



经典畅销书系“深入理解Android”系列Framework模块续篇。数十万
Android开发工程师首选读物。

从源代码层面全面、详细剖析了Android框架UI系统的实现原理和工作机制，
以及优秀代码的设计思想，填补市场空白。

移动开发



盛大伟〇著

Understanding Android Internals, Volume III

深入理解Android 卷 III



机械工业出版社
China Machine Press

目录

[推荐序](#)

[前言](#)

[第 1 章 开发环境部署](#)

[1.1 获取Android源代码](#)

[1.2 Android的编译](#)

[1.3 在IDE中导入Android源代码](#)

[1.3.1 将Android源代码导入Eclipse](#)

[1.3.2 将Android源代码导入SourceInsight](#)

[1.4 调试Android源代码](#)

[1.4.1 使用Eclipse调试Android Java源代码](#)

[1.4.2 使用gdb调试Android C/C++源代码](#)

[1.5 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分析](#)

[2.2.1 Binder架构总览](#)

[2.2.2 初始化Java层Binder框架](#)

[2.2.3 窥一斑，可见全豹乎](#)

[2.2.4 理解AIDL](#)

[2.2.5 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AudioService](#)

[3.1 概述](#)

[3.2 音量管理](#)

[3.2.1 音量键的处理流程](#)

[3.2.2 通用的音量设置函数setStreamVolume \(\)](#)

[3.2.3 静音控制](#)

[3.2.4 音量控制小结](#)

[3.3 音频外设的管理](#)

[3.3.1 WiredAccessoryObserver设备状态的监控](#)

[3.3.2 AudioService的外设状态管理](#)

[3.3.3 音频外设管理小结](#)

[3.4 AudioFocus机制的实现](#)

[3.4.1 AudioFocus最简单的例子](#)

[3.4.2 AudioFocus实现原理简介](#)

[3.4.3 申请AudioFocus](#)

[3.4.4 释放AudioFocus](#)

[3.4.5 AudioFocus小结](#)

[3.5 AudioService的其他功能](#)

[3.6 本章小结](#)

[第 4 章 深入理解WindowManagerService](#)

[4.1 初识WindowManagerService](#)

[4.1.1 一个从命令行启动的动画窗口](#)

[4.1.2 WMS的构成](#)

[4.1.3 初识WMS的小结](#)

[4.2 WMS的窗口管理结构](#)

[4.2.1 理解WindowToken](#)

[4.2.2 理解WindowState](#)

[4.2.3 理解DisplayContent](#)

[4.3 理解窗口的显示次序](#)

[4.3.1 主序、子序和窗口类型](#)

[4.3.2 通过主序与子序确定窗口的次序](#)

[4.3.3 更新显示次序到Surface](#)

[4.3.4 关于显示次序的小结](#)

[4.4 窗口的布局](#)

[4.4.1 从relayoutWindow \(\) 开始](#)

[4.4.2 布局操作的外围代码分析](#)

[4.4.3 初探performLayoutAndPlaceSurfacesLockedInner \(\)](#)

[4.4.4 布局的前期处理](#)

[4.4.5 布局DisplayContent](#)

[4.4.6 布局的最终阶段](#)

[4.5 WMS的动画系统](#)

[4.5.1 Android动画原理简介](#)

[4.5.2 WMS的动画系统框架](#)

[4.5.3 WindowAnimator分析](#)

[4.5.4 深入理解窗口动画](#)

[4.5.5 交替运行的布局系统与动画系统](#)

[4.5.6 动画系统总结](#)

[4.6 本章小结](#)

[第 5 章 深入理解Android输入系统](#)

[5.1 初识Android输入系统](#)

[5.1.1 getevent与sendevent工具](#)

[5.1.2 Android输入系统简介](#)

[5.1.3 IMS的构成](#)

[5.2 原始事件的读取与加工](#)

[5.2.1 基础知识：INotify与Epoll](#)

[5.2.2 InputReader的总体流程](#)

[5.2.3 深入理解EventHub](#)

[5.2.4 深入理解InputReader](#)

[5.2.5 原始事件的读取与加工总结](#)

[5.3 输入事件的派发](#)

[5.3.1 通用事件派发流程](#)

[5.3.2 按键事件的派发](#)

[5.3.3 DispatcherPolicy与InputFilter](#)

[5.3.4 输入事件的派发总结](#)

[5.4 输入事件的发送、接收与反馈](#)

[5.4.1 深入理解InputChannel](#)

[5.4.2 连接InputDispatcher和窗口](#)

[5.4.3 事件的发送](#)

[5.4.4 事件的接收](#)

[5.4.5 事件的反馈与发送循环](#)

[5.4.6 输入事件的发送、接收与反馈总结](#)

[5.5 关于输入系统的其他重要话题](#)

[5.5.1 输入事件ANR的产生](#)

[5.5.2 焦点窗口的确定](#)

[5.5.3 以软件方式模拟用户操作](#)

[5.6 本章小结](#)

第6章 深入理解控件系统

6.1 初识Android的控件系统

6.1.1 另一种创建窗口的方法

6.1.2 控件系统的组成

6.2 深入理解WindowManager

6.2.1 WindowManager的创建与体系结构

6.2.2 通过WindowManagerGlobal添加窗口

6.2.3 更新窗口的布局

6.2.4 删除窗口

6.2.5 WindowManager的总结

6.3 深入理解ViewRootImpl

6.3.1 ViewRootImpl的创建及其重要的成员

6.3.2 控件系统的心跳：performTraversals ()

6.3.3 ViewRootImpl总结

6.4 深入理解控件树的绘制

6.4.1 理解Canvas

6.4.2 View.invalidate () 与脏区域

6.4.3 开始绘制

6.4.4 软件绘制的原理

6.4.5 硬件加速绘制的原理

6.4.6 使用绘图缓存

[6.4.7 控件动画](#)

[6.4.8 绘制控件树的总结](#)

[6.5 深入理解输入事件的派发](#)

[6.5.1 触摸模式](#)

[6.5.2 控件焦点](#)

[6.5.3 输入事件派发的综述](#)

[6.5.4 按键事件的派发](#)

[6.5.5 触摸事件的派发](#)

[6.5.6 输入事件派发的总结](#)

[6.6 Activity与控件系统](#)

[6.6.1 理解PhoneWindow](#)

[6.6.2 Activity窗口的创建与显示](#)

[6.7 本章小结](#)

[第 7 章 深入理解SystemUI](#)

[7.1 初识SystemUI](#)

[7.1.1 SystemUIService的启动](#)

[7.1.2 状态栏与导航栏的创建](#)

[7.1.3 理解IStatusBarService](#)

[7.1.4 SystemUI的体系结构](#)

[7.2 深入理解状态栏](#)

[7.2.1 状态栏窗口的创建与控件树结构](#)

[7.2.2 通知信息的管理与显示](#)

[7.2.3 系统状态图标区的管理与显示](#)

[7.2.4 状态栏总结](#)

[7.3 深入理解导航栏](#)

[7.3.1 导航栏的创建](#)

[7.3.2 虚拟按键的工作原理](#)

[7.3.3 SearchPanel](#)

[7.3.4 关于导航栏的其他话题](#)

[7.3.5 导航栏总结](#)

[7.4 禁用状态栏与导航栏的功能](#)

[7.4.1 如何禁用状态栏与导航栏的功能](#)

[7.4.2 StatusBarManagerService对禁用标记的维护](#)

[7.4.3 状态栏与导航栏对禁用标记的响应](#)

[7.5 理解SystemUIVisibility](#)

[7.5.1 SystemUIVisibility在系统中的漫游过程](#)

[7.5.2 SystemUIVisibility发挥作用](#)

[7.5.3 SystemUIVisibility总结](#)

[7.6 本章小结](#)

[第 8 章 深入理解Android壁纸](#)

[8.1 初识Android壁纸](#)

[8.2 深入理解动态壁纸](#)

[8.2.1 启动动态壁纸的方法](#)

[8.2.2 壁纸服务的启动原理](#)

[8.2.3 理解UpdateSurface \(\) 方法](#)

[8.2.4 壁纸的销毁](#)

[8.2.5 理解Engine的回调](#)

[8.3 深入理解静态壁纸——ImageWallpaper](#)

[8.3.1 获取用作静态壁纸的位图](#)

[8.3.2 静态壁纸位图的设置](#)

[8.3.3 连接静态壁纸的设置与获取——WallpaperObserver](#)

[8.4 WMS对壁纸窗口的特殊处理](#)

[8.4.1 壁纸窗口Z序的确定](#)

[8.4.2 壁纸窗口的可见性](#)

[8.4.3 壁纸窗口的动画](#)

[8.4.4 壁纸窗口总结](#)

[8.5 本章小结](#)

本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供

最新最全的优质电子书下载！！！

推荐序

回顾

今天是一个值得高兴的日子。历经两年多的艰苦奋斗，张大伟同学的这本著作，同时也是“深入理解Android”系列三卷中的最后一卷终于完成了。从2011年我和华章公司的杨福川编辑一起开创这一迄今为止国内Android技术书籍市场上唯一一套兼具广度和深度的“深入理解Android”系列书籍算起，四个年头已经过去。在这四年中，本系列书籍的作者们和出版社的编辑们共同奋斗，成果斐然：

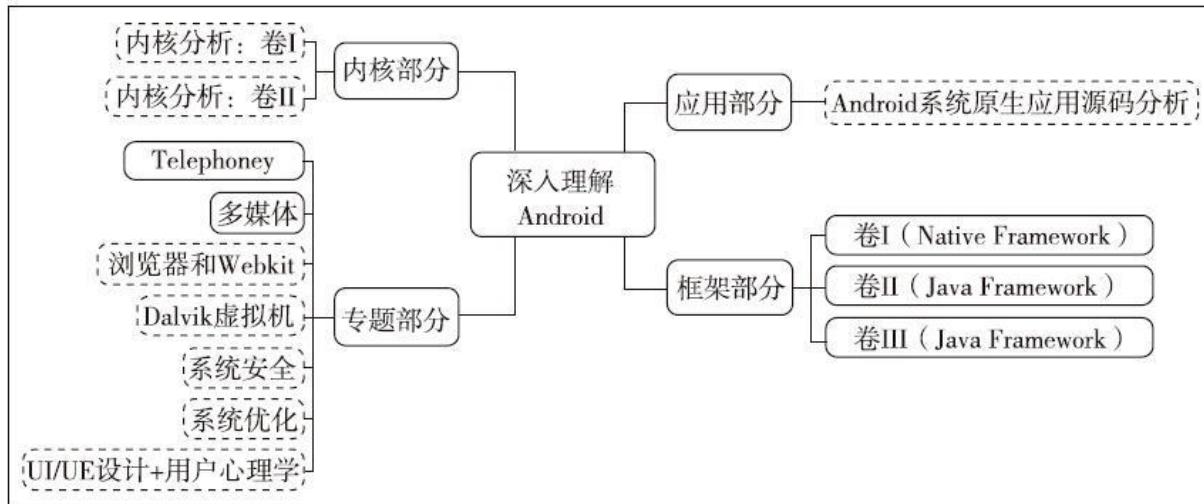
- 2011年9月 《深入理解Android：卷I》发布。
- 2012年8月 《深入理解Android：卷II》发布。
- 2013年1月，本系列的第一本专题卷《深入理解Android：Telephony原理剖析与最佳实践》发布，作者是杨青平。
- 2014年4月，本系列的第二本专题卷《深入理解Android：Wi-Fi，NFC和GPS卷》发布。
- 2015年，《深入理解Android：卷III》发布，作者即是本书的主人公张大伟。

·2015年及以后，我们还要发布“深入理解Android”系列书籍中的WebKit专题卷、自动化测试卷、蓝牙专题卷等。

从技术层面来说，本书填补了深入理解Android Framework卷中的一个主要空白，即Android Framework中和UI相关的部分。在一个特别讲究“颜值”的时代，本书分析了Android 4.2中WindowManagerService、ViewRoot、Input系统、StatusBar、Wallpaper等重要“颜值绘制/处理”模块。虽然在写书的两年中，Android版本已经从4.2进化到M，但“面虽新，神依在”。所以，我可以很负责任地说，对那些掌握了本书精髓的读者而言，即使Android未来升级到了X，那也只不过是换了一个“马甲”罢了。

展望

我在《深入理解Android：卷II》中曾经详细阐述过“深入理解Android”这一系列的路线图 (<http://blog.csdn.net/innost/article/details/7648869>) 。



本系列书大体分为应用部分、Framework部分、专题部分和内核部分。

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平台上进行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，更能精确熟练地掌握Android应用开发的各种技能。

2) Framework部分。关注Android的框架，包括三本书。

· 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，读者

如若扎实地掌握并理解了其中的内容，那么以后再研究Android 2.3或4.0版本中对应的模块，也是轻而易举之事了。

·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。而的卷III将以输入系统、WindowManagerService、UI相关服务为主要目标。

Framework部分这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平台中的某一些专业方向，进行深度挖掘，这一部分拟规划如下专题：

·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，涉及包括Java虚拟机的实现、Android的一些特殊定制等内容。现在来看，Dalvik已经被ART替换，所以这本书的目标就应该是ART虚拟机专题了。
 - 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。这套安全专题我已经在自己的博客上写了部分内容，包括Java Security、设备加密等。
 - 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。

今天，正是由于大伟的努力，我们的Framework部分得以完美收官。高兴的同时，我们认为前路依然艰辛。在此，我和福川兄再次诚挚邀请国内外有热情、愿分享、有责任心的兄弟姐妹们来一起继续发扬光大“深入理解Android”这一系列书籍。

还是杨澜的那句话，“原来我只佩服成功的人，现在我更尊敬那些正在努力的人”。让我们一起成为被尊敬的人吧！

轶事

我和大伟相知相识的过程还颇有点意思。

那时我们都在中科创达工作，有一次，我们俩要一起重构一个和音频相关的解码模块。当时我噼噼啪啪把几段和多线程相关的同步代码块改写后，引起了大伟的强烈质疑。在质疑（challenge）和争论（argue）的过程中，我发现大伟思路清晰，技术能力较强，是一个不可多得的好苗子，便有意交往。虽然吵得很激烈，不过最终实践的结果是这次改写比较成功，这使得我赢得了大伟的信任。

交手过后，我们便成了好兄弟。2012年夏天，我和大伟被派遣到上海高通公司。当时我刚完成了卷II的撰写，同时也在思考很多读者提出的

一个问题，即什么时候能详细分析一下Android Framework UI部分。古语云“书如其人”，对于我这样一个对“颜值”不是很讲究的人来说，写这本书肯定不是最合适的选择。因为我觉得这边书的作者需要耐心、细心，同时还需要一定审美观。在我认识的技术能力较强的兄弟们中，大伟无疑是最适合撰写本书的人选。

当然，对于一个从未写过书籍的人而言，这样的重任最初还是让大伟觉得紧张，感觉没有信心。所以，我和大伟一起参与出版合同签署事宜，让他觉得自己不是孤身作战。另外，在一些技术难点上，我会编写一些小例子，让大伟去完善，并以这些例子为出发点来分析Framework的实现。最后，大伟凭借自己的天分和努力，很快就从一个跟随者变成了这本书的主导者和唯一作者。

在本书的审稿过程中，我很欣慰地发现这本书细节深入、知识全面，是一本诚意之作。在此，我个人非常感谢大伟的努力，这本书了却了我多年的一桩心愿。

我曾经很羡慕那些有战友之情的士兵们。在和平年代的今天，我觉得我和大伟、福川、杨青平等作者、编辑都曾为了一个共同目标一起努力过，奋斗过，我们之间的感情应该能够媲美战友之情吧。

邓凡平

本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供

最新最全的优质电子书下载！！！

前言

本书的主要内容及特色

本书是“深入理解Android”系列的第三本，也是完结篇。按照“深入理解Android”系列图书的路线图，本书所关注的重点是Android中有关用户交互的Framework的知识。总体来说，本书所涵盖的内容分为两个部分：

- 第一部分是对构成Android用户交互基础的WindowManagerService、输入系统以及控件系统的介绍。

- 第二部分是以StatusBarManagerService、NotificationManagerService以及WallpaperManagerService为例，对Android在第一部分内容基础之上所实现的UI相关的服务进行探讨。

具体内容如下：

- 第1章介绍进行Android分析的一些准备工作，包括如何获取与编译代码，使用IDE进行代码的阅读及调试等。

- 第2章，根据邓凡平的建议，由《深入理解Android：卷II》第2章内容升级而来，将Android升级到4.2.2版本，并增加了与AIDL相关的内容。

Binder与MessageQueue是Android进程间通信与任务调度的重要工具。因此，进行Android的深入研究之前理解这两个工具的工作原理十分重要。

- 第3章主要介绍与AudioService服务相关的内容，包括音量控制、AudioFocus以及音量控制面板等内容。
- 第4章介绍WindowManagerService的工作原理，其中涵盖与窗口的创建、布局及动画相关的知识。
- 第5章介绍Android输入系统的工作原理，主要探讨输入事件的监听、读取、翻译、封装以及派发循环等内容。
- 第6章介绍Android控件系统的工作原理，包括控件系统的测量、布局、绘制、动画以及输入事件的派发。
- 第7章主要介绍与SystemUI相关的内容，其中包括StatusBarManagerService与NotificationManagerService两个系统服务，以及与状态栏和导航栏有关的知识。
- 第8章介绍与Android壁纸相关的内容，包括WallpaperManagerService系统服务、动态壁纸与静态壁纸的工作原理。另外还介绍 WindowManagerService对壁纸窗口的一些特殊处理。

其中第1章和第2章是全书的基础。第3章的内容相对独立，主要介绍与用户交互直接相关的音频方面的知识。而第4~6章是本书的重点内容，介绍Android UI的通用实现。在深入理解这三章的知识之后，读者可以通过借鉴第7章和第8章所介绍的SystemUI与壁纸的架构来提高Android与用户进行交互的深度定制能力。另外，Android源代码作为一个优秀的开源项目，大到架构的设计，小到某段代码的实现，都包含值得我们细细品味与吸收的设计思想，并且可以应用于自己所设计的代码上。因此，本书在代码分析的过程中尽可能地给出Android采用某种特定实现的原因或对其优秀的设计思路进行提取，希望读者能够知其然更知其所以然，进而能够在代码研究的过程中跳出代码的具体实现来体会其设计思想，而这正是本书根本目的所在。

读者对象

·Android应用开发者。

通过本书可以理解SDK中与用户交互相关的API或工具的工作原理，而拥有这部分知识有助于应用开发者设计出更健壮、更有效率，而且更加细腻的代码实现。

·Android系统开发工程师。

Android系统开发工程师将是本书所面向的最主要的读者群。同“深入理解Android”系列的其他书籍一样，本书将为这些读者提供其最感兴趣的

系统实现方面的内容。

·对Android系统的运行原理感兴趣的读者。

Android系统源代码中所体现的设计思想并不仅仅局限于Android，它对Android以外的开发工作同样极具借鉴意义。

如何阅读本书

本书所讨论的Android版本号为4.2.2，读者可以通过本书第1章所介绍的方法获取或在线阅读此版本的源代码。因为版本差异可能会使得某些源文件与类定义的位置发生变化，读者可以通过IDE集成的代码搜索功能进行查找。截至本书结稿之日，Android的最新版本为L，即5.0。在这个版本中，与输入系统相关的代码从frameworks/base/services/input文件夹移动到frameworks/native/services/inputflinger中，但本书所介绍的内容在这个版本中仍然适用。

读者需要注意，自第4章起，后一章的部分内容会以上一章为基础，尤其是第4~6章。虽说更加关注某一部分的读者可以直接阅读相关章节，但是笔者建议在阅读过程中至少先完成第4章有关窗口管理与布局内容的学习，因为这部分知识是后续内容的基础中的基础。

本书沿用了“深入理解Android”系列图书的代码引用风格，即在每章的开篇给出所有引用代码的完整路径，并在引用某一段代码之前指明这

段代码来自哪个文件、哪个类的哪个方法（或函数），并以注释的方式对代码中的知识点进行介绍。如下所示：

```
[ WindowManagerService.java-->WindowManagerService.addWindow() ] //  
普通的单行注释 /* 多行 注释 */ // ① 粗体+数字编号表示了代码中需要  
读者留意的关键点
```

另外，作为“深入理解Android”系列图书的一员，本书的内容与卷I、卷II有一定的联系，例如卷I的Surface系统、卷II的ActivityManagerService等在本书都会有所提及。读者可以将其作为本书的补充资料。

勘误和支持

由于本书涉及的内容及代码量巨大而且复杂，加之笔者的水平限制，书中难免会有一些不准确甚至错误的地方，还望各位读者不吝批评指正。另外，Android仍处在快速发展的过程中，卷III的成书也绝不是笔者对Android系统深入研究的一个句号。因此，倘若读者有关于本书的任何问题或建议，都可以与笔者进行讨论。

致谢

首先要感谢华章公司的杨福川以及本书的编辑姜影，拙稿得以付梓离不开他们耐心的支持与细致的校正，在此向他们致以最诚挚的感谢与敬意！还要感谢邓凡平在本书编写过程中给予的指导与建议，更要感

谢他一直以来给予我的帮助与信任，能够为“深入理解Android”系列图书贡献一份力量真是我莫大的荣幸！

感谢我的妻子郭晓丽与我的家人。在我写书的过程中忽略了对他们的陪伴，而他们却给予了我一如既往的理解与支持。

还要感谢我在写作本书时所任职的中科创达与索尼移动通信两家公司的领导与同事，有幸与这样优秀的团队一起完成一个个富有挑战而又激动人心的工作让我由衷地感到开心，而他们给予我的信任与鼓励是我得以进步的最好动力。

还要感谢我的师长与朋友对我的关心与帮助，祝福他们！

最应当感谢的是关注本书的各位读者。倘若本书能够为各位的学习、工作尽些绵薄之力，这将是我最大的荣幸，而各位的意见、建议甚至批评则会是我努力的方向。

**本书由“ePUBw.COM”整理，ePUBw.COM 提供
最新最全的优质电子书下载！！！**

第1章 开发环境部署

本章的主要内容：

- 介绍获取Android源代码的方法
- 介绍如何将Android源代码导入IDE中以方便阅读代码
- 介绍如何对Android的Java、C/C++源代码进行调试

1.1 获取Android源代码

在深入研究Android之前，首先必须获得一套Android的源代码。Google提供官方Android源代码的获取方法如下：

<https://source.android.com/source/downloading.html>

这个页面介绍了使用repo脚本进行Android源码的下载的两个基本步骤。

1) 首先通过repo init命令将当前文件夹初始化为repo脚本的工作区。其命令格式如下：

```
repo init -u <repository地址> -b <分支名称>
```

repo init命令会在当前文件夹下创建一个.repo文件夹，并从-u参数所指定的repository中下载一个manifest.xml文件到这个.repo文件夹。这个manifest.xml文件定义了Android源代码中所有git项目的清单，如下所示：

```
[manifest.xml] <manifest> ..... <project name="platform/frameworks/base"  
path="frameworks/base" revision="..."/> <project  
name="platform/packages/apps/Music" path="packages/apps/Music"  
revision="..."/> ..... <!-- 其他项目的定义 --> </manifest>
```

其中每一个project项都描述了一个git项目，而每一个git项目中则包含了负责某项功能的源代码。其中，name属性指定了git项目的名称，path属性指定了git项目将被下载到哪一个文件夹，而revision则指定了需要下载git项目的哪一个分支。上述manifest.xml片段中所给出的两个git项目分别存储了Android基本框架的代码以及Music应用程序的代码，它们将被分别下载到frameworks/base以及packages/apps/Music文件夹。

2) 在完成repo工作区的初始化之后，便可以通过repo sync命令下载代码了。repo sync命令的原理就是解析.repo/manifest.xml中的内容，然后通过git工具逐个下载清单中所列举的git项目。repo sync可以接受-j参数进行多线程的代码下载以提高下载速度，例如repo sync -j8表示将使用8个线程。

由于整套Android源代码由数百个这样的git项目组成，因此进行Android源代码的完整下载是非常耗时的。倘若开发者只关心其中的某个部分，例如上述manifest.xml片段中所给出的Music应用程序的代码，那么可以这么做：

```
repo sync platform/packages/apps/Music
```

也就是说，在repo sync后面添加git项目的名称作为参数则可以单独下载这个项目的代码。在迫切地需要对Android的某个局部模块进行研究

时，这个命令十分有用。



说明

其实manifest.xml也隶属于一个git项目，而这个git项目的名称与下载地址正是通过-u参数所指定的repository。所以通过在repo init中的-b参数指定的不同的分支可以获得不同内容的manifest.xml，进而repo sync得以下载不同的Android源代码。

遗憾的是，Google官方所给出的repository所在的服务器在国内访问十分困难。除了使用官方提供的repository之外，一些芯片厂商提供的镜像repository可以用于源代码下载。通过官方服务器下载源代码遇到问题的读者可以在codeaurora.org以及omapzoom.org上找到用于下载Android源代码的镜像repository的地址。

倘若不需要进行代码编译及调试，那么在线阅读Android源代码无疑是一个非常方便的选择。基于OpenGrok代码搜索引擎的androidxref.com就是一个在线阅读Android源代码的站点。这个站点存储了自Android 1.6以来所有版本的Android源代码，并且在OpenGrok引擎的支持下可以非常快速地实现源代码的查找与跳转。如图1-1和图1-2所示。

AndroidXRef Jelly Bean 4.2.2

Home Sort by: last modified time | **relevance** | path

Full Search Definition Symbol mAnimator File Path WindowManagerService.java History

In Project(s) selectall invert selection

dalvik
development
device
docs
external
frameworks

Searched **+path:"windowmanagerservice . java" +refs:mAnimator** (Results 1 - 1 of 1) sorted by relevance

/frameworks/base/services/java/com/android/server/wm/

```
651 * locking on either mWindowMap or mAnimator and then on mLayoutToAnim */
654 final WindowAnimator mAnimator; field in class:WindowManagerService
782 mAnimator.mAboveUniverseLayer = mPolicy.getAboveUniverseLayer();
840 mAnimator = new WindowAnimator(this);
2759 // TODO(cmautner): synchronize on mAnimator or win.mWinAnimator.
3032 animating = mAnimator.mAnimating;
5051 mAnimator.getScreenRotationAnimationLocked(Display.DEFAULT_DISPLAY);
6117 mAnimator.getScreenRotationAnimationLocked(Display.DEFAULT_DISPLAY);
6948 mAnimator.setDisplayDimensions(dw, dh, appWidth, appHeight);
7254 mAnimator
[all...]
```

Completed in 107 milliseconds

图1-1 使用AndroidXRef进行代码查找

AndroidXRef Jelly Bean 4.2.2

xref: /frameworks/base/services/java/com/android/server/wm/Session.java

[Home](#) | [History](#) | [Annotate](#) | [Line#](#) | [Navigate](#) | [Download](#)

```
118
119     @Override
120     public boolean onTransact(int code, Parcel data, Parcel reply, int flags)
121             throws RemoteException {
122         try {
123             return super.onTransact(code, data, reply, flags);
124         } catch (RuntimeException e) {
125             // Log all 'real' exceptions thrown to the caller
126             if (!(e instanceof SecurityException)) {
127                 Slog.e(WindowManagerService.TAG, "Window Session Crash", e);
128             }
129             throw e;
130         }
131     }
132
133     public void binderDied() {
134         // Note: it is safe to call in to the input method manager
135         // here because we are not holding our lock.
136         try {
137             if (mService.mInputMethodManager != null) {
```

图1-2 使用AndroidXRef浏览代码

1.2 Android的编译

在将下载到本地的代码添加到Eclipse或其他IDE之前，最好先进行一次完整的Android编译。这是因为某些代码文件是在编译过程中由aidl文件或资源文件所生成的，只有经过完整编译之后才能保证导入IDE中的Android源代码的完整性。

编译Android源代码非常简单，其步骤如下：

- 1) 执行souce build/envsetup.sh，此命令将初始化Android的编译环境，并且声明一系列方便操作源代码的bash函数，如mmm、 mm、 cgrep、 jgrep等。
- 2) 输入lunch full-eng并执行。它是envsetup.sh中定义的一个函数，用于设置即将编译的项目以及类型。读者也可以通过等效的choosecombo命令对编译进行更精细设置。
- 3) 输入make并执行Android编译。编译的中间结果以及最终产物（包括由aidl文件与资源所生成的代码文件）都存储在Android源代码根目录下的out文件夹中。

1.3 在IDE中导入Android源代码

尽管Android的源代码并不依赖IDE进行编译，但是使用IDE进行代码的浏览、查找与跳转无疑是最快的选择。本书所涉及的Android源代码主要是由Java语言以及C/C++语言编写的。对Java代码来说，Eclipse是最佳选择，而对C/C++代码来说，本书推荐使用速度更快的SourceInsight。

1.3.1 将Android源代码导入Eclipse

首先需要将development/ide/eclipse/.classpath文件复制到源代码的根目录下。这个文件将在导入代码时告诉Eclipse在源代码的哪些文件夹中保存了Java代码，其内容片段如下：

```
[.classpath] ..... ..... .....
```

这个xml文件由一系列classpathentry组成，每一个classpathentry指定了一个包含Java代码的文件夹，而Eclipse会从这些文件夹中查找并导入Java代码。这其中一批文件夹的路径以out文件夹打头，它们就是在Android编译过程中所产生的代码。倘若在没有进行过完整编译的情况下进行代码导入就会因为这些代码的丢失而使得某些引用无法解析。另外，倘若读者对某些文件夹下的代码不感兴趣，可以在进行代码导

入前将它们从.classpath中注释掉以避免花费过长的导入时间以及过多的内存占用。就本书的内容而言，建议只保留framework相关的文件夹（注意，out文件夹下的framework相关文件夹也需要保留）。

修改完.classpath文件之后，便可以通过Eclipse下的菜单“File → New → Java Project”所打开的New Java Project对话框进行Android源代码的导入，如图1-3所示。在这个对话框中为项目取一个名字，然后将Location设置为存放Android源代码的根目录，然后点击Finish按钮就可以开始导入代码。这个过程会比较慢，读者需要耐心等待。

在导入完成之后，Android将会以一个Java工程的形式出现在Eclipse的Package Explorer中，接下来就可以在Eclipse中浏览代码了。

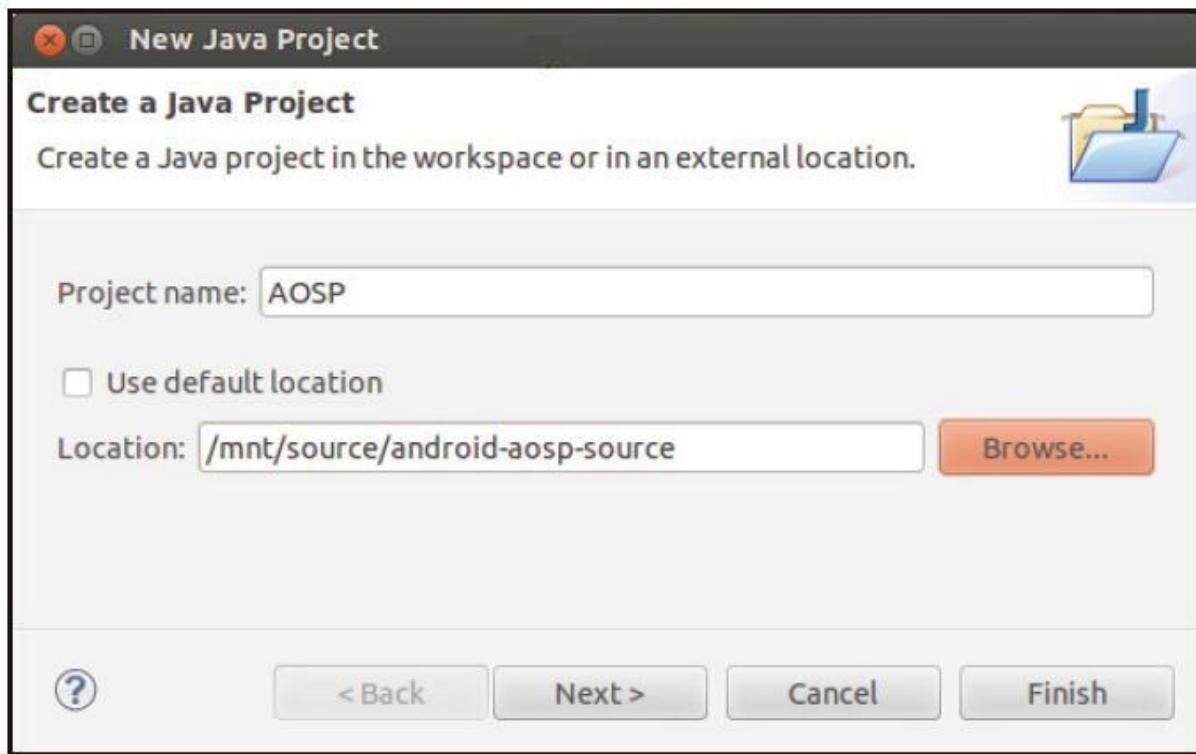


图1-3 在Eclipse中导入Android源代码



技巧

熟悉一些常用的快捷键可以极大地提高代码的阅读效率。其中最常用的有：

- CTRL+SHIFT+T**组合键，跳转到一个类或接口的定义。
- CTRL+O**组合键，跳转到当前文件中所定义的一个成员（内部类、成员变量或方法等）。
- CTRL+SHIFT+R**组合键，跳转到一个特定的代码文件。
- CTRL+SHIFT+G**组合键，查找选中的元素在工作区中的引用位置。

另外，可以在另一个Android程序的项目中将导入的Android源代码设置为一个依赖项目，这样就可以在APP代码中实现到Android Framework代码的无缝跳转。

1.3.2 将Android源代码导入SourceInsight

相对于Eclipse，SourceInsight更适合用来阅读Android中的C/C++代码。读者可以从其官方网站上下载并获得30天的免费试用权。

将Android源代码导入SourceInsight非常简单。点击主菜单上的“Project→New Project”，在弹出的对话框中为新项目取一个名字然后点击OK按钮，如图1-4所示。

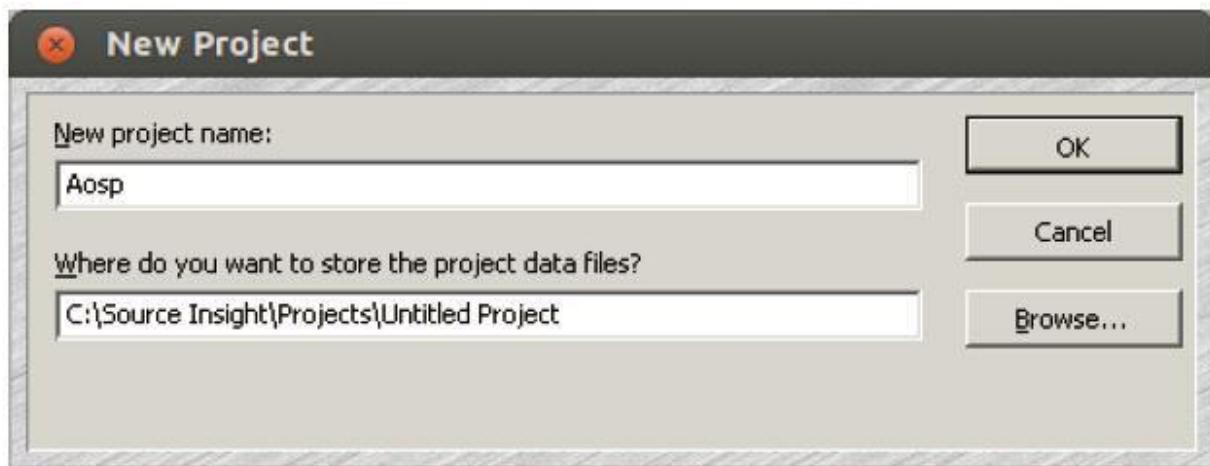


图1-4 新建SourceInsight项目

在弹出的对话框的Project Source Directory中设置好Android源代码的根目录，点击OK按钮，如图1-5所示。

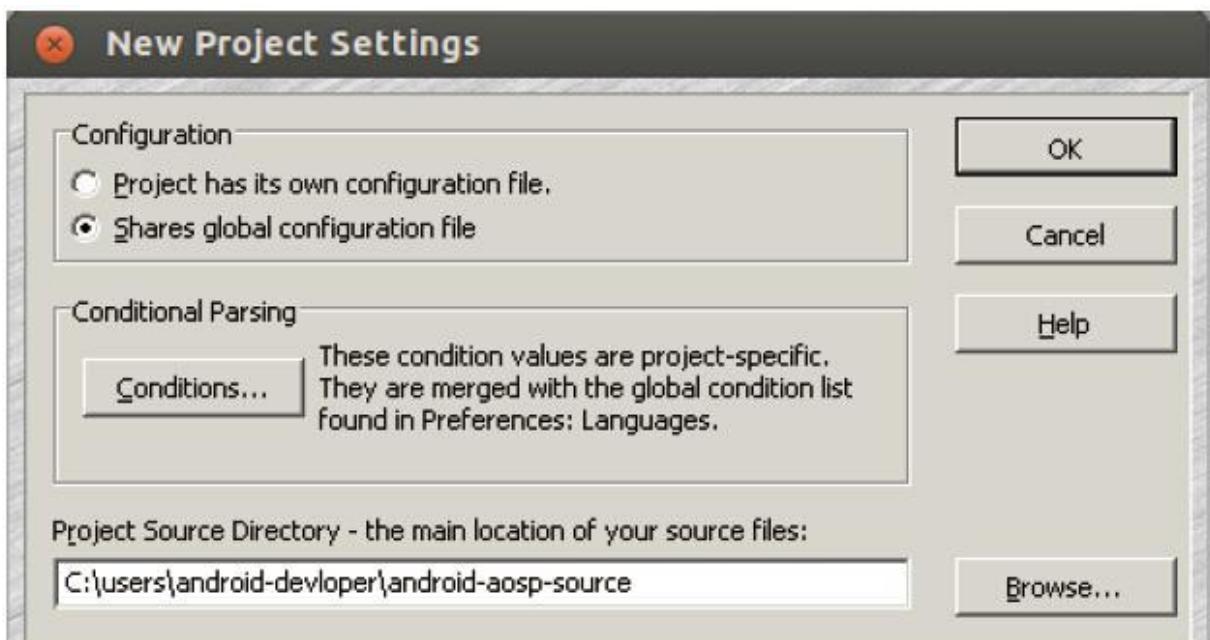


图1-5 设置Android源代码的根目录

在新对话框中选中希望导入的代码所在的文件夹，然后点击Add（仅导入选中文件夹下的代码）或Add Tree（导入选中文件夹及其子文件夹下的代码）按钮，如图1-6所示，之后点击Close按钮结束代码的导入。

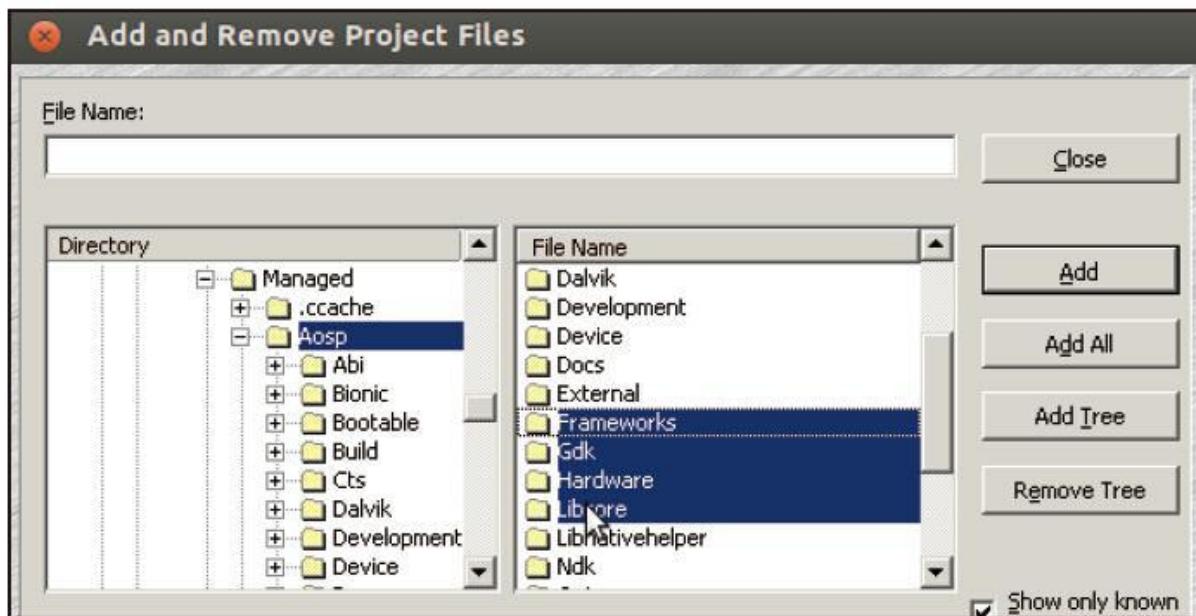


图1-6 在SourceInsight中选择导入的代码路径

接下来就可以方便地在SourceInsight中阅读代码了。



技巧

在SourceInsight中追加导入或移除代码文件十分方便。只要点击主菜单上的“Project → Add and Remove Project Files”就可以打开如图1-6所示的对话框，然后进行代码的追加导入或删除。

1.4 调试Android源代码

调试是分析问题与印证对代码的理解的最有效手段，对Android这种复杂而庞大的系统来说尤为如此。Android的源代码主要由Java代码以及C/C++代码构成，因此调试Android源代码需要从Java的调试以及C/C++的调试两个方面说起。

1.4.1 使用Eclipse调试Android Java源代码

由于Android源代码是以一个普通的Java工程的方式导入的，于是在Eclipse中不能通过ADT所提供的DDMS直接对其进行远程调试。

1) 首先需要通过DDMS获取调试进程的端口号。将设备通过USB连接PC，然后打开Eclipse的DDMS视图。在视图左侧Device进程列表中可以找到对应进程的调试端口号，如图1-7所示。

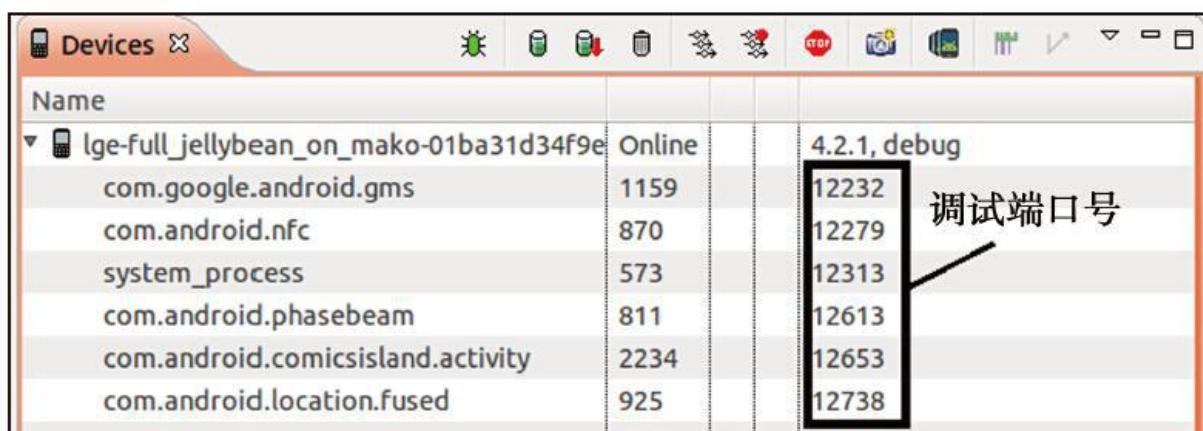


图1-7 获取调试目标进程的端口号

2) 回到Java视图，在Package Explorer中，用右键点击Android源代码所在的项目，在菜单中依次选择“Debug As → Debug Configurations”。在弹出的对话框中通过双击Remote Java Application新建一个远程调试配置。编辑Host为“localhost”，Port为调试进程的端口号之后点击Apply按钮保存配置。最后点击Debug按钮即可将Eclipse调试器绑定到对应的进程上，如图1-8所示。



图1-8 启动对目标进程的远程调试

完成上述步骤后就可以通过Eclipse的Debug视图调试进程了。



注意

在点击Debug进行调试时，Eclipse可能会提示代码中存在错误（最常见的原因是注释掉了.classpatch中的代码路径而导致引用解析失败）。不过不用理会，直接点击Proceed按钮继续调试即可。

导入Eclipse中的源代码可能运行在任何一个进程中，这份源代码可以对设备中任何一个Java进程进行调试。本书所介绍的各种系统服务均运行在system_server进程中，而其他的内容如SystemUI、WallpaperService则运行在其他的进程里。因此，读者需要注意正确选择调试目标进程。

1.4.2 使用gdb调试Android C/C++源代码

下面介绍使用gdb调试C/C++代码的步骤。

首先通过adb shell ps获取需要进行调试的进程号，比如795。然后通过执行adb shell进入手机端的shell。

输入gdbserver : 5039--attach 795并执行。其中5039是端口号，795是待调试的进程号。于是gdbserver便会绑定在795进程上，并通过5039端口与PC端的调试器进行通信。

保持gdbserver运行，然后回到PC端执行adb forward tcp : 5039 tcp : 5039。这个命令可以在设备上的5039端口与PC上的5039端口之间建立一个映射。于是PC端的调试器可以通过本机的5039端口与设备上的gdbserver进行通信。

接下来便需要运行Android源代码中附带的复合设备及其架构的gdb工具，并连接到本机的5039端口进行调试。对ARM架构来说，这个工具为prebuilts/gcc/linux-x86/arm/arm-eabi-4.X/bin/arm-eabi-gdb。



说明

Android源代码中提供了用于ARM、x86以及MIPS等目标机器架构的编译工具链。

倘若读者不清楚需要使用哪种机器架构下的编译工具链，可以先完成代码编译时source build/envsetup.sh以及choosecombo的执行以确定目标设备的类型。这样一来Android编译系统会将目标设备使用的机器架构对应的编译工具链所在的路径加入PATH环境变量中。然后就可以通过echo\$PATH得知用于当前设备的编译工具链所在的路径，进而得知机器架构的类型。

进入gdb之后，依次执行如下命令：

```
# 连接本机的5039端口，进而连接到运行在设备中的gdbserver <gdb>
target remote :5039 # 指定调试进程可执行文件的路径。注意需要选择编
译结果中symbols路径下的文件。这些文件中保存了 # 用于进行调试的
符号表 <gdb> file
out/target/product/<product_name>/symbols/system/bin/<proc_file_name>
# 设置用于搜索so文件的路径。注意需要选择编译结果中symbols路径
下的so文件所在的路径。读者 # 可以从
out/target/product/<product_name>/XXXgdb.cmds文件中的内容中得知这
个路径 <gdb> set solib-search-path .....
```

之后便可以开始使用gdb的命令调试795进程了。



技巧

倘若读者觉得使用上述步骤进行调试比较繁复，可以使用Android源代码中所提供的gdbclient工具。gdbclient是定义在build/envsetup.sh中的一个Shell函数，它会根据choosecombo的结果判断可执行文件的路径以及so文件的路径，并自动完成上文所述的工作。因此，使用它之前必须完成source build/envsetup以及choosecombo的执行。使用gdbclient调试一个进程的方法如下：

```
gdbclient <可执行文件名> :<端口号> <进程号>
```

例如，调试795的mediaserver进程可以使用如下命令：

```
gdbclient mediaserver :5039 795
```

另外，倘若使用gdb调试Java进程中的C/C++代码，需要使用app_process作为可执行文件进行调试。

1.5 本章小结

本章介绍了获取Android源代码、使用IDE进行源代码的阅读以及调试的方法。接下来让我们开始Android源代码的研究之旅吧。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第2章 深入理解Java Binder和 MessageQueue

本章主要内容：

- 介绍Binder系统的Java层框架

- 介绍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/intenal/os/BinderInternal.java

- android_util_Binder.cpp

frameworks/base/core/jni/android_util_Binder.cpp

·SystemServer.java

frameworks/base/services/java/com/android/servers/SystemServer.java

·ActivityManagerService.java

frameworks/base/services/java/com/android/servers/ActivityManagerService.java

·ServiceManager.java

frameworks/base/core/java/android/os/ServiceManager.java

·ServcieManagerNative.java

frameworks/base/core/java/android/os/ServcieManagerNative.java

·MessageQueue.java

frameworks/base/core/java/android/os/MessageQueue.java

·android_os_MessageQueue.cpp

frameworks/base/core/jni/android_os_MessageQueue.cpp

·Looper.cpp

frameworks/base/native/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

由于本书所介绍的内容是以Java层的系统服务为主，因此Binder相关的应用在本书中比比皆是。而MessageQueue作为Android中重要的任务调度工具，它的使用也是随处可见。所以本书有必要对这两个工具有所介绍。根据邓凡平的同意与推荐，本章由卷II第2章升级到4.2.2，并且增加了对AIDL相关知识点的分析。

本章作为本书Android源代码分析之旅的开篇，将重点关注两个基础知识点，它们是：

- 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。



建议

读者先阅读《深入理解Android：卷I》（以下简称“卷I”）的第6章“深入理解Binder”。网上有样章可下载。

2.2 Java层中的Binder分析

2.2.1 Binder架构总览

如果读者读过卷I的第6章，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。

由图2-1可知：

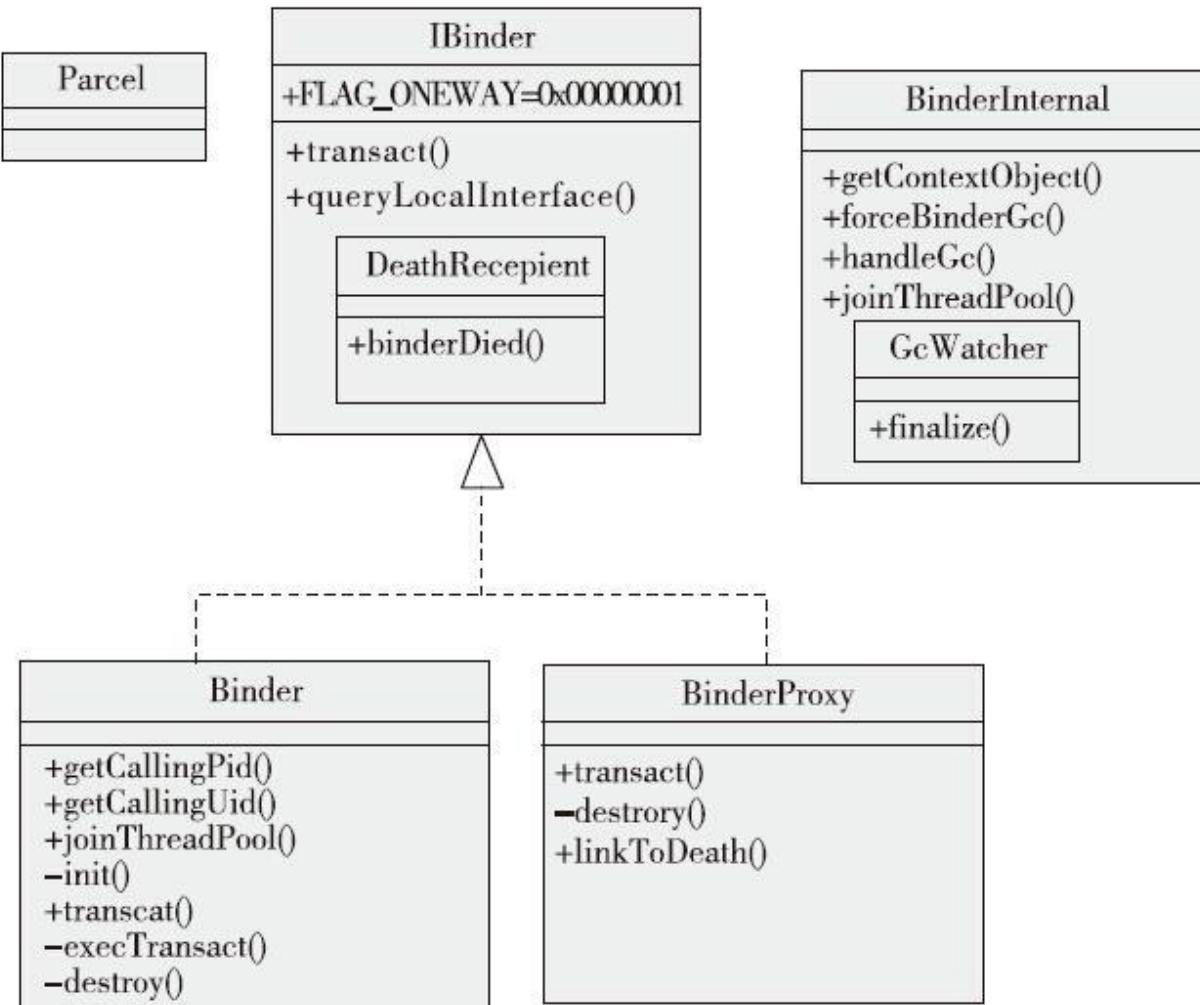


图2-1 Java层中的Binder家族

- 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框架使用的类。它内部有一个GcWatcher类，该类专门用于处理和Binder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叫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，在指明了FLAG_ONEWAY标志后，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：对使用FLAG_ONEWAY的函数来说，客户端仅向服务端发出请求，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系统的一个镜像，但这个镜像终究还需借助Native层Binder系统来开展工作，即镜像和Native层Binder有着千丝万缕的关系，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; ..... return 0; }
```

据上面的代码可知， register_android_os_Binder函数完成了Java Binder架构中最重要的三个类的初始化工作。

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execTranscat的methodID,Binder类中成员mObject的fieldID */

```

```
gBinderOffsets.mClass = (jclass) env->NewGlobalRef(clazz);  
gBinderOffsets.mExecTransact = env->GetMethodID(clazz, "execTransact",  
"(IIIZ"); 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)); }
```

由上面代码可知，gBinderOffsets对象保存了和Binder类相关的某些在JNI层中使用的信息。它们将用来在JNI层对Java层的Binder对象进行操作。execTransact () 函数以及mObject成员的用途将在2.2.3节介绍。



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：

- 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; // ①
gWeakReferenceOffsets用来和WeakReference类打交道 clazz = env->FindClass("java/lang/ref/WeakReference");
gWeakReferenceOffsets.mClass = (jclass) env->NewGlobalRef(clazz); // 获取WeakReference类get函数的MethodID gWeakReferenceOffsets.mGet=env->GetMethodID(clazz, "get", "()Ljava/lang/Object;"); // ②
gErrorOffsets用来和Error类打交道 clazz = env->FindClass("java/lang/Error");
gErrorOffsets.mClass = (jclass) env->NewGlobalRef(clazz); // ③
gBinderProxyOffsets用来和BinderProxy类打交道 clazz = env->FindClass(kBinderProxyPathName);
gBinderProxyOffsets.mClass = (jclass) env->NewGlobalRef(clazz);
gBinderProxyOffsets.mConstructor=env->GetMethodID(clazz, " ", "()V");
..... //获取BinderProxy的一些信息 // ④
gClassOffsets用来和Class类打交道 clazz = env->FindClass("java/lang/Class");
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)); }
```

据上面代码可知，`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。看来`BinderProxy`对象的生命周期会委托`WeakReference`来管理，所以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`。

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。

另外，这个过程中所创建的几个全局静态对象为JNI层访问Java层的对象提供了依据。而在每个初始化函数中所执行的`registerNativeMethods()`方法则为Java层访问JNI层打通了道路。换句话说，Binder初始化的工作就是通过JNI建立起Native Binder与Java Binder之间互相通信的桥梁。

下面通过一个例子来分析Java Binder的工作流程。

2.2.3 窥一斑，可见全豹乎

这个例子源自ActivityManagerService，我们试图通过它揭示Java层Binder的工作原理。先来描述一下该例子的分析步骤：

- 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，其代码如下所示：

```
[ActivityManagerService.java-->ActivityManagerService.setSystemProcess()] public static void setSystemProcess() { try { ActivityManagerService m = mSelf; // 将 ActivityManagerService服务注册到ServiceManager中 ServiceManager.addService("activity", m);..... } catch { ... } return; }
```

上面所示代码行的目的是将ActivityManagerService服务（以后简称AMS）加到ServiceManager中。

在整个Android系统中有一个Native的ServiceManager（以后简称SM）进程，它统筹管理Android系统上的所有服务。成为一个服务的首要条件是先在SM中注册。下面来看Java层的服务是如何向SM注册的。

1.向ServiceManager注册服务

(1) 创建ServiceManagerProxy

向SM注册服务的函数叫addService，其代码如下：

```
[ServiceManager.java-->ServiceManager.addService()] public static void  
addService(String name, IBinder service) { try { // getIServiceManager返回  
什么 getIServiceManager().addService(name, service); } ..... }
```

首先需要搞清楚getIServiceManager () 方法返回的是一个什么对象？

参考其实现：

```
[ServiceManager.java-->ServiceManager.getIServiceManager()] private  
static IServiceManager getIServiceManager() { ..... // 调用asInterface，传  
递的参数类型为IBinder sServiceManager =  
ServiceManagerNative.asInterface( BinderInternal.getContextObject());  
return sServiceManager; }
```

asInterface () 方法的参数为BinderInternal.getContextObject () 的返回值。于是这个简短的方法中有两个内容值得讨论：

BinderInternal.getContextObject () 以及asInterface () 。

BinderInternal.getContextObject () 方法是一个native函数，参考其实现：

```
[android_util_Binder.cpp-->android_os_BinderInternal_getContextObject()]

static jobject android_os_BinderInternal_getContextObject(JNIEnv* env,
jobject clazz) { /* 下面这句代码在卷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 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); }
```

可见，Java层的ServiceManager需要在Native层获取指向Native进程中ServiceManager的BpProxy。这个BpProxy不能由Java层的ServiceManager直接使用，于是

android_os_BinderInternal_getContextObject () 函数通过javaObjectForIBinder () 函数将创建一个封装了这个BpProxy的一个Java对象并返回给调用者。ServiceManager便可以通过这个Java对象实现对BpProxy的访问。参考这个Java对象的创建过程：

```
[android_util_Binder.cpp-->javaObjectForIBinder()] jobject
javaObjectForIBinder(JNIEnv* env, const sp & val) { // mProxyLock是一个
全局静态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，那就需要删除这个旧的对象*/ 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); } //

① 创建一个新的BinderProxy对象，并将它注册到Native BpBinder对象的ObjectManager中 jobject object = env-

>NewObject(gBinderProxyOffsets.mClass, gBinderProxyOffsets.mConstructor); if (object != NULL) { /* ② 把Native 层的BpProxy的指针保存到BinderProxy对象的成员字段mObject中。于是BinderProxy对象的Native方法可以通过mObject获取BpProxy对象的指针。这个操作是将BinderProxy与BpProxy联系起来的纽带 */ 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 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 (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的代码有点多，简单整理一下，可知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。

接下来讨论asInterface () 方法，大家还记得在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-->ServiceManagerNative.asInterface()] static
public IServiceManager asInterface(IBinder obj) { ..... // 以obj为参数，创
建一个ServiceManagerProxy对象 return new ServiceManagerProxy(obj); }
```

上面代码和Native层interface_cast宏非常类似，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。ServiceManagerProxy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。



说明

实际上BpProxy也不会直接和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：

```
[ServcieManagerNative.java-->ServiceManagerProxy.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，其实现函数的代码如下所示：

```
[android_util_Binder.cpp-->android_os_BinderProxy_transact()]
static
jboolean android_os_BinderProxy_transact(JNIEnv* env, jobject obj, jint
code, jobject dataObj, jobject replyObj, jint flags) { ..... // 从Java的Parcel对
象中得到作为参数的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最终还是要借助Native的Binder进行通信的。



说明

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，不论架构的编写采用的是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的目的虽然简单（即打开binder设备，然后读请求和写回复），但是架构复杂（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。脱离目的的实现，如缘木求鱼，很容易偏离事物本质。

在对addService进行分析时曾提示writeStrongBinder是一个特别的函数。那么它特别在哪里呢？下面将给出解释。

(3) 三人行之Binder、JavaBBinderHolder和JavaBBinder

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

```
[ActivityManagerNative.java-->ActivityManagerNative] 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-->ActivityManagerNative.ActivityManagerNative()] public  
ActivityManagerNative() { attachInterface(this, descriptor); // 该函数很简单  
// 读者可自行分析 }
```

而ActivityManagerNative父类的构造函数则是Binder的构造函数：

```
[Binder.java-->Binder.Binder()] public Binder() { init(); }
```

Binder构造函数会调用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。那么，JavaBBinderHolder是何方神圣呢？其定义如下：

```
[android_util_Binder.cpp-->JavaBBinderHolder] class JavaBBinderHolder :  
public RefBase { public: sp get(JNIEnv* env, jobject obj) { AutoMutex  
_l(mLock); sp 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 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_writeStrongBinder()] 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  
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; }
```

根据上面的介绍会发现，addService实际添加到Parcel的并不是AMS本
身，而是一个叫JavaBBinder的对象。而最终传递到Binder驱动的正是
这个JavaBBinder对象。

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Java Binder、JavaBBinderHolder和JavaBBinder三者的关系。

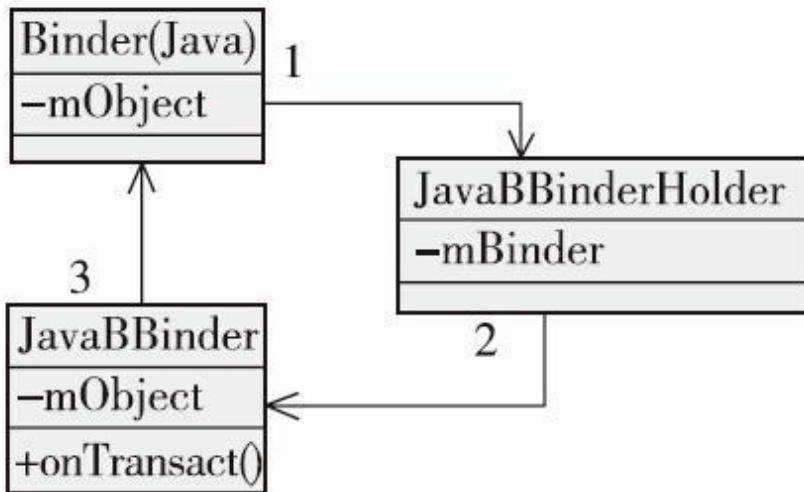


图2-2 Java Binder、JavaBBinderHolder和JavaBBinder三者的关系

从图2-2可知：

- 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JavaBBinder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

对此有更好的解释的读者，不妨与大家分享一下。

2.ActivityManagerService响应请求

初见JavaBBinde时，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。

而在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，我们必须查看它的onTransact函数。当收到请求时，系统会调用这个函数。



说明

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-->JavaBBinder::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  
execTranscat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exec-Transact () 方法，该方法在Binder类中实现，具体代码如下：

```
[Binder.java-->Binder.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); } catch { ... }  
reply.recycle(); data.recycle(); return res; }
```

ActivityManagerNative类实现了onTransact函数，代码如下：

```
[ActivityManagerNative.java-->ActivityManagerNative.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; } .... // 处理其  
他请求的情况 } }
```

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exeTransact。

- 该Binder对象的exeTransact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 Binder对象了。图2-3展示AMS响应请求的整个流程。

在图2-3中，右上角的大方框表示AMS对象，其中的虚线箭头表示调用子类重载的函数。

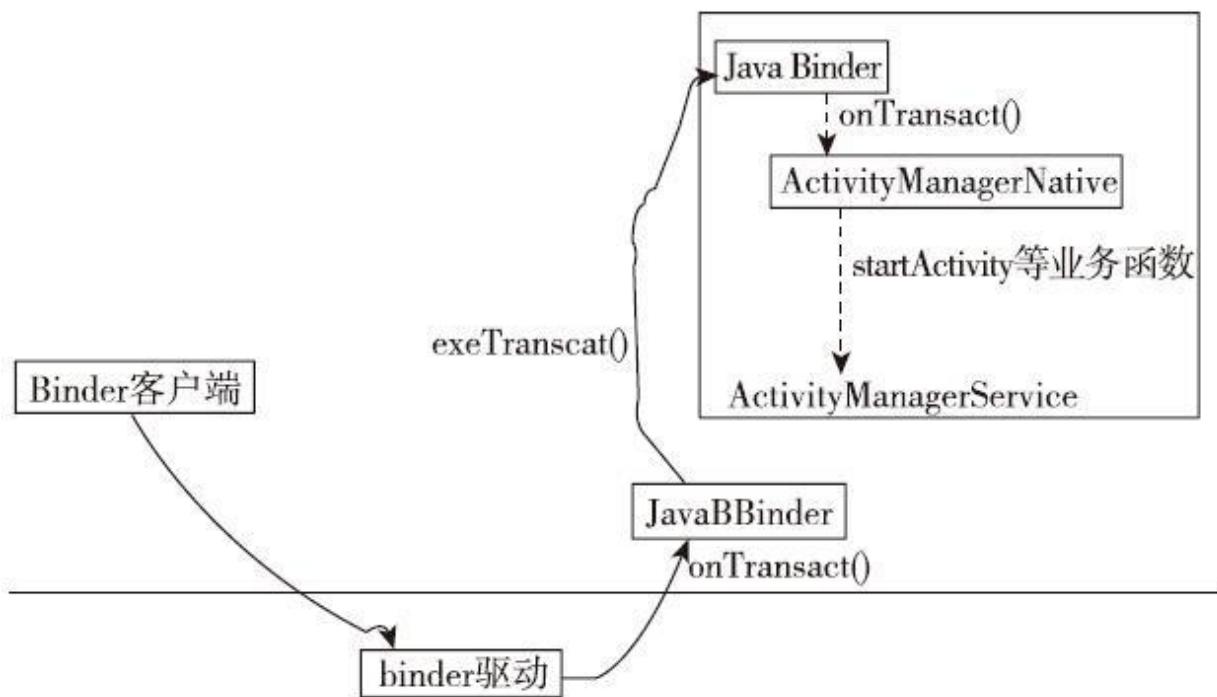


图2-3 AMS响应请求的流程

2.2.4 理解AIDL

经过上一节的介绍，你已经明白在Java层Binder的架构中，Bp端可以通过BinderProxy的transact () 方法与Bn端发送请求，而Bn端通过继承Binder类重写onTransact () 接收并处理来自Bp端的请求。这个结构非常清晰而且简单，但是实现起来却颇为烦琐。于是Android提供了AIDL语言以及AIDL解释器自动生成一个服务的Bn端，即Bp端用于处理Binder通信的代码。

AIDL的语法与定义一个Java接口的语法非常相似。为了避免业务实现对分析的干扰，本节通过一个最简单的例子对AIDL的原理进行介绍。

```
[IMyServer.aidl] package com.understanding.samples; interface IMyServer
{ int foo(String str); }
```

IMyServer.aidl定义了一个名为IMyServer的Binder服务，并提供了一个可以跨Binder调用的接口foo () 。可以通过aidl工具将其解析为一个实现了Bn端及Bp端通过Binder进行通信的Java源代码。具体命令如下：

```
aidl com/understanding/samples/IMyServer.aidl
```

生成的IMyServer.java可以在com/understanding/samples/文件夹下找到。



建议

读者可以阅读aidl有关的文档了解此工具的详细功能。

[IMyServer.java-->IMyServer] package com.understanding.samples; /* ①
首先，IMyServer.aidl被解析为一个Java接口IMyServer。这个接口定义了AIDL文件中所定义 的接口foo() */ public interface IMyServer extends android.os.IInterface { /* ② aidl工具生成了一个继承自IMyServer接口的抽象类IMyServer.Stub。这个抽象类实现了Bn 端通过onTransact()方法接收来自Bp端的请求的代码。本例中的foo()方法在这个类中会被定义 成一个抽象方法。因为aidl工具根本不知道foo()方法是做什么的，它只能在onTransact()中得知Bp端希 望对foo()方法进行调用，所以Stub类是抽象的 */ public static abstract class Stub extends android.os.Binder implements com.understanding.samples.IMyServer { // Stub类的其他实现 /* onTransact()根据code的值选择调用IMyServer接口中的不同方法。本例中 TRANSACTION_foo意味着需要通过调用foo()方法完成请求 */ public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) throws android.os.RemoteException {

```
switch (code) { ..... case TRANSACTION_foo: { ..... // 从data中读取参数
_arg0 // Stub类的子类需要实现foo()方法 int _result = this.foo(_arg0); .....
// 向reply中写入_result return true; } } return super.onTransact(code, data,
reply, flags); } /* ③ aidl工具还生成了一个继承自IMyServer接口的类
Proxy，它是Bp端的实现。与Bn端的Stub 类不同，它实现了foo()函数。
因为foo()函数在Bp端的实现是确定的，即将参数存储到Parcel 中,然后
执行transact()方法将请求发送给Bn端，然后从reply中读取返回值并返
回给调用者 */
private static class Proxy implements
com.understanding.samples.IMyServer { ..... // Proxy类的其他实现
public
int foo(java.lang.String str) throws android.os.RemoteException {
android.os.Parcel _data = android.os.Parcel.obtain(); android.os.Parcel
_reply = android.os.Parcel.obtain(); int _result; try { ..... // 将参数str写入参
数_data // mRemote就是指向IMyServer Bn端的BinderProxy
mRemote.transact(Stub.TRANSACTION_foo, _data, _reply, 0); .....// 从
_reply中读取返回值_result } finally { ..... } return _result; } } //
TRANSACTION_foo常量用于定义foo()方法的code static final int
TRANSACTION_foo =
(android.os.IBinder.FIRST_CALL_TRANSACTION + 0); } // 声明
IMyServer所提供的接口 public int foo(java.lang.String str) throws
android.os.RemoteException; }
```

可见一个AIDL文件被aidl工具解析之后会有三个产物：

·IMyServer接口。它仅仅用来在Java中声明IMyServer.aidl中所声明的接口。

·IMyServer.Stub类。这个继承自Binder类的抽象类实现了Bn端与Binder通信相关的代码。

·IMyServer.Stub.Proxy类。这个类实现了Bp端与Binder通信相关的代码。

在完成aidl的解析之后，为了实现一个Bn端，开发者需要继承IMyServer.Stub类并实现其抽象方法。如下所示：

```
class MyServer extends IMyServer.Stub { int foo(String str) { // 做点什么都  
可以 return str.length(); } }
```

于是每一个MyServer类的实例，都具有了作为Bn端的能力。典型的做法是将MyServer类的实例通过ServiceManager.addService () 将其注册为一个系统服务，或者在一个Android标准Service的onBind () 方法中将其作为返回值使之可以被其他进程访问。另外，也可以通过Binder调用将其传递给另外一个进程，使之成为一个跨进程的回调对象。

那么Bp端将如何使用IMyServer.Proxy呢？在Bp端所在进程中，一旦获取了IMyServer的BinderProxy（通过ServiceManager.getService () 、

onServiceConnected () 或者其他方式) , 就可以通过如下方式获得一个IMyServer.Proxy :

```
// 其中binderProxy就是通过ServiceManager.getService()获取的
IMyServer remote = IMyServer.Stub.asInterface(binderProxy);
remote.foo("Hello AIDL!");
```

IMyServer.Stub.asInterface () 的实现如下：

```
[IMyServer.java-->IMyServer.Stub.asInterface()]
public static
com.understanding.samples.IMyServer asInterface( android.os.IBinder obj)
{ ..... // 创建一个IMyServer.Stub.Proxy, 其中参数obj将会被保存为
Proxy类的mRemote成员 return new
com.understanding.samples.IMyServer.Stub.Proxy(obj); }
```

可见， AIDL使得构建一个Binder服务的工作大大简化了。

2.2.5 Java层Binder架构总结

图2-4展示了Java层的Binder架构。

根据图2-4可知：

- 对代表客户端的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。

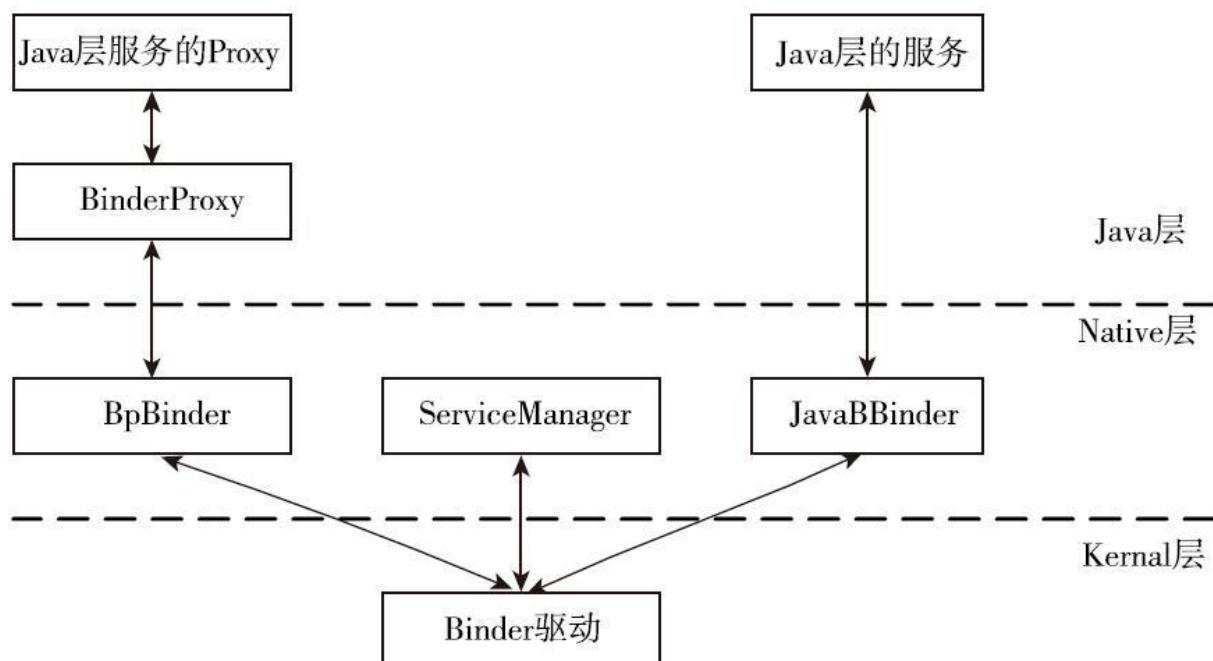


图2-4 Java层Binder架构

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

要深入了解这一问题，有必要阅读卷I的第6章。

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

nativeInit () 方法的真正实现为android_os_MessageQueue_nativeInit () 函数，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::NativeMessageQueue() { /* 代表消息循环的Looper
也在Native层中呈现身影了。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) { /* 如果是第
一次进来，则该线程没有设置本地存储，所以需要先创建一个
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()方法。当消息队列为空时，next就会阻塞。MessageQueue同时支持Java层和Native层的事件，那么其next()方法该怎么实现呢？具体代码如下：

```
[MessageQueue.java-->MessageQueue.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用于给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 */ .....  
pendingIdleHandlerCount = 0; nextPollTimeoutMillis = 0; } } }
```

看到这里，可能会有人觉得这个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。请思考下面的情况：

在nativePollOnce () 返回后，next () 方法将从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 () 将在Native层等待，就表明Native层也可以投递Message，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。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 () 不仅在等待Java层来的Message，实际上在Native层还做了大量工作。

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：

```
[MesssageQueue.java-->MessageQueue.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加到消息尾 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剩余有消息，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。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(ptr); return nativeMessageQueue->wake(); // 调用它的wake函数 } [android_os_MessageQueue.cpp-->NativeMessageQueue::wake()] void NativeMessageQueue::wake() { mLooper->wake(); // 层层调用，现在转到mLooper的wake函数 }
```

Native Looper的wake函数代码如下：

```
[Looper.cpp-->Looper::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); }
```

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_nativePollOnce()]
static void android_os_MessageQueue_nativePollOnce(JNIEnv* env, jobject obj, jint ptr, jint timeoutMillis)
NativeMessageQueue* nativeMessageQueue = reinterpret_cast(ptr);
// 取出NativeMessageQueue对象，并调用它的
pollOnce nativeMessageQueue->pollOnce(timeoutMillis); }
```

分析pollOnce函数：

```
[android_os_MessageQueue.cpp-->NativeMessageQueue::pollOnce()]
void NativeMessageQueue::pollOnce(int timeoutMillis) {
    mLooper->pollOnce(timeoutMillis);
    // pollOnce()操作被传递到Looper的pollOnce函数 }
}
```

Looper的pollOnce函数如下：

```
[Looper.cpp-->Looper::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。
- 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。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基础知识介绍

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。

下面通过一个简单例子来分析epoll的工作流程。

```
/* ① 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); /* ② 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)和EPOLLHUP(句柄断)。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表示系统默认会将EPOLLERR加入事件集合中 */ listenEvent.events = EPOLLIN;// 指定该句柄的可读事件 // epoll_event中有一个联合体叫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; /* ③ EPOLL_CTL_ADD将监听fd和监听事件加入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) { /* ④ 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 { // ⑤ 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) { /* ⑥ 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  
下文信息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，不过作为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，都需要把感兴趣的事件复制到内核中，而epoll只在epoll_ctl进行加入的时候复制一次。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。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。

最后，关于pipe，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，可参见邓凡平的一篇博文《随笔之如何实现一个线程池》。网址为

[http://www.cnblogs.com/innost/archive/2011/11/24/2261454.html。](http://www.cnblogs.com/innost/archive/2011/11/24/2261454.html)

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-->Looper::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，实际
并没有做什么处理。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，可能会让人有些丈二和尚摸不着头脑。但是把pollInner () 函数分析完毕，大家就会明白很多。pollInner () 函数非常长，把用于调试和统计的代码去掉，结果如下：

```
[Looper.cpp-->Looper::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  
    // 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; } #ifdef 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。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 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 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：

- 首先需要计算一下真正需要等待的时间。
- 调用epoll_wait函数等待。
- epoll_wait函数返回，这时候可能有三种情况：
 - 发生错误，则跳转到Done处。
 - 超时，这时候也跳转到Done处。
 - epoll_wait监测到某些文件句柄上有事件发生。
 - 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：
 - 如果是管道读端有事件，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。

- 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。下面通过从Native的Activity找到的一个例子来分析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，这里为
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 () 代码如下所示：

```
[Looper.cpp-->Looper::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; } ..... } #ifdef  
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。

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

此函数如下：

```
[Looper.cpp-->Looper::pushResponse()] void Looper::pushResponse(int  
events, const Request& request) { Response response; response.events =  
events; response.request = request; //其实很简单，就是保存所发生的事情  
和对应的Request mResponses.push(response); //然后保存到mResponse数  
组 }
```

根据前面的知识可知，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。

下面来了解如何添加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4.0开始，Native层也支持sendMessage () 。

sendMessage () 的代码如下：

```
[Looper.cpp-->Looper::sendMessage()] void Looper::sendMessage(const sp<Handler> & 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 } [Looper.java-->Looper::sendMessageAtTime()] void Looper::sendMessageAtTime(nsecs_t uptime, const sp<Handler> & handler, const Message& message) { size_t i = 0; { AutoMutex _l(mLock); size_t messageCount = mMessageEnvelopes.size(); // 按时间排序，将消息插入正确的位置 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(); } }
```

2.3.4 MessageQueue总结

想不到，一个小小的MessageQueue竟然有如此多的内容。在后面分析Android输入系统时，会再次在Native层和MessageQueue碰面，这里仅是为后面的相会打下一定的基础。

现在将站在一个比具体代码更高的层次来认识一下MessageQueue及其伙伴。

1.消息处理的大家族合照

MessageQueue只是消息处理大家族的一员，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：

·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。

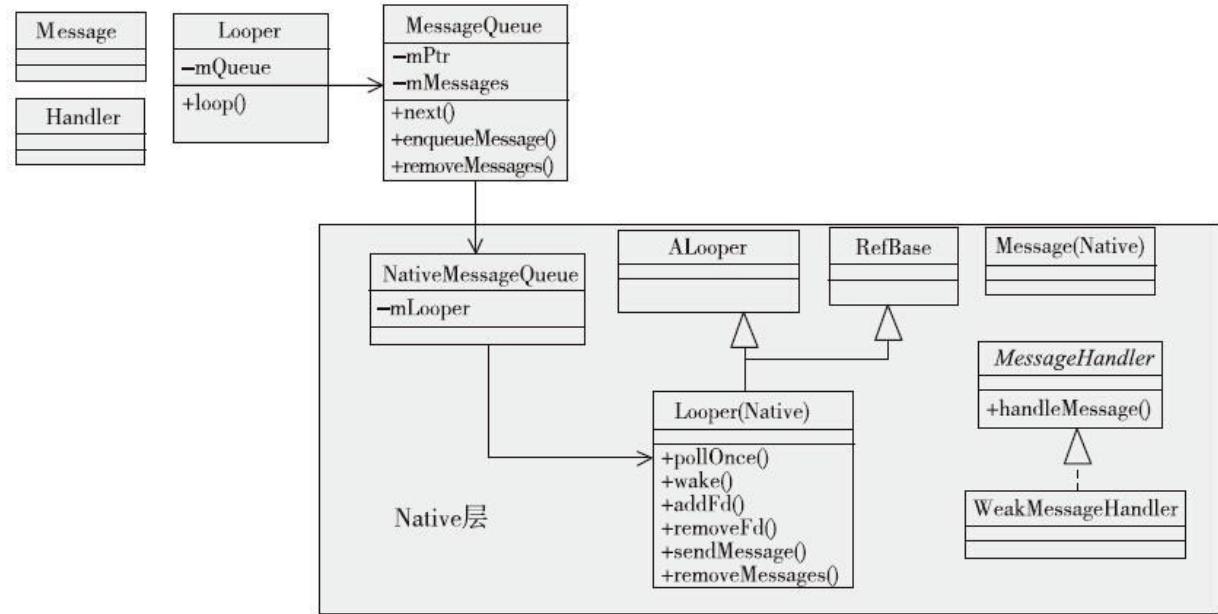


图2-5 消息处理的家族合照

- 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层的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MessageHandler抽象类。在编码时，一般使用的是MessageHandler的派生类WeakMessageHandler。



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团队的人编写的。

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，总结后有以下几点：

- 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层的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Native的Message，然后是Native的Request，最后才是Java的Message。

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的Binder在通信上还是依赖Native层的Binder。建议想进一步了解Native Binder工作原理的读者，阅读卷I第6章。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 Message派发以及处理来自所监控的文件句柄的事件。

本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供

最新最全的优质电子书下载！！！

第3章 深入理解AudioService

本章主要内容：

- 探讨AudioService如何进行音量管理
- 了解音频外设的管理机制
- 探讨AudioFocus的工作原理

本章涉及的源代码文件名及位置：

· AudioManager.java

framework/base/media/java/android/media/ AudioManager.java

· AudioService.java

framework/base/media/java/android/media/ AudioService.java

· AudioSystem.java

framework/base/media/java/android/media/ AudioSystem.java

· VolumePanel.java

framework/base/core/java/android/view/VolumePanel.java

·WiredAccessoryObserver.java

framework/base/services/java/com/android/server/WiredAccessoryObserver.java

·PhoneWindow.java

framework/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

·Activity.java

framework/base/core/java/android/app/Activity.java

3.1 概述

通过对卷I第7章的学习，相信大家已经对AudioTrack、AudioRecord、音频设备路由等知识有了深入了解。这一章将详细介绍音频系统在Java层的实现，围绕AudioService这个系统服务深入探讨在Android SDK中看到的音频相关的机制的实现。

在分析Android音频系统时，习惯将其实现分为两个部分：数据流和策略。数据流描述了音频数据从数据源流向目的地的过程。而策略则是管理及控制数据流的路径与呈现的过程。在卷I所探讨的Native层音频系统中，AudioTrack、AudioRecord和AudioFlinger可以划归到数据流的范畴去讨论，而AudioPolicy相关的内容则属于策略范畴。

音频系统在Java层中基本上是不参与数据流的。虽然有AudioTrack和AudioRecord这两个类，但是它们只是Native层同名类的Java封装。抛开这两个类，AudioService这个系统服务包含或使用了几乎所有与音频相关的内容，所以说AudioService是一个音频系统的大本营，它的功能非常多，而且它们之间的耦合性也不大。本章将从三个方面来探讨AudioService。

·音量管理。

从按下音量键到弹出音量调节提示框的过程，以及静音功能的工作原理。

·音频IO设备的管理。

我们将详细探讨从插入耳机到声音经由耳机发出这个过程中，
AudioService的工作内容。

·AudioFocus机制。

AudioService在Android 2.3及以后版本中提供了AudioFocus机制，用以结束多个音频应用混乱的交互现状。音频应用在播放音频的过程中需要合理地申请与释放AudioFocus，并且根据AudioFocus所有权的变化来调整自己的播放行为。我们将从音频应用开始播放音频，到播放完成的过程中探讨AudioFocus的作用及原理。

AudioService的类图如图3-1所示。

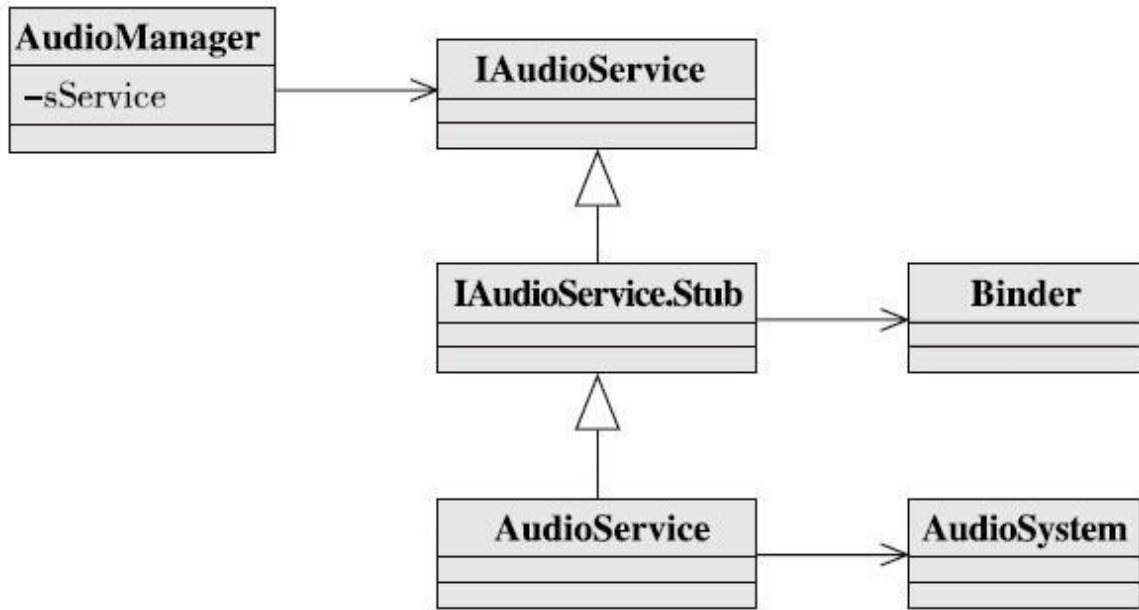


图3-1 AudioService的类图

由图3-1可知：

- AudioService继承自IAudioService.Stub。IAudioService.Stub类是通过IAudioService.aidl自动生成的。AudioService位于Bn端。
- AudioManager拥有AudioService的Bp端，是AudioService在客户端的一个代理。几乎所有客户端对AudioManager进行的请求，最终都会交由AudioService实现。
- AudioService的功能实现依赖AudioSystem类。AudioSystem无法实例化，它是Java层到native层的代理。AudioService将通过它与AudioPolicyService及AudioFlinger进行交互。

下面开始我们的AudioService之旅吧。

3.2 音量管理

Android手机有两种改变系统音量的方式。最直接的做法就是通过手机的音量键进行音量调整，还有一种做法是从设置界面中调整某一种类型音频的音量。另外，应用程序可以随时将某种类型的音频静音。它们都是通过AudioService进行的。

本节将从上述三个方面对AudioService的音量管理进行探讨。

3.2.1 音量键的处理流程

1. 触发音量键

在音量键被按下后，Android输入系统将该事件一路派发给Activity，如果无人截获并处理这个事件，承载当前Activity的显示PhoneWindow类的onKeyDown () 或onKeyUp () 函数将会处理，从而开始通过音量键调整音量的处理流程。输入事件的派发机制及PhoneWindow类的作用将在后续章节中详细介绍，现在只需要知道，PhoneWindow描述了一片显示区域，用于显示与管理我们所看到的Activity和对话框等内容。同时，它还是输入事件的派发对象，而且只有显示在最上面的PhoneWindow才会收到事件。



注意

按照Android的输入事件派发策略，Window对象在事件的派发队列中位于Activity的后面，所以应用程序可以重写自己的Activity.onKeyDown()函数以截获音量键的消息，将其用作其他的功能。比如说，在一个相机应用中，按下音量键所执行的动作是拍照而不是调节音量。

PhoneWindow的onKeyDown () 函数实现如下：

```
[PhoneWindow.java-->PhoneWindow.onKeyDown()] ..... switch (keyCode)
{ case KeyEvent.KEYCODE_VOLUME_UP: case
KeyEvent.KEYCODE_VOLUME_DOWN: case
KeyEvent.KEYCODE_VOLUME_MUTE: { // 直接调用到 AudioManager
的handleKeyUp里面去，是不是很简单而且直接呢
getAudioManager().handleKeyDown(event, mVolumeControlStreamType);
return true; } .....
```

注意handleKeyDown () 函数的第二个参数，它的意义是指定音量键将要改变哪一种流类型的音量。在Android中，音量的控制与流类型是密

不可分的，每种流类型都独立地拥有自己的音量设置，它们在绝大部分情况下互不干扰，例如音乐音量、通话音量就是相互独立的。所以说，离开流类型谈音量是没有意义的。在Android中，音量这个概念描述的一定是某一种流类型的音量。

这里传入了mVolumeControlStreamType，那么这个变量的值是从哪里来的呢？Activity类中有一个函数名为setVolumeControlStream（int streamType）。应用可以通过调用这个函数来指定显示这个Activity时音量键所控制的流类型。这个函数的内容很简单，就一行，如下：

[Activity.java-->Activity.setVolumeControlStream()]

```
getWindow().setVolumeControlStream(streamType);
```

getWindow（）的返回值就是用于显示当前Activity的PhoneWindow。从名字就可以看出，这个调用改变了mVolumeControlStreamType，于是也改变了按下音量键后传入AudioManager.handleKeyUp（）函数的参数，从而达到setVolumeControlStream的目的。同时，还应该能看出，这个设置被绑定到Activity的Window上，在不同Activity之间切换时，接收按键事件的Window也会随之切换，所以应用不需要去考虑在其生命周期中音量键所控制的流类型的切换问题。

AudioManager的handleKeyDown（）的实现很简单，在一个switch中，它调用了AudioService的adjustSuggestedStreamVolume（），所以直接

看一下AudioService的这个函数。

2.adjustSuggestedStreamVolume () 分析

我们先来看函数原型：

```
public void adjustSuggestedStreamVolume(int direction, int  
suggestedStreamType, int flags)
```

adjustSuggestedStreamVolume () 有三个参数，第一个参数direction指示了音量的调整方向，1为增大，-1为减小；第二个参数suggestedStreamType表示要求调整音量；第三个参数flags的意思就不那么容易理解了。其实AudioManager在handleKeyDown () 中设置了两个flag，分别是FLAG_SHOW_UI和FLAG_VIBRATE。从名字上我们就能看出一些端倪。前者告诉AudioService我们需要弹出一个音量控制面板。而在handleKeyUp () 里设置了FLAG_PLAY_SOUND，这是为什么当在松开音量键后“有时候”会有一个提示音。注意，handleKeyUp () 中设置了FLAG_PLAY_SOUND，但只是有时候这个flag才会生效，在下面的代码中可以看到这是为什么。还需要注意的是，第二个参数名为suggestedStreamType，从其命名来推断，这个参数传入的流类型对AudioService来说只是一个建议，是否采纳这个建议，AudioService有自己的考虑。

看一下它的实现：

[AudioService.java-->AudioService.adjustSuggestedStreamVolume()]

```
public void adjustSuggestedStreamVolume(int direction, int
suggestedStreamType, int flags) { int streamType; // ① 从这一小段代码中
可以看出，在 AudioService 中还有地方可以强行改变音量键控制的流类
型 if (mVolumeControlStream != -1) { streamType =
mVolumeControlStream; } else { // ② 通过 getActiveStreamType() 函数获取
要控制的流类型 // 这里根据建议的流类型与 AudioService 的实际情况，
返回一个值 streamType = getActiveStreamType(suggestedStreamType); }
// ③ 这个冗长条件判断的目的，就是只有在特定的流类型下，并且没有
处于锁屏状态时才会播放声音 if ((streamType !=
STREAM_REMOTE_MUSIC) && (flags&
AudioManager.FLAG_PLAY_SOUND) != 0 &&
((mStreamVolumeAlias[streamType] != AudioSystem.STREAM_RING) ||
(mKeyguardManager != null && mKeyguardManager.
isKeyguardLocked())))) { flags&= ~AudioManager.FLAG_PLAY_SOUND;
} if (streamType == STREAM_REMOTE_MUSIC) { ..... // 我们不讨论远
程播放的情况 } else { // ④ 调用 adjustStreamVolume
adjustStreamVolume(streamType, direction, flags); } }
```



注意

初看这段代码时，可能有读者对下面这句代码感到疑惑：

```
VolumeStreamState streamState =  
mStreamStates[mStreamVolumeAlias[streamType]];
```

其实，这是为了满足所谓的“将铃声音量用作通知音量”这种需求。这就需要实现在两个有这个需求的流A与B之间建立起一个A → B映射。当我们对A流进行音量操作时，实际上是在操作B流。笔者个人认为这个功能对用户体验的提升并不大，却给AudioService的实现增加了不小的复杂度。直观上来想，我们可以使用一个HashMap解决这个问题，键是源流类型，值是目标流类型。而Android使用了一个更简单却不是那么好理解的方法来完成这件事。AudioService用一个名为mStreamVolumeAlias的整型数组来描述这个映射关系。

要实现“以铃声音量用作音乐音量”，只需要修改相应位置的值为STREAM_RING即可，就像下面这样：

```
mStreamVolumeAlias[AudioSystem.STREAM_MUSIC] =  
AudioSystem.STREAM_RING;
```

之后，因为需要对A流进行音量操作时，实际上是在操作B流，所以就不难理解为什么在很多和流相关的函数里都会先做这样的一个转换：

```
streamType = mStreamVolumeAlias[streamType];
```

其具体的工作方式就留给读者思考。在本章的分析过程中，大可忽略这种转换，这并不影响我们对音量控制原理的理解。

简单来说，这个函数做了三件事：

- 确定要调整音量的流类型。
- 在某些情况下屏蔽FLAG_PLAY_SOUND。
- 调用adjustStreamVolume () 。

关于这个函数有几点仍需要说明一下。在函数刚开始的时候有一个判断，条件是一个名为mVolumeControlStream的整型变量是否等于-1，从这块代码来看，mVolumeControlStream比参数传入的suggestedStreamType厉害多了，只要它不是-1，要调整音量的流类型就是它。那这么厉害的控制手段的作用是什么？其实，mVolumeControlStream是VolumePanel通过forceVolumeControlStream

() 函数设置的。什么是VolumePanel呢？就是我们按下音量键后的那个音量调节通知框。VolumePanel在显示时会调用 forceVolumeControlStream 强制后续的音量键操作固定为促使它显示的那个流类型，并在它关闭时取消这个强制设置，即设置 mVolumeControlStream 为 -1。这个在后面分析 VolumePanel 时会看到。

接下来我们继续看一下 adjustStreamVolume () 的实现。

3.adjustStreamVolume () 分析

```
[AudioService.java-->AudioService.adjustStreamVolume()]
public void
adjustStreamVolume(int streamType, int direction, int flags) { // 首先还是获
取streamType映射到的流类型。这个映射的机制确实给我们的分析带来
不小的干扰 // 在非必要的情况下忽略它们吧
int streamTypeAlias =
mStreamVolumeAlias[streamType]; // 注意VolumeStreamState类
VolumeStreamState streamState = mStreamStates[streamTypeAlias];
final
int device = getDeviceForStream(streamTypeAlias); // 获取当前音量，注
意第二个参数的值，它的目的是如果这个流被静音，则取出它被静音
前的音量
final int aliasIndex = streamState.getIndex(device,
(streamState.muteCount() != 0)
boolean adjustVolume = true; // rescaleIndex
用于将音量值的变化量从源流类型变换到目标流类型下 // 由于不同的
流类型的音量调节范围不同，所以这个转换是必需的
int step =
rescaleIndex(10, streamType, streamTypeAlias); //上面准备好了所需的所
```

有信息，接下来要做一些真正有用的动作了 // 比如说
checkForRingerModeChange()。调用这个函数可能变更情景模式 // 它的
返回值adjustVolume是一个布尔变量，用来表示是否有必要继续设置音
量值 // 这是因为在一些情况下，音量键用来改变情景模式，而不是设
置音量值 if (((flags & AudioManager.FLAG_ALLOW_RINGER_MODES)
!= 0) || (streamTypeAlias == getMasterStreamType())) { adjustVolume =
checkForRingerModeChange(aliasIndex, direction, step); } int index; //
取出调整前的音量值。这个值稍后被用在sendVolumeUpdate()的调用中
final int oldIndex = mStreamStates[streamType].getIndex(device,
(mStreamStates[streamType].muteCount() != 0) /* lastAudible */); // 接下来
我们可以看到，只有流没有被静音时，才会设置音量到底层去，否则
只调整其静音前的音量 // 为了简单起见，暂不考虑静音时的情况 if
(streamState.muteCount() != 0) { } else { // 为什么还要判断
streamState.adjustIndex的返回值呢? // 因为如果音量值在调整 (adjust)
之后并没有发生变化，比如说达到了最大值，就不需要继续后面的操
作了 if (adjustVolume && streamState.adjustIndex(direction * step, device))
{ // 发送消息给AudioHandler // 这个消息在setStreamVolumeInt()函数的
分析中已经看到过 // 这个消息将把音量设置到底层去，并将其存储到
SettingsProvider中 sendMsg(mAudioHandler,
MSG_SET_DEVICE_VOLUME, SENDMSG_QUEUE, device, 0,
streamState, 0); } index = mStreamStates[streamType].getIndex(device,

```
false /* lastAudible */); } // 最后，调用sendVolumeUpdate函数，通知外界音量值发生了变化 sendVolumeUpdate(streamType, oldIndex, index, flags); }
```

在这个函数的实现中，有一个非常重要的类型：VolumeStreamState。前面提到过，Android的音量是依赖于某种流类型的。如果Android定义了N个流类型，AudioService就需要维护N个音量值与之对应。另外每个流类型的音量等级范围不一样，所以还需要为每个流类型维护它们的音量调节范围。VolumeStreamState类的功能就是为了保存与一个流类型所有音量相关的信息。AudioService为每一种流类型都分配了一个VolumeStreamState对象，并且以流类型的值为索引，保存在一个名为mStreamStates的数组中。在这个函数中调用了VolumeStreamState对象的adjustIndex () 函数，于是就改变了这个对象中存储的音量值。不过，仅仅是改变了它的存储值，并且没有把这个变化设置到底层。

总结一下这个函数都做了什么。

- 准备工作。计算按下音量键的音量步进值。细心的读者一定注意到了，这个步进值是10而不是1。原来，在VolumeStreamState中保存的音量值是其实际值的10倍。为什么这么做呢？这是为了在不同流类型之间进行音量转换时能够保证一定精度的一种实现，其转换过程读者可以参考rescaleIndex () 函数的实现。我们可以将这种做法理解为在转

换过程中保留了小数点后一位的精度。其实，直接使用float类型来保存岂不更简单？

- 检查是否需要改变情景模式。checkForRingerModeChange () 和情景模式有关。读者可以自行研究其实现。
- 调用adjustIndex () 更改VolumeStreamState对象中保存的音量值。
- 通过sendMsg () 发送消息MSG_SET_DEVICE_VOLUME到mAudioHandler。
- 调用sendVolumeUpdate () 函数，通知外界音量发生了变化。

我们将重点分析后面三项内容：adjustIndex () 、MSG_SET_DEVICE_VOLUME消息的处理和sendVolumeUpdate () 。

4. VolumeStreamState的adjustIndex () 分析

我们先看一下这个函数的定义：

```
[AudioService.java-->VolumeStreamState.adjustIndex()] public boolean  
adjustIndex(int deltaIndex, int device) { // 将现有的音量值加上变化量,  
然后调用setIndex进行设置 // 返回值与setIndex一样 return  
setIndex(getIndex(device, false /* lastAudible */) + deltaIndex, device, true  
/* lastAudible */); }
```

这个函数很简单，下面再看一下setIndex () 的实现：

```
[AudioService.java-->VolumeStreamState.setIndex()] public synchronized  
boolean setIndex(int index, int device, boolean lastAudible) { int oldIndex =  
getIndex(device, false /* lastAudible */); index = getValidIndex(index); // 在  
VolumeStreamState中保存设置的音量值，注意使用了一个HashMap  
mIndex.put(device, getValidIndex(index)); if (oldIndex != index) { // 保存到  
lastAudible if (lastAudible) { mLastAudibleIndex.put(device, index); } // 同  
时设置所有映射到当前流类型的其他流的音量 boolean currentDevice =  
(device == getDeviceForStream(mStreamType)); int numStreamTypes =  
AudioSystem.getNumStreamTypes(); for (int streamType =  
numStreamTypes - 1; streamType >= 0; streamType--) { ..... } return true; }  
else { return false; } }
```

在这个函数中有三项工作要做：

- 首先保存设置的音量值。这是VolumeStreamState的本职工作，这和Android 4.1之前的版本不一样，音量值与设备相关联了。因此对同一种流类型来说，在不同的音频设备下将会拥有不同的音量值。
- 然后根据参数的要求保存音量值到mLastAudibleIndex中。从名字就可以看出，它保存了静音前的音量。当取消静音时，AudioService就会恢
复到这里保存的音量。

·再就是对流映射的处理。既然A → B，那么在设置B的音量的同时要改变A的音量。这就是后面那个循环的作用。

可以看出，VolumeStreamState.adjustIndex () 除了更新自己所保存的音量值外，没有做其他的事情。接下来再看一下MSG_SET_DEVICE_VOLUME的消息处理做了什么。

5.MSG_SET_DEVICE_VOLUME消息的处理

adjustStreamVolume () 函数使用sendMsg () 函数发送MSG_SET_DEVICE_VOLUME消息给mAudioHandler，这个Handler运行在AudioService的主线程上。直接看一下在mAudio-Handler中负责处理MSG_SET_DEVICE_VOLUME消息的setDeviceVolume () 函数：

```
[AudioService.java-->AudioHandler.setIndex()]
private void setDeviceVolume(VolumeStreamState streamState, int device) { /* 调用
    VolumeStreamState的applyDeviceVolume。这个函数的内容很简单，就是在调用Audio-
    System.setStreamVolumeIndex()。到这里，音量就被设置到底层的AudioFlinger中 */
    streamState.applyDeviceVolume(device); // 和上面一样，需要处理流音量映射的情况。这段代码和上面setIndex的相关代码很像，不是吗
    int numStreamTypes =
        AudioSystem.getNumStreamTypes(); for (int streamType =
        numStreamTypes - 1; streamType >= 0; streamType--) { ..... } /* 发送消息
```

给mAudioHandler，其处理函数将会调用persistVolume()函数，这将会把音量的设置信息存储到SettingsProvider中。AudioService在初始化时，将会从SettingsProvider中将音量设置读取出来并进行设置 */

```
sendMsg(mAudioHandler, MSG_PERSIST_VOLUME,  
SENDMSG_QUEUE, PERSIST_CURRENT|PERSIST_LAST_AUDIBLE,  
device, streamState, PERSIST_DELAY); }
```



注意

sendMsg () 是一个异步操作，这就意味着，完成adjustIndex () 更新音量信息后adjustStreamVolume () 函数就返回了，但是音量并没有立刻被设置到底层。不过由于Handler处理多个消息的过程是串行的，这就隐含着一种风险：如果当Handler正在处理某一个消息时发生了阻塞，那么按下音量键，虽然调用adjustStreamVolume () 可以立刻返回，并且从界面上看或用getStreamVolume () 获取音量值都是没有问题的，但是手机发出声音时的音量大小并没有改变。

6.sendVolumeUpdate () 分析

接下来，分析一下sendVolumeUpdate () 函数，它用于通知外界音量发生了变化。

```
[AudioService.java-->AudioService.sendVolumeUpdate()] private void  
sendVolumeUpdate(int streamType, int oldIndex, int index, int flags) { /* 读者可能会感觉这句代码有点奇怪，mVoiceCapable是从SettingsProvider中取出来的一个常量。从某种意义上来说，它可以用来判断设备是否拥有通话功能。对没有通话能力的设备来说，RING流类型自然也就没有意义了。这句话应该算是一种从语义操作上进行的保护 */ if  
(!mVoiceCapable && (streamType == AudioSystem.STREAM_RING)) {  
    streamType = AudioSystem.STREAM_NOTIFICATION; } //  
  
mVolumePanel是一个VolumePanel类的实例，就是它显示了音量提示框  
mVolumePanel.postVolumeChanged(streamType, flags); /*发送广播。可以看到它们都有(x+5)/10的一个操作。为什么除以10可以理解，但是+5的意义是什么呢？原来是为了实现四舍五入 */ oldIndex = (oldIndex + 5) /  
10; index = (index + 5) / 10; Intent intent = new  
Intent(AudioManager.VOLUME_CHANGED_ACTION);  
intent.putExtra(AudioManager.EXTRA_VOLUME_STREAM_TYPE,  
streamType);  
intent.putExtra(AudioManager.EXTRA_VOLUME_STREAM_VALUE,  
index);
```

```
intent.putExtra(AudioManager.EXTRA_PREV_VOLUME_STREAM_VAL  
UE, oldIndex); mContext.sendBroadcast(intent); }
```

这个函数将音量的变化通过广播的形式通知给其他感兴趣的模块。同时，它还特别通知了mVolumePanel。mVolumePanel是VolumePanel类的一个实例。我们所看到的音量调节通知框就是它。

至此，从按下音量键开始的整个处理流程就完结了。在继续分析音量调节通知框的工作原理之前，先对之前的分析过程进行总结，参考图3-2的序列图。

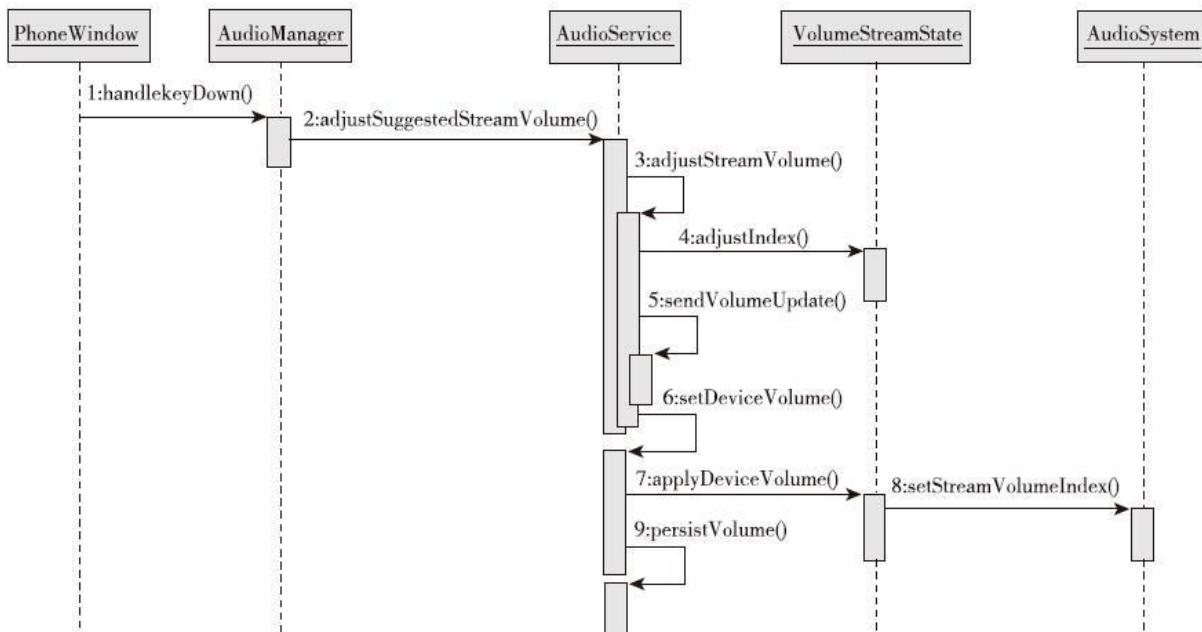


图3-2 通过音量键调整音量的处理流程

结合上面分析的结果，由图3-2可知：

- 音量键处理流程的发起者是PhoneWindow。
- AudioManager仅仅起到代理的作用。
- AudioService接受AudioManager的调用请求，操作VolumeStreamState的实例进行音量的设置。
- VolumeStreamState负责保存音量设置，并且提供了将音量设置到底层的方法。
- AudioService负责将设置结果以广播的形式通知外界。

到这里，相信大家对音量调节的流程已经有了一个比较清晰的认识。接下来我们将介绍音量调节通知框的工作原理。

7. 音量调节通知框的工作原理

在分析sendVolumeUpdate () 函数时曾经注意到，它调用了mVolumePanel的post-VolumeChanged () 函数。mVolumePanel是一个VolumePanel的实例，作为一个Handler的子类，它承接了音量变化的UI/声音的通知工作。在继续上面的讨论之前，先了解一下VolumePanel工作的基本原理。

VolumePanel位于android.view包下，却没有在API中提供，因为它只能被AudioService使用，所以和AudioService放在一个包下可能更合理一

些。从这个类的注释上可以看到，谷歌的开发人员对它被放在 android.view下也有极大不满（What A Mass！他们这么写道.....）。

VolumePanel下定义了两个重要的子类型，分别是StreamResources和 StreamControl。StreamResources实际上是一个枚举，它的每一个可用元素保存了一个流类型的通知框所需要的各种资源，如图标、提示文字等。StreamResources的定义就像下面这样：

```
[VolumePanel.java-->VolumePanel.StreamResources] private enum StreamResources {  
    BluetoothSCOStream(AudioManager.STREAM_BLUETOOTH_SCO,  
    R.string.volume_icon_description_bluetooth, R.drawable.ic_audio_bt,  
    R.drawable.ic_audio_bt, false), // 我们省略了后面的几个枚举项的构造参数，这些与BluetoothSCOStream的内容是一致的 RingerStream(...),  
    VoiceStream(...), AlarmStream(...), MediaStream(...),  
    NotificationStream(...), MasterStream(...), RemoteStream(...); int streamType; // 流类型 int descRes; // 描述信息 int iconRes; // 图标 int iconMuteRes; // 静音图标 boolean show; // 是否显示 // 构造函数 StreamResources(int streamType, int descRes, int iconRes, int iconMuteRes , boolean show) { ..... } };
```

这几个枚举项组成了一个名为STREAM的数组，如下：

```
[VolumePanel.java-->VolumePanel.STREAMS] private static final StreamResources[] STREAMS = { StreamResources.BluetoothSCOStream, StreamResources.RingerStream, StreamResources.VoiceStream, StreamResources.MediaStream, StreamResources.NotificationStream, StreamResources.AlarmStream, StreamResources.MasterStream, StreamResources.RemoteStream };
```

VolumePanel将从这个STREAMS数组中获取它所支持的流类型的相关资源。这么做是不是有点啰嗦呢？事实上，在这里使用枚举并没有什么特殊的意义，使用一个普通的Java类来定义StreamResources就已经足够了。

StreamControl类则保存了一个流类型的通知框所需要显示的控件，其定义如下：

```
[VolumePanel.java-->VolumePanel.StreamControl] private class StreamControl { int streamType; ViewGroup group; ImageView icon; SeekBar seekbarView; int iconRes; int iconMuteRes; }
```

很简单对不对？StreamControl实例中保存了音量调节通知框中所需的所有控件。关于这个类在VolumePanel的使用，我们可能很直观地认为只有一个StreamControl实例，在对话框显示时，使其保存的控件按需加载指定流类型的StreamResources实例中定义的资源。其实不然，出

于对运行效率的考虑，StreamControl实例也是每个流类型人手一份，和StreamResources实例形成一一对应的关系。所有的StreamControl实例被保存在一个以流类型的值为键的Hashtable中，名为mStreamControls。我们可以在StreamControl的初始化函数createSliders() 中一窥端倪。

```
[VolumePanel-->VolumePanel.createSliders()] private void createSliders() {  
..... // 遍历STREAM中所有的StreamResources实例 for (int i = 0; i <  
STREAMS.length; i++) { StreamResources streamRes = STREAMS[i]; int  
streamType = streamRes.streamType; ..... // 为streamType创建一个  
StreamControl StreamControl sc = new StreamControl(); // 这里将初始化sc  
的成员变量 ..... // 将初始化好的sc放入mStreamControls中  
mStreamControls.put(streamType, sc); } }
```

值得一提的是，这个初始化的工作并没有在构造函数中进行，而是在postVolumeChanged() 函数中处理的。

既然已经有了通知框所需要的资源和通知框的控件，接下来就要有一个对话框承载它们。没错，VolumePanel保存了一个名为mDialog的Dialog实例，这就是通知框的本身了。每当有新的音量变化到来时，mDialog的内容就会被替换为指定流类型对应的StreamControl中所保存的控件，并且根据音量变化情况设置其音量条的位置，最后调用

`mDialog.show ()` 显示出来。同时，发送一个延时消息 `MSG_TIMEOUT`，这条延时消息生效时，将会关闭提示框。

`StreamResource`、`StreamControl`与`mDialog`的关系就像图3-3所示的那样，`StreamControl`可以说是`mDialog`的配件，随需拆卸。

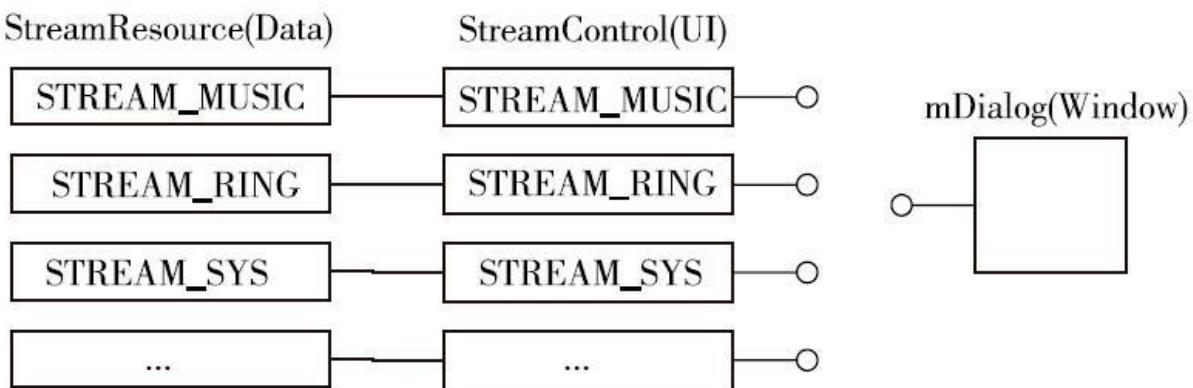


图3-3 StreamResource、 StreamControl与mDialog的关系

接下来具体看一下VolumePanel在收到音量变化通知后都做了什么。我们在上一小节中说到了`mVolumePanel.postVolumeChanged ()` 函数。它的内容很简单，直接发送了一条消息`MSG_VOLUME_CHANGED`，然后在`handleMessage`中调用`onVolumeChanged ()` 函数进行真正的处理。



注意

VolumePanel在MSG_VOLUME_CHANGED的消息处理函数中调用onVolumeChanged () 函数，而不是直接在postVolumeChanged () 函数中直接调用。这么做是有实际意义的。由于Android要求只能在创建控件的线程中对控件进行操作。postVolumeChanged () 作为一个回调性质的函数，不能要求调用者位于哪个线程中。所以必须通过向Handler发送消息的方式，将后续的操作转移到指定的线程中。在设计具有UI Controller功能的类时，VolumePanel的实现方式有很好的参考意义。

下面看一下onVolumeChanged () 函数的实现：

```
[VolumePanel.java-->VolumePanel.onVolumeChanged()] protected void  
onVolumeChanged(int streamType, int flags) { // 需要flags中包含  
AudioManager.FLAG_SHOW_UI 才会显示音量调节通知框 if ((flags &  
AudioManager.FLAG_SHOW_UI) != 0) { synchronized (this) { if  
(mActiveStreamType != streamType) { reorderSliders(streamType); // 在  
Dialog里装载需要的StreamControl } // 这个函数负责最终的显示  
onShowVolumeChanged(streamType, flags); } } // 是否播出Tone音，注意  
有个小延迟 if ((flags & AudioManager.FLAG_PLAY_SOUND) != 0 && !  
mRingIsSilent) { removeMessages(MSG_PLAY_SOUND);  
sendMessageDelayed(obtainMessage(MSG_PLAY_SOUND, streamType,
```

```
flags), PLAY_SOUND_DELAY); } // 取消声音与振动的播放 if ((flags &
AudioManager.FLAG_REMOVE_SOUND_AND_VIBRATE) != 0) {
removeMessages(MSG_PLAY_SOUND);
removeMessages(MSG_VIBRATE); onStopSounds(); } // 开始安排回收资
源 removeMessages(MSG_FREE_RESOURCES);
sendMessageDelayed(obtainMessage(MSG_FREE_RESOURCES),
FREE_DELAY); // 重置音量框超时关闭的时间 resetTimeout(); }
```

注意最后一个resetTimeout () 的调用，其实它重新延时发送了MSG_TIMEOUT消息。当MSG_TIMEOUT消息生效时，mDialog将被关闭。

之后就是onShowVolumeChanged了。这个函数负责为通知框的内容填充音量、图表等信息，然后再显示通知框（如果还没有显示）。以铃声音量为例，省略其他的代码。

[VolumePanel.java-->VolumePanel.onShowVolumeChanged()] protected void onShowVolumeChanged(int streamType, int flags) { // 获取音量值 int index = getStreamVolume(streamType); // 获取音量最大值，这两个将用来设置进度条 int max = getStreamMaxVolume(streamType); switch (streamType) { // 在这个switch语句中，我们要根据每种流类型的特点进
行各种调整。 // 例如Music有时就需要更新它的图标，因为使用蓝牙耳
机时的图标和平时的不一样， // 所以每一次都需要更新一下 case

```
AudioManager.STREAM_MUSIC: { if  
((mAudioManager.getDevicesForStream(AudioManager.STREAM_MUSIC  
) & (AudioManager.DEVICE_OUT_BLUETOOTH_A2DP |  
AudioManager.DEVICE_OUT_BLUETOOTH_A2DP_HEADPHONES |  
AudioManager.DEVICE_OUT_BLUETOOTH_A2DP_SPEAKER)) != 0) {  
setMusicIcon(R.drawable.ic_audio_bt, R.drawable.ic_audio_bt_mute); // 设  
置为蓝牙图标 } else { setMusicIcon(R.drawable.ic_audio_vol,  
R.drawable.ic_audio_vol_mute); // 设置为普通图标 } break; ..... } // 取出  
Music流类型对应的StreamControl，并设置其SeekBar的音量显示  
StreamControl sc = mStreamControls.get(streamType); if (sc != null) { if  
(sc.seekbarView.getMax() != max) { sc.seekbarView.setMax(max); }  
sc.seekbarView.setProgress(index); ..... } if (!mDialog.isShowing()) { // 如  
果对话框还没有显示 /*forceVolumeControlStream()的调用在这里，一旦  
此通知框被显示，之后按下音量键都只能 调节当前流类型的音量。直  
到通知框关闭时，重新调用forceVolumeControlStream(),并设 置  
streamType为-1*/  
mAudioManager.forceVolumeControlStream(streamType); // 为Dialog设置  
显示控件 /*注意, mView目前已经在reorderSlider()函数中安装好Music  
流所对应的StreamControl了 */ mDialog.setContentView(mView); ..... //  
显示对话框 mDialog.show(); } }
```

至此，音量调节通知框就被显示出来了，下面总结一下它的工作过程：

- postVolumeChanged () 是VolumePanel显示的入口。
- 检查flags中是否有FLAG_SHOW_UI。
- VolumePanel会在第一次被要求弹出时初始化其控件资源。
- mDialog加载指定流类型对应的StreamControl，也就是控件。
- 显示对话框并开始超时计时。
- 超时计时到达，关闭对话框。

到此为止，AudioService对音量键的处理流程介绍完毕。而Android还有另外一种改变音量的方式，即音量设置函数setStreamVolume ()，下面对其进行介绍。

3.2.2 通用的音量设置函数setStreamVolume ()

除了可以通过音量键调节音量以外，用户还可以在系统设置中进行调节。AudioManager.setStreamVolume () 是系统设置界面中调整音量所使用的接口。

1.setStreamVolume () 分析

`setStreamVolume()` 是SDK中提供给应用的API，它的作用是为特定的流类型设置范围内允许的任意音量。我们看一下它的实现：

```
[AudioService.java-->AudioService.setStreamVolume()]
public void setStreamVolume(int streamType, int index, int flags) {
    // 这里先判断一下流类型这个参数的有效性
    ensureValidStreamType(streamType);
    // 获取保存了指定流类型音量信息的VolumeStreamState对象
    // 注意，这里面使用mStreamVolumeAlias对这个数组进行流类型转换
    VolumeStreamState streamState = mStreamStates[mStreamVolumeAlias[streamType]];
    // 获取当前流将使用哪一个音频设备进行播放。它最终会被调用到
    AudioPolicyService中
    final int device = getDeviceForStream(streamType);
    // 获取流当前的音量
    final int oldIndex = streamState.getIndex(device,
        (streamState.muteCount() != 0) /* lastAudible */);
    // 将原流类型下的音量值映射到目标流类型下的音量值
    // 因为不同流类型的音量值刻度不一样，所以需要进行转换
    index = rescaleIndex(index * 10, streamType,
        mStreamVolumeAlias[streamType]);
    //暂时先忽略下面这段if中的代码。
    //它的作用是根据flags的要求修改手机的情景模式
    if (((flags & AudioManager.FLAG_ALLOW_RINGER_MODES) != 0) ||
        (mStreamVolumeAlias[streamType] == getMasterStreamType())))
        { ..... }
    // 调用setStreamVolumeInt()
    setStreamVolumeInt(mStreamVolumeAlias[streamType], index, device,
        false, true);
    // 获取设置的结果
    index =
```

```
mStreamStates[streamType].getIndex(device,  
(mStreamStates[streamType].muteCount() != 0) /* lastAudible */); // 广播通  
知 sendVolumeUpdate(streamType, oldIndex, index, flags); }
```

看明白这个函数了吗？抛开被忽略掉的那个if块可以归纳为：这个函数的工作其实很简单，就执行了下面三方面的工作：

- 为调用setStreamVolumeInt () 准备参数。
- 调用setStreamVolumeInt () 。
- 广播音量发生变化的通知。

下面分析的主线将转向setStreamVolumeInt () 的内容。

2.setStreamVolumeInt () 分析

看一下setStreamVolumeInt () 函数的代码，和前面一样，暂时忽略目前与分析目标无关的部分代码。

```
[AudioService.java-->AudioService.setStreamVolumeInt()] private void  
setStreamVolumeInt(int streamType, int index, int device, boolean force,  
boolean lastAudible) { // 获取保存音量信息的VolumeStreamState对象  
VolumeStreamState streamState = mStreamStates[streamType]; if  
(streamState.muteCount() != 0) { // 这里的内容是为了处理当流被静音后
```

```
的情况。我们在讨论静音的实现时再考虑这段代码 ..... } else { // 调用  
streamState.setIndex() if (streamState.setIndex(index, device, lastAudible) ||  
force) { // 如果setIndex返回true，或者force参数为true，就在这里向  
mAudioHandler发送消息 sendMsg(mAudioHandler,  
MSG_SET_DEVICE_VOLUME, SENDMSG_QUEUE, device, 0,  
streamState, 0); } } }
```

此函数有两个工作内容，一个是调用streamState.setIndex ()，另一个则是根据setIndex () 的返回值和force参数决定是否发送MSG_SET_DEVICE_VOLUME消息。这两项内容在3.2.1节中已经介绍过，在此不再赘述。

至此，setStreamVolume () 的分析完成。



注意

分析完setStreamVolume () 的工作流程后，读者是否觉得有些熟悉呢？如果我们用setStreamVolumeInt () 的代码替换setStreamVolume () 中对setStreamVolumeInt () 的调用，再和adjustStreamVolume ()

函数进行以下比较，就会发现它们的内容出奇得相似。Android在其他地方也有这样的情况。从这一点上来说，已经发展到4.1版本的Android源代码仍然不够精致。读者可以思考一下，有没有办法把这两个函数融合为一个函数呢？

到此，对于音量设置相关的内容就告一段落。接下来我们将讨论和音量相关的另一个重要的内容——静音。

3.2.3 静音控制

静音控制的情况与音量调节有很大的不同。因为每个应用都有可能进行静音操作，所以为了防止状态发生紊乱，就需要为静音操作进行计数，也就是说多次静音后需要多次取消静音。

不过，进行了静音计数后还会引入另外一个问题。如果一个应用在静音操作（计数加1）后因为某种原因不小心崩溃了，那么将不会有人再为它进行取消静音的操作，静音计数无法再回到0，也就是说这个“倒霉”的流将被永远静音下去。

那么怎么处理应用异常退出后的静音计数呢？AudioService的解决办法是记录下每个应用自己的静音计数，当应用崩溃时，在总的静音计数中减去崩溃应用自己的静音计数，也就是说，为这个应用完成它没能完成的取消静音这个操作。为此，VolumeStreamState定义了一个继承自DeathRecipient的内部类，名为VolumeDeathHandler，并且为每个进

行静音操作的进程创建一个实例。VolumeDeathHandler的实例保存了对应进程的静音计数，并在进程死亡时进行计数清零的操作。从这个名字来看可能是Google希望这个类将来能够承担更多与音量相关的事情吧，不过眼下它只负责静音。我们将在后续的内容中对这个类进行深入讲解。

经过前面的介绍，我们不难得出AudioService、VolumeStreamState与VolumeDeathHandler的关系，如图3-4所示。

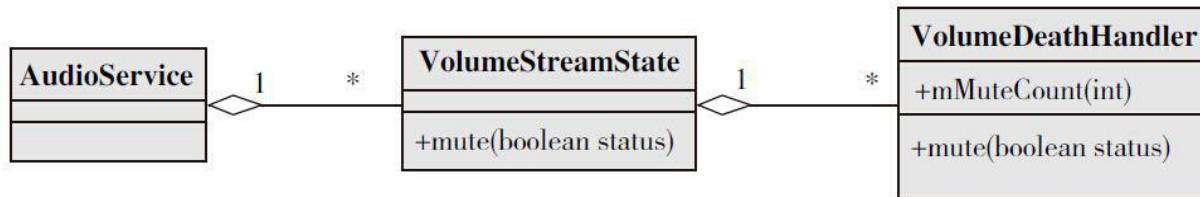


图3-4 与静音相关的类

1.setStreamMute () 分析

同音量设置一样，静音控制也是相对于某一个流类型而言的。正如本节开头所提到的，静音控制涉及引用计数和客户端进程的死亡监控。所以相对于音量控制来说，静音控制有一定的复杂度。还好，静音控制对外入口只有一个函数，就是`AudioManager.setStreamMute ()`。其第二个参数`state`为`true`，表示静音，否则表示解除静音。

```
[AudioManager.java-->AudioManager.setStreamMute()] public void  
setStreamMute(int streamType, boolean state) { IAudioService service =  
getService(); try { // 调用AudioService的setStreamMute，注意第三个参数  
mICallBack service.setStreamMute(streamType, state, mICallBack); } catch  
(RemoteException e) { Log.e(TAG, "Dead object in setStreamMute", e); } }
```

AudioManager一如既往地充当着AudioService代理的一个角色，不过这次有一个很小却很重要的动作：AudioManager为AudioService传入了一个名为mICallBack的变量。查看一下mICallBack的定义：

```
private final IBinder mICallBack = new Binder();
```

真是简单得不得了。全文搜索一下，我们发现mICallBack只用来作为AudioService的几个函数调用的参数。从AudioManager角度看它没有任何实际意义。其实，这在Android的进程间交互通信中是一种常见且非常重要的技术。mICallBack这个简单Binder对象可以充当Bp端在Bn端的一个唯一标识。而且AudioService拿到这个标识后，就可以通过DeathRecipient机制获取Bp端异常退出的回调。这是AudioService维持静音状态正常变迁的一个基石。



注意

服务端把客户端传入的这个Binder对象作为客户端的一个唯一标识的时候，往往会以这个标识为键创建一个Hashtable，用来保存每个客户端的相关信息。这在Android各个系统服务的实现中是一种很常见的用法。

另外，本例传入的mICallBack是直接从Binder类实例化出来的，是一个很原始的IBinder对象。进一步讲，如果传递了一个通过AIDL定义的IBinder对象，那么这个对象就有了交互能力，服务端可以通过它向客户端进行回调。在后面探讨AudioFocus机制时会遇到这种情况。

2. VolumeDeathHandler分析

我们继续跟踪AudioService.setStreamMute () 的实现，记得注意第三个参数cb，它代表特定客户端的标识。

```
[AudioService.java-->AudioService.setStreamMute()] public void  
setStreamMute(int streamType, boolean state, IBinder cb) { // 只有可以静
```

音的流类型才能执行静音操作。这说明，并不是所有的流都可以被静音

```
if (isStreamAffectedByMute(streamType)) { // 直接调用了流类型对应的mStreamStates的mute()函数 // 这里没有进行那个令人讨厌的流类型的映射。这是出于操作语义上的原因。读者可以自行思考一下mStreamStates[streamType].mute(cb, state); } }
```

接下来是VolumeStreamState的mute () 函数。VolumeStreamState的确是音量相关操作的核心类型。

```
[AudioService.java-->VolumeStreamState.mute()] public synchronized void mute(IBinder cb, boolean state) { // 这句话是一个重点，VolumeDeathHandler与cb一一对应 // 用来管理客户端的静音操作，并且监控客户端的生命状态 VolumeDeathHandler handler = getDeathHandler(cb, state); if (handler == null) { Log.e(TAG, "Could not get client death handler for stream:"+mStreamType); return; } // 通过VolumeDeathHandler执行静音操作 handler.mute(state); }
```

上述代码引入了静音控制的主角，VolumeDeathHandler，也许叫做MuteHandler更合适一些。它其实只有两个成员变量，分别是mICallBack和mMuteCount。其中mICallBack保存了客户端传进来的标识， mMuteCount则保存了当前客户端执行静音操作的引用计数。另外，它继承自IBinder.DeathRecipient，所以它拥有监听客户端生命状态的能力。而VolumeDeathHandler () 的成员函数只有两个，分别是mute

() 和binderDied () 。说到这里，再看看上面VolumeStreamState.mute () 的实现，读者能想象到VolumeDeathHandler的具体实现是什么样子的吗？

继续上面的脚步，看一下它的mute () 函数。它的参数state的取值指定了进行静音还是取消静音。所以这个函数也就被分成两部分，分别是处理静音与取消静音两个操作。其实，这完全可以放在两个函数中完成。先看看静音操作是怎么实现的吧。

```
[AudioService.java-->VolumeDeathHandler.mute()part 1] public void  
mute(boolean state) { if (state) { // 静音操作 if (mMuteCount == 0) { // 如  
果mMuteCount 等于0，则表示客户端是第一次执行静音操作 // 此时  
linkToDeath开始对客户端的生命状况进行监听 // 这样做的好处是可以  
避免非静音状态下额外占用Binder资源 try { // 为什么要判断linkToDeath  
是否为空？ AudioManager不是传递进一个有效的 // Binder吗？原来  
AudioManager也可能会调用mute() // 此时的mICallback为空 if  
(mICallback != null) { mICallback.linkToDeath(this, 0); } // 保存到  
mDeathHandlers列表中 mDeathHandlers.add(this); // muteCount()是对全  
局的静音操作的引用计数 // 如果它的返回值为0，则表示这个流目前还  
没有被静音 if (muteCount() == 0) { // 在这里设置流的音量为0 ..... } }  
catch (RemoteException e) { ..... } } // 引用计数加1 mMuteCount++; }  
else { // 暂时先不给出取消静音的操作 ..... } }
```

看明白了吗？这个函数的条件嵌套比较多，仔细归纳一下，就会发现这段代码的思路是非常清晰的。静音操作根据条件满足与否，完成三个任务：

- 无论在什么条件下，只要执行这个函数，静音操作的引用计数都会加1。
- 如果这是客户端第一次执行静音，则开始监控其生命状态，并且把自己加入VolumeStreamState的mDeathHandlers列表中。这是这段代码中很精练的一个操作，只有在客户端执行过静音操作后才会对其生命状态感兴趣，才有保存其VolumeDeathHandler的必要。
- 更进一步的是，如果这是这个流类型第一次被静音，则设置流音量为0，这才是真正的静音动作。

不得不说，这段代码是非常精练的，不是说代码量少，而是它的行为非常干净，决不会做多余的操作，也不会保存多余的变量。

下面我们要看一下取消静音的操作。取消静音作为静音的逆操作，相信读者已经可以想象到它都做什么事情了吧？这里就不再对其进行说明了。

```
[AudioService.java-->VolumeDeathHandler.mute() part 2] public void  
mute(boolean state) { if (state) { // 忽略掉静音操作 ..... } else { if
```

```
(mMuteCount == 0) { Log.e(TAG, "unexpected unmute for stream:  
"+mStreamType); } else { // 引用计数先减1 mMuteCount--; if  
(mMuteCount == 0) { // 如果这是客户端最后一次有效地取消静音  
mDeathHandlers.remove(this); if (mICallback != null) {  
mICallback.unlinkToDeath(this, 0); } if (muteCount() == 0) { // 将流的音量  
值设置回静音前的音量，也就是lastAudibleIndex ..... } } } }
```

下面就剩下最后的binderDied () 函数了。当客户端发生异常，没能取消其执行过的静音操作时，需要替它完成它应该做却没做的事情。

```
[AudioService.java-->VolumeDeathHandler.binderDied()] public void  
binderDied() { if (mMuteCount != 0) { mMuteCount = 1; mute(false); } }
```

这个实现不难理解，读者可以自行分析一下为什么这么做可以消除意外退出的客户端遗留下来的影响。

3.2.4 音量控制小结

音量控制是AudioService最重要的功能之一。经过上面的讨论，相信读者对AudioService的音量管理流程已经有了一定的理解。

总结一下我们在这一节里所学到的内容：

- AudioService音量管理的核心是VolumeStreamState。它保存了一个流类型所有的音量信息。

·VolumeStreamState保存了运行时的音量信息，而音量的生效则是在底层AudioFlinger完成的。所以进行音量设置需要做两件事情：更新VolumeStreamState存储的音量值，设置音量到Audio底层系统。

·VolumeDeathHandler是VolumeStreamState的一个内部类。它的实例对应在一个流类型上执行静音操作的一个客户端，是实现静音功能的核心对象。

3.3 音频外设的管理

这一节将探讨AudioService的另一个重要功能，那就是音频外设的管理。看过卷I第7章的读者应该对音频外设这个概念并不陌生。在智能机全面普及的时代，对有线耳机、蓝牙耳机等音频外设的支持已经是手机的标准，有些机型甚至支持HDMI、USB声卡等输出接口。再加上手机本身自带的扬声器与听筒，这样一来，一台手机上同时能进行音频输出的设备往往会有三四种甚至更多。如何协调这些设备的工作，使其符合用户的使用习惯、满足用户的需求变得非常重要。

卷I的第7章详细介绍过AudioPolicy如何进行设备的路由切换，然而并没有讨论音频设备为什么出现在AudioPolicy的设备候选列表中，这一节将以有线耳机为例讨论这个问题。

3.3.1 WiredAccessoryObserver设备状态的监控

1. WiredAccessoryObserver简介

这要从WiredAccessoryObserver开始讲起，它是内核通知有线耳机插入事件所到达的第一个环节。

WiredAccessoryObserver继承自UEventObserver。UEventObserver是Android用来接收UEvent的一个工具类。UEventObserver类维护着一个

读取UEvent的线程，注意这个线程是UEventObserver的一个静态成员，也就是说，一个进程只有一个。当调用UEventObserver的startObserving()函数开始监听时，会告诉这个线程UEventObserver关心什么样的UEvent，当匹配的事件到来时，监听线程会通过回调UEventObserver的onUEvent函数进行通知。读者可以看一下UEventObserver的源代码以了解其具体实现，这并不复杂。

WiredAccessoryObserver接收内核上报的和耳机/HDMI/USB相关的UEvent事件，并将其翻译成设备的状态变化。由于每种外设都有自己的UEvent与状态文件，因此WiredAccessoryObserver定义了一个内部类名为UEventInfo，并且为自己感兴趣的每一个音频外设创建一个实例，其内部保存了对应外设的名字、UEvent地址及状态文件的地址。每当有合适的UEvent到来时，WiredAccessoryObserver就会查找匹配的UEventInfo实例，并且更新可用设备的状态列表，同时通知AudioService。

关于可用外设的状态列表，虽然称为列表，事实上，它只是一个整型的变量，名为mHeadsetState。在可用外设的状态列表中用一个二进制标志位表示某个外设的状态可用与否，这与AudioPolicyManager的mAvailableOutputDevices的用法是一样的。下面是各种外设的标志位的定义：

```
private static final int BIT_HEADSET = (1 << 0); private static final int  
BIT_HEADSET_NO_MIC = (1 << 1); private static final int  
BIT_USB_HEADSET_ANLG = (1 << 2); private static final int  
BIT_USB_HEADSET_DGTL = (1 << 3); private static final int  
BIT_HDMI_AUDIO = (1 << 4); private static final int  
SUPPORTED_HEADSETS = (BIT_HEADSET|BIT_HEADSET_NO_MIC|  
BIT_USB_HEADSET_ANLG|BIT_USB_HEADSET_DGTL|  
BIT_HDMI_AUDIO); private static final int HEADSETS_WITH_MIC =  
BIT_HEADSET;
```

举个例子，如果mHeadsetState等于0x00000002，也就是
BIT_HEADSET_NO_MIC，表示目前手机上插入一个不带麦克风的耳
机。而如果mHeadsetState等于0x00000011，也就是
HEADSETS_WITH_MIC|BIT_HDMI_AUDIO，则表示目前手机上同时
插入一个带有麦克风的耳机及HDMI输出线。

WiredAccessoryObserver工作原理就这么简单，我们接下来将以有线耳
机为例子对其进行详细讨论。

2.启动与初始化

虽然WiredAccessoryObserver不是一个服务，但是它拥有系统服务的待
遇——在system_server中同系统服务一起被加载，如下所示：

```
[SystemServer.java-->ServerThread.run()] try { new  
WiredAccessoryObserver(context); } catch (Throwable e) {  
reportWtf("starting WiredAccessoryObserver", e); }
```

只有一个构造函数，其实，构造函数中并没有做太多的初始化工作，而是注册了一个BroadcastReceiver，监听ACTION_BOOT_COMPLETE。其真正的初始化工作是在这个BootCompletedReceiver中完成的。

```
[WiredAccessoryObserver.java-->BootCompletedReceiver.onReceive()]  
public void onReceive(Context context, Intent intent) { // 初始化 init(); // 开始对所有感兴趣的UEvent进行监听 for (int i = 0; i < uEventInfo.size();  
++i) { UEventInfo uei = uEventInfo.get(i);  
startObserving("DEVPATH="+uei.getDevPath()); } }
```

这里的init () 函数的作用是为了在开机后对外设的状态进行初始化。

```
[WiredAccessoryObserver.java-->WiredAccessoryObserver.init()] private  
synchronized final void init() { char[] buffer = new char[1024];  
mPrevHeadsetState = mHeadsetState; for (int i = 0; i < uEventInfo.size();  
++i) { UEventInfo uei = uEventInfo.get(i); try { int curState; /* 打开状态文件并从中读取状态信息。状态文件中保存着一个整数，非0则表示设备已插入。通过UEventInfo的定义可以知道，有线耳机的状态文件路径
```

```
为 /sys/class/switch/h2w/state */ FileReader file = new  
FileReader(uei.getSwitchStatePath()); int len = file.read(buffer, 0, 1024);  
file.close(); curState = Integer.valueOf((new String(buffer, 0, len)).trim()); //  
如果设备已插入，则更新设备的状态，否则不作处理 if (curState > 0) {  
updateState(uei.getDevPath(), uei.getDevName(), curState); } } catch  
(Exception e) { ..... } }
```

到这里WiredAccessoryObserver已经完成初始化了，已经对第一条UEvent的到来准备就绪。

3. 耳机插入或拔出时的处理

如果有外设被插入或拔出，WiredAccessoryObserver的onUEvent () 函数会被回调。参数event中保存了其详细的信息。

[WiredAccessoryObserver.java-->WiredAccessoryObserver.onUEvent()]

```
public void onUEvent(UEventObserver.UEvent event) { try { // UEvent事件  
的路径 String devPath = event.get("DEVPATH"); // 这个name其实就是  
UEventInfo中的mDevName，通过这个变量确定发生状态变化的设备名  
字 String name = event.get("SWITCH_NAME"); // 这个state与保存在状态  
文件中的数值的意义是一致的 // 事实上，当这条UEvent上报时，状态  
文件中的值也被更新成这个值 int state =  
Integer.parseInt(event.get("SWITCH_STATE")); // 像初始化的init()函数一
```

```
样，调用updateState()进行状态更新 updateState(devPath, name, state); }  
catch (NumberFormatException e) { ..... } }  
[WiredAccessoryObserver.java-->WiredAccessoryObserver.updateState()]  
private synchronized final void updateState(String devPath, String name, int  
state) { for (int i = 0; i < uEventInfo.size(); ++i) { UEventInfo uei =  
uEventInfo.get(i); if (devPath.equals(uei.getDevPath())) { // 找到状态发生  
变化的外设所对应的UEventInfo并更新状态 update(name,  
uei.computeNewHeadsetState(mHeadsetState, state)); return; } } }
```

看到这里，读者是否觉得updateState的实现有些笨拙了呢？如果以devName为键，将uEventInfo保存在Hashtable中，无论对代码的整洁还是执行的效率都是有帮助的。

注意uei.computeNewHeadsetState () 这个函数，它的目的是通过UEvent上报的状态值计算出新的可用外设列表。



注意

computeNewHeadsetState () 这个函数的扩展性并不是太好，只是目前够用而已，读者可以自行研究。

继续前面的脚步，现在到了update () 函数。这个函数的目的是对前面传入的新State进行全面检查，防止出现不正确的状态。这个函数的运算稍多些，为了方便分析，仅留下和有线耳机（h2w）相关的代码。

```
[WiredAccessoryObserver.java-->WiredAccessoryObserver.update()]  
private synchronized final void update(String newName, int newState) { /*  
从headsetState中去掉不支持的外设，所以，如果不希望手机支持某种  
外设，比如说USB_HEADSET，不需要从kernel改起，只要将其从  
SUPPORTED_HEADSETS中去 掉即可 */ int headsetState = newState &  
SUPPORTED_HEADSETS; int h2w_headset = headsetState &  
(BIT_HEADSET | BIT_HEADSET_NO_MIC); boolean h2wStateChange =  
true; // 下面这行代码比较有意思，首先我们的目的是判断有线耳机的  
状态是否发生了变化 // mHeadsetState == headsetState这种条件很好理  
解，可是后面那个条件呢 if (mHeadsetState == headsetState ||  
(h2w_headset & (h2w_headset - 1)) != 0)) { h2wStateChange = false; } //  
如果是不正确的状态转换则直接忽略 if (!h2wStateChange) { return; } //  
更新可用外设列表 mHeadsetName = newName; mPrevHeadsetState =  
mHeadsetState; mHeadsetState = headsetState; // 为什么要申请一个电源  
锁呢 mWakeLock.acquire(); // 状态已经更新完毕，发送消息给
```

```
mHandler, 我们可以想象出接下来要做什么了, 通知AudioSevicervice  
// 注意mHandler的定义, 可以看出它运行在创建  
WiredAccessoryObserver的ServerThread中  
mHandler.sendMessage(mHandler.obtainMessage(0, mHeadsetState,  
mPrevHeadsetState, mHeadsetName)); }
```



注意

这个函数的意图比较很明显，只是其中一个判断条件让人一时摸不着头脑， $(h2w_headset \& (h2w_headset - 1)) != 0$ 。按照注释中的说法，此函数不接受同时有两种耳机出现的情况，也就是`h2w_headst==BIT_HEADSET|BIT_HEADSET_NO_MIC`，直接做这个判断不就可以了？仔细琢磨就能发现写这个条件的人的聪明之处。直接判断仅限于只有两种可能的外设时才能起作用，超过两个就很难处理了。而谷歌的这个做法既快捷，又可以应对任意多种可能的外设。读者可以思考一下为什么。

另外，这段代码在执行mHandler.sendMessage () 的调用之前先申请了一个电源锁。这是一个很细节但很重要的做法。当发送消息给一个Handler时，必须考虑设备有可能在Handler得以处理消息之前进入深睡眠状态的极端情况（对延时消息来说，可能就是常见情况了）。在这种情况下，CPU将会进入休眠状态，从而使得消息无法得到及时处理，影响程序执行的正确性。

可用外设列表更新完毕后发送了一条消息给mHandler。当消息生效时，直接调用setDevicesState () 函数，它会遍历所有SUPPORTED_HEADSET，然后对每个外设调用setDeviceState () 。注意，这两个函数是devices与device的区别。setDeviceState () 的目的就是要把指定外设的状态汇报给AudioService，我们看一下它的实现：

[WiredAccessoryObserver.java--

```
>WiredAccessoryObserver.setDeviceState()]\ private final void  
setDeviceState(int headset, int headsetState, int prevHeadsetState, String  
headsetName) { if ((headsetState & headset) != (prevHeadsetState &  
headset)) { // 只有当这个外设的接入状态发生变化时才会继续 int  
device; int state; // 1表示可用， 0 表示不可用 if ((headsetState & headset)  
!= 0) { state = 1; } else { state = 0; } // 翻译可用外设列表中的外设为  
Audio系统的设备号 if (headset == BIT_HEADSET) { device =  
AudioManager.DEVICE_OUT_WIRED_HEADSET; } else if (headset ==
```

```
BIT_HEADSET_NO_MIC){ device =
    AudioManager.DEVICE_OUT_WIRED_HEADPHONE; } else if (headset
== BIT_USB_HEADSET_ANLG) { device =
    AudioManager.DEVICE_OUT_ANLG_DOCK_HEADSET; } else if
(headset == BIT_USB_HEADSET_DGTL) { device =
    AudioManager.DEVICE_OUT_DGTL_DOCK_HEADSET; } else if
(headset == BIT_HDMI_AUDIO) { device =
    AudioManager.DEVICE_OUT_AUX_DIGITAL; } else { Slog.e(TAG,
"setDeviceState() invalid headset type: "+headset); return; } // 通知
AudioService mAudioManager.setWiredDeviceConnectionState(device,
state, headsetName); } }
```

之后，程序的流程将会离开WiredHeadsetObserver，再次前往
AudioService。

4.总结一下WiredAccessoryObserver

对WiredHeadsetObserver的分析就先告一段落，这里再简单回顾一下关于它的知识。

- 它是站在最前方的一个哨兵，时刻监听着和音频外设拔插相关的UEvent事件。
- 它接收到UEvent事件后，会翻译事件的内容为外设可用状态的变化。

- 它是为AudioService服务的，一旦有变化就立刻通知AudioService。
- 它虽然不是一个服务，但是它却运行在system_server中。
- 它不是唯一的音频外设状态监听者，它只负责监控有线连接的音频外设。其他的，如蓝牙耳机，在其他相关模块中维护。但是它们的本质是类似的，最终都要通知给AudioService。有兴趣的读者可以自行研究。

3.3.2 AudioService的外设状态管理

最终还是要回到AudioService中来，它才是音频相关操作的主基地。

1.处理来自WiredAccessoryObserver的通知

AudioService会如何处理外设的可用状态变化呢？仔细想想，在开发播放器的时候一定接触过ACTION_AUDIO_BECOMING_NOISY和ACTION_HEADSET_PLUG这两个广播吧。另外，更重要的是，这些变化需要让底层的AudioPolicy知道。所以，笔者认为AudioService外设状态管理分为三个内容：

- 管理发送ACTION_AUDIO_BECOMING_NOISY广播。
- 发送设备状态变化的广播，通知应用。
- 将其变化通知底层。

从WiredHeadsetObserver调用的setWiredDeviceConnectionState () 函数开始：

[AudioService.java-->AudioService.setWiredDeviceConnectionState()]

```
public void setWiredDeviceConnectionState(int device, int state, String name) { synchronized (mConnectedDevices) { // 发送 ACTION_AUDIO_BECOMING_NOISY广播的地方 int delay = checkSendBecomingNoisyIntent(device, state); // 又是发送消息给 mAudioHandler, 注意这个消息有可能是延时的 // 这取决于 checkSendBecomingNoisyIntent的返回值 : delay queueMsgUnderWakeLock(mAudioHandler, MSG_SET_WIRED_DEVICE_CONNECTION_STATE, device, state, name, delay); } }
```

此函数负责两项工作：调用checkSendBecomingNoisyIntent () 函数及发送SET_WIRED_DEVICE_CONNECTION_STATE消息给mAudioHandler。

checkSendBecomingNoisyIntent () 函数的目的是判断当前状态的变化是否有必要发送BECOMING_NOISY广播。这个广播用于警告所有媒体播放应用声音即将从手机外放中进行播放。在绝大部分情况下，收到这个广播的应用都应当立即暂停播放，以避免用户无意识地泄露自己的隐私或打扰到周围的其他人。另外，这个函数的返回值决定了

SET_WIRED_DEVICE_CONNECTION_STATE消息是否需要延时处理。其代码如下：

```
[AudioService.java-->AudioService.checkSendBecomingNoisyIntent()]  
private int checkSendBecomingNoisyIntent(int device, int state) { int delay  
= 0; // 发送BECOMING_NOISY广播的前两个条件如下： // 1.外设被拔  
除 // 2.外设是mBecomingNoisyIntentDevices指定的外设之一 // 既然这些  
设备从手机拔除后会AUDIO_BECOMING_NOISY，不妨称它们为安静  
外设 if ((state == 0) && ((device & mBecomingNoisyIntentDevices) != 0))  
{ int devices = 0; // 收集所有连接在手机上的安静外设 for (int dev :  
mConnectedDevices.keySet()) { if ((dev & mBecomingNoisyIntentDevices)  
!= 0) { devices |= dev; } } // 发送 BECOMING_NOISY广播的第三个条  
件：移除的设备必须是连接在手机上的最后一个安静外设 // 同时也是  
推迟后续处理的第一个条件：发送了BECOMING_NOISY广播 if  
(devices == device) { delay = 1000; // 确定后续对这个状态变化的处理向  
后推迟1秒 sendBecomingNoisyIntent(); // 发送BECOMING_NOISY广播  
} } // 推迟后续处理的另外一个条件：如果有和外设连接状态相关的延  
迟消息尚未被处理，那么 // 也必须推后消息的处理 if  
(mAUDIOHandler.hasMessages(MSG_SET_A2DP_CONNECTION_STATE)  
||  
mAUDIOHandler.hasMessages(MSG_SET_WIRED_DEVICE_CONNECTI  
ON_STATE)) { delay = 1000; } return delay; }
```

代码不长，有价值的内容不少。BECOMING_NOISY广播发出的条件是最后一个安静外设被拔出，这个很好理解。而推迟MSG_SET_WIRED_DEVICE_CONNECTION_STATE消息的生效时间这种做法可能一时难以弄明白。不过暂时先不管它，等我们了解了外设连接状态变化的流程后再解释它的意义。

回到setWiredDeviceConnectionState () ，调用checkSendBecomingNoisyIntent () 函数后，它发送MSG_SET_WIRED_DEVICE_CONNECTION_STATE给mAudioHandler，此消息生效后，mAudioHandler调用onSetWiredDeviceConnectionState函数。

[AudioService.java-->AudioHandler.onSetWiredDeviceConnectionState()]
private void onSetWiredDeviceConnectionState(int device, int state, String name) { synchronized (mConnectedDevices) { // 如果拔下普通耳机，则会强制要求使用蓝牙耳机作为输出设备 if ((state == 0) && ((device == AudioSystem.DEVICE_OUT_WIRED_HEADSET) || (device == AudioSystem.DEVICE_OUT_WIRED_HEADPHONE))) { setBluetoothA2dpOnInt(true); } // 这个函数对AudioPolicy进行了通知 handleDeviceConnection((state == 1), device, ""); // 如果插入普通耳机，则会取消强制使用蓝牙耳机的设置 if ((state != 0) && ((device == AudioSystem.DEVICE_OUT_WIRED_HEADSET) || (device ==

```
AudioSystem.DEVICE_OUT_WIRED_HEADPHONE))) {  
    setBluetoothA2dpOnInt(false); } // 从名字就可以看出，这是用来广播外  
设状态变化通知的，将通知对此感兴趣的应用程序  
    sendDeviceConnectionIntent(device, state, name); } }
```

在这个函数中，我们需要重点关注的是对handleDeviceConnection () 和sendDevice-ConnectionIntent两个函数的调用。它们分别用来通知 AudioPolicy与上层应用。

另外，还可以看到，在handleDeviceConnection () 函数上下有一对关于蓝牙耳机的操作。从其实现上可以看出，如果拔出普通耳机，系统将会强制使用蓝牙耳机进行输出。如果插入耳机则会取消这个设置。这种操作完全可以放在AudioPolicyManager中实现。

看一下通知AudioPolicy的handleDeviceConnection () 函数的实现吧！

```
[AudioService.java-->AudioService.handleDeviceConnection()] private  
boolean handleDeviceConnection(boolean connected, int device, String  
params){ synchronized (mConnectedDevices) { boolean isConnected =  
(mConnectedDevices.containsKey(device) && (params.isEmpty() ||  
mConnectedDevices.get(device).equals(params))); if (isConnected &&  
!connected) { // 外设被拔出，通过AudioSystem将状态设置到底层的  
AudioPolicyService AudioSystem.setDeviceConnectionState(device,
```

```
AudioSystem.DEVICE_STATE_UNAVAILABLE,  
mConnectedDevices.get(device)); mConnectedDevices.remove(device);  
return true; } else if (!isConnected && connected) { // 外设被插入  
AudioSystem.setDeviceConnectionState(device,  
AudioSystem.DEVICE_STATE_AVAILABLE, params);  
mConnectedDevices.put(new Integer(device), params); return true; } }  
return false; }
```

很简单吧？如果读者对卷I第7章的内容比较熟悉，那么一定知道 `AudioSystem.setDeviceConnectionState ()` 这个函数意味着什么。它将更新底层的 `AudioPolicy` 中缓存的可用设备列表，同时，如果正在进行音频播放，那么这个函数还将触发音频设备的重新选择。



注意

这一节提到“可用设备列表”的次数很多，很多地方都使用了这个概念。归纳一下，在本节所讨论的内容里，有三个地方有可用设备列表：

- 1) WiredAccessoryObserver : 目的是确认外设的状态变化是否合法，是否需要报告给AudioService。
- 2) AudioService : 它以一个Hashtable的形式保存了一个可用设备列表，它为AudioService向应用及底层AudioPolicyManager发送通知提供依据。
- 3) AudioPolicyManager : 它保存的可用设备列表在AudioPolicyManager需要重新选择音频输出设备时提供候选。

2. 关于推迟处理外设状态

前面讨论checkSendBecomingNoisyIntent () 函数的实现时提到了根据某些条件，有可能使

MSG_SET_WIRED_DEVICE_CONNECTION_STAT延迟生效1秒。在这种情况下应用会在1秒之后才能收到设备状态变化的广播，同时，AudioPolicy也要在1秒之后才能更新可用设备列表并进行必要的设备切换。为什么要这么做呢？想想推迟的条件：

- 最后一个安静外设被移除，发送了BECOMING_NOISY广播。

- 队列中尚有两个消息在等候处理：

MSG_SET_WIRED_DEVICE_CONNECTION_STATE和

MSG_SET_A2DP_CONNECTION_STATE。

只要这两个条件有一个满足，就会发生1秒推迟。下面分别讨论。

关于第一个条件，当最后一个安静外设被移除后，手机上可用的音频输出设备就只剩下扬声器了（听筒不能算是常规的音频输出设备，它只有在通话过程中才会用到）。那么在

MSG_SET_WIRED_DEVICE_CONNECTION_STAT生效后，
AudioPolicyManager将会切换输出到扬声器，此时正在播放的音频就会被外放出来。

很多时候，这并不是用户所期望的，用户可能不希望他人知道自己在听什么，或者不希望在某些场合下扬声器发出的声音打扰到其他人。何况耳机被拔除有可能还是个意外。所以，正在进行音频播放的应用可能希望收到耳机等安静设备被拔出时的通知，并且在收到后暂停播放。

读者可能会有疑问，在sendDeviceConnectionIntent () 中不是发送了状态通知的广播了吗？其实，这个状态通知广播用在其他情况下可以，但是用在上述情况中是有问题的。按照上面的讨论，执行sendDeviceConnectionIntent () 之前，先执行了handleDeviceConnection () ，它会更新底层的可用设备列表，并且触发设备切换。于是应用有可能在收到状态通知之前，输出设备已经被切换成扬声器了，直到应用收到通知后暂停回放，这段时间内就会发生扬声器的漏音。

所以，Android引入了一个新的广播来应对这个问题，那就是BECOMING_NOISY广播。这个广播只有在最后一个安静外设被移除后才会发出，于是应用可以精确地知道音频即将从扬声器进行播放，而且后续的设备切换等动作被推迟了1秒，应用就有充足的时间收到BECOMING_NOISY广播并暂停播放。在正常情况下，这种做法可以杜绝漏音的情况出现。这是第一个延时条件的意义。

至于第二个条件，队列中尚有以下两个消息等候处理：
MSG_SET_WIRED_DEVICE_CONNECTION_STATE和
MSG_SET_A2DP_CONNECTION_STATE，这其实是不得已的一种做法。考虑一下，为什么队列中尚有这两个消息在等候处理呢？一个是mAudioHandler所在的线程发生了阻塞，另一个就是这两个消息被延迟发送了。根据Handler现有的接口没有办法得知是哪一种情况，但是在正常情况下都是第二种，也是比较麻烦的一种情况。因为在这种情况下，如果正常发送MSG_SET_WIRED_DEVICE_CONNECTION_STATE消息，那么它的生效时间将会早于正在队列中排队的那两个消息。如此一来，就会发生外设可用状态紊乱的问题。所以，AudioService迫不得已在这种情况下推迟发送1秒。读者可以做个试验，快速地在手机上拔插耳机，将会看到通知栏内的耳机图标的变化总是会延迟1秒。



注意

我们在之前的分析中没有见过

MSG_SET_A2DP_CONNECTION_STATE，它和讨论的
MSG_SET_WIRED_DEVICE_CONNECTION_STATE意义是一样的，而
且有着几乎相同的处理逻辑，不过它是与蓝牙耳机相关的。

3.3.3 音频外设管理小结

这一节以有线音频外设为例，探讨了从WiredAccessoryObserver收到UEvent开始到AudioService通知底层应用为止的AudioService对音频外设的管理机制。

总结一下音频外设拔插的处理过程：

- 由负责相关外设的模块监听从硬件上报的状态通知。将状态变化提交给AudioService进行处理。

- AudioService得到相关模块发来的通知，根据需要发送BECOMING_NOISY消息给应用，并更新自己的可用设备列表。
- AudioService将外设可用状态的变化通知AudioPolicy。 AudioPolicy更新自己的可用设备列表，并重新选取音频输出设备。
- AudioService将外设可用状态以广播的形式发送给应用等其他对此感兴趣的应用程序或系统模块。

蓝牙模块负责蓝牙耳机的连接/断开状态的监控并通知AudioService。 AudioService收到此通知之后的代码路径虽然与本节所讨论的内容不完全一样，但其处理原则与有线耳机是一致的，读者可以自行分析学习。

3.4 AudioFocus机制的实现

AudioFocus是自Android 2.3建立起来的一个新的机制。这套新机制的目的在于统一协调多个回放实例之间的交互。

我们知道，手机的多媒体功能越来越强大，听音乐、看视频、听收音机已经成为这台小小的设备的重要功能。加上手机本身的闹铃、信息通知以及电话铃声等，一台手机中有很多情况需要播放音频。我们称每一次音频播放为一次回放实例。这就需要我们能够对这些回放实例的并发情况做好协调，否则就会出现多个音频不合理地同时播放的恼人结果。

在2.3以前，Android并没有一套统一的管理机制。每个音频回放实例只能通过发送广播的方式告知其他人自己的播放状态。这不仅造成了广播满天飞的情况，而且可扩展性与一致性非常差，基本上只能在同一厂商的应用之间使用。好在，Android 2.3对AudioFocus的引入大大地改善了这个状况。

AudioFocus的含义可以和Windows的窗口焦点机制做类比，只不过我们的焦点对象是音频的回放实例。在同一时间，只能有一个音频回放实例拥有焦点。每个回放实例开始播放前，必须向AudioService申请获取AudioFocus，只有申请成功才允许开始回放。在回放实例播放结束

后，要求释放AudioFocus。在回放实例播放的过程中，AudioFocus有可能被其他回放实例抢走，这时，被抢走AudioFocus的回放实例需要根据情况采取暂停、静音或降低音量的操作，以突出拥有AudioFocus的回放实例的播放。当AudioFocus被还回来时，回放实例可以恢复被抢走之前的状态，继续播放。

总体上来说，AudioFocus是一个没有优先级概念的抢占式的机制。在一般情况下后一个申请者都能从前一个申请者的手中获取AudioFocus。不过只有一个例外，就是通话。通话作为手机的首要功能，同时也是一种音频的播放过程，所以从来电铃声开始到通话结束这个过程，Telephony相关的模块也会申请AudioFocus，但是它的优先级是最高的。Telephony可以从所有人手中抢走AudioFocus，但是任何人无法从它手中将其夺回。这在后面的代码分析中可以看到。

值得一提的是，AudioFocus机制完全是一个建议性而不是强制性的机制。也就是说，上述的行为是建议回放实例遵守，而不是强制的。所以，市面上仍有一些带有音频播放功能的应用没有采用这套机制。

3.4.1 AudioFocus最简单的例子

AudioFocus相关的API一共有三个，分别是requestAudioFocus（）
abandonAudioFocus（）以及AudioFocusListener回调接口，它们分别负责AudioFocus的申请、释放以及变化通知。

为了让大家了解AudioFocus的实现原理，我们先来看一个AudioFocus使用方法。下面是一个播放器的部分代码：

```
public void play() { // 在开始播放前，先申请AudioFocus，注意传入的参数
    int result = mAudioManager.requestAudioFocus(
        mAudiOfocusListener, AudioManager.STREAM_MUSIC,
        AudioManager.AUDIOFOCUS_GAIN); // 只有成功申请到AudioFocus之后才能开始播放
    if (result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED)
        mMediaPlayer.start();
    else // 申请失败，如果系统没有问题，这一定是正在通话过程中，所以，还是不要播放了
        showMessageCannotStartPlaybackDuringACall();
} public void stop() { // 停止播放时，需要释放AudioFocus
    mAudioManager.abandonAudioFocus(mAudiOfocusListener);
}
```

在开始播放前，应用首先要申请AudioFocus。申请AudioFocus时传入了3个参数。

- mAudiOfocusListener：参数名为l。AudioFocus变化通知回调。当AudioFocus被其他AudioFocus使用者抢走或归还时将通过这个回调对象进行通知。

·`AudioManager.STREAM_MUSIC`：参数名为`streamType`。这个参数表明了申请者即将使用哪种流类型进行播放。目前这个参数仅提供一项信息而已，对`AudioFocus`的机制没有任何影响，不过Android在后续的升级中可能会使用此参数。

·`AudioManager.AUDIOFOCUS_GAIN`：参数名为`durationHint`。这个参数指明了申请者将拥有这个`AudioFocus`多长时间。例子中传入的`AUDIOFOCUS_GAIN`表明了申请者将长期占用这个`AudioFocus`。另外还有两个可能的取值，它们是`AUDIOFOCUS_GAIN_TRANSIENT`和`AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK`。这两个取值的含义都表明申请者将暂时占用`AudioFocus`，不同的是，后者还指示了即将被申请者抢走`AudioFocus`的使用者不需要暂停，只要降低一下音量就可以了（这就是“DUCK”的意思）。

在停止播放时，需要调用`abandonAudioFocus`释放`AudioFocus`，会将其归还给之前被抢走`AudioFocus`的那个使用者。

接下来，我们看一下`mAudioFocusListener`是如何实现的。

```
private onAudioFocusChangeListener mAudioFocusListener =  
    new OnAudioFocusChangeListener() { // 当AudioFocus发生变化时，这个  
        // 函数将会被调用。其中参数focusChange指示发生了什么变化  
        public void onAudioFocusChange(int focusChange) { switch(focusChange) {
```

```
// AudioFocus被长期夺走，需要中止播放，并释放AudioFocus // 这种情
况对应于抢走AudioFocus的申请者使用了AUDIOFOCUS_GAIN case
AUDIOFOCUS_LOSS: stop(); break; // AudioFocus被临时夺走，不久就
会被归还，只需要暂停，AudioFocus被归还后再恢 复播放 // 这对应于
抢走AudioFocus的申请者使用了AUDIOFOCUS_GAIN_TRANSIENT
case AUDIOFOCUS_LOSS_TRANSIENT: saveCurrentPlayingState();
pause(); break; // AudioFocus被临时夺走，允许不暂停，所以降低音量 //
这对应于抢走AudioFocus的回放实例使用了
AUDIOFOCUS_GAIN_TRANSIENT_MAY_DUCK case
AUDIOFOCUS_LOSS_TRANSIENT_CAN_DUCK:
saveCurrentPlayingState(); setVolume(getVolume()/2); break; // AudioFocus
被归还，这时需要恢复被夺走前的播放状态 case
AUDIOFOCUS_GAIN: restorePlayingState(); break; } } };
```

从这里能够看出，AudioFocus机制的逻辑是完整而清晰的。理解上述例子以后，相信AudioFocus的工作原理已经浮现在脑海里了吧？如果有兴趣，可以不用着急阅读下面的分析，先思考一下AudioFocus可能的工作原理，然后再与Android的实现进行比较。



注意

从调用requestAudioFocus () 进行申请到abandonAudioFocus () 释放的这段时间内，只能说这个使用者参与了AudioFocus机制，不能保证一直拥有AudioFocus。

3.4.2 AudioFocus实现原理简介

看了前面的示例代码，可以推断出，AudioFocus的实现基础应该是一个栈。栈顶的使用者拥有AudioFocus。

在申请AudioFocus成功时，申请者被放置在栈顶，同时，通知之前在栈顶的使用者，告诉它新的申请者抢走了AudioFocus。

当释放AudioFocus时，使用者将从栈中被移除。如果这个使用者位于栈顶，则表明释放前它拥有AudioFocus，因此AudioFocus将被返还给新的栈顶。

工作原理参考图3-5。

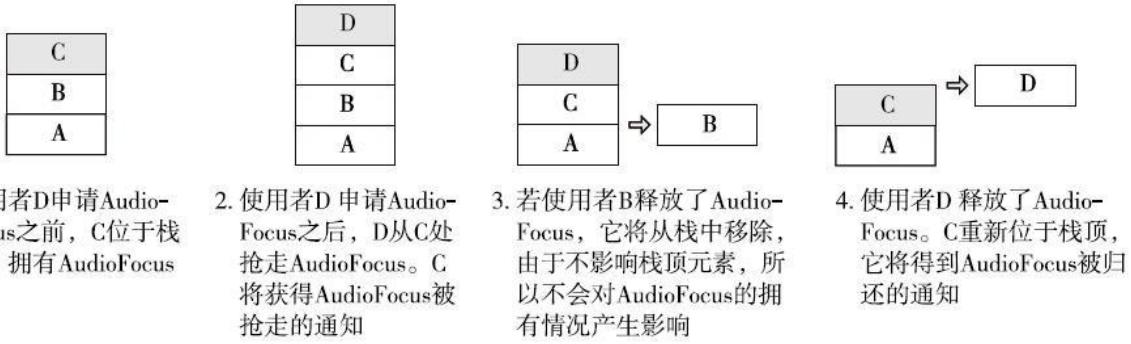


图3-5 AudioFocus的工作原理

AudioFocus的工作原理就是如此。下面将分别对申请和释放这两个过程进行详细分析。

3.4.3 申请AudioFocus

先看一下AudioFocus的申请过程。

```
[ AudioManager.java-->AudioManager.requestAudioFocus()]
public int requestAudioFocus(OnAudioFocusChangeListener l, int streamType, int durationHint) { int status = AUDIOFOCUS_REQUEST_FAILED; // 注册 AudioFocusListener registerAudioFocusListener(l); IAudioService service = getService(); try { // 调用AudioService的requestAudioFocus函数，参数很多，分别是什么意思呢 status = service.requestAudioFocus(streamType, durationHint, mICallBack, mAudioFocusDispatcher, getIdForAudioFocusListener(l), mContext.getPackageName()); } catch (RemoteException e) { } return status; }
```

这段代码有两个关键的地方，分别是调用registerAudioFocusListener() 和调用Audio-Service的requestAudioFocus() 函数。这个函数的参数很多，随着分析的深入，这些参数的意思会逐渐明朗。

下面先看一下registerAudioFocusListener做了什么。

```
[AudioManager.java-->AudioManager.registerAudioFocusListener()] public  
void registerAudioFocusListener(OnAudioFocusChangeListener l) {  
    synchronized(mFocusListenerLock) { if  
(mAudioFocusIdListenerMap.containsKey(getIdForAudioFocusListener(l)))  
{ return; } mAudioFocusIdListenerMap.put(getIdForAudioFocusListener(l),  
l); } }
```

原来，应用程序提供的OnAudioFocusChangeListener是被注册进 AudioManager的一个Hashtable中而不是AudioService中。注意，这个 Hashtable的KEY是getIdForAudioFocus-Listener () 分配的一个字符串型的Id。这样看来， AudioManager这一侧一定有一个代理负责接受 AudioService的回调并从这个Hashtable中通过Id将回调转发给相应的 Listener。

究竟是谁在做这个代理呢？就是稍候将作为参数传递给AudioService的 mAudio-FocusDispatcher。

```
[ AudioManager.java-->AudioManager.mAudioFocusDispatcher] private final  
IAudioFocusDispatcher mAudioFocusDispatcher = new  
IAudioFocusDispatcher.Stub() { public void dispatchAudioFocusChange(int  
focusChange, String id) { Message m =  
mAudioFocusEventHandlerDelegate.getHandler()  
.obtainMessage(focusChange, id);  
mAudioFocusEventHandlerDelegate.getHandler().sendMessage(m); } };
```

可以看出，mAudioFocusDispatcher的实现非常轻量级，直接把
focusChange和listener的id发送给了一个Handler去处理。



注意

这个看似繁冗的做法其实是很必要的。要知道，目前这个回调尚在
Binder的调用线程中，如果在这里因为用户传入的Listener的代码有问
题而报出异常或阻塞甚至恶意拖延，则会导致Binder的另一端因异常而
崩溃或阻塞。到这里为止，AudioService已经尽到了通知义务，应该通

过Handler将后续的操作发往另一个线程，使AudioService尽可能远离回调实现的影响。

看一下这个Handler的消息处理函数，msg.what存储了focusChange参数，msg.obj1存储了Id。因此handleMessage函数一定像下面这个样子：

[AudioManager.java--

```
>FocusEventHandlerDelegate.Handler.handleMessage() public void  
handleMessage(Message msg) { OnAudioFocusChangeListener listener =  
null; synchronized(mFocusListenerLock) { listener =  
findFocusListener((String)msg.obj); } if (listener != null) { // 通知使用者  
AudioFocus的归属发生了变化 listener.onAudioFocusChange(msg.what); }  
}
```

现在我们了解了AudioService的回调是如何传递给回放实例的。概括来说，mAudioFocusDispatcher作为AudioService与AudioManager的沟通桥梁，将回调操作以消息的方式发送给mFocusEventHandlerDelegate的Handler，在Handler的消息处理函数中通知回放实例。



注意

读者也许会有疑问，为什么不让AudioFocusChangeListener直接继承自AIDL描述的接口，非要由mAudioFocusDispatcher去做转发，这不是很麻烦呢？这是为了节约Binder的资源。虽说mAudioFocusDispatcher不是一个服务，但是其对Binder资源的占用却与服务一样，所以大量使用Binder回调是有待商榷的。这种把多个回调的信息保存在Bp端，使用一个拥有Binder通信能力的回调对象做它们的代理是一种很值得推荐的做法。

回到AudioManager.requestAudioFocus () 的实现中。我们调用AudioService.requestAudioFocus () 时传入的参数很多。它们都是什么意义呢？为了方便后面探讨，我们先把AudioService.requestAudioFocus的参数意义搞清楚。

- mainStreamType：这个参数目前没有被使用，这是为了便于以后功能扩展所预留的一个参数。
- focusChangeHint：这个参数指明了申请者持有AudioFocus的方式。
- cb：IBinder类型的一个参数，在探讨静音控制时曾经见过这个参数，就是AudioManager的mICallBack。应该能够联想到，AudioService又要做linkToDeath了。

·fd : IAudioFocusDispatcher对象，我们刚刚分析过，它是AudioService回调回放实例的中介。

·clientId : 参考 AudioManager.requestAudioFocus () 的实现，它是通过 getget-IdForAudioFocusListener () 函数获取的一个字符串，用于唯一标识一个Audio-FocusChangeListener。



注意

这个Id真的是唯一的吗？getIdForAudioFocusListener () 返回的其实就是一个toString () 。这样做是无法严格区分两个不同的 AudioFocusListener实例的。

·callingPackageName : 回放实例所在的包名。

接下来看看AudioService.requestAudioFocus () 的工作原理。

```
[AudioService-->AudioService.requestAudioFocus()] public int  
requestAudioFocus(int mainStreamType, int focusChangeHint, IBinder cb,  
IAudioFocusDispatcher fd, String clientId, String callingPackageName) { //
```

首先检查客户端提供的mICallBack是不是一个有效的Binder // 否则对其作linkToDeath没有任何意义 if (!cb.pingBinder()) { return AudioManager.AUDIOFOCUS_REQUEST_FAILED; }

synchronized(mAudioFocusLock) { // 检查一下当前情况下AudioService是否能够让申请者获取AudioFocus // 前面说过，如果通话占用了AudioFocus，任何人都不能够再申请AudioFocus if (!canReassignAudioFocus()) { return AudioManager.AUDIOFOCUS_REQUEST_FAILED; } /* 看到AudioFocusDeathHandler后，可以对比一下VolumeDeathHandler的工作，这里之所以需要监控其生命状态，就是为了防止一个使用者还没有来得及调用abandonAudioFocus就崩溃。所以读者一定知道VolumeDeathHandler的binderDied()函数的内容是什么 */

AudioFocusDeathHandler afdh = new AudioFocusDeathHandler(cb); try { cb.linkToDeath(afdh, 0); } catch (RemoteException e) { return AudioManager.AUDIOFOCUS_REQUEST_FAILED; } // mFocusStack就是AudioFocus机制所基于的栈 // mFocusStatck的元素类型为FocusStackEntry，它保存了一个回放实例相关的所有信息 // 这里先处理一下特殊情况，如果申请者已经拥有AudioFocus，那怎么办呢 if(!mFocusStack.empty()&&mFocusStack.peek().mClientId.equals(clientId)) { // 申请者已经位于栈顶的位置，也就是拥有了AudioFocus if (mFocusStack.peek().mFocusChangeType ==

focusChangeHint) { // 持有方式也没有变化，则直接返回成功 // 看到了吗？这里又要unlinkToDeath了。这明显表明没有好好规划这段代码
cb.unlinkToDeath(afdh, 0); return
AudioManager.AUDIOFOCUS_REQUEST_GRANTED; } //改变了持有方式，就把这个回放线程从栈上先拿下来，后面再重新加进去。这相当于重新申请 FocusStackEntry fse = mFocusStack.pop();
fse.unlinkToDeath(); } /* 接下来，我们通知位于栈顶，也就是当前拥有 AudioFocus的回放实例，它的AudioFocus 要被夺走了。这个操作的主角自然是我们已经熟悉的AudioFocusDispatcher。 */ if
(!mFocusStack.empty() && (mFocusStack.peek().mFocusDispatcher !=
null)){ try {
mFocusStack.peek().mFocusDispatcher.dispatchAudioFocusChange(-1 *
focusChangeHint, // GAIN和LOSS是相反数，很聪明，不是吗？
mFocusStack.peek().mClientId); } catch (RemoteException e) { } } /*由于
申请者可能曾经调用过requestAudioFocus，但是目前被别人夺走了。所
以它现在应该在 栈的某个位置上。先把它从栈中删除,注意第二个参数
的意思是不通知AudioFocus的变化 */ removeFocusStackEntry(clientId,
false); // 现在，为申请者创建新的FocusStackEntry放置到栈顶，使其拥
有AudioFocus mFocusStack.push(new FocusStackEntry(mainStreamType,
focusChangeHint, fd, cb, clientId, afdh, callingPackageName,
Binder.getCallingUid())); // AudioFocus易主，我们需要通知RemoteView,

但这不是我们目前的讨论范围 } // 告诉申请者它成功获得了

AudioFocus return

AudioManager.AUDIOFOCUS_REQUEST_GRANTED; }

代码比较长，但是思路还是比较清晰的。总结一下申请AudioFocus的工作内容：

- 通过canReassignAudioFocus () 判断当前是否在通话中，如果在通话过程中，则直接拒绝申请。具体如何让通话拥有最高优先级的问题可参考canReassignAudioFocus () 的实现。
- 对申请者进行linkToDeath。使得在申请者意外退出后可以代其完成abandon-AudioFocus操作。
- 对于已经持有AudioFocus的情况，如果没有改变持有方式，则不作任何处理，直接返回申请成功。否则将其从栈顶删除，暂时将栈顶让给下一个回放实例。
- 通过回调告知此时处于栈顶的回放实例，它的AudioFocus将被夺走。
- 将申请者的信息加入栈顶，成为新的拥有AudioFocus的回放实例。

申请AudioFocus的方式已经了解，那么释放AudioFocus是什么样的一个流程呢？读者可以自己先思考一下。

3.4.4 释放AudioFocus

先看AudioManager的abandonAudioFocus () 函数，从这个函数中可以看出，向Audio-Service申请释放AudioFocus需要提供两个“证件”：mAudioFocusDispatcher和AudioFocus-ChangeListener的ID。

```
[AudioManager.java-->AudioManager.abandonAudioFocus()] public int  
abandonAudioFocus(OnAudioFocusChangeListener l) { int status =  
AUDIOFOCUS_REQUEST_FAILED; // 取消注册listener。这里将把l从  
Hashtable中删除 unregisterAudioFocusListener(l); IAudioService service =  
getService(); try { // 调用AudioService的abandonAudiofocus()  
status =  
service.abandonAudioFocus(mAudioFocusDispatcher,  
getIdForAudioFocusListener(l)); } catch (RemoteException e) { ..... } return  
status; }
```

AudioService的abandonAudioFocus并没有更多的内容，只是调用了removeFocus-StackEntry () 函数而已。参考requestAudioFocus的实现过程，可以推断这个函数的工作有：

- 从mFocusStack中删除拥有指定clientId的回放实例的信息。
- 执行unlinkToDeath，取消监听其死亡通知。

·如果被删除的回放实例位于栈顶的位置，说明AudioFocus还给了另外一个回放实例，这时就要通过它的mFocusDispatcher回调，通知它重新获得了AudioFocus。

removeFocusStackEntry () 的工作就是如此，只是实现得不够简练。

```
[AudioService.java-->AudioService.removeFocusStackEntry()] private void  
removeFocusStackEntry(String clientToRemove, boolean signal) { if  
(!mFocusStack.empty() &&  
mFocusStack.peek().mClientId.equals(clientToRemove)) { // 取消对生命状  
态的监控 FocusStackEntry fse = mFocusStack.pop(); fse.unlinkToDeath();  
// 通知栈顶的回放实例， AudioFocus回来了 if (signal) {  
notifyTopOfAudioFocusStack(); // 通知RemoteControl， AudioFocus发生  
变化，这个不是我们讨论的主题 synchronized(mRCStack) {  
checkUpdateRemoteControlDisplay_syncAfRcs(RC_INFO_ALL); } } }  
else { // 从栈中寻找要删除的回放实例,然后将其从栈中删除 Iterator  
stackIterator = mFocusStack.iterator(); while(stackIterator.hasNext()) {  
FocusStackEntry fse = (FocusStackEntry)stackIterator.next();  
if(fse.mClientId.equals(clientToRemove)) { stackIterator.remove();  
fse.unlinkToDeath(); } } }
```



注意

看完AudioFocus的申请与释放的实现代码，读者能否感受到它们的实现在细节上确实有些臃肿和重复。对比曾经分析过的静音控制相关的代码，实在差距不小。阅读Android源代码的时候，我们不仅仅是在学习某些功能的原理，同时也是博取其代码组织与书写的精妙之处，发现其不足的地方并引以为戒。

关于AudioFocus，读者是否有自己的想法改造其实现，让其更加精炼吗？

3.4.5 AudioFocus小结

这一节学习了AudioFocus机制的工作原理。AudioFocus机制有三部分内容：申请、释放与回调通知，这些内容都是围绕一个名为mFocusStack的栈完成的。

在对代码的分析过程中，可以看到AudioFocus基本上是自成一个小的系统，没有和外部服务，尤其是Audio底层打过交道，而且AudioFocus

的回调通知只是告诉回放实例AudioFocus发生了变化，无法保证回放实例在回调中做什么。这说明了AudioFocus作为一个协调工具，是没有任何强制力的。希望在以后版本的Android中AudioFocus可以适当地增加一些约束能力使得这套机制可以发挥更大的作用。

即便如此，AudioFocus作为唯一的通用的音频交互策略，建议每一个涉及音频播放的应用都能参与这套机制，并且认真遵守其规则，这样才能保证Android音频“社会”的和谐。

3.5 AudioService的其他功能

这一章已经介绍了音量控制、外设管理及AudioFocus几个常用重要功能的实现。然而，AudioService仍然有很多其他相互独立的功能。限于篇幅，这里没有办法一一详细说明。在这里简单介绍一下，以便读者自行研究。

(1) RemoteControlClient/Display机制

RemoteControlClient/Display是从Android 4.0引入的一套新机制。它定义了一个远程控制端、一个远程显示端。这使得媒体播放过程中的元数据（例如标题、艺术家等）与其他信息可以跨应用显示。远程控制端由进行播放的应用管理，而远程显示端被一个显示界面管理，比如说一个AppWidget。由AudioService作为中介为它们进行配对与数据传递。

(2) MediaButton的管理

所谓的MediaButton是指线控耳机上的一个按键，虽然耳机线上只有一个按键，但是它的功能却异常得多，例如接听/挂断电话，启动音乐播放器，暂停/继续/下一首，等等。加上其使用方便，很多应用，尤其是播放器，争相操作（Handle）这个按键的事件。AudioService就是为了能够协调争抢这个按键的应用才插手管理这个按键的派发。

(3) 指定声音的输出设备

这个功能在AudioManger中表现为一系列名为setXXXOn的函数，其中的XXX表示了一个音频输出设备的名字。它们其实都使用了AudioService的setForceUse () 函数。准确地说，AudioService并没有为这个功能做过实际工作，只是作为应用到AudioPolicy的一个中介。

(4) 音效管理

AudioService在启动时，会使用SoundPool工具预加载一系列的音效文件，用于系统中的一些短小而频繁的音频播放，比如按键音。

SoundPool的工作原理是什么呢？在初始化时，AudioService要求SoundPool加载所需的音频文件。SoundPool会把这些音频文件解码为PCM音频流并缓存。同时为每段音频流分配一个ID，每当AudioService需要播放一段音效时，把对应的ID传递给SoundPool，SoundPool就会找到这块缓存的音频流，通过AudioTrack直接写入AudioFlinger中，实现音效播放。相对于MediaPlayer，由于每次播放时省却了prepare与编解码的过程，因此效率比其高很多，很适合用在游戏等对声音的及时性要求很高的场合。问题是，这个工具太消耗内存了。

(5) 情景模式

情景模式和音量的关联是比较紧密的，或者说，情景模式是在音量控制的基础上实现的一个功能。

(6) 音频状态管理

在AudioService中就是两个函数：getMode () 与setMode () 。音频状态表示了手机的4种状态，待机状态、音频通话状态、视频/VoIP通话状态与响铃状态。这4种状态对底层的音频输出设备的选择影响很大，同时也影响了AudioService的一些行为，例如MediaButton的管理。

3.6 本章小结

这一章介绍了AudioService的几个重要的功能，相信大家通过这章对Audio系统在Java Framework层面所做的事情有了一个比较深入的了解。由于AudioService的功能太过繁杂，本章只能将几个有代表意义并且实际接触比较多的内容进行讲解与探讨。若想更加了解AudioService及其周边模块的工作原理仍需要读者不懈努力。

另外，AudioService的一些功能都涉及AudioPolicy的相关内容，所以在学习本章时要多参考AudioPolicy的相关知识。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第4章 深入理解

WindowManagerService

本章主要内容：

- 介绍最原始、最简单的窗口创建方法

- 研究WMS的窗口管理结构

- 探讨WMS布局系统的工作原理

- 研究WMS动画系统的工作原理

本章涉及的源代码文件名及位置：

- SystemServer.java

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

- WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

·ActivityStack.java

frameworks/base/services/java/com/android/server/am/ActivityStack.java

·WindowState.java

frameworks/base/services/java/com/android/server/wm/WindowState.java

·PhoneWindowManager.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow
Manager.java

·AccelerateDecelerateInterpolator.java

frameworks/base/core/java/android/view/animation/AccelerateDecelerateInt
erpolator.java

·Animation.java

frameworks/base/core/java/android/view/animation/Animation.java

·AlphaAnimation.java

frameworks/base/core/java/android/view/animation/AlphaAnimation.java

·WindowAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowAnimator.java

va

·WindowStateAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowStateAnimat

or.java

4.1 初识WindowManagerService

WindowManagerService（以下简称WMS）是继ActivityManagerService与Package-ManagerService之后又一个复杂却十分重要的系统服务。

在介绍WMS之前，首先要了解窗口（Window）是什么。

Android系统中的窗口是屏幕上的一块用于绘制各种UI元素并可以响应用户输入的一个矩形区域。从原理上来讲，窗口的概念是独自占有一个Surface实例的显示区域。例如Dialog、Activity的界面、壁纸、状态栏以及Toast等都是窗口。

卷I第8章曾详细介绍了Activity通过Surface来显示自己的过程：

- Surface是一块画布，应用可以随心所欲地通过Canvas或者OpenGL在其上作画。
- 然后通过SurfaceFlinger将多块Surface的内容按照特定的顺序（Z-order）进行混合并输出到FrameBuffer，从而将Android“漂亮的脸蛋”显示给用户。

既然每个窗口都有一块Surface供自己涂鸦，必然需要一个角色对所有窗口的Surface进行协调管理。于是WMS应运而生。WMS为所有窗口分

配Surface，掌管Surface的显示顺序（Z-order）以及位置尺寸，控制窗口动画，并且还是输入系统的一个重要中转站。



说明

一个窗口拥有显示和响应用户输入这两层含义，本章将侧重于分析窗口的显示，而响应用户输入的过程则在第5章中详细介绍。

本章将深入分析WMS的两个基础子系统的工作原理：

- 布局系统（Layout System）：计算与管理窗口的位置、层次。
- 动画系统（Animation System）：根据布局系统计算的窗口位置与层次渲染窗口动画。

为了让读者对WMS的功能以及工作方式有一个初步认识，并见识一下WMS的强大，本节将从一个简单而神奇的例子开始WMS的学习之旅。

4.1.1 一个从命令行启动的动画窗口

1.SampleWindow的实现

在这一节里将编写一个最简单的Java程序SampleWindow，仅使用WMS的接口创建并渲染一个动画窗口。此程序将抛开Activity、Wallpaper等UI架构的复杂性，直接了当地揭示WMS的客户端如何申请、渲染并注销自己的窗口。同时这也初步地反映了WMS的工作方式。

这个例子很简单，只有三个文件：

- SampleWindow.java 主程序源代码。

- Android.mk 编译脚本。

- sw.sh 启动器。

分别看一下这三个文件的实现：

```
[SampleWindow.java-->SampleWindow] package  
understanding.wms.samplewindow; ..... public class SampleWindow {  
public static void main(String[] args) { try { //SampleWindow.Run()是这个  
程序的主入口 new SampleWindow().Run(); } catch (Exception e) {  
e.printStackTrace(); } } // IWindowSession 是客户端向WMS请求窗口操  
作的中间代理，并且是进程唯一的 IWindowSession mSession = null; //  
InputChannel 是窗口接收用户输入事件的管道。在第5章中将对其进行  
详细探讨 InputChannel mInputChannel = new InputChannel(); // 下面的三  
个Rect保存了窗口的布局结果。其中mFrame表示了窗口在屏幕上的位
```

置与尺寸 // 在4.4节中将详细介绍它们的作用以及计算原理 Rect mInsets = new Rect(); Rect mFrame = new Rect(); Rect mVisibleInsets = new Rect(); Configuration mConfig = new Configuration(); // 窗口的Surface，在此Surface上进行的绘制都将在此窗口上显示出来 Surface mSurface = new Surface(); // 用于在窗口上进行绘图的画刷 Paint mPaint = new Paint(); // 添加窗口所需的令牌，在4.2节将会对其进行介绍 IBinder mToken = new Binder(); // 一个窗口对象，本例演示了如何将此窗口添加到WMS中，并在其上进行绘制操作 MyWindow mWindow = new MyWindow(); // WindowManager.LayoutParams 定义窗口的布局属性，包括位置、尺寸以及窗口类型等 LayoutParams mLp = new LayoutParams(); Choreographer mChoreographer = null; // InputHandler 用于从 InputChannel 接收按键事件并做出响应 InputHandler mInputHandler = null; boolean mContinueAnime = true; public void Run() throws Exception { Looper.prepare(); // 获取WMS服务 IWindowManager wms = IWindowManager.Stub.asInterface(ServiceManager.getService(Context.WINDOW_SERVICE)); // 通过 WindowManagerGlobal 获取进程唯一的IWindowSession实例。它将用于向WMS // 发送请求。注意这个函数在较早的Android版本（如4.1）位于ViewRootImpl类中 mSession = WindowManagerGlobal.getWindowSession(Looper.myLooper()); // 获取屏幕分辨率 IDisplayManager dm = IDisplayManager.Stub.asInterface(

```
ServiceManager.getService(Context.DISPLAY_SERVICE)); DisplayInfo di  
= dm.getDisplayInfo(Display.DEFAULT_DISPLAY); Point scrnSize = new  
Point(di.appWidth, di.appHeight); // 初始化WindowManager.LayoutParams  
initLayoutParams(scrnSize); // 将新窗口添加到WMS installWindow(wms);  
// 初始化Choreographer的实例，此实例为线程唯一。这个类的用法与  
Handler // 类似，不过它总是在VSYNC同步时回调，所以比Handler更适合  
做动画的循环器 mChoreographer = Choreographer.getInstance(); // 开始  
处理第一帧的动画 scheduleNextFrame(); // 当前线程陷入消息循环，直  
到Looper.quit() Looper.loop(); // 标记不要继续绘制动画帧  
mContinueAnime = false; // 卸载当前Window uninstallWindow(wms); }  
public void initLayoutParams(Point screenSize) { // 标记即将安装的窗口  
类型为SYSTEM_ALERT，这将使得窗口的ZOrder顺序比较靠前  
mLp.type = LayoutParams.TYPE_SYSTEM_ALERT;  
mLp.setTitle("SampleWindow"); // 设定窗口的左上角坐标以及高度和宽  
度 mLp.gravity = Gravity.LEFT | Gravity.TOP; mLp.x = screenSize.x / 4;  
mLp.y = screenSize.y / 4; mLp.width = screenSize.x / 2; mLp.height =  
screenSize.y / 2; // 和输入事件相关的Flag，希望当输入事件发生在此窗  
口之外时，其他窗口也可以接收输入事件 mLp.flags = mLp.flags |  
 WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL; } public  
void installWindow(IWindowManager wms) throws Exception { // 首先向  
WMS声明一个Token，任何一个Window都需要隶属于一个特定类型的
```

```
Token wms.addWindowToken(mToken,
    WindowManager.LayoutParams.TYPE_SYSTEM_ALERT); // 设置窗口所
隶属的Token mLp.token = mToken; // 通过IWindowSession将窗口安装进
WMS，注意，此时仅仅是安装到WMS，本例的Window // 目前仍然没
有有效的Surface。不过，经过这个调用后，mInputChannel已经可以
用来接收 // 输入事件了 mSession.add(mWindow, 0, mLp, View.VISIBLE,
mInsets, mInputChannel); /*通过IWindowSession要求WMS对本窗口进行
重新布局，经过这个操作后，WMS将会为窗口 创建一块用于绘制的
Surface并保存在参数mSurface中。同时，这个Surface被WMS放置在
LayoutParams所指定的位置上 */ mSession.relayout(mWindow, 0, mLp,
mLp.width, mLp.height, View.VISIBLE, 0, mFrame, mInsets,
mVisibleInsets, mConfig, mSurface); if (!mSurface.isValid()) { throw new
RuntimeException("Failed creating Surface."); } // 基于WMS返回的
InputChannel创建一个Handler，用于监听输入事件 // mInputHandler一旦
被创建，就已经在监听输入事件了 mInputHandler = new
InputHandler(mInputChannel, Looper.myLooper()); } public void
uninstallWindow(IWindowManager wms) throws Exception { // 从WMS处
卸载窗口 mSession.remove(mWindow); // 从WMS处移除之前添加的
Token wms.removeWindowToken(mToken); } public void
scheduleNextFrame() { // 要求在显示系统刷新下一帧时回调
mFrameRender，注意，只回调一次
```

```
mChoreographer.postCallback(Choreographer.CALLBACK_ANIMATION ,  
mFrameRender, null); } // 这个Runnable对象用于在窗口上描绘一帧  
public Runnable mFrameRender = new Runnable() { @Override public void  
run() { try { // 获取当期时间戳 long time =  
mChoreographer.getFrameTime() % 1000; // 绘图 if (mSurface.isValid()) {  
Canvas canvas = mSurface.lockCanvas(null);  
canvas.drawColor(Color.DKGRAY); canvas.drawRect(2 * mLp.width *  
time / 1000 - mLp.width, 0, 2 * mLp.width * time / 1000, mLp.height,  
mPaint); mSurface.unlockCanvasAndPost(canvas);  
mSession.finishDrawing(mWindow); } if (mContinueAnime)  
scheduleNextFrame(); } catch (Exception e) { e.printStackTrace(); } } }; //  
定义一个类继承InputEventReceiver，用于在其onInputEvent()函数中接  
收窗口的输入事件 class InputHandler extends InputEventReceiver {  
Looper mLooper = null; public InputHandler(InputChannel inputChannel,  
Looper looper) { super(inputChannel, looper); mLooper = looper; }  
@Override public void onInputEvent(InputEvent event) { if (event  
instanceof MotionEvent) { MotionEvent me = (MotionEvent)event; if  
(me.getAction() == MotionEvent.ACTION_UP) { // 退出程序  
mLooper.quit(); } } super.onInputEvent(event); } } // 实现一个继承自  
IWindow.Stub的类MyWindow class MyWindow extends IWindow.Stub { //  
保持默认的实现即可 } }
```

(注：关于Choreographer，请参考卷I和卷II的作者邓凡平的博客文章
《Android Project Butter》分析
(<http://blog.csdn.net/innost/article/details/8272867>）。)

由于此程序使用了大量的隐藏API（即SDK中没有定义这些API），因此需要放在Android源代码环境中进行编译。对应的Android.mk如下：

```
[Android.mk] LOCAL_PATH:= $(call my-dir) include $(CLEAR_VARS)  
LOCAL_SRC_FILES := $(call all-subdir-jar-files)  
LOCAL_MODULE_TAGS := optional LOCAL_MODULE :=  
samplewindow include $(BUILD_JAVA_LIBRARY)
```

将这两个文件放在\$TOP/frameworks/base/cmds/samplewindow/下，然后用make或mm命令进行编译。最终生成的结果是samplewindow.jar，文件位置在out/target/<ProductName>/system/framework/下。将该文件通过adb push到手机的/system/framework/下。



提示

读者可使用Android 4.2模拟器来运行此程序。

然而，samplewindow.jar不是一个可执行程序，故需借助Android的app_process工具来加载并执行。笔者编写了一个脚本作为启动器：

```
[sw.sh] base=/system export  
CLASSPATH=$base/framework/samplewindow.jar exec app_process  
$base/bin understanding.wms.samplewindow.SampleWindow "$@"
```



注意

app_process其实就是大名鼎鼎的zygote。不过，只有使用--zygote参数启动时它才会改名为zygote[1]，否则就像java-jar命令一样，运行指定类的main静态函数。

在手机中执行该脚本，其运行结果是一个灰色的方块不断地从屏幕左侧移动到右侧，如图4-1所示。

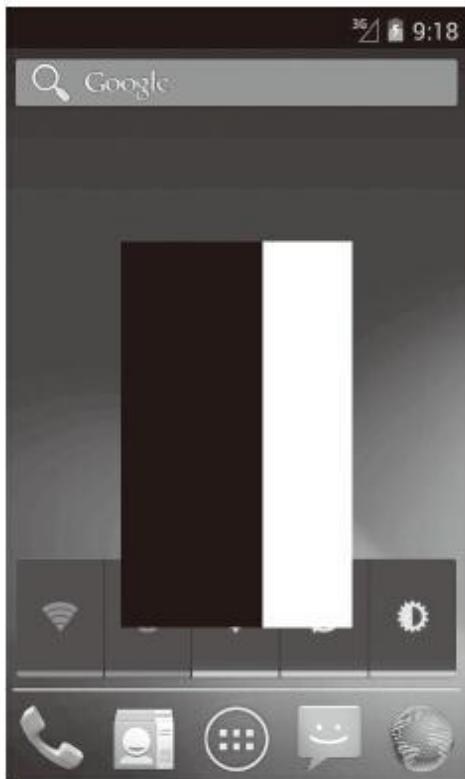


图4-1 SampleWindow在手机中的运行效果

2.初识窗口的创建、绘制与销毁

SampleWindow的这段代码虽然简单，但是却很好地提炼了一个窗口的创建、绘制以及销毁的过程。注意，本例没有使用任何WMS以外的系统服务，也没有使用Android系统4大组件的框架，也就是说，如果你愿意，可以利用WMS实现自己的UI与应用程序框架，这样就可以衍生出一个新的平台。

总结在客户端创建一个窗口的步骤：

· 获取IWindowSession和WMS实例。客户端可以通过IWindowSession向WMS发送请求。

· 创建并初始化 WindowManager.LayoutParams。注意这里是 WindowManager下的LayoutParams，它继承自 ViewGroup.LayoutParams 类，并扩展了一些窗口相关的属性。其中最重要的是 type 属性。这个属性描述了窗口的类型，而窗口类型正是 WMS 对多个窗口进行 ZOrder 排序的依据。

· 向 WMS 添加一个窗口令牌（WindowToken）。本章后续将分析窗口令牌的概念，目前读者只要知道，窗口令牌描述了一个显示行为，并且 WMS 要求每一个窗口必须隶属于某一个显示令牌。

· 向 WMS 添加一个窗口。必须在 LayoutParams 中指明此窗口所隶属于的窗口令牌，否则在某些情况下添加操作会失败。在 SampleWindow 中，不设置令牌也可成功完成添加操作，因为窗口的类型被设为 TYPE_SYSTEM_ALERT，它是系统窗口的一种。而对于系统窗口，WMS 会自动为其创建显示令牌，故无须客户端操心。此话题将会在后文进行具体讨论。

· 向 WMS 申请对窗口进行重新布局（relayout）。所谓的重新布局，就是根据窗口新的属性去调整其 Surface 相关的属性，或者重新创建一个 Surface（例如窗口尺寸变化导致之前的 Surface 不满足要求）。向 WMS

添加一个窗口之后，其仅仅是将它在WMS中进行注册而已。只有经过重新布局之后，窗口才拥有WMS为其分配的画布。有了画布，窗口之后就可以随时进行绘制工作了。

而窗口的绘制过程如下：

- 通过Surface.lock () 函数获取可以在其上作画的Canvas实例。
- 使用Canvas实例进行作画。
- 通过Surface.unlockCanvasAndPost () 函数提交绘制结果。



提示

关于Surface的原理与使用方法，请参考卷I第8章。

这是对Surface作画的标准方法。在客户端也可以通过OpenGL进行作画，不过这超出了本书的讨论范围。另外，在SampleWindow例子中使用了Choreographer类进行动画帧安排。Choreographer意为编舞指导，是Jelly Bean新增的一个工具类。其用法与Handler的post () 函数非常

之像，都会在后续的某个时机回调传入的Runnable对象。不同的它们的回调时机有所差异。因为Handler回调时机取决于消息队列的处理情况，而Choreographer的回调时机则为下一次VSYNC（垂直刷新同步）。在WMS内部一样使用了Choreographer类进行窗口的平移、缩放、旋转等动画的渲染。

销毁窗口的操作则简单了很多，只要通过IWindowSession.remove ()方法窗口从WMS中删除即可。如果客户端已经完成了自己的工作并且不会再显示新的窗口，则需要从WMS将之前添加的显示令牌一并删除。

3. 窗口的概念

在SampleWindow例子中，有一个名为mWindow（类型为IWindow）的变量。读者可能会理所当然地认为它就是窗口了。其实这种认识并不完全正确。IWindow继承自Binder，并且其Bn端位于应用程序一侧（在例子中IWindow的实现类MyWindow就继承自IWindow.Stub），于是其在WMS一侧只能作为一个回调，以及起到窗口Id的作用。

那么，窗口的本质是什么呢？

是进行绘制所使用的画布：Surface。

当一块Surface显示在屏幕上时，就是用户所看到的窗口了。客户端向WMS添加一个窗口的过程，其实就是WMS为其分配一块Surface的过程，一块块Surface在WMS的管理之下有序地排布在屏幕上，Android才得以呈现出多姿多彩的界面。所以从这个意义上讲，WindowManagerService被称为SurfaceManagerService也说得通。

于是，根据对Surface的操作类型可以将Android的显示系统分为三个层次，如图4-2所示。

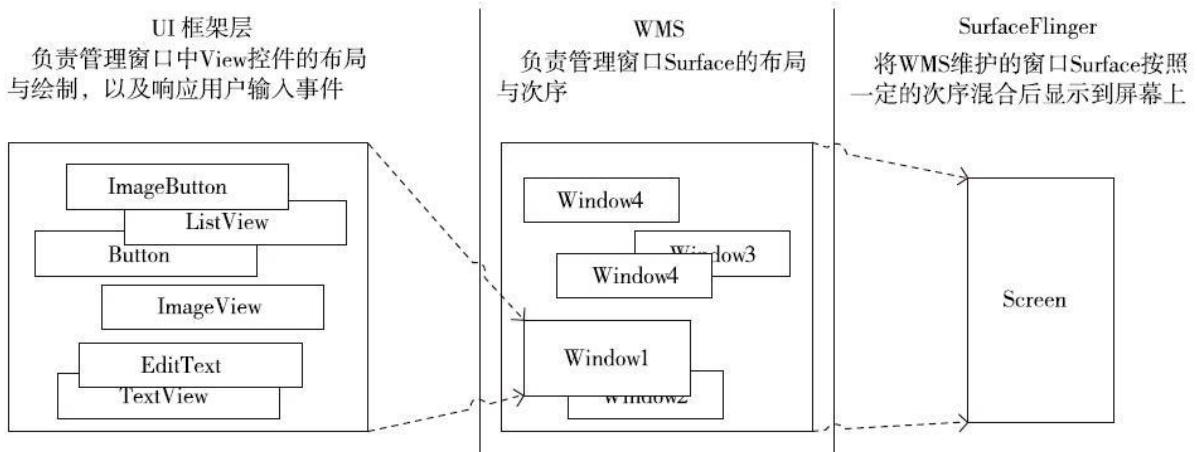


图4-2 Android显示系统的三个层次

在图4-2中：

- 第一个层次是UI框架层，其工作为在Surface上绘制UI元素以及响应输入事件。
- 第二个层次为WMS，其主要工作是管理Surface的分配、层级顺序等。

·第三个层次为SurfaceFlinger，负责将多个Surface混合并输出。

经过这个例子的介绍，相信大家对WMS的功能有了一个初步了解。接下来，我们要进入WMS的内部，通过其启动过程一窥它的构成。

[1] 读者可阅读卷I第4章“深入理解Zygote”来了解和Zygote相关的知识。

4.1.2 WMS的构成

俗话说，一个好汉三个帮！WMS的强大是由很多重要的成员互相协调工作而实现的。了解WMS的构成将会为我们深入探索WMS打下良好的基础，进而分析它的启动过程，这是再合适不过了。

1.WMS的诞生

和其他的系统服务一样，WMS的启动位于SystemServer.java中ServerThread类的run（）函数内。

```
[SystemServer.java-->ServerThread.run()]
Public void run() { .....
    WindowManagerService wm = null; ..... try { ..... // ① 创建WMS实例 /*
    通过WindowManagerService的静态函数main()创建
    WindowManagerService的实例。 注意main()函数的两个参数wmHandler
    和uiHandler。这两个Handler分别运行于由 ServerThread所创建的两个
    名为“WindowManager”和“UI”的两个HandlerThread中 */
    wm =
    WindowManagerService.main(context, power, display, inputManager,
```

```
uiHandler, wmHandler, factoryTest !=  
SystemServer.FACTORY_TEST_LOW_LEVEL, !firstBoot, onlyCore); // 添  
加到ServiceManager中  
ServiceManager.addService(Context.WINDOW_SERVICE, wm); ..... catch  
(RuntimeException e) { ..... } ..... try { // ② 初始化显示信息  
wm.displayReady(); } catch (Throwable e) {.....} ..... try { // ③ 通知  
WMS，系统的初始化工作完成 wm.systemReady(); } catch (Throwable e)  
{.....} ..... }
```

由此可以看出，WMS的创建分为三个阶段：

- 创建WMS的实例。
- 初始化显示信息。
- 处理systemReady通知。

接下来，将通过以上三个阶段分析WMS从无到有的过程。

看一下WMS的main（）函数的实现：

```
[WindowManagerService.java-->WindowManagerService.main()] public  
static WindowManagerService main(final Context context, final  
PowerManagerService pm, final DisplayManagerService dm, final  
InputManagerService im, final Handler uiHandler, final Handler
```

```
wmHandler, final boolean haveInputMethods, final boolean showBootMsgs,  
final boolean onlyCore) { final WindowManagerService[] holder = new  
WindowManagerService[1]; // 通过由SystemServer为WMS创建的Handler  
新建一个WindowManagerService对象 // 此Handler运行在一个名为  
WindowManager的HandlerThread中 wmHandler.runWithScissors(new  
Runnable() { @Override public void run() { holder[0] = new  
WindowManagerService(context, pm, dm, im, uiHandler,  
haveInputMethods, showBootMsgs, onlyCore); } }, 0); return holder[0]; }
```



注意

Handler类在Android 4.2中新增了一个API：runWithScissors（）。这个函数将会在Handler所在的线程中执行传入的Runnable对象，同时阻塞调用线程的执行，直到Runnable对象的run（）函数执行完毕。

WindowManagerService.main（）函数在ServerThread专为WMS创建的线程“Window-Manager”上创建了一个WindowManagerService的新实

例。WMS中所有需要的Looper对象，例如Handler、Choreographer等，将会运行在“WindowManager”线程中。

接下来看一下其构造函数，看一下WMS定义了哪些重要的组件。

[WindowManagerService.java--

```
>WindowManagerService.WindowManagerService() { private  
WindowManagerService(Context context, PowerManagerService pm,  
DisplayManagerService displayManager, InputManagerService  
inputManager, Handler uiHandler, boolean haveInputMethods, boolean  
showBootMsgs, boolean onlyCore) { ..... mDisplayManager =  
(DisplayManager)context.getSystemService(Context.DISPLAY_SERVICE);  
mDisplayManager.registerDisplayListener(this, null); Display[] displays =  
mDisplayManager.getDisplays(); /* 初始化DisplayContent列表。  
DisplayContent是Android4.2为支持多屏幕输出所引入的一个 概念。一  
个DisplayContent指代一块屏幕，屏幕可以是手机自身的屏幕，也可以  
是基于Wi-FiDisplay 技术的虚拟屏幕 for (Display display : displays) {  
createDisplayContentLocked(display); } ..... /* 保存InputManagerService。  
输入事件最终要分发给具有焦点的窗口，而WMS是窗口管理者，所以  
WMS是输入系统中的重要一环。关于输入系统的内容将在第5章中深  
入探讨*/ mInputManager = inputManager; // 这个看起来其貌不扬的  
mAnimator，事实上具有非常重要的作用。它管理着所有窗口的动画
```

```
mAnimator = new WindowAnimator(this, context, mPolicy); // 在“UI”线程  
中将对另一个重要成员mPolicy，也就是WindowManagerPolicy进行初始化  
initPolicy(uiHandler); // 将自己加入到Watchdog中  
Watchdog.getInstance().addMonitor(this); ..... }
```

(注：关于Wi-Fi Display的详细信息，读者可参考
<http://blog.csdn.net/innoST/article/details/8474683>的介绍。)

第二步，displayReady () 函数的调用主要是初始化显示尺寸的信息。
其内容比较琐碎，这里就先不介绍了。不过值得注意的一点是，在
displayReady () 完成后，WMS会要求ActivityManagerService进行第一
次Configuration更新。

第三步，在systemReady () 函数中，WMS本身将不会再进行任何操
作，直接调用mPolicy的systemReady () 函数。

2.WMS的重要成员

总结一下在WMS的启动过程中所创建的重要成员，参考图4-3。

以下是对图4-3中重要成员的简单介绍：

·mInputManager，InputManagerService（输入系统服务）的实例。用于
管理每个窗口的输入事件通道（InputChannel）以及向通道上派发事
件。关于输入系统的详细内容将在本书第5章详细探讨。

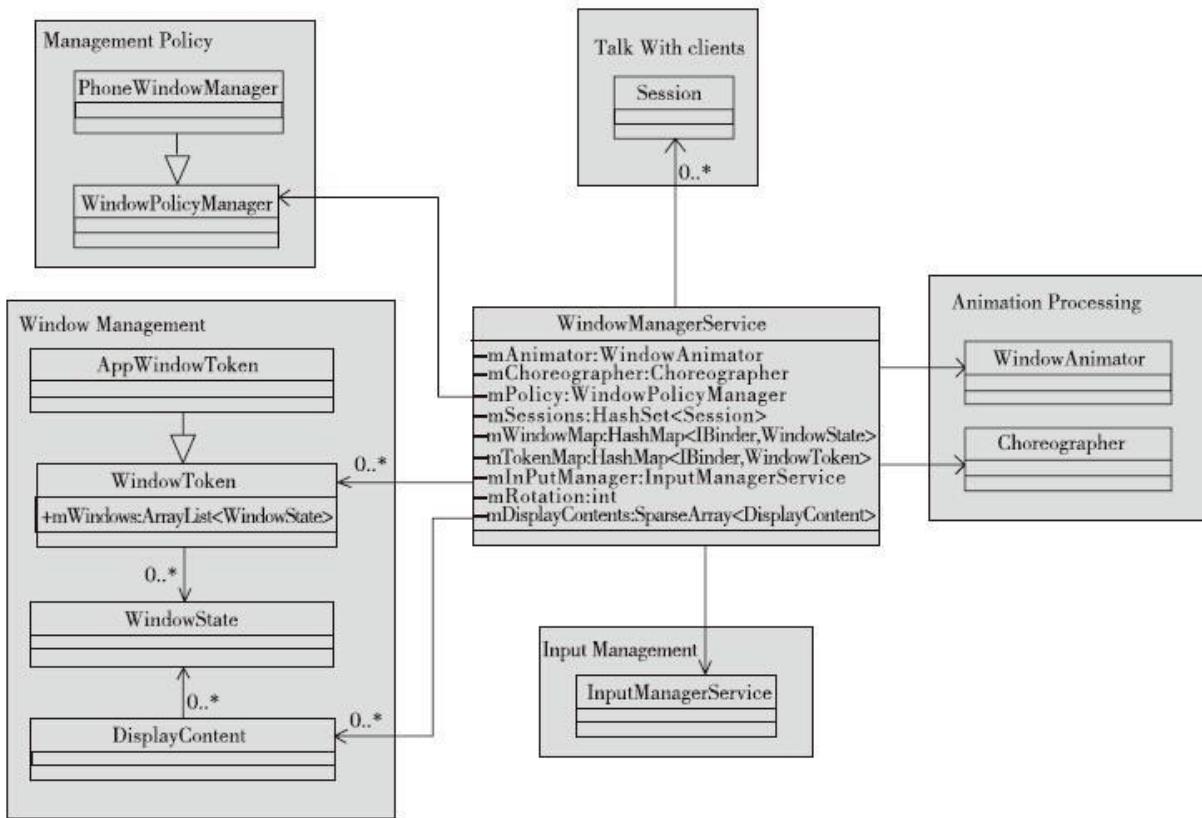


图4-3 WMS的重要成员

·mChoreographer，Choreographer的实例，在SampleWindow的例子中已经见过了。Choreographer的意思是编舞指导。它拥有从显示子系统获取VSYNC同步事件的能力，从而可以在合适的时机通知渲染动作，避免在渲染的过程中因为发生屏幕重绘而导致的画面撕裂。从这个意义上讲，Choreographer的确是指导Android翩翩起舞的大师。WMS使用Choreographer负责驱动所有的窗口动画、屏幕旋转动画、墙纸动画的渲染。

·mAnimator，WindowAnimator的实例。它是所有窗口动画的总管（窗口动画是一个WindowStateAnimator对象）。在Choreographer的驱动下，逐个渲染所有的动画。

·mPolicy，WindowPolicyManager的一个实现。目前它只有PhoneWindowManager一个实现类。mPolicy定义了很多窗口相关的策略，可以说是WMS的首席顾问！每当WMS要做什么事情的时候，都需要向这个顾问请教应当如何做。例如，告诉WMS某一个类型的Window的ZOrder的值是多少，帮助WMS矫正不合理的窗口属性，会为WMS监听屏幕旋转的状态，还会预处理一些系统按键事件（例如HOME、BACK键等的默认行为就是在这里实现的），等等。所以，mPolicy可谓WMS中最重要的一个成员了。

·mDisplayContents，一个DisplayContent类型的列表。Android 4.2支持基于Wi-Fi Display的多屏幕输出，而一个DisplayContent描述了一块可以绘制窗口的屏幕。每个DisplayContent都用一个整型变量作为其ID，其中手机默认屏幕的ID由Display.DEFAULT_DISPLAY常量指定。DisplayContent的管理是由DisplayManagerService完成的，在本章不会去探讨DisplayContent的实现细节，而是关注DisplayContent对窗口管理与布局的影响。

下面的几个成员的初始化并没有出现在构造函数中，不过它们的重要性一点也不亚于上面几个。

·mTokenMap，一个HashMap，保存了所有的显示令牌（类型为WindowToken），用于窗口管理。在SampleWindow例子中曾经提到过，一个窗口必须隶属于某一个显示令牌。在那个例子中所添加的令牌就被放进了这个HashMap中。从这个成员中还衍生出几个辅助的显示令牌的子集，例如mAppTokens保存了所有属于Activity的显示令牌（WindowToken的子类AppWindowToken），mExitingTokens则保存了正在退出过程中的显示令牌等。其中mAppTokens列表是有序的，它与AMS中的mHistory列表的顺序保持一致，反映了系统中Activity的顺序。

·mWindowMap，也是一个HashMap，保存了所有窗口的状态信息（类型为WindowState），用于窗口管理。在SampleWindow例子中，使用IWindowSession.add () 所添加的窗口的状态将会被保存在mWindowMap中。与mTokenMap一样，mWindowMap一样有衍生出的子集。例如mPendingRemove保存了那些退出动画播放完成并即将被移除的窗口，mLosingFocus则保存了那些失去了输入焦点的窗口。在DisplayContent中，也有一个windows列表，这个列表存储了显示在此Display-Content中的窗口，并且它是有序的。窗口在这个列表中的位置决定了其最终显示时的Z序。

·mSessions，一个List，元素类型为Session。Session其实是SampleWindow例子中的IWindowSession的Bn端。也就是说，mSessions

这个列表保存了当前所有想向WMS寻求窗口管理服务的客户端。注意Session是进程唯一的。

·mRotation，只是一个int型变量。它保存了当前手机的旋转状态。

WMS定义的成员一定不止这些，但是它们是WMS每一种功能最核心的变量。读者在这里可以先对它们有一个感性认识。在本章后续的内容里将会详细分析它们在WMS的各种工作中所发挥的核心作用。

4.1.3 初识WMS的小结

这一节通过SampleWindow的例子向读者介绍了WMS的客户端如何使用窗口，然后通过WMS的诞生过程简单剖析了WMS的重要组成成员，以期通过本节的学习能够为后续的学习打下基础。

从下一节开始，我们将会深入探讨WMS的工作原理。

4.2 WMS的窗口管理结构

经过上一节的介绍，读者应该对WMS的窗口管理有了一个感性认识。从这一节开始将深入WMS的内部去剖析其工作流程。

根据前述内容可知，SampleWindow添加窗口的函数是IWindowSession.add()。IWindowSession是WMS与客户端交互的一个代理，add则直接调用了WMS的addWindow()函数。我们将从这个函数开始WMS之旅。本小节只讨论它的前半部分。



注意

由于篇幅所限，本章不准备讨论removeWindow的实现。

```
[ WindowManagerService.java-->WindowManagerService.addWindow()  
Part1] public int addWindow(Session session, IWindow client, int seq,  
WindowManager.LayoutParams attrs, int viewVisibility,int displayId Rect  
outContentInsets, InputChannel outInputChannel) { //首先检查权限，没有
```

权限的客户端不能添加窗口 int res = mPolicy.checkAddPermission(attrs);
..... // 当为某个窗口添加子窗口时， attachedWindow将用来保存父窗口
的实例 WindowState attachedWindow = null; // win就是即将被添加的窗
口 WindowState win = null; final int type = attrs.type;
synchronized(mWindowMap) { //① 获取窗口要添加的DisplayContent
/* 在添加窗口时，必须通过displayId参数指定添加到哪一个
DisplayContent。 SampleWindow例子没有指定displayId参数， Session会
替SampleWindow选择 DEFAULT_DISPLAY， 也就是手机屏幕 */ final
DisplayContent displayContent = getDisplayContentLocked(displayId); if
(displayContent == null) { return
WindowManagerGlobal.ADD_INVALID_DISPLAY; } // 如果要添加
的窗口是另一个子窗口， 就要求父窗口必须已经存在 // 注意，
attrs.type表示了窗口的类型， attrs.token则表示窗口所隶属的对象 // 对
子窗口来说， attrs.token表示父窗口 if (type >= FIRST_SUB_WINDOW
&&.type <= LAST_SUB_WINDOW) { attachedWindow =
windowForClientLocked(null, attrs.token, false); if (attachedWindow ==
null) { return
WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN; } //在这里
还可以看出WMS要求窗口的层级关系最多为两层 if
(attachedWindow.mAttrs.type >= FIRST_SUB_WINDOW &&
attachedWindow.mAttrs.type <= LAST_SUB_WINDOW) { return

```
WindowManagerGlobal.ADD_BAD_SUBWINDOW_TOKEN; } } boolean
addToken = false; // ② WindowToken出场！根据客户端的attrs.token取出
已注册的WindowToken WindowToken token =
mTokenMap.get(attrs.token); // 下面的if语句块初步揭示了WindowToken
和窗口之间的关系 if (token == null) { // 对于以下几种类型的窗口，必
须通过LayoutParams.token成员为其指定一个已经 // 添加至WMS的
WindowToken if (type >= FIRST_APPLICATION_WINDOW && type <=
LAST_APPLICATION_WINDOW) { return
WindowManagerGlobal.ADD_BAD_APP_TOKEN; } if (type ==
TYPE_INPUT_METHOD) { return
WindowManagerGlobal.ADD_BAD_APP_TOKEN; } if (type ==
TYPE_WALLPAPER) { return
WindowManagerGlobal.ADD_BAD_APP_TOKEN; } if (type ==
TYPE_DREAM) { return
WindowManagerGlobal.ADD_BAD_APP_TOKEN; } // 其他类型的窗口
则不需要事先向WMS添加WindowToken，因为WMS会在这里隐式地创
// 建一个。注意最后一个参数false，这表示此WindowToken由WMS隐
式创建 token = new WindowToken(this, attrs.token, -1, false); addToken =
true; } else if (type >= FIRST_APPLICATION_WINDOW && type <=
LAST_APPLICATION_WINDOW) { // 对于APPLICATION类型的窗
口，要求对应的WindowToken的类型也为APPLICATION // 并且是
```

WindowToken的子类：AppWindowToken

```
AppWindowToken atoken = token.appWindowToken; if (atoken == null) { return WindowManagerImpl.ADD_NOT_APP_TOKEN; } else if (atoken.removed) { return WindowManagerImpl.ADD_APP_EXITING; } if (type==TYPE_APPLICATION_STARTING && atoken.firstWindowDrawn) { return WindowManagerImpl.ADD_STARTING_NOT_NEEDED; } } else if (type == TYPE_INPUT_METHOD) { // 对于其他几种类型的窗口也有类似的要求：窗口类型必须与WindowToken的类型一致 if (token.windowType != TYPE_INPUT_METHOD) { return WindowManagerGlobal.ADD_BAD_APP_TOKEN; } } else if (type == TYPE_WALLPAPER) { if (token.windowType != TYPE_WALLPAPER) { return WindowManagerGlobal.ADD_BAD_APP_TOKEN; } } else if (type == TYPE_DREAM) { if (token.windowType != TYPE_DREAM) { return WindowManagerGlobal.ADD_BAD_APP_TOKEN; } } } // ③ WMS为要添加的窗口创建了一个WindowState对象 // 这个对象维护了一个窗口的所有状态信息 win = new WindowState(this, session, client, token, attachedWindow, seq, attrs, viewVisibility, displayContent); ..... // WindowManagerPolicy出场了。这个函数的调用会调整LayoutParams的一些成员的取值 mPolicy.adjustWindowParamsLw(win.mAttrs); res = mPolicy.prepareAddWindowLw(win, attrs); if (res !=
```

```
 WindowManagerGlobal.ADD_OKAY) { return res; } // 接下来将刚刚隐式  
 创建的WindowToken添加到mTokenMap中。通过这行代码 //读者应该能  
 想到，所有的WindowToken都被放入这个HashTable中 ..... if (addToken)  
 { mTokenMap.put(attrs.token, token); } win.attach(); // 然后将WindowState  
 对象加入mWindowMap中 mWindowMap.put(client.asBinder(), win); // 剩  
 下的代码稍后再分析 ..... } }
```

addWindow () 函数的前段代码展示了三个重要的概念，分别是 WindowToken、WindowState以及DisplayContent。并且在函数开始处对窗口类型的检查判断也初步揭示了它们之间的关系：除子窗口外，添加任何一个窗口都必须指明其所属的WindowToken；窗口在WMS中通过一个WindowState实例进行管理和保管。同时必须在窗口中指明其所属的DisplayContent，以便确定窗口将被显示到哪一个屏幕上。

4.2.1 理解WindowToken

1.WindowToken的意义

为了搞清楚WindowToken的作用是什么，看一下其位于 WindowToken.java中的定义。虽然它没有定义任何函数，但其成员变量的意义却很重要。

·WindowToken将属于同一个应用组件的窗口组织在一起。所谓应用组件可以是Activity、InputMethod、Wallpaper以及Dream。在WMS对窗

口的管理过程中，用WindowToken指代一个应用组件。例如在进行窗口ZOrder排序时，属于同一个WindowToken的窗口会被安排在一起，而且在其中定义的一些属性将会影响所有属于此WindowToken的窗口。这些都表明了属于同一个WindowToken的窗口之间的紧密联系。

·WindowToken具有令牌的作用，是对应用组件的行为进行规范管理的一个手段。WindowToken由应用组件或其管理者负责向WMS声明并持有。应用组件在需要新的窗口时，必须提供WindowToken以表明自己的身份，并且窗口的类型必须与所持有的WindowToken类型一致。从前面的代码可以看到，在创建系统类型的窗口时不需要提供一个有效的Token，WMS会隐式地为其声明一个WindowToken，看起来谁都可以添加一个系统级的窗口。难道Android为了内部使用方便而置安全于不顾吗？非也，addWindow（）函数一开始的mPolicy.checkAddPermission（）的目的就是如此。它要求客户端必须拥有SYSTEM_ALERT_WINDOW或INTERNAL_SYSTEM_WINDOW权限才能创建系统类型的窗口。

2. 向WMS声明WindowToken

既然应用组件在创建一个窗口时必须指定一个有效的WindowToken才行，那么WindowToken究竟该如何声明呢？

在SampleWindow应用中，使用wms.addWindowToken () 函数声明 mToken作为它的令牌，所以在添加窗口时，通过设置lp.token为mToken 向WMS出示，从而获得WMS添加窗口的许可。这说明，只要是一个 Binder对象（随便一个），都可以作为Token向WMS进行声明。对 WMS的客户端来说，Token仅仅是一个Binder对象而已。

为了验证这一点，来看一下addWindowToken的代码，如下所示：

[WindowManagerService.java--

```
>WindowManagerService.addWindowToken() @Override public void  
addWindowToken(IBinder token, int type) { // 需要声明Token的调用者拥  
有MANAGE_APP_TOKENS的权限 if  
(!checkCallingPermission(android.Manifest.permission.MANAGE_APP_T  
OKENS, "addWindowToken()")) { throw new SecurityException("Requires  
MANAGE_APP_TOKENS permission"); } synchronized(mWindowMap) {  
..... // 注意其构造函数的参数与addWindow()中不同，最后一个参数为  
true，表明这个Token // 是显式声明的 wtoken = new WindowToken(this,  
token, type, true); mTokenMap.put(token, wtoken); ..... } }
```

使用addWindowToken () 函数声明Token，将会在WMS中创建一个 WindowToken实例，并添加到mTokenMap中，键值为客户端用于声明 Token的Binder实例。与addWindow () 函数中隐式地创建 WindowToken不同，这里的WindowToken被声明为显式的。隐式与显式

的区别在于，当隐式创建的WindowToken的最后一个窗口被移除后，此WindowToken会被一并从mTokenMap中移除。显式创建的WindowToken只能通过removeWindowToken () 显式地移除。

addWindowToken () 这个函数告诉我们，WindowToken其实有两层含义：

- 对显示组件（客户端）而言的Token，是任意一个Binder的实例，对显示组件（客户端）来说仅仅是一个创建窗口的令牌，没有其他的含义。

- 对WMS而言的WindowToken，这是一个WindowToken类的实例，保存了对应于客户端一侧的Token（Binder实例），并以这个Token为键，存储于mTokenMap中。客户端一侧的Token是否已被声明，取决于其对应的WindowToken是否位于mTokenMap中。



注意

在一般情况下，称显示组件（客户端）一侧Binder的实例为Token，而称WMS一侧的WindowToken对象为WindowToken。但是为了叙述方便，在没有歧义的前提下不会过分仔细地区分这两个概念。

接下来，看一下各种显示组件是如何声明WindowToken的。

(1) Wallpaper和InputMethod的Token

Wallpaper的Token声明在WallpaperManagerService中。参考以下代码：

```
[WallpaperManagerService.java--  
>WallpaperManagerService.bindWallpaperComponentLocked()  
Boolean bindWallpaperComponentLocked(...) { ..... WallpaperConnection  
newConn = new WallpaperConnection(wi, wallpaper); .....  
mIWindowManager.addWindowToken(newConn.mToken,  
WindowManager.LayoutParams.TYPE_WALLPAPER); ..... }
```

WallpaperManagerService是Wallpaper管理器，它负责维护系统已安装的所有Wall-paper并在它们之间进行切换，而这个函数的目的是准备显示一个Wallpaper。newConn.mToken与SampleWindow例子一样，是一个简单的Binder对象。这个Token将在即将显示的Wallpaper被连接时传递给它，之后Wallpaper即可通过这个Token向WMS申请创建绘制壁纸所需的窗口了。



注意

WallpaperManagerService向WMS声明的Token类型为TYPE_WALLPAPER，所以，Wallpaper仅能本地创建TYPE_WALLPAPER类型的窗口。

相应，WallpaperManagerService会在detachWallpaperLocked()函数中取消对Token的声明：

```
[WallpaperManagerService.java-->WallpaperManagerService.detachWallpaperLocked()]
booleandetachWallpaperLocked(WallpaperData wallpaper) { .....
mIWindowManage.removeWindowToken(wallpaper.connection.mToken);
..... }
```

在此之后，如果这个被detach的Wallpaper想再要创建窗口便不再可能了。

WallpaperManagerService使用WindowToken对一个特定的Wallpaper做出如下限制：

- Wallpaper只能创建TYPE_WALLPAPER类型的窗口。
- Wallpaper显示的生命周期由WallpaperManagerService牢牢地控制着。仅有当前的Wallpaper才能创建窗口并显示内容。其他Wallpaper由于没有有效的Token，而无法创建窗口。

InputMethod的Token的来源与Wallpaper类似，其声明位于InputMethodManager-Service的startInputInnerLocked（）函数中，取消声明的位置在InputmethodManagerService的unbindCurrentMethodLocked（）函数。InputMethodManager通过Token限制着每一个InputMethod的窗口类型以及显示生命周期。

（2）Activity的Token

Activity的Token的使用方式与Wallpaper和InputMethod类似，但是其包含更多的内容。毕竟，对于Activity，无论是其组成还是操作都比Wallpaper以及InputMethod复杂得多。对此，WMS专为Activity实现了一个WindowToken的子类：AppWindowToken。

既然AppWindowToken是为Activity服务的，那么其声明自然在ActivityManagerService中。具体位置为ActivityStack.startActivityLocked

()，也就是启动Activity的时候。相关代码如下：

```
[ActivityStack.java-->ActivityStack.startActivityLocked()]
private final void
startActivityLocked(.....) { .....
    mService.mWindowManager.addAppToken(addPos, r.appToken,
    r.task.taskId, r.info.screenOrientation, r.fullscreen); .....
}
```

startActivityLocked () 向WMS声明r.appToken作为此Activity的Token，这个Token是在ActivityRecord的构造函数中创建的。然后在realStartActivityLocked () 中将此Token交付给即将启动的Activity。

```
[ActivityStack.java-->ActivityStack.realStartActivityLocked()]
final
boolean realStartActivityLocked(.....) { .....
    app.thread.scheduleLaunchActivity(new Intent(r.intent), r.appToken,
    System.identityHashCode(r), r.info, new
    Configuration(mService.mConfiguration), r.compat, r.icicle, results,
    newIntents, !andResume, mService.isNextTransitionForward(), profileFile,
    profileFd, profileAutoStop); .....
}
```

启动后的Activity即可使用此Token创建类型为TYPE_APPLICATION的窗口了。

取消Token的声明则位于ActivityStack.removeActivityFromHistoryLocked () 函数中。

Activity的Token在客户端是否和Wallpaper一样，仅仅是一个基本的Binder实例呢？其实不然。看一下r.appToken的定义可以发现，这个Token的类型是IApplicationToken.Stub。其中定义了一系列和窗口相关的通知回调，它们是：

- windowsDrawn ()，当窗口完成初次绘制后通知AMS。
- windowsVisible ()，当窗口可见时通知AMS。
- windowsGone ()，当窗口不可见时通知AMS。
- keyDispatchingTimeout ()，窗口没能按时完成输入事件的处理。这个回调将会导致ANR。
- getKeyDispatchingTimeout ()，从AMS处获取界定ANR的时间。

AMS通过ActivityRecord表示一个Activity。而ActivityRecord的appToken在其构造函数中被创建，所以每个ActivityRecord拥有其各自的appToken。而WMS接受AMS对Token的声明，并为appToken创建了唯一的一个AppWindowToken。因此，这个类型为IApplicationToken的Binder对象appToken粘结了AMS的ActivityRecord与WMS的AppWindowToken，只要给定一个ActivityRecord，都可以通过appToken在WMS中找到一个对应的AppWindowToken，从而使得AMS拥有了操纵Activity的窗口绘制的能力。例如，当AMS认为一个Activity需要被隐

藏时，以Activity对应的ActivityRecord所拥有的appToken作为参数调用WMS的setAppVisibility () 函数。此函数通过appToken找到其对应的AppWindowToken，然后将属于这个Token的所有窗口隐藏。



注意

每当AMS因为某些原因（如启动/结束一个Activity，或将Task移到前台或后台）而调整ActivityRecord在mHistory中的顺序时，都会调用WMS相关的接口移动AppWindowToken在mAppTokens中的顺序，以保证两者的顺序一致。在后面讲解窗口排序规则时会介绍到，AppWindowToken的顺序对窗口的顺序影响非常大。

4.2.2 理解WindowState

从 WindowManagerService.addWindow () 函数的实现中可以看出，当向WMS添加一个窗口时，WMS会为其创建一个WindowState。WindowState表示一个窗口的所有属性，所以它是WMS中事实上的窗口。这些属性将在后面遇到时再介绍。

类似于WindowToken，WindowState在显示组件一侧也有个对应的类型：IWindow.Stub。IWindow.Stub提供了很多与窗口管理相关通知的回调，例如尺寸变化、焦点变化等。

另外，从 WindowManagerService.addWindow () 函数中看到新的WindowState被保存到mWindowMap中，键值为IWindow的Bp端。mWindowMap是整个系统所有窗口的一个全集。



注意

对比一下mTokenMap和mWindowMap。这两个HashMap维护了WMS中最重要的两类数据：WindowToken及WindowState。它们的键都是IBinder，区别是：mTokenMap的键值可能是IApWindowToken的Bp端（使用addAppToken () 进行声明），或者是其他任意一个Binder的Bp端（使用addWindowToken () 进行声明）；而mWindowToken的键值一定是IWindow的Bp端。

关于WindowState的更多细节将在后面的讲述中进行介绍。不过经过上面的分析，不难得到WindowToken和WindowState之间的关系，参考图

4-4。

更具体一些，以一个正在回放视频并弹出两个对话框的Activity为例，
WindowToken与WindowState的意义如图4-5所示。

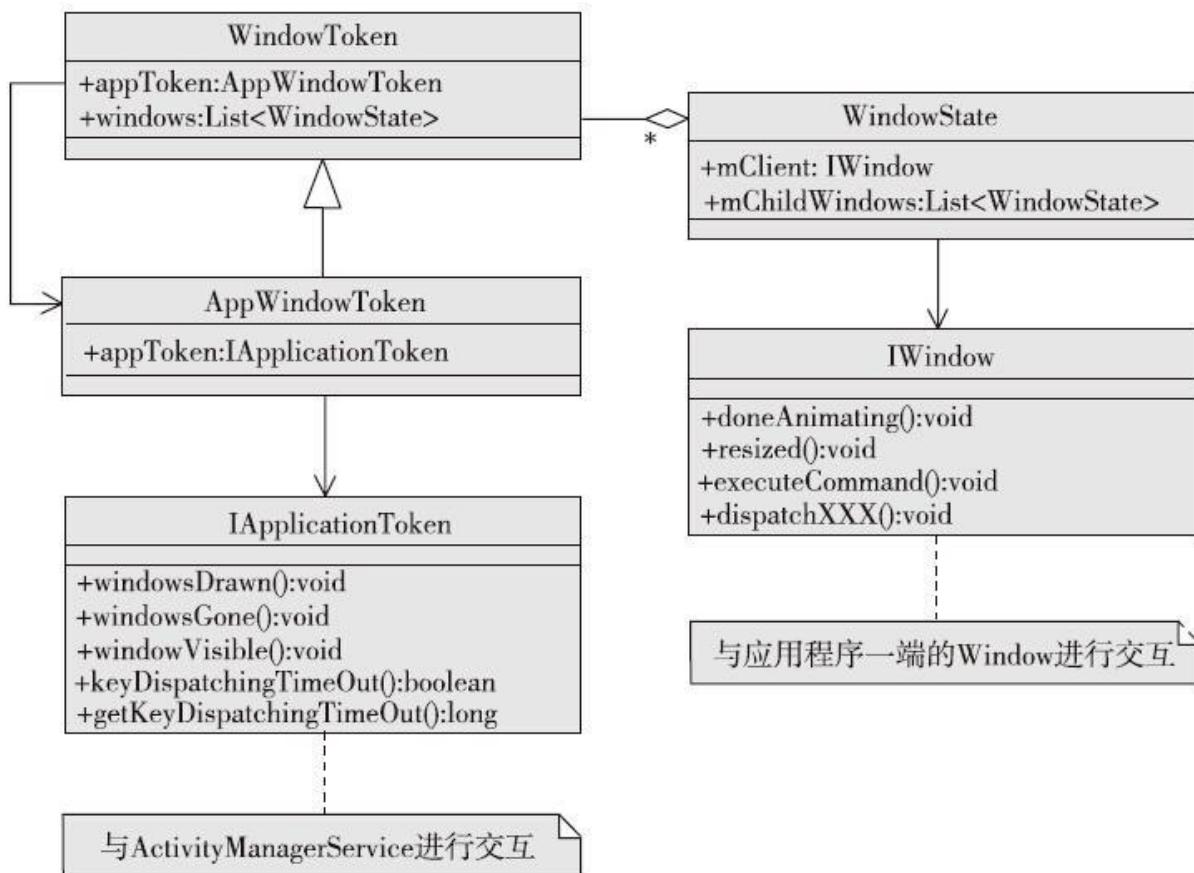


图4-4 WindowToken与WindowState的关系

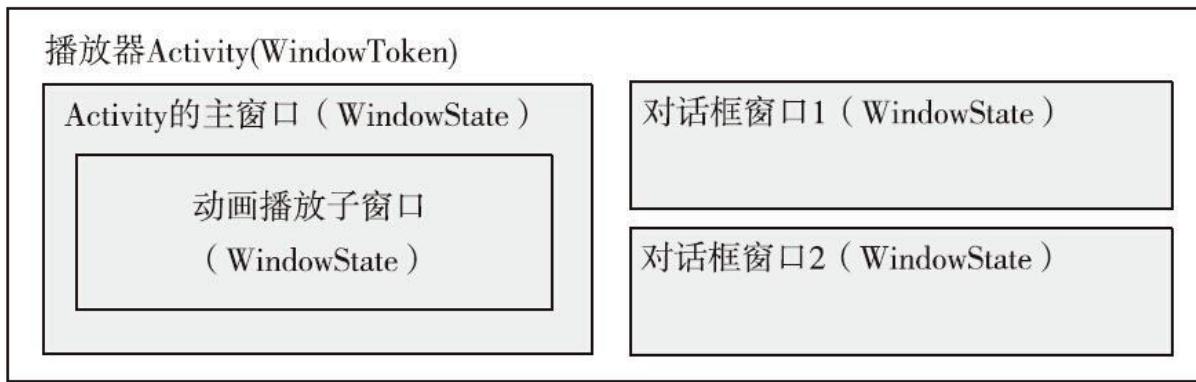


图4-5 WindowState与WindowToken的从属关系

4.2.3 理解DisplayContent

如果说WindowToken按照窗口之间的逻辑关系将其分组，那么DisplayContent则根据窗口的显示位置将其分组。隶属于同一个DisplayContent的窗口将会被显示在同一个屏幕中。每一个DisplayContent都对应这一个唯一的ID，在添加窗口时可以通过指定这个ID决定其将被显示在哪个屏幕中。

DisplayContent是一个非常具有隔离性的一个概念。处于不同DisplayContent的两个窗口在布局、显示顺序以及动画处理上不会产生任何耦合。因此，就这几个方面来说，DisplayContent就像一个孤岛，所有这些操作都可以在其内部独立执行。因此，这些本来属于整个WMS全局性的操作，变成了DisplayContent内部的操作。

4.3 理解窗口的显示次序

在addWindow () 函数的前半部分中，WMS为窗口创建了用于描述窗口状态的WindowState，接下来便会为新建的窗口确定显示次序。手机屏幕是以左上角为原点，向右为X轴方向，向下为Y轴方向的一个二维空间。为了方便管理窗口的显示次序，手机的屏幕被扩展为一个三维空间，即多定义了一个Z轴，其方向为垂直于屏幕表面指向屏幕外。多个窗口依照其前后顺序排布在这个虚拟的Z轴上，因此窗口的显示次序又称为Z序（Zorder）。本节将深入探讨WMS确定窗口显示次序的过程以及其影响因素。

4.3.1 主序、子序和窗口类型

看一下WindowState的构造函数：

[WindowState.java-->WindowState.WindowState()]

```
WindowState(WindowManagerService service, Session s, IWindow c,  
WindowToken token, WindowState attachedWindow, int seq,  
WindowManagerLayoutParams a, int viewVisibility, final DisplayContent  
displayContent) { ..... // 为子窗口分配ZOrder if ((mAttrs.type >=  
FIRST_SUB_WINDOW && mAttrs.type <= LAST_SUB_WINDOW)) { //  
这里的mPolicy就是WindowManagerPolicy mBaseLayer =
```

```
mPolicy.windowTypeToLayerLw( attachedWindow.mAttrs.type) *  
 WindowManagerService.TYPE_LAYER_MULTIPLIER +  
 WindowManagerService.TYPE_LAYER_OFFSET; mSubLayer =  
 mPolicy.subWindowTypeToLayerLw(a.type); ..... } else { // 为普通窗口分  
 配ZOrder mBaseLayer = mPolicy.windowTypeToLayerLw(a.type) *  
 WindowManagerService.TYPE_LAYER_MULTIPLIER +  
 WindowManagerService.TYPE_LAYER_OFFSET; mSubLayer = 0; ..... }  
 ..... }
```

窗口的显示次序由两个成员字段描述：主序mBaseLayer和子序mSubLayer。主序用于描述窗口及其子窗口在所有窗口中的显示位置。而子序则描述了一个子窗口在其兄弟窗口中的显示位置。

·主序越大，则窗口及其子窗口的显示位置相对于其他窗口的位置越靠前。

·子序越大，则子窗口相对于其兄弟窗口的位置越靠前。对父窗口而言，其主序取决于其类型，其子序则保持为0。而子窗口的主序与其父窗口一样，子序则取决于其类型。从上述代码可以看到，主序与子序的分配工作是由WindowManagerPolicy的两个成员函数

windowTypeToLayerLw () 和subWindowTypeToLayerLw () 完成的。

表4-1与表4-2列出了所有可能的窗口类型以及其主序与子序的值。

表4-1 窗口的主序

窗口类型	主序	窗口类型	主序
TYPE_UNIVERSE_BACKGROUND	11 000	TYPE_WALLPAPER	21 000
TYPE_PHONE	31 000	TYPE_SEARCH_BAR	41 000
TYPE_RECENTS_OVERLAY	51 000	TYPE_SYSTEM_DIALOG	51 000
TYPE_TOAST	61 000	TYPE_PRIORITY_PHONE	71 000
TYPE_DREAM	81 000	TYPE_SYSTEM_ALERT	91 000
TYPE_INPUT_METHOD	101 000	TYPE_INPUT_METHOD_DIALOG	111 000
TYPE_KEYGUARD	121 000	TYPE_KEYGUARD_DIALOG	131 000
TYPE_STATUS_BAR_SUB_PANEL	141 000	应用窗口与未知类型的窗口	21 000

表4-2 窗口的子序

子窗口类型	子序
TYPE_APPLICATION_PANEL	1
TYPE_APPLICATION_ATTACHED_DIALOG	1
TYPE_APPLICATION_MEDIA	-2
TYPE_APPLICATION_MEDIA_OVERLAY	-1
TYPE_APPLICATION_SUB_PANEL	2



注意

表4-2中的MEDIA和MEDIA_OVERLAY的子序为负值，这表明它们的显示次序位于其父窗口的后面。这两个类型的子窗口是SurfaceView控件创建的。SurfaceView被实例化后，会向WMS添加一个类型为MEDIA的子窗口，它的父窗口就是承载SurfaceView控件的窗口。这个子窗口的Surface将被用于视频回放、相机预览或游戏绘制。为了不让这个子窗口覆盖住所有的父窗口中承载的其他控件（如拍照按钮、播放器控制按钮等），它必须位于父窗口之后。

从表4-1所描述的主序与窗口类型的对应关系中可以看出，WALLPAPER类型的窗口的主序竟和APPLICATION类型的窗口主序相同，这看似有点不合常理，WALLPAPER不是应该显示在所有Activity之下吗？其实WALLPAPER类型的窗口是一个很不安分的角色，需要在所有的APPLICATION窗口之间跳来跳去。这是因为，有的Activity指定了`android:windowShowWallpaper`为true，则表示窗口要求将用户当前壁纸作为其背景。对WMS来说，最简单的办法就是将WALLPAPER窗口放置到紧邻拥有这个式样的窗口的下方。在这种需求下，为了保证主序决定窗口顺序的原则，WALLPAPER使用了与APPLICATION相同的主序。另外，输入法窗口也是一个很特殊的情况，输入法窗口会选择输入目标窗口，并将自己放置于其上。在本章中不讨论这两个特殊的例子，WALLPAPER的排序规则将在第8章中进行介绍，而输入法的排序则留给读者自行研究。

虽然知道了窗口的主序与子序是如何分配的，不过我们仍然存有疑问：如果有两个相同类型的窗口，那么它们的主序与子序岂不是完全相同？如何确定它们的显示顺序呢？事实上，表4-1和表4-2中所描述的主序和子序仅仅是排序的依据之一，WMS需要根据当前所有同类型窗口的数量为每个窗口计算最终的现实次序。

4.3.2 通过主序与子序确定窗口的次序

回到WMS的addWindow () 函数中，继续往下看：

```
[ WindowManagerService.java-->WindowManagerService.addWindow() ]  
public int addWindow(Session session, IWindow client, int seq,  
 WindowManagerLayoutParams attrs, int viewVisibility, int displayId, Rect  
 outContentInsets, InputChannel outInputChannel) { .....  
  
 synchronized(mWindowMap) { // 在前面的代码中，WMS验证了添加窗  
 口的令牌的有效性，并为新窗口创建了新的WindowState对象 // 新的  
 WindowState对象在其构造函数中根据窗口类型初始化了其主序  
 mBaseLayer和mSubLayer ..... // 接下来，将新的WindowState按照显示  
 次序插入当前DisplayContent的mWindows列表中 // 为了代码结构的清  
 晰，不考虑输入法窗口和壁纸窗口的处理 if (type ==  
 TYPE_INPUT_METHOD) { ..... } else if (type ==  
 TYPE_INPUT_METHOD_DIALOG) { } else { // 将新的WindowState按显  
 示次序插入当前DisplayContent的mWindows列表中 }
```

```
addWindowToListInOrderLocked(win, true); if (type ==  
TYPE_WALLPAPER) { ..... } } ..... // 根据窗口的排序结果，为  
DisplayContent的所有窗口分配最终的显示次序  
assignLayersLocked(displayContent.getWindowList()); ..... } ..... return res;  
}
```

这里有两个关键点：

- addWindowToListInOrderLocked () 将新建的WindowState按照一定的顺序插入当前DisplayContent的mWindows列表中。在分析WMS的重要成员时提到过这个列表。它严格地按照显示顺序存储所有窗口的WindowState。

- assignLayersLocked () 将根据mWindows的存储顺序对所有WindowState的主序和子序进行调整。

接下来分别分析一下这两个函数。

1.addWindowToListInOrderLocked () 分析

addWindowToListInOrderLocked () 的代码很长，不过其排序原则却比较清晰。这里直接给出其处理原则，感兴趣的读者可根据这些原则自行深究相关代码。



注意

再次强调一下，mWindows列表是按照主序与子序的升序进行排序的，所以显示靠前的窗口放在列表靠后的位置，而显示靠后的窗口，则位于列表的前面。也就是说，列表顺序与显示顺序是相反的。这点在阅读代码时要牢记，以免混淆。

在后面的叙述中，非特别强调，所谓的前后都是指显示顺序，而不是在列表的存储顺序。

子窗口的排序规则：子窗口的位置计算是相对父窗口的，并根据其子序进行排序。由于父窗口的子序为0，所以子序为负数的窗口会放置在父窗口的后面，而子序为正数的窗口会放置在父窗口的前面。如果新窗口与现有窗口子序相等，则正数子序的新窗口位于现有窗口的前面，负数子序的新窗口位于现有窗口的后面。

非子窗口的排序则是依据主序进行的，但是其规则较为复杂，分为应用窗口和非应用窗口两种情况。之所以要区别处理应用窗口，是因为所有的应用窗口的初始主序都是21000，并且应用窗口的位置应该与它

所属的应用的其他窗口放在一起。例如应用A显示于应用B的后方，当应用A因为某个动作打开一个新的窗口时，新窗口应该位于应用A其他窗口的前面，但是不得覆盖应用B的窗口。只依据主序进行排序是无法实现这个管理逻辑的，还需要依赖Activity的顺序。在WindowToken一节的讲解中，曾经简单分析了mAppTokens列表的性质，它所保存的AppWindowToken的顺序与AMS中ActivityRecord的顺序时刻保持一致。因此，AppWindowToken在mAppTokens的顺序就是Activity的顺序。

非应用窗口的排序规则：依照主序进行排序，主序高者排在前面，当现有窗口的主序与新窗口相同时，新窗口位于现有窗口的前面。

应用窗口的排序规则：如上所述，同一个应用的窗口的显示位置必须相邻。如果当前应用已有窗口在显示（当前应用的窗口存储在其WindowState.appWindowToken.windows中），新窗口将插入其所属应用其他窗口的前面，但是保证STARTING_WINDOW永远位于最前方，BASE_APPLICATION永远位于最后方。如果新窗口是当前应用的第一个窗口，则参照其他应用的窗口顺序，将新窗口插入位于前面的最后一个应用的最后一个窗口的后方，或者位于后面的第一个应用的最前一个窗口的前方。如果当前没有其他应用的窗口可以参照，则直接根据主序将新窗口插入列表中。

窗口排序的总结如下：

·子窗口依据子序相对于其父窗口进行排序。相同子序的窗体，正子序则越新越靠前，负子序则越新越靠后。

·应用窗口参照本应用其他窗口或相邻应用的窗口进行排序。如果没有任何窗口可以参照，则根据主序进行排序。

·非应用窗口根据主序进行排序。

经过addWindowToListInOrderLocked () 函数的处理之后，当前DisplayContent的窗口列表被插入了一个新的窗口。然后等待assignLayersLocked () 进一步处理。

2.assignLayersLocked分析

assignLayersLocked () 函数将根据每个窗口的主序以及它们在窗口列表中的位置重新计算最终的显示次序mLayer。

[WindowManagerService.java--

```
>WindowManagerService.assignLayersLocked() private final void  
assignLayersLocked(WindowList windows) { int N = windows.size(); int  
curBaseLayer = 0; // curLayer表示当前分配到的Layer序号 int curLayer =  
0; int i; // 遍历列表中所有的窗口，逐个分配显示次序 for (i=0; i < N &&  
w.mIsWallpaper)) { // 为具有相同主序的窗口在curLayer上增加一个偏移  
量，并将curLayer作为最终的显示次序 curLayer +=
```

```
WINDOW_LAYER_MULTIPLIER; w.mLayer = curLayer; } else { // 此窗口拥有不同的主序，直接将主序作为其显示次序并更新curLayer  
curBaseLayer = curLayer = w.mBaseLayer; w.mLayer = curLayer; } // 如果显示次序发生了变化则进行标记 if (w.mLayer != oldLayer) {  
layerChanged = true; anyLayerChanged = true; } ..... } ..... // 向当前DisplayContent的监听者通知显示次序的更新 if (anyLayerChanged) {  
scheduleNotifyWindowLayersChangedIfNeededLocked(  
getDefaultDisplayContentLocked()); } }
```

assignLayersLocked () 的工作原理比较绕，简单来说，如果某个窗口在整个列表中拥有唯一的主序，则该主序就是其最终的显示次序。如果若干个窗口拥有相同的主序（注意，经过 addWindowToListInOrderLocked () 函数处理后，拥有相同主序的窗口都是相邻的），则第i个相同主序的窗口的显示次序为在主序的基础上增加*i**WINDOW_LAYER_MULTIPLIER的偏移。

经过assignLayersLocked () 之后，一个拥有9个窗口的系统的显示次序的信息如表4-3所示。

表4-3 窗口最终的显示次序信息

	窗口 1	窗口 2	窗口 3	窗口 4	窗口 5	窗口 6	窗口 7	窗口 8	窗口 9
主序 mBaseLayer	11 000	11 000	21 000	21 000	21 000	21 000	71 000	71 000	101 000
子序 mSubLayer	0	0	0	-1	0	0	0	0	0
显示次序 mLAYER	11 000	11 005	21 000	21 005	21 010	21 015	71 000	71 005	101 000

在确定最终的显示次序mLayer后，又计算了WindowStateAnimator另一个属性：mAnimLayer。如下所示：

```
[WindowManagerService.java-->assignLayersLocked()]
final
WindowStateAnimator winAnimator = w.mWinAnimator; ..... if
(w.mTargetAppToken != null) { // 输入目标为Activity的输入法窗口，其
mTargetAppToken是其输入目标所属的AppToken
winAnimator.mAnimLayer = w.mLayer +
w.mTargetAppToken.mAppAnimator.animLayerAdjustment; } else if
(w.mAppToken != null) { // 属于一个Activity的窗口
winAnimator.mAnimLayer = w.mLayer +
w.mAppToken.mAppAnimator.animLayerAdjustment; } else {
winAnimator.mAnimLayer = w.mLayer; } .....
```

对绝大多数窗口而言，其对应的WindowStateAnimator的mAnimLayer就是mLayer。而当窗口附属为一个Activity时，mAnimLayer会加入一个来自AppWindowAnimator的矫正：animLayerAdjustment。

WindowStateAnimator和AppWindowAnimator是动画系统中的两员大将，它们负责渲染窗口动画以及最终的Surface显示次序的修改。回顾一下4.1.2节中的WMS的组成结构图，WindowState属于窗口管理体系的类，因此其所保存的mLayer的意义偏向于窗口管理。

WindowStateAnimator/AppWindowAnimator则是动画体系的类，其mAnimLayer的意义偏向于动画，而且由于动画系统维护着窗口的Surface，因此mAnimLayer是Surface的实际显示次序。

在没有动画的情况下，mAnimLayer与mLayer是相等的，而当窗口附属为一个Activity时，则会根据AppTokenAnimator的需要适当地增加一个矫正值。这个矫正值来自AppTokenAnimator所使用的Animation。当Animation要求动画对象的ZOrder必须位于其他对象之上时

(Animation.getZAdjustment () 的返回值为
Animation.ZORDER_TOP) ，这个矫正是一个正数
 WindowManagerService.TYPE_LAYER_OFFSET (1000) ，这个矫正值很大，于是窗口在动画过程中会显示在其他同主序的窗口之上。相反，如果要求ZOrder必须位于其他对象之下时，矫正为-
 WindowManagerService.TYPE_LAYER_OFFSET (-1000) ，于是窗口会显示在其他同主序的窗口之下。在动画完结后，mAnimLayer会被重新赋值为WindowState.mLayer，使得窗口回到其应有的位置。

动画系统的工作原理将在4.5节详细探讨。



注意

矫正值为常数1000，也就出现一个隐藏的bug：当同主序的窗口的数量大于200时，APPLICATION窗口的mLayer值可能超过22000。此时，在对mLayer值为21000的窗口应用矫正后，仍然无法保证动画窗口位于同主序的窗口之上。不过超过200个应用窗口的情况非常少见，而且仅在动画过程中才会出现bug，所以Google貌似也懒得解决这个问题。

4.3.3 更新显示次序到Surface

再回到WMS的addWindow () 函数中，发现再没有可能和显示次序相关的代码了。mAnimLayer是如何发挥自己的作用呢？不要着急，事实上，新建的窗口目前尚无Surface。回顾一下SimpleWindow例子，在执行session.relayout () 后，WMS才为新窗口分配了一块Surface。也就是说，只有执行relayout () 之后才会为新窗口的Surface设置新的显示次序。

为了不中断对显示次序的调查进展，就直接开门见山地告诉大家，设置显示次序到Surface的代码位于

WindowStateAnimator.prepareSurfaceLocked () 函数中，是通过 Surface.setLayer () 完成的。在4.5节会为大家深入揭开WMS动画子系统的面纱。

4.3.4 关于显示次序的小结

这一节讨论了窗口类型对窗口显示次序的影响。窗口根据自己的类型得出其主序及子序，然后addWindowToListInOrderLocked () 根据主序、子序以及其所属的Activity的顺序，按照升序排列在DisplayContent 的mWindows列表中。然后assignLayersLocked () 为mWindows中的所有窗口分配最终的显示次序。之后，WMS的动画系统将最终的显示次序通过Surface.setLayer () 设置进SurfaceFlinger。

4.4 窗口的布局

在本节中将讨论WMS如何对窗口进行布局。窗口布局是WMS的一项重要工作内容，而且其过程非常复杂，所以这将是本章中必须啃也是最难啃的一块骨头。回顾一下SampleWindow这个例子，将窗口添加到WMS后，需要执行IWindowSession.relayout () 函数后才能获得一块可供作画的Surface，并将其放置在SampleWindow例子所指定的位置上。简单来说，IWindowSession.relayout () 函数的作用就在于根据客户端提供的布局参数（LayoutParameters）为窗口重建Surface（如果有必要），并将其放置在屏幕的指定位置。同IWindowSession.add () 函数一样，IWindowSession.relayoutWindow () 函数对应于WMS的relayoutWindow () 函数。

事实上，relayoutWindow () 函数并不是本节的重点，它是用来引出WMS中最重要也是最复杂的一个函数——

performLayoutAndPlaceSurfacesLocked () 。relayoutWindow () 函数修改指定窗口的布局参数，然后performLayoutAndPlaceSurfacesLocked () 遍历所有窗口并对它们进行重新布局。这种牵一发而可能动全身的做法看似效率低下，但是确实很有必要，因为多个窗口之间的布局是相互影响的。

另外，通过这一节的学习与分析，读者将加深对WindowState和WindowToken这两个概念的认识。

再正式开始本节的内容之前，先看一下WMS窗口属性更新的总体过程，如图4-6所示。



图4-6 属性更新的总体过程

图4-6揭示了WMS布局三步走的流程。即将讨论的`reLayoutWindow()`函数属于第一步，除此之外，屏幕旋转等操作也属于第一步的范畴。第二步的布局“子系统”这个说法其实并不太准确，因为这个所谓的子系统只是以`performLayoutAndPlaceSurfaceLocked()`函数为主入口的一组函数的集合。而动画子系统则是WMS中由Choreographer驱动的一系列的WindowAnimator，这一部分将在后续内容中介绍。

接下来，就从relayoutWindow () 开始，揭开WMS窗口布局管理的面纱吧。

4.4.1 从relayoutWindow () 开始

首先我们要理解relayoutWindow () 的参数的意义。WMS的relayoutWindow () 的签名如下：

```
[ WindowManagerService.java-->WindowManagerService.relayoutWindow() ] public int  
    relayoutWindow(Session session, IWindow client, int seq,  
    WindowManager.LayoutParams attrs, int requestedWidth, int  
    requestedHeight, int viewVisibility, int flags, Rect outFrame, Rect  
    outContentInsets, Rect outVisibleInsets, Configuration outConfig, Surface  
    outSurface);
```

参数很多，从名字上可以看出有些参数是返回给调用者的。下面介绍它们的意义。

- session：调用者所在进程的Session实例。
- client：需要进行relayout的窗口。
- seq：一个和状态栏/导航栏可见性相关的序列号，在第6章再进行探讨。

· attrs : 窗口的新布局属性。 `layoutWindow ()` 的主要目的就是根据 attrs所提供的布局参数重新布局一个窗口。客户端可以通过 `layoutWindow ()` 函数改变 attrs 中所定义的几乎所有布局属性。但是唯独无法改变窗口类型。

· `requestedWidth` 与 `requestedHeight` : 客户端所要求的窗口尺寸。在重新布局的过程中，WMS会尽量将窗口的尺寸布局为客户端所要求的大小。

· `viewVisibility` : 窗口的可见性。

· `flags` : 定义一些布局行为。

· `outFrame` : 由 `layoutWindow ()` 函数返回给调用者的一个 `Rect` 类型的实例。它保存了窗口被重新布局后的位置与大小。

· `outContentInsets` 与 `outVisibleInsets` : 这两个参数表示了窗口可以绘制内容的矩形边界与可视矩形边界在四个方向上到 `mFrame` 的像素差。

· `outConfiguration` : 重新布局后，WMS为此窗口计算出的 Configuration。

· `outSurface` : 用来接收 WMS 为此窗口分配的 Surface。窗口的第一次 `layout` 完成后就可以通过它在窗口中进行绘图了。卷 I 的 Surface 系统一章中曾经介绍了这个参数的具体细节。本节不再赘述。

[WindowManagerService.java--

```
>WindowManagerService.relayoutWindow() public int  
relayoutWindow(Session session, IWindow client, int seq,  
WindowManager.LayoutParams attrs, int requestedWidth, int  
requestedHeight, int viewVisibility, int flags, Rect outFrame, Rect  
outContentInsets, Rect outVisibleInsets, Configuration outConfig, Surface  
outSurface) { boolean toBeDisplayed = false; boolean inTouchMode;  
boolean configChanged; boolean surfaceChanged = false; boolean  
animating; ..... //先做一些权限相关的检查 /* 接下来的操作将在锁住  
mWindowMap的情况下完成。在WMS中，几乎所有的操作都是在锁住  
mWindowMap 的情况下完成的。在后面分析  
performLayoutAndPlaceSurfaceLocked()函数时会发现，WMS对 窗口的  
操作复杂而漫长，为了防止线程间竞态发生，WMS统一使用  
mWindowMap对所有窗口操作进行同步 */ synchronized(mWindowMap)  
{ // 从mWindowMap中获取需要进行relayout的WindowState。  
relayoutWindow()将根据传入 // 的参数更新WindowState的属性  
WindowState win = windowForClientLocked(session, client, false); ..... /*  
① 接下来长长的一段代码用来根据用户传入的参数更新WindowState对  
象的相关属性。它们是： mRequestedWidth/Height(客户端要求的窗口  
尺寸) mSystemUiVisibility(状态栏与导航栏的可见性)、 mAlpha(窗口透  
明度)等 当然，保存了几乎所有窗口属性的mAttrs也被更新了*/
```



注意

`relayoutWindow ()` 基本上可以为窗口更新所有在`LayoutParams`类中定义的属性，但是窗口类型是一个例外。窗口类型必须在添加窗口时加以指定，并且不允许再做更改。

```
..... // ② 根据Window的可见性更新或创建Surface及启动动画效果 // 因为这部分内容和布局关系不大，所以简单带过 if (viewVisibility == View.VISIBLE && (win.mAppToken == null || !win.mAppToken.clientHidden)) { /* 对于处于可见状态的窗口主要有以下处理： 1.如果窗口没有Surface，则为其创建一块Surface 2.如果客户端改变了窗口的色彩格式（由LayoutParams.format指定）发生了变化则为其 重新创建一块指定格式的Surface 3.如果窗口尚未被显示，并且窗口的客户端已经完成了绘制，则为其启动一个淡入动画 */ ..... } else { /* 对于处于非可见状态下的窗口，主要的处理如下： 1.标记WindowState的mExiting属性为true 2.如果窗口目前正被显示，则为其启动一个淡出动画 3.释放客户端所持有的Surface对象，自此之后，客户端无法再更新窗口的内容*/ ..... } // ③ 接下来更新窗口焦点、壁纸可
```

见性以及屏幕旋转等 /* 更新焦点窗口，因为窗口的显示与退出，或者窗口与焦点相关的flag发生了变化 (例如FLAG_NOT_FOCUSABLE) ，都会要求WM S 重新计算焦点窗口。焦点相关的内容会在 第5章中详细讨论 */ if (focusMayChange) { } //焦点窗口的变化可能会引发输入法窗口的变化，所以需要对输入法进行更新 if (imMayMove) { } /* 窗口有一个flag为FLAG_SHOW_WALLPAPER。意思就是此窗口要求使用系统当前的壁纸作为其 背景，一个典型的例子就是launcher和home。这要求Wallpaper窗口的显示次序紧临此窗口之下。所以，在这里需要更新Wallpaper窗口在WindowList中的次序。其实现原理与上一节所分 析的addWindowToListInOrderLocked()非常相近 */ if (wallpaperMayMove) { } /* 回顾在4.3节所讨论的内容，一旦WindowList中的次序发生变化后，需要调用assignLayersLocked() 函数为所有窗口重新计算最终的mLayer，以确定其显示次序 */ if (assignLayers) { assignLayersLocked(win.getWindowList()); } /* 新的窗口被显示，或者被隐藏，意味着屏幕的旋转状态可能会发生变化。因为窗口所属的Activity 可能指定了必须工作在横屏状态或者竖屏状态。所以此时需要根据窗口的要求，调整屏幕的旋转状态。本章并不打算介绍屏幕旋转的内容，屏幕旋转不过是对布局系统与动画系统的一个运用。读者在 深入理解布局系统与动画系统的工作原理之后可以自行研究 */ configChanged = updateOrientationFromAppTokensLocked(false); //将窗口所在的DisplayContent标记为需要进行重新布局。这一个赋值看

似简单，其实很重要。 // 因为如果不进行这个标记，哪怕
DisplayContent中的窗口变化得天翻地覆，也不会对其重新布局
win.mDisplayContent.layoutNeeded = true; /* ④ 神奇的
performLayoutAndPlaceSurfacesLocked()登场! 它将遍历所有
DisplayContent 的所有窗口，为它们计算布局尺寸，并将布局尺寸设置
给它们的Surface */ performLayoutAndPlaceSurfacesLocked(); /* ⑤ 返回布
局结果。完成performLayoutAndPlaceSurfacesLocked()的调用之后，
WMS便根据 调用参数的要求完成窗口的布局，现在是将布局结果返回
给调用者的时候了 */ outFrame.set(win.mCompatFrame);
outContentInsets.set(win.mContentInsets);
outVisibleInsets.set(win.mVisibleInsets); } // 向AMS更新
Configuration，因为屏幕可能发生旋转 if (configChanged) {
sendNewConfiguration(); } return; }

这段代码稍稍有点长，下面总结一下它的工作内容：

- 根据参数更新窗口的WindowState对象的相应属性，这些属性是后续对
其进行布局的依据。
- 处理窗口的显示与退出。这里主要是一些涉及Surface的创建／销毁，
以及动画相关的操作。这不是本节讨论的重点。

·更新和窗口相关的其他机制，例如焦点、输入法、壁纸以及屏幕旋转等。

·调用performLayoutAndPlaceSurfaceLocked () 函数进行布局。

·将布局结果返回给relayoutWindow () 的返回者。

之所以使用relayoutWindow () 作为布局的切入点，一是因为它是WMS提供的最常用的API之一，二是它体现了WMS窗口相关操作的一个通用做法：

·修改一些属性（窗口属性、屏幕属性、焦点属性等）。

·标记相关的DisplayContent为relayoutNeeded。调用performLayoutAndPlaceSurfaceLocked () 进行全局布局。

·对布局结果进行进一步的操作。

通过这个通用做法可以对performLayoutAndPlaceSurfaceLocked () 函数拥有一个感性认识。简单地说，窗口的属性可以分为两类：一类是布局控制属性，例如mRequestedWidth/Height、mAttrs等，由客户端根据需要进行设置，用来向WMS表达它所期望的窗口布局；另一类是布局结果属性，例如mFrame、mContentInsets、mVisibleInsets等，它们是经过布局过程计算出来的，直接影响到窗口的实际布局与显示，客户端无法直接干预这些属性，只能被迫接受这些布局结果。

`performLayoutAndPlaceSurfaceLocked ()` 函数就是通过布局控制属性计算布局结果属性这一过程的场所。



提示

客户端只能被迫接受窗口的布局结果，是因为客户端调用 `relayoutWindow ()` 时所要求的尺寸和位置参数和最终的布局结果有出入。研究一下Android控件系统的核心类ViewRootImpl的实现就可以发现，它作为控件系统到窗口系统的桥梁，采用了协商式的函数来绘制控件。在窗口显示之前，ViewRootImpl首先对整个控件树进行尺寸测量，得到一个理想尺寸，并将这个理想尺寸作为`relayoutWindow ()` 的 `requestedWidth/Height` 参数交给WMS进行布局。布局之后，ViewRootImpl会根据WMS的实际布局结果（`frame`, `contentInsets`, `visibleInsets`）重新对控件树进行一次测量，得到最终尺寸，然后进行绘制。

有兴趣的读者可以做一个试验，在`relayoutWindow ()` 函数最后修改 `outFrame` 的 `bottom` 为原来的 `1/2`，看一下是什么样的结果。

4.4.2 布局操作的外围代码分析

接下来开始剖析WMS的布局过程的征途吧。当然是从
performLayoutAndPlaceSurfacesLocked () 开始。

[WindowManagerService.java--

```
>WindowManagerService.performLayoutAndPlaceSurfaceLocked() private
final void performLayoutAndPlaceSurfacesLocked() { int loopCount = 6; do
{ mTraversalScheduled = false;
performLayoutAndPlaceSurfacesLockedLoop();
mH.removeMessages(H.DO_TRAVERSAL); loopCount--; } while
(mTraversalScheduled&&loopCount > 0); }
```

实现比较简单，很明显可以看出，此函数中有意义的事情都是在
performLayoutAndPlace-SurfacesLockedLoop () 中完成的。

虽然这个函数只有短短的7行，但是却包含了一个乍看上去有点不明所
以的do-while循环。它的循环条件是mTraversalScheduled为true并且
loopCount大于0。为了搞清楚这个循环的目的所在，首先要找到
mTraversalScheduled变量的意义何在。原来，修改这个变量的只有
WMS为外界提供的一个名为requestTraversal () 的接口函数。其实现
如下：

[WindowManagerService.java--

```
>WindowManagerService.requestTraversal() public void requestTraversal()  
{ synchronized (mWindowMap) { requestTraversalLocked(); } }
```

[WindowManagerService.java--

```
>WindowManagerService.requestTraversalLocked() void  
requestTraversalLocked() { if (!mTraversalScheduled) { // 设置为true。也  
就是说， performLayoutAndPlaceSurfaceLocked()要多循环一次了  
mTraversalScheduled = true; // H.DO_TRAVERSAL消息的处理就是调用  
performLayoutAndPlaceSurfaceLocked()  
mH.sendEmptyMessage(H.DO_TRAVERSAL); } }
```

这个实现自WindowManagerFuncs接口的函数向外界提供了一个触发WMS立刻进行重新布局的手段。注意，requestTraversalLocked () 是在被mWindowMap这个锁的保护下调用的，这与

performLayoutAndPlaceSurfaceLocked () 是一样的。也就是说外部线程不可能影响这个循环的判定条件。那就只剩下一个可能：

performLayoutAndPlaceSurfacesLockedLoop () 函数中会调用requestTraversalLocked () 。现在就到这个函数内部去求证一下吧。

[WindowManagerService--

```
>WindowManagerService.performLayoutAndPlaceSurfacesLockedLoop()  
private final void performLayoutAndPlaceSurfacesLockedLoop() { // 首先
```

是省略掉的安全性检查。不过这个函数中有一个特殊要求需要注意：该函数禁止递归调用 /* 在正式开始重新布局之前，先删除所谓的“僵尸窗口”，并释放它们所持有的Surface。该操作其实与 Java GC很类似。僵尸窗口是指其持有者已经异常退出或者处于隐藏状态却拥有Surface的窗口。当Surface的操作因为内存原因而失败时，WMS会将所有僵尸窗口收集到mForceRemoves列表中，然后在这个位置（也就是重新布局开始之前）统一清理。关于这个过程的详细内容，读者可以参考reclaimSomeSurfaceMemoryLocked()函数的实现与使用 */ boolean recoveringMemory = false; try { if (mForceRemoves != null) { // 标记此次布局已经完成内存回收了，此次布局再发生Surface操作异常时，不会再尝试回收 recoveringMemory = true; for (int i=0; i

这个函数仍不算复杂。主要的布局逻辑应该位于performLayoutAndPlaceSurfacesLockedIn-ner () 中。需要注意的地方应该就是requestTraversalLocked () 的两个调用条件了。

第一个条件是needsLayout () 函数的返回值。还记得在分析relayoutWindow () 时所提到的DisplayContent.mLayoutNeeded字段吗？needsLayout () 函数遍历所有DisplayContent并检查它们的mLayoutNeeded字段的值。只要有一个DisplayContent需要重新布局，此函数就会返回true。这说明一个问题，就像罗马不是一天建成的一样，布局也不是一遍就能完成的！

至于第二个条件，`mLayoutRepeatCount<6`，则表示最多连续重新布局6次，这和`performLayoutAndPlaceSurfaceLocked ()`中的循环条件有所重复。这说明了一个更严重的问题，布局有可能重复6次也无法完成，为了防止调用者过度等待，以致需要增加一个6次的限制。

到这里，我们对布局的外围有了一个简单认识，如图4-7所示。

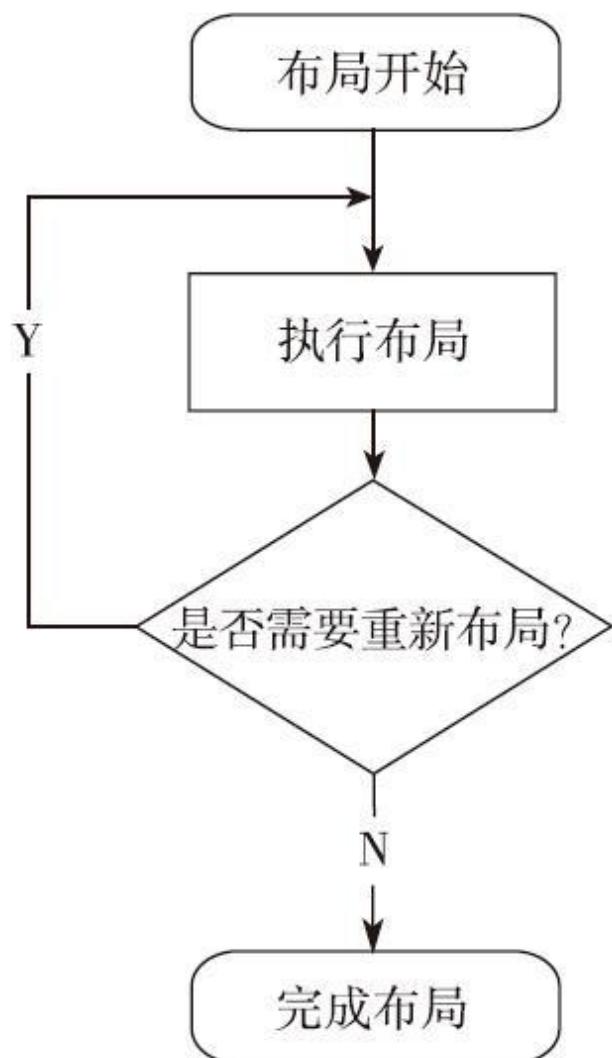


图4-7 布局的总体流程

4.4.3 初探performLayoutAndPlaceSurfacesLockedInner ()

接下来要继续深入WMS的布局过程。performLayoutAndPlaceSurfacesLockedInner () 函数不只名字长，其实现更长，有近600行之多。而且Android 4.2已经通过提炼一部分代码到独立的函数中为其减肥。为了防止在研究这个函数的过程中迷路，先向大家提供将这个函数提炼后的伪代码：

```
[WindowManagerService.java-->WindowManagerService.performLayoutAndPlaceSurfacesLockedInner()]
private void performLayoutAndPlaceSurfacesLockedInner() { 布局前的预
    处理; 遍历所有DisplayContent { 遍历DisplayContent下的所有窗口 { 对
        窗口进行布局; } 对布局结果进行检查, 是否有必要重新对
        DisplayContent执行布局; 对DisplayContent的布局后处理; } 完成布局
    后的策略处理; }
```

可以看出，这个函数的整体逻辑是比较清晰的。不过由于它的全能性，每一个步骤的内容都相当繁杂。这一点从其注释中可以看出：Something has changed ! Let's make it correct now ! 不用担心，我们将一步一个脚印地探索这个函数的实现。

4.4.4 布局的前期处理

[WindowManagerService.java-->

WindowManagerService.performLayoutAndPlaceSurfaceLockedInner()]
private final void performLayoutAndPlaceSurfacesLockedInner(boolean
recoveringMemory) { // ① 首先更新处于焦点状态的窗口 if
(mFocusMayChange) { mFocusMayChange = false;
updateFocusedWindowLocked(UPDATE_FOCUS_WILL_PLACE_SURFACE,
false /*updateInputWindows*/); } // 将所有处于退出状态的Tokens标
记为不可见 for (i=mExitingTokens.size()-1; i>=0; i--) {
mExitingTokens.get(i).hasVisible = false; } for
(i=mExitingAppTokens.size()-1; i>=0; i--) {
mExitingAppTokens.get(i).hasVisible = false; } /* ② 初始化mInnerFields中
的状态变量。客户端可以通过窗口的LayoutParams中的一些属性 或
flag来指明屏幕的亮度、按键背光的亮度、是否保持屏幕唤醒状态以及
输入事件的超时时间。这些 设置都可以在窗口的LayoutParams中找到
与之对应的属性。 WMS 使用 mInnerFields集合了这些状态，在布局过
程中， WMS会遍历所有窗口以查找这些设置，并将这些设置保存到
mInnerFields中。在讨论DisplayContent的布局后处理时将作深入探讨*/
// mHoldScreen是一个Session类型的变量，保存了要求保持屏幕唤醒状
态的窗口所在的进程 mInnerFields.mHoldScreen = null; //取值范围从0到
1的一个float类型变量，保存了完成布局后的屏幕亮度
mInnerFields.mScreenBrightness = -1; //与mScreenBrightness一样的取值

范围，保存了完成布局后的键盘背光亮度

mInnerFields.mButtonBrightness = -1; //以毫秒为单位，表示了输入事件在此窗口上发生ANR的时间 mInnerFields.mUserActivityTimeout = -1; //这个属性表示了当前DisplayContent显示内容的类别。在本章中不作讨论 mInnerFields.mDisplayHasContent = LayoutFields.DISPLAY_CONTENT_UNKNOWN; // ③ 递增布局序列号 mTransactionSequence++; // 获取手机屏幕的宽高尺寸，这个尺寸将用来布局水印和StrictMode的红色警告框 final DisplayContent defaultDisplay = getDefaultDisplayContentLocked(); final DisplayInfo defaultInfo = defaultDisplay.getDisplayInfo(); final int defaultDw = defaultInfo.logicalWidth; final int defaultDh = defaultInfo.logicalHeight; Surface.openTransaction(); try { // ④ 布局水印和StrictMode警告框 if (mWatermark != null) { mWatermark.positionSurface(defaultDw, defaultDh); } if (mStrictModeFlash != null) { mStrictModeFlash.positionSurface(defaultDw, defaultDh); } // 接下来的两个变量将在后续遇到时再作介绍。在这里，它们都被设置为假 boolean focusDisplayed = false; boolean updateAllDrawn = false; 遍历所有 DisplayContent { 遍历DisplayContent下的所有窗口 { 对窗口进行布局； } 对布局结果进行检查，是否有必要重新对DisplayContent执行布局； 对DisplayContent的布局后处理； } } catch (Exception e) { } finally { Surface.closeTransaction(); } 完成布局后的策略处理； }

在布局的前期，主要工作有：

- 如果有必要，计算新的焦点窗口。第一次见到 updateFocusedWindowLocked () 函数是在layoutWindow () 中，如果客户端要求layout的窗口的可见性发生变化，则重新计算焦点窗口。
- 初始化mInnerFields中的一部分字段， mInnerFields是LayoutFields类的一个实例。事实上， LayoutFields类本身并没有什么实际意义，不过是把一些布局相关的状态变量组合到一起而已。在后面使用时再对其进行说明。
- 递增布局序号mTransactionSequence。 WMS每进行一次布局都会导致序号递增。 AppWindowToken中也保存了一个相应的布局序号，在布局的过程中， WMS通过对比这两个序号的值以确定AppWindowToken的布局状态是否最新。
- 布局水印和StrictMode警告框。水印用以在屏幕上显示一段固定的信息，而StrictMode警告框则在任何一个应用或服务发生违例操作时在屏幕上闪烁一个红色方框。它们其实是两块Surface，而它们的显示次序分别为1000000和1000001，高于任何一个窗口的显示次序，所以它们将显示在所有窗口之上。它们的布局非常简单：占据整个显示画面。

布局的前期处理看起来没有什么复杂的。接下来开始对每个DisplayContent进行布局。

4.4.5 布局DisplayContent

需要再次强调一下，Android 4.2之后将支持多块屏幕输出（目前第二块屏幕是WiFi-Display设备）。而一个DisplayContent则用来描述一块屏幕。在performLayoutAndPlaceSurfacesLocked () 函数的后续代码中，将遍历系统中所有的DisplayContent，并分别对它们各自所拥有的窗口进行布局。这一部分内容是performLayoutAndPlaceSurfacesLockedInner () 函数的核心所在。

[WindowManagerService.java--

```
>WindowManagerService.performLayoutAndPlaceSurfaceLockedInner()]\nprivate final void performLayoutAndPlaceSurfacesLockedInner(boolean\nrecoveringMemory) { // 布局的前期处理 ..... try { DisplayContentsIterator\niterator = new DisplayContentsIterator(); /* 遍历所有的DisplayContent。注意，WMS使用了一个内部类DisplayContentsIterator对\nmDisplayContents列表进行遍历 */ while (iterator.hasNext()) { // 首先，提取displayContent的属性，用于后续的布局操作。比较重要的有 //\nwindows、dw/dh、innerDw/innerDh、isDefaultDisplay等 final\nDisplayContent displayContent = iterator.next(); // windows列表中保存了\ndisplayContent所拥有的所有窗口 WindowList windows =
```

```
displayContent.getWindowList(); DisplayInfo displayInfo =  
displayContent.getDisplayInfo(); final int displayId =  
displayContent.getDisplayId(); /* dw/dh存储了当前DisplayContent所描述  
的显示屏的逻辑尺寸。注意这时逻辑尺寸与 物理尺寸是有区别的。例  
如当前DisplayContent正在模拟一个低分辨率的显示屏时，则获取 为模  
拟显示屏的尺寸，而非实际物理尺寸 */ final int dw =  
displayInfo.logicalWidth; final int dh = displayInfo.logicalHeight; /*  
innerDw/innerDh 描述了当前显示屏用于显示应用程序的区域尺寸，就  
是逻辑尺寸减去系统 装饰的尺寸。所谓的系统装饰包括状态栏、导航  
栏等 */ final int innerDw = displayInfo.appWidth; final int innerDh =  
displayInfo.appHeight; /* 虽然isDefaultDisplay仅仅是一个布尔变量，不  
过其作用很大。Default display 是指屏幕的DisplayContent。与其他屏幕  
不同的是，手机屏幕拥有状态栏、导航条，并 应输入事件，而其他的  
屏幕则没有这些特性。因此在布局的过程中，必须对此加以区分 */  
final boolean isDefaultDisplay = (displayId == Display.DEFAULT_  
DISPLAY); ..... /* ① 接下来是一个do-while循环。是否似曾相识？读者  
可能想到了，DisplayContent的 布局也是需要进行多次尝试的。循环的  
条件就是DisplayContent的pendingLayoutChanges 段值不为0。完成一次  
布局后会检查布局结果，并通过设置pendingLayoutChanges决定是 否需  
要重新布局 */ int repeats = 0; do { repeats++; // 限制尝试次数在 6 次以  
if (repeats > 6) { displayContent.layoutNeeded = false; break; } // 接下来是
```

处理pendingLayoutChanges字段的代码。在真正开始窗口的布局前，需要先 // 根据pendingLayoutChanges的值做相应的处理。在这里暂且忽略这些代码，等 // 清楚pendingLayoutChanges的取值之后再做分析 /*接下来调用performLayoutLockedInner()函数。注意其名称，与我们正在分析的函数相比少了AndPlaceSurface字样。它完成了对当前DisplaContent的所有窗口的布局工作，也将是后续分析的重点。注意，当repeat次数大于4后，将不再执行此函数 */ if (repeats < 4) {
 performLayoutLockedInner(displayContent, repeats == 1, false
 /*updateInputWindows*/); } // 清空pendingLayoutChanges字段
 displayContent.pendingLayoutChanges = 0; // 由PhoneWindowManager检查布局，并将结果保存到pendingLayoutChanges中 if (isDefaultDisplay) {
 mPolicy.beginPostLayoutPolicyLw(dw, dh); for (i = windows.size() - 1; i >= 0; i--) { WindowState w = windows.get(i); if (w.mHasSurface) {
 mPolicy.applyPostLayoutPolicyLw(w, w.mAttrs); } }
 displayContent.pendingLayoutChanges |=
 mPolicy.finishPostLayoutPolicyLw(); } // 如果PhoneWindowManager认为布局结果仍需进行调整，则重新再来一遍 } while
(displayContent.pendingLayoutChanges != 0); // ② 对DisplayContent的布局后处理，这部分稍候再作考虑 } // 对DisplayContent的布局到此结束 } catch (Exception e) { } finally { Surface.closeTransaction(); } // 完成布局后的策略处理 }

整理一下目前的思路。布局DisplayContent整体上分为两个部分：

- 布局循环。这个do-while循环的主要工作是对DisplayContent所拥有的窗口进行布局，其工作侧重于PerformLayout。
- 布局后处理。后处理用于根据布局结果设置Surface参数，应用一些动画效果等，其工作侧重于PlaceSurfaces。

1.深入理解窗口布局的原理

接着上面的讨论，深入研究第一部分的布局循环来探究WMS是如何完成窗口布局的。仔细归纳一下其循环体，可以发现其明显分为三个阶段：

- pendingLayoutChanges处理阶段。这个阶段先不讨论。当学习了结果检查阶段后再回头探讨这个阶段。
- 布局阶段。主要内容就是performLayoutLockedInner () 函数。这个函数将对Display-Content下的所有窗口进行布局。
- 结果检查阶段。在所有窗口的布局完成后，通过一些状态量的检查，决定是否重做这三个阶段的工作。其检查内容主要是状态栏、导航栏可见性是否与顶层窗口的属性冲突，是否需要解除锁屏状态等。

这个循环不断地对DisplayContent中的所有窗口进行布局操作，并处理pending-LayoutChanges，直到WMP的finishPostLayoutLw () 认为当前布局不需要再做任何改动。如果在循环4次之后仍无法满足finishPostLayoutLw () 的要求，则不再尝试对窗口进行布局，仅处理pendingLayoutChanges。如果6次尝试之后仍然无法满足要求，则放弃继续尝试。

接下来先深入学习布局阶段的工作原理。通过这一节的学习，希望读者能够了解WMS如何计算每一个窗口的位置与尺寸。

(1) 初识performLayoutLockedInner ()

我们首先来看performLayoutLockedInner () 是如何对窗口完成布局的。参考其代码：

```
[WindowManagerService.java-->WindowManagerService.performlayoutLockedInner()] private final void performLayoutLockedInner(final DisplayContent displayContent, boolean initial, boolean updateInputWindows) { // 只有displayContent的 layoutNeeded为真时才会进行布局 if (!displayContent.layoutNeeded) { return; } displayContent.layoutNeeded = false; // 获取displayContent所拥有的所有Window WindowList windows = displayContent.getWindowList(); // 确定是否是手机屏幕 boolean isDefaultDisplay =
```

```
displayContent.isDefaultDisplay; // 获取当前displayContent的尺寸信息  
DisplayInfo displayInfo = displayContent.getDisplayInfo(); final int dw =  
displayInfo.logicalWidth; final int dh = displayInfo.logicalHeight; ..... // ①  
通知WindowPolicyManager，使其为即将开始的布局操作做准备  
mPolicy.beginLayoutLw(isDefaultDisplay, dw, dh, mRotation); ..... //递增  
mLayoutSeq int seq = mLayoutSeq+1; if (seq < 0) seq = 0; mLayoutSeq =  
seq; // 这个局部变量用来指示第一个非顶级窗口在列表中所处的位置。  
用来节省一部分查找时间 int topAttached = -1; // ② 对所有的顶级窗口进  
行布局 for (i = N-1; i >= 0; i--) { final WindowState win = windows.get(i);  
..... //为了节省篇幅，这里省略了判断条件。其判断目的是节省布局的  
时间开销，使一些不可见的窗口 //不参与布局 if (...) { if  
(!win.mLayoutAttached) { // 在本次循环中仅布局顶级窗口 // 调用窗口的  
prelayout()函数，使其做好被布局的准备 win.prelayout(); // 调用  
 WindowManagerPolicy的layoutWindowLw()函数对窗口进行布局  
mPolicy.layoutWindowLw(win, win.mAttrs, null); // 更新窗口的布局版本  
号 win.mLayoutSeq = seq; } else { // 如果当前窗口不是顶级窗口，则记  
录下其位置。目的是在后面对非顶级窗口进行布局时 // 直接从这个位  
置开始，从而省略一部分时间 if (topAttached < 0) topAttached = i; } }  
..... } ..... // ③ 接着对所有的非顶级窗口进行布局 for (i = topAttached; i  
>= 0; i--) { // 这部分内容与顶级窗口布局几乎完全一致，读者可自行对  
比研究 } // 布局完成后，窗口的尺寸和位置可能发生变化，此时，需要
```

输入系统更新窗口的状态

```
mInputMonitor.setUpdateInputWindowsNeededLw(); if  
(updateInputWindows) { mInputMonitor.updateInputWindowsLw(false  
/*force*/); } // ④ 通知WindowManagerPolicy，本次布局已完成。可以对  
布局过程中所使用的资源进行清理 mPolicy.finishLayoutLw(); }
```

可以看出，窗口布局过程的规律性还是很强的。

- 在进行布局前，首先通过执行WindowManagerPolicy.beginLayoutLw() 通知WMP为即将开始的布局进行准备。这个准备过程就是通过屏幕的尺寸、状态栏/导航栏的可见性、屏幕旋转状态等因素来计算布局所使用的参数。
- 首先对所有顶级窗口进行布局。这个布局过程也分为两步，分别为WindowState.prelayout() 和WindowManagerPolicy.layoutWindowLw()。其工作就是使用上一步所计算出的布局参数计算出窗口的位置属性，并保存在WindowState中。
- 之后对所有子窗口进行布局，其布局过程与顶级窗口一致。之所以要将顶级窗口与子窗口分开进行布局，是因为子窗口的布局依赖于其父窗口的布局结果。
- 最后，执行WindowManagerPolicy.finishLayoutLw()。告知WMP本次布局已完成，可以清理布局过程中所使用过的资源。

可以看出，在窗口布局过程中，WindowManagerPolicy发挥了近乎决定性的作用。在Android 4.2中，WindowManagerPolicy是由PhoneWindowManager实现的，所以上述的关键函数将以PhoneWindowManager的实现为基础进行讨论。

接下来将深入研究这三个阶段的工作原理。

(2) 窗口布局的8个准绳

在PhoneWindowManager.beginLayoutLw () 中究竟为布局准备了什么参数呢？简单来说，这些参数描述了屏幕上的8个矩形区域，而这8个区域构成了PhoneWindowManager布局窗口的准绳。由于beginLayoutLw () 函数内容相对琐碎，因此下面直接给描述这8个矩形区域的参数的意义以及其计算依据。建议感兴趣的读者在读完本节后可以仔细研究一下这个函数，以加深理解。

- mUnrestrictedScreenLeft/Top/Width/Height：描述了整个屏幕的逻辑显示区域，也就是 (0, 0-dw, dh) 。
- mRestrictedScreenLeft/Top/Width/Height：描述了屏幕中导航栏之后的区域。这个区域会受到导航栏的可见性影响。当导航栏不可见时，这块区域与Unrestricted区域等价。

UnrestrictedScreen区域与Restricted区域在窗口布局的过程中常常被用作布局容器。布局容器可以用来应用窗口的对齐方式，或者对窗口的尺寸及位置加以限制。

·mStableFullscreenLeft/Top/Right/Bottom：描述了整个屏幕的逻辑显示区域，和Unrestricted区域等价。

·mStableLeft/Top/Right/Bottom：描述了屏幕排除状态栏和导航栏之后的区域。这个区域并不受状态栏与导航栏的可见性影响。即便状态栏或导航栏被隐藏，Stable区域仍然为排除状态栏与导航栏之后的区域。这也是Stable这个名称的由来。

StableFullscreen区域和Stable区域并不直接参与窗口的布局过程，而是为WMS的客户端提供一个不受状态栏／导航条的可见性所影响的显示区域大小及位置。

·mDockLeft/Top/Right/Bottom：Dock区域用来描述可用来放置停靠窗口的区域。所谓的停靠窗口是指显示在屏幕某一侧的半屏窗口，例如输入法窗口。Dock区域最主要的用途就是用来作为输入法窗口的布局容器。

·mContentLeft/Top/Right/Bottom：描述屏幕中排除了状态栏、导航栏以及输入法（如果显示）后的屏幕区域。与Restricted区域相比，Content

区域额外地受到了状态栏及输入法的影响。当没有状态栏及输入法显示时，Content区域与Restricted区域是等价的。

·mCurLeft/Top/Right/Bottom：与Content区域一样，它也描述了屏幕中排除了状态栏、导航栏以及输入法（如果显示的话）后的屏幕区域。

在大部分情况下，Cur区域与Content区域是相同的。但是这两个区域的计算依据不同。

Content区域和Cur区域与其他区域有个不同的地方：它们的区域并不是在beginLayoutLw（）中被最终确立下来。因为它们受输入法窗口的尺寸影响，所以必须在输入法窗口完成布局后，在offsetInputMethodWindowLw（）函数中得到最终的位置。而在此之前，它们与Dock区域是相同的。

·mSystemLeft/Top/Right/Bottom：与Dock区域的绝大多数情况是相同的，但它们有一个很微妙的不同。状态栏与导航栏的可见性发生变化时有一个淡入淡出的动画效果，这两个区域在这个淡入淡出的过程中是不一致的。Dock认为在淡入淡出的动画过程中，状态栏与导航栏仍然是可见的，而System区域则认为它们是不可见的。

图4-8与图4-9较为直观地描述了布局参数的具体位置。

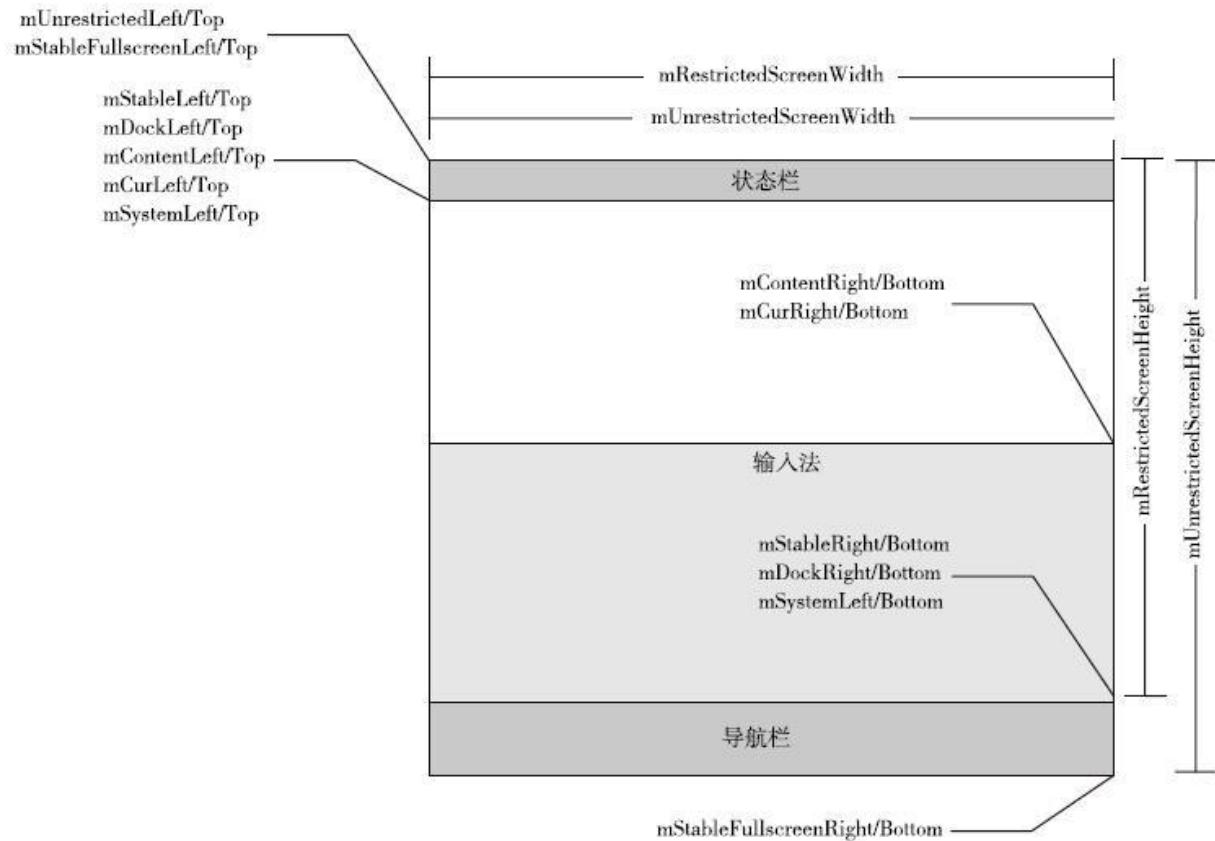


图4-8 坚屏下布局参数所指示的位置

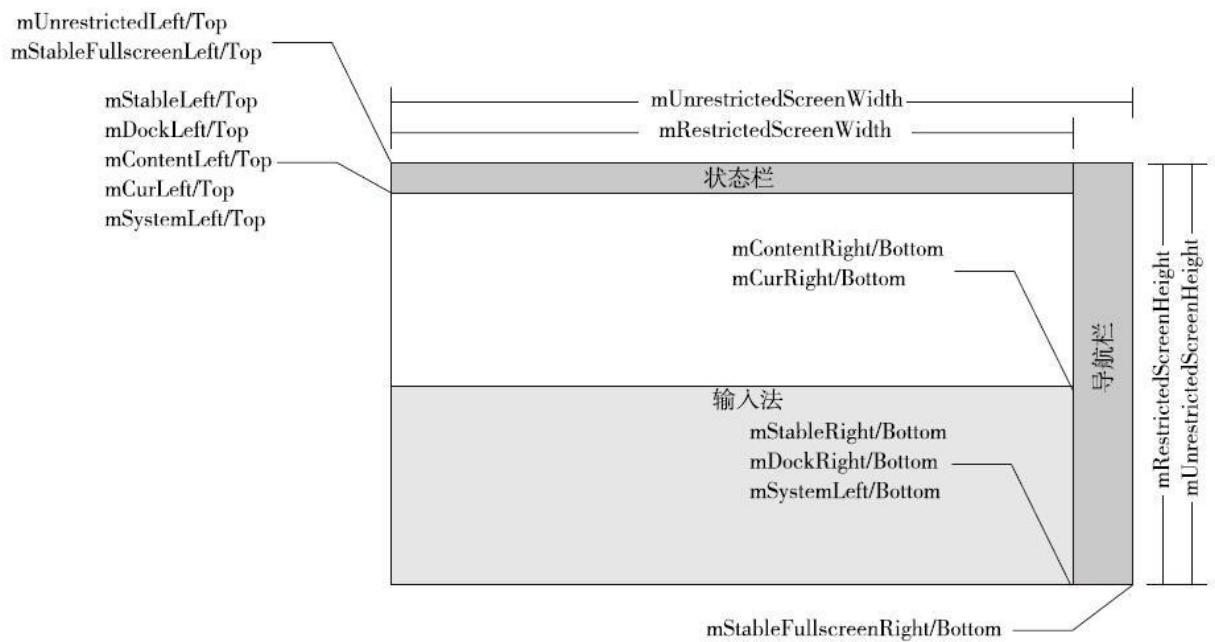


图4-9 横屏下布局参数所指示的位置

一旦这8个矩形区域确定下来，PhoneWindowManager就准备好对窗口进行布局了。



提示

读者可以通过执行adb shell dumpsys window命令来查看这8个区域的当前取值。它们显示在WINDOW MANAGER POLICY STATE一栏中。

(3) 窗口布局的4个参数

上一小节讨论了PhoneWindowManager的beginLayoutLw（）所准备好的布局参数。接下来讨论如何通过这些布局参数确定一个窗口的位置和尺寸。

如前所述，布局窗口使用了两个函数，分别是WindowState.prelayout（）和WindowManager-Policy.layoutWindowLw（）。prelayout（）函数初始化了一个尺寸放大系数，用于在兼容模式下显示窗口，在正常

情况下，此放大系数保持为1，本章不准备讨论这个问题，所以，直接看一下PhoneWindowManager的layoutWindowLw () 的实现：

[PhoneWindowManager.java-->PhoneWindowManager.layoutWindowLw()]

```
public void layoutWindowLw(WindowState win,
    WindowManager.LayoutParams attrs, WindowState attached) { // 状态栏与
    导航栏不再被布局。事实上，这两个窗口是由beginLayoutLw()函数在
    计算布局参数时完成 // 布局的，这一点很容易理解，因为一些布局参
    数是依赖于状态栏和导航栏的尺寸的 if (win == mStatusBar || win ==
    mNavigationBar) { return; } /* 接下来的4个矩形是重点。不要介意
    mTmp***Frame这4个变量。PhoneWindowManager不希望 在布局窗口
    的过程中频繁创建和销毁这 4 个矩形，故而提供了 4 个成员变量用作
    缓存。当完成后续的 计算之后，四个矩形的内容都会被更新 */ final
    Rect pf = mTmpParentFrame; final Rect df = mTmpDisplayFrame; final
    Rect cf = mTmpContentFrame; final Rect vf = mTmpVisibleFrame; // 一大
    段代码，用来计算pf、df、cf和vf ..... // 将计算好的4个矩形交给
    WindowState，由窗口自己计算自己的布局结果
    win.computeFrameLw(pf, df, cf, vf); // 下面这段代码与当前窗口的布局操
    作本身没有什么关系。如上一节所述，当完成输入法窗口的布局后 //
    需要更新Content区域与Cur区域。这里就是完成这个地方 if
    (attrs.type == TYPE_INPUT_METHOD &&
    win.isVisibleOrBehindKeyguardLw() && !win.getGivenInsetsPendingLw())
```

```
{ setLastInputMethodWindowLw(null, null);  
offsetInputMethodWindowLw(win); } }
```

layoutWindowLw () 首先计算了pf、df、cf和vf这4个矩形，然后将这4个矩形作为参数调用WindowState的computFrameLw () 函数完成布局，这4个矩形便成为窗口布局的4个关键参数。

- ParentFrame (pf) : 描述了放置窗口的容器的位置与尺寸。通过 LayoutParam为窗口指定的布局参数如x、y、width、height及gravity等都是相对于pf进行计算的。也就是说，pf对窗口的位置与尺寸计算的影响在4个矩形中起到了决定作用。
- DisplayFrame (df) : df用来限制窗口的最终位置。当窗口通过 ParentFrame完成位置与尺寸的计算后，需要通过df再进行一次校正，要求窗口必须完全位于df之内。
- ContentFrame (cf) : cf不会直接影响窗口布局位置与尺寸，但是它影响了窗口内容的绘制。cf表示当前屏幕上排除所有系统窗口（状态栏、导航栏以及输入法）后所留下的矩形区域。
- VisibleFrame (vf) : 和cf一样，vf也不会直接影响窗口布局的位置与尺寸，而是会影响窗口内容的绘制。vf表示在当前屏幕上，完全不被任何系统窗口所遮挡的一块矩形区域。

那么，这4个参数与前面一节所提到的8个准绳是什么关系呢？简单分析一下layoutWindowLw () 计算这4个参数的代码便可知道，这4个参数是layoutWindowLw () 根据需要从8个准绳当中选出来的。例如，根据窗口的类型、flag的不同，pf可能是Restricted、Unrestricted、Dock或Content区域。df则在绝大部分情况下与pf的取值保持着一致。cf则根据LayoutParams.softInputMode以及flag的取值不同而可能是Content、Dock或Restricted区域。vf则根据LayoutParams.softInputMode的取值选择与cf保持一致，或者选择Cur区域。另外，对于子窗口，其4个参数的取值依赖于其父窗口。

由于需要考虑到的情况非常多，因此layoutWindowLw () 花了很大篇幅计算这4个参数。虽然繁琐却并不复杂，所以这里就不贴出来了，读者可以根据自己的实际需要去探索这段代码。

了解computeFrameLw () 的参数之后，接下来就要研究computeFrameLw () 的工作原理，以及经过这么复杂的流程，对一个窗口进行布局后的产出是什么。

(4) 窗口布局的产出物

参考computeFrameLw () 的代码：

```
[WindowState.java-->WindowState.computeFrameLw()] public void  
computeFrameLw(Rect pf, Rect df, Rect cf, Rect vf) { mHaveFrame = true;
```

```
// 首先，设置mContainingFrame为pf final Rect container =  
mContainingFrame; container.set(pf); // 设置mDisplayFrame为df final Rect  
display = mDisplayFrame; display.set(df); // 计算container的宽高 final int  
pw = container.right - container.left; final int ph = container.bottom -  
container.top; int w,h; /* 接下来计算窗口的尺寸，并保存到w和h中。代  
码比较简单，这里就不贴了，其基本原则是：如果将  
LayoutParams.width或height指定为MATCH_PARENT，则w和h为pf的宽  
和高。否则，如果窗口拥有FLAG_SCALED，则w和h为  
LayoutParams.width与height。倘若未指定FLAG_SCALED，w和h为客  
户端调用setLayoutWindow函数所传的参数mRequestedWidth/Height */ .....  
  
// 将cf保存到mContentFrame final Rect content = mContentFrame;  
content.set(cf); // 将vf保存到mVisibleFrame final Rect visible =  
mVisibleFrame; visible.set(vf); // WindowState的mFrame成员变量表示窗  
口的当前位置与尺寸 final Rect frame = mFrame; // 调用Gravity.apply()函  
数根据gravity、pf、w、h、x、y计算窗口的mFrame。也就是说， //  
对pf对窗口的位置起到决定性作用 Gravity.apply(mAttrs.gravity, w, h,  
container, (int) (x + mAttrs.horizontalMargin * pw), (int) (y +  
mAttrs.verticalMargin * ph), frame); // 调用Gravity.applyDisplay()函数根  
据gravity、df来修正mFrame。其原则为，mFrame必须 // 完全位于df之  
内。经过修正后的mFrame就是窗口的最终位置与尺寸  
Gravity.applyDisplay(mAttrs.gravity, df, frame); // 根据计算出的mFrame调
```

整mContentFrame和mVisibleFrame。 窗口最终的mVisibleFrame和 //
mContentFrame是系统当前的cf或vf与窗口的mFrame的交集 if
(content.left < frame.left) content.left = frame.left; if (content.top <
frame.top) content.top = frame.top; if (content.right > frame.right)
content.right = frame.right; if (content.bottom > frame.bottom)
content.bottom = frame.bottom; if (visible.left < frame.left) visible.left =
frame.left; if (visible.top < frame.top) visible.top = frame.top; if
(visible.right > frame.right) visible.right = frame.right; if (visible.bottom >
frame.bottom) visible.bottom = frame.bottom; // 计算mContentInsets。这个
成员变量用来指示mContentFrame到mFrame的4条边界的距离 final Rect
contentInsets = mContentInsets; contentInsets.left = content.left-frame.left;
contentInsets.top = content.top-frame.top; contentInsets.right = frame.right-
content.right; contentInsets.bottom = frame.bottom-content.bottom; // 计算
mVisibleInsets。这个成员变量用来指示mVisibleFrame到mFrame的4条
边界的距离 final Rect visibleInsets = mVisibleInsets; visibleInsets.left =
visible.left-frame.left; visibleInsets.top = visible.top-frame.top;
visibleInsets.right = frame.right-visible.right; visibleInsets.bottom =
frame.bottom-visible.bottom; }

整理一下computFrameLw () 的产出。

·mFrame : 描述了窗口的位置与尺寸。

·mContainerFrame与mParentFrame：这两个矩形是相等的，保存了pf参数。

·mDisplayFrame：保存了df参数。

·mContentFrame与mContentInsets：mContentFrame表示了在当前窗口中可以用来显示内容的区域，由cf与mFrame相交得出。而mContentInsets则表示了mContentFrame与mFrame的4条边界之间的距离。

layoutWindow () 函数有一个输出参数outContentInsets，即mContentInsets。

·mVisibleFrame与mVisibleInsets：mVisibleFrame表示在当前窗口中不被系统窗口所遮挡的区域，由vf与mFrame相交得出。mVisibleInsets同mContentInsets一样，表示mVisibleFrame与mFrame的4条边界之间的距离。layoutWindow () 函数的另一个输出参数outVisibleInsets就是mVisibleInsets。客户端可以通过outVisibleInsets的取值，适当地滚动自己的显示，以确保一些重要信息不被遮挡。

至此，单个窗口的布局过程便完成了。在经历布局之后，窗口的位置尺寸、内容区域的位置尺寸、可视区域的位置尺寸都得到了更新。这些更新将会影响到窗口Surface的位置尺寸，并且还会以回调的方式通知窗口的客户端，进而影响窗口内容的绘制。

回到performLayoutLockedInner () 函数，在DisplayContent下的所有窗口都已完成布局之后，则调用WindowManagerPolicy.finishLayoutLw () 函数，要求WMP释放布局过程中所使用的资源，或者做一些策略处理。不过WMP的实现者PhoneWindowManager在这个函数中并没有做任何事情。

(5) 关于窗口布局的小结

到此为止，performLayoutLockedInner () 函数的整个工作流程的分析便完成了。performLayoutLockedInner () 对给定的DisplayContent下的所有窗口进行布局，使窗口的mFrame等布局变量的值为最新，以便在后续工作中根据这些布局变量的取值来放置各个窗口的Surface，应用一些显示效果，以及影响客户端显示内容的绘制。

performLayoutLockedInner () 函数的主要工作流程如图4-10所示。

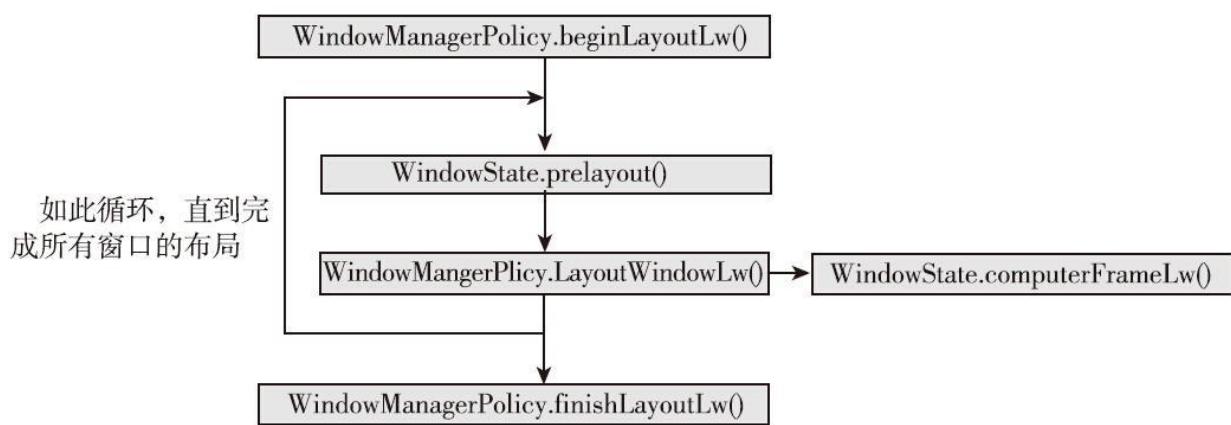


图4-10 performLayoutLockedInner () 的工作流程

窗口布局的演算过程为：根据DisplayContent的属性，以及状态栏、导航栏及输入法窗口的状态，确定8个布局区域。然后根据每个窗口的状态，从8个布局区域中选出4个布局参数pf、df、cf以及vf（可以重复选择）。之后，窗口根据这4个参数计算出布局结果并保存在WindowState中。

2. 检查窗口布局结果

上一节所探讨的内容主要围绕在窗口位置尺寸的布局计算上。经过perform-LayoutLockedInner（）之后，窗口已经各就各位。然而，一些窗口的flag对布局系统还有更多的要求，例如：

- FLAG_FORCE_NOT_FULLSCREEN，拥有这个flag的窗口被显示时，必须同时显示状态栏等系统窗口。
- FLAG_FULLSCREEN，拥有这个flag的满屏窗口被显示时，必须隐藏状态栏等系统窗口。注意，这个flag与FLAG_FORCE_NOT_FULLSCREEN是冲突的。如果两个窗口分别指定了这两个flag，那么FLAG_FORCE_NOT_FULLSCREEN的优先级要高。FLAG_FULLSCREEN将被忽略。
- SYSTEM_UI_FLAG_FULLSCREEN，这是通过view.setSystemUIVisibility（）设置的一个flag。其作用与FLAG_FULLSCREEN一样。

`·FLAG_SHOW_WHEN_LOCKED`, 在锁屏情况下，拥有这个flag的窗口被显示时，隐藏锁屏界面，以使窗口得以呈现给用户。当窗口关闭时，自动恢复锁屏。由于这个flag无视用户设置的加密解锁，因此出于安全考虑，这个flag仅对能够充满屏幕（不包括状态栏等系统窗口）的窗口起作用。

`·FLAG_DISMISS_KEYGUARD`, 与`FLAG_SHOW_WHEN_LOCKED`类似，拥有这个flag的窗口被显示时，将会退出锁屏状态，而且窗口关闭时不会再恢复锁屏。不过此flag的作用也有限制，如果用户设置了密码、图形等安全解锁措施，则在用户输入解锁密码或图形之前，此窗口不会被显示。这个flag仅对能够充满屏幕（不包括状态栏等系统窗口）的窗口起作用。

由于这些flag影响了系统窗口、锁屏界面的可见性，也就会影响窗口的布局过程。然而让人头痛的是，这些flag的生效条件是要求窗口能够覆盖整个屏幕，因此窗口的布局也影响了这些flag的有效性。这两方面互相依赖又互相影响，因此，WMS引入了`pendingLayoutChanges`机制。

这个机制的原理是：首先依据当前的系统窗口、锁屏界面的有效性进行布局，然后基于这个布局来生效上述flag，并检查系统窗口、锁屏界面的可见性是否发生了变化。如果可见性发生变化，则通过`pendingLayoutChanges`变量标记下需要额外完成何种工作。在完成这些

工作之后，重新进行布局，再进行检查，如此反复，直到系统窗口与锁屏界面的可见性不再发生变化为止。

接下来将分析pendingLayoutChanges的工作机制。回顾一下performLayoutAndPlaceSurfacesLockedInner () 函数中的那个布局循环。

[WindowManagerService.java-->

```
 WindowManagerService.performLayoutAndPlaceSurfacesLockedInner()  
 do { // 处理pendingLayoutChanges ..... // 调用  
     performLayoutLockedInner()进行布局 ..... // 清空pendingLayoutChanges  
     字段 displayContent.pendingLayoutChanges = 0; // 布局检查仅发生在手  
     机主屏幕上，因为只有手机主屏幕才有系统窗口及锁屏界面 if  
     (isDefaultDisplay) { // ① 首先调用WMP的beginPostLayoutPolicyLw()函  
     数要求WMP初始化检查所需的状态变量  
     mPolicy.beginPostLayoutPolicyLw(dw, dh); // 按照窗口的Z序自上而下进  
     行遍历 for (i = windows.size() - 1; i >= 0; i--) { WindowState w =  
     windows.get(i); if (w.mHasSurface) { /* ② 检查窗口 WMP的  
     applyPostLayoutPolicyLw()不会改变窗口的任何属性，而是 记录可能影  
     响到系统窗口或锁屏界面可见性的窗口 */  
     mPolicy.applyPostLayoutPolicyLw(w, w.mAttrs); } } // ③ 通过WMP的  
     finishPostLayoutPolicyLw(), 修改系统窗口及锁屏界面的可见性
```

```
displayContent.pendingLayoutChanges |=  
mPolicy.finishPostLayoutPolicyLw(); } // 如果PhoneWindowManager认为  
布局结果仍需进行调整，则重新再来一遍 } while  
(displayContent.pendingLayoutChanges != 0);
```

很显然，我们的调查重点在于WMP的三个函数上。

(1) beginPostLayoutPolicyLw () 分析

beginPostLayoutPolicyLw () 在PhoneWindowManager中的实现比预期的要简单很多，仅仅初始化了一些状态变量。而且方便将布局结果检查过程中所需要的参数一网打尽。

[PhoneWindowManager.java--

```
>PhoneWindowManager.beginPostLayoutPolicyLw() public void  
beginPostLayoutPolicyLw(int displayWidth, int displayHeight) { //  
WindowState类型的变量，保存显示在最上方的满屏窗口  
mTopFullscreenOpaqueWindowState = null; // 指定是否需要强制显示状  
态栏等系统窗口 mForceStatusBar = false; // 指定是否需要在锁屏状态下  
显示状态栏等系统窗口 mForceStatusBarFromKeyguard = false; // 指定是  
否需要隐藏锁屏界面 mHideLockScreen = false; // 指示是否允许在屏幕  
点亮的情况下自动进行锁屏 mAllowLockscreenWhenOn = false; // 指示  
是否需要解除锁屏 mDismissKeyguard = DISMISS_KEYGUARD_NONE;
```

```
// 指示当前是否正在显示锁屏界面 mShowingLockscreen = false; // 指示  
是否正在显示屏保 mShowingDream = false; }
```

相信读者应该能想到三个函数的作用了：beginXXX () 用来初始化参数，applyXXX () 用来设置上述参数，而finishXXX () 则用来根据上述参数执行相应的动作。

(2) applyPostLayoutPolicyLw () 分析

[PhoneWindowManager.java--

```
>PhoneWindowManager.applyPostLayoutPolicyLw() public void  
applyPostLayoutPolicyLw(WindowState win,  
WindowManager.LayoutParams attrs) { /* 整个函数被以下判断条件包裹  
着。注意此函数的调用顺序是沿着Z序自上而下对每个窗口进行调用 因  
此，可以得出一个结论: 仅那些位于第一个可见的满屏窗口（含）之上  
的可见窗口才能影响系统窗口与锁屏界面的可见性*/ if  
(mTopFullscreenOpaqueWindowState == null&&  
win.isVisibleOrBehindKeyguardLw()&&!win.isGoneForLayoutLw()) { //  
只要有窗口拥有FLAG_FORCE_NOT_FULLSCREEN的flag，必然要求  
强制显示系统窗口 if ((attrs.flags & FLAG_FORCE_NOT_FULLSCREEN)  
!= 0) { if (attrs.type == TYPE_KEYGUARD) {  
mForceStatusBarFromKeyguard = true; } else { mForceStatusBar = true; } }  
// 如果有窗口的类型为TYPE_KEYGUARD，表示正在显示锁屏界面 if
```

```
(attrs.type == TYPE_KEYGUARD) { mShowingLockscreen = true; }

boolean applyWindow = attrs.type >= FIRST_APPLICATION_WINDOW
&& attrs.type <= LAST_APPLICATION_WINDOW; ..... // 接下来的操作
针对满屏的应用窗口 if (applyWindow && attrs.x == 0 && attrs.y == 0
&& attrs.width == WindowManager.LayoutParams.MATCH_PARENT &&
attrs.height == WindowManager.LayoutParams.MATCH_PARENT) { // 将
这个满屏的应用窗口记录到mTopFullscreenOpaqueWindowState中。之
后对此函数的 // 的调用将不再做任何事情

mTopFullscreenOpaqueWindowState = win; // 如果这个满屏窗口拥有
FLAG_SHOW_WHEN_LOCKED， 则标记mHideLockScreen为真 if
((attrs.flags & FLAG_SHOW_WHEN_LOCKED) != 0) {

mHideLockScreen = true; mForceStatusBarFromKeyguard = false; } // 处理
满屏窗口的FLAG_DISMISS_KEYGUARD if ((attrs.flags &
FLAG_DISMISS_KEYGUARD) != 0 && mDismissKeyguard ==
DISMISS_KEYGUARD_NONE) { // 如果此窗口已经解锁了锁屏界面，
设置mDismissKeyguard为CONTINUE， 要求维持 // 解锁状态。否则设
置为START， 要求执行解锁动作 mDismissKeyguard =
mWinDismissingKeyguard == win ? DISMISS_KEYGUARD_CONTINUE
: DISMISS_KEYGUARD_START; mWinDismissingKeyguard = win;
mForceStatusBarFromKeyguard = false; } ..... } } }
```

在遍历所有的窗口之后，beginPostLayoutPolicyLw () 中遇到的状态变量都已被设置了合适的值。对我们的分析目标来说，比较重要的状态变量有：

- mTopFullscreenOpaqueWindowState，类型是WindowState，保存了第一个满屏窗口。这个满屏窗口的FLAG_FULLSCREEN将会导致系统窗口被隐藏。
- mForceStatusBar和mForceStatusBarFromKeyguard，这两个状态变量都会要求强制显示系统窗口，对应于FLAG_FORCE_NOT_FULLSCREEN。不同的地方仅仅在于带有这个flag的窗口的类型是否是TYPE_KEYGUARD。
- mHideLockScreen，要求隐藏锁屏界面，对应于FLAG_SHOW_WHEN_LOCKED。从代码中可以看出，仅当第一个满屏显示的应用窗口拥有这个flag时才会被置为true。
- mDismissKeyguard，表示是否执行解锁。这是一个int型的变量，取值为DISMISS_KEYGUARD_NONE/START/CONTINUE三种。NONE表示不会解锁，START表示执行解锁动作，而CONTINUE则表示维持当前状态。

(3) finishPostLayoutPolicyLw () 分析

接下来，`finishPostLayoutPolicyLw()` 将会根据上述状态变量，调整系统窗口与锁屏界面的可见性，因为逻辑非常简单直接，这里就不贴代码了。

需要提醒的是，`FLAG_FULLSCREEN`和
`SYSTEM_UI_FLAG_FULLSCREEN`这两个flag并没有相应的状态变量与之对应。在`finishPostLayoutPolicyLw()` 函数中，当`mForceStatusBar`及`mForceStatusBarFromKeyguard`都为`false`时，会从`mTopFullscreenOpaqueWindowState`以及`mLastSystemUIVisibility`中查找这两个flag，并尝试隐藏系统窗口。

`finishPostLayoutPolicyLw()` 的另外一个工作就是确定`pendingLayoutChanges`的值。当系统窗口的可见性发生变化时，仅会影响窗口的尺寸，因此在这种情况下`pending-LayoutChanges`的值为`FINISH_LAYOUT_REDO_LAYOUT`。要求`performLayoutLockedInner()` 函数在新的系统窗口可见性下重新对所有窗口进行布局。

而当锁屏窗口的可见性发生变化时，情形就比较复杂了，因为当锁屏窗口被隐藏时，显示给用户的窗口所要求的屏幕旋转方向与锁屏窗口可能不一致，这就需要WMS重新计算屏幕方向。另外，显示给用户的窗口可能要求系统壁纸作为其背景，所以WMS还需要重新调整壁纸显示次序，而且无论如何，重新对所有窗口进行布局也是难免的。因此，在锁屏窗口的可见性发生变化后，`pendingLayoutChanges`的值为

FINISH_LAYOUT_REDO_LAYOUT|FINISH_LAYOUT_REDO_WALLPAPER|FINISH_LAYOUT_REDO_CONFIG。

当然，如果两者的可见性都没有发生变化，pendingLayoutChanges的值为0。

3. 处理pendingLayoutChanges

窗口布局完成了，布局检查也完成了，pendingLayoutChanges也被设置为合适的值。接下面回过头再看

performLayoutAndPlaceSurfacesLockedInner () 的那个布局循环中处理 pendingLayoutChanges 的代码：

[WindowManagerService.java-->

WindowManagerService.performLayoutAndPlaceSurfacesLockedInner()]

do { //处理FINISH_LAYOUT_REDO_WALLPAPER。通过调
adjustWallpaperWindowsLocked() //函数调整壁纸窗口的属性，如果有必要，还会重新分配窗口之间的显示层级。壁纸相关的内容将在 //后续的
章节中详细讨论 if (isDefaultDisplay &&

((displayContent.pendingLayoutChanges &

WindowManagerPolicy.FINISH_LAYOUT_REDO_WALLPAPER) != 0)

&& ((adjustWallpaperWindowsLocked() &

ADJUST_WALLPAPER_LAYERS_CHANGED) != 0)) {

```
assignLayersLocked(windows); displayContent.layoutNeeded = true; } // 处理FINISH_LAYOUT_REDO_CONFIG。调用updateOrientationFromAppTokensLocked() // 函数根据最顶层的应用窗口的要求更新屏幕的旋转方向 if (isDefaultDisplay && (displayContent.pendingLayoutChanges & WindowManagerPolicy.FINISH_LAYOUT_REDO_CONFIG) != 0) { if (updateOrientationFromAppTokensLocked(true)) { displayContent.layoutNeeded = true; mH.sendEmptyMessage(H.SEND_NEW_CONFIGURATION); } } // 处理FINISH_LAYOUT_REDO_LAYOUT。这个简单多了，仅仅是将layoutNeeded设置为true // 目的是强制performLayoutLockedInner()对所有窗口进行布局。不过，如果重复次数大于 4， // 将不会再尝试布局 if ((displayContent.pendingLayoutChanges & WindowManagerPolicy.FINISH_LAYOUT_REDO_LAYOUT) != 0) { displayContent.layoutNeeded = true; } // 调用performLayoutLockedInner()进行布局 ..... // 清空pendingLayoutChanges字段 displayContent.pendingLayoutChanges = 0; if (isDefaultDisplay) { ..... displayContent.pendingLayoutChanges |= mPolicy.finishPostLayoutPolicyLw(); } } while (displayContent.pendingLayoutChanges != 0);
```

至此本节分析了DisplayContent布局过程中的布局循环部分的工作原理。

4. DisplayContent的布局后处理

在布局过程几经周折离开布局循环后，窗口的位置与尺寸都已经确定。不过，确定的仅仅是每个窗口的尺寸和位置信息，窗口Surface尚未更新。也就是说，到目前为止，用户还没有看到窗口被放置到布局过程中所指定的位置。

在接下来的布局后处理中，performLayoutAndPlaceSurfacesLockedInner()将会为布局好的窗口设置其Surface大小与尺寸，并附加一些动画效果，例如弹出对话框后的变暗效果（Dimming）等。另外，在布局的前期处理中初始化了mInnerFields对象所保存的一些手机状态，现在是时候更新它们了。看一下DisplayContent的布局后处理的代码：

```
[WindowManagerService.java--> WindowManager.performLayoutAndPlaceSurfaceLockedInner()] private  
final void performLayoutAndPlaceSurfacesLockedInner(boolean  
recoveringMemory) { // 布局的前期处理 ..... try { DisplayContentsIterator  
iterator = new DisplayContentsIterator(); while (iterator.hasNext()) { do { //  
布局循环 } while (displayContent.pendingLayoutChanges != 0); /* 离开布  
局循环后，所有窗口的布局信息都已计算完毕。接下来的主要工作是
```

需要设置窗口 Surface的位置与尺寸 */ // 初始化3个mInnerFields中重要的状态变量 /* mObscured 在后续的窗口遍历过程中指示当前所处理的窗口是否处于被完全遮挡状态。 所谓的完全遮挡是指窗口位于一个非透明的的满屏窗口之下。注意满屏与全屏的区别，全屏是 指窗口将隐藏状态栏与导航栏，而满屏是指窗口将充满除状态栏与导航栏之外的所有区域。几乎所有Activity的窗口都是满屏的 */

mInnerFields.mObscured = false; /* mDimming 表示是否有窗口要求使用 Dimming(变暗)效果。例如，当一个AlertDialog弹出时，其后的所有窗口都会变暗。如果一个窗口启用了透明效果或不是满屏的，则可以通过向 LayoutParameters.flag追加FLAG_DIM_BIHIND以达到此效果 */

mInnerFields.mDimming = false; // mSysWin是另外一种遮挡状态，表示在遍历过程中，当前的窗口是否被系统窗口所遮挡 // 注意，被系统窗口所遮挡不要求系统窗口非透明且满屏 mInnerFields.mSyswin = false; // 自顶向下遍历所有窗口。注意windows列表同时描述了窗口的显示顺序

```
final int N = windows.size(); for (i=N-1; i>=0; i--) { WindowState w = windows.get(i); final boolean obscuredChanged = w.mObscured != mInnerFields.mObscured; // 更新窗口的遮挡状态。窗口的遮挡状态在一定程度上也描述了窗口的可见性 w.mObscured = mInnerFields.mObscured; /* 如果当前窗口未被遮挡，则调用 handleNotObscuredLocked函数。在4.4.4节中介绍mInnerFields时提到了这个函数。在这个函数中WMS将从窗口 的LayoutParams里获取有关
```

Dimming、屏幕亮度、键盘背光等属性的设置，然后将这些设置保存到mInnerFields中。与此同时，此函数还检查窗口的尺寸以及类型，以此更新当前的遮挡状态。当然，可能有多个窗口都指定上述设置，谁说了算呢？WMS的解决办法是：位于上层的窗口的设置拥有更高的优先级。不过，如果窗口被完全遮挡，或被系统窗口所遮挡，这些设置将被忽略。顺带提一句，Dimming效果也是handleNotObscuredLocked()发起的。而屏幕亮度、键盘背光等状态的设置则是在布局的最终阶段完成的 */ if (!mInnerFields.mObscured) { handleNotObscuredLocked(w, currentTime, innerDw, innerDh); } // 关于壁纸的一些操作，在后续的章节中再作介绍 // 接下来就是设置Surface的位置与尺寸了 final WindowStateAnimator winAnimator = w.mWinAnimator; /* w.shouldAnimateMove()表示窗口的位置是否发生变化（也就是mFrame.left与mFrame.top发生变化）。WMS通过一个定义的动画anim.window_move_from_decor来完成窗口Surface的移动。WMS做了一个非常聪明的小动作：这个动画是一个TranslateAnimation，并且动画的移动量为100%p，也就是移动量为容器尺寸。WMS将mFrame的位置变化量作为动画的容器尺寸，从而达到不改变动画自身的属性，实现移动到任意位置的动画效果 */ if (w.mHasSurface && w.shouldAnimateMove()) { Animation a = AnimationUtils.loadAnimation(mContext, com.android.internal.R.anim.window_move_from_decor);

```
winAnimator.setAnimation(a); winAnimator.mAnimDw =  
w.mLastFrame.left - w.mFrame.left; winAnimator.mAnimDh =  
w.mLastFrame.top - w.mFrame.top; try { w.mClient.moved(w.mFrame.left,  
w.mFrame.top); } catch (RemoteException e) { } } if (w.mHasSurface) { /*  
调用WindowStateAnimator的commitFinishDrawingLocked()。如果当前  
窗口的客户端已经完成对Surface的绘制，并尚未被显示，则向动画系  
统提交绘制完毕的通知，在动画的下一帧处理时，窗口将被显示出来。  
WMS为窗口的绘制状态实现了一个状态机，用来与客户端同步  
Surface的绘制状态。关于这个状态机的详细内容在动画系统中再深入  
探讨 */ final boolean committed =  
winAnimator.commitFinishDrawingLocked(currentTime); ..... /* 更新窗口  
的Surface的位置与尺寸。注意，WMS并没有为Surface的尺寸变化提供  
动画效果。这是因为Surface尺寸的变化将会导致GraphicBuffer的尺寸  
变化，这就需要重新进行内存分配。而且对客户端来说，Surface尺寸  
变化还意味着重绘的必要性。窗口尺寸变化的开销是很大的。因此，  
WMS并没有对尺寸变化使用动画效果。另外，此函数同时也设置了窗  
口的位置，而且并不是窗口的mFrame所指定的位置而是mShown-  
Frame。在布局的过程中并没有计算过这个矩形。所以窗口的实际位置  
并没有在此时此刻被更新。mShownFrame是由动画系统根据窗口动画  
的进度实时计算的。在4.5节中将深入介绍窗口动画的工作原理 */  
winAnimator.setSurfaceBoundariesLocked(recoveringMemory); ..... /* 接
```

下来，如果窗口属于一个AppWindowToken，则计算此AppWindowToken已绘制的窗口数量。在AppWindowToken下的所有窗口都已完成绘制后，AppWindowToken的allDrawn状态会被置为true。allDrawn的目的是让Activity的所有窗口能够同时显示给用户。属于AppWindowToken的窗口经过commitFinishDrawingLocked()的处理而变为READ_TO_SHOW状态后必须等待allDrawn为true后才会显示出来，而其他窗口则没有这个限制 */ } /* 将位置尺寸、ContentInsets或VisibleInsets在布局过程中发生变化的窗口添加到mResizingWindow列表中。在完成布局时，列表中窗口的客户端将会收到resized()回调通知 */ updateResizingWindows(w); } // 遍历所有窗口的循环结束 // 如果handleNotObscuredLocked()函数没有将mDimming状态设置为true，则表示当前 // 没有任何窗口有启用Dimming效果的要求，于是要求WindowAnimator停止Dimming效果 if (!mInnerFields.mDimming && mAnimator.isDimmingLocked(displayId)) { stopDimmingLocked(displayId); } } // 对DisplayContent的布局到此结束 } catch (Exception e) { } finally { Surface.closeTransaction(); } // 完成布局后的策略处理 }

DisplayContent的布局后处理的主要工作内容是：

- 设置窗口的遮挡状态。

- 从窗口的LayoutParams中提取关于屏幕亮度、键盘亮度、输入超时等设置。
- 发起或取消Dimming效果。
- 设置窗口Surface的位置与尺寸。其中，位置的变化是有动画效果的，而WMS出于性能考虑，尺寸的变化则没有动画效果。
- 如果窗口的客户端已经完成对Surface的绘制工作，则显示这个窗口。

完成布局后处理后，当前DisplayContent的布局就完成了，所有的窗口都已各就各位。

5. 关于DisplayContent布局的小结

经历了漫长的代码分析，DisplayContent布局过程终于完结了。再次回顾一下布局过程中的两大阶段：

- 以窗口布局计算为主要任务的布局循环。布局循环以performLayoutLockedInner（）函数为核心，根据屏幕尺寸以及状态栏、导航栏、输入法窗口等系统窗口确定了作为窗口布局准绳的8个矩形，再从8个矩形中选择4个矩形作为窗口布局的直接参数，并以这4个参数计算出窗口的最终位置。

·根据布局计算结果设置Surface的位置与尺寸，并更新一些由窗口指定的系统属性为目的布局后处理。

DisplayContent的布局其实就是窗口的布局，而窗口的布局中又以窗口的布局计算为核心。这部分内容需要读者尽可能深刻地理解。

4.4.6 布局的最终阶段

在布局的最终阶段中，所有的DisplayContent都已基本完成布局（之所以说基本完成，是因为如果布局循环的次数大于6次，有可能DisplayContent.pendingLayoutChanges仍然不为0）。

布局的最终阶段的工作相对繁杂。包括设置屏幕亮度、背光亮度、保持屏幕唤醒等（基于布局后处理时设置的mInnerFields中的相关状态），通知窗口客户端其布局发生变化等。在这里不再一一说明。

其中比较重要的是，在DisplayContent的布局循环中有可能没能将pendingLayoutChanges清零，此时需要设置DisplayContent的layoutNeeded为true，由布局外围循环重新进行一遍完整布局。

另外，在设置窗口的位置时使用了动画，并且Dimming效果也是基于动画实现的，因此在布局的最后，需要通过调用updateLayoutToAnimationLocked () 启动动画系统。

4.5 WMS的动画系统

在本章前面的讨论中，曾经多次看到WindowStateAnimator的身影：在 `perpare-SurfaceLocked ()` 中设置Surface的显示次序。在 `relayoutWindow ()` 中创建及销毁Surface，在 `performLayoutAndPlaceSurfacesLockedInner ()` 中设置Surface的尺寸与位置，等等。WindowStateAnimator在Surface的操作过程中发挥了极大的作用，而且布局过程中还有一些和动画系统相关的重要的内容尚未解释，例如`mShownFrame`的计算、窗口绘制的状态机的原理等，这一节将深入探讨WMS的动画系统。



注意

这里再强调一下，WMS不负责窗口的具体绘制，因此WMS动画系统仅会影响窗口的位置、显示尺寸与透明度，不会影响窗口上所绘制的内容。窗口内容的动画在窗口的客户端由ViewRootImpl驱动完成。

另外，动画系统中所改变的显示尺寸与布局过程中的Surface尺寸是两个不同的概念。布局过程中的Surface尺寸是Surface的实际尺寸，这个尺寸决定了其GraphicBuffer的大小以及Canvas可以绘制的区域。而动画过程中的尺寸则是渲染尺寸，只是在最终输出的过程中将Surface的内容放大或缩小。

4.5.1 Android动画原理简介

在正式开始WMS动画系统的探讨之前，有必要先了解一下Android动画的工作原理。

1.Animation类与Transform类

Android提供了平移（Translate）、缩放（Scale）、旋转（Rotate）以及透明度（Alpha）4种类型的动画。这些动画分别由TranslateAnimation类、ScaleAnimation类、RotateAnimation类以及AlphaAnimation类实现，它们都是Animation类的子类。

直观上，很多人可能都会认为Animation及其子类在开始动画后会在某一个线程中以一定的频率不断地操作动画目标的相关属性，从而实现动画效果。其实不然，Android中的Animation类的功能非常轻量级。在给定了初始状态、结束状态、启动时间与持续时间后，该类可以为使用者计算其动画目标在任意时刻的变换（Transformation），这是Animation类唯一的用途。

Transformation类描述了一个变换。它包含了两个分量：透明度及一个二维变换矩阵。同变换矩阵的运算一样，多个Transformation也可以进行类似于矩阵的先乘、后乘等叠加操作。其计算方法为，透明度分量相乘，变换矩阵分量进行相应的先乘、后乘。



说明

二维变换矩阵可以在二维空间中对变换目标实现平移、旋转、缩放与切变等效果。将两个矩阵相乘可以组合其各自的变换效果。关于其变换的原理与运算，有很多图形学相关的书籍与文章可以参考，本书不再赘述。

本节将探讨一下Animation类是如何计算Transformation的。获取Transformation使用Animation类的getTransformation () 函数，这个函数根据给定的时间戳，计算动画目标所需要进行的变换。另外，其返回值如果为true，则表示动画尚未完结，调用者需要继续处理下一帧动画。

```
public boolean getTransformation(long currentTime, Transformation  
outTransformation)
```

在真正的变换计算开始之前，Animation首先将currentTime进行一个“标准化”操作。因为动画的时间参数有起始时间、滞后时间、持续时间等。其中滞后时间是指从起始时间开始，到真正开始执行动画的时间间隔，滞后时间往往用于两个动画的衔接。倘若让这些时间参数参与到变换计算中无疑会大大增加计算的复杂度。“标准化”正是为了解决这个问题。

```
normalizedTime = ((float) (currentTime - (mStartTime +  
startOffset)))/(float) duration;
```

normalizedTime就是currentTime经过标准化后的时间，我们称为标准时间。标准时间隐藏了上述几个时间参数，后续的计算只需关心标准时间相对于0、1的取值即可，大大减少了复杂度。

接下来，Animation考虑的是时间线效果的实现，例如，调用者也许希望实现一个先加速后减速的移动效果，于是Animation需要对刚刚计算出的标准时间再做进一步加工，如下所示：

```
final float interpolatedTime =  
mInterpolator.getInterpolation(normalizedTime);
```

mInterpolator是一个实现了Interpolator接口的对象。Interpolator接口定义了一个getInterpolation () 函数，用于根据一个现有的标准时间计算其对应的插值时间（或者说应用了时间线效果后的时间）。其子类众

多，分别实现了各种各样的时间线效果。调用者可以通过Animation类的setInterpolator () 函数设置所期望的效果。以 AccelerateDecelerateInterpolator这个插值类的实现来说明一下插值器是如何工作的。其getInterpolation () 函数的实现为：

```
[AccelerateDecelerateInterpolator.java-->AccelerateDecelerateInterpolator.getInterpolation()] public float getInterpolation(float input) { return (float)(Math.cos((input + 1) * Math.PI) / 2.0f) + 0.5f; }
```

它的返回值实际上是一个余弦函数的图像。如图4-11所示。当标准时间正常地匀速流逝时，插值时间则经历了一个先加速后减速的流逝过程。后续的变换计算直接使用插值时间即可“免费”地获得这个平滑、圆润的效果。

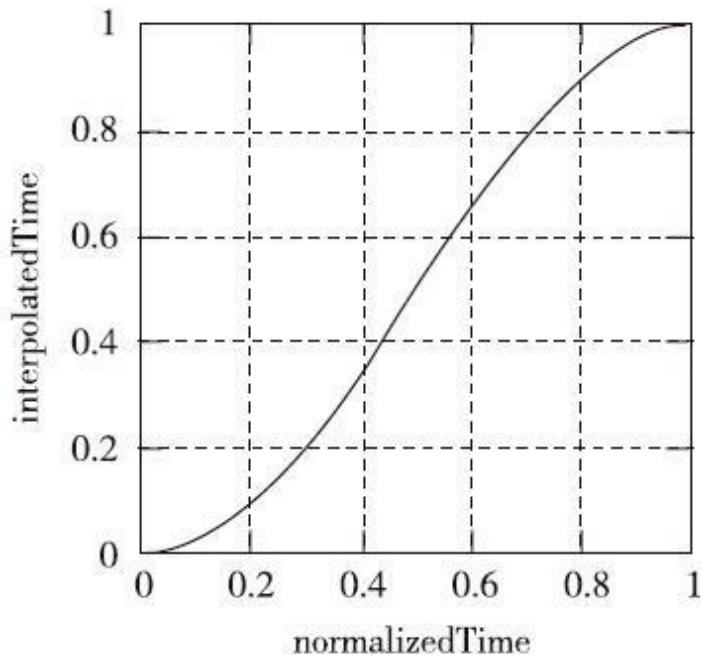


图4-11 插值时间与标准时间

最后，Animation类使用插值时间调用由子类实现的applyTransformation()函数，计算当前的变换：

`applyTransformation(interpolatedTime, outTransformation);`

计算结果将被保存至调用者所提供的Transformation对象中。

以最简单的AlphaAnimation为例，applyTransformation () 的实现如下所示：

[AlphaAnimation.java-->AlphaAnimation.applyTransformation()]
`protected void applyTransformation(float interpolatedTime, Transformation t) { final`

```
float alpha = mFromAlpha; t.setAlpha(alpha + ((mToAlpha - alpha) *  
interpolatedTime)); }
```

非常简单！这完全得益于Animation类优秀的设计。至此，Animation类的工作原理已经非常清楚了。既然Animation类仅仅根据时间戳计算变换，那么是谁对动画一帧一帧地进行渲染呢？对，就是本章开始时的例子SampleWindow中所使用的Choreographer类。

2. 动画驱动器Choreographer类

Choreographer类是Android 4.2新增的一个API。此类的功能与读者所熟知的Handler的post () 函数非常类似。区别就在于，Choreographer类处理回调的时机为屏幕的垂直同步（VSync）事件到来之时，其处理回调的过程被当作渲染下一帧的工作的一部分。

Choreographer为驱动动画提供了以下函数：

- postCallback (int callbackType, Runnable action, Object token)

在下一次VSync时执行action参数所指定的操作。callbackType的取值为CALLBACK_INPUT、CALLBACK_ANIMATION和CALLBACK_TRAVERSAL，表示action所指定的回调的工作内容分别为处理输入事件、处理动画、进行布局。当一次VSync事件到来时，Choreographer将优先执行所有INPUT类型的回调，然后执行

ANIMATION，最后才执行TRAVERSAL。这不正是游戏的主循环的工作流程吗？token参数对回调的执行不会产生影响。只是当取消一次回调时，除提供回调对象之外，必须提供相同的Token。

·postCallbackDelayed (int callbackType, Runnable action, Object token, long delayMillis)

同上一个函数一样，不过增加了一个延迟参数。其作用与Handler的postDelayed () 函数一致。

·postFrameCallback (FrameCallback callback)

在下一次VSync时执行callback所指定的回调。与postCallpack () 的本质功能没有太大区别。不过其回调类型被强制为 CALLBACK_ANIMATION，而且FrameCallback接口的定义的函数为：doFrame (long frameTimeNanos) ，参数是一个纳秒级的时间戳。因此这个函数天生就是为处理动画帧所设计的。

·postFrameCallback (FrameCallback callback, int timeDelayed)

同上，只是增加了一个延迟时间。

3. 使用Choreographer和Animation实现动画

在学习Choreographer和Animation的使用方法后，就可以通过它们勾勒出Android动画的工作骨架了。下面是一个演示类，这个类向外界提供一个名为startAnimation () 的函数，用于立刻开始运行参数所指定的动画。

```
public class SampleAnimation { // Choreographer是线程唯一的。通过  
getInstance()获取当前线程上的实例 Choreographer mChoreographer =  
Choreographer.getInstance(); // 一个Animation Animation mAnimation =  
null; //保存需要运行的动画，并通过scheduleNextFrame准备执行第一帧  
public synchronized void startAnim(Animation anim) { mAnimation =  
anim; scheduleNextFrame(); } // 将负责执行渲染的mAnimationRunnable  
抛给Choreographer private void scheduleNextFrame() {  
mChoreographer.postCallback(Choreographer.CALLBACK_ANIMATION ,  
mAnimationRunnable , null); } // 这个Runnable实现了如何渲染一帧  
Runnable mAnimationRunnable = new Runnable() { public void run() {  
synchronized (SampleAnimation.this) { // 从这里正式开始渲染动画的一  
帧 if (mAnimation != null) { // 获取当前时间 long time =  
SystemClock.uptimeMillis(); // 新建一个Transformation用以保存  
Animation的变换计算结果 Transformation transform = new  
Transformation(); // 计算出Transformation，返回值more表示动画是否需  
要继续执行 boolean more = mAnimation.getTransformation(time,  
transform); // 使用Animation计算出的Transformation进行渲染
```

```
PERFORM_RENDER_WITH_TRANSFORMATION(transform); // 如果  
mAnimation表示动画尚未结束，则向Choreographer申请处理下一帧 //  
否则清空mAnimation并不再向Choreographer发送请求 if (more)  
scheduleNextFrame(); else mAnimation = null; } } } };
```

可以看出，在Android 4.2中实现动画的方法为：

通过Choreographer发送一个Runnable以处理一帧动画。在处理动画时，使用Animation.getTransformation () 函数获取动画对象所需要进行的变换，然后根据变换对动画对象进行渲染。所谓渲染，可以是绘制对象或者改变对象的属性。如果动画需要继续，则继续向Choreographer发送下一帧的处理请求。



注意

虽然Choreographer用来处理动画，但是其postCallback () 以及 postFrameCallback () 函数并不会自动重复调用传入的Runnalbe或 FrameCallback对象，而是仅仅调用一次。因此，当调用者处理完一帧

后，如果还需要处理下一帧，必须重新调用上述函数，否则动画将会停止。

上述的代码是Android 4.2中最基本的动画实现原理，所有动画都是在这个骨架之上实现的，包括复杂的WMS。

4.5.2 WMS的动画系统框架

学习Android动画的实现原理之后将以此为基础，研究WMS的动画子系统的框架。

在WMS中，有一个同上一节SampleAnimation.scheduleNextFrame () 功能一样的函数，用于启动窗口动画。其定义如下：

```
[WindowManagerService.java-->WindowManagerService.scheduleAnimationLocked()]\ void  
scheduleAnimationLocked() { // mLayoutToAnim与其类名一样，用于保  
存来自布局子系统传递给动画系统的参数 //在后续内容中将讨论  
mLayoutToAnim成员 final LayoutToAnimatorParams layoutToAnim =  
mLayoutToAnim; // 如果动画帧已经被发送给Choreographer，则没有必  
要重复发送 if (!layoutToAnim.mAnimationScheduled) {  
layoutToAnim.mAnimationScheduled = true; // 处理动画帧的Runnable是  
mAnimator.mAnimationRunnable。 mAnimator在分析WMS构成 // 时介绍  
过，它是一个WindowAnimator类的对象，是WMS所有动画的管理者
```

```
mChoreographer.postCallback( Choreographer.CALLBACK_ANIMATION,  
mAnimator.mAnimationRunnable, null); } }
```

负责处理动画帧的mAnimationRunnable的定义为：

[WindowAnimator.java-->WindowAnimator.mAnimationRunnable]

```
mAnimationRunnable = new Runnable() { @Override public void run() { //  
注意，动画帧处理过程是一个双锁！首先锁住mWindowMap，然后是  
mAnimator。因此无论如 // 如何都要避免以相反的顺序使用这两个锁。  
例如，当动画系统中有函数访问布局系统时，要尤为慎重
```

```
synchronized(mService.mWindowMap) {  
synchronized(WindowAnimator.this) { // 将布局系统提供的参数“解压”到  
动画系统，这个话题随后再讨论 copyLayoutToAnimParamsLocked(); //  
渲染一帧动画 animateLocked(); } } };
```

也就是说，WMS动画的渲染集中实现在WindowAnimator类的
animateLocked () 函数中。在深入讨论这个函数之前，先了解一下
WindowAnimator以及其组成部分。

4.5.3 WindowAnimator分析

1.WindowAnimator的组成

本章4.2节曾经讨论了WMS的窗口管理结构是由以下三种成员组成的：DisplayContent、WindowToken和WindowState，分别对应屏幕、显示组件和窗口本身。其实动画系统中也有与之对应的组成结构。它们分别是以屏幕为动画目标的DisplayContentAnimator、以Activity为动画目标的AppWindowAnimator，以及以窗口为动画目标的WindowStateAnimator。而WindowAnimator则是协调它们工作的管理者。另外，WMS管理的不仅仅有窗口动画，还有一些特效动画，如Dimming、屏幕旋转等。所以与之对应的还有DimAnimator、ScreenRotateAnimation这两种Animator。

这些Animator的作用如下：

- DisplayContentAnimator：Android 4.2新增的Animator，仅仅用于存储位于其屏幕上的其他类型的Animator。例如ScreenRotateAnimation、DimAnimator以及WindowStateAnimator。
- AppWindowAnimator：用于对一个AppWindowToken进行动画处理。由AppWindowAnimator计算得出的Transformation将被应用在此Token所拥有的所有窗口上。AppWindowToken的动画主要是用来表现Activity的进入与退出。
- WindowStateAnimator：用于对一个窗口进行动画处理。它计算得出的Transformation将与AppWindowAnimator、ScreenRotateAnimation以及

父窗口的Window-StateAnimator三者的Transformation一并应用到窗口的Surface上。所以窗口的动画其实就是Surface的动画。

·ScreenRotateAnimation：用于处理转屏动画。由它所计算出的Transformation将被应用在其所属屏幕的所有窗口之上。

·DimAnimator：用于实现Dimming效果。这个Animator的动画对象不是窗口，而是一块黑色的Surface。当需要Dimming效果时，DimAnimator会动画地将这块Surface以一定的透明度衬于需要Dimming效果的窗口之下。之所以使用一个Animator来完成这个工作，是为了提供舒服的淡入淡出效果。

除了DimAnimator和DisplayContentsAnimator之外，其他的Animator都有一个名为stepAnimationLocked (int timestamp) 函数。这个函数顾名思义，将其状态迁移到由时间戳timestamp所指定的一帧上。完成stepAnimationLocked () 的调用之后，Animator便更新了绘制当前帧时其动画目标所需的Transformation。

这些Animator的从属关系如图4-12所示。

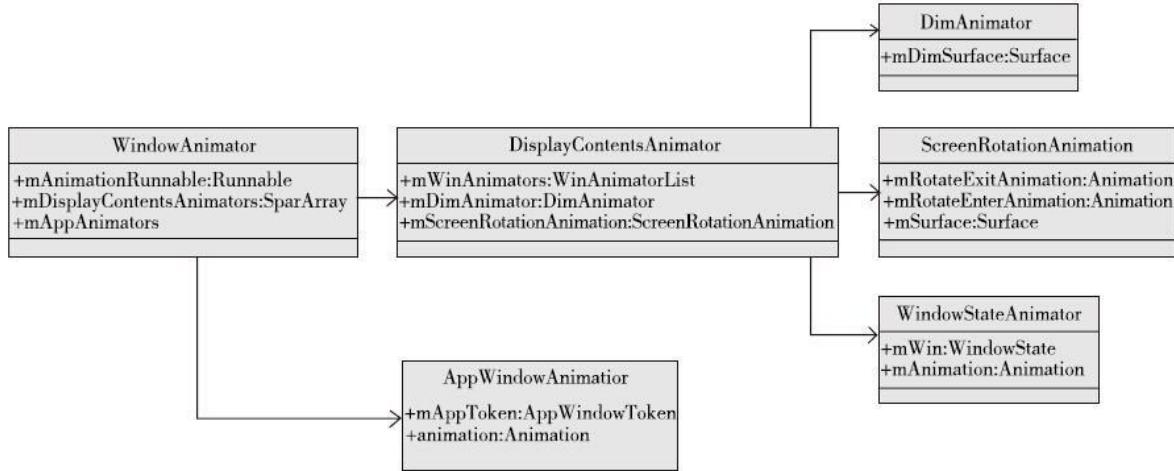


图4-12 Animator的从属关系

2.WindowAnimator的animateLocked () 分析

接下来，看一下WindowAnimator是如何处理一帧动画并协调上述5种Animator的工作的。从WindowAnimator.animateLocked () 函数开始：

```
[WindowAnimator.java-->WindowAnimator.animateLocked()]
private void
animateLocked() { if (!mInitialized) { return; } // 同上一节的
SampleAnimation例子一样，WindowAnimator需要获取当前帧的时间戳
mCurrentTime = SystemClock.uptimeMillis(); // 记录处理当前帧之前的动画状态到wasAnimating中。wasAnimating可以用来确定动画是刚刚开始
// 第一帧或是最后一帧。可以以此进行一些初始化或清理工作
boolean
wasAnimating = mAnimating; mAnimating = false; //开启一个Surface的
Transaction
Surface.openTransaction(); Surface.setAnimationTransaction();
try { /* ① 首先处理AppWindowAnimator的动画。这个函数会调用
```

AppWindowAnimator的step- AnimationLocked()函数以更新

AppWindowAnimator的Transformation，这个变换将会在后面影响窗口的Surface的位置 */ updateAppWindowsLocked(); // 接下来会遍历所有的DisplayContentsAnimator，并处理其上的旋转动画与窗口动画 final int numDisplays = mDisplayContentsAnimators.size(); for (int i = 0; i < numDisplays; i++) { final int displayId = mDisplayContentsAnimators.keyAt(i); DisplayContentsAnimator displayAnimator = mDisplayContentsAnimators.valueAt(i); //② 处理屏幕旋转动画 final ScreenRotationAnimation screenRotationAnimation = displayAnimator.mScreenRotationAnimation; if (screenRotationAnimation != null && screenRotationAnimation.isAnimating()) { // 更新旋转动画的 Transformation，它一样会在后面影响窗口的Surface的位置 if (screenRotationAnimation.stepAnimationLocked(mCurrentTime)) { mAnimating = true; } else { /* 当旋转完成后，AMS会向Activity发送新的Configuration。这个动作的起点就是这里了。stepAnimationLocked() 的返回值为false时，表示这是动画的最后一帧。此时通过向 mBulkUpdateParams添加SET_UPDATE_ROTATION棋标，以此通知布局系统旋转动画完毕，需要通知AMS进行更新 */ mBulkUpdateParams |= SET_UPDATE_ROTATION; screenRotationAnimation.kill(); // 由此可见，当屏幕没有处于旋转过程中时，DisplayContentsAnimator的 // mScreenRotationAnimation是null

displayAnimator.mScreenRotationAnimation = null; } } /* ③ 接下来处理
WindowStateAnimator的动画。这个处理分为两部分，普通窗口的动画
与壁纸 动画。此函数更新了WindowStateAnimator的Transformation的
值。同时，在这一步中 也会检查窗口绘制状态机的状态，如果窗口在
DisplayContent的布局后处理中完成了commit- FinishDrawingLocked()的
提交，那么窗口的状态会在这一步中变为IS_DRAWN，随后
prepareSurfaceLocked()会将窗口显示出来*/

performAnimationsLocked(displayId); final WinAnimatorList
winAnimatorList = displayAnimator.mWinAnimators; final int N =
winAnimatorList.size(); for (int j = 0; j < N; j++) { /*④ 渲染动画
WindowStateAnimator的prepareSurfaceLocked()将集合
AppWindowAnimator、 ScreenRotationAnimator以及
WindowStateAnimator的Transformation到一起，修改Surface的layer、
matrix、 alpha等属性，从而实现窗口动画的渲染*/
winAnimatorList.get(j).prepareSurfaceLocked(true); } } // 至此窗口动画的
一帧渲染完成 // ⑤ 渲染Dimming和屏幕旋转效果 for (int i = 0; i <
numDisplays; i++) { // 首先是屏幕旋转动画 if
(screenRotationAnimation != null) {
screenRotationAnimation.updateSurfacesInTransaction(); } // 处理Dimming
动画。前面提过， Dimming效果是由一块黑色的Surface完成的 //
Dimming动画的处理分为以下两个步骤 final DimAnimator.Parameters

```
dimParams = displayAnimator.mDimParams; final DimAnimator  
dimAnimator = displayAnimator.mDimAnimator; // 更新Diming参数，包  
括Dimming的尺寸、 Dimming的颜色深度以及Dimming的目标等 // 所谓  
Dimming的目标是指DimmingSurface将要衬于其下的那个窗口 if  
(dimAnimator != null && dimParams != null) {  
    dimAnimator.updateParameters(mContext.getResources(), dimParams,  
    mCurrentTime); } // 然后根据当前的时间戳,更改Dimming Surface的透明  
度,实现Dimming的淡入淡出效果 if (dimAnimator != null &&  
dimAnimator.mDimShown) { mAnimating |=  
    dimAnimator.updateSurface(isDimmingLocked(displayId), mCurrentTime,  
    !mService.okToDisplay()); } } finally { Surface.closeTransaction(); } // 至  
此，动画的一帧已经渲染完成 ..... /* ⑥ 如果有必要，向布局系统请求  
一次布局 在动画的过程中，有时需要布局系统“搭把手”，例如前面所  
见到的，在屏幕旋转动画完成后需要布局系统更新Configuration到  
AMS。为了实现从动画系统到布局系统的请求传递，WindowAnimator  
定义了以下两个成员以在动画渲染的过程中收集所有对布局系统的请  
求，并在动画帧渲染结束后，将其通过 updateAnimToLayoutLocked()一  
次性发送给布局系统 */ if (mBulkUpdateParams != 0 ||  
mPendingLayoutChanges.size() > 0) { updateAnimToLayoutLocked(); } //  
⑦ 如果仍有动画处于运行状态，通过WMS安排下一帧的处理 if  
(mAnimating) { synchronized (mService.mLayoutToAnim) {
```

```
mService.scheduleAnimationLocked(); } } else if (wasAnimating) { //  
    wasAnimating为true而mAnimating为false表示所有动画在这一帧停止 //  
    此时需要令布局系统重新进行一次布局。其实，  
    updateAnimToLayoutLocked()就已经 // 向WMS请求一次重新布局了，所  
    以这里略显多余 mService.requestTraversalLocked(); } }
```

这个函数相对较长，不过结构非常清晰：

- 首先通过updateAppWindowsLocked () 计算AppWindowToken动画在当前时间所要求的变换。
- 针对每一个DisplayContentAnimator，计算其屏幕旋转动画在当前时间所要求的变换。
- 针对每一个DisplayContent下的每一个窗口，计算其自身动画在当前时间所要求的变换。
- 针对每一个DisplayContent下的每一个窗口，将上述3个变换同时应用到窗口的Surface上，实现窗口动画帧的渲染。
- 渲染效果动画，包括屏幕旋转以及Dimming效果。
- 如果动画渲染的过程中，需要布局系统有相应的动作，则将请求放置在mBulkUpdateParams与mPendingLayoutChanges两个成员中，动画渲

染完毕后统一通过updateAnimToLayoutLocked () 将动作请求发送给布局系统。

·如果动画仍需进行，通过WMS.scheduleAnimationLocked () 安排下一帧的处理。



注意

需要注意的是，WindowAnimator的animateLocked () 函数并不是只在运行一个动画，而是一批保存在各个Animator中的动画。

可以看出，影响窗口的最终显示位置的Animator是AppWindowAnimator、Screen-RotateAnimationAnimator以及WindowStateAnimator。限于篇幅，本节将仅探讨三个Animator中最重要的WindowStateAnimator的工作原理。剩下的两个Animator读者可以参照WindowStateAnimator自行研究，因为它们的本质是一致的，而且也比WindowStateAnimator简单。

4.5.4 深入理解窗口动画

1.WindowStateAnimator初探

研究窗口动画离不开WindowStateAnimator（以下简称WSA）。WSA在WindowState的构造函数中随之一起被创建，二者互持双方的引用。WindowState保存了窗口管理方面的属性，例如窗口位置、尺寸、主序、子序、父窗口的引用等，而WSA则保存了窗口的Surface的属性。前者侧重于窗口管理，后者则侧重于窗口显示。WSA的几个重要成员变量如下：

- mSurface保存了窗口的Surface，同时也是WSA的动画目标。
- mWin保存了WSA所对应的窗口。
- mAnimation是由WMS设置给WSA的一个Animation类的对象，保存了WMS要求窗口执行的动画。当窗口没有运行动画时，mAnimation对象保持为空。
- mAnimDw和mAnimDh描述了窗口动画时的“容器尺寸”。容器尺寸一般是DisplayInfo.appWidth/Height，也就是屏幕上排除状态栏/导航栏等系统窗口之后供应用程序显示的区域尺寸。在为Animation的属性设置带有“p”后缀时，水平方向与垂直方向的参考值就是这两个变量。另外，读者可以回顾一下在布局后的处理过程中使用动画移动窗口位置的情况，当时将这两个值分别设置成窗口位置变化量，而动画的移动距离都是100%p。

·mAnimLayer，对除Wallpaper与IME之外的窗口来说，其值等于WindowState的mLayer，也就是窗口最终显示次序。在prepareSurfaceLocked () 中，它将作为Surface的layer被设置到显示系统。

·mAlpha和mShownAlpha都表示窗口的Surface的透明度。它们两个的区别是，mAlpha是由窗口的LayoutParams.alpha指定的，也就是客户端要求的透明度。而mShownAlpha是mAlpha在混合了WindowStateAnimator、AppWindowAnimator、ScreenRotateAnimation以及父窗口的WindowStateAnimator的Transformation中的alpha分量后的透明度，也就是实际透明度。

·mDsDx、mDtDx、mDsDy和mDtDy这4个变量自己本身没有什么意义，它们组合起来就是Surface的变换矩阵。它们是4个Animator的Transformation变换矩阵组合之后（相乘）的缩放与旋转分量。通过Surface.setMatrix () 方法可以改变Surface最终显示的角度与缩放尺寸。



说明

二维变换矩阵是 3×3 的矩阵，其左上角的 2×2 子阵负责缩放与旋转。

·mDrawState保存了窗口的绘制状态。从窗口最初的创建，到最终得以显示到屏幕上，共经历了NO_SURFACE、DRAW_PENDING、COMMIT_DRAW_PENDING，READ_TO_SHOW和HAS_DRAWN 5个状态。后面将深入探讨这5个状态的迁移过程。

接下来将从开始动画及渲染一帧动画两个方面学习WSA的工作原理。

2. 开始窗口动画

(1) 动画的选择与设置

在WMS的relayoutWindow () 函数中，当窗口由不可见变为可见时，执行了下面一条语句：

```
winAnimator.applyEnterAnimationLocked();
```

WSA.applyEnterAnimationLocked () 将会为当前窗口开始一个淡入动画，将窗口显示出来。它调用了WSA中更为通用的启动窗口动画的函数applyAnimationLocked () 。

```
[WindowStateAnimator.java--
```

```
>WindowStateAnimator.applyAnimationLocked()] boolean  
applyAnimationLocked(int transit, boolean isEntrance) { // transit 参数用来
```

指定动画的意图，WSA将通过动画的意图寻找合适的动画资源 if (mService.okToDisplay()) { /* 首先尝试由WindowManagerPolicy获取合适的动画资源。WindowManagerPolicy的select- AnimationLw()为状态栏、导航栏等系统窗口分配了一些特殊的动画 */ int anim = mPolicy.selectAnimationLw(mWin, transit); int attr = -1; Animation a = null; if (anim != 0) { // 如果WMP为当前窗口指定了动画资源，则选择此动画 a = anim != -1 ? AnimationUtils.loadAnimation(mContext, anim) : null; } else { // 否则根据transit选择WSA默认的动画资源 switch (transit) { } if (attr >= 0) { a = mService.loadAnimation(mWin.mAttrs, attr); } } if (a != null) { // 保存选取的Animation对象 setAnimation(a); } } else { // 如果屏幕尚未准备好，则不进行任何动画 clearAnimation(); } return mAnimation != null; }

setAnimation () 的实现为：

[WindowStateAnimator.java-->WindowStateAnimator.setAnimation()]

```
public void setAnimation(Animation anim) { mAnimating = false;
mLocalAnimating = false; // 保存选择的anim到mAnimation mAnimation =
anim; ..... // 初始化WSA的Transformation mTransformation.clear(); /* 标记以下变量为true，表示在stepAnimation()时需要将WSA自身的
Transformation参与计算 当WSA本身没有动画时，完全可以保持
```

```
mTransformation为单位变换状态，并在stepAnimation()中不加判断地让  
其参与计算，但是会降低效率 */ mHasLocalTransformation = true; }
```

可以看出，WSA并没有给WMS非常大的自由去选择任意动画。WMS需要通过一个transit参数向WSA提出动画的意图，WMP和WSA再根据此意图选择一个既有动画保存在mAnimation中。



注意

判断一个窗口是否正在运行动画的方法是判断其对应的WSA的mAnimation成员是否为null。

仅仅如此尚不能让窗口动起来，因为WindowAnimator可能正处于空闲状态。即便WindowAnimator正在不停地工作，此WSA并没有位于DisplayContentsAnimator的mWindowAnimators列表中，因此其动画一样不能得到渲染。那怎么办呢？

(2) 从布局系统到动画系统

重新回到WMS的layoutWindow()函数，随后它调用了performLayoutAndPlaceSurfacesLocked()函数，发起了一次重新布局。在布局的最终阶段，布局系统调用updateLayoutToAnimationLocked()将包括WSA在内的所有Animator传递给了动画系统。

```
[WindowManagerService.java-->WindowManagerService.updateLayoutToAnimationLocked()] void updateLayoutToAnimationLocked() { // mLayoutToAnim是布局系统向动画系统传递信息的桥梁 final LayoutToAnimatorParams layoutToAnim = mLayoutToAnim; synchronized (layoutToAnim) { // WinAnimatorList其实就是ArrayList<WindowStateAnimator> SparseArray<WinAnimatorList> allWinAnimatorLists = layoutToAnim.mWinAnimatorLists; // 注意，每次布局系统向动画系统传递Animator时都会先将列表清空 allWinAnimatorLists.clear(); // 遍历所有的DisplayContent DisplayContentsIterator iterator = new DisplayContentsIterator(); while (iterator.hasNext()) { // 获取DisplayContent下的所有窗口列表windows ..... for (int i = 0; i < N; i++) { final WindowStateAnimator winAnimator = windows.get(i).mWinAnimator; /* 如果WSA拥有Surface，就将其添加到Animator列表中，为什么只要有Surface就会将其添加到列表中，而不需要判断WSA是否有动画在进行？因为WSA的地位不同，WSA即便自己没有动画，但它肩负着将AppWindowAnimator和ScreenRoatation-Animator的变换应用到窗口的Surface中的任务 */ if
```

```
(winAnimator.mSurface != null) { winAnimatorList.add(winAnimator); } }

// 保存Window列表

allWinAnimatorLists.put(displayContent.getDisplayId(), winAnimatorList);

} // 接下来还要向mLayoutToAnim添加壁纸动画相关的参数和

AppWindowAnimator ..... // 一切准备完毕后，安排渲染下一帧

scheduleAnimationLocked(); } }
```

在完成布局后，布局系统会将所有布局后的窗口的WSA保存至mLayoutToAnim对象中，然后安排动画系统开始处理下一帧动画的渲染。mLayoutToAnim就像一辆列车，载满了WSA驶向动画系统。

再次来到动画系统处理动画帧的入口mAnimationRunnable.run () 函数处，第一个调用的函数就是copyLayoutToAnimParamsLocked () 。这个函数将位于mLayoutToAnim中的WSA接下车，再保存到DisplayContentsAnimator的mWinAnimators列表中。

```
[WindowAnimator.java-->WindowAnimator.copyLayoutToAnimParamsLocked()]

private void copyLayoutToAnimParamsLocked() { final LayoutToAnimatorParams layoutToAnim = mService.mLayoutToAnim; synchronized(layoutToAnim)

{ // 从mLayoutToAnim中取出与Wallpaper及Dimming效果相关的参数

..... final int numDisplays = mDisplayContentsAnimators.size(); for (int i = 0; i < numDisplays; i++) { ..... displayAnimator.mWinAnimators.clear(); //
```

从mLayoutToAnim中取出所有保存的WSA，并悉数添加到DisplayContentsAnimator中

```
final WinAnimatorList winAnimators =
layoutToAnim.mWinAnimatorLists.get(displayId); if (winAnimators != null) { displayAnimator.mWinAnimators.addAll(winAnimators); } } // 取出 AppWindowAnimator并放到mAppAnimators列表中 ..... } }
```

经过一番从布局系统到动画系统的辗转，WSA终于出现在DispalyContentsAnimator的mWinAnimators列表中。在接下来的animateLocked () 中，WSA将会参与其中。

3.WSA的Transformation计算

回顾一下WindowAnimator.animateLocked () ，在完成AppWindowAnimator和Screen-RotationAnimator的Transformation计算后，通过执行performAnimationsLocked () 函数计算WSA的Transformation。

(1) performAnimationsLocked分析

[WindowAnimator.java-->WindowAnimator.performAnimationsLocked()]

```
private void performAnimationsLocked(final int displayId) { // 处理当前
DisplayContent所拥有的所有WSA
updateWindowsLocked(displayId); // 处理Wallpaper的动画，暂不讨论
updateWallpaperLocked(displayId); }
```

继续updateWindowsLocked () 的代码：

[WindowAnimator.java-->WindowAnimator.updateWindowsLocked()]

```
private void updateWindowsLocked(final int displayId) { final
WinAnimatorList winAnimatorList =
getDisplayContentsAnimatorLocked(displayId).mWinAnimators; ..... for
(int i = winAnimatorList.size() - 1; i >= 0; i--) { WindowStateAnimator
winAnimator = winAnimatorList.get(i); WindowState win =
winAnimator.mWin; final int flags = winAnimator.mAttrFlags; if
(winAnimator.mSurface != null) { ..... // 计算WSA的变换 final boolean
nowAnimating = winAnimator.stepAnimationLocked(mCurrentTime); /* 接
下来处理force hiding相关的逻辑。Force hiding 是指当WMP认为某些窗
口具有force hiding的特性时，所有位于其下的窗口都应被隐藏 */ ..... }
/* 处理WSA的绘制状态。当WSA的状态为READY_TO_SHOW时，执行
WSA.performShowLocked() 将状态切换为HAS_DRAWN */ final
AppWindowToken atoken = win.mAppToken; if (winAnimator.mDrawState
== WindowStateAnimator.READY_TO_SHOW) { if (atoken == null ||
atoken.allDrawn) { if (winAnimator.performShowLocked()) {
mPendingLayoutChanges.put(displayId,
 WindowManagerPolicy.FINISH_LAYOUT_REDO_ANIM); } } } /* 更新
AppWindowAnimator的thumbnailLayer。在遍历所有窗口的过程中，寻
找一个最靠前的 窗口的显示次序作为其窗口所属的AppWindowToken的
```

```
缩略图的显示次序 */ final AppWindowAnimator appAnimator =  
winAnimator.mAppAnimator; if (appAnimator != null &&  
appAnimator.thumbnail != null) { ..... } } .....
```

在这段代码中，和我们讨论的内容相关的有以下两点：

- 通过调用WSA的stepAnimationLocked () 更新其所维护的mTransformation。这个Transformation将在后面WSA.prepareSurfaceLocked () 的调用中影响Surface的最终位置。
- 如果窗口的绘制状态为READY_TO_SHOW，则通过调用performShowLocked () 函数将绘制状态改为HAS_DRAWN。拥有HAS_DRAWN状态的窗口在WSA.prepareSurfaceLocked () 函数中被显示出来。先记住这个操作，在后续的小节中将整理窗口的绘制状态的迁移以及其意义。

接下来看一下WSA的stepAnimationLocked () 是如何更新其Transformation的。

(2) stepAnimationLocked () 分析

```
[WindowStateAnimator.java-->WindowStateAnimator.stepAnimationLocked()] boolean  
stepAnimationLocked(long currentTime) { mWasAnimating = mAnimating;
```

```
if (mService.okToDisplay()) { if (mWin.isDrawnLw() && mAnimation !=  
null) { // mAnimation 不为null， 表示WSA拥有自己的动画， 因此需要计  
算其Transformation mHasTransformation = true;  
mHasLocalTransformation = true; // 如果此时mLocalAnimating为false，  
表示这是WSA的第一帧动画， 需要初始化其Animation if  
(!mLocalAnimating) { // 注意initialzied的参数， 动画对象的尺寸为窗口  
的mFrame的尺寸 // 而容器尺寸为mAnimDw和mAnimDh  
mAnimation.initialize(mWin.mFrame.width(), mWin.mFrame.height(),  
mAnimDw, mAnimDh); // mAnimDw与mAnimDh有可能在布局后处理中  
被改成窗口的移动量， 因此最好在其 // 完成mAnimation初始化后将其  
还原回屏幕上应用程序区的尺寸 final DisplayInfo displayInfo =  
mWin.mDisplayContent.getDisplayInfo(); mAnimDw =  
displayInfo.appWidth; mAnimDh = displayInfo.appHeight; // currentTime就  
是动画的起始时间 mAnimation.setStartTime(currentTime);  
mLocalAnimating = true; mAnimating = true; } if ((mAnimation != null)  
&& mLocalAnimating) { // 执行stepAnimation()函数更新mTransformation  
if (stepAnimation(currentTime)) { // 注意， 如果动画在持续，在这里就返  
回了 return true; } } /* 如果WSA没有动画或动画已经结束，则标记下  
述状态变量为false。 表示在计算Surface最终的变换时， WSA的  
mTransformation不参与计算 mHasLocalTransformation = false; } // 当执  
行到这里时， WSA的动画已经播放完毕， 接下来便是一些动画的清理
```

动作 // 将mLocalAnimating设置为false，这样当下次开始动画时，
就可以对Animation进行初始化了 mLocalAnimating = false; /* 回顾4.3节
最后关于mAnimLayer的讨论，在assignLayersLocked()中mAnimLayer可
能会被增加一个来自AppWindowAnimator的矫正。现在，动画结束
了，应该撤销这个矫正让窗口回到其本来的位置，也就是mLayer所指
定的位置 */ mAnimLayer = mWin.mLayer; // 返回false 表示动画已结束
return false; }

stepAnimationLocked () 将较多的经历花费在第一帧时动画的初始化
与完成最后一帧后的清理工作上。在动画持续的过程中，其工作还是
很简单的，就是调用WSA.stepAnimation () 函数以更新
Transformation，注意没有locked () 。

[WindowStateAnimator.java-->WindowStateAnimator.stepAnimation()]

```
private boolean stepAnimation(long currentTime) { if ((mAnimation == null) || !mLocalAnimating) { return false; } mTransformation.clear(); final boolean more = mAnimation.getTransformation(currentTime, mTransformation); return more; }
```

很简单，不是吗？更新Transformation调用Animation.getTransformation
() 函数即可。其原理在4.1节便已经介绍过。

4.WSA的动画渲染

经过stepAnimationsLocked () 之后，WSA的状态已经准备好正式渲染动画帧了。回到WindowAnimator的animateLocked () 函数，在完成performAnimationsLocked () 之后，再次遍历DisplayContent下的所有WSA，并分别执行它们的prepareSurfaceLocked () 。prepare-SurfaceLocked () 将完成动画帧的渲染动作。



注意

prepareSurfaceLocked () 是在众多Animator类中WSA所特有的函数。其他Animator的Transformation通过WSA的这个函数影响窗口动画的渲染。

(1) mShownFrame、Surface变换矩阵以及mShownAlpha的计算

prepareSurfaceLocked () 函数的第一个工作就是通过调用computeShownFrameLocked () 计算mShownFrame、Surface的变换矩阵以及mShownAlpha (Surface透明度) 的计算。我们可以将其称为渲染参数。

Android提供的动画工具类可以实现平移、旋转、缩放以及透明度4种动画。因此在窗口动画渲染过程中，需要提取这4种分量，并以此设置Surface的相关属性，从而实现动画帧的渲染。其中mShownFrame提取了窗口平移分量，Surface变换矩阵提取了旋转与缩放分量，mShownAlpha自然提取了透明度分量。注意mShownFrame仅仅提取了窗口平移分量，也就是说只有其left和top是变换后的结果，而其宽度与高度则与mFrame保持一致。

上述三个渲染参数的计算过程繁杂却不失规律性：将AppWindowAnimator、Screen-RotationAnimation、父窗口的WSA以及窗口自身的WSA的4个Transformation组合在一起，然后从组合后的矩阵中提取平移分量交给mShownFrame，将旋转与缩放子阵作为Surface的变换矩阵，再将4个Transformation的透明度分量相乘作为mShownAlpha。

下面分析computeShownFrameLocked () 的实现。



注意

这个函数应用了很多的矩阵乘法操作，注意矩阵乘法是不满足交换律的。例如一个平移矩阵T乘以一个缩放矩阵S的结果与S乘以T的结果是不一样的，因此矩阵乘法分为左乘和右乘两种，要格外注意操作顺序。在computeShownFrameLocked () 中只使用了右乘。

[WindowStateAnimator.java--

```
>WindowStateAnimator.computeShownFrameLocked() void  
computeShownFrameLocked() { final boolean selfTransformation =  
mHasLocalTransformation; /*获取父窗口与AppWindowAnimator的  
Transformation。如果窗口没有父窗口或不属于 AppWindowToken，亦  
或二者不处于动画过程中，则对应的Transformation为null */  
Transformation attachedTransformation = ..... Transformation  
appTransformation = ..... // 暂不考虑壁纸动画相关的内容 ..... // 获取窗  
口所在DisplayContent下的ScreenRotationAnimation final int displayId =  
mWin.getDisplayId(); final ScreenRotationAnimation  
screenRotationAnimation =  
mAnimator.getScreenRotationAnimationLocked(displayId); final boolean  
screenAnimation = ..... /* 目前，computeShownFrameLocked()收集了4种  
Animator的变换，如果只要有一种Animator 正在运行中，就需要整合  
这些Transformation进行mShownFrame的计算 if (selfTransformation ||  
attachedTransformation != null || appTransformation != null ||  
screenAnimation) { // 窗口的mFrame final Rect frame = mWin.mFrame; //
```

tmpFloats用于保存tmpMatrix的9个矩阵元素 final float tmpFloats[] = mService.mTmpFloats; // 二维变换矩阵，用于存储Transformation矩阵分量的组合结果 final Matrix tmpMatrix = mWin.mTmpMatrix; // ① 首先将窗口缩放添加到结果矩阵中。postScale()函数用于右乘一个缩放矩阵 tmpMatrix.postScale(mWin.mGlobalScale, mWin.mGlobalScale); // ② 将WSA自身的Transformation添加到结果矩阵中。postConcat()用于右乘任意矩阵 if (selfTransformation) { tmpMatrix.postConcat(mTransformation.getMatrix()); } // ③ 将窗口的实际位置添加到结果矩阵中。postTranslate()函数用于右乘一个平移矩阵 tmpMatrix.postTranslate(frame.left + mWin.mXOffset, frame.top + mWin.mYOffset); // ④ 将父窗口的动画变换添加到结果矩阵中 if (attachedTransformation != null) { tmpMatrix.postConcat(attachedTransformation.getMatrix()); } // ⑤ 将AppWindowAnimator的变换添加到结果矩阵中 if (appTransformation != null) { tmpMatrix.postConcat(appTransformation.getMatrix()); } /* ⑥ 将UniverseBackground的变换添加到结果矩阵中。UniverseBackground是类型为TYPE_UNIVERS_BACKGROUND的窗口，由SystemUI实现。虽然它与其他窗口并没有什么实际的从属关系，但是Android将其作为所有真实窗口的容器，或者如其名字所说，它是Android显示世界的宇宙。将这个变换添加到结果矩阵的意义是：既然容器移动了，内部的所有窗口都要跟着移动 */ // ⑦ 将ScreenRotationAnimation的变换加

入结果矩阵中 if (screenAnimation) { tmpMatrix.postConcat(
screenRotationAnimation.getEnterTransformation().getMatrix()); } // ⑧ 将
放大镜效果的变换加入结果矩阵中 MagnificationSpec spec =
mWin.getWindowMagnificationSpecLocked(); if (spec != null &&
!spec.isNop()) { tmpMatrix.postScale(spec.mScale, spec.mScale);
tmpMatrix.postTranslate(spec.mOffsetX, spec.mOffsetY); } /* 结果矩阵已
经计算完成，提取它们的缩放旋转子阵作为Surface的变换矩阵，并将
平移分量保 存到mShownFrame中 */ mHaveMatrix = true; // 将结果矩阵
的元素放到tmpFloats中 tmpMatrix.getValues(tmpFloats); // 保存Surface的
变换矩阵 mDsDx = tmpFloats[Matrix.MSCALE_X]; mDtDx =
tmpFloats[Matrix.MSKEW_Y]; mDsDy = tmpFloats[Matrix.MSKEW_X];
mDtDy = tmpFloats[Matrix.MSCALE_Y]; // 计算mShownFrame，注意只
有left和top分量取自变换结果， width和height同mFrame float x =
tmpFloats[Matrix.MTRANS_X]; float y = tmpFloats[Matrix.MTRANS_Y];
int w = frame.width(); int h = frame.height(); mWin.mShownFrame.set(x, y,
x+w, y+h); /* 接下来计算mShownAlpha。注意，由于对支持透明绘制的
Surface的透明度是通过软件 实现的而不是硬件加速，因此为了保证动
画的流畅度，正在做平移/缩放/旋转变换的支持透 明绘制的Surface将不
做透明度的变换，以省去通过软件渲染透明度动画的时间进而保证其
他动画的流畅。在framework/base/core/res/res/values/config.xml 中将
config_sf_limitedAlpha设置为false可以取消这个限制 */ mShownAlpha =

```
mAlpha; if (!mService.mLimitedAlphaCompositing ||
(!PixelFormat.formatHasAlpha(mWin.mAttrs.format) ||
(mWin.isIdentityMatrix(mDsDx, mDtDx, mDsDy, mDtDy) &&x ==
frame.left && y == frame.top))) { // 透明度是通过乘法混合的 if
(selfTransformation) { mShownAlpha *= mTransformation.getAlpha(); } if
(attachedTransformation != null) { mShownAlpha *=
attachedTransformation.getAlpha(); } if (appTransformation != null) {
mShownAlpha *= appTransformation.getAlpha(); } if
(mAnimator.mUniverseBackground != null) { mShownAlpha *=
mAnimator.mUniverseBackground.mUniverseTransform.getAlpha(); } if
(screenAnimation) { mShownAlpha *=
screenRotationAnimation.getEnterTransformation().getAlpha(); } } else { }
return; } /* 当没有与窗口相关的动画在运行时，仍然要设置
UniversBackground以及放大镜效果，其原理一样，这里不再赘述 */
..... }
```

注意变换的顺序，它们是不能轻易改动的，因为矩阵乘法不满足交换律。为什么Android要采用这个顺序呢？将参与变换的对象按照顺序排列出来：动画窗口，动画窗口的父窗口，动画窗口所属的Activity，UniverseBackground，屏幕旋转。不难发现规律，它们是按照从属关系由小到大排列的。这是因为Android希望父窗体的动画应用于所有子对象。例如，当使用上述变换顺序，子窗口的变换矩阵为A而父窗口的变

换矩阵为B时，子窗口的最终变换为AB，父窗口的最终变换仍然是B。当子窗口的变换固定，无论父窗口的变换B为任何值，子窗口相对于父窗口的变换固定为A，也就是说父窗口做任何动画，子窗口都会如影随形地相对于父窗口保持静止。因此，倘若读者需要增加一个ActivityGroundAnimator，那么这个变换应插入AppWindowAnimator与UniverseBackground之间。

至此，mShownFrame、Surface的变换矩阵与透明度计算完毕。接下来将它们设置给Surface。

(2) 设置变换到Surface

看下prepareSurfaceLocked()的实现：

```
[WindowStateAnimator.java-->WindowStateAnimator.prepareSurfaceLocked()]
public void
prepareSurfaceLocked(final boolean recoveringMemory) { final
WindowState w = mWin; // 计算渲染参数 computeShownFrameLocked();
/* 还是setSurfaceBoundariesLocked()，将计算好的mShownFrame设置为
Surface的位置，完成平移操作。注意，尺寸仍为mFrame的尺寸。动画
所要求的缩放变换将通过变换矩阵完成 */
setSurfaceBoundariesLocked(recoveringMemory); if (...) { ..... } else if
(mLastLayer != mAnimLayer // 如果Surface参数发生了变化 ||
```

```
mLastAlpha != mShownAlpha || mLastDsDx != mDsDx|| mLastDtDx !=  
mDtDx || mLastDsDy != mDsDy || mLastDtDy != mDtDy || w.mLastHScale  
!= w.mHScale || w.mLastVScale != w.mVScale || mLastHidden) { if  
(mSurface != null) { try { // 设置透明度 mSurfaceAlpha = mShownAlpha;  
mSurface.setAlpha(mShownAlpha); // 设置窗口的显示次序  
mSurfaceLayer = mAnimLayer; mSurface.setLayer(mAnimLayer); // 设置  
Surface的变换矩阵，缩放与旋转变换 mSurface.setMatrix(  
mDsDx*w.mHScale, mDtDx*w.mVScale, mDsDy*w.mHScale,  
mDtDy*w.mVScale); // 如果窗口的绘制状态是HAS_DRAWN， 并且尚  
未显示，则显示Surface // showSurfaceRobustlyLocked()将通过  
Surface.show()将窗口显示出来 if (mLastHidden && mDrawState ==  
HAS_DRAWN) { if (showSurfaceRobustlyLocked()) {...} else {.....} } .....  
} catch (RuntimeException e) { ..... } } ..... }
```

很简单，上文提到的三个渲染参数都被设置到了Surface。用户可以看到，在这一帧中，窗口的位置发生了变化！

5. 窗口的绘制状态与从新建到显示的过程

在布局系统与动画系统中曾多次看到窗口的绘制状态这个概念。窗口绘制状态的迁移体现了窗口从最初的创建到显示在屏幕的过程。这一小节将介绍这个状态的一些细节。

第一个状态：NO_SURFACE。窗口的绘制状态保存在WSA.mDrawState中。当一个窗口刚刚被WMS的addWindow () 函数创建时，WSA在WindowState的构造函数中被一并创建。此时，窗口的绘制状态为NO_SURFACE，因为在relayoutWindow () 之前，窗口是没有Surface的，当然也不可能显示出来。

第二个状态：DRAW_PENDING。随后，客户端调用了relayoutWindow () ，此时WMS通过WSA的createSurfaceLocked () 为窗口创建了一块Surface。此时窗口的绘制状态被设置为DRAW_PENDING。也就是说，窗口正拥有一块空白的Surface，此时需要客户端在Surface上作画。由于Surface仍是空白状态，因此此时仍不能让窗口显示出来。

第三个状态：COMMIT_DRAW_PENDING。回顾SampleWindow的例子，在通过Canvas完成在Surface上的绘制之后，调用IWindowSession.finishDrawing () 函数，通知WMS客户端已经完成在Surface上的绘制。此时，窗口的绘制状态便成为COMMIT_DRAW_PENDING，意思是窗口的绘制已经完成，正在等待由布局系统进行提交，窗口距离显示在屏幕上已经进了一步。

第四个状态：READY_TO_SHOW。之后工作就比较多了，WMS会继续调用performLayoutAndPlaceSurfacesLocked () 启动一次重新布局。正如在布局后处理中所看到的，会调用WSA的commitFinishDrawingLocked () 。如果窗口的状态为

COMMIT_DRAW_PENDING时，窗口的状态会再迁移到READ_TO_SHOW，此时窗口距离最终显示已经很接近了。READ_TO_SHOW表示窗口可以随时被显示，但是为什么不直接将其显示出来呢？因为窗口可能属于某一个AppWindowToken，Android希望当AppWindowToken所有的窗口都已准备好后再将它们一并显示出来。当然，如果窗口不属于AppWindowToken，或者AppWindowToken下的所有窗口都已准备好显示或已经显示，commitFinishDrawingLocked () 会立刻调用performShowLocked ()，进行绘制状态的下一步迁移。

第五个状态：HAS_DRAWN。performShowLocked () 在布局系统中被commitFinish-DrawingLocked () 调用，也可能在动画系统中被WindowAnimator的updateWindowsLocked () 在处理动画帧的过程中调用。performShowLocked () 会将窗口的绘制状态进一步迁移为最终状态HAS_DRAWN。拥有这个状态的窗口距离显示已经无限接近了。

最后，在WSA的prepareSurfaceLocked () 中，处于HAS_DRAWN状态却未被显示的窗口通过showSurfaceRobustlyLocked () 完成最终显示。

值得一提的是，当窗口的旋转方向发生变化后，窗口的状态会被重置为DRAW_PENDING，表示窗口必须重新绘制自己的内容。完成之后，再一步一步按照上述状态迁移将新的内容显示出来。

4.5.5 交替运行的布局系统与动画系统

动画系统在处理一帧动画时，同时保持着WMS.mWindowMap以及WMS.mAnimator两个锁。而布局系统则保持着WMS.mWindowMap锁，因此，布局系统的布局过程与动画帧处理过程是互斥的。然而，动画帧处理和布局过程可能是交替的。

1.从布局系统到动画系统

布局的最终阶段，也就是performLayoutAndPlaceSurfacesLocked () 的最后位置，会执行一个名为updateLayoutToAnimLocked () 函数。这点在“开始窗口动画”中已经介绍过。此函数会刷新动画系统需要在下一帧执行的Animator列表。因此在上一帧还在运行的Animator，经过一次布局后，下一帧可能就忽然不见了，同时，新的Animator有可能被加入列表中并参与下一帧的绘制过程。这个Animator列表被保存在WMS.mLayoutToAnim中。updateLayoutToAnimLocked () 立刻向WindowAnimator发出处理下一帧动画的命令。在开始处理下一帧动画时，WindowAnimator会将Animator列表从mLayoutToAnim取出，并逐个处理。

注意，updateLayoutToAnimLocked () 和发送处理下一帧动画的命令是无条件的。也就是说，只要进行了一次重新布局，必然会“惊扰”动画系统。

2.从动画系统到布局系统

在分析WindowAnimator的animateLocked () 函数时，曾经说明过mBulkUpdateParams和mPendingLayoutChanges成员变量以及updateAnimToLayoutLocked () 函数。

在动画的过程中，某些动画的开始或结束，或者某些操作对布局系统来说有着特别的意义，需要布局系统对此做出反应。此时可以将需要布局系统做出变化的要求或状态存入mPendingLayoutChanges以及mBulkUpdateParams中。mPendingLayoutChanges保存了动画系统要求重新布局DisplayContent时所要做的更改，也就是pendingLayoutChanges，而mBulkUpdateParams主要收集了要求修改保留在mInnerFileds的状态的请求。了解布局系统的工作原理后，读者对这两个概念应该很熟悉了。

在处理完一帧后，上述的两个变量便完成了信息的收集工作。在animateLockd () 函数的末尾处会调用updateAnimToLayoutLocked () ，这将它们保存在WindowaAnimator.mAnimToLayout中并传递给布局系统。布局系统解析这两个变量收集的请求，并检查这些请求是否需要进行重新布局。如果需要，则调用performLayoutAndPlaceSurfacesLocked () 开始布局。

因此，窗口布局与动画帧处理是交替运行的。但是最终一定会以布局系统解析mBulkUpdateParams与mPendingLayoutChanges时发现无须重新布局为交替运行的终点。

4.5.6 动画系统总结

这一节以WindowAnimator与WindowStateAnimator为例对WMS的动画系统做了介绍。

WindowAnimator是一个强大的驱动器，在它的控制下，多种类型的Animator有条不紊地完成各自的渲染工作。

在WindowAnimator之下有各种类型的Animator，分别掌管不同类型对象的动画。它们是WindowStateAnimator、AppWindowAnimator、ScreenRotationAnimation、DimAnimator以及一个名不副实的DisplayContentsAnimator。除DimAnimator与DisplayContentAnimator以外，其他类型的Animator都有一个stepAnimationLocked（）函数用以计算当前时间下动画对象所需的变换。

虽然受到WindowAnimator的管理，但WindowStateAnimator的重要性却非常重要，因为它不仅要进行动画变换的计算，还要管理窗口的Surface，并且所有其他类型的Animator的变换都要汇集到WindowStateAnimator中完成窗口最终变换的计算。在

`prepareSurfaceLocked ()` 中，`WindowStateAnimator`完成所有相关
`Animator`的变换的组合过程，并将变换结果设置到`Surface`中。

剩余的`AppWindowAnimator`、`ScreenRotationAnimation`以及
`DimAnimator`等`Animator`的工作原理与`WindowStateAnimator`类似，而且
更加简单，读者在经过本节学习后应该可以较快地完成对它们的研
究。

4.6 本章小结

漫长的WMS之旅就此告一段落。这一章首先讨论了WMS的窗口管理结构，然后详细地分析了WMS的布局和动画两个系统的工作原理。这三部分构成了WMS完成所有工作的基础。类似于屏幕旋转等功能性的内容在本章中并没有探讨，因为这些内容不过是在此基础之上的应用，完成本章的学习之后，扩展到WMS的其他内容也就不难了，所以这部分便留给读者研究吧。

另外，在分析的过程中，本章忽略了和壁纸及输入事件相关内容，它们将在第5章以及第6章进行分析。另外，状态栏的实现在WMS中也有一部分内容值得进一步探讨。因此在后面的几章中，仍会再回到WMS中来，重拾被本章忽略的有价值的内容。

本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供

最新最全的优质电子书下载！！！

第5章 深入理解Android输入系统

本章主要内容：

- 研究输入事件从设备节点开始到窗口处理函数的流程
- 介绍原始输入事件的读取与加工的原理
- 研究事件派发机制
- 讨论事件在输入系统与窗口之间传递与反馈的过程
- 介绍焦点窗口的选择、ANR的产生以及以软件方式模拟用户操作的原理

本章涉及的源代码文件名及位置：

·SystemServer.java

frameworks\base\services\java\com\android\server\SystemServer.java

·InputManagerService.java

frameworks\base\services\java\com\android\server\input\InputManagerService.java

· WindowManagerService.java

frameworks\base\services\java\com\android\server\wm\WindowManagerService.java

·WindowState.java

frameworks\base\services\java\com\android\server\wm\WindowState.java

·InputMonitor.java

frameworks\base\services\java\com\android\server\wm\InputMonitor.java

·InputEventReceiver.java

frameworks\base\core\java\android\view\InputEventReceiver.java

·com_android_server_input_InputManagerService.cpp

frameworks\base\services\jni\com_android_server_input_InputManagerService.cpp

·android_view_InputEventReceiver.cpp

frameworks\base\core\jni\android_view_InputEventReceiver.cpp

·InputManager.cpp

frameworks\base\services\input\InputManager.cpp

·EventHub.cpp

frameworks\base\services\input\EventHub.cpp

·EventHub.h

frameworks\base\services\input\EventHub.h

·InputDispatcher.cpp

frameworks\base\services\input\InputDispatcher.cpp

·InputDispatcher.h

frameworks\base\services\input\InputDispatcher.h

·InputTransport.cpp

frameworks\base\libs\androidfw\InputTransport.cpp

·InputTransport.h

frameworks\base\include\androidfw\InputTransport.h

5.1 初识Android输入系统

第4章通过分析WMS详细讨论了Android的窗口管理、布局及动画的工作机制。窗口不仅是内容绘制的载体，同时也是用户输入事件的目标。本章将详细讨论Android输入系统的工作原理，包括输入设备的管理、输入事件的加工方式以及派发流程。因此本章的探讨对象有两个：输入设备和输入事件。

触摸屏与键盘是Android最普遍也是最标准的输入设备。其实Android所支持的输入设备的种类不止这两个，鼠标、游戏手柄均在内建的支持之列。当输入设备可用时，Linux内核会在/dev/input/下创建对应的名为event0~n或其他名称的设备节点。而当输入设备不可用时，则会将对应的节点删除。在用户空间可以通过ioctl的方式从这些设备节点中获取其对应的输入设备的类型、厂商、描述等信息。

当用户操作输入设备时，Linux内核接收到相应的硬件中断，然后将中断加工成原始的输入事件数据并写入其对应的设备节点中，在用户空间可以通过read（）函数将事件数据读出。

Android输入系统的工作原理概括来说，就是监控/dev/input/下的所有设备节点，当某个节点有数据可读时，将数据读出并进行一系列的翻译加工，然后在所有的窗口中寻找合适的事件接收者，并派发给它。

以Nexus 4为例，其/dev/input/下有evnet0~5六个输入设备的节点。它们都是什么输入设备呢？用户的一次输入操作会产生什么样的事件数据呢？获取答案的最简单的办法就是用getevent与sendevent工具。

5.1.1 getevent与sendevent工具

Android系统提供了getevent与sendevent两个工具供开发者从设备节点中直接读取输入事件或写入输入事件。

getevent监听输入设备节点的内容，当输入事件被写入节点时，getevent会将其读出并打印在屏幕上。由于getevent不会对事件数据做任何加工，因此其输出的内容是由内核提供的最原始的事件。其用法如下：

```
adb shell getevent[-选项] [device_path]
```

其中，device_path是可选参数，用以指明需要监听的设备节点路径。如果省略此参数，则监听所有设备节点的事件。

打开模拟器，执行adb shell getevent -t（-t参数表示打印事件的时间戳），并按一下电源键（不要松手），可以得到以下一条输出，输出的部分数值会因机型的不同而有所差异，但格式一致：

```
[ 1262.443489] /dev/input/event0: 0001 0074 00000001
```

松开电源键时，又会产生以下一条输出：

```
[ 1262.557130] /dev/input/event0: 0001 0074 00000000
```

这两条输出便是按下和抬起电源键时由内核生成的原始事件。注意其输出是十六进制的。每条数据有5项信息：产生事件时的时间戳（[1262.443489]）、产生事件的设备节点（/dev/input/event0）、事件类型（0001）、事件代码（0074）以及事件的值（00000001）。其中时间戳、类型、代码、值便是原始事件的4项基本元素。除时间戳外，其他三项元素的实际意义依照设备类型及厂商的不同而有所区别。在本例中，类型0x01表示此事件为一条按键事件，代码0x74表示电源键的扫描码，值0x01表示按下，0x00则表示抬起。这两条原始数据被输入系统包装成两个KeyEvent对象，作为两个按键事件派发给Framework中感兴趣的模块或应用程序。



注意

一个原始事件所包含的信息量是比较有限的。而在Android API中所使用的某些输入事件，如触摸屏点击/滑动，其中包含了很多的信息，如

XY坐标、触摸点索引等，其实是输入系统整合了多个原始事件后的结果。这个过程将在5.2.4节中详细探讨。

为了对原始事件有一个感性的认识，读者可以在运行getevent的过程中尝试一下其他的输入操作，观察一下每种输入所对应的设备节点及4项元素的取值。

输入设备的节点不仅在用户空间可读，而且是可写的，因此可以将原始事件写入节点中，从而实现模拟用户输入的功能。sendevent工具的作用正是如此。其用法如下：

```
sendevent <节点路径> <类型> <代码> <值>
```

可以看出，sendevent的输入参数与getevent的输出是对应的，只不过sendevent的参数为十进制。电源键的代码0x74的十进制为116，因此可以通过快速执行如下两个命令实现点击电源键的效果：

```
adb shell sendevent /dev/input/event0 1 116 1 #按下电源键 adb shell  
sendevent /dev/input/event0 1 116 0 #抬起电源键
```

执行完这两个命令后，可以看到设备进入休眠或被唤醒，与按下实际电源键的效果一模一样。另外，执行这两个命令的时间间隔便是用户按住电源键所保持的时间，所以如果执行第一个命令后迟迟不执行第二个，则会产生长按电源键的效果——出现关机对话框。很有趣不是

吗？输入设备节点在用户空间可读可写的特性为自动化测试提供了一条高效的途径。

现在，读者对输入设备节点以及原始事件有了直观认识，接下来看一下Android输入系统的基本原理。

5.1.2 Android输入系统简介

上一节讲述了输入事件的源头是位于`/dev/input/`下的设备节点，而输入系统的终点是由WMS管理的某个窗口。最初的输入事件为内核生成的原始事件，而最终交付给窗口的则是`KeyEvent`或`MotionEvent`对象。因此Android输入系统的主要工作是读取设备节点中的原始事件，将其加工封装，然后派发给一个特定的窗口以及窗口中的控件。这个过程由`InputManagerService`（以下简称IMS）系统服务为核心的多个参与者共同完成。

输入系统的总体流程和参与者如图5-1所示。

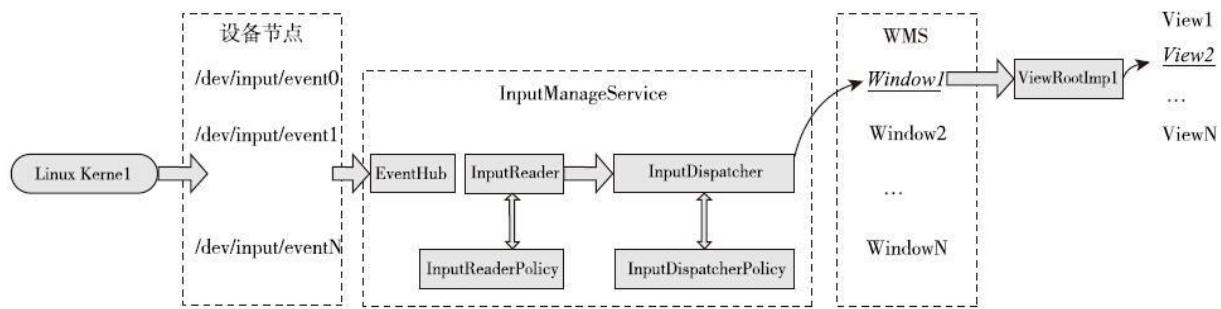


图5-1 输入系统的总体流程与参与者

图5-1描述了输入事件的处理流程以及输入系统中最基本的参与者。它们是：

- Linux内核，接受输入设备的中断，并将原始事件的数据写入设备节点中。
- 设备节点，作为内核与IMS的桥梁，它将原始事件的数据暴露给用户空间，以便IMS可以从中读取事件。
- InputManagerService，一个Android系统服务，它分为Java层和Native层两部分。Java层负责与WMS通信。而Native层则是InputReader和InputDispatcher两个输入系统关键组件的运行容器。
- EventHub，直接访问所有的设备节点。并且正如其名字所描述的，它通过一个名为getEvents () 的函数将所有输入系统相关的待处理的底层事件返回给使用者。这些事件包括原始输入事件、设备节点的增删等。
- InputReader，是IMS中的关键组件之一。它运行于一个独立的线程中，负责管理输入设备的列表与配置，以及进行输入事件的加工处理。它通过其线程循环不断地通过getEvents () 函数从EventHub中将事件取出并进行处理。对于设备节点的增删事件，它会更新输入设备列表与配置。对于原始输入事件，InputReader对其进行翻译、组装、

封装为包含更多信息、更具可读性的输入事件，然后交给 InputDispatcher进行派发。

· InputReaderPolicy，它为InputReader的事件加工处理提供一些策略配置，例如键盘布局信息等。

· InputDispatcher，是IMS中的另一个关键组件。它也运行于一个独立的线程中。InputDispatcher中保管了来自WMS的所有窗口的信息，其收到来自InputReader的输入事件后，会在其保管的窗口中寻找合适的窗口，并将事件派发给此窗口。

· InputDispatcherPolicy，它为InputDispatcher的派发过程提供策略控制。例如截取某些特定的输入事件用作特殊用途，或者阻止将某些事件派发给目标窗口。一个典型的例子就是HOME键被InputDispatcherPolicy 截取到PhoneWindowManager中进行处理，并阻止窗口收到HOME键按下的事件。

· WMS，虽说不是输入系统中的一员，但是它却对InputDispatcher的正常工作起到了至关重要的作用。当新建窗口时，WMS为新窗口和IMS 创建了事件传递所用的通道。另外，WMS还将所有窗口的信息，包括窗口的可点击区域、焦点窗口等信息，实时地更新到IMS的 InputDispatcher中，使得InputDispatcher可以正确地将事件派发到指定的窗口。

`·ViewRootImpl`, 对某些窗口, 如壁纸窗口、`SurfaceView`的窗口来说, 窗口就是输入事件派发的终点。而对其他的如Activity、对话框等使用了Android控件系统的窗口来说, 输入事件的终点是控件 (View) 。`ViewRootImpl`将窗口所接收的输入事件沿着控件树将事件派发给感兴趣的控件。

简单来说, 内核将原始事件写入设备节点中, `InputReader`不断地通过 `EventHub` 将原始事件取出来并翻译加工成Android输入事件, 然后交给 `InputDispatcher`。`InputDispatcher`根据WMS提供的窗口信息将事件交给合适的窗口。窗口的 `ViewRootImpl` 对象再沿着控件树将事件派发给感兴趣的控件。控件对其收到的事件做出响应, 更新自己的画面、执行特定的动作。所有这些参与者以IMS为核心, 构建了Android庞大而复杂的输入体系。

Linux内核对硬件中断的处理超出了本书的讨论范围, 因此本章将以 IMS为重点, 详细讨论除Linux内核以外的其他参与者的工作原理。

5.1.3 IMS的构成

同以往一样, 本节通过IMS的启动过程探讨IMS的构成。上一节提到, IMS分为Java层与Native层两个部分, 其启动过程是从Java部分的初始化开始, 进而完成Native部分的初始化。

1.IMS的诞生

同其他系统服务一样，IMS在SystemServer中的ServerThread线程中启动。

```
[SystemServer.java-->ServerThread.run()] public void run() { .....  
InputManagerService inputManager = null; ..... // ① 新建IMS对象。注意  
第二个参数wmHandler，这说明IMS的一部分功能可能会在WMS的线  
程中完成 inputManager = new InputManagerService(context, wmHandler);  
// 将IMS发布给ServiceManager，以便其他人可以访问IMS提供的接口  
ServiceManager.addService(Context.INPUT_SERVICE, inputManager); //  
设置向WMS发起回调的callback对象  
inputManager.set WindowManagerCallbacks(wm.getInputMonitor()); // ②  
正式启动IMS inputManager.start(); ..... /* 设置IMS给  
DisplayManagerService。DisplayManagerService将会把屏幕的信息发送  
给输入系统作为事件加工的依据。在5.2.4节将会讨论到这些信息的作  
用 */ display.setInputManager(inputManager); }
```

IMS的诞生分为两个阶段：

- 创建新的IMS对象。
- 调用IMS对象的start () 函数完成启动。

(1) IMS的创建

IMS的构造函数如下：

```
[InputManagerService.java-->InputManagerService.InputManagerService()]
public InputManagerService(Context context, Handler handler) { /* 使用
wmHandler的Looper新建一个InputManagerHandler。
InputManagerHandler将运行在WMS 的主线程中*/
    this.mHandler = new
    InputManagerHandler(handler.getLooper()); ..... // 每一个分为Java和
Native两部分的对象在创建时都会有一个nativeInput函数
    mPtr =
    nativeInit(this, mContext, mHandler.getLooper().getQueue()); }
```

可以看出，IMS的构造函数非常简单。看来绝大部分的初始化工作都位于Native层。参考nativeInit（）函数的实现。

```
[com_android_server_input_InputManagerService.cpp-->nativeInit()]
static
jint nativeInit(JNIEnv* env, jclass clazz, jobject serviceObj, jobject
contextObj, jobject messageQueueObj) { sp<MessageQueue> messageQueue =
    android_os_MessageQueue_getMessageQueue(env, messageQueueObj); /*
新建了一个NativeInputManager对象， NativeInputManager， 此对象将是
Native层组件与Java 层IMS进行通信的桥梁 */
    NativeInputManager* im =
    new NativeInputManager(contextObj, serviceObj, messageQueue-
    >getLooper()); im->incStrong(serviceObj); // 返回NativeInputManager对象
    的指针给Java层的IMS， IMS将其保存在mPtr成员变量中
    return reinterpret_cast<IMService>(im); }
```

nativeInit () 函数创建了一个类型为NativeInputManager的对象，它是 Java层与Native层互相通信的桥梁。

看下这个类的声明可以发现，它实现了InputReaderPolicyInterface与 InputDispatcher-PolicyInterface两个接口。这说明上一节曾经介绍过的两个重要的输入系统参与者InputReaderPolicy和InputDispatcherPolicy是由 NativeInputManager实现的，然而它仅仅为两个策略提供接口实现而已，并不是策略的实际实现者。NativeInputManager通过JNI回调Java层的IMS，由它完成决策。本节暂不讨论其实现细节，读者只要先记住两个策略参与者的接口实现位于NativeInputManager即可。

接下来看一下NativeInputManager的创建：

```
[com_android_server_input_InputManagerService.cpp --
>NativeInputManager::NativeInputManager()
NativeInputManager::NativeInputManager(jobject contextObj, jobject
serviceObj, const sp & looper) : mLooper(looper) { ..... // 出现重点了,
NativeInputManager创建了EventHub sp eventHub = new EventHub(); // 接
着创建了Native层的InputManager mInputManager = new
InputManager(eventHub, this, this); }
```

在NativeInputManager的构造函数中，创建了两个关键人物，分别是 EventHub与InputManager。EventHub复杂的构造函数使其在创建后便拥

有了监听设备节点的能力，本节中暂不讨论它的构造函数，读者仅需知道EventHub在这里初始化即可。紧接着便是InputManager的创建了，看一下其构造函数：

[InputManager.cpp-->InputManager::InputManager()]

```
InputManager::InputManager( const sp & eventHub, const sp &
readerPolicy, const sp & dispatcherPolicy) { // 创建InputDispatcher
mDispatcher = new InputDispatcher(dispatcherPolicy); // 创建 InputReader
mReader = new InputReader(eventHub, readerPolicy, mDispatcher); // 初始化
initialize(); }
```

再看initialize () 函数：

[InputManager.cpp-->InputManager::initialize()] void

```
InputManager::initialize() { // 创建供InputReader运行的线程
InputReaderThread mReaderThread = new InputReaderThread(mReader); //
创建供InputDispatcher运行的线程InputDispatcherThread
mDispatcherThread = new InputDispatcherThread(mDispatcher); }
```

InputManager的构造函数也比较简洁，它创建了4个对象，分别为IMS的核心参与者InputReader与InputDispatcher，以及它们所在的线程InputReaderThread与InputDispatcherThread。注意InputManager的构造

函数的参数readerPolicy与dispatcherPolicy，它们都是NativeInputManager。

至此，IMS的创建已完成。在这个过程中，输入系统的重要参与者均完成创建，并得到了如图5-2所描述的一套体系。

(2) IMS的启动与运行

完成IMS的创建之后，ServerThread执行InputManagerService.start () 函数以启动IMS。InputManager的创建过程分别为InputReader与InputDispatcher创建了承载它们运行的线程，然而并未启动这两个线程，因此IMS的各员大将仍处于待命状态。此时start () 函数的功能就是启动这两个线程，使得InputReader与InputDispatcher开始工作。

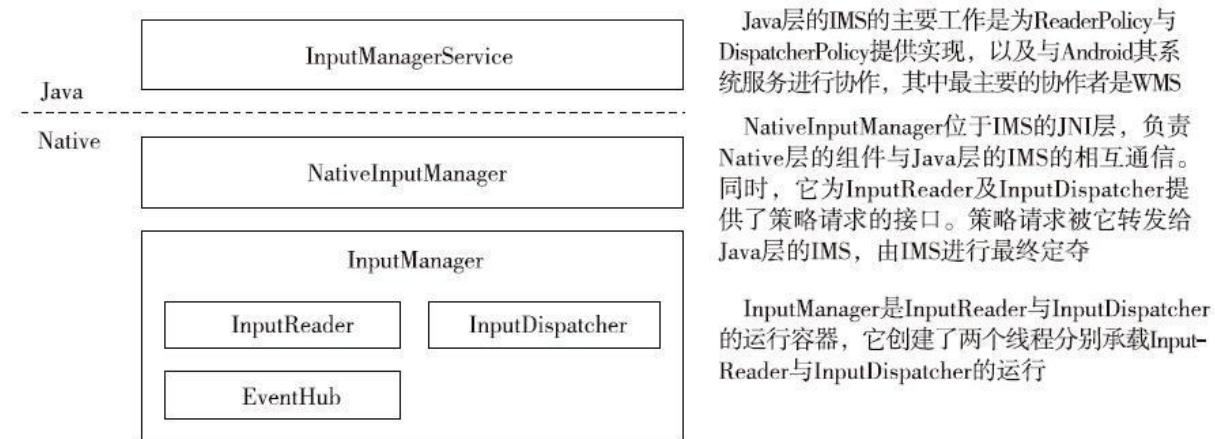


图5-2 IMS的结构体系

当两个线程启动后。InputReader在其线程循环中不断地从EventHub中抽取原始输入事件，进行加工处理后将加工所得的事件放入InputDispatcher的派发队列中。InputDispatcher则在其线程循环中将派发队列中的事件取出，查找合适的窗口，将事件写入窗口的事件接收管道中。窗口事件接收线程的Looper从管道中将事件取出，交由事件处理函数进行事件响应。整个过程共有三个线程首尾相接，像三台水泵似的一层层地将事件交付给事件处理函数。如图5-3所示。

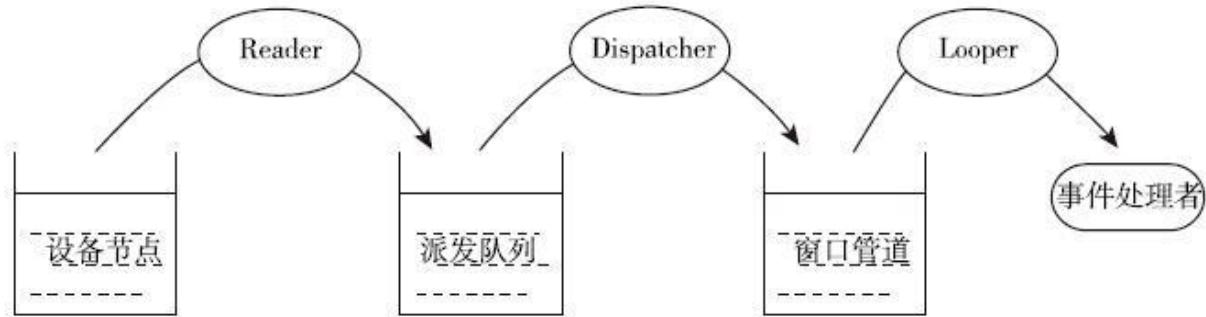


图5-3 三个线程，三台水泵

InputManagerService.start () 函数的作用，就像为Reader线程、Dispatcher线程这两台水泵按下开关，而Looper这台水泵在窗口创建时便已经处于运行状态了。自此，输入系统动力十足地开始运转，设备节点中的输入事件将被源源不断地抽取给事件处理者。本章的主要内容便是讨论这三台水泵的工作原理。

2.IMS的成员关系

根据对IMS的创建过程的分析，可以得到IMS的成员关系如图5-4所示，这幅图省略了一些非关键的引用与继承关系。



注意

IMS内部做了很多的抽象工作，EventHub、InputReader以及InputDispatcher等实际上都继承自相应的名为XXXInterface的接口，并且仅通过接口进行相互之间的引用。鉴于这些接口各自仅有唯一的实现，为了简化叙述我们将不提及这些接口，但是读者在实际学习与研究时需要注意这一点。

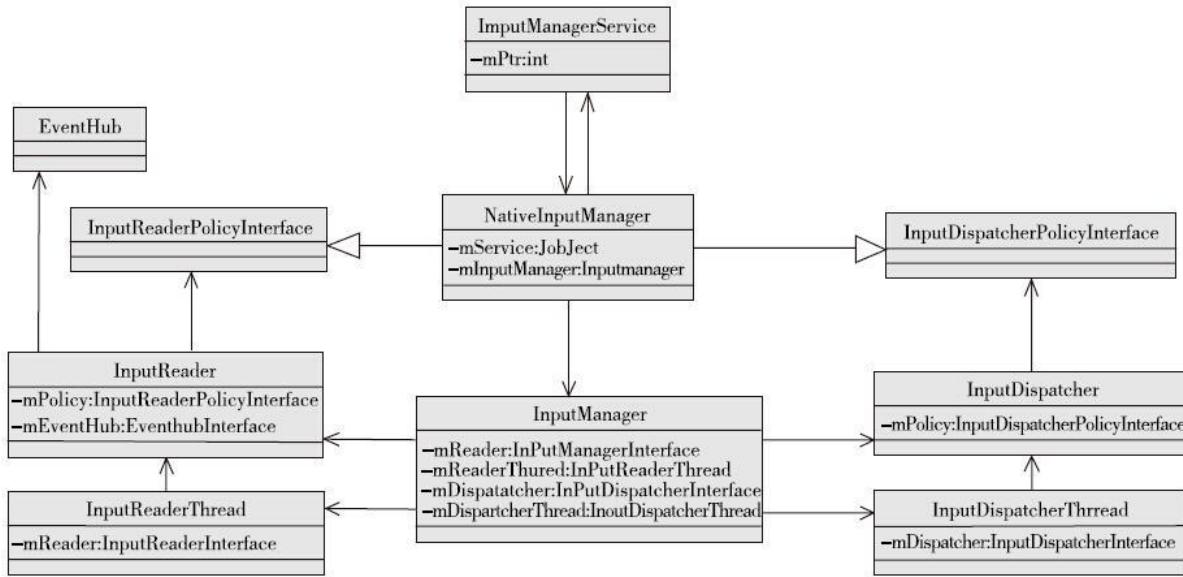


图5-4 IMS的成员关系

在图5-4中，左侧部分为Reader子系统对应于图5-3中的第一台水泵，右侧部分为Dispatcher子系统，对应于图5-3中的第二台水泵。了解了IMS的成员关系后便可以开始我们的IMS深入理解之旅了！

5.2 原始事件的读取与加工

本节将深入探讨第一台水泵——Reader子系统的工作原理。Reader子系统的输入端是设备节点，输出端是Dispatcher子系统的派发队列。从设备节点到派发队列之间的过程发生了什么呢？本章一开始曾经介绍过，一个设备节点对应一个输入设备，并且其中存储了内核写入的原始事件。因此设备节点拥有两个概念：设备与原始事件。因此Reader子系统需要处理输入设备以及原始事件两种类型的对象。

设备节点的新建与删除表示输入设备的可用与无效，Reader子系统需要加载或删除对应设备的配置信息；而设备节点中是否有内容可读表示了是否有新的原始事件到来，有新的原始事件到来时Reader子系统需要开始对新事件进行加工并放置到派发队列中。问题是应该如何监控设备节点的新建与删除动作以及如何确定节点中有内容可读呢？最简单的方法是在线程循环中不断地轮询，然而这会导致非常低下的效率，更会导致电量在无谓的轮询中消耗。Android使用由Linux提供的两套机制INotify与Epoll优雅地解决了这两个问题。在正式探讨Reader子系统的工作原理之前，需要首先了解这两套机制的使用方法。

5.2.1 基础知识：INotify与Epoll

1.INotify介绍与使用

INotify是Linux内核所提供的一种文件系统变化通知机制。它可以为应用程序监控文件系统的变化，如文件的新建、删除、读写等。INotify机制有两个基本对象，分别为inotify对象与watch对象，都使用文件描述符表示。

inotify对象对应一个队列，应用程序可以向inotify对象添加多个监听。当被监听的事件发生时，可以通过read（）函数从inotify对象中将事件信息读取出来。INotify对象可以通过以下方式创建：

```
int inotifyFd = inotify_init();
```

而watch对象则用来描述文件系统的变化事件的监听。它是一个二元组，包括监听目标和事件掩码两个元素。监听目标是文件系统的一个路径，可以是文件也可以是文件夹。而事件掩码则表示了需要监听的事件类型，掩码中的每一位代表一种事件。可以监听的事件种类很多，其中就包括文件的创建（IN_CREATE）与删除（IN_DELETE）。读者可以参阅相关资料以了解其他可监听的事件种类。以下代码即可将一个用于监听输入设备节点的创建与删除的watch对象添加到inotify对象中：

```
int wd = inotify_add_watch (inotifyFd, "/dev/input", IN_CREATE |  
IN_DELETE);
```

完成上述watch对象的添加后，当/dev/input/下的设备节点发生创建与删除操作时，都会将相应的事件信息写入inotifyFd所描述的inotify对象中，此时可以通过read () 函数从inotifyFd描述符中将事件信息读取出来。

事件信息使用结构体inotify_event进行描述：

```
struct inotify_event { __s32 wd; /* 事件对应的Watch对象的描述符 */  
    __u32 mask; /* 事件类型，例如文件被删除，此处值为IN_DELETE */  
    __u32 cookie; __u32 len; /* name字段的长度 */ char name[0]; /* 可变长的  
    字段，用于存储产生此事件的文件路径 */};
```

当监听事件发生时，可以通过如下方式将一个或多个未读取的事件信息读取出来：

```
size_t len = read (inotifyFd, events_buf, BUF_LEN);
```

其中events_buf是inotify_event的数组指针，能够读取的事件数量取决于数组的长度。成功读取事件信息后，便可根据inotify_event结构体的字段判断事件类型以及产生事件的文件路径。

总结一下INotify机制的使用过程：

- 通过inotify_init () 创建一个inotify对象。

- 通过inotify_add_watch将一个或多个监听添加到inotify对象中。
- 通过read () 函数从inotify对象中读取监听事件。当没有新事件发生时，inotify对象中无任何可读数据。

通过INotify机制避免了轮询文件系统的麻烦，但是还有一个问题，INotify机制并不是通过回调的方式通知事件，而需要使用者主动从inotify对象中进行事件读取。那么何时才是读取的最佳时机呢？这就需要借助Linux的另一个优秀的机制Epoll了。

2.Epoll介绍与使用

无论是从设备节点中获取原始输入事件还是从inotify对象中读取文件系统事件，都面临一个问题，就是这些事件都是偶发的。也就是说，大部分情况下设备节点、inotify对象这些文件描述符中都是无数据可读的，同时又希望有事件到来时可以尽快地对事件做出反应。为解决这个问题，我们不希望不断地轮询这些描述符，也不希望为每个描述符创建一个单独的线程进行阻塞时再读取，因为这都将会导致资源被极大浪费。

此时最佳的办法是使用Epoll机制。Epoll可以使用一次等待监听多个描述符的可读/可写状态。等待返回时携带了可读的描述符或自定义的数据，使用者据此读取所需的数据后可以再次进入等待。因此不需要为

每个描述符创建独立的线程进行阻塞读取，避免了资源浪费的同时又可以获得较快的响应速度。

Epoll机制的接口只有三个函数，十分简单。

·epoll_create (int max_fds) : 创建一个epoll对象的描述符，之后对epoll的操作均使用这个描述符完成。max_fds参数表示此epoll对象可以监听的描述符的最大数量。

·epoll_ctl (int epfd, int op, int fd, struct epoll_event*event) : 用于管理注册事件的函数。这个函数可以增加/删除/修改事件的注册。

·int epoll_wait (int epfd, struct epoll_event*events, int maxevents, int timeout) : 用于等待事件到来。当此函数返回时，events数组参数中将会包含产生事件的文件描述符。

接下来以监控若干描述符可读事件为例介绍一下epoll的用法。

(1) 创建epoll对象

首先通过epoll_create () 函数创建一个epoll对象：

```
Int epfd = epoll_create(MAX_FDS)
```

(2) 填充epoll_event结构体

接着为每一个需监控的描述符填充epoll_event结构体，以描述监控事件，并通过epoll_ctl（）函数将此描述符与epoll_event结构体注册进epoll对象。epoll_event结构体的定义如下：

```
struct epoll_event { __uint32_t events; /* 事件掩码，指明需要监听的事件种类 */ epoll_data_t data; /* 使用者自定义的数据，当此事件发生时该数据将原封不动地返回给使用者 */ };
```

epoll_data_t联合体的定义如下，当然，同一时间使用者只能使用一个字段：

```
typedef union epoll_data { void *ptr; int fd; __uint32_t u32; __uint64_t u64; } epoll_data_t;
```

epoll_event结构中的events字段是一个事件掩码，用以指明需要监听的事件种类，同INotify一样，掩码的每一位代表了一种事件。常用的事 件有EPOLLIN（可读）、EPOLLOUT（可写）、EPOLLERR（描述符发生错误）、EPOLLHUP（描述符被挂起）等。关于更多支持的事件读者可参考相关资料。

data字段是一个联合体，它让使用者可以将一些自定义数据加入事件通知中，当此事件发生时，用户设置的data字段将会返回给使用者。在实际使用中经常设置epoll_event.data.fd为需要监听的文件描述符，事件发

生时便可以根据epoll_event.data.fd得知引发事件的描述符。当然，也可以设置epoll_event.data.fd为其他便于识别的数据。

填充epoll_event的方法如下：

```
struct epoll_event eventItem; memset(&eventItem, 0, sizeof(eventItem));  
eventItem.events = EPOLLIN | EPOLLERR | EPOLLHUP; // 监听描述符  
可读以及出错的事件 eventItem.data.fd = listeningFd; // 填写自定义数据  
为需要监听的描述符
```

接下来就可以使用epoll_ctl () 将事件注册进epoll对象了。epoll_ctl () 的参数有4个：

- epfd是由epoll_create () 函数所创建的epoll对象的描述符。
- op表示EPOLL_CTL_ADD/DEL/MOD三种操作，分别表示增加/删除/修改注册事件。
- fd表示了需要监听的描述符。
- event参数是描述监听事件的详细信息的epoll_event结构体。

注册方法如下：

```
// 将事件监听添加到epoll对象中 result = epoll_ctl(epfd,  
EPOLL_CTL_ADD, listeningFd, &eventItem);
```

重复这个步骤可以将多个文件描述符的多种事件监听注册到epoll对象中。完成监听的注册之后，便可以通过epoll_wait（）函数等待事件到来。

（3）使用epoll_wait（）函数等待事件

epoll_wait（）函数将会使调用者陷入等待状态，直到其注册的事件之一发生之后才会返回，并且携带刚刚发生的事件的详细信息。其签名如下：

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int  
timeout);
```

- epfd是由epoll_create（）函数所创建的epoll对象描述符。
- events是一个epoll_event的数组，此函数返回时，事件的信息将被填充至此。
- maxevents表示此次调用最多可以获取多少个事件，当然，events参数必须能够足够容纳这么多事件。
- timeout表示等待超时的事件。

epoll_wait（）函数返回值表示获取了多少个事件。

4.处理事件

epoll_wait返回后，便可以根据events数组中所保存的所有epoll_event结构体的events字段与data字段识别事件的类型与来源。

Epoll的使用步骤总结如下：

- 通过epoll_create () 创建一个epoll对象。
- 为需要监听的描述符填充epoll_events结构体，并使用epoll_ctl () 注册到epoll对象中。
- 使用epoll_wait () 等待事件发生。
- 根据epoll_wait () 返回的epoll_events结构体数组判断事件的类型与来源并进行处理。
- 继续使用epoll_wait () 等待新事件发生。

3.INotify与Epoll的小结

INotify与Epoll这两套由Linux提供的事件监听机制以最小的开销解决了文件系统变化以及文件描述符可读可写状态变化的监听问题。它们是Reader子系统运行的基石，了解了这两个机制的使用方法之后便为对Reader子系统的分析学习铺平了道路。

5.2.2 InputReader的总体流程

在了解INotify与Epoll的基础知识之后便可以正式开始分析Reader子系统的工作原理。首先要理解InputReader的运行方式。在5.1.3节介绍了InputReader被InputManager创建，并运行于InputReaderThread线程中。InputReader如何在InputReaderThread中运行呢？

InputReaderThread继承自C++的Thread类，Thread类封装了pthread线程工具，提供了与Java层Thread类相似的API。C++的Thread类提供了一个名为threadLoop () 的纯虚函数，当线程开始运行后，将会在内建的线程循环中不断地调用threadLoop ()，直到此函数返回false，则退出线程循环，从而结束线程。

InputReaderThread仅仅重写了threadLoop () 函数：

```
[InputReader.cpp-->InputReaderThread::threadLoop()] bool  
InputReaderThread::threadLoop() { mReader->loopOnce(); // 执行  
InputReader的loopOnce()函数 return true; }
```

InputReaderThread启动后，其线程循环将不断地执行InputReader.loopOnce () 函数。因此这个loopOnce () 函数作为线程循环的循环体包含了InputReader的所有工作。



注意

C++层的Thread类与Java层的Thread类有一个显著的区别。C++层Thread类内建了线程循环，`threadLoop ()` 就是一次循环而已，只要返回值为`true`，`threadLoop ()` 将会不断地被内建的循环调用。这也是`InputReader.loopOnce ()` 函数名称的由来。而Java层Thread类的`run ()` 函数则是整个线程的全部，一旦其退出，线程也便完结。

接下来看一下`InputReader.loopOnce ()` 的代码，分析一下`InputReader` 在一次线程循环中做了什么。

```
[InputReader.cpp-->InputReader::loopOnce()] void InputReader::loopOnce()
{ ..... /* ① 通过EventHub抽取事件列表。读取的结果存储在参数
mEventBuffer中，返回值表示事件的个数当 EventHub中无事件可以抽
取时，此函数的调用将会阻塞直到事件到来或者超时 */ size_t count =
mEventHub->getEvents(timeoutMillis , mEventBuffer,
EVENT_BUFFER_SIZE); { AutoMutex _l(mLock); ..... if (count) { // ② 如
果有抽取事件，则调用processEventsLocked()函数对事件进行加工处理
```

```
processEventsLocked(mEventBuffer, count); } ..... } ..... /* ③ 发布事件。  
processEventsLocked()函数在对事件进行加工处理之后，便将处理后的  
事件存储在 mQueuedListener中。在循环的最后，通过调用flush()函数  
将所有事件交付给InputDispatcher */ mQueuedListener->flush(); }
```

InputReader的一次线程循环的工作思路非常清晰，一共三步：

- 首先从EventHub中抽取未处理的事件列表。这些事件分为两类，一类是从设备节点中读取的原始输入事件，另一类则是输入设备可用性变化事件，简称为设备事件。
- 通过processEventsLocked () 对事件进行处理。对于设备事件，此函数对根据设备的可用性加载或移除设备对应的配置信息。对于原始输入事件，则在进行转译、封装与加工后将结果暂存到mQueuedListener 中。
- 所有事件处理完毕后，调用mQueuedListener.flush () 将所有暂存的输入事件一次性地交付给InputDispatcher。

这便是InputReader的总体工作流程。而我们接下来将详细讨论这三步的实现。

5.2.3 深入理解EventHub

InputReader在其线程循环中的第一个工作便是从EventHub中读取一批未处理的事件。EventHub是如何工作的呢？

EventHub的直译是事件集线器，顾名思义，它将所有的输入事件通过一个接口get-Events () 把从多个输入设备节点中读取的事件交给InputReader，它是输入系统最底层的一个组件。它是如何工作呢？没错，正是基于前文所述的INotify与Epoll两套机制。

1.设备节点监听的建立

在EventHub的构造函数中，它通过INotify与Epoll机制建立起对设备节点增删事件以及可读状态的监听。在继续之前，请先回忆一下INotify与Epoll的使用方法。

EventHub的构造函数如下：

```
[EventHub.cpp-->EventHub::EventHub()] EventHub::EventHub(void) :  
mBuiltInKeyboardId(NO_BUILT_IN_KEYBOARD), mNextDeviceId(1),  
mOpeningDevices(0), mClosingDevices(0),  
mNeedToSendFinishedDeviceScan(false), mNeedToReopenDevices(false),  
mNeedToScanDevices(true), mPendingEventCount(0),  
mPendingEventIndex(0), mPendingINotify(false) { /* ① 首先使用  
epoll_create()函数创建一个epoll对象。EPOLL_SIZE_HINT指定最大监  
听个数为8 这个epoll对象将用来监听设备节点是否有数据可读（有无事
```

件) /* mEpollFd = epoll_create(EPOLLOUT_SIZE_HINT); // ② 创建一个
inotify对象。这个inotify对象将被用来监听设备节点的增删事件
mINotifyFd = inotify_init(); /* 将存储设备节点的路径/dev/input作为监听
对象添加到inotify对象中。当此文件夹下的设备节点发生 创建与删除
事件时，都可以通过mINotifyFd读取事件的详细信息 */ int result =
inotify_add_watch(mINotifyFd, DEVICE_PATH, IN_DELETE |
IN_CREATE); /* ③ 接下来将mINotifyFd作为epoll的一个监控对象。当
inotify事件到来时， epoll_wait()将立刻返回， EventHub便可从
mINotifyFd中读取设备节点的增删信息，并进行相应处理 */ struct
epoll_event eventItem; memset(&eventItem, 0, sizeof(eventItem));
eventItem.events = EPOLLIN; // 监听mINotifyFd可读 // 注意这里并没有
使用fd字段，而使用了自定义的值EPOLL_ID_INOTIFY
eventItem.data.u32 = EPOLL_ID_INOTIFY; // 将对mINotifyFd的监听注
册到epoll对象中 result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD,
mINotifyFd, &eventItem); /* 在构造函数剩余的代码中， EventHub创建
了一个名为wakeFds的匿名管道，并将管道读取端的描述符 的可读事件
注册到epoll对象中。因为InputReader在执行getEvents()时会因无事件而
导致其线程 阻塞在epoll_wait()的调用里，然而有时希望能够立刻唤醒
InputReader线程使其处理一些请求。此时只须向wakeFds管道的写入端
写入任意数据，此时读取端有数据可读，使得epoll_wait()得以返回，
从而 达到唤醒InputReader线程的目的*/ }

EventHub的构造函数初始化了Epoll对象和INotify对象，分别监听原始输入事件与设备节点增删事件。同时将INotify对象的可读性事件也注册到Epoll中，因此EventHub可以像处理原始输入事件一样监听设备节点增删事件。

构造函数同时也揭示了EventHub的监听工作分为设备节点和原始输入事件两个方面，接下来将深入探讨这两方面的内容。

2.getEvents () 函数的工作方式

正如前文所述，InputReaderThread的线程循环为Reader子系统提供了运转的动力，EventHub的工作也是由它驱动的。InputReader::loopOnce () 函数调用EventHub::getEvents () 函数获取事件列表，所以这个getEvents () 是EventHub运行的动力所在，几乎包含了EventHub的所有工作内容，因此首先要将getEvents () 函数的工作方式搞清楚。

getEvents () 函数的签名如下：

```
size_t EventHub::getEvents(int timeoutMillis, RawEvent* buffer, size_t  
bufferSize)
```

此函数将尽可能多地读取设备增删事件与原始输入事件，将它们封装为RawEvent结构体，并放入buffer中供InputReader进行处理。RawEvent结构体的定义如下：

```
[EventHub.cpp-->RawEvent] struct RawEvent { nsecs_t when; /* 发生事件  
时的时间戳 */ int32_t deviceId; /* 产生事件的设备Id，它是由EventHub  
自行分配的，InputReader可以根据它从EventHub中获取此设备的详细  
信息 */ int32_t type; /* 事件的类型 */ int32_t code; /* 事件代码 */ int32_t  
value; /* 事件值 */ };
```

可以看出，RawEvent结构体与getevent工具的输出十分一致，包含了原始输入事件的4个基本元素，因此用RawEvent结构体表示原始输入事件是非常直观的。RawEvent同时也用来表示设备增删事件，为此，EventHub定义了三个特殊的事件类型DEVICE_ADD、DEVICE_REMOVED以及FINISHED_DEVICE_SCAN，用以与原始输入事件进行区别。

由于getEvents () 函数较为复杂，为了给后续分析铺平道路，本节不讨论其细节，先通过伪代码理解此函数的结构与工作方式，在后续深入分析时思路才会比较清晰。

getEvents () 函数的本质就是读取并处理Epoll事件与INotify事件。参考以下代码：

```
[EventHub.cpp--> EventHub::getEvents()] size_t EventHub::getEvents(int  
timeoutMillis, RawEvent* buffer, size_t bufferSize) { /* event指针指向在  
buffer中下一个可用于存储事件的RawEvent结构体。每存储一个事件，
```

event 指针都会向后偏移一个元素 */ RawEvent* event = buffer; /*
capacity记录了buffer中剩余的元素数量。当capacity为0时，表示buffer
已满，此时需要停止继续处理新事件，并将已处理的事件返回给调用
者 */ size_t capacity = bufferSize; /* 接下来的循环是getEvents()函数的主
体。在这个循环中，会先将可用事件放入buffer中并返回。如果 没有可
用事件，则进入epoll_wait()等待事件到来， epoll_wait()返回后会重新执
行循环体，将新事件 放入buffer */ for (;;) { nsecs_t now =
systemTime(SYSTEM_TIME_MONOTONIC); /* ① 首先进行与设备相
关的工作。某些情况下，如EventHub创建后第一次执行getEvents() 函数
时，需要扫描/dev/input文件夹下的所有设备节点并将这些设备打开。
另外，当设备节点 的增删事件发生时，会将这些事件存入buffer中 */
..... /* ② 处理未被InputReader取走的输入事件与设备事件。 epoll_wait()
所取出的epoll_event 存储在mPendingEventItems中，
mPendingEventCount指定mPendingEventItems数组 所存储的事件个数。
而mPendingEventIndex指定尚未处理的epoll_event的索引 */ while
(mPendingEventIndex < mPendingEventCount) { const struct epoll_event&
eventItem = mPendingEventItems[mPendingEventIndex++]; /* 在这里分析
每一个epoll_event，如果表示设备节点可读，则读取原始事件并放置到
buffer中。如果表示mINotifyFd可读，则设置mPendingINotify为true，当
InputReader 将现有的输入事件都取出后读取mINotifyFd中的事件，并
加载与卸载相应的设备。另外，如果此 epoll_event表示wakeFds的读取

端有数据可读，则设置awake标志为true，此时无论此次getEvents()调用是否取到事件，都不会调用epoll_wait()进行事件等待 */ } // ③ 如果mINotifyFd有数据可读，说明设备节点发生了增删操作 if (mPendingINotify && mPendingEventIndex >= mPendingEventCount) { /* 读取mINotifyFd中的事件，同时对输入设备进行相应的加载与卸载操作。这个操作必须当Input- Reader将现有输入事件读取并处理完毕后才能进行，因为现有的输入事件可能来自需要被卸载的输入设备， InputReader处理这些事件依赖于对应的设备信息 */ deviceChanged = true; } // 设备节点增删操作发生时，则重新执行循环体，以便将设备变化的事件放入buffer中 if (deviceChanged) { continue; } // 如果此次 getEvents()调用成功获取了一些事件，或者要求唤醒InputReader，则退出循环并 // 结束getEvents()的调用，使InputReader可以立刻处理事件 if (event != buffer || awoken) { break; } /* ④ 如果此次getEvents()调用没能 获取事件，说明mPendingEventItems中没有事件可用。于是执行 epoll_wait()函数等待新的事件到来，将结果存储到mPendingEventItems 里，并重置 mPendingEventIndex为0 */ mPendingEventIndex = 0; int pollResult = epoll_wait(mEpollFd, mPendingEventItems, EPOLL_MAX_EVENTS, timeoutMillis); mPendingEventCount = size_t(pollResult); // 从epoll_wait()中得到新的事件后，重新循环，对新 事件进行处理 } // 返回本次getEvents()调用所读取的事件数量 return event - buffer; }

`getEvents ()` 函数使用Epoll的核心是`mPendingEventItems`数组，它是一个事件池。`getEvents ()` 函数会优先从这个事件池获取epoll事件进行处理，并将读取相应的原始输入事件返回给调用者。当因为事件池枯竭而导致调用者无法获得任何事件时，会调用`epoll_wait ()` 函数等待新事件到来，将事件池重新注满，然后再重新处理事件池中的Epoll事件。从这个意义来说，`getEvents ()` 函数的调用过程，就是消费`epoll_wait ()` 所产生的Epoll事件的过程。因此可以将从`epoll_wait ()` 的调用开始，到将Epoll事件消费完毕的过程称为EventHub的一个监听周期。依据每次`epoll_wait ()` 产生的Epoll事件的数量以及设备节点中原始输入事件的数量，一个监听周期包含一次或多次`getEvents ()` 调用。周期中的第一次调用会因为事件池枯竭而直接进入`epoll_wait ()`，而周期中的最后一次调用一定会将最后的事件取走。



注意

`getEvents ()` 采用事件池机制的根本原因是buffer的容量限制。由于一次`epoll_wait ()` 可能返回多个设备节点的可读事件，每个设备节点又有可能读取多条原始输入事件，一段时间内原始输入事件的数量可能

大于buffer的容量。因此需要一个事件池以缓存因buffer容量不够而无法处理的epoll事件，以便在下次调用时可以将这些事件优先处理。这是缓冲区操作的一个常用技巧。

当有INotify事件可以从mINotifyFd中读取时，会产生一个epoll事件，EventHub便得知设备节点发生了增删操作。在getEvents () 将事件池中的所有事件处理完毕后，便会从mINotifyFd中读取INotify事件，进行输入设备的加载/卸载操作，然后生成对应的RawEvent结构体并返回给调用者。

通过上述分析可以看到，getEvents () 包含了原始输入事件读取、输入设备加载/卸载等操作。这几乎是EventHub的全部工作。如果没有getEvents () 的调用，EventHub将对输入事件、设备节点增删事件置若罔闻，因此可以将一次getEvents () 调用理解为一次心跳，EventHub的核心功能都会在这次心跳中完成。

getEvents () 的代码还揭示了另外一个信息：在一个监听周期内的设备增删事件与Epoll事件的优先级。设备事件的生成逻辑位于Epoll事件的处理之前，因此getEvents () 将优先生成设备增删事件，完成所有设备增删事件的生成之前不会处理Epoll事件，也就是不会生成原始输入事件。

接下来我们将从设备管理与原始输入事件处理两个方面深入探讨 EventHub。

3. 输入设备管理

因为输入设备是输入事件的来源，并且决定了输入事件的含义，因此首先讨论EventHub的输入设备管理机制。

输入设备是一个可以为接收用户操作的硬件，内核会为每一个输入设备在/dev/input/下创建一个设备节点，而当输入设备不可用时（例如被拔出），将其设备节点删除。这个设备节点包含了输入设备的所有信息，包括名称、厂商、设备类型、设备的功能等。除了设备节点外，某些输入设备还包含一些自定义配置，这些配置以键值对的形式存储在某个文件中。这些信息决定了Reader子系统如何加工原始输入事件。EventHub负责在设备节点可用时加载并维护这些信息，并在设备节点被删除时将其移除。

EventHub通过一个定义在其内部的名为Device的私有结构体来描述一个输入设备。其定义如下：

```
[EventHub.h-->EventHub::Device] struct Device { Device* next; /* Device  
结构体实际上是一个单链表 */ int fd; /* fd表示此设备的设备节点的描述  
符，可以从此描述符中读取原始输入事件 */ const int32_t id; /* id在输入  
系统中唯一标识这个设备，由EventHub在加载设备时进行分配 */ const
```

```
String8 path; /* path存储了设备节点在文件系统中的路径 */ const
InputDeviceIdentifier identifier; /* 厂商信息，存储了设备的供应商、型号等信息 这些信息从设备节点中获得 */ uint32_t classes; /* classes表示了设备的类别、键盘设备、触控设备等。一个设备可以同时属于多个设备类别。类别决定了InputReader如何加工其原始输入事件 */
/* 接下来是一系列的事件位掩码，它们详细地描述了设备能够产生的事件类型。设备能够产生的事件类型决定了此设备所属的类型 */
uint8_t keyBitmask[(KEY_MAX + 1) / 8]; ..... /* 配置信息。以键值对的形式存储在一个文件中，其路径取决于identifier字段中的厂商信息，这些配置信息将会影响InputReader对此设备的事件的加工行为 */
String8 configurationFile; PropertyMap* configuration; /* 键盘映射表。对于键盘类型的设备，这些键盘映射表将原始事件中的键盘扫描码转换为Android 定义的按键值。这个映射表也是从一个文件中加载的，文件路径取决于identifier字段中的厂商信息 */
VirtualKeyMap*
virtualKeyMap; KeyMap keyMap; sp overlayKeyMap; sp
combinedKeyMap; // 力反馈相关的信息。有些设备如高级的游戏手柄支持力反馈功能，目前暂不考虑
bool ffEffectPlaying; int16_t ffEffectId; };
```

Device结构体所存储的信息主要包括以下几个方面：

- 设备节点信息：保存输入设备节点的文件描述符、文件路径等。

·厂商信息：包括供应商、设备型号、名称等信息，这些信息决定了加载配置文件与键盘映射表的路径。

·设备特性信息：包括设备的类别、可以上报的事件种类等。这些特性信息直接影响了InputReader对其所产生的事件的加工处理方式。

·设备的配置信息：包括键盘映射表及其他自定义的信息，从特定位置的配置文件中读取。

另外，Device结构体还存储了力反馈所需的一些数据。在本节中暂不讨论。

EventHub用一个名为mDevices的字典保存当前处于打开状态的设备节点的Device结构体。字典的键为设备Id。

(1) 输入设备的加载

EventHub在创建后第一次调用getEvents () 函数时完成对系统中现有输入设备的加载。

再看一下getEvents () 函数中相关内容的实现：

```
[EventHub.cpp-->EventHub::getEvents()] size_t EventHub::getEvents(int  
timeoutMillis, RawEvent* buffer, size_t bufferSize) { for (;;) { // 处理输入  
设备卸载操作 ..... /* 在EventHub的构造函数中， mNeedToScanDevices
```

被设置为true，因此创建后第一次调用getEvents()函数会执行
scanDevicesLocked(), 加载所有输入设备 */ if (mNeedToScanDevices) {
mNeedToScanDevices = false; /* scanDevicesLocked()将会把/dev/input下
所有可用的输入设备打开并存储到Device结构体中 */
scanDevicesLocked(); mNeedToSendFinishedDeviceScan = true; } }
return event - buffer; }

加载所有输入设备由scanDevicesLocked () 函数完成。看一下其实现：

[EventHub.cpp-->EventHub::scanDevicesLocked()] void
EventHub::scanDevicesLocked() { // 调用scanDirLocked()函数遍
历/dev/input文件夹下的所有设备节点并打开 status_t res =
scanDirLocked(DEVICE_PATH); // 错误处理 // 打开一个名为
VIRTUAL_KEYBOARD的输入设备。这个设备时刻打开着。它是一个
虚拟的输入设备，没有对应的输入节点。读者先记住有这么一个输入
设备存在输入系统中即可 */ if
(mDevices.indexOfKey(VIRTUAL_KEYBOARD_ID) < 0) {
createVirtualKeyboardLocked(); } }

scanDirLocked () 遍历指定文件夹下的所有设备节点，分别对其执行
openDeviceLocked () 完成设备的打开操作。在这个函数中将为设备节
点创建并加载Device结构体。参考其代码：

[EventHub.cpp-->EventHub::openDeviceLocked()] status_t

EventHub::openDeviceLocked(const char *devicePath) { // 打开设备节点的文件描述符，用于获取设备信息以及读取原始输入事件 int fd = open(devicePath, O_RDWR | O_CLOEXEC); // 接下来的代码通过ioctl()函数从设备节点中获取输入设备的厂商信息 InputDeviceIdentifier identifier; // 分配一个设备Id并创建Device结构体 int32_t deviceId = mNextDeviceId++; Device* device = new Device(fd, deviceId, String8(devicePath), identifier); // 为此设备加载配置信息 loadConfigurationLocked(device); // ① 通过ioctl函数获取设备的事件位掩码。事件位掩码指定了输入设备可以产生何种类型的输入事件 ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(device->keyBitmask)), device->keyBitmask); ioctl(fd, EVIOCGPROP(sizeof(device->propBitmask)), device->propBitmask); // 接下来的一大段内容是根据事件位掩码为设备分配类别，即设置classes字段 /* ② 将设备节点的描述符的可读事件注册到Epoll中。当此设备的输入事件到来时，Epoll会在get-Events()函数的调用中产生一个epoll事件 */ struct epoll_event eventItem; memset(&eventItem, 0, sizeof(eventItem)); eventItem.events = EPOLLIN; eventItem.data.u32 = deviceId; /* 注意，epoll_event的自定义信息是设备的Id if (epoll_ctl(mEpollFd, EPOLL_CTL_ADD, fd, &eventItem)) { } // ③ 调用addDeviceLocked()将Device添加到mDevices字典中 addDeviceLocked(device); return 0; }

`openDeviceLocked ()` 函数打开指定路径的设备节点，为其创建并填充 `Device` 结构体，然后将设备节点的可读事件注册到 `Epoll` 中，最后将新建的 `Device` 结构体添加到 `mDevices` 字典中以供检索之需。整个过程比较清晰，但仍有以下几点需要注意：

- `openDeviceLocked ()` 函数从设备节点中获取了设备可能上报的事件类型，并据此为设备分配了类别。整个分配过程非常繁琐，由于它和 `InputReader` 的事件加工过程关系紧密，因此这部分内容将在 5.2.4 节再详细讨论。
- 向 `Epoll` 注册设备节点的可读事件时，`epoll_event` 的自定义数据被设置为设备的 Id 而不是 `fd`。
- `addDeviceLocked ()` 将新建的 `Device` 对象添加到 `mDevices` 字典中的同时也会将其添加到一个名为 `mOpeningDevices` 的链表中。这个链表保存了刚刚被加载，但尚未通过 `getEvents ()` 函数向 `InputReader` 发送 `DEVICE_ADD` 事件的设备。

完成输入设备的加载之后，通过 `getEvents ()` 函数便可以读取此设备所产生的输入事件了。除了在 `getEvents ()` 函数中使用 `scanDevicesLockd ()` 一次性加载所有输入设备，当 `INotify` 事件告知有新的输入设备节点被创建时，也会通过 `openDeviceLocked ()` 加载设备，这些内容稍后再讨论。

(2) 输入设备的卸载

输入设备的卸载由closeDeviceLocked () 函数完成。由于此函数的工作内容与openDeviceLocked () 函数正好相反，在此就不列出其代码了。设备的卸载过程为：

- 从Epoll中注销对描述符的监听。
- 关闭设备节点的描述符。
- 从mDevices字典中删除对应的Device对象。
- 将Device对象添加到mClosingDevices链表中，与mOpeningDevices类似，这个链表保存了刚刚被卸载，但尚未通过getEvents () 函数向InputReader发送DEVICE_REMOVED事件的设备。

同加载设备一样，在getEvents () 函数中有根据需要卸载所有输入设备的操作（比如当EventHub要求重新加载所有设备时，会先将所有设备卸载）。并且当INotify事件告知有设备节点删除时也会调用closeDeviceLocked () 将设备卸载。

(3) 设备增删事件

在分析设备的加载与卸载时可以发现，新加载的设备与新卸载的设备会被分别放入mOpeningDevices与mClosingDevices链表之中。这两个链

表将是在getEvents () 函数中向InputReader发送设备增删事件的依据。

参考getEvents () 函数的相关代码，以设备卸载事件为例看一下设备增删事件是如何产生的：

```
[EventHub.cpp-->EventHub::getEvents()] size_t EventHub::getEvents(int  
timeoutMillis, RawEvent* buffer, size_t bufferSize) { for (;;) { // 遍历  
mClosingDevices链表，为每一个已卸载的设备生成  
DEVICE_REMOVED事件 while (mClosingDevices) { Device* device =  
mClosingDevices; mClosingDevices = device->next; /* 分析getEvents()函  
数的工作方式时介绍过，event指针指向buffer中下一个可用于填充事  
件的RawEvent对象 */ event->when = now; // 设置产生事件的事件戳 event-  
>deviceId = device->id == mBuiltInKeyboardId ?  
BUILT_IN_KEYBOARD_ID : device->id; event->type =  
DEVICE_REMOVED; // 设置事件的类型为DEVICE_REMOVED event  
+= 1; // 将event指针移动到下一个可用于填充事件的RawEvent对象  
delete device; // 生成DEVICE_REMOVED事件之后，被卸载的Device对  
象就不再需要了 mNeedToSendFinishedDeviceScan = true; // 随后发送  
FINISHED_DEVICE_SCAN事件 /* 当buffer已满则停止继续生成事件，  
将已生成的事件返回给调用者。尚未生成的事件将在下次getEvents()调  
用时生成并返回给调用者 */ if (--capacity == 0) { break; } } // 接下来生
```

成DEVICE_ADDED事件，此过程与生成DEVICE_REMOVED事件一致

```
..... } return event - buffer; }
```

可以看到，在一次getEvents () 调用中会尝试为所有尚未发送增删事件的输入设备生成对应的事件并返回给调用者。表示设备增删事件的RawEvent对象包含三个信息：产生事件的事件戳、产生事件的设备Id以及事件类型（DEVICE_ADDED或DEVICE_REMOVED）。

当生成设备增删事件时，会设置mNeedToSendFinishedDeviceScan为true，这个动作的意思是完成所有DEVICE_ADDED/REMOVED事件的生成之后，需要向getEvents () 的调用者发送一个FINISHED_DEVICE_SCAN事件，表示设备增删事件的上报结束。这个事件仅包括时间戳与事件类型两个信息。

经过以上分析可知，EventHub可以产生的设备增删事件一共有三种，而且这三种事件拥有固定的优先级，DEVICE_REMOVED事件的优先级最高，DEVICE_ADDED事件次之，FINISHED_DEVICE_SCAN事件最低。而且，getEvents () 完成当前高优先级事件的生成之前，不会进行低优先级事件的生成。因此，当发生设备的加载与卸载时，EventHub所生成的完整的设备增删事件序列如图5-5所示，其中R表示DEVICE_REMOVED，A表示DEVICE_ADDED，F表示FINISHED_DEVICE_SCAN。

R1	R2	...	Rm	A1	A2	...	An	F
----	----	-----	----	----	----	-----	----	---

图5-5 设备增删事件的完整序列

由于参数buffer的容量限制，这个事件序列可能需要通过多次getEvents () 调用才能完整地返回给调用者。另外，根据5.2.2节的讨论，设备增删事件相对于Epoll事件拥有较高的优先级，因此从R1事件开始生成到F事件生成之前，getEvents () 不会处理Epoll事件，也就是说不会生成原始输入事件。

总结一下设备增删事件的生成原理：

- 当发生设备增删时，addDeviceLocked () 函数与closeDeviceLocked () 函数会将相应的设备放入mOpeningDevices和mClosingDevices链表中。
- getEvents () 函数会根据mOpeningDevices和mClosingDevices两个链表生成对应DEVICE_ADDED和DEVICE_REMOVED事件，其中后者的生成拥有高优先级。
- DEVICE_ADDED和DEVICE_REMOVED事件都生成完毕后，getEvents () 会生成FINISHED_DEVICE_SCAN事件，标志设备增删事件序列的结束。

(4) 通过INotify动态地加载与卸载设备

通过前文的介绍可以知道openDeviceLocked () 和closeDeviceLocked () 加载与卸载输入设备。接下来分析EventHub如何通过INotify进行设备的动态加载与卸载。在EventHub的构造函数中创建一个名为mINotifyFd的INotify对象的描述符，用以监控/dev/input下设备节点的增删。之后将mINotifyFd的可读事件加入Epoll中。于是可以确定动态加载与卸载设备的工作方式为：首先筛选epoll_wait () 函数所取得的Epoll事件，如果Epoll事件表示了mINotifyFd可读，便从mINotifyFd中读取设备节点的增删事件，然后通过执行openDeviceLocked () 或closeDeviceLocked () 进行设备的加载与卸载。

看一下getEvents () 中与INotify相关的代码：

```
[EventHub.cpp-->EventHub::getEvents()] size_t EventHub::getEvents(int  
timeoutMillis, RawEvent* buffer, size_t bufferSize) { for (;;) { ..... // 设备  
增删事件处理 while (mPendingEventIndex < mPendingEventCount) {  
const struct epoll_event& eventItem =  
mPendingEventItems[mPendingEventIndex++]; /* ① 通过Epoll事件的data  
字段确定此事件表示mINotifyFd可读注意EPOLL_ID_INOTIFY 在  
EventHub的构造函数中作为data字段向Epoll注册mINotifyFd的可读事件  
*/ if (eventItem.data.u32 == EPOLL_ID_INOTIFY) { if (eventItem.events  
& EPOLLIN) { mPendingINotify = true; // 标记INotify事件待处理 } else {
```

```
..... } continue; // 继续处理下一条Epoll事件 } ..... // 其他Epoll事件的处  
理 } // 如果INotify事件待处理 if (mPendingINotify &&  
mPendingEventIndex >= mPendingEventCount) { mPendingINotify = false;  
/* ② 调用readNotifyLocked()函数读取并处理存储在mINotifyFd中的  
INotify事件这个函数将完成设备的加载与卸载 */ readNotifyLocked();  
deviceChanged = true; } // ③ 如果处理了INotify事件，则返回到循环开  
始处，生成设备增删事件 if (deviceChanged) { continue; } } }
```

getEvents () 函数中与INotify相关的代码共有三处：

- 识别表示mINotifyFd可读的Epoll事件，并通过设置mPendingINotify为true以标记有INotify事件待处理。getEvents () 并没有立刻处理INotify事件，因为此时进行设备的加载与卸载是不安全的。其他Epoll事件可能包含了来自即将被卸载的设备的输入事件，因此需要将所有Epoll事件都处理完毕后再进行加载与卸载操作。
- 当epoll_wait () 所返回的Epoll事件都处理完毕后，调用readNotifyLocked () 函数读取mINotifyFd中的事件，并进行设备的加载与卸载操作。
- 完成设备的动态加载与卸载后，需要返回到循环最开始处，以便设备增删事件处理代码生成设备的增删事件。

其中第一部分与第三部分比较容易理解。接下来看一下
readNotifyLocked () 是如何工作的。

```
[EventHub.cpp-->EventHub::readNotifyLocked()] status_t  
EventHub::readNotifyLocked() { ..... // 从mINotifyFd中读取INotify事件列  
表 res = read(mINotifyFd, event_buf, sizeof(event_buf)); ..... // 逐个处理列  
表中的事件 while(res >= (int)sizeof(*event)) { strcpy(filename, event-  
>name); // 从事件中获取设备节点路径 if(event->mask & IN_CREATE) {  
openDeviceLocked(devname); // 如果事件类型为IN_CREATE，则加载对  
应设备 } else { closeDeviceByPathLocked(devname); // 否则卸载对应设  
备 } ..... // 移动到列表中的下一个事件 } return 0; }
```

(5) EventHub设备管理总结

至此，EventHub的设备管理相关的知识便讨论完毕了。在这里进行一
下总结：

- EventHub通过Device结构体描述输入设备的各种信息。
- EventHub在getEvents () 函数中进行设备的加载与卸载操作。设备的
加载与卸载分为按需加载或卸载以及通过INotify动态加载或卸载特定
设备两种方式。

`getEvents ()` 函数进行设备的加载与卸载操作后，会生成 `DEVICE_ADDED`、`DEVICE_REMOVED` 以及 `FINISHED_DEVICE_SCAN` 三种设备增删事件，并且设备增删事件拥有高于Epoll事件的优先级。

4. 原始输入事件的监听与读取

本节将讨论EventHub另一个核心的功能——监听与读取原始输入事件。

回忆一下输入设备的加载过程，当设备加载时，`openDeviceLocked ()` 会打开设备节点的文件描述符，并将其可读事件注册进Epoll中。于是当设备的原始输入事件到来时，`getEvents ()` 函数将会获得一个Epoll事件，然后根据此Epoll事件读取文件描述符中的原始输入事件，将其填充到RawEvents结构体并放入buffer中被调用者取走。

`openDeviceLocked ()` 注册了设备节点的EPOLLIN和EPOLLHUP两个事件，分别表示可读与被挂起（不可用），因此`getEvents ()` 需要分别处理这两种事件。

看一下`getEvents ()` 函数中的相关代码：

```
[EventHub.cpp-->EventHub::getEvents()] size_t EventHub::getEvents(int  
timeoutMillis, RawEvent* buffer, size_t bufferSize) { for (;;) { ..... // 设备  
增删事件处理 while (mPendingEventIndex < mPendingEventCount) {
```

```
const struct epoll_event& eventItem =  
mPendingEventItems[mPendingEventIndex++]; ..... // INotify与wakeFd的  
Epoll事件处理 /* ① 通过Epoll的数据u32字段获取设备Id，进而获取对  
应的Device对象。如果无法找到对应的Device对象，说明此Epoll事件  
并不表示原始输入事件的到来，可忽略 */ ssize_t deviceIndex =  
mDevices.indexOfKey(eventItem.data.u32); Device* device =  
mDevices.valueAt(deviceIndex); ..... if (eventItem.events & EPOLLIN) { /*  
② 如果Epoll事件为EPOLLIN，表示设备节点有原始输入事件可读。此  
时可以从描述符中读取。读取结果作为input_event结构体并存储在  
readBuffer中，注意事件的个数受到capacity的限制 */ int32_t readSize =  
read(device->fd, readBuffer, sizeof(struct input_event) * capacity); if (...) {  
..... // 一些错误处理 } else { size_t count = size_t(readSize) / sizeof(struct  
input_event); /* ② 将读取到的每一个input_event结构体中的数据转换为  
一个RawEvent对象，并存储在buffer参数中以返回给调用者 */ for  
(size_t i = 0; i < count; i++) { const struct input_event& iev = readBuffer[i];  
..... event->when = now; event->deviceId = deviceId; event->type =  
iev.type; event->code = iev.code; event->value = iev.value; event += 1; // 移  
动到buffer的下一个可用元素 } /* 接下来的一个细节需要注意，因为  
buffer的容量限制，可能无法完读设备节点中存储的原始事件。一旦  
buffer满了则需要立刻返回给调用者。设备节点中剩余的输入事件将在  
下次getEvents()调用时继续读取，也就是说，当前的Epoll事件并未处
```

理完毕。mPendingEventIndex -= 1的目的就是使下次getEvents() 调用能够继续处理这个Epoll事件 */ capacity -= count; if (capacity == 0) { mPendingEventIndex -= 1; break; } } else if (eventItem.events & EPOLLHUP) { deviceChanged = true; // 如果设备节点的文件描述符被挂起则卸载此设备 closeDeviceLocked(device); } else { } } // 读取并处理INotify事件 // 等待新的Epoll事件 } return event - buffer ; }

getEvents () 通过Epoll事件的data.u32字段在mDevices列表中查找已加载的设备，并从设备的文件描述符中读取原始输入事件列表。从文件描述符中读取的原始输入事件存储在input_event结构体中，这个结构体的4个字段存储了事件的事件戳、类型、代码与值等元素。然后逐一将input_event的数据转存到RawEvent中并保存至buffer以返回给调用者。



注意

为了叙述简单，上述代码使用了调用getEvents () 的时间作为输入事件的时间戳。由于调用getEvents () 函数的时机与用户操作的时间差的存在，会使得此时间戳与事件的真实时间有所偏差。从设备节点中

读取的input_event中也包含了一个时间戳，这个时间戳消除了getEvents () 调用所带来的时间差，因此可以获得更精确的时间控制。可以通过打开HAVE_POSIX_CLOCKS宏以使用input_event中的时间而不是将getEvents () 调用的时间作为输入事件的时间戳。

需要注意的是，由于Epoll事件的处理优先级低于设备增删事件，因此当发生设备加载与卸载动作时，不会产生设备输入事件。另外还需注意，在一个监听周期中，getEvents () 在将一个设备节点中的所有原始输入事件读取完毕之前，不会读取其他设备节点中的事件。

5.EventHub总结

本节针对EventHub的设备管理与原始输入事件的监听读取两个核心内容介绍了EventHub的工作原理。EventHub作为直接操作设备节点的输入系统组件，隐藏了INotify与Epoll以及设备节点读取等底层操作，通过一个简单的接口getEvents () 向使用者提供抽取设备事件与原始输入事件的功能。EventHub的核心功能都在getEvents () 函数中完成，因此深入理解getEvents () 的工作原理对于深入理解EventHub至关重要。

getEvents () 函数的本质是通过epoll_wait () 获取Epoll事件到事件池，并对事件池中的事件进行消费的过程。从epoll_wait () 的调用开始到事件池中最后一个事件被消费完毕的过程称为EventHub的一个监

听周期。由于buffer参数的尺寸限制，一个监听周期可能包含多个getEvents () 调用。周期中的第一个getEvents () 调用一定会因事件池的枯竭而直接进行epoll_wait () ，而周期中的最后一个getEvents () 一定会将事件池中的最后一个事件消费完毕并将事件返回给调用者。前文所讨论的事件优先级都是在同一个监听周期内而言的。

在本节中出现了很多种事件，有原始输入事件、设备增删事件、Epoll事件、INotify事件等，存储事件的结构体有RawEvent、epoll_event、inotify_event、input_event等。图5-6可以帮助读者理清这些事件之间的关系。

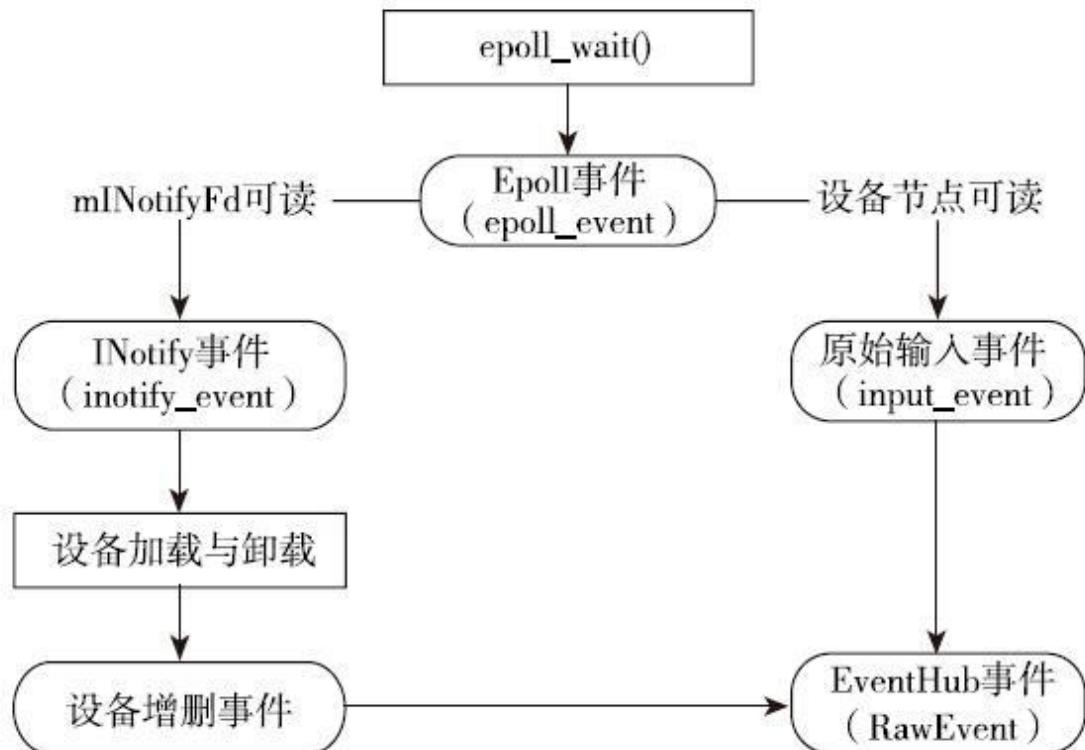


图5-6 EventHub的事件关联

另外，`getEvents ()` 函数返回的事件列表依照事件的优先级拥有特定的顺序。并且在一个监听周期中，同一输入设备的输入事件在列表中是相邻的。

至此，相信读者对EventHub的工作原理，以及EventHub的事件监听与读取机制有了深入的了解。接下来的内容将讨论EventHub所提供的原始输入事件如何被加工为Android输入事件，这个加工者就是Reader子系统中的另一员大将：InputReader。

5.2.4 深入理解InputReader

前文讨论过，InputReader运行在InputReaderThread中，其线程循环像一台水泵一样，从EventHub中抽取事件，进行加工处理后将处理完成的事件注入InputDispatcher的派发队列中。本节将深入讨论这台水泵的工作原理。

5.2.2节介绍了InputReader的`loopOnce ()` 函数，此函数的工作有三步：

- 通过调用`mEventHub.getEvents ()` 获取事件列表。
- 调用`processEventsLocked ()` 函数处理列表中的所有事件。

·调用mQueuedListener.flush () 将事件注入InputDispatcher的派发队列中。

5.2.3节详细探讨了第一步getEvents () 函数的工作原理。接下来探讨剩余的两个步骤。

1.初识原始输入事件的加工

不用说，了解原始事件加工原理的入口是processEventsLocked () 函数。看一下其实现：

```
[InputReader.cpp-->InputReader::processEventsLocked()] void  
InputReader::processEventsLocked(const RawEvent* rawEvents, size_t  
count) { // 遍历所有事件 for (const RawEvent* rawEvent = rawEvents;  
count;) { int32_t type = rawEvent->type; // 根据事件的类型区分原始输入  
事件或设备增删事件 if (type <  
EventHubInterface::FIRST_SYNTHETIC_EVENT) { ..... // 执行  
processEventsForDeviceLocked()处理属于某一设备的一批事件 //  
batchSize表示属于此设备的输入事件的个数  
processEventsForDeviceLocked(deviceId, rawEvent, batchSize); } else {  
..... // 处理设备增删事件，先不讨论部分内容 } } }
```

processEventsLocked () 会分别处理原始输入事件与设备增删事件。这里先不讨论设备增删事件的处理。对于原始输入事件，由于EventHub

会将属于同一输入设备的原始输入事件放在一起，因此 processEventsLocked () 可以使processEventsForDeviceLocked () 同时 处理来自同一输入设备的一批事件。

再看processEventsForDeviceLocked () 函数的实现：

```
[InputReader.cpp-->InputReader::processEventsForDeviceLocked()] void
InputReader::processEventsForDeviceLocked(int32_t deviceId, const
RawEvent* rawEvents, size_t count) { /* InputReader中也保存了一个
mDevices字典，这个字典以设备Id为键存储了一系列的InputDevice 对
象。InputDevice是什么呢*/ ssize_t deviceIndex =
mDevices.indexOfKey(deviceId); InputDevice* device =
mDevices.valueAt(deviceIndex); ..... // 调用InputDevice的process()函数对
这批事件进行处理 device->process(rawEvents, count); }
```

出现了一个不曾见过的InputDevice类，先不管它，看一下 InputDevice :: process () 函数的实现再说。

```
[InputReader.cpp-->InputDevice::process()] void InputDevice::process(const
RawEvent* rawEvents, size_t count) { for (const RawEvent* rawEvent =
rawEvents; count--; rawEvent++) { if (mDropUntilNextSync) { ..... // 处理
事件同步错误 } else { for (size_t i = 0; i < numMappers; i++) { /*
InputDevice中有一个InputMapper对象的列表，可以看出实际的输入事
```

件处理位于InputMapper的process()函数中，这些Mapper又是什么呢*/
InputMapper* mapper = mMappers[i]; mapper->process(rawEvent); } } } }

processEventsLocked () 函数处理原始输入事件的逻辑是非常简单的，首先将属于同一设备的输入事件列表交由

processEventsForDeviceLocked () 处理， processEventsForDeviceLocked () 再将事件列表交给InputDevice :: process () 处理， InputDevice :: process () 将事件逐个交给每一个InputMapper的 process () 事件处理。

InputDevice和InputMapper究竟是何方神圣呢？

原来，在InputReader中也有一个类来存储输入设备的信息，它就是 InputDevice。与EventHub一样， InputDevice描述了一个输入设备，并且以设备Id为键保存在mDevices字典中。InputDevice类与EventHub的 Device结构体类似，也保存了设备的Id、厂商信息以及设备所属的类别。它们之间的一个重要差别是， InputDevice相对于Device结构体多了一个InputMapper列表。

从上面的代码可以看到， InputMapper是InputReader中实际进行原始输入事件加工的场所，它有一系列的子类，分别用于加工不同类型的原始输入事件。而InputDevice的process () 函数使用InputMapper的方式是一个简化了的职责链（chain of responsibility）设计模式。InputDevice

不需要知道哪一个InputMapper可以处理一个原始输入事件，只须将一个事件逐个交给每一个InputMapper尝试处理，如果InputMapper可以接受这个事件则处理之，否则什么都不做。

现在我们了解了InputReader处理原始输入事件的大致流程，而且知道了InputDevice和InputMapper二者在处理过程中起到了至关重要的作用。因此在继续分析原始输入事件处理之前，首先要弄清楚InputDevice与InputMapper从何而来。

2. InputDevice与InputMapper

InputDevice既然用来表示一个输入设备，因此不难联想到，InputDevice的创建与销毁操作与EventHub的设备增删事件有关。

processEventsLocked () 函数会根据DEVICE_ADDED事件调用addDeviceLocked () 函数创建InputDevice，并会根据DEVICE_REMOVED事件调用removeDeviceLocked () 将InputDevice删除。对于FINISHED_DEVICE_SCAN事件，InputReader会产生一个ConfigurationChanged事件并将其发给InputDispatcher。

(1) InputDevice的创建

由于InputDevice中所保存的设备信息与EventHub的Device结构体差不多，因此InputDevice的创建过程的主体就是从EventHub中读取InputReader所感兴趣的设备信息，然后根据这些信息创建InputDevice对

象。不过仍有需要关注的知识点。看一下addDevicesLocked () 函数的实现：

```
[InputReader.cpp-->InputReader::addDeviceLocked()] void  
InputReader::addDeviceLocked(nsecs_t when, int32_t deviceId) { // ① 从  
EventHub中获取厂商信息与设备类别 InputDeviceIdentifier identifier =  
mEventHub->getDeviceIdentifier(deviceId); uint32_t classes = mEventHub-  
>getDeviceClasses(deviceId); // ② 通过createDeviceLocked()函数创建一  
个InputDevice对象 InputDevice* device = createDeviceLocked(deviceId,  
identifier, classes); /* ③ 使用InputReader中保存的策略配置信息对新建的  
InputDevice进行策略配置，并通过 reset()进行设备重置 */ device-  
>configure(when, &mConfig, 0); device->reset(when); // ④ 将设备放入  
mDevices字典中 mDevices.add(deviceId, device); }
```

这个函数比较好理解，不过代码中③处内容需要说明一下，通过createDeviceLocked () 创建InputDevice对象之后需要使用mConfig变量对其进行策略配置，注意是“策略”配置。mConfig的类型是InputReaderConfiguration，这一来自InputReaderPolicy的配置信息使得IMS以及应用程序得以在一定程度上影响输入事件的处理过程。在随后对原始输入事件加工过程的详细探讨中将会对这一策略配置所产生的影响进行介绍。

接下来看createDeviceLocked () 的实现：

```
[InputReader.cpp-->InputReader::createDeviceLocked()] InputDevice*
InputReader::createDeviceLocked(int32_t deviceId, const
InputDeviceIdentifier& identifier, uint32_t classes) { /* ① 根据设备Id、厂
商信息以及设备类型创建一个InputDevice对象。InputDevice的构造函
数保存了这些信息，并没有进行其他操作 */ InputDevice* device = new
InputDevice(&mContext, deviceId, bumpGenerationLocked(), identifier,
classes); ..... // ② 后续的代码将按照设备类型为InputDevice添加特定类
型的InputMapper if (classes & INPUT_DEVICE_CLASS_SWITCH) {
device->addMapper(new SwitchInputMapper(device)); } ..... return device;
}
```

InputDevice的创建过程就是如此。新建的InputDevice保存了设备Id、厂商信息，以及设备类型，并根据设备类型向InputDevice中添加了一系列各种类型的InputMapper。InputReader还会使用其当前的策略配置信息对InputDevice进行配置，使其事件处理过程能够根据IMS或应用程序的需求进行调整。

(2) InputMapper的分配

如前面所述，InputMapper完成了原始输入事件的加工处理，因此了解InputMapper的分配依据十分重要。

从createDeviceLocked () 函数的实现可知，InputMapper的分配依据是设备类型，这个设备类型来自EventHub的Device结构体。在5.2.3节分析EventHub的设备管理相关内容时提到过Device结构体中所保存的设备类别的设置依据来自从设备节点中读取的事件位掩码。

Device结构体的事件位掩码描述了4种类型的输入事件：

- EV_KEY，按键类型的事件。能够上报这类事件的设备有键盘、鼠标、手柄、手写板等一切拥有按钮的设备（包括手机上的实体按键）。在Device结构体中，对应的事件位掩码keyBitmask描述了设备可以产生的按键事件的集合。按键事件的全集包括字符按键、方向键、控制键、鼠标键、游戏按键等。
- EV_ABS，绝对坐标类型的事件。这类事件描述了在空间中的一个点，触控板、触摸屏等使用绝对坐标的输入设备可以上报这类事件。事件位掩码absBitmask描述了设备可以上报的事件的维度信息（ABS_X、ABS_Y、ABS_Z），以及是否支持多点事件。
- EV_REL，相对坐标类型的事件。这类事件描述了事件在空间中相对于上次事件的偏移量。鼠标、轨迹球等基于游标指针的设备可以上报此类事件。事件位掩码relBitmask描述了设备可以上报的事件的维度信息（REL_X、REL_Y、REL_Z）。

·EV_SW，开关类型的事件。这类事件描述了若干固定状态之间的切换。手机上的静音模式开关按钮、模式切换拨盘等设备可以上报此类事件。事件位掩码swBitmask表示了设备可切换的状态列表。

另外还有两个事件位掩码描述了设备的反馈能力。ledBitmask描述了设备是否支持光反馈，而ffBitmask则描述了设备是否支持力反馈。对于这类设备，EventHub提供了接口setLedState () 与vibrate () 来启动光反馈与力反馈。不过这不是本章的讨论重点，感兴趣的读者可以自行研究。

在EventHub的openDeviceLocked () 函数通过事件位掩码确定了设备可以上报的事件类型后，便可以据此确定设备的类型了。Android在EventHub.h中定义了12种设备类型，为了叙述简洁，我们省略了INPUT_DEVICE_CLASS_前缀：

- KEYBOARD，可以上报鼠标按键以外的EV_KEY类型事件的设备都属于此类。如键盘、机身按钮（音量键、电源键等）。
- ALPHAKEY，可以上报字符按键的设备，例如键盘。此类型的设备必定同时属于KEYBOARD。
- DPAD，可以上报方向键的设备。例如键盘、手机导航键等。这类设备同时也属于KEYBOARD。

·GAMEPAD，可以上报游戏按键的设备，如游戏手柄。这类设备同时也属于KEYBOARD。

·TOUCH，可以上报EV_ABS类型事件的设备都属于此类，如触摸屏和触控板。

·TOUCH_MT，可以上报EV_ABS类型事件，并且其事件位掩码指示其支持多点事件的设备属于此类。例如多点触摸屏。这类设备同时也属于TOUCH类型。

·CURSOR，可以上报EV_REL类型的事件，并且可以上报BTN_MOUSE子类的EV_KEY事件的设备属于此类，例如鼠标和轨迹球。

·SWITCH，可以上报EV_SW类型事件的设备。

·JOYSTICK，属于GAMEPAD类型，并且属于TOUCH类型的设备。

·VIBRATOR，支持力反馈的设备。

·VIRTUAL，虚拟设备。

·EXTERNAL，外部设备，即非内建设备。如外接鼠标、键盘、游戏手柄等。

确定设备类型之后，InputReader的createDeviceLocked（）函数为InputDevice分配InputMapper。除了VIRTUAL和EXTERNAL没有对应的InputMapper，以及KEYBOARD、AL-PHKEY、DPAD与GAMEPAD 4者共用KeyboardInputMapper以外，InputReader为每种设备类型定义了对应的InputMapper。图5-7直观地展现了从事件类型到InputMapper的分配过程。

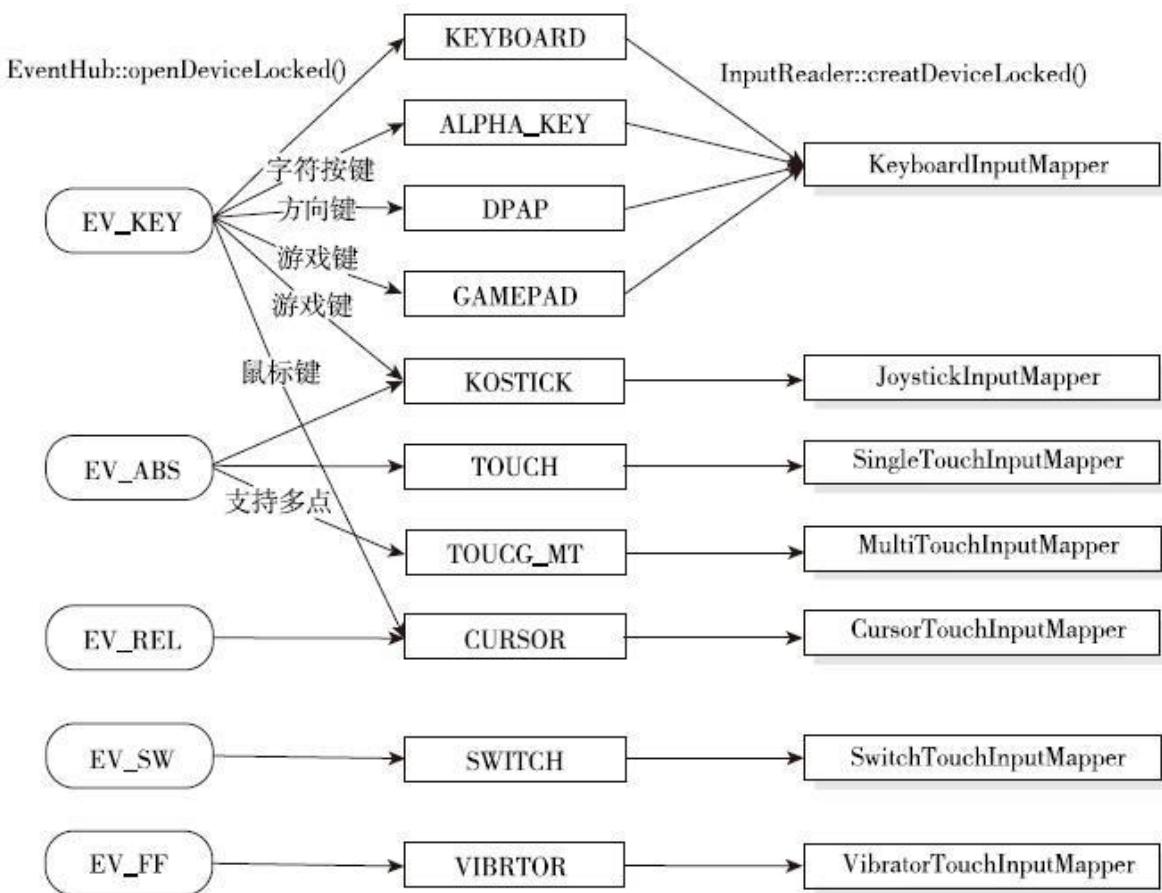


图5-7 InputMapper的分配



注意

一个设备节点可能托管很多种类型的物理输入设备。例如Android 4.2的模拟器中只有event0一个设备节点，但是其负责按键、触摸屏事件的上报工作。此时，这个设备节点所对应的InputDevice将会拥有KeyboardInputMapper与SingleTouchInputMapper两个InputMapper。不过得益于InputMapper的职责链式的设计模式，一个InputDevice处理多种输入事件一点也不吃力。

当InputDevice的InputMapper分配完成之后，便随时可以进行原始输入事件的处理了！

3. Keyboard类型事件的加工处理

InputMapper的种类实在很多，对所有类型都进行详细分析并不现实。本章选择KeyboardInputMapper和MultiTouchInputMapper两个常用并且具有代表性的InputMapper进行探讨。在理解这两种InputMapper后，如果读者感兴趣可以对其他类型自行研究。

(1) KeyboardInputMapper的配置

InputDevice创建完成后，会通过InputReader下的mConfig成员变量对其进行策略配置，以影响原始输入事件的处理行为。InputDevice的InputMapper自然也会随InputDevice一起进行配置。对KeyboardInputMapper来说，有哪些策略配置信息会影响其工作呢？

KeyboardInputMapper所感兴趣的信息是屏幕旋转状态。想象下面这种情况，当一个有4个方向键的手机顺时针旋转90度横置时，这4个方向键的功能是否会发生改变呢？确实如此，在这种情况下，原来的上方向键变成了右方向键，原来的右方向键变成了下方向键，以此类推。因此手机的旋转状态成为影响按键事件处理的因素。InputReaderConfiguration类的getDisplayInfo()函数可以提供屏幕旋转信息。读者可以自行参考KeyboardInputMapper::configure()函数的实现。



注意

由此可见，当屏幕旋转状态发生变化时，需要更新InputReaderConfiguration并对KeyboardInputMapper进行重新配置。

(2) 键盘扫描码与虚拟键值

在正式开始探讨KeyboardInputMapper的事件处理过程之前，首先需要弄清楚两个概念：键盘扫描码（ScanCode）与虚拟键值（KeyCode）。

键盘扫描码这个概念来自矩阵式键盘。所谓矩阵式键盘是指在其内部存在由若干行线与列线构成的开关矩阵电路，而每一个按键位于行线与列线的相交处的键盘。当按键按下时，行线与列线接通，如图5-8所示。键盘中的电路会不断地对行线进行扫描，被扫描的行线被设置为低电平（0），其他行线被设置为高电平（1），然后通过检查列线中是否有低电平（0）以确定是否有列线与当前行线连通，进而确定被按下的按键位置。参考图5-8，当扫描R0行时，因为R0与C0、C1未连通，因此列线C0、C1保持高电平（1），由此可以得知R0行没有按键按下。当扫描R1行时，因为R1与C0连通，使得C0变为低电平（0），由此便可得知R1C0位置的C键被按下。然后将R0R1C0C1（1001）的电平状态编码为二进制的形式组合到一起送入主机。由于这一代表特定按键的二进制码是通过扫描的方式获得的，因此被称为键盘扫描码。在本例中，二进制1001即是C键的扫描码。

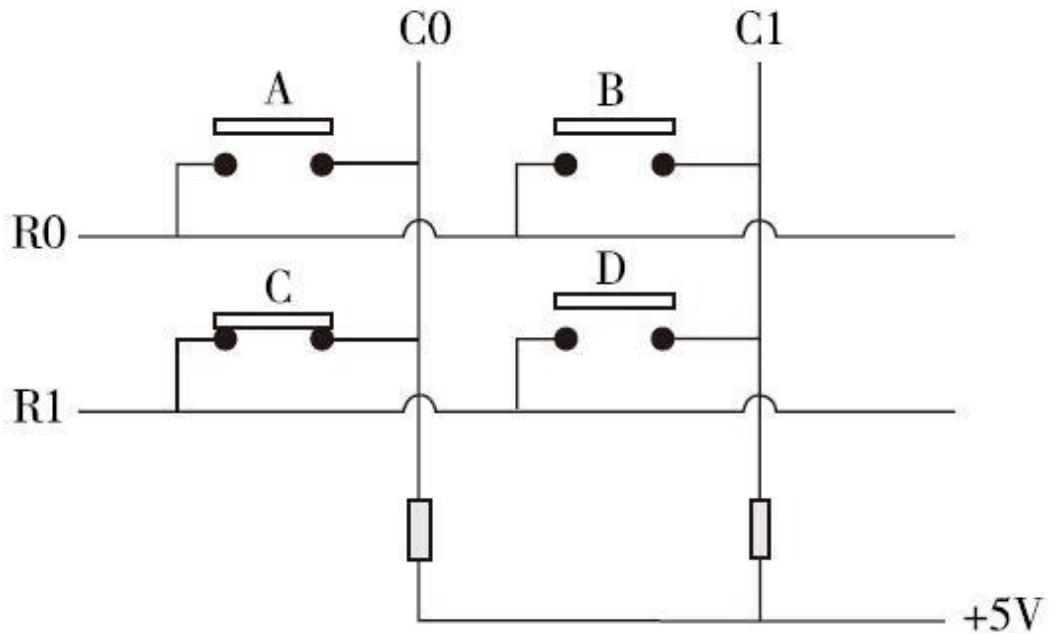


图5-8 矩阵式键盘的工作原理

可见扫描码与硬件实现关系非常密切，同一个按键的扫描码在不同的键盘中可能不一样，所以扫描码并不适合在软件开发中用作标识按键。为此，操作系统每个已知按键定义了唯一的虚拟键值，并通过驱动程序或其他方式在扫描码与虚拟键值之间建立映射，以屏蔽因硬件原因所带来的扫描码差异。在不同的操作系统中，虚拟键值的定义也不一样。

因此，扫描码与虚拟键值的相同点在于它们都描述了一个具体的按键。而区别在于范畴不同，扫描码是硬件实现范畴，而虚拟键值则是操作系统的实现范畴。

在Android中，扫描码到虚拟键值的映便是由KeyboardInputMapper完成的。

(3) 扫描码到虚拟键值的映射

看一下KeyboardInputMapper的process () 函数的实现：

```
[InputReader.cpp-->KeyboardInputMapper::process()] void
KeyboardInputMapper::process(const RawEvent* rawEvent) { switch
(rawEvent->type) { case EV_KEY: { // 按键类的原始输入事件类型为
EV_KEY (0x01) int32_t scanCode = rawEvent->code; // 事件的代码保存
了键盘扫描码 int32_t usageCode = mCurrentHidUsage;
mCurrentHidUsage = 0; if (isKeyboardOrGamepadKey(scanCode)) { /* 排
除对鼠标按键的处理。鼠标按键由 CursorInputMapper 处理 */
int32_t
keyCode; uint32_t flags; /* ① 通过EventHub的mapKey()函数进行映射。
映射的输入为scanCode和usageCode，而输出为keyCode和flags */ if
(getEventHub()->mapKey(getDeviceId(), scanCode, usageCode, &keyCode,
&flags)) { // 如果映射失败（例如扫描码不被EventHub识别），则使用
UNKNOWN作为事件的 虚拟键值keyCode = AKEYCODE_UNKNOWN;
flags = 0; } /* ② 对映射后的事件做进一步处理 */
processKey(rawEvent-
>when, rawEvent->value != 0, keyCode, scanCode, flags); } break; } ..... }
}
```

KeyboardInputMapper的process () 函数比较简单。有以下两个主要工作：

- 通过EventHub::mapKey () 函数将原始事的扫描码映射为Android系统所定义的虚拟键值。
- 通过processKey () 按键事件做进一步处理。

另外，这段代码也揭示了Keyboard类型（EV_KEY）的原始输入事件除时间戳外的基本元素的意义：

- Type， EV_KEY， 表示此事件类型为Keyboard事件。
- Code， 存储了按键的扫描码。
- Value， 1表示按键按下， 0表示按键抬起。

EventHub如何从扫描码映射为虚拟键值呢？

如前面所述，在EventHub通过openDeviceLocked () 函数加载输入设备时，会根据设备所提供的事件位掩码为设备分配类别。当设备的类别为INPUT_DEVICE_CLASS_KEYBOARD/JOYSTICK时，说明此设备可以上报EV_KEY类型的原始输入事件，因此openDeviceLocked () 会调用EventHub::loadKeyMapLocked () 函数为此设备加载键盘布局配置文件。键盘布局配置文件的路径是由设备配置文件中的

keyboard.layout项与设备厂商信息联合产生的位于/system/usr/input/下的一个.kl文件。这个.kl文件的每一行描述了一个从扫描码到虚拟键值的映射信息。.kl文件中一行的典型格式如下：

key <scan_code> <key_name> <policy_flag>

·Key，关键字，表示本行描述了一个键盘映射。

·scan_code，十进制的扫描码。

·key_name，虚拟键值的名称。例如“A”代表按键A，“POWER”代表电源键。虚拟键值的名称到虚拟键值的映射可以在frameworks/base/include/androidfw/keycodelabels.h中定义的KEYCODES数组中找到。

·policy_flag，表明此按键所附加的策略的名称。例如“WAKE”表示此按键具有唤醒功能。“ALT”则表示此按键将被翻译为指定的虚拟键值+Alt键的组合键。每个名称所定义的数值可以在keycodelabels.h的FLAGS数组中找到。

例如，描述电源键的一行如下所示：

key 116 POWER WAKE

loadKeyMapLocked () 函数分析.kl文件，每一行都将名称转换为虚拟键值后存储在Key结构体中。所得到的Key结构体集合以扫描码为键，存储在KeyLayoutMap对象中。于是，EventHub :: mapKey () 函数可以根据设备Id找到对应的KeyLayoutMap，进而根据扫描码找到对应的Key结构体中所保存的虚拟键值（传出参数keycode）以及策略值（传出参数flag），从而完成从扫描码到虚拟键值的映射工作。如图5-9所示。



图5-9 扫描码到虚拟键值的映射过程

(4) 按键事件的加工处理

根据扫描码获取虚拟键值以及功能值后，KeyboardInputMapper :: process () 调用了processKey () 函数对按键事件做进一步处理。

```
[InputReader.cpp--> KeyboardInputMapper::processKey()] void  
KeyboardInputMapper::processKey(nsecs_t when, bool down, int32_t  
keyCode, int32_t scanCode, uint32_t policyFlags) { // processKey()会对按
```

键的按下与抬起分别进行处理 if (down) { // ① 当按下时，首先需要根据屏幕的方向对按键的虚拟键值进行旋转变换 if
(mParameters.orientationAware && mParameters.hasAssociatedDisplay) {
keyCode = rotateKeyCode(keyCode, mOrientation); } /* ② 从接下来的代码中可以看出，KeyboardInputMapper维护了一个KeyDown结构体的集合。当按键按下时会生成一个保存了扫描码与键值的KeyDown对象并添加到集合中。通过扫描码对集合的查找结果表明了按键是否是重复按下 */ ssize_t keyDownIndex = findKeyDown(scanCode); if
(keyDownIndex >= 0) { /* 对于重复按下的按键，需要确保后续的处理过程中的虚拟键值与第一次按下时一致，以免重复的过程中屏幕方向的变化导致虚拟键值的变化，使得后续InputDispatcher无法正常识别重复按键的动作 */ keyCode =
mKeyDowns.itemAt(keyDownIndex).keyCode; } else { // 生成KeyDown结构体并添加到集合中 mKeyDowns.push(); KeyDown& keyDown =
mKeyDowns.editTop(); keyDown.keyCode = keyCode; keyDown.scanCode = scanCode; } } else { ssize_t keyDownIndex = findKeyDown(scanCode); // ③ 对于抬起的按键，将对应的KeyDown对象从集合中移除 if
(keyDownIndex >= 0) { keyCode =
mKeyDowns.itemAt(keyDownIndex).keyCode;
mKeyDowns.removeAt(size_t(keyDownIndex)); } else { return; // 如果设备节点上报了一个并未按下的按键的抬起事件，这个事件将被忽略 } } /*

④ 接下来设置metaState。所谓的metaState是指控制键的按下状态。控制键有左右Shift、 Alt、 Ctrl、 Fn、 CapsLock、 NumLock、 ScrollLock 等。当这些按键被按下或抬起时， mMetaState会将相应的位置1或置0

```
/* bool metaStateChanged = false; int32_t oldMetaState = mMetaState; //  
updateMetaState 会检查此次按键是否为控制键并返回新的metaState  
int32_t newMetaState = updateMetaState(keyCode, down, oldMetaState); if  
(oldMetaState != newMetaState) { // 保存新的metaState到mMetaState  
mMetaState = newMetaState; metaStateChanged = true; } if  
(metaStateChanged) { /* 使InputReader的Context更新全局的metaState。  
Context保存了在InputReader中可以 被所有InputMapper使用的信息。注  
意与Android API中的Context不是一个概念 */ getContext()->  
>updateGlobalMetaState(); } ..... // ⑤ 将所有的按键事件信息封装为  
NotifykeyArgs对象， 并将此对象通知给listener NotifyKeyArgs  
args(when, getDeviceId(), mSource, policyFlags, down ?  
AKEY_EVENT_ACTION_DOWN : AKEY_EVENT_ACTION_UP,  
AKEY_EVENT_FLAG_FROM_SYSTEM, keyCode, scanCode,  
newMetaState, downTime); getListener()->notifyKey(&args); }
```

按键事件的加工很简单，主要动作就是根据屏幕旋转方向进行虚拟键值的变换，设置metaState以及容错处理。

进行变换的目的是使得按键动作能够体现其在某个屏幕方向下的实际意义（如方向键）。这个变换过程是通过定义在InputReader.cpp中的一个名为keyCodeRotationMap的二维数组完成的。

metaState描述了功能键的按下状态。事件的接受者往往需要根据Ctrl、Alt以及Shift等按键的按下状态对输入事件做特殊处理（例如C键的metaState为Ctrl按下时进行复制操作），因此metaState是输入事件的一个重要信息。另外，不止是按键事件，其他类型事件也要携带metaState信息。因此KeyboardInputMapper会调用getContext () ->updateGlobalMetaState () 将最新产生的metaState更新至InputReader的Context中，当其他的InputMapper产生事件时可以从Context里获取最新的metaState并添加到自己的事件信息里。

容错处理主要是为了处理重复的按下与抬起事件。

值得注意的是，在代码里最后调用getListener () ->notifykey () 函数。到达这个调用时，按键事件已经在InputReader中加工完毕，准备提交给InputDispatcher进行派发。这个调用就是提交给InputDispatcher的第一步。在本节（5.2.4）的第5小节将介绍输入事件从InputReader提交到InputDispatcher的过程。

（5）Keyboard类型事件的处理小结

本小节介绍了KeyboardInputMapper的加工过程。Keyboard类型事件比较简单，一条原始输入事件即可完整地描述一次按键动作，因此KeyboardInputMapper的加工过程也很简单。在本节中需要读者注意的关键点如下：

- 扫描码与虚拟键值的概念与区别。
- 扫描码到虚拟键值的映射过程。
- 虚拟键值根据屏幕方向进行变换的过程与意义。
- 对于重复按下与重复抬起事件的容错处理。

4.Touch类型事件的加工处理

虽然Touch类型事件与Keyboard类型事件的处理共用了一套InputMapper框架，但是由于Touch类型事件所包含的信息量多很多，因此其处理过程也复杂很多。这一节将探讨Touch类型原始输入事件的处理原理，以及Android处理复杂输入事件的方式。

(1) Touch类型事件的信息与原始事件的组织方式

Touch类型事件描述了用户的一次点击操作，对应于EV_ABS类型的原始输入事件。点击的对象可以是触摸屏、触控板等，我们可以统称其

为传感器，而发生点击动作的可以是手指，也可以是触控笔，可以称为触摸工具。

Android输入系统对一次点击操作的描述信息的支持非常多，对一次点击操作的完整描述信息包括以下内容：

- 点击坐标，描述点击操作的坐标位置。和点击坐标有关的传感器属性包括坐标精度（传感器的密集程度）、坐标范围（传感器总面积）以及噪音漂移（传感器的抗噪能力）等。

- 点击区域，无论是手指还是触控笔，落在传感器上不可能是一个点，而是一个区域。如果支持点击区域的识别，输入设备使用一个长轴（Major Axis）及短轴（Minor Axis）表示的椭圆来近似地描述触摸工具落在传感器上的区域。

- 触摸工具到传感器的距离，有些高级的传感器可以识别悬停在其上的触摸工具，因此对于这种设备，触摸工具到传感器的距离是一项必要的信息。

- 压力，高级的传感器可以识别触摸工具施加在传感器上的压力大小。

- 触摸工具的类型，有些传感器可以对手指及触控笔做出区分。

由于一次点击动作包含如此多的信息，因此一条原始输入事件是无法完整描述的。为了解决这个问题，Linux的输入子系统使用多个原始输

入事件对一次复杂的输入事件进行描述，其中每一个事件描述一项信息，并在完整描述设备所产生的所有信息之后，使用一条特殊类型的事件标识一次输入事件上报的完成。

举例来说，在Android模拟器上点击鼠标并完成一次移动，可以通过getevent工具得到以下三条原始事件的输出：

```
[ 278.791198] /dev/input/event0: 0003 0000 000000b6 [ 278.791271]  
/dev/input/event0: 0003 0001 00000229 [ 278.791313] /dev/input/event0:  
0000 0000 00000000
```

前两条的事件类型为0x03，即EV_ABS，表示它描述点击事件的一条信息。其Code字段的值分别为ABS_X（0x00）与ABS_Y（0x01），表示它们分别携带X坐标和Y坐标的信息。而第三条的事件类型为EV_SYN（0x00），并且其Code为SYN_REPORT，即所谓的表示事件上报完成可以进行派发的特殊类型的事件。

因此，Touch类型事件的加工处理其实就是收集所有描述点击动作的原始输入事件所包含的各种新信息，并在EV_SYN类型事件到来时，将这些信息进行整合，在进行一些处理之后，交付给InputDispatcher进行分发。

（2）TouchInputMapper的体系

负责处理点击事件的InputMapper是TouchInputMapper。由于点击事件的信息量大，并且还分为单点与多点两种类型，TouchInputMapper的组成与逻辑相比KeyboardInputMapper要复杂得多。为了减少后面深入学习的麻烦，首先要了解TouchInputMapper的体系结构与参与者的作用，如图5-10所示。

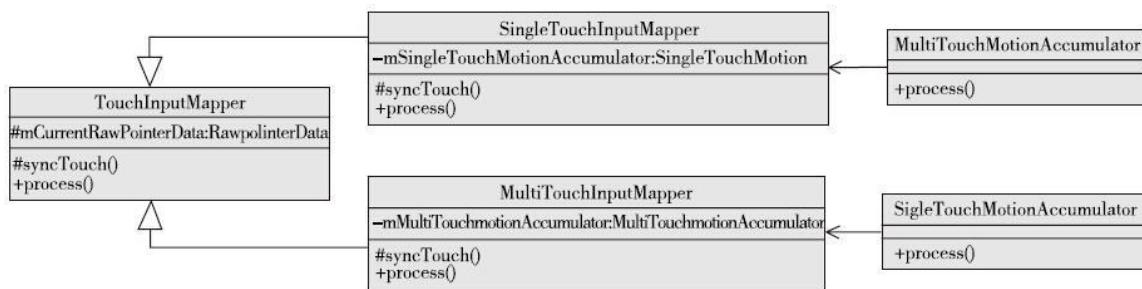


图5-10 TouchInputMapper的体系

TouchInputMapper的点击事件处理过程中的参与者有以下几个：

- TouchInputMapper，负责在接收到EV_SYN事件时调用子类的 syncTouch () 函数，令子类将获取的描述点击事件的信息存储到其 mCurrentRawPointerData对象中。将mCurrentRawPointerData解析为 ACTION_DOWN、ACTION_MOVE以及ACTION_UP等高级点击事件后，将其交付给InputDispatcher。
- MultiTouchInputMapper，主要是为了实现syncTouch () 函数，目的是将MultiTouch-MotionAccumulator收集到的点击事件信息存储到 mCurrentRawPointerData中。另外，MultiTouchInputMapper也负责从

EventHub中获取与多点触控相关的设备配置信息，如所支持的触控点数量以及各项信息的取值范围、精度、噪音偏移量等指标。

- MultiTouchMotionAccumulator，多点触控信息的累加器。它负责接收处理每条EV_ABS事件，并将事件中携带的点击信息保存下来，并在EV_SYN事件到来时提供给MultiTouchInputMapper。
- SingleTouchInputMapper与SingleTouchMotionAccumulator，负责单点触控事件的收集与整合工作。其工作方式与多点触控类似。

本节将会以多点触控的事件解析与整合为例，介绍TouchInputMapper的工作原理。

(3) MultiTouchInputMapper的配置

MultiTouchInputMapper及其累加器的构造函数没有什么内容。事件信息收集与整合所需要的信息的获取位于其配置过程中，也就是其基类TouchInputMapper的configure () 函数中。

TouchInputMapper的configure () 函数的调用时机同KeyboardInputMapper一样。在TouchInputMapper被创建之后对configure () 的第一次调用中，TouchInputMapper会调用由子类实现的configureRawPointerAxes () 函数用以获取由设备节点提供的技术信息。这些技术信息包括输入设备所支持的触控点数量以及各项信息的

取值范围、精度、干扰偏移量等指标。它们是通过EventHub，使用ioctl的方式获取的。一项信息的技术指标由RawAbsoluteAxisInfo结构体描述。其定义如下：

```
[EventHub.h-->RawAbsoluteAxisInfo] struct RawAbsoluteAxisInfo { bool valid; //此项信息是否受到输入设备的支持 int32_t minValue; //此项信息的最小值 int32_t maxValue; //此项信息的最大值 /* 以下三个字段为精度信息 */ int32_t flat; int32_t fuzz; //容错范围。表示因干扰所导致的最大偏移量 int32_t resolution; //精度，表示在1毫米的范围中的解析点的数量 }
```

例如，对设备的X坐标这项信息而言，RawAbsoluteAxisInfo的minValue和maxValue表示了事件上报的X坐标范围，Resolution则表示传感器在每毫米距离中可以产生的X坐标点的数量（ppm）。X坐标与Y坐标的这些技术指标构建了传感器的物理坐标系。而对压力值这项信息而言，如果其RawAbsoluteAxisInfo的valid值为false，则表示此设备不支持识别压力值。当然，这些字段并不是对所有类型的信息都有效，例如对于传感器所支持的触控点数量这项信息，仅有valid和maxValue两个字段有效，valid为true表示设备支持通过slot协议进行触控点索引的识别，而maxValue则表示最大触控点的数量为maxValue+1。

所有信息的技术指标被MultiTouchInputMapper保存在它的一个名为mRawPointerAxes的RawPointerAxes结构体中。

另外，TouchInputMapper的configure（）调用的configureSurface（）函数会通过Display-ViewPort获取屏幕的方向以及屏幕坐标系的信息。屏幕方向的不同会导致触控位置的X和Y两个方向的颠倒或交换。通过计算屏幕坐标系与传感器物理坐标系之间的差异，使得在事件到来时可以将点击位置从传感器的物理坐标系转换到屏幕坐标系中。



注意

读者可以从键盘的扫描码与虚拟键值的关系中类比地理解物理坐标系与屏幕坐标系的概念。EV_ABS类型的原始输入事件上报的坐标系是传感器的物理坐标系，其范围与设备的硬件实现有关，并且与屏幕的实际坐标系有较大出入。举个极端的例子，一个做得很粗糙的触摸屏在X方向上的分辨率仅为10，当点击发生在屏幕右边缘时，其原始输入事件上报的X坐标为10，这是输入设备的物理坐标。然而手机所配备的屏幕在X方向上的分辨率为1920，这是屏幕坐标。因此必须将物理坐标值10根据两个坐标系的比例与位置关系，换为屏幕坐标系下的1920才能使用。

总结一下MultiTouchInputMapper的配置内容：

- MultiTouchInputMapper的configureRawPointerAxes () 函数获取来自设备节点的各项触控信息的技术指标。同时，这些指标构建了传感器的物理坐标系。
- TouchInputMapper的configureSurface () 函数获取来自DisplayViewPort 的屏幕方向以及屏幕坐标系的信息，并计算物理坐标系到屏幕坐标系的差异信息。

(4) 点击事件信息的收集

看一下MultiTouchInputMapper的process () 函数的实现：

```
[InputReader.cpp-->MultiTouchInputMapper::process] void  
MultiTouchInputMapper::process(const RawEvent* rawEvent) { // 调用基  
类的process()函数。基类的process()函数主要处理EV_SYN事件并进行  
事件信息整合 TouchInputMapper::process(rawEvent); // 转而调用  
MultiTouchMotionAccumulator::process()进行事件信息收集  
mMultiTouchMotionAccumulator.process(rawEvent); }
```

这个函数首先调用基类的process () 函数，用以处理EV_SYN事件，下一节再详细讨论这个处理。由于事件信息的收集工作交由 MultiTouchMotionAccumulator类的process () 函数完成。在多点触控事

件的信息收集工作中，需要识别信息所属的触控点索引。如何识别触控点的索引有两种方式：基于Slot协议的显式识别方式与基于事件顺序的隐式识别方式。我们先分析显式识别下的信息收集方式，然后再探讨隐式识别与显式识别的区别。

参考以下代码（注意，这段代码删除了与隐式识别相关的代码）：

```
[InputReader.cpp-->MultiTouchMotionAccumulator::process()] void  
MultiTouchMotionAccumulator::process(const RawEvent* rawEvent) { // ①  
    点击原始输入事件的类型为EV_ABS if (rawEvent->type == EV_ABS) {  
        bool newSlot = false; if (rawEvent->code == ABS_MT_SLOT) { // ② 代码  
            为ABS_MT_SLOT的事件指明后续事件对应的触控点的索引  
            mCurrentSlot = rawEvent->value; newSlot = true; } if (mCurrentSlot < 0 ||  
            size_t(mCurrentSlot) >= mSlotCount) { ..... // mCurrentSlot越界，打印一  
            条警告信息而已 } else { // ③ 根据触控点的索引获取一个Slot对象。随  
            后所收集的信息将放置在这个Slot对象中 Slot* slot =  
            &mSlots[mCurrentSlot]; // ④ 提取事件中携带的信息，并存储在触控点  
            所对应的Slot对象中 switch (rawEvent->code) { // 原始输入事件的代码指  
            明了它所携带的信息类型 case ABS_MT_POSITION_X: slot->mInUse =  
            true; // 设置mInUse为true表示这个slot中包含有效的信息 slot-  
            >mAbsMTPositionX = rawEvent->value; // 将信息保存在Slot相应的字段  
            里 break; ..... // 其他信息的收集 } } }
```

MultiTouchMotionAccumulator保存了一个名为mSlots的Slot对象数组。Slot对象是收集特定触控点的点击信息的场所，而其在数组中的索引就是触控点的Id。从这段代码中可以看到，当输入设备开始上报某一个触控点的信息时，首先会发送一条类型为EV_ABS，代码为ABS_MT_SLOT的事件（后面以类型+代码的方式称呼这类事件），这个事件的值表明随后携带点击信息的事件属于哪一个触控点。携带信息的事件到来时，其事件代码指明了它所携带的信息类型，MultiTouchMotionAccumulator从mSlots数组中以触控点的id为索引将对应的Slot对象取出，并将信息值保存在Slot的对应字段中。随着携带信息的事件不断到来，Slot对象中的信息也渐渐丰满起来。当设备将这个触控点的点击信息全部上报完毕后，会发送一个EV_SYN+SYN_REPORT事件启动TouchInputMapper的信息整合与加工工作，或者发送另一个事件代码为ABS_MT_SLOT的事件，开始上报另一个触控点的信息。

由此可见，携带点击信息的原始输入事件的格式如下：

- Type，取值为EV_ABS，表明属于点击事件。
- Code，取值为ABS_MT_XXX，其中的XXX表明了信息的类型。
- Value，信息的值。

而在一个点击信息收集过程中所收到的原始输入事件序列如下所示：

EV_ABS ABS_MT_SLOT 触控点1 EV_ABS 信息1 信息1的值 EV_ABS
信息2 信息2的值 EV_ABS ABS_MT_SLOT 触控点2 // 触控点2的信
息以及更多的触控点 EV_SYN SYN_REPORT

因此整个点击事件信息的收集过程从指定触控点的事件开始，以一条
EV_SYN+SYN_REPORT事件结束。在这个过程后，
MultiTouchMotionAccumulator便在mSlots数组中保存了一个或多个触控
点的点击信息。

这种通过ABS_MT_SLOT事件代码由设备显式地指明触控点的方式称
为显式识别方式，也叫slot协议方式。这种方式的好处是灵活多变，而
坏处是，当只有0号触控点处于活动状态时（也就是单点操作）仍需要
一个ABS_MT_SLOT事件，这是一种浪费。

另外一种识别方式为隐式识别。在隐式识别方式下，代表触控点信息
的事件组由EV_SYN+SYN_MT_REPORT的事件分隔（注意不是标志开
始进行信息整合的代码SYN_REPORT）。分割的第一批事件携带的信
息被认为属于0号索引点的触控点，随后事件组所属的触控点索引依次
增加1。最后，使用EV_SYN+SYN_REPORT事件通知开始进行信息的
整合工作。因此，隐式识别下的原始输入事件序列如下：

EV_ABS 信息1 信息1的值 // 0号触控点的信息 EV_ABS 信息2 信息2的
值 EV_SYN SYN_MT_REPORT // 1号触控点的信息 EV_SYN

SYN_MT_REPORT EV_ABS 信息1 信息1的值 // 2号触控点的信息

EV_SYN SYN_REPORT

隐式识别的好处在于，当单点操作时，不需要额外的ABS_MT_SLOT事件对触控点索引进行指明，从而减少单点操作情况下的原始输入事件数量与处理时间。而缺点也很明显，在多点操作过程中，如果低索引值的触控点陆续抬起，仅剩下一个高索引值的触控点处于活动状态时，在上报其信息之前，仍需相应数量EV_SYN+SYN_MT_REPORT事件进行补齐，这也是一种浪费。

因此两种方式各有利弊，一个多点触控设备采取哪种方式应当根据其应用场景进行适合选择。

(5) 点击事件信息的整合、变换与高级事件的生成

如前面所述，事件信息的整合加工工作由EV_SYN+SYN_REPORT触发。看一下TouchInputMapper的process () 函数：

```
[InputReader.cpp-->TouchInputMapper::process()] void
TouchInputMapper::process(const RawEvent* rawEvent) { ..... if
(rawEvent->type == EV_SYN && rawEvent->code == SYN_REPORT) {
sync(rawEvent->when); // 通过sync()函数开始整合 } }
```

再看一下sync () 函数的实现，这个函数比较长而且复杂，因为它需要整合加工触摸、笔触（Stylus）以及悬停的多种事件。为了简单起见，只保留了与触摸事件相关的代码：

```
[InputReader.cpp-->TouchInputMapper::sync()] void
TouchInputMapper::sync(nsecs_t when) { /* ① 调用子类也就是
MultiTouchInputMapper的syncTouch()函数。将mSlots数组中的触控点信
息 转存到mCurrentRawPointerData中 */ syncTouch(when,
&havePointerIds); if (mDeviceMode == DEVICE_MODE_DISABLED) { //如果设备被禁用，则不做处理 } else { ..... /* ② 通过cookPointerData()函
数将mCurrentRawPointerData中的触控点信息从传感器的物理坐标系转
换到屏幕坐标系 */ cookPointerData(); if (mDeviceMode ==
DEVICE_MODE_POINTER) { ..... } else { ..... // ③ 进行高级触摸事件的
生成与派发 dispatchTouches(when, policyFlags); } ..... } // ④ 将本次整合
触控点信息保存到mLastRawPointerData中
mLastRawPointerData.copyFrom(mCurrentRawPointerData); ..... }
```

在移除不重要的代码后，sync () 函数的工作也就清晰了很多。

首先转存子类进行触控点的点击信息。TouchInputMapper使用了一个名为mCur-rentRawPointerData的RawPointerData对象接收来自子类的触控点的点击信息。这个收集过程由syncTouch () 函数完成。

RawPointerData类其实是其内部Pointer类的集合。Pointer类保存了一个

触控点的点击事件信息，并且它和MultiTouchInputMapper的Slot对象内容是一致的。因此syncTouch () 的主要工作便非常明了，就是将mSlots中存有点击信息的Slot对象转存到mCurrentRawPointerData所保存的Pointer对象中。

将收集到的触控点信息转存到mCurrentRawPointerData中后，sync () 函数便通过调用cookPointerData () 函数完成将触控点信息从传感器物理坐标系到屏幕坐标系的变换过程。变换的依据是在TouchInputMapper的配置过程中所获取的屏幕坐标系信息以及传感器物理坐标系信息的差异。cookPointerData () 的名字非常形象，基于物理坐标系的触控点信息对Android来说是尚不可食的生的食物，转换到屏幕坐标系后就是可口的佳肴了。

坐标系的转换完成后便通过dispatchTouches () 函数将当前的触控点信息与上次触控点信息进行比对，为每个触控点生成ACTION_DOWN、ACTION_MOVE以及ACTION_UP等高级触摸事件，然后与KeyboardInputMapper一样，将事件封装为NotifyMotionArgs对象，并通过getListener->notifyMotion () 调用，准备将事件提交给InputDispatcher进行派发。其中，Android在生成高级触摸事件时使用了位操作这种较为晦涩的方法，不过其原理非常简单——前后两次触控点列表的比对。如果本次的一个触控点没有出现在上次的触控点列表中，则表示此触控点刚刚被按下，于是便产生了ACTION_DOWN。如

果上次的一个触控点没有出现在本次触控点列表中，则表示此触控点刚刚被抬起，于是便有了ACTION_UP。当一个触控点同时出现在前后两次列表中时，便有了ACTION_MOVE。

在sync () 函数的最后，将本次的触控点信息mCurrentRawPointerData保存到mLastRawPointerData中，以便下次生成高级触摸事件时进行比对。

至此，TouchInputMapper完成了对触摸事件的信息整合、变换以及高级触摸事件的生成工作。

(6) Touch类型事件的处理小结

Touch类型事件的加工处理过程较Keyboard类型事件来说要复杂很多。限于篇幅，本节没能对坐标系变换、高级触摸事件的生成的细节进行探讨。不过在理解这些过程的目的与原理之后，相信读者进行更细腻的学习已不会太困难。

Touch类型事件的一大特点就是信息量大，需要用多个原始输入事件对其进行描述。本节介绍了使用多个原始输入事件描述一个输入动作的方式，以及Android收集这些事件所携带的信息并进行整合的方法。以下是读者需要注意的一些重点知识：

- 设备节点上报复杂输入事件的方式是使每个原始输入事件描述输入动作的一项信息，并在最后使用一个EV_SYN+SYN_REPORT事件表示输入动作的信息上报完毕。
- 在多点触控操作时，通过原始输入事件识别触控点索引的方式有显式与隐式两种。显式方式是由输入设备通过一条EV_ABS+ABS_MT_SLOT事件指明随后的原始输入事件所属触控点索引。而隐式方式则是通过EV_SYN+SYN_MT_REPORT事件将属于不同触控点的事件组隔开，并约定事件组的顺序就是触控点的索引。
- MultiTouchMotionAccumulator收集事件信息的方式。它根据触控点的索引从mSlots数组中获取触控点对应的Slot对象，并将原始输入事件所携带的触控点信息保存在Slot对象相应的字段中。当EV_SYN+SYN_REPORT事件到来时，其mSlots数组中便保存了所有触控点的点击信息。
- 触摸事件的坐标系变换与高级触摸事件生成方式。
TouchInputMapper::sync() 函数将触摸点信息从传感器物理坐标系变换到屏幕坐标系。而高级触摸事件的生成则依赖于前后两次触控点列表的对比结果。

5. 从InputReader到InputDispatcher

无论是Keyboard类型事件还是Touch类型事件，它们在Reader子系统中的终点都是通过InputMapper :: getListener () 所获取的继承自 InputListenerInterface接口的对象，它是输入事件从InputReader到 InputDispatcher的通道。那么这个函数所返回的对象真身为谁呢？跟踪一下getListener () 函数的实现，可以发现它所返回的对象正是 InputReader :: loopOnce () 函数中最后一步的mQueuedListener。

这个类型为QueuedInputListener的对象在InputReader的构造函数中被创建，并封装了同样继承自InputListenerInterface接口的InputDispatcher。这看似多此一举，为什么不直接使用InputDispatcher作为事件的接受者呢？事实上，QueuedInputListener这个中间人的实现非常简单，但是其意义却非常大。

分析QueuedInputListener的实现不难发现，QueuedInputListener避免了在原始事件的加工过程中向InputDispatcher进行事件提交，而是将事件信息缓存起来。在Input-Reader :: loopOnce () 函数的末尾，也就是 InputReader处理完抽取自EventHub的所有原始输入事件之后，QueuedInputListener :: flush () 函数的调用将缓存的事件信息取出，并提交给InputDispatcher。

为什么要避免在原始事件的加工过程中实时地向InputDispatcher提交事件呢？前面曾经提过，InputDispatcher会在线程循环中不断地将派发队列中的事件派发给合适的窗口。当派发队列为空时，为了节约资源，

InputDispatcher的线程就会进入休眠状态。当InputReader把新事件注入InputDispatcher的派发队列时，就需要将休眠的InputDispatcher唤醒以进行派发工作。于是问题就出现了，倘若InputDispatcher派发一个事件的速度快于InputReader加工一个原始输入事件的速度，则一次InputReader线程循环所产生的多个事件会导致InputDispatcher的线程多次休眠与唤醒，这种开销是不必要的。QueuedInputListener的存在，使得InputReader所产生的输入事件尽可能密集地注入InputDispatcher的派发队列中，从而让InputDispatcher在进入休眠之前可以派发尽可能多的事件，从而减少其休眠与唤醒的次数。

另外，除了notifyKey () 与notifyMotion () 两个函数以外，InputListenerInterface接口还定义了另一个向InputDispatcher提交输入事件的函数notifySwitch () 。这三个接口是Input-Reader输出事件的出口，同时也体现了InputReader事件输出的三种基本类型：按键类型、手势类型以及开关类型。这三种类型的事件分别由NotifyKeyArgs、NotifyMotionArgs以及NotifySwitchArgs三个结构体描述。可以说，EventHub的RawEvent是InputReader的输入，而上述三个结构体则是InputReader的输出。

6.InputReader总结

InputReader的工作原理介绍到这里也就结束了。其实，InputReader的名字应该叫做InputTranslator更合适一些，毕竟读取操作是由EventHub完

成的，而原始输入事件的整合、变换与Android输入事件的生成才是它的主要工作。

原始输入事件结构简单，信息量少，可用性不好，而且其所携带的事件信息往往都与硬件实现有关。因此有必要将原始输入事件转换为信息更丰富、可用性更佳的高级事件，这也就是InputReader存在于输入系统的意义所在。

本节的重点内容有以下几点：

- InputReader一次线程循环中所做的工作。
- InputDevice类、Device结构体之间的关联与区别。
- InputMapper家族成员以及其分配规则。
- 按键事件的处理过程，包括扫描码与虚拟键值的区别与映射关系等。
- 触摸事件的处理过程，包括触控点的识别、点击信息的收集方式、坐标系变换以及高级触摸事件的生成等。
- 使用多个原始输入事件描述一次输入动作的方式。

由于篇幅所限，本节没有讨论其他类型的InputMapper。但是Keyboard与Touch两个InputMapper已经足以代表所有其他类型的InputMapper的

工作原理。读者可以依照分析它们的方法来详细地学习其他类型的InputMapper。

5.2.5 原始事件的读取与加工总结

本节详细探讨输入系统中的第一台水泵——Reader子系统的工作原理。Reader子系统分为读取（EventHub）和加工处理（InputReader）两个部分。经过这一节的学习，读者应该能够较为深入地理解原始输入事件的概念、输入设备的加载/卸载、原始输入事件的监听读取与加工的原理。至此，读者应该能够完成将新的输入设备集成到Android输入系统中的工作。

图5-11描述了在Reader子系统中，输入事件从设备节点进入InputDispatcher的过程中所流经的重要参与者与演变过程。

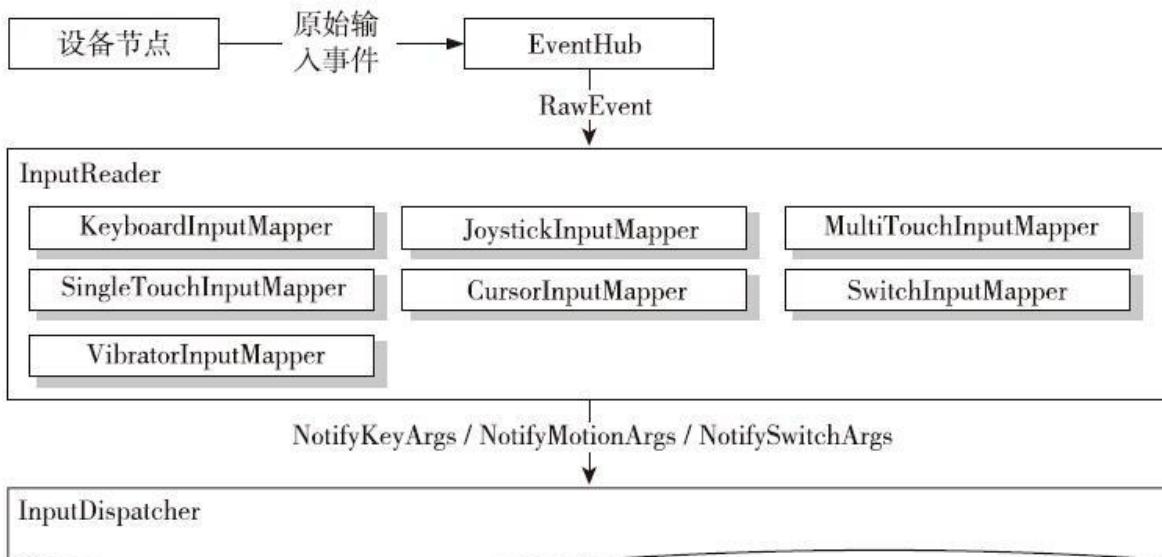


图5-11 原始事件的读取与加工过程

接下来学习输入系统的另一个核心内容——事件的派发。

5.3 输入事件的派发

本节将探讨输入系统的第二台水泵——InputDispatcher。InputDispatcher也运行在一个独立的线程（即InputDispatcherThread，以下称为派发线程）中，在其一次线程循环中会获取位于派发队列队首位置的事件，并将其派发给合适的窗口。根据上一节的介绍，当事件进入InputDispatcher时，输入事件已经被InputReader加工为三种类型：Key事件、Motion事件以及Switch事件。本节的探讨内容就是InputDispatcher的通用事件派发流程以及这三者各自独有的派发特点。另外，事件的派发过程较多地受到DispatcherPolicy的影响。

在这三种事件中，Switch事件最为简单。它根本就没有派发的过程，而是直接将Switch事件交给了DispatcherPolicy进行处理。由于它无法体现输入事件的派发流程，因此本节不对其进行讨论。

剩下的两种事件中，Motion事件的派发过程较为简洁，因此本节将以Motion事件为例阐述通用的事件派发流程。然后再以此为基础讨论Key事件特有的派发特点，以及DispatcherPolicy对派发过程的影响。

5.3.1 通用事件派发流程

1. 将事件注入派发队列

如5.2节所述，InputDispatcher实现了InputListenerInterface，并在InputReader线程循环的最后，QueuedInputListener调用此接口将InputReader产生的事件以NotifyXXXArgs结构体的形式提交给InputDispatcher。

那么，InputDispatcher如何安置这些事件呢？我们以Motion事件为例，参考以下代码：

```
[InputDispatcher.cpp-->InputDispatcher::notifyMotionLocked()] void
InputDispatcher::notifyMotion(const NotifyMotionArgs* args) { // 首先验
证参数的有效性。对Motion事件来说，主要是为了验证触控点的数量
与Id是否在合理范围 if (!validateMotionEvent(args->action, args-
>pointerCount, args->pointerProperties)) { return; } // ① 在输入事件进入
派发队列之前，首先向DispatcherPolicy获取本事件的派发策略 uint32_t
policyFlags = args->policyFlags; policyFlags |=
POLICY_FLAG_TRUSTED; // 从InputReader过来的事件都是TRUSTED
的 /* 注意，policyFlag是一个按引用传递的参数，也就是说
DispatcherPolicy会更改policyFlags的 取值 */ mPolicy-
>interceptMotionBeforeQueueing(args->eventTime, /*byref*/ policyFlags);
{ // ② 在进入派发队列之前还有一件重要的事情，就是把事件交给
InputFilter进行过滤 if (shouldSendMotionToInputFilterLocked(args)) {
MotionEvent event; // 使用NotifyMotionArgs参数中所保存的事件信息填
```

```
充一个MotionEvent对象 event.initialize(args->deviceId, .....); policyFlags  
|= POLICY_FLAG_FILTERED; // 通过policyFlags标记此事件已被过滤 //  
由DispatcherPolicy启动过滤动作。注意，当过滤结果为false时，此事件  
将被忽略 if (!mPolicy->filterInputEvent(&event, policyFlags)) { return; } }  
/* ③ 使用NotifyMotionArgs参数中的事件信息构造一个MotionEntry，并  
通过 enqueueInboundEventLocked()函数将其放入派发队列中 */  
  
MotionEntry* newEntry = new MotionEntry( args->eventTime, ...,  
policyFlags, ...); // 注意返回值needWake，它指示派发线程是否处于休眠  
状态 needWake = enqueueInboundEventLocked(newEntry); } // ④ 唤醒派  
发线程 if (needWake) { mLooper->wake(); } }
```

其中第①步和第②步为IMS或其他模块提供了修改事件派发策略，甚至是使得此事件被忽略。随后再详细讨论这两个内容。

再看一下enqueueInboundEventLocked () 函数的实现：

```
[InputDispatcher.cpp-->InputDispatcher::enqueueInboundEventLocked()]  
bool InputDispatcher::enqueueInboundEventLocked(EventEntry* entry) { /*  
如果mInboundQueue为空，则表示此时派发线程处于休眠状态，因此将  
事件注入mInboundQueue 后需要将其唤醒 */ bool needWake =  
mInboundQueue.isEmpty(); // 将事件注入mInboundQueue的队尾  
mInboundQueue.enqueueAtTail(entry); /* 为HOME键或其他窗口上的点  
击事件提高响应速度的优化操作，这些所谓的优化操作可能会导致派
```

发队列之前的所有事件被丢弃。详情将在本节后面的第4小节关于事件被丢弃的原因中探讨 */ return needWake; }

通过这两个函数可以看出，到达InputDispatcher的Motion事件被保存在MotionEntry类中，然后排在mInboundQueue列表的队尾，这个mInboundQueue就是InputDispatcher的派发队列。MotionEntry是EventEntry的一个子类，保存了Motion事件的信息。Key事件也有一个KeyEntry与之对应。EventEntry是输入事件在InputDispatcher中的存在形式。

另外，由于InputDispatcher在没有事件可以派发时（mInboundQueue为空），将会进入休眠状态，因此在将事件放入派发队列时，需要将派发线程唤醒。



注意

notifyMotion () 由InputReader在其线程循环中调用，因此在此函数执行的两个策略相关的动作interceptMotionBeforeQueueing () 和InputFilter都发生在InputReader线程，而不是派发线程中。

现在一个事件已被注入派发队列中，接下来看一下派发线程的工作流程。

2. 派发线程的线程循环

InputDispatcher工作在InputDispatcherThread（派发线程）中，与InputReader一次线程循环可以读取并生成多个输入事件不同，InputDispatcher至少需要一次线程循环才能完成一个输入事件的派发动作。其线程循环由InputDispatcher::dispatchOnce()函数完成。看一下其实现：

```
[InputDispatcher.cpp-->InputDispatcher::dispatchOnce()]
void InputDispatcher::dispatchOnce() {
    nsecs_t nextWakeupTime = LONG_LONG_MAX;
    /* ① 通过dispatchOnceInnerLocked()进行输入事件的派发。其中的传出参数nextWakeupTime决定了下次派发线程循环的执行时间 */
    if (!haveCommandsLocked()) {
        dispatchOnceInnerLocked(&nextWakeupTime);
    } // ② 执行命令队列中的命令
    if (runCommandsLockedInterruptible()) {
        nextWakeupTime = LONG_LONG_MIN;
        /* 设置nextWakeupTime为LONG_LONG_MIN将使派发线程立刻开始下次线程循环 */
    }
} // ③ 如果有必要，将派发线程进入休眠状态，并由nextWakeupTime确定休眠的具体时间
nsecs_t currentTime = now();
int timeoutMillis =
```

```
toMillisecondTimeoutDelay(currentTime, nextWakeupTime); mLooper->pollOnce(timeoutMillis); }
```

派发线程的一次循环包括如下三项工作：

- 进行一次事件派发。事件的派发工作仅当命令队列中没有命令时才会进行。派发工作会设置nextWakeupTime指明随后休眠时间长短。
- 执行命令列表中的命令。所谓的命令，不过是一个符合Command签名的回调函数，可以通过InputDispatcher::postCommandLocked()函数将其追加到命令列表中。读者可以参照Handler的工作方式来理解InputDispatcher执行命令的过程。本节不会讨论这个内容。
- 陷入休眠状态。Looper的pollOnce()的实质就是epoll_wait()，不了解Epoll机制的读者可以参考5.2.1节的介绍。因此派发线程的休眠在三种情况下可能被唤醒：调用Looper::wake()函数主动唤醒（有输入事件注入派发队列中时），到达nextWakeupTime的事件点时唤醒，以及epoll_wait()监听的fd有epoll_event发生时唤醒（这种唤醒方式将在介绍ANR机制时讨论）。

可见派发线程的线程循环是比较清晰的，不过读者可能对nextWakeupTime的存在意义有一些费解。它对派发线程的执行过程有着举足轻重的作用。例如，当派发队列中最后一个事件的派发完成后，nextWakeupTime将被设置为LONG_LONG_MAX，使之在新的输

入事件或命令到来前休眠以节约资源。另外，有时因为窗口尚未准备好接受事件（如已经有一个事件发送给窗口，但此窗口尚未对其进行反馈），则可以放弃此事件的派发并设置nextWakeupTime为一个合理的时间点，以便在下次循环时再尝试派发。

3. 派发工作的整体流程

dispatchOnceInnerLocked () 函数体现了派发过程的整体流程。接下来以Motion事件为例，探讨一下dispatchOnceInnerLocked () 函数的工作原理，并总结事件派发工作的特点。

```
[InputDispatcher.cpp--> InputDispatcher::dispatchOnceInnerLocked()]
void InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {
    /* 如果InputDispatcher被冻结，则不做任何派发操作。
       setInputDispatchMode()函数可以使得 InputDispatcher在禁用、冻结与正常三种状态下切换。详细信息将在本节的第4小节中讨论 */
    if (mDispatchFrozen) { return; } // ① 从派发队列中取出一事件进行派发
    if (! mPendingEvent) { if (mInboundQueue.isEmpty()) { /* 如果派发队列为
        空，则直接返回。此时nextWakeupTime将保持LONG_LONG_MAX，
        因此派发队列将进入无限期休眠状态 */ if (!mPendingEvent) { return; }
        else { /* 从派发队列中将位于队首的一条EventEntry取出并保存在
            mPendingEvent成员变量中。 mPendingEvent表示处于派发过程中的一
            个输入事件。之所以使用一个成员变量而不是局部变量保存它，是由
    
```

于此次线程循环有可能不能完成此事件的派发*/ mPendingEvent = mInboundQueue.dequeueAtHead(); } // 为此事件重置ANR信息。关于 ANR的内容将在5.5.1节探讨 resetANRTimeoutsLocked(); } bool done = false; // ② 检查事件是否需要被丢弃 // dropReason描述了事件是否需要被丢弃。在后面的代码中可以看到各种导致事件被丢弃的原因

```
DropReason dropReason = DROP_REASON_NOT_DROPPED; if (!  
(mPendingEvent->policyFlags & POLICY_FLAG_PASS_TO_USER)) { /*  
在将事件注入派发队列时曾向DispatcherPolicy询问过派发策略。倘若  
派发策略不允许 此事件被派发给用户，则设置对应的dropReason */  
dropReason = DROP_REASON_POLICY; } else if (!mDispatchEnabled) {  
/* 如果InputDispatcher被禁用(通过  
InputDispatcher::setInputDispatchMode())， 则此事件也会被丢弃。注意  
类似的还有一种InputDispatcher被冻结的情况。注意两者区别是  
InputDispatcher被冻结时不会将事件丢弃，而是等解冻后继续派发 */  
dropReason = DROP_REASON_DISABLED; } // 根据不同的事件类型采  
取不同的派发流程 switch (mPendingEvent->type) { ..... case  
EventEntry::TYPE_MOTION: { MotionEntry* typedEntry = static_cast  
(mPendingEvent); // 事件因为HOME键没有能被及时响应而丢弃 if  
(dropReason == DROP_REASON_NOT_DROPPED && isAppSwitchDue)  
{ dropReason = DROP_REASON_APP_SWITCH; } // 事件因为过期而被  
丢弃 if (dropReason == DROP_REASON_NOT_DROPPED &&
```

```
isStaleEventLocked(currentTime, typedEntry)) { dropReason =
DROP_REASON_STALE; } // 事件因为阻碍了其他窗口获得事件而被丢
弃 if (dropReason == DROP_REASON_NOT_DROPPED &&
mNextUnblockedEvent) { dropReason = DROP_REASON_BLOCKED; }

/* ③ 执行dispatchMotionLocked()进行Motion事件的派发。如果派发完
成，无论是成功派发 还是事件被丢弃，都返回true，否则返回false，以
便在下次循环时再次尝试此事件的派发 */
done =
dispatchMotionLocked(currentTime, typedEntry, &dropReason,
nextWakeupTime); break; } } // ④ 如果事件派发完成，则准备派发下一
个事件 if (done) { // 如果事件因为某种原因被丢弃，为了保证窗口收到
的事件仍能保持DOWN/UP，ENTER/EXIT的 配对，还需要对事件进行
补发*/ if (dropReason != DROP_REASON_NOT_DROPPED) {
dropInboundEventLocked(mPendingEvent, dropReason); } // 设置
mPendingEvent对象为NULL，使之在下次循环时可以处理派发队列中
的下一条事件 releasePendingEventLocked(); /* 立刻开始下一次循环。如
果此时派发队列为空，下次循环调用此函数时会保持nextWakeup- Time
为LONG_LONG_MAX并直接返回，使得派发线程进入无限期休眠 */
*nextWakeupTime = LONG_LONG_MIN; } }
```

dispatchOnceInnerLocked () 函数的实现解释了以下几个问题：

- 如果派发队列为空，则会使派发线程陷入无限期休眠状态。

- 即将被派发的事件从派发队列中取出并保存在mPendingEvent成员变量中。
- 事件有可能因为某些原因而被丢弃，被丢弃的原因保存在dropReason中。
- 不同类型的事件使用不同的派发函数进行实际的派发动作。如本例中的Motion事件使用dispatchMotionLocked（）函数进行派发。
- 派发一个事件至少需要一次线程循环才能完成。是否在下次循环继续尝试此事件的派发由派发函数的返回值决定。
- 事件的派发是串行的，在排在队首的事件完成派发或被丢弃之前，不会对后续的事件进行派发。理解InputDispatcher的这一特点非常重要。



注意

派发一个事件至少需要一次线程循环才能完成的原因是事件的目标窗口有可能正在处理先前的一个输入事件，在窗口完成先前事件的处理并给予反馈之前，InputDispatcher不会再向此窗口派发新事件。

另外，读者可能对此函数最后设置nextWakeupTime为LONG_LONG_MIN，使派发线程立刻进行下一次循环有一些疑问。如果此时派发队列为空，为什么不设置其为LONG_LONG_MAX使其进入无限期休眠状态呢？在这里先给读者提个醒，当派发队列为空时，派发线程可能需要在下次循环中生成重复按键事件，因此不能直接进入休眠。5.3.2节将讨论这个话题。

在继续讨论Motion事件的派发之前，先来看一下导致事件被丢弃的原因。

4.事件被丢弃的原因

前面的代码介绍了因为DispatcherPolicy认为事件不应派发给用户以及InputDispatcher被停用而导致事件被丢弃。DropReason枚举完整地描述了事件被丢弃的所有原因。

- DROP_REASON_POLICY：某些输入事件具有系统级的功能，例如HOME键、电源键、电话接听/挂断键等被系统处理，因此DispatcherPolicy不希望这些事件被窗口所捕获。当InputDispatcher在将输入事件放入派发队列前向DispatcherPolicy询问此事件的派发策略时，DispatcherPolicy会将POLICY_FLAG_PASS_TO_USER策略去掉。没有这个派发策略的对象会被丢弃。

·DROP_REASON_APP_SWITCH：dispatchOnceInnerLocked（）函数说明了InputDispatcher的事件派发是串行的。因此在前一个事件的派发成功并得到目标窗口的反馈前，后续的输入事件都会被其阻塞。当某个窗口因程序缺陷而无法响应输入时，懊恼的用户可能会尝试使用HOME键退出这个程序。然而遗憾的是，由于派发的串行性，用户所按的HOME键在其之前的输入事件成功派发给无响应的窗口之前无法获得派发的机会，因此在ANR对话框弹出之前的5秒里，用户不得不面对无响应的应用程序欲哭无泪。为了解决这个问题，InputDispatcher为HOME键设置了限时派发的要求。当InputDispatcher的enqueueInboundEventLocked（）函数发现HOME键被加入派发队列后，便要求HOME键之前的所有输入事件在0.5秒（由APP_SWITCH_TIMEOUT常量定义）之前派发完毕，否则这些事件将都会被丢弃，使得HOME键至少能够在0.5秒内得到响应。

·DROP_REASON_BLOCKED：和APP_SWITCH原因类似，如果是因为一个窗口无法响应输入事件，用户可能希望在其他窗口上进行点击，以尝试是否能得到响应。因为派发的串行性，这次尝试会以失败而告终。为此，当enqueueInboundEventLocked（）发现有窗口正阻塞着派发的进行，并且新入队的触摸事件的目标是另外一个窗口，则将这个新事件保存到mNextUnblockedEvent中。随后的dispatchOnceInnerLocked（）会将此事件之前的输入事件全部丢弃，使得用户在其他窗口上进行点击的尝试可以立刻得到响应。

·DROP_REASON_DISABLED：因为InputDispatcher被禁用而使得事件被丢弃。InputDispatcher::setInputDispatchMode () 函数可以使得InputDispatcher在禁用、冻结与正常三种状态之间进行切换。禁用状态会使得所有事件被丢弃，冻结将会使得dispatchOnceInnerLocked () 函数直接返回从而停止派发工作。InputDispatcher的这三种状态的切换由Java层的IMS提供接口，由AMS和WMS根据需要进行设置。例如，当手机进入休眠状态时，InputDispatcher会被禁用，而屏幕旋转过程中，InputDispatcher会被暂时冻结。

·DROP_REASON_STALE：在dispatchOnceInnerLocked () 函数准备对事件进行派发时，会先检查一下事件所携带的时间戳与当前时间的差距。如果事件过于陈旧（10秒以上，由常量STALE_EVENT_TIMEOUT 所指定），则此事件需要被抛弃。

当事件幸运地避开了所有上述原因之后，才能由InputDispatcher尝试派发。对Motion事件来说，下一步是dispatchMotionLocked () 函数。在这个函数中，InputDispatcher将为事件寻找合适的目标窗口。

5.Motion事件目标窗口的确定

接下来分析dispatchMotionLocked () 函数。这个函数专门用来为Motion事件寻找合适的目标窗口。参考其代码：

[InputDispatcher.cpp-->InputDispatcher::dispatchMotionLocked()] bool InputDispatcher::dispatchMotionLocked(nsecs_t currentTime, MotionEntry* entry, DropReason* dropReason, nsecs_t* nextWakeupTime) { /* 标记事件已正式进入派发流程 */ if (!entry->dispatchInProgress) { entry->dispatchInProgress = true; } // ① 对于那些不幸被丢弃的事件，直接返回 if (*dropReason != DROP_REASON_NOT_DROPPED) { return true; } // inputTargets 列表保存了此事件的发送目标 Vector inputTargets; /* ② 根据Motion事件的类型，寻找合适的目标窗口。其返回值 injectionResult指明寻找结果， 而找到的合适的目标窗口信息将被保存在inputTargets列表中 */ if (isPointerEvent) { /* 对于基于坐标点形式的事件，如触摸屏点击等，将根据坐标点、窗口ZOrder与区域寻找目标窗口 */ injectionResult = findTouchedWindowTargetsLocked(currentTime, entry, inputTargets, nextWakeupTime, &conflictingPointerActions); } else { // 对于其他类型的Motion事件（例如轨迹球），将以拥有焦点的窗口作为目标 injectionResult = findFocusedWindowTargetsLocked(currentTime, entry, inputTargets, nextWakeupTime); } /* 返回值PENDING表明找到了一个窗口，不过如果窗口处于无响应状态，则返回false，也就是说这个事件尚未派发完成，将在下次派发线程的循环中再次尝试派发 */ if (injectionResult == INPUT_EVENT_INJECTION_PENDING) { return false; } /* 如果返回值不为SUCCEEDED，表明无法为此事件找到合适的窗口，例如没有窗口处于焦点状态，或点击的位置没能落在任何一个窗口上 */ }

个窗口内，这个事件将被直接丢弃 */ if (injectionResult != INPUT_EVENT_INJECTION_SUCCEEDED) { return true; } /* 接下来向 inputTargets列表中添加特殊的接收目标。可以看出这些名为Monitoring targets的接收者可以监听所有的输入事件 */ if (isMainDisplay(entry->displayId)) { addMonitoringTargetsLocked(inputTargets); } // ③ 调用 dispatchEventLocked() 将事件派发给inputTargets列表中的目标 dispatchEventLocked(currentTime, entry, inputTargets); return true; }

这个函数主要包括以下三项工作：

- 对于那些被丢弃的事件，直接返回true。
- 为事件寻找合适的目标窗口。目标窗口分为两种：普通窗口以及监听窗口（monitoring）。普通窗口通过按点查找与按焦点查找两种方式获得，而监听窗口则无条件监听所有输入事件。在5.4.2节介绍 InputChannel 的注册时将会对监听窗口进行探讨。普通窗口的查找结果决定了此次线程循环是否可以完成事件派发。
- 如果成功地找到了可以接收事件的目标窗口，则通过 dispatchEventLocked () 函数完成实际的派发工作。

可以看到，查找到的派发目标使用InputTarget结构体存储。其中重要的字段定义如下：

[InputDispatcher.h-->InputTarget] struct InputTarget { /* 在第4章中的 SampleWindow例子中见过这个InputChannel，它是连接InputDispatcher 与窗口 的通信管道。InputDispatcher通过它将事件派发给目标窗口，并 通过它读取来自目标窗口对事件 的响应 */ sp inputChannel; /* 事件的派 发选项。这个选项将使得事件派发时可以自动产生一些辅助的输入事 件。向此目标窗口发送 的事件内容有可能因这个flag的设置而发生改 变。在5.4.3节将会对其影响做出介绍 */ int32_t flags; /* 目标窗口相对于 屏幕坐标系的偏移与缩放参数。InputReader将输入事件从传感器物理 坐标系转换为 屏幕坐标系，而这几个参数则可以将输入事件从屏幕坐 标系转换到窗口自身的坐标系 */ float xOffset, yOffset; float scaleFactor; // pointerIds字段描述在这个窗口中有哪些触控点被按下 BitSet32 pointerIds; }

InputTarget的这些字段并不难以理解，接下来分别看一下采用按点查找 与按焦点查找两种方式如何获取合适的InputTarget对象。

(1) 根据坐标点查找目标窗口

dispatchMotionLocked () 使用findTouchedWindowTargetsLocked () 函 数根据事件的坐标点获取合适的目标窗口。由于多点触控下会出现某 触控点移出其按下窗口的情况，此时需要将多点触控事件分割为多条 单点事件。这个处理也位于此函数中，使得其逻辑十分复杂，为了能

够清晰地分析根据坐标点查找窗口的原理，下面的代码仅保留了关键的逻辑：

```
[InputDispatcher.cpp-->InputDispatcher::  
findTouchedWindowTargetsLocked()] int32_t  
InputDispatcher::findTouchedWindowTargetsLocked(nsecs_t currentTime,  
const MotionEntry* entry, Vector & inputTargets, nsecs_t*  
nextWakeupTime, bool* outConflictingPointerActions) { // 从MotionEntry  
中获取事件的坐标点 int32_t x = .....; int32_t y = .....; /* InputDispatcher  
保存了一个名为mWindowHandles的列表。这个列表中所保存的  
InputWindow- Handler类保存了窗口的InputChannel以及  
InputWindowInfo结构体。 InputWindowInfo结构体则保存了窗口的各种  
布局信息，包括可见性、位置和尺寸、flag等。可以说，  
InputWindowHandle是WMS中的WindowState在输入系统中的化身。而  
mWindowHandles 列表则按照Z轴顺序保存了当前WMS中的所有窗口。  
很明显，这个列表是由WMS更新进InputDispatcher 的。这个更新过程  
在5.4.2节探讨，读者此时只要知道它们与WMS窗口之间的联系即可 */  
size_t numWindows = mWindowHandles.size(); // ① 遍历  
mWindowHandles列表中所有的WindowHandle， 检查事件坐标点是否落  
在其上 for (size_t i = 0; i < numWindows; i++) { sp<windowHandle> =  
mWindowHandles.itemAt(i); // 获取保存窗口布局信息的windowInfo  
const InputWindowInfo* windowInfo = windowHandle->getInfo(); // 获取
```

窗口的flag int32_t flags = windowInfo->layoutParamsFlags; if
(windowInfo->visible) { if (! (flags &
InputWindowInfo::FLAG_NOT_TOUCHABLE)) { // 检查窗口是否指明
了FLAG_NOT_TOUCH_MODAL选项 isTouchModal = (flags &
(InputWindowInfo::FLAG_NOT_FOCUSABLE |
InputWindowInfo::FLAG_NOT_TOUCH_MODAL)) == 0; /* 以下是选择
目标窗口的两个关键条件：坐标点落在窗口之上，或者此窗口没有指
定FLAG_NOT_TOUCH_MODAL选项 */ if (isTouchModal || windowInfo-
>touchableRegionContainsPoint(x, y)) { newTouchedWindowHandle =
windowHandle; break; } } } // 把选中的窗口保存到TempTouchState中，
以便后续处理

TempTouchState.addOrUpdateWindow(newTouchedWindowHandle,
targetFlags, pointerIds); // ② 检查TempTouchState中所有目标窗口是否已
准备好接收新的输入事件 for (size_t i = 0; i <
mTempTouchState.windows.size(); i++) { const TouchedWindow&
touchedWindow = mTempTouchState.windows[i]; // 确定窗口是否可以接
收新的事件 if (!isWindowReadyForMoreInputLocked(currentTime,
touchedWindow.windowHandle, entry)) { /* 如果窗口不能接收新事件，
则记录不能接收事件的原因，并设置nextWakeUpTime为5s后，如果5s
后线程仍未能将此事件派发成功而进入这个分支，则会开始向Java层通
报ANR。 在这里，injectionResult被设置为

```
INPUT_EVENT_INJECTION_PENDING */ injectionResult =
handleTargetsNotReadyLocked(currentTime, entry, NULL,
touchedWindow.windowHandle, nextWakeuptime, .....); // 因为此次查找
到的窗口不能接收事件，所以跳过后续产生InputTarget的过程 goto
Unresponsive; } } /* ③ 如果执行到这里，说明窗口的查找过程一切顺
利。设置injectionResult为SUCCEEDED，并将生成的InputTarget放入参
数inputTargets中 */
injectionResult =
INPUT_EVENT_INJECTION_SUCCEEDED; for (size_t i = 0; i <
mTempTouchState.windows.size(); i++) { const TouchedWindow&
touchedWindow = mTempTouchState.windows.itemAt(i); // 为每一个
mTempTouchState中的窗口生成InputTargets
addWindowTargetLocked(touchedWindow.windowHandle,
touchedWindow.targetFlags, touchedWindow.pointerIds, inputTargets); }
Unresponsive: return injectionResult; }
```

这个函数中有一个关键点，那就是mWindowHandles列表。如代码注释所述，它是WMS所有窗口在输入系统的存在形式，它所保存的布局信息为查找输入窗口提供了依据。因此，每当窗口布局发生变化时，WMS都会通过IMS将所有窗口的信息提交到InputDispatcher，并完成mWindowHandles每一个元素的更新工作。注意mWindowHandler中的窗口顺序是索引越小，ZOrder越大，这与WMS中的窗口列表顺序相反。

总体来说，此函数包括三项主要工作：

- 根据窗口的点击区域与事件发生的坐标点选取合适的目标窗口。注意其遍历顺序是沿ZOrder由上至下进行遍历，因此ZOrder越靠上，则拥有获取事件的优先权。另外，如果窗口没有在其LayoutParams.flag中指明FLAG_NOT_TOUCH_MODAL选项，说明是一个模式窗口。模式窗口将会阻止点击事件被派发给位于其下的窗口，无论点击事件是否发生在它的可点击区域内。
- 检查所找到的窗口是否可以接收新的按键事件。这个检查工作是由isWindowReadyForMoreInputLocked () 函数完成的。如果窗口尚无法接收事件，则说明此窗口有可能发生ANR。
handleTargetsNotReadyLocked () 会记录下这一事实，并将injectionResult设置为PENDING，要求下次派发线程的循环中重试此事件的派发。因为检查窗口是否可以接收新的输入事件需要我们清楚地理解向窗口派发事件以及窗口对事件做出反馈的过程，因此这里暂不做深入讨论。
- 如果找到的窗口可以接收新的事件，则由addWindowTargetLocked () 生成一个InputTarget，并放入参数inputTargets列表中。由于InputTarget中几乎所有字段都可以从InputWindowHandle中找到，所以其生成过程就不再赘述。

(2) 根据焦点查找目标窗口

根据焦点查找目标窗口是由findFocusedWindowTargetsLocked () 函数完成的。这个函数的主要工作与findTouchedWindowTargetsLocked () 函数一样，而且其窗口查找过程毫不费力。因为InputDispatcher有一个名为mFocusedWindowHandle的InputWindowHandle对象，所以InputDispatcher只要判断此对象是否为NULL即可。

WMS在进行布局操作时，会根据处于焦点状态的Activity、窗口的属性与ZOrder确定处于焦点状态的窗口，并随同窗口列表一起提交至InputDispatcher。焦点窗口的确定将在5.5.2节详细讨论。

6.向窗口发送事件

合适的目标窗口被确定下来之后，便可以开始将实际的事件发送给窗口了。dispatchMotionLocked () 使用dispatchEventLocked () 函数将事件发送给指定的InputTarget。从函数的名称可以知道，我们又回到了通用的派发流程中。参考其代码：

```
[InputDispatcher.cpp-->InputDispatcher:: dispatchEventLocked()] void  
InputDispatcher::dispatchEventLocked(nsecs_t currentTime, EventEntry*  
eventEntry, const Vector & inputTargets) { // 遍历每一个InputTarget for  
(size_t i = 0; i < inputTargets.size(); i++) { const InputTarget& inputTarget =  
inputTargets.itemAt(i); // 根据InputTarget中的InputChannel，获取对应
```

```
Connection对象的索引 ssize_t connectionIndex =  
getConnectionIndexLocked(inputTarget.inputChannel); if (connectionIndex  
>= 0) { sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex); // 调  
用prepareDispatchCycleLocked(), 针对当前InputTarget启动事件发送循  
环 prepareDispatchCycleLocked(currentTime, connection,  
eventEntry,&inputTarget); } else { ..... // 没有对应的Connection, 事件被  
丢弃 } } }
```

代码很短，但疑问却多。Connection类是什么呢？事件发送循环又是什么呢？在通过WMS角度分析InputChannel注册以及窗口信息向InputDispatcher的提交过程之前，可能无法讨论清楚它们的细节。因此，本节暂以此函数作为派发工作的终点，目前读者仅需知道dispatchEventLocked () 函数可以将输入事件发送给特定的窗口即可。在5.4节关于输入事件的接收与反馈的讨论中，再来深刻理解InputChannel类、Connection类以及事件发送循环的原理。

7.通用事件派发流程总结

本节完成了输入事件从注入派发队列开始，到通过InputTargets将事件发送给指定窗口的过程。图5-12描述了通用的事件派发流程。另外，图中没能表现出来的几点需要注意：

- 在事件进入派发队列之前的处理位于InputReader线程中。而其他操作则位于派发线程中。
- 事件以EventEntry子类的形式存在于InputDispatcher中。
- 事件派发是串行的，在队首的事件派发完成之前，不会进行其他事件的派发。
- 在选择InputTarget的过程中，如果发现有一个目标窗口尚未准备好接收事件，则暂停当前事件的派发，并通过设置nextWakeupTime在下次派发循环时再试派发。

接下来探讨Key事件派发过程与Motion事件派发的不同点。

5.3.2 按键事件的派发

按键事件的派发流程基本遵循上节以Motion事件为基础所讨论的通用事件派发流程。不过，由于按键往往携带系统功能，例如HOME键、POWER键以及Volume键等，InputDispatcher需要对按键事件做更多附加工作。本节就按键事件相对于Motion事件派发流程的不同点进行探讨。

1. 将事件注入派发队列

将按键事件注入派发队列的函数是notifyKey () 函数。参考其实现：

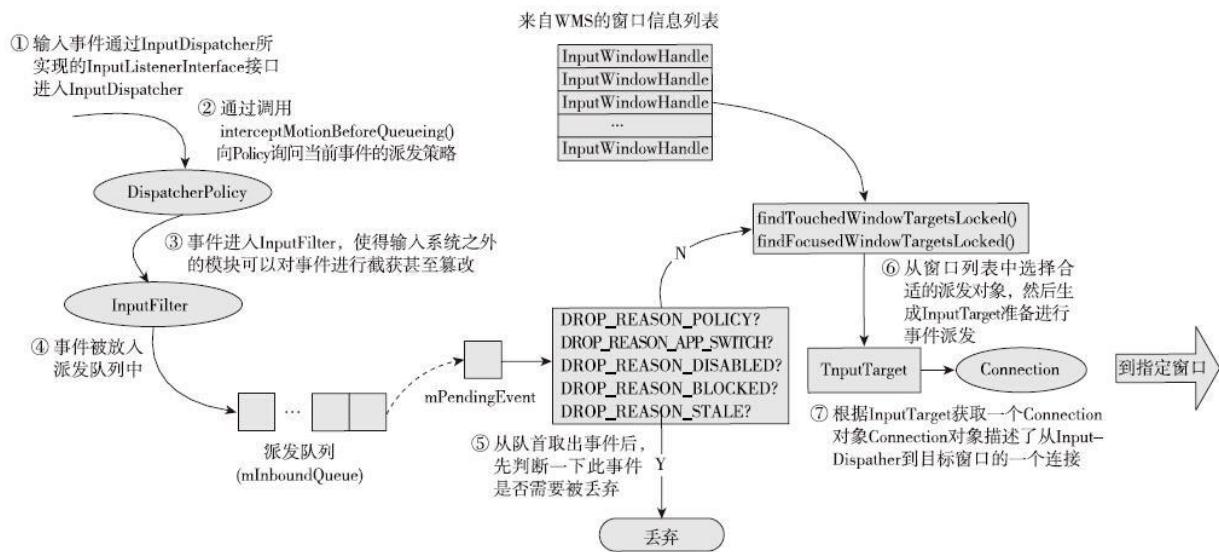


图5-12 通用的事件派发流程

```
[InputDispatcher.cpp-->InputDispatcher::notifyKey()] void
InputDispatcher::notifyKey(const NotifyKeyArgs* args) { // 验证事件信息
    的合法性。其实就是验证Action是ACTION_DOWN/UP其中之一 if
    (!validateKeyEvent(args->action)) { return; } uint32_t policyFlags = args-
    >policyFlags; int32_t flags = args->flags; int32_t metaState = args-
    >metaState; // ① 根据policyFlag修改此事件的meataState if (policyFlags &
    POLICY_FLAG_ALT) { metaState |= AMETA_ALT_ON |
    AMETA_ALT_LEFT_ON; } ..... if (policyFlags &
    POLICY_FLAG_FUNCTION) { metaState |= AMETA_FUNCTION_ON; }
    policyFlags |= POLICY_FLAG_TRUSTED; /* 同notifyMotion一样，凡是
    从InputReader过来的事件都输入TRUSTED */ /* ② 接下来向
    DispatcherPolicy询问按键事件的派发策略。与notifyMotion中的
```

intercept- MotionBeforeQueueing()不同，询问按键派发策略的函数需要 KeyEvent作为参数 */ KeyEvent event; event.initialize(args->deviceId,
args->source, args->action, flags, args->keyCode, args->scanCode,
metaState, 0, args->downTime, args->eventTime); mPolicy-
>interceptKeyBeforeQueueing(&event, /*byref*/ policyFlags); /*
DispatcherPolicy可能认为此按键的到来将会唤醒设备。用
AKEY_EVENT_FLAG_WOKE_HERE标记这个特殊的按键 */ if
(policyFlags & POLICY_FLAG_WOKE_HERE) { flags |=
AKEY_EVENT_FLAG_WOKE_HERE; } { // 将事件交给InputFilter
进行过滤 int32_t repeatCount = 0; // ③ 构建一个KeyEntry，并像
MotionEntry一样注入到派发队列 KeyEntry* newEntry = new
KeyEntry(args->eventTime, args->deviceId, args->source, policyFlags, args-
>action, flags, args->keyCode, args->scanCode, metaState, repeatCount,
args->downTime); needWake = enqueueInboundEventLocked(newEntry); }
// 唤醒派发线程 if (needWake) { mLooper->wake(); } }

按键事件向派发队列注入的过程与Motion事件大同小异。也经历了询问派发策略与进行InputFilter过滤的操作，不过也有自己的特点。

按键事件可以根据policyFlag修改自己的metaState。最初的policyFlag来自于键盘布局文件中描述一条键盘映射的行的第四列policy_flag信息。

通过policyFlag修改metaState的行为使得设备可以通过使用一个物理按键来模拟组合按键的效果。

另外，在询问派发策略时，notifyKey () 向DispatcherPolicy提交了一个KeyEvent对象。因为与Motion事件属于单纯的点击不同，按键事件的键值不同则其对应的功能以及其重要性大不相同，DispatcherPolicy需要根据按键的详细信息做出不同的策略选择。关于派发策略的影响将在5.4节详细讨论。

最后一个不同点是，按键事件存储在EventEntry的另一个子类KeyEntry中。

当按键事件注入派发队列以后，通过mLooper->wake () 唤醒派发线程对此事件进行派发。这点与Motion事件相同。

2.额外的派发策略查询

接下来进入派发流程中dispatchOnceInnerLocked () 函数的环节。参考以下代码：

```
[InputDispatcher.cpp-->InputDispatcher::dispatchOnceInnerLocked()] void  
InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {  
..... // 取出mPendingEvent switch (mPendingEvent->type) { case  
EventEntry::TYPE_KEY: { KeyEntry* typedEntry = static_cast
```

```
(mPendingEvent); ..... // 处理dropReason // 派发按键事件的函数是  
dispatchKeyLocked() done = dispatchKeyLocked(currentTime, typedEntry,  
&dropReason, nextWakeupTime); break; } } ..... }
```

可以看出，派发按键事件的函数是dispatchKeyLocked ()，这个函数的实现与dis-patchMotionLocked () 类似。最主要的不同点在于，按键事件在派发之前，又会向Dis-patcherPolicy询问一次后续的派发策略。注入派发队列前询问的策略是beforeQueueing，而这一次则是beforeDispatching，并且派发策略的查询结果将被放置在KeyEntry :: interceptKeyResult中。此次派发策略的查询主要是为了给DispatcherPolicy优先处理按键事件的机会，并决定是否将此按键派发给用户。本节也将这个操作称为派发策略查询，读者需要加以区别。看一下dispatchKeyLocked () 函数：

```
[InputDispatcher.cpp-->InputDispatcher::dispatchKeyLocked()] bool  
InputDispatcher::dispatchKeyLocked(nsecs_t currentTime, KeyEntry* entry,  
DropReason* dropReason, nsecs_t* nextWakeupTime) { //  
dispatchKeyLocked()函数的前半部分用来控制重复按键的生成。这个内  
容在下一节再讨论 ..... /* 如果上次的策略查询结果是  
TRY AGAIN LATER，表示上次进行策略查询时，DispatcherPolicy尚  
未决定好对此事件所使用的策略，要求InputDispatcher等一段时间后再  
做策略查询。此时要检查是否可以再进行策略查询 */ if (entry-
```

```
>interceptKeyResult == KeyEntry::INTERCEPT_KEY_RESULT_TRY AGAIN LATER) { // 派发线程被过早地唤醒，设置nextWakeupTime后重新进入休眠状态 if (currentTime < entry->interceptKeyWakeUpTime) { if (entry->interceptKeyWakeUpTime < *nextWakeupTime) { *nextWakeupTime = entry->interceptKeyWakeUpTime; } return false; } // 如果当前时间点可以进行重试，则重置派发策略相关的字段，准备重新进行策略查询 entry->interceptKeyResult = KeyEntry::INTERCEPT_KEY_RESULT_UNKNOWN; entry->interceptKeyWakeUpTime = 0; } // 如果此事件尚未进行过派发策略查询，则通过发送一个命令的方式查询派发策略 if (entry->interceptKeyResult == KeyEntry::INTERCEPT_KEY_RESULT_UNKNOWN) { if (entry->policyFlags & POLICY_FLAG_PASS_TO_USER) { // 发送一个 command，在这个command中进行策略查询 CommandEntry* commandEntry = postCommandLocked(&doInterceptKeyBeforeDispatchingLockedInterruptible); ..... // 终止此次派发，等待策略查询完成 return false; } else { entry->interceptKeyResult = KeyEntry::INTERCEPT_KEY_RESULT_CONTINUE; } } else if (entry->interceptKeyResult == KeyEntry::INTERCEPT_KEY_RESULT_SKIP) { // 如果派发策略的查询结果是SKIP，则表示InputDispatcher需要丢弃这
```

```
个事件 if (*dropReason == DROP_REASON_NOT_DROPPED) {  
    *dropReason = DROP_REASON_POLICY; } } ..... // 检查 dropReason,  
看是否需要继续进行派发 // 按键事件的目标窗口是通过按焦点的方式  
获得的 Vector inputTargets; int32_t injectionResult =  
findFocusedWindowTargetsLocked(currentTime, entry, inputTargets,  
nextWakeupTime); // 回到通用的派发流程中  
dispatchEventLocked(currentTime, entry, inputTargets); }
```

从这段代码中可以看出，在绝大多数情况下，按键事件都需要两次派发循环才能完成派发工作。按键事件第一次进入 dispatchKeyLocked () 函数时，其 interceptKeyResult 为 UNKNOWN，因此需要向 DispatcherPolicy 询问派发策略。询问过程是通过一个 Command 完成的，因此 dispatchKeyLocked () 将这个 Command 加入 InputDispatcher 的命令队列之后就直接返回。这个 Command 命令执行 doInterceptKeyBeforeDispatchingLockedInterruptible () 函数，向 DispatcherPolicy 查询派发策略，并将查询结果保存在 interceptKeyResult 中。命令完成后的下次派发循环将再次进入 dispatchKeyLocked ()，此时便可根据 interceptKeyResult 做出不同的处理。

看一下 doInterceptKeyBeforeDispatchingLockedInterruptible () 是如何工作的：

[InputDispatcher.cpp-->InputDispatcher::
doInterceptKeyBeforeDispatchingLockedInterruptible()]

```
void InputDispatcher::doInterceptKeyBeforeDispatchingLockedInterruptible(  
    CommandEntry* commandEntry) { // 根据commandEntry中的KeyEntry创  
    建一个KeyEvent对象 ..... // 通过interceptKeyBeforeDispatching()调用从  
    DispatcherPolicy中获取的派发策略 nsecs_t delay = mPolicy-  
    >interceptKeyBeforeDispatching(commandEntry->inputWindowHandle,  
    &event, entry->policyFlags); // 返回至delay的取值范围决定了  
    interceptKeyResult的三种取值 if (delay < 0) { entry->interceptKeyResult =  
        KeyEntry::INTERCEPT_KEY_RESULT_SKIP; } else if (!delay) { entry-  
        >interceptKeyResult =  
        KeyEntry::INTERCEPT_KEY_RESULT_CONTINUE; } else { entry-  
        >interceptKeyResult =  
        KeyEntry::INTERCEPT_KEY_RESULT_TRY AGAIN_LATER; entry-  
        >interceptKeyWakeUpTime = now() + delay; } }
```

为按键询问派发策略的函数是DispatcherPolicy的
interceptKeyBeforeDispatching () 。这个函数对按键事件进行处理，并
返回延迟派发的时间delay。delay的三种取值范围决定了
interceptKeyResult的三种取值：

- INTERCEPT_KEY_RESULT_SKIP，当delay小于0时，表示DispatcherPolicy自己处理了按键事件，或不希望此事件被派发给用户。
- INTERCEPT_KEY_RESULT_CONTINUE，当delay等于0时，表示DispatcherPolicy认为此事件可以正常地派发给用户。
- INTERCEPT_KEY_RESULT_TRY AGAIN_LATER，：当delay大于0时，表示DispatcherPolicy尚未决定好如何处理这个按键事件。要求InputDispatcher先暂停对此事件的派发工作，稍后再进行一次策略查询。暂停的时间由delay决定。

再回到dispatchKeyLocked () 函数中，可以看到它对这三种查询结果的处理方式。对于TRY AGAIN_LATER，设置了nextWakeupTime，使派发线程在休眠指定的延迟后唤醒重新进行策略查询。对于SKIP，则通过设置dropReason为DROP RESEAON POLICY，从而将此事件丢弃。

读者需要深入理解dispatchKeyLocked () 通过两次派发线程循环确定事件的派发策略，以及派发策略对事件的派发过程的影响。DispatcherPolicy的interceptKeyBeforeDispatching () 的具体工作将在5.3.3节中介绍。

3.重复按键事件

在进行Android应用开发过程中，在用户持续按下一个按键到抬起之间，应用程序往往能够收到多次onKeyDown () 的调用，每次onKeyDown () 调用时的KeyEvent.getRepeatCount () 的返回值会不断累加，并且当且仅当第二次调用时KeyEvent.isLongPress () 的返回值为true。InputDispatcher就是完成此项工作的场所。

虽然有些按键输入设备支持按键重复按下事件的回报工作（如KeyboardInputMapper中做过了重复按键的容错处理），但是大部分输入设备仅上报初次按下与抬起两个事件。因此，InputDispatcher必须对重复按键事件做出模拟。本节将讨论这个模拟的过程。参考dispatchOnceInnerLocked () 函数的实现。

```
[InputDispatcher.cpp-->InputDispatcher::dispatchOnceInnerLocked()] void  
InputDispatcher::dispatchOnceInnerLocked(nsecs_t* nextWakeupTime) {  
..... if (! mPendingEvent) { if (mInboundQueue.isEmpty()) { /* 如果  
mKeyRepeatState中的lastKeyEntry字段不为空，说明需要产生一个重复  
按键 事件 */ if (mKeyRepeatState.lastKeyEntry) { if (currentTime >=  
mKeyRepeatState.nextRepeatTime) { /* 如果当前时间点等于或晚于  
mKeyRepeatState所要求的产生重复按键的时间点则通过  
synthesizeKeyRepeatLocked()产生一个重复按键事件，并保存到  
mPendingEvent中，作为随后的派发对象 */ mPendingEvent =  
synthesizeKeyRepeatLocked(currentTime); } else { /* 如果当前时间点早
```

于mKeyRepeatState所要求产生重复事件的时间点，则设置nextWakeupTime使派发线程沉睡一段时间，并在指定的时间点醒来再进 重复按键的生成 */ if (mKeyRepeatState.nextRepeatTime < *nextWakeupTime) { *nextWakeupTime = mKeyRepeatState.nextRepeatTime; } } // 如果不需要生成重复按键事件，或还没有到达合适的时间点，则直接返回， 派发线程再次沉睡 if (!mPendingEvent) { return; } } else { // 从派发队列的队首取出 mPendingEvent } } switch (mPendingEvent->type) { // 选择合适的派发函数 }

dispatchOnceInnerLocked () 指示了两个按键事件的处理特点：

- InputDispatcher会对重复按键做出模拟。
- 负责派发按键事件的函数名为dispatchKeyLocked () 。

通过这个函数可以看到，InputDispatcher通过mKeyRepeatState对象控制重复按键模拟的过程。mKeyRepeatState的类型是KeyRepeatState。它保存了两个字段，KeyEntry类型的lastKeyEntry及nsecs_t类型的nextRepeatTime，分别指明需要进行重复模拟的按键事件，以及下次进行重复模拟的时间。如果不需要进行模拟，则lastKeyEntry字段为空。不难想象，InputDispatcher一定会在一个虚拟键值的DOWN事件到来时，设置lastKeyEntry字段为有效值，并设置下次模拟的时间，然后，

当此虚拟键值的UP事件到来时，将lastKeyEntry置空。为了验证这个想法，接下来看一下dispatchKeyLocked () 函数的实现。

(1) 开启与关闭重复按键模拟

```
[InputDispatcher.cpp-->InputDispatcher::dispatchKeyLocked()] bool  
InputDispatcher::dispatchKeyLocked(nsecs_t currentTime, KeyEntry* entry,  
DropReason* dropReason, nsecs_t* nextWakeupTime) { // 仅当按键事件  
第一次尝试进行派发时，才会操作重复按键的开启与关闭 if (! entry-  
>dispatchInProgress) { // ① 这个判断就是开启重复按键模拟的条件 if  
(entry->repeatCount == 0 // repeatCount为0表示非重复按键事件 &&  
entry->action == AKEY_EVENT_ACTION_DOWN // 表示一定是按下事  
件 && (entry->policyFlags & POLICY_FLAG_TRUSTED)// 事件一定来  
自InputReader // DispatcherPolicy允许对此事件进行重复模拟 && (!  
(entry->policyFlags & POLICY_FLAG_DISABLE_KEY_REPEAT))) { if  
(mKeyRepeatState.lastKeyEntry && mKeyRepeatState.lastKeyEntry-  
>keyCode == entry->keyCode) { // ② 硬件主动上报了重复按键事件，则  
关闭模拟 entry->repeatCount = mKeyRepeatState.lastKeyEntry-  
>repeatCount + 1; resetKeyRepeatLocked();  
mKeyRepeatState.nextRepeatTime = LONG_LONG_MAX; } else { // ③ 开  
启重复模拟 // 重置mKeyRepeatState对象 resetKeyRepeatLocked(); // 设置  
下次模拟重复按键是在mConfig.keyRepeatTimeout所指定的时间之后
```

```
(0.5s) mKeyRepeatState.nextRepeatTime = entry->eventTime +
mConfig.keyRepeatTimeout; } // 保存当前事件到lastKeyEntry
mKeyRepeatState.lastKeyEntry = entry; } else if (! entry->syntheticRepeat)
{ // ④ 对于不符合要求的事件，重置mkeyRepeatState，关闭重复模拟 /*
注意，由于模拟出来的重复按键事件的repeatCount不为0，因此需要根据synthetic- Repeat是否为false来避免错误地关闭重复模拟 */
resetKeyRepeatLocked(); } ..... entry->dispatchInProgress = true; } // 后续
的派发工作 ..... }
```

可以看出，开启重复模拟操作的条件有4个。为了理解这4个条件的意义，需要读者注意一点，模拟的重复按键事件除了其来源不是派发队列以外，其他方面都与真实的按键事件一样，也会进入 dispatchKeyLocked () 进行派发。

这4个条件是：

- repeatCount等于0，也就是此事件尚未进行重复模拟。
- event->action等于ACTION_DOWN，也就是仅对DOWN事件进行重复模拟。
- 事件拥有TRUSTED选项，即事件来源于InputReader。

·事件没有DISABLE_KEY_REPEAT选项，以及DispatcherPolicy允许对此事件进行重复模拟。

而关闭重复模拟的条件是：

·硬件上报了一个重复按下的事件。这说明硬件支持按下事件的重复上报，因此也就不再需要进行模拟了。

·不满足上述4个条件，并且不是模拟的重复按键。由于模拟的重复按键也会进入dispatchKeyLocked () 进行派发，而这个事件并不满足repeatCount等于0的条件，为了避免因为这个事件导致重复模拟被关闭，需要对syntheticRepeat为true的事件放行。

进行重复模拟的时间点由mConfig.keyRepeatTimeout指定，mConfig中保存了两个字段，分别为keyRepeatTimeout (500ms) 与keyRepeatDelay (50ms)。由此可以看出，实际按下按键到第一次重复模拟按键的时间间隔为500ms。

当dispatchKeyLocked () 完成了重复模拟的开启与关闭后，便继续当前事件的派发。随后派发线程循环的dispatchOnceInnerLocked () 将会尝试模拟重复事件。

(2) 重复按键的生成

回到dispatchOnceInnerLocked () 函数，可以看到仅当派发队列为空时才会考虑重复按键的生成动作，因此重复按键事件的优先级是最低的。

重复按键的生成由synthesizeKeyRepeatLocked () 函数完成。重复按键的信息将继承自被模拟的按键。看一下其实现：

```
[InputDispatcher.cpp-->InputDispatcher::synthesizeKeyRepeatLocked()]
InputDispatcher::KeyEntry* InputDispatcher::synthesizeKeyRepeatLocked(
nsecs_t currentTime) { KeyEntry* entry = mKeyRepeatState.lastKeyEntry;
/* 模拟按键事件的派发策略继承自被模拟的事件，但是被
DispatcherPolicy赋予的派发策略被清理掉了(RAW_MASK以外的派发
策略)，也就是只保留了由InputReader产生的派发策略 */ uint32_t
policyFlags = (entry->policyFlags & POLICY_FLAG_RAW_MASK) |
POLICY_FLAG_PASS_TO_USER | POLICY_FLAG_TRUSTED; if
(entry->refCount == 1) { // 如果entry可以被重用，则重用之 entry-
>recycle(); entry->eventTime = currentTime; // 时间点是当前时间 entry-
>policyFlags = policyFlags; // 被清理过的派发策略 entry->repeatCount +=
1; // repeatCount递增 } else { ..... // 否则新建一个KeyEntry并填充必要的
时间信息 } entry->syntheticRepeat = true; ..... // 设置下次模拟重复按键
的时间是mConfig.keyRepeatDelay之后(50ms)
```

```
mKeyRepeatState.nextRepeatTime = currentTime +  
mConfig.keyRepeatDelay; return entry; }
```

不难看出，被模拟的按键事件有以下特点：

- 被模拟的重复按键事件的KeyEvent是可以被重用的。由于事件在派发前会被DispatcherPolicy设置新的派发策略，因此需要将新模拟的事件的派发策略进行清理。使之好像一个新的事件一样。
- 重复按键事件的时间戳是产生事件时的时间。
- 每一次模拟事件的产生，都会使得repeatCount递增。
- 产生一个模拟事件之后，会设置下一个模拟事件产生的时间为mConfig.keyRepeatDelay所指定的时间段之后。

新的模拟事件生成之后，将作为mPendingEvent对象，进入后续的派发流程完成派发动作。

(3) 重复按键事件小结

重复按键事件由dispatchOnceInnerLocked () 函数与dispatchKeyLocked () 函数协作完成，它们分别负责重复按键事件的生成与控制。 dispatchKeyLocked () 开启重复事件生成的条件十分严格，任何不满足这4个条件的按键事件都会终止生成。生成重复事件的时间点由

mKeyRepeatState.nextRepeatTime控制。另外，模拟的重复事件的信息继承自被模拟的事件，但是对InputDispatcher来说，它们是独立的两个事件。

4.按键事件派发过程的总结

在依照Motion事件完成通用派发过程的学习之后，按键事件的派发过程的分析简单了很多。这一节主要介绍按键事件的派发相对于通用事件派发流程的不同点。在这里总结一下：

- 按键事件通过notifyKey () 函数进入InputDispatcher，并在注入派发队列前，使用DispatcherPolicy的interceptKeyBeforeQueueing () 函数询问后续的派发策略policyFlag。
- 按键事件在正式派发给窗口前，进行了一次额外的派发策略查询。这次查询的结果保存在KeyEntry :: interceptKeyResult中。查询结果决定了此事件是正常派发、稍后派发，还是丢弃。
- 当按键按下到按键抬起之间的时间里，dispatchOnceInnerLocked () 和 dispatch-KeyLocked () 函数会协同工作完成对重复按键事件的模拟。
- 按键事件的派发目标仅通过焦点方式进行查找。

5.3.3 DispatcherPolicy与InputFilter

在分析事件的派发过程中，曾多次提到DispatcherPolicy与InputFilter。这两种机制都为输入系统以外的模块提供了调整派发流程、提前处理输入事件以及修改事件内容的机会。本节将探讨它们的工作原理以及对派发过程的影响。

1. DispatcherPolicy的功能

DispatcherPolicy是一个实现了DispatcherPolicyInterface的对象，保存在InputDispatcher的mPolicy成员变量中。在分析IMS的构成时曾经介绍过，实现这个接口的对象就是位于输入系统JNI层的NativeInputManager。来自InputDispatcher的策略查询都被它转而交给Java层的IMS进行处理。本章所讨论的三个重要的策略查询函数interceptMotionBeforeQueueing ()、interceptKeyBeforeQueueing () 以及interceptKeyBeforeDispatching () 的最终实现其实并不在IMS中。对它们的调用由IMS转给WMS的InputMonitor，然后再由InputMonitor转给PhoneWindowManager。因此这三个重要的策略其实都在PhoneWindowManager的同名函数中。

interceptMotionBeforeQueueing () 的调用发生在Reader线程中，其主要作用就是返回policyFlags派发策略。PhoneWindowManager可以根据设备的状态返回如下策略：

·POLICY_FLAG_WOKE_HERE， PhoneWindowManager可能认为此次点击可以将设备唤醒。虽然此派发策略不会对派发流程产生影响，但是在从Java层的调用返回时， NativeInputManager会检查这个派发策略，并调用PowerManagerService的相关函数将设备唤醒。可以看出，设备唤醒操作发生在输入事件进入派发队列之前，因此它不会被ANR所阻塞。

·POLICY_FLAG_BRIGHT_HERE， 功能同POLICY_FLAG_WOKE_HERE类似。当屏幕变暗时，PhoneWindowManager为输入事件返回此派发策略会使NativeInputManager调用PowerManager相关函数促使屏幕恢复正常亮度。

·POLICY_FLAG_PASS_TO_USER：这个派发策略决定了事件是否会进入正式的派发流程。PhoneWindowManager认为事件不需要派发给用户时，就会从policyFlags中移除这个派发策略，在dispatchOnceInnerLocked () 函数中会将事件丢弃。

而interceptKeyBeforeQueueing () 则有更多的工作要做。interceptKeyBeforeQueueing () 其实是系统处理一些重要的系统功能按键的场所。这些系统按键包括VOLUME键、POWER键以及ENDCALL键等。这些按键的特点是需要系统能够快速响应，而且不应受到ANR的影响（设想一下，当发生ANR后连电话都无法挂断时用户有多懊恼）。PhoneWindowManager在完成这些按键的功能之后会像

interceptMotionBeforeQueueing () 一样返回那些policyFlags。另外，虽然PhoneWindowManager没有这么做，不过可以通过返回 POLICY_FLAG_DISABLE_KEY_REPEAT 派发参数禁止InputDispatcher 为此按键生成重复事件。



注意

由于这两个函数都是在Reader线程中调用的，如果开发者在这两个函数中做了不该做的事而导致它们的调用被死锁，则会因为InputReader无法继续运转而导致无法接受后续的用户输入事件。这是一种类似ANR但是却比ANR更糟糕的情况，因为Reader线程没有超时保护，一旦被这两个函数的调用所卡死，将导致系统持续无法响应用户输入，直到卸下电池为止。

第三个函数interceptKeyBeforeDispatching () 是按键事件所特有的。它为PhoneWindowManager提供了处理那些不那么关键的系统按键事件的机会，例如HOME、MENU以及SEARCH键等。它通过返回一个延迟时间来控制派发流程。延迟时间小于0，则表示PhoneWindowManager已

经自己处理了这个事件，从而要求InputDispatcher将其丢弃，不再进行派发。延迟时间大于0，则表示PhoneWindowManager尚不清楚应该如何处理此按键事件，要求InputDispatcher在指定的时间之后再做尝试。而延迟时间为0，则表示InputDispatcher可以正常地将此事件派发给目标窗口。

丢弃事件的原因比较好理解。不过为什么PhoneWindowManager会出现尚不清楚如何处理事件的情况呢？举个例子，我们知道，Android原生支持通过按下音量下+电源键进行屏幕截图操作。这个组合键的操作是在interceptKeyBeforeQueueing () 中完成的。用户完美地同时按下这两个按键是不可能的，因此PhoneWindowManager以音量下键进入派发队列为组合键的前导，然后在规定的时间内等待电源键的到来。如果电源键在时间内到来，则组合键生效，启动截屏动作，此时的音量下键则被PhoneWindowManager消耗掉，需要InputDispatcher将其抛弃。而如果电源键在限定的时间内没能到来，则说明用户仅仅是按下音量下而已，需要将其派发给用户。这就出现了一个问题，在音量下键被注入派发队列后，到等待电源键到来的这个短暂的时间里，如果InputDispatcher通过interceptKeyBeforeDispatching () 向PhoneWindowManager询问音量下键的是否应该发送给用户时，PhoneWindowManager就迷糊了，在电源键按下或等待超时之前，它不知道是否应该将其派发给用户。此时，PhoneWindowManager就返回一个大于0的延迟事件，告诉InputDispatcher：还是等等再说吧。



注意

这个函数的调用发生在派发线程中，此函数中的阻塞会使得系统对输入事件失去响应，并且也使得InputDispatcher失去检测ANR的能力。

另外，在通过notifyMotion () 或notifyKey () 将事件注入派发队列之前，会为事件增加POLICY_FLAG_TRUSTED派发策略，表明来自InputReader的事件是受信任的。既然有受信任的事件，自然也会有不受信任的事件。不受信任的事件是指通过InputDispatcher :: injectInputEvent () 函数以软件方式注入，并且调用者没有 android.permission.INJECT_EVENTS权限的事件。这个函数由Java层的IMS提供接口供其他模块调用，用于以软件方式模拟用户操作，一个典型的使用者就是Monkey。在这三个函数的调用中，不受信任的事件在经过NativeInputManager时便被忽略了，因此不会被PhoneWindowManager处理。这很好理解，由于PhoneWindowManager所做的都是一些重要的工作，自然不希望某些软件可以在没有申请对应权限的情况下通过注入按键事件的方式达到自己的目的。

这三个函数的详细内容涉及系统的方方面面，这超出了本章的讨论范围。感兴趣的读者可以实际分析一下它们的实现，会从中发现很多有趣的内容。

2. InputFilter

在notifyMotion () 与notifyKey () 之前，都会将输入事件通过mPolicy->filterInputEvent () 将事件交给DispatcherPolicy进行输入事件的过滤。如果这个函数返回了false，则事件将不再被注入派发队列中。这个调用被NativeInputManager转发给Java层IMS的同名函数中。参考以下实现代码：

```
[InputManagerService.java::InputManagerService.filterInputEvent()]
final
boolean filterInputEvent(InputEvent event, int policyFlags) { synchronized
(mInputFilterLock) { if (mInputFilter != null) { try {
mInputFilter.filterInputEvent(event, policyFlags); } catch (RemoteException
e) { ... // 忽略异常 } return false; // 只要InputFilter处理了事件，都会返回
false } } return true; // 仅当没有mInputFilter时才会返回true }
```

可以看到，实际的事件过滤者是mInputFilter，它是由IMS提供的接口setInputFilter () 设置的一个实现了IInputFilter接口的对象。注意，这个IInputFilter和EditText控件的InputFilter没有任何关系。

使用者可以通过setInputFilter () 函数将自己的IInputFilter对象设置给IMS，从此便可以开始进行输入事件的监听操作了。不过需要注意的是，IMS认为一旦事件被一个InputFilter截获过，这个事件便被丢弃了（返回false）。如果IInputFilter的使用者不作为，将会导致输入事件无法响应，也不会引发ANR。因此，IInputFilter的使用者截获事件之后，需要以软件的方式将事件重新注入InputDispatcher。使用者可以从IInputFilter的install () 回调所提供的IInputFilterHost对象完成事件的重新注入。不过，对重新注入的事件内容是什么输入系统则不做要求，因此InputFilter机制为使用者提供了篡改输入事件的能力。

5.3.4 输入事件的派发总结

至此，本节讨论了输入事件从注入到派发队列开始，到选取合适的InputTarget进行派发的详细过程。需要读者重点掌握的内容有：

- 派发线程循环的间歇性工作方式，nextWakeupTime对线程循环的响应及其使用方法。
- 输入事件被InputDispatcher丢弃的原因。
- 选择目标窗口的两种方式。
- 重复按键事件的产生过程。
- DispatcherPolicy对派发过程的影响。

本节所讨论的派发流程在 dispatchEventLocked () 函数中终止了。这个函数将会使得输入事件离开 InputDispatcher，进入指定的目标窗口。下一节将详细讨论这一过程。

5.4 输入事件的发送、接收与反馈

InputDispatcher选择好输入事件的目标窗口后，便准备将事件发送给它。然而，InputDispatcher运行于system_server进程中，而窗口运行于其所在的应用进程中。它们之间如何互动呢？本节将详细讨论这个问题。

InputDispatcher与窗口之间的通信的核心在于InputChannel。因此首先要深入理解InputChannel的工作原理。

5.4.1 深入理解InputChannel

什么是InputChannel呢？InputChannel的本质是一对SocketPair（非网络套接字）。SocketPair用来实现在本机内进行进程间的通信。一对SocketPair通过socketpair（）函数创建，其使用者可以因此而得到两个相互连接的文件描述符。这两个描述符可以通过套接字接口send（）和recv（）进行写入和读取，并且向其中一个文件描述符写入的数据，可以从另一个描述符中读取。同pipe（）所创建的管道不同，SocketPair的两个文件描述符是双通的，因此非常适合用来进行进程间的交互式通信。

InputChannel就是SocketPair描述符及其操作的封装，而且是成对使用的。配对的两个InputChannel分别保有一个SocketPair的描述符，并分别分配给InputDispatcher与窗口。因此InputDispatcher向其保有的InputChannel中写入的输入事件，可以由窗口从自己的InputChannel中读取。并且窗口可以将事件处理完毕的反馈写入InputChannel中，InputDispatcher再将反馈进行读取。

InputChannel的创建就如同SokectPair一样，需要同时创建一个InputChannel对。创建InputChannel对的方法是通过InputChannel的静态函数：openInputChannelPair（）。看一下其实现：

```
[InputTransport.cpp-->InputChannel::openInputChannelPair()]
status_t
InputChannel::openInputChannelPair(const String8& name, sp &
outServerChannel, sp & outClientChannel) { int sockets[2]; // 通过
socketpair()函数创建一对SocketPair，并保存在sockets数组中 if
(socketpair(AF_UNIX, SOCK_SEQPACKET, 0, sockets)) { ..... // 错误处
理 } ..... // 配置两个套接字的读写缓冲区尺寸 // 创建Server端的
InputChannel对象 outServerChannel = new
InputChannel(serverChannelName, sockets[0]); // 创建Client端的
InputChannel对象 outClientChannel = new
InputChannel(clientChannelName, sockets[1]); return OK; }
```

其实SocketPair的两端的地位是对等的，并不存在Service端与Client端之分。Android为了便于管理从而区分了这两端。

InputChannel的构造函数十分简单，保存描述符与名称之后，通过fcntl设置描述符为非阻塞式读写，也就是当描述符中没有数据时或写入缓冲区满时立即返回错误代码而不是阻塞。

InputChannel提供的接口有两个，分别为负责将一个事件写入另一端的sendMessage（）和从另一端读取一个事件的receiveMessage（）。它们的实现都非常简单，不过是通过send（）与recv（）进行数据的发送与接收而已。另外，这两个函数的参数为代表一个输入事件的InputMessage结构体。

为了方便通过socket接口进行传输，InputMessage结构体的定义比较复杂，如下所示：

```
[InputTransport.h-->InputMessage] struct InputMessage { struct Header {  
    uint32_t type; // 描述Body中所存储的事件类型 } header; union Body { //  
这是一个联合体，根据携带事件的类型不同而使用不同的数据结构  
    struct Key { uint32_t seq; // 注意seq字段，它唯一地表示了一个事件 .....  
    } key; // key字段保存了按键事件的所有信息 struct Motion { uint32_t seq;  
..... } motion; // motion字段保存了Motion事件的所有信息 struct Finished
```

```
{ uint32_t seq; bool handled; } finished; // finished字段保存了窗口的反馈信息 } body; };
```

不管它的定义多么复杂，只要知道InputMessage是为了便于在socket中进行传输而对输入事件信息所进行的一个封装即可。

InputChannel相对简单，相信读者对其原理与功能已经有了深入理解。图5-13描述了InputChannel对的工作原理。

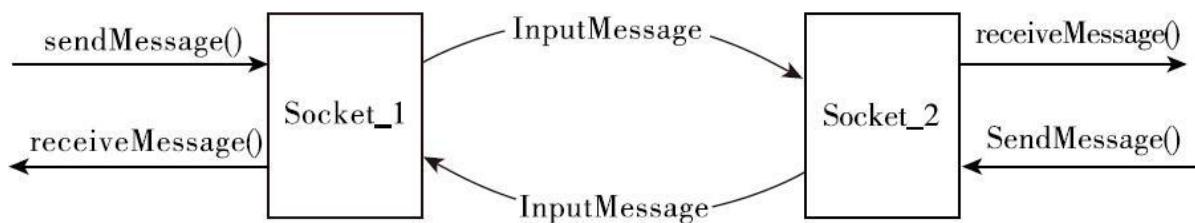


图5-13 InputChannel对的工作原理

5.4.2 连接InputDispatcher和窗口

上一节介绍了InputChannel对可以为两个进程提供互相传输InputMessage结构体的功能。接下来探讨InputChannel对如何完成InputDispatcher和窗口之间的连接。

回顾一下第4章最开始的SampleWindow例程，它在向WMS添加窗口时，将一个InputChannel作为传出参数传递给了IWindowSession.add()函数，其调用如下：

```
mSession.add(mWindow, 0, mLp, View.VISIBLE, mInsets,  
mInputChannel);
```

完成这个调用后，`mInputChannel`便成为`InputChannel`对的客户一端。看一下WMS的`addWindow ()`如何完成这个工作：

[`WindowManagerService.java--> WindowManagerService.addWindow()`]

```
public int addWindow(....., InputChannel outInputChannel) ..... // 仅当窗口  
的inputFeatures未指定NO_INPUT_CHANNEL选项时才会为此窗口创建  
InputChannel对 if (outInputChannel != null && (attrs.inputFeatures &  
WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANN  
EL) == 0) { String name = win.makeInputChannelName(); /*  
openInputChannelPair()函数调用了Native层的openInputChannelPair()函  
数，并将两个Native层的InputChannel对象封装为Java对象返回 */  
InputChannel[] inputChannels =  
InputChannel.openInputChannelPair(name); // 其中一个InputChannel端交  
给WindowState保存 win.setInputChannel(inputChannels[0]); // 另一个  
InputChannel则通过transferTo()函数传递给调用者  
inputChannels[1].transferTo(outInputChannel); // 将WindowState所保存的  
InputChannel向IMS进行注册  
mInputManager.registerInputChannel(win.mInputChannel,  
win.mInputWindowHandle); } ..... // 将所有的窗口信息更新到
```

```
InputManagerService mInputMonitor.updateInputWindowsLw(false  
/*force*/); }
```

可以看出，在WMS添加窗口时，会创建一对InputChannel，其中一个保存在Window-State中，并注册给IMS，它是服务端。另外一个则通过传出参数outInputChannel交给调用者，是客户端。因此需要从客户端与服务端两个方面探讨InputDispatcher和窗口建立连接的过程。

1.服务端连接的建立

首先来探讨服务端InputChannel的去向。

在addWindow () 函数中，有以下三项工作：

- 通过WindowState.setInputChannel () 函数保存服务端的InputChannel。
- 通过IMS.registerInputChannel () 将InputChannel注册到IMS中。
- 通过InputMonitor.updateInputWindowsLw () 将所有窗口的信息更新到IMS。

(1) 理解InputWindowHandle

首先来探讨WindowState.setInputChannel () ，这个函数的实现非常简单，如下所示：

```
[WindowState.java-->WindowState.setInputChannel()] void  
setInputChannel(InputChannel inputChannel) { ..... mInputChannel =  
inputChannel; mInputWindowHandle.inputChannel = inputChannel; }
```

其中，mInputWindowHandle是一个InputWindowHandle对象。经过5.3.1节的学习，读者应该熟悉这个名字。没错，InputDispatcher中的InputWindowHandle就是这个Java对象在Native层的代理。每一个WindowState都通过final关键字定义了一个InputWindow-Handle对象，一个Java层的InputWindowHandle对象也都唯一对应着一个Native层的InputWindowHandle对象，而且它们三者这种一一对应的关系终身不变。

由Java层的InputWindowHandle类的定义可以发现，它位于com.android.server.input包内，因此属于输入系统范畴的一个类，而它的成员字段则描述了与窗口布局相关的信息，包括窗口的位置与尺寸、可见性、是否具有焦点等。因此，如果说WindowState是窗口存在于WMS中的形式，那么InputWindowHandle则是窗口存在于输入系统中的形式。

setInputChannel () 函数使得InputWindowHandle对象保存了服务端的InputChannel，不过并没有将窗口的布局信息也更新到InputWindowHandle里。布局信息的更新发生在InputMonitor.updateInputWindowsLw () 中，这个内容稍后讨论。



注意

为了简洁流程，后续的讨论忽略了JNI层的处理。InputDispatcher所保存的InputWindowHandle其实是一个抽象类，需要子类实现updateInfo()函数。其真正实现是位于JNI层的NativeInputWindowHandle，它所实现的updateInfo () 函数既是将Java层InputWindowHandle中所保存的窗口布局信息搬运到Native层的InputWindowHandle中名为mInfo的InputWindowInfo结构体中。InputDispatcher选取目标窗口时所使用的布局信息就来自这个结构体。

(2) 向InputDispatcher注册InputChannel

接下来看IMS.registerInputChannel () 的工作原理。

[InputManagerService.java-->InputManagerService.registerInputChannel()]
public void registerInputChannel(InputChannel inputChannel,
InputWindowHandle inputWindowHandle) { // 直接调用
nativeRegisterInputChannel, 由Native层完成注册工作 // 注意最后一个

参数monitor的值为false nativeRegisterInputChannel(mPtr, inputChannel, inputWindowHandle, false /* monitor ? */); }

实际的注册过程由JNI层的NativeInputManager交给InputDispatcher的同名函数完成注册。

[InputDispatcher.cpp-->InputDispatcher::registerInputChannel()] status_t
InputDispatcher::register InputChannel(const sp & inputChannel, const sp &
inputWindowHandle, bool monitor) { { // ① 为传入的InputChannel创
建一个Connection对象并对其进行封装 sp connection = new
Connection(inputChannel, inputWindowHandle, monitor); // 以
InputChannel的描述符为键， 将Connection对象保存在
mConnectionsByFd字典中 int fd = inputChannel->getFd();
mConnectionsByFd.add(fd, connection); /* 如果传入的monitor参数为
true， 将其添加到mMonitoringChannels列表中。 在介绍目标窗口的查找
时， 凡是位于此列表中的inputChannel都会被放在目标列表中。因此
monitor 参数为true时， 注册的InputChannel将可以收到所有输入事件(被
DispatcherPolicy认定需要丢弃的除外) */ if (monitor) {
mMonitoringChannels.push(inputChannel); } /* ② 监听InputChannel的可
读性。 mLooper的pollOnce()本质上就是epoll_wait()， 因此 Looper对象
具有监听文件描述符可读性事件的能力，在此注册InputChannel可读性
事件，并在事件到来时通过handleReceiveCallback()回调进行处理 */

```
mLooper->addFd(fd, 0, ALOOPER_EVENT_INPUT,  
handleReceiveCallback, this); } mLooper->wake(); return OK; }
```

在InputDispatcher中，InputChannel被封装为一个Connection对象。Connection类描述了从InputDispatcher到目标窗口中的一个连接，其中保存了向窗口发送的事件的状态信息。在Connection中，重要的成员有：

- mInputPublisher，InputPublisher类的一个对象，它封装InputChannel并直接对其进行写入和读取。另外，它也负责InputMessage结构体的封装与解析。
- outboundQueue，用于保存等待通过此Connection进行发送的事件队列。
- waitQueue，用于保存已经通过此Connection将事件发送给窗口，正在等待窗口反馈的事件队列。

随后，registerInputChannel（）将InputChannel的可读性事件注册到mLooper中（本质是Epoll机制，不熟悉的读者可参考5.2.1节）。当来自窗口的反馈到来时，派发线程的mLooper->pollOnce（）将会被唤醒，并回调handleReceiveCallback（）进行处理。因此，窗口反馈的到来也会导致派发线程进入下一次派发循环，不过是在handleReceiveCallback（）回调完成后。

完成服务端InputChannel的注册之后，InputDispatcher便拥有了向客户端的InputChannel发送InputMessage，以及通过回调handleReceiveCallback()响应来自客户端的反馈的能力。这个过程的基本原理是，先将输入事件放入Connection的outboundQueue队列中，然后再由mInputPublisher依次将队列中的事件封装为InputMessage并写入InputChannel，直到队列为空，或InputChannel的写入缓冲区满。写入的事件将被移存到waitQueue队列里。随后派发线程陷入休眠状态。当窗口在另一端读取事件并发来反馈后，派发线程因InputChannel可读而被唤醒，并在handleReceiveCallback()中通过Connection的mInputPublisher读取反馈信息，将其与waitQueue中等待反馈的事件进行配对成功后，将事件从waitQueue中移除，完成事件派发。

整个过程如图5-14所示，其中实线描述了事件发送过程，而虚线则描述了反馈接收的过程。

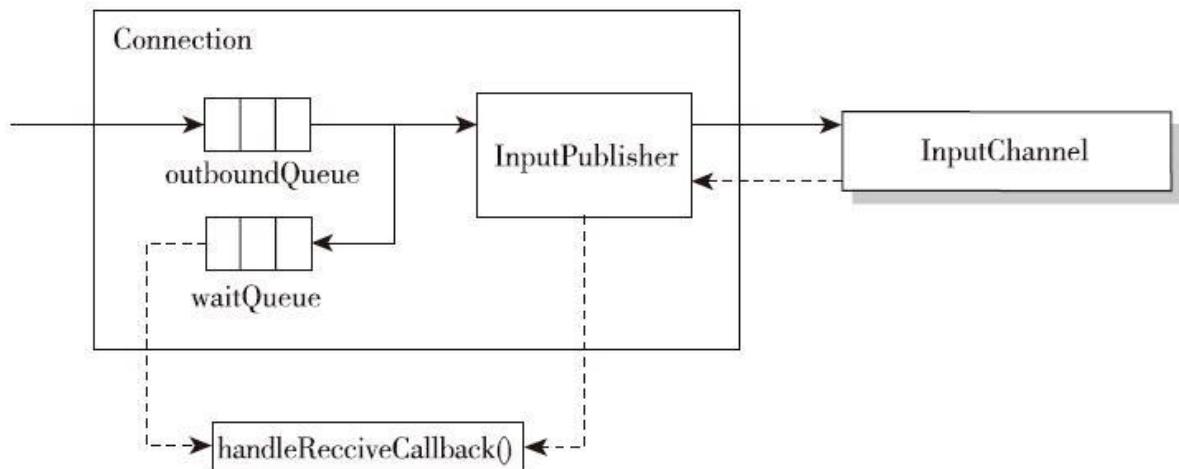


图5-14 Connection工作原理

(3) 布局信息的更新

前文讨论了InputWindowHandle的概念，不过尚未讨论InputWindowHandle如何获取窗口的布局信息，以及InputWindowHandle如何放入InputDispatcher的mInputWindow-Handles列表中。接下来讨论这个问题。

WMS.addWindow () 函数最后会调用

InputMonitor.updateInputWindowsLw () 函数。它的功能就是将WMS中所有的窗口布局信息填充到对应的InputWindowHandle对象里，然后提交给InputDispatcher。看一下其实现：

```
[InputMonitor.java-->InputMonitor.updateInputWindowsLw()]
public void updateInputWindowsLw(boolean force) { ..... /* 获取用于遍历所有WMS
窗口的枚举器。其参数表示将采用反向遍历。这是因为InputDispatcher
中 InputWindowHandle列表的ZOrder顺序与WMS相反 */
final AllWindowsIterator iterator = mService.new AllWindowsIterator(
 WindowManagerService.REVERSE_ITERATOR); while
(iterator.hasNext()) { final WindowState child = iterator.next(); final
InputWindowHandle inputWindowHandle = child.mInputWindowHandle; /* 将窗口的布局信息转存到inputWindowHandle中，然后将
```

```
inputWindowHandle保存到mInputWindowHandles列表中 */
addInputWindowHandleLw(inputWindowHandle, child, flags, type,
isVisible, hasFocus, hasWallpaper); } // mInputWindowHandles列表提交给
InputManagerService
mService.mInputManager.setInputWindows(mInputWindowHandles); // 清
空mInputWindowHandles列表，以便下次使用
clearInputWindowHandlesLw(); }
```

IMS的setInputWindows () 由InputDispatcher的同名函数实现：

```
[InputDispatcher.cpp-->InputDispatcher::setInputWindows()] void
InputDispatcher::setInputWindows( const Vector >& inputWindowHandles)
{ { /* ① 首先使用传入的InputWindowHandle列表替换现有列表。并将
现有列表保存在oldWindowHandles中 */ Vector > oldWindowHandles =
mWindowHandles; mWindowHandles = inputWindowHandles; sp
newFocusedWindowHandle; /* 接下来，遍历新的InputWindowHandle列
表，使其从Java层搬运窗口的布局信息，并确定处于焦点状态的
InputWindowHandle */
for (size_t i = 0; i < mWindowHandles.size(); i++) {
const sp & windowHandle = mWindowHandles.itemAt(i); // ② 调用
updateInfo()函数。将Java层的布局信息搬运到其InputWindowInfo结构
体中 if (!windowHandle->updateInfo()||windowHandle-
>getInputChannel()==NULL) { ..... // 错误处理 } // 记录处于焦点状态的
```

```
窗口 if (windowHandle->getInfo()->hasFocus) {  
    newFocusedWindowHandle = windowHandle; } } // ③ 将焦点窗口保存到  
mFocusedWindowHandle中 if (mFocusedWindowHandle !=  
newFocusedWindowHandle) { ..... mFocusedWindowHandle =  
newFocusedWindowHandle; } ..... /* 接下来遍历旧有的  
InputWindowHandle列表，检查其中的窗口是否在新列表中已经不存在  
了。如果不存在，则意味着窗口已经被WMS删除，此时需要释放被删  
除的窗口信息 */ for (size_t i = 0; i < oldWindowHandles.size(); i++) {  
    const sp & oldWindowHandle = oldWindowHandles.itemAt(i); if  
(!hasWindowHandleLocked(oldWindowHandle)) { oldWindowHandle-  
>releaseInfo(); } } // ④ 因为窗口列表的更新，需要唤醒派发线程重新  
尝试派发 mLooper->wake(); }
```

可以看出，InputDispatcher采用了比较激进的窗口列表更新方式——直接使用传入的列表替换了当前列表。替换后，对列表中的每个InputWindowHandle进行updateInfo () 的调用。updateInfo () 函数将Java层InputWindowHandle中保存的布局信息搬运到InputWindowInfo结构体中保存。另外，在遍历的过程中如果发现了处于焦点状态的窗口，则将其保存在mFocusedWindowHandle成员中。

setInputWindows () 函数的动作产生的结果有两个：

·更新mWindowHandles列表，使之可以正确地反映当前时刻WMS中所有窗口的布局状态。这个列表为按坐标点的方式查找目标窗口提供了依据。

·更新mFocusedWindowHandle成员。它为按照焦点查找目标窗口提供了依据。

因此，setInputWindows（）函数为事件派发的目标提供了新的选择。如果派发线程此时正在为无法将mPendingEvent派发给一个窗口而处于休眠状态，此时可以将其唤醒，以重新尝试派发动作。这就是在函数最后调用mLooper->wake（）的原因。

鉴于InputDispatcher对窗口布局信息的强烈依赖，在WMS中，每当布局信息发生变化，都需要通过调用setInputWindows（）通知输入系统。读者可以在WMS中搜索InputMonitor.updateInputWindowsLw（）函数的调用了解更新布局信息的时机。

（4）服务端的连接总结

至此，输入系统一端的连接便建立完成了。这一端的连接通过registerInputChannel（）和setInputWindows（）两个步骤完成。

registerInputChannel（）为InputChannel创建了一个Connection，并监听了InputChannel的可读事件，使得InputDispatcher拥有了将事件发送给

InputChannel并接受反馈的能力。

setInputWindows () 则将窗口的布局信息更新至InputDispatcher，使窗口处于目标窗口的候选列表中。

接下来探讨一下在窗口端如何使用InputChannel。

2. 窗口端连接的建立

当窗口端通过addWindow () 函数获取InputChannel后，便会使用它创建一个Input-EventReceiver对象。InputEventReceiver对象可以接收来自InputChannel的输入事件，并触发其onInputEvent () 回调。参考SimpleWindow的相关代码：

```
// 使用InputChannel与当前线程的Looper创建一个InputHandler对象  
InputHandler = new InputHandler(mInputChannel, Looper.myLooper());
```

InputHandler是SampleWindow实现的一个继承自InputEventReceiver的类，看一下这个类的实现：

```
class InputHandler extends InputEventReceiver { Looper mLooper = null;  
public InputHandler(InputChannel inputChannel, Looper looper) {  
super(inputChannel, looper); // 调用InputEventReceiver的构造函数 }  
@Override public void onInputEvent(InputEvent event) { ..... // 对event进  
行处理 super.onInputEvent(event); } }
```

可以看出，在客户端使用InputChannel是非常简单的，只要构造一个继承自Input-EventReceiver的类，并在重写的onInputEvent () 方法中添加想要的事件处理即可。那么InputEventReceiver是如何工作的呢？看到其构造函数所需的参数之后，读者应该能够想象到：将InputChannel的可读事件注册到Looper中，然后在事件到来时从InputChannel中读取InputMessage，并翻译成InputEvent，然后回调InputEventReceiver的onInputEvent () 函数。

是不是这样呢？看一下InputEventReceiver的构造过程。

InputEventReceiver的构造函数调用了nativeInit () 函数，从Native层进行构造。参考其实现：

```
[android_view_InputEventReceiver.cpp-->nativeInit()]
static jint
nativeInit(JNIEnv* env, jclass clazz, jobject receiverObj, jobject
inputChannelObj, jobject messageQueueObj) { ..... // 创建一个
NativeInputEventReiver对象
sp receiver = new
NativeInputEventReceiver(env, receiverObj, inputChannel, messageQueue);
// 执行其初始化操作
status_t status = receiver->initialize(); ..... }
```

NativeInputEventReceiver的构造函数也很简单。它保存了Java层InputEventReceiver对象的引用，并创建了一个InputConsumer类型的对象对InputChannel进行封装。

InputConsumer与InputPublisher一样，它也封装了InputChannel，负责对其进行写入和读取操作，同时也负责InputMessage的封装与解析。不过它们的功能正好相反，InputConsumer接收的是输入事件，发送的则是反馈。

再看一下receiver->initialize () 函数的实现：

```
[android_view_InputEventReceiver.cpp-->NativeInputEventReceiver::initialize()] status_t NativeInputEventReceiver::initialize() { // 获取InputChannel的fd int receiveFd = mInputConsumer.getChannel()->getFd(); // 通过Looper注册 InputChannel的可读性事件 mMessageQueue->getLooper()->addFd(receiveFd, 0, ALOOPER_EVENT_INPUT, this, NULL); return OK; }
```

正好印证了之前的猜测，果然是通过Looper监听InputChannel的可读性事件。参考addFd的函数签名可知，当有InputMessage可读时，NativeInputEventReceiver的handleEvent () 函数会被Looper调用，此时便可以通过InputConsumer从InputChannel中读取事件，然后回调Java层的onInputEvent () 函数。在Java层完成事件的处理后，便可通过InputConsumer发送处理完毕的反馈给InputDispatcher。

在窗口端，输入事件回调的发起者是创建InputEventReceiver时所使用的Looper。因此这个Looper便是输入系统中的第三台水泵。

可见窗口端的连接要简单很多，如图5-15所示。

3.InputDispatcher与窗口的连接

至此，双方对InputChannel的封装工作便完成了，它们的事件传输连接也就此建立完毕。如果将图5-14与图5-15通过InputChannel连接起来，便构成了InputDispatcher到窗口的事件发送循环。在理解InputDispatcher到窗口的连接与基本工作原理后，对事件发送循环的分析便轻松多了。

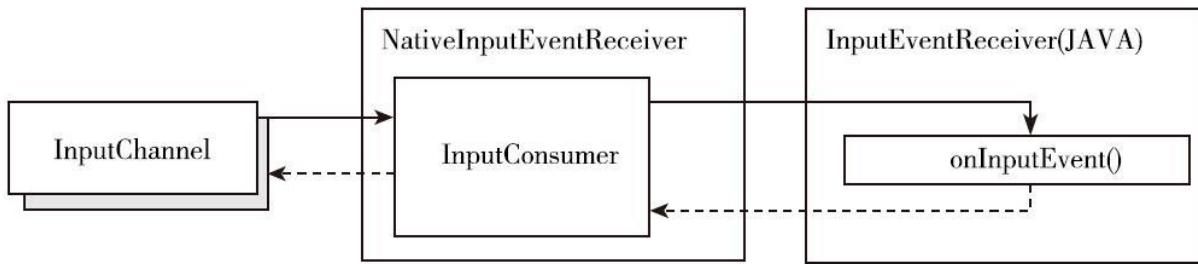


图5-15 窗口端的连接

5.4.3 事件的发送

本节将讨论事件发送循环的细节。注意事件发送循环与5.3节所说的派发循环的不同。派发循环是指InputDispatcher不断地从派发队列取出事件、寻找合适的窗口并进行发送的过程，是InputDispatcher线程的主要

工作。而事件发送循环则是InputDispatcher通过Connection对象将事件发送给窗口，并接受其反馈的过程。

回到5.3节的终点InputDispatcher的dispatchEventLocked () 函数，InputDispatcher已经为事件选择好了数个由InputTarget描述的派发目标，这个函数的工作就是将事件发送过去：

```
[InputDispatcher.cpp-->dispatchEventLocked::dispatchEventLocked()] void
InputDispatcher::dispatchEventLocked(nsecs_t currentTime, EventEntry*
eventEntry, const Vector & inputTargets) { for (size_t i = 0; i <
inputTargets.size(); i++) { const InputTarget& inputTarget =
inputTargets.itemAt(i); // 首先根据发送目标InputChannel的fd获取对应的
Connection对象 ssize_t connectionIndex =
getConnectionIndexLocked(inputTarget.inputChannel); if (connectionIndex
>= 0) { sp<Connection> connection = mConnectionsByFd.valueAt(connectionIndex); // 发
送事件 prepareDispatchCycleLocked(currentTime, connection, eventEntry,
&inputTarget); } else { ..... } } }
```

此函数十分简单，根据InputTarget中的InputChannel找到其对应的Connection，然后调用prepareDispatchCycleLocked () 函数发送事件。

再看prepareDispatchCycleLocked () 的实现：

[InputDispatcher.cpp--> InputDispatcher::prepareDispatchCycleLocked()]

void InputDispatcher::prepareDispatchCycleLocked(.....) { // 错误处理，以及对多点触摸事件分割的处理。本书不做讨论 // 将事件添加到 Connection 的发送队列中 enqueueDispatchEntriesLocked(currentTime, connection, eventEntry, inputTarget); }

enqueueDispatchEntriesLocked() : [InputDispatcher.cpp--> InputDispatcher::enqueueDispatchEntriesLocked()]

void InputDispatcher::enqueueDispatchEntriesLocked(.....) { bool wasEmpty = connection->outboundQueue.isEmpty(); /* 将事件信息封装成 DispatchEntry，然后注入Connection 的发送队列中。

FLAG_DISPATCH_AS_IS 表示不修改事件的action 的类型，按原样添加到发送队列。为了使流程清晰，这里省略了使用其他选项向发送队列添加DispatchEntry 的代码 */ enqueueDispatchEntryLocked(connection, eventEntry, inputTarget, InputTarget::FLAG_DISPATCH_AS_IS); // 如果 Connection 的发送队列从空到有事件，则立刻启动发送循环 if (wasEmpty && !connection->outboundQueue.isEmpty()) { startDispatchCycleLocked(currentTime, connection); } }

这个函数首先使用enqueueDispatchEntryLocked () 函数将事件信息封装为DispatchEntry结构体，然后追加到Connection的发送队列中。 DispatchEntry附加了一个seq字段，这个字段是事件的序号，窗口对事件的反馈将携带这个序号。

其实这个函数使用不同的选项多次调用了enqueueDispatchEntryLocked ()。FLAG_DISP-ATCH_AS_IS表示不修改事件的action类型，而其他的选项则会使得action类型发生变化。举个例子，当HOVER_MOVE事件从一个窗口移动到另一个窗口时，InputDispatcher会将两个窗口都作为发送目标，但是会对它们设置不同的选项（AS_HOVER_EXIT和AS_HOVER_ENTER）。于是在enqueueDispatchEntryLocked () 中，发送给第一个窗口的事件的action从HOVER_MOVE被修改为HOVER_EXIT，而发送给第二个窗口的action则变成HOVER_ENTER。感兴趣的读者可以参考findTouchedWindowTargetsLocked () 中的相关代码了解这些选项的设置过程。

当事件被封装为DispatchEntry并追加到Connection的发送队列后，就可以通过startDispatchCycleLocked () 开始发送循环了。



说明

是否开始发送循环有一个判断：当追加事件之前发送队列为空。也就是说，如果发送队列不为空，发送循环应该是已经启动了。当完成事件反馈的探讨后，读者即可理解这个判断的意义了。

接下来再看startDispatchCycleLocked () 的实现：

```
[InputDispatcher.cpp-->InputDispatcher::startDispatchCycleLocked()] void
InputDispatcher::startDispatchCycleLocked(nsecs_t currentTime, const sp &
connection) { // 这个循环会不断地从发送队列中获取DispatchEntry，并
将事件发送到InputChannel中 while (connection->status ==
Connection::STATUS_NORMAL && !connection-
>outboundQueue.isEmpty()) { DispatchEntry* dispatchEntry = connection-
>outboundQueue.head; EventEntry* eventEntry = dispatchEntry-
>eventEntry; // 根据不同的事件类型，选择InputPublisher不同的发送函
数 switch (eventEntry->type) { case EventEntry::TYPE_KEY: { status =
connection->inputPublisher.publishKeyEvent(...); break; } case
EventEntry::TYPE_MOTION: { MotionEntry* motionEntry = static_cast
(eventEntry); status = connection-
>inputPublisher.publishMotionEvent(...); break; } } if (status) { // 如果发
送失败，例如InputChannel的发送缓冲区满，则停止发送，下次执行此
函数时再试 ..... return; } // 将DispatchEntry转存到waitQueue中
connection->outboundQueue.dequeue(dispatchEntry); connection-
>waitQueue.enqueueAtTail(dispatchEntry); } }
```

至此，输入事件被InputPublisher以InputMessage的形式写入
InputChannel中。然后将事件转存到waitQueue中等待窗口的反馈。

5.4.4 事件的接收

当InputPublisher将事件以InputMessage的形式写入inputChannel中之后，如5.4.2节所述，窗口端的Looper会因此而被唤醒，并执行NativeInputEventReceiver的handle-Event ()。handleEvent () 直接调用了consumeEvent () 函数：

```
[android_view_InputEventReceiver.cpp-->NativeInputEventReceiver::consumeEvent()] status_t NativeInputEventReceiver::consumeEvents(JNIEnv* env, bool consumeBatches, nsecs_t frameTime) { ..... /* 类似于 startDispatchCycleLocked(),consumeEvents()也通过一个循环，尽可能多地从Input- Channel中读取事件信息 */ for (;;) { uint32_t seq; InputEvent* inputEvent; /* ① 通过mInputConsumer的consume()函数从InputChannel中读取一条InputMessage。解析为 InputEvent后，通过inputEvent参数传出 */ status_t status = mInputConsumer.consume(&mInputEventFactory, consumeBatches, frameTime, &seq, &inputEvent); ..... // ② 接下来，根据事件的类型分别创建KeyEvent与MotionEvent类型的Java对象 jobject inputEventObj; switch (inputEvent->getType()) { case AINPUT_EVENT_TYPE_KEY: inputEventObj = android_view_KeyEvent_fromNative(env, static_cast (inputEvent)); break; case AINPUT_EVENT_TYPE_MOTION: inputEventObj =
```

```
        android_view_MotionEvent_obtainAsCopy(env, static_cast (inputEvent));  
        break; } if (inputEventObj) { // ③ 通过JNI回调Java层的  
        InputEventReceiver的dispatchInputEvent()函数 env-  
        >CallVoidMethod(mReceiverObsjGlobal,  
        gInputEventReceiverClassInfo.dispatchInputEvent, seq, inputEventObj); }  
        else { ..... } } ..... } }
```

可见，当NativeInputEventReceiver通过Looper发觉InputChannel有输入事件到来时，会通过InputConsumer从InputChannel中读取一个InputEvent，然后通过这个InputEvent生成对应类型的Java层的InputEvent对象。最后，通过JNI回调Java层InputEventReceiver的dispatchInputEvent () 函数。

再看Java层的处理：

[InputEventReceiver.java-->InputEventReceiver.dispatchInputEvent()]

```
private void dispatchInputEvent(int seq, InputEvent event) { /* 以event的序  
列号为键，将来自InputDispatcher的序列号保存到字典中。Java层  
InputEvent在 创建时也会分配一个唯一的序号，用来对其进行唯一标识  
*/ mSeqMap.put(event.getSequenceNumber(), seq); // 调用onInputEvent()  
函数 onInputEvent(event); }
```

dispatchInputEvent函数首先在字典中保存来自InputDispatcher的事件序列号，以满足发送反馈之需。之后便调用onInputEvent () 函数，交由子类进行输入事件的实际处理工作。

到这里，事件的接收工作就完成了。onInputEvent () 函数可由使用者重写，从而实现各种各样的工作。例如在SampleWindow例子中，收到点击事件后就会退出程序，而Android控件系统的根ViewRootImpl在收到事件后会将其派发给特定的控件。

然而在onInputEvent () 中完成事件的处理后，千万别忘记给输入系统一个反馈。这个事件还在Connection对象的waitQueue中等着呢。

5.4.5 事件的反馈与发送循环

事件的反馈动作可由InputEventReceiver.finishInputEvent () 发起。

```
[InputEventReceiver.java-->InputEventReceiver.finishInputEvent()]
public
final void finishInputEvent(InputEvent event, boolean handled) { // 从字典
    中将此事件对应的序列号取出 int index =
    mSeqMap.indexOfKey(event.getSequenceNumber()); int seq =
    mSeqMap.valueAt(index); mSeqMap.removeAt(index); /* 由
    nativeFinishInputEvent完成反馈动作，其中，handled参数表示此事件是
    否被处理，由具体事件 处理者根据情况设置 */
    nativeFinishInputEvent(mReceiverPtr, seq, handled); }
```

不用说，这个发送过程由NativeInputEventReceiver完成。

```
[android_view_InputEventReceiver.cpp-->NativeInputEventReceiver::finishInputEvent()] status_t  
NativeInputEventReceiver::finishInputEvent(uint32_t seq, bool handled) { //  
    由mInputConsumer将seq与handle两个信息以InputMessage的形式写入  
    InputChannel中    status_t status = mInputConsumer.sendFinishedSignal(seq,  
    handled); ..... return status; }
```

回顾一下5.4.2节InputChannel注册的相关内容可知，当窗口端的InputChannel被写入数据时，会触发服务端InputChannel可读事件，因此InputDispatcher的派发线程被唤醒并执行handleReceiveCallback()回调。

```
[InputDispatcher.cpp-->InputDispatcher::handleReceiveCallback()] int  
InputDispatcher::handleReceiveCallback(int fd, int events, void* data) {  
    InputDispatcher* d = static_cast (data); { // ① 首先，根据可读的  
    InputChannel的描述符获取对应的Connection对象    ssize_t  
    connectionIndex = d->mConnectionsByFd.indexOfKey(fd); sp connection =  
    d->mConnectionsByFd.valueAt(connectionIndex); if (!(events &  
    (ALOOPER_EVENT_ERROR | ALOOPER_EVENT_HANGUP))) { for  
    (;;) { // ② 然后在循环中不断地读取尽可能多的反馈信息    status =  
    connection->inputPublisher .receiveFinishedSignal(&seq, &handled); if
```

```
(status) { break; } // ③ 调用finishDispatchCycleLocked()函数完成对反馈的处理 d->finishDispatchCycleLocked(currentTime, connection, seq, handled); gotOne = true; } ..... } }
```

finishDispatchCycleLocked () 函数只是将向InputDispatcher发送一个命令。最终处理反馈的函数是doDispatchCycleFinishedLockedInterruptible () 。

```
[InputDispatcher.cpp-->InputDispatcher::  
doDispatchCycleFinishedLockedInterruptible()] void  
InputDispatcher::doDispatchCycleFinishedLockedInterruptible(  
CommandEvent* commandEntry) { sp<Connection> connection = commandEntry->connection; uint32_t seq = commandEntry->seq; bool handled =  
commandEntry->handled; // 从waitQueue中，按照序号取出反馈对应的事件 if (dispatchEntry == connection->findWaitQueueEntry(seq)) { // 将事件从waitQueue中移除 connection->waitQueue.dequeue(dispatchEntry);  
..... } // 启动下一次发送循环 startDispatchCycleLocked(now(),  
connection); }
```

对于输入事件反馈的处理主要有两个方面：

- 将事件从Connection的waitQueue队列中删除。这个删除动作标志着此事件的派发流程完成，也意味着这个事件经过漫长的加工、传递之旅

后生命的结束。

·最后调用startDispatchCycleLocked () 函数继续尝试发送队列中的下一个事件，又回到5.4.3节中所讨论的发送流程。

从startDispatchCycleLocked () 开始向窗口发送事件，窗口接受事件并完成处理后发送反馈，InputDispatcher收到反馈后再次调用startDispatchCycleLocked () 函数发送下一个事件。这个过程构成了一个循环，即输入事件的发送循环。

区别于InputDispatcher的派发循环，发送循环仅涉及Connection对象与窗口之间的事件发送与反馈。另外，由于反馈的接收由mLooper触发并完成，因此发送循环也工作于派发线程中，是派发循环的一部分。

注意Connection的两个队列outboundQueue和waitQueue，它们是否拥有元素决定了客户端窗口的响应状态。当两者中任意一个队列拥有事件时，都标志着窗口尚未接受事件，或发送事件反馈。这表示窗口使用者有可能处于ANR状态。



说明

为了突出发送循环的特点，本节所引用的 doDispatchCycleFinishedLockedInterruptible () 函数做了不少简化。事实上，事件反馈中所携带的handled参数对发送循环的执行过程是有影响的。当handled为false，即窗口端没有处理这个事件时，会触发所谓的fallback机制。举个生活中的例子，小明同时买了A和B两种猫粮，先把A猫粮喂了猫，发现它根本不吃。于是小明又递上了B猫粮，期望他的猫会喜欢。在fallback机制里，InputDispatcher就是小明，猫粮就是事件，而窗口就是猫。当事件没有被窗口处理时，InputDispatcher会向 DispatcherPolicy询问此事件是否拥有备选事件（由 PhoneWindowManager.dispatchUnhandledKey () 函数确定），如果有，则会将备选事件放到outboudQueue的队首，在接下来的发送循环中发送给窗口。读者可以参考InputDispatcher：： afterKeyEventLockedInterruptible () 和Phone-WIndowMananger.dispatchUnhandledKey () 两个函数自行学习这个机制。

5.4.6 输入事件的发送、接收与反馈总结

这一节讨论了InputDispatcher与窗口之间的连接体系，以及 InputDispatcher将事件发送给窗口并接收其反馈的过程。读者需要重点理解的内容有：

- InputChannel的概念与原理。

- InputDispatcher对InputChannel的封装与事件发送。

- InputEventReceiver对InputChannel的封装与事件接收。

- 事件反馈与发送循环的原理。

完成这一节的讨论之后，我们也就完成了输入事件从设备节点开始，经过Input-Reader、InputDispatcher以及InputChannel最终进入窗口事件处理函数，并从窗口以反馈的形式回到InputDispatcher的漫长过程。事件从Connection的waitQueue队列中移除的时刻，就是输入事件完成使命而结束其生命的时刻。

下一节将讨论一些没能体现在输入事件处理的总体流程中的重要话题。

5.5 关于输入系统的其他重要话题

5.5.1 输入事件ANR的产生

在5.3.1节介绍目标窗口查找时提到过，作为派发目标的窗口必须已经准备好接收新的输入事件，否则判定窗口处于未响应状态，终止事件的派发过程，并在一段时间后再试。倘若5s后窗口仍然未准备好接收输入事件，将导致ANR。直接引发ANR的原因有很多，例如Activity生命周期函数调用超时，服务启动超时，以及最常见的输入事件处理超时等。本节将从输入事件超时的角度讨论ANR的产生原因与过程。

参考一下按焦点查找目标窗口的函数findFocusedWindowTargetsLocked()：

```
[InputDispatcher.cpp-->InputDispatcher::findFocusedWindowTargetsLocked()] int32_t  
InputDispatcher::findFocusedWindowTargetsLocked(.....) { ..... // 通过  
isWindowReadyForMoreInputLocked()函数检查窗口是否准备好接受新  
事件 if (!isWindowReadyForMoreInputLocked(currentTime,  
mFocusedWindowHandle, entry)) { /* 如果尚未准备好，则通过  
handleTargetsNotReadyLocked()对原因进行记录，并安排时间尝试重试  
派发，或者引发ANR */ injectionResult =
```

```
handleTargetsNotReadyLocked(currentTime, entry,
mFocusedApplicationHandle, mFocusedWindowHandle, nextWakeupTime,
"Waiting because the focused window has not finished " "processing the
input events that were previously delivered to it."); goto Unresponsive; } .....
return injectionResult; }
```

这个函数首先判断窗口是否可以接收事件，如果窗口无法接收事件，则安排稍后再试，或者引发ANR。

1. 窗口可以接收事件的条件

```
[InputDispatcher.cpp-->InputDispatcher::
isWindowReadyForMoreInputLocked()] bool
InputDispatcher::isWindowReadyForMoreInputLocked(nsecs_t
currentTime, const sp & windowHandle, const EventEntry* eventEntry) { //
首先获取窗口的Connection
ssize_t connectionIndex =
getConnectionIndexLocked(windowHandle->getInputChannel()); if
(connectionIndex >= 0) { sp connection =
mConnectionsByFd.valueAt(connectionIndex); // 共通原因：
InputPublisher被阻塞 if (connection->inputPublisherBlocked) { return
false; } // 对按键事件来说，要求Connection必须处于空闲状态 if
(eventEntry->type == EventEntry::TYPE_KEY) { return connection-
>outboundQueue.isEmpty() && connection->waitQueue.isEmpty(); } // 对
```

```
Motion事件来说，可以发送事件的条件相对宽松些，只要窗口能在0.5s  
内发送反馈即可 if (!connection->waitQueue.isEmpty() && currentTime  
>= connection->waitQueue.head->eventEntry->eventTime +  
STREAM_AHEAD_EVENT_TIMEOUT) { return false; } } return true; }
```

可以看出，判断窗口是否可以接受事件的依据有两个：InputPublisher是否被阻塞以及Connection两个队列的状态。

InputPublisher的工作是将事件信息写入InputChannel中，如果窗口端因为某种原因迟迟未能从InputChannel中将事件读取就会导致SocketPair的写入缓冲区满。

Connection两个队列的状态体现了发送循环的状态。如果两个队列至少有一个队列为空，则表示Connection正处于发送循环的过程中，否则处于空闲状态。

对按键事件来说，仅当Connection处于空闲状态，也就是窗口已经完成对之前事件的响应之后才会发送给窗口。因为之前的输入事件有可能会影响焦点窗口，进而影响按键事件的接收者。例如，用户快速地按下了两次BACK键，第一个BACK键将会发送给位于顶端的窗口，这个事件可能会导致窗口关闭，因此第一个BACK键的处理行为决定了第二个BACK应该发送给哪个窗口。因此按键事件的发送要求窗口完成对所有之前事件的处理。

而Motion事件的条件则相对宽松些，允许Connection处于发送循环的过程中，但是如果等待队列中的第一个事件没能在0.5s获得反馈，则判定窗口处于未响应状态。这是因为Motion事件具有实时性的特点——用户的意图就是希望输入事件发送给他所看到的窗口，所以不在乎之前事件的处理结果。

2.重试派发与ANR的引发

如果isWindowReadyForMoreInputLocked () 判定窗口无法接收事件，则调用handleTargetsNotReadyLocked () 安排重试，或引发ANR。看一下其实现：

```
[InputDispatcher.cpp--> InputDispatcher::handleTargetsNotReadyLocked()]
int32_t InputDispatcher::handleTargetsNotReadyLocked( ..... ) { if
(applicationHandle == NULL && windowHandle == NULL) { ..... // 这种
情况说明系统尚未完成启动，可忽略 } else { // ① 如果是第一次发生窗
口未响应的情况，则记录下未响应的窗口信息，并设置引发ANR的时
间点 if (mInputTargetWaitCause !=
INPUT_TARGET_WAIT_CAUSE_APPLICATION_NOT_READY) { // 获
取引发ANR的超时时间 if (windowHandle != NULL) { // 如果有目标窗
口，则获取由窗口所指定的超时时间 timeout = windowHandle-
>getDispatchingTimeout(
DEFAULT_INPUT_DISPATCHING_TIMEOUT); } else if
```

```
(applicationHandle != NULL) { // 如果没有目标窗口，则从AMS获取超  
时时间 timeout = applicationHandle->getDispatchingTimeout(  
DEFAULT_INPUT_DISPATCHING_TIMEOUT); } else { timeout =  
DEFAULT_INPUT_DISPATCHING_TIMEOUT; } // 接下来记录窗口超时  
的信息 mInputTargetWaitCause =  
INPUT_TARGET_WAIT_CAUSE_APPLICATION_NOT_READY;  
mInputTargetWaitStartTime = currentTime; // 检测到未响应的时间  
mInputTargetWaitTimeoutTime = currentTime + timeout; // 设置引发ANR  
的时间 mInputTargetWaitTimeoutExpired = false; // 当引发ANR后将被置  
true ..... } } ..... // ② 检查是否引发ANR if (currentTime >=  
mInputTargetWaitTimeoutTime) { // 当前事件大于引发ANR的时间后，  
则引发ANR onANRLocked(currentTime, applicationHandle,  
windowHandle, entry->eventTime, mInputTargetWaitStartTime, reason);  
*nextWakeUpTime = LONG_LONG_MIN; return  
INPUT_EVENT_INJECTION_PENDING; } else { // 如果尚未到达引发  
ANR的时间点，设置nextWakeUpTime后返回，等待下次再试 if  
(mInputTargetWaitTimeoutTime < *nextWakeUpTime) { *nextWakeUpTime  
= mInputTargetWaitTimeoutTime; } return  
INPUT_EVENT_INJECTION_PENDING; } }
```

这段代码应该分为两个部分进行分析。首先是当第一次发生无法将事
件发送给窗口的情况时，也就是代码中的①处，在这里设置了引发

ANR的时间点。在随后的重试过程中，将当前时间点与引发ANR的时间点进行比对，并决定是否引发ANR或再次重试。

将isWindowReadyForMoreInputLocked () 和 handleTargetsNotReadyLocked () 放到派发流程中去分析，可以得到如下一个流程：

InputDispatcher从派发队列中获取了一个事件mPendingEvent，并为它查找目标窗口，然后通过isWindowReadyForMoreInputLocked () 确定此窗口是否可以接收事件。如果可以则将事件放入窗口的Connection对象的发送队列中并启动发送循环，否则调用handleTargetsNotReadyLocked () 计算引发ANR的时间点，然后通过返回 INPUT_EVENT_INJECTION_PENDING停止对mPendingEvent的派发工作，并通过设置nextWakeupTime使派发循环进入休眠状态。休眠的过程有可能因为窗口反馈到来、新输入事件到来或新的窗口信息到来而唤醒，派发线程便重新开始对mPendingEvent的派发过程，进而重新寻找目标窗口，再通过isWindowReadyForMoreInputLocked () 检查目标窗口是否准备好接收事件，如果可以接收事件，则将其提交给 Connection进行发送，并重置之前所设置的ANR信息。否则再次进入 handleTargetsNotReadyLocked ()，这时将当前时间与ANR时间进行对比，以决定引发ANR还是再次使派发线程进入休眠。

5.5.2 焦点窗口的确定

InputDispatcher在派发按键事件时使用了mFocusedWindowHandle作为目标窗口。这个mFocusedWindowHandle成员变量表示系统中处于焦点状态的窗口，它的设置来自InputDispatcher::setInputWindows()函数（参考5.4.2节的布局信息的更新等相关内容）。也就是说，焦点窗口的确定是由WMS完成的。本节将讨论焦点窗口是如何被确定的。

在WMS中，用于更新焦点窗口的函数是updateFocusedWindowLocked()。参考其实现：

```
[WindowManagerService.java-->WindowManagerService.updateFocusedWindowLocked()]
private boolean updateFocusedWindowLocked(int mode, boolean updateInputWindows) {
    // 使用computeFocusedWindowLocked()函数计算新的焦点窗口
    WindowState newFocus = computeFocusedWindowLocked();
    if (mCurrentFocus != newFocus) {
        mCurrentFocus = newFocus;
        ..... // 将焦点变化通知给窗口使用者，以及其他感兴趣的组件
        return true;
    }
    return false;
}
```

updateFocusedWindowLocked()通过computeFocusedWindowLocked()函数计算出新的焦点窗口，并保存在mCurrentFocus成员中。当焦点发生变化时，会通知窗口的使用者或其他感兴趣的组件（如PhoneWindowManager、IMS等）。

computeFocusedWindowLocked () 函数依照Id从小到大的顺序对每一个 DisplayContent (对应一块屏幕，关于其概念的详细介绍请参考第4章) 调用findFocusedWindowLocked () 函数，从中查找焦点窗口。其实每个DisplayContent都拥有自己的焦点窗口，然而真正的焦点窗口只有一个。于是Id越小的DisplayContent的焦点窗口具有更高的优先级。设备的主屏幕Id为0，因此主屏幕将拥有最高的焦点优先级。

由于findFocusedWindowLocked () 函数的逻辑有些绕，因此这里不给出它的代码了，直接向读者介绍其工作原理。寻找焦点窗口的基本原则是沿ZOrder的顺序从上向下遍历窗口，第一个
WindowState.canReceiveKeys () 返回值为true的窗口拥有焦点。
canReceiveKeys () 体现了窗口可以获得焦点的条件，其实现如下：

```
[WindowState.java-->WindowState.canReceiveKeys()] public final boolean  
canReceiveKeys() { return isVisibleOrAdding() // 窗口不会即将被移除  
&& (mViewVisibility == View.VISIBLE) // 窗口可见 // LayoutParams没有  
指定FLAG_NOT_FOCUSABLE选项 && ((mAttrs.flags &  
 WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE) == 0); }
```

也就是说，一个DisplayContent中的焦点窗口是位于其窗口列表中沿 ZOrder从上到下的顺序第一个满足以下条件的窗口：

- 窗口不会即将被移除。

- 窗口可见。
- 没有指定FLAG_NOT_FOCUSABLE选项。

这是findFocusedWindowLocked () 函数查找焦点窗口的基本逻辑，很简单不是吗？不过这个函数逻辑比较绕的主要原因是由于焦点窗口的另外一个限制：mFocusedApp。mFocusedApp是一个AppWindowToken（参考第4章），在WMS中用来表示当前处于Resume状态的Activity。它是由AMS在开始启动一个Activity时调用WMS的setFocusedApp () 函数设置的。

考虑以下应用场景，当用户从Launcher中启动一个Activity之后，在新Activity的窗口显示之前便立刻按下了BACK键。很明显，用户的意图是关闭刚刚启动的Activity，而不是退出Launcher。然而，由于BACK键按得过快，新Activity尚未创建窗口，因此按照之前讨论的焦点窗口的查找条件，Launcher的窗口将会作为焦点窗口而接收BACK键，从而使得Launcher被退出。这与用户的意图是相悖的。

为了解决这个问题，WMS要求焦点窗口必须属于mFocusedApp，或者位于mFocusedApp的窗口之上。再看添加这个限制之后，上面的应用场景会变成什么样子。当启动新Activity时，尽管其窗口尚未创建，但这个窗口的AppWindowToken已经被添加到mAppTokens列表的顶部，并通过setFocusedApp () 设置为mFocusedApp。此时再更新焦点窗口

时发现第一个符合焦点条件的窗口属于Launcher，但Launcher的AppWindowToken位于mFocusedApp也就是新Activity之下，因此它不能作为焦点窗口。这样，在新Activity创建窗口之前的一个短暂时段内，系统处于无焦点窗口的情况。此时到来的BACK键在InputDispatcher中会因找不到目标窗口而触发handleTargetsNotReadyLocked ()，从而进入等待重试状态。随后新Activity创建了自己的窗口并添加到WMS里，这时WMS可以成功地将这个窗口作为焦点，并通过IMS.setInputWindows () 将其更新到InputDispatcher。这个更新动作会唤醒派发线程立刻对BACK键进行重试，这次重试便找到了新Activity的窗口作为目标，并将事件发送过去，从而正确地体现用户的意图。



说明

这正体现了“百分之九十的代码用于处理百分之十的情况”这句箴言。

另外，还有一种应用场景，就是当前处于Resume状态的Activity将自己所有的窗口都添加到另一块屏幕中（另一个DisplayContent）的情况，读者可以自行分析在这种情况下mFocusedApp对焦点窗口选择的影响。

因此，可以得到焦点窗口的选择有如下原则：

- DisplayContent的Id值越低，其内部窗口拥有越高的焦点优先级。
- 窗口在DisplayContent中的显示次序越靠前，其拥有越高的焦点优先级。
- 焦点窗口必须处于正常显示状态（没有调用removeWindow（）），处于可见状态，并且没有指定FLAG_NOT_FOCUSABLE选项。
- 所有位于当前Activity之下的窗口不得获取焦点。若当前Activity以及其上都没有窗口满足上述条件，则此DisplayContent没有焦点窗口。

由此可以得出影响窗口焦点的动作有以下几个：

- 窗口的增删操作。
- 窗口次序的调整。
- Activity的启动与退出。
- DisplayContent的增删操作。

因此当这些动作发生时，WMS都会触发对updateFocusedWindowLocked（）的调用，并更新窗口的布局信息到输入系统，以实现按键事件正确派发。

5.5.3 以软件方式模拟用户操作

随着软件功能的复杂度增加，人工测试越来越耗时。为此，自动化测试则日益受到测试工程师的重视。自动化测试的基础就是通过软件的方式模拟用户操作，也就是用软件方式向输入系统注入输入事件。经过本章前面内容的讨论可以发现，在Android输入系统中有两个可能的注入点：

- 直接向设备节点中写入原始输入事件。sendevent工具就是这么做的。其好处是使用便捷，可以使用脚本进行控制。而缺点就是原始输入事件可读性差，并且事件内容与硬件实现有关，因此通用性差。另外，sendevent工具并不适合对实时性要求较高的事件注入需求。因为sendevent一次只能发送一个事件，如果需要连续发送多个（如Motion事件）则进程调度将会占去过多时间而影响模拟效果。在这种情况下，最好的方式是写一个程序批量地向设备节点中写入事件数据。
- 向InputDispatcher的派发队列中注入输入事件。InputManagerService提供的injectInputEvent（）接口可以实现这个功能。其好处是通用性好，输入事件的信息可读性佳。而缺点是需要编写额外的代码。以这种方式注入事件的一个典型例子就是monkey测试程序。

第一种方式非常简单，读者可以参考sendevent工具的实现方式。它的实现代码位于/system/core/toolbox/sendevent.c中。

至于第二种方式，读者可以参考InputDispatcher的injectInputEvent () 函数的实现。其实，它与notifyKey () /notifyMotion () 等函数的行为相类似。它们最大的不同在于进行inject-InputEvent () 时会检查调用者是否拥有android.permission.INJECT_EVENTS权限。如果调用者没有这个权限，将只能向自己进程的窗口注入事件，并且会剥夺事件的TRUSTED策略选项。正如5.3.3节所说，被剥夺了TRUSTED策略选项的事件将无法被DispatcherPolicy用于触发系统级的动作。

5.6 本章小结

本章讨论了输入事件从设备节点开始到窗口完成事件处理并发送反馈为止的详细流程。将本章中的图5-11、图5-12、图5-14以及图5-15首尾相接，便产生了输入事件的完整生命周期。

本章所介绍的重要输入系统参与者有专门负责监听并读取原始输入事件的EventHub，从EventHub抽取事件进行加工处理的InputReader、选择目标窗口并进行派发的Input-Dispatcher，以及接收事件并发送反馈的InputEventReceiver等。

EventHub与InputReader所在的InputReaderThread，InputDispatcherThread，再加上InputEventReceiver所在的窗口线程构成了输入系统的三台首尾相接的水泵，它们全速运转时便将输入事件从设备节点一层一层地泵取到窗口的事件处理函数中。因此这三台水泵的健康状态直接影响输入系统是否能够正常运转。在实际的应用中，最容易出现问题的便是第三台水泵——InputEventReceiver所在的窗口线程。当它出现问题时，便会发生常见的ANR现象。

与这三台水泵对应的，本章讨论了三种循环：InputReader的事件读取与加工循环，InputDispatcher针对mPendingEvent的派发与重试的派发循环，Connection对象与InputEventReceiver之间的发送与反馈循环。这

三个者相互环扣，用精巧的协作演绎了以输入系统为舞台的优美舞蹈。

另外，输入系统的架构与实现构思非常优秀。在深入学习输入系统的过程中，读者可以从中领会到多种实用的开发模式与技术，如缓冲区操作、Epoll与INotify机制、线程间通信等。另外，派发循环与发送循环的协作尤为精巧，非常值得读者仔细钻研。

至此，输入系统的讨论到此便告一段落。相信经过本章与第4章的学习，读者对Android的窗口已经有了本质性认识。在后面的章节中将会以此为基础为读者详细解读Android中与窗口的使用相关的功能与系统服务。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第 6 章 深入理解控件系统

本章主要内容：

- 介绍创建窗口的新方法以及WindowManager的实现原理
- 探讨ViewRootImpl的工作方式
- 讨论控件树的测量、布局与绘制
- 讨论输入事件在控件树中的派发
- 介绍PhoneWindow的工作原理以及Activity窗口的创建方式

本章涉及的源代码文件名及位置：

·ContextImpl.java

frameworks/base/core/java/android/app/ContextImpl.java

·WindowManagerImpl.java

frameworks/base/core/java/android/view/WindowManagerImpl.java

·WindowManagerGlobal.java

frameworks/base/core/java/android/view/WindowManagerGlobal.java

· ViewRootImpl.java

frameworks/base/core/java/android/view/ViewRootImpl.java

· View.java

frameworks/base/core/java/android/view/View.java

· ViewGroup.java

frameworks/base/core/java/android/view/ViewGroup.java

· TabWidget.java

frameworks/base/core/java/android/widget/TabWidget.java

· HardwareRenderer.java

frameworks/base/core/java/android/view/HardwareRenderer.java

· FocusFinder.java

frameworks/base/core/java/android/view/FocusFinder.java

· Activity.java

frameworks/base/core/java/android/app/Activity.java

· PhoneWindow.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

· Window.java

frameworks/base/core/java/android/view/Window.java

· ActivityThread.java

frameworks/base/core/java/android/app/ActivityThread.java

6.1 初识Android的控件系统

第4章和第5章分别介绍了窗口的两个最核心的内容：显示与用户输入，同时也介绍了在Android中显示一个窗口并接收输入事件的最基本方法。但是这种方法过于基本，不便于使用。直接使用Canvas绘制用户界面以及使用InputEventReceiver处理用户输入是一项非常繁琐恼人的工作，因为你不得不亲历亲为以下复杂的工作：

- 测量各个UI元素（一段文字、一个图片）的显示尺寸与位置。
- 对各个UI元素进行布局计算与绘制。
- 当显示内容需要发生变化时进行重绘。出于效率考虑，你必须保证重绘区域尽可能小。
- 分析InputEventReceiver所接收的事件的类型，并确定应该由哪个UI元素响应这个事件。
- 需要处理来自WMS的很多与窗口状态相关的回调。

所幸Android的控件系统使得这些事情不需要我们亲历亲为。

自1983年苹果公司发布第一款搭载图形用户界面（GUI）操作系统的个人电脑Lisa以来的三十多年里，图形用户界面已经发展得相当成熟。无

论是运行于桌面系统还是Web，每一个面向图形用户界面的开发工具包（SDK）都至少内置实现了用户和开发者所公认的一套UI元素，尽管名称可能有所差异。例如文本框、图片框、列表框、组合框、按钮、单选按钮、多选按钮，等等。Android的控件系统不仅延续了对各种标准UI元素的支持，还针对移动平台的操作特点增加了使用更加方便、种类更加丰富的一系列新型的UI元素。



注意

在Android中，一个UI元素被称为一个视图（View），然而，笔者认为控件才是UI元素的更贴切的名字。因为UI元素不仅仅是为了向用户显示一些内容，更重要的是它们响应用户的输入并进行相应工作。本书后续部分将以控件来称呼UI元素（View）。

另外，本章的目的并不是介绍如何使用各种Android控件，而是介绍Android控件系统的工作原理。本章要求读者至少应了解使用Android控件的基本知识。

读者所熟知的Activity、各种对话框、弹出菜单、状态栏与导航栏等都是基于这套控件系统实现的。因此控件系统将是继WMS与IMS两大系统服务之后又一个需要我们攻克的目标。

6.1.1 另一种创建窗口的方法

在这一小节里将介绍另外一种创建窗口的方法，并以此为切入点来开始对Android控件系统的探讨。

这个例子将会在屏幕中央显示一个按钮，它会浮在所有应用之上，直到用户点击它为止。市面上某些应用的悬浮窗就是如此实现的。

·首先，读者使用Eclipse建立一个新的Android工程，并新建一个Service。然后在这个Service中增加如下代码：

```
// 将按钮作为一个窗口添加到WMS中 private void  
installFloatingWindow() { // ① 获取一个WindowManager实例 final  
WindowManager wm =  
(WindowManager) getSystemService(Context.WINDOW_SERVICE); // ②  
新建一个按钮控件 final Button btn = new Button(this.getBaseContext());  
btn.setText("Click me to dismiss!"); // ③ 生成一个  
WindowManager.LayoutParams，用于描述窗口的类型与位置信息  
LayoutParams lp = createLayoutParams(); // ④ 通过  
WindowManager.addView()方法将按钮作为一个窗口添加到系统中
```

```
wm.addView(btn, lp); btn.setOnClickListener(new View.OnClickListener()
{ @Override public void onClick(View v) { // ⑤ 当用户点击按钮时，将按钮从系统中删除 wm.removeView(btn); stopSelf(); } });
private LayoutParams createLayoutParams() { LayoutParams lp = new
 WindowManager.LayoutParams(); lp.type = WindowManager.LayoutParams.TYPE_PHONE;
 lp.gravity = Gravity.CENTER; lp.width =
 WindowManager.LayoutParams.WRAP_CONTENT; lp.height =
 WindowManager.LayoutParams.WRAP_CONTENT; lp.flags =
 WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE |
 WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL; return lp; }
```

·然后在新建的Service的onStartCommand () 函数中增加对installFloatingWindow () 的调用。

·在应用程序的主Activity的onCreate () 函数中调用startService () 以启动这个服务。

·在应用程序的AndroidManifest.xml中增加对权限 android.permission.SYSTEM_ALERT_WINDOW的使用声明。

当完成这些工作之后，运行这个应用即可得到如图6-1所示的效果。一个名为“Click me to dismiss！”的按钮浮在其他应用之上。而点击这个按钮后，它便消失了。



图6-1 浮动窗口例子的运行效果

读者可以将本例与第4章的例子SampleWindow做一个对比。它们的实现效果大同小异。而然，本章的这个例子无论是从最终效果、代码量、API的复杂度或可读性上都有很大的优势。这得益于对控件系统的使用。在这里，控件Button托管了窗口的绘制过程，并且将输入事件封装为更具可读性的回调。并且添加窗口时所使用的 WindowManager 实例掩盖了客户端与WMS交互的复杂性。更重要的是，本例所使用的接

口都来自公开的API，也就是说可以脱离Android源代码进行编译。这无疑会带来更方便的开发过程以及更好的程序兼容性。

因此，除非需要进行很底层的窗口控制，使用本例所介绍的方法向系统中添加窗口是最优的选择。

6.1.2 控件系统的组成

从这个例子中可以看到在添加窗口过程中的两个关键组件：Button和WindowManager。Button是控件的一种，继承自View类。不止Button，任何一个继承自View类的控件都可以作为一个窗口添加到系统中。

WindowManager其实是一个继承自ViewManager的接口，它提供了添加/删除窗口，更新窗口布局的API，可以看作WMS在客户端的代理类。不过WindowManager的接口与WMS的接口相差很大，几乎已经无法通过WindowManager看到WMS的模样。这也说明了WindowManager为了精简WMS的接口做过大量的工作。这部分内容也是本章的重点。

因此控件系统便可以分为继承自View类的一系列控件类与WindowManager两个部分。

6.2 深入理解WindowManager

WindowManager的主要功能是提供简单的API使得使用者可以方便地将一个控件作为一个窗口添加到系统中。本节将探讨它的工作原理。

6.2.1 WindowManager的创建与体系结构

首先需要搞清楚WindowManager是什么。

准确地说，WindowManager是一个继承自ViewManager的接口。

ViewManager定义了三个函数，分别用于添加/删除一个控件，以及更新控件的布局。

ViewManager接口的另一个实现者是ViewGroup，它是容器类控件的基本类，用于将一组控件容纳到自身的区域中，这一组控件称为子控件。

ViewGroup可以根据子控件的布局参数（LayoutParams）在其自身的区域中对子控件进行布局。

读者可以将WindowManager与ViewGroup进行类比：设想WindowManager是一个ViewGroup，其区域为整个屏幕，而其中的各个窗口就是一个一个的View。WindowManager通过WMS的帮助将这些View按照其布局参数（LayoutParams）显示到屏幕的特定位置。二者的

核心工作是一样的，因此 WindowManager 与 ViewGroup 都继承自 ViewManager。

接下来看一下 WindowManager 接口的实现者。本章最开始的例子通过 Context.getSystemService (Context.WINDOW_SERVICE) 的方式获取了一个 WindowManager 的实例，其实现如下：

```
[ContextImpl.java-->ContextImpl.getSystemService()]
public Object
getSystemService(String name) { // 获取WINDOW_SERVICE所对应的
ServiceFetcher ServiceFetcher fetcher =
SYSTEM_SERVICE_MAP.get(name); // 调用fetcher.getService()获取一个
实例 return fetcher == null ? null : fetcher.getService(this); }
```

Context 的实现者 ContextImpl 在其静态构造函数中初始化了一系列的 ServiceFetcher 来响应 getSystemService () 的调用并创建对应的服务实例。看一下 WINDOW_SERVICE 所对应的 ServiceFetcher 的实现：

```
[ContextImpl.java-->ContextImpl.static()]
registerService(WINDOW_SERVICE, new ServiceFetcher() { public Object
getService(ContextImpl ctx) { // ① 获取Context中所保存的Display对象
Display display = ctx.mDisplay; /* ② 倘若Context中没有保存任何Display
对象，则通过DisplayManager获取系统 主屏幕所对应的Display对象 */
if (display == null) { DisplayManager dm =
```

```
(DisplayManager)ctx.getOuterContext().getSystemService(  
    Context.DISPLAY_SERVICE); display =  
    dm.getDisplay(Display.DEFAULT_DISPLAY); } // ③ 使用Display对象作  
    为构造函数创建一个WindowManagerImpl对象并返回 return new  
    WindowManagerImpl(display); }});
```

由此可见，通过Context.getSystemService（）的方式获取的 WindowManager其实是WindowManagerImpl类的一个实例。这个实例的构造依赖于一个Display对象。第4章介绍过DisplayContent的概念，它在WMS中表示一块屏幕。而这里的Display对象与DisplayContent的意义是一样的，也用来表示一块屏幕。

再看一下WindowManagerImpl的构造函数：

```
[WindowManagerImpl.java--  
>WindowManagerImpl.WindowManagerImpl() public  
WindowManagerImpl(Display display) { this(display, null); } private  
WindowManagerImpl(Display display, Window parentWindow) { mDisplay  
= display; mParentWindow = parentWindow; }
```

其构造函数实在是出奇得简单，仅仅初始化了mDisplay与mParentWindow两个成员变量而已。从这两个成员变量的名字与类型来

推断，它们将决定通过这个WindowManagerImpl实例所添加的窗口的归属。



说明

WindowManagerImpl的构造函数引入了一个Window类型的参数parentWindow。Window类是什么呢？以Activity为例，一个Activity显示在屏幕上时包含了标题栏、菜单按钮等控件，但是在setContentView()时并没有在layout中放置它们。这是因为Window类预先为我们准备好了这一切，它们被称为窗口装饰。除了产生窗口装饰之外，Window类还保存了窗口相关的一些重要信息。例如窗口ID(IWindow.asBinder()的返回值)以及窗口所属Activity的ID(即AppToken)。在6.6.1节将会对这个类做详细介绍。

也许在WindowManagerImpl的addView()函数的实现中可以找到更多的信息。

```
[WindowManagerImpl.java-->WindowManagerImpl.addView()] public void  
addView(View view, ViewGroup.LayoutParams params) {  
    mGlobal.addView(view, params, mDisplay, mParentWindow); }
```

`WindowManagerImpl.addView()` 将实际操作委托给一个名为`mGlobal`的成员来完成，它随着`WindowManagerImpl`的创建而被初始化：

```
private final WindowManagerGlobal mGlobal =  
    WindowManagerGlobal.getInstance();
```

可见`mGlobal`的类型是`WindowManagerGlobal`，而且`WindowManagerGlobal`是一个单例模式，即一个进程中最多仅有一个`WindowManagerGlobal`实例。所有`WindowManagerImpl`都是这个进程唯一的`WindowManagerGlobal`实例的代理。

此时便对`WindowManager`的结构体系（见图6-2）有了一个清晰认识。

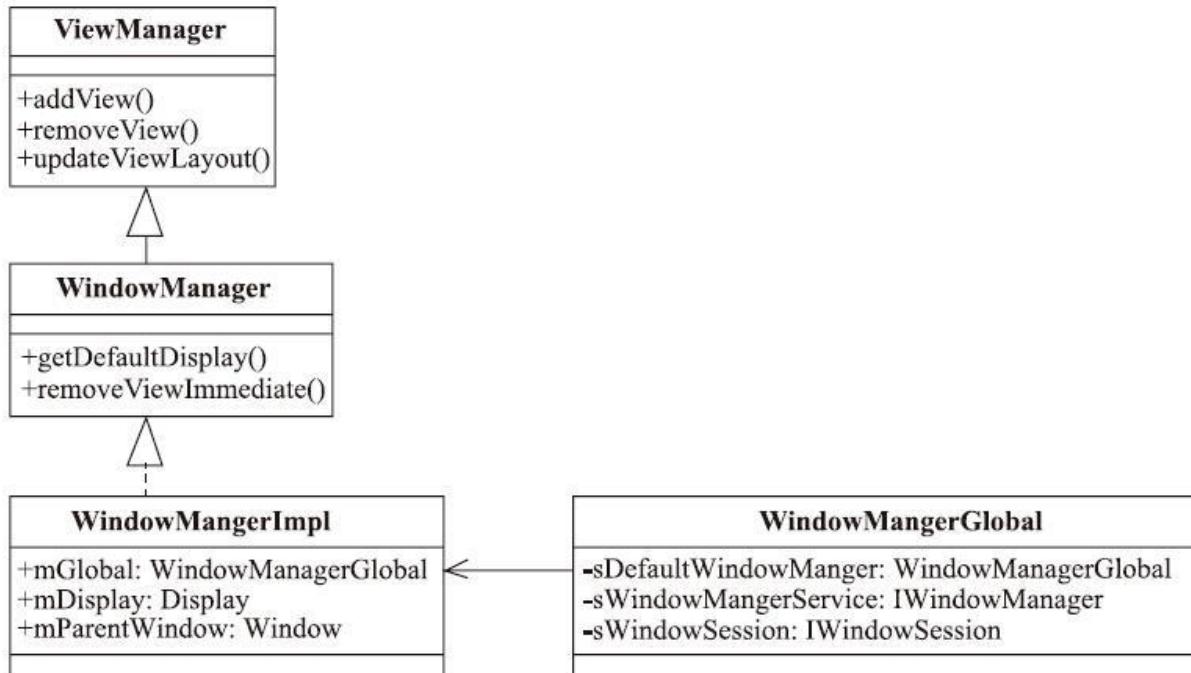


图6-2 WindowManager的结构体系

- ViewManager接口： WindowManager体系中最基本的接口。 WindowManager继承自这个接口说明了 WindowManager与 ViewGroup 在本质上的一致性。
- WindowManager接口： WindowManager接口继承自 ViewManager 接口的同时，根据窗口的一些特殊性增加了两个新的接口。
getDefalutDisplay () 用以得知这个 WindowManager 的实例会将窗口添加到哪个屏幕上。而 removeViewImmediate () 则要求 WindowManager 必须在这个调用返回之前完成所有的销毁工作。
- WindowManagerImpl类： WindowManager接口的实现者。它自身没有什么实际的逻辑， WindowManager 所定义的接口都是交由 WindowManagerGlobal 完成的。但是它保存了两个重要的只读成员，它们分别指明了通过这个 WindowManagerImpl 实例所管理的窗口将被显示在哪个屏幕上，以及将会作为哪个窗口的子窗口。因此在一个进程中， WindowManagerImpl 的实例可能有多个。
- WindowManagerGlobal类： 它没有继承上述任何一个接口，但它是 WindowManager 的最终实现者。它维护了当前进程中所有已经添加到系统中的窗口的信息。另外，在一个进程中仅有一个 WindowManagerGlobal 的实例。

在理清了WindowManager的结构体系后，便可以探讨WindowManager是如何完成窗口管理的。其管理方式体现在其对ViewManager的三个接口的实现上。为了简洁起见，我们将直接分析WindowManagerGlobal中的实现。

6.2.2 通过WindowManagerGlobal添加窗口

参考WindowManagerGlobal.addView () 的代码：

```
[WindowManagerGlobal.java-->WindowManagerGlobal.addView()] public  
void addView(View view, ViewGroup.LayoutParams params, Display  
display, Window parentWindow) { ..... // 参数检查 final  
WindowManager.LayoutParams wparams =  
(WindowManager.LayoutParams)params; /* ① 如果当前窗口需要被添加  
为另一个窗口的附属窗口（子窗口），则需要让父窗口视自己的情况  
对当前窗口的布局参数(LayoutParams)进行一些修改 */ if (parentWindow  
!= null) { parentWindow.adjustLayoutParamsForSubWindow(wparams); }  
ViewRootImpl root; View panelParentView = null; synchronized (mLock) {  
..... // WindowManager不允许同一个View被添加两次 int index =  
findViewLocked(view, false); if (index >= 0) { throw new  
IllegalStateException("....."); } // ② 创建一个ViewRootImpl对象并保存在  
root变量中 root = new ViewRootImpl(view.getContext(), display);  
view.setLayoutParams(wparams); /* ③ 将作为窗口的控件、布局参数以
```

及新建的ViewRootImpl以相同的索引值保存在三个数组中。到这步为止，我们可以认为完成了窗口信息的添加工作 */ mViews[index] = view; mRoots[index] = root; mParams[index] = wparams; } try { /* ④ 将作为窗口的控件设置给ViewRootImpl。这个动作将导致ViewRootImpl向WMS 添加新的窗口、申请Surface以及托管控件在Surface上的重绘动作。这才是真正意义上完成了 窗口的添加操作 */ root.setView(view, wparams, panelParentView); } catch (RuntimeException e) { } }

添加窗口的代码并不复杂。其中的关键点有：

·父窗口修改新窗口的布局参数。可能修改的只有LayoutParams.token和LayoutParams.mTitle两个属性。mTitle属性不必赘述，仅用于调试。而token属性则值得一提。回顾一下第4章的内容，每一个新窗口必须通过LayoutParams.token向WMS出示相应的令牌才可以。在addView () 函数中通过父窗口修改这个token属性的目的是减少开发者的负担。开发者不需要关心token到底应该被设置为什么值，只须将LayoutParams丢给一个WindowManager，剩下的事情就不用再关心了。

父窗口修改token属性的原则是：如果新窗口的类型为子窗口（其类型大于等于LayoutParams.FIRST_SUB_WINDOW，并小于等于LayoutParams.LAST_SUB_WINDOW），则LayoutParams.token所持有的令牌为其父窗口的ID（也就是IWindow.asBinder () 的返回值）。否则LayoutParams.token将被修改为父窗口所属的Activity的ID（也就是在

第4章中所介绍的AppToken），这对类型为TYPE_APPLICATION的新窗口来说非常重要。

从这点来说，当且仅当新窗口的类型为子窗口时，`addView ()` 的 `parentWindow`参数才是真正意义上的父窗口。这类子窗口有上下文菜单、弹出式菜单以及游标等，在WMS中，这些窗口对应的WindowState 所保存的`mAttachedWindow`就是`parentWindow`所对应的WindowState。然而另外还有一些窗口，如对话框窗口，类型为 TYPE_APPLICATION，并不属于子窗口，但需要AppToken作为其令牌，为此`parentWindow`将自己的AppToken赋予了新窗口的 `LayoutParams.token`。此时parent-Window便不是严格意义上的父窗口了。

·为新窗口创建一个ViewRootImpl对象。顾名思义，ViewRootImpl实现了一个控件树的根。它负责与WMS直接通信，负责管理Surface，负责触发控件的测量与布局，负责触发控件的绘制，同时也是输入事件的中转站。总之，ViewRootImpl是整个控件系统正常运转的动力所在，无疑是本章最关键的一个组件。

·将控件、布局参数以及新建的ViewRootImpl以相同的索引值添加到三个对应的数组`mViews`、`mParams`以及`mRoots`中，以供之后的查询之需。控件、布局参数以及ViewRootImpl三者共同组成了客户端的一个

窗口。或者说，在控件系统中的窗口就是控件、布局参数与 ViewRootImpl 对象的一个三元组。



注意

笔者并不认同将这个三元组分别存储在三个数组中的设计。如果创建一个 Window-Record 类来统一保存这个三元组将可以省去很多麻烦。

另外，mViews、mParams 以及 mRoots 这三个数组的容量是随着当前进程中的窗口数量的变化而变化的。因此在 addView () 以及随后的 removeView () 中都伴随着数组的新建、拷贝等操作。鉴于一个进程所添加的窗口数量不会太多，而且也不会很频繁，所以这些时间开销是可以接受的。不过笔者仍然认为相对于数组，ArrayList 或 CopyOnWriteArrayList 是更好的选择。

· 调用 ViewRootImpl.setView () 函数，将控件交给 ViewRootImpl 进行托管。这个动作将使得 ViewRootImpl 向 WMS 添加窗口、获取 Surface 以及 重绘等一系列操作。这一步是控件能够作为一个窗口显示在屏幕上的 根本原因！

总体来说， WindowManagerGlobal在通过父窗口调整布局参数之后， 将新建的ViewRootImpl、 控件以及布局参数保存在自己的三个数组中， 然后将控件交由新建的ViewRootImpl进行托管， 从而完成窗口的添加。 WindowManagerGlobal管理窗口的原理如图6-3所示。

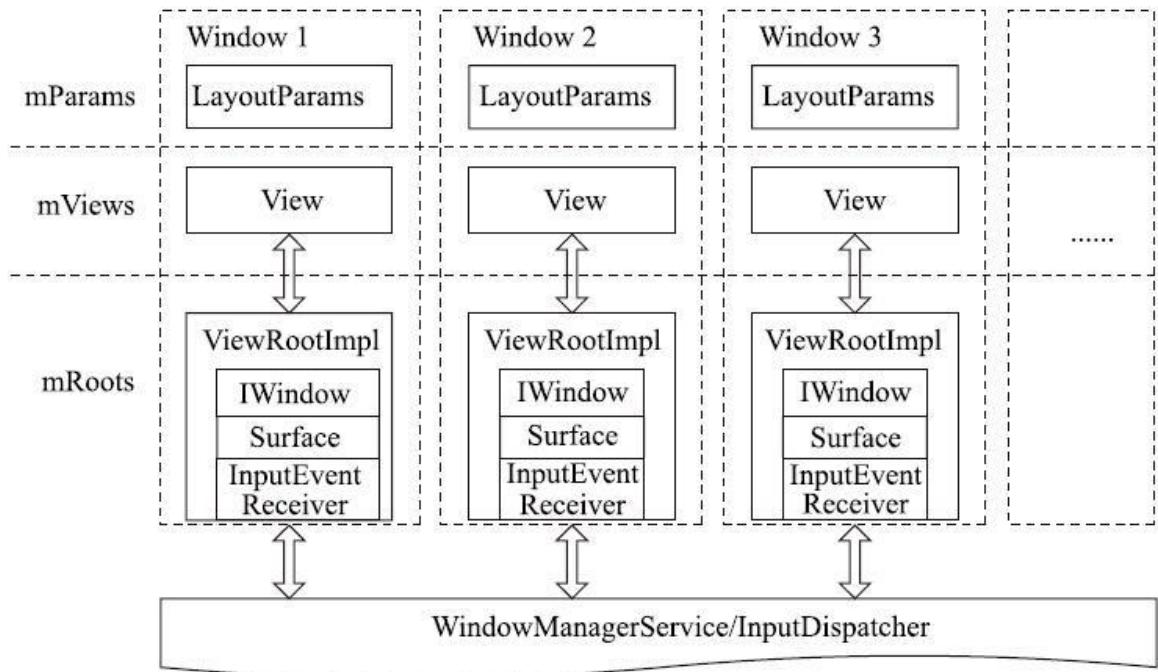


图6-3 WindowManagerGlobal的窗口管理

6.2.3 更新窗口的布局

ViewManager所定义的另外一个功能就是更新View的布局。在 WindowManager中，则是更新窗口的布局。窗口的布局参数发生变化时，如LayoutParams.width从100变为200，则需要将这个变化通知给

WMS使其调整Surface的大小，并让窗口进行重绘。这个工作在 WindowManagerGlobal中由updateViewLayout () 函数完成。

[WindowManagerGlobal.java--

```
>WindowManagerGlobal.updateViewLayout() public void  
updateViewLayout(View view, ViewGroup.LayoutParams params) { ..... //  
参数检查 final WindowManager.LayoutParams wparams =  
(WindowManager.LayoutParams)params; // 将布局参数保存到控件中  
view.setLayoutParams(wparams); synchronized (mLock) { // 获取窗口在三  
个数组中的索引 int index = findViewLocked(view, true); ViewRootImpl  
root = mRoots[index]; // 更新布局参数到数组中 mParams[index] =  
wparams; // 调用ViewRootImpl的setLayoutParams()使得新的布局参数生  
效 root.setLayoutParams(wparams, false); } }
```

更新窗口布局的工作在WindowManagerGlobal中是非常简单的，主要是保存新的布局参数，然后调用ViewRootImpl.setLayoutParams () 进行更新。

6.2.4 删除窗口

接下来探讨窗口的删除操作。在了解了WindowManagerGlobal管理窗口的方式后应该可以很容易地推断出删除窗口所需要做的工作：

- 从三个数组中删除此窗口所对应的元素，包括控件、布局参数以及ViewRootImpl。
- 要求ViewRootImpl从WMS中删除对应的窗口（IWindow），并释放一切需要回收的资源。

这个过程十分简单，这里就不引用相关的代码了。只是有一点需要说明一下：要求ViewRootImpl从WMS中删除窗口并释放资源的方法是调用ViewRootImpl.die（）函数。因此可以得出这样一个结论：
ViewRootImpl的生命从setView（）开始，到die（）结束。

6.2.5 WindowManager的总结

经过前面的分析，相信读者对WindowManager的工作原理有了深入认识。

- 鉴于窗口布局和控件布局的一致性，WindowManager继承并实现了接口View-Manager。
- 使用者可以通过Context.getSystemService
(Context.WINDOW_SERVICE) 来获取一个WindowManager的实例。
这个实例的真实类型是WindowManagerImpl。Window-ManagerImpl一旦被创建就确定了通过它所创建的窗口所属哪块屏幕、哪个父窗口。

- WindowManagerImpl除了保存窗口所属的屏幕以及父窗口以外，没有任何实质性的工作。窗口的管理都交由WindowManagerGlobal的实例完成。
- WindowManagerGlobal在一个进程中只有一个实例。
- WindowManagerGlobal在三个数组中统一管理整个进程中所有窗口的信息。这些信息包括控件、布局参数以及ViewRootImpl三个元素。
- 除了管理窗口的上述三个元素以外，WindowManagerGlobal将窗口的创建、销毁与布局更新等任务交给ViewRootImpl完成。



说明

在实际的应用开发过程中，有时会在logcat的输出中遇到有关WindowLeaked的异常输出。WindowLeaked异常发生于WindowManagerGlobal中，其原因是Activity在destroy之前没有销毁其附属窗口，如对话框、弹出菜单等。

如此看来，WindowManager的实现仍然是很轻量的。窗口的创建、销毁与布局更新都指向了一个组件：ViewRootImpl。

6.3 深入理解ViewRootImpl

ViewRootImpl实现了ViewParent接口，作为整个控件树的根部，它是控件树正常运作的动力所在，控件的测量、布局、绘制以及输入事件的派发处理都由ViewRootImpl触发。另一方面，它是 WindowManagerGlobal工作的实际实现者，因此它还需要负责与WMS 交互通信以调整窗口的位置大小，以及对来自WMS的事件（如窗口尺寸改变等）做出相应的处理。

本节将对ViewRootImpl的实现做深入探讨。

6.3.1 ViewRootImpl的创建及其重要的成员

ViewRootImpl创建于WindowManagerGlobal的addView () 方法中，而调用addView () 方法的线程就是此ViewRootImpl所掌控的控件树的UI 线程。ViewRootImpl的构造主要是初始化了一些重要的成员，事先对这些重要的成员有个初步的认识，这对随后探讨ViewRootImpl的工作原理有很大的帮助。其构造函数代码如下：

```
[ViewRootImpl.java-->ViewRootImpl.ViewRootImpl()]
public
ViewRootImpl(Context context, Display display) /* ① 从
WindowManagerGlobal中获取一个IWindowSession的实例。它是
```

ViewRootImpl和WMS 进行通信的代理 */mWindowSession = WindowManagerGlobal.getWindowSession(context.getMainLooper());// ②
保存参数display，在后面setView()调用中将会把窗口添加到这个Display
上mDisplay = display;CompatibilityInfoHolder cih =
display.getCompatibilityInfo();mCompatibilityInfo = cih != null ? cih : new
CompatibilityInfoHolder(); /* ③ 保存当前线程到mThread。这个赋值操
作体现了创建ViewRootImpl的线程如何成为UI主线程。在
ViewRootImpl处理来自控件树的请求时（如请求重新布局、请求重
绘、改变焦点等），会检查发起请求的thread与这个mThread是否相
同。倘若不同则会拒绝这个请求并抛出一个异常 */mThread =
Thread.currentThread();..... /* ④ mDirty用于收集窗口中的无效区域。所
谓无效区域是指由于数据或状态发生改变时而需要进行重绘 的区域。
举例说明，当应用程序修改了一个TextView的文字时，TextView会将自
己的区域标记为无效 区域，并通过view.invalidate()方法将这块区域添
加到这里的mDirty中。当下次绘制时，TextView便 可以将新的文字绘
制在这块区域上 */mDirty = new Rect();mTempRect = new
Rect();mVisRect = new Rect();/* ⑤ mWinFrame描述了当前窗口的位置和
尺寸。与WMS中的WindowState.mFrame保持一致 */mWinFrame = new
Rect();/* ⑥ 创建一个W类型的实例，W是IWindow.Stub的子类。即它将
在WMS中作为新窗口的ID，并接收来自 WMS的回调 */mWindow =
new W(this);...../* ⑦ 创建mAttachInfo。mAttachInfo是控件系统中很重

要的对象。它存储了当前控件树所贴附的窗口的各种有用信息，并且会派发给控件树中的每一个控件。这些控件会将这个对象保存在自己的mAttachInfo 变量中。mAttachInfo中所保存的信息有WindowSession、窗口的实例（即mWindow） 、ViewRootImpl 实例、窗口所属的Display、窗口的Surface以及窗口在屏幕上的位置等。所以，当需要在一个View中查询与当前窗口相关的信息时，非常值得在mAttachInfo中搜索一下 */ mAttachInfo = new View.AttachInfo(mWindowSession, mWindow, display, this, mHandler, this);/* ⑧ 创建FallbackEventHandler。这个类同PhoneWindowManger一样，定义在android.policy包中，其实现为PhoneFallbackEventHandler。FallbackEventHandler是一个处理未经任何人消费的输入事件的场所。在6.5.4节中将会介绍它 */ mFallbackEventHandler = PolicyManager.makeNewFallbackEventHandler(context);/* ⑨ 创建一个依附于当前线程，即主线程的Choreographer，用于通过VSYNC特性安排重绘行为 */ mChoreographer = Choreographer.getInstance();

在构造函数之外，还有另外两个重要的成员被直接初始化：

·mHandler，类型为ViewRootHandler，一个依附于创建ViewRootImpl的线程，即主线程上的，用于将某些必须在主线程进行的操作调度主线程中执行。mHandler与mChoreographer的同时存在看似有些重复，其实它们拥有明确不同的分工与意义。由于mChoreographer处理消息时具有

VSYNC特性，因此它主要用于处理与重绘相关的操作。但是由于mChoreographer需要等待VSYNC的垂直同步事件来触发对下一条消息的处理，因此它处理消息的即时性稍逊于mHandler。而mHandler的作用则是为了将发生在其他线程中的事件安排在主线程上执行。所谓发生在其他线程中的事件是指来自于WMS，由继承自IWindow.Stub的mWindow引发的回调。由于mWindow是一个Binder对象的Bn端，因此，这些回调发生在Binder的线程池中。而这些回调会影响到控件系统的重新测量、布局与绘制，因此需要此Handler将回调安排到主线程中。



说明

mHandler与mThread两个成员都是为了单线程模型而存在的。Android的UI操作不是线程安全的，而且很多操作也是建立在单线程的假设之上（如scheduleTraversals（））。采用单线程模型的目的是降低系统的复杂度，并且降低锁的开销。

·mSurface，类型为Surface。采用无参构造函数创建的一个Surface实例。mSurface此时是一个没有任何内容的空壳子，在WMS通过

`relayoutWindow ()` 为其分配一块Surface之前尚不能使用。

`mWinFrame`、`mPendingContentInset`、`mPendingVisibleInset`以及`mWidth`和`mHeight`。这几个成员存储了窗口布局相关的信息。其中`mWinFrame`、`mPendingContentInset`s、`mPendingVisibleInset`s与窗口在WMS中的`Frame`、`ContentInset`s、`VisibleInset`s是保持同步的。这是因为这三个成员不仅会作为`relayoutWindow ()` 的传出参数，而且`ViewRootImpl`在收到来自WMS的回调`IWindow.Stub.resize ()` 时，立即更新这三个成员的取值。因此这三个成员体现了窗口在WMS中的最新状态。

与`mWinFrame`记录了窗口在WMS中的尺寸有所不同的是，`mWidth/mHeight`记录了窗口在`ViewRootImpl`中的尺寸，二者在绝大多数情况下是相同的。当窗口在WMS中被重新布局而导致尺寸发生变化时，`mWinFrame`会首先被`IWindow.Stub.resize ()` 回调更新，此时`mWinFrame`便会与`mWidth/mHeight`产生差异。此时`ViewRootImpl`即可得知需要对控件树进行重新布局以适应新的窗口变化。在布局完成后，`mWidth/mHeight`会被赋值为`mWinFrame`中所保存的宽和高，二者重新统一。在随后分析`performTraversals ()` 方法时，读者将会看到这一处理。另外，与`mWidth/mHeight`类似，`ViewRootImpl`也保存了窗口的位置信息`Left/Top`以及`ContentInset`s/`VisibleInset`s供控件树查询，不过这4项信息被保存在`mAttachInfo`中。

ViewRootImpl在其构造函数中初始化了一系列成员变量，然而其创建过程仍未完成。仅在为其指定了一个控件树进行管理，并向WMS添加了一个新的窗口之后，ViewRootImpl承上启下的角色才算完全确立下来。因此需要进一步分析ViewRootImpl.setView()方法。

[ViewRootImpl.java-->ViewRootImpl.setView()] public void setView(View view, WindowManager.LayoutParams attrs, View panelParentView) {
synchronized (this) { if (mView == null) { // ① mView保存了控件树的根
mView = view; // ② mWindowAttributes保存了窗口所对应的
LayoutParams mWindowAttributes.copyFrom(attrs); /* 在添加窗口之
前，先通过requestLayout()方法在主线程上安排一次“遍历”。所谓“遍
历”是指 ViewRootImpl中的核心方法performTraversals()。这个方法实现
对控件树进行测量、布 局、向WMS申请修改窗口属性以及重绘的所有
工作。由于此“遍历”操作对于初次遍历做了一些 特殊处理，而来自
WMS通过mWindow发生的回调会导致一些属性发生变化，如窗口的尺
寸、Insets 以及窗口焦点等，从而有可能使得初次“遍历”的现场遭到破
坏。因此，需要在添加窗口之前，先 发送一个“遍历”消息到主线程。
在主线程中向主线程的Handler发送消息，如果使用得当，可以产生很
精妙的效果。例如本例中 可以实现如下的执行顺序：添加窗口→初次
遍历→处理来自WMS的回调 */ requestLayout(); /* ③ 初始化
mInputChannel。参考第5章，InputChannel是窗口接收来自
InputDispatcher的 输入事件的管道。注意，仅当窗口的属性

inputFeatures不含有INPUT_FEATURE_NO_INPUT_CHANNEL时才会创建InputChannel，否则mInputChannel为空，从而导致此窗口无法接收任何输入事件 */ if ((mWindowAttributes.inputFeatures & WindowManager.LayoutParams.INPUT_FEATURE_NO_INPUT_CHANNEL) == 0) { mInputChannel = new InputChannel(); } try { /* 将窗口添加到WMS中。完成这个操作之后，mWindow已经被添加到指定的Display中而且 mInputChannel（如果不为空）已经准备好接收事件。只是由于这个窗口没有进行过relayout()，因此它还没有有效的Surface可以进行绘制 */ res = mWindowSession.addToDisplay(mWindow, mSeq, mWindowAttributes, getHostVisibility(), mDisplay.getDisplayId(), mAttachInfo.mContentInsets, mInputChannel); } catch (RemoteException e) {.....} finally { } if (res < WindowManagerGlobal.ADD_OKAY) { // 错误处理。窗口添加失败的原因通常是权限问题、重复添加或者 token无效 } /* ④ 如果mInputChannel不为空，则创建 mInputEventReceiver，用于接收输入事件。注意第二个参数传递的是 Looper.myLooper()，即mInputEventReceiver将在主线程上触发 输入事件的读取与onInputEvent()。这是应用程序可以在onTouch()等事件响应中直接进行 UI操作等的根本原因 */ if (mInputChannel != null) { mInputEventReceiver = new WindowInputEventReceiver(mInputChannel, Looper.myLooper()); } /* ViewRootImpl将作为参数view的parent。所

以， ViewRootImpl可以从控件树中任何一个 控件开始， 通过回溯
getParent()的方法得到 */ view.assignParent(this); } } }

至此， ViewRootImpl所有重要的成员都已经初始化完毕， 新的窗口也
已经添加到WMS中。 ViewRootImpl的创建过程是由构造函数和setView
() 方法两个环节构成的。 其中构造函数主要进行成员的初始化，
setView () 则是创建窗口、 建立输入事件接收机制的场所。 同时， 触
发第一次“遍历”操作的消息已经发送给主线程，在随后的第一次“遍
历”完成后， ViewRootImpl将会完成对控件树的第一次测量、 布局，并
从WMS获取窗口的Surface以进行控件树的初次绘制工作。

在本节的最后， 通过图6-4对ViewRootImpl中的重要成员进行了分类整
理。



图6-4 ViewRootImpl中的主要成员

6.3.2 控件系统的心跳：performTraversals ()

ViewRootImpl在其创建过程中通过requestLayout () 向主线程发送了一条触发“遍历”操作的消息，“遍历”操作是指performTraversals () 方法。它的性质与WMS中的performLayoutAndPlaceSurfacesLocked () 类似，是一个包罗万象的方法。ViewRootImpl中接收的各种变化，如来自WMS的窗口属性变化、来自控件树的尺寸变化及重绘请求等都引发performTraversals () 的调用，并在其中完成处理。View类及其子类中的onMeasure () 、onLayout () 以及onDraw () 等回调也都是在performTraversals () 的执行过程中直接或间接地引发。也正是如此，一次次的performTraversals () 调用驱动着控件树有条不紊地工作，一旦此方法无法正常执行，整个控件树都将处于僵死状态。因此，performTraversals () 函数可谓是ViewRootImpl的心跳。

由于布局的相关工作是此方法中最主要的内容，为了简化分析，并突出此方法的工作流程，本节将以布局的相关工作为主线进行探讨。待完成这部分内容的分析之后，庞大的performTraversals () 方法将不再那么难以驯服，读者可以轻易地学习其他的工作。

1.performTraversals () 的工作阶段

`performTraversals ()` 是Android源代码中最庞大的方法之一，因此在正式探讨它的实现之前最好先将其划分为以下几个工作阶段作为指导。

- 预测量阶段。这是进入`performTraversals ()` 方法后的第一个阶段，它会对控件树进行第一次测量。测量结果可以通过`mView.getMeasuredWidth () /Height ()` 获得。在此阶段中将计算出控件树为显示其内容所需的尺寸，即期望的窗口尺寸。在这个阶段中，`View`及其子类的`onMeasure ()` 方法将会沿着控件树依次得到回调。
- 布局窗口阶段。根据预测量的结果，通过`IWindowSession.relayout ()` 方法向WMS请求调整窗口的尺寸等属性，这将引发WMS对窗口进行重新布局，并将布局结果返回给`ViewRootImpl`。
- 最终测量阶段。预测量的结果是控件树所期望的窗口尺寸。然而由于在WMS中影响窗口布局的因素很多（参考第4章），WMS不一定会将窗口准确地布局为控件树所要求的尺寸，而迫于WMS作为系统服务的强势地位，控件树不得不接受WMS的布局结果。因此在这个阶段，`performTraversals ()` 将以窗口的实际尺寸对控件进行最终测量。在这个阶段中，`View`及其子类的`onMeasure ()` 方法将会沿着控件树依次被回调。

- 布局控件树阶段。完成最终测量之后便可以对控件树进行布局。测量确定的是控件的尺寸，而布局则是确定控件的位置。在这个阶段中，View及其子类的onLayout（）方法将会被回调。
- 绘制阶段。这是performTraversals（）的最终阶段。确定控件的位置与尺寸后，便可以对控件树进行绘制。在这个阶段中，View及其子类的onDraw（）方法将会被回调。



说明

很多文章都倾向于将performTraversals（）的工作划分为测量、布局与绘制三个阶段。然而笔者认为如此划分隐藏了WMS在这个过程中的地位，并且没能体现出控件树对窗口尺寸的期望、WMS对窗口尺寸做出最终的确定，最后控件树以WMS给出的结果为准再次进行测量的协商过程。而这个协商过程充分体现了ViewRootImpl作为WMS与控件树的中间人的角色。

接下来将结合代码对上述5个阶段进行深入分析。

2. 预测量与测量原理

本节将探讨performTraversals () 以何种方式对控件树进行预测量，同时，本节也会对控件的测量过程与原理进行介绍。

(1) 预测量参数的候选

预测量也是一次完整的测量过程，它与最终测量的区别仅在于参数不同而已。实际的测量工作在View或其子类的onMeasure () 方法中完成，并且其测量结果需要受限于来自其父控件的指示。这个指示由onMeasure () 方法的两个参数进行传达：widthSpec与heightSpec。它们是被称为MeasureSpec的复合整型变量，用于指导控件对自身进行测量。它有两个分量，结构如图6-5所示。

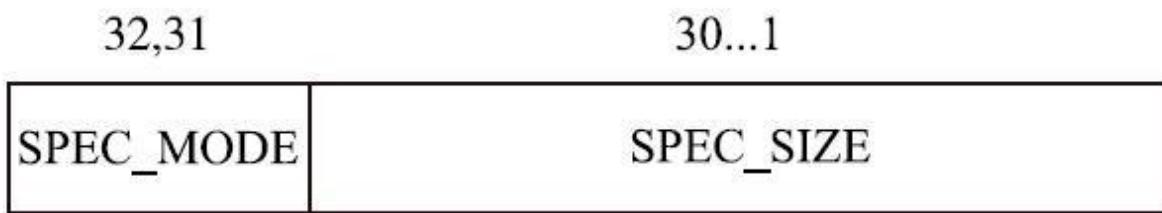


图6-5 MeasureSpec的结构

其1到30位给出了父控件建议尺寸。建议尺寸对测量结果的影响依SPEC_MODE的不同而不同。SPEC_MODE的取值取决于此控件的LayoutParams.width/height的设置，可以是如下三种值之一。

- MeasureSpec.UNSPECIFIED (0) : 表示控件在进行测量时，可以无视SPEC_SIZE的值。控件可以是它所期望的任意尺寸。

·MeasureSpec.EXACTLY (1) : 表示子控件必须为SPEC_SIZE所指定的尺寸。当控件的LayoutParams.width/height为一确定值，或者是MATCH_PARENT时，对应的MeasureSpec参数会使用这个SPEC_MODE。

·MeasureSpec.AT_MOST (2) : 表示子控件可以是它所期望的尺寸，但是不得大于SPEC_SIZE。当控件的LayoutParams.width/height为WRAP_CONTENT时，对应的MeasureSpec参数会使用这个SPEC_MODE。

Android提供了一个MeasureSpec类用于组合两个分量成为一个MeasureSpec，或者从MeasureSpec中分离任何一个分量。

那么ViewRootImpl会如何为控件树的根mView准备其MeasureSpec呢？

参考如下代码，注意desiredWindowWidth/Height的取值，它们将是SPEC_SIZE分量的候选。另外，这段代码分析中也解释了与测量无关，但是比较重要的代码段。

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void
performTraversals() { // 将mView保存在局部变量host中，以此提高对
mView的访问效率
final View host = mView;.....// 声明本阶段的主角，这
两个变量将是mView的SPEC_SIZE分量的候选
int
desiredWindowWidth;int desiredWindowHeight;.....Rect frame =
```

```
mWinFrame; // 如上一节所述， mWinFrame表示窗口的最新尺寸 if  
(mFirst) { /* mFirst表示这是第一次遍历， 此时窗口刚刚被添加到  
WMS， 此时窗口尚未进行relayout， 因此mWin- Frame中没有存储有效  
的窗口尺寸 */ if (lp.type ==  
WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL) { ..... // 为  
状态栏设置desiredWindowWidth/Height， 其取值是屏幕尺寸 } else { //  
① 第一次“遍历”的测量， 采用了应用可以使用的最大尺寸作为  
SPEC_SIZE的候选 DisplayMetrics packageMetrics =  
mView.getContext().getResources().getDisplayMetrics();  
desiredWindowWidth = packageMetrics.widthPixels; desiredWindowHeight  
= packageMetrics.heightPixels; } /* 由于这是第一次进行“遍历”， 控件树  
即将第一次被显示在窗口上， 因此接下来的代码填充了mAtt- achInfo中  
的一些字段， 然后通过mView发起dispatchAttachedToWindow()的调  
用， 之后每一个位于控件树中的控件都会回调onAttachedToWindow()  
*/ ..... } else { // ② 在非第一次遍历的情况下， 会采用窗口的最新尺寸作  
为SPEC_SIZE的候选 desiredWindowWidth = frame.width();  
desiredWindowHeight = frame.height(); /* 如果窗口的最新尺寸与  
ViewRootImpl中的现有尺寸不同， 说明WMS单方面改变了窗口的尺寸  
这将产生如下三个结果 */ if (desiredWindowWidth != mWidth ||  
desiredWindowHeight != mHeight) { // 需要进行完整的重绘以适应新的  
窗口尺寸 mFullRedrawNeeded = true; // 需要对控件树进行重新布局
```

mLayoutRequested = true; /* 控件树有可能拒绝接受新的窗口尺寸，比如在随后的预测量中给出不同于窗口尺寸的测量结果。产生这种情况时，就需要在窗口布局阶段尝试设置新的窗口尺寸 */

windowSizeMayChange = true; } }...../* 执行位于RunQueue中的回调。

RunQueue是ViewRootImpl的一个静态成员，就是说它是进程唯一的，并且可以在进程的任何位置访问RunQueue。在进行多线程任务时，开发者可以通过调用View.post()或View. postDelayed()方法将一个Runnable对象发送到主线程执行。这两个方法的原理是将Runnable 对象发送到ViewRootImpl的mHandler。当控件已经加入控件树时，可以通过AttachInfo轻易获取 这个Handler。而当控件没有位于控件树中时，则没有mAttachInfo可用，此时执行View.post()/PostDelay() 方法，Runnable将会被添加到这个RunQueue队列中。在这里，ViewRootImpl将会把RunQueue中的Runnable发送到mHandler，进而得到执行。所以无论控件是否显示在控件树中，View.post()/postDelay()方法都是可用的，除非当前进程中没有任何处于 活动状态的ViewRootImpl

*/getRunQueue().executeActions(attachInfo.mHandler);boolean

layoutRequested = mLayoutRequested && !mStopped; /* 仅当

layoutRequested为true时才进行预测量。 layoutRequested为true表示在进行“遍历”之前requestLayout()方法被调用过。 requestLayout()方法用于要求ViewRootImpl进行一次“遍历”并对控件树重新进行测量与布局 */if

(layoutRequested) { final Resources res =

mView.getContext().getResources(); if (mFirst) { // 确定控件树是否需要进入TouchMode，将在6.5.1节介绍 TouchMode } else { /* 检查WMS是否单方面改变了ContentInsets与VisibleInsets。注意对二者的处理的差异，ContentInsets描述了控件在布局时必须预留的空间，这样会影响控件树的布局，因此将insetsChanged标记为true，以此作为是否进行控件布局的条件之一。而VisibleInsets则描述了 被遮挡的空间，ViewRootImpl在进行绘制时，需要调整绘制位置以保证关键控件或区域，如正在 进行输入的TextView等不被遮挡，这样VisibleInsets的变化并不会导致重新布局，所以这里 仅仅是将VisibleInsets保存到 mAttachInfo中，以便绘制时使用 */ if

```
(!mPendingContentInsets.equals(mAttachInfo.mContentInsets)) {  
    insetsChanged = true; } if  
(!mPendingVisibleInsets.equals(mAttachInfo.mVisibleInsets)) {  
    mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets); } /* 当窗口的  
width或height被指定为WRAP_CONTENT时，表示这是一个悬浮窗口。  
此时会对 desiredWindowWidth/Height进行调整。在前面的代码中，这  
两个值被设置为窗口的 当前尺寸。而根据MeasureSpec的要求，测量结  
果不得大于SPEC_SIZE。然而，如果这个悬浮窗口需要更大的尺寸以  
完整显示其内容时，例如为AlertDialog设置了一个更长的消息 内容，  
如此取值将导致无法得到足够大的测量结果，从而导致内容无法完整  
显示。因此，对于此等类型的窗口，ViewRootImpl会调整
```

desiredWindowWidth/Height为此应用 可以使用的最大尺寸 */ if (lp.width == ViewGroup.LayoutParams.WRAP_CONTENT || lp.height == ViewGroup.LayoutParams.WRAP_CONTENT) { // 悬浮窗口的尺寸取决于测量结果。因此有可能需要向WMS申请改变窗口的尺寸
windowSizeMayChange = true; if (lp.type == WindowManager.LayoutParams.TYPE_STATUS_BAR_PANEL) { // } else { // ③ 设置悬浮窗口SPEC_SIZE的候选为应用可以使用的最大尺寸
DisplayMetrics packageMetrics = res.getDisplayMetrics();
desiredWindowWidth = packageMetrics.widthPixels; desiredWindowHeight = packageMetrics.heightPixels; } } // ④ 进行预测量。通过 measureHierarchy()方法对desiredWindowWidth/Height进行测量
windowSizeMayChange |= measureHierarchy(host, lp, res, desiredWindowWidth, desiredWindowHeight); } // 其他阶段的处理..... }

由此可知， 预测量时的SPEC_SIZE按照如下原则进行取值：

- 第一次“遍历”时， 使用应用可用的最大尺寸作为SPEC_SIZE的候选。
- 此窗口是一个悬浮窗口， 即LayoutParams.width/height其中之一被指定为WRAP_CONTENT时， 使用应用可用的最大尺寸作为SPEC_SIZE的候选。
- 在其他情况下， 使用窗口最新尺寸作为SPEC_SIZE的候选。

最后，通过`measureHierarchy ()` 方法进行测量。

(2) 测量协商

`measureHierarchy ()` 用于测量整个控件树。传入的参数 `desiredWindowWidth` 与 `desiredWindowHeight` 在前述代码中根据不同的情况做了精心的挑选。控件树本可以按照这两个参数完成测量，但是 `measureHierarchy ()` 有自己的考量，即如何将窗口布局得尽可能优雅。

这是针对将`LayoutParams.width`设置为`WRAP_CONTENT`的悬浮窗口而言。如前文所述，在设置为`WRAP_CONTENT`时，指定的 `desiredWindowWidth` 是应用可用的最大宽度，如此可能会产生如图6-6 左图所示的丑陋布局。这种情况较容易发生在`AlertDialog`中，当 `AlertDialog` 需要显示一条比较长的消息时，由于给予的宽度足够大，因此它有可能将这条消息以一行显示，并使得其窗口充满整个屏幕宽度，这种布局在横屏模式下尤为丑陋。

倘若能够对可用宽度进行适当限制，迫使`AlertDialog` 将消息换行显示，则产生的布局结果将会优雅得多，如图6-6右图所示。但是，倘若不分青红皂白地对宽度进行限制，当控件树真正需要足够的横向空间时，会导致内容无法显示完全，或者无法达到最佳的显示效果。例如，当一个悬浮窗口希望尽可能大地显示一张照片时就会出现这样的情况。

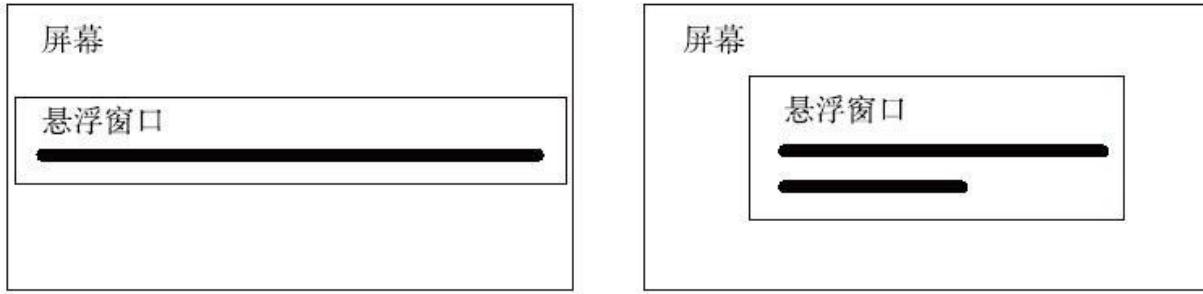


图6-6 丑陋的布局与优雅的布局

那么`measureHierarchy ()`如何解决这个问题呢？它采取了与控件树进行协商的办法，即先使用`measureHierarchy ()`所期望的宽度限制尝试对控件树进行测量，然后通过测量结果来检查控件树是否能够在此限制下满足其充分显示内容的要求。倘若无法满足，则`measureHierarchy ()`进行让步，放宽对宽度的限制，然后再次进行测量，再做检查。倘若仍不能满足则再度进行让步。

参考代码如下：

```
[ViewRootImpl.java-->ViewRootImpl.measureHierarchy()]
private boolean measureHierarchy(final View host, final WindowManager.LayoutParams lp,
final Resources res, final int desiredWindowWidth, final int
desiredWindowHeight) {int childWidthMeasureSpec; // 合成后的用于描述
宽度的MeasureSpecint childHeightMeasureSpec; // 合成后的用于描述高
度的MeasureSpecboolean windowHeightMayChange = false; // 表示测量结
果是否可能导致窗口的尺寸发生变化boolean goodMeasure = false; //
```

goodMeasure表示测量能否满足控件树充分显示内容的要求// 测量协商
仅发生在LayoutParams.width被指定为WRAP_CONTENT的情况下if
(lp.width == ViewGroup.LayoutParams.WRAP_CONTENT) { /* ① 第一次
协商。measureHierarchy()使用它期望的宽度限制进行测量。这一宽度
限制定义为一个系统资源。可以在
frameworks/base/core/res/res/values/config.xml找到它的定义 */
res.getValue(com.android.internal.R.dimen.config_prefDialogWidth,
mTmpValue, true); int baseSize = 0; // 宽度限制被存放在baseSize中 if
(mTmpValue.type == TypedValue.TYPE_DIMENSION) { baseSize =
(int)mTmpValue.getDimension(packageMetrics); } if (baseSize != 0 &&
desiredWindowWidth > baseSize) { // 使用getRootMeasureSpec()函数组合
SPEC_MODE与SPEC_SIZE为一个MeasureSpec childWidthMeasureSpec
= getRootMeasureSpec(baseSize, lp.width); childHeightMeasureSpec =
getRootMeasureSpec(desiredWindowHeight, lp.height); // ② 第一次测量。
由performMeasure()方法完成 performMeasure(childWidthMeasureSpec,
childHeightMeasureSpec); /* 控件树的测量结果可以通过mView的
getMeasuredWidthAndState()方法获取。如果控件 树对这个测量结果不
满意，则会在返回值中添加MEASURED_STATE_TOO_SMALL位 */ if
((host.getMeasuredWidthAndState()&View.MEASURED_STATE_TOO_S
MALL) == 0) { goodMeasure = true; // 控件树对测量结果满意，测量完
成 } else { // ③ 第二次协商。上次测量结果表明控件树认为

measureHierarchy()给予的宽度太小，在此适当地放宽对宽度的限制，使用最大宽度与期望宽度的中间值作为宽度限制 */ baseSize = (baseSize+desiredWindowWidth)/2; childWidthMeasureSpec = getRootMeasureSpec(baseSize, lp.width); // ④ 第二次测量 performMeasure(childWidthMeasureSpec, childHeightMeasureSpec); // 再次检查控件树是否满足此次测量 if ((host.getMeasuredWidthAndState()&View.MEASURED_STATE_TOO_SMALL) == 0) { goodMeasure = true; // 控件树对测量结果满意，测量完成 } } } if (!goodMeasure) { /* ⑤ 最终测量。当控件树对上述两次协商的结果都不满意时，measureHierarchy()放弃所有限制做最终测量。这一次将不再检查控件树是否满意，因为即便其不满意，measurehierarchy()也没有更多的空间供其使用了 */ childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width); childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height); performMeasure(childWidthMeasureSpec, childHeightMeasureSpec); /* 最后，如果测量结果与ViewRootImpl中当前的窗口尺寸不一致，则表明随后可能有必要进行窗口尺寸的调整 */ if (mWidth != host.getMeasuredWidth() || mHeight != host.getMeasuredHeight()) { windowHeightMayChange = true; } } // 返回窗口尺寸是否可能需要发生变化 return windowHeightMayChange; }

显然，对于非悬浮窗口，即当LayoutParams.width被设置为MATCH_PARENT时，不存在协商过程，直接使用给定的desiredWindowWidth/Height进行测量即可。而对于悬浮窗口，measureHierarchy () 可以连续进行两次让步。因而在最不利的情况下，在ViewRootImpl的一次“遍历”中，控件树需要进行三次测量，即控件树中的每一个View.onMeasure () 会被连续调用三次之多，如图6-7所示。所以相对于onLayout () ，onMeasure () 方法对性能的影响比较大。

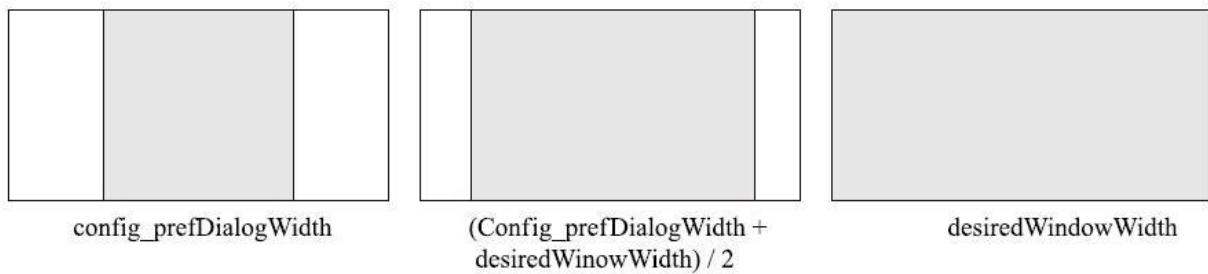


图6-7 协商测量的三次尝试

接下来通过performMeasure () 看控件树如何进行测量。

(3) 测量原理

performMeasure () 方法的实现非常简单，它直接调用mView.measure () 方法，将measureHierarchy () 给予的widthSpec与heightSpec交给mView。

看下View.measure () 方法的实现：

```
[View.java-->View.measure()] public final void measure(int widthMeasureSpec, int heightMeasureSpec) /* 仅当给予的MeasureSpec发生变化，或要求强制重新布局时，才会进行测量。所谓强制重新布局，是指当控件树中的一个子控件的内容发生变化时，需要进行重新测量和布局的情况下。在这种情况下，这个子控件的父控件（以及其父控件的父控件）所提供的MeasureSpec必定与上次测量时的值相同，因而导致从ViewRootImpl到这个控件的路径上的父控件的measure()方法无法得到执行，进而导致子控件无法重新测量其尺寸或布局。因此，当子控件因内容发生变化时，从子控件沿着控件树回溯到 ViewRootImpl，并依次调用沿途父控件的requestLayout()方法。这个方法会在mPrivateFlags 中加入标记PFLAG_FORCE_LAYOUT，从而使得这些父控件的measure()方法得以顺利执行，进而这个子控件有机会进行重新测量与布局。这便是强制重新布局的意义 */ if ((mPrivateFlags & PFLAG_FORCE_LAYOUT) == PFLAG_FORCE_LAYOUT || widthMeasureSpec != mOldWidthMeasureSpec || heightMeasureSpec != mOldHeightMeasureSpec) { /* ① 准备工作。从mPrivateFlags中将 PFLAG_MEASURED_DIMENSION_SET标记去除。 PFLAG_MEASURED_DIMENSION_SET标记用于检查控件在 onMeasure()方法中是否通过调用set- MeasuredDimension()存储测量结果 */ mPrivateFlags &= ~PFLAG_MEASURED_DIMENSION_SET; ..... /*
```

② 对本控件进行测量。每个View子类都需要重载这个方法以便正确地对自身进行测量。 View类的onMeasure()方法仅仅根据背景Drawable或style中设置的最小尺寸作为测量结果*/ onMeasure(widthMeasureSpec, heightMeasureSpec); /* ③ 检查onMeasure()的实现是否调用了setMeasuredDimension()。 setMeasuredDimension()会将PFLAG_MEASURED_DIMENSION_SET标记重新加入mPrivateFlags中。之所以做这样的检查，是由于onMeasure()的实现可能由开发者完成，而在Android看来，开发者是不可信的 */ if ((mPrivateFlags & PFLAG_MEASURED_DIMENSION_SET) != PFLAG_MEASURED_DIMENSION_SET) { throw new IllegalStateException("....."); } // ④ 将PFLAG_LAYOUT_REQUIRED标记加入mPrivateFlags。这一操作会对随后的布局操作放行 mPrivateFlags |= PFLAG_LAYOUT_REQUIRED; // 记录父控件给予的MeasureSpec，用以检查之后的测量操作是否有必要进行mOldWidthMeasureSpec = widthMeasureSpec;mOldHeightMeasureSpec = heightMeasureSpec; }

从这段代码可以看出，View.measure () 方法没有实现任何测量算法，它的作用在于引发onMeasure () 的调用，并对onMeasure () 行为的正确性进行检查。另外，在控件系统看来，一旦控件执行了测量操作，那么随后必须进行布局操作，因此在完成测量之后，将PFLAG_LAYOUT_REQUIRED标记加入mPrivateFlags，以便View.layout () 方法可以顺利进行。

onMeasure () 的结果通过setMeasuredDimension () 方法尽量保存。

setMeasuredDimension () 方法的实现如下：

```
[View.java-->View.setMeasuredDimension()] protected final void  
setMeasuredDimension(int measuredWidth, int measuredHeight) {/* ① 测  
量结果被分别保存在成员变量mMeasuredWidth与mMeasuredHeight中  
mMeasuredWidth = measuredWidth;mMeasuredHeight = measuredHeight;//  
② 向mPrivateFlags中添加PFLAG_MEASURED_DIMENSION_SET，以  
此证明onMeasure()保存了 测量结果mPrivateFlags |=  
PFLAG_MEASURED_DIMENSION_SET; }
```

其实现再简单不过。存储测量结果的两个变量可以通过

getMeasuredWidthAndState () 与getMeasuredHeightAndState () 两个方法获得，就像ViewRootImpl.measureHierarchy () 中所做的一样。此方法虽然简单，但需要注意，与MeasureSpec类似，测量结果不仅仅是一个尺寸，而是一个测量状态与尺寸的复合型变量。其0至30位表示了测量结果的尺寸，而31、32位则表示了控件对测量结果是否满意，即父控件给予的MeasureSpec是否可以使得控件完整地显示其内容。当控件对测量结果满意时，直接将尺寸传递给setMeasuredDimension () 即可，注意要保证31、32位为0。倘若对测量结果不满意，则使用View.MEASURED_STATE_TOO_SMALL|measuredSize作为参数传递给

`setMeasuredDimension ()` 以告知父控件对`MeasureSpec`进行可能的调整。

既然明白了`onMeasure ()` 的调用如何发起，以及它如何将测量结果告知父控件，那么`onMeasure ()` 方法应当如何实现呢？对非`ViewGroup`的控件来说其实现相对简单，只要按照`MeasureSpec`的原则如实计算其所需的尺寸即可。而对`ViewGroup`类型的控件来说情况则复杂得多，因为它不仅拥有自身需要显示的内容（如背景），它的子控件也是其需要测量的内容。因此它不仅需要计算自身显示内容所需的尺寸，还要考虑其一系列子控件的测量结果。为此它必须为每一个子控件准备`MeasureSpec`，并调用每一个子控件的`measure ()` 函数。

由于各种控件所实现的效果形形色色，开发者还可以根据需求自行开发新的控件，因此`onMeasure ()` 中的测量算法也会变化万千。不过 Android系统实现的角度仍能得到如下`onMeasure ()` 算法的一些实现原则：

- 控件在进行测量时，控件需要将它的`Padding`尺寸计算在内，因为`Padding`是其尺寸的一部分。

- `ViewGroup`在进行测量时，需要将子控件的`Margin`尺寸计算在内。因为子控件的`Margin`尺寸是父控件尺寸的一部分。

· ViewGroup为子控件准备MeasureSpec时，SPEC_MODE应取决于子控件的LayoutParams.width/height的取值。取值为MATCH_PARENT或一个确定的尺寸时应为EXACTLY，WRAP_CONTENT时应为AT_MOST。至于SPEC_SIZE，应理解为ViewGroup对子控件尺寸的限制，即ViewGroup按照其实现意图所允许子控件获得的最大尺寸。并且需要扣除子控件的Margin尺寸。

· 虽然测量的目的在于确定尺寸，与位置无关。但是子控件的位置是ViewGroup进行测量时必须首先考虑的。因为子控件的位置既决定了子控件可用的剩余尺寸，也决定了父控件的尺寸（当父控件的LayoutParams.width/height为WRAP_CONTENT时）。

· 在测量结果中添加MEASURED_STATE_TOO_SMALL需要做到实事求是。当一个方向上的空间不足以显示其内容时应考虑利用另一个方向上的空间，例如对文字进行换行处理，因为添加这个标记有可能导致父控件对其进行重新测量从而降低效率。

· 当子控件的测量结果中包含MEASURED_STATE_TOO_SMALL标记时，只要有可能，父控件就应当调整给予子控件的MeasureSpec，并进行重新测量。倘若没有调整的余地，父控件也应当将MEASURED_STATE_TOO_SMALL加入自己的测量结果中，让它的父控件尝试进行调整。

· ViewGroup在测量子控件时必须调用子控件的measure () 方法，而不能直接调用其onMeasure () 方法。直接调用onMeasure () 方法的最严重后果是子控件的PFLAG_LAYOUT_REQUIRED标识无法加入mPrivateFlag中，从而导致子控件无法进行布局。

综上所述，测量控件树的实质是测量控件树的根控件。完成控件树的测量之后，ViewRootImpl便得知控件树对窗口尺寸的需求。

(4) 确定是否需要改变窗口尺寸

接下面回到performTraversals () 方法。在ViewRootImpl.measureHierarchy () 执行完毕之后，ViewRootImpl了解了控件树所需的空间。于是便可确定是否需要改变窗口尺寸以便满足控件树的空间要求。前述的代码中多处设置windowSizeMayChange变量为true。window-SizeMayChange仅表示有可能需要改变窗口尺寸。而接下来的这段代码则用来确定窗口是否需要改变尺寸。

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()] private void  
performTraversals() {..... // 测量控件树的代码/* 标记mLayoutRequested  
为false。因此在此之后的代码中，倘若控件树中任何一个控件执行了  
requ- estLayout()，都会重新进行一次“遍历” */if (layoutRequested) {  
mLayoutRequested = false;}// 确定窗口是否确实需要改变尺寸boolean  
windowShouldResize = layoutRequested && windowSizeMayChange &&
```

```
((mWidth != host.getMeasuredWidth() || mHeight !=  
host.getMeasuredHeight()) || (lp.width ==  
ViewGroup.LayoutParams.WRAP_CONTENT && frame.width() <  
desiredWindowWidth && frame.width() != mWidth) || (lp.height ==  
ViewGroup.LayoutParams.WRAP_CONTENT && frame.height() <  
desiredWindowHeight && frame.height() != mHeight)); }
```

确定窗口尺寸是否确实需要改变的条件看起来比较复杂，这里进行一下总结，先介绍必要条件：

·layoutRequested为true，即ViewRootImpl.requestLayout () 方法被调用过。View中也有requestLayout () 方法。当控件内容发生变化从而需要调整其尺寸时，会调用其自身的requestLayout ()，并且此方法会沿着控件树向根部回溯，最终调用到ViewRootImpl.requestLayout ()，从而引发一次performTraversals () 调用。之所以这是一个必要条件，是因为performTraversals () 还有可能因为控件需要重绘时被调用。当控件仅需要重绘而不需要重新布局时（例如背景色或前景色发生变化时），会通过invalidate () 方法回溯到ViewRootImpl，此时不会通过requestLayout () 触发performTraversals () 调用，而是通过scheduleTraversals () 进行触发。在这种情况下layoutRequested为false，即表示窗口尺寸不需要发生变化。

·`windowSizeMayChange`为true，如前文所讨论的，这意味着WMS单方面改变了窗口尺寸而控件树的测量结果与这一尺寸有差异，或当前窗口为悬浮窗口，其控件树的测量结果将决定窗口的新尺寸。

在满足上述两个条件的情况下，以下两个条件满足其一：

- 测量结果与`ViewRootImpl`中所保存的当前尺寸有差异。
- 悬浮窗口的测量结果与窗口的最新尺寸有差异。



注意

`ViewRootImpl`对是否需要调整窗口尺寸的判断是非常小心的。第4章介绍WMS的布局子系统时曾经介绍过，调整窗口尺寸所必须调用的`performLayoutAndPlaceSurfacesLocked ()`函数会导致WMS对系统中的所有窗口进行重新布局，而且会引发至少一个动画帧渲染，其计算开销相当之大。因此`ViewRootImpl`仅在必要时才会惊动WMS。

至此，预测量阶段完成。

(5) 总结

这一阶段的工作内容是为了给后续阶段做参数的准备并且其中最重要的工作是对控件树的预测量，至此ViewRootImpl得知了控件树对窗口尺寸的要求。另外，这一阶段还准备了后续阶段所需的其他参数：

·viewVisibilityChanged。即View的可见性是否发生了变化。由于mView是窗口的内容，因此mView的可见性即窗口的可见性。当这一属性发生变化时，需要通过WMS改变窗口的可见性。

·LayoutParams。预测量阶段需要收集应用到LayoutParams的改动，这些改动一方面来自于WindowManager.updateViewLayout（），而另一方面则来自于控件树。以SystemUIVisibility为例，View.setSystemUIVisibility（）所修改的设置需要反映到LayoutParams中，而这些设置确却保存在控件自己的成员变量里。在预测量阶段会通过
ViewRootImpl.collectViewAttributes（）方法遍历控件树中的所有控件以收集这些设置，然后更新LayoutParams。

3.布局窗口与最终测量

接下来进入窗口布局阶段与最终测量阶段。窗口布局阶段以
reLayoutWindow（）方法为核心，并根据布局结果进行相应处理。而当
布局结果使得窗口尺寸发生改变时，最终测量阶段将会被执行。最终
测量使用performMeasure（）方法完成，因此其过程与预测量完全一

致，区别仅在于MeasureSpec参数的不同，所以本小节将一并探讨这两个阶段。另外，由于布局窗口会对Surface产生影响，这个阶段中会出现与硬件加速相关的代码。关于硬件加速的详细内容将在6.4节中进行讨论，而在本节中，读者仅需了解Surface的状态以及窗口尺寸对硬件加速的影响即可。

(1) 布局窗口的条件

如前文所述，窗口布局的开销很大，因此必须限制窗口布局阶段的执行。另外，倘若不需要进行窗口布局，则WMS不会在预测量之后修改窗口的尺寸。在这种情况下预测量的结果是有效的，因此不再需要进行最终测量。参考如下代码：

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void performTraversals() {..... // 预测量阶段的代码// 进行布局窗口的4个条件
if (mFirst || windowShouldResize || insetsChanged || viewVisibilityChanged
|| params != null) { ..... // 布局窗口与最终测量阶段代码} else { /* 倘若不
符合执行布局窗口的条件，则说明窗口的尺寸不需要进行调整。在这种情况下，有可能是 窗口的位置发生了变换，于是仅需将窗口的最新
位置保存到mAttachInfo中 */ final boolean windowMoved =
(attachInfo.mWindowLeft != frame.left || attachInfo.mWindowTop !=
frame.top); if (windowMoved) { ..... attachInfo.mWindowLeft = frame.left;
```

```
attachInfo.mWindowTop = frame.top; } } // 布局控件树阶段与绘制阶段代码..... }
```

在进行窗口布局时，以下4个条件满足其一即进入布局窗口阶段，此4个条件的意义如下：

- mFirst，即表示这是窗口创建以来的第一次“遍历”，此时窗口仅仅是添加到WMS中，但尚未进行窗口布局，并且没有有效的Surface进行内容绘制。因此必须进行窗口布局。
- windowShouldResize，正如在预测量阶段所述，当控件树的测量结果与窗口的当前尺寸有差异时，需要通过布局窗口阶段向WMS提出修改窗口尺寸的请求以满足控件树的要求。
- insetsChanged，表示WMS单方面改变了窗口的ContentInsets。这种情况一般发生在SystemUI的可见性发生了变化或输入法窗口弹出或关闭的情况下（请参考第4章）。严格来说，在这种情况下不需要重新进行窗口布局，只不过当ContentInsets发生变化时，需要执行一段渐变动画使窗口的内容过渡到新的ContentInsets下，而这段动画的启动动作发生在窗口布局阶段。稍后的代码分析中将介绍ContentInsets的影响，以及这段动画的实现。
- params != null，在进入performTraversals（）方法时，params变量被设置为null。当窗口的使用者通过 WindowManager.updateViewLayout（）

函数修改窗口的LayoutParams，或者在预测量阶段通过collectViewAttributes（）函数收集到的控件属性使得LayoutParams发生变化时，params将被设置为新的LayoutParams，此时需要将新的LayoutParams通过窗口布局更新到WMS中使其对窗口依照新的属性进行重新布局。

当上述4个条件全部不满足时，表示窗口布局是不必要的，而且窗口的尺寸也没有发生变化，因此仅需将窗口的新位置（如果发生了变化）更新到mAttachInfo中以供控件树查询。

（2）布局窗口前的准备工作

参考如下代码：

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void
performTraversals() {..... // 预测量阶段if /*引发布局窗口的4个条件*/ {
// ① 记录下在布局窗口之前是否拥有一块有效的Surface boolean
hadSurface = mSurface.isValid(); try { // ② 记录下在布局窗口之前Surface
的版本号 final int surfaceGenerationId = mSurface.getGenerationId(); // ③
通过relayoutWindow()方法布局窗口 relayoutResult =
relayoutWindow(params, viewVisibility, insetsPending); ..... // 处理布局窗
口产生的变化Part1 } catch { ..... } ..... // 处理布局窗口产生的变化Part1}
else { ..... }..... // 布局控件树阶段与绘制阶段 }
```

上述代码体现了在布局窗口前做两个准备工作：

·hadSurface，保存在布局窗口之前是否拥有一个有效的Surface。当窗口第一次进行“遍历”，或当前正处于不可见状态（mView的Visibility为INVISIBLE或GONE）时不存在有效的Surface。此变量可以在完成窗口布局后决定是否初始化或销毁用于绘制的HardwareRenderer。

·surfaceGenerationId，即Surface的版本号。每当WMS为窗口重新分配Surface时都会使得Surface的版本号增加。当完成窗口布局后Surface的版本号发生改变，在原Surface上建立的HardwareRenderer以及在其上进行的绘制都将无效，因此此变量用于决定窗口布局后是否需要将Surface重新设置到HardwareRenderer以及是否需要进行一次完整绘制。

可以看出，布局窗口的准备工作目的是保存布局前的状态，以便在布局后判断状态变化并做出相应处理。其实Surface两个状态的存储只是准备工作的一部分，有些用于判断状态变化的属性已经隐式地准备好了，它们有mWidth/mHeight、mAttachInfo.mLeft/mTop/mContentInsets/mVisibleInsets等。在布局完成后，它们将同hadSurface与surfaceGenerationId一起决定布局后的工作。

(3) 布局窗口

ViewRootImpl使用relayoutWindow（）方法进行窗口布局，参考以下代码来实现。

```
[ViewRootImpl.java-->ViewRootImpl.relayoutWindow()]
private int
relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,
boolean insetsPending) throws RemoteException {.....int relayoutResult =
mWindowSession.relayout( mWindow, mSeq, params, (int)
(mView.getMeasuredWidth() * appScale + 0.5f), (int)
(mView.getMeasuredHeight() * appScale + 0.5f), viewVisibility,
insetsPending ? WindowManagerGlobal.RELAYOUT_INSETS_PENDING
: 0, mWinFrame, mPendingContentInsets, mPendingVisibleInsets,
mPendingConfiguration, mSurface);.....return relayoutResult; }
```

relayoutWindow () 是IWindowSession.relayout () ，即 WMW.relayoutWindow () 的包装方法。它将窗口的LayoutParams、预测量时的结果以及mView的可见性作为输入，并将mWinFrame、mPendingContentInsets/VisibleInsets、mSurface作为输出。



说明

relayoutWindow () 并没有直接将预测量的结果交给WMS，而是乘以了appScale这个系数。appScale用于在兼容模式下显示一个窗口。当窗

口在设备的屏幕尺寸下显示异常时，Android会尝试使用兼容尺寸显示它（例如 320×480 ），此时测量与布局控件树都将以此兼容尺寸为准。为了充分利用屏幕，或避免窗口内容显示在屏幕外，Android计算了用以使兼容尺寸变换到屏幕尺寸的一个缩放系数，即appScale。这时窗口测量、控件树的布局都将以兼容尺寸进行以保证布局的正确性，而生成Surface，在绘制过程中将会使用appScale进行缩放，以保证最终显示的内容能够充分利用屏幕。

另外，传出参数mPendingConfiguration在之前的章节中并没有做过详细介绍。作为一个Configuration类型的实例，其意义是WMS给予窗口的当前配置。其中的字段描述了设备当前的语言、屏幕尺寸、输入方式（触屏或键盘）、UI模式（夜间模式、车载模式等）、dpi等。其中WMS有可能更改的最常用的字段是orientation，即屏幕方向。

(4) 布局窗口后的处理——Insets

relayoutWindow () 方法完成布局窗口后，回到performTraversals () 方法，对布局结果所导致的变化进行处理，首先是ContentInsets和VisibleInsets。参考如下代码：

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()] private void  
performTraversals() { ..... // ① 对比布局结果检查Insets是否发生了变化  
contentInsetsChanged = !mPendingContentInsets.equals(
```

```
mAttachInfo.mContentInsets); visibleInsetsChanged =  
!mPendingVisibleInsets.equals( mAttachInfo.mVisibleInsets); // 如果  
ContentInsets发生变化 if (contentInsetsChanged) { /* 启动过渡动画以避  
免内容发生突兀的抖动。其条件非常多，总结一下： 1> 布局窗口之前  
mWidth/mHeight有效，即之前曾经完成过布局，即布局有效，此时可  
以依照 Insets发生变化前的布局进行绘制。 2> systemUIVisibility没有指  
定要求隐藏状态栏或导航栏。因为当指定了此类systemUIVisi- bility  
后，控件树布局时将不会考虑ContentInsets而是充满屏幕。 3> 拥有有  
效的Surface，这个条件是不言而喻的。 4> 此窗口采用硬件加速方式进行  
绘制，并且其HardwareRenderer处于有效状态。因为这一过渡动画是  
以硬件加速方式实现的 */ 5> 窗口的Surface不得支持透明度。因为  
Android当前的硬件实现不支持在支持透明度的Sur- face上进行透明度  
变换。而这个过渡动画正是一个透明度动画 */ /*启动一个透明度动  
画，使得ContentInsets发生变化时产生的画面移位不那么突兀。在介绍  
硬件 加速绘制之后再讨论这一动画的细节 */ ..... // ② 将最新的  
ContentInsets保存到mAttachInfo中
```

```
mAttachInfo.mContentInsets.set(mPendingContentInsets); } if  
(contentInsetsChanged || mLasteSystemUiVisibility !=  
mAttachInfo.mSystemUiVisibility || mFitSystemWindowsRequested) {  
mLastSystemUiVisibility = mAttachInfo.mSystemUiVisibility;  
mFitSystemWindowsRequested = false;
```

```
mFitSystemWindowsInsets.set(mAttachInfo.mContentInsets); /* ③ 要求  
mView及其子控件适应这一ContentInsets。 View.fitSystemWindow()将会  
把ContentInsets作为其Padding属性保存下来。 Padding是指控件的边界  
到其内容边界的距离。在测量布局及绘制时需要将Pandding属性 计算  
在内 */ host.fitSystemWindows(mFitSystemWindowsInsets); } /* ④ 如果  
VisibleInsets发生变化，将其保存到mAttachInfo中 如前所述，  
VisibleInsets不影响测量与布局，而仅仅影响绘制时的偏移，因此除了  
将其保存下来 供绘制时使用以外无须进行其他操作 */ if  
(visibleInsetsChanged) {  
    mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets); }..... }
```

可见，当ContentInsets发生变化时，会进行如下动作：

- 启动一个过渡动画以避免突兀的画面位移。参考图6-8，当状态栏被隐藏时窗口顶部的ContentInsets为0，内容布局在屏幕的顶部。当状态栏重新显示时窗口顶部的ContentInsets为状态栏的高度，此时内容将会向下错位，布局在状态栏的底部。在用户看来，ContentInsets变化所导致的是窗口内容的突兀抖动，因此ViewRootImpl通过一个过渡动画使得画面从左图以渐变的方式过渡到右图以减轻这种突兀感。这一动画的渲染过程将在6.4节介绍。

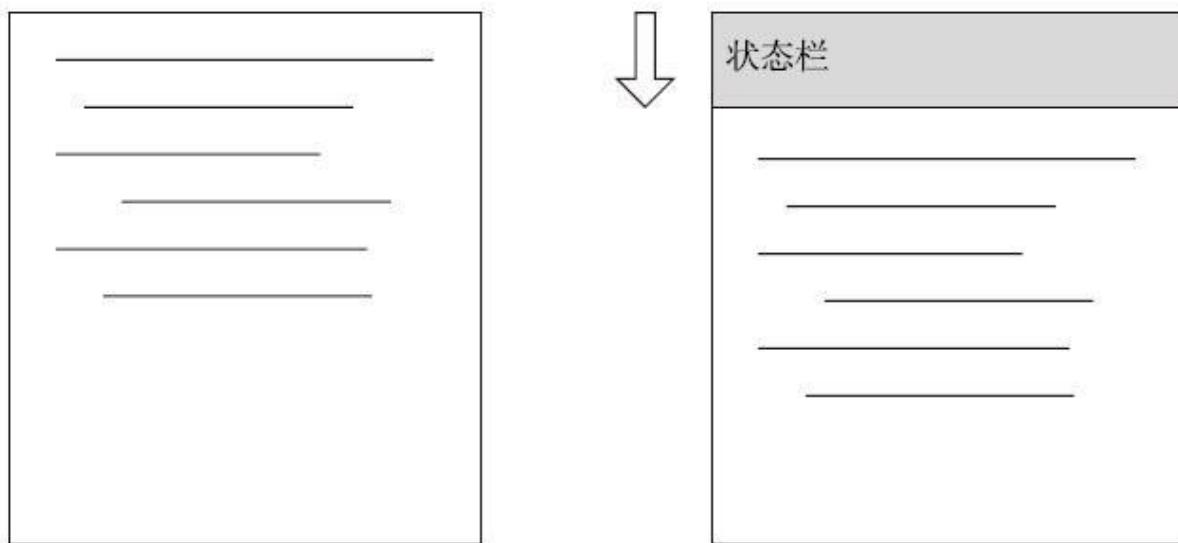


图6-8 ContentInsets发生变化时所导致的内容错位

- 保存新的ContentInsets到mAttachInfo。
- 通过执行mView.fitSystemWindows () 使控件树应用新的ContentInsets。其实质是设置控件的Padding属性以便在其进行测量、布局与绘制时让出ContentInsets所指定的空间。从此方法在ViewGroup中的实现可以看出它与onMeasure () 一样会沿着控件树遍历给所有的控件。不过在Activity、Dialog等由Android辅助添加的窗口中，此操作会被其控件树的根控件DecorView阻断，即DecorView完成其Padding设置后便不会再将此操作传递给其子控件。其合理性在于DecorView作为整个控件树的根，只要它设置好Padding值并在测量、布局以及绘制时让出ContentInsets指定的空间，其子控件必然会位于ContentInsets之外，因此子控件不需要对ContentInsets进行处理。当使用

WindowManager.addView () 方法添加一个控件作为窗口时，可以调用此控件的makeOptionalFitsSystemWindows () 方法使此控件拥有DecorView的这一特性。

而当VisibleInsets发生变化时则简单得多，仅需保存新的值即可。在绘制前会根据此值计算绘制偏移，以保证关键控件或区域位于可视区域之内。

(5) 布局窗口后的处理——Surface

在完成Insets的处理之后，便会处理Surface在窗口布局过程中的变化。这里将会用到在准备工作中所保存的状态hadSurface与surfaceGenerationId。

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void
performTraversals() {.....if (!hadSurface) { if (mSurface.isValid()) { /* ① 布
局窗口之前没有有效的Surface，而布局窗口之后有了。当此窗口启用
硬件加速时(mHardwareRenderer != null)，将使用新的Surface初始化用
于硬件渲染的HardwareRenderer。第一次“遍历”或窗口从不可见变为可
见时符合此种情况 */
..... if (mAttachInfo.mHardwareRenderer != null) {
try { hwInitialized = mAttachInfo.mHardwareRenderer.initialize(
mHolder.getSurface()); } catch (Surface.OutOfResourcesException e) {.....}
} } } else if (!mSurface.isValid()) { ..... /* ② 布局窗口之前拥有有效的
```

Surface，但布局窗口之后没有了。当窗口启用硬件加速时则销毁当前使用的HardwareRenderer，因为不需要再进行绘制。当窗口从可见变为不可见或窗口被移除时符合此种情况 */ if
(mAttachInfo.mHardwareRenderer != null &&
mAttachInfo.mHardwareRenderer.isEnabled()) {
mAttachInfo.mHardwareRenderer.destroy(true); } } else if
(surfaceGenerationId != mSurface.getGenerationId() && mSurfaceHolder
== null && mAttachInfo.mHardwareRenderer != null) { /* ③ 经过窗
口布局后，Surface发生了改变。底层的Surface被置换了，此时需要将
Surface重新设置给HardwareRenderer以将其与底层新的 Surface进行绑
定 */ try {
mAttachInfo.mHardwareRenderer.updateSurface(mHolder.getSurface()); }
catch (Surface.OutOfResourcesException e) {.....} }

显然，处理Surface的变化是为了硬件加速而服务的。其原因在于以软
件方式进行绘制时可以通过Surface.lockCanvas () 函数直接获得，因
此仅需在绘制前判断一下Surface.isValid () 决定是否绘制即可。而在
硬件加速绘制的情况下，将绘制过程委托给HardwareRenderer并且需要
将其与一个有效的Surface进行绑定。因此每当Surface的状态变换都需
要通知HardwareRenderer。软件与硬件加速的绘制原理将在6.4节中介
绍。

另外，代码中的mHolder是一个空的SurfaceHolder子类的实例，其getSurface () 方法所返回的Surface就是mSurface。



说明

在这段代码之后，还有一段类似的代码处理Surface状态的变化，并将状态变化通过一个SurfaceHolder汇报给mSurfaceHolderCallback。这段代码为了支持一种叫作Native-Activity的机制而实现的。NativeActivity是一类Activity，其逻辑实现由C++完成并由NDK进行编译。由于NativeActivity可以直接访问Surface，ViewRootImpl需要通过这段代码通知其Surface的状态变化。

(6) 布局窗口后的处理——更新尺寸

接下来，ViewRootImpl会保存最新的窗口位置与尺寸，同时，如果窗口采用了硬件加速，则更新其坐标系以适应新的窗口尺寸。

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()] private void  
performTraversals() {.....// ① 保存窗口的位置与尺寸信息  
attachInfo.mWindowLeft = frame.left;attachInfo.mWindowTop =
```

```
frame.top;if (mWidth != frame.width() || mHeight != frame.height()) {  
    mWidth = frame.width(); mHeight = frame.height();}...../* ② 更新尺寸信  
息到HardwareRenderer。 HardwareRenderer的setup()方法将会使用此宽  
高信息 来设置其ViewPort,这样便可以在Surface上建立  
(0,0,mWidth,mHeight)的坐标系 */if (mAttachInfo.mHardwareRenderer !=  
null && mAttachInfo.mHardwareRenderer.isEnabled()) { if (hwInitialized ||  
windowShouldResize || mWidth !=  
mAttachInfo.mHardwareRenderer.getWidth() || mHeight !=  
mAttachInfo.mHardwareRenderer.getHeight()) {  
    mAttachInfo.mHardwareRenderer.setup(mWidth, mHeight); if  
(!hwInitialized) {  
        mAttachInfo.mHardwareRenderer.invalidate(mHolder.getSurface());  
        mFullRedrawNeeded = true; } }..... }
```

在最终的窗口位置与尺寸得到更新之后，便进入下一阶段——窗口的最终测量。

(7) 最终测量

最终测量阶段与预测量阶段一样使用预测量阶段所介绍的 performMeasure () 进行，而区别在于参数不同。预测量使用了屏幕的可用空间或窗口的当前尺寸作为候选，使用measureHierarchy () 方法以协商的方式确定MeasureSpec参数，并且其测量结果体现了控件树所

期望的窗口尺寸。在窗口布局时这一期望尺寸交给WMS以期能将窗口重新布局为这一尺寸，然而由于WMS布局窗口时所考虑的因素很多，这一期望不一定如愿。在最终测量阶段，控件树将被迫接受WMS的布局结果，以最新的窗口尺寸作为MeasureSpec参数进行测量。这一测量结果将是随后布局阶段中设置控件边界时的依据。参考这一阶段的代码：

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
private void performTraversals() {.....if (!mStopped) { // 窗口布局还会影响另一个状态的变化：TouchMode，其相关内容将在6.5节中介绍
    boolean focusChangedDueToTouchMode = ensureTouchModeLocally(
        (relayoutResult& WindowManagerGlobal.RELAYOUT_RES_IN_TOUCH_MODE) != 0); /* ① 进行最终测量的条件：TouchMode发生变化；最新的窗口尺寸不符合预测量的结果；Content- Insets发生变化（导致Padding发生变化） */
    if (focusChangedDueToTouchMode || mWidth != host.getMeasuredWidth() || mHeight != host.getMeasuredHeight() ||
        contentInsetsChanged) { // ② 最终测量的参数为窗口的最新尺寸
        int childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);
        int childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height); // ③ 使用与预测量相同的performMeasure()方法进行
        performMeasure(childWidthMeasureSpec, childHeightMeasureSpec); .....
    } }..... }
```

之所以最终测量使用performMeasure () 而不是measureHierarchy () ，是因为measure-Hierarchy () 包含协商的算法，以期确定最佳的测量尺寸。而在最终测量中，窗口尺寸已然确定下来，是没有协商余地的。

(8) 总结

布局窗口与最终测量两个阶段至此便分析完毕。布局窗口阶段得以进行的原因是控件系统有修改窗口属性的需求，如第一次“遍历”需要确定窗口的尺寸以及一块Surface，预测量结果与窗口当前尺寸不一致需要进行窗口尺寸更改，mView可见性发生变化需要将窗口隐藏或显示，LayoutParams发生变化需要WMS以新的参数进行重新布局。而最终测量阶段得以进行的原因是窗口布局阶段确定的窗口尺寸与控件树的期望尺寸不一致，控件树需要对窗口尺寸进行妥协。

完成这两个阶段之后，performTraversals () 中剩余的变数已所剩无几，窗口的尺寸、控件树中控件的尺寸都已最终确定。接下来便是控件树的布局与绘制了。

4. 布局控件树阶段

经过前面的测量，控件树中的控件对于自己的尺寸显然已经了然于胸，而且父控件对于子控件的位置也有了眉目（因为为子控件准备MeasureSpec时有可能需要计算子控件的位置）。布局阶段将会把测量

结果付诸行动，即把测量结果转化为控件的实际位置与尺寸。控件的实际位置与尺寸由View的mLeft、mTop、mRight以及mBottom 4个成员变量存储的坐标值来表示。因此，控件树的布局过程就是根据测量结果为每一个控件设置这4个成员变量的过程。



注意

必须时刻注意一个事实：mLeft、mTop、mRight以及mBottom这些坐标值是相对于其父控件的左上角的距离，也就是说这些坐标值是以父控件左上角为坐标原点进行计算的。另一种说法是这些坐标位于父控件的坐标系中。倘若需要获取控件在窗口坐标系中的位置可以使用View.getLocationInWindow () 方法，相应，也可以通过View.getLocationOnScreen () 方法获取控件在屏幕坐标系下的位置。这两个方法的实现原理是一个沿着控件树向根部进行递归调用，其递归过程可以简单总结为控件在窗口中的位置等于其在父窗口中的位置加上父窗口在窗口中的位置。

参考如下代码：

[ViewRootImpl.java-->ViewRootImpl.performTraversals()]

```
private void performTraversals() {..... // 前序阶段的代码// ① 布局控件树阶段的条件是layoutRequestedfinal boolean didLayout = layoutRequested && !mStopped;if (didLayout) { // ② 通过performLayout()方法进行控件树的布局 performLayout(); /* ③ 如果有必要，计算窗口的透明区域，并将此透明区域设置给WMS。计算透明区域的条件是mView 的mPrivateFlags 中存在一个特定的标记。 */ if ((host.mPrivateFlags & View.PFLAG_REQUEST_TRANSPARENT_REGIONS) != 0) { host.getLocationInWindow(mTmpLocation); // 透明区域被初始化为整个 mView的区域 mTransparentRegion.set(mTmpLocation[0], mTmpLocation[1], mTmpLocation[0] + host.mRight - host.mLeft, mTmpLocation[1] + host.mBottom - host.mTop); /* 通过 gatherTransparentRegion () 遍历控件树中的每一个控件，倘若控件有 内容需要绘制，则会将其所在区域从mTransparentRegion中剪除 */ host.gatherTransparentRegion(mTransparentRegion); /* mTransparentRegion目前位于窗口坐标系中，mTranslator将此区域映射 到屏幕坐标系中。这么做的原因是WMS管理窗口是在屏幕坐标系中进 行的 */ if (mTranslator != null) { mTranslator.translateRegionInWindowToScreen(mTransparentRegion); } // 将透明区域设置到WMS if (!mTransparentRegion.equals(mPreviousTransparentRegion)) {
```

```
mPreviousTransparentRegion.set(mTransparentRegion); try {  
    mWindowSession.setTransparentRegion(mWindow, mTransparentRegion);  
} catch (RemoteException e) {} } } }..... // 绘制阶段的代码 }
```

布局控件树的条件并不像布局窗口那么严格。只要layoutRequested为true，即调用过requestLayout () 方法即可。requestLayout () 的用途在于当一个控件因为某种原因（如内容的尺寸发生变化）而需要调整自身尺寸时，向ViewRootImpl申请进行一次新的“遍历”以便使此控件得到一次新的测量布局与绘制。所以，只要requestLayout () 被调用则布局控件树阶段一定会执行。而requestLayout () 是否引发布局窗口阶段则取决于前述的4个条件是否满足。

为什么当一个控件调整尺寸时需要通过requestLayout () 使ViewRootImpl对整个控件树都做同样的事情呢？从之前阶段的分析可知，一个控件的测量结果可能直接影响控件树上各个父控件的测量结果，甚至是窗口的布局。所以为了能够完整地处理一个控件的变化所产生的影响，ViewRootImpl都会将整个过程从头来一遍。这可能会引发对运行效率的担心，不过不用担心，requestLayout () 所调用的用于引发一次“遍历”的scheduleTraversals () 会检查是否在主线程上已经安排了一次“遍历”。因此，倘若在一段代码中一次性地调用10个TextView的setText () 函数可能会导致requestLayout () 被调用10次，而scheduleTraversals () 的检查则会保证随后仅执行一次“遍历”。

布局控件树阶段主要做了两件事情：

- 进行控件树布局。
- 设置窗口的透明区域。

(1) 控件树布局

ViewRootImpl使用performLayout () 方法进行控件树布局，参考以下代码实现：

```
[ViewRootImpl.java-->ViewRootImpl.performLayout()]
private void performLayout() {.....final View host = mView;// try { // 调用
mView.layout()方法启动布局 host.layout(0, 0, host.getMeasuredWidth(),
host.getMeasuredHeight());} finally {.....} }
```

此方法的实现非常简单，直接调用mView.layout () 方法：

```
[View.java-->View.layout()]
public void layout(int l, int t, int r, int b) {// 保
存原始坐标int oldL = mLeft;int oldT = mTop;int oldB = mBottom;int oldR
= mRight;// ① setFrame()方法将l、t、r、b分别设置到mLeft、mTop、
mRight与mBottomboolean changed = setFrame(l, t, r, b);/* 是否还记得
PFLAG_LAYOUT_REQUIRED标记？它在View.measure()方法中被添加
到mPrivateFlags。按照常理来说，当此控件的布局没有发生改变时是
没有必要继续对子控件进行布局的，而这个标记则会将其放行，以保
```

证真正需要布局的子控件得到布局 */if (changed || (mPrivateFlags & PFLAG_LAYOUT_REQUIRED) == PFLAG_LAYOUT_REQUIRED) { /*
② 执行onLayout()。如果这是一个ViewGroup， onLayout()中需要依次调用子控件的lay- out()方法 */ onLayout(changed, l, t, r, b); // 清除 PFLAG_LAYOUT_REQUIRED标记 mPrivateFlags &= ~PFLAG_LAYOUT_REQUIRED; /* ③ 通知每一个对此控件的布局变化感兴趣的监听者 ListenerInfo li = mListenerInfo; if (li != null && li.mOnLayoutChangeListeners != null) { ArrayList listenersCopy = (ArrayList) li.mOnLayoutChangeListeners.clone(); int numListeners = listenersCopy.size(); for (int i = 0; i < numListeners; ++i) { listenersCopy.get(i) .onLayoutChange(this, l, t, r, b, oldL, oldT, oldR, oldB); } } } }

Layout () 方法主要做了三件事情：

- 通过setFrame () 设置布局的4个坐标。
- 调用onLayout () 方法，使子类得到布局变更的通知。如果此类是一个ViewGroup，则需要在onLayout () 方法中依次调用每一个子控件的layout () 方法使其得到布局。切记不能调用子控件的onLayout () 方法，这会导致子控件没有机会调用setFrame () ，从而使得此控件的坐标信息无法得到更新。

·通知每一个对此控件的布局变化感兴趣的监听者。可以通过调用View.addOnLayoutChangeListener()加入对此控件的监听。



注意

通知监听者的代码与控件树布局的核心逻辑无关，仍然将其帖附在这里的原因是由于它的实现很值得开发者借鉴。注意到layout()方法必定会在主线程中被performTraversals()调用，而addOnLayoutChangeListener()没有限定调用线程，却没有增加任何同步锁的保护。那么如何保证两个方法对mListenerInfo访问的同步呢？layout()的策略是将mListenerInfo通过clone()做一份拷贝，然后遍历这份拷贝，从而避免遍历过程中mListenerInfo发生变化而导致的越界。ArrayList内部也采用类似的方法来保证线程安全。

另外，既然已经有onLayout()方法监听布局的变化，为什么还需要监听者呢？onLayout()有它的局限性，即只能在类内部访问，因此它更适合做类内部的监听与处理。而监听者则给予类外部的对象监听其内部状态变化的能力，二者并不重复。

对比测量与布局两个过程有助于加深对它们的理解。

·测量确定的是控件的尺寸，并在一定程度上确定了子控件的位置。而

布局则是针对测量结果来实施，并最终确定子控件的位置。

·测量结果对布局过程没有约束力。虽说子控件在onMeasure () 方法中计算出了自己应有的尺寸，但是由于layout () 方法是由父控件调用，因此控件的位置尺寸的最终决定权在父控件手中，测量结果仅仅是一个参考。

·一般来说，子控件的测量结果影响父控件的测量结果，因此测量过程是后根遍历。而父控件的布局结果影响子控件的布局结果（例如位置），所以布局过程是先根遍历。

完成performLayout () 调用之后控件树的所有控件都已经确定了其最终位置，只等绘制了。

(2) 窗口透明区域

布局阶段另一个工作是计算并设置窗口的透明区域，这一功能主要是为SurfaceView服务。设想一个视频播放器的窗口，它包含一系列的控制按钮，位于主窗口上，而其进行视频内容渲染的SurfaceView所建立的子窗口则位于主窗口之下（参考第4章）。为了保证负责显示视频的

子窗口能够透过主窗口显示出来，Android引入了窗口透明区域的机制。

所谓的透明区域是指Surface上的一块特定区域，在SurfaceFlinger进行混层时，Surface上的这个块区域将会被忽略，就好似在Surface上切下一个洞一般，如图6-9所示。

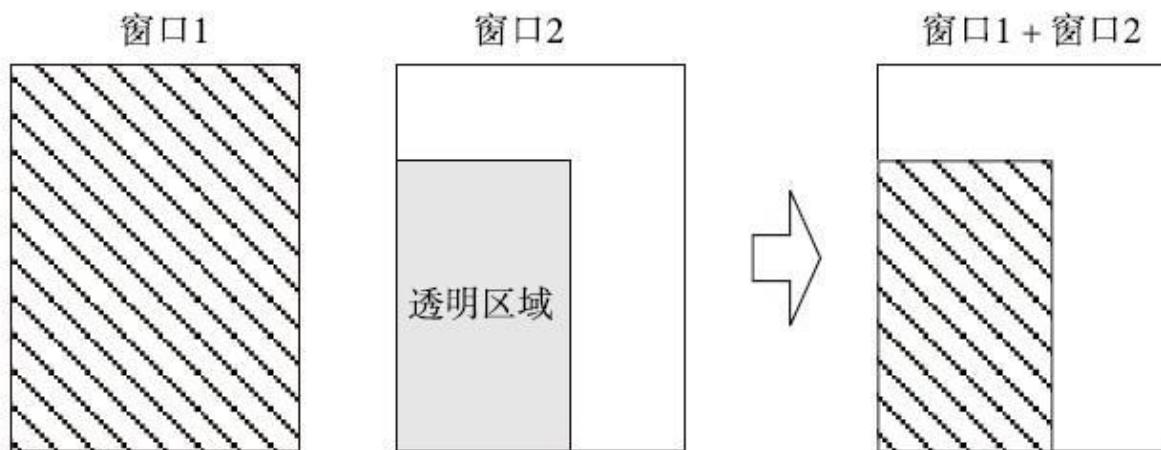


图6-9 带有透明区域的窗口与其他窗口合成后的效果

当控件树中存在SurfaceView时，它会通过调用 ViewParent.requestTransparentRegion () 方法启用这一机制。这一方法的调用会沿着控件树回溯到ViewRootImpl，并沿途将 PFLAG_REQUEST_TRANSPARENT_REGIONS 标记加入父控件的 mPrivateFlags 字段中。此标记会导致ViewRootImpl完成控件树的布局后将进行透明区域的计算与设置。

透明区域的计算由View.gatherTransparentRegion（）方法完成。透明区域的计算采用了挖洞法，及默认整个窗口都是透明区域，在gatherTransparentRegion（）遍历到一个控件时，如果这个此控件有内容需要绘制，则将其所在的区域从当前透明区域中删除，就好似在纸上裁出一个洞一样。当遍历完成后，剩余的区域就是最终的透明区域。

这个透明区域将会被设置到WMS中，进而被WMS设置给SurfaceFlinger。SurfaceFlinger在进行Surface的混合时，本窗口的透明区域部分将被忽略，从而用户能够透过这部分区域看到后面窗口（如SurfaceView的窗口）的内容。



注意

作为透明区域的主要服务对象，在gatherTransparentRegion（）中SurfaceView与其他类型控件的做法正好相反。SurfaceView会将其区域并到当前透明区域中。因此，先于SurfaceView被遍历的控件所在的区域有可能被SurfaceView所设置的透明区域覆盖，此时这些控件被覆盖

的区域将不会被SurfaceFlinger渲染。读者可以通过对比View类与SurfaceView类的gatherTransparentRegion () 方法实现的差异加深对透明区域的理解。

5.绘制阶段

经历了前述4个阶段，每一个控件都已经确定好了自己的尺寸与位置，接下来就是最终的绘制阶段。参考以下代码：

```
[ViewRootImpl.java-->ViewRootImpl.performTraversals()]
public void
performTraversals () {..... // 前4个阶段的代码
boolean skipDraw = false;
if (mFirst) { // 第一次“遍历”时，会进行第一次焦点控件的计算。在介绍
控件焦点的6.5.2节会介绍这一过程 ..... /* 第一次“遍历”时，必然会进
行窗口的重新布局以便确定窗口尺寸并获取一块Surface。这意味着窗
口将第一次可见，即所谓的弹出。在默认情况下，WMS会为窗口添加
一个弹出动画，此时会在布局的返回值中增加
RELAUDIO_LAYOUT_RES_ANIMATION标记。出于效率考虑，在窗口进行动画
的过程中将跳过绘制*/
if ((reLayoutResult &
WindowManagerGlobal.RELAUDIO_LAYOUT_RES_ANIMATING) != 0) {
mWindowsAnimating = true; } } else if (mWindowsAnimating) { // ① 如果
窗口处在动画过程中，则跳过绘制阶段以提高动画的效率
skipDraw =
true; } mFirst = false; mWillDrawSoon = false; mNewSurfaceNeeded =
false; mViewVisibility = viewVisibility; /* ② 确定是否需要向WMS发送绘
```

制完成的通知。第4章介绍过窗口的绘制状态，当窗口初次获得Surface时其绘制状态被设置为DRAW_PENDING，仅当WMS接收到窗口的finishDrawingWindow()回调时，才会使窗口迁移到COMMIT_DRAW_PENDING，进而迁移到READY_TO_SHOW。如果没有调用WMS.finish- DrawingWindow()，即便在Surface上绘制了内容，WMS也会因为窗口的绘制状态不为READY_TO_SHOW而不会将窗口显示出来。mReportNextDraw是ViewRootImpl用来确定是否需要向WMS发起finishDrawingWindow()回调的条件。在这里，窗口初次获得了一块Surface，此时窗口绘制状态必然为DRAW_PENDING，因此将mReportNext- Draw设置为true */if ((reLayoutResult & WindowManagerGlobal.RELAYOUT_RES_FIRST_TIME) != 0) {
 mReportNextDraw = true;}// ③ 当mView不可见时，自然也不需要进行绘制boolean cancelDraw = || viewVisibility != View.VISIBLE;if (!cancelDraw && !newSurface) { if (!skipDraw || mReportNextDraw) {
 // ④ performDraw()负责整个控件树的绘制 performDraw(); } } else { /* ⑤
 如果窗口在此次“遍历”中获取了Surface，则跳过本次“遍历”的绘制。并且通过schedule- Traversals()方法重新做一次“遍历”，并在新的“遍历”中完成绘制 */ if (viewVisibility == View.VISIBLE) { scheduleTraversals(); }
 } }

可见，同其他阶段一样，绘制也是有可能被跳过的。下面介绍跳过绘制的原因。

·skipDraw：当窗口处于动画状态时，skipDraw会被置true使得跳过绘制。在Android看来，用户很容易注意到窗口动画的平滑性，因此它跳过了窗口的绘制使得更多的CPU/GPU资源用来处理动画，在这个过程中窗口的内容是被冻结的。另外需要注意的是，skipDraw的设置会因为mReportNextDraw而失效。上面的代码分析中介绍mReportNextDraw的作用是为了在窗口是DRAW_PENDING状态时向WMS发起finishDrawingWindow（）回调。因此mReportNextDraw为true时窗口的Surface尚未被显示出来并且没有任何内容。倘若此时不进行绘制工作会导致窗口迟迟不能迁移到COMMIT_DRAW_PENDING状态进而被显示出来，那么窗口动画也就无从谈起了。

·cancelDraw：当mView不可见时，自然也不需要进行绘制。

·newSurface：newSurface表明窗口在本次“遍历”中获取了一块Surface（可能由于这是第一次“遍历”或者mView从不可见变为可见）。在这种情况下，ViewRootImpl选择通过调用scheduleTraversals（）在下次“遍历”中进行绘制，而不是在本次进行绘制。

可见，绘制阶段的限制条件相对于前4个阶段来说要宽松得多，在常态下只要performTraversals（）被调用，则一定会执行绘制阶段。

performDraw（）方法就是控件树的绘制入口。由于控件树的绘制十分复杂，因此，perform-Draw（）方法的工作原理将在6.4节中单独介

绍。

6.performTraversals () 方法总结

本节通过将performTraversals () 方法拆分为5个阶段进行详细介绍。图6-10以布局为主线体现了这5个过程的基本流程关系。

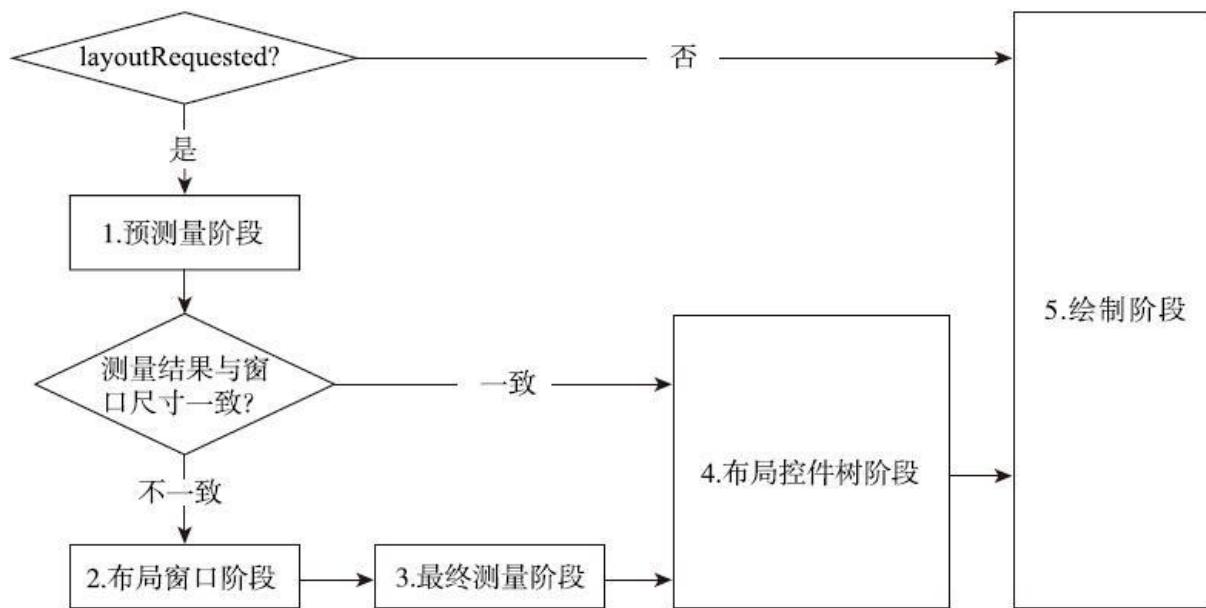


图6-10 `performTraversals ()` 的5个工作阶段的工作流程

可见，前4个阶段以`layoutRequested`为执行条件，即在“遍历”之前调用了`request-Layout ()`方法。这是由于前4个阶段的主要工作目的是确定控件的位置与尺寸。因此，仅当一个或多个控件有改变位置或尺寸的需求时（此时会调用`requestLayout ()`）才有执行的必要。

即使requestLayout () 未被调用过，绘制阶段也会被执行。因为很多时候需要在不改变控件位置与尺寸下进行重绘，例如某个控件改变了其文字颜色或背景色的情况下。与requestLayout () 方法相对，这种情况发生在控件调用了invalidate () 方法的时候。



注意

即便requestLayout () 没有被调用过，有可能由于LayoutParams、mView的可见性等与窗口有关的属性发生变化时，“遍历”流程仍会进入第二阶段。由于这些属性与布局没有直接联系，因此图6-10并没有体现这一点。

6.3.3 ViewRootImpl总结

本节主要介绍了ViewRootImpl的创建，以及核心performTraversals () 方法的实现原理。读者从中可以对ViewRootImpl架构与工作方式有深入理解。作为整个控件树的管理者，ViewRootImpl十分复杂与庞大，不过其很多工作都会落在本节所介绍的5个阶段中完成，所以深入理解

这5个阶段的实现可以为学习ViewRootImpl的其他工作打下坚实的基础。在本章后面的内容中将介绍控件树的绘制与输入事件派发两个方面的内容，那时会重新回到View-RootImpl，探讨那些位于5个阶段之外的工作。

6.4 深入理解控件树的绘制

接下来将讨论控件系统中非常核心的内容——控件树的绘制。在开发Android自定义控件时，往往都需要重写View.onDraw（）方法以绘制其内容到一个给定的Canvas中，而且开发者不需要知道控件以外的任何细节。本节将详细介绍Android控件系统是如何为这个简单的onDraw（）方法提供支持，以及隐藏在onDraw（）方法后面的有趣内容。

另外，由于绘制是一种开销很大的操作，因此在相关代码中对效率的优化随处可见，读者可以留意其改善绘制效率的思想与方式。

6.4.1 理解Canvas

既然要讨论绘制，就不得不提Canvas。Canvas是一个绘图工具类，其API提供了一系列绘图指令供开发者使用。根据绘制加速模式的不同，Canvas有软件Canvas与硬件Canvas之分。不过无论软件还是硬件，Canvas的这些绘图指令都可以分为如下两部分：

- 绘制指令。这些最常用的指令由一系列名为drawXXX（）的方法提供。它们用来实现实际的绘制行为，例如绘制点、线、圆以及方块等。

·辅助指令。这些用于提供辅助功能的指令将会影响后续绘制指令的效果，如设置变换、剪裁区域等。Canvas还提供了save () 与restore () 用于撤销一部分辅助指令的效果。

既然Canvas是一个绘制工具类，那么通过它绘制的内容到哪里去了呢？

1.Canvas的绘制目标

对软件Canvas来说，其绘制目标是一个建立在Surface之上的位图Bitmap，如图6-11所示。

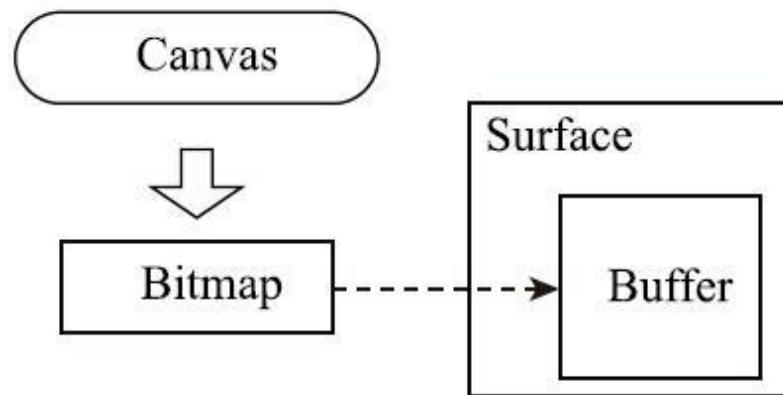


图6-11 软件Canvas的绘制目标

当通过Surface.lockCanvas () 方法获取一个Canvas时会以Surface的内存创建一个Bitmap，通过Canvas所绘制的内容都会直接反映到Surface中。

而硬件Canvas的绘制目标有两种。一种是Hardware-Layer，读者可以将其理解为一个GL Texture（即纹理），或者更简单地认为它是一个硬件加速下的位图（Bitmap）。而另外一种绘制目标则比较特别，被称为DisplayList。与Bitmap及HardwareLayer不同的是，DisplayList不是一块Buffer，而是一个指令序列。DisplayList会将Canvas的绘制指令编译并优化为硬件绘制指令，并且可以在需要时将这些指令回放到一个HardwareLayer上，而不需要重新使用Canvas进行绘制。

Bitmap、HardwareLayer以及DisplayList都可以称为Canvas的画布。



说明

从使用角度来说，HardwareLayer与Bitmap十分相似。开发者可以将一个Bitmap通过Canvas绘制到另一个Bitmap上，也可以将一个HardwareLayer绘制到另一个HardwareLayer上。二者的区别仅在于使用时采用了硬件加速还是软件加速。

另外，将DisplayList回放到HardwareLayer上，与绘制一个Bitmap或HardwareLayer的结果并没有什么不同。只不过DisplayList并不像Bitmap那样存储了绘制的结果，而是存储了绘制的过程。

理解这三者的统一性对于后续的学习十分重要。

2.Canvas的坐标变换

相对于绘制指令，Canvas的辅助指令远不那么直观。而且在Android控件绘制的过程中大量使用了这些指令，因此有必要先对Canvas的辅助指令做下了解。其中最常用的辅助指令莫过于变换指令了。

参考一个例子，当需要在（100， 200）的位置绘制一个宽度为50，高度为100的矩形时会怎么做呢？下面是一种常规的实现：

```
[Sample Code A] float x = 100f, y = 200f, w = 50f, h = 100f;  
mCanvas.drawRect(x, y, x + w, y + h, mPaint);
```

这种实现十分直观，按照需求直接在绘制指令中糅合了位置与尺寸参数。

再看另外一种等效的实现方法：

```
[Sample Code B] float x = 100f, y = 200f, w = 50f, h = 100f; // 先使用  
translate()方法将后续绘制的坐标系原点变换到(x, y)的位置  
mCanvas.translate(x, y); // 在新的坐标系下绘制矩形  
mCanvas.drawRect(0, 0, w, h, mPaint);
```

这种实现通过Canvas的translate () 变换指令剥离了矩形位置信息与尺寸信息，使得绘制矩形时的参数得到了简化。translate () 变换指令改变了后续绘图指令所使用的坐标系，将其在水平方向平移了100个点并在垂直方向平移了200个点，如图6-12所示。随后的drawRect () 便在这个新的坐标系中完成。由于坐标系相对于原始坐标系已经偏移了(100, 200)，因此在(0, 0)位置绘制的结果与Sample Code A一致。

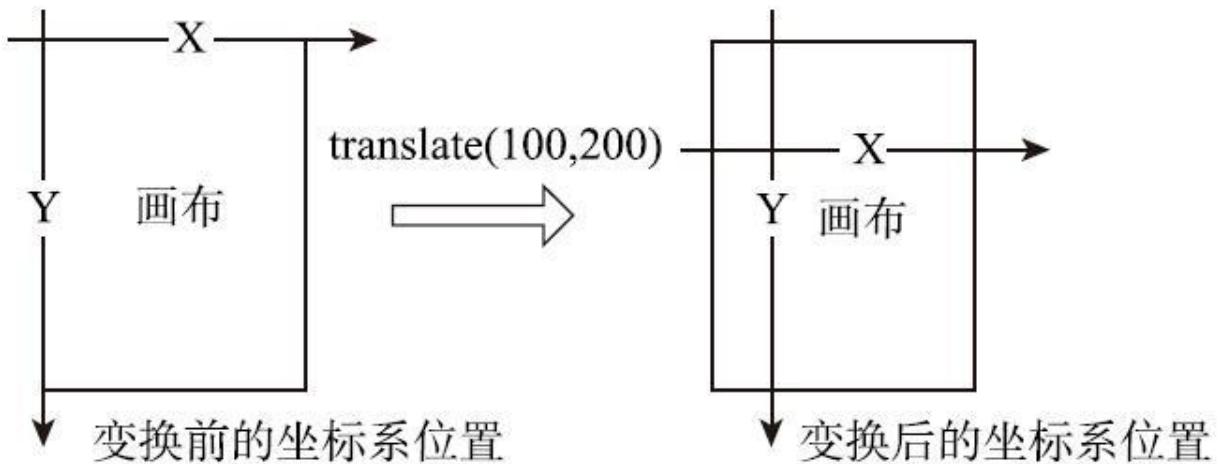


图6-12 `translate()` 变换指令对坐标系的影响

使用变换指令进行绘图看似更加麻烦，而且在一定程度上难于理解。然而试想一下，这里不是绘制一个矩形，而是一连串的图形，使用变换指令可以省却为每一个图形的绘制指令增加位置参数的麻烦。尤其是，当这些图形由另外的开发者实现时（如`onDraw()` 函数），开发者无须关心任何位置信息，仅需在原点进行绘制，而位置的计算则在

外部通过一个变换指令一次性完成。从这个意义上讲，变换指令极大地降低了复杂绘制的难度。

另外，在某些绘图需求下，不使用变换指令基本上是无法实现的。例如绘制一个顺时针旋转 60° 的矩形时，使用常规方法无法实现，但是却可以通过Canvas () 的rotate () 指令将坐标系顺时针旋转 90° ，然后就可以轻易绘制这个矩形。

除了平移坐标系以及旋转坐标系以外，Canvas还提供了以下变换指令对坐标系进行修改。

·scale：对坐标系的刻度进行缩放。例如执行Canvas.scale (2, 3) 之后，新坐标系的刻度将是原坐标系的两倍，即新坐标系下的 (10, 30) 相当于原坐标系的 (20, 90) 。

·skew：将坐标系进行切变，其参数为切变方向角度的tan值。在经过这种变换的坐标系中，所绘制的矩形将会变成菱形，因此这种变换非常适合做影子效果。其效果如图6-13所示。

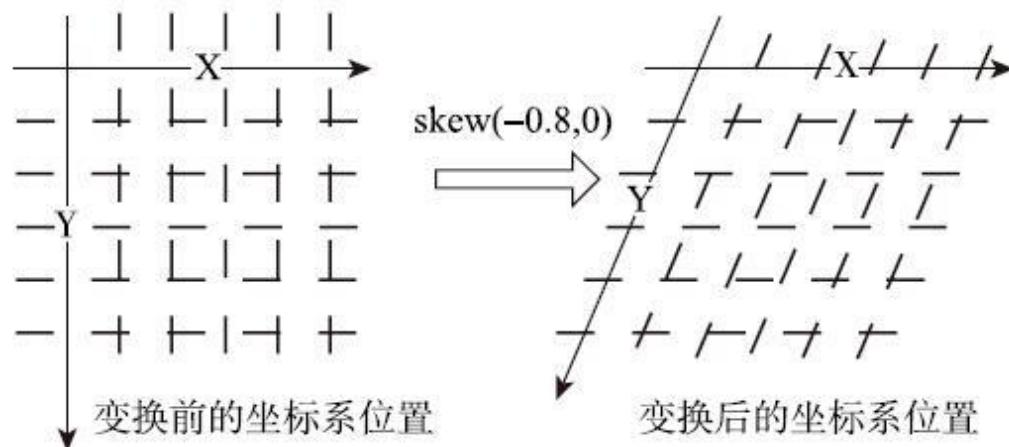


图6-13 skew对坐标系的影响



注意

当连续进行多次变换时，后一次变换都是在前一次变换后的坐标系中进行的。例如执行`scale (2, 3)` ; `translate (2, 3)` 时，新坐标系的原点将会变为原坐标系的 (4, 9) 位置，并且其刻度大小为原坐标系的2倍。

当一个Canvas执行多次变换指令后，要恢复成变换之前的坐标系似乎变得非常困难，因为必须逐次进行相应的逆变换才行。为了解决这个

问题，Canvas提供了配套使用的save () 与restore () 方法用以撤销不需要的变换。参考下面的代码：

```
..... // 其他代码 // save()将会创建一个保存点 mCanvas.save(); // 根据需求进行各种变换 mCanvas.translate(...); mCanvas.scale(...);  
mCanvas.rotate(...); ..... // 在新的坐标系下进行绘制 /* restore()方法将会把坐标系恢复到执行save()时的状态，即save()与restore()之间的坐标变换都会被舍弃 */ mCanvas.restore();
```

save () /restore () 可以嵌套调用，在这种情况下restore () 将会把坐标系状态返回到与其配对的save () 所创建的保存点。另外，也可以通过保存某个save () 的返回值，并将这个返回值传递给restoreToCount () 方法的方式来显式地指定一个保存点。

那么坐标系变换对于控件树的绘制有什么意义呢？开发者在重写View.onDraw () 方法时，从未考虑过控件的位置、旋转或缩放等状态。这说明在onDraw () 方法执行之前，这些状态都已经以变换的方式设置到Canvas中了，因此onDraw () 方法中Canvas使用的是控件自身的坐标系。而这个控件自身的坐标系就是通过Canvas () 的变换指令从窗口坐标系沿着控件树一步一步变换出来的。当一个父控件进行绘制时，它会首先根据自己的位置、滚动位置以及缩放旋转等属性对Canvas进行变换，使得Canvas的坐标系变换为其自身的坐标系，再调用onDraw () 方法绘制自己。然后将这个Canvas交给其第一个子控

件，子控件会首先通过`save ()`方法将其父控件的坐标系保存下来，将`Canvas`变换为自己的坐标系，再通过`onDraw ()`进行绘制，然后将`Canvas`交给孙控件进行变换与绘制。当子控件及孙控件都完成绘制之后通过`restore ()`方法将`Canvas`恢复为父控件的坐标系，父控件再将`Canvas`交给第二个子控件进行绘制，以此类推。在后面对控件树绘制的代码分析中将会看到这种递推的变换关系是如何体现的。

6.4.2 View.invalidate () 与脏区域

为了保证绘制的效率，控件树仅对需要重绘的区域进行绘制。这部分区域称为“脏区域”，即`Dirty Area`。当一个控件的内容发生变化而需要重绘时，它会通过`View.invalidate ()`方法将其需要重绘的区域沿着控件树提交给`ViewRootImpl`，并保存在`ViewRootImpl`的`mDirty`成员中，最后通过`scheduleTraversals ()`引发一次“遍历”，进而进行重绘工作。`ViewRootImpl`会保证仅有位于`mDirty`所描述的区域得到重绘，从而避免不必要的开销。

另外，`View.invalidate ()`在回溯到`ViewRootImpl`的过程中会将沿途的控件标记为脏的，即将`PFLAG_DIRTY`或`PFLAG_DIRTY_OPAQUE`两者之一添加到`View.mPrivateFlags`成员中。两者都表示控件需要随后进行重绘，不过二者在重绘效率上有区别。`View`有一个方法`isOpaque ()`供其子类进行重写，用于通过返回值确认此控件是否为“实心”的。所谓的“实心”控件，是指在其`onDraw ()`方法中能够保证此控件的所有区

域都会被其所绘制的内容完全覆盖。换句话说，透过此控件所属的区域无法看到此控件之下的内容，也就是既没有半透明也没有空缺的部分。在invalidate () 的过程中，如果控件是“实心”的，则会将此控件标记为PFLAG_DIRTY_OPAQUE，否则为PFLAG_DIRTY。控件系统在重绘过程中区分这两种标记以决定是否为此控件绘制背景。对“实心”控件来说，其背景是被onDraw () 的内容完全遮挡的，因此便可跳过背景的绘制工作从而提高效率。注意isOpaque () 方法的返回值不是一成不变的。以ListView为例，其isOpaque () 方法会根据其ListItem是否可以铺满其空间来决定返回值。当ListItem比较少时它是非“实心”的，而当ListItem比较多时它则变成“实心”控件。

invalidate () 方法必须在主线程执行，而scheduleTraversals () 所引发的“遍历”也是在主线程执行（因为scheduleTraversals () 是向主线程的Handler发送消息）。所以调用invalidate () 方法并不会使得“遍历”立即开始，这是因为在调用invalidate () 的方法执行完毕之前（准确地说是主线程的Looper处理完其他消息之前），主线程根本没有机会处理scheduleTraversals () 所发出的消息。这种机制带来的好处是，在一个方法中可以连续调用多个控件的invalidate () 方法，而不用担心会由于多次重绘而产生的效率问题。另外，多次调用invalidate () 方法会使得ViewRootImpl多次接收到设置脏区域的请求，ViewRootImpl会将这些脏区域累加到mDirty中，进而在随后的“遍历”中一次性地完成所有脏区域的重绘。

有些时候需要忽略mDirty的设置以进行完整绘制，例如窗口的第一次绘制，或者窗口的尺寸发生变化的时候。在这些情况下ViewRootImpl的mFullRedrawNeeded成员将被设置为true，这会使得在绘制之前将mDirty所描述的区域扩大到整个窗口，进而实现完整重绘。

6.4.3 开始绘制

回到ViewRootImpl，performTraversals () 的最后一个阶段（即绘制阶段），发现调用了performDraw () 方法，而这个方法就是绘制控件树的入口。参考其实现：

```
[ViewRootImpl.java-->ViewRootImpl.performDraw()]
private void performDraw() {.....// ① 调用draw()方法进行实际的绘制工作try {
    draw(fullRedrawNeeded);} finally {.....}/* ② 通知WMS绘制已经完成。
如前文所述，如果mReportNextDraw为true，表示WMS正在等待
finishDrawingWindow()回调以便将窗口的绘制状态切换至
COMMIT_DRAW_PENDING */if (mReportNextDraw) {
    mReportNextDraw = false; ..... try {
        mWindowSession.finishDrawing(mWindow); } catch (RemoteException e)
    {}} }
```

performDraw () 方法的工作很简单，一是通过draw () 方法执行实际的绘制工作，二是如果需要，则向WMS通知绘制已经完成。注意draw

() 方法的参数fullRedrawNeeded来自于前文所述的成员变量
mFullRedrawNeeded。

参考draw () 方法的实现：

```
[ViewRootImpl.java-->ViewRootImpl.draw()] private void draw(boolean fullRedrawNeeded) {Surface surface = mSurface;...../* 计算mView在垂直方向的滚动量 (ScrollY) , 滚动将保存在mScroller与mScrollY中, ViewRootImpl所计算的滚动量的目的与ScrollView或ListView计算的滚动量的意义有别。在VisibleInsets存在的 情况下, ViewRootImpl需要保证某个关键的控件是可见的。例如当输入法弹出时, 接收输入的 TextView 必须位于不被输入法遮挡的区域内。倘若布局结果使得它被输入法遮挡, 就必须根据VisibleInset 与它的相对位置计算一个滚动量, 使得整个控件树的绘制位置产生偏移从而将TextView露出来。计算所得的滚动量被保存在mScroller中 */scrollToRectOrFocus(null, false);int yoff; /* 上述的滚动量记录在mScroller中, 为的是这个滚动显得不那么突兀, ViewRootImpl使用mScroller产生一个动画效果。 mScroller类似于一个插值器, 用于计算本次绘制的时间点所需要使用的滚动量 */boolean animating = mScroller != null && mScroller.computeScrollOffset();if (animating) { // 倘若mScroller正在执行滚动动画, 则采用mScroller所计算的滚动量 yoff = mScroller.getCurrY();} else { // 倘若mScroller的动画已经结束, 则直接使
```

用上面的scrollToRectOrFocus()所计算的滚动量 $yoff = mScrollY;$ /* 倘若新计算的滚动量与上次绘制的滚动量不同，则必须进行完整重绘。这很容易理解，因为发生滚动时，整个画面都需要更新 */ if ($mCurScrollY \neq yoff$) { $mCurScrollY = yoff$; fullRedrawNeeded = true; } // 如果存在一个ResizeBuffer动画，则计算此动画相关的参数 // 如果需要进行完整重绘，则修改脏区域为整个窗口 if (fullRedrawNeeded) { dirty.set(0, 0, (int) (mWidth * appScale + 0.5f), (int) (mHeight * appScale + 0.5f)); } if (!dirty.isEmpty() || mIsAnimating) { /* ① 当满足下列条件时，表示此窗口采用硬件加速的绘制方式。硬件加速的绘制入口是HardwareRenderer.draw()方法 */ if (attachInfo.mHardwareRenderer != null && attachInfo.mHardwareRenderer.isEnabled()) { if (attachInfo.mHardwareRenderer.draw(mView, attachInfo, this, animating ? null : mCurrentDirty)) { } // ② 而软件绘制的入口则是drawSoftware()方法 } else if (!drawSoftware(surface, attachInfo, yoff, scalingRequired, dirty)) { return; } } // 如果mScroller仍在动画过程中，则立即安排下一次重绘 if (animating) { mFullRedrawNeeded = true; scheduleTraversals(); } }

ViewRootImpl.draw () 方法中产生了硬件加速绘制与软件绘制两个分支，其分支条件为mAttachInfo.mHardwareRenderer是否存在并且有效。在ViewRootImpl.setView () 中会调用enableHardwareAcceleration () 方法，倘若窗口的LayoutParams.flags中包含FLAG_HARDWARE_ACCELERATED标记，这个方法会通过

`HardwareRenderer.createGlRenderer()` 创建一个`HardwareRenderer`并保存在`mAttachInfo`中。因此`mAttachInfo`所保存的`HardwareRenderer`是否存在便成为区分使用硬件加速绘制还是软件绘制的依据。

硬件加速绘制与软件绘制的流程是一致的，因此接下来将先通过较为简单的软件绘制来了解控件树绘制的基本流程，然后再以此基本流程为指导来讨论硬件加速绘制所特有的内容。

6.4.4 软件绘制的原理

软件绘制由`ViewRootImpl.drawSoftware()`方法完成，参考以下代码：

```
[ViewRootImpl.java-->ViewRootImpl.drawSoftware()]
private boolean drawSoftware(Surface surface, AttachInfo attachInfo, int yoff, boolean scalingRequired, Rect dirty) {.....// 定义绘制所需的Canvas
    canvas;try { /* ① 通过Surface.lockCanvas()获取一个以此Surface为画布的Canvas。注意其参数为前面所计算的脏区域*/
        canvas =
        mSurface.lockCanvas(dirty);} catch (...) {.....}try { ..... /* 绘制即将开始之前，首先清空之前所计算的脏区域。这样一来，如果在绘制的过程中执行了View.invalidate()，则可以重新计算脏区域 */
        dirty.setEmpty(); /* 将当前的时间戳保存在AttachInfo.mDrawingTime中，随后控件进行绘制时可以根据这个时间戳以确定动画参数 */
        attachInfo.mDrawingTime =
        SystemClock.uptimeMillis(); ..... try { /* ② 使Canvas进行第一次变换。
    
```

此次变换的目的是使得其坐标系按照之前所计算的滚动量进行相应的滚动。随后绘制的内容都会在滚动后的新坐标系下进行 */
canvas.translate(0, -yoff); // ③ 通过mView.draw()在Canvas上绘制整个控件树 mView.draw(canvas); } finally {.....} } finally { try { // ④ 最后的步骤，通过Surface.unlockCanvasAndPost()方法显示绘制后的内容
surface.unlockCanvasAndPost(canvas); } catch (IllegalArgumentException e) {.....} }return true; }

不难看出，drawSoftware () 主要有4步工作：

- 第一步，通过Surface.lockCanvas () 获取一个用于绘制的Canvas。
- 第二步，对Canvas进行变换以实现滚动效果。
- 第三步，通过mView.draw () 将根控件绘制在Canvas上。
- 第四步，通过Surface.unlockCanvasAndPost () 显示绘制后的内容。

其中第二步与第三步是控件绘制过程的两个基本阶段，即首先通过 Canvas的变换指令将Canvas的坐标系变换到控件自身的坐标系下，然后再通过控件的View.draw (Canvas) 方法将控件的内容绘制在这个变换后的坐标系中。

注意，在View中还有draw (Canvas) 方法的另外一个重载，即 View.draw (ViewGroup, Canvas, long) 。二者的区别在于后者是在父

控件的绘制过程中所调用的（参数ViewGroup就是其父控件），并且参数Canvas所在的坐标系为其父控件的坐标系。View.draw (ViewGroup, Canvas, long) 会根据控件的位置、旋转、缩放以及动画对Canvas进行坐标系的变换，使得Canvas的坐标系从父控件的坐标系变换到本控件的坐标系，并且会在变换完成后调用draw (Canvas) 来在变换后的坐标系中进行绘制。由此看来，相对于另外一个重载，draw (Canvas) 的绘制工作更加纯粹，它用来在不做任何加工的情况下将控件的内容绘制在给定的Canvas上。这也是为什么将控件内容输出到一个Bitmap中时使用draw (Canvas)，从而无论控件如何被拉伸、旋转，目标Bitmap中存储的都是其最原始的样子。因此draw (Canvas) 方法是探讨控件绘制原理的最佳切入点。



注意

View.draw (ViewGroup, Canvas, long) 的工作远不止坐标系变换那么简单，它还包含了硬件加速、绘图缓存以及动画计算等工作。但是在讨论它与draw (Canvas) 之间的关系时，最重要的还当属坐标系变

换。在后面内容的学习中读者会逐步地认识View.draw (ViewGroup, Canvas, long)。

1. 纯粹的绘制：View.draw (Canvas)

参考View.draw (Canvas) 的实现：

```
[View.java-->View.draw()] */ public void draw(Canvas canvas) {final int  
privateFlags = mPrivateFlags;// 通过检查PFLAG_DIRTY_OPAQUE是否  
存在于mPrivateFlags中以确定是否是“实心”控件final boolean  
dirtyOpaque = (privateFlags & PFLAG_DIRTY_MASK) ==  
PFLAG_DIRTY_OPAQUE && (mAttachInfo == null ||  
!mAttachInfo.mIgnoreDirtyState); // ① 首先是绘制背景。注意，如6.4.2节  
所述，为了提高效率，“实心”控件的背景绘制工作会被跳过if  
(!dirtyOpaque) { final Drawable background = mBackground; if  
(background != null) { // 将背景绘制到Canvas上 if ((scrollX | scrollY) ==  
0) { background.draw(canvas); } else { /* 这里是一个有趣的处理。注意  
draw(Canvas)方法是在控件自身的坐标系下调用的。就是说Canvas已经  
根据其mScrollX/Y对Canvas进行了变换以实现控件滚动的效果，从而所  
绘制的背景也会被滚动。不过Android希望仅滚动控件的内容，而保持  
背景静止。因此在绘制背景时会首先进行滚动的逆变换以撤销先前实  
行的滚动变换，完成背景绘制之后再将滚动变换重新应用到Canvas上  
*/ canvas.translate(scrollX, scrollY); background.draw(canvas);
```

```
canvas.translate(-scrollX, -scrollY); } } } //倘若控件不需要绘制渐变边界，则可以进入简便绘制流程 if (!verticalEdges && !horizontalEdges) { //  
② 通过调用onDraw()方法绘制控件自身的内容 if (!dirtyOpaque)  
onDraw(canvas); /* ③ 通过调用dispatchDraw()方法绘制子控件。如果当前控件不是一个ViewGroup，此方法什么都不做 */  
dispatchDraw(canvas); // ④ 如果有必要，根据滚动状态绘制滚动条  
onDrawScrollBars(canvas); // 完成此控件的绘制 return; } /*接下来是完整绘制流程，完整绘制流程除了包含上面的简便流程之外，还包含绘制渐变边界的工作*/..... }
```

可见纯粹的控件绘制过程非常简单，主要有以下4步：

- 绘制背景，注意背景不会受到滚动的影响。
- 通过调用onDraw () 方法绘制控件自身的内容。
- 通过调用dispatchDraw () 绘制其子控件。
- 绘制控件的装饰，即滚动条。

除非特殊需要，子控件应当只重载onDraw () 方法而不是draw (Canvas) 方法，以保证背景、子控件和装饰器得以正确绘制。

2.确定子控件绘制的顺序：dispatchDraw ()

从分析的轨迹来看，前述的View.draw () 被 ViewRootImpl.drawSoftware () 调用，因此View.draw () 仅仅绘制了根控件自身的内容。那么控件树的其他控件是如何得到重绘的呢？这将有必要探讨View.dispatchDraw () 方法。其在View类中的实现是一个空方法，而ViewGroup重写了它。此方法是重绘工作得以从根控件mView延续到控件树中每一个子控件的重要纽带。参考实现如下：

```
[ViewGroup.java-->ViewGroup.dispatchDraw()] protected void  
dispatchDraw(Canvas canvas) {final int count = mChildrenCount;final  
View[] children = mChildren;int flags = mGroupFlags;..... // 动画相关的处  
理int saveCount = 0; /* ① 设置剪裁区域。有时候子控件可能部分或者完  
全位于ViewGroup之外。在默认情况下，ViewGroup的下列代码通过  
Canvas.clipRect()方法将子控件的绘制限制在自身区域之内。超出此区  
域的绘制内容将会被裁剪。是否需要进行越界内容的裁剪取决于  
ViewGroup.mGroupFlags中是否包含CLIP_TO_PADDING_ MASK标  
记，因此开发者可以通过ViewGroup.setClipToPadding()方法修改这一行  
为，使得子控件超 出的内容仍然得以显示*/final boolean clipToPadding  
= (flags & CLIP_TO_PADDING_ MASK) ==  
CLIP_TO_PADDING_ MASK;if (clipToPadding) { // 首先保存Canvas的状  
态，随后可以通过Canvas.restore()方法恢复到这个状态 saveCount =  
canvas.save(); // Canvas.clipRect()将保证给定区域之外的绘制都会被裁  
剪 canvas.clipRect(mScrollX + mPaddingLeft, mScrollY + mPaddingTop,
```

mScrollX + mRight - mLeft - mPaddingRight, mScrollY + mBottom - mTop - mPaddingBottom);}boolean more = false;// 获取当前的时间戳，用于子控件计算其动画参数final long drawingTime = getDrawingTime();/* ② 遍历绘制所有的子控件。根据mGroupFlags中是否存在FLAG_USE_CHILD_DRAWING_ORDER标记， dispatchDraw()会采用两种不同的绘制顺序 */if ((flags & FLAG_USE_CHILD_DRAWING_ORDER) == 0) { /* 在默认情况下,dispatchDraw()会按照mChildren列表的索引顺序进行绘制。

ViewGroup.addView()方法默认会将子控件添加到列表末尾，同时它提供了一个重载允许开发者将子控件添加到列表的一个指定位置。就是说默认情况下的绘制顺序与子控件加入ViewGroup的先后关系或调用addView()时所指定的位置有关 */ for (int i = 0; i < count; i++) { final View child = children[i]; if ((child.mViewFlags & VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null) { // ③ 调用drawChild()方法绘制一个子控件 more |= drawChild(canvas, child, drawingTime); } } } else { /*倘若mGroupFlags成员中存在FLAG_USE_CHILD_DRAWING_ORDER标记，则表示此ViewGroup希望按照其自定义的绘制顺序进行绘制。

自定义的绘制顺序由getChildDrawingOrder()方法实现 */ for (int i = 0; i < count; i++) { /* 与默认绘制次序时的循环变量i的意义不同，在这里的i并不是指mChildren的索引，而是指已经完成绘制的子控件的个数。

getChildDrawingOrder()的实现者可以根据已完成绘制子控件的个数决

```
定下一个需要进行绘制的子控件的索引 */ final View child =  
children[childDrawingOrder(count, i)]; if ((child.mViewFlags &  
VISIBILITY_MASK) == VISIBLE || child.getAnimation() != null) { // 与默  
认绘制顺序一样，通过drawChild()方法绘制一个子控件 more |=  
drawChild(canvas, child, drawingTime); } } }..... // 动画相关的处理// ④ 通  
过Canvas.restoreToCount()撤销之前所做的剪裁设置if (clipToPadding) {  
canvas.restoreToCount(saveCount); }..... // 动画相关的处理 }
```

在dispatchDraw () 方法中有很多与动画相关的工作，但它们与绘制的主线流程没有太大关系，因此本节将它们忽略了，而在6.4.7节中会做详细介绍。

在本方法中，最重要的莫过于它所定义的两种重绘顺序。重绘顺序对子控件来说意义非常大，因为当两个或多个子控件有重叠时，后绘制的控件会覆盖先绘制的控件，也就是说，后绘制的控件拥有被用户看到的优先权。在默认情况下，后加入ViewGroup的子控件位于mChildren的尾部，因此绘制顺序与加入顺序一致。ViewGroup可以通过ViewGroup.setChildrenDrawingOrderEnabled () 方法将FLAG_USE_CHILD_DRAWING_ORDER标记加入mGroupFlags，并重写getChildDrawingOrder () 来自定义绘制的顺序。

举例来说，TabWidget维护了一系列的Tab页的标签，而每个Tab页的标签都是它的一个子控件。TabWidget对绘制顺序的需求是：用户所选择

的那个Tab页的标签无论位于mChildren的什么位置，都希望它能够最后被绘制，以便用户可以不被遮挡地、完整看到它。显然默认的绘制顺序无法满足其需求。因此其在初始化时调用了

`setChildrenDrawingOrderEnabled (true)`，并以如下方式重写了
`getChildDrawingOrder ()`。

```
[TabWidget.java-->TabWidget.getChildDrawingOrder()]
protected int
getChildDrawingOrder(int childCount, int i) {
    if (mSelectedTab == -1) {
        return i;
    } else { // 最后一次绘制永远留给被选中的Tab
        if (i == childCount - 1) {
            return mSelectedTab; // 而其他的绘制在跳过了选中的Tab之后按照
        } else if (i >= mSelectedTab) {
            return i + 1;
        } else {
            return i;
        }
    }
}
```

如此一来，无论发生什么情况都可以保证被选中的Tab会被最后绘制。



注意

`getChildDrawingOrder ()` 决定了绘制的顺序，也就决定了覆盖顺序。覆盖顺序影响了另一个重要的工作，即触摸事件的派发。按照使用习

惯，用户触摸的目标应当是他所能看到的东西。因此当用户触摸到两个子控件的重叠区域时，覆盖者应当比被覆盖者拥有更高的事件处理优先级。因此在6.5.5节讨论触摸事件的派发时将再次看到类似确定绘制顺序的代码。

另外，`dispatchDraw()` 在设置裁剪区域的时候把滚动量也考虑在内了。为了解释如此计算的原因，就必须弄清楚View的`mScrollX/Y`两个值的深层含义。`mScrollX/Y`会导致包括控件自身的内容以及其子控件的位置都产生偏移，这个作用与控件的位置`mLeft/mTop`可以说是一样的。那么二者的区别是什么呢？`mScrollX/Y`描述了控件内容在本控件中的坐标系位置。而`mLeft/mTop`则描述了控件本身在父控件坐标系中的位置。控件就好比嵌在墙壁上的一扇窗，`mLeft/mTop`就相当于这扇窗相对于墙壁左上角的位置。而控件的内容就像是位于窗后的一幅画，而这幅画相对于窗子的左上角的位置就是`mScrollX/Y`，参考图6-14。从这个意义上来说，当Canvas针对此控件的`mScrollX/Y`做过变换之后（`Canvas.translate(-mScrolX, -mScrollY)`）的坐标系准确地来说应该是控件内容的坐标系，即以那幅画的左上角为原点的坐标系。在控件内容的坐标系中，控件的位置（即那扇窗的位置）是（`mScrollX, mScrollY`）（如图6-15所示），因此以控件边界做裁剪时，必须将`mScrollX/Y`纳入计算之列。在不产生歧义的情况下，随后的叙述中不会区分控件自身的坐标系与控件内容的坐标系。读者只要理解二者之间的差异在于`mScrollX/Y`即可。

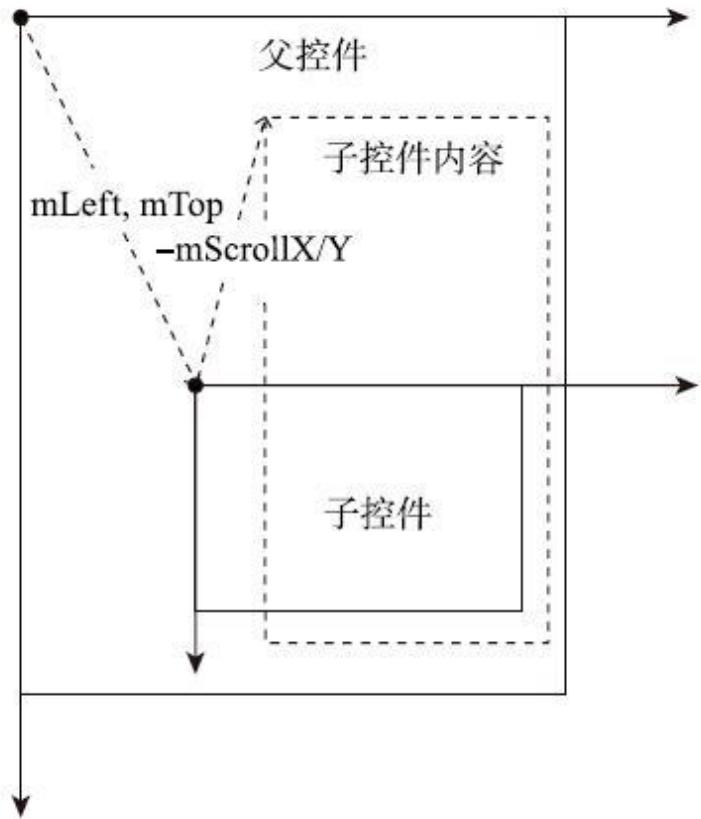


图6-14 mLeft/Top与mScrollX/Y的区别和联系

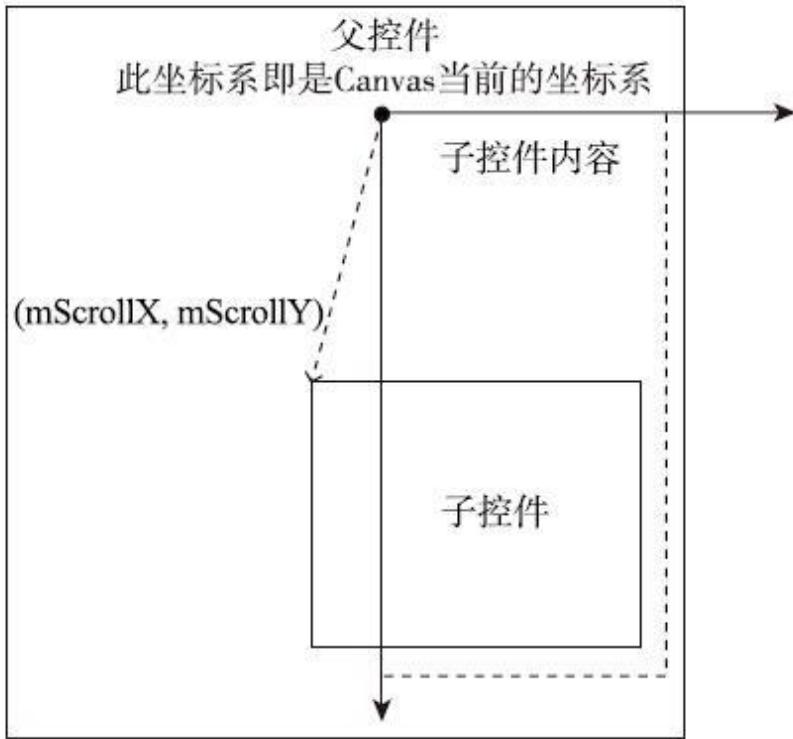


图6-15 控件在其内容坐标系下的位置

确定绘制顺序之后，通过ViewGroup.drawChild（）方法绘制子控件。

drawChild（）没有什么额外的工作，仅仅是调用子控件的View.draw（ViewGroup，Canvas，long）方法进行子控件的绘制而已。

3. 变换坐标系：View.draw（ViewGroup，Canvas，long）

在绘制子控件时，父控件调用子控件的View.draw（ViewGroup，Canvas，long）方法。在6.4.4节开始时曾简单介绍了这个方法的工作内容——为随后调用的View.draw（Canvas）准备坐标系。接下来将详细探讨坐标系的准备过程。另外此方法包含了硬件加速、绘图缓存以及

动画计算等工作，本节仅讨论软件加速、不使用绘图缓存并且动画计算已经完成的情况下所剩余的工作，以便能够专注在坐标系变换的原理分析上。参考代码如下：

```
[View.java-->View.draw(ViewGroup, Canvas, long)] boolean draw(Canvas  
canvas, ViewGroup parent, long drawingTime) { // 如果控件处于动画过程  
中, transformToApply会存储动画在当前时点所计算出的  
Transformation Transformation transformToApply = null; // ① 进行动画的  
计算，并将结果存储在transformToApply中。这是进行坐标变换的第一  
个因素..... /* ② 计算控件内容的滚动量。计算是通过computeScroll()完  
成的, computeScroll()将滚动的计算 结果存储在mScrollX/Y两个成员变  
量中。在一般情况下, 子类在实现computeScroll()时会考虑 使用  
Scroller类以动画的方式进行滚动。向Scroller设置一下目标的滚动量,  
以及滚动动画的持续 时间, Scroller会自动计算在动画过程中本次绘制  
所需的滚动量。 注意这是进行坐标变换的第二个因素 */ int sx = 0, sy =  
0;if (!hasDisplayList) { computeScroll(); sx = mScrollX; sy =  
mScrollY;}...../* ③ 使用Canvas.save()保存Canvas的当前状态。此时  
Canvas的坐标系为父控件的坐标系。在随后 将Canvas变换到此控件的  
坐标系并完成绘制后, 会通过Canvas.restoreTo()方法将Canvas重置到此  
时 的状态, 于是Canvas便可以继续用来绘制父控件的下一个子控件了  
*/ int restoreTo = -1;if (!useDisplayListProperties || transformToApply !=  
null) { restoreTo = canvas.save(); }/* ④ 第一次变换, 对应控件位置与滚
```

动量。最先处理的是子控件位置mLeft/mTop，以及滚动量。注意子控件的位置mLeft/Top是进行坐标变换的第三个因素 */if (offsetForScroll) { canvas.translate(mLeft - sx, mTop - sy); }...../* 倘若此控件的动画所计算出的变换存在（即有动画在执行），或者通过View.setScaleX/Y()等方法修改了控件自身的变换，则将它们所产生的变换矩阵应用到Canvas中 */if (transformToApply != null || alpha < 1 || !hasIdentityMatrix() || (mPrivateFlags3 & PFLAG3_VIEW_IS_ANIMATING_ALPHA) == PFLAG3_VIEW_IS_ANIMATING_ALPHA) { if (transformToApply != null || !childHasIdentityMatrix) { int transX = 0, transY = 0; // 记录滚动量 if (offsetForScroll) { transX = -sx; transY = -sy; } // 将动画产生的变换矩阵应用到Canvas if (transformToApply != null) { if (concatMatrix) { if (useDisplayListProperties) { /* useDisplayListProperties表示使用硬件加速，由于硬件加速与软件绘制方式上的差异，应用变换矩阵的方式也不同。在讨论硬件加速时再分析这部份内容 */ } else { /* ⑤ 将动画产生的变换矩阵应用到Canvas中。注意，这里首先撤销了对滚动量的变换，在将动画的变换矩阵应用给Canvas之后，重新应用滚动量变换*/ canvas.translate(-transX, -transY); canvas.concat(transformToApply.getMatrix()); canvas.translate(transX, transY); } } } /* ⑥ 将控件自身的变换矩阵应用到Canvas中。和动画矩阵一样，首先撤销了滚动量的变换，然后应用变换矩阵到Canvas后重新应用滚动量。控件自身的变换矩阵是进行坐标系变换的第四个

```
因素 */ if (!childHasIdentityMatrix && !useDisplayListProperties) {  
    canvas.translate(-transX, -transY); canvas.concat(getMatrix());  
    canvas.translate(transX, transY); } } .....} else if ((mPrivateFlags &  
PFLAG_ALPHA_SET) == PFLAG_ALPHA_SET) {.....}/* ⑦ 设置裁剪。  
当父控件的mGroupFlags包含FLAG_CLIP_CHILDREN时，子控件在绘  
制之前必须通过 canvas.clipRect()方法设置裁剪区域。注意要和  
dispatchDraw()中的裁剪工作加以区分。 dispatchDraw()中的裁剪是为了  
保证所有的子控件绘制的内容不得越过ViewGroup的边界。其设置由  
setClipToPadding()方法完成。而FLAG_CLIP_CHILDREN则表示所有子  
控件的绘制内容不得超出 子控件自身的边界，由setClipChildren()方法  
启用或禁用这一行为。另外注意，如上一小节所述， Canvas此时已经  
过了mScrollX/Y的变换，正处在控件内容的坐标系下，因此设置裁剪  
区域时需要将 mScrollX/Y计算在内 */if ((flags &  
ViewGroup.FLAG_CLIP_CHILDREN) ==  
ViewGroup.FLAG_CLIP_CHILDREN && !useDisplayListProperties) { if  
(offsetForScroll) { canvas.clipRect(sx, sy, sx + (mRight - mLeft), sy +  
(mBottom - mTop)); } else {.....}}.....// 由于本节讨论的是在不使用绘图  
缓存情况下的绘制过程，所以hasNoCache为trueif (hasNoCache) {  
boolean layerRendered = false; ..... // 使用硬件加速绘图缓存的方式对控  
件进行绘制，本节暂不关注 if (!layerRendered) { if (!hasDisplayList) { /*  
⑧ 使用变换过的Canvas进行最终绘制。 在这里见到了熟悉的
```

dispatchDraw()和draw(Canvas)两个方法。完成坐标系的变换之后，Canvas已经位于控件自身的坐标系之下，也就可以通过draw(Canvas)进行控件内容的实际绘制工作，这样一来，绘制流程便回到了“纯粹的绘制”位置，进而绘制背景、调用onDraw()及dispatchDraw()再加上绘制滚动条，其中dispatchDraw()还会把绘制工作延续给此控件的所有子控件。注意，当本控件的mPrivateFlags中包含PFLAG_SKIP_DRAW时，则以dispatchDraw()取代调用draw(Canvas)。这是一种效率上的优化。

对大多数ViewGroup来说，它们没有自己的内容，即onDraw()的实现为空，在为其设置null作为背景，并且又不需要绘制滚动条时，其绘制工作便仅剩下dispatchDraw()了。对于这种控件，控件系统会为其加上PFLAG_SKIP_DRAW标记，以便在这里直接调用dispatchDraw()这一捷径从而提高重绘的效率。PFLAG_SKIP_DRAW标记的设定请参考

```
View.setFlags() */ if ((mPrivateFlags & PFLAG_SKIP_DRAW) ==  
PFLAG_SKIP_DRAW) { dispatchDraw(canvas); } else { draw(canvas); } }  
else { // 以硬件加速的方式绘制控件，本节暂不讨论 } } } else if (cache  
!= null) {..... // 使用绘图缓存绘制控件}/* ⑨ 恢复Canvas的状态到一切  
开始之前。于是Canvas便回到父控件的坐标系。于是父控件的dispat-  
chDraw()便可以将这个Canvas交给下一个子控件的draw(ViewGroup,  
Canvas, long)方法 */if (restoreTo >= 0) {  
canvas.restoreToCount(restoreTo); }.....// more来自动画计算，倘若动画仍  
继续，则 more为true return more; }
```

此方法比较复杂，本节着重讨论两方面的内容，分别是坐标系变换与控件的最终绘制。

首先是坐标系变换。将Canvas从父控件的坐标系变换到子控件的坐标系依次需要变换如下参数：

- 控件在父控件中的位置，即mLeft/Top。使用了Canvas.translate () 方法。
- 控件动画过程中所产生的矩阵。在绘制的过程中控件可能正在进行着一个或者多个动画，如ScaleAnimation、RotateAnimation、TranslateAnimation等。这些动画根据当前的时间点计算出Transformation，再将其中所包含的变换矩阵通过Canvas.concat () 方法设置给Canvas，使得坐标系发生相应变换。
- 控件自身的变换矩阵。除了动画可以产生矩阵使得控件发生旋转、缩放、位移等效果之外，View类还提供了诸如setScaleX/Y () 、setTranslationX/Y () 、setRotation/X/Y () 等方法使得控件产生上述效果。这一系列方法所设置的变换信息被整合在View.mTransformation成员变量中，并且可以通过View.getMatrix () 方法从这个成员变量中提取一个整合了所有变换信息的变换矩阵。View.draw (ViewGroup, Canvas, long) 将这个矩阵concat () 到Canvas中，使得这些方法得以产生应用的效果。

·控件内容的滚动量，即mScrollX/Y。虽说在一开始滚动量就和控件位置一起通过Canvas.translate () 进行了变换。然而在进行另外两种矩阵变换时，都会先将滚动量撤销，完成变换后再将滚动量重新应用。这说明滚动量是在4种变换因素中最后被应用的。

Canvas针对上述4个因素进行变换之后，其坐标系已经是控件自身坐标系了，接着调用draw (Canvas) 进行控件内容的绘制。于是便回到了本节所讨论的起点，draw (Canvas) 绘制了控件的背景，通过onDraw () 绘制了控件的内容，并且通过它的dispatchDraw () 方法将绘制工作延伸到属于它的每一个子控件。

4.以软件方式绘制控件树的完整流程

前三个小节从View.draw (Canvas) ，到ViewGroup.dispatchDraw () ，再到子控件的draw (ViewGroup, Canvas, long) ，以及子控件的View.draw (Canvas) ，构成了一个从根控件开始沿着控件树的递归调用。于是便可以将控件树绘制的完整流程归纳出来。如图6-16所示，ViewRootImpl将mScrollY以translate变换的方式设置到Canvas之后，Canvas便位于根控件的坐标系之中，接下来便通过View.draw (Canvas) 方法绘制根控件的内容。根控件的dispatchDraw () 方法会将绘制工作延续给子控件的View.draw (ViewGroup, Canvas, long) ，这个方法首先以4个变换因素对Canvas进行坐标系变换，使得Canvas进入此控件的坐标系，然后调用View.draw (Canvas) 进行绘制，接着此

控件的dispatchDraw（）又会将绘制工作延续给子控件。如此一来，绘制工作便沿着控件树传递给每一个控件。

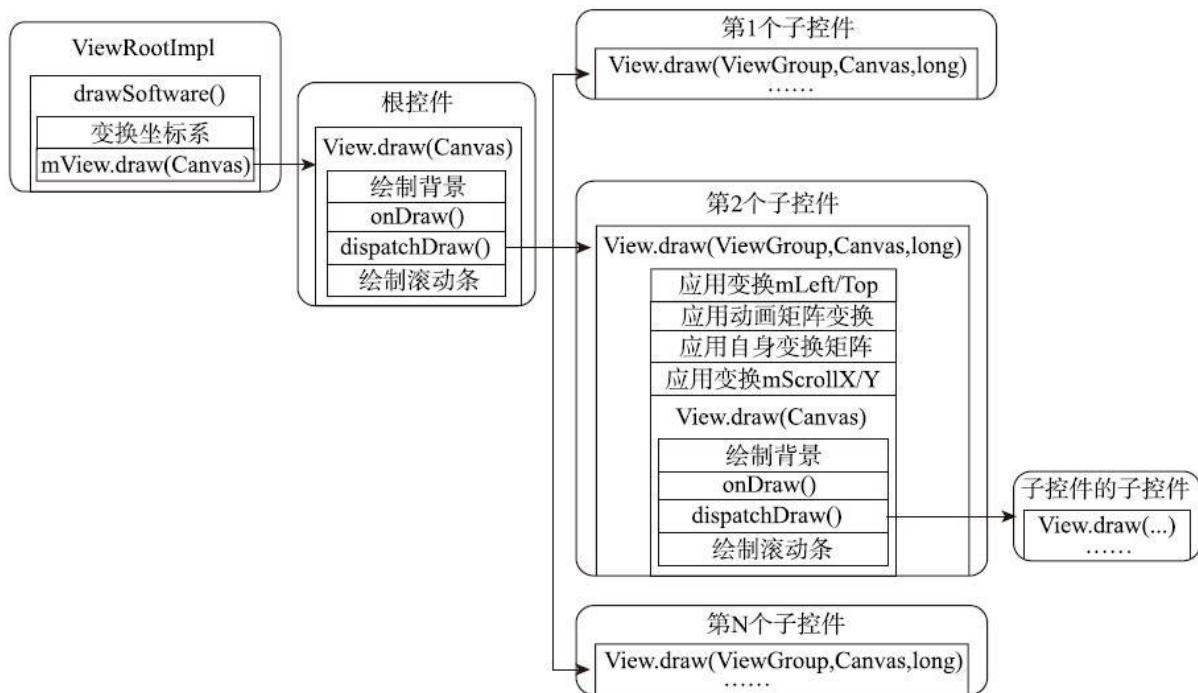


图6-16 控件树绘制的完整流程

在整个绘制过程中，`dispatchDraw()` 是使得绘制工作得以在父子控件之间延续的纽带，`draw (ViewGroup, Canvas, long)` 是准备坐标系的场所，而`draw (Canvas)` 则是实际绘制的地方。

另外，留意在图6-16所描述的整个绘制流程中，各个控件都使用了同一个`Canvas`，并且它们的内容通过这个`Canvas`直接绘制到了Surface之上，图6-17描述了这一特点。在随后讨论硬件加速与绘图缓存时将会看

到与之结构类似但又有所不同的图。将它们进行对比将有助于深刻理解这几种不同的绘制方式之间的异同以及优缺点。

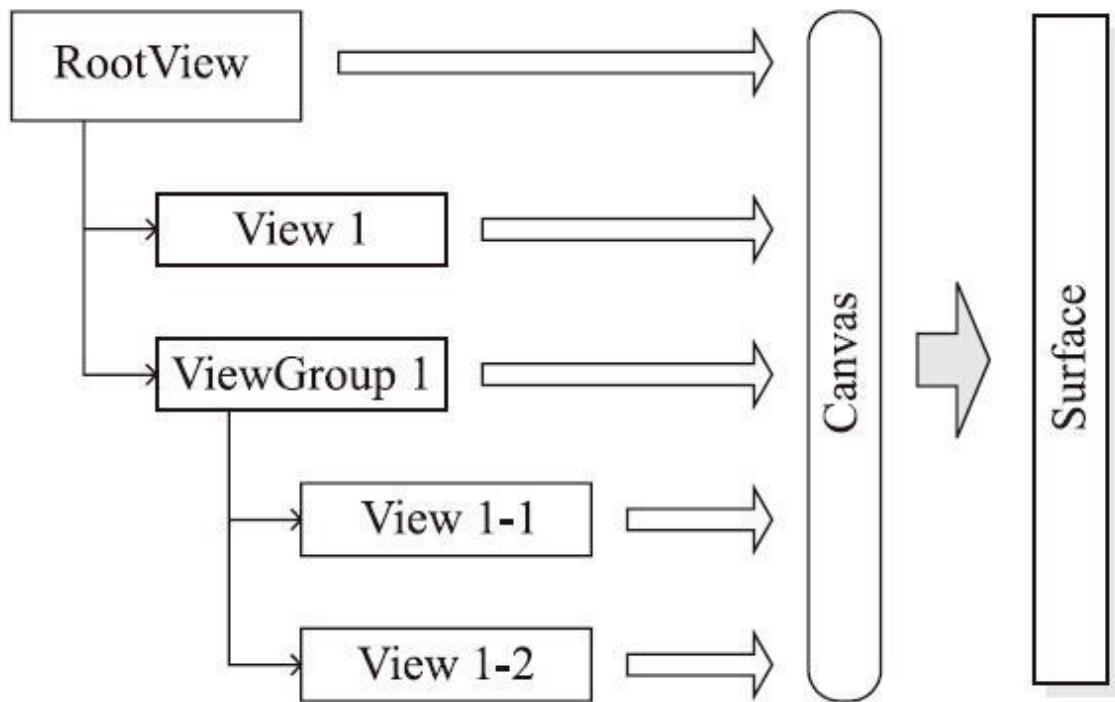


图6-17 软件绘制的流程特点

6.4.5 硬件加速绘制的原理

在6.4.3节对ViewRootImpl.draw () 方法的分析中可以看到，如果 `mAttachInfo.mHardwareRenderer` 存在并且有效，则会选择使用硬件加速的方式绘制控件树。相对于软件绘制，硬件加速绘制可充分利用 GPU 的性能，极大地提高了绘制效率。

mAttachInfo.mHardwareRenderer存在并有效的条件是：窗口的LayoutParams.flags中包含FLAG_HARDWARE_ACCELERATED。开发者可以在AndroidManifest.xml中的application节点或activity节点中声明hardwareAccelerated属性为true，使得属于application或activity的窗口获得FLAG_HARDWARE_ACCELERATED标记，进而启用硬件加速进行绘制。另外，通过绑定在一个现有窗口的WindowManager（参考6.2节）所创建的子窗口会自动继承父窗口的这一标记。



说明

在AndroidManifest.xml中声明hardwareAccelerated属性为true，这会使得从Package-Manager中所解析出来的ActivityInfo.flags中包含FLAG_HARDWARE_ACCELERATED标记。在初始化对应的Activity时，会根据从AMS处获取的ActivityInfo.flags选择是否为其窗口添加FLAG_HARDWARE_ACCELERATED标记。详情请参考Activity.attach()方法的实现。

本节将讨论硬件加速绘制的原理以及与软件绘制的异同。

1. 硬件加速绘制简介

倘若窗口使用了硬件加速，则ViewRootImpl会创建一个HardwareRenderer并保存在mAttachInfo中，因此，首先需要理解HardwareRenderer是什么。顾名思义，HardwareRenderer是用于硬件加速的渲染器，它封装了硬件加速的图形库，并以Android与硬件加速图形库的中间层的身份存在。它负责从Android的Surface生成一个HardwareLayer，供硬件加速图形库作为绘制的输出目标，并提供了一系列工厂方法用于创建硬件加速绘制过程中所需的DisplayList、HardwareLayer、HardwareCanvas等工具。正如6.3.2节所述，HardwareRenderer中间层的身份通过它所提供的updateSurface()、setup()等（用于向OpenGL ES提交Surface的变化）得到了很好的体现。

注意，HardwareRenderer及DisplayList、HardwareLayer、HardwareCanvas等都是抽象类，它们抽象了在Android控件系统中进行硬件加速绘制的操作，以便选择在不同硬件加速图形库的情况下控件系统可以无差别地使用它们。目前Android使用OpenGL ES 2.0进行硬件加速的绘制，因此上述抽象类的实际实现者分别为GL20Renderer、GLES20DisplayList、GLES20Layer、GLES20Canvas。在这套实现中，Android的Surface将会被封装为一个EGLSurface用作输出目标。

显而易见，倘若需要使用另外一个不同的硬件加速图形库则需要基于这个图形库实现一整套HardwareRenderer、DisplayList、HardwareLayer

以及HardwareCanvas。不过目前Android中仅有OpenGL ES这一套实现。



说明

从HardwareRenderer的继承关系可以体现Android组织不同硬件加速图形库的方式。HardwareRenderer的直接子类其实是GLRenderer，而GLRenderer的子类才是GL20-Renderer。因此第一级的子类(GLRenderer)体现了图形库的种类，而第二级的子类(GL20Renderer)则体现了图形库的版本。

另外，HardwareCanvas是Canvas的一个子类，它提供了诸如drawDisplayList ()、drawHardwareLayer () 等硬件加速绘制过程中所特有的操作。而HardwareCanvas的子类则将Canvas类中的操作重定向到了硬件加速图形库中。例如，GLES20Canvas重写了drawColor ()，使得OpenGL ES 2.0负责完成这个操作，而不是使用Skia软件图形库。

在本节后续的内容中读者将会看到这些工具如何在控件绘制中大展身手。

2.硬件加速绘制的入口HardwareRenderer.draw ()

在6.4.3节中介绍硬件加速绘制与软件绘制分道扬镳的地方是ViewRootImpl.draw () 方法。在这个方法中，硬件加速绘制通过以下代码完成：

```
[ViewRootImpl.java-->ViewRootImpl.draw()] private void draw(boolean fullRedrawNeeded) {if (!dirty.isEmpty() || mIsAnimating) { if (attachInfo.mHardwareRenderer != null && attachInfo.mHardwareRenderer.isEnabled()) { ..... // 硬件绘制采用的是 HardwareRenderer.draw()方法 if (attachInfo.mHardwareRenderer.draw(mView, attachInfo, this, animating ? null : mCurrentDirty)) {.....} // 软件绘制 } else if (!drawSoftware(surface, attachInfo, yoff, scalingRequired, dirty)) { return; } } }
```

由于HardwareRenderer是一个抽象类，因此draw () 方法由其子类GLRenderer实现。参考如下代码：

```
[HardwareRenderer.java-->GLRenderer.draw()] boolean draw(View view, View.AttachInfo attachInfo, HardwareDrawCallbacks callbacks, Rect dirty) {if (canDraw()) { ..... // ① 将EGLSurface设置为OpenGL ES当前的输出目标 final int surfaceState = checkCurrent(); if (surfaceState != SURFACE_STATE_ERROR) { // ② 获取对应的HardwareCanvas
```

HardwareCanvas canvas = mCanvas; attachInfo.mHardwareCanvas = canvas; int saveCount = 0; try { // 如果根控件被invalidate过，则标记它随后需要刷新其DisplayList view.mRecreateDisplayList = (view.mPrivateFlags & View.PFLAG_INVALIDATED) == View.PFLAG_INVALIDATED; DisplayList displayList; // ③ 获取根控件的DisplayList try { displayList = view.getDisplayList(); } finally { } /* onPreDraw()方法用于向HardwareCanvas设置dirty区域。与软件绘制时通过在创建Canvas时指定dirty区域不同，HardwareCanvas是一直存在的，因此需要采取不同的方式设置dirty区域 */ try { status = onPreDraw(dirty); } finally { } /* 之后的代码将有可能对Canvas进行变换操作，因此首先保存其状态并在绘制完成后恢复，以便不影响下次绘制 */ saveCount = canvas.save(); /* ④ 由callbacks在绘制前进行一些必要的操作。Callbacks其实是ViewRootImpl。它在onHardwarePreDraw()中将会调用Canvas.translate()以设置Y方向上的滚动量，在软件绘制的drawSoftware()中也有过这个操作。由于硬件绘制由HardwareRender托管，因此这一操作只能以回调方式完成 */ callbacks.onHardwarePreDraw(canvas); if (displayList != null) { // ⑤ 绘制根控件的DisplayList try { status |= canvas.drawDisplayList(displayList, mRedrawClip, DisplayList.FLAG_CLIP_CHILDREN); } finally { } } else { } } finally { /* ⑥ 由callbacks（即ViewRootImpl）进行绘制后的工作。在这

```
里ViewRootImpl将绘制ResizeBuffer动画 */
callbacks.onHardwarePostDraw(canvas); // 恢复Canvas到绘制前的状态
canvas.restoreToCount(saveCount); view.mRecreateDisplayList = false; .....
} ..... if ((status & DisplayList.STATUS_DREW) ==
DisplayList.STATUS_DREW) { ..... // ⑦ 发布绘制的内容。与软件绘制
中的Surface.unlockCanvasAndPost()工作一致
sEgl.eglSwapBuffers(sEglDisplay, mEglSurface); ..... } ..... return dirty ==
null; } }return false; }
```

以之前所探讨过的drawSoftware () 中的4个主要工作作为对比来看
HardwareRenderer.draw () 方法的实现：

- 获取Canvas。不同于软件绘制时用Surface.lockCanvas () 新建一个
Canvas， HardwareCanvas在HardwareRenderer创建之初便已被创建并绑
定在由Surface创建的EGLSurface上。
- 对Canvas进行变换以实现滚动效果。由于硬件绘制的过程位于
HardwareRenderer内部，因此ViewRootImpl需要在onHardwarePreDraw
() 回调中完成这个操作。
- 绘制控件内容。这是硬件加速绘制与软件绘制最根本的区别。软件绘
制是通过View.draw () 以递归的方式将整个控件树用给定的Canvas直
接绘制在Surface上。而硬件加速绘制则先通过View.getDisplayList ()

获取根控件的DisplayList，然后再将这个DisplayList绘制在Surface上。通过View.getDisplayList () 所获取的DisplayList中包含了已编译过的用于绘制整个控件树的绘图指令。如果说软件绘制是直接绘制，那么硬件加速绘制则是通过DisplayList间接绘制。获取DisplayList将是本节所重点讨论的内容。

·将绘制结果显示出来。硬件加速绘制通过sEgl.swapBuffers () 将绘制内容显示出来。其本质与Surface.unlockCanvasAndPost () 方法一致，都是通过ANativeWindow ::queue-Buffer () 将绘制内容发布给SurfaceFlinger。

除了上述4个主要的不同以外，硬件加速绘制还有一个可选的工作，即在onHard-warePostDraw () 回调中由ViewRootImpl完成ResizeBuffer的绘制工作。



说明

关于ResizeBuffer动画，在6.3.2节介绍的performTraversals () 方法中曾经提到过，倘若在performTraversals () 方法调用之前（窗口的resize () 回调）或在其调用过程中（relayoutWindow () 方法）使得窗口的

ContentInsets发生了变化，performTraversals () 会以ContentInsets变化前的布局将控件树绘制在一个名为mResizeBuffer的HardwareLayer上，并启动ResizeBuffer动画（ResizeBuffer动画进行的条件是mResizeBuffer不为null）。

在ViewRootImpl.draw () 内进行绘制之前，会先根据时间点计算ResizeBuffer动画的透明度。如下所示：

```
if (mResizeBuffer != null) { long deltaTime = SystemClock.uptimeMillis() -  
    mResizeBufferStartTime; if (deltaTime < mResizeBufferDuration) { float  
    amt = deltaTime/(float) mResizeBufferDuration; amt =  
    mResizeInterpolator.getInterpolation(amt); animating = true; resizeAlpha =  
    255 - (int)(amt*255); } else { disposeResizeBuffer(); } } mResizeAlpha =  
    resizeAlpha;
```

这段代码的意义是在ResizeBuffer持续的时间内计算一个随时间流逝而递减透明度并保存在mResizeAlpha中。而当前时间点超过了ResizeBuffer的持续时间后则通过销毁mResizeBuffer以结束ResizeBuffer动画。

而在本节所介绍的onHardwarePostDraw () 回调中，ViewRootImpl做了如下工作：

```
if (mResizeBuffer != null) { mResizePaint.setAlpha(mResizeAlpha);  
    canvas.drawHardwareLayer(mResizeBuffer, 0.0f, mHardwareYOffset,  
    mResizePaint);} 
```

就是说，在完成控件树的实际绘制之后，ViewRootImpl在实际绘制的内容之上将mResizeBuffer（即ContentInsets）改变前的控件树的快照以动画计算出的透明度绘制下来。因此当状态栏的可见性发生变化时，用户可以看到一个渐变的动画效果使控件树以一种布局过渡到另一种布局。

3.DisplayList的创建与渲染

经过前面的分析可以看出硬件加速绘制与软件绘制的根本不同。除了绘制工具以外，便是硬件加速绘制使用了DisplayList进行间接绘制。总体来看，硬件加速绘制过程中的View.getDisplayList（）与HardwareCanvas.drawDisplayList（）的组合相当于软件绘制过程中的View.draw（）。因此在学习getDisplayList（）时可以与View.draw（）进行类比。

DisplayList的作用是录制来自HardwareCanvas的指令，但是使用录制这个词并不是很好理解。由于它在使用上与一个Bitmap没有太大区别，因此后文在不影响准确性的情况下将称其录制过程为渲染。

参考getDisplayList（）的代码：

```
[View.java-->View.getDisplayList()] public DisplayList getDisplayList() {/*
通过getDisplayList()的另外一个重载获取一个渲染过的DisplayList。注意第二个参数永远为false */mDisplayList = getDisplayList(mDisplayList,
false);return mDisplayList; }
```

getDisplayList () 使用了下面这个重载完成DisplayList的创建与渲染：

```
private DisplayList getDisplayList(DisplayList displayList, boolean isLayer)
```

这个重载接受一个DisplayList，以及一个布尔型变量作为参数，并返回经过渲染后的DisplayList。此重载看似多此一举，其实不然。它的无参版本的含义是获取此控件本身经过渲染后的DisplayList，而第二个重载则是将控件的内容渲染到任何一个给定的DisplayList上，isLayer参数确定给定的DisplayList是否来自一个绘图缓存。关于绘图缓存的内容将会在6.4.6节介绍，读者此时只须了解获取控件本身的DisplayList时，其isLayer参数永远为false即可。

另外，当控件第一次使用硬件加速进行绘制时，其mDisplayList为null。在这种情况下，getDisplayList () 的有参重载会创建一个DisplayList并返回，而其无参重载则会将这个新创建的DisplayList保存在mDisplayList成员变量中。

接下来看下geDisplayList () 有参版本的实现，这里暂时忽略isLayer为true以及使用绘图缓冲的情况：

[View.java-->View.getDisplayList()] private DisplayList
getDisplayList(DisplayList displayList, boolean isLayer) /* 进行
DisplayList的创建与渲染的条件如下： 1> mPrivateFlags不存在
PFLAG_DRAWING_CACHE_VALID标记，表示此控件的绘图缓存无
效，此时需要 重新渲染其绘图缓存，本节暂不讨论相关内容 2>
displayList == null， 表示此控件是第一次使用硬件加速进行绘制，因此
需要创建一个Display- List并对其进行渲染 3> displayList.isValid()， 当
控件从控件树上拿下时， 此displayList会被标记为invalid,当其 重新回到
控件树时， 需要对DisplayList进行重新渲染 4> mRecreateDisplayList为
true， 正如在GLRenderer.draw()中所见， 当控件被invalidate之后，
mRecreateDisplayList会被置为true， 因此需要进行重新渲染 */if
((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 ||
displayList == null || !displayList.isValid() || (!isLayer &&
mRecreateDisplayList))) { /* ① 如果参数中没有提供DisplayList，则
创建一个DisplayList。在本例中，这种情况对应着控件第一次使用硬
件加速进行绘制 */ if (displayList == null) { final String name =
getClass().getSimpleName(); /* 使用HardwareRenderer的工厂方法创建一
个DisplayList，在目前的版本中它返回的Dis- playList的类型为
GLES20DisplayList */ displayList =
mAttachInfo.mHardwareRenderer.createDisplayList(name); } /* ② 获
取绑定在DisplayList上的HardwareCanvas。DisplayList.start()方法将使

Dis- playList返回一个HardwareCanvas，并准备好从此HardwareCanvas中录制绘图操作 */ final HardwareCanvas canvas = displayList.start(); try {
// 设置Canvas的坐标系为(0,0,width,height) canvas.setViewport(width,
height); /* 通过onPreDraw()设置dirty区域为null，即完整重绘。所以
DisplayList的渲染是不会考虑dirty区域的 */ canvas.onPreDraw(null); int
layerType = getLayerType(); if (!isLayer && layerType !=
LAYER_TYPE_NONE) { /* layerType指定了此控件所使用的绘图缓冲的
类型，不为LAYER_TYPE_NONE时表示此控件使用了某种类型的绘图
缓冲，本节不讨论此内容 } else { // ③ 计算并设置控件的滚动量
computeScroll(); canvas.translate(-mScrollX, -mScrollY); // 接下来的
这段代码在View.draw(ViewGroup,Canvas,long)中见到过 if
((mPrivateFlags & PFLAG_SKIP_DRAW) == PFLAG_SKIP_DRAW) { /*
跳过本控件的绘制，直接绘制子内容。以此作为无背景ViewGroup绘制
时的一条 节省开销的快速通道 */ dispatchDraw(canvas); } else { // ④
draw()方法将控件自身的内容绘制到HardwareCanvas draw(canvas); } } }
finally { // ④ 结束录制。至此为止，通过View.draw()方法对
HardwareCanvas进行的绘制工作都被 displayList录制并编译优化为一系
列的绘图指令。DisplayList的渲染完成，而先前所创 建的
HardwareCanvas也被回收 */ displayList.end(); } } // 将渲染过的
DisplayList返回给调用者return displayList; }

getDisplayList () 的实现体现了DisplayList的使用方法。DisplayList渲染与Surface的绘制十分相似，分为如下三个步骤：

- 通过DisplayList.start () 创建一个HardwareCanvas并准备好开始录制绘图指令。
- 使用HardwareCanvas进行与Canvas一样的变换与绘制操作。
- 通过DisplayList.end () 完成录制并回收HardwareCanvas。

getDisplayList () 还体现了另一个重要信息，即DisplayList的渲染也是使用我们所熟悉的View.draw () 方法完成的，而且View.draw () 方法的实现在硬件加速下与软件绘制下完全一样。

另外，这一方法还体现了另一个与软件绘制的重要区别：软件绘制的整个过程都使用了来自Surface.lockCanvas () 的同一个Canvas；而在硬件加速时，控件使用由自己的DisplayList所产生的Canvas进行绘制。在这种情况下，每个控件onDraw () 方法的Canvas参数各不相同。

还需注意在getDisplayList () 中进行了滚动量的变换，因此在硬件加速绘制的情况下，View.draw (ViewGroup, Canvas, long) 方法不需要进行滚动量的变换，因而这个方法的流程与软件绘制时的流程会有差异。

4.硬件加速绘制下的子控件绘制

DisplayList的实际渲染由View.draw () 完成，可以很自然地联想到硬件加速绘制与软件绘制进入了统一绘制流程，然而得出这个结论为时尚早。首先回顾一下软件绘制时的总体流程：

- View.draw (Canvas) 绘制控件本身的内容，并引发dispatchDraw () 的调用以绘制子控件。
- ViewGroup.dispatchDraw () 根据特定的顺序依次调用子控件的 View.draw (ViewGroup, Canvas, long) 。
- 子控件的View.draw (ViewGroup, Canvas, long) 根据控件的4个坐标系变换因素将Canvas坐标系从父控件变换到子控件，然后调用其 View.draw (Canvas) 将该坐标系及其子控件绘制到Canvas上。

硬件加速绘制与软件绘制在前两步是完全相同的。区别在于第三步，即在View.draw (ViewGroup, Canvas, long) 的流程上，二者几乎完全不同。产生不同的根本原因在于硬件加速绘制希望在Canvas上绘制子控件的DisplayList，而不是使用View.onDraw () 直接绘制。这一需求也直接导致了坐标系变换的方法有了不小的差异。本节将讨论此种情况下View.draw (ViewGroup, Canvas, long) 的工作方式。参考如下有关代码：

```
[View.java-->View.draw(ViewGroup,Canvas,long)] boolean draw(Canvas  
canvas, ViewGroup parent, long drawingTime) { // ① 此方法通过
```

useDisplayListProperties决定是否将变换设置在DisplayList上boolean
useDisplayListProperties = mAttachInfo != null &&
mAttachInfo.mHardwareAccelerated;.....// ② hardwareAccelerated表示
Canvas是否是一个HardwareCanvasfinal boolean hardwareAccelerated =
canvas.isHardwareAccelerated();if ((flags &
ViewGroup.FLAG_CHILDREN_DRAWN_WITH_CACHE) != 0 || (flags &
ViewGroup.FLAG_ALWAYS_DRAWN_WITH_CACHE) != 0) {} else
{ /* ③ caching表示是否使用缓存。在这里DisplayList也被认为是一种广
义上的缓存。因此caching被 设置为true */ caching = (layerType !=
LAYER_TYPE_NONE) || hardwareAccelerated;}..... // 动画计算
DisplayList displayList = null;// ④ hasDisplayList表示此控件是否拥有
DisplayList。这取决于HardwareRenderer是否可用boolean hasDisplayList
= false;if (caching) { if (!hardwareAccelerated) { // 生成软件绘制下的
绘图缓存 } else { switch (layerType) { // 硬件加速绘制下，根据绘图
缓存类型的不同选择不同的操作 case LAYER_TYPE_NONE: // 在
HardwareRenderer可用的情况下， hasDisplayList为true hasDisplayList =
canHaveDisplayList(); break; } } } // 仅当控件拥有缓存时才可以将变换应
用到DisplayList上useDisplayListProperties &= hasDisplayList;if
(useDisplayListProperties) { // ⑤ 通过getDisplayList()获取本控件经过渲
染的DisplayList displayList = getDisplayList(); if (!displayList.isValid()) {
// 倘若获取displayList失败，则在后续的流程中使用软件绘制

```
displayList = null; hasDisplayList = false; useDisplayListProperties = false;
} }.....// ⑥ hasNoCache表示并非在软件绘制下使用软件绘图缓存。硬件
加速模式下它永远为truefinal boolean hasNoCache = cache == null ||
hasDisplayList;// 倘若有动画正在执行并产生了一个变换矩阵if
(transformToApply != null || ..... ) { if (transformToApply != null ||
!childHasIdentityMatrix) { ..... if (transformToApply != null) { if
(concatMatrix) { if (useDisplayListProperties) { // ⑦ 将变换矩阵设置到
DisplayList中
displayList.setAnimationMatrix(transformToApply.getMatrix()); } else {
..... // 软件绘制时，在Canvas中应用动画所产生的变换矩阵 } } ..... }
.....}..... // 设置控件的剪裁区域if (hasNoCache) { boolean layerRendered
= false; ..... // 绘制硬件绘图缓存，倘若没有使用绘图缓存，则
layerRendered保持为false if (!layerRendered) { if (!hasDisplayList) { ..... //
软件绘制 } else { // ⑧ drawDisplayList()将控件的DisplayList绘制在给定
的Canvas上 ((HardwareCanvas) canvas).drawDisplayList(displayList, null,
flags); } } } else if (cache != null) {.....}.....return more; }
```

View.draw (ViewGroup, Canvas, long) 方法同时兼顾了软件绘制、硬件加速绘制以及使用绘图缓存等情况，因此其实现内部的分支非常复杂。如下几个局部变量的取值保证了此方法以硬件加速绘制的方式执行：

- useDisplayListProperties，表示应当通过DisplayList的成员函数来设置坐标系变换，而不是像软件加速那样通过Canvas的变换指令完成。
- hardwareAccelerated，表示Canvas是否是一个HardwareCanvas。
- hasDisplayList，表示控件是否可能拥有一个DisplayList。这个值取决于Hardware-Renderer是否可用。
- caching以及hasNoCache，是与绘图缓冲相关的两个条件变量。由于DisplayList被视为广义上的一种缓存，因此在硬件加速绘制时caching为true，而DisplayList又不属于真正意义上的缓存，因此hasNoCache为false。

一般情况下，前三个变量的取值是一致的，全部为true则为硬件加速绘制，而全部为false则为软件绘制。

对比软件绘制情况下draw（ViewGroup，Canvas，long）的工作，不难发现它使用View.getDisplayList（）与HardwareCanvas.drawDisplayList（）的组合取代了直接使用View.draw（）。View.draw（）被View.getDisplayList（）间接调用。

再看坐标系的变换。由于useDisplayListProperties为true，软件绘制时对Canvas的坐标系变换全部被绕过了，取而代之的是通过DisplayList的相关方法进行设置。在6.4.4节曾经介绍过，从父控件坐标系变换到控件

坐标系需要应用4个变换因素，分别为控件位置、动画矩阵、自身变换矩阵以及滚动量。但是在View.draw (ViewGroup, Canvas, long) 中只看到通过DisplayList.setAnimationMatrix () 设置了动画矩阵，并且在View.getDisplayList () 的分析中看到滚动量通过与DisplayList绑定的HardwareCanvas.translate () 进行了变换。那么另外两个变换因素是在哪里被应用的呢？回到View.getDisplayList () 方法的有参重载，可以看到在完成了View.draw () 之后有如下的操作：

```
[View.java-->View.getDisplayList()]
private DisplayList
getDisplayList(DisplayList displayList, boolean isLayer) { .....try { .....
draw(canvas); // 将控件内容绘制到DisplayList上} finaly { ..... // 重点所
在 : setDisplayListProperties做了什么 ?
setDisplayListProperties(displayList); } }
```

再看setDisplayListProperties () 的代码：

```
void setDisplayListProperties(DisplayList displayList) {if (displayList != null) { // ① 设置DisplayList的位置和尺寸。注意位置是控件的位置
displayList.setLeftTopRightBottom(mLeft, mTop, mRight, mBottom); ..... if
(mTransformationInfo != null) { ..... /* ② 设置控件自身的变换信息到
DisplayList。控件自身的变换矩阵就是来自于mTransfor- mationInfo，即
这种设置和View.getMatrix()是等效的 */
displayList.setTransformationInfo(alpha,
```

```
mTransformationInfo.mTranslationX, mTransformationInfo.mTranslationY,  
mTransformationInfo.mRotation, mTransformationInfo.mRotationX,  
mTransformationInfo.mRotationY, mTransformationInfo.mScaleX,  
mTransformationInfo.mScaleY);} }
```

原来不止滚动量，控件的位置以及自身的变换矩阵也实现于
View.getDisplayList () 之中。由此总结硬件加速绘制下View.draw
(ViewGroup, Canvas, long) 与软件绘制下的不同之处在于：

- 变换因素的应用方法不同。软件绘制时通过Canvas的变换操作将坐标系变换到子控件自身坐标系。而硬件加速绘制时Canvas的坐标系仍保持在父控件的坐标系下，然后通过DisplayList的相关方法将变换因素设置给DisplayList，HardwareCanvas.drawDisplayList () 会按照这些变换因素再以这些变换绘制（准确地说是回放）DisplayList。

- 绘制方法不同。软件绘制时可说是直接绘制。硬件加速绘制时使用的是View.getDisplayList () 与HardwareCanvas.drawDisplayList () 的组合进行间接绘制。

正如如软件绘制的流程一样，View.getDisplayList () 中所调用的View.draw (Canvas) 会继续调用ViewGroup.dispatchDraw () 将绘制工作延伸到下一级的子控件，如此递归下去直到完成整个控件树的绘制。

5.硬件加速绘制的总结

相对于软件绘制，硬件加速绘制的流程要复杂一些。其根本原因在于硬件加速绘制是间接绘制，即在讨论的软件绘制的递归流程之上增加了一部分额外的操作，图6-18体现了硬件加速绘制在递归方式上的差异。

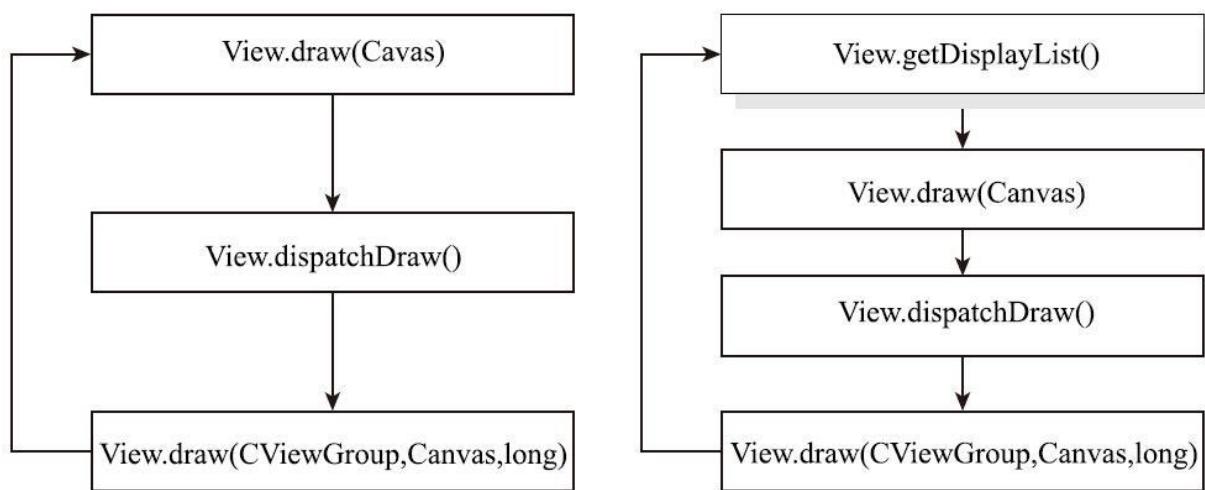


图6-18 软件绘制与硬件加速绘制的递归方式的差异

`View.getDisplayList ()` 这一额外操作使得硬件加速绘制拥有如图6-19所示的特点，其中灰色方块表示一个DisplayList对象。

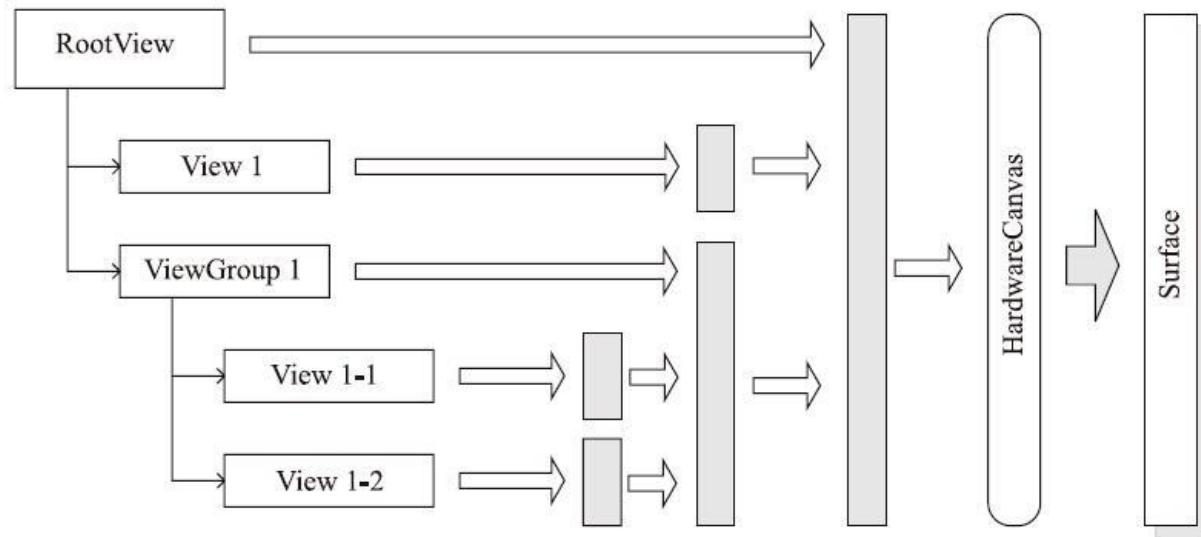


图6-19 硬件加速绘制的流程特点

从图6-19可以看出，控件内容不再是通过Canvas直接绘制到Surface，而是绘制到DisplayList。同时，子控件的DisplayList会被绘制到父控件的DisplayList。最终整个控件树的内容被集合在根控件的DisplayList，并且这个DisplayList通过HardwareCanvas绘制到Surface之上。

6.4.6 使用绘图缓存

绘制是一个消耗软硬件资源的工作，如果有可能Android希望能够尽可能地减少绘制工作，以减少电量的使用以及提高其动画的流畅性。考虑用户拖动一个ListView的过程中，其内容每次移动位置都需要ListView及其列表项进行一次完整绘制。尤其是当列表项是一个复杂的控件树时，一次完整绘制的开销是很大的。而实际上ListView的每一个列表项的内容在拖动过程中往往都是不会发生变化的，变化的仅仅是

其位置而已。因此是否可以省略掉列表项的绘制过程以减少开销呢？绘图缓存就是解决此问题的一个方案。

所谓绘图缓存是指一个Bitmap或一个HardwareLayer，它保存了控件及其子控件的一个快照。当控件的父控件需要重绘它时，可以考虑将其绘图缓存绘制Canvas上，而不是执行其完整的绘制流程，以此节省此控件的绘制开销。这一机制在控件及其子控件所构成的控件树十分庞大或者其重绘需要进行复杂计算时尤为有效。

绘图缓存有两种类型，即软件缓存（Bitmap）和硬件缓存（HardwareLayer）。开发者可以通过View.setLayerType（）为LAYER_TYPE_SOFTWARE和LAYER_TYPE_HARDWARE决定此控件使用哪种类型的缓存。在默认情况下，控件的缓存类型为LAYER_TYPE_NONE，即不使用缓存机制。尽管开发者可以随意设置绘图缓存的类型，但是由于硬件缓存HardwareLayer的渲染依赖于HardwareCanvas，因此在软件绘制的情况下，缓存类型被设置为LAYER_TYPE_HARDWARE的控件仍然会选择使用软件缓存。而在硬件加速绘制的情况下，可以在硬件缓存和软件缓存中任选其一。

另外，View.setLayerType（）除了接受缓存类型作为参数之外，还接受一个Paint类型的参数，用于指示控件的绘图缓存将以何种效果绘制在Canvas上。所支持的Paint的效果有透明度、Xfermode以及ColorFilter。

因此绘图缓存除了可以用来提高效率之外，还可以用来实现一些显示效果。

1. 软件绘制下的软件缓存

首先讨论软件绘制下的软件缓存的工作原理。经过之前关于绘制原理的讨论，不难联想到绘图缓存的相关操作位于View.draw(ViewGroup, Canvas, long) 方法内。参考相关代码：

```
[View.java-->View.draw(ViewGroup,Canvas,long)] boolean draw(Canvas  
canvas, ViewGroup parent, long drawingTime) {.....// ① 首先获取其缓存  
类型int layerType = getLayerType();final boolean hardwareAccelerated =  
canvas.isHardwareAccelerated();if ((flags &  
ViewGroup.FLAG_CHILDREN_DRAWN_WITH_CACHE) != 0 || (flags &  
ViewGroup.FLAG_ALWAYS_DRAWN_WITH_CACHE) != 0) { ..... //  
ViewGroup会强制子控件使用缓存进行绘制，本节最后再讨论这个话  
题} else { // ② 如果缓存类型不为NONE则表示启用缓存 caching =  
(layerType != LAYER_TYPE_NONE) || hardwareAccelerated;}..... // 动画  
处理// cache是一个Bitmap类型的变量，用于保存软件缓存Bitmap cache  
= null;if (caching) { // 软件绘制时的缓存处理 if (!hardwareAccelerated) {  
if (layerType != LAYER_TYPE_NONE) { // 软件绘制情况下不支持硬件  
缓存，因此将强制使用软件缓存 layerType =  
LAYER_TYPE_SOFTWARE; // ③ buildDrawingCache() 将在必要时刷新
```

缓存的内容 buildDrawingCache(true); } // ④ 通过getDrawingCache()获取控件的软件缓存，并保存在cache中 cache = getDrawingCache(true); }

else { // 硬件加速下的缓存处理 } }...../* 接下来这段代码需要留意，在使用绘图缓存时不会进行滚动量的变换。这是为什么呢？因为滚动量的变换操作在生成绘图缓存时便已经完成了。绘图缓存作为控件内容的快照，应该如实地反映 控件当前的模样，例如一个ListView的快照应当能够反映其滚动的位置。因此在这里便不可以再对 滚动量做变换

```
    */final boolean offsetForScroll = cache == null && !hasDisplayList && layerType != LAYER_TYPE_HARDWARE;.....if (offsetForScroll) { canvas.translate(mLeft - sx, mTop - sy);} else { if (!useDisplayListProperties) { canvas.translate(mLeft, mTop); }}.....final boolean hasNoCache = cache == null || hasDisplayList;..... // 冗长的坐标系变换操作if (hasNoCache) { ..... // 硬件缓存操作与无缓存绘制的代码} else if (cache != null) { ..... /* ⑤ 将软件缓存cache绘制到Canvas上。其中 cacePaint保存了View.setLayerType()时所传 入的参数 */ canvas.drawBitmap(cache, 0.0f, 0.0f, cachePaint); }.....return more; }
```

显而易见，使用软件缓存进行绘制时使用View.buildDrawingCache () /getDrawingCache () 与canvas.drawBitmap () 的组合替代无缓存模式下的View.draw (Canvas) 。此种模式是否有些似曾相识的感觉呢？猜对了，和硬件加速绘制时的处理如出一辙。类比地推测一下，在负责刷新缓存的View.buildDrawingCache () 中一定包含了对

View.draw (Canvas) 的调用。而且事实的确如此。参考
View.buildDrawingCache () 的代码：

```
[View.java-->View.buildDrawingCache()] public void  
buildDrawingCache(boolean autoScale) /* 仅当软件缓存为invalidate (伴  
随着View的invalidate) 时或控件尚未生成缓存时，此方法才会执行。  
倘若在控件没有被invalidate时仍生成软件缓存，那便失去了缓存的意  
义 */if ((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 ||  
(autoScale ? mDrawingCache == null : mUnscaledDrawingCache == null))  
{ ..... // 软件缓存有两块，区别在于mDrawingCache会根据兼容模式进  
行放大和缩小，而mUnscaledDrawingCache则反映了控件的真实尺  
寸，这两者的用途是不一样的。 mDrawingCache用于做绘制时的软件  
缓存，因为绘制到窗口时需要根据兼容模式进行缩放。而mUn-  
scaleDrawingCache则往往被用作控件截图等用途。 注意到  
View.draw(ViewGroup,Canvas,long)调用buildDrawingCache()时autoScale  
参数为true，所以mDrawingCache才是本节所讨论的那个绘图缓存 */  
Bitmap bitmap = autoScale ? mDrawingCache : mUnscaledDrawingCache;  
// 倘若缓存不存在，或者控件的最新尺寸与缓存尺寸不一致则需要新建  
一块缓存 if (bitmap == null || bitmap.getWidth() != width ||  
bitmap.getHeight() != height) { ..... if (bitmap != null) bitmap.recycle(); try  
{ // ① 新建一个缓存。可见软件缓存是一个不折不扣的普通Bitmap  
bitmap = Bitmap.createBitmap(mResources.getDisplayMetrics(), width,
```

height, quality);

bitmap.setDensity(getResources().getDisplayMetrics().densityDpi); // 保存 Bitmap到合适的成员变量中 if (autoScale) { mDrawingCache = bitmap; } else { mUnscaledDrawingCache = bitmap; } } catch (OutOfMemoryError e) {.....} } // 确保缓存已经创建之后，就要准备Canvas了 Canvas canvas; if (attachInfo != null) { // Canvas使用了mAttachInfo中的mCanvas。 AttachInfo.mCanvas可说是一个缓存。所有需要刷新软件缓存的控件都可以从这里取出这个业已创建好的Canvas进行绘制，以避免每个控件在每次刷新缓存时创建和销毁Canvas所带来的开销 */ canvas = attachInfo.mCanvas; if (canvas == null) { canvas = new Canvas(); } // ② 设置软件缓存为Canvas的绘制目标 canvas.setBitmap(bitmap); // 这是一个有趣却重要的小技巧。AttachInfo.mCanvas是一个公共的缓存。那么控件树中任何一个需要刷新软件缓存的控件都会到这个成员，而且都会通过上述setBitmap()调用将其绘制目标设置为各自的软件缓存。设想一下，如果此控件的某个子控件也使用软件缓存，那么这个Canvas的绘制目标会被这个子控件篡改，结果将是灾难性的。因此在这里将其设置为null，使其子控件不得不创建自己的Canvas。不过兄弟控件之间共用这一个Canvas是没有问题的，因为它们的绘制是串行的 */

attachInfo.mCanvas = null; } else { } // ③ 计算滚动量并对坐标系进行滚动量变换 computeScroll(); canvas.translate(-mScrollX, -mScrollY); /* ④ draw(Canvas)或是dispatchDraw()，这已是第三次见到这一熟悉

的操作了。这将会把控件及其子控件的内容绘制在Bitmap，即软件缓存之上 */ if ((mPrivateFlags & PFLAG_SKIP_DRAW) ==
PFLAG_SKIP_DRAW) { mPrivateFlags &= ~PFLAG_DIRTY_MASK;
dispatchDraw(canvas); } else { draw(canvas); } // 将Canvas放回
AttachInfo.mCanvas中，于是其他控件又可以使用这个Canvas了 if
(attachInfo != null) { attachInfo.mCanvas = canvas; } } }

果不其然，View.buildDrawingCache () 的实现方式与
View.getDisplayList () 方法几乎完全一致。只不过它的目标是一个
Bitmap而不是DisplayList。

而View.getDrawingCache () 则返回mDrawingCache或
mUnscaledDrawingCache。这个方法虽然简单，但仍然有值得讨论的地
方。参考实现如下：

```
public Bitmap getDrawingCache(boolean autoScale) {/* ①  
WILL_NOT_CACHE_DRAWING标记表示此控件不使用软件缓存。某  
些控件会添加这个标记，因为它们有意无意地已经实现类似绘图缓存  
的机制。例如ImageView，它的绘制本身就是绘制一个图片，将其绘制  
到软件缓存，再将其软件缓存绘制到Canvas上，这明显不如直接绘制  
到Canvas上来得快。参考View.setWillNotCacheDrawing()*/if  
((mViewFlags & WILL_NOT_CACHE_DRAWING) ==  
WILL_NOT_CACHE_DRAWING) { return null; }/* ②
```

DRAWING_CACHE_ENABLED表示了控件使用软件缓存。这岂不是和WILL_NOT_CACHE_DRAWING重复？二者作用类似，但是意图却不一样。WILL_NOT_CACHE_DRAWING表示控件根本不希望使用软件缓存，即便通过setLayerType()设置了一种缓存类型也不行，控件本身的性质决定了是否添加WILL_NOT_CACHE_DRAWING标记，其意图在于禁用软件缓存。而DRAWING_CACHE_ENABLED标记的意图则在于启用软件缓存，但不是在用在绘制过程中，而是用于通过getDrawingCache()进行控件截图时使用的。即便不存在这一标记，只要通过setLayerType()设置了一种缓存类型，在绘制过程中依然会采用绘图缓存的方式进行绘制。另外，ViewGroup也会通过这个方法临时启用其直接子控件的软件缓存。在本节的最后将会讨论这一话题 */if
((mViewFlags & DRAWING_CACHE_ENABLED) ==
DRAWING_CACHE_ENABLED) { buildDrawingCache(autoScale); } // 返回
mDrawingCache或mUnscaledDrawingCache
return autoScale ?
mDrawingCache : mUnscaledDrawingCache; }

只要理解了硬件加速绘制的原理，那么理解软件绘制下的软件缓存的工作原理并不困难。图6-18以及图6-19完全适用于描述软件绘制下的软件缓存的流程特点。只需要将View.getDisplayList () 换作View.buildDrawingCache ()，以及将DisplayList换作Bitmap即可。

2. 硬件加速绘制下的绘图缓存

接下来讨论硬件加速下的绘图缓存的实现原理。硬件加速绘制时，绘图缓存的实现位于View.getDisplayList () 而不是View.draw (ViewGroup, Canvas, long) 中。如果将DisplayList理解为一种缓存，那么硬件加速绘制下的绘图缓存则是在DisplayList的基础之上的另外一级缓存，即二级绘图缓存。参考View.getDisplayList () 中的相关代码，注意此时第二个参数isLayer仍然为false：

```
[View.java-->View.getDisplayList()]
private DisplayList
getDisplayList(DisplayList displayList, boolean isLayer) { .....if
(((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 ||
displayList == null || !displayList.isValid() || (!isLayer &&
mRecreateDisplayList))) { ..... // 创建DisplayList
boolean caching = false;
final HardwareCanvas canvas = displayList.start(); try { ..... // ① 获取缓存
类型 int layerType = getLayerType(); if (!isLayer && layerType != LAYER_TYPE_NONE) { // 处理硬件缓存
if (layerType == LAYER_TYPE_HARDWARE) { // ② 通过getHardwareLayer()方法获取刷新后的硬件缓存
final HardwareLayer layer = getHardwareLayer(); if (layer != null && layer.isValid()) { /* ③ 通过
HardwareCanvas.drawHardwareLayer()方法将硬件缓存绘制到
DisplayList上 */
canvas.drawHardwareLayer(layer, 0, 0, mLayerPaint); }
else {.....} // 处理软件缓存 } else { // ④ 软件缓存的处理方法与软件绘制
时的软件缓存完全一致
buildDrawingCache(true); Bitmap cache =
```

```
getDrawingCache(true); if (cache != null) { canvas.drawBitmap(cache, 0, 0,  
mLayerPaint); } } else { ..... // 在无缓存绘制的情况下，调用  
View.draw(Canvas)对DisplayList进行渲染 } } finally {.....}}return  
displayList; }
```

首先，软件缓存的处理方式与软件绘制时完全一致，因此不必赘述。仅需注意硬件加速时的软件缓存被绘制在DisplayList上即可。



注意

硬件加速绘制的判断条件是AttachInfo.mHardwareRenderer不为空并且有效。这说明硬件加速特性的启用与否是窗口级的，即对整个控件树有效，好似整个控件树中的控件都会以硬件加速的方式进行绘制。其实不然，软件缓存的存在会导致控件树的某个子树退化为软件绘制。因为当一个控件在硬件加速绘制的情况下启用软件缓存时，它的View.draw (Canvas) 方法将会在View.buildDrawingCache () 中调用，并使用AttachInfo.mCanvas进行绘制。这导致这一控件及其子控件都得使用软件Canvas进行绘制。

这一现象更体现了硬件加速绘制的复杂性，因为在一棵控件树中可能同时存在着多种绘制方式：标准的硬件加速绘制、软件绘制、软件缓存绘制以及硬件缓存绘制。

而硬件缓存的处理则引入了一个新的方法：View.getHardwareLayer()。HardwareLayer就是我们所说的硬件缓存。它的作用与View.buildDrawingCache() /getDrawingCache() 的组合是一致的，即刷新并获取绘图缓存。于是可以预料到View.getHardwareLayer() 的实现应该与buildDrawingCache() 十分类似。参考以下实现：

```
HardwareLayer getHardwareLayer() {..... // 检查硬件加速是否可用final  
int width = mRight - mLeft;final int height = mBottom - mTop;.....// 此方法  
的执行条件与View.buildDrawingCache的相同if ((mPrivateFlags &  
PFLAG_DRAWING_CACHE_VALID) == 0 || mHardwareLayer == null) {  
if (mHardwareLayer == null) { // ① 如果控件尚无硬件缓存，则通过  
HardwareRenderer创建一个 mHardwareLayer =  
mAttachInfo.mHardwareRenderer.createHardwareLayer( width, height,  
isOpaque()); ..... } else { // ② 当尺寸发生变化时，与软件缓存时必须重  
新创建不同，硬件缓存可以直接修改尺寸 if  
(mHardwareLayer.getWidth() != width || mHardwareLayer.getHeight() !=  
height) { if (mHardwareLayer.resize(width, height)) {  
mLocalDirtyRect.set(0, 0, width, height); } } ..... } ..... // 设置将硬件缓存
```

绘制到Canvas上时所使用的paint

```
mHardwareLayer.setLayerPaint(mLayerPaint); // ③ 设置用来刷新硬件缓存的DisplayList。这一DisplayList由View.getHardwareLayer-  
DisplayList()方法获得 */  
  
mHardwareLayer.redrawLater(getHardwareLayerDisplayList(mHardwareLa-  
yer), mLocalDirtyRect); ViewRootImpl viewRoot = getViewRootImpl(); //  
由ViewRootImpl通知HardwareRenderer尽快使用给定的DisplayList刷新  
硬件缓存 if (viewRoot != null)  
  
viewRoot.pushHardwareLayerUpdate(mHardwareLayer); ..... }return  
mHardwareLayer; }
```

其实现与预想之中似乎有一些不同，它并没有直接调用View.draw(Canvas) 进行绘制，而是使用View.getHardwareLayerDisplayList () 方法获取了一个DisplayList，并以这一DisplayList对HardwareLayer进行刷新。这究竟是一个什么样的DisplayList呢？参考get-HardwareLayerDisplayList () 方法的实现：

[View.java-->View.getHardwareLayerDisplayList()] private DisplayList
getHardwareLayerDisplayList(HardwareLayer layer) {/* 又一次回到
View.getDisplayList()，注意这一次getDisplayList()的参数与之前的不
同。传入 的DisplayList来自给定的HardwareLayer，而不是控件自身的
mDisplayList,并且第二个参数 isLayer的值为true，而不是以往的false

```
*/DisplayList displayList = getDisplayList(layer.getDisplayList(),
true);layer.setDisplayList(displayList);return displayList; }
```

看来View.getHardwareLayerDisplayList () 所返回的DisplayList也来自View.getDisplayList () ，只不过采用了不同的参数而已。可以断定此DisplayList中记录的就是控件的绘制结果，不过isLayer参数会为true带来什么样的效果呢？再回过头来看View.getDisplayList () 的实现，注意这一次isLayer为true：

```
[View.java-->View.getDisplayList()] private DisplayList
getDisplayList(DisplayList displayList, boolean isLayer) { .....if
(((mPrivateFlags & PFLAG_DRAWING_CACHE_VALID) == 0 ||
displayList == null || !displayList.isValid() || (!isLayer &&
mRecreateDisplayList))) { ..... // 创建DisplayList boolean caching = false;
final HardwareCanvas canvas = displayList.start(); try { ..... // ① 获取缓存
类型，注意这时类型为硬件缓存 int layerType = getLayerType(); // ② 这
次isLayer为true，即不会进行硬件缓存的绘制 if (!isLayer && layerType
!= LAYER_TYPE_NONE) { ..... // 硬件缓存或软件缓存的绘制 } else { //
③ 通过View.draw(Canvas)渲染DisplayList ..... // 无缓存绘制的情况下调
用View.draw(Canvas)对DisplayList进行渲染 } } finally {.....}}return
displayList; }
```

原来，`isLayer`最大的意图就是为了区分`getDisplayList()`的结果是用于实际绘制，还是用于刷新硬件缓存。如果是实际绘制（`isLayer`为`false`），则会根据`getLayerType()`的返回值确定采用硬件缓存、软件缓存或`View.draw(Canvas)`之一对`DisplayList`进行渲染。如果是用于刷新硬件缓存（`isLayer`为`true`），则仅使用`View.draw(Canvas)`进行渲染。因此在启用硬件缓存之后，`View.getDisplayList()`会被调用两次，第一次调用是为了渲染控件自身的`DisplayList`，其渲染内容为绘制一个硬件缓存，即`HardwareLayer`。而第二次调用则是为了渲染`HardwareLayer`的`DisplayList`，其渲染内容是`View.draw(Canvas)`方法。

总结硬件加速下的绘图缓存，无论硬件缓存还是软件缓存都可以得到比图6-19更复杂的图6-20，其中斜纹方框为绘图缓存，而灰色方框则是`DisplayList`。

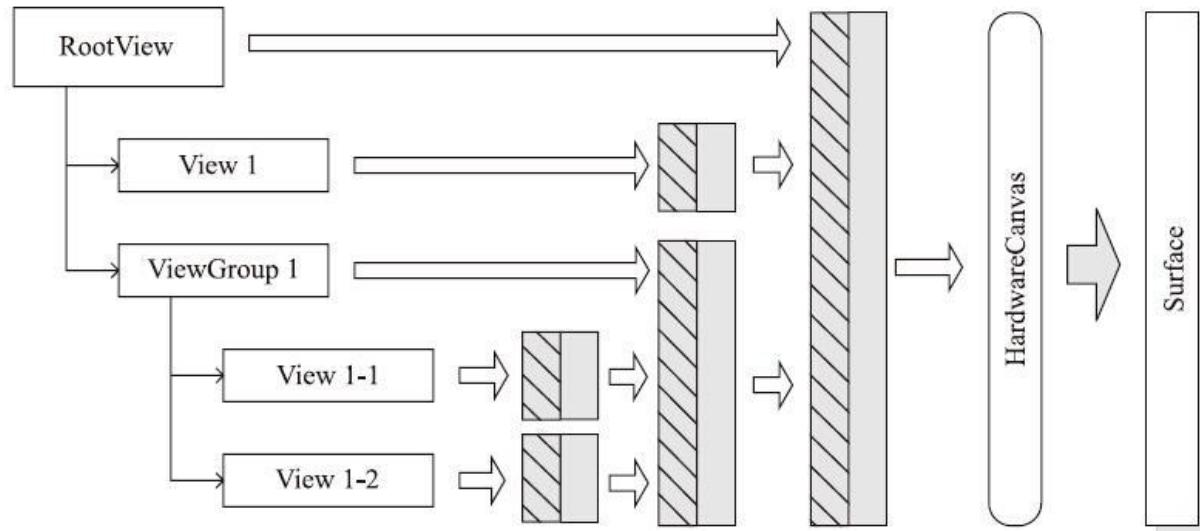


图6-20 硬件加速绘制启用绘图缓存后的特点

3.绘图缓存的利与弊

绘图缓存的目的在于减轻控件的绘制负担，但是对比图6-17、图6-19以及图6-20，可以很明显地发现绘图缓存的引入增加了将绘图缓存绘制到实际目标（Surface或DisplayList）的操作，这个相对于正常绘制的额外动作，反而增加了绘制过程的开销。因此倘若不警慎地使用绘图缓存反而会使绘制性能下降。因此有必要讨论一下究竟应该如何能够在开发过程中充分利用绘图缓存的优势，并将其带来的额外开销降到最低。

参考图6-21所标示的控件树。

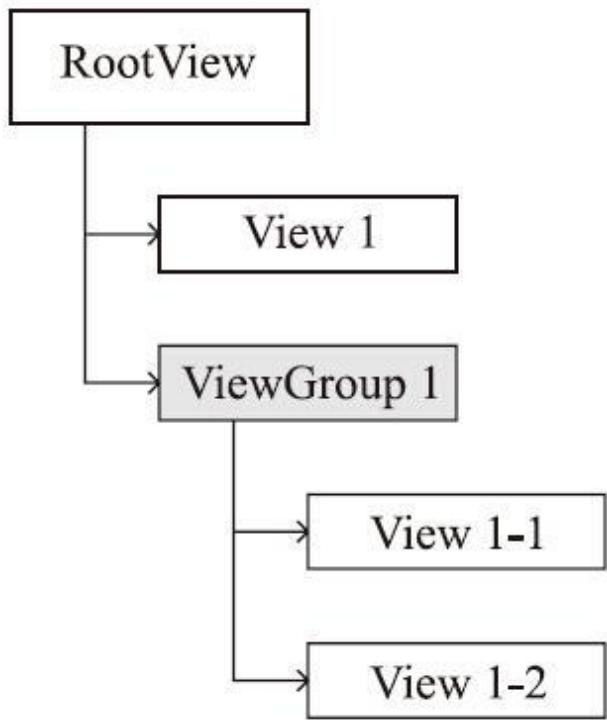


图6-21 一个控件树

假设View1执行了`invalidate ()` 操作而导致控件树重绘。在没有启用绘图缓存的情况下，则会使得并没有发生内容变化的ViewGroup1、View1-1以及View1-2发生重绘。而倘若ViewGroup1启用了绘图缓存，则会通过将ViewGroup1的缓存绘制到Canvas从而省略三个控件的重绘过程。此时绘图缓存会带来性能的提升，前提是ViewGroup1及其子控件的绘制工作的负荷大于绘制一个Bitmap或HardwareLayer。反之，如果View1-1或View1-2的构成十分简单，绘制动作开销也不大（例如它们是个`TextView`或者一个简单色块），那么绘制缓存的开销反而比重绘要大，使用绘图缓存就得不偿失了。

再考虑另外一种情况，图6-21所述的ViewGroup1是一个ListView。当发生拖动操作时，倘若没有为View1-1或View1-2启用绘图缓存，会使得每一次改变它们的拖动位置时产生重绘操作。如果它们是包含了很多控件的控件树，其拖动效率会很低。在为View1-1以及View1-2启用绘图缓存后，会省略它们的重绘而代之以绘制其缓存，从而使拖动效率得到提升。

仍然是图6-21所述的控件树，在ViewGroup1启用绘图缓存的情况下，View1-1执行了invalidate () 操作而导致控件树重绘，这会使得ViewGroup1的绘图缓存无效而进行绘图缓存的更新，因此ViewGroup1、View1-1以及View1-2产生重绘操作，使得绘图缓存得到更新以反映View1-1的最新内容。进一步，更新后的绘图缓存被绘制到RootView所给予的Canvas上。相对于没有启用绘图缓存的情况，绘制绘图缓存的动作变成了额外的负担，因此绘图缓存的存在降低了绘图的效率。

从上述几个情况的讨论可以总结出使用绘图缓存的原则：

- 首要原则是不要为十分轻量级的控件启用绘图缓存。因为缓存绘制的开销可能大于此控件的重绘开销。
- 为很少发生内容改变的控件启用绘图缓存。因为启用了绘图缓存的控件在invalidate () 时会产生额外的缓存绘制操作。

·当父控件要频繁改变子控件的位置或变换时对其子控件启用绘图缓存。这会避免频繁地重绘子控件。

针对第三个原则，ViewGroup提供了setAlwaysDrawnWithCacheEnabled（）以及setChildrenDrawnWithCacheEnabled（）两个方法，用于一次性为所有子控件启用绘制缓存。当通过setAlwaysDrawnWithCacheEnabled（）将FLAG_ALWAYS_DRAWN_WITH_CACHE标记置入mGroupFlags后，将总是使用缓存的方式绘制子控件。而setChildrenDrawnWithCacheEnabled（）则允许根据需求启用或禁用以缓存的方式绘制子控件。正如在View.draw（ViewGroup，Canvas，long）方法中所看到的，当FLAG_ALWAYS_DRAWN_WITH_CACHE或FLAG_CHILDREN_DRAWN_WITH_CACHE标记存在时，局部变量cache被置为true，无论View的layerType被设置为何种值。AbsListView通过这个方法以提高其拖动时的性能。有兴趣的读者可以参考其实现。

6.4.7 控件动画

控件系统中存在三种方式实现控件的动画。其一是使用ValueAnimator类或其子类ObjectAnimator，或者ViewPropertyAnimator类周期性地改变控件的某个属性从而达到动画的效果。其二是使用LayoutTransition类，用于在ViewGroup中删除或者添加一个控件时对其应用一个进入或

移出的动画，LayoutTransition类使用了ObjectAnimator类实现其动画，当子控件被添加或删除时，ViewGroup会根据LayoutTransition中的设置启动对应的动画。其三是使用View.startAnimation () 方法启动一个动画。

使用ValueAnimator、ObjectAnimator进行控件动画的原理与绘制的内部过程并没有十分紧密的联系。在ValueAnimator内部有一个实现了Runnable接口的、线程唯一的AnimationHandler类。当动画运行时，ValueAnimator会将这个AnimationHandler不断地抛给Choreographer，并在VSYNC事件到来时修改指定的控件属性，控件属性的变化引发invalidate () 操作进而进行重绘，以此实现动画效果。虽然ViewPropertyAnimator与Value-Animator没有继承关系，但它其实是ValueAnimator的一个封装，通过其实现动画效果。

而View.startAnimation () 的动画与控件绘制的内部过程联系十分紧密。因为在控件的坐标系变换中，动画矩阵是变换因素之一。相信读者对于控件的坐标系变换已经十分熟悉，本节将在此基础之上讨论其View.startAnimation () 动画的实现原理。

1.启动动画

参考View.startAnimation () 的实现：

```
[View.java-->View.startAnimation()] public void startAnimation(Animation  
animation) {.....// 保存Animation对象setAnimation(animation);.....//  
invalidate()操作，以此触发一次重绘invalidate(true); } public void  
setAnimation(Animation animation) {mCurrentAnimation = animation;.....  
}
```

可以看出，startAnimation（）方法启动的动画依托于Animation类的子类。启动动画时首先将给定的Animation通过setAnimation（）保存到mCurrentAnimation成员中，再通过invalidate（）方法触发一次重绘。

接下来就看重绘时如何使用这个动画了。

2.计算动画变换

既然动画是以坐标系变换的方式产生效果的，因此进行动画计算的代码位于View.draw（ViewGroup，Canvas，long）方法中。

```
[View.java-->View.draw(ViewGroup,Canvas,long)] boolean draw(Canvas  
canvas, ViewGroup parent, long drawingTime) {.....// ① 获取  
startAnimation()所给予的Animation对象final Animation a =  
getAnimation();if (a != null) { /* ② 通过drawAnimation()方法计算当前时  
间点的变换(Transformation)。结果保存在 parent.mChildTransformation  
中 */ more = drawAnimation(parent, drawingTime, a, scalingRequired); //  
transformToApply = parent.mChildTransformation;} else { ..... } // 随后的
```

过程在介绍绘制原理时已经说明过了，即把transformToApply应用到坐标系变换中..... }

接下来参考drawAnimation () 方法的实现：

```
[View.java-->View.drawAnimation()]
private boolean drawAnimation(ViewGroup parent, long drawingTime, Animation a, boolean scalingRequired) {.....final boolean initialized = a.isInitialized();/*
① 首先检查Animation是否初始化。当尚未初始化时，表明这是动画的第一帧。此时View.onAnimati- onStart()回调将被调用 */if (!initialized) {
/* 介绍WMS窗口动画时曾经介绍过Animation.initialize()的作用。当动画使用了相对参数时，此处传入的矩形区域将作为参考 */
a.initialize(mRight - mLeft, mBottom - mTop, parent.getWidth(), parent.getHeight()); ..... /* 注意，onAnimationStart()的默认实现会将PFLAG_ANIMATION_STARTED标记加入View.mPri- vateFlags中，用以标记控件正在运行动画。因此重写onAnimationStart()时一定要调用基类的onAnimationStart()方法以保证控件系统的正常工作 */
onAnimationStart();}/*
② 计算当前时刻的变换。变换被保存在父控件的mChildTransformation成员中
boolean more = a.getTransformation(drawingTime, parent.mChildTransformation, 1f);/*
倘若动画继续进行，则需要再次进行invalidate()以便进行下一帧的计算与绘制。为了提高效率需要 指定invalidate()的边界以便在下次绘制时仅更
```

新此控件的区域。正常来说，只要调用parent.invalidate (mLeft, mTop, mRight, mBottom)即可。但是因为动画可能会改变控件的绘制位置（如 ScaleAnimation, TranslateAnimation等），因此mLeft、mTop、mRight 以及mBottom已经不能用来表示 控件在父控件中的最终位置了。因此对上述4个参数进行同样的动画变换以正确表示最终位置。为此drawAnimation()使用一个局部变量invalidationTransformation 用以表示进行 invalidate()时需要进行的变换 */if (scalingRequired && mAttachInfo.mApplicationScale != 1f) { /* 在兼容模式下，控件都被进行了缩放，这个缩放同样会影响invalidate()的区域。这一缩放所产生的变换保存在ViewGroup.mInvalidationTransformation中 */ if (parent.mInvalidationTransformation == null) { parent.mInvalidationTransformation = new Transformation(); } invalidationTransform = parent.mInvalidationTransformation; // 将动画的变换附加到兼容模式的变换之后 a.getTransformation(drawingTime, invalidationTransform, 1f); } else { // 非兼容模式下，invalidate参数所需的变换与动画变换相同 invalidationTransform = parent.mChildTransformation; } // ③ 如果动画还将继续，则通过调用父控件的invalidate()触发下一次的重绘 if (more) { if (!a.willChangeBounds()) { // 如果动画不改变控件的边界（如AlphaAnimation），则按照控件正常的区域进行invalidate } else { // 使用invalidateTransformation对 invalidate()的4个参数进行变换 final RectF region =

```
parent.mInvalidateRegion; a.getInvalidateRegion(0, 0, mRight - mLeft,
mBottom - mTop, region, invalidationTransform); ..... // 对4个参数进行修
正，并进行invalidate() final int left = mLeft + (int) region.left; final int top =
mTop + (int) region.top; parent.invalidate(left, top, left + (int)
(region.width() + .5f), top + (int) (region.height() + .5f)); } }return more; }
```

在学习WMS窗口动画的原理后，不难理解这个过程。drawAnimation () 通过Animation.getTransformation () 计算当前时间点的变换，并将其保存在父控件的mChildTransformation成员中，然后在View.draw (ViewGroup, Canvas, long) 方法中将这个变换以坐标系变换的方式应用到Cavnas或者DisplayList中，从而对最终的绘制结果产生影响。倘若动画还将继续，则调用invalidate () 以便在下次VSYNC事件到来时进行下一帧的计算与绘制。

本来，这个代码可以很简单，即Animation.getTransformation () +view.invalidate () 即可，但是正如在控件系统其他方面的工作中所见到的，效率永远是控件系统需要考虑的第一位因素，因此 drawAnimation () 才不惜篇幅地计算invalidate区域。

3.动画的结束

倘若drawAnimation () 的返回值more为false，则表示动画已经结束。那么与动画开始相对的，应该会调用View.onAnimationEnd () 。看一

下View.draw (ViewGroup, Canvas, long) 对这一情况的处理：

```
[View.java-->View.draw(ViewGroup,Canvas,long)] boolean draw(Canvas  
canvas, ViewGroup parent, long drawingTime) {..... // 到这里控件已经完  
成绘制过程if (a != null && !more) { ..... /* 很意外，这里并没有直接调  
用View.onAnimationEnd(), 而使用父控件的 finishAnimatingView() */  
parent.finishAnimatingView(this, a);} .....return more; }
```

为什么动画结束的操作要交由父控件完成呢？先看一下

finishAnimatingView () 的实现：

```
[ViewGroup.java-->ViewGroup.finishAnimatingView()] void  
finishAnimatingView(final View view, Animation animation) {final  
ArrayList disappearingChildren = mDisappearingChildren; /* ①  
mDisappearingChildren的存在就是当动画结束时的操作必须交由父控件  
完成的原因。原来，如果在进行动画将这个控件从父控件中移除时，  
ViewGroup会将其从mChildren中移除，但会同时将其放置到  
mDisappearingChildren数组中，并等待动画结束。由于  
mDisappearingChildren中的控件依然会得到绘制（参考  
ViewGroup.dispatchDraw()）。因此在执行了ViewGroup.removeView()  
之后，用户仍然可以看到动画中的控件，直到动画结束后控件才会消  
失。另外，前面提到的LayoutTransition也依赖于这一机制，使得其移  
出动画而被用户看到 */ if (disappearingChildren != null) { if
```

```
(disappearingChildren.contains(view)) { // 把控件从mDisappearingChildren  
中删除，这样控件真正地被移除了  
disappearingChildren.remove(view); /* 动画执行过程中将控件从父控件  
中移除并不会立刻触发onDetachedFromWindow(), 而是当动画完成之后  
才会调用。这是因为控件的绘制依赖于mAttachInfo */ if  
(view.mAttachInfo != null) { view.dispatchDetachedFromWindow(); } //  
clearAnimation()终止动画并将mCurrentAnimation置为null  
view.clearAnimation(); mGroupFlags |=  
FLAG_INVALIDATE_REQUIRED; } }/* ② clearAnimation()将会终止动  
画并将mCurrentAnimation设置为null。于是下次重绘时没有  
mCurrentAnimation，便不会产生动画变换，因而控件恢复到了动画执  
行前的状态。注意仅当Animation.setFillAfter()为false时才会这么做。  
因为FillAfter表示当动画结束时将 会使控件停留在最后一帧的状态，因  
此必须保留mCurrentAnimation使得动画变换持续生效 */if (animation !=  
null && !animation.setFillAfter()) { view.clearAnimation(); } // ③ 最后调用  
View.onAnimationEnd(), 动画终止if ((view.mPrivateFlags &  
PFLAG_ANIMATION_STARTED) == PFLAG_ANIMATION_STARTED)  
{ view.onAnimationEnd(); ..... } }
```

控件动画的工作过程就是如此，开始于View.startAnimation ()，计算
于View.drawAni-mation ()，绘制于View.draw (ViewGroup, Cavavs,
long)，终结于ViewGroup.finishAnimatingView ()。其核心工作是

Animation.getTransformation () 以及控件的坐标系变换，这与WMS的窗口动画的实现十分一致。

6.4.8 绘制控件树的总结

关于控件树绘制的介绍至此结束。这一节讨论了软件/硬件加速绘制、绘图缓存的原理以及控件动画的原理4个方面的内容。控件树的绘制是一个先根遍历的过程，控件首先绘制其自身的内容，然后再将绘制操作传递给它的每一个子控件，因此控件树的绘制工作可以分为如下两个基本过程：

- 控件自身的绘制：由draw (Canvas) 完成，包括背景绘制、onDraw () 以及绘制边界效果。
- 将绘制传递给子控件：从dispatchDraw () 开始到draw (ViewGroup, Canvas, long) 为止，包括坐标系变换、缓存处理、动画计算等。

另外，由于绘制是一个开销较大的操作，因此在相关的代码中对效率的优化随处可见。Android曾经饱受诟病的画面流畅度问题得以改善，不仅仅得益于硬件加速绘制、三重缓冲以及VSYNC的引入，软件代码中精益求精的优化一样功不可没。读者可以在对控件树绘制的学习中仔细体会Android在效率优化上的思想与苦心。

6.5 深入理解输入事件的派发

本节讨论控件系统中另一个重要的话题——输入事件派发。本书第5章讨论了输入事件从设备节点开始一路经过InputReader、InputDispatcher再到InputEventReceiver为止的加工与派发的过程，不过当时所讨论的派发是以窗口为目标，并没有介绍输入事件如何从窗口传递到指定的控件。本节将继续第5章的内容，以InputEventReceiver为起点介绍输入事件在控件树中的派发与处理。

另外，在View类中有很多手段可以用于输入事件的处理，如dispatchKeyEvent ()、onTouch () 以及OnKeyListener等。Activity、Dialog等也有类似的方式可以处理输入事件。它们的优先级、特点以及区别等往往使开发者感到困扰。经过本节的学习读者可以对它们拥有清晰的认识。

控件树中的输入事件派发是由ViewRootImpl为起点，沿着控件树一层一层传递给目标控件，最终再回到ViewRootImpl的一个环形过程。这一过程发生在创建ViewRootImpl的主线程之上，但是却独立于ViewRootImpl.performTraversals () 之外，就是说输入事件的派发并不依赖于ViewRootImpl的“心跳”作为动力，而是有它自己的动力源泉。经过第5章的学习可以知道，这一动力源泉来自用于构建InputEventReceiver的Looper，当一个输入事件被派发给ViewRootImpl所

在的窗口时，Looper会被唤醒并触发InputEventReciever.onInputEvent()回调，控件树的输入事件派发便起始于这一回调。

在正式讨论派发过程之前，首先需要讨论对派发过程有着决定性影响的两个概念——触摸模式以及焦点。

6.5.1 触摸模式

在早期的键盘操作方式的移动设备中，用户选择某项操作的方式以方向键+确定键为主。用户可以通过点击方向键在若干个操作项中进行选择（如菜单和经典的九宫格），并且被选中的操作项会以背景高亮等方式突出显示，然后用户通过点击确定键执行被选中的操作。而现今触摸方式已经成为操作移动设备的主要方式。在以触摸方式操作的界面中，并不存在被选中的操作项的概念，而是通过直接点击期望的项目完成对应的操作。

虽然在移动设备中使用按键操作似乎有些过时，不过在以办公为目的的设备中键盘的存在仍会带来触摸方式不能提供的高效性与便利性。为此，Android同时支持按键与触摸两种操作方式，并且可以非常自然地在二者之间进行切换。早期的Android设备都提供了方向键与确认键，而现有的Android设备虽然都已不再提供这些按键，但是依然可以通过外接键盘的方式（USB或者蓝牙）实现键盘操作。



小知识

事实上，Android项目最初是为了开发一个用在键盘手机之上的智能手机系统，不过在iPhone发布之后，Android迅速改变了设计意图使之可以支持触屏操作。所以Android可以同时支持键盘与触摸两种操作方式更是理所当然了。

为了同时支持这两种模式，必须清楚这两种操作模式的区别与共同点。可以从焦点的角度讨论这一问题。以一个拥有若干项的菜单为例，在键盘操作方式中，当用户通过方向键选中一个菜单项时，这一菜单项便会获得焦点（高亮显示），当点击确认键时这一按键的事件被派发给拥有焦点的菜单项，进而执行相应的动作。在这种模式下，必定有一个菜单项处于焦点状态，以便用户知道按下确认键后会发生什么事情。而在触摸方式下，菜单项不会获取焦点，而是直接响应触摸事件执行相应的动作。这种模式下不需要任何一个菜单项处于焦点状态，因为用户会通过点击选择自己希望的操作，相反，倘若有一个菜单项处于高亮状态反而会使用户产生迷惑而使得悬空的手指点不下去。二者也有共同点，例如一个文本框，无论在哪种操作方式下，它

都可以获得焦点以接受用户的输入。也就是说可以获取焦点的控件分为两类：

- 在任何情况下都可以获取焦点的控件，如文本框。
- 仅在键盘操作时可以获取焦点的控件，如菜单项、按钮等。

触摸模式（TouchMode）正是为管理二者的差异而引入的概念，Android通过进入或退出触摸模式实现在二者之间的无缝切换。在非触摸模式下，文本框、按钮、菜单项等都可以获取焦点，并且可以通过方向键使得焦点在这些控件之间游走。而在进入触摸模式后，某些控件如菜单项、按钮将不再可以保持或获取焦点，而文本框则仍然可以保持或获取焦点。

触摸模式是一个系统级的概念，就是说会对所有窗口产生影响。系统是否处于触摸模式取决于WMS中的一个成员变量mInTouchMode，而确定是否进入或者退出触摸模式则取决于用户对某一个窗口所执行的操作。

导致退出触摸模式的操作有：

- 用户按下了方向键。
- 用户通过键盘按下一个字母键（A、B、C、D等按键）。

·开发者执行了View.requestFocusFromTouch () 。

而进入触摸模式的操作只有一个，就是用户在窗口上进行了点击操作。

窗口的ViewRootImpl会识别上述操作，然后通过WMS的接口setInTouchMode () 设置WMS.mInTouchMode使得系统进入或退出触摸模式。而当其他窗口进行relayout操作时会在WMS.relayoutWindow () 的返回值中添加或删除REAYOUT_RES_IN_TOUCH_MODE标记使得它们得知系统目前的操作模式。



注意

只有拥有ViewRootImpl的窗口才能影响触摸模式，或对触摸模式产生响应。通过WMS的接口直接创建的窗口必须手动地维护触摸模式。

当系统进入或退出触摸模式时会对控件系统产生怎样的影响呢？根据上述内容的介绍可以得知它影响的是焦点的选择策略。通过接下来所

讨论的控件树焦点相关的内容，可以使得读者对触摸模式拥有更深入的理解。

6.5.2 控件焦点

和第5章所讨论的窗口焦点类似，控件的焦点影响了按键事件的派发。另外，控件的焦点还影响了控件的表现形式，拥有焦点的控件往往会上高亮显示以区别其他控件。

1. 获取焦点的条件

控件获取焦点的方式有很多种，例如从控件树中按照一定策略查找到某个控件并使其获得焦点，或者用户通过方向键选择某个控件使其获得焦点等。而最基本的方式是通过View.requestFocus () 。本节将通过介绍View.requestFocus () 的实现原理揭示控件系统管理焦点的方式。

View.requestFocus () 的实现有两种，即View和ViewGroup的实现是不同的。当实例是一个View时，表示期望此View能够获取焦点。而当实例是一个ViewGroup时，则会根据一定的焦点选择策略选择其一个子控件或ViewGroup本身作为焦点。本小节将首先讨论实例是一个View时的情况以揭示控件系统管理焦点的方式，随后再讨论ViewGroup下requestFocus () 方法的实现。

参考requestFocus () 代码：

[View.java-->View.requestFocus()] public final boolean requestFocus() {/*
调用requestFoscus()的一个重载。View.FOCUS_DOWN表示焦点的寻找
方向。当本控件是一个View- Group时将会从左上角开始沿着这个方向
查找可以获取焦点的子空间。不过在本例只讨论控件是一个View 时的
情况，此时该参数并无任何效果 */return
requestFocus(View.FOCUS_DOWN); }public final boolean requestFocus(int
direction) {/* 继续调用另外一个重载，新的重载中增加了一个Rect作为
参数。此Rect表示了上一个焦点控件的区域。 它表示从哪个位置开始
沿着direction所指定的方向查找焦点控件。仅当本控件是ViewGroup时
此参数 才有意义 */return requestFocus(direction, null); }public boolean
requestFocus(int direction, Rect previouslyFocusedRect) {/* requestFocus()
的这一重载便是View和Viewgroup分道扬镳的地方。
requestFocusNoSearch()方法 的意义就是无须查找，直接使本控件获取
焦点 */return requestFocusNoSearch(direction,
previouslyFocusedRect); }private boolean requestFocusNoSearch(int
direction, Rect previouslyFocusedRect) { // 首先检查一下此控件是否符合
拥有焦点的条件// ① 首先，此控件必须是Focusable的。可以通过
View.setFocusable()方法设置控件是否focusable if ((mViewFlags &
FOCUSABLE_MASK) != FOCUSABLE || (mViewFlags &
VISIBILITY_MASK) != VISIBLE) { return false; } // ② 再者，如果系统目
前处于触摸模式，则要求此控件必须可以在触摸模式下拥有焦点if

```
(isInTouchMode() && (FOCUSABLE_IN_TOUCH_MODE !=  
(mViewFlags & FOCUSABLE_IN_TOUCH_MODE))) { return false;}// ③  
最后，如果任一父控件的DescendantFocusability取值为  
FOCUS_BLOCK_DESCENDANTS时，阻止此控件获取焦点。  
hasAncestorThatBlocksDescendantFocus()会沿着控件树一路回溯到整个  
控件树的 根控件并逐一检查DescendantFocusability特性的取值 */if  
(hasAncestorThatBlocksDescendantFocus()) { return false;}// ④ 最后调用  
handleFocusGainInternal()使此控件获得焦点  
handleFocusGainInternal(direction, previouslyFocusedRect);return true; }
```

并不是所有控件都可以获取焦点的。控件系统通过View.setFocusable()
() 设置一个控件能否获取焦点。View.setFocusable () 会将
FOCUSABLE或者NOT_FOCUSABLE标记加入View.mViewFlags成员
中。

当FOCUSABLE标记存在时也不一定能够获取焦点，例如虽然6.5.1节中
所介绍的菜单项拥有FOCUSABLE标记，但是在触摸模式下它仍无法获
取焦点。控件系统通过View.setFocusableInTouchMode () 以区分这类
控件。View.setFocusableInTouchMode () 会将
FOCUSABLE_IN_TOUCH_MODE标记加入或移出View.mViewFlags。
于是，当系统处于触摸模式时，仅当拥有
FOCUSABLE_IN_TOUCH_MODE标记的控件才能获取焦点。

最后，控件能否获取焦点还取决于它的父控件的一个特性 DescendantFocusability，这一特性描述了子控件与父控件之间的焦点获取策略。DescendantFocusability 可以有三种取值，其中一种为 FOCUS_BLOCK_DESCENDANTS。当父控件的这一特性取值为 FOCUS_BLOCK_DESCENDANTS 时，父控件将会阻止其子控件或子控件的子控件获取焦点。在介绍 ViewGroup.requestFocus() 时将会有对这一特性进行详细介绍。

因此控件能否获取焦点的策略如下：

- 当 NOT_FOCUSABLE 标记位于 View.mViewFlags 时，无法获取焦点。
- 当控件的父控件的 DescendantFocusability 取值为 FOCUS_BLOCK_DESCENDANTS 时，无法获取焦点。
- 当 FOCUSABLE 标记位于 View.mViewFlags 时分为两种情况：
 - a) 位于非触摸模式时，控件可以获取焦点。
 - b) 位于触摸模式时，View.mViewFlags 中存在 FOCUSABLE_IN_TOUCH_MODE 标记时可以获取焦点，否则不能获取焦点。

接下来分析 View.requestFocusInternal()。

2. 获取焦点

```
[View.java-->View.handleFocusGainInternal()] void  
handleFocusGainInternal(int direction, Rect previouslyFocusedRect) {if  
((mPrivateFlags & PFLAG_FOCUSED) == 0) { // ① 把  
PFLAG_FOCUSED标记加入mPrivateFlags中。这便表示此控件已经拥  
有焦点了 mPrivateFlags |= PFLAG_FOCUSED; /* ② 将这一变化通知其  
父控件。这一操作的主要目的是保证控件树中只有一个控件拥有焦  
点，并且 在ViewRootImpl中触发一次“遍历”以便对控件树进行重绘 */  
if (mParent != null) { mParent.requestChildFocus(this, this); } /* ③ 通知对  
此控件焦点变化感兴趣的监听者。在这个方法中，  
View.onFocusLose()、OnFocusChangeListener.onFocusChange()都会被  
调用。另外，控件焦点决定了输入法的输入对象，因此  
InputMethodManager的focusIn()和focusOut()也会在这里被调用以更新输  
入法的状态 */ onFocusChanged(true, direction, previouslyFocusedRect); //  
④ 更新控件的Drawable状态。这将使得控件在随后的绘制中得以高亮  
显示 refreshDrawableState(); .....} }
```

View.handleFocusGainInternal () 方法的4个工作都在情理之中，首先将PFLAG_FOCUSED标记加入mPrivateFlags成员中以宣称此控件是焦点的所有者。然后通过mParent.requestChildFocus () 将这一变化通知父控件以将焦点从上一个焦点控件中夺走，并触发一次重绘。接着通

过View.onFocusChanged () 方法将焦点变化通知给感兴趣的监听者。

最后更新控件的DrawableState以使控件的绘制内容反映焦点的变化。

接下来讨论mParent.requestChildFocus () 的实现。PFLAG_FOCUSED 是一个控件是否拥有焦点的最直接体现，然而这并不是焦点管理的全部。这一标记仅仅体现了焦点在个体级别上的特性，而 mParent.requestChildFocus () 则体现了焦点在控件树的级别上的特性。

3. 控件树中的焦点体系

mParent.requestChildFocus () 是一个定义在ViewParent接口中的方法，其实现者为ViewGroup及ViewRootImpl。ViewGroup实现的目的之一是用于将焦点从上一个焦点控件手中夺走，即将PFLAG_FOCUSED标记从控件的mPrivateFlags中移除。而另一个目的则是将这一操作继续向控件树的根部进行回溯，直到ViewRootImpl，ViewRootImpl的 requestChildFocus () 会将焦点控件保存起来备用，并引发一次“遍历”。参考ViewGroup.requestChildFocus () 方法的实现：

```
[ViewGroup.java-->View.requestChildFocus()] public void  
requestChildFocus(View child, View focused) {/* ① 如果上一个焦点控件  
就是这个ViewGroup，则通过调用View.unFocus()将PLFAG_FOCUSED  
标记移除，以释放焦点 */super.unFocus();if (mFocused != child) { /* ②
```

如果上一个焦点控件在这个ViewGroup所表示的控件树之中，即 mFocused不为null，则调用 mFocused.unFocus()以释放焦点*/ if (mFocused != null) { mFocused.unFocus(); } // ③ 设置mFocused成员为 child。注意child参数并不是实际拥有焦点的控件。而是此ViewGroup的直接子控件，同时它是实际拥有焦点的控件的父控件 */ mFocused = child;}if (mParent != null) { /* ④ 将这一操作继续向控件树的根部回溯。注意child参数是此ViewGroup，而不是实际拥有焦点的 focused */ mParent.requestChildFocus(this, focused); }

此方法中mFocused是一个View类型的变量，它是控件树焦点管理的核心所在。围绕着mFocused， ViewGroup.requestChildFocus () 方法包含了新的焦点体系的建立过程，以及旧有焦点体系的销毁过程。

新的焦点体系的建立过程是通过在ViewGroup.requestChildFocus () 方法的回溯过程中进行mFocused=child这一赋值操作完成的。当回溯完成后， mFocused=child将会建立起一个单向链表，使得从根控件开始通过mFocused成员可以沿着这一单向链表找到实际拥有焦点的控件，即实际拥有焦点的控件位于这个单向链表的尾端，如图6-22所示。

而旧有的焦点体系的销毁过程则是通过在回溯过程中调用 mFocused.unFocus () 完成的。unFocus () 方法有ViewGroup和View两种实现。首先看一下ViewGroup.unFocus () 的实现：

```
[ViewGroup.java-->ViewGroup.unFocus()] void unFocus() {if (mFocused == null) { /* 如果mFocused为空，则表示此ViewGroup位于mFocused单向链表的尾端，即此ViewGroup是焦点的实际拥有者，因此调用View.unFocus()使此ViewGroup放弃焦点 */ super.unFocus();} else { // 否则将unFocus()传递给链表的下一个控件 mFocused.unFocus(); // 最后将mFocused设置为null mFocused = null;} }
```

可见ViewGroup.unFocus () 将unFocus () 调用沿着mFocused所描述的链表沿着控件树向下遍历，直到焦点的实际拥有者。焦点的实际拥有者会调用View.unFocus () ，它会将PFLAG_FOCUSED移除，当然也少不了更新DrawableState以及onFocusChanged () 方法的调用。

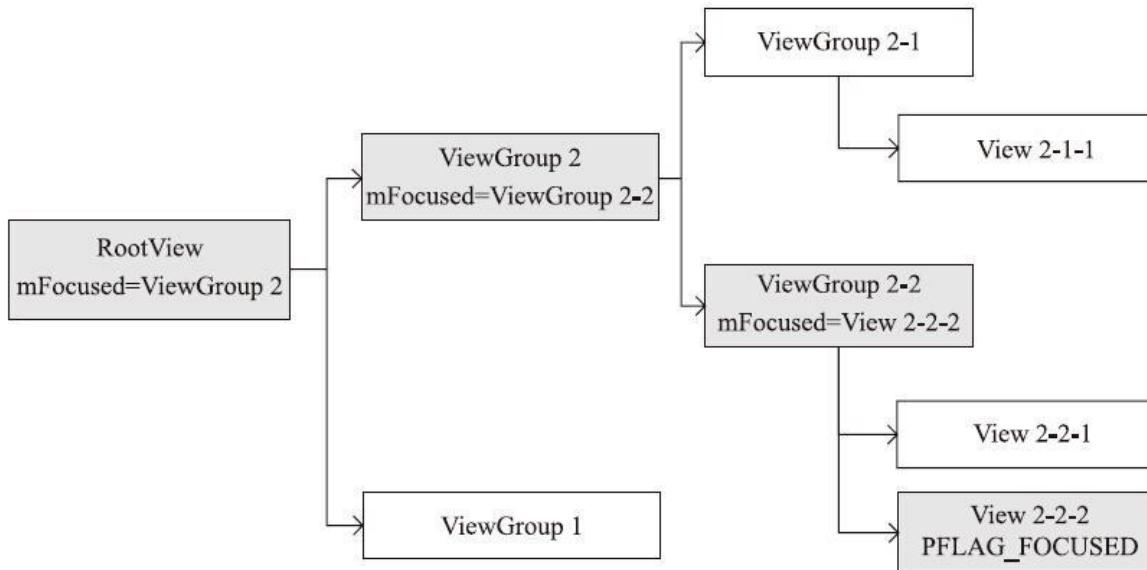


图6-22 `mFocused`所建立的焦点体系

以图6-22的控件树的焦点状态为例来描述旧有焦点体系的销毁以及新焦点体系的建立过程。当View2-1-1通过View.requestFocus () 尝试获取焦点时，首先会将PFLAG_FOCUSED标记加入其mPrivateFlags成员中以声明其拥有焦点。然后调用ViewGroup2-1的requestChildFocus () ，此时ViewGroup2-1会尝试通过unFocus () 销毁旧有的焦点体系，但是由于其mFocused为null，它无法进行销毁，于是它将其mFocused设置为View2-1-1后将requestChildFocus () 传递给ViewGroup2。此时ViewGroup2的mFocused指向了ViewGroup2-2，于是调用ViewGroup2-2的unFocus () 进行旧有焦点体系的销毁工作。ViewGroup2-2的unFocus () 将此操作传递给View2-2-2的unFocus () 以移除View2-2-2的PFLAG_FOCUSED标记，并将其mFocused置为null。回到ViewGroup2的requestChildFocus () 方法后，ViewGroup2将其mFocused重新指向到ViewGroup2-1。在这些工作完成后，图6-22所描述的焦点体系则变为图6-33所示。

总而言之，控件树的焦点管理分为两个部分：其一是描述个体级别的焦点状态的PFLAG_FOCUSED标记，用于表示一个控件是否拥有焦点；其二是描述控件树级别的焦点状态的ViewGroup.mFocused成员，用于提供一条链接控件树的根控件到实际拥有焦点的子控件的单向链表。这条链表提供了在控件树中快速查找焦点控件的简便办法。另外，由于焦点的排他性，当一个控件通过requestFocus () 获取焦点以创建新的焦点体系时伴随着旧有焦点体系的销毁过程。

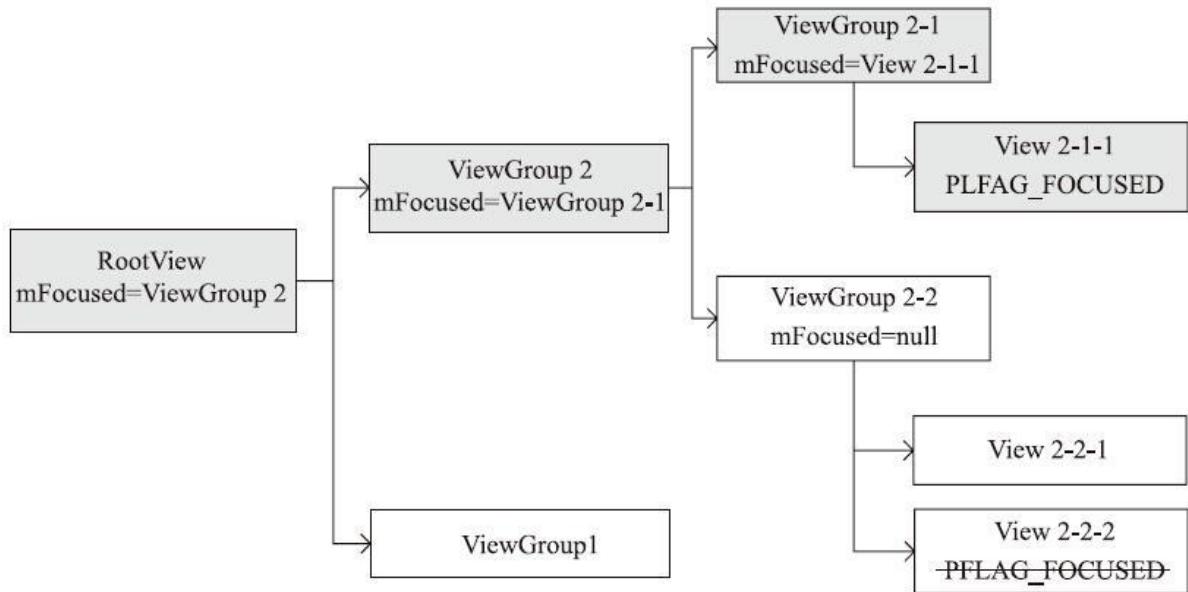


图6-23 View2-1-1获取焦点后的新焦点体系



说明

View类下有两个查询控件焦点状态的方法——isFocused () 以及 hasFocus ()，二者的区别在于：isFocused () 表示的是狭义的焦点状态，即控件是否拥有PFLAG_FOCUSED标记；而hasFocus () 表示的是广义的焦点状态，即拥有PFLAG_FOCUSED标记或mFocused不为空，可以理解为hasFocus () 表示焦点是否在其内部（自身拥有焦点，或者拥有焦点的控件在其所代表的控件树中）。在图6-23中，所有灰色

的控件的hasFocus () 返回值都为true，而仅有View2-1-1的isFocused () 返回值为true。

至此，相信读者已经对焦点的体系有了深刻理解。接下来的内容将会讨论一种稍微复杂的情况，即尝试在ViewGroup上调用requestFocus () 会发生什么。

4. ViewGroup的requestFocus ()

在本节开始时曾经讨论了获取焦点的最基本方式是View.requestFocus () 。如果调用此方法的实例是一个控件（非ViewGroup），其意义非常明确，即希望此控件能够获取焦点。而倘若调用此方法的实例是一个ViewGroup时又当如何呢？本节将讨论这一问题。

ViewGroup重写了View.requestFocus (int direction, Rect previouslyFocusedRect) 以应对这种情况。参考如下代码：

```
[ViewGroup.java-->ViewGroup. requestFocus()] public boolean  
requestFocus(int direction, Rect previouslyFocusedRect) { // ① 首先获取  
ViewGroup的DescendantFocusability特性的取值int  
descendantFocusability = getDescendantFocusability(); // 根据不同的  
DescendantFocusability特性，requestFocus()会产生不同的效果switch  
(descendantFocusability) { case FOCUS_BLOCK_DESCENDANTS: /* ②  
FOCUS_BLOCK_DESCENDANTS：ViewGroup将会阻止所有子控件获
```

取焦点，于是调用 View.requestFocus() 尝试自己获取焦点 */ return
super.requestFocus(direction, previouslyFocusedRect); case
FOCUS_BEFORE_DESCENDANTS: { /* ③
FOCUS_BEFORE_DESCENDANTS : ViewGroup 将有优先于子控件获
取焦点的权利。因此会首先调用 View.requestFocus() 尝试自己获取焦
点，倘若自己不满足获取焦点的条件则通过调用
onRequestFocusInDescendants() 方法将获取焦点的请求转发给子控件 */
final boolean took = super.requestFocus(direction, previouslyFocusedRect);
return took ? took : onRequestFocusInDescendants(direction,
previouslyFocusedRect); } case FOCUS_AFTER_DESCENDANTS: { /* ④
FOCUS_AFTER_DESCENDANTS : 子控件将有优先于此 ViewGroup 获
取焦点的权利。因此会首先调用 onRequestFocusInDescendants() 尝试将
获取焦点的请求转发给子控件。倘若所有子控件都无法获取焦点，再
调用 View.requestFocus() 尝试自己获取焦点 */ final boolean took =
onRequestFocusInDescendants(direction, previouslyFocusedRect); return
took ? took : super.requestFocus(direction, previouslyFocusedRect); }
default: // 抛出异常} }

可见，在 ViewGroup 上调用 requestFocus () 方法会根据其
DescendantsFocusability 特性的不同而产生三种可能的结果。开发者可
以通过 ViewGroup.setDescendantFocusability () 方法修改这一特性。

在FOCUS_BLOCK_DESCENDANTS特性下， ViewGroup将会拒绝所有子控件获取焦点，此时调用ViewGroup.requestFocus () 会产生唯一的结果，即ViewGroup会尝试自己获取焦点。此时的流程与View.requestFocus () 没有什么区别。

而在其他两种特性下调用ViewGroup.requestFocus () 则会产生View.requestFocus () 与ViewGroup.onRequestFocusInDescendants () 两种可能的结果，不同的特性下二者的优先级不同。

onRequestFocusInDescendants () 负责遍历其所有子控件，并将requestFocus () 转发给它们。参考其实现：

```
[ViewGroup.java-->ViewGroup. onRequestFocusInDescendants()] protected  
boolean onRequestFocusInDescendants(int direction, Rect  
previouslyFocusedRect) {/*此方法的目的是按照direction参数所描述的  
方向在子控件列表中依次尝试使其获取焦点这里direction 所描述的方向  
并不是控件在屏幕上的位置，而是它们在mChildren列表中的位置因此  
direction仅有 按照索引递增(FOCUS_FORWARD)或递减两种方向可选  
*/int index, increment, end, count = mChildrenCount;if ((direction &  
FOCUS_FORWARD) != 0) { index = 0; increment = 1; end = count;} else {  
index = count - 1; increment = -1; end = -1;}final View[] children =  
mChildren;for (int i = index; i != end; i += increment) { View child =  
children[i]; //首先子控件必须是可见的 if ((child.mViewFlags &
```

```
VISIBILITY_MASK) == VISIBLE) { // 调用子控件的requestFocus(), 如果子控件获取了焦点, 则停止继续查找 if (child.requestFocus(direction, previouslyFocusedRect)) { return true; } } } return false; }
```

ViewGroup.onRequestFocusInDescendants () 其实是一种最简单的焦点查找的算法。它按照direction所指定的方向，在mChildren列表中依次调用子控件的requestFocus () 方法，直到有一个子控件获取了焦点。另外，需要注意子控件有可能也是一个ViewGroup，此时将会重复本节所讨论的工作，直到找到一个符合获取焦点条件的控件并使其获得焦点为止。

至此，焦点管理中的requestFocus () 已经介绍完成。与其相对的还有一个clearFocus () 方法用于清除控件的焦点。requestFocus () 与 clearFocus () 作为互为反作用的一对双胞胎，它们的执行方式与 requestFocus () 是一致的。只不过它的执行过程是销毁现有的焦点体系而已（移除PFLAG_FOCUSED以及将mFocused设置为null）。需要注意的是，在现有的焦点体系被销毁后，它还会调用 ViewRootImpl.mView.requestFocus () 方法设置一个新的焦点。根据 ViewGroup.requestFocus () 的工作原理，这一行为会在控件树中寻找一个合适的控件并将焦点给它。而如果所选中的控件正好是执行 clearFocus () 的控件，那么它会重新获得焦点。



说明

`ViewGroup.requestFocus ()` 方法还有另外一个重要用处，就像 `View.clearFocus ()` 最后会设置新的焦点一样，当控件树被添加到 `ViewRootImpl` 之后也会调用 `ViewRootImpl.mView.requestFocus ()` 设置初始的焦点。

接下来讨论关于焦点的另外一个重要话题，即下一个焦点控件的查找。

5. 下一个焦点控件的查找

当一个控件获取焦点之后，用户往往通过按下方向键移动焦点到另一个控件上。这时控件系统需要在控件树中指定的方向上寻找距离当前控件最近的一个控件，并将焦点赋予它。与

`ViewGroup.onRequestFocusInDescendants ()` 方法按照控件在 `mChildren` 数组中的顺序查找不同，这一查找依赖于控件在窗口中的位置。这一工作由 `View.focusSearch ()` 方法完成。参考代码如下：

```
[View.java-->View.focusSearch()] public View focusSearch(int direction) {if  
(mParent != null) { // 查找工作会交给父控件完成 return  
mParent.focusSearch(this, direction);} else { /* 如果控件没有父控件就直  
接返回null，毕竟一个控件没有添加到控件树中查找下一个焦点是没有  
意义的 */ return null;} }
```

View.focusSearch () 会调用父控件的focusSearch (View focused, int direction) 方法，由父控件决定下一个焦点控件是谁。参考其在 ViewGroup中的实现：

```
[ViewGroup.java-->focusSearch()] public View focusSearch(View focused,  
int direction) {if (isRootNamespace()) { /* ① 如果isRootNamespace()返回  
true，则表示这是一个根控件。此时ViewGroup拥有整个控件树，因此  
它是负责焦点查找的最合适的人选。它使用了FocusFinder工具类进行  
焦点查找 */ return FocusFinder.getInstance().findNextFocus(this, focused,  
direction);} else if (mParent != null) { // ② 如果这不是根控件，则继续向  
控件树的根部回溯 return mParent.focusSearch(focused, direction);}return  
null; }
```

如果此ViewGroup不是根控件，则继续向控件树的根部回溯，一直回溯到根控件后，便使用FocusFinder的findNextFocus () 方法查找下一个焦点。这个方法的三个参数的意义如下：

·this，即root。findNextFocus（）方法通过这个参数获取整个控件树中所有的候选控件。

·focused，表示当前拥有焦点的控件。findNextFocus（）方法会以这个控件所在的位置开始查找。

·direction表示了查找的方向。

参考findNextFocus（）方法的代码：

```
[FocusFinder.java-->FocusFinder.findNextFocus()]
public final View
findNextFocus(ViewGroup root, View focused, int direction) {return
findNextFocus(root, focused, null, direction);}
private View
findNextFocus(ViewGroup root, View focused, Rect focusedRect, int
direction) {View next = null; // ① 首先将尝试依照开发者的设置选择下一
个拥有焦点的控件
if (focused != null) { next =
findNextUserSpecifiedFocus(root, focused, direction);}
if (next != null) {
return next;}/* ② 内置算法。倘若开发者没有为当前的焦点控件设置下
一个拥有焦点的控件，将会使用控件系统内置的算 法进行下一个焦点
的查找*/
ArrayList focusables = mTempList;
try { focusables.clear(); /* ③
将控件树中所有可以获取焦点的控件存储到focusables列表中。后续的
将会在这个列表中进行查找 */
root.addFocusables(focusables, direction);
if (!focusables.isEmpty()) { // ④ 调用findNextFocus()的另一个重载完成查
```

```
找 next = findNextFocus(root, focused, focusedRect, direction, focusables);
} } finally { focusables.clear(); } return next; }
```

FocusFinder.findNextFocus () 会首先尝试通过 findNextUserSpecifiedFocus () 获取由开发者设置的下一个焦点控件。有时候控件系统内置的焦点查找算法并不能满足开发者的需求，因此开发者可以通过View.setNextFocusXXXId () 方法设置此控件的下一个可获取焦点的控件的Id。其中XXX可以是Left、Right、Top、Bottom和Forward，分别用来设置不同方向下的下一个焦点控件。 findNextUserSpecifiedFocus () 会在focused上调用getNextFocusXXXId () 方法获取对应的控件并返回。

倘若开发者在指定方向上没有设置下一个焦点控件，则 findNextUserSpecifiedfocus () 方法会返回null，findNextFocus () 会使用内置的搜索算法进行查找。这个内置算法会首先将控件树中所有可以获取焦点的控件添加到一个名为focusables的列表中，并以这个列表作为焦点控件的候选集合。这样做的目的不仅仅是提高效率，更重要的是这个列表打破了控件在控件树中的层次关系。它在一定程度上体现了焦点查找的一个原则，即控件在窗口上的位置是唯一查找依据，与控件在控件树中的层次无关。

最后调用的findNextFocus () 的另一个重载将在focusables列表中选出下一个焦点控件。参考以下实现：

[FocusFinder.java-->FocusFinder.findNextFocus()] private View findNextFocus(ViewGroup root, View focused, Rect focusedRect, int direction, ArrayList focusables) {
// ① 首先需要确定查找的起始位置
if (focused != null) { /* 当focused不为null时，起始位置即focused所在的位置。View.getFocusedRect()所返回 的并不是控件的mLeft、 mTop、 mRight、 mBottom。因为Scroll的存在它们并不能反映控件的 真实位置。View.getFocusedRect()会将Scroll所产生的偏移考虑在内，但是 Transformation(如 setScaledX()等设置)并没有计算在内，因此它们并不会影响焦点查找的结果 */ focused.getFocusedRect(focusedRect); /*
View.getFocusedRect()所返回的结果基于View本身的坐标系。为了使得控件之间的位置可以 比较，必须将其转换到根控件所在的多坐标系中
/ root.offsetDescendantRectToMyCoords(focused, focusedRect); } else { if (focusedRect == null) { / 当focusedRect为null时，表示查处在指定方向上的第一个可以获取焦点的控件。此时会以根控件的某个角所在位置作为起始位置。例如对Left和Up两个方向来说，起始位置会被 设置为根控件的右下角这个点，而对Right和Bottom来说，起始位置将会是根控件的左上角 */ } } // 接下来便会根据不同的方向选择不同的查找算法
switch (direction) { case View.FOCUS_FORWARD: case View.FOCUS_BACKWARD: /* ② 对FOCUS_FORWARD/BACKWARD 来说将选择相对位置进行查找。这种查找与控件位置无关， 它会选择focusables列表中索引近邻focused的控件作为查找结果 */ return

```
findNextFocusInRelativeDirection(focusables, root, focused, focusedRect,  
direction); case View.FOCUS_UP: case View.FOCUS_DOWN: case  
View.FOCUS_LEFT: case View.FOCUS_RIGHT: /* ③ 对于UP、  
DOWN、 LEFT、 RIGHT 4个方向会根据控件的实际位置进行查找  
return findNextFocusInAbsoluteDirection(focusables, root, focused,  
focusedRect, direction); default: throw new  
IllegalArgumentException("Unknown direction: " + direction); } }
```

在这个方法中首先确定了查找的起点位置，然后根据direction参数的取值选择两种不同的查找策略。为FORWARD和BACKWARD两种查找方向所选择的查找策略比较简单，即首先确定focused所表示的控件在focusables列表中的索引index，然后选择在focusable列表中索引为index+1或index-1的两个控件之一作为查找结果。因此使用这两种方向进行查找的结果与ViewGroup.onRequestFocusInDescendants () 类似，它反映了控件在ViewGroup.mChildren列表中的顺序。而对于其他4种方向的查找则复杂得多。参考findNextFocusInAbsoluteDirection () 的代码：

[FocusFinder.java-->FocusFinder. findNextFocusInAbsoluteDirection()]

```
View findNextFocusInAbsoluteDirection(ArrayList focusables, ViewGroup  
root, View focused, Rect focusedRect, int direction) { // ① 首先确定第一个  
最佳候选控件的位置。focusedRect即查找的起始位置
```

```
mBestCandidateRect.set(focusedRect);.....// closest表示在指定的方向上距离起始位置最接近的一个控件View closest = null;int numFocusables = focusables.size();// 遍历focusable列表进行查找for (int i = 0; i < numFocusables; i++) { View focusable = focusables.get(i); /* 既然是查找下一个焦点控件，那么已经拥有焦点的控件自然不能算作候选者。另外根控件也不能作为候选对象 */ if (focusable == focused || focusable == root) continue; /* ② 与获取起始位置一样，获取候选控件的位置。将其位置转换到根控件的坐标系中，以便能够与起始位置进行比较 */ focusable.getFocusedRect(mOtherRect); root.offsetDescendantRectToMyCoords(focusable, mOtherRect); /* ③ 通过isBetterCandidate()方法比较现有的mBestCandidateRect与候选控件的位置。倘若候选控件的位置更佳，则设置候选控件为closest，设置候选控件的位置为mBestCandidateRect。如此往复，当所有候选控件都经过比较之后，closest便是最后的查找结果 */ if (isBetterCandidate(direction, focusedRect , mOtherRect, mBestCandidateRect)) { mBestCandidateRect.set(mOtherRect); closest = focusable; } } // 返回closest作为下一个焦点控件return closest; }
```

这个方法的实现非常直观。在遍历focusables列表的过程中使用isBetterCandidate () 方法不断地将mBestCandidateRect与候选控件的位置进行比较，并在遍历过程中保存最佳的候选控件到closest变量中。在遍历完成后，closest即下一个焦点。整个过程与插入排序非常相似。

那么isBetterCandidate（）方法又是如何确定两个位置谁更合适呢？由于其算法实现十分繁琐并且难以理解，这里直接给出其比较原则：

·首先，与起始位置比较，倘若一个控件A位于指定方向上，而控件B位于指定方向的另外一侧，则控件A是更佳候选。如图6-24的原则1所示，以LEFT为查找方向时，由于控件B位于Focused控件的右侧，因此控件A为更佳的候选。

·其次，将起始位置沿着查找方向延伸到无限远，形成的形式被称为BEAM一条杠。倘若一个控件A与BEAM存在交集，而另一个控件B没有，则与BEAM存在交集的控件A为更佳候选。如图6-24的原则2所示。

·最后，当无法通过BEAM确定更佳候选时（如两个控件与BEAM同时存在交集，或同时不存在交集），则通过比较两控件与焦点控件相邻边的中点的距离进行确定，距离近者为更佳候选。注意在进行距离计算时FocusFinder为指定方向增加了一个权重，以LEFT方向查找为例，其距离计算公式为 $(13*dx*dx+dy*dy)$ ，就是说这个距离对于X方向的距离更加敏感。以图6-24的原则3为例，相对于控件A，控件B到Focused实际距离是更小的。但由于在进行计算时X方向的距离有了3.6倍的加成，因此其计算距离远大于控件A，由此推断控件A是更佳候选。

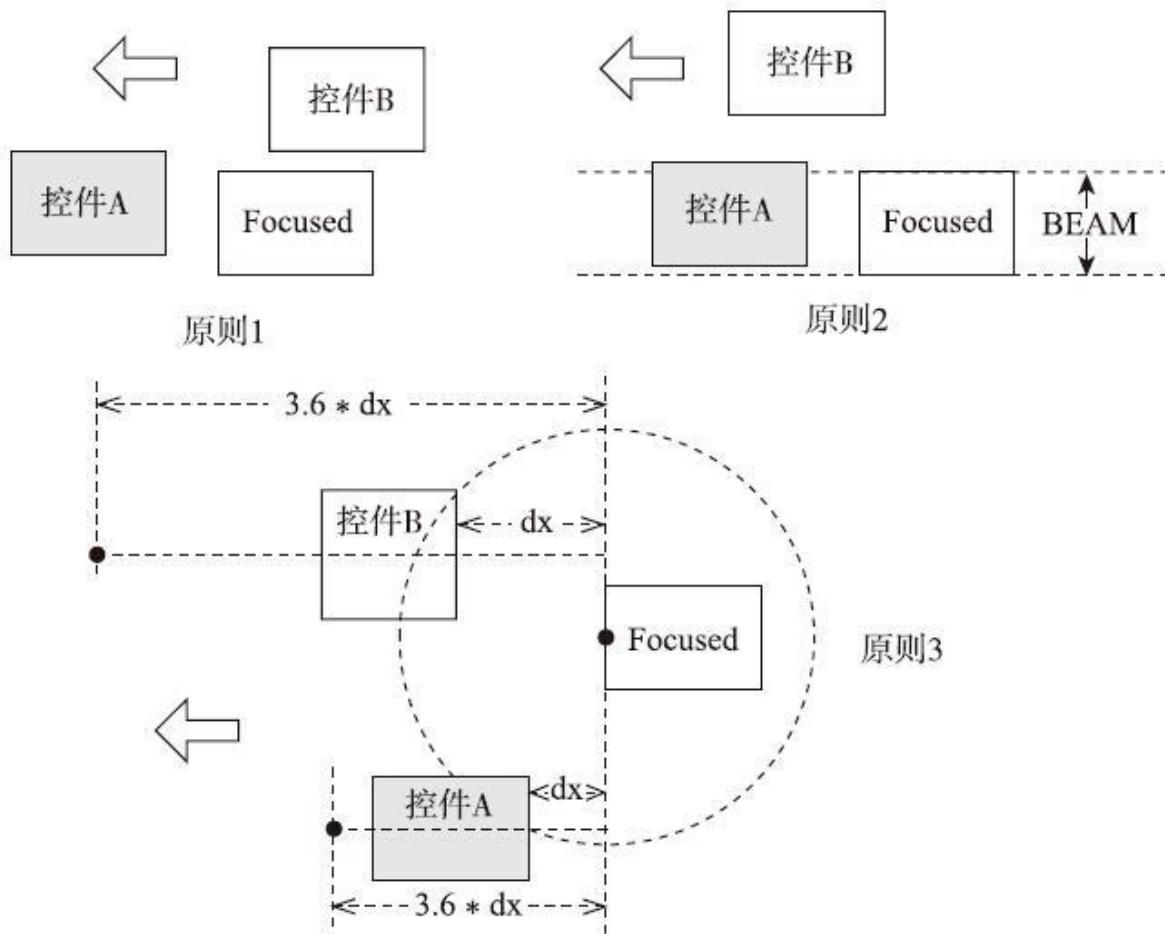


图6-24 更佳焦点候选的比较原则 (以LEFT方向为例)



注意

围绕BEAM的比较中（原则2）还有更细节的原则。例如，当一个控件与另一个方向的BEAM有交集时另一个控件是更佳候选，因为前者可以通过另一个方向查找到。读者可以通过阅读FocusFinder.beamBeats()方法学习其细节。

至此，下一个焦点控件的查找便结束了，总结其查找过程如下：

- 倘若开发者通过View.setNextFocusXXXId()显式地指定了某一方向上下一个焦点控件的Id，使用这一Id所表示的控件作为下一个焦点控件。
- 当开发者在指定的方向上没有指定下一个焦点控件时，则采用控件系统内置的焦点查找算法进行查找。
- 对于FORWARD/BACKWARD两个查找方向，根据当前焦点控件在focusables列表中的位置index，将位于index-1或index+1的控件作为下一个焦点控件。
- 对于LEFT、UP、RIGHT、DOWN 4个查找方向，将使用FocusFinder.isBetterCandidate()方法从focusables列表中根据控件位置选择一个最佳候选作为下一个焦点控件。

在选出下一个焦点控件之后，便可以通过调用它的requestFocus()方法将其设置为焦点控件了。



说明

相对于提供一个新的工具类FocusFinder，将查找下一个焦点的算法实现在ViewGroup中看起来是一个更加直观的做法。但是这样一来查找算法在实现过程中难免会和焦点的体系结构耦合起来。将其独立到FocusFinder工具类中使得其实现更加纯粹，而且独立于焦点的体系结构之后使得其适用范围更加广泛。例如，开发者可以通过FocusFinder.findNextFocus () 获取控件A的下一个焦点控件，而此时控件A不一定需要拥有焦点。假如这一算法与控件焦点的体系结构严重耦合，这一用法将是不存在的。

6.控件焦点的总结

至此，关于控件焦点的讨论便完成了。本节以View.requestFocus () 为起点深入探讨了控件系统对焦点的管理方式及其体系结构，并在最后介绍了查找下一个焦点的算法实现。相信读者对于控件焦点已经有了非常深刻的理解。这将为后续讨论输入事件派发的相关内容提供重要的基础。

6.5.3 输入事件派发的综述

在第5章关于输入系统的探讨中可以发现，按键事件与触摸事件采取了两种不同的派发策略。按键事件是基于焦点的派发，而触摸事件是基于位置的派发。控件系统中事件的派发一样采取了这两种策略。在深入讨论这两种策略在控件系统中的实现之前，首先讨论一下二者的共通内容——ViewRootImpl处理输入事件的总体流程。

第5章中介绍了输入系统的派发终点是InputEventReceiver。作为控件系统最高级别的管理者，ViewRootImpl便是InputEventReceiver的一个用户，它从InputEventReceiver中获取事件，然后将它们按照一定的流程派发给所有可能感兴趣的对象，包括View、PhoneWindow、Activity以及Dialog等。因此本节的探讨将从InputEventReceiver.onInputEvent () 开始。

1. ViewRootImpl的输入事件队列

在ViewRootImpl.setView () 中，新的窗口被创建之后，ViewRootImpl 使用WMS分配的InputChannel以及当前线程的Looper一起创建了 InputEventReceiver的子类WindowInputEvent-Receiver的一个实例，并将其保存在ViewRootImpl.mInputEventReceiver成员之中。这标志着从设备驱动到本窗口的输入事件通道的正式建立。至此每当有输入事件到来时，ViewRootImpl都可以通过WindowInputEventReceiver.onInputEvent () 回调得到这个事件并进行处理。参考其实现：

[ViewRootImpl.java-->WindowInputEventReceiver.onInputEvent()] public void onInputEvent(InputEvent event) {// 通过enqueueInputEvent将输入事件入队，注意第三个参数为trueenqueueInputEvent(event, this, 0, true); }

再看enqueueInputEvent () 的实现：

[ViewRootImpl.java-->ViewRootImpl.enqueueInputEvent()] void enqueueInputEvent(InputEvent event, InputEventReceiver receiver, int flags, boolean processImmediately) {/* ① 将InputEvent对应的InputEventReceiver封装为一个QueuedInputEvent。 QueuedInputEvent将是输入事件在ViewRootImpl中的存在形式 */QueuedInputEvent q = obtainQueuedInputEvent(event, receiver, flags);/* ② 将新建的QueuedInputEvent追加到mFirstPendingInputEvent所表示的一个单向链表之中。 ViewRootImpl将会沿着链表从头至尾地逐个处理输入事件 */QueuedInputEvent last = mFirstPendingInputEvent;if (last == null) {mFirstPendingInputEvent = q;} else { while (last.mNext != null) { last = last.mNext; } last.mNext = q;}if (processImmediately) { // ③ 倘若第三个参数为true，则直接在当前线程中开始对输入事件的处理工作doProcessInputEvents();} else { // ④ 否则将处理事件的请求发送给主线程的Handler，随后进行处理 scheduleProcessInputEvents();} }

此方法揭示了ViewRootImpl管理输入事件的方式。同InputDispatcher一样，在ViewRootImpl中也存在着一个输入事件队列

mFirstPendingInputEvent。输入事件在队列中以QueuedInputEvent的形式存在。QueuedInputEvent保存了输入事件的实例、接收事件的InputEventReceiver，以及一个next成员用于指向下一个QueuedInputEvent。

注意此方法的第三个参数processImmediately。对于从InputEventReceiver收到的正常事件来说，此参数永远为true，即入队的输入事件会立刻得到执行。而当此参数为false时，则会将事件的处理发送到主线程的Handler中随后处理。推迟事件处理的原因是什么呢？原来ViewRootImpl会将某些类型的输入事件转换成为另外一种输入事件，并将新的输入事件入队。由于此时仍处于旧有事件的处理过程中，倘若立即处理新事件会导致输入事件的递归处理，即前一个事件尚未处理完毕时开始了新的事件处理流程。为了避免这一情况，需要在入队时将processImmediately参数设置为false，在一切都完成之后再来处理新的事件。



说明

ViewRootImpl转换输入事件的一个例子是轨迹球（TrackBall）事件的处理。操作轨迹球时在驱动和输入系统层面会产生MotionEvent。ViewRootImpl根据MotionEvent.getSource () 得知这是一个来自轨迹球的事件后会根据其事件的数据将其转换为方向键（DPAD）的 KeyEvent，并将其通过enqueueInputEvent () 入队随后处理。这也是轨迹球的实际效果与方向键（或五向导航键）一致的原因。

接下来看doProcessInputEvent () 的实现：

```
[ViewRootImpl.java-->ViewRootImpl.doProcessInputEvent()]
void doProcessInputEvents() { // 遍历整个输入事件队列，逐个处理这些事件
    while (mCurrentInputEvent == null && mFirstPendingInputEvent != null) {
        QueuedInputEvent q = mFirstPendingInputEvent;
        mFirstPendingInputEvent = q.mNext;
        q.mNext = null; // ① 正在处理的输入事件会被保存为
        mCurrentInputEvent mCurrentInputEvent = q; // ② deliverInputEvent()方法
        将会完成单个事件的整个处理流程 deliverInputEvent(q); }..... }
```

显而易见，doProcessInputEvents () 方法不动则已，一动则一发不可收拾，直到将输入事件队列中的所有事件处理完毕之前不会退出，换言之在所有输入事件处理完成之前它不会放下对于主线程的占用权。这种看似粗犷的处理方式其实大有深意。ViewRootImpl最繁重的工作performTraversals () “遍历”就发生在主线程之上，而引发这一“遍历”操作的最常见的原因就是在输入事件处理时修改控件的内容。

doProcessInputEvents () 这种粗犷的处理方式使得performTraversals () 无法在单个输入事件处理后立刻得到执行，因输入事件所导致的requestLayout () 或invalidate () 操作会在输入事件全部处理完毕之后由一次performTraversals () 统一完成。当队列中存在较多事件时这种方式所带来的效率提升是不言而喻的。

2. 分道扬镳的事件处理

接下来分析deliverInputEvent () 的工作原理。参考代码如下：

```
[ViewRootImpl.java-->ViewRootImpl.deliverInputEvent()]
private void
deliverInputEvent(QueuedInputEvent q) {try { if (q.mEvent instanceof
KeyEvent) { // 处理按键事件 deliverKeyEvent(q); } else { final int source
= q.mEvent.getSource(); if ((source &
InputDevice.SOURCE_CLASS_POINTER) != 0) { // 处理触摸事件
deliverPointerEvent(q); } else if ((source &
InputDevice.SOURCE_CLASS_TRACKBALL) != 0) { // 处理轨迹球事件
deliverTrackballEvent(q); } else { // 处理其他Motion事件，如悬浮
(HOVER)、游戏手柄等 deliverGenericMotionEvent(q); } } } finally
{.....} }
```

可以看到在deliverInputEvent () 方法中不同类型的输入事件的处理终于分道扬镳了。根据InputEvent的子类类型或Source的不同，分别用4个

方法处理4种类型的事件。

·deliverKeyEvent ()，用于派发按键类型的事件。它选择的是基于焦点的派发策略。

·deliverPointerEvent ()，用于派发标准的触摸事件。它选择的是基于位置的派发策略。

·deliverTrackballEvent ()，用于派发轨迹球事件。它的实现比较特殊，在使用基于焦点的派发策略将事件派发之后，倘若没有任何一个派发目标处理此事件，它将会把事件转化为一个表示方向键的按键事件并添加到ViewRootImpl的输入事件队列中。

·deliverGenericMotionEvent ()，用于派发其他的Motion事件。这里一个大杂烩，悬浮事件、游戏手柄等会在这里被处理。

由于篇幅的原因，本节将只介绍deliverKeyEvent () 以及 deliverPointerEvent () 两个最常见同时也是最具代表性的事件处理流程。其他类型事件的处理方式直接采用或借鉴了这两种事件处理流程中所体现的思想和流程，感兴趣的读者可以自行研究。

3. 共同的终点——finishInputEvent ()

无论deliverInputEvent () 中分成了多少条不同的事件处理通道，应输入系统事件发送循环的要求，最终都要汇聚到一个方法中——

`ViewRootImpl.finishInputEvent ()`。这个方法用于向`InputDispatcher`发送输入事件处理完毕的反馈，同时也标志着一条输入事件的处理流程的终结。

参考`ViewRootImpl.finishInputEvent ()`的实现：

```
[ViewRootImpl.java-->ViewRootImpl.finishInputEvent()]
private void
finishInputEvent(QueuedInputEvent q, boolean handled) {/* 倘若被完成的
输入事件不是mCurrentInputEvent，则抛出异常。 ViewRootImpl不允许
事件的嵌套处理 */
if (q != mCurrentInputEvent) { throw new
IllegalStateException("finished input event out of order");} // ① 回收输入事
件并向InputDispatcher发送反馈
if (q.mReceiver != null) { /* 如果
mReceiver不为null，表示这是一个来自InputDispatcher的事件，需要向
InputDispatcher 发送反馈。事件实例的回收由InputEventReceiver托管完
成 */
q.mReceiver.finishInputEvent(q.mEvent, handled); } else { /* 如果
mReceiver为null，表示这是ViewRootImpl自行创建的事件，此时只要将
事件实例回收即可。 不需要惊动InputDispatcher */
q.mEvent.recycleIfNeededAfterDispatch(); } // ② 回收不再有效的
QueuedInputEvent实例。被回收的实例会组成一个以
mQueuedInputEventPool为 头部的单向链表中。下次使用
obtainQueuedInputEvent()时可以复用这个实例
*/recycleQueuedInputEvent(q); // 设置mCurrentInputEvent为
```

```
nullmCurrentInputEvent = null;// 如果队列中有了新的输入事件，则重新  
启动输入事件的派发if (mFirstPendingInputEvent != null) {  
    scheduleProcessInputEvents(); } }
```

至此，输入事件在ViewRootImpl中从onInputEvent () 开始到finishInputEvent () 终结的总体流程便终结了。不难得出输入事件在ViewRootImpl中派发的总体流程如图6-25所示。

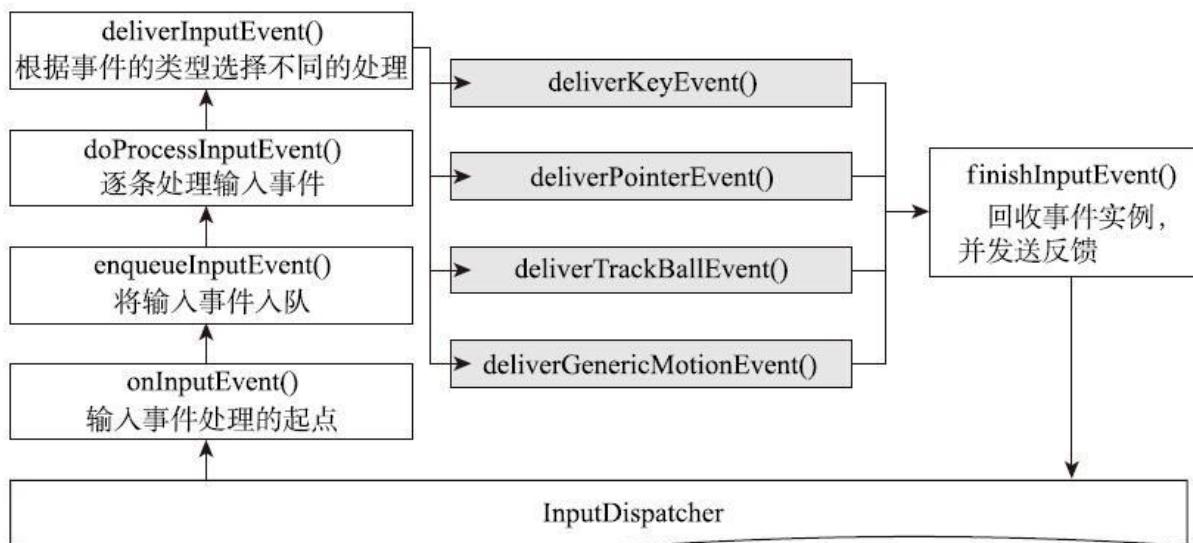


图6-25 输入事件在ViewRootImpl中的处理流程

接下来将深入探讨deliverKeyEvent () 与deliverPointerEvent () 的工作原理。

6.5.4 按键事件的派发

本节讨论按键事件在控件系统中的派发流程。按键事件是基于焦点进行派发的，即拥有焦点的控件将是事件的目标接受者，因此深刻理解控件焦点的管理原理非常重要。控件焦点的体系结构使得从根控件开始可以通过mFocused成员一路查找到最终的焦点控件，因此按键事件的派发流程就是沿着mFocused成员所构成的单向链表进行遍历的过程。这看似是一个十分简单的工作，但是因为有两个扰局者——输入法以及DecorView——的存在使得这个过程并没有想象中的简单。所幸在Activity及Dialog之外的用例下，控件树中并不存在DecorView，因此本节暂不考虑DecorView对事件派发所带来的影响。

参考ViewRootImpl.deliverKeyEvent () 方法的实现：

```
[ViewRootImpl.java-->ViewRootImpl.deliverKeyEvent()]
private void
deliverKeyEvent(QueuedInputEvent q) { // 从QueuedInputEvent中获取
    KeyEvent final KeyEvent event = (KeyEvent)q.mEvent;...../* 首先按键事
    件会尝试派发给输入法。将按键事件派发给输入法有三个条件，其中
    mView不为null，以及 mAdded为true比较好理解，它们分别表示
    ViewRootImpl中存在一棵控件树，并且其所在窗口已经被创建。而第
    三个条件是QueuedInputEvent.mFlags中不存在
    FLAG_DELIVER_POST ime标记。这一标记表示此按键事件之前已
    经派发给了输入法，但是输入法并没有消费它。所以当存在这一标记
    时无须再将事件派发给输入法 */
    if (mView != null && mAdded &&
```

```
(q.mFlags & QueuedInputEvent.FLAG_DELIVER_POST ime) == 0) { /*  
① 首先， View.dispatchKeyEventPreIme()方法将输入事件派发给控件  
树。这是控件树中的控件 第一次有机会处理按键事件 */ if  
(mView.dispatchKeyEventPreIme(event)) { // 如果控件消费了这一事件，  
则结束派发工作 finishInputEvent(q, true); return; } // ② 将按键事件派发  
给输入法。mLastWasImTarget表示此窗口可能是输入法的输入目标 if  
(mLastWasImTarget) { InputMethodManager imm =  
InputMethodManager.peekInstance(); if (imm != null) { // 通过  
InputMethodManager.dispatchKeyEvent()将事件派发给输入法。同时终  
止此次 事件的派发过程。注意此方法的最后一个参数  
mInputMethodCallback， 它是一个实现了  
InputMethodManager.FinishedEventCallback接口的回调。无论输入法是  
否消费这个 事件， 此回调都会收到通知， ViewRootImpl可以在收到这  
一通知之后销毁此事件并结束 派发， 或将事件重新派发给控件树 */  
final int seq = event.getSequenceNumber();  
imm.dispatchKeyEvent(mView.getContext(), seq, event,  
mInputMethodCallback); // 终止派发工作， 对于此事件的后续处理将在  
mInputMethodCallback回调中进行 return; } } }/* ③ 将输入事件派发给控  
件树。执行到这一步往往是由QueuedInputEvent.mFlags中存在FLAG_  
DELIVER_POST ime标记而deliverKeyEventPostIme()将会再次把事件
```

派发给控件树。这是控件 第二次有机会处理按键事件

```
 */deliverKeyEventPostIme(q); }
```

围绕着输入法， ViewRootImpl.deliverKeyEvent () 方法揭示了按键事件派发的三个阶段。首先控件树中的控件可以在输入法处理按键事件之前，通过View.dispatchKeyEventPreIme () 方法获得处理机会。倘若控件并未在此时消费事件，那么按键事件将会被派发给输入法。倘若输入法也没有消费这一事件，则ViewRootImpl.deliverKeyEventPostIme () 将使得控件第二次有机会处理此事件。

接下来将详细讨论这三个阶段的实现。

1.按键事件的初次派发

当按键事件第一次进入ViewRootImpl.deliverKeyEvent () 方法时，很明显FLAG_DELI-VER_POST_IME标记是不存在的。因此View.dispatchKeyEventPreIme () 方法会被执行。从而使得控件树中的控件可以优先于输入法获得事件的处理权利。毕竟这个事件是派发给本窗口，而不是输入法所在的窗口。

同View.requestFocus () 方法一样， View.dispatchKeyEventPreIme () 拥有ViewGroup和View两种不同的实现，参考这两种实现的代码：

```
[View.java-->View.dispatchKeyEventPreIme()] public boolean  
dispatchKeyEventPreIme(KeyEvent event) { // 调用onKeyPreIme()尝试消  
费这个事件。View.onPreIme()在View中是一个空的实现，直接返回  
false表示此控件并不希望在输入法之前消费此事件。View的子类可以  
重写这个方法以消费它，并通过返回true阻止输入法获得这一事件  
*/return onKeyPreIme(event.getKeyCode(), event); } [ViewGroup.java--  
>ViewGroup.dispatchKeyEventPreIme()] public boolean  
dispatchKeyEventPreIme(KeyEvent event) {if ((mPrivateFlags &  
(PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) == (PFLAG_FOCUSED |  
PFLAG_HAS_BOUNDS)) { /* 如果此ViewGroup是焦点的拥有者，则直  
接调用View.dispatchKeyEventPreIme()尝试消费此事件 */ return  
super.dispatchKeyEventPreIme(event);} else if (mFocused != null &&  
(mFocused.mPrivateFlags & PFLAG_HAS_BOUNDS) ==  
PFLAG_HAS_BOUNDS) { // 倘若mFocused不为null，则把  
dispatchKeyEventPreIme()传递给mFocused return  
mFocused.dispatchKeyEventPreIme(event);}return false; }
```

这两个实现都很简单，同时也体现了按键事件所采用的基于焦点的派
发策略——按键事件将由拥有焦点的控件进行处理。ViewGroup的实现
用于将事件沿着mFocused链表向着焦点控件所在的方向进行传递。而
View的实现则通过调用View.onKeyPreIme () 方法尝试进行事件的消
费。

开发者可以通过重写View.onKeyPreIme（）获得优先于输入法进行按键事件的处理。同样，Android并不阻止开发者通过重写View.dispatchKeyEventPreIme（）做同样的事情。二者的区别在于，dispatchKeyEventPreIme（）将先于onKeyPreIme（）获得事件的处理权。另外更重要的是，onKeyPreIme（）仅当控件拥有焦点时才会被调用，而在mFocused链表上的所有控件的dispatchKeyEventPreIme（）都会被调用。因此重写dispatchKeyEventPreIme（）往往用于在一个ViewGroup中拦截特定的按键事件进行处理并阻止其子控件获得它。



注意

dispatchKeyEventPreIme（）与onKeyPreIme（）两个方法的区别同样适用于其他与输入事件相关的dispatchXXX（）与onXXX（）。一般来讲，dispatchXXX（）的作用是为了从根控件开始将事件传递给目标控件，而onXXX（）则用于在目标控件中处理事件。

另外，ViewGroup得以将事件沿着mFocused链表传递按键事件的另外一个条件是PFLAG_HAS_BOUNDS标记的存在。PFLAG_HAS_BOUNDS

表示控件的mLeft/mTop/mRight/mBottom已被View.setFrame () 方法进行了设置，即控件已经完成了layout操作。未经过layout操作的控件可以理解为尚未初始化完毕，控件系统会拒绝这样的控件获取事件。

2. 输入法对按键事件的处理

回到ViewRootImpl.deliverKeyEvent ()，倘若没有任何控件在View.dispatchKeyEvent-PreIme () 的过程中消费这一事件，那么它将被派发给输入法。

为什么按键事件需要派发给输入法呢？通过第5章关于输入事件的分析可知，按键事件将会派发给处于焦点状态的窗口。而输入法所在的窗口是无法获取焦点的，因为它的LayoutParams.flags被InputMethodService放置了FLAG_NOT_FOCUSABLE标记（参考SoftInputWindow.initDockWindow () 方法的实现），因此在默认的派发机制下，输入法窗口是无法获取按键事件的，包括其他非基于位置的事件如轨迹球事件等。不允许其获得焦点是由于输入法仅仅是一个辅助工具，它的存在不应对目标窗口的功能或行为产生影响。然而输入法确实拥有处理按键事件的需求，例如通过BACK键将输入法关闭，或通过方向键在输入法中进行选词等。为了解决这一矛盾，ViewRootImpl将会在收到事件后首先转发给输入法，当输入法对此事件不感兴趣时再将其发送给控件树。不过，正如上一节所述，作为事

件的正统接收者，控件树可以通过重写View.dispatchKeyEventPreIme () 或View.onKeyPreIme () 先于输入法处理事件。

派发给输入法的条件是mLastWasImTarget成员为true，即本窗口可能是输入法的输入目标。这一成员的取值来自于窗口的LayoutParams.flags中FLAG_NOT_FOCUSABLE及FLAG_ALT_FOCUSABLE_IM两个标记的存在情况。读者可以参考

WindowManager.LayoutParams.mayUseInputMethod () 的实现了解这两个标记的功能。当本窗口不能作为输入法的输入目标时，便不会有输入法覆盖其上，自然不需要输入法对按键事件进行处理。

InputMethodManager.dispatchKeyEvent () 方法将会通过Binder将按键事件发送给当前输入法所在的InputMethodService，并在那里的onKeyXXX () 系列事件处理方法中得到处理。这一过程的细节超出了本节的讨论范围，读者可以自行研究。

在输入法完成事件的处理之后，无论是否消费了这一事件，都会通过InputMethodCallback.finishedEvent () 回调将其处理结果通知ViewRootImpl。后者会通过handleImeFinishedEvent () 方法继续对事件的派发工作。参考实现如下：

```
[ViewRootImpl.java-->ViewRootImpl.handleImeFinishedEvent()]
void handleImeFinishedEvent(int seq, boolean handled) {final QueuedInputEvent
```

```
q = mCurrentInputEvent;// 继续派发工作的前提是seq必须与
mCurrentInputEvent一致，否则这一回调将会被忽略if (q != null &&
q.mEvent.getSequenceNumber() == seq) { if (handled) { // ① 如果输入法
消费了这一事件，则终止此事件的派发工作 finishInputEvent(q, true); }
else { if (q.mEvent instanceof KeyEvent) { KeyEvent event =
(KeyEvent)q.mEvent; ..... // ② 通过deliverKeyEventPostIme()将输入事件
重新派发给控件树 deliverKeyEventPostIme(q); } else { ..... // 其他类型的
输入事件 } } } else {.....} }
```

显而易见，当输入法完成事件处理后，ViewRootImpl有两种可能的处理：倘若事件已被输入法消费，则直接终止后续的派发工作；反之，则通过deliverKeyEventPostIme () 将事件重新派发给控件树。

3.按键事件的最终派发

deliverKeyEventPostIme () 负责按键事件的最终派发。在这里，开发者最熟悉的View.onKeyDown () 系列回调以及OnKeyListener监听者都会得到触发。同时一些系统内置的按键功能也将在这里进行处理。参考实现如下：

```
[ViewRootImpl.java-->ViewRootImpl.deliverKeyEventPostIme()]
private
void deliverKeyEventPostIme(QueuedInputEvent q) {final KeyEvent event
= (KeyEvent)q.mEvent;...../* ① 首先，检查此按键事件是否会退出触摸
```

模式。一般来说，方向键、字母键的按下事件都标识着用户将会以按键的方式操纵Android，此时需要退出触摸模式 */if

```
(checkForLeavingTouchModeAndConsume(event)) { // 当这一事件导致触摸模式的退出时，意味着此事件已经被消费了，因此终止这一事件的派发工作 finishInputEvent(q, true); return; } // ② 在正式开始事件的派发之前，首先让mFallbackEventHandler过目一下
```

```
mFallbackEventHandler.preDispatchKeyEvent(event); // ③ 将事件派发给控件树。这是本方法最重要的工作 if (mView.dispatchKeyEvent(event)) { // 如果有控件消费了这一事件，则终止事件的派发 finishInputEvent(q, true); return; } ..... // ④ 如果没有控件消费这一事件，则尝试将事件派发给mFallbackEventHandler if
```

```
(mFallbackEventHandler.dispatchKeyEvent(event)) { // 如果事件被消费，则终止事件的派发 finishInputEvent(q, true); return; } // ⑤ 处理方向键的按下事件。用于使焦点在控件之间游走 if (event.getAction() == KeyEvent.ACTION_DOWN) { int direction = 0; // 根据 KeyEvent.getKeyCode() 确定焦点游走的方向 switch (event.getKeyCode()) { case KeyEvent.KEYCODE_DPAD_LEFT: if (event.hasNoModifiers()) { /* 按下左键，则表示将焦点移动到当前焦点控件左侧的控件上。 */ KeyEvent.hasNoModifiers() 表示此时 Alt/Ctrl/Shift 没有按下 */ direction = View.FOCUS_LEFT; } break; case KeyEvent.KEYCODE_DPAD_RIGHT: ..... // 识别其他方向键 } if (direction != 0) { // 获取当前的焦点控件 View
```

```
focused = mView.findFocus(); if (focused != null) { /* 熟悉的  
View.focusSearch()方法，它将根据6.5.2节中所介绍的算法查找下一个应  
当 获取焦点的控件 */ View v = focused.focusSearch(direction); if (v !=  
null && v != focused) { ..... // 通过View.requestFocus()方法使此控件获取  
焦点 if (v.requestFocus(direction, mTempRect)) { ..... // 结束事件的派发  
finishInputEvent(q, true); return; } } ..... } } } // 最终，此事件没有任何一  
个对象对其感兴趣，终止事件的派发finishInputEvent(q, false); }
```

可以说，View.dispatchKeyEventPreIme () 以及
InputMethodManager.dispatchKeyEvent () 都是按键事件派发的前奏而
已。ViewRootImpl.deliverKeyEventPostIme () 才是重头戏。在这个方
法中，可以根据优先级列出如下几个可能消费事件的对象或行为：

- TouchMode。如果导致了触摸模式的终止，此事件会被消费。
- 控件树中的控件。View.dispatchKeyEvent () 方法会将事件派发给控
件树。
- mFallbackEventHandler。它在ViewRootImpl的构造函数中通过
PolicyManager.makeNewFallbackEventHandler () 创建，是一个
PhoneFallbackEventHandler类的实例。与PhoneWindowManager类似，
它提供了一个进行系统级按键处理的场所，只不过它的处理优先级低
得多，当需要为某个按键定义一个系统级的功能，并允许应用程序修

改此按键的功能时，可以在PhoneFallbackEventHandler类中进行实现，例如使用音量键调整系统音量的工作就在这里完成。因此应用程序可以将音量键挪作他用，例如在相机中用来调整焦距。需要注意的是，与PhoneWindowManager在系统中只有一个实例不同，每个ViewRootImpl都有各自的PhoneFallbackEventHandler实例，因此它并不适合存储一些系统级的状态。

·焦点游走。它主要感兴趣的是方向键和TAB键的按下事件，它将根据按键选择一个焦点的查找方向，然后通过View.focusSearch () 方法选择一个控件并使其获得焦点。

其中，最感兴趣的内容自然是负责将事件派发给控件树的View.dispatchKeyEvent () 。它与View.dispatchKeyEventPreIme () 十分相似，拥有ViewGroup与View两种不同的实现，参考它们的代码：

```
[View.java-->View.dispatchKeyEvent()]
public boolean
dispatchKeyEvent(KeyEvent event) {.....// ① 首先由OnKeyListener监听者
尝试处理事件。它可以通过View.setOnKeyListener()进行设置
ListenerInfo li = mListenerInfo;if (li != null && li.mOnKeyListener != null
&& (mViewFlags & ENABLED_MASK) == ENABLED &&
li.mOnKeyListener.onKey(this, event.getKeyCode(), event)) { // 如果
OnKeyListener消费了事件则返回true return true;}// ② 通过
event.dispatch()方法将事件发送给View的指定回调，如
```

```
onKeyDown()/onKeyUp()等if (event.dispatch(this, mAttachInfo != null ?  
mAttachInfo.mKeyDispatchState : null, this)) { // 如果控件的  
onKeyDown()/onKeyUp()等回调消费了事件则返回true return true;}// 此  
控件没有消费这个事件return false; } [ViewGroup.java-->  
>ViewGroup.dispatchKeyEvent()] public boolean  
dispatchKeyEvent(KeyEvent event) {.....if ((mPrivateFlags &  
(PFLAG_FOCUSED | PFLAG_HAS_BOUNDS)) == (PFLAG_FOCUSED |  
PFLAG_HAS_BOUNDS)) { // ① 如果此ViewGroup拥有焦点，则调用  
View.dispatchKeyEvent()尝试消费事件 if (super.dispatchKeyEvent(event))  
{ return true; } } else if (mFocused != null && (mFocused.mPrivateFlags &  
PFLAG_HAS_BOUNDS) == PFLAG_HAS_BOUNDS) { // ② 倘若此  
ViewGroup不拥有焦点，则将事件沿着mFocused链表进行传递 if  
(mFocused.dispatchKeyEvent(event)) { return true; } }.....return false; }
```

显而易见，ViewGroup.dispatchKeyEvent () 的实现与
ViewGroup.dispatchKeyEventPreIme () 别无二致。它将按键事件沿着
mFocused链表向尾端传递并沿途调用ViewGroup.dispatchKeyEvent ()
方法。并最终由焦点拥有者的View.dispatchKeyEvent () 尝试事件的消
费。

View.dispatchKeyEvent () 的本质与View.dispatchKeyEventPreIme ()
一致，只不过由于事件处理回调的不同而有所差别。在这个方法里，

通过View.setOnKeyListener () 所设置的监听者具有较高的优先级，当这个监听者对事件不感兴趣时，通过KeyEvent.dispatch () 方法引发的View.onKeyDown () /onKeyUp () 才有机会进行事件的处理。

4.按键事件派发的总结

至此，关于按键事件的派发过程的介绍便结束了。总结其派发与处理流程如下：

- 首先按键事件将通过View.dispatchKeyEventPreIme () 派发给控件树。此时开发者可以进行事件处理的场所是View.dispatchKeyEventPreIme () 或View.onKeyPreIme () 。
- 然后按键事件将通过InputManager.dispatchKeyEvent () 派发给输入法。此时处理事件的场所是当前输入法所在的InputMethodService的onKeyDown () /onKeyUp () 等系列回调。
- 之后按键事件将被View.checkForLeavingTouchModeAndConsume () 方法用来尝试退出触摸模式。
- 再之后按键事件将被View.dispatchKeyEvent () 在此派发给控件树。此时开发者可以进行事件处理的场所是View.dispatchKeyEvent () ，通过View.setOnKeyListener () 方法设置的OnKeyListener，以及View.onKeyDown () /onKeyUp () 等系列回调。

·PhoneFallbackEventHandler将在上述对象都没有消费事件时尝试对事件进行处理。

·最后ViewRootImpl将尝试通过按键事件使焦点在控件之间游走。

另外，按键事件是基于焦点进行派发的。事件将从根控件开始沿着mFocused链表向拥有焦点的控件进行传递，沿途的ViewGroup都将有机会通过重写其dispatchKey-EventPreIme () 或dispatchKeyEvent () 方法进行拦截处理。而onKeyPreIme () /onKeyDown () 等系列回调仅会发生在拥有焦点的控件上。

另外，如果窗口属于一个Activity或者Dialog，其根控件是一个DecorView，它的dispatch-KeyEvent () 方法还会尝试将事件派发给其他组件。这部分内容将在6.6.1节介绍。

6.5.5 触摸事件的派发

触摸事件是基于位置进行派发的。相对于按键事件可以通过mFocused链表进行传递并最终到达焦点控件，触摸事件的派发过程由于坐标系变换、多点触摸的存在而复杂得多。另外，ViewRootImpl并不需要将触摸事件派发给输入法，因为InputDispatcher会将点击到输入法的窗口的事件直接派发给它，因而不需要ViewRootImpl转发。

触摸事件派发的起点是ViewRootImpl.dispatchPointerEvent () 。参考实现如下：

```
[ViewRootImpl.java-->ViewRootImpl.dispatchPointerEvent()]
private void
deliverPointerEvent(QueuedInputEvent q) {final MotionEvent event =
(MotionEvent)q.mEvent;final boolean isTouchEvent =
event.isTouchEvent();.....// ① 当这是一个按下事件时，将会进入触摸模
式。此时将会重新设置焦点控件final int action = event.getAction();if
(action == MotionEvent.ACTION_DOWN || action ==
MotionEvent.ACTION_SCROLL) { /* ensureTouchMode()负责进入或退
出触摸模式，它会重新设置焦点控件，并将触摸模式同步到WMS，以
便以后所创建的窗口可以从WMS得知应当工作在何种模式下 */
ensureTouchMode(true); }/* ViewRootImpl所收到的触摸事件位于窗口的
坐标系下。将其派发给根控件时需要将其坐标转换到根控件下。根控
件的坐标系与窗口坐标系的区别在于Y方向上的滚动量mCurScrollY */if
(mCurScrollY != 0) { event.offsetLocation(0, mCurScrollY); }.....// ② 将触
摸事件派发给控件树boolean handled =
mView.dispatchPointerEvent(event);if (handled) { // 事件已被消费，结束
派发工作 finishInputEvent(q, true); return; } // 事件没有被消费，结束派发
工作finishInputEvent(q, false); }
```

由于不需要将事件派发给输入法，ViewRootImpl.deliverPointerEvent() 的工作相比deliverKeyEvent() 要简单得多。一是强制进入触摸模式，二是通过View.dispatchPointerEvent() 将触摸事件派发给控件树。

再看View.dispatchPointerEvent() 的实现：

```
[View.java-->View.dispatchPointerEvent()]
public final boolean
dispatchPointerEvent(MotionEvent event) { if (event.isTouchEvent()) { // 如
果是一个触摸事件，则通过dispatchTouchEvent()进行派发 return
dispatchTouchEvent(event); } else { // 否则通过
dispatchGenericMotionEvent()进行派发 return
dispatchGenericMotionEvent(event); } }
```

这里所谓的PointerEvent其实包含以MotionEvent.getAction() 进行区分的两种事件：一种是实际的触摸事件如ACTION_DOWN/MOVE/UP等表示实际接触到屏幕所产生的事件，而另外一种则是ACTION_HOVER_ENTER/MOVE/UP等未接触到屏幕所产生的事件。其中第一种事件才是真正意义上的触摸事件，本节将以此类事件为例进行探讨，即主要分析View.dispatchTouchEvent() 的实现。至于HOVER类型的实现，其派发思想与实现原理与触摸事件的派发十分相似，读者可以类比研究。

1.MotionEvent与触摸事件的序列

触摸事件的派发十分复杂，因此有必要在继续分析代码之前先介绍一下关于触摸事件派发的基本知识及其派发思想，这样才不会被代码中复杂的逻辑所迷惑。

触摸事件被封装为一个继承自InputEvent类的MotionEvent中。它包含了多种用于描述一次触摸的详细信息，其中最基本的两个信息是通过getAction () 方法获取的动作信息以及通过getX () /getY () 方法所获得的位置信息。由于多点触摸的存在，这几个方法在实际的使用过程中与KeyEvent类中的getAction () 以及getKeyCode () 有所不同。首先通过MotionEvent.getAction () 所获得的动作信息是一个复合值，它在低8位描述了实际的动作如ACTION_DOWN、ACTION_UP等，而在其9~16位描述了引发此事件的触控点从0开始的索引号。因此在实际的使用过程中，往往需要将这两个信息分离出来。开发者可以通过MotionEvent.getActionMasked () 获取实际的动作，然后通过MotionEvent.getActionIndex () 获取此事件所代表的触控点的索引号。另外，虽然一个MotionEvent由一个触控点所引发，然而它却包含了所有触控点的位置信息，以便开发者可以在收到一个MotionEvent时根据所有触控点的信息进行计算与决策，因此MotionEvent.getX () 与getY () 两个方法可以接受触控点的索引号作为参数，以返回特定触摸点的触摸位置。

触控点的索引号是什么呢？在MotionEvent的内部有一个数组mSamplePointerCoords，其每一个元素是一个PointerProperties结构体，描述了一个触控点的信息。所谓的索引号就是一个触控点在这个数组中所在的位置。在多点触摸的过程中，伴随着用户手指的抬起与按下，一个触控点在数组中的位置不是一成不变的。例如用户所按下的第二个点B的索引号为1，当用户首先抬起其所按下的第一个点A（其索引号为0）之后，A在mSamplePointerCoords中的信息会被删除，从而使得点B的索引号变为0，因此触控点的索引号并不能用来识别或追踪一个特定触控点。开发者需要通过触控点的ID达到识别或追踪一个特定触控点的目的。触控点的ID存储在PointerProperties结构体中，可以通过MotionEvent.getPointerId () 方法获得。作为触控点的属性之一，与getX () /getY () 类似，这一方法需要索引号作为参数。

可见MotionEvent的使用远比KeyEvent复杂。一般而言，开发者在收到一个MotionEvent之后，首先需要通过MotionEvent.getActionMasked () 获取其实际的动作，然后通过MotionEvent.getPointerIndex () 获取引发这一事件的触控点的索引号，然后再根据索引号获取触控点的坐标信息，以及其ID。开发者最终关心的信息是实际的动作，坐标信息以及触控点的ID，索引号只不过是获取这些信息的一个临时的工具而已。



注意

当MotionEvent所携带的动作是ACTION_MOVE时，其getAction () 所获得的动作信息并不包含触控点的索引，因为ACTION_MOVE并不会导致增加或减少触控点，不过它仍然保存了所有触控点的位置/ID等信息。开发者可以通过MotionEvent.getPointerCount () 得知此时有多少触控点处于活动状态，并通过一个for循环遍历每一个触控点的位置/ID信息。从这一事实可以得知，getAction () 中所包含的触控点索引号其实是为了通知开发者是否产生了新的触控点（按下），或某个触控点被移除（抬起）。

接下来是触摸事件的序列。它是用户从第一个手指按下开始到最后一个手指抬起这一过程中所产生的MotionEvent序列。最简的情况是单点触摸的事件序列，它从一个ACTION_DOWN开始，经历一系列的ACTION_MOVE，以一个ACTION_UP结束。而多点触摸时这一序列则复杂一些，它同样以一个ACTION_DOWN开始，经历一系列的ACTION_MOVE，而当用户另一个手指按下时会产生一个ACTION_POINTER_DOWN，当某一手指抬起时会产生一个

ACTION_POINTER_UP，当最后一个手指抬起时，以一个ACTION_UP结束事件序列。在这个过程中，开发者可以通过事件所携带的触控点的ID追踪某一个触控点的始末。图6-26描述了拥有三个触控点的事件序列，从中可以看出，序列以ACTION_DOWN开始以ACTION_UP结束，某一触控点在中途的按下与抬起由ACTION_POINTER_DOWN/UP事件表示，它们在getAction () 中所携带的索引号指示了发生这一动作的触控点的索引，并且这一索引随着当前处于活动状态的触控点的数量的变化而变化，但是触控点的ID则始终保持不变。同时整个事件序列中每一个事件都包含了所有触控点的信息。

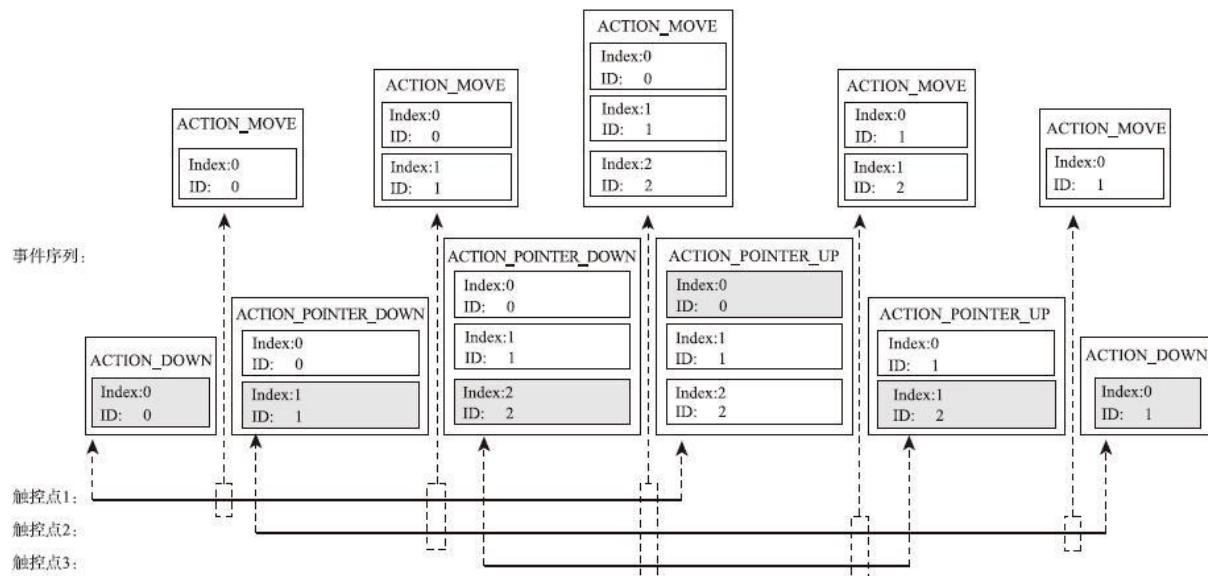


图6-26 拥有三个触控点的事件序列

不难想到，多点触摸下的事件序列中每一个触控点的信息都足以独立形成一个事件序列。仍以图6-26为例，整个事件序列可以通过某种手段

将触控点1拆分出来，从而形成两条新的事件序列，即触控点1的信息所组成的一条单点序列，以及由触控点2和3所组成的一条双点序列。这两条序列被称为原始序列的子序列，而这一行为则被称为事件序列的拆分（Split）。事件序列的拆分是通过拆分序列中的每一个 MotionEvent实现的。MotionEvent的拆分可以通过MotionEvent.split（）方法完成，它可以从当前MotionEvent中产生一个新的仅包含特定触控点信息的MotionEvent，而这个新产生的MotionEvent则成为子序列的一部分。为什么会出现事件序列的拆分呢？参考图6-27的左图，一个 ViewGroup中包含两个控件，当用户的两个手指分别按在View1与View2之上时，因为两个触控点都落在ViewGroup之内，因此ViewGroup会收到一条双点的事件序列，当ViewGroup将事件派发给View1和View2时，就必须将其拆分为两个单点序列，并分别派发给View1和View2。

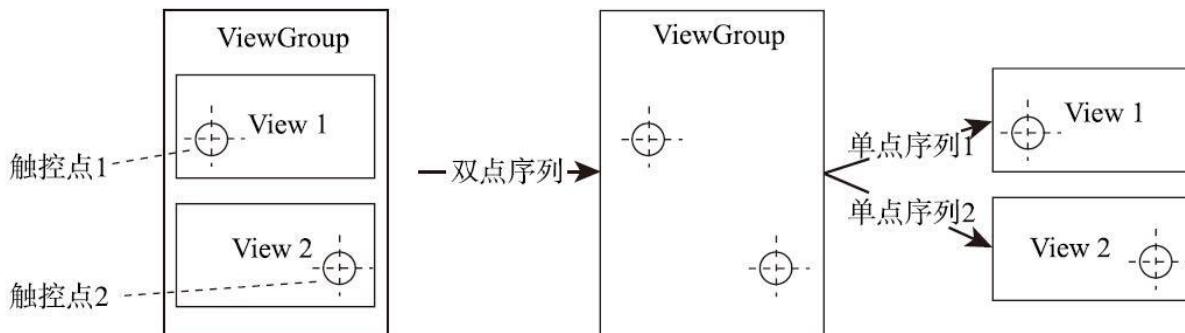


图6-27 触摸事件序列的拆分

之所以讨论输入事件序列的概念，是由于Android在派发触摸事件时有一个很重要的原则，就是事件序列的不可中断性，即一旦一个控件决

定了接受一个触控点的ACTION_DOWN/或ACTION_POINTER_DOWN事件（在事件处理函数中返回true），表示控件将接受序列的所有后续事件，即便触控点移动到控件区域之外也是如此。理解事件序列的概念对在后续代码分析过程中把握这一原则十分重要。



注意

触摸事件的序列除ACTION_UP以外还有一种结束标志——ACTION_CANCEL。比如，一个正在接受事件序列的控件从控件树中被移除，或者发生了Activity切换等，那么它将收到ACTION_CANCEL而不是ACTION_UP，此时控件需要中断对事件的处理并将自己恢复到接受事件序列之前的状态。

2. 控件对触摸事件的接收与处理

回过头来继续代码的讨论，ViewRootImpl将触摸事件交给了根控件的View.dispatchPointerEvent()，View.dispatchPointerEvent()又将事件交给了View.dispatchTouchEvent()。同View.dispatchKeyEvent()一样，View.dispatchTouchEvent()拥有ViewGroup以及View两种实

现。类比可知ViewGroup的实现负责将触摸事件沿着控件树向子控件进行派发，而View的实现则主要用于事件接收与处理工作。

首先分析相对简单的View实现。

```
[View.java-->View.dispatchTouchEvent()] public boolean  
dispatchTouchEvent(MotionEvent event) { ...../* ① 首先触摸事件必须经  
过onFilterTouchEventForSecurity()过滤。出于对最终用户的信息安全角  
度的考虑，当本窗口位于另外一个非全屏窗口之下时，可能会阻止控  
件处理触摸事件 */if (onFilterTouchEventForSecurity(event)) { // ② 尝试  
让此控件的OnTouchListener处理触摸事件 ListenerInfo li =  
mListenerInfo; if (li != null && li.mOnTouchListener != null &&  
(mViewFlags & ENABLED_MASK) == ENABLED &&  
li.mOnTouchListener.onTouch(this, event)) { return true; } // ③ 倘若  
OnTouchListener对事件不感兴趣，则尝试令onTouchEvent()回调处理事  
件 if (onTouchEvent(event)) { return true; }.....return false; }
```

可见View.dispatchTouchEvent () 与View.dispatchKeyEvent () 的实现
如出一辙，用于使OnTouchListener或onTouchEvent () 进行事件的处
理。区别在于它多了一道由onFilterTouchEventForSecurity () 完成的
验证程序。这道验证程序是检查触摸事件是否带有
FLAG_WINDOW_IS_OBSCURED标记（这一标记由InputDispatcher设
置，依据是派发目标窗口在WMS中WindowState.mObscured成员的取

值，参考本书第4章），以及控件的mViewFlags中是否存在FILTER_TOUCHES_WHEN_OBSCURED标记，当两者同时存在时，将会阻止事件被此控件处理。事件中存在FLAG_WINDOW_IS_OBSCURED标记表明此窗口部分或完整地被另外一个窗口所遮挡，此时用户有可能因为无法看到当前窗口的一些敏感信息或被遮挡窗口的恶意信息所蒙骗而进行一些不安全的操作，如图6-28所示。因此，Android提供了这一机制避免用户的点击事件得到响应从而降低安全风险。开发者可以通过在执行敏感行为的控件上调用View.setFilterTouchesWhenObscured（）方法在mViewFlags中添加FILTER_TOUCHES_WHEN_OBSCURED标记，以便在这个控件上启用这一机制。

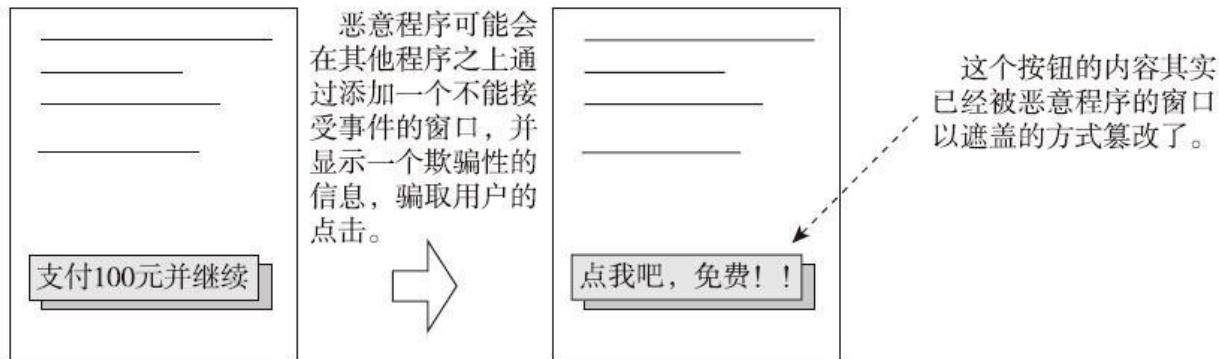


图6-28 通过新窗口篡改其他程序的敏感信息

3.ViewGroup对触摸事件的派发

接下来讨论dispatchTouchEvent () 的ViewGroup版本的实现。

ViewGroup的主要工作是将触摸事件派发给合适的子控件。在本节中，读者将会看到ViewGroup对前面所介绍的知识的具体运用，以及对事件序列不可中断性原则的坚持。

事件的派发分为确定派发目标和执行派发两个工作。

对触摸事件来说，根据序列不可中断性原则，确定派发目标发生在收到ACTION_DOWN或ACTION_POINTER_DOWN的时刻。此时 ViewGroup会按照逆绘制顺序依次查找事件坐标所落在的子控件，并将事件发送给子控件的dispatchTouchEvent () 方法，然后根据返回值确定第一个愿意接受这一序列的子控件，将其确定后续事件为派发目标。一旦通过ACTION_DOWN或ACTION_POINTER_DOWN确定派发目标，ViewGroup会将此触控点的ID与目标建立绑定关系，属于此触控点的事件序列都会发送给这一目标。ViewGroup通过一个TouchTarget类的实例来描述这一个绑定，这个类保存了一个触控点ID的列表以及一个View实例，以此实现从触控点ID到目标控件的映射。

执行派发的工作主要是将事件传递给目标的dispatchTouchEvent () 方法。由于多点触控的存在，执行派发时可能需要将MotionEvent进行拆分，将ViewGroup所收到的事件序列拆分成多个子序列并发送给多个子控件。因此不难理解ViewGroup在其派发过程中可能维护着多个TouchTarget实例。TouchTarget与ViewRootImpl的QueuedInputEvent类一

样存在着一个next成员，即它是一个单向链表。ViewGroup将所有的TouchTarget存储在一个以mFirstTouchTarget为表头的单向链表中。

另外在ViewGroup.dispatchTouchEvent () 中会通过onInterceptTouchEvent () 尝试对输入事件进行截获。为了减少复杂度，本节不会对事件的截取进行讨论。

本节将按照上述的两个工作对ViewGroup.dispatchTouchEvent () 进行讨论。

(1) 派发目标的确定

参考ViewGroup.dispatchTouchEvent () 的代码：

```
[ViewGroup.java-->ViewGroup.dispatchTouchEvent()]
public boolean dispatchTouchEvent(MotionEvent ev) {
    .....boolean handled = false;// 同样， ViewGroup也会对遮盖状态进行检查与过滤
    if (onFilterTouchEventForSecurity(ev)) {
        final int action = ev.getAction();
        // 剔除触控点的索引号以获取实际的动作
        final int actionMasked = action &
MotionEvent.ACTION_MASK;
        if (actionMasked ==
MotionEvent.ACTION_DOWN) { /* ① ACTION_DOWN意味一条崭新的事件序列的开始。此时ViewGroup会重置所有与触摸事件 派发相关
的状态，包括清空TouchTarget列表，这样一来ViewGroup便准备好进行一次崭新的 触摸事件派发工作了 */
            cancelAndClearTouchTargets(ev);
        }
    }
}
```

```
resetTouchState(); } ..... // 检查此ViewGroup是否需要对事件进行截取，  
暂不考虑截取的情况 /* canceled表示ViewGroup所收到的这一事件序列  
是否被取消（由于被移出控件树或发生了Activity切换等），为了简单起  
见，本节忽略被取消的情况 */ final boolean canceled =  
  
resetCancelNextUpFlag(this) || actionMasked ==  
  
MotionEvent.ACTION_CANCEL; /* split表示此控件树是否启用前述的  
事件序列的拆分机制，开发者可以通过setMotionEventSpli-  
ttingEnabled()方法启用或禁用这一机制 */ final boolean split =  
(mGroupFlags & FLAG_SPLIT_MOTION_EVENTS) != 0; /* 如果此次事  
件产生了新的派发目标（仅发生于ACTION_DOWN或  
ACTION_POINTER_DOWN），那么新的TouchTarget实例将保存在这  
里 */ TouchTarget newTouchTarget = null; /* 由于确定派发目标使用了子  
控件的dispatchTouchEvent()，因此当确定派发目标之后这一事件实际  
上已经完成派发了。这种情况下此变量将会被设置为true，以跳过后续  
的派发过程 */ boolean alreadyDispatchedToNewTouchTarget = false; // 倘  
若事件序列没有被取消，也没有被当前ViewGroup所截取，才有进行派  
发目标查找的必要 if (!canceled && !intercepted) { /* 如果事件的实际动  
作是ACTION_DOWN或者ACTION_POINTER_DOWN，标志着一个子  
序列的开始，此时需要进行派发目标的确定 */ if (actionMasked ==  
MotionEvent.ACTION_DOWN || (split && actionMasked ==  
MotionEvent.ACTION_POINTER_DOWN) || actionMasked ==
```

```
MotionEvent.ACTION_HOVER_MOVE) { // 获取这一按下事件的触控点  
    的索引号 final int actionIndex = ev.getActionIndex(); /* ② 通过索引号获  
    取触控点的ID。为了能够在一个整型变量中存储一个ID列表，这里通  
    过 将1进行左移若干个位的方式将ID转换为2的ID次方的形式并存储在  
    idBitsToAssign 变量中。当找到一个派发目标之后，会将这个  
    idBitsToAssign添加到派发目标所对应的TouchTarget中，从而使得这一  
    触控点被绑定在TouchTarget上。由此可见，Android 控件系统最多可以  
    支持32个触控点。注意，当split为false及ViewGroup没有启用序列的拆  
    分时，idBitsToAssign被设 置为TouchTarget.ALL_POINTER_IDS，意思是  
    是所有触控点的事件都会被派发给后续被 确定的目标控件 */ final int  
    idBitsToAssign = split ? 1 << ev.getPointerId(actionIndex) :  
    TouchTarget.ALL_POINTER_IDS; ..... /* 接下来就要开始对子控件按照  
    逆绘制顺序进行遍历，检查哪一个控件对这一新的事件子序 列感兴趣  
    */ final int childrenCount = mChildrenCount; if (childrenCount != 0) { final  
    View[] children = mChildren; /* 由于触摸事件是基于位置进行派发目标  
    的查找，因此必须获取事件的坐标。注意这里通过触控点的索引号获  
    取坐标 */ final float x = ev.getX(actionIndex); final float y =  
    ev.getY(actionIndex); /* 按照逆绘制顺序的遍历循环，注意顺序是从  
    childrenCount-1开始到0 final boolean customOrder =  
    isChildrenDrawingOrderEnabled(); for (int i = childrenCount - 1; i >= 0; i--)  
    { final int childIndex = customOrder ?
```

```
getChildDrawingOrder(childrenCount, i) : i; final View child =
children[childIndex]; /* ③ 首先检查事件坐标是否落在控件之内。如果
没有位于控件内则继续查找下一个控件 */ if
(!canViewReceivePointerEvents(child) ||
!isTransformedTouchPointInView(x, y, child, null)) { continue; } /* ④ 从
mFirstTouchTarget列表中查找控件所对应的TouchTarget。倘若子控件所
对应的TouchTarget已经存在，表明此控件已经在接收另外一个事件子
序列， ViewGroup会默认此控件对这一条子序列也感兴趣。此时将触
控点ID绑定在其上，并终止派发目标的查找。后续的派发工作会据此
将此事件派发给这一控件 */ newTouchTarget = getTouchTarget(child); if
(newTouchTarget != null) { newTouchTarget.pointerIdBits |=
idBitsToAssign; break; } ..... /* ⑤ 使用dispatchTransformedTouchEvent()
方法尝试将事件派发给当前子控件。此方法主要包含4个工作：1>根
据最后一个参数idBitsToAssign将其指定的触控点的信息从原始事件 ev
中分离出来并产生一个新的MotionEvent。2>如果有必要，修改新
MotionEvent的Action。3>把事件的坐标转换到子控件的坐标系下。4>
将新的MotionEvent派发给子控件。此方法的返回值确定了子控件是否
决定接受这一事件序列。稍后会详细讨论此方法的实现 */ if
(dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) { ..... /*
当子控件决定接受这一事件时，为其创建一个TouchTarget并保存在
mFirstTouchTarget链表中，从此来自此触控点的事件都会派发给这个子
```

控件 */ newTouchTarget = addTouchTarget(child, idBitsToAssign); /* 如前文所述，此事件已经在子控件中得到了处理。因此标记如下变量为true后续的事件派发流程将不会再次发送此事件到这一子控件 */
alreadyDispatchedToNewTouchTarget = true; break; } } } /* ⑥ 在上述的遍历过程中没能找到一个合适的子控件以接受这一事件序列的情况下 View- Group会将这一事件序列强行交给最近一次接受事件序列的子控件。这看似不讲理的做法有它的实际意义。因为用户一根手指按时，其意图往往是为了配合上一根按下的手指以进行多点操作。因此，这是ViewGroup猜测用户使用习惯的一种策略 */ if (newTouchTarget == null && mFirstTouchTarget != null) { newTouchTarget = mFirstTouchTarget; while (newTouchTarget.next != null) { newTouchTarget = newTouchTarget.next; } // 将事件序列的触控点ID绑定到控件的 TouchTarget中 newTouchTarget.pointerIdBits |= idBitsToAssign; } } } // 实际派发工作的代码 }

简单来说，ViewGroup.dispatchTouchEvent () 方法前半部分的目的是以更新mFirstTouch-Target列表的方式确定一系列的派发目标。每个派发目标由TouchTarget表示，并在Touch-Target中的pointerIdBits中保管目标所感兴趣的触控点的列表。这一信息是后续进行实际派发的关键依据。ViewGroup确定派发目标的原则如下：

·仅当事件的动作为ACTION_DOWN或ACTION_POINTER_UP时才会进行派发目标的查找。因为这些动作标志着新的事件子序列的开始，ViewGroup仅需要对新的序列查找一个派发目标。

·ViewGroup会沿着绘制顺序相反的方向，即从上到下进行查找。毕竟用户往往都是希望点击在他能看得到的东西上。因此ZOrder越靠上的控件拥有越高的接受事件的优先级。

·ViewGroup会将一个控件作为派发目标候选的先决条件是事件的坐标位于其边界内部。当事件落入子控件内部并且它接受过另外一条事件序列时，则直接认定它就是此事件序列的派发目标。因为当用户将手指按在一个控件上，再把另一根手指也按在其上时，很可能是他想对此控件进行多点操作。

·ViewGroup会首先通过dispatchTransformedTouchEvent () 尝试将事件派发给候选控件，倘若控件在其事件处理函数中返回true，则可以确定它就是派发目标，否则继续测试下一个子控件。

·当遍历了所有子控件后都无法找到一个合适的派发目标时（事件落在了所有子控件之外，或者所有子控件对此事件都不感兴趣），ViewGroup会强行将接受了上一条事件序列的子控件作为派发目标。因为ViewGroup猜测用户以相邻次序按下的两根手指应该包含着能够共同完成某种任务的期望。可以看出，ViewGroup尽其所能地将事件派发给

子控件，而不是将事件留给自己处理。不过当目前没有任何一个子控件正在接受事件序列时（`mFirstTouchTarget`为null），`ViewGroup`便不得不将事件交给自己处理了。

(2) 依据TouchTarget进行触摸事件的派发

接下来是实际的派发工作。不难想到派发工作是围绕着`mFirstTouchTarget`列表完成的。`ViewGroup.dispatchTouchEvent ()`方法会遍历列表中的每一个TouchTarget，从MotionEvent中提取TouchTarget所感兴趣的触控点的信息并组成新的MotionEvent，然后将其派发给TouchTarget所代表的子控件。

参考`dispatchTouchEvent ()`实现：

```
[ViewGroup.java-->ViewGroup.dispatchTouchEvent()]
public boolean
dispatchTouchEvent(MotionEvent ev) {.....boolean handled = false;// 当
然，实际的派发工作也位于安全检查之内if
(onFilterTouchEventForSecurity(ev)) { ..... // 查找派发目标的代码 if
(mFirstTouchTarget == null) { /* ① 当mFirstTouchTarget为null时，表明之
前未能找到任何一个合适的子控件接受事件 序列，此时只能由
ViewGroup自己处理输入事件。注意将事件派发给ViewGroup自己也使
用了dispatchTransformedTouchEvent()方法，不过会将child参数设置为
null */ handled = dispatchTransformedTouchEvent(ev, canceled, null,
```

```
TouchTarget.ALL_POINTER_IDS); } else { // 遍历mFirstTouchTarget链表，为每一个TouchTarget派发事件 TouchTarget predecessor = null;  
TouchTarget target = mFirstTouchTarget; while (target != null) { final  
TouchTarget next = target.next; if (alreadyDispatchedToNewTouchTarget  
&& target == newTouchTarget) { /* ② 倘若TouchTarget是新确定的  
TouchTarget，那么在确定的过程中目标控件 已经完成了事件处理，因  
此不需要再派发 */ handled = true; } else { /* cancelChild表示因为某种原  
因需要中断目标控件继续接受事件序列，这往往由于 目标控件即将被  
移出控件树，或者ViewGroup决定截取此事件序列。此时仍然会 将事  
件发送给目标控件，但是其动作会被改成ACTION_CANCEL */ final  
boolean cancelChild = resetCancelNextUpFlag(target.child) || intercepted; //  
③ 使用dispatchTransformedTouchEvent()方法派发事件给目标控件 if  
(dispatchTransformedTouchEvent(ev, cancelChild, target.child,  
target.pointerIdBits)) { handled = true; } /* 倘若决定终止目标控件继续接  
受事件序列，则将其对应的TouchTarget从链表中删除，并回收。下次  
事件到来时将不会为其进行事件派发 */ if (cancelChild) { if (predecessor  
== null) { mFirstTouchTarget = next; } else { predecessor.next = next; }  
target.recycle(); target = next; continue; } } predecessor = target; target =  
next; } } ..... } }
```

派发过程的实现思路十分清晰。即倘若没能找到合适的派发目标，则将事件派发给ViewGroup自己。否则遍历每一个TouchTarget并将事件派

发给它。

在这里再一次看到了ViewGroup.dispatchTransformedTouchEvent () 方法。可见它是将事件派发给子控件的必由之路，因此它非常重要。参考实现如下：

```
[ViewGroup.java-->ViewGroup.dispatchTransformedTouchEvent()]
private
boolean dispatchTransformedTouchEvent(MotionEvent event, boolean
cancel, View child, int desiredPointerIdBits) {final boolean handled; /* ① 首
先处理当需要终止子控件对事件序列进行处理的情况。此时仅需要将
事件的动作替换为ACTION_CANCEL并调用子控件的
dispatchTouchEvent()即可。此时并没有进行上节所述的如坐标变换等动
作，因为ACTION_CANCEL仅仅是一个要求接受者立刻终止事件处理
并恢复到事件处理之前状态的一个记号而已，此时其所携带的其他信
息如坐标等都是没有意义的 */
final int oldAction = event.getAction();if
(cancel || oldAction == MotionEvent.ACTION_CANCEL) {
event.setAction(MotionEvent.ACTION_CANCEL); if (child == null) { /*
当child参数为null时表示事件需要派发给ViewGroup自己。注意使用的
super.dispatchTouchEvent()是前文所讨论过的View版本的实现 */
handled
= super.dispatchTouchEvent(event); } else { handled =
child.dispatchTouchEvent(event); } event.setAction(oldAction); // 事件派发
完成，直接返回
return handled;}/* 这两个局部变量是确定是否需要进行
```

事件序列分割的依据。oldPointerIdBits表示了原始事件中所有触控点的列表。而newPointerIdBits则表示了目标希望接受的触控点的列表，它是oldPoinerIdBits 的一个子集。既然desiredPointeIdBits参数已经描述了目标希望接收的触控点，为什么newPointerIdBits是必要的呢？因为desiredPointerIdBits的值有可能是TouchTarget.ALL_POINTER_IDS此时它并不能准确地表示实际需要派发的触控点列表 */final int oldPointerIdBits = event.getPointerIdBits();final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;...../* tranformedEvent是一个来自原始MotionEvent的新的MotionEvent，它只包含了目标所感兴趣的触控点，派发给目标事件对象是它而不是原始事件 */final MotionEvent transformedEvent;// ② 生成transformedEvent。比较newPointerIdBits以及oldPointerIdBits，如果二者相等，则表示目标对原始事件的所有触控点全盘接受，因此transformedEvent仅仅是原始事件的一个复制。而当二者不相等时，transformedEvent是原始事件的一个子集，此时需要使用MotionEvent.split() 方法将这一子集分离出来以构成transformedEvent */if (newPointerIdBits == oldPointerIdBits) { /* 当目标控件不存在通过setScaleX()等方法进行的变换时，为了效率会将原始事件简单地进行控件位置与滚动量变换之后发送给目标的dispatchTouchEvent()方法并返回。因为这一计算在后面的代码中仍有体现，因此为了篇幅省略了这部分代码 */ transformedEvent = MotionEvent.obtain(event); // 复制原始事件} else { transformedEvent = event.split(newPointerIdBits); // 分离原

始事件中的一个子集}// ③ 对transformedEvent进行坐标系变换，并发送给目标if (child == null) { // 与之前一样，当child参数为null时表示将事件发送给ViewGroup自己 handled =
super.dispatchTouchEvent(transformedEvent);} else { // 对transformedEvent进行坐标系变换，使之位于派发的坐标系之中 // 首先是计算ViewGroup的滚动量以及目标控件的位置 final float offsetX = mScrollX - child.mLeft; final float offsetY = mScrollY - child.mTop;
transformedEvent.offsetLocation(offsetX, offsetY); /* 然后当目标控件中存在使用setScaleX()等方法设置的矩阵变换时，将对事件坐标进行变换。此次变换完成之后，事件坐标点便位于目标控件的坐标系了 */ if (! child.hasIdentityMatrix()) {
transformedEvent.transform(child.getInverseMatrix()); } // 通过 dispatchTouchEvent()发送给目标控件 handled =
child.dispatchTouchEvent(transformedEvent);}// 销毁
transformedEvent.recycle();return handled; }

抛开其中用于处理ACTION_CANCEL的代码，不难总结出此方法的三个步骤：

- 生成transformedEvent，根据目标所感兴趣的触控点列表，transformedEvent有可能是原始事件的一个副本，或者仅包含部分触控点信息的一个子集。

·对transformedEvent进行坐标系变换，使之位于目标控件的坐标系之中。

·通过dispatchTouchEvent () 将transformedEvent发送给目标控件。

当然，将transformedEvent进行销毁也是必需的一个步骤，不过它并不算是核心步骤。

虽然仅仅包含三个步骤，但是上一节却介绍此方法完成了4项工作。不错，修改事件的Action并没有直接地体现在这一方法之中。事实上，这一工作发生在MotionEvent.split () 方法之中。在分析MotionEvent.split () 方法如何修改事件的Action之前，首先讨论一下为什么要修改事件的Action。

在讨论事件序列的概念时曾经提到过它一定是以ACTION_DOWN开始，经历一系列的

ACTION_POINTER_DOWN/ACTION_POINTER_UP/ACTION_MOVE之后以一条ACTION_UP结束。假设一个ViewGroup收到如图6-26所示的一条事件序列，而它的一个子控件仅对触控点3对应的子序列感兴趣，于是，此ViewGroup通过MotionEvent.split () 方法将触控点3的信息分离出来构成新的transformedEvent，并派发给子控件。问题在于，触控点3的子序列起始事件的动作作为ACTION_POINTER_DOWN，并且终止事件的动作作为ACTION_POINTER_UP，这对ViewGroup来说是合

理的，因为它是一条子序列。但是它在目标子控件看来就不合理了，因为这是子控件的一条完整的事件序列，而且根据事件序列的性质，要求其以ACTION_DOWN开始并以ACTION_UP结束。因此，在为子控件分离触控点3的起始与终止事件时，必须修改事件的动作作为ACTION_DOWN以及ACTION_UP，以保证子控件能够正常工作。这种情况的需求是将ACTION_POINTER_DOWN/UP修改为ACTION_DOWN/UP。如图6-29所示。

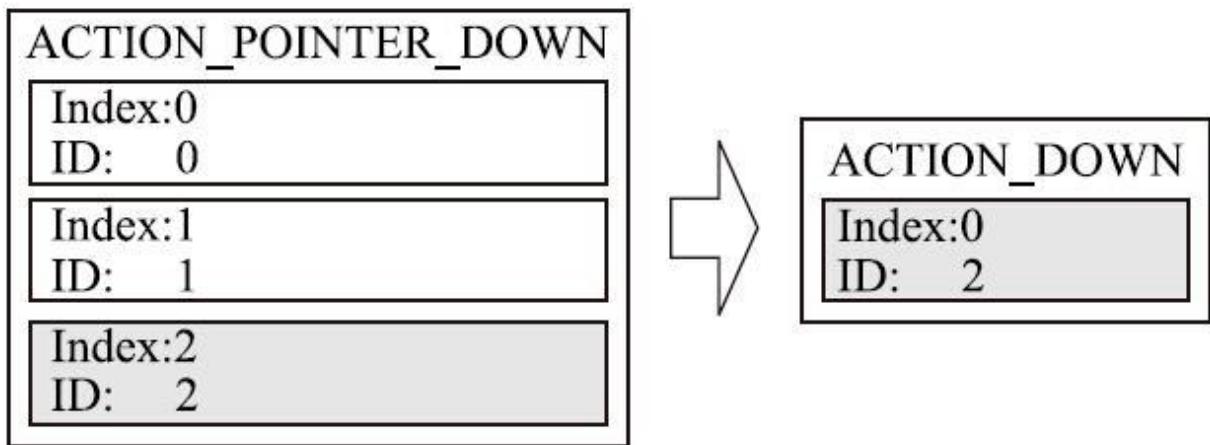


图6-29 子控件需要修改分离后事件的动作作为ACTION_DOWN

还有第二种情况，仍假设一个ViewGroup收到如图6-26所示的一个事件序列，而它的一个子控件仅对触控点2对应的子序列感兴趣。那么当触控点3的按下事件发生时，原始事件的动作是ACTION_POINTER_DOWN，很明显子控件对这一按下动作是不感兴趣的，但是它却感兴趣原始事件中所携带的触控点2的坐标等信息。因

此当 ViewGroup 为子控件分离触控点2的信息到一个 transformedEvent 时，需要将事件的动作修改为 ACTION_MOVE。而触控点3的抬起事件到来时亦然。这种情况的需求是将 ACTION_POINTER_DOWN/UP 修改为 ACTION_MOVE，如图6-30所示。

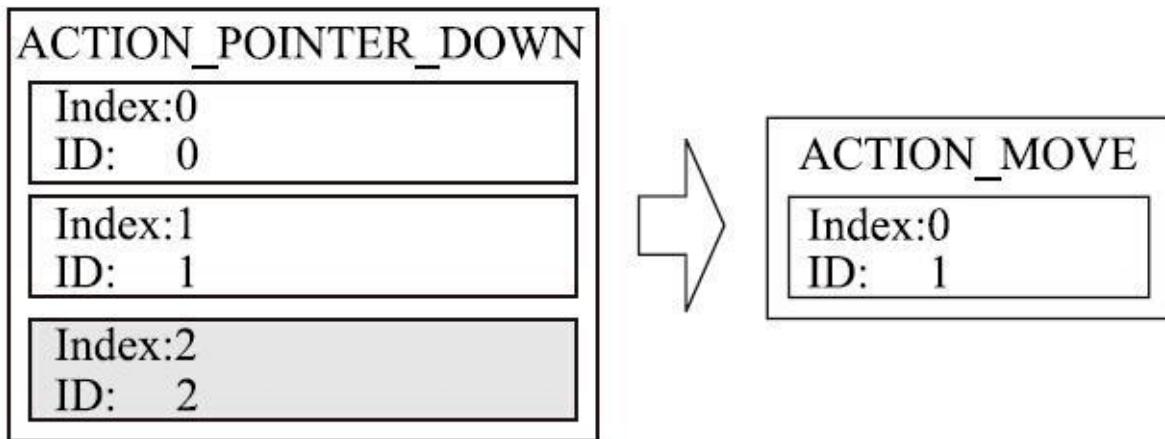


图6-30 子控件需要修改分离后事件的动作为 ACTION_MOVE

还有最后一种稍微复杂一些的情况。仍以图6-26为例，如果子控件同时对触控点2与触控点3感兴趣。那么当触控点3按下时，子控件确实需要一个 ACTION_POINTER_DOWN 事件。此时看似不需要进行事件动作的修改，其实不然。如前文所述，事件的动作作为 ACTION_POINTER_DOWN/UP 时，其高位存储了触控点的索引（即触控点信息在 mSamplePointerCoords 数组中的位置），触控点的索引也是事件动作的一部分。事实上，由于通过 MotionEvent.split () 分离出来的 MotionEvent 使用了自己的 mSample-PointerCoords 数组，所以对本例

的子控件来说，为其分离出的用于描述触控点3按下ACTION_POINTER_DOWN对应的触控点索引相对于原始事件已经发生了变化。这种情况的需求是修改ACTION_POINTER_DOWN/UP所对应的触控点索引，如图6-31所示。

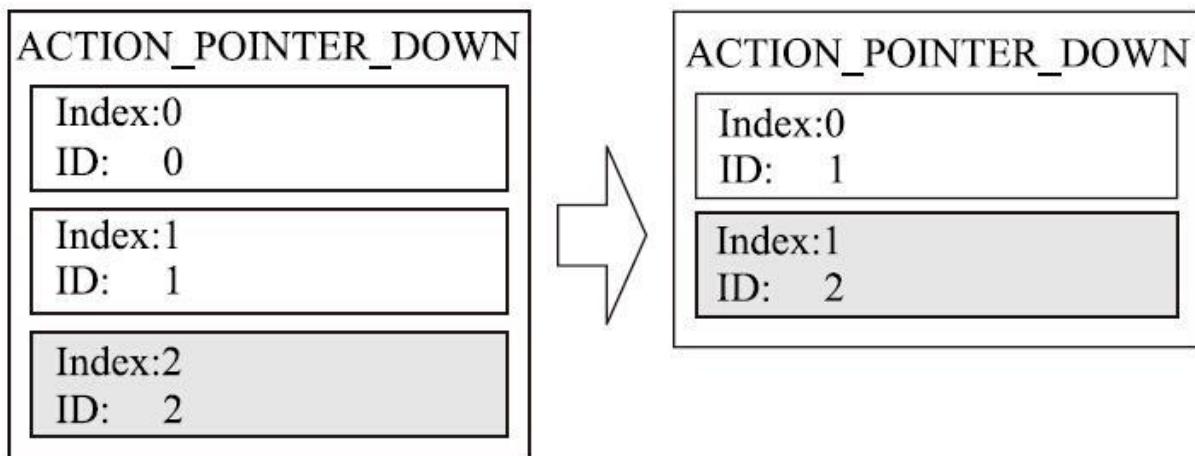


图6-31 子控件希望保持ACTION_POINTER_DOWN，但是需要修改触控点的索引

了解了修改事件动作的原因，那么分析MotionEvent.split () 如何修改事件动作就简单了。

- 首先，修改事件动作的情况仅发生在原始事件的动作作为ACTION_POINTER_DOWN或ACTION_POINTER_UP的情况下。
- 当传递给MotionEvent.split () 的触控点ID列表中仅包含一个触控点，并且它是引发ACTION_POINTER_DOWN/UP的触控点时，分离出的事

件的动作将会被设置为ACTION_DOWN/UP。对应上述第一种情况。

·当传递给MotionEvent.split () 的触控点ID列表中不包含引发ACTION_POINTER_DOWN/UP的触控点时，则表示不关心这一按下或抬起动作，分离出的事件的动作将会被设置为ACTION_MOVE。对应上述第二种情况。

·当传递给MotionEvent.split () 的触控点ID列表中包含多个触控点，并且其中之一是引发ACTION_POINTER_DOWN/UP的触控点时，分离出的事件动作将会被保持为ACTION_POINTER_DOWN/UP，但是其包含的触控点索引将会根据新事件内部的mSamplePointerCoords数组的状况重新计算。

修改事件动作的代码在这里就不详细讨论了，读者可以根据上述内容自行研究。

在dispatchTransformedTouchEvent () 方法中，当child参数为null时，表示派发对象是ViewGroup本身，此时它通过super.dispatchTouchEvent () 方法转而调用其View版本的实现，从而使ViewGroup的onTouch () 以及OnTouchListener得以对事件进行处理。而当child参数不为null时，则会调用child.dispatchTouchEvent () 。如果child是一个ViewGroup，那么自然而然，child又会按照本节所讨论的派发过程，先

确定其派发目标，然后再通过dispatchTransformedTouchEvent () 将事件派发给child的子控件，如此递归下去。

(3) 移除派发目标

ViewGroup.dispatchTouchEvent () 先确定派发目标控件，为其创建 TouchTarget并追加到mFirstTouchTarget链表中，然后依照这一链表依次将事件派发给对应的目标控件。那么什么时候会将TouchTarget从链表中移除呢？显然，当目标控件所感兴趣的最后一条事件序列结束时，或者ViewGroup收到一个ACTION_CANCEL或ACTION_UP时，就是将其从mFirstTouchTarget链表中移除的时机了。参考

ViewGroup.dispatchTouchEvent () 方法最后一部分代码：

```
[ViewGroup.java-->ViewGroup.dispatchTouchEvent()]
public boolean dispatchTouchEvent(MotionEvent ev) {
    .....
    boolean handled = false;
    if (onFilterTouchEventForSecurity(ev)) {
        ..... // 查找派发目标的代码
        ..... // 进行实际派发工作的代码
        if (canceled || actionMasked ==
            MotionEvent.ACTION_UP || actionMasked ==
            MotionEvent.ACTION_HOVER_MOVE) {
            /* 当ViewGroup收到一个 ACTION_CANCEL事件或ACTION_UP事件时，整个事件序列已经结束，因此删除所有TouchTarget */
            resetTouchState();
        } else if (split && actionMasked ==
            MotionEvent.ACTION_POINTER_UP) {
            /* 当事件是一个ACTION_POINTER_UP时，将其对应的触控点ID从对应的
               触控点ID列表中移除 */
            removeTouchPointId(actionMasked);
        }
    }
}
```

TouchTarget中 移除。 removePointersFromTouchTargets()会在TouchTarget 的最后一个触控点ID被 移除的同时， 将这个TouchTarget从 mFirstTouchTarget链表中删除并销毁 */ final int actionBarIndex = ev.getActionIndex(); final int idBitsToRemove = 1 << ev.getPointerId(actionIndex); removePointersFromTouchTargets(idBitsToRemove); } }.....return handled; }

4.触摸事件派发的总结

相对于按键事件， 触摸事件的派发过程与策略要复杂得多。其复杂的根本原因是多点触摸与事件序列拆分机制的存在。理解这一机制的来龙去脉之后就可以发现触摸事件的派发过程清晰了很多：其本质与按键事件派发一样， 经历了从根控件开始在子控件中寻找目标并发送给目标子控件， 然后目标子控件在其子控件中寻找目标再发送给子控件的子控件， 如此递归下去直到事件得到最终处理。

而多点触摸与事件序列拆分是围绕着TouchTarget这一核心完成的。 TouchTarget是绑定触控点与目标控件的纽带。每当一个事件子序列到来时， ViewGroup都会新建或选择一个现有的TouchTarget，并将子序列对应的触控点ID绑定在其上，从而生成TouchTarget感兴趣的触控点列表。随后的事件到来时ViewGroup会从其所收到的MotionEvent中为每一个TouchTarget分离出包含它所感兴趣的触控点的新MotionEvent，然

后派发给对应的子控件。TouchTarget的数量就事件序列经过此ViewGroup后拆分成的子序列的个数。

至此，关于触摸事件的派发的讨论便告一段落。

6.5.6 输入事件派发的总结

本节继续第5章的讨论，介绍从ViewRootImpl通过InputEventReceiver接收输入事件到目标控件对事件进行处理的完成过程。其中主要讨论了以下几个内容：

- 触摸模式。当系统处于触摸模式时，表示用户主要通过触摸的方式对系统进行输入操作，此时部分控件将无法获取焦点。
- 焦点。焦点是按键事件派发的唯一依据，同时它也通过DrawableState机制影响了控件的表现方式。
- 按键事件派发。按键事件从根控件开始沿着mFocused链表派发给最终拥有焦点的控件。在按键事件正式派发给目标控件之前，它会先到输入法中走一遭。不过控件仍然可以通过重写onKeyPreIme () 方法在输入法之前获得处理事件的机会。
- 触摸事件的派发。触摸事件可能包含着多个触控点的信息。

ViewRootImpl收到一个触摸事件后并在控件树中派发的过程里可能会被拆分成多个触摸事件，沿着不同的路径到达多个目标控件。其拆分

的依据是ViewGroup中TouchTarget的列表，而一个TouchTarget可以理解为触摸事件的一条派发路径，因此ViewGroup中有多少个TouchTarget，触摸事件机会被拆分成几份。

本节仅在控件系统的范畴中对事件的派发进行讨论，因此并没有讨论Activity、Dialog等因素存在时所产生的影响。Activity、Dialog作为控件系统的一个用户同样遵从本节所讨论的派发原理，只不过它们的控件树的根控件DecorView的特殊性使得输入事件的派发有了一些新的内容。接下来的6.6节将会讨论Activity与Dialog对控件系统的使用，其中会涉及它们对输入事件派发所产生的影响。

6.6 Activity与控件系统

此前的内容都是在WindowManager、ViewRootImpl与控件树三者所组成的体系之下进行讨论的。在绝大多数情况下，窗口以及控件树往往存在于一些更加高级别的概念中。这些高级别的概念即Activity以及Dialog。本节将会讨论在Activity及Dialog之下的窗口以及控件树的特性。

6.6.1 理解PhoneWindow

在正式讨论Activity与Dialog相关的话题之前，本节先介绍一个新的窗口的概念，它是com.view.Window，一个抽象类，它从更高级别的层次上描述了一个窗口的特性。

当开发者通过WindowManager、LayoutParams以及控件树创建一个窗口时，需要手动初始化LayoutParams以及自行构建控件树。虽说这种方式已经比直接使用WMS、IWindow与Surface这些更加底层的方式进步了不少，但是仍显繁琐。现今窗口中的内容往往都有不成文的规范，如在指定的位置有标题栏、动作栏、图标等，手动创建符合这些规范的控件树绝不是一件令人愉快的事情。为此Android提供了com.view.Window（后文中在不产生歧义的情况下直接称其为Window）类以在更高的级别上操作窗口。

那么Window类究竟负责做些什么呢？Window类中有三个最核心的组件：Window-Manager.LayoutParams、一棵控件树以及Window.Callback。因此它的作用也要从这三个组件分别说起。

针对 WindowManager.LayoutParams，Window类提供了一系列set方法用于设置LayoutParams属性。这项工作看似没什么用处，还不如开发者自行管理LayoutParams来得方便。不过Window类的优势在于它可以根据用例初始化LayoutParams中的属性，例如窗口令牌、窗口名称以及FLAG等。

针对其所包含的那一棵控件树，Window为使用者提供了多种多样的控件树模板。这些模板可以为窗口提供形式多样却又风格统一的展示方式以及辅助功能，例如标题栏、图标、顶部进度条、动作栏等，甚至它还为使用者提供了选项菜单的实现。使用者仅需将显示其所关心内容的控件树交给它，它就会将其嵌套在模板的合适位置。这一模板就是最终显示时的窗口外观。（为了叙述时更好理解，后文中将这一控件树模板称为外观模板。）Window类提供了接口用于模板选择、指定期望显示的内容以及修改模板的属性（如标题、图标、进度条进度等）。

Window.Callback是一个接口，Window的使用者可以实现这个接口并注册到Window中，于是每当窗口中发生变化时都可以得到通知。可以通

过这一接口得到的通知内容有输入事件、窗口属性的变更、菜单的弹出/选择等。

简单来说，Window类是一个模板，它大大简化了一个符合使用习惯的控件树的创建过程。使得使用者仅需要关注控件树中其真正感兴趣的部分，并且仅需少量的工作就可以使这部分嵌套在一个漂亮而专业的窗口外观之下，而不用关心这一窗口外观的控件树的构成。另外由于Window类是一个抽象类，因此使用不同的Window类的实现时还可以在不修改应用程序原有逻辑的情况下提供完全不同的窗口外观。

目前Android中使用的Window类的实现是PhoneWindow。Window类中提供了用于修改LayoutParams的接口等通用功能实现，而PhoneWindow类则负责具体的外观模板的实现。



注意

PhoneWindow与PhoneWindowManager之间没有任何关系。

PhoneWindowManager是WMS的一个组成部分，用于提供与窗口管理相

关的策略。而PhoneWindow是一个用于快速构建窗口外观的工具类，相较于PhoneWindow是一个更接近于控件系统的概念。

而这唯一有关系的地方是它们都是由com.android.internal.policy.impl.Policy类中的工厂方法创建的。从字面上来看，Google最初应该是希望在手机上使用PhoneWindow的实现，而在其他设备（如平板电脑）上使用其他的实现（如TabletWindow）以提供更加差异化的窗口外观，只不过这一想法没有最终实施。

本节将着重讨论PhoneWindow类如何为窗口创建窗口外观。

1.选择窗口外观与设置显示内容

只要读者开发过任何一个包含Activity的Android程序，相信一定会对Activity.requestWindowFeature () 以及Activity.setContentView () 两个方法十分熟悉。前者负责指定Activity窗口的特性，如是否拥有标题栏，是否存在一个进度条，程序图标的位置等。换言之，Activity.requestWindowFeature () 方法决定了窗口的外观模板。而后者则设置一棵控件树用于显示在Activity中。在了解了Window的作用之后应该能猜测到Activity使用了Window类，并且这两个方法的工作应该是在Window类中完成的。而事实正是如此，这两个方法的实现如下：

```
[Activity.java-->Activity.requestWindowFeature()] public final boolean  
requestWindowFeature(int featureId) { // 直接将请求转发给Window  
return  
getWindow().requestFeature(featureId); } [Activity.java--  
>Activity.setContentView()] public void setContentView(int layoutResID)  
{// 直接将请求转发给  
Window getWindow().setContentView(layoutResID);initActionBar(); }
```

Activity何时创建了Window类的实例留待6.6.2节进行讨论。本节只讨论这两个方法如何影响Window为Activity创建控件树。由于Android使用了PhoneWindow作为Window的实现，因此分析的重点在PhoneWindow对这两个方法的实现上。

首先是PhoneWindow.requestFeature () 。参考其实现：

```
[PhoneWindow.java-->PhoneWindow.requestFeature()] public boolean  
requestFeature(int featureId) /* 倘若mContentParent不为null时调用了  
requestFeature方法则会抛出一个运行时异常mContent- Parent是一个  
ViewGroup，是用户通过setContentView()设置的控件树的直接父控件。  
当它不为 null时表示外观模板已经建立，那么此时再进行  
requestFeature()操作为时已晚 */if (mContentParent != null) { throw new  
AndroidRuntimeException(".....");}/* 接下来是对feature进行相容性检  
查。因为PhoneWindow允许使用者设置多个feature，而不同的feature 之  
间可能存在互斥性，例如，当要求窗口外观不存在标题栏时，就不再
```

允许窗口带有动作条，因为动作条是标题栏的一部分 */.....// 最后调用Window类的requestFeature()实现完成feature的最终设置return super.requestFeature(featureId); }

再看Window.requestFeature () 方法：

[Window.java-->Window.requestFeature()] public boolean requestFeature(int featureId) { // 与处理触控点ID的方式类似，Window以bit的方式存储feature列表final int flag = 1 <

可见设置窗口特性的工作是十分简单的。请记住窗口的特性被保存在Window.mFeatures成员之中。requestFeature () 方法并没有立刻创建外观模板，但是mFeatures成员将会为创建外观模板提供依据。

接下来分析PhoneWindow.setContentView () 方法。

[PhoneWindow.java-->PhoneWindow.setContentView()] public void setContentView(int layoutResID) { // ① 首先是为窗口准备外观模板if (mContentParent == null) { /* 当mContentParent为null时，表明外观模板尚未创建，此时会通过installDecor()方法创建一个外观模板。创建完成之后mContentParant便会被设置模板中的一个ViewGroup并且随后它会作为使用者提供的控件树的父控件 */ installDecor(); } else { /* 倘若外观模板已经创建，则清空mContentParent的子控件，使其准备好作为新的控件树的父控件 */ mContentParent.removeAllViews(); } /* ② 将使用者给

定的layout实例化为一棵控件树，然后作为子控件保存在mContentParent之中。完成这个操作之后，PhoneWindow便完成了整棵控件树的创建 */mLayoutInflater.inflate(layoutResID, mContentParent);/*
③ Callback接口被Window用来向使用者通知其内部所发生的变化。此时通知使用者Window的控件树发生了改变。作为Window的使用者，Activity类实现了这一接口，因此开发者可以通过重写Activity的这一方法从而对这些变化做出反应 */final Callback cb = getCallback();if (cb != null && !isDestroyed()) { cb.onContentChanged(); } }

可见，Window控件树的创建是在Window的setContentView（）方法中完成的。其创建过程分为创建外观模板以及实例化使用者提供的控件树并添加到模板中两个步骤。在完成控件树的创建之后Window会通过Callback接口将这一变化通知给其使用者，例如Activity。

实例化使用者所提供的控件树并没有什么可讨论的地方。本节更关心的是Phone-Window如何创建外观模板，因此有必要讨论一下PhoneWindow.installDecor（）的实现。参考其代码：

```
[PhoneWindow.java-->PhoneWindow.installDecor()] private void  
installDecor() { // ① 首先创建控件树的根控件，并保存在mDecor成员之中  
if (mDecor == null) { // 使用generateDecor()方法创建根控件 mDecor =  
generateDecor(); // 设置根控件的焦点优先级为子控件优先  
mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCEN
```

DANTS); /* 设置mDecor为RootNamespace。在6.5.2节介绍下一个焦点控件的查找时曾经通过isRootNameSpace()方法确定一个ViewGroup是根控件，并委托它进行控件查找。所以说setIsRootNamespace() 是使根控件区别于其他控件的一个重要手段 */

mDecor.setIsRootNamespace(true);}/* ② 生成外观模板。根控件 mDecor创建完成之后，mDecor之中没有任何内容，此时它不过是个光杆司令，接下来的generateLayout()方法将会完成外观模板的创建，并作为子控件添加到mDecor之中 */if (mContentParent == null) { /* generateLayout()方法负责生成外观模板，并返回模板中用于作为使用者提供的控件树的父控件的FrameLayout对象 */ mContentParent = generateLayout(mDecor); /* ③ 从模板中获取具有特定功能的控件并对其进行初始化的属性设置。findViewById()委托mDecor实现，负责查找具有特定Id的控件。可见虽然模板的形式可能多种多样，不过其中具有相同功能的控件使用了相同的Id，以此保证无差别地进行访问 */ mTitleView = (TextView)findViewById(com.android.internal.R.id.title);} }

在此方法中出现了另一个十分重要的成员mDecor，它是整个控件树的根控件，它是一个由generateDecor () 方法创建的类型为DecorView的一个ViewGroup。DecorView是PhoneWindow的一个内部类，因此它除了担当一个ViewGroup应有的责任之外，还和Phone-Window有着密切的互动。在接下来的小节中将会介绍DecorView的特性。

根控件创建完成之后，便开始通过generateLayout () 方法进行外观模板的创建了。外观模板的创建是一个非常繁琐的过程，因为它不仅受前文所述窗口特性的影响，而且还需要考虑窗口的样式设置、Android 的版本等。generateLayout () 会首先对这些因素进行集中解析。参考这一部分的代码，因为整个解析过程非常繁杂，这里只挑选几个拥有代表性的解析为例进行介绍：

```
[PhoneWindow.java-->PhoneWindow.generateLayout()]
protected
ViewGroup generateLayout(DecorView decor) {/*
① 首先解析窗口样式表。所谓样式表其实是定义在资源系统中的一个xml文件，指定了窗口各式各样的属性。比如窗口是浮动（对话框）还是全屏（Activity），最小尺寸，是否具有标题栏，是否显示壁纸等。这些属性设置一部分影响了前文所提到的窗口特性(如标题栏)，一部分影响了窗口的LayoutParams 中的属性（如是否浮动，是否显示壁纸等），还有一部分影响了控件树的工作方式（如最小尺寸，它会影响根控件
DecorView的测量）// 获取窗口的样式表并存储在变量a中
TypedArray a = getWindowStyle();*/
/* 首先以检查样式表中是否定义窗口为浮动窗口（非全屏）为例，这个样式影响了LayoutParams中的属性 */
mIsFloating = a.getBoolean(com.android.internal.R
.styleable.Window_windowIsFloating, false);
int flagsToUpdate =
(FLAG_LAYOUT_IN_SCREEN|FLAG_LAYOUT_INSET_DECOR)...if
(mIsFloating) { // 对浮动窗口来说，其LayoutParams.width/height必须是
```

```
WRAP_CONTENT setLayout(WRAP_CONTENT, WRAP_CONTENT); /*  
并且LayoutParams.flags中的IN_SCREEN和INSET_D或OR标记必须被移  
除。因为浮动窗口在 布局时不能被状态栏导航栏等遮挡 */ setFlags(0,  
flagsToUpdate);} else { /* 对非浮动窗口来说，其  
LayoutParams.width/height将保持默认的MATCH_PARENT并且需要增加  
IN_SCREEN和INSET_D或两个标记 */  
setFlags(FLAG_LAYOUT_IN_SCREEN|FLAG_LAYOUT_INSET_DECOR,  
flagsToUpdate);}// 然后检查样式表中是否定义了无标题栏或拥有动  
作栏两个样式。这两个样式影响了窗口的特性if  
(a.getBoolean(com.android.internal.R.styleable.Window_windowNoTitle,  
false)) { /* 因为这些样式影响了窗口的特性，因此PhoneWindow会自行  
根据样式修改窗口特性。这些特性会 影响随后外观模板的创建 */  
requestFeature(FEATURE_NO_TITLE);} else if  
(a.getBoolean(com.android.internal.R  
.styleable.Window_windowActionBar, false)) {  
requestFeature(FEATURE_ACTION_BAR);..... // 其他样式的检查，同  
样会引发修改LayoutParams以及窗口特性等操作/* 检查样式中是否定义  
了最小宽度。这种样式影响了DecorView测量时的计算。因此其效果并  
不会体现在 这里。将样式中的值保存在mMinWidthMajor/Minor成员  
中，并在需要的时候使用。其中Major的 含义是在横屏情况下的最小宽  
度，而Minor则是在竖屏情况下的最小宽度。 在介绍DecorView时将会
```

体现这些成员的用处

```
*/a.getValue(com.android.internal.R.styleable.Window_windowMinWidthMajor,  
mMinWidthMajor);a.getValue(com.android.internal.R.styleable.Window_windowMinWidthMinor, mMinWidthMinor);.....// ② 接下来是影响外观模板  
的另外一种因素，即Android的版本final Context context =  
getContext();final int targetSdk =  
context.getApplicationInfo().targetSdkVersion;final boolean  
targetPreHoneycomb = targetSdk <  
android.os.Build.VERSION_CODES.HONEYCOMB;.....if  
(targetPreHoneycomb || (targetPreIcs && targetHcNeedsOptions &&  
noActionBar)) { /* 在Honeycomb之前的版本中，选项菜单的呼出动作由  
菜单键完成，因此在需要选项菜单时需要 导航栏提供虚拟的菜单键。  
将NEEDS_MENU_KEY标记放入LayoutParams中，当此窗口处于焦点  
状态时，WMS会向SystemUI请求显示虚拟菜单键（这部分内容会在第  
7章讨论） */  
addFlags(WindowManager.LayoutParams.FLAG_NEEDS_MENU_KEY);}  
else { /* 而在Honeycomb之后的版本中，选项菜单由动作栏中的菜单按  
钮完成，因此需要将标记移除 */  
clearFlags(WindowManager.LayoutParams.FLAG_NEEDS_MENU_KEY);}  
}// 创建外观模板..... }
```

可见generateLayout () 方法在创建外观模板之前会首先解析样式表以及Android的版本，再根据这些因素修改LayoutParams中的设置，申请或删除窗口特性，或者保存一些信息以备后用。这部分代码体现了使用Window创建控件树以及管理LayoutParams的优越性。使用者仅需声明其所需的样式，而具体的工作都由Window完成。事实上，窗口所需的样式往往已经由Android事先定义好，因此在默认情况下使用者甚至都不用关心样式的存在。

接下来是创建窗口外观的工作：

```
[PhoneWindow.java-->PhoneWindow.generateLayout()]
protected
ViewGroup generateLayout(DecorView decor) { ..... // 解析样式表、
Android版本的代码/* ① 首先选择合适的外观模板。所有的窗口外观模
板已经实现被定义在系统资源之中。generateLayout()的工作就是根据
窗口特性选择一个合适的外观模板的资源id。layoutResource变量保存
了选择的结果 */
int layoutResource; int features = getLocalFeatures(); if
((features & ((1 << FEATURE_LEFT_ICON) | (1 <<
FEATURE_RIGHT_ICON))) != 0) { // 如果窗口特性是包含标题栏以及
程序图标 if (mIsFloating) { // 对浮动窗口来说，其窗口外观被保存在
dialogTitleIconsDecorLayout 样式中 TypedValue res = new TypedValue();
getContext().getTheme().resolveAttribute(
com.android.internal.R.attr.dialogTitleIconsDecorLayout , res, true);
```

```
layoutResource = res.resourceId; } else { // 对全屏窗口来说，选择
screen_title_icons布局所定义的控件树 layoutResource =
com.android.internal.R.layout.screen_title_icons; } .....} else if ((features &
((1 << FEATURE_PROGRESS) | (1 <<
FEATURE_INDETERMINATE_PROGRESS)) != 0 && (features & (1 <<
FEATURE_ACTION_BAR)) == 0) { // 如果窗口特性期望有一个进度
条，则选择screen_progress布局所定义的控件树 layoutResource =
com.android.internal.R.layout.screen_progress;} else if (.....) { // 针对其他
窗口特性进行选择 .....}// ② 将所选择的布局资源实例化为一棵控件树
并保存在变量in之中。这便是最终的外观模板View in =
mLayoutInflater.inflate(layoutResource, null); // 将外观模板作为子控件添
加到DecorView中decor.addView(in, new
ViewGroup.LayoutParams(MATCH_PARENT, MATCH_PARENT));/* ③
从外观模板控件树中获取作为使用者提供的控件树的父控件的
ContentParent。 ContentParent的Id 是ID_ANDROID_CONTENT。无论哪
种模板，必须存在一个拥有此Id的ViewGroup。否则generateLayout() 将
会抛出异常 */ ViewGroup contentParent =
(ViewGroup)findViewById(ID_ANDROID_CONTENT);.....return
contentParent; }
```

简单来说，generateLayout () 方法花费了较大的精力根据各种窗口特
性选择一个系统定义好的外观模板的布局资源，然后将其实例化后作

为子控件添加到根控件DecorView中。布局资源的来源有直接定义的（如screen_title_icon等），还有的是在样式表中定义的（如dialogTitleIconsDecorLayout等）。

回过头来看窗口控件树构建的整个过程，PhoneWindow.setContentView()首先调用installDecor()方法完成外观模板的创建，然后将使用者提供的控件树嵌入模板的mContentParent中。而installDecor()方法首先创建一个类型为DecorView的ViewGroup作为根控件，然后使用generateLayout()方法通过解析样式、Android版本、窗口特性等创建合适的模板控件树。因此在完成setContentView()的调用之后，PhoneWindow中便包含一个以DecorView为根控件，包含使用者期望显示的内容，外加一系列特性（标题栏、进度条、动作栏等）作为窗口外观的一棵控件树，如图6-32所示。与此同时，窗口的LayoutParams也得到了精心设置。使用者还可以通过Window提供的接口修改每一个特性的取值（标题、进度、图标）。

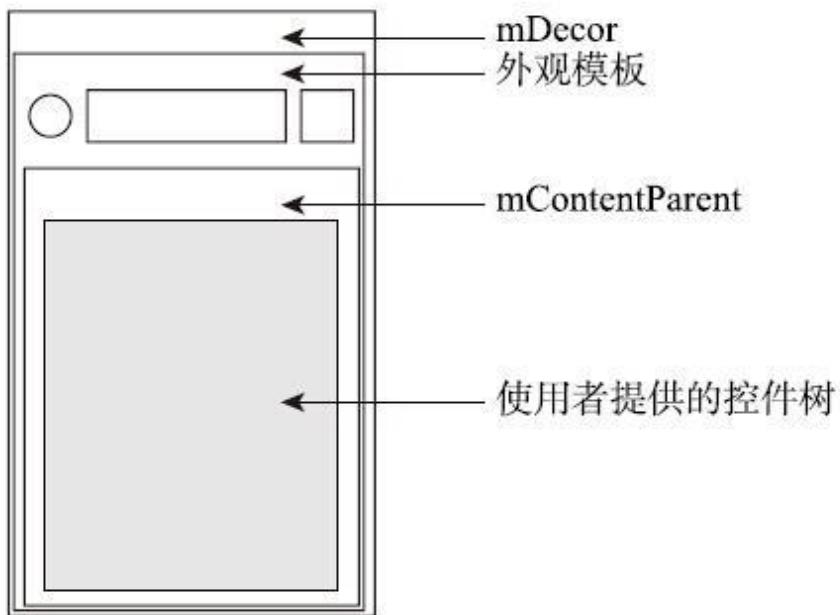


图6-32 由PhoneWindow生成的控件树

为了使窗口得以显示，使用者接下来所要做的事情就是从 PhoneWindow中获取这一棵控件树的根控件DecorView以及 LayoutParams，并通过WindowManager的接口将其添加到WMS中。

2. DecorView的特点

在结束关于PhoneWindow的讨论之前，有必要讨论一下DecorView——这一Phone-Window独有的ViewGroup。在使用Activity或者Dialog的过程中，一定对它们的成员方法onAttached/DetachedFromWindow ()、 dispatch/onTouchEvent () 等十分熟悉。Activity或Dialog并不是控件系统中的一个控件，为什么它也会拥有这些控件所特有的回调呢？原来秘密就隐藏在这个DecorView之中。

作为PhoneWindow的一个内部类，DecorView与PhoneWindow的关系十分紧密。它利用自己根控件的身份为PhoneWindow“偷取”了很多关于控件系统内部的信息，其中就包括上述的控件的生命周期信息以及输入事件。

以触摸事件为例，看一下DecorView.dispatchTouchEvent () 的实现：

```
[PhoneWindow.java-->DecorView.dispatchTouchEvent()]
public boolean
dispatchTouchEvent(MotionEvent ev) { // Callback即Window.Callback的实
例
final Callback cb = getCallback(); /* 当Callback不为null，并且
mFeatureId小于0时（这表示DecorView是一个顶级窗口的控件树的根
控件），DecorView直接将触摸事件发送给Callback */
return cb != null &&
!isDestroyed() && mFeatureId < 0 ? cb.dispatchTouchEvent(ev) :
super.dispatchTouchEvent(ev); }
```

可见DecorView作为整个控件树的根，它并没有像其他ViewGroup那样将事件派发给子控件，而是将事件发送给Window.Callback。作为Window.Callback实现者的Activity或Dialog自然就有能力接收输入事件了，而且它们还能够先于控件树中的其他控件获得处理事件的机会。其他的回调诸如onDetachedFromWindow () 等也是同理，都是DecorView这个控件系统的“叛徒”将这些信息先交给了Callback所致。

从代码来看，DecorView将事件交给Callback处理之后就结束了。那么控件树的其他控件岂不是没有机会再处理这个事件了？这里以Activity为例分析一下它的dispatchTouchEvent () 的实现：

```
[Activity.java-->Activity.dispatchTouchEvent()] public boolean  
dispatchTouchEvent(MotionEvent ev) {.....// 首先调用Window的  
superDispatchTouchEvent()if (getWindow().superDispatchTouchEvent(ev))  
{ // 如果superDispatchTouchEvent()消费了事件，则直接返回 return  
true;}// 倘若superDispatchTouchEvent()没有消费事件，则由  
Activity.onTouchEvent()处理事件return onTouchEvent(ev); }
```

那么Window.superDispatchTouchEvent () 做了些什么呢？

```
[PhoneWindow.java-->PhoneWindow.superDispatchTouchEvent()] public  
boolean superDispatchTouchEvent(MotionEvent event) {// 事件又转入了  
DecorViewreturn mDecor.superDispatchTouchEvent(event); }
```

再看DecorView.superDispatchTouchEvent () :

```
[PhoneWindow.java-->DecorView.superDispatchTouchEvent()] public  
boolean superDispatchTouchEvent(MotionEvent event) {// 调用  
ViewGroup.dispatchTouchEvent()按照常规流程对事件进行派发return  
super.dispatchTouchEvent(event); }
```

原来如此，当DecorView接收到事件之后，会首先将其交给Callback（即本例中的Activity）的dispatchTouchEvent（）过目。Callback的dispatchTouchEvent（）会将事件交还给DecorView进行常规的事件派发，倘若事件在派发过程中没有被消费掉，Callback再自行消费这一事件。这个过程对于其他事件一样适用。

因此可以将DecorView理解为Callback的实现者在控件树中的替身。DecorView所接收的关于控件树的各种事件，Callback的实现者一样可以接收到，就好像它也是控件树中的一员。可见Callback的实现者如Activity或Dialog中的dispatchXXXX（）会先于控件树中的任何一个控件进行事件处理，而它们的onXXXX（）则仅当事件没有被任何一个控件消费时才有机会进行事件处理。

另外，在分析PhoneWindow.generateLayout（）方法时提到了一个样式用于限制窗口的最小尺寸。DecorView负责对这一样式进行实现。它重写了onMeasure（）方法，并在其内部依据样式的设置限制了测量结果。在6.3.2节介绍ViewRootImpl的预测量时介绍了ViewRootImpl对测量结果的限制。当二者发生冲突时，DecorView的限制拥有更高的决定权。

尽管DecorView拥有其特殊性，但其特殊性在控件系统中并不会体现出来。对控件系统来说（包括ViewRootImpl以及控件树），它与一个普通的控件别无二致。

3. 关于PhoneWindow的总结

关于PhoneWindow的讨论就此告一段落。相对于之前所讨论的窗口，PhoneWindow更接近于一个控件系统的概念，因为它的主要工作在于对控件树的操作，而不是用来管理一个窗口。而将它称为一个Window的原因是其中的两个组件：LayoutParams及控件树正是通过 WindowManager添加窗口的两个充要条件。因此从WindowManager的角度来看，PhoneWindow的确是一个不折不扣的窗口。

经过本节的讨论，创建窗口的方式再次得到了简化，从 WindowManager、LayoutParams再加一棵控件树的创建方式变为 WindowManager加上一个PhoneWindow。这次简化的内容是 LayoutParams的设置以及控件树的构建过程。而Activity或Dialog正是以这种方式创建窗口的。

6.6.2 Activity窗口的创建与显示

本节将从Activity启动的生命周期中讨论Activity窗口的创建与显示的过程，并在这个过程中寻找PhoneWindow的创建、控件树的创建以及窗口的显示与Activity的生命周期之间的关系。

当第一次启动Activity时，第一件事情就是创建一个Activity的对象（绝大多数情况下这一对象是开发者所实现的Activity的子类）。而Activity对象创建完成后的第一件事情就是对其进行初始化，以便将窗口令牌

等重要的信息移交给新生的Activity。这一初始化的动作发生在Activity.attach () 方法中。这一方法的参数非常多，这里省略了与本节内容无关的参数。

```
[Activity.java-->Activity.attach()]
final void attach(..., IBinder token,
...CharSequence title, ...) {.....// 首先Activity通过PolicyManager创建一个
新的PhoneWindow对象，并保存在mWindow中mWindow =
PolicyManager.makeNewWindow(this); // 然后将Activity作为
Window.Callback实例设置给
PhoneWindowmWindow.setCallback(this);.....// token是
IAplicationToken，即添加一个Activity窗口所需的令牌mToken =
token;.....// 将一个WindowManager实例保存到PhoneWindow中，窗口令牌也一并做了保存mWindow.set WindowManager(
(WindowManager)context.getSystemService(Context.WINDOW_SERVICE
), mToken, mComponent.flattenToString(), (info.flags &
ActivityInfo.FLAG_HARDWARE_ACCELERATED) != 0);.....// 最后从
PhoneWindow中取出WindowManager并保存到Activity中
mWindowManager = mWindow.getWindowManager(); }
```

可见在创建Activity之后，它立刻便拥有创建窗口所需的所有条件：PhoneWindow、WindowManager，以及一个来自AMS的窗口令牌。

接下来就是生成控件树。开发者往往在Activity.onCreate () 中，通过setContentView () 设置一个布局资源。如6.6.1节所述，setContentView () 将会使得PhoneWindow完成Layout-Param的设置以及控件树的创建。因此在经历onCreate () 之后，Activity已经随时准备好显示其窗口。

Activity窗口的显示发生在onResume () 之后，参考ActivityThread.handleResumeActivity () 方法的相关实现如下：

```
[ActivityThread.java-->ActivityThread.handleResumeActivity()]
final void handleResumeActivity(IBinder token, boolean clearHide, boolean
isForward, boolean reallyResume) {.....// ① 在performResumeActivity()
中，Activity.onResume()方法将会被调用ActivityClientRecord r =
performResumeActivity(token, clearHide);if (r != null) { /* ② 创建窗口 当
ActivityClientRecord的window成员为null时，表示此Activity尚未创建窗
口。此时需要 将PhoneWindow中的控件树交给WindowManager完成窗
口的创建。这种情况对应于Activity初 次创建的情况（即onCreate()被调
用的情况）。如果Activity因为某种原因被暂停，如新的Activity 覆盖其
上或者用户按下了HOME键，虽说Activity不再处于Resume状态，但是
其窗口并没有从WMS中 移除，只不过它不可见而已 */ if (r.window ==
null && !a.mFinished && willBeVisible) { // 获取PhoneWindow的实例
r.window = r.activity.getWindow(); // 获取DecorView View decor =
```

```
r.window.getDecorView(); // 注意Activity的窗口在初创时是不可见的。  
因为尚不确定是否真的要显示窗口给用户  
decor.setVisibility(View.INVISIBLE); // 获取WindowManager  
ViewManager wm = a.getWindowManager();  
 WindowManager.LayoutParams l = r.window.getAttributes(); // 设置窗口类  
型为BASE_APPLICATION。这表示窗口属于一个Activity l.type =  
WindowManager.LayoutParams.TYPE_BASE_APPLICATION; if  
(a.mVisibleFromClient) { a.mWindowAdded = true; // 将DecorView添加到  
WMS完成窗口的创建 wm.addView(decor, l); } } // ③ 使Activity可见 if  
(!r.activity.mFinished && willBeVisible && r.activity.mDecor != null &&  
!r.hideForNow) { ..... if (r.activity.mVisibleFromClient) { // 最后通过  
Activity.makeVisible()使得Activity可见 r.activity.makeVisible(); } } } }
```

可见，当Activity.onResume () 被调用时，Activity的窗口其实尚未显
示，甚至尚未创建。也就是说Activity可见发生在onResume () 之后。

其实除非Activity被销毁（即onDestroy () 被调用），其所属的窗口都
会存在于WMS之中，这期间的onStart () /onStop () 所导致的可见性
的变化都是通过修改DecorView的可见性实现窗口的隐藏与显示的。另
外，Activity提供了Activity.setVisible () 方法用于让开发者手动地设置
Activity的可见性，此方法一样是通过更改DecorView的可见性达到显示
或隐藏一个Activity的目的。

至于Activity窗口的销毁，则发生在ActivityThread.handleDestroyActivity()之中，读者可以自行研究。

另外，Dialog一样使用了WindowManager加PhoneWindow的方式创建及显示其窗口。在理解了WindowManager以及PhoneWindow的实现原理之后可以很容易地对这一过程进行分析。本书便不再赘述。

6.7 本章小结

本章着重讨论了控件系统中的几个重要话题，包括ViewRootImpl的工作原理、控件的测量布局与绘制、输入事件的派发等。

另外，本章还介绍了另外两种创建窗口的方式：使用 WindowManager、LayoutParams加控件树进行手动窗口创建，以及使用 WindowManager加PhoneWindow通过外观模板进行窗口创建。再加上第4章介绍的使用WMS加Surface的创建方式，目前共介绍了三种方式进行窗口的创建方式。这三种方式在使用的过程中存在着使用难度上的差异，但是并不是说使用难度大的创建方式没有用，关键是根据窗口需要完成的任务的不同而选择合适的创建方式。举例来说，6.6节所介绍的Activity使用了PhoneWindow，因为Activity作为应用程序主要的显示组件，提供风格统一的UI更加重要。而运行于SystemUI的状态栏及导航栏需要与用户频繁进行交互，但是却不需要向应用程序的界面那样形势死板，因此使用灵活的WindowManager、LayoutParams加控件树的方式更加合适。至于WMS加Surface这种最底层的窗口创建方法，也有它存在的意义。Android壁纸其实是一个窗口，它不需要复杂的用户交互，却需要提供更加高效的运行效率与更低资源占用，以及更自由的绘图方式，而WMS加Surface的方式恰恰满足了其需求。

至此本书通过3章将WindowManagerService、输入系统以及控件系统这三个负责Android显示与交互的主要部分讨论完毕。接下来的内容就是讨论建立在这三个系统之上的两种特点应用：SystemUI以及壁纸了。读者除了要关注两个模块自身的实现原理之外，还应体会它们是如何完成其窗口的管理以及如何处理用户的交互。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第7章 深入理解SystemUI

本章主要内容：

- 探讨状态栏与导航栏的启动过程
- 介绍状态栏中的通知信息、系统状态图标等信息的管理与显示原理
- 介绍导航栏中的虚拟按键、SearchPanel的工作原理
- 介绍SystemUIVisibility

本章涉及的源代码文件名及位置：

· SystemServer.java

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

· SystemUIService.java

frameworks/base/packages/SystemUI/src/com/android/systemui/SystemUIService.java

· PhoneWindowManager.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java

· PhoneStatusBar.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/PhoneStatusBar.java

· BaseStatusBar.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/BaseStatusBar.java

· StatusBarManager.java

frameworks/base/core/java/android/app/StatusBarManager.java

· StatusBarManagerService.java

frameworks/base/services/java/com/android/server/StatusBarManagerService.java

· NotificationManager.java

frameworks/base/core/java/android/app/NotificationManager.java

· NotificationManagerService.java

frameworks/base/services/java/com/android/server/NotificationManagerService.java

· KeyButtonView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/policy/Key-ButtonView.java

· NavigationBarView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/phone/NavigationBarView.java

· DelegateViewHelper.java

frameworks/base/packages/SystemUI/src/com/android/systemui/statusbar/Delegate-ViewHelper.java

· SearchPanelView.java

frameworks/base/packages/SystemUI/src/com/android/systemui/SearchPanelView.java

· PhoneWindow.java

frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindow.java

·InputMethodService.java

frameworks/base/core/java/android/inputmethodservice/InputMethodService.java

·View.java

frameworks/base/core/java/android/view/View.java

·ViewRootImpl.java

frameworks/base/core/java/android/view/ViewRootImpl.java

· WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

7.1 初识SystemUI

顾名思义，SystemUI是为用户提供系统级别的信息显示与交互的一套UI组件，因此它所实现的功能包罗万象。屏幕顶端的状态栏、底部的导航栏、图片壁纸以及Recent-Panel（近期使用的APP列表）都属于SystemUI的范畴。SystemUI中还有一个名为TakeScreenshotService的服务，用于在用户按下音量下键与电源键时进行截屏操作。在第5章曾介绍了PhoneWindowManager监听这一组合键的机制，当它捕捉到这一组合键时便会向TakeScreenShotService发送请求从而完成截屏操作。

SystemUI还提供了PowerUI和RingtonePlayer两个服务。前者负责监控系统的剩余电量并在必要时为用户显示低电警告，后者则依托AudioService向其他应用程序提供播放铃声的功能。SystemUI的博大不止如此，读者可以通过查看其AndroidManifest.xml来了解它所实现的其他功能。本章将着重介绍其中最重要的两个功能的实现：状态栏和导航栏。

7.1.1 SystemUIService的启动

尽管SystemUI的表现形式与普通的Android应用程序大相径庭，但它却是以一个APK的形式存在于系统之中，即它与普通的Android应用程序并没有本质上的区别。无非是通过Android 4大组件中的Activity、

Service、BroadcastReceiver接受外界的请求并执行相关的操作，只不过它们所接受的请求主要来自各个系统服务而已。

SystemUI包罗万象，并且大部分功能之间相互独立，比如RecentPanel、TakeScreen-shotService等均是按需启动，并在完成其既定任务后退出，这与普通的Activity以及Service别无二致。比较特殊的是状态栏、导航栏等组件的启动方式。它们运行于一个称为SystemUIService的Service中。因此讨论状态栏与导航栏的启动过程其实也就是System-UIService的启动过程。

1.SystemUIService的启动时机

那么SystemUIService在何时由谁启动呢？作为一个系统级别的UI组件，自然要在系统的启动过程中寻找答案。

在负责启动各种系统服务的ServerThread中，当核心系统服务启动完成后ServerThread会通过调用ActivityManagerService.systemReady () 方法通知AMS系统已经就绪。这个systemReady () 拥有一个名为goingCallback的Runnable实例作为参数。顾名思义，当AMS完成对systemReady () 的处理后将会回调这一Runnable的run () 方法。而在这一run () 方法中可以找到SystemUI的身影：

[SystemServer.java-->ServerThread]

```
ActivityManagerService.self().systemReady(new Runnable() {public void
```

```
run() { // 调用startSystemUi() if (!headless) startSystemUi(contextF); .....}  
}
```

进一步，在startSystemUI () 方法中：

```
[SystemServer.java-->ServerThread.startSystemUi()] static final void  
startSystemUi(Context context) {Intent intent = new Intent();// 设置  
SystemUIService作为启动目标intent.setComponent(new  
ComponentName("com.android.systemui",  
"com.android.systemui.SystemUIService")); // 启动  
SystemUIServicecontext.startServiceAsUser(intent, UserHandle.OWNER);  
}
```

可见，当核心的系统服务启动完毕后，ServerThread通过
Context.startServiceAsUser () 方法完成了SystemUIService的启动。

2.SystemUIService的创建

参考SystemUIService的onCreate () 的实现：

```
[SystemUIService.java-->SystemUIService.onCreate()] /* ① SERVICES数  
组定义了运行于SystemUIService之中的子服务列表。当  
SystemUIService服务启动时将会依次启动列表中所存储的子服务 */final  
Object[] SERVICES = new Object[] { 0, // 0号元素存储的其实是一个字符
```

串资源号，这个字符串资源存储实现状态栏/导航栏的类名
com.android.systemui.power.PowerUI.class,
com.android.systemui.media.RingtonePlayer.class,};public void onCreate()
{.....IWindowManager wm =
WindowManagerGlobal.getWindowManagerService();try { /* ② 根据
IWindowManager.hasSystemNavBar()的返回值选择一个合适的状态栏与
导航栏的实现 */ SERVICES[0] = wm.hasSystemNavBar() ?
R.string.config_systemBarComponent :
R.string.config_statusBarComponent;} catch (RemoteException e)
{.....}final int N = SERVICES.length;// mServices数组中存储了子服务的
实例mServices = new SystemUI[N];for (int i=0; i

除了onCreate () 方法之外，SystemUIService没有其他有意义的代码
了。显而易见，SystemUIService是一个容器。在其启动时，将会逐个
实例化定义在SERVICES列表中的继承自SystemUI抽象类的子服务。
在调用子服务的start () 方法之后，SystemUIService便不再做任何其他
的事情，任由各个子服务自行运行。而状态栏导航栏则是这些子服务
中的一个。

值得注意的是，onCreate () 方法根据
IWindowManager.hasSystemNavBar () 方法的返回值为状态栏/导航栏
选择了不同的实现。进行这一选择的原因是为了能够在大尺寸的设备

中更有效地利用屏幕空间。在小屏幕设备如手机中，由于屏幕宽度有限，Android采取了状态栏与导航栏分离的布局方案，也就是说导航栏与状态栏占用了更多的垂直空间，使得导航栏的虚拟按键尺寸足够大以及状态栏的信息量足够多。而在大屏幕设备如平板电脑中，由于屏幕宽度比较大，足以在一个屏幕宽度中同时显示足够大的虚拟按键以及足够多的状态栏信息量，此时可以选择将状态栏与导航栏功能集成在一起成为系统栏作为大屏幕下的布局方案，以节省对垂直空间的占用。

hasSystemNavBar () 的返回值取决于 PhoneWindowManager.mHasSystemNavBar 成员的取值。因此在 PhoneWindowManager.setInitialDisplaySize () 方法中可以得知 Android 在两种布局方案中进行选择的策略。

```
[PhoneWindowManager.java-->PhoneWindowManager.setInitialDisplaySize()]\ public void setInitialDisplaySize(Display display, int width , int height, int density) {.....// ① 计算屏幕短边的DP宽度int shortSizeDp = shortSize * DisplayMetrics.DENSITY_DEFAULT / density;// ② 屏幕宽度在720dp以内时，使用分离的布局方案if (shortSizeDp < 600) { mHasSystemNavBar = false; mNavigationBarCanMove = true;} else if (shortSizeDp < 720) { mHasSystemNavBar = false; mNavigationBarCanMove = false;}..... }
```

在SystemUI中，分离布局方案的实现者是PhoneStatusBar，而集成布局方案的实现者则是TabletStatusBar。二者的本质功能是一致的，即提供虚拟按键、显示通知信息等，区别仅在于布局的不同、以及由此所衍生出的定制行为而已。因此不难想到，它们是从同一个父类中继承出来的。这一父类的名字是BaseStatusBar。本章将主要介绍PhoneStatusBar的实现，读者可以类比地对TabletStatusBar进行研究。

7.1.2 状态栏与导航栏的创建

如7.1.1节所述，状态栏与导航栏的启动由其PhoneStatusBar.start () 完成。参考其实现：

```
[PhoneStatusBar.java-->PhoneStatusBar.start()] public void start() {.....// ①  
调用父类BaseStatusBar的方法进行初始化super.start();// 创建导航  
栏的窗口addNavigationBar();// ② 创建PhoneStatusBarPolicy。  
  
PhoneStatusBarPolicy定义了系统通知图标的设置策略mIconPolicy = new  
PhoneStatusBarPolicy(mContext); }
```

参考BaseStatusBar.start () 的实现，这段代码比较长，并且涉及本章随后会详细介绍的内容。因此，倘若读者阅读起来比较吃力可以仅关注那三个关键步骤。在完成本章的学习之后再回过头来阅读这部分代码便会发现十分简单了。

[BaseStatusBar-->BaseStatusBar.start()] public void start() {/* 由于状态栏的窗口不属于任何一个Activity，所以需要使用第6章所介绍的 WindowManager进行窗口 的创建 */mWindowManager = (WindowManager)mContext.getSystemService(Context.WINDOW_SERVICE);/* 在第4章介绍窗口的布局时曾经提到状态栏的存在对窗口布局有着重要的影响。因此状态栏中所发生的变化有必要通知给WMS */mWindowManagerService = WindowManagerGlobal.getWindowManagerService();.....*/
mProvisioningObserver是一个ContentObserver。它负责监听 Settings.Global.DEVICE_PROVISIONED设置的变化。这一设置表示此设备是否已经归属 于某一个用户。比如当用户打开一个新购买的设备时，初始化设置向导将会引导用户阅读使用条款、设置账户 等一系列的初始化操作。在初始化设置向导完成之前，
Settings.Global.DEVICE_PROVISIONED的值为false， 表示这台设备并未归属于某一个用户。当设备并未归属于某一个用户时，状态栏会禁用一些功能以避免泄露信息。mProvisioningObserver用来 监听设备归属状态的变化，以禁用或启用某些功能
*/mProvisioningObserver.onChange(false); // set
upmContext.getContentResolver().registerContentObserver(
Settings.Global.getUriFor(Settings.Global.DEVICE_PROVISIONED), true,
mProvisioningObserver);/* ① 获取IStatusBarService的实例。

IStatusBarService是一个系统服务，由ServerThread启动并常驻system_server进程中。IStatusBarService为那些对状态栏感兴趣的其他系统服务定义了一系列API，然而对SystemUI而言，它更像是一个客户端。因为IStatusBarService会将操作状态栏的请求发送给SystemUI，并由后者完成请求 *mBarService = IStatusBarService.Stub.asInterface(ServiceManager.getService(Context.STATUS_BAR_SERVICE));/* 随后BaseStatusBar将自己注册到IStatusBarService之中。以此声明本实例才是状态栏的真正实现者，IStatusBarService会将其所接受的请求转发给本实例。“天有不测风云”，SystemUI难免会因为某些原因而意外终止。而状态栏中所显示的信息并不属于状态栏自己，而是属于其他的的应用程序或其他的系统服务。因此当SystemUI重新启动时，便需要恢复其终止前所显示的信息以避免信息的丢失。为此，IStatusBarService中保存了所有需要状态栏进行显示的信息的副本，并在新的状态栏实例启动后，这些副本将会伴随着注册的过程传递给状态栏并进行显示，从而避免信息丢失。从代码分析的角度来看，这一从IstatusBarService中取回信息副本的过程正好完整地体现了状态栏所能显示的信息的类型*/* iconList是向IStatusBarService进行注册的参数之一。它保存了用于显示在状态栏的系统状态区中的状态图标列表。在完成注册之后，IStatusBarService将会在其中填充两个数组，一个字符串数组用于表示状态的名称，一个StatusBarIcon类型的数组用于存储需要显示的图标资源。关于系统状态区的工作原理将在7.2.3节介绍

```
*/StatusBarIconList iconList = new StatusBarIconList();/* notificationKeys  
和StatusBarNotification则存储了需要显示在状态栏的通知区中的通知信  
息。前者 存储了一个用Binder表示的通知发送者的ID列表。而  
notifications则存储了通知列表。二者通过索引 号一一对应。关于通知  
的工作原理将在7.2.2节介绍 */ArrayList notificationKeys = new ArrayList  
();ArrayList notifications = new ArrayList ();/* mCommandQueue是  
CommandQueue类的一个实例。CommandQueue继承自IStatusBar.Stub。  
因此 它是IStatusBar的Bn端。在完成注册后，这一Binder对象的Bp端将  
会保存在IStatusBarService之中。因此它是IStatusBarService与  
BaseStatusBar进行通信的桥梁。 */ mCommandQueue = new  
CommandQueue(this, iconList);/* switches则存储了一些杂项：禁用功能  
列表，SystemUIVisibility，是否在导航栏中显示虚拟的菜单 键，输入法  
窗口是否可见，输入法窗口是否消费BACK键，是否接入了实体键盘，  
实体键盘是否被启用。在后文中 将会介绍它们的具体影响 */int[]  
switches = new int[7];ArrayList binders = new ArrayList ();try { // ② 向  
IStatusBarService进行注册，并获取所有保存在IStatusBarService中的信  
息副本 mBarService.registerStatusBar(mCommandQueue, iconList,  
notificationKeys, notifications, switches, binders);} catch (RemoteException  
ex) {.....}// ③ 创建状态栏与导航栏的窗口。由于创建状态栏与导航栏  
的窗口涉及控件树的创建，因此它由子类Phone- StatusBar或  
TabletStatusBar实现，以根据不同的布局方案选择创建不同的窗口与控
```

```
件树 */createAndAddWindows();/* 应用来自IStatusBarService中所获取的信息mCommandQueue已经注册到IStatusBarService中， 状态栏与导航栏的窗口与控件树也都创建完毕因此接下来的任务就是应用从IStatusBarService中所获 取的信息 */disable(switches[0]); // 禁用某些功能setSystemUiVisibility(switches[1], 0xffffffff); // 设置SystemUIVisibilitytopAppWindowChanged(switches[2] != 0); // 设置菜单键的可见性// 根据输入法窗口的可见性调整导航栏的样式setImeWindowStatus(binders.get(0), switches[3], switches[4]);// 设置硬件键盘信息setHardKeyboardStatus(switches[5] != 0, switches[6] != 0); // 依次向系统状态区添加状态图标int N = iconList.size();.....// 依次向通知栏添加通知N = notificationKeys.size();...../* 至此， 与IStatusBarService的连接已建立， 状态栏与导航栏的窗口也已完成创建与显示，并且保存在IStatusBarService中的信息都已完成了显示或设置。状态栏与导航栏的启动正式完成 */ }
```

可见，状态栏与导航栏的启动分为如下几个过程：

- 获取IStatusBarService， IStatusBarService是运行于system_server的一个系统服务， 它接受操作状态栏/导航栏的请求并将其转发给BaseStatusBar。为了保证SystemUI意外退出后不会发生信息丢失， IStatusBarService保存了所有需要状态栏与导航栏进行显示或处理的信息副本。

- 将一个继承自IStatusBar.Stub的CommandQueue的实例注册到IStatusBarService以建立通信，并将信息副本取回。
- 通过调用子类的createAndAddWindows () 方法完成状态栏与导航栏的控件树及窗口的创建与显示。
- 使用从IStatusBarService取回的信息副本。

7.1.3 理解IStatusBarService

那么IStatusBarService的真身如何呢？它的实现者是StatusBarManagerService。由于状态栏导航栏与它的关系十分密切，因此需要对其有所了解。

与WindowManagerService、InputManagerService等系统服务一样，StatusBarManagerService在ServerThread中创建。参考如下代码：

```
[SystemServer.java-->ServerThread.run()] public void run() {try { /* 创建一个StatusBarManagerService的实例，并注册到ServiceManager中使其成为一个系统服务 */ statusBar = new StatusBarManagerService(context, wm); ServiceManager.addService(Context.STATUS_BAR_SERVICE, statusBar);} catch (Throwable e) {.....} }
```

再看其构造函数：

```
[StatusBarManagerService.java-->StatusBarManagerService.StatusBarManagerService()] public StatusBarManagerService(Context context, WindowManagerService windowManager) {mContext = context;mWindowManager = windowManager;// 监听实体键盘的状态变化mWindowManager.setOnHardKeyboardStatusChangeListener(this); // 初始化状态栏的系统状态区的状态图标列表。关于系统状态区的工作原理将在7.2.3节介绍final Resources res = context.getResources();mIcons.defineSlots(res.getStringArray(com.android.internal.R.array.config_statusBarIcons)); }
```

这基本上是系统服务中最简单的构造函数了，在这里并没有发现能够揭示StatusBar-ManagerService的工作原理的线索（由此也可以预见StatusBarManagerService的实现十分简单）。

接下来参考StatusBarManagerService.registerStatusBar（）的实现。这个方法由SystemUI中的BaseStatusBar调用，用于建立其与StatusBarManagerService的通信连接，并取回保存在其中的信息副本。

```
[StatusBarManagerService.java-->StatusBarManagerService.registerStatusBar()] public void registerStatusBar(IStatusBar bar, StatusBarIconList iconList, List notificationKeys, List notifications, int switches[], List binders) /* 首先是
```

权限检查。状态栏与导航栏是Android系统中的一个十分重要的组件，因此必须避免其他应用调用此方法对状态栏与导航栏进行偷梁换柱。因此要求方法的调用者必须具有一个签名级的权限

```
android.permission.STATUS_BAR_SERVICE  
*/enforceStatusBarService()/* ① 将bar参数保存到mBar成员中。bar的类  
型是IStatusBar，它就是BaseStatusBar中的CommandQueue的Bp端。从  
此之后，StatusBarManagerService将通过mBar与BaseStatusBar进行通  
信。因此可以理解mBar就是SystemUI中的状态栏与导航栏 */mBar =  
bar;// ② 接下来依次为调用者返回信息副本// 系统状态区的图标列表  
synchronized (mIcons) { iconList.copyFrom(mIcons); } // 通知区的通知信  
息synchronized (mNotifications) { for (Map.Entry e:  
mNotifications.entrySet() { notificationKeys.add(e.getKey());  
notifications.addValue(e.getValue()); } } // switches中的杂项synchronized  
(mLock) { switches[0] = gatherDisableActionsLocked(mCurrentUserId);  
.....}..... }
```

可见StatusBarManagerService.registerStatusBar() 的实现也十分简单。主要是保存BaseStatus-Bar中的CommandQueue的Bp端到mBar成员中，然后再把信息副本填充到参数里。尽管简单，但是从其实现中可以预料到StatusBarManagerService的工作方式：当它接受到操作状态栏与导航栏的请求时，首先将请求信息保存到副本之中，然后再将这一请求

通过mBar发送给BaseStatusBar。以设置系统状态区图标这一操作为例，参考如下代码：

[StatusBarManagerService.java-->StatusBarManagerService.setIcon()]

```
public void setIcon(String slot, String iconPackage, int iconId, int iconLevel,
String contentDescription) {/* 首先同样是权限检查，与
registerStatusBar()不同，这次要求的是一个系统级别的权限android.permission.STATUS_BAR。因为设置系统状态区图标的操作不允许普通应用程序进行。其他的操作诸如 添加一条通知则不需要此权限
*/enforceStatusBar();synchronized (mIcons) { int index =
mIcons.getSlotIndex(slot); ..... StatusBarIcon icon = new
StatusBarIcon(iconPackage, UserHandle.OWNER, iconId, iconLevel, 0,
contentDescription); // ① 将图标信息保存在副本中 mIcons.setIcon(index,
icon); // ② 将设置请求发送给BaseStatusBar if (mBar != null) { try {
mBar.setIcon(index, icon); } catch (RemoteException ex) {.....} } }
```

纵观StatusBarManagerService中的其他方法，会发现它们与setIcon () 方法的实现十分类似。从而可以得知StatusBarManagerService的作用与工作原理如下：

- 它是SystemUI中的状态栏与导航栏在system_server中的代理。所有对状态栏或导航来有需求的对象都可以通过获取StatusBarManagerService

的实例或Bp端达到其目的。只不过使用者必须拥有能够完成操作的相应权限。

·它保存了状态栏／导航栏所需的信息副本，用于在SystemUI意外退出之后的恢复。

7.1.4 SystemUI的体系结构

完成对SystemUI的启动过程的分析之后便可以对其体系结构做出总结，如图7-1所示。

·SystemUIService，一个普通的Android服务，它以一个容器的角色运行于SystemUI进程中。在它内部运行着多个子服务，其中之一便是状态栏与导航栏的实现者——BaseStatusBar的子类之一。

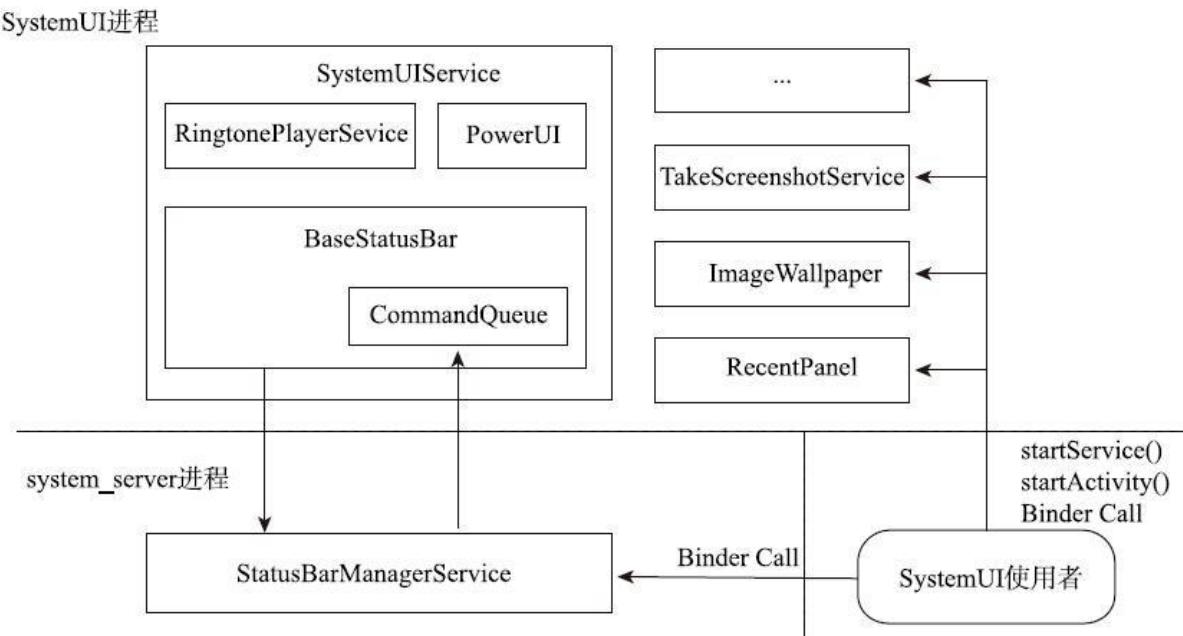


图7-1 SystemUI的体系结构

- IStatusBarService，即系统服务StatusBarManagerService是状态栏/导航栏向外界提供服务的前端接口，运行于system_server进程中。
- BaseStatusBar及其子类是状态栏与导航栏的实际实现者，运行于SystemUIService中。
- IStatusBar，即SystemUI中的CommandQueue是联系StatusBarManagerService与BaseStatusBar的桥梁。
- SystemUI中还包含了ImageWallpaper、RecentPanel以及TakeScreenshotService等功能的实现。它们是Service、Activity等标准的Android应用程序组件，并且互相独立。对这些功能感兴趣的使用者可以通过startService () /startActivity () 等方式方便地启动相应功能。

在本章将主要介绍SystemUI中最常用的状态栏、导航栏以及RecentPanel的实现。Image-Wallpaper将在第8章中详细介绍。而SystemUI其他的功能读者可以自行研究。

7.2 深入理解状态栏

如7.1.1节所述，SystemUI中存在两种状态栏与导航栏的实现——状态栏与导航栏分离布局的PhoneStatusBar以及状态栏与导航栏集成布局的TabletStatusBar。除了布局差异之外，二者并无本质上的差别，因此本节将主要介绍PhoneStatusBar下的状态栏的实现。

作为一个将所有信息集中显示的场所，状态栏对需要显示的信息分为以下5种。

·通知信息：它可以在状态栏左侧显示一个图标以引起用户的主意，并在下拉卷帘中为用户显示更加详细的信息。这是状态栏所能提供的信息显示服务之中最灵活的一种功能。它对信息种类以及来源没有做任何限制。使用者可以通过StatusBarManagerService所提供的接口向状态栏中添加或移除一条通知信息。

·时间信息：显示在状态栏最右侧的一个小型数字时钟，是一个名为Clock的继承自TextView的控件。它监听了几个和时间相关的广播：ACTION_TIME_TICK、ACTION_TIME_CHANGED、ACTION_TIMEZONE_CHANGED以及ACTION_CONFIGURATION_CHANGED。当其中一个广播到来时从Calendar类中获取当前的系统时间，然后进行字符串格式化后显示出

来。时间信息的维护工作在状态栏内部完成，因此外界无法通过API修改时间信息的显示或行为。

·电量信息：显示在数字时钟左侧的一个电池图标，用于提示设备当前的电量情况。它是一个被BatteryController类所管理的ImageView。

BatteryController通过监听`android.intent.action.BATTERY_CHANGED`广播以从BatteryService中获取电量信息，并根据电量信息选择一个合适的电池图标显示在ImageView上。同时间信息一样，这也是在状态栏内部维护的，外界无法干预状态栏对电量信息的显示行为。

·信号信息：显示在电量信息的左侧的一系列ImageView，用于显示系统当前的WiFi、移动网络的信号状态。用户所看到的WiFi图标、手机信号图标、飞行模式图标都属于信号信息的范畴。它们被

NetworkController类维护着。NetworkController监听了一系列与信号相关的广播如`WIFI_STATE_CHANGED_ACTION`、`ACTION_SIM_STATE_CHANGED`、`ACTION_AIRPLANE_MODE_CHANGED`等，并在这些广播到来时显示、更改或移除相关的ImageView。同样，外界无法干预状态栏对信号信息的显示行为。

·系统状态图标区：这个区域用一系列图标标识系统当前的状态，位于信号信息的左侧，与状态栏左侧通知信息隔岸相望。与通知信息类似，StatusBarManagerService通过`setIcon()`接口为外界提供了修改系

统状态图标区的图标的途径，然而它对信息的内容有很强的限制。首先，系统状态图标区无法显示图标以外的信息，另外，系统状态图标区对其所显示的图标数量以及图标所表示的意图有着严格的限制。

由于时间信息、电量信息以及信号信息的实现原理比较简单而且与状态栏外界相对隔离，因此读者可以通过分析上文所介绍的相关组件自行研究。本节将主要介绍状态栏以下几个方面的内容：

- 状态栏窗口的创建与控件树结构。

- 通知的管理与显示。

- 系统状态图标区的管理与显示。

7.2.1 状态栏窗口的创建与控件树结构

1. 状态栏窗口的创建

在7.1.2节所引用的BaseStatusBar.start () 方法的代码中调用了createAndAddWindows () 方法进行状态栏窗口的创建。很显然，createAndAddWindow () 由PhoneStatusBar或Tablet-StatusBar实现。以PhoneStatusBar为例，参考其代码：

```
[PhoneStatusBar.java-->PhoneStatusBar.createAndAddWindow()] public  
void createAndAddWindows() {addStatusBarWindow(); // 直接调用
```

```
addStatusBarWindow()方法 }
```

在addStatusBarWindow () 方法中，PhoneStatusBar将会构建状态栏的控件树并通过 WindowManager 的接口为其创建窗口。

```
[PhoneStatusBar.java-->PhoneStatusBar.addStatusBarWindow()] private  
void addStatusBarWindow() { // ① 通过getStatusBarHeight()方法获取状态  
栏的高度final int height = getStatusBarHeight(); // ② 为状态栏创建  
WindowManager.LayoutParamsfinal WindowManager.LayoutParams lp =  
new WindowManager.LayoutParams(  
ViewGroup.LayoutParams.MATCH_PARENT, // 状态栏的宽度为充满整  
个屏幕宽度 height, // 高度来自getStatusBarHeight()方法  
WindowManager.LayoutParams.TYPE_STATUS_BAR, // 窗口类型  
WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE // 状态栏不接  
受按键事件 /* FLAG_TOUCHABLE_WHEN_WAKING这一标记将使得  
状态栏接受导致设备唤醒的触摸事件。通常这一事件会在  
interceptMotionBeforeQueueing()的过程中被用于唤醒设备（或从变暗  
状态下恢复），而InputDispatcher会阻止这一事件发送给窗口。 */ |  
WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING  
// FLAG_SPLIT_TOUCH允许状态栏支持触摸事件序列的拆分 |  
WindowManager.LayoutParams.FLAG_SPLIT_TOUCH,  
PixelFormat.TRANSLUCENT); // 状态栏的Surface像素格式为支持透明
```

度// 启用硬件加速lp.flags |=

```
WindowManager.LayoutParams.FLAG_HARDWARE_ACCELERATED;//  
StatusBar的gravity是LEFT和FILL_HORIZONTALlp.gravity =  
getStatusBarGravity();lp.setTitle("StatusBar");lp.packageName =  
mContext.getPackageName();// ③ 创建状态栏的控件树  
makeStatusBarView();// ④ 通过WindowManager.addView()创建状态栏的  
窗口mWindowManager.addView(mStatusBarWindow, lp); }
```

此方法提供了很多重要的信息。

首先是状态栏的高度，由getStatusBarHeight () 从资源
com.android.internal.R.dimen.status_bar_height中获得。这一资源定义在
frameworks\base\core\res\res\values\dimens.xml中， 默认为25dip。此资源
同样在PhoneWindowManager中被用来计算作为布局准绳的8个矩形。

然后是状态栏窗口的LayoutParams的创建。LayoutParams描述了状态栏
是怎样的一个窗口。TYPE_STATUS_BAR使得PhoneWindowManager为
状态栏的窗口分配了较大的layer值，使其可以显示在其他应用窗口之
上。FLAG_NOT_FOCUSABLE、
FLAG_TOUCHABLE_WHEN_WAKING和FLAG_SPLIT_TOUCH则定
义了状态栏对输入事件的响应行为。



注意

通过创建窗口所使用的LayoutParams来推断一个窗口的行为十分重要。在分析一个需要创建窗口的模块的工作原理时，从窗口创建过程往往是一个不错的切入点。

另外需要知道的是，窗口创建之后，其LayoutParams是会发生变化的。以状态栏为例，创建窗口时其高度为25dip，flags描述其不可接受按键事件。不过当用户按下状态栏导致卷帘下拉时，PhoneStatusBar会通过 WindowManager.updateViewLayout () 方法修改窗口的LayoutParams的高度为MATCH_PARENT，即充满整个屏幕以使得卷帘可以满屏显示，并且移除FLAG_NOT_FOCUSABLE，使得PhoneStatusBar可以通过监听BACK键以收回卷帘。

在makeStatusBarView () 完成控件树的创建之后，
WindowManager.addView () 将根据控件树创建出状态栏的窗口。显而易见，状态栏控件树的根控件被保存在mStatusBarWindow成员中。

createStatusBarView () 负责从R.layout.super_status_bar所描述的布局中实例化出一棵控件树。从这棵控件树中取出一些比较重要的控件并保存在对应的成员变量中。因此从R.layout.super_status_bar入手可以很容易地得知状态栏的控件树的结构。

2. 状态栏控件树的结构

参考SystemUI中super_status_bar.xml所描述的布局内容，可以看到其根控件是一个名为StatusBarWindowView的控件，它继承自FrameLayout。在其下的两个直接子控件如下：

- @layout/status_bar所描述的布局。这是用户平时所见的状态栏。
- PenelHolder：这个继承自FrameLayout的控件是状态栏的卷帘。在其下的两个直接子控件@layout/status_bar_expanded以及@layout/quick_settings分别对应卷帘中的通知列表面板以及快速设定面板。

在正常情况下，StatusBarWindowView中只有@layout/status_bar所描述的布局是可见的，并且状态栏窗口为com.android.internal.R.dimen.status_bar_height所定义的高度。当StatusBar-WindowView截获ACTION_DOWN的触摸事件后，会修改窗口的高度为MATCH_PARENT，然后将PenelHolder设为可见并跟随用户的触摸轨迹，由此实现了卷帘的下拉效果。



说明

PenelHolder集成自FrameLayout。那么它如何做到在 @layout/status_bar_expanded以及@layout/quick_settings两个控件之间切换显示呢？答案就在第6章所介绍的ViewGroup.getChildAtDrawingOrder()方法中。此方法的返回值影响了子控件的绘制顺序，同时也影响了控件接收触摸事件的优先级。当PenelHolder希望显示 @layout/status_bar_expanded面板时，它在此方法中将此面板的绘制顺序放在最后，使其在绘制时能够覆盖@layout/quick_settings，并且优先接受触摸事件。反之则将@layout/quick_settings的绘制顺序放在最后即可。

因此状态栏控件树的第一层结构如图7-2所示。

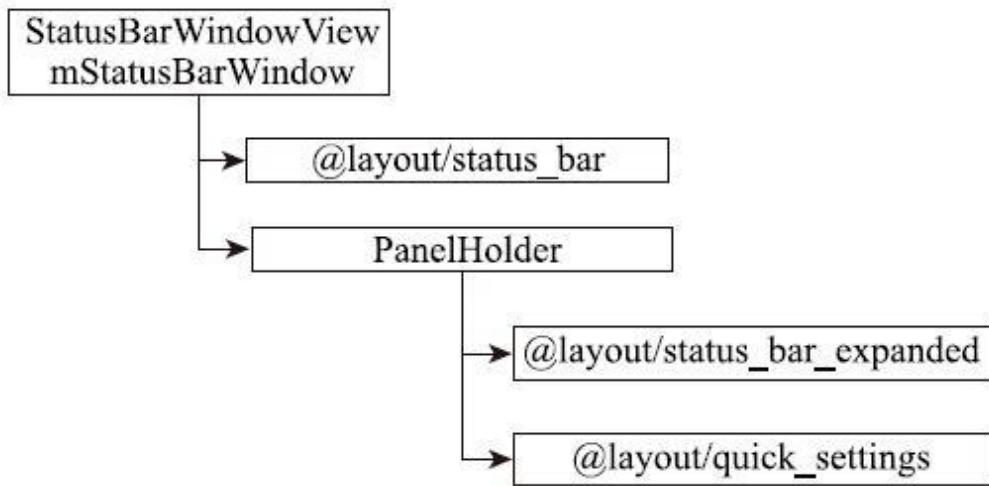


图7-2 状态栏控件树的结构1

再看status_bar.xml所描述的布局内容，其根控件是一个继承自FrameLayout的名为StatusBarView类型的控件，makeStatusBarView（）方法会将其保存为mStatusBarView。其直接子控件有三个：

·@id/notification_lights_out，一个ImageView，并且一般情况下它是不可见的。在SystemUIVisibility中有一个名为SYSTEM_UI_FLAG_LOW_PROFILE的标记。当一个应用程序希望让用户的注意力更多集中在它所显示的内容时，可以在其SystemUIVisibility中添加这一标记。

SYSTEM_UI_FLAG_LOW_PROFILE会使得状态栏与导航栏进入低辨识度模式。低辨识度模式下的状态栏将不会显示任何信息，只是在黑色背景中显示一个灰色圆点而已。而这一个黑色圆点即这里的id/notification_lights_out。

·@id/status_bar_contents，一个LinearLayout，状态栏上各种信息的显示场所。

·@id/ticker，一个LinearLayout，其中包含了一个ImageSwitcher和一个TickerView。在正常情况下@id/ticker是不可见的。当一个新的通知到来时（例如一条新的短信），状态栏上会以动画方式逐行显示通知的内容，使得用户可以在无须下拉卷帘的情况下了解新通知的内容。这一功能在状态栏中被称为Ticker。而@id/ticker则是完成Ticker功能的场所。makeStatusBarView () 会将@id/ticker保存为mTickerView。

至此，状态栏控件树的结构可以扩充为图7-3所示。

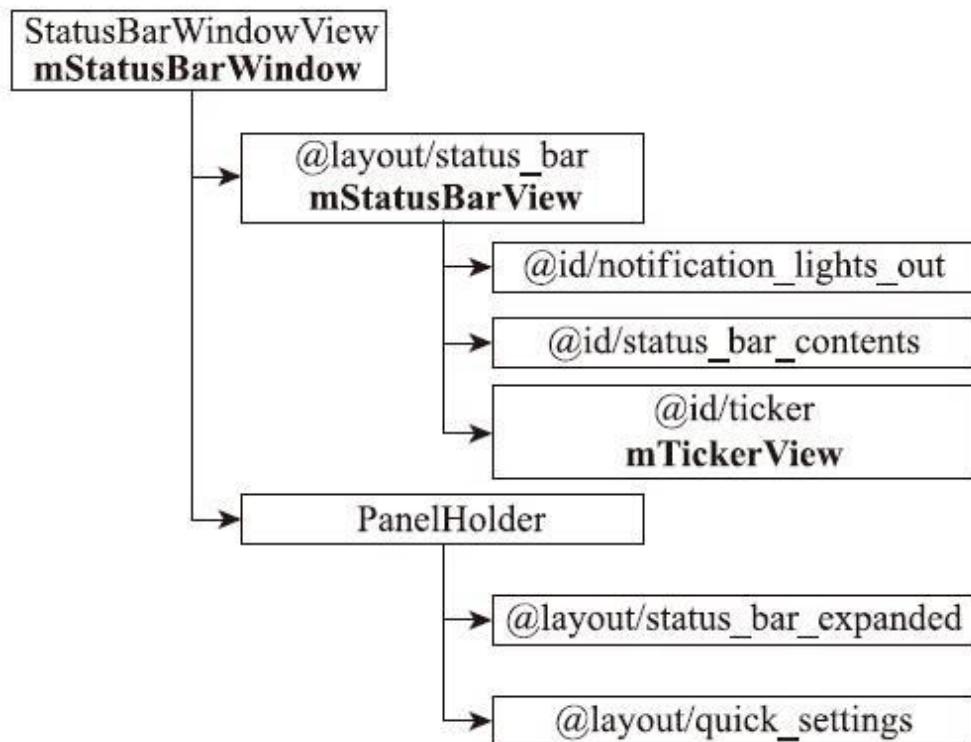


图7-3 状态栏控件树的结构2

再来分析@id/status_bar_contents所包含的内容。如前文所述，状态栏所显示的信息共有5种，因此@id/status_bar_contents中的子控件分别用来显示这5种信息。其中通知信息显示在@id/notification_icon_area里，而其他4种信息则显示在@id/system_icon_area之中。

·@id/notification_icon_area，一个LinearLayout。包含了两个子控件分别是类型为Status-BarIconView的@id/moreIcon以及一个类型为IconMerger的@id/notificationIcons。IconMerger继承自LinearLayout。通知信息的图标都会以一个StatusBarIconView的形式存储在IconMerger之中。而Icon-Meger和LinearLayout的区别在于，如果它在onLayout（）的过程中发现其内部所容纳的StatusBarIconView的总宽度超过了它自身的宽度，则会设置@id/moreIcon为可见，使得用户得知有部分通知图标因为显示空间不够而被隐藏。makeStausBarView（）会将@id/notificationIcons保存为成员变量mNotificationIcons。因此当新的通知到来时，只要将一个StatusBarIconView放置到mNotificationIcons即可显示此通知的图标。

·@id/system_icon_area，也是一个LinearLayout。它容纳了除通知信息的图标以外的4种信息的显示。在其中有负责显示时间信息的@id/clock，负责显示电量信息的@id/battery，负责信号信息显示的@id/signal_cluster以及负责容纳系统状态区图标的一个LinearLayout

——@id/statusIcons。其中@id/statusIcons会被保存到成员变量mStatusIcons中，当需要显示某一个系统状态图标时，将图标放置到mStatusIcons中即可。



注意

@id/system_icon_area的宽度定义为WRAP_CONTENT，而@id/notification_icon_area的weight被设置为1。在这种情况下，@id/system_icon_area将在状态栏右侧根据其所显示的图标个数调整其尺寸。而@id/notification_icon_area则会占用状态栏左侧的剩余空间。这说明了一个问题：系统图标区将优先占用状态栏的空间进行信息的显示。这也是IconMerger类以及@id/moreIcon存在的原因。

于是可以将图7-3扩展为图7-4。

另外，在@layout/status_bar_expanded之中有一个类型为NotificationRowLayout的控件@id/latestItems，并且会被makeStatusBarView（）保存到mPile成员变量中。它位于下拉卷帘中，是通知信息列表的容器。

在分析控件树结构的过程中发现了如下几个重要的控件：

- mStatusBarWindow，整个状态栏的根控件。它包含了两棵子控件树，分别是常态下的状态栏以及下拉卷帘。
- mStatusBarView，常态下的状态栏。它所包含的三棵子控件树分别对应状态栏的三种工作状态——低辨识度模式、Ticker以及常态。这三棵控件树会随着这三种工作状态的切换交替显示。
- mNotificationIcons，继承自LinearLayout的IconMerger控件的实例，负责容纳通知图标。当mNotificationIcons的宽度不足以容纳所有通知图标时，会将@id/moreIcon设置为可见以告知用户存在未显示的通知图标。
- mTickerView，实现了当新通知到来时的动画效果，使得用户可以在无须下拉卷帘的情况下了解新通知的内容。
- mStatusIcons，一个LinearLayout，它是系统状态图标区，负责容纳系统状态图标。
- mPile，一个NotificationRowLayout，它作为通知列表的容器被保存在下拉卷帘中。因此，当一个通知信息除了需要将其图标添加到mNotificationIcons以外，还需要将其详细信息（标题、描述等）添加到mPile中，使得用户在下拉卷帘中可以看到它。

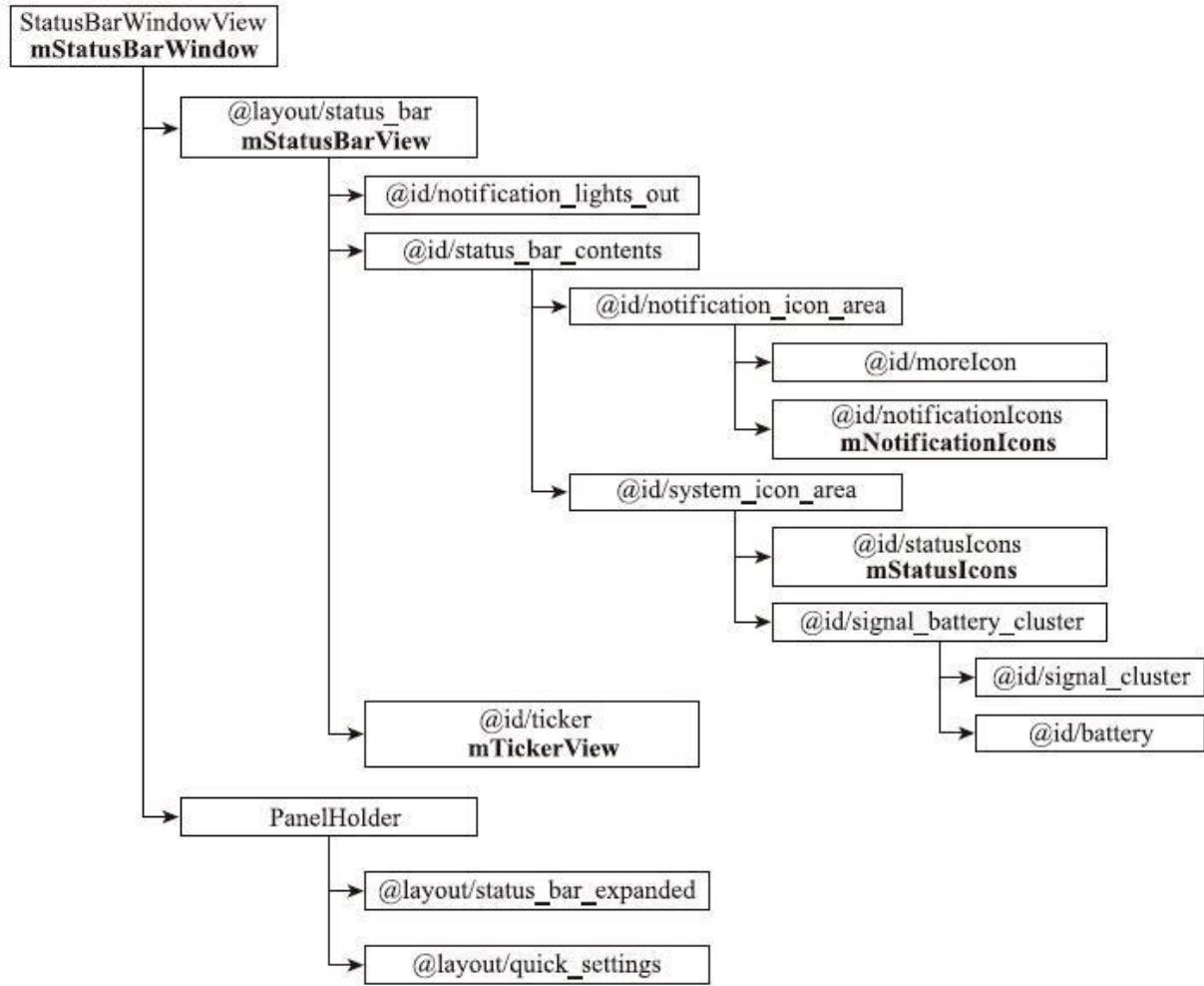


图7-4 状态栏控件树的结构3

对状态栏控件树的结构分析至此告一段落。接下来将从通知信息以及系统状态图标两个方面介绍状态栏的工作原理。希望读者能够理解本节所介绍的几个重要控件所在的位置以及其基本功能，这将使得后续内容的学习更加轻松。

7.2.2 通知信息的管理与显示

通知信息是状态栏中最常用的功能之一。根据用户是否拉下下拉卷帘，通知信息表现为一个位于状态栏的图标，或在下拉卷帘中的一个条目。另外，通知信息还可以在其添加入状态栏之时发出声音，以提醒用户注意查看。通知信息既可以表示一条事件，如新的短消息到来、出现了一条未接来电等，也可以用来表示一个正在后台持续进行的工作，如正在下载某一文件、正在播放音乐等。

1. 通知信息的发送

任何使用者都可以通过NotificationManager所提供的接口向状态栏添加一则通知信息。通知信息的详细内容可以通过一个Notification类的实例来描述。

Notification类中包含如下几个用于描述通知信息的关键字段。

·icon，一个用于描述一个图标的资源id，用于显示在状态栏之上。每条通知信息必须提供一个有效的图标资源，否则此信息将会被忽略。

·iconLevel，如果icon所描述的图标资源存在level，那么iconLevel则用于告知状态栏将显示图标资源的那一个level。

·number，一个int型变量用于表示通知数目。例如，当有三条新的短信时，没有必要使用三个通知，而是将一个通知的number成员设置为3，状态栏会将这一数字显示在通知图标上。

`.contentIntent`, 一个PendingIntent的实例, 用于告知状态栏当在下拉卷帘中点击本条通知时应当执行的动作。`contentIntent`往往用于启动一个Activity以便让用户能够查看关于此条通知的详细信息。例如, 当用户点击一条提示新短信的通知时, 短信应用将会被启动并显示短信的详细内容。

`.deleteIntent`, 一个PendingIntent的实例, 用于告知状态栏当用户从下拉卷帘中删除本条通知时应当执行的动作。`deleteIntent`往往用在表示某个工作正在后台进行的通知中, 以便当用户从下拉卷帘中删除通知时, 发送者可以终止此后台工作。

`.tickerText`, 一条文本。当通知信息被添加时, 状态栏将会在其上逐行显示这条信息。其目的在于使用户无须进行卷帘下拉操作即可从快速获取通知的内容。

`.fullScreenIntent`, 一个PendingIntent的实例, 用于告知状态栏当此条信息被添加时应当执行的动作, 一般这一动作是启动一个Activity用于显示与通知相关的详细信息。`fullScreenIntent`其实是一个替代`tickerText`的设置。当Notification中指定了`fullScreenIntent`时, StatusBar将会忽略`tickerText`的设置。因为这两个设置的目的都是为了让用户可以在第一时间了解通知的内容。不过相对于`tickerText`, `fullScreenIntent`强制性要明显得多, 因为它将打断用户当前正在进行的工作。因此

fullScreenIntent应该仅用于通知非常重要或紧急的事件，比如说来电或闹钟。

·contentView/bigContentView，RemoteView的实例，可以用来定制通知信息在下拉卷帘中的显示形式。一般来讲，相对于contentView，bigContentView可以占用更多空间以显示更加详细的内容。状态栏将根据自己的判断选择将通知信息显示为contentView或bigContentView。

·sound与audioStreamType，指定一个用于播放通知声音的Uri及其所使用的音频流类型。在默认情况下，播放通知声音所用的音频流类型为STREAM_NOTIFICATION。

·vibrate，一个float数组，用于描述震动方式。

·ledARGB/ledOnMS/ledOffMS，指定当此通知被添加到状态栏时设备上的LED指示灯的行为，这几个设置需要硬件设备的支持。

·defaults，用于指示声音、震动以及LED指示灯是否使用系统的默认行为。

·flags，用于存储一系列用于定制通知信息行为的标记。通知信息的发送者可以根据需求在其中加入这样的标记：FLAG_SHOW_LIGHTS要求使用LED指示灯；FLAG_ONGOING_EVENT指示通知信息用于描述一个正在进行的后台工作；FLAG_INSISTENT指示通知声音将持续播

放直到通知信息被移除或被用户查看；FLAG_ONLY_ALERT_ONCE指示任何时候通知信息被加入到状态栏时都会播放一次通知声音；FLAG_AUTO_CANCEL指示当用户在下拉卷帘中点击通知信息时自动将其移出；FLAG_FOREGROUND_SERVICE指示此通知用来表示一个正在以foreground形式运行的服务。

·priority，描述了通知的重要性级别。通知信息的级别从低到高共分为MIN (-2)、LOW (-1)、DEFAULT (0) 以及HIGH (1) 4级。低优先级的通知信息有可能不会显示给用户，或显示在通知列表中靠下的位置。

在随后的讨论中将会详细介绍这些信息如何影响通知信息的显示与行为。

当通知信息的发送者根据需求完成Notification实例的创建之后，便可以通过NotificationManager.notify () 方法将通知显示在状态栏上。

notify () 方法要求通知信息的发送者除了提供一个Notification实例之外，还需要提供一个字符串类型的参数tag，以及int类型的参数id，这两个参数一并确定了信息的意图。当一条通知信息已经被提交给NotificationManager.notify () 并且仍然显示在状态栏中时，它将会被新提交的拥有相同意图（即相同的tag以及相同的id）的通知信息所替换。

参考NotificationManager.notify () 方法的实现：

```
[NotificationManager.java-->NotificationManager.notify()] public void  
notify(String tag, int id, Notification notification) {int[] idOut = new int[1];//  
① 获取NotificationManagerService的Bp端代理INotificationManager  
service = getService();// ② 获取信息发送者的包名String pkg =  
mContext.getPackageName();.....try { // ③ 将包名、 tag、 id以及  
Notification实例一并提交给NotificationManagerService  
service.enqueueNotificationWithTag(pkg, tag, id, notification, idOut,  
UserHandle.myUserId());} catch (RemoteException e) {.....} }
```

NotificationManager会将通知信息发送给NotificationManagerService，并由NotificationManagerService对信息进一步处理。注意，Notification将通知发送者的包名作为参数传递给NotificationManagerService。对一个应用程序来说，tag与id二者一起确定了通知的意图。由于NotificationManagerService作为一个系统服务需要接受来自各个应用程序的通知信息，因此对NotificationManagerService来说，确定通知的意图需要在tag与id之外再增加一项：通知发送者的包名。因此由于包名不一样，来自两个应用程序的具有相同tag与id的通知信息之间不会发生任何冲突。另外将包名作为通知意图的元素之一的原因出于对信息安全的考虑。

而将一则通知信息从状态栏中移除则简单得多，

NotificationManager.cancel () 方法可以提供这一操作，它接受tag、id作为参数用于指明希望移除的通知所具有的意图。

2.NotificationManagerService中的通知信息

接下来看NotificationManagerService如何对通知信息进行管理。

参考NotificationManagerService.enqueueNotificationWithTag () 的实现：

[NotificationManagerService.java--

```
>NotificationManagerService.enqueueNotificationWithTag() public void
enqueueNotificationInternal(String pkg, int callingUid, int callingPid, String
tag, int id, Notification notification, int[] idOut, int userId){/* ①首先安全
检查。在提交通知信息时提供包名可以用于保证通知信息的安全。
checkCallerIsSystemOrSameApp()会获取通知信息提交者的UID，并与
PackageManager中获取分配给拥有指定包名的应用程序的UID进行比
对。倘若二者不相同，则表示有恶意软件尝试通过冒用包名的方式恶
意篡改其他应用程序发出的通知信息，check-
CallerIsSystemOrSameApp()会抛出一个运行时异常以禁止这种行为
*/checkCallerIsSystemOrSameApp(pkg);//倘若包名为android，表示通知
来自Android系统服务final boolean isSystemNotification =
```

("android".equals(pkg));.....// 限制每个应用程序最多只能提交50个通知。这是为了防止恶意软件或通过注册大量通知导致系统瘫痪if
(!isSystemNotification) { if (count >= MAX_PACKAGE_NOTIFICATIONS) { return; } }/* ② 接下来为根据通知信息的重要性（即priority字段）对信息进行打分。由于priority的取值为-2、 -1、 0、 1 4种，因此score此时的取值范围为-20到10之间。随后会根据情况对通知信息的打分结果进行修正。倘若其分数过低，此通知信息便会被忽略 */int score = notification.priority * NOTIFICATION_PRIORITY_MULTIPLIER;/* 倘若发送者所在的应用程序已经被禁止发送通知，则将此通知信息的分数设置为垃圾分数(JUNK_SCORE)垃圾 分数为-1000。默认情况下 NotificationManagerService并未启用这一机制。系统开发者可以通过将ENABLE_BLOCKED_NOTIFICATIONS并通过调用setNotificationEnabledForPackage()方法禁止某一应用程序发送通知 */if (ENABLE_BLOCKED_NOTIFICATIONS && !isSystemNotification && !areNotificationsEnabledForPackageInt(pkg)) { score = JUNK_SCORE;}//倘若分数过低，则忽略此通知信息if (score < SCORE_DISPLAY_THRESHOLD) { return;}// 经过打分检查的通知信息都将得到NotificationManagerService的受理synchronized (mNotificationList) { // ③ 首先会创建一个NotificationRecord实例用于包装所有与此通知相关的信息。包括身份信息 (pkg、 tag、 id、 调用者信

息、打分以及Notification实例等) NotificationRecord r = new
NotificationRecord(pkg, tag, id, callingUid, callingPid, userId, score,
notification); // old表示因为与新通知具有相同的意图而被新通知所替换
的NotificationRecord实例 NotificationRecord old = null; // ④ 将新建的
NotificationRecord添加到mNotificationList列表中进行保存。 // 获取具
有相同意图的通知在mNotificationList列表中的位置 int index =
indexOfNotificationLocked(pkg, tag, id, userId); if (index < 0) { // 相同意
图的通知不存在则直接将其添加到mNotificationList列表中
mNotificationList.add(r); } else { /* 倘若相同意图的通知已经存在，则将
已存在的NotificationRecord从列表中删除，然后再将新的
NotificationRecord加入列表 */ old = mNotificationList.remove(index);
mNotificationList.add(index, r); } /* 倘若通知信息用于描述一个正在
以前台形式运行的服务，则为其添加FLAG_ONGOING_EVENT以及
FLAG_NO_CLEAR两个标记，以确保它不能被用户从下拉卷帘中删除
*/ if ((notification.flags&Notification.FLAG_FOREGROUND_SERVICE)
!= 0) { notification.flags |= Notification.FLAG_ONGOING_EVENT |
Notification.FLAG_NO_CLEAR; } /* ⑤ 将通知信息提交给
StatusBarManagerService。Notification中的icon字段的值对这里的流程
起到了决定性的影响。当icon不为0时，表示这是一个有效的通知信
息，因而它会被提交给StatusBarManagerService。反之则表示这不是
一个有效的通知信息，它非但不会被提交给StatusBar-

ManagerService，还会导致StatusBarManagerService中与它具有相同意图通知被删除 */ if (notification.icon != 0) { /* 创建一个 StatusBarNotification，包装所有与此通知有关的信息。其内容与 NotificationRecord大同小异。它们最大的区别是：NotificationRecord 是通知信息存在于NotificationManagerService中的形式，而 StatusBarNotification则是通知信息存在于StatusBarManagerService中的形式 */ final StatusBarNotification n = new StatusBarNotification(pkg, id, tag, r.uid, r.initialPid, score, notification, user); if (old != null && old.statusBarKey != null) { /* 倘若具有相同意图的通知信息已存在于 NotificationManagerService中，则选择通过updateNotification()对通知进行更新。注意，新的NotificationRecord将继承旧有NotificationRecord的 statusBarKey成员 的值。statusBarKey是一个Binder实例，用于在 StatusBarManagerService中唯一 表示一个通知 */ r.statusBarKey = old.statusBarKey; /* 使用新的StatusBarNotification实例更新 r.statusBarKey所对应的位于StatusBar- ManagerService的通知 */ mStatusBar.updateNotification(r.statusBarKey, n); } else { /* 向 StatusBarManagerService提交新建的StatusBarNotification实例。StatusBarManagerService会为此通知创建一个Binder对象，以此作为通知在 StatusBarManagerService中的唯一标识。当NotificationManagerService 需要对StatusBarManagerService中的通知进行更新或删除操作时，必须提供这一唯一标识 */ r.statusBarKey = mStatusBar.addNotification(n);

```
..... } } else { // 新通知的icon字段为0会导致StatusBarManagerService中的相应通知被删除 mStatusBar.removeNotification(old.statusBarKey); } /*当新通知以StatusBarNotification的形式提交给StatusBarManagerService之后，Notification会根据新通知的要求进行通知音、震动以及LED指示灯的操作。不过这与本章所讨论的System- UI关系不大，因此这里不再赘述。感兴趣的读者可以自行研究 */ .....}idOut[0] = id; }
```

可见，NotificationManagerService接收到一则通知信息时，会首先对一些不希望进行显示的通知进行过滤。这一过滤动作主要分为两个方面：

- 安全性过滤，避免恶意应用通过冒用包名、tag以及id对其他应用程序的信息进行篡改。
- 打分过滤，用于将一些低重要性的，或者由那些位于黑名单中的应用所发送的通知排斥在外。

当一则通知通过安全性以及打分的两层过滤之后，NotificationManagerService会将其封装为一个NotificationRecord，并将其添加到mNotificationList列表中，表示已经接受了这则通知信息。NotificationManagerService使用pkg、tag以及id三个信息一并描述通知的意图。新的通知倘若和现有通知具有相同的意图，NotificationManagerService会将旧有通知从列表中删除，并将新的通知

加入列表。从这个意义上讲，pkg、tag以及id在NotificationManagerService中共同构成了一则通知的唯一标识。



注意

自Android 4.2开始，Android开始支持多用户机制。因此目标用户的userId也构成了通知意图之一，因为发送给用户A的通知不应影响到发送给用户B的通知。因此准确地讲构成通知的意图或唯一标识的应该是pkg、tag、id以及userId 4者共同构成的。不过多用户在最常用的手机设备中并没有被启用。

在NotificationManagerService保存NotificationRecord之后，便开始向StatusBarManager-Service提交通知的信息，以便通知最终能够显示在状态栏上。StatusBarManagerService以StatusBarNotification的形式保存一则通知，其内容与NotificationRecord大同小异。

StatusBarManagerService会为每一个StatusBarNotification分配一个Binder对象。这个Binder对象并没有包含任何IPC操作，它存在的目的是在StatusBarManagerService中唯一地标识一个StatusBarNotification实例。

NotificationRecord会保存这一Binder对象到它的statusBarKey成员中，以便与StatusBarManagerService中的StatusBarNotification实例建立联系。

将通知发送给StatusBarManagerService之后，可以认为通知已经可以显示在状态栏以及下拉卷帘中了。NotificationManagerService下一步便是通过Notification实例中的sound、vibrate、ledARGB等字段进行提示操作。至此在NotificationManagerService中添加通知信息的阶段完成。

NotificationManagerService也提供了接口cancelNotification（）用于响应NotificationManager.cancel（）方法的调用。在了解了新增一则通知的原理之后，不难想到移除一条通知的流程。NotificationManagerService首先会从mNotificationList中找出具有给定意图的NotificationRecord，将其从列表中删除。然后再将NotificationRecord中statusBarKey所标识的StatusBarNotification从StatusBarManagerService中删除。另外，Notification.deleteIntent便是在这一过程中被发送的。

3.StatusBarManagerService中的通知信息

如前文所述，通知信息由NotificationManagerService封装为StatusBarNotification并通过StatusBarManagerService.addNotification（）方法传递给StatusBarManagerService。本节将讨论StatusBarManagerService将如何对其进一步处理。

参考addNotification（）方法的实现：

[StatusBarManagerService.java-->StatusBarManagerService.addNotification()]

```
public IBinder addNotification(StatusBarNotification notification) { synchronized (mNotifications) { // ① 首先创建一个新的Binder对象用作通知的唯一标识
    IBinder key = new Binder(); // ② 以key为键，将通知保存在名为 mNotifications的一个Hashmap中
    mNotifications.put(key, notification); /* 倘若mBar不为null，则表示SystemUI中的BaseStatusBar正处在运行状态。如7.1.3节所述，mBar就是BaseStatusBar中的CommandQueue的Bp端，负责为StatusBarManagerService提供访问BaseStatusBar的渠道 */
    if (mBar != null) { try { /* ③ 通过mBar.addNotification()方法将 StatusBarNotification实例提交给 SystemUI中的BaseStatusBar */
        mBar.addNotification(key, notification); } catch (RemoteException ex)
    {.....} } return key; } }
```

正如7.1.3节所述，StatusBarManagerService是一个简单到不能再简单的系统服务。它不过是状态栏的一个代理。它将外界（如NotificationManagerService）对通知信息的操作转发给运行于SystemUI进程中的状态栏，并且在本地保存了一个通知信息的副本——mNotifications。

注意，即便SystemUI因为某种原因崩溃而使得mBar为null，StatusBarManagerService仍然会接受添加通知信息的请求，为其分配唯

一标识并添加到mNotifications中。因为当SystemUI再次成功启动后调用StatusBarManagerService.registerStatusBar () 方法时，StatusBarManagerService会将mNotifications中存储的所有通知一并返回给SystemUI中的BaseStatusBar用于显示（参考7.1.2节）。因而，StatusBarManagerService的存在使得通知信息的丢失概率降到最低点。

相应的更新或移除通知的操作在StatusBarManagerService中的实现也十分简单。即修改或移除mNotifications列表中的StatusBarNotification，然后再将操作通过mBar转发给SystemUI进程中的BaseStatusBar。

4.状态栏中的通知信息

StatusBarManagerService将StatusBarNotification通过mBar.addNotification () 提交到状态栏中。状态栏将会为通知信息创建相应的控件，并将其添加到其控件树中。

接受StatusBarNotification的场所位于BaseStatusBar.addNotification () 方法中，不过由于BaseStatusBar作为一个抽象类并没有提供addNotification () 的实现，因此需要参考其子类之一的PhoneStatusBar的addNotification () :

```
[PhoneStatusBar.java-->PhoneStatusBar.addNotification()] public void  
addNotification(IBinder key, StatusBarNotification notification) { // ① 通过  
addNotificatoinViews()将通知的内容添加到控件树中StatusBarIconView
```

```
iconView = addNotificationViews(key, notification);if (iconView == null)  
return; // 倘若返回值为null则表示添加失败.....// ② 为新的通知启动  
fullScreenIntent或进行tickerif (notification.notification.fullScreenIntent !=  
null) { /* fullScreenIntent拥有比tickerText更高的优先级。因此当  
fullScreenIntent存在时，将启动fullScreenIntent */ try {  
    notification.notification.fullScreenIntent.send(); } catch  
(PendingIntent.CanceledException e) {} } else { // 否则进行ticker动作 if  
(mCurrentlyIntrudingNotification == null) { tick(null, notification, true);  
} } // ③ 更新周边控件。例如调整下拉卷帘中控件的位置尺寸、设置“清  
空通知”按钮的可用状态等  
setAreThereNotifications();updateExpandedViewPos(EXPANDED_LEAVE  
_ALONE); }
```

PhoneStatusBar.addNotification () 主要分为三个步骤，首先是通过 addNotificationViews () 方法为通知信息创建相应的控件（主要分为通知栏图标以及下拉卷帘中的详细信息两个部分），然后在 fullScreenIntent以及tickerText之间选择一个以提醒用户有新通知，最后更新周边的相关控件。

本节重点介绍addNotificationViews () 方法如何为通知信息创建相应的控件。

addNotificationViews () 的实现位于BaseStatusBar中。

[BaseStatusBar.java-->BaseStatusBar.addNotificationViews()] protected StatusBarIconView addNotificationViews(IBinder key, StatusBarNotification notification) {/* ① 首先创建一个类型为 StatusBarIconView的控件，它用于显示通知的图标。StatusBarIconView 的祖父类是ImageView。它为了适应状态栏图标的需求进行了一些 定制。例如可以在图标之上显示一个数字(Notification.number)，自动 为图标设置level(Notification.iconLevel)，以及从通知发送者的APK中 加载图标资源等。读者将其理解为一个ImageView即可 */final StatusBarIconView iconView = new StatusBarIconView(mContext, notification.pkg + "/0x" + Integer.toHexString(notification.id), notification.notification);iconView.setScaleType(ImageView.ScaleType.CENTER_INSIDE);/* ② 创建StatusBarIcon实例，用于保存在Notification实例 中与通知图标相关的信息*/final StatusBarIcon ic = new StatusBarIcon(notification.pkg, notification.user, notification.notification.icon, notification.notification.iconLevel, notification.notification.number, notification.notification.tickerText);/* 设置 StatusBarIconView所显示的图标*/if (!iconView.set(ic)) { return null; }/* ③ 创建NotificationData.Entry实例。保存key，StatusBarNotification以及 StatusBarIconView。NotificationData.Entry是通知信息在状态栏中存在 的形式 */NotificationData.Entry entry = new NotificationData.Entry(key, notification, iconView);/* ④ 创建通知在下拉卷帘中的控件树，并将其添

```
加到mPile中。关于mPile，请参考7.2.1节if (!inflateViews(entry, mPile)) {  
    return null;}// ⑤ 将保存了新通知的NotificationData.Entry实例保存到  
mNotificationData中int pos = mNotificationData.add(entry);/* ⑥ 更新通知  
在下拉卷帘中的展开状态。一般情况下，所有的通知信息在下拉卷帘  
中都有一个固定的高度。在这一固定高度下通知所能显示的信息十分  
有限。自Android 4.2开始，状态栏允许下拉卷帘中的通知可以展开状  
态。处于展开状态的通知的控件树的LayoutParams.height将会被设置为  
WRAP_CONTENT而不是一个固定高度，使其得以获取充足的空间  
以显示足够详细的信息 */updateExpansionStates();// ⑦ 更新状态栏中所  
有通知图标的显示状态updateNotificationIcons();return iconView; }
```

addNotificationViews () 方法中的内容比较多，总结如下：

- 创建用于在状态栏中显示图标的类型为StatusBarIconView的控件，并为此控件设置需要其显示的图标——StatusBarIcon。StatusBarIcon收集了来自Notification实例中与图标相关的信息供StatusBarIconView显示。
- 为新通知创建一个NotificationData.Entry类型的实例，它保存了StatusBarIconView控件，以及StatusBarNotification实例。NotificationData.Entry是通知信息在状态栏中的存在形式，并且会被添加到mNotificationData列表中。mNotificationData并不是一个简单的ArrayList列表，而是一个在其内部维护了一个ArrayList的NotificationData类的实例。它的add () 方法保证所有Entry都按照其打

分(score)的升序排列。由于通知在状态栏的图标显示顺序以及在下拉卷帘中的显示顺序都与其在NotificationData中的顺序相关。这也是决定打分结果的Notification.priority最大的影响所在。另外，倘若两个通知的打分相同，则通过Notification.when，即通知的时间进行排序，可以理解为新的通知比旧的通知拥有更高的优先级。

·通过inflateViews()方法为新通知创建用于显示在下拉卷帘中的控件树。并将这个控件树放置到mPile中。此时用户可以通过下拉卷帘看到这条通知。

·接下来通过updateExpansionStates()设置通知在下拉卷帘中的展开状态。它会使得在mNotificationData中最后一个通知处于展开状态（高度为WRAP_CONTENT），而其他通知则处于折叠准状态（高度为一个固定高度）。换句话说，此时处于展开状态的是打分最高的通知，或者是最新的通知。

·最后通过updateNotificationIcons()更新状态栏中所显示的图标列表。为什么不简单地将新建的StatusBarIconView添加到mNotificationIcons（参考7.2.1）中呢？因为图标在状态栏中的显示顺序依赖于通知在mNotificationData中的顺序。updateNotificationIcons()会将待显示的StatusBarIconView按照其对应的通知在mNotificationData中的顺序进行排序，使得具有最高打分或最新通知图标的已显示在状态栏的最左侧。至于为什么显示最左侧的，则是因为靠近右侧的图标

有可能会因mNotificationData的宽度不足以显示全部通知图标而变得不可见。

接下来讨论inflateViews () 方法的实现，因为它解释了通知在下拉卷帘中的控件树结构如何。了解了通知在下拉卷帘中的控件树便可以解释为什么通知可以拥有折叠与展开两种状态了。如7.2.2节所述，通知的发送者通过Notification.contentView以及Notification.bigContentView两个RemoteView定义通知在下拉卷帘中的显示内容。因此inflateView () 的重点是如何将这两个RemoteView放置在mPile中，以及如何在这两者之间做出选择。参考其实现：

```
[BaseStatusBar.java-->BaseStatusBar.inflateViews()]
protected boolean inflateViews(NotificationData.Entry entry, ViewGroup parent) {
    // 通知在下拉卷帘中的最小高度为64dp
    int minHeight = mContext.getResources()
        .getDimensionPixelSize(R.dimen.notification_min_height);
    // 通知在下拉卷帘中的最大高度为256dp
    int maxHeight = mContext.getResources()
        .getDimensionPixelSize(R.dimen.notification_max_height);
    StatusBarNotification sbn = entry.notification;
    /* ① 首先获取Notification实例中的
       contentView以及bigContentView。它们分别被保存于oneU 以及large之中 */
    RemoteViews oneU = sbn.notification.contentView;
    RemoteViews large = sbn.notification.bigContentView;
    if (oneU == null) { return false; }
    // contentView是必要的
    LayoutInflater inflater =
```

```
(LayoutInflater)mContext.getSystemService(  
Context.LAYOUT_INFLATER_SERVICE);/* ② 使用LayoutInflater从  
layout资源中创建一棵控件树，并保存在row中。这棵控件树就是通知  
在下拉卷帘中的模板。注意row直接被创建到mPile之中 */View row =  
inflater.inflate(R.layout.status_bar_notification_row, parent, false);.....// ③  
从模板中获取两个ViewGroup，分别是content和adaptive。其中content  
是adaptive的父控件，主要用于接受用户的触摸事件，以便在用户点击  
时启动Notification.contentIntent。而adaptive则用来容纳oneU和large所  
描述的控件树*/ViewGroup content =  
(ViewGroup)row.findViewById(R.id.content);ViewGroup adaptive =  
(ViewGroup)row.findViewById(R.id.adaptive);// ④ 为content设置  
OnClickListener。当用户点击被content消费时将会触发  
contentIntentPendingIntent contentIntent = sbn.notification.contentIntent;if  
(contentIntent != null) { final View.OnClickListener listener = new  
NotificationClicker(contentIntent, sbn.pkg, sbn.tag, sbn.id);  
content.setOnClickListener(listener);} else {  
content.setOnClickListener(null);}// ⑤ 创建由  
Notification.contentView/bigContentView所描述的控件树View  
expandedOneU = null;View expandedLarge = null;try { // 从  
oneU(Notification.contentView)中创建出控件树并保存在expandedOneU  
中 expandedOneU = oneU.apply(mContext, adaptive, mOnClickHandler); if
```

```
(large != null) { /* 如果large(Notification.bigContentView)不为null，则创建出一个控件树并保存在 expendedLarge中 */ expendedLarge = large.apply(mContext, adaptive, mOnClickHandler); } } catch (RuntimeException e) {.....}// ⑥ 从Notification.contentView所创建的控件树被添加到adaptive中if (expandedOneU != null) {  
    SizeAdaptiveLayout.LayoutParams params = new  
    SizeAdaptiveLayout.LayoutParams(expandedOneU.getLayoutParams()); //  
    注意LayoutParams中的最小高度和最大高度都被设置为minHeight  
    params.minLength = minHeight; params.maxLength = minHeight;  
    adaptive.addView(expandedOneU, params);}// ⑦ 从  
Notification.bigContentView所创建的控件树被添加到adaptive中if  
(expandedLarge != null) { SizeAdaptiveLayout.LayoutParams params = new  
SizeAdaptiveLayout.LayoutParams(expandedLarge.getLayoutParams()); //  
注意LayoutParams中的最小高度被设置为minHeight+1，而最大高度被  
设置为maxLength params.minLength = minHeight+1; params.maxLength =  
maxLength; adaptive.addView(expandedLarge, params);}.....// 设置通知是否支持展开状态。依据是Notification是否提供了  
bitContentViewrow.setTag(R.id.expandable_tag, Boolean.valueOf(large !=  
null));// 将本方法所创建的控件树保存到Notification.Entry中entry.row =  
row;entry.content = content;entry.expanded =  
expandedOneU;entry.setLargeView(expandedLarge);return true; }
```

`inflateViews ()` 方法为通知创建了如下5个重要的控件。

·`row`：通知在下拉卷帘中的根控件，除此之外并没有什么特殊性。它存储在`NotificationData.Entry.row`中。

·`content`：根控件下的一个子`ViewGroup`。它通过`OnClickListener`监听那些没有被其子控件所消费的用户点击事件，并在点击到来时出发`Notification.contentIntent`。它存储在`NotificationData.Entry.content`中。

·`adaptive`：`content`的一个子`ViewGroup`，作为`Notification.contentView`以及`Notification.bigContentView`的父控件，它的类型是`SizeAdaptiveLayout`。

·`expandedOneU`：从`Notification.contentView`中创建出来的控件树。它是通知处于折叠状态时所显示的内容。它存储在`NotificationData.Entry.expanded`中。

·`expandedLarge`：从`Notification.bigContentView`中创建出来的控件树。它是处于展开状态时所显示的内容，存储在`NotificationData.Entry.expanded`中。

这里面似乎并没看到对`contentView`以及`bigContentView`进行选择的代码。不过，注意将它们添加到`adaptive`时对`LayoutParams.minHeight/maxHeight`所做的设置，能发现一些有趣的事

情：contentView的LayoutParams.minHeight/maxHeight被设置为minHeight (26dp) , 而bigContentView的LayoutParams.minHeight/maxHeight分别被设置为minHeight (26dp) +1, maxHeight (256dp) 。也就是说，bigContentView的最小高度也比contentView的最大高度要高。

倘若参考一下它们的父控件SizeAdaptiveLayout的selectActiveChild ()方法，可以发现它会根据onMeasure () 时所提供的MeasureSpec在子控件中寻找一个尺寸合适的控件作为Active Child。这个Active Child会在onLayout () 时被设置为VISIBLE，而其他的子控件则会被设置为GONE。因此，当给予adaptive的高度足够时，将会选择bigContentView作为其显示内容，而当高度不足以显示bigContentView时，则会将contentView作为其显示内容。

参考BaseStatusBar中用于展开一则通知的expandView () 方法的实现：

```
protected boolean expandView(NotificationData.Entry entry, boolean expand) { // rowHeight为26dp, 与inflateView()中的minHeight一致。 int rowHeight = mContext.getResources().getDimensionPixelSize(R.dimen.notification_row_min_height); ViewGroup.LayoutParams lp = entry.row.getLayoutParams(); if (entry.expandable() && expand) { /* ① 设置row的高度为WRAP_CONTENT。这使得它能够尽可能地给予adptive足够的高度空间以容 纳其子控件。在这种情况下
```

```
bigContentView将极有可能被选择为Active Child。此时通知将处于 展  
开状态 */ lp.height = ViewGroup.LayoutParams.WRAP_CONTENT;} else  
{ /* ② 设置row的高度为26dp。与contentView的高度相同，却小于  
bigContentView的最小高度。此时adaptive必定会选择contentView作为  
Active Child。此时通知将处于折叠状态 */ lp.height =  
rowHeight;}entry.row.setLayoutParams(lp);return expand; }
```

可见由于SizeAdaptiveLayout的帮助，设置通知在下拉卷帘中的展开与折叠十分简单。



注意

当row的LayoutParams.height被设置为WRAP_CONTENT时，并不一定会选择big-ContentView作为Active Child，因为WRAP_CONTENT并没有给予子控件完全无限制的尺寸。从理论上讲，倘若row的父控件无法给予大于26dp的高度空间，则adaptive仍然会显示contentView而不是bigContentView，只是这种情况实在少见。

总结状态栏接受一则通知信息的关键信息如下：

- 关于状态栏上的通知图标。状态栏将Notification中与图标相关的信息封装在一个StatusBarIcon中，其中包括Notification下的icon、iconLevel、number等字段。StatusBarIcon会交给StatusBarIconView控件显示。
- 关于下拉卷帘中的通知信息。状态栏从R.layout.status_bar_notification_row中创建一棵控件树作为模板。在这棵控件树中，R.id.content用于接受用户的点击事件，并在被点击时触发Notification.contentIntent。R.id.adaptive则用来容纳Notification.contentView以及Notification.bigContentView。R.id.adaptive会在测量时根据可用的高度空间在contentView以及bigContentView二者之间做出选择。
- 通知的StatusBarIconView会作为mNotificationIcons的子控件显示在状态栏上。而通知的R.layout.status_bar_notification_row则会作为mPile的子控件显示在下拉卷帘中。
- 通知的所有信息包括StatusBarNotification以及通知所对应的控件StatusBarIconView、R.layout.status_bar_notification_row、contentView以及bigContentView，都会被保存在NotificationData.Entry中。因此NotificationData.Entry是通知在状态栏中存在的形式。

·所有的NotificationData.Entry按照其打分以升序保存在NotificationData中。Notification-Data.Entry在NotificationData中的顺序决定了通知在状态栏以及下拉卷帘中的显示顺序。

5.总结

本节完整地讨论了通知信息发送者从NotificationManager.notify () 起到通知信息被显示在状态栏及其下拉卷帘中的过程。通知信息先后经历了通知发送者、NotificationManager-Service、StatusBarManagerService、状态栏4个参与者，最终呈现在用户面前。

通知发送者负责定制Notification实例，以描述通知信息的内容及行为。在这里通知信息存在的形式是Notification，并且以tag及id作为通知信息的唯一标识。

NotificationManagerService负责对通知信息进行安全性检查并按照其priority进行打分。打分的结果将决定通知信息在状态栏中的显示顺序，过低的打分甚至会导致此通知信息被忽略。在NotificationManagerService中，通知信息存在的方式是NotificationRecord，被保存在mNotificationList列表中，并且通知发送者的包名pkg、tag以及id是通知信息的唯一标识。新的通知信息会替换具有相同唯一标识的现有通知信息。

StatusBarManagerService负责将通知提交给状态栏。通知信息在这里的存在方式是StatusBarNotification，并以一个Binder实例作为其唯一标识。

状态栏负责将通知信息显示给用户。通知信息在这里的存在方式是NotificationData.Entry，其中保存了StatusBarNotification，以及负责显示的控件。其中通知图标由StatusBarIcon-View显示在mNotificationIcons中，而通知的详细信息则由R.layout.status_bar_notification_row显示在mPile中。

7.2.3 系统状态图标区的管理与显示

系统状态图标区是指状态栏右侧的用于显示一系列用于指示系统状态的图标的区域，用于提示用户系统的当前状态。闹钟、同步等指示图标都显示在这一区域中。从表现形式上来看它与通知图标并无区别，只是状态栏对这些图标的意图进行了严格限定。

就通知的发送者而言，通知的意图由tag与id定义，而NotificationManagerService以及状态栏并不关心意图是什么而一律准予显示。系统状态图标区的图标意图由一个字符串描述。

StatusBarManagerService维护了一个准许显示在系统状态区的预定义的意图列表，这个列表由frameworks/base/core/res/res/values/config.xml中的字符串数组资源config_statusBarIcons定义。StatusBarManagerService

会拒绝使用者提交上述预定义的意图之外的图标。读者可以参考 configStatusBarIcons 的内容了解系统通知区可以显示的意图。



注意

虽说 configStatusBarIcons 中定义了 phone_signal、battery、clock 等意图，不过用户所见的信号图标、电量状态及系统时钟并不属于系统状态图标区。它们由 SystemUI 中的 SignalCluster、BatteryController 以及 Clock 单独维护。请参考 7.2 节开篇所述的内容。

尽管对系统图标的意图限定十分严格，不过系统图标的设置者可以为某个意图设置任意图标。

1. 在系统状态图标区显示图标的方法

在系统状态图标区中显示一个图标可以使用 StatusBarManager.setIcon() 方法完成。这一方法需要以下 4 个参数：

· slot，一个字符串，用于声明图标的意图。它必须存在于 configStatusBarIcons 所预定义的意图列表之中。

·icon，一个用于显示的图标资源id。

·iconLevel，指出图标资源的level。

·contentDescription，一个字符串，用于详细描述图标的含义。

StatusBarManager.setIcon () 方法的实现如下：

```
[StatusBarManager.java->StatusBarManager.setIcon()]
public void
setIcon(String slot, int iconId, int iconLevel, String contentDescription) {try
{ final IStatusBarService svc = getService(); if (svc != null) { // 直接将
setIcon请求转发给StatusBarManagerService svc.setIcon(slot,
mContext.getPackageName(), iconId, iconLevel, contentDescription); }
} catch (RemoteException ex) {.....} }
```

2.StatusBarManagerService对系统状态图标的管理

首先参考StatusBarManagerService.setIcon () 方法的实现：

```
[StatusBarManagerService.java-->StatusBarManagerService.setIcon()]
public void setIcon(String slot, String iconPackage, int iconId, int iconLevel,
String contentDescription) {/* ① 首先是安全性检查。既然系统状态图标
用于表示系统状态，因此必须限定图标设置者的身份。图标 设置者必
须拥有签名级系统权限android.permission.STATUS_BAR才能设置系统
状态图标 */enforceStatusBar();synchronized (mIcons) { /* ② 从mIcons中
```

获取意图的索引。mIcons中存储了所有预定义的意图列表。因此，倘若没能找到给定意图的索引，则说明这不是预定义的意图之一。

```
StatusBarManagerService将会抛出一个 异常终止图标设置工作 */ int  
index = mIcons.getSlotIndex(slot); if (index < 0) { throw new  
SecurityException("invalid status bar icon slot: " + slot); } /* ③ 创建一个  
StatusBarIcon，用于封装与图标相关的信息。注意，系统状态图标使用  
了和通知图标一样的数据结构StatusBarIcon，因此，可以自然而然地  
想到系统图标的显示同样使用了用于显示通知图标的  
StatusBarIconView */ StatusBarIcon icon = new  
StatusBarIcon(iconPackage, UserHandle.OWNER, iconId, iconLevel, 0,  
contentDescription); // ④ 将新的StatusBarIcon保存到mIcons中  
mIcons.setIcon(index, icon); // ⑤ 将新的StatusBarIcon提交给SystemUI中  
的状态栏 if (mBar != null) { try { mBar.setIcon(index, icon); } catch  
(RemoteException ex) {.....} } }
```

设置系统状态图标需要一个签名级系统权限

android.permission.STATUS_BAR。这说明StatusBarManagerService对图标的设置者的身份限制十分严格。不仅仅是由于系统状态图标指示了系统级的状态因而需要对图标的来源足够信任，还因为系统状态图标区对状态栏的空间具有优先占用权（参考7.2.1节），过多的系统状态图标会挤压通知图标的可用空间从而影响用户的体验。因此StatusBarManagerService必须确保设置者是值得信任的。

mIcons是StatusBarIconList类的实例，用于保存系统图标的列表。当StatusBarManager-Service初始化时会通过StatusBarIconList.defineSlots()方法使用config_statusBarIcons所定义的意图列表对StatusBarIconsList进行初始化。初始化后的StatusBarIconsList中会包含两个数组——mSlots以及mIcons（注意不是StatusBarManagerService.mIcons）。其中mSlots数组存储了预定义的图标意图，而mIcons存储了某一意图下所显示的图标信息StatusBarIcon，二者通过数组索引建立关联。因此StatusBarManagerService可以通过StatusBarIconList.getSlotIndex()检查给定的意图是否处于预定义意图列表中。

StatusBarManagerService会为新图标创建一个StatusBarIcon实例用于封装与图标相关的信息，这一数据结构同样被用于存储通知图标的信。新的StatusBarIcon实例会被保存在mIcons（StatusBarIconList）的mIcons中。其在mIcons中的索引与图标意图在mSlots中的索引相同，以保证二者的对应关系。



说明

StatusBarIconList的特点使得将系统状态图标的意图称为slot变得十分形象。当mSlots的每个元素都存储一个预定义的意图之后，mIcons数组中每个索引位置便成为对应意图的插槽。

最后新的StatusBarIcon会通过mBar.addIcon () 方法提交给SystemUI中的状态栏。

3.在状态栏的系统图标区显示图标

流程又来到SystemUI的CommandQueue。CommandQueue中与StatusBarManagerService一样保存了一个StatusBarIconList的实例 mList。CommandQueue会检查给定的意图（这时的意图已经从一个字符串转换为意图在StatusBarIconList中的索引）在mList中是否已经存在一个StatusBarIcon。倘若不存在则会通过调用BaseStatusBar.addIcon () 方法添加一个图标，否则通过BaseStatusBar.updateIcon () 更新图标。

由于BaseStatusBar并没有提供addIcon () 方法的实现，因此需要从其子类中寻找这个方法。以PhoneStatusBar为例，参考其代码：

```
[PhoneStatusBar.java-->PhoneStatusBar.addIcon()]
public void addIcon(String slot, int index, int viewIndex, StatusBarIcon icon) { // 创建一个用于在状态栏中显示图标的StatusBarIconView
    StatusBarIconView view = new StatusBarIconView(mContext, slot, null); // 设置
```

```
StatusBarIconview.set(icon);/* 将新的StatusBarIconView添加到  
mStatusIcons中。 mStatusIcons就是在状态栏中构成系统状态 图标区的  
ViewGroup。 其中的viewIndex参数由图标意图在  
StatusBarIconList.mSlots中的索引产生。 详情请参考Status-  
BarList.getViewIndex()方法 */mStatusIcons.addView(view, viewIndex, new  
LinearLayout.LayoutParams(mIconSize, mIconSize)); }
```

updateIcon () 方法的实现也与之类似：

```
public void updateIcon(String slot, int index, int viewIndex, StatusBarIcon  
old, StatusBarIcon icon) { // 根据viewIndex从系统图标区中获取用于显示  
此意图图标的StatusBarIconViewStatusBarIconView view =  
(StatusBarIconView)mStatusIcons.getChildAt(viewIndex); // 更新  
StatusBarIconView的显示内容view.set(icon); }
```

可见系统状态图标的显示相较于通知的显示来说简单得多。关于
mStatusIcons的介绍请参考7.2.1节。

4. 系统状态图标的主要设置者——PhoneStatusBarPolicy

尽管StatusBarManager提供了设置系统图标的接口，不过这一接口的使
用者并不多。绝大多数系统状态图标都是被一个名为
PhoneStatusBarPolicy类进行设置的。

在7.1.2节所介绍的PhoneStatusBar.start () 方法的最后，PhoneStatusBarPolicy被创建并且保存在PhoneStatusBar.mIconPolicy中。

PhoneStatusBarPolicy的工作原理与7.2节开篇所介绍的NetworkController、BatteryController等组件一样，通过监听一系列与系统状态相关的广播，并在这些广播到来之时通过调用StatusBarManager.setIcon () 接口修改系统状态图标。

PhoneStatusBarPolicy所监听的广播列表如下：

- Intent.ACTION_ALARM_CHANGED。
- Intent.ACTION_SYNC_STATE_CHANGED。
- AudioManager.RINGER_MODE_CHANGED_ACTION。
- BluetoothAdapter.ACTION_STATE_CHANGED。
- BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED。
- TelephonyIntents.ACTION_SIM_STATE_CHANGED。
- TtyIntent.TTY_ENABLED_CHANGE_ACTION。

在某一广播到来时，PhoneStatusBarPolicy会根据广播的类型分别调用updateAlarm () 、updateSyncState () 、updateBluetooth () 、

`updateVolume ()`、`updateSimState ()`、`updateTTY ()` 对特定意图的系统图标进行设置。

5.总结

状态栏系统状态图标区的管理与显示比较简单。读者在理解如何设置系统状态图标之外，还应当理解如何扩充预定义的图标意图列表。简简单来说，扩充位于`frameworks/base/core/res/res/values/config.xml`中的`config_statusBarIcons`即可。

7.2.4 状态栏总结

本节主要介绍状态栏中两种类型信息的管理与显示的原理，即通知信息以及系统状态图标。其他三种信息（电量信息、信号信息以及系统时钟）的实现相对简单，读者可以以本节的介绍为引导自行研究。

另外，状态栏的行为与现实还受到一个重要信息的影响，即`SystemUIVisibility`。由于它同时还影响了导航栏的显示与行为。因此本章将在完成导航栏的介绍之后再对`SystemUI-Visibility`进行讨论。

7.3 深入理解导航栏

导航栏是指显示在屏幕底端或右端容纳了一排虚拟按键的一个窗口。

导航栏之所以存在有两个目的：

- 通过提供虚拟按键作为物理键的替代品。其中最常用的是BACK、HOME以及RECENT这三个键。
- 可以为某些行为提供最快捷的入口。因为导航栏是常驻屏幕的窗口，因此导航栏是启动某些行为最快捷也是最方便的场所。目前Android在导航栏上提供了快速启动搜索的入口。

导航栏的存在占用了屏幕的一块显示区域。尽管很多用户对此颇有微词，然而在移动设备的屏幕尺寸越来越大的情况下，导航栏所能提供的好处已远远超过了它所占用的那一块小小的空间的价值。导航栏取代物理按键无疑节约了设备的制造成本，而更重要的是，由于软件永远比硬件灵活，导航栏能够通过增加或删除其上的虚拟按键以适应不同的应用场景。例如导航栏可以仅为拥有选项菜单的应用显示菜单键，也可以为不能使用BACK键的应用隐藏BACK键。另外，导航栏还可以作为快速启动某些行为的入口。因此基于导航栏的用户体验大有文章可做，这是物理按键所不能比拟的。



说明

在平板电脑等大屏幕设备上是没有独立的导航栏的。这些设备上所拥有的是集成了状态栏与导航栏的系统栏。不过除了表现形式上的差异之外，实质的工作原理是类似的。本节所讨论的是应用于小尺寸屏幕上的独立导航栏。读者可以类比地对系统栏进行研究。

7.3.1 导航栏的创建

同状态栏一样，导航栏的窗口也是通过`WindowManager.addView ()`方法进行创建的。因此讨论导航栏的创建需要留意两个内容，即导航栏控件树的创建，以及导航栏窗口的创建两部分。前者可以揭示导航栏存在那些功能，以及这些功能的显示方式，而后者则可以揭示导航栏的窗口特点。

1. 导航栏控件树的创建与结构

作为小屏幕设备特有的组件，导航栏控件树的创建很自然地位于`PhoneStatusBar`之中。`PhoneStatusBar`的`makeStatusBarView ()`方法不

仅创建了状态栏的控件树，同时也创建了导航栏的控件树。参考如下代码：

```
[PhoneStatusBar.java-->PhoneStatusBar.makeStatusBarView()]
protected PhoneStatusBarView makeStatusBarView() {..... // 创建状态栏控件树的代码
try { // ① 首先向WMS询问是否需要导航栏
    boolean showNav = mWindowManagerService.hasNavigationBar();
    if (showNav) { // ② 从 R.layout.navigation_bar 中创建导航栏的控件树
        mNavigationBarView = (NavigationBarView) View.inflate(context, R.layout.navigation_bar, null);
        // 设置在导航栏中被禁用的功能
        mNavigationBarView.setDisabledFlags(mDisabled);
        // 将PhoneStatusBar设置给导航栏，以便导航栏可以向状态栏查询一些状态
        mNavigationBarView.setBar(this);
    }
} catch (RemoteException ex) {.....}
}
```

是否创建导航栏的控件树取决于

WindowManagerService.hasNavigationBar () 。这一方法的返回值取决于 PhoneWindowManager 中的 mHasNavigationBar 成员的取值。与

mHasSystemNavBar 类似， mHasNavigationBar 的设置位于 PhoneWindowManager.setInitialDisplaySize () 之中。

mHasNavigationBar 为 true 的前提是不使用系统栏，即

mHasSystemNavBar 为 false。之后 mHasNavigationBar 的值取决于位于 frameworks/base/core/res/res/values/config.xml 中的

`config_showNavigationBar`的取值。对拥有物理按键的设备来说，可以将`config_showNavigationBar`设置为`false`，从而避免创建导航栏。简单来说，对短边大于720dp的设备来说不需要导航栏，而对小于720dp的设备来说，是否需要导航栏取决于`config_showNavigationBar`的设置。



说明

在屏幕短边小于720dp但是大于600dp的设备上，导航栏会固定在屏幕的底部，而对短边在600dp以下的设备来说，导航栏的位置会随着设备的方向而改变。在这种情况下，如果屏幕处于竖直状态（portrait）则导航栏位于屏幕的底部，如果屏幕处于水平状态（landscape）则导航栏会位于屏幕的右侧。这是因为屏幕处于水平状态时底部的导航栏会占用本不宽裕的高度。

导航栏的控件树来源于`R.layout.navigation_bar`。它的根控件类型为`NavigationBarView`，并且被保存在`PhoneStatusBar.mNavigationBarView`中。

之后`PhoneStatusBar`向`NavigationBarView`传递了两个信息——禁用功能列表`mDisabled`以及`PhoneStatusBar`自身。`mDisabled`通过按位与的方式

存储了一系列被禁用的功能，NavigationBarView会据此隐藏某些虚拟按键或禁止用户通过滑动手指启动搜索界面。PhoneStatusBar则被NavigationBarView用来查询与状态栏相关的一些状态。NavigationBar需要通过这些状态决定是否允许用户通过在导航栏上通过滑动手指启动搜索界面。需要禁止启动搜索界面的状态主要有：此操作被mDisabled所禁止、状态栏的下拉卷帘正在显示（这种情况下在导航栏上滑动手指被用来关闭下拉卷帘），以及设备尚未完成初始设置。

在PhoneStatusBar.makeStatusBarView () 方法所揭示的这些关于导航栏的信息中，最重要的当属R.layout.navigation_bar所描述的控件树。

R.layout.navigation_bar定义在

frameworks/base/packages/SystemUI/res/layout/navigation_bar.xml中。参考这个文件的内容，可以发现NavigationBarView是其根控件，并且有趣的是在这一根控件之中定义了两套导航栏的控件树——以水平方式进行布局的@id/rot0，以及以垂直方式进行布局的@id/rot90。除了布局方向有所差异之外，二者所包含的内容完全一致。其中@id/rot0是导航栏位于屏幕底部时所使用的控件树，而@id/rot90则是导航栏位于屏幕右侧时所使用的控件树。由于二者的内容完全一样（包括子控件的id），因此导航栏可以根据其显示位置在二者之间进行无缝切换。

无论是水平布局的R.id.rot0还是垂直布局的R.id.rot90，它们的控件树结构如图7-5所示。

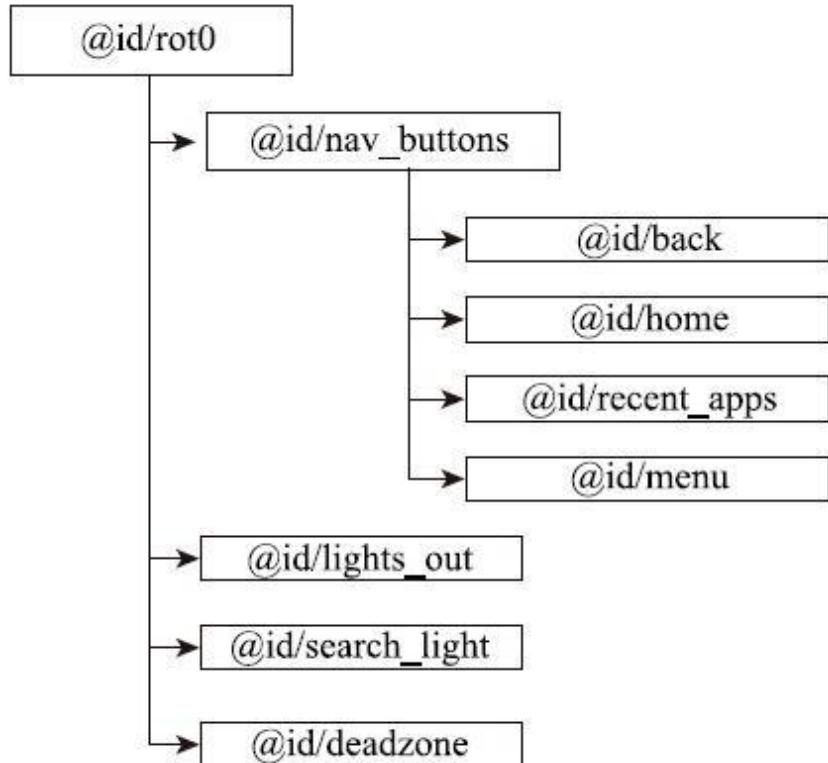


图7-5 R.id.rot0的控件树结构

- @id/nav_buttons，一个LinearLayout。其内部包含了4个类型为 KeyButtonView的子控件：@id/back、@id/home、@id/recent_apps和 @id/menu。它们分别对应虚拟的BACK键、HOME键、RECENT键以及 MENU键。
- @id/lights_out，一个LinearLayout，覆盖在@id/nav_buttons之上（因为 @id/rot0以及@id/rot90是FrameLayout），并且在其中拥有三个 ImageView。这些ImageView都用来显示 @drawable/ic_sysbar_lights_out_dot_large所描述的一个小型的灰色圆点。@id/lights_out在绝大多数情况下都处于不可见状态。同状态栏一

样，导航栏也有低辨识度模式。处于低辨识度模式下的导航栏会将@id/lights_out设为可见并隐藏@id/nav_buttons。此时的导航栏显示为三个不明显的灰色圆点，以降低对用户视线的干扰。

·@id/search_light，一个KeyButtonView，覆盖在@id/lights_out之上并水平方向居中显示（在@id/rot90中为垂直居中），并且在绝大多数情况下都是不可见的。它存在的意义是当HOME键被禁用之后，倘若通过滑动手指启动搜索界面的功能没有被禁用，则将这个KeyButtonView设置为可见，用于提示用户搜索功能仍然可用。

·@id/deadzone，一个DeadZone类型的控件，覆盖于其他所有控件之上。它存在的意义在于避免用户的误操作。由于导航栏紧邻着应用程序的窗口，于是当用户点击应用程序中靠近导航栏位置的界面元素时便会有概率误触导航栏上的虚拟按键（想象一下当用户在一个应用程序中花费半天时间填写一份表格之后不小心按了BACK键后有多恼火吧）。由于DeadZone的存在并且位于虚拟按键之上，用户的触摸事件会首先被DeadZone接收。DeadZone会将那些过于接近导航栏边缘的触摸事件当作用户的误操作，并将其消费掉，从而避免触摸事件派发给虚拟按键。

至此相信读者对导航栏控件树的相关信息已经有所了解。接下来讨论导航栏窗口的创建。

2. 导航栏窗口的创建

创建导航栏窗口的时机位于7.1.2节所介绍的PhoneStatusBar.start () 方法中。参考其代码：

```
[PhoneStatusBar.java-->PhoneStatusBar.start()]
public void start() {...../*
BaseStatusBar.start()方法会调用PhoneStatusBar.makeStatusBarView() 因
此导航栏控件树的创建在这里完成 */super.start(); // 创建导航栏的窗口
addNavigationBar();..... }
```

进一步，参考addNavigationBar () 的实现：

```
[PhoneStatusBar.java-->PhoneStatusBar.addNavigationBar()]
private void
addNavigationBar() {/*倘若mNavigationBarView为null则表示
makeStatusBarView()由于WMS.hasNavigationBar()的返回值为false，即
系统不需要导航栏。因此跳过导航栏窗口的创建 */
if
(mNavigationBarView == null) return; /* ① prepareNavigationBarView()负
责为NavigationBarView中的虚拟按键(KeyButtonView)设置 用于响应用
户触摸事件的监听器OnClickListner或OnTouchListener。这些监听器将
用来产生并向输入 系统注射虚拟的按键事件。 另外，这里还在前文所
述的@id/rot0以及@id/rot90两棵控件树之间做出选择
*/prepareNavigationBarView(); // ② 将mNavigationBarView作为根控件创
```

```
建导航栏的窗口mWindowManager.addView(mNavigationBarView,  
getNavigationBarLayoutParams()); }
```

在使用mNavigationBarView作为根控件创建导航栏窗口之前，PhoneStatusBar首先通过prepareNavigationBarView () 方法对NavigationBarView进行一些准备工作。其中包括为虚拟按键设置监听器，以及在@id/rot0与@id/rot90之间做出选择。这些内容将在7.3.4节介绍。此刻最值得关注的信息是getNavigationBarLayoutParams () 会为NavigationBar创建怎样的LayoutParams。



说明

navigation_bar.xml中还定义了@id/rot270，用作导航栏处于屏幕左端时的控件树。不过目前导航栏永远位于屏幕右端，所以@id/rot270并没有被使用。

```
[PhoneStatusBar.java-->PhoneStatusBar.getNavigationBarLayoutParams()]  
private WindowManager.LayoutParams getNavigationBarLayoutParams()  
{WindowManager.LayoutParams lp = new WindowManager.LayoutParams(  
// ① 导航栏的宽度与高度都是MATCH_PARENT
```

```
LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT, // ②  
类型是TYPE_NAVIGATION_BAR  
WindowManager.LayoutParams.TYPE_NAVIGATION_BAR, 0 |  
WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING |  
WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE // 不接受按键  
事件 | WindowManager.LayoutParams.FLAG_NOT_TOUCH_MODAL //  
不阻挡位于其下的 // 窗口获取点击事件 // ③ 当用户在其他窗口上点击  
时可以收到通知 |  
WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH |  
WindowManager.LayoutParams.FLAG_SPLIT_TOUCH,  
PixelFormat.OPAQUE);.....lp.setTitle("NavigationBar"); // 窗口名称  
lp.windowAnimations = 0; // 不使用窗口动画return lp; }
```

getNavigationBarLayoutParams () 方法为导航栏所创建的LayoutParams 与状态栏比较相似，其中值得关注的不同之处有以下三处。

首先，导航栏窗口的类型是TYPE_NAVIGATION_BAR。

PhoneWindowManager在windowTypeToLayer () 方法中为此类型的窗
口分配了非常高的layer值——19，因而它将显示在绝大多数窗口之
上。为了在小屏幕设备中使导航栏能够在屏幕底端和右端之间移动，
PhoneWindowManager为导航栏创建了下文所述的特殊布局策略，而且
Phone-WindowManager会根据SystemUIVisibility的设置对导航栏的窗口

进行隐藏或显示操作。因此PhoneWindowManager需要保存导航栏的WindowState以便进行这些操作。TYPE_NAVIGATION_BAR是PhoneWindowManager识别导航栏WindowState的唯一依据。

另外，导航栏的宽度与高度都是MATCH_PARENT。读者可能会比较好奇，如此设置导航栏岂不会充满整个屏幕？回顾第4章所介绍的与窗口布局相关的内容可知，将窗口的尺寸设置MATCH_PARENT并不是充满整个屏幕，而是充满那个由PhoneWindowManager为此窗口所提供的ParentFrame。导航栏的ParentFrame的计算是在PhoneWindowManager.beginLayoutLw () 中完成的。参考相关代码：

[PhoneWindowManager.java-->PhoneWindowManager.begineLayoutLw()]
public void beginLayoutLw(boolean isDefaultDisplay, int displayWidth, int displayHeight, int displayRotation) {.....// 倘若存在导航栏的窗口if (mNavigationBar != null) { /* ① 首先选择导航栏所在的位置。
mNavigationBarCanMove决定了导航栏的位置是否可以在屏幕底端和右端之间移动。它是在前文所述的
PhoneWindowManager.setInitialDisplaySize()方法中进行设置。当屏幕的短边尺寸小于600dp时，mNavigationBarOnBottom为true，即可以移动。否则为false，即导航栏会被固定在屏幕的底端。在
mNavigationBar为true的情况下，倘若设备的宽度小于高度，则显示在屏幕低端，否则显示在屏幕的右端。换句话说，在导航栏可以移动的

情况下，它永远显示在屏幕的短边之上 */ mNavigationBarOnBottom =
(!mNavigationBarCanMove || displayWidth < displayHeight); // ② 计算导航栏的ParentFrame
if (mNavigationBarOnBottom) { int top =
displayHeight - mNavigationBarHeightForRotation[displayRotation]; /* 当
导航栏位于屏幕底端时，其ParentFrame是一个高度为
mNavigationBarHeightFor- Rotation的位于屏幕底部的矩形 */
mTmpNavigationFrame.set(0, top, displayWidth, displayHeight); } else
{ int left = displayWidth -
mNavigationBarWidthForRotation[displayRotation]; /* 当导航栏位于屏幕
右端时，其ParentFrame是一个高度为mNavigationBarWidthForRot- ation
的位于屏幕右侧的矩形 */ mTmpNavigationFrame.set(left, 0,
displayWidth, displayHeight); } // ③ 最后对导航栏进行布局。可
见不止ParentFrame，导航栏的DisplayFrame、ContentFrame以及
VisibleFrame都是mTmpNavigationFrame */
mNavigationBar.computeFrameLw(mTmpNavigationFrame,
mTmpNavigationFrame, mTmpNavigationFrame,
mTmpNavigationFrame); } }

PhoneWindowManager对导航栏进行布局时，会首先根据屏幕的宽高确
定设备方向为竖直或水平。当设备处于竖直方向时会在屏幕底部选择
一个矩形作为其ParentFrame，而处于水平方向时会在屏幕右侧选择一
个矩形作为其ParentFrame。由于导航栏LayoutParams中所设置的尺寸为

MATCH_PARENT，因此ParentFrame就是导航栏窗口最终的显示区域。计算ParentFrame过程中所使用的mNavigationBarHeightForRotation以及mNavigationBarWidthForRotation两个数组也是在PhoneWindowManager.setInitialDisplaySize（）方法中初始化的。它们的取值来自位于frameworks/base/core/res/res/values/dimens.xml中的navigation_bar_height以及navigation_bar_height_landscap两个资源，所以，如果需要调整导航栏的尺寸，可以从修改这两个资源入手。导航栏将其窗口宽与高都设置为MATCH_PARENT的目的就是将自己的尺寸全权交于PhoneWindowManager决定。

导航栏布局参数中另外一个独特的地方是它声明了FLAG_WATCH_OUTSIDE_TOUCH标记。声明这一标记的窗口可以在用户点击其他窗口时收到一个名为ACTION_OUTSIDE的触摸事件。在导航栏中，对ACTION_OUTSIDE感兴趣的是上一小节所介绍的DeadZone。DeadZone用于屏蔽那些发送给导航栏，但是距离导航栏边界很近的触摸事件以防止用户误按虚拟按键，这相当于DeadZone在导航栏边界附近设置了一个不会响应任何用户操作的死区。事实上，这一死区的高度（或显示在屏幕右端时的宽度）并不是固定不变的。在一般情况下，这一死区的高度为12dp（定义于res/values/dimens.xml中的navigation_bar_deadzone_size）。而当用户点击了导航栏以外的区域使得DeadZone收到ACTION_OUTSIDE之后，死区的高度会立刻增至32dp（定义于res/values/dimens.xml中的

navigation_bar_deadzone_size_max），并在随后的333毫秒（定义于res/values/config.xml中的navigation_bar_deadzone_decay）中逐渐缩小至正常状态的12dp。动态的死区高度降低了用户在操作应用程序的过程中误触导航栏虚拟按键的概率，因为操作应用程序的过程中ACTION_OUTSIDE事件使得死区高度很大。同时也保证用户点击虚拟按键时尽可能少地受到死区的影响，因为当用户终止操作应用程序之后死区的高度会变得很小。

7.3.2 虚拟按键的工作原理

作为物理按键的替代品，维护虚拟按键是导航栏最主要的工作。在第5章介绍InputDispatcher时所提供的injectInputEvent（）函数是虚拟按键的实现基础。它会将调用者自制的一个输入事件加入InputDispatcher的派发队列中并派发，就仿佛这一输入事件来自InputReader。

InputDispatcher::injectInputEvent（）函数经由InputManager.injectInputEvent（）方法向java层的使用者提供调用接口。而导航栏中的KeyButtonView就是这一接口的使用者之一。KeyButtonView继承自ImageView，它对ImageView最主要的扩展就是根据派发给它的触摸事件转化为按键事件，并通过InputManager.injectInputEvent（）方法将按键事件注入InputDispatcher的派发队列。KeyButtonView中最重要的字段是mCode，用于指示其生成的按键事件的键值。倘若没有为KeyButtonView指定mCode，那么它

的行为与一般的ImageView没什么实质的区别。在KeyButtonView中另外一个重要的字段是mSupportsLongPress，它的取值决定了用户长按KeyButtonView时是否产生按键的长按事件。这两个字段可以在layout中通过systemui : keyCode以及systemui : keyRepeat进行设置。

导航栏中有4个KeyButtonView，分别是@id/back、@id/home、@id/recent_app以及@id/menu。除了@id/recent_app以外，其他三个KeyButtonView都被设置了相应的mCode，用于产生KEY_BACK、KEY_HOME以及KEY_MENU三种按键事件。@id/recent_app并不会产生按键事件，在其上的点击动作会显示或隐藏RecentPanel。

1.从触摸事件到按键事件的映射

既然KeyButtonView的工作是将触摸事件转换为按键事件，因此其核心工作位于onTouchEvent()方法中。参考其代码：

```
[KeyButtonView.java-->KeyButtonView.onTouchEvent()]
public boolean
onTouchEvent(MotionEvent ev) {final int action = ev.getAction();int x,
y;switch (action) { case MotionEvent.ACTION_DOWN: mDownTime =
SystemClock.uptimeMillis(); setPressed(true); /* ① 如果KeyButtonView被
设置了mCode，则创建并发送给InputDispatcher一个ACTION_DOWN
的按键事件 */ if (mCode != 0) { sendEvent(KeyEvent.ACTION_DOWN,
0, mDownTime); } else {.....} // 倘若KeyButtonView被设置为支持长按事
```

件，则发送一个名为mCheckLongPress的 Runnable，并在延迟一段时间后执行它。这个Runnable会重新发送一个带有LONG_PRESS标记的

```
ACTION_DOWN* if (mSupportsLongpress) {  
    removeCallbacks(mCheckLongPress); postDelayed(mCheckLongPress,  
        ViewConfiguration.getLongPressTimeout()); } break; ..... case  
MotionEvent.ACTION_CANCEL: setPressed(false); if (mCode != 0) { // ②  
当触摸事件被取消，则发送一个带有FLAG_CANCELED标记的  
ACTION_UP按键事件 sendEvent(KeyEvent.ACTION_UP,  
KeyEvent.FLAG_CANCELED); } // 因为触摸事件被取消，所以不需要  
发送长按事件。此时需要取消尚未执行的mCheckLongPress if  
(mSupportsLongpress) { removeCallbacks(mCheckLongPress); } break;  
case MotionEvent.ACTION_UP: final boolean doIt = isPressed();  
setPressed(false); if (mCode != 0) { if (doIt) { // ③ 发送一个ACTION_UP  
按键事件 sendEvent(KeyEvent.ACTION_UP, 0); } else { ..... } } else { if  
(doIt) { // ④ 倘若KeyButtonView中没有设置mCode，则在此时触发  
OnClickListener performClick(); } } // 既然用户已经抬起按在  
KeyButtonView上的手指，因此不再需要发送长按事件 if  
(mSupportsLongpress) { removeCallbacks(mCheckLongPress); }  
break;}return true; }
```

简单来说，当KeyButtonView被设置了一个有效键值的情况下，将按照如下方式完成触摸事件到按键事件的映射：

- 触摸事件的ACTION_DOWN对应按键事件的ACTION_DOWN。
- 触摸事件的ACTION_UP对应按键事件的ACTION_UP。
- 触摸事件的ACTION_CANCEL对应按键事件的ACTION_UP+FLAG_CANCELED。
- 当用户长按KeyButtonView时， 对应按键事件的ACTION_DOWN+FLAG_LONG-PRESSED

导航栏中的@id/back、 @id/home以及@id/menu三个KeyButtonView采用了上述工作方式。

另外，如果KeyButtonView没有被设置一个有效的键值，那么当用户点击它时并不会产生任何按键事件，而是触发监听器OnClickListener。导航栏中的@id/recent_app采用了这种工作方式。

2. 键盘事件的发送

KeyButtonView.onTouchEvent () 完成了从触摸事件到按键事件的映射，而KeyButtonView.sendEvent () 则完成了按键事件的创建与发送。
参考其代码：

```
[KeyButtonView.java-->KeyButtonView.sendEvent()] void sendEvent(int action, int flags, long when) { // 计算repeatCountfinal int repeatCount =
```

```
(flags & KeyEvent.FLAG_LONG_PRESS) != 0 ? 1 : 0;// 创建  
KeyEvent final KeyEvent ev = new KeyEvent(mDownTime, when, action,  
mCode, repeatCount, 0, KeyCharacterMap.VIRTUAL_KEYBOARD, 0,  
flags | KeyEvent.FLAG_FROM_SYSTEM |  
KeyEvent.FLAG_VIRTUAL_HARD_KEY,  
InputDevice.SOURCE_KEYBOARD);/* 通过  
InputManager.injectInputEvent()方法将KeyEvent加入InputDispatcher的派  
发队列。INJECT_INPUT_EVENT_MODE_ASYNC表示  
injectInputEvent()会在事件加入派发队列后立刻返回，不等待事 件的派  
发成功与否 */InputManager.getInstance().injectInputEvent(ev,  
InputManager.INJECT_INPUT_EVENT_MODE_ASYNC); }
```



注意

KeyButtonView必须自行发送长按事件并维护repeatCount，这是因为
InputDispatcher仅会为来自InputReader的事件生成长按事件以及
repeatCount。

7.3.3 SearchPanel

导航栏几乎一直显示在屏幕之上，所以它可以是用来启动某个工作的最快捷的入口，而SearchPanel就利用了导航栏的这个优势。通过在导航栏上滑动拇指来启动搜索界面是一个既酷又方便的操作。本节将讨论SearchPanel的启动方式以及实现原理，并且希望读者通过本节的学习可以扩展SearchPanel以启动除了SearchPanel之外的其他功能。

1. SearchPanel的创建

与导航栏一样，SearchPanel通过WindowManager.addView () 的方式创建其窗口。

在PhoneStatusBar.addNavigationBar () 方法使用mNavigationBarView通过WindowManager.addView () 创建导航栏窗口之前，PhoneStatusBar会首先调用PhoneStatusBar.prepareNavigationBar () 。而prepareNavigationBar () 会调用PhoneStatusBar.updateSearchPanel () 进行SearchPanel的创建工作。

[PhoneStatusBar.java-->PhoneStatusBar.updateSearchPanel()]
protected void updateSearchPanel() /*由BaseStatusBar.updateSearchPanel()进行
SearchPanel的创建工作。 创建好的SearchPanel的控件树会被保存在
mSearchPanelView成员中。 SearchPanel并不是小屏幕设备独有的特
性，因此它的创建位于BaseStatusBar */super.updateSearchPanel();// 将导

航栏与SearchPanel关联起来，以便二者可以互相访问

```
mSearchPanelView.setStatusBarView(mNavigationBarView);mNavigationB  
arView.setDelegateView(mSearchPanelView); }
```

于是需要进入BaseStatusBar.updateSearchPenl () 对SearchPanel的创建情况进行讨论。

[BaseStatusBar.java-->BaseStatusBar.updateSearchPanel()] protected void updateSearchPanel() {...../* ① 从R.layout.status_bar_search_panel中创建 SearchPanel的根控件。保存在mSearchPanelView 成员变量之中 */
mSearchPanelView = (SearchPanelView)
LayoutInflater.from(mContext).inflate(R.layout.status_bar_search_panel,
tmpRoot, false /* will not add to tmpRoot */);.....// SearchPanel默认情况下是不可见的mSearchPanelView.setVisibility(View.GONE); // ② 创建用于 SearchPanel的LayoutParams WindowManager.LayoutParams lp =
getSearchLayoutParams(mSearchPanelView.getLayoutParams());/* ③ 创建 SearchPanel的窗口。不过虽然被创建，但是由于其visibility为GONE，因此SearchPanel 仍然是不可见的
*/mWindowManager.addView(mSearchPanelView, lp);..... }

那么R.layout.status_bar_search_panel描述了什么样的一棵控件树呢？参考res/layout/status_bar_search_panel.xml的内容，可以发现其根控件是一个类型为SearchPanelView的控件，并且在嵌套几个ViewGroup之后，有

一个类型为GlowPadView的控件。SearchPanelView以及GlowPadView是SearchPanel中最重要的两个组件。前者作为SearchPanel控件树的根控件为NavigationBarView以及BaseStatusBar提供操作SearchPanel的接口，而后者则是用户在启动SearchPanel之后所看到的东西。GlowPadView在SearchPanel窗口的底部渲染了一个半圆形的区域并在其上绘制了几个图标，当用户的手指滑动到其中某个图标之上并松手时，GlowPadView会向SearchPanelView引发一个回调，SearchPanelView会根据图标的id执行特定的动作。在默认情况下，GlowPadView之上仅有一个放大镜模样的图标，当用户手指滑动其上时，SearchPanelView会启动搜索界面。

getSearchLayoutParams () 方法为SearchPanel所创建的布局参数规定了窗口的类型是TYPE_NAVIGATION_BAR_PANEL。这个类型的窗口将拥有比导航栏更高的layer值（20），因此SearchPanel将会显示在导航栏之上。另外SearchPanel的尺寸与导航栏一样是MATCH_PARENT，但是由于PhoneWindowManager并不会对TYPE_NAVIGATION_BAR_PANEL类型的窗口做特殊的布局计算，因此SearchPanel的窗口其实是充满整个屏幕的，只不过除了GlowPadView以外的区域都是透明的而已。

2. SearchPanel的启动

SearchPanel的启动触发于用户在导航栏上的滑动操作，所以需要到NavigationBarView对触摸事件的处理中寻找SearchPanel的触发方法。

参考NavigationBarView.onInterceptTouchEvent () 的实现：

[NavigationBarView.java-->NavigationBarView.onInterceptTouchEvent()]

```
public boolean onInterceptTouchEvent(MotionEvent event) { return  
mDelegateHelper.onInterceptTouchEvent(event); }
```

onInterceptTouchEvent () 方法会在触摸事件被派发给子控件之前首先获得对事件进行处理的机会，可以用来识别当前ViewGroup所感兴趣的手势。如果onInterceptTouchEvent () 方法的实现者识别出手势而返回 true，那么表示此ViewGroup决定独自消费这一事件序列，于是事件序列中后续的触摸事件将会转移到当前ViewGroup.onTouchEvent () 中进行处理，并且ViewGroup的子控件将不再拥有对此事件序列的处理权。

NavigationBarView会将其截获的触摸事件发送给mDelegateHelper——一个辅助类DelegateViewHelper的实例。而DeletegetViewHelper会通过触摸事件分析用户的手势以启动SearchPanel。

接下来参考DelegateViewHelper的onInterceptTouchEvent () 方法的实现：

[DelegateViewHelper.java-->DelegateViewHelper.onInterceptTouchEvent()]

```
public boolean onInterceptTouchEvent(MotionEvent event) { .....if  
(!mPanelShowing && event.getAction() ==  
MotionEvent.ACTION_MOVE) { for (int k = 0; k < historySize + 1; k++) {
```

```
..... /* ① 计算用户手指滑动的距离，倘若这一距离大于  
mTriggerThreshhold，则通过调用mBar.showSearchPanel()方法显示  
SearchPanel */ if (distance > mTriggerThreshhold) {  
    mBar.showSearchPanel(); mPanelShowing = true; // 标记SearchPanel已经  
    启动 break; } } // 这部分省略的代码用于将触摸事件的坐标系从  
NavigationBarView转换到SearchPanelView中...../* ② 将导航栏所收到的  
触摸事件通过dispatchTouchEvent()注入SearchPanelView中。这里的  
mDe- legateView就是SearchPanelView。SearchPanelView会对触摸事件  
做进一步处理，例如将事件派发给GlowPadView绘制动画效果，以及  
根据用户的手势触发特定的功能  
*/mDelegateView.dispatchTouchEvent(event); // 这部分省略的代码用于将  
触摸事件的坐标系从SearchPanelView转换回NavigationBarView...../* ③  
当SearchPanel启动后，触摸事件序列中后续的事件需要被发送给  
SearchPanel而不是Navigation- BarView中的子控件。此时需要返回  
true，以便从子控件手中夺取事件的处理权 */return mPanelShowing;}
```



注意

DelegateViewHelper.onInterceptTouchEvent () 方法中的参数event其实是InputDispatcher派发给导航栏窗口的触摸事件，但是它却通过View.dispatchTouchEvent () 方法传递给SearchPanelView这个位于另一个窗口的控件。正如第5章中所说，当ACTION_DOWN被派发给一个窗口之后，其随后的事件序列都会被派发到这个窗口。这种默认的行为使得触摸事件可以让NavigationBarView启动SearchPanel，但是却无法被SearchPanel消费。如此一来，用户需要在SearchPanel窗口显示后抬起手指结束当前的事件序列，再重新按下手指使得新的事件序列被派发给SearchPanel，不过这无疑会严重影响用户的体验。因此DelegateViewHelper存在的意义就很明显了，它从NavigationBarView中截获事件，并将截获的事件通过View.dispatchTouchEvent () 发送给SearchPanel进行消费，使得NavigationBarView以及SearchPanelView这两个属于不同窗口的控件树得以共享同一个事件序列。这是操作输入事件的一个很聪明也很有用的技巧。

3.启动搜索界面

SearchPanelView中的GlowPadView是消费触摸事件的场所。它在屏幕底部的一个半圆形轨迹上显示一系列图标，并根据触摸事件检测用户手指的移动轨迹。当用户抬起手指后，GlowPadView会通过调用OnTriggerListener.onTrigger () 回调将用户通过滑动手指所选取的图标告知对此行为感兴趣的监听者。GlowPadView上的图标是否可定制呢？

答案是肯定的。在通过XML声明GlowPadView的布局时（也就是res/layout/status_bar_search_panel.xml），可以通过为GlowPadView：targetDrawables属性指定一个Drawable的数组来定制出现在GlowPadView中的图标内容与个数。默认情况下这个数组中只含有@android : drawable/ic_action_assist_generic一个图标资源，因此用户在启动SearchPanel后只能看到一个放大镜模样的图标。开发者可以通过扩充这一数组的内容以定制SearchPanel中的图标。

SearchPanelView提供了实现接口OnTriggerListener.onTrigger () 的GlowPadTriggerLi-stener。因此可以在GlowPadTriggerListener.onTrigger () 的实现中了解搜索界面是如何被启动的。

[SearchPanelView.java-->GlowPadTriggerListener.onTrigger()] public void onTrigger(View v, final int target) {/* ① onTrigger()中的参数target表示用户选定的图标在数组中的索引。不过GlowPadTriggerListener根本不知道这一索引的含义是什么。因此，它通过GlowPadView.getResourceIdForTarget()方法获取此索引上的图标资源id，于是便可以根据不同的资源id做不同的事情 */final int resId = mGlowPadView.getResourceIdForTarget(target);switch (resId) { case com.android.internal.R.drawable.ic_action_assist_generic: /* ② ic_action_assist_generic图标被用户选中，因此启动搜索界面

```
mWaitingForLaunch = true; startAssistActivity(); vibrate(); // 震动一下  
break; } }
```

同理，开发者可以在上述switch语句块中增加对新图标资源的处理。

关于SearchPanel的内容便介绍到这里。经过本节的讨论，希望读者能够领会通过View的事件派发方法（本例是View.dispatchTouchEvent()）使多个窗口的控件共享一条事件序列的技巧，以及如何扩展SearchPanel的功能以利用这个最快捷的功能入口。

7.3.4 关于导航栏的其他话题

除了虚拟按键以及SearchPanel之外，导航栏中还有一些值得讨论的细节问题。

1. 菜单键的可见性

一般情况下，用户在导航栏中所见到的虚拟按键只有三个：BACK键、HOME键以及RECENT键。MENU键仅在那些提供了选项菜单的应用运行的时候才能见到。为什么呢？

其实，用于呼出选项菜单的菜单键（无论是物理按键还是导航栏中的虚拟按键）从Android 4.0之后都不再鼓励使用。ActionBar即动作条成为菜单键的替代品，用于为用户提供更好的用户体验。不过为了保持对旧版本的兼容性，导航栏中还是提供了对菜单键的支持。

当PhoneWindow为创建控件树的模板时，即
PhoneWindow.generateLayout () 方法中会根据应用程序所基于的
Android版本选择是否应当为此应用程序提供菜单键的支持。参考其代
码：

```
[PhoneWindow.java-->PhoneWindow.generateLayout()]
protected
ViewGroup generateLayout(DecorView decor) {if (targetPreHoneycomb ||
(targetPreIcs && targetHcNeedsOptions && noActionBar)) { /* 倘若应用
程序所基于的Android版本比较老，则为其窗口的LayoutParams.flags添
加FLAG_NEEDS_MENU_KEY标记，以启用对菜单键的支持 */
addFlags(WindowManager.LayoutParams.FLAG_NEEDS_MENU_KEY);}
else { // 否则确保FLAG_NEEDS_MENU_KEY不存在于
LayoutParams.flags中
clearFlags(WindowManager.LayoutParams.FLAG_NEEDS_MENU_KEY);
} }
```

因此，窗口的LayoutParams.flags中是否存在
FLAG_NEEDS_MENU_KEY是决定导航栏中是否显示菜单键的根本原
因，不过这一标记是如何影响SystemUI的导航栏呢？菜单键既然是一个按键，那么它所产生的事件一定会发送给处于焦点状态的窗口。因此焦点状态的窗口是否存在FLAG_NEEDS_MENU_KEY便成为关键所在。当WMS完成焦点窗口的选择之后（参考第5章），会通过

PhoneWindowManager.focusChangedLw () 方法将新的焦点窗口告知 PhoneWindowManager。在此方法中，PhoneWindowManager通过 updateSystemUIVisibilityLw () 方法将焦点窗口上与SystemUI相关的设置发送给SystemUI。这些设置其中之一就是 FLAG_NEEDS_MENU_KEY标记。参考updateSystemUIVisibilityLw () 方法的代码：

[PhoneWindowManager.java--

```
>PhoneWindowManager.updateSystemUIVisibilityLw() private int  
updateSystemUIVisibilityLw() {...../* ① 检查是否需要菜单键。  
mFocusedWindow是当前的焦点窗口。而getNeedsMenuLw()对  
FLAG_NEEDS_MENU_KEY标记的 检查并不仅限于焦点窗口，它还会  
检查位于焦点窗口之下，处于mTopFullscreenOpaqueWindowState  
(含) 之上的窗口。只要这些窗口其中之一含有此标记，则表示需要  
在导航栏中显示菜单键 */final boolean needsMenu =  
mFocusedWindow.getNeedsMenuLw(mTopFullscreenOpaqueWindowState)  
;.....mHandler.post(new Runnable() { public void run() { try {  
IStatusBarService statusbar = getStatusBarService(); if (statusbar != null) {  
/* ② 通过StatusBarManagerService.topAppWindowChanged()方法将是否  
需要菜单键通知给SystemUI里的导航栏 */  
statusbar.topAppWindowChanged(needsMenu); } } catch (RemoteException  
e) {...} } }); }
```

既然设置菜单键可见性使用了熟悉的StatusBarManagerService，那么最终处理这一请求的一定是BaseStatusBar或其子类。参考PhoneStatusBar.topAppWindowChanged () 的实现：

```
[PhoneStatusBar.java-->PhoneStatusBar.topAppWindowChanged()] public void topAppWindowChanged(boolean showMenu) { // 如果导航栏存在，则通过setMenuVisibility()设置菜单键的可见性 if (mNavigationBarView != null) { mNavigationBarView.setMenuVisibility(showMenu); }..... }
```

至于NavigationBarView.setMenuVisibility () 的实现，请参考：

```
[NavigationBarView.java-->NavigationBarView.setMenuVisibility()] public void setMenuVisibility(final boolean show, final boolean force) {if (!force && mShowMenu == show) return;mShowMenu = show;// 设置菜单键的可见性getMenuButton().setVisibility(mShowMenu ? View.VISIBLE : View.INVISIBLE); }
```

综上所述，是否为一个窗口显示菜单键取决于PhoneWindow中generateLayout () 方法是否为窗口添加FLAG_NEEDS_MENU_KEY，它仅对基于Android 4.0之前版本的应用程序添加此标记。当Phone WindowManager发现焦点窗口发生变化之后，倘若从焦点窗口到第一个全屏不透明窗口（mTopFullscreenOpaqueWindowState）之间的所有窗口中，至少有一个窗口指定了FLAG_NEEDS_MENU_KEY，则

会决定显示菜单键，否则隐藏菜单键。PhoneWindowManager会通过 StatusBarManagerService.topAppWindowChanged () 方法对导航栏中菜单键的可见性进行设置。最终NavigationBarView通过View.setVisibility () 方法将菜单键所对应的KeyButtonView进行显示或隐藏操作。

2.修改BACK键的图标

细心的读者应该能够发现，正常情况下导航栏上的BACK键是一个向左的箭头。不过一旦弹出了输入法窗口，BACK键则变为了一个向下的箭头，用以指示当用户按下BACK键后将退出输入法而不是当前应用程序。其实这个行为就导航栏而言，它仅仅修改了KeyButtonView所显示的图标而已。



说明

BACK键变为向下的箭头主要是为了引发用户对输入法窗口退出时所使用的向下滑出屏幕动画的联想，因此不要尝试修改输入法退出时的这种动画形式。

当一个文本框使用InputMethodManager.showSoftInput () 尝试弹出输入法窗口时，这一请求经由InputMethodManagerService转发给实现输入法的InputMethodService，并由后者着手创建并显示输入法的窗口。

[InputMethodService.java-->InputMethodImpl.showSoftInput()] public void showSoftInput(int flags, ResultReceiver resultReceiver) {..... // 显示输入法窗口的代码boolean showing = onEvaluateInputViewShown();/* 将输入法窗口的显示状态通知给InputMethodManagerService。注意，当窗口处于显示状态时，此方法 的第二个参数包含IME_VISIBLE标记 */mImm.setImeWindowStatus(mToken, IME_ACTIVE | (showing ? IME_VISIBLE : 0), mBackDisposition);..... }

随后，由setImeWindowStatus () 所设置的输入法窗口状态辗转经过InputMethodManagerService、 StatusBarManagerService来到PhoneStatusBar：

[PhoneStatusBar.java-->PhoneStatusBar.setImeWindowStatus()] public void setImeWindowStatus(IBinder token, int vis, int backDisposition) {// altBack 决定是否修改Back键的图标。其中一个充分条件就是第二个参数中存在IME_VISIBLE标记boolean altBack = (backDisposition == InputMethodService.BACK_DISPOSITION_WILL_DISMISS) || ((vis & InputMethodService.IME_VISIBLE) != 0);/* 通过mCommandQueue.setNavigationIconHints()方法修改BACK键的图标。*/}

注意，倘若altBack为true，即需要修改BACK键的图标时，

NAVIGATION_HINT_BACK_ALT标记存在于参数之中

```
*/mCommandQueue.setNavigationIconHints( altBack ?  
(mNavigationIconHints |  
StatusBarManager.NAVIGATION_HINT_BACK_ALT) :  
(mNavigationIconHints&  
~StatusBarManager.NAVIGATION_HINT_BACK_ALT));..... }
```

NavigationBarView.setNavigationIconHints () 方法用于对导航栏上的虚拟按键的图标进行微调。其参数hint可以是下列标记的组合：

- NAVIGATION_HINT_BACK_NOP，表示此时的BACK键无意义。参数中含有此标记时，NavigationBarView将通过调整BACK键的透明度使其可见性降低。

- NAVIGATION_HINT_HOME_NOP，表示此时的HOME键无意义。参数中含有此标记时，NavigationBarView将通过调整HOME键的透明度使其可见性降低。

- NAVIGATION_HINT_RECENT_NOP，表示此时的RECENT键无意义。参数中含有此标记时，NavigationBarView将通过调整RECENT键的透明度使其可见性降低。

- NAVIGATION_HINT_BACK_ALT，修改BACK键的图标。

那么这些标记是如何生效的呢？请参考

NavigationBarView.setNavigationIconHints () 的实现：

[NavigationBarView.java-->NavigationBarView.setNavigationIconHints()]

```
public void setNavigationIconHints(int hints, boolean force) {.....// 保存  
hints  
mNavigationIconHints = hints;// 根据三个_NOP标记设置三个虚拟按  
键的透明度getBackButton().setAlpha( (0 != (hints &  
StatusBarManager.NAVIGATION_HINT_BACK_NOP)) ? 0.5f :  
1.0f);getHomeButton().setAlpha( (0 != (hints &  
StatusBarManager.NAVIGATION_HINT_HOME_NOP)) ? 0.5f :  
1.0f);getRecentsButton().setAlpha( (0 != (hints &  
StatusBarManager.NAVIGATION_HINT_RECENT_NOP)) ? 0.5f : 1.0f);/*  
倘若hints参数中存在NAVIGATION_HINT_BACK_ALT，则使用  
mBackAltIcon或mBackAltLandIcon 作为BACK键的图标（下箭头），否  
则使用mBackIcon或mBackLandIcon（左箭头）  
*/((ImageView)getBackButton()).setImageDrawable( (0 != (hints &  
StatusBarManager.NAVIGATION_HINT_BACK_ALT)) ? (mVertical ?  
mBackAltLandIcon : mBackAltIcon) : (mVertical ? mBackLandIcon :  
mBackIcon));..... }
```

可见，这些标记并不会实质性地改变虚拟按键的行为，仅调整了它们的显示方式而已。

另外，目前并没有哪个系统组件需要对导航栏设置_NOP结尾的三个标记。因为当某个虚拟按键没有意义时（例如当不响应BACK键的应用程序处于运行状态时），最好的办法是将其完全隐藏，而不是仅仅让其可见性降低。这就需要用到SystemUI的另外一个机制DisableActions。我们将在7.4节介绍。

3. 导航栏方向的选择

如前面所述，导航栏的根控件NavigationBarView之下有两套不同方向的控件树@id/rot0和@id/rot90。当导航栏处于屏幕底端时的布局为@id/rot0，而处于屏幕右端时的布局为@id/rot90。导航栏是如何在这两者之间进行切换的呢？常规的思路是通过监听ACTION_CONFIGURATION_CHANGED广播，并根据广播到来时所携带的屏幕方向信息对这两棵控件树进行选择。不过，鉴于PhoneWindowManager对导航栏的特殊布局计算方式，导航栏选择了一个更聪明的做法。

每当屏幕方向发生旋转时， WindowManagerService会对所有窗口进行重新布局，进而使得PhoneWindowManager.beginLayoutLw () 被调用。在这个方法中，PhoneWindowManager根据屏幕在当前方向下的宽高信息选择将导航栏的窗口置于屏幕底端或者右端。导航栏窗口的位置发生变化时，其尺寸同样会发生变化（底端时为宽大于高，而右端是为高大于宽）。导航栏窗口尺寸的变化会引发ViewRootImpl的“遍

历”操作，进而使得导航栏的控件树进行重新布局。而控件树重新布局的结果是NavigationBarView控件的尺寸发生变化，于是NavigationBarView重写了View.onSizeChanged () 方法并在这里完成@id/rot0与@id/rot90的选择。参考其代码：

```
[NavigationBarView.java-->NavigationBarView.onSizeChanged()] protected  
void onSizeChanged(int w, int h, int oldw, int oldh) { // 检查  
NavigationBarView处于竖直状态还是水平状态final boolean newVertical  
= w > 0 && h > w; // 当竖直或水平状态发生变化时if (newVertical !=  
mVertical) { mVertical = newVertical; // reorient()会在@id/rot0、@id/rot90  
与@id/rot90之间做出选择 reorient(); } super.onSizeChanged(w, h, oldw,  
oldh); }
```

再看reorient () 方法的实现：

```
[NavigationBarView.java-->NavigationBarView.reorient()] public void  
reorient() { // 首先获取屏幕的旋转角度：0、90、180或270final int rot =  
mDisplay.getRotation(); /* mRotatedViews数组存储了用于特定方向的导  
航栏控件树。进行控件树的选择之前，首先将所有控件树的Visibility  
设置为GONE */ for (int i=0; i<4; i++) {  
mRotatedViews[i].setVisibility(View.GONE); } // ① 选择对应当前旋转方向  
的控件树作为mCurrentView，并将其设置为可见mCurrentView =  
mRotatedViews[rot]; mCurrentView.setVisibility(View.VISIBLE); // 更新
```

```
mDeadZone使之引用当前控件树中的Deadzone  
mDeadZone = (DeadZone)  
mCurrentView.findViewById(R.id.deadzone); // ② 为当前控件树同步各种  
状态// 低辨识度状态setLowProfile(mLowProfile, false, true /* force */); //  
禁用功能setDisabledFlags(mDisabledFlags, true /* force */); // 菜单键的可  
见性setMenuVisibility(mShowMenu, true /* force */); // 修改虚拟按键的图  
标setNavigationIconHints(mNavigationIconHints, true); }
```

其中， mRotatedViews数组的初始化位于

NavigationBarView.onFinishInflate () 方法。此方法在LayoutInflater完
成从layout中创建控件树之后被调用，因此初始化此数组的时机在
PhoneStatusBar.makeStatusBar () 方法。参考其实现：

```
[NavigationBarView.java-->NavigationBarView.onFinishInflate()] public  
void onFinishInflate() { // 0度和180度（竖直状态）使用  
@id/rot0mRotatedViews[Surface.ROTATION_0] =  
mRotatedViews[Surface.ROTATION_180] = findViewById(R.id.rot0); // 90  
度（水平状态）使用@id/rot90mRotatedViews[Surface.ROTATION_90] =  
findViewById(R.id.rot90); // NAVBAR_ALWAYS_AT_RIGHT目前是  
true，因此270度的水平状态下仍然使用  
@id/rot90mRotatedViews[Surface.ROTATION_270] =  
NAVBAR_ALWAYS_AT_RIGHT ? findViewById(R.id.rot90) :
```

```
findViewById(R.id.rot270);mCurrentView =  
mRotatedViews[Surface.ROTATION_0]; }
```

简单来说，导航栏选择不同控件树的时机在onSizeChanged () 方法中，然后reorient () 通过mDisplay.getRotation () 获取屏幕方向，进而 在mRotatedViews数组中选择@id/rot0和@id/rot90两棵控件树之一进行显示。

7.3.5 导航栏总结

对导航栏的介绍至此告一段落。总结一下本节的重点内容：

- 导航栏的核心工作是将用户的触摸事件转换为相应的按键事件，并发送给InputDispatcher。而这一功能的主要实现者是导航栏内的KeyButtonView。
- PhoneWindowManager会使用特殊的形式对导航栏的窗口进行布局。屏幕处于竖直方向时的导航栏会被布局在屏幕的底端，而水平方向的导航栏会被布局在屏幕的右端。在这两种布局下，导航栏会分别选择@id/rot0以及@id/rot90作为其控件树进行显示。
- 导航栏还提供了setMenuVisibility () 以及setNavigationIconHints () ，以允许PhoneStatusBar对导航栏的行文进行微调。

另外，导航栏还支持进入低辨识度模式，以及可以禁用一些功能。这些内容涉及SystemUIVisibility以及DisableAction两个话题，它们将在下一节讨论。

7.4 禁用状态栏与导航栏的功能

在某些情况下，Android需要禁用状态栏或导航栏中的某些功能。例如在锁屏状态下，用户点击HOME键、BACK键以及RECENT键是无意义的，因此最好将这些虚拟按键隐藏掉。同样在锁屏状态下，尤其是用户通过图案或密码进行解锁保护的时候，Android不希望用户可以拉出下拉卷帘而导致那些如短信等私人信息遭到暴露。

7.4.1 如何禁用状态栏与导航栏的功能

为此StatusBarManager提供了disable () 方法使得开发者可以禁用状态栏或导航栏的一些功能。

```
[StatusBarManager.java-->StatusBarManager.disable()]
public void disable(int what) {try { final IStatusBarService svc = getService(); if (svc != null) { // 向StatusBarManagerService发送禁用某些功能的请求 svc.disable(what, mToken, mContext.getPackageName()); } } catch (RemoteException ex) {.....} }
```

disable () 方法的参数what可以是如下禁用标记中的一个或多个的组合：

- DISABLE_EXPAND，禁止用户拉出下拉卷帘。

- DISABLE_NOTIFICATION_ICONS，隐藏状态栏中的通知图标。
- DISABLE_NOTIFICATION_ALERTS，禁用通知声音。
- DISABLE_NOTIFICATION_TICKER，禁用Ticker。
- DISABLE_SYSTEM_INFO，隐藏状态栏中系统状态图标区、信号信息、电量信息以及系统时钟。
- DISABLE_HOME，隐藏HOME键。
- DISABLE_RECENT，隐藏RECENT键。
- DISABLE_BACK，隐藏BACK键。
- DISABLE_SEARCH，禁止启动SearchPanel。
- DISABLE_CLOCK，隐藏时钟。

另外值得一提的是mToken参数。mToken是随着StatusBarManager一起被创建的一个Binder实例，可以理解为它是StatusBarManager的一个可以跨进程传递的ID。那么它存在的意义是什么呢？以一个合理的逻辑来看，既然禁用某一个功能，那么一定要在之后的某一个时机恢复这一功能。倘若一个应用程序通过StatusBarManager.disable () 禁用了某一个功能之后因为某种原因意外退出，那么这一被禁用的功能有可能永远无法恢复。为了避免这个问题，StatusBarManagerService要求

StatusBarManager必须提供一个Binder实例，以便StatusBarManager所在的进程意外退出之后StatusBarManagerService可以通过DeathRecipient机制得到通知，并恢复这一进程所禁用的功能。



注意

禁用状态栏或导航栏的功能需要StatusBarManager.disable () 方法的调用者拥有签名级的系统权限android.permission.STATUS_BAR。

7.4.2 StatusBarManagerService对禁用标记的维护

在StatusBarManagerService中，接受StatusBarManager.disable () 请求的方法是StatusBarManagerService.disableLocked () 。

[StatusBarManagerService.java--

```
>StatusBarManagerService.disableLocked()]
```

```
private void disableLocked(int userId, int what, IBinder token, String pkg) {/* ① 在  
manageDisableListLocked()里，userId、what、token、pkg等信息会被封  
装为一个DisableRecord实例，然后保存在mDisableRecords列表中
```

```
*/manageDisableListLocked(userId, what, token, pkg);/* ②  
gatherDisableActionsLocked()会在mDisableRecords中遍历所有  
DisableRecord实例，然后收集它们所保存的禁用标记并作为返回值返  
回给net */final int net = gatherDisableActionsLocked(userId);/* ③ 最后把  
收集到的禁用标记发送给NotificationManagerService以及SystemUI中的  
状态栏与导航栏 */if (net != mDisabled) { mDisabled = net; /* 把禁用标  
记设置给NotificationManagerService。 NotificationManagerService所关心  
的禁用标记仅有DISABLE_NOTIFICATION_ALERT一个，用来在添加  
通知之时跳过通知音的播放或震动 */ mHandler.post(new Runnable() {  
    public void run() { mNotificationCallbacks.onSetDisabled(net); } }); // 把禁  
用标记通过mBar设置给SystemUI中的状态栏与导航栏 if (mBar != null)  
{ try { mBar.disable(net); } catch (RemoteException ex) { } } }
```

作为一个系统服务，StatusBarManagerService可以通过StatusBarManager为多个进程提供disable () 方法的调用。为了避免不同进程所设置的禁用标记发生冲突（如进程A所设置的禁用标记把进程B所设置的禁用标记替换掉），StatusBarManagerService为每一个需要禁用功能的进程维护了一个DisableRecord并保存在mDisableRecords中。于是StatusBarManagerService可以通过遍历所有的DisableRecord来收集每个进程所提供的禁用标记，并以收集的结果设置给NotificationManagerService以及SystemUI中的状态栏与导航栏，以保证满足每个进程对禁用功能的需求。

7.4.3 状态栏与导航栏对禁用标记的响应

BaseStatusBar并没有提供disable () 方法的实现，这是由子类完成的。以PhoneStatus-Bar的实现为例，状态栏与导航栏对禁用标记的响应方式主要是设置特定控件的可见性。PhoneStatusBar.disable () 的实现十分清晰简单，这里直接列出此方法对不同禁用标记的响应行为，如下所示：

- DISABLE_SYSTEM_INFO，使mSystemIconArea以动画的方式将透明度设置为0，使之变得不可见。mSystemIconArea是系统状态图标区、信号信息、电量信息以及系统时钟的父控件。
- DISABLE_CLOCK，设置控件@id/clock的可见性为GONE。@id/clock是维护系统时钟的控件Clock。
- DISABLE_EXPAND，立刻收起下拉卷帘，并且禁止PhoneStatusBarView.onTouch-Event () 对用户手势的识别。
- DISABLE_HOME、DISABLE_BACK、DISABLE_RECENT由NavigationBar-View进行处理。NavigationBarView会将对应的KeyButtonView的可见性设置为INVISIBLE。
- DISABLE_SEARCH由NavigationBarView进行处理。DISABLE_SEARCH禁用标记的存在会在

DelegateViewHelper.onInterceptTouchEvent () 方法中阻止 DelegateViewHelper 对触摸事件进行处理。进而使得 SearchPanel 因为无法启动而被禁用。尤其是，倘若 DISABLE_HOME 标记存在而 DISABLE_SEARCH 标记不存在，则 Navigation-BarView 中的 @id/search_light 控件将会显示出来，以提示用户虽然 HOME 键不可用，但仍然可以通过在导航栏上滑动手指启动 SearchPanel。

· DISABLE_NOTIFICATION_ICONS，使 mNotificationIcons 以动画的方式将透明度设置为 0，使之变得不可见。mNotificationIcons 是容纳通知图标的容器。

· DISABLE_NOTIFICATION_TICKER，立刻终止正在进行的 ticking 动作，并在 Phone-StatusBar.tick () 方法中阻止 tick。这样一来通知的内容将不会出现在状态栏之上。

· DISABLE_NOTIFICATION_ALERT，这个禁用标记并没有在 PhoneStatusBar 中进行处理。对这个禁用标记感兴趣的是 NotificationManagerService，用来在添加通知之时跳过通知音的播放与震动。

其实，查看这些禁用标记在 StatusBarManager 中的定义，可以发现它们一一对应于定义在 View 类中的以 STATUS_BAR_ 为前缀的标记。这些 STATUS_BAR_ 标记属于另外一个话题—— SystemUIVisibility。或者

说，除了通过StatusBarManager.disable () 方法之外，
SystemUIVisibility是另外一种禁用状态栏与导航栏功能的方式。

7.5 理解SystemUIVisibility

尽管StatusBarManager以及StatusBarManagerService为应用程序以及系统服务提供了操作状态栏与导航栏的所有接口，但是这些接口并不适用于那些没有系统签名的普通应用程序。倘若一个普通的应用程序希望对状态栏以及导航栏进行操作，就必须使用SystemUIVisibility机制。

SystemUIVisibility与禁用标记一样，是一个int型的变量，其中可以按位容纳多个定义在View类中用于调整状态栏与导航栏行为的标记。其实SystemUIVisibility可容纳的标记的丰富性远远不止控制SystemUI的可见性而已。按照这些标记所产生的影响，它们可以分为以下三类。

影响状态栏与导航栏可见性的标记：

- SYSTEM_UI_FLAG_LOW_PROFILE，使得状态栏与导航栏进入低辨识度模式。
- SYSTEM_UI_FLAG_HIDE_NAVIGATION，隐藏导航栏。窗口在布局时的ParentFrame、ContentFrame以及DisplayFrame都会延展到整个屏幕。
- SYSTEM_UI_FLAG_FULLSCREEN，隐藏状态栏。窗口在布局时的ParentFrame、ContentFrame以及DisplayFrame都会延展到屏幕的顶部。



注意

SYSMTE_UI_FLAG_FULLSCREEN并不是同时隐藏状态栏与导航栏。

影响窗口布局结果的标记：

- SYSTEM_UI_FLAG_LAYOUT_STABLE，声明这个标记的窗口的ContentFrame不会随着导航栏或状态栏的显示或隐藏而发生变化。
- SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION，PhoneWindowManager在对声明了这个标记的窗口进行布局时会认为导航栏已经隐藏，无论导航栏是否真的被隐藏。
- SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN，PhoneWindowManager在对声明了这个标记的窗口进行布局时会认为导航栏和状态栏都已经隐藏，无论状态栏和导航栏是否真的被隐藏。

用于禁用状态栏与导航栏的某些功能的标记：

- STATUS_BAR_DISABLE_EXPAND

- STATUS_BAR_DISABLE_NOTIFICATION_ICONS
- STATUS_BAR_DISABLE_NOTIFICATION_ALERTS
- STATUS_BAR_DISABLE_NOTIFICATION_TICKER
- STATUS_BAR_DISABLE_SYSTEM_INFO
- STATUS_BAR_DISABLE_HOME
- STATUS_BAR_DISABLE_BACK
- STATUS_BAR_DISABLE_CLOCK
- STATUS_BAR_DISABLE_RECENT
- STATUS_BAR_DISABLE_SEARCH



说明

这些以STATUS_BAR_DISABLE_为前缀的标记的功能与在StatusBarManager中所定义的禁用标记一一对应并且功能相同，并且可以通过systemUIVisibility&StatusBarManager.DISABLE_MASK这种方式

式将它们从SystemUIVisibility中提取出来，然后作为StatusBarManager.disable () 方法的参数设置给状态栏与导航栏。不过目前Android中并没有做出这种行为的代码。因此，通过SystemUIVisibility的方式进行状态栏与导航栏功能的禁用目前是行不通的。

设置SystemUIVisibility的方式有两种。一是在任意一个已经显示在窗口上的控件调用View.setSystemUiVisibility () ，二是直接在窗口的LayoutParams.systemUiVisibility上进行设置并通过WindowManager.updateViewLayout () 方法使其生效。

7.5.1 SystemUIVisibility在系统中的漫游过程

从可用的标记列表可以看出SystemUIVisibility主要涉及状态栏和导航栏的行为以及窗口布局两个方面，因此SystemUIVisibility的消费者有SystemUI中的BaseStatusBar，以及负责窗口布局的PhoneWindowManager。本节将从View.setSystemUiVisibility () 开始追寻SystemUIVisibility从控件到BaseStatusBar以及PhoneWindowManager的过程。

1. 控件树中的SystemUIVisibility

参考View.setSystemUiVisibility () 的实现：

```
[View.java-->View.setSystemUIVisibility()] public void  
setSystemUiVisibility(int visibility) {if (visibility != mSystemUiVisibility) {  
// ① 首先， SystemUIVisibility会保存在View自己的成员变量  
mSystemUiVisibility中 mSystemUiVisibility = visibility; // ② 通知父控件  
这一变化 if (mParent != null && mAttachInfo != null &&  
!mAttachInfo.mRecomputeGlobalAttributes) {  
mParent.recomputeViewAttributes(this); } } }
```

recomputeViewAttributes () 会沿着控件树向根部回溯，最终
ViewRootImpl的recompute-ViewAttributes () 会对此做出如下处理：

```
[ViewRootImpl.java-->ViewRootImpl.recomputeViewAttributes()] public  
void recomputeViewAttributes(View child) {checkThread(); // 必须在窗口  
的主线程中调用if (mView == child) { // ① 标记随后需要处理  
SystemUIVisibility的变化 mAttachInfo.mRecomputeGlobalAttributes =  
true; if (!mWillDrawSoon) { // ② 触发一次“遍历”动作  
scheduleTraversals(); } } }
```

在ViewRootImpl的“遍历”过程中会执行
ViewRootImpl.collectViewAttributes () 方法收集控件树中每个View所
保存的SystemUIVisibility。其实现如下：

```
[ViewRootImpl.java-->ViewRootImpl.collectViewAttributes()] private  
boolean collectViewAttributes() {final View.AttachInfo attachInfo =  
mAttachInfo;if (attachInfo.mRecomputeGlobalAttributes) { ..... // ① 清空  
AttachInfo中所保存的SystemUIVisibility attachInfo.mSystemUiVisibility =  
0; /* ② View.dispatchCollectViewAttributes()会遍历整个控件树。 并在这  
个过程中以按位或的方式将每个控件所存储的SystemUIVisibility累加到  
AttachInfo.mSys- temUiVisibility之上 */  
mView.dispatchCollectViewAttributes(attachInfo, 0); // 从收集到的  
SystemUIVisibility中移除被禁用的SystemUIVisibility标记  
attachInfo.mSystemUiVisibility &=  
~attachInfo.mDisabledSystemUiVisibility; WindowManager.LayoutParams  
params = mWindowAttributes; if (... ||attachInfo.mSystemUiVisibility !=  
params.subtreeSystemUiVisibility ....) { // ③ 将SystemUIVisibility保存到  
窗口的LayoutParams中 params.subtreeSystemUiVisibility =  
attachInfo.mSystemUiVisibility; ..... return true; } }return false; }
```

可见，在控件树中，每一个控件都在View.mSystemUiVisibility中保存了本控件所期望的SystemUIVisibility。当ViewRootImpl进行“遍历”操作时，会通过View.dispatchCollectViewAttributes () 方法将每个控件所期望的SystemUIVisibility收集到AttachInfo.mSystemUiVisibility成员中，并保存到当前窗口的LayoutParams.sutreeSystemUiVisibility作为本窗口所期望的SystemUIVisibility。当ViewRootImpl通过

WMS.relayoutWindow () 方法对窗口进行重新布局时，本窗口所期望的SystemUIVisibility会伴随着LayoutParams.subtree-SystemUiVisibility以及LayoutParams.systemUiVisibility两个字段进入WMS。



说明

在View.dispatchCollectViewAttributes () 收集所有控件所期望的SystemUIVisibility之后，会将AttachInfo.mDisabledSystemUiVisibility中所定义的标记从收集结果中移除。这是因为ActionBar（动作条）的存在所导致的。Android的UE行为规定，当ActionBar以Overlay的形式显示的时候，状态栏必须也处于显示状态。于是，在ActionBar显示时，会通过View.setDisabledSystemUiVisibility () 将SYSTEM_UI_FLAG_FULLSCREEN作为禁用的标记并保存在AttachInfo.mDisabledSystemUiVisibility中。进而在这里将SYSTEM_UI_FLAG_FULLSCREEN从收集到的SystemUiVisibility中移除。在Phone-WindowManager中也有类似的操作。

2.WMS中的SystemUIVisibility

既然一个窗口所期望的SystemUIVisibility会通过WMS.relayoutWindow

() 方法进入WMS，于是可以参考WMS.relayoutWindow () 中与 SystemUIVisibility相关的代码：

[WindowManagerService.java--

```
>WindowManagerService.relayoutWindow() public int  
relayoutWindow(Session session, IWindow client, int seq,  
WindowManagerLayoutParams attrs, .....){.....int systemUiVisibility = 0;if  
(attrs != null) { /* ① LayoutParams中的systemUiVisibility以及从控件树中  
收集到的subtreeSystem-UiVisibility会被整合在一起 */  
systemUiVisibility =  
(attrs.systemUiVisibility|attrs.subtreeSystemUiVisibility); /* 因为  
SystemUIVisibility中以STATUS_BAR_DISABLE_为前缀的标记其实是  
7.4节所介绍的禁用 标记，因此必须确保调用者拥有相应的权限。否则  
这些禁用标记会被移除 */ if ((systemUiVisibility &  
StatusBarManager.DISABLE_MASK) != 0) { if  
(mContext.checkSelfPermission(android.Manifest.permission.STATUS_BAR) !=  
PackageManager.PERMISSION_GRANTED) { systemUiVisibility &=  
~StatusBarManager.DISABLE_MASK; }  
}.....synchronized(mWindowMap) { // 获取执行relayout操作的窗口的  
WindowState win = windowForClientLocked(session, client,
```

```
false); ..... // ② 将窗口所期望的SystemUIVisibility保存到  
WindowState.mSystemUiVisibility中 if (attrs != null && seq == win.mSeq)  
{ win.mSystemUiVisibility = systemUiVisibility; } ..... }
```

所以在WMS中，每个窗口的WindowState在其成员变量mSystemUiVisibility中保存了各自期望的SystemUIVisibility。SystemUIVisibility消费者之一的PhoneWindowManager可以通过WindowState.getSystemUiVisibility () 获取这一信息并据此对窗口进行布局，或设置状态栏与导航栏的可见性。

不过SystemUiVisibility在系统中漫游的脚步仍然没有停止，因为除了PhoneWindow-Manager之外，SystemUI进程中的BaseStatusBar及其子类也是SystemUIVisibility的消费者。不过在WMS中并没有像ViewRootImpl一样将所有窗口所期望的SystemUIVisibility收集在一起，那么究竟哪个窗口的SystemUIVisibility才能传递给SystemUI中的BaseStatusBar呢？答案是焦点窗口。当焦点窗口发生变化时所调用的PhoneWindowManager.focusChangedLw ()（请参考第5章与窗口焦点相关的内容），以及WMS在布局过程所调用的PhoneWindowManager.finishPostLayoutLw ()（请参考第4章与窗口布局相关的内容）两个方法中会调用Phone-WindowManager.updateSystemUiVisibilityLw () 方法，将焦点窗口的

SystemUIVisibility传递给BaseStatusBar。另外，设置导航栏中菜单键可见性的也是这个方法。参考其实现：

[PhoneWindowManager.java--

```
>PhoneWindowManager.updateSystemUiVisibilityLw() private int  
updateSystemUiVisibilityLw() {...../* ① 从焦点窗口中获取它所期望的  
SystemUIVisibility。 PhoneWindowManager删除  
mResettingSystemUiFlags以及mForceClearedSystemUiFlags中所有的标  
记。其目的类似于AttachInfo.mDisabledSystemUiVisibility。有时候  
PhoneWindowManager不希望存在某些标记，这部分内容将在介绍  
SYSTEM_UI_FLAG_HIDE_NAVIGATION的处理时进行介绍 */final int  
visibility = mFocusedWindow.getSystemUiVisibility() &  
~mResettingSystemUiFlags & ~mForceClearedSystemUiFlags;// ② 过滤后  
的SystemUIVisibility被保存到  
PhoneWindowManager.mLastSystemUiFlagsmLastSystemUiFlags =  
visibility;.....mHandler.post(new Runnable() { public void run() { try {  
IStatusBarService statusbar = getStatusBarService(); if (statusbar != null) {  
// ③ 将SystemUIVisibility设置给SystemUI进程中的BaseStatusBar  
statusbar.setSystemUiVisibility(visibility, 0xffffffff); } } catch  
(RemoteException e) {.....} }});return diff; }
```

可见在PhoneWindowManager中，焦点窗口所期望的SystemUIVisibility会被保存到PhoneWindowManager.mSystemUiVisibility中。

3. 状态栏与导航栏中的SystemUIVisibility

BaseStatusBar并没有提供setSystemUiVisibility的实现，因此仍然以PhoneStatusBar的实现为例进行探讨。

```
[PhoneStatusBar.java-->PhoneStatusBar.setSystemUiVisibility()] public void setSystemUiVisibility(int vis, int mask) { .....if (diff != 0) { // ① 保存 SystemUIVisibility 到 mSystemUiVisibility 成员变量之中 mSystemUiVisibility = newVal; // ② PhoneStatusBar 只负责处理 SYSTEM_UI_FLAG_LOW_PROFILE 标记 if (0 != (diff & View.SYSTEM_UI_FLAG_LOW_PROFILE)) { final boolean lightsOut = (0 != (vis & View.SYSTEM_UI_FLAG_LOW_PROFILE)); ..... if (mNavigationBarView != null) { // 使导航栏进入或推出低辨识度模式 mNavigationBarView.setLowProfile(lightsOut); } // 使状态栏进入低辨识度模式 statusBarLowProfile(lightsOut); } // ③ 将 mSystemUiVisibility 设置回 WMS notifyUiVisibilityChanged(); } }
```

PhoneStatusBar会将PhoneWindowManager所给予的SystemUIVisibility保存到PhoneStatusBar.mSystemUiVisibility。另外可以看出PhoneStatusBar只对SYSTEM_UI_FLAG_LOW_PROFILE标记进行处理。随后，

notifyUiVisibilityChanged () 的调用将会把mSystemUiVisibility设置到WMS。

4.回到WMS的SystemUiVisibility

WindowManagerService中的statusBarVisibilityChanged () 方法将会接受来自PhoneStatus-Bar的SystemUiVisibility。

[WindowManagerService.java--

```
>WindowManagerService.statusBarVisibilityChanged() public void  
statusBarVisibilityChanged(int visibility) {.....synchronized  
(mWindowMap) { // ① 保存到mLastStatusBarVisibility成员中  
mLastStatusBarVisibility = visibility; /* ② 过滤来自SystemUI的  
SystemUIVisibility PhoneWindowManager.adjustSystemUiVisibility()会移  
除mResetSettingsSystemUi-Flags以及mForceClearedSystemUiFlags中所定  
义的SystemUIVisibility标记 */ visibility =  
mPolicy.adjustSystemUiVisibilityLw(visibility); // ③ 把修正后的  
SystemUIVisibility通知给每一个窗口  
updateStatusBarVisibilityLocked(visibility); } }
```

WMS将SystemUIVisibility保存到mLastStatusBarVisibility中。

最后updateStatusBarVisibility () 方法将会被PhoneWindowManager修正过的SystemUI-Visibility存储到每一个WindowState中，并发送给窗口所

在进程的ViewRootImpl。如前文所属，PhoneWindowManager对SystemUIVisibility的修正在于移除mResettingSystemUiFlags以及mForceClearedSystemUiFlags所定义的标记。

5.回到ViewRootImpl的SystemUiVisibility

修正过的SystemUiVisibility会通过IWindow.dispatchSystemUiVisibilityChanged () 方法通知到窗口所在进程的ViewRootImpl。ViewRootImpl会通过View.updateLocalSystemUiVisibility () 将遍历控件树中的每一个控件以更新控件中所保存的View.mSystemUiVisibility。同时注册在控件中的OnSystemUiVisibilityChangeListener监听器会通知对SystemUiVisibility的变化感兴趣的组件。

6.总结

可见，SystemUiVisibility从一个控件的setSystemUiVisibility () 开始，先后经历了View-RootImpl、WMS、PhoneWindowManager、PhoneStatusBar、WMS、ViewRootImpl几个组件的处理与保存，最后回到控件中。在这个过程中如下几个位置保存了SystemUIVisibility以供使用：

·View.mSystemUiVisibility，保存了一个控件所期望的SystemUIVisibility。

- AttachInfo.mSystemUiVisibility，保存了整个控件树所期望的 SystemUiVisibility，它是控件树中每个控件所期望的SystemUiVisibility 之和。
- LayoutParams.systemUiVisibility 与 LayoutParams.subtreeSystemUiVisibility，二者一起构成窗口所期望的 SystemUiVisibility。其中LayoutParams.subtreeSystemUiVisibility 等同于 AttachInfo.mSystemUiVisibility。
- WindowState.mSystemUiVisibility，在WMS中保存了一个窗口所期望的 SystemUi-Visibility，将被PhoneWindowManager用来调整对此窗口的布局行为。尤其是，当此窗口是焦点窗口时，它将会影响状态栏与导航栏的可见性。
- PhoneWindowManager.mLastSystemUiFlags，在PhoneWindowManager 中保存焦点窗口所期望的SystemUiVisibility，将用来影响状态栏与导航栏的可见性。
- PhoneStatusBar.mSystemUiVisibility，其值等同于 PhoneWindowManager.mLastSystemUi-Flags。目前PhoneStatusBar会根据其中的SYSTEM_UI_FLAG_LOW_PROFILE标记将状态栏与导航栏设置为低辨识度模式。

· WindowManagerService.mLastStatusBarVisibility，在WMS中保存了焦点窗口所期望的SystemUiVisibility，其值等同于 PhoneStatusBar.mSystemUiVisibility。当PhoneWindowManager因为某种原因希望清除某些标记（如 SYSTEM_UI_FLAG_HIDE_NAVIGATION）时，WMS会以此成员的内容为基础，在删除了PhoneWindowManager希望清除的标记之后通过 updateStatusBarVisibilityLocked () 将修正后的SystemUiVisibility保存到每一个WindowState之中并将这一变化通知窗口所在进程的 ViewRootImpl。

7.5.2 SystemUIVisibility发挥作用

接下来讨论每一种SystemUIVisibility标记是如何发挥作用的。

1.SYSTEM_UI_FLAG_LOW_PROFILE

处理这一标记的位置位于PhoneStatusBar.setSystemUiVisibility () 中。当PhoneStatusBar.mSystemUiVisibility中包含这一标记时，PhoneStatusBar会通过分别调用PhoneStatusBar.setStatusBarLowProfile () 和NavigationBarView.setLowProfile () 两个方法使状态栏与导航栏进入低辨识度模式。

其中PhoneStatusBar.setStatusBarLowProfile () 会将容纳通知信息的 @id/notification_icon_area、容纳系统状态图标的@id/statusIcons、显示

信号信息的@id/signal_cluster、显示电量信息的@id/battery以及显示系统时钟的@id/clock这5个控件的透明度在0或0.5与1之间切换，使状态栏进入或退出低辨识度模式。



说明

在低辨识度模式下，@id/battery与@id/clock的透明度其实是0.5即半透明状态，其他三个控件的透明度是0，即完全不可见。

而NavigationBarView.setLowProfile () 则通过设置容纳4个虚拟按键的@id/nav_buttons以及容纳三个灰色圆点的@id/lights_out这两个控件的透明度来完成低辨识度模式的进入或退出。在低辨识度模式下@id/nav_buttons的透明度为0，而@id/lights_out的透明度为1。在正常模式下@id/nav_buttons的透明度为1，而@id/lights_out的透明度为0。

2.SYSTEM_UI_FLAG_HIDE_NAVIGATION

对SYSTEM_UI_FLAG_HIDE_NAVIGATION的处理位于窗口布局过程的开端：PhoneWindowManager.beginLayoutLw ()。这个方法会通过检查mLastSystemUiFlags中是否存在这一标记并更改导航栏窗口的可见

性。导航栏可见性的更改会进一步影响作为窗口布局准绳的一些矩形的计算。参考其代码：

```
[PhoneWindowManager.java-->PhoneWindowManager.beginLayoutLw()]  
public void beginLayoutLw(boolean isDefaultDisplay, int displayWidth , int  
displayHeight, int displayRotation) {.....// ① 检查mLastSystemUiFlags中是  
否存在此标记。倘若存在则标记navVisible为falseboolean navVisible =  
(mLastSystemUiFlags&View.SYSTEM_UI_FLAG_HIDE_NAVIGATION)  
== 0;..... // 向WMS添加或移除用于截获用户触摸事件的FakeWindow/*  
倘若mCanHideNavigationBar为false，则强制导航栏显示。  
mCanHideNavigationBar被设置于  
PhoneWindowManager.setInitialDisplaySize()中。 mCanHideNavigationBar  
标记是否允许标记SYSTEM_UI_FLAG_HIDE_NAVIGATION隐藏导航  
栏。当SystemUI采用状态栏与导航栏分离式的布局方案时，  
mCanHideNavigationBar永远为true。因为横屏情况下导航栏位于屏幕  
的右侧，那么隐藏它可以为横屏下的宽屏视频播放提供更大的空间。  
而当SystemUI采用集成状态栏与导航栏为系统栏的方案时，会计算屏  
幕的宽高比。当宽高比大于16：9时，将会使得  
mCanHideNavigationBar为true，以致在横屏下的宽屏视频播放时可以充  
分利用屏幕的高度 */navVisible |= !mCanHideNavigationBar;if  
(mNavigationBar != null) { // 确定导航栏的方向  
mNavigationBarOnBottom = (!mNavigationBarCanMove || displayWidth <
```

```
displayHeight); if (mNavigationBarOnBottom) { // 计算导航栏的
ParentFrame，保存在mTmpNavigationFrame中 int top = displayHeight -
mNavigationBarHeightForRotation[displayRotation];
mTmpNavigationFrame.set(0, top, displayWidth, displayHeight); /* ② 根据
navVisible设置导航栏窗口的可见性，并据此调整作为窗口布局准绳包
括Dock、Rest- ricted、System、Content几个矩形 */ // 更新Stable和
StableFullscreen矩形，它与导航栏的窗口可见性无关 mStableBottom =
mStableFullscreenBottom = mTmpNavigationFrame.top; if (navVisible) { //
通过调用WindowState.showLw()显示导航栏的窗口
mNavigationBar.showLw(true); // 更新Dock以及Restricted矩形
mDockBottom = mTmpNavigationFrame.top; mRestrictedScreenHeight =
mDockBottom - mDockTop; } else { // 通过WindowState.hideLw()方法隐
藏导航栏的窗口 mNavigationBar.hideLw(true); } if (navVisible &&
!mNavigationBar.isAnimatingLw()) { // 更新System矩形 mSystemBottom
= mTmpNavigationFrame.top; } } else { ..... // 当导航栏位于屏幕右侧时
的处理方式与导航栏位于屏幕底部时的处理方式类似 } // 更新Content
矩形 mContentTop = mCurTop = mDockTop; mContentBottom =
mCurBottom = mDockBottom; mContentLeft = mCurLeft = mDockLeft;
mContentRight = mCurRight = mDockRight; .....}..... }
```

首先，PhoneWindowManager.beginLayout () 通过
WindowState.showLw () /hideLw () 进行导航栏窗口的显示与隐藏。

这两个方法是在WMS范畴内隐藏或显示现有窗口的一个十分便捷的方法。注意hideLw () 只是将窗口隐藏，而不是将窗口从WMS中移除。

另外，当焦点窗口声明SYSTEM_UI_FLAG_HIDE_NAVIGATION标记进而影响导航栏的可见性之后，会影响在PhoneWindowManager中作为窗口布局准绳的几个矩形：Dock、Restricted、System以及Content。因此在一个窗口中声明这个标记会对所有窗口的布局造成影响。

另外，通过canHideNavigationBar在PhoneWindowManager.setInitialDisplaySize () 方法中的计算过程可以体会到，Android极其不希望可以隐藏导航栏。因为导航栏所提供的BACK键、HOME键为用户提供了离开当前程序的途径（尤其是HOME键，通过第5章关于事件派发的知识可以知道，HOME键可以强制AMS启动桌面程序）。而声明了SYSTEM_UI_FLAG_HIDE_NAVIGATION标记并使之隐藏导航栏的往往就是当前应用程序。为了避免恶意应用程序无限期地隐藏导航栏，PhoneWindowManager通过FakeWindow机制为用户提供了一种绕过当前应用程序快速呼出导航栏的方法。

FakeWindow顾名思义，就是假窗口。经过本书第4章及第5章的介绍可知，一个窗口拥有两个重要特点：一是拥有一块Surface用于显示内容，另一个拥有一个InputWindowHandle用于接收用户事件。而FakeWindow则只有InputWindowHandle却没有Surface。

当PhoneWindowManager隐藏导航栏时，它会通过 WindowManagerService.addFakeWindow() 方法创建一个窗口类型为 TYPE_HIDDEN_NAV_CONSUMER 的 FakeWindow—— mHideNavFakeWindow。TYPE_HIDDEN_NAV_CONSUMER 类型在所有窗口类型中拥有最高的 Z 序，因此它将覆盖在所有可能的窗口之上。由于 FakeWindow 拥有 InputWindowHandle，所以用户在屏幕上的任何点击操作都会被 mHideNavFakeWindow 截获。PhoneWindowManager 会在 mHideNavFakeWindow 截获用户的点击之后强行将 SYSTEM_UI_FLAG_HIDE_NAVIGATION 标记从 WindowManagerService.mLastStatusBarVisibility 中移除，并重新对所有窗口进行布局使得导航栏再次出现在用户面前。

参考 PhoneWindowManager 创建 FakeWindow 的代码：

```
[PhoneWindowManager.java-->PhoneWindowManager.beginLayoutLw()]
public void beginLayoutLw(boolean isDefaultDisplay, int displayWidth, int
displayHeight, int displayRotation) {/* 通过检查
SYSTEM_UI_FLAG_HIDE_NAVIGATION 标记以及
mCanHideNavigationBar 计算 navVisible (/.....if (navVisible) { // ① 倘若导
航栏可见，则移除 FakeWindow。因为用作呼出导航栏的它已经不再需
要了 if (mHideNavFakeWindow != null) {
mHideNavFakeWindow.dismiss(); mHideNavFakeWindow = null; } } else if
```

```
(mHideNavFakeWindow == null) { // ② 倘若导航栏不可见，则创建  
FakeWindow。新的FakeWindow被保存在mHideNavFakeWindow中。注  
意mHideNavInputEventReceiverFactory参数，作为一个工厂类它可以用  
来创建一个InputEventReceiver，用于接收FakeWindow所截获的用户  
事件 */ mHideNavFakeWindow =  
m.WindowManagerFuncs.addFakeWindow( mHandler.getLooper(),  
mHideNavInputEventReceiverFactory, "hidden nav",  
WindowManager.LayoutParams.TYPE_HIDDEN_NAV_CONSUMER, 0,  
false, false, true); }..... // 进行导航栏的显示或隐藏 }
```

可见创建与移除FakeWindow的方法十分简单，其中最值得讨论的便是mHideNavInputEventReceiverFactory所创建的InputEventReceiver是如何呼出已经隐藏的导航栏。

PhoneWindowManager实现InputEventReceiver的一个子类
HideNavInputEventReceiver，并在mHideNavInputEventReceiverFactory
中创建它。

参考HideNavInputEventReceiver.onInputEvent () 的代码：

```
[PhoneWindowManager.java--  
>HideNavInputEventReceiver.onInputEvent()] public void  
onInputEvent(InputEvent event) {boolean handled = false;try { //
```

HideNavInputEventReceiver只对触摸事件的ACTION_DOWN感兴趣 if
(event instanceof MotionEvent && (event.getSource() &
InputDevice.SOURCE_CLASS_POINTER) != 0) { final MotionEvent
motionEvent = (MotionEvent)event; if (motionEvent.getAction() ==
MotionEvent.ACTION_DOWN) { boolean changed = false; synchronized
(mLock) { /* ① 首先刷新mResettingSystemUiFlags。
mResettingSystemUiFlags用于在系统现有的SystemUiVisibility中清除
所有会对状态栏/导航栏的可见性产生影响的标记*/ int newVal =
mResettingSystemUiFlags |
View.SYSTEM_UI_FLAG_HIDE_NAVIGATION |
View.SYSTEM_UI_FLAG_LOW_PROFILE |
View.SYSTEM_UI_FLAG_FULLSCREEN; if (mResettingSystemUiFlags
!= newVal) { mResettingSystemUiFlags = newVal; changed = true; } /* ②
刷新mForceClearedSystemUiFlags。 mForceClearedSystemUiFlags用于在
随后的一段时间内对会导致隐藏导航栏的
SYSTEM_UI_FAG_HIDE_NAVIGATION标记进行屏蔽 */ newVal =
mForceClearedSystemUiFlags |
View.SYSTEM_UI_FLAG_HIDE_NAVIGATION; if
(mForceClearedSystemUiFlags != newVal) { mForceClearedSystemUiFlags
= newVal; /* 在一秒之后从mForceClearedSystemUiFlags中删除
SYSTEM_UI_FLAG_HIDE_NAVIGATION，以取消对此标记的屏蔽 */

```
mHandler.postDelayed(new Runnable() { @Override public void run() {  
    synchronized (mLock) { mForceClearedSystemUiFlags &=  
        ~View.SYSTEM_UI_FLAG_HIDE_NAVIGATION; }  
    mWindowManagerFuncs.reevaluateStatusBarVisibility(); } }, 1000); } } //  
③ 调用WMS.reevaluateStatusBarVisibility()方法刷新SystemUIVisibility if  
(changed) { mWindowManagerFuncs.reevaluateStatusBarVisibility(); } } } }  
finally {.....} }
```

当HideNavInputEventReceiver收到用户点击事件之后主要做了三个方面的工作：

- 刷新mResettingSystemUiFlags。被刷新的mResettingSystemUiFlags将包含会影响状态栏或导航栏的可见性的三个标记：
SYSTEM_UI_FLAG_LOW_PROFILE、
SYSTEM_UI_FLAG_HIDE_NAVIGATION以及
SYSTEM_UI_FLAG_FULLSCREEN。当窗口通过WMS.relayoutWindow()
对窗口进行重新布局，或WMS调用PhoneWindow-
Manager.adjustSystemUiVisibilityLw()
时，mResettingSystemUiFlags会将这些标记从中移除。

- 刷新mForceClearedSystemUiFlags。mForceClearedSystemUiFlags只负责清除SYSTEM_UI_FLAG_HIDE_NAVIGATION。

mForceClearedSystemUiFlags的存在是为了阻止恶意程序不断隐藏导航栏以阻挠用户对导航栏进行操作。本小节随后将对其原理进行介绍。

·调用WMS.reevaluateStatusBarVisibility () 。这会诱使WMS调用PhoneWindowManager.adjustSystemUiVisibilityLw () 方法使得PhoneWindowManager将mResettingSystemUiFlags以及mForceClearedSystemUiFlags中定义的标记从WMS.mLastStatusBarVisibility中删除，然后通过WMS.updateSystemUiVisibility () 方法将删除这些标记之后的SystemUiVisibility更新至每一个窗口的WindowState以及窗口所在进程的ViewRootImpl，最后进行一次performLayoutAndPlaceSurfacesLocked () ，使得新的SystemUiVisibility在布局过程中生效（在beginLayoutLw () 和finishPostLayoutLw () 中）。

参考WMS.reevaluateStatusBarVisibility () 的实现：

[WindowManagerService.java--

```
>WindowManagerService.reevaluateStatusBarVisibility() public void  
reevaluateStatusBarVisibility() {synchronized (mWindowMap) { /* ① 对  
mLastStatusBarVisibility进行过滤。
```

```
PhoneWindowManager.adjustSystemUiVisibilityLw()将删除  
mLastStatusBarVisibility 中的那些定义在mResettingSystemUiFlags以及  
mForceClearedSystemUiFlags中的标记*/ int visibility =
```

mPolicy.adjustSystemUiVisibilityLw(mLastStatusBarVisibility); /* ② 把过滤后的SystemUiVisibility更新到每一个窗口的WindowState以及窗口所在进程的View-RootImpl。于是WindowState以及控件树中的控件所保存的SystemUiVisibility都会被更新 */

updateStatusBarVisibilityLocked(visibility); /* ③ 进行一次完整的窗口布局操作。在这个过程中，新的SystemUiVisibility将通过PhoneWindowManager的beginLayoutLw()、finishPostLayoutLw()以及updateSystemUiVisibility()三个方法生效。最后PhoneStatusBar通过WMS.statusBarVisibilityChanged()将新的SystemUiVisibility保存在WMS.mLastStatusBarVisibility中 */

performLayoutAndPlaceSurfacesLocked(); } }

当用户在FakeWindow上产生点击后，WMS通过对mLastStatusBarVisibility进行过滤从而得到不包含导致状态栏与导航栏不可见的标记的SystemUiVisibility，并将其推送到每一个窗口（包括WindowState以及ViewRootImpl），然后启动一次完整的窗口布局过程使得PhoneWindowManager使用新的SystemUiVisibility对窗口进行布局以及设置状态栏与导航栏的可见性。在这个过程中，

PhoneWindowManager.updateSystemUiVisibility()会将新的SystemUiVisibility发送给PhoneStatusBar，而PhoneStatusBar通过调用WindowManager.status-BarVisibilityChanged()方法将新的SystemUiVisibility保存到WMS的mLastStatusBarVisibility中。简单来

说，通过FakeWindow呼出被隐藏的导航栏是从mLastStatusBarVisibility开始，通过过滤、通知、布局之后再把新的SystemUiVisibility保存到mLastStatusBarVisibility这样一个环形的过程。

在这当中还需要讨论的是mResettingSystemUiVisibility与mForceClearedSystemUiVisibility的区别。考察所有它们所生效的代码都如下所示：

```
newVisibility = visibility & ~mResettingSystemUiFlags &  
~mForceClearedSystemUiFlags;
```

从这一点上来说，而这没有任何区别，它们都用来阻止某些标记出现在SystemUIVisibility中。它们唯一的区别是——何时重置，即何时终止它们对特定标记的阻止行为。

首先是mResettingSystemUiFlags。在PhoneWindowManager.adjustSystemUiVisibilityLw () 中有如下代码：

```
[PhoneWindowManager.java-->PhoneWindowManager.adjustSystemUiVisibilityLw()] public int  
adjustSystemUiVisibilityLw(int visibility) {/* mResettingSystemUiFlags取消对某一个标记的阻止行为的时机是系统中的SystemUIVisibility 中不包含这一标记 */mResettingSystemUiFlags &= visibility;return visibility &  
~mResettingSystemUiFlags & ~mForceClearedSystemUiFlags; }
```

以SYSTEM_UI_FLAG_LOW_PROFILE为例介绍这一行代码的工作原理。假设WMS.mLastStatusBarVisibility中包含这一标记并且导航栏处于隐藏状态。当用户点击FakeWindow之后，WMS会第一次调用PhoneWindowManager.adjustSystemUiVisibilityLw () 方法，此时由于这一标记存在， mResettingSystemUiFlags&=visibility这一条语句会使得mResettingSystemUiVisibility仍然保持这一标记，从而将返回值中的这一标记去除。标记去除后的SystemUIVisibility最终通过PhoneStatusBar调用WMS.statusBar-VisibilityChanged () 方法再次进入WMS，而WMS会再次调用PhoneWindowManager.adjustSystemUiVisibilityLw () ，此时SystemUIVisibility中已经不再包含SYSTEM_UI_FLAG_LOW_PROFILE标记了，因此mResettingSystemUiFlags&=visibility这一条语句会使得mResettingSystemUiFlags移除这一标记，因此取消了对这一标记的阻止。在此之后，应用程序可以立刻重新声明SYSTEM_UI_FLAG_LOW_PROFILE。也就是说，mResetting-SystemUIVisibility存在的意义就是为了当用户点击FakeWindow后呼出状态栏与导航栏，它并不阻止应用程序随后对状态栏或导航栏进行隐藏的尝试。

而mForceClearedSystemUiFlags被清空的位置在HideNavInputEventReceiver.onInput-Event () 方法所抛出的那个Runnable中。也就是说，一旦FakeWindow呼出了导航栏，那么在随后

的1秒内，`mForceClearedSystemUiFlags`中将会持续保持对`SYSTEM_UI_FLAG_HIDE_NAVIGATION`的阻止行为。或者说，在这1秒的时间内导航栏不会再次被隐藏。`mForceClearedSystemUiVisibility`存在的意义就是为了防止恶意程序不断通过声明`SYSTEM_UI_FLAG_HIDE_NAVIGATION`而使得用户无法对导航栏进行操作。

总结一下`SYSTEM_UI_FLAG_HIDE_NAVIGATION`对系统的影响。当焦点窗口拥有这一标记时，导航栏会被隐藏，作为窗口布局准绳的几个矩形会被重新计算进而影响所有窗口的布局结果。当这一标记是导航栏被隐藏时，一个`FakeWindow`覆盖在所有窗口之上。一旦用户点击屏幕时，导航栏会被重新呼出，`SYSTEM_UI_FLAG_HIDE_NAVIGATION`标记会被强制移除，并且在随后的1秒内，应用程序无法再通过这一标记对导航栏进行隐藏操作。

3. SYSTEM_UI_FLAG_FULLSCREEN

首先需要澄清的是，千万不要被这个标记的名字误导，这一标记仅仅用来隐藏状态栏，而不是同时隐藏状态栏与导航栏。这一标记的目的是希望用户能够更多地将注意力专注在其应用程序的内容上，并占用状态栏的空间。对此标记进行处理的位置位于`PhoneWindowManager.finishPostLayoutLw()`中。参考相关代码：

[PhoneWindowManager.java--

>PhoneWindowManager.finishPostLayoutLw()] public int
finishPostLayoutPolicyLw() {int changes = 0;boolean topIsFullscreen =
false; /* ① 会影响状态栏可见性的窗口不是焦点窗口，而是
mTopFullscreenOpaqueWindowState，即显示 在最上面的全屏窗口。这
与SYSTEM_UI_FLAG_HIDE_NAVIGATION标记不同 */final
WindowManager.LayoutParams lp = (mTopFullscreenOpaqueWindowState
!= null) ? mTopFullscreenOpaqueWindowState.getAttrs() : null;.....if
(mStatusBar != null) { if (mForceStatusBar ||
mForceStatusBarFromKeyguard) { /* ② 强制状态栏显示。倘若
mTopFullscreenOpaqueWindowState之上有窗口在其flags中声明
FLAG_FORCE_NOT_FULLSCREEN标记，那么状态栏将会被强制显
示。此时SYSTEM_UI_FLAG_FULLSCREEN 会被忽略 */ if
(mStatusBar.showLw(true)) changes |=
FINISH_LAYOUT_REDO_LAYOUT; } else if
(mTopFullscreenOpaqueWindowState != null) { /* ③ 检查是否需要隐
藏状态栏。可见SYSTEM_UI_FLAG_FULLSCREEN只是隐藏状态栏的
方法之一 */ topIsFullscreen = (lp.flags &
WindowManager.LayoutParams.FLAG_FULLSCREEN) != 0 ||
(mLastSystemUiFlags & View.SYSTEM_UI_FLAG_FULLSCREEN) != 0;
if (topIsFullscreen) { // 通过WindowState.hideLw()隐藏状态栏 if

```
(mStatusBar.hideLw(true)) { changes |=  
    FINISH_LAYOUT_REDO_LAYOUT; ..... } } else { // 通过  
    WindowState.showLw()显示状态栏 if (mStatusBar.showLw(true)) changes  
    |= FINISH_LAYOUT_REDO_LAYOUT; } }..... // 对锁屏的操作/* ④ 如  
有必要，重新开始一次布局。当状态栏的可见性发生变化之后，说明  
作为窗口布局准绳的矩形需要重新计算，所以返回的changes 会包含  
FINISH_LAYOUT_REDO_LAYOUT标记以重新启动一次布局。在新的  
布局过程中PhoneWindow- Manager.beginLayoutLw()会对布局准绳进行  
修改 */return changes; }
```

首先与SYSTEM_UI_FLAG_HIDE_NAVIGATION所不同的是，只有当显示在最上面的全屏窗口声明此标记时才会导致隐藏状态栏。这个区别的存在其实非常好理解。隐藏导航栏会使得用户无法发送虚拟按键，因而焦点窗口是这一行为的利益攸关方。所以Android将这一权利赋予焦点窗口。而隐藏状态栏的目的是让用户将注意力集中在窗口所显示的内容上，并占用状态栏所处的空间，因而仅当一个可以充满屏幕的窗口才能从中获利。所以Android将这一权利赋予显示在最上面的全屏窗口。

这一个区别也导致两者处理位置的不同。对SYSTEM_UI_FLAG_HIDE_NAVIGATION来说，确定焦点窗口非常简单而且与布局无关，因此WMS可以在布局之前便将焦点窗口确定下来

并保存到PhoneWindowManager。于是PhoneWindowManager可以在布局一开始的beginLayoutLw () 中从焦点窗口中获取这一标记，并隐藏／显示状态栏进而修改布局准绳，最后再对每个窗口进行布局。这一过程如顺流而下十分自然。对SYSTEM_UI_FLAG_FULLSCREEN来说，显示在最上面的全屏窗口的确定依赖于布局结果，因此PhoneWindowManager必须在完成一轮布局之后才能得知哪一个窗口的SYSTEM_UI_FLAG_FULLSCREEN需要进行处理。所以对SYSTEM_UI_FLAG_FULLSCREEN的处理位于布局结尾处的finishPostLayoutLw ()，并且当状态栏可见性发生变化后通过在返回值中增加FINISH_LAYOUT_REDO_LAYOUT标记导致重新布局，并在新的布局过程的beginLayoutLw () 中根据状态栏当前的可见性重新计算作为窗口布局准绳的矩形，然后重新布局所有窗口。

另外，除了SYSTEM_UI_FLAG_FULLSCREEN之外，窗口还可以通过在Layout-Params.flags添加FLAG_FULLSCREEN标记完成对状态栏的隐藏。二者有什么区别呢？经过上一节对SYSTEM_UI_FLAG_HIDE_NAVIGATION的探讨可知，SYSTEM_UI_FLAG_FULLSCREEN这一标记的存在性并不稳定，因此它适用于需要在用户处于特定交互的情况下临时地隐藏状态栏。对于那些需要持续隐藏状态栏的情况（例如全屏游戏），不会被PhoneWindowManager清除的FLAG_FULLSCREEN是一个更好的选择。



注意

显而易见，SYSTEM_UI_FLAG_HIDE_NAVIGATION与
SYSTEM_UI_FLAG_FULLSCREEN由于会改变作为窗口布局准绳的矩
形的位置和尺寸，因此它们会对所有参与布局的窗口产生影响。

4.SYSTEM_UI_FLAG_LAYOUT_STABLE

这个标记并不会对状态栏或导航栏的可见性产生任何影响，它只会对
声明这一标记的窗口的布局产生影响。

由于状态栏与导航栏所在的区域不属于窗口的内容区。于是窗口的
ContentFrame和Frame之间所产生的差异会导致一个ContentInset发送到
ViewRootImpl中。控件在绘制的过程中，需要绕开ContentInset所在的
区域，以免其被状态栏或导航栏所覆盖。当状态栏在隐藏与显示两种
状态之间切换时，会使得ContentFrame不断变化，进而使得控件所绘制
的内容上下跳动。

为了避免这种令人不快的跳动，窗口可以通过声明 SYSTEM_UI_FLAG_LAYOUT_STABLE 标记以获得一个稳定的 (STABLE) 的 ContentFrame，从而使得状态栏在显示与隐藏之间切换时不会对窗口的绘制产生影响。由于这一标记主要用于描述窗口针对状态栏的布局行为，因此此标记仅对那些声明了与状态栏相关的标记的窗口才有效。SYSTEM_UI_FLAG_LAYOUT_STABLE 需要配合以下标记使用：

- 在 LayoutParams.flags 中指定了 FLAG_LAYOUT_IN_SCREEN。
- 声明了 SYSTEM_UI_FLAG_HIDE_NAVIGATION_BAR。
- 声明了 SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN。

这三个标记的共同特点是 PhoneWindowManager 为声明它们的窗口提供了充满屏幕顶部的 ParentFrame，使得这种窗口顶部有可能会被状态栏所覆盖。因而 SYSTEM_UI_FLAG_LAYOUT_STABLE 才有存在的必要。

PhoneWindowManager 会在其 applyStableConstraints () 方法中检测这一标记并修改窗口的 ContentFrame。参考其代码，其中参数 r 为窗口的 ContentFrame。

[PhoneWindowManager.java--

```
>PhoneWindowManager.applyStableConstraints()]{ private void  
applyStableConstraints(int sysui, int fl, Rect r) { // 检查窗口的  
SystemUIVisibility中是否声明了  
SYSTEM_UI_FLAG_LAYOUT_STABLEif ((sysui &  
View.SYSTEM_UI_FLAG_LAYOUT_STABLE) != 0) { // 根据  
LayoutParams.flags中是否有FLAG_FULLSCREEN标记会返回不同的  
ContentFrame if ((fl & FLAG_FULLSCREEN) != 0) { // ① 当窗口声明  
FLAG_FULLSCREEN时， ContentFrame以StableFullscreen矩形为准 if  
(r.left < mStableFullscreenLeft) r.left = mStableFullscreenLeft; if (r.top <  
mStableFullscreenTop) r.top = mStableFullscreenTop; if (r.right >  
mStableFullscreenRight) r.right = mStableFullscreenRight; if (r.bottom >  
mStableFullscreenBottom) r.bottom = mStableFullscreenBottom; } else { //  
② 当窗口没有声明FLAG_FULLSCREEN时， ContentFrame以Stable矩形  
为准 if (r.left < mStableLeft) r.left = mStableLeft; if (r.top < mStableTop)  
r.top = mStableTop; if (r.right > mStableRight) r.right = mStableRight; if  
(r.bottom > mStableBottom) r.bottom = mStableBottom; } } }
```

可见，一旦窗口声明了SYSTEM_UI_FLAG_LAYOUT_STABLE标记，那么它的ContentFrame都会按照两种Stable矩形进行校正。这两种Stable矩形都不会随着状态栏的可见性的变化而变化。因此窗口的控件树可

以获得一个稳定的ContentInsets，进而避免状态栏可见性变化时所产生的内容跳动。

Stable矩形和StableFullscreen矩形之间唯一的区别在于它们的top值不同。Stable矩形的top值为状态栏的高度（无论状态栏显示与否），因此当窗口没有同时声明FLAG_FULLSCREEN时，它将固定地获得一个将状态栏排除在外的ContentInsets，因此，无论状态栏显示与否都不会遮挡窗口控件树所绘制的内容。而StableFullscreen矩形的top值为0，因此状态栏显示时会遮挡同时声明了FLAG_FULLSCREEN的窗口控件树所绘制的内容。

5.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION

这个标记不会对状态栏与导航栏的可见性产生任何影响。它只会影响声明这个标记的窗口的布局。PhoneWindowManager在对声明这个标记的窗口进行布局时会假想导航栏已经被隐藏，即导航栏所在的区域也会被用来进行窗口布局。与SYSTEM_UI_FLAG_LAYOUT_STABLE相比，这个标记同时影响了ParentFrame、DisplayFrame以及ContentFrame。

处理这个标记的代码位于PhoneWindowManager.layoutWindowLw () 中。参考如下代码：

[PhoneWindowManager.java-->PhoneWindowManager.layoutWindowLw()]

```
public void layoutWindowLw(WindowState win,  
    WindowManager.LayoutParams attrs, WindowState attached) {.....if ((fl &  
    (FLAG_LAYOUT_IN_SCREEN | FLAG_FULLSCREEN |  
    FLAG_LAYOUT_INSET_DECOR)) == (FLAG_LAYOUT_IN_SCREEN |  
    FLAG_LAYOUT_INSET_DECOR) && (sysUiFl &  
    View.SYSTEM_UI_FLAG_FULLSCREEN) == 0) { if (.....) { ..... } else if  
    (mCanHideNavigationBar && (sysUiFl &  
    View.SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION) != 0 &&  
    attrs.type >=  
    WindowManager.LayoutParams.FIRST_APPLICATION_WINDOW &&  
    attrs.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) { /*  
    ① 对声明SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION的窗口来  
    说，其ParentFrame、DisplayFrame都来自Unrestricted矩形 */ pf.left =  
    df.left = mUnrestrictedScreenLeft; pf.top = df.top =  
    mUnrestrictedScreenTop; pf.right = df.right =  
    mUnrestrictedScreenLeft+mUnrestrictedScreenWidth; pf.bottom =  
    df.bottom = mUnrestrictedScreenTop+mUnrestrictedScreenHeight; } else {  
    /* ② 未声明此标记的窗口，其ParentFrame、DisplayFrame来自  
    Restricted矩形 pf.left = df.left = mRestrictedScreenLeft; pf.top = df.top =  
    mRestrictedScreenTop; pf.right = df.right =
```

```
mRestrictedScreenLeft+mRestrictedScreenWidth; pf.bottom = df.bottom =  
mRestrictedScreenTop+mRestrictedScreenHeight; } }..... }
```

可见SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION标记决定窗口的ParentFrame以及DisplayFrame选择Restricted矩形还是Unrestricted矩形。这两个矩形唯一的差别就是，Unrestricted矩形为整块屏幕，而Restricted矩形则不包括导航栏。因此，这一标记的效果是使得窗口的布局延展到整个屏幕。为了防止声明这一标记的窗口覆盖导航栏，因此要求声明这一标记的窗口必须来自应用程序，以保证窗口的Z序位于导航栏以下。

另外，参考代码最外层的if语句指出了上述行为的限定条件，即窗口同时在LayoutParams.flags中声明FLAG_LAYOUT_IN_SCREEN以及FLAG_LAYOUT_INSET_DECOR，但是没有声明FLAG_FULLSCREEN，也没有声明SYSTEM_UI_FLAG_FULLSCREEN。当不满足上述条件时，SYSTEM_UI_FLAG_FULLSCREEN不仅仅会修改DisplayFrame以及ParentFrame，还会使用Unrestricted矩形作为窗口ContentFrame。

产生这种不同的核心原因在于FLAG_LAYOUT_INSET_DECOR。这一标记表示声明它的窗口需要通过ContentInset知道状态栏与导航栏相对于窗口的位置与尺寸，以便在绘制时做出相应的处理。所以上述所引用的代码中，SYSTEM_UI_FLAG_LAYOUT_HIDE_NAVIGATION标记

仅仅影响ParentFrame以及DisplayFrame，而ContentFrame依然按照默认的（考虑状态栏与导航栏的位置）计算方式进行。于是ContentInsets便可以反映出状态栏与导航栏相对于窗口的位置和尺寸。

6.SYSTEM_UI_FLAG_LAYOUT_FULLSCREEN

这一标记并不会影响状态栏与导航栏的可见性。但是影响声明它的窗口的布局。这使得PhoneWindowManager对声明此标记的窗口进行布局时假设状态栏已经隐藏，但是导航栏依然处于显示状态。因此这个标记会使得窗口的ParentFrame、DisplayFrame以及ContentFrame使用Restricted矩形进行布局。于是窗口可以被布局在状态栏之下，并且不会因为状态栏的存在而产生ContentInsets。对这一标记的处理也位于PhoneWindowManager.layoutWindowLw () 中。读者可以自行研究。

7.总结

本节共介绍了6种SystemUIVisibility标记的处理过程及其所能产生的效果。

前三种都会对状态栏以及导航栏产生实质性影响，并且会影响所有窗口的布局结果。而后三种

SYSTEM_UI_FLAG_LAYOUT_STABLE/HIDE_NAVIGATION/FULLSCREEN会对声明它们的窗口布局产生影响。

另外后三者同时影响ParentFrame、DisplayFrame以及ContentFrame中的一个或多个。那么当这些标记同时存在时是否存在互斥或者优先级呢？通过分析PhoneWindowManager.layoutWindowLw（）方法的代码不难看出，HIDE_NAVIGATION与FULLSCREEN之间存在互斥，当两者同时存在时，使用HIDE_NAVIGATION对窗口进行布局。而STABLE与其他两者可以共同发生作用。STABLE会对HIDE_NAVIGATION或FULLSCREEN所计算出的ContentFrame进行修正。

7.5.3 SystemUIVisibility总结

关于SystemUIVisibility的话题便介绍到这里。SystemUIVisibility是应用程序与状态栏和导航栏进行互动的主要手段，主要用于影响状态栏与导航栏的可见性以及窗口的布局方式。另外，对系统开发者来说，相对扩充StatusBarManager的接口以扩展状态栏/导航栏与应用程序的交互行为，扩展SystemUIVisibility标记并在PhoneStatusBar.setSystemUiVisibility（）中进行相应处理无疑是一个更加方便而且影响较小的途径。

7.6 本章小结

本章详细介绍了SystemUI中最常用也是最重要的两个功能：状态栏与导航栏的工作原理。根据屏幕尺寸不同，Android实现两套状态栏与导航栏。本章仅讨论了其中一套应用于小屏幕设备上的状态栏与导航栏分离布局的方案。

读者需要理解状态栏与导航栏运行于一个名为SystemUIService的由system_server进程通过Context.startService（）方式启动的常规Android服务中，并且通过WindowManager.addView（）创建它们的窗口。这种由系统服务启动一个常规服务，并在这个常规服务中创建窗口的工作模式在Android系统中并不是独此一家。第8章中所介绍的Android壁纸也使用了这种工作模式，而且负责管理壁纸的WallpaperManagerService与负责壁纸显示的WallpaperService之间的交互以及窗口创建过程更加复杂。因此深入理解状态栏与导航栏的启动、窗口创建以及它们与StatusBarManagerService进行通信的方式对于继续学习第8章的内容有很大帮助作用。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第8章 深入理解Android壁纸

本章主要内容：

- 讨论动态壁纸的实现
- 在动态壁纸的基础上讨论静态壁纸的实现
- 讨论WMS对壁纸窗口所做的特殊处理

本章涉及的源代码文件名及位置：

·WallpaperManagerService.java

frameworks/base/services/java/com/android/server/WallpaperManagerService.java

·WallpaperService.java

frameworks/base/core/java/android/service/wallpaper/WallpaperService.java

·ImageWallpaper.java

frameworks/base/packages/SystemUI/src/com/android/systemui/ImageWallpaper.java

·WallpaperManager.java

frameworks/base/core/java/android/app/WallpaperManager.java

·WindowManagerService.java

frameworks/base/services/java/com/android/server/wm/WindowManagerService.java

·WindowStateAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowStateAnimator.java

·WindowAnimator.java

frameworks/base/services/java/com/android/server/wm/WindowAnimator.java

8.1 初识Android壁纸

本章将对壁纸的实现原理进行讨论。在Android中，壁纸分为静态与动态两种。静态壁纸是一张图片，而动态壁纸则以动画为表现形式，或者可以对用户的操作做出反应。这两种形式看似差异很大，其实二者的本质是统一的。它们都以一个Service的形式运行在系统后台，并在一个类型为TYPE_WALLPAPER的窗口上绘制内容。进一步讲，静态壁纸是一种特殊的动态壁纸，它仅在窗口上渲染一张图片，并且不会对用户的操作做出反应。因此本章将首先通过动态壁纸的实现讨论Android壁纸的实现与管理原理，然后再对静态壁纸的实现做介绍。

Android壁纸的实现与管理分为三个层次：

- WallpaperService与Engine。同SystemUI一样，壁纸运行在一个Android服务之中，这个服务的名字叫作WallpaperService。当用户选择一个壁纸之后，此壁纸所对应的WallpaperService便会启动并开始进行壁纸的绘制工作，因此继承并定制WallpaperService是开发者进行壁纸开发的第一步。Engine是WallpaperService中的一个内部类，实现了壁纸窗口的创建以及Surface的维护工作。另外，Engine提供了可供子类重写的一系列回调，用于通知壁纸开发者关于壁纸的生命周期、Surface状态的变化以及对用户的输入事件进行响应。可以说，Engine类是壁纸实现

的核心所在。壁纸开发者需要继承Engine类，并重写其提供的回调以完成壁纸的开发。这一层次的内容主要体现了壁纸的实现原理。

·WallpaperManagerService，这个系统服务用于管理壁纸的运行与切换，并通过WallpaperManager类向外界提供操作壁纸的接口。当通过WallpaperManager的接口进行壁纸的切换时，WallpaperManagerService会取消当前壁纸的WallpaperService绑定，并启动新壁纸的WallpaperService。另外，Engine类进行窗口创建时所使用的窗口令牌也是由WallpaperManagerService提供的。这一层次主要体现了Android对壁纸的管理方式。

· WindowManagerService，用于计算壁纸窗口的Z序、可见性以及为壁纸应用窗口动画。壁纸窗口（TYPE_WALLPAPER）的Z序计算不同于其他类型的窗口。其他窗口依照其类型会有固定的mBaseLayer以及mSubLayer，并结合它们所属的Activity的顺序或创建顺序进行Z序的计算，因此这些窗口的Z序相对固定。而壁纸窗口则不然，它的Z序会根据FLAG_SHOW_WALLPAPER标记在其他窗口LayoutParams.flags中的存在情况而不断调整。这一层次主要体现Android对壁纸窗口的管理方式。

本章将通过对动态壁纸切换的过程进行分析以揭示WallpaperService、Engine以及WallpaperManagerService三者的实现原理以及协作情况。静态壁纸作为动态壁纸的一种特殊情况，将会在完成动态壁纸的学习之

后于8.3节讨论。而WindowManagerService对壁纸窗口的处理将在8.4节介绍。

8.2 深入理解动态壁纸

8.2.1 启动动态壁纸的方法

启动动态壁纸可以通过调用WallpaperManager.getIWallpaperManager() .setWallpaper-Component () 方法完成。它接受一个 ComponentName类型的参数，用于将希望启动壁纸的WallpaperService 的ComponentName告知WallpaperManagerService。

WallpaperManager.getIWallpaperManager () 方法返回的是 WallpaperManagerService的Bp端。因此setWallpaper-Component () 方法的实现位于WallpaperManagerService中。参考其实现：

```
[WallpaperManagerService.java-->WallpaperManagerService.setWallpaperComponent()] public void setWallpaperComponent(ComponentName name) { // 设置动态壁纸需要调用者拥有一个签名级的系统权限 checkPermission(android.Manifest.permission.SET_WALLPAPER_COMPONENT); synchronized (mLock) { /* ① 首先从mWallpaperMap中获取壁纸的运行信息WallpaperData。 WallpaperManagerService支持多用户机制，因此设备上的每个用户可以设置自己的壁纸。 mWallpaper- Map为每个用户保存了一个WallpaperData实例，这个实例中保存和壁纸运行状态相关的信息。例如 WallpaperService的ComponentName，到
```

WallpaperService的ServiceConnection等。于是当发生用户切换时，WallpaperManagerService可以从mWallpaperMap中获取新用户的Wallpaper- Data，并通过保存在其中的ComponentName重新启动该用户所设置的壁纸。因此，当通过setWall- paperComponent()设置新壁纸时，需要获取当前用户的WallpaperData，并在随后更新其内容使之保存新壁纸的信息 */ int userId = UserHandle.getCallingUserId();
WallpaperData wallpaper = mWallpaperMap.get(userId); final long ident = Binder.clearCallingIdentity(); try { // ② 启动新壁纸的
WallpaperService bindWallpaperComponentLocked(name, false, true,
wallpaper, null); } finally { Binder.restoreCallingIdentity(ident); } }



注意

WallpaperManager.getIWallpaperManager () 并没有作为SDK的一部分提供给开发者。因此第三方应用程序是无法进行动态壁纸的设置的。

8.2.2 壁纸服务的启动原理

(1) 壁纸服务的验证与启动

bindWallpaperComponentLocked () 方法将会启动由ComponentName所指定的Wallpaper-Service，并向WMS申请用于添加壁纸窗口的窗口令牌。不过在此之前，bindWallpaper-ComponentLocked () 会对ComponentName所描述的Service进行一系列验证，以确保它是一个壁纸服务。而这一系列的验证过程体现了一个Android服务可以被当作壁纸必要的条件。

[WallpaperManagerService.java--

```
>WallpaperManagerService.setWallpaperComponentLocked() boolean  
bindWallpaperComponentLocked(ComponentName componentName,  
boolean force,boolean fromUser, WallpaperData wallpaper,  
IRemoteCallback reply) { ..... try { /* 当componentName为null时表示使用  
默认壁纸。 这里会将componentName参数改为默认壁纸的  
componentName */ if (componentName == null) { /* 首先会尝试从  
com.android.internal.R.string.default_wallpaper_component中获取默认壁  
纸的componentName。这个值的设置位于res/values/config.xml中，开发  
者可以通过修改这个值设置默认壁纸 */ String defaultComponent =  
mContext.getString(  
com.android.internal.R.string.default_wallpaper_component); if  
(defaultComponent != null) { componentName =
```

ComponentName.unflattenFromString(defaultComponent); } /* 倘若在上述资源文件中没有指定一个默认壁纸，即default_wallpaper_component的值被设置为@null)，则使用ImageWallpaper代替默认壁纸。

ImageWallpaper就是前文所述的静态壁纸 */ if (componentName == null)

{ componentName = IMAGE_WALLPAPER; } } /* 接下来

WallpaperMangerService会尝试从PackageManager中获取

ComponentName所指定的Service的描述信息,获取此信息的目的在于确

认该Service是一个符合要求的壁纸服务 */ int serviceUserId =

wallpaper.userId; ServiceInfo si =

mIPackageManager.getServiceInfo(componentName,

PackageManager.GET_META_DATA |

PackageManager.GET_PERMISSIONS, serviceUserId); /* ① 第一个检

查，要求这个Service必须声明其访问权限为BIND_WALLPAPER。这个

签名级的系统权限 用于防止壁纸服务被第三方应用程序启动而产生混

乱 */ if

(!android.Manifest.permission.BIND_WALLPAPER.equals(si.permission))

{ if (fromUser) { throw new SecurityException(msg); } return false; }

WallpaperInfo wi = null; /* ② 第二个检查，要求这个Service必须可以

来处理android.service.wallpaper.WallpaperService这个Action。其检查

方式是从PackageManager中查询所有可以处理android.

service.wallpaper.WallpaperService的服务，然后检查即将启动的服务是

否在PackageManager 的查询结果之中 */ Intent intent = new Intent(WallpaperService.SERVICE_INTERFACE); if (componentName != null && !componentName.equals(IMAGE_WALLPAPER)) { // 获取所有可以处理android.service.wallpaper.WallpaperService的服务信息 List ris =mIPackageManager.queryIntentServices(intent, intent.resolveTypeIfNeeded(mContext.getContentResolver()), PackageManager.GET_META_DATA, serviceUserId); /* ③ 第三个检查，要求这个Service必须在其meta-data中提供关于壁纸的描述信息。如果即将启动的服务位于查询结果之中，便可以确定这是一个壁纸服务。此时会创建一个Wall- paperInfo的实例以解析并存储此壁纸服务的描述信息。壁纸服务的描述信息包含了壁纸的开发者、缩略图、简单的描述文字以及用于对此壁纸进行参数设置的Activity的名字等。壁纸开发者可以在AndroidManifest.xml中将一个包含上述信息的xml文件设置在名为android. service.wallpaper的meta-data中以提供这些信息 */ for (int i=0; i

可见WallpaperManagerService要求被启动的目标Service必须满足以下三个条件：

- 该服务必须以android.permission.BIND_WALLPAPER作为其访问权限。虽然壁纸是一个标准的Android服务，但是通过其他途径（如第三方应用程序）启动壁纸所在的服务是没有意义的。因此Android要求作

为壁纸的Service必须使用这个签名级的系统权限进行访问限制，以免被意外的应用程序启动。

·该服务必须被声明为可以处理

android.service.wallpaper.WallpaperService这个Action。

WallpaperManagerService会使用这个Action对此服务进行绑定。

·该服务必须在其AndroidManifest.xml中提供一个名为

android.service.wallpaper的meta-data，用于提供动态壁纸的开发者、缩略图与描述文字。

一旦目标服务满足上述条件，WallpaperManagerService就会着手进行目标服务的启动与绑定。

参考setWallpaperComponentLocked () 方法的后续代码：

[WallpaperManagerService.java--

```
>WallpaperManagerService.setWallpaperComponentLocked() boolean  
bindWallpaperComponentLocked(ComponentName componentName,  
boolean force,boolean fromUser, WallpaperData wallpaper,  
IRemoteCallback reply) { ..... // 检查服务是否符合要求的代码 /* ① 创建  
一个WallpaperConnection。它不仅实现ServiceConnection接口用于监听  
和WallpaperService之间的连接状态，同时还实现  
IWallpaperConnection.Stub，也就是说它支持跨进程通信。在服务绑定
```

成功后的WallpaperConnection.onServiceConnected()方法调用中，
Wallpaper- Connection的实例会被发送给WallpaperService，使其作为
WallpaperService向Wallpaper- ManagerService进行通信的桥梁 */

```
WallpaperConnection newConn = new WallpaperConnection(wi, wallpaper);
// 为启动壁纸服务准备Intent intent.setComponent(componentName);
intent.putExtra(Intent.EXTRA_CLIENT_LABEL,
com.android.internal.R.string.wallpaper_binding_label);
intent.putExtra(Intent.EXTRA_CLIENT_INTENT,
PendingIntent.getActivityAsUser( mContext, 0, Intent.createChooser(new
Intent(Intent.ACTION_SET_WALLPAPER),
mContext.getText(com.android.internal.R.string.chooser_wallpaper)), 0,
null, new UserHandle(serviceUserId))); /* ② 启动指定的壁纸服务。当服
务启动完成后，剩下的启动流程会在WallpaperConnection.onSer
viceConnected()中继续 */ if (!mContext.bindService(intent,newConn,
Context.BIND_AUTO_CREATE, serviceUserId)) { } // ③ 新的壁纸服务启
动成功后，便通过detachWallpaperLocked()销毁旧有的壁纸服务 if
(wallpaper.userId == mCurrentUserId && mLastWallpaper != null) {
detachWallpaperLocked(mLastWallpaper); } // ④ 将新的壁纸服务的运行
信息保存到WallpaperData中 wallpaper.wallpaperComponent =
componentName; wallpaper.connection = newConn; /* 设置
wallpaper.lastDiedTime。这个成员变量与其说描述壁纸的死亡时间戳，
```

不如说是描述其启动的时间戳。它用来在壁纸服务意外断开时（即壁纸服务非正常停止）检查此壁纸服务的存活时间。当存活时间小于一个特定的时长时将会认为这个壁纸的软件质量不可靠从而选择使用默认壁纸，而不是重启这个壁纸服务 */ wallpaper.lastDiedTime = SystemClock.uptimeMillis(); newConn.mReply = reply; /* ⑤ 最后向WMS申请注册一个WALLPAPER类型的窗口令牌。这个令牌会在onServiceConnected()之后被传递给WallpaperService用作后者添加窗口的通行证 */ try { if (wallpaper.userId == mCurrentUserId) { mIWindowManager.addWindowToken(newConn.mToken, WindowManager.LayoutParams.TYPE_WALLPAPER); mLastWallpaper = wallpaper; } } catch (RemoteException e) {} catch (RemoteException e) {} return true; }

bindWallpaperComponentLocked () 主要做了如下几件事情：

- 创建WallpaperConnection。由于实现ServiceConnection接口，因此它将负责监听WallpaperManagerService与壁纸服务之间的连接状态。另外由于继承了IWallpaperConnection.Stub，因此它具有跨进程通信的能力。在壁纸服务绑定成功后，WallpaperConnection实例会被传递给壁纸服务作为壁纸服务与WallpaperManagerService进行通信的桥梁。
- 启动壁纸服务。通过Context.bindService () 方法完成。可见启动壁纸服务与启动一个普通的服务没有什么区别。

- 终止旧有的壁纸服务。
- 将属于当前壁纸的WallpaperConnection实例、componentName机器启动时间戳保存到WallpaperData中。
- 向WMS注册WALLPAPER类型的窗口令牌。这个窗口令牌保存在WallpaperConnection.mToken中，并随着WallpaperConnection的创建而创建。

仅仅将指定的壁纸服务启动起来尚无法使得壁纸显示，因为新启动起来的壁纸服务由于没有可用的窗口令牌而导致其无法添加窗口。

WallpaperManagerService必须通过某种方法将窗口令牌交给壁纸服务才行。所以壁纸显示的后半部分的流程将在

WallpaperConnection.onServiceConnected () 回调中继续。同其他服务一样，WallpaperManagerService会在这个回调之中获得一个Binder对象。因此在进行onServiceConnected () 方法的讨论之前，必须了解WallpaperManagerService在这个回调中将会得到一个什么样的Binder对象。

现在把分析目标转移到WallpaperService中。和普通服务一样，WallpaperService的启动也会经历onCreate () 、 onBind () 这样的生命周期回调。为了了解WallpaperManagerService可以从

onServiceConnected () 获取怎样的Binder对象，需要看下

WallpaperService.onBind () 的实现：

```
[WallpaperService.java-->WallpaperService.onBind()] public final IBinder  
onBind(Intent intent) { /* onBind()新建了一个IWallpaperServiceWrapper  
实例，并将其返回给WallpaperManagerService */ return new  
IWallpaperServiceWrapper(this); }
```

IWallpaperServiceWrapper类继承自IWallpaperService.Stub。它保存了WallpaperService的实例，同时也实现了唯一的一个接口attach ()。很显然，当这个Binder对象返回给Wallpaper-ManagerService之后，后者定会调用这个唯一的接口attach () 以传递显示壁纸所必需的包括窗口令牌在内的一系列参数。

(2) 向壁纸服务传递创建窗口所需的信息

重新回到WallpaperManagerService，当WallpaperService创建IWallpaperServiceWrapper实例并返回后，WallpaperManagerService将会在WallpaperConnection.onServiceConnected () 中收到回调。参考其实现：

```
[WallpaperManagerService.java--  
>WallpaperConnection.onServiceConnected()] public void  
onServiceConnected(ComponentName name, IBinder service) {
```

```
synchronized (mLock) { if (mWallpaper.connection == this) { // 更新壁纸  
    的启动时间戳 mWallpaper.lastDiedTime = SystemClock.uptimeMillis(); //  
    ① 将WallpaperService传回的IWallpaperService接口保存为mService  
    mService = IWallpaperService.Stub.asInterface(service); /* ② 绑定壁纸服  
    务。attachServiceLocked()会调用IWallpaperService.attach()方法 以传递  
    壁纸服务创建窗口所需的信息 */ attachServiceLocked(this, mWallpaper);  
    // ③ 保存当前壁纸的运行状态到文件系统中，以便在系统重启或发生  
    用户切换时可以恢复 saveSettingsLocked(mWallpaper); } } }
```

进一步，attachServiceLocked（）方法会调用IWallpaperService.attach（）方法，将创建壁纸窗口所需的信息发送给壁纸服务。

```
[WallpaperManagerService.java-->WallpaperManagerService.attachServiceCLocked()] void  
attachServiceLocked(WallpaperConnection conn, WallpaperData wallpaper)  
{ try { /* 调用IWallpaperService的唯一接口attach()， 将创建壁纸窗口所  
需要的参数传递给WallpaperService */ conn.mService.attach(conn,  
conn.mToken, WindowManager.LayoutParams.TYPE_WALLPAPER, false,  
wallpaper.width, wallpaper.height); } catch (RemoteException e) {.....} }
```

attach（）方法的参数很多，它们的意义如下：

·conn即WallpaperConnection，WallpaperService将通过它向WallpaperManagerService进行通信。WallpaperConnection继承自IWallpaperConnection，只提供了两个接口的定义，即attachEngine ()以及engineShown ()。虽说WallpaperManager是WallpaperManagerService向外界提供的标准接口，但是这里仍然选择使用WallpaperConnection实现这两个接口的原因是，attachEngine ()以及engineShown ()是只有WallpaperService才需要用到而且是它与WallpaperManagerService之间比较底层且私密的交流，将它们的实现放在通用的接口WallpaperManager中显然不合适。这两个接口中比较重要的当属attachEngine ()了。如前文所述，Engine类是实现壁纸核心所在，而WallpaperService只是一个用于承载壁纸运行的容器而已。因此相对于WallpaperService，Engine是WallpaperManagerService更加关心的对象。所以当WallpaperService完成Engine对象的创建之后，就会通过attachEngine ()方法将Engine对象的引用交给WallpaperManagerService。

·conn.mToken就是在bindWallpaperComponent ()方法中向WMS注册过的窗口令牌。是WallpaperService有权添加壁纸窗口的凭证。

· WindowManager.LayoutParams.TYPE_WALLPAPER指明了WallpaperService需要添加TYPE_WALLPAPER类型的窗口。读者可能会质疑这个参数的意义：壁纸除了是TYPE_WALLPAPER类型以外难道还

有其他的可能吗？的确，在实际壁纸显示中WallpaperService必然需要使用TYPE_WALLPAPER类型添加窗口。但是有一个例外，即壁纸预览。在LivePicker应用中选择一个动态壁纸时，首先会使得用户对选定的壁纸进行预览。这一预览并不是真的将壁纸设置给WallpaperManagerService，而是LivePicker应用自行启动对应的壁纸服务，并要求壁纸服务使用TYPE_APPLICATION_MEDIA_OVERLAY类型创建窗口。这样一来，壁纸服务所创建的窗口将会以子窗口的形式衬在LivePicker的窗口之下，从而实现动态壁纸的预览。

·false的参数名是isPreview，用以指示启动壁纸服务的意图。当被实际用作壁纸时取值为false，而作为预览时则为true。仅当LivePicker对壁纸进行预览时才会使用true作为isPreview的取值。壁纸服务可以根据这一参数的取值对自己的行为进行调整。

当WallpaperManagerService向WallpaperService提供用于创建壁纸窗口的足够信息之后，WallpaperService便可以开始着手创建Engine对象。

(3) Engine的创建

调用IWallpaperService.attach () 是WallpaperManagerService在壁纸服务启动后第一次与壁纸服务进行联系。参考其实现：

```
[WallpaperService.java-->IWallpaperServiceWrapper.attach()]
public void attach(IWallpaperConnection conn, IBinder windowToken, int windowType,
```

```
boolean isPreview, int reqWidth, int reqHeight) { // 使用  
WallpaperManagerService提供的参数构造一个IWallpaperEngineWarapper  
实例 new IWallpaperEngineWrapper(mTarget, conn, windowToken,  
windowType, isPreview, reqWidth, reqHeight); }
```

顾名思义，在attach（）方法中所创建的IWallpaperEngineWrapper将会创建并封装Engine实例。IWallpaperEngineWrapper继承自IWallpaperEngine.Stub，因此它也支持跨Binder调用。在随后的代码分析中可知，它将被传递给WallpaperManagerService，作为WallpaperManagerService与Engine进行通信的桥梁。

另外需要注意的是，attach（）方法的实现非常奇怪，它直接创建一个实例但是并没有将这个实例赋值给某一个成员变量，在attach（）方法结束时岂不是会被垃圾回收？不难想到，在IWallpaperEngineWrapper的构造函数中一定有些动作可以使得这个实例不被释放。参考其实现：

```
[WallpaperService.java--  
>IWallpaperEngineWrapper.IWallpaperEngineWrapper()  
IWallpaperEngineWrapper(WallpaperService context, IWallpaperConnection  
conn, IBinder windowToken, int windowType, boolean isPreview, int  
reqWidth, int reqHeight) { /* 创建一个HandlerCaller。 HandlerCaller是  
Handler的一个封装，而它与Handler的区别是额外提供了一个  
executeOrSend- Message()方法。当开发者在HandlerCaller所处的线程执
```

行此方法时会使得消息的处理函数立刻执行，在其他线程中执行此方法的效果则与Handler.sendMessage()别无二致。除非阅读代码时遇到这个方法，读者只需要将其理解为Handler即可。注意，通过其构造函数的参数可知HandlerCaller保存IWallpaperEngineWrapper的实例 */ mCaller = new HandlerCaller(context, mCallbackLooper != null ? mCallbackLooper : context.getMainLooper(), this); // 保存WallpaperManagerService所提供的参数 mConnection = conn; // conn即WallpaperManagerService中的 WallpaperConnection mWindowToken = windowToken; mWindowType = windowType; mIsPreview = isPreview; mReqWidth = reqWidth; mReqHeight = reqHeight; // 发送DO_ATTACH消息。后续的流程转到 DO_ATTACH消息的处理中进行 Message msg = mCaller.obtainMessage(DO_ATTACH); mCaller.sendMessage(msg); }



注意

在这里貌似并没有保存新建的IWallpaperEngineWrapper实例，它岂不是有可能在DO_ATTACH消息执行前就被Java的垃圾回收机制回收？其实不是这样的。HandlerCaller的构造函数以及最后的sendMessage（）操

作使得这个IWallpaper-EngineWrapper的实例得以坚持到DO_ATTACH消息可以处理的时刻。sendMessage () 方法的调用使得Message被目标线程的MessageQueue引用，并且对应的Handler被Message引用，而这个Handler是HandlerCaller的内部类，因此在Handler中有一个指向HandlerCaller的隐式引用，最后在HandlerCaller中又存在IWallpaperEngineWrapper的引用。因此IWallpaperEngineWrapper间接地被HandlerCaller所在线程的MessageQueue所引用，因此在完成DO_ATTACH消息的处理之前，IWallpaperEngineWrapper并不会被回收。虽然这是建立在对Java引用以及Handler工作原理的深刻理解之上所完成的精妙实现，但是它确实已经接近危险的边缘了。

在这里所创建的mCaller具有十分重要的地位。它是一个重要的线程调度器，所有壁纸相关的操作都会以消息的形式发送给mCaller，然后在IWallpaperEngineWrapper的executeMessage () 方法中得到处理，从而这些操作转移到mCaller所在的线程上进行（如壁纸绘制、事件处理等）。可以说mCaller的线程就是壁纸的工作线程。默认情况下这个mCaller运行在壁纸服务的主线程上即context.getMainLooper ()。不过当WallpaperService.mCallbackLooper不为null时会运行在mCallbackLooper所在的线程。mCaller运行在壁纸服务的主线程上听起来十分合理，然而提供手段以允许其运行在其他线程的做法却有些意外。其实这是为了满足一种特殊的需求，以ImageWallper壁纸服务为例，它是SystemUI的一部分而SystemUI的主线程主要用来作为状态

栏、导航栏的管理与绘制的场所，换句话说其主线程的工作已经比较繁重了。因此ImageWallpaper可以通过这一手段将壁纸的工作转移到另外一个线程中进行。不过因为这一机制可能带来同步上的问题，因此在Android 4.4及后续版本中被废除了。

接下来分析DO_ATTACH消息的处理：

[WallpaperService.java-->IWallpaperEngineWrapper.executeMessage()]

```
public void executeMessage(Message message) { switch (message.what) { case DO_ATTACH: { try { /* ① 把IWallpaperEngineWrapper实例传递给 WallpaperConnection进行保存。至此这个实例便名花有主，再也不用担心被回收了，而且WallpaperManagerService还可以通过它与实际的 Engine进行通信 */ mConnection.attachEngine(this); } catch (RemoteException e) {} /* ② 通过onCreateEngine()方法创建一个 Engine。onCreateEngine()是定义在WallpaperService中的一个抽象方法。WallpaperService的实现者需要根据自己的需要返回一个自定义的 Engine子类 */ Engine engine = onCreateEngine(); mEngine = engine; /* ③ 将新建的Engine添加到WallpaperService.mActiveEngines列表中。读者可能会比较奇怪，为什么是列表？难道一个Wallpaper可能会有多个 Engine？这个奇怪之处还是壁纸预览所引入的。当壁纸A已经被设置为当前壁纸时，系统中会存在一个它所对应的WallpaperService，以及在其内部会存在一个Engine。此时当LivePicker或其他壁纸管理工具预览壁
```

纸A时，它所对应的WallpaperService 仍然只有一个，但是在其内部会变成两个Engine。这一现象更能说明，WallpaperService仅仅是提供壁纸运行的场所，而Engine才是真正壁纸 的实现 */

```
mActiveEngines.add(engine); // ④ 最后engine.attach()将会完成窗口的创建、第一帧的绘制等工作 engine.attach(this); return; } } }
```

正如前面所述，作为拥有跨Binder调用的IWallpaperEngineWrapper通过attachEngine () 方法将自己传递给WallpaperConnection，后者将其保存在WallpaperConnection.mEngine成员中。从此，WallpaperManagerService便可以通过WallpaperConnection.mEngine与壁纸服务进程中的IWallpaperEngineWrapper进行通信，而IWallpaperEngineWrapper进一步将来自WallpaperManagerService的请求或设置转发给Engine对象，从而实现WallpaperManager-Service对壁纸的控制。

到目前为止，WallpaperManagerService与壁纸服务之间已经出现了三个用于跨Binder通信的对象。它们分别是：

- IWallpaperService，实现在壁纸服务进程之中，它所提供的唯一方法attach () 用于在壁纸服务启动后接收窗口创建所需的信息，或者说用于完成壁纸的初始化工作。除此之外IWallpaperService不负任何功能，WallpaperManagerService对壁纸进行的请求与设置都交由在attach () 的过程中所创建的IWallpaperEngineWrapper实例完成。

·WallpaperConnection，实现在WallpaperManagerService中，并通过IWallpaper-Service.attach () 方法传递给了IWallpaperEngineWrapper。壁纸服务通过Wallpaper-Connection的attachEngine () 方法将IWallpaperEngineWrapper实例传递给Wallpaper-ManagerService进行保存。另外壁纸服务还通过它的engineShown () 方法将壁纸显示完成的事件通知给WallpaperManagerService。

·IWallpaperEngineWrapper，实现在壁纸进程中。Engine实例是壁纸实现的核心所在。作为Engine实例的封装者，它是WallpaperManagerService对Engine进行请求或设置的唯一接口。

总体来说，IWallpaperService与WallpaperConnection主要服务于壁纸的创建阶段，而IWallpaperEngineWrapper则用于在壁纸的运行阶段对Engine进行操作与设置。



说明

按照常规的思想来推断，WallpaperManagerService与WallpaperService之间应该仅仅需要IWallpaperService提供接口对壁纸进行操作与设置。为什么要增加一个IWallpaperEngineWrapper呢？这得从WallpaperService

与Engine之间的关系说起。IWall-paperService在WallpaperManagerService看来表示的是WallpaperService，而IWall-paperEngineWrapper则表示的是Engine。WallpaperService是Engine运行的容器，因此它所提供的唯一方法attach () 用来在WallpaperService中创建新的Engine实例（由创建一个IWallpaperEngineWrapper实例来完成）。Engine则是壁纸的具体实现，因此IWallpaperEngineWrapper所提供的方法用来对壁纸进行操作与设置。从这个意义上来说，IWallpaperService与IWallpaperEngineWrapper同时存在是合理的。另外，将IWallpaperService与IWallpaperEngineWrapper分开还有简化实现的意义。从DO_ATTACH消息的处理过程可知，WallpaperService中可以同时运行多个Engine实例。而WallpaperManagerService或LivePicker所关心的只是某一个Engine，而不是WallpaperService中的所有Engine，因此相对于使用IWallpaperService的接口时，必须在参数中指明所需要操作的Engine，直接操作IWallpaperEngineWrapper更加简捷直接。

Engine创建完毕之后会通过Engine.attach () 方法完成Engine的初始化工作。参考其代码：

```
[WallpaperService.java-->Engine.attach()]
void attach(IWallpaperEngineWrapper wrapper) { ..... // 保存必要的信息
    mIWallpaperEngine = wrapper; mCaller = wrapper.mCaller; mConnection =
    wrapper.mConnection; mWindowToken = wrapper.mWindowToken; /* ①
```

mSurfaceHolder是一个BaseSurfaceHolder类型的内部类的实例。 Engine 对其进行简单定制。开发者可以通过mSurfaceHolder定制所需要的 Surface类型 */ mSurfaceHolder.setSizeFromLayout(); mInitializing = true; // 获取WindowSession,用于与WMS进行通信 mSession = WindowManagerGlobal.getWindowSession(getMainLooper()); // mWindow 是IWindow的实现，窗口创建之后它将用于接收来自WMS的回调 mWindow.setSession(mSession); // Engine需要监听屏幕状态。这是为了 保证在屏幕关闭之后，动态壁纸可以停止动画的渲染以节省电量 mScreenOn = ((PowerManager)getSystemService(Context.POWER_SERVICE)).isScreen On(); IntentFilter filter = new IntentFilter(); filter.addAction(Intent.ACTION_SCREEN_ON); filter.addAction(Intent.ACTION_SCREEN_OFF); registerReceiver(mReceiver, filter); /* ② 调用Engine.onCreate()。 Engine 的子类往往需要重写此方法以修改mSurfaceHolder的属性，如像素格 式、尺寸等。注意此时尚未创建窗口，在这里所设置的SurfaceHolder 的属性将会在创建窗口时生效 */ onCreate(mSurfaceHolder); mInitializing = false; mReportedVisible = false; /* ③ 最后updateSurface将会根据 SurfaceHolder的属性创建窗口以及Surface，并进行壁纸的第一次绘制 */ updateSurface(false, false, false); }

`Engine.attach ()` 方法执行的结束标志着壁纸启动工作完成，至此在最后的`updateSurface ()` 方法结束之后新的壁纸便显示出来。

(4) 壁纸的创建流程

可见，壁纸的创建过程比较复杂。在这个过程中存在着多个Binder对象之间的互相调用。因此有必要对此过程进行简单整理：

- 首先，壁纸管理程序（如LivePicker）调用
`IWallpaperManager.setWallpaperComponent ()` 要求
WallpaperManagerService设置指定的壁纸。
- WallpaperManagerService通过调用`bindWallpaperComponentLocked ()` 启动给定的壁纸服务。同时会终止旧有的壁纸服务。
- WallpaperManagerService成功连接壁纸服务后，调用壁纸服务的`attach ()` 方法将窗口令牌等参数交给壁纸服务。
- 壁纸服务响应`attach ()` 的调用，创建一个`Engine`。
- `Engine`的`updateSurface ()` 方法将会创建壁纸窗口及Surface，并绘制壁纸。

而在这个过程中，WallpaperManagerService中存在如下重要的数据结构：

·WallpaperInfo，存储动态壁纸的开发者、缩略图与描述信息。这个数据结构创建于WallpaperManagerService.bindWallpaperComponentLocked()方法，其内容来自壁纸所在应用程序的AndroidManifest.xml中名为 android.service.wallpaper的meta-data。

·WallpaperConnection，它不仅仅是壁纸服务与WallpaperManagerService进行通信的渠道，它同时也保存与壁纸服务相关的重要运行时信息，如IWallpaperService、IWallpaperEngineWrapper、WallpaperInfo以及用于创建窗口所需的窗口令牌。WallpaperConnection创建于WallpaperManagerService.bindWallpaperComponentLocked()方法。

·WallpaperData，它保存一个壁纸在WallpaperManagerService中可能用到的所有信息，包括壁纸服务的ComponentName、WallpaperConnection、壁纸服务的启动时间等。WallpaperData被保存在一个名为mWallpaperMap的SparseArray中，而且设备中每一个用户都会拥有一个固定的WallpaperData实例。当前用户进行壁纸切换时会更新WallpaperData的内容，而不是新建一个WallpaperData实例。另外，WallpaperData中还保存与静态壁纸相关的一些信息，关于静态壁纸的内容将在8.3节介绍。

壁纸的创建过程同时体现了壁纸服务与WallpaperManagerService之间的关系，如图8-1所示。

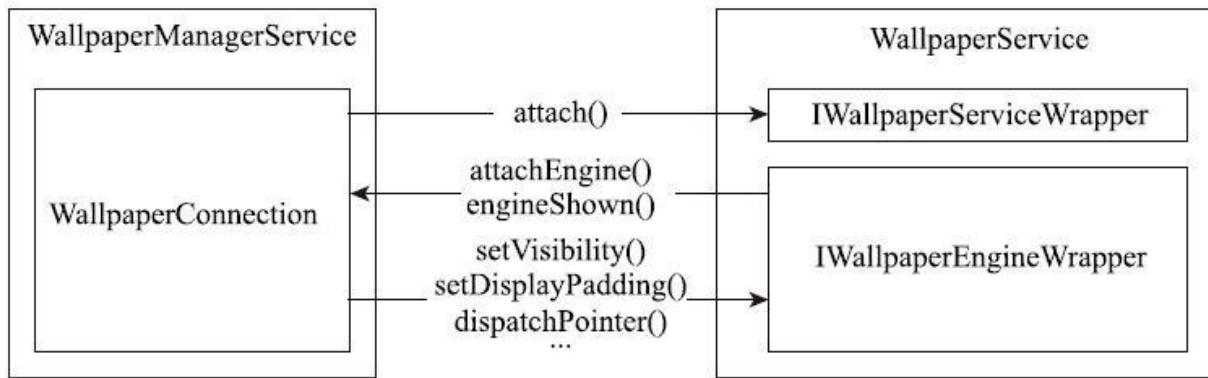


图8-1 壁纸服务与WallpaperManagerService之间的关系

8.2.3 理解UpdateSurface () 方法

Engine.attach () 最后所调用的Engine.updateSurface () 方法是Engine所提供的壁纸框架的核心所在。此方法的意义对壁纸来说，就好比performLayoutAndPlaceSurfacesLocked () 对WMS以及performTraversals () 对控件树一样。updateSurface () 的主要目的是将SurfaceHolder中的设置以及窗口属性设置（如窗口的flag）过WMS的relayoutWindow () 同步到Surface与窗口。在这个过程中Surface的状态发生变化时会触发SurfaceHolder相关的回调。另外，当壁纸窗口尚未创建或Surface尚未创建时，updateSurface () 会通过WMS的接口进行窗口或Surface的创建。

updateSurface () 方法比较大，本节按照其工作内容将其划分为几个部分进行讨论。updateSurface () 中既然包含了许多的工作，那么必须根

据某些条件确定这些工作是否真的有必要做，以避免浪费CPU资源。

参考如下代码：

```
[WallpaperService.java-->Engine.updateSurface()] void  
updateSurface(boolean forceRelayout, boolean forceReport, boolean  
redrawNeeded) { ..... /* 获取mSurfaceHolder中所保存的尺寸。倘若这一  
尺寸为默认情况下的(-1,-1)，则updateSurface() 会认为其表示的尺寸为  
MATCH_PARENT */ int myWidth = mSurfaceHolder.getRequestedWidth();  
if (myWidth <= 0) myWidth =  
    ViewGroup.LayoutParams.MATCH_PARENT; int myHeight =  
    mSurfaceHolder.getRequestedHeight(); if (myHeight <= 0) myHeight =  
    ViewGroup.LayoutParams.MATCH_PARENT; // 下面的一组变量是更新  
surface的条件。 // ① 窗口尚未创建 final boolean creating = !mCreated; //  
② Surface尚未创建 final boolean surfaceCreating = !mSurfaceCreated; // ③  
mSurfaceHolder中的像素格式发生了变化 final boolean formatChanged =  
    mFormat != mSurfaceHolder.getRequestedFormat(); // ④ 尺寸发生变化  
boolean sizeChanged = mWidth != myWidth || mHeight != myHeight; // ⑤  
Surface的类型发生了变化 final boolean typeChanged = mType !=  
    mSurfaceHolder.getRequestedType(); // ⑥ 窗口的Flag发生了变换 final  
boolean flagsChanged = mCurWindowFlags != mWindowFlags ||  
    mCurWindowPrivateFlags != mWindowPrivateFlags; // 只要以上条件满足  
其一便有必要更新Surface if (forceRelayout || creating || surfaceCreating ||
```

```
formatChanged || sizeChanged || typeChanged || flagsChanged ||  
redrawNeeded || !mIWallpaperEngine.mShownReported) { ..... // 壁纸窗口  
的创建、重新布局以及在必要时触发SurfaceHolder的回调 } }
```

从这些条件中可以看到updateSurface () 方法可能进行的工作如下：

- 创建壁纸窗口，由mCreated成员指定。
- 从WMS申请Surface，由mSurfaceCreated成员指定。
- 修改Surface的像素格式，由SurfaceHolder.getRequestedFormat () 的返回值指定。
- 修改Surface的尺寸，由SurfaceHolder.getRequestedWidth () /Height () 的返回值指定。
- 修改Surface内存的类型，即NORMAL、GPU、HARDWARE以及PUSH_BUFFERS。由SurfaceHolder.getRequestedType () 的返回值指定。
- 修改窗口的flags，由mWindowFlags成员指定。对壁纸窗口来说，窗口flags的变化主要是由于Engine.setTouchEventsEnabled () 方法增加或删除了FLAG_NOT_TOUCHABLE标记。

在updateSurface () 后续的代码中将会看到这些变量如何对Surface产生影响。

```
[WallpaperService.java-->Engine.updateSurface()] void  
updateSurface(boolean forceLayout, boolean forceReport, boolean  
redrawNeeded) { ..... // Surface更新条件的计算 if (forceLayout ||  
creating || surfaceCreating || formatChanged || sizeChanged || typeChanged ||  
flagsChanged || redrawNeeded || !mIWallpaperEngine.mShownReported) {  
try { /* 将SurfaceHolder中的设置转储到Engine的成员变量中。用于在下  
次updateSurface() 的调用中检查它们是否发生变化 */ mWidth =  
myWidth; mHeight = myHeight; mFormat =  
mSurfaceHolder.getRequestedFormat(); mType =  
mSurfaceHolder.getRequestedType(); // ① 更新窗口的LayoutParams。上述  
的像素格式、尺寸、内存类型以及窗口flags会使用 LayoutParams经过  
窗口的重新布局以完成设置 mLAYOUT.x = 0; mLAYOUT.y = 0;  
mLAYOUT.width = myWidth; mLAYOUT.height = myHeight; .....  
mLAYOUT.flags = mWindowFlags |  
 WindowManager.LayoutParams.FLAG_LAYOUT_NO_LIMITS |  
 WindowManager.LayoutParams.FLAG_LAYOUT_IN_SCREEN |  
 WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE ; ..... //  
mWindowToken来自WallpaperManagerService的WallpaperConnection  
mLAYOUT.token = mWindowToken; // ② 倘若壁纸窗口尚未创建，则进行
```

窗口创建 if (!mCreated) { // 窗口的类型来自
IWallpaperEngineWrapper.attach()方法 mLayout.type =
mIWallpaperEngine.mWindowType; mInputChannel = new
InputChannel(); // 创建窗口，注意壁纸窗口会被添加到
DEFAULT_DISPLAY中 if (mSession.addToDisplay(mWindow,
mWindow.mSeq, mLayout, View.VISIBLE, Display.DEFAULT_DISPLAY,
mContentInsets, mInputChannel) < 0) { return; // 创建窗口失败就直接返
回 } mCreated = true; // 标记窗口已经创建完成 // 创建InputEventReceiver
实例用于接收触摸事件 mInputEventReceiver = new
WallpaperInputEventReceiver(mInputChannel, Looper.myLooper()); } /*
接下来的操作会修改Surface，因此必须将SurfaceHolder锁住，以免其他
线程在这个过程中尝试通过SurfaceHolder.lockCanvas()修改Surface的内
容 */ mSurfaceHolder.mSurfaceLock.lock(); mDrawingAllowed = true; /*
③ 重新布局窗口。它将SurfaceHolder中的设置以及窗口属性同步到
Surface及其窗口值中。倘若壁纸窗口刚刚完成创建，则经过重新布局
后其Surface也会变得有效 */ final int relayoutResult = mSession.relayout(
mWindow, mWindow.mSeq, mLayout, mWidth, mHeight, View.VISIBLE,
0, mWinFrame, mContentInsets, mVisibleInsets, mConfiguration,
mSurfaceHolder.mSurface); /* 尽管SurfaceHolder的设置给出期望的尺
寸，但是WMS拥有决定窗口最终尺寸的权利。因此同
ViewRootImpl.performTraversals()中的行为一样，updateSurface()将

```
WMS的布局结果设置给SurfaceHolder，强迫其接受这个结果 */ int w =  
mWinFrame.width(); if (mCurWidth != w) { sizeChanged = true;  
mCurWidth = w; } int h = mWinFrame.height(); if (mCurHeight != h) {  
sizeChanged = true; mCurHeight = h; }  
mSurfaceHolder.setSurfaceFrameSize(w, h); // Surface更新成功，解除对  
SurfaceHolder的锁定 mSurfaceHolder.mSurfaceLock.unlock(); ..... // 触发  
SurfaceHolder的回调 } catch (RemoteException ex) { ..... } } }
```

这部分updateSurface () 的代码完成Surface更新。其原理十分简单：

- 倘若窗口尚未创建，则通过WMS.addWindow () 完成窗口的创建。
- 通过WMS.relayoutWindow () 对窗口进行重新布局。重新布局的结果是倘若窗口尚没有一块可用的Surface（如在这个方法中刚刚完成创建工作），Engine将会拥有一块可用的Surface。另外，存储在LayoutParams中与Surface或窗口有关的参数都会被WMS接纳并据此修改Surface的属性（尺寸、像素格式、内存类型以及窗口的flags）。

因此updateSurface () 会使得Surface发生创建或者改变，而这些变化正是SurfaceHolder的使用者所关心的。在完成Surface的更新工作后，updateSurface () 会触发SurfaceHolder的回调以通知所有SurfaceHolder的使用者（即由开发者所实现的Engine的子类）。参考updateSurface () 的后续代码：

[WallpaperService.java-->Engine.updateSurface()] void updateSurface(boolean forceLayout, boolean forceReport, boolean redrawNeeded) { // Surface更新条件的计算 if (forceLayout || creating || surfaceCreating || formatChanged || sizeChanged || typeChanged || flagsChanged || redrawNeeded || !mIWallpaperEngine.mShownReported) { try { // 进行壁纸窗口的创建以及重新布局的工作 try { mSurfaceHolder.ungetCallbacks(); if (surfaceCreating) { /* ① 发起 onSurfaceCreated() 回调。Engine 将会收到这一回调，并对即将到来的绘制工作做好初始化工作 onSurfaceCreated(mSurfaceHolder); // 将创建 Surface 的消息通知给 SurfaceHolder 的同名回调 } redrawNeeded |= creating || (layoutResult & WindowManagerGlobal.RELAYOUT_RES_FIRST_TIME) != 0; if (forceReport || creating || surfaceCreating || formatChanged || sizeChanged) { /* ② 发起 onSurfaceChanged() 回调。Engine 将会收到这一回调，并根据新的尺寸或 像素格式等信息对即将到来的绘制工作做好必要的准备工作 */ onSurfaceChanged(mSurfaceHolder, mFormat, mCurWidth, mCurHeight); // 将 Surface 属性发生改变的消息通知给 SurfaceHolder 的同名回调 } if (redrawNeeded) { /* ③ 发起 onSurfaceRedrawNeeded() 回调。Engine 将会收到这一回调，并绘制壁纸 */ onSurfaceRedrawNeeded(mSurfaceHolder); // 将 Surface 需要重绘的消息通知给 SurfaceHolder 的同名回调 } } finally { mIsCreating = false;

```
mSurfaceCreated = true; /* 将绘制完毕的消息通知WMS。相信读者经过  
第4章以及第6章的学习之后对这个方法的作用已经非常熟悉。它是显  
示新建窗口的必要条件 */ if (redrawNeeded) {  
    mSession.finishDrawing(mWindow); } // 将绘制完毕的消息通知  
WallpaperManagerService mIWallpaperEngine.reportShown(); } } catch  
(RemoteException ex) { ..... } } }
```

总体来说， Engine.updateSurface () 方法的工作主要有两个部分：

·根据SurfaceHolder的设置创建或更新Surface。 SurfaceHolder中的设置被开发者修改时， WallpaperService会接受来自SurfaceHolder的回调，并通过调用updateSurface () 方法使新的设置生效。 SurfaceHolder允许修改的设置有尺寸、像素格式、内存类型，以及是否防止屏幕休眠4种。然而，除非有特殊需求，像素格式以及内存类型两种设置一般在Surface创建后便不会发生变化。至于防止屏幕休眠，很显然壁纸的Surface并不适用，因此当对壁纸的SurfaceHolder尝试修改这一设置时会引发一个异常。尺寸的设置比较特殊， SurfaceHolder提供了两个接口可以对Surface的尺寸进行修改，它们是setFixedSize (int w, int h) 和setSizeFromLayout () 。前者可以接受指定的宽高作为参数并保存，然后在Engine.updateSurface () 中使用这一宽高布局窗口进而修改Surface的尺寸。而后者不接受任何参数，而是将SurfaceHolder中保存的宽高复位为-1，然后在Engine.updateSurface () 中使用默认的宽高——如代码

中所示的MATCH_PARENT——进行窗口的布局。简而言之，在壁纸这一用例中，SurfaceHolder.setFixedSize () 用于直接设置壁纸的尺寸，而SurfaceHolder.setSizeFromLayout () 则是将壁纸的尺寸恢复到默认的MATCH_PARENT，即充满屏幕。遗憾的是，并不是所有的壁纸应用都可以使用setFixedSize () 方法设置Surface的尺寸，因为这种方式实在太过自由。因此Android只允许内置ImageWallpaper使用这一方法。另外，Engine.setTouchEventsEnabled () 会在LayoutParams.flags中添加或删除FLAG_NOT_TOUCHABLE标记，以允许或禁止壁纸窗口接受触摸事件。这一设置同样需要updateSurface () 方法得以生效。

·引发与Surface有关的各种回调。这些回调包括onSurfaceCreated () 、onSurfaceChanged () 和onSurfaceRedrawNeeded () 。Engine还有另外一个会在壁纸被销毁时触发回调onSurfaceDestroyed () 。这些回调从Surface的角度向开发者提供了关于壁纸的生命周期相关的信息。开发者可以通过对这些回调做出响应从而保证正确显示壁纸。

8.2.4 壁纸的销毁

在8.2.2节介绍WallpaperManagerService.bindWallpaperComponentLocked () 方法时提到过，当WallpaperManagerService启动新的壁纸时，会通过detachWallpaperLocked () 方法销毁之前的壁纸。壁纸的销毁工作要比壁纸的启动简单得多。启动壁纸时的工作主要有注册窗口令牌、启

动对应的壁纸服务以及IWallpaperService.attach () 三个步骤。那么销毁工作自然是将这三个动作反过来执行。参考其代码实现：

[WallpaperManagerService.java--

```
>WallpaperManagerService.detachWallpaperLocked()] void  
detachWallpaperLocked(WallpaperData wallpaper) { if  
(wallpaper.connection != null) { ..... /* ① 执行IWallpaperEngine.destroy()  
方法。这个方法会在壁纸服务的进程中触发onSurface- Destroyed()回调  
以通知开发者销毁壁纸运行期间所使用的资源，并移除壁纸窗口 */ if  
(wallpaper.connection.mEngine != null) { try {  
wallpaper.connection.mEngine.destroy(); } catch (RemoteException e) {} }  
// ② unbindService()将终止WallpaperManagerService对壁纸服务的绑定  
mContext.unbindService(wallpaper.connection); /* ③ 注销用来添加壁纸窗  
口的令牌。即便被销毁的壁纸服务所在的进程用某种方法（比如说反  
射）偷偷保存了一份窗口令牌，以期能够在壁纸销毁之后强行进行壁  
纸窗口的创建是徒劳的。因为在壁纸被 销毁之后，它所保有的窗口令  
牌也就变得无效了 */ try {  
mIWindowManager.removeWindowToken(wallpaper.connection.mToken); }  
catch (RemoteException e) {} ..... } }
```

而在壁纸服务的进程收到Engine.destroy () 的调用之后会从
mActiveEngines列表中将当前Engine删除，然后调用Engine.detach ()

方法：

```
[WallpaperService.java-->Engine.detach()] void detach() { ..... // ①  
reportSurfaceDestroyed()会触发onSurfaceDestroyed()回调  
reportSurfaceDestroyed(); // ② 触发onDestroy()回调 onDestroy(); if  
(mCreated) { try { ..... // ③ 移除壁纸窗口 mSession.remove(mWindow); }  
catch (RemoteException e) {.....} ..... } }
```

从壁纸开发者的实现而言，壁纸的销毁过程会触发onSurfaceDestroyed () 以及onDe-stroyed () 两个回调以通知开发者进行相关资源的销毁。

8.2.5 理解Engine的回调

在分析壁纸的启动、销毁以及updateSurface () 的过程中遇到以下几个 Engine类的回调：

- onCreate () ，此回调发生在Engine.attach () 方法中，表示Engine生命周期的开始。
- onDestroy () ，此回调发生在Engine.detach () 方法中，表示Engine生命周期的结束。
- onSurfaceCreated () /onSurfaceChanged () /onSurfaceDestroyed () /onSurfaceRedraw-Needed () 4个回调发生在updateSurface () 以

及Engine.detach () 方法中。重写这些回调与在SurfaceHolder中注册Callback是等效的。它们向壁纸开发者描述了Surface的生命周期以及状态变化。

另外，Engine中还有一些上文并未提及的重要回调。

onVisibilityChanged () ，通知Engine当前壁纸的可见性发生了变化。读者可能会很自然地将这个可见性与壁纸窗口的可见性联系到一起。其实它们是两种不同的概念。窗口可见性表示的是窗口是否拥有一块可供绘制的Surface。而从这个意义上讲，只要WallpaperManagerService启动一个壁纸并且创建窗口，那么在壁纸被销毁之前此窗口永远是可见的，这一点从updateSurface () 方法的实现中可以看到。那么onVisibilityChanged () 的可见性所指为何呢？这得从WMS对壁纸窗口的优化说起。壁纸窗口从用户眼中所看到的形式并不像是一个窗口，它更像是另一个窗口的背景。是否以壁纸窗口作为其背景取决于窗口是否在其flags中指定FLAG_SHOW_WALLPAPER标记。于是当指定此标记的窗口与未指定此标记的窗口交替显示时（例如在HOME应用与短信应用之间切换时），壁纸窗口需要随着指定此标记的窗口在可见与不可见状态之间切换。通过第4章的学习可以知道，使窗口从不可见变为可见并不是一件简单的事情。它经历了窗口的重新布局、Surface的创建、等待窗口完成第一次绘制等工作。其中窗口完成第一次绘制的开销对壁纸窗口来说尤为巨大，因为从存储设备中载

入并显示一个高分辨率的图片所导致的延迟是很大的。这会使得壁纸窗口的显示很容易滞后于指定此标记的窗口的显示。在这个滞后的时段内，用户透过窗口的内容所看到的可能是一个黑乎乎的背景，或者是上一个应用程序的界面。为了解决这个问题，Android索性让壁纸窗口永远处于可见状态，只是当WMS认为壁纸窗口不应被显示时，将其藏在所有窗口的最后面使用户看不到。而当一个指定FLAG_SHOW_WALLPAPER标记的窗口处于前台显示时，再将壁纸窗口移动到这一窗口的下方使得用户可以看到它。这样一来便避免了壁纸窗口初次显示的时间开销。不过这样一来又会引入新的问题，动态壁纸往往是一个动画，它会不断地刷新其显示内容从而不断占用CPU的资源。当WMS将壁纸窗口隐藏于所有窗口底部时，这种资源占用既不必要又很浪费。于是，onVisibilityChanged () 回调应运而生。onVisibilityChanged () 并不是表示WMS销毁或创建壁纸窗口的Surface，而是表示WMS将壁纸窗口衬于某一个窗口下方作为其背景，或是将壁纸窗口藏在所有窗口之下使得用户看不到。所以当onVisibilityChanged () 报告壁纸的可见性为false时，Engine必须停止一切消耗资源的操作，反之Engine应当全力工作以讨好用户的眼球。



注意

当WMS将壁纸窗口置于其他窗口底部时，它会调用Surface.hide () 方法将壁纸的Surface隐藏。这种隐藏相对于通过relayoutWindow () 使窗口变得不可见来说非常轻量。它不会导致释放Surface或者清除Surface中已经被绘制的内容。当需要显示壁纸时会调用Surface.show () 方法，由于Surface中已绘制的内容并没有被清除，所以不需要重绘Engine对象。

onTouchEvent () ，用于处理用户的触摸事件。开发者可以通过调用Engine.setTouch-EventsEnabled () 方法在壁纸窗口的flags中增加或删除FLAG_NOT_TOUCHABLE标记，从而设置壁纸窗口是否可以接收用户的输入事件。壁纸窗口可以接受触摸事件这个事实有一些匪夷所思，因为InputDispatcher查找触摸事件的派发目标的基本算法是这样的：它会沿着窗口的Z序从上向下遍历查找触摸事件落在其上的第一个窗口，并将这个窗口作为派发目标。很显然，要求显示壁纸的窗口A位于壁纸窗口之上，那么窗口A自然会阻挡壁纸窗口作为触摸事件的派发目标。据此推断，壁纸窗口岂不是永远都无法接受触摸事件，进而这个onTouchEvent () 回调便成了无用的摆设？其实不然，这得益于InputDispatcher为壁纸窗口做了一个小小的特殊化处理。在InputDispatcher :: findTouchedWindowTargetsLocked () 函数通过上述策略找到触摸事件的派发目标窗口之后，会检查这个窗口是否要求显

示壁纸，如果是，则会在窗口列表中寻找类型为TYPE_WALLPAPER的壁纸窗口，并将其作为一个额外的派发目标添加到派发目标窗口列表中，并在随后将触摸事件同时发送给这两个窗口。

onOffsetChanged ()，用于通知Engine对象需要对壁纸所绘制的内容进行偏移。这个回调引发于WallpaperManager.setWallpaperOffsets ()。当要求显示壁纸的窗口的内容发生滚动时，它可能希望壁纸的内容也能随之偏移。一个典型的例子就是HOME应用，当用户在主界面中拖动图标时，HOME应用会通过WallpaperManager.setWallpaperOffsets () 告知壁纸服务中的Engine对象需要进行偏移的量，而这个偏移量是一个0到1的浮点值。如何表现这个偏移量在Android中并没有给出一个标准，最传统的做法是类似于静态壁纸那样对绘制内容进行滚动。不过壁纸的开发者可以根据自己的创意自由发挥，例如将偏移量体现为颜色的变化、粒子效果的变化等。

onCommand ()，用于处理来自WallpaperManager.sendWallpaperCommand () 的一个命令。Android并没有规定sendWallpaperCommand () 将会发送什么样的命令给engine。开发者可以根据需要自定义命令。换句话说，WallpaperManager.sendWallpaperCommand () 与onCommand () 回调共同为开发者提供了一种通信方式。

`onDesiredSizeChanged ()`，用于处理来自
`WallpaperManager.suggestDesiredDimensions ()` 方法的请求。
`suggestDesiredDimensions ()` 由显示壁纸的窗口调用，用于指明它所
期望的壁纸的尺寸。例如，当一个窗口中所显示的所有内容的宽度为
2000、高度为3000时，它可以通过这个方法尝试将壁纸的宽度与高度
分别设置为2000与3000，从而当用户对内容进行拖动时，可以在
`setWallpaperOffsets ()` 方法的帮助下使得壁纸与内容同步滚动。

这些回调可以理解为Android特别为壁纸规定的一套协议。壁纸实现者
可以在这些回调的驱动下完成对壁纸的实现。

8.3 深入理解静态壁纸——ImageWallpaper

本节将会对静态壁纸进行介绍。所谓的静态壁纸是运行于SystemUI进程中的一名名为ImageWallpaper的特殊的动态壁纸而已。首先它是一个动态壁纸，因为它是基于前文所述的动态壁纸的架构实现的。而说它是静态的，是因为它的内容是一张静态的图片。之所以将它区别于其他众多内置的动态壁纸，是因为Android为这个静态壁纸提供了特殊的API以及实现机制，使得开发者可以方便地将一张静态图片设置为壁纸。

8.3.1 获取用作静态壁纸的位图

静态壁纸所对应的壁纸服务实现于SystemUI之中，它定义了一个继承自Wallpaper-Service的服务ImageWallpaper作为其运行容器，并在其onCreateEngine () 方法中创建了一个继承自Engine的DrawableEngine对象作为其Engine实现。

在DrawableEngine中有一个Bitmap类型的成员变量mBackground。这个mBackground即作为壁纸的位图。每当需要对静态壁纸进行重绘时，DrawableEngine会通过调用draw-FrameLocked () 方法选择通过软件方式或硬件加速的方式将这个mBackground绘制到壁纸窗口的Surface上。

这一切都很简单，读者可以自行对这个过程进行研究。然而需要讨论的问题是，mBack-ground从哪里来呢？纵览ImageWallpaper的代码可以发现，mBackground的赋值位于DrawableEngine.updateWallpaperLocked()方法之中。参考其代码：

```
[ImageWallpaper.java-->DrawableEngine.updateWallpaperLocked()]
private void updateWallpaperLocked() { Throwable exception = null; try { // ① 忘记已经加载的壁纸 mWallpaperManager.forgetLoadedWallpaper();
    mBackground = null; // ② 从WallpapaerManager.getBitmap()方法中获取作为壁纸的位图 mBackground = mWallpaperManager.getBitmap(); } catch (RuntimeException e) { exception = e; } } .....
```

此时理解WallpaperManager.forgetLoadedWallpaper()方法的意义可能比较困难，因此，首先讨论WallpaperManager.getBitmap()的实现。

```
[WallpaperManager.java-->WallpaperManager.getBitmap()]
public Bitmap getBitmap() { /* sGlobals是一个Globals类型的进程唯一的静态成员，它是WallpaperManager与WallpaperManagerService通信的代理人。其中true表示当位图加载失败时是否返回默认的壁纸位图 */ return sGlobals.peekWallpaperBitmap(mContext, true); }
```

进一步分析Globals.peekWallpaperBitmap()的实现。

```
[WallpaperManager.java-->Globals.peekWallpaperBitmap()] public Bitmap  
peekWallpaperBitmap(Context context, boolean returnDefault) {  
    synchronized (this) { // ① 倘若已加载了壁纸位图，则返回 if  
        (mWallpaper != null) { return mWallpaper; } // ② 倘若没有加载壁纸位  
        图，则返回已加载的默认壁纸位图 if (mDefaultWallpaper != null) {  
            return mDefaultWallpaper; } mWallpaper = null; try { // ③ 通过  
            getCurrentWallpaperLocked()方法获取壁纸位图，并缓存在mWallpaper  
            中 mWallpaper = getCurrentWallpaperLocked(context); } catch  
            (OutOfMemoryError e) {.....} if (returnDefault) { if (mWallpaper == null) {  
                // ④ 倘若壁纸位图加载失败，则加载默认的壁纸位图并返回  
                mDefaultWallpaper = getDefaultWallpaperLocked(context); return  
                mDefaultWallpaper; } else { // 倘若壁纸位图加载成功，则释放已加载的  
                默认位图 mDefaultWallpaper = null; } } return mWallpaper; } }
```

peekWallpaperBitmap () 方法中出现了两个比较重要的成员：
mWallpaper以及mDefault-Wallpaper。在二者都未被加载时，
peekWallpaperBitmap () 会首先尝试通过getCurrentWall-paperLocked
() 方法获取用作壁纸的位图并保存在mWallpaper成员中。于是下次再
调用peekWallpaperBitmap () 时，就会直接返回mWallpaper，从而避免
再次加载位图的开销。然而，倘若位图加载失败（比如说位图尺寸太
大导致OOM），peekWallpaperBitmap () 会通过
getDefaultWallpaperLocked () 方法加载默认位图存储在

mDefaultWallpaper中并返回。那么下次再调用peekWallpaperBitmap () 时，就会直接返回mDefaultWallpaper。简单来说，在这个只有少许代码的peekWallaperBitmap () 中竟提供了两套保证静态壁纸稳定运行的机制：缓存以及备用方案。peekWallpaperBitmap () 会尽可能地返回已经加载的位图，并且当指定的位图加载失败时，将会返回系统默认的位图。

于是WallpaperManager.forgetLoadedWallpaper () 方法的意义也就清楚了。它的实现是将mDefaultWallpaper以及mWallpaper两个成员设置为null，于是随后进行Globals.peekWallpaperBitmap () 调用时便会重新加载位图，而不是返回已经加载过的位图。



说明

getDefaultWallpaperLocked () 尝试通过加载com.android.internal.R.drawable.default_wallpaper资源所描述的位图作为系统的默认壁纸。不过倘若连这个位图的加载也失败了，那么只好返回null。ImageWallpaper在这种最糟糕的情况下会显示为一片黑色。

那么getCurrentWallpaperLocked () 方法如何获取被设置为壁纸的位图呢？参考其实现：

```
[WallpaperManager.java-->Globals.getCurrentWallpaperLocked()]
private
Bitmap getCurrentWallpaperLocked(Context context) { try { Bundle params
= new Bundle(); /* ① 通过WallpaperManagerService.getWallpaper()方法获
取存储了被设置为壁纸的位图文件的描述符fd。其中params将会携带
着位图的尺寸信息返回给Globals */ ParcelFileDescriptor fd =
mService.getWallpaper(this, params); if (fd != null) { int width =
params.getInt("width", 0); int height = params.getInt("height", 0); try {
BitmapFactory.Options options = new BitmapFactory.Options(); //② 将文件
解码为一个位图，此时位图的格式为位图文件中所定义的格式 Bitmap
bm = BitmapFactory.decodeFileDescriptor( fd.getFileDescriptor(), null,
options); //③ 对位图进行转码，使其density、像素格式符合Android系统
的要求 return generateBitmap(context, bm, width, height); } catch
(OutOfMemoryError e) {.....} } } catch (RemoteException e) {...} return
null; }
```

原来，被设置为壁纸的位图来自WallpaperManagerService.getWallpaper () 方法。奇特的是getWallpaper () 方法返回的并不是一个Bitmap，而是一个文件描述符。

那么这个文件描述符又是从何而来呢？再看

WallpaperManagerService.getWallpaper () 方法的实现：

[WallpaperManagerService.java--

```
>WallpaperManagerService.getWallpaper() public ParcelFileDescriptor  
getWallpaper(IWallpaperManagerCallback cb, Bundle outParams) {  
    synchronized (mLock) { /* 正如在动态壁纸的启动过程中所讨论过的那  
样，Android壁纸支持多用户设置，动态壁纸如此，静态壁纸也是如此，所以不同的用户会拥有不同的壁纸位图。因此第一步仍然是鉴定  
调用者的用户身份 */ int callingUid = Binder.getCallingUid(); int  
wallpaperUserId = 0; if (callingUid == android.os.Process.SYSTEM_UID) {  
    // 调用者若为系统代码，则返回当前用户的壁纸位图 wallpaperUserId =  
    mCurrentUserId; } else { // 否则获取调用者所属的用户ID  
    wallpaperUserId = UserHandle.getUserId(callingUid); } // ① 从  
    mWallpaperMap中获取对应用户的WallpaperData WallpaperData  
    wallpaper = mWallpaperMap.get(wallpaperUserId); try { // 从  
    WallpaperData中获取壁纸位图的尺寸信息并保存在参数outParams中 if  
(outParams != null) { outParams.putInt("width", wallpaper.width);  
    outParams.putInt("height", wallpaper.height); } // ② 打开指定位置的一个  
    位图文件 File f = new File(getWallpaperDir(wallpaperUserId),  
    WALLPAPER); if (!f.exists()) { return null; } // ③ 将文件包装为一个文件  
    描述符并返回给调用者 return ParcelFileDescriptor.open(f,
```

```
MODE_READ_ONLY); } catch (FileNotFoundException e) {...} return  
null; } }
```

可见，壁纸位图存储在文件夹getWallpaperDir (wallpaperUserId) 下的WALLPAPER文件中。查看getWallpaperDir () 的实现可知这个文件的完整路径是/data/system/users/ /wallpaper。所以，设置静态壁纸的本质工作就是将壁纸的位图写入上述路径之后，然后ImageWallpaper从这个路径解码这幅位图，并将其绘制在壁纸窗口的Surface之上。



注意

倘若设备启用了加密文件系统，那么存储壁纸位图的路径则变为/data/secure/system/users/ /wallpaper。

8.3.2 静态壁纸位图的设置

于是问题随之而来，是谁将壁纸位图写入这个路径的呢？当这个路径中所保存的位图被更新时ImageWallpaper如何得到通知并重新载入然后重绘呢？这就要从设置静态壁纸的API的实现说起。

不似动态壁纸的设置需要签名级系统权限，任何一个申请了普通权限 android.permission.SET_WALLPAPER 的应用程序都可以通过 WallpaperManager 相关的接口进行静态壁纸的设置。WallpaperManager 提供了 setBitmap()、setResource() 以及 setStream() 这三个方法进行静态壁纸的设置，其区别仅在于接受参数的类型不同，开发者可以根据实际情况自由选择。以 setBitmap() 为例：

```
[WallpaperManager.java-->WallpaperManager.setBitmap()] public void  
setBitmap(Bitmap bitmap) throws IOException { ..... try { // ① 通过  
WallpaperManagerService.setWallpaper()方法获取一个文件描述符  
ParcelFileDescriptor fd = sGlobals.mService.setWallpaper(null); if (fd ==  
null) { return; } FileOutputStream fos = null; try { // ② 将Bitmap写入这个  
文件描述符所指向的文件中 fos = new  
ParcelFileDescriptor.AutoCloseOutputStream(fd);  
bitmap.compress(Bitmap.CompressFormat.PNG, 90, fos); } finally { // ③ 关  
闭文件 if (fos != null) { fos.close(); } } } catch (RemoteException e) { ... }  
}
```

看来，把位图数据写入文件的操作是由进行壁纸设置的进程完成的。WallpaperManager-Service 通过 setWallpaper() 方法为壁纸设置进程提供了一个文件描述符，然后壁纸设置进程将位图数据写入这个描述符中。很显然，通过 setWallpaper() 与 getWallpaper() 两个方法所返回

的文件描述符必然指向同一个文件。所以可以断定 WallpaperManagerService 在静态壁纸的设置过程是以一个中间人的身份存在的。参考 WallpaperManagerService.setWallpaper () 方法的实现以印证这一想法：

[WallpaperManagerService.java--

```
>WallpaperManagerService.setWallpaper() public ParcelFileDescriptor  
setWallpaper(String name) { // 要求调用者必须申请  
    android.permission.SET_WALLPAPER权限  
    checkPermission(android.Manifest.permission.SET_WALLPAPER);  
    synchronized (mLock) { /* ① 与getWallpaper()方法一样，必须确定设置  
        壁纸的用户，并根据用户的ID获取对应的 WallpaperData */ int userId =  
        UserHandle.getCallingUserId(); WallpaperData wallpaper =  
        mWallpaperMap.get(userId); ..... try { // ②  
        updateWallpaperBitmapLocked()将创建一个文件描述符  
        ParcelFileDescriptor pfd = updateWallpaperBitmapLocked(name,  
        wallpaper); if (pfd != null) { // ③ 设置imageWallpaperPending为true  
            wallpaper.imageWallpaperPending = true; } return pfd; } finally { ... } } }
```

再看 updateWallpaperBitmapLocked () 的实现：

[WallpaperManagerService.java--

```
>WallpaperManagerService.setWallpaper() ParcelFileDescriptor
```

```
updateWallpaperBitmapLocked(String name, WallpaperData wallpaper) {  
    try { // ① 文件路径的计算方法与getWallpaper()方法一模一样  
        File dir =  
        getWallpaperDir(wallpaper.userId); .....  
        File file = new File(dir,  
        WALLPAPER); // ② 创建一个可供写入的文件描述符并返回  
        ParcelFileDescriptor fd = ParcelFileDescriptor.open(file,  
        MODE_CREATE|MODE_READ_WRITE); .....  
        return fd; } catch  
(FileNotFoundException e) {.....} }
```

8.3.3 连接静态壁纸的设置与获取——WallpaperObserver

从这两个方法的实现来看，只能说明WallpaperManagerService为壁纸设置者与Image-Wallpaper提供了同一个文件的路径，使得位图数据可以从壁纸设置者传递到ImageWallpaper中。但是上述设置静态壁纸的过程ImageWallpaper并没有参与，那么ImageWallpaper如何知道刚刚进行了静态壁纸的设置呢？原来在WallpaperManagerService中还存在着一个对静态壁纸来说很重要的组件：WallpaperObserver。

WallpaperObserver继承自FileObserver。这个基于inotify机制的FileObserver是Android框架所以提供的一个工具类，用于监听文件系统中所发生的文件创建、删除与修改等事件。FileObserver的使用者可以监听一个文件夹或者一个文件，并在上述增删修改动作发生时，抽象方法onEvent（）便会得到回调并且可以根据其参数得知发生这些动作的文件路径。对WallpaperManagerService来说，这个功能显得尤为有

用。WallpaperObserver监听文件夹/data/system/users/ /下的CLOSE_WRITE事件，于是，当静态壁纸设置者完成位图的写入并关闭文件之后，WallpaperObserver的onEvent () 便会得到通知。不难想到，在这个onEvent () 方法中，WallpaperManagerService会尝试通知ImageWallpaper进行壁纸的重新载入与绘制。



注意

WallpaperObserver除了CLOSE_WRITE之外还监听DELETE以及DELETE_SELF两个事件。其中DELETE事件是指被监听的文件夹下任何文件被删除，而DELETE_SELF则表示被监听的文件夹被删除。因为作为静态壁纸的位图数据被保存在文件系统之中，而在root权限下删除或修改这个文件易如反掌，因此它并没有受到足够有效保护。监听DELETE以及DELETE_SELF两个事件的目的就是应对位图被删除的情况。

参考onEvent () 方法的实现：

[WallpaperManagerService.java-->WallpaperObserver.onEvent()] public void onEvent(int event, String path) { synchronized (mLock) { // ① 获取FileObserver上报的发生CLOSE_WRITE事件的文件路径 File changedFile = new File(mWallpaperDir, path); /* mWallpaperFile是 WallpaperObserver的一个成员变量，在WallpaperObserver的构造函数中它被赋值指向/data/system/users/ /wallpaper。因为/data/system/users/ /文件夹下还存在其他与当前用户有关的数据，因此这个判断的目的是排除发生在无关文件上的事件 */ if (mWallpaperFile.equals(changedFile)) { /* ② 确定是否通知ImageWallpaper进行壁纸更新。其条件有三： 1. wallpaperCompoenent == null意味着目前系统没有任何壁纸 2. event != CLOSE_WRITE意味着位图文件被删除 3. imageWallpaperPending为 true。在WallpaperManagerService.setWallpaper() 会将这个值设置为 true。满足这个条件表示这个CLOSE_WRITE事件由设置静态壁纸所引发。所以，私自打开存储壁纸位图的文件并写入数据并不会引发静态壁纸的切换 */ if (mWallpaper.wallpaperComponent == null || event != CLOSE_WRITE || mWallpaper.imageWallpaperPending) { if (event == CLOSE_WRITE) { mWallpaper.imageWallpaperPending = false; } // ③ bindWallpaperComponentLocked()方法将会启动IMAGE_WALLPAPER bindWallpaperComponentLocked(IMAGE_WALLPAPER, true, false, mWallpaper, null); // 保存壁纸的状态。以便系统重启或发生用户切换时可以恢复到目前的壁纸设置 saveSettingsLocked(mWallpaper); } } } }

这个方法的核心工作是前面所介绍的bindWallpaperComponentLocked () 方法。注意bindWallpaperComponentLocked () 方法的工作是启动一个新的壁纸并终止旧有壁纸的运行，所以它会在ImageWallpaper中创建一个新的DrawableEngine用以渲染新的壁纸位图。换句话说，Android并没有以某种方式通知现有的DrawableEngine重新载入位图并重绘，而是重新启动一个新的DrawableEngine对象，并将旧有的DrawableEngine对象销毁。新的DrawableEngine对象通过调用它的updateWallpaperLocked () 从位图文件中加载新的位图，并通过drawFrameLocked () 方法将其绘制在壁纸窗口之上。

总而言之，设置静态壁纸的完整过程如下：

- 静态壁纸的设置者通过WallpaperManagerService.setWallpaper () 获取一个文件描述符。
- 静态壁纸的设置者将作为壁纸的位图写入这个描述符所指向的文件中。
- WallpaperManagerService中的WallpaperObserver得到文件写入完毕的通知，然后通过bindWallpaperComponentLocked () 方法绑定壁纸服务ImageWallpaper。在这个过程中，新的DrawableEngine在ImageWallpaper中被创建。

·新建的DrawableEngine通过WallpaperManager.getWallpaper () 获取存储位图的文件描述符，然后将其解码为一个Bitmap对象并保存在mBackground成员中。

·DrawableEngine通过drawFrameLocked () 方法将mBackground绘制在壁纸窗口的Surface上。

相对于动态壁纸，静态壁纸的特殊性在于WallpaperManager及WallpaperManager-Service为静态壁纸的设置提供了一套机制，使得静态壁纸设置者可以将位图传递给ImageWallpaper。

虽然静态壁纸与动态壁纸的设置方式与流程存在较大不同，但是它们也有着高度的统一性。首先，二者的本质都是在WallpaperService之中运行一个Engine对象。其次，虽然WallpaperManagerService为动态壁纸以及静态壁纸提供不同的接口进行设置，但是它们的设置流程都将会合到bindWallpaperComponentLocked () 方法。再者，无论是调用setWallpaperComponent () 还是setWallpaper () ，都会在对应的壁纸服务内创建新的Engine对象。

至此，对WallpaperManagerService以及壁纸实现相关内容的讨论告一段落。很显然，壁纸窗口的存在意义是为其他窗口提供背景，所以它注定会对其他窗口的Z序、可见性等状态有很强的依赖性。在处理这些依

赖关系上，WallpaperManagerService很显然是无能为力的。于是这就需要窗口的管理者WMS出场了。

8.4 WMS对壁纸窗口的特殊处理

壁纸存在的意义是为其他窗口提供背景。当一个窗口希望壁纸作为其背景时，可以将FLAG_SHOW_WALLPAPER标记加入其flags中。当WMS检测到处于显示状态的窗口声明这一标记时，会将壁纸窗口衬于此窗口之下，于是用户便可以透过此窗口的透明区域看到壁纸窗口的内容。所以，与其他窗口不同，壁纸窗口在窗口列表中的Z序与声明FLAG_SHOW_WALLPAPER标记的窗口紧密联系。另外，动态壁纸往往十分消耗计算资源，WMS必须确保当用户看不到壁纸的时候能够通过onVisibilityChanged () 回调将这个状态通知动态壁纸，使后者立即停止任何消耗资源的操作。再者，申请这个标记的窗口希望在进行窗口动画时，壁纸窗口能够同步地进行动画，就好像壁纸是此窗口的一部分一样。于是在WMS的窗口动画子系统中也存在着对壁纸窗口的特殊处理。

壁纸窗口在WMS中的特殊行为主要体现在三个方面：壁纸窗口的Z序、壁纸窗口的可见性以及壁纸窗口的动画。

8.4.1 壁纸窗口Z序的确定

既然壁纸窗口要衬托在声明了FLAG_SHOW_WALLPAPER标记的窗口之下，因此壁纸窗口的Z序取决于声明这个标记的窗口的Z序。在WMS

中，声明这个标记的窗口称为壁纸目标，并保存在WMS的一个名为mWallpaperTarget的WindowState类型的成员中。所以确定壁纸窗口Z序的核心工作就是寻找这个壁纸目标。

寻找壁纸目标的工作可能很简单，也可能很复杂。最简单的情况是系统中仅有一个窗口声明FLAG_SHOW_WALLPAPER标记，并且这个窗口目前是用户可见的，在这种情况下它就是壁纸目标。稍微复杂一些的情况是系统中有多个窗口声明FLAG_SHOW_WALLPAPER标记，此时拥有最高Z序并且用户可见的窗口会成为壁纸目标。而更加复杂的情况则是有多个窗口声明FLAG_SHOW_WALLPAPER标记，同时它们还在进行窗口动画，此时便需要在这些窗口之间进行相对复杂的选择策略。

选择壁纸目标并调整壁纸窗口Z序的策略位于WMS.adjustWallpaperWindowsLocked () 方法中。每当窗口列表发生变化，例如有窗口添加或移除都有可能使得壁纸窗口的Z序发生变化，此时WMS便会调用这一方法重新计算壁纸目标并调整壁纸窗口在窗口列表中的位置，并在此之后通过调用assignLayersLocked () 方法刷新窗口列表中每个窗口的layer值从而调整它们的Z序。

这个方法的内容较多，为了便于分析可以将它的工作内容分为两个方面。第一个是查找壁纸目标窗口，而第二个则是将壁纸窗口移动到窗口目标下。

第一个工作又可以分为两子阶段：

- 首先通过在窗口列表中查找FLAG_SHOW_WALLPAPER标记初步确定一个壁纸目标。这个初步确定的壁纸目标必须满足两个基本条件：声明上述标记，并且用户可见。
- 如果旧的壁纸目标与初步确定的新壁纸目标这两个窗口都处于动画过程中，WMS会比较这两个目标窗口的Z序，并用mLowerWallpaperTarget以及mUpperWallpaperTarget两个成员记录这两个窗口。在这种情况下，虽然真正的壁纸目标mWallpaperTarget可能是它们二者之一，但是壁纸窗口却被放置在mLowerWallpaperTarget之下。当二者之一停止动画之后，再次调用adjustWallpaperWindowsLocked()时，会跳过这个阶段，从而使得壁纸窗口被正确地放置在mWallpaperTarget下。

第二个工作同样也可以分为两个子阶段：

- 根据第一个阶段所确定的壁纸目标计算壁纸窗口在窗口列表中的位置。正常情况下，壁纸窗口的位置应该位于壁纸目标mWallpaperTarget所指定的窗口下。不过也有例外。一是当新旧两个壁纸目标都处于动画状态时，壁纸窗口的位置以mLowerWallpaperTarget所指定的窗口为准。二是当目标窗口拥有父窗口或STARTING窗口时，壁纸窗口会位于所有这些与目标窗口有关的窗口之下。

·将壁纸窗口移动到这一位置上。

本节将分别对这4个阶段进行介绍。

1.初步确定壁纸目标

初步确定壁纸目标的原则是在窗口列表中寻找第一个声明

FLAG_SHOW_WALLPAPER标记且用户可见的窗口。不过这个策略会根据候选窗口的动画情况进行细微调整。

[WindowManagerService.java--

```
>WindowManagerService.adjustWallpaperWindowsLocked() int  
adjustWallpaperWindowsLocked() { ..... // 局部变量的声明与初始化 // 从  
上到下依次遍历所有窗口 while (i > 0) { i--; w = windows.get(i); ..... if  
((w.mAttrs.flags&FLAG_SHOW_WALLPAPER) != 0 &&  
w.isReadyForDisplay() && (mWallpaperTarget == w || w.isDrawnLw())) { //  
① 如果声明FLAG_SHOW_WALLPAPER标记， 并且用户可见， 那么它  
作为壁纸目标的候选 foundW = w; foundI = i; if (w == mWallpaperTarget  
&& w.mWinAnimator.isAnimating()) { /* ② 如果候选窗口是旧的壁纸目  
标， 并且处在动画过程中， 那么将继续向下查找可能成为壁 纸目标的  
窗口 */ continue; } else { // ③ 否则终止查找， 确定此窗口为窗口目标的  
候选 break; } } ..... } ..... }
```

在这个阶段中，候选的壁纸目标会保存在foundW局部变量中，并且它在窗口列表中的索引会被保存在foundI中。这两个变量将会是后续阶段的工作依据。后面将会以跟踪foundW与foundI的变化作为主线讨论adjustWallpaperWindowsLocked () 的工作原理。

另外，`w==mWallpaperTarget&&w.mWinAnimator.isAnimating ()` 这个条件判断可能令人费解。adjustWallpaperWindowsLocked () 下一个阶段中会检查旧有的壁纸目标与新壁纸目标的动画情况，并在两个目标都进行动画时设置mLowerWallpaperTarget与mUpper-WallpaperTarget。于是，如果候选窗口并没有进行动画，则下一阶段的处理便不需要进行，那么候选目标就是最终的壁纸目标，所以adjustWallpaperWindowsLocked () 会终止对候选窗口的查找。否则，就必须找到mWallpaperTarget之外的一个候选窗口作为新的壁纸目标foundW，并在下一阶段在二者之间选择赋值给mLowerWallpaperTarget以及mUpperWallpaperTarget。

2. 设置mLowerWallpaperTarget以及mUpperWallpaperTarget

接下来看adjustWallpaperWindowsLocked () 如何设置mWallpaperTarget、mLowerWallpaperTarget以及mUpperWallpaperTarget。

[WindowManagerService.java--

```
>WindowManagerService.adjustWallpaperWindowsLocked() int  
adjustWallpaperWindowsLocked() { if (mWallpaperTarget != foundW &&  
(mLowerWallpaperTarget == null || mLowerWallpaperTarget != foundW)) {  
..... // 重置mLowerWallpaperTarget以及mUpperWallpaperTarget  
mLowerWallpaperTarget = null; mUpperWallpaperTarget = null; /* ① 设置  
mWallpaperTarget为上一阶段所找到的新的壁纸目标。旧的壁纸目标会  
被保存在 oldW中 */ WindowState oldW = mWallpaperTarget;  
mWallpaperTarget = foundW; if (foundW != null && oldW != null) { // 旧  
的壁纸目标是否在动画中？ boolean oldAnim =  
oldW.mWinAnimator.mAnimation != null || (oldW.mAppToken != null &&  
oldW.mAppToken.mAppAnimator.animation != null); // 新的壁纸目标是否  
在动画中？ boolean foundAnim = .....; ..... /* 倘若二者同时处在动画状  
态中，则设置mLowerWallpaperTarget为位置较低的窗口，并设置  
mUpperWallpaperTarget为位置较高的窗口 */ if (foundAnim &&  
oldAnim) { int oldI = windows.indexOf(oldW); ..... if (oldI >= 0) { ..... if  
(foundI > oldI) { /* ② 新的壁纸目标窗口位于旧的壁纸目标窗口之上。  
所以设置mUpperWallpaperTarget为新的壁纸目标，  
mLowerWallpaperTarget为旧的壁纸目标。另外，用于确定壁纸窗口插  
入位置的foundW与foundI也会被设置为 mLowerWallpaperTarget */  
mUpperWallpaperTarget = foundW; mLowerWallpaperTarget = oldW;
```

```
foundW = oldW; foundI = oldI; } else { /* ③ 新的壁纸目标窗口位于新的  
壁纸目标窗口下。所以设置mUpperWallpaperTarget为旧的壁纸目标，  
mLowerWallpaperTarget为新的壁纸目标 */ mUpperWallpaperTarget =  
oldW; mLowerWallpaperTarget = foundW; } } } // 其他情况下，保持  
foundW的状态不变 } else if (mLowerWallpaperTarget != null) { /* 如果  
mLowerWallpaperTarget或mUpperWallpaperTarget停止动画，则将这两个  
变量重置为null */ ..... } }
```

这个阶段主要是为了处理新旧两个壁纸目标窗口都处于动画状态这种特殊情況的。处于这种情况下，用于指示壁纸窗口添加位置的foundW以及foundI会被设置为mLowerWallpaperTarget所指定的窗口及其在窗口列表中的索引。于是真正的壁纸目标mWallpaperTarget有可能会与壁纸窗口的实际位置发生分离，即中间隔着mLowerWallpaperTarget。而这种情况是很少见的，绝大多数情况下，mLowerWallpaperTarget以及mUpperWallpaperTarget是null，而mWallpaperTarget会被设置为foundW。



说明

mLowerWallpaperTarget以及mUpperWallpaperTarget表示同时有两个用户可见的壁纸目标。而壁纸窗口只有一个。那么谁说了算呢？WMS的策略是，壁纸窗口的位置与可见性（onVisibilityChanged（））是mLowerWallpaperTarget说了算。而壁纸的Offset则是mWallpaperTarget说了算。至于壁纸窗口的动画，则是谁说了都不算——这种情况下壁纸窗口将会保持静止。

经过这一阶段，mWallpaperTarget被设置为foundW，而foundW与foundI又根据动画情况被设置为mWallpaperTarget或mLowerWallpaperTarget。

3. 确定壁纸窗口在窗口列表中的位置

接下来便根据foundW确定壁纸窗口在窗口列表中的最终位置。在这个过程中会同时计算壁纸窗口的可见性。关于可见性的详细讨论请参考8.4.2节。

```
[WindowManagerService.java--> WindowManagerService.adjustWallpaperWindowsLocked() int
adjustWallpaperWindowsLocked() { // 壁纸窗口可见的最基本要求是找到
一个壁纸目标foundW boolean visible = foundW != null; if (visible) { // 根
据foundW的状态最终确定壁纸是否可见 visible =
isWallpaperVisible(foundW); /* ① 计算壁纸在动画过程中的layer调整。
在第4章与窗口Layer计算有关的内容介绍过，Activity执行动画时会通
```

过Animation.setZAlphaAdjustment()方法临时调整隶属于Activity的窗口的layer。壁纸窗口作为其目标窗口的一个附属，当目标窗口存在layer调整时，自然需要跟着调整自己的layer。WMS将这一调整存储在mWallpaperAnimLayerAdjustment成员中注意，仅当mLowerWallpaperTarget为null时才会这样设置。因为此时有两个壁纸目标正在进行动画，所以壁纸窗口无法得知使用那个目标的layer调整。于是索性不再对壁纸的layer进行调整 */
mWallpaperAnimLayerAdjustment = (mLowerWallpaperTarget == null && foundW.mAppToken != null) ?
 foundW.mAppToken.mAppAnimator.animLayerAdjustment : 0; /* ②
maxLayer是PhoneWindowManager所允许壁纸拥有的layer上限（不含）。这一上限与状态栏的layer相同。即壁纸无论如何不能覆盖状态栏与导航栏 */ final int maxLayer = mPolicy.getMaxWallpaperLayer() *
 TYPE_LAYER_MULTIPLIER + TYPE_LAYER_OFFSET; /* ③ 确保壁纸位于layer上限、目标窗口、目标窗口的父窗口以及STARTING窗口之下。离开这个循环之后，foundW将指向这些窗口中位置最靠下的那个窗口，而foundI则指向此窗口在列表中的位置 */ while (foundI > 0) {
 WindowState wb = windows.get(foundI - 1); /* 下面这个判断的合理性建立在如下事实之上：窗口、窗口的父窗口以及其所属Activity的STARTING窗口紧邻在一起 */ if (wb.mBaseLayer < maxLayer &&
 wb.mAttachedWindow != foundW && (foundW.mAttachedWindow == null

```
|| wb.mAttachedWindow != foundW.mAttachedWindow) &&
(wb.mAttrs.type != TYPE_APPLICATION_STARTING || foundW.mToken
== null || wb.mToken != foundW.mToken) { break; } foundW = wb; foundI-
-; } } if (foundW == null && topCurW != null) { /* 倘若自始至终没能找
到foundW，即没有哪个窗口声明FLAG_SHOW_WALLPAPER标记则
foundW与foundI被设置为topCurW与topCurI+1。topCurW与topCurI在
第一阶段中被赋值，用于指示壁 纸窗口在
adjustWallpaperWindowsLocked()执行之前的位置。事实上仅当壁纸窗
口位于窗口 列表底部时，topCurW才不为null。因此，这里的赋值的意
思是保持壁纸窗口位于窗口列表的底 部。倘若foundW与topCurW都为
null，foundI则保持为初始的0，所以即便无法进入这个分支 分支进行
赋值，在foundW为null时也会因为foundI为0而使得壁纸窗口从当前位置
被移动到窗口 列表的底部 */ foundW = topCurW; foundI = topCurI+1;
} else { // ④ foundW此时被设置为壁纸窗口即将插入的位置上的那个窗
口 foundW = foundI > 0 ? windows.get(foundI-1) : null; } /* 从
mWallpaperTarget中获取壁纸的Offset并保存在WMS的成员变量之中。
随后这些Offset信息将会 被发送给壁纸窗口 */ if (visible) { if
(mWallpaperTarget.mWallpaperX >= 0) { mLastWallpaperX =
mWallpaperTarget.mWallpaperX; mLastWallpaperXStep =
mWallpaperTarget.mWallpaperXStep; } if (mWallpaperTarget.mWallpaperY
```

```
>= 0) { mLastWallpaperY = mWallpaperTarget.mWallpaperY;  
mLastWallpaperYStep = mWallpaperTarget.mWallpaperYStep; } } }
```

经过这一阶段的计算，foundI-1最终指示了壁纸窗口在窗口列表中应当处于的位置。

4. 移动壁纸窗口到指定的位置

接下来便是移动壁纸窗口到foundI-1的位置。虽说在正常情况下系统中只有一个壁纸窗口。但是也有例外。

WallpaperManagerService.bindWallpaperComponentLocked () 的实现是首先启动新壁纸，然后终止旧壁纸。而且壁纸的启动与终止分别在对应壁纸服务的进程中异步进行。于是在一个较短的时间里，系统中可能会存在两个壁纸窗口。所以adjustWallpaperWindowsLocked () 在进行壁纸窗口的移动时必须将这种情况考虑在内。

参考其实现：

```
[WindowManagerService.java--  
>WindowManagerService.adjustWallpaperWindowsLocked()] int  
adjustWallpaperWindowsLocked() { int curTokenIndex =  
mWallpaperTokens.size(); // 遍历所有的WallpaperToken while  
(curTokenIndex > 0) { curTokenIndex--; WindowToken token =  
mWallpaperTokens.get(curTokenIndex); ..... int curWallpaperIndex =
```

```
token.windows.size(); // 遍历WallpaperToken下所有的壁纸窗口 while  
(curWallpaperIndex > 0) { curWallpaperIndex--; WindowState wallpaper =  
token.windows.get(curWallpaperIndex); ..... // ① 首先将壁纸窗口从窗口  
列表中移除 int oldIndex = windows.indexOf(wallpaper); if (oldIndex >= 0)  
{ windows.remove(oldIndex); mWindowsChanged = true; if (oldIndex <  
foundI) { foundI--; } } // ② 再将壁纸插入foundI所指定的位置上  
windows.add(foundI, wallpaper); mWindowsChanged = true; changed |=  
ADJUST_WALLPAPER_LAYERS_CHANGED; } } }
```

至此，壁纸窗口的位置调整完毕。根据第4章的知识可知，由于窗口列表的窗口位置发生变化，于是在adjustWallpaperWindowsLocked () 方法结束之后，必须尽快调用WMS.assignLayersLocked () 为所有窗口重新计算layer值。

总体来说，在正常情况下，mWallpaperTarget是窗口列表中从上往下第一个用户可见并且声明FLAG_SHOW_WALLPAPER标记的窗口，而壁纸窗口则会被移动到mWallpaperTarget窗口或其父窗口、STARTING窗口之下。而在新旧两个壁纸目标都存在动画时，那么壁纸窗口则会被移动到mLowerWallpaperTarget窗口或其父窗口、STARTING窗口之下。

8.4.2 壁纸窗口的可见性

壁纸窗口的可见性的计算处于WMS.adjustWallpaperWindowsLocked() 的第三个阶段中。从这个阶段的代码可知，壁纸窗口可见的第一个条件就是在系统之中存在一个壁纸目标。然后 WMS.isWallpaperVisible () 方法会根据这个壁纸目标的状态做出壁纸是否可见的最终决定。参考这一方法的实现：

[WindowManagerService.java--

```
>WindowManagerService.adjustWallpaperWindowsLocked() final boolean  
isWallpaperVisible(WindowState wallpaperTarget) { return (wallpaperTarget  
!= null && (!wallpaperTarget.mObscured || (wallpaperTarget.mAppToken !=  
null && wallpaperTarget.mAppToken.mAppAnimator.animation != null))) ||  
mUpperWallpaperTarget != null || mLowerWallpaperTarget != null; }
```

显而易见，只要满足三个条件之一壁纸即为可见的：

- mUpperWallpaperTarget或mLowerWallpaperTarget不为null。这表示目前存在着新旧两个壁纸目标同时进行动画的情况。
- wallpaperTarget不被另一个全屏窗口遮挡。即这个窗口部分或者全部用户可见。此时自然需要显示壁纸。
- wallpaperTarget隶属于一个Activity而这个Activity正在执行动画。Activity执行动画时会附带layer的调整，于是即便布局结果显示wallpaperTarget被另一个全屏窗口所遮挡，但是在实际显示的过程中由

于layer调整的存在可能使得wallpaperTarget显示在其他窗口之上，所以此时仍然需要显示壁纸。



注意

isWallpaperVisible () 的参数wallpaperTarget并不是mWallpaperTarget，而是foundW。

isWallpaperVisible () 方法的返回值保存在
adjustWallpaperWindowsLocked () 方法中，并在最后一个阶段进行壁
纸窗口移动之前，通过调用WMS.dispatchWallpaperVisibility () 方法将
壁纸的可见性通知给WallpaperService中的Engine对象。

[WindowManagerService.java--

```
>WindowManagerService.dispatchWallpaperVisibility() void  
dispatchWallpaperVisibility(final WindowState wallpaper, final boolean  
visible) { if (wallpaper.mWallpaperVisible != visible) { /* ① 将wallpaper的  
可见性保存在壁纸窗口的WindowState.mWallpaperVisible中。这个看似  
不起眼的设置其实非常重要。在动画系统中，WindowStateAnimator的
```

prepareSurfaceLocked()方法中会检查这个成员的取值。倘若为false，则会调用WindowStateAnimator.hide()方法，进而调用Surface.hide()将壁纸窗口隐藏。相反，会调用WindowStateAnimator.showSurfaceRobustlyLocked()进而调用Surface.show()使壁纸窗口重新可见 */ wallpaper.mWallpaperVisible = visible; try { /* ② 通过IWindow.dispatchAppVisibility()方法将可见性传递给WallpaperService下的Engine对象 */ wallpaper.mClient.dispatchAppVisibility(visible); } catch (RemoteException e) {} } }

dispatchWallpaperVisibility () 合理地使用IWindow.dispatchAppVisibility () 回调将壁纸的可见性通知给WallpaperService里的Engine对象。Engine对象的mWindow成员则在接收到这个回调后将其解释为onVisibilityChanged () 。Engine对象可以根据这一回调重启或终止画面的绘制工作。

由第4章的知识可知，WMS中布局子系统的任何变化都会引发动画系统进行一帧的处理。而在这一帧的处理之中，每个窗口的WindowStateAnimator.prepareSurfaceLocked () 方法都会被调用以更新Surface的状态。壁纸窗口也不例外。在prepareSurfaceLocked () 方法中可以看到如下处理：

```
[WindowStateAnimator.java-->WindowStateAnimator.prepareSurfaceLocked()] public void
```

```
prepareSurfaceLocked(final boolean recoveringMemory) { if (mIsWallpaper  
&& !mWin.mWallpaperVisible) { /* ① 本窗口是壁纸窗口， 并且  
WindowState.mWallpaperVisible为false， 于是通过Hide() 方法将壁纸窗  
口的Surface隐藏 */ hide(); } else if (w.mAttachedHidden ||  
!w.isReadyForDisplay()) { ..... /* ② 本窗口不是壁纸窗口， 但是本窗口有  
可能是壁纸目标， 因此调用WindowAnimator的hide- WallpapersLocked()  
尝试隐藏壁纸 */ mAnimator.hideWallpapersLocked(w, true); ..... } else if  
(.....) { // ③ 否则设置Surface为可见 } }
```

倘若经过WMS.dispatchWallpaperVisibility () 的设置， 壁纸窗口的
WindowStateAnimator.prepareSurfaceLocked () 就会对其所设置的
mWallpaperVisible做出反应， 隐藏或者显示壁纸的Surface。

倘若一个非壁纸窗口被隐藏时， 则会通过
WindowAnimator.hideWalpapersLocked () 尝试隐藏壁纸。
hideWalpapersLocked () 会检查此窗口是否是mWallpaperTarget， 如果
是， 并且mLowerWallpaperTarget为null即没有进行动画， 则会遍历每
一个壁纸窗口并为其调用WindowStateAnimator.hide () 将壁纸隐藏。

所以说， 壁纸的可见性有两层含义， 第一层是相对WallpaperService中
的Engine对象而言， 壁纸的可见性决定了Engine对象是否进行壁纸内容
的绘制工作。第二层则是对WMS中的窗口管理而言， 壁纸的可见性会
使得壁纸窗口的Surface被隐藏或显示。

壁纸的可见性不同于常规意义上的窗口可见性。常规意义上的窗口可见性的变化伴随着Surface的创建与销毁，而壁纸的可见性只会使得Surface显示或隐藏。这一区别使得显示或隐藏壁纸的效率变得非常高，但是带来的负面影响则是Surface的内存永远处于占用状态。

8.4.3 壁纸窗口的动画

在默认情况下，当壁纸目标窗口发生动画时，壁纸窗口也会随着目标窗口产生同步的动画。这一效果的实现位于
WindowStateAnimator.computeShownFrameLocked () 方法中。参考相关代码：

```
void computeShownFrameLocked() { ..... /* ① 壁纸窗口会额外地应用来自壁纸目标窗口的变换，前提是mLowerWallpaperTarget为null。这个前提其实很简单，因为mLowerWallpaperTarget不为null时表示有两个壁纸目标在进行动画。这种情况下壁纸窗口无法决定使用哪一个目标的动画变换，于是壁纸窗口便保持了静止不动 */ if (mIsWallpaper && mAnimator.mLowerWallpaperTarget == null && mAnimator.mWallpaperTarget != null) { /* 不要为wallpaperAnimator这个名字所迷惑。这个WindowStateAnimator其实是壁纸 目标窗口的 Animator */ final WindowStateAnimator wallpaperAnimator = mAnimator.mWallpaperTarget.mWinAnimator; /* ② 设置 attachedTransformation为壁纸目标窗口的动画变换，前提是壁纸目标窗
```

窗口动画的getDetachedWallpaper()返回为false。从这个行为来看，壁纸窗口在动画过程中扮演了壁纸目标窗口子窗口的角色 */ if
(wallpaperAnimator.mHasLocalTransformation &&
wallpaperAnimator.mAnimation != null &&
!wallpaperAnimator.mAnimation.getDetachWallpaper()) {
attachedTransformation = wallpaperAnimator.mTransformation; } /* ③ 设
置appTransformation为壁纸目标窗口所属Activity的动画变换，前提是
Activity的动画的getDetachedWallpaper()返回为false */ final
AppWindowAnimator wpAppAnimator = mAnimator.mWpAppAnimator; if
(wpAppAnimator != null && wpAppAnimator.hasTransformation &&
wpAppAnimator.animation != null &&
!wpAppAnimator.animation.getDetachWallpaper()) { appTransformation =
wpAppAnimator.transformation; } } /* 剩下的代码会按照常规窗口的方式
将attachedTransformation与appTransformation集成到tmpMatrix中，并由
此计算出Surface最终的透明度、位置以及DsDx、DtDx、DsDy以及
DtDy */ }

这段代码体现了如下几个关键点：

- 一般情况下，壁纸窗口会随着其目标窗口进行动画变换。
- 如果不希望壁纸窗口随着目标窗口进行动画变换，可以通过调用窗口动画的Animation.setDetachWallpaper()方法并传递参数false实现。

·当mLowerWallpaperTarget存在时，壁纸窗口会保持静止。

其实能够作为其他窗口背景的除了本章所介绍的壁纸之外，还有一种特殊的背景。窗口可以通过Animation.setBackgroundColor () 方法设置一个动画背景色。WindowAnimator在进行动画的过程中倘若发现存在一个窗口指定了动画背景色，就会在此窗口之下放置一个Surface充当此窗口在动画过程中的背景。目前WindowAnimator只会使用背景色的透明度分量，其他的RGB分量会被忽略。

这个动画背景和壁纸之间存在着一个小小的冲突，倘若指定了动画背景色的窗口是mWall-paperTarget、 mLowerWallpaperTarget以及mUpperWallpaperTarget三者之一，这块黑色的Surface就会将壁纸窗口遮挡，在用户看来就仿佛壁纸消失了一样。于是会在添加这块Surface之前检查指定动画背景色的窗口是否是上述壁纸目标窗口。如果是，则会将Surface放置在壁纸窗口之下。参考代码如下：

```
[WindowAnimator.java-->WindowAnimator.updateWallpaperLocked()]
private void updateWallpaperLocked(int displayId) { ..... // 遍历所有的
WindowStateAnimator，从中查找指定动画背景的窗口 for (int i =
winAnimatorList.size() - 1; i >= 0; i--) { WindowStateAnimator
winAnimator = winAnimatorList.get(i); // 首先检查窗口动画是否设置了
动画背景色 if (winAnimator.mAnimating) { if (winAnimator.mAnimation
!= null) { final int backgroundColor =
```

```
winAnimator.mAnimation.getBackgroundColor(); if (backgroundColor != 0)
{ /* ① 将指定动画背景的窗口的WindowStateAnimator保存在
windowAnimation- Background中。稍后将会把作为动画背景的Surface
放置在这个WindowState- Animator所属的窗口下。另外，从这个判断
可以看出，倘若有两个窗口指定动画背景色，那么将以layer值最小
的，也就是位置最靠下的那个窗口为准 */ if
(windowAnimationBackground == null || (winAnimator.mAnimLayer <
windowAnimationBackground.mAnimLayer)) {
windowAnimationBackground = winAnimator;
windowAnimationBackgroundColor = backgroundColor; } } } } // 然后检
查窗口所属的Activity的动画是否指定背景色。其判断方法与上述代码
完全一致 ..... } /* 完成上述循环之后，
windowAnimationBackgroudnColor指定动画背景色。而
windowAnimation- Background指定需要添加背景Surface的窗口 */ if
(windowAnimationBackgroundColor != 0) { // ② animLayer正常情况下的
取值是windowAnimationBackground窗口的layer int animLayer =
windowAnimationBackground.mAnimLayer; WindowState win =
windowAnimationBackground.mWin; if (mWallpaperTarget == win ||
mLowerWallpaperTarget == win || mUpperWallpaperTarget == win) { /* ③
倘若windowAnimationBackground的窗口是一个壁纸目标，于是就需要
将animLayer修正 为壁纸窗口的显示layer。于是遍历所有的窗口列表并
```

在其中寻找壁纸窗口，然后将anim- Layer修正为壁纸窗口的layer */

```
final int N = winAnimatorList.size(); for (int i = 0; i < N; i++) {  
    WindowStateAnimator winAnimator = winAnimatorList.get(i); if  
    (winAnimator.mIsWallpaper) { animLayer = winAnimator.mAnimLayer;  
    break; } } } // ④ 最后将作为动画背景的Surface显示在animLayer-  
    LAYER_OFFSET_DIM的位置上 if (windowAnimationBackgroundSurface  
!= null) { windowAnimationBackgroundSurface.show(mDw, mDh,  
    animLayer - WindowManagerService.LAYER_OFFSET_DIM,  
    windowAnimationBackgroundColor); } } else { // 倘若没有窗口指定动画  
    背景色，则隐藏作为动画背景的Surface if  
    (windowAnimationBackgroundSurface != null) {  
        windowAnimationBackgroundSurface.hide(); } } }
```

updateWallpaperLocked () 方法在WindowAnimator每一帧的处理中都会调用。它的主要工作是遍历所有动画中的窗口，并为其显示或隐藏一块作为动画背景的Surface。倘若要求显示动画背景的窗口是一个壁纸的目标，那么这块Surface会被放置在壁纸窗口之下。

8.4.4 壁纸窗口总结

总的来说，WMS对壁纸窗口的特殊处理主要分为Z序的调整、可见性的计算与处理以及壁纸动画三个方面。

壁纸窗口Z序的调整是其中最复杂且最重要的处理，其目的是让壁纸窗口能够正确地位于希望壁纸作为背景的窗口的下方。其调整方法简单来说就是倘若有窗口声明FLAG_SHOW_WALLPAPER，则壁纸会位于这一窗口之下，并称为壁纸目标，否则壁纸会被放置在窗口列表的底部。一个特殊情况是在两个壁纸目标窗口之间切换时，倘若它们都在执行动画，则Z序较低的那个目标窗口将是真正的壁纸目标。

壁纸可见性的处理目的是能够提高壁纸显示与隐藏的效率。当WMS认为壁纸不可见时，会通过Surface.hide () 将壁纸隐藏，否则通过Surface.show () 将其显示。在隐藏与显示切换的过程中不会发生Surface的创建与销毁工作，因而免去了壁纸服务中Engine的重绘过程，提高了效率。WMS认为壁纸可见的基本条件是有窗口可以作为壁纸目标，在满足这个基本条件之后，还需要能够满足三个条件之一：壁纸目标窗口未被遮挡，壁纸目标窗口所属的Activity正在进行动画，或者新旧两个壁纸目标都在进行动画。

至于壁纸动画，默认情况下壁纸窗口会随着目标窗口同步进行动画。除非目标窗口的Animation中设置了DetachWallpaper，或者新旧两个壁纸目标都在进行动画。

8.5 本章小结

关于壁纸的内容就介绍到这里。希望读者经过本章的学习之后能够深入了解Android壁纸的运行机制，体会WallpaperService中Engine对象如何通过WMS的接口直接创建与操作窗口的方法，以及WallpaperManagerService如何通过窗口令牌对Engine对象创建TYPE_WALLPAPER类型的窗口进行授权。理解WMS对壁纸窗口的特殊处理也很重要，因为这些特殊处理说明除了使用BaseLayer以及SubLayer以外精细地调整窗口的Z序的方法，以及对于那些需要频繁显示与隐藏的窗口所进行的优化。

Android壁纸使用了系统服务（管理者）加标准Android服务（实现者）的两层架构。当系统希望某些系统级UI由第三方进行实现与扩展，同时又不希望给予第三方过多的操作窗口的权限时，这种两层架构无疑是最佳选择。系统服务负责提供必要的系统级操作，而标准的Android服务可以在系统服务所规范的框架范围实现自由定制。第7章所介绍的SystemUI以及输入法都采用了这种架构。在深入理解壁纸的运行机制之后，感兴趣的读者可以类比地研究Android输入法的实现。

Table of Contents

推荐序

前言

第1章 开发环境部署

 1.1 获取Android源代码

 1.2 Android的编译

 1.3 在IDE中导入Android源代码

 1.3.1 将Android源代码导入Eclipse

 1.3.2 将Android源代码导入SourceInsight

 1.4 调试Android源代码

 1.4.1 使用Eclipse调试Android Java源代码

 1.4.2 使用gdb调试Android C/C++源代码

 1.5 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分析

 2.2.1 Binder架构总览

 2.2.2 初始化Java层Binder框架

 2.2.3 窥一斑，可见全豹乎

 2.2.4 理解AIDL

 2.2.5 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AudioService

3.1 概述

3.2 音量管理

3.2.1 音量键的处理流程

3.2.2 通用的音量设置函数setStreamVolume ()

3.2.3 静音控制

3.2.4 音量控制小结

3.3 音频外设的管理

3.3.1 WiredAccessoryObserver设备状态的监控

3.3.2 AudioService的外设状态管理

3.3.3 音频外设管理小结

3.4 AudioFocus机制的实现

3.4.1 AudioFocus最简单的例子

3.4.2 AudioFocus实现原理简介

3.4.3 申请AudioFocus

3.4.4 释放AudioFocus

3.4.5 AudioFocus小结

3.5 AudioService的其他功能

3.6 本章小结

第4章 深入理解WindowManagerService

4.1 初识WindowManagerService

4.1.1 一个从命令行启动的动画窗口

4.1.2 WMS的构成

4.1.3 初识WMS的小结

4.2 WMS的窗口管理结构

4.2.1 理解WindowToken

4.2.2 理解WindowState

4.2.3 理解DisplayContent

4.3 理解窗口的显示次序

4.3.1 主序、子序和窗口类型

4.3.2 通过主序与子序确定窗口的次序

4.3.3 更新显示次序到Surface

4.3.4 关于显示次序的小结

4.4 窗口的布局

4.4.1 从`relayoutWindow ()` 开始

4.4.2 布局操作的外围代码分析

4.4.3 初探`performLayoutAndPlaceSurfacesLockedInner ()`

4.4.4 布局的前期处理

4.4.5 布局`DisplayContent`

4.4.6 布局的最终阶段

4.5 WMS的动画系统

4.5.1 Android动画原理简介

4.5.2 WMS的动画系统框架

4.5.3 WindowAnimator分析

4.5.4 深入理解窗口动画

4.5.5 交替运行的布局系统与动画系统

4.5.6 动画系统总结

4.6 本章小结

第 5 章 深入理解Android输入系统

5.1 初识Android输入系统

5.1.1 `getevent`与`sendevent`工具

5.1.2 Android输入系统简介

5.1.3 IMS的构成

5.2 原始事件的读取与加工

5.2.1 基础知识：`INotify`与`Epoll`

5.2.2 `InputReader`的总体流程

5.2.3 深入理解`EventHub`

5.2.4 深入理解InputReader

5.2.5 原始事件的读取与加工总结

5.3 输入事件的派发

5.3.1 通用事件派发流程

5.3.2 按键事件的派发

5.3.3 DispatcherPolicy与InputFilter

5.3.4 输入事件的派发总结

5.4 输入事件的发送、接收与反馈

5.4.1 深入理解InputChannel

5.4.2 连接InputDispatcher和窗口

5.4.3 事件的发送

5.4.4 事件的接收

5.4.5 事件的反馈与发送循环

5.4.6 输入事件的发送、接收与反馈总结

5.5 关于输入系统的其他重要话题

5.5.1 输入事件ANR的产生

5.5.2 焦点窗口的确定

5.5.3 以软件方式模拟用户操作

5.6 本章小结

第 6 章 深入理解控件系统

6.1 初识Android的控件系统

6.1.1 另一种创建窗口的方法

6.1.2 控件系统的组成

6.2 深入理解WindowManager

6.2.1 WindowManager的创建与体系结构

6.2.2 通过WindowManagerGlobal添加窗口

6.2.3 更新窗口的布局

6.2.4 删除窗口

6.2.5 WindowManager的总结

6.3 深入理解ViewRootImpl

- 6.3.1 ViewRootImpl的创建及其重要的成员
- 6.3.2 控件系统的心跳：performTraversals ()
- 6.3.3 ViewRootImpl总结

6.4 深入理解控件树的绘制

- 6.4.1 理解Canvas
- 6.4.2 View.invalidate () 与脏区域
- 6.4.3 开始绘制
- 6.4.4 软件绘制的原理
- 6.4.5 硬件加速绘制的原理
- 6.4.6 使用绘图缓存
- 6.4.7 控件动画
- 6.4.8 绘制控件树的总结

6.5 深入理解输入事件的派发

- 6.5.1 触摸模式
- 6.5.2 控件焦点
- 6.5.3 输入事件派发的综述
- 6.5.4 按键事件的派发
- 6.5.5 触摸事件的派发
- 6.5.6 输入事件派发的总结

6.6 Activity与控件系统

- 6.6.1 理解PhoneWindow
- 6.6.2 Activity窗口的创建与显示

6.7 本章小结

第 7 章 深入理解SystemUI

7.1 初识SystemUI

- 7.1.1 SystemUIService的启动
- 7.1.2 状态栏与导航栏的创建
- 7.1.3 理解IStatusBarService

7.1.4 SystemUI的体系结构

7.2 深入理解状态栏

7.2.1 状态栏窗口的创建与控件树结构

7.2.2 通知信息的管理与显示

7.2.3 系统状态图标区的管理与显示

7.2.4 状态栏总结

7.3 深入理解导航栏

7.3.1 导航栏的创建

7.3.2 虚拟按键的工作原理

7.3.3 SearchPanel

7.3.4 关于导航栏的其他话题

7.3.5 导航栏总结

7.4 禁用状态栏与导航栏的功能

7.4.1 如何禁用状态栏与导航栏的功能

7.4.2 StatusBarManagerService对禁用标记的维护

7.4.3 状态栏与导航栏对禁用标记的响应

7.5 理解SystemUIVisibility

7.5.1 SystemUIVisibility在系统中的漫游过程

7.5.2 SystemUIVisibility发挥作用

7.5.3 SystemUIVisibility总结

7.6 本章小结

第 8 章 深入理解Android壁纸

8.1 初识Android壁纸

8.2 深入理解动态壁纸

8.2.1 启动动态壁纸的方法

8.2.2 壁纸服务的启动原理

8.2.3 理解UpdateSurface () 方法

8.2.4 壁纸的销毁

8.2.5 理解Engine的回调

8.3 深入理解静态壁纸——ImageWallpaper

- 8.3.1 获取用作静态壁纸的位图
- 8.3.2 静态壁纸位图的设置
- 8.3.3 连接静态壁纸的设置与获取——WallpaperObserver

8.4 WMS对壁纸窗口的特殊处理

- 8.4.1 壁纸窗口Z序的确定
- 8.4.2 壁纸窗口的可见性
- 8.4.3 壁纸窗口的动画
- 8.4.4 壁纸窗口总结

8.5 本章小结



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>