  
        parser=si.loadXmlMetaData (pm, mMetaDataName) ;  
        AttributeSet attrs=Xml.asAttributeSet (parser) ;  
        int type ;  
        .....  
        String nodeName=parser.getName () ;  
        //调用子类实现的parseServiceAttributes得到一个真实的对象，在本例中  
它是  
        //AuthenticatorDescription。注意，传递给parseServiceAttributes  
的第一个  
        //参数代表MetaData中的resource信息。详细内容见下文Email的图例  
        V v=parseServiceAttributes (  
            pm.getResourcesForApplication (si.applicationInfo) ,
```

```
si.packageName, attrs) ;  
final android.content.pm.ServiceInfo  
serviceInfo=service.serviceInfo ;  
final ApplicationInfo  
applicationInfo=serviceInfo.applicationInfo ;  
final int uid=applicationInfo.uid ;  
return new ServiceInfo<v> (v, componentName, uid) ;  
}.....finally{  
if (parser !=null) parser.close () ;  
}  
}
```

parseServiceInfo将解析Service中的MetaData信息，然后调用子类实现的parseService-Attributes函数，以获取特定类型Service的信息。

下面通过实例向读者展示最终的解析结果。

(3) AccountAuthenticatorCache分析总结

在Email应用的AndroidManifest.xml中定义一个AAS，如图8-4所示。

```
<service  
    android:name=".service.EasAuthenticatorService"  
    android:exported="true"  
    android:enabled="true"  
>  
<intent-filter>  
    <action  
        android:name="android.accounts.AccountAuthenticator" />  
</intent-filter>  
<meta-data  
    android:name="android.accounts.AccountAuthenticator"  
    android:resource="@xml/eas_authenticator"  
    />  
</service>
```

图 8-4 Email AAS定义

由图8-4可知，在Email中这个Service对应为EasAuthenticatorService，其Intent匹配的Action为 android.accounts.AccountAuthenticator，其 MetaData 的 name 为 android.accounts.AccountAuthenticator，而 MetaData的具体信息保存在resource资源中，在本例中，它指向另外一个XML文件，即 eas_authenticator.xml，此文件的内容如图8-5所示。

```
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"  
    android:accountType="com.android.exchange"  
    android:icon="@drawable/ic_exchange_selected"  
    android:smallIcon="@drawable/ic_exchange_minitab_selected"  
    android:label="@string/exchange_name"  
    android:accountPreferences="@xml/account_preferences"  
/>
```

图 8-5 eas_authenticator.xml的内容

这个XML文件中的内容是有严格要求的，其中：

accountType标签用于指定账户类型（账户类型和具体应用有关，Android并未规定账户的类型）。

icon、smallIcon、label和accountPreferences等用于界面显示。例如，当需要用户输入账户信息时，系统会弹出一个Activity，上述几个标签就用

于界面显示。详细情况可阅读SDK文档AbstractAccountAuthenticator的说明。

android.accounts.AccountAuthenticator.xml的内容如图8-6所示。

```
shell@android:/data/system/registered services # ll
-rw----- system system 223 2012-04-13 14:16 android.accounts.AccountAuthenticator.xml
-rw----- system system 1149 2012-02-04 10:28 android.content.SyncAdapter.xml
ounts.AccountAuthenticator.xml
<
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<services>
<service uid="10015" type="com.android.exchange" />
<service uid="10015" type="com.android.email" />
<service uid="10047" type="com.google" />
</services>
```

图 8-6
android.accounts.AccountAuthenticator.xml的内容

由图8-6可知，笔者的测试机器上有3个AAS服务，其中同一个uid有两个服务（即uid为10015对应的两个Service）。

提示 uid是在为PackageManagerService解析APK文件时赋予APK的。读者不妨自行阅读frameworks/base/services/java/com/android/server/pm/Settings.java中的newUserIdLPw函数。

下面来看AccountManagerService的构造函数。

2.AccountManagerService构造函数分析

AccountManagerService 构造函数的代码如下：

[-->AccountManagerService.java :
AccountManagerService]

```
public AccountManagerService ( Context context,  
PackageManager packageManager,  
IAccountAuthenticatorCache authenticatorCache) {  
mContext=context ;  
mPackageManager=packageManager ;  
synchronized (mCacheLock) {  
//此数据库文件对应为 /data/system/accounts.db  
mOpenHelper=new DatabaseHelper (mContext) ;  
}  
mMessageThread=new HandlerThread ("AccountManagerService") ;  
mMessageThread.start () ;  
mMessageHandler=new MessageHandler (mMessageThread.getLooper  
) ;  
mAuthenticatorCache=authenticatorCache ;  
//为 AccountAuthenticatorCache 设置一个监听者，一旦 AAS 服务发生变化，  
//AccountManagerService 需要做对应处理  
mAAuthenticatorCache.setListener (this, null/*Handler*/) ;  
sThis.set (this) ;  
// 监听 ACTION_PACKAGE_REMOVED 广播  
IntentFilter intentFilter=new IntentFilter () ;  
intentFilter.addAction (Intent.ACTION_PACKAGE_REMOVED) ;  
intentFilter.addDataScheme ("package") ;  
mContext.registerReceiver (new BroadcastReceiver () {  
public void onReceive (Context context1, Intent intent) {  
purgeOldGrants () ;  
}  
}, intentFilter) ;  
/*
```

accounts.db数据库中有一个grants表，用于存储授权信息，该信息用于保存有权限获取账户信息的Package。下面的函数将根据grants表中的数据查询PKMS判断这些

Package是否还存在。如果系统中已经不存在这些Package，则grants表需要更新

```
*/
```

```
purgeOldGrants () ;
```

```
/*
```

accounts.db中有一个accounts表，该表中存储了账户类型和账户名。其中，账户类型

就是AuthenticatorDescription中的accountType，它和具体应用有关。下面这个

函数将比较accounts表中的内容与AccountAuthenticatorCache中服务的信息，如果

AccountAuthenticatorCache已经不存在对应账户类型的服务，则需要删除accounts表

中的对应项

```
*/
```

```
validateAccountsAndPopulateCache () ;
```

```
}
```

AccountManagerService的构造函数较简单，有兴趣的读者可自行研究以上代码中未详细分析的函数。

8.3.2 AccountManager addAccount分析

这一节将分析AccountManagerService中的一个重要的函数，即addAccount，其功能是为某项账户添加一个用户。下面以前面提及的Email为例来认识AAS的处理流程。

AccountManagerService 是一个运行在SystemServer中的服务，客户端进程必须借助AccountManager 提供的 API 来使用AccountManagerService 服务，所以，本例需从AccountManager的addAccount函数讲起。

1.AccountManager的addAccount发起请求

AccountManager 的 addAccount 函数的参数和返回值较复杂，先看其函数原型：

```
public AccountManagerFuture<Bundle>addAccount (
    final String accountType,
    final String authTokenType,
    final String[] requiredFeatures,
    final Bundle addAccountOptions,
    final Activity activity,
    AccountManagerCallback<Bundle>callback,
    Handler handler)
```

在以上代码中：

addAccount 的返回值类型是 AccountManagerFuture<Bundle>。其中， AccountManager-

Future是一个模板Interface，其真实类型只有在分析addAccount的实现时才能知道。现在可以告诉读者的是，它和Java并发库（concurrent库）中的FutureTask有关，是对异步函数调用的一种封装^[1]。调用者在后期只要调用它的getResult函数即可取得addAccount的调用结果。由于addAccount可能涉及网络操作（例如，AAS需要把账户添加到网络服务器上），所以这里采用了异步调用的方法以避免长时间的阻塞。这也是AccountManagerFuture的getResult不能在主线程中调用的原因。

addAccount的第一个参数accountType代表账户类型。该参数不能为空。就本例而言，它的值为com.android.email。

authTokenType、requiredFeatures 和 addAccountOptions与具体的AAS服务有关。如果想添加指定账户类型的Account，则须对其背后的AAS有所了解。

activity：此参数和界面有关。例如有些AAS需要用户输入用户名和密码，故需启动一个Activity。在这种情况下，AAS会返回一个Intent，客户端将通过这个Activity启动Intent所标示的Activity。我们将通过下文的分析介绍这一点。

callback和handler：这两个参数与如何获取addAccount返回结果有关。如这两个参数为空，客户端则须单独启动一个线程去调用AccountManagerFuture的getResult函数。addAccount的代码如下：

[-->AccountManager.java：addAccount]

```
public AccountManagerFuture<Bundle>addAccount (final String accountType,
final String authTokenType, final String[]requiredFeatures,
final Bundle addAccountOptions, final Activity activity,
AccountManagerCallback<Bundle>callback, Handler handler) {
if (accountType==null) //accountType不能为null
throw new IllegalArgumentException ("accountType is null") ;
final Bundle optionsIn=new Bundle () ;
if ( addAccountOptions ! =null ) //保存客户端传入的
addAccountOptions
optionsIn.putAll (addAccountOptions) ;
optionsIn.putString (KEY_ANDROID_PACKAGE_NAME,
mContext.getPackageName () ) ;
//构造一个匿名类对象，该类继承自AmsTask，并实现了dowork函数。
addAccount返回前
//将调用该对象的start函数
return new AmsTask (activity, handler, callback) {
```

```
public void doWork () throws RemoteException{  
    //mService用于和AccountManagerService通信  
    mService.addAccount (mResponse, accountType, authTokenType,  
    requiredFeatures, activity !=null, optionsIn) ;  
}  
}.start () ;  
}
```

在以上代码中，AccountManager的addAccount函数将返回一个匿名类对象，该匿名类继承自AmsTask类。那么，AmsTask又是什么呢？

(1) AmsTask介绍

先来看AmsTask的继承关系，如图8-7所示。

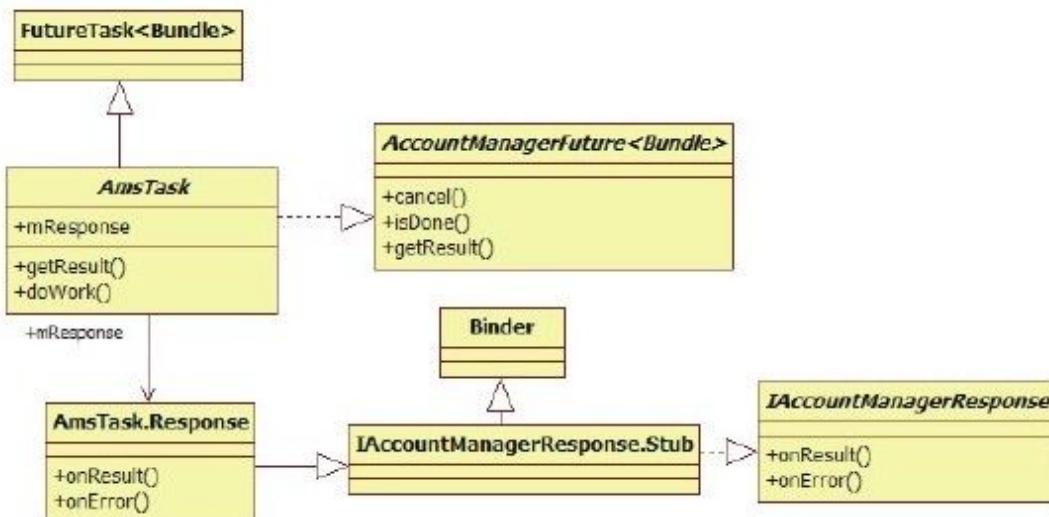


图 8-7 AmsTask继承关系

由图8-7可知：

AmsTask 继承自 FutureTask，并实现了 AccountManagerFuture 接口。FutureTask 是 Java concurrent 库中一个常用的类。AmsTask 定义了一个 doWork 虚函数，该函数必须由子类来实现。

一个 AmsTask 对象中有一个 mResponse 成员，该成员的类型是 AmsTask 中的内部类 Response。从 Response 的派生关系可知，Response 将参与 Binder 通信，并且它是 Binder 通信的 Bn 端。而 AccountManagerService 的 addAccount 将得到它的 Bp 端对象。当添加完账户后，AccountManagerService 会通过这个 Bp 端对象的 onResult 或 onError 函数向 Response 通知处理结果。

(2) AmsTask 匿名类处理分析

AccountManager 的 addAccount 最终返回给客户端的是一个 AmsTask 的子类，首先来了解它的构造函数，其代码如下：

[--> AccountManager.java : AmsTask]

```
public AmsTask (Activity activity, Handler handler,
AccountManagerCallback<Bundle>callback) {
.....//调用基类构造函数
//保存客户端传递的参数
mHandler=handler ;
mCallback=callback ;
mActivity=activity ;
```

```
mResponse=new Response () ;//构造一个Response对象，并保存到  
mResponse中  
}
```

下一步调用的是这个匿名类的start函数，代码如下：

[-->AccountManager.java : AmsTask.start]

```
public final AccountManagerFuture<Bundle>start () {  
try{  
dowork () ;//调用匿名类实现的dowork函数  
}.....  
return this ;  
}
```

匿名类实现的doWork函数即下面这个函数：

[-->AccountManager.java : addAccount 返回
的匿名类]

```
public void dowork () throws RemoteException{  
//调用AccountManagerService的addAccount函数，其第一个参数是  
mResponse  
mService.addAccount (mResponse, accountType, authTokenType,  
requiredFeatures, activity!=null, optionsIn) ;  
}
```

AccountManager的addAccount函数的实现比较新奇，它内部使用了Java的concurrent类。不熟悉Java并发编程的读者有必要了解相关知识。

下面转到 AccountManagerService 中去分析 addAccount 的实现。

2.AccountManagerService addAccount 转发请求

AccountManagerService addAccount 的代码如下：

[-->AccountManagerService.java :
addAccount]

```
public void addAccount ( final IAccountManagerResponse response,
    final String accountType, final String authTokenType,
    final String[] requiredFeatures, final boolean expectActivityLaunch,
    final Bundle optionsIn) {
    .....
    //检查客户端进程是否有“android.permission.MANAGE_ACCOUNTS”的权限
    checkManageAccountsPermission () ;
    final int pid=Binder.getCallingPid () ;
    final int uid=Binder.getCallingUid () ;
    //构造一个Bundle类型的options变量，并保存传入的optionsIn
    final Bundle options= ( optionsIn==null ) ?new Bundle () :
    optionsIn ;
    options.putInt ( AccountManager.KEY_CALLER_UID, uid) ;
    options.putInt ( AccountManager.KEY_CALLER_PID, pid) ;
    long identityToken=clearCallingIdentity () ;
    try{
        //创建一个匿名类对象，该匿名类派生自Session类。最后调用该匿名类的bind
        函数
```

```
new Session ( response, accountType, expectActivityLaunch,
true) {
    public void run () throws RemoteException{
        mAuthenticator.addAccount (this, mAccountType,
        authTokenType, requiredFeatures, options) ;
    }
    protected String toDebugString (long now) {
        .....//实现toDebugString函数
    }
    }.bind () ; }finally{
    restoreCallingIdentity (identityToken) ;
}
}
```

由以上代码可知，AccountManagerService的addAccount函数最后也创建了一个匿名类对象，该匿名类派生自Session。addAccount最后还调用了这个对象的bind函数。其中最重要的内容就是Session。那么，Session又是什么呢？

(1) Session介绍

Session家族成员如图8-8所示。

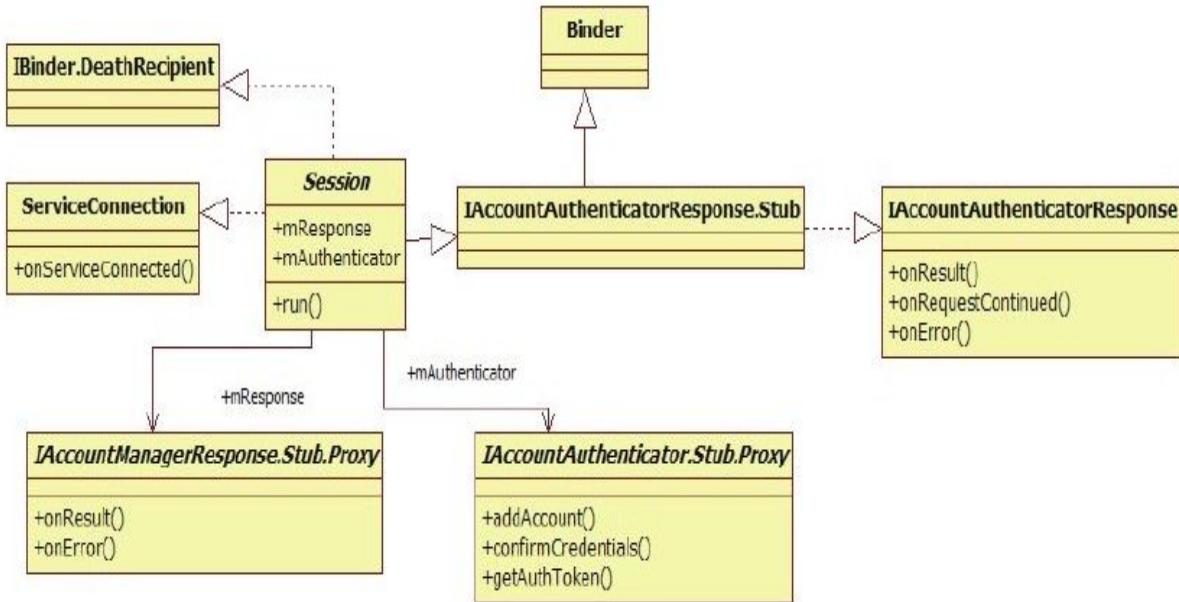


图 8-8 Session家族示意图

由图8-8可知：

Session 从 IAccountAuthenticatorResponse.Stub 派生，这表明它将参与Binder通信，并且它是Bn端。那么这个Binder通信的目标是谁呢？它正是具体的AAS服务。Account-ManagerService会将自己传递给AAS，这样，AAS就得 to IAccountAuthenticator-Response 的 Bp 端对象。当 AAS 完成了具体的账户添加工作后，会通过 IAccount-AuthenticatorResponse 的 Bp 端对象向 Session 返回处理结果。

Session 通过 mResponse 成员变量指向来自客户端的 IAccountManagerResponse 接口，当 Session 收到 AAS 的返回结果后，又通过

IAccountManagerResponse的Bp端对象向客户端返回处理结果。

Session mAuthenticator 变量的类型是 IAccountAuthenticator，它用于和远端的AAS通信。客户端发起的请求将通过 Session 经由 mAuthenticator调用对应AAS中的函数。

由图8-7和图8-8可知，AccountManagerService 在addAccount流程中起桥梁的作用，具体如下：

客户 端 将 请 求 发 送 给 AccountManagerService ，然 后 AccountManagerService再转发给对应的AAS。

AAS 处 理 完 的 结 果 先 返 回 给 AccountManagerService ，再 由 AccountManagerService返回给客户端。

由于图8-7和图8-8中定义的类名较相似，因此读者阅读时应仔细一些。

下面来看Session匿名类的处理。

(2) Session匿名类处理分析

首先调用Session的构造函数，代码为：

[-->AccountManagerService.java : Session]

```
public Session ( IAccountManagerResponse response, String
accountType,
    boolean           expectActivityLaunch,           boolean
stripAuthTokenFromResult) {
    super () ;
    .....
/*
注意其中的参数，expectActivityLaunch由客户端传来，如果用户调用
AccountManager addAccount时传入了activity参数，则该值为true，
stripAuthTokenFromResult的默认值为true
*/
mStripAuthTokenFromResult=stripAuthTokenFromResult ;
mResponse=response ;
mAccountType=accountType ;
mExpectActivityLaunch=expectActivityLaunch ;
mCreationTime=SystemClock.elapsedRealtime () ;
synchronized (mSessions) {
//将这个匿名类对象保存到AccountManagerService中的mSessions成员中
mSessions.put (toString () , this) ;
}
try{//监听客户端死亡消息
response.asBinder () .linkToDeath (this, 0) ;
}.....
}
```

获得匿名类对象后，addAccount将调用其bind函数，该函数由Session实现，代码如下：

[-->AccountManagerService. java : Session : bind]

```
void bind () {
```

```
// 绑定到 mAccountType 指定的 AAS。在本例中，AAS 的类型  
是“com.android.email”  
    if (!bindToAuthenticator (mAccountType) ) {  
        onError ( AccountManager.ERROR_CODE_REMOTE_EXCEPTION , "bind  
failure") ;  
    }  
}
```

bindToAuthenticator的代码为：

[-->AccountManagerService.java : Session :
bindToAuthenticator]

```
private      boolean      bindToAuthenticator      (      String  
authenticatorType) {  
    //从mAuthenticatorCache中查询满足指定类型的服务信息  
    AccountAuthenticatorCache.ServiceInfo<  
    AuthenticatorDescription>  
    authenticatorInfo=  
    mAuthenticatorCache.getServiceInfo (   
    AuthenticatorDescription.newKey (authenticatorType) ) ;  
    .....  
    Intent intent=new Intent () ;  
    intent.setAction  
(AccountManager.ACTION_AUTHENTICATOR_INTENT) ;  
    //设置目标服务的componentName  
    intent.setComponent (authenticatorInfo.componentName) ;  
    //通过bindService启动指定的服务，成功与否将通过第二个参数传递的  
    //ServiceConnection接口返回  
    if      (      !      mContext.bindService      (      intent,      this,  
    Context.BIND_AUTO_CREATE) ) {  
    .....  
    }  
    return true ;
```

```
}
```

由以上代码可知，Session的bind函数将启动指定类型的Service，这是通过bindService函数完成的。如果服务启动成功，Session的onServiceConnected函数将被调用，这部分代码如下：

[-->AccountManagerService.java : Session :
onServiceConnected]

```
public void onServiceConnected (ComponentName name, IBinder  
service) {  
    //得到远端AAS返回的IAccountAuthenticator接口，这个接口用于  
    //AccountManagerService和该远端AAS交互  
    mAuthenticator=IAccountAuthenticator.Stub.asInterface  
(service) ;  
    try{  
        run () ;//调用匿名类实现的run函数  
    }.....  
}
```

匿名类实现的run函数非常简单，代码如下：

[-->AccountManagerService.java :
addAccount返回的匿名类]

```
new Session ( response, accountType, expectActivityLaunch,  
true) {  
    public void run () throws RemoteException{  
        //调用远端AAS实现的addAccount函数
```

```
mAuthenticator.addAccount (this, mAccountType,  
authTokenType, requiredFeatures, options) ;  
}
```

由以上代码可知，AccountManagerService在addAccount最终将调用AAS实现的add-Account函数。

下面来看本例中满足“com.android.email”类型的服务是如何处理addAccount的请求的。该服务就是Email应用中的EasAuthenticatorService，下面来分析它。

3.EasAuthenticatorService处理请求

EasAuthenticatorService 的 创建 是 AccountManagerService调用了bindService完成的，该函数会触发EasAuthenticatorService的onBind函数的调用，这部分代码如下：

[-->EasAuthenticatorService.java : onBind]

```
public IBinder onBind (Intent intent) {  
if (AccountManager.ACTION_AUTHENTICATOR_INTENT.equals (  
intent.getAction () )) {  
//创建一个EasAuthenticator类型的对象，并调用其getIBinder函数  
return new EasAuthenticator (this) .getIBinder () ;  
}else return null;  
}
```

下面来分析EasAuthenticator。

(1) EasAuthenticator介绍

EasAuthenticator 是 EasAuthenticatorService 定义的内部类，其家族关系如图8-9所示。

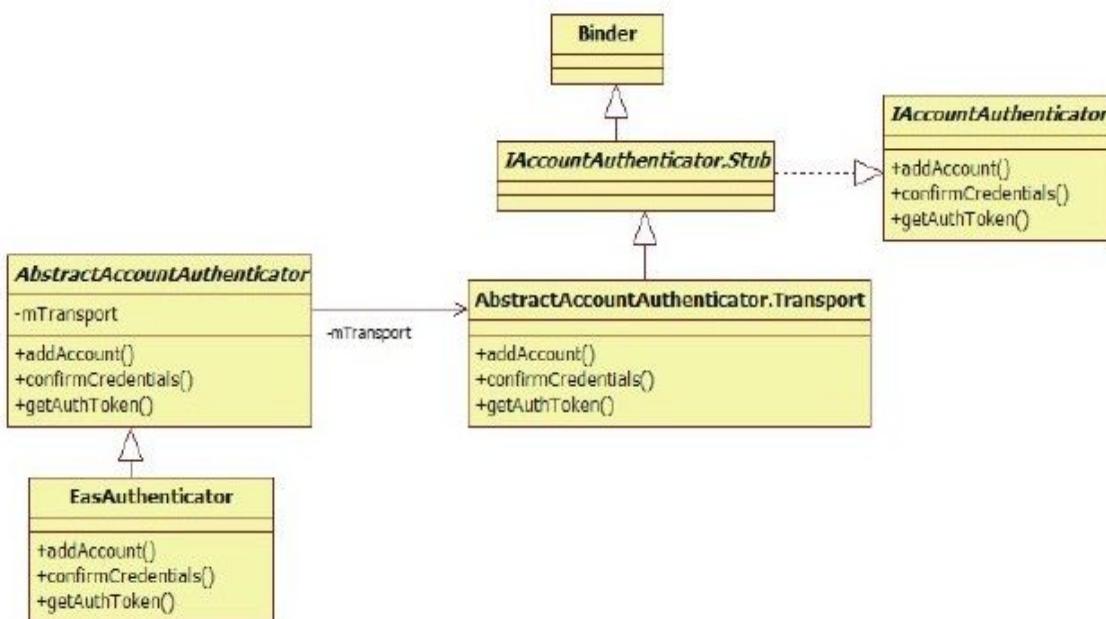


图 8-9 EasAuthenticator家族类图

由图8-9可知：

EasAuthenticator
AbstractAccountAuthenticator 从类派生。
AbstractAccountAuthenticator 内部有一个
mTransport的成员变量，其类型是AbstractAccountAuthenticator的内部类Transport。在前面的onBind

函数中，EasAuthenticator的getIBinder函数返回的就是这个变量。

Transport类继承自Binder，故它将参与Binder通信，并且是IAccountAuthenticator的Bn端。Session匿名类通过onServiceConnected函数将得到一个IAccountAuthenticator的Bp端对象。

当由AccoutManagerService的addAccount创建的那个Session匿名类调用IAccountAuthenticator Bp端对象的addAccount时，将触发位于Emai进程中的IAccountAuthenticator Bn端的addAccount。下面来分析Bn端的addAccount函数。

(2) EasAuthenticator的addAccount函数分析

根据上文的描述可知，Emai进程中首先被触发的是IAccountAuthenticator Bn端的addAccount函数，其代码如下：

[-->AbstractAccountAuthenticator.java :
Transport : addAccount]

```
private class Transport extends IAccountAuthenticator.Stub{
    public void addAccount ( IAccountAuthenticatorResponse response,
        String accountType, String authTokenType,
        String[]features, Bundle options)
        throws RemoteException{
```

```
//检查权限
checkBinderPermission () ;
try{
//调用AbstractAccountAuthenticator子类实现的addAccount函数
final Bundle result=
AbstractAccountAuthenticator.this.addAccount (
new AccountAuthenticatorResponse (response) ,
accountType, authTokenType, features, options) ;
//如果返回的result不为空，则调用response的onResult返回结果。
// 这个 response 是 IAccountAuthenticatorResponse 类型， 它 的
onResult
//将触发位于Session匿名类中mResponse变量的onResult函数被调用
if (result !=null)
response.onResult (result) ;
}.....
}
```

本例中AbstractAccountAuthenticator子类（即EasAuthenticator）实现的addAccount函数，代码如下：

[-->EasAuthenticatorService.java :
EasAuthenticator.addAccount]

```
public  Bundle  addAccount  ( AccountAuthenticatorResponse
response,
String accountType, String authTokenType,
String[]requiredFeatures, Bundle options)
throws NetworkErrorException{
//EasAuthenticator addAccount的处理逻辑和Email应用有关，读者只做
简单了解即可。
//如果用户传递的账户信息保护了密码和用户名，则走if分支。注意，其中有一
些参数名是
```

```
//通用的，例如OPTIONS_PASSWORD, OPTIONS_USERNAME等
if (options !=null & &options.containsKey (OPTIONS_PASSWORD)
&&options.containsKey (OPTIONS_USERNAME) ) {
//创建一个Account对象，该对象仅包括两个成员，一个是name，用于表示账户
名；
//另一个是type，用于表示账户类型
final Account account=new
Account (options.getString (OPTIONS_USERNAME) ,
AccountManagerTypes.TYPE_EXCHANGE) ;
//调用AccountManager 的 addAccountExplicitly 将 account 对象和
password传递
//给AccountManagerService处理。读者可自行研究这个函数，在其内部将这
些信息写入
//accounts.db的account表中
AccountManager.get (EasAuthenticatorService.this) .
addAccountExplicitly (account,
options.getString (OPTIONS_PASSWORD) , null) ;
//根据Email应用的规则，下面将判断和该账户相关的数据是否需要设置自动数
据同步
//首先判断是否需要处理联系人自动数据同步
boolean syncContacts=false ;
if (options.containsKey (OPTIONS_CONTACTS_SYNC_ENABLED) & &
options.getBoolean (OPTIONS_CONTACTS_SYNC_ENABLED) )
syncContacts=true ;
ContentResolver.setIsSyncable (account,
ContactsContract.AUTHORITY, 1) ;
ContentResolver.setSyncAutomatically (account,
ContactsContract.AUTHORITY, syncContacts) ;
boolean syncCalendar=false ;
.....//判断是否需要设置Calendar自动数据同步
boolean syncEmail=false ;
//如果选项中包含Email同步相关的功能，则需要设置Email数据同步的相关参
数
if (options.containsKey (OPTIONS_EMAIL_SYNC_ENABLED) & &
options.getBoolean (OPTIONS_EMAIL_SYNC_ENABLED) )
syncEmail=true ;
```

```
/*
下面这两个函数将和ContentService中的SyncManager交互。注意这两个函数：第一个函数setIsSyncable将设置Email对应的同步服务功能标志，第二个函数setSyncAutomatically将设置是否自动同步Email。
数据同步的内容留待8.4节再详细分析
*/
ContentResolver.setIsSyncable ( account,
EmailContent.AUTHORITY, 1 ) ;
ContentResolver.setSyncAutomatically ( account,
EmailContent.AUTHORITY,
syncEmail ) ;
//构造返回结果，注意，下面这些参数名是Android统一定义的，如
KEY_ACCOUNT_NAME等
Bundle b=new Bundle () ;
b.putString (AccountManager.KEY_ACCOUNT_NAME,
options.getString (OPTIONS_USERNAME) ) ;
b.putString (AccountManager.KEY_ACCOUNT_TYPE,
AccountManagerTypes.TYPE_EXCHANGE) ;
return b ;
}else{
//如果没有传递password，则需要启动一个Activity，该Activity对应的
Intent
//由actionSetupExchangeIntent返回。注意下面几个通用的参数，如
//KEY_ACCOUNT_AUTHENTICATOR_RESPONSE和KEY_INTENT
Bundle b=new Bundle () ;
Intent intent=AccountSetupBasics.actionSetupExchangeIntent (
EasAuthenticatorService.this) ;
intent.putExtra (
AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
response) ;
b.putParcelable (AccountManager.KEY_INTENT, intent) ;
return b ;
}
}
```

不同的AAS有自己特定的处理逻辑，以上代码向读者展示了EasAuthenticatorService的工作流程。虽然每个AAS的处理方式各有不同，但Android还是定义了一些通用的参数，例如，OPTIONS_USERNAME 用于表示用户名，OPTIONS_PASSWORD 用于表示密码等。关于这方面的内容，读者可查阅SDK中AccountManager的文档说明。

4. 返回值的处理流程

在EasAuthenticator 的 addAccount 返回处理结果后，AbstractAuthenticator 将通过 IAccountAuthenticatorResponse 的 onResult 将其返回给由 AccountManagerService 创建的 Session 匿名类对象。来看Session的onResult函数，其代码如下：

[-->AccountManagerService. java : Session :
onResult]

```
public void onResult (Bundle result) {  
    mNumResults++ ;  
    //从返回的result中取出相关信息，关于下面这个if分支处理和状态栏中相关的  
    //逻辑，  
    //读者可自行研究  
    if (result !=null&& !TextUtils.isEmpty (  
        result.getString (AccountManager.KEY_AUTHTOKEN) ) ) {  
        String accountName=result.getString  
        (AccountManager.KEY_ACCOUNT_NAME) ;
```

```
String accountType=
result.getString (AccountManager.KEY_ACCOUNT_TYPE) ;
if (!TextUtils.isEmpty (accountName) & &
!TextUtils.isEmpty (accountType) ) {
Account account=new Account (accountName, accountType) ;
cancelNotification (
getSigninRequiredNotificationId (account) ) ;
}}
IAccountManagerResponse response ;
//如果客户端传递了activity参数，则mExpectActivityLaunch为true。如
果
//AAS返回的结果中包含KEY_INTENT，则表明需要弹出Activity以输入账户和
密码
if (mExpectActivityLaunch & & result !=null
&& result.containsKey (AccountManager.KEY_INTENT) ) {
response=mResponse ;
}else{
/*
getResponseAndClose 返回的也是 mResponse ， 不过它会调用
unBindService
断开和AAS服务的连接。就整个流程而言，此时addAccount已经完成AAS和
AccountManagerService的工作，故无须再保留和AAS服务的连接。而由于上
面的if
分支还需要输入用户密码，因此以AAS和AccountManagerService之间的工作
还没有真正完成
*/
response=getResponseAndClose () ;
}
if (response !=null) {
try{
.....
if (mStripAuthTokenFromResult)
result.remove (AccountManager.KEY_AUTHTOKEN) ;
//调用位于客户端的IAccountManagerResponse的onResult函数
response.onResult (result) ;
}.....
```

```
    }  
}
```

客户端的 IAccountManagerResponse 接口由 AmsTask 内部类 Response 实现，其代码为：[--> AccountManager.java : AmsTask : Response.onResult]

```
public void onResult (Bundle bundle) {  
    Intent intent=bundle.getParcelable (KEY_INTENT) ;  
    //如果需要弹出Activity，则要利用客户端传入的Activity去启动AAS指定的  
    //Activity。这个Activity由AAS返回的Intent来表示  
    if (intent !=null&&mActivity !=null) {  
        mActivity.startActivity (intent) ;  
    }else if (bundle.getBoolean ("retry") ) {  
        //如果需要重试，则再次调用doWork  
        try{  
            doWork () ;  
        }.....  
    }else{  
        //将返回结果保存起来，当客户端调用getResult时，就会得到相关结果  
        set (bundle) ;  
    }  
}
```

5.AccountManager的addAccount分析总结

AccountManager 的 addAccount 流程分析起来会给人一种行云流水般的感觉。该流程涉及3个模块，分别是客户端、AccountManagerService 和 AAS。整体难度虽不算大，但架构却比较巧妙。

总结起来addAccount相关流程如图8-10所示。

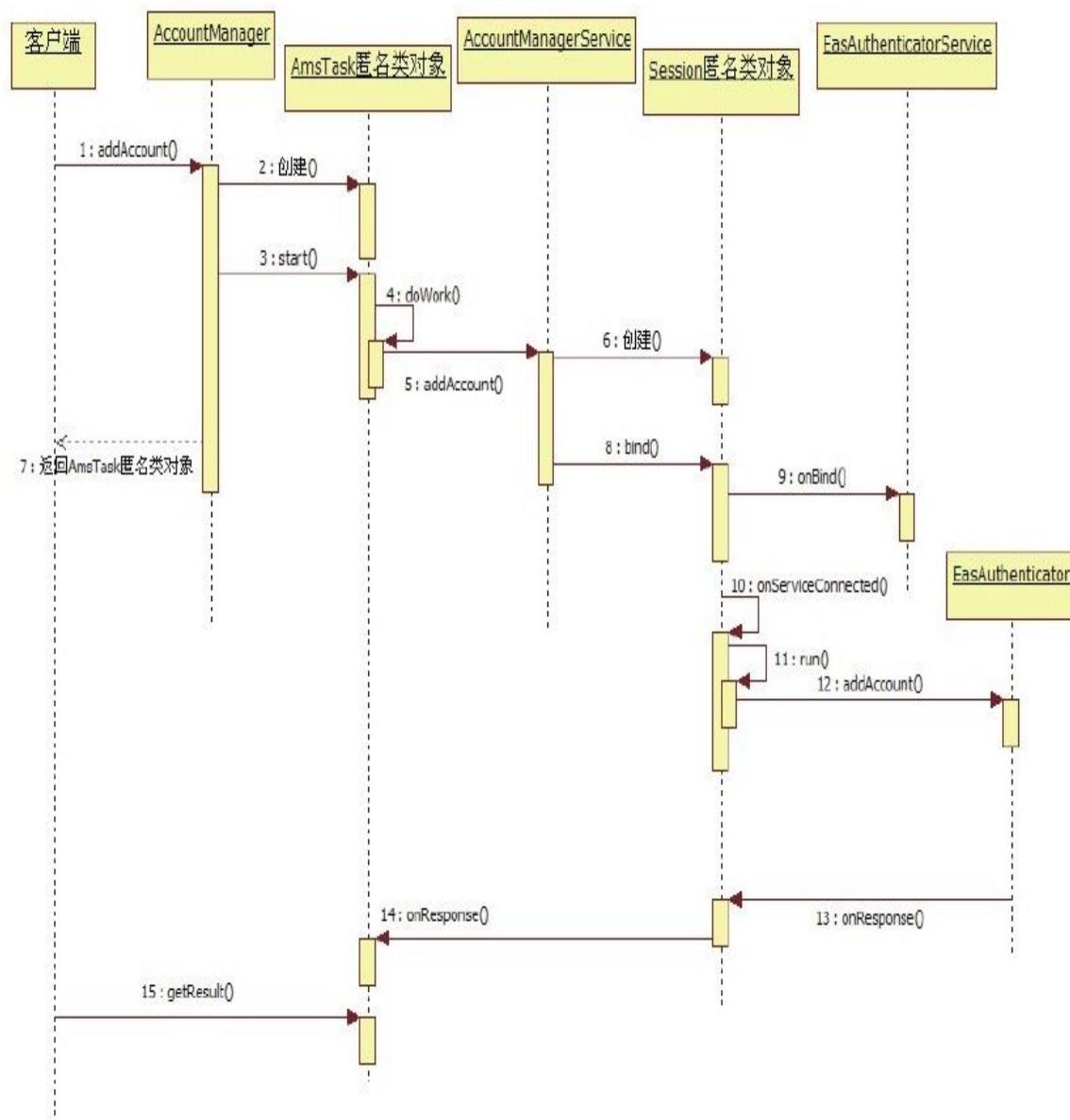


图 8-10 AccountManager的addAccount处理流程

为了让读者看得更清楚，图8-10中略去了一些细枝末节的内容。另外，图8-10中第10步的onServiceConnected函数应由位于SystemServer中

的ActivityThread对象调用，为方便阅读，这里没有画出ActivityThread对象。

[1]从设计模式角度来说，这属于Active Object模式。详细内容可阅读《Pattern.Oriented.Software.Architecture.Volume.2》的第2章“Concurrency Patterns”。

8.3.3 AccountManagerService的分析总结

本节对AccountManagerService进行分析，从技术上说，本节涉及和Java concurrent类相关的知识。另外，对并发编程来说，架构设计是最重要的，因此读者务必阅读脚注中提到的参考书籍《Pattern.Oriented.Software.Architecture.Volume.2》。

就整体而言，AccountManagerService及相关类的设计非常巧妙，读者不妨重温一下并认真体会一下RegisteredServicesCache的结构及addAccount的处理流程。

8.4 数据同步管理SyncManager分析

本节将分析ContentService中负责数据同步管理的SyncManager。SyncManager和AccountManagerService之间的关系比较紧密。同时，由于数据同步涉及手机中重要数据（例如联系人信息、Email、日历等）的传输，因此它的控制逻辑非常严谨，知识点也比较多，难度相对比AccountManagerService大。

先来认识数据同步管理的核心类SyncManager。

8.4.1 初识SyncManager

SyncManager的构造函数的代码较长，可分段来看。下面先来介绍第一段的代码。

1.SyncManager介绍

这部分的代码如下：

[-->SyncManager.java : SyncManager]

```
public SyncManager (Context context, boolean factoryTest) {
```

```
mContext=context ;
//SyncManager中的几位重要成员登场。见下文的解释
SyncStorageEngine.init (context) ;
mSyncStorageEngine=SyncStorageEngine.getSingleton () ;
mSyncAdapters=new SyncAdaptersCache (mContext) ;
mSyncQueue=new SyncQueue (mSyncStorageEngine,
mSyncAdapters) ;
HandlerThread syncThread=new HandlerThread
("SyncHandlerThread",
Process.THREAD_PRIORITY_BACKGROUND) ;
syncThread.start () ;
mSyncHandler=new SyncHandler (syncThread.getLooper ()) ;
mMainHandler=new Handler (mContext.getMainLooper ()) ;
/*
mSyncAdapters 类似于 AccountManagerService 中的
AccountAuthenticatorCache,
```

它用于管理系统中和 SyncService 相关的服务信息。下边的函数为 mSyncAdapters 增加一个

监听对象，一旦系统中的 SyncService 发生变化（例如安装了一个提供同步服务的 APK 包），则

SyncManager 需要针对该服务发起一次同步请求。同步请求由 scheduleSync 函数发送，

后文再分析此函数

*/

```
mSyncAdapters.setListener (new
RegisteredServicesCacheListener<SyncAdapterType> () {
public void onServiceChanged (SyncAdapterType type,
boolean removed) {
if (!removed) {
scheduleSync (null, type.authority, null, 0, false) ;
}
}
}, mSyncHandler) ;
```

在以上代码中，首先见到的是SyncManager的几位重要成员，它们之间的关系如图8-11所示。

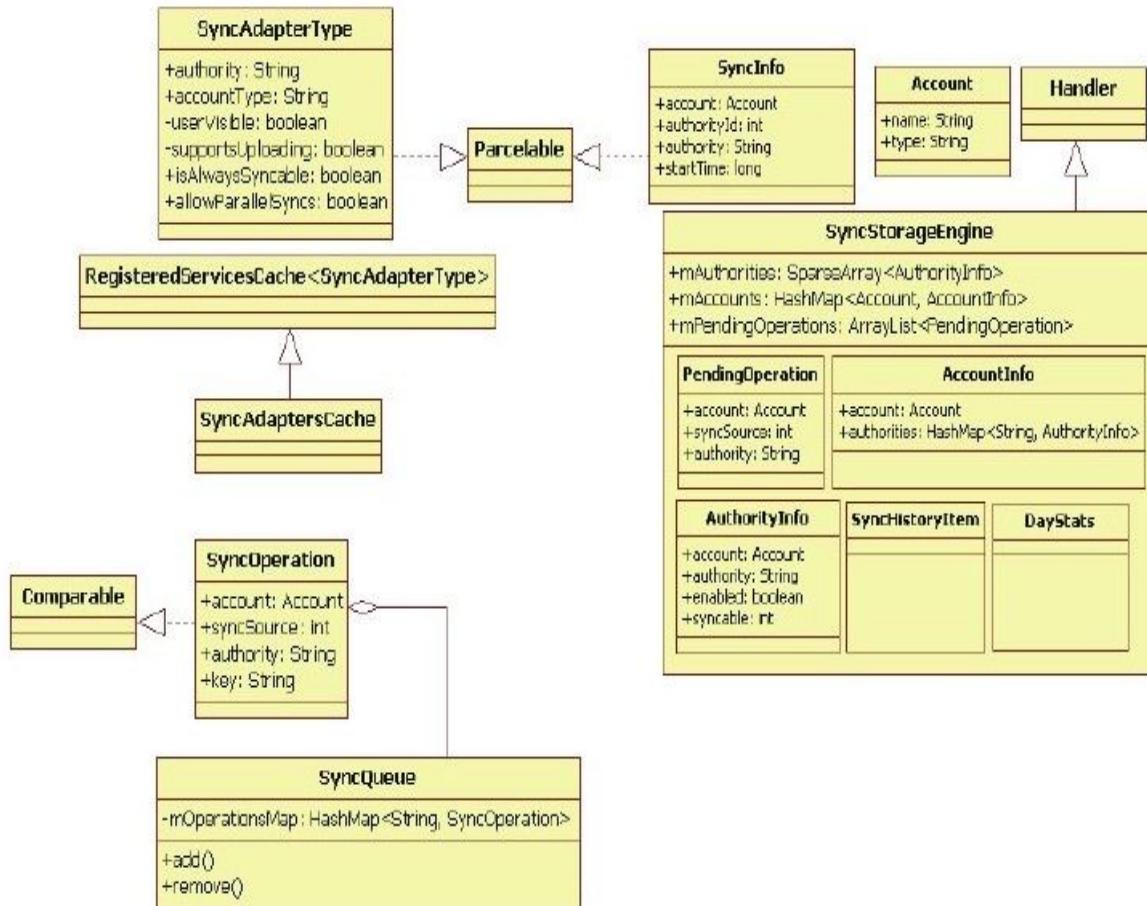


图 8-11 SyncManager成员类图

由图8-11可知，SyncManager的这几位成员的功能大体可分为3部分：

左上角是SyncAdaptersCache类，从功能和派生关系上看，它和AccountManager-Service中的AccountAuthenticatorCaches类相似。SyncAdaptersCache用于管理系统中SyncService服

务的信息。在SyncManager中，SyncService的信息用SyncAdapter-Type类来表示。

左下角是SyncQueue和SyncOperation类，SyncOperation代表一次正在执行或等待执行的同步操作，而SyncQueue通过mOperationsMap保存系统中存在的Sync-Operation。

右半部分是SyncStorageEngine，由于同步操作涉及重要数据的传输，加之可能耗时较长，所以SyncStorageEngine提供了一些内部类来保存同步操作中的一些信息。例如PendingOperation代表保存在本地文件中的还没有执行完的同步操作的信息。另外，SyncStorageEngine还需要对同步操作进行一些统计，例如耗电量统计等。SyncStorageEngine包含的内容较多，读者学习完本章后可自行研究相关内容。

SyncManager家族成员的责任分工比较细，后续分析时再具体讨论它们的作用。下面接着看SyncManager的构造函数：

[-->SyncManager.java : SyncManager]

```
....  
//创建一个用于广播发送的PendingIntent，该PendingIntent用于和  
//AlarmManagerService交互  
mSyncAlarmIntent=PendingIntent.getBroadcast(
```

```
mContext, 0, new Intent (ACTION_SYNC_ALARM) , 0) ;  
//注册CONNECTIVITY_ACTION广播监听对象，用于同步操作需要使用网络来传  
输数据，所以  
//此处需要监听和网络相关的广播  
IntentFilter intentFilter=  
    new IntentFilter (ConnectivityManager.CONNECTIVITY_ACTION) ;  
context.registerReceiver ( mConnectivityIntentReceiver,  
intentFilter) ;  
if (!factoryTest) {  
    //监听BOOT_COMPLETED广播  
    intentFilter=new IntentFilter  
(Intent.ACTION_BOOT_COMPLETED) ;  
    context.registerReceiver ( mBootCompletedReceiver,  
intentFilter) ;  
}  
//监听BACKGROUND_DATA_SETTING_CHANGED广播。该广播与是否允许后台传  
输数据有关，  
//用户可在Settings应用程序中设置对应选项  
intentFilter=  
    new IntentFilter (ConnectivityManager.  
ACTION_BACKGROUND_DATA_SETTING_CHANGED) ;  
context.registerReceiver ( mBackgroundDataSettingChanged,  
intentFilter) ;  
//监视设备存储空间状态广播。由于SyncStorageEngine会保存同步时的一些  
信息到存储  
//设备中，所以此处需要监视存储设备的状态  
intentFilter=new IntentFilter  
(Intent.ACTION_DEVICE_STORAGE_LOW) ;  
    intentFilter.addAction (Intent.ACTION_DEVICE_STORAGE_OK) ;  
    context.registerReceiver ( mStorageIntentReceiver,  
intentFilter) ;  
//监听SHUTDOWN广播。此处设置优先级为100，即优先接收此广播  
intentFilter=new IntentFilter (Intent.ACTION_SHUTDOWN) ;  
    intentFilter.setPriority (100) ;  
    context.registerReceiver ( mShutdownIntentReceiver,  
intentFilter) ;
```

```
if (!factoryTest) { //和通知服务交互，用于在状态栏上提示用户
    mNotificationMgr= (NotificationManager)
    context.getSystemService (Context.NOTIFICATION_SERVICE) ;
    //注意，以下函数注册的广播将针对前面创建的mSyncAlarmIntent
    context.registerReceiver (new SyncAlarmIntentReceiver () ,
    new IntentFilter (ACTION_SYNC_ALARM) ) ;
}

mPowerManager= (PowerManager)
context.getSystemService (Context.POWER_SERVICE) ;
//创建wakeLock，防止同步过程中掉电
mHandleAlarmWakeLock=
    mPowerManager.newWakeLock (PowerManager.PARTIAL_WAKE_LOCK,
    HANDLE_SYNC_ALARM_WAKE_LOCK) ;
mHandleAlarmWakeLock.setReferenceCounted (false) ;
mSyncManagerWakeLock=
    mPowerManager.newWakeLock (PowerManager.PARTIAL_WAKE_LOCK,
    SYNC_LOOP_WAKE_LOCK) ;
mSyncManagerWakeLock.setReferenceCounted (false) ;
//知识点一：监听SyncStorageEngine的状态变化，具体见下文解释
mSyncStorageEngine.addStatusChangeListener (
ContentResolver.SYNC_OBSERVER_TYPE_SETTINGS,
new ISyncStatusObserver.Stub () {
    public void onStatusChanged (int which) {
        sendCheckAlarmsMessage () ;
    }
}) ;
//知识点二：监视账户的变化。如果用户添加或删除了某个账户，则需要做相应
处理。
//具体见下文解释
if (!factoryTest) {
    AccountManager.get (mContext) .addOnAccountsUpdatedListener (
        SyncManager.this,
        mSyncHandler, false) ;
    onAccountsUpdated (AccountManager.get (mContext) .getAccounts
() ) ;
}
```

}

在以上代码中，有两个重要知识点。

第一，SyncManager为SyncStorageEngine设置了一个状态监听对象。根据前文的描述，在SyncManager家族中，SyncStorageEngine专门负责管理和保存同步服务中绝大部分的信息，所以当外界修改了这些信息时，SyncStorageEngine需要通知状态监听对象。我们可以通过一个例子了解其工作流程。下面的setSyncAutomatically函数的作用是设置是否自动同步某个账户的某项数据，代码如下：

[-->ContentService.java :
setSyncAutomatically]

```
public void setSyncAutomatically ( Account account, String providerName,  
boolean sync) {  
.....//检查WRITE_SYNC_SETTINGS权限  
long identityToken=clearCallingIdentity () ;  
try{  
SyncManager syncManager=getSyncManager () ;  
if (syncManager !=null) {  
/*  
通过SyncManager找到SyncStorageEngine对象，并调用它的  
setSyncAutomatically函数。在其内部会修改对应账户的同步服务信息，然后  
通知  
监听者，而这个监听者就是SyncManager设置的那个状态监听对象  
*/
```

```
        syncManager.getSyncStorageEngine () .setSyncAutomatically ( account, providerName, sync) ;  
    }  
    }finally{  
        restoreCallingIdentity (identityToken) ;  
    }  
}
```

在以上代码中，最终调用的是 SyncStorageEngine 的函数，但 SyncManager 也会因状态监听对象被触发而做出相应动作。实际上，ContentService 中大部分设置同步服务参数的 API，其内部实现就是先直接调用 SyncStorageEngine 的函数，然后再由 SyncStorageEngine 通知监听对象。读者在阅读代码时，仔细一些就可明白这一关系。

第二，SyncManager 将为 AccountManager 设置一个账户更新监听对象（注意，此处是 AccountManager ，而不是 AccountManagerService 。 AccountManager 这部分功能的代码不是很简单，读者有必要反复研究）。在 Android 平台上，数据同步和账户的关系非常紧密，并且同一个账户可以对应不同的数据项。例如，在 EasAuthenticator 的 addAccount 实现中，读者会发现一个 Exchange 账户可以对应 Contacts 、 Calendar 和 Email 三种不同的数据项。在添加 Exchange 账户时，还可以选择是否同步其中

的某项数据（通过判断实现addAccount时传递的options是否含有对应的同步选项，例如同步邮件数 据 时 需 要 设 置 的 OPTIONS_EMAIL_SYNC_ENABLED选项）。由于SyncManager和账户之间的这种紧密关系的存在，SyncManager就必须监听手机中账户的变化情况。

提示 上述两个知识点涉及的内容都是一些非常细节的问题，本章将它们作为小任务，读者可自行进行研究。

下面来认识一下SyncManager家族中的几位主要成员，首先是SyncStorageEngine。

2.SyncStorageEngine介绍

SyncStorageEngine负责整个同步系统中信息管理方面的工作。先看其init函数代码：

[-->SyncStorageEngine.java : init]

```
public static void init (Context context) {  
    if (sSyncStorageEngine !=null) return ;  
    /*
```

得到系统中加密文件系统的路径，如果手机中没有加密的文件系统（根据系统属性“persist.security.efs.enabled”的值来判断），则返回的路径为/data，目前的Android手机大部分都没有加密文件系统，故dataDir为/data
*/

```
File dataDir=Environment.getExternalStorageDirectory () ;
```

```
//创建SyncStorageEngine对象
sSyncStorageEngine=new SyncStorageEngine ( context,
dataDir) ;
}
```

SyncStorageEngine的构造函数代码为：

[-->SyncStorageEngine.java :
SyncStorageEngine]

```
private SyncStorageEngine (Context context, File dataDir) {
    mContext=context;
    sSyncStorageEngine=this;
    mCal=Calendar.getInstance ( TimeZone.getTimeZone
("GMT+0") );
    File systemDir=new File (dataDir, "system");
    File syncDir=new File (systemDir, "sync");
    syncDir.mkdirs ();
    //mAccountInfoFile指向/data/system-sync/accounts.xml
    mAccountInfoFile=new AtomicFile ( new File
(syncDir, "accounts.xml") );
    //mStatusFile指向/data/system-sync/status.bin, 该文件记录
    //一些和同步服务相关的一些状态信息
    mStatusFile=new AtomicFile ( ( new File
(syncDir, "status.bin") );
    //mStatusFile指向/data/system-sync/pending.bin, 该文件记录了当
前处于pending
    //状态的同步请求
    mPendingFile=new AtomicFile ( ( new File
(syncDir, "pending.bin") );
    //mStatusFile指向/data/system-sync/stats.bin, 该文件记录同步服务
管理运行过程
    //中的一些统计信息
```

```
mStatisticsFile=new           AtomicFile      (       new           File  
(syncDir, "stats.bin") ) ;  
/*
```

解析上述4个文件，从下面的代码看，似乎是先读（readxxx）后写（writexxx），这是怎

么回事？答案在AtomicFile中，它内部实际包含两个文件，其中一个用于备份，防止数据丢失。

感兴趣的读者可以自行研究AtomicFile类，其实现非常简单

```
*/
```

```
readAccountInfoLocked () ;  
readStatusLocked () ;  
readPendingOperationsLocked () ;  
readStatisticsLocked () ;  
readAndDeleteLegacyAccountInfoLocked () ;  
writeAccountInfoLocked () ;  
writeStatusLocked () ;  
writePendingOperationsLocked () ; writeStatisticsLocked () ;  
}
```

上述init和SyncStorageEngine的构造函数都比较简单，故不再详述。下面将分析accounts.xml。以下是笔者Kindle Fire（CM9的ROM）机器中的accounts.xml文件，如图8-12所示。

图8-12中两个黑框中内容的作用如下：

第一个框中的listen-for-tickles，该标签和Android平台中的Master Sync有关。Master Sync用于控制手机中是否所有账户对应的所有数据项都自动同步。用户可通过ContentResolver setMasterSyncAutomatically进行设置。

第二个框代表一个 AuthorityInfo。 AuthorityInfo记录了账户和SyncService相关的一些信息。此框中的 account 为笔者的邮箱， type 为“com.google”。另外，从图8-12中还可发现，一个账户（包含account和type两个属性）可以对应多种类型的数据项，例如此框中对应的数据项是“com.android.email.provider”，而此框前面一个 AuthorityInfo 对应的数据项是“com.google.android.apps.books”。 AuthorityInfo 中的 periodicSync 用于控制周期同步的时间，单位是秒，默认是 86400 秒，也就是 1 天。另外， AuthorityInfo 中还有一个重要属性 syncable，它的可选值为 true、 false 和 unknown （在代码中，这 3 个值分别对应整型值 1、 0 和 -1 ）。

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<accounts version="2" nextAuthorityId="18" listen-for-tickles="false">
<authority id="9" account="fanping.deng@gmail.com" type="com.google" authority="com.android.calendar" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="10" account="fanping.deng@gmail.com" type="com.google" authority="com.android.browser" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="11" account="fanping.deng@gmail.com" type="com.google" authority="com.google.android.music.MusicContent" enabled="false" syncable="false">
<periodicSync period="86400" />
</authority>
<authority id="12" account="fanping.deng@gmail.com" type="com.google" authority="subscribedfeeds" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="13" account="fanping.deng@gmail.com" type="com.google" authority="gmail-ls" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="14" account="fanping.deng@gmail.com" type="com.google" authority="com.android.contacts" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="15" account="fanping.deng@gmail.com" type="com.google" authority="com.google.android.gallery3d.GooglePhotoProvider" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="16" account="fanping.deng@gmail.com" type="com.google" authority="com.google.android.apps.books" enabled="true" syncable="true">
<periodicSync period="86400" />
</authority>
<authority id="17" account="fanping.deng@gmail.com" type="com.google" authority="com.android.email.provider" enabled="true" syncable="unknown">
<periodicSync period="86400" />
</authority>
</accounts>
```

图 8-12 accounts.xml内容展示

syncable的unknown状态较难理解，它和参数SYNC_EXTRAS_INITIALIZE有关，官方的解释如下：

```
/**  
 * Set by the SyncManager to request that the SyncAdapter  
 * initialize itself for  
 * the given account/authority pair. One required initialization  
 * step is to  
 * ensure that setIsSyncable () has been called with a>=0  
 * value.  
 * When this flag is set the SyncAdapter does not need to do a  
 * full sync,  
 * though it is allowed to do so.  
 */  
public static final String  
SYNC_EXTRAS_INITIALIZE="initialize";
```

由以上解释可知，如果某个SyncService的状态为unknown，那么在启动它时必须传递一个SYNC_EXTRAS_INITIALIZE选项，SyncService解析该选项后即可知自己尚未被初始化。当它完成初始化后，需要调用setIsSyncable函数设置syncable属性值为1。另外，SyncService初始化完成后，是否可接着执行同步请求呢？目前的设计是，它们并不会立即执行同步，需要用户再次发起请求。读者在后续小节中会看到与此相关的处理。

此处先来看setIsSyncable的使用示例，前面分析的EasAuthenticator addAccount中有如下的函数调用：

```
//添加完账户后，将设置对应的同步服务状态为1  
ContentResolver.setIsSyncable ( account,  
EmailContent.AUTHORITY, 1 ) ;  
ContentResolver.setSyncAutomatically ( account,  
EmailContent.AUTHORITY,  
syncEmail ) ;
```

在EasAuthenticator中，一旦添加了账户，就会设置对应SyncService的syncable属性值为1。SyncManager将根据这个状态做一些处理，例如立即发起一次同步操作。

注意 是否设置syncable属性值和具体应用有关，图8-12第二个框中的gmail邮件同步服务就没有因为笔者添加了账户而设置syncable属性值为1。

3.SyncAdaptersCache介绍

再看SyncAdaptersCache，其构造函数代码如下：

```
[-->SyncAdaptersCache.           java      :  
SyncAdaptersCache]
```

```
SyncAdaptersCache (Context context) {  
/*  
调用基类RegisteredServicesCache的构造函数，其中SERVICE_INTERFACE  
和SERVICE_META_DATA的值为“android.content.SyncAdapter”，  
ATTRIBUTES_NAME为字符串"sync-adapter”  
*/  
super (context, SERVICE_INTERFACE, SERVICE_META_DATA,  
ATTRIBUTES_NAME, sSerializer) ;  
}  
}
```

SyncAdaptersCache 的 基 类 是 RegisteredServicesCache。8.3.1 节已经分析过 Registered-ServicesCache了，此处不再赘述。下面展示一个实际的例子，如图8-13所示。

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>  
<services>  
<service uid="10016" authority="com.android.contacts" accountType="com.android.exchange" />  
<service uid="10006" authority="com.google.android.apps.books" accountType="com.google" />  
<service uid="10047" authority="com.android.contacts" accountType="com.google" />  
<service uid="10016" authority="com.android.email.provider" accountType="com.android.exchange" />  
<service uid="10015" authority="com.android.email.provider" accountType="com.android.email" />  
<service uid="10019" authority="gmail-ls" accountType="com.google" />  
<service uid="10047" authority="subscribedfeeds" accountType="com.google" />  
<service uid="10026" authority="com.google.android.music.MusicContent" accountType="com.google" />  
<service uid="10016" authority="com.android.calendar" accountType="com.android.exchange" />  
<service uid="10047" authority="com.android.browser" accountType="com.google" />  
<service uid="10048" authority="com.google.android.gallery3d.GooglePhotoProvider" accountType="com.google" />  
<service uid="10032" authority="com.android.calendar" accountType="com.google" />  
</services>
```

图 8-13 android.content.SyncAdapter.xml

图8-13列出了笔者Kindle Fire上安装的同步服务，其中黑框列出的是“om.android.exchange”，该项服务和Android中Exchange应用有关。Exchange的AndroidManifest.xml文件的信息如图8-14所示。

图8-14列出了Exchange应用所支持的针对邮件提供的同步服务，即EmailSyncAdapterService。该服务会通过meta-data中的resource来描述自己，这部分内容如图8-15所示。

```
<service
    android:name="com.android.exchange.EmailSyncAdapterService"
    android:exported="true">
    <intent-filter>
        <action
            android:name="android.content.SyncAdapter" />
    </intent-filter>
    <meta-data android:name="android.content.SyncAdapter"
        android:resource="@xml/syncadapter_email" />
</service>
```

图 8-14 Exchange AndroidManifest.xml文件示意

```
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.android.email.provider"
    android:accountType="com.android.exchange"
    android:supportsUploading="false"
/>
```

图 8-15 syncadapter_email.xml内容展示

图8-15告诉我们，EmailSyncAdapterService对应的账户类型是“com.android.exchange”，需要同步的邮件数据地址由contentAuthority表示，即本例中的“com.android.email.provider”。注意，EmailSyncAdapterService只支持从网络服务端同步数据到本机，故supportsUploading为false。

再看SyncManager家族中最后一位成员SyncQueue。

4.SyncQueue介绍

SyncQueue 用于管理同步操作对象 SyncOperation。SyncQueue 的构造函数代码为：

[-->SyncQueue.java : SyncQueue]

```
public SyncQueue (SyncStorageEngine syncStorageEngine,
final SyncAdaptersCache syncAdapters) {
    mSyncStorageEngine=syncStorageEngine ;
    //从SyncStorageEngine中取出上次没有完成的同步操作信息，这类信息由
    //PendingOperations表示
    ArrayList<SyncStorageEngine.PendingOperation>ops
    =mSyncStorageEngine.getPendingOperations () ;
    final int N=ops.size () ;
    for (int i=0 ; i<N ; i++) {
        SyncStorageEngine.PendingOperation op=ops.get (i) ;
        //从SyncStorageEngine中取出该同步操作的backoff信息
        final Pair<Long, Long>backoff=
            syncStorageEngine.getBackoff (op.account, op.authority) ;
        //从SyncAdaptersCache中取出该同步操作对应的同步服务信息，如果同步服
       务已经不存在，
        //则无须执行后面的流程
        final RegisteredServicesCache.ServiceInfo<SyncAdapterType>
        syncAdapterInfo=syncAdapters.getServiceInfo (
            SyncAdapterType.newKey (op.authority,
            op.account.type) ) ;
        if (syncAdapterInfo==null) continue ;
        //构造一个SyncOperation对象
        SyncOperation syncOperation=new SyncOperation (
            op.account, op.syncSource, op.authority, op.extras, 0,
            backoff !=null?backoff.first : 0,
            syncStorageEngine.getDelayUntilTime (op.account,
            op.authority) ,
            syncAdapterInfo.type.allowParallelSyncs () ) ;
        syncOperation.expedited=op.expedited ;
```

```
syncOperation.pendingOperation=op ;  
//将SyncOperation对象保存到mOperationsMap变量中  
add (syncOperation, op) ;  
}  
}
```

SyncQueue比较简单，其中一个比较难理解的概念是backoff，后文再对此作解释。

至此，SyncManager及相关家族成员已介绍完毕。下面将通过实例分析同步服务的工作流程。在本例中，将同步Email数据，目标同步服务为EmailSyncAdapterService。

8.4.2 ContentResolver的requestSync分析

ContentResolver提供了一个requestSync函数，用于发起一次数据同步请求。在本例中，该函数的调用方法如下：

```
Account emailSyncAccount=new Account ("fanping.deng@gmail",
"com.google") ;
String emailAuthority="com.android.email.provider" ;
Bundle emailBundle=new Bundle () ;
.....//为emailBundle添加相关的参数。这些内容和具体的同步服务有关
//发起Email同步请求
ContentResolver.requestSync (emailSyncAccount,
emailAuthority, emailBundle) ;
```

1.客户端发起请求

ContentResolver的requestSync的代码如下：

[-->ContentResolver.java : requestSync]

```
public static void requestSync ( Account account, String
authority,
Bundle extras) {
//检查extras携带的参数的数据类型，目前只支持float、int和String等几
种类型
validateSyncExtrasBundle (extras) ;
try{
//调用ContentService的requestSync函数
```

```
    getContentService ( ) .requestSync ( account, authority,  
extras) ;  
}.....  
}
```

与添加账户（addAccount）相比，客户端发起一次同步请求所要做的工作就太简单了。

下面转战ContentService去看它的requestSync函数。

2.ContentService的requestSync函数分析

这部分的代码如下：

[-->ContentService.java : requestSync]

```
public void requestSync (Account account, String authority,  
Bundle extras) {  
    ContentResolver.validateSyncExtrasBundle (extras) ;  
    long identityToken=clearCallingIdentity () ;  
    try{  
        SyncManager syncManager=getSyncManager () ;  
        if (syncManager !=null) {  
            //调用syncManager的scheduleSync  
            syncManager.scheduleSync (account, authority, extras,  
            0, false) ;  
        }  
    }finally{  
        restoreCallingIdentity (identityToken) ;  
    }  
}
```

ContentService将工作转交给SyncManager来完成，其调用的函数是scheduleSync。

(1) SyncManager的scheduleSync函数分析

先行介绍的scheduleSync函数非常重要，其调用代码如下：

```
/*
scheduleSync一共5个参数，其作用分别如下。
requestedAccount表明要进行同步操作的账户。如果为空，SyncManager将同步所有账户。
requestedAuthority表明要同步的数据项。如果为空，SyncManager将同步所有数据项。
extras指定同步操作中的一些参数信息。这部分内容后续分析时再来介绍。
delay指定本次同步请求是否延迟执行。单位为毫秒。
onlyThoseWithUnkownSyncableState决定是否只同步那些处于unknown状态的同步服务。
该参数在代码中没有注释。结合前面对syncable为unknown的分析，如果该参数为true，则
本次同步请求的主要作用就是通知同步服务进行初始化操作
*/
public void scheduleSync ( Account requestedAccount, String
requestedAuthority,
Bundle extras, long delay, boolean
onlyThoseWithUnkownSyncableState)
```

关于scheduleSync的代码将分段分析，其相关代码如下：

[-->SyncManager.java : scheduleSync]

```
boopublic void scheduleSync (Account requestedAccount,
String requestedAuthority, Bundle extras,
long delay, boolean onlyThoseWithUnkownSyncableState)
//判断是否允许后台数据传输
final boolean backgroundDataUsageAllowed= ! mBootCompleted || 
getConnectivityManager () .getBackgroundDataSetting () ;
if (extras==null) extras=new Bundle () ;
//下面将解析同步服务中特有的一些参数信息，下面将逐条解释
//SYNC_EXTRAS_EXPEDITED参数表示是否立即执行。如果设置了该选项，则
delay参数不起作用
//delay参数用于设置延迟执行时间，单位为毫秒
Boolean expedited=extras.getBoolean (
ContentResolver.SYNC_EXTRAS_EXPEDITED, false) ;
if (expedited)
delay=-1 ;
Account[]accounts ;
if (requestedAccount !=null) {
accounts=new Account[]{requestedAccount} ;
}.....
//SYNC_EXTRAS_UPLOAD参数设置本次同步是否为上传。从本地同步到服务端为
Upload,
//反之为download
final boolean uploadOnly=extras.getBoolean (
ContentResolver.SYNC_EXTRAS_UPLOAD, false) ;
//SYNC_EXTRAS_MANUAL等同于SYNC_EXTRAS_IGNORE_BACKOFF加
//SYNC_EXTRAS_IGNORE_SETTINGS
final boolean manualSync=extras.getBoolean (
ContentResolver.SYNC_EXTRAS_MANUAL, false) ;
//如果是手动同步，则忽略backoff和settings参数的影响
if (manualSync) {
//知识点一：SYNC_EXTRAS_IGNORE_BACKOFF：该参数和backoff有关，见下
文的解释
extras.putBoolean (
ContentResolver.SYNC_EXTRAS_IGNORE_BACKOFF, true) ;
//SYNC_EXTRAS_IGNORE_SETTINGS：忽略设置
extras.putBoolean (
```

```
ContentResolver.SYNC_EXTRAS_IGNORE_SETTINGS, true) ;  
}  
final boolean ignoreSettings=extras.getBoolean (  
ContentResolver.SYNC_EXTRAS_IGNORE_SETTINGS, false) ;  
//定义本次同步操作的触发源，见下文解释  
int source ;  
if (uploadOnly) {  
source=SyncStorageEngine.SOURCE_LOCAL ;  
}else if (manualSync) {  
source=SyncStorageEngine.SOURCE_USER ;  
}else if (requestedAuthority==null) {  
source=SyncStorageEngine.SOURCE_POLL ;  
}else{  
source=SyncStorageEngine.SOURCE_SERVER ;  
}
```

在以上代码中，有两个知识点需要说明。

知识点一和backoff（这个词不太好翻译）有关。和其相关的应用场景是，如果本次同步操作执行失败，则尝试休息一会再执行，而backoff在这个场景中的作用就是控制休息时间。由以上代码可知，当用户设置了手动（Manual）参数后，就无须对这次同步操作使用backoff模式。

另外，在后续的代码中，我们会发现和backoff有关的数据被定义成一个Pair<Long, Long>，即backoff对应两个参数。这两个参数到底有什么用呢？笔者在SyncManager代码中找到了一个setBackoff函数，其参数的命名很容易理解。setBackoff函数的原型如下：

[--> SyncManager.java : setBackoff]

```
public void setBackoff ( Account account, String providerName,  
    long nextSyncTime, long nextDelay)
```

在调用这个函数时，Pair<Long, Long>中的两个参数分别对应nextSyncTime和nextDelay，所以，Pair中的第一个参数对应nextSyncTime，第二个参数对应nextDelay。backoff的计算中实际上存在着一种算法。读者不妨先研究setBackoff，然后再和我们一起分享这个算法的相关知识。

知识点二和SyncStorageEngine定义的触发源有关。说白了，触发源就是描述该次同步操作是因何而起的。SyncStorageEngine一共定义了4种类型的触发源，这里笔者直接展示其原文解释：

```
/*Enum value for a local-initiated sync.*/
public static final int SOURCE_LOCAL=1;
/**Enum value for a poll-based sync (e.g., upon connection  
to network) */
public static final int SOURCE_POLL=2;
/*Enum value for a user-initiated sync.*/
public static final int SOURCE_USER=3;
/*Enum value for a periodic sync.*/
public static final int SOURCE_PERIODIC=4;
```

触发源的作用主要是为了方便 SyncStorageEngine 开展对应的统计工作。本书不深究这部分内容，感兴趣的读者可在学习完本节后自行研究。

关于 scheduleSync 下一阶段的工作，代码如下：

[-->SyncManager.java : scheduleSync]

```
//从SyncAdaptersCache中取出所有SyncService信息
final HashSet<String> syncableAuthorities=new HashSet<
String> () ;
for (RegisteredServicesCache.ServiceInfo<SyncAdapterType>
syncAdapter : mSyncAdapters.getAllServices () ) {
syncableAuthorities.add (syncAdapter.type.authority) ;
}
//如果指定了本次同步的authority，则从上述同步服务信息中找到满足要求的
SyncService
if (requestedAuthority !=null) {
final boolean hasSyncAdapter=
syncableAuthorities.contains (requestedAuthority) ;
syncableAuthorities.clear () ;
if (hasSyncAdapter) syncableAuthorities.add
(requestedAuthority) ;
}
final boolean masterSyncAutomatically=
mSyncStorageEngine.getMasterSyncAutomatically () ;
for (String authority : syncableAuthorities) {
for (Account account : accounts) {
//取出AuthorityInfo中的syncable属性值，如果为1，则其状态为true，
//如果为-1，则其状态为unknown
int isSyncable=mSyncStorageEngine.getIsSyncable (
```

```
account, authority) ;
if (isSyncable==0) continue; //syncable为false，则不能进行同步操作
final RegisteredServicesCache.ServiceInfo<SyncAdapterType>
syncAdapterInfo=
mSyncAdapters.getServiceInfo (
SyncAdapterType.newKey (authority, account.type) ) ;
.....
//有些同步服务支持多路并发同步操作
final boolean allowParallelSyncs=
syncAdapterInfo.type.allowParallelSyncs () ;
final boolean isAlwaysSyncable=syncAdapterInfo.type.
isAlwaysSyncable () ;
//如果该同步服务此时的状态为unknown，且它又是永远可同步的(AlwaysSyncable)，
//那么通过setIsSyncable设置该服务的状态为1
if (isSyncable<0&&isAlwaysSyncable) {
mSyncStorageEngine.setIsSyncable (account, authority, 1) ;
isSyncable=1 ;
}
//如果只能操作unknow状态的同步服务，但该服务的状态不是unknown，则不允许后续操作
if (onlyThoseWithUnkownSyncableState && isSyncable>=0)
continue ;
//如果此同步服务不支持上传，但本次同步又需要上传，则不允许后续操作
if ( ! syncAdapterInfo.type.supportsUploading ( ) & &
uploadOnly)
continue ;
//判断是否允许执行本次同步操作。如果同步服务状态为unknown，则总是允许发起同步请求，
//因为这时的同步请求只是为了初始化SyncService
boolean syncAllowed= (isSyncable<0) ||ignoreSettings
|| (backgroundDataUsageAllowed &&masterSyncAutomatically
&&mSyncStorageEngine.getSyncAutomatically (
account, authority) ) ;
.....
```

```
//取出对应的backoff参数
Pair<Long, Long>backoff=mSyncStorageEngine.getBackoff (
account, authority) ;
//获取延迟执行时间
long delayUntil=mSyncStorageEngine.getDelayUntilTime (
account, authority) ;
final long backoffTime=backoff !=null?backoff.first:0;
if (isSyncable<0) {
Bundle newExtras=new Bundle () ;
//如果syncable状态为unknown，则需要设置一个特殊的参数，即
//SYNC_EXTRAS_INITIALIZE，它将通知SyncService进行初始化操作
newExtras.putBoolean
(ContentResolver.SYNC_EXTRAS_INITIALIZE, true) ;
scheduleSyncOperation (
new SyncOperation (account, source, authority, newExtras, 0,
backoffTime, delayUntil, allowParallelSyncs) ) ;
}
if (!onlyThoseWithUnkownSyncableState)
scheduleSyncOperation (
new SyncOperation ( account, source, authority, extras,
delay,
backoffTime, delayUntil, allowParallelSyncs) ) ;
}//for循环结束
}
```

scheduleSync函数较复杂，难点在于其策略控制。建议读者反复阅读这部分内容。

scheduleSync最后将构造一个SyncOperation对象，并调用scheduleSyncOperation处理它。scheduleSyncOperation内部会将这个SyncOperation对象保存到mSyncQueue中，然后发送

MESSAGE_CHECK_ALARMS 消息让
mSyncHandler处理。由于scheduleSyncOperation函
数比较简单，因此下面将直接去mSyncHandler的
handleMessage 函数中分析
MESSAGE_CHECK_ALARMS的处理过程。

(2) 处理MESSAGE_CHECK_ALARMS消息

SyncHandler的handleMessage代码如下：

[-->SyncManager.java : SyncHandler :
handleMessage]

```
public void handleMessage (Message msg) {  
    long earliestFuturePollTime=Long.MAX_VALUE ;  
    long nextPendingSyncTime=Long.MAX_VALUE ;  
    try{  
        waitUntilReadyToRun () ;  
        mDataConnectionIsConnected=readDataConnectionState () ;  
        //获得WakeLock，防止在同步过程中掉电  
        mSyncManagerWakeLock.acquire () ;  
        //处理周期同步的操作  
        earliestFuturePollTime=scheduleReadyPeriodicSyncs () ;  
        switch (msg.what) {  
            ....  
            case SyncHandler.MESSAGE_CHECK_ALARMS :  
                // 调用maybeStartNextSyncLocked函数，返回一个时间。见下文解释  
                nextPendingSyncTime=maybeStartNextSyncLocked () ;  
                break ;  
            ....  
        } //switch结束
```

```
    }finally{
        manageSyncNotificationLocked () ;
        /*
         将上边函数调用的返回值传递给manageSyncAlarmLocked，该函数内部与
         AlarmManagerService交互，其实就是定义一个定时提醒。在Alarm超时后，
         就会广播
         在 SyncManager 构 造 函 数 中 定 义 的 那 个 PendingIntent
         mSyncAlarmIntent，

         而SyncManager收到该广播后又会做对应处理。相关内容读者可自行阅读
         */
        manageSyncAlarmLocked (      earliestFuturePollTime,
        nextPendingSyncTime) ;
        mSyncTimeTracker.update () ;
        mSyncManagerWakeLock.release () ;
    }
}
```

如以上代码所述，MESSAGE_CHECK_ALARMS消息的处理就是调用maybeStartNext-SyncLocked函数。这个函数内容较烦琐，它主要做了以下几项工作。

检查SyncQueue中保存的同步操作对象SyncOperation，判断它们对应的同步服务的状态是否为false，如果为false，则不允许执行该同步操作。

查询ConnectivityManager以判断目标同步服务是否使用了网络。如果该服务当前没有使用网络，则不允许执行该同步操作。

判断同步操作对象的执行时间是否已到，如果未到，则不允许执行该操作。

将通过上述判断的同步操作对象SyncOperation与当前系统中正在执行的同步操作上下文对象进行比较。系统当前正在执行的同步操作上下文对象对应的数据类是ActiveSyncContext，它是在同步操作对象之上的一一个封装，包含了能和同步服务交互的接口。由于并非所有同步服务都支持多路并发同步操作，因此这里需做一些处理，以避免不必要的同步操作。另外，如果一个仅对应初始化同步服务的同步操作执行时间过长（由系统属性“sync.max_time_per_sync”控制，默认是5分钟），则系统需要做一些处理。

提示 maybeStartNextSyncLocked是笔者在本节留给读者自行分析的函数中最难的一个。读

者务必阅读完下面的分析后，尝试去研究此函数。

1 通过上述层层考验后，manageSyncAlarmLocked 最后将调用 dispatchSyncOperation真正去派发一个同步操作。下面来看dispatchSyncOperation的代码。

[-->SyncManager.java
dispatchSyncOperation]

```
private boolean dispatchSyncOperation (SyncOperation op) {  
    SyncAdapterType syncAdapterType=SyncAdapterType.  
    newKey (op. authority, op.account.type) ;  
    RegisteredServicesCache. ServiceInfo<SyncAdapterType>  
    syncAdapterInfo=  
    mSyncAdapters. getServiceInfo (syncAdapterType) ;  
    .....  
    //构造一个ActiveSyncContext对象，它就是前面提到的同步操作上下文对象  
    ActiveSyncContext activeSyncContext=  
    new ActiveSyncContext (op,  
    insertStartSyncEvent (op) , syncAdapterInfo.uid) ;  
    activeSyncContext.mSyncInfo=  
    mSyncStorageEngine.addActiveSync (activeSyncContext) ;  
    //mActiveSyncContexts保存了当前系统中所有的ActiveSyncContext对象  
    mActiveSyncContexts.add (activeSyncContext) ;  
    //为该对象绑定到具体的同步服务上  
    if ( ! activeSyncContext.bindToSyncAdapter  
(syncAdapterInfo) ) {  
        closeActiveSyncContext (activeSyncContext) ;  
        return false ;  
    }  
    return true ;  
}
```

ActiveSyncContext是SyncManager和同步服务交互的关键类，其家族图谱如图8-16所示。

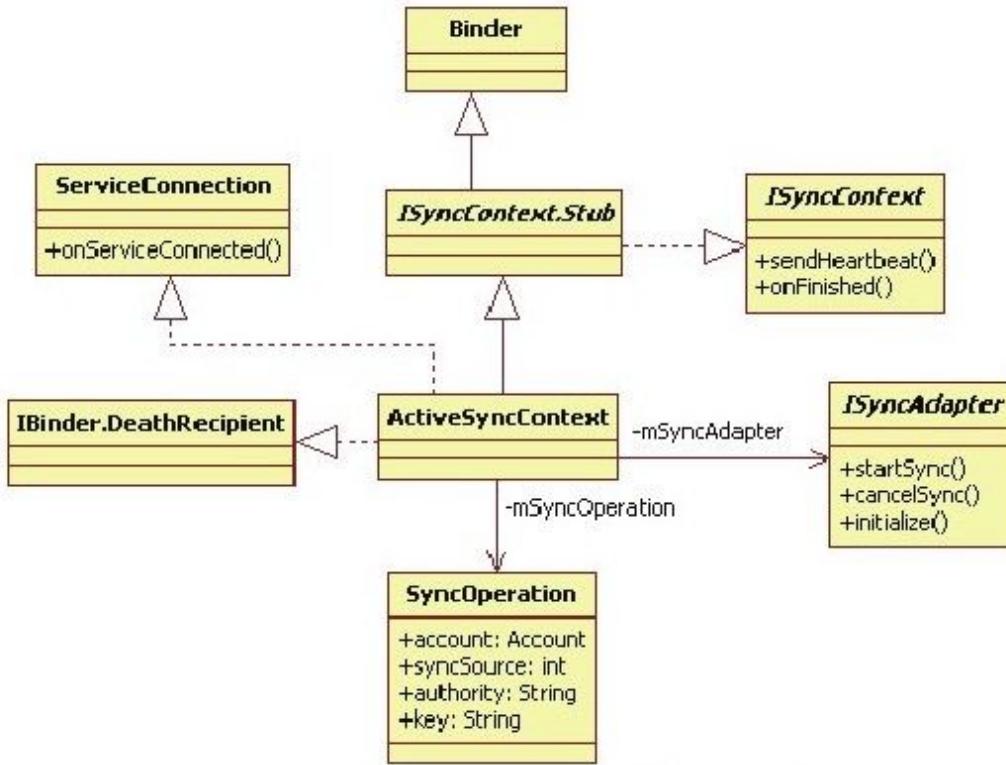


图 8-16 ActiveSyncContext 的 UML 类图

图 8-16 中的 ActiveSyncContext 和图 8-8 中的 Session 非常像。ActiveSyncContext 的主要工作包括下面两部分。

它将首先通过 bindService 方式启动 SyncService，并在 onServiceConnected 函数中得到用于和 SyncService 交互的接口对象，即参与 Binder 通信的 ISyncAdapter Bp 端。

ActiveSyncContext 是 ISyncContext 接口的 Binder 通信的 Bn 端，它在调用 ISync-Adapter 的 startSync 时，会把自己传递给同步服务。同步服务得到的当然是 ISyncContext 的 Bp 端对象。当同步服

务完成此次同步操作后就会调用ISyncContext的Bp端对象的onFinished函数以通知ActiveSyncContext同步操作的执行结果。

下面来看第一部分的工作。

(3) ActiveSyncContext派发请求

这部分的代码如下：

[-->SyncManager.java :
ActiveSyncContext.bindToSyncAdapter]

```
boolean bindToSyncAdapter  
(RegisteredServicesCache.ServiceInfo info) {  
    Intent intent=new Intent () ;  
    intent.setAction ("android.content.SyncAdapter") ;  
    //设置目标同步服务的ComponentName  
    intent.setComponent (info.componentName) ;  
    intent.putExtra (Intent.EXTRA_CLIENT_LABEL,  
    com.android.internal.R.string.sync_binding_label) ;  
    intent.putExtra (Intent.EXTRA_CLIENT_INTENT,  
    PendingIntent.getActivity (  
    mContext, 0,  
    new Intent (Settings.ACTION_SYNC_SETTINGS) , 0) ) ;  
    mBound=true ;  
    //调用bindService启动指定的同步服务  
    final boolean bindResult=mContext.bindService ( intent,  
this,  
    Context.BIND_AUTO_CREATE|Context.BIND_NOT_FOREGROUND  
    |Context.BIND_ALLOW_OOM_MANAGEMENT) ;  
    if (!bindResult)  
        mBound=false ;
```

```
    return bindResult;  
}
```

当目标SyncService从其onBind函数返回后，
ActiveSyncContext 的 onServiceConnected 将被调用，该函数的内部处理流程如下：

[-->SyncManager. java :
ActiveSyncContext.onServiceConnected]

```
public void onServiceConnected (ComponentName name, IBinder  
service) {  
    Message msg=mSyncHandler.obtainMessage () ;  
    msg.what=SyncHandler.MESSAGE_SERVICE_CONNECTED ;  
    // 构造一个 ServiceConnectionData 对象，并发送  
MESSAGE_SERVICE_CONNECTED消息  
    //给mSyncHandler。第二个参数就是SyncService在onBind函数中返回的  
ISyncAdapter的  
    //Binder通信对象。不过在ActiveSyncContext中，它是Bp端  
    msg.obj=new ServiceConnectionData (this,  
    ISyncAdapter.Stub.asInterface (service) ) ;  
    mSyncHandler.sendMessage (msg) ;  
}
```

[-->SyncManager. java :
SyncHandler.handleMessage]

```
case SyncHandler.MESSAGE_SERVICE_CONNECTED :  
    ServiceConnectionData msgData= ( ServiceConnectionData )  
msg.obj ;  
    if (isSyncStillActive (msgData.activeSyncContext) )  
        //调用runBoundToSyncAdapter函数处理
```

```
    runBoundToSyncAdapter (msgData.activeSyncContext,  
    msgData.syncAdapter) ;  
    break ;  
}
```

[-->SyncManager.java :
runBoundToSyncAdapter]

```
private void runBoundToSyncAdapter (final ActiveSyncContext  
activeSyncContext,  
    ISyncAdapter syncAdapter) {  
    activeSyncContext.mSyncAdapter=syncAdapter ;  
    final SyncOperation  
    syncOperation=activeSyncContext.mSyncOperation ;  
    try{  
        activeSyncContext.mIsLinkedToDeath=true ;  
        syncAdapter.asBinder () .linkToDeath (activeSyncContext, 0) ;  
        //调用目标同步服务的startSync函数  
        syncAdapter.startSync (activeSyncContext,  
        syncOperation.authority,  
        syncOperation.account, syncOperation.extras) ;  
    }.....  
}
```

对SynManager工作的分析到此为止，下面将分析目标同步服务。

3.EmailSyncAdapterService处理请求

在本例中，目标同步服务位于EmailSyncAdapterService中，先看它通过onBind函数返回给ActiveSyncContext的是什么。

(1) onBind分析

这部分的代码如下：

[-->EmailSyncAdapterService. java : onBind]

```
public IBinder onBind (Intent intent) {  
    //sSyncAdapter是EmailSyncAdapterService的内部类对象，见下文解释  
    return sSyncAdapter.getSyncAdapterBinder () ;  
}
```

在以上代码中，sSyncAdapter的类型是EmailSyncAdapterService中的内部类SyncAdapterImpl。它的派生关系如图8-17所示。

由图8-17可知：

AbstractThreadSyncAdapter是核心类，其内部有一个成员变量mISyncAdapterImpl，该变量用于和ActiveSyncContext交互，是ISyncAdapter Binder通信的Bn端。该对象也是以上代码中onBind函数的返回值。

SyncThread从Thread派生。从这一点可看出，同步服务将创建工作线程来执行具体的同步操作。AbstractThreadSyncAdapter中的mSyncThreads保存该同步服务中所有的SyncThread对象。

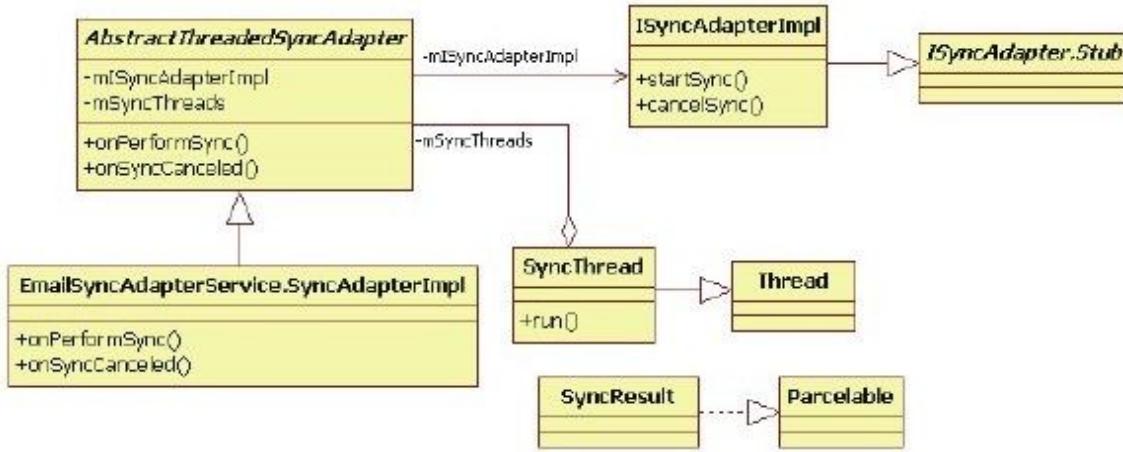


图 8-17 SyncAdapterImpl派生关系图

同步操作的结果将通过 SyncResult 返回给 SyncManager。

下面再来看 SyncManager runBoundToSyncAdapter 函数最后调用的 startSync 函数。

(2) startSync分析

在 SyncService 中，首先被调用的函数是 ISyncAdapterImpl 的 startSync 函数，其代码为：

[-->AbstractThreadedSyncAdapter.java :
ISyncAdapterImpl.startSync]

```

public void startSync ( ISyncContext syncContext, String
authority,
Account account, Bundle extras) {
//构造一个SyncContext对象，用于保存上下文信息

```

```
    final SyncContext syncContextClient=new SyncContext
(syncContext) ;
    boolean alreadyInProgress ;
    final Account threadsKey=toSyncKey (account) ;
    synchronized (mSyncThreadLock) {
        //判断是否存在已经在执行的SyncThread
        if (!mSyncThreads.containsKey (threadsKey) ) {
            if (mAtoInitialize
                &&extras !=null&&extras.getBoolean (
ContentResolver.SYNC_EXTRAS_INITIALIZE, false) ) {
                //一般而言，mAtoInitialize都为true，表示同步服务支持自动初始化
                //如果该服务对应的syncable状态为unknown，则重新设置syncable为1
                if (ContentResolver.getIsSyncable (account, authority) <0)
                    ContentResolver.setIsSyncable (account, authority, 1) ;
                //直接返回，不再做后续的处理，实际上后续的流程是可以继续进行的
                syncContextClient.onFinished (new SyncResult () ) ;
                return ;
            }
            //创建一个新的SyncThread对象
            SyncThread syncThread=new SyncThread (
                "SyncAdapterThread-"++
                mNumSyncStarts.incrementAndGet () ,
                syncContextClient, authority, account, extras) ;
            mSyncThreads.put (threadsKey, syncThread) ;
            syncThread.start () ;//启动工作线程
            alreadyInProgress=false ;
        }else{
            alreadyInProgress=true ;
        }
    }
    if (alreadyInProgress)
        syncContextClient.onFinished
(SyncResult.ALREADY_IN_PROGRESS) ;
}
```

假如没有匹配的工作线程（根据account生成一个key作为标志来查找是否已经存在对应的工作线程），SyncService将创建一个SyncThread，其run函数代码如下：

[-->AbstractThreadedSyncAdapter.java :
ISyncAdapterImpl.run]

```
public void run () {
    Process.setThreadPriority
    (Process.THREAD_PRIORITY_BACKGROUND) ;
    SyncResult syncResult=new SyncResult () ;
    ContentProviderClient provider=null;
    try{
        if (isCanceled ()) return;
        // 获得同步操作指定的ContentProvider, provider 是
        ContentProviderClient
        //类型, 用于和目标ContentProvider交互
        provider=mContext.getContentResolver () .
        acquireContentProviderClient (mAuthority) ;
        if (provider !=null) {
            //调用AbstractThreadedSyncAdapter子类的onPerformSync函数
            AbstractThreadedSyncAdapter.this.onPerformSync (mAccount,
                mExtras, mAuthority, provider, syncResult) ;
        }else
            syncResult.databaseError=true ;
    }finally{
        if (provider !=null)
            provider.release () ;
        if (!isCanceled ()) //通知结果
            mSyncContext.onFinished (syncResult) ;
        //工作完成, 将该线程从mSyncThreads中移除
        synchronized (mSyncThreadLock) {
```

```
mSyncThreads. remove (mThreadsKey) ;  
}  
}  
}
```

来看AbstractThreadedSyncAdapter子类实现的onPerformSync函数，在本例中，子类是SyncAdapterImpl，代码如下：

[-->EmailSyncAdapterService.java :
SyncAdapterImpl.onPerformSync]

```
public void onPerformSync (Account account, Bundle extras,  
String authority,  
ContentProviderClient provider, SyncResult syncResult) {  
try{  
    //调用EmailSyncAdapterService performSync完成真正的同步，这部分  
    代码和  
    //Email业务逻辑相关，此处不再深入研究  
    EmailSyncAdapterService.performSync ( mContext, account,  
extras,  
    authority, provider, syncResult) ;  
}.....  
}
```

执行完onPerformSync函数后，ISyncAdapterImpl.run返回前会调用mSyncContext.onFinished函数，通知位于SyncManager中的ActiveSyncContext同步操作的结果。读者可自行研究这部分内容。

4.ContentResolver requestSync分析总结

总结 requestSync 的工作流程，如图 8-18 所示。

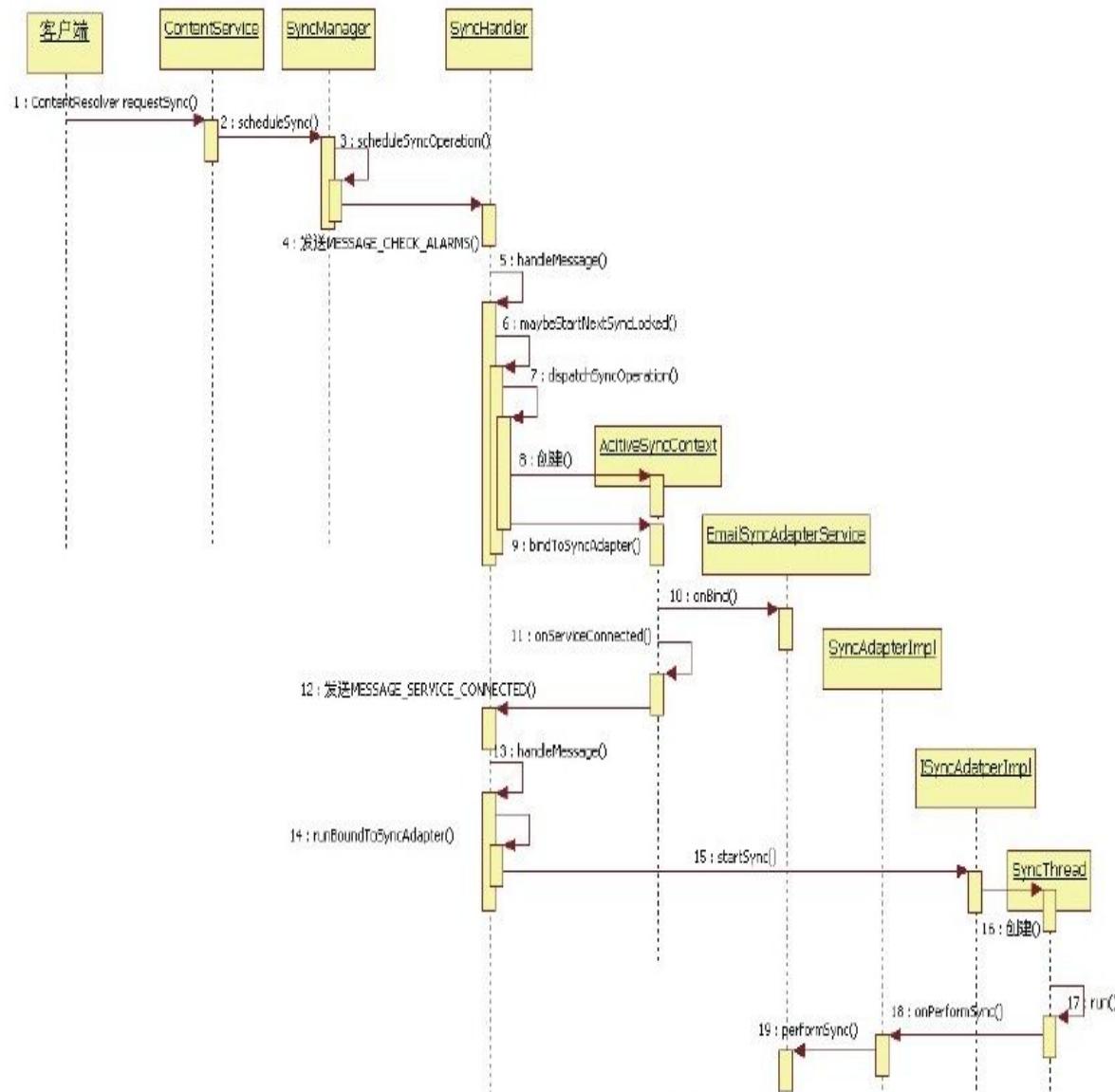


图 8-18 requestSync流

由图8-18可知，requestSync涉及的对象及调用流程比较烦琐。但从技术上看，则没有什么需

要特別注意的地方。

8.4.3 数据同步管理SyncManager分析总结

本节对ContentService中第二个主要功能即数据同步管理SyncManager进行了研究，这部分内容主要包括3个方面：

SyncManager及其相关成员的作用。

通过requestSync展示了SyncManager各个模块的作用及交互过程。

穿插于上述两方面之中的是数据同步的一些处理策略和规则。笔者未对这部分内容展开更细致的讨论。感兴趣的读者可自行学习。

8.5 本章学习指导

本 章 对 ContentService 和 AccountManagerService 进行了研究，在学习过程中，需注意以下几方面。

ContentService 包括两个不同部分，一个是数据更新通知机制，另一个是同步服务管理。在这两部分中，ContentService 承担了数据更新通知机制的工作，同步服务管理的工作则委托给 SyncManager 来完成。基于这种分工，本章先分析了 ContentService 的数据更新通知机制。这部分内容非常简单，读者应能轻松掌握。不过，作为深入学习的入口，笔者有意留了几个问题，旨在让读者仔细思考。

由于同步服务管理和 AccountManagerService 关系密切，因此本章先分析了 AccountManagerService。在这部分代码中，读者需要了解 RegisteredServicesCache 的作用。另外，感兴趣的读者也可以学习如何编写 AuthenticatorService，相关文档在 SDK 关于 AbstractAccountAuthenticator 类的说明中。

SyncManager是本章理解难度最大的知识点。笔者觉得，其难度主要体现在同步策略上。另外，缺乏相关文档资料的参考也是导致学习SyncManager难度较大的原因之一。读者在把本部分内容搞清楚的基础上，可自行研究本章没有提及的内容。另外，读者如想学习如何编写同步服务，可参考SDK文档中关于AbstractThreadSync-Adapter的说明。

除上述内容外，AccountManager的ddAccount函数实现中所使用的Java concurrent类，也是读者必须学习并掌握的基础知识。

8.6 本章小结

本 章 对 ContentService 和 AccountManagerService 进行了较为深入的探讨，其中：

8.2 节和8.4节分别分析了ContentService中数据更新通知机制的实现及同步服务管理方面的工作。

8.3 节 分 析 了 AccountManagerService 及 addAccount的实现。

本章留出以下问题供读者自行研究，这些问题包括：

8.2 节最后提到的开放性问题和Cursor query 中ContentObserver的使用分析。

8.4 节涉及SyncManager的同步请求处理的策略和maybeStartNextSyncLocked函数分析等。另外，读者如果有兴趣，不妨研究一下 SyncStorageEngine中同步信息的统计等内容。

“深入理解Android”系列书籍的规划路线图

一、Roadmap

“深入理解Android”系列书籍从卷I推出以后就受到广大读者的喜爱。在和读者交流的过程中，笔者被问及最多的一个问题就是，卷 II什么时候推出？内容会是什么？实际上，笔者和策划编辑杨福川在系列书籍的编写过程中，也在考虑这个问题：Android涉及的内容简直是浩如烟海，然而，哪些知识点能帮助读者更快、更好地了解Android？一方面帮助大家在深入了解Android系统的基础上，能更娴熟地进行应用开发；另一方面能帮助读者搭建一个兼具Android甚至嵌入式系统的具有相当深度和广度的知识架构？在反复讨论和仔细研究之后，我们试规划了如图1所示的“深入理解Android”系列书籍的Roadmap：



图 1 “深入理解Android”系列书籍的Roadmap

图1将整个系列分为4个部分：应用部分、框架部分、专题部分和内核部分。这几部分内容规划的大致思路为：

1. 应用部分

这部分拟以Android源码中自带的那些应用程序为分析目标，充分展示Google在自家SDK平台上做应用开发的深厚功力。这些应用包括Contacts、Gallery2、Mms、Browser等，它们的分析难度都不可小觑。通过对这些系出名门的应用的分析，我们希望读者不仅能把握商业级应用程序开发的精髓，还能更精熟地掌握Android应用开发的各种技能。

2. Framework部分

这部分关注Android的框架，拟包括3本书：

卷I：以Native层Framework模块为分析对象。知识点包括init、binder、zygote、jni、Message和Handler、audio系统、surface系统、vold、rild和mediascanner。本书已于2011年9月出版，虽然是基于Android 2.2的，读者如果扎实地掌握并理解了其中的内容，那么以后再研究2.3或4.0版本中对应的模块，也是轻而易举之事。

卷II和卷III：以Java层Framework模块为分析对象。卷II基于4.0.1版，包括UI相关服务和Window系统之外的一些重要服务，如PackageManagerService、ActivityManagerService、PowerManagerService、ContentService、ContentProvider等。而卷III将以输入系统、 WindowManagerService、UI相关服务为主要目标。相对于其他模块来说，UI相关服务可能会随着Android系统升级而发生较大的变化，所以卷III也许会基于Android 4.1。

Framework部分所包括的这3本书的目的是让读者对整个Android系统有较大广度、一定深度的认识，这有益于读者构建一个更为完整的Android系统知识结构。应当指出，这3本书不可能覆盖Android Framework中的所有知识点，因此，尚需

读者在此基础上，结合不同需求，进行进一步的深入研究。

3.专题部分

这部分旨在帮助读者沿着Android平台中的某些专业方向，进行深度挖掘，拟规划如下专题：

Telephony专题：涵盖SystemServer中相关的通信服务、rild、短信、电话等模块。

多媒体专题：涵盖MultiMedia相关的模块，包括Stagefright、OMX等。另外，我们也打算引入开源世界中最流行的一些编解码引擎和播放引擎作为分析对象。

浏览器和Webkit专题，该专题难度非常大，但其重要性却不言而喻。

Dalvik虚拟机专题：该专题希望对Dalvik进行一番深度研究，涉及面包括Java虚拟机的实现、Android的一些特殊定制等内容。

Android系统安全专题：该专题的目标是分析Android系统上提供的安全方面的控制机制。另外，Linux平台上的一些常用安全机制（例如，文件系统加密等）也是本书所要考虑的。

UI/UE设计以及心理学专题：该专题希望能提供一些心理学方面的指导以及具体的UI/UE设计方面的指南以帮助开发人员开发出更美、更体贴和更方便的应用。

专题部分隐含着的一个极为重要的宗旨：即基于Android，而高于Android。换言之，这些书籍虽都以Android为切入点，但我们更希望读者学到的知识、掌握的技术却不局限于Android平台。

4.内核部分

这部分拟以Linux内核为主。虽然这方面的经典教材非常多，但要么是诸如《Linux内核情景分析》之类的鸿篇巨帙，要么是类似《Linux内核设计与实现》，内容过于简洁。另外，现有书籍使用的内核源码都已比较陈旧。为此，我们希望能有一本难度适中、知识面较广、深度适宜的书籍。

另外，除了通过书籍传播知识外，我们也会引入博客等其他渠道来补充Roadmap中未能涉及的知识点。

二、英雄帖

Google为这个世界贡献了Android这一奇葩，那么我们能做些什么呢？我们的理想是，围绕深

入理解Android这一系列，编撰出国内最高品质的系列书籍，以飨读者。如果读者是正在前沿拼杀的“攻城狮”，那么我们甘愿做及时为您运送精神食粮的后勤兵！

主持人杨澜有一句话说得好：“原来我只佩服成功的人，现在我更尊敬那些正在努力的人”。我们也盛情邀请那些已付出艰辛努力或正在努力、有热情、懂技术，并乐于与人分享的朋友们加入到我们这个团队里来，共同为“攻城狮”们做些实事！目前卷III、Telephony专题已有合适的后勤兵们正在为之努力奋斗，相信不久的将来读者们就能享受到他们通过奋斗结出的硕果了！

请有意的朋友们和我们联系，我们的电子邮箱：

杨福川：linux1689@gmail.com 邓凡平：
fanping.deng@gmail.com

杨福川、邓凡平



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se singlelogin.re go-to-zlibrary.se single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>