



“内核剖析”乍一听起来挺吓唬人的，但这个词语存在两个问题，第一个是什么才能称为内核？另一个是“谁”才有能力或者有机会写一本“内核剖析”的书？

本书之所以在前言中提出这个问题，就是为了不吓唬大家，并给大家一种信心，相信自己有能力理解本书的内容。

首先来回答第一个问题，什么才能称为内核？大家都知道，Linux 内核的本质包含了线程调度、内存管理及输入/输出管理，那么请问 Windows 操作系统的内核是什么呢？我们常说，苹果的操作系统 Mac OS X 的内核是基于 UNIX 的，那么可以说 Mac OS 的内核是 UNIX 吗？

如果仅从线程调度、内存管理，以及输入/输出的角度来区分 Windows 和 Mac OS 系统的话，能很明显地感觉到缺少点什么，那就是图形用户接口（GUI），Android、Windows、Mac OS 三者的操作方式完全不同，因此，对于图形操作系统而言，本人倾向于将 GUI 也划归到内核的范畴，这也就是为什么本书使用“内核”作为标题的原因。本书所谓的“内核剖析”的核心也正在于 Android 所设计的 GUI 框架的内部原理。Android 操作系统是基于 Linux 实现的，本书并不是去剖析 Linux。

下面再来回答第二个问题，即“谁”才有能力或者有机会写一本“内核剖析”的书？如果有人告诉你，一个非微软公司的技术人员写了一本 Windows 操作系统内核剖析的书，你信吗？反正我不信，原因是，没有阅读过 Windows 内核源码的人是不可能写出这样的书的，幸运的是 Android 的源码是开放的。可是源码开放就一定能写这样一本书吗？

在本书截稿时，我未曾见过一本真正分析 Android 内核的书，大多数书籍都是关于 Android SDK 应用开发的。在过去的工作经历中，常常遇到一些同事，由于对 Android 内核不了解，导致在应用程序开发时遇到一些无法解决的问题。遗憾的是，IT 类优秀书籍本来就很少，中文原创的更少，Android 领域的几乎没有，本人之前也写过一本《Android 程序设计》，坦白地讲，当我对 Android 内核彻底剖析后，觉得那本书有“误人子弟”的成分。因此，我才决定要将自己对 Android 的理解分享给更多的读者。

那么，我有可能写出一本真正的“内核剖析”的书吗？

我 2003 年毕业于西安电子科技大学通信工程学院，毕业后与两名同学一起创业，当时我们的目标是做一个“心情播放器”，其本质是一个彩屏多媒体掌上设备，当初想把它做成一个能够根据人的心情自动播放音乐的设备，不谈产品，仅从技术的角度来讲，我们基于美国德州仪器（TI）公司的一款 DSP 处理器完成了“心情播放器”的设计，包括软件和硬件，该软件系统包括支持最多 16 个线程的多线程管理、内存管理、FAT16 文件系统、GUI 子系统，以及一套标准应用程序开发框

架、桌面程序等，在这里要再次感谢同宿舍的陈静军同学，他是我到目前为止见过的写代码最优秀的人，在这个项目中，静军设计了这个操作系统的内核及 GUI 子系统，而我设计了硬件主板、驱动、系统开发框架等，说到这里，如果静军来写一本内核剖析的书，肯定会比我写得更好，在当初设计操作系统前，由于静军还没有加入到我们团队，我才花时间研究了嵌入式操作系统，并设计了一些简单的接口，而当静军加入后，这些工作就由他完成了，因此，从严格意义上讲，我并没有实际编写过操作系统内核代码，只不过从硬件、驱动、系统等不同层面设计了一个系统框架而已。

在这个项目中，一切只是从一颗处理器入手，没有基于任何代码，所有底层代码都是我们编写的，包括汇编和 C 语言程序设计，因此，在这个过程中，我彻底了解了 C 语言如何被编译成汇编代码，以及特定处理器如何影响上层的 C 程序。

当然，这个故事是以失败而告终的，后来我继续从事嵌入式产品设计，包括使用 TI 高性能 DSP 处理器、x86 处理器，ARM 处理器等，不过，仅过了两年时间，又去从事互联网产品的设计，并开始使用 Java、C++、PHP、JavaScript、Erlang 等不同语言进行软件开发，在使用各种语言时，我常常思考这些语言与底层系统的关系，并从编译原理的角度来理解每一种语言，从而能够理解不同语言的运行环境和操作系统的关系。

直到 Android 的诞生，我当时对 Android 的描述是，这是一个把嵌入式系统和互联网应用集合在一起的一个技术。幸运的是这些我都还算熟悉，因此就开始了 Android 的开发，最开始的时候仅仅是应用程序的开发，虽然也常常考虑 Android 底层的问题，但由于没有源码，所以也就没有仔细研究，后来发现，这也是一件好事，因为如果不熟悉上层的开发接口，则很难理解内核的一些概念。

后来，应用层积累得差不多了，源码也开放了，于是我就迫不及待地开始了内核之旅，所有的分析都是基于源码的阅读和测试，中间的过程的确是辛苦的，包括在 Ubuntu 及 Mac OS 上建立编译环境、思考 Android 中的异步调度架构、平衡工作和学习的时间等，早上坐地铁也常常看 Google groups 中关于 Android 的各种问答。不过，每当你明白一个大的架构的关键之处时，也是一件很开心的事情。

谈及以上履历的目的在于启发正在读大学的朋友，一名电子工程师一定要理论、硬件、软件及梦想同时具备，不要把自己区分为“硬件工程师”、“软件工程师”，我们可以称自己为电子工程师或者“梦想家”。另外，学习一定要循序渐进，如果你还不了解微机原理，那么就不要再学习 C 语言，如果你还不了解数字电路，那么就不要再学习微机原理，上层的软件开发需要对底层基础知识的理解，只有这样才能成为一名创造者，并设计出卓越的产品。

多么希望我们中国的大学生在今后的将来也能创造出像 Google、Microsoft、Facebook 这样著名的企业。

## 内容介绍

本书内容分为五大部分，分别如下：

第 1 部分，基础篇。因为 Android 内核研究必须基于 Unix-Like 的主机系统上，常见的有 Ubuntu

和 Mac OS X, 因此, 该部分介绍 Linux 的一些基础知识, 以及在 Linux 上管理源码的工具 git。

第 2 部分, 内核篇。Android 内核的核心就是一套 GUI 系统。该部分主要包含视图的内部工作机制及视图管理器 (Window Manager Service) 和 Activity 管理器 (Activity Manager Service) 的内部工作机制。

第 3 部分, 系统篇。内核不等于操作系统, Android 是一个操作系统, 因此, 除了内核之外, 还必须定义一套系统架构, 比如应用程序的格式定义, 以及应用程序如何被安装和卸载、输入法框架等, 有时候这部分内容也叫做外壳 (Shell)。

第 4 部分, 编译篇。Android 相关的源码据说超过 1000 万行, 这套源码由众多的子项目组成, 因此, 联合编译这些子项目就是一个复杂的问题。Android 源码中定义了一套编译框架, 该框架可以方便地编译不同类型的子项目, 比如一个动态链接库项目、Jar 包项目等。了解该套编译架构后, 就可以自由地在源码中添加需要的子项目, 并控制系统中已有子项目的编译过程。

第 5 部分, 硬件驱动篇。Android 目前最成功的产品当然就是智能手机, 但同时由于 Android 开源的特点, 也就可以应用于其他一些特定的产品, 比如玩具、学习机、税控机、门禁系统等, 因此, 该部分介绍了一款硬件开源的 Android 开发板卡。本来, 该部分内容还包括 OpenGL 框架、多媒体框架及 Android 硬件抽象层 (HAL) 三方面内容, 但由于出版时间原因, 暂未包含, 本书下一版将包含这些内容。

## 读者对象

本书适合于五类读者。

第一类, 开发过 Android 应用程序的工程师。如果你刚开始接触 Android, 那么这本书可能会很难理解, 建议去 Android 官方网站用两周的时间学习基本的 Android 应用程序开发, 或者去看本人早期创作的《Android 程序设计》一书, 但要带着怀疑的态度去读。

第二类, Android 技术相关的产品经理。对于产品经理而言, 了解项目的技术难度及技术可行性, 将有助于制定产品开发时间表。虽然产品经理不需要详细了解技术如何实现, 但起码应该知道产品技术的复杂度。

第三类, 有扎实的开发经验, 却未曾接触过 Android 的开发人员。系统框架的表面尽管各有千秋, 但其内涵却不会差别太大, 对于有扎实开发经验的朋友而言, 只需要重新了解一下 Android 中的新概念, 就能快速地将这些新概念与已有的知识融合起来, 这样, 便可以节省大量的时间。

第四类, 正在基于 iOS 开发的工程师。本人最近正在研究 iOS 的开发, 令人惊讶的是 iOS 和 Android 开发框架是如此相似。Object-C 语言和 Java 语言的语法虽然差别较大, 但其思想却很相似, 包括单继承、动态性、内存回收机制等。iOS 和 Android 的 Framework 也惊人相似, 比如都使用 sqlite 进行数据存储, 也使用 Preference 进行参数存储; 视图系统的 API 接口也类似的地方, 都可以使用 OpenGL 进行界面绘制。当然, iOS 和 Android 视图系统还是有一定的差别, 比如 iOS 中每一个 View 对象都有两个 Layer, 从而可以方便地使用 OpenGL 绘制任何一个 View 对象, 而 Android 却只有一

个，所以 Android 的动画效果没有 iOS 那样灵活。遗憾的是 iOS 不是开源的，因此，我们没有机会去了解 iOS 内部的详细机制，不过既然 iOS 和 Android 有这么多相似的地方，那么就可以通过了解 Android 的内核机制去思考 iOS 的一些特性。

第五类，想要编写一个 GUI 子系统的学生。Android 虽然更多地用于手机产品，但其内部的 GUI 子系统的实现却是一种通用的思路，因此，可以完全抛开 Android 的系统特性，而仅仅去研究其 GUI 子系统的实现思路，有了这种思路就可以使用各种语言设计自己想要的 GUI 子系统。

欢迎朋友们与我进行进一步的交流，我的 E-mail 是 [yuandanke@gmail.com](mailto:yuandanke@gmail.com)。

作者





## 第 1 部分 基础篇

第 1 章 Linux 基础 .....	2
1.1 Linux 文件系统概述 .....	2
1.2 Linux 启动过程 .....	4
1.3 常用 Linux 命令 .....	6
1.4 Shell 脚本备忘 .....	9
1.4.1 获取输入 .....	10
1.4.2 变量定义 .....	10
1.4.3 条件判断 .....	11
1.4.4 while []...do... done 语句 .....	12
1.4.5 for 循环 .....	13
1.4.6 函数 .....	14
1.4.7 常用内置符号常量 .....	15
1.5 Make 脚本备忘 .....	15
1.5.1 一个简单的 Makefile 文件 .....	16
1.5.2 变量的定义与赋值 .....	18
1.5.3 条件控制语句 .....	18
1.5.4 宏（函数）定义 .....	19
1.5.5 内置符号和变量 .....	22
1.5.6 模板目标（Pattern target） .....	23
1.5.7 目标特定的变量赋值（Target-specific variable） .....	24
1.5.8 常用选项 .....	25
第 2 章 Java 基础 .....	26
2.1 类装载器 DexClassLoader .....	26
2.1.1 DexClassLoader 的调用方法 .....	27

2.1.2	基于类装载器设计一种“插件”架构	29
2.2	JNI 调用机制	32
2.2.1	Java 访问 C	33
2.2.2	C 访问 Java	35
2.2.3	在 C 中使用持久对象	37
2.3	异步消息处理线程	37
2.3.1	实现异步线程的一般思路	38
2.3.2	Android 中异步线程的实现方法	38
第 3 章	Android 源码下载及开发环境配置	44
3.1	Mac 系统的配置	44
3.1.1	硬盘格式的配置	44
3.1.2	port 的用法	46
3.2	在 Linux 中配置 USB 连接	46
3.3	在 Eclipse 中调试 Framework	46
3.3.1	一段防止下载异常的脚本	47
3.3.2	调试 Framework 中的代码	47
第 4 章	使用 git	51
4.1	安装 git	52
4.2	git 仓库管理	52
4.2.1	仓库的组成	52
4.2.2	创建仓库	54
4.2.3	分支管理	55
4.3	git merge 用法	57
4.4	git rebase 用法	58
4.5	git cherry-pick 用法	61
4.6	git reset 用法	62
4.7	恢复到无引用提交	63
4.8	git remote 用法	65
4.9	git 配置	67
4.9.1	基本信息配置	68
4.9.2	merge、diff 工具配置	68
4.9.3	.gitignore 配置	70
4.10	同时使用 git 和 svn	71
4.11	其他 git 常用命令示例	72

第 7 章 理解 Context	98
7.1 Context 是什么	98
7.2 一个应用程序中包含多少个 Context 对象	99
7.3 Context 相关类的继承关系	99
7.4 创建 Context	100
7.4.1 Application 对应的 Context	101
7.4.2 Activity 对应的 Context	102
7.4.3 Service 对应的 Context	103
7.4.4 Context 之间的关系	104
第 8 章 创建窗口的过程	106
8.1 窗口的类型	106
8.2 token 变量的含义	108
8.2.1 Activity 中的 mToken	108
8.2.2 Window 中的 mAppToken	109
8.2.3 WindowManager.LayoutParams 中的 token	109
8.2.4 View 中的 token	110
8.3 创建应用窗口	111
8.4 创建子窗口	121
8.4.1 Dialog 的创建	122
8.4.2 PopupWindow 的创建	126
8.4.3 ContextMenu 的创建	127
8.4.4 OptionMenu 的创建	132
8.5 系统窗口 Toast 的创建	136
8.5.1 Toast 调用流程	137
8.5.2 Toast 添加窗口	139
8.6 创建窗口示例	139
第 9 章 Framework 的启动过程	142
9.1 Framework 运行环境综述	142
9.2 Dalvik 虚拟机相关的可执行程序	143
9.2.1 dalvikvm	144
9.2.2 dvz	144
9.2.3 app_process	145
9.3 zygote 的启动	147
9.3.1 在 init.rc 中配置 zygote 启动参数	147

4.11.1	git branch .....	72
4.11.2	git checkout .....	72
4.11.3	git log .....	73
4.11.4	git commit --amend .....	73
4.11.5	git cherry-pick sha-1 .....	73
4.11.6	git merge-base .....	74
4.11.7	git diff master...dev .....	74
4.11.8	git revert .....	75
4.11.9	git diff .....	75
4.11.10	git rm .....	75
4.11.11	git tag .....	76

## 第 2 部分 内核篇

第 5 章	Binder .....	78
5.1	Binder 框架 .....	78
5.2	设计 Servier 端 .....	80
5.3	Binder 客户端设计 .....	81
5.4	使用 Service 类 .....	82
5.4.1	获取 Binder 对象 .....	82
5.4.2	保证包裹内参数顺序 aidl 工具的使用 .....	83
5.5	系统服务中的 Binder 对象 .....	88
5.5.1	ServiceManager 管理的服务 .....	88
5.5.2	理解 Manager .....	90
第 6 章	Framework 概述 .....	92
6.1	Framework 框架 .....	92
6.1.1	服务端 .....	92
6.1.2	客户端 .....	93
6.1.3	Linux 驱动 .....	94
6.2	APK 程序的运行过程 .....	94
6.3	客户端中的线程 .....	94
6.4	几个常见问题 .....	95
6.4.1	Acitivity 之间如何传递消息（数据） .....	95
6.4.2	窗口相关的概念 .....	96

9.3.2	启动 Socket 服务端口	148
9.3.3	加载 preload-classes	151
9.3.4	加载 preload-resources	152
9.3.5	使用 fork 启动新的进程	152
9.4	SystemServer 进程的启动	155
9.4.1	启动各种系统服务线程	156
9.4.2	启动第一个 Activity	158
第 10 章	AmS 内部原理	160
10.1	Activity 调度机制	160
10.1.1	几个重要概念	161
10.1.2	AmS 中的一些重要调度相关变量	163
10.1.3	startActivity() 的流程	165
10.1.4	stopActivityLocked() 停止 Activity	183
10.1.5	按“Home”键回到桌面的过程	186
10.1.6	按“Back”键回到上一个 Activity	187
10.1.7	长按“Home”键	189
10.1.8	Activity 生命期的代码含义	190
10.2	内存管理	192
10.2.1	关闭而不退出	192
10.2.2	Android 与 Linux 的配合	194
10.2.3	各种关闭程序的过程	196
10.2.4	释放内存详解	197
10.3	对 AmS 中数据对象的理解	211
10.3.1	常见的对象操作	212
10.3.2	理解 Activity	213
10.3.3	Android 多进程吗, 是同时在运行多个应用程序吗	213
10.4	ActivityGroup 的内部机制	214
10.4.1	TabActivity 使用时的类关系结构	215
10.4.2	LocalActivityManager 的内部机制	217
10.4.3	ActivityGroup 内部的 Activity 生命期控制	220
第 11 章	从输入设备中获取消息	221
11.1	Android 消息获取过程概述	221
11.2	与消息处理相关的源码文件分布	223
11.3	创建 InputDispatcher 线程	226

11.4	把窗口信息传递给 InputDispatcher 线程	227
11.5	创建 InputChannel	229
11.6	在 WmS 中注册 InputChannel	232
11.7	在客户进程中注册 InputChannel	233
11.8	WmS 中处理消息的时机	234
11.9	客户窗口获取消息的时机	235
第 12 章	屏幕绘图基础	237
12.1	绘制屏幕的软件架构	237
12.2	Java 客户端绘制调用过程	239
12.3	C 客户端绘制过程	241
12.4	Java 客户端绘制相关类的关系	244
第 13 章	View 工作原理	247
13.1	导论	247
13.2	用户消息类型	249
13.2.1	按键消息	249
13.2.2	触摸消息	250
13.3	按键消息派发过程	252
13.3.1	KeyEvent.DispatcherState 中的长按监测	252
13.3.2	按键消息总体派发过程	254
13.3.3	根视图内部派发过程	256
13.3.4	Activity 内部派发过程	257
13.3.5	View 类内部的 onKeyDown()和 onKeyUp()	260
13.3.6	Activity 中的 onKeyDown()和 onKeyUp()	261
13.3.7	PhoneWindow 内部消息派发过程	262
13.4	按键消息在 WmS 中的派发过程	263
13.5	触摸消息派发过程	266
13.5.1	触摸消息总体派发过程	266
13.5.2	根视图内部消息派发过程	267
13.5.3	ViewGroup 内部消息派发过程	268
13.5.4	各种消息监测的基本实现方法	271
13.5.5	View 内默认消息派发过程	272
13.6	导致 View 树重新遍历的时机	274
13.6.1	状态的分类	274
13.6.2	导致 View 树重新遍历的总体诱因图	275

13.6.3	refreshDrawableList()	276
13.6.4	onFocusedChanged()	278
13.6.5	ensureTouchMode()	279
13.6.6	setVisibility()	282
13.6.7	setEnabled()	284
13.6.8	setSelected()	285
13.6.9	invalidate()	286
13.6.10	requestFocus()	290
13.6.11	requestLayout()	292
13.7	遍历 View 树 performTraversals()的执行过程	293
13.8	计算视图大小 (measure) 的过程	296
13.8.1	measure 内部设计思路	297
13.8.2	ViewGroup 中的 measureChildWithMargins()	301
13.8.3	LinearLayout 中的 onMeasure()过程举例	304
13.9	布局 (layout) 过程	308
13.9.1	layout 过程的设计思路	308
13.9.2	LinearLayout 中 onLayout()内部过程	309
13.9.3	TextView 中 gravity 与 layout 的关系	311
13.10	绘制 (draw) 过程	313
13.10.1	视图中可绘制的元素	313
13.10.2	绘制过程的设计思路	314
13.10.3	ViewRoot 中 draw()的内部流程	315
13.10.4	View 类中 draw()函数内部流程	318
13.10.5	ViewGroup 类中绘制子视图 dispatchDraw()内部流程	322
13.10.6	ViewGroup 类中 drawChild()过程	325
13.10.7	绘制滚动条	328
13.11	动画的绘制	331
13.11.1	动画的设计思路	332
13.11.2	ViewGroup 类中 drawChild()函数中视图动画绘制过程	334
13.11.3	ViewGroup 中 dispatchDraw()中布局动画绘制流程	337
第 14 章	WmS 工作原理	340
14.1	概述	340
14.1.1	窗口的定义	340
14.1.2	窗口管理要解决的核心问题	341
14.1.3	解决核心问题所使用的相关的变量列表	343



14.1.4	几个操作的概念	346
14.1.5	什么是 Policy, 以及其与 WmS 的关系	346
14.1.6	WmS 接口结构	347
14.2	WmS 主要内部类	348
14.2.1	表示窗口的数据类	348
14.2.2	DimAnimator	348
14.2.3	FadeInOutAnimation	349
14.2.4	InputMonitor 类	350
14.2.5	PolicyThread	351
14.2.6	Session	352
14.2.7	Watermark	353
14.2.8	WMThread	354
14.3	窗口的创建和删除	355
14.3.1	创建窗口的时机和过程	355
14.3.2	assignLayersLocked() 的执行过程	360
14.3.3	addWindowToListInOrderLocked() 的执行过程	362
14.3.4	删除窗口的时机	364
14.3.5	删除窗口的过程	366
14.3.6	removeWindowInnerLocked()	367
14.4	计算窗口的大小	371
14.4.1	描述窗口尺寸的变量	371
14.4.2	窗口大小的变化过程	372
14.4.3	Policy 中 layoutWindowLw() 的执行过程	375
14.4.4	输入法窗口如何影响应用窗口的大小	378
14.5	切换窗口	379
14.5.1	切换要解决的问题	379
14.5.2	InputManager 和 WmS 的接口	381
14.5.3	AmS 与 WmS 的接口	383
14.5.4	从 A 到 B 的切换	387
14.5.5	从 B 回到 A 的过程	390
14.5.6	A 中长按“Home”键切换到 B	391
14.5.7	setAppVisiblity() 与销毁 Surface	393
14.5.8	computeFocusedWindowLocked()	396
14.6	performLayoutAndPlaceSurfacesLockedInner() 的执行过程	398
14.6.1	总体过程	399

16.3.4	mSettings.readLP()	452
16.3.5	scanPackageLI()内部过程	454
16.3.6	mSettings.writeLP()	455
16.4	应用程序的安装和卸载	455
16.4.1	各主要功能类关系	456
16.4.2	应用程序安装过程	457
16.4.3	应用程序的卸载过程	461
16.5	intent 匹配框架	463
16.5.1	主要功能类的关系	463
16.5.2	主体调用过程	465
第 17 章	输入法框架	467
17.1	输入法框架组成概述	468
17.2	输入法中各 Binder 对象的创建过程	469
17.2.1	InputConnection	469
17.2.2	IInputMethodClient	471
17.2.3	InputMethodSession	472
17.2.4	InputMethod	475
17.3	输入法主要操作过程	477
17.3.1	输入法相关模块的启动过程	477
17.3.2	切换输入法	478
17.3.3	启动输入法	480
17.3.4	显示输入法	485
17.3.5	输入法操作过程中的重要变量总结	489
17.4	输入法窗口内部的显示过程	490
17.4.1	IMS 中的 showWindow()的内部执行过程	491
17.4.2	标准布局的 IMS	496
17.4.3	自定义布局的 IMS	502
17.5	向编辑框传递字符	503
17.6	输入法相关源码清单	504

## 第 4 部分 编译篇

第 18 章	Android 编译系统	508
18.1	Android 源码文件结构	509

14.6.2	第一大步骤：计算窗口的大小 .....	401
14.6.3	第二大步骤：计算窗口的可视状态 .....	401
14.6.4	第三大步骤：通知 SurfaceFlinger 进行窗口重绘 .....	404
14.7	窗口动画 .....	406
14.8	屏幕旋转及 Configuration 的变化过程 .....	409

## 第 3 部分 系统篇

第 15 章	资源访问机制 .....	414
15.1	定义资源 .....	414
15.2	存储资源 .....	415
15.3	styleable、style、attr、theme 的意义 .....	417
15.4	AttributeSet 与 TypedArray 类 .....	420
15.5	获取 Resources 的过程 .....	425
15.5.1	通过 Context 获取 .....	425
15.5.2	通过 PackageManager 获取 .....	429
15.6	Framework 资源 .....	431
15.6.1	加载和读取 .....	432
15.6.2	添加 .....	434
15.6.3	实现真正主题切换的两种思路 .....	436
第 16 章	程序包管理 (Package Manager Service) .....	439
16.1	包管理概述 .....	439
16.2	packages.xml 文件格式 .....	442
16.2.1	last-platform-version 标签 .....	443
16.2.2	permissions 标签 .....	443
16.2.3	cert 标签 .....	444
16.2.4	sigs 标签 .....	444
16.2.5	perms 标签 .....	444
16.2.6	package 标签 .....	444
16.2.7	shared-user 标签 .....	445
16.3	包管理服务的启动过程 .....	446
16.3.1	各主要功能类的关系 .....	446
16.3.2	PmS 主体启动过程 .....	448
16.3.3	readPermission() 内部过程 .....	450

18.2	从调用 make 命令开始说起 .....	509
18.2.1	编译命令 .....	510
18.2.2	编译结构猜想 .....	510
18.3	编译所需脚本文件之间的协同关系 .....	512
18.3.1	编译系统内部功能模块图 .....	512
18.3.2	脚本文件的包含关系 .....	514
18.3.3	从子项目中提取编译目标 .....	518
18.3.4	生成编译规则 .....	519
18.3.5	设置编译输出目录 .....	521
18.3.6	生成最终的 Image 文件 .....	522
18.4	如何增加一个 product .....	523
18.4.1	什么是一个 product .....	523
18.4.2	如何增加一个 product .....	527
18.5	如何增加一个项目 .....	528
18.5.1	项目类别和项目路径 .....	529
18.5.2	添加一个 C 项目 .....	530
18.5.3	添加一个 APK 项目 .....	531
18.6	APK 编译过程 .....	533
18.6.1	总体编译过程概述 .....	533
18.6.2	生成 R.java .....	535
18.6.3	编译 aidl 文件 .....	536
18.6.4	包含 Java 静态库 .....	536
18.6.5	编译 Java 源文件生成 Jar 包 .....	538
18.6.6	将 Jar 包转换为 dex 文件 .....	539
18.6.7	编译资源文件生成 APK 包 .....	540
18.6.8	将 dex 文件添加到 APK 包中 .....	541
18.6.9	添加 JNI 所需的动态库文件 .....	541
18.6.10	对 APK 文件进行签名 .....	543
18.6.11	使用 zipalign 优化 APK 内部存储 .....	543
18.7	Framework 的编译 .....	544
18.7.1	总体编译过程 .....	544
18.7.2	framework/core/ext 三个 Jar 文件的区别 .....	546
18.8	编译 android.jar .....	547
18.8.1	资源文件 .....	547
18.8.2	aidl 文件 .....	551

18.8.3	Java 文件	551
18.9	编译 adt 插件	553
18.10	总结	554
第 19 章	编译自己的 Rom	555
19.1	嵌入式系统的内存地址空间	555
19.2	各种映像 (Image) 文件的作用	559
19.3	编译 Nexus S (NS) 的 Image 文件	562
19.3.1	编译 Linux Kernel	562
19.3.2	提取 NS 的私有驱动文件	563
19.3.3	编译 system.img 文件	564
19.3.4	创建 ramdisk.img	565
19.3.5	创建 boot.img 文件	566
19.4	使用 fastboot 写入 Image 文件	566
19.5	最后验证	567
19.5.1	解决触摸按键问题	568
19.5.2	解决音量和电源键	568
19.5.3	WIFI 问题	570
19.5.4	安装 Google Mobile Service (GMS)	571

## 第 5 部分 硬件驱动篇

第 20 章	基于 TI OMAP 处理器的 Techshine 开发板介绍	573
20.1	Techv-35XX 开发板概述	574
20.2	交叉编译环境配置	575
20.3	x-loader 编译	578
20.4	u-boot 编译	578
20.5	Techv-35XX Linux 驱动和内核配置及编译	579
20.5.1	Touchscreen 驱动配置	579
20.5.2	KeyBoard 驱动配置	580
20.5.3	Audio 驱动配置	581
20.5.4	4MMC/SD 驱动配置	582
20.5.5	NandFlash 驱动配置	582
20.5.6	LCD 驱动配置	583
20.5.7	内核编译	583

20.6	Techv-35XX Android 驱动编写.....	584
20.7	Techv-35XX Android 开发环境建立.....	589
20.8	编译 Android Donut.....	590
20.9	Android 根文件系统的制作.....	591
20.10	相关 Image 文件的烧写.....	591
20.11	Android 根文件系统安装.....	593



## 第1部分 基础篇

- 第1章 Linux 基础
- 第2章 Java 基础
- 第3章 Android 源码下载及开发环境配置
- 第4章 使用 git





## 第 1 章 Linux 基础

基础篇  
第 1 章

Android Framework 的研究和开发必须在 Unix-like（和 Unix 很像）的主机系统上进行，常用的主机平台包括 Ubuntu 和 Mac OS。另外，Android 的底层任务管理及驱动都是基于 Linux 系统，所以，本书第一章首先来介绍一些 Linux 的基础知识。如果读者已经熟悉 Linux 系统，则可以跳过本章。

### 1.1 Linux 文件系统概述

估计大多数读者最初使用的都是 Windows 操作系统，并习惯了 Windows 中的文件系统。在以往的概念中，文件系统有以下特点：

- 文件系统是由文件及目录组成。
- 每个目录或者文件在磁盘上都对应了一定的存储空间。
- 每个目录或者文件都可以被复制或者删除，除了一些只读的系统文件外。
- 如果有多个存储实体，比如磁盘、U 盘，那么，它们将对应多个并列的、不同的文件系统。
- 文件系统有不同的类型，比如早期的 FAT16，以及后来的 FAT32、NTFS 等。

而在 Linux 系统中，文件系统的概念却与之有较大的差别。现在请开启电脑，一定要是 Unix-like 的系统，然后开启一个 Terminal，就是很像 Windows 中的 DOS 界面的一个终端程序，并执行以下命令：

```
$cd /  
$ls
```

第一个命令用于跳转到根目录，第二个命令用于列出当前目录下的所有文件。如果该目录下包含隐藏文件，则可以使用 `ls -a` 命令，参数 `-a` 用于显示所有文件。在 Unix-like 系统中，根目录一般包含如表 1-1 所示的目录。

表 1-1 Linux 系统根目录结构

目录名	描 述
bin	用户级二进制工具
boot	Linux 内核镜像文件，由 bootloader 程序读取并装载
dev	各种系统硬件设备
etc	系统配置文件及其他配置文件
home	用户工作目录
lib	系统运行时所需的各种库文件
opt	操作系统附带的一些应用程序
proc	内核及进程所虚拟的系统文件
root	管理员工作目录
sbin	与 bin 的区别在于，该目录下的二进制工具程序仅用于管理员，s 的含义是 system administrator
sys	一般是驱动程序对应的虚拟系统文件
usr	管理安装的、所有用户都可以访问的应用程序
var	系统运行时所产生的一些调试信息文件或者相关统计文件

Unix-like 系统中“文件系统”的概念包含两个意思，第一是“根文件系统”，第二是“存储类文件系统”。后者的概念基本等同于 Windows 操作系统，而前者则与 Windows 差别较大，它并不是用于存储实际文件的。本节就来讨论“根文件系统”的概念，根文件系统简称 rootfs (Root File System)，它有如下特点。

第一，“文件”不仅是指硬盘上的数据，它还包括任何设备资源。在 Unix-like 系统中，所有的硬件设备都被看做是文件，“文件”是内核范畴的概念，磁盘、U 盘、内存、网络，甚至 CPU 都被内核抽象为“文件”。为了区别于一般意义上的文件，内核级别的文件被称为“设备文件”或“设备虚拟文件”，这些设备文件在 rootfs 目录中可以被看到，表面上就像 Windows 系统中的磁盘文件。

第二，并不是所有的目录或文件都对应磁盘上的存储空间。比如，sys、proc、dev 这三个目录，它们对应的不是存储空间，而是设备文件，这三个目录中的内容由内核及相应的驱动程序维护。

第三，“存储类文件系统”不能和 rootfs 并列存在，而只能挂载到 rootfs 下的一个子目录上，这与 Windows 系统中的文件系统完全不同。新建一个目录，然后把一个磁盘中的文件系统和这个目录连接起来，这在 Windows 系统中是不可想象的，而在 Unix-like 系统中，这却是标准的挂载一个“存储类文件系统”的方法。

第四，Unix-like 中“存储类文件系统”内部则等同于 Windows 中的文件系统，包括文件系统类型。Windows 系统中常见的文件系统类型包括 FAT16、FAT32、NTFS，Linux 系统也支持这些文件系统类型，但更常见的却是 ext2、ext3、ext4、yaffs 等。

Unix-like 系统中，操作系统只能有一个根文件系统，但可以包含多个“存储类文件系统”。假设，某 Linux 系统根目录下的 home 目录中有以下三个文件路径：

```
/home/user1/work  
/home/user1/enjoy/sports  
/home/user2/jokes
```

此时有一个磁盘，该磁盘上有三个分区，如果在 Windows 系统中查看该磁盘的话，其内部包含 E、F、G 三个磁盘。而在 Linux 中，则可以把这三个分区分别挂载到以上三个路径下，从而，用户会在以上三个路径下查看到和 Windows 系统下 E、F、G 三个分区完全相同的内容。唯一的区别是，在 Linux 中用户不能通过这三个文件夹来查看磁盘的信息，而在 Windows 系统中，用户是可以通过 E、F、G 直接查看磁盘信息的。

执行挂载、卸载“存储文件系统”的操作可以在 Terminal 下使用 `mount` 和 `umount` 命令。

## 1.2 Linux 启动过程

Linux 启动过程对于初学者而言有点扑朔迷离，这是因为启动过程关乎处理器配置、内存配置、外围硬件配置，而不同的处理器和硬件系统会采用不同的策略，从而具体的启动过程会有所差异。但无论差异如何，从计算机系统的角度来看，启动过程一般分为三个步骤，如图 1-1 所示。

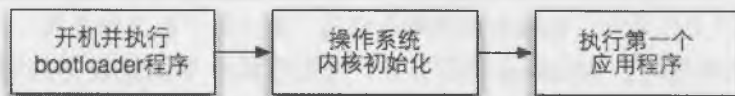


图 1-1 通用系统的启动过程

首先是开机，开机就是给系统开始供电，此时硬件电路会产生一个确定的复位时序，保证 CPU 是最后一个被复位的器件。为什么 CPU 要最后被复位呢？因为，如果 CPU 第一个被复位，则当 CPU 复位后开始运行时，其他硬件内部的寄存器状态可能还没有准备好，比如磁盘或者内存，那么就可能出现外围硬件初始化错误。

当正确完成复位后，CPU 开始执行第一条指令，该指令所在的内存地址是固定的，这由 CPU 的制造者指定。不同的 CPU 可能会从不同的地址获取指令，但这个地址必须是固定的，这个固定地址所保存的程序往往被称为“引导程序” (Bootloader)，因为其作用是装载真正的用户程序。

至于如何装载，则是一个策略问题，不同的 CPU 会提供不同的装载方式，比如有的是通过普通的并口存储器，有的则是通过 SD 卡，还有的是通过 RS232 接口。无论硬件上使用何种接口装载，装载过程必须提供以下信息，具体包括：

- 从哪里读取用户程序？
- 用户程序的长度是多少？
- 装载完用户程序后，应该跳转到哪里，即用户程序的执行入口在哪里？

不同硬件系统会采用不同的策略，但只要以上三个信息是确定的，用户程序就会被装载到确定的地址，并执行相同的操作。

(续表)

函数名称	所在内核源码路径	描述
decompress_kernel()	./arch/XXX/boot/compress/misc.c	C 语言函数, 用于解压内核, Linux Kernel 的映像文件是一个 zlib 压缩格式的数据, 因此需要先解压
startup_32()	./arch/XXX/kernel/head.S	进行 CPU 的页表配置, 主要用于虚拟内存, 并检测该 CPU 是否有浮点处理单元(FPU)支持, 此时该进程为系统进程, 即 0 号进程
start_kernel()	./init/main.c	内核内部数据初始化, 配置中断向量表, 并挂载 ramdisk, 最后调用 kernel_thread()方法
kernel_thread()	./arch/XXX/kernel/process.c	该函数会根据 ramdisk 中的一个叫做 init.rc 的文本文件的配置内容, 启动不同的应用程序, 第一个启动的程序就是第一个用户级的程序
cpu_idle()	./init/main.c	此时, 内核已经可以按照进程的优先级进行调度了, 当没有其他进程运行时, 就调用该方法

以上过程中, init.rc 文件在 Android 手机中的系统根目录下, 可以使用 adb pull 命令提取出该文件:

```
./adb pull /init.rc ~/Desktop
```

init.rc 的内容格式类似于一种脚本, 但是它却不是标准的 Linux 脚本, 而是仅用于启动的脚本。关于 init.rc 的完整格式请参考 Linux 相关文档。

## 1.3 常用 Linux 命令

Linux 中的命令非常多, 本节仅介绍一些在开发 Android 过程中常用的 Linux 命令。

- man

man 的含义是 manual, 即手册。当我们不清楚某个 Linux 命令的作用和用法时, 可以使用 man command 进行查询, command 为具体的命令名称, 比如 man ls。

- find

find 命令用于查找某个文件或者文件夹, 比如:

```
$find . -name "*.java"
```

该命令用于查找当前目录下扩展名为 Java 的所有文件, find 命令后面的“点”代表当前目录, \*为通配符, 代表任何名称。

- grep

grep 命令为正则表达式匹配命令, 该命令用于字符串匹配。比如, 想查找 hello.java 文件中包含“Activity”字符串的所有地方, 可使用如下命令。

```
$grep "Activity" hello.java
```

grep 和 find 的区别在于, find 用于查找目录或者文件, 而 grep 用于查找指定的字符串, 并且字符

第二步是执行内核程序，这里所说的内核程序在上一步中指的就是“用户程序”。因为从 CPU 的角度来看，除 Bootloader 之外的所有的程序都是用户程序，只是从软件的角度来看，用户程序被分为“内核程序”和“应用程序”，而本步执行的是“内核程序”。

内核程序初始化时执行的操作包括，初始化各种硬件，包括内存、网络接口、显示器、输入设备，然后建立各种内部数据结构，这些数据结构将用于多线程调度及内存的管理等。当内核初始化完毕后，就开始运行具体的应用程序了。在一般情况下，习惯于将第一个应用程序称为“Home 程序”。

第三步就是运行 Home 程序，比如 Windows 系统的桌面，就是一个典型的 Home 程序。之所以称其为 Home 程序，是因为通过该程序可以方便地启动其他应用程序。而传统的 Linux 系统启动后，第一个运行程序一般是一个 Terminal，尽管它表面上就像一个 Dos 界面，但它也可以被称为 Home 程序，因为 Home 程序设计的目标就是提供一个入口，用户可以通过该入口启动其他应用程序。

以上从通用操作系统的角度介绍了启动过程，接下来，具体来看 Android 所使用的 Linux 内核的启动过程。因为目前的 Android 系统多运行在 ARM 处理器之上，因此，下面主要分析运行于 ARM 处理器上的 Linux 的启动过程。在介绍之前，先来简单区分三个概念：ARM、处理器、CPU。

ARM 本身是一个公司的名称，从技术的角度来看，它又是一种微处理器内核的架构。

处理器是一种统称，可以指具体的 CPU 芯片，比如 Intel 8086 处理器，苹果的 A8 处理器等。处理器内部一般包含 CPU、片上内存、片上外设接口等不同的硬件逻辑。

CPU 是处理器内部的中央处理单元的缩写，CPU 可以按照类型分为短指令集架构和长指令集架构两大类，ARM 属于短指令集架构的一种。

对于 ARM 处理器，当复位完毕后，处理器首先执行其片上 ROM 中的一小块程序。这块 ROM 的大小一般只有几 KB，该段程序就是 Bootloader 程序，这段程序执行时会根据处理器上一些特定引脚的高低电平状态，选择从何种物理接口上装载用户程序，比如 USB 口、串口、SD 卡、并口 Flash 等。

多数基于 ARM 的实际硬件系统，会从并口 NAND Flash 芯片中的 0x00000000 地址处装载程序。对于一些小型嵌入式系统而言，该地址中的程序就是最终要执行的用户程序；而对于 Android 而言，该地址中的程序还不是 Android 程序，而是一个叫做 uboot 或者 fastboot 的程序，其作用是初始化硬件设备，比如网口、SDRAM、RS232 等，并提供一些调试功能，比如向 NAND Flash 中写入新的数据，这可用于开发过程中的内核烧写、升级等。

当 uboot (fastboot) 被装载后便开始运行，它一般会先检测用户是否按下了某些特别按键，这些特别按键是 uboot 在编译时预先约定好的，用于进入调试模式。如果用户没有按这些特别的按键，则 uboot 会从 NAND Flash 中装载 Linux 内核，装载的地址是在编译 uboot 时预先约定好的。

Linux 内核被装载后，就开始进行内核初始化的过程，该过程如表 1-2 所示。

表 1-2 Linux 内核的启动步骤

函数名称	所在内核源码路径	描 述
start()	./arch/XXX/boot/head.S	汇编语言，进行一些 CPU 寄存器的配置，并调用 startup_32()函数
startup_32()	./arch/xxx/boot/compress/head.S	汇编语言，配置堆栈，并对 BSS 段进行清空

串可以由正则表达式描述。关于正则表达式请参考其他相关资料。

- xargs。

确切的说, xargs 并不是一个命令, 而是一个标识, 代表了上一个命令的执行结果, 并作为下一个命令的参数。Linux 命令可以流水线执行, 也叫做“多管道”执行, 即两个命令用“|”分隔符隔开。比如, 想查找当前目录下文件名中包含“oa”的所有文件, 可使用以下命令:

```
$ls | xargs grep "oa"
```

该命令中 ls 用于列出所有文件, “多管道”后面的命令把前面命令的输出结果作为后面命令的参数。再举一个例子, 想查找当前目录下所有包含“Activity”字符串的 Java 文件, 命令如下:

```
$find . -name "*.java" | xargs grep "Activity"
```

再举例, 想查找当前目录下子目录名称为 res 下的所有.xml 文件, 并且列出这些.xml 文件中包含“status\_bar\_size”字符串的地方。注意, 本例的限制条件是, 查找的是 res 目录下的.xml 文件, 而不是所有的.xml 文件。命令如下:

```
$find `find . -name res` -name "*.xml" | xargs grep status_bar_size
```

该命令中, find 命令进行了嵌套使用, 内部 find 包含在“`”符号之中。注意这不是单引号, 而是键盘上数字键 1 左边对应的那个符号。内嵌的 find 命令用于查找名称为 res 的目录或文件, 外部的 find 命令从得到的这些目录下继续查找.xml 文件, “多管道”后面的命令用于查找指定的字符串。

- cat。

cat 命令用于连接文件内容并在 Terminal 中输出文件内容, 该命令后面如果只有一个文件名称, 则仅输出该文件内容。cat 参数中的文件可以是普通意义上的有存储空间的文件, 也可以是 Linux 系统中的设备文件。比如, 可以查看当前目录下 main.java 的文件内容, 使用如下命令:

```
$cat main.java
```

也可以查看设备文件, 以下命令用于查看/dev/disk0 的内容:

```
$sudo cat /dev/disk0
```

该命令中 sudo 的含义是使用管理员权限执行后续的命令, 因为有些命令要求有管理员的权限。

- chmod。

在 Linux 系统中, 文件的访问者被划分为三类, 并针对这三类用户指定不同的访问权限。这三类访问者是:

- user (u), 用户自身, 即创建该文件的用户。
- group (g), 用户所在组, 即与创建在一个组里面的用户。
- other (o), 其他用户。

chmod 命令就是用于设置这三类访问者对某文件的访问权限。访问权限分为读 (r) 权限、写 (w)

权限、执行 (x) 权限，文件类型不同，“执行”的含义不同。可以通过 `ls -l` 命令查看文件的访问权限，Linux 系统使用 10 位 (bit) 数据表示访问的权限，比如某文件的访问权限如下。

```
$ ls -l mybin
total 32
-rw-r--r--@ 1 keyd staff 73 Mar 9 23:32 Makefile.mk
drwxr-xr-x 4 keyd staff 136 Aug 27 2010 bash
```

- bit 0: 使用 `-` 或者 `d` 表示，前者表示这是一个文件，后者表示这是一个目录 (directory)。
- bit 1~bit 3: 用户自身 (user) 对该文件的访问权限。
- bit 4~bit 6: 用户组 (group) 对该文件的访问权限。
- bit 7~bit 9: 其他用户 (other) 对该文件的访问权限。

举例，某个文件的属性为 `-rwxrwx---`，这表示它是个文件，用户对该文件拥有读取、写入、执行权限，用户组对该文件也拥有读取、写入、执行权限，而其他用户则不能读取、不能写入、不能执行。

再举一个例子，`dr-x-x---` 表示它是一个文件夹。用户对该文件夹拥有读取、执行权限，但不能修改；用户组对该文件拥有执行权限，即只能打开该文件夹，但不能读取和修改；其他用户则不能读取、不能修改，也不能打开该文件夹。

下面来看 `chmod` 命令如何修改文件的访问权限。假设有一个脚本文件 `copy.sh`，创建该文件时，默认访问权限不包括执行 (x)，但要运行该脚本，用户必须拥有对该文件的执行权限，于是使用以下命令为用户添加执行权限：

```
$ chmod u+x copy.sh
```

命令中 `u+x` 的含义是给 user 添加执行 (x) 的权限，类似的也可以是 `o+x`、`o+rw`、`g+rw` 等。为了便于使用，可以用 `a` 代表三类用户，比如 `a+x`、`a+rw` 等，这里的 `a` 代表 all，即所有三类用户。

对于某些 Linux 系统，比如 Android 手机底层的 Linux，`chmod` 命令不识别 `r`、`w`、`x` 这样的参数，而只能使用 8 进制数字值来表示，比如：

```
$ chmod 777 copy.sh
```

其中的 777 为 8 进制的数，对应的二进制数据为 111 111 111，这就分别代表了 `u`、`g`、`o` 三类用户的访问权限。

- `ps`, `kill`。

`ps` 用于列出当前运行的所有进程，`kill` 用于杀死某个进程。这两个命令多用于系统调试，比如，可以先用 `ps` 列出所有进程，从输出信息中得到每个进程的 id 值，即 `pid`，然后调用 `kill -9 pid`，就可以杀死指定 `pid` 对应的进程。`-9` 是一个参数，详情可使用 `man kill` 查看。

- `export`。

该命令用于将某个变量值的作用域设为全局范围。比如，可以将某个路径赋值给系统环境变量 `PATH`，然后再 `export PATH`，从而，其他所有程序都可以使用该路径。

该命令还有另外一个巧用。有些公司为了资产管理的需求，往往会给每一台 PC 机进行编号、命名，



比如 `xxxcom_yyy part_username`, 这就导致打开 Terminal 后提示符之前的字符串太长。此时, 可以使用 `export PS1=me` 命令, `PS1` 一般代表了当前 Terminal, 等号后面的字符串则任由用户定制, 它将代替那串冗长的提示符。执行完后, 用户会发现 Terminal 变得简单多了。

## 1.4 Shell 脚本备忘

要全面介绍 Shell 脚本需要单独的一本书, 而这不是本书, 这就是为什么本节名称中使用“备忘”的原因。本节主要介绍脚本的基本概念及常用内容, 以做个备忘, 供以后参考。

读者常常会听到“脚本”这个词语, 比如 `JavaScript` 被称之为 `Java` 脚本 (尽管这与 `Java` 语言没有丝毫关系), `PHP` 语言也被称为 `PHP` 脚本, 那么, 到底什么是脚本?

脚本的英文是 `script`, 该单词的含义是“剧本”的意思, 即“一场戏的演出流程”。脚本本身不代表任何计算机语言, 也不属于任何计算机语言的范畴, 但到目前为止, 人们往往把那种不需要编译的程序文件称之为脚本。举个反例, `C` 语言在执行前必须要经过编译、链接, 然后才能执行, 所以它不是脚本, `Java` 语言也需要编译, 因此也不是脚本, 而对于 `JavaScript` 语言, 它的代码是不需要编译的, 而是被“解释”执行, 脚本运行时需要一个“解释器”, 解释器能够读懂预先定义好的脚本语法, 并执行之。

所有的脚本有两个共同的特点:

脚本内容是字面上可以读懂的文本文件 (Human Readable)。

不同的脚本必须由不同的解释器解释执行, 而脚本的语法则由解释器的设计者来定义。

在 `Linux` 系统中, 常用的脚本包括 `Bash`、`Perl`, 本书仅讲解 `Bash` 脚本的使用, `Perl` 脚本请参考其他文档。

前面说过, `script` 的本义是“剧本”, 因此, 可以想象, 脚本文件的结构应该是一个流程, 剧本中应该包含不同角色要表演的动作。`Linux` 命令其实就是“剧本”中的角色动作, `script` 解释器除了能够导演这些角色外, 还定义了一些条件判断语法, 角色可以根据不同的条件进行表演, 这种条件判断的语法就像 `C` 语言中的 `if/else`、`switch/case` 语句, 不过具体的使用方法则有所差别。下面是一个简单的脚本文件:

```
#!/bin/bash
echo "Hell bash" #display one message
```

把以上代码保存为一个文本文件, 名称为 `me` 命令, 文件的扩展名可以任意, 但常用 `.sh` 作为扩展名, 意思是 `shell`, 或者干脆不要扩展名, 因为 `Linux` 系统不像 `Windows` 那样“在乎”扩展名。保存后, 执行 `chmod a+x me` 命令, 为 `me` 添加执行权限, 然后, 可以在 Terminal 中运行该脚本:

```
$/me
Hello bash
```

在该脚本文件中, 第一行 `#!/bin/bash` 这个语句是必需的, 并且必须在首行, 从而操作系统让该文件选择 `Bash` 来解释执行脚本文件。如果是 `Perl` 脚本, 第一行则为 `#!/bin/perl`。

第二行中的 `echo` 是一个 `Linux` 命令, `#` 后面是注释。

在脚本中可以调用各种 Linux 命令，因此，脚本的用途非常广泛，凡是需要按一定的次序执行多个 Linux 命令的场合都可以使用脚本来完成。下面分别介绍脚本中的一些基本语法。

### 1.4.1 获取输入

输入包含三种，第一种是执行脚本时用户的输入；第二种是将前一个脚本的输出作为该脚本的输入；第三种是脚本函数的参数，关于函数参数见后面小节。

用户的输入可以用 \$n 表示，n 为 1~9 自然数，分别代表输入中的第 n 个参数。比如“echo i love you”，该命令中第一个参数为 i，第二个参数为 love，第三个为 you，参数是以空格或者 Tab 键分隔的。举例如下：

```
#!/bin/bash
echo $1
echo $2
echo $3
```

执行该脚本，输出为：

```
$. /me i love you
i
love
you
```

### 1.4.2 变量定义

就像 C 语言一样，脚本语言也可以定义变量，但脚本中的变量类型一般比较简单，所以在 Bash 脚本中的变量没有类型，所有的变量都是字符串。变量不要单独定义，可直接赋值，赋值语句中不能有空格。引用变量时只需在变量前加一个 \$ 符号即可。为了避免引用混淆，常常使用双引号包含要引用的变量，如下代码所示：

```
#!/bin/bash
A=b
All=all
echo "$A"ll
echo $All
```

该段代码的两次输出分别为 bll 和 all，因为第一句 echo 语句中引用了变量 A，而第二句中则引用了变量 All。

### 1.4.3 条件判断

条件判断主要是判断两个字符串是否相等、两个数字是否相等。其语法格式如下代码所示：

```
#!/bin/bash
if [ "$1" = "normal" ]
then
    echo "this is normal case"
elif [ -z "$1" ]
then
    echo "no input, input..."
fi
```

该段代码判断用户输入的第二个参数，如果其值等于“normal”字符串，则显示“this is normal case”，如果第二个参数为空，则显示“no input, input...”。

条件判断语法中需要注意以下几点：

- 每个 if 语句后面的执行部分必须跟在 then 后面，这似乎有点多余，但是 Bash 脚本就是这么规定的。
- 多个判断分支可以使用 elif 语句。
- 条件语句必须使用 fi 结束。
- 条件语句中 “[” 符号后面必须要有一个空格，比如 if [ "\$1" = "normal" ] 不能写为 if ["\$1" = "normal"]，也就是说在 “[” 和 “\$1 = normal” 之间都必须要有空格。其原因是 Bash 脚本中 “[” 符号并不是简单的一个中括弧，而是一个可执行文件（命令），该命令是一个条件判断命令。这个听起来有点奇怪，但如果把 “[” 想象为一个函数名称，比如 test，则该语句可写为 if test "\$1" = "normal"，这样看起来似乎更容易理解。

在以上代码中，对于第一个条件判断语句 if [ "\$1" = "normal" ]，有人可能会问，是否可以写为 if [ "\$1" = "23" ]，即用数字代替字符串？答案是肯定的，不过实际上并没有用数字代替字符串，这里的“23”加了双引号，因此，会按照字符串处理。如果写成 if [ "\$1" = 23 ] 是否可以呢？答案也是可以的，但这里使用的依然不是数字，而还是“23”字符串，只是在编码时，为了避免混淆，对于字符串一般都使用双引号，这与 C 语言中使用字符串是不同的。那么到底如何才能把 23 当做数字进行对比呢？答案是使用“-eq”，比如 if [ "\$1" -eq "23" ]，Bash 脚本的比较语句中严格区分操作数，字符串的比较和整数的比较使用不同的操作符，如表 1-3 所示。

表 1-3 Bash 脚本的比较运算符

操作符	返回 true 的条件	操作数个数
-n	操作数的长度不为零	1
-z	操作数的长度为零	1
-d	操作数对应一个目录	1
-f	操作数对应一个文件	1

(续表)

操作符	返回 true 的条件	操作数个数
-eq	操作数为整数，并且相等	2
-neq	操作数为整数，但不相等，与 -eq 相反	2
=	操作数为字符串，并且相等	2
!=	操作数为字符串，不相等，与 = 相反	2
-lt	小于(less than)，操作数为整数	2
-gt	大于(great than)，操作数为整数	2
-ge	大于等于(great equal)，操作数为整数	2
-le	小于等于(less equal)，操作数为整数	2

#### 1.4.4 while [...]do... done 语句

该语句的作用类似于 C 语言中的 do...while 语句，举例如下：

```
#!/bin/bash
echo "please use add or delete or exit"
ACTION="default"
while [ -n $ACTION ]
do
    read ACTION
    case $ACTION in
        add)
            echo "add somebody"
            ;;
        delete)
            echo "delete somebody"
            ;;
        exit)
            echo "complete"
            break
            ;;
        *)
            echo "invalid action, please re-enter"
            ;;
    esac
done
```

以上脚本的作用是根据用户输入的参数，执行不同内容。

while 语句和 if 语句中的条件判断格式完全相同。read 是一个 Linux 命令，用于提示用户输入并以按回车键 (LF) 结束输入。输入的内容保存到变量 ACTION 中，接下来的 case 语句根据 ACTION 的值，执行不同的动作。

in 本身也是一个 Linux 命令，每一个 case 分支用一个具体的值表示，并带一个小括弧，case 分支

也可以有通配符，即星号，每一个分支的结束符号是两个分号。esac 也是一个 Linux 命令，这里我们可以理解为 case 语句的结束标识。

要退出整个 while 循环，可以使用 break 语句，done 语句是整个 while 语句的结束标识。

### 1.4.5 for 循环

Bash 中的 for 循环类似于 Java 中的 foreach 语句，for 一般和 in 联合使用，用于从某个集合中逐个取出元素并对其进行操作，如以下代码所示：

```
#!/bin/bash
for X in 1 2 3 4 5 hello
do
    echo $X
done
```

该段代码显示 in 集合中的所有元素，in 语句后面紧跟目标集合，X 是一个变量名称，for 语句的结束标识是 done。

再举一个例子，如以下代码所示：

```
#!/bin/bash
for X in `ls`
do
    echo `basename $X`
    echo `dirname $X`
done
uname
```

在这段代码中，in 集合的来源是 Linux ls 命令，即当前目录下所有文件。需要注意的是如果要把 Linux 的某个命令的输出作为集合的输入，或者作为其他命令的输入，需要把该命令包含在“”符号之中，该符号是键盘上!符号左边的那个按键；而如果仅仅是执行某个 Linux 命令则不需要“”符号，如以上代码最后一句 uname，该命令用于显示操作系统的内部名称。

代码中 basename 和 dirname 命令用于显示一个字符串中的路径和文件名称。比如，字符串 /usr/local/lib/libexpat.so，对应的 dirname 是 /usr/local/lib，对应的 basename 是 libexpat.so。basename 和 dirname 本身并不去检测该文件是否真正存在于磁盘上，而仅仅是针对字符串的操作。

下面以 for 语句和 if 语句为例，写了一段脚本，功能是给当前目录下没有扩展名的文件加上.txt 扩展名，这个工具在实际工作时有可能会用到。

```
#!/bin/bash
#rename files without ext name to txt file.
#author: Yuandan Kerr
for X in `ls`
do
    Base=`basename $X`
    if [ -z `echo $Base | grep "\. "` ]
```

```

then
    `mv $Base $Base.txt`
    echo $Base
fi
done

```

### 1.4.6 函数

Bash 脚本中的函数与 C 语言中的函数在参数传递及返回值方面有很大不同，看以下一个例子。

```

#!/bin/bash
strcat()
{
    OUT="$1"" "$2"
    return 0
}

strcat2()
{
    echo "$1"" "$2"
    return 3
}

A="bird"
B="mouse"

OUT=""
strcat $A $B
echo $OUT
OUT2=`strcat2 $A $B`
echo $?
echo $OUT2

```

该脚本定义了两个函数，分别为 `strcat` 和 `strcat2`，脚本中的函数具有以下特点：

- 定义函数时，不需要定义参数，在函数实体中可以直接使用 `$1`、`$2` 代表输入的第 `n` 个参数。
- 函数中可以使用 `return` 关键字返回整数数值，但 `return` 返回数值并不能通过等号赋值给函数的调用者，事实上 `return` 返回的是函数执行完毕的退出值，就像 C 库中的 `exit(int)` 函数。如以上代码中 `OUT2=`strcat2 $A $B`` 语句，该语句并不是将 `strcat2` 中的 `return 3` 赋值给 `OUT2`。而是将 `strcat2` 中的 `echo "$1"" "$2"` 输出的结果赋值给 `OUT2`，也就是说，函数必须使用 `echo` 命令返回内容给调用者，同时，由于脚本不支持字符串的加号操作，所以 `echo "$1"" "$2"` 不能写成 `echo "$1"+"$2"`，后者会输出 `bird+mouse`。
- 调用函数时，如果要获得函数的输出，并且赋值给某个变量，则函数的执行必须包含在 `` 符号之间，该符号是键盘上数字 1 按键左边的那个按键。
- 如果要获得函数的返回值而不是输出，即函数中 `return` 关键字后面的数值则可以使用  `$?`  符号。该符号代表了上一个命令的返回值，所谓的上一个命令可以是一个函数，也可以是一个 Linux 命

令。在一般情况下, `return` 关键字返回的数值应该用于表明脚本的执行状态结果, 比如成功执行、失败执行等。

以上代码的输出结果为:

```
bird mouse
3
bird mouse
```

### 1.4.7 常用内置符号常量

在脚本的解释器中, 往往预先定义了一些特殊符号, 这些符号具有特殊的含义, 有时也称之为脚本解释器的内置符号。可将其理解为脚本语法的一部分, 因为它们是解释器定义的, 比如上一小节例子中的 `$?` 代表上一个命令或函数的返回值。

内置符号非常多, 具体可以参考 Bash 脚本手册, 地址为: <http://www.gnu.org/software/bash/manual/bashref.html>。

表 1-4 列出了一些常用的符号。

表 1-4 内置符号常量

符号值	意 义
<code>\$@</code>	代表全部参数, 比如 <code>test a b c</code> , 此处为 “a” “b” “c”, 展开后为三个字符串
<code>\$*</code>	全部参数, 比如 <code>test a b c</code> , 此处为 “a b c”, 展开后为一个字符串
<code>\$#</code>	参数的个数, 比如 <code>test a b c</code> , 此处为 3
<code>\$?</code>	上一个命令的返回结果, 如果上一个命令是一个脚本函数, 则对应函数中 <code>return</code> 的数值, 比如 <code>ls   echo "\$?"</code> , 结果为 0
<code>\$\$</code>	当前命令所在的进程号 (PID)

除了以上内置符号外, Bash 脚本中还定义了一些内部变量。比如 `UID`, 代表执行该脚本的用户 ID, 可直接使用 `echo $UID` 显示当前用户的 ID 值, 如果当前用户为 `root` 用户, 那么 `UID` 的值为 0。

## 1.5 Make 脚本备忘

Linux 系统中包含一个 Make 脚本的解释器, 它可以读取 Make 脚本的内容, 并执行之, Make 脚本多用于自动编译过程, 但并不是说 Make 脚本只能用于自动编译。本节仅介绍 Make 脚本的基本概念和一般用法, 关于 Make 脚本内部的更多信息请参考 *GNU Make* 一书, 作者为: Richard M. Stallman, Roland McGrath 和 Paul D. Smith。

Make 脚本的基本语法如下:

```
目标(target): 条件(prerequisite)
      (Tab 键) 命令
```



在该语法中，目标可以是任意一个字符串名称，也可以是具体文件的名称。条件可以是其他目标的名称，也可以是具体文件的名称。执行 Make 脚本时，Make 解释器会检查目标和条件中包含的文件时间戳是否相同，如果不同的话，解释器就会执行 Tab 键后面的“命令”，命令可以是任何可执行程序。

自动编译的基本原理就是将目标文件作为“目标”，将源文件作为“条件”，因此，当源文件修改后，目标文件的时间戳就会早于源文件，于是 Make 解释器就会自动执行指定的“命令”。此时可以将执行编译的命令作为这里的“命令”，从而达到自动编译的目的。

因此，只要是想基于文件的时间戳来自动执行一些操作的场合，都可以考虑使用 Make 脚本，这就是为什么 Make 脚本不仅仅可以用于自动编译的原因。

下面来看 Make 脚本的编写方法。

### 1.5.1 一个简单的 Makefile 文件

本节以一个简单的 Makefile 文件为例，来说明 Makefile 的使用方法，该文件名称为 Makefile，其源码如下：

```
#Filename Makefile
#this file is used for show how to use makefile
$(info start working)
hello: hello.c
    echo "nothing"

hello.bin: hello.c
    @echo "now make hello.bin"
    gcc hello.c -o hello.bin

.PHONY: he
he: hello.c
    @echo "now make he"
    gcc hello.c -o hello.bin
```

这段代码有以下特点：

- #符号是注释符，可用在代码中任何地方。
- 第三行中\$是函数调用符号，info 是一个函数名称，作用是输出一段信息。类似的信息输出函数还包括 warning、error 两个函数，不过 error 函数执行后会终止执行并退出。
- 目标定义前不能加任何空格，而命令行前必须以 Tab 键开始。
- .PHONY 关键字用于声明一个目标，被 .PHONY 声明的目标将总是执行其指定的命令，而如果不声明的话，则仅当目标后面的条件变动后才执行。
- 命令前面的@符号的作用是，不显示被执行的命令。因为在默认情况下，Make 解释器在执行命令时会打印出执行的命令。

- 对于 `hello.bin` 目标，该目标本身就是一个文件，其依赖的文件是 `hello.c` 文件。因此当 `hello.c` 文件被修改后，将会执行 `gcc` 命令重新对该 `c` 文件编译，并输出 `hello.bin` 文件。

要执行以上脚本，可运行以下命令：

```
$ make -f Makefile hello
```

在该命令中，`-f` 参数用于指定要执行的脚本文件名称。如果不指定文件名称，则解释器会自动从当前目录下寻找名称为 `Makefile` 的脚本文件，如果找不到，则执行失败。

`hello` 代表要执行的具体目标，因为一个脚本文件中可能包含多个目标的定义。如果不指定目标，则解释器默认执行脚本中定义的第一个目标。指定的目标不同，执行的命令也将不同。

下面是不同目标对应的执行结果，首先是当不指定目标，并连续两次调用：

```
$ make
start working
echo "nothing"
nothing
$ make
start working
echo "nothing"
nothing
```

接着是执行 `hello.bin` 目标，并连续两次调用：

```
$ make hello.bin
start working
now make hello.bin
gcc hello.c -o hello.bin
$ make hello.bin
start working
make: 'hello.bin' is up to date.
```

最后是执行 `he` 目标，并连续两次调用：

```
$ make he
start working
now make he
gcc hello.c -o hello.bin
$ make he
start working
now make he
gcc hello.c -o hello.bin
```

以上执行结果说明了两个问题：

- 对于 `hello` 和 `he` 目标，因为它们不是文件名称，所以每次 `Make` 该目标时，都会执行指定的命令；而对于 `hello.bin`，由于它是一个文件，因此，只有当 `hello.c` 被修改后才会执行编译命令。
- 对于 `hello` 和 `he` 目标，由于这两个目标不是文件，所以其条件中所指定的 `hello.c` 文件其实并没有什么意义，其对应的命令总是会被执行。

1.5.2 变量的定义与赋值

Makefile 中的变量不需要单独定义，可直接赋值，常见赋值方式如表 1-5 所示。

表 1-5 Make 脚本中的变量赋值符号

赋值符号	意 义
:=	简单展开型，它在该 Makefile 被解析时就立即展开并赋值
=	递归展开型，该赋值方式只有当所定义的变量在使用的时候才展开
?=	条件赋值，只有当该变量还没有值时才赋值
+=	附加赋值

Make 解释器执行脚本的过程可分为两个步骤：

① 装载 Makefile 及 Makefile 中 include 的其他 Makefile。装载完所有相关的脚本文件后，系统内部会创建一张图，该图描述了各个 Makefile 的依赖关系。

② 根据用户指定的 target 找出该 target 的全部依赖关系，并判断依赖条件中的文件的时间戳。如果时间戳较新，就开始执行 target 所对应的命令。

而对于变量定义和赋值，解释器会根据不同的赋值方式选择是立即赋值还是延迟赋值，赋值过程也称为展开过程。所谓立即赋值，是指在读取脚本时就给变量进行赋值；而延迟赋值，是指读取时暂不赋值，只有在执行脚本时用到了该变量，才对其进行赋值。不同赋值方式对应的展开时机如表 1-6 所示。

表 1-6 Make 脚本中变量的展开时机

定 义	展 开 a	展 开 b
a = b	立即	延迟
a ?= b	立即	延迟
a := b	立即	立即
a += b	立即	当该语句出现在 target 后面时，则会延迟，而在一般的赋值时，则立即展开
define a b... b... b... endef	立即	延迟

1.5.3 条件控制语句

Make 脚本的条件控制可分为两类，一类是在解释器解析脚本文件时处理，另一类是在执行脚本时处理。

本节讲的是第一类条件控制语句，它更像是 C 语言中的 `ifdef` 预处理语句。关于第二类条件控制，实际上是使用函数进行条件控制，即函数的功能就是条件控制，这一点比较特别，大家要注意区分。

下面来看第一类条件控制语句的语法模型，如以下代码所示：

```
if-condition
text if the condition is true
endif
```

或者

```
if-condition
text if the condition is true
else
text if the condition is false
endif
```

其中 `condition` 只能进行两种判断，一种是判断表达式是否相等，另一种是判断表达式是否被定义，如下：

- `ifdef var`，判断变量 `var` 是否被定义过。如前所述，脚本并不需要单独定义一个变量，只要给该变量赋值过，那么它就被定义了。
- `ifndef var`，与 `ifdef` 相反，判断变量 `var` 是否还没有被定义。
- `ifeq test`，判断表达式 `test` 是否相等，表达式可写为 `"a" "b"` 或 `(a, b)`。
- `ifneq test`，与 `ifeq` 相反。

#### 1.5.4 宏（函数）定义

**Make** 脚本中函数，按被调用的方式可分为三类。

第一类是内置函数，即 **Make** 解释器内部定义好的函数，在任何脚本文件中可直接调用，调用的格式为：

```
$(fname, param...)
```

`fname` 是函数的名称，`param` 是参数，多个参数用逗号分隔。

第二类是用户定义的、带参数的函数，使用 `define` 关键字进行定义，调用的格式为：

```
$(call fname, param...)
```

`call` 是调用的关键字，`fname` 代表函数名称，`param` 是函数参数，多个参数使用逗号分隔。

第三类也是用户定义的，但不带参数，该类函数也称之为宏，其调用的格式为：

```
$(fname)
```

既不使用 `call` 关键字，也不包含参数。

用户函数的定义方式如下：

```
define fname
各种具体的命令
endif
```

在自定义函数中，命令前不需要加 Tab 键，因为当宏被展开时，Make 解释器会自动在每一行命令前加 Tab 键。函数内部使用 \$(n) 代表调用函数时的参数，n 为自然数，\$(0) 代表函数名称本身，\$(1) 代表第一个参数，\$(2) 代表第二个参数。

下面以一个例子来说明自定义函数的使用：

```
define showFirstName
    @echo $(1)
endif

.PHONY: name
name:
    $(call showFirstName,Yuandan, Kerr)
```

这段代码中定义了函数 showFirstName，其作用是将传入的第一个参数返回给调用者。使用 make name 命令执行以上脚本，执行结果为：

```
$ make name
Yuandan
```

下面将介绍一些常用的内置函数。

### 1. 字符串操作函数

内置的常用字符串函数如表 1-7 所示。

表 1-7 常用字符串操作函数

函数用法举例	执行结果
\$(filter pattern..., text)	提取出 text 中所有满足 pattern 模板的独立字符串，并以空格分开。比如当参数为 *.c, fl.c fl.h f2.c 时，其结果为 fl.c f2.c
\$(filter-out pattern..., text)	与 filter 的执行效果相反
\$(findstring string, text)	从 text 中寻找指定的字符串 string。比如当参数为 he, hello he haiii 时，其结果为 "he"。注意，如果没有找到，则其中包含一个空字符
\$(subst from,to,text)	把 text 字符串中出现的 from 替换为 to
\$(patsubst from,to,text)	把 text 字符串中出现的、满足模板 from 中条件的字符串替换为 to。比如参数为 %.c,%o,x.c.c bar.c 的执行结果为 x.c.o bar.o
\$(strip text)	去掉 text 字符串前后的空格，类似于 Java 中的 trim 函数
\$(words text)	计算出 text 中字符串的个数，比如 \$(words i love android) 的结果为 3

(续表)

函数用法举例	执行结果
<code>\$(word n,text)</code>	提取出 <code>text</code> 中的第 <code>n</code> 个字符串, <code>n</code> 从 1 开始, 比如 <code>\$(word 2,i love android)</code> 的结果为 <code>love</code> 字符串
<code>\$(firstword text)</code>	提取出 <code>text</code> 中的第一个字符串
<code>\$(wordlist start,end,text)</code>	提取出 <code>text</code> 中从 <code>start</code> 到 <code>end</code> 的字符串, 比如 <code>\$(wordlist 1,2,i love android)</code> 的结果为 <code>i love</code>
<code>\$(sort list)</code>	对 <code>list</code> 进行字母顺序排列, 比如 <code>\$(sort z1 abc acd)</code> 的结果为 <code>abc acd z1</code>

## 2. 文件名称操作

用于处理文件路径、名称的常用函数如表 1-8 所示。

表 1-8 常用文件名称处理函数

函数用法举例	执行结果
<code>\$(wildcard *.c)</code>	列出当前目录下所有的指定类型文件, 此处列出所有 <code>.c</code> 文件, 比如 <code>f1.c f2.c main.c</code>
<code>\$(dir f1.c src/f2.c)</code>	提取出指定文件的路径, 此处为 <code>./ src/</code>
<code>\$(notdir f1.c src/f2.c)</code>	与 <code>dir</code> 作用互补, 提取出文件名称, 此处为 <code>f1.c f2.c</code>
<code>\$(suffix f1.c f1.h)</code>	提取出文件的后缀, 此处为 <code>.c .h</code>
<code>\$(basename f1.c src/f1.h)</code>	与 <code>suffix</code> 作用互补, 提取出文件名称, 此处为 <code>f1 src/f1</code>
<code>\$(addsuffix .c, f1 src/f2)</code>	给文件添加上指定后缀名称, 此处为 <code>f1.c src/f2.c</code>
<code>\$(addprefix src/, f1.c f2.c)</code>	给文件添加指定前缀, 此处为 <code>src/f1.c src/f2.c</code>
<code>\$(join f1 f2, .c .h)</code>	连接前后列表, 此处为 <code>f1.c f2.h</code>

## 3. 过程控制函数

前面介绍过 `ifeq`、`ifdef` 等条件判断语句, 它们仅仅是在脚本被解析的时候起作用, 有点 C 语言中“预处理”的意思。而如果要在脚本运行时动态地进行条件判断, 则需要使用过程控制函数, 这一点和 C 语言中的 `if/else` 截然不同。在 C 语言中 `if/else` 是语法本身定义的关键字, 而脚本中的过程控制却是由函数完成的。

过程控制函数如表 1-9 所示。

表 1-9 过程控制函数

函数用法举例	执行结果
<code>\$(if condition, then-part, else-part)</code>	<code>condition</code> 为 <code>true</code> 的条件是其不为空, 比如 <code>\$(if i love android, i am android fans, i am not)</code> 的结果是 <code>i am android fans</code>

(续表)

函数用法举例	执行结果
\$(error text)	退出函数，执行该函数后，make 停止执行，并输出 text 文本，输出的信息同时会包含当前正在执行的脚本文件名称及所在的行号
\$(foreach variable, list, body)	从 list 中提取出每一个字符串，并将其赋值给 variable 变量，然后执行 body 中的逻辑。比如\$(foreach name, zhao wang zhao qian, \$(shell echo hello \$(name)),)的执行结果是 hello zhao, hello wang, hello zhao, hello qian

1.5.5 内置符号和变量

Make 解释器内部定义了一些特别的符号和一些特别名称的变量，在编写用户脚本时，可以直接使用这些符号、变量，而不需要定义。

1. 内置符号

常见的符号变量如表 1-10 所示。

表 1-10 Make 脚本中的内置符号

符 号	意 义
\$@	target 的名称
\$*	和\$@类似，只是不包含 target 的后缀
^	所有的先决条件名称，以空格(space)分隔，如果先决条件中有重复，则自动去除重复
?	有更新的先决条件列表，比如，有 4 个先决条件，其中有 3 个刚刚被修改过，则\$?就代表刚刚被修改过的先决条件，以空格分隔
+	所有的先决条件，和\$^类似，只是\$+包含了重复的先决条件，而\$^会自动去除重复的先决条件
<	第一个先决条件名称

2. 内置变量

除了内置符号变量外，还有一些固定名称的内置变量，这些变量如表 1-11 所示。

表 1-11 Make 脚本中的内置变量

变量名称	意 义
MAKE_VERSION	make 版本
CURDIR	执行 make 时的目录
MAKEFILE_LIST	本次 make 命令执行时，所有被包含的 makefile 列表
VARIABLE	所有的变量列表

因为 Make 脚本主要是用于对 C/C++源码的编译，因此，其内部也定义了一些专用于 C/C++编译的变量，如表 1-12 所示。

表 1-12 用于 C/C++ 编译的内置变量

变量名称	意 义
CC	C 编译器，默认是 gcc
CXX	C++ 编译器，默认是 g++
CXXFLAGS	C++ 的编译选项
CPPFLAGS	仅为 .cpp 文件的编译选项
TARGET_ARCH	目标主机架构，比如 arm、x86 等，默认为空
LDLIBS	连接器库选项

### 1.5.6 模板目标 (Pattern target)

读到这里，读者已经可以编写基本的 Make 脚本了，下面以一个例子来说明什么是模板目标。

假设当前有一个 C 源码的项目，其中包含 3 个 C 源文件，名称分别为 f1.c、f2.c、main.c，其中 f1 和 f2 中定义了两个函数，main.c 中会使用这两个函数，然后可以编写一个脚本文件，脚本源码如下：

```
.PHONY: test
test: f1.o f2.o main.o
    gcc -o main.bin f1.o f2.o main.o

f1.o: f1.c
    gcc f1.c f1.o

f2.o: f2.c
    gcc f2.c -c f2.o

main.o: main.c
    gcc main.c -c main.o
```

可以想象，如果源码数量较多，则脚本文件的定义就会变得繁琐起来。读者可以发现，对于每一个 C 文件的编译基本上是相同的，因此，能否用一种简单的办法来描述这种规则呢？这就是“模板目标”的作用，即使用一种“模板”来定义目标，使用模板目标后，以上脚本可简化如下：

```
OBJ = f1.o f2.o main.o
.PHONY: test
test: $(OBJ)
    gcc $(OBJ) -o main.bin

%.o: %.c
    gcc -c -o $@ $<
```

%.o 就是“模板目标”，百分号就是模板通配符，意思是所有 .o 文件。除了通配符外，代码中另外一个关键是使用符号常量 \$@ 和 \$<，前者代表目标名称，后者代表第一个先决条件的名称。使用模板



目标后，脚本文件就简化了许多。

### 1.5.7 目标特定的变量赋值 (Target-specific variable)

在脚本文件中，当给某个变量赋值后，则之后无论 Make 哪个目标，该变量的值都是相同的，如以下代码所示：

```
CFLAGS = -c
.PHONY: tar1
tar1 :
    gcc $(CFLAGS) main.c

tar2 : CFLAGS =
tar2 :
    gcc $(CFLAGS) main.c
```

CFLAGS 被赋值为 -c，其值在整个脚本文件范围内都是有效的。比如当 Make tar1 时，就会执行 gcc -c main.c。但是在 tar2 目标中，我们希望能够执行 gcc main.c，因此可以在 tar2 目标中对 CFLAGS 变量重新赋值，该赋值仅在 tar2 目标的命令中有效。这就是所谓的“目标特定”变量赋值，因为该赋值仅针对该目标。

语法上需要注意，对 tar2 目标的赋值不能直接写成以下形式：

```
tar2 : CFLAGS =
    gcc $(CFLAGS) main.c
```

而是必须分开写，即把目标变量赋值和目标规则分开写。

特定变量赋值解决了特定目标中变量赋值的问题，但有时用户却希望强制使用命令行中变量的赋值，而不是特定目标中的赋值，于是在调用 make 命令时使用 -e 选项，比如：

```
$ make tar2 -e CFLAGS="-c -g"
gcc -c -g main.c
```

但是，有些目标却必须使用目标特点的变量赋值，因此，可以在变量赋值前加 override 关键字，如下代码所示：

```
tar2 : override CFLAGS =
tar2 :
    gcc $(CFLAGS) main.c
```

这样就导致强制使用目标特定的赋值，执行结果如下：

```
$ make tar2 -e CFLAGS="-c -g"
gcc main.c
```

以上过程听起来就像是一场战斗，战斗的双方是“Make 解释器”和“特定目标”：起初，特定目标

希望有特定的变量值,但是 Make 解释器不允许,便使用 `-e` 武器阻止,最后特定目标不屈服,使用 `override` 超级武器,特定目标胜利了。

### 1.5.8 常用选项

调用 `make` 命令时,可以在命令后加一些选项,以进行不同的操作。这些选项特别多,详情可使用 `man make` 命令查看,本节仅介绍以下几种常用选项,这些选项在分析 Framework 内核时有重要的作用。

- `--environment-overrides (-e)`: 强制使用 `make` 命令参数中的变量赋值。
- `--touch(-t)`: 仅仅更新文件的时间戳,却不执行 `target` 对应的命令。
- `--directory(-c)`: 指定要执行的 Makefile 的路径,在默认情况下,Make 解释器仅从当前目录下寻找指定的脚本文件。
- `--file(-f)`: 指定要执行的 Makefile 的名称,如果没有指定,则 Make 解释器会自动寻找名称为 `Makefile` 或 `Makefile.mk` 的文件。
- `--just-print(-n)`: 仅仅打印出要执行命令,而不真正执行目标对应的命令。该选项在分析 Android 编译系统中很有用,可以避免冗长的编译过程。
- `-l` 的参数: 指定编译时所需的 `lib` 文件。比如 `lName` 的搜寻方式为先寻找 `libName.so`,如果没有找到,则寻找 `libName.a`。



## 第 2 章 Java 基础

本章简要介绍相对于 C 语言的 Java 特别语法，适应于有 C 语言基础但却不熟悉 Java 的读者。本章虽说讲解的是 Java 基础内容，但却不是分析 Framework 的必要基础，因此如果第一次阅读时觉得难以理解，则可先跳过本章，以后需要的时候再回头阅读。

### 2.1 类装载器 DexClassLoader

在 Java 环境中，有个概念叫做“类装载器”(ClassLoader)，其作用是动态装载 Class 文件。标准的 Java SDK 中有一个 ClassLoader 类，借助它可以装载想要的 Class 文件，每个 ClassLoader 对象在初始化时必须指定 Class 文件的路径。

没有使用过 ClassLoader 的读者可能会问：“在过去的程序开发中，当我们需要某个类时，只需要使用 import 关键字包含该类就可以了，为什么还要类装载器呢？”简单的讲，import 中所引用的类文件有两个特点：

- 必须存在于本地，当程序运行时需要该类时，内部类装载器会自动装载该类，这对程序员来讲是透明的，即程序员感知不到该过程。
- 编译时必须在现场，否则编译过程会因找不到引用文件而不能正常编译。

但在有些情况下，所需要的类却不能满足以上两个条件。比如当该类是从远程下载并在本地执行时，典型的例子就是通过浏览器中的 AppleLet 执行的 Java 程序，这些要执行的程序是在服务器端。另一种情况是，要引用的 Class 文件不方便在编译时直接参与，而只能在运行时动态调用。举例来讲，在 Android Framework 中，所包含的 Class 文件是一些通用的类文件，但对于一些设备商而言，他们需要扩充 Framework，扩充的具体工作包括两点：

- 需要增加一些额外的类文件，这些类文件提供厂商自定义的功能，这些文件一般以独立的 Jar 包存在。
- 需要修改 Framework 中的已有类文件，比如 WindowManagerService 类，在该类中添加使用自定义 Jar 包中的代码。使用自定义 Jar 包的常用方法是使用 import 关键字包含自定义的类，但为了保持和原生 Framework 的兼容性、对原生 Framework 最少化修改，可以使类装载机动态装载自定义 Jar 包。

这就是使用 ClassLoader 的原因。

在一般情况下，应用程序不需要创建一个全新的 ClassLoader 对象，而是使用当前环境已经存在的 ClassLoader。因为 Java 的 Runtime 环境在初始化时，其内部会创建一个 ClassLoader 对象用于加载 Runtime 所需的各种 Java 类。

每个 ClassLoader 必须有一个父 ClassLoader，在装载 Class 文件时，子 ClassLoader 会先请求其父 ClassLoader 加载该 Class 文件，只有当其父 ClassLoader 找不到该 Class 文件时，子 ClassLoader 才会继续装载该类，这是一种安全机制。关于 ClassLoader 的内部过程，大家可以参考《Inside the Java Virtual Machine》一书，作者为 Bill Venners，相关链接如下：

<http://www.artima.com/insidejvm/ed2/index.html>。

对于 Android 的应用程序，本质上虽然也是用 Java 开发，并且使用标准的 Java 编译器编译出 Class 文件，但最终的 APK 文件中包含的却是 dex 类型的文件。dex 文件是将所需的所有 Class 文件重新打包，打包的规则不是简单地压缩，而是完全对 Class 文件内部的各种函数表、变量表等进行优化，并产生一个新的文件，这就是 dex 文件。由于 dex 文件是一种经过优化的 Class 文件，因此要加载这样特殊的 Class 文件就需要特殊的类装载机，这就是 DexClassLoader，Android SDK 中提供了 DexClassLoader 类就是出于这个目的。

下面来看 DexClassLoader 的使用方法。

### 2.1.1 DexClassLoader 的调用方法

本小节以一个例子来说明 DexClassLoader 的使用方法。假设有两个 APK，第一个叫做 Host，第二个叫做 Plugin，其中 Plugin 中定义了一个类 PluginClass，该类中定义了一个函数 function1()，其代码如下：

```
public class PluginClass {  
    public PluginClass(){  
        Log.i("Plugin", "PluginClass client initialized");  
    }  
  
    public int function1(int a, int b){  
        return a + b;  
    }  
}
```

本例将要演示如何在 Host 中使用 DexClassLoader 来动态装载 PluginClass 类，并调用其 function1() 函数。

得到 Plugin.apk 后将其安装到 Android 设备中，然后再新建一个 Host 工程，该工程中主 Activity 的调用代码如下：

```
public void useDexClassLoader() {
    Intent intent = new Intent("com.haiii.android.plugin.client", null);
    PackageManager pm = getPackageManager();
    final List<ResolveInfo> plugins = pm.queryIntentActivities(intent, 0);
    ResolveInfo rinfo = plugins.get(0);
    ActivityInfo ainfo = rinfo.activityInfo;

    String div = System.getProperty("path.separator");
    String packageName = ainfo.packageName;
    String dexPath = ainfo.applicationInfo.sourceDir;
    String dexOutputDir = getApplicationInfo().dataDir;
    String libPath = ainfo.applicationInfo.nativeLibraryDir;

    DexClassLoader cl = new DexClassLoader(dexPath, dexOutputDir,
        libPath, this.getClass().getClassLoader());
    try {
        Class<?> clazz = cl.loadClass(packageName + ".PluginClass");
        Object obj = clazz.newInstance();
        Class[] params = new Class[2];
        params[0] = Integer.TYPE;
        params[1] = Integer.TYPE;
        Method action = clazz.getMethod("function1", params);
        Integer ret = (Integer)action.invoke(obj, 12, 34);
        Log.i("Host", "return value is " + ret);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

DexClassLoader 构造函数的参数意义如下：

- dexPath，指目标类所在的 APK 或 Jar 文件的路径。类装载器将从该路径中寻找指定的目标类，该路径必须是 APK 或 Jar 的全路径，比如/data/app/com.haiii.android.plugin.apk。如果要包含多个路径，路径之间必须使用特定分隔符进行分隔，这个特定分隔符可使用 System.getProperty("path.separator") 获得。
- dexOutputDir，由于 dex 文件被包含在 APK 或者 Jar 文件中，因此在装载目标类之前需要先从 APK 或 Jar 文件中解压出 dex 文件，该参数就是指定解压出的 dex 文件存放路径。在 Android 系统中，一个应用程序一般对应一个 Linux 用户 id，应用程序仅对属于自己的数据目录路径有写的权限，因此，该参数可以使用该程序的数据路径，本例中，该路径具体是 /data/data/com.haiii.android.pluginhost。
- libPath，指目标类中所使用的 C/C++ 库存放的路径。
- 最后一个参数是指该装载器的父装载器，一般为当前执行类的装载器。

创建了 DexClassLoader 对象后，接下来就可以调用 loadClass() 装载指定的类了，该函数返回的是一

个 Class 对象，注意区分 Class 对象和目标类 PluginClass 对象。Class 对象是 ClassLoader 所能识别的类，而 PluginClass 是程序执行后所能识别的类，此时仅仅装载了 PluginClass 的程序代码，却还没有创建出 PluginClass 对象，因此接下来调用 Class 对象的 newInstance() 方法。该方法内部会调用 PluginClass 的构造函数，并返回一个真正的 PluginClass 对象。

尽管返回的是一个 PluginClass 对象，但是 Host 本地却没有任何 PluginClass 的定义，因此不能直接去调用 PluginClass 对象的任何方法，而只能使用通用的反射机制去调用 PluginClass 中的方法。

反射机制调用时主要使用了 Method 类，Class 对象的 getMethod() 方法可以返回该类中的任何方法，比如本例中的名称为 function1 的方法。getMethod() 函数有两个参数，第一个是需要访问的函数名称，后面一个参数是该函数参数的类型。

得到目标类的函数对象的 Method 后，就可以调用该 Method 对象的 invoke() 方法。该方法的第一个参数是指执行目标函数的对象，此处该对象就应该是真正的 PluginClass 对象，该对象是调用 Class 类的 newInstance() 创建的。

以上代码执行后，Log 输出为：

```
05-14 19:24:39.961: INFO/Plugin(704): PluginClass client initialized
05-14 19:24:53.751: INFO/Host(704): return value is 46
```

### 2.1.2 基于类装载器设计一种“插件”架构

首先，读者先别考虑“插件”架构，一起回顾一下上一节内容。

在上一节中，读者可以感觉到，通过 ClassLoader 装载的类，调用其内部函数的过程有点烦琐，包括首先构造出 Method 对象，并构造出 Method 对象所使用的参数对象，然后才能调用。有没有一种办法，既能通过动态装载，利用动态装载的灵活性，又能像直接类引用那样方便地调用其函数？

仍以上一节的内容为例，可以使用如下的一种方法。

首先定义一个 interface 接口，interface 仅仅定义函数的输入和输出，却不定义函数的具体实现。该 interface 类一方面存在于 Plugin 项目中，另一方面也存在于 Host 项目中，即该类同时参与两个项目的编译。

使用该办法后，Plugin 项目中原来的代码可修改如下：

```
public class PluginClass implements Comm{
    public PluginClass(){
        Log.i("Plugin", "PluginClass client initialized");
    }

    public int function1(int a, int b){
        return a + b;
    }
}
```

其中 Comm 接口的定义如下：

```
public interface Comm {
    public int function1(int a, int b);
}
```

修改完毕后，可重新安装 Plugin.apk。

接下来在 Host 项目中，对于 Class 对象 newInstance() 返回的对象就可以强制转换为 Comm 接口对象了，相关代码如下：

```
try {
    Class<?> clazz = cl.loadClass(packageName + ".PluginClass");
    Comm comm = (Comm)clazz.newInstance();
    Integer ret = comm.function1(12, 34);
    Log.i("Host", "return value is " + ret);
}
```

该段代码的执行结果和上一节相同。

下面再来看“插件”的概念，插件是一个逻辑概念，而不是什么技术标准。总的来讲，插件的概念包含以下意思：

- 插件不能独立运行，而必须运行于一个宿主程序中，即由宿主程序去调用插件程序。
- 插件一般可以独立安装。
- 宿主程序中管理不同的插件，包括查看插件的多少、禁用或使用某个插件，如果多个插件的功能是互斥的，则可以切换插件。
- 宿主程序应该保证插件的向下兼容性，即新版本的宿主程序可以运行较老版本的插件，或者说较老版本的插件能够在新版本的宿主程序中运行。

由于 ClassLoader 具有动态装载程序的特点，因此，可以使用该技术来实现一种插件架构。下面从插件的角度来审视一下上面的这个 Host 和 Plugin 项目的代码结构，如图 2-1 所示。

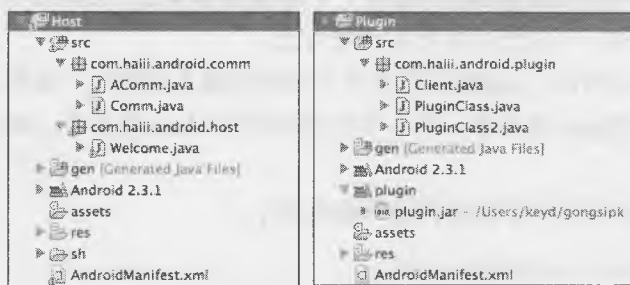


图 2-1 Host 和 Plugin 项目中的文件结构

该结构的特点如下：

- 接口类一般被定义在 Host 项目中，比如本例中的 Comm.java。
- Plugin 项目中需要引用 Comm 类时，则必须通过一个外部的 Jar 包，并且该 Jar 包必须是以 Library 的方式被添加到 Plugin 项目的 build path 路径中的，而不能以“外部 Jar”方式添加，如图 2-2

所示。原因是外部 Jar 会作为程序的一部分被打包到最终的程序文件中，从而使得 Plugin 和 Host 项目中存在包名相同但验证码不同的类文件，这最终会导致运行时出现错误信息“Class ref in pre-verified class resolved to unexpected implementation”。

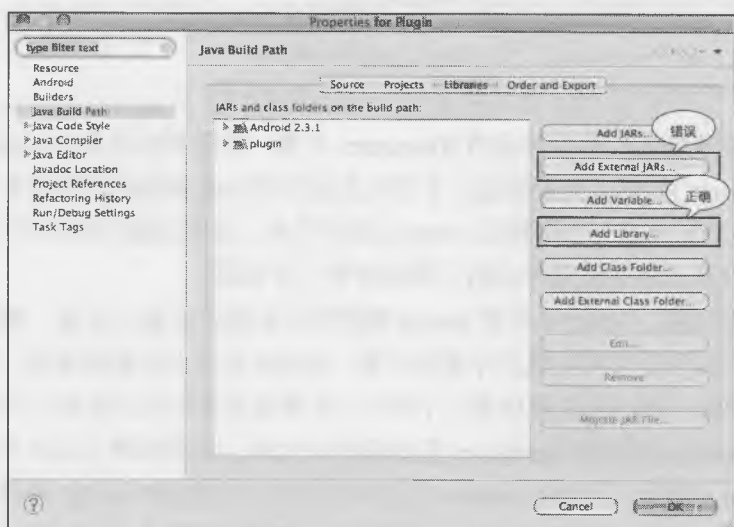


图 2-2 在 Plugin 项目中添加接口定义 Jar 包

该 Jar 包的内容正是 Host 项目中所定义的 Comm 类，为方便操作，可使用以下脚本产生该 Jar 包，该脚本文件一般可以放在 Host 项目中的 sh 目录中，目录名称任意。

```
#!/bin/sh
cd /Users/keyd/gongsipk/Host/bin
jar -cvf ~/gongsipk/plugin.jar ./com/haiii/android/comm
```

在该代码结构中，宿主程序 Host 要想知道系统中都有哪些插件，可以定义一个特定的 action 字符串，本例中使用“com.haiii.android.plugin.client”。每一个插件项目内部必须定义一个空 Activity，并在 AndroidManifest.xml 文件中声明对该 action 的处理，如以下代码所示：

```
<activity android:name=".Client"
    android:label="@string/app_name">
    <intent-filter>
        <action android:name="com.haiii.android.plugin.client" />
    </intent-filter>
</activity>
```

这样的话，宿主程序中就可以使用 PackageManager 类的 queryIntentActivities() 函数查询相关的插件程序列表了。

为了保证宿主程序和插件的兼容性，插件中往往需要声明该插件的版本号及一些名称信息，这些信



息则可以被包含在插件程序的 res 目录中，比如在插件程序的 res/values/string.xml 文件中定义一些相关的信息。信息的名称必须事先和宿主程序约定好，这样的话，宿主程序就可以通过以下代码获取插件相关的信息：

```
res = pm.getResourcesForApplication(packageName);
int id = 0;
id = res.getIdentifier("version", "string", packageName);
String version = res.getString(id);
```

这段代码中，首先获得插件程序对应的 Resources 对象，接着得到名称为 version 字段的字符串 id 值，然后再调用 getString() 获得该变量的值，于是宿主程序就知道插件程序的版本号了。在插件程序的 res/values/string.xml 中应该定义一个名称为 version 的字符串，表明该插件的版本号。

笔者曾使用该插件架构设计过两种应用，在此介绍一下思路。

第一种是系统主题架构。Android 中的 theme 概念并不是真正的系统主题，笔者对系统主题的定义是，所有的系统图标可以通过一种主题文件进行切换，比如状态栏上的系统图标、各种 UI 控件的图标、桌面等。设计思路是把每一种主题文件作为一个插件，在系统设置中可以使用不同的插件作为当前系统主题，然后修改 Framework 中读取 Resources 资源的相关代码，达到切换主题的目的。

第二种是 App Store 架构。标准的 Android 电子商店程序只能连接 Google 的服务器，但在中国，有很多厂商都希望能提供独立的应用商店，但如果让用户安装多个商店客户端显然是比较麻烦的事情。因此可以设计一个通用的电子商店客户端，并定义一种标准的和电子商店服务器通信的机制，把这种通信机制定义为一个或多个 interface 文件，不同商店厂商只要实现这些 interface 即可。然后，把实现后的程序作为一个插件安装到手机中，从而使得用户可以选择不同的插件，而不再需要安装多个客户端。

## 22

## JNI 调用机制

Java Native Interface (JNI) 是 Java 本地接口，所谓的本地 (native) 一般是指 C/C++ (以下统称 C) 语言。当使用 Java 进行程序设计时，一般主要有三种情况需要 C 语言的协助。

- 调用驱动。由于操作系统所提供的驱动一般都是 C 接口，Java 语言本身不具备操作这些驱动的能力。
- 对于某些大量数据处理的模块，Java 的效率可能远低于 C，因此，程序员希望使用 C 去完成。
- 对于某些功能模块，可能 Java 和 C 的效率差不多，但是这些模块已经存在已有的 C 代码，程序员不想再用 Java 重写，而只想重新利用已有的 C 代码。

这就是 Java 提出 JNI 概念的原因。无论出于什么原因，从程序的角度来看，JNI 接口主要包含两种情况。第一种是从 Java 中访问 C，第二种是从 C 中访问 Java，只要解决了这两个问题，那么就可以任意进行 Java 和 C 的应用组合。Framework 中大量使用 JNI 完成本地接口的实现，因此，理解本节内容对于后面阅读 Framework 代码将非常有帮助。

### 2.2.1 Java 访问 C

Java 中可以定义某个函数为 `native` 类型, 对于 `native` 函数, 只需要声明即可, 因为该函数的实现是 `native` 的, 即由相应的 C 去实现。Java 编译器遇到 `native` 函数时, 不会关心该函数的具体实现, 因此, 编译上不会出任何差错。

程序运行时, 在调用 `native` 方法之前, 程序员必须把 C 所生成的动态库装载进来, 否则程序会因为找不到相应的 `native` 方法而出错。关于如何把 C 程序编译为动态库, 将在第 18 章中的 Make 系统中介绍, 这牵扯到包含 `jni.h` 头文件等及编译选项的设置。

当调用 `native` 函数时, Java 会自动产生一个对应的 C 中的函数名称, 因为 Java 中声明的函数名称和 C 中实现的函数名称是不同的。其关系为, 后者等于包名加前者的名称, 并且中间以下画线分隔, 比如, Framework 中 `AssetManager` 类中声明了以下方法:

```
private native final void init();
```

该方法在 C 中对应的是:

```
static void android_content_AssetManager_init(JNIEnv* env, jobject clazz)
```

这种映射关系并不是 Java 编译器内含的, 程序员完全可以改变, 但这是一种编程规范。事实上, 当 Java 调用 `native` 时, 编译器会向 `native` 引擎传递调用者的包名, 以及函数名称, 还有参数类型, 仅此而已, 当然这也足矣, `native` 引擎根据这些信息决定应该具体调用哪个本地函数。`native` 引擎中 `AndroidRuntime` 类提供了一个 `registerNativeMethods()` 函数, 可以通过该函数来定义 Java `native` 函数和 C 函数名称的映射关系。

在产生的 C 函数中, 会包含至少两个参数。前者是 `JNIEnv` 对象, 该对象是一个 Java 虚拟机(JVM)所运行的环境, 相当于 JVM 的“管家”, 通过它可以访问 JVM 内部的各种对象; 第二个参数 `jobject` 是调用该函数的对象, 本例中指的是 `AssetManager` 对象。

如果 `native` 声明的函数本身也有参数, 那么, 这些参数会依次放在以上两个参数的后面, 比如 `native` 中包含以下方法:

```
private native final String[] getArrayStringResource(int arrayRes);
```

其对应的 C 函数为:

```
static
jobjectArray android_content_AssetManager_ getArrayStringResource (JNIEnv* env, jobject clazz,
jint arrayResId);
```

在以上的转换关系中, 大家可能会注意到 `native` 中所使用的类型和 Java 中有所不同。比如 Java 中的 `int` 在 `native` 中为 `jint`, 返回值中 `String[]` 变为 `jobjectArray`, 这些具体的定义实际上是在 `jni.h` 中, 该文件所在的路径为:

```
./dalvik/libnativehelper/include/nativehelper/jni.h
./external/webkit/WebKit/android/JavaVM/jni.h
```

```

./ndk/build/platforms/android-3/arch-arm/usr/include/jni.h
./ndk/build/platforms/android-4/arch-arm/usr/include/jni.h
./ndk/build/platforms/android-5/arch-arm/usr/include/jni.h
./ndk/build/platforms/android-5/arch-x86/usr/include/jni.h
./ndk/build/platforms/android-8/arch-arm/usr/include/jni.h
./ndk/build/platforms/android-8/arch-x86/usr/include/jni.h

```

即不同的 `platforms` 会有不同的定义，这个很好理解，因为 Java 的类型是跨平台的，而各自平台的 CPU 数据宽度是不同的，所以必须有各自的类型定义。

以上介绍了 Java 和 C 函数的名称转换，那么，具体怎么操作呢？在程序设计时，如果你已经定义好了 Java 代码，如何实现相应的 native C 代码呢？

你可能会想：“那就按照这种转换关系，手工编写相应的 C 代码，然后编译成动态库，并在 Java 代码执行时加载该库就可以了。”没错，是这个样子，为了辅助你这样做，Java 还提供了一个 `javah` 工具，该工具可以从一个 Java 文件自动生成相应的头文件，剩下的就是你根据这些头文件再实现具体的内部代码即可。比如，我们的 `Foo.java` 内容如下：

```

1 package com.haiii.android.client;
2
3 public class Foo {
4     native void foo1();
5     native int foo2(int a, String b);
6 }

```

使用 `javah` 命令：

```
javah -d ~/Desktop -jni com.haiii.android.client.Foo
```

该命令中需要注意，`-d` 选项的含义是指定输出路径，并且 `-d` 必须在 `-jni` 前面，`-jni` 选项的意思是产生 `jni` 头文件，后面的类名是 Class 文件所在的路径。也就是说，调用 `javah`，当前路径必须在该 Class 包名的根目录。生成的头文件默认名称为 `com_haiii_android_client_Foo.h`，其内容如下：

```

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

#ifndef _Included_com_haiii_android_client_Foo
#define _Included_com_haiii_android_client_Foo
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      com_haiii_android_client_Foo
 * Method:     foo1
 * Signature:  ()V
 */
JNIEXPORT void JNICALL Java_com_haiii_android_client_Foo_foo1
(JNIEnv *, jobject);

/*
 * Class:      com_haiii_android_client_Foo
 * Method:     foo2
 * Signature:  (ILjava/lang/String;)I
 */
JNIEXPORT jint JNICALL Java_com_haiii_android_client_Foo_foo2
(JNIEnv *, jobject, jint, jstring);

#ifdef __cplusplus
}
#endif
#endif

```

关于如何把 C 代码编译为动态库请参照后面第 18.6.9 章节。产生动态库以后,在 Java 代码调用 native 函数前,需要使用 `System.loadLibrary("lib_name")` 函数装载该库。

Java 代码中不能直接访问 C 中的变量,原因很简单,C 中的变量对于 Java 来讲都是私有的。如果想访问某个变量,那么就让 C 提供一个 get/set 类型的方法,以达到间接访问的目的。

## 2.2.2 C 访问 Java

这种情况似乎比较少,C 为什么还要访问 Java 呢?这个也容易理解,如果 C 中需要使用 Java 的某个变量而进行相应的处理,或者 C 中也想调用 Java 中的某个函数完成某些操作,那么 C 就要访问 Java。

由于 Java 中的函数在 native 引擎中并没有直接的函数指针,Java 函数只能由 Java 引擎去执行,而不是 C。所以,C 访问 Java 不能通过函数指针,而只能通过通用的参数接口,正如 Java 调用 C 一样。Java 把类名、函数名称、参数类型传递给 native 引擎,然后由 native 引擎处理 C 函数,同理,C 调用 Java 时,也需要把想要访问的类名、函数名称、参数传递给 Java 引擎。其步骤如下:

### ① 获取 Java 对象的类。

```
cls = env->GetObjectClass(jobject)
```

其中 env 为 Java 调用 C 函数时的第一个参数,这意味着 C 调用 Java 函数只能在 Java 调用 C 函数中进行,否则无法获取 env 变量。换句话说,对于 C 来讲,就是“你不惹我,我不惹你”。jobject 为第二个参数.cls 的类型是 jclass。

### ② 获取 Java 函数的 id 值。

```
jmethodId mid = env->GetMethodId(cls, "method_name", "([Ljava/lang/String;)V");
```

该方法中第二个参数为 Java 中的函数名称,第三个参数值得注意,它代表了 Java 函数的参数和返回值,参数在括弧之中,返回值在括弧之外。本例中,参数 `[Ljava/lang/String` 代表了 String 类型的参数,由于 String 本身是一个类,而不是 Java 的原子类型,所以前面加了包的名称,并用斜线分隔,最前面还要用一个中括弧进行标识,后面还要用分号隔离。返回值 V 代表 void。GetMethodId 中这种参数格式的定义如表 2-1 所示。

表 2-1 Java 和 native 数据类型对应表

Java 类型	native 类型	Java 类型	native 类型
boolean	Z	int	I
byte	B	long	L
char	C	Object	'L'+'packageName'+';'
double	D	short	S
float	F		

Java 提供了一个 `javap` 工具，`p` 的意思可能是参数 (Params)，该工具可以查看 Java 函数的输入、返回参数，用法如下：

```
javap -s com/haiiii/android/client/Foo
```

选项 `-s` 的含义是签名 (Signature)，上面所说的参数也叫做函数的签名，因为 Java 允许函数重载，所以不同的参数、返回值代表着不同的函数，或者说是不同函数签名。该工具的输出为：

```
Compiled from "Foo.java"
public class com.haiiii.android.client.Foo extends java.lang.Object{
    public com.haiiii.android.client.Foo();
        Signature: ()V
    native void foo1();
        Signature: ()V
    native int foo2(int, java.lang.String);
        Signature: (ILjava/lang/String;)I
}
```

③ 找到了函数后，就可以调用该函数了。

```
env->CallXXXMethod(jobject, mid, ret);
```

其中 `XXX` 代表了函数的返回值类型，具体包括 `Void`、`Object`、`Boolean`、`Byte`、`Char`、`Short`、`Int`、`Long`、`Float`、`Double`。在我看来，JNI 提供的这种按类型调用并不是必需的，只是为了某种灵活，因为该函数的第三个参数是保存返回值的变量，所以 JNI 内部完全可以根据 `ret` 的类型来选择把 Java 的执行结果进行格式转换。第二个参数 `mid` 即为第二步中所获得的函数 `id`。

通过以上三步，实现了 C 中调用 Java 函数的目标。还有一个问题，C 中如何访问 Java 中的变量呢？实现步骤如下：

① 与前面相同。

```
cls = env->GetObjectClass(jobject)
```

② 获取变量的 `id` 值。

```
jfieldId fid = env->GetFieldId(cls, "filed_name", "I");
```

参数 `filed_name` 为 Java 变量的名称，第三个参数为变量的类型，其格式与上面相同。

③ 获取变量值。

```
value = env->GetXXXField(env, jobject, fid)
```

该函数的参数与上面的显著不同，其中第一、第二个参数为原装 Java 访问 C 函数的前两个参数，该方法以返回值的方式获取变量值，而不是通过参数引用。

2.2.3 在 C 中使用持久对象

C 中的函数都是被调用的，本身内部无法保存持久对象，但是，有时候 C 中需要一个持久对象，以便下次调用时能够引用。

解决这个问题有一种技巧，这就是本节所讲的内容。

该技巧可如图 2-3 所示。

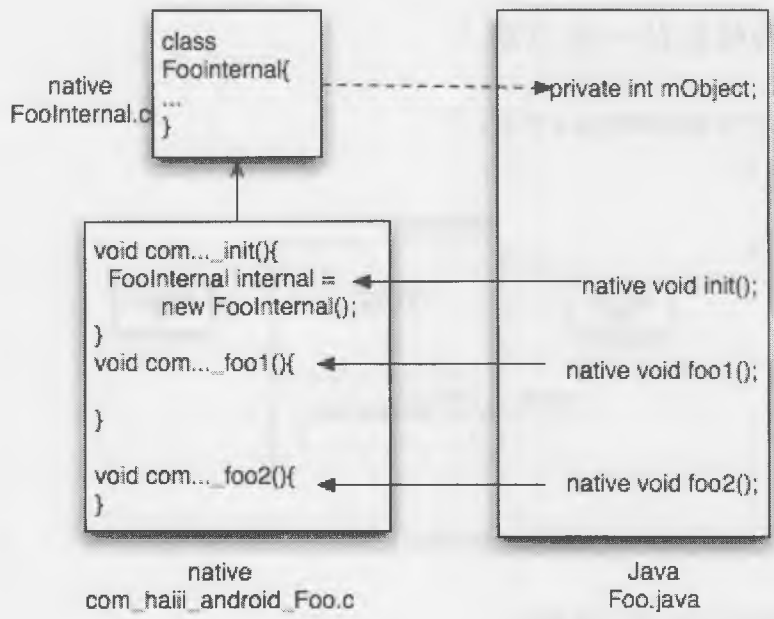


图 2-3 C 中使用持久对象代码结构示意图

对于 **Foo.java** 而言，其内部一般会包含一个 **init()** 函数，在 **native** 的实现中，**init()** 可能会创建一个本地对象 **FooInternal**。然而，**FooInternal** 对象却无法保存在 **native** 端，所以，它就把这个对象当成一个 **int** 值传递给 **Java** 端的 **mObject** 变量，在 **native** 端下次如果需要再引用 **FooInternal** 对象时，比如在 **foo1()** 或者 **foo2()** 函数中，就可以先读取 **mObject** 的值，然后再把该 **int** 值强制类型转换为一个 **FooInternal** 对象。

2.3 异步消息处理线程

对于普通的线程而言，执行完 **run()** 方法内的代码后线程就结束。而异步消息处理线程是指，线程启动后会进入一个无限循环体之中，每循环一次，从其内部的消息队列中取出一个消息，并回调相应的消息处理函数，执行完一个消息后则继续循环。如果消息队列为空，线程会暂停，直到消息队列中有新的消息。

异步消息处理线程的本质仍然是一个线程，只不过这种线程的执行代码被设置成如上所讲的逻辑而已，一般而言，当同时有以下两种需求时使用异步消息处理线程：

- 任务需要常驻。比如用于处理用户交互的任务。
- 任务需要根据外部传递的消息做不同的操作。

当有这两种需求时，就应该使用一个异步消息处理线程，异步线程在操作系统领域被广泛使用。本节就来介绍异步线程的一般实现思路，以及在 Android 中具体如何实现。

### 2.3.1 实现异步线程的一般思路

实现异步线程的一般思路如图 2-4 所示。

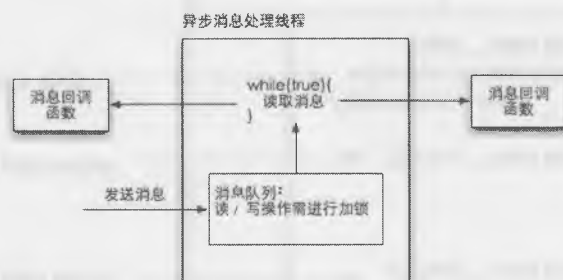


图 2-4 异步消息处理线程内部组成

实现异步线程要解决的问题具体包括：

- 每个异步线程内部包含一个消息队列（MessageQueue），队列中的消息一般采用排队机制，即先到达的消息会先得到处理。
- 线程的执行体中使用 `while(true)` 进行无限循环，循环体中从消息队列中取出消息，并根据消息的来源，回调相应的消息处理函数。
- 其他外部线程可以向本线程的消息队列中发送消息，消息队列内部的读/写操作必须进行加锁，即消息队列不能同时进行读/写操作。

不同的操作系统内部有不同的异步线程实现方式，但无论哪种方式，要解决的问题是相同的，下一小节将介绍在 Android 中如何解决这三个问题。

### 2.3.2 Android 中异步线程的实现方法

Android 中异步线程的实现原理如图 2-5 所示。



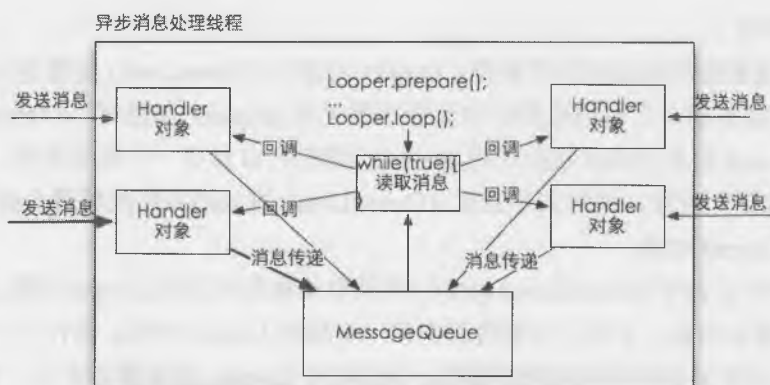


图 2-5 Android 中异步线程内部结构

在线程内部有一个或多个 Handler 对象，外部程序通过该 Handler 对象向线程发送异步消息，消息经由 Handler 传递到 MessageQueue 对象中。线程内部只能包含一个 MessageQueue 对象，线程主执行函数中从 MessageQueue 中读取消息，并回调 Handler 对象中的回调函数 handleMessage()。

下面就来分析线程内部的 Handler、MessageQueue、Looper 类的调用过程。

### 1. 线程局部存储 (Thread Local Storage)

程序员通过调用 Looper 类的静态方法 prepare() 为线程创建 MessageQueue 对象，该函数的代码如下：

```

71 public static final void prepare() {
72     if (sThreadLocal.get() != null) {
73         throw new RuntimeException("Only one Looper may be created per thread");
74     }
75     sThreadLocal.set(new Looper());
76 }

```

在该段代码中，变量 sThreadLocal 的类型是 ThreadLocal，该类的作用是提供“线程局部存储”，那么什么是线程局部存储 (TLS)？这个问题可以从变量作用域的角度来理解。

变量的常见作用域一般包括以下几种。

- 函数内部的变量。其作用域是该函数，即每次调用该函数时，该变量都会重新回到初始值。
- 类内部的变量。其作用域是该类所产生的对象，即只要该对象没有被销毁，则对象内部的变量值一直保持。
- 类内部的静态变量。其作用域是整个进程，即只要在该进程中，则该变量的值就一直保持，无论使用该类构造过多少个对象，该变量只有一个赋值，并一直保持。

对于类内部的静态变量而言，无论是从进程中哪个线程引用该变量，其值总是相同的，因为在编译器内部为静态变量分配了单独的内存空间。但有时我们却希望，当从同一个线程中引用该变量时，其值总是相同，而从不同的线程中引用该变量时，其值应该不同，即我们需要一种作用域为线程的变量定义，



这就是“线程局部存储”。

ThreadLocal 就是能够提供这种功能的类，Looper 内部的 sThreadLocal 变量是当该进程第一次调用 Looper.prepare() 时被复制的，之后该进程中的其他线程调用 prepare() 函数时，sThreadLocal 变量就已经被赋值了。sThreadLocal 对象内部会根据调用 prepare() 线程的 id 保存一个数据对象，这个数据对象就是所谓的“线程局部存储”对象，该对象是通过 sThreadLocal 的 set() 方法设置进去的，Looper 类中保存的这个对象是一个 Looper 对象。

prepare() 函数中首先调用 sThreadLocal.get() 函数获取该线程对应的 Looper 对象，如果该线程已经存在 Looper 对象，则提示出错，否则，为该线程创建一个新的 Looper 对象。为什么一个线程中只能有一个 Looper 对象呢？这仅仅是异步线程所需要的，因为每个 Looper 对象都会定义一个 MessageQueue 对象，一个异步线程中只能有一个消息队列，所以也就只能有一个 Looper 对象，这与“线程局部存储”本身没有什么关系，换句话说，这不是 ThreadLocal “惹的祸”，程序员可以使用 ThreadLocal 类来保存任何数据对象，这就是为什么 ThreadLocal 是一个模板类的原因。

了解了 ThreadLocal 的作用后，再重新从变量的作用域审视一下变量类型，可如表 2-2 所示。

表 2-2 不同作用域的变量类型

变量作用域类型	意义
函数成员变量	仅在函数内部有效
类成员变量	仅在对象内部有效
线程局部存储 (TLS) 变量	在本线程内的任何对象内保持一致
静态变量	在本进程内的任何对象内保持一致
跨进程通信 (IPC) 变量	一般使用 Binder 进行定义，在所有进程中保持一致

## 2. Looper

Looper 的作用有两点，第一是为调用该类中静态函数 prepare() 的线程创建一个消息队列；第二是提供静态函数 loop()，使调用该函数的线程进行无限循环，并从消息队列中读取消息。下面先看第一点，如何创建消息队列。

在 Looper 的静态函数 prepare() 中，会给线程局部存储变量中添加一个新的 Looper 对象，Looper 的构造函数中则会创建一个 MessageQueue 对象，如下代码所示：

```

162 private Looper() {
163     mQueue = new MessageQueue();
164     mRun = true;
165     mThread = Thread.currentThread();
166 }
```

对程序员来讲，当需要把一个线程变为异步消息处理线程时；应该在 Thread 类的 run() 函数中先调用 Looper.prepare() 为该线程创建一个 MessageQueue 对象，然后再调用 Looper.loop() 函数，使当前线程进入消息处理循环。loop() 函数的代码如下：

```

106 public static final void loop() {
107     Looper me = myLooper();
108     MessageQueue queue = me.mQueue;
109     while (true) {
110         Message msg = queue.next(); // might block
111         //if (!me.mRun) {
112         //    break;
113         //}
114         if (msg != null) {
115             if (msg.target == null) {
116                 // No target is a magic identifier for the quit
117                 return;
118             }
119             if (me.mLogging != null) me.mLogging.println(
120                 ">>>> Dispatching to " + msg.target + " "
121                 + msg.callback + ": " + msg.what
122             );
123             msg.target.dispatchMessage(msg);
124             if (me.mLogging != null) me.mLogging.println(
125                 "<<<< Finished to " + msg.target + " "
126                 + msg.callback);
127             msg.recycle();
128         }
129     }
130 }

```

该段代码的执行流程如下：

① 调用 `myLooper()` 函数返回当前线程的 `Looper` 对象，该函数内部仅仅通过调用 `sThreadLocal.get()` 方法返回当前线程 `id` 对应的 `Looper` 对象。

② 进入 `while(true)` 无限循环。

(1) 调用 `MessageQueue` 对象的 `next()` 函数取出队列中的消息。注意，如果当前队列为空，则当前线程会被挂起，也就是说，`next()` 函数内部会暂停当前线程。关于 `MessageQueue` 内部的消息读/写将在下面小节中介绍。

(2) 回调 `msg.target.dispatchMessage()` 函数，完成对该消息的处理，也就是说，消息的具体处理实际上是由程序指定的。`msg` 变量的类型是 `Message`，`msg.target` 的类型是 `Handler`。

(3) 每处理完该消息后，需要调用 `msg.recycle()` 回收该 `Message` 对象占用的系统资源。因为 `Message` 类内部使用了一个数据池保存 `Message` 对象，从而避免不停地创建和删除 `Message` 类对象，因此，每次处理完该消息后，需要将该 `Message` 对象表明为空闲状态，以便使该 `Message` 对象可以被重用。

### 3. MessageQueue

消息队列采用排队方式对消息进行处理，即先到的消息会先得到处理，但如果消息本身指定了被处理的时刻，则必须等到该时刻才能处理该消息。消息在 `MessageQueue` 中使用 `Message` 类表示，队列中的消息以链表的结构进行保存，`Message` 对象内部包含一个 `next` 变量，该变量指向下一个消息。

MessageQueue 中的两个主要函数是“取出消息”和“添加消息”，分别为函数 next() 和 enqueueMessage()。

首先来看取出消息的函数 next()，该函数的内部流程分三步。

① 调用 nativePollOnce(mPtr, int time)。这是一个 JNI 函数，其作用是从消息队列中取出一个消息。MessageQueue 类内部本身并没有保存消息队列，真正的消息队列数据保存在 JNI 中的 C 代码中，也就是说，在 C 环境中创建了一个 NativeMessageQueue 数据对象，这就是 nativePollOnce() 第一个参数的意义。它是一个 int 型变量，在 C 环境中，该变量将被强制转换为一个 NativeMessageQueue 对象。在 C 环境中，如果消息队列中没有消息，将导致当前线程被挂起 (wait)；如果消息队列中有消息，则 C 代码中将把该消息赋值给 Java 环境中的 mMessages 变量。

② 接下来这段代码被包含在 synchronized(this) 关键字中，this 被用做取消息和写消息的锁，在 enqueueMessage() 函数中也使用 synchronized(this) 进行代码同步。本步代码比较简单，仅仅是判断消息所指定的执行时间是否到了。如果到了，就返回该消息，并将 mMessages 变量置空；如果时间还没有到，则尝试读取下一个消息。

③ 如果 mMessages 为空，则说明 C 环境中的消息队列没有可执行的消息了，因此，执行 mPendingIdleHandlers 列表中的“空闲回调函数”。程序员可以向 MessageQueue 中注册一些“空闲回调函数”，从而当线程中没有消息可处理时去执行这些“空闲代码”。

下面再来看添加消息 enqueueMessage()，该函数内部分为两步。

① 将参数 msg 赋值给 mMessages。

② 调用 nativeWake(mPtr)。这是一个 JNI 函数，其内部会将 mMessages 消息添加到 C 环境中的消息队列中，并且如果消息线程正处于挂起 (wait) 状态，则唤醒该线程。

#### 4. Handler

尽管 MessageQueue 提供了直接读/写的函数接口，但对于应用程序员而言，一般不直接读/写消息队列。前面讲过，在 Looper.loop() 函数中，当取出消息后，会回调 msg.target 对象的 handleMessage() 函数，而 msg.target 的类型正是 Handler。

程序员一般使用 Handler 类向消息队列中发送消息，并重载 Handler 类的 handleMessage() 函数添加消息处理代码。

Handler 对象只能添加到有消息队列的线程中，否则会发生异常。以下代码来自于 Handler 类的构造函数中：

```

143     mLooper = Looper.myLooper();
144     if (mLooper == null) {
145         throw new RuntimeException(
146             "Can't create handler inside thread that has not call
147     }

```

因此，在构造 Handler 对象前，必须已经执行过 Looper.prepare()，但 prepare() 不能被执行两次。

创建 Handler 对象可以在执行 Looper.loop() 函数之前，也可以在执行之后。在以往的应用程序开发

中，程序员一般在 Activity 对象的初始化代码中添加 Handler 对象，事实上，在 Activity 对象被构造前，Activity 所在的线程已经执行了 `Looper.prepare()` 函数，关于这一点本书将详细介绍。

一个线程中可以包含多个 Handler 对象。在 `Looper.loop()` 函数中，不同的 Message 对应不同的 Handler 对象，从而回调不同的 `handleMessage()` 函数。

异步消息处理线程在 Framework 中被广泛使用，除了用于多线程消息传递外，它还和跨进程调用（IPC）一起被使用，用于实现异步跨进程调用。读者以后只要看到 Handler 对象，就应该想到异步消息处理线程。



## 第 3 章 Android 源码下载及开发环境配置

Android 源码的下载方法及系统的配置在 Android 的官方网站上有详细介绍，网址为：<http://source.android.com/source/index.html>。因此，本书不准备介绍该部分内容，但关于下载及系统配置却有一些注意事项，这些在官方网站上没有，本节仅介绍这些内容。

### 3.1 Mac 系统的配置

Mac 系统是 Unix-like 的系统，可完全用于 Android Framework 及应用程序的开发，本书中的大部分内容都是在 Mac 系统下验证的。笔者曾多次安装 Mac 系统，原因是各种不兼容导致的，所以本节介绍一些安装和配置 Mac 系统的经验。

#### 3.1.1 硬盘格式的配置

Mac 系统出厂时默认的硬盘格式不区分大小写（Case-insensitive），但 Android Framework 却必须区分大小写。因此，笔者曾尝试重新安装系统并使用“区分模式”。

安装后，一切正常，直到笔者想使用 Photoshop 这个著名的图像处理软件时才发现，这个软件竟然不能运行在区分大小写的系统上，这怎么办呢？笔者最终采用如下解决方法（建议大家采取此方法）。

- ① 重新安装系统，选择不区分大小写的硬盘格式。
- ② 系统安装好后，使用 Applications/Utilities/Disk Utility 工具，创建一个区分大小写的磁盘映像文

件，名称可以为 Android，大小为 20GB，如图 3-1 所示。

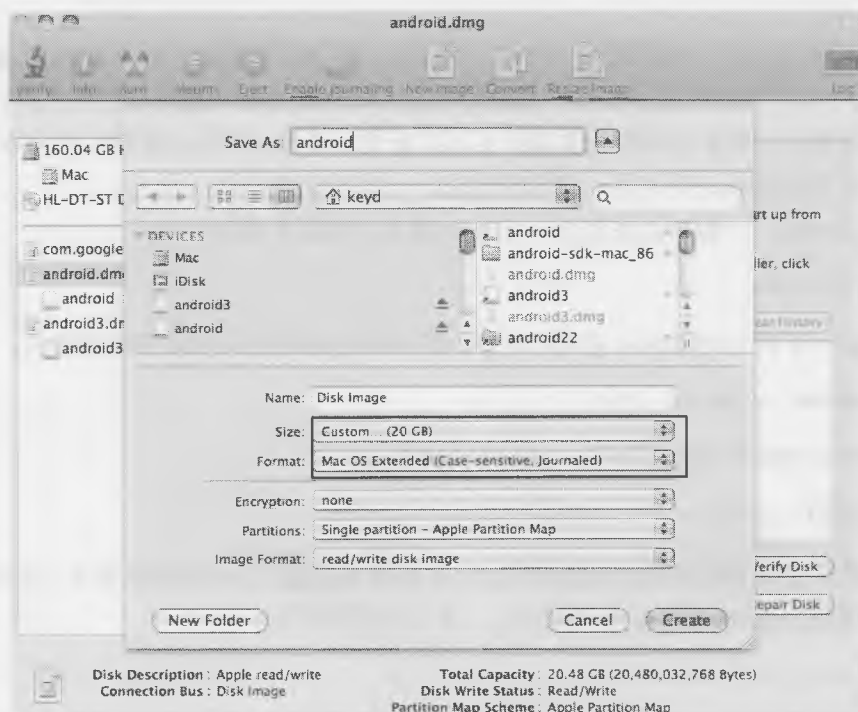


图 3-1 在 Mac 系统中创建一个文件系统镜像

③ 创建好的 android.dmg 是一个磁盘映像文件系统，可以把将来下载的 Android 源码放到该文件系统上。安装 dmg 的方法是使用 hdiutil 命令，为了便于安装，可以将安装命令写成一段脚本，并保存到用户目录中的 .profile 中，如以下代码所示：

```
function mountAndroid { hdiutil attach ~/android.dmg -mountpoint /Users/keyd/android; }
```

该段代码定义了一个脚本函数，其内部主要是调用 hdiutil 命令，参数中 -mountpoint 选项指定 dmg 文件系统被挂载到哪个路径，此处为 /Users/keyd/android，keyd 是笔者的用户名，android 是指一个普通的文件夹。

从而，每次系统启动后，当我们需要挂载 android.dmg 时，只需要开一个 Terminal，然后运行 mountAndroid 命令即可，如以下代码所示：

```
keyd-2:~ keyd$ mountAndroid
/dev/disk2      Apple_partition_scheme
/dev/disk2s1    Apple_partition_map
/dev/disk2s2    Apple_HFSX                      /Users/keyd/android
```

然后，/Users/keyd/android 目录中的文件就是 android.dmg 文件系统，这是一个区分大小写的文件系统。

### 3.1.2 port 的用法

port 是 Mac 系统中用于下载软件的一个客户端命令，其作用类似于 Ubuntu 中的 apt-get 命令，在 Android 官方网站中会提到使用 port 安装一些编译 Framework 所需的库程序。在使用 port 命令的过程中，有时会出现安装异常，导致本地的库无法和服务器重新进行同步，因此就需要先清理本地的相关文件，才能再次下载，本节介绍 port 命令的一些常见用法。

当首次运行 port 前，一般可以先更新一下 port 自身，如以下代码所示：

```
$sudo port selfupdate
```

然后可以更新一下所有使用 port 安装的程序，这个不是必需的，如以下代码所示：

```
$sudo port upgrade outdated
```

接下来，就可以安装想要的程序了，如以下代码所示：

```
$sudo port install <portname>
```

有时，安装过程会由于网络原因而意外终止，并且当重新运行 port install 命令后会出现一些莫名的错误。此时，可以使用 port clean 命令清除残存，如以下代码所示：

```
$sudo port clean --all <portname>
```

## 3.2 在 Linux 中配置 USB 连接

通过 USB 连接 Android 设备后，就可以使用 adb 命令和设备进行交互了。这在 Mac 系统上不需要任何配置，但在 Linux 中却需要额外的配置，原因是 Linux 系统中默认并没有给该 USB 设备赋予权限，配置的方法如下：

- ① 连接 USB 设备后，使用 lsusb 命令查看该设备的 vendorId 和 productId。
- ② 在 /etc/udev/rules.d/ 目录下新建一个规则文件 99-android.rules，该文件在系统启动后会被自动加载，文件内容中包含上一步获得的 vendorId 和 productId，如以下代码所示：

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="18d1", ATTRS{idProduct}=="4e12",
MODE="0666", OWNER="keyd"
```

其中 OWNER 是指当前的 Linux 用户名称。

- ③ 重启 usb 服务，命令如下：

```
sudo /etc/init.d/udev restart
```

## 3.3 在 Eclipse 中调试 Framework

本节主要介绍如何来调试 Framework 的代码。

### 3.3.1 一段防止下载异常的脚本

在调试代码之前，首先自然是需要下载 Android 源码，这在官方网站上已有介绍，不再赘述。但在实际下载的过程中，由于 Android 源码大小在 2GB 左右，往往需要较长时间的下载，中间有时会出现网络异常。因此，本节给出一段防止下载异常的下载脚本，该脚本的想法很简单，就是尝试捕获 repo sync 命令的执行结果，如果结果不正常，则重新执行该命令，脚本整体代码如下：

```
#!/bin/bash
#FileName get-android.sh
PATH=./bin:$PATH
repo init -u git://android.git.kernel.org/platform/manifest.git -b master
repo sync
while [ $? = 1 ]; do
echo "=====sync failed, re-sync again ====="
sleep 3
repo sync
done
```

### 3.3.2 调试 Framework 中的代码

要调试 Framework 源码，则必须将源码导入到 Eclipse 中，方法如下：

① 将/root/to/android/development/ide/eclipse 目录下的.classpath 文件复制到 android 根目录下，该文件记录了项目中默认包含的源码列表。

② 在 Eclipse 中新建一个 Java 工程，工程名可叫做 android2，路径指到 android 根目录下。

这样，Eclipse 中有了一个名称为 android2 的 Java 工程，这个工程仅仅是用于浏览源码和调试源码的。尽管 Eclipse 后台会自动调用 Java 编译器对该项目进行编译，但其编译结果实际上无任何意义，也不产生任何可执行的程序。

有时读者会发现，有些 Android 源码默认并没有被包含到 android2 项目中，原因是.classpath 中没有列出该源码。比如，packages/inputmethods/PinyinIME 子项目在以上新建的 android2 项目中就看不到。解决的办法就是修改.classpath 文件，比如此处就可以添加以下代码：

```
<classpathentry kind="src" path="packages//inputmethods/PinyinIME/src"/>
```

在以往的调试经验中，读者习惯于使用 Eclipse 的菜单命令 run/debug 来调试一个独立的 Android 应用程序，而 Framework 并不是一个应用程序，那么如何调试呢？要深入理解这个问题，还需要阅读本书后续章节的介绍，这里简单说明以下几点。

- Framework 虽然是 Android 的“内核”，但其内部也包含一个应用程序，其名称为 system\_process，其对应的 TCP 端口地址一般为 8600，因此也可以对该程序进行调试。



- Framework 中包含的 SDK 源码在程序空间中只有一份副本，但是每个应用程序都会调用这部分程序，调用时，该代码是在应用程序所在的用户进程中运行的。

基于以上两点，理论上可以对普通应用程序进行调试，并将上面的 android2 项目中的源码附属到调试程序中，那么就可以单步调试到 Framework 的源码了。另一方面，也可以单独对 system\_process 进行调试。

下面就以三个例子说明这种调试方法。

第一例子，以普通的应用程序为例，使用菜单命令 run/debug，当调试到需要调用 Framework 代码的地方，Eclipse 会提示找不到所需的源码，如图 3-2 所示。

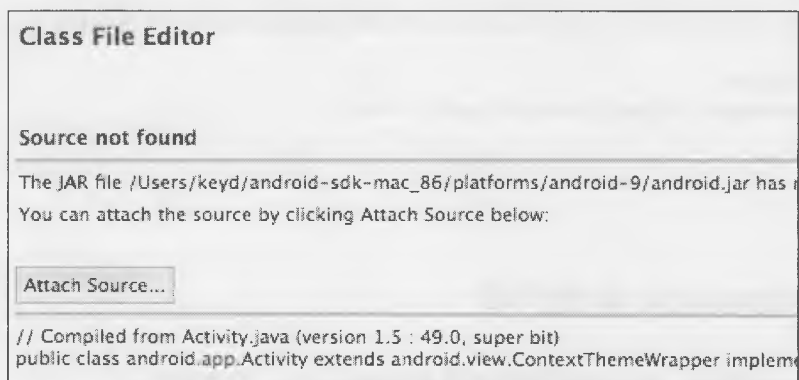


图 3-2 为项目添加 SDK 源码

此时，可以单击“Attach Source...”按钮，并指向到上面的 android2 项目，其作用就是将 android2 中的源文件作为 SDK 中 android.jar 的源码，从而就可以单步运行到相应的 Framework 源码中。

第二个例子，以系统应用程序为例，比如 packages/apps/Contacts 项目。该项目中引用了 Framework 中的一些特别类，这些类在默认情况下并没有被包含到 SDK 中的 android.jar 包中，因此，Eclipse 下无法直接编译该项目并生成 Contacts.apk。要生成 Contacts.apk 可以有两种方法，第一种是把 Contacts 项目中所需的特别类文件打包成一个 Jar 包，并把该 Jar 包包含到 Contacts 项目中，从而就可以直接在 Eclipse 中编译出 Contacts.apk，另一种方法是直接在 Terminal 中使用 \$make Contacts 命令编译出 Contacts.apk，一般多采用后一种方法。

使用第二种方法时，由于 Eclipse 无法直接编译出 Contacts.apk，因此也就无法直接使用菜单命令 run/debug 来启动调试，因此可使用以下方法来启动调试。

① 在 Android 设备上运行 Contacts 程序，运行后可以在 ddms 界面上看到 Contacts 程序对应的 TCP 端口。本例中 Contacts 对应的进程名称为 android.process.acore，对应的 TCP 端口为 8602，如图 3-3 所示。



Name	Online	master [AOSP, debug]
emulator-5554		
system_process	66	8600
com.android.inputmethod.latin	121	8601
com.android.systemui	132	8604
com.android.launcher	134	8605
com.android.smpush	205	8607
com.android.phone	264	8603
com.cooliris.media	366	8613
com.android.defcontainer	397	8606
com.android.voicedialer	407	8611
com.svox.pico	419	8610
com.android.quicksearchbox	427	8612
android.process.acore	899	8602 / 8700

图 3-3 查看应用进程对应的 TCP 端口

- ② 调用菜单命令“run/Debug Configurations...”，并进行如图 3-4 所示的配置。

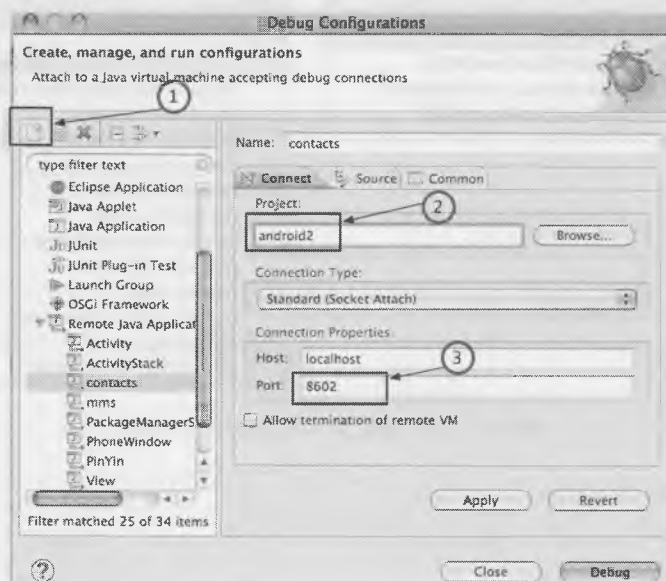


图 3-4 添加一个 Remote Java Application 的调试配置

该配置中包含三点，首先进入 Remote Java Application 列表，然后单击添加按钮新建一个配置。在 Project 编辑框中选择前面的 android2 项目，因为 Contacts 的源码被包含在该项目中，然后在 Port 编辑框中指定 Contacts 对应的 TCP 端口。最后单击“Debug”按钮开始调试，剩下的调试过程就跟调试普通程序完全相同，可以直接在 Contacts 的源码上设置断点，单击“Debug”按钮后并运行到该断点时就会暂停。

下面再来看最后一个例子，本例是直接调试 system\_process 进程。为什么要调试 system\_process 进

程呢？因为 Android 系统的架构是一种跨进程合作机制，系统将窗口管理、Activity 管理等操作全部放在 `system_process` 进程中，客户端进程向服务进程发起服务命令，然后服务进程进行相关的操作，因此，当需要调试服务端的 Framework 时就可以调试 `system_process` 进程。调试的过程类似于第二个例子。

① 寻找 `system_process` 的 TCP 端口号。

② 在 `run/Debug Configuration...` 菜单弹出的对话框中添加一个 Remote Java Application，并使用第一步得到的 TCP 端口号。

③ 在 `system_process` 所使用的源码中设置断点，比如 `WindowManagerService.java`。由于 `system_process` 进程也会和普通进程一样使用 SDK 中的代码，因此，如果在 SDK 代码中设置断点，调试时请注意该调用是来自 `system_process` 进程还是普通应用进程。

以上三个例子介绍了调试 Framework 的具体方法，Eclipse 中还支持同时调试多个进程，具体方法就是添加多个 Remote Java Application。当需要同时调试 Contacts 应用进程和服务进程之间的调用关系时，就可以同时启动 Contacts 的调试和 `system_process` 的调试，如图 3-5 所示。

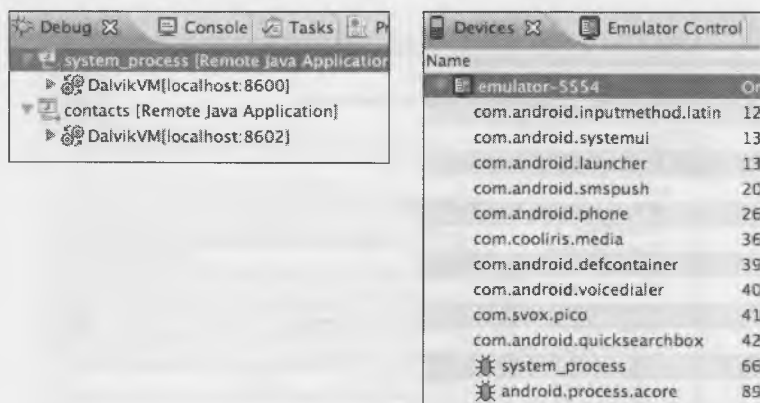


图 3-5 同时调试多个项目的界面示意图



## 第 4 章 使用 git

git 是用于代码管理的工具，代码管理主要完成两个功能，一个是备份，另一个是代码合并（merge）与分离（difference）。git 的作者是鼎鼎有名的 Linux 作者 Linus Torvalds。

在开发 git 前，Linus 使用名称为 BitKeeper 的代码管理工具对 Linux 内核代码进行管理，可是在 2005 年，Linus 和 BitKeeper 的关系崩了，BitKeeper 宣称不再让 Linux 项目免费使用，同时，Linus 等人也觉得 BitKeeper 存在诸多不足，于是便决定开发一个新的代码管理工具，这个就是 git。git 在设计之初要求具备如下特性。

- 高速。
- 实现简单。
- 超强支持非线性开发。
- 完全的分布式开发。
- 支持超大项目（比如 Linux），包括速度和所使用的磁盘空间。

目前，Linux Kernel 所使用的管理工具正是 git，Google 的 Android 也是采用 git 进行代码管理的，这也是我们这里介绍 git 的原因。

git 有以下两个特点。

- 每个开发机都是一个仓库（Repository），代码的提交可以先提交到本地仓库，然后再从本地仓库推送到另外一个远程（Remote）仓库，这也叫做分布式开发。这不像 SVN，只有服务器上有一个仓库，开发机仅仅是一个副本，开发机必须和服务器有网络连接时才能提交。
- 每次代码提交（Commit）都会生成一个快照（Snapshot），这个快照是所有被修改过文件的一个副本，而不是增量（Delta）。这个特点决定了 git 的操作速度比较快，因为每次分支间的切换不需要根据增量重新计算出最终文件，而是直接从快照中提取出文件。

正如本书宗旨，凡是笔者知道有更好的关于某个主题的介绍，就不会另起炉灶，写一些蹩脚内容浪费大家时间，而是会介绍一些笔者心得及一些关键备忘以便查阅。关于 git 的详细介绍请参考网站：<http://progit.org/book/>，这本书内容比较简洁明了。

### 4.1 安装 git

git 同时支持 Window、Linux、Mac，实际上本书建议使用 Mac 或者 Linux，因此，以下仅介绍这两者的安装。

git 是一个开源免费软件，网址是 <http://git-scm.com/>，安装 git 可以通过下载源码安装，但是笔者还是喜欢直接安装二进制，因为简单而且够用。

Linux 下安装命令如下，其中 git-core 是 git 的安装包，其他的是 git 所依赖的安装包。

```
$apt-get install libcurl4-gnutls-dev libexpat1-dev gettext libz-dev git-core
```

Mac 下的安装命令如下：

```
sudo port install git-core +svn +doc +bash_completion +gitweb
```

### 4.2 git 仓库管理

git 是通过仓库来保存版本管理所需的信息的，本节首先介绍仓库的组成，然后再介绍创建仓库及管理仓库中的分支。

#### 4.2.1 仓库的组成

仓库，英文为 Repository，这也是 Google 的 repo 脚本名称的含义。在日常代码管理中，与用户直接相关的概念是分支（branch）和提交（commit）。提交也可以称之为“节点”，提交的含义是提交当前所作的修改，但是所作的修改并不是指与前一次提交的差异（delta），而是修改后的整个内容，这点也是 git 与 svn 的主要差别之一。仓库就是存放这些分支和提交的地方。

git 系统包含三种对象（object），分别是提交（commit）、树（tree），以及原文（blob），每个对象都以文件的方式保存，文件的名称为该对象的 sha-1 标识。三者的关系如图 4-1 所示。

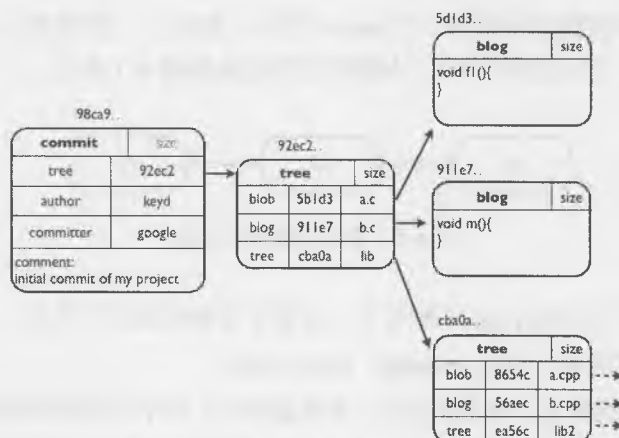


图 4-1 commit、tree 和 blob 之间的关系

每一个 commit 中包含了所修改的文件列表，用树（tree）表示，同时包含了作者（author）、提交者（committer）、备注（comment）等信息。其中 tree 是一个对象地址引用，指向了包含该 tree 信息的对象。author 和 committer 的区别在于，author 是编写代码的人，committer 是调用 git commit 提交代码的人。

tree 中列出本次提交具体包含的文件列表，包括源文件名称，以及该文件的对象地址引用。每个文件列表对应一个 blob 对象，除了 blob 对象，tree 中内部也可以包含另一个 tree 对象，这点有点类似于文件夹。tree 内部嵌套一个子目录的好处是减少相同文件的多份副本，至于什么时候会创建一个子目录，这完全是 git 系统内部通过特定的检测方法自动完成的。

blob 则是具体的原文，比如，对于 C 程序文件而言，就是源码本身。

每次用户提交时，git 系统就生成一个快照（Snapshot），该快照即为图 4-2 所包含的内容。

当多次提交后，提交记录如图 4-2 所示。

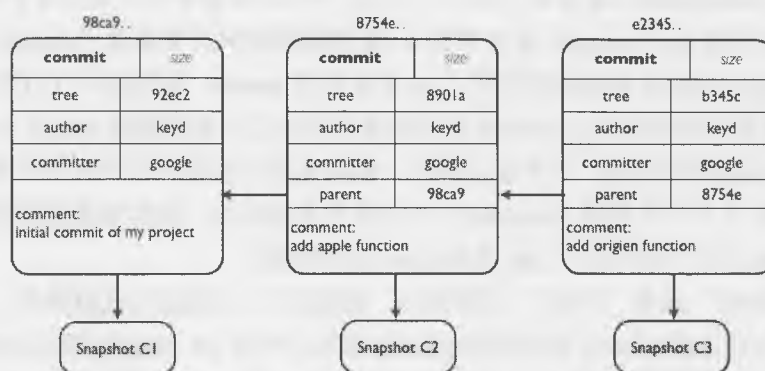


图 4-2 commit 和 Snapshot 之间的关系

在如图 4-2 所示中, 每次提交中包含一个 **parent** 字段, 指向上一次提交, 从而多次提交生成一个线性的链条。为了描述方便, 在后续章节中, 本图可以简化为如图 4-3 所示。

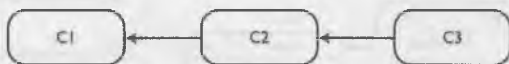


图 4-3 简化描述提交记录

git 仓库存放在当前工作目录的 **.git** 文件夹下, 主要子目录包括以下几项。

- **.git/objects**: 所有的对象, 包括 **commit**、**tree**、**blob**。
- **.git/refs/heads**: 所有的分支, 分支仅仅是一个地址引用, 每个引用指向该分支中最后一次提交记录。
- **.git/refs/tags**: 所有的标签 (**tags**), 可以为某次提交打上标签, 方便以后查看。
- **.git/logs/HEAD**: **HEAD** 的变化历史。
- **.git/logs/refs/heads**: 除 **master** 分支外其他分支的 **HEAD** 变化历史。

## 4.2.2 创建仓库

创建仓库分两种方式, 一种是对本地的某个文件夹进行 **git** 管理, 另一种是从某个远程仓库中克隆。首先来看第一种, 如以下命令所示:

```

$cd ~/gongsipk/mydir
$echo "this is file A" >> a.c
$echo "this is file B" >> b.c
$git init
$git add *
$git status
$git commit -a -m "first created"
  
```

此时, 已经在 **~/gongsipk/mydir** 目录下建立了仓库, **\*** 表示该仓库包含该目录下所有文件。**git status** 命令用于查看当前工作状态。**commit** 命令中的 **-a** 选项标识提交所有修改 (**modify**) 过的文件及删除 (**delete**) 的文件, 但并不包含新添加的文件。**-m** 的意思是 **memo**, 即添加一个注释。

注意, 如果不使用 **-a** 选项的话, **commit** 命令提交的是指已经被添加到 **staged area** 区域的修改。在 **git** 内部系统中, 一共有三个区域, 一个是工作区, 也就是用户能看到文件夹下的文件改动; 另一个是 **staged area**, 是指 **git** 认为下次提交 (**commit**) 时应该包含的内容; 最后就是已经提交的区域, 该区域是指工作目录下的 **.git** 目录下的内容, **.git** 目录由 **git** 自动创建。

**stage** 本意为“舞台”或者“阶段”, 直译则为“阶段区”。在我国古代有街亭, 用于走累了歇一会儿, **stage** 的意思即为此。**stage area** 的内容由 **git add** 添加, 每次 **git commit** 后又会清空 **stage area** 的内容。因此, 当使用 **git add** 后, 如果在工作区对文件进行了第二次修改而没有再次调用 **git add**, 那么直接调用 **git commit** 后不会提交第二次所作的修改, 除非使用 **-a** 选项。

此时，可以用 `git branch` 查看当前已有的分支。

```
$ git branch
* master
```

以上信息显示，当前的分支为 `master`，`*`代表当前工作区正在相应 `branch` 上工作。

接下来介绍克隆远程仓库。克隆的命令如下：

```
$git clone git://url local_dir
```

该命令从 `git://url` 地址中克隆一份远程仓库到本地 `local_dir` 目录下，同时会把仓库中的当前分支对应的文件 `checkout` 出来放到工作区，即所在文件目录下。比如，Android 的源码 `git` 仓库列表见 <http://android.git.kernel.org/>。可以尝试克隆 `Contacts` 应用对应的仓库，命令如下：

```
$git clone git://android.git.kernel.org/platform/packages/apps/Contacts.git
```

### 4.2.3 分支管理

分支，英文 `branch`。当修改完代码后，一般要先提交（`commit`），每次提交后，`git` 内部都会生成一个快照（`snapshot`），该快照会保存所有修改过的文件，并生成一个标识，用以标识本次提交，这个标识是一个 40 位的 SHA-1 字符串。当多次提交后，用户可以根据之前的提交标识恢复到当次提交后的内容，从而便于用户进行快速代码内容切换。每次提交都有且仅有一个父提交，因为每次提交必须基于之前的提交，这些连续的提交就构成了一个分支。分支也用一个 40 位的 SHA-1 标识，该标识指向分支中最后一次提交，同时为了便于操作，每个分支都有一个名称，比如 `Master`、`development` 等，如图 4-4 所示。

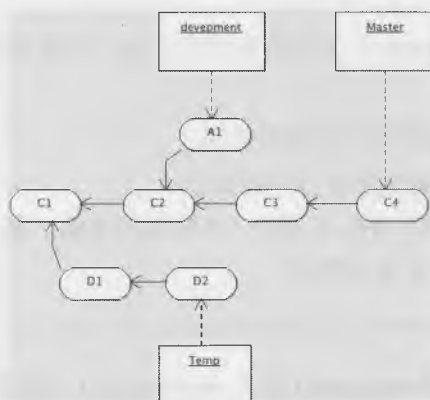


图 4-4 分支示意图

该图中有三个分支，分别是 `development`、`Master` 以及 `Temp`，其中 A、C、D 代表不同的单次提交。多个分支的概念实际上是基于某次提交创建的两个不同的记录，每个分支上的提交都是基于该分支的最



后一次提交。

创建分支可以基于某个已有分支，也可以基于某次提交。基于前面的~/gongsipk/mygit 目录下，创建一个如图 4-1 所示的分支结构，首先创建 temp 分支，如以下代码所示：

```
$git branch -b temp
$echo "add D1" >> a.c
$git add a.c
$git commit -m "this is D1"
$echo "add D2" >> a.c
$git add a.c
$git commit -m "this is D2"
```

接下来可以修改 master 分支：

```
$git checkout master
$echo "add C2" >> a.c
$git add a.c
$git commit -m "this is C2"
$echo "add C3" >> a.c
$git add a.c
$git commit -m "this is C3"
$echo "add C4" >> a.c
$git add a.c
$git commit -m "this is C4"
```

最后，创建 development 分支。此时，由于 development 分支并不能基于任何已有的 branch，而只能基于 master 分支的 C2，因此，需要首先知道 C2 的标识，如以下命令所示：

```
$git checkout master
$git log
```

git log 将会列出当前分支（master）上的所有提交记录，也可以使用以下选项获得不同的 log 输出效果。

- git log -p，同时列出本次提交前后具体的差别。
- git log --pretty=oneline，每条记录仅占一行显示。

通过 git log 可以获得 C2 对应的 SHA-1 标识（在笔者的电脑上是 39c2...），接下来便可以根据这个标识创建 development 分支，如以下命令所示。

```
git branch development 39c2a2399a7e2721a4c3490c57ba0c51fe68a417
```

然后就可以使用 git checkout development 进入 development 分支做相应的修改，以下略。

创建完分支后，如果想要丢弃某个分支，可以使用以下命令删除分支。

```
$git branch -d development
```

删除分支仅仅是删除该分支的名称而已，即在使用 git branch 命令时看不到该分支，而不会删除该分支所包含的任何 commit 记录。这也就是为什么在 git 系统中，只要是提交过的内容就不会丢失。然

而，删除分支后，原分支所引用的提交记录就有可能成为“无引用”的状态，比如以上的 A1 提交。git 系统提供了一个清除无引用提交的命令，此处暂且不述。

### 4.3 git merge 用法

merge 为“合并”之意，merge 的对象为两个或多个不同的分支。同一个分支中的多个提交之间不存在 merge 之说。

举例如下：

```
$git checkout master
$git merge development
```

merge 的语义是 merge from，即从指定的分支合并到当前的分支，以上代码的含义就是把 development 分支中的改动合并到 master 分支中。如果合并中出现冲突，系统会自动以一种特殊的格式将冲突写入源文件，一般的冲突格式如下：

```
<<<<<< HEAD
this is C5
=====
this is A_2
>>>>>> development
```

冲突格式的开头是 7 个“小于号”加“HEAD”字符串，结尾是 7 个“大于号”加分支名称，中间的 7 个“等于号”是冲突的分隔符，前面是当前分支中的内容，后面是指定分支的内容，解决冲突的方法只能是手工修改以上内容。git 框架也提供了自动调用外部 merge 工具解决冲突，但这些外部工具仅是查看而已，至于到底要合并成什么样子，还得取决于用户想使用哪部分代码。

如果合并时没有冲突，则系统会自动将合并后的内容提交，如图 4-5 所示。

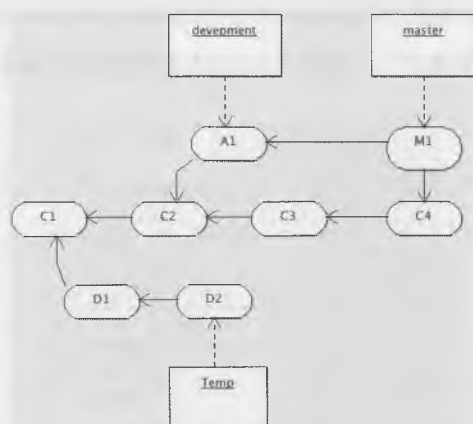


图 4-5 merge 示意图

其中 M1 记录是 merge 后无冲突或者冲突解决后系统自动提交的内容, 此时, master 分支就指向该记录, 而 development 分支没有任何变化。

git merge 常见的选项包括以下几个。

- --commit, --no-commit: 默认为 commit, 即 merge 后自动进行 commit, 如果使用 --no-commit, 则 merge 后仅仅改变当前分支中的工作区的文件内容。比如:

```
$git merge --no-commit development
```

- -m: merge 后自动提交时所使用的注释, 比如:

```
$git merge -m "merge from development" development
```

merge 发生冲突的条件是同一个地方包含不同的内容, 所谓同一个地方是指在源文件中的相同行相同列。举例如下, 假设 master 分支中的 a.c 源文件第 8 行包含以下代码。

```
public void f1(){
    // master branch
}
```

在 developer 分支中的 a.c 源文件第 8 行包含以下代码。

```
public void f1(){
    // developer branch
}
```

那么, 当合并 master 分支和 developer 分支时就会出现冲突, 因为 git 不能确切知道到底是使用第一段代码还是第二段代码。

当出现冲突后, 可以执行 git mergetool 启动不同 merge 工具辅助进行 merge 操作。详情参见第 4.9 节关于 mergetool 的配置。

当手工 merge 成功后, 需要再次调用 git commit 将所作的修改进行提交, merge 才能生效。

## 4.4 git rebase 用法

顾名思义, rebase 意为“改变基点”, 假设当前的分支状况如图 4-6 所示。

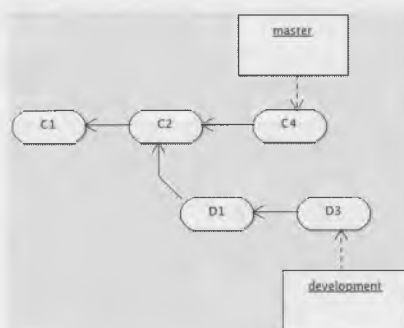


图 4-6 rebase 之前

假设 master 是主流分支, development 是某个测试分支, 当测试完局部功能后接到 master 分支 owner 的通知, 说 master 已经有了新的改动, 所有的局部功能测试必须基于新的 master 状态, 此时对于 development 分支来讲, 就需要对自己当前的分支“改变基点”。如以下代码所示:

```
$git checkout development
$git rebase master
```

rebase 后如果有冲突, 其解决方法和 merge 相同, 冲突解决后, 调用 git commit 进行提交即可。rebase 成功后如图 4-7 所示。

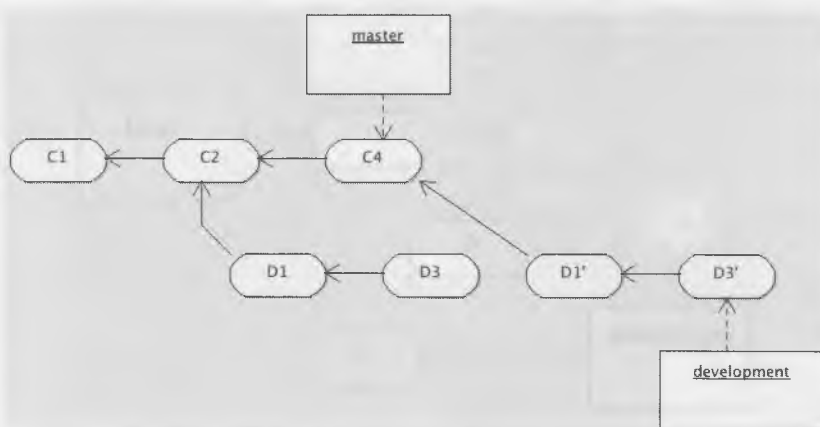


图 4-7 rebase 后的状态图

rebase 后, 之前的提交并没有被删除, 而会依然存在, 并且自动新添加了两个提交, 分别为 D1' 和 D3'。新添加的提交内容 D1' 是 D1 和 C2 的差别, D3' 是 D3 和 C2 的差别。rebase 的具体过程是找到两个分支的共同父节点, 本例即为 C2, 找到父节点后, 逐个找到当前分支中每个节点与父节点的差别, 并保存到一个临时文件中, 然后把这个临时文件应用到目标基点上。

理论上讲, development 最终的状态 D3, 也可以通过合并 D3 和 C4 实现, 然后, merge 和 rebase 还是有一些差别。

- rebase 和 merge 所产生的提交记录不同, 并且 rebase 后会产生“无引用”的提交, 比如 D3 和 D1。
- rebase 的理念是改变“局部小功能”的基点, 基点对应的是“主体大功能”; 而 merge 的理念是相同重量级功能的合并。
- rebase 一般是“局部小功能”发起的, 而 merge 则是“主体大功能”或者“相同重量级”发起的。
- rebase 一般是临时性的合并, 比如本例中, development 分支可能还需要继续针对该功能进行开发, 当前只是为了和基础代码保持一致而已; 而 merge 一般是阶段性的合并, 合并后有可能不再针对该功能单独开发, 并且合并后会删除临时性的分支。

rebase 除了改变当前分支的基点外, 还可以改变当前分支中某段修改的基点。这句话的含义是当开

发某个局部功能时，假设局部功能由三个小功能组成：f1、f2、f3，当前测试表明 f2 和 f3 是比较稳定的，而 f1 有些不稳定。可是 master 分支要求尽快提交已经成熟的功能，此时就可以仅仅针对 f2 和 f3 改变基点，这真是太灵活了，在执行 rebase 之前的分支状态如图 4-8 所示。

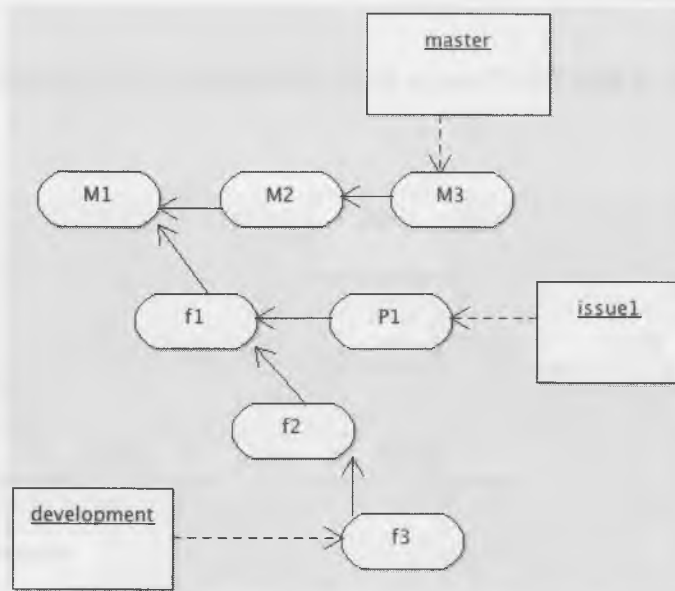


图 4-8 rebase——onto 之前

执行以下命令后的分支状态如图 4-9 所示。

```
$rebase --onto master development issue1
```

--onto 选项的含义是把 development 分支中与 issue1 分支“共父”的那一段，rebase 到 master 分支上。

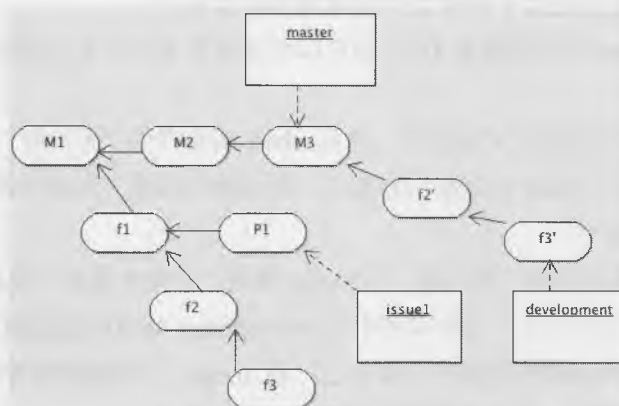


图 4-9 rebase——onto 之后

## 4.5 git cherry-pick 用法

cherry-pick 在英语中的含义是，寻找最好的樱桃，一般多用于投资领域中，即意为在众多的投资对象中挑选最优的一个。

cherry-pick 和 merge、rebase 有相似的地方，所实现的功能也能通过迂回使用 merge 或 rebase 完成，但 cherry-pick 所包含的意图能让开发人员的思路更清晰。假设当前仓库的分支状态如图 4-10 所示。

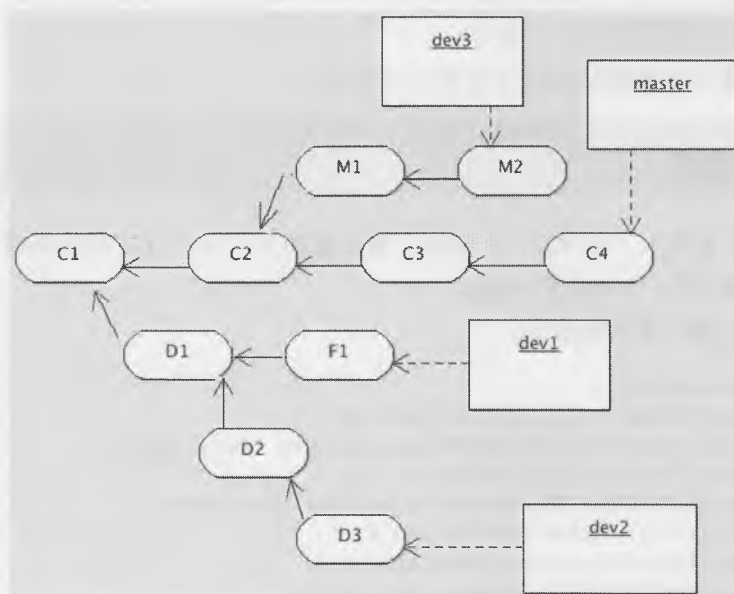


图 4-10 cherry-pick 命令执行前的仓库分支状态

当前的分支为 master，此时可以使用 cherry-pick，把 dev1、dev2、dev3 三个分支中的任何一个提交应用到 master 分支，比如 M1、M2、F1、D2、D3 等。如以下代码所示：

```
$git cherry-pick sha-1
```

sha-1 是目标节点的标识，可以通过 git log、git reflog、git fsck 等命令获取该标识。cherry-pick 命令是 git 系统特有的，因为 git 系统的每次提交都是一个快照（snapshot），该快照保存了每个文件的全部内容，而不是增量（delta）。对于一般基于增量的版本管理系统，比如 svn，每次提交保存的仅仅是增量，那么如果要把某次提交应用到当前分支中，必须经过多次增量运算才能完成。比如在如图 4-10 所示的状态下，要把 F1 应用到 master，必须经过 F1-D1-C1 获得一个临时文件，然后再通过 C4-C3-C2-C1 获得另外一个临时文件，然后再把两个文件合并。而在 git 系统中，可以直接合并 F1 和 C4，速度是显而易见的，并且对用户来讲，理念非常清晰——“把某次提交后的状态应用到当前分支”，这就是 cherry-pick。

前面讲过, cherry-pick 可以通过迂回调用 git merge 或 git rebase 代替。比如, 在如图 4-10 所示中, 要把 D2 应用到 master 分支中, 可以使用以下两个方法代替。

方法一, 首先在 D2 处新建一个分支, 然后调用 git merge 将新建的分支合并到 master 即可。

方法二, 在 D2 处新建一个分支, 然后使用 git rebase 把新建的分支的基点 C1 重新指向到 C4, 然而 master 分支中并没有增加 D2 所修改的内容。

尽管 cherry-pick 和 merge 及 rebase 有一定的相似, 但是, 它们三者的语义却不同。

- cherry-pick: 侧重于把某个模块功能分支中的某次提交应用到主干分支。
- git merge: 侧重于把某两个分支合并。
- git rebase: 侧重于更新模块功能分支对应的基点。



## git reset 用法

reset 用于将当前分支的头 (HEAD) 复位到之前任意的某个提交点, 即忽略或者删除最近的某些提交。reset 常用的选项有两个, --soft 和 --hard。

首先来看软复位, 如以下代码所示:

```
$ git log --pretty=oneline
3ae2be7814116102da641813bdf762a5a74a880e rest ok
54a47f1d4f51f9b97e8f283648705ac754ee8b96 Merge branch 'development'
7baab210e6f9e516467b0528443c04e14681a7f2 add A_1
3f6c344fe1224151371ddb5f03dde2ffcc8abbcc merge from development
41714d84231cf1a362c019171f4ea51ec5dab379 add A1
0b677419b7f3c5dd0bea087ba691badb12ac89a9 this is C4
071523bd42a41393b0a1c4361365d50160090dd4 this is C3
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72bea1271ea first create
$ git reset --soft HEAD~2
$ git log --pretty=oneline
3f6c344fe1224151371ddb5f03dde2ffcc8abbcc merge from development
41714d84231cf1a362c019171f4ea51ec5dab379 add A1
0b677419b7f3c5dd0bea087ba691badb12ac89a9 this is C4
071523bd42a41393b0a1c4361365d50160090dd4 this is C3
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72bea1271ea first create
```

--soft 选项告知系统, 仅仅复位提交区, 而工作区和阶段区的内容保持不变。HEAD 表示当前分支的头, HEAD~2 的含义是越过 HEAD 前的 2 个提交, 对应 reset 前的第 4 个提交。HEAD 有另外一种写法 —— HEAD^^, 这个等价于 HEAD~2。

--hard 选项和 --soft 选项的区别在于, 除了复位提交区, 同时复位工作区和阶段区的内容。

hard 的语义是丢弃所做的修改, 因为工作区和阶段区的内容都会复位到指定的节点。soft 的语义是把过去某些提交合并为一个提交, 所以, 一般在软复位后, 往往需要调用 commit 命令再提交一次, 因为工作区和阶段区的内容依然保持了最后的状态。

`reset` 除了使用 `HEAD` 指定提交的位置外，也可以直接指定 `commit` 的标识，即 40 位 SHA-1 十六进制字符串。如以下代码所示：

```
$git reset --soft 41714d
```

由于 `reset` 可以直接指定要复位到的节点，这就意味着 `reset` 不但可以向前复位，还可以向后复位，只要知道目标节点的标识即可。

## 4.7 恢复到无引用提交

在 `git` 的提交记录中，有些已经成为“无引用”的状态，然而后来却发现这些“无引用”的提交是很有意义的，需要恢复。导致“无引用”的操作一般包含以下情况。

- 调用 `reset` 对当前 `branch` 的 `HEAD` 进行复位，复位后，被复位的提交就成了无引用的状态。

认为某个 `branch` 已经不再需要，于是删除了该 `branch`，此时该 `branch` 所特有的提交就成了“无引用”状态。

- 恢复“无引用”提交的方式可以非常灵活，无论采用什么方式，目标都是要找到无引用提交的标识，找到标识后就很容易恢复其中的内容。以下列出几种恢复的例子。

方式一：`git log`。列出当前分支中的提交历史。找到提交历史后，就可以从任意的节点重新建立一个分支，或者将当前分支的 `HEAD` 重新指向新的节点即可。这种方式适合于在 `reset` 之前刚刚调用过 `git log` 查看过提交记录，`reset` 之后又想立即恢复到之前的某个节点。如以下代码所示：

```
localhost:mygit keyd$ git log --pretty=oneline
a3be14bea69b687c4affdbf96108d906cc9dfa5a reset new
071523bd42a41393b0a1c4361365d50160090dd4 this is C3
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72bea1271ea first create
localhost:mygit keyd$
localhost:mygit keyd$ git reset --soft HEAD^^
localhost:mygit keyd$ git log --pretty=oneline
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72bea1271ea first create
localhost:mygit keyd$ git reset --soft a3be14bea69b687c4affdbf96108d906cc9dfa5a
localhost:mygit keyd$ git log --pretty=oneline
a3be14bea69b687c4affdbf96108d906cc9dfa5a reset new
071523bd42a41393b0a1c4361365d50160090dd4 this is C3
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72bea1271ea first create
```

方式二：`git reflog`。列出 `HEAD` 的变化历史，系统中只有一个 `HEAD`，因此，`HEAD` 历史会覆盖到任何分支的节点。这种方式适合于同一分支中由于 `reset` 而引起的“无引用”的情况，如图 4-11 所示。





图 4-11 M1 记录处于“无引用”状态

在这种情况下，刚开始认为 M1 没什么用，于是调用 `reset` 恢复到之前的 C2，然后针对 C2 继续开发，并连续提交了 C3 和 C4，可是此时又发现之前的 M1 是有用的，但是，使用 `git log` 找不到 M1，因为 `git log` 仅仅列出该分支上的线性提交历史。这个时候就可以使用 `git reflog`，该命令会依次列出 C4、C3、C2、M1，因为 M1 曾经也是当前分支的 HEAD。如以下代码所示：

```

localhost:mygit keyd$ git log --pretty=oneline
489e07ab49423bb4f56075107a2e42723413be7e C4
66702461d3daf095573ced4f04fae53833023d5d C3
39c2a2399a7e2721a4c3490c57ba0c51fe68a417 this is C2
31492482a0a3db01745a21f26f37c72beal271ea first create
localhost:mygit keyd$ git reflog -l -5
489e07a HEAD@{0}: commit: C4
6670246 HEAD@{1}: commit: C3
39c2a23 HEAD@{2}: HEAD^: updating HEAD
fe09052 HEAD@{3}: commit: M1
39c2a23 HEAD@{4}: 39c2a2399a7e2721a4c3490c57ba0c51fe68a417: updating HEAD
localhost:mygit keyd$ git branch mlrestore fe09052
  
```

`reflog` 后面的 `-l -5` 选项意思是限制 (limit) 输出最前面的 5 条。最后一条命令用于从 M1 处创建一个新的分支，名称为 `mlrestore`。

方式三：`git fsck -full`。列出所有未被引用的提交。这种方式适合于由于删除某个 `branch` 并且手工删除了 `.git/logs/` 目录后，而引起的“无引用”记录。这种情况不太多，因为一般情况下并不会手工删除 `.git/logs` 目录，`.git/logs/` 目录下保存了 HEAD 的变化情况，删除该目录的目的是为了清理系统，一旦从该目录中删除，那些“无引用”的提交将会真正被系统自动清理。如图 4-12 所示。

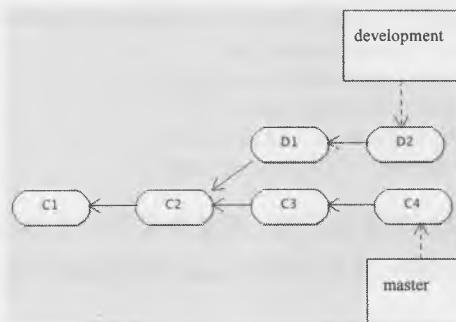


图 4-12 彻底删除记录后的状态

本例中,删除 developmetn 分支后,D1 和 D2 就成了无引用状态,当接着删除.git/logs 目录后,git reflog 就看不到 D1 和 D2。此时如果系统还没有启动清理无用的提交,那么就可以使用 git fsck 命令查看。该过程如以下代码所示:

```
$git branch -D development
$rm -Rf ./git/logs/
$git reflog
$git fsck --full
dangling commit 07b9badcc38304df92f9d0de5f72113c6fa25000
dangling commit 20cfb66627a3099c36ce08806da26b27a099bc7a
dangling commit 3ae2be7814116102da641813bdf762a5a74a880e
dangling blob 4170f6e80c1c3fab501abd2a5369971c42252f18
dangling blob 832004c148a0515b68d01fcb15507d154d2c00a6
```

此时,仅仅列出了一些“无引用”的对象,对象类型包括 commit、blob、tree 三种。我们需要的是 commit 类型,然而,从以上列表中并不能直接看出哪个 commit 对应 D2,哪个 ccommit 对应 D1,于是可以再使用 cat-file 命令,如以下代码所示:

```
localhost:logs keyd$ git cat-file -p 07b9badcc38304df92f9d0de5f72113c6fa25000
tree a8508b95d9ebe0bada8bc2a4407371e82d02f8c6
parent d32229c7f1d98b6ab7a244bfb5ea5a5530a3d634
author Yuandan Kerr <yuandanke@gmail.com> 1284953366 +0800
committer Yuandan Kerr <yuandanke@gmail.com> 1284953366 +0800

D2
$git branch temp 07b9badcc
```

git cat-file 用于显示出 commit 的详细信息,-p 的意思是 pretty,即以优雅的方式输出。找到该标识后,就可以基于该标识重新建立一个分支。

使用 git fsck 命令需要注意,一旦系统启动自动清理,那么就再也看不到之前的节点了。因此删除.git/logs/目录要慎重,除非确信“无引用”的对象已经无用。

## 4.8 git remote 用法

remote 命令用于和远程仓库进行交互。远程仓库、本地仓库、分支三者的关系如图 4-13 所示。

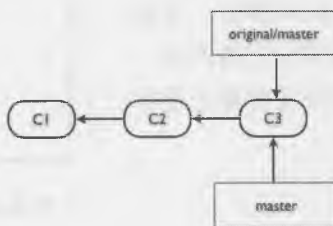


图 4-13 包含远程仓库引用的本地仓库组成

本地仓库会包括两个分支，如图 4-13 所示，其中 `original` 为远程仓库的名称，`master` 为远程仓库的分支名称，本地以 `name/branch` 的格式引用远程分支。对远程分支的操作不能像本地分支那样任意移动 HEAD 的位置，任意合并等，而必须依赖 `remote` 相关的命令进行有限的操作。如图 4-13 所示中的 `master` 分支是一个本地分支，这个也叫做 `original/master` 分支的跟踪（tracking）分支，即所有的修改首先在 `master` 分支中进行，修改完成后，再调用 `remote` 相关的命令，将 `master` 分支中的内容更新到 `original/master` 分支上。

要操作远程仓库，首先必须在本本地建立一份远程仓库的副本，`git clone` 命令用于复制一个远程仓库，而该命令实际上是 `remote` 相关命令的一个组合。`remote` 操作包括以下常见内容。

- 在本地建立一个远程仓库的引用。如以下命令所示：

```
$git remote add name url
```

比如：

```
$git remote add provider git://git.android.kernel.org/platform/msm.git
```

该命令仅仅是在本地建立了一个引用而已，命令执行完后，本地暂时还没有任何远程仓库的内容。

- 查看本地仓库都有哪些远程仓库的引用。

```
$git remote 或者 $git remote -v。
```

```
$git remote show name, 查看远程仓库 name 中的分支。
```

- 从远程仓库中取出内容，做一定的修改，然后再提交到远程仓库。包含两个命令，分别是 `fetch` 和 `pull`。`fetch` 会让本地仓库与远程仓库的内容保持一致，而 `pull` 除了拥有 `fetch` 的功能外，还会把新取出来的内容 `merge` 到本地跟踪（tracking）分支。如果 `merge` 有冲突，则调用标准的冲突解决方法，解决冲突后，可以重新提交，然后再调用 `git push` 命令把修改更新到远程仓库。如以下代码所示：

```
$git checkout master
$git fetch original
```

该命令执行前后的分支状态如图 4-14 所示，在执行 `fetch` 后，仅仅是改变了本地远程仓库分支的内容，即 `original/master` 内容改变，而本地的分支内容并没有改变。

再看以下代码：

```
$git pull original
```

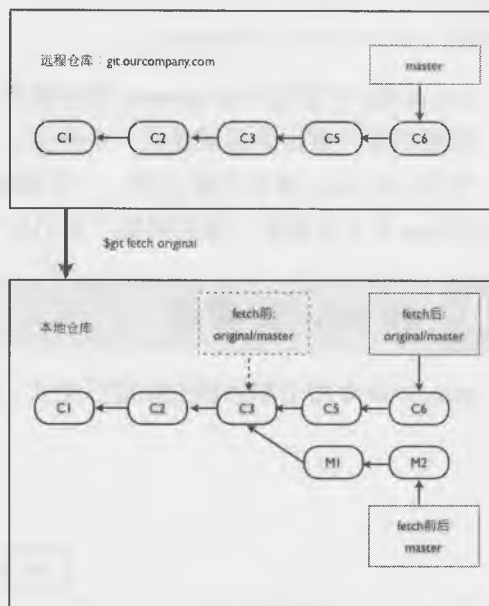


图 4-14 执行 `git fetch` 命令前后的状态变化

此处应该解决冲突，并接着运行以下命令：

```
$git commit -a -m "merge complete"
$git push original
```

git push 的语法如下：

```
git push git_name src:des
```

该命令参数有点像复制 (cp) 命令，git\_name 代表远程仓库的名称，src 代表本地分支的名字，des 是远程分支的名称，如果不写 des，则默认 des 和 src 同名称，即 git push git\_name master 和 git push git\_name master:master 是等价的。

- 创建一个远程分支的本地跟踪 (tracking) 分支。如以下命令所示：

```
$git checkout -b new_branch_name remote/name
```

或者运行以下代码。

```
$git checkout --track remote/name
```

该命令自动创建一个名称为 name 的本地分支，作为 remote/name 的跟踪分支。

- 删除一个远程分支。把一个空内容 push 到远程分支，就意味着删除该远程分支，如以下命令所示。

```
$git push git_name :des
```

即 des 前面的 src 为空。

## 4.9 git 配置

git 可以通过命令行配置，也可以直接修改相关配置文件，两者的结果是一样的，就是给相关的配置文件写入配置内容。配置的内容主要包括三个部分。第一部分，基本信息类配置，比如提交者的名称、e-mail 地址等；第二部分，第三方工具类配置，比如使用什么第三方工具进行合并 (merge) 操作等；第三部分，gitignore 配置，即告诉 git 系统排出对某些类型的管理。

git 配置文件包含三个，三者只是作用的范围不同，其中包含的配置内容则完全相同。这三个文件分别如下。

- /etc/gitconfig: 适应于全局的配置，使用 git config --system name value 进行配置。Snow leopard 对应的文件为 /opt/local/etc/gitconfig。
  - ~/.gitconfig: 适应于当前用户，使用 git config --global name value 进行配置，每个用户目录下都可以包含一个该文件。
  - .git/config: 适应于当前工作的仓库，使用 git config --local name value 进行配置。
- 三个文件的优先级是前者屈服后者，即当前工作仓库下的 .git/config 文件具有最高的优先权。

### 4.9.1 基本信息配置

基本信息包括提交者的名称、提交者的 E-mail，如以下代码所示：

```
$git config user.name 'Yuandan Kerr'
$git config user.email 'yuandanke@gmail.com'
```

可以使用 `git config --list` 命令查看当前的配置列表，或者使用 `git config prop_name` 命令查看指定属性的名称，如以下代码所示：

```
keyd:mygit keyd$ git config --list
color.ui=auto
user.name=Yuandan Kerr
user.email=yuandanke@gmail.com
core.excludesfile=/Users/keyd/.gitignore
merge.tool=extMerge
mergetool.extMerge.cmd=extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"

mergetool.trustexitcode=false
diff.external=extDiff
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
keyd:mygit keyd$ git config user.name
Yuandan Kerr
keyd:mygit keyd$
```

### 4.9.2 merge、diff 工具配置

`merge` 工具的最基本作用就是提供了一种友好的视图，让用户更直观地看到两个源文件之间的差别。笔者在 Linux 上一般使用 `meld`，而在 Mac 上常使用 `p4merge`，`p4merge` 是免费的，其网址为 <http://www.perforce.com/perforce/products/merge.html>。当进行各种 `merge` 操作时，如果有冲突，`git` 系统会提示用户 `merge` 失败。此时，用户可以立即调用 `$git mergetool` 命令启动相应的 `merge` 工具，`git` 则会以冲突的文件作为 `merge` 工具的启动参数，给用户一个友好的视图。当用户修改完后，可以再次调用 `$git mergetool` 修改下一个冲突的地方，直到全部修改完。修改完后，用户可以调用 `$git commit` 重新提交，然后再重新进行 `merge` 操作。

`diff` 工具主要用于产生文件的差异内容。

以下介绍在 Mac 上配置 `merge`、`diff` 工具的步骤，Linux 系统可以参照编写。

① 编写两个脚本文件，分别命名为 `extDiff` 和 `extMerge`，文件内容如下：

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
```

```
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

```
$ cat /usr/local/bin/extDiff
```

```
#!/bin/sh
```

```
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

② 将这两个文件保存到用户可执行文件目录下，对于 Mac 是 /usr/local/bin，并为这两个文件增加执行权限。如以下代码所示：

```
$ sudo chmod +x /usr/local/bin/extMerge
```

```
$ sudo chmod +x /usr/local/bin/extDiff
```

③ 配置 git 所使用的工具，如以下代码所示：

```
$ git config --global merge.tool extMerge
```

```
$ git config --global mergetool.extMerge.cmd \
```

```
'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
```

```
$ git config --global mergetool.trustExitCode false
```

```
$ git config --global diff.external extDiff
```

下面来检验一下 merge 工具的效果，如以下命令所示：

```
keyd:mygit keyd$ git merge dev
```

```
Auto-merging a.c
```

```
CONFLICT (content): Merge conflict in a.c
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

```
keyd:mygit keyd$ git mergetool
```

以上命令提示有冲突，此时立即执行 git mergetool，则开始运行以下 p4merge，如图 4-15 所示。

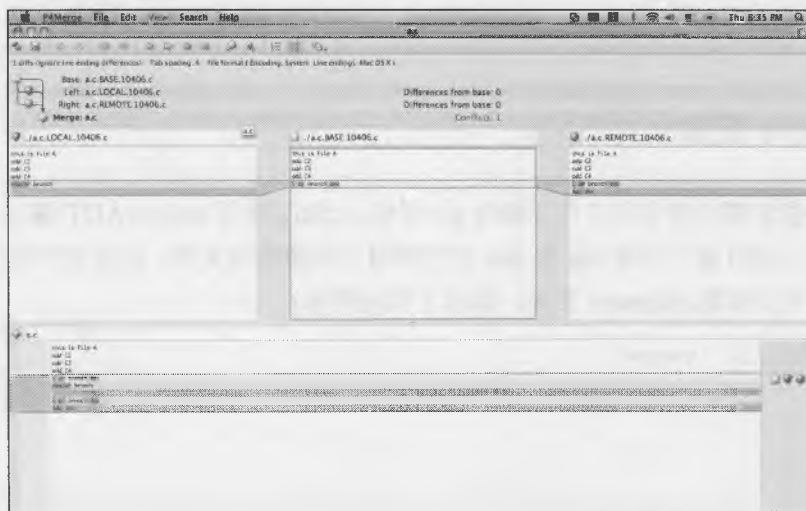


图 4-15 使用 p4merge 的程序操作界面示意

此时可以手工修改冲突的地方，修改完毕后保存，然后执行以下命令。

```
keyd:mygit keyd$ git commit -a -m "commit merge"
```

这样就完成了 merge→冲突→修改→提交→成功。

下面再检验一下 mergediff 工具，在命令行上运行以下代码。

```
$git checkout master  
$git diff
```

运行结果如图 4-16 所示。

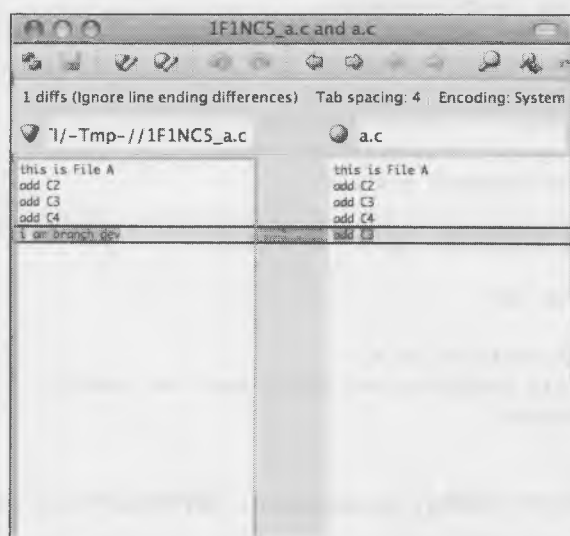


图 4-16 git diff 工具界面

### 4.9.3 .gitignore 配置

.gitignore 文件指定哪些类型的文件不接受 git 管理。比如，使用 eclipse ADT 编写 Android 应用程序时，ADT 会自动在工程目录下创建 bin 和 gen 文件夹用于保留临时文件，这些文件对于版本管理来讲毫无意义。此时，就可以编辑.gitignore 文件，如以下代码所示：

```
# a comment - this is ignored  
bin  
gen  
.classpath  
.project
```

.gitignore 支持的语法规则如下。

- [abc]: 包含 abc 任意字符。

- [0-9]: 包含 0~9 之间任意字符。
- ?: 单个任意字符。
- \*: 任意字符串。
- !: 排除。

使用.gitignore时要注意一个“半路出家”的问题，即建立仓库时，把所有的文件添加到了git仓库，而半路上才意识到需要排除某些类型的文件，那么，此时修改.gitignore是不会起作用的。要解决“半路出家”问题，首先需要删除掉仓库中的所有需要排除的文件，如下命令所示：

```
$git rm -r --cached bin
```

--cached 的含义是仅仅删除仓库中的内容，工作目录中的文件保持不变。

有时，某种类型的文件会存在于工作目录的各个文件夹下，则可以使用以下连环命令，比如要删除所有目录下的svn目录。

```
$find . -iname ".svn" | xargs git rm -r --cached
```

#### 4.10 同时使用 git 和 svn

由于历史原因，很多公司内部依然在使用svn版本管理系统。svn的不便在于客户端只能保持一个和服务端相同的分支。如果某位程序员想先在本地做一些尝试性的开发，并希望暂时不要把这些代码提交到服务器，因为这些新的功能可能不稳定，然而，公司却要求每位程序员都要定期（比如每两天或者一个礼拜）将代码提交到服务器上，此时，就可以考虑在本地使用git。

svn客户端每次调用svn ci命令时，提交的内容相对于git来讲，就是工作区的内容。所以，可以在工作目录下建立一个git仓库，仓库中可以包含多个分支，包括程序员想做的任意功能创新，而当公司要求提交时，则使用git checkout命令把工作区更新为稳定的分支即可，如图4-17所示。

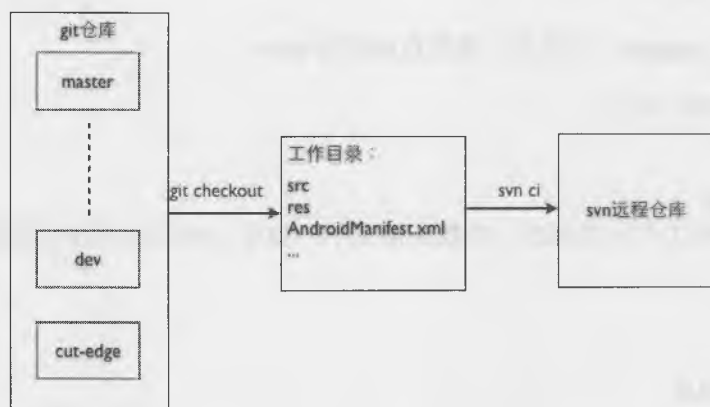


图 4-17 git 和 svn 协同工作的结构



在 svn 的客户端中，每个目录里面都包含一个.svn 文件夹，这对于 git 来讲是不需要的。因此，需要配置工作目录下面的.gitignore 文件，告知 git 排除.svn 即可。

创建 svn 工作目录并将其加入 git 管理的步骤如以下代码所示，假设用户的 svn 服务器仓库为 svn://192.168.1.3/svn/contacts。

```
$svn co svn://192.168.1.3/svn/contacts
$cd contacts
$git init
$echo ".svn" >> .gitignore
$git add *
$git commit -a -m "first creation"
```

此时，就可以对 contacts 目录下的代码进行各种修改了，包括建立更多的 git 分支等。要提交某个 branch 对应的内容到 svn 服务器时，只需要先 checkout 出该分支，然后调用 svn ci 即可。

## 4.11

### 其他 git 常用命令示例

以下仅列出一些其他常用命令，而各个命令的完整用法可以在命令行下输入 `git help cmd_name`，来查询名称为 `cmd_name` 的用法。比如，输入 `git help branch` 就可以查询 `git branch` 的全部用法。

#### 4.11.1 git branch

创建名为 sensor 的分支。

```
$git branch sensor
```

创建名为 sensor 的分支，该分支基于 hardware 分支。

```
$git branch sensor hardware
```

创建基于某个节点（commit）的分支，该节点标识为 sha-1。

```
$git branch branch_name sha-1
```

删除 sensor 分支。

```
$git branch -D sensor
```

-D 选项表示的是忽略工作区的修改，在默认情况下，如果工作区的内容没有提交（commit），git 会阻止删除当前分支。

#### 4.11.2 git checkout

创建名为 sensor 的分支，并将其提取到当前工作区。

```
$git checkout -b sensor
```

创建名为 **sensor** 的分支，并将其提取到当前工作区，该分支基于 **hardware**。

```
$git checkout -b sensor hardware
```

如果知道某个节点(commit)的 sha-1 标识，也可以基于该标识创建新的分支。

```
$git checkout -b branch_name sha-1
```

### 4.11.3 git log

限制 log 输出 5 个最新的节点(commit)。

```
$git log -l -5
```

希望每条 log 仅占一行输出。

```
$git log --pretty=oneline
```

仅查看某两个节点之间的 log。

```
$git log since..untill
```

其中 **since** 和 **untill** 对应节点的 sha-1 标识。

### 4.11.4 git commit --amend

当用户刚刚提交后，却觉得有点低级错误，比如备注信息填写有误等，想修改 (**amend**) 一下上次的提交，则可以调用该命令。调用该命令后，git 会启动 vi 编辑器，提示用户重新写入备注信息。

该命令等价于以下命令：

```
$git reset --soft HEAD^
$git commit -c ORIG_HEAD
```

其中 **ORIG\_HEAD** 指向 **HEAD** 的上一个位置。**ORIG\_HEAD** 并不总是等于 **HEAD@{1}**，在正常的提交过程中，**HEAD@{1}** 总是变化的，但是 **ORIG\_HEAD** 只有当使用 **reset** 等直接改变 **HEAD** 位置时才改变。

### 4.11.5 git cherry-pick sha-1

“挑最好的那个樱桃”，这是 **cherry-pick** 的意思，该命令一般用于把某个提交单独应用于当前的分支。

#### 4.11.6 git merge-base

想找到两个分支的最优共同父节点，如图 4-18 所示。

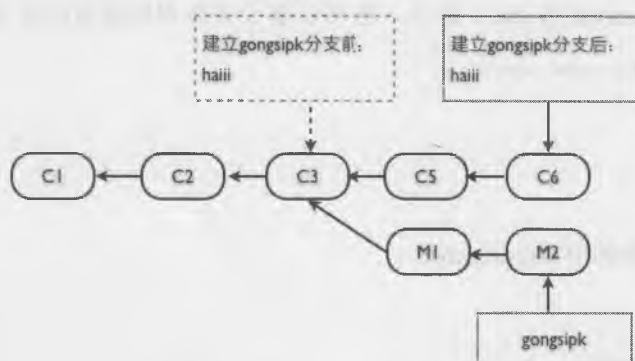


图 4-18 寻找两个分支的最优父节点

在该图中，haiiii 和 gongsipk 的共同父节点为 C3，则可以使用如下命令显示出 C3 的 sha-1 标识。

```
$git merge-base haiiii gongsipk
66702461d3daf095573ced4f04fae53833023d5d
```

#### 4.11.7 git diff master...dev

该命令用于查看两个分支的不同，如上一小节图 4-18 所示，想查看 gongsipk 和 haiiii 分支的差别，可以使用命令 `git diff gongsipk haiiii`。但这里有个问题，我们本来的目的可能是想查看 M2 和 C3 之间的差别，但是，该命令却反映的是 C6 和 M2 的区别，原因是在创建 gongsipk 分支后，原来的 haiiii 分支有了新的提交——C5、C6。

解决这个问题一个常规办法是使用以下命令组合。

```
$git merge-base haiiii master
66702461d3daf095573ced4f04fae53833023d5d
$git checkout gongsipk
$git diff 66702461d3daf095573ced4f04fae53833023d5d
```

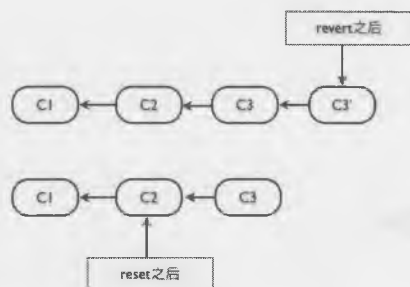
这个问题的另一个办法就是使用以下命令。

```
$git diff haiiii...gongsipk
```

其中 haiiii 和 gongsipk 之间有个“...”，该命令的含义是查看后者（gongsipk）与前者（haiiii）共同父节点（C3）之间的差异。

### 4.11.8 git revert

`revert` 和 `reset` 都有恢复、撤销、回滚的意思，从工作区的角度来看，`revert` 和 `reset` 基本是相同的，都会把工作区的内容刷新为之前的某次提交。



```
$git revert HEAD
$git revert HEAD^
```

但是从提交区的角度来看，`revert` 和 `reset` 却很不一样。`reset` 会忽略被复位的提交，在 `log` 中将不会再出现这些提交，而 `revert` 仅仅是把工作区的内容刷新为之前的某次提交，并且在 `log` 中重新添加了一个新的提交，如图 4-19 所示。

从程序员的角度来讲，如果认为过去的某些提交没什么意义，则可以使用 `reset` 进行恢复；如果只是暂时回到过去某个提交，则应该使用 `revert`。尽管如此，为了思路清楚，笔者建议使用新建 `branch` 来开辟一个新的工作区，而避免使用 `revert` 进行恢复。

### 4.11.9 git diff

比较当前工作区的内容和上一个提交之间的差别。

```
$git diff HEAD
```

比较当前阶段区和上一个提交之间的差别，如果当前的修改没有被添加到阶段区，输出为空。

```
$git diff --cached HEAD
```

比较三个分支之间的差别，基准是 `master`，规则是 `diff(master, dev)` 和 `diff(master, temp)` 的并集。

```
$git diff master dev temp
```

### 4.11.10 git rm

在 `git` 系统中，不要直接从工作中的文件夹删除某个文件，而是应该使用 `git rm` 命令。否则，`git` 认为仅仅是某个提交中需要删除该文件而已。

与 `git rm` 命令作用相反的是 `git add`，后者用于告知 `git` 把某个文件（夹）添加到 `git` 管理仓库中，而 `git rm` 则是告知 `git` 从仓库中移除某个文件（夹）。

删除仓库中的 `.svn` 文件夹，同时删除工作目录下的 `.svn` 文件夹：

```
$git rm -rf .svn
```

-R 选项用于递归删除。f 选项是强制删除，在默认情况下，如果修改后的文件没有提交（commit），那么是不能删除该文件（夹）的。

删除仓库中的.svn，而保留工作目录下的.svn 文件：

```
$git rm -Rf --cached .svn
```

--cached 选项的意思是仅删除 git 仓库的内容。

### 4.11.11 git tag

在下载 Android 源码时，会发现 Android 仓库中有很多 tag，比如：

```
android-1.6_r1.1
android-1.6_r1.2
android-1.6_r1.3
android-1.6_r1.4
android-1.6_r1.5
android-1.6_r2
```

git 提供了给某个提交附加一个标签的功能，从而方便以后直接从这个标签来引用这个提交。给当前分支添加一个 tag：

```
$git tag -a v1.4 -m "my version 1.4"
```

给某个 commit 添加一个 tag：

```
$git tag -a v1.2 sha-1 ,sha-a 为该 commit 的标识
```

查看当前所有 tag：

```
$git tag
```

使用通配符，查看 tag 列表：

```
$git tag -l 'v1.4.2.*'
```

查看指定名称的 tag：

```
$git show v1.4
```



## 第2部分 内核篇

- 第5章 Binder
- 第6章 Framework 概述
- 第7章 理解 Context
- 第8章 创建窗口的过程
- 第9章 Framework 的启动过程
- 第10章 Ams 内部原理
- 第11章 从输入设备中获取消息
- 第12章 屏幕绘图基础
- 第13章 View 工作原理
- 第14章 WmS 工作原理



## 第 5 章 Binder

Binder，英文的意思是别针、回形针。我们经常用别针把两张纸“别”在一起，而在 Android 中，Binder 用于完成进程间通信（IPC），即把多个进程“别”在一起。比如，普通应用程序可以调用音乐播放服务提供的播放、暂停、停止等功能。

Binder 工作在 Linux 层面，属于一个驱动，只是这个驱动不需要硬件，或者说其操作的硬件是基于一小段内存。从线程的角度来讲，Binder 驱动代码运行在内核态，客户端程序调用 Binder 是通过系统调用完成的。

### 5.1 Binder 框架

Binder 是一种架构，这种架构提供了服务端接口、Binder 驱动、客户端接口三个模块，如图 5-1 所示。

首先来看服务端。一个 Binder 服务端实际上就是一个 Binder 类的对象，该对象一旦创建，内部就启动一个隐藏线程。该线程接下来会接收 Binder 驱动发送的消息，收到消息后，会执行到 Binder 对象中的 `onTransact()` 函数，并按照该函数的参数执行不同的服务代码。因此，要实现一个 Binder 服务，就必须重载 `onTransact()` 方法。

可以想象，重载 `onTransact()` 函数的主要内容是把 `onTransact()` 函数的参数转换为服务函数的参数，而 `onTransact()` 函数的参数来源是客户端调用 `transact()` 函数时输入的，因此，如果 `transact()` 有固定格式的输入，那么 `onTransact()` 就会有固定格式的输出。

下面再看 Binder 驱动。任意一个服务端 Binder 对象被创建时，同时会在 Binder 驱动中创建一个 `mRemote` 对象，该对象的类型也是 Binder 类。客户端要访问远程服务时，都是通过 `mRemote` 对象。

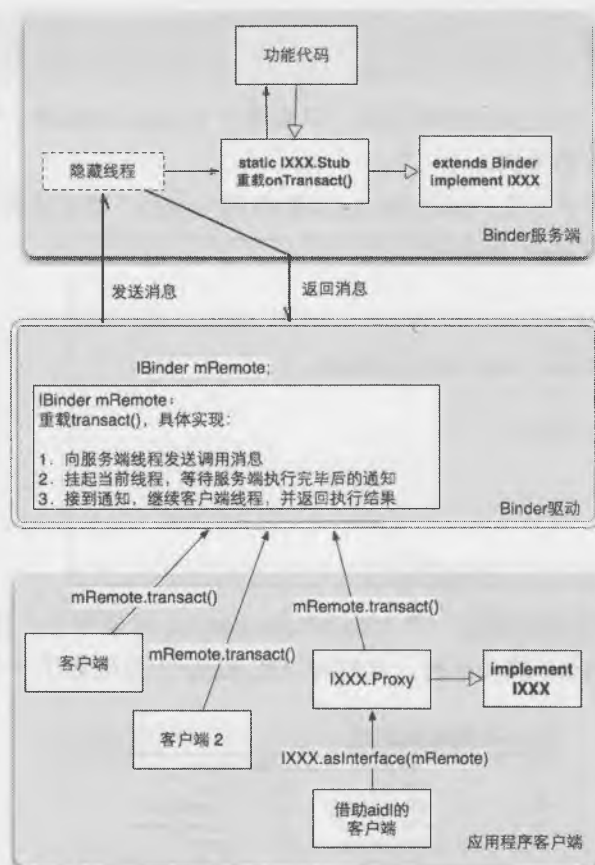


图 5-1 Binder 框架

最后来看应用程序客户端。客户端要想访问远程服务，必须获取远程服务在 Binder 对象中对应的 mRemote 引用，至于如何获取，下面几节将要介绍。获得该 mRemote 对象后，就可以调用其 transact() 方法，而在 Binder 驱动中，mRemote 对象也重载了 transact() 方法，重载的内容主要包括以下几项。

- 以线程间消息通信的模式，向服务端发送客户端传递过来的参数。
- 挂起当前线程，当前线程正是客户端线程，并等待服务端线程执行完指定服务函数后通知 (notify)。
- 接收到服务端线程的通知，然后继续执行客户端线程，并返回到客户端代码区。

从这里可以看出，对应用程序开发人员来讲，客户端似乎是直接调用远程服务对应的 Binder，而事实上则是通过 Binder 驱动进行了中转。即存在两个 Binder 对象，一个是服务端的 Binder 对象，另一个则是 Binder 驱动中的 Binder 对象，所不同的是 Binder 驱动中的对象不会再额外产生一个线程。



## 5.2 设计 Servier 端

设计 Service 端很简单,从代码的角度来讲,只要基于 Binder 类新建一个 Servier 类即可。以下以设计一个 MusicPlayerService 类为例。

假设该 Service 仅提供两个方法: start(String filePath)和 stop(),那么该类的代码可以如下:

```
public class MusicPlayerService extends Binder{
    @Override
    protected boolean onTransact(int code, Parcel data, Parcel reply, int flags)
        throws RemoteException {
        return super.onTransact(code, data, reply, flags);
    }

    public void start(String filePath){
    }

    public void stop(){
    }
}
```

当要启动该服务时,只需要初始化一个 MusicPlayerService 对象即可。比如可以在主 Activity 里面初始化一个 MusicPlayerService,然后运行,此时可以在 ddms 中发现多了一个线程,如图 5-2 所示。

ID	Tid	Status	utime	stime	Name
1	437	wait	25	30	main
*2	438	vmwait	0	9	HeapWorker
*3	439	vmwait	0	0	Signal Catcher
*4	440	running	5	9	JDWP
5	441	native	0	0	Binder Thread #1
6	442	native	0	0	Binder Thread #2
7	443	native	0	0	Binder Thread #3

图 5-2 使用 ddms 工具查看 Binder 对应的线程

如果不创建 MusicPlayerService,则只有三个 Binder 对象对应的线程。

定义了服务类后,接下来需要重载 onTrasact()方法,并从 data 变量中读出客户端传递的参数,比如 start()方法所需要的 filePath 变量。然而,这里有个问题,服务端如何知道这个参数在 data 变量中的位置?因此,这就需要调用者和服务者双方有个约定。

这里假定客户端在传入的包裹 data 中放入的第一个数据就是 filePath 变量,那么, onTransact()的代码可以如下所示:

```
switch(code){
    case 1000:
        data.enforceInterface("MusicPlayerService");
        String filePath = data.readString();
        start(filePath);
        //replay.writeXXX();
        break;
}
```

code 变量用于标识客户端期望调用服务端的哪个函数,因此,双方需要约定一组 int 值,不同的值代表不同的服务端函数,该值和客户端的 transact() 函数中第一个参数 code 的值是一致的。这里假定 1000 是双方约定要调用 start() 函数的值。

enforceInterface() 是为了某种校验,它与客户端的 writeInterfaceToken() 对应,具体见下一小节。

readString() 用于从包裹中取出一个字符串。取出 filePath 变量后,就可以调用服务端的 start() 函数了。

如果该 IPC 调用的客户端期望返回一些结果,则可以在返回包裹 reply 中调用 Parcel 提供的相关函数写入相应的结果。

### 5.3 Binder 客户端设计

要想使用服务端,首先要获取服务端在 Binder 驱动中对应的 mRemote 变量的引用,获取的方法后面小节将有介绍。获得该变量的引用后,就可以调用该变量的 transact() 方法。该方法的函数原型如下:

```
public final boolean transact(int code, Parcel data, Parcel reply, int flags)
```

其中 data 表示的是要传递给远程 Binder 服务的包裹(Parcel),远程服务函数所需要的参数必须放入这个包裹中。包裹中只能放入特定类型的变量,这些类型包括常用的原子类型,比如 String、int、long 等,要查看包裹可以放入的全部数据类型,可以参照 Parcel 类。除了一般的原子变量外,Parcel 还提供了一个 writeParcel() 方法,可以在包裹中包含一个小包裹。因此,要进行 Binder 远程服务调用时,服务函数的参数要么是一个原子类,要么必须继承于 Parcel 类,否则,是不能传递的。

因此,对于 MediaPlayerService 的客户端而言,可以如下调用 transact() 方法。

```
IBinder mRemote = null;
String filePath = "/sdcard/music/heal_the_world.mp3";
int code = 1000;
Parcel data = Parcel.obtain();
Parcel reply = Parcel.obtain();
data.writeInterfaceToken("MediaPlayerService");
data.writeString(filePath);
mRemote.transact(code, data, reply, 0);
IBinder binder = reply.readStrongBinder();
reply.recycle();
data.recycle();
```

现在来分析以上代码。首先,包裹不是客户端自己创建的,而是调用 Parcel.obtain() 申请的,这正如生活中的邮局一样,用户一般只能用邮局提供的信封(尤其是 EMS)。其中 data 和 reply 变量都由客户端提供,reply 变量用户服务端把返回的结果放入其中。

writeInterfaceToken() 方法标注远程服务名称,理论上讲,这个名称不是必需的,因为客户端既然已经获取指定远程服务的 Binder 引用,那么就不会调用到其他远程服务。该名称将作为 Binder 驱动确保客户端的确想调用指定的服务端。

writeString() 方法用于向包裹中添加一个 String 变量。注意,包裹中添加的内容是有序的,这个顺序必须是客户端和服务端事先约定好的,在服务端的 onTransact() 方法中会按照约定的顺序取出变量。

接着调用 `transact()` 方法。调用该方法后，客户端线程进入 Binder 驱动，Binder 驱动就会挂起当前线程，并向远程服务发送一个消息，消息中包含了客户端传进来的包裹。服务端拿到包裹后，会对包裹进行拆解，然后执行指定的服务函数，执行完毕后，再把执行结果放入客户端提供的 `reply` 包裹中。然后服务端向 Binder 驱动发送一个 `notify` 的消息，从而使得客户端线程从 Binder 驱动代码区返回到客户端代码区。

`transact()` 的最后一个参数的含义是执行 IPC 调用的模式，分为两种：一种是双向，用常量 0 表示，其含义是服务端执行完指定服务后会返回一定的数据；另一种是单向，用常量 1 表示，其含义是不返回任何数据。

最后，客户端就可以从 `reply` 中解析返回的数据了，同样，返回包裹中包含的数据也必须是有序的，而且这个顺序也必须是服务端和客户端事先约定好的。

## 5.4 使用 Service 类

以上手工编写 Binder 服务端和客户端的过程存在两个重要问题。

第一，客户端如何获得服务端的 Binder 对象引用。

第二，客户端和服务端必须事先约定好两件事情。

- 服务端函数的参数在包裹中的顺序。
- 服务端不同函数的 `int` 型标识。

关于第一个问题，请思考为什么要用 Binder。答案很简单，即为了提供一个全局服务，所谓的“全局”，是指系统中的任何应用程序都可以访问。很明显，这是一个操作系统应该提供的最基本的功能之一，Android 的工程师自然也是这么认为的，因此，他们提供了一个更傻瓜的解决方法，那就是 `Service`。它是 Android 应用程序四个基本程序片段（Component）之一，四个基本片段包括 `Activity`、`Service`、`Content Provider`、`Receiver`。

无论是否使用 `Service` 类，都必须解决以上两个重要问题。因此，下面先介绍如何解决第一个问题。

### 5.4.1 获取 Binder 对象

事实上，对于有创造力的程序员来讲，可以完全不使用 `Service` 类，而仅仅基于 `Binder` 类编写服务程序，但只是一部分。具体来讲，可以仅使用 `Binder` 类扩展系统服务，而对于客户端服务则必须基于 `Service` 类来编写。所谓的系统服务是指可以使用 `getSystemService()` 方法获取的服务，所谓的客户端服务是指应用程序提供的自定义服务。

关于 `Service` 的内部机制请参考第 14 章，本章仅指出 `Service` 和 `Binder` 的关系。那么，`Service` 类是如何解决本节开头所提出的两个重要问题的呢？

首先，`AmS` 提供了 `startService()` 函数用于启动客户服务，而对于客户端来讲，可以使用以下两个函数来和一个服务建立连接，其原型在 `android.app.ContextImpl` 类中。

```
public ComponentName startService(Intent intent);
```

该函数用于启动 `intent` 指定的服务，而启动后，客户端暂时还没有服务端的 `Binder` 引用，因此，暂时还不能调用任何服务功能。

```
public boolean bindService(Intent service, ServiceConnection conn, int flags);
```

该函数用于绑定一个服务，这就是第一个重要问题的关键所在。其中第二个参数是一个 `interface` 类，该 `interface` 的定义如以下代码所示：

```
public interface ServiceConnection {
    public void onServiceConnected(ComponentName name, IBinder service);
    public void onServiceDisconnected(ComponentName name);
}
```

请注意该 `interface` 中的 `onServiceConnected()` 方法的第二个变量 `Service`。当客户端请求 `AmS` 启动某个 `Service` 后，该 `Service` 如果正常启动，那么 `AmS` 就会远程调用 `ActivityThread` 类中的 `ApplicationThread` 对象，调用的参数中会包含 `Service` 的 `Binder` 引用，然后在 `ApplicationThread` 中会回调 `bindService` 中的 `conn` 接口。因此，在客户端中，可以在 `onServiceConnected()` 方法中将其参数 `Service` 保存为一个全局变量，从而在客户端的任何地方都可以随时调用该远程服务。这就解决了第一个重要问题，即客户端如何获取远程服务的 `Binder` 引用。

以上流程如图 5-3 所示。

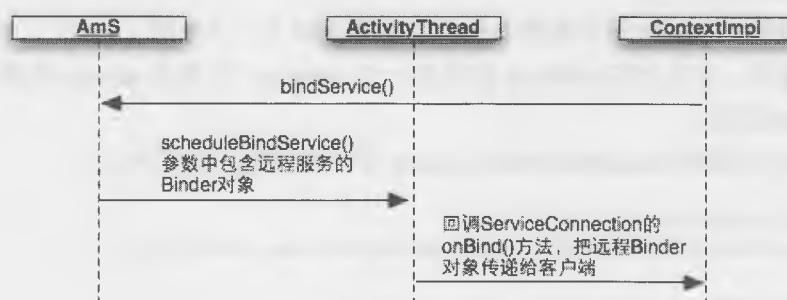


图 5-3 Binder 客户端和服务端的调用过程

## 5.4.2 保证包裹内参数顺序 `aidl` 工具的使用

关于第二个问题，`Android` 的 `SDK` 中提供了一个 `aidl` 工具，该工具可以把一个 `aidl` 文件转换为一个 `Java` 类文件，在该 `Java` 类文件，同时重载了 `transact` 和 `onTransact()` 方法，统一了存入包裹和读取包裹参数，从而使设计者可以把注意力放到服务代码本身上。

`aidl` 工具不是必需的，对于有经验的程序员来讲，手工编写一个参数统一的包裹存入和包裹读出代

码并不是一件复杂的事情。

接下来看 `aidl` 工具都做了什么。如本章第一节示例，此处依然假设要编写一个 `MusicPlayerService` 服务，服务中包含两个服务函数，分别是 `start()` 和 `stop()`。那么，可以首先编写一个 `IMusicPlayerService.aidl` 文件。如以下代码所示：

```
package com.haiiii.android.client;
interface IMusicPlayerService{
    boolean start(String filePath);
    void stop();
}
```

该文件的名称必须遵循一定的规范，第一个字母“`I`”不是必需的，但是，为了程序风格的统一，“`I`”的含义是 `IInterface` 类，即这是一个可以提供访问远程服务的类。后面的命名——`MusicPlayerService` 对应的是服务的类名，可以是任意的，但是，`aidl` 工具会以该名称命名输出的 `Java` 类。这些规则都只是 `Eclipse` 下 `ADT` 插件的默认规则，`aidl` 本身只是一个命令程序，借助命令行的话，则可以灵活指定输出文件的名称及位置。具体使用方法参照 `aidl` 的执行帮助信息。

`aidl` 文件的语法基本类似于 `Java`，`package` 指定输出后的 `Java` 文件对应的包名。如果该文件需要引用其他 `Java` 类，则可以使用 `import` 关键字，但需要注意的是，包裹内只能写入以下三个类型的内容。

- `Java` 原子类型，如 `int`、`long`、`String` 等变量。
- `Binder` 引用。
- 实现了 `Parcelable` 的对象。

因此，基本上来讲，`import` 所引用的 `Java` 类也只能是以上三个类型。

`interface` 为关键字，有时会在 `interface` 前面加一个 `oneway`，代表该 `service` 提供的方法都是没有返回值的，即都是 `void` 类型。

下面看看该 `aidl` 生成的 `IMusicPlayerService.java` 文件的代码。如下所示：

```
package com.haiiii.client;
public interface IMusicPlayerService extends android.os.IInterface
{
    /** Local-side IPC implementation stub class. */
    public static abstract class Stub extends android.os.Binder
        implements com.haiiii.client.IMusicPlayerService
    {
        private static final java.lang.String DESCRIPTOR =
            "com.haiiii.client.IMusicPlayerService";

        /** Construct the stub at attach it to the interface. */
        public Stub()
        {
            this.attachInterface(this, DESCRIPTOR);
        }

        /**
         * Cast an IBinder object into an com.haiiii.client.IMusicPlayerService interface,
```

```

    * generating a proxy if needed.
    */
    public static com.haiii.client.IMusicPlayerService
        asInterface(android.os.IBinder obj)
    {
        if ((obj==null)) {
            return null;
        }

        android.os.IInterface iin =
            (android.os.IInterface) obj.queryLocalInterface(DESCRIPTOR);

        if (((iin!=null)&&(iin instanceof com.haiii.client.IMusicPlayerService))) {
            return ((com.haiii.client.IMusicPlayerService) iin);
        }
        return new com.haiii.client.IMusicPlayerService.Stub.Proxy(obj);
    }

    public android.os.IBinder asBinder()
    {
        return this;
    }

    @Override
    public boolean onTransact(int code, android.os.Parcel data,
        android.os.Parcel reply, int flags) throws android.os.RemoteException
    {
        switch (code)
        {
            case INTERFACE_TRANSACTION:
            {
                reply.writeString(DESCRIPTOR);
                return true;
            }
            case TRANSACTION_start:
            {
                data.enforceInterface(DESCRIPTOR);
                java.lang.String _arg0;
                _arg0 = data.readString();
                boolean _result = this.start(_arg0);
                reply.writeNoException();
                reply.writeInt((_result)?(1):(0));
                return true;
            }
            case TRANSACTION_stop:
            {
                data.enforceInterface(DESCRIPTOR);
                this.stop();
                reply.writeNoException();
                return true;
            }
        }
    }

```

```

        return super.onTransact(code, data, reply, flags);
    }

    private static class Proxy implements com.haiiii.client.IMusicPlayerService
    {
        private android.os.IBinder mRemote;
        Proxy(android.os.IBinder remote)
        {
            mRemote = remote;
        }

        public android.os.IBinder asBinder()
        {
            return mRemote;
        }

        public java.lang.String getInterfaceDescriptor()
        {
            return DESCRIPTOR;
        }

        public boolean start(java.lang.String filePath) throws android.os.RemoteException
        {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            boolean _result;
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                _data.writeString(filePath);
                mRemote.transact(Stub.TRANSACTION_start, _data, _reply, 0);
                _reply.readException();
                _result = (0!=_reply.readInt());
            }
            finally {
                _reply.recycle();
                _data.recycle();
            }
            return _result;
        }

        public void stop() throws android.os.RemoteException
        {
            android.os.Parcel _data = android.os.Parcel.obtain();
            android.os.Parcel _reply = android.os.Parcel.obtain();
            try {
                _data.writeInterfaceToken(DESCRIPTOR);
                mRemote.transact(Stub.TRANSACTION_stop, _data, _reply, 0);
                _reply.readException();
            }
            finally {
                _reply.recycle();
                _data.recycle();
            }
        }
    }

```

```

        static final int TRANSACTION_start = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
        static final int TRANSACTION_stop = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
    }

    public boolean start(java.lang.String filePath) throws android.os.RemoteException;
    public void stop() throws android.os.RemoteException;
}

```

这些代码主要完成以下三个任务。

- 定义一个 Java interface，内部包含 aidl 文件所声明的服务函数，类名称为 `IMusicPlayerService`，并且该类基于 `IInterface` 接口，即需要提供一个 `asBinder()` 函数。
- 定义一个 Proxy 类，该类将作为客户端程序访问服务端的代理。所谓的代理主要就是为了前面所提到的第二个重要问题——统一包裹内写入参数的顺序。
- 定义一个 Stub 类，这是一个 abstract 类，基于 Binder 类，并且实现了 `IMusicPlayerService` 接口，主要由服务端来使用。该类之所以要定义为一个 abstract 类，是因为具体的服务函数必须由程序员实现，因此，`IMusicPlayerService` 接口中定义的函数在 Stub 类中可以没有具体实现。同时，在 Stub 类中重载了 `onTransact()` 方法，由于 `transact()` 方法内部给包裹内写入参数的顺序是由 aidl 工具定义的，因此，在 `onTransact()` 方法中，aidl 工具自然知道应该按照何种顺序从包裹中取出相应参数。

在 Stub 类中还定义了一些 int 常量，比如 `TRANSACTION_start`，这些常量与服务函数对应，`transact()` 和 `onTransact()` 方法的第一个参数 `code` 的值即来源于此。

刚接触 `IMusicPlayerService` 时，对以上描述的三个任务不容易从代码中看出，原因是这三个任务似乎更应该是分离的三个类，而 aidl 工具却把这些都放入了一个类中。理论上讲，的确可以把这三个任务写成三个类，但那会增加代码维护的烦琐。

在 Stub 类中，除了以上所述的任务外，Stub 还提供了一个 `asInterface()` 函数。提供这个函数的作用是这样的：首先需要明确的是，aidl 所产生的代码完全可以由应用程序员手工编写，`IMusicPlayerService` 中的函数只是一种编码习惯而已，`asInterface` 即如此，提供这个函数的原因是服务端提供的服务除了其他进程可以使用外，在服务进程内部的其他类也可以使用，对于后者，显然是不需要经过 IPC 调用，而可以直接在进程内部调用的，而 Binder 内部有一个 `queryLocalInterface (String description)` 函数，该函数是根据输入的字符串判断该 Binder 对象是一个本地的 Binder 引用。在如图 5-1 所示中曾经指出，当创建一个 Binder 对象时，服务端进程内部创建一个 Binder 对象，Binder 驱动中也会创建一个 Binder 对象。如果从远程获取服务端的 Binder，则只会返回 Binder 驱动中的 Binder 对象，而如果从服务端进程内部获取 Binder 对象，则会获取服务端本身的 Binder 对象。听起来有点复杂，请看图 5-4 示意。



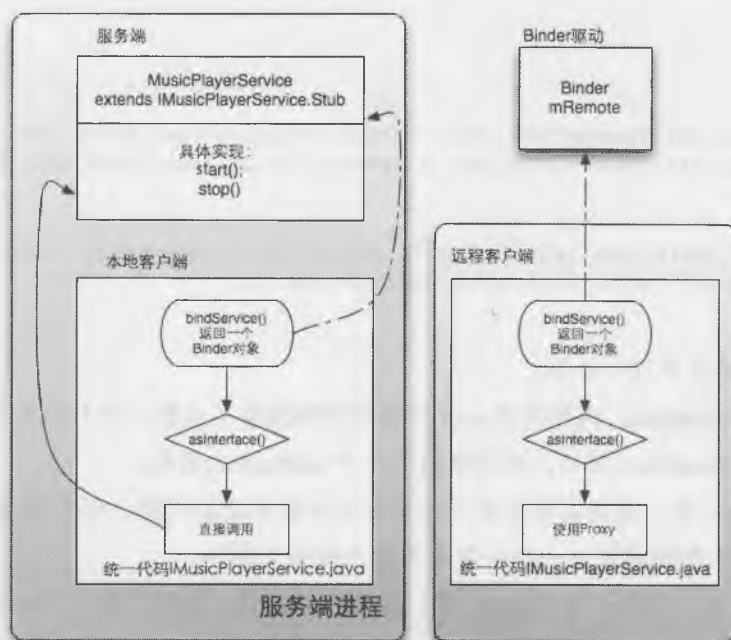


图 5-4 MusicPlayerService 服务中的代码结构

因此，asInterface()函数正是利用了 queryLocalInterface()方法，提供了一个统一的接口。无论是远程客户端还是本地端，当获取 Binder 对象后，可以把获取的 Binder 对象作为 asInterface()的参数，从而返回一个 IMusicPlayerService 接口，该接口要么使用 Proxy 类，要么直接使用 Stub 所实现的相应服务函数。

## 5.5 系统服务中的 Binder 对象

在应用程序编程时，经常使用 getSystemService(String serviceName)方法获取一个系统服务，那么，这些系统服务的 Binder 引用是如何传递给客户端的呢？须知系统服务并不是通过 startService()启动的。

getSystemService()函数的实现是在 ContextImpl 类中，该函数所返回的 Service 比较多，具体可参照源码。这些 Service 一般都由 ServiceManager 管理。

### 5.5.1 ServiceManager 管理的服务

ServiceManager 是一个独立进程，其作用如名称所示，管理各种系统服务，管理的逻辑如图 5-5 所示。

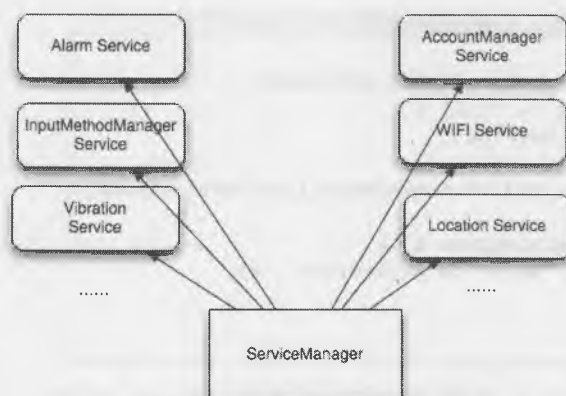


图 5-5 ServiceManager 所管理的服务列表

ServiceManager 本身也是一个 Service，Framework 提供了一个系统函数，可以获取该 Service 对应的 Binder 引用，那就是 `BinderInternal.getContextObject()`。该静态函数返回 `ServiceManager` 后，就可以通过 `ServiceManager` 提供的方法获取其他系统 Service 的 Binder 引用。这种设计模式在日常生活中到处可见，`ServiceManager` 就像是一个公司的总机，这个总机号码是公开的，系统中任何进程都可以使用 `BinderInternal.getContextObject()` 获取该总机的 Binder 对象，而当用户想联系公司中的其他人（服务）时，则要经过总机再获得分机号码。这种设计的好处是系统中仅暴露一个全局 Binder 引用，那就是 `ServiceManager`，而其他系统服务则可以隐藏起来，从而有助于系统服务的扩展，以及调用系统服务的安全检查。其他系统服务在启动时，首先把自己的 Binder 对象传递给 `ServiceManager`，即所谓的注册（`addService`）。

下面从代码实现来看以上逻辑。可以查看 `ContextImpl.getSystemService()` 中各种 Service 的具体获取方式，比如 `INPUT_METHOD_SERVICE`，代码如下：

```

922         } else if (INPUT_METHOD_SERVICE.equals(name)) {
923             return InputMethodManager.getInstance(this);
  
```

而 `InputMethodManager.getInstance (this)` 的关键代码如下：

```

458         synchronized (mInstanceSync) {
459             if (mInstance != null) {
460                 return mInstance;
461             }
462             IBinder b = ServiceManager.getService(Context.INPUT_METHOD_SERVICE);
463             IInputMethodManager service = IInputMethodManager.Stub.asInterface(b);
464             mInstance = new InputMethodManager(service, mainLooper);
465         }
466         return mInstance;
  
```

即通过 `ServiceManager` 获取 `InputMethod Service` 对应的 Binder 对象 `b`，然后再将该 Binder 对象作为 `IInputMethodManager.Stub.asInterface()` 的参数，返回一个 `IInputMethodManager` 的统一接口。

`ServiceManager.getService()` 的代码如下：

```

49 public static IBinder getService(String name) {
50     try {
51         IBinder service = sCache.get(name);
52         if (service != null) {
53             return service;
54         } else {
55             return getIServiceManager().getService(name);
56         }
57     } catch (RemoteException e) {
58         Log.e(TAG, "error in getService", e);
59     }
60     return null;
61 }

```

即首先从 sCache 缓存中查看是否有对应的 Binder 对象，有则返回，没有则调用 getIServiceManager().getService(name)，第一个函数 getIServiceManager() 即用于返回系统中唯一的 ServiceManager 对应的 Binder，其代码如下：

```

private static IServiceManager getIServiceManager() {
    if (sServiceManager != null) {
        return sServiceManager;
    }

    // Find the service manager
    sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
    return sServiceManager;
}

```

以上代码中，BinderInternal.getContextObject() 静态函数即用于返回 ServiceManager 对应的全局 Binder 对象，该函数不需要任何参数，因为它的作用是固定的。从这个角度来看，这个函数的命名似乎更应该明确一些，比如，可以命名为 getServiceManager()。

其他所有通过 ServiceManager 获取的系统服务的过程与以上基本类似，所不同的就是传递给 ServiceManager 的服务名称不同，因为 ServiceManager 正是按照服务的名称（String 类型）来保存不同的 Binder 对象的。

关于使用 addService() 向 ServiceManager 中添加一个服务一般是在 SystemService 进程启动时完成的，具体见第 9 章中关于 Framework 启动过程的描述。

## 5.5.2 理解 Manager

ServiceManager 所管理的所有 Service 都是以相应的 Manager 返回给客户端，因此，这里简述一下 Framework 中关于 Manager 的语义。在我们中国的企业里，Manager 一般指经理，比如项目经理、人事经理、部门经理。经理本身的含义比较模糊，其角色有些是给我们分配任务，比如项目经理；有些是给我们提供某种服务，比如人事经理；有些则是监督我们的工作等。而在 Android 中，Manager 的含义更应该翻译为经纪人，Manager 所 manage 的对象是服务本身，因为每个具体的服务一般都会提供多个 API 接口，而 Manager 所 manage 的正是这些 API。客户端一般不能直接通过 Binder 引用去访问具体的服务，

而是要经过一个 Manager，相应的 Manager 类对客户端是可见的，而远程的服务类对客户端则是隐藏的。

而这些 Manager 的类内部都会有一个远程服务 Binder 的变量，而且在一般情况下，这些 Manager 的构造函数参数中会包含这个 Binder 对象。简单地讲，即先通过 ServiceManager 获取远程服务的 Binder 引用，然后使用这个 Binder 引用构造一个客户端本地可以访问的经纪人，然后客户端就可以通过该经纪人访问远程的服务。

这种设计的作用是屏蔽直接访问远程服务，从而可以给应用程序提供灵活的、可控的 API 接口，比如 AmS。系统不希望用户直接去访问 AmS，而是经过 ActivityManager 类去访问，而 ActivityManager 内部提供了一些更具可操作性的数据结构，比如 RecentTaskInfo 数据类封装了最近访问过的 Task 列表，MemoryInfo 数据类封装了和内存相关的信息。

这种通过本地 Manager 访问远程服务的模型如图 5-6 所示。

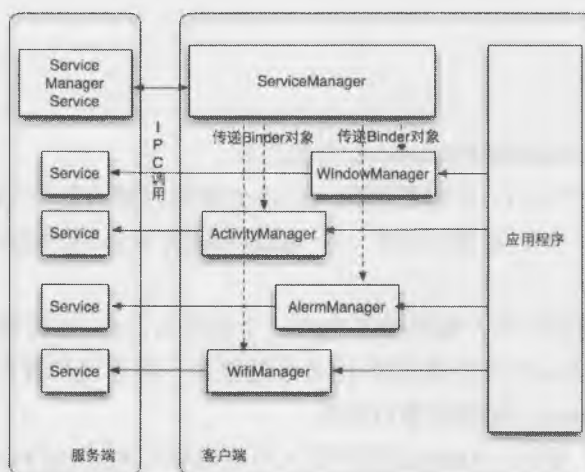


图 5-6 Manager 类和 Service 类之间的关系



## 第 6 章 Framework 概述

从本章开始，真正进入 Android Framework 内核之旅。

任何控制类程序都有一个入口，汇编程序的入口由处理器内部的复位（Reset）中断向量表决定；C 程序的入口是 main() 函数，一个 C 程序只能有一个 main() 函数；Java 程序的入口必须是某个类的静态成员函数 main()。

对于依赖于操作系统的程序，客户程序除了包含一个程序入口外，还需要和相关系统服务一起运行，以完成指定的任务。比如，Win32 程序需要和 GUI 系统服务一起实现带有可视窗口的功能；X Window 程序也需要和 X Window Server 一起实现窗口功能。

Android 程序也不例外，那么，Android 程序的入口在哪里？Android Framework 都包含哪些必需的系统服务？这些系统服务是如何与 Android APK 程序配合的？本章将回答这些问题。关于各组件内部具体实现过程将在后续各章中分别阐述。

### 6.1 Framework 框架

Framework 定义了客户端组件和服务端组件功能及接口。以下阐述中，“应用程序”一般是指“.apk”程序。

框架中包含三个主要部分，分别为服务端、客户端和 Linux 驱动。

#### 6.1.1 服务端

服务端主要包含两个重要类，分别是 WindowManagerService（WmS）和 ActivityManagerService

(AmS)。WmS 的作用是为所有的应用程序分配窗口，并管理这些窗口。包括分配窗口的大小，调节各窗口的叠放次序，隐藏或者显示窗口。AmS 的作用是管理所有应用程序中的 Activity。

除此之外，在服务端还包括两个消息处理类。

- KeyQ 类：该类为 WmS 的内部类，继承于 KeyInputQueue 类，KeyQ 对象一旦创建，就立即启动一个线程，该线程会不断地读取用户的 UI 操作消息，比如按键、触摸屏、trackball、鼠标等，并把这些消息放到一个消息队列 QueueEvent 类中。
- InputDispatcherThread 类：该类的对象一旦创建，也会立即启动一个线程，该线程会不断地从 QueueEvent 中取出用户消息，并进行一定的过滤，过滤后，再将这些消息发送给当前活动的客户端程序中。

### 6.1.2 客户端

客户端主要包括以下重要类。

- ActivityThread 类：该类为应用程序的主线程类，所有的 APK 程序都有且仅有一个 ActivityThread 类，程序的入口为该类中的 static main() 函数。
- Activity 类：该类为 APK 程序的一个最小运行单元，一个 APK 程序中可以包含多个 Activity 对象，ActivityThread 主类会根据用户操作选择运行哪个 Activity 对象。
- PhoneWindow 类：该类继承于 Window 类，同时，PhoneWindow 类内部包含了一个 DecorView 对象。简而言之，PhoneWindow 是把一个 FrameLayout 进行了一定的包装，并提供了一组通用的窗口操作接口。
- Window 类：该类提供了一组通用的窗口（Window）操作 API，这里的窗口仅仅是程序层面上的，WmS 所管理的窗口并不是 Window 类，而是一个 View 或者 ViewGroup 类，一般就是指 DecorView 类，即一个 DecorView 就是 WmS 所管理的一个窗口。Window 是一个 abstract 类型。
- DecorView 类：该类是一个 FrameLayout 的子类，并且是 PhoneWindow 中的一个内部类。Decor 的英文是 Decoration，即“修饰”的意思，DecorView 就是对普通的 FrameLayout 进行了一定的修饰，比如添加一个通用的 Title bar，并响应特定的按键消息等。
- ViewRoot 类：WmS 管理客户端窗口时，需要通知客户端进行某种操作，这些都是通过异步消息完成的，实现的方式就是使用 Handler，ViewRoot 就是继承于 Handler，其作用主要是接收 WmS 的通知。
- W 类：该类继承于 Binder，并且是 ViewRoot 的一个内部类。
- WindowManager 类：客户端要申请创建一个窗口，而具体创建窗口的任务是由 WmS 完成的，WindowManager 类就像是一个部门经理，谁有什么需求就告诉它，由它和 WmS 进行交互，客户端不能直接和 WmS 进行交互。

### 6.1.3 Linux 驱动

Linux 驱动和 Framework 相关的主要包含两部分，分别是 SurfaceFlinger (SF) 和 Binder。每一个窗口都对应一个 Surface，SF 驱动的作用是把各个 Surface 显示在同一个屏幕上。

Binder 驱动的作用是提供跨进程的消息传递。

## 6.2 APK 程序的运行过程

首先，ActivityThread 从 main() 函数中开始执行，调用 prepareMainLooper() 为 UI 线程创建一个消息队列(MessageQueue)。

然后创建一个 ActivityThread 对象，在 ActivityThread 的初始化代码中会创建一个 H (Handler) 对象和一个 ApplicationThread (Binder) 对象。其中 Binder 负责接收远程 AmS 的 IPC 调用，接收到调用后，则通过 Handler 把消息发送到消息队列，UI 主线程会异步地从消息队列中取出消息并执行相应操作，比如 start、stop、pause 等。

接着 UI 主线程调用 Looper.loop() 方法进入消息循环体，进入后就会不断地从消息队列中读取并处理消息。

当 ActivityThread 接收到 AmS 发送 start 某个 Activity 后，就会创建指定的 Activity 对象。Activity 又会创建 PhoneWindow 类→DecorView 类→创建相应的 View 或者 ViewGroup。创建完成后，Activity 需要把创建好的界面显示到屏幕上，于是调用 WindowManager 类，后者于是创建一个 ViewRoot 对象，该对象实际上创建了 ViewRoot 类和 W 类，创建 ViewRoot 对象后，WindowManager 再调用 WmS 提供的远程接口完成添加一个窗口并显示到屏幕上。

接下来，用户开始在程序界面上操作。KeyQ 线程不断把用户消息存储到 QueueEvent 队列中，InputDispatcherThread 线程逐个取出消息，然后调用 WmS 中的相应函数处理该消息。当 WmS 发现该消息属于客户端某个窗口时，就会调用相应窗口的 W 接口。

W 类是一个 Binder，负责接收 WmS 的 IPC 调用，并把调用消息传递给 ViewRoot，ViewRoot 再把消息传递给 UI 主线程 ActivityThread，ActivityThread 解析该消息并做相应的处理。在客户端程序中，首先处理消息的是 DecorView，如果 DecorView 不想处理某个消息，则可以将该消息传递给它内部包含的子 View 或者 ViewGroup，如果还没有处理，则传递给 PhoneWindow，最后再传递给 Activity。

## 6.3 客户端中的线程

在多任务操作系统中，任何程序都运行在线程之中。系统首先会为客户端程序分配一个线程，然后该线程从程序的入口处开始执行。那么，请思考以下问题。

- Android APK 程序中都有哪些线程？



- 什么是 UI 线程？
- 程序中自定义 Thread 和 UI 线程的区别是什么？

首先，很明确地讲，包含有 Activity 的客户端程序至少包含三个线程，如图 6-1 所示。每个 Binder 对象都对应一个线程，Activity 启动后会创建一个 ViewRoot.W 对象，同时 ActivityThread 会创建一个 ApplicationThread 对象，这两个对象都继承于 Binder，因此会启动两个线程，负责接收 Linux Binder 驱动发送 IPC 调用。最后一个主要线程也就是程序本身所在的线程，也叫做用户交互（UI）线程，因为所有的处理用户消息，以及绘制界面的工作都在该线程中完成。

为了验证这一点，可以在 Eclipse 中新建一个 Hello Android 的程序，然后以 debug 的方式运行，在 debug 窗口中会看到如图 6-1 所示的界面。

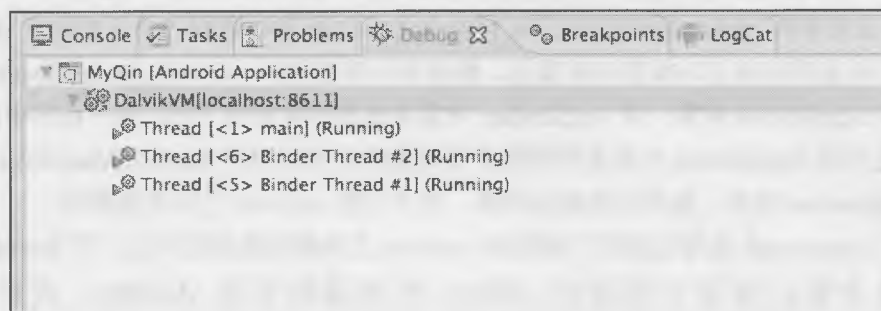


图 6-1 程序中线程

自定义 Thread 和 UI 线程的区别在于，UI 线程是从 ActivityThread 运行的，在该类中的 main() 方法中，已经使用 Looper.prepareMainLooper() 为该线程添加了 Looper 对象，即已经为该线程创建了消息队列（MessageQueue），因此，程序员才可以在 Activity 中定义 Handler 对象（因为声明 Handler 对象时，所在的线程必须已经创建了 MessageQueue）。而普通的自定义 Thread 是一个裸线程，因此，不能直接在 Thread 中定义 Handler 对象，从使用场景的角度讲，即不能直接给 Thread 对象发消息，但是却可以给 UI 线程发消息。

## 6.4 几个常见问题

在过去的实践中，经常有同学问到以下几个问题，故特意将此作为一节。

### 6.4.1 Activity 之间如何传递消息（数据）

首先，提出这个问题的原因是，程序员需要在不同的 Activity 之间传递数据，然而，这个问题本身就有问题。所谓“传递消息”一般是指多个线程之间，而 Activity 本身并不是线程，ActivityThread 才



是一个线程，即 UI 线程。同一个程序中的多个 Activity 都由 ActivityThread 进行调用，Activity 本身只是一个 Java 类而已，就像 Rect、Trigle 类一样，如果有人问“Rect 类和 Trigle 类之间如何传递消息”，你会不会觉得有点奇怪？

事实上，如果要在两个类中传递数据，方法可以有很多。

方法一：可以先实例化某个类，获得该类的引用，当其他类需要该对象的内部数据时，可以直接通过该引用去访问该类的内部数据。

方法二：对于 A、B 两个类之间，可以先实例化一个第三方类 C，然后两个类都可以把需要传递的数据存入 C 中，或从 C 中取出。

这些方法理论上都可以用在 Activity 类之间传递数据。然而，与普通类传递数据有所不同，普通类的实例化都是程序员显式完成的，而 Activity 类的实例化却是由 Framework 完成的，程序员只能使用 startActivity() 方法来告诉 Framework 去运行哪个 Activity，这就意味着程序员不能得到 Activity 对象的引用，那么就不能直接访问该对象的内部数据。解决的办法是使用 Activity.getApplication() 函数，该函数能够返回一个 Application 对象，该 Application 对象在该程序中是唯一的，同一程序中的不同 Activity 调用该函数所返回的 Application 对象是相同的，该对象的名称可以在 AndroidManifest.xml 中指定。一旦获取了该 Application 对象，就可以借助该对象，在不同的 Activity 之间传递数据。

除此之外，Framework 本身也提供了标准的 Activity 之间传递数据的方法，即 Intent 类。该类作为 startActivity() 的参数，仅用于在启动 Activity 时传递给目标 Activity，同时，如果调用 startActivityForResult()，目标 Activity 在结束后，也会返回一个 Intent 对象给原 Activity。

另外，从设计理念的角度来看，Android 认为，两个 Activity 如果要共享数据，可以通过 Preference Storage 或者文件、数据库进行，同时，在一般情况下，设备上只会有一个 Activity 在运行，因此，多个 Activity 之间传递数据也不是必需的。如果某个 Activity 需要在停止后还能处理某些数据，那么，该 Activity 似乎更应该被设计为一个后台的 Thread 或者一个 Service，无论是 Thread 还是 Service 都很容易获得其引用。

### 6.4.2 窗口相关的概念

在源码和本书中，会经常提到以下概念，窗口、Window 类、ViewRoot 类以及 W 类，本节简单介绍这些概念的联系和区别。

首先澄清几个概念。

- 窗口 (Window)：这是一个纯语义的说法，即程序员所看到的屏幕上的某个独立的界面，比如一个带有 Title Bar 的 Activity 界面、一个对话框、一个 Menu 菜单等，这些都称之为窗口。本书中所说的窗口管理一般也都泛指所有这些窗口，在 Android 的英文相关文章中则直接使用 Window 这个单词。而从 WmS 的角度来讲，窗口是接收用户消息的最小单元，WmS 内部用特

定的类表示一个窗口，以实现对流口的管理。WmS 接收到用户消息后，首先要判断这个消息属于哪个窗口，然后通过 IPC 调用把这个消息传递给客户端的 ViewRoot 类。

- Window 类：该类在 android.view 包中，是一个 abstract 类，该类是对包含有可视界面的窗口的一种包装。所谓的可视界面就是指各种 View 或者 ViewGroup，一般可以通过 res/layout 目录下的 xml 文件描述。
- ViewRoot 类：该类在 android.view 包中，客户端申请创建窗口时需要一个客户端代理，用以和 WmS 进行交互，这个就是 ViewRoot 的功能，每个客户端的窗口都会对应一个 ViewRoot 类。
- W 类：该类是 ViewRoot 类的一个内部类，继承于 Binder，用于向 WmS 提供一个 IPC 接口，从而让 WmS 控制窗口客户端的行为。

描述一个窗口之所以使用这么多类的原因在于，窗口的概念存在于客户端和服务端（WmS）之中，客户端所理解的窗口和服务端理解的窗口是不同的，因此，在客户端和服务端会用不同的类来描述窗口。同时，无论是在客户端还是服务端，对窗口都有不同层面的抽象，比如在客户端，用户能看到的窗口一般是 View 或者 ViewGroup 组成的窗口，而与 Activity 对应的窗口却是一个 DecorView 类，而具备常规 Phone 操作接口的窗口却又是一个 PhoneWindow 类。



## 第 7 章 理解 Context

Context 在应用程序开发中会经常被使用，在一般的计算机书籍中，Context 被翻译为“上下文”，而笔者认为 Android 中的 Context 应该被翻译为“场景”。

### 7.1 Context 是什么

一个 Context 意味着一个场景，一个场景就是用户和操作系统交互的一种过程。比如当你打电话时，场景包括电话程序对应的界面，以及隐藏在界面后的数据；当你看短信时，场景包括短信界面，以及隐藏在后面的数据。

读者可曾想过，从程序的角度来看，Context 到底是什么？如果笔者告诉你，一个 Activity 就是一个 Context，一个 Service 也是一个 Context，你会觉得奇怪吗？

我们先不看代码，而从语义的角度来审视一下 Context。Android 程序员把“场景”抽象为 Context 类，如上所述，他们认为用户和操作系统的每一次交互都是一个场景，比如打电话、发短信，这些都是有界面的场景，还有一些没有界面的场景，比如后台运行的服务（Service）。一个应用程序可以认为是一个工作环境，用户在这个工作环境中会切换到不同的场景，这就像一个前台秘书，她可能需要接待客人，可能要打印文件，还可能要接听客户电话，而这些就称之为不同的场景，前台秘书可称之为一个应用程序。

接下来看代码。Activity 类的确是基于 Context，而 Service 类也是基于 Context。Activity 除了基于 Context 类外，还实现了一些其他重要接口，从设计的角度来看，interface 仅仅是某些功能，而 extends 才是类的本质，即 Activity 的本质是一个 Context，其所实现的其他接口只是为了扩充 Context 的功能而已，扩充后的类称之为一个 Activity 或者 Service。

也可以从另一个角度来理解 Context 和 Activity 的关系。

大家都知道 Activity 内部包含一些特别的方法，比如 onCreate()、onPause()、onStart()等方法，这些是只有 Activity 才有的。那么试想一下，假如 Activity 类是一个 interface 的话，那么就可以提出一种真正意义上的场景类，假设命名为 Task，同时假设 Context 类不是一个 abstract 类，而是一个 interface，那么 Task 类的定义将会像下面这个样子。

```
class Task implements Activity, Context, ...
```

可事实上，由于 Context 类就是为“场景应用”而专门设计的，它包含了一个场景中的基本函数接口，因此，Google 程序员认为 Task 本质上就是 Context，Activity 尽管有一些特别的函数接口，但本质上就是一个 Context。所以，Context 被设计为一个 abstract 类，Activity 仅仅是基于 Context 而已。Google 程序员还认为 Service 本质上是一个 Context，所以 Service 也是仅仅基于 Context 而已。

## 7.2 一个应用程序中包含多少个 Context 对象

在以往的应用程序开发中，经常会调用 Context 的一些方法，这些方法看起来似乎会返回一些全局的对象，而不仅仅是某个 Activity，于是有的读者就有点疑惑，一个应用程序到底有多少个 Context 对象呢？比如，Context.getResources()返回该应用程序所对应的 Resources 类对象，无论从哪个 Activity 中调用，都会返回同一个 Resources 对象。

这里可以明确的是：

- 一个 Activity 就是一个场景（Context），一个 Service 也是一个场景，所以，应用程序中有多少个 Activity 或者 Service，就会有多少个 Context 对象。
- getResources()等方法的确返回的是同一个全局对象。至于这是如何实现的，请看下一节。

## 7.3 Context 相关类的继承关系

Context 相关的类的继承关系如图 7-1 所示。

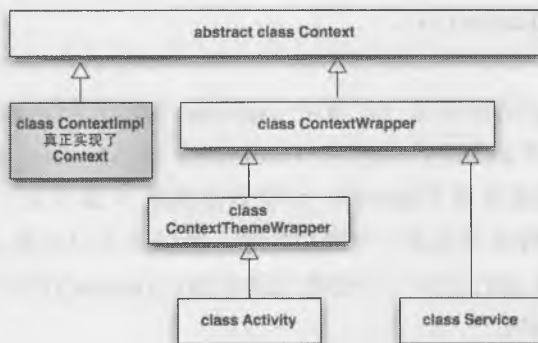


图 7-1 Context 类及其子类的继承关系

Context 类本身是一个纯 abstract 类。

为了方便使用，又定义了 ContextWrapper 类，如其名所言，这只是一个包装而已，ContextWrapper 构造函数中必须包含一个真正的 Context 引用，同时 ContextWrapper 中提供了 attachBaseContext() 用于给 ContextWrapper 对象中指定真正的 Context 对象，调用 ContextWrapper 的方法都会被转向其所包含的真正的 Context 对象。

ContextThemeWrapper 类，如其名称所示，其内部包含了与主题 (Theme) 相关的接口，这里所说的主题就是指在 AndroidManifest.xml 中通过 android:theme 为 Application 元素或者 Activity 元素指定的主题。

当然，只有 Activity 才需要主题，Service 是不需要主题的，因为 Service 是没有界面的后台场景，所以 Service 直接继承于 ContextWrapper。

ContextImpl 类真正实现了 Context 中所有的函数，应用程序中所调用的各种 Context 类的方法，其实现均来自于该类。

## 7.4 创建 Context

前面说每一个 Activity 本质上就是一个 Context，因为 Activity 就是基于 ContextWrapper 类，然而对于 ContextWrapper 类，必须为其指定真正的 Context 对象，而真正实现了 Context 类的是 ContextImpl 类。所以本节就来研究这个“真正的 Context” (ContextImpl) 是如何创建的。

每一个应用程序在客户端都是从 ActivityThread 类开始的，创建 Context 对象也是在该类中完成，具体创建 ContextImpl 类的地方一共有 7 处，分别如下：

- 在 PackageInfo.makeApplication() 中。
- 在 performLaunchActivity() 中。
- 在 handleCreateBackupAgent() 中。
- 在 handleCreateService() 中。
- 在 handleBindApplication() 中。
- 还是在 handleBindApplication() 中。
- 在 attach() 方法中。

关于以上方法的调用时机请参照第 10 章中 Activity 启动过程，本节主要介绍以上方法中创建 ContextImpl 的内部逻辑。其中 attach() 方法仅在 Framework 进程(system\_server)启动时调用，应用程序运行时不会调用到该方法，因此本章不做介绍，详情请参照第 9 章中关于 Framework 的启动。需要再次明确的是 system\_server 进程本身也是一个应用程序，所以其入口也是 ActivityThread 类，只是这个 ActivityThread 和一些系统服务运行在同一个进程空间中而已。attach() 方法中创建 ContextImpl 对象的逻辑与下面三节所描述的基本相同。

### 7.4.1 Application 对应的 Context

每个应用程序在第一次启动时，都会首先创建一个 Application 对象，默认的 Application 是应用程序的包名，用户可以重载默认的 Application。方法是在 AndroidManifest.xml 的 Application 标签中声明一个新的 Application 名称，然后在源码中添加该名称的类，该类的父类使用 Application 类即可。

程序第一次启动时，会辗转调用到 handleBindApplication() 方法中，该方法中有两处创建了 ContextImpl 对象，但都是在 if(data.instrumentationName != null) 条件中。该条件在一般的程序执行时都不会被执行到，只有当创建了一个 Android Unit Test 工程时，相应的 Test 程序会满足这个条件。关于 Android Unit Test 请参照 Android SDK 文档，此处不做介绍。

而如果不是测试工程的话，则调用 makeApplication() 方法，如以下代码所示：

```
4232 Application app = data.info.makeApplication(data)
```

而在 makeApplication() 方法中，主要包含以下代码：

```
642 ContextImpl appContext = new ContextImpl();
643 appContext.init(this, null, mActivityThread);
644 app = mActivityThread.mInstrumentation.newApp
645     cl, appClass, appContext);
646 appContext.setOuterContext(app);
```

即创建一个 ContextImpl 对象，然后调用 init() 方法，其中第一个参数 this 指的就是当前 PackageInfo 对象，该对象将赋值给 ContextImpl 类中的重要成员变量 mPackageInfo。

这里需要注意，这个 PackageInfo 对象又是从何而来呢？这就要回溯到是谁调用了 makeApplication() 方法，回溯流程如下。

首先，AmS 通过远程调用到 ActivityThread 的 bindApplication() 方法，该方法的参数包括两种。一种是 ApplicationInfo，这是实现了 Parcelable 接口的数据类，意味着这个对象是由 AmS 创建的，通过 IPC 传递到 ActivityThread 中，另一种是其他相关参数。

在 bindApplication() 方法中，会用以上两种参数构造一个本地 AppBindData 数据类，然后再去调用 handleBindApplication()。

在调用 handleBindApplication() 的时候，AppBindData 对象中的 info 变量还是空值，然后会使用 data.info = getPackageInfoNoChecked() 方法为 info 变量赋值，而这个方法的内部实际上会根据 AppBindData 中的 ApplicationInfo 中的 mPackageName 创建一个 PackageInfo 对象，并把这个对象保存为 ActivityThread 类的全局对象。显然，一个应用程序中的所有 Activity 或者 Application 对象对应的 mPackageName 都是一样的，即为包的名称，所以 ActivityThread 中会产生一个全局 PackageInfo 对象。

接下来，就回到上面所说的调用 data.info.makeApplication() 方法了，这就是 PackageInfo 对象的来源。

以上流程可总结为如图 7-2 所示。

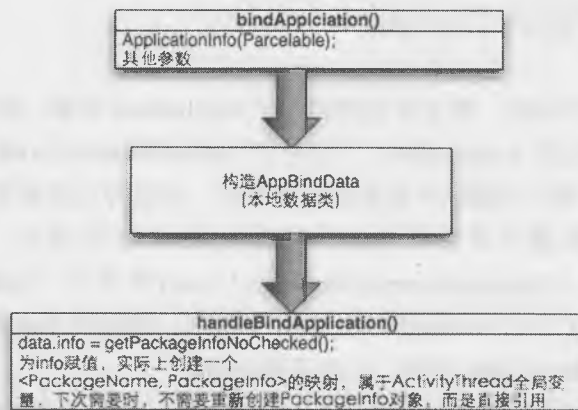


图 7-2 创建 Context 时 PackageInfo 对象的来源

## 7.4.2 Activity 对应的 Context

启动 Activity 时, AmS 会通过 IPC 调用到 ActivityThread 的 `scheduleLaunchActivity()` 方法, 该方法包含两种参数。一种是 `ActivityInfo`, 这是一个实现了 `Parcelable` 接口的数据类, 意味着该对象是 AmS 创建的, 并通过 IPC 传递到 `ActivityThread`; 另一种是其他一些参数。

`scheduleLaunchActivity()` 方法中会根据以上两种参数构造一个本地 `ActivityRecord` 数据类, `ActivityThread` 内部会为每一个 Activity 创建一个 `ActivityRecord` 对象, 并使用这些数据对象来管理 Activity。

接着会调用到 `handleLaunchActivity()`, 然后再调用到 `performLaunchActivity()`, 该方法中创建 `ContextImpl` 的代码如下:

```

2603 ContextImpl appContext = new ContextImpl();
2604 appContext.init(r.packageInfo, r.token, this);
2605 appContext.setOuterContext(activity);
  
```

这段代码与上一节基本相同。同样要问到一个问题, `appContext.init()` 方法中第一个参数 `r.packageInfo` 是如何而来的, 答案如以下代码所示:

```

2556 ActivityInfo aInfo = r.activityInfo;
2557 if (r.packageInfo == null) {
2558     r.packageInfo = getPackageInfo(aInfo.applicationInfo,
2559                                     Context.CONTEXT_INCLUDE_CODE);
2560 }
  
```

即在 `performLaunchActivity()` 开始执行时, 首先为 `r.packageInfo` 变量赋值, `getPackageInfo` 方法的执行逻辑和 `getPackageInfoNoChecked()` 基本相同。所以, `r.packageInfo` 对象的 `PackageInfo` 对象和 `Application` 对应的 `packageInfo` 对象是同一个。



以上流程可总结为如图 7-3 所示。

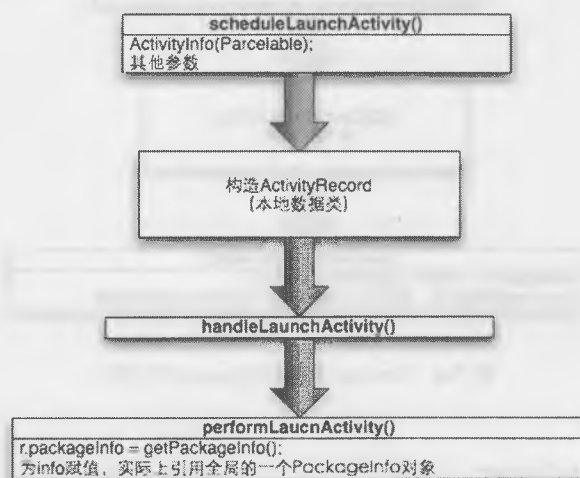


图 7-3 Activity 中创建 Context 的过程

### 7.4.3 Service 对应的 Context

启动 Service 时, AmS 首先会通过 IPC 调用到 ActivityThread 的 scheduleCreateService()方法, 该方法也包含两种参数。第一种是 ServiceInfo, 这是实现了一个 Parcelable 接口的数据类, 意味着该对象由 AmS 创建, 并通过 IPC 传递到 ActivityThread 内部; 第二种是其他参数。

在 scheduleCreateService()方法中, 会使用以上两种参数构造一个 CreateServiceData 的数据对象, ActivityThread 会为其所包含的每一个 Service 创建该数据对象, 并通过这些对象来管理 Service。

接着, 会执行 handleCreateService()方法, 其中创建 ContextImpl 对象的代码如下:

```

2952 ContextImpl context = new ContextImpl();
2953 context.init(packageInfo, null, this);
2954 |
2955 Application app = packageInfo.makeApplication(false,
2956 context.setOuterContext(service);
  
```

这与前两节基本相同, 调用 context.init()方法的第一个参数 packageInfo 赋值的代码如下:

```

2935 PackageInfo packageInfo = getPackageInfoNoCheck(
2936 data.info.applicationInfo);
  
```

赋值代码同样使用了 getPackageInfoNoCheck()方法, 这就意味着 Service 对应的 Context 对象内部的 mPackageInfo 与 Activity、Application 中是完全相同的。

其流程总结为如图 7-4 所示。



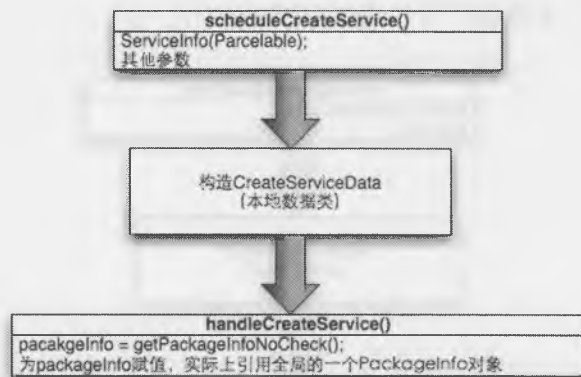


图 7-4 Service 中创建 Context 的过程

#### 7.4.4 Context 之间的关系

从以上三节可以看出, 创建 Context 对象的过程基本上是相同的, 包括代码的结构也十分类似, 所不同的仅仅是针对 Application、Activity、Service 使用了不同的数据对象, 可总结为如表 7-1 所示。

表 7-1 不同 Context 子类中 PackageInfo 对象的来源

类名	远程数据类	本地数据类	赋值方式
Application	ApplicationInfo	AppBindData	getPackageInfoNoCheck()
Activity	ActivityInfo	ActivityRecord	getPackageInfo()
Service	ServiceInfo	CreateServiceData	getPackageInfoNoCheck()

从以上三节也可以看出, 实际上一个应用程序包含的个数应该为:

Context 个数 = Service 个数 + Activity 个数 + 1

最后一个 1 代表的是 Application 类本身也会对应一个 Context 对象, 而 Context 对象的结构关系可如图 7-5 所示。

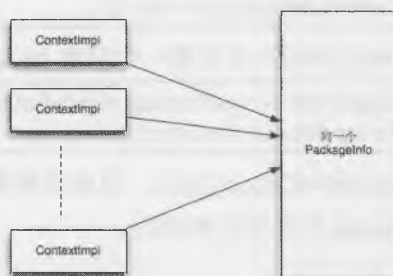


图 7-5 ContextImpl 和 PackageInfo 对象的对应关系

应用程序中包含多个 ContextImpl 对象，而其内部变量 mPackageInfo 却指向同一个 PackageInfo 对象，这种设计结构一般意味着 ContextImpl 是一种轻量级类，而 PackageInfo 是一个重量级类，那么事实是不是这样呢？通过查看 ContextImpl 代码可知，的确是这样，ContextImpl 中的大多数重量级函数实际上都是转向了 mPackageInfo 对象相应的方法，即事实上是调用了同一个 PackageInfo 对象，因此，从系统效率的角度来看也是合理的。



## 第 8 章 创建窗口的过程

从 WmS 的角度来看，一个窗口并不是 Window 类，而是一个 View 类。WmS 收到用户消息后，需要把消息派发到窗口，View 类本身并不能直接接收 WmS 传递过来的消息，真正接收用户消息的必须是 IWindow 类，而实现 IWindow 类的是 ViewRoot.W 类，每一个 W 内部都包含了一个 View 变量。

WmS 并不介意该窗口（View）是属于哪个应用程序的，WmS 会按一定的规则判断哪个窗口处于活动状态，然后把用户消息给 W 类，W 类再把用户消息传递给内部的 View 变量，剩下的消息处理就由 View 对象完成。

### 8.1 窗口的类型

Framework 定义了三种窗口类型，三种类型的定义在 WindowManager 类中。

- 第一种为应用窗口。所谓的应用窗口是指该窗口对应一个 Activity，由于加载 Activity 是由 AmS 完成的，因此，对于应用程序来讲，要创建一个应用类窗口，只能在 Activity 内部完成。
- 第二种是子窗口。所谓的子窗口是指，该窗口必须有一个父窗口，父窗口可以是一个应用类型窗口，也可以是任何其他类型的窗口。
- 第三类是系统窗口。系统窗口不需要对应任何 Activity，也不需要父窗口。对于应用程序而言，理论上是无法创建系统窗口的，因为所有的应用程序都没有这个权限，然而系统进程却可以创建系统窗口。

WindowManager 类对这三种类型进行了细化，把每一种类型都用一个 int 常量表示，这些实际上代表了窗口对应的层（Layer）。WmS 在进行窗口叠加时，会按照该 int 常量的大小分配不同层，int 值越大，代表层的位置越靠上面，即所谓的 z-order。

类型一的定义如表 8-1 所示。

表 8-1 应用窗口类型

定 义	意 义
FIRST_APPLICATION_WINDOW = 1;	第一个应用窗口
TYPE_BASE_APPLICATION = 1;	第一个应用窗口
TYPE_APPLICATION = 2;	所有 Activity 对应的窗口
TYPE_APPLICATION_STARTING = 3;	Activity 启动时, 可以指定一个启动窗口首先显示, 直到真正的 Activity 窗口配置好后, 删除该启动窗口, 显示 Activity 窗口
LAST_APPLICATION_WINDOW = 99	最后一个应用窗口

所有的 Activity 默认的窗口类型都是 TYPE\_APPLICATION, WmS 在进行窗口叠加时, 会动态改变应用窗口的层值, 但层值不会大于 99。

类型二为子窗口, 如表 8-2 所示。

表 8-2 子窗口类型

定 义	意 义
FIRST_SUB_WINDOW(F) = 1000	第一个子窗口
TYPE_APPLICATION_PANEL = F	应用窗口的子窗口, PopupWindow 的默认类型
TYPE_APPLICATION_MEDIA = F + 1	尚未使用
TYPE_APPLICATION_SUB_PANEL = F + 2	尚未使用
TYPE_APPLICATION_ATTACHED_DIALOG = F + 3	OptionsMenu、ContextMenu 的默认类型
TYPE_APPLICATION_MEDIA_OVERLAY = F + 4	尚未使用
LAST_SUB_WINDOW = 1999	最后一个子窗口

同样, 创建子窗口时, 客户端可以指定窗口类型介于 1000~1999 之间, 而 WmS 在进行窗口叠加时, 会动态调整层值。

类型三为系统窗口, 如表 8-3 所示。

表 8-3 系统窗口类型

定 义	意 义	定 义	意 义
FIRST_SYSTEM_WINDOW(F) = 2000	第一个系统窗口	TYPE_SYSTEM_DIALOG	似乎等同于滑动状态条后出现的窗口
TYPE_STATUS_BAR	状态条窗口	TYPE_KEYGUARD_DIALOG	屏保弹出的对话框
TYPE_SEARCH_BAR =	搜索条窗口	TYPE_SYSTEM_ERROR	系统错误窗口
TYPE_PHONE =	来电显示窗口	TYPE_INPUT_METHOD	输入法窗口
TYPE_SYSTEM_ALERT	警告对话框	TYPE_INPUT_METHOD_DIALOG	输入法中各选框对应的窗口
TYPE_KEYGUARD	屏保	TYPE_WALLPAPER	墙纸对应的窗口
TYPE_TOAST	Toast 对应的窗口	TYPE_STATUS_BAR_PANEL	滑动状态条后出现的窗口
TYPE_SYSTEM_OVERLAY =	尚未使用	LAST_SYSTEM_WINDOW = 2999	最后一个系统窗口
TYPE_PRIORITY_PHONE	在屏幕保护下的来电显示窗口		

同样，当具备创建系统窗口权限时，创建系统窗口可以指定层值在 2000~2999 之间，WmS 在进行窗口叠加时，会动态调整该层值。所不同的是，由于有些系统窗口只能出现一个，即不能添加多个，否则用户会觉得很乱，比如输入法窗口，再比如系统状态条窗口，因此，WmS 在接收到创建窗口的消息时，会进行一定的检查，确保该窗口只能被创建一次。

## 8.2 token 变量的含义

阅读本节时，如果觉得不理解，可先跳过，继续看后面几节，看完后再回过来看。

token 在英语中的含义为象征、符号、代表等。在创建窗口时，多处定义了和 token 相关的变量，而无论该变量具体的名称是什么，该变量的类型一般都是一个 IBinder 对象。既然是 IBinder 对象，其作用也就是显而易见的，即为了进行 IPC 调用。而与创建窗口相关的 IPC 对象一般只有两种，一种是指向某个 W 类的 token，另一种是指向某个 HistoryRecord 的 token。其中 HistoryRecord 对象是 AmS 内部为运行的每一个 Activity 创建的一个 Binder 对象，客户端的 Activity 可以通过该 Binder 对象通知当前 Activity 的状态。

具体来讲，token 的定义出现在以下几处，如表 8-4 所示。

表 8-4 token 相关的源码

位 置	定 义
Activity	IBinder mToken
Window	IBinder mAppToken
WindowManager.LayoutParams	IBinder token
ViewRoot	View.AttachInfo mAttachInfo;
View	View.AttachInfo mAttachInfo;
View.AttachInfo	IBinder mWindowToken; IBinder mPanelParentWindowToken; IWindow mWindow;

后面的章节将对表 8-4 中的定义进行解释。

### 8.2.1 Activity 中的 mToken

如上所述，AmS 内部为每一个运行的 Activity 都创建了一个 HistoryRecord 对象，该对象的基类是一个 Binder，因此 mToken 变量的意义正是指向了该 HistoryRecord 类。该变量的值是在 Activity.init() 函数中完成的，其流程如图 8-1 所示。

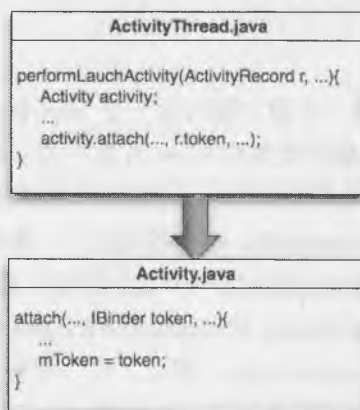


图 8-1 初始化 Activity 类中 mToken 的过程

### 8.2.2 Window 中的 mAppToken

每一个 Window 对象中都有一个 mAppToken 变量，注意这里说的是 Window 对象，而不是窗口。前面说过，一个窗口本质上是一个 View，而 Window 类却是一个应用窗口的抽象，这就好比 Window 侧重于一个窗口的交互，而窗口（View）则侧重于窗口的显示。所以，mAppToken 并不是 W 类的引用，事实上正如其名称所指，它是 AmS 在远程为每一个 Activity 创建的 HistoryRecord 的引用。

事实上，Window 类中还有其他的 Binder 对象，同时由于 Window 并不一定要对应一个 Activity，因此，如果 Window 类不属于某个 Activity，mAppToken 的变量则为空，否则 mAppToken 的值与 Activity 中的 mToken 值是相同的。

### 8.2.3 WindowManager.LayoutParams 中的 token

WindowManager.LayoutParams 中 token 的意义正如其所在的类的名称，该类是在添加窗口时指定该窗口的布局参数，而 token 的意义正是指定该窗口对应的 Binder 对象，因为 WmS 需要该 Binder 对象，以便对客户端进行 IPC 调用。

那么，该 Binder 对象应该谁呢？你可能会认为该 token 应该是 W 类，没错，这只是一种情况而已，具体来讲，该 token 变量的值可以有三种。

- 如果创建的窗口是应用窗口，token 的值和 Window 中 mAppToken 值相同。
- 如果创建的窗口为子窗口，token 为其父窗口的 W 对象。
- 如果创建的窗口是系统窗口，那么，token 值为空。

## 8.2.4 View 中的 token

首先来看 ViewRoot，客户端的每一个窗口都对应一个 ViewRoot 对象，该对象内部的 mAttachInfo 是该对象被构造时同时创建的。该变量的类型和 View 对象中的 mAttachInfo 相同。

View 类中的 mAttachInfo，其含义是当该 View 对象被真正作为某个窗口 W 类的内部 View 时，该变量就会被赋值为 ViewRoot 中的 mAttachInfo。在一般情况下，屏幕上所有的 View 对象的 mAttachInfo 都是被赋值过的，因为当 W 类中的 View 被添加为一个真正的窗口后，ViewRoot 会调用 performTraversal() 方法，而该方法则会调用 View 或者 ViewGroup 的 dispatchToWindow() 方法。在后者调用中，会把 ViewRoot 中的 mAttachInfo 赋值给 View 中的 mAttachInfo，所以，同一个窗口中包含的所有 View 对象，其内部的 mAttachInfo 的内容都是相同的。该变量的赋值过程如图 8-2 所示。

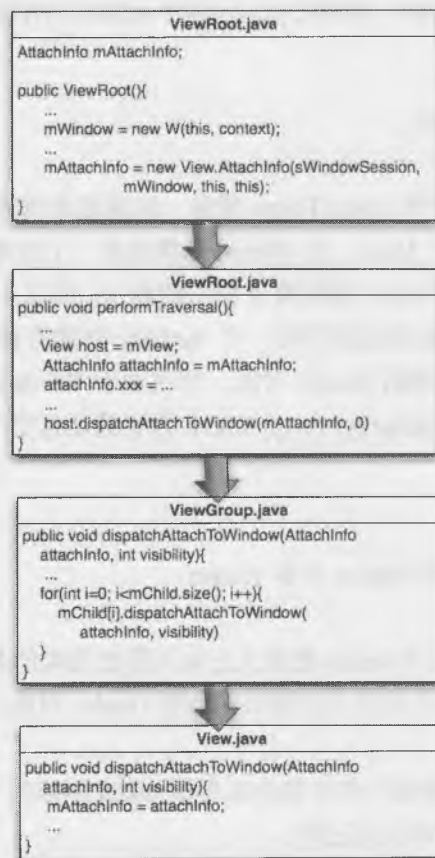


图 8-2 View 对象中 token 的赋值过程

mAttachInfo 变量中，包含了三个 Binder 变量。

- mWindowToken，如其名称所示，指的是该窗口对应的 W 对象。

- `mPanelParentWindowToken`，如果该窗口是子窗口的，那么该变量即为父窗口中的 `W` 对象。该变量赋值和 `mWindowToken` 是互斥的，因为 `mWindowToken` 如果不为空，则意味着该窗口没有父窗口。
- `mWindow`，注意，尽管该变量也是一个 `Binder` 对象，但它却更是一个 `IWindow` 对象，关于 `IBinder` 与 `IWindow` 的区别请参照第5章中关于 `Binder` 的介绍。由此也可以看出，`mWindowToken` 似乎是多余的，因为 `mWindowToken=IWindow.asBinder()`，但也无所谓，只是多了一个变量而已，使用起来更方便。

以上所有 token 的关系如图 8-3 所示。

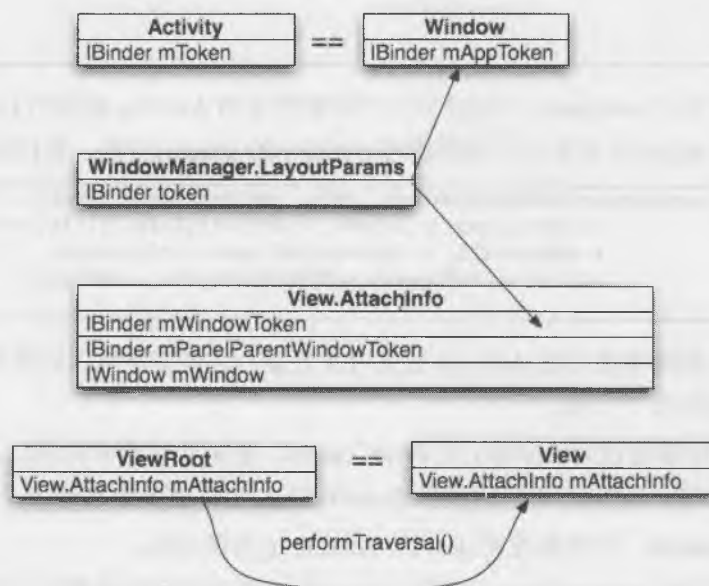


图 8-3 各 token 变量之间的关系

### 8.3 创建应用窗口

以上了解了不同类型窗口的含义，接下来介绍 Framework 是如何创建这些窗口的。首先来看应用窗口的创建。其总体流程如附图 2 所示，下面结合该图中的步骤逐一介绍。

① 每个应用类窗口都对应一个 `Activity` 对象，因此，创建应用类窗口首先需要创建一个 `Activity` 对象。当 `AmS` 决定启动某个 `Activity` 时，会通知客户端进程，而每个客户端进程都对应一个 `ActivityThread` 类，任何 `Activity` 都必须隶属于一个应用程序，因此，启动 `Activity` 的任务最终由 `ActivityThread` 完成。

启动某个 `Activity` 的代码本质是构造一个 `Activity` 对象，其代码如下所示：



```

2574     Activity activity = null;
2575     try {
2576         java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
2577         activity = mInstrumentation.newActivity(
2578             cl, component.getClassName(), r.intent);
2579         r.intent.setExtrasClassLoader(cl);
2580         if (r.state != null) {
2581             r.state.setClassLoader(cl);
2582         }
2583     } catch (Exception e) {
2584         if (!mInstrumentation.onException(activity, e)) {
2585             throw new RuntimeException(
2586                 "Unable to instantiate activity " + component
2587                 + ": " + e.toString(), e);
2588         }
2589     }

```

在以上代码中，使用 ClassLoader 从程序文件中装载指定的 Activity 对应的 Class 文件。

② 构造好指定的 Activity 对象后，接着调用 Activity 的 attach() 方法，其代码如下：

```

2610         activity.attach(appContext, this, getInstrumentation(), r.token,
2611             r.ident, app, r.intent, r.activityInfo, title, r.parent,
2612             r.embeddedID, r.lastNonConfigurationInstance,
2613             r.lastNonConfigurationChildInstances, config);
2614     }

```

attach() 的作用是为刚刚构造好的 Activity 设置内部变量，这些变量是以后进行 Activity 调度所必需的。这些重要的变量包括以下几项。

- appContext: 该对象将作为 Activity 的 BaseContext。在第 7 章中曾经讲过，Activity 的本质是一个 Context，而同时 Activity 有时继承于 ContextWrapper，该类中需要一个真正的 Context 对象，而这就是 appContext，该对象使用 new ContextImpl() 方法创建。
- this: 这就是指当前 ActivityThread 对象，Activity 对象内部可能会需要主程序的引用。
- r.token: r 是一个 ActivityRecord 对象，其内部变量 token 的含义是 AmS 中的一个 HistoryRecord 对象。
- r.parent: 一个 Activity 可以有一个父 Activity，这种理念是为了允许把一个 Activity 嵌入到另一个 Activity 内部执行。在应用程序使用时，常用 ActivityGroup 类，而 ActivityGroup 功能的内部支持的正是该变量。

③ 在 attach() 方法内部，除了进行重要变量赋值外，另一件重要的事情就是为该 Activity 创建 Window 对象，这是通过调用 PolicyManager 的静态方法 makeNewWindow() 完成的。

PolicyManager 会根据 com.android.internal.policy.impl.Policy 的配置创建不同产品类型的窗口。尽管如此，这仅仅是一种程序设计的灵活性，其代码归根结底只不过是创建了一个 PhoneWindow 对象而已。当前的 Framework 中仅仅定义有两种 Window 的具体实现，一种是 MidWindow 类，另一种是 PhoneWindow 类，而前者基本上没有使用。从这一点也可以看出，Android 的设计之初所设想的两种应

用，其中一种是手机，另一种是便携上网设备（Mobile Internet Device）。

创建好 Window 对象后，将其赋值给 Activity 的内部变量 mWindow，并设置该 Window 的 Callback 接口为当前的 Activity 对象，这就是为什么用户消息能够传递到 Activity 中的原因。如以下代码所示。

```

3737- final void attach(Context context, ActivityThread atThread,
3738-     Instrumentation instr, IBinder token, int ident,
3739-     Application application, Intent intent, ActivityInfo info,
3740-     CharSequence title, Activity parent, String id,
3741-     Object lastNonConfigurationInstance,
3742-     HashMap<String, Object> lastNonConfigurationChildInstances,
3743-     Configuration config) {
3744-     attachBaseContext(context);
3745-
3746-     mWindow = PolicyManager.makeNewWindow(this);
3747-     mWindow.setCallback(this);

```

④ 创建好 Window 对象后，需要给 Window 对象中的 mWindowManager 变量赋值，该变量的类型是 WindowManager 类。每一个 Window 内部都有一个 WindowManager 对象，你可能会觉得，WindowManager 类是一个重量级的类，如果每个 Window 中都包含一个会不会是一种浪费呢？事实上，WindowManager 类仅仅是一个 interface 类而已，真正实现该接口的有两个类，一个是 Window.LocalWindowManager 子类，另一个是 WindowManagerImpl 类。LocalWindowManager 仅仅是一个壳，这就有点像 ContextWrapper，本身虽然也提供了 WindowManager 接口的全部功能，然而真正实现这些功能的却是壳里面的 WindowManager 对象，这就是 WindowManagerImpl 类，其关系如图 8-4 所示。

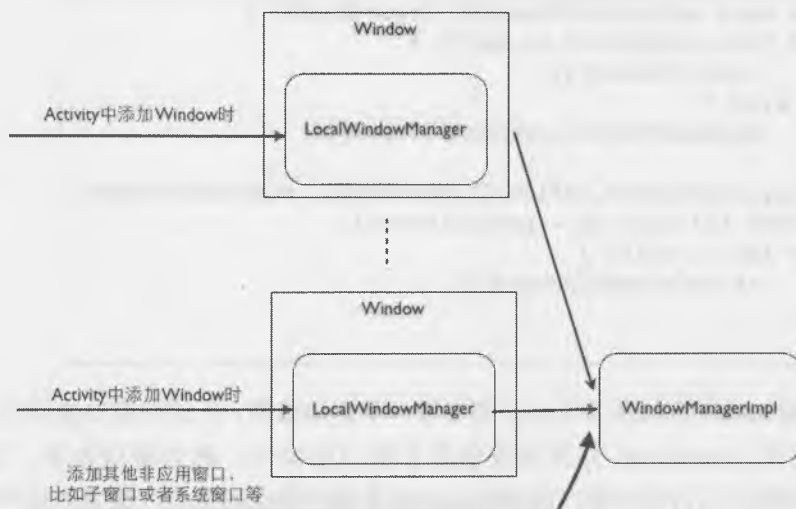


图 8-4 WindowManager 相关的类关系

也正是这种关系，所以，每个 Window 内部才包含了一个 WindowManager 的壳以便进行其他操作，给 mWindowManager 赋值的代码如下：

```

3767         mWindow.setWindowManager(null, mToken, mComponent
3768         if (mParent != null) {
3769             mWindow.setContainer(mParent.getWindow());
3770         }
3771         mWindowManager = mWindow.getWindowManager();

```

在以上代码中，setWindowManager()方法的第一个参数为 null，而在 Window 类该方法实现中，如果第一个参数为 null，其内部就会创建一个 LocalWindowManager 对象。第二个参数正是 AmS 中 Activity 对应的 HistroyRecord 的 Binder 引用，该变量将作为 Window 中的 mAppToken 的值。

每个 Activity 内部也有一个 mWindowManager 对象，其值和 Window 类中的同名变量相同。

⑤ 配置好了 Activity 和 Window 对象后，接下来就需要给该窗口中添加真正的显示元素 View 或者 ViewGroup。这是从 performLaunchActivity()内部调用 callActivityOnCreate()开始的，并会辗转调用到 Activity 的 onCreate()方法中，该方法对于应用程序开发者来讲再熟悉不过了。

在过去的开发经验中，大家都知道给 Activity 添加界面是在 onCreate()方法中调用 setContentView()方法，而该方法实际上却又调用到了其所对应的 Window 对象的 setContentView()方法，如以下代码所示：

```

1646     public void setContentView(int layoutResID) {
1647         getWindow().setContentView(layoutResID);
1648     }

```

因此，下面继续分析 Window 中如何把一个 layout.xml 文件作为 Window 界面。

⑥ PhoneWindow 的 setContentView()方法如以下代码所示：

```

192     public void setContentView(int layoutResID) {
193         if (mContentParent == null) {
194             installDecor();
195         } else {
196             mContentParent.removeAllViews();
197         }
198         mLayoutInflater.inflate(layoutResID, mContentParent);
199         final Callback cb = getCallback();
200         if (cb != null) {
201             cb.onContentChanged();
202         }
203     }

```

首先调用 installDecor()方法为 Window 类安装一个窗口修饰，所谓的窗口修饰就是界面上常见的标题栏，程序中指定的 layout.xml 界面将被包含在窗口修饰中，称为窗口内容。窗口修饰也是一个 ViewGroup，窗口修饰及其内部的窗口内容加起来就是我们所说的窗口，或者叫做 Window 的界面。窗口修饰及窗口内容如图 8-5 所示。

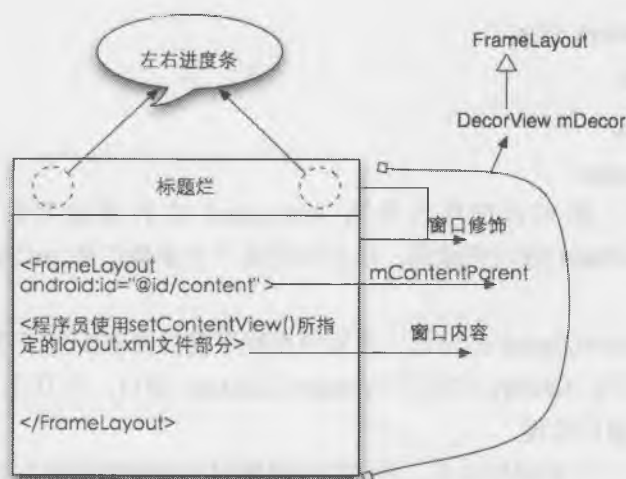


图 8-5 Activity 窗口的组成

Framework 中定义了多种窗口修饰, `installDecor()`代码如下, 该段代码主要完成三件工作。

```

2213 private void installDecor() {
2214     if (mDecor == null) {
2215         mDecor = generateDecor();
2216         mDecor.setDescendantFocusability(ViewGroup.FOCUS_AFTER_DESCENDANTS);
2217         mDecor.setIsRootNamespace(true);
2218     }
2219     if (mContentParent == null) {
2220         mContentParent = generateLayout(mDecor);

```

- 使用 `generateDecor()` 创建一个 `DecorView` 对象, 并赋值给 `mDecor` 变量。该变量并不完全等同于窗口修饰, 窗口修饰是 `mDecor` 内部的唯一一个子视图。
- 根据用户指定的参数选择不同的窗口修饰, 并把该窗口修饰作为 `mDecor` 的子窗口, 这是在 `generateLayout()` 中调用 `mDecor.addView()` 完成的。
- 给 `mContentParent` 变量赋值, 其值是通过调用 `ViewGroup contentParent=(ViewGroup)findViewById(ID_ANDROID_CONTENT)` 获得的, `ID_ANDROID_CONTENT` 正是 `id=content` 的 `FrameLayout`。

不同窗口修饰的区别不大, 比如是否有标题栏, 是否显示左右进度条等, 这些修饰窗口共同的特点是其内部必须包含一个 `id=content` 的 `FrameLayout`, 因为内容窗口正是被包含在该 `FrameLayout` 之中。常见的窗口修饰对应的 XML 文件如下, 其路径为 `frameworks/base/core/res/res/layout`。

- `R.layout.dialog_title_icons`
- `R.layout.screen_title_icons`
- `R.layout.screen_progress`
- `R.layout.dialog_custom_title`

- R.layout.screen\_custom\_title
- R.layout.dialog\_title
- R.layout.screen\_title
- R.layout.screen\_simple

安装完窗口修饰后, 就可以把用户界面 layout.xml 文件添加到窗口修饰中, 这是通过在 setContentView() 中调用 inflate() 方法完成的, 该方法的第二个参数正是 mContentParent, 即 id=content 的 FrameLayout。

最后, 回调 cb.onContentChanged() 方法, 通知应用程序窗口内容发生了改变, 因为从无到有了。而 cb 正是 Activity 自身, 因为 Activity 实现了 Window.Callback 接口, 并且在 attach() 方法中将自身作为 Window 对象的 Callback 接口实现。

以上三件工作中的第二个要特别说明, 所谓的“根据用户指定的参数”中“用户指定”有两个地方可以指定。

第一个地方是在 Activity 的 onCreate() 方法中调用得到当前 Window, 然后调用 requestFeature() 指定。generateLayout() 方法中使用 getLocalFeature() 获取 feature 值, 并根据这些值选择不同的窗口修饰。

另一个是在 AndroidManifest.xml 中 Activity 元素内部使用 android:theme="xxx" 指定。generateLayout() 方法中使用 getWindowStyle() 方法获取这些值, 该方法的调用流程如下:

getWindowStyle(): PhoneWindow 中 generateLayout()

→ obtainStyleAttributes(): Window 中

→ getTheme().obtainStyledAttributes(): Context 中

流程的最后一个调用是 getTheme(), 而这正是使用 android:theme 赋值的结果。

以上“用户指定”及 generateLayout() 方法内部的流程可以总结为如图 8-6 所示。

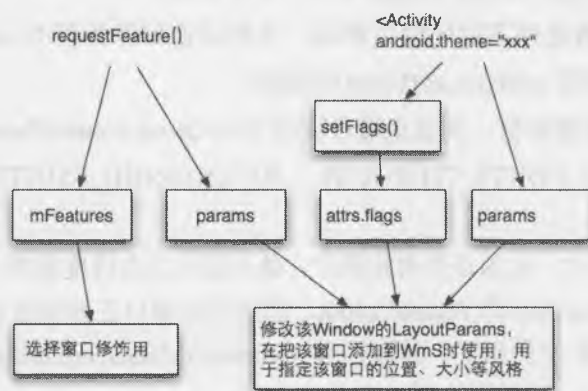


图 8-6 窗口参数的赋值方式

⑦ 给 Window 类设置完其视图元素后, 剩下的就是把创建的这个窗口告诉给 WmS, 以便 WmS

能够把该窗口显示在屏幕上。首先要做的是，当 Activity 准备好后会通知 AmS，然后 AmS 经过各种条件的判断并最终调用到 Activity 的 `makeVisible()` 方法，该方法及后续的各种调用将完成真正的把窗口添加进 WmS 之中。其代码如下：

```
3202 void makeVisible() {
3203     if (!mWindowAdded) {
3204         WindowManager wm = getWindowManager();
3205         wm.addView(mDecor, getWindow().getAttributes());
3206         mWindowAdded = true;
3207     }
3208     mDecor.setVisibility(View.VISIBLE);
3209 }
```

⑧ 在 `makeVisible()` 方法中，首先获得该 Activity 内部的 `WindowManager` 对象，这实际上就是 `Window.LocalWindowManager` 对象，然后调用该对象的 `addView()` 方法，注意这不是 `WindowManagerImpl` 类的 `addView()` 方法。

`addView()` 方法的第一个参数 `mDecor` 是一个 `DecorView` 对象，也就是用户所能看得到的、一个 Activity 对应的全部界面内容；第二个参数是在构造 `Window` 对象时默认构造的 `WindowManager.LayoutParams` 对象，如以下代码所示，该代码在 `Window` 类的初始化代码中。

```
112 // The current window attributes.
113 private final WindowManager.LayoutParams mWindowAttributes =
114     new WindowManager.LayoutParams();
```

`WindowManager.LayoutParams` 的构造函数如下：

```
834 public LayoutParams() {
835     super(LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT);
836     type = TYPE_APPLICATION;
837     format = PixelFormat.OPAQUE;
838 }
```

以上构造函数说明，在默认情况下窗口参数的类型是一个 `TYPE_APPLICATION` 类型，即应用程序类型的窗口。

Activity 中添加窗口时为什么不直接使用 `WindowManagerImpl` 类而是使用一个 `LocalWindowManager` 类呢？其原因是，后者会检查 `WindowManager.LayoutParams` 的值，并给其内部 `token` 变量赋值，以便能够正确添加，而使用 `WindowManagerImpl` 类添加窗口时则不检查 `params` 的值，`LocalWindowManager` 相当于说一道关卡。

`addView()` 的重点代码如下，即上面所说的“关卡”的作用。

```

384     if (wp.type >= WindowManager.LayoutParams.FIRST_SUB_WINDOW &&
385         wp.type <= WindowManager.LayoutParams.LAST_SUB_WINDOW) {
386         if (wp.token == null) {
387             View decor = peekDecorView();
388             if (decor != null) {
389                 wp.token = decor.getWindowToken();
390             }
391         }

```

如果添加的是子窗口，该关卡会检查 params 中的 token，如果 token 为空，则把 Activity 对应的窗口的 token 赋值给 params 的 token。

如果添加的不是子窗口，则把 mAppToken 赋值给 params 的 token。当然，如果该 Activity 被某个 Activity 包含，则把父 Activity 的 mAppToken 赋值给 params 的 token，如以下代码所示。

```

413     if (wp.token == null) {
414         wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;
415     }

```

⑨ 过了 LocalWindowManager 的 addView() 的那道关卡后，就需要来点真格的了，即调用 WindowManagerImpl 的 addView() 方法。一个应用程序内部无论有多少个 Activity，但只有一个 WindowManagerImpl 对象，在 WindowManagerImpl 类中维护三个数组，用于保存该应用程序中所拥有的窗口的状态，它们分别是：

- View[] mViews。这里的每一个 View 对象都将成为 WmS 所认为的一个窗口。
- ViewRoot[] mRoots。所有的 ViewRoot 对象，mViews 中每个 View 对象都对应的 ViewRoot 对象。
- WindowManager.LayoutParams[] mParams。当把 mViews 中的 View 对象当做一个窗口添加进 WmS 中，WmS 要求每个被添加的窗口都要对应一个 LayoutParams 对象，mParams 正是保存了每一个窗口对应的参数对象。

addView() 的执行流程如下。

- (1) 检查所添加的窗口是否已经添加过了，不允许重复添加。
- (2) 如果所添加的窗口为子窗口类型，找到其父窗口，并保存在内部临时变量 panelParentView 中，该变量将作为后面调用 ViewRoot 的 setView() 的参数。
- (3) 创建一个新的 ViewRoot，因为前面说过，每个窗口都对应一个 ViewRoot 对象。
- (4) 调用 ViewRoot 的 setView() 方法，完成最后的、真正意义上的添加工作。

⑩ 完成上面所说的新建一个 ViewRoot 对象后，需要把新建的 ViewRoot 对象添加到 mRoots 对象中，其添加的逻辑是，新建三个长度都加 1 的数组，然后把原来数组 mViews、mRoots、mParams 的内容复制到新建数组，并把新创建的 View、ViewRoot 及 WindowManager.LayoutParams 对象保存到三个数组的最后。如图 8-7 所示。



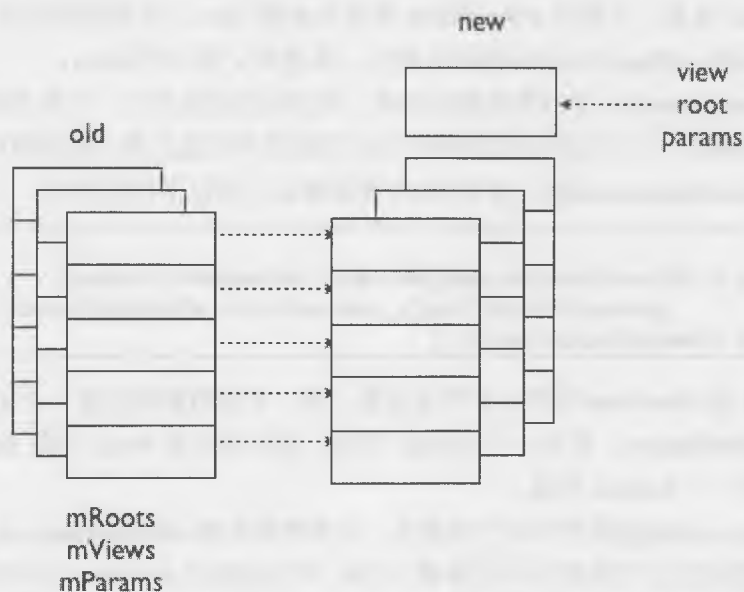


图 8-7 ViewRoot 对象的复制过程

⑪ 调用 ViewRoot 对象的 `setView` (`View view`、`WindowManager.LayoutParams attrs`、`View panelParentView`) 方法，该方法将完成最后的窗口添加工作，三个参数的意义如下。

- `view`: 是 `WindowManagerImpl` 中 `mViews` 数组的一个元素，也就是新建的窗口界面。
- `attrs`: 即为添加窗口的参数，该参数描述该窗口的呈现风格、大小、位置等，尤其是其内部变量 `token`，指明了该窗口和相关 `Activity` 的关系（如果有的话）。
- `panelParentView`: 该对象也是 `WindowManagerImpl` 中 `mViews` 数组的一个元素，仅当该窗口有父窗口时，该值才有意义。

`setView()` 的执行流程如下。

(1) 给 ViewRoot 的重要变量赋值。包括 `mView`、`mWindowAttributes` 及 `mAttachInfo`，如以下代码所示。

```

435         mView = view;
436         mWindowAttributes.copyFrom(attrs);

459         mSoftInputMode = attrs.softInputMode;
460         mWindowAttributesChanged = true;
461         mAttachInfo.mRootView = view;

465         if (panelParentView != null) {
466             mAttachInfo.mPanelParentWindowToken
467                 = panelParentView.getApplicationWindowToken();
468         }
469         mAdded = true;

```



对于 `mAttachInfo` 变量，其成员 `mRootView` 赋值为参数 `view`。如果添加的是子窗口，那么同时给 `mAttachInfo` 的成员 `mPanelParentWindowToken` 赋值，其值为父窗口的 `token`。

(2) 调用 `requestLayout()`，发出界面重绘请求。该方法仅仅是发出一个异步消息，以便 UI 线程下一个消息处理是界面重绘，从而让该窗口在响应任何其他用户消息之前首先变得可见。

(3) 调用 `sWindowSession.add()`，通知 WmS 添加窗口，如以下代码所示。

```
476     try {
477         res = sWindowSession.add(mWindow, mWindowAttributes,
478                                 getHostVisibility(), mAttachInfo.mContentInsets);
479     } catch (RemoteException e) {
```

`sWindowSession` 是 `ViewRoot` 中的一个静态变量，每一个应用程序仅有一个 `sWindowSession` 对象，该对象类型为 `IWindowSession`，即为一个 `Binder` 引用，该引用对应 WmS 中的 `Session` 子类，WmS 为每一个应用程序分配一个 `Session` 对象。

从 `sWindowSession.add()` 这段代码似乎可看出，只要能够获得 `sWindowSession` 的引用，就可以任意创建窗口，而不需要经过以上冗长的十一个步骤，比如，可以不经过 `Activity`，也不经过 `WindowManager`，也不需要创建 `Window` 对象，然而，事实并非如此。因为 `sWindowSession` 这个变量的访问权限为“包内访问”，不加任何权限修饰符即为包内访问，如以下代码所示。

```
97     static IWindowSession sWindowSession;
```

这就意味着应用程序无法直接获取该对象，也就无法调用该对象的 `add()` 方法，而这是客户程序请求 WmS 添加窗口的唯一入口。那么不通过 `ViewRoot` 的 `sWindowSession`，还有没有其他办法获得 `sWindowSession` 呢？这就要问：`sWindowSession` 变量的值是怎么来的呢？答案是，在 `ViewRoot` 的构造函数中。构造函数中会调用 `getWindowSession()`，而该方法内部会判断 `sWindowSession` 是否为空，如果为空则创建一个，如以下代码所示，这也是所谓的工厂模式创建对象。

```
214     public static IWindowSession getWindowSession(Looper mainLooper) {
215         synchronized (mStaticInit) {
216             if (!mInitialized) {
217                 try {
218                     InputMethodManager imm = InputMethodManager.getInstance();
219                     sWindowSession = IWindowManager.Stub.asInterface(
220                         ServiceManager.getService("window"))
221                         .openSession(imm.getClient(), imm.getInputConfiguration());
222                     mInitialized = true;
223                 } catch (RemoteException e) {
224                 }
225             }
226             return sWindowSession;
227         }
228     }
```

由以上代码的 219 行可以看出，`sWindowSession` 是通过 `IWindowManager.Stub.asInterface()` 而来的，函数的参数实际上是一个 `WindowManager` 对象，可问题是，`IWindowManager` 类在源码中是 `@hide` 的，

如以下代码所示。

```

65 /**
7  * System private interface to the window manager.
8  *
9  * {@hide}
10 */
11 public interface IWindowManager extends android.os.IInterface
12 {

```

hide 意味着, SDK 中将不包含该类, 这再次打消了我们试图不经过 ViewRoot 类添加窗口的念头。看来, 在标准的程序设计中, 只能通过 WindowManager 类来创建窗口。如果产品有特别需要, 可以根据以上流程改变相应的 Framework 代码, 以达到灵活配置的需求。

到此为止, 从客户端的角度来讲, 已经完成了窗口创建的全部工作。

## 8.4 创建子窗口

上一节介绍了创建应用类窗口的过程, 本节介绍子窗口的创建。事实上, 对于 WmS 来讲, 不是十分在意客户端要创建什么类型的窗口, 它都会一视同仁地进行创建工作, 所不同的仅仅是在 WmS 端会保存和窗口的父子关系, 以便在需要的时候使用这些信息。因此, 创建子窗口和应用窗口的差别不是很大。本节就以 Framework 提供的一些典型的子窗口为例, 来说明创建的过程。

首先需要明确的是, “子” 的含义是什么?

从程序的角度来看, 与 “子” 的含义相关的有三个变量, 分别如下。

第一个变量, View.AttachInfo 类中的 IBinder mPanelParentWindowToken:

每一个 View 中都有一个 AttachInfo 对象, 而每一个窗口实际上都对应一个 View 类。如果该变量不为空, 则代表着它有一个父窗口, 其值为父窗口的同名变量值。如果其父窗口不是子窗口, 那么其值为 mWindowToken 的值, 如以下代码所示, 而其赋值的时机是在 ViewRoot 中的 setView() 方法中。

```

5979 public IBinder getApplicationWindowToken() {
5980     AttachInfo ai = mAttachInfo;
5981     if (ai != null) {
5982         IBinder appWindowToken = ai.mPanelParentWindowToken;
5983         if (appWindowToken == null) {
5984             appWindowToken = ai.mWindowToken;
5985         }
5986         return appWindowToken;
5987     }
5988     return null;
5989 }

```

第二、三个变量, Window 中的 boolean mHasChild 和 Window mContainer。

前者指明该 Window 对象是否有子窗口, 同时, 后者表示其父 Window 对象。这两个变量是在 setContainer() 方法中赋值的, 如以下代码所示:

```

332 public void setContainer(Window container) {
333     mContainer = container;
334     if (container != null) {
335         // Embedded screens never have a title.
336         mFeatures |= 1<<FEATURE_NO_TITLE;
337         mLocalFeatures |= 1<<FEATURE_NO_TITLE;
338         container.mHasChildren = true;
339     }
340 }

```

遍历整个 Framework 代码，会发现调用该方法的时机只有一处，那就是在 Activity 类的 attach() 方法中，在调用 attach() 方法时，可以指定 Activity 对象是否有父 Activity。如果有的话，会把父 Activity 的 Window 对象作为子 Activity 的 Window 对象的父窗口，如以下代码所示。从这里也可以看出，Framework 中提供的所有独立窗口(Window)控件，虽然名义上是一个子窗口，但是，其内部的 mContainer 和 mHasChildren 都没有被赋值，有些形同虚设，但 AttachInfo 中的变量 mPanelParentWindowToken 却是无论如何都被赋值了。

至于 Framework 将来会如何使用以上三个变量，我们将会在 WmS 的内部调度机制章节中介绍。

下面就来介绍具体的几个子窗口。

#### 8.4.1 Dialog 的创建

首先需要声明，Dialog 对应的窗口其默认类型并不是“子窗口”，而是应用类窗口，之所以把它放在这里讲，原因是很多 Dialog 的子类会改变默认窗口类型，使之成为一个“子窗口”。

在 Dialog 的构造函数中完成创建一个 Window 对象，实际上也是 PhoneWindow 类，如以下代码所示：

```

138 public Dialog(Context context, int theme) {
139     mContext = new ContextThemeWrapper(
140         context, theme == 0 ? com.android.internal.R.style.Theme
141         mWindowManager = (WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
142     Window w = PolicyManager.makeNewWindow(mContext);
143     mWindow = w;
144     w.setCallback(this);
145     w.setWindowManager(mWindowManager, null, null);
146     w.setGravity(Gravity.CENTER);
147     mUiThread = Thread.currentThread();
148     mListenersHandler = new ListenersHandler(this);
149 }

```

以上代码主要完成以下几件重要的事情。

- 创建 Dialog 内部的 Context 对象，并赋值给 mContext，这是通过调用 new ContextThemeWrapper() 实现的。在理解 Context 一章中曾说过，真正实现 Context 的是 ContextImpl 类，所有 Dialog 中的 mContext 也仅仅是一个“壳”，而壳中的真正 Context 还需要在构造函数的参数中指定。

- 调用 `makeNewWindow()` 方法创建一个 `Window` 对象，这种调用方法和 `Activity` 中创建 `Window` 对象的方法是相同的。
- 调用 `w.setCallback()` 方法指定该 `Window` 的消息回调接口为本 `Dialog` 对象。还记得在 `Activity` 中的初始化代码中，也调用过 `setCallback()` 方法，`Activity` 把其包含的 `Window` 的 `Callback` 接口设置为 `Activity` 对象，从这点也可以看出，从对消息处理的角度来看，`Dialog` 等同于 `Activity`。
- 在设置 `Window` 对象的内部 `WindowManager` 对象。注意这里是调用参数 `Context` 对象，在一般情况下，该 `Context` 对象应该是某个 `Activity`，而 `Activity` 类内部也有一个 `WindowManager` 对象，因此，这里实际上把 `Activity` 的 `WindowManager` 对象赋值给了 `Dialog`。
- 创建一个回调句柄 `mListenerHandler`，主要用于在内部发送异步消息。该变量的赋值说明，一般来讲，`Dialog` 对象必须在 UI 线程中创建，而不能在普通的 `Thread` 类中创建，除非为普通的 `Thread` 线程也添加一个 `Looper` 对象，因为 `Handler` 对象只能在拥有 `Looper` 对象的线程中创建。

`Dialog` 创建了内部的 `Window` 对象后，接下来，当应用程序调用 `show()` 方法时，该 `Dialog` 就会显示到屏幕上。然而，在调用 `show()` 方法前，`Dialog` 内部仅仅是创建了一个 `Window` 对象，而并没有告知 `WmS` 添加一个可以显示的窗口。因此，`show()` 方法中必须实现向 `WmS` 中添加一个真正的可以显示的窗口，而不仅仅是 `Window` 对象。

`show()` 方法内部主要完成以下几件事情。

第一，如果该窗口已经存在，则直接显示该窗口，并返回，如以下代码所示。

```

217         if (mShowing) {
218             if (mDecor != null) {
219                 mDecor.setVisibility(View.VISIBLE);
220             }
221             return;
222         }

```

第二，如果该窗口还不存在，则给应用程序一个设置该窗口内容的机会，即回调 `Dialog` 的 `onCreate()` 方法，应用程序可以在该回调函数中使用 `setContentView()` 方法为窗口添加具体要显示的内容，如以下代码所示。

```

224         if (!mCreated) {
225             dispatchOnCreate(null);
226         }

```

你可能会心存疑虑，如果在 `onCreate()` 方法中要调用 `setContentView()` 方法，前提是该窗口内部的 `DecorView` 对象必须已经存在，否则，`setContentView()` 会因为没有父 `View` 而无法创建。事实上，在 `PhoneWindow` 的 `setContentView()` 方法中，它会检测该 `Window` 内部的 `DecorView` 对象是否存在，如果不存在，则调用 `installDecor()` 为该 `Window` 创建一个 `DecorView` 对象。所以，不用担心，`PhoneWindow` 中的 `setContentView` 部分代码如下：

```

192     public void setContentView(int layoutResID) {
193         if (mContentParent == null) {
194             installDecor();
195         } else {

```

第三, 为该 Dialog 中的 mDecor 变量赋值, 并且设置添加 Dialog 窗口所使用的 LayoutParams 参数, 如以下代码所示。

```

229         mDecor = mWindow.getDecorView();
230         WindowManager.LayoutParams l = mWindow.getAttributes();
231         if ((l.softInputMode
232             & WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARDING) != 0) {
233             WindowManager.LayoutParams nl = new WindowManager.LayoutParams(
234                 l.width, l.height, l.type, l.packageName, l.flags,
235                 nl.softInputMode |
236                 WindowManager.LayoutParams.SOFT_INPUT_IS_FORWARDING);
237             l = nl;
238         }

```

第四, 调用 WindowManager 的 addView() 方法, 添加窗口。这与 Activity 中添加窗口是一样的, 事实上, 程序员要添加自己的窗口都是通过调用该方法完成的, 如以下代码所示。

```

240         try {
241             mWindowManager.addView(mDecor, l);
242             mShowing = true;
243         }

```

第五, 给应用程序一个机会, 当窗口显示后可以做点什么。程序员可以调用 Dialog 的 setOnShowListener() 为 Dialog 添加一个回调接口。有点特别的是, 执行该回调函数是异步的, 即在 UI 线程的下一个消息循环中执行而不是在当前循环中, 因为 Dialog 此处是调用 sendMessage() 发送了一个异步消息。为什么回调 onShow() 被设计成异步处理而不是同步处理呢? 比如 onCreate() 回调就是同步回调。一般来讲, 异步和同步能得到相同的结果, 但作为设计习惯, 有以下几点需要区别异步和同步。

- 如果回调的目的是为本次循环设置某些参数, 那么, 必须采用同步回调。比如 onCreate() 回调中是为了给 Window 添加显示内容, 以便接下来的 addView() 使用, 因此必须采用同步回调。
- 如果回调函数中指定的代码依赖本次循环后续代码的执行结果, 那么必须采用异步回调, 否则, 回调代码会因为某种条件不具备而无法正确执行。
- 如果回调代码的执行时间比较长, 那么尽量采用异步回调, 因为系统为每一个循环的执行时间设置的阈值是 5 秒, 如果 5 秒之内还没有执行下一个循环, 系统会弹出应用程序无响应 (ANR) 的对话框。

以上规则可如图 8-8 和图 8-9 所示。

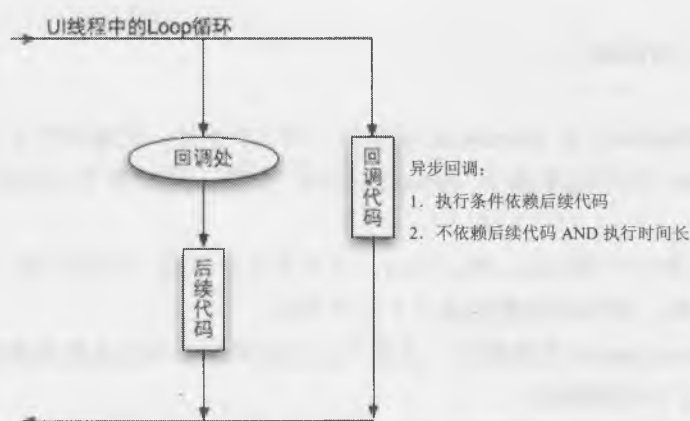


图 8-8 同步回调条件

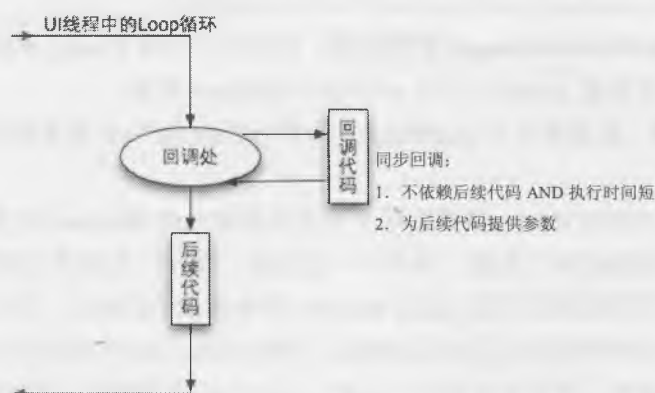


图 8-9 异步回调条件

而在调用 `mWindowManager.addView()` 的内部，关于 `Window` 的 `token` 赋值需要注意。因为 `mWindowManager` 实际上是创建 `Dialog` 的 `Activity` 中的同名变量，因此，`addView()` 方法首先会执行到 `Window.LocalWindowManager` 中的 `addView()` 方法，而在该方法中判断 `wp` 参数时，由于 `Dialog` 的默认类型是应用类窗口，所以，代码会执行到以下地方：

```

412         } else {
413             if (wp.token == null) {
414                 wp.token = mContainer == null ? mAppToken : mContainer.mAppToken;
415             }

```

以上代码给 `wp.token` 赋值时，变量 `mContainer` 指的是 `Activity` 对应的窗口的父窗口，`mAppToken` 也是属于 `Activity` 对应的窗口。在一般情况下，`mContainer` 为空，因此 `Dialog` 的 `token` 值即为创建其 `Activity` 的窗口的 `mAppToken` 值。

以上就是 `Dialog` 内部添加窗口的过程。

## 8.4.2 PopupWindow 的创建

弹出窗口（PopupWindow）是 Framework 提供的一种 UI 控件，比如那种下拉列表实际上就是一个 PopupWindow，AutoText 控件也是基于 PopupWindow 实现，因此本节介绍这种常见的子窗口——PopupWindow。

PopupWindow 并不继承于 Window 类，所以，该类本身并不是一个窗口类。该类有多个构造函数，但无论是哪一个构造函数，其中必须要完成以下几件事情。

第一，为该类中的 mContext 变量赋值。该值可以直接来源于构造函数的参数，也可以根据参数中的 View 对象获得，如以下代码所示：

```
2748 public PopupWindow(View contentView, int width, int height, boolean focusable) {
275     if (contentView != null) {
276         mContext = contentView.getContext();
```

第二，为该类中的 mWindowManager 变量赋值。赋值的方法和 Dialog 相同，即该变量的值实际上是创建该 PopupWindow 对象的 Activity 中的 mWindowManager 对象。

除了以上两个赋值外，还需要为 PopupWindow 中的 mContentView 对象赋值，该值就是该窗口中真正要显示的内容。

分析 PopupWindow 的源码可知，该类实际上并没有创建任何 Window 对象，换句话说，它添加的是一个纯粹的窗口，而不添加任何“杂质”。本书中一直强调，“窗口”是指真正看得见的东西，而 Window 类仅仅是一个针对窗口交互的抽象而已，比如 Window 类中有菜单的概念，还可以处理系统按键，比如“Home”、“Back”等，而纯粹的窗口是没有这些的，纯粹的窗口会把所有处理用户消息的工作交给所包含的 View/ViewGroup 去处理，即不包含任何“杂质”。而在 WmS 的眼里，所有的窗口都是纯粹的窗口。

因此，PopupWindow 中不存在，也不需要创建 Window 对象，该类提供了以下 API 给程序员，程序员可以调用这些 API 把 mContentView 对应的窗口显示到界面上的任何位置。

```
public void showAtLocation(View parent, int gravity, int x, int y);
public void showAsDropDown(View anchor, int xoff, int yoff);
```

这两个方法的第一个参数其实都应该命名为 parent 或者 anchor，其内部唯一的区别是指定窗口显示位置的方式不同，如其函数名称所示，此处不再赘述。而两个方法内部的调用流程却是相同的，如图 8-10 所示。

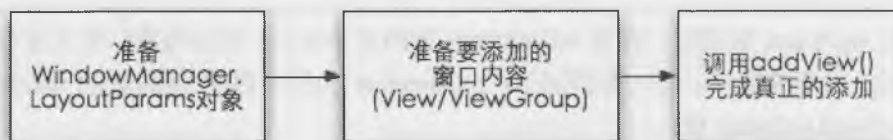


图 8-10 添加窗口的标准过程



首先来看 `LayoutParams` 对象。这是通过调用 `createPopupLayout()` 方法完成的, 该方法的参数是一个 `IBinder` 对象, 其值应该为父窗口的 `token` 值, 因此, 这里通过调用 `parent.getWindowToken()` 获得。

创建的 `LayoutParams` 的内部变量值如下:

```

841     WindowManager.LayoutParams p = new WindowManager.LayoutParams
842     // these gravity settings put the view at the top left corner
843     // screen. The view is then positioned to the appropriate location
844     // by setting the x and y offsets to match the anchor's bottom
845     // left corner
846     p.gravity = Gravity.LEFT | Gravity.TOP;
847     p.width = mLastWidth - mWidth;
848     p.height = mLastHeight - mHeight;
849     if (mBackground != null) {
850         p.format = mBackground.getOpacity();
851     } else {
852         p.format = PixelFormat.TRANSLUCENT;
853     }
854     p.flags = computeFlags(p.flags);
855     p.type = WindowManager.LayoutParams.TYPE_APPLICATION_PANEL;

```

以上主要注意该 `LayoutParams` 对象的 `type` 类型, 即“子窗口”。根据之前的介绍, 添加子窗口时, `LocalWindowManager` 要检查其 `token` 值必须为父窗口的 `token`, 因此, 这里同时给 `p.token` 赋值为父窗口的 `token` 值。

接下来要准备窗口内容。`PopupWindow` 提供了 `setContentView()` 方法用于设置窗口内容, 但同时又提供了 `setBackgroundDrawable()` 方法用于设置窗口背景。因此, 在准备窗口内容的 `preparePopup()` 方法中, 会判断是否有窗口背景, 如果没有的话, 窗口内容就是 `mContentView`, 否则会创建一个 `PopupViewContainer` 视图, 它实际上是一个 `FrameLayout`, 然后把 `mContentView()` 添加到该 `FrameLayout` 中并返回。具体代码参照源码。

最后, 在 `invokePopup()` 方法中调用 `mWindowManager.addView()` 方法完成窗口的添加。

`PopupWindow` 尽管已经添加好了窗口, 然而, 该窗口还不能向 `Window` 类中包含的窗口那样能够自动处理“Home”、“Back”等系统按键, 所有的消息处理必须由 `mContentView` 内部去完成。在本章最后一节的例子中演示了如何处理这些消息。

### 8.4.3 ContextMenu 的创建

情景菜单 (`ContextMenu`) 一般是长按 `ListView` 的某个 `Item` 时弹出的菜单, 当然, 那只是场景菜单弹出的一般模式, 程序可以给任意一个 `View` 设置场景菜单, 并且启动的方式不一定是长按, 可以是任意消息组合, 比如双击、单击、滑动等。

场景菜单实际上是一个 `Dialog`, 所不同的是, 场景菜单把创建好了的 `Dialog` 添加到 `WmS` 时, 会修改该 `Dialog` 窗口的类型, 即从默认的应用类型修改为子窗口类型。



## 1. 触发情景菜单的消息

显示场景菜单一般有两种方式，第一种是当用户长按某个 View 时，如果该 View 已经添加过场景菜单，则会弹出一个场景菜单窗口；另一种是程序员调用 `openContextMenu()` 方法。如图 8-11 所示。

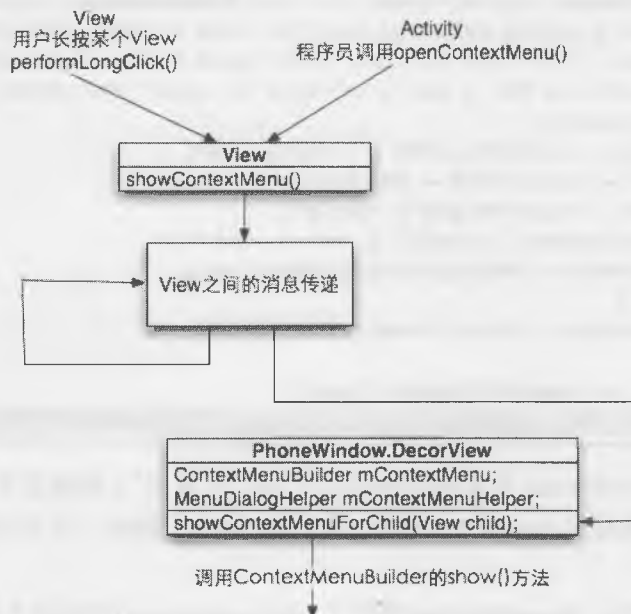


图 8-11 情景菜单的触发过程

为什么长按时会触发情景菜单呢？因为在 View 类的 `performLongClick()` 方法中代码如下：

```

handled = mOnLongClickListener.onLongClick(View.this);
if (!handled) {
    handled = showContextMenu();
}

```

即首先执行长按的回调函数，如果用户没有处理该消息，则调用 `showContextMenu()` 方法，这就启动了显示情景菜单的流程。

## 2. 菜单功能中的几个类关系

在 `showContextMenu()` 内部则调用其 `mParent` 的 `showContextMenuForChild()`。对于应用类窗口，任何 View 的根视图都是 DecorView 对象，所以最终都会调用到 DecorView 的同名方法，其源码如下：

```

1851     public boolean showContextMenuForChild(View originalView) {
1852         // Reuse the context menu builder
1853         if (mContextMenu == null) {
1854             mContextMenu = new ContextMenuBuilder(getContext());
1855             mContextMenu.setCallback(mContextMenuCallback);
1856         } else {
1857             mContextMenu.clearAll();
1858         }
1859
1860         mContextMenuHelper = mContextMenu.show(originalView, originalView.getWindowToken());
1861         return mContextMenuHelper != null;
1862     }

```

PhoneWindow 中有两个重要变量，其类型分别为 ContextMenuBuilder 和 MenuDialogHelper。前者的作用是管理菜单中的数据并提供操作这些数据的方法，而后者仅仅是为了把菜单数据以窗口的方式显示到屏幕上，即完成真正的窗口添加的工作。这两个类之中又包含了其他一些菜单相关的类，其关系如图 8-12 所示。

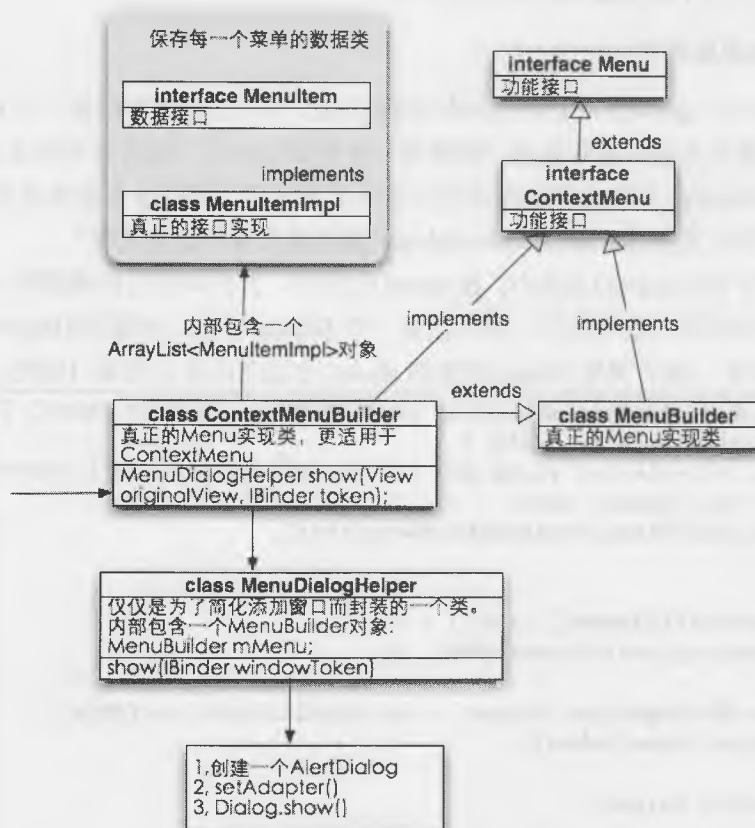


图 8-12 菜单相关的类关系

图 8-12 所示中几个类的关系解释如下。

- **Menu**: 该类是一个 interface, 描述了一个菜单应该具备的操作接口, 这里所说的“一个菜单”是指整个菜单窗口, 而不是一个菜单条目。ContextMenu 类仅仅是对 Menu 作了一定的补充, 以更确切地描述情景菜单。MenuBuilder 则是一个菜单的真正实现, 从架构的角度来看, 似乎没有必要再提供 ContextMenuBuilder 了, 然而由于 MenuBuilder 是同时面向情景菜单和选项菜单的, 因此缺乏一些情景菜单特有的操作, 所以才再提供了一个 ContextMenuBuilder 类。
- **MenuItem**: 该类是一个 interface, 描述一个菜单条目应该有的操作接口。MenuItemImpl 则真正实现了 MenuItem 类。这两个类侧重于保存每一个菜单条目的数据, 因此可认为是数据类。
- **ContextMenuBuilder**: 该类中包含了一个 ArrayList<MenuItemImpl> 变量, 用于保存菜单中的每一个条目信息, 并提供了情景菜单所特有的接口, 是包含了情景菜单的完整实现。
- **MenuDialogHelper**: 由于 ContextMenuBuilder 侧重于提供菜单内容的操作, 而如何把菜单显示在界面上则可以非常灵活, 比如一般的情景菜单和选项菜单的显示方式就不同, 因此 MenuDialogHelper 对象专门用于根据菜单内容创建窗口。

### 3. 给菜单中添加菜单条目

接前面 DecorView 的 showContextMenuForChild() 方法, 该方法首先会创建一个 ContextMenuBuilder 对象, 此时该对象内部没有任何菜单条目, 仅仅是一个空架子而已, 然后再调用其 show() 方法。

这里请思考, 该 Builder 对象此时还尚未添加任何菜单条目, 那么这些菜单条目是何时添加进去的呢? 以往开发应用程序时重载 onCreateContextMenu() 和这里又有什么关系呢?

答案全部都在接下来的 show() 方法中。在 show() 方法中, 首先调用长按视图的 createContextMenu() 为上面创建的空架子里面添加菜单条目, 然后创建一个 Helper 对象, 注意该 Helper 对象的构造函数的参数即为该 Builder 对象, 最后调用 Helper 对象的 show() 方法完成真正的窗口创建, 如以下代码所示:

```

77 public MenuDialogHelper show(View originalView, IBinder token) {
78     if (originalView != null) {
79         // Let relevant views and their populate context listener
80         // the context menu
81         originalView.createContextMenu(this);
82     }
83
84     if (getVisibleItems().size() > 0) {
85         EventLog.writeEvent(50001, 1);
86
87         MenuDialogHelper helper = new MenuDialogHelper(this);
88         helper.show(token);
89
90         return helper;
91     }

```

其中, 长按视图的 createContextMenu() 方法中有三次机会添加菜单内容。如以下代码所示:

```

4214 public void createContextMenu(ContextMenu menu) {
4215     ContextMenuInfo menuInfo = getContextMenuInfo();
4216
4217     // Sets the current menu info so all items added to menu
4218     // my extra info set.
4219     ((MenuBuilder)menu).setCurrentMenuInfo(menuInfo);
4220
4221     onCreateContextMenu(menu);
4222     if (mOnCreateContextMenuListener != null) {
4223         mOnCreateContextMenuListener.onCreateContextMenu(menu,
4224     }
4225
4226     // Clear the extra information so subsequent items that are
4227     // have my extra info.
4228     ((MenuBuilder)menu).setCurrentMenuInfo(null);
4229
4230     if (mParent != null) {
4231         mParent.onCreateContextMenu(menu);
4232     }
4233 }

```

第一次, onCreateContextMenu(menu)。如果该视图是自定义视图,那么程序员可以重载该方法,给 menu “空架子”中添加具体的菜单条目。

第二次:

```

if (mOnCreateContextMenuListener != null) {
    mOnCreateContextMenuListener.onCreateContextMenu(menu,
this, menuInfo);
}

```

视图类中的该 Listener 对象是调用 setOnCreateContextMenuListener()方法赋值的,而该方法又是在 Activity 的 registerForContextMenu()方法中调用的,如以下代码所示。这就是为什么要使 Activity 中的 onCreateContextMenu()方法生效,必须调用 registerForContextMenu()的原因。

```

2367 public void registerForContextMenu(View view) {
2368     view.setOnCreateContextMenuListener(this);
2369 }

```

第三次,调用父视图的同名方法。这就意味着父视图中添加的菜单条目将会出现在每一个子视图中。

通过以上步骤,完成为菜单“空架子”中添加具体的菜单条目,接下来就要调用 Helper 的 show()方法添加真正的菜单窗口了。

#### 4. 创建菜单窗口并显示

Helper 的 show()方法的内部执行流程如下。

① 获得菜单“架子”(此时已经不是空架子了)内部的 ListAdapter 对象。该对象实际上是 Builder 内部根据菜单条目而产生的一个 ListAdapter 对象。为什么要这个对象呢?因为 AlertDialog 允许把一个 ListAdapter 对象作为对话框的内容,并以列表的方式显示出来。

② 设置菜单窗口的标题及图标。

③ 设置菜单窗口的 onKeyListener 为本 Helper 对象。因为情景菜单状态下,会对“Menu”键及“Back”

键做一些特别的处理。

④ 调用 `builder.create()` 方法创建一个 `AlertDialog` 对象。

⑤ 改变 `AlertDialog` 的默认窗口类型为子窗口类型，并调用 `Dialog` 对象的 `show()` 方法完成添加，如以下代码所示。

```

79     WindowManager.LayoutParams lp = mDialog.getWindow().getAttributes();
80     lp.type = WindowManager.LayoutParams.TYPE_APPLICATION_ATTACHED_DIALOG;
81     if (windowToken != null) {
82         lp.token = windowToken;
83     }
84     lp.flags |= WindowManager.LayoutParams.FLAG_ALT_FOCUSABLE_IM;
85     mDialog.show();
86 
```

关于 `Dialog` 的 `show()` 方法内部过程参见上一节。

#### 8.4.4 OptionMenu 的创建

选项菜单（OptionMenu）一般是指用户按下“Menu”键后弹出的菜单。在这一点上，Android 继承了传统手机的操作模式，即按下一个键才弹出菜单，而在 iPhone 中，菜单一般是直接放在屏幕下方的，用户可以直接操作。也正因为这一点的区别，当前很多 Android 手机在模仿 iPhone 的菜单模式时，却不知道如何平衡“Menu”按键和直接显示菜单，往往给用户造成一定的困惑，并造成 Android 应用程序菜单操作的不一致性。笔者个人认为，应该保持 Android 应用操作的一致性，所以，不用为了牵强所谓的“最少的操作步骤”而把菜单内容直接显示到屏幕下方。

##### 1. “Menu”键的消息处理流程

启动选项菜单有两种方式，一种要按下“Menu”键，另一种是调用 `openOptionsMenu()` 方法。“Menu”键的处理过程如图 8-13 和图 8-14 所示。



图 8-13 “Menu”键 down 消息的处理过程

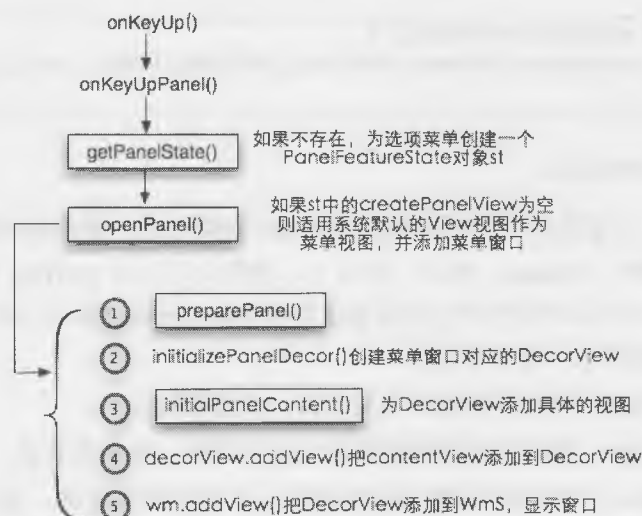


图 8-14 “Menu” 键 up 消息的处理过程

对于通用的窗口而言，所有的用户消息都由窗口中的视图去处理，而在一个 Activity 中，程序员并没有去处理“Menu”键，那么为什么还会有一个选项菜单窗口弹出来呢？这就归功于 PhoneWindow 类，这也是 PhoneWindow 的一个特点。前面曾说过 Window 类与普通窗口的本质区别就是 Window 定义了窗口交互行为，而普通窗口则侧重于窗口显示的内容，而这个处理“Menu”键的行为正是在 Window 类中定义，并且在 PhoneWindow 的实现中默认处理了该“Menu”按键消息，其代码如下，在 PhoneWindow 的 onKeyDown()方法中调用 onKeyDownPanel()方法，添加一个选项菜单视图。

```

1191         case KeyEvent.KEYCODE_MENU: {
1192             onKeyDownPanel((featureId < 0) ? FEATURE_OPTIONS_PANEL :
1193                 return true;
1194         }
  
```

然后在 onKeyUp()方法中调用 onKeyUpPanel()显示选项菜单，如下代码所示。

```

1318         case KeyEvent.KEYCODE_MENU: {
1319             onKeyUpPanel(featureId < 0 ? FEATURE_OPTIONS_PANEL :
1320                 event);
1321             return true;
1322         }
  
```

既然在 PhoneWindow 中已经处理了“Menu”键，所以，在 Activity 中，我们不再需要去处理该键消息，而是仅仅实现一些回调函数添加具体的菜单数据即可。

在 Activity 调用 showOptionsMenu()时，则会直接导致调用 PhoneWindow 的 openPanel()方法，如下代码所示。

```

2329 public void openOptionsMenu() {
2330     mWindow.openPanel(Window.FEATURE_OPTIONS_PANEL, null);
2331 }

```

## 2. 选项菜单中相关的类关系

PhoneWindow 中有一个重要的子类 PanelFeatureState, 该类包含了该 PhoneWindow 中所有的子窗口, 选项菜单窗口只是一种特色 (Feature) 而已。事实上, 程序员可以为 Activity 中 Window 类添加任意多个不同的特色窗口, 而所有这些特色窗口的信息保存在 PanelFeatureState[] mPanels 变量中。

PanelFeatureState 中包含以下重要变量。

- DecorView decorView: 每个特色窗口都应该有一个根视图。
- View createPanelView: 程序员使用回调提供的窗口视图, 如果有的话。
- View showPanelView: 菜单窗口最终显示的视图, 从设计的角度讲, 该变量不是必需的。
- MenuBuilder menu: 选项菜单中的菜单实体, 其继承关系参照前面几节内容。

以上重要变量的类调用关系如图 8-15 所示。

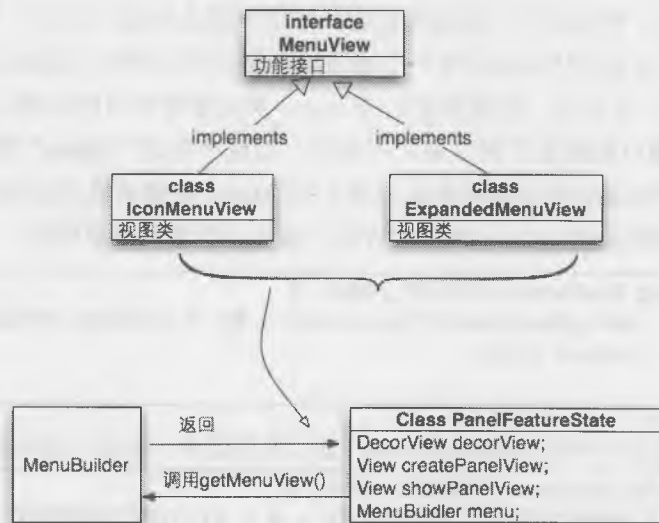


图 8-15 选项菜单相关的类关系

首先, PhoneWindow 会检查 PanelFeatureState 对象 st 中的 createPanelView 是否有值, 如果没有的话就调用 menu 的 getMenuView() 方法, 而该方法会返回一个实现了 MenuView 接口的一个 View 对象, 并将返回的对象赋值给 showPanelView。

IconMenuView 就是常见的选项菜单, 每个菜单带一个图标, 所以叫做 IconMenuView。

ExpandedMenuView 是那种展开式的菜单, 一般当菜单内容超过 5 个后, 会有一个“更多”选项, 选择“更多”选项后, 就会弹出一个 ExpandedMenuView。



getMenuView()方法是在 openPanel()→initPanelContent()中调用的,该方法中创建 MenuView 对象的代码有点特别,如下所示:

```
196     if (menuView == null) {
197         menuView = (MenuView) getInflater().inflate(
198             LAYOUT_RES_FOR_TYPE[mMenuType], parent, false);
199         menuView.initialize(MenuBuilder.this, mMenuType);
```

inflate()方法返回的是一个 View 对象,而 MenuView 仅仅是一个 interface 而已,却被强制转换为 MenuView 对象,看起来有点费解。其原因在于 inflalte() 的第一个参数中定义的常量 LAYOUT\_RES\_FOR\_TYPE,该常量的定义如下:

```
79     static final int LAYOUT_RES_FOR_TYPE[] = new int[] {
80         com.android.internal.R.layout.icon_menu_layout,
81         com.android.internal.R.layout.expanded_menu_layout,
82         0,
83     };
```

即 inflate()出来的 View 只能是常量中描述的两种,而查看这两个 layout 文件发现,其分别对应 IconMenuView 和 ExpandedMenuView 两个类,而这两个类又都实现了 MenuView 接口,这就是为什么能够把 inflalte()返回的结果强制转换为 MenuView 的原因。

### 3. 为选项菜单添加视图

添加视图的步骤上面已经介绍过,这里不再赘述,需要特别说明的是,系统为应用程序员提供了两次机会去添加自定义的选项菜单。

第一次,可以重载 Activity 的 onCreatePanelView()方法提供一个完全自定义的选项菜单视图。

第二次,可以重载 onCreatePanelMenu()或者 onPreparePanel()往选项菜单中添加具体的菜单。

除了这种标准的做法外,如果想从 Framework 级别自定义菜单,那么可以修改 initPanelContent()方法及 getMenuView()方法,包括提供自定义的实现 MenuView 接口视图类。

### 4. 添加菜单窗口

以上小节中准备好了窗口内容,最后一步就是把这些内容添加到 WmS 中,如以下代码所示。

```
453 WindowManager.LayoutParams lp = new WindowManager.LayoutParams(
454     WRAP_CONTENT, WRAP_CONTENT,
455     st.x, st.y,
456     WindowManager.LayoutParams.TYPE_APPLICATION_ATTACHED_DIALOG,
457     WindowManager.LayoutParams.FLAG_DITHER
458     | WindowManager.LayoutParams.FLAG_ALT_FOCUSABLE_IM,
459     st.decorView.mDefaultOpacity);
460
461 lp.gravity = st.gravity;
462 lp.windowAnimations = st.windowAnimations;
463
464 wm.addView(st.decorView, lp);
```



从以上代码可以看出, PanelFeatureState 对象 st 中 decorView 是最终窗口的根视图, 窗口的类型为 TYPE\_APPLICATION\_ATTACHED\_DIALOG, 即子窗口类型。而 lp 此时的 token 值为空, 在之后的 LocalWindowManager.addView()方法内部会把 Activity 对象的窗口的 token 赋值给 lp 的 token。

至此完成了添加选项菜单窗口。

对比一下添加情景菜单的流程发现, 情景菜单实际上是一个 Dialog 对象, 即会创建一个 PhoneWindow 对象, 而选项菜单仅仅是一个 View。

另外, 选项菜单对象 st 中的 menu 对象的内容将一直保存在内存中, 而情景菜单的内容会根据不同的长按视图而不断变化, 因此, 选项菜单中的 st 对象中的 menu 对象可以保存为 Activity 中的持久变量使用, 而情景菜单中的 menu 对象却不能。

## 8.5 系统窗口 Toast 的创建

系统窗口的含义有两个方面。

- 系统窗口不依赖于应用, 而应用类窗口都必须有一个应用 Activity 与之对应;
- 系统窗口是由系统创建的, 应用程序没有权限创建。但有三个系统窗口例外, 这三个系统窗口分别为 TYPE\_TOAST、TYPE\_INPUT\_METHOD、TYPE\_WALLPAPER。

从代码的角度来看, 为什么添加其他系统窗口需要权限检查, 而这三个不需要呢?

首先, 在 WmS 这端, 当客户端请求添加窗口时, 就会调用 WmS 中的 addWindow()方法, 而该方法首先要进行权限检查, 如以下代码所示:

```
1865         int res = mPolicy.checkAddPermission(attrs);
1866         if (res != WindowManagerImpl.ADD_OKAY) {
1867             return res;
1868         }
```

如果权限检查失败, 则会返回错误标识 res, 而执行权限的是 WindowManagerPolicy 对象 mPolicy, WindowManagerPolicy 仅仅是一个 interface, 真正的实现是 PhoneWindowManager 类。从命名上看, 这两个类的命名有点不清晰, 更合理的命名应该是:

```
WindowManagerPolicy → WindowPolicy;
PhoneWindowManager → PhoneWindowPolicy。
```

于是, 继续调用到 PhoneWindowManager 的 checkAddPermission()方法, 而该方法内部则忽略了对以上三个系统窗口的权限检查, 如以下代码所示:

```
654         switch (type) {
655             case TYPE_TOAST:
656                 // XXX right now the app process has complete control over
657                 // this... should introduce a token to let the system
658                 // monitor/control what they are doing.
659                 break;
660             case TYPE_INPUT_METHOD:
661             case TYPE_WALLPAPER:
662                 // The window manager will check these.
663                 break;
```

从源码中的注释可以看出，Android 的程序员也觉得不对 Toast 窗口检查似乎有点不安全。除了这三个窗口外，则会检查调用者是否拥有以下代码所示的两个权限。

```
case TYPE_SYSTEM_OVERLAY:
    permission = android.Manifest.permission.SYSTEM_ALERT_WINDOW;
    break;
default:
    permission = android.Manifest.permission.INTERNAL_SYSTEM_WINDOW;
```

而对于应用程序而言，是无法获取这两个权限的，关于系统中的权限声明和检查的具体过程见第16章。

Framework 中有一个系统进程 `system_process`，该进程是有权限创建系统窗口的，常见的系统窗口包括状态栏窗口、系统错误对话框等，本小节以 Toast 为例介绍系统窗口的创建。如果抛开权限问题，应用程序其实也可以创建任何系统窗口，WmS 在创建窗口时，检查完权限后，就会像创建普通窗口一样创建系统窗口。

### 8.5.1 Toast 调用流程

添加 Toast 的流程如图 8-16 所示。

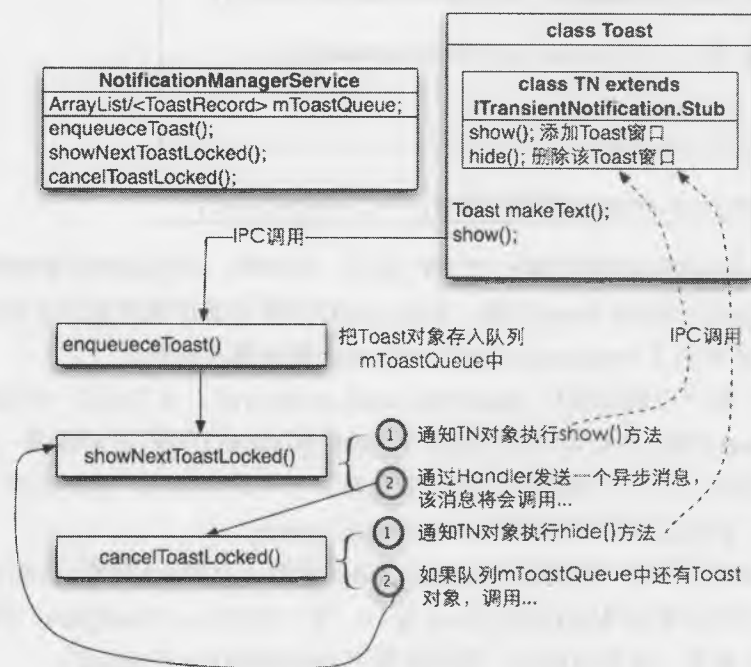


图 8-16 Toast 窗口的内部工作原理

在应用程序中，一般使用以下代码启动一个 Toast 显示。

```
Toast.makeText(context, "Hello Android", Toast.LENGTH_SHORT).show();
```

makeText() 方法用于创建一个 Toast 对象，该方法中主要是根据系统指定的 layout 文件 com.android.internal.R.layout.transient\_notification 创建一个 View 对象，并把该对象赋值给 Toast 的 mNextView 变量，如以下代码所示。这就为自定义 Toast 窗口提供了机会，当然，这仅限于 Framework 层面的定制，应用程序是无法自定义 Toast 窗口风格的。

```
Toast result = new Toast(context);

LayoutInflater inflate = (LayoutInflater)
    context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
View v = inflate.inflate(com.android.internal.R.layout.transient_notification,
    TextView tv = (TextView)v.findViewById(com.android.internal.R.id.message);
tv.setText(text);

result.mNextView = v;
result.mDuration = duration;
```

创建了 Toast 对象后，接下来调用 show() 方法，该方法则通过调用“通知管理器(Notification Manager)”的 enqueueToast()，把该 Toast 对象加入到管理器的消息队列中。如以下代码所示：

```
101     INotificationManager service = getService();
102
103     String pkg = mContext.getPackageName();
104
105     TN tn = mTN;
106
107     try {
108         service.enqueueToast(pkg, tn, mDuration);
```

通知管理器的 enqueueToast() 方法是一个 IPC 调用，在远端，该方法会往通知管理器所在的队列变量 mToastQueue 中添加客户端的 Toast 对象，所有应用程序都是通过调用该方法添加客户端的 Toast 请求的。因此，该方法中使用了 synchronized(mToastQueue) 解决重入问题。

添加完 Toast 后，接下来就会执行 showNextToastLocked() 方法。该方法第一步会从 mToastQueue 中取出最前面的一个 Toast 对象，然后调用其内部的 TN 对象的 show() 方法，该对象是一个客户端的 Binder，其 show() 方法真正完成向 WmS 添加窗口；第二步会发送一个异步消息，也就是在 Toast 显示一定时间后清除该 Toast 窗口，该消息的处理函数是 cancelToastLocked()。

cancelToastLocked() 方法第一步会取出 mToastQueue 中第一个 Toast 对象，然后调用其内部的 TN 对象的 hide() 方法，该方法会通知 WmS 移除 Toast 窗口；第二步会从 mToastQueue 中移除该 Toast 对象，并检查是否还有 Toast 对象，如果有的话，则继续调用 showNextToastLocked()。

以上流程实际上是 NotificationManager 把客户端显示 Toast 的工作进行了串行化。串行化的原因是 WmS 在给系统窗口分配层值时，不会像其他类型窗口那样去动态调整窗口的层值，而且从产品设计的

角度来讲,每一种系统要求仅有一个显示在屏幕上,所以,客户端的请求被串行化,以保证只有一个该类窗口。

### 8.5.2 Toast 添加窗口

Toast 的子类 `TN` 内部添加和隐藏窗口的方法和其他类型的基本相同,唯一不同的仅仅是所提供的 `WindowManager.LayoutParams` 参数中的 `type` 不一样而已,如以下代码所示。

```
final WindowManager.LayoutParams params = mParams;
params.height = WindowManager.LayoutParams.WRAP_CONTENT;
params.width = WindowManager.LayoutParams.WRAP_CONTENT;
params.flags = WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
    | WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE
    | WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON;
params.format = PixelFormat.TRANSLUCENT;
params.windowAnimations = com.android.internal.R.style.Animation_Toast;
params.type = WindowManager.LayoutParams.TYPE_TOAST;
params.setTitle("Toast");
```

然后调用 `mWM.addView()` 方法添加窗口。需要注意的是该 `mWM` 变量是通过调用 `WindowManagerImpl.getDefault()` 获得的,而不是 `Window.LocalWindowManager`,所以,调用 `addView()` 时不会对 `params` 的 `token` 进行检查,如以下代码所示:

```
354 mWM = WindowManagerImpl.getDefault();
373 mWM.addView(mView, mParams);
```

删除 Toast 窗口则要调用 `mWM.removeView()`, 如以下代码所示:

```
383 if (mView.getParent() != null) {
384     if (localLOGV) Log.v(
385         TAG, "REMOVE! " + mView +
386         mWM.removeView(mView);
387 }
```

## 8.6 创建窗口示例

本章前面几节介绍了添加各种类型窗口的原理,本节给出一个示例,从应用程序中添加窗口,并处理窗口的消息响应,如要添加系统窗口,则需要一定的权限,该权限可以在 `AndroidManifest.xml` 中声明。

添加子窗口的关键是要指定窗口的 `token` 值,本例假设存在一个 `Activity` 界面,其中包含一个按钮,单击后弹出一个子窗口,该窗口就是本例要添加的子窗口,因此该子窗口的 `token` 应该是 `Activity` 所对应窗口的 `token` 值,该值即为按钮的 `mAttachInfo` 变量中的 `token`。具体实现代码如下:

```

IBinder token = v.getWindowToken();
params.type = WindowManager.LayoutParams.TYPE_APPLICATION_PANEL;
params.token = token;
mWin = popup;
mWm = (WindowManager)mContext.getSystemService("window");
mWm.addView(popup, params);

```

该程序执行后，单击按钮即可弹出一个内容为 TextView 的窗口，但此时按下“Back”键，却没有任何反应。原因是该窗口没有和任何 PhoneWindow 对应，因此，所有用户消息都必须由窗口本身去处理，因此，程序员需要为该窗口增加按键消息，如以下代码所示：

```

TextView popup = new TextView(mContext);
popup.setOnKeyListener(mKeyListener);

```

mKeyListener 的定义如下：

```

OnKeyListener mKeyListener = new OnKeyListener(){
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        if(keyCode == KeyEvent.KEYCODE_BACK){
            mWm.removeView(mWin);
        }
        return false;
    }
};

```

于是，该窗口就可以响应“Back”键了。要处理更多的用户消息，则需要在 onKey()回调函数中添加更多的消息响应。

下面讨论添加一个任意的系统窗口，有朋友说，想添加一个悬浮窗口，就像 FTP 下载时屏幕右上方的那种悬浮窗口，这种窗口实际上就是一个系统窗口，添加系统窗口的关键包括两点。

第一点，要在 AndroidManifest.xml 中声明添加系统窗口的权限，代码如下所示：

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
```

第二点，为窗口指定合适的 Token 值。

下面以一个例子来说明如何添加一个系统窗口。对于系统窗口而言，一般不是由 Activity 中的事件触发导致添加，而应该是由某个系统事件引起，比如想当有系统来电时给屏幕上添加一个系统窗口，显示来电的地址；再比如当发现有 WIFI 或者 GPRS 连接时，弹出一个系统窗口动态显示网络的流量。因此，添加系统窗口一般是在一个 Service 中实现，当然，无论由什么原因引起，程序上仅仅需要一个 Context 对象即可，所以，理论上无论是在 Activity 中还是在 Service 中都可以添加系统窗口。

下面给出笔者经过测试过的添加系统窗口的代码：

```
ctx = getApplicationContext();
Button hook = new Button(ctx);
hook.setText("Hello window");
hook.setOnClickListener(mHookClickListener);
WindowManager.LayoutParams lp = new WindowManager.LayoutParams(
    150,
    100,
    WindowManager.LayoutParams.TYPE_PRIORITY_PHONE,
    WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE
    | WindowManager.LayoutParams.FLAG_TOUCHABLE_WHEN_WAKING,
    PixelFormat.RGBX_8888);
lp.gravity = Gravity.CENTER_HORIZONTAL | Gravity.BOTTOM;
lp.setTitle("title");
lp.token = mToken;
WindowManager wm = (WindowManager)ctx.getSystemService("window");
hook.setTag(new String(inNumber));
wm.addView(hook, lp);
mHook = hook;
```

这段代码的关键在于给该窗口赋予合适的 token 值，这段代码中的 `mToken = new Binder()`，为什么要创建一个新的 Binder 对象，而不是使用其父窗口的 token 值呢？原因是，在 WmS 中，当进行窗口重新排序时，就算一个系统窗口，如果其 token 值是其父窗口的 token，那么当其父窗口隐藏后，该系统窗口也会被隐藏，有些用户不了解这一点，从而在 Activity 中添加系统窗口时，使用父窗口的 token 值，就像前面所讲的 `v.getWindowToken()` 作为系统窗口的 token，这样做的结果是，当该 Activity 没有停止时，系统窗口会处于屏幕最上方，甚至可以覆盖掉系统状态栏窗口，而一旦该 Activity 窗口处于隐藏时，新加的系统窗口也会被隐藏起来。在 WmS 内部，当判断系统窗口的 token 没有父窗口时，才会把该窗口的层值调整到屏幕上方，使之显示出来。

理解清楚了添加窗口的过程后，添加任意需要的、不同类型的窗口，并为窗口添加事件响应就是一件很轻松的事情了。



## 第9章 Framework 的启动过程

本书第1章中介绍过 Linux 系统的启动过程,在该过程的最后,内核将读取 init.rc 文件,并启动该文件中定义的各种服务程序。由于 Android 系统相对于 Linux 内核而言仅仅是一个 Linux 应用程序而已,因此,该程序也是在 init.rc 中被声明,从而当 Linux 内核启动后能够接着运行 Android 内核。本章将从 init.rc 文件开始,继续介绍 Android 内核的启动过程,以及该过程中相关的重要模块的交互逻辑。

### 9.1 Framework 运行环境综述

任何系统启动过程的本质都是要建立一套系统运行所需的环境,因此,本节首先介绍 Framework 的运行环境组成,然后再具体分析环境中所需子模块的启动过程。Framework 的运行环境如图 9-1 所示。

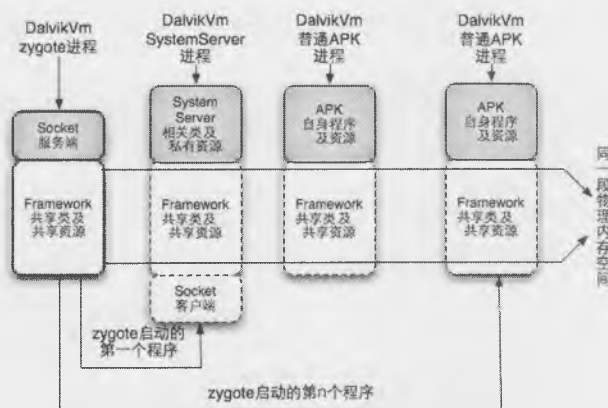


图 9-1 Dalvik 虚拟机进程间的关系



系统中运行的第一个 Dalvik 虚拟机程序叫做 `zygote`，该名称的意义是“一个卵”，因为接下来的所有 Dalvik 虚拟机进程都是通过这个“卵”孵化出来的。

`zygote` 进程中包含两个主要模块，分别如下：

- **Socket 服务端。**该 Socket 服务端用于接收启动新的 Dalvik 进程的命令。
- **Framework 共享类及共享资源。**当 `zygote` 进程启动后，会装载一些共享的类及资源，其中共享类是在 `preload-classes` 文件中被定义，共享资源是在 `preload-resources` 中被定义。因为 `zygote` 进程用于孵化出其他 Dalvik 进程，因此，这些类和资源装载后，新的 Dalvik 进程就不需要再装载这些类和资源了，这也就是所谓的共享。

`zygote` 进程对应的具体程序是 `app_process`，该程序存在于 `system/bin` 目录下，启动该程序的指令是在 `init.rc` 中进行配置的。

`zygote` 孵化出的第一个 Dalvik 进程叫做 `SystemServer`，`SystemServer` 仅仅是该进程的别名，而该进程具体对应的程序依然是 `app_process`，因为 `SystemServer` 是从 `app_process` 中孵化出来的。

`SystemServer` 中创建了一个 Socket 客户端，并有 `AmS` 负责管理该客户端，之后所有的 Dalvik 进程都将通过该 Socket 客户端间接被启动。当需要启动新的 APK 进程时，`AmS` 中会通过该 Socket 客户端向 `zygote` 进程的 Socket 服务端发送一个启动命令，然后 `zygote` 会孵化出新的进程。

从系统架构的角度来讲，就在于此即先创建一个 `zygote`，并加载共享类和资源，然后通过该 `zygote` 去孵化新的 Dalvik 进程，该架构的特点有两个。

- 每一个进程都是一个 Dalvik 虚拟机，而 Dalvik 虚拟机是一种类似于 Java 虚拟机的程序，并且从开发的过程来看，与标准的 Java 程序开发基本一致。因此对于程序员来讲，不需要学习新的语言，并可以使用 Java 程序在过去几十年中已经成熟的各种类库资源。
- `zygote` 进程预先会装载共享类和共享资源，这些类及资源实际上就是 SDK 中定义的大部分类和资源。因此，当通过 `zygote` 孵化出新的进程后，新的 APK 进程只需要去装载 APK 自身包含的类和资源即可，这就有效地解决了多个 APK 共享 Framework 资源的问题。

## 9.2 Dalvik 虚拟机相关的可执行程序

在 Android 源码中，大家会发现好几处和 Dalvik 这个概念相关的可执行程序，正确区分这些可执行程序的区别将有助于理解 Framework 内部结构。这些可执行程序的名称和源码路径如表 9-1 所示。

表 9-1 和虚拟机相关的源码

名 称	源码路径
<code>dalvikvm</code>	<code>dalvik/dalvikvm</code>
<code>dvz</code>	<code>dalvik/dvz</code>
<code>app_process</code>	<code>frameworks/base/cmds/app_process</code>

下面将分别介绍这些可执行程序的作用。



### 9.2.1 dalvikvm

当 Java 程序运行时，都是由一个虚拟机来解释 Java 的字节码，它将这些字节码翻成本地 CPU 的指令码，然后执行。对 Java 程序而言，负责解释并执行的就是一个虚拟机，而对于 Linux 而言，这个进程只是一个普通的进程，它与一个只有一行代码的 Hello World 可执行程序无本质区别。所以启动一个虚拟机的方法就跟启动任何一个可执行程序的方法是相同的，那就是在命令行下输入可执行程序的名称，并在参数中指定要执行的 Java 类。

`dalvikvm` 的作用就是创建一个虚拟机并执行参数中指定的 Java 类，下面以一个例子来说明该程序的使用方法。

首先新建一个 `Foo.java` 文件，如以下代码所示：

```
class Foo {
    public static void main(String[] args) {
        System.out.println("Hello dalvik");
    }
}
```

然后编译该文件，并生成 Jar 文件，如以下代码所示：

```
$ javac Foo.java
$ PATH=/Users/keyd/android/out/host/darwin-x86/bin:$PATH
$ dx --dex --output=foo.jar Foo.class
```

`dx` 工具的作用是将 `.class` 转换为 `dex` 文件，因为 Dalvik 虚拟机所执行的程序不是标准的 Jar 文件，而是将 Jar 文件经过特别的转换以提高执行效率，而转换后的文件就是 `dex` 文件。`dx` 工具是 Android 源码的一部分，其路径是在 `out` 目录下，因此在执行 `dx` 之前，需要添加该路径。

`dx` 执行时，`--output` 参数用于指定 Jar 文件的输出路径，注意该 Jar 文件内部包含已经不是纯粹的 `.class` 文件，而是 `dex` 格式文件，Jar 仅仅是 `zip` 包。

生成了该 Jar 包后，就可以把该 Jar 包 `push` 到设备中，并执行，如以下代码所示：

```
$ adb push foo.jar /data/app
$ adb shell dalvikvm -cp /data/app/foo.jar Foo
Hello dalvik
```

以上命令首先将该 Jar 包 `push` 到 `/data/app` 目录下，因为该目录一般用于存放应用程序，接着使用 `adb shell` 执行 `dalvikvm` 程序。`dalvikvm` 的执行语法如下：

```
dalvikvm -cp 类路径 类名
```

从这里也可以感觉到，`dalvikvm` 的作用就像在 PC 上执行 Java 程序一样。

### 9.2.2 dvz

`dvz` 的作用是从 `zygote` 进程中孵化出一个新的进程，新的进程也是一个 Dalvik 虚拟机。该进程与

dalvikvm 启动的虚拟机相比，区别在于该进程中已经预装了 Framework 的大部分类和资源，下面以一个具体的例子来看 dvz 的使用方法。

首先在 Eclipse 下新建一个 APK 项目，包名称为 com.haiii.android.helloapk，默认的 Activity 名称为 Welcome，其内容如下：

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

    public static void main(String[] args) {
        System.out.println("Hello dalvik");
    }
}
```

该段代码是大家非常熟悉的 Hello Android 代码，唯一的区别在于增加了一个 static main() 函数，因为该函数将作为 Welcome 类的入口。

接下来，将生成的 APK 文件 push 到 /data/app 目录下，然后运行 Welcome 类，如以下代码所示：

```
# dvz -classpath /data/app/HelloApk.apk com.haiii.android.helloapk.Welcome
Hello dalvik
In mgmain JNI_OnLoad
```

dvz 的语法如下：

```
dvz -classpath 包名称 类名
```

讲到这里，有的读者可能猜想，是否可以在 main() 函数内部构造一个 Welcome 对象，从而可以达到运行该 APK 的目的？答案是否定的，因为 Welcome 类并不是该应用程序的入口类，在后面的章节中，大家将看到，一个 APK 的入口类是 ActivityThread 类，Activity 类仅仅是被回调的类，因此不可以通过 Activity 类来启动一个 APK，dvz 工具仅仅用于 Framework 开发过程的调试。

### 9.2.3 app\_process

以上讲述的 dalvikvm 和 dvz 是通用的两个工具，然而 Framework 在启动时需要加载运行两个特定 Java 类，一个是 ZygoteInit.java，另一个是 SystemServer.java。为了便于使用，系统才提供了一个 app\_process 进程，该进程会自动运行这两个类，从这个角度来讲，app\_process 的本质就是使用 dalvikvm 启动 ZygoteInit.java，并在启动后加载 Framework 中的大部分类和资源。

下面就来对比一下 app\_process 和 dalvikvm 的主要执行过程。首先来看 dalvikvm，其源码文件在 dalvik/dalvikvm/Main.c 中，该源码中的关键代码有两处。

第一处是创建一个 vm 对象，如以下代码所示：

```

209  /*
210   * Start VM. The current thread becomes the main thread of the VM.
211   */
212   if (JNI_CreateJavaVM(&vm, &env, &initArgs) < 0) {
213       fprintf(stderr, "Dalvik VM init failed (check log file)\n");
214       goto bail;
215   }

```

该段代码通过调用 JNI\_CreateJavaVM() 同时创建了 JavaVm 对象和 JNIEnv 对象，这两个对象的定义如下：

```

JNIEnvExt* pEnv = NULL;
JavaVMExt* pVM = NULL;

```

该函数的参数是“指针的指针”类型，其原型如下：

```

jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args)

```

创建好了 JavaVm 对象后，就可以使用该对象去加载指定的类了，如以下关键代码所示：

```

249   startClass = (*env)->FindClass(env, slashClass);
250   if (startClass == NULL) {
251       fprintf(stderr, "Dalvik VM unable to locate class '%s'\n", slashClass);
252       goto bail;
253   }
254
255   startMeth = (*env)->GetStaticMethodID(env, startClass,
256       "main", "(Ljava/lang/String;)V");
257   if (startMeth == NULL) {
258       fprintf(stderr, "Dalvik VM unable to find static main(String[]) in '%s'",
259           slashClass);
260       goto bail;
261   }
262
263   /*
264    * Make sure the method is public. JNI doesn't prevent us from calling
265    * a private method, so we have to check it explicitly.
266    */
267   if (!methodIsPublic(env, startClass, startMeth))
268       goto bail;
269
270   /*
271    * Invoke main().
272    */
273   (*env)->CallStaticVoidMethod(env, startClass, startMeth, strArray);

```

该段代码首先通过调用 FindClass() 找到指定的 class 文件，然后调用 GetStaticMethodID() 找到 main() 函数，最后调用 CallStaticVoidMethod 执行该 main() 函数。

接下来再来看 app\_process 中是如何创建虚拟机并执行指定的 class 文件的。其源文件在

frameworks/base/cmds/app\_main.cpp 中, 该文件中的关键代码有两处, 第一处是先创建一个 AppRuntime 对象, 如以下代码所示:

```
130 AppRuntime runtime;
```

第二处是调用 runtime 的 start() 方法启动指定的 class, 如以下代码所示:

```
153 if (0 == strcmp("--zygote", arg)) {
154     bool startSystemServer = (i < argc) ?
155         strcmp(argv[i], "--start-system-server") == 0 : false;
156     setArgv0(argv0, "zygote");
157     set_process_name("zygote");
158     runtime.start("com.android.internal.os.ZygoteInit",
159         startSystemServer);
```

系统中只有一处使用 app\_process, 那就是在 init.rc 中, 使用时参数包含了 --zygote 及 --start-system-server, 因此, 这里仅分析包含这两个参数的情况。start() 方式是 AppRuntime 类的成员函数, 而 AppRuntime 是在该文件中定义的一个应用类, 其父类是 AndroidRuntime, 该类的实现文件在 frameworks/base/core/jni/AndroidRuntime.cpp 中。start() 函数中首先调用 startVm() 创建一个 vm 对象, 然后就和 dalvikvm 一样先找到 Class(), 再执行 class 中 main() 函数。

startVm() 函数使用以下代码创建 vm 对象:

```
865 if (JNI_CreateJavaVM(pJavaVM, pEnv, &initArgs) < 0) {
866     LOGE("JNI_CreateJavaVM failed\n");
867     goto bail;
868 }
```

由此可以看出, app\_process 和 dalvikvm 在本质上是相同的, 唯一的区别就是 app\_process 可以指定一些特别的参数, 这些参数有利于 Framework 启动特定的类, 并进行一些特别的系统环境参数设置。有兴趣的读者可以依照以上流程详细分析其内部差别。

## 9.3 zygote 的启动

前面小节介绍了 Framework 的运行环境, 以及 Dalvik 虚拟机的相关启动方法, zygote 进程是所有 APK 应用进程的父进程, 接下来就详细介绍 zygote 进程的内部启动过程。

### 9.3.1 在 init.rc 中配置 zygote 启动参数

init.rc 存在于设备的根目录下, 读者可以使用 adb pull /init.rc ~/Desktop 命令取出该文件, 文件中和 zygote 相关的配置信息如下:

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
```

```
socket zygote stream 666
onrestart write /sys/android_power/request_state wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

首先第一行中使用 `service` 指令告诉操作系统将 `zygote` 程序加入到系统服务中, `service` 的语法如下:

```
service service_name 可执行程序的路径 可执行程序自身所需的参数列表
```

此处的服务被定义为 `zygote`, 理论上讲该服务的名称可以是任意的。可执行程序的路径正是 `/system/bin/app_process`, 也就是前面所讲的 `app_process`, 参数一共包含四个, 分别如下:

- `-Xzygote`, 该参数将作为虚拟机启动时所需要的参数, 是在 `AndroidRuntime.cpp` 类的 `startVm()` 函数中调用 `JNI_CreateJavaVM()` 时被使用的。
- `/system/bin`, 代表虚拟机程序所在目录, 因为 `app_process` 完全可以不和虚拟机在同一个目录, 而在 `app_process` 内部的 `AndroidRuntime` 类内部需要知道虚拟机所在的目录。
- `--zygote`, 指明以 `ZygoteInit` 类作为虚拟机执行的入口, 如果没有 `--zygote` 参数, 则需要明确指定需要执行的类名。
- `--start-system-server`, 仅在指定 `--zygote` 参数时才有效, 意思是告知 `ZygoteInit` 启动完毕后孵化出第一个进程 `SystemServer`。

接下来的配置命令 `socket` 用于指定该服务所使用到的 `socket`, 后面的参数依次是名称、类型、端口地址。

`onrestart` 命令指定该服务重启的条件, 即当满足这些条件后, `zygote` 服务就需要重启, 这些条件一般是一些系统异常条件。

### 9.3.2 启动 Socket 服务端

当 `zygote` 服务从 `app_process` 开始启动后, 会启动一个 `Dalvik` 虚拟机, 而虚拟机执行的第一个 `Java` 类就是 `ZygoteInit.java`, 因此接下来的过程就从 `ZygoteInit` 类的 `main()` 函数开始说起。 `main()` 函数中做的第一个重要工作就是启动一个 `Socket` 服务端口, 该 `Socket` 端口用于接收启动新进程的命令。

启动 `Socket` 服务端口是在静态函数 `registerZygoteSocket()` 中完成的, 如以下代码所示:

```
private static void registerZygoteSocket() {
    if (sServerSocket == null) {
        int fileDesc;
        try {
            String env = System.getenv(ANDROID_SOCKET_ENV);
            fileDesc = Integer.parseInt(env);
            ...
            try {
                sServerSocket = new LocalServerSocket(
```

```
createFileDescriptor(fileDesc));
```

在该段代码中，首先调用 `System.getenv()` 获取系统为 `zygote` 进程分配的 `Socket` 文件描述符号，然后再调用 `createFileDescriptor()` 创建一个真正的文件描述符，最后以该描述符为参数，构造了一个 `LocalServerSocket` 对象。

关于 `Socket` 编程的基础知识，读者可以参考《UNIX Networking Programming》一书，作者为 W.Richard Stevens。这里要说明的是，在 Linux 系统中，所有的系统资源都可以看成是文件，甚至包括内存和 CPU，因此，像标准的磁盘文件或者网络 `Socket` 自然也被认为是文件，这就是为什么 `LocalServerSocket` 构造函数的参数是一个文件描述符。

`Socket` 编程中有两种方式去触发 `Socket` 数据读操作。一种是使用 `listen()` 监听某个端口，然后调用 `read()` 去从这个端口上读数据，这种方式被称为阻塞式读操作，因为当端口没有数据时，`read()` 函数将一直等待，直到数据准备好后才返回；另一种是使用 `select()` 函数将需要监测的文件描述符作为 `select()` 函数的参数，然后当该文件描述符上出现新的数据后，自动触发一个中断，然后在中断处理函数中再去读指定文件描述符上的数据，这种方式被称为非阻塞式读操作。`LocalServerSocket` 中使用的正是后者，即非阻塞读操作。

当 `LocalServerSocket` 端口准备好后，`main()` 函数中调用 `runSelectLoopMode()` 进入非阻塞读操作，该函数中首先将 `sServerSocket` 加入到被监测的文件描述符列表中，如以下代码所示：

```
654      FileDescriptor[] fdArray = new FileDescriptor[4];
655
656      fds.add(sServerSocket.getFileDescriptor());
```

然后在 `while(true)` 循环中将该文件描述符添加到 `select` 的列表中，并调用 `ZygoteConnection` 类的 `runOnce()` 函数处理每一个 `Socket` 接收到的命令，如以下代码所示：

```
try {
    fdArray = fds.toArray(fdArray);
    index = selectReadable(fdArray);
} catch (IOException ex) {
    throw new RuntimeException("Error in select()", ex);
}

if (index < 0) {
    throw new RuntimeException("Error in select()");
} else if (index == 0) {
    ZygoteConnection newPeer = acceptCommandPeer();
    peers.add(newPeer);
    fds.add(newPeer.getFileDescriptor());
} else {
    boolean done;
    done = peers.get(index).runOnce();
```

```

    if (done) {
        peers.remove(index);
        fds.remove(index);
    }
}

```

selectReadable()函数的返回值有三种。一种是-1,代表着内部错误;第二种是 0,代表着没有可处理的连接,因此会以 Socket 服务端口重新建立一个 ZygoteConnection 对象,并等待客户端的请求;第三种是大于 0,代表着还有没有处理完的连接请求,因此需要先处理该请求,而暂时不需要建立新的连接等待。

runOnce()函数的核心代码是基于 zygote 进程孵化出新的应用进程,如以下代码所示:

```

212     pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
213                                   parsedArgs.gids, parsedArgs.debugFlags, rlimits);

```

关于 folk 的概念将在后面小节中介绍。

以上介绍的是 Socket 的服务端,而在 SystemServer 进程中则会创建一个 Socket 客户端,具体的实现代码是在 Process.java 类中,而调用 Process 类是在 AmS 类中的 startProcessLocked()函数中,如以下代码所示,关于该函数的调用实际参照第 10 章 AmS 原理。

```

1875     int pid = Process.start("android.app.ActivityThread",
1876                             mSimpleProcessManagement ? app.processName : null, uid, uid,
1877                             gids, debugFlags, null);

```

而 start()函数内部又调用了静态函数 startViaZygote(),该函数的实体正是使用一个本地 Socket 向 zygote 中的 Socket 发送进行启动命令,其执行流程如图 9-2 所示。

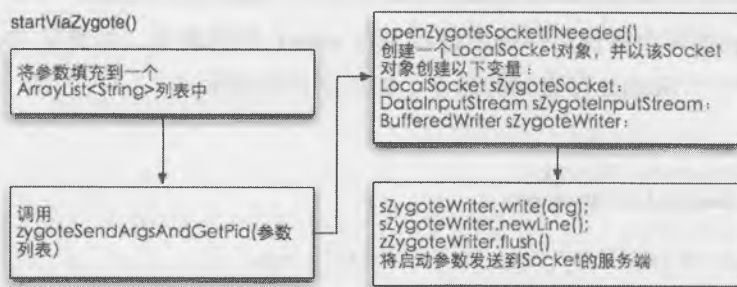


图 9-2 startViaZygote()函数的执行过程

该流程的主要过程就是将 startViaZygote()的函数参数转换为一个 ArrayList<String>列表,然后再构造出一个 LocalSocket 本地 Socket 接口,并通过该 LocalSocket 对象构造出一个 BufferedWriter 对象,最后通过该对象将 ArrayList<String>列表中的参数传递给 zygote 中的 LocalServerSocket 对象,而在 zygote 端,就会调用 Zygote.forkAndSpecialize()函数孵化出一个新的应用进程。



### 9.3.3 加载 preload-classes

在 `ZygoteInit` 类的 `main()` 函数中, 创建完 `Socket` 服务端后还不能立即孵化新的进程, 因为这个“卵”中还没有必须“核酸”, 这个“核酸”就是指预装的 `Framework` 大部分类及资源。

预装的类列表是在 `framework.jar` 中的一个文本文件列表, 名称为 `preload-classes`, 该列表的原始定义在 `frameworks/base/preload-classes` 文本文件中, 而该文件又是通过 `frameworks/base/tools/preload/WritePreloadedClassFile.java` 类生成的。产生 `preload-classes` 的方法是在 `Android` 根目录下执行以下命令:

```
$java -Xss512M -cp /path/to/preload.jar WritePreloadedClassFile /path/to/.compiled
1517 classes were loaded by more than one app.
Added 147 more to speed up applications.
1664 total classes will be preloaded.
Writing object model...
Done!
```

在该命令中, `/path/to/preload.jar` 是指 `out/host/darwin-x86/framework/preload.jar`, 该 `Jar` 是由 `frameworks/base/tools/preload` 子项目编译而成的。

`/path/to/.compiled/` 是指 `frameworks/base/tools/preload` 目录下的那几个 `.compiled` 文件。

参数 `-Xss` 用于执行该程序所需要的 `Java` 虚拟机栈大小, 此处使用 `512MB`, 默认的大小不能满足该程序的运行, 会抛出 `java.lang.StackOverflowError` 错误信息。

`WritePreloadedClassFile` 是要执行的具体类。

执行完以上命令后, 会在 `frameworks/base` 目录下产生 `preload-classes` 文本文件。从该命令的执行情况来看, 预装的 `Java` 类信息包含在 `.compiled` 文件中, 而这个文件却是一个二进制文件, 尽管我们目前能够确知如何产生 `preload-classes`, 但却无法明确这个 `.compiled` 文件是如何产生的, 一个可能的假设如下:

在 `Android` 项目组内部可能会存在一个测试项目, 该项目一旦运行, 就会装载一些 `Java` 类。当然, 这些 `Java` 类是测试项目中的程序代码主动装载的, 而这些程序代码被认为是大多数 `Android` 程序运行时都会执行的代码。一旦该运行环境建立后, `Dalvik` 虚拟机内存中就记录了所有被装载的 `Java` 类, 然后该测试项目会使用一个特别的工具从虚拟机内存中读取所有装载过的类信息, 并生成 `compiled` 文件。当然, 这只是一种假设。

在 `Android` 源码编译的时候, 会最终把 `preload-classes` 文件打包到 `framework.jar` 中, 关于其详细过程参见本书 `Android` 源码编译的相关章节。

有了这个列表后, `ZygoteInit` 中通过调用 `preloadClasses()` 完成装载这些类。装载的方法很简单, 就是读取 `preload-classes` 列表中的每一行, 因为每一行代表了一个具体的类, 然后调用 `Class.forName()` 装载目标类, 如下代码所示。在装载的过程中, 忽略以 `#` 开始的目标类, 并忽略换行符及空格。



```

286         while ((line = br.readLine()) != null) {
287             // Skip comments and blank lines.
288             line = line.trim();
289             if (line.startsWith("#") || line.equals("")) {
290                 continue;
291             }
292
293             try {
294                 if (Config.LOGV) {
295                     Log.v(TAG, "Preloading " + line + "...");
296                 }
297                 Class.forName(line);

```

### 9.3.4 加载 preload-resources

preload-resources 是在 frameworks/base/core/res/res/values/arrays.xml 中被定义的，包含两类资源，一类是 drawable 资源，另一类是 color 资源，如以下代码所示：

```

<array name="preloaded_drawables">
    <item>@drawable/sym_def_app_icon</item>
    ...
</array>

<array name="preloaded_color_state_lists">
    <item>@color/hint_foreground_dark</item>
    ...
</array>

```

而加载这些资源是在 preloadResources() 函数中完成的，该函数中分别调用 preloadDrawables() 和 preloadColorStateLists() 加载这两类资源。加载的原理很简单，就是把这些资源读出来放到一个全局变量中，只要该类对象不被销毁，这些全局变量就会一直保存。

保存 Drawable 资源的全局变量是 mResources，该变量的类型是 Resources 类，由于该类内部会保存一个 Drawable 资源列表，因此，实际上缓存这些 Drawable 资源是在 Resources 内部；保存 Color 资源的全局变量也是 mResources，同样，Resources 类内部也有一个 Color 资源的列表。

关于 Resources 内部如何保存这些资源，请参照资源访问章节。

### 9.3.5 使用 fork 启动新的进程

fork 是 Linux 系统的一个系统调用，其作用是复制当前进程，产生一个新的进程。新进程将拥有和原始进程完全相同的进程信息，除了进程 id 不同。进程信息包括该进程所打开的文件描述符列表、所分配的内存等。当新进程被创建后，两个进程将共享已经分配的内存空间，直到其中一个需要向内存中

写入数据时，操作系统才负责复制一份目标地址空间，并将要写的数据写入到新的地址中，这就是所谓的 copy-on-write 机制，即“仅当写的时候才复制”，这种机制可以最大限度地在多个进程中共享物理内存。

第一次接触 folk 的读者可能觉得奇怪，为什么要复制进程呢？在大家熟悉的 Windows 操作系统中，一个应用程序一般对应一个进程，如果说要复制进程，可能的结果就是从计算器程序复制出一个 Office 程序，这听起来似乎很不合理。要立即复制进程就需要首先了解进程的启动过程。

在所有的操作系统中，都存在一个程序装载器，程序装载器一般会作为操作系统的一部分，并由所谓的 Shell 程序调用。当内核启动后，Shell 程序会首先启动。常见的 Shell 程序包含两大类，一类是命令行界面，另一类是窗口界面，Windows 系统中 Shell 程序就是桌面程序，Ubuntu 系统中的 Shell 程序就是 GNOME 桌面程序。Shell 程序启动后，用户可以双击桌面图标启动指定的应用程序，而在操作系统内部，启动新的进程包含三个过程。

第一个过程，内核创建一个进程数据结构，用于表示将要启动的进程。

第二个过程，内核调用程序装载器函数，从指定的程序文件读取程序代码，并将这些程序代码装载到预先设定的内存地址。

第三个过程，装载完毕后，内核将程序指针指向到目标程序地址的入口处开始执行指定的进程。当然，实际的过程会考虑更多的细节，不过大致思路就是这么简单。

在一般情况下，没有必要复制进程，而是按照以上三个过程创建新进程，但当满足以下条件时，则建议使用复制进程：即两个进程中共享了大量的程序。

举个例子，去澳大利亚看袋鼠和去澳大利亚看考拉，这是两个进程，但完成这两个进程的大多数任务都是相同的，即先订机票，然后带照相机，再坐地铁到首都机场，最后再坐 14 个小时的飞机到澳大利亚，到了之后唯一不同就是看考拉和袋鼠。为了更有效地完成这两个任务，可以先雇佣一个精灵进程，让它订机票、带相机、坐地铁、乘飞机，一直到澳大利亚后，从这个精灵进程中复制出两个进程，一个去看考拉，另一个去看袋鼠。如果你愿意，还可以去悉尼歌剧院，这就是进程的复制，其好处是节省了大量共享的内存。

由于 folk() 函数是 Linux 的系统调用，Android 中的 Java 层仅仅是对该调用进行了 JNI 封装而已，因此，接下来以一段 C 代码来介绍 folk() 函数的使用，以便大家对该函数有更具体的认识。

```
/**
 *FileName: MyFolk.c
 */
#include <sys/types.h>
#include <unistd.h>
int main(){
    pid_t pid;
    printf("pid = %d, Take camera, by subway, take air! \n", getpid());
    pid = folk();
    if(pid > 0){
        printf("pid=%d, 我是精灵! \n", getpid());
        pid = folk();
        if(!pid) printf("pid=%d, 去看考拉! \n", getpid());
    }
}
```

```

else if (!pid) printf("pid=%d, 去看袋鼠! \n", getpid());
else if (pid == -1) perror("folk");
getchar();
}

```

以上代码的执行结果如下:

```

$ ./MyFolk.bin
pid = 3927, Take camera, by subway, take air!
pid=3927, 我是精灵!
pid=3929, 去看袋鼠!
pid=3930, 去看考拉!

```

folk()函数的返回值与普通函数调用完全不同。当返回值大于 0 时,代表的是父进程;当等于 0 时,代表的是被复制的进程。换句话说,父进程和子进程的代码都在该 C 文件中,只是不同的进程执行不同的代码,而进程是靠 folk()的返回值进行区分的。

由以上执行结果可以看出,第一次调用 folk()时复制了一个“看袋鼠”进程,然后在父进程中再次调用 folk()复制了“看考拉”的进程,三者都有各自不同的进程 id。

zygote 进程就是本例中的“精灵进程”,那些“拿相机、坐地铁、乘飞机”的操作就是 zygote 进程中加载的 preload-classes 类具备的功能。

ZygoteInit.java 中复制新进程是通过在 runSelectLoopMode()函数中调用 ZygoteConnection 类的 runOnce()函数完成的,而该函数中则调用了以下代码用于复制一个新的进程。

```

212 pid = Zygote.forkAndSpecialize(parsedArgs.uid, parsedArgs.gid,
213     parsedArgs.gids, parsedArgs.debugFlags, rlimits);

```

forkAndSpecialize()函数是一个 native 函数,其内部的执行原理和上面的 C 代码类似。

当新进程被创建好后,还需要做一些“善后”工作。因为当 zygote 复制新进程时,已经创建了一个 Socket 服务端,而这个服务端是不应该被新进程使用的,否则系统中会有多个进程接收 Socket 客户端的命令。因此,新进程被创建好后,首先需要在新进程中关闭该 Socket 服务端,并调用新进程中指定的 Class 文件的 main()函数作为新进程的入口点。而这些正是在调用 forkAndSpecialize()函数后根据返回值 pid 完成的,如以下代码所示:

```

223 if (pid == 0) {
224     // in child
225     handleChildProc(parsedArgs, descriptors, newStderr);
226     // should never happen
227     return true;
228 } else { /* pid != 0 */
229     // in parent...pid of < 0 means failure
230     return handleParentProc(pid, descriptors, parsedArgs);
231 }

```

pid 等于 0 时,代表的是子进程,handleChildProc()函数中的关键代码如下,首先是关于 Socket 服务端。

```

696         } else {
697             closeSocket();
698             ZygoteInit.closeServerSocket();

```

接着从指定 Class 文件的 main()函数处开始执行, 如以下代码所示:

```

742         try {
743             ZygoteInit.invokeStaticMain(cloader, className, mainArgs);

```

至此, 新进程就完全脱离了 zygote 进程的孵化过程, 成为一个真正的应用进程。

## 9.4 SystemServer 进程的启动

SystemServer 进程是 zygote 孵化出的第一个进程, 该进程是从 ZygoteInit.java 的 main()函数中调用 startSystemServer()开始的。与启动普通进程的差别在于, zygote 类为启动 SystemServer 提供了专门的函数 startSystemServer(), 而不是使用标准的 forAndSpecilize()函数, 同时, SystemServer 进程启动后首先要做的事情和普通进程也有所差别。

startSystemServer()函数的关键代码有三处。

第一处, 定义了一个 String[]数组, 数组中包含了要启动的进程的相关信息, 其中最后一项指定新进程启动后装载的第一个 Java 类, 此处即为 com.android.server.SystemServer 类, 如以下代码所示:

```

511     String args[] = {
512         "--setuid-1000",
513         "--setgid-1000",
514         "--setgroups-1001,1002,1003,1004,1005",
515         "--capabilities-130104352,130104352",
516         "--runtime-init",
517         "--nice-name=system_server",
518         "com.android.server.SystemServer",
519     };

```

第二处, 调用 forkSystemServer()从当前的 zygote 进程孵化出新的进程, 如以下代码所示。该函数是一个 native 函数, 其作用与 folkAndSpecilize()相似。

```

537     pid = Zygote.forkSystemServer(
538         parsedArgs.uid, parsedArgs.gid,
539         parsedArgs.gids, debugFlags, null,
540         parsedArgs.permittedCapabilities,
541         parsedArgs.effectiveCapabilities);

```

第三处, 新进程启动后, 首先执行的代码如下:

```

546     /* For child process */
547     if (pid == 0) {
548         handleSystemServerProcess(parsedArgs);
549     }

```

在 handleSystemServerProcess()函数中主要完成两件事情, 第一是关闭 Socket 服务端, 第二是执行

`com.android.server.SystemServer` 类的 `main()` 函数。除了这两个主要事情外，还做了一些额外的运行环境配置，这些配置主要在 `commonInit()` 和 `zygoteInitNative()` 两个函数中完成。有兴趣的读者可以仔细分析这两个函数的内部实现。

一旦 `SystemServer` 的进程环境配置好后，就从 `SystemServer` 类的 `main()` 函数中开始运行，本节主要介绍该类的内部执行过程。

### 9.4.1 启动各种系统服务线程

`SystemServer` 进程在 Android 的运行环境中扮演了“神经中枢”的作用，APK 应用中能够直接交互的大部分系统服务都在该进程中运行，常见的比如 `WindowManagerServer`（`Wms`）、`ActivityManagerSystemService`（`AmS`）、`PackageManagerServer`（`PmS`）等，这些系统服务都是以一个线程的方式存在于 `SystemServer` 进程中。下面就来介绍到底都有哪些服务线程，及其启动的顺序。

`SystemServer` 的 `main()` 函数首先调用的是 `init1()` 函数，这是一个 `native` 函数，内部会进行一些与 Dalvik 虚拟机相关的初始化工作。该函数执行完毕后，其内部会调用 Java 端的 `init2()` 函数，这就是为什么 Java 源码中没有引用 `init2()` 的地方，主要的系统服务都是在 `init2()` 函数中完成的。

该函数首先创建了一个 `ServerThread` 对象，该对象是一个线程，然后直接运行该线程，如以下代码所示：

```
626 public static final void init2() {  
627     Slog.i(TAG, "Entered the Android system server!");  
628     Thread thr = new ServerThread();  
629     thr.setName("android.server.ServerThread");  
630     thr.start();  
631 }
```

于是，从 `ServerThread` 的 `run()` 方法内部开始真正启动各种服务线程。

基本上每个服务都有对应的 Java 类，从编码规范的角度来看，启动这些服务的模式可归类为三种，如图 9-3 所示。



图 9-3 不同服务的启动方式

- 模式一是指直接使用构造函数构造一个服务，由于大多数服务都对应一个线程，因此，在构造函数内部就会创建一个线程并自动运行。
- 模式二是指服务类会提供一个 `getInstance()` 方法，通过该方法获取该服务对象，这样的好处是保证系统中仅包含一个该服务对象。
- 模式三是指从服务类的 `main()` 函数中开始执行。

无论以上何种模式，当创建了服务对象后，有时可能还需要调用该服务类的 `init()` 或者 `systemReady()` 函数以完成该对象的启动，当然这些都是服务类内部自定义的。为了区分以上启动的不同，以下采用一种新的方式描述该启动过程。比如当一个服务对象是通过模式一创建，并调用 `init()` 完成该服务的启动，我们就用模式 1.2 表示；如果构造函数返回后就已经启动，而无须任何其他调用，即什么都不做（nothing），我们就用模式 1.1 表示。

表 9-2 列出了 `SystemService` 中所启动的所有服务，以及这些服务的启动模式。

表 9-2 `SystemService` 中启动服务列表

服务类名称	作用描述	启动模式
<code>EntropyService</code>	提供伪随机数	1.0
<code>PowerManagerService</code>	电源管理服务	1.2/3
<code>ActivityManagerService</code>	最核心的服务之一，管理 Activity	自定义
<code>TelephonyRegistry</code>	通过该服务注册电话模块的事件响应，比如重启、关闭、启动等	1.0
<code>PackageManagerService</code>	程序包管理服务	3.3
<code>AccountManagerService</code>	账户管理服务，是指联系人账户，而不是 Linux 系统的账户	1.0
<code>ContentService</code>	<code>ContentProvider</code> 服务，提供跨进程数据交换	3.0
<code>BatteryService</code>	电池管理服务	1.0
<code>LightsService</code>	自然光强度感应传感器服务	1.0
<code>VibratorService</code>	震动器服务	1.0
<code>AlarmManagerService</code>	定时器管理服务，提供定时提醒服务	1.0
<code>WindowManagerService</code>	Framework 最核心的服务之一，负责窗口管理	3.3
<code>BluetoothService</code>	蓝牙服务	1.0 +
<code>DevicePolicyManagerService</code>	提供一些系统级别的设置及属性	1.3
<code>StatusBarManagerService</code>	状态栏管理服务	1.3
<code>ClipboardService</code>	系统剪切板服务	1.0
<code>InputMethodManagerService</code>	输入法管理服务	1.0
<code>NetStatService</code>	网络状态服务	1.0
<code>NetworkManagementService</code>	网络管理服务	<code>NMS.create()</code>
<code>ConnectivityService</code>	网络连接管理服务	2.3
<code>ThrottleService</code>	暂不清楚其作用	1.3

(续表)

服务类名称	作用描述	启动模式
AccessibilityManagerService	辅助管理程序截获所有的用户输入, 并根据这些输入给用户一些额外的反馈, 起到辅助的效果	1.0
MountService	挂载服务, 可通过该服务调用 Linux 层面的 mount 程序	1.0
NotificationManagerService	通知栏管理服务, Android 中的通知栏和状态栏在一起, 只是界面上前者在左边, 后者在右边	1.3
DeviceStorageMonitorService	磁盘空间状态检测服务	1.0
LocationManagerService	地理位置服务	1.3
SearchManagerService	搜索管理服务	1.0
DropBoxManagerService	通过该服务访问 Linux 层面的 Dropbox 程序	1.0
WallpaperManagerService	墙纸管理服务, 墙纸不等同于桌面背景, 在 View 系统内部, 墙纸可以作为任何窗口的背景	1.3
AudioService	音频管理服务	1.0
BackupManagerService	系统备份服务	1.0
AppWidgetService	Widget 服务	1.3
RecognitionManagerService	身份识别服务	1.3
DiskStatsService	磁盘统计服务	1.0

AmS 的启动模式如下:

- ① 调用 main() 函数, 返回一个 Context 对象, 而不是 AmS 服务本身。
- ② 调用 AmS.setSystemProcess()。
- ③ 调用 AmS.installProviders()。
- ④ 调用 systemReady(), 当 AmS 执行完 systemReady() 后, 会相继启动相关联服务的 systemReady() 函数, 完成整体初始化。

关于具体某个服务的内部启动过程, 请参照源码, 这些过程一般都比较简单。

关于具体某个服务的内部启动过程, 请参照源码, 这些过程一般都比较简单。

## 9.4.2 启动第一个 Activity

当以上服务线程都启动后, 其中 ActivityManagerService (AmS) 服务是以 systemReady() 调用完成最后启动的, 而在 AmS 的 systemReady() 函数内部的最后一段代码则发出了启动任务队列中最上面一个 Activity 的消息, 如下代码所示:

```
6294 mMainStack.resumeTopActivityLocked(null);
```

由于系统刚启动时, mMainStack 队列中并没有任何 Activity 对象, 因此, ActivityStack 类中将调用 startHomeActivityLocked() 函数, 如下代码所示:



```

1053     ActivityRecord next = topRunningActivityLocked(null);
1054
1055     // Remember how we'll process this pause/resume situation,
1056     // that the state is reset however we wind up proceeding.
1057     final boolean userLeaving = mUserLeaving;
1058     mUserLeaving = false;
1059
1060     if (next == null) {
1061         // There are no more activities! Let's just start up
1062         // Launcher...
1063         if (mMainStack) {
1064             return mService.startHomeActivityLocked();
1065         }
1066     }

```

从这里可以看出, 开机后系统从哪个 Activity 开始执行完全取决于 mMainStack 中的第一个 Activity 对象。可以想象, 如果在 ActivityManagerService 启动时能够构造一个 Activity 对象 (并不是说构造出一个 Activity 类的对象), 并将其放到 mMainStack 中, 那么第一个运行的 Activity 就是这个 Activity, 这一点不像其他操作系统中通过设置一个固定程序作为第一个启动程序。

在 AmS 的 startHomeActivityLocked() 中, 系统发出了一个 category 字段包含 CATEGORY\_HOME 的 intent, 如以下代码所示:

```

1964     intent.setComponent(mTopComponent);
1965     if (mFactoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
1966         intent.addCategory(Intent.CATEGORY_HOME);
1967     }

```

无论是哪个应用程序, 只要声明自己能够响应该 intent, 那么就可以被认为是 Home 程序, 这就是为什么在 Android 领域会存在各种所谓的“Home 程序”的原因。系统并没有给任何程序赋予“Home”特权, 而把这个权利交给了用户, 当系统中有多个程序能够响应该 intent 时, 系统会弹出一个对话框, 请求用户选择启动哪个程序, 并允许用户记住该选择, 从而使得以后每次按“Home”键后都启动相同的 Activity。

这就是第一个 Activity 的启动过程。





## 第 10 章 AmS 内部原理

ActivityManagerService.java 文件，简称 AmS，一共有 14684 行代码，是 Android 内核的三大核心功能之一，另外两个是 WindowManagerService.java 和 View.java。

尽管 AmS 的代码庞大，逻辑纷繁，但从所提供的功能来看，却没有看上去那么复杂。笔者分析 AmS 源码时，首先猜想并验证 AmS 所提供的各种功能，然后根据这些功能再去找相应的代码，从而能够对该文件进行有效的逻辑分隔。

AmS 所提供的主要功能包括以下几项：

- 统一调度各应用程序的 Activity。应用程序要运行 Activity，会首先报告给 AmS，然后由 AmS 决定该 Activity 是否可以启动，如果可以，AmS 再通知应用程序运行指定的 Activity。换句话说，运行 Activity 是各应用进程的“内政”，AmS 并不干预，但是 AmS 必须知道各应用进程都运行了哪些 Activity。
- 内存管理。Android 官方声称，Activity 退出后，其所在的进程并不会被立即杀死，从而下次在启动该 Activity 时能够提高启动速度。这些 Activity 只有当系统内存紧张时，才会被自动杀死，应用程序不用关心这个问题。而这些正是在 AmS 中完成的。
- 进程管理。AmS 向外提供了查询系统正在运行的进程信息的 API。这些都比较简单。

下面就根据这些功能点，来分析 AmS 的代码。

### 10.1 Activity 调度机制

在 Android 中，Activity 调度的基本思路是这样的：各应用进程要启动新的 Activity 或者停止当前的 Activity，都要首先报告给 AmS，而不能“擅自处理”。AmS 在内部为所有应用进程都做了记录，当

AmS 接到启动或停止的报告时，首先更新内部记录，然后再通知相应客户进程运行或者停止指定的 Activity。由于 AmS 内部有所有 Activity 的记录，也就理所当然地能够调度这些 Activity，并根据 Activity 和系统内存的状态自动杀死后台的 Activity。

具体来讲，启动一个 Activity 有以下几种方式。

- 在应用程序中调用 `startActivity()` 启动指定的 Activity。
- 在 Home 程序中单击一个应用图标，启动新的 Activity。
- 按 “Back” 键，结束当前 Activity，自动启动上一个 Activity。
- 长按 “Home” 键，显示出当前任务列表，从中选择一个启动。

这四种启动方式的主体处理流程都会按照第一种启动方式运行，后面三种方式只是在前端消息处理上各有不同，因此，后面首先介绍第一种启动方式，然后介绍其他启动方式的前端处理差异。

### 10.1.1 几个重要概念

AmS 中定义了几个重要的数据类，分别用来保存进程（Process）、活动（Activity）和任务（Task）。

#### 1. 进程数据类 ProcessRecord

该类在 `framework/base/services/java/com/android/server/am/` 路径下，该路径最后的 `am` 代表 Activity Manager，和 AmS 有关的重要类都在该目录下。

一个 APK 文件运行时会对应一个进程，当然，多个 APK 文件也可以运行在同一个进程中。ProcessRecord 正是记录一个进程中的相关信息，该类中内部变量可分为三个部分，大家先不用琢磨具体某个变量如何被使用，而只需要先了解它们的作用。这三个部分如表 10-1 所示。

表 10-1 ProcessRecord 类内部信息

变量作用	主要包含
进程文件信息	也就是与该进程对应的 APK 文件的内部信息，比如 <code>ApplicationInfo info;</code> <code>String processName;</code> <code>HashSet&lt;String&gt; pkgList;</code> //保存该进程中的所有 APK 文件包名
该进程的内存状态信息	这些信息将用于 Linux 系统的 Out Of Memory(OOM)情况的处理，当发生系统内部不够用时，Linux 系统会根据进程的内存状态信息，杀掉优先级比较低的进程。 <code>int maxAdj;</code> <code>int curAdj;</code>  变量中 <code>adj</code> 的含义是 <code>adjustment</code> ，即“调整值”
进程中包含的 Activity、Provider、Service 等	<code>ArrayList activities;</code> <code>ArrayList services;</code>

## 2. HistoryRecord 数据类

AmS 中使用 HistoryRecord 数据类来保存每个 Activity 的信息，有些读者可能奇怪，Activity 本身也是一个类啊，为什么还要用 HistoryRecord 来保存 Activity 的信息，而不直接使用 Activity 呢？因为，Activity 是具体的功能类，这就好比每一个读者都是一个 Activity，而“学校”要为每一个读者建立一个档案，这些档案中并不包含每个读者具体的学习能力，而只是学生的籍贯信息、姓名、出生年月、家庭关系等。HistoryRecord 正是 AmS 为每一个 Activity 建立的档案，该数据类中的变量主要包含两部分，如表 10-2 所示。

表 10-2 HistoryRecord 类内部信息

变量作用	主要包含
环境信息	该 Activity 的工作环境，比如，隶属于哪个 Package，所在的进程名称、文件路径、数据路径、图标、主题，等等，这些信息基本上是固定的
运行状态信息	比如 idle、stop、finishing 等，这些变量一般为 boolean 类型，这些状态值与应用程序中所说的 onCreate、onPause、onStart 等状态有所不同

需要注意，HistoryRecord 类也是一个 Binder，它基于 IApplicationToken.Stub 类，因此，可以被 IPC 调用，一般是在 WmS 中进行该对象的 IPC 调用。

## 3. TaskRecord 类

AmS 中使用任务的概念确保 Activity 启动和退出的顺序。比如以下启动流程，A、B、C 分别代表三个应用程序，数字 1、2、3 分别代表该应用中的 Activity。

A1→A2→A3→B1→B2→C1→C2，此时应该处于 C2，如果 AmS 中没有任务的概念，此时又要从 C2 启动 B1，那么会存在以下两个问题：

- 虽然程序上是要启动 B1，但是用户可能期望启动 B2，因为 B1 和 B2 是两个关联的 Activity，并且 B2 已经运行于 B1 之后。如何提供给程序员一种选择，虽然指定启动 B1，但如果 B2 已经运行，那么就启动 B2。
- 假设已经成功从 C2 跳转到 B2，此时如果用户按“Back”键，是应该回到 B1 呢，还是应该回到 C2？

任务概念的引入正是为了解决以上两个问题，HistoryRecord 中包含一个 int task 变量，保存该 Activity 所属哪个任务，程序员可以使用 Intent.FLAG\_NEW\_TASK 标识告诉 AmS 为启动的 Activity 重新创建一个 Task。

有了 Task 的概念后，以上情况将会是这样的：

虽然程序明确指定从 C2 启动到 B1，程序员可以在 intent 的 FLAG 中添加 NEW\_TASK 标识，从而使得 AmS 会判断 B1 是否已经在 mHistory 中。如果在，则找到 B1 所在的 Task，并从该 Task 中的最上面的 Activity 处运行，此处也就是 B2。当然，如果程序的确要启动 B1，那么就不要使用 NEW\_TASK 标识，使用的话，mHistory 中会有两个 B1 记录，隶属于不同的 Task。

TaskRecord 类中的变量如表 10-3 所示。

表 10-3 TaskRecord 类内部变量

变量名称	描 述
int taskId;	每一个任务对应一个 Int 型的标识
Intent intent;	创建该任务时对应的 intent
int numActivities;	该任务中的 Activity 数目

需要注意的是,TaskRecord 中并没有该任务中所包含的 Activity 列表,比如 ArrayList<HistoryRecord> 或者 HistoryRecord[] 之类的变量,这意味着不能直接通过任务 id 找到其所包含的 Activity。要达到这个目的,可以遍历 AmS 中 mHistory 中的全部 HistoryRecord,然后根据每一个 HistoryRecord 中的 TaskRecord task 变量确定是否属于指定的任务。

### 10.1.2 AmS 中的一些重要调度相关变量

要了解 AmS 调度、管理系统中的 Activity 的细节,必须了解 AmS 中定义的重要内部变量。要一下了解这些变量的使用时机并非易事,因此,本节仅简要说明一些变量的作用,至于具体使用的时机,要结合调度的具体过程了解。

#### 1. 系统常量

- static final int MAX\_ACTIVITIES = 20;

系统只能有一个 Activity 处于执行状态,对于非执行状态的 Activity,AmS 会在内部暂时缓存起来,而不是立即杀死,但如果后台的 Activity 数目超过该常量,则会强制杀死一些优先级较低的 Activity,所谓的“优先级高低”的规则见第 10.2 节。

- static final int MAX\_RECENT\_TASKS = 20;

AmS 会记录最近启动的 Activity,但只记录 20 个,超过该常量后,AmS 会舍弃最早记录的 Activity。

- static final int PAUSE\_TIMEOUT = 500;

当 AmS 通知应用进程暂停指定的 Activity 时,AmS 的忍耐是有限的,因为只有 500 毫秒,如果应用进程在该常量时间内还没有暂停,AmS 会强制暂停关闭该 Activity。这就是为什么在应用程序设计时,不能在 onPause() 中做过多事情的原因。

- static final int LAUNCH\_TIMEOUT = 10\*1000;

当 AmS 通知应用进程启动 (Launch) 某个 Activity 时,如果超过 10s,AmS 就会放弃。

- static final int PROC\_START\_TIMEOUT = 10\*1000;

当 AmS 启动某个客户进程后,客户进程必须在 10 秒之内报告 AmS 自己已经启动,否则 AmS 会认为指定的客户进程不存在。

## 2. 等待序列

由于 AmS 采用 Service 机制运作，所有的客户进程要做什么事情，都要先请求 AmS，因此，AmS 内部必须有一些消息序列保存这些请求，并按顺序依次进行相应的操作。

- `final ArrayList mHistory = new ArrayList();`

这是最最重要的内部变量，该变量保存了所有正在运行的 Activity，所谓正在运行是指该 HistoryRecord 的 finishing 状态为 true。比如当前和用户交互的 Activity 属于正在运行，从 A1 启动到 A2，尽管 A1 看不见了，但是依然是正在运行，从 A2 按“Home”键回到桌面，A2 也是正在运行，但如果从 A2 按“Back”键回到 A1，A2 就不是正在运行状态了，它会从 mHistory 中删除掉。

- `private final ArrayList mLRUActivities = new ArrayList();`

LRU 代表 Latest Recent Used，即最近所用的 Activity 列表，它不像 mHistory 仅保存正在运行的 Activity，mLRUActivities 会保存所有过去启动过的 Activity。

- `final ArrayList<PendingActivityLaunch> mPendingActivityLaunches  
= new ArrayList<PendingActivityLaunch>();`

当 AmS 内部还没有准备好时，如果客户进程请求启动某个 Activity，那么会被暂时保存到该变量中，这也就是 Pending 的含义。这种情况一般发生在系统启动时，系统进程会查询系统中所有属性为 Persistent 的客户进程，此时由于 AmS 也正在启动，因此，会暂时保存这些请求。

- `final ArrayList<HistoryRecord> mStoppingActivities  
= new ArrayList<HistoryRecord>();`

在 AmS 的设计中，有这样一个理念：优先启动，其次再停止。即当用户请求启动 A2 时，如果 A1 正在运行，AmS 首先会暂停 A1，然后启动 A2，当 A2 启动后再停止 A1。在这个过程中，A1 会被临时保存到 mStoppingActivities 中，知道 A2 启动后并处于空闲时，再回过头来停止 mStoppingActivities 中保存的 HistoryRecord 列表。

- `final ArrayList mFinishingActivities = new ArrayList();`

和 mStoppingActivities 有点类似，当 AmS 认为某个 Activity 已经处于 finish 状态时，不会立即杀死该 Activity，而是会保存到该变量中，直到超过系统设定的警戒线后，才去回收该变量中的 Activity。

## 3. 当前不同状态的 HistoryRecord

- `HistoryRecord mPausingActivity = null;`

正在暂停的 Activity，该变量只有在暂停某个 Activity 时才有值，代表正在暂停的 Activity。

- `HistoryRecord mResumedActivity = null;`

当前正在运行的 Activity，这里的正在运行并不见得一定是正在与用户交互。比如当用户请求执行 A2 时，当前正在运行 A1，此时 AmS 会首先暂停 A1，而在暂停的过程中，AmS 会通知 WmS 暂停获取用户消息，而此时 mResumedActivity 依然是 A1。

- `HistoryRecord mFocusedActivity = null;`

这里的 Focus 并非是正在和用户交互, 而是 AmS 通知 WmS 应该和用户交互的 Activity, 而在 WmS 真正处理这个消息之前, 用户还是不能和该 Activity 交互。

- HistoryRecord mLastPausedActivity = null;  
上一次暂停的 Activity。

### 10.1.3 startActivity() 的流程

程序员可以直接调用 startActivity() 启动指定的 Activity。前面说过, 尽管从用户的角度来看, 启动 Activity 有不同的方式, 但是其主体流程是完全相同的, 前端各种交互方式最后都会调用到 startActivity() 启动。因此, 首先介绍 startActivity() 的流程。

#### 1. 概述

调用流程可大致归结为如图 10-1 和图 10-2 所示, 由于版面限制, 该图被划分为两部分。

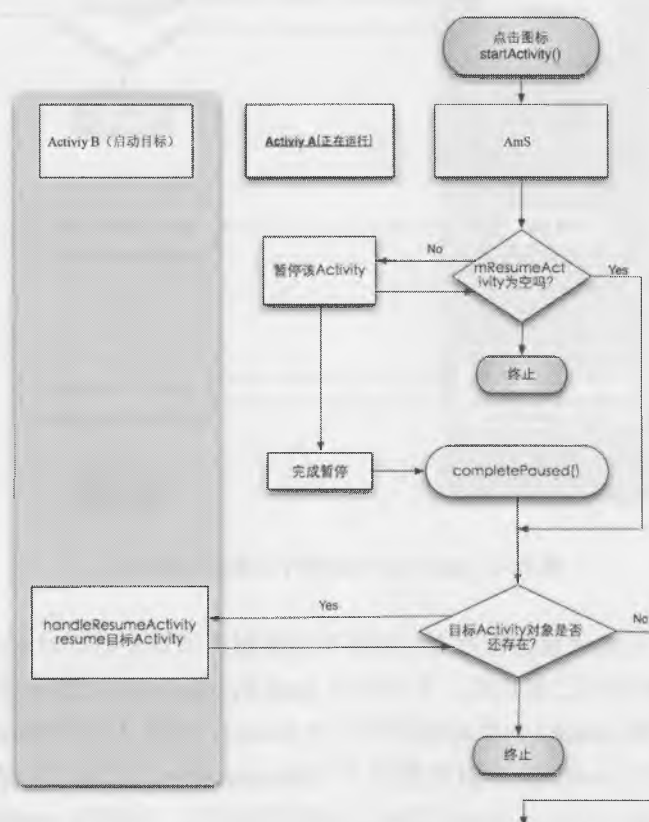


图 10-1 startActivity() 的内部调用流程

一般调用 `startActivity()` 方法是从用户单击桌面图标开始的, 经过各种调用, 最后导致调用 `startActivity()`。本例假设当前正在运行 A, 而单击图标后会运行 B。

AmS 收到客户请求 `startActivity()` 后, 会首先暂停当前的 Activity, 因此要判断 `mResumedActivity` 是否为空。在一般情况下, 该值都不为空, 如果不为空, AmS 会通知 A 所在客户进程暂停, 执行该 Activity, 并立即返回, 返回后 AmS 就又“睡觉”去了。读者可能觉得奇怪, AmS “睡觉”去了, 谁来启动 B 呢? 这正是 AmS 的异步机制的好处, 异步机制允许执行完一个命令后就去休息, 而当远程执行完某个命令后再回调。这种方式广泛应用于 Android 的内核实际中, 大家要迅速解除这种疑惑。

当 A 进程完成暂停后, 报告 AmS, 这时 AmS 开始执行 `completePaused()`。该方法中先要检查目标 Activity 是否在 `mHistory` 列表中, 如果在, 说明目标进程还在运行, 目标 Activity 只是处于 stop 状态, 还没有 finish, 所以通知 B 进程直接 `resume` 指定的 Activity 即可。如果不在 `mHistory` 列表中, 则继续执行如图 10-2 所示的流程。

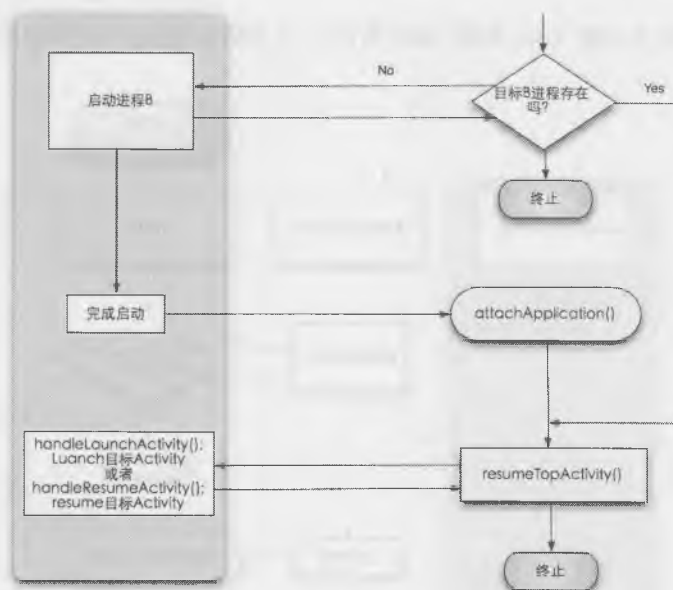


图 10-2 `startActivity()` 的内部调用流程 (续)

检查目标 Activity 所在的进程是否存在, 如果不存在则必须首先启动对应的进程。当对应的进程启动后, B 进程会报告 AmS 自己已经启动, 于是执行 AmS 的 `attachApplication()` 方法, 该方法可理解为 B 进程请求 AmS 给自己安排(attach)一个具体要执行的 Activity, 此时 AmS 继续调用 `resumeTopActivity()`, 通知 B 进程执行指定的 Activity。如果指定的 `HistoryRecord` 在 B 进程中不存在, 则 B 调用 `handleLaunchActivity()` 创建一个该 Activity 实例; 如果已经存在, 则调用 `handleResumeActivity()` 恢复已有的 Activity 的运行。这个逻辑的意思就是说, 在 `ActivityThread` 中可以存在同一个 Activity 的多个实例, 对应了 AmS 中 `mHistory` 的多个 `HistoryRecord` 对象。在一般情况下, 当调用 `startActivity(intent)` 的



FLAG 为 NEW\_TASK 时, AmS 会首先从 mHistory 中找到指定 Activity 所在的 Task, 然后启动 Task 中的最后面一个 Activity; 如果 FLAG 不为 NEW\_TASK, 那么 AmS 会在当前 Task 中重新创建一个 HistoryRecord, 这就有可能创建同一个 Activity 的多个 HistoryRecord 对象, 在 ActivityThread 端就可能创建同一个 Activity 类的多个实例。

## 2. 单击图标的过程

关于 startActivity() 的细节过程如附图 3 所示, 本图绘制时遵循以下规则:

假设 class A 的内部函数关系如下示意代码。

```
public class Persedu {
    void f1() {
        f2();
    }

    void f2() {
        f3();
    }

    void f3() {
        x1();
        x2();
        x3();
    }

    void x1(){};
    void x2(){};
    void x3(){};
}
```

该类中的函数调用有两个特点, f1 调用 f2, f2 又调用 f3, 属于垂直调用, 而在 f3 中先后调用 x1、x2、x3, 这属于包含式调用。为了区分这两种调用, 前者在图中以垂直方式表示, 而后者以水平方式表示, 如图 10-3 所示。

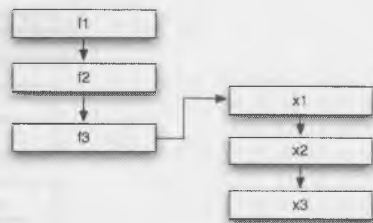


图 10-3 函数调用关系绘图规则示例

startActivity() 的细节过程可分为七步, 将分节介绍, 首先从用户单击图标开始。

当用户单击某个应用图标后, 执行程序会在该图标的 onClick 事件中调用 startActivity() 方法, 该方法属于 Activity 类的内部方法, 然后该方法会调用 startActivityForResult(), 调用时自动把第二个参数设为 -1。所以, startActivity() 和 startActivityForResult() 两者没有本质区别。

在 forResult 方法内部, 会调用 Instrumentation 对象的 executeStartActivity() 方法, 该对象是在 ActivityThread 创建 Activity 时创建的, 一个应用程序中只有一个 Instrumentation 对象, 每个 Activity 内部都有一个该对象的引用。Instrumentation 可以理解为应用进程的管家, ActivityThread 要创建或者暂停某个 Activity 时, 是通过这个“管家”进行的, 设置这个管家的好处是可以统计所有的“开销”, 开销



的信息保存在“管家”那里。其实正如其名称所示，Instrumentation 就是为了“测量”、“统计”，因为管家掌握了所有的“开销”，自然也就具有了统计的功能。当然，Instrumentation 类和 ActivityThread 的分工是有本质区别的，后者就像是这个家里的主人，负责创建这个“家庭”，并负责和外界打交道，比如接收 AmS 的通知等。

在 Instrumentation 中，则继续调用 AmS 的 startActivity() 方法，从而真正把启动的意图传递给 AmS。Android 中 IPC 调用是同步的，只有当 AmS 执行完 startActivity() 方法后才返回，而在此期间，Instrumentation 会一直处于线程等待状态。由于一个应用进程一般只有一个主线程，这就意味着应用主线程在这期间会一直等待，所幸的是 AmS 执行 startActivity() 是异步的，基本上会在很短的时间内返回，然后使用异步通知机制，回调 caller 的 onActivityResult() 方法并返回执行结果。这就是为什么在 AmS 的 startActivity() 参数中需要包含一个 IBinder 类型的 token。该参数来源于 Activity 对象内部的 mToken 变量，而这个变量对应的是 AmS 中的一个 HistoryRecord，AmS 正是通过 HistoryRecord 来识别客户进程中的 Activity 的。当 AmS 执行完指定的 Activity 后，如果 caller 需要 forResult，AmS 就会从 HistoryRecord 中取出 HistoryRecord.app.thread 变量，该变量的类型是 ActivityThread.ApplicationThread 子类，并调用该类的 scheduleSendResult(r, list) 方法，该方法中的 r 代表了 HistoryRecord。ActivityThread 收到这个调用后，将根据参数 r 判断属于哪个 Activity，然后调用相应 Activity 的 onActivityResult() 方法。这个过程如图 10-4 所示。

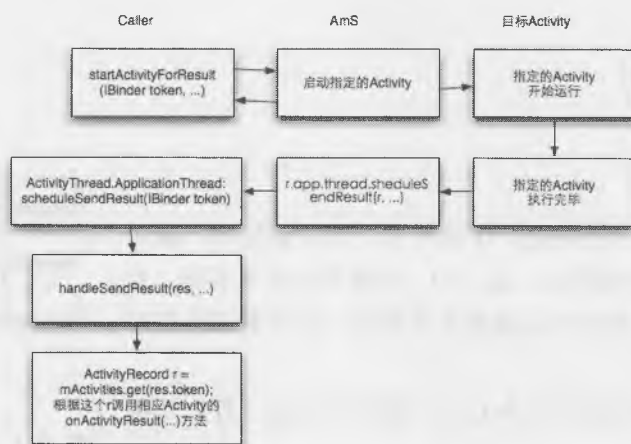


图 10-4 startActivityForResult() 的执行过程

### 3. 运行环境检查

当 AmS 收到启动某个 Activity 的意图 (intent) 后，首先要检查一下环境，比如 Caller 是否有启动的权限、意图是否对应存在的 Activity 等。这就相当于去医院看病时的导医一样，先进行一些简单的检查，如果不满足条件，则立即返回。

这些具体的检查请参照图 10-4 所示。包括如下：

首先, 根据 intent 的信息找到匹配的 Component 信息, 这就是为什么 intent 中可直接指定 Activity 名称的原因, AmS 会调用系统内部的 PackageManager 查询具体的 Component 名称。当然, 如果没有 Component 匹配该 intent, 则返回失败。

接着调用 startActivityLocked(caller, intent, ...)方法, AmS 中有两个同名该方法, 注意其使用的时机和作用, 以下篇幅以不同的参数类型表示不同的方法。该方法的后缀是 Locked, 意思是该方法必须是线程安全的, 或者叫不可重入的, 即该函数体只能由一个线程调用, 另一个线程也要调用该函数时, 必须等待前一个线程执行完。

在 startActivityLocked(caller, intent, ...)的包含调用中, 主要做了以下几件事情。

① 检查当前正在运行的 Activity 是否和目标 Activity 一致, 如果一致, 则直接返回。这就是说, 程序员在 Activity 使用 startActivity 启动自己, 不会达到重启当前 Activity 的目的。

② 处理 INTENT\_FLAG\_FORWARD\_RESULT 标志。从代码的角度来看, 这个标志有一个特殊的作用, 就是能够跨 Activity 传递 Result。比如 A1→A2, 此时如果从 A2 中启动 A3, 并且设置的启动标志为 FORWARD\_RESULT, 那么 A3 运行时, 可以在 A3 中调用 setResult, 然后 finish(), 其结果会从 A3 直接返回到 A1, 并且 A1 会得到 A3 所 set 的 result。要满足这种调用, 必须使用以下方式启动。

A1(startActivityForResult) → A2(StartActivity) → A3。注意 A2 不能使用ForResult 的方式启动 A3, 否则会发生冲突 START\_FORWARD\_AND\_REQUEST\_CONFLICT。

③ 在此检查 Caller 的 app 是否存在, 这种情况发生在发出 startActivity 命令后, Caller 所在的进程被意外杀死, 如果是这样, AmS 拒绝继续往下执行。

④ 检查 Caller 是否具备启动指定 Activity 的权限。

⑤ 如果 AmS 中有 IActivityController 对象, 则通知该 Controller 进行相应的控制。在一般情况下, mController 始终不存在, 可能是 Android 为了某种系统测试而设置的 debug 开关。

⑥ 创建一个临时的 HistoryRecord 对象, 该对象只是为了后面调用过程中的各种对比, 不一定会最终被加入到 mHistory 列表中。

⑦ 检查当前是否允许切换 Activity, 不允许切换的情况一般发生在当前已经暂停的正在执行的 Activity, 正在等待下一个 Activity 的启动时, 此时不能进行 Activity 切换。如果是这样, 则把指定的 Activity 临时添加到 mPendingActivityLaunches 列表中, 等待系统恢复后再继续执行。

⑧ 判断 mPendingActivityLaunches 列表中是否有等待的 Activity 要启动, 如果有的话, 应当先运行这行等待的 Activity。

⑨ 如果等待序列为空, 则调用 startActivityUncheckedLocked()方法。此时, 要启动 Activity 已经通过重重检验, 被确认为是一个“正当”的启动请求。接下来, AmS 就会开始判断以何种方式来启动指定的 Activity。

#### 4. 找到或创建合适的 Task

前面说过, Task 的作用是确保 Activity 按照指定的方式退出, 并当用户按“Back”键时按照指定的方式执行下一个 Activity。在真正启动目标 Activity 之前, AmS 先要确定是否需要为目标 Activity 创建

一个新的 Task，过程如下。

① 处理 FLAG\_ACTIVITY\_NO\_USER\_ACTION 标识。在一般情况下，启动一个 Activity 时都不使用该标识，如果不包含该标识，AmS 会判断一定的时间内是否有用户交互。如果没有用户交互的话，AmS 会通知 Activity 回调 onUserLeaving() 方法，然后再回调 onPause() 方法，如果使用了该标识，说明目标 Activity 不和用户交互，所以也就不需要回调 onUserLeaving() 方法。

② 确定是否现在就启动，在一般情况下都是立即启动。如果立即启动，就把 r.delay Resume 为 true，意思是不要延迟启动。

③ 处理 FLAG\_ACTIVITY\_PREVIOUS\_IS\_TOP 标志，这个标识基本无用。接着再处理 onlyIfNeed 参数，这个参数会和该标志一起使用，不过都是基本无用。

④ 为目标 Activity 赋予合法的权限。目标 Activity 在安装时会指定所需要的权限，此处正是把这些允许的权限添加到缓存中。

⑤ 根据 launchMode 变量设置局部变量 launchFlags，因为接下来要根据 launchFlags 决定启动 Activity 的方式。注意 launchMode 和 launchFlags 是两码事，mode 是 Activity 自己声明的运行方式，它是在 AndroidManifest.xml 中指定的；而 flag 是调用者希望以何种方式运行指定的 Activity，是在调用 startActivity() 时参数 intent 中指定的。

⑥ 对 NEW\_TASK 和 SINGLE\_TASK 标识以及 SINGLE\_INSTANCE 模式进行处理。请注意，这两个标志必须在 r.resultTo==0 的条件中，即这两个标识只能用在 startActivity() 的方法中，而不能用在 startActivityForResult() 方法中。因为从 Task 的角度来看，Android 认为不同 Task 之间的 Activity 是不能传递数据的，所以不能使用 NEW\_TASK 标识，但是还要调用 forResult() 方法。

NEW\_TASK、SINGLE\_TASK 以及 SINGLE\_INSTANCE 三者的作用如图 10-5 所示。

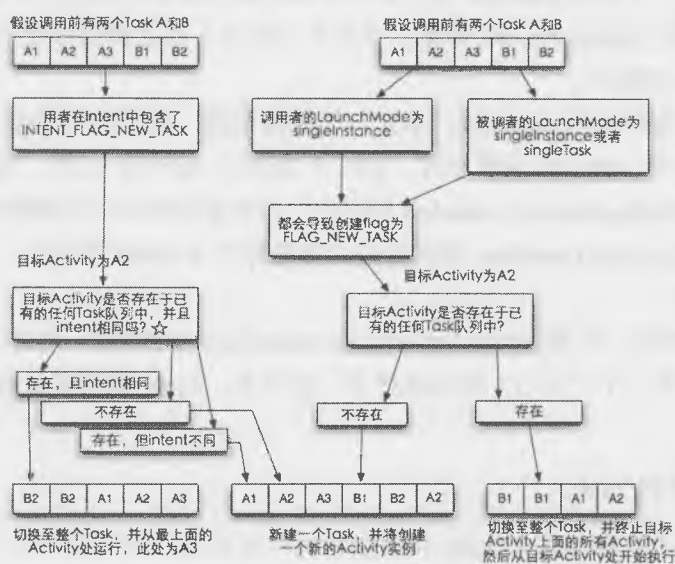


图 10-5 Activity 启动模式

上图中注意“☆”符号处所讲的“intent 相同”的含义，它是指想要启动的 Activity 对应的 intent 和 Task 中第一个 Activity 启动时所使用的 Intent。系统认为，只有 Intent 相同才意味着用户真的是想启动已有的 Task，否则应该创建一个新的 Task。“Intent 相同”具体包含了 intent 中的 Action 字段、Category 字段、data 字段、ComponentName 字段等都要相等。

SINGLE\_TASK（简称 ST）和 SINGLE\_INSTANCE（简称 SI）的相同点在于，如果目标 Activity 不存在，则两者都会创建一个新的 Task，而如果目标 Activity 存在，则两者都会切换到已有的 Task，并终止目标 Activity 上面的所有 Activity，并从 Activity 处开始执行。

ST 和 SI 的区别有两点：第一，调用者本身如果是 SI，那么肯定会创建一个新的 Task，无论目标 Activity 是否存在，但对于 ST，就没有这么“极端”；第二，SI 所在的 Task 只会有一个 Activity 对象，就是这个 SI，而 ST 中可以包含多个 Activity，但 ST 所对应的 Activity 肯定是根 Activity。

⑦ 如果上一步发现可以从 History 中 resume 一个已经存在的 HistoryRecord 对象，那么就不会执行到这一步。如果找不到，则先判断是否需要添加一个新的 Task，如果需要则创建，并把指定的 Activity 添加到新 Task 中；如果不需要创建一个新的 Task，那么就在当前的 Task 中添加目标 HistoryRecord。

如果要在当前 Task 中添加 HistoryRecord 对象，则要处理以下两个启动标识，这两个标识仅适用于已有的 Task 中包含目标 Activity。

- FLAG\_ACTIVITY\_CLEAR\_TOP：如果指定的 Activity 已经存在，则清空其上面的 Activity。
- FLAG\_ACTIVITY\_REORDER\_TO\_FRONT：该标识没有 CLEAR\_TOP 那么严厉，它只是把自己放到最上面，而不清空其他已有的。

⑧ 解决了 Task 的问题，并且对 mHistory 列表中的顺序已经作了应有的调整，接下来，就可以按照常规方式去启动相应的 Activity 了。

读者可能会觉得以上八步在代码实现上有些复杂，因此，以下给出另一种思路理解这些代码。

这八步的最终目的是调整 mHistory 中 Activity 的顺序，在调整时，Android 的设计理念遵守了一个简单的原则。即如果是跨 Task 切换，调用者是没有权利更改被调 Task 里面的 Activity 顺序的，除非被调者自己声明了某些属性。这个原则是合理的，因为只有用户有权利去更改自己当前所操作的 Task 中的 Activity 顺序。举例来讲，假设 mHistory 中的 Activity 顺序如下：

A1→A2→A3→B1→B2→B3→C1→C2→C3→C4。其中，A、B、C 代表三个不同的 Task，此时正在运行 C4，如果程序此时需要从 C 切换到 B 中的任何一个 Activity，切换后的顺序只能是：

A1→A2→A3→C1→C2→C3→C4→B1→B2→B3→B4。程序不能对 B 中的 Activity 顺序进行更改，但程序却可以对 C 中的 Activity 的顺序进行调整，因为此时用户正在和 C 交互，系统认为在当前交互的任务中切换 Activity 是合法的。

系统提供了两种方式完成以上的切换，第一种是在 AndroidManifest.xml 文件中声明 Activity 自身的启动属性，另一种是在启动时给 intent 中添加不同的 flag。前者包括：

- android:launchMode=standard/singleTop/singleTask/singleInstance
- android:clearTaskOnLaunch=true/false

- android:finishonTaskLaunch=true/false
- android:allowTaskReparent=true/false

后者包括：

- Intent.FLAG\_ACTIVITY\_NEW\_TASK
- Intent.FLAG\_ACTIVITY\_RESET\_TASK\_IF\_NEEDED
- Intent.FLAG\_ACTIVITY\_CLEAR\_TOP
- Intent.FLAG\_ACTIVITY\_REORDER\_TO\_FRONT
- Intent.FLAG\_ACTIVITY\_NO\_HISTORY
- Intent.FLAG\_ACTIVITY\_SINGLE\_TOP

这些设置表面上看起来挺复杂，实际上按功能可分为三类，从而理解起来很简单。第一类是为了完成在 Task 之间切换，第二类是为了完成在当前 Task 中改变 Activity 的顺序，第三类是为了在 Task 切换时 Task 内部自动重排所属的 Activity。首先来看第一类。

第一类：在 Task 之间切换。如上面例子，系统允许程序从 B 任务切换到 C 任务，具体实现就是在启动的 intent 中设置 FLAG\_ACTIVITY\_NEW\_TASK。如果 intent 所匹配到的 Activity 已经存在于已有的 Task 中，那么就切换 Task；如果没有，则创建一个新的 Task。而这只是一种实现方式，实际的需求有时是调用者希望被调者出现在一个新的 Task 中，有时则是被调者要求自己必须在一个新的 Task 中，对于后者，则使用 launchMode 来完成，如图 10-6 所示。

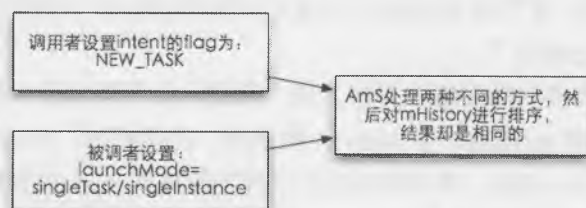


图 10-6 使用 launchMode 启动 Activity

第二类：在同一个 Task 中调整 Activity 的顺序，由于这些 FLAG 是用于在当前 Task 中切换的，因此不能和 NEW\_TASK 标识一起使用。这类常见的包括以下几种。

- CLEAR\_TOP，作用是清除目标 Activity 上面的 Activity。比如 C1-C2-C3，如果此时要启动 C2，结果将是 C1-C2，C3 会被结束掉。
- REORDER\_TO\_FRONT，简单重排，结果会是 C1-C3-C2。
- NO\_HISTORY，目标 Activity 不出现在 mHistory 中。比如 C1-C2，此时如果要启动 C3，而在 C3 中又启动了 C4，那么 mHistory 中将会是 C1-C2-C4，C4 返回后将从 C2 处继续执行。

第三类：允许被调 Task 在启动前主动重排。首先调用者要发起这个请求，其次是被调者要有这个

功能。发起是通过设置 intent 标识为 RESET\_IF\_NEEDED 实现的，而功能是在 manifest 文件中使用 android:clearTaskOnLaunch 和 android:finishOnTaskLaunch 属性，其具体的作用如其名称所示，此处不再赘述。

本节内容所对应的 Android 源码正是以上三类功能的代码实现。本节代码执行完毕后，无论是否需要创建新的 Task，也无论是否需要创建新的 HistoryRecord 对象，mHistory 列表中已经包含了这些新的 Task 或者新的 HistoryRecord，下一节中将会从 mHistory 中取出第一个 HistoryRecord 并尝试去执行，尽管该 HistoryRecord 对应的实际 Activity 对象可能还不存在，但这个环节已经结束。

### 5. 运行 mHistory 中最后一个 HistoryRecord

上一节执行完后，调用 resumeTopActivity(null) 启动真正的 Activity。

- ① 调用 topRunningActivityLocked() 方法取出当前正在运行的 HistoryRecord 对象。
- ② 判断 mHistory 中是否有记录，如果没有意味着还从未运行任何 Activity，需要首先调用 startHomeActivityLocked() 方法启动所谓的“主界面程序”。
- ③ 判断正在执行的 Activity 是否和目标 Activity 一样，如果一样，则万事大吉。
- ④ 判断当前系统是否处于睡眠状态，如果是，也只能作罢，否则继续执行。
- ⑤ 从 mStoppingActivities 和 mWaitingVisibleActivities 列表中删除目标对象，因为目标对象接下来就会被启动。
- ⑥ 检查当前是否正在暂停某个其他的 Activity，如果是则还不能运行。
- ⑦ 检查当前是否有正在运行的 Activity，一般情况下都会有，所以需要首先暂停当前 Activity。这一点也是手机系统和普通桌面系统的主要差别之一，Android 仅允许同时执行一个 Activity，而桌面系统则可以同时打开多个应用窗口。此时，如果需要暂停当前 Activity，则 AmS 的执行会暂时告一段落，剩下的就是等待被暂停的客户进程报告暂停完毕。
- ⑧ 执行到这一步实际上并不是从 startActivity() 调用过来的，而是从另外两种情况过来的。第一种是上一步被暂停的客户进程报告 AmS 已经暂停完毕，于是 AmS 从 activityPaused() 开始执行并间接调用到此；另一种是当用户按“Back”键时，客户进程会请求 AmS 终结 (finish) 当前 Activity，而在 finish 的过程中也会间接调用到此。

这一步中，设置了上一个 HistoryRecord 的内部变量 waitingVisible，从 false 变为 true，即它也处于等待显示状态，因为接下来 next 要变成显示的状态。

- ⑨ 通知 WmS 添加 Activity 切换的动画，有可能是 Task 切换，也有可能是 Activity 切换。
- ⑩ 这一步比较重要。此时要执行的 HistoryRecord 对象仅仅存在于 mHistory 列表中，而它仅代表了用户的意图，实际上它对应的客户进程还并不存在。所以首先要判断客户进程是否存在，源码中使用以下代码判断：

```
2728 if (next.app != null && next.app.thread != null) {
```

注意，为什么不是仅仅判断 next.app，而是还要判断 next.app.thread？因为 next.app 对应的是一个 ProcessRecord 对象，有时客户进程被杀死而该对象却会依然保留，thread 才是真正的客户进程引用。所



以，只有满足这两个条件才说明该 Activity 对应的客户进程处于已运行状态，并且相应的 Activity 对象也已经存在，所要做的仅仅是 resume 该 Activity。否则，要么是目标进程不存在，要么是目标进程中还需要创建一个新的 Activity 实例。举个例子，假设当前 mHistory 的内容是 C1→C2→C3，此时如果以默认方式启动 C1 类，姑且称之为 C1'，那么，目标进程 C 已经存在，C1 实例也已经存在，然而由于新启动的 C1' 是新建的 C1 对象，所以其内部的 app 和 app.thread 都为空，所以不能直接 resume 对应的 C1。

**11** 如果上面不能直接 resume 已有的 Activity 对象，那么接下来就要判断目标 Activity 对应的进程是否存在。这是通过调用 startSpecificActivityLocked() 完成的，如其函数名所指，其内部要启动一个特定的 Activity。

在该方法中，调用 getProcessLocked() 方法首先获取目标 Activity 对应的进程对象 ProcessRecord app。如果 app 不为空，并且 app.thread 也不为空，说明目标进程还处于活动状态，所以只需要让目标进程再创建一个指定 Activity 的对象并执行之即可。相反，如果目标进程还不存在，则需要首先启动目标进程。这个判断语句和上一步中判断目标 Activity 是否存在的代码有一定形式的相似，但其逻辑是完全不同的，请注意区别，如以下代码所示：

```
1895 if (app != null && app.thread != null) {
```

该代码中的 app 变量是目标 Activity 在 AmS 中对应的 ProcessRecord 对象的，而前面代码中的 app 变量却是 mHistory 列表中要被执行的 HistoryRecord 对象的。

## 6. 应用进程 pause 指定的 Activity

前面讲过，在启动目标 Activity 前，如果当前正在运行任何 Activity，那么必须首先暂停，表面上看起来，这种逻辑严重违反了我们已有的 PC 使用体验，在 PC 中，可以同时运行多个应用程序，最大化一个窗口只是剥夺了其他窗口在屏幕上显示的权利，而该程序本身丝毫没有受到影响，而 Android 中却要强调当启动另一个 Activity 时，当前 Activity 必须首先暂停。事实上，这种规则没有想象的那么不同，因为暂停 Activity 是一种比较轻量级的操作。

暂停 Activity 的流程如图 10-7 所示。

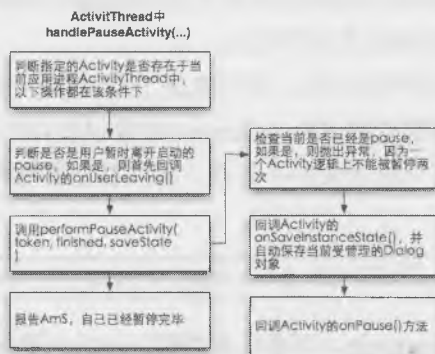


图 10-7 暂停 Activity 的过程

① 判断该 Activity 对象是否存在。

② 判断是否为用户离开产生的暂停消息。Android 后台会自动检测用户多长时间没有和当前 Activity 交互了，如果程序允许该 Activity 接收用户离开消息，那么系统判断是用户离开，就会暂停目标 Activity。此时 `handlePauseActivity` 的参数 `userLeave` 就为 `true`，程序员可以重载 Activity 的 `onUserLeaving()` 方法。

③ 调用 `performPauseActivity()` 完成真正的暂停代码，尽管实际的代码也很简单。这就是为什么说暂停是一种轻量级操作的原因。

(1) 检查当前是否已经 `pause` 了，条件就是当前 `ActivityRecord` 对象的 `pause` 变量值，`true` 代表已经暂停。该步骤对应的代码中，如果已经 `pause`，会抛出一个异常，说 “Performing pause of activity that is not resumed”，但这种情况实际上是不会发生的。

(2) 回调 Activity 的两个暂停操作函数，一个是 `onSaveInstanceState()`，另一个就是应用程序员熟知的 `onPause()`。前者实际上给程序员提供了保存当前状态的机会，该函数的参数是一个 `Bundle` `outstate` 变量，即程序员可以把需要保存的变量内容保存在系统提供的 `outState` 中，除此之外系统还自动把当前处于被管理状态的 `Dialog` 也保存起来；后者则提供了一个无参数的回调，在笔者看来，完全可以把 `onSaveInstanceState()` 和 `onPause()` 合并到一起，即把 `onSaveInstanceState()` 作为 `onPause()` 的默认处理方式，而只需要给 `onPause()` 添加两个参数，如下：

```
onPause(Bundle outState, boolean saveState)
```

这样的话，程序员不会在这两个回调函数中迷惑了，尤其是当考虑应该在哪个函数中保存当前 Activity 内部数据的时候。

④ 报告 AmS，自己已经暂停完毕，通过 IPC 调用到 AmS 的 `completePausedActivity()` 方法。

从以上流程可看出，暂停的操作的开销最多是在 `saveInstanceState()` 方法中，而这段代码不一定会被执行，只有当参数 `saveState` 为 `true` 时才执行，而 Activity 的内部所有变量都依然会持续保留在内存中，下一个要执行的 Activity 如果不是全屏的话，用户应该还可以看到刚才暂停的 Activity 界面。唯一的区别是被暂停的 Activity 将不能接收用户消息，因为 WmS 在接收到用户消息后，仅仅把该消息 `dispatch` 到了当前的活动窗口。由于被暂停的 Activity 内部的所有数据依然保留，包括内部的 `Handler` 对象，其所在的进程也依然保存，因此可以想象，此时如果从当前的 Activity 向被暂停的后台 Activity 的 `Handler` 对象发送一个消息，那么后台 Activity 中的 `Handler` 对象的 `handleMessage()` 方法将依然会被执行。

再一次强调，被暂停的 Activity 仅仅是被 WmS 剥夺了获取用户消息的权利，如果改变 WmS 内部的消息派发机制，允许根据用户单击的屏幕位置，把消息派发到相应的 Activity 窗口，那么这些 Activity 都将处于活动状态，也就是所谓的多窗口多任务。

## 7. activityPaused()

上一节中，当 AmS 通知当前 Activity 暂停后，AmS 会立即返回，而在目标进程中则是发送一个暂停的消息，处理完该暂停消息后，目标进程会调用 AmS 的 `activityPaused()` 报告 AmS 自己已经暂停完毕，从而 AmS 将开始启动真正的 Activity。那么，这个 `activityPaused()` 方法内部的流程如何呢？



① 调用 `indexOfTokenLocked()` 找到指定的 token 在 `mHistory` 中对应的 `HistoryRecord`。如果说 token 的确存在, 那么实际上可以简单地直接将 token 转换为 `HistoryRecord` 即可。

② 如果 `activityPaused()` 的第三个参数 `timeout` 为 `false`, 说明是非超时的、正常时间内暂停的 `Activity`, 所以, 首先需要从 `mHandler` 中移除还没有被处理的超时消息。这种逻辑整体应用于 `AmS` 和 `WmS` 中的很多函数调用中。

③ 如果参数 `r` 和 `AmS` 内部变量 `mPausingActivity` 相同, 说明 `r` 正是刚刚需要暂停的 `Activity`, 因此, 可以调用 `completePausedLocked()` 执行完成暂停后的行为。

(1) 给 `prev` 赋值 `mPausingActivity`, 即此时可以认为上一个被执行的 `Activity` 就是刚刚暂停的 `Activity`。

(2) 如果 `prev` 的 `finishing` 标识为 `true`, 说明上一个 `Activity` 已经完成, 因此需要调用 `finishCurrentActivityLocked()` 执行相关操作。而对于一般的启动流程来讲, 无论是从 A 启动到 B, 还是从 B 按 “Back” 键返回到 A, 都不会是 `finishing` 的, 而仅仅是 `stop` 状态。这个条件似乎只有在内存回收时才会被执行。

(3) 将 `mPausingActivity` 变量置为空, 因为当前已经不存在正在暂停的 `Activity` 了。

④ 调用 `resumeTopActivityLocked()` 方法正式启动目标 `Activity`, 而其内部的调用流程是从 `mHistory` 中取出最后一个 `Activity` 对象, 并去执行。此处不再赘述。

## 8. 通知应用进程 resume 指定的 Activity

在上一节中, 如果目标 `Activity` 已经在运行的序列中, 则直接通知目标进程 resume 暂停的 `Activity` 对象即可。

① 通知 `WmS`, 把目标对象的窗口设为可交互状态。这是通过调用 `WmS` 的 `setAppVisible` 方法完成的, 其参数是目标 `Activity` 对应的 `HistoryRecord` 对象。

② 改变一些内部变量的状态, 比如 `mResumedActivity=next`, 即把目标 `Activity` 作为当前正在运行的 `Activity`。

③ 通知目标进程继续运行目标 `Activity`, 这是通过调用 `next.app.thread.scheduleResumeActivity(next...)` 完成的。从这里可以再次看出, `Android` 的架构中始终认为改变目标 `Activity` 状态的具体任务应该由目标进程完成, `AmS` 仅发出指令。

④ 调用 `completeResumeLocked()` 对内部其他相关变量进行设置。

下面对以上第三步调用 `ActivityThread` 中 `scheduleResumeActivity()` 函数的执行流程作更进一步的分析。该函数本身仅仅是发一个异步消息, 并立即返回到 `AmS`, 而该消息响应函数是 `handleResumeActivity()`, 该函数的内部执行过程如图 10-8 所示。

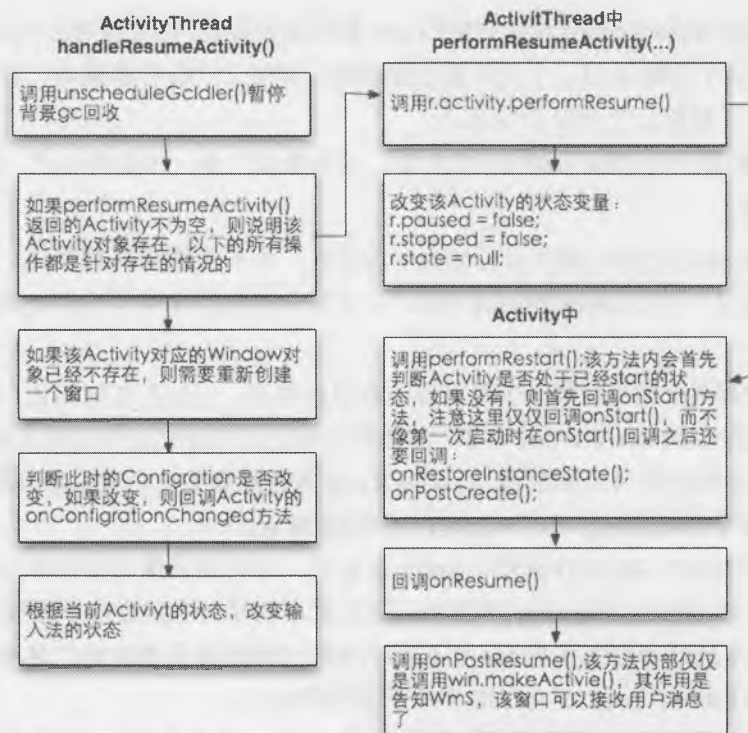


图 10-8 resume Activity 的流程

以上过程描述如下:

- ① 调用 `unscheduleGcIdler()` 暂停停止内部回收。关于内存回收参见本章第 10.2 小节。
- ② 调用 `performResumeActivity()`。该方法内部主要是回调 Activity 对象的 `onResume()` 方法, 但是, 如果此刻 Activity 处于 stop 状态, 即 `r.stopped == true`, 那么就需要先 start 该 Activity, 具体就是仅仅回调 Activity 的 `onStart()` 函数。
- ③ 判断该 Activity 对应的 Window 是否存在。在一般情况下, 该 Window 对象都会存在, 只有一种例外, 那就是系统内存紧张到把后台窗口所对应的显存也临时释放了。在这种情况下, 当被回收的 Activity 对应的窗口重新显示时, 系统会重新为 Activity 分配一个窗口显存(Canvas 对象)。
- ④ 查看当前的 Configuration 是否改变, 比如用户停止该 Activity 时是竖屏退出, 而继续却是横屏进入, 则此时应该回调 Activity 的 `onConfiguration()` 函数。
- ⑤ 根据当前 Activity 的状态改变输入法的状态, 关于输入的逻辑参见本书第 17 章输入法原理。

## 9. 创建应用进程

如果目标 Activity 所在的进程还不存在, 则首先需要启动应用进程, 其大致过程是 AmS 调用 Process 进程类启动一个新的应用进程, 新的应用进程会从 ActivityThread 的 `main()` 函数处开始执行, 当目标进程启动后, 再报告 AmS 自己已经启动, 然后 AmS 再通知目标进程启动目标 Activity。

① 首先判断目标 `HistorRecord` 对象对应的 `app` 数据是否存在,并且其对应的应用进程是否也存在。正如源码中的注释,这个判断是由三个具体条件完成的,如果三个条件都满足,则说明目标进程的确存在。这个判断中指出了相关三个变量的含义。

- `app`: 这代表的是一个 `ProcessRecord` 对象,该对象仅仅是一个数据而已,并不代表目标进程的任何实际状态。
- `app.pid`: 如果 `pid` 大于 0,说明目标进程一起存在,但有可能正在初始化。
- `app.thread`: 这是一个 `ActivityThread` 对象,只有目标进程存在并且已经初始化完毕,该变量才具备有效值。

以上判断一般会有两个结果。第一个是目标进程已经存在,但是正在初始化,这种情况意味着不需要再做什么,仅仅返回该 `ProcessRecord` 对象,并等待目标进程初始化完毕并报告给 `AmS` 即可;另一种情况是 `app` 存在但其不对应任何目标进程,这说明该 `app` 数据对象是一个死对象,就像 Linux 中的 zombie 进程数据,因此,调用 `handleAppDiedLocked()` 方法清除该 `app`。

② 处理 `FROM_BACKGROUND` 标识。Android 认为,如果以背景方式启动目标进程,那么如果该进程已经列为死亡名单 `mBackProcesses` 列表中,那么就不需要再尝试启动,系统认为这是一个不可能启动的目标进程;如果以非背景方式启动,那么则不判断目标进程是否在死亡名单中,而是再次尝试启动目标进程,这就是 `FROM_BACKGROUND` 的代码层面的含义。

③ 判断目标 `HistoryRecord` 的 `app` 变量是否为空。如果为空,则需要为其创建一个新的 `ProcessRecord` 对象,并把该 `record` 对象添加到 `mProcessNames` 列表中,并且同时给 `app.info` 赋值。这点很重要,因为当目标进程启动后,`AmS` 会命令目标进程运行 `app.info` 中包含的 `Activity` 对象,而 `app.info` 的内容的最初来源是 `caller` 调用 `AmS` 的 `startActivity()` 方法中根据调用的 `intent` 从系统中查询到的相关 `ApplicationInfo` 信息。

④ 再次判断 `AmS` 是否处于已启动状态,这只有在系统启动时才会发生,而在一般情况下,系统始终都是处于已启动状态。当然,如果还没有启动好,则把目标进程请求暂时存储到 `mProcessOnHold` 列表中。

⑤ 调用 `startProcessLocked` (三个参数) 启动真正创建一个进程。注意,这与本节的入口函数 `startProcessLocked` (七个参数) 是同名的,注意区分其不同。

⑥ 进入到 `startProcessLocked()` 方法中,从 `mPidsSelfLocked` 列表中删除目标进程,当然,此时该列表中大多数情况下并没有目标进程的记录。该列表是一个 `SparseArray` 对象,保存了所有正在运行目标进程的 `pid` 到 `ProcessRecord` 的映射。从目标功能的角度来看,这个变量和 `mProcessNames` 变量都保存了当前正在运行的 `ProcessRecord` 对象,只是前者提供了 `pid` 映射,而后者提供了进程名称映射。这也说明了无论是 `pid` 还是进程名称都必须是唯一的,这也就是为什么在 Android 中进程名称一般使用包名的原因。

⑦ 为变量 `mProcDeath` 赋值。该变量也是一个 `int` 型数组,内部包含最近 20 个应用进程的意外关闭次数,如果某应用进程没有关闭,则其对应的值为 0,这仅仅是一个统计数据。

⑧ 调用 `Process.start()` 方法，这是一个 native 的方法，内部将会从 `zygote` 进程中 folk 出一个新的应用进程，并且应用进程会从 `ActivityThread` 类的 `main()` 方法中开始执行，关于系统的启动过程参见本书第9章。

⑨ 把上一步返回的 `pid` 和变量 `app` 添加到 `mPidsSelfLocked` 容器中，在后面的 `attachApplication()` 方法中还要根据 `pid` 获得该 `app` 对象。

至此，启动进程的事情就算结束了，接下来 AmS 会等待目标进程报告启动完毕，届时，目标进程会 IPC 调用 AmS 的 `attachApplication()` 方法请求 AmS 给目标进程指定目标 Activity 去运行。而此时，`mHistory` 中最上面的 `HistoryRecord` 对象的 `app` 变量依然为空，而在 `mPidsSelfLocked` 和 `mProcessNames` 两个容器中却都保存了新创建的 `ProcessRecord` 对象。而一旦 `HistoryRecord` 对应的 Activity 被启动，那个相应的 `ProcessRecord` 对象才会被赋值给 `HistoryRecord` 的 `app` 变量。

### 10. attachApplication 的过程

上一节说到，当目标进程启动后，就会报告给 AmS，自己已经启动完毕可以启动 Activity 了，这实际上是通过 IPC 调用 AmS 的 `attachApplication()` 方法完成的，而该方法的内部执行过程如下。

① 根据 `Binder.getCallingPid(thread)` 获得客户进程的 `pid` 值，并调用 `attachApplicationLocked(thread, pid)` 方法，其中参数包含了 `pid` 值。

② 在 `...locked()` 方法中，首先根据 `pid` 找到对应的 `ProcessRecord` 对象，如果找不到对应的 `ProcessRecord` 对象，说明该 `pid` 客户进程是一个没有经过 AmS 允许的“野进程”。为什么呢？因为 AmS 在启动任何客户进程前，都已经在内部为未来的客户进程创建了相应的 `ProcessRecord` 对象，并且在调用 `Process.startProcess()` 返回时，将客户进程的 `pid` 赋值到了 `app.pid` 变量，而此处竟然找不到，那一定是“野进程”。

③ 给 `app` 对象内部的其他相应变量赋值，比如 `app.thread = thread` 等。

④ 确保目标程序（APK）文件已经被转换为了 `odex` 文件。Android 中的安装程序是 APK 文件，该文件实际上是一个 `zip` 文件，该文件在安装时会自动在 `/data/app` 目录产生一个 `dex` 文件。这个 `dex` 文件是从 `zip` 文件中提取的，而为了提高运行效率，该 `dex` 文件在运行前一般都会被系统转换为 `odex` 文件，即所谓的 `Optimized dex` 文件。本步中如果检查还没有生成 `odex` 文件，则先要生成 `odex` 文件。

⑤ 调用 `thread.bindApplication(..., app.info, ...)` 方法通知（命令）客户进程运行指定的 Activity 所在的 APK 文件，这个“指定的 Activity”被包含在了参数 `app.info` 中。`info` 值的最初来源是根据调用者 `intent` 的内容从系统中查询到的 `ApplicationInfo` 信息，参见 10.1.3 节第 7 小节中的第三步。通知完毕后，顺便调用 `updateLruProcess()` 更新“最近调用的进程列表”内容。

⑥ 调用 `realStartActivityLocked()` 通知客户进程运行指定的 Activity。注意，上一步仅仅是告诉目标进程从 APK 文件中加载 `Application` 类并运行，但没有真正启动指定的 Activity，而这一步则会真正启动目标 Activity。

⑦ 查询是否有依赖于目标进程的等待 Service 或等待 Broadcast 服务，如果有的话一并通知目标进程执行之。这两个信息保存在变量 `mPendingService` 和 `mPendingBroadcast` 中。

除了以上这七步外，其中还包含一些其他不重要的细节，有兴趣的读者可以再研究源码。

截至本步结束，尽管客户进程已经存在，然而这个客户进程所包含的程序文件仅仅是 Framework 中的 ActivityThread 基类，而没有涉及应用程序本身的任何文件。这点类似于 Linux 程序的启动过程，即首先由操作系统产生一个空进程，然后再通知空进程去加载具体的要执行的程序文件。

## 11. 应用进程 launch 指定的 Activity

上一节结束后，目标进程（空进程）就需要根据 AmS 所提供的 ApplicationInfo 的信息来加载具体的 Activity 了。请注意 ApplicationInfo 是实现了 Parcelable 接口的类，所以才能在进程间传递数据。

在上一节中，目标进程向 AmS 报告自己已经启动，AmS 会执行 attachApplication()，而在该方法中则先后调用了两个 ActivityThread.ApplicationThread 的方法，分别是 scheduleLaunchActivity() 和 scheduleResumeActivity()，如以下代码片段所示。

```
5559 thread.bindApplication(processName, app.instrum
1822 app.thread.scheduleLaunchActivity(new I
```

这两个函数在 ActivityThread 中是异步执行的，即向 ActivityThread 所在的线程发送两个异步消息，并立即返回，而后 ActivityThread 再按顺序处理这两个消息。如图 10-9 所示。

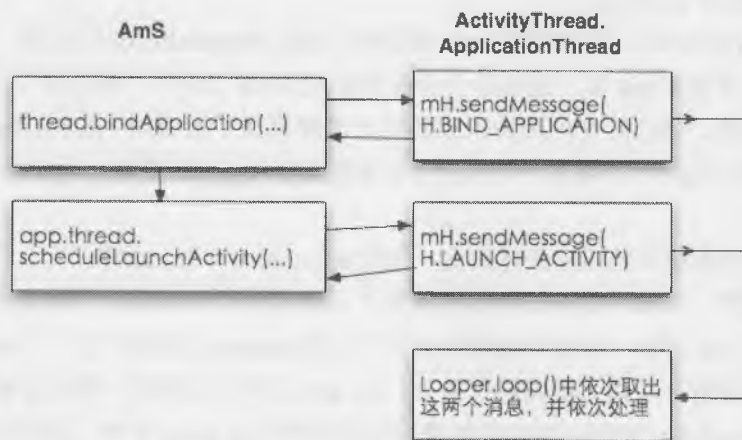


图 10-9 launch Activity 的流程

而这两个消息的处理函数分别是 handleBindApplication()和 handleLaunchActivity()，因此本节来分析这两个函数的处理过程。

前者的启动流程如图 10-10 所示。



图 10-10 handleBindApplication()的内部过程

① 重新设置时区和区域。为什么呢？因为所有的应用进程都是从 `zygote` 进程复制出来的，`zygote` 本身已经装载了系统资源，不同的区域对应了不同的资源内容，而在 `zygote` 装载系统资源时，系统区域是默认的，而此时用户有可能已经改变了系统区域，因此重新设置区域。当然，该步骤内部会判断当前区域是否和默认区域一样，如果不一样才会重新装载。

② 如果是 `debug` 模式，则弹出一个等待对话框，从这点可以看出，在调试应用程序时，界面上弹出的“Waiting for debugger”对话框其实是应用程序本身的一部分，而不是调试过程特有的。

③ 在一般情况下，应用程序都不会有自定义的 `Instrumentation` 类，只有一个例外，那就是 `Android` 中的单元测试 (`UnitTest`) 模块。`Android` 提供了一个单元测试框架，允许产生一个专门测试应用程序的单元测试 `APK`，该框架正是利用了自定义 `Instrumentation` 类，使得真正的应用程序在单元测试的“容器”中启动，关于 `UnitTest` 请参照 `Android` 官方 `sdk` 说明文档。

④ 调用 `makeApplication()` 创建 `Application` 类的对象。该函数主要是调用目标 `APK` 文件中的 `Application` 类对象，一个 `APK` 文件中只能有一个 `Application` 对象，程序员可以基于 `Application` 类定义



自己的类，并且在 AndroidManifest 的<Application>标签中使用 android:name 属性给出。如果程序没有自定义的 Application 类，系统会加载默认的 Application 类。

从这里可以看出，Application 类才是一个应用程序被加载后运行的第一个类，而该对象在应用程序中是全局唯一的，因此，对于一些需要全局保存的值可以放在该类中。

⑤ 如果该应用中包含 Provider 对象，则启动这些 Provider 对象。从这里可以看出，应用程序启动时 Provider 会先于 Activity 运行。

⑥ 调用 Application 类的 onCreate()方法，注意，这是 Application 类的，而不是 Activity 类的。

接下来再看 handleLaunchActivity()的执行过程。该函数的主要作用是最终加载指定的 Activity 并运行，其流程如图 10-11 所示。

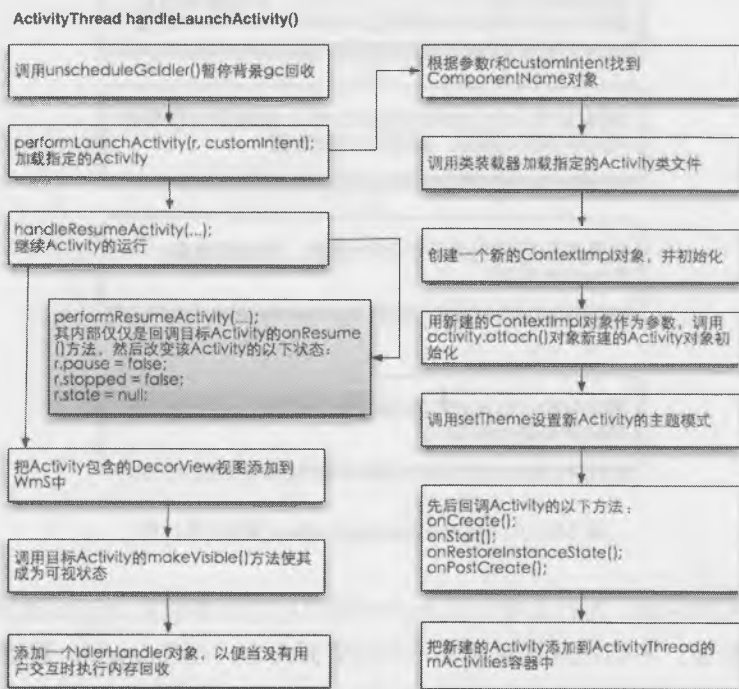


图 10-11 handleLaunchActivity()的内部过程

① 暂停背景运行的内存回收器。该回收器实际上是一个 Handler 对象，该对象会在系统空闲时接收回收消息，并执行相应的回收函数。关于内存回收参见第 10.2 节。

② 调用 performLaunchActivity()继续完成 Activity 的加载。该函数中：

(1) 根据调用参数创建一个 ComponentName 对象。

(2) 加载目标 Activity 类。

(3) 创建一个 ContextImpl 对象。每一个 Activity 都包含一个 ContextImpl 对象，该对象是一个轻量类。创建该对象后，调用 activity.attach()方法，用该 ContextImpl 对象初始化 activity 对象，实际上就

是把 ContextImpl 对象作为 Activity 的 base Context。

(4) 调用 `setTheme()` 为 Activity 设置主题模式。该主题是从 `AndroidManifest.xml` 中取得的，如果 `xml` 文件中没有声明自定义的 `theme`，则系统使用默认的主题模式。

(5) 先后调用目标 Activity 的 `onXXX()` 四个函数。

(6) 把新建的 Activity 对象加入到 ActivityThread 的 `mActivities` 列表中。这里需要注意，`mActivities` 列表中保存的是 Activity 在 AmS 中对应的 `HistoryRecord` 对象，而不是直接的 Activity 对象，添加代码如下：

```
2656 mActivities.put(r.token, r);
```

参数 `r.token` 正是 `HistoryRecord` 对象的引用，`r` 是本地的 `ActivityRecord` 对象，因此，如果是客户端程序中同一个 Activity 类，则有可能有多个对象。

③ 这个时候 `performLaunchActivity()` 方法返回，继续执行 `handleResumeActivity()`，该方法内部又调用 `performResumeActivity()` 方法，其内部仅仅是调用了目标 Activity 的 `onResume()` 方法，并改变了该 Activity 在 ActivityThread 中对应的 `ActivityRecord` 数据对象的 `pause`、`stopped` 变量。

④ 至此，对用户来讲，还没在界面上看到任何内容。尽管以上步骤中 ActivityThread 和 AmS 风风火火地打成一团，并装载了相关的执行类，然而这些都还没有通知给 WmS，而此时，一切就绪。因此本步骤中从 Activity 对象中找到其包含的 `DecorView` 对象，并把它添加到 WmS 中。

⑤ 调用目标 Activity 的 `makeVisible()` 函数，该函数其内部又会辗转和 WmS 进行各种调用，并最终导致 Activity 包含的 `DecorView` 显示到屏幕上，关于这一点请参照第 14 章。

⑥ 添加一个 `IdleHandler` 对象，因为在一般情况下，该步骤执行完毕后，Activity 就会进入空闲状态，所以可以进行内存回收。关于内存回收请参照第 10.2 节。

执行完这两个函数后，指定的 Activity 就真正启动了，用户此时应该可以看到该 Activity 的界面。

#### 10.1.4 stopActivityLocked() 停止 Activity

在前面几节中，从 A 启动到 B 时，仅说了需要先暂停 A，然后再启动 B，读者是否想过那么 A 到底是什么时候停止 (stop) 或者销毁 (destroy) 的呢？本节就来回答这个问题。

在 Activity 启动的过程中，AmS 一直没有通知相应的应用进程停止 (stop) Activity，而仅仅是暂停 (pause)。这对于 WmS 来讲已经足够了，因为 WmS 只需要知道把用户消息发向哪一个窗口就可以了，所以被暂停的 Activity 没有必要再执行停止操作。

然而，AmS 内部提出了一种主动内存管理机制，即 AmS 会主动根据当前运行的 Activity 的状态，在系统内部不够用的时候杀死优先级较低的应用以释放内存。而如何判断优先级的高低呢，这就需要为不同的 Activity 赋予不同的权值，而这个权值正是 Activity 的状态，包括正在运行、停止、销毁等。这就是为什么 Activity 提供了三个和停止相关的回调 (分别是 `onPause()`、`onStop()` 和 `onDestroy()`) 的原因，不同的回调发生在不同停止时机。比如当 AmS 希望当前的 Activity 暂停和用户交互时，就会执行



onPause()回调;当 AmS 希望把该 Activity 的权值改为 stop 状态时,则执行 onStop()回调;当 AmS 准备在后台杀死该 Activity 时,则执行 onDestroy()回调。

那么,接下来分析 Activity 是什么时候被停止,以及是怎样被停止的。

前面 10.1.3 节中讲述在 resume 具体的 Activity 时,当目标进程 ActivityThread 执行完 handleResumeActivity()时,会添加一个 IdleHandler 对象,如下代码所示:

```
3222         r.nextIdle = mNewActivities;
3223         mNewActivities = r;
3224         if (localLOGV) Slog.v(
3225             TAG, "Scheduling idle handler for " + r);
3226         Looper.myQueue().addIdleHandler(new Idler());
```

以上代码最后一句 addIdleHandler(new Idler())创建一个 Idler 对象,并添加到线程消息队列中。为什么要此时添加呢?因为 ActivityThread 已经完成了启动 Activity 的全部的工具,接下来就应该是 idle 状态了,因此可以执行一些 idle 行为,而 Idler 类的内部处理函数则是调用 WmS 的 activityIdle()方法,如下代码所示:

```
2169         try {
2170             am.activityIdle(a.token, a.createdConfig);
```

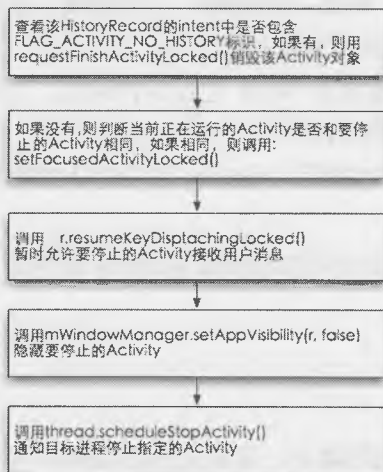


图 10-12 stopActivityLocked()的内部过程

而 AmS 中 activityIdle()的执行流程如图 10-12 所示,在该函数的内部则会辗转调用到 stopActivityLocked()方法。本小节暂不讨论这个辗转的过程,因为这更多的是和 AmS 内部的主动内存管理机制相关,因此把它放到本章第 10.2 节的内存管理中。而本小节则仅关系 stopActivityLocked()的内部执行过程,如图 10-12 所示。

① 处理 FLAG\_ACTIVITY\_NO\_HISTORY 标识。该标识的作用是不要让该 Activity 出现在 mHistory 中,即这是一次性的,比如 A→B→C,如果 B 的启动标识包含该标识,那么当从 C 返回后则会继续执行 A 而不是 B。

② 这一步实际上真的没有必要,因为既然要停止该 Activity,就完全没有必要再让它成为活动的。

③ 允许目标 Activity 接收用户消息,这一步也是多余的,原因同上。大家可以从代码中删除这两步重新生成 sdk,其运行结果是不受影响的,如下代码所示:

```
4052     } else if (r.app != null && r.app.thread != null)
4053     //         if (mFocusedActivity == r) {
4054     //             setFocusedActivityLocked(topRunningActi
4055     //         }
4056     //         r.resumeKeyDispatchingLocked();
```

④ 调用 setAppVisibility()隐藏目标 Activity,注意这里的第二个参数为 false,所以是隐藏。

⑤ 调用 thread.scheduleStopActivity()向目标进程的 ActivityThread 对象发送一个异步消息,通知目

标进程停止指定的 Activity。

执行完以上过程后，ActivityThread 中处理该异步消息的函数 handleStopActivity() 将开始执行，其执行过程如图 10-13 所示。

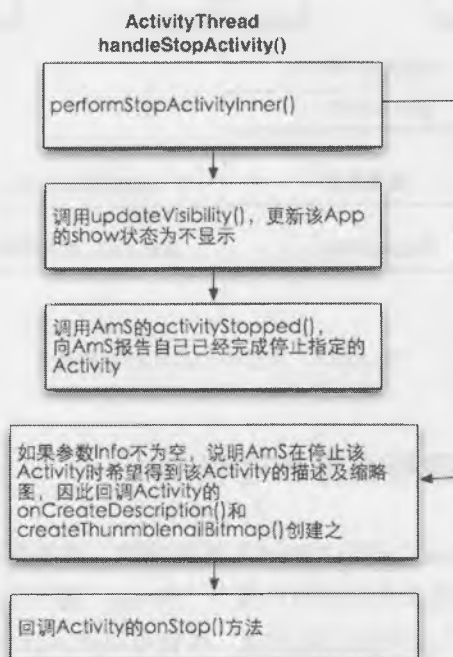


图 10-13 handleStopActivity()的内部流程

#### ① 调用 performStopActivityInner()。

(1) 如果参数 Info 不为空，则回调 Activity 对象的 onCreateDescription() 产生该 Activity 的描述，AmS 可能需要该描述进行一定的统计。info.thumbnail 是一个 Bitmap 型变量，既然该变量也要使用 IPC 进行传递，那么它必须是一个实现了 Parcelable 接口的类，这通过查看 Bitmap 类的代码原型得到了验证。

(2) 回调 onStop() 方法，默认的实现是什么都不做。

② 调用 updateVisibility() 方法。前面在 pause 的处理过程中，仅仅是告诉 WmS 屏蔽了对该 Activity 的消息派发，而该 Activity 对应的窗口却依然是显示的状态，而此时 stop 的含义则包括不显示该窗口，即 View v = r.activity.mDecor 窗口的现实状态被设为 false, v.setVisibility(false)。

③ 调用 AmS 的 activityStopped()，向 AmS 报告，自己已经完成了停止该 Activity。在 AmS 端，该函数内部比较简单，主要是调用 trimApplication() 进行一定的内存回收工作，这点可参照第 10.2 节。

对比以上步骤和 pause 的步骤发现，两者的主要区别有以下几点。

- 发生在不同的时机。pause 发生在目标 Activity 启动之前，而 stop 则发生在目标 Activity 启动之后。
- 回调 Activity 的不同函数。

- 被调方式不同。pause 是由 AmS “有意”调用的，而 stop 则是 AmS “无意”调用的。如图 10-14 所示。

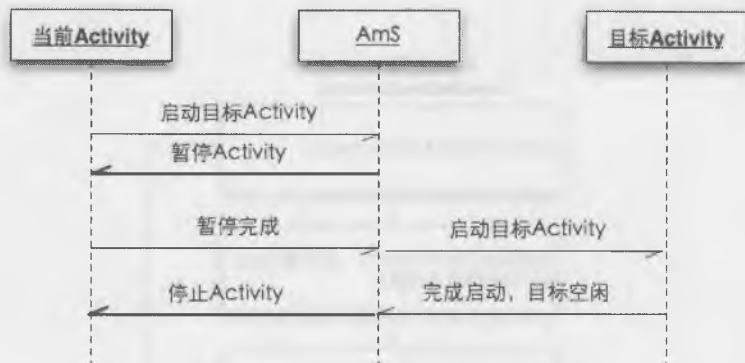


图 10-14 从暂停到启动的全过程

### 10.1.5 按“Home”键回到桌面的过程

前面曾说过，无论是按“Home”还是“Back”等系统键，最终都会导致执行 startActivit()，区别仅在于前端消息处理的过程。本节就来介绍当用户按下“Home”键后的前端处理过程。

在以往的应用程序开发中，大家可能已经发现，尽管可以在 Activity 的 onKey() 函数中截获“Back”键，但是却不能够截获“Home”键，其原因是 WmS 在进行消息派发时，对“Home”键做了默认处理，而没有把它派发到应用程序里面去。这纯粹是一种产品理念，Android 认为无论应用程序是什么样的，用户都可以通过“Home”键回到桌面程序，这一点和 iPhone 的设计理念相同。

Home 键的具体处理过程参见附图 3。

① 在 WmS 中，调用了 PhoneWindowManager 类中 interceptKeyTi() 方法截获所有消息，而在该函数内部处理了一些系统按键消息，这其中就包括“Home”键。当发现是“Home”键时，则调用 LaunchHomeFromHotKey() 方法。

(1) 判断当前是否处于锁屏状态，如果是，则什么都不做。此时用户应该按“Menu”键进行解除屏幕锁定。

(2) 判断是否处于解锁状态，如果是则弹出解锁对话框。这种情况一般仅发生在开机的时候，此时如果 SIM 卡有启动密码，则系统首先弹出一个对话框请用户输入密码。

(3) 在一般情况下，屏幕没有锁屏，并且也没有密码锁定，此时先通知 AmS 暂停程序切换，然后关闭系统对话框。

(4) 调用 startDockOrHome()，这个方法应该是在 Android2.0 版本以后增加的，系统提供了两个 Home 程序，一个是所谓的插座(Dock)方法，另一个是标准的 Home 程序。前者是当用户把手机插入到了 Android 专用的底座硬件中，而这则是通过底层提供 API 接口进行查询的。

① 调用 `createHomeDockIntent()` 方法构造一个能够启动 Dock 或者 Home 的 Intent 对象。

② 调用 `mContext.startActivity()` 启动 Intent 对应的 Activity。此处需要特别注意，这里是调用 `ContextImpl` 类的 `startActivity()` 方法，而不是调用 `Activity` 类的 `startActivity()` 方法。本章之前所讲的 `startActivity()` 均是指从 `Activity` 中执行，而在 `Activity` 的 `startActivity()` 中会调用 `startActivityForResult()` 方法，而该方法内部则继续调用 `Instrumentation` 类的 `execStartActivity()` 方法，而在 `ContextImpl` 中的 `startActivity()` 方法中也是调用 `Instrumentation` 类的 `execStartActivity()` 方法，但在调用之前检查了一下 Intent 中是否有 `NEW_TASK` 的标识，如以下代码所示：

```

612     public void startActivity(Intent intent) {
613         if ((intent.getFlags() & Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
614             throw new AndroidRuntimeException(
615                 "Calling startActivity() from outside of an Activity
616                 + " context requires the FLAG_ACTIVITY_NEW_TASK fl
617                 + " Is this really what you want?");
618         }
619         mMainThread.getInstrumentation().execStartActivity(
620             getOuterContext(), mMainThread.getApplicationThread(), nul
621     }

```

以上代码说明，调用 `startActivity()` 有两种方法，一种是从 `Activity` 中调用，另一种是直接从 `ContextImpl` 中调用，但是后者调用必须包含 `NEW_TASK` 的标识。这种限制的原因是，如果没有 `NEW_TASK` 标识，而且不是从 `Activity` 中调用，意味着此时 AmS 中可能会是一个空 Task，而这是不应该的。因此，在 `createDockHomeIntent()` 方法中，该 Intent 包含了 `NEW_TASK` 标识，如以下代码所示：

```

505     mDeskDockIntent = new Intent(Intent.ACTION_MAIN, null);
506     mDeskDockIntent.addCategory(Intent.CATEGORY_DESK_DOCK);
507     mDeskDockIntent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK
508         | Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);

```

`Instrumentation.execStartActivity()` 以后的流程就和本章前面讲的 `startActivity()` 完全相同了。

`mContext` 是从系统进程中创建的 `Context` 对象，该对象不与任何 `Activity` 对应。

总结“Home”键与普通 `startActivity()`，其本质区别在于前者构造了一个能够启动 Home 程序的特殊 Intent，并用该 Intent 调用 `ContextImpl` 类中的 `startActivity()`；而后者则是根据运行环境提供的 Intent，并从 `Activity` 类中调用 `startActivity()`。

### 10.1.6 按“Back”键回到上一个 Activity

“Back”键不是系统按键，但是“Back”键却有默认处理方法，因此，应用程序设计时可以在 `Activity` 的 `onKey()` 方法中截获该消息。

① 在 `Activity` 类的 `onKey()` 方法中，会检测是否是“Back”键，如果是，则调用 `onBackPressed()` 方法。

② 该方法则会调用 `finish()` 方法, 该方法也就是应用程序设计时所使用的 `finish()` 方法, 也就是说用户按下 “Back” 键和程序员主动调用 `finish()` 的作用完全相同。

③ 在 `finish()` 方法中, 远程调用 AmS 的 `finishActivity()` 方法。

至此, Activity 的任务就完成了, 剩下的就是 AmS 执行 `finishActivity()` 方法了。

在 AmS 端的 `finishActivity()` 的执行过程如附图 3 所示。

① 调用 `requestFinishActivityLocked()`。

② 判断该 Activity 是不是最后一个 Activity, 判断的方法如以下代码所示:

```
4101 if (!p.finishing && p != r) {
```

这里正是利用了 `mHistory` 列表的 `z-order` 排列顺序, 即最后一个总是当前正在运行的 Activity。比如  $A \rightarrow B \rightarrow C$ , 那么 A 是第一个启动的 Activity, B 是第二个, C 是第三个, 也是当前正在运行的 Activity, 以上代码中的两个条件是指 p 没有结束, 并且 p 不是要停止的 Activity, 因为 `mHistory` 中最后一个 Activity 肯定是当前要 finish 的, 所以 A 和 B 都不等于 C, 这就意味着 C 不是最后一个 Activity。如果不是最后一个 Activity, 则可以正常继续执行 `finishActivityLocked()` 结束该 Activity。

③ 如果 C 是最后一个 Activity, 并且 C 是 Home 类型的 Activity, 那么什么都不做, 因为 Home 类型的 Activity 不应该返回到其他应用程序中。但如果不是 Home 类型, 并且是最后一个 Activity 会如何呢? 答案是依然会继续执行 `finishActivityLocked()` 方法, 实际上在 `finishActivityLocked()` 内部的确会结束该 Activity, 只是在结束后会判断 `mHistory` 中是否有 Activity, 如果没有, 则会重新启动 Home 类型的 Activity。

调用 `finishActivityLocked()` 时, 参数 `r` 代表要停止的 `HistoryRecord` 对象, `index` 代表该对象在 `mHistoryRecord` 中的位置。

(1) 改变该 `HistoryRecord` 的 `finishing` 状态, 当然如果该状态已经为 `true`, 意味着已经发出停止的命令了, 所以会返回 `false`。

(2) 如果该 Activity 的上面还有其他的 Activity, 则把它上面的 `HistoryRecord` 的对象 `next` 的 `frontOfTask` 状态改为 `true`。这种情况一般很少发生, 因为 `finishActivityLocked()` 只有在按 “Back” 键的时候才被调用, 而此时要停止的 Activity 正是当前正在运行的 Activity, 所以 `index` 始终等于 `mHistory.size() - 1`。只有一种例外, 那就是程序员主动调用 `finish()` 方法, 而且目标对象是非活动的 Activity。如果这种情况发生, 则必须把它上面的 `HistoryRecord` 的 `frontOfTask` 置为 `true`, 该变量的含义是该 Activity 是否为 Task 中的第一个 Activity。比如, 假设当前 `mHistory` 的列表如下:  $A1 \rightarrow A2 \rightarrow B1 \rightarrow B2 \rightarrow B3 \rightarrow C1 \rightarrow C2$ , A、B 和 C 代表三个不同的 Task, 此时如果要停止 B1, 显然 B1 是 B 任务中的第一个 Activity 对象, 停止后 B2 将取而代之, 因此必须把 B2 的 `frontOfTask` 置为 `true`。

(3) 暂停对该 `HistoryRecord` 的消息派发。

(4) 查看当前是否有正在运行的 Activity, 如果有并且就是要停止的 Activity 对象, 则调用 `startPausingLocked()` 先暂停该 Activity 对象。

(5) 如果当前正在运行的 Activity 不是要停止的 Activity, 则直接调用 `finishCurrentActivityLocked`

(三个参数版本)完成停止工作。该函数内部实际上牵扯到 AmS 的内存回收机制,因此,把它放在第 10.2 节里面详述。

接下来继续分析以上(4)中的 `startPausingLocked()`内部的过程。在前面介绍 `startActivity()`的启动过程中,如果当前正在执行某个 Activity,也是首先调用 `startPausingLocked()`暂停当前 Activity,该方法与外部方法的调用关系参见附图 3,此处截取其内部过程如图 10-15 所示。

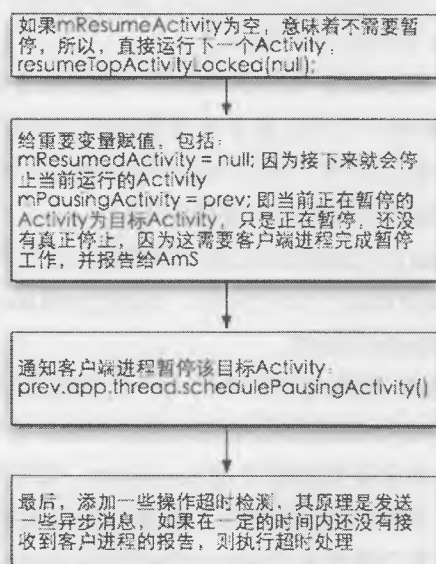


图 10-15 `startPausingLocked()`的内部流程

- ① 判断 `mResumeActivity` 是否为空,在一般情况下,该值肯定不会为空。
- ② 把 `mResumeActivity` 设为空,并给 `mPausingActivity` 赋值 `prev`,而 `prev` 正是当前需要暂停的 Activity。
- ③ 通知客户进程暂停该 Activity 对象。
- ④ 发送一个超时检测消息。

至此,AmS 就完成了暂停操作,然后就会等待客户进程暂停目标 Activity 后,报告 AmS 暂停完成,而 AmS 会继续从 `activityPaused()`方法处开始执行。

### 10.1.7 长按“Home”键

长按“Home”键也是系统消息,应用程序无法对长按“Home”键的消息做自定义处理。

- ① 在 `PhoneWindowManager` 的 `interceptKeyTi()`方法中检测是否是长按“Home”键的消息,如果是,则发送一个异步消息。处理该异步消息的是一个线程对象,名称为 `mHomeLongPress`,该对象是在构造



函数中创建的，其作用就是用户所看到的弹出一个对话框，里面包含了最近启动的应用列表。而该线程内部的执行过程具体如下：

- (1) 调用 `performHapticFeedback()` 给用户一个震动提醒，Haptic 的意思是“触觉反馈”。
- (2) 关闭所有系统窗口，常见的系统窗口包括状态栏、Toast 窗口、系统警告或者错误对话框等。
- (3) 调用 `showRecentDialog()` 方法弹出最近任务列表对话框。而该方法内部则是首先创建一个 `RecentApplicationsDialog` 类对象，该对象基于 `Dialog` 类，然后再调用 `show()` 方法将该对话框显示出来。

关于 `RecentApplicationsDialog` 类的执行过程如下。

① 在该类的构造函数中，为该 `Dialog` 对象添加视图，使用的是系统内置的 `recent_apps_dialog.xml` 布局文件。该文件中包含了六个 `Button` 视图，如果读者想自定义该视图，则可以修改该文件。

② 在该 `Dialog` 对象的 `onStart()` 方法中，调用 `reloadButtons()` 为这六个 `Button` 视图设置具体的显示内容及 `onClick()` 事件。`reloadButtons()` 方法的内部流程如下。

(1) 调用 `AmS` 的 `getRecentTask()` 方法获取最近的 `Task` 列表，这个列表来源于 `AmS` 中的 `mRecentTask` 列表。但是并不是直接返回 `mRecentTask` 里面的内容，而是把 `mRecentTask` 里面的 `TaskRecord` 数据类转换为 `ActivityManager.RecentTaskInfo` 数据类。

(2) 根据返回的 `RecentTaskInfo` 设置对话框中的六个按钮的显示和 `onClick()` 事件。显示包括任务名称和任务图标，而 `onClick()` 事件主要则是获取任务对应的 `Intent` 对象。

需要注意的是，任务的 `Intent` 对象是指该任务的 `baseIntent`，即创建该任务时的 `Intent`。同时，还把这个 `baseIntent` 添加了一个 `FLAG_NEW_TASK` 标识，添加这个标识的原因有两个，第一是系统希望在一个新的任务中启动 `baseIntent`，第二是当从 `Dialog` 类中调用 `startActivity()` 时只能使用 `Context` 类的 `startActivity()`，而 `Context` 的 `startActivity()` 方法要求 `Intent` 的标识中必须包含 `NEW_TASK`。

此时，长按“Home”键的操作就告一段落，接下来，如果用户单击任务对话框中的某个 `Button` 后，就会调用 `getContext().startActivity()` 启动对应的 `Activity`。注意，这里是使用 `Context` 中的 `startActivity()` 方法，而不是 `Activity` 的 `startActivity()` 的方法。

### 10.1.8 Activity 生命期的代码含义

在过去的应用程序开发中，读者大多数已经了解了 `Activity` 生命期中的几个主要状态，并知道如何在这些状态中做不同的事情。但多多少少还是存在一些疑惑，比如 `start` 和 `stop` 状态从代码的意义上讲，差别到底在哪里。尽管你可能会说：“`stop` 代表了 `Activity` 的停止，而 `start` 代表了 `Activity` 的开始”，那么，问题是“开始”和“停止”的差别又在哪里？诸如此类，本节就来揭示这些不同状态背后所隐藏的代码级别的差别。

首先，不同的状态对于应用程序开发者来讲是通过不同的回调函数来区分的，因此，下面先列出所有和状态有关的回调函数，如表 10-4 所示。

表 10-4 Activity 中所有和状态相关的回调函数

回调函数名称	使用场合
onCreate()	Activity 实例被创建后第一次运行时
onNewIntent()	<p>有两种情况会执行该回调：</p> <ul style="list-style-type: none"> <li>● Intent 的 Flag 中包含 CLEAR_TOP，并且目标 Activity 已经存在当前任务队列中。</li> <li>● Intent 的 Flag 中包含 SINGLE_TOP，并且目标 Activity 已经存在当前任务队列中。</li> </ul> <p>前者和后者的区别在于，举例如下：当前任务队列为 ABCD，此时目标 Activity 为 B，那么前者会把队列改变为 AB，而后者则会改变为 ABCDB；但假设当前为 ABCD，目标为 D，前者则会改变为 ABCD，后者则还是 ABCD，而不会重新创建 D 对象，即 SINGLE_TOP 只对目标在 top 上时才有效</p>
onStart()	Activity 从 stop 状态重新运行时
onRestore InstanceState(Bundle savedInstanceState)	与 onPostCreate()相同，只是先于 onPostCreate()调用
onPostCreate()	如果 Activity 实例是第一次启动，则不调用，否则，以后的每次重新启动都会调用
onResume()	Activity 继续运行时
onSave InstanceState(outState)	与 onPause()相同，只是会先于 onPause()调用
onPause()	Activity 暂停时被调用。导致暂停的原因除了 onStop()中描述四个原因外，还包括一个，即当用户长按“Home”键出现最近任务列表时，此时正在运行的 Activity 将被执行 onPause()
onCreateDescription()	仅是要停止 Activity 时调用，先于 onStop()
onStop()	<p>一般会导致变为 stop 状态的原因有以下几个：</p> <ul style="list-style-type: none"> <li>● 用户按“Back”键后</li> <li>● 用户正在运行 Activity 时，按“Home”键</li> <li>● 程序中调用 finish()后</li> <li>● 用户从 A 启动 B 后，A 就会变为 stop 状态</li> </ul>
onDestroy()	<p>当 Activity 被销毁时，销毁的情况包括：</p> <ul style="list-style-type: none"> <li>● 当用户按下“Back”键后</li> <li>● 程序中调用 finish()后</li> </ul>

下面，按照 AmS 的调度过程重新总结以上回调函数在不同场合下的被调过程，如表 10-5 所示。

表 10-5 不同场合下回调函数被调用的顺序

场合描述	回调过程
创建 Activity 对象并启动	onCreate(); onStart(); onRestoreInstanceState(); onPostCreate(); onResume();



(续表)

场合描述	回调过程
Activity 对象已经存在并继续运行	onStart(); onResume();
Activity 对象已经存在, 并且满足表 10-4 中 onNewIntent() 的条件	onNewIntent(); onStart(); onResume();
AmS 中调用 startPausingLocked()	onSaveInstanceState(); onPause();
AmS 空闲时调用 stopActivityLocked()	如果没有暂停则先调用: onSaveInstanceState(); onPause(); 然后调用: onCreateDescription(); onStop();
AmS 空闲时调用 destroyActivityLocked()	如果没有暂停则先调用: onSaveInstanceState(); onPause(); 如果没有停止则继续调用: onCreateDescription(); onStop(); 然后调用: onDestroy();

以上就是 Activity 不同生命期对应的回调函数的具体执行过程。

## 10.2 内存管理

Android 中的内存管理分为两个部分。第一部分是当应用程序关闭后, 后台对应的进程并没有真正退出, 以便下次再启动时能够快速启动; 第二部分是, 当系统内存不够用时, AmS 会主动根据一定的优先规则退出优先级较低的进程。下面就分别讲述这两个部分的细节内容。

### 10.2.1 关闭而不退出

“关闭”仅仅是使其对应的窗口不显示, 而对应的进程却会一直保存。于是有的读者就开始怀疑, 如果后台有很多进程同时存在的话, 运行速度会变慢。事实上, 这种机制除了占用内存外, 基本上不会

降低前台程序的运行速度，下面就来分析为什么。

每个应用程序的主体都对应一个 `ActivityThread` 类，该类初始化后，就进入 `Looper.loop()` 函数中无限循环，如以下代码所示：

```
4622     Looper.prepareMainLooper();
4623
4624     ActivityThread thread = new ActivityThread();
4625     thread.attach(false);
4626
4627     Looper.loop();
```

以后则依靠消息机制运行，即当有消息处理时处理消息，而当没有消息时进程会进入到 `sleep` 状态。`loop()` 方法的内部代码如下所示：

```
1065     public static final void loop() {
107         Looper me = myLooper();
108         MessageQueue queue = me.mQueue;
109         while (true) {
110             Message msg = queue.next(); // might block
```

在 `queue.next()` 方法中，会从消息队列中取消息，如果没有消息，该函数内部则会导致当前线程进入 `sleep` 状态，并且只有新消息到达后 `next()` 函数才继续执行并返回新的消息。为了验证这个结论，读者可以在 110 行的下一行代码处设一个断点，然后 debug 当前正在运行的任何应用程序，其结果是显而易见的。如果用户不和当前的 `Activity` 交互，程序是执行不到断点的，而当用户点击屏幕后，则会立即执行到断点。用户没有点击屏幕时，在 `ddms` 中查看应用进程的状态则显示为 `wait`，如图 10-16 所示，该图显示的是线程 id 为 134 的 `com.android.launcher` 应用程序。

com.android.launcher		134	8603 / 8700	
ID	Tid Status	utime	stime	Name
1	134 wait	372	94	main
*2	135 vmwait	14	34	HeapWorker
*3	138 vmwait	0	0	Signal Catcher
*4	144 running	11	17	JDWP
5	152 native	1	0	Binder Thread #1
6	153 native	0	0	Binder Thread #2

图 10-16 从 `ddms` 中查看线程 id 为 134 的线程状态

除此之外，也可以使用 `adb shell` 进入设备终端，然后使用 `ps` 命令查看当前进程的状态，同样可以发现当前进程为 `sleep` 状态，如图 10-17 所示。

```
root@android:/ # ps 134
USER      PID    PPID  VSIZE  RSS      WCHAN    PC         NAME
app_14    134    33    108136 23856    ffffffff afd0e828 S com.android.launcher
```

图 10-17 从 `adb shell` 中查看线程 id 为 134 的线程状态

而在 Linux 的内核调度中，如果一个线程的状态为 `sleep`，则除了占用调度本身的时间外，本身则不会占用 CPU 的时间片。因此，在 100 以内的进程数目基本上不会影响当前进程的执行速度，换句话说，系统运行一个进程与 100 个进程的速度是相同的，只要其他 99 个线程都处于 `sleep` 状态。所以，假如有人说他可以提供一个进程查看器，在里面可以杀死不用的进程以提高系统运行速度，读者别相信这个。

有的读者就要问了，当前进程都处于 `sleep` 状态了，那还怎么执行啊？这正是所谓的消息驱动机制，`queue.next()` 方法可以被以下三种情况重新唤醒。

定时器中断。如果应用程序中设置了定时器任务，那么当定时器发生时，操作系统会回调该定时器任务，程序员一般会在这个定时器任务中向 `looper` 主线程中发送一个消息，从而 `next()` 方法会被重新唤醒，并返回所获得的消息。

用户按键消息。当有用户按键消息时，`WmS` 中的专门处理输入消息的线程会把这个消息发送到 `looper` 主线程，这就是上面例子中所说的，当点击屏幕时会从 `next()` 方法中跳出。

Binder 中断。Binder 是 Linux 系统中的一个驱动设备，应用程序中可以包含一个 Binder 对象，该对象会在应用程序中自动创建一个相应的线程，当 Binder 驱动接收到 Binder 消息，并派发到客户端的 Binder 对象后，Binder 线程会开始执行。如果在 Binder 的消息处理中向 `looper` 主线程发送一个消息，而 `next()` 会继续执行。

以上三种情况的关系如图 10-18 所示。

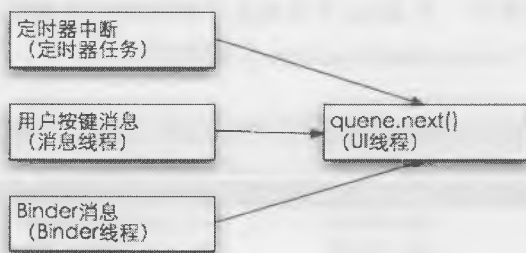


图 10-18 各种能触发 MessageQueue 从 `next()` 函数返回的情况

基于以上认识，Android 系统设计时才考虑了让应用程序“关闭”而“不退出”，因为这不会降低程序运行的速度。当然，如果所有的应用程序都是某种服务，而不是 UI 程序，那么在这种设计模式下，当应用程序增加时，每个程序的运行速度会立即变慢，CPU 会被这些应用程序“瓜分”。幸运的是，手机不是服务器，大多数应用程序都是消息驱动模式，没有消息时程序就会 `sleep`，直到产生消息。

### 10.2.2 Android 与 Linux 的配合

关于内存回收的时机，Android 官方文档提出，Activity 所占用的内存一般情况下不会被回收，

只有在系统内存不够用的时候才会回收，并且回收时会有一个潜规则。大致情况是前台 Activity 最后回收，其次是包含前台的 Service 或者 Provider，再次是后台 Activity，最后是后台空进程。而这种机制就意味着 AmS 需要和 Linux 操作系统有个约定，因为“系统内存是否够用”属于 Linux 内核的内存管理控制的事情，AmS 是无法预知的。

你可能会想，AmS 是运行在 Java 环境中的，当 Java 进行内存分配时，如果发生 Out Of Memory 不就不知道系统内存低了吗？事实并非如此，这有两个原因：

- 尽管 AmS 运行在 Java 环境里，但是应用程序和 AmS 运行在两个独立的 Java 虚拟机中，应用程序申请内存并不会通知 AmS，所以 AmS 无法感知应用程序申请内存的状况，除非每次应用程序申请内存并发生 Out Of Memory 时通知 AmS，但实际却没有。
- Java 虚拟机运行时都有各自独立的内存空间，应用程序 A 发生 Out Of Memory 并不意味着应用程序 B 也会发生 Out Of Memory，很有可能仅仅是 A 程序用光了自己内存的上限，而系统内存却还是有的。

以上说明，单纯的 AmS 是无法获知系统内存是否低的。

那么，什么是“系统内存低”或者“系统内存不够用”呢？从 Android 底层的 Linux 来讲，由于并未采用磁盘虚拟内存机制，所以应用程序能够使用的内存大小完全取决于实际物理内存的大小，所以，“内存低”的含义就是实际物理内存已经被用得所剩无几了。而这又是如何与 AmS 联系到一起的呢？如图 10-19 所示。

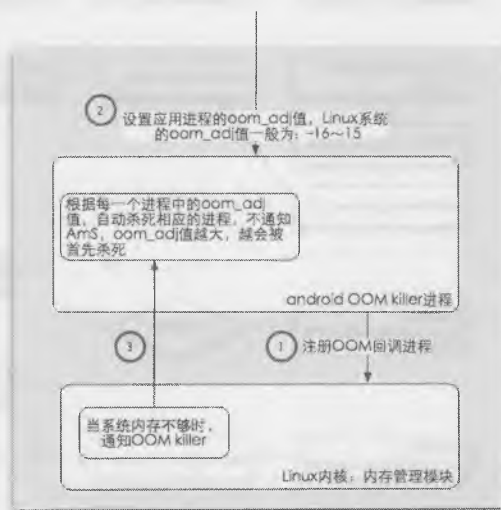


图 10-19 Android 中内存 OOM 机制示意图

在 Android 中运行了一个 OOM 进程，即 Out Of Memory。该进程启动时会首先向 Linux 内核中把自己注册为一个 OOM Killer，即当 Linux 内核的内存管理模块检测到系统内存低的时候就会通知已经注册的

OOM 进程，然后这些 OOM Killer 就可以根据各种规则进行内存释放了，当然也可以什么都不做。

Android 中的 OOM Killer 进程是仅仅适用于 Android 应用程序的，该进程在运行时，AmS 需要把每一个应用程序的 oom\_adj 值告知给 Killer。这个值的范围在 -16 到 15，值越低，说明越重要，这个值类似于 Linux 系统中的进程 nice 值，只是在标准的 Linux 中，有其自己的一套 Killer 机制。

当发生内存低的条件时，Linux 内核管理模块通知 OOM Killer，Killer 则根据 AmS 所告知的优先级，强制退出优先级低的应用进程。

以上就是 Android 和 Linux 关于内存管理的配合，分析完相关代码可知，OOM Killer 是一种可选的方案，对于 OEM 厂商来讲，完全可以不支持 OOM，只有在不支持的情况下，Android 才会采用之前所声称的“潜规则”优先退出后台的 Activity。这个结论很重要，因为在现在主流的 Android 平台中都包含了 OOM Killer 模块，这就意味着 AmS 仅仅是根据应用进程前后台的关系给其分配一个 oom\_adj 值，剩下的就是 oom\_killer 要干的事情了。

### 10.2.3 各种关闭程序的过程

为了总体了解内存回收的细节，本节先对和内存回收相关的程序关闭过程作一个总体概括，如图 10-20 所示。

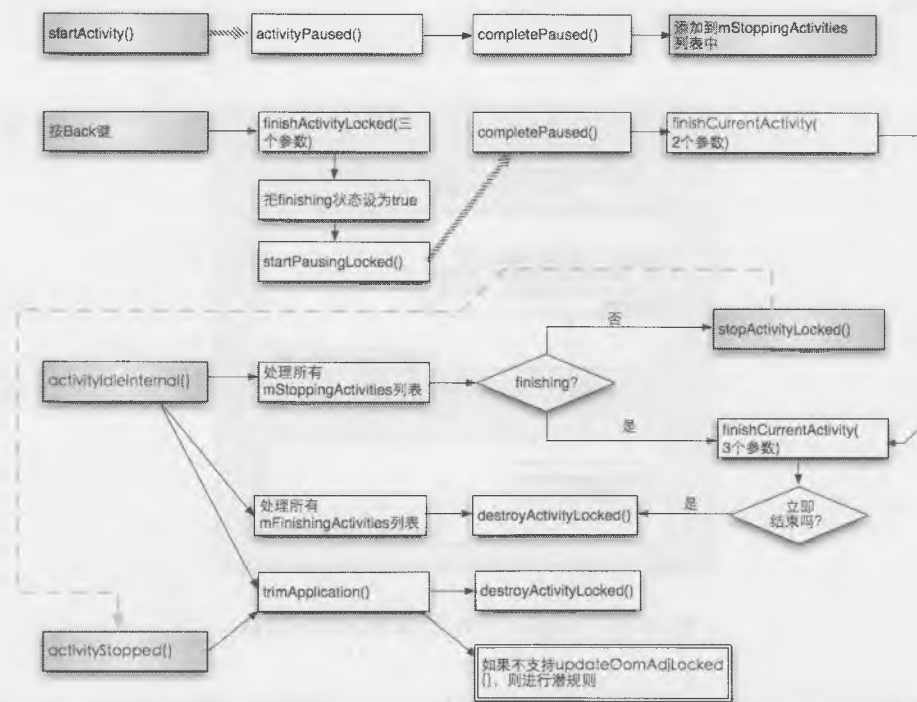


图 10-20 各种关闭当前 Activity 操作引起的内存回收流程

关闭程序的所有过程包含以下三种情况。

第一种，从调用 `startActivity()` 开始，在一般情况下，当前都有正在运行的 Activity，所以需要先暂停当前 Activity，而暂停完毕后，AmS 会收到一个 Binder 消息，并开始从 `completePaused()` 处执行。在该函数中，由于上一个 Activity 并没有 finishing，仅仅是 stop，所以这里会把上一个 Activity 添加到 `mStoppingActivities` 列表中。当目标 Activity 启动后，会向 AmS 发送一个请求进行内存回收的消息，这会导致 AmS 在内部调用 `activityIdleInternal()` 方法，该方法中首先会处理 `mStoppingActivities` 列表中 Activity 对象，这就会调用到 `stopActivityLocked()` 方法。这又会通过 IPC 调用，通知应用进程 stop 指定的 Activity，当 stop 完毕后，再报告给 AmS，于是 AmS 再从 `activityStopped()` 处开始执行，而这会调用 `trimApplications()` 方法，该方法中则会执行和内存相关的操作。

第二种，当按“Back”键后，会调用 `finishActivityLocked()`，然后把该 Activity 的 finishing 标识设为 true，然后再调用 `startPausingLocked()`，当目标 Activity 完成暂停后，就会报告 AmS，此时 AmS 又会从 `completePaused()` 处开始执行。与第一种情况不同，由于此时暂停的 Activity 的 finishing 状态已经设置为 true，所以会执行 `finishingActivityLocked()`，而不是像第一种情况中仅仅把该 Activity 添加到 `mStoppingActivities` 列表。

第三种，当 Activity 启动后，会向 AmS 发送一个 Idle 消息，这会导致 AmS 开始执行 `activityIdleInternal()` 方法。该方法中会首先处理 `mStoppingActivities` 列表中的对象，接着处理 `mFinishingActivities` 列表，最后再调用 `trimApplication()` 方法。

以上即为所有关闭程序的情况，包括 stop 和 destroy，客户进程中与之对应的就是 `onStop()` 和 `onDestroy()` 的调用。

而从内存回收的角度来看，释放内存的地点包含三个。

- 第一个是在 AmS 中进行，即 Android 所声称的当系统内存低时，优先释放没有任何 Activity 的进程，然后释放非前台 Activity 对应的进程。
- 第二个是在 OOM Killer 中，此时 AmS 只要告诉 OOM 各个应用进程的优先级，然后 OOM 就会调用 Linux 内部的进程管理方法杀死优先级较低的进程。这个部分不在本书讲解的范围内。
- 第三个是在应用进程本身之中，当 AmS 认为目标进程需要被杀死时，首先会通知目标进程进行内存释放，这包括调用目标进程的 `scheduleLowMemory()` 方法和 `processInBackground()` 方法。

#### 10.2.4 释放内存详解

上一节介绍了各种关闭、退出程序的过程，本节介绍这些操作过程中所导致的各种具体内存回收的细节，如附图 4 所示。

##### 1. activityIdleInternal()

首先来看 `activityIdleInternal()` 方法的调用，该方法被调用的时机如图 10-21 所示。



图 10-21 引起 activityIdleInternal()被调用的各种原因

最常见的调用该方法的情况发生在目标 Activity 启动后, 这是在 ActivityThread 中调用的, 即当指定的 Activity 启动后会报告 AmS, 因为此时上一个 Activity 仅仅处于 pause 状态, 还没有 stop 或者 destroy。这也是 AmS 的调度特点之一, 即优先启动下一个 Activity, 只有当启动后, 才通知 AmS 去 stop 或者 destroy 上一个 Activity, 从而让用户能在最短的时间内看到下一个 Activity 的界面。ActivityThread 中调用该方法的代码如下所示:

```

3226     Looper.myQueue().addIdleHandler(new Idler());
2176     am.activityIdle(a.token,

```

- 首先在 handleResumeActivity()方法中添加一个 Idler 对象, 然后在该 Idler 对象的 queueIdle()方法中调用 AmS 的 activityIdle()方法。

除此之外, activityIdle()还会在另外两种情况下被调用, 这两种情况都是向 AmS 所在的线程发送一个异步 IDLE\_NOW\_MSG 消息, 而消息的处理函数正是 activityIdle()。

- 一种情况是在 HHistoryRecord 类中的 windowVisible()函数中调用, 而这个函数是从 WmS 中调用的, 即当目标窗口由“非显示”变为“显示”状态时, 会告诉 AmS。因为当窗口显示后如果没有用户操作的话, 可认为是空闲状态, 这有点类似于在 ActivityThread 中 resume 完指定的 Activity 后也被认为是空闲状态一样。有的读者可能就要问了: 对于一般的 Activity 而言, 既然 resume 后要通知 AmS 进行空闲回收, 而当其窗口显示出来后也要通知 AmS 进行空闲回收, 岂不是有点重复? 原因是有些窗口仅仅是窗口, 而不对应任何 Activity, 并且 activityIdle()方法内部并不是仅仅回收刚刚暂停的 Activity, 而是整个系统内部的状态检查。
- 另一种情况是 completePausedLocked()方法中, 在正常的操作中, 似乎不会执行到这段代码, 如下所示:

```

2217         mStoppingActivities.add(prev);
2218         if (mStoppingActivities.size() > 3) {
2219             // If we already have a few activities waiting
2220             // then give up on things going idle and start
2221             // them out.
2222             if (DEBUG_PAUSE) Slog.v(TAG, "Too many pending :");
2223             Message msg = Message.obtain();
2224             msg.what = ActivityManagerService.IDLE_NOW_MSG;
2225             mHandler.sendMessage(msg);
2226         }

```



这段代码意思是，如果当前要 stop 的 Activity 数量超过 3 时，才发送 IDLE\_NOW\_MSG 消息。而实际系统运行时，由于每一次 activity 启动后，都会调用到 AmS 的 activityIdle() 方法，而该方法中则会处理 mStoppingActivities 列表中的对象。所以，单次 completePausedLocked() 调用时，一般不会积攒到 3 个，而仅仅是 1 个，所以这段代码似乎永远执行不到。

以上三种情况都是通过 activityIdle() 间接调用到 activityIdleInternal() 方法，另外一种调用到 activityIdleInternal() 方法的情况是发生在 IDL\_TIMEOUT\_MSG 消息中。

- 第四种情况，该消息是在 completeResumeLocked() 方法中发出的，其作用是检测启动超时。即如果规定的时间 (IDLE\_TIMEOUT) 内不能启动指定的 activity，AmS 中设置的是 10 秒，则会调用 activityIdleInternal() 释放相关资源。completeResumeLocked() 方法是在 resumeTopActivityLocked() 中被调用的。

以上四种情况就是全部调用到 activityIdleInternal() 的情况，下面就来分析该函数内部是如何实现内存回收的。

① 通知所有需要回收内存的客户进程进行内存回收。这些客户进程列表保存在 mProcessesToGc 列表中，关于客户进程中如何进行内存回收参照 10.2.4 节。

② 取出 mStoppingActivities 列表中的内容，并存放临时列表 stops 中，再取出 mFinishingActivities 列表中的内容，并存放临时列表 finishes 中，然后删除原有的记录。这里并不是简单的列表内容复制，而是要判断里面 HistoryRecord 的状态，源码比较简单。

③ 首先对 stops 列表中的内容进行处理，在该处理过程中，有一个判断列表中 Activity 的 finishing 状态是否为 true 的条件。这里需要注意，有的读者可能会觉得奇怪，因为 stops 列表的内容来源于 mStoppingActivities 中，该列表的内容一般是指 finishing 状态为 false，并且 stopped 状态为 true，既然这两个状态都是事先确定的，那么为什么此处还要判断呢？须知，mStoppingActivities 列表中的对象的 finishing 状态不一定全部都是 false，比如在按下“Back”键后，finishing 状态 true，接着在 completePaused() 中调用到了 finishCurrentActivity (3 个参数) 函数，该函数中则会指定把指定的 Activity 添加到 mStoppingActivities 列表中，显然，此时该 Activity 的 finishing 状态为 true。

在该步中，如果 finish 状态为 false，则调用 stopActivityLocked() 通知客户进程停止该 Activity，这种情况一般发生在调用 startActivity() 后。如果 finish 状态为 true，则执行 f2()、f3() 函数，在 f3() 函数内部则要判断是不是要立即停止，如果要立即停止则调用 destroyActivityLocked() 通知目标进程调用 onDestroy() 方法，否则，先调用 resumeTopActivity() 运行下一个 Activity。

④ 接着对 finishes 列表中的对象进行处理。由于 finishes 列表中的对象其 finishing 状态都为 true，所以，可以直接调用 destroyActivityLocked() 通知客户进程销毁目标 Activity。

以上四步中，除了第一步是通知客户进程进行主动内存回收外，其他三步仅仅是针对不同状态通知客户进程执行不同的回调而已，而没有进行任何内存回收工作。如果 Android 的程序员愿意，他们可以给这三个步骤起任何一个名称，换句话说，stop 和 destroy 仅仅是一个状态的变化，而没有在真正意义上对内存进行实质性的停止或者销毁。

- ⑤ 调用 trimApplications()。该步骤内部真正地进行了内存回收工作，包括杀死不必要的进程等，



详见以下两个小节。

## 2. trimApplications()

该方法除了在 `activityIdleInternal()` 中被调用外，也在 `stopActivityLocked()` 中被调用，其内部执行流程如附图 4 所示。以下为了叙述方便，截取和该函数相关的大图片段，如图 10-22 所示。

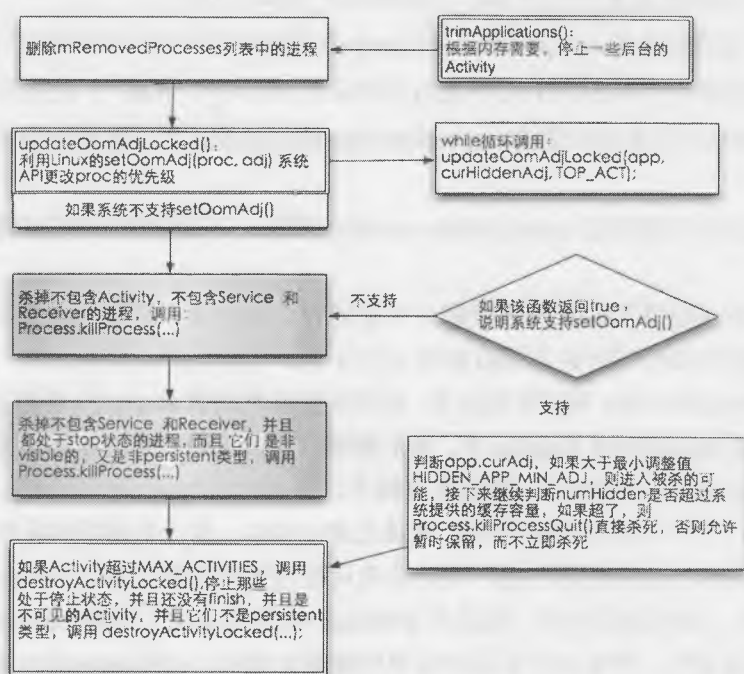


图 10-22 trimApplication()的内部调用过程

- 1 删除 `mRemovedProcesses` 列表中包含的应用进程，该列表的内容来自四种情况。
  - 第一种是当某个进程 `crash` 后，会被添加到该列表。
  - 第二种是当某个程序的 UI 线程在 5 秒之内没有响应时，系统会弹出一个 ANR 对话框，此时如果用户选择强制关闭，该进程就会被添加到该列表。
  - 第三种是当程序员调用 `AmS` 提供的 `killBackgroundProcess()` 方法时，调用者必须包含 `KILL_BACKGROUND_PROCESSES` 权限。这说明，程序员调用 `killBackgroundProcess()` 方法后并不会同步杀死指定进程，而仅仅是添加到列表。
  - 第四种是当系统启动时，`AmS` 的 `systemReady()` 方法中，如果发现启动了非 `persistent` 类型的应用进程，则把这些进程添加到列表中，这种情况基本上不会发生。
- 2 调用 `updateOomAdjLocked()` 方法，该方法的作用是告诉 OOM Killer 指定进程的优先级，即

oom\_adj 值，该值的范围为 -16~15，Android 中的进程仅仅使用了 0~15。值越小说明优先级越高，越不能被优先杀死。该函数的返回值类型为 boolean，如果底层的 Linux 系统包含 OOM Killer 则返回 true，否则返回 false。关于该方法内部的具体执行过程参照下一小节。

③ 如果第二步中不支持 OOM，则要执行 AmS 内部的“潜规则”，即 Android 官方宣称的优先杀死后台 Activity 的具体过程，详细分析见第 4 小节。

④ 最后，无论是使用 OOM 机制还是“潜规则”，杀死后台进程后，如果此时运行的 Activity 数量依然超过 MAX\_ACTIVITIES (20)，则需要继续销毁满足以下三个条件的 Activity。

- Activity 必须已经 stop，但却没有 finishing。
- 必须是不可见的，即该 Activity 窗口上面有其他全屏的窗口，如果不是全屏，则后面的 Activity 是可见的。
- 不能是 persistent 类型，即常驻进程不能被杀死。

trimApplications()方法执行完毕后，如果依然内存不够用，那就无能为力了，只能说明要么是硬件内存太少，要么是应用程序设计不合理，比如 persistent 的应用太多等。

### 3. updateOomAdjLocked()

该方法是在 trimApplication()中被调用的，其作用是告诉 OOM 进程指定应用进程的优先级。介绍该函数内部的逻辑之前需要先理解 ProcessRecord 类中和 OOM 模块相关的几个变量的含义，如表 10-6 所示。

表 10-6 ProcessRecord 类内部和 OOM 相关的变量

变量名称	含 义
curAdj	当前的 adj 值
setAdj	上一个设置值
setRawAdj	curAdj 的含义是实际该 app 的 adj 值，这个值经过了两个过程的评估，这在 computeOomAdjLocked()方法中有详细介绍，而 rawAdj 是在第一个过程评估后认为该 app 应该有的优先级，而实际的优先级还要经过第二个过程的调整
curRawAdj	
setShedGroup	上一个设置值
curShedGroup	当前所在的调度组
hiddenAdj	该变量不是反映当前进程的 hidden 值，而是反映了当前系统中的 hidden 值，即当前系统中如果 adj 值大于 hiddenAdj，则都处于隐藏状态

应用进程的 adj 值是可以调整的，以达到动态调整应用进程优先级的目的；shedGroup 也是可以调整的；hiddenAdj 保存着当前系统中的 hidden 边界。

updateOomAdjLocked() 函数中，循环对 mLruProcesses 列表中的所有进程依次调用 updateOomAdjLocked(..)，注意这是两个同名函数，前者没有参数，后者的参数包含了指定的客户进程。本节主要讨论的是后者，其执行过程主要包含两大步。

① 调用 computeOomAdjLocked()方法，估算指定进程的 oom\_adj 值，估算的流程如下。

(1) 判断该进程是否是 TOP\_APP。TOP\_APP 是一个局部变量，保存了当前正在运行的 ProcessRecord 对象，如果是，那么其优先级自然是最高，为常量 FOREGROUND\_APP\_ADJ，该值是在 Linux 系统启动后在 init.rc 文件中初始化的，其值为 0。以下截取 init.rc 的代码片段：

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
```

(2) 判断该进程中是否包含 instrumentationClass，该值一般在 UnitTest 进程中会存在，而在一般的应用程序中都不包含该对象。

(3) 判断进程中是否包含持久的 Activity 对象，如果有，其优先级也是最高的。

(4) 判断是否包含正在执行的 Receiver 对象，如果有，也是最高优先级。

(5) 判断当前是否有正在执行的 Service，如果有，也是最高优先级。

以上五步中，adj 的赋值都是 FOREGROUND\_APP\_ADJ，即在这五种情况下，指定的客户进程都不能被杀死，其优先级最高，而且平等。

(6) 判断指定的进程是否正在调用前台进程的 service 对象。如果是的话，则其优先级为 VISIBLE\_APP\_ADJ，该值为 1，说明这种情况的进程优先级略低于前台进程，因为它和前台进程正处于互动，所以还是有一定的优先级。

(7) 判断 forcingToForeground 变量是否为空，如果不为空，说明该进程正在被用户强制调到前台。这种情况只是瞬间的，当然如果是这样，其优先级自然比较高，为 VISIBLE\_APP\_ADJ。

(8) 判断是否为 Home 进程，是的话优先级调整为 HOME\_APP\_ADJ，其值为 4。有些读者觉得奇怪，为什么 Home 进程的优先级还低于前面的情况呢？原因是，如果当前正在前台执行 Home 程序，其优先级自然是前面几步的情况，而如果不是前台进程的话，其优先级仅仅比普通进程高一点点而已。

(9) 如果当前进程包含的 Activity 数目大于 0，并且包含了 visible 的 Activity，那么其优先级为 VISIBLE\_APP\_ADJ。如果没有 visible 的 Activity，则意味着该进程可以被隐藏了，所以优先级为 hiddenAdj，即当前系统中的隐藏值。

(10) 对于其他情况，说明仅仅是一个空进程，优先级最低，此时设置 app.empty 变量为 true，并且其 adj 值为 hiddenAdj。

以上十步处理的是进程中包含处于运行状态对象的情况，此时已经获得了一个 adj 值，而这个值正是前面所说的 rawAdj 值的含义，也就是源码注释中所说的 unlimited，即无限制条件下应有的优先级值，如下代码所示：

```

13895         app.adjSeq = mAdjSeq;
13896         app.curRawAdj = adj;

```

```

58     int curRawAdj;           // Current OOM unlimited adjustment for this process
59     int setRawAdj;          // Last set OOM unlimited adjustment for this process

```

除此之外，进程中还包含了一些不处于运行状态的对象，包括 service 和 provider 两种。因此，接下来继续根据这个条件设置 adj 值。

(1) 判断是否为 mBackupTarget 进程。由于“备份进程”要在后台持续收集数据，因此，其优先级应该高于一般的进程，为 BACKUP\_APP\_ADJ(2)，但却要低于 VISIBLE 进程。

(2) 处理当前进程中包含非前台 service 的情况。所谓的非前台 service 是指该 service 不属于前台进程，也没有和前台进程正在交互，其处理过程如下面子流程。

① 首先以下所有处理均在该 app 包含的 service 对象中，并且其优先级不是前台进程 FOREGROUND\_APP\_ADJ 或者所在调度组为“背景非交互”类型，如以下代码所示：

```

13908         if (app.services.size() != 0 && (adj > FOREGROUND_APP_ADJ
13909             || schedGroup == Process.THREAD_GROUP_BG_NONINTERACTIVE)) {

```

② 循环针对每一个 service 的状态调整 adj 值。在每一个 service 对象中，判断该 service 是否被请求启动过，即判断 s.startRequested 是否为 true。如果为 true，说明有某个进程请求启动过该 service，否则该 service 还没有被启动过。如果已经启动，并且还没有超出“待机时间 (MAX\_SERVICE\_INACTIVITY)”30 分钟，那么其优先级会被设置为 SECONDARY\_SERVER\_ADJ，仅次于 VISIBLE。

③ 处理和该 service 有连接的客户端(client)，并根据其客户端的优先级调整该 service 的优先级。首先判断客户端是否是该进程本身，即在同一个进程中调用其内部的 service，这种当然就不用再处理了，保持已有的优先级即可，如以下代码所示：

```

13942         if (cr.binding.client == app) {
13943             // Binding to ourself is not interesting.
13944             continue;
13945         }

```

④ 判断该 service 是否以 BIND\_AUTO\_CREATE 方式启动，如果是则说明该 service 的重要程度取决于客户端本身，否则不做任何处理。处理的逻辑是，首先根据 client 的 hiddenAdj 值调整当前的 hiddenAdj 值，规则是如果当前值大于 client 的值，则把当前值调为 client 值，但不能大于 VISIBLE\_APP\_ADJ。

获得了新的 myHiddenAdj 值后，以此为新的参数，递归调用 computeOomAdjLocked(..., myHiddenAdj, ...)，并计算该 client 的 adj 值。同样，得到返回的 adj 后，如果当前 adj 大于 client 的 adj，则使用 client 的 adj 作为该 app 的 adj。

以上规则的另一个意思就是，如果当前 app 没有 client 重要，则将其重要性调整成和 client 相同重要，否则保持当前的重要性。

⑤ 判断这个客户连接是否有对应的 Activity, 并且该 Activity 是否处于 RESUMED 状态或者正在暂停 (PAUSING) 状态。如果是的话, 则说明这个客户连接很重要, 同样, 该 app 也就很重要了, 优先级重新设置为 FOREGROUND\_APP\_ADJ。

⑥ 估算完 service 后, 如果 adj 值大于 hiddenAdj, 则使用 hiddenAdj 作为 adj 值。这样仅仅是为了避免“滥杀无辜”, 因为如果不调整 adj 为 hiddenAdj, 那么该进程的优先级会很低, 导致会被优先杀死, 然而, 既然该 app 包含了 service, 所以可以暂时让其存在, 以便后面再使用时能够快速启动。

(3) 接下来处理 provider 的情况。其处理逻辑和处理 service 的基本相同, 不同处在于, ContentProviderRecord 对象有一个 externals 对象。这是一个 int 值, 代表了连接该 provider 的非 framework 的进程 id。这是什么意思呢? 即可以使用 C/C++ 写一个 Linux 应用程序, 运行于 Android 底层的 Linux 系统之上, 并且使用 Android 所提供的底层 provider 库来访问系统中已有的 provider 对象, 而这个 externals 就是这个非 framework 进程的 id 号。如果有这种情况存在, 则调整 adj 的优先级为最高, 因为 framework 并不想破坏 native 进程的访问, 毕竟既然能用 native 去访问, 足见该 provider 的重要性。

(4) 以上步骤终于计算出了一个合适的 adj 值, 最后要做的就是查看该 adj 值是否超过了 app 所指定的最大值, 即 maxAdj。如果超过了, 则赋值为 maxAdj, 该变量是在该 app 进程启动时指定的, 其含义是就像在说: “如果系统需要, 可以降低我的重要性, 但是不能超过 maxAdj, 这是我的容忍极限。”

(5) 此时, adj 已经是大家都满意的 adj 值了, 于是把它赋值给 app 的 curAdj 值, 并把 schedGroup 赋值给 app 的 curSchedGroup 变量。

以上仅仅是 updateOomAdjLocked() 的第一大步而已, 接下来看第二步。

② 判断当前 app 是否要从前台切换到后台, 判断的方法是对于 curRawAdj 的 setRawAdj, 前者代表了当前 rawAdj, 后者代表了上一个 rawAdj。这里值得疑惑的是, 为什么要使用 rawAdj 而不是 adj, 因为 adj 才是该 app 最终的优先级值? 尽管在一般情况下, rawAdj 的值和 adj 的相等。如果是的话, 则需要通知客户进程进行主动内存回收, 通知调用 scheduleAppGcLocked(), 该函数将在 5 小节中进行详细介绍。

③ 查看真正的优先级 adj 是否改变, 如果改变, 则调用 Process.setOomAdj() 通知 OOM Killer 进程当前 app 的新优先级。当然如果系统不支持 OOM 管理, 则该方法会返回 false, 相应的, updateOomAppLocked() 则也会返回 false。

④ 查看“所在调度组” curSchedGroup 是否变化, 如果变化, 则调用 Process.setProcGroup 改变当前 app 进程所在的用户组。

至此, 指定进程的 updateOomAppLocked(app,...) 调用就结束了, 正常结束时, 会返回 true, 说明系统是支持 OOM 的。

下面再回到 updateOomAppLocked(无参数) 方法中。处理完一个 app 进程后, 要继续判断该 app 进程的 curAdj 是否大于 HIDDEN\_APP\_MIN\_ADJ (一般为 7), 如果大于则说明该 app 可以隐藏。接着需要查看总的隐藏数量是否超过系统的最大容量值 MAX\_HIDDEN\_APPS (一般为 15), 如果超过, 则立即杀死该进程, 并且将其 ProcessRecord 对象的 killedBackground 变量置为 true, 代表该进程是被后台杀死的, 如下代码所示。

```

14350         if (app.curAdj >= HIDDEN_APP_MIN_ADJ) {
14351             if (!app.killedBackground) {
14352                 numHidden++;
14353                 if (numHidden > MAX_HIDDEN_APPS) {
14354                     Slog.i(TAG, "No longer want " + app
14355                         + " (pid " + app.pid + ")");
14356                     EventLog.writeEvent(EventLogTags.AM
14357                         app.processName, app.setAdj
14358                         app.killedBackground = true;
14359                     Process.killProcessQuiet(app.pid);

```

以上就是 updateOomAdjLocked（无参数）的内部执行过程。

#### 4. AmS 内部内存回收的潜规则

在 Android 的官方文档中关于内存回收大致是这样描述的：系统按照以下优先级关闭进程以释放内存，程序员不需要主动去关心退出进程的事情。

- 前台进程(forground process)，是指那些和用户正在做的事情相关的进程，具体包括：
  - 正在和用户交互的 Activity，即该 Activity 的 onResume()已经执行过。
  - 包含一个 service，该 service 正在服务于和用户交互的 Activity。打个比方，该进程可能不是“领导”，但是却正在和领导谈话，则它依然很重要。
  - 包含一个 service，该 service 正在执行 onCreate()，或者 onStart()，或者 onDestroy()。
  - 包含一个 BroadcastReceiver，正在执行 onReceive()函数。
- 可视进程(visible process)，尽管没有和用户交互，但是却可以影响用户所能看得到的内容。
  - 尽管没有包含和用户交互的 Activity，但是用户却可以看得见该 Activity 的窗口，比如一个 Activity 上面弹出一个对话框的情况。
  - 包含一个 service，该 service 服务于可视的 Activity，即上面说的看得见却不能交互的 Activity。
- 服务进程(service process)，凡是使用 startService()所启动的 service 对象，其所在的进程都称之为服务进程。当然，如果该 service 满足上面两个优先级中的条件，则会上升为相应的优先级。
- 后台进程(background process)，不满足以上任何条件的进程，同时该进程中还包含一些不可见的 Activity，这些进程不影响正在和用户交互的 Activity。
- 空进程(empty process)，进程中不包含任何 component，包括 Activity、service、receiver 对象。之所以还保留这些进程的原因是为了减少重新创建该进程的开销，创建一个空进程的开销包括创建进程本身，以及加载该应用中包含的 resources.arsc 资源文件，这些都是比较耗时的。

通过分析 AmS 源码可知，以上规则仅适用于支持 OOM 模块的系统，当然从目前来看，似乎大多数 Android 系统的底层都包含了该模块。关于支持 OOM 时，AmS 按照以上规则给应用进程设置优先级的细节见第 3 小节，这是在 updateOomAdjLocked()方法中完成的。本节主要是结合源码对以上规则中



的部分规则作一个代码层面上的说明。

- 前台进程。

➤ “正在执行一个 service 的回调”，对应的代码如下：

```
13839         } else if (app.executingServices.size() > 0) {
13840             // An app that is currently executing a service
13841             // counts as being in the foreground.
13842             adj = FOREGROUND_APP_ADJ;
13843             schedGroup = Process.THREAD_GROUP_DEFAULT;
13844             app.adjType = "exec-service";
13845         } else if (app.foregroundServices) {
```

ProcessRecord 中的 executingServices 是一个 ArrayList 类型变量，当 AmS 中执行 startService 或者 stopService 时，对应的 service 都会被临时添加到该列表中，直到执行完毕。

➤ “包含一个正在执行的 Receiver 对象”，对应的代码如下：

```
13832         } else if (app.curReceiver != null ||
13833             (mPendingBroadcast != null && mPendingBroadcast.curApp == app)) {
13834             // An app that is currently receiving a broadcast also
13835             // counts as being in the foreground.
13836             adj = FOREGROUND_APP_ADJ;
13837             schedGroup = Process.THREAD_GROUP_DEFAULT;
13838             app.adjType = "broadcast";
```

ProcessRecord 中的 curReceiver 变量是当执行某个 onReceiver() 方法时被置为 true，执行完毕后重新设置为 false。

- 可视进程。

➤ “尽管不是前台 Activity，但是却是可视”，如以下代码所示：

```
13850         } else if (app.forcingToForeground != null) {
13851             // The user is aware of this app, so make it visible.
13852             adj = VISIBLE_APP_ADJ;
13853             schedGroup = Process.THREAD_GROUP_DEFAULT;
13854             app.adjType = "force-foreground";
13855             app.adjSource = app.forcingToForeground;
```

ProcessRecord 对象中的 forcingToForeground 变量是一个 IBinder 类型，这是导致该 Activity 可视而不可操作的 Binder 对象。读者们可能会觉得奇怪，为什么一个弹出的对话框还会对应一个 Binder 对象呢？这个所说的弹出对话框并不是 Dialog 类，而是一个具有 Dialog 风格的 Activity 对象，而这个 Binder 对象正是对应了该 Activity 在 AmS 中对应的 HistoryRecord 对象。

- “服务于可视 Activity 的 Service 对象”，如以下代码所示：

```
13845         } else if (app.foregroundServices) {
13846             // The user is aware of this app, so make it visible.
13847             adj = VISIBLE_APP_ADJ;
13848             schedGroup = Process.THREAD_GROUP_DEFAULT;
13849             app.adjType = "foreground-service";
```

- “服务进程”，如以下代码所示：

```

13908         if (app.services.size() != 0 && (adj > FOREGROUND_APP_ADJ
13909             || schedGroup == Process.THREAD_GROUP_BG_NONINTERACTIVE)) {
13910             final long now = SystemClock.uptimeMillis();
13911             // This process is more important if the top activity is
13912             // bound to the service.
13913             Iterator jt = app.services.iterator();
13914             while (jt.hasNext() && adj > FOREGROUND_APP_ADJ) {
13915                 ServiceRecord s = (ServiceRecord)jt.next();
13916                 if (s.startRequested) {
13917                     if (now < (s.lastActivity+MAX_SERVICE_INACTIVITY)) {
13918                         // This service has seen some activity within
13919                         // recent memory, so we will keep its process ahead
13920                         // of the background processes.
13921                         if (adj > SECONDARY_SERVER_ADJ) {
13922                             adj = SECONDARY_SERVER_ADJ;
13923                             app.adjType = "started-services";
13924                             app.hidden = false;

```

如果包含服务进程，其优先级设为 SECONDARY\_SERVER\_ADJ，其值为 2，仅次于 VISIBLE 进程。

- 后台进程。后台进程并没有在 updateOomAdjLocked(三个参数)函数中处理，而是在 updateOomAdjLocked(无参数)函数中处理。当对比当前 app 的 curAdj 大于 HIDDEN\_APP\_MIN\_ADJ 时，则认为该 app 处于可隐藏状态，此时如果隐藏的 app 数量大于 MAX\_HIDDEN\_APPS，则立即杀死该 app 进程，如以下代码所示：

```

14350         if (app.curAdj >= HIDDEN_APP_MIN_ADJ) {
14351             if (!app.killedBackground) {
14352                 numHidden++;
14353                 if (numHidden > MAX_HIDDEN_APPS) {
14354                     Slog.i(TAG, "No longer want " + app.processName
14355                         + " (pid " + app.pid + "): hidden #" + numHidden);
14356                     EventLog.writeEvent(EventLogTags.AM_KILL, app.pid,
14357                         app.processName, app.setAdj, "too many background");
14358                     app.killedBackground = true;
14359                     Process.killProcessQuiet(app.pid);

```

- “空进程”在代码中是指 ProcessRecord 中的 thread 变量为空，此时尽管 thread 已经为空，但该客户进程可能还依然存在。在这种情况下，只需要把该进程的 adj 设置为 EMPTY\_APP\_ADJ 即可，如以下代码所示：

```

13790         if (app.thread == null) {
13791             app.adjSeq = mAdjSeq;
13792             app.curSchedGroup = Process.THREAD_GROUP_BG_NONINTERACTIVE;
13793             return (app.curAdj=EMPTY_APP_ADJ);
13794         }

```

EMPTY\_APP\_ADJ 的值为 15，这个优先级是最低的，剩下的杀死该进程的任务则交给 OOM 模块了。当然，如果系统不支持 OOM，那么实际上则没有“空进程”的概念。

以上讨论了 OOM 模式下的“潜规则”，而如果系统不支持 OOM，则 AmS 会采用一种更为简单的方式杀掉进程。



① 杀掉那些非 persistent 类型，不包含任何 Activity，并且当前没有正在执行的 Receiver 对象，内部也不包含 service 对象的进程，如以下代码所示：

```

14446         final ProcessRecord app = mLRUProcesses.get(i);
14447         // Quit an application only if it is not currently
14448         // running any activities.
14449         if (!app.persistent && app.activities.size() == 0
14450             && app.curReceiver == null && app.services.size() == 0) {
14451             Slog.i(
14452                 TAG, "Exiting empty application process "
14453                     + app.processName + " ("
14454                     + (app.thread != null ? app.thread.asBinder() : null)
14455                     + ")\n");
14456             if (app.pid > 0 && app.pid != MY_PID) {
14457                 Process.killProcess(app.pid);

```

② 对于那些包含了 Activity 的进程，如果该 Activity 已经处于 stop 状态，并且是不可见的，同时已经在 HistoryRecord 对象中保存了该 Activity 的 state 数据，那么则可以停止该 Activity 所在的进程，相应的 state 数据保存在了 HistoryRecord 的 icicle 变量中。这是一个 Bundle 类型，从而使得当下一次重新启动该 Activity 时把该数据传过去。代码参照 AmS 源码第 14486 行，以下为截取片段。

```

14486         boolean canQuit = !app.persistent && app.curReceiver == null
14487             && app.services.size() == 0
14488             && app.persistentActivities == 0;

```

以上两步仅在系统不支持 OOM 模块的情况下使用。无论系统是否支持 OOM，执行完以上操作后，会进行最后一步内存回收工作。其逻辑是判断 mLRUActivities 列表中保存的已经运行 Activity 的数量是否超过了系统允许的最大值 MAX\_ACTIVITIES (20)，如果超过了，则要强行关闭一些 Activity。这里的强行关闭也是有条件的，因此，可能出现死循环，即永远达不到这个条件，如以下代码所示：

```

14542         for ( i=0;
14543             i<mLRUActivities.size()
14544             && mLRUActivities.size() > curMaxActivities;
14545             i++) {
14546             final HistoryRecord r
14547                 = (HistoryRecord)mLRUActivities.get(i);
14548
14549             // We can finish this one if we have its icicle saved and
14550             // it is not persistent.
14551             if ((r.haveState || !r.stateNotNeeded) && !r.visible
14552                 && r.stopped && !r.persistent && !r.finishing) {
14553                 final int origSize = mLRUActivities.size();
14554                 destroyActivityLocked(r, true);
14555
14556                 // This will remove it from the LRU list, so keep
14557                 // our index at the same value. Note that this check to
14558                 // see if the size changes is just paranoia -- if
14559                 // something unexpected happens, we don't want to end up
14560                 // in an infinite loop.
14561                 if (origSize > mLRUActivities.size()) {
14562                     i--;
14563                 }
14564             }
14565         }

```

该代码中要求,被停止的 Activity 必须属于 persistent 类型。因此,假设系统中有大于 20 个 persistent 类型的 Activity,而且它们的启动顺序是 1 启动 2,2 启动 3,依次类推,19 启动 20,20 启动后面的 21、22 等,在这种情况下,以上代码将会出现死循环而不能退出。当然如果没有这么多的 persistent 类型的 Activity,这里会 destroy 掉相应的 Activity 对象。

### 5. 客户进程内存回收

在 AmS 中调用 scheduleAppGcLocked(ProcessRecord app)方法时,就会通知指定的 app 对象进行内存回收,以下对这个过程进行描述。

该函数首先要检测指定 app 上一次进行 gc 的时间,并和当前时间对比。如果时间间隔还没有超过最小间隔,则仅仅把指定的 app 放入 mProcessesToGc 列表中。否则,调用同名函数 scheduleAppGcLocked(无参数)从 mProcessesToGc 列表中取出下一个 app,并发送一个 1 分钟延迟的 GC\_BACKGROUND\_PROCESSES\_MSG 消息。之所以选择 1 分钟延迟,是为了减少客户进程进行 gc 的频率,为什么是 1 分钟而不是 2 分钟或者 30 秒呢?这或许是一个经验值。

处理 GC\_BACKGROUND\_PROCESSES\_MSG 消息的函数是 AmS 中的 performAppGcsIfAppropriate()函数,其内部首先会判断当前是否可以“适合(Appropriate)”进行内存回收,如果可以则调用 performAppGcsLocked(无参数),否则,再次调用 scheduleAppGcLocked(无参数)发送一个异步消息。

而是否“适合”则是通过调用 canGcNowLocked()方法进行判断的,其代码如下:

```

14103 private final boolean canGcNowLocked() {
14104     return mParallelBroadcasts.size() == 0
14105         && mOrderedBroadcasts.size() == 0
14106         && (mSleeping || (mResumedActivity != null &&
14107             mResumedActivity.idle));
14108 }

```

以上代码包含以下条件。

- 当前没有 Broadcast 在执行,包括 mParallelBradcasts 和 mOrderedBroadcasts。
- 当前处于 mSleeping 状态或者有正在执行的 Activity 但却处于 idle 状态。

以上条件的语义是系统当前必须处于空闲状态,有读者可能问了:“为什么条件中有 Broadcast 但却没有 Service 或者 Receiver 对象呢?”原因是,无论是 Service 还是 Receiver,都是在相应的服务进程中执行代码的,与 AmS 无关,而 Broadcast 则需要 AmS 去派发的,而当派发完毕后,mParalleBroadcasts 或者 mOrderedBroadcasts 列表的内容会被清除。

在 performAppGcLocked(无参数)函数内部,使用 while 循环,从 mProcessesToGc 列表中逐个取出每个需要进行 gc 的 ProcessRecord 对象,并针对该对象调用 performAppGcLocked(app)方法,如以下代码所示:

```

14120         while (mProcessesToGc.size() > 0) {
14121             ProcessRecord proc = mProcessesToGc.remove(0);
14122             if (proc.curRowAdj > VISIBLE_APP_ADJ || proc.reportLowMemory) {
14123                 if ((proc.lastRequestedGc+GC_MIN_INTERVAL)
14124                     <= SystemClock.uptimeMillis()) {
14125                     // To avoid spamming the system, we will GC processes on
14126                     // at a time, waiting a few seconds between each.
14127                     performAppGcLocked(proc);
14128                     scheduleAppGcsLocked();
14129                     return;

```

这里需要注意该 while 循环的内部处理。调用完 performAppGcLocked(proc)方法后，再调用 scheduleAppGcsLocked()，然后使用 return 返回。也就是说，每次仅针对一个 app 进行内存回收，可能 mProcessesToGc 列表中有多个 app 都满足回收内存的条件，但是也只能等到 1 分钟以后才能再次被调度到。

performAppGcLocked(proc)方法的内部流程如下，它会判断当前 app 的 reportLowMemory 标识是否为 true。如果是则调用 app.thread.scheduleLowMemory()函数处理因为内存低而导致的内存释放，否则调用 app.thread.processInBackground()函数处理背景模式内存释放。是在 AmS 通知客户进程启动指定的 Activity 时，如果客户进程报告启动失败，AmS 就会认为客户进程可能出现内存问题，从而将其 reportLowMemory 标识置为 true。

下面分别介绍客户进程中的 scheduleLowMemory()函数和 processInBackground()函数到底都做了哪些与内存回收相关的工作。首先来看前者，客户进程中凡是以 schedule 开始的函数，其内部都是发送一个异步消息并立即返回，即所谓的异步处理，该消息对应的处理函数为 handleLowMemory()，其内部流程如附图 4 所示，此处截取主要片段如图 10-23 所示

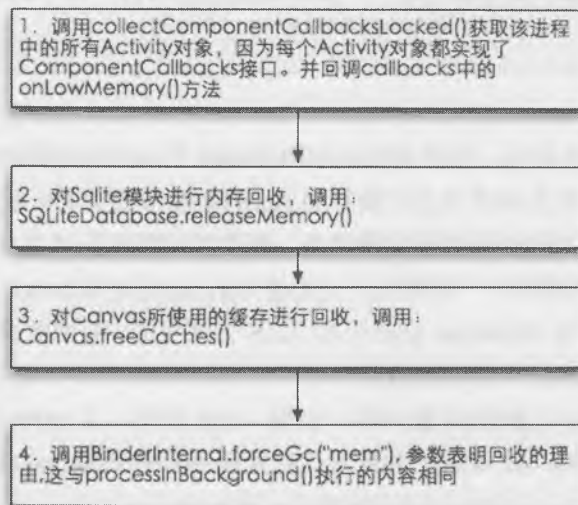


图 10-23 ActivityThread 中 handleLowMemory()内部流程

① 回调该进程中所包含的所有 Component 对象中的 onLowMemory()方法。Component 包括四种，分别为 Activity、Service、Provider、Application，这四个对象分别在 ActivityThread 中的 mActivities、mServices、mProviderMap、mAllApplications 容器中保存。

② 调用 SQLiteDatabase 的静态方法 releaseMemory()释放该进程中 SQLite 模块所占用的内存。尽管该方法是一个 native 方法，但从这种调用方式上可以猜出，SQLite 的内部肯定为每一个调用者进程分配了一块独立的内存空间缓存查询结果，而 releaseMemory()方法则会判断调用者进程 id，然后根据此值释放相关的内存。

③ 调用 Canvas 的静态方法 freeCaches()释放该应用中所包含的 Canvas 对象。注意这里 Caches 用了复数而不是 Cache，再次提示大家，每个应用中会包含多个 Canvas 对象。

④ 调用 BinderInternal.forceGc(“mem”)方法释放该进程中的 Binder 对象。

以上四步中三步都和 native 的方法相关，而且都是静态方法，这也再次说明，native 所申请的系统内存无法从 Java 环境中清除，这就是为什么 native 方法常常引起应用程序内存泄露的原因，尤其是按照非对称方式调用 native 方法时。所谓的“非对称”是指，native 一般会提供一对函数，一个包括了申请内存的操作，另一个包括了释放该内存的操作，而 Java 程序仅仅调用了前者而忘记调用后者，那么即使强制关闭该 Java 程序，也依然会出现系统内存泄露的情况。

上面介绍了 scheduleLowMemory()的处理过程，下面继续介绍 processInBackground()方法，该方法的处理过程十分简单。当然，首先依然是发送一个异步消息，而该消息会执行到 Idler 对象中的 doGcIfNeeded()方法，而该方法仅仅是调用了 BinderInternal.forceGc(“bg”)函数而已，即与上面的第四步的执行内容相同。

为什么 processInBackground()仅仅释放 Binder 相关的内存呢？原因是此时系统未出现内存低的情况，因此，只需要释放影响 CPU 运行的部分即可。在第 10.1 节中的关于系统效率中曾说过，应用程序占用 CPU 的部分由三个部分组成，其中一项就是 Binder 对象。

### 10.3 对 AmS 中数据对象的理解

AmS 中定义了各种变量，用于表示各种数据对象，这些对象包括“进程”、“任务”、“运行片段(Component)”等，这些变量如表 10-7 所示。

表 10-7 AmS 中定义的数据变量

变量名称	含 义
MAX_ACTIVITY	允许最大的后台 Activity 的数目，为 20
mHistory	所有处于非 finising 状态的 HistoryRecord 列表，按照启动顺序排列
MAX_RECENT_TASK	mRecentTask 列表中允许存储的最大值为 20
mRecentTask	最近启动过的 Task 列表，按照最后活动状态时刻排列，每个 TaskRecord 内部都包含 lastActiveTime，表示该 Task 最后处于活动状态的时刻

(续表)

变量名称	含 义
MAX_HIDDEN_APP	允许处于 hidden 状态的进程数目的最大值, 超过这个值后, AmS 会杀掉优先级低的进程, 为 15
mLRUActivites	最近启动过的 HistoryRecord 列表。与 mHistory 的区别在于, mLRUActivites 仅仅记录了不同 Activity 的最后启动时刻, 而 mHistory 内部包含了 Task 的信息, 并且可能包含同一个 Activity 的多个实例。LRU 的含义是 Lastest Recent Used
mLruProcess	最近启动过的进程列表, 不包含 persistent 类型的进程
mProcessNames	所有运行的进程列表, 按照名称进行区分
mPidSelfLocked	与 mProcessNames 保存的进程对象相同, 但是按照 pid 进行区分
mServices	系统中已经启动的 Service 列表, 按照 Service 的名称区分
mServicesByIntent	同 mService, 但是按照启动该 Service 的 Intent 区分
mServiceConnections	与当前所有 Service 建立了连接关系的客户端列表
mProviderByName	系统中已经启动的 Provider 列表, 按照名称区分
mProviderByClass	同 mProviderByName, 按照 Provider 的实现类名称区分

### 10.3.1 常见的对象操作

在表 10-7 的基础上, 可以实现以下几种常见的操作。

- 如何获取系统中运行的类型为 persistent 的进程列表?

可以遍历 mProcessNames 列表, 并且判断每一个 ProcessRecrod 中的 persistent 变量是否为 true。

- 如何获取最近的 Task 列表?

直接遍历 mRecentTask 列表即可, 长按“Home”键时出现的最近六个任务列表正是从该列表中获取的。

- 当前 Task 中都包含哪些 Activity 对象?

首先通过 mCurTask 变量获得当前 Task 的 id 值, 然后遍历 mHistory 中的所有 HistoryRecord 对象, 判断其 taskId 变量是否等于 mCurTask 的 id 值。

- 系统中已经运行了哪些进程?

直接遍历 mProcessNames 或者 mPidSelfLocked 列表即可。

- 一个进程中都包含有哪些正在运行的 Activity 对象?

遍历 mHistory 列表, 查找每一个 HistoryRecord 的 app 变量 (这是一个 ProcessRecord 对象), 然后再判断该 app 对象的 pid 值是否为指定的 pid 值。

- 如何查看与系统中的 Service 对象有连接的客户端?

直接遍历 mServiceConnections 列表即可。如果要判断指定 Service 都包含哪些客户端, 则可以先从 mServices 中按照 Service 的名称找到目标 ServiceRecord 对象, 然后遍历 mServiceConnections 列表, 获

得 ServiceConnection 对象,再获得该对象内部的 AppBindRecord 对象,最后判断其包含的 ServiceRecord 对象是否是指定的 ServiceRecord 对象即可。

- 查看指定进程中都有哪些 Service 或者 Provider 对象在运行?

思路是遍历 mService 和 mProviderByName,然后分别从其对应的数据对象判断是否属于指定的进程 id。

### 10.3.2 理解 Activity

Activity 本身只是一段程序代码而已,它所执行的内容没有任何的系统调用,客户端程序总是从 ActivityThread 开始执行。这个类已经创建了客户进程,然后从 Activity 所对应的 Class 文件中装载进来程序代码,实际上就只是一个 Activity 类对象,然后继续执行该对象内部的各种代码,这些执行的代码默认不会再创建线程。

有的读者会问一个不是问题的问题:“Android 只允许一个 Activity 在运行,有没有办法让多个 Activity 同时运行,就像 Windows 操作系统中的同时开一个计算器程序,再开一个浏览器程序一样呢?”之所以说这个问题不是问题,原因在于,这个问题成立的条件如下:

- 每个 Activity 都应该对应一个应用程序。
- 每个 Activity 都应该包含一个交互窗口。
- 系统必须为每一个 Activity 的窗口分配显存,并且只有在该应用程序退出时才释放这段显存。

下面对这三个条件逐一进行解析。

- Activity 并不对应一个应用程序,ActivityThread 才对应一个应用程序,所以 Android 允许同时运行多个应用程序,实际上是允许多个 ActivityThread 同时运行。
- 默认的 Activity 实现中的确会添加一个窗口,但实际上可以修改 Activity 的默认实现,使其不添加任何窗口,而且 AmS 对这种修改不会在意,依然会正常运行。换句话说,AmS 会按照定义好的调度顺序来启动、关闭 Activity,至于 Activity 内部是添加窗口或不添加窗口也好,无所谓。
- 在 Android 中,尽管默认的 Activity 会添加一个窗口,但是当系统内存低的时候,后台 Activity 对应的显存会被释放掉,而且这是在用户没有感知的条件下进行的。

因此,以上问题可以这么来问:“Android 是多窗口系统吗?”即在一个屏幕上可以存在不同 Activity 对应的窗口,而且这些窗口可以层叠、移动。这个问题留在 WmS 源码分析一章中解答,不过结论是:Android 窗口系统是一种简单的单窗口系统。

### 10.3.3 Android 多进程吗?是同时在运行多个应用程序吗

Android 的确是多进程的,因为同时有多个应用程序进程存在。但不幸的是,由于 WmS 子系统的

设计特点,导致非前台应用程序没有机会获得消息,从而没有机会占用 CPU。并且 Android 的底层 Linux 未采用磁盘虚拟内存机制,程序只能使用物理内存作为最大内存,所以 AmS 中采用了自动杀死优先级较低进程的方法以达到释放内存的目的,而这会导致用户在多个(超过 20 个)应用程序中快速切换时就会出现速度变慢的现象。

这并不能说就是 Android 系统的缺点,因为毕竟 Android 是面向移动设备的,对于手机用户来讲,来回在超过 20 个程序间切换似乎是不常发生的事情。

另外,从技术角度来讲,AmS 的 WmS 调度机制也能用来定义“上网本”的概念。在过去的 2 年里,“上网本”被认为是消费电子行业的一个新产品,很多电脑公司都推出了所谓的上网本产品,这里先不说硬件的配置,而来说所使用的操作系统,在 iPad 诞生之前,流行的上网本基本上都是采用 Windows 操作系统。人们对上网本的认识似乎就是“价格低一点,速度慢一点,待机时间长一点”,而从功能的角度来讲,则完全等同于 PC。

如果说 iPad 也是上网本,则操作系统就应该被重新定义了,这里主要就是 AmS 和 WmS 的调度机制的差别。

首先,Windows 系统是完整的多进程、多窗口操作系统,而 iPad 或者 Android 只能算作多进程、单窗口系统。同时,为了配合单窗口设计,AmS 在应用程序调度上也做了一定的调整,使得虽然 Linux 底层是多进程,而 Framework 层却变成了单进程。

单窗口与多窗口系统设计的主要区别在于以下几点。

- 系统必须为每一个应用程序分配窗口,并且这些窗口应该常驻,并且只有在程序退出时才释放。
- 用户的按键等消息要同时被发送到所有的应用窗口,而不仅仅是所谓的“当前交互”窗口。
- 窗口系统应该能处理多个应用窗口之间的层叠、位移等变化,而不能把这个任务交给一个应用窗口自己去处理。
- 窗口系统在将多个窗口绘制到屏幕上时,也就是 Android 中的 SurfaceFlinger 模块,要同时把所有的窗口中进行运算,而不仅仅是把当前的窗口绘制到屏幕。

当然,要实现以上四点,对 CPU 来讲也有一定难度,尤其是当所有的窗口都不处于全屏的时候。这就是为什么在 Windows 操作系统中的窗口有一个“最大化”的按钮的原因,因为在“最大化”模式下,用户按键消息的派发及屏幕的绘制都能够简化一些,从而有助于系统的运行效率。当然这只是技术上的考虑,从用户的角度来讲,或许最大化也是他们所想要的功能,而在苹果电脑系统中,则没有最大化的按钮。

以上四点中,Android 仅仅满足第四点,所以说它不是真正的多窗口系统。而 iPad 或者 Android PAD 只是 30%的游戏 + 30%的阅读 + 30%的工具 + 10%的生活乐趣。

### 10.4 ActivityGroup 的内部机制

分析完 AmS 内部机制后,读者们已经明白,系统一次只允许一个 Activity 处于运行状态,要让另外一个 Activity 处于运行状态必须先暂停当前正在运行的 Activity。而在应用程序设计时,Framework



提供了一个 `ActivityGroup` 类，它允许把一个 `Activity` 嵌入到另一个 `Activity` 中运行，常见的基于该类的实现包括 `TabActivity`，这到底是怎么回事呢？难道 `ActivityGroup` 类具备同时使多个 `Activity` 处于运行状态的能力吗？

本节就以 `TabActivity` 类的实现为例，说明 `ActivityGroup` 的内部机制。

### 10.4.1 `TabActivity` 使用时的类关系结构

在使用 `TabActivity` 设计应用程序时，其内部相关的类关系如图 10-24 所示。

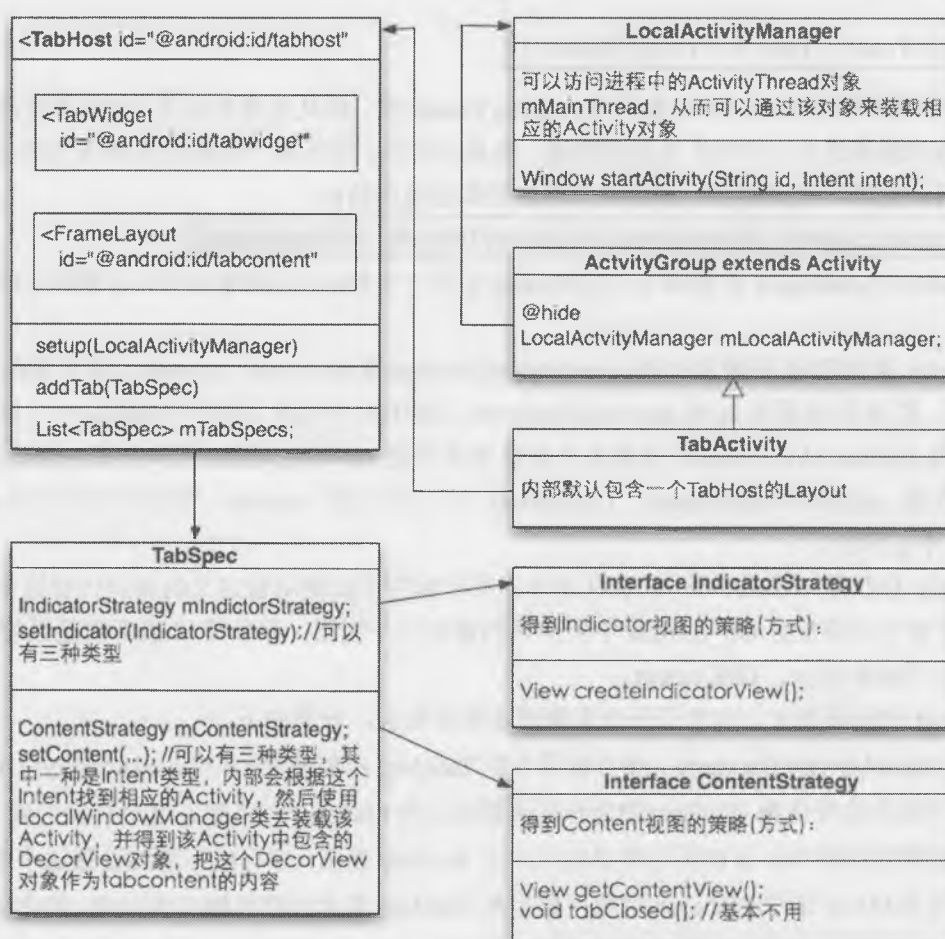


图 10-24 `TabActivity` 相关类的关系

首先 `TabActivity` 的祖先类也是 `Activity` 类，因此，从 AmS 的角度来看，`TabActivity` 与普通的 `Activity` 没有什么区别，其生命期包括标准的 `start`、`stop`、`resume`、`destroy` 等，而且系统中只允许同时运行一个

TabActivity。

所不同的是, TabActivity 基于 ActivityGroup 类, 该类的父类是 Activity 类, 但 ActivityGroup 内部有一个重要成员变量, 其类型为 LocalActivityManager, 该类的最大特点在于它可以访问应用进程的主类, 即 ActivityThread 类。前面分析过, AmS 要启动某个 Activity 或者暂停某个 Activity 都是通过 ActivityThread 类执行的, 而 LocalActivityManager 能够访问 ActivityThread 类就意味着可以通过它来装载不同的 Activity, 并且控制 Activity 的不同的状态。

为什么 LocalActivityManager 可以访问主类 ActivityThread, 而普通的应用程序无法访问主类呢? 因为在编译输出 Android SDK 时, ActivityThread 类是被隐藏的, 如以下代码所示:

```
115 * {@hide}
116 */
117 public final class ActivityThread {
```

所以, 应用程序无法通过 SDK 来引用 ActivityThread 类。那又有读者问了, 如果不使用 SDK 编译, 而直接使用源码编译是否可用呢? 真的很抱歉, 也是不可以, 因为客户进程的主类变量的访问权限被设置为包内访问, 如以下代码所示, 未加权限限定符即指包内访问。

```
157 static final ThreadLocal<ActivityThread> sThreadLocal
```

而 LocalActivityManager 正是和 ActivityThread 在同一个包内, 这就是为什么它能够访问主类对象的原因。

TabActivity 类的默认布局文件是 com.android.internal.R.layout.tab\_content, 该布局的内容是一个 TabHost 视图, 其 id 值为系统 id 值 android:id/tabhost。TabHost 中包含了两个子视图, 一个是 TabWidget, 其 id 值必须是 android:id/tabwidget, 这就是大家经常看到的 Tab 页的效果; 另一个是 FrameLayout 视图, 其 id 值必须是 android:id/tabcontent, TabActivity 所包含内置 Activity 对应的窗口就是被添加到该 FrameLayout 中显示的。

tab\_content 仅仅是 TabActivity 的默认布局, 应用程序可以使用自定义的布局代替这个布局, 但自定义的布局文件中必须包含 tab\_content 中所声明的最少三个视图, 并且其 id 值必须是系统的 id 值, 分别为 TabHost、TabWidget、TabContent。

在 TabHost 的视图类中, 包含了三个重要的函数或变量, 分别如下。

- **setup(LocalActivityManager):** 该方法用于给 TabHost 内部设置一个 LocalActivityManager 对象。读者可能会觉得奇怪, TabHost 中为什么还需要这个 Manager 对象呢? 这个对象的确不是必需的, 假如你所实现的 Tab 页视图不需要嵌入一个 Activity 视图, 那么就不需要设置这个对象, 否则必须告诉 TabHost 这个 Manager 对象, 要不然 TabHost 是无法获得指定 Activity 的内部视图的。
- **addTab(TabSpec):** 该方法用于向 TabHost 中的 TabWidget 视图添加一个 Tab 页。表面上看, 既然是添加一个 Tab 页, 那么 TabSpec 类应该是一个视图类才对, 而事实上 TabSpec 是一个功能类, 其本身并不是一个视图, 但是它却可以提供产生视图的方法。

- `List<TabSpec> mTabSpecs`: 这是 `TabHost` 中所包含的全部 `Tab` 页列表, 每一项都是一个 `TabSpec` 对象。

下面, 再来介绍 `TabSpec` 是如何产生 `Tab` 页视图的。

`TabSpec` 类中有两个重要变量, 其类型分别是 `IndicatorStrategy` 和 `ContentStrategy`。这两个子类的名称有点意思, `Indicator` 的意思是“指示”, 它指示用户该 `Tab` 页的意义, `Content` 即为单击某个 `Tab` 页后对应的“内容”, 而 `Strategy` 是策略的意思, 因此这两个子类的作用分别是“产生 `Tab` 页的策略”和“产生内容的策略”。同时, 这两个子类仅仅是一个 `interface`, 为什么只是一个接口呢? 因为既然是策略就意味着可以有不同的策略, 在 `TabHost` 中也的确实现了不同的策略, 其中 `Tab` 页策略包括以下几项。

- `LableAndIconIndicatorStrategy`: 可以产生一个带图标和标签文字的标签页。
- `LableIndicatorStrategy`: 仅仅有标签文字的标签页。
- `ViewIndicatorStrategy`: 可以使用自定义 `View` 作为标签页视图。

内容策略包括以下。

- `FactoryContentStrategy`: 使用回调方式允许应用程序动态创建内容视图。
- `IntentContentStrategy`: 嵌入 `Activity` 视图作为内容视图, 这正是 `TabActivity` 所使用的。
- `ViewIdContentStrategy`: 使用一个静态的布局作为内容视图。

看了这么多的策略后, 大家可能会觉得奇怪: “新建一个 `TabSpec` 后, 如何指定其标签和内容使用何种策略呢?” 事实上, 当新建一个 `TabSpec` 对象后, `TabSpec` 提供了三个设置标签策略的函数 `setIndicator(xxx)`, 其中不同参数对应不同的标签策略; 同时也提供了三个设置内容策略的函数 `setContent(xxx)`, 不同的参数对应不同的内容策略。其中 `setContent(Intent intent)` 函数正是使用嵌入 `Activity` 方式, 参数 `Intent` 用来匹配相应的 `Activity`。

#### 10.4.2 LocalActivityManager 的内部机制

`LocalActivityManager` 内部机制的核心在于, 它使用了主线程对象 `mActivityThread` 来装载指定的 `Activity`。注意, 这里是装载, 而不是启动, 这点很重要。

所谓的启动, 一般是指会创建一个进程(如果所在应用进程还不存在)运行该 `Activity`, 而装载仅仅是指把该 `Activity` 作为一个普通类进行加载, 并创建一个该类的对象而已, 而该类的任何函数都没有被运行。

`LocalActivityManager` 提供了一个重要方法 `startActivity()`, 该方法正是利用主线程 `mActivityThread` 去装载指定的 `Activity`, 其执行过程如图 10-25 所示。

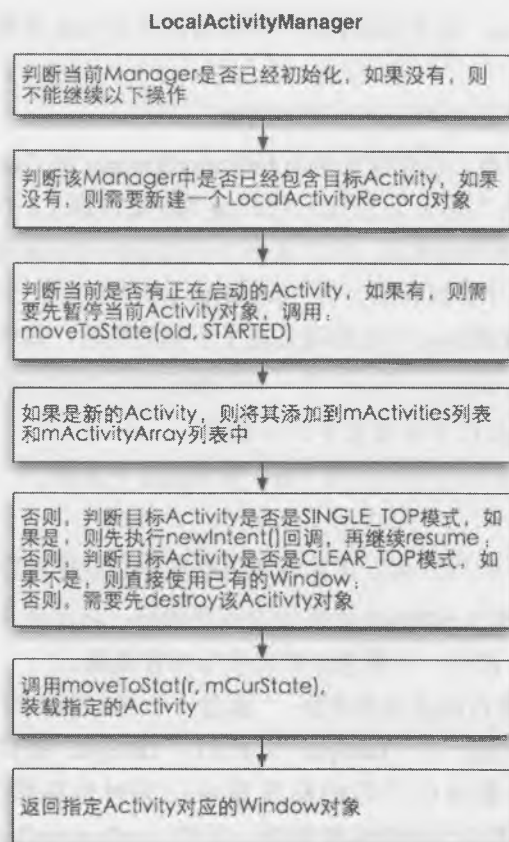


图 10-25 LocalActivityManager 类中 startActivity() 的执行流程

① Manager 对象必须已经被初始化, 初始化的工作是在 `dispatchCreate()` 方法中首先被完成的, 而这又是在 `ActivityGroup` 类中的 `onCreate()` 中被调用的。也就是说, `LocalActivityManager` 的 `startActivity()` 方法必须在所在的 `Activity` 的 `onCreate()` 方法执行完毕后被调用。

② 判断目标 `Activity` 是否包含在该 `Manager` 中。`Manager` 中使用两个列表变量保存已经装载过的 `Activity` 对象, 分别是 `mActivities` 和 `mActivityArray`。前者是一个 `HashMap` 类型, 每一个 `LocalActivityRecord` 按照一个字符串对应; 后者是一个 `ArrayList` 列表。

③ 判断装载的 `Activity` 对象是否有正处于 `resume` 状态的, 如果有, 则要先暂停, 事实上可以完全不用暂停, 暂停仅仅是 `Manager` 希望完全按照 `Activity` 对象本身的执行顺序调用它, 从而使得看上去更像是一个标准的子 `Activity` 启动方式。而暂停则是通过调用 `moveToState()` 完成的。

④ 如果目标 `Activity` 已经被装载到了当前 `Manager` 中, 下面就需要判断是直接使用该 `Activity` 的当前窗口呢, 还是需要先销毁该 `Activity`, 并重新调用其 `onCreate()`? 注意, 这里所说的销毁仅仅是指把 `Activity` 变成“销毁”的状态而已, 并不是说销毁该 `Activity` 对象。而判断的规则有点类似于 `AMS` 中根据 `Activity` 的 `flag` 执行不同的操作, 其中包括是否先调用目标 `Activity` 的 `onNewIntent()`, 还包括是

否是 CLEAR\_TOP 模式。一般作为 TabActivity 的嵌入式 Activity 都不会是 CLEAR\_TOP 模式，否则，如果多个 Tab 页使用同一个 Activity 对象将导致所显示的内容完全相同。

5 调用 moveToState() 改变指定 Activity 到 resume 状态。

6 返回 Activity 所对应的 Window 窗口。

从以上步骤可以看出，装载 Activity 对象的过程对 AmS 来讲是完全不可见的，因为这是装载而不是启动，因此看似 TabActivity 同时运行了多个 Activity，而实际上仅仅是运行了 ActivityGroup 一个 Activity。那些嵌入的 Activity 仅仅是贡献了自己所包含的 Window 窗口而已，TabActivity 正是把这些 Window 窗口的 DecorView 作为 tabcontent 的子视图而已。

下面对 moveToState(LocalActivityRecord r, int desireState) 函数的过程进行说明，参数 r 代表目标 Activity 对象，desireState 代表期望把目标 Activity 改变成哪种状态。

moveToState() 函数内部首先判断 r.curState 是否是 RESTORED 或者 DESTROY 状态，如果是则直接返回。因为 RESTORED 代表刚刚创建了目标 Activity 对象，还没有执行 onCreate() 方法，所以不能改变状态；DESTROY 代表已经销毁，也不能改变状态。

接着，判断 r.curState 是否是 INITIALIZE 状态，这种情况只有在第一次调用 startActivity() 装载目标 Activity 对象时才会执行到，其内部主要包括调用 startActivityNow() 和 performResumeActivity() 将目标 Activity 改变到 STARTED 或者 RESUMED 状态。

由于 Activity 当前状态不同，要想达到不同的期望状态自然需要经过不同的步骤。moveToState() 函数内部正是使用 switch 语句先判断当前处于什么状态，然后再在 case 里面使用 if...else 语句判断期望的状态，最后再调用不同的函数。其状态和调用关系如表 10-8 所示，该表中调用的函数名称使用了简写，比如 performRestartActivity 简写为 Restart。

表 10-8 Activity 在不同状态中转换时需执行的操作

目标状态 当前状态	CREATED	STARTED	RESUMED
CREATED		Restart();	Restart(); Resume();
STARTED	Stop();		Resume();
RESUMED	Pause(); Stop();	Pause();	

从以上的步骤可以看出，startActivity() 的内部执行逻辑有点像 AmS 中根据当前 Activity 状态调用不同方法。这两者就像《西游记》中的小雷音寺和大雷音寺，两者的本质区别在于 LocalActivityManager 仅仅是为了获取 Activity 对应的 Window 对象，中间的状态切换仅仅是为了保证 Activity 本身的执行过程，从而保证 Window 对象的视图内容有一个正确的呈现。

### 10.4.3 ActivityGroup 内部的 Activity 生命期控制

前面分析了 LocalActivityManager 内部的执行原理, 接下来分析一个有意思的问题: “在 TabActivity 的多个 Tab 页切换时, 内嵌的 Activity 对象会在 onPause() 和 onResume() 之间切换吗?”

从上面的分析可知, Manager 装载 Activity 的目的仅仅是为了获取其所包含的 Window 对象, 而一旦获取后, 则似乎不需要再纠缠于 Activity 本身的生命期状态变换操作。其实笔者也是这么认为的, 可以尝试屏蔽以下代码, 该段代码的作用是在装载下一个 Activity 之前先暂停当前的 Activity 对象。

```
287 //         if (old != null && old != r && mCurState == RESUMED) {
288 //             moveToState(old, STARTED);
289 //         }
```

屏蔽后, 运行结果丝毫不受影响, 原因很简单, 这里做暂停的目的仅仅为了保持 Activity 原有的生命期过程, 从而可以保持原有 Activity 释放相关资源的行为。比如当 Activity A 以启动的方式运行时, 如果另一个 Activity B 要启动, 则会先暂停 A。在一般的程序设计中, 暂停会回调 onPause() 操作, 如果该 Activity 使用了大量的内存或者其他资源, 在 onPause() 函数中程序员可能会尝试释放这些资源以提高系统效率, 这就是为什么在 LocalActivityManager 中也保持了这种流程的原因。当然, 如果你不释放, 也不会发生什么逻辑错误。

而在以上代码的 moveToState() 操作中调用了 mActivityThread 的 onPause() 或者 onStop() 操作, 这就是为什么在 Tab 页切换时, 对应的 Activity 也会执行 onPause() 或者 onStop() 的原因。这完全与 AmS 无关, 因此不要因为这个而产生 TabActivity 同时运行了两个 Activity (TabActivity 本身和嵌入的 Activity) 的错觉。

另外还有一个有意思的问题, 请思考下面的操作过程。

打开一个 TabActivity, 比如“联系人”程序, 在上面的四个 Tab 页上都点一次, 然后再按“Home”键回到桌面, 然后再从联系人图标中进入该程序。请思考此时 TabActivity 内嵌的 Activity 会发生生命期状态改变吗?

首先 TabActivity 当然会从 stop 状态转变为 start 状态, 并先后调用 onStart() 和 onPause()。因为它是一个标准的 Activity, TabActivity 的父类是 ActivityGroup, 而在该类中, 相应的 onXXX() 方法内部都增加了 mLocalActivityManager.dispatchXXX() 代码, 比如:

```
555 @Override
556 protected void onResume() {
557     super.onResume();
558     mLocalActivityManager.dispatchResume();
559 }
```

而在 LocalActivityManager 中, dispatchXXX() 则会把相应的 action 再 dispatch 到所包含的所有嵌入式 Activity 对象中。所以, 以上问题的答案是内嵌的 Activity 生命期会从 stop 状态转换到 resume 状态。仔细查看 ActivityGroup 的 onXXX() 函数发现, 唯独没有 onStart() 方法, 其原因是在 LocalActivityManager 的 moveToState() 方法中可以直接把子 Activity 的状态从 stop 改变到 resume, 所以, 此处可以省略对 onStart() 的重载。



## 第 11 章 从输入设备中获取消息

从输入设备中获取消息是所有 GUI 系统的核心问题之一，该问题主要包含三个方面。

- 获取原始的用户消息，包括按键、触摸屏、鼠标、轨迹球等各种输入设备的消息；
- 对原始消息进行一定的加工，使之转换为程序可以理解的消息。比如所有的按键消息都包括“按下、弹起”等原始消息，而对程序来讲，可能只关心该按键被“按了一次”或者“长按”，因此，需要把原始的消息转换为程序可以理解的消息；
- 把转换后的消息发送到相应的用户窗口所在进程。如果消息获取线程和用户线程在同一个进程空间中，则传递消息比较简单，但对于多进程系统来讲，消息获取线程和用户线程往往在不同的进程空间中，因此需要使用 IPC 机制把消息传递到用户窗口所在的线程中。



### Android 消息获取过程概述

在 Android 2.2 之前（包括 2.2）的所有版本中，获取用户消息的方式基本上是相同的，其过程可参照附图 5，具体如下。

在 WmS 中有一个子类 KeyQ，该类基于 KeyInputQueue 类，而该类内部则包含一个线程对象，即 KeyQ 对象是一个线程对象。该线程的任务是调用 native 方法从输入设备中读取用户消息，包括按键、触摸屏、鼠标、轨迹球等各种消息，并把读取到的消息保存到一个 QueueEvent 队列中。

在 WmS 中有另外一个子类叫做 InputDispatcherThread，该类也是一个线程类，即内部包含一个线程。该线程的任务是从上面的 QueueEvent 队列中读取用户消息，并对这些消息进行一定的加工，然后判断应该把这个消息发送给哪个应用窗口。

在每一个应用窗口对象 ViewRoot 中都包含一个 W 子类，该类是一个 Binder 类，InputDispatchThread



通过 IPC 方式调用 W 所提供的函数，从而把消息发送给对应的客户端窗口。

2.2 版本中的这种处理过程有两点被 Android 社区所诟病。

- 对所有原始消息的加工都在 KeyQ 类的 Java 代码中完成，这加大了消息处理的延迟。
- InputDispatcherThread 是通过 Binder 方式传递按键消息的，而 Binder 的延迟降低了用户操作的响应速度。

以上两点给用户的体验就是界面操作的延迟，比如滑动触摸屏时，界面的移动会延迟于指尖的移动。当然，不同硬件性能可能会有所差异，所以给用户的体验也有所不同。

于是，从 2.3 开始，我们欣喜地看到，Android 团队基本上对消息处理逻辑进行了重构，与以前版本的主要不同就是针对以上两个问题的。

- 获取消息的代码全部使用 C++ 完成，包括对消息进行加工转换。
- 抛弃了使用 Binder 方式传递用户消息到客户端，而是使用 Linux 的 Pipe 机制。由于笔者未详细分析 Binder 和 Pipe 的内部实现原理，对两者的效率差异尚不清楚，因此，本书仅介绍 Android 2.3 中是如何使用 Pipe 的，而不对比 Pipe 和 Binder 机制的效率。

Android 2.3 中的消息获取过程可简要描述为如图 11-1 所示。

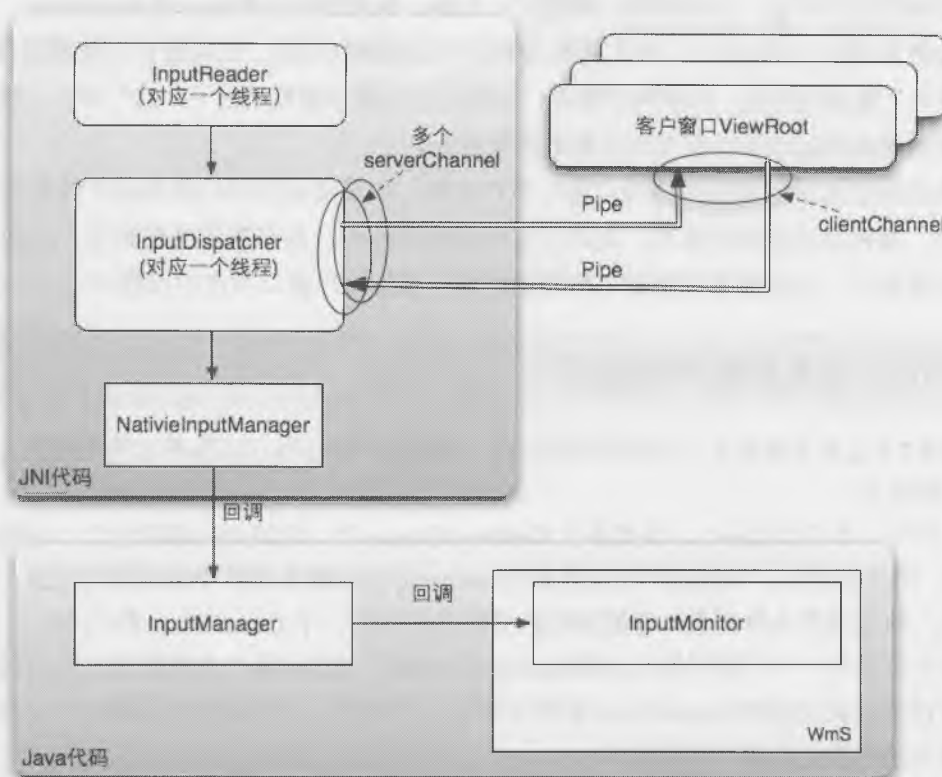


图 11-1 输入消息获取过程

首先, `InputReader` 线程会持续调用输入设备的驱动, 读取所有用户输入的消息, 该线程和 `InputDispatcher` 线程都在系统进程 (`system_process`) 空间中运行。`InputDispatcher` 线程从自己的消息队列中取出原始消息, 取出的消息有可能经过两种方式进行派发。第一种是经过管道 (`Pipe`) 直接派发到客户窗口中, 另一种则是先派发到 `WmS` 中, 由 `WmS` 经过一定的处理, 如果 `WmS` 没有处理该消息, 则再派发到客户窗口中, 否则, 不派发到客户窗口。

应用程序添加窗口时, 会在本地创建一个 `ViewRoot` 对象, 然后通过 `IPC` 调用 `WmS` 中的 `Session` 对象的 `addWindow()` 方法, 从而请求 `WmS` 创建一个窗口。`WmS` 会把窗口的相关信息保存在内部的一个窗口列表类 `InputMonitor` 中, 然后使用 `InputManager` 类把这些窗口信息传递给 `InputDispatcher` 线程。传递的过程中, `InputManager` 类需调用 `JNI` 代码, 把这些窗口信息传递到 `NativeInputManager` 对象中。

当 `InputDispatcher` 得到用户消息后, 会根据 `NativeInputManager` 中保存的所有窗口信息判断当前的活动窗口是哪个, 并把消息传递到该活动窗口。另外, 如果是按键消息, `InputDispatcher` 会先回调 `InputManager` 中定义的回调函数, 这既会回调 `InputMonitor` 中的回调函数, 这又会调用 `WmS` 中定义的相关函数, 所有这些回调函数的返回值类型都是 `boolean`。对于系统按键消息, 比如“Home”键、电话按键等, `WmS` 内部会按照默认的方式处理, 并返回 `false`, 从而 `InputDispatcher` 不会继续把这些按键消息传递给客户窗口; 对于触摸屏消息, `InputDispatcher` 则直接传递给客户窗口。

在 `InputDispatcher` 和客户窗口之间使用了管道 (`Pipe`) 机制进行消息传递。`Pipe` 是 `Linux` 的一种系统调用, `Linux` 会在内核地址空间中开辟一段共享内存, 并产生一个 `Pipe` 对象。每个 `Pipe` 对象内部都会自动创建两个文件描述符, 一个用于读, 另一个用于写。应用程序可以调用 `pipe()` 函数产生一个 `Pipe` 对象, 并获得该对象中的读、写文件描述符。文件描述符是全局唯一的, 从而使得两个进程之间可以借助这两个描述符, 一个往管道中写数据, 另一个从管道中读数据。管道只能是单向的, 因此, 如果两个进程要进行双向消息传递, 必须创建两个管道。

当客户窗口请求 `WmS` 创建窗口时, `WmS` 内部会创建两个管道, 其中一个管道用于 `InputDispatcher` 向客户窗口传递消息, 另一个用于客户窗口向 `InputDispatcher` 报告消息的执行结果。因此, 有多少个客户窗口, 就有多少个管道与 `InputDispatcher` 相连。

由于创建管道属于 `Linux` 系统调用, `Java` 不能直接调用, 另外也由于程序结构的需要, `Android` 中把调用管道的相关操作封装到了 `InputChannel.java` 类中, 同时该类中保存了 `InputDispatcher` 和客户窗口的管道收、发描述符。因此, `InputChannel` 也可以理解为一个“通道”, 在 `InputDispatcher` 端存在一个“服务端消息通道 (`serverChannel`)”, 在客户窗口端存在一个“客户端消息通道 (`clientChannel`)”。

“管道”与“通道”的区别是, “管道”是 `Linux` 系统的概念, 使用 `pipe()` 系统调用就可以创建一个管道, 而通道是 `Android` 内部定义的一个概念, 主要保存了通信双方所使用的“管道描述符”。

## 11.2 与消息处理相关的源码文件分布

和消息获取相关的源文件主要分布在三个地方, 如下:

第一个地方是: `frameworks/base/services/java: com.android.service`。

该处文件主要是 WmS 端要执行的功能，比如，保存所有的窗口信息，并向 InputDispatcher 提供回调函数，以便当 InputDispatcher 收到按键（Key）消息时，首先把该消息发送到 WmS 中，从而使得 WmS 可以优先处理一些系统按键。这些文件具体包括：

- WindowManagerService.InputMonitor
- InputWindow.java
- InputWindowList.java
- InputApplication.java
- InputManager.java
- ../jni/com\_android\_server\_InputManager.cpp

其中，InputWindow 记录了客户窗口的信息，但要注意，记录这些信息并不是给 WmS 用的，而是给 InputDispatcher 用的，如果没有 InputPatcher，则真的不需要该类。比如在 2.2 之前的版本中，就没有该类，而记录客户窗口信息并给 WmS 使用的是 WindowState 类。它与 InputWindow 的主要区别在于，InputWindow 将向 InputDispatcher 提供客户窗口的大小、位置、所在层等信息，从而使得 InputDispatcher 可以根据这些信息判定一个消息应该派发给哪个客户窗口，而 WindowState 除记录这些信息外，内部还包含很多函数，所以可以认为 InputWindow 是一个信息类，而 WindowState 是一个功能类。

InputMonitor 仅仅是辅助 WmS 进行消息处理，该子类在 Android 2.2 之前是没有的，但其功能却是存在的。在 Android 2.2 之前，无论是 AmS，还是客户端程序都有可能通知 WmS 改变某个窗口的显示状态，或者暂停某个窗口的消息派发，而这些状态改变后，都会导致其他相关窗口的状态也随之变化。很显然，这部分工作可以独立出来，所以就产生了 InputMonitor 子类。

InputManager 类是 WmS 与 InputDispatcher 模块的接口。

第二个地方是：frameworks/base/core/java: android.view.\*。

从这个地方就可以看出，这些文件肯定都是给客户端使用的，具体包括：

- InputChannel.java
- ../jni/android\_view\_InputChannel.cpp
- ../jni/android\_view\_InputQueue.cpp
- InputDevice.java
- InputEvent.java, abstract 类型
- InputQueue.java
- InputHandler.java, interface 类型
- KeyEvent.java
- MotionEvent.java

其中，InputEvent 是所有输入消息的基类，它是一个 abstract 类型，并且实现了 Parcelable 接口，即所有的输入消息都是可以跨进程传递的数据类。KeyEvent 和 MotionEvent 是 InputEvent 的两个实现，分别对应按键（Key）消息和触摸屏消息。

Android 2.3 版本之前没有 `InputQueue` 类，而只有 `MessageQueue` 类。同时，Android 2.3 版本中对 `MessageQueue` 的 `next()` 方法进行了扩展，以往该方法只从 `MessageQueue` 中读取消息，而在新版中，`next` 方法会首先从 native 的消息队列中读取消息，而 native 的消息队列正是 `InputQueue` 类，这也就是为什么 `InputQueue` 类的注释中说该类仅在 native 中使用的原由。如果 native 的消息队列有消息，则会直接回调该消息对应的 `InputHandler` 对象。从这点也可以看出，以后 `MessageQueue` 将主要用于保存非用户消息，而 native 消息队列则用于用户消息，这与 Android 2.3 之前的版本是完全不同的。在 Android 2.3 版本之前，所有的用户消息经过 IPC 传递到客户进程后，则首先发送到本地的 `MessageQueue` 中。

`InputHandler` 是一个 interface，内部包含了两个方法，`handleKey(...)` 和 `handleMotion(...)`。在 Android 2.3 版本中的 `Looper` 对象能够从所包含的文件描述符中读取消息，而该文件描述符一般都是以 `InputChannel` 作为参数传递给 native `Looper` 对象内部的，传递同时可以指定一个 `InputHandler` 对象，当 native `Looper` 从指定的文件描述符读取到消息时，首先会回调该 `InputHandler` 接口。因此，在客户端的 `ViewRoot` 中，会定义一个 `InputHandler` 对象，用于处理 `InputDispatcher` 传递的用户消息，如图 11-2 所示。

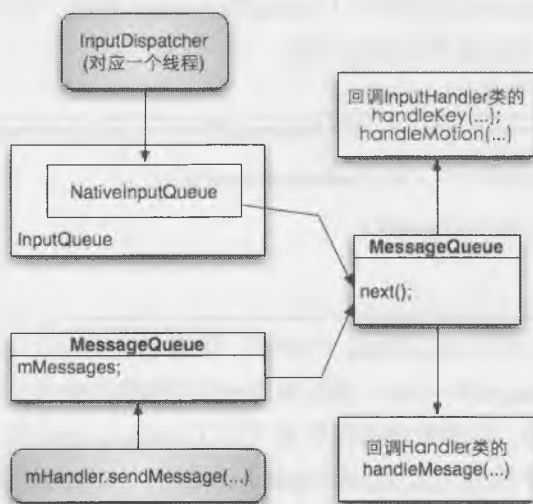


图 11-2 `MessageQueue` 的两种消息来源

第三个地方是：base/libs/ui。

从这个地方可以看出，这些是 native 代码，也就是 JNI 调用中所使用的基础 C/C++ 类，主要包括：

- `InputDispatcher.cpp`
- `InputManager.cpp`
- `InputReader.cpp`
- `InputTransport.cpp`

其中，`InputDispatcher` 类就是进行消息派发的线程类，`InputReader` 是进行消息读取的线程类，

InputTransport 是 native InputChannel 的实现类, 这个文件名称或许叫做 InputChannel.cpp 才更合适。InputManager 是 native 的 InputManager 类, Java 层面也有一个 InputManager 类, 两者的关系如图 11-3 所示。

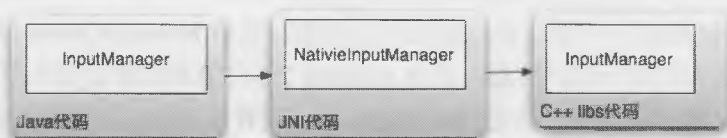


图 11-3 native 和 Java 层面的两个 InputManager 模块

### 11.3 创建 InputDispatcher 线程

InputDispatcher 线程是在 WmS 初始化时创建的。WmS 的构造函数中创建了一个 InputManager 对象, 然后调用 mInputManager.start()方法启动了 InputDispatcher 线程。那么, InputManager 如何对应到 InputDispatcher 线程? 下面就分析其代码调用过程。

InputManager 的构造函数如下:

```

110 public InputManager(Context context, WindowManagerService windowManagerService) {
111     this.mContext = context;
112     this.mWindowManagerService = windowManagerService;
113
114     this.mCallbacks = new Callbacks();
115
116     init();
117 }

```

这里需要注意两点。第一点是 mCallback 的赋值, 这里创建了一个 Callbacks 对象, 而该对象内部的函数则会利用 mWindowManagerService, 调用到 WmS 内部的其他方法。

第二点是 init()函数的调用, 该函数内部仅仅是调用了 nativeInit()函数, 而在 nativeInit()函数中将会创建一个 native 的线程, 而这个线程正是 InputDispatcher 线程。initNative()函数内部的执行流程如图 11-4 所示。

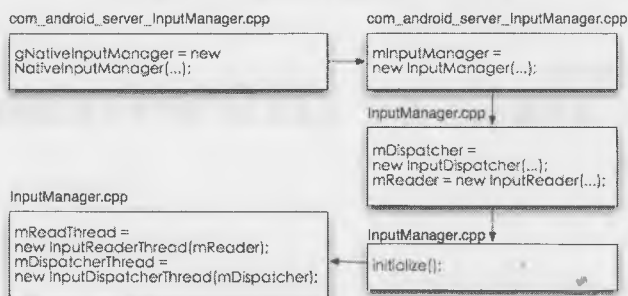


图 11-4 底层 InputManager 模块的初始化过程

首先在 InputManager.java 对应的 JNI 文件中创建一个 NativeInputManager 对象，而该对象内部的构造函数中又会创建一个 InputManager 对象。注意，这里的 InputManager 是 native 的，而不是 Java 层面的 InputManager 对象。

在 native 的 InputManager 的构造函数中会创建两个重要对象，一个是 InputDispatcher，另一个是 InputReader。前者的作用是进行消息派发，后者的作用是读取用户消息。InputManager 中接下来会继续调用 initialize() 方法创建真正的两个线程，分别是 InputReaderThread 和 InputDispatchThread。从这里也可以看出，InputDispatcher 本身并不是一个线程，而是一个功能对象，内部要根据窗口参数判断当前是哪个窗口，并根据触摸消息的位置判断应该把触摸消息派发到哪个窗口，只是 InputDispatcher 内部包含一个 InputDispatcherThread 线程，在这个线程中调用 InputDispatcher 中定义的各种功能。

从线程的角度来看，系统中只有一个 InputDispatcher 线程，也只有一个 InputReaderThread 线程，它们都运行在 system\_process 进程中，并且这两个线程都是在 WmS 构造函数中创建的。

#### 11.4 把窗口信息传递给 InputDispatcher 线程

前面讲过，InputDispatcher 要进行消息的窗口选择，那么它必须知道所有客户窗口的相关信息，而客户窗口的信息正是保存在 InputWindow.java 类中的。试想，InputDispatcher 要想知道用户消息应该派发给哪个窗口，至少要知道该窗口在屏幕上的大小，以及该窗口所在的层值，如图 11-5 所示。

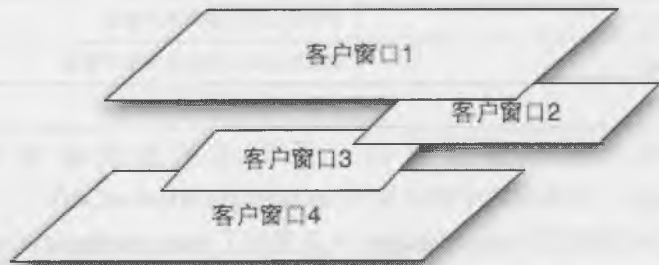


图 11-5 多窗口的层叠关系

当然，实际代码中除了包含这两个主要信息外，还保存了一些用户更细节控制的变量，比如窗口是否是 visible 的，窗口有没有用墙纸作为背景等。这些信息请参照源码，此处不再赘述。

那么，这些窗口信息如何及什么时候告诉给 InputDispatcher 呢？

先抛开源码本身，如果是我们设计，那么自然是当窗口状态发生变化时，就应该把变化后的窗口信息传递给 InputDispatcher，而能引起窗口状态变化的条件包括：

- 当某个窗口大小变化时。本来触摸位置对应的是窗口 1，而窗口大小变化后，该消息有可能就触摸到了窗口 2 上。

- 当某个窗口从显示状态变为隐藏状态，反之亦然。
- 当某个窗口暂停对消息的处理，或者重新恢复对消息的处理。
- 当某个窗口所在的层值发生变化时。
- 当添加一个新的窗口或者删除一个已有的窗口。

当然，实际设计时，可能还不止以上几种条件。根据这个思路，大家基本上可以在 WmS 中找到以上五个条件对应的代码。

InputMonitor 类中有一个 updateInputWindowLw() 函数，该函数会从 WmS 中的 mWindow 列表中取出当前所有客户窗口的属性，并把这些属性转换成 InputWindow 对象，然后调用 InputManager 的 setInputWindows(...) 函数，把转换后的对象传递给 InputDispatcher 对象。因此，可以认为，凡是调用 updateInputWindowLw() 的地方都是以上五种情况出现的地方，具体如表 11-1 所示。

表 11-1 WmS 中向 InputDispatcher 模块传递窗口信息的地方

调用 updateInputWindowLw() 的地方	描述
removeWindowInnerLocked()	删除窗口
relayoutWindow()	重新对窗口进行排版
removeWindowToken();	删除窗口
setTokenVisibilityLocked()	改变窗口的显示状态
InputMonitor.setInputFocusLw()	设置某个窗口为消息输入的对象窗口
InputMonitor.pauseDispatchingLw()	暂停指定窗口接收用户消息
InputMonitor.resumeDispatchingLw()	恢复指定窗口继续接收用户消息
performLayoutLockedInner()	对窗口重新进行排版

在表中你可能没有发现添加窗口，原因是所有的添加窗口都会间接调用 performLayoutLockedInner()，而该函数中则会调用 updateInputWindowLw()。

updateInputWindowLw() 函数在 InputManager 中又调用了 nativeSetInputWindow() 函数，所以，实际上把 InputWindow 信息传递给 InputDispatcher 是在 InputManager 的 JNI 代码中完成的。com\_android\_server\_InputManager.cpp 中 nativeSetInputWindow 函数的代码如下：

```
gNativeInputManager->setInputWindows(env, windowObjArray);
```

这又会调用 nativeInputManager 的相应方法，其方法的大致过程是从 windowObjArray 变量中取出所有的 InputWindow 信息，并将其转换为一个 native 的 InputWindow 类，转换后保存到一个 Vector 列表 windows 中，然后继续调用 InputDispatcher 类的 setInputWindows() 函数，如以下代码所示：

```
Vector<InputWindow> windows;
jsize length = env->GetArrayLength(windowObjArray);
for (jsize i = 0; i < length; i++) {
    ... //取出窗口信息并保存到 windows 变量中
}
mInputManager->getDispatcher()->setInputWindows(windows);
```



从而最终完成把 Java 中的 `InputWindow` 窗口信息传递给 `InputDispatcher` 线程对象中。

## 11.5 创建 `InputChannel`

前面说过, `InputDispatcher` 和客户窗口是通过 `Pipe` 传递消息的, 而 `Pipe` 是 Linux 系统调用的一部分, 但无论是 `InputDispatcher` 还是客户窗口, 真正感兴趣的只是 `Pipe` 所包含的读、写描述符。而为了程序设计的便利, Android 中增加了一个 `InputChannel` 类, 该类的基本作用有两个, 一个是保存消息端口对应的 `Pipe` 的读、写描述符, 另一个是封装了 Linux 的 `Pipe` 系统调用, 即程序员可以使用 `InputChannel` 所提供的函数创建底层的 `Pipe` 对象。

阅读源码时, 容易让大家产生困惑的地方在于, 一般认为, 既然 `InputChannel` 包含了 `Pipe` 的读、写描述符, 那么它就应该提供一个获取描述符的函数, 比如 `getReceiveFd()` 或者 `getSendFd()`, 然而 `InputChannel.java` 源码中并没有类似于这两个功能的函数。

为什么会这样呢? 有以下两个原因。

- 应用程序不应该获取读、写描述符。大家可能觉得奇怪, 既然应用程序真正感兴趣的是读、写描述符, 为什么又不应该获取呢? 因为, 就算能够获取描述符, 之后还需要把该描述符告诉给客户窗口中的 `InputQueue`, 而假如 `InputQueue` 就能够直接接收 `InputChannel`, 那么读、写描述符自然就不需要获取了, 相当于黑盒操作。
- 在 native 环境中也存在一个 `InputChannel` 对象, 该对象的确包含了 `getReceiveFd()` 和 `getSendFd()` 函数, 分别用于获取读、写描述符, 因为毕竟最底层的消息读、写都必须基于文件描述符。

Java 端 `InputChannel` 类提供了一个 `openInputChannelPair()` 函数用于一次性创建两个 `InputChannel` 对象, 为什么要创建两个呢? 一个用于 `InputDispatcher` 向客户窗口发送消息, 即所谓的服务端通道 (`serverChannel`), 另一个用于客户窗口向 `InputDispatcher` 发送消息。这里大家可能会觉得奇怪, 为什么客户窗口还要向 `InputDispatcher` 传递消息呢? 这的确有点冗余, 传统的嵌入式系统设计时, 只需要 `InputDispatcher` 把消息派发到客户窗口即可, 而在 Android 的消息子系统中, 当客户窗口处理完一个消息后还需要报告消息的执行结果, 而这就是 `clientChannel` 的作用。

首先, 创建 `InputChannel` 是从添加窗口开始的。当客户端需要添加窗口时, 会新建一个 `ViewRoot` 对象, 并调用 `ViewRoot` 的 `setView()` 方法, 该方法中会通过 IPC 方式调用 `WmS` 中的子类 `Session` 的 `addWindow()` 方法, 其中参数包括一个 `InputChannel` 对象, 但这个 `InputChannel` 只是一个空壳, 内部还不含任何读、写描述符。实际上, `ViewRoot` 正是希望 `WmS` 创建一个真正的 `InputChannel`, 并把其中的文件描述复制到这个壳子里。

`addWindow()` 方法内开始调用 `openInputChannelPair()` 方法, 如下代码所示:

```

1896         if (outInputChannel != null) {
1897             String name = win.makeInputChannelName();
1898             InputChannel[] inputChannels = InputChannel.openInputChannelPair(name);
1899             win.mInputChannel = inputChannels[0];
1900             inputChannels[1].transferToBinderOutParameter(outInputChannel);
1901
1902             mInputManager.registerInputChannel(win.mInputChannel);
1903         }

```

接下来分析 `openInputChannelPair()` 的执行过程，总体过程参见附图 5，为了叙述便利，下面截取部分过程，如图 11-6 所示。

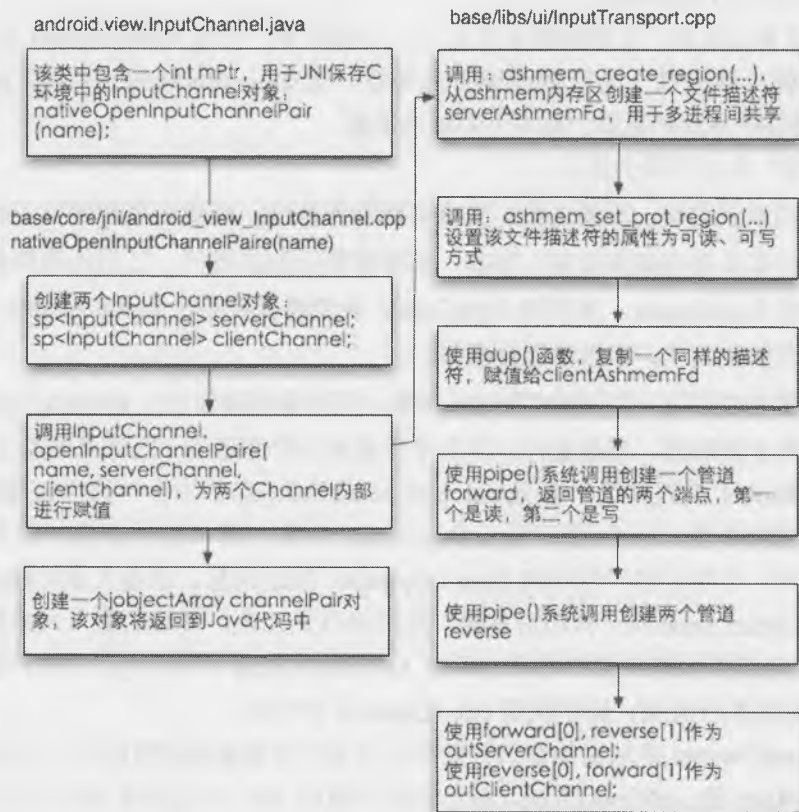


图 11-6 `openInputChannelPair()` 的内部过程

① 在 `InputChannel.java` 中调用 `nativeOpenInputChannelPair()` 方法。注意 `InputChannel.java` 类中有一个 `int mPtr` 变量，这个变量正是 `native` 环境中的 `nativeInputChannel` 对象，而 `mPtr` 是该对象的引用，这是典型的为 `native` 环境保持持久对象的做法，详情请参见本书第 2.2 节。

② 在 `android_view_InputChannel.cpp` 的 JNI 代码中，新建两个通道变量，分别是 `serverChannel` 和 `clientChannel`。此时，这两个变量只是两个空壳，内部的读、写描述符都是空的。

③ 接着调用 native 的 InputChannel 类的 openInputChannelPaire(...)函数, 参数包括“空壳子”通道, 因为在接下来的函数体中, 将为这两个空壳子赋值。注意, libs 目录中的 C++ 文件名称并不是 InputChannel.cpp, 而是 InputTransport.cpp, 这个显得有点不协调。

(1) 调用 ashmem\_create\_region(..)函数在匿名共享内存区申请一段共享内存, 该函数会返回一个文件描述, 应用程序可以像读写普通文件一样读写这段内存。

(2) 调用 ashmem\_set\_prot\_region(..)设置该段内存的访问方式, 比如只读、只写, 或者读写, 此处使用可读、可写属性。

(3) 使用 dup()系统调用, 把刚创建的 serverAshmemFd 文件描述再复制一个, 并赋值给 clientAshmemFd 变量, 从而无论是访问 server 还是 client, 实际上都对同一段内存。为什么要 dup()呢? 因为 InputDispatcher 线程和客户线程将共享这段内存区存储交互消息, 或者说, 两者都可以向这段内存区写数据, 至于这些消息是来自于 InputDispatcher 还是来自客户窗口, 则在消息本身中进行了标注。

(4) 使用 Linux 系统调用 pipe()创建一个管道, 名称为 int forward[2], 其中数组 0 代表了管道的写描述符, 数组 1 代表了管道的读描述符, 管道是单向的。

(5) 再创建一个管道, 名称为 int reverse[2]。

(6) 使用刚刚创建的共享内存和这两个管道创建两个 native 环境中的 InputChannel 对象, 如下代码所示:

```
outServerChannel = new InputChannel(serverChannelName,
    serverAshmemFd, forward[0], reverse[1]);

String8 clientChannelName = name;
clientChannelName.append(" (client)");
outClientChannel = new InputChannel(clientChannelName,
    clientAshmemFd, reverse[0], forward[1]);
```

代码中, 管道和通道的对应关系如图 11-7 所示, 其中 f 代表 forward 数组, r 代表 reverse 数组。

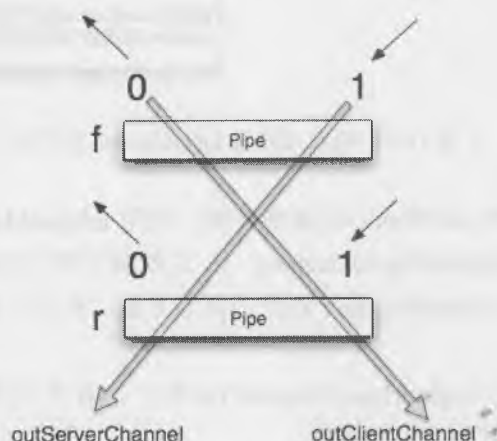


图 11-7 管道和通道的对应关系

这里请注意区别管道 (pipe), 共享内存 (ashmem)、通道 (channel) 三个概念的区别。pipe 用于 InputDispatcher 和客户窗口进行消息传递, 但有时可能需要传递大量数据, 因此, 又开辟了一段共享内存。这段共享内存不是必需的, 而通道是每个通信端点保存的一个数据类, 保存了 pipe 和共享内存的相关信息, 因此, 在 ViewRoot 类中包含一个 InputChannel 内部变量 mInputChannel, 而在 WmS 的 mInputManager 对象中也保存了和每个客户窗口通信的 InputChannel 对象列表。同时, 如图 11-7 所示中的两个 pipe 完全可以互换, 也就是说用 f0 和 r1 作为 outServerChannel, 用 r0 和 f1 作为 outClientChannel, 这都不影响系统的运行。

④ 从 native 环境中创建两个 Java 环境的 InputChannel 对象, 并返回到 Java 环境中。

至此, Java 环境中就有了两个真正的 InputChannel 对象。

## 11.6 在 WmS 中注册 InputChannel

上节说到, 创建了两个 InputChannel 对象, 一个用于服务端, 一个用于客户端。那么, 接下来就需要把服务端的 InputChannel 对象告知 InputDispatcher 线程, 其过程如图 11-8 所示。

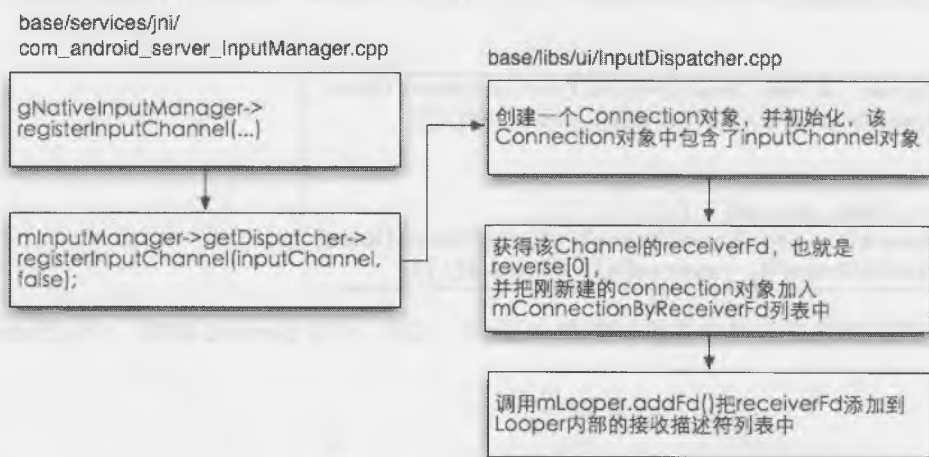


图 11-8 WmS 端注册 InputChannel 的过程

① 该过程是从 WmS 中的 addWindow()函数开始, 调用 mInputManager.registerInputChannel(..), 而 InputManager 内则调用 nativeRegisterInputChannel(...), 这就调用到了 JNI 代码中。

JNI 代码中则调用了 NativeInputManager 对象的同名方法, 然后又先获得 InputDispatcher 对象, 再调用该对象的同名方法。

在 InputDispatcher.cpp 的 registerInputChannel()函数中, 首先创建一个 Connection 对象, 并把 inputChannel 给该 Connection 对象的内部变量。

② 获得该 inputChannel 的 receiveFd, 此处对应的是上节中创建的 pipe 的 reverse[0]描述符, 然后

把该 `Connection` 对象加入到 `InputDispatcher` 的 `mConnectionByReceiverFd` 列表中。换句话说，`InputDispatcher` 和每一个 `serverChannel` 并不直接打交道，而是先把 `serverChannel` 转换为一个 `Connection` 对象，然后再操作这个 `Connection` 对象。

③ 调用 `mLooper.addFd()` 把 `receiverFd` 添加到 `Looper` 内部的接收描述符列表中。这就是之前为什么说线程感兴趣的仅仅是 `pipe` 的读描述符而已的原因，`native` 环境中的 `Looper` 类可以监控管道文件描述符，当该管道收到数据时会得到一个通知。

至此，无论是 `InputReader` 线程发往 `InputDispatcher` 的消息，还是客户窗口发送到 `InputDispatcher` 的消息，它都能接收到了。

有些读者可能有点好奇，如果不把这个 `serverChannel` 注册到 `InputDispatcher` 中，结果将会怎样？可以简单分析一下，首先，用户消息不是通过 `pipe` 发送到 `InputDispatcher` 线程中的，因此不注册应该不影响正常的消息处理。但另一方面，客户窗口却使用 `pipe` 机制传递每一个消息的执行状态到 `InputDispatcher` 中，因此，`InputDispatcher` 可能会因为没有收到上一个消息执行的确认而运行受阻。为了验证这一点，可以尝试在 `WmS` 中注释掉以下代码：

```
1902 //InputManager.registerInputChannel(win.mInputChannel);
```

然后重新执行 `-j4 sdk`，验证执行结果，启动的过程完全正常，而在桌面划屏解锁的过程中，按下屏幕的一瞬间，界面开始变化。接下来，除了状态栏外，界面就不动了。`Logcat` 的输出显示，系统出现了 `ANR` 提示，只是对应的窗口没有显示出来而已，因为屏幕还没有解锁，否则就会显示一个标准的 `ANR` 窗口。这个结果也就验证了 `serverChannel` 是用来接收消息处理结果的，并且可以想象，如果把消息处理的逻辑改成不需要确认消息的执行结果，那么这个 `serverChannel` 则就完全没有必要了。

## 11.7 在客户进程中注册 InputChannel

`WmS` 中创建了一对 `InputChannel`，其中 `serverChannel` 被注册到 `InputDispatcher` 线程中，另一个 `clientChannel` 则需要注册到客户进程中，从而使得客户进程可以接收到 `InputDispatcher` 发送的用户消息。

在客户端注册 `InputChannel` 和在 `InputManager` 中注册 `InputChannel` 的本质都是相同的，即告诉所在进程的 `native Looper` 对象，让它监控指定的文件描述符即可。客户端 `InputChannel` 的来源是调用 `WmS.Session` 的 `addWindow()` 方法时，最后一个参数是一个 `InputChannel` 类型的输出参数，即 `WmS` 中创建该对象，并赋值到这个参数中，因此 `InputChannel` 也必须实现 `Parcelable` 接口。

在 `ViewRoot` 中 `IPC` 调用 `addWindow()` 返回后，接下来就需要把该 `InputChannel` 注册到本线程中。`ViewRoot` 类中直接调用 `InputQueue.registerInputChannel(..)`，然后转而调用 `JNI` 函数 `nativeRegisterInputChannel()`，转而再调用 `native InputQueue` 的 `registerInputChannel(...)` 方法，该方法的内部流程如图 11-9 所示。

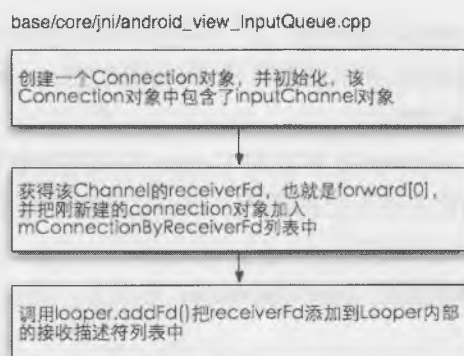


图 11-9 InputQueue 中注册通道的流程

- ① 创建一个 Connection 对象，并把 InputChannel 作为该对象的内部参数。
- ② 获得 InputChannel 的 receiverFd，即用于接收消息的文件描述符，这里应该对应的是 forward[0] 的值，然后把该 Connection 对象加入到 mConnectionByReceiverFd 列表中。对于客户进程而言，该 Looper 对象内部的 Connection 数目等于该进程所包含的 ViewRoot 的数目。而在 InputDispatcher 线程中，Connection 的数目等于所有客户进程中的 ViewRoot 总和。
- ③ 调用 looper 对象的 addFd() 把 receiverFd 添加到内部的接收描述符列表中，从而使得客户窗口可以接收到发往该文件描述符中的消息。

## 11.8 WmS 中处理消息的时机

用户消息可分为两类，一类是 Key 消息，另一类是 Motion 消息。在 Android 2.3 版本的消息系统设计中，对于 Motion 消息，InputDispatcher 会使用 pipe 直接把消息发往客户窗口，WmS 类不能对这些消息进行任何的前置（pre）处理，而对于 Key 消息，则会首先回调 WmS 中的 Key 消息处理函数，在 WmS 中不处理该消息时，才把消息发往客户窗口中。在一般情况下，WmS 中仅处理一些系统 Key 消息，比如“Home”键、照相按键、声音按键等。

在 WmS 中注册服务通道时，调用 Java 环境中的 InputManager 对象 mInputManager，而该类的构造函数中创建了一个 Callbacks 子类对象，并赋值给 InputManager 中的 mCallbacks 变量，然后在 init() 函数中调用 nativeInit(mCallbacks) 进行了初始化。变量 mCallbacks 作为初始化的参数传递到 native 环境中的 InputManager 对象中，从而使得在 InputDispatcher 进行回调时首先回调到 Java 环境的 InputManager.Callbacks 子类中。

比如，Callbacks 中包含了 interceptKeyBeforeDispatching(...) 函数，当 InputDispatcher() 接收到 Key 消息时，首先回调该函数，而该函数的内部代码中又调用了 WmS 对象中的 InputMonitor 的同名函数，如以下代码所示：

```

389     public boolean interceptKeyBeforeDispatching(InputChannel focus, int action,
390           int flags, int keyCode, int metaState, int repeatCount, int policyFlags) {
391         return mWindowManagerService.mInputMonitor.interceptKeyBeforeDispatching(focus,
392           action, flags, keyCode, metaState, repeatCount, policyFlags);
393     }

```

这就是 WmS 处理消息的时机。

## 11.9 客户窗口获取消息的时机

上节介绍了 Key 类消息首先被传递到 WmS 的时机, 本节介绍这些消息又是如何被客户窗口获取的。

无论是 Key 消息还是 Motion 消息, 都是通过 pipe 传递到客户窗口的。所有的客户进程都有一个主线程, 即 ActivityThread 类, 该类每次开始时就会进入到一个 Looper 循环中, 然后就不断地从 MessageQueue 中读取消息, 如果没有消息则进入到 wait 状态, 直到下一个消息。

而在 Android 2.3 版本中, MessageQueue 对象都会在 native 环境中创建一个 NativeMessageQueue 对象, 该对象对应了一个 native 的 Looper 对象, 该 Looper 对象可以从所指定的文件描述符中读取消息。

在 Java 中的 Looper.loop() 函数内部, 会调用 MessageQueue 的 next() 方法获取消息, next() 函数的部分代码如下:

```

111     final Message next() {
112         int pendingIdleHandlerCount = -1; // -1 only during first iteration
113         int nextPollTimeoutMillis = 0;
114
115         for (;;) {
116             if (nextPollTimeoutMillis != 0) {
117                 Binder.flushPendingCommands();
118             }
119             nativePollOnce(mPtr, nextPollTimeoutMillis);
120
121             synchronized (this) {
122                 // Try to retrieve the next message. Return if found.

```

该段代码的重点在于调用了 nativePollOnce() 函数, 该函数的第一个参数为 mPtr, 在 JNI 代码中, 会把这个 mPtr 强制转换为一个 native 的 NativeMessageQueue 对象。nativePollOnce() 内部会调用 native 的 Looper 对象的 pollOnce() 方法从所包含的文件描述符中读取一个用户消息, 如果没有用户消息, 则 UI 线程会进入 wait() 状态, 如果有消息, 则回调 ViewRoot 类中的 mInputHandler 对象, 回调完毕后, nativePollOnce() 也就执行完毕, 接下来继续调用 synchronized(this) 代码, 处理 MessageQueue 中的其他消息。

为什么 native 代码会回调 ViewRoot 中的 mInputHandler 对象呢? 因为, 在 ViewRoot 中的添加窗口函数 setView() 中, 向 NativeMessageQueue 对象添加了消息源 (文件描述符) 和回调函数, 如以下代码所示:

```

567         InputQueue.registerInputChannel(mInputChannel, mInputHandler,
568           Looper.myQueue());

```



其中 `mInputChannel` 中包含了消息源, `mInputHandler` 为回调接口, `Looper.myQueue()` 是 UI 线程的 `MessageQueue` 对象, native 环境中的 `Looper` 对象需要这个对象作为同步锁。同步锁的作用如下: 假设 Java 环境中开始调用 `nativePollOnce()`, 而此时没有用户消息, 这就会导致 UI 线程进入 wait 状态, 而接下来当客户进程中的其他线程向该 `MessageQueue` 中发送了一个消息时, 那么, 此时 `next()` 函数需要从 `nativePollOnce()` 函数中醒过来, 以便对该消息进行处理, 可是怎么醒过来呢?

查看 `MessageQueue` 的 `enqueueMessage()` 代码可以发现, 在函数的最后调用了一个 native 方法 `nativeWake(mPtr)`, 而在 `nativeWake()` 函数内部则使用了 `Looper.myQueue()` 作为同步对象, 并调用了该对象的 `notify()` 方法, 从而让 UI 线程从 `nativePollOnce()` 函数中跳出来。

`ViewRoot` 中的 `mInputHandler` 对象的定义如以下代码所示:

```
2794 private final InputHandler mInputHandler = new InputHandler() {
2795     public void handleKey(KeyEvent event, Runnable finishedCallback) {
2796         startInputEvent(finishedCallback);
2797         dispatchKey(event, true);
2798     }
2799
2800     public void handleMotion(MotionEvent event, Runnable finishedCallback) {
2801         startInputEvent(finishedCallback);
2802         dispatchMotion(event, true);
2803     }
2804 };
```

该回调对象中, 分别调用了 `ViewRoot` 中的 `dispatchKey()` 和 `dispatchMotion()` 函数, 而之后的过程就和 Android 2.3 版本之前的基本相同了, 即这两个函数会从窗口的根视图开始, 把消息依次派发给相应的子视图。关于 View 树内部的消息派发流程参见本书第 13 章。



## 第 12 章 屏幕绘图基础

屏幕绘图是 GUI 系统的基础，了解底层是如何在屏幕上绘制图形有助于理解 GUI 系统，并可开发具有任意绘制特性的 GUI 框架。

Android 中的 GUI 系统使用客户端和服务端配合的窗口系统，即后台运行了一个绘制服务，每个应用程序都是该服务的一个客户端，当客户端需要绘制屏幕时，首先请求服务端创建一个窗口，然后给该窗口绘制内容。对于每个客户端而言，它们都感觉自己独占了屏幕，而对于服务端而言，它会给每一个客户窗口分配不同的层值，并根据用户的交互情况动态改变窗口的层值，从而对用户来讲，出现了所谓的“前台窗口”和“后台窗口”。

本章就来介绍 Android 中的绘图框架的具体实现方式。

### 12.1 绘制屏幕的软件架构

Android 的屏幕绘制架构如图 12-1 所示。

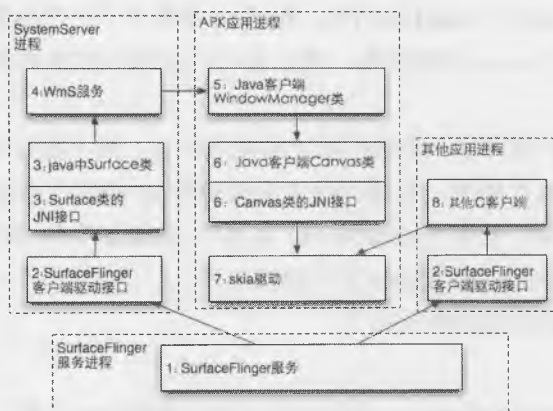


图 12-1 Framework 中的绘图架构

从进程的角度来看,该架构由  $2+x$  个进程组成,  $x$  的数量取决于系统当前正在运行的应用程序的数量,应用程序可以是 APK 对应的程序,也可以是任何 C 语言编写的 Linux 可执行程序,2 代表了 SurfaceFlinger 服务进程和 SystemServer 进程。

首先来看 SurfaceFlinger 服务进程,后面简称 sf。该进程在整个系统中只有一个实例,在系统开机后自动运行,它的作用是给每个客户端分配窗口,程序中用 Surface 类表示这个窗口。正如 Surface 字面的意义,它是一个“平面”,即每个窗口是一个平面,每个平面在程序中都对应一段内存,也就是所谓的屏幕缓冲区,不同窗口的缓冲区大小不同,这取决于该窗口的大小,即宽度和高度,一般来讲缓冲区的大小为宽度 $\times$ 高度。

sf 的客户端必须使用 SurfaceFlinger 的客户端接口驱动来和 sf 打交道,系统中使用该接口驱动的最重要的进程就是 SystemServer 进程。

当一个 APK 程序需要创建窗口时,首先在本地创建一个 Surface 对象,然后调用 WindowManager 类向 WmS 服务发起一个请求,请求的参数中包含该 Surface 对象。Surface 类仅仅是一个空壳,虽然它表示的是一个窗口,但它必须经过初始化后才真正能够对应一个在屏幕上显示的窗口,而初始化的本质就是给该 Surface 对象分配一段屏幕缓冲区内存。

WmS 收到这个请求后,会通过 Surface 类的 JNI 调用到 SurfaceFlinger\_client 驱动,通过该 client 接口驱动请求 sf 进程创建指定的窗口。于是 sf 创建一段屏幕缓冲区,并在 sf 内部记录该窗口,然后 sf 会把该窗口的缓冲区地址传递给 WmS, WmS 再用这个地址去初始化 APK 程序传入的 Surface 对象,并最终回到 APK 程序中。此时 APK 程序中的 Surface 对象是一个真正的 Surface 对象了,因为它包含的屏幕缓冲区已经由 sf 创建并备案了。

APK 程序有了这个 Surface 后,就可以给这个平面上绘制任意的内容了,比如绘制矩形、绘制文本、绘制图片、绘制渐变矩形等。然后 Surface 类本质上仅仅代表了一个平面,而绘制不同图案显然是一种操作,而不是一段数据,因此,Android 中使用了一个叫做 Skia 的绘图驱动库,该库使用 C/C++ 语言编写而成,其作用就是能够进行各种平面绘制。在程序中用 Canvas 类来表示这个功能对象,Canvas 类有很多绘制函数,比如 drawColor()、drawLine()等。Surface 类包含了一个函数 lockCanvas(), APK 应用程序可以通过该函数返回一个 Canvas 功能对象,然后就可以调用该对象的各种绘图函数完成对平面的绘制。

当然,如果想创建一个平面对象,给该平面上绘制点什么,而不想让 WmS 对此有所察觉,那么就可以直接编写一个 Linux 可执行程序,在该程序中直接调用 sf 的客户端接口驱动创建一个平面,然后调用 Skia 库对该平面进行绘制,就像 SystemServer 进程中 Surface 类的 JNI 函数中所做的一样,其本质就是使用 C/C++ 去调用 SurfaceFlinger\_client 客户端接口库。

系统中实现以上功能模块的源码路径如下。

- SurfaceFlinger 服务。

frameworks/base/service/surfaceflinger: 同时包含了头文件和 cpp 文件。

- SurfaceFlinger\_client 驱动接口。

➤ frameworks/base/libs/surfaceflinger\_client: 仅仅包含了 cpp 文件。

- frameworks/base/include/surfaceflinger: 包含 client 所需的头文件。
- Java 中的 Surface 类及其 JNI 接口。
  - frameworks/base/core/java/android/view/Surface.java。
  - frameworks/base/core/jni/android\_view\_Surface.cpp: 该 cpp 文件中需要包含 client 的头文件及 Skia 的头文件。
- WmS 服务中创建 Surface 的源码。

frameworks/base/services/java/com/android/server/WindowManagerService.java: 第 6316 行创建了一个 Surface 对象, 并且在第 6347 行中的 Surface.openTransaction() 和第 6369 行中的 Surface.closeTransaction() 代码区间给该 Surface 设置了层值、位置, closeTransaction() 执行完毕后, 该 Surface 就成了一个真正的 Surface。

- Java 客户端中的 WindowManager 类。
  - frameworks/base/core/android/view/WindowManager.java: 当应用程序需要创建一个窗口时, 可以调用该类的 addView() 函数。
- Java 客户端中的 Canvas 类及其 JNI 接口。
  - frameworks/base/graphics/java/android/graphics/Canvas.java。
  - frameworks/base/graphics/jni/: 该路径下什么都没有, 该路径应该包含下面路径的内容。
  - frameworks/base/core/jni/android/graphics/Canvas.cpp: 提供了 Canvas.java 所需的所有 JNI 支持, 这个路径设置得不太合理, 因为 JNI 一般应该放到 Java 源码所在的路径下。
- Skia 绘图驱动库。
  - external/skia/: 包含了所有的头文件和 cpp 文件。

## 12.2 Java 客户端绘制调用过程

上一节大致介绍了绘制架构及其调用过程, 本节从函数调用的角度来具体分析 Java 客户端绘制的调用过程。

首先来看创建 Surface 的过程。

对于基于 SDK 开发的 APK 程序而言, 不能直接创建 Surface 对象, 而只能使用 WindowManager 类的 addView() 方法创建一个窗口, 因为 Surface 的构造函数都是 @hide, 即不会出现在 SDK 中。Surface 类有两个构造函数, 分别如下:

```

216 public Surface() {
217     if (DEBUG_RELEASE) {
218         mCreationStack = new Exception();
219     }
220     mCanvas = new CompatibleCanvas();
221 }
```

```

186 public Surface(SurfaceSession s,
187               int pid, int display, int w, int h, int format, int flags)
188     throws OutOfResourcesException {
189     if (DEBUG_RELEASE) {
190         mCreationStack = new Exception();
191     }
192     mCanvas = new CompatibleCanvas();
193     init(s, pid, null, display, w, h, format, flags);
194 }

```

第一个构造函数没有参数，使用该构造函数创建的 Surface 对象仅仅是一个空壳，因为每个 Surface 内部都会对应一段屏幕缓冲区内存，对于空壳子 Surface 而言，这段内存并不存在。

第二个构造函数包含窗口大小相关的参数，使用该构造函数会创建一个真正的 Surface 对象。

WindowManager 类的 addView() 函数会创建一个 ViewRoot 对象，而 ViewRoot 类中则使用 Surface 的无参数构造函数创建了一个 Surface 对象，此时该 Surface 是一个空壳，然后 ViewRoot 类调用 WmS 中的 IWindowSession 服务为该 Surface 对象分配真正的屏幕缓冲区内容，如以下代码所示：

```

2647 int relayLayoutResult = sWindowSession.relayout(
2648     mWindow, params,
2649     (int) (mView.mMeasuredWidth * appScale + 0.5f),
2650     (int) (mView.mMeasuredHeight * appScale + 0.5f),
2651     viewVisibility, insetsPending, mWinFrame,
2652     mPendingContentInsets, mPendingVisibleInsets,
2653     mPendingConfiguration, mSurface);

```

而在 WmS 中则使用了 Surface 的第二个构造函数创建一个真正的 Surface 对象，并将该对象复制到 ViewRoot 的 Surface 对象中。下面就来看 Surface 的第二个构造函数的底层是如何实现的。

Surface 的构造函数中首先创建一个 Canvas 对象，该对象将作为底层 JNI 函数的一个外部引用，然后调用 init() 函数初始化该 Surface 对象。init() 函数是一个 JNI 函数，该函数中将创建一个真正的 Surface，该函数的实现在 android\_view\_Surface.cpp 中，其主要代码如下：

```

212 SurfaceComposerClient* client =
213     (SurfaceComposerClient*)env->GetIntField(session, sso.client);
214
215 sp<SurfaceControl> surface;
216 if (jname == NULL) {
217     surface = client->createSurface(pid, dpy, w, h, format, flags);

```

该段代码首先获得一个 SurfaceComposerClient 对象，该类正是 native 层面上 SurfaceFlinger 服务的客户端对象，然后调用该对象的 createSurface() 函数创建一个真正的 Surface 对象，参数中包含了当前客户端的进程标识 pid，以及请求的窗口的宽度和高度。

创建好了 Surface 后，应用程序可以使用 Surface 类的 lockCanvas() 获取一个 Canvas 对象，因此，下面接着来看 lockCanvas() 的内部过程。该函数内部调用了 native 函数 lockCanvasNative()，该函数的实现文件同样在 android\_view\_Surface.cpp 中，其内部执行过程分为以下几步。

- 1 首先获取 native 中的 Surface 对象。

- ② 调用 Surface 的 lock() 函数获取一个 SurfaceInfo 对象。
- ③ 从该 SurfaceInfo 对象中获取该 Surface 对应屏幕缓冲区内存地址。
- ④ 以上一步获得的地址构造一个 SkBitmap 对象。
- ⑤ 以上一步的 SkBitmap 对象构造一个 SkCanvas 对象，而这个 SkCanvas 对象是底层真正进行绘制的功能类对象，Java 层面的 Canvas 类仅仅是该类的包装而已。
- ⑥ 将该 Canvas 对象返回到 Java 端。

有了 Canvas 对象后，就可以调用该 Canvas 对象的绘制函数进行绘制了，下面以 drawColor() 为例说明其调用过程。Canvas.java 类中的 drawColor() 函数仅仅封装了 native 函数 native\_drawColor()，该函数是在 Canvas.cpp 中定义的，如以下代码所示：

```

324 static void drawColor__I(JNIEnv* env, jobject, SkCanvas* canvas,
325                          jint color) {
326     canvas->drawColor(color);
327 }
```

读者可能会觉得奇怪，为什么这个函数名称不是 native\_drawColor 呢？因为该函数名称已经被映射过了，关于 JNI 调用中的函数映射参见本书第 2 章。而此处映射的代码如下：

```

914 {"native_drawColor", "(II)V", (void*) SkCanvasGlue::drawColor__I},
```

该函数中仅仅是调用了 SkCanvas 对象的 drawColor() 函数，关于 SkCanvas 内部如何绘制一个颜色则完全属于 SkCanvas 的“内政”了。

以上就是 Java 客户端绘制的调用过程。

### 12.3 C 客户端绘制过程

为了理解 Android 中绘制架构中的客户端、服务端模式，本节实现一个 C 程序客户端窗口，并在该程序中绘制屏幕，该程序执行的结果是在屏幕的 (10, 10) 位置上绘制一个 176×144 大小的色块，其效果如图 12-2 所示。

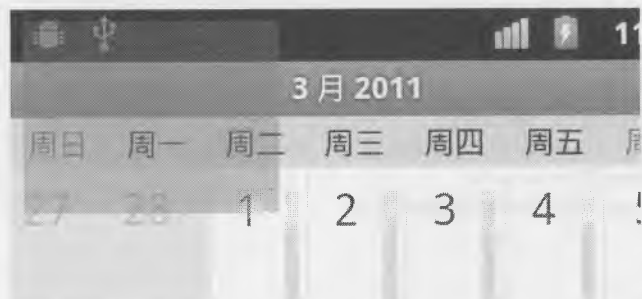


图 12-2 使用 C 程序直接绘制屏幕的效果

因为这是一个可执行程序，因此，读者可以在 Android 根目录的 external 目录下新建一个文件夹，姑且就叫做 MySurface 吧。然后在该目录下新建两个文件，分别为 Android.mk 和 Welcome.cpp，其中 Android.mk 是该工程的编译脚本文件，Welcome.cpp 是该项目中唯一的源文件。

Android.mk 的代码如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
    Welcome.cpp

LOCAL_SHARED_LIBRARIES := \
    libcutils \
    libutils \
    libui \
    libsurfaceflinger_client \
    libskia

LOCAL_C_INCLUDES += \
    $(LOCAL_PATH)/../../include/ui \
    $(LOCAL_PATH)/../../include/utils \
    external/skia/include/core \
    external/skia/include/utils \

LOCAL_MODULE := hellosurface
LOCAL_MODULE_TAGS := optional
include $(BUILD_EXECUTABLE)
```

这段代码是一个标准的 Android 编译系统的脚本文件，关于 Android 编译系统的结构见本书第 18 章。该文件有以下两点说明。

- LOCAL\_SHARED\_LIBRARIES 指定该项目依赖的库文件，此处依赖五个库文件，前面三个是一些标准库，后面两个分别是 SurfaceFlinger 的客户端接口库及 Skia 绘图驱动库。
- LOCAL\_C\_INCLUDES 指定该项目包含的头文件的搜索路径，由于本程序使用了 Skia 库，因此也需要包含该库的相关头文件。

Welcome.cpp 的文件源码如下。

```
#include <stdlib.h>
#include <stdio.h>

//the following is for surfacesupport
#include <surfaceflinger/SurfaceComposerClient.h>
#include <surfaceflinger/Surface.h>
#include <surfaceflinger/ISurfaceComposer.h>

#include <SkCanvas.h>
#include <SkBitmap.h>
#include <SkRegion.h>
```



```

using namespace android;

int main(int argc, char** argv)
{
    int pid = getpid();
    int x = 176, y = 144;

    sp<SurfaceComposerClient> videoClient
        = new SurfaceComposerClient;

    sp<SurfaceControl> spSurCtrl = videoClient->createSurface(pid,
        0, x, y, PIXEL_FORMAT_RGBA_8888, 0);
    SurfaceControl* surCtrl = spSurCtrl.get();
    const sp<Surface>& sur = surCtrl->getSurface();

    videoClient->openTransaction();
    surCtrl->setSize(x, y);
    surCtrl->setPosition(10, 10);
    surCtrl->setLayer(100000);
    videoClient->closeTransaction();

    surCtrl->show();

    SkBitmap *pmap = new SkBitmap();
    pmap->setConfig(SkBitmap::kARGB_8888_Config, x, y);

    Surface::SurfaceInfo info;
    sur->lock(&info);
    pmap->setPixels(info.bits);
    sur->unlockAndPost();

    SkCanvas *canvas = new SkCanvas(*pmap);
    canvas->drawColor(0x80509a47, SkXfermode::kSrc_Mode);

    while(1);
    // delete pmap;
    // delete canvas;
    // delete videoClient;
    return 0;
}

```

关于这段代码有以下几点说明:

第一, 在包含头文件中, 由于系统默认的头文件路径仅在 `frameworks/base/include` 中, 因此, `SurfaceFlinger_client` 中定义的头文件前面还需要加上 `SurfaceFlinger` 路径。而 `Skia` 相关的头文件路径已经包含在了 `Android.mk` 中, 因此此处仅写出头文件的名称而不需要路径。

第二, `videoClient->createSurface()` 返回的是一个 `sp<SurfaceControl>` 对象, 而不是 `Surface` 对象, 因此, 后面还需要调用 `surCtrl->getSurface()` 获取 `Surface` 对象。`sp` 是 `Android` 中定义的一个模板类, 其作用相当于 `Java` 语法中的 `SoftReference`, 由于 `C++` 语法中本身没有弱引用的概念, 所以程序员必须负责内存的分配和释放, 而 `sp` 类的作用正是能够自动释放内存, 当然这需要一些运行库的支持, `sp` 可理解

为 Soft Pointer。

同时需注意该函数的最后两个参数，分别是像素模式、缓冲模式，由于本例中绘制的是平面图形，该图形的颜色包含阿尔法通道，因此这两个参数分别为 `RGBA_8888` 和 `0`。在实际应用中，有时候需要创建一个 `Surface`，该 `Surface` 的作用是为了显示 `Camera` 捕获的视频，那么此时图像就不应该包含阿尔法通道，因此，这两个参数应该分别为：`PIXEL_FORMAT_OPAQUE` 和 `ISurfaceComposer::ePushBuffers`。

不同的像素模式是在 `frameworks/base/include/ui/PixelFormat.h` 文件中定义的，这就是为什么需要在 `Android.mk` 中包含该路径的原因。

第三，每个 `Surface` 包含的基本属性包括大小、位置、层值，要调整这些属性必须在 `openTransaction()` 和 `closeTransaction()` 中进行，而绘制屏幕的操作则没有这个限制。

第四，`Surface` 创建好后，默认状态是非显示状态，因此，必须调用 `show()` 方法使其显示出来，而 `show()` 方法是 `SurfaceControl` 类的方法，而不是 `Surface` 的。

第五，`Surface` 对应的屏幕缓冲内存地址在 `SurfaceInfo` 对象的 `bits` 变量中，而该对象是通过 `sur->lock()` 函数获取的，`lock()` 函数必须和 `unlockAndPost()` 函数成对出现，否则系统会死锁。

第六，`while(1)` 的作用仅仅是为了不让该程序结束，因为如果该程序结束，`SurfaceFlinger` 会自动将该程序申请的 `Surface` 删除掉，这就是 `createSurface()` 函数的第一个参数 `pid` 的作用，`SurfaceFlinger` 内部会记录每个 `Surface` 属于哪个进程。

编写完以上代码后，可以在 `Android` 源码根目录下运行 `make hellosurface` 命令进行该项目的编译，编译完成后会生成 `out/target/product/generic/system/bin/hellosurface` 可执行文件。然后可以使用 `adb push` 命令将该可执行文件复制到手机的 `system/bin` 目录下，最后再通过 `adb shell` 进入设备的命令行，并执行 `hellosurface`，运行结果就是本节开始所给出那张效果图。

关于如何对 NS 解锁，以及编译的更多内容，参见本书第 18 章。

## 12.4 Java 客户端绘制相关类的关系

在 `Java` 客户端，和绘制相关的类包括以下几种。

- `Surface` 类，该类用于描述一个绘制平面，其内部仅仅包含了该平面的大小、在屏幕上的位置，以及一段屏幕缓冲内存区。不过在 `Java` 端，不能直接访问这段内存，同时也不能通过该类直接设置该平面的大小和位置，而只能通过 `SurfaceHolder` 类。

在一般情况下，客户端程序对应的 `Surface` 是由底层的 `ViewRoot` 类进行创建的，而 `ViewRoot` 中创建 `Surface` 的函数在 `SDK` 中并没有开放，所以应用程序不能通过 `ViewRoot` 类直接创建 `Surface` 对象，而只能通过 `SurfaceView` 类间接创建。

关于如何使用 `SurfaceView` 创建 `Surface` 请参考作者的另一本书《`Android` 应用程序设计》。

`Canvas` 是一个功能类，该类包含各种绘制函数，比如 `drawColor()`、`drawLine()`、`drawText()` 等。构造 `Canvas` 对象时，必须为该 `Canvas` 指定一段内存地址，因为绘制的结果实际上就是给这段内存地址中填充不同的像素值。这段内存有两种类型，一种是普通的内存，另一种是屏幕缓冲区内存，当 `Canvas`

对应的内存为屏幕缓冲区内存时，绘制函数执行后就可以在屏幕上看到，如果是一段普通内存，则不会在屏幕上看到，不过却可以将这段内存复制到拥有屏幕缓冲内存的 Canvas 中，这种方式就是游戏开发中常用的方式。

- **Drawable** 类是一个抽象类，该类是一个功能类，但它与 Canvas 的相同之处是两者可以给内存缓冲区中绘制图案，两者的区别有两点。
  - **Drawable** 类内部不存在一段内存缓冲区，当应用程序需要绘制某种图案时，可以将一个包含内存缓冲区的 Canvas 对象传递给 Drawable，然后 Drawable 就可以给该 Canvas 上绘制相应的图案。
  - 每个具体的 Drawable 对象仅仅绘制某个特定的图案，SDK 中包含的 Drawable 实现类有 BitmapDrawable、NinePatchDrawable、PictureDrawable、ColorDrawable 等，这些类的名称标识该 Drawable 所能绘制的图案。

所以，打个比方就是，Drawable 是来料加工，而 Canvas 却是综合加工厂。

下面从应用的角度介绍一些 Canvas 绘制的常用方法，如以下代码所示：

```
final static int GRAY = 0xFFa0a0a0;
final static int DARK = 0xFF000000;
private void onDraw2(Canvas canvas){
    canvas.drawColor(0xFFFFFFFF);
    int h = canvas.getHeight();
    int w = canvas.getWidth();
    if(mPaint == null){
        mPaint = new Paint();
    }
    int textSize = (int) mPaint.getTextSize();

    Rect rt = canvas.getClipBounds();
    String str1 = rt.toShortString();
    canvas.drawText(str1, 0, 0 + textSize, mPaint);
    canvas.drawText("h="+ h + ",w=" + w, 0, 100, mPaint);

    canvas.translate(100, 50);
    canvas.drawCircle(0, 0, 2, mPaint);
    int saveCount = canvas.save();
    canvas.clipRect(20, 0, 40, 20);
    canvas.drawColor(GRAY);
    canvas.restoreToCount(saveCount);

    Matrix m = new Matrix();
    m.setScale(2,4);
    canvas.concat(m);
    canvas.clipRect(20, 0, 40, 20);
    canvas.drawColor(DARK);
}
```

这段代码绘制的效果如图 12-3 中白色区域所示。

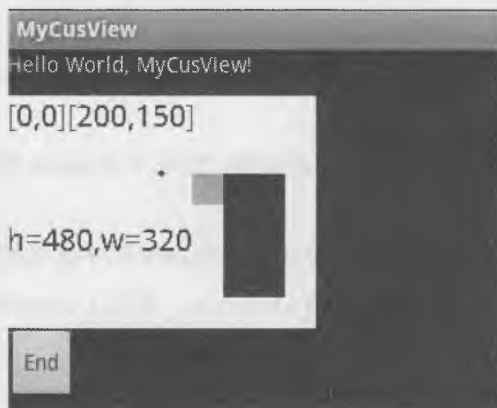


图 12-3 使用 Canvas 类绘图

这段代码的关键点在于以下几个方面。

第一，调用 `canvas.drawText()` 函数时，坐标原点相对的是该 Canvas 经过父窗口裁剪后的左上角，因此，当指定 (0,0) 点时，绘制就从左上角开始。

第二，当调用 `canvas.translate()` 后，坐标原点会调整到指定的位置，此处调整后的位置是相对老坐标的 (100,50) 处，所以 `drawCircle()` 参数为 (0,0) 时，就从 (100,50) 处开始绘制。

第三，无论子视图对 Canvas 做任何坐标平移或者裁剪，Canvas 本身的大小是不会改变的，所以，变量 `h` 和 `w` 始终为屏幕本身大小，即 480 和 320。

第四，Matrix 类的作用是改变绘制图案的形状参数，包括旋转、平移、缩放、阿尔法变换四种。值得注意的是这种改变仅仅针对的是剪切区中的图案，而并不是针对整个图案，同时，转换的内部不仅包括图案，还包括剪切区本身，这就是为什么本例中的灰色矩形区经过 Matrix 的缩放后会变为黑色矩形区。Matrix 的这种特性是后面将要介绍的动画原理的基础，正是因为这种特性，所以当使用 Matrix 对原来的图形进行动态变换时，变换后的图形可以超出本身剪切区进行显示。

关于 Matrix 的内部参数的意义，请读者自行试验研究，并参照大学课程中的《线性代数》一书，了解矩阵运算的相关知识，因为 Matrix 内部变换的核心是进行矩阵运算。



## 第 13 章 View 工作原理

### 13.1 导论

View 系统定义了从用户输入消息到消息处理的全过程，对于任何图形系统而言，该过程基本上都是相同的，如图 13-1 所示。

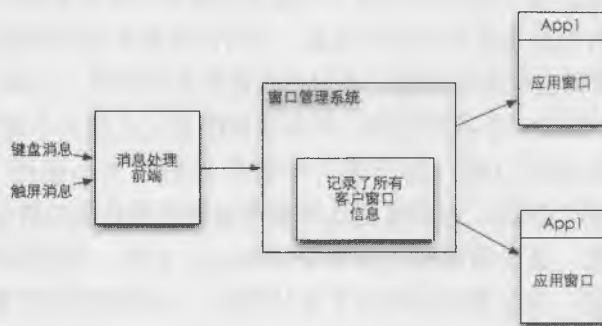


图 13-1 通用图形系统的消息处理过程

用户通过触摸屏或键盘等输入设备产生输入消息，该消息首先被消息处理前端转换为更明确的消息值，比如，物理键盘产生的原始消息一般为 DOWN/UP 消息，消息包含了原始的键盘码值。这些值由特定的硬件系统决定，不同的硬件系统会产生不同的消息值，而对于后续的消息处理模块而言，它需要统一的消息值，比如“A”、“B”、“C”、“D”等统一的字符值，消息处理前端的作用正是把这些特定的硬件消息值转换成操作系统所需要的统一值。

接下来,窗口管理系统(WmS)根据所有窗口的状态判断用户正在与哪个窗口进行交互,然后把该消息发送给当前窗口。因为窗口是由 WmS 创建的,因此 WmS 知道所有客户窗口的信息,包括窗口的大小、位置等。当消息到达时,如果是按键消息,则直接发送给当前窗口,而如果是触摸消息,则 WmS 会根据消息的位置坐标去匹配所有的窗口,判断该坐标落到了哪个窗口区域中,然后把该消息发送给相应的窗口。

最后,消息顺利到达目标窗口,至于目标窗口内部如何处理该消息则完全是窗口的“内政”。对于不同的 GUI 系统而言,会定义一些不同的、默认的处理逻辑,应用程序一般不直接处理该消息,而是重载一些特定的方法或实现一些特定回调函数,这也体现了这样一个概念:操作系统在调用我们,而不是我们在调用操作系统。这种架构设计仅仅是为了便利应用程序的开发。

在 Android 中,窗口管理系统的主体实现代码在 WindowManagerService 类中,其内部工作原理见第 14 章,本章主要介绍目标窗口内部的默认处理逻辑,或者叫做 View 系统。

View 系统获得消息后,会按照默认的逻辑来派发消息,主要就是把该消息派发给所有的子视图(View),以便相应的子视图能够获得消息并执行不同的任务。如果任务是一个纯后台任务,即该任务不会引起任何界面变化,那么 View 系统接下来仅仅按照默认逻辑继续派发下一个用户消息;而如果该任务会引起界面变化,那么 View 系统则要重新绘制界面。

重绘界面的过程大致分为三步:

① 计算该窗口中所有视图的大小,该步骤也称之为测量(measure)过程,即测量出所有视图的实际尺寸大小。有读者可能会问,如果仅仅是在编辑框中输入文字,所有的视图大小都没有变化,为什么还要计算视图大小呢?实际上,并不是所有导致界面重绘的操作都需要重新计算窗口的大小,在 View 的内部逻辑中,使用了一个内部变量保存相应的状态,当用户的某个操作导致改变了视图大小时,会设置该变量,而 View 的内部逻辑会根据该变量,决定是否需要重新测量。比如,当用户的某个操作导致程序调用 addView()时,由于添加一个新的视图会导致该视图的父视图大小发生改变,所以,需要重新测量。再比如,当第一次绘制窗口时,由于窗口中有些视图的大小使用 WRAP\_CONTENT 或者 MATCH\_PARENT 这些相对值,因此,也需要把这些参考值转换为具体的数值。

② 为所有视图分配位置,该步骤也称之为布局(layout)过程,即应该把视图放在屏幕的什么位置上。视图必须要有一个位置,该位置坐标相对于其父视图。不同视图的位置可以重叠,View 系统本身并不限定子视图的位置,Framework 中为了便于应用程序的设计,提供了 LinearLayout、FrameLayout 等不同的父视图,这些视图内部的子视图会按照定义的方式自动获取位置。

③ 把视图绘制到屏幕上。当系统获知了窗口中所有视图的大小和位置后,就可以完全确定屏幕上应该显示哪些视图了。在绘制时,系统内部为每一个窗口创建了一个画布(Canvas)对象,并把这个 Canvas 对象传递给从根视图到所有子视图。View 系统将 Canvas 传递给子视图时,都先将对该画布进行一次裁剪(Clip),从而在子视图看来,总是从画布的(0,0)坐标开始绘制。

本章内容的顺序是,首先介绍 View 系统内部的默认消息派发逻辑。默认消息处理中很多会引起视图本身状态的变化,并导致界面重绘,所以接着对视图状态的变化进行介绍。接下来再具体介绍 measure、layout、draw 三个操作的内部具体过程。最后,界面动画是 Android 中的一个特别 UI 效果,该效果本

身并没有超越 `measure`、`layout`、`draw` 三个操作，但却定义了一个巧妙的框架，使得应用开发可以方便地创建动画，所以本章一并对动画的内部框架进行介绍。

## 13.2 用户消息类型

用户消息是指经过消息处理前端把硬件物理消息转换成 Framework 内部定义的统一格式后的消息。这些消息目前分为三类，分别是按键消息(`KeyEvent`)、点消息(`Pointer`)或者也叫触摸消息、轨迹球(`Trackball`)消息。

本章不介绍轨迹球的消息及其处理过程，一方面因为其处理过程类似其他两种，而且更为简单，另一方面是因为轨迹球在新的 Android 设备中逐渐不被使用。

### 13.2.1 按键消息

按键消息的实现类是 `android.view.KeyEvent`，该类定义了消息包含的参数，以及获取这些参数的 API 接口。常用的 API 接口包括以下几项。

- `getAction()`：该函数返回按键动作，只有两种动作，`DOWN` 和 `UP`。
- `getKeyCode()`：该函数返回按键代码，这些代码是 Android 内部统一定义的，原始消息必须被转换成此代码才能被 Framework 处理。比如数字 0~9，字符 A~Z 等。当然，如果需要硬件的原始消息键值，可调用 `getScanCode()` 获得。

同时，`getKeyCode()` 可以用来处理组合键，比如“Shift + A”、“Alt + A”等，处理时先调用 `getKeyCode()` 获取具体的键值，比如 A，然后调用 `isShiftPressed()`、`isAltPressed()`、`isSymPressed()` 等分别判断是否有组合键按下。

- `getRepeat()`：该函数返回从按下后重复的次数。简单的讲，如果在 DOS 窗口中，按住字符“A”键不放，那么 `getRepeat()` 的次数就等于屏幕上显示的 A 的个数减 1。

当用户按“A”键，并持续一段时间，最后释放按键，假设系统默认逻辑中没有处理这些按键消息，那么该动作所产生的一系列消息如表 13-1 所示。

表 13-1 按键消息输出

消息 loop 次数	<code>getAction()</code>	<code>getKeyCode()</code>	<code>getRepeat()</code>
1	0	A	0
2	0	A	1
3	0	A	2
4	0	A	3
5	0	A	4
...	...	...	...
直到用户释放按键	1	A	0



以上过程需要注意两点。

第一点，loop 次数从 1 到 2，该过程往往会用比较多的时间，而从 2 到 3、3 到 4、4 到 5 等用的时间却相同，并且少于从 1 到 2。这种设计的原因是人类本身的生理特点，按下某个按键到弹起需要花一定的时间，如果 CPU 处理太快，带来的后果将是每当用户按一次按键，CPU 会处理多次该消息。举个例子，比如在 Nokia 手机的联系人列表中，用户要用方向键寻找某个联系人，如果用户按了一下“方向下”按键，并立即释放，用户的意思本来只想让光标移动到下一个联系人，此时，如果内部消息处理逻辑不给第一次 DOWN 消息加延时处理，那么 CPU 将会收到多次 DOWN 消息，就会把光标连续移动好几次，这显然是不好的。但同时，如果用户按下“方向下”按键后没有释放，这就意味着用户可能想连续多次移动光标，并且需要快速移动，此时消息内部逻辑不需要再加延时，所以用户会感觉移动得很快，这也正是用户想要的。

因为这种用户体验在任何带有光标移动的应用中都是适用的，因此，Android 中把这种消息延迟处理逻辑放在了内部消息获取前端中，应用程序不需要关心这种第一次时间延迟，而只需要按普通的 DOWN 消息处理即可。

第二点，View 系统内部所能获得的消息已经被前端处理过了，这种处理包括第一次按键的延时处理，而在 View 内部还进行了长按监测，因此，实际上从 DOWN 消息到发生长按，消息处理回调的时间由两部分组成，一个是消息处理前端所指定的第一次延迟，另一个则是 View 内部定义的一个长按时限。关于这点，将在后续小节详细介绍。

### 13.2.2 触摸消息

触摸消息的实现类在 `android.view.MotionEvent` 中，该类定义了和触摸相关的消息参数，并提供了一组 API 接口让用户获取这些参数，常用的 API 接口如下。

- `getAction()`: 获取消息动作，触摸消息动作的定义远远大于按键消息的动作，原因是当前的大多数触摸屏都支持多点触控，因此，触摸消息中必须包含是哪个点按下或者释放。
- `getEventTime()`和 `getDownTime()`: 前者获取本次消息发生的时间，而后者获取 DOWN 消息发生的时间，如果本次消息就是 DOWN 消息，两者的值相同，如果本次不是 DOWN 消息，那么则为前面最后一次 DOWN 消息发生的时间。
- `getPressure()`: 获取用户点击力量的大小，其值可以大于 1。
- `getSize()`: 该函数仅用于电容式触摸屏，它近似反映了用户触摸面积的大小，其值在 0~1 之间。对于压力触摸屏而言，由于内部逻辑只能定位到一个具体的点，所以无法使用该函数。
- `getX(int index)`和 `getY(int index)`: 返回指定触摸点对应的坐标，对于多点触控而言，参数 index 代表哪个点，从 0 开始。

触摸过程一般分为以下三种。

第一种，1 下→2 下→2 上→1 上，其对应的消息流程如下。

ACTION\_DOWN

ACTION\_POINTER\_DOWN 2

ACTION\_POINTER\_UP 2

ACTION\_UP

第二种，1 下→2 下→1 上→2 上，其对应的消息流程如下。

ACTION\_DOWN

ACTION\_POINTER\_DOWN 2

ACTION\_POINTER\_UP 1

ACTION\_UP

第三种，对于大于三点触摸，比如在 Nexus S 手机上，该手机实际支持 5 点触摸，选择 5 点的原因是人类有 5 根手指。当按下顺序为 1、2、3 时，弹起的顺序如表 13-2 所示。

表 13-2 多点触控的消息输出

弹起顺序	产生的 ACTION 序列
1, 2, 3	ACTION_POINTER_UP 1 ACTION_POINTER_UP 1 ACTION_UP
2, 1, 3	ACTION_POINTER_UP 2 ACTION_POINTER_1 ACTION_UP
3, 2, 1	ACTION_POINTER_UP 3 ACTION_POINTER_UP 2 ACTION_UP
3, 1, 2	ACTION_POINTER_UP 3 ACTION_PINTER_UP 1 ACTION_UP

对于更多的 4 点和 5 点，触点弹起顺序的不同将产生不同的 ACTION 序列，这些序列的规律有二。其一，如果小点已经弹起，那么后续的大点的 ACTION 都对应该小点；其二，最后一个点弹起时，直接发送 ACTION\_UP，无论该点是哪个点。

在 Android 的触摸消息处理中，从消息前端传递过来时，存在抖动问题。比如，当用户在某个位置按下时，虽然用户主观认为手指还没有在屏幕上移动，可实际上却做了微小的移动，因此导致 View 系统会收到多个 ACTION\_MOVE 消息，这就是为什么 Android 程序界面使用起来感觉有点晃动或者漂移，尤其是手指按着滑动时。相反，iPhone 的界面给人的感觉是“粘性”很好，界面似乎是被手指粘住了一样，笔者猜测在 iPhone 的 View 系统的消息处理前端增加了抖动逻辑，即当触摸电路获得原始的消息后，先判断该消息与之前消息的距离差，对于小于某个特定值的移动，比如不到二分之一毫米，则忽略

该消息，具体使用多大的抖动值，可以参考相关领域的经验值。

当产生 ACTION\_MOVE 消息时，可以使用 `getX(int index)` 获取指定点移动后的坐标。

尽管应用程序可以获得多点触控的消息，然而在目前的 View 系统中，并未提供多点触控的标准消息处理。比如，对于两个手指“捏、放”的动作，Android 并未提供标准的消息回调，应用程序必须自己去根据原始的触摸消息，包括 DOWN、UP、MOVE、POINTER 1, PONTNER 2 等，来计算移动的过程并判断是否是相应的动作。这显然增加了应用程序的复杂度。除此之外，更不用说三点以上的标准逻辑处理，比如，在 Mac 电脑的触控板上，可以通过三指或者四指产生一些标准的消息。然而，要提供一个稳定、高效的多点触控实现的确是一件挺复杂的事情，这牵扯到大量的实验并寻找一个合适的参数，不过这对于广大 Android 开发人员来讲也是一个机会，可以针对 Nexus 的 5 点触控来改造 View 系统，使之提供一个标准的多点触控接口以便应用程序方便调用。

### 13.3

### 按键消息派发过程

本节将介绍 View 系统内部如何派发按键消息，以及应用程序的消息处理代码在何时被回调。

#### 13.3.1 KeyEvent.DispatcherState 中的长按监测

上一节介绍过，消息处理前端内部有一定的延迟处理，其作用是消除人类生理反应时间，消息传过来后，View 系统内部又实现一个长按监测，相当于说实现了两次“长按”的处理。有必要先对这两个“长按”加以区别，以扫清后面分析消息处理过程的混乱。

首先来定义这两个长按。

第一个，消息处理前端的“长按”。当用户按下时，会产生一个 DOWN 消息，如果用户没有松手，那么会继续产生 DOWN 消息，第一个“长按”指的就是在这种情况下两次 DOWN 消息之间的时间间隔，该时间是在 C++ 源码中定义的。为叙述方便，以后称之为“生理反应时间”。

第二个，当用户按下时，会产生一个 DOWN 消息，在 View 类的 Java 代码中，会判断该消息对应的视图是否具有点击 (Click) 属性或者长点击 (long click) 属性，如果有并且该按键消息是来自于 DPAD\_CENTER，也就是方向键中间的那个“OK”键，此时 View 会发送 (Post) 一个异步消息，该消息会在指定的时间后执行一段 Runnable 代码，这段代码的作用就是回调为视图添加的 `onLongClick()` 接口。而这段时间就是 Java 代码中定义的“长按”时间，这段时间当前是 500ms，即半秒。

下面分析以上两个概念在代码中是如何体现的。

当发生按键消息后，经过重重处理，最后有可能执行到 KeyEvent 的 `dispatch()` 函数中，该函数中要分别针对 DOWN、UP 进行处理。在 DOWN 消息处理中，首先执行 receiver 的 `onKeyDown()` 消息，receiver 对象有可能是一个视图 (View)，也有可能是一个 Activity。

如果应用程序想响应“生理长按”消息，则需要重载 View 类的 `onKeyLongPress()` 函数；如果想响应“长按”消息，则仅需要调用视图的 `setOnLongClick()` 设置一个 Listener。

对于响应“生理长按”而言，除了重载 `onKeyLongPress()` 函数外，还必须在 `onKeyDown()` 消息中针对第一次 DOWN 消息返回 `true`，否则返回 `false`，如以下代码所示：

```
public boolean onKeyDown(int keyCode, KeyEvent event){
    if(keyCode == KeyEvent.DPAD_CENTER && event.getRepeat() == 0){
        return true;
    }else{ return false}
}
```

只有这样，在 `KeyEvent.dispatch()` 中调用 `receiver.onKeyDown()` 时才能返回 `true` 并且自动执行 `state.startTracking()` 进行消息跟踪，从而当第二次 DOWN 消息来临时，才能调用 `receiver` 的 `onKeyLongPress()`，如以下代码所示：

```
1261         } else if (isLongPress() && state.isTracking(this)) {
1262             try {
1263                 if (receiver.onKeyLongPress(mKeyCode, this)) {
1264                     if (DEBUG) Log.v(TAG, " Clear from long pr
1265                         state.performedLongPress(this);
1266                         res = true;
```

条件中调用 `isLongPress()` 方法判断是否是“长按”，这里的“长按”指的就是“生理长按”。该函数内部仅仅是从 `mFlags` 中取出 `FLAG_LONG_PRESS` 标志位，而 `mFlags` 变量则是由 native 的 C++ 代码监测到“生理长按”时进行相应的赋值。

紧接着，如果应用程序处理了“生理长按”，则 `onKeyLongPress()` 会返回 `true`，从而中止执行 `state.performedLongPress()` 函数。该函数内部仅仅是把该 `KeyEvent` 对象添加到 `state` 的一个内部列表中，当后面执行 UP 消息时，会执行到 `state` 的 `handleUpEvent()` 中，而在该函数中会判断 `KeyEvent` 对象是否在内部队列中。如果在，则意味着用户已经执行了长按处理，因此，需要把 `event` 的 `flag` 设置为 `FLAG_CANCELLED|FLAG_CANCELLED_LONG_PRESS`，以便后续处理对此有所知晓，从而进行不同的处理，如以下代码所示：

```
1381         int index = mActiveLongPresses.indexOfKey(keyCode);
1382         if (index >= 0) {
1383             if (DEBUG) Log.v(TAG, " Index: " + index);
1384             event.mFlags |= FLAG_CANCELED | FLAG_CANCELED_LONG_PRESS;
1385             mActiveLongPresses.removeAt(index);
1386         }
```

`KeyEvent.DispatcherState` 对象是包含在 `View` 内部的，每个 `View` 都有一个自己的该对象，该对象是在 `View` 的 `AttachInfo` 初始化代码中创建的。

下面分析“长按”的处理逻辑。`dispatch()` 中调用 `onKeyDown()`，在 `onKeyDown()` 函数的默认处理中，仅当按键是 `DPAD_CENTER` 时才启动长按监测，如以下代码所示：

```

4202         case KeyEvent.KEYCODE_DPAD_CENTER:
4203         case KeyEvent.KEYCODE_ENTER: {
4204             if ((mViewFlags & ENABLED_MASK) == DISABLED) {
4205                 return true;
4206             }
4207             // Long clickable items don't necessarily have to be clickable
4208             if (((mViewFlags & CLICKABLE) == CLICKABLE ||
4209                 (mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) &&
4210                 (event.getRepeatCount() == 0)) {
4211                 setPressed(true);
4212                 if ((mViewFlags & LONG_CLICKABLE) == LONG_CLICKABLE) {
4213                     postCheckForLongClick(0);
4214                 }
4215                 return true;

```

在该代码中，首先判断该视图是否是 Enable 状态，如果不是则直接返回 true，而不是 false。然后判断是否是第一次按下，即 `getRepeatCount()` 是否为 0，并且是 `CLICKABLE` 或者 `LONG_CLICKABLE`，如果满足，则调用 `setPressed(true)` 改变该视图的 Press 状态。接着又判断了一次视图是否是 `LONG_CLICKABLE`，如果是，才调用 `postCheckForLongClick(0)`，参数代表消息延迟的偏移，0 的意思就是不偏移，仅适用标准长按延迟。

在 `postCheckForLongClick()` 函数中，则调用 `postDelay()` 向 UI 线程发送一个 `Runnable` 消息，delay 的时间是 `ViewConfiguration.getLongPressTimeout()`，该静态函数返回值为常量 `LONG_PRESS_TIMEOUT`，也就是那个 500 毫秒。该 `Runnable` 对应的变量是 `mPendingCheckForLongPress`，如果过 500 毫秒后，消息依然存在，其内部的执行代码开始执行，调用 `performLongClick()`，进而回调应用程序提供的 `onLongClick()` 实现。而如果在 500ms 之前从消息队列中删除该 `Runnable` 消息，则不会执行长按消息。

当长按消息发生后，用户在 `onLongClick()` 回调中消耗了该消息，则 `View` 类中会将变量 `mHasPerformedLongPress` 置为 true。在随后的 UP 消息处理中，正是判断该变量是否为 false，如果为 false，则意味着长按消息还没有发生，于是调用 `removeMessage()` 从消息队列中移除长按消息，从而使其对应的 `Runnable` 不会执行，也就不会回调应用程序中的 `onLongClick()` 了。

以上过程的详细内容见后面小节。

### 13.3.2 按键消息总体派发过程

消息总体派发过程请参见附图 6。

首先，在 `ViewRoot` 中定义了一个 `InputHandler` 对象，当底层得到按键消息后，会回调到该 `InputHandler` 对象的 `handleKey()` 函数，该函数再调用 `ViewRoot` 中的 `dispatchKey()` 函数，该函数内部发送一个异步 `DISPATCH_KEY` 消息，消息的处理函数为 `deliverKeyEvent()`，该函数内部分以下三步执行。

① 调用 `mView.dispatchKeyEventPreIme()`，这里的 `PreIme` 的意思就是“在 Ime”之前，即输入法之前。因为对于 `View` 系统来讲，如果有输入法窗口存在，会先将按键消息派发到输入法窗口，只有当输入法窗口没有处理该消息时，才会把消息继续派发到真正的视图。所以，对于有输入法窗口显示的情况，如果应用程序员想在输入法截获消息之前处理该消息，则可以重载 `dispatchKeyEventPreIme()`，从

而处理一些特定的按键消息。执行完 `dispatchKeyEventPreIme()` 后, 如果该函数返回为 `true`, 则可以直接返回了, 但在返回之前如果 `WmS` 要求返回一个处理回执, 则需要先调用 `finishInputEvent()` 报告给 `WmS` 已经处理的该消息, 从而使得 `WmS` 可以继续派发下一个消息。

② 接下来就需要把该消息派发到输入法窗口。当然, 此时输入法窗口必须存在, 如果不存在的话, 则直接派发到真正的视图。

③ 调用 `deliverKeyEventToViewHierarchy()`, 将消息派发给真正的视图。该函数内部又可分为四步执行。

(1) 调用 `checkForLeavingTouchModeAndConsume()` 判断该消息是否会导致离开触摸模式, 并且会消耗掉该消息, 一般情况下该函数总是会返回 `false`。

(2) 调用 `mView.dispatchKeyEvent()` 将消息派发给根视图。对于应用窗口而言, 根视图就是 `PhoneWindow` 的 `DecorView` 对象; 对于非应用窗口, `mView` 就是任何 `ViewGroup` 的一个实现, 比如状态栏窗口, 它仅仅是一个 `FrameLayout` 视图。该函数内部则真正处理了按键消息, 并回调程序员所实现的消息处理代码, 该函数的详细分析见后面单独小节。

(3) 如果应用程序中没有处理该消息, 则默认会判断该消息是否会引起视图焦点的变化, 如果会, 则进行焦点切换。比如, 桌面上有好多个图标, 当按方向“上下”键时, 会聚焦到上一个或下一个图标, 具体过程如下。

① 根据键值判断是哪个方向, 当然, 如果不是方向键, 则会直接返回。

② 调用 `mView.findFocus()`, 寻找当前视图 `mView` 中的拥有焦点的子视图。一般情况下都会找到相应的子视图, 除非 `mView` 中的所有子视图都是不可 `Focus` 的。如果找到了, 则赋值给临时变量 `focused` 并继续往下执行。

③ 调用 `focused.focusSearch(dir)` 寻找 `focused` 视图的下一个视图。参数 `dir` 代表方向, 即往哪个方向上寻找下一个焦点视图。如果找到了, 并且下一个视图不是当前焦点视图, 则将下一个焦点视图赋值给临时变量 `v`。

④ 既然存在下一个焦点视图, 接下来就需要让该视图获取焦点。由于下一个焦点窗口需要知道上一个焦点区的位置, 从而确定下一个焦点窗口内部具体应该聚焦到什么地方, 所以需要首先计算出上一个焦点区的坐标。

a. 调用 `focused.getFocusedRect()`, 该函数返回的是焦点区相对于父视图的位置坐标。

b. 调用 `mView.offsetDescendantToMyCoords()`, 即把这个焦点区的坐标转换到 `mView` 所在的坐标区。

c. 调用 `mView.offsetRectIntoDescendantsCoords()`, 即把这个新的区域再转换到指定子视图坐标区中的位置。

以上三次转换过程如图 13-2 所示。

假设 `focused` 是当前拥有焦点的 `View`, 它的父视图是 1, 下一个焦点视图是 `v`, 它与 1 同在一个 `ViewGroup` 中。`focused.getFocusedRect()` 所获得的焦点区仅仅是相对于 1 点, 因此最终需要把这个区域转换到和 `v` 相同的父视图即 3 所在的坐标区。`offsetDescendantsRectToMyCoords()` 可以把焦点区转换到相对于 2 的坐标区, 而 `offsetRectIntoDescendantsRect()` 则会把相对于 1 的区域转换到相对于 2 的区域。

以上调用过程注意其参数的传递。

为什么 `v` 在获取焦点之前还需要上一个焦点区的具体区域值呢？在一般情况下，的确不需要，但如果 `v` 本身是一个 `ViewGroup`，而不是一个简单的 `View`，则 `v` 内部还有几个不同的子视图，如图 13-3 所示。

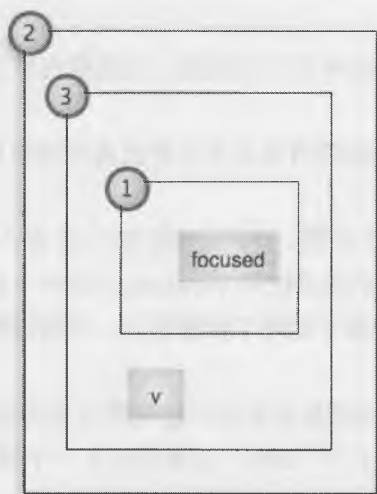


图 13-2 消息在不同视图中的位置

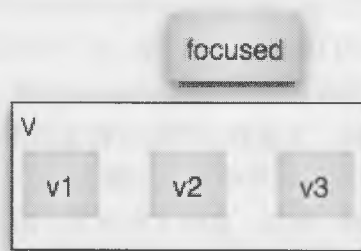


图 13-3 ViewGroup 中包含多个 View 对象

在这种情况下，`v` 内部到底是应该给哪一个子视图分配焦点呢？这自然与 `focused` 的位置有关系，如果 `focused` 靠左，那么应该 `v1` 获得焦点，如果靠右，则 `v3` 应该获得焦点。

⑤ 调用 `v.requestFocus()` 为 `v` 获取焦点。在一般情况下都会获取成功，如果获取失败，则调用 `dispatchUnhandleMove()`，应用程序可以重载该函数以做别的事情。

(4) 到这里，按键消息就处理完毕了，此时应该返回。如果 `sendDone` 参数为 `true`，即 `WmS` 需要客户窗口发送执行完毕的回执，则在返回前调用一次 `finishInputEvent()`。

总体调用过程可以简要概括为：如果窗口存在输入法窗口，则先把按键消息交给输入法窗口处理，不过在处理之前程序员可以重载那个 `preIme` 函数以截获一些特别的按键消息。输入法如果没有消耗该消息，消息则进入 `View` 树进行处理，处理完所有消息后如果 `WmS` 要求发送执行完毕回执，则调用 `finishInputEvent()`。

### 13.3.3 根视图内部派发过程

上节讲到，系统调用 `mView.dispatchKeyEvent()` 将消息派发到根视图，这个根视图要么是 `PhoneWindow.DecorView`，要么是一个普通的 `ViewGroup` 实现，比如 `FrameLayout` 或者 `LinearLayout`，对于所有在 `Activity` 中包含的窗口而言，都会对应到 `DecorView` 中，本节就来介绍 `DecorView` 这个根视



图内部的按键消息派发流程。整体流程图参见附图 6。

在 DecorView 的 dispatchKeyEvent() 中:

① 处理音量键。这段代码仅仅是增加了一个用户体验而已, 在一般情况下, 当用户按音量键后, 都会调用音量调节逻辑, 调用的参数中会指定是否产生发音, 但如果用户按下音量键后立即按其他键 (300ms 之内), 本段代码在设置音量的调用参数中会取消产生发音, 从而避免下一个操作可能产生的音频和音量音产生混叠。

② 处理系统快捷键。在应用程序开发时, 程序可以给菜单指定快捷键, 本段代码正是处理这些快捷键的。主要包含两种模式, 一种是菜单窗口没有显示时, 这需要组合键才能产生快捷操作, 另一种是菜单窗口显示时, 此时只需要快捷字符键就可以启动快捷操作。

本段代码仅适应于第一个 DOWN 消息, 就是用户刚刚按下的那个 DOWN 消息, 此时消息的 repeat 值为 0。

(1) 判断 mPanelChordingKey 是不是为 0。当按下 “Menu” 键时该变量会给出赋值为 “Menu” 键的键值, 一般为 82, 当 “Menu” 键弹起时, 该值会被清空为 0。该变量中 Chording 的语义是 “和弦”, 即组合的意思。如果已经按下 “Menu” 键, 则调用 performPannelShortcut() 处理菜单快捷键。

(2) 如果菜单快捷键中没有处理该键值, 则调用 dispatchKeyShortcutEvent() 将这个 “Menu + 具体键值” 的组合键发送给 View 树去处理。

(3) 判断当前菜单窗口是否打开, 如果打开了, 则可以直接响应具体的快捷键值, 而不需要 “Menu” 组合键按下。调用 performPannelShortcut(), 注意该函数最后一个参数和第 (1) 步中所用的参数不同。

③ 如果按键不是菜单快捷键, 那么调用根 View 中的 Callback 对象的 dispatchKeyEvent(), 该 Callback 对象一般就是当前的 Activity, 大家可以查看 Activity 类的定义, 它实现了 Window.Callback 接口。当然, 如果该 PhoneWindow 对应的不是 Activity, 那么则调用 super.dispatchKeyEvent(), 这就导致调用到 ViewGroup 的同名函数。关于 Activity 内部的消息派发过程将在后续小节中单独介绍。

④ 如果 Activity 中没有消耗该按键消息, 那么就调用 PhoneWindow 中定义的 onKeyDown() 对该消息进行最后的处理, 当然, 该函数内部只处理少数特定的按键消息。该过程将在后续小节中详细介绍。

### 13.3.4 Activity 内部派发过程

当消息进入 Activity 内部后, Activity 内部的 dispatchKeyEvent() 有默认的处理逻辑。它会先回调该 Activity 包含的 Window 对象的相应方法, 并将消息派发给 DecorView 对象的视图树内部, 然后才回调 Activity 内部的 onKeyDown() 和 onKeyUp() 函数。你可能觉得奇怪, Activity 处理消息的时机不是从 Window 类中传递过来的吗? 为什么还要再回调呢, 岂不是多此一举? 这种设计的原因是, 应用程序可以重载 dispatchKeyEvent(), 从而不按照默认的逻辑进行处理, 这有可能导致按键消息永远不会传递到 View 树内部的视图中, 所以一般情况下并不重载该函数。

下面具体分析 Activity 内部的消息派发过程。

## 1. 主体派发过程

派发过程参见附图 6。如果应用程序没有重载 `dispatchKeyEvent()` 函数，那么该函数将按照内部的默认逻辑进行消息处理。

- ① 回调 `onUserInteraction()` 函数。应用程序可重载 `Activity` 的该函数，以便在消息交互前做点什么。
- ② 回调 `Activity` 包含的 `Window` 对象的 `superDispatchKey()`，`Window` 类中该函数继而调用 `mDecor.superDispatchKeyEvent()`，这里的 `mDecor` 对象正是 `PhoneWindow` 中的 `DecorView` 对象。该函数继而又调用 `super.dispatchKeyEvent()`，`DecorView` 的父类是 `FrameLayout`，该类并没有重载 `dispatchKeyEvent()` 函数，因此，`super` 最后会对应到 `ViewGroup` 的同名函数。

在 `ViewGroup` 中，如果 `ViewGroup` 本身拥有焦点，则调用 `super.dispatchKeyEvent()` 将该消息派发到该视图本身；如果该 `ViewGroup` 本身没有获取焦点，但其包含的子视图却拥有焦点，则调用子视图 `mFocused` 的 `dispatchKeyEvent()`。

这里注意区别“`ViewGroup` 本身拥有焦点”和“子视图拥有焦点”，源码中判断 `ViewGroup` 本身是否拥有焦点的代码如下：

```

787         if ((mPrivateFlags & (FOCUSED | HAS_BOUNDS)) == (FOCUSED | HAS_BOUNDS)) {
788             return super.dispatchKeyEvent(event);
789         } else if (mFocused != null && (mFocused.mPrivateFlags & HAS_BOUNDS) == HAS_BOUNDS) {
790             return mFocused.dispatchKeyEvent(event);
791         }
792         return false;

```

其中变量 `mPrivateFlags` 是在 `View` 中定义的，因为 `ViewGroup` 本身也是一个 `View`。`ViewGroup` 如果拥有焦点的话，那么 `HAS_BOUNDS` 标识必须存在，而 `mFocused` 是该 `ViewGroup` 所包含的子视图。对于多层 `ViewGroup` 嵌套而言，比如 `A` 中包含 `B`，`B` 中又包含 `C`，假设 `C` 是一个具体的 `View`，并且拥有焦点，那么只有 `C` 中的 `mPrivateFlags` 拥有 `FOCUSED` 标识和 `HAS_BOUNDS` 标识，对于 `B` 则只有 `FOCUSED` 标识，并且其中的 `mFocused` 变量对应 `C`，对于 `A`，也只有 `FOCUSED` 标识，并且其中的 `mFocused` 变量对应 `B`。`dispatchKeyEvent()` 首先从 `C` 中开始执行，然后递归调用到 `B` 的 `dispatchKeyEvent()`，最后才能调用到 `C` 的 `dispatchKeyEvent()`，也就是 `View` 的 `dispatchKeyEvent()`。

在 `View` 类的 `dispatchKeyEvent()` 函数中，首先回调 `onKey()` 函数，应用程序可以重载该函数以实现自定义的消息处理。如果在 `onKey()` 函数中没有消耗该消息，那么系统将执行默认的消息处理，这是通过调用 `event` 对象的 `dispatch()` 函数完成的，在调用该函数时，第一个参数 `receiver` 对应的是 `View` 对象本身。

- ③ 如果拥有焦点的 `View` 没有处理该按键消息，则继续调用 `event` 的 `dispatch()` 函数。注意本次调用该函数时第一个参数 `receiver` 对应的是 `Activity` 对象。

## 2. KeyEvent 的 dispatch() 过程

上一节中的第二步和第三步，都调用了 `event` 对象的 `dispatch()` 函数，该 `event` 对象是一个 `KeyEvent` 类。这两步中的区别在于调用 `dispatch()` 函数时的第一个参数 `receiver` 针对的对象不同。

`dispatch()` 内部使用 `switch case` 语句分别针对 `ACTION_DOWN` 和 `ACTION_UP` 及

ACTION\_MULTIPLE 进行处理。

首先来看 ACTION\_DOWN, 该段代码如下:

```

1253         mFlags &= ~FLAG_START_TRACKING;
1254         if (DEBUG) Log.v(TAG, "Key down to " + target + " in " + state
1255             + ": " + this);
1256         boolean res = receiver.onKeyDown(mKeyCode, this);
1257         if (state != null) {
1258             if (res && mRepeatCount == 0 && (mFlags & FLAG_START_TRACKING) != 0) {
1259                 if (DEBUG) Log.v(TAG, "Start tracking!");
1260                 state.startTracking(this, target);
1261             } else if (isLongPress() && state.isTracking(this)) {
1262                 try {
1263                     if (receiver.onKeyLongPress(mKeyCode, this)) {
1264                         if (DEBUG) Log.v(TAG, "Clear from long press!");
1265                         state.performedLongPress(this);
1266                         res = true;
1267                     }
1268                 } catch (AbstractMethodError e) {
1269                 }
1270             }
1271         }
1272         return res;

```

① 清除 mFlags 中的 FLAG\_START\_TRACKING 标识。

② 回调 receiver.onKeyDown()。receiver 可能是 View 对象, 也可能是 Activity 对象, 两者的具体处理过程见后面小节。

③ state 是一个 DispatcherState 对象, 每个 View 对象内部都包含一个该对象, 它是当 View 被添加到 View 树中时自动创建的, 其作用是为该 View 对象保持一个消息处理状态, 该对象一般都不为空。该步主要实现“生理长按”监测, 并当发生“生理长按”消息时, 回调 receiver 的 onKeyLongPress() 函数。

(1) 调用 state.startTracking() 开始对消息进行跟踪, 注意 KeyEvent 类本身也有一个无参数的 startTracking() 函数, 其作用仅仅是为 mFlags 添加 FLAG\_START\_TRACKING 标识, 而此处调用的是 DispatcherState 对象的 startTracking() 方法。该方法要被调用, 有如下三个条件必须同时满足:

- res 为 true, 即 receiver 的 onKeyDown() 必须返回 true, 其语义就是该 receiver 必须要对该 DOWN 消息有兴趣。对于一般的消息处理函数而言, 返回 true 意味着消耗了该消息, 而对于 receiver 的 onKeyDown() 而言, 只有它消耗了该消息, 才能进入。
- mRepeat 等于 0, 即按下后的第一个 DOWN 消息。
- mFlags 包含 FLAG\_START\_TRACKING 标识。大家可能觉得奇怪, 第一步中不是已经清除了该标识吗? 的确已经清除了, 因此, 如果应用程序想要能响应“生理长按”消息, 就必须在 onKeyDown() 重载中为 mFlags 添加该标识, 这就是上面说的 KeyEvent 内部的 startTracking() 函数的作用。回调 onKeyDown() 时, 第二个参数正是 KeyEvent 对象本身, 因此, 可以在 onKeyDown() 的函数实现中调用该 KeyEvent 对象的 startTracking() 函数。

(2) 回调 receiver 对象的 onKeyLongPress() 完成用户指定长按处理代码。回调的条件有二。

- `isLong()`函数返回为 `true`。该函数内部仅仅是判断 `mFlags` 中是否包含 `FLAG_LONG_PRESS` 标识, 而该标识是 `native` 的 `C++`代码进行消息获取时赋值的。
- `state.isTracking()`返回 `true`, 即当前正在进行监测。

以上代码虽然简单却容易让人混淆, 问题的关键就是要区分“生理长按”和“长按”。再次强调, “生理长按”是 `native` 的 `C++`代码在产生消息时指定的, 而“长按”是在 `Java` 代码中通过发送一个异步延迟消息进行监测的, 延迟时间为 `500ms`。

应用程序要处理“生理长按”必须做以下几件事情。

- 必须重载 `onKeyDown()`, 并且要返回 `true`。
- 在 `onKeyDown()`中必须调用 `event.startTracking()`对象消息进行跟踪。
- 重载 `onKeyLongPress()`, 执行长按消息的处理代码。

接着来看 `ACTION_UP`。该段代码中首先调用 `state.handleUpEvent()`, 该函数的作用是判断是否已经发生“生理长按”消息、应用程序是否消耗了该“生理长按”消息。如果发生并且消耗了, `handleUpEvent()` 函数内部将给该 `event` 添加一个 `FLAG_CANCELED` 标识和 `FLAG_CANCELED_LONG_PRESS` 标识, 然后继续调用 `receiver.onKeyUp()`消息。因此, 在 `onKeyUp()`的重载中, 应用程序一般应该检查该消息的 `flag`, 判断在该 `UP` 消息之前是否处理了“生理长按”, 以便做出更友好的行为。

最后是 `ACTION_MULTIPLE` 消息, 该消息基本上不会发生, 这段代码仅仅是做了一些容错处理, 有兴趣的读者可自行分析。

### 13.3.5 View 类内部的 `onKeyDown()`和 `onKeyUp()`

上节讲到, 在 `KeyEvent()`的 `dispatch()`方法中会调用 `receiver` 的 `onKeyDown()`或者 `onKeyUp()`函数, 而 `receiver` 有可能是 `View` 对象也有可能是 `Activity` 对象。本节就先介绍当 `receiver` 是 `View` 对象的情况。

首先介绍 `onKeyDown()`。

该函数本身是可以被重载的, 如果没有重载, 则执行默认的逻辑。

① 判断按键消息是否是 `DPAD_CENTER` 或者 `KEYCODE_ENTER`, 两个按键的共性就是“确定”, 只有是确定键时, 才有默认的处理, 对于其他按键, 默认什么都不干, 直接返回 `false`。

② 判断当前 `View` 对象是否是 `ENABLE` 状态, 如果不是, 则直接返回 `false`。

③ 如果 `View` 是可以点击或者可以长按的, 则调用 `setPress()`将该视图的状态改为 `PRESSED`。

④ 如果该视图是可以长按的, 则调用 `postCheckForLongClick()`。注意, 这里的长按不是“生理长按”, 而是应用程序经常所说的那个“长按”。

`postCheckForLongClick()`函数内部调用 `postDealy()`函数发送一个异步延迟消息, 延迟的时间是 `ViewConfiguration.getLongPressTimeout()`, 也就是前面所说的那个 `500 ms`, 消息对应的 `Runnable` 对象是 `mPendingCheckForLongPress`。

如果过了 `500 ms`, 用户还没有释放该按键, 那么这个 `Runnable` 就会开始执行, 执行过程如下。

- (1) 如果该视图有长按监听者, 则回调其 `onLongClick()` 函数, 这就是程序员经常实现的长按回调。
- (2) 如果程序员没有自定义的回调处理, 那么系统就会调用 `showContextMenu()` 显示与该视图相关的情景菜单。
- (3) 如果消耗了该消息, 系统会向用户发出一个触觉反馈, 并将变量 `mHasPerformedLongPress` 置为 `true`, 如果没有消耗, 则返回 `false`。

下面接着来看 View 中的 `onKeyUp()` 默认处理逻辑。

- ① 如果该视图是不可点击的, 则直接返回 `false`。
- ② 如果可以点击, 并且处于 `press` 状态, 则调用 `setPress(false)` 参数为 `false` 将状态改变为非 `Pressed`。
- ③ 如果在释放按键时还没有到 500ms, 则需要从线程消息队列中删除上面添加的那个 `mPendingCheckForLongPress` 消息。判断是否超过 500ms 是通过变量 `mHasPerformedLongPress` 完成的, 因为当长按发生并执行后, 会把该变量置为 `true`。
- ④ 调用 `performClick()` 函数, 把该消息流当做点击处理, 该函数仅仅回调 `onClick()`。

### 13.3.6 Activity 中的 `onKeyDown()` 和 `onKeyUp()`

下面继续介绍当 `KeyEvent` 类中 `dispatch()` 执行时, 参数 `receiver` 为 `Activity` 时的消息处理。

首先来看 `onKeyDown()` 函数。

① 处理 “Back” 键。在 Eclair 版本之前, 当 “Back” 键按下时, 直接调用 `onBackPressed()`, 该函数的作用是退出当前 `Activity`, 当用户按下 “Back” 键不放时, 会连续退出 `Activity`, 这有时会引起误操作, 尤其是当按得比较慢时。因此, 在 Android 2.3 版本中, 并没有直接调用 `onBackPressed()`, 而是先调用 `event.startTracking()`, 也就是说, 当用户按住 “Back” 键不放时, 是不会退出当前 `Activity` 的。

② 判断 `mDefaultKeyMode`。在一般情况下该变量的值为 `DEFAULT_KEYS_DISABLE`, 所以会直接返回 `false`, 应用程序可以在 `Activity` 的 `onCreate()` 函数中调用 `setDefaultKeyMode()` 设置该属性值。你可能很少看到在任何 `Activity` 界面中直接按数字键后启动打电话程序, 实际上这完全是可能的, 属性值包含以下几种。

- `DEFAULT_KEYS_DISABLE`: 按键后什么都不干。
- `DEFAULT_KEYS_DIALER`: 按键后启动拨号程序。
- `DEFAULT_KEYS_SHORTCUT`: 按键后执行快捷键。
- `DEFAULT_KEYS_SEARCH_LOCAL`: 按键后启动本地搜索。
- `DEFAULT_KEYS_SEARCH_GLOBAL`: 按键后启动全局搜索。

③ 如果是 `SHORTCUT` 模式, 则调用 `Activity` 所包含的 `Window` 对象的 `performPanelShortcut()` 函数, 这与最先处理快捷键的方式相同。

④ 如果不是按下后的第一次 `DOWN` 消息, 或者该按键 `isSystem()`, 则返回 `false`。

⑤ 将按键转换为一个具体的字符, 因为不同的键盘界面会对应不同的字符。比如同样是 PC 键盘,

美式键盘和中文键盘有些按键会产生不同的字符，而这正是 `TextKeyListener.getInstance().onKeyDown()` 转换完成的。`getInstance()` 会返回一个 `TextKeyListener` 对象，该对象可以有不同实例，比如具体的实例在 `android.text.method.xxxListener` 包中，比如 `QwertyKeyListener`。

⑥ 获得转换后的字符后，根据当前的 `keyMode`，启动不同的 Activity，包括拨号程序、搜索等。

下面继续来看 Activity 中的 `onKeyUp()` 默认处理。在 Android 2.2 版本之前，该函数内部什么都没有做，而在 Android 2.2 版本之后，由于在 `onKeyDown()` 消息中对“Back”键仅执行了 `event.startTracking()`，因此，在 `onKeyUp()` 函数中必须对该“生理长按”予以“补偿”。最简单的情况就是应该在此时调用 `onBackPressed()`，但实际代码中还需要判断应用程序是否消耗掉了“Back”键的“生理长按”，如果消耗了，则该 `event` 的 `flag` 中会包含 `FLAG_CANCELED` 标识和 `FLAG_CANCELED_LONG_PRESSED` 标识，因此，当 `event` 包含这两个任何标志时，都不应该调用 `onBackPressed()`。对于其他按键消息，`onKeyUp()` 中没有任何默认处理。

### 13.3.7 PhoneWindow 内部消息派发过程

当按键消息在 View 树内部及 Activity 内部没有被处理时，就会调用到 `PhoneWindow` 中的 `onKeyDown()` 或者 `onKeyUp()` 消息，这是按键消息的最后一段默认旅程。

首先来看 `onKeyDown()` 的过程。

① 当是音量调节按键时，调用 `AudioManager` 类的相应方法进行音量调节。

② 当是音乐播放控制按键时，则调用 `MediaPlayer` 的相关方法进行音乐播放控制。处理时需要先判断当前是否正在打电话，如果是，则不要执行任何音乐播放控制。在目前的绝大多数 Android 设备中，都没有音乐播放控制按键。

③ 处理照相机按键。该步骤中首先判断是否处于锁屏状态，如果是，则什么都不做。接着调用 `startTracking()` 进入生理长按监测，并当下一次长按的 DOWN 消息产生时，才发送一个广播，广播的 Intent 中对应了照相机程序。该步骤说明，如果照相机按键按得太快的话，那么可能不会启动照相机程序。

④ 处理“Menu”键，这里仅仅是调用 `onKeyDownPanel()`，该函数内部会启动打开选项菜单的过程。

⑤ 处理拨号键。首先判断是否在锁屏状态，如果是，则直接返回 `false`。接着执行和以上几步相同的生理长按监测逻辑，并当发生长按消息时，启动语音拨号。这里请注意是语音拨号，如果用户没有长按，则会在 UP 消息中启动普通的电话拨号。

⑥ 处理“Search”键，这里同样执行了生理长按监测，如果是长按消息，则启动搜索程序，注意这和 UP 消息中“Search”键的处理不同，此处启动 Intent 内容如下：

```
Intent intent = new Intent(Intent.ACTION_SEARCH_LONG_PRESS);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

至此，`onKeyDown()` 消息就处理完毕了，下面来看 `onKeyUp()` 的处理过程。

① 首先调用 `DispatcherState` 对象的 `handleUpEvent()`，因为在 DOWN 消息中进行了生理长按监测，



此处需要对其内部状态做一个调整。

② 处理音量调节按键。大家可能觉得奇怪，不是在 DOWN 消息中已经处理了音量调节吗？的确是已经处理了，所以该步骤有点多余，所幸的是在调用 AudioManager 相关函数时，参数使用的是 ADJUST\_SAME，SAME 就是“相同”的意思，这会导致仅仅显示音量调节窗口而不调整具体音量值。

③ 处理“Menu”键。注意，这里调用的是 onKeyUpPanel()，而在 DOWN 消息中调用的是 onKeyDownPanel()。

④ 处理“Back”键。此时首先判断是否打开了菜单窗口，如果已经打开，则仅仅是关闭菜单窗口；如果当前打开的是二级菜单窗口，则调用 reopenMenu()重新打开第一级菜单窗口。

⑤ 处理音乐播放控制按键。注意和 DOWN 消息中有所不同，Intent 中的 EXTRA\_KEY\_EVENT 分别包含各自的 event。

⑥ 处理照相机按键。此处什么都没做，因为 DOWN 消息中发生长按时已经打开照相机程序了。

⑦ 处理拨号键。只有当 DOWN 消息中没有处理生理长按消息时才会启动拨号程序，然而只要发生生理长按，DOWN 消息就一定会处理。因此，要启动拨号程序就不能发生生理长按，也就是说用户必须快速按一下拨号键才能启动拨号程序。

⑧ 处理“Search”键。注意和 DOWN 消息的区别，DOWN 消息是在长按发生时才处理，其启动的搜索程序对应的 Intent 中 Action 字段内容是 ACTION\_SEARCH\_LONG\_PRESS，而 UP 消息中则是回调 Activity 中的 onSearchRequest()函数。

至此，onKeyUp()处理结束，一次按键消息到此就结束了旅程了。

### 13.4 按键消息在 WmS 中的派发过程

按键消息和触摸消息从流程上讲有一个重要的不同，当消息获取模块得到消息后，前者首先经过 WmS 中进行了一定的处理，然后才发送到客户端窗口，而后者则是直接把消息传递到客户端窗口中。本节就来分析按键消息在 WmS 中的处理过程，由于这段处理过程对应用程序是完全不可见的，我们不妨将其称为“消息在胎儿期的处理过程”。

在前面章节中曾经说过，当 InputDispatcher 获得输入消息后，如果消息是按键消息，则首先会回调 InputManager 类中的 interceptKeyBeforeDispatching()，如以下代码所示：

```

389     public boolean interceptKeyBeforeDispatching(InputChannel focus, int action,
390           int flags, int keyCode, int metaState, int repeatCount, int policyFlags) {
391         return mWindowManagerService.mInputMonitor.interceptKeyBeforeDispatching(focus,
392           action, flags, keyCode, metaState, repeatCount, policyFlags);
393     }

```

InputMonitor 中的相应函数内容如下：



```

5262 public boolean interceptKeyBeforeDispatching(InputChannel focus,
5263         int action, int flags, int keyCode, int metaState, int repeatCount,
5264         int policyFlags) {
5265     WindowState windowState = getWindowStateForInputChannel(focus);
5266     return mPolicy.interceptKeyBeforeDispatching(windowState, action, flags,
5267         keyCode, metaState, repeatCount, policyFlags);
5268 }

```

此处的 mPolicy 对象正是 PhoneWindowManager 类，也就是所谓的“胎儿期”。因此，接下来具体分析该类中是如何处理按键消息的，具体流程如图 13-4 所示。

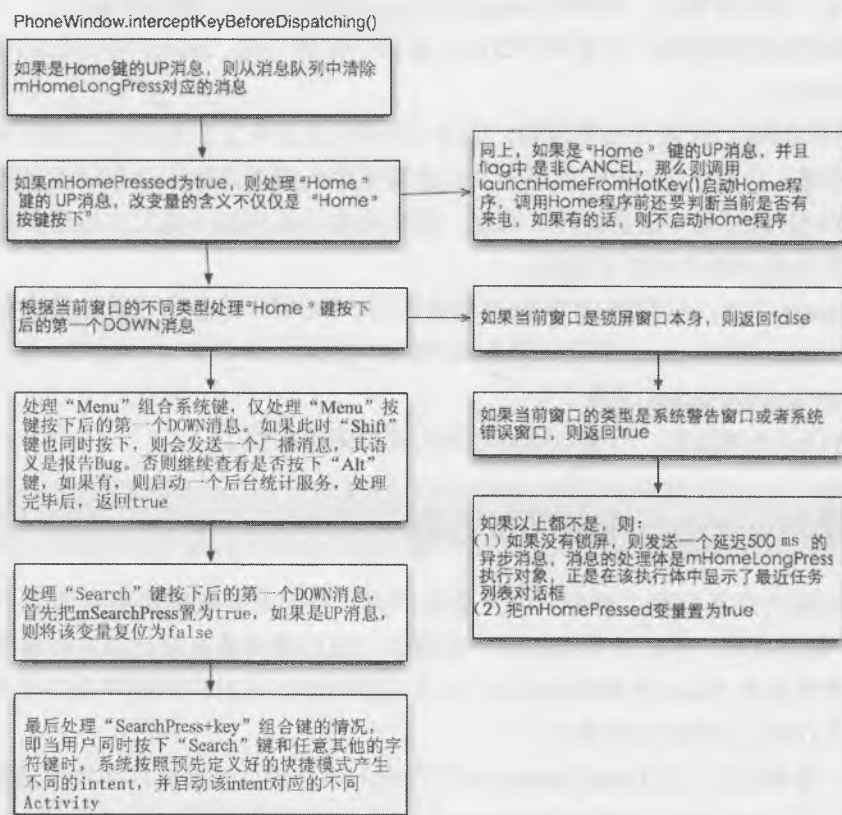


图 13-4 在 PhoneWindow 类中的消息预处理

读者可能会感到有点迷惑，前面几节中已经介绍了 DecorView 中对系统按键的默认处理，比如“Back”键返回、“Menu”键显示菜单窗口等，为什么此处又有对这些键的默认处理呢？总的来讲，PhoneWindowManager 中的默认处理都是针对整个系统的，而前面介绍的则是针对应用程序的。所谓“整个系统”指启动系统中特定的服务或者 Activity，并且不是单按某个系统按键产生的，而是同时按下系统键和其他按键，应用程序对于这些快捷键是完全没有兴趣的，下面就来看其具体过程。

- ① 如果是“Home”键的 UP 消息，则移除 mHomeLongPress 对应的消息，该变量是一个 Runnable

对象。这里需要注意,由于该过程是在应用程序截获消息之前发生的,因此,应用程序无法通过 `onKey()` 函数的内部实现改变该默认行为。

② 判断 `mHomePressed` 是否为 `true`, 处理“Home”键的 UP 消息。简单的讲, `mHomePressed` 变量是当“Home”键按下后第一个 DOWN 消息中被赋值为 `true`, 从而接下来发生 UP 消息时, 调用 `launchHomeFormHotKey()` 启动 Home 程序。然而事实并非这么简单, 如果当前窗口是一些特殊的窗口, `mHomePressed` 将不会被赋值, 从而之后的 UP 消息也就不会调用 `launchHomeFormHotKey()` 启动 Home 程序了。那么, 什么是特殊的窗口呢?

特殊的窗口有三个, 分别是锁屏窗口、系统警告窗口、系统错误窗口, 即如果当前窗口是这三种窗口时, 用户按一下“Home”键是不会启动 Home 程序的。

③ 处理“Home”键按下后的第一个 DOWN 消息。该步骤中处理的正是上面第二步中所说的特殊窗口, 除此之外, 在最后给 `mHomePressed` 赋值之前, 还启动了一个长按监测, 时限为 500ms。该长按消息对应的执行对象是 `mHomeLongPress`, 它是一个 `Runnable` 对象, 执行体中正是用户长按“Home”键所看到的最近任务列表窗口。由于该处理发生在应用程序处理之前, 因此, 应用程序无法修改“Home”键这种默认行为, 从而保证了系统按键操作的一致性。

④ 处理“Menu”组合系统键。注意这里并不是执行“Menu”键启动菜单窗口的操作, 而是处理当用户按“Shift+Menu”及“Alt+Menu”组合键的情况。前者会发送一个系统广播, 对应的 `Intent` 中 `Action` 字段的值为 `ACTION_BUG_REPORT`; 后者会启动一个 `Service`, 名称为 `com.android.server.LoadAverageService`, 该服务的作用是进行系统统计。

⑤ 处理“Search”键按下后的第一个 DOWN 消息, 其内部仅仅是给变量 `mSearchPress` 赋值为 `true`, 意思就是说, “Search”键已经按下了。如果是 UP 消息, 则复位该变量。

⑥ 处理普通按键, 前提是此时“Search”键已经按下, 即处理“Search+Key”组合键的情况。该段代码中的 `if` 语句中的片段代码如下:

```
1196         if (mSearchKeyPressed) {
1197             if (down && repeatCount == 0 && !keyguardOn) {
```

此处 `repeatCount == 0` 是指任意“Key”键按下后的第一个 DOWN 消息, 而不是指“Search”键, 以上条件的语义就是: “Search”键按下后, 还没有释放, 接着按了任意一个其他按键, 并且是按下后的第一个 DOWN 消息, 同时没有锁屏。当这个条件成立时, 则根据 `keyCode` 产生一个 `Intent`, 然后再调用 `startActivity()` 根据该 `Intent` 启动不同的 `Activity`。

其中根据 `keyCode` 产生 `Intent` 的功能是调用 `mShortcutManager.getIntent()` 函数实现的, 参数包含了 `keyCode`。 `mShortcutManager` 变量是一个 `ShortcutManager` 对象, 该变量是在 `PhoneWindowManager` 初始化时创建的。

至此, “胎儿期”的过程就结束了, 如果该函数返回 `true`, 那么应用程序中将不会感知到该消息, 如果返回 `false`, 则应用程序开始处理该消息。

## 13.5 触摸消息派发过程

触摸消息派发过程和按键消息相比有四点不同，在阅读本节内容时注意区别。

- 触摸消息是消息获取模块直接派发给应用程序的，而不像按键消息先要经过“胎儿期”。
- 触摸消息在处理时，需要根据触摸坐标计算该消息应该派发给哪个 View/ViewGroup，在按键消息处理中不存在该计算过程。
- 没有类似于“系统按键”的“系统触摸键”，应用程序可完全控制触摸行为。
- 子视图优先父视图处理消息，即首先是子视图处理该消息，只有当子视图消耗该消息时，父视图才有机会处理，这与按键消息的处理完全相反。按键消息派发流程中，ViewGroup 的 dispatchKeyEvent() 函数中，首先调用 super.dispatchKeyEvent()，即让该 ViewGroup 自身处理该按键消息，如果自身没有消耗该消息时，才调用子视图 mFocused 的 dispatchKeyEvent()。

源码中对于触摸消息相关函数命名，有时使用 pointer，有时却使用 motion 或者 touch，但其含义却是一致的，阅读时请稍加留意。

### 13.5.1 触摸消息总体派发过程

和按键派发类似，当消息获取模块通过 pipe 将消息传递到客户端，InputQueue 中的 next() 函数内部调用 nativePollOnce() 函数中会读取该消息。如果有消息，则回调 ViewRoot 内部的 mInputHandler 对象的 dispatchMotion() 函数，该函数仅仅是发起一个 DISPATCH\_POINTER 异步消息，消息的处理函数是 deliverPointerEvent()。执行完该函数后，调用 finishInputEvent() 向消息获取模块发送一个回执，以便其进行下一次消息派发，真正完成回执的代码是 native C++ 编写的。

下面就来介绍 deliverPointerEvent() 的具体过程，整体调用流程图参见附图 6。

① 进行物理像素到逻辑像素的转换。在一般情况下，物理屏幕的像素等于操作系统中定义的屏幕像素，不需要转换，只有当两者不同时才需要转换。比如对于 800×480 像素分辨率的屏幕，操作系统却将其定义成 480×320 像素，触摸消息本对应物理屏幕，所以需要转换到系统逻辑坐标。

② 如果是 DOWN 消息，调用 ensureTouchMode(true) 函数则进入触摸模式，与之相反的是“非触摸模式”，即按键模式。该函数会引起相关视图状态的变化，其内部执行过程见后面小节。

③ 将屏幕坐标转换到视图坐标。触摸消息本身的坐标位置是相对于屏幕左上角，对于 800×480 像素的屏幕，视图可以认为是没有边界的，它内部处理消息时所需要的坐标是相对于视图本身的，如图 13-5 所示。转换的方法很简单，变量 mCurScrollY 记录了该视图在屏幕坐标中的 Y 轴滚动，这里请注意，对于根视图而言，没有 X 轴的滚动，因为根视图的宽度已经被设置为屏幕本身的宽度。



图 13-5 屏幕坐标和视图坐标的关系

④ 调用 `mView.dispatchTouchEvent()` 将消息派发给根视图，该函数内部会继而将消息派发到整个 View 树。根视图有两种情况，对于 Activity 包含的窗口，根视图就是 PhoneWindow 中的 DecorView；对于非应用窗口，根视图只是一个普通的 ViewGroup。

⑤ 如果以上根视图及其所有子视图都没有消耗该消息，最后处理屏幕边界偏移。屏幕边界偏移在程序中用英文 `edge slop` 表示，它的作用是当用户正好触摸到屏幕边界时，系统自动对原始消息进行一定的偏移，然后在新的偏移后的位置上寻找是否有匹配的视图，如果有则将消息派发到该视图。为什么要有“屏幕偏移”呢？因为对于触摸屏而言，尤其是电容触摸屏，人类手指尖有一定的大小，当触摸到边界时，力量会被自动吸附到屏幕边界，所以，此处根据上下左右不同的边界对象消息原始位置进行一定的偏移。

以上即为触摸消息的总体派发过程，下面几个小节分别分析其中的几个重要步骤。

### 13.5.2 根视图内部消息派发过程

首先来看 `mView.dispatchTouchEvent()` 的派发过程。该函数是在 ViewRoot 中调用的，mView 的类型可能有两种情况，对于应用窗口而言，mView 是一个 PhoneWindow 中的 DecorView 类型；对于非应用窗口而言，mView 是一般的 ViewGroup 类型。

在 DecorView 中，首先判断是否存在 Callback 对象，它和按键消息派发时的 Callback 对象一样，就是 Activity 类。如果没有 Callback 对象，则直接调用 DecorView 基类 ViewGroup 中的 `dispatchTouchEvent()` 函数。

在 Activity 中, `dispatchTouchEvent()` 的过程如下。

- ① 如果是 `ACTION_DOWN` 消息, 则调用 `onUserInteraction()`。这与按键消息一样, 给应用程序一个机会, 以便在消息处理前做点什么, 该函数默认什么都不干。
- ② 调用所包含的 Window 对象的 `superDispatchTouchEvent()`。
- ③ 如果 Window 类没有消耗该消息, 则调用 `onTouchEvent()`, 该函数默认也是什么都不干, 仅仅是给应用程序一个处理消息的机会。

下面接着看 Window 类中的 `superDispatchTouchEvent()`。此时 Window 类的实现就是 `PhoneWindow` 类, 该函数继而调用 `mDecor` 的 `superDispatchTouchEvent()`, 而在 `DecorView` 的该函数中又调用 `super.dispatchTouchEvent()`, 即 `ViewGroup` 的 `dispatchTouchEvent` 函数。注意这里的调用过程, 一般的消息处理流程是当上一步没有消耗消息时才执行下一个处理逻辑, 而在根视图 `DecorView` 中, 则是当没有 `Callback` 时才调用 `ViewGroup` 的消息处理逻辑, 而不是当 `Callback` 没有消耗消息时才调用 `ViewGroup` 的消息处理逻辑, 原因就是 `Callback` 本身就会调用到 `ViewGroup` 的消息处理逻辑。

### 13.5.3 ViewGroup 内部消息派发过程

`ViewGroup` 内部的处理逻辑也采用递归方式, 但与按键处理的递归有所不同。触摸消息处理中首先会把消息派发给 `View` 树中最后一个子视图, 如果子视图没有消耗该消息, 才递归派发给其父视图, 而在按键消息处理时, 递归的过程正好相反。

- ① 将布局坐标转换为视图坐标。这两个坐标的概念如图 13-6 所示。



图 13-6 布局坐标和视图坐标

视图坐标中, 视图大小取决于视图本身包含多少内容, 不受物理屏幕大小限制。而布局坐标则是有限的, 它是指父视图给某子视图所分配的布局 (layout) 大小, 超过这个大小的区域将不能显示到父视图的区域中。转换的方法也很简单, 只需要使用 `getX()/getY()` 获取布局坐标, 然后再加上 `mScrollX/mScrollY` 即可。对于递归调用中第一次调用该函数时, `getX()` 的值实际上就是屏幕上的 X 轴

坐标, `getY()` 的值就是屏幕上 Y 轴的值减去状态栏的高度。

为什么要转换呢? 因为接下来要判断该坐标点落到了该 `ViewGroup` 中的哪个子视图中, 子视图的位置都是相对于该 `ViewGroup` 的视图坐标的。

② 处理 DOWN 消息, 其作用是判断该视图坐标落到了哪个子视图中。

(1) 首先判断该 `ViewGroup` 本身是否被禁止获取 TOUCH 消息, 如果没有禁止, 并且回调函数 `onInterceptTouchEvent()` 中没有消耗该消息, 则意味着该消息可以传递给子视图。如果子视图消耗了该 DOWN 消息, 则直接返回 `true`。

(2) 开始寻找子视图。调用 `child.getHitRect(frame)` 函数获取该子视图在父视图中的布局坐标, 即该 `ViewGroup` 为该 `child` 分配的位置是什么, 这个位置相对于该 `child` 来讲是布局坐标, 而相对于该 `ViewGroup` 来讲却是视图坐标, 参数 `frame` 是执行完毕后的位置输出矩形。得到位置后, 就可以调用 `frame.contains()` 方法判断该消息位置是否被包含到了该 `child` 中, 如果包含, 并且该 `child` 也是一个 `ViewGroup`, 则准备递归调用该 `child` 的 `dispatchTouchEvent()`, 在调用之前, 首先要把坐标重新转换到 `child` 的坐标系中。

以上过程听起来有些绕, 为了清晰明了, 该过程可表示为如图 13-7 所示。该图以一个例子来说明这种坐标转换过程, 图中空白区域不计尺寸, 黑色边框是一个 `ViewGroup`, 其中包含了五个 `child`, 每个方块都是一个 `child`, 宽为 60, 高为 20, 其中黑点为触摸位置, 对应位置上的 `child` 也是一个 `ViewGroup`, 内部包含了两个白色方框子 `View`。

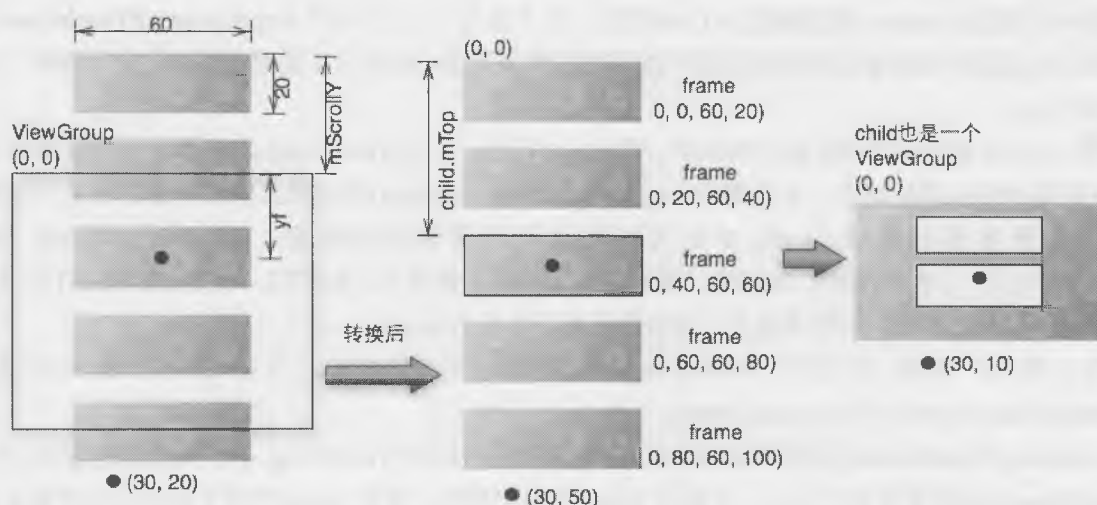


图 13-7 根据坐标位置寻找对应视图的过程

(3) 上步中完成了递归操作前的坐标转换工作, 接下来判断该 `child` 是否是 `ViewGroup` 类。如果是就递归调用到 `ViewGroup` 的 `dispatchTouchEvent()`, 重新从第一步开始执行; 如果 `child` 不是 `ViewGroup`, 而是一个 `View`, 则意味着递归调用的结束。



③ 如果是 UP 或者 CANCEL 消息, 则清除 `mGroupFlags` 中的 `FLAG_DISALLOW_INTERCEPT` 标识, 即允许该 `ViewGroup` 截获消息。换句话说, 常见的情况就是当用户释放手指, 下一次再按下时, 该 `ViewGroup` 本身可以重新截获消息, 而在按下还没有释放期间, `ViewGroup` 本身是不允许截获消息的。

④ 判断 `target` 变量是否为空。空代表了所有的子窗口没有消耗该消息, 所以该 `ViewGroup` 本身需要处理该消息。在第二步中, 如果匹配到某个 `child`, 并且该 `child` 消耗了消息后, 会将该 `child` 赋值给父视图中的 `mMotionTarget` 变量。在该步中, 首先要还原消息的原始位置, 因为在第二步中, 为了判断子视图是否包含该消息中的位置, 对位置进行了从布局坐标到视图坐标的转换, 而此时则需要把视图坐标重新转换为布局坐标, 因为接下来要调用 `super.dispatchTouchEvent()`, 即 `View` 类的该函数。`View` 类中处理该函数时, 需要布局坐标, 详情参见本章后续小节。转换好之后, 直接调用 `super.dispatchTouchEvent()`, 并返回其执行结果, 该函数内部仅仅是回调 `onTouchEvent()`, 调用之前先判断 `mPrivateFlags` 中是否包含 `CANCEL_NEXT_UP_EVENT` 标识, 该标识在一般情况下都不会存在, 如果存在, 则将消息的 `action` 类型改为 `ACTION_CANCEL`。

⑤ 处理 `target` 存在, 并且变量 `disallowIntercept` 为 `false`, 即允许截获, 在默认情况下 `ViewGroup` 都是允许截获消息的, 只有当该 `ViewGroup` 的子视图调用父视图的 `requestDisallowInterceptTouchEvent()` 函数时, 方可禁止父视图再次截获消息, 但每次 UP 消息或者 CANCEL 消息之后, 该 `ViewGroup` 又会重新截获消息。注意, 在本步中, 如果不允许截获消息, 那么也就不会调用 `onInterceptTouchEvent()` 函数了, 如果允许, 并且 `onInterceptTouchEvent()` 消耗了该消息, 才执行本步的操作。如果这种情况发生, 在代码中将消息的 `action` 类型修改为 `CANCEL`, 即“取消”, 然后调用 `target.dispatchTouchEvent()`, 从而使得目标视图内部能够据此取消之前可能存在的消息跟踪, 比如为了监测长按、特定手势等, 执行完毕后返回 `true`。

⑥ 在大多数情况下都会执行到该步, 即 `target` 存在, 并且 `ViewGroup` 本身不允许截获消息或者允许截获但是没有消耗消息, 于是调用 `target.dispatchTouchEvent()` 把该消息继续交给目标视图处理。在调用该函数前需要检查 `target` 中是否声明过要取消随后的消息, 即 `mPrivateFlags` 中包含 `CANCEL_NEXT_UP_EVENT`, 如果是, 则把消息 `action` 值修改为 `CANCEL`, 置空 `mMotionTarget` 变量, 因为 `target` 不想处理接下来的消息了, 那么就可以认为没有 `target` 了。

以上就是 Touch 消息在 `ViewGroup` 内部的递归和派发, 分析以上过程时注意区分 `onInterceptTouchEvent()` 和 `onTouchEvent()`。

`onInterceptTouchEvent()` 是在 `ViewGroup` 中定义的, 即只有 `ViewGroup` 的子类能够重载该方法。而 `onTouchEvent()` 函数有两个定义, 一个是在 `View` 类中定义的, 所有 `View` 类的子类都可以重载该方法, 包括 `ViewGroup`, 另一个是在 `Activity` 中定义的, 用户 `Activity` 可以重载该函数。`View` 系统的消息处理机制中, 会先执行视图内部的 `onTouchEvent`, 如果没有处理, 才会调用 `Activity` 中的 `onTouchEvent()`。

另外, 对于 `ViewGroup` 而言, 在一般情况下会先调用 `onInterceptTouchEvent()`, 只有当该函数没有消耗掉消息, 并且其包含的子视图也没有消耗掉该消息时, 才会执行该 `ViewGroup` 的 `onTouchEvent()`。而对于 `View` 而言, 没有 `onInterceptTouchEvent()` 被调用。但并不是所有的消息处理过程都是先调用



onInterceptTouchEvent(), 只有以下两种情况才会调用到 onInterceptTouchEvent()。

- 即在以上第 ② 步骤中, 当是 DOWN 消息, 并且 ViewGroup 允许截获消息时。
- 即在以上第 ⑤ 步骤中, 当 ViewGroup 中存在 target 对象, 并且允许截获消息时。

### 13.5.4 各种消息监测的基本实现方法

本来本节应该介绍消息从 ViewGroup 最终派发到 View 后, View 内部的处理流程, 但是, 在 View 内部处理中使用了一种跟踪监测机制, 其作用是能够产生长按等不同的消息, 因此, 本节首先来介绍这种监测的实现方法, 然后在下节中具体介绍 View 内部的消息派发流程。

在触摸消息中, 已经实现的监测包括三种, 一种叫做 pre-pressed, 另一种是 pressed, 最后一种是长按, 三者是按时间划分的, 如图 13-8 所示。

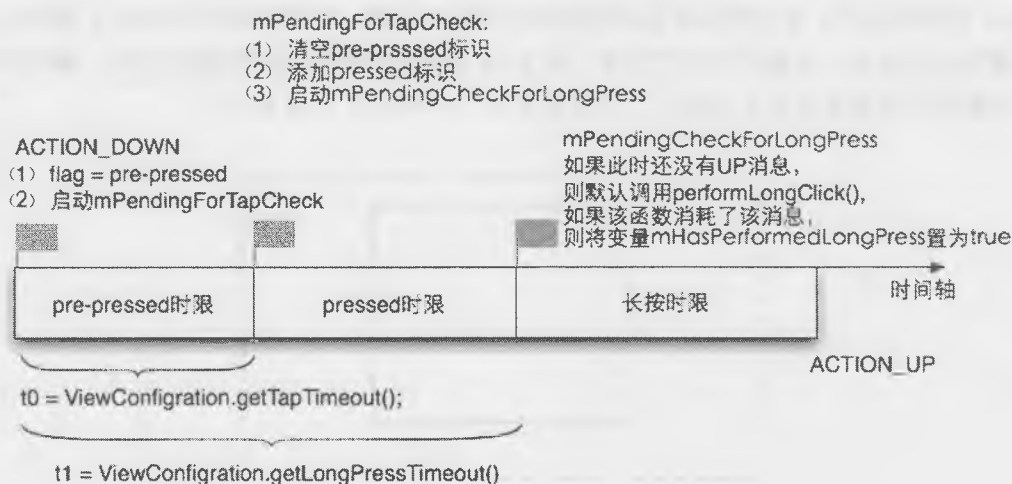


图 13-8 按键消息随时间的变化

实现监测的基本原理是利用 Handler 发送一个异步延迟消息, 在如图 13-8 所示中, 当发生 ACTION\_DOWN 消息时, 首先发送一个延迟为 t0 的异步消息, 如果在 t0 时间内, 用户释放了屏幕, 即 ACTION\_UP 消息在 t0 时间段中产生, 则本次触摸过程对应的是 pre-pressed 处理代码, 其语义是“用户轻触 (tap) 了一下”。否则, 系统会启动长按监测。如果用户在 t1 时间段内释放屏幕, 那么系统认为本次操作是一个“press”操作, 否则超过 t0 时间后释放屏幕就认为是一个长按消息, 超过 t0 时间后, 系统没有再进行监测。

以上监测过程是在 View 类的 onTouchEvent() 实现的, 如果应用程序重载了该函数, 并且没有调用 super.onTouchEvent(), 那么以上三种消息的回调就不会得到执行。换句话说, 如果重载了 onTouchEvent(), 那么该 View 对象的 onLongClick() 回调将不被执行。

### 13.5.5 View 内默认消息派发过程

本节分析触摸消息在 View 类内部的执行过程。

① 调用 `onFilterTouchEventForSecurity()` 处理窗口处于模糊显示状态下的消息。所谓的模糊显示是指，应用程序可以设置当前窗口为模糊状态，此时窗口内部的所有视图将显示为模糊效果。这样做的目的是为了隐藏窗口中的内容，对于其中各个视图而言，可以设置该视图的 `FILTER_TOUCHES_WHEN_OBSCURED` 标识，如存在该标识，则意味着用户希望不要处理该消息。

② 回调视图监听者的 `onTouch()` 函数，如果监听者消耗了该消息，则直接返回。

③ 调用 `onTouchEvent()`，应用程序可以重载该函数，但如果没有重载的话，该函数内部有默认的执行方式。默认的执行流程如下。

(1) 判断该视图是否为 `disable` 状态，如果是，什么都不干，返回 `true`，即消耗该消息。

(2) 处理消息代理 `TouchDelegate`。所谓的消息代理是指，可以给某个 View 指定一个消息处理代理，当 View 收到消息时，首先将该消息派发给其代理进行处理。如果代理内部消耗了该消息，则 View 不需要再进行任何处理；如果代理没有处理，则 View 继续按照默认的逻辑进行处理。源码中该类的注释中说，该类的目的是为了扩大点击区，意思很简单，举例如图 13-9 所示。

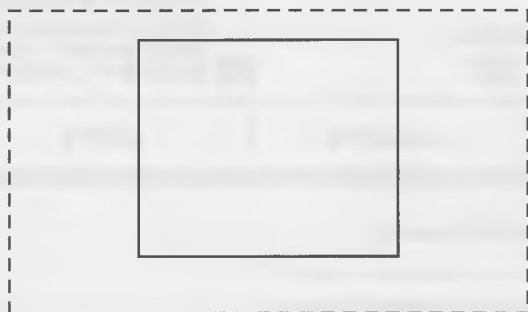


图 13-9 视图区域和可触摸区域的关系

黑实线框代表了视图本身在屏幕中的大小，在一般情况下，只有当用户点击到该区域时，该 View 对象才能处理 Touch 消息。然而有时却希望实际的点击区域能够大于 View 本身的区域，如虚线区所示，这种情况一般发生在该 View 本身周围没有其他视图时，为了提高点击的准确率故意设置。这个想法是好的，但是源码中却没有真正实现这个目的，因为要实现这个目的，必须在将消息匹配到对应窗口的判断中使用代理视图的大小，而不能使用视图本身的大小，否则，该视图将因为点击位置没有落到视图区而被忽略。然而源码中却没有经过这个判断，所以 `TouchDelegate` 形同虚设。

(3) 判断该视图是否可以点击的，如果不可点击，则直接返回 `false`，即不处理该消息。否则，真正开始执行触摸消息的默认处理逻辑，该逻辑中分别处理了 `ACTION_DOWN`、`MOVE` 和 `UP` 消息，具体过程如下。

① 在 ACTION\_DOWN 消息中, 给 mPrivateFlags 变量添加 PRESSED 标识, 并将变量 mHasPerformLongPress 置为 false, 然后启动 tap 监测, 即发送一个异步延迟消息, 延迟时间为 ViewConfiguration.getTapTimeout()。

② 对于触摸消息而言, 消息本身没有 repeat 的属性, 这与按键消息有所不同, 一次触摸消息只有一个 DOWN 消息, 接下来就是连续的 MOVE 消息, 并最终 UP 消息结束。因此, 本步骤中对 MOVE 消息进行处理。具体逻辑包括, 判断是否移动到了视图区以外, 如果是, 则删除 tap 或者 longPress 的监测, 并清除 mPrivateFlags 中的 PRESSED 标识, 然后调用 refreshDrawableState()刷新视图的状态, 这将导致对背景根据状态进行重绘。

这个处理逻辑有点问题。在一般的 GUI 系统中, 比如 HTC magic 手机, 或者苹果系统, 或者 Windows 系统, 当用户按下某个按钮时, 如果移动光标到按钮外, 此时按钮会从高亮恢复到普通状态, 但如果用户把光标再移回到该按钮上时, 按钮又会重新变为高亮。然而在目前的 Android 系统中, 代码的设计是, 当用户把光标再次移回到视图区时, 视图不会重新获得聚焦, 实际测试结果也是如此。

③处理 ACTION\_UP 消息, 代码中判断该 UP 消息是发生在前面所讲的哪一个监测时间段中, 并据此进行不同的处理。

- a. 查看是否发生在 pre-pressed 时间段内, 如果是, 则给变量 prepressed 赋值为 true。
- b. 无论是发生在 pre-pressed 区还是发生在 press 区, 都应该让该视图获取焦点, 前提是该视图在 Touch 模式下可以拥有焦点。
- c. 如果发生在 press 之后, 即长按, 则什么都不做, 因为长按的异步消息处理中已经处理了长按消息。如果是发生在长按之前, 则变量 mHasPerformedLongPress 为 false, 此时调用 removeLongPressCallback()移除还没有执行的长按监测消息。
- d. 判断变量 focusTaken 的局部变量是否为 false, 在一般情况下, 该变量都是 false, 因为一般情况下视图默认在 Touch 模式下是不能获得焦点的, 所以 requestFocus()会返回 false。当然, 只有当返回 false 的情况下, 该 UP 消息才能导致回调 performClick()函数, 否则, 当 focusTaken 为 true 时, 用户点击某个视图, 该视图仅仅是获得焦点, 必须再点击一次才能执行 performClick()函数。因为再点击一次时, 该视图已经获得了焦点, 从而不会调用 requestFocus(), 所以变量 focusTaken 为初始值 false。该步说明了 tap 和 press 动作的相同及区别, 相同的地方在于它们都会引起视图聚焦或者执行 performClick(), 区别在于 tap 不会改变视图的状态, 而 press 会把视图的状态改变为 PRESSED。从功能的角度来看, 两者没有本质区别, 仅仅是反映的 UI 效果有少许差别而已。

(5) 分别处理 tap 和 press 动作。如果是 press 动作, 则清除 PRESSED 标识, 并改变视图状态; 如果是 tap, 仅仅发送一个 UnSetPressedState 异步消息。

(6) 调用 removeTapCallback(), 关闭 tap 监测。

④ 处理 ACTION\_CANCEL 消息, 这里只需要清除 PRESSED 标识, 刷新视图状态, 然后再关闭 tap 监测即可。

至此, Touch 消息的一次处理过程就结束了。

## 13.6 导致 View 树重新遍历的时机

遍历 View 树意味着整个 View 需要重新对其包含的子视图分配大小并重绘。在一般情况下，导致重新遍历的原因主要有三个，一个是视图本身内部状态变化引起重绘，第二个是 View 树内部添加或者删除了 View，最后一个是 View 本身的大小及可见性发生变化。

本节中首先对引起重绘的各种状态进行介绍，然后介绍一些重要的函数，这些函数是最终会引起 View 树遍历的原因，至于添加、删除视图，以及 View 大小、可见性变化等操作都会间接调用到这些函数中。

### 13.6.1 状态的分类

在 View 视图中定义了多种和界面效果相关的状态，比如拥有焦点 (Focus)、按下 (Pressed) 等，不同的状态一般会显示不同的界面效果，有多种操作会引起这些状态的改变。Android 中应用程序是按照消息机制执行的，每次处理一个消息，如果该消息引起状态改变，则代码中仅仅做一些状态标识，然后发送一个异步消息，而不是立即重绘。然后在下一次消息处理中，根据保存的状态数据，绘制不同的界面效果。

视图中和显示效果相关的状态一共有 15 种，详细内容参见源码 `android.graphics.drawable.StateListDrawable` 类，本书仅介绍一些常用的状态，如表 13-3 所示。

表 13-3 视图状态的意义

状态名称	含 义
enable	是否使能，应用程序可以调用 <code>setEnabled()</code> 改变该状态
focused	是否被聚焦，一个窗口中只能有一个视图拥有焦点，一般由用户交互导致，不需要应用程序直接改变
pressed	是否被按下，一般由用户交互导致，不需要应用程序直接改变
selected	是否被选择，应用程序可以调用 <code>setSelected()</code> 改变该状态
window_focused	视图所在的窗口是否是当前交互窗口，由系统自动决定，应用程序不能改变该状态

其中 `selected` 和 `focused` 的区别有以下几点：

- 一个窗口中只能有一个视图获得焦点，当用户按“上/下”、“左/右”键时，获得焦点的视图会变得高亮起来，而一个窗口可以有多个视图处于 `selected` 状态。
- 按键消息最终会传递到 `focused` 视图中，而不是 `selected` 视图中。
- 当某个视图处于 `pressed` 状态时，如果将其 `selected` 状态设为 `false`，那么该视图的 `pressed` 状态就会被清空。
- `focused` 状态一般是由按键操作引起，`pressed` 状态是由触摸消息引起，`selected` 则完全是由应用程序主动调用 `setSelected()` 进行控制。

`focused` 和 `selected` 这两个状态经常令开发者感到迷惑,其根本原因是这两个概念直接来源于 PC 的 GUI 系统,而 Android 设备一般又多了一个触摸屏,从而导致输入的复杂。从输入类型的角度来看, `focused` 是针对按键的,即所有的按键消息将派发给 `focused` 视图; `pressed` 是针对鼠标或者触摸消息的,仅仅是当用户按下视图时给用户一个状态指示;而 `selected` 状态是为了程序处理之用,在一般情况下,具有 `selected` 状态的视图不会有什么界面上的变化,除非应用程序为该视图指定了 `selected` 时所使用的背景图。相反, iPhone 的状态就比较简单,因为它的 UI 系统完全抛弃了键盘的操作。

当视图重绘时,会根据当前不同的状态选择不同的绘制背景图,应用程序可以在 `res/xml` 目录下使用 `xml` 文件定义一组状态背景。以下代码为 Framework 中给一个 `Button` 视图设置背景图。

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_window_focused="false" android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_window_focused="false" android:state_enabled="false"
        android:drawable="@drawable/btn_default_normal_disable" />
    <item android:state_pressed="true"
        android:drawable="@drawable/btn_default_pressed" />
    <item android:state_focused="true" android:state_enabled="true"
        android:drawable="@drawable/btn_default_selected" />
    <item android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_focused="true"
        android:drawable="@drawable/btn_default_normal_disable_focused" />
    <item
        android:drawable="@drawable/btn_default_normal_disable" />
</selector>
```

该代码中定义了不同状态下所使用的背景图,每个 `<item>` 元素指定一种情况,并用 `android:state_xxx` 指定这种情况下包含的具体状态。`View` 类中使用 `StateListDrawable` 类保存这个 `drawable` 对象,并当绘制视图背景时,调用 `StateListDrawable` 的 `draw()` 方法完成具体的绘制,关于这个过程细节参加后面关于绘制的小节。

### 13.6.2 导致 View 树重新遍历的总体诱因图

下面介绍各种能引起 `View` 树重新遍历的操作,这些操作总的来讲可以分为三类。一类是导致视图大小发生变化;第二类是导致 `ViewGroup` 重新为子视图分配位置;第三类是视图显示情况发生变化需要重绘。这三类情况最后都直接或间接调用到三个函数,分别为 `invalidate()`、`requestLayout()` 及 `requestFocus()`,而这三个函数最终都会调用到 `ViewRoot` 中的 `scheduleTraversals()` 函数,该函数然后发起一个异步消息,消息处理中调用 `performTraversals()` 开始对整个 `View` 进行重新遍历,如图 13-10 所示。

能导致调用 `invalidate()` 函数的包含三种情况:当应用程序改变视图显示属性时,调用 `setVisibility()`;当改变视图 `Selected` 状态时,调用 `setSelected()`;当改变视图 `Enable` 状态时,调用 `setEnabled()` 函数。

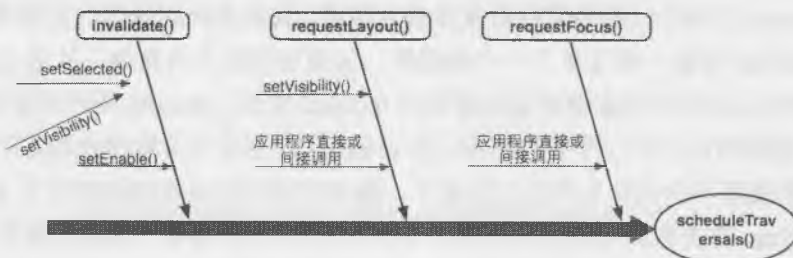


图 13-10 performTraversals()的总体执行过程

导致调用 requestLayout()函数的情况包含两种：当应用程序改变视图显示属性时，调用 setVisibility()，由于显示或者不显示将影响其他兄弟视图的位置，因此会调用到 requestLayout()；第二种是应用程序直接或间接调用该函数，间接调用是指应用程序调用了 View 类的其他函数，从而间接调用到 requestLayout()。

requestFocus()一般由程序直接调用，间接调用是指当用户按“上/下”、“左/右”键时，相关的处理逻辑会间接调用到该函数。

后面几个小节将介绍和状态相关的主要函数的内部执行过程，关于这些函数的调用时机请参照前面相关小节中消息派发过程的描述。

### 13.6.3 refreshDrawableList()

该函数的作用是根据状态标识，为视图赋予不同的 Drawable 对象，其执行过程如图 13-11 所示。

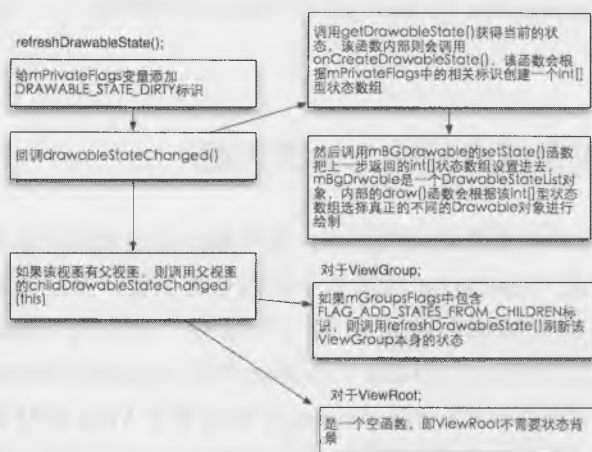


图 13-11 refreshDrawableList()的内部执行过程

① 给 mPrivateFlags 添加 DRAWABLE\_STATE\_DIRTY 标识，该标识仅在后面调用 getDrawableState() 函数中用于判断是否发生状态变化。

② 调用 drawableStateChanged()。该函数是一个 protected 类型，只有 Framework 中的 View 子类可以重载该函数，一般来讲，就是 ViewGroup 重载了该函数。ViewGroup 中重载该函数的作用仅仅是为了配合 FLAG\_ADD\_STATES\_FROM\_CHILDREN 标识，后面将会讲到该标识的作用。View 类内部，该函数的默认实现包括以下几项。

(1) 调用 getDrawableState() 获得视图的当前状态，然后再调用 onCreateDrawableState() 将这些状态转换为一个 int[] 型数组，这个数组的内部格式是预先定义好的，DrawableStateList 类可以识别该 int[] 数组。最后再将第一步设置的标识进行清除。

(2) mBGDrawable 变量是该视图的背景图，它包含一个 setState() 函数，函数的参数正是上一步获得的 int[] 型数组，该函数内部会根据该 int[] 型数组为 mBGDrawable 找到真正的 Drawable 对象。

③ 如果该视图有父视图，则调用父视图的 childDrawableStateChanged()。父视图要么是一个 ViewGroup 类，要么是一个 ViewRoot 类。

对于 ViewGroup，如果该 ViewGroup 中的 mGroupFlags 中包含 FLAG\_ADD\_STATES\_FROM\_CHILDREN 标识，则意味着该 ViewGroup 的背景图也可以随着其子视图的状态而变化，于是调用 refreshDrawableState() 刷新该 ViewGroup 自身的背景图。

应用程序可以调用 ViewGroup 的 setAddStatesFromChildren() 函数将该 ViewGroup 的背景图设置为和子视图的背景图同步，即，当 ViewGroup 对象调用 refreshDrawableState 函数时，会调用 ViewGroup 中的 onCreateDrawableState()。该函数中首先判断是否设置了 FLAG\_ADD\_STATES\_FROM\_CHILDREN，如果没有该标识，则直接调用 View 类的该方法，而如果存在该标识，则把所有子视图的标识合并成一个，并作为该 ViewGroup 的背景 int[] 型数组。比如当该 ViewGroup 有两个子视图时，只要有一个子视图的标识包含 PRESSED 标识，则该 ViewGroup 就包含 PRESSED 状态。如以下代码所示：

```

3380 protected int[] onCreateDrawableState(int extraSpace) {
3381     if ((mGroupFlags & FLAG_ADD_STATES_FROM_CHILDREN) == 0) {
3382         return super.onCreateDrawableState(extraSpace);
3383     }
3384
3385     int need = 0;
3386     int n = getChildCount();
3387     for (int i = 0; i < n; i++) {
3388         int[] childState = getChildAt(i).getDrawableState();
3389
3390         if (childState != null) {
3391             need += childState.length;
3392         }
3393     }
3394
3395     int[] state = super.onCreateDrawableState(extraSpace + need);
3396
3397     for (int i = 0; i < n; i++) {
3398         int[] childState = getChildAt(i).getDrawableState();
3399
3400         if (childState != null) {
3401             state = mergeDrawableStates(state, childState);
3402         }
3403     }
3404
3405     return state;
3406 }

```



该设计的目的是为了使 ViewGroup 的背景状态图和子视图的背景状态图同步，比如，当用户按下 ViewGroup 中的一个 Button 视图时，整个 ViewGroup 的背景也会高亮。

### 13.6.4 onFocusChanged()

该函数的作用是处理一些由于焦点变化而导致的其他状态变化的逻辑，其过程如图 13-12 所示。

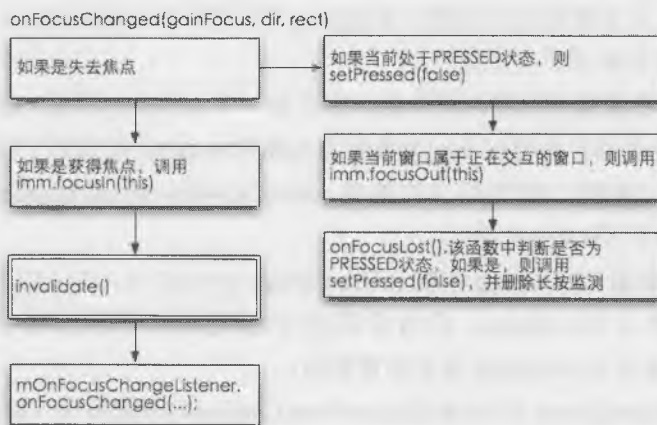


图 13-12 onFocusChanged()内部执行过程

#### ① 判断是获得了焦点还是失去焦点。如果是失去焦点：

(1) 如果当前处于 PRESSED 状态，则必须清楚 PRESSED 状态，即调用 setPressed(false)，注意参数为 false。这种情况似乎很难发生，因为当用户正在按下某个按钮时又怎么会失去焦点呢？所以，这段逻辑有点多余。

(2) 如果当前窗口正在和输入法交互，则调用 imm.focusOut(this)，该函数内部会隐藏输入法窗口（如果输入法窗口显示），并断开输入法服务和当前窗口的连接。比如，当前用户正在一个 EditText 中输入内容，输入完毕后，按方向“下”键，这会导致该 EditText 失去焦点，并间接调用到 onFocusChanged() 函数。此时用户会发现，如果下一个焦点窗口不是可编辑的，输入法窗口就会自动隐藏。注意，该步中所谓的“当前窗口正在和输入法交互”，并不是指有某个视图正在和输入法交互，因为代码中是查询变量 mAttachInfo 的 mHasWindowFocus，如以下代码所示：

```

2730     if (imm != null && mAttachInfo != null
2731         && mAttachInfo.mHasWindowFocus) {
2732         imm.focusOut(this);
2733     }
  
```

mHasWindowFocus 变量的含义是当前窗口是否是“前台窗口”，即正在和用户交互的窗口。因此，以上代码中条件成立对应的语境是：用户正在和该视图所在的窗口交互，并且该窗口中包含可编辑的视

图，至于输入法窗口是否显示，则不一定。关于输入相关逻辑参照本书第17章。

(3) 调用 `onFocusOut()` 函数，该函数类型为 `protected`，默认的实现中会判断是否为 `PRESSED` 状态。如果是，则调用 `setPressed(false)` 清除该状态，并删除长按监测。

② 如果该视图是获得焦点，则调用 `imm` 的 `focusIn(this)`。该函数中，如果该视图是可编辑的，将会启动输入法，并显示输入法窗口。

③ 调用 `invalidate()`，该函数将会导致对该视图的重绘，因为在一般情况下焦点改变后，会引起视图背景图的变化。

④ 回调 `mFocusChangeListener` 的 `onFocusChanged()` 方法，应用程序可以实现该回调接口，以便进行其他操作。

### 13.6.5 ensureTouchMode()

这个函数的命名不够准确，从该函数内部分析来看，其作用是在 `Touch` 和非 `Touch` 直接切换时对视图的焦点状态进行处理。比如当用户按“上/下”、“左/右”键后，又直接在屏幕上点击一下，这会导致从非 `Touch` 到 `Touch` 模式的转换，原来有焦点的视图将会失去焦点，相反的过程亦如此。因此，该函数的名称可以叫做 `switchTouchMode()`，其过程如图 13-13 所示。

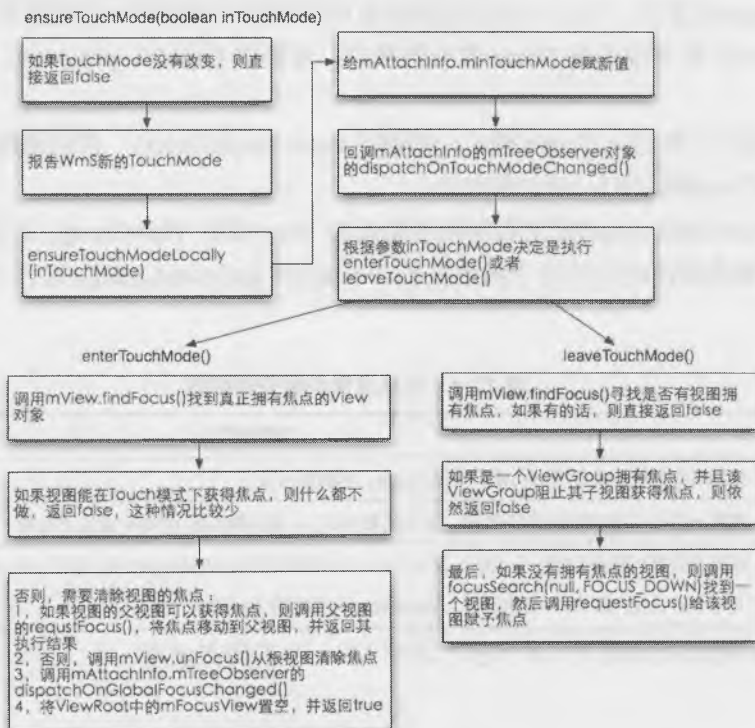


图 13-13 `ensureTouchMode()` 的内部执行过程

① 判断参数传入的 Touch 模式是否和当前的 Touch 模式相同，如果相同，什么都不用做，直接返回 false。该函数的返回值代表是否做了改变，而不是改变后的 Touch 模式。当前模式值保存在 mAttachInfo.mInTouchMode 变量中，这是一个 boolean 变量。

② 如果需要改变，则首先报告 WmS 当前 Touch 模式发生变化，因为 WmS 在进行客户窗口布局时，需要根据客户窗口的 Touch 模式进行不同的处理，详情参照本书第 14 章介绍。代码中调用 sWindowSession.setInTouchMode() 函数进行报告。

③ 调用 ensureTouchModeLocally 具体处理变化后的状态。

(1) 首先给 mAttachInfo.mInTouchMode 赋新值。

(2) 每个 View 中都包含一个 mAttachInfo 对象，该对象来源于 ViewRoot 中的 mAttachInfo，该对象内部又有一个 mTreeObserver 变量，它是一个 ViewTreeObserver 对象，该步中调用该对象的 dispatchOnTouchModeChanged() 函数。

对应用程序而言，可以调用 View 类的 getTreeObserver() 函数获取该 View 所在窗口的 ViewTreeObserver 对象，同一个窗口中不同 View 对象的该方法返回的是同一个对象，即一个窗口中只有一个该对象。ViewTreeObserver 内部有一个列表容器，程序员可以调用该类的 addOnGlobalFocusChangeListener() 向该列表中添加一个接口元素。本步中调用 dispatchOnTouchModeChanged() 实际上是循环调用该列表容器中多个接口对象的 onGlobalFocusChanged() 方法，而这个方法是程序员可以自定义实现的，以便处理 Focus 改变的情况。

ViewTreeObserver 类中除了有 Focus 改变的接口，还提供 Layout、pre-draw、scroll 等不同的接口，详情参照源码。

(3) 如果该函数是要进入 Touch 模式，则调用 enterTouchMode()，否则调用 leaveTouchMode()。

④ 执行 enterTouchMode()，该过程如下。

(1) 调用 mView.findFocus() 找到真正拥有焦点的 View 或者 ViewGroup，这里 mView 正是根视图。与 findFocus() 函数容易混淆的另外两个函数是 hasFocus() 和 getFocusChild()，这三个函数的区别如表 13-4 所示。

表 13-4 与焦点操作相关的函数

focus 相关的函数名称	函数的意义
mView.findFocus()	找到真正拥有焦点的 View 或者 ViewGroup，本例返回 A
mView.getFocusChild()	找到 mView 中包含焦点的子视图，本例返回 VG_1，该函数返回 mView 直接子视图
hasFocus()	判断是否拥有焦点，本例 A、VG_1 及 mView 三个视图调用 hasFocus() 时都会返回 true，而 B、VG_2 则都会返回 false。 其中 hasFocus() 是 View 类的函数，ViewGroup 中重载了该函数

此时假设 View 树结构如图 13-14 所示，其中 View A 正拥有焦点。

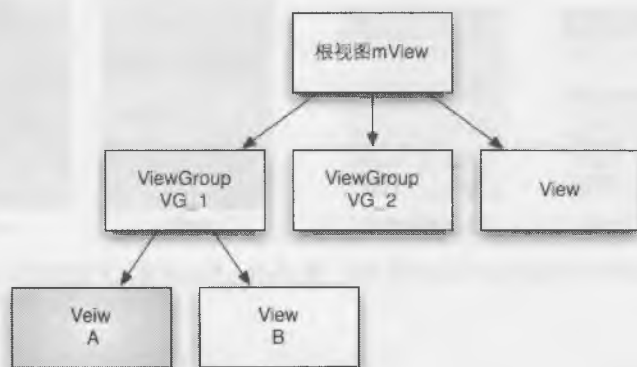


图 13-14 视图焦点和父视图之间的关系

(2) 如果存在焦点视图，并且该视图在 Touch 模式下也能够聚焦（高亮），那么直接返回 false，这种情况比较少。应用程序可以调用 `setFocusableInTouchMode()` 改变该属性，在默认情况下，视图在 Touch 模式下是不能拥有焦点的。

(3) 大多数情况下，Touch 模式下不能拥有焦点，因此，需要清除焦点。不过，该步骤中首先判断该视图的父视图是否可以在 Touch 模式下拥有焦点，如果可以的话，则调用父视图的 `requestFocus()` 并返回其结果，如果不可以，才开始真正清除焦点。

① 用 `mView.unFocus()`，该函数将从根上清除所有子视图中的焦点。

② 调用 `ViewTreeObserver` 的 `dispatchOnFocusChanged()`，以便应用程序可以对此执行相关操作，注意这里是 Focus 回调，上面是 Touch 回调。

③ `mFocusView` 变量置空，并返回 true，返回值代表是否做了改变。

⑤ 如果是离开 Touch 模式，则调用 `leaveTouchMode()`，过程如下。

(1) 同样，首先要去找到真正拥有焦点的视图，如果找到，并且该视图就是一个 View 对象，就不用重新给该视图赋予焦点了，可以直接返回 false。如果该视图是一个 ViewGroup 对象，并且该 ViewGroup 对象阻止其子视图获得焦点，那么也会直接返回 false。比如，ListView 就是可以获得焦点的 ViewGroup，并且 ListView 不会直接把焦点传递给其包含的 item，而是使用特别的逻辑让 item 获得焦点。

(2) 如果没有找到拥有焦点的视图，则调用 `focusSearch(null, FOCUS_DOWN)` 找到一个能够拥有焦点的视图，并调用其 `requestFocus()` 使该视图获得焦点。注意该函数的参数，null 代表当前拥有焦点的视图，即当前没有视图拥有焦点，DOWN 代表往下方找，这就是为什么当用户从按键模式切换到 Touch 模式，然后再进入按键模式时，上一个拥有焦点的视图不能再次获得焦点，而是第一个视图获得焦点的原因，如图 13-15 所示。而对于 ListView 而言，上一个获得焦点的视图，当重新切换到按键模式后依然能够获得焦点，因为 ListView 使用了特别的逻辑为所包含的 item 赋予焦点，如图 13-16 所示。



图 13-15 在 Touch 模式和按键模式切换时的焦点视图变化 图 13-16 Touch 模式和按键模式切换时的 ListView 内焦点变化

### 13.6.6 setVisibility()

该函数用于改变视图的可视状态，可视状态包括 GONE、VISIBLE、INVISIBLE 三种。该函数内部很简单，首先调用 setFlags()，然后调用 mBGDrawable.setVisible() 函数改变该视图背景图的显示状态。

setFlags() 函数在 View 类中被广泛使用，setEnabled()、setClickable() 等很多函数都调用到该函数。在 View 中使用 mViewFlags 和 mPrivateFlags 变量保存大多数属性，其中 mViewFlags 保存和视图状态相关的属性，mPrivateFlags 保存和内部逻辑相关的属性。这两个变量都是使用标识位(bit)保存不同的状态，这种保存方式在嵌入式系统设计中被广泛使用。setFlags() 函数内部使用了一种异或程序技巧进行不同状态的判断，如图 13-17 所示。

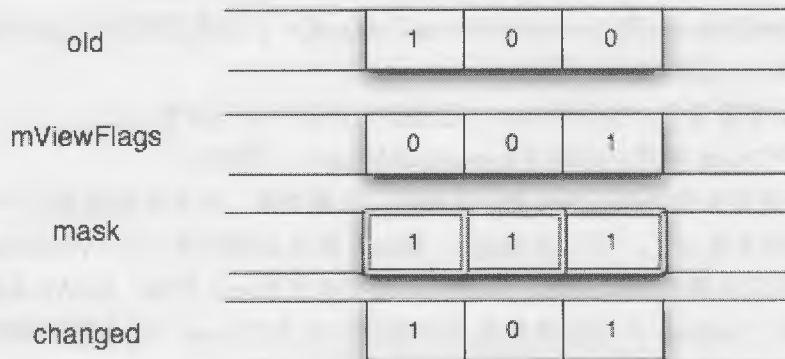


图 13-17 mViewFlags 中处理逻辑示意

其中 old 变量保存的是 mViewFlags 在改变之前的值，mask 对应了不同标识位的位置，在图 13-17 中，假设某个标识使用三个位保存状态，那么，对 old 和 mViewFlags 进行异或运算后，得到一个新的变量 changed，该变量中凡是位值为 1 就意味着该位上保存的状态发生了变化。

同时，在采用如图 13-18 所示的逻辑运算后，对应的语义也是很明确的。

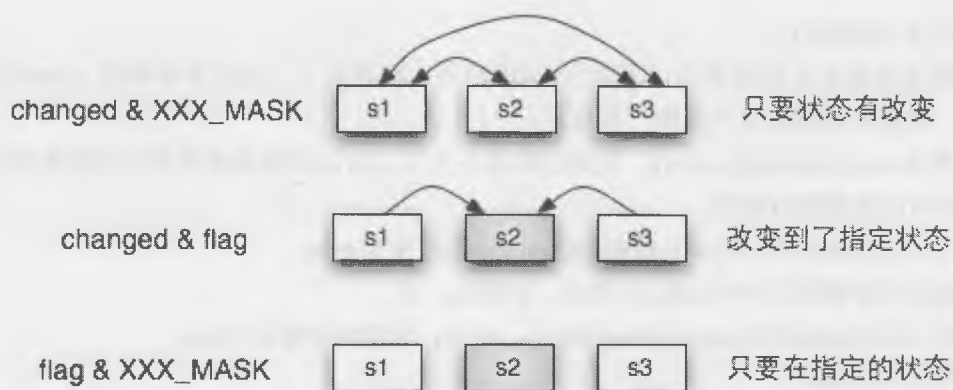


图 13-18 代码逻辑对应的语义示意

下面具体分析 setFlags()函数的执行过程，其流程参见附图 7。

① 判断当前状态是否一样要指定的状态，如果一样，则意味着不需要改变，直接返回。

② 判断改变后是否会引起 FOCUSABLE 状态改变，如果发生改变，则分别处理以下情况。

(1) 如果是从 focus 变化到 un-focus，则需要调用 unfocus()函数清除当前视图的焦点。

(2) 如果是从不可聚焦变化到可以聚焦，并且当前视图还没有拥有焦点，那么调用 mParent 的 focusableViewAvailable(this)，通知父视图 mParent 有了新的子视图可以获得焦点，参数 this 代表该视图本身。这里仅仅是通知父视图，至于父视图是否要把焦点分配给该视图则不一定。

③ 判断是否 VISIBLE 状态有改变，并且变化到了 VISIBLE 状态，源码中的判断逻辑有点啰嗦。如果是，则执行以下操作。

(1) 为该视图添加 DRAWN 标识，该标识将导致下一个 View 树重绘时绘制该视图，因为它已经是显示状态了。

(2) 调用 needGlobalAttributesUpdate()，该函数将给 mAttachInfo 对象中的 mRecomputeGlobal Attributes 变量赋值为 true。该变量的含义实际上是指“需要重新计算 View 树中视图的位置”，因为该视图显示出来后一般会导致其他视图在界面上的位置发生变化，因此 View 系统需要重新计算位置。

(3) 通知 mParent 有了新的子视图，这里同样是调用 mParent 的 focusableViewAvailable(this)。注意本小步执行的条件是 mBottom>mTop && mRight>mLeft，即该视图的大小不能为 0。也就是说，如果程序员添加了一个视图，尽管该视图状态是可视的，并且可以获得焦点，但是如果其大小为 0，则依然不能真正获取焦点。

④ 判断是否“改变到了”GONE 状态，注意所使用的判断条件，正如图 13-18 所指出的位运算。如果是，则执行：

(1) 调用 needGlobalAttributesUpdate(true)函数，打一个标识，以便 View 树重绘前重新计算每个视图的位置。因为它不见了，其他视图的位置将受到影响。

(2) 调用 requestLayout()，请求 View 树进行重新布局。

(3) 调用 `invalidate()`。

(4) 判断该视图是否拥有焦点，因为它(GONE)马上就不见了，因此需要调用 `cleanFocus()`释放该视图的焦点，以便其他视图能“接替”焦点。

(5) 调用 `destroyDrawingCache()`，因为它就要不见了，所以调用该函数释放该视图所使用的缓存，当然并不是所有的视图都有缓存。

(6) 将 `mAttachInfo` 的 `mViewVisibilityChanged` 变量置为 `true`。

⑤ 判断是否改变到了 `INVISIBLE` 状态，如果是，则：

(1) 调用 `needGlobalAttributesUpdate(false)`，注意，这里的参数是 `false`。

(2) 调用 `invalidate()`。

(3) 同样，如果该视图拥有焦点，则需要清除其焦点，调用 `clearFocus()`。

(4) 将 `mAttachInfo` 对象中的 `mViewVisibilityChanged` 置为 `true`。

看到这里，读者们会发现，其中第④步和第⑤步中都调用了 `invalidate()`请求 View 树进行重绘，而第③步中却没有，难道状态改变到 `VISIBLE` 不需要界面重绘吗？当然不是，事实上，第③步中给 `mPrivateFlags` 变量中添加了 `DRAWN` 标识，而该标识在随后的处理中将导致调用 `invalidate()`。

另外，大家还可以发现，在变化到 `GONE` 状态和 `INVISIBLE` 状态中，前者调用了 `requestLayout()`，而后者却没有调用 `requestLayout()`。这是因为从 `VISIBLE` 变化到 `INVISIBLE` 不需要重新进行布局，重绘即可。如果是从 `GONE` 变化到 `INVISIBLE` 就需要重新布局，只是这段代码中所使用的位操作不够标准，导致从 `GONE` 变化到 `INVISIBLE` 时，同时会引起 `GONE` 状态变化，所以，`GONE` 条件中的代码也会执行，即依然会调用 `requestLayout()`。

⑥ 如果 `VISIBILITY` 有改变，则调用 `dispatchVisibilityChanged()`，该函数会回调 `onVisibilityChanged()`，应用程序可以重载该函数以便执行其他操作。执行完该函数后，还会执行一些稀疏代码处理，此处略。

⑦ 判断 `mPrivateFlags` 中是否包含 `DRAWN` 标识。如果有，则分别调用 `requestLayout()`和 `invalidate()`请求 View 树进行重新布局并且重绘。

### 13.6.7 setEnable()

`ENABLE` 状态仅仅是内部的一个逻辑，不会引起重新布局，仅仅是引起视图重绘，其流程如图 13-19 所示。



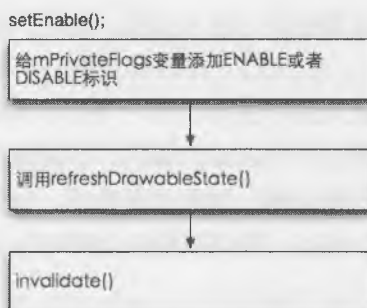


图 13-19 setEnabled()内部主要流程

- ① 给 `mPrivateFlags` 变量添加 `ENABLE` 或者 `DISABLE` 标识，这由 `setEnabled()` 的参数决定。
- ② 调用 `refreshDrawableState()` 重新获取背景图。
- ③ 调用 `invalidate()` 请求 View 树重绘。

### 13.6.8 setSelected()

`SELECTED` 的逻辑和 `ENABLE` 非常类似，只是名称不同而已，其流程自然也与 `setEnabled()` 函数流程相似，如图 13-20 所示。

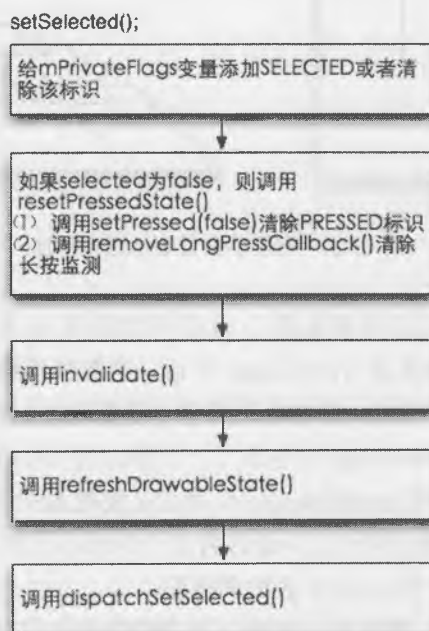


图 13-20 setSelected()内部主要流程

① 给 `mPrivateFlags` 变量添加或者删除 `SELECTED` 标识。这与 `setEnabled()` 的第一步稍有不同, `ENABLE` 状态分别由两个标识位区分, 而 `SELECTED` 却只有一个标识位标识, 从这点上看, 这个设计有点混乱, 事实上 `ENABLE` 和 `SELECTED` 状态都只需一个标识位即可。

② 如果 `selected` 为 `false`, 则调用 `resetPressedState()`。该函数内部仅仅是清除 `PRESSED` 标识, 并调用 `removeLongPressCallback()` 清除长按监测。这段代码表明, 当视图正处于 `PRESSED` 状态时, 程序员主动又给该视图添加非 `SELECTED` 状态, 则该视图将会同时失去 `PRESSED` 状态, 并忽略后续的长按消息处理。

③ 调用 `invalidate()` 请求 `View` 树重绘。

④ 调用 `refreshDrawableState()` 重新获取视图背景图, 该步与第 ③ 步执行可以不分先后。

⑤ 调用 `dispatchSetSelected()`, 应用程序可以重载该函数以便此刻进行其他操作。

### 13.6.9 invalidate()

该函数的作用是请求 `View` 树进行重绘, 当应用程序需要重绘某个视图时, 可以调用该函数。视图及其父视图在界面上是分层先后显示的, 如图 13-21 所示。

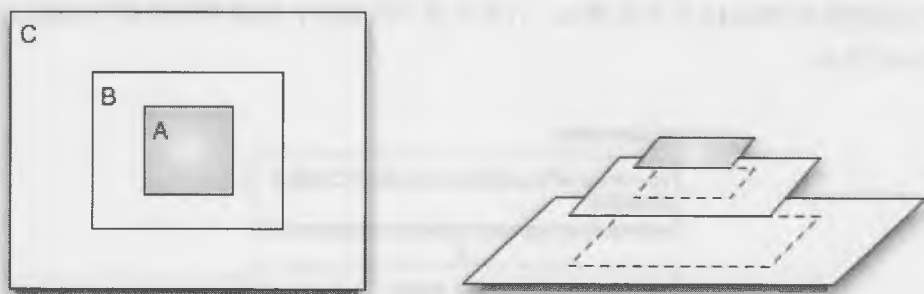


图 13-21 视图的分层显示逻辑示意

绘制流程中, 首先绘制最底层的根视图, 然后再绘制其包含的子视图。子视图或者是一个 `ViewGroup`, 或者是一个 `View`, 如果是 `ViewGroup` 的话, 则继续再绘制 `ViewGroup` 内部的子视图, 绘制过程一般并不会对所有视图进行重绘, 而仅绘制那些“需要重绘”的视图。那么, 什么是“需要绘制”的视图呢? `View` 类内部变量 `mPrivateFlags` 中包含了一个标志位 `DRAWN`, 当该视图需要绘制时, 就给 `mPrivateFlags` 中添加该标识位。函数 `invalidate()` 的作用就是要根据所有视图中的该标志位计算具体哪个区域需要重绘, 这个区域将用一个矩形标识, 并最终将这个矩形存放到 `ViewRoot` 类中的 `mDirty` 变量中, 之后的重绘过程将重绘所有包含在该 `mDirty` 区中的视图。

对于图 13-21 中所包含的 `View` 树而言, 如果 A 视图没有阿尔法通道, 即完全不透明, 那么理论上讲, 如果只是 A 视图需要重绘, 则不会导致 A 视图所在的父视图也进行重绘, 只有当 A 视图隐藏时,

再通知父视图 B 绘制之前 A 所占的区域即可。然而，对于包含阿尔法通道的视图系统而言，屏幕的效果是由 A 和 B 及底层所有父视图共同产生，因此，每次当 A 视图需要重绘时，其底层的父视图 B 及所有父视图都需要重绘该区域。所以，当 A 需要重绘时，必须通知父视图也绘制该区域。

这就是 `invalidate()` 函数的内部思想，下面来分析该函数的具体执行流程（参照附图 7）。

首先，查看视图中变量 `mPrivateFlags` 是否有 `DRAWN` 标识，并且视图的大小是否为 0。如果不是，则不需要重绘，直接返回；如果需要重绘，则先将需要重绘的区域保存到临时变量 `dirty` 中，这是一个矩形变量，然后调用父视图 `mParent` 的 `invalidateChild(this, dirty)` 函数，通知父视图该区域需要重绘，参数 `this` 代表该视图对象本身，`dirty` 的大小默认就是该视图的大小。

① 定义一个 `int[2]` 型局部变量 `location`，保存子视图左上角坐标，该坐标是相对于父视图的。这里仅仅是定义，该变量引用了 `mAttachInfo` 的 `mInvalidateChildLocation`。

② 判断子视图当前是否正在进行动画，如果正在进行动画，子视图的 `mPrivateFlags` 会包含 `DRAW_ANIMATION`。

③ 判断子视图是否是不透明的。不透明的条件有三，首先该视图的 `isOpaque()` 必须返回 `true`；其次是当前没有进行动画；最后是 `getAnimation()` 不为空。判断完毕后，将结果保存到变量 `isOpaque` 中。

④ 根据该 `child` 的以上判断结果，为其父视图添加不同的属性，换句话说，子视图的属性会改变父视图的属性。View 系统在后面的绘制过程中需要了解到这种关系，以便进行不同方式的处理。

(1) 首先，如果子视图正在进行动画，那么需要父视图也添加动画表示，而父视图要么是 `ViewGroup` 对象，要么是 `ViewRoot` 对象。对于 `ViewGroup`，给 `mPrivateFlags` 变量中添加 `DRAW_ANIMATION` 标识；而对于 `ViewRoot`，则给其内部变量 `mIsAnimating` 赋值为 `true`。

(2) 设置 `dirty` 标识。如果子视图是不透明的，则父视图的 `dirty` 标识是 `DIRTY_OPAQUE`，否则就是 `DIRTY`。这两个标识含义的共同点是都需要重绘，区别在于如果是 `DIRTY_OPAQUE`，父视图可以暂时不重绘，因为子视图是不透明的。

(3) 调用 `parent.invalidateChildInParent()` 函数。这里 `parent` 如果写成 `view` 大家可能更容易理解，实际上 `parent` 变量和 `view` 变量在这里是完全相同的，该函数的返回值是该视图的父视图。返回值的类型有两种情况，一种是 `ViewRoot`，一种是 `ViewGroup`。由于第 ④ 步代码使用 `do{}while()` 语句，当调用 `invalidateChildInParent()` 返回 `parent` 后，会循环执行该步骤，直到返回的 `parent` 为空，即循环到 View 树的根视图。

执行完第 ④ 步后，从最初的 `child` 到所有父视图都设置好了绘制所需要的标识，包括 `DRAW_ANIMATION`、`DIRTY_OPAQUE`、`DIRTY`。

下面继续来看 `ViewGroup` 以及 `ViewRoot` 中对 `invalidateChildInParent()` 函数的不同实现。

① 判断该视图的 `mPrivateFlags` 是否也已经包含 `DRAWN` 标识。该标识的含义是视图是否已经绘制到屏幕上了，这与视图的 `VISIBLE` 状态有所不同，就算视图是 `VISIBLE` 的，但是如果视图没有显示到屏幕上，则其 `DRAWN` 标识依然为 `false`。

② 如果已经完成动画，或者没有动画，则寻找此参数中 `dirty` 区和该视图显示区的合集，判断条件如下代码所示，其中引用了两个标识，分别是 `FLAG_OPTIMIZE_INVALIDATE` 和

## FLAG\_ANIMATION\_DOWE.

```

2531     if ((mPrivateFlags & DRAWN) == DRAWN) {
2532         if ((mGroupFlags & (FLAG_OPTIMIZE_INVALIDATE | FLAG_ANIMATION_DONE)) !=
2533             FLAG_OPTIMIZE_INVALIDATE) {
2534             dirty.offset(location[CHILD_LEFT_INDEX] - mScrollX,
2535                          location[CHILD_TOP_INDEX] - mScrollY);
2536
2537             final int left = mLeft;
2538             final int top = mTop;
2539
2540             if (dirty.intersect(0, 0, mRight - left, mBottom - top) ||
2541                 (mPrivateFlags & DRAW_ANIMATION) == DRAW_ANIMATION) {
2542                 mPrivateFlags &= ~DRAWING_CACHE_VALID;

```

这段代码的逻辑判断中使用了 `if(flags & (X | Y) != X)` 这样的语法，其语义如下。

要么 `flags` 中没有 `X` 标识，要么是同时有 `X` 和 `Y` 标识，翻译一下就是要么没有动画，要么有动画同时通话已经完成。对于这两种条件，处理逻辑是相同的，那就是首先对 `dirty` 区域进行 `offset()`，参数 `location[]` 保存的是子视图 `child` 在父视图中的位置，经过 `offset()` 后，就将该区域转换到了子视图在父视图显示区中的位置，然后再经过 `intersect()` 运算，最终找到交叉区，该过程如图 13-22 所示。

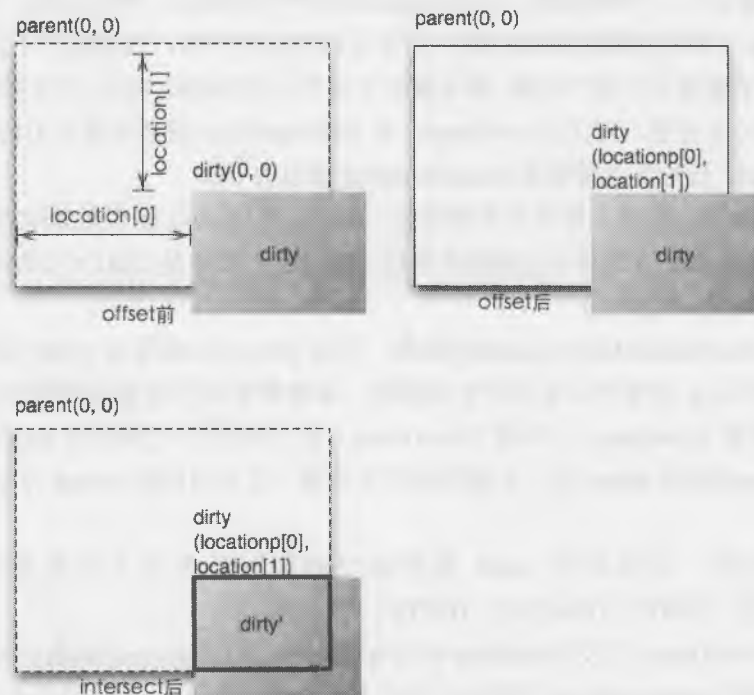


图 13-22 寻找最终需重绘(dirty)的区域

读者可能觉得奇怪，图 13-22 中的 `dirty` 区转换到父视图的 `dirty'` 区中后，区域变小了。的确是这样，因为超过 `dirty'` 的区域会被父视图遮住，所以不需要重绘，该变换多次执行后，`dirty` 区就会越来越小，

最后产生的 dirty 区域才是真正在屏幕上显示的区域，而重绘只需要绘制该区域即可。

③ 如果正在进行动画，那么就把该 ViewGroup 的整个区域添加到 dirty 区。这样做虽然处理简单，但却增加了以后绘制的工作量，但又有什么办法呢？因为动画影响的就是整个区域。

④ 给 location[] 赋值为该 ViewGroup 的 mLeft 和 mTop，并返回该 ViewGroup 的 mParent。

下面再来看 ViewRoot 中是如何处理 invalidateChildInParent() 的。ViewRoot 中该函数仅仅是调用 ViewRoot 中的 invalidateChild()，这个函数与 ViewGroup 中的函数同名，注意区分。并且，在 ViewRoot 中，invalidateChild() 中的第一个参数没有任何用处，其执行流程图参见附图 7，具体过程如下。

① 调用 checkThread()，确保该函数是从 UI 线程中执行。

② 对 dirty 区进行一定的调整，主要是把 dirty 区转换到 ViewRoot 中的屏幕坐标上，具体包括：

(1) 同样调用 offset() 方法，将 dirty 区转换到 ViewRoot 的屏幕显示区。

(2) 如果该 ViewRoot 包含 Translator 对象 mTranslator，则请求该对象对该 dirty 区域进行转换。Translator 对象只有当屏幕的物理像素和程序的逻辑像素不相等时才存在，比如当屏幕的分辨率为 800×480 像素，而 Framework 却把程序窗口配置成 480×320 像素时，于是对于 dirty 这个矩形区，需要进行一定的转换。

(3) 调用 dirty.inset(-1, -1) 对区域进行内嵌 1 像素，这仅在 mAttachInfo 的 mScalingRequired 为 true 时才执行。

③ 在 ViewRoot 类中有一个 mDirty 矩形变量，它保存了上一次 invalidate 所产生的 dirty 区，应用程序可以在一次 MessageQueue 消息处理过程中多次调用 invalidate() 函数，从而不断更新 ViewRoot 中的 mDirty 变量。本步骤中正是把新的 dirty 添加到 mDirty 变量中。

④ 此时，万事俱备，应该请求 View 树进行重绘了。在发出请求之前，先查看是否已经发出过请求，并且 View 内部是否还没有处理该请求。如果是，则不用再次发送。代码中是通过变量 mTraversalScheduled 查看是否发出过请求，该变量会在执行 performTraversals() 方法开始前被置为 false，其生命期如图 13-23 所示。

至此，一次 invalidate() 就结束了，回顾一下，它大致做了两件事情。其一是给所有需要重绘的视图添加了一个 DIRTY 或者 DIRTY\_OPAQUE 标记；其二是通过矩形运算，找到真正需要重绘的矩形区，并将其保存在了 ViewRoot 类中的 mDirty 变量中。有了这两个信息，View 树重绘就能够决定通知哪些 View 进行重绘，并告诉它们应该重绘什么区域。

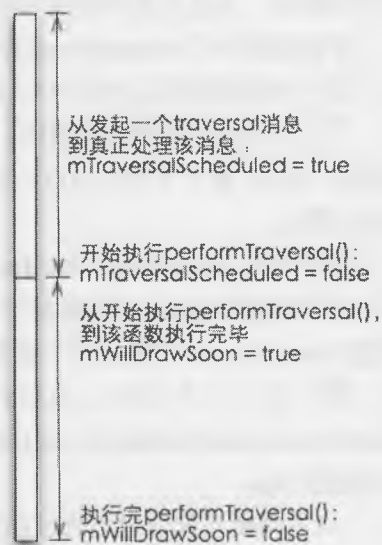


图 13-23 mWillDrawSoon 变量的生命期

## 13.6.10 requestFocus()

要想让某个视图获得焦点，一种方法是用户使用方向键将焦点移动到该视图，另一种方法是程序员直接调用视图的 `requestFocus()` 函数，当然，前者实际上内部也是调用该函数完成的。

和 `invalidate()` 的调用有点相似，`requestFocus()` 也是不能独自完成的，当一个视图想要获取焦点时，必须请求它的父视图完成该操作，为什么呢？因为父视图知道当前哪个视图正在拥有焦点，如果要进行焦点切换，则必须先告诉原先的视图放弃焦点，而这些操作所需要的信息是在父视图中保存的，所以 `requestFocus()` 也必须由父视图完成。

该函数有如下三个不同的版本。

- `requestFocus()`：无参数，它被转换成 `requestFocus(View.FOCUS_DOWN)`。
- `requestFocus(int direction)`：它被转换成 `requestFocus(direction, null)`。
- `requestFocus(int direction, Rect preFocusRect)`：第一个参数代表往哪个方向上寻找下一个视图，第二个参数为当前拥有焦点的视图所占的矩形区，这个区域是相对该视图的直接父视图。

从这三个函数的转换关系可以看出，应用程序中针对某个视图调用 `requestFocus`（无参数）时，并不一定会把焦点赋给该视图。因为该函数内部实际上在 `DOWN` 方向上找下一个可以获得焦点的视图，至于哪个视图就不一定了，这取决于父视图的执行逻辑，这就是为什么该函数的返回值是一个 `boolean` 类型的原因，其意义是该视图到底能不能获得焦点。

下面就来分析 `requestFocus(direction, preFocusRect)` 函数的内部执行过程，流程图参见附图 7。

- ① 判断该视图是不是 `FOCUSABLE` 的，如果不是，则直接返回 `false`。
- ② 如果当前是 `Touch` 模式，但是视图的 `FOCUSABLE_IN_TOUCH_MODE` 却为 `false`，即该视图不能在 `Touch` 模式下获得焦点，则直接返回 `false`。代码中调用 `View` 类的 `isInTouchMode()` 判断是否是 `Touch` 模式。
- ③ 调用 `hasAncestorThatBlockDescendantFocus()` 判断是否父视图阻止该子视图获得焦点，如果阻止，则直接返回 `false`。应用程序可以调用 `ViewGroup` 的 `setDescendantFocusability(int focusability)` 方法设置该 `ViewGroup` 是否阻止其子视图获得焦点，默认情况下都不阻止。
- ④ 以上三步实际上执行的都是前期检查，下面将真正进行焦点获取的操作。该步调用 `handleFocusGainInternal(dir, rect)` 进行具体的焦点获取操作，执行完该函数后，则该视图肯定会获取焦点，所以返回 `true`。

(1) 判断当前窗口是否已经获得焦点，如果已经获得，则直接返回。相当于说，应用程序连续调用两次 `requestFocus()`，第二次调用时就直接返回了。

(2) 如果还没拥有焦点，则给 `mPrivateFlags` 变量中添加 `FOCUSED` 标识，这意味着该视图已经真正拥有焦点了。

(3) 调用父视图 `mParent` 的 `requestChildFocus(this, this)`，第一个参数代表 `child` 视图，第二个参数代表 `focused` 视图。比如，`C` 中包含 `B`，`B` 中包含 `A`，如果从 `B` 中调用该函数时，就会执行

C.requestFocusChildFocus(B, A)。该函数是 requestFocus()函数的核心过程，其内部会进行递归调用，并最终调用到 ViewRoot 中的 requestChildFocus()。

(4) 执行完上一步操作后，该视图的父视图及父父视图都已经获知了本次焦点获取请求，因此，接下来需要做一些收尾工作了。本步中回调 onFocusChanged()，应用程序可以重载该函数以便进行其他操作。这里大家顺便区分一下两个英语时态的语义，这里回调的是 onFocusChanged()，而不是 onFocusChange()，抽象一下就是 xxxed()，而不是 xxx()。在 Framework 的其他地方，有时回调的是 onXXXed()，有时却是 onXXX()，两者的区别在于前者是当执行完指定操作后才回调，而后者是在指定操作执行前回调。程序员需要遵守这种函数命名规则，以便调用者能够更清晰地理解代码逻辑。

(5) 调用 refreshDrawableState()，因为 focus 状态改变后，视图的背景图有可能也需要改变。

下面再来具体分析上面第 (3) 步中提到的 requestChildFocus(View child, View focus)函数的内部执行过程。一般来讲，父视图也是一个 ViewGroup 对象，直到最后一个父视图，才是 ViewRoot 对象，因此，首先来看 ViewGroup 中的该函数。

① 判断 mGroupFlags 是否包含 FOCUS\_BLOCK\_DESCENDANTS 标识，如果有，则意味着阻止子视图获得焦点。这一步其实是多余的，因为在前面第 ③ 步中已经进行过同样的判断了，能执行到这里肯定不会阻止。

② 调用 super.unFocus()。这里把该 ViewGroup 当作一个普通的视图处理，因为 ViewGroup 本身也是一个 View，所以 ViewGroup 本身也可以获取焦点。此处调用 super.unFocus()就是当该 ViewGroup 本身拥有焦点时，就先让它释放焦点。

③ mFocused 变量代表了之前拥有焦点的子视图，如果该变量不为空，则需要先通知该视图释放焦点，此处调用 mFocused.unFocus()，然后给该变量赋上新值 child。该 child 是该 ViewGroup 的直接子视图，而不是真正拥有焦点的视图。举个例子，D 中包含 C，C 中包含 B，B 中包含 A，假设 A 是最终会获得焦点的视图，那么执行完毕后，相关的状态如表 13-5 所示。

表 13-5 视图及其父视图中焦点状态变化

对象名称	mFocused 值	FOCUSED 标识
D	C	无
C	B	无
B	A	无
A	无此变量	有

④ 如果该 ViewGroup 也有父视图，则递归调用父视图的 requestChildFocus(this, focused)。第一个参数为该 ViewGroup 对象，每次递归调用时都不同；第二个参数是 focused，该参数一直都指向最终应该获得焦点的视图。

以上步骤最终递归到 ViewRoot 中的 requestChildFocus()函数，该函数的执行过程如下。

- ① 调用 checkThread()确保是在 UI 线程中执行该调用。
- ② 如果目标焦点视图就是当前焦点视图，则什么都不用做，ViewRoot 中用变量 mFocusedView 保



存真正拥有焦点的视图。这个判断是多余的，因为在前面第④步第(4)中已经检查过目标视图是否已经拥有焦点，所以，能执行到这里，意味着 `mFocusedView` 不是目标焦点视图，于是调用 `scheduleTraversal()` 发起一个 View 遍历请求。

③ 最后，将新的焦点视图赋值给 `mFocusedView`。

至此，就结束了 `requestFocus()` 的操作。

### 13.6.11 requestLayout()

该函数的执行过程比较简单，因为当 View 树进行重新布局时，总是重新给所有的视图进行布局，因为，最简单的想法就是只要设置一个标识就好了。其执行过程如图 13-24 所示。

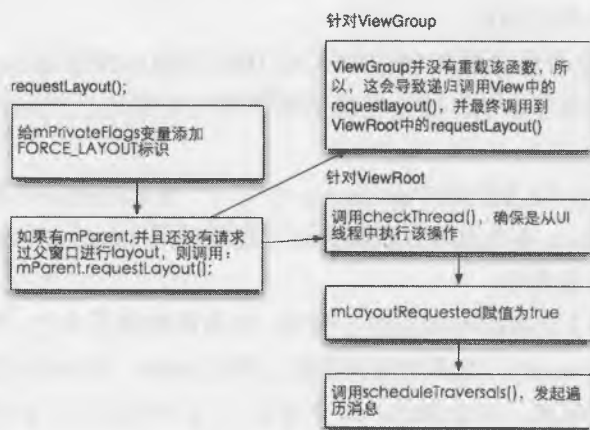


图 13-24 requestLayout()内部执行过程

首先给 `mPrivateFlags` 添加 `FORCE_LAYOUT` 标识，然后调用 `mParent` 的 `requestLayout()` 函数。对于一个具体的 View 对象而言，其父视图要么是一个 `ViewGroup` 实例，要么是一个 `ViewRoot` 实例，而前者并没有对该函数重载。也就是说，`ViewGroup` 会按照基类 (`View`) 中的该函数进行处理，如果有多层视图嵌套，这就会产生一个递归调用，并最终调用到 `ViewRoot` 类的 `requestLayout()` 函数。该函数的执行流程如下。

① 调用 `checkThread()` 确保本次调用是在 UI 线程中执行的，非 UI 线程执行该函数将导致状态管理的混乱，并最终 crash 掉。

② 给 `ViewRoot` 中的变量 `mLayoutRequested` 赋值为 `true`，之后真正进行布局的代码将检查该变量，并决定是否需要重新布局。

③ 调用 `scheduleTraversals()` 发起一个 View 树遍历的消息，该消息是异步处理的，对应的处理函数为 `performTraversals()`。

下面将具体介绍 `performTraversals()` 的内部执行过程。

### 13.7 遍历 View 树 `performTraversals()` 的执行过程

`performTraversals()` 函数正是系统内进行 View 树遍历工作的核心函数，该函数内部逻辑稍有复杂，一个函数代码长度约 600 行。虽然该函数的代码很长，但其主体逻辑却是很清晰的，其执行过程可简单概括为根据之前所有设置好的状态，判断是否需要重新计算视图大小（`measure`）、是否需要重新安置视图的位置（`layout`），以及是否需要重绘（`draw`）视图，其框架过程如图 13-25 所示。

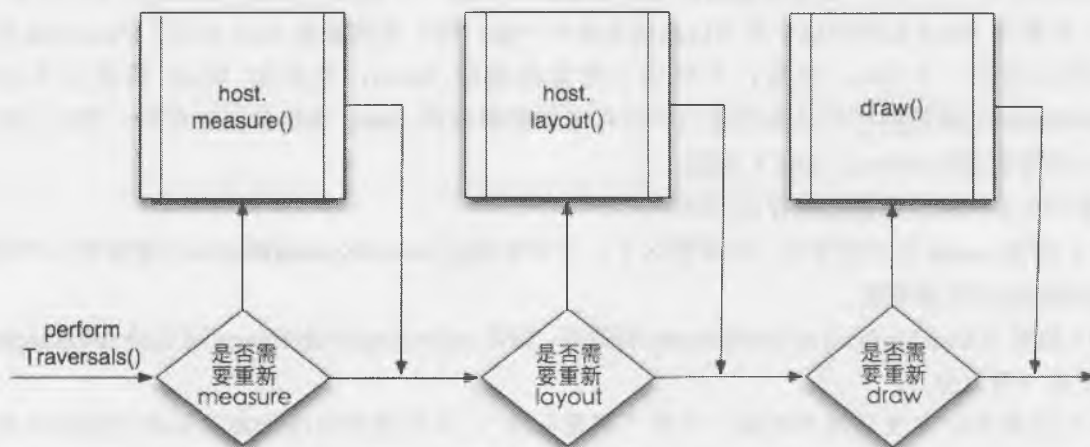


图 13-25 `performTraversals()` 主要过程

关于 `measure()`、`layout()` 及 `draw()` 函数的内部执行流程将在后面小节中单独介绍，它们代表了视图系统的核心逻辑。

下面具体来看以上三大环节是如何在代码中实现的，其流程图参照附图 8。

① 处理 `mAttachInfo` 的初始化，并根据 `resize`、`visibility` 改变的情况，给相应的变量赋值。

(1) 如果是第一次进行 View 树遍历，则对 `mAttachInfo` 中的变量进行初始化。所谓的“第一次”`mFirst` 变量是指，该窗口创建好以后的第一次显示，在后面整个运行过程中，都不是第一次，直到应用程序退出，该窗口被销毁为止。初始化后，调用 `host.dispatchAttachedToWindow()`，参数包含 `mAttachInfo`，所有的子视图都将把 `mAttachInfo` 的值复制到自己的 `mAttachInfo` 变量中。

(2) 如果不是第一次，则查看是否 `resize` 过。`resize` 一般是用于输入窗口显示后，`WmS` 通知客户端窗口调整窗口大小，`resize` 发生后，客户端窗口需要做以下工作。

- 将 `fullRedrawNeeded` 置为 `true`，即需要全部重绘。
- 将 `mLayoutRequested` 置为 `true`，即需要重新为视图指定位置。
- 将 `windowResizesToFitContent` 置为 `true`，该变量将在下面被使用。

(3) 如果窗口的 visibility 状态发生改变, 比如窗口从后台切换到前台或者相反, 则给 mAttachInfo 中的变量 mWindowVisibility 赋新值, 并调用 host.dispatchWindowVisibilityChanged() 函数将这个变化信息传递给所有的子视图。

② 如果需要重新布局, 即 mRequestedLayout 变量为 true, 则首先需要重新计算所有视图的大小。换句话说, 要重新 layout, 就必须重新 measure。

(1) 如果 mFirst 为 true, 则调用 host.fitSystemWindow(attachInfo.mContentInsets)。参数 mContentInsets 是 WmS 为应用程序设置的“嵌入区”, inset 本意是“嵌入”、“插图”, Content 是指视图区用于真正显示内容的区域, 所以, mContentInsets 的大小一般是指状态栏窗口的大小, 当应用程序全屏运行时, mContentInsets 的大小为 0。这里有意思的是该变量的类型是一个 Rect, 而这个 insets 的大小是由 WmS 指定的, 问题是 WmS 如何把这个矩形信息传递给客户窗口呢? 答案就是 Rect 实现了 Parcelable 接口, 客户端可以创建一个 Rect 变量, 并把这个变量传递给 WmS, 然后由 WmS 修改后再返回。fitSystemWindow() 函数的作用是告诉窗口中所有子视图根据该 Inset 调整自己的布局, 实际上就是用 inset 大小改变视图的 mPaddingXXX 的值。

如果不是第一次, 则继续执行以下流程。

(2) 判断 insets 是否有变化, 如果变化了, 也需要调用 host.fitSystemWindow() 通知窗口中的视图修改 mPaddingXXX 参数值。

(3) 如果 mAttachInfo 的 mVisibleInsets 有改变, 则将 mPendingVisibleInsets 赋值给 mVisibleInsets, 该矩形变量一般为空。

(4) 根据窗口宽度获得子视图大小的“测量标准”, 其中宽度标准和高度标准分别保存到变量 childWidthMeasureSpec 和 childHeightMeasureSpec, 以下简称 widthSpec 和 heightSpec。而这两个变量的赋值过程如以下代码所示:

```
childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);
childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);
```

本步操作中, 有三个相关的变量容易混淆, 一个是 lp, 一个是 desiredWindowWidth, 另一个是 measureSpec。

lp 变量代表的是根视图的 LayoutParams, lp.width 或者 lp.height 直接来源于用户的定义, 比如 WRAP\_CONTENT、MATCH\_PARENT 等。

desiredWindowWidth 是指实际窗口在屏幕上的大小, 默认情况下就是屏幕大小, 即 displayMetrics.widthPixels, 当窗口大小改变后, 其值将变为 mWinFrame 的大小。比如当有输入法窗口时, mWinFrame 的大小是屏幕大小减去输入法窗口的大小。

measureSpec 是调用 getRootMeasureSpec() 函数的返回值, 分两种, 分别是 MeasureSpec.EXACTLY 和 MeasureSpec.AT\_MOST, 这两个常量分别是 0x4000 0000 和 0x8000 0000。measureSpec 是调用 MeasureSpec.makeMeasureSpec(size, 类型) 产生真正的 measureSpec 值, size 代表了真正的大小, 换句话说, 该变量的高 16 位代表类型, 低 16 位代表实际大小。measureSpec 将作为测量子视图大小的标准, 关于 measure() 的具体过程见后面小节。

(5) 以上一步获得“测量标准”，接着调用 `host.measure(widthSpec, heightSpec)`，该函数内部计算所有子视图的真正大小。简单的讲，该函数的作用就是把那种 `WRAP_CONTENT` 和 `MATCH_PARENT` 等的相对值转换成真正的数值大小。

③ 查看 `mAttachInfo` 中的 `mRecomputeGlobalAttributes` 是否为 `true`。该变量的含义是有子视图的 `visibility` 等属性发生过了改变，`ViewRoot` 需要重新获取这些属性，于是调用 `host.dispatchCollectViewAttributes(0)`，参数 0 代表 `View.VISIBILITY`。

④ 如果是第一次，或者发生了 `visibility` 变化，则需要修正 `lp` 中的 `softInputMode`。当然这只是在 `softInputMode` 没有指定的情况下，如果指定了就不用修改。

⑤ 无论是 `resize` 还是 `inset` 变化了，或者是窗口的 `visibility` 发生了变化了，都意味着需要重新设置客户窗口的大小，并重新计算窗口中视图的大小。请注意区分本步操作和第 2 步操作的区别，第 2 步是指窗口内部视图变化引起的重新计算大小，而本步是指窗口本身大小发生了变化。

(1) 调用 `relayoutWindow()`，该函数内部则会调用 `sWindowSession.rel原因out()` 请求 `WmS` 按照指定的大小重新分配窗口大小，如以下代码所示：

```
2647         int relayoutResult = sWindowSession.rel原因out(
2648             mWindow, params,
2649             (int) (mView.mMeasuredWidth * appScale + 0.5f),
2650             (int) (mView.mMeasuredHeight * appScale + 0.5f),
2651             viewVisibility, insetsPending, mWinFrame,
2652             mPendingContentInsets, mPendingVisibleInsets,
2653             mPendingConfiguration, mSurface);
```

最后一个参数 `mSurface` 是客户窗口创建的，`WmS` 内部将为该 `Surface` 对象分配真正的显存，等该函数返回后，应用程序就可以在该 `Surface` 中绘制了。

另外，参数 `mPendingContentInsets`、`mPendingVisibleInsets` 都是输出参数，`WmS` 会给这些变量中填入新值。

(2) 如果 `mPendingContentInsets` 发生了改变，则需要调用 `fitSystemWindow()` 通知所有子窗口根据这种改变调整自身的大小。

(3) 如果 `mPendingVisibleInsets` 发生了改变，则将新改变的值赋给 `mAttachInfo` 的 `mVisibleInsets`。

(4) 如果之前 `Surface` 无效，而此时有效了，则需要设置两个变量的值，一个是设置 `newSurface` 为 `true`，另一个是设置 `fullRedrawNeeded` 为 `true`。本步中的条件的判断有点意思，如以下代码所示：

```
983         if (!hadSurface) {
984             if (mSurface.isValid()) {
```

其中变量 `hadSurface` 是在调用 `relayoutWindow()` 之前赋值的，其值为 `mSurface.isValid()` 的返回值。这也就是为什么这个变量叫做 `hadSurface` 而不是 `hasSurface` 的原因，因为它保存的是“之前”是否有。而当调用 `relayoutWindow()` 后，再次调用 `mSurface.isValid()` 肯定会返回 `true`。所以，本步条件成立的情况实际上就是当 `mFirst` 为 `true` 时发生。

(5) 给 `mAttachInfo` 中变量 `mWindowLeft` 和 `mWindowTop` 赋新值。该新值保存在 `mWinFrame` 中，

而该变量是在调用 `sWindowSession.relayout()` 时作为参数传递给 `WmS` 的, `WmS` 会对该值进行修改。

(6) 处理 `SurfaceHolder.Callback` 逻辑。

(7) 最终, 如果窗口尺寸发生了改变, 则调用 `host.measure()` 重新计算窗口中视图的大小。

⑥ 接下来, 根据以上步骤的执行结果, 判断是否需要进行重新布局。比如当任意视图的大小发生变化时, 它会影响其他视图的布局, 如果是, 则调用 `host.layout()` 进行真正的布局操作。该函数的内部详细流程参见后面小节。

⑦ 如果根视图包含一个 `insets` 变化的监听者, 那么, 需要执行监听者指定的动作, 即调用 `attachInfo.mTreeObserver.dispatchOnComputeInternalInsets()`。执行完毕后, 监听者将设置新的 `insets` 值, 如果 `insets` 发生了变化, 则需要把这个变化报告给 `WmS`, 即调用 `sWindowSession.setInsets()` 函数。

⑧ 如果 `mFirst` 为 `true`, 则让该视图获得焦点, 即调用 `mView.requestFocus()`。

⑨ 如果 `mAttachInfo.mHasWindowFocus` 为 `true`, 则意味着该窗口已经是当前交互窗口, 然后接着调用 `WM.LP.mayUseInputMethod()` 判断当前窗口是否会使用输入法, 应用程序可以在 `AndroidManifest.xml` 指定是否应用窗口不使用输入法, 默认情况下都使用。如果是的话, 则调用 `imm.onWindowFocus()` 通知 `InputMethodManager` 有了新的窗口及视图, 如果该视图可以获得焦点并可以编辑, 则输入法窗口将会显示出来。

⑩ 最后, 就需要将所有的视图绘制到屏幕上, 开始真正的 `draw()` 流程。关于 `draw()` 函数的内部执行流程, 见后面小节。

(1) 判断没有取消绘制, 并且不是 `newSurface`, 则调用 `draw()` 函数开始绘制。代码中使用 `cancelDraw` 变量表示是否“取消绘制”, 该值来源于 `TreeObserver` 的 `dispatchPreDraw()` 回调, 应用程序可以添加一个 `OnDispatchPreDraw` 接口对象, 从而在开始绘制之前执行操作, 并可以阻止重绘。`newSurface` 变量的含义是指, `ViewRoot` 中创建了 `Surface` 对象, 但是该对象还没有被 `WmS` 分配真正的显存, `ViewRoot` 中是调用 `sWindowSession.relayoutWindow()` 为该 `Surface` 对象中分配真正的显存, 在一般情况下, 此处的 `newSurface` 都是 `false`。

(2) 如果 `mFirst` 为 `true`, 则绘制完毕后, 需要向 `WmS` 报告绘制完成, 即调用 `sWindowSession.finishDrawing()`。

(3) 如果 `cancelDraw` 或者 `newSurface` 条件不满足, 则重新发起一次 `View` 遍历请求, 以便当条件满足后继续绘制。

至此, `performTraversals()` 过程就介绍完了。

## 13.8 计算视图大小 (measure) 的过程

首先需要澄清一个概念, 即“视图大小”, 视图大小是指什么? 应用程序开发时, 程序员可以在 `layout.xml` 文件中使用 `android:layout_height` 及 `layout_width` 属性设置宽和高, 这指的是视图大小吗?

事实上, `View` 系统希望程序员能够理解, 画布 (`Canvas`) 是没有边界的, 即无穷大, 程序员在自定义一个具体的 `View` 时, 应该在 `onDraw(Canvas canvas)` 函数中向画布中绘制视图的界面。此时程序员

应该认为，该画布是无穷大的，即可以绘制任意多的内容或任意大的图形（只要内存足够大），然而实际应用中并不会绘制一个无穷大的界面，那么到底应该绘制一个多大的界面呢？对于不同类型的 View，其绘制的大小有所不同，一般分为两种情况，一种是内容型视图，另一种是图形型视图。

所谓内容型视图一般是指，该视图将显示一段文本内容，比如 TextView，其绘制指的就是显示一段文本内容，内容有多少，就应该绘制多少，所以视图的大小由内容的多少决定。换句话说，该视图“主宰”了自己的大小。

所谓图形型视图一般是指，该视图显示的是一个图形，比如三角形、正方形、背景图等，此时该视图的大小往往会根据父视图为该 View 开了一个多大的“窗口”而动态调整。“窗口”是指，在父视图看来该 View 应该占多大的区域，即布局（layout）大小。换句话说，该视图的大小是“被主宰”的，当然，如果该 View 自己愿意，技术上讲完全可以不被父视图的“窗口”限制，这种“被主宰”只是实际的需求而已。比如程序员设计了一个三角形视图 TriView，使用者可以通过 layout\_height 和 layout\_width 设置该三角形的布局大小，使用者一般会期望该三角形的大小能根据布局大小自动调整，而不是显示一个固定大小的三角形。

因此，回答上面提出的问题，视图大小是无穷的，或者说是没有大小的。layout.xml 文件中 layout\_height 和 layout\_width 属性设置的宽和高不是指视图的大小，而是指父视图给该 View 设置的“窗口”大小，这就是为什么这个属性的名称是以“layout\_”为前缀，而不是直接使用 width 和 height 的原因。该属性值可以是一个“相对值”，比如 WRAP\_CONTENT、MATCH\_PARENT 等，也可以是一个具体值，比如 100dip。

尽管理论上讲视图是没有大小的，但从语言本身的角度讲，人们喜欢“视图大小”这个词语，其含义实际上被篡改了。该词语的语义实际上指父视图为子视图分配的那个“窗口”的大小，更确切地讲应该是视图的布局（layout）大小，View 类内部用两个变量 measuredWidth 和 measuredHeight 保存其值，这两个更应该被命名为 layoutWidth 和 layoutHeight。有读者可能要问了，View 内部不是有 mLeft、mRight、mTop、mBottom 四个变量吗，它与 measuredWidth 又有什么关系呢？实际上，这四个变量指的是该 View 在父视图所占的区域，mRight-mLeft 的大小一般就是 measuredWidth 的大小，mBottom-mTop 就是 measuredHeight 的大小。因此，本章以后所讲的视图大小就是指视图的布局大小。

measure 过程的本质是把视图布局时使用的“相对值”转换为具体值的过程，即把 WRAP\_CONTENT 及 MATCH\_PARENT 转换为具体的值。如果 Framework 中不使用相对值，那么也就完全不需要 measure 过程了。

### 13.8.1 measure 内部设计思路

View 系统启动 measure 过程是从 ViewRoot 中调用 host.measure() 开始，如图 13-26 所示。



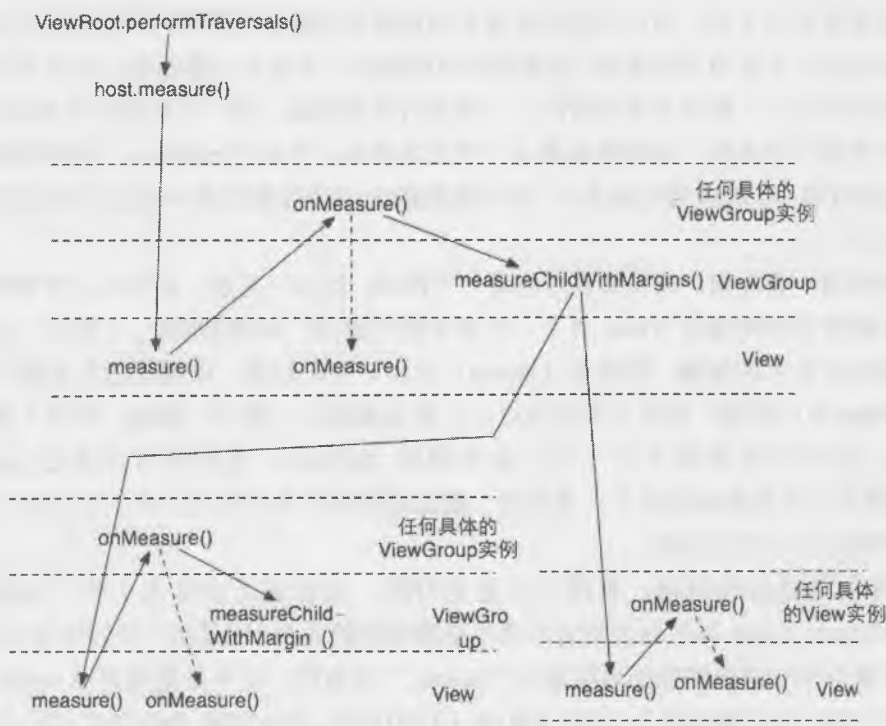


图 13-26 measure()、onMeasure()的递归调用过程

`host.measure()`会调用到 `View` 类的 `measure()`函数，该函数然后回调 `onMeasure()`。在一般情况下，`host`对象是一个 `ViewGroup`实例，该 `ViewGroup`会重载 `onMeasure()`，当然如果 `host`没有重载 `onMeasure()`则会执行 `View`类中默认的 `onMeasure()`。在一般情况下，程序员需要在重载的 `onMeasure()`函数中逐一对所包含的子视图执行 `measure()`操作，为了简化程序设计，`ViewGroup`类内部提供了 `measureChildWithMargin()`函数，该函数内部会进行一定的参数调整，然后再次调用子视图的 `measure()`函数，这就又调用到 `onMeasure()`。如果子视图是一个 `ViewGroup`实例，则应该继续调用 `measureChildWithMargin()`对下一层的子视图进行 `measure`操作；如果子视图就是一个具体的 `View`实例，则在重载的 `onMeasure()`函数内部就不需要再次调用 `measureChildWithMargins()`，从而一次 `measure()`过程结束。

以上过程是 View 系统定义的一个框架模型，在这个模型中，ViewGroup 类是一个 abstract 类，应用程序必须实现一个具体的 ViewGroup 实例。在该实例中，程序员应该调用 measureChildWithMargin() 对下一层的子视图继续进行 measure() 操作，但这不是必需的，因为 measure() 的结果仅仅是把 layout\_width 和 layout\_height 所设置的相对值转换为具体值，这些值将在 layout 过程中辅助父视图对子视图进行布局操作。如果某个 ViewGroup 实例对子视图的布局不依赖子视图的大小，那么就不需要对所包含的子视图进行 measure 操作。

View 中 `measure()` 函数原型为:



```
public final void measure(int widthMeasureSpec, int heightMeasureSpec);
```

在该函数定义中, `final` 关键字说明, 该函数是不能被重载的, 即 View 系统定义的这个 `measure` 框架不能被修改。参数 `widthMeasureSpec` 和 `heightMeasureSpec` 分别对应宽和高的 `MeasureSpec`, `MeasureSpec` 是一个 `int` 型值, 当父视图对子视图进行 `measure` 操作时, 会调用子视图的 `measure()` 函数, 该参数的意思是父视图所提供的 `measure` 的“规格”, 因为父视图最终为子视图提供的“窗口”大小是由父视图和子视图共同决定的。该值由高 32 位和低 16 位组成, 其中高 32 位保存的是 `specMode`, 低 16 位为 `specSize`。`specMode` 有三种, 分别如下。

- `MeasureSpec.EXACTLY`: “确定的”, 意思是说, 父视图希望子视图的大小应该是 `specSize` 中指定的。在一般情况下, View 的设计者应该遵守该指示, 将 View 的 `measuredHeight` 和 `measuredWidth` 设置成指定的值, 当然, 也可以不遵守。
- `MeasureSpec.AT_MOST`: “最多”, 意思是说, 子视图的大小最多是 `specSize` 中指定的值。在一般情况下, View 的设计者应该尽可能小地设置视图的大小, 并且不能超过 `specSize`, 当然, 也可以超过 `specSize`。
- `MeasureSpec.UNSPECIFIED`: “没有限制”, 此时 View 的设计者可以根据自身的特性设置视图的大小。

以上介绍了 `MeasureSpec` 的语义, 该参数是父视图传递给子视图的, 那么最根儿上的 `MeasureSpec` 是怎么产生的呢? 这就要从 `ViewRoot` 中调用 `host.measure()` 函数来分析。

`ViewRoot` 中, 调用 `host.measure()` 函数时, 参数分别是局部变量 `childWidthMeasureSpec` 和 `childHeightMeasureSpec`, 这两个变量的赋值最初是通过调用 `getRootMeasureSpec()` 函数获得的, 如以下代码所示:

```
832 childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);
833 childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);
```

其中参数 `desiredWindowHeight` 代表了窗口希望的大小, 其值一般为窗口本身大小, 参数 `lp` 等于 `mWindowAttributes`, 而 `mWindowAttributes` 在 `ViewRoot` 对象初始化时创建, 如以下代码所示:

```
861 public LayoutParams() {
862     super(LayoutParams.MATCH_PARENT, LayoutParams.MATCH_PARENT);
863     type = TYPE_APPLICATION;
864     format = PixelFormat.OPAQUE;
865 }
```

也就是说, `lp.height` 的值是 `MATCH_PARENT`。下面再来看 `getRootMeasureSpec()` 函数的内容过程, 如以下代码所示:

```

1326 private int getRootMeasureSpec(int windowSize, int rootDimension) {
1327     int measureSpec;
1328     switch (rootDimension) {
1329
1330     case ViewGroup.LayoutParams.MATCH_PARENT:
1331         // Window can't resize. Force root view to be windowSize.
1332         measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.EXACTLY);
1333         break;
1334     case ViewGroup.LayoutParams.WRAP_CONTENT:
1335         // Window can resize. Set max size for root view.
1336         measureSpec = MeasureSpec.makeMeasureSpec(windowSize, MeasureSpec.AT_MOST);
1337         break;
1338     default:
1339         // Window wants to be an exact size. Force root view to be that size.
1340         measureSpec = MeasureSpec.makeMeasureSpec(rootDimension, MeasureSpec.EXACTLY);
1341         break;
1342     }
1343     return measureSpec;
1344 }

```

这段代码正是产生 `measureSpec` 的过程，它揭示了 `lp` 中 `MATCH_PARENT` 及 `WRAP_CONTENT` 和 `measureSpec` 的关系。由于上面 `lp.height` 的值为 `MATCH_PARENT`，所以，最根儿上的 `measureSpec` 类型就是 `EXACTLY`，并且 `specSize` 等于 `windowSize`，即视图窗口的大小。

看到这里有些读者可能会觉得有点混淆，请看以下 `layout.xml` 文件：

```

<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="0dip"
        android:layout_weight="2"
        android:text="I am alive"/>
</LinearLayout>

```

这段代码中 `LinearLayout` 中声明的 `layout_width` 属性将作为 `TextView` 的 `measureSpec` 呢，还是作为 `LinearLayout` 本身的 `measureSpec`？有些读者可能认为 `LinearLayout` 是 `TextView` 的父视图，所以答案是前者，即作为 `TextView` 的 `measureSpec`。如果是这样，`TextView` 中定义的 `layout_height` 属性又将作为谁的 `measureSpec`？所以答案是后者，即作为 `LinearLayout` 本身的 `measureSpec`，如果是这样，那么最根儿上的 `measureSpec` 又将作为谁的 `measureSpec`？

所以，这里就需要指出前面曾经说过的“视图大小是由父视图和子视图共同决定的”，因此，上面的这个问题本身就是错误的。正确的答案应该是 `LinearLayout` 中的 `layout_height` 扮演了两个角色，一个是和 `LinearLayout` 的父视图一起对 `LinearLayout` 本身进行 `measure` 操作；另一个角色是作为 `TextView` 的 `measureSpec`，并和 `TextView` 中的 `layout_height` 一起对 `TextView` 进行 `measure` 操作，关于这个过程细节见后面小节。

至此，我们介绍了 `measure` 框架的一些基本概念及大致思路，下面就开始具体分析这种思路在代码中是如何实现的。

### 13.8.2 ViewGroup 中的 measureChildWithMargins()

ViewGroup 提供了三个类似的函数用于对子视图进行 measure() 操作，分别如下。

- measureChildren(): 如其名称所示，children 是 child 的复数，该函数内部使用 for() 循环调用 measureChild() 对每一个子视图进行 measure 操作。
- measureChild(): 为指定的子视图进行 measure 操作。
- measureChildWithMargins(): 该函数与 measureChild 的唯一区别在于，measure 时考虑把 margin 及 padding 也作为子视图大小的一部分。

由于以上三个函数的本质是一样的，因此接下来仅分析 measureChildWithMargins() 函数。该函数的原型如下：

```
protected void measureChildWithMargins(View child,
    int parentWidthMeasureSpec, int widthUsed,
    int parentHeightMeasureSpec, int heightUsed)
```

其中参数 parentHeightMeasureSpec 是该 child 的父视图传递过来的 measureSpec，参数 heightUsed 是在该 measureSpec 中，父视图已经使用的高度。该函数内部流程分三步。

① 调用 child.getLayoutParams() 获得子视图的 LayoutParams 属性。该步虽然看起来只有一行代码，但其中却蕴涵了一组函数调用。首先来看该函数的返回值，在 View 类中，该函数的默认返回值是 ViewGroup.LayoutParams，而此处却被强制类型转换为 ViewGroup.MarginLayoutParams 类型。从程序的角度来讲，大家都知道，强制类型转换一般只能把子类强制转换为父类，而不能把父类强制转换为子类，除非该对象本身就是子类对象，否则会发生转换错误。而此处 MarginLayoutParams 是子类，其父类是 LayoutParams，但代码中却把 getLayoutParams() 返回值强制转换为了一个子类 MarginLayoutParams 对象，这是为什么呢？

这就要来看 View 对象的 LayoutParams 属性是如何被赋值的。创建一个 View 对象一般有两种方法，一种是使用 XML 文件静态描述，另一种是程序动态调用 View 类的构造函数，构造函数中指明所使用 LayoutParams 参数。两种方法的本质是相同的，因此这里仅介绍第一种方法。

使用 XML 文件创建 View 对象时，程序员一般使用 LayoutInflater 服务从 XML 文件中创建一个 View 对象，如下代码所示：

```
LayoutInflater inflater = (LayoutInflater) context.
    getSystemService(Activity.LAYOUT_INFLATER_SERVICE);
inflater.inflate(R.layout.testlayout, null);
```

该段代码中首先调用 getSystemService() 获得 LayoutInflater 服务，然后调用 inflate() 函数创建参数中指定的 View 对象，该函数的第一个参数对应 XML 文件，第二个参数指定该 View 的父视图，一般为空。XML 文件中则用具体的标签来描述一个 View 对象，比如，可以直接使用 <LinearLayout> 描述一个 ViewGroup 对象，也可以直接使用 <TextView> 来描述一个 View 对象，等等。而在 inflate() 函数内部会根

据这些指定的标签找到对应类，比如 `LinearLayout` 对应的就是 `LinearLayout` 类，`TextView` 标签对应的就是 `TextView` 类。找到这些类后，就会调用相应的构造函数创建 `View` 对象，创建好后，此时该对象内部的 `LayoutParams` 属性是空的，所有的 `View` 对象要想被显示到屏幕上，必须得先调用 `addView()` 方法把该 `View` 对象添加到当前窗口的 `View` 树中。在 `addView()` 函数中的代码如下：

```

1822 public void addView(View child, int index) {
1823     LayoutParams params = child.getLayoutParams();
1824     if (params == null) {
1825         params = generateDefaultLayoutParams();
1826         if (params == null) {
1827             throw new IllegalArgumentException("generate
1828         }
1829     }
1830     addView(child, index, params);
1831 }

```

在该段代码中，由于 `child` 当前的 `LayoutParams` 属性为空，于是调用 `generateDefault..()` 函数生成一个 `LayoutParams` 对象。所有的 `ViewGroup` 实例一般都会重载该函数，比如 `LinearLayout` 中就重载了该函数，并返回一个 `LinearLayout.LayoutParams` 对象，而该类的父类正是 `ViewGroup.MarginLayoutParams`。这就是以上为什么能够被强制转换的原因，对于 `RelativeLayout` 及 `FrameLayout` 等常见 `ViewGroup` 实例也一样，其内部都定义了相应的 `LayoutParams` 类，并且这些类都是基于 `ViewGroup.MarginLayoutParams` 的。

当然，你可以设计一个 `ViewGroup` 实例，并且不重载 `generateDefaultLayoutParams()`。如果这样的话，应用程序为在向你定义的这个 `ViewGroup` 中添加 `View` 时，将使用 `ViewGroup` 中默认的 `ViewGroup.LayoutParams`，这将导致程序员在 XML 中使用的 `margin` 的属性不会起作用，如以下代码所示：

```

<ViewGroup class="com.haiii.android.CusViewGroup"
    android:orientation="vertical"
    android:layout_width="250dip"
    android:layout_height="30dip"
    android:layout_margin="30dip" //这一句将不会起作用
    android:background="#ff0000ff">
    <TextView
        android:id="@+id/test_text"
        android:layout_width="60dip"
        android:layout_height="120dip"
        android:background="#ff00ff00" />
</ViewGroup>

```

并且你不能在你自定义的 `ViewGroup` 中使用 `measureChildWithMargins()`，而只能使用 `measureChild()`，否则会因为强制类型转换错误而发生异常。

② 调用两次 `getChildMeasureSpec()` 函数，分别计算出该子视图的宽度和高度的 `Spec`。前面曾经说过，子视图所占布局的大小取决于两个方面，一个是父视图提供的规格，另一个是子视图本身希望占用

的大小，而这就是该函数参数的意义。该函数的原型如下：

```
public static int getChildMeasureSpec(
    int spec,
    int padding,
    int childDimension)
```

参数 `spec` 正是父视图提供的 `spec`，源码中对应的变量是 `parentHeightMeasureSpec`，这个值是 `int` 型。高 16 位代表 `specMode`，一共有三种模式，分别是 `AT_MOST`、`EXACTLY` 及 `UNSPECIFIED`；低 16 位代表 `specSize`，对于非 `UNSPECIFIED` 模式，该值指父视图所允许的最大尺寸或者期望的理想尺寸。

参数 `padding` 代表的是在父视图提供的 `spec` 尺寸中，已经使用的多少，源码中该值包含了 `padding` 填充、`margin` 空余，以及父视图所提供的已经使用的高度 `heightUsed`：

```
mPaddingTop + mPaddingBottom + lp.topMargin + lp.bottomMargin
+ heightUsed
```

`getChildMeasureSpec()` 中会从 `parent` 的 `specSize` 中减去该 `padding`，以便子视图使用已用的空间。

参数 `childDimension` 即为子视图期望获得的高度，其值来源于 `lp.height`，这个值正是 XML 文件中 `android:layout_height` 对应的值，可能是一个具体的值，也可能是 `MATCH_PARENT` 或者 `WRAP_CONTENT`。

`getChildMeasureSpec()` 的作用就是根据以上两个条件确定子视图的“测量规格”，注意，为什么这里确定的不是最终子视图的尺寸而只是“测量规格”？读者可以换一个角度来理解 `parentMeasureSpec` 和 `lp.height` 这两个参数，前者实际上是指父视图可以提供的尺寸，而对一个应用程序而言，程序员必须负责父视图所包含的所有子视图的排列关系，以达到一个优美的界面布局，因此，程序员才使用 `lp.height` 指定期望该子视图的大小，而至于子视图到底有多大，还需要征询子视图本身的意思，所以该函数并不能最终确定视图的尺寸。该函数内部的执行逻辑如表 13-6 所示。

表 13-6 父视图的尺寸规格和子视图尺寸规格的对应关系

parentMeasureSpec	lp.height	childMeasureSpec
EXACTLY + size	childDimension	EXACTLY + childDimension
同上	WRAP_CONTENT	AT_MOST + size
同上	MATCH_PARENT	EXACTLY + size
AT_MOST + size	childDimension	EXACTLY + size
同上	WRAP_CONTENT	AT_MOST + size
同上	MATCH_PARENT	AT_MOST + size
UNSPECIFIED + size	childDimension	EXACTLY + childDimension
同上	WRAP_CONTENT	UNSPECIFIED + 0
同上	MATCH_PARENT	UNSPECIFIED + 0

以上逻辑值得注意的地方是，就算父视图限制子视图的高度，比如 400dip，而程序员依然可以设置

lp.height=1000dip, 这将导致最终子视图的大小是 1000dip, 这就会超出父视图的可视窗口。

③ 上一步已经确定了子视图的测量规格, 最后一步就是“征求子视图本身的意思”了, 即调用 child.measure() 函数。子视图可以重载 onMeasure() 函数, 并可调用 setMeasureDimension() 函数设置任意大小的布局, 而这将成为该子视图的最终布局大小。

以上逻辑是一种非常“民主”的逻辑, 该逻辑由三方组成, 包括父视图、程序员、子视图的设计者。父视图提供了上一级可以提供的大小, 程序员在 XML 文件中使用 layout\_height 设置希望的大小, 而视图最终的大小则由子视图的设计者“拍板”, 良好的视图设计者一般会根据子视图的 measureSpec 设置合适的布局大小, 以尊重程序员的意图。

### 13.8.3 LinearLayout 中的 onMeasure() 过程举例

由于 View 系统中的 measure 过程必须由相应的 ViewGroup 实例共同参与才能完成一个真正的 measure 操作, 因此, 本节以常用的 ViewGroup 实例 LinearLayout 来说明 measure 框架的过程细节。

View 中的 measure() 函数是不能被重载的, 以保证 View 系统中 measure() 的基本流程。ViewGroup 的实例一般需要重载 onMeasure() 函数, 并在该函数中调用 ViewGroup 的 measureChild() 相关函数对每一个子视图进行 measure 操作。LinearLayout 中的 onMeasure() 函数内部, 首先判断该 LinearLayout 是水平的还是垂直的, 并分别调用 measureHorizontal() 和 measureVertical(), 下面仅分析垂直方向的 measure 过程。

该过程表面上似乎挺复杂, 实际上逻辑却比较简单, 可总体分为三个步骤。

① 从第 340 行到第 450 行的第一个 for 循环中, 先计算所有子视图的高度。该过程如图 13-27 所示。

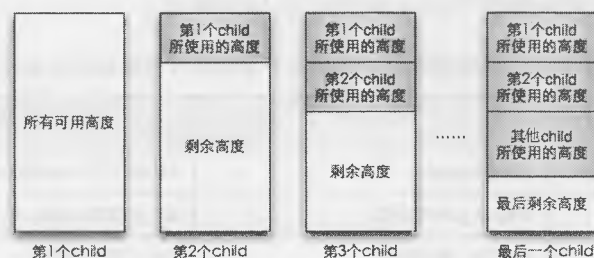


图 13-27 逐个计算子视图高度的过程

源码中使用变量 mTotalLength 保存已经 measure 过的 child 所占用的高度, 该变量刚开始时是 0。for() 循环中调用 measureChildBeforeLayout() 对每一个 child 进行测量, 该函数实际上仅仅是调用了上节所说的 measureChildWithMargins(), 在调用该函数时, 使用了两个参数。其中一个是 heightMeasureSpec, 该参数为 LinearLayout 本身的 measureSpec; 另一个参数就是 mTotalLength, 代表该 LinearLayout 已经被其他子视图所占用的高度, 这个变量更应该命名为 mTotalHeight。注意第二个参数 mTotalHeight, 并

不是说所有的 child 调用时都使用 mTotalHeight, 如果 totalWeight>0 的话, 则第二个参数为 0。

每次 for() 循环对 child 测量完毕后, 调用 child.getMeasuredHeight() 获取该子视图最终的高度, 并将这个高度添加到 mTotalLength 中。

在本步骤中, 暂时避过了 lp.weight>0 的子视图, 即暂时先不测量这些子视图, 因为后面将把父视图剩余的高度按照 weight 大小均匀分配给相应的子视图。源码中使用了一个局部变量 totalWeight 累计所有子视图的 weight 值。处理 lp.weight>0 的情况需要注意, 如果变量 heightMode 是 EXACTLY, 那么, 当其他子视图占满父视图的高度后, weight>0 的子视图有可能分配不到布局空间, 从而不被显示, 只有当 heightMode 是 AT\_MOST 或者 UNSPECIFIED 时, weight>0 的视图才能优先获得布局高度。请看以下两段代码。

第一段代码:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="100dip">

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="20dip" //无论 height 是多少, 都会被释放掉
        android:layout_weight="2"
        android:textSize="30sp"
        android:text="I am alive"/>

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="#ff00ff00" />
</LinearLayout>
```

第二段代码:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="100dip"
    >
    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >
        <TextView
            android:layout_width="fill_parent"
            android:layout_height="0dip"
            android:layout_weight="2"
            android:textSize="30sp"
```



```

        android:text="I am alive"/>
    </LinearLayout>

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="#ff00ff00" />
</LinearLayout>

```

其中,在第一段代码中,由于 LinearLayout 的 lp.height 为 FILL\_PARENT,因此对应 heightMode 是 EXACTLY,其包含的 weight>0 的 TextView 的高度无论是多少,都将由于其后的 ListView 占满屏幕,从而使得 TextView 所占的空间被消耗掉,也就是不能显示到屏幕上,其效果如图 13-28 所示。但如果 ListView 不能占满整个屏幕,那么 TextView 的 height 将起作用。

请看以下第三段代码:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="100dip">

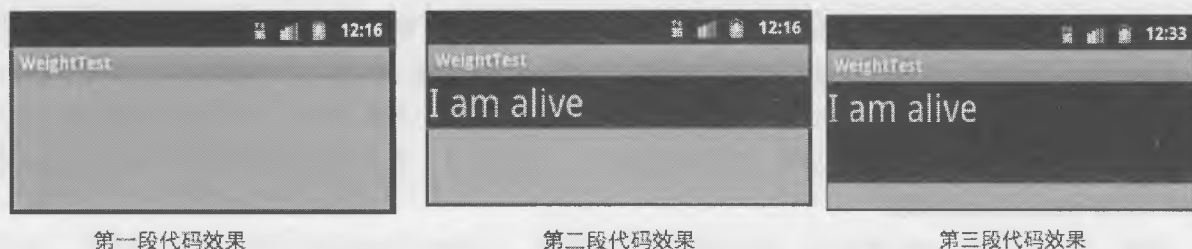
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="60dip"
        android:layout_weight="2"
        android:textSize="30sp"
        android:text="I am alive"/>

    <ListView
        android:layout_width="fill_parent"
        android:layout_height="0dip"
        android:layout_weight="2"
        android:background="#ff00ff00" />
</LinearLayout>

```

在这段代码中,由于 ListView 本身的高度仅 share 剩余部分,所以 TextView 将有足够的空间显示,其 height 为 60,所以将首先获得 60 高度,此时父窗口 LinearLayout 将剩余 40,再把这 40 均匀分配给 TextView 和 ListView,因为它们的 weight 值都为 2。最终的结果是 TextView 将获得 80 高度,而 ListView 获得 20 高度,其效果如图 13-28 所示。

而对于第二段代码,因为 TextView 的父视图 LinearLayout 的 lp.height 是 WRAP\_CONTENT,因此对应的 heightMode 是 AT\_MOST。因此,TextView 的 lp.height 被修改为 WRAP\_CONTENT,才能够优先于 ListView 显示,所以将被显示到屏幕上,如图 13-28 所示。



第一段代码效果

第二段代码效果

第三段代码效果

图 13-28 使用 weight 属性控制视图高度

以上 weight 的处理逻辑对应的代码如下：

```

364         if (heightMode == MeasureSpec.EXACTLY && lp.height == 0 && lp.weight > 0) {
365             // Optimization: don't bother measuring children who are going to use
366             // leftover space. These views will get measured again down below if
367             // there is any leftover space.
368             final int totalLength = mTotalLength;
369             mTotalLength = Math.max(totalLength, totalLength + lp.topMargin + lp.bot
370         } else {
371             int oldHeight = Integer.MIN_VALUE;
372
373             if (lp.height == 0 && lp.weight > 0) {
374                 // heightMode is either UNSPECIFIED or AT_MOST, and this
375                 // child wanted to stretch to fill available space.
376                 // Translate that to WRAP_CONTENT so that it does not end up
377                 // with a height of 0
378                 oldHeight = 0;
379                 lp.height = LayoutParams.WRAP_CONTENT;
380             }

```

② 本步是指从第 452 行到第 475 行中的第二个 for() 循环, 该段逻辑实际上对应用程序是不可见的, 因为它处理的是一种叫做 useLargestChild 的模式, 这种模式本应该使用 android:layout\_useLargestChild 在 XML 中指定, 然而 SDK 中却暂时没有开放这个逻辑, 其内部逻辑似乎有些问题。因为该段代码仅仅是根据最大子视图高度修正了所有子视图的高度总和, 而修正的算法仅仅是把最高子视图的 margin 重新计算在内, 而这在第一步已经计算过了。因此, 这段代码我们暂且不理, 可以认为不会执行到该段代码。

③ 上一步计算了正常的子视图的高度, 接下来需要判断父视图中是否还有剩余空间, 并将剩余空间均匀分配给 weight>0 的子视图们, 对应的代码是从第 478 行到第 575 行。

在该步骤中, 首先调用 resolveSize() 获取所有子视图最终能够占用的布局大小, 因为 mTotalHeight 可能很大, 它仅代表所有子视图最终的高度总和。如果有些子视图不配合, 可能给自己设置一个超过父视图能提供的高度值, 因此需要根据该 LinearLayout 的 heightMeasureSpec 和 mTotalHeight 计算这些子视图们最终可以占用的布局高度, 并重新赋值给 heightSize。

此时, heightSize 代表了父视图能够提供的并且子视图肯定会占用完的高度, 当然, 可能这个值并不够子视图们使用。接着用 heightSize 和子视图们真正的高度产生一个 delta 值, delta 小于 0 意味着空间不够用, 因此需要那些 weight>0 的子视图腾出空间; 如果 delta 大于 0, 则意味着空间还有富余, 因此把富余的空间平均分配给那些 weight>0 的子视图。

该步骤中有一个变量 `mWeightSum`，在默认情况下该值为-1，应用程序可以在 XML 文件中使用 `android:layout_weightSum` 设置该值，其意义是设置一个 `weight` 总和，以便其子视图可以使用比例值设置子视图的权重。比如可以设置 `LinearLayout` 的 `weightSum` 为-2.5，然后设置子视图的 `weight` 值为 0.5，那么该子视图将获得剩余空间的 20%。在默认情况下，该值为-1，因此代码执行是使用 `totalWeight` 作为局部变量 `weightSum` 的值。

接着，将 `mTotalLength` 置为 0，并开始调用 `for()` 循环。过程有些类似第 ① 步，所不同的是，第 ① 步中已经 `measure()` 过了所有 `height` 不为 0 的视图，因此，本步中不需要再重新 `measure()`，而只需要调用 `child.getMeasureHeight()` 获取线程的高度即可。然后对原来的高度增加一个 `share`，`share` 有可能是负值也有可能是正值，修改高度后，再根据这个新高度重新生成一个 `MeasureSpec`，并以该 `MeasureSpec` 要求 `child` 再次进行 `measure`。从该逻辑可以看出，`View` 中的 `weight` 属性并不见得一定是占用剩余空间。笔者曾在第一本书《Android 应用开发》中打过一个比喻，说 `weight` 就像是“股权”，当有剩余时，才能“分红”，而现在看来，也可能有“亏损”，并且一旦亏损，首先是这些“股东”要释放资源。

`for()` 循环执行完毕后，如果那些 `height` 为 0、`weight>0` 的视图配合，会给各自增加或者减少 `share` 指定的高度，但它们也可能不配合。此时，如果子视图们的高度和还超过 `LinearLayout` 所能提供的高度，那么 `LinearLayout` 也无能为力了。最后，重新调用 `resolveSize()` 获取高度，然后调用 `setMeasuredDimension()` 设置该 `LinearLayout` 本身所占用的布局高度。

至此，`LinearLayout` 的 `measure` 过程就结束了。

## 13.9 布局 (layout) 过程

当上一节执行完 `measure` 操作后，接下来的 `layout` 过程其实是比较简单的，其目的就是父视图按照子视图的大小及布局参数，将子视图放置到合适的位置上。布局参数最核心的是处理 `gravity` 参数。

### 13.9.1 layout 过程的设计思路

layout 过程的设计思路如图 13-29 所示。

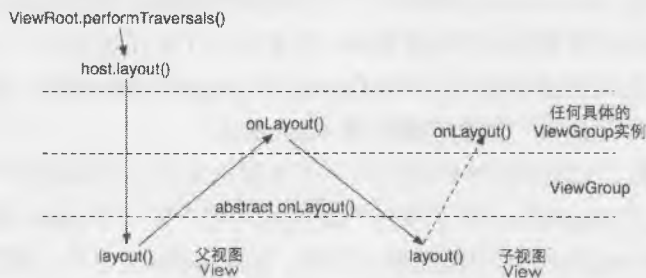


图 13-29 View 系统内 layout 过程的设计思路

最初调用 `layout()` 函数是从 `ViewRoot` 类的 `performTraversals()`, `host` 是一个 `View` 对象, `View` 类中 `layout()` 函数的类型是 `final`, 其子类不能重载该函数, 以保证 `View` 系统中 `layout` 过程不变。

在 `layout()` 函数中, 首先调用 `setFrame()` 函数给当前视图设置参数中指定的位置, 然后回调 `onLayout()` 函数。`ViewGroup` 类中重载了 `onLayout()` 函数, 并且将其函数类型设置成了一个 `abstract` 类型, 因此, 所有的 `ViewGroup` 实例必须实现 `onLayout()`。`View` 系统希望程序员能够在 `onLayout()` 函数中对该视图所包含的子视图进行 `layout` 操作, 当该视图是 `ViewGroup` 时, 程序员一般会在 `onLayout()` 函数中调用子视图 `child` 的 `layout()` 方法, 从而完成对子视图的位置分配。当然, 如果子视图也是一个 `ViewGroup` 实例, 就又会调用相应的 `onLayout()` 函数。

`View` 类中 `layout()` 函数的原型为:

```
public final void layout(int l, int t, int r, int b)
```

参数由调用者即父视图提供, 参数指定了该子视图在父视图中的左、上、右、下的位置。该函数的内部流程如下。

① 调用 `setFrame(l,t,r,b)` 将位置参数保存起来, 这些参数将保存到 `View` 内部变量 (`mLeft`、`mTop`、`mRight`、`mBottom`)。保存完变量前, 会先对比这些参数是否和原来的相同, 如果相同, 则什么都不做, 如果不同才进行重新赋值, 并且在赋值前, 会给 `mPrivateFlags` 中添加 `DRAWN` 标识, 同时调用 `invalidate()` 告诉 `View` 系统原来占用的位置需要重绘。注意是先调用 `invalidate()`, 而后赋值的, 因为需要重绘的是老的区域。

② 回调 `onLayout()`, `View` 中定义的 `onLayout()` 函数默认什么都不做, `View` 系统提供 `onLayout()` 函数的目的是为了系统包含有子视图的父视图能够在 `onLayout()` 函数对子视图进行位置分配, 正因为如此, 如果是父视图, 就必须重载 `onLayout()`, 也正因为如此, `ViewGroup` 类才把 `onLayout()` 重载修改为了一个 `abstract` 类型。

③ 清除 `mPrivateFlags` 中的 `LAYOUT_REQUIRED` 标识, 因为 `layout` 的操作已经完成了。

想要对 `layout` 过程有更深的体会, 就必须了解一个具体的 `ViewGroup` 实例是如何在 `onLayout()` 为子视图进行位置分配的, 因此, 下面一节将以 `LinearLayout` 为例说明 `ViewGroup` 实例的设计者应该如何给子视图分配位置。

### 13.9.2 LinearLayout 中 onLayout() 内部过程

`LinearLayout` 中的子视图有两种排列方式, 一种是水平的, 另一种是垂直的, 本节仅说明垂直方式下的工作流程。

`onLayout()` 函数中首先根据 `mOrientation` 变量判断是水平还是垂直, 如果是垂直, 则调用 `layoutVertical()` 开始进行 `layout` 操作。

① 获得子视图可用的宽度。读者可能觉得奇怪, 这里是进行垂直方向上的布局, 为什么却要可用的宽度, 而不是可用的高度呢? 因为, 就算是垂直方向, 子视图本身也可以水平居中, 而要居中就得知

道可用的宽度是多少,从而计算出子视图左边沿的位置,如图 13-30 所示,至于高度信息将在后面步骤中获得。



图 13-30 视图居中示意图

② 根据父视图中的 `gravity` 属性,决定子视图的起始位置。在默认情况下, `gravity` 是 `TOP`,应用程序可以设置为 `BOTTOM` 或者 `CENTER_VERTICAL`,这三者对应的起始位置如图 13-31 所示。

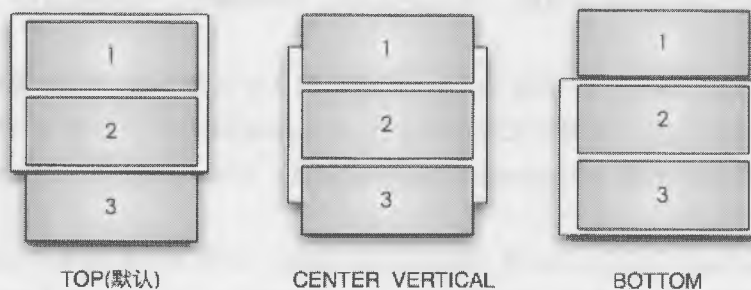


图 13-31 不同 `gravity` 对应的布局效果示意

③ 此时已经确定了子视图的垂直方向上的位置,于是就开始 `for()` 循环为每一个子视图分配位置。

(1) 取出子视图的 `LayoutParams`,并得到 `gravity` 的值。注意这个 `gravity` 值并不是子视图中 `android:gravity` 的值,而是 `android:layout_gravity` 的值,关于这一点的解释见下一小节。

(2) 根据 `gravity` 的值确定水平方向上的起始位置,分为三种,分别是 `LEFT`、`CENTER_HORIZONTAL` 及 `RIGHT`,三者对应的起始位置如图 13-32 所示。

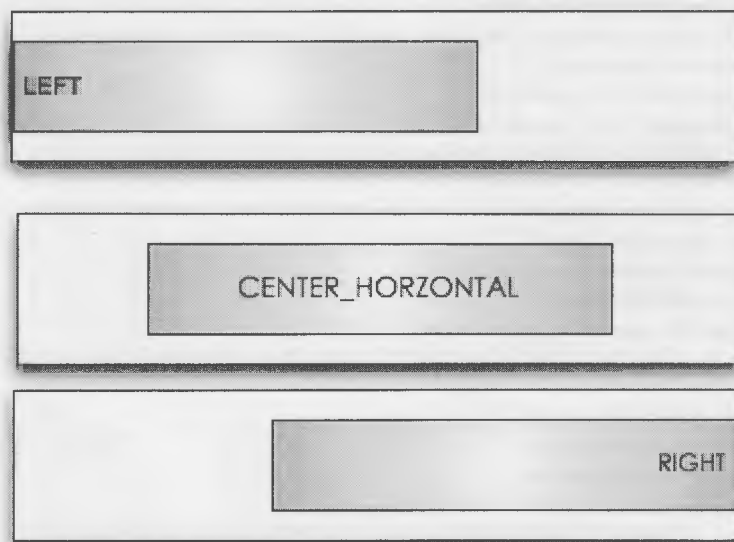


图 13-32 水平方向不同的 gravity 对应的布局效果示意

(3) 调用 `setChildFrame()`, 该函数内部实际上就是调用 `child.layout()` 为子视图设置布局位置。至此, `LinearLayout` 就完成了为所包含的子视图分配布局位置的过程。

### 13.9.3 TextView 中 gravity 与 layout 的关系

上一节说到, 子视图中 `gravity` 的值并不是 `android:gravity` 的值, 而是 `android:layout_gravity`, 这是怎么回事呢? 这个问题其实包含了一个重要概念, 那就是 `LayoutParams` 是如何被赋值的, 这个概念经常让很多读者感到疑惑。

前面曾经说过, 应用程序可以在 `layout.xml` 文件中使用诸如 `<LinearLayout>`、`<TextView>` 等标签定义一个界面。无论是 `ViewGroup` 还是 `View`, 这些标签定义的仅仅是一个视图, 而不能定义 `LayoutParams`, 因为 `LayoutParams` 正如其名称所讲, 它是一个布局属性, 而一个视图在没有被布局时又哪里来的布局属性呢?

而在过去的开发经验中, 大家几乎都使用过 `android:layout_height`、`android:layout_width`、`android:gravity` 等 XML 语法为一个 `LinearLayout` 或者 `TextView` 添加相应的布局属性, 并且这些值似乎都工作正常, 表面上看, 它就是该视图对应的 `LayoutParams` 值, 这又作何解释?

另外, 在 `LayoutInflater` 类中有一个 `inflate(int resId, View root)` 函数, 在大多数情况下, 大家调用该函数时, 第二个参数 `root` 都为空。那么这个参数的意义又是什么? 为什么大多数情况下都要为空呢?

要回答以上问题, 就需要弄清楚系统内部是如何从一个 XML 文件创建一个对应的 `View` 树对象的, 以下面这段 XML 代码为例说明该过程:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:layout_width="200dip"
        android:layout_height="60dip"
        android:gravity="center_horizontal"
        android:background="#ff0000ff"
        android:text="am I center_vertical"/>

    <ListView
        android:layout_width="300dip"
        android:layout_height="100dip"
        android:background="#ff00ff00" />
</LinearLayout>

```

该段代码包含了一个 `LinearLayout`，其中又包含两个子视图，分别是一个 `TextView` 及一个 `ListView`。从 XML 文件中构造对应的 View 树是在 `LayoutInflater` 中的 `inflate()` 函数中完成的，如果该段代码是作为 `setContentView()` 的参数，则会被当作 Activity 的界面，那么 `LinearLayout` 中定义的 `layout_width` 及 `height` 都是有意义的。但如果这段代码仅仅是应用程序调用 `inflate()` 函数创建一个 View 对象，那么 `layout_width` 及 `height` 就没有任何意义。

`inflate()` 函数会首先判断参数 `root` 是否为空，如果为空，就以 XML 文件中的根视图构造一个临时的 View 对象，此处是 `LinearLayout` 对象，此时该 `LinearLayout` 中声明的 `layout_width` 及 `height` 都没有任何用处。接下来继续读取 XML 文件中 `TextView`，读取 `TextView` 后，就需要把该 `TextView` 添加到 `LinearLayout` 中。因为要添加，所以必须有一个 `LayoutParams` 对象，于是，`inflate()` 函数中就以 `TextView` 中定义的 `layout_width` 及 `height` 为参数，回调 `TextView` 父视图的 `generateLayoutParams()` 函数。注意，`TextView` 的父视图正是 `LinearLayout`，而在 `LinearLayout` 的 `generateLayoutParams(attrs)` 函数中，参数 `attrs` 正是来源于 `TextView` 中所声明的全部属性值，除了 `layout_width` 和 `height` 外，还包含 `background`、`text` 等所有声明的属性值，但 `generateLayoutParams(attrs)` 中却只对其中四个属性感兴趣，如以下代码所示：

```

1399 public LayoutParams(Context c, AttributeSet attrs) {
1400     super(c, attrs);
1401     TypedArray a =
1402         c.obtainStyledAttributes(attrs, com.android.internal.R.styleable.LinearLayout_Layout);
1403
1404     weight = a.getFloat(com.android.internal.R.styleable.LinearLayout_Layout_layout_weight, 0);
1405     gravity = a.getInt(com.android.internal.R.styleable.LinearLayout_Layout_layout_gravity, -1);
1406     a.recycle();
1407 }
1408

```

在以上代码中，`super(c, attrs)` 中将读取 `layout_height` 和 `layout_width` 的属性，如以下代码所示。然后继续读取 `weight` 属性和 `gravity` 属性，而这两个属性的名称却分别是 `layout_weight` 和 `layout_gravity`。



```

1399 public LayoutParams(Context c, AttributeSet attrs) {
1400     super(c, attrs);
1401     TypedArray a =
1402         c.obtainStyledAttributes(attrs, com.android.internal.R.styleable.LinearLayout_Layout);
1403
1404     weight = a.getFloat(com.android.internal.R.styleable.LinearLayout_Layout_layout_weight, 0);
1405     gravity = a.getInt(com.android.internal.R.styleable.LinearLayout_Layout_layout_gravity, -1);
1406
1407     a.recycle();
1408 }

```

generateLayoutParams(attrs) 最终将根据 TextView 中的以上四个属性构造一个 LinearLayout.LayoutParams 对象，并以该对象把 TextView 添加到 LinearLayout 中。

回过头来再看上面提出的问题，<TextView>标签中包含的 android:layout\_width、height 等并没有定义 LayoutParams 对象，它只是提供了构造 LayoutParams 参数所需要的值，而真正构造 LayoutParams 对象的是它的父视图，这些参数最终会产生什么样的 LayoutParams 取决于该 TextView 的父视图中 generateLayoutParams(attrs) 的具体实现。

这也就是上一节中所说的为什么要使用 layout\_gravity 而不是 gravity 的原因，因为在 LinearLayout 的 generateLayoutParams() 函数中，是把 android:layout\_gravity 属性值赋值给了 LayoutParams.gravity，而不是 android:gravity。事实上，android:gravity 属性将作为 TextView 内部的参数，TextView 在内部的 onDraw() 函数中将根据 android:gravity 的值决定把文字显示在什么地方。

而当调用 setContentView(int resId) 时，系统内部也同样调用了 inflate() 方法从指定的 XML 文件中产生 View 树，并且调用时参数 root 设置的是内部的一个 FrameLayout 对象。root 参数的意义在于它将提供 generateLayoutParams() 函数，该函数产生的 LayoutParams 对象将作为 XML 文件中包含的视图被添加到 root 时的布局属性。

如果你在一个 XML 文件中仅仅定义了一个 TextView 标签，然后调用 inflate(xml, null) 从这个 XML 文件中 inflate 出一个 TextView 对象，那么该 TextView 标签中所包含的 android:layout\_width、height 将没有任何意义。当你想把 inflate 出来的 TextView 对象添加到某个 ViewGroup 时，还必须构造一个 LayoutParams 对象。

## 13.10 绘制 (draw) 过程

绘制过程主要是把 View 对象绘制到屏幕上，并且如果该 View 是一个 ViewGroup，则需要递归绘制该 ViewGroup 中所包含的所有子视图。

### 13.10.1 视图中可绘制的元素

在介绍视图绘制之前，先来了解一下一个视图中都包含哪些需要绘制的元素，比如一个 TextView，除了具体的文字外，还需要绘制文字的背景等。那么，视图中都包含哪些绘制元素呢？

总的来讲，绘制元素包含四个，分别如下：

- View 背景。每个视图都有一个背景，比如 LinearLayout、TextView，背景可以是一个颜色值，也可以是一幅图片，甚至可以是任何 Drawable 对象，比如一个 Shader、一个 DrawableState 等。应用程序可以使用 setBackgroundColor()、setBackgroundDrawable()、setBackgroundResource()三个函数设置不同的背景。
- 视图本身的内容。比如，对于 TextView 而言，内容就是具体的文字，对于一个 ImageButton 而言，内容就是一幅图片。应用程序会在视图的 onDraw()函数中绘制具体的内容。
- 渐变边框，英文名称是 Fading Edge。渐变边框的作用是为了让视图的边框看起来更有层次感，其本质是一个 Shader 对象。
- 滚动条，英文名称是 ScrollBar。与 PC 上的滚动条不同，Android 中的滚动条仅仅是显示滚动的状态，而不能被用户直接下拉。

在以上四个元素中，应用程序一般只需要重载 View 的 onDraw()函数，并绘制视图本身，其他的三个元素都是由 View 系统自动完成的。并且 View 系统提供了相应的 API 接口，应用程序只需要设置元素所使用的具体颜色或者 Drawable 对象即可。

### 13.10.2 绘制过程的设计思路

绘制的总体过程如图 13-33 所示，图中虚线代表了重载关系。

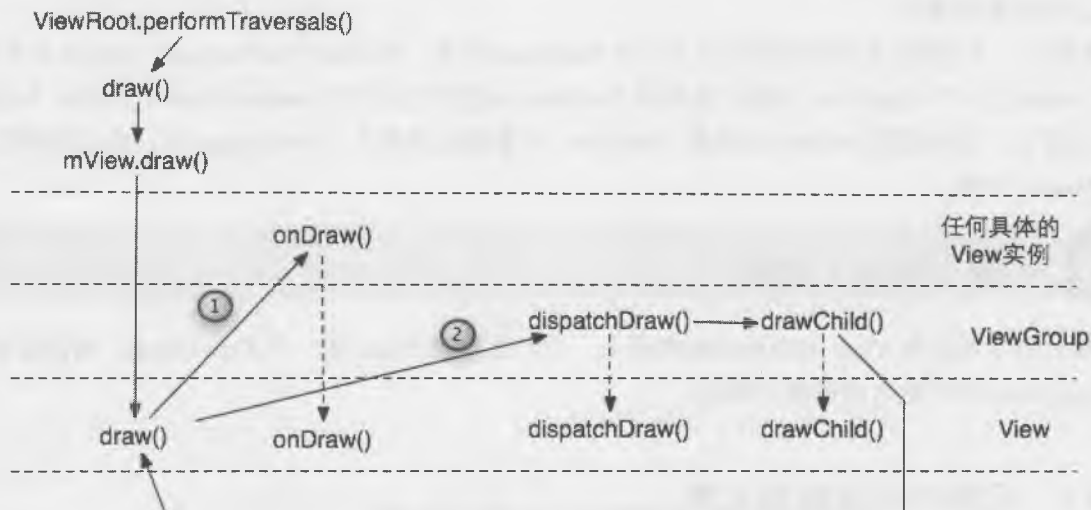


图 13-33 绘制操作的总体过程

绘制过程从 ViewRoot 的 performTraversals()函数中开始，首先调用 ViewRoot 中的 draw()函数，该

函数中进行一定的前端处理后,再调用 `mView.draw()`。`mView` 对象就是窗口的根视图,对于 Activity 而言,就是 `PhoneWindow.DecorView` 对象。

在一般情况下,View 对象不应该重载 `draw()` 函数,因此, `mView.draw()` 就调用到了 View 类的 `draw()` 函数。该函数的内部过程也就是 View 系统绘制过程的核心过程,该函数中会依次绘制上一节所讲的四种种绘制元素,其中绘制视图本身的具体实现就是回调 `onDraw()` 函数,应用程序一般会重载 `onDraw()` 函数以绘制所设计的 View 的真正界面内容。

绘制完界面内容后,如果该视图内部还包含子视图,则调用 `dispatchDraw()` 函数, `ViewGroup` 重载了该函数。因此,实际调用的是 `ViewGroup` 中的 `dispatchDraw()` 函数,应用程序不应该再重载 `ViewGroup` 类中的 `dispatchDraw()` 函数,因为该函数内部已经有了默认的实现,并且该实现代表了 View 系统的内部流程。

`dispatchDraw()` 内部会在一个 `for()` 循环语句中,循环调用 `drawChild()` 分别绘制每一个子视图,而 `drawChild()` 内部又会调用 `draw()` 函数完成子视图的内部绘制工作。当然,如果子视图本身也是一个 `ViewGroup`,就会递归执行以上流程。

从以上设计思路来看,它与 `measure` 及 `layout` 的过程极其相似。

下面就具体来看以上主要函数的内部执行过程。

### 13.10.3 ViewRoot 中 draw() 的内部流程

`ViewRoot` 中的 `draw()` 函数主要处理一些根视图中的特有属性,并且处理完毕后同样要调用 View 类中的 `draw()` 进行具体的绘制。

`Surface` 按照底层的驱动模式可以分为两种,一种是使用图形加速支持的 `Surface`,俗称显卡,另一种是使用 CPU 及内存模拟的 `Surface`。因此,根视图中将针对不同的 `Surface` 采用不同的方式从该 `Surface` 中获取一个 `Canvas` 对象,并将该 `Canvas` 对象派发到整个视图中,对于非根视图而言,它并不区分底层是使用显卡模式,还是使用 CPU 模式。

该函数的内部流程如图 13-34 所示。

① 检查 `Surface` 是否无效。在正常情况下, `Surface` 都是有效的,除非 `WmS` 发生异常不能为该客户端分配有效的 `Surface`, `isValid()` 才会返回 `false`。如果 `Surface` 无效,则终止绘制过程。

② 执行注册过的 `Runnable` 对象。`ViewRoot` 中使用一个静态列表,可以向该静态列表中添加一些 `Runnable` 对象,本步骤则把这些 `Runnable` 对象调用 `post()` 发送到 `Handler` 队列中,以便下次消息循环时处理这些 `Runnable` 对象。这个变量的名称是 `sFirstDrawComplete`,有些读者可能觉得变量名称有点奇怪,为什么是 `Complete` 呢,明明还没有开始绘制?因为该变量中保存的 `Runnable` 对象会在下个消息循环中执行,而执行前,接下来的绘制过程必须先被执行。

③ 调用 `scrollToRectOrFocus()`。几乎在所有的情况下,该函数内部都不会执行什么,所以其内部执行流程忽略。该函数本来的设计目的是对 `mScrollY` 变量进行调整,调整的依据是调整到第一个 `Focus`

视图中。

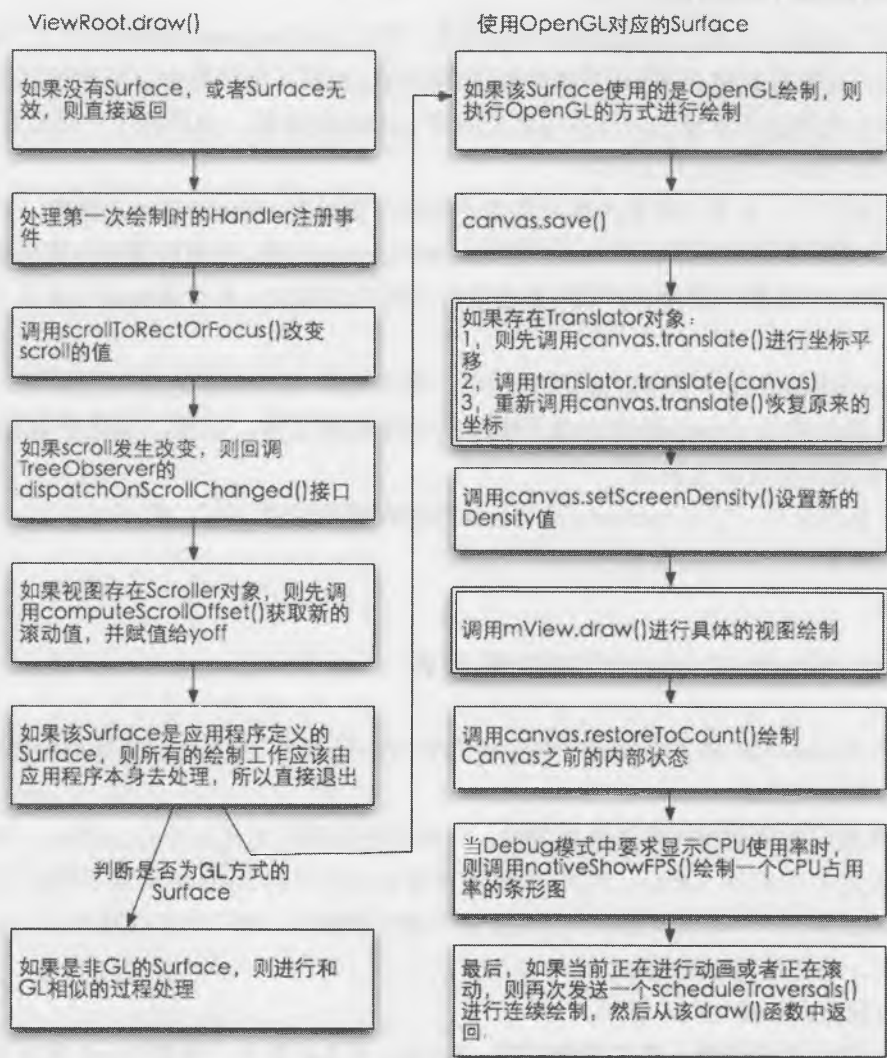


图 13-34 绘制操作的详细过程

④ 如果根视图内部包含 Scroller 对象, 则调用该对象的 computeScrollOffset() 获取新的滚动值, 并赋值给局部变量 yoff, 关于 Scroller 的详细意义将在后面小节中单独介绍。该对象的 computeScrollOffset() 的意义是计算是否发生滚动, 该函数返回值类型为 boolean, 比如, 当用户在一个 ListView 上滑动手指后, 会在一小段时间内发生滚动, ListView 内部有一个 Scroller 对象, 在这个期间调用 computeScrollOffset() 将返回 true。

⑤ 判断该 Surface 是否有 SurfaceHolder 对象。如果有则意味着该 Surface 是应用程序创建的, 因

此所有的绘制操作应该由应用程序自身去负责，于是 View 系统退出绘制。如果不是，才开始 View 绘制的内部流程。

⑥ 如果 Surface 是由 OpenGL 实现的，则开始按照 GL 的处理方式进行处理。

(1) 以全局变量 mGLCanvas 作为 canvas 的值，并调用 canvas.save() 保存该 Canvas 内部的各种属性及状态，因为接下来 View 树内部在绘制的过程中会修改 Canvas 对象的相关属性。

(2) 如果根视图内部包含 Translator 对象，则需要先经过 Translator 对象对该 canvas 对象进行一定的调整。Translator 的作用主要是根据设备的硬件参数对 Canvas 的相关绘制属性进行一定的调整，该函数的内部一般由驱动设计者支持实现。调整的过程由三个函数调用组成，源码中的代码如下：

```

1404         canvas.translate(0, -yoff); |
1405         if (mTranslator != null) {
1406             mTranslator.translateCanvas(canvas);
1407         }

```

这段代码有一个 bug，实际上，该段代码应该改写为：

```

if (mTranslator != null) {
    canvas.translate(0, -yoff);
    mTranslator.translateCanvas(canvas);
    canvas.translate(0, yoff);
}

```

原因是，Translator 对象转换的是整个 canvas，而不仅仅是显示的那部分。因此，canvas.translate() 函数应该是在 if 语句内部，并且调用完 translateCanvas() 函数后，再将 canvas 恢复到转换后的区域。这个逻辑在 View 内部的 draw() 函数中也是相同的，此处这个 bug 之所以不被开发者察觉，其原因是基本上所有已有的使用中，yoff 的值都是 0。

(3) 调用 canvas 的 setScreenDensity() 设置屏幕密度。这里需要区分不同语境中 density 的含义的不同。

首先，定义屏幕密度 (density) 概念的作用是为了让不同分辨率的屏幕显示的视图能够看起来大小相同。在传统的编程方式中，如果用像素指定视图的大小，对于分辨率高的显示器而言，像素密度高，所以相同像素的实际尺寸会变小，于是引入 density 的概念。Android 中 160dpi 的屏幕的 density 值定义为 1，如果屏幕分辨率增加，比如 240dpi，那么 density 的值也就越高，为  $240/160=1.5$ ，而分辨率低的屏幕，比如 120dpi，其密度就低，为  $120/160=0.75$ 。有了 density 后，应用程序可以设置视图的大小单位为 dip，即密度无关像素 (density independent pixel)，从而在绘制视图时，View 系统会根据不同的屏幕分辨率将其换算成不同的像素。

而本步设置的 screenDensity，却是指屏幕的分辨率值。当参数 scalingRequired 为 true 时，该值为 DisplayMetric.DENSITY\_DEVICE，该常量是在系统启动时调用 getProp() 函数获取的设备参数，比如 240、160、120 等；如果 scalingRequired 为 false，那么 screenDensity 将被赋值为 0，0 是一个特殊值，并不是说屏幕分辨率为 0，而是指视图绘制没有指定具体的分辨率，从而在绘制时一个 dpi 将对应一个真实的物理像素。

(4) 调用 mView.draw(canvas)。该步骤才真正启动视图树的绘制过程，注意这里是将 canvas 作为

参数，这也就是为什么应用程序中不能保存这个 Canvas 的原因，因为它是一个临时变量。mView.draw() 实际调用的是 View 类的 draw() 函数，关于其内部流程将在后面小节中介绍。

(5) 完成视图树的绘制后，绘制工作就算结束了，因为调用 Canvas 的 restoreToCount() 将 Canvas 的内部状态恢复到绘制之前，该步骤与前面的 canvas.save() 函数是对称调用的。

(6) 如果是 Debug 模式，并且模式中要求显示 FPS，即 CPU 的使用率，则调用一个 native 函数 nativeShowFPS() 给屏幕上方绘制一个条状的统计图。

(7) 最后，如果屏幕正在滚动，则需要再次发起一个重绘命令 scheduleTrasavals()，以便接着绘制，直到滚动结束，滚动的标志 scrolling 来源于 Scroller 对象的 computeScrollOffset() 函数返回值。

⑦ 如果 Surface 不是 OpenGL 实现的，则开始按照非 GL 的处理方式进行处理。该步骤内部与上一步基本上是相同的，唯一的区别在于如何获得 Canvas 对象。GL 方式中，内部使用 mGlCanvas 全局变量保存 canvas 对象，该变量是在 GL 的初始化时进行赋值的；而非 GL 方式中，Canvas 对象需要调用 surface 对象的 lockCanvas() 获取，其他过程完全相同，此处不再赘述。

至此，ViewRoot 中的 draw() 函数就执行完毕，一次绘制过程也就结束了，下一次的绘制将在下一个消息循环中执行。

#### 13.10.4 View 类中 draw() 函数内部流程

该函数的内部流程正如源码中的注释所讲，分为 5 步。

① 绘制背景。变量 dirtyOpaque 表示 dirty 区是否是不透明，只有透明时才需要绘制背景。Android 中的视图几乎都是透明的，因为视图支持阿尔法通道，所以 dirtyOpaque 总是为 false，所以背景总是需要绘制。如果 View 系统不支持阿尔法通道，那么则不需要绘制背景，因为视图本身会占满整个区域，背景会完全被挡住。

绘制背景时，首先根据滚动值对 canvas 的坐标进行调整，然后再恢复坐标，源码如下：

```

6865         if ((scrollX | scrollY) == 0) {
6866             background.draw(canvas);
6867         } else {
6868             canvas.translate(scrollX, scrollY);
6869             background.draw(canvas);
6870             canvas.translate(-scrollX, -scrollY);
6871         }

```

在以上代码中，为什么需要先调用 translate() 平移 Canvas 的坐标呢？因为对每一个视图而言，Canvas 的坐标原点 (0, 0) 对应的都是该视图的内部区域，如图 13-35 所示。



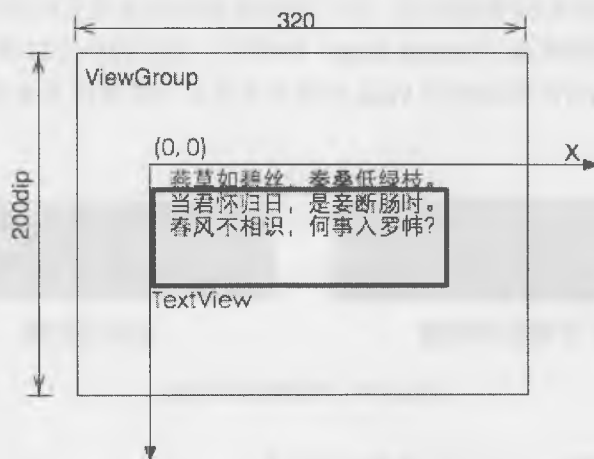


图 13-35 子视图的坐标原点示意图

图中外面是一个任意 ViewGroup 的实例，内部包含一个 TextView 对象，粗实线区域代表该 TextView 在 ViewGroup 中的位置，TextView 中的文字由于滚动，一部分已经超出了粗实线区域，从而不可见。此时，如果调用 `canvas.getClipBounds()` 返回的矩形区域是指粗实线所示的区域，该矩形的坐标是相对其父视图 ViewGroup 的左上角，并且如果调用 `canvas` 的 `getHeight()` 和 `getWidth()` 方法将返回父视图的高度和宽度，此处分别为 200dip 和 320dip。

如果 ViewGroup 中包含多个子视图，那么每个子视图内部的 `onDraw()` 函数中参数 `canvas` 的大小都是相同的，为父视图的大小。唯一不同的是“剪切区”，这个剪切区正是父视图分配给子视图的显示区域。

`canvas` 之所以被设计成这样正是为了 View 树的绘制，对于任何一个 View 而言，绘制时都可以认为原点坐标就是该 View 本身的原点坐标，从而对于 View 而言，当用户滚动屏幕时，应用程序只需要调用 View 类的 `scrollBy()` 函数即可，而不需要在 `onDraw()` 函数中做任何额外的处理，View 的 `onDraw()` 函数内部可以完全忽略滚动值。

由于背景本身针对的是可视区域的背景，而不是整个 View 内部的背景，因此，本步中先调用 `translate()` 将原点移动到粗实线的左上角，从而使得背景 Drawable 对象内部绘制的是粗实线的区域。当绘制完背景后，还需要重新调用 `translate()` 将原点坐标再移回到 TextView 本身的 (0,0) 坐标。

② 如果该程序员要求显示视图的渐变框，则需要先为该操作做一点准备，但是大多数情况下都不需要显示渐变框，因此，源码中针对这种情况进行快速处理，即略过该准备。

③ 绘制视图本身，实际上回调 `onDraw()` 函数即可，View 的设计者可以在 `onDraw()` 函数中调用 `canvas` 的各种绘制函数进行绘制。

④ 调用 `dispatchDraw()` 绘制子视图。如果该视图内部不包含子视图，则不需要重载该函数，而对所有的 ViewGroup 实例而言，都必须重载该函数，否则它也就不是 ViewGroup 了。



⑤ 回调 `onDrawScrollbars()` 绘制滚动条。关于滚动条的详细流程见后面小节。

下面继续分析需要绘制渐变框 (Fading Edge) 的情况, 应用程序可以调用 `setVerticalFadingEdge()` 和 `setHorizontalFadingEdge()` 告诉系统绘制 View 对象的垂直方向渐变框及水平方向渐变框, 渐变框的效果如图 13-36 所示。

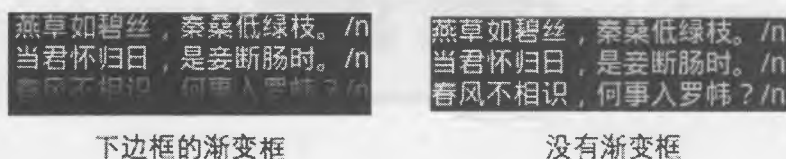


图 13-36 渐变框效果示意

源码中处理渐变框的逻辑中, 定义了以下相关变量。

- `mScrollCache`: 该变量的类型是 `ScrollabilityCache`, 该类中的作用是保存一个缓存对象, 并且该缓存内部定义了一个 `Matrix`、一个 `Shader`、一个 `Paint`, 这三个对象联合起来可以使用 `Paint` 绘制一个 `Shader`, 并且可以在绘制时使用 `Matrix` 对该 `Shader` 进行缩放、平移、旋转、扭拉四种操作, 具体见 `Matrix` 的介绍。
- `length`: 对于垂直方向上的渐变框, `length` 指的是该 `Shader` 的高度, 对于水平方向, 是指 `Shader` 的宽度。
- `xxxFadeStrength`: `xxx` 代表 `left`、`top`、`right`、`bottom`, 该变量将作为后面 `matrix.setScale()` 的参数, 表面意思是“渐变强度”, 但从其效果来看应该是“渐变拉伸度”, 因为 `Shader` 对象内部的原始图像仅仅是一个像素宽, 所以才调用 `matrix.setScale()` 对该像素的 `Shader` 进行缩放 (拉伸), 以产生一个矩形。`xxxFadeStrength` 的范围是 0~1, 源码中使用该变量乘以 `fadeHeight` 或者 `fadeLength`。理解了以上变量的含义后, 剩下的过程就变得简单了, 具体流程如下。

① 得到渐变框的 `length` 值, 如果 `length` 的值大于视图本身的高度, 则需要缩小 `length` 的值, 否则会出现上下渐变重影或者左右渐变重影, 影响视觉效果。

② 回调 `getXXXFadingEdgeStrength()`。该函数一般由 View 的设计者重载, 如前对 `xxxFadeStrength` 变量的解释, 如果该函数返回为 0, 意味着不对 `Shader` 拉伸, 那么也就不会绘制 `Shader` 了。因此, 本步骤正是通过回调这些函数, 从而决定都要绘制上下左右哪些渐变框, 并用四个变量 `drawXXX` 表示, `xxx` 代表 `Left`、`Top`、`Right`、`Bottom`。

③ 获得 `Shader` 渐变的色调。这段代码内部有点差强, 源码设计者的本意是想调用 `canvas.saveLayer()` 对后面的绘制进行缓存, 然而却仅仅设计成当颜色值为 0 时才缓存。在笔者看来, 如果要缓存, 则无论什么颜色都可以缓存, 因此, 那段 `canvas.saveLayer()` 实际上没有什么意义。在一般情况下, 如果颜色不为 0, 则调用 `setFadeColor()` 将该颜色设置到 `mScrollCache` 内部的画笔 (`Paint`) 中, 应用程序可以重载 View 类的 `getSolideColor()` 用于设置渐变色。

④ 和前面的非渐变绘制流程相同，先后绘制视图本身、子视图。

⑤ 此时，才开始真正绘制渐变框，根据第二步中保存的变量 `drawXXX` 分别绘制不同的渐变框，绘制中主要是对 `Matrix` 进行变换，然后调用 `canvas.draw()` 进行绘制。读者可能觉得奇怪，这个 `Matrix` 对象是 `ScrollAbilityCache` 对象内部的，为什么设置后影响的却是 `Canvas` 对象呢？原因就在于 `canvas.draw()` 中最后一个参数 `p`，它是一个 `Paint` 对象，该对象正是来源于 `ScrollAbilityCache` 中的 `Paint`，而该 `Paint` 内部已经和该 `Matrix` 关联了。对 `Matrix` 的变换包含以下四点。

- `matrix.setScale()`：该函数的作用正是把只有一个像素宽度的 `Shader` 缩放成一个真正的矩形渐变框。
- `matrix.postTranslate()`：对坐标进行平移。
- `matrix.postRotate()`：对图形进行旋转
- `fade.setLocalMatrix()`：该调用正是把该 `Matrix` 和该 `Shader` 关联起来。

源码中绘制上渐变框和下渐变框调用以上函数时的执行效果如图 13-37、图 13-38 所示，绘制左右边框与此类似。

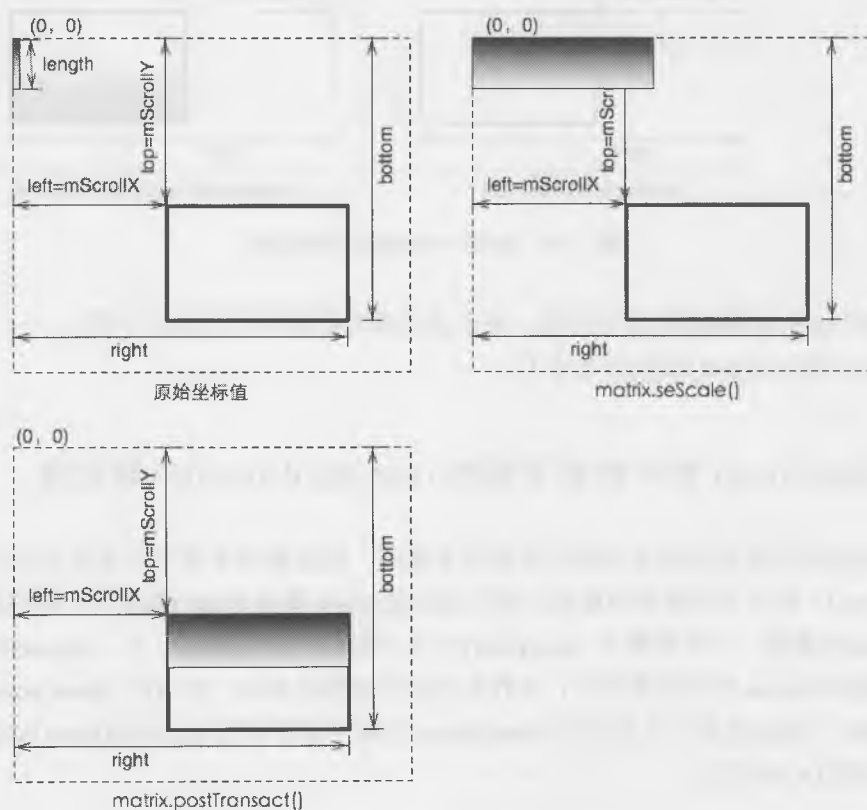


图 13-37 绘制上渐变框的变换过程

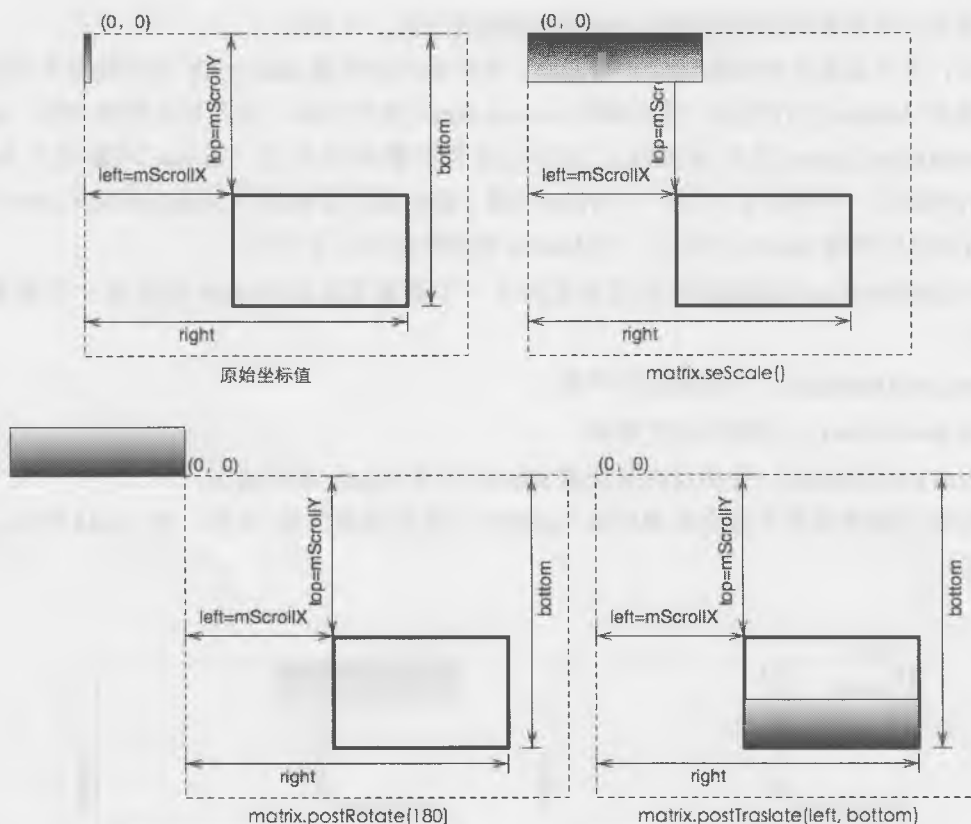


图 13-38 绘制下渐变框的变换过程

- ⑥ 最后调用 `onScrollBar()` 绘制滚动条，关于滚动条的绘制细节见后面小节。至此，View 内部的 `draw()` 就执行完毕了。

### 13.10.5 ViewGroup 类中绘制子视图 `dispatchDraw()` 内部流程

`dispatchDraw()` 的作用是绘制父视图所包含的子视图，该函数的本质作用是给不同的子视图分配合适的画布 (Canvas)，至于子视图如何绘制，则又递归到 View 类的 `draw()` 函数中。应用程序一般不需要重载 `dispatchDraw()` 函数，而只需要在 `onLayout()` 中为子视图分配合适的大小，`dispatchDraw()` 将根据前面分配的大小调整 Canvas 的内部剪切区，并作为绘制子视图的画布。所有的 ViewGroup 实例的内部绘制基本上都是如此，这就是为什么具体的 ViewGroup 实例不需要重载 `dispatchDraw()` 的原因。该函数内部的执行流程如图 13-39 所示。

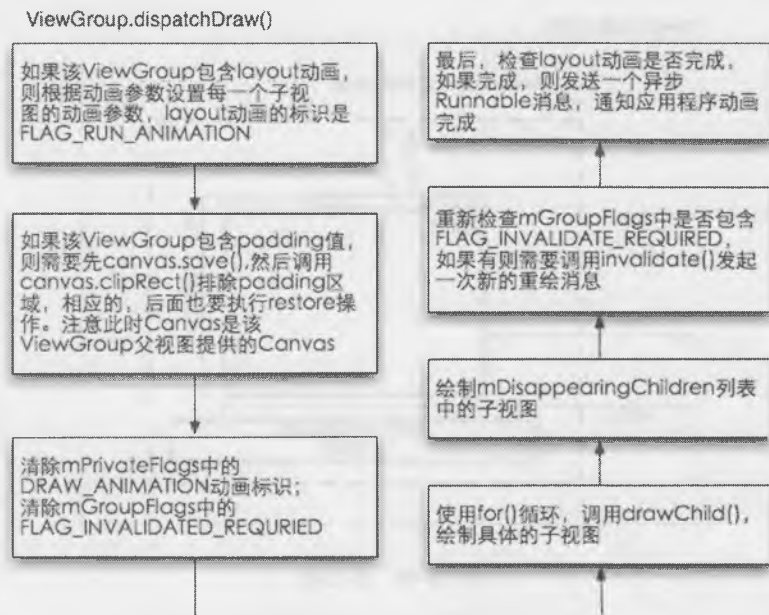


图 13-39 ViewGroup 类中 dispatchDraw() 内部执行过程

1 判断 `mGroupFlags` 中是否设置 `FLAG_RUN_ANIMATION` 标识，该标识并不是该 `ViewGroup` 的“动画标识”，而是该 `ViewGroup` “布局动画标识”。动画标识指的是一个 `View` 自身的动画，而布局动画只存在于 `ViewGroup` 对象中，指的是该 `ViewGroup` 在显示内部的子视图时，为内部子视图整体设置的动画。典型的例子就是，应用程序可以在 XML 文件中的 `LinearLayout` 标签中设置 `android:layoutAnimation` 属性，从而使该 `LinearLayout` 的子视图在显示时出现逐行显示、随机显示、落下等不同的动画效果，而这些效果正是在本步骤实现的。关于动画的详细过程见后面小节，本节只分析没有动画的情况。

2 处理 `padding` 属性。该属性是 `ViewGroup` 特有的，程序员只能给一个 `ViewGroup` 设置 `padding`，而不能给一个 `View` 设置 `padding`。如果 `ViewGroup` 包含 `padding` 值，则 `CLIP_PADDINT_MASK` 标识将存在。对于 `View` 系统而言，当绘制到某个 `View` 时，`View` 系统并不区分该 `View` 是一个具体的 `View` 还是一个 `ViewGroup` 实例，都会在 `View.draw()` 函数中调用 `dispatchDraw(canvas)`，参数 `Canvas` 的绘制区原点坐标是该 `View` 内部区域的左上角，`Canvas` 的剪切区仅仅是根据 `scroll` 值进行了剪切。由于 `padding` 是 `ViewGroup` 所特有的属性，因此 `ViewGroup` 的 `dispatchDraw()` 需要对该属性进行自身的处理。

源码中首先调用 `canvas.save()` 保存当前 `Canvas` 内部状态，然后调用 `canvas.clipRect()` 进行剪切。在执行 `dispatchDraw()` 函数前，`Canvas` 的剪切区已经根据 `scroll` 值进行了剪切，剪切坐标的原点是 `View` 自身的左上角，所以此处仅仅需要从左边加 `paddingLeft`，从上边加 `paddingTop`，从右边减 `paddingRight`，从下边减 `paddingBottom`。本步骤执行前后的位置如图 13-40 和图 13-41 所示。

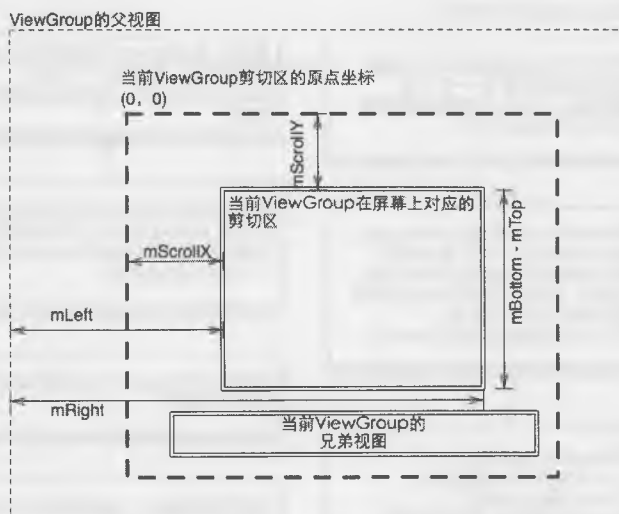


图 13-40 执行前

执行前，View 系统给该 ViewGroup 分配的剪切区为双实线所表示的区域。

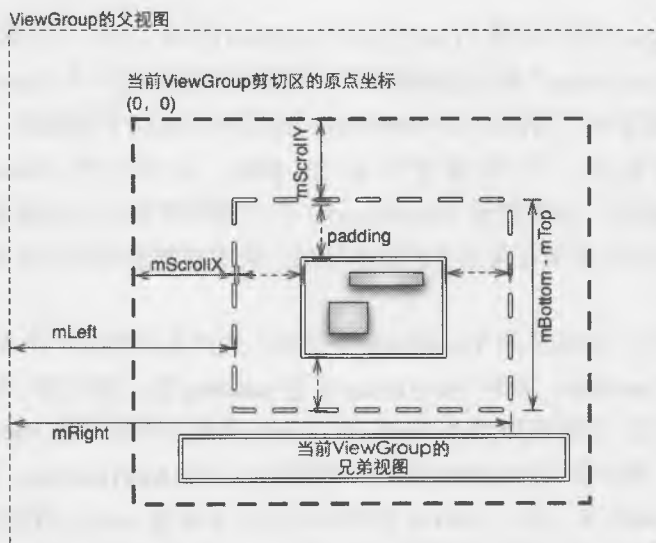


图 13-41 执行后

执行后，就会根据 padding 的值缩小剪切区。这里需要注意，缩小的仅仅是剪切区，也就是用户在屏幕上看到的区域，而 ViewGroup 本身的大小没有变化。

③ 清除 mPrivateFlags 的 DRAW\_ANIMATION 标识，因为接下来就会绘制视图了；同时清除 mGroupFlags 的 FLAG\_INVALIDATED\_REQUIRED 标识，因为接下来绘制后就意味着已经满足

“REQUIRED”这个需求了。

④ 使用 for() 循环, 针对该 ViewGroup 的子视图逐个调用 drawChild() 函数。在一般情况下, 绘制子视图的顺序是按照子视图被添加的顺序逐个绘制, 但应用程序可以重载 ViewGroup 的 getChildDrawingOrder() 函数, 提供不同的顺序。关于 drawChild() 的内部过程见后面小节。

⑤ 绘制 mDisappearingChildren 列表中的子视图。这个变量需要着重解释一下, 当从 ViewGroup 中 removeView() 时, 指定的 View 对象会从 mChildren 变量中移除, 因此, 当进行消息派发时, 被删除的 View 就绝不会获得用户消息。当被删除的 View 对象包含一个移除动画时, 则该 View 会被添加到 mDisappearingChildren 列表中, 从而使得在进行 dispatchDraw() 时, 该 View 依然会被绘制到屏幕上, 直到动画结束, 在动画期间, 用户虽然能够看到该视图, 但却无法点击该视图, 因为它已经从 mChildren 列表中被删除, 消息处理时会认为没有该 View 的存在。

⑥ 重新检查 mGroupFlags 中是否包含 FLAG\_INVALIDATED\_REQUIRED 标识, 因为 drawChild() 调用后, 可能需要重绘该 ViewGroup, 如果需要, 则调用 invalidate() 发起一个重绘请求。

⑦ 本步骤与第 ① 步是对称的, 第 ① 步中会先处理“布局动画”, 而本步骤则处理布局动画是否完成, 如果完成, 发送一个 Handler 消息。该消息是一个 Runnable 对象, 其作用是回调 ViewGroup 中 AnimationListener 接口的 onAnimationEnd() 函数, 通知应用程序布局动画完成了。

至此, ViewGroup 的 dispatchDraw() 就结束了。

### 13.10.6 ViewGroup 类中 drawChild() 过程

drawChild() 的核心过程是为子视图分配合适的 Canvas 剪切区, 剪切区的大小取决于 child 的布局大小, 剪切区的位置取决于 child 的内部滚动值及 hild 内部的当前动画。该函数内部的主体流程如图 13-42 所示, 分为八大步。

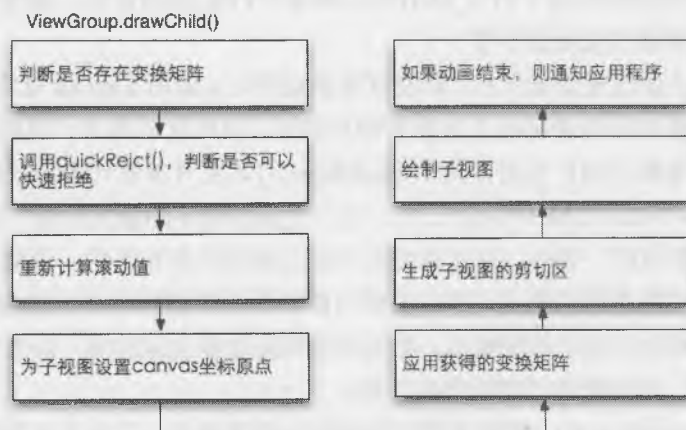


图 13-42 drawChild() 内部主体流程

① 判断该子 View 是否存在“变换矩阵”。所谓的“变换矩阵”是指该 child 在绘制时将通过一个矩阵 (Matrix) 进行变换, 变换的元素包括四点, 分别为平移、旋转、缩放、扭曲, 该矩阵就称之为变换矩阵。存在变换矩阵意味着, 该图像变换后会改变边框的大小。可以通过两种方式为视图设置变换矩阵, 一种是动画, 另一种是静态回调。

所谓动画是指, 应用程序可以调用 View 类的 `setAnimation()` 为该 View 设置一个动画。动画的本质是对 View 视图在指定的时间内进行某种变换 (Transformation), 变换包括图像平移、旋转、缩放、扭曲及图像颜色阿尔法通道变化, 然后将变换后的图像绘制到屏幕上, 系统会在指定的时间内连续进行绘制, 并在不同时间得到不同的变换参数, 从而使其看起来就像是一个“动画”。动画中又使用一个 Transformation 类来保存这些变换参数, Transformation 类是一个数据类, 内部包含变换的相关参数, 变换按照类型分为四种:

- TYPE\_IDENTIFY: identify 的意思是“相同的”, 即该变换实际上不会引起任何变换。
- TYPE\_ALPHA: 将引起图像颜色阿尔法通道变换。
- TYPE\_MATRIX: 将引起矩阵变换, 矩阵变换包括平移、旋转、缩放、扭曲四种。
- TYPE\_BOTH: both 的含义是同时包含 ALPHA 和 MATRIX。

所谓的静态回调是指, ViewGroup 实例的设计者可以重载 ViewGroup 类的 `getChildStaticTransformation()` 函数, 从而为其包含的子视图指定一个静态的变换对象。

本步骤中包含两个重要局部变量。

- Transformation transformToApply: 保存了子视图的变换对象, 可来源于动画, 也可静态指定。源码中首先判断是否存在动画, 如果存在动画, 则调用 Animation 对象的 `getTransformation()` 获取变换对象; 如果没有动画, 则回调 `getChildStaticTransformation()` 获取变换对象。
- boolean concatMatrix: 该变量代表是否存在变换矩阵。对于动画而言, 调用 Animation 对象的 `willChangeTransformationMatrix()` 判断是否存在变换矩阵; 而对于“静态回调”, 则直接判断变换的类型, 只有当变换类型是 TYPE\_MATRIX 或者 TYPE\_BOTH 时, 该变量才为 true。

关于动画的更多变换细节见后面小节。

② 如果以上变换不会改变边框大小, 即没有变换矩阵时, 调用 Canvas 对象的 `quickReject()` 函数快速判断该子视图对应的剪切区是否超出了父视图的剪切区, 超出意味着该子视图不能显示到屏幕上, 所以就不用绘制了, 因为绘制了用户也看不见。`quickReject()` 调用时参数代表该子视图在父视图中的布局 (layout) 位置。注意本步成立的条件包含三个, 这三个条件最终的意义是指, 该 View 内部没有进行动画, 并且不存在“静态回调”变换, 并且剪切区不在父视图的剪切区中。该意义的反义是指, 如果当前正在进行动画或者存在静态回调变化, 那么就算当前视图的剪切区不在父视图的剪切区中, 都要进行绘制操作。为什么呢? 因为存在矩阵变换后, 会引起子视图边框位置改变, 而改变后的区域有可能又落到了父视图的剪切区中, 从而变成“可看得见”的。

③ 回调 `child.computeScroll()`, 重新计算子视图的当前滚动值, 因为子视图的滚动值在每次绘制之后都有可能变化。应用程序一般会重载 View 对象的 `computeScroll()` 函数, 比如对于 ListView, 当用户



手指在屏幕上滑动时，将导致屏幕做惯性滚轮运动，而在这个运动的过程中，`computeScroll()`函数会不断改变该 View 的滚动值。

④ 上一步得到了子视图的滚动值，本步就要根据该滚动值设置子视图 Canvas 坐标的原点。对当前的 Canvas 而言，其坐标原点是该 ViewGroup 布局区域的左上角①点，如图 13-43 所示，而本步正是要将这个坐标原点移动到指定子视图的自身显示区域的左上角②点。

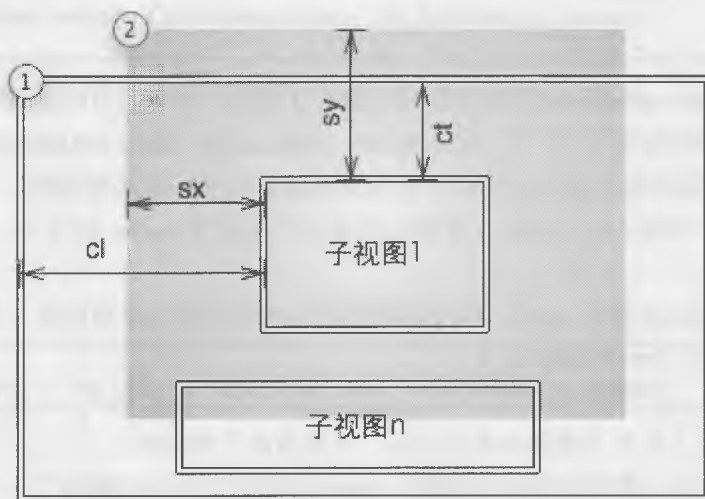


图 13-43 坐标变换示意

源码中，针对是否有 `cache` 的情况分别处理。在一般情况下，没有 `cache`，所以 `translate()` 参数中水平方向是先向右移动 `cl`，然后再向左移动 `sx`，垂直方向类似；而对于有 `cache` 的情况，则忽略滚动值，因为有 `cache` 时，视图本身需要处理滚动值，实际上如果有 `cache`，视图的滚动值都会设置为 0，因此水平方向仅平移 `cl`，垂直方向仅平移 `ct`。

⑤ 上面第一步得到了变换对象 `transformToApply`，本步就要将该变换对象应用于子视图。首先判断是否存在变换矩阵 `concatMatrix`，并应用变换矩阵，然后再应用视图颜色阿尔法变换。

变换矩阵针对的是视图本身的整个区域，而不仅是在屏幕上的显示区域，但是变换矩阵中的数据却是相对子视图可视区域的左上角，因此，在变换前需要先将坐标原点调整到视图本身的左上角，然后再应用变换，最后再将原点调整回子视图本身的左上角，如以下代码所示：

```
1600 // Undo the scroll translation, apply the transformation matrix,
1601 // then redo the scroll translate to get the correct result.
1602 canvas.translate(-transX, -transY);
1603 Log.i("ViewGroup", "matrix = " + transformToApply.getMatrix());
1604 canvas.concat(transformToApply.getMatrix());
1605 canvas.translate(transX, transY);
1606 mGroupFlags |= FLAG_CLEAR_TRANSFORMATION;
```

针对视图颜色的阿尔法变换主要是调用 `canvas.saveLayoutAlpha()` 函数完成，在调用该函数前先回调

子视图的 `child.onSetAlpha()`。

⑥ 此时 `Canvas` 内部的变换参数已经确定，子视图的滚动值也确定了，因此也就可以确定子视图中剪切区的位置了。同样，剪切区也分是否有 `cache` 的情况，因为如果有 `cache` 的话，系统将认为子视图没有滚动值，滚动的处理完全由子视图内部控制，所以剪切区将忽略滚动值，剪切算法如下：

```
1631         if (!scalingRequired) {
1632             canvas.clipRect(0, 0, cr - cl, cb - ct);
1633         } else {
1634             canvas.clipRect(0, 0, cache.getWidth(), cache.getHeight());
1635         }
```

在以上代码中，`scalingRequired` 变量是指是否进行了缩放。如果没有缩放，则边框没有变化。边框的位置就等于该子视图的布局位置，即 `child.mLeft`、`child.mTop`、`child.mRight` 及 `child.mBottom`，此时滚动值一般都为 0，所以实际的剪切区就等于子视图布局的大小。如果有缩放，比如缩放大于 1，则子视图的显示边框将大于布局边框，因此，剪切区的大小应该使用 `cache` 的大小，即 `cache.getWidth()` 及 `cache.getHeight()`。

而在一般情况下视图都没有 `cache`，因此需要根据滚动的位置确定剪切区，剪切算法如下：

```
1628         if (hasNoCache) {
1629             canvas.clipRect(sx, sy, sx + (cr - cl), sy + (cb - ct));
```

该算法的结果实际上就是子视图的布局大小，只是考虑了滚动而已。

⑦ 剪切区设置好后，就可以调用子视图的 `draw()` 函数进行具体的绘制了。同样，源码中分别针对是否有 `cache` 的情况做了不同处理。

一般情况下没有 `cache`，那么最简单的过程就是直接调用 `child.draw()` 即可，只是在调用之前先判断子视图的 `mPrivateFlags` 是否包含 `SKIP_DRAW` 标识，该标识告知 `View` 系统暂时跳过对该子视图的绘制。如果需要跳过，则仅调用 `child.dispatchDraw()`，即跳过视图本身的绘制，但要绘制视图可能包含的子视图。

如果有 `cache`，则只需要把视图对应的 `cache` 绘制到屏幕上即可，即调用 `canvas.drawBitmap()` 函数，参数中包含缓冲区 `cache`，它实际上是一个 `Bitmap` 对象。

⑧ 最后，绘制到此就结束，结束前，需要先恢复 `Canvas` 绘制前的状态，并且如果该绘制是一次动画绘制，那么当该动画结束，则调用 `finishAnimatingView()` 通知应用程序该子视图动画绘制完成了。

至此，针对该子视图的一次绘制就结束了。

### 13.10.7 绘制滚动条

绘制滚动条是通过在 `View` 类的 `draw()` 函数中调用 `onDrawScrollBar()` 完成的。每个视图都可以有滚动条，请注意是“每个”，因为滚动条是视图中包含的基本元素，就像视图的背景一样。举个例子，一个按钮（`Button`）也可以有滚动条，读者可能觉得奇怪，按钮怎么会有滚动条呢？但的确是这样，而且让按钮显示滚动条是件非常容易的事情，因为按钮本身也是一个视图，只是从用户的角度来讲，按钮不

需要显示这个滚动条而已。

滚动条包含垂直滚动条和水平滚动条，滚动条本身是一个 Drawable 对象，View 类内部使用 ScrollBarDrawable 类表示滚动条，View 可以同时绘制水平滚动条和垂直滚动条。

在 ScrollBarDrawable 类中，包含三个基本尺寸，分别是 range、offset、extent。

- range：代表该滚动条从头到尾滚动中所跨越的范围有多大。比如想用个滚动条标识一万行代码，那么 range 可以设为 10000。
- offset：代表滚动条当前的偏移量。比如当前在看第 600 行代码，那么 offset 就是 600。
- extent：代表该滚动条在屏幕上的实际高度，比如 200，单位是 dip。

有了以上三个尺寸后，ScrollBarDrawable 内部就可以计算出滚动条的高度及滚动条的位置。

除了以上尺寸外，ScrollBarDrawable 类内部还包含两个 Drawable 对象，一个标识滚动条的背景，另一个标识滚动条自身。这两个 Drawable 对象分别是 track 和 thumb，水平方向和垂直方向分别有两个该 Drawable 对象。

以上尺寸及 Drawable 对象的显示关系如图 13-44 所示。

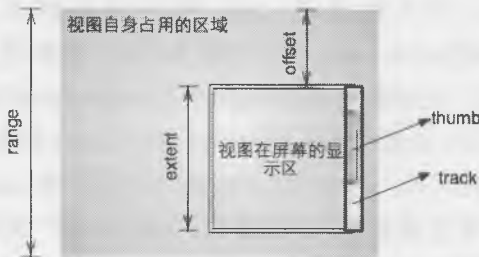


图 13-44 视图滚动条示意

本节仅分析垂直方向的滚动条绘制过程，水平方向与此类似。该绘制过程中相关的类关系如图 13-45 所示。

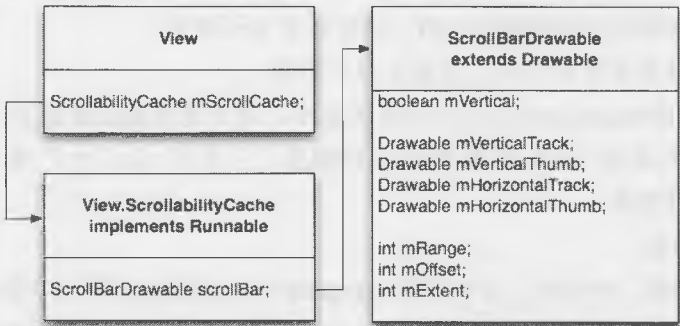


图 13-45 滚动条绘制相关的类关系

onDrawScrollBar()函数内部的执行流程如下。

① 判断该 View 对象内部是否存在 ScrollabilityCache 对象，即变量 mScrollCache 是否为空。对应用程序而言，当需要视图显示滚动条时，可以在 XML 文件中使用 android:scrollbars 属性为视图指定滚动条，属性值可以是 vertical、horizontal、none，从而在 View 的构造函数中将会调用 initializeScrollbars() 创建一个 ScrollabilityCache 对象。如果 View 对象不是从 XML 文件中产生，而是通过程序动态产生，则可以调用 View 类的 setScrollbarFadingEnable(boolean) 函数设置滚动条是否“自动隐藏(fading)”，该函数内部则会调用 initializeScrollbars() 初始化 ScrollabilityCache 对象。什么是“自动隐藏”？Android 中的滚动条和 PC 上的滚动条有所不同，对于 PC 上的滚动条而言，在一般情况下滚动条会一直存在，在默认情况下滚动条并不显示，只有当用户做滚动操作时才会显示，滚动完毕后，滚动条就会自动隐藏起来，这就叫自动隐藏。应用程序可以调用 setScrollbarFadingEnable() 设置是否自动隐藏，在默认情况下是自动隐藏。

② 如果 cache 存在，则继续判断 cache 的状态 (state)。ScrollabilityCache 内部有两种状态，分别为 ON 和 OFF，ON 意味着滚动条处于显示状态，OFF 意味着滚动条处于隐藏状态。因此，本步骤判断如果是 OFF 状态的话，就直接返回。

那么，什么时候是 ON，什么时候是 OFF 呢？如果滚动条是自动隐藏，那么，在 setScrollbarFadingEnable() 函数中会将该 cache 的状态置为 OFF，否则置为 ON。如果是自动隐藏，那么当应用程序调用 scrollBy() 函数时，该函数内部会间接调用 awakenScrollBars()，该函数中会把 cache 的状态“暂时置”为 ON。为什么这里是暂时呢？因为该函数将 cache 状态置为 ON 后，紧接着会发送一个异步延迟消息，在指定的延迟时间后，消息的处理函数又会重新将 cache 的状态置为 OFF，从而使得在下次绘制滚动条时会在本步骤中直接返回，这也就是“自动隐藏”的具体过程。

awakenScrollBars() 函数的类型是 protected，意味着该函数只能被重载，而不能被应用程序直接调用，应用程序一般只能调用 scrollBy() 函数。而对于自定义 View 而言，比如 ListView，其内部实际上并没有滚动值，所以也就不能调用 scrollBy() 函数，而它实现滚动条自动隐藏的效果正是借助于 awakenScrollBars() 函数。

③ 判断 cache 的状态是否为 FADING。FADING 的本质依然是 ON，只是它“正在隐藏”，所谓“正在隐藏”的效果一般就是滚动条“逐渐消失”，其实现方法是逐渐减少滚动条的阿尔法通道值。如果不是 FADING，则将阿尔法值设为 0xff，也就是完全不透明。

④ 如果存在水平或者垂直滚动条，则逐个进行绘制。

(1) 首先调用 scrollbar.getSize() 获得滚动条的大小。对于垂直滚动条而言，大小就是指 track 或者 thumb 的宽度；水平方向是指 track 或者 thumb 的高度，只有当 size 小于 0 时，才会使用 XML 中 android:scrollbarSize 属性的值。

(2) 绘制水平滚动条。

(3) 绘制垂直滚动条。绘制时，首先回调 computeVerticalRange() 等三个函数，获得当前的 range、offset 及 extent 值，然后将这三个值设置到 scrollbar 中。这就是为什么滚动条会“滚动”的原因。自定义视图的设计者应该重载这三个 computeXXX() 函数，并在函数实现中根据自定义的滚动情况返回相应

的值，从而使得 View 系统能够绘制出具有正确位置的滚动条。源码中有一段注释是关于 RTL 语言的，RTL 是指从右向左阅读的语言。设置完这三个值后，回调 `onDrawVerticalScrollBar()` 函数，该函数的代码如下：

```
6034 scrollBar.setBounds(l, t, r, b);
6035 scrollBar.draw(canvas);
```

即首先设置滚动条对应的剪切区，然后调用 `scrollBar` 的 `draw()` 函数将其绘制到剪切区中。

⑤ 以上步骤已经完成了一次绘制，但如果当前滚动条正处于滚动状态，则需要继续调用 `invalidate()` 发起一次重绘消息。而判断是否处于滚动状态的变量是 `invalidate`，其值正是当 `cache` 状态为 `FADDING` 时被赋值为 `true`。调用 `invalidate()` 时，参数对应的矩形区仅仅是滚动条所在的区域。

至此，滚动条绘制就完成了。为了便于以后应用程序开发参考，表 13-7 列出了和滚动条相关的程序代码。

表 13-7 控制滚动条的相关代码列表

相关对象	关注内容
XML 文件中	<ul style="list-style-type: none"> <li>● <code>android:scrollBars</code>: 指定垂直和（或）水平滚动条</li> <li>● <code>android:scrollBarStyle</code>: 滚动条样式</li> <li>● <code>android:scrollBarFadeDuration</code>: 逐渐消失所用的时间</li> <li>● <code>android:scrollBarSize</code>: 滚动条的大小</li> <li>● <code>android:scrollBarTrack</code>: track 对应的 Drawable 对象</li> <li>● <code>android:scrollBarThumb</code>: thumb 对应的 Drawable 对象</li> </ul>
View 调用者	<ul style="list-style-type: none"> <li>● <code>scrollBy(int dx, int dy)</code>: 使视图内部滚动指定值</li> </ul>
View 重载者	<ul style="list-style-type: none"> <li>● <code>computeHorzontalScrollXXX()</code>, XXX 代表 <code>Range</code>、<code>Offset</code>、<code>Extent</code>，重载者必须重载这三个方法并返回合适的位置</li> <li>● <code>awakenScrollBars()</code>，重载者可以调用该函数显示滚动条</li> </ul>

## 13.11 动画的绘制

动画就是让“画”动起来，其原理就像电影的胶片，通过不断在荧屏上绘制不同静态图像，从而达到动画的效果。GUI 系统中动画的本质也是这样，在 View 类的 `draw()` 函数中，会判断当前视图是否包含动画，如果包含，就根据动画的参数对当前 View 做一定的图形变换，比如缩放、平移等，然后将变换后的图像绘制到屏幕上。绘制完后，再发起一个重绘的消息，就这样连续绘制，直到动画参数指示动画结束。

从以上过程来看，实现动画的难点在于如何从程序的角度抽象“动画”，以及动画中所包含的参数应该如何定义。

从效果的角度看，View 系统中包含的动画可以分为三类，分别是窗口动画、视图动画、布局动画。

- 窗口动画：指窗口对应的动画。窗口可以是一个 Activity 对应的窗口，也可以是一个对话框对应的窗口，还可以是应用程序调用 WindowManager 类的 `addView()` 函数添加的任意窗口。窗口动画一般定义了窗口在显示、消失时的动画，关于窗口动画的实现方式见第 14 章。
- 视图动画：指 View 对象在显示及消失时对应的动画，它影响的是视图自身的动画效果。
- 布局动画：指 ViewGroup 对象包含的动画。该动画定义了 ViewGroup 中子视图第一次显示时的动画，它影响的是 ViewGroup 中子视图的整体动画效果，其本质过程是根据布局动画为子视图分别设置不同的动画，从而使得整体上看来像是一个布局动画的效果。所以说，布局动画仅仅是一个概念。

本节首先介绍 Android 中动画的设计思路，以及主要功能类的关系，然后分别介绍视图动画及布局动画的绘制细节。

### 13.11.1 动画的设计思路

首先需要明确一点，动画是一个比较复杂的事情，不同的动画背后都必须有一个特定的数学模型与之对应。简单的动画（比如匀速移动），其数学模型比较简单，只需要知道移动前后的位置，以及移动的速度即可。而对于复杂的动画（比如一个人打太极拳），这个过程基本上不能用一个数学模型表示，于是就使用录像录制整个过程，从而产生动画。

对于 View 系统而言，所能提供的动画更是有限，Android 中 View 系统仅支持基本的五种动画，分别为平移、缩放、旋转、扭曲及颜色阿尔法通道变化，这称之为“动画参数”或“动画类型”。应用程序也可以将这五种基本动画进行组合以产生新的动画，但也仅限于此。

View 系统中动画的设计思路如下。

首先要有一个动画主体，实际上就是一个 View 对象，然后可以为该 View 对象指定一个动画。动画使用一个 Animation 类来表示，当 View 要开始动画时，从 Animation 类中获取动画的参数，并根据这些参数对 View 进行图形变换，然后将变换后的图形绘制到屏幕上。

Animation 类中会保存动画的起始时间，并且在动画开始后，Animation 会在不同的时间返回不同的动画参数，从而使得 View 在随后的时间中会变换出不同的图像。View 系统会连续从 Animation 中取出动画参数，并将变换后的图像绘制到屏幕上，直到动画结束。而对用户来讲，这个过程感觉上就是 View 在变换，也就是“动画”。

Animation 类是一个 abstract 类型，它仅仅定义了动画和 View 类的 API 接口，至于要提供具体什么样的动画参数，则需要继承 Animation 类，并实现所定义的 API。不同动画的设计者可以在定义的 API 中返回不同的动画参数，从而产生不同的动画效果。

除了 Animation 类，与动画相关的还有一个重要类 Interpolator。该类是一个 interface，它的作用是什么呢？如上所述，假设给一个 View 指定了一个移动动画，动画的参数中包括起始位置 X，目标位置 Y，起始时间  $t_0$ ，终止时间  $t_1$ ，在默认情况下，动画开始后，会在  $t_0$  到  $t_1$  时间段上做匀速运动。再举



一个例子，假设给一个 View 指定了一个旋转动画，动画的参数中包含起始角度 A0，终止角度 A1，起始时间 t0，终止时间 t1，在默认情况下，动画开始后，会在 t0 到 t1 时间段上做匀速旋转。这两个例子中，无论是哪种动画类型，从时间轴上来看都是匀速的，而从应用的角度来看，往往需要非匀速的变换，而如何实现非匀速变换呢？这正是 Interpolator 类的作用，程序员可以实现 Interpolator 接口中定义的唯一一个函数 `getInterpolation()`，该函数的原型如下：

```
float getInterpolation(float input);
```

参数 `input` 代表的是时间轴上的 t0 到 t1，`input` 的范围是 0~1，即把 t0 到 t1 时间段进行归一化。比如当 t0 为 2s，t1 为 12s，当前时间为 3s，`input` 的值就是  $(t-t_0)/(t_1-t_0) = (3-2)/10$ ，即 0.1。至于 `getInterpolation()` 的返回值应该是多少，则取决于该函数的实现者了。对于匀速变换而言，该函数直接返回 `input` 即可；而对于非匀速移动，比如加速运动，则返回值和 `input` 的关系如图 13-46 所示，时间点 t 上对应的返回值等于  $(y-y_0)/(y_1-y_0)$ 。不同的加速曲线会产生不同的返回值，程序员甚至可以在不同的时间段使用不同的加速曲线，从而达到特别的动画效果，这实际上是一个数学模型。

另外，在动画设计中还包含一个重要数据类 Transformation，正如其名称所指，该数据类保存了 Animation 中的“动画参数”。针对前面所讲的五种动画参数，Transformation 类分别用以下变量进行保存。

- Matrix mMatrix：该矩阵变量中保存了旋转、缩放、移动、扭曲相关的变换参数。
- float alpha：该变量保存了颜色阿尔法通道的值。

Animation 类中至少需要重载的 API 有三个，如表 13-8 所示。

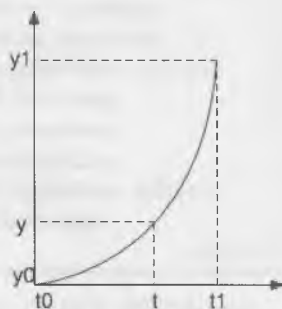


图 13-46 动画过程的加速曲线

表 13-8 一个具体 Animation 类必须重载的函数列表

函数名称	作用
<code>applyTransformation(float interpolatorTime, Transformation t)</code>	View 系统回调该函数，希望 Animation 的设计者根据需要改变参数 t 中的相关属性，返回后 View 系统将使用改变后的 Transformation 对视图进行变换
<code>boolean willChangeTransformationMatrix()</code>	该 Animation 是否会改变 Matrix 值，一般只有缩放、扭曲、移动、旋转才会改变 Matrix 值
<code>boolean willChangeBounds()</code>	该 Animation 是否会改变边框的大小，凡是变换后的图像超出视图的原来边框的，该函数都应该返回 true

Animation 的具体实现类名称一般为 XXXAnimation，其中 XXX 代表变换的名称，比如 AlphaAnimation 类可以实现 Alpha 变换，RotateAnimation 可以实现旋转变换，等等。



### 13.11.2 ViewGroup 类中 drawChild()函数中视图动画绘制过程

视图动画是在 ViewGroup 类中 drawChild()函数中完成的。drawChild()函数首先会判断该视图是否包含 Animation 对象, 如果包含, 则利用 Animation 中的动画参数对该视图进行变换, 然后把变换后的图像绘制到 canvas 中。

应用程序可以使用 res/anim/xxx.xml 文件描述一个动画, 对于视图动画而言, 一般使用如下的 xml 标签描述一个动画。

```

24 <set xmlns:android="http://schemas.android.com/apk/res/android"
25     android:interpolator="@android:anim/decelerate_interpolator"
26     android:zAdjustment="top">
27     <scale android:fromXScale="1.0" android:toXScale=".5"
28         android:fromYScale="1.0" android:toYScale=".5"
29         android:pivotX="50%p" android:pivotY="50%p"
30         android:duration="@android:integer/config_mediumAnimTime" />
31     <alpha android:fromAlpha="1.0" android:toAlpha="0"
32         android:duration="@android:integer/config_mediumAnimTime"/>
33 </set>

```

<set>标签用于包含一组动画参数, 具体的动画参数分为四种, 分别如下。

- <scale>标签: 表示缩放动画, 标签中可以使用 fromXScale 等属性指定缩放的相关值。
- <alpha>标签: 表示颜色阿尔法通道的变换。
- <rotate>标签: 表示旋转变换。
- <translate>标签: 表示平移变换。

关于使用 XML 文件描述动画变换的更多细节参见笔者的另一本书《Android 程序开发》。

描述好了动画后, 程序中可以调用 AnimationUtils.loadAnimation()函数从这个 XML 文件中产生一个 Animation 对象, 然后可以调用 View 类的 setAnimation()把这个动画设置给该视图。以后, 当需要该视图开始指定的动画时, 就可以调用 View 类的 startAnimation()。startAnimation()函数内部实际上只做了一件事情, 即调用 invalidate(), 这就发起了一次重绘请求, 剩下的过程就是在 ViewGroup 类中的 drawChild()函数中完成的。

drawChild()中和动画相关的代码从第 1496 行的 if(a != null)开始到第 1552 行的 if 条件结束, 其具体流程如下。

- ① 创建一个矩形区 region, 该矩形区将保存视图经过动画变换后的大小。
- ② 查看动画对象是否已经初始化。这里所谓的“初始化”的本质是, 在 Animation 类内部有一个 boolean 类型的 mInitialized 变量, 当调用 Animation 的 reset()方法后, 该变量将被重新置为 false, 而是否初始化实际上是指该变量是否为 true。

如果还没有初始化, 则:

- 先调用 `a.initialize()` 初始化该动画对象, 初始化的参数中包含了子视图的宽度、高度, 以及父视图的宽度和高度。
- 接着调用 `a.initializeInvalidateRegion()` 设置初始无效区, 初始值即为子视图的大小。
- 接着调用 `child.onAnimationStarted()` 通知应用程序 “视图动画就要开始了”。

3 如前所述, `Animation` 类内部使用一个 `Transformation` 对象来保存动画参数, 因此本步调用 `a.getTransformation()` 获得该对象, 并赋值给局部变量 `transformToApply`, 变量的语义是 “将要应用给子视图的变换”。注意, 该调用是通过参数传递的, 而并非通过返回值。`getTransformation()` 的返回值仅仅是一个 `boolean` 类型, 指明该动画是否结束, 返回 `true` 代表着还没有结束, 没有结束的话, 执行完本次绘制后, 还需要再调用 `invalidate()` 发起一个绘制消息。

4 判断本次变换是否会改变变换中 `Matrix` 的值, 并赋值给局部变量 `concatMatrix`。什么变换会改变 `Matrix` 呢? 平移、缩放、旋转、扭曲等变换都会改变 `Matrix` 的值。由于这些变换针对的是子视图中整个区域, 包括在屏幕上看得见的以及由于剪切区限制而看不见的, 因此, 在后面的对子视图的变换中将根据是否改变 `Matrix` 而做不同的处理。

5 如果第 3 步调用 `getTransformation()` 返回 `false`, 则意味着动画已经结束了, 于是就不需要再调用 `invalidate()` 发起重绘请求, 否则, 就要发起重绘请求。

`invalidate()` 函数的参数代表了需要重绘的区域, 如果动画变换不改变子视图边框大小的话, 这个区域就是子视图在父视图中的大小, 而如果动画变换超出了子视图本来的大小, 则重绘的区域就需要扩大。源码中先调用 `a.willChangeBounds()` 判断是否本次变换会改变边框大小, 然后再调用 `a.getInvalidRegion()` 计算出本次动画变换后的区域 `region`。该 `region` 相对的坐标原点是子视图的左上角, 因此, 当变换后的区域增加时, `region` 的 `left` 就是一个负值。得到 `region` 后, 就可以根据 `region` 的值重新设定需要重绘的区域。

至此, 动画的参数设置就结束了, `drawChild()` 函数内部之后就要使用这些动画参数对 `canvas` 进行一定的变换, 即判断 `transformToApply` 是否为空。如果不为空, 则用该对象中的 `Matrix` 对 `canvas` 进行变换, 如以下代码所示:

```
1602 canvas.translate(-transX, -transY);
1603 Log.i("ViewGroup", "matrix = " + transformToA
1604 canvas.concat(transformToApply.getMatrix());
1605 canvas.translate(transX, transY);
1606 mGroupFlags |= FLAG_CLEAR_TRANSFORMATION;
```

这段代码首先将 `canvas` 的坐标原点移动到当前显示区, 因为变换仅针对子视图的显示区, 然后调用 `concat()` 进行变换, 变换后再将坐标原点重新移动到滚动后的状态。

进行动画参数变换后, 接下来就需要设置子视图的剪切区, 如以下代码所示:

```

1628         if (hasNoCache) {
1629             canvas.clipRect(sx, sy, sx + (cr - cl), sy + (cb - ct));
1630         } else {
1631             if (!scalingRequired) {
1632                 canvas.clipRect(0, 0, cr - cl, cb - ct);
1633             } else {
1634                 canvas.clipRect(0, 0, cache.getWidth(), cache.getHeight());
1635             }
1636         }
1637     }

```

这段代码初看起来有点让人费解,大家都知道,当某个子视图进行放大动画时,该视图在屏幕上的显示区域会大于原来的区域,而在以上代码中, `canvas.clipRect()` 参数中指定的大小却始终是该子视图的原始大小,这岂不怪哉? 事实上,读者需要注意, `clipRect()` 最终剪切的区域取决于两点,第一点是 `clipRect()` 参数中指定的矩形区域,而第二点却是当前 `canvas` 内部的 `Matrix` 值。这就是为什么前面使用 `canvas.concat()` 的原因, `concat()` 函数能够用指定的 `Matrix` 和 `canvas` 原有的 `Matrix` 进行矩阵相乘,所得 `Matrix` 中的 `scale` 属性将用于最终决定剪切区的大小。比如,以下代码 `clipRect()` 指定的矩形高度为 20, 宽度为 100, 而在调用 `clipRect()` 前却设置了 `canvas` 中的 `scale` 属性为 (1, 2), 因此最终获得的剪切区矩形高度将为 40, 宽度为 100。

```

Matrix m = new Matrix();
m.setScale(1, 2);
canvas.concat(m);
canvas.clipRect(20, 0, 120, 20);

```

计算出剪切区后,剩下的具体绘制就和绘制普通视图完全相同了。

动画执行过程中的消息处理逻辑有以下特点。

第一,动画被变换后尽管显示的大小发生了变换,但其在消息处理过程中所占据的窗口大小却并没有改变,因此,在动画的运行过程中,该视图依然会获取用户消息。比如当对一个 `Button` 对象进行缩放动画时,在动画的过程中,用户依然可以点击该 `Button`,但是点击的区域并不是看到的区域,而是 `Button` 在父视图中的布局区域,一般为动画前或动画后看到的 `Button` 所占的区域,如图 13-47 所示。

第二,由于动画过程是逐个绘制子视图,因此,后面子视图的动画绘制区域可能覆盖或部分覆盖前面子视图动画绘制区域,如图 13-48 所示。

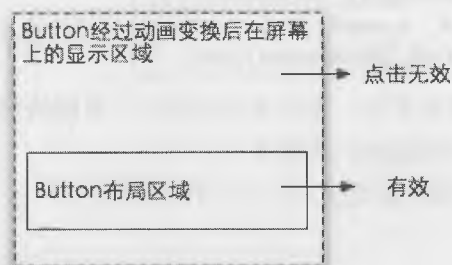


图 13-47 动画引起的实际屏幕显示区

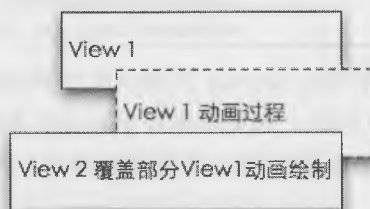


图 13-48 动画过程中的视图覆盖情况

### 13.11.3 ViewGroup 中 dispatchDraw() 中布局动画绘制流程

布局动画是在 ViewGroup 类中 dispatchDraw() 函数中完成的。dispatchDraw() 函数首先会获得该 ViewGroup 中包含的布局动画参数, 然后根据该参数逐个设置子视图的动画参数, 最后调用 drawChild() 函数将子视图绘制到屏幕上。

与视图动画相似, 应用程序也可以使用 res/anim/xxx.xml 文件描述一个布局动画, 所不同的是布局动画所使用的标签及属性与视图动画不同, 如以下代码所示:

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="50%"
    android:animation="@anim/slide_top_to_bottom" />
```

在以上代码中, layoutAnimation 标签是 Framework 内部定义的, 所有的 ViewGroup 实例都识别该标签。android:animation 属性指定了具体的动画文件, 该动画对应的是视图动画, 它将应用到每一个子视图中。android:delay 属性指定每一个视图动画的时间间隔, 比如当 animation 指定的动画持续时间为 3 秒时, 那么 delay 为 50% 的意思就是每当上一个动画开始 1.5 秒后, 下一个子视图就开始动画。

下面来分析以上 XML 文件是如何被读取的, 以及 ViewGroup 中的 dispatchDraw() 如何使用这些参数。

首先, 假设上面这段代码对应的 XML 文件名称为 anim\_tab.xml, 那么, 应用程序可以在 layout 文件中的具体 ViewGroup 实例的标签中使用 android:layoutAnimation 设置该 ViewGroup 实例的布局动画文件, 如以下代码所示:

```
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layoutAnimation="@anim/anim_tab" />
```

源码中, ViewGroup 类的构造函数将负责解析 layoutAnimation 属性, 如以下代码所示:

```
case R.styleable.ViewGroup_layoutAnimation:
    int id = a.getResourceId(attr, -1);
    if (id > 0) {
        setLayoutAnimation(AnimationUtils.loadLayoutAnimation(mContext, id));
    }
    break;
```

该段代码中, 当遇到 layoutAnimation 属性时, 首先调用 a.getResourceId() 获得对应的动画文件, 本例中动画文件就是 anim\_tab.xml。然后调用 AnimationUtils 类的静态功能函数 loadLayoutAnimation() 装载该 anim\_tab.xml 文件, 该函数将返回一个 LayoutAnimationController 对象, 接着再调用 setLayoutAnimation() 函数将该 Controller 对象赋值给 ViewGroup 中的全局变量 mLayoutAnimationController。

以上就是 ViewGroup 类构造函数中为布局动画所做的准备，其关键是要产生 setLayoutAnimation() 函数设置 Controller 对象，XML 文件的作用仅仅是用它产生这个 Controller 对象。

接下来，就要在 dispatchDraw() 函数中使用这个 Controller 对象了，具体从处理以下代码处开始：

```
1318         if ((flags & FLAG_RUN_ANIMATION) != 0 && canAnimate()) {
1319             final boolean cache = (mGroupFlags & FLAG_ANIMATION_CACHE)
```

① 判断 mGroupFlags 中是否包含 FLAG\_RUN\_ANIMATION 标识。注意，该标识并不是视图动画标识，而是 ViewGroup 中布局动画的标识，它是在 ViewGroup 类中定义的。如果存在该标识，说明需要启动布局动画，布局动画只需要启动一次。该标识是在 setLayoutAnimation() 函数中被置位，本次 dispatchDraw() 函数执行完毕后，会清除该标识。

② 判断 mGroupFlags 中是否包含 FLAG\_ANIMATION\_CACHE 标识，在默认情况下，mGroupFlags 中都会包含该标识。该标识的意义是说，在布局动画期间，为该 ViewGroup 中的每一个子视图创建一个显示缓冲区，这样的好处是提高绘图效率。

③ 使用 for 循环，为每一个子视图设置视图动画，注意，是视图动画。因为布局动画最终都是通过视图动画绘制到屏幕上的，这就是为什么 layoutAnimation 标签中都包含一个 android:animation 属性的原因，该属性值必须对应一个具体的视图动画。

(1) 调用 attachLayoutAnimationParams() 设置子视图的 LayoutParams 属性中的 LayoutAnimationParams 变量，该变量默认为空，其类型是 LayoutAnimationController.AnimationParameters 类。该类中有两个成员变量，分别是 count 和 index，count 代表该布局动画中一共有多少个兄弟视图，index 代表当前子视图在父视图中的序号。这些类关系如图 13-49 所示。

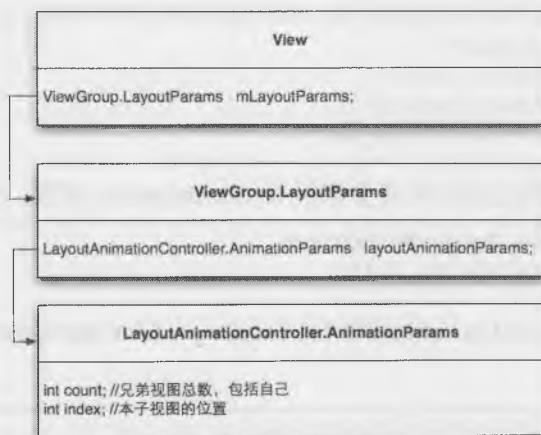


图 13-49 父视图控制子视图动画相关的参数关系

(2) 调用 bindLayoutAnimation() 设置具体的子视图动画。该函数中，调用布局动画控制器 mLayoutAnimationController 的 getAnimationForView(child) 获得指定子视图的动画，该函数的内部原理如下。

前面讲过, Controller 是根据布局动画的 XML 文件产生的, 该文件中包含两个重要属性, 一个是 delay, 另一个是 Animation。delay 表示子视图动画之间的延迟, Animation 标识应该应用给每个子视图的具体动画。

因为在第(1)步中已经设置了每一个子视图中 AnimationParams 对象的 count 和 index 值, 并且可以知道 animation 对应的动画持续时间, 所以理论上就可以计算出每个子视图动画的开始时间。Controller 对象的 getAnimationForView(child)的核心作用就是计算每个子视图动画的开始时间, 获得时间后, 再使用 Animation 的 clone()函数克隆一个 Animation 对象, 因为每个子视图都必须对应一个独立的 Animation 对象。最后再将计算出的起始时间赋值到新的 Animation 对象, 并把该 Animation 赋值给子视图即可。

(3) 如果需要使用绘制缓存, 则调用 child.buildDrawingCache(true)为子视图创建一个缓存。

④ 以上就完成了将布局动画转换为具体的子视图所需的视图动画, 由于以上过程可能会花费一点时间, 因此, 在设置好视图动画后, 默认是并不启动动画中的计时, 而此时完成了以上转换工作, 因此, 可以调用 controller.start()开始动画了。

⑤ 清除 mGroupFlags 中的 FLAG\_RUN\_ANIMATION 标识及 FLAG\_ANIMATION\_DONE 标识。前者代表是否需要启动布局动画, 由于以上几步已经完成了布局动画的设置, 因此, 接下来就不需要了, 所以要清除该标识。换句话说, 在一般情况下, 当一个 ViewGroup 被绘制后, 布局动画只出现一次, 除非应用程序再次调用 setLayoutAnimation() 或者再次调用 startLayoutAnimation()。FLAG\_ANIMATION\_DONE 代表布局动画是否完成, 由于动画才刚刚开始, 肯定还没有完成, 所以需要清除该标识。

⑥ 回调 mAnimationListener.onAnimationStart()函数。应用程序可以调用 setLayoutAnimationListener()设置一个监听器, 从而当布局动画开始时能够有机会执行其他操作。与此类似的还有当布局动画结束后, 会回调监听者的 onAnimationEnd()。

至此, 布局动画就结束了。

对于应用程序而言, 如果需要更为特别的布局动画, 则需编写自定义的 LayoutAnimationController 对象, 并且新的 Controller 必须继承于 LayoutAnimationController 类, 其内部逻辑就是要根据你想要的布局动画效果设计合适的子视图动画起始时间、起始位置等, 最后再调用 ViewGroup 类的 setLayoutAnimation()将新的 Controller 设置给该 ViewGroup 对象。

比如, 除了 LayoutAnimation 标签外, Framework 中还为 GridVeiw 定义了特别的 gridLayoutAnimation 标签, 在该标签中, 应用程序可以使用以下属性设置特别的效果。

- rowDelay: 设置每行的延迟。
- columnDelay: 设置每列的延迟。
- directionPriority: 其值可以是“row”或者“column”。
- direction: 设置动画的方向, 其值可以是 right\_to\_left, top\_to\_bottom 等。

关于 GridView 中这些属性的解析及意义参见 GridView 的源码, 其实现原理与前面所讲相同, 此处不再赘述。



## 第 14 章 WmS 工作原理

### 14.1 概述

WmS 是 Android 中图形用户接口的引擎，它管理着所有窗口。所谓的“管理”大致包括创建、删除窗口，以及将某个窗口设置为焦点窗口，焦点窗口是指当前正在和用户交互的窗口。本节将介绍一般窗口管理中的一些基本概念，以及窗口管理要解决的核心问题，理解这些基本概念及功能的目标有助于理解 Android 中 WmS 的实现方式。

#### 14.1.1 窗口的定义

在第 13 章中，我们曾对“窗口”、“Window”、“View”这三个概念做过区分。在本书的语义中，“窗口”是一种通用的描述，指一个独立的界面，比如一个对话框窗口、一个 Activity 交互的窗口、一个菜单窗口等；“Window”是一个类，其实现类是 PhoneWindow 类，Activity 类实现了 Window.Callback 接口，从而成了具有通用操作方式的窗口，所谓的“通用操作方式”包括，当用户按下“Menu”键后会弹出一个菜单，按“Back”键会退出当前 Activity 等；View 也是一个类，通常翻译为视图，指一个独立的交互元素，比如一个按钮、一个文本框等。

在 WmS 中，窗口是由两部分内容构成的，一部分是描述该窗口的类 WindowState，另一部分是该窗口在屏幕上对应的界面 Surface。换句话说，如果我们闭着眼睛操作手机的话，则窗口对应的 Surface 就完全是多余的，Surface 仅仅用于在屏幕上画一点界面，而负责将用户输入的触摸消息及按键消息派发到正在交互的窗口则与 Surface 一点关系都没有。



### 14.1.2 窗口管理要解决的核心问题

对于所有的窗口系统而言，窗口管理服务端要解决的核心问题如下。

#### 1. 窗口如何布局

窗口布局一般分为两种，一种是平铺式布局，另一种是层叠式布局，如图 14-1 所示。

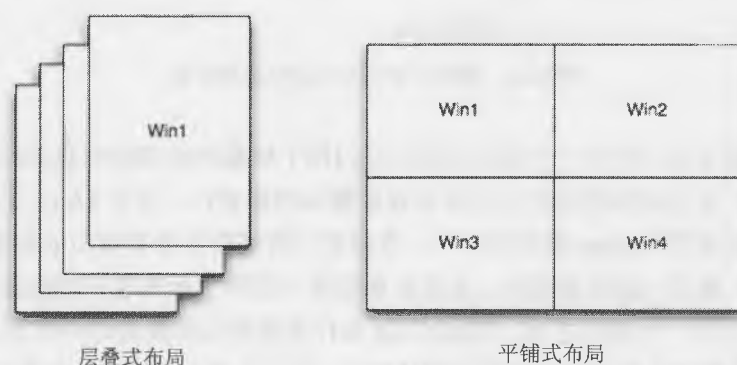


图 14-1 窗口布局的两种方式

早期的单窗口操作系统多使用平铺式布局，该布局的特点是所有的窗口都在一个平面中，设计上比较简单，但缺点是显而易见的，即当需要多个窗口时，界面很难排版。从程序设计的角度上来看，该布局一般不需要窗口管理服务端，只需要事先给每个窗口分配特定的窗口位置，然后由各个窗口内部的程序代码负责绘制该屏幕；而在消息交互时，则由输入模块直接根据触摸位置将触摸消息传递给相应的窗口程序即可。

层叠式布局的特点在于允许多个窗口层叠显示，Android 中采用的就是该方式。该布局一般都需要一个窗口管理服务端，从程序设计的角度来看，有两种设计模式可以实现服务端。一种是采用独立进程方式，另一种是采用共享库方式，如图 14-2、图 14-3 所示。

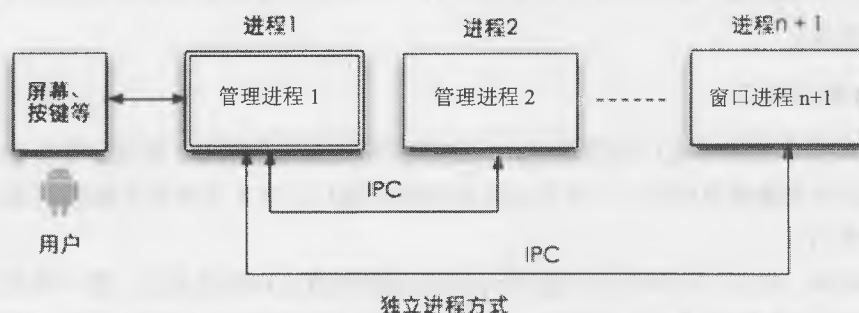


图 14-2 使用独立进程方式进行进程通信

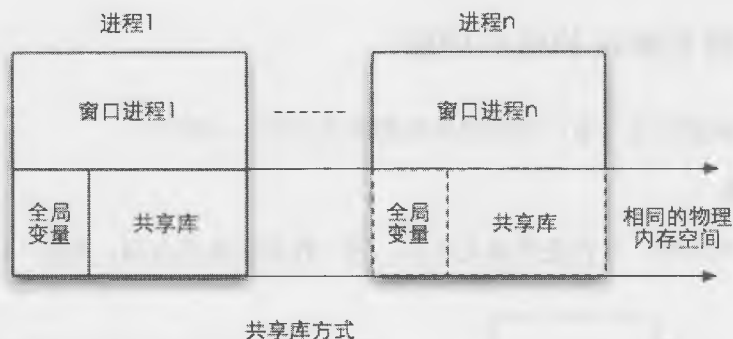


图 14-3 使用共享库方式进行进程通信

所谓独立进程方式是指，使用一个独立的进程专门用于屏幕的绘制和消息处理，所有的其他应用程序当需要创建窗口时，通过进程通信的方式请求管理服务创建窗口。比如 Linux 上的 X-window 就是一种独立进程的方式，它使用 Socket 通信的方式，通知窗口管理服务进行窗口的创建及交互消息传递。

共享库方式是指，使用一段共享程序，该段共享程序中保存了所有客户端的窗口信息，共享库和每个客户端程序都运行于同一个进程之间。Windowsg 操作系统使用的就是这种方式，很多嵌入式系统也使用这种方式。该方式的优点是窗口管理的开销比较小，尤其是窗口的交互，因为它不需要进程间通信，其缺点是任何一个客户端的不适当操作都可能导致窗口系统“崩溃”。

## 2. 窗口尺寸受制于哪些因素

在没有任何限制的情况下，窗口的尺寸应该完全由客户端自身指定，当然，指定的尺寸不应超过屏幕本身的尺寸，而针对具体的系统而言，窗口尺寸可能还要受到一些其他的所谓“系统窗口”的影响。比如，对于 Windows 操作系统而言，应用窗口的大小等于屏幕的大小减去状态栏的大小；对于苹果操作系统而言，应用窗口的大小等于屏幕的大小减去菜单栏的大小。

而在 Android 系统中，所谓的系统窗口包含两个部分，一个是状态栏，位于屏幕上方，另一个是输入法窗口。由于手机应用的特殊性，输入法窗口默认都会引起应用窗口尺寸发生变化，这会导致应用窗口需要根据新的窗口尺寸重新绘制界面。

为了支持这些受制因素，窗口管理系统需要定义一些特别的系统变量保存受制因素，并在计算窗口大小时使用这些变量。

## 3. 如何寻找焦点窗口

一种比较简单的寻找焦点窗口的逻辑是，一次只能有一个焦点窗口。在这种情况下，用户需要和其他窗口交互时，必须先激活目标窗口，使之成为新的焦点窗口，接下来的所有触摸消息或者按键消息都只能发送给焦点窗口。

而在实际系统中，却存在两种特别的需求分别针对按键消息和触摸消息。第一种是所谓的“系统按键”，即当被预定义的系统按键消息产生时，不是由当前的焦点窗口去处理，而是由系统进程处理，为了达到这个目的，窗口管理服务必须添加额外的逻辑来处理这种情况。第二种是跨窗口操作，典型的例

子如图 14-4 所示, 假设当前焦点窗口为进程 2, 此时当用户在黑点位置处滑动鼠标滚轮(或者任何其他导致相同语义的触摸消息)时, 窗口进程 1 中的内容是否应该被滚动, 即是否应该响应该滚动消息?

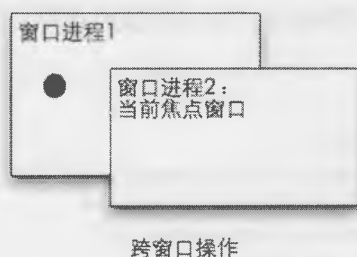


图 14-4 跨窗口的消息处理模式

在 Windows 操作系统中, 不会发生滚动, 而在苹果操作系统中, 则可以发生滚动。不滚动当然容易实现, 而如果要滚动, 则牵扯到如何寻找焦点窗口的问题, 即不仅要能根据当前焦点窗口派发消息, 还要能够根据窗口的层叠顺序将非按键消息派发给非焦点窗口。从这里也可以看出苹果窗口操作系统的高级性, 因为从用户的角度来看, 滚动似乎更合理一点。

在 Android 系统中, 尽管不存在这样的操作, 但这的确是窗口管理服务应该考虑的一个问题。

#### 4. 窗口切换时的动画策略

窗口切换时的动画是指, 当新窗口显示时或者老窗口消失时, 可以指定一个动画, 从而提供一种友好的界面效果。

在传统的 PC 操作系统中, 窗口系统是完全的多窗口, 即多个窗口可以层叠显示在屏幕上, 屏幕不被任何一个应用窗口所独占, 因此, 窗口的启动动画和关闭动画是彼此分离的, 新窗口的启动不会导致老焦点窗口的消失。而在智能手机系统中, 屏幕一般都会被应用窗口所独占, 无论是 Android 还是 iPhone 都是这种情况, 因此, 新窗口的启动动画同时会伴随着老窗口的关闭动画, 这就是一种特别的动画逻辑。同时, 在 Android 系统中, 每个应用窗口可以同时指定是否使用桌面背景作为窗口背景, 而在窗口动画的过程中却要保持桌面背景的静态性, 这就又增加了动画逻辑的复杂度。

除了普通窗口的启动、关闭动画外, Android 中增加了任务窗口动画逻辑, 即在不同任务之间切换窗口时所使用的动画, 这与相同任务之间窗口切换时的动画效果一般不同, 不过动画的机制本身是相同的。

### 14.1.3 解决核心问题所使用的相关的变量列表

当读者了解了以上核心问题后, 相信也能设置出一种纯粹的逻辑解决这些核心问题, 从而也就可以设计出一套窗口管理系统。然而程序是一门工程, 必须把这种纯粹的逻辑转换为代码, 分析源码的目标

就是为了得到这种逻辑。

程序上,任何逻辑都可以由过程处理和变量组成,窗口管理系统也不例外。为了详细了解这种逻辑,先来看看程序中都定义了哪些逻辑所需要的变量。

这些变量主要包含五类,如以下代码所示,此处仅仅列出这些变量的类型及作用,用于后面分析一些具体过程参考。

第一类:窗口管理相关

```
final HashSet<Session> mSessions;
final HashMap<IBinder, WindowState> mWindowMap;
final ArrayList<WindowState> mWindows; //Z-ordered
final HashMap<IBinder, WindowToken> mTokenMap =
    new HashMap<IBinder, WindowToken>();
final ArrayList<WindowToken> mTokenList;
final ArrayList<AppWindowToken> mAppTokens; //Z-ordered
/* Windows whose animations have ended and now must be removed.
final ArrayList<WindowState> mPendingRemove
// Windows whose surface should be destroyed.
final ArrayList<WindowState> mDestroySurface;
```

```
ArrayList<WindowState> mForceRemoves;
```

第二类:窗口动画相关

```
// Window tokens that are in the process of exiting, but still
// on screen for animations.
final ArrayList<WindowToken> mExitingTokens
final ArrayList<AppWindowToken> mExitingAppTokens;
final ArrayList<AppWindowToken> mFinishedStarting ;
/**
 * This was the app token that was used to retrieve the last enter
 * animation. It will be used for the next exit animation.
 */
AppWindowToken mLastEnterAnimToken;
/**
 * These were the layout params used to retrieve the last enter animation.
 * They will be used for the next exit animation.
 */
LayoutParams mLastEnterAnimParams;
/**
 * Windows whose animations have ended and now must be removed.
 */
final ArrayList<WindowState> mPendingRemove;
// State management of app transitions.
int mNextAppTransition = WindowManagerPolicy.TRANSIT_UNSET;
String mNextAppTransitionPackage;
int mNextAppTransitionEnter;
int mNextAppTransitionExit;
boolean mAppTransitionReady = false;
boolean mAppTransitionRunning = false;
boolean mAppTransitionTimeout = false;
boolean mStartingIconInTransition = false;
boolean mSkipAppTransitionAnimation = false;
```

```

final ArrayList<AppWindowToken> mOpeningApps;
final ArrayList<AppWindowToken> mClosingApps;
final ArrayList<AppWindowToken> mToTopApps;
final ArrayList<AppWindowToken> mToBottomApps;
float mWindowAnimationScale = 1.0f;
float mTransitionAnimationScale = 1.0f;

```

第三类, 输出法窗口管理相关

```

IInputMethodManager mInputMethodManager;
final ArrayList<WindowState> mResizingWindows;
// This just indicates the window the input method is on top of, not
// necessarily the window its input is going to.
WindowState mInputMethodTarget = null;
WindowState mUpcomingInputMethodTarget = null;
boolean mInputMethodTargetWaitingAnim;
int mInputMethodAnimLayerAdjustment;
WindowState mInputMethodWindow = null;
final ArrayList<WindowState> mInputMethodDialogs;

```

第四类, 墙纸窗口管理相关

```

final ArrayList<WindowToken> mWallpaperTokens;
// If non-null, this is the currently visible window that is associated
// with the wallpaper.
WindowState mWallpaperTarget = null;
// If non-null, we are in the middle of animating from one wallpaper target
// to another, and this is the lower one in Z-order.
WindowState mLowerWallpaperTarget = null;
// If non-null, we are in the middle of animating from one wallpaper target
// to another, and this is the higher one in Z-order.
WindowState mUpperWallpaperTarget = null;
int mWallpaperAnimLayerAdjustment;
float mLastWallpaperX = -1;
float mLastWallpaperY = -1;
float mLastWallpaperXStep = -1;
float mLastWallpaperYStep = -1;
// This is set when we are waiting for a wallpaper to tell us it is done
// changing its scroll position.
WindowState mWaitingOnWallpaper;
// The last time we had a timeout when waiting for a wallpaper.
long mLastWallpaperTimeoutTime;
// We give a wallpaper up to 150ms to finish scrolling.
static final long WALLPAPER_TIMEOUT = 150;
// Time we wait after a timeout before trying to wait again.
static final long WALLPAPER_TIMEOUT_RECOVERY = 10000;

```

第五类, 焦点窗口管理相关

```

/* Windows that have lost input focus and are waiting for the new
 * focus window to be displayed before they are told about this.
 */
ArrayList<WindowState> mLosingFocus = new ArrayList<WindowState>();
WindowState mCurrentFocus = null;
WindowState mLastFocus = null;
AppWindowToken mFocusedApp = null;

```

#### 14.1.4 几个操作的概念

在分析 WmS 内部逻辑中, 会进行三种常见的操作, 具体的操作可能会对应不同的函数名称, 但是, 这些操作的语义是相同的。三种常见的操作为 assign layer、perform layout, 以及 place surface。

- assign layer 的语义是, 为窗口分配层值。在 WmS 中, 每个窗口都使用 WindowState 类来描述, 而窗口要在界面上显示时, 需要指定窗口的层值。从用户的视角来看, 层值越大, 其窗口越靠近用户, 窗口之间的层叠正是按照层值进行的。
- perform layout 的语义是, 计算窗口的大小。每个窗口对象都必须有一个大小, 即窗口大小, perform layout 将根据状态栏大小、输入法窗口的状态、窗口动画状态计算该窗口的大小。
- place surface 的语义是, 调整 Surface 对象的属性, 并重新将其显示到屏幕上。由于 assign layer 和 perform layout 的执行结果影响的仅仅是 WindowState 中的参数, 而能够显示到屏幕上的窗口都包含一个 Surface 对象, 因此只有将以上执行结果中的窗口层值、大小设置到 Surface 对象中, 屏幕上才能看出该窗口的变化。place surface 的过程就是将这些值赋值给 Surface 对象, 并告诉 Surface Flinger 服务重新显示这些 Surface 对象。

#### 14.1.5 什么是 Policy, 以及其与 WmS 的关系

在 WmS 类内部有一个 WindowManagerPolicy mPolicy 变量, 这是窗口管理的策略机制, 那么什么是策略? 它与 WmS 的关系是什么?

首先, 用一个比喻来理解这里所讲的“策略”的意思。假设某厂商制造了一个塑料器皿, 比如一个杯子, 但在制造时, 厂商并没有宣称这是一个杯子, 而是一个“比较通用”的容器, 至于用户用这个容器装水、饮料、醋等则取决于用户。但是, 由于这种塑料材质的特殊性, 厂商给出建议: 如果用该器皿装水, 则有效期为 2 个月, 如果装饮料, 有效期为 1 个月, 如果装醋, 有效期为 10 天。

这个“建议”就是策略。理论上讲, WmS 可以任意添加或者删除窗口, 它是一种“比较通用”的窗口管理系统, 然而当 Android 被用于手机产品时, WmS 在添加及删除窗口时需要做一些额外的调整。比如系统状态栏窗口不允许被添加两个, 这就是策略, 该策略的实现类是 PhoneWindowManager, 正如其名称所示, 它是当 WmS 系统用于 Phone (电话) 时应该遵守的“建议”, 该名称或许更应该叫做 PhoneWindowManagerPolicy。

因此, 策略是为了限制 WmS 的功能, 或者使 WmS 遵守某种规则, 而不是扩充 WmS 的功能。WindowManagerPolicy 是一个 interface, 其内部定义的 API 接口会在 WmS 中不同场景下被使用。如果某天我们想基于 Android 设计一种新 Mid 产品, 该产品的窗口特性和电话的不同, 比如该产品的输入法窗口不会引起应用窗口大小的变化、系统窗口包含一个状态栏和一个系统菜单栏, 等等, 那么可以基于 WindowManagerPolicy 重新实现一种新的策略, 名称可以叫做 MidWindowManager。

## 14.1.6 WmS 接口结构

WmS 接口结构是指 WmS 功能模块与其他功能模块之间的交互接口，其中主要包括与 AmS 模块及应用程序客户端的接口，其关系如图 14-5 所示。

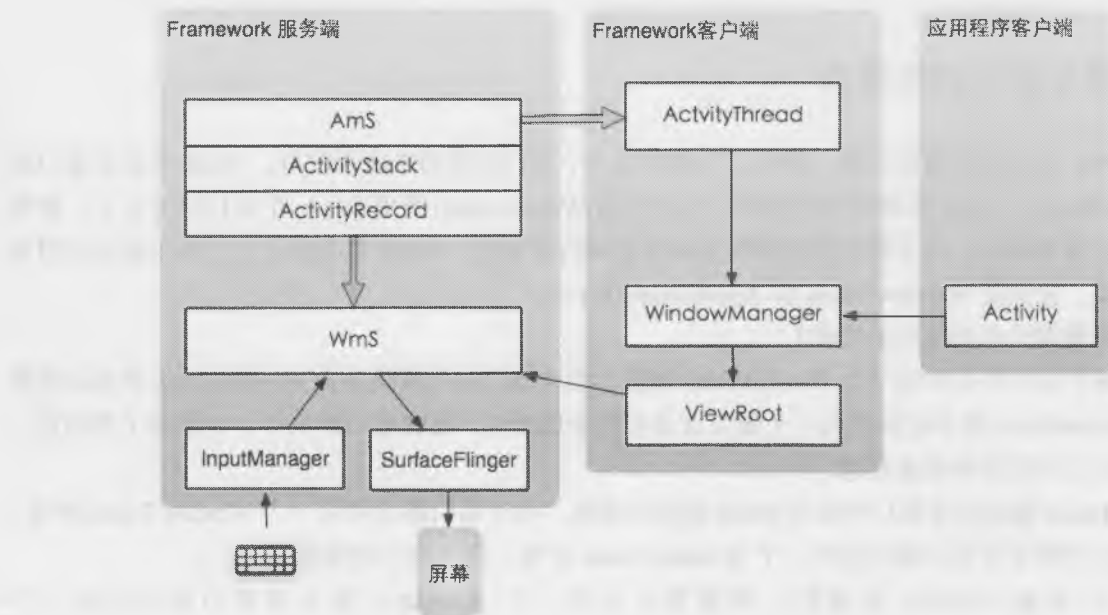


图 14-5 WmS 在系统中的接口结构

该结构中的主要交互过程如下。

① 应用程序在 Activity 中添加、删除窗口。具体实现就是通过调用 WindowManager 类的 addView() 和 removeView() 函数完成，这会转而调用 ViewRoot 类的相关方法，然后通过 IPC 调用到 WmS 中的相关方法完成添加、删除过程。

② 当 AmS 通知 ActivityThread 销毁某个 Activity 时，ActivityThread 会直接调用 WindowManager 中的 removeView() 方法删除窗口。

③ AmS 中直接调用 WmS，这种调用一般都不是请求 WmS 创建或者删除窗口，而是告诉 WmS 一些其他信息。比如某个新的 Activity 就要启动了，从而 WmS 会保存一个该 Activity 记录的引用，关于具体的调用后面小节会在具体的操作中介绍。

而在 WmS 内部，则全权接管了输入消息的处理和屏幕的绘制。其中输入消息的处理是借助于 InputManager 类完成的，详情参见第 11 章；而绘制屏幕则是借助于 SurfaceFlinger 模块完成的，SurfaceFlinger 是 Linux 的一个驱动，它内部会使用芯片的图形加速引擎完成对界面的绘制。



## 14.2 WmS 主要内部类

在 WmS 内部定了很多内联类，了解这些内联类的作用有助于理解 WmS 源码的工作逻辑。

### 14.2.1 表示窗口的数据类

由于 WmS 是用来管理窗口的，因此，需要定义一个专门的类用来表示窗口，WmS 中表示窗口的类是 WindowState。从设计原理的角度来讲，似乎使用 WindowState 类来表示一个窗口就可以了，然而从程序实现的角度来讲，出于编程的便利性及程序逻辑的清晰性，WmS 类内部还定了两个额外的用来表示窗口的类，分别是 WindowToken 和 AppWindowToken。

为什么还需要这两个额外的类呢？

首先，每个窗口都会对应一个 WindowState 对象。因为窗口的本质就是由 WindowState 类描述的数据对象，WindowState 类中记录作为一个窗口应该有的全部属性，比如窗口的大小、在屏幕上的层值，以及窗口动画过程的各种状态信息。

WindowToken 描述的是窗口对应的 token 的相关属性，每个窗口都会对应一个 WindowToken 对象，但是是一个窗口的所有子窗口将对应同一个 WindowToken 对象，即多对一的关系。

如果窗口是由 Activity 创建的，即该窗口对应一个 Activity，那么该窗口同时对应一个 AppWindowToken 对象。

从数量上看，WindowState、WindowToken、AppWindowToken 三者的对比如下，其中 AppWindowToken 数量最少，其次是 WindowToken，最后是 WindowState。

```
=====AppWindowToken
=====WindowToken
=====WindowState
```

### 14.2.2 DimAnimator

Dim 的字面意思是“变暗”，DimAnimator 类的作用是保存“变暗动画”过程所需的各种状态变量。DimAnimator 不是 Animation 类的子类。

在 WmS 内部的处理逻辑中，和“变暗”相关并且容易混淆的有以下几个概念，如表 14-1 所示，请注意区分。

表 14-1 和“变暗”相关的概念区别

概念名称	描 述
blur	模糊的意思，就好像是照相机拍照时焦点没对准而引起的模糊一样
dim	变暗的意思。dim 和 blur 的区别在于，此时窗口有可能很暗淡，但是却还是挺清晰。dim 和 blur 在 WM.LP 类中都对应一个 FLAG，应用程序可以使用该 FLAG 指定窗口的效果，blur 和 dim 的效果如图 14-6 所示
obscured	本意有点类似 dim，但更侧重于抽象上的暗淡，比如晦涩，纯粹是一个心理上对暗淡的抽象。在程序中，其意思是不透明，和 opaque 相似，但是 opaque 用于表明一个窗口是否透明，可以通过调用 WindowSate 的 isOpaque()函数查询是否是 opaque 的；而 obscured 则表示多个窗口在一起后还能看得见该窗口，如果看不见，则 WindowState 的内部变量 mObscured 为 true，mObscured 的值会根据多个窗口在一起后的透明效果而动态改变
opaque	表明该窗口是否透明，即绘制窗口的颜色值是否包含阿尔法通道。如果包含，则 opaque 为 false，否则为 true
update effect	更新效果，在程序中所谓的“effect”是指 dim 或者 blur 的效果

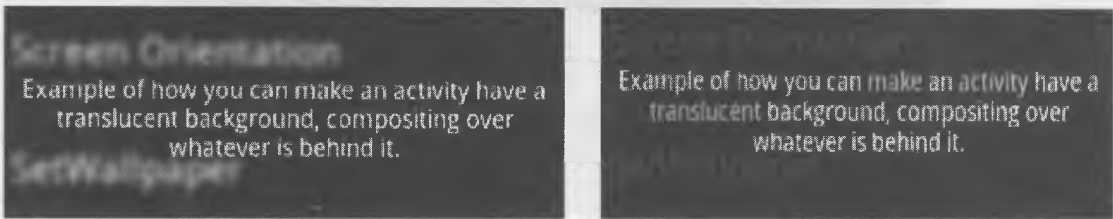


图 14-6 背景窗口为 blur 和 dim 的效果对比

14.2.3 FadeInOutAnimation

这是一个 Animation 的子类，用于实现渐入、渐出的动画效果。其核心在于使用了一个名称为 AccelerateInterpolator 的插入器，该插入器的方程如以下代码所示：

```
60 public float getInterpolation(float input) {
61     if (mFactor == 1.0f) {
62         return input * input;
```

其特点是进入的时候刚开始比较缓慢，然后会一下子变快。当一个窗口要显示时，用户会感觉到它是慢慢地开始显示，然后一下子就全部显示出来了；当一个窗口要退出时，用户会感觉到它是慢慢变模糊，然后一下子消失。

该插入器是在 FadeInOutAnimation 的构造函数中指定的。  
当然，插入器只是将自然时间转换为一种变换后的时间，而具体的渐入、渐出效果必须由阿尔法通道的变化来实现。因此在 applyTransformation()函数中，将会根据当前时间获取插入器返回的阿尔法值，并将该值设置到参数 Transformation 变量中。该动画中阿尔法和转换后的时间方程是  $y = (x - 0.5) \times 2$ ,

如果再使用 `AccelerateInterpolator` 中的转换关系, 那么最终的阿尔法方程为  $y = (t * t - 0.5) \times 2$ , 其方程图如图 14-7 所示。

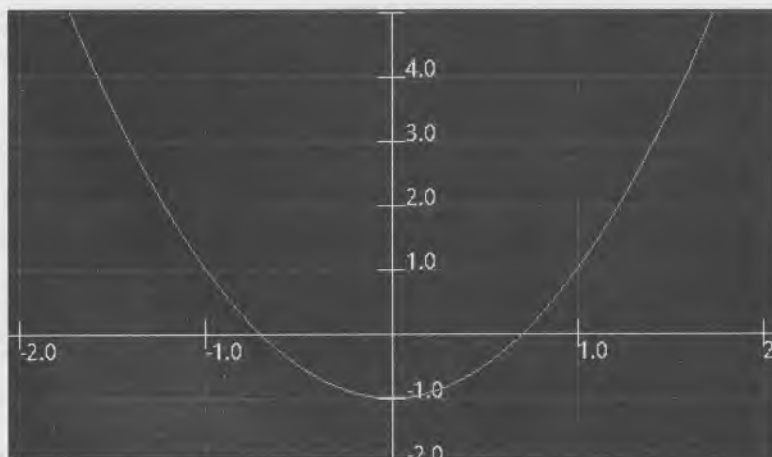


图 14-7 加速曲线方程

并且当时间值约为总时间的 0.689 时,  $y$  值小于 0。在这种情况下, 窗口的大小被设置为 0, 从而使得用户不能看见该窗口, 如以下代码所示。

```
10634         if (x < 0.5) {  
10635             // move the window out of the screen.  
10636             t.getMatrix().setTranslate(mWidth, 0);  
10637         } else {  
10638             t.getMatrix().setTranslate(0, 0); // show  
10639             t.setAlpha((x - 0.5f) * 2);  
10640         }
```

#### 14.2.4 InputMonitor 类

在第 11 章中介绍过 `InputMonitor` 与 `InputManager` 的作用, 本节从具体的调用关系上再介绍一下该类。

`InputMonitor` 在 `WmS` 中的作用如图 14-8 所示。

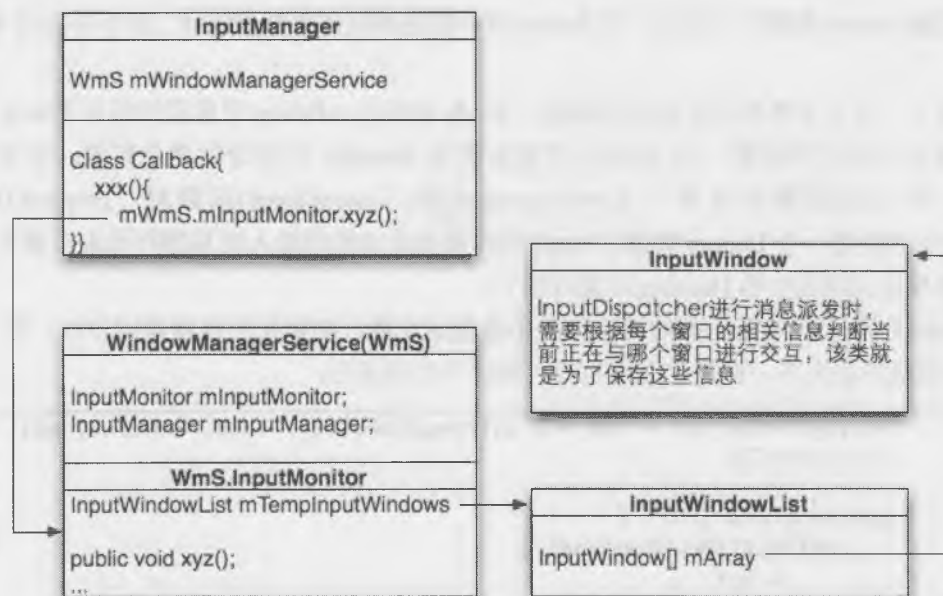


图 14-8 InputMonitor 在 Wms 中的作用

首先，在 WmS 类中有两个全局变量，一个是 InputMonitor 对象，另一个是 InputManager 对象。而在 InputManager 类中，有一个全局变量 mWindowManagerService，它正是 WmS 类。当底层的 InputDispatcher 线程接收到消息后，首先会回调 InputManager 中的一个 Callback 对象的相应函数，而这些函数内部大多数都是调用 WmS 类中的 mInputMonitor 对象的相应函数，这些函数往往都需要对 mTempInputWindows 变量中的窗口信息进行一定的调整。该变量的类型是 InputWindowList 类，该类内部有一个 InputWindow 数组，保存的正是每一个窗口的信息。InputWindow 类与 WindowState 的区别是，前者仅保存一些为了寻找焦点窗口所需的各种信息，而 WindowState 则保存了一个窗口中的所有信息，以及窗口应有的各种函数。

### 14.2.5 PolicyThread

PolicyThread 是一个线程类，其父类是 Thread，该类中包含的核心变量就是 mPolicy，其类型是 WindowManagerPolicy。该变量的名称与 WmS 类中的 mPolicy 名称相同并且类型相同，事实上，PolicyThread 中的 mPolicy 是在构造函数中被赋值为 WmS 中的 mPolicy，而 WmS 中的 mPolicy 是在 WmS 初始化时创建的，如以下代码所示。

```
267 final WindowManagerPolicy mPolicy = PolicyManager.makeNewWindowManager();
```

PolicyThread 可被认为是 PhoneWindowManager 类的一个包装，从一般的设计角度来看，如果 WmS 中需要使用 Policy 类，那么只需要构造一个 Policy 的对象，然后直接调用其方法即可，为什么还要用

一个线程来包装 Policy 类呢？原因是，在 Policy 类中有些函数需要异步执行，而不是占用 WmS 所在的线程。

一般情况下，对于不需要异步执行的函数，WmS 会通过 mPolicy 变量直接调用 Policy 的函数，而对于那些需要异步执行的函数，在 Policy 内部会使用 Handler 机制进行消息传递。这就是为什么在 PolicyThread 的 run() 函数中使用了 Looper.prepare() 和 Looper.loop() 函数对，prepare() 的作用是为 PolicyThread 线程创建一个 Looper 对象，loop() 的作用是让该线程进入消息循环状态，接下来的所有异步消息处理就都在该线程中的 Handler 对象内部了。

PolicyThread 类是在 WmS 类的构造函数中被初始化的，初始化的过程是同步的，即 WmS 等待 PolicyThread 线程启动完毕，然后才继续执行，如以下代码所示：

```

632     PolicyThread thr = new PolicyThread(mPolicy, this, context, pm);
633     thr.start();
634
635     synchronized (thr) {
636         while (!thr.mRunning) {
637             try {
638                 thr.wait();
639             } catch (InterruptedException e) {
640             }
641         }
642     }

```

PolicyThread 类中对应的通知代码如下：

```

589         synchronized (this) {
590             mRunning = true;
591             notifyAll();
592         }
593
594         Looper.loop();

```

即在调用 loop() 函数之前发出通知消息，从而使得 WmS 得以继续运行。

## 14.2.6 Session

和 SurfaceFlinger 直接打交道的类本来是 SurfaceSession。当应用程序需要创建 Surface 时，会请求 WmS 去完成创建的工作，WmS 会为每一个应用程序分配一个 SurfaceSession 对象。然而一个 SurfaceSession 对象不足以表示一个客户端，因此，WmS 定义了 Session 类，它可被认为是 SurfaceSession 的一个包装。Session 类中最重要的成员变量是 SurfaceSession mSurfaceSession，除了该变量外，还有应用程序对应的 pid 和 uid 等。

Session 对象是当应用程序调用 WmS 的 openSession() 函数时创建的，而应用程序又是在 ViewRoot 类中调用 openSession 的，ViewRoot 中有一个静态变量，如以下代码所示：

```
100 static IWindowSession sWindowSession;
```

sWindowSession 只会被创建一次，这就保证了一个应用程序中只有一个 IWindowSession 实例，而在 WmS 中也就只对应一个 Session 对象。

Session 对象创建好后，接下来当应用程序需要执行和窗口管理相关的操作时，都是通过 IPC 调用该 Session 对象中的相关函数实现的，因为 Session 类是 IWindowSession.Stub 的 Binder 实现，如以下代码所示：

```
55725 private final class Session extends IWindowSession.Stub
```

### 14.2.7 Watermark

从一般的角度来讲，水印（Watermark）有两个作用：第一个作用是为了防止图像被篡改，而给图像中添加一些视觉上不易被发现的像素，这些像素只有被程序处理时才能提取出来，从而在一定程度上保护图像的真实性；第二个作用是仅仅添加一些通用背景，比如大家在看一些 pdf 文档时，文档的背景上经常会有类似“Confidential”的字样，这种水印仅仅是一种背景，只不过这种背景文字或图案会存在于整幅页面中，并循环出现，从而在一定程度上防止用户对页面中信息的任意复制，或者也是给用户某种提醒。

WmS 中的 Watermark 类目前来看其作用是后者，而且该水印是一种文字水印。首先来看该水印的效果图，如图 14-9 所示。



图 14-9 添加水印效果后的界面

实现以上水印的过程如下。

首先，在 `/system/etc/` 目录下新建一个名称为 `setup.conf` 的文本文件，然后给该文件的第一行中添加以下文本内容：

```
iloveyou%
```

该文本的特点是：第一，这段文本必须在 `setup.conf` 的第一行；第二，这段文本必须以百分号结束。

`Watermakr` 类中会从 `setup.conf` 文件中读取第一行的内容，并截取百分号前面的这段文本，然后经过以下代码所示的逻辑，将 16 进制字符转换为非打印字符，目前尚不清楚为什么要这样转换。并且在默认情况下，`/system/etc/setup.conf` 文件并不存在，所以，水印文字默认为空，屏幕上不会看到以上所示的水印效果。

```
10082         int len = mTokens[0].length();
10083         len = len & ~1;
10084         for (int i=0; i<len; i+=2) {
10085             int c1 = mTokens[0].charAt(i);
10086             int c2 = mTokens[0].charAt(i+1);
10087             if (c1 >= 'a' && c1 <= 'f') c1 = c1 - 'a' + 10;
10088             else if (c1 >= 'A' && c1 <= 'F') c1 = c1 - 'A' + 10;
10089             else c1 = '0';
10090             if (c2 >= 'a' && c2 <= 'f') c2 = c2 - 'a' + 10;
10091             else if (c2 >= 'A' && c2 <= 'F') c2 = c2 - 'A' + 10;
10092             else c2 = '0';
10093             builder.append((char)(255-((c1*16)+c2)));
10094         }
```

`WmS` 初始化时会创建一个用于显示这些字符的 `Surface`，并且该 `Surface` 的层值很大，如以下代码所示。

代码中参数 100 代表第 100 号类型，由于实际上所使用的类型约有 10 种左右，因此，100 是一个很大的值，这里的作用仅仅是让水印的层值在最上面而已。

最后，在 `performLayoutAndPlaceSurfaceLockedInner()` 函数中调用了以下代码，从而使得每次总是会显示水印图层，以达到整个屏幕的水印效果。

```
8509         if (mWatermark != null) {
8510             mWatermark.positionSurface(dw, dh);
8511         }
```

## 14.2.8 WMThread

正如在 `WmS` 中创建 `Policy` 对象一样，`WmS` 对象是在 `SystemService` 类中创建的。当 `SystemService` 创建 `WmS` 对象时，同样希望 `WmS` 对象是在一个独立的线程中运行的，因为 `WmS` 内部会有一些需要异步执行的函数，这就是 `WMThread` 类存在的理由。



WMThread 也是一个线程类，就好比 PolicyThread 一样。并且 WMThread 线程的启动方式和 PolicyThread 几乎完全一样。

首先，在SystemService 类中调用 WmS 的 main()函数创建一个 WMThread 线程对象，然后启动该线程，如以下代码所示：

```

513 public static WindowManagerService main(Context context,
514     PowerManagerService pm, boolean haveInputMethods) {
515     WMThread thr = new WMThread(context, pm, haveInputMethods);
516     thr.start();
517
518     synchronized (thr) {
519         while (thr.mService == null) {
520             try {
521                 thr.wait();
522             } catch (InterruptedException e) {
523             }
524         }
525         return thr.mService;
    
```

而在 WMThread 线程的 run 函数中，会调用 Looper.prepare()和 Looper.loop()函数对，为该线程创建一个 Looper 对象，然后再通知 SystemServer 所在的线程，从而使得 SystemServer 线程得以继续运行。

```

552         synchronized (this) {
553             mService = s;
554             notifyAll();
    
```

## 14.3 窗口的创建和删除

从本节之后就开始具体介绍一些核心的窗口管理过程，首先来看窗口的创建和删除。

创建、删除操作逻辑的关键问题是：第一，什么情况下需要添加、删除操作，这实际上指窗口的运作机制；第二，如何创建、删除窗口，即程序上是如何实现这些逻辑的。

### 14.3.1 创建窗口的时机和过程

创建窗口的时机可分为两种，第一种是程序员主动调用 WindowManager 类的 addView()方法；另一种是当用户启动一个新的 Activity 或者显示一个对话框、菜单栏等的时候，在这种情况下，程序员并不直接调用 addView()函数，但是这些类的内部同样会间接调用 addView()函数。关于第二种情况的细节请参照第 8 章，本节主要介绍调用 addView()以后 WmS 内部的主要执行过程。

当客户端调用 WindowManager 类的 addView()方法后，该方法会创建一个新的 ViewRoot 对象，然后调用 ViewRoot 类的 setView()方法，该方法中会通过 IPC 方式调用 WmS 类中内联类 Session 的 add()方法，该过程如图 14-10 所示。

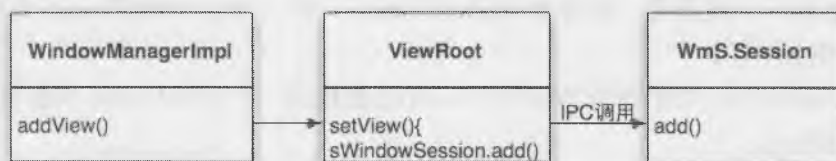


图 14-10 创建窗口的主要调用过程

Session 类的 add() 方法又会间接调用 WmS 的 addWindow() 方法，下面将详细分析 addWindow() 的内部执行流程。该流程内部可粗略分为三个小过程，第一个过程是进行前置处理，即首先判断参数的合法性，以确保接下来的添加操作能够顺利进行；第二个过程是具体添加和窗口相关的数据；第三个过程是后置处理，即添加窗口会引起相关状态的变化，因此需要把这些变化反映到相关的数据中。

第一个过程如图 14-11 所示。

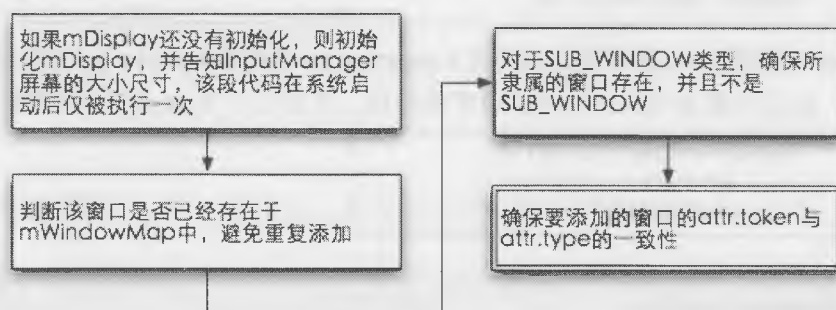


图 14-11 创建窗口之前置处理

① 判断 mDisplay 是否为空。该变量是全局的，代表了当前屏幕的 Display 属性，得到该属性后，会调用 mInputManager.setDisplaySize() 将该属性告知给 InputManager 对象。因此 InputManager 需要知道屏幕的大小尺寸，以便根据该尺寸判断输入的消息对应哪个窗口。该段代码当系统启动后仅被执行一次。

② 判断目标窗口是否已经存在 mWindowMap 列表中。此时，目标窗口是 IWindow 类型，其对应 ViewRoot 类中的 W 对象。

③ 如果目标窗口参数 attr.type 是 SUB\_WINDOW，即子窗口类型，那么目标窗口所隶属的父窗口必须是存在的，并且不能也是 SUB\_WINDOW 类型，否则添加无效。

④ 确保三个特殊窗口的 attr.token 和 attr.type 的一致性，这三个特殊窗口分别是应用窗口、输入法窗口及墙纸窗口。此处的一致性具体是指：如果是应用类窗口，那么 attr.token 不能为空，并且 attr.token.appWindowToken 不为空，appWindowToken 是由 AmS 请求 WmS 创建的；如果是输入法窗口，则 attr.token 必须存在，并且其类型必须是 INPUT\_METHOD，对于应用程序而言，没有权限创建输入法窗口，该窗口是在 IMMS 中创建的，创建之前，它会先创建一个 WindowToken 对象；如果是墙纸窗口，则 token 必须存在，并且其类型必须是 TYPE\_WALLPAPER，应用程序没有权限创建墙纸窗口，该

窗口是在 WallpaperManagerService 服务中创建，创建之前，它会先创建一个 WindowToken 对象。

前置检查完成后，如果没什么错误，就开始真正的窗口添加，其过程如图 14-12 所示。

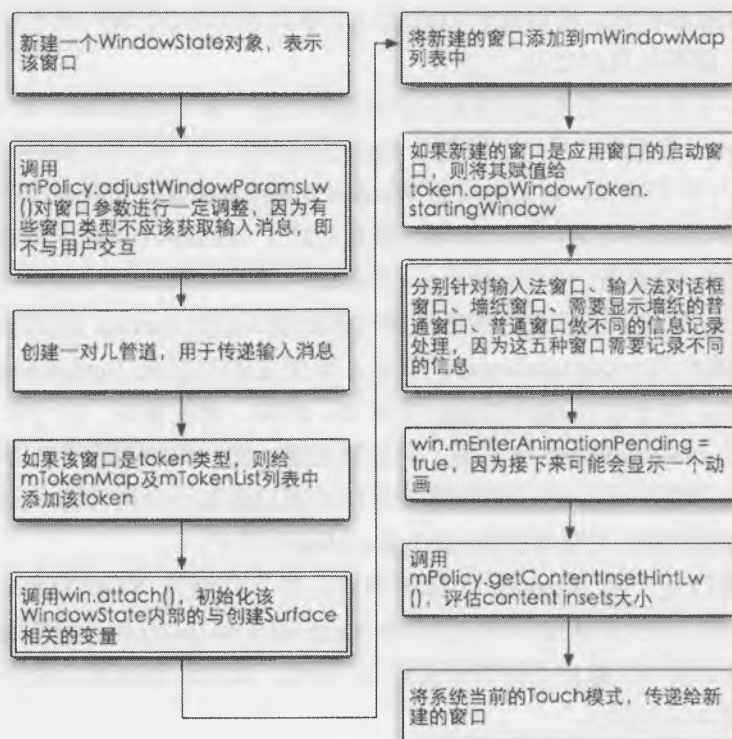


图 14-12 创建窗口之实际添加

① 新建一个 WindowState 对象，窗口都是由 WindowState 类表示，每个窗口对应一个 WindowState 对象。创建该对象的代码如下：

```

1878         win = new WindowState(session, client, token,
1879                               attachedWindow, attrs, viewVisibility);
  
```

构造函数参数的意义如下：

- session: 每一个应用程序客户端都对应一个 Session 类对象。
- client: 指 ViewRoot 中的内联类 W 对象。对 WmS 而言，所谓的窗口客户端 (client) 就是指 ViewRoot.W 类，因为 WmS 与客户端的通信正是要靠该对象，W 类本身是一个 Binder。
- token: 每个窗口都必须有一个 attr.token 对象，在调用 addWindow() 方法时，attr.token 也有可能为空，不过，在上面的步骤中，如果为空，则会新建一个 WindowToken 对象。
- attachedWindow: 指该窗口所隶属的窗口，只有当该窗口为 SUB\_WINDOW 时，该变量才不为空。

② 调用 `mPolicy.adjustWindowParamsLw()` 对窗口参数进行调整。前面讲过，“策略”的作用是为了做某种限制，该函数内部具体的“限制”很简单：当新建窗口的类型是 `SYSTEM_OVERLAY`、`SECURE_SYSTEM_OVERLAY`、`TOAST` 三种类型时，给 `attr.flags` 中增加 `FLAG_NOT_FOCUSABLE` 和 `FLAG_NOT_TOUCHABLE` 标识。因为对于 `OVERLAY` 窗口和 `TOAST` 窗口是不应该获得输入焦点的，也不能响应触摸消息，即不与用户交互。

③ 创建一对 pipe，关于 pipe 的使用详情参见获取输入消息一章。此处创建一对 pipe，然后将其中一个赋值给输出参数 `outchannel` 变量中，另一个赋值给 `InputDispatcher` 中，这对管道将用于输入消息的传递。

④ 如果该窗口是一个子窗口，则其对应的 `WindowToken` 对象肯定已经存在，否则，前面肯定已经创建了一个新的 `WindowToken` 对象，因此需要把新建的这个 `WindowToken` 对象添加到 `mTokenMap` 列表和 `mTokenList` 列表中。前者是一个 `HashMap`，后者是一个 `ArrayList`。

⑤ 调用 `win.attach()` 方法，该方法内部将初始化和创建真正的 `Surface` 相关的变量，主要就是初始化该 `WindowState` 对象内的 `mSurfaceSession` 变量。该变量的类型是 `SurfaceSession`，该类是直接和 `SurfaceFlinger` 交互的类，用于向 `SurfaceFlinger` 中添加、删除、变换窗口。由于每个应用程序仅对应一个 `Session` 对象，因此，`mSurfaceSession` 实际上只会被创建一次，即应用程序中的第一个窗口创建时会构造一个 `SurfaceSession` 对象，同一个应用程序中后续的窗口添加不会再构造 `SurfaceSession` 对象。

⑥ 将新建的 `WindowState` 对象添加到 `mWindowMap` 列表中。

⑦ 如果新建的窗口是一个 `Activity` 对应的窗口，并且类型为 `STARTING`，则意味着该窗口是 `Activity` 的一个启动窗口，因此将该窗口赋值给 `token.appWindowToken.startingWindow`。

⑧ 由于不同类型的窗口信息由不同的变量保存，因此，此处会根据窗口类型调用不同的函数完成信息的保存，具体如图 14-13 所示。

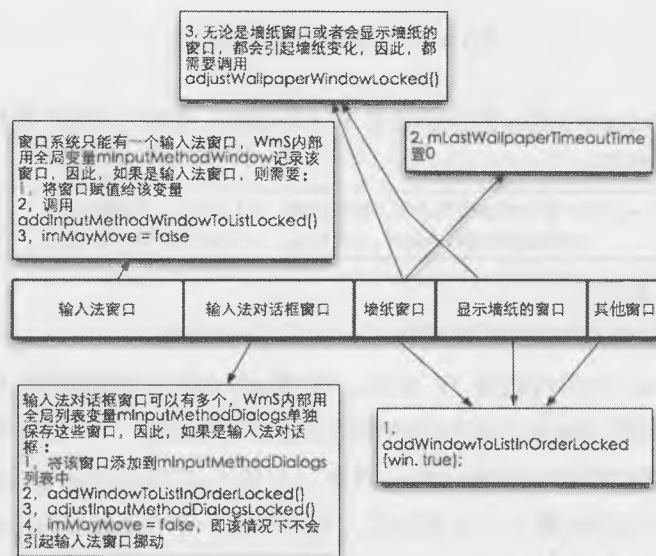


图 14-13 不同类型窗口的处理方式

⑨ 将 `win.mEnterAnimationPending` 变量置为 `true`，因为添加新的窗口后会引发窗口进入动画，即有新的窗口要显示了。

⑩ 调用 `mPolicy.getContentInsetHintLw()` 评估内容边框填充大小，并且将评估结果输出到参数 `outContentInsets` 中，该参数将返回到客户端的 `ViewRoot` 中。该函数执行的本质是将系统窗口的大小作为目标窗口内容边框的填充大小，从而避免目标窗口和系统窗口重叠，其效果如图 14-14 所示。

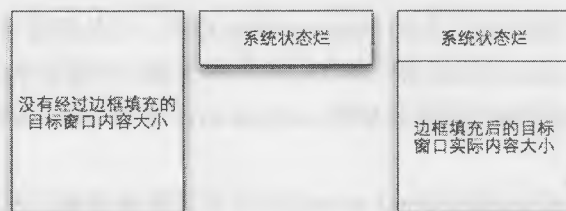


图 14-14 窗口元素布局

⑪ 将系统当前的 `touch` 模式传递给新窗口，典型的情况就是如果上一个 `Activity` 正在和按键进行交互，那么转到下一个 `Activity` 时应该还继续有按键的焦点窗口，这可以保持 `touch` 模式的连续性。`WmS` 中使用 `mInTouchMode` 表示当前处于触摸模式或按键模式。

完成了以上和窗口相关变量的赋值后，由于新建窗口后会引发其他的一些效果，比如输入法窗口可能需要重新寻找目标视图，因此，接下来需要进行这些所谓的“后置处理”，其过程如图 14-15 所示。

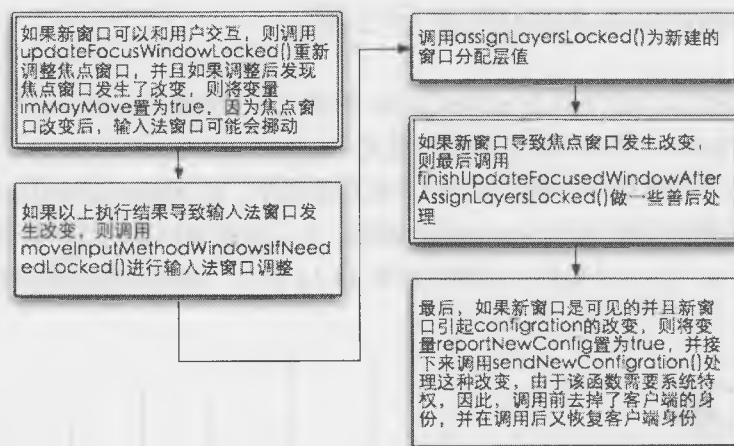


图 14-15 创建窗口之后置处理

① 判断新加的窗口是否是一个可交互的窗口，像 `Toast` 这样的窗口是一个不可交互的窗口。如果是可交互的，则需要调用 `updateFocusWindowLocked()` 重新计算新的焦点窗口，在一般情况下新加的窗

口将成为新的焦点窗口。但也有例外，比如当前正在和一个层值很大的窗口进行交互，而新加的窗口层值却可能是比较小的，则不会发生焦点窗口切换。如果发生了焦点窗口切换，则将变量 `imMayMove` 置为 `true`，此处的 `im` 是指 `Input Method`，即输入法窗口，因为焦点窗口改变后，输入法窗口也会跟随着重新寻找下一个焦点视图。

② 如果 `imMayMove` 为 `true`，则调用 `moveInputMethodWindowsIfNeededLocked()` 调整输入法窗口的焦点视图。

③ 截止到现在，尽管已经添加了新的 `WindowState` 对象，但是新窗口的具体层值还是不确定的，因此需要调用 `assignLayersLocked()` 根据窗口的类型及次序为窗口分配新的层值，层值将决定窗口在界面上的层叠顺序。该函数内部的执行本质是根据 `mBaseLayer` 计算 `mLayer` 的值。关于该函数的具体执行过程见下一小节。

④ 如果第一步中 `updateFocusWindowLocked()` 执行后发现焦点窗口发生了变化，并且上一步也为新窗口分配了具体的层值，本步骤还需要调用 `finishUpdateFocusedWindowAfterAssignLayersLocked()` 做一些额外的处理。其内部很简单，仅仅是通知 `InputManager` 对象新的窗口信息，因为消息输入模块需要每一个窗口的层值，从而决定输入消息应该派发给哪个窗口。

⑤ 至此，添加窗口的操作基本完成，最后要做的仅仅是判断新窗口的 `configuration` 是否和系统当前的 `configuration` 相同。比如新窗口的 `origination` 是否和当前的不同，如果不同的话，则需要调用 `sendNewConfiguration()`，该函数的本质是调用 `AmS` 的 `updateConfiguration()`。由于 `sendNewConfiguration()` 操作需要系统权限，因此，该函数调用前先隐蔽了客户端身份，并在调用完毕后又恢复客户端身份。

### 14.3.2 assignLayersLocked()的执行过程

该函数的作用是根据窗口的类型及 `mBaseLayer` 的值，计算新的 `mLayer` 值，其过程虽然简单，但却体现了 `WmS` 中 `mWindows` 列表中窗口顺序的意义。

`mWindows` 中窗口的排列顺序反映了窗口应该被层叠顺序，而 `assignLayersLocked()` 函数的作用正是根据 `mWindows` 自身的顺序给窗口的 `mLayer` 进行赋值。为了更清楚地反映该函数的执行过程，假设当前 `mWindows` 列表中包含  $3 + 2 + 4$  个窗口，其顺序如图 14-16 所示，相同高度的线段代表窗口的类型相同。

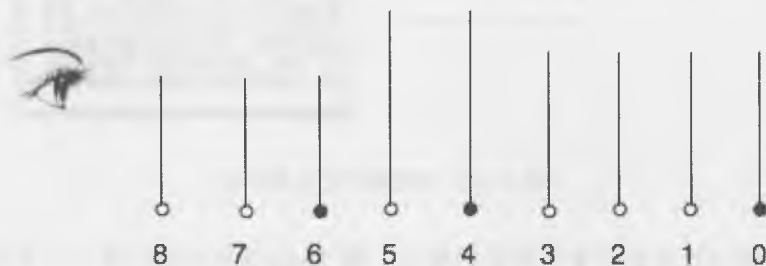


图 14-16 不同层值窗口相对人眼的逻辑位置



assignLayersLocked()的执行过程是先从 0 号窗口开始分配,0 号窗口的 mLayer 值等于 mBaseLayer, mBaseLayer 是在 WindowState 对象的构造函数中进行赋值的。与 0 号窗口类型相同的窗口是指 mBaseLayer 值相同,1、2、3 号窗口的类型和 0 号相同,因此 mLayer 的值分别为 mBaseLayer + 5, mBaseLayer + 2×5, mBaseLayer + 3×5。

4、5 号窗口的类型相同,其 mLayer 值分别为 4 号窗口的 mBaseLayer 和 mBaseLayer + 5。

6、7、8 号窗口类型相同,其 mLayer 值分别为 6 号窗口的 mBaseLayer 和 mBaseLayer + 5, mBaseLayer + 2×5。

下面来看 WindowState 构造函数中是如何根据窗口类型产生 mBaseLayer 的,相关代码如下:

```

6040         if ((mAttrs.type >= FIRST_SUB_WINDOW &&
6041             mAttrs.type <= LAST_SUB_WINDOW)) {
6042             // The multiplier here is to reserve space for multiple
6043             // windows in the same type layer.
6044             mBaseLayer = mPolicy.windowTypeToLayerLw(
6045                 attachedWindow.mAttrs.type) * TYPE_LAYER_MULTIPLIER
6046                 + TYPE_LAYER_OFFSET;
6047             mSubLayer = mPolicy.subWindowTypeToLayerLw(a.type);

6056         } else {
6057             // The multiplier here is to reserve space for multiple
6058             // windows in the same type layer.
6059             mBaseLayer = mPolicy.windowTypeToLayerLw(a.type)
6060                 * TYPE_LAYER_MULTIPLIER
6061                 + TYPE_LAYER_OFFSET;
6062             mSubLayer = 0;

```

即无论窗口类型 attrs.type 为 SUB\_WINDOW 还是非子窗口,都需要首先调用 mPolicy 的 windowTypeToLayerLw()函数得到一个类型层值,该函数非常简单。在 Policy 中定义了 13 种窗口类型的层值,分别对应 13 种窗口类型,得到类型层值后,再乘以 TYPE\_LAYER\_MULTIPLIER,该值为 10000,然后再偏移 TYPE\_LAYER\_OFFSET,其值为 1000。这样的话,假设窗口类型为 APPLICATION\_LAYER,其值为 2,那么该类型最终生成的 mBaseLayer 就是 2×10000 + 1000。TYPE\_LAYER\_MULTIPLIER 的语义是类型层值倍数,这也就是说,两个不同类型窗口之间最多可容纳 10000 个窗口。

当窗口类型为 SUB\_WINDOW 时,会使用其父窗口 attachedWindow 的类型计算类型层值,同时,还会调用 mPolicy 的 subWindowToLayerLw()计算窗口的 mSubLayer 值,该函数的执行很简单。Policy 中仅仅定义了五种子窗口类型,其对应的 mSubLayer 如以下代码所定义。

```

159     static final int APPLICATION_MEDIA_SUBLAYER = -2;
160     static final int APPLICATION_MEDIA_OVERLAY_SUBLAYER = -1;
161     static final int APPLICATION_PANEL_SUBLAYER = 1;
162     static final int APPLICATION_SUB_PANEL_SUBLAYER = 2;

```



### 14.3.3 addWindowToListInOrderLocked()的执行过程

该函数是在 `addWindow()` 中被调用的，其作用是将新建的 `WindowState` 对象添加到 `mWindows` 列表中。表面上看，该函数的执行过程有点复杂，但其执行的目的就是当添加新建的窗口后，`mWindows` 列表能够保持其内在顺序上的固有规则，因此，如果按照这种固有规则去阅读源码，会比较容易了解其具体执行过程。

决定新窗口在 `mWindows` 列表中次序的因素有三个，这三个因素是“或”的关系，而不是“与”的关系。

- 添加的顺序。比如两个类型相同的窗口，后添加的将处于前者的上面。
- 添加的类型。`Policy` 中定义了 13 种不同的类型，比如系统窗口总是位于应用类窗口的上面，因此，就算应用窗口是后添加的，它依然处于系统窗口之后。
- 子窗口类型。这种情况仅适用于当新窗口是一个子窗口时，该函数会根据不同的子窗口类型确定子窗口之间的顺序，这有点类似于第二种情况，只是范围更小而已。

图 14-17 给出了该函数的内部逻辑的语义，图中分支的关系体现了源码中各种 `if/else` 的逻辑关系，读者可以对照图 14-17 来理解源码。

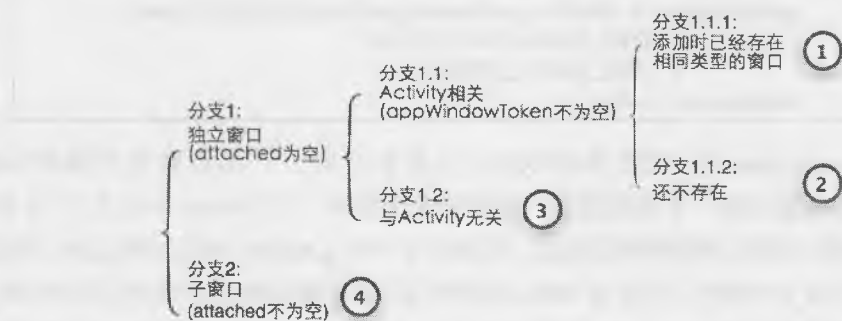


图 14-17 添加窗口中源码逻辑示意

该函数最终的判断逻辑包含了四个，下面分别介绍。

首先是 1.1.1 分支的逻辑，该逻辑的作用如图 14-18 所示。

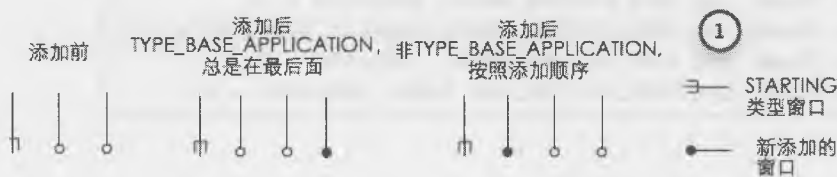


图 14-18 添加窗口逻辑一

该逻辑的特点有两个：当新窗口的类型为 `BASE_APPLICATION` 时，新窗口必须位于所有的窗口之后；否则，按照添加顺序排序，即最后添加的总是位于前面，而 `STARTING` 类型的窗口则始终位于最前面。

其次是 1.1.2 分支的逻辑，其逻辑的作用如图 14-19 所示。

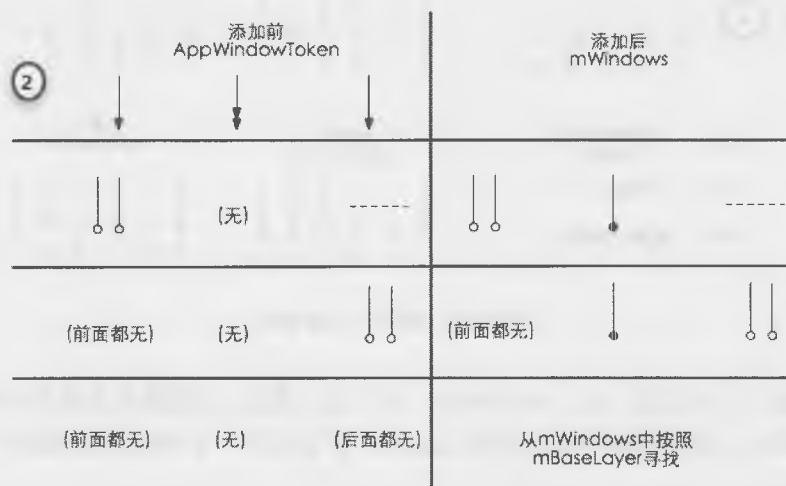


图 14-19 添加窗口逻辑二

由于该分支的新窗口类型是 `Activity` 相关的，而所有的 `Activity` 相关的窗口都事先对应一个 `AppWindowToken` 对象，每个 `AppWindowToken` 对象中包含的窗口在成员变量 `allAppWindows` 中。当然，`AppWindowToken` 对象也可能尚未包含任何窗口对象，因此逻辑中分别针对三种情况进行处理。遍历 `AppWindowToken` 列表时，假设目标 `token` 对象前面的 `token` 包含有窗口对象，则将新窗口放在该 `token` 所对应的窗口之后；假设目标 `token` 前面没有窗口对象，但后面的 `token` 对象却包含窗口，则将新窗口放到后面窗口之前；如果前后都没有窗口，则重新遍历 `mWindows` 列表，并根据 `mBaseLayer` 的值找到新窗口的位置。

第三是 1.2 分支的逻辑，其逻辑的作用如图 14-20 所示。



图 14-20 添加窗口逻辑三

该分支对应的新窗口类型是与 `Activity` 无关的，比如一些系统窗口，这些类型的窗口一般都是一个类型对应一个层值，因此只需要根据 `mBaseLayer` 遍历 `mWindows`，找到目标窗口 `mBaseLayer` 的位置，

然后将新窗口放到该位置上即可。

最后是 2 分支的逻辑, 如图 14-21 所示。

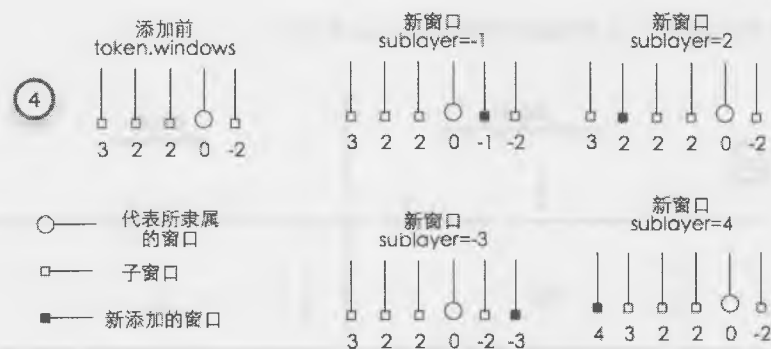


图 14-21 添加窗口逻辑四

该分支对应的新窗口类型是 SUB\_WINDOW, 即子窗口类型。该逻辑本身非常简单, 即按照 sublayer 的值从大到小排列即可, 然而源码中却又针对 sublayer 大于或小于 0 的情况分别进行了处理, 导致表面上看起来有点复杂。

该函数的内部过程反映了三个问题。

- 全局变量 mWindows 中列表顺序的意义是, 列表最后面的窗口在最上面, 即最靠近用户。
- WindowState 中的变量 mSubLayer 表示, 当该窗口为子窗口时, 相对于父窗口的层值。
- mAppTokens 列表中包含的都是 AppWindowToken 对象, 其顺序表示了其对应的窗口的顺序。AppWindowToken 类中的 allAppWindows 变量保存了对应该 token 的 Activity 相关的窗口, 比如一个 Activity 的启动窗口、关闭窗口, 以及 Activity 本身的窗口。
- 子窗口并不是 Activity 窗口所特有的, 任何非子窗口理论上都可以拥有子窗口, 比如状态栏、输入法窗口等, 它们都可以有子窗口。

#### 14.3.4 删除窗口的时机

Android 中的窗口和 Windows 操作系统中的窗口管理有一个明显差别, 就是窗口的右上角没有关闭按钮, 即用户不能直接关闭窗口。

从 SDK 的角度来看, SDK 希望程序员不要直接去操作窗口, SDK 已经对窗口进行了各种封装, Activity、菜单、对话框等, 这些控件的背后都对应一个窗口, 然而对程序员来讲, 不需要关心这些控件背后的窗口, 而是直接使用这些控件类。

对于 Activity 而言, 当程序员要关闭该窗口时, 可以调用 finish() 方法; 对于一个对话框, 当需要关闭其窗口时, 可以调用 dismiss() 方法。本节将要分析隐藏在这些方法背后的秘密, 即背后到底是如何真

正地删除一个窗口的。

当然, SDK 并没有完全封装窗口的操作, 它提供了一个 `WindowManager` 类, 当程序员需要一个完全自定义的窗口时, 依然可以调用该类的 `addView()` 方法添加一个窗口, 并可以调用 `removeView()` 删除一个窗口。这里 `View` 的语义实际上就是指一个真正的窗口, 尽管表面上看来这两个方法的操作是 `View` 对象, 而实际上内部会自动创建一个真正的窗口, 并将参数中的 `View` 对象作为窗口的可视对象。

因此, 删除窗口的时机可以大致分为两类。

第一类是隐性删除, 即不直接调用 `WindowManager` 类的 `removeView()` 来删除窗口, 比如 `Activity` 窗口, 程序只要调用 `finish()` 方法, 后台便会删除该 `Activity` 对应的窗口。

第二类是显性删除, 即直接调用 `WindowManager` 类的 `removeView()` 来删除窗口。

关于隐性删除, 本节仅介绍 `Activity` 对象的删除过程, 对于对话框、菜单框等窗口的删除过程, 其本质上都是通过调用 `WindowManager` 类的 `removeView()` 完成的, 读者可自行分析。

隐性删除常见的情况包括: 当用户按下 `Back` 键后, `AmS` 会间接调用 `destroyActivity()`; 当程序员调用 `finish()` 函数后, `AmS` 也会间接调用到 `destroyActivity()`; 当系统内存低时, `AmS` 也会调用到 `destroyActivity()`。总之, 这三种情况都是 `AmS` 调用 `ActivityThread` 的 `destroyActivity`, 关于其具体过程请参照本书第 10 章。

`ActivityThread` 中的 `scheduleDestroyActivity()` 函数中将处理删除窗口的操作, 其流程如图 14-22 所示。

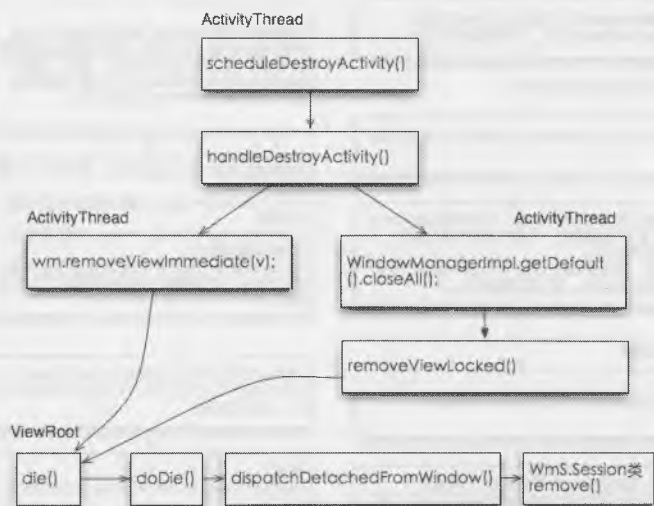


图 14-22 销毁 Activity 引起的窗口删除操作过程

首先, 无论何种情况, `WmS` 要销毁一个 `Activity` 相关的窗口时, 都必须调用 `ActivityThread.AppicationThread` 类的 `scheduleDestroyActivity()`。该函数是异步执行的, 它会发送一个 `Handler` 消息, 然后返回。消息的处理函数是 `handleDestroyActivity`, 该函数内部调用了两个重要函数, 第一个是 `removeViewImeediate()`, 另一个是 `closeAll()`。前者用于删除 `Activity` 窗口, 后者用于删除 `Activity`

相关的窗口，比如 Activity 的启动窗口、菜单窗口、对话框窗口等。

这两个函数又都会调用各自窗口对应的 ViewRoot 对象的 die() 函数，每个窗口都会对应于一个 ViewRoot 对象，该函数又会调用 doDie()，doDie() 又会调用 dispatchDetachedFromWindow()，该函数内部调用 sWindowSession.remove()。这是一个 IPC 远程调用，sWindowSession 是客户端中的一个全局静态变量，每个客户端对应一个该对象。

以上就是隐性删除窗口的时机，至于显性删除，则直接从 WindowManager 类的 removeViewImmediate() 开始，过程与隐性删除相同。

### 14.3.5 删除窗口的过程

上一小节中介绍了客户端最后调用到 Wms.Session 类的 remove() 方法，该方法中将完成真正删除窗口的过程。该过程可以看做是添加窗口的逆过程，主要包括对各种相关变量的赋值，以及 SurfaceFlinger 删除指定窗口。

remove() 函数内部仅仅是调用了 removeWindow()，因此，本节就从该函数开始分析，其执行过程如图 14-23 所示。

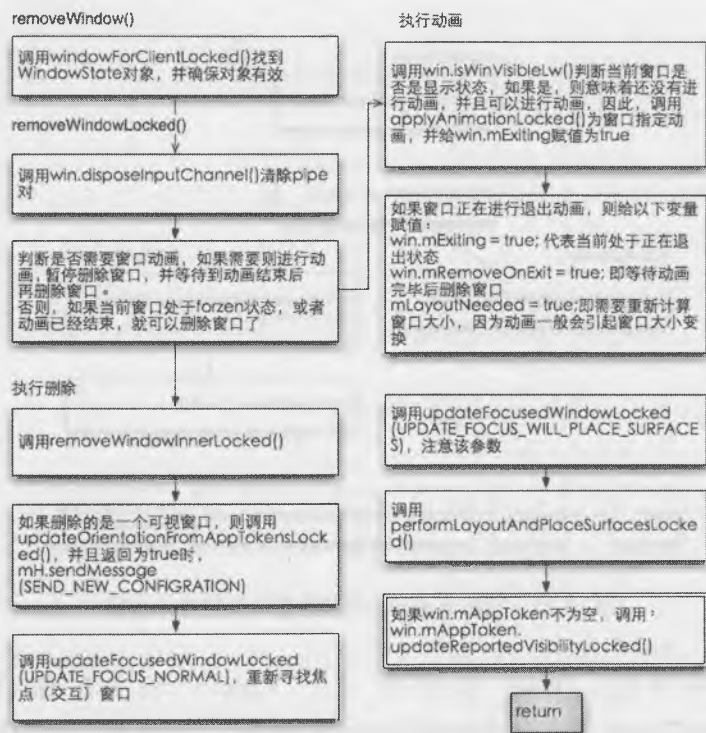


图 14-23 removeWindow()的内部执行流程

该函数中, 首先调用 `windowForClientLocked()` 寻找目标窗口是否存在, 读者可能觉得奇怪, 既然是删除窗口, 窗口为什么会不存在呢? 典型的情况就是当程序员连续调用两次 `removeWindow()`, 并且参数相同时, 则第二次调用时, `windowForClientLocked()` 就会返回为空。该函数内部逻辑很简单, 就是从全局变量 `mWindowMap` 中查看目标窗口是否存在, 对 `mWindowMap` 而言, 一个窗口就是指 `ViewRoot` 类中的 `W` 对象, 它是一个 `IWindow` 对象。

如果存在, 则调用 `removeWindowLocked()` 开始删除, 该函数内部的关键逻辑是, 首先会为窗口添加一个动画, 这就是所谓的窗口关闭动画, 并且直到该动画结束后才真正删除窗口。具体执行过程如下。

① 调用 `win.disposeInputChannel()`, 其作用是清除 `pipe` 对, 因为在创建窗口的同时创建了一个 `pipe` 对用于传递输入消息, 而此时窗口就要被删除了, 所以 `pipe()` 对也就应该被删除。由于删除 `pipe` 对发生在窗口关闭动画之前, 因此, 在动画运行期间, 用户虽然能够看见该窗口, 但是却不能同该窗口交互。

② 判断是否可以为该窗口添加一个动画。可以添加动画的条件有三: 第一是该窗口已经存在 `Surface`; 第二是屏幕没有被冻结 (`frozen`), 冻结是指当屏幕发生旋转时, 系统会暂时冻结屏幕, 直到屏幕被重绘; 第三是屏幕处于打开状态 (`screen on`), 当发生锁屏后, 屏幕就处于非打开状态。

(1) 如果可以进行动画, 则先调用 `win.isWinVisibleLw()` 判断当前是否已经设置过动画, 因为如果设置过, 该函数就会返回 `false`, 即当窗口正在进行动画时, 是不可见的。因此, 如果该函数返回为 `true`, 则说明这是第一次调用 `removeWindowLocked()`, 关闭动画还没有设置, 因此, 调用 `applyAnimationLocked()` 为该窗口设置一个动画。关于窗口动画的详细信息见后面小节。

(2) 如果 `isWinVisibleLw()` 返回 `false`, 则说明已经设置过动画, 因此继续判断动画是否结束, 如果没有结束, 则调用 `performLayoutAndPlaceSurfacesLocked()` 对窗口进行重排。该函数是 `WmS` 内部的最核心函数, 其作用是根据窗口列表的参数值, 重新计算窗口大小, 并调整窗口层值, 使之真正被反映到屏幕上, 关于该函数的内部执行过程见后面小节。重排结束后就直接返回, 因为正在进行动画, 所以不能删除窗口。

③ 能执行到本步, 说明动画已经结束, 事实上, 动画结束的标志是 `win` 对象的 `mSurface` 为空, 关于这一点将在动画小节中介绍。此时就可以调用 `removeWindowInnerLocked()` 方法, 该方法将从记录目标窗口的变量中移除目标窗口, 具体执行过程见下一小节。

④ 窗口删除后, 下一个焦点窗口的屏幕旋转方向有可能与当前的旋转方向不一致, 因此需要调用 `updateOrientationFromAppTokensLocked()` 查看是否需要调整旋转方向。如果需要, 则发送一个异步 `Handler` 消息 `SEND_NEW_CONFIGURATION`, 消息的处理函数将通知 `AmS` 设置新的旋转方向。

⑤ 最后, 调用 `updateFocusedWindowLocked()` 寻找新的焦点窗口。

### 14.3.6 removeWindowInnerLocked()

该函数的作用是从各种记录窗口信息的变量中移除目标窗口对应的记录, 因此了解清楚记录窗口相关的变量就很容易理解该函数的执行过程。该函数有两个参数, 分别为 `Session session` 和 `WindowState`

win, 第一个参数实际上并没有被使用, 第二个参数代表要被删除的目标窗口。

保存窗口信息的主要变量包括:

- `HashMap mWindowMap<IBinder, WindowState>`。
- `ArrayList<WindowState> mWindows`。
- `ArrayList<WindowState> WindowToken.windows`, 一个 `WindowToken` 可能会对应多个 `WindowState` 对象, 这些 `WindowState` 对象保存在 `windows` 列表中。
- `ArrayList<WindowState> AppWindowToken.allAppWindows`, 一个 `AppWindowToken` 可能会对应多个 `WindowState` 对象, 这些 `WindowState` 对象都和 `Activity` 相关, 并且这些窗口对象保存在 `allAppWindows` 列表中。
- `HashMap mTokenMap<IBinder, WindowToken>`, 不是每个窗口都对应一个 `WindowToken` 对象。
- `ArrayList mTokenList<WindowToken>`。
- `WindowState mInputMethodWindow`。
- `ArrayList<WindowState> mInputMethoDialogs`。

注意 `mWindowMap` 和 `mTokenMap` 中 `IBinder` 的区别, 前者对应的是一个 `ViewRoot.W` 对象, 而后的值会根据窗口类型有所不同, 两者的添加过程如图 14-24 所示。

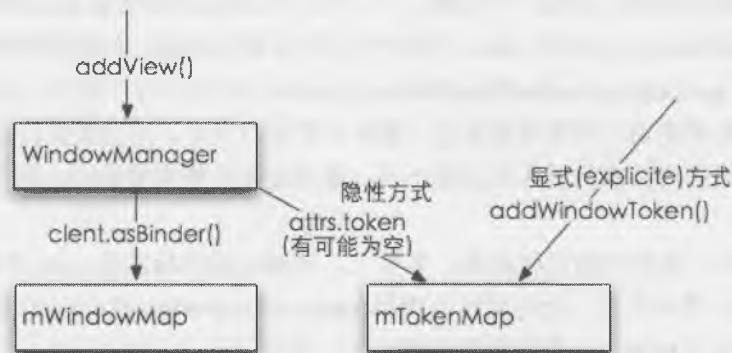


图 14-24 添加 WindowToken 对象的两种方式

所有的窗口添加都是经过 `WindowManager` 类的 `addView()` 函数, 该函数会最终给 `WmS` 的 `mWindowMap` 列表中添加一个新的 `WindowState` 对象, 并且该对象的 `IBinder` 是由 `client.asBinder` 而来的。而 `client` 则是指 `ViewRoot.W` 对象, 给 `mWindowMap` 添加的同时, 也会给 `mTokenMap` 列表中添加一个 `WindowToken` 对象, 该对象中的 `token` 来自于 `attrs.token` 的值, 该值可能为空, 如果为空, `mTokenMap` 中将不包含新产生的 `WindowToken` 对象。给 `mTokenMap` 中添加 `WindowToken` 的另外一个地方就是调用 `WmS` 的 `addWindowToken()` 函数, 该函数系统中有两处调用, 一处是在 `WallpaerManagerService` 中, 另一处是在 `IMMS` 中。换句话说, 墙纸窗口和输入法窗口对应的 `WindowToken` 对象是在调用 `addView()` 函数之外单独添加的。



下面具体来看 removeWindowInnerLocked() 的执行过程，如图 14-25 所示。

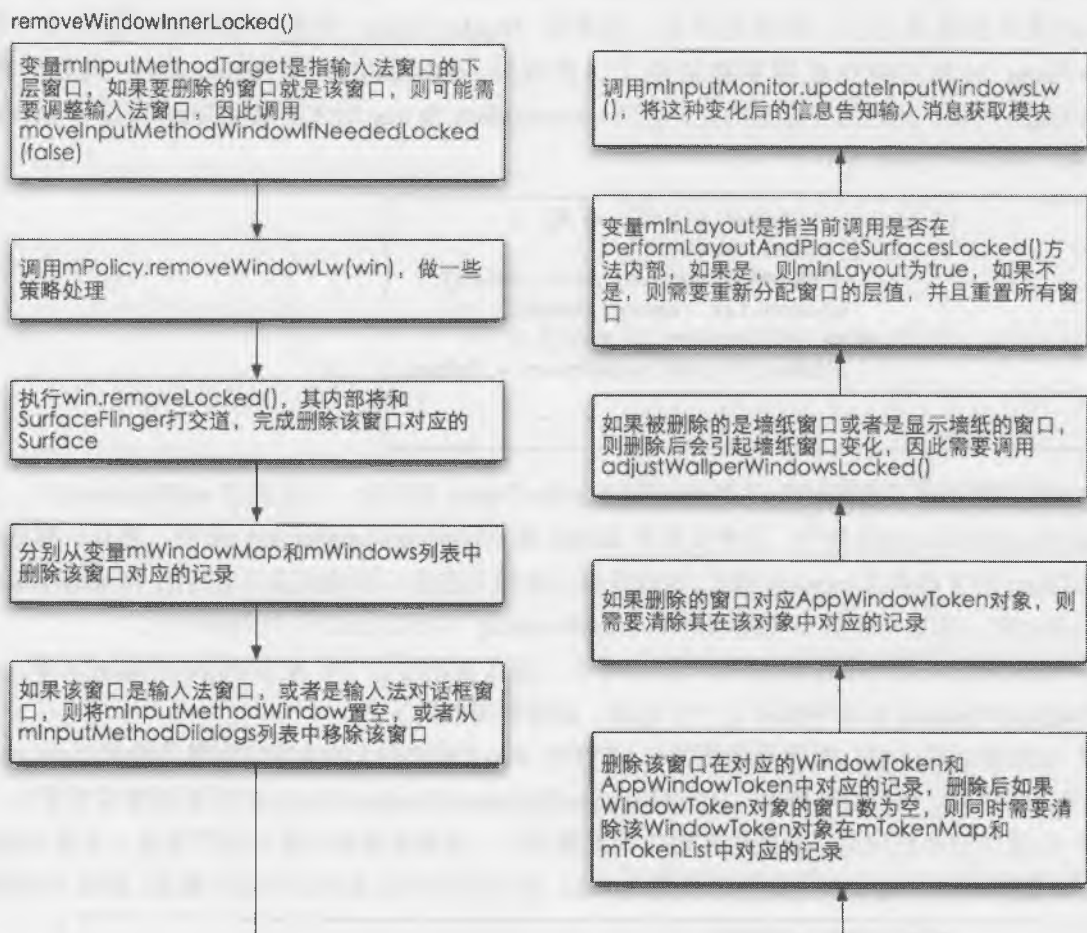


图 14-25 removeWindowInnerLocked() 内部执行流程

- ① 判断要删除的窗口是否是 mInputMethodTarget。该变量保存的是输入法下层的窗口，因此如果要删除的窗口就是该窗口，则可能需要调整输入法窗口。
- ② 如前面对“策略”的解释，此处删除窗口时，也需要先通知策略，以便它在删除之前做点什么。
- ③ 执行 win.removeLocked()，该函数内部真正完成删除窗口对应的 Surface 操作。
- ④ 分别从 mWindowMap 和 mWindows 列表中删除该窗口对应的记录。
- ⑤ 由于 WmS 内部用了另外两个变量保存和输入法窗口相关的窗口，因此如果此处删除的就是输入法相关的窗口，那么同时需要从这两个变量中清除相关记录，分别是 mInputMethodWindow 和 mInputMethodDialogs。前者是一个 WindowState 对象，对应输入法窗口，系统中只能有一个输入法窗口；

后者是输入法对话框窗口，可以有多个输入法对话框。

⑥ 由于每个窗口在它对应的 WindowToken 对象中还有一个记录，即 token.windows 列表，因此还需要从列表中清除该记录。清除完毕后，如果该 WindowToken 对象内部的窗口数为 0，那么该 WindowToken 对象本身也就需要被清除了，因此从 mTokenMap 和 mTokenList 列表中清除该 WindowToken 对象。此处需要注意的是，仅当 token.explicit 为 true 时才清除 mTokenMap 和 mTokenList 中的记录，如以下代码所示：

```

2114         if (token.windows.size() == 0) {
2115             if (!token.explicit) {
2116                 mTokenMap.remove(token.token);
2117                 mTokenList.remove(token);
2118             } else if (atoken != null) {
2119                 atoken.firstWindowDrawn = false;
2120             }
2121         }

```

explicit 的意思是“显式的”，系统中创建 WindowToken 有两处，一处是在 addWindow() 时，另一处是在 addWindowToken() 函数中，后者仅用于 IMMS 和 WallpaperManagerService 中，并且只有后者创建 WindowToken 时才使用了 explicit 模式。也就是说只有输入法窗口和墙纸窗口对应的 WindowToken 对象才是 explicit 的，即这两个窗口在删除时不从 mTokenMap 和 mTokenList 中清除记录。

⑦ 如果该窗口是一个和 Activity 相关的窗口，那么它还对应一个 AppWindowToken 对象，并在该对象的 allAppWindows 列表中保存了一个记录，因此需要清除。

⑧ 如果被删除的窗口类型是墙纸窗口 (TYPE\_WALLPAPER) 或者窗口背景是墙纸 (win.attrs.flags 包含 FLAG\_SHOW\_WALLPAPER)，则调用 adjustWallpaperWindowsLocked() 重新调整墙纸窗口。

⑨ 如果当前函数调用不是发生在窗口重置调用中，则需要重新对窗口进行重置。变量 mInLayout 在重置调用前被置为 true，并在调用后被置为 false，用于标识当前是否正在进行重置，如以下代码所示：

```

8327         mInLayout = true;
8328         try {
8329             performLayoutAndPlaceSurfacesLockedInner(recoveringMemory);

```

如果当前 mInLayout 为 false，则首先重新分配窗口层值，然后调用重置函数，如以下代码所示：

```

2149         if (!mInLayout) {
2150             assignLayersLocked();
2151             mLayoutNeeded = true;
2152             performLayoutAndPlaceSurfacesLocked();
2153             if (win.mAppToken != null) {
2154                 win.mAppToken.updateReportedVisibilityLocked();
2155             }
2156         }

```

⑩ 至此，已经清楚了被删除窗口相关的全部信息，最后要做的就是把这种变化通知给消息输入模块，从而使其能够根据这种变化选择正确的新的焦点窗口。通知的方法很简单，就是调用

mInputMonitor.updateInputWindowsLw()。

## 14.4 计算窗口的大小

本节将介绍 WmS 如何为窗口分配大小，并当输入法窗口显示时，如何调整相应窗口的大小，以适应这种变化。

### 14.4.1 描述窗口尺寸的变量

WmS 中用 WindowState 类表示一个窗口，该类中和窗口尺寸相关的变量关系如图 14-26 所示。

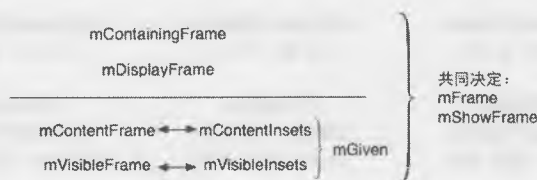


图 14-26 窗口尺寸相关的变量关系

首先是 mContainingFrame，它代表的该窗口的父窗口（如果存在的话）的矩形区，如果父窗口不存在，则其值一般为屏幕的大小。

mDisplayFrame 的意义和 mContainingFrame 虽说有所区别，但其值永远相同，因此它们两者的意义可以认为是相同的。

其次是 mContentFrame 和 mVisibleFrame，由于两者的值总是相同，因此可以认为其意义也是相同的，它们代表的是窗口内容的实际尺寸。

第三是 mContentInsets 和 mVisibleInsets，它们代表的是 Content 的边框空白区域。

第四是 mFrame 和 mShowFrame，其中 mFrame 代表的是该窗口从程序上来看在屏幕上应该占据的矩形区，mShowFrame 代表的是该窗口最终的屏幕上的矩形区。如果没有动画的话，两者的值是相同的，而一旦该窗口正在进行动画，则 mShowFrame 会不断变化，而 mFrame 则保持不变。

最后是 mGivenContentFrame 和 mGivenVisibleFrame，其值与 mContentInsets 和 mVisibleInsets 相同，因此其意义可认为是相同的。

这些变量全部都是 Rect 类型，但其中 insets 变量的数值的意义是距离相应边框的距离，insets 的本意就是“空余、填充物”。

为了从感性上理解这些变量的含义，图 14-27 以三个典型的例子来说明其作用。

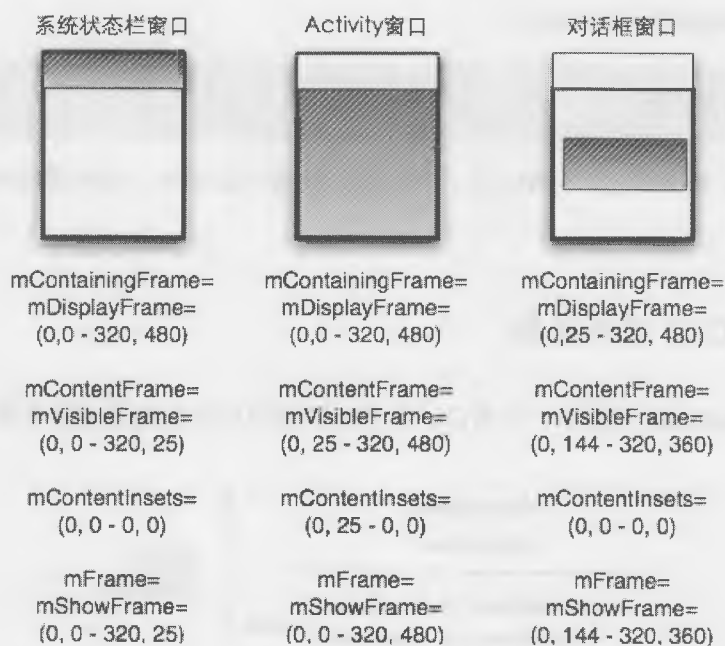


图 14-27 不同尺寸的窗口

该图中黑色栅格线填充区代表目标窗口，粗实线区域代表其 mContainingFrame 区域。从本例也可以看出这些变量之间遵循的逻辑关系，如下：

第一，由于系统状态栏窗口没有父窗口，因此其 mContainingFrame 的区域就是整个屏幕，而对话框窗口的父窗口是 Activity 对应的窗口，因此 mContainingFrame 的值就是 Activity 窗口的内容区域。同理，Activity 窗口由于没有父窗口，所以 mContainingFrame 的值就是整个屏幕区域。

第二，mContentFrame 代表的是该窗口的实际大小，图中分别为栅格线填充区。

第三，mFrame 代表将在屏幕上显示的显示区域，对于系统状态栏窗口和对话框窗口而言，显示区域就是 mContentFrame 的区域，而对于 Activity 窗口而言，则包含了 mContentInsets 区域。也就是说，mFrame = mContentFrame + mContentInsets 的复合区域。

而计算窗口大小的本质就是根据窗口的类型及相关参数为窗口分配具体的尺寸，并使这些尺寸满足以上的逻辑。

#### 14.4.2 窗口大小的变化过程

窗口的变化过程可简要描述为如图 14-28 所示。

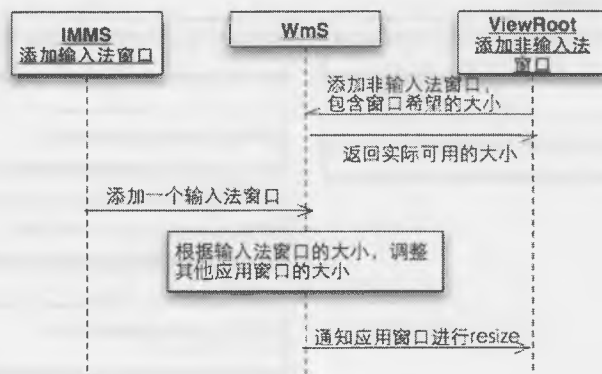


图 14-28 窗口大小变化的两种原因

无论是添加输入法窗口还是非输入法窗口，都是经过 ViewRoot 的，该图中将 IMMS 和 ViewRoot 分开的原因仅仅是为了显示两者的区别。

首先，当添加一个普通窗口（非输入法窗口）时，会调用 WmS.Session 类的 `relayout()` 函数，该函数中包含了窗口希望的大小，而这个大小则是对窗口中的 View 对象进行 `measure` 时获得的，如以下代码所示。

```

2647     int relayoutResult = sWindowSession.relayout(
2648         mWindow, params,
2649         (int) (mView.mMeasuredWidth * appScale + 0.5f),
2650         (int) (mView.mMeasuredHeight * appScale + 0.5f),
2651         viewVisibility, insetsPending, mWinFrame,
2652         mPendingContentInsets, mPendingVisibleInsets,
2653         mPendingConfiguration, mSurface);|
  
```

在这段代码中，`mView.mMeasuredWidth` 和 `mMeasuredHeight` 就是该窗口希望的大小，而其他参数——`mWinFrame`、`mPendingContentInsets` 等矩形变量都是输出参数。当 WmS 中执行完该函数后，为填充这些矩形变量，告知该窗口实际上应该在屏幕上的相应尺寸。

以上就是普通窗口的初始大小，而在程序运行的过程中，尤其是当该窗口需要和输入法窗口进行交互时，假如该窗口中包含一个编辑框，当用户点击编辑框后，IMMS 中会添加一个输入法窗口，此时 WmS 需要根据输入法窗口重新调整目标窗口的大小，并将通过 IPC 回调 ViewRoot.W 类的 `resized()` 函数，通知目标窗口新的窗口大小，此时目标窗口会自动重新进行 `measure` 和 `layout` 操作，并重绘。

而当输入法窗口消失时，WmS 同样会重新计算所有窗口的尺寸，并通过客户端窗口进行 `resize` 操作。

这就是窗口变化的简要过程，下面继续分析其中的一些细节过程。

首先来看在 WmS 中的 `relayoutWindow()` 中是如何根据客户端传入的窗口类型和希望的尺寸来计算实际的窗口尺寸的。该过程中的关键调用如图 14-29 所示。

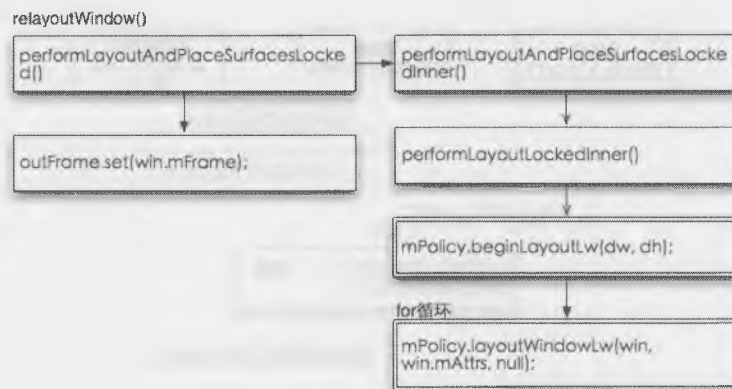


图 14-29 relayWindow()内关键调用

relayWindow()中包含两处调用，第一个调用是为了计算窗口的大小，调用完毕后，win.mFrame中的值就是实际的窗口大小。因此就把这个调用设置给 outFrame 矩形变量，该变量正是 ViewRoot 中调用 sWindowSession.layout()中参数 outFrame。

下面来看 mPolicy.beginLayoutLw(dw,dh)。该函数中参数 dw 和 dh 来源于屏幕的宽度和高度，由于真正的计算发生在 mPolicy 之中，因此，本函数的作用就是要告诉 Policy 当前的屏幕大小。该函数内部仅仅是给 Policy 中的几个矩形变量赋值，这些矩形变量包括三类，分别是 mDockXXX、mCurXXX 和 mContentXXX，这三个变量的实际意义如图 14-30 所示，黑实线区域代表矩形变量的有效区。



图 14-30 Policy 中保存窗口尺寸的几个变量意义

三者的关系如下：

第一，三者的 mTop 值都是状态栏窗口的下边沿，这一点从以下代码可以看出。

```
1285 mDockTop = mContentTop = mCurTop = mStatusBar.getFrameLw().bottom;
```

第二，mCurXXX 和 mContentXXX 的值完全相同，并且其下边沿是输入法窗口的上边沿，而 mDockXXX 的大小是除状态栏之外的所有区域。

beginLayoutLw()的作用就是给 mPolicy 中的这些矩形变量赋初值。

赋完初值后，就循环调用 mPolicy.layoutWindowLw()计算每一个窗口的实际大小，下一节将介绍该函数的内部执行过程。

### 14.4.3 Policy 中 layoutWindowLw()的执行过程

该函数的作用是计算窗口的实际大小。mPolicy 中计算大小的思路如下：首先 mPolicy 会从 beginLayoutLw()被调中获取屏幕的大小及系统状态栏的大小，然后开始依次计算窗口的大小，每次计算时都先计算当前可用的区域，然后再调用窗口的 computeFrameLw()方法，由窗口本身调整自身的大小。该过程有点类似于 View 系统中计算每个视图大小的过程。换句话说，mPolicy 首先计算出窗口的最大可用空间，然后再由窗口自身在这个范围内确定自己的最终实际大小。

Policy 中定义了四个全局矩形变量，用于临时存放 parent、display、content、visible 对应的矩形区，变量以 mTmpXXX 为前缀，并在函数中一般使用四个简写变量 pf、df、cf、vf 代替。

首先来看 beginLayoutLw()的执行过程，其内部流程如图 14-31 所示。

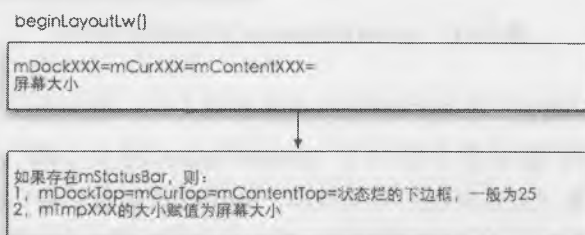


图 14-31 beginLayoutLw()主要过程

接下来，会在 for 循环中依次调用 layoutWindowLw()，该函数的流程如图 14-32 所示。

① 如果是输入法窗口，则仅仅是给 pf、df、cf、vf 赋值为 mDockXXX，此时 mDockXXX 的大小是在 beginLayoutLw()中被赋值的，即为屏幕大小。大家可能觉得奇怪，输入法窗口怎么会是全屏呢？别奇怪，这里再次强调，layoutWindowLw()的作用是给出窗口的可用大小，而不是最终窗口的大小。

② 如果不是输入法窗口，则又分为三种情况，因为不同情况下的窗口大小限制不同。

(1) 第一种情况是指需要考虑状态栏的窗口，常见的就是 Activity 对应的窗口，这些标志信息保存在 attrs.flags 变量中。在这种情况下，窗口的可用大小基本上就是屏幕大小，但当窗口的 softInputMode 为 SOFT\_INPUT\_ADJUST\_RESIZE 时，则需要额外考虑输入法窗口，即当包含该标志时，contentFrame 的大小应该不包含输入法窗口所在区域。



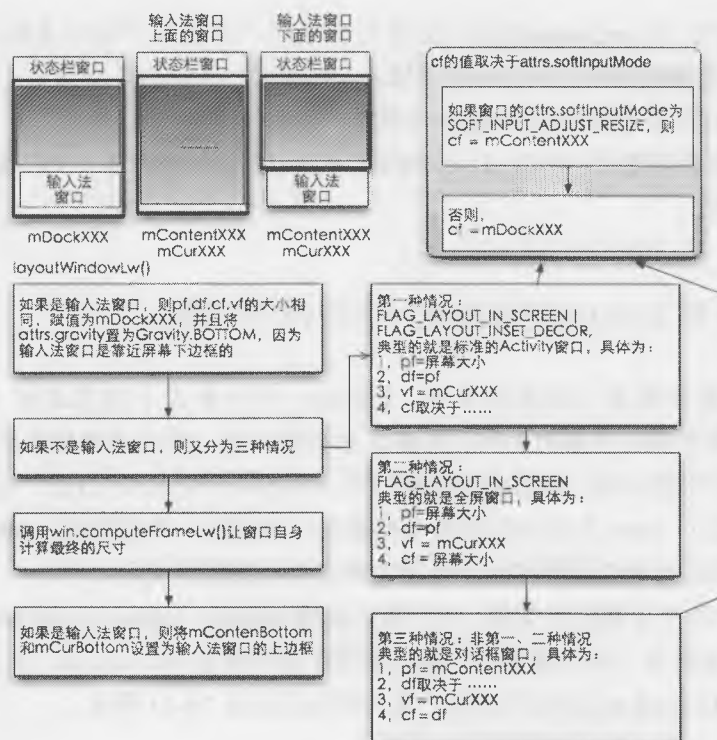


图 14-32 layoutWindowLw()的内部执行过程

(2) 全屏模式, 在这种情况下, 窗口的大小就是屏幕大小。但注意, 这并不意味着目标窗口可以全部被显示出来, 因为能够显示多少取决于 contentFrame 的大小和 visibleFrame 的大小。尽管 contentFrame 是全屏, 而如果 visibleFrame 不是全屏, 则用户只能看到窗口的一部分。

(3) 对于其他情况, 首先 pf 的大小为当前 mContentXXX 的大小, 而 df 和 cf 相同, 都取决于窗口的 softInputMode。这个和第 ② 步中 (1) 的状况相同, 即如果考虑输入法的话, cf 的大小应该排除输入法窗口所在区域。

③ 执行到这步时, 窗口的可用大小就确定了, 而最终窗口的大小还需要由窗口自己去决定, 关于 computeFrameLw() 的内部过程在下面继续介绍。

④ 在执行本步内容之前的所有窗口的可用大小中 vf 的值都是全屏的, 因为 vf 的值等于 mCurXXX 的值, 而 mCurXXX 只有在该步骤中被改变。本步的作用是判断窗口是否为输入法窗口, 如果是, 则将 mCurBottom 和 mContentBottom 置为输入法窗口的上边沿。这两个变量是 Policy 中的全局变量, 应用于所有的窗口, 当本步执行后, 后面的窗口的可用大小都将排除输入法窗口所在的区域。这一点的巧妙在于 mWindows 列表中的窗口是 z-ordered 排序的, 即列表最后的窗口是最靠近用户的, 而 layoutWindowLw() 在 for 循环被调用时正是从最靠近用户的窗口开始, 因此, 只有输入法窗口之后的窗口才会考虑输入法的存在。

下面继续看 win.computeFrameLw() 的执行过程。

该过程主要包含四个步骤，四步围绕一个核心逻辑进行，即“在 xyz 的限制下，计算最终的窗口尺寸”，xyz 是指不同的条件，如图 14-33 所示。

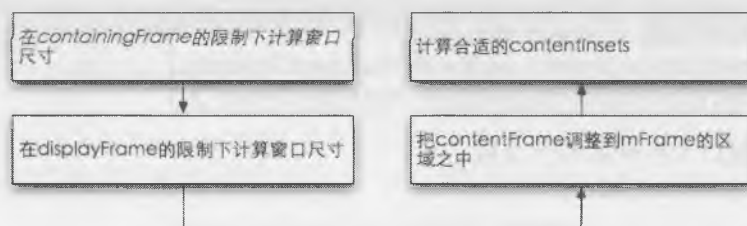


图 14-33 计算窗口尺寸所经历的各种条件限制

① 在 containingFrame 的限制下计算窗口大小，源码中使用了 Gravity 类的 apply() 函数来进行计算，如以下代码所示：

```

6146 Gravity.apply(mAttrs.gravity, w, h, container,
6147               (int) (mAttrs.x + mAttrs.horizontalMargin * pw),
6148               (int) (mAttrs.y + mAttrs.verticalMargin * ph), frame);
  
```

该段代码的关键在于参数 w 和 h，这两个值是窗口期望的宽度和高度，其最初来源于客户端在进行 measure() 操作时，计算的窗口内部视图的宽度和高度。此时，如果窗口的大小属性为 MATCH\_PARENT，则窗口的最终大小将等于 pf 的大小，而如果属性为 WRAP\_CONTENT，则就是窗口中视图的大小，如以下代码所示：

```

6129 w = mAttrs.width == mAttrs.MATCH_PARENT ? pw : mRequestedWidth;
6130 h = mAttrs.height == mAttrs.MATCH_PARENT ? ph : mRequestedHeight;
  
```

apply() 函数的最后一个参数 frame 是一个输出变量，即函数中将根据 mAttrs.gravity 和 w、h 及 container 的大小来确定 frame 的大小。

② 此时 frame 的大小基本上就是最终窗口的大小了，但是为了以防万一，因此还需要在 df 的限制下调整 frame 的大小，此处使用以下代码进行调整：

```

6152 // Now make sure the window fits in the overall display.
6153 Gravity.applyDisplay(mAttrs.gravity, df, frame);
  
```

由于几乎所有的 pf 和 df 的大小都相同，因此本步执行结果对 frame 无影响，可删除该段代码。

③ contentFrame 变量的含义在本步之前都是指目标窗口最大的可用空间，而此时，这个含义已经有所变化，它即将限制到窗口最终区域之中。即当 contentFrame 的区域超出 frame 时，被限定到 frame 的区域；当 contentFrame 没有超出 frame 的区域时，则保持原有的区域。

④ 上一步中对 contentFrame 的区域进行了调整，如果 contentFrame 和 frame 完全吻合，则 insets 就为空，因为不需要任何填充，而如果 contentFrame 区域小于 frame 区域，则需要一定的填充，因为客户端窗口使用 frame 作为绘制区间。计算填充值的方法就是用 frame 减去 contentFrame。

执行完以上所有操作后，也就完成了从 ViewRoot 调用 sWindowSession.relayout() 的操作，此时，

relayout()函数中的输出参数将被填充合适的变量, 主要包括:

mWinFrame 将被填充为 WindowState 的 mFrame; mPendingContentInsets 将被填充为 WindowState 的 mContentInstes, 如以下代码所示。

```
2529         outFrame.set(win.mFrame);
2530         outContentInsets.set(win.mContentInsets);
2531         outVisibleInsets.set(win.mVisibleInsets);
```

至此, 就完成了窗口大小计算的全过程。

#### 14.4.4 输入法窗口如何影响应用窗口的大小

事实上, 上一节已经介绍了输入法窗口影响其他窗口大小的过程, 本节之所以单独列出, 是因为很多读者对输入法窗口影响应用窗口的过程比较好奇, 并认为该过程区别于其他窗口大小变化的过程。

本节要强调的是, 添加输入法窗口和普通窗口的主要执行过程是一致的, 唯一的区别就在于 Policy, PhoneWindowManager 这个 Policy 处理输入法时有两个特点:

- 只允许输入法窗口被添加一次。
- 认为输入法窗口下面不应该显示其他窗口的任何内容, 所以, 当计算窗口时, 对于输入法后面的所有窗口的区域被限制在了输入法区域之外, 这就导致输入法显示或隐藏时, 其后面窗口的大小会发生变化, 仅此而已。

应用窗口能够根据输入法窗口的显示、隐藏而重新设置大小的原因并不是因为它是输入法窗口, 而是因为它的出现导致 WmS 通过 IPC 回调到了 ViewRoot.W 类的 resized()方法。因此, 任何能够回调该 resized()方法的行为都会导致应用窗口重新设置大小。比如常见的一种情况就是在工程模式下, 可以开启显示 CPU 利用率窗口, 这同样会导致应用窗口重新计算大小。所以读者在心中要有这样一个概念: 窗口大小变化和输入法显示并无直接关系, 输入法窗口和应用窗口可以看做是典型的一个多任务多窗口, 就像我们一边在使用计算器程序, 一边在使用 Word 程序, 如图 14-34 所示。

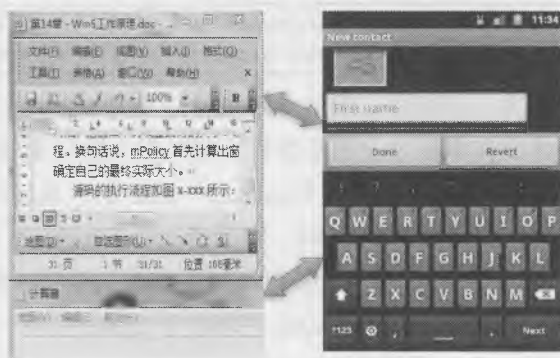


图 14-34 输入法是独立的应用进程示意

偶尔有读者会问,如何得知当前的软键盘是否处于显示状态?且不论这种需求是否合理,先来打一个比方,这就好比 Word 程序想知道计算器程序的窗口是否处于显示状态,有这个必要吗?这就是为什么 InputMethodManager 类中没有提供查询软键盘是否显示的 API 接口的原因,事实上,程序员可以在 Manifest 文件的 Activity 标签中配置 softInputMode,告知 WmS 该 Activity 是否会因为输入法窗口的显示或隐藏被 resize,除此之外 Activity 窗口几乎不需要再与输入法窗口本身进行任何交互。有些读者可能会争辩:“查询当前输入法窗口是否显示可以用来对 Activity 窗口的大小做一些调整。”然而,Activity 窗口被 resize 的条件表面上是输入法窗口,而实际上则有可能来自其他的一些系统窗口,因此,Activity 窗口仅仅根据输入法窗口做界面调整是不合适的,而应该根据 WmS 传递过来的 resize 尺寸做调整。

## 14.5

## 切换窗口

“切换”是指当有新窗口出现时,输入焦点需要从老窗口移动到新窗口,或者是当前窗口被关闭时,上一个窗口应该获得输入焦点。Android 中的窗口系统中,用户直接操作的并不是窗口,而是 Activity,因此 Activity 的启动和关闭和窗口的启动和关闭之间需要一定的同步关系。

本节内容的组织思路是,首先提出切换窗口时要解决的具体问题,读者可以带着这些问题阅读后续章节及源码,从而能够有针对性地把握其内部主要逻辑。

然后介绍 InputManager 和 WmS 的接口,前者负责判断输入消息属于哪个窗口,而后者则记录了所有窗口的状态信息,WmS 中将根据这些绘制屏幕。在 InputManager 和 WmS 之间必须保证一定的同步,从而确保“所见即所得”,即当用户看到窗口时,相应的触摸或者按键消息能被正确派发到所看见的窗口中。

接着再介绍 AmS 和 WmS 的具体接口,这些接口将用于在 Activity 的启动、关闭和窗口的启动、关闭之间进行逻辑同步。

介绍了以上的逻辑框架后,后面将以三个最常见的切换场景为例,介绍切换的具体执行过程,这些过程将涉及 InputManager、AmS、WmS 之间的同步操作,以便读者能清楚地看到这三者之间的代码关系。这三个常见场景分别是当从 A 启动到 B 时,以及从 B 又回到 A 时,以及长按“Home”键,从最近任务列表中又选择 B 时。

介绍了以上过程后,后面的几个小节将针对以上操作过程中的一些重要函数内部过程进行分析,以便读者再对源码中的一些相关类的相关变量含义有所把握。

### 14.5.1 切换要解决的问题

Android 中的窗口管理系统中,首先在 AmS 中保存所有与 Activity 相关的信息,在 Activity 启动和关闭时,AmS 会通知 WmS 同步 Activity 窗口的状态,而在 WmS 中使用 WindowState 类保存一个窗口的信息,这些窗口信息需要根据 Activity 的状态而动态改变。在 WmS 中另外有一个 InputManager 对象,该对象内部保存了输入消息处理时所需要的窗口信息,借助这些信息,InputDispatcher 能够决定输入消

息应该对应哪个窗口，WindowState 类的信息必须与 InputManager 内部的窗口信息也保持同步。

切换过程中要解决的问题可归纳为三类，第一类是状态同步问题，即 AmS 如何把状态传递给 WmS，WmS 如何保存这些状态，又如何把这些状态传递给 InputManager；第二类问题是屏幕绘制问题，即当 AmS 启动或者关闭一个 Activity 时，用户一般会看到一个动画，那么 WmS 如何定义这个动画，并如何在动画绘制前先隐藏目标窗口，直到动画结束后才显示目标窗口，而要实现这种动画的绘制就需要一种特别的变量来保存动画窗口和目标窗口的关系；第三类问题是消息处理问题，即在窗口切换的过程中，是老窗口应该继续捕获用户消息呢还是新窗口？

首先来看状态同步问题。

在 AmS 中使用 ActivityRecord 类来保存一个 Activity 相关的信息，ActivityRecord 类本身是一个 Binder 类，名称为 IApplicationToken.Stub。每个 ActivityRecord 都会在 WmS 中对应一个 AppWindowToken 类，该类保存了和 ActivityRecord 相关的所有窗口信息，比如启动窗口、实际窗口、关闭窗口等。当启动一个新的 Activity 时，AmS 中会先创建一个 ActivityRecord 对象，并请求 WmS 中也创建一个 AppWindowToken 对象；当销毁一个 Activity 时，AmS 会请求 WmS 删除 AppWindowToken 对象。

AppWindowToken 中包含的窗口对象在 WmS 中的 mWindows 等列表变量中也都有记录，当新窗口启动时，必须保证这些列表中的对象和 AppWindowToken 中保存的窗口信息之间的同步。

WmS 中有一个 InputManager 对象 mInputManager，该对象中保存了输入消息处理所需的窗口信息，当有新窗口添加或者旧窗口被删除时，该对象中的窗口信息同时需要被更新。

下面再来看绘制问题。当新建一个窗口时，AmS 会首先判断该窗口是同一个 Task 中的 Activity 还是一个新的 Task。如果是同一个 Task，则会指定一个 Activity 切换效果的动画，而如果是一个新的 Task，则会指定一个 Task 切换的动画效果，这些动画效果的实施都是在 WmS 中完成的。每一个动画实际上都仅仅是一个窗口而已，动画的过程可以简单地理解为对同一个窗口进行不同的变化，并在连续的时间将其显示出来，从而形成动画。而在动画的过程中，如果目标窗口也已经创建好，则在动画结束之前不能显示目标窗口，只有当动画结束后才能显示目标窗口；而如果在动画结束后目标窗口还没有被创建，则启动窗口不消失，直到目标窗口被创建好才消失。这个逻辑的具体实现是在 performLayoutAndPlaceSurfaceLocked() 函数中完成的，当然该函数可能仅仅是个包装，内部又会调用 performLayoutAndPlaceSurfaceLockedInner() 函数。为了叙述的方便，本节以后将这两个函数统称为 traversal，因为它们的功能就是根据所有窗口的状态参数，调整其在屏幕上的显示效果，有点类似于 ViewRoot 类中的 performTraversals() 函数。

最后再来看消息处理问题。当要添加一个新窗口时，如果该窗口需要启动动画，那么在动画结束前，应该是哪个窗口获得输入消息呢？在 WmS 中提供了诸如 startAppFreezingScreen()、topAppFreezingScreen()、resumeKeyDispatching()、pauseKeyDispatching()、setEventDispatch(true/false) 这样的接口，用于暂停、恢复消息处理，这些函数将影响在动画过程前后的输入消息处理。那么这些 API 接口具体是被如何使用的？

以上介绍了窗口切换逻辑设计中需要解决的问题，读者在阅读后面小节时需要经常思考这些问题，以便把握该逻辑的“命门”。

### 14.5.2 InputManager 和 WmS 的接口

WmS 的源码虽说超过一万行，但其功能可归纳为两个。第一个是保持窗口的层次关系，以便 SurfaceFlinger 能够据此绘制屏幕；第二个是把窗口信息传递给 InputManager 对象，以便 InputDispatcher 能够把输入消息派发给和屏幕上显示一致的窗口。

关于第一点很容易理解，而关于第二点，有必要特别说明一下。首先做这样一个场景假设，试想，当用户闭着眼睛在手机屏幕上操作时，谁能确定屏幕上显示的内容和用户想象的一致？当然，因为用户随时可能会睁开眼睛，因此 WmS 不能够作假，只能如实显示正确的内容。大家是否能够想象，当闭上眼睛时，用户以为自己在和一个音乐播放器进行交互，并且也飘来了悦耳的音乐，然而此时屏幕上完全可能显示的不是音乐播放器的界面，而是一只愤怒的小鸟的照片。如果让我们实现这样的场景，该如何去做？

以上场景正是 InputManager 和 WmS 之间关系的体现，如要实现以上恶作剧场景，只需要让 WmS 给 InputManager 传递正确地窗口信息，而在 WmS 中却保存错误的窗口信息。这样一来，尽管用户闭着眼睛，但是输入的信息却能够被 InputDispatcher 正确地派发给对应的应用程序，而当用户睁开眼睛时，由于 WmS 中保存了错误的窗口信息，因此 SurfaceFlinger 就会绘制出错误的界面。这再次说明，作为一名程序员，应该明确地认识到，获取用户输入消息的仅仅是触摸屏，而不是触摸屏下面的屏幕，屏幕仅仅是给眼睛看的，所以不要根据眼睛看见的去操作，病毒就是利用这一点。当然，这么说有点耸人听闻，大家实际上并不需要担心这一点，因为目前的病毒似乎还没有能力侵入到 WmS 内部，所以，“所见即所得”的概念在此依然适用。

WmS 中的 InputManager 对象被封装到了一个内部类 InputMonitor 中，从 InputManager 的角度来看，InputDispatcher 有两个特点：

- 可以暂停整个 Dispatcher 的消息派发，该情况下，所有的窗口就都得不到输入消息了。该暂停属性保存在 InputDispatcher 内的全局变量中。
- 也可以暂停对某个窗口消息派发，在该情况下，除被禁止的窗口外，其他窗口的消息派发都正常。该禁止属性保存在每一个窗口的内部。

基于以上两点，InputMonitor 中提供了两个函数，分别是 `updateInputDisptachModeLw()` 和 `updateInputWindowLw()`，前者用于设置 InputDispatcher 整体的消息暂停属性，后者用于设置某个窗口的禁止属性。这些变量的关系如图 14-35 所示。



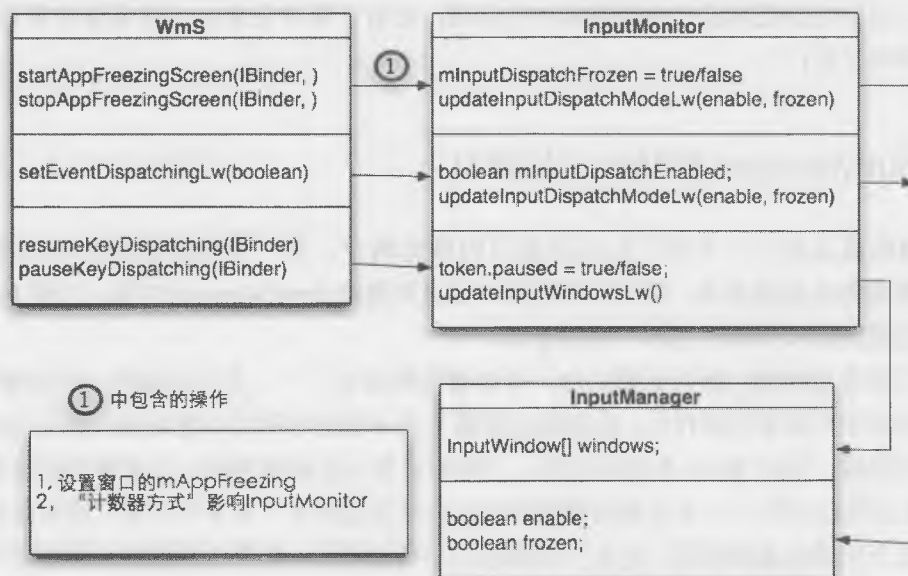


图 14-35 窗口切换过程相关的变量关系

首先，WmS 提供了五个函数，前两个用于 freeze 屏幕，这两个函数内部一般会执行两个重要操作。第一个是将对应窗口的 mAppFreezing 变量置为 true/false，当窗口的 mAppFreezing 变量为 true 时，该窗口暂时不允许被绘制到屏幕。第二个是以计数器的方式影响 InputMonitor。所谓的“计数器方式”是指，当第一次调用时，InputMonitor 将 freeze 消息输入，而在随后的再次调用时，仅仅是将计数器加一，而不再影响 InputMonitor。也就是说，任何一个窗口的 freeze 都会同时 freeze InputMonitor 中的消息输入，而直到最后一个窗口解冻之后，InputMonitor 才会被解冻。这就是为什么这两个函数的参数是一个指定的 IBinder 对象的原因，因为每一个 IBinder 对应的仅仅是一个 ActivityRecord 对象。

第三个函数是 setEventDispatchingLw()，该函数的参数只有一个 boolean 值，也就是说，该调用是影响整个 InputMonitor 的。

最后两个函数是 pause/resumeKeyDispatching()，其作用是暂停对指定 ActivityRecord 的消息派发，参数 IBinder 正是指定的 ActivityRecord 对象。InputManager 中有一组 InputWindow 对象，每个 InputWindow 对象都对应一个具体的窗口，该函数最后影响的就是对该窗口的消息派发，暂停一个窗口的消息派发不会影响其他窗口的消息派发。

以上介绍的是影响 InputDispatcher 对窗口进行派发的状态参数，除此之外，WmS 还需要告知 InputManager 中每一个窗口的具体信息，包括窗口的大小等，这些信息保存在 InputWindow 类中。可以想象，WmS 类和底层的 InputDispatcher 类可能不是一个程序员编写的，并且从功能上讲，两个模块也是可以分离的。InputDispatcher 中需要窗口的一些参数用于对输入消息的派发，WmS 中也需要窗口的参数信息用于 SurfaceFlinger 绘制屏幕，这两者所需要的窗口信息虽然有相同的地方，但也有不同。因



此, 两者定义了一个接口, 这就是 InputWindow 类, 该类定义了 InputDispatcher 所需要的窗口信息, WmS 中每次对窗口进行更新后都需要调用 InputMonitor 类的 updateInputWindowLw() 函数, 把 WindowState 的信息传递给 InputWindow。

updateInputWindowLw() 的源码本身就反映了 InputWindow 所需要的各种信息, 这些信息大多数是易于理解的, 因此下面仅对一些特别变量进行说明。

- inputChannel: 输入消息传递管道, 该管道是在 WmS 中创建的, 对于触摸消息, InputDispatcher 将直接通过该管道把消息传递给客户端窗口, 所以需要该管道信息。
- hasFocus: 指该窗口是否是焦点窗口, 在所有的窗口中, 只能有一个窗口是当前的焦点窗口。
- paused: 表示当前窗口的消息派发是否被暂停, 该状态正是来自于 WmS 提供的 pause/resumeKeyDispatching() 接口。

当 InputDispatcher 收到这些窗口信息后, 将不再根据这些信息计算当前的焦点窗口。当焦点窗口发生变化时, WmS 必须调用 InputMonitor 对象的 setFocusedAppLw() 函数告知 InputDispatcher 新的焦点窗口, 该函数则又是调用 InputManager 对象的 setFocusedApplication() 函数, 如以下代码所示:

```
5298         mTempInputApplication.token = newApp;
5299
5300         mInputManager.setFocusedApplication(mTempInputApplication);
```

此处值得注意的是该调用的参数 mTempInputApplication 变量, 其类型是 InputApplication 类。该类中有三个成员变量, 其最重要的是最后一个变量 Object token, 该变量在使用时被赋值为一个 AppWindowToken 对象, 尚不清楚为什么 InputDispatcher 还需要这个 AppWindowToken 对象。

由于 InputDispatcher 中焦点窗口完全来自于 WmS 中的 setFocusedAppLw(), 因此如果 WmS 给 InputManager 传递一个错误的焦点窗口, 则后果将是屏幕上显示都正确, 但却不能和用户进行正确的交互。另一种情况, 如果 WmS 给 InputDispatcher 传递了正确的焦点窗口, 却没有同步 WmS 内部的窗口信息, 则屏幕上有可能显示的是错误的窗口, 此时用户只能闭着眼睛来操作了。因此, WmS 和 InputManager 之间的接口的使用流程是: WmS 首先保证其内部窗口的正确顺序, 然后再向 InputManager 中传递该窗口信息, 而这也正是 InputManager 类中 updateInputWindowsLw() 的执行过程。该过程就是从 mWindows 变量中读取窗口信息, 并从中提取出 InputWindow 所需要的窗口信息。

### 14.5.3 AmS 与 WmS 的接口

AmS 和 WmS 都是窗口管理系统的核心, 不过 AmS 侧重于对 Activity 的管理, 而 WmS 侧重于对窗口的管理。系统启动后首先是由 AmS 接管主控制权力, 然后 AmS 开始调度并运行 Activity, WmS 作为 AmS 的辅助服务, 接受应用程序的请求创建窗口。当窗口显示后, 用户和窗口的交互控制则交由 WmS 和应用程序本身来完成, 而当应用程序需要启动新的 Activity 时, 则又交给 AmS 去处理, 系统就这样周而复始地运行。该过程在前面相关章节中有多次说明, 尤其在第 10 章中详细介绍了 AmS 的内

部调度过程，本节就来介绍 AmS 和 WmS 之间的接口。

总的来看，WmS 仅仅是接收各种创建窗口的命令，至于这个窗口是一个 Activity 窗口还是一个系统窗口，WmS 并不是特别在意（当然也有一定的区别）。因此，WmS 和 AmS 之间的接口总的来看比较简单。

为了从代码的角度审视 AmS 和 WmS 之间的联系，表 14-2、表 14-3、表 14-4 列出了 WmS 和 AmS 之间的所有调用场景。

Activity 的管理实际上由三个主要类完成，分别是 AmS、ActivityRecord 及 ActivityStack。AmS 的作用不再赘述。AmS 内部用 ActivityRecord 类来表示一个 Activity 对象。ActivityStack 是一个描述 Task 的类，所有和 Task 相关的数据以及控制都由 ActivityStack 来实现。因此，表 14-2、表 14-3、表 14-4 中的第一列表示的是从这三个类中的哪一个中调用 WmS 的接口，第二列表示的是在类中的哪个函数中调用 WmS 的接口，第三列表示调用的是 WmS 中的什么接口。AmS 类内部定义了一个 mWindowManager 变量，三个类中凡是调用 WmS 的接口都是通过该变量。

表 14-2 从 ActivityStack 类中调用 WmS 接口的地方

源自	调用者函数	调用 WmS 中的函数
A.S	completeResumeLocked(ActivityRecord next)	executeAppTransition()
A.S	resumeTopActivityLocked(...)	setAppVisibility() prepareAppTransition() updateOrientationFromAppTokens() executeAppTransition() setAppStartingWindow() addAppToken() validateAppTokens()
A.S	ensureActivitiesVisibleLocked(...)	for 循环中， setAppVisibility(r, true)
A.S	finishActivityLocked( ActivityRecord r,)	prepareAppTransition(endTask? RANSIT_TASK_CLOSE: TRANSIT_ACTIVITY_CLOSE)
A.S	moveTaskToBackLocked(...)	prepareAppTransition() moveAppTokensToBottom(moved) validateAppTokens(mHistory)
A.S	moveTaskToFrontLocked(...)	prepareAppTransition() moveAppTokensToTop(moved) validateAppTokens(mHistory)
A.S	processStoppingActivitiesLocked(...)	for 循环中， setAppVisibility(s, false)

(续表)

源自	调用者函数	调用 WmS 中的函数
A.S	realStartActivityLocked(...)	setAppVisibility(r, true) updateOrientationFromAppTokens()
A.S	removeActivityFromHistoryLocked( ActivityRecord r)	removeAppToken(r)
A.S	resetTaskIfNeededLocked(...)	setAppGroupId(...) moveAppToken(dstPos, p) validateAppTokens(mHistory)
A.S	startActivityLocked(ActivityRecord,)	addAppToken() validateAppTokens() prepareAppTransition() setAppStartingWindow() validateAppTokens()
A.S	stopActivityLocked(ActivityRecord r)	setAppVisibility(r, false)

表 14-3 从 AmS 类中调用 WmS 接口的地方

源自	调用者函数	调用 WmS 中的函数
AmS	closeSystemDialogs(String reason)	closeSystemDialogs(reason)
AmS	enableScreenAfterBoot()	enableScreenAfterBoot()
AmS	getRequestedOrientation(IBinder token)	getAppOrientation(r)
AmS	goingToSleep()	setEventDispatching(false)
AmS	handleAppDiedLocked(...)	while 循环中, removeAppToken(r)
AmS	isNextTransitionForward()	getPendingAppTransition()
AmS	overridePendingTransition(...)	overridePendingAppTransition(...)
AmS	setFocusedActivityLocked( ActivityRecord r)	setFocusedApp(r, true)
AmS	setRequestedOrientation(IBinder token, int requestedOrientation)	setAppOrientation(r,) updateOrientationFromAppTokens()
AmS	shutdown(int timeout)	setEventDispatching(false)
AmS	updateConfiguration(Configuration values)	computeNewConfiguration()
AmS	updateConfigurationLocked(...)	setNewConfiguration(mConfiguration)
AmS	wakingUp()	setEventDispatching(true)

表 14-4 从 ActivityRecord 类中调用 WmS 接口的地方

源自	调用者函数	调用 WmS 中的函数
A.R	pauseKeyDispatchingLocked()	pauseKeyDispatching(this)
A.R	resumeKeyDispatchingLocked()	resumeKeyDispatching(this)
A.R	startFreezingScreenLocked()	startAppFreezingScreen(this,)
A.R	stopFreezingScreenLocked( boolean force)	stopAppFreezingScreen(this, force)

以上三个表格中包含的对 WmS 的调用表面上看起来似乎有几十个，但它们之间却有一种内在的联系，并可被抽象归类为以下几类调用。

首先是在 ActivityStack 中的调用，主要包含：

- add/removeAppToken(), 其作用是向 WmS 中添加、删除一个 AppWindowToken 对象。因为每个 ActivityRecord 对象在 WmS 中都对应一个 AppWindowToken 对象，当创建一个该对象时，对应的窗口一般还没有被创建。
- setAppVisibility(true/false), 其作用是告诉 WmS 指定的窗口是否可以被显示。在 WmS 中使用 WindowState 类表示一个窗口，其内部有两个定义。一个为变量 mReadyToShow, 该变量表示当前窗口内的 Surface 是否已经准备好，并且客户端是否已经给该 Surface 上绘制了内容。如果是，则说明该窗口已经“准备好了被显示”，这就是 Ready to show 的意思，可认为这是窗口能够被显示的内因。另一个定义为函数 isReadyForDisplay(), 该函数的返回值表示 AmS 是否允许该窗口被显示出来，AmS 调用 setAppVisibility()的作用就是是否允许该窗口被显示出来，该条件可认为是窗口能够被显示的外因。在 WmS 中的 performLayoutAndPlaceSurface()执行中，只有当窗口同时满足被显示的内因和外因时，才能真正被绘制到屏幕上。
- setAppStartingWindow()、prepareAppTransition()、executeAppTransition(), 这三个函数用于实现窗口切换的动画。setAppStartingWindow()设置启动窗口的图标和标题，因为当目标窗口还没有被创建时，WmS 会显示一个仅包含图标和标题栏的启动窗口，并当目标窗口被创建后该启动窗口才消失。prepareAppTransition()告知 WmS 窗口切换是哪种类型，比如是不同 Task 之间的切换呢，还是同一个 Task 中的不同 Activity 的切换：executeAppTransition()的作用则是告知 WmS 开始执行切换。

接下来是 AmS 中的调用，主要包含：

- setEventDispatching(true/false), 其作用是告知 WmS 停止或者开始输入消息的处理。比如在 AmS 的 shutDown()函数中，调用 setEventDispatch(false), 即当 AmS 处于关闭状态时，停止输入消息的派发，而当 AmS 执行 wakeUp()时，则又恢复消息的派发。
- setFocusApp(), 其作用是告知 WmS 当前哪个应用窗口应该获得焦点。比如当用户长按“Home”键并选择之前运行过的 Activity 时，焦点窗口应该变为该 Activity 对应的窗口。由于 AmS 中并

未保存窗口信息，因此只能告诉 WmS 对应的 ActivityRecord 获得了焦点，然后在 WmS 内部，该 ActivityRecord 对应了一个 AppWindowToken 对象，该对象内部包含了所有与该 ActivityRecord 相关的 WindowState 对象，于是 WmS 根据该 Focus 的信息重新调整这些窗口的层值。

最后是 ActivityRecord 中的调用，主要包含：

- pause/resume KeyDispatching，其作用是告知 WmS 暂停对指定 Activity 窗口的按键消息派发。
- start/stop FreezeScreen，其作用是冻结或解冻当前屏幕，当屏幕处于冻结状态时，SurfaceFlinger 将不被调用，从而界面就不会被重绘。

除了从 AmS 中调用 WmS 外，WmS 也需要调用 AmS，不过只有四处对应 AmS 中的三个函数调用，如表 14-5 所示，其中 P.W.P 代表 PhoneWinowManager 类。

表 14-5 窗口服务调用 Activity 服务的地方

源自	调用者函数	调用 AmS 中的函数
P.W.M	goHome()	stopAppSwitch()
P.W.M	launchHome()	stopAppSwitch()
WmS	reclaimSomeSurfaceMemoryLocked()	killPids()
WmS	sendNewConfiguration()	updateConfiguration()

WmS 中调用 AmS 的接口如下：

- stopAppSwitch()，告知 AmS 暂停应用窗口的切换。
- killPids()，当系统内存低时，AmS 会要求客户端释放内存，而可能会释放 Surface 对应的内存，而这是由 WmS 具体完成的。当 WmS 释放了 Surface 内存后，该 Surface 对应的窗口就无效了，则该窗口对应的 Activity 也就无效了，则 Activity 所在的进程也就无效了，因此需要调用 killPids() 杀掉指定的进程。
- updateConfiguration()，其作用是告知 AmS 重新获取应用窗口的 Configuration 值，并将该 Configuration 广播给相应的 Listener。因为在 WmS 中保存着系统当前的 Orientation（屏幕当前的方向），当一个 Activity 启动后，其窗口首先总是按照垂直方向被处理，而当启动后发现与当前的方向不一致，则会通知 AmS，AmS 再将这个转变通知给相应的 Activity。

以上从语义的角度解释了 AmS 和 WmS 的主要接口，尤其是进行窗口切换时的主要接口，至于这些函数在源码中具体被如何调用，将在后面小节的具体场景中详细介绍。

#### 14.5.4 从 A 到 B 的切换

本节就来介绍最常见的一种窗口切换场景。比如在 Home 界面上，当用户点击某个应用图标时，AmS 会启动该应用，从而使得该应用的窗口显示到屏幕上。我们将这个过程的窗口切换称为 A 到 B 的切换，其逻辑过程如图 14-36 所示。

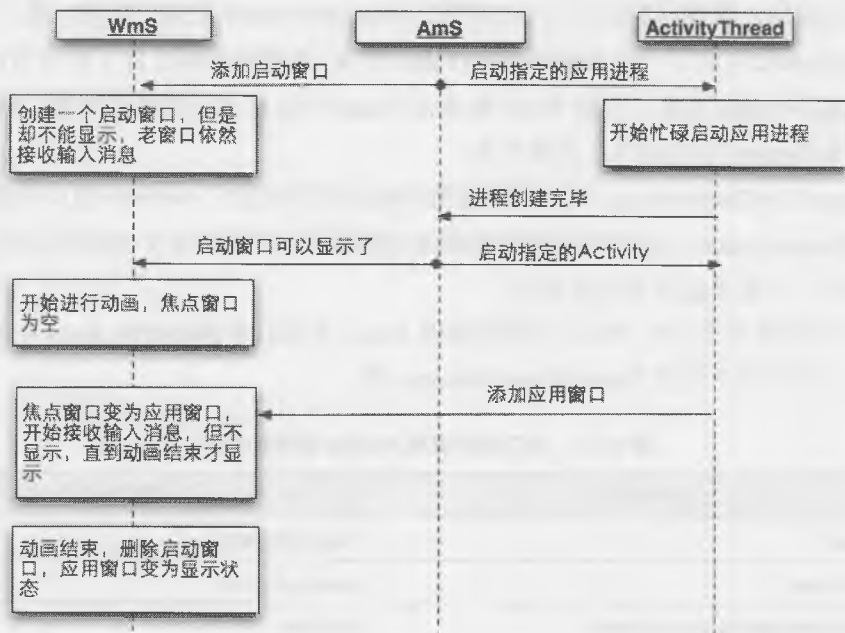


图 14-36 Activity 的切换过程——由 A 到 B

下面从代码的角度解释以上执行过程。

首先, 当要启动 B 时, AmS 会调用 WmS 的 `addAppToken()` 添加一个 token, 该 token 对应的是新的 Activity。然后再调用 WmS 的 `setAppStartingWindow()` 告诉 WmS 启动窗口的标题和图标, 以便 WmS 能根据这两个信息创建一个启动窗口。WmS 接收到这个命令后就开始去创建启动窗口了, 创建的具体过程就是标准的添加窗口的过程。

与此同时, AmS 还去启动 B 对应的进程, 如果进程已经存在, 则运行一个 `ActivityThread` 实例。前面章节曾经讲过, 每个 Activity 都是从 `ActivityThread` 开始执行的, `ActivityThread` 类是客户端程序的主类, Activity 仅仅是一个回调而已。

在接下来的一段时间里, AmS 处于空闲状态; WmS 内部则开始创建启动窗口, 并可能已经创建完毕了启动窗口, 但暂时不能显示该启动窗口; 而 `ActivityThread` 内部也忙碌地启动进程并使 `ActivityThread` 就绪。

当 `ActivityThread` 就绪后, 就会通过 IPC 调用 AmS 的 `attachApplication()`, 通知 AmS 自己已经就绪, 可以运行任何指定的 Activity 了。当 AmS 收到这个通知后, 一方面会调用 WmS 的 `setAppVisibility()` 使其开始显示启动窗口, 并调用 WmS 中的 `setFocusedApp()` 将新的 `AppWindowToken` 设为焦点窗口, 然而此时由于真正的窗口还没有就绪, 所以焦点窗口被调整为 `Null`。另一方面则调用 `ActivityThread` 中内联类 `ApplicationThread` 的 `scheduleLaunchActivity()`, 请求其开始运行指定的 Activity, 这最终会调用执行到 Activity 类的 `onCreate()` 函数中。

在接下来的一段时间里，在 WmS 中，由于真正的 Activity 窗口还没有被创建，因此当前的焦点窗口被调整为 null，并且开始了启动动画。另一方面，在 ActivityThread 类中则开始运行 Activity 的 onCreate()，该函数最终会调用到 setContentView()，这会间接地创建一个真正的 Activity 窗口，关于该过程可参照前面相关章节。

当 ActivityThread 内部执行到创建真正的 Activity 窗口时，会调用到 WmS 中的 addWindow() 函数，在该函数中，当添加完新窗口后，就会把焦点调整到新窗口中。此时，根据启动动画是否执行完毕，会有两种情况：如果动画还没有执行完毕，则暂时不显示 Activity 窗口，不过尽管看不到该窗口，但是该窗口却能接收输入消息，换句话说该窗口只是隐形；另一种情况，如果动画已经执行结束，则新窗口会立即显示到屏幕上并接收输入消息。

该过程反映了一个问题，即焦点窗口切换有两种途径。一种是由 AmS 发起，调用 WmS 的 setFocusedApp()，设置当前新的焦点；另一种是在添加窗口 addWindow() 时，在一般情况下当添加的不是启动窗口时，新窗口都会成为新的焦点窗口。

如果上面的动画还没有结束，则继续执行启动动画，直到动画结束。当动画结束时，WmS 内部会发送一个 Handler 消息，名称为 FINISHED\_STARTING，该消息具体是在 performShowLocked() 函数中发出的，在该消息的处理代码中，会调用 removeWindow() 删除启动窗口。

由于以上过程是在两个独立进程、三个独立线程中异步完成的，所以调用过程初看起来比较复杂，然而只要从语义上理解其步骤，代码上则比较容易把握。同时，对于 Activity 的启动过程及其和 AmS 的交互过程要经常回顾前面的章节。

为了使以上过程的逻辑能更生动一些，下面用一幅漫画来描述该过程，如图 14-37 所示。



图 14-37 漫画示意由 A 到 B 的过程



该漫画中，小鸟代表 A，小猪代表 B，“玩儿”就是指和用户交互。

### 14.5.5 从 B 回到 A 的过程

从 B 到 A 是指，当前正在运行 B，然后用户按“Back”键回到 A 的过程，其流程如图 14-38 所示。

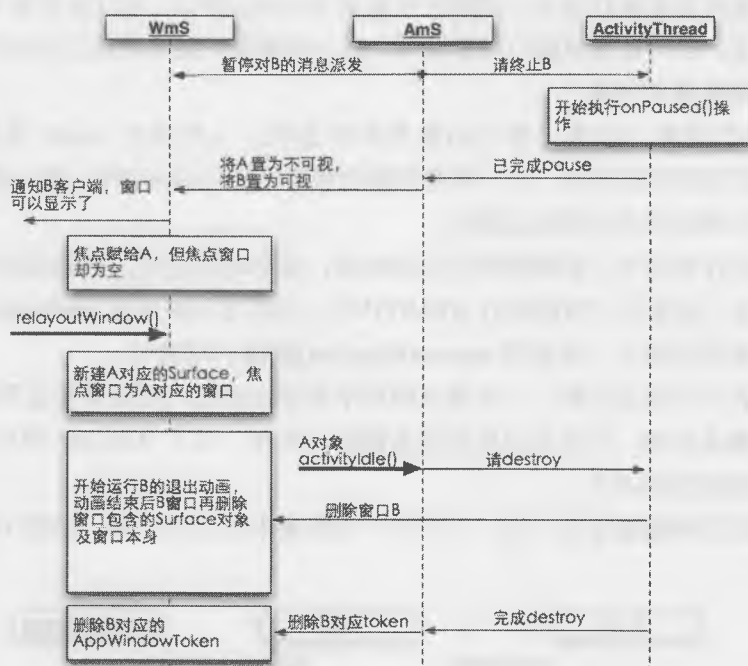


图 14-38 Activity 的切换过程——由 B 返回到 A

当按“Back”键后，会调用到 AmS 的 `finishActivityLocked()`，关于这个过程细节参见第 10 章。在 AmS 的 `finishActivityLocked()` 中，会命令 B 客户端终止当前正在运行的 Activity，B 接到这个命令后就开始执行 Activity 中的 `onPaused()`。另一方面，AmS 会通知 WmS 暂停对 B 窗口的消息派发，这是通过调用 WmS 的 `pauseKeyDispatching()` 函数完成的。

接下来，对用户来讲，此时当前屏幕依然是 B 窗口，然而却不能接收消息输入，不过仅仅是 B 窗口，状态栏窗口则依然可以接收消息输入。而在 B 所在的 ActivityThread 中，则正忙于暂停 B，当然，这个过程一般很短暂，最长不超过 5 秒。至于 WmS 和 AmS，则当前都处于空闲状态。

直到 B 完成了暂停的过程，它会通过 IPC 调用 AmS 的 `completePauseLocked()`，AmS 会通知 WmS 将 B 置为不可视，将 A 置为可视。而在 WmS 中将 A 置为可视时，会同时通知 A 的客户端说：“窗口可以显示了”，这是通过 IPC 调用客户端中 ViewRoot.W 类中的 `dispatchAppVisibility()` 完成的。当客户端

收到该调用后，会在 ViewRoot 内部执行一次 performTraversals()，而该函数内部则又会回调到 WmS 中的 relayoutWindow()。

接上面，在 WmS 内部，一方面通过 IPC 调用客户端的 performTraversals()，另一方面则将焦点赋值给 A，然而由于此时 A 对应的窗口内部的 mSurface 为空，所以真正的焦点窗口为空。为什么 A 中的 mSurface 会为空呢？在上一节介绍从 A 启动 B 时讲到，当 B 启动后，WmS 会调用 A 客户端的 dispatchAppVisibility(false)告知 A：“嗨，伙计，你的窗口可以被隐藏了！”A 接收到这个消息后会调用 WmS 的 relayoutWindow()，其中参数 viewVisibility 的值为 false。因此，在 relayoutWindow()函数中会执行 win.destroySurfaceLocked()。从而销毁了 A 对应的 Surface，这样做的作用有两个，一个是释放了 Surface 对应的内存，另一个是减少了 SurfaceFlinger 内部的 Surface 对象，可以加速对屏幕的绘制。

当 WmS 收到 A 客户端的 relayoutWindow()调用时，就开始重新创建 A 对应的 Surface，并在 Surface 创建完毕后将焦点窗口切换到该窗口，然后开始执行退出动画。从现在开始，用户可能还不能看见 A 窗口的全部区域，但是当用户输入消息时则能够被 A 窗口处理。

在动画的过程中，当然也可能是在动画结束后，AmS 会调用 WmS 的 removeWindowLocked()函数删除 B 窗口。在该函数中，如果当前正在运行退出动画，则将目标窗口的 mRemoveOnExit 变量置为 true，从而使当动画结束后再删除指定的窗口，如下代码所示：

```
2050 // The exit animation is running... wait for it!
2051 //Slog.i(TAG, "*** Running exit animation...");
2052 win.mExiting = true;
2053 win.mRemoveOnExit = true;
```

那么，AmS 中为什么会调用到 WmS 的 removeWindowLocked()呢？这是发生在当 A 内部准备好后，会通过 IPC 调用到 AmS 的 activityIdle()，于是 AmS 就会销毁那些已经 finish 的 Activity，其中就包括 B。这是通过 IPC 调用 B 进程 ActivityThread 中的 handleDestroyActivity()完成的，该函数内部又会调用 removeWindow()等相关的函数，这实际上就调用到了 WmS 中的 removeWindow()。

当退出动画结束后，会执行 WindowState 类内部的 finishExit()，这就会把被删除的窗口的 Surface 暂时保存到 mDestroySurface 列表中，并把被删除窗口本身放在 mPendingRemove 列表中，从而使得在 WmS 中的下一次 traversal 中删除窗口对应的 Surface 对象及 WindowState 对象本身。

最后，当 B 完成 destroy 时，会通知 AmS 说：“我已经完成了 destroy”，这是通过 IPC 回调 AmS 的 activityDestroyed()实现的。在 AmS 的 activityDestroyed()函数中，则调用了 WmS 的 removeAppToken，从而删除 B 在 WmS 中对应的 AppWindowToken 对象。有读者会问：“删除 WindowState 对象和删除 AppWindowToken 对象有先后顺序吗？”答案是肯定的，因为在 ActivityThread 中，如果还没有删除窗口对象则意味着还没有完成 destroy 的操作。

至此，就完成了从 B 到 A 的整个切换。

#### 14.5.6 A 中长按“Home”键切换到 B

长按“Home”键时，界面会弹出一个最近的任务列表，假设在长按之前正在运行 A，然后在弹出

的任务列表中又选择 B，那么这个切换过程如何呢？该过程与从 A 中启动 B 有什么区别呢？

且不谈具体的切换过程如何，但从结果的角度大家都可以想象，主要的区别至少包含两点：

- A 对应的 WindowState 对象在这种情况下不会被删除，因为 A 窗口仅仅是隐藏。
- A 窗口对应的 AppWindowToken 对象也不会被删除，因为 A 不会 destroy。

下面就来具体分析其执行过程，如图 14-39 所示。

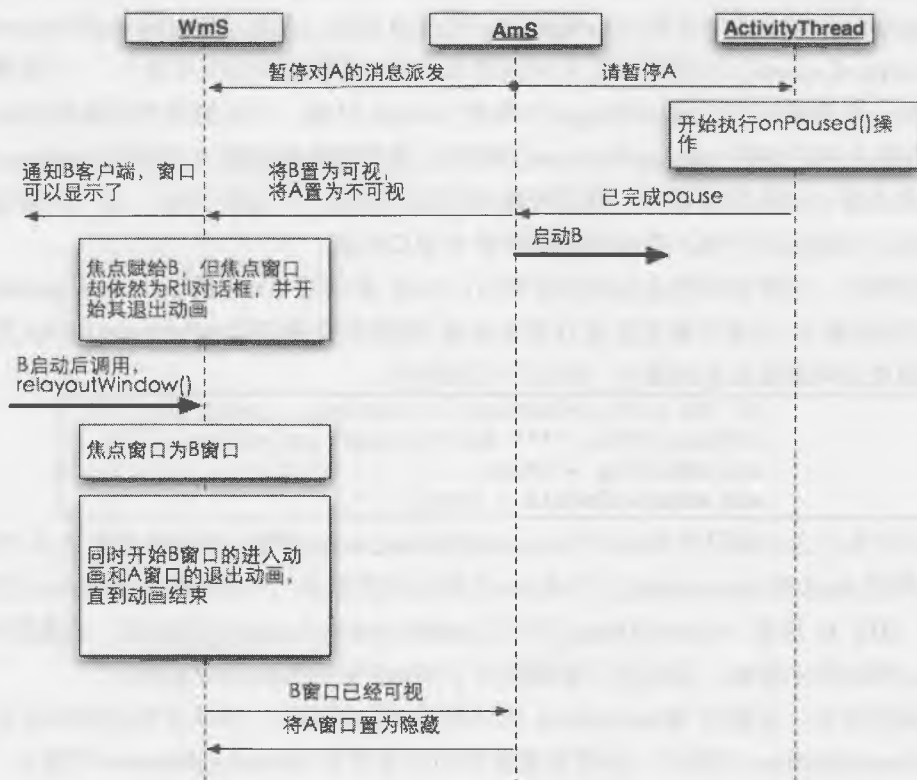


图 14-39 从任务列表中启动 Activity 的过程

当前正在运行 A，长按“Home”键后会弹出最近任务列表对话框（Recent Task List），以后简称 Rtl。当用户点击该对话框中的任务图标 B 时，就会调用 `startActivity()` 启动 B，详情参见 `RecentApplicationDialog` 类内部代码。

当 AmS 接收到该启动请求后，就开始通知当前 A 所在的 ActivityThread 暂停 A，于是 ActivityThread 开始执行暂停相关的操作，包括回调 Activity 的 `onPause()` 函数。另一方面，AmS 会通知 WmS 暂停对 A 的消息派发，于是，用户尽管能看到 A 窗口，但却不能和该窗口继续进行交互，事实上，在显示 Rtl 窗口时 A 窗口已经暂停消息派发了。

接下来，WmS 和 AmS 皆处于空闲状态了，只等 ActivityThread 中完成对 A 的暂停操作，运气不好的话可能要等几秒钟，不过最长不会超过 5 秒。

当 ActivityThread 完成暂停后, 会通过 IPC 回调 AmS 的 completePauseLocked() 通知 AmS 说: “我已经完成暂停操作了。” AmS 接到这个通知后就开始忙活起来了, 这包括通知 WmS 将 A 置为不可视状态, 将 B 置为可视状态, 这是通过调用 WmS 的 setAppVisibility() 完成的。同时, AmS 通过 IPC 调用 B 所在 ActivityThread 中的 scheduleResumeActivity() 启动 B。

在接下来的时间里, 在 WmS 中开始执行 Rtl 窗口的退出动画, 动画方式是阿尔法渐变, 因此用户会看到 Rtl 窗口逐渐消失。如果在该动画消失前, B 窗口所在的客户端还没有完成对 B 窗口的绘制, 那么用户将看到 A 窗口依然存在于屏幕之上, 但在一般情况下, 在 Rtl 动画消失前 B 客户端都会完成对窗口的绘制。

当 B 客户端完成对 B 的启动后, 会经过多层调用, 并最终调用到 ViewRoot 的 relayoutWindow() 函数。它又会通过 IPC 调用 WmS 的 relayoutWindow(), 该函数中将为 B 窗口创建所需的 Surface, 并且改变当前的焦点窗口。此时焦点窗口将被修改为 B 窗口, 从此刻起, 尽管用户还没有看见 B 窗口的“庐山真面目”, 但是却可以和 B 窗口进行交互了, 包括点击屏幕。但要记住, 此时屏幕的点击坐标并不是实际看到的 B 窗口坐标, 而是 B 窗口动画结束后的坐标。

B 窗口的进入动画和 A 窗口的退出动画是同时运行的, 当该动画结束后, 用户就可以看到完整的 B 窗口, 此时 WmS 会通知 AmS 窗口 B 已经完全显示出来的, 这是通过调用 ActivityRecord 类的 windowVisible() 或者 windowGone() 完成的, 如下代码所示:

```

7991         if (DEBUG_VISIBILITY) Slog.v(
7992             TAG, "Reporting visible in " +
7993             + " visible-" + nowVisible
7994             + " gone-" + nowGone);
7995         if (nowVisible) {
7996             wtoken.appToken.windowsVisible();
7997         } else {
7998             wtoken.appToken.windowsGone();
7999         }

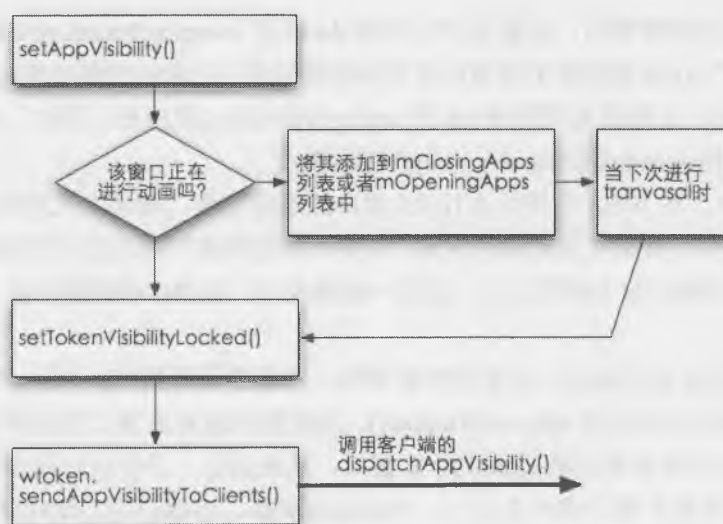
```

当 AmS 收到该通知后, 会再调用 WmS 的 setAppVisibility() 将 A 窗口置为隐藏状态。当 A 窗口被隐藏后, A 窗口对应的 Surface 会被销毁。这个过程是普遍的, 任何一个应用窗口的状态被置为隐藏后, 都会导致其 Surface 被销毁。

#### 14.5.7 setAppVisibility() 与销毁 Surface

在以上三个过程中, AmS 经常调用 WmS 的 setAppVisibility() 设置指定应用窗口的可视状态。在设置的过程中, 如果当前窗口正在进行动画, 则会延迟改变其状态, 直到动画结束。另外, 如果窗口被设为隐藏状态又会导致 Surface 的消息, 这些过程听起来有些含混, 因此本节就来理清其中的逻辑关系。

总的来讲, 先不要考虑销毁 Surface 的事情, 单独 setAppVisibility() 的调用过程如图 14-40 所示。

图 14-40 `setAppVisibility()`的调用过程

`setAppVisibility()`仅在 AmS 端被调用，WmS 内部是不会调用该函数的，其作用是 AmS 告诉 WmS 指定应用窗口的可视状态应该如何变化。在 WmS 中，会先判断该窗口是否正在进行动画，确切地讲应该是该动画是否结束，因为如果窗口还没有被设置过动画则会首先给窗口设置一个动画，为什么这样呢？这仅仅是为了界面的效果，因为设计者希望用户能看到窗口进入及退出的动画，从而让用户感觉到操作的连贯性。

对于进入的窗口，会被添加到 `mOpeningApps` 列表中，而退出的窗口则会被添加到 `mClosingApps` 列表中，然后直到下次 traversal 时，才调用 `setTokenVisibilityLocked()`真正改变该窗口的状态。

而在 `setTokenVisibilityLocked()` 中，最重要的事情莫过于调用 `AppWindowToken` 的 `sendAppVisibilityToClients()`，该函数内部会调用到客户端窗口 `ViewRoot.W` 类的 `dispatchAppVisibility()`。为什么说这个很重要呢？因为这在 `ViewRoot` 内部会调用其内部的 `sWindowSession.relayoutWindow()`，这样又会回调到 WmS 中，从而使得要么是创建窗口对应的 `Surface`，要么是销毁窗口对应的 `Surface`，这也正是 AmS 调用 WmS 的 `setAppVisibility()`的最终目的。

下面再来看销毁 `Surface` 的过程。首先需要明确一点，在两种情况下会销毁窗口对应的 `Surface`。一种是上面的 `relayoutWindow()`，其根源是 `setAppVisibility()`；另一种情况是当要删除窗口时，这个很容易理解，因为“皮之不存、毛将焉附”，窗口都不存在了，其内部的 `Surface` 自然应该被销毁。这两种情况的具体过程如图 14-41 所示。

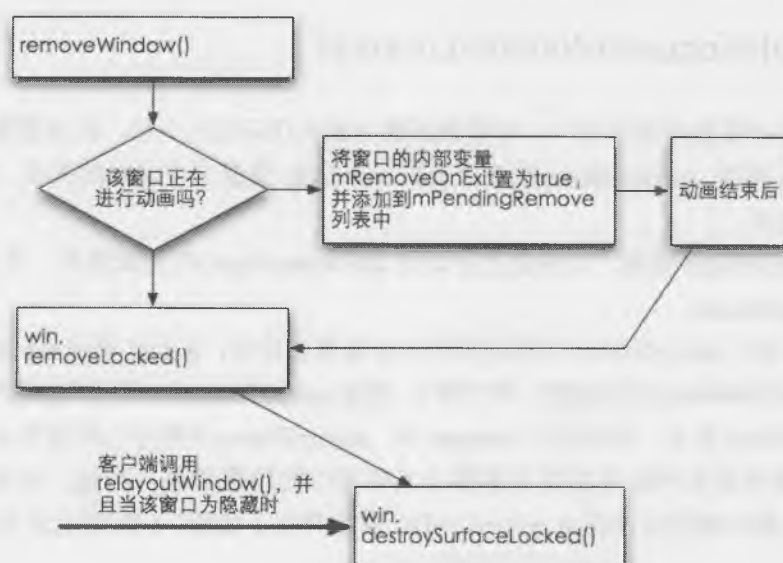


图 14-41 两种引起销毁 Surface 对象的情况

对于第一种情况，当调用 `removeWindow()` 时，判断该窗口是否正在进行动画。如果是，则将该窗口对象的 `mRemoveOnExit` 置为 `true`，意思是说“当退出动画结束后就删除”，然后将该窗口添加到 `mPendingRemove` 列表中。

由于动画的过程实际上是在 `traversal` 时执行的，因此，每次在 `traversal` 时会判断窗口的动画是否结束，如果结束就立即调用窗口的 `removeLocked()`，该函数内部则会调用到 `destroySurfaceLocked()`，从而销毁 `Surface`。关于动画在 `traversal` 内部的具体过程将在下面小节中介绍。

对于第二种情况，无论什么原因导致客户端调用 `WmS` 的 `relayoutWindow()`，如果此时客户端传递过来的窗口状态为不可视，`WmS` 就会销毁该 `Surface`，如以下代码所示：

```

2460         } else {
2461             if (mInputMethodWindow == win) {
2462                 mInputMethodWindow = null;
2463             }
2464             win.destroySurfaceLocked();
  
```

那么，什么原因会导致执行 `relayoutWindow()` 呢？一种是当客户端的 `Activity` 启动，发现窗口的 `Surface` 为空时，会请求 `WmS` 创建一个新的 `Surface`，于是会调用 `relayoutWindow()`；另一种情况是当 `AmS` 调用 `WmS` 的 `setAppVisibility()`，`WmS` 内部又会调用到客户端 `ViewRoot.W` 类的 `dispatchAppVisibility()`，该函数执行时，如果发现窗口状态改变，则又会回过头来调用 `WmS` 的 `relayoutWindow()`。

这就是销毁 `Surface` 的过程。

`setAppVisibility()` 和销毁 `Surface` 在前面的三种切换中都有使用，读者可以对照理解。

## 14.5.8 computeFocusedWindowLocked()

该函数的作用是计算当前焦点窗口，尽管该函数内部的代码仅几十行，但该逻辑却并不容易理解，关键在于 `mWindows` 列表、`mAppTokens` 列表及 `mFocusedApp` 变量三者之间的关系，因此本节就来揭示这三者的内在逻辑关系。

首先来看 `mFocusedApp` 变量，该变量是在调用 `setFocusedApp()` 时被赋值的，并且立即生效，赋值完成后才开始进行 `compute`。

在执行 `compute` 前，`mAppTokens` 内部的顺序已经是调整后的，这是在调用 `moveAppTokensToTop()` 或者 `moveAppTokensToBottom()` 中完成的。举个例子，假设 `mAppTokens` 内部原来的顺序是  $A \rightarrow B \rightarrow C \rightarrow D$ ，而新的 `AppWindowToken` 是 `B`，则在进行 `compute` 时，`mAppTokens` 的顺序已经变为  $A \rightarrow C \rightarrow D \rightarrow B$ 。

`mWindows` 列表中的顺序始终反映了屏幕上当前窗口的层叠关系，因此，在进行 `compute` 时，`mWindows` 列表中的窗口顺序依然是  $A \rightarrow B \rightarrow C \rightarrow D$ 。这就产生了如图 14-42 所示的关系。

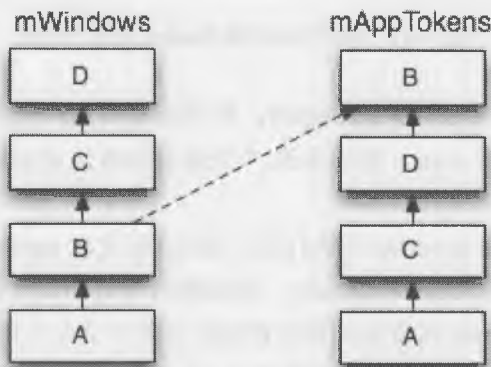


图 14-42 `mWindows` 和 `mAppTokens` 中窗口的顺序

因此，`compute` 内部的逻辑就容易理解了。在执行 `compute` 时，`mFocusedApp` 是 `B`，而遍历 `mWindows` 是从 `D` 开始的，因此，在这种情况下，会计算出焦点窗口为空，因为程序执行时会执行到 `nextApp == mFocusedApp` 的条件，如以下代码所示：

```

9918         if (nextApp == mFocusedApp) {
9919             // Whoops, we are below the focused app... no f
9920             // for you!
9921             if (localLOGV || DEBUG_FOCUS) Slog.v(
9922                 TAG, "Reached focused app: " + mFocusedApp);
9923             return null;

```

那么，在什么时候才会找到真正的焦点窗口呢？答案就是当 `mWindows` 中的窗口顺序被调整到和 `mAppTokens` 中的顺序一致时。那么什么时候会调整为一致呢？这就是当 `B` 对应的窗口进行



relayoutWindow()时, WmS 内部会重新调整 mWindows 内部的窗口顺序。

在 compute 内部, 并不是 mWindows 列表中最上面的窗口就是焦点窗口, 满足焦点窗口的条件必须是 canReceiveKeys()返回为 true, 如以下代码所示:

```

9940      // Dispatch to this window if it is wants key events.
9941      if (win.canReceiveKeys()) {
9942          if (DEBUG_FOCUS) Slog.v(
9943              TAG, "Found focus @ " + i + " - " + win);
9944          result = win;
9945          break;

```

canReceiveKeys()函数的语义是指“能够接收按键消息”, 能够接收按键消息的窗口需满足以下两个主要条件。

- 该窗口是可视的。
- 该窗口不能是 NOT\_FOCUSABLE。比如状态栏窗口就不能是焦点窗口。因为它的窗口 flag 是包含 NOT\_FOCUSABLE 标识的, 这就是为什么尽管 mWindows 最上面的是状态栏窗口, 但状态栏却不是焦点窗口的原因。

系统中往往会包含两个特殊窗口, 一个是启动窗口, 另一个是 Toast 窗口。在以往的开发经验中, 读者都知道这两个窗口不能获得焦点, 其内在的原因就是它们的 flag 中包含 NOT\_FOCUSABLE 标识, 除了该标识之外, 它们还包含 NOT\_TOUCHABLE 标识, 这些标识都是在创建时指定的, 因为这两个窗口也不能接收触摸消息。讲到这里, 请读者思考, 前面章节中曾经讲过, 按键消息和触摸消息的一个区别在于, 对于按键消息, InputDispatcher 会先回调 WmS 中的预处理, 然后才把该消息传递给窗口对象, 而对于触摸消息则是直接传递给窗口对象, 那么 InputDispatcher 如何得知该窗口是否是 NOT\_TOUCHABLE 呢? 答案就在 WmS.InputMonitor 类中的 updateInputWindowLw()的关键赋值代码, 如以下代码所示:

```

5173      final int flags = child.mAttrs.flags;
5174      final int type = child.mAttrs.type;
5175
5176      final boolean hasFocus = (child == mInp
5177      final boolean isVisible = child.isVisib
5178      final boolean hasWallpaper = (child ==
5179          && (type != WindowManager.Layou
5180
5181      // Add a window to our list of input wi
5182      final InputWindow inputWindow = mTempIn
5183      inputWindow.inputChannel = child.mInput
5184      inputWindow.name = child.toString();
5185      inputWindow.layoutParamsFlags = flags;

```

代码中 flags 中就包含了该窗口是否为 NOT\_TOUCHABLE 的标识, 然后将该值赋给了 inputWindow 对象的 layoutParamsFlags 变量, InputDispatcher 有了这个信息后, 就能确知哪些窗口是 NOT\_TOUCHABLE, 从而那些窗口就不能获得触摸消息。从实际的效果来看, InputDispatcher 内部应

该是忽略了 NOT\_TOUCHABLE 窗口，因此，当用户触摸到这些窗口上时，InputDispatcher 会认为是触摸到了这些窗口下面的能够获取输入消息的窗口。

## 14.6 performLayoutAndPlaceSurfacesLockedInner() 的执行过程

该函数是 WmS 中最核心的函数，也是最复杂的函数，在介绍该函数的内部过程之前先来讲一个有趣的事情。

WmS 源码的作者是 Dianne Hackborn，Dianne 翻译成中文就是“黛安娜”，读者可以在 Android 的官方论坛中看到 Dianne 的身影，她经常会对 Framework 相关的问题进行解答，其中一个问题就是和该函数相关的，原文如下：

```
I was blown away to find yesterday that the Gingerbread version of
WindowManagerService.performLayoutAndPlaceSurfacesLockedInner comes to
1233 lines. That's, well, unimaginably far from "one screen".
```

```
Is it just too risky to touch? (Obviously, it's a lot harder to write
unit tests for GUI code than for engine code.) Or is it simply that no
one has had the time to refactor it?
```

```
Actually, both these are proxies for my real question: If I broke it
into a series of calls to smaller private methods, would this be a
plausible patch candidate, or is it a 'protected species'?
```

下面来看“黛安娜”的回复，原文如下：

```
Hi Jon,
```

```
What, you have a problem with my 1000 line function?? :)
```

```
Actually there are much bigger aspects to WindowManagerService that I would like to refactor to
ease maintenance -- the first being turning this nearly 12,000 line file with multiple inner classes
into a bunch of separate classes in their own package.
```

```
I have been wanting to do this for over a year. The problem is finding a good time to do it that
won't cause huge merge headaches -- we often tend to have multiple development branches running
in parallel, so if you do this at the wrong time you suddenly have painful merges between them.
I think we have an opportunity coming up for this post-HC, but we'll see.
```

```
What that means for AOSP contributions, though, is that refactoring code like this is fairly
problematic to accept as a contribution, because of the merge pain that will arise from that. Even
changes that do things like just "fix" a bunch of whitespace are difficult to deal with.
```

回答中最有趣的就是第一句话，“怎么了，你对本姑娘的 1000 行代码函数有问题??”注意该句后面的那个笑脸，估计黛安娜在回复这个问题时，内心早已释然，因为这个问题她早已意识到，但截止到目前只是没有找到合适的时间来重构这段代码而已。从这个有趣的问答中大家也能感觉到，一个包含 1200 行代码的函数对读者是多么大的一个困惑，包括代码作者，也都觉得这段代码不易维护。笔者从

未写过一个函数，其内部代码超过 1000 行。

下面具体来看该函数的内部逻辑。

### 14.6.1 总体过程

要理解该函数的内部过程，首先应该了解该函数的总体功能及内部总体过程，从而不至于陷入内部的一些小逻辑中。

WmS 的设计目标可以用一句话来概括，那就是把窗口绘制到屏幕上。为了实现这个目标，WmS 内部定义了很多数据变量，比如 mWindows、mTokenList 等，这些变量用来保存窗口相关的数据。同时 WmS 提供了一些 API 接口，客户端和 AmS 端都可以调用这些 API 接口，比如 addWindow()、removeWindow()、setAppVisibility()等。外界调用这些 API 后会修改窗口的参数，每次修改完这些参数后，WmS 内部都会调用 traversal 函数，该函数会根据窗口的参数将窗口重新绘制到屏幕上。动画、模糊背景、黯淡背景、显示、隐藏等这些效果都属于“绘制”的范畴。

当然，WmS 内部除了绘制外，还会把窗口的一些参数传递给 InputDispatcher 模块，但相比显示窗口而言这是微不足道的，因此如果忽略这一点，假设用户是闭着眼睛操作的话，则 WmS 基本上就没什么作用了，可完全不用。当然，这只是理论上的情况，实际代码中 WmS 会和 Policy 一起进行配合，在对窗口添加、删除、计算大小的过程中进行一定的限制和调整。因此，理论上可以编译一个完全没有 WmS 参与的“闭眼 Android 系统”，只不过要对相关的接口进行一定的调整。当然，这种“闭眼 Android 系统”是没有任何实际意义的，此处做这种假设只是帮助大家理解 WmS 在系统中的作用。

traversal 内部的总体过程如图 14-43 所示。

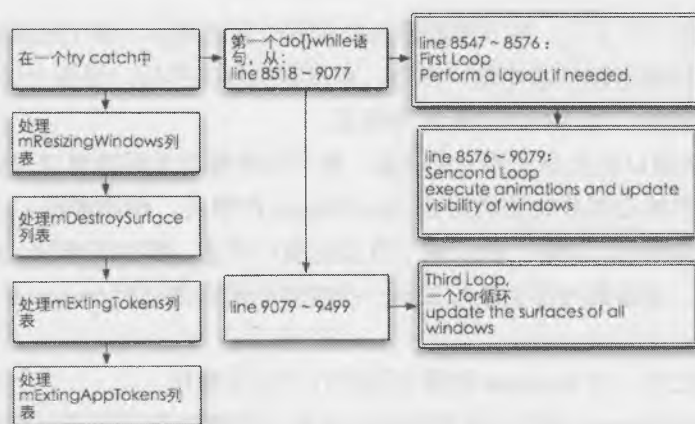


图 14-43 WmS 中 traversal 的主体过程

首先，大部分主要操作都在一个 try catch 中完成，这样可以确保万一有任何的错误而不至于系统崩

溃。在 try catch 中执行的任务根据源码的注释来看，主要包含三个步骤。

① 首先计算每一个窗口的大小尺寸，这段代码主要是调用 `performLayoutLockedInner()` 函数，关于如何计算每个窗口尺寸的细节参见本章前面相关小节。这里值得注意的是，源码中似乎连续执行了 4 次该函数，如以下代码所示：

```

8547         // FIRST LOOP: Perform a layout, if needed.
8548         if (repeats < 4) {
8549             changes = performLayoutLockedInner();
8550             if (changes != 0) {
8551                 continue;
8552             }
8553         } else {
8554             Slog.w(TAG, "Layout repeat skipped after too
8555                 changes = 0;
8556         }

```

即只要当 `performLayoutLockedInner()` 返回值不等于 0 时，就 `continue`，这样就会重新调用一次该函数，然而实际情况是 `performLayoutLockedInner()` 总会返回 0，因此，该函数实际上仅执行一次。源码中之所以这样编写，笔者猜测是对代码进行了重构，并且保留了之前的一点痕迹，这可以认为是多余的。在我们的程序设计中也存在同样的情况，所以读者不必在此疑惑为什么要重复 4 次。

`performLayoutLockedInner()` 之所以总是返回 0 的原因在于，该函数中，返回值来源于 `mPolicy.finishLayoutLw()`，而 `Policy` 的该函数则简单地返回了 0 值，如以下代码所示：

```

1486         /** {@inheritDoc} */
1487         public int finishLayoutLw() {
1488             return 0;
1489         }

```

② 计算完所有的窗口尺寸后，接下来就需要决定哪个应该显示、哪个应该隐藏的问题了。该步骤中包含了处理进入窗口和退出窗口的动画。当然，本步骤执行完毕后，界面上还不能反映出这些变化，因为本步骤仅仅是给和隐藏、显示相关的变量中赋值。

③ 此时，已经获知窗口的大小及显示的状态，剩下的事情就是根据窗口变量中的这些信息将窗口真正地体现到屏幕上。其核心部分就是调用 `SurfaceFlinger` 的服务，绘制屏幕。

完成了以上三个大步骤后，屏幕上就反映了真实的窗口状态。实际工作时，由于动画的存在，因此在一个动画尚未结束时，该函数内部会继续发送一些能够引起再次进行 `traversal` 调用的 `Handler` 消息，直到动画结束。

除了以上三大步骤之外，在 `traversal` 的最后还进行了以下操作。

① 处理 `mResizingWindows` 列表。该列表保存的是当前哪些窗口的尺寸都发生过变化，对于这些大小变化的窗口，`WmS` 需要将这个变化通知给窗口客户端，以便客户端能根据新的大小重新绘制窗口，具体就是会 `IPC` 调用到 `ViewRoot.W` 类的 `resized()` 函数。

② 处理 `mDestroySurface` 列表。该列表保存了那些处于不可视的窗口，有两个原因会导致窗口被

添加到该列表中，一个是当该窗口隐藏时，比如从 A 启动 B 后，A 会变为隐藏，其对应的 Surface 会被销毁，所以 A 窗口会被添加到该列表中；另一个是当该窗口被删除后。如果一个窗口被删除，则由于窗口退出动画的存在导致该窗口的 Surface 不会被立即销毁，因此 mDestroySurface 列表暂时就不会包含该窗口，直到退出动画结束后，该窗口才会被添加到该列表，并在下次进行 traversal 时销毁对应的 Surface 对象。

③ 处理 mExitingTokens 和 mExitingAppTokens 列表。前者保存的是 WindowToken 对象，后者保存的是 AppWindowToken 对象，即当一个窗口对象被销毁后，窗口对应的 WindowToken 和 AppWindowToken 对象也“可能”会被销毁，本操作就是简单地删除这些对象而已。这里之所以说“可能”，是因为一个 WindowToken 对象可能会对应多个 WindowState 对象，因此只有当对应的所有 WindowState 对象被删除时才能删除该 WindowToken 对象。同理，一个 AppWindowToken 可能也对应多个 WindowState 对象。

以上就是 traversal 中执行的主体思路。

下面分节逐个介绍三大步骤。

### 14.6.2 第一大步骤：计算窗口的大小

关于计算窗口的大小参见 14.4 小节，此处之所以设置该小节，仅仅是为了逻辑上的完整，从而使读者不致忽略此重要步骤。

### 14.6.3 第二大步骤：计算窗口的可视状态

影响窗口可视状态的因素主要有两个。一个是动画因素，即在动画前、动画中，以及动画后窗口自身的显示状态会不断变化；第二个因素是当一个窗口显示或隐藏时，会影响其他窗口的显示或隐藏状态。该步骤的执行流程如图 14-44 所示。

在一个 for 循环中设置 mWindows 列表中的每一个窗口的动画，当然，如果窗口没有动画的话就会跳过该窗口。

① 判断窗口的 mSurface 是否为空，如果为空，则该窗口实际上在 SurfaceFlinger 中就没有真正的 Surface 对象，因此就跳过动画。

(1) 当 mSurface 不为空时，才执行后续步骤。首先调用窗口的 commitFinishDrawingLocked() 函数，该函数返回值的意义代表当前窗口对象客户端是否已经完成对窗口的绘制，如果完成，其内部会调用 performShowLocked() 将窗口置于可视状态。当返回值为 true 并且该窗口的 flag 中标明要显示墙纸时，则将变量 wallpaperMayChange 置为 true，以便在接下来的操作中重新绘制墙纸。

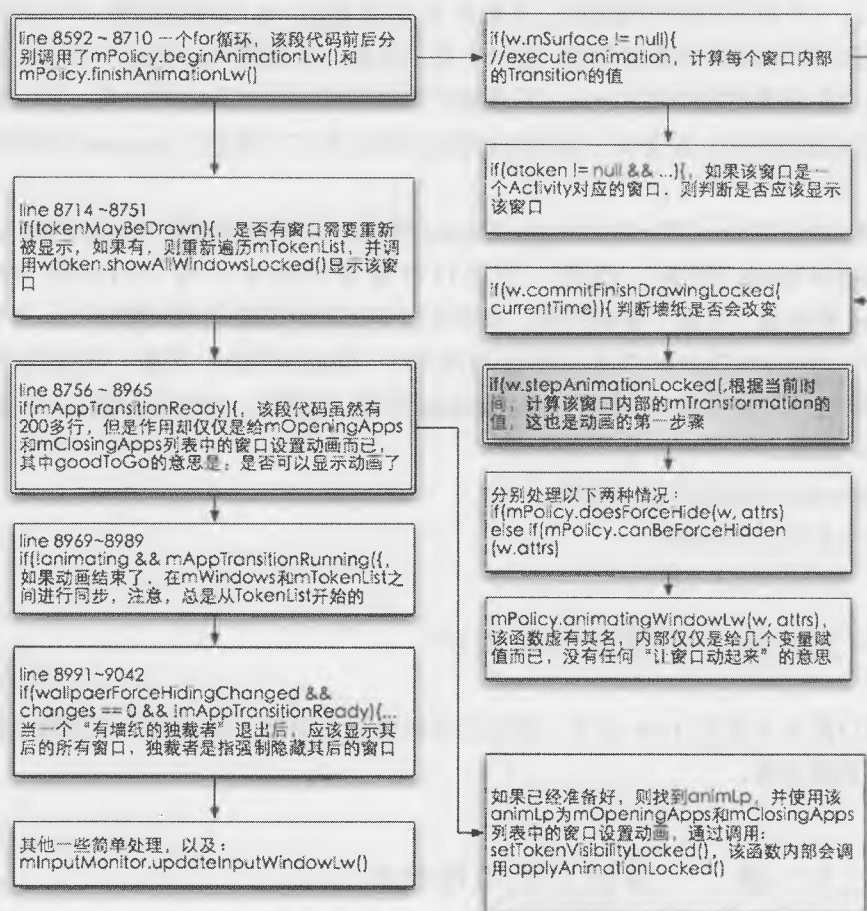


图 14-44 计算窗口可视状态的过程

(2) 调用 `w.stepAnimationLocked()`, 其作用是根据当前时间计算窗口内部的 `mTransformation` 值。该变量保存了窗口动画的参数, 如果该窗口没有动画, 则该函数返回 `false`, 这就是为什么该函数的参数包含了当前时间的原因, 因为窗口动画仅在一小段时间内有效, 当过了动画时间后, 窗口的动画就会为空。

(3) 窗口有两个和隐藏相关的特点, 第一个是窗口可以隐藏其后面的所有窗口, 第二个是窗口是否被前面的窗口隐藏。这两个特点可以在窗口的 `attrs` 属性中进行配置, 本步就是分别处理这两种情况。本步执行的结果是, 如果窗口被设置为隐藏其后面的所有窗口, 则后面的窗口将被置为不可视状态, 具体就是调用窗口的 `hideLw()` 函数, 否则将窗口置为可视状态, 具体就是调用 `showLw()` 函数。`hideLw()` 和 `showLw()` 仅仅是改变窗口的显示参数, 而并未真正将窗口显示到屏幕上, 这需要后续的 `SurfaceFlinger` 完成真正的显示、隐藏。如果窗口被设置为不可被隐藏, 则即使上面的窗口想强制隐藏后面的窗口, 后面的窗口依然会被显示。

(4) 调用 `mPolicy.animationWindowLw()`。该函数表面上似乎是“让窗口动起来”, 实际代码的执



行却并不是这个意思，函数内部仅仅是对不重要的变量进行赋值而已。因为这仅仅是 Policy，而此处的 Policy 实际上并无特别的限制，该函数仅仅是当每次设置完窗口动画时被调用，从而给 Policy 一个收尾的机会。

② 以上单从 mWindows 中各窗口之间的关系设置了窗口的可视状态，但并不代表最终窗口的状态，还需要考虑其他因素。后续的几个步骤中也都是这种思路，即考虑一些其他的因素，从而决定窗口最终的显示状态，这就好比，一个窗口必须“过五关、斩六将”后，方能最终到达目的地。本步就是要检查窗口对应的 AppWindowToken 对象是否可视，该对象是调用 setAppVisibility() 时被设置的，如果窗口对应的 AppWindowToken 对象已经被隐藏，则该窗口是不能被显示的。

③ 判断 tokenMayBeDrawn 是否为 true，该变量为 true 代表了有新的窗口其客户端已经完成了绘制，什么意思呢？在执行本步前，由于有些窗口处于隐藏的原因是其客户端还没有完成绘制，因此，此时应该继续评估一下，看客户端是否已经完成了绘制。如果完成，则需要将该窗口重新置于可视状态，本步中是调用 AppWindowToken 类的 showAllWindowsLocked() 将该 token 对应的窗口置为可视状态的。

④ 判断 mAppTransitionReady 是否为 true。该变量一般是当 AmS 调用 WmS 的 executeAppTransition() 时被置为 true 的，意思是说“请进行 Activity 的切换吧”，当 WmS 收到这个指令后，就可以将进行切换的窗口置为可视窗口了。那些被切换的窗口对象被临时保存到了 mOpeningApps 和 mClosingApps 列表中，前者表示那些启动的窗口，而后者代表那些退出（关闭）的 AppWindowToken 对象。有读者可能要问，mWindows 列表中也包含了应用窗口，mOpeningApps 和 mClosingApps 也包含了同样的应用窗口，那么为什么本步还要单独处理呢？原因上面已经说过，窗口需要通过层层“考验”才能最终决定是否被显示，前面仅仅考虑的是非 transitionReady 的情况，而一旦 transitionReady 后，窗口的可视状态就会发生变化。本步中不但修改状态为可视状态，同时会为 mOpeningApps 和 mClosingApps 中的窗口指定具体的动画方式，比如是以 Task 的方式被退出还是以 Activity 的动画被退出。

⑤ 判断动画是否结束。在动画期间，mWindows 为了完成动画的效果，其内部的窗口顺序及可视状态仅仅是视觉上的，而不是真正程序上看到的。换句话说，用户睁开眼睛去操作和闭着眼睛去操作的结果会有所差异，比如，当某个窗口正在执行退出动画时，界面上虽然依然能看到该窗口中的按钮，但点击后却并不是该窗口的按钮，而很可能是其后面窗口的按钮。而对程序来讲，真正的窗口顺序被保存在 mTokenList 列表中，因此，本步就需要根据 mTokenList 中窗口的顺序对 mWindows 中窗口的顺序进行调整，或者叫同步。

⑥ 在 traversal 时，会存在这样的窗口，它本身会以墙纸为背景，同时又强行隐藏其后的窗口。对于这样的窗口，当其退出后，就会导致其后面所有的窗口被重新显示，同时调整墙纸的显示。

⑦ 执行到此，所有的条件也就检查完毕了，此时的窗口状态就是此刻最终应该在屏幕上的显示状态，还需要做的仅仅是对一些其他变量的赋值，并将所有窗口的信息传递给 InputDispatcher 而已。

至此，计算窗口显示状态的操作就结束了，下一节就将根据这些状态修改窗口对应的 Surface 对象。



## 14.6.4 第三大步骤：通知 SurfaceFlinger 进行窗口重绘

通知 SurfaceFlinger 进行窗口重绘的原理很简单，仅仅是调用 Surface 类的 `setLayout()`、`setSize()`、`setPosition()`、`show()`、`hide()` 等函数告知 SurfaceFlinger 窗口的层值、大小、位置、显示、隐藏状态即可，剩下的事情就由 SurfaceFlinger 去完成了。当然，实际代码中除了这些操作外，还包含一些其他逻辑，比如为 BLUR、DIM 属性的窗口增加一些特别的 Surface 窗口，这些 Surface 不对应 WindowState 对象，而仅仅是以一个 Surface 对象存在，下面就来详细介绍该过程，其流程如图 14-45 所示。

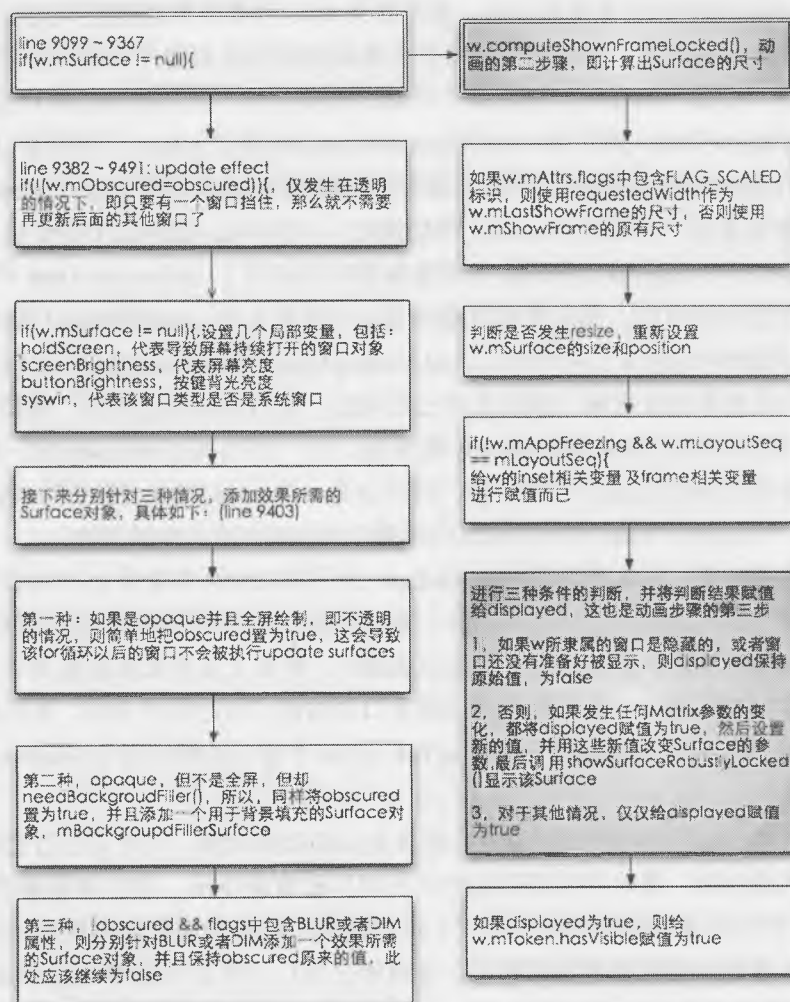


图 14-45 修改窗口对应的 Surface 对象的过程

- ① 计算窗口的显示大小。

(1) 前面步骤中计算的窗口大小是指在程序看来该窗口的大小 `mFrame`，而本步则是调用窗口对象 `w` 的 `computeShowFrameLocked()`。如其函数名称所示，其作用是计算在屏幕上应该显示的大小 `mShowFrame`，在一般情况下，只有当窗口在进行动画时 `mShowFrame` 才不等于 `mFrame`。换句话说，`mShowFrame` 的值是实际屏幕上该窗口显示的大小。

(2) 处理 `attrs` 中包含 `FLAG_SCALED` 标识的情况。该标识的作用是告知系统使用指定的大小显示窗口，在一般情况下，如果没有该标识，则系统会按照 `mShowFrame` 的大小显示窗口，这就会显示出动画的变化过程，因此，如果包含 `FLAG_SCALED` 标识，则用户将不能看到该窗口的动画过程。

(3) 判断窗口尺寸是否发生变化，即 `resize` 变量是否为 `true`。如果变化，则设置窗口对应的 `Surface` 对象的大小 (`size`) 和位置 (`position`)。

(4) 给窗口的 `inset` 和 `frame` 相关变量进行赋值，这些变量包含了和窗口的尺寸相关的信息，详见前面小节中关于窗口尺寸的描述。

(5) 进行三种条件的判断，并将判断结果赋值给 `displayed` 变量。该变量的含义是指窗口是否最终被显示到屏幕上了，实际上，代码中给 `displayed` 变量赋值后却没有使用过该变量，因此，该变量可认为是无意义的。这三种条件分别如下：

① 如果窗口所隶属的窗口是隐藏的，则该窗口不能被显示，或者窗口的 `isReadyForDisplay()` 返回为 `false`，该窗口也不能被显示。如果是隐藏的话，则应该调用窗口内 `Surface` 对象的 `hideLw()` 函数将其置为隐藏。

② 否则，如果窗口动画的 `Matrix` 值发生任何变化，该窗口都应该被重新显示。该步内部调用窗口包含的 `Surface` 对象的 `setAlpha()`、`setLayout()`、`setMatrix()` 等函数用于改变 `Surface` 对象的显示参数。执行完毕后，将 `displayed` 变量置为 `true`。

③ 对于其他情况，仅仅是将 `displayed` 置为 `true`。

(6) 如果 `displayed` 为 `true`，则将 `w.mToken.hasVisible` 置为 `true`，意思是说该窗口对应的 `WindowToken` 对象包含可视的窗口。

② 设置了几个和窗口效果相关的变量。

③ 接下来分别针对三种显示效果添加所需的 `Surface` 对象。

(1) 第一种情况，如果不透明 (`opaque`) 并且全屏显示，则把 `obscured` 置为 `true`，这会导致 `for` 循环后面的窗口不会被 `update surfaces`。也就是说，在这种情况下就不需要更新其后面的窗口了，因为反正用户看不见后面的窗口。对于 `Activity` 切换时，大部分都是这种情况，`Activity` 窗口默认都是全屏并且不透明。

(2) 第二种情况，不透明、不是全屏并且 `needBackgroundFilter()`，在这种情况下，将添加一个特殊的 `Surface` 紧跟该窗口层值之后。这个特殊的 `Surface` 就是 `mBackgroundFilterSurface`，其特殊性在于它不包含阿尔法通道，因此对于用户而言，将看不到窗口背后的内容，这种情况默认并不会出现。

(3) 第三种情况，透明，并且窗口参数包含 `DIM` 或者 `BLUR` 标识，这种情况比较常见，前面也介绍过两者的界面效果。这种效果的本质是添加了一个特殊的 `Surface`，`DIM` 对应的 `Surface` 在创建时包含参数 `Surface.FX_SURFACE_DIM`，而 `BLUR` 包含的参数为 `Surface.FX_SURFACE_BLUR`。这两个参

数是如何起作用的呢？这就要说到 SurfaceFlinger(后面简称 sf)的内部原理了，sf 操作的对象是 Surface，而从设备的屏幕来讲，它并没有这么多的 Surface，sf 仅仅是对逻辑上的 Surface 进行软件运算，从而计算出这些 Surface 最终在屏幕上显示的像素。并且，对于显示器而言，也不存在什么阿尔法通道，这些都仅仅是软件上的概念。当然有些 ARM 处理器内部包含了一些图像加速引擎，这部分引擎是硬件实现的，它会对外提供一些操作接口，这种接口可能会包含阿尔法通道。但无论是通过硬件加速引擎实现还是纯软件实现，在进行 Surface 复合时，一般都会从最底层的窗口开始进行复合，然后逐层进行，直到计算出最终的显示像素值。在这个复合的过程中，如果 Surface 的参数是 DIM 或者 BLUR，则 sf 会将该 Surface 底层的显示内容重新计算出一个显示效果，并将该效果作为后面继续处理的窗口的背景进行复合。因此，对于用户来讲，所看到的 DIM 或者 BLUR 窗口的背景窗口实际上已经不是原始的窗口，而是 sf 用原始窗口构造出的一个新界面。该过程可以用以下代码来表示：

```
Screen outScreen;
for(WindowState win: mWindows){
    win.draw(outScreen);
    if(flag == DIM || flag == BLUR){
        createNewScreen(outScreen);
    }
}
outScreen.drawOnScreen();
```

至此，一次 traversal 过程就结束了。

## 14.7 窗口动画

Framework 中的动画可分为两类，一类是视图动画，另一类就是窗口动画。视图动画前面已经介绍过，其基本原理是用纯软件的方式对视图进行变换然后重绘到屏幕上；而窗口动画则是 SurfaceFlinger 对窗口 Surface 进行变换，然后重绘到屏幕上，至于 SurfaceFlinger 内部使用纯软件还是硬件的方式则取决于底层平台，Framework 本身并不在意平台使用何种方式。

由于窗口动画变换的对象是 Surface，而视图动画变换的对象是 View，因此窗口动画的变换方式没有视图动画那么灵活。视图动画理论上可以对视图做任意的变换，从而产生任意动画，而窗口动画则一般出于性能考虑，仅能进行有限的变换。窗口动画所能支持的动画方式已经全部在 Surface 类中被定义，这些可以被变换的参数包括位置、大小、缩放比例、阿尔法值四个，唯独没有角度参数，这就是为什么窗口动画无法添加旋转效果的原因。

窗口动画的实现原理比较简单，和视图动画相似，其思路是先给窗口设置一个动画方式，然后再根据当前时间计算出该动画对应的位置、大小、缩放比例、阿尔法值，然后每次在 SurfaceFlinger 绘制之前将这些新的参数设置给对应的 Surface 对象，最后 SurfaceFlinger 就会根据这些参数自动对窗口进行变换，并将变换后的窗口绘制到屏幕上。

下面从代码的角度来看动画的具体流程。

窗口的动画是由 Animation 类表示的，每个 WindowState 对象内部都有一个 mAnimation 变量，用

于保存当前窗口的动画，那么该变量是如何被赋值的呢？请看图 14-46 所示。

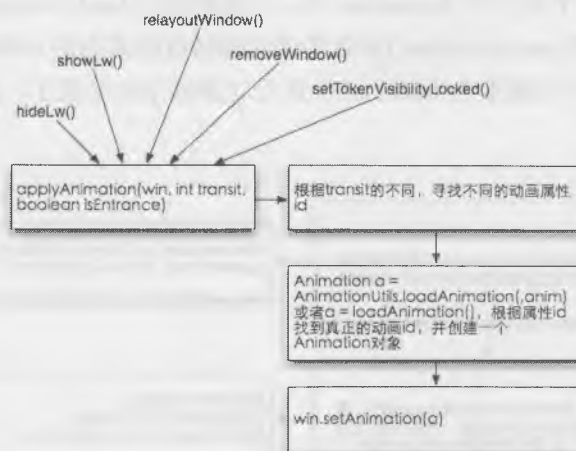


图 14-46 窗口动画的赋值方式

在 WmS 内部有一个 `applyAnimation()` 函数，用于给 `WindowState` 对象指定一个动画。该函数在很多情况下都会被调用，具体包括 `hideLw()`、`showLw()`、`relayoutWindow()`、`removeWindow()`、`setTokenVisibiltyLocked()`，等等，关于这些函数的调用时机请参照前面相关小节。`applyAnimation()` 函数有三个参数，第一个为要操作的 `WindowState` 对象；第二个是一个 `int` 型变量 `transit`，表明动画的类型，函数内部会将这个 `int` 型变量转换为具体的 `Animation` 对象，并赋值给 `win` 对象中的 `mAnimation` 变量；第三个变量为 `true` 时，则如果正在进行动画，那么就不要再更改该动画，为 `false` 时，则更改当前正在进行的动画。

在 `applyAnimation()` 内部，通过三个函数调用把 `int` 型变量 `transit` 转换为一个 `Animation` 对象。

① `transit` 变量代表了动画的类型，比如进入动画、退出动画等。应用程序可以给不同的属性赋值用于指定不同的动画方式，比如可以给 `windowEnterAnimation`、`windowExitAnimation` 参数赋值用于指定具体的动画。因此，在本步中，系统需要决定读取哪个属性。常见的 `transit` 值对应的属性 `id` 如表 14-6 所示。

表 14-6 不同 `transit` 值对应的动画属性

transit 值	对应的属性 id
WindowManagerPolicy. TRANSIT_ENTER	WindowAnimation_windowEnterAnimation
TRANSIT_EXIT	WindowAnimation_windowExitAnimation
TRANSIT_SHOW	WindowAnimation_windowShowAnimation
TRANSIT_HIDE	WindowAnimation_windowHideAnimation

② 得到属性 id 后, 就可以读取该 id 指定的值, 该值一般都对应了动画文件的 id, 于是就可以根据最终的动画文件 id 创建一个真正的 Animation 类。该过程是在 loadAnimation() 函数中完成的。

③ 调用 WindowState 的 setAnimation() 函数将该动画赋值给其内的 mWindow 变量。

完成了动画赋值后, 接下来就会在 traversal 中进行动画效果的呈现了, 其内部过程可分为三步, 如图 14-47 所示。

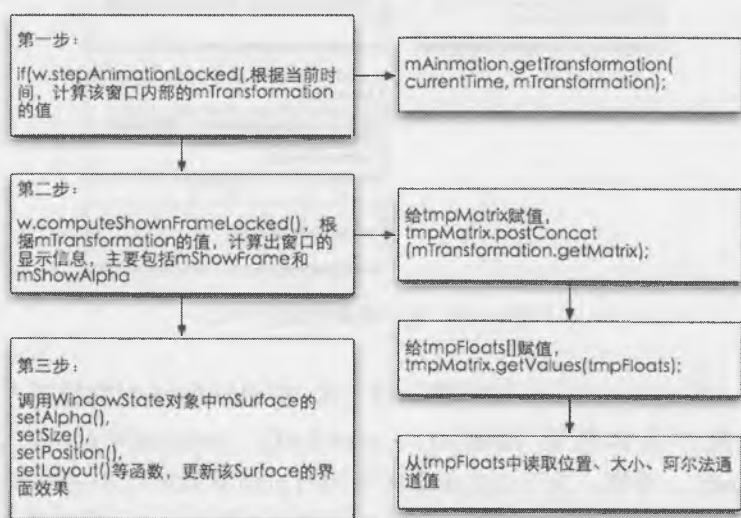


图 14-47 动画的赋值和绘制过程

这些步骤都是在 traversal 中完成的, 但这三步并不是在一个 for 循环中被执行的。首先对于第一步, 这是在一个 for() 循环中执行的, 循环的主体是 mWindows 列表, 即首先会针对每一个 WindowState 对象执行其 stepAnimationLocked() 函数。该函数内部的一句关键代码是 mAnimation.getTransformation(), 该句代码中 mAnimation 代表窗口包含的动画。当然, 窗口不一定都有动画, 如果没有动画的话, 则 mAnimation 为空, stepAnimationLocked() 返回为 false。对于有动画的情况, 该句代码就会根据当前时间重新计算变换后的 Transformation 的值, 并将其保存到窗口对象的内部变量 mTransformation 中。

② 该步骤是在另外一个 for() 循环中, 循环的主体依然是 mWindows, 即逐个针对每一个窗口调用其 computeShownFrameLocked() 方法, 其作用是根据窗口中的 mTransformation 值计算出其对应的窗口显示相关的信息, 计算的过程如下。

(1) 给局部变量 tmpMatrix 赋值, 其值来源于 mTransformation, 也就是说将这个 mTransformation 值的格式转换为一个 Matrix 类型。

(2) 调用 tmpMatrix.getValues() 函数, 将这个 Matrix 中的值转换到一个数组 tmpFloats 中, 以便引用。

(3) 从 tmpFloats 数组中读取窗口显示相关的信息, 包括位置、大小、阿尔法通道值三个。

③ 该步骤和第二步同在一个 for 循环中, 即每次计算一个窗口的屏幕显示信息后, 就立即将其通

知给 SurfaceFlinger，具体包括调用窗口对象内部变量 mSurface 的 setAlpha() 等函数。当然，调用完这些函数后界面不会立即变化，而是当调用 Surface.closeTransaction() 后才会生效，而调用 Surface.closeTransaction() 是在 traversal 过程的最后。

这三个步骤在源码中的顺序可用以下代码表示：

```
Surface.openTransaction();
for(WindowState w : mWindows){
    w.stepAnimationLocked();
}
for(WindowState w: mWindows){
    w.computeShownFrameLocked();
    w.mSurface.setAlpha();
    w.mSurface.setSize();
    ...
}
Surface.closeTransaction();
```

当然，以上三个步骤仅仅完成了动画过程中的一帧画面的绘制，因此如果动画没有结束会导致再次执行 traversal，从而能够继续绘制下一帧。在每次 traversal 时，stepAnimationLocked 都会产生不同步的 Transformation 值，因为时间在变化，这就最终产生了用户所看到的“动画”。从这点上来看，对 SurfaceFlinger 而言，其实并不存在“动画”的概念，SurfaceFlinger 只是按照请求对 Surface 进行了重绘而已，所以，如果脱离 Android 的 Framework，要自己用 C 语言实现一个动画，则简单地、不断地重绘界面即可，这也是所有动画底层的实现方式，Framework 也不例外。

## 14.8

## 屏幕旋转及 Configuration 的变化过程

当用户旋转屏幕后，在默认情况下应用程序会重新绘制屏幕，应用程序会在 Activity 中收到 onConfigurationChanged() 的回调。本节就来分析这种 Configuration 回调的内部原理，以及最常引起 Configuration 变化的原因——屏幕旋转。

Activity 中 onConfigurationChanged() 函数是在 AmS 中的 updateConfigurationLocked() 函数中被调用的，别无他处，因此，所有和 Configuration 变化相关的函数调用都必须经由 AmS 的这个函数通知客户端 Activity 发生了 Configuration 的改变。

尽管 Configuration 包含许多种因素，比如屏幕方向、键盘类型、语言类型，等等，但对一个固定的 Android 设备而言，最常见的能引起 Configuration 变化的却是屏幕方向。因为不用的应用程序可以设定该应用所希望的屏幕方向，并且屏幕方向也会因用户旋转设备而发生变化。图 14-48 给出了三种常见的引起 Configuration 发生变化的情况。

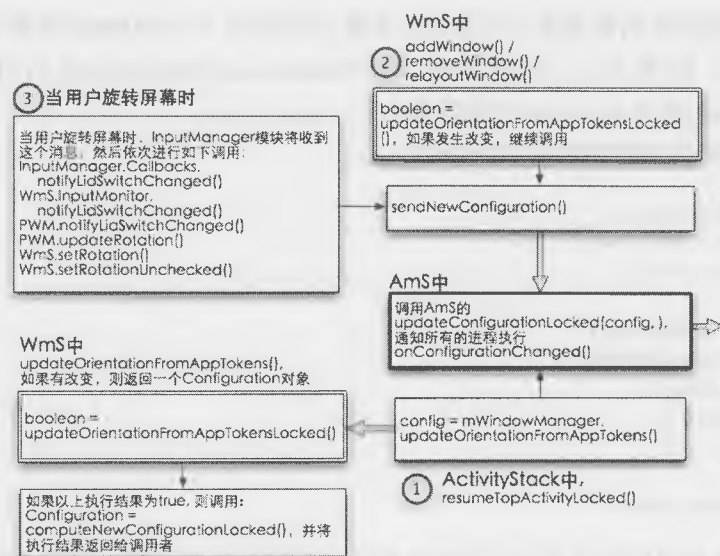


图 14-48 引起 Configuration 改变的三种情况

第一种情况, 在 ActivityStack 中执行 resumeTopActivityLocked()函数时。这种情况包括当从一个 Activity 切换到另一个 Activity 时, 或者从当前 Activity 返回到上一个 Activity 时, 凡是需要重新启动一个 Activity 的情况基本上都会调用到 ActivityStack 的该函数。在这种情况下, ActivityStack 会先询问 WmS, 看 WmS 中是否会改变该 Activity 的屏幕方向, 为什么 WmS 会改变屏幕方向呢? 这就要从配置 Activity 屏幕方向的方式说起了。

应用程序可以在 AndroidManifest.xml 的 Activity 标签中指定 screenOrientation 的属性值, 从而指定该 Activity 窗口的屏幕旋转的方式, 一共有四种方式, 如表 14-7 所示。

表 14-7 指定 Activity 屏幕方向的参数

XML 定义的值	对应 ActivityInfo 类中的常量	解 释
unspecified	SCREEN _ORIENTATION _UNSPECIFIED	未指定默认值。系统将会用当前的屏幕方向作为新窗口的方向
landscape	_LANDSCAPE	强制使用风景模式, 即水平方向
portrait	_PORTRAIT	强制使用人像默认, 即垂直方向
user	_USER	使用用户自定义方向, 即用户可以在系统设置中指定一个固定的方向, 从而使该窗口使用该方向
behind	_BEHIND	使用后面窗口的方向, 比如当从 A 启动到 B 时, B 使用 A 的方向, 因为 A 在 B 后, 这样可以保证方向的一致性

当然, 除了在 XML 中静态指定外, 还可以在程序中动态指定。添加窗口时需要一个



WindowManager.LayoutParams 类型的参数对象, 该对象中包含一个 screenOrientation 变量, 可以在程序中指定该变量的值, 从而改变该窗口的方向模式。

WmS 中 updateOrientationFromAppTokensLocked() 函数的作用就是具体处理这四种模式, 并选择相应的屏幕方向。WmS 内部使用全局变量 mForcedAppOrientation 保存在计算前的屏幕方向, 如果发生方向变化, 则该函数返回 true。于是接下来继续调用 computeNewConfigurationLocked() 函数创建一个新的 Configuration 值, 然后将这个 Configuration 值返回给 ActivityStack, 这也是为什么 Configuration 类实现 Parcelable 接口的原因, 因为它传递的信息是通过 IPC 调用完成的。

第二种情况, 在 WmS 中调用 addWindow()、removeWindow()、relayoutWindow() 函数, 也需要重新计算屏幕方向是否发生改变, 如果有改变, 则调用 sendNewConfiguration() 向 AmS 报告新的 Configuration 值。

第三种情况, 当用户人为旋转设备时, 旋转消息首先会被 InputManager 模块截获, 从而使 InputDispatcher 能够根据旋转后的屏幕派发触摸消息。接下来会调用到 WmS 的 setRotation(), 再调用 setRotationLocked(), 然后又调用到 sendNewConfiguration() 函数。

以上三种情况下最终调用到了 AmS 的 updateConfigurationLocked() 函数, 该函数将通知所有客户端 Configuration 发生了改变, 这会导致所有的正在运行的 Activity 收到 onConfigurationChanged() 回调, 不仅是当前在屏幕上显示的 Activity, 还有后台那些没有 destroy 的 Activity。

下面继续来看 updateOrientationFromAppTokensLocked() 函数内部是如何计算屏幕方向的, 其流程如图 14-49 所示。

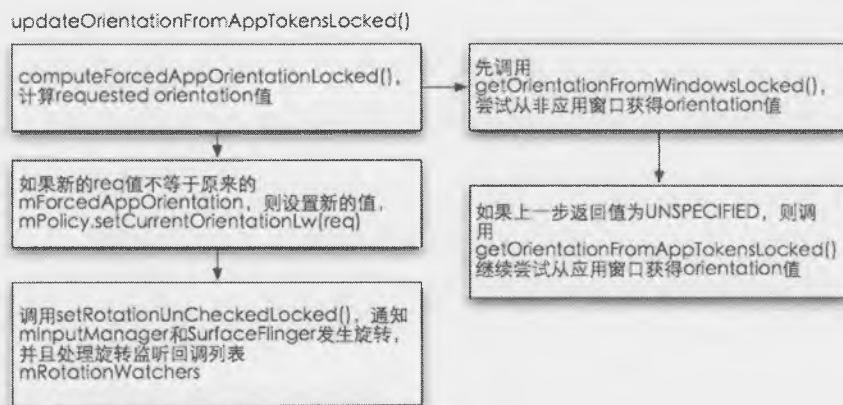


图 14-49 WmS 中计算当前屏幕方向的流程

#### ① 调用 computeForcedAppOrientationLocked() 计算屏幕方向, 其内部包含两个重要步骤。

(1) 首先尝试从那些非应用窗口获得方向值, 即从 mWindows 列表中最上层的窗口开始计算, 并且窗口的 mAppToken 变量必须为空, 如果遇到应用类窗口, 则立即返回, 并尝试下一步的应用类窗口计算。

(2) 尝试从 `mAppTokens` 列表对应的窗口计算其方向值。

② 如果计算出的方向值和 `WmS` 中的 `mForcedAppOrientation` 值不相同, 则设置新的值, 并调用 `mPolicy.setCurrentOrientationLw()` 通知 `Policy` 新的屏幕方向。

③ 调用 `setRotationUncheckedLocked()` 设置新的方向值, 具体包括通知 `mInputManager` 和 `SurfaceFlinger` 有新的方向值, 然后再回调 `mRotationWatchers` 列表中注册的方向旋转监听代码。

以上就是 `Configuration` 变化的系统内部调度过程。



## 第 3 部分 系统篇

第 15 章 资源访问机制

第 16 章 程序包管理

(PackageManagerService)

第 17 章 输入法框架



## 第 15 章 资源访问机制

资源访问机制

Android 提供了一种灵活的资源架构，程序员可以使用 XML 文件描述各种资源，包括文本字符串、颜色值、界面布局、图片等，从而使得纯粹的程序和这些资源的设计可以很好地进行分离，这有助于 UI 设计师和程序员协同工作。

本章首先介绍资源的定义及在最终文件中的存储，然后介绍资源调用的过程，包括调用用户自定义资源及系统资源，最后介绍一种实现切换系统主题的思路。

### 15.1 定义资源

Android 中的资源从类型的角度来看，包括 drawable、layout、字符串、颜色值、menu、animation 等，而对于像 layout、animation、menu 这些资源来讲，仅仅是从指定的 XML 文件中获取数据而已。该过程比较简单，而复杂的是对某个特定的 XML 里面的 element 进行解析，比如对一个 layout 文件中的某个 TextView 所包含的属性及属性值的解析，这些是本章所讲的重点。因此，本章所讲的资源定义特指这种属性及属性值的定义。

如上所述，资源的定义可以分为两类，一类是属性的定义，一类是值的定义，这就像是变量名称的定义和变量赋值。

比如，要描述一个 TextView 的特征，可以在相关的 Layout 文件中为 TextView 的属性指定特定的值。比如 `android:textColor="#ff0000ff"`，textColor 就是一个属性，而 #ff0000ff 则是一个赋值。再比如，`android:background="@drawable/bkg"`，background 就是一个属性，@drawable/bkg 则是一个赋值。

常见的值一般有以下几种。

- String、Color、boolean、int 类型：res/values/xxx.xml 文件中指定。
- Drawable 类型：res/drawable/xxx.png 文件中指定。

- 布局(layout): res/layout/xxx.xml 文件中指定。
- 样式(style): res/values/yyy.xml 文件中指定, yyy 一般为 “style”, 样式本质上是一些指定的属性配上指定的值, 是一种模块化的设计, 可以为一些具有相同样式的 View/ViewGroup 使用 style="style\_name" 语句进行赋值, 从而简化开发。

属性的定义文件在 res/values/zzz.xml, zzz 一般为 “attrs”, 属性的定义规范如下代码所示:

```
<declare-styleable name="Window">
    <attr name="windowBackground" />
    <attr name="windowContentOverlay" />
    <attr name="windowFrame" />
    <attr name="windowNoTitle" />
</declare-styleable>
```

styleable 的含义是 “可以样式化的”, 从代码的角度来看, 这里所谓的样式和之前所说的 style 基本上没有关系, 这也是让很多用户困惑的原因, 我们总以为 style 和 styleable 有某种对应关系, 实际上 styleable 仅仅是在 R.java 中增加了一个 int[] 数组而已, aapt 为每一个 attr 分配一个指定的 id 值, 这个数组的内容正是该 styleable 所包含的所有 attr 的 id 值。

name 后面的名称表面上看是对应了某个 Java 类, 而代码上并无依赖关系, 可以使用任何的名称。

attr 的名称在该应用程序范围中必须是唯一的, 即无论有几个资源文件, 或者无论定义几个 styleable, attr 必须是唯一的。如果在 Window 中声明了某个 attr, 比如 windowBackground, 那么就不能在其他的 styleable 中再次声明, 不过这里要注意 “声明” 和 “引用” 的区别, 如果在 attr 后面仅仅有一个 name, 而没有 format, 那么就是引用, 否则就是声明, 只能有一个声明, 但可以有多个引用。有读者说: 有时的确需要在多个样式中包含相同的属性, 比如上面代码中的 windowBackground 属性, 除了在 Windows 中需要, 也有可能 Activity 中需要 (如果定义了一个 Activity styleable)。这个问题实际上混淆了两个概念。

第一, 所谓的 Window 或者 Activity 在实际需求中应该对应的是两个样式 (style), 而不是 styleable, 后者仅仅是为了获取某个 View/ViewGroup 的属性值而设计的一种结构, 而样式 (style) 可能才是你所需要的。因此, 你可以定义两个样式, 并且在两个样式中同时为 windowBackground 赋值。

第二, 在一个 styleable 中使用 attr 指定的属性并不都是 “声明”, 也可以是 “引用”。所以, 如果你真的是需要定义两个 styleable, 那么, 可以一次声明多次引用, 比如, 上面代码中所定义的 attr 都是引用, 而具体的定义则如下代码所示:

```
<attr name="windowBackground" format="reference" />
```

有了以上定义的属性后, 就可以在 XML 文件的 element 中使用这些属性了。

## 15.2 存储资源

要弄清楚程序如何使用资源, 需要先弄清楚资源的存储格式。

应用程序最终都是以一个 APK 方式发布的，首先来看看 APK 里面都有些什么。

APK 本身是一个 zip 文件，这就像 Jar 包一样，都是一个 zip 文件，因此，可以使用 unzip 解压一个 APK，看看里面到底都包含了什么，如以下命令所示。

```
$ unzip ./bin/MyQin.apk -d temp
```

解压后的内容在 temp 目录中，具体包含以下文件以及目录。

- **AndroidManifest.xml**: 该文件的内容与开发时的文件内容基本相同，所不同的是这是一个二进制的 XML 文件。查看二进制的 XML 文件可以使用 AXMLPrinter.jar 工具包，该工具包可以从 <http://code.google.com/p/android4me/downloads/list> 处下载。用法如下。

```
$ java -jar ~/Desktop/AXMLPrinter2.jar AndroidManifest.xml
```

- **META-INF**: 前面说过，APK 文件本身是个 zip 文件，同时也是一个 Jar 文件，该文件夹的内容与普通 Jar 包中的 META-INF 文件类似。
- **res**: 该目录包含了开发工程中 res 目录下除 values 以外的所有内容，一般包含各种 layout 和 drawable，layout 的内容都是以二进制 XML 格式保存的，而所有的 drawable 图像资源都是未经压缩过的原始图像文件。事实上，一般所见的 png、jpg 图像文件本身就是已经压缩过的文件，因此，无须再压缩。
- **classes.dex**: 该文件是真正的类似于 Java Class 的文件，dex 文件格式是 google 发明的，这种格式的作用和 Java Class 文件相同。所不同的是，为了能够快速读取 Class 文件并让 Class 在解释时占用更少的内存，google 对标准的 Class 文件进行了重新的格式优化，这就是 dex。
- **resources.arsc**: 该文件是一种二进制格式的文件，与二进制的 XML 完全不同。aapt 在对资源文件进行编译时，会为每一个资源分配唯一的 id 值，程序在执行时会根据这些 id 值读取特定的资源，而 resources.arsc 文件正是包含了所有 id 值的一个数据集合。在该文件中，如果某个 id 对应的资源是 String 或者数值(包括 int, long 等)，那么该文件会直接包含相应的值，如果 id 对应的资源是某个 layout 或者 drawable 资源，那么该文件会存入对应资源的路径地址。这里的 arsc 的含义有可能是 assembled resource，但这仅仅是猜测。

APK 程序在执行时，并不是直接从 APK 中读取程序文件，可以这么认为，APK 是一种具有安装格式的文件，安装后将是另外一个样子。

与安装后相关的目录如下。

- **/data/app**: 当安装一个 APK 时，系统首先会把该文件复制到该目录下，如果是系统程序，比如出厂自带的程序，则一般位于/system/app 目录下。
- **/data/dalvik-cache**: 如上一节所讲，每个 APK 中都包含一个 classes.dex 文件，系统在安装 APK 程序时，会把该 APK 中包含的 classes.dex 解压出来放到该目录下，为的是当用户启动该程序时能够快速读取类文件，系统应用程序中的 classes.dex 文件同样也放到该目录下。该目录中的文件命名格式为“apk 路径 + classes.dex”，其中 APK 路径中的子目录符号用@代替，比如：

data@app@com.android.contacts-1.apk@classes.dex。

在一些早期的 Android 版本中, 由于不同的平台有一定的差异, 因此有时还可以对 dex 进行一定的优化, 最后生成一种 odex 文件, 该文件也在该目录下。

任何程序都可以读/写该目录, 这就为类的动态装载提供了可能。比如可以安装某个 APK, 该 APK 往往是所谓的插件程序, 本身并不能独立执行, 而仅仅是为其他程序提供一些实现类, 然后在该目录下找到对应的 dex 文件, 再调用系统中的 Dex ClassLoader, 即可实现动态加载, 关于动态加载参见本书第 2 章。

- /data/data: 每个应用程序都有自己的数据目录, 目录名称为应用程序的包名, 所有的数据目录都在该目录下。

并不是说程序安装好后, /data/app 或者 /system/app 目录下的原始文件就可以删除了, 事实上, 当程序运行时, 所需要的资源文件都要在原始文件中读取。加载资源时, 首先加载 resources.arsc, 然后根据 id 值找到指定的资源。



### styleable、style、attr、theme 的意义

styleable 一般和 attr 联合使用, 用于定义一些属性, 理论上讲, 可以完全不用 styleable, 而只用 attr 就可以了。如果是这样, 那么当从一个属性数组中获取所感兴趣的某些属性值时就会显得有些烦琐, 如下所示。

```
<element
  attr_name1 = "value1"
  attr_name2 = "value2"
  ...
  attr_nameN = "valueN"
/>
```

对于这样一个定义, 当从 XML 解析时, 会返回一个 AttributeSet 对象, 该对象包括了 element 元素中定义的所有属性和值, 从 attr\_name1 到 attr\_nameN。如果想要获取 attr\_name2、attr\_name4、attr\_name5 三个属性的值, 那么必须把这三个属性对应的 id 作为 AttributeSet 对象的相应方法的参数, 比如: R.id.attr\_name2、R.id.attr\_name4、R.id.attr\_name5。这种写法显然有些不容易理解, 不能体现出这三个属性联合在一起的意义, 于是才出现了 styleable。

如果把这三个属性放在一个 styleable 中, 如下所示。

```
<declare-styleable name="Flower">
  <attr name="attr_name2" />
  <attr name="attr_name4" />
  <attr name="attr_name5" />
</declare-styleable>
```

这样的话, 当从 AttributeSet 对象中获取这三个属性时, 就可以把 R.styleable.Flower 作为参数, 该参数实际上会被 aapt 编译成为一个 int[] 数组, 数组的内容正是所包含的 attr 的 id, 本质上与不用 styleable



是相同的。

由此也可以看出，styleable 与 style 没有任何关系，除了名称容易让人混淆外。

Theme 一般翻译为“主题”，主题本身是一个抽象化的概念，比如 Windows 操作系统中的主题包含了桌面背景、系统图标、字体大小、按钮风格等，而在 Android 中，从代码的角度来看，主题包含三个方面，其出处和用法如下所述。

第一，在 framework/base/core/res/res/values/attrs.xml 中定义了一个名称为 theme 的 styleable，其中包含了几十项属性，包括窗口样式、字体颜色、大小、按钮样式、进度条等。这些属性大部分只能用作 AndroidManifest 文件中 Activity 元素或 Application 元素，而不能用于普通的 View/ViewGroup。原因很简单，attrs 本身已经为所有的 View/ViewGroup 定义了可用的属性，前面曾说过，styleable 仅仅是为了更方便地访问一些特定的 attr 集合，而把这些特定的 attr 放到一个 styleable 中，theme 也是这样，它把适合于 Activity 或者 Application 的属性单独放到了一起。

第二，在 framework/base/core/res/res/values/attrs\_manifest.xml 中定义了一个名称为 theme 的属性，`<attr name="theme" format="reference" />`。该属性值只有定义到 AndroidManifest.xml 中才是有意义的。进一步讲，只有定义到 Application 和 Activity 元素内部才是有意义的，因为程序中只有在这两个元素中才读取 theme 属性的值。理论上讲，用户也可以给 TextView 添加 theme 属性，aapt 照样能够编译通过，因为 theme 已经被声明过了，所以不会出错，然而，TextView 却并不会理会 theme 属性对应的值，因为在 TextView 类的内部不会读取该属性。theme 的赋值类型是一个引用，那么，应该引用到什么类型呢？答案是 style，即 theme 应该赋值到某个 style，而 style 中包含的属性应该是 Application 或者 Activity 所识别的属性，也就是名称为 theme 的 styleable 中所包含的属性。

第三，在 framework/base/core/res/res/values/theme.xml 中定义了一些名称为 theme 的 style，这些 style 将作为 Activity 或者 Application 的 theme 的值，而这些 style 中所包含的属性名称全部取自于 styleable name="Theme" 处。

以上三者的关系可总结为如图 15-1 所示。

如图 15-1 所示主要分为上下两个部分，上图是在 AndroidManifest.xml 文件中使用属性与 theme 相关属性 (attr) 的过程，下图是在普通 layout 文件中为 View/ViewGroup 指定与 style 相关的过程。

在 AndroidManifest.xml 中，指定 theme 时必须使用命名空间 android，赋值为某个 style，style 中必须包含一些属性 (attr)，这些属性的定义源自 attrs.xml。theme 只能用于 AndroidManifest.xml 中的 Activity 或者 Application 元素，当然，语法上讲用户可以为一个 TextView 元素指定一个 android:theme 属性，但是这是不起作用的，因为只有 Activity 提供了 setTheme() 方法。实际上 Activity 的 setTheme() 方法会调用 context.setTheme() 方法，而实现 Context 接口的是 ContextImpl 类，该类中处理了 styleable 为 theme 的属性值。

在 View/ViewGroup 中使用 style 时则不能使用命名空间，原因是 style 本身并不是一个 attr。这不像 theme，当程序读取 XML 文件时，如果发现是一个 style，则会从 style 所指向的 id 处重新找到该 style 所包含的属性值。

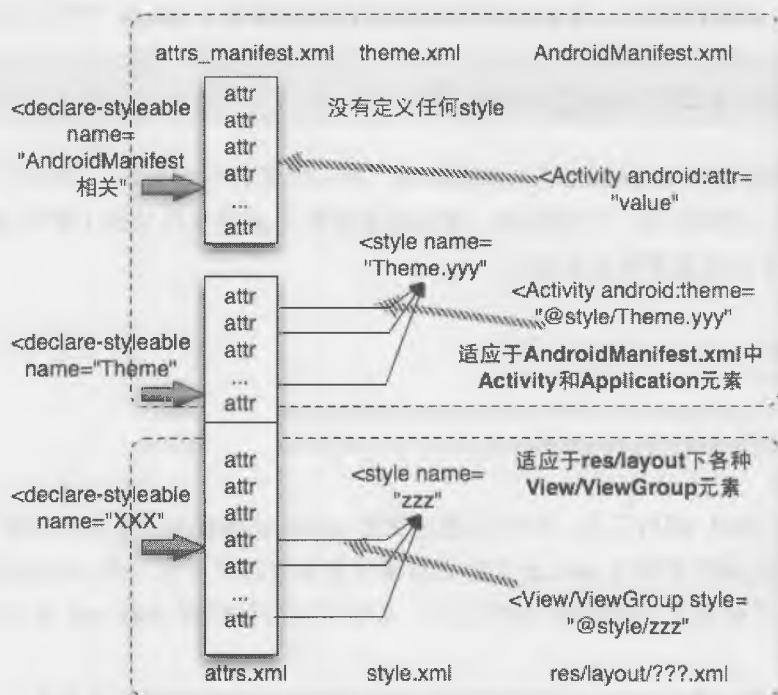


图 15-1 主题相关定义的关系

在笔者看来，framework 中 styleable、style、theme 这几个概念的实现有三个瑕疵。

首先，既然 styleable 和 style 没有任何的关系，那么这两个概念的名称应该分开。style 很容易理解，即为样式，有相同样式的元素其视觉效果也应该相同。而 styleable 的意思——“样式化的”则有点费解，而且其实质只是一些属性的名称集合而已，在一般情况下，一个集合对应的是某个 android.view.widget 的视图属性。因此，styleable 可重新命名为 viewparas，即“视图参数”，相应的 `<declare-styleable>` 也可重新定义为 `<declare-viewparams>`，这样似乎大家就更容易理解了。

其次，framework/base/core/res/res/values/attrs.xml 和 manifest\_attrs.xml 中共同定义了 XML 文件中元素可以有的所有属性，但很明显，attrs 主要用于视图类的属性定义，而 manifest\_attrs 主要用于 AndroidManifest.xml，那么就应该把名称为 Theme 的 styleable 的定义放入 manifest\_attrs.xml 文件中，而不是 attrs.xml 文件中。

最后，在 AndroidManifest.xml 指定 Activity 的 theme 时需要加上命名空间 android，即必须写成 “android:theme=” 的格式，而在视图类中却直接使用 “style=” 的方式，这让程序员有点混淆：“为什么不一样呢？”而事实上完全可以把 style 也作为一个 attr，从而完全统一。

从图 15-1 中也可以看出，theme 本身只是一种属性集合，只是一个 styleable，而要实现一个主题，还必须使用 style 定义一组具体的属性值对。主题仅用于某个 Activity 或整个程序 Application，而不能用于某个视图，系统提供了 API 接口 `setTheme()` 可以为某个 Activity 指定特别的主题。该函数的参数是

一个 style 类型，即一组属性值对，该 style 所包含的属性必须源自于 theme 中包含的属性名称。

## 15.4 AttributeSet 与 TypedArray 类

AttributeSet 类的位置在 android.util.AttributeSet，单从位置就可以看出，该类的作用与 Framework 的内核并无直接关系，而纯粹是一个辅助类。事实也是这样，该类仅仅是为了解析 XML 文件之用。

一般 XML 文件中的元素都有如下格式：

```
<ElementName
  attr_name1="value1"
  attr_name2="value2"
  ...
  attr_nameN="valueN"
/>
```

如果使用一般的 XML 解析工具，则可以通过类似 getElementById() 等方法获取属性的名称及属性的值，然而却还没有在属性名称与 attrs.xml 定义的属性名称间建立关系。而 AttributeSet 类正是在这之间建立了某种联系，并提供了一些新的 API 接口，从而可以方便根据 attrs.xml 中已有的名称获取相应的值。

AttributeSet 本身是一个 interface，Framework 中如何实现这些接口我们并不关心，而只是关心该接口所提供的几个重要 API。AttributeSet 对象一般作为 View 的构造函数的参数传递过来，程序员只需要在构造函数中通过该对象获取相应的属性值即可。比如，TextView 的构造函数如以下代码所示：

```
324 public TextView(Context context,
325                  AttributeSet attrs,
326                  int defStyle) {
```

AttributeSet 中的 API 可按功能划分为以下几类，假定该 XML 的格式为：

```
<View class="android.widget.TextView"
  android:layout_width="@dimen/general_width"
  android:layout_height="wrap_content"
  android:text="@string/hello"
  android:id="@+id/output"
  style="@style/Test"
/>
```

第一类，操作特定属性，包括以下几类。

- public String getIdAttribute(): 获取 id 属性对应的字符串，返回值为"@+id/output"。
- public String getClassAttribute(): 获取 class 对应的字符串，返回值为"android.widget.TextView"。
- public int getStyleAttribute(): 获取 style 对应的字符串，返回值为"@style/Test"。
- public int getIdAttributeResourceValue(int defaultValue): 返回 id 属性对应的 int 值，此值应该是 R.id.output 变量的值。

第二类，操作通用属性，包括以下几类。

- `public int getAttributeCount()`, 获取属性的数目, 本例中, 返回值为 6。
- `public String getAttributeName(int index)`, 根据属性所在的位置返回相应的属性名称。本例中, 相应的属性位置如下:

```
class=0
layout_width=1
layout_height=2
text=3
id=4
style=5
```

如果 `index` 为 1, 则返回 `android:layout_width`。

- `public String getAttributeValue(int index)`: 根据位置返回属性值。本例中, 如果 `index` 为 1, 则返回 `"@dimen/general_width"`。
- `public String getAttributeValue(String namespace, String name)`: 返回指定命名空间、指定名称的属性值, 该函数说明 `AttributeSet` 允许给一个 XML Element 的属性中添加多个命名空间的属性值。
- `public int getAttributeNameResource(int index)`: 返回指定位置的属性的 `id` 值。本例中, 如果 `index` 为 1, 则返回 `R.attr.layout_width`, 因为系统为每一个属性 (`attr`) 分配了唯一的 `id` 值。

第三类, 获取特定类型的值。

```
public XXXType getAttributeXXXTypeValue(int index,
XXXType defaultValue);
```

其中 `XXXType` 包括 `int`、`unsigned int`、`boolean` 及 `float` 类型, 使用该方法时, 必须明确知道某个位置 (`index`) 对应的数据类型, 否则会返回错误。而且该方法仅适用于特定的类型, 如果某个属性的值为一个 `style` 类型, 或者为一个 `layout` 类型, 那么返回值都将无效。

`TypedArray` 类又是对 `AttributeSet` 数据类的某种抽象。比如上面例子中, `android:layout_width="@dimen/general_width"`, 如果使用 `AttributeSet` 的方法, 仅能获取 `"@dimen/general_width"` 字符串, 而实际上该字符串对应了一个 `dimen` 类型的数据, 因此, 程序员还需要再去解析 `id` 为 `general_width` 对应的具体的 `dimen` 的值。`TypedArray` 正是免去了在这个过程, 可以将某个 `AttributeSet` 作为构造一个 `TypedArray` 对象的参数, `TypedArray` 将提供更方便的方法直接获取该 `dimen` 值。

从一个 `AttributeSet` 对象构造一个 `TypedArray` 对象的方法如下:

```
TypedArray a = context.obtainStyledAttributes(
    attrs, com.android.internal.R.styleable.XXX, defStyle, 0);
```

函数 `obtainStyledAttributes()` 的第一个参数为一个 `AttributeSet` 对象, 它包含了一个 XML 元素中所定义的所有属性。第二个参数就是前面定义的 `styleable`, `aapt` 会把一个 `styleable` 编译为一个 `int[]` 数组, 该数组所包含的内容正是 `styleable` 中所包含的 `attr` 对应的 `id` 值, 它代表了调用者所感兴趣的属性名称。该函数的内部实现正是通过遍历 `AttributeSet` 中的每一个属性, 找到用户感兴趣的属性, 然后把值和属

性经过重定位，返回一个 TypedArray 对象。

下面分析一下 TypedArray 类的内部接口和重要成员变量。

该类的重要成员变量包括：

- `int[] mData;`
- `private TypedValue mValue = new TypedValue();` TypedValue 类是一个数据类，其意义是为了保存一个属性值，比如 `layout_width`、`textSize`、`textColor` 等，该类中有四个重要成员变量，分别如下。
  - `int type;`：类型包括 `int`、`boolean`、`float`、`String`、`reference` 等。
  - `int data;`：如果 `type` 是一个 `int`、`boolean`、`float` 类型，则 `data` 包含了具体的数据。
  - `int resourceId;`：如果 `type` 是一个 `reference` 类型，则该值为对应的 `resource id`。
  - `CharSequence string;`：如果 `type` 是一个 `string` 类型，则该值为具体的 `string` 值。

TypedArray 中的 `mValue` 起了一个内部缓存的作用。`mData` 则包含了指定 `styleable` 中的所有属性值，`mData` 的长度为 `styleable` 中属性的个数乘以 `AssetManager.STYLE_NUM_ENTRIES`，而系统中该常量的定义为 6。为什么要用 6 个 `int` 值表示一个属性值呢？下面来看这 6 个 `int` 值都包含了什么。

AssetManager 类中关于这 6 个值的定义如下：

```

689      /*package*/ static final int STYLE_NUM_ENTRIES = 6;
690      /*package*/ static final int STYLE_TYPE = 0;
691      /*package*/ static final int STYLE_DATA = 1;
692      /*package*/ static final int STYLE_ASSET_COOKIE = 2;
693      /*package*/ static final int STYLE_RESOURCE_ID = 3;
694      /*package*/ static final int STYLE_CHANGING_CONFIGURATIONS = 4;
695      /*package*/ static final int STYLE_DENSITY = 5;

```

在以上 6 个值中，常用的包含了三个。

- `STYLE_TYPE(0)`：包含了值的类型。
- `STYLE_DATA(1)`：包含了特定的值。
- `STYLE_RESOURCE_ID(3)`：如果 `STYLE_TYPE` 为一个 `reference` 类型的话，则该值对应了相应的 `resource id`。

下面来看看 TypedArray 的接口是如何使用以上几个成员变量的。在应用程序开发时，在 XML 文件中引用某个资源，比如 `android:background="@drawable/bkg"`，该引用对应的元素一般为某个 View/ViewGroup，而 View/ViewGroup 的构造函数中一般会通过函数 `obtainStyledAttributes()` 方法返回一个 TypedArray 对象，然后再调用该对象中的相应 `getDrawable()` 方法。

下面就以常见的 `getString()` 和 `getDrawable()` 为例说明 TypedArray 内部接口的工作原理。`getString()` 的代码如下：

```

1179 public String getString(int index) {
118     index *= AssetManager.STYLE_NUM_ENTRIES;
119     final int[] data = mData;
120     final int type = data[index+AssetManager.STYLE_TYPE];
121     if (type == TypedValue.TYPE_NULL) {
122         return null;
123     } else if (type == TypedValue.TYPE_STRING) {
124         return loadStringValueAt(index).toString();
125     }
126
127     TypedValue v = mValue;
128     if (getValueAt(index, v)) {
129         Log.w(Resources.TAG, "Converting to string: " + v);
130         CharSequence cs = v.coerceToString();
131         return cs != null ? cs.toString() : null;
132     }
133     Log.w(Resources.TAG, "getString of bad type: 0x"
134         + Integer.toHexString(type));
135     return null;
136 }

```

首先,传递进来的 index 必须乘以 6,因为每一个值都占用连续的 6 个 int 值。前面说过每一个 styleable 都将被 aapt 编译为一个 int[] 数组,数组中的内容为 styleable 所包含的每一个属性(attr)对应的 id 值,而在调用 getString() 方法时,其参数 index 的含义是该 attr 在 styleable 中的位置,范围是 0~styleable 的长度。当定义一个 styleable 时,aapt 同时生成了 attr 在 styleable 中的位置,比如 TextView 是一个 styleable,其中包含的 attr 有 textSize,所以 aapt 会自动生成一个 TextView\_textSize 常量。该常量的名称格式是固定的,为“styleable 名称\_attr 名称”,即用一个下划线连接两个名称,其值为 textSize 在 TextView 中的位置,引用时可以使用 R.styleable.TextView\_textSize。

接着从 mData 中取出值的类型,即 index+AssetManager.STYLE\_TYPE 处,然后判断该类型。如果为 TYPE\_NULL,则说明此处无值,直接返回空。如果值的类型为 TYPE\_STRING,则调用 loadStringValueAt() 找到该 String 并返回。

如果类型不为 TYPE\_STRING,那么就需要把 mData 中保存的值强行转换为一个 String 值。转换的步骤是使用 mValue 作为一个临时的缓冲值,而 mValue 是一个 TypedValue 类,该类内部有一个 coerceToString() 方法,该方法会把 mValue 中的值进行转换。其逻辑很简单,就是强制类型或者简单格式的转换,比如当使用 getString() 去读取一个 int 型属性时,其结果返回一个“@”符号加上具体的 int 值。

getValueAt(int index, v) 函数的作用仅仅是把 mData 中偏移为 index 处的 6 个值复制到临时缓冲 mValue 中而已。

下面接着看 loadStringValueAt(index) 方法,该方法源码如下:

```

717 private CharSequence loadStringValueAt(int index) {
718     final int[] data = mData;
719     final int cookie = data[index+AssetManager.STYLE_ASSET_COOKIE];
720     if (cookie < 0) {
721         if (mXml != null) {
722             return mXml.getPooledString(
723                 data[index+AssetManager.STYLE_DATA]);
724         }
725         return null;
726     }
727     //System.out.println("Getting pooled from: " + v);
728     return mResources.mAssets.getPooledString(
729         cookie, data[index+AssetManager.STYLE_DATA]);
730 }

```

该方法中，首先从 `mData` 的 6 个值中找到 `cookie` 值，如果 `cookie` 小于 0 并且 `mXml` 对象存在，则从 `mXml` 对象中得到 `string` 值。`mXml` 对象是专门用于解析二进制 XML 文件的，`mXml` 会在对象内部建立一个 `String` 池，存储 XML 文件中的所有类型为字符串的属性值，并且这个池要用一个索引才能取出，而这个索引正是保存在 `mData` 6 个值中的 `STYLE_DATA`。关于 `mXml` 对象内部的具体实现过程本书不再阐述，因为它已经是一个完全独立的功能类，不与 `Framework` 有内在的联系，有兴趣的读者可自行分析。

接以上分析，如果 `cookie` 大于 0，那么 `cookie` 将作为 `mResource.mAssets` 的内部方法 `getPooledString()` 的参数。其意义类似于 `mXml`，只不过 `mAssets` 对象内部会有多个 `mXml` 对象，所以 `cookie` 要作为 `mAssets` 内部相应的 `mXml` 的索引，而 `STYLE_DATA` 作为字符串的索引，关于从 `Resource` 类中获取资源的流程见下一节。

以上调用过程可总结为如图 15-2 所示。

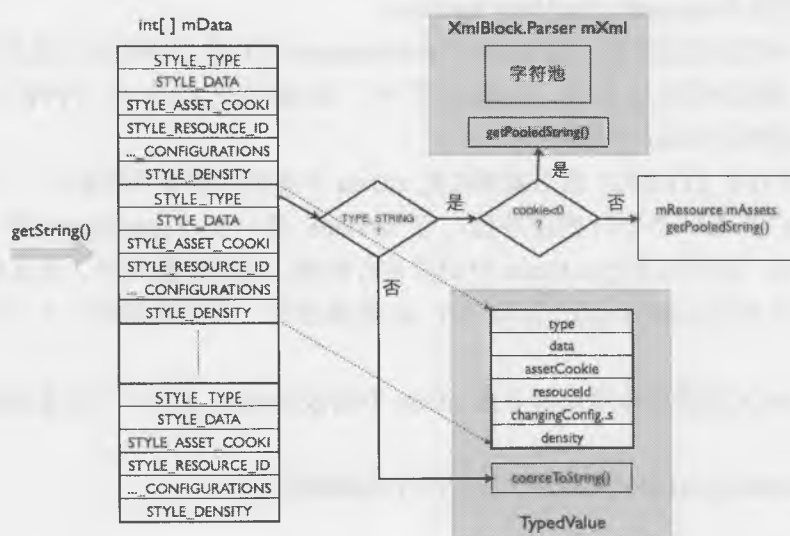


图 15-2 读取资源文件的过程



下面来看 `getDrawable()` 的流程。

`getDrawable()` 的调用比 `getString()` 更为简单，原因是 `XmlBlock.Parser` 中没有缓存 `Drawable` 对象，因此，必须通过 `Resource` 类来获取。其代码如下：

```
public Drawable getDrawable(int index) {
    final TypedValue value = mValue;
    if (getValueAt(index*AssetManager.STYLE_NUM_ENTRIES, value)) {
        return mResources.loadDrawable(value, value.resourceId);
    }
    return null;
}
```

首先调用 `getValueAt()` 把 `mData` 的值赋给 `TypedValue`，然后再把该 `TypedValue` 作为 `loadDrawable()` 的参数。

本节之所以用 `getString()` 和 `getDrawable()` 做例子，原因是这两个方法具有普适性，`TypedArray` 中的其他 `getXXX()` 方法的结构和这两个方法基本类似。

## 15.5 获取 Resources 的过程

获取 `Resources` 有两种方法，第一种是通过 `Context`，第二种是通过 `PackageManager`。

### 15.5.1 通过 Context 获取

在以往的应用程序开发时，大家经常使用 `getResources().getXXX()` 方法获取 XML 文件中定义的资源，比如 `getDrawable()`、`getString()`、`getBoolean()` 等。那么，这些资源到底是如何被访问的呢？其调用的流程是怎样的？这将是本节要讨论的问题。

首先来看 `getResources()` 方法。该方法是 `Context` 类的成员函数，一般是在 `Activity` 对象或者 `Service` 对象中调用的，因为 `Activity` 或者 `Service` 本质上是一个 `Context`，而真正实现 `Context` 接口的是 `ContextImpl` 类。

`ContextImpl` 对象是在 `ActivityThread` 类中创建的，具体的创建逻辑参见第 7 章（理解 `Context`）。所以 `getResources()` 方法实际上是调用到 `ContextImpl` 类中的 `getResources()` 方法。在 `ContextImpl` 类中，该方法仅仅是返回其内部的 `mResources` 变量，而对该变量赋值是在 `init()` 方法中，在创建 `ContextImpl` 对象后，一般会调用 `init()` 方法对 `ContextImpl` 对象的内部变量进行初始化，其中就包括初始化 `mResources` 变量，如以下代码所示。

```
1493-    final void init(ActivityThread.PackageInfo packageInfo,
1494-                   IBinder activityToken, ActivityThread mainThread,
1495-                   Resources container) {
1496-        mPackageInfo = packageInfo;
1497-        mResources = mPackageInfo.getResources(mainThread);
```

在以上代码中，mResources 又是调用 mPackageInfo 的 getResources() 方法进行赋值。而在前文曾说过，一个应用程序中的多个 ContextImpl 对象实际上共享了同一个 PackageInfo 对象，这就意味着，多个 ContextImpl 对象中的 mResources 变量实际上也是同一个 Resources 对象。

PackageInfo 的 getResources() 方法的代码如下：

```
620 public Resources getResources(ActivityThread mainThread) {
621     if (mResources == null) {
622         mResources = mainThread.getTopLevelResources(mResDir, this);
623     }
624     return mResources;
625 }
```

以上代码中，参数 mainThread 指的就是 ActivityThread 对象，一个应用程序中只有一个该对象，其 getTopLevelResource() 方法的语义是得到本应用程序对应的资源对象，其代码请参照源文件，对其执行说明如下：

变量 mActivieResources 对象内部保存了该应用程序所使用到的所有 Resources 对象，其类型为 HashMap<ResourceKey, WeakReference<Resources>>，即用 ResourceKey 映射到 Resources 类，并且这些 Resources 都是以一个弱引用的方式保存，以便在内存紧张时可以释放 Resources 所占用的内存。

该 HashMap 的参数 ResourceKey 仅仅是一个数据类，其构造的方式如下：

```
ResourcesKey key = new ResourcesKey(resDir, compInfo.applicationScale);
```

即用 resDir 和另一个变量构造，resDir 变量的含义是资源文件所在路径，实际指的就是 APK 程序所在的路径，比如可以是：/data/app/ com.haiii.android.client-2.apk，该 apk 会对应/data/dalvik-cache 目录下的：data@app@com.haiii.android.client-2.apk@classes.dex 文件，这两个文件也是一个应用程序安装后自动生成的文件。

所以，如果一个应用程序没有访问该程序以外的资源，那么 mActivieResources 变量中就仅有一个 Resources 对象。这也从侧面说明，mActivieResources 内部可能包含多个 Resources 对象，条件是必须有不同的 ResourceKey，也就是必须有不同的 resDir，这就意味着一个应用程序可以访问另外的 APK 文件，并从中读取其资源。这个结论非常有意义，简单的想法就是，可以设计一种框架，不同的资源对应不同的 APK，然后主应用程序可以读取不同的资源文件，从而有可能实现系统主题的切换，或者叫“换肤”。

接上面，如果 mActivieResources 对象中还没有包含所要的 Resources 对象，那么，就重新建立一个 Resources 对象，建立的方法如以下代码所示：

```
212     AssetManager assets = new AssetManager();
213     if (assets.addAssetPath(resDir) == 0) {
214         return null;
215     }
216
217     //Slog.i(TAG, "Resource: key=" + key + ", display metrics=" + metr
218     DisplayMetrics metrics = getDisplayMetricsLocked(false);
219     r = new Resources(assets, metrics, getConfiguration(), compInfo);
```

这里可以发现，构造 Resources 对象需要先构造一个 AssetManager 对象，然后把该对象作为

Resources 构造函数的参数即可。AssetManager 是一个什么东西呢？

在应用程序开发时，曾经用过该对象，但是构造该对象的方法是使用 Resources 对象的 getResources() 方法，而并没有直接调用构造函数。实际上 getResources() 所获得的 AssetManager 对象正是这里创建的，AssetManager 其实并不只是访问项目中 res/assets 目录下的资源，而是访问 res 下所有的资源。

AssetManager 类中的几个关键函数都是 native 实现的。以上代码中的 addAssetPath(resDir) 非常关键，它为所创建的 AssetManager 对象添加一个资源路径，剩下的事情就由 AssetManager 内部完成了，内部会从指定的路径处获取任何资源。

AssetManager 类的构造函数如下：

```

87~ public AssetManager() {
88     synchronized (this) {
89         if (DEBUG_REFS) {
90             mNumRefs = 0;
91             incRefsLocked(this.hashCode());
92         }
93         init();
94         if (localLOGV) Log.v(TAG, "New asset
95             ensureSystemAssets();
96     }
97 }

```

该构造函数有两个方法非常重要，一个是 init()，另一个是 ensureSystemAssets()，这两个方法都是 native 实现的。

init() 的作用是初始化 AssetManager 的内部环境变量，而初始化过程的一个关键任务是把 Framework 中的资源路径添加到该 AssetManager 对象中，其 native 代码如下，该代码见 android\_util\_AssetManager.cpp 文件。

```

1595 static void android_content_AssetManager_init(JNIEnv* env, jobject clazz)
1596 {
1597     AssetManager* am = new AssetManager();
1598     if (am == NULL) {
1599         doThrow(env, "java/lang/OutOfMemoryError");
1600         return;
1601     }
1602
1603     am->addDefaultAssets();
1604
1605     LOGV("Created AssetManager %p for Java object %p\n", am, clazz);
1606     env->SetIntField(clazz, gAssetManagerOffsets.mObject, (jint)am);
1607 }

```

以上代码首先创建一个 AssetManager 类，这是一个 C++ 类，不是 Java 中的 AssetManager 类。然后调用 am->addDefaultAssets() 方法，该方法的作用就是把 Framework 的资源文件添加到这个 AssetManager 对象的路径中。最后调用 setIntField() 方法把 C++ 创建的 AssetManager 对象的引用保存到 Java 端的 mObject 变量中，该变量可以在 Java 端的 AssetManager 类中找到，其类型为 int，这种用法的原理参见

第 12 节的介绍。

addDefaultAssets()的代码如下, 见 AssetManager.cpp 文件。

```

139 bool AssetManager::addDefaultAssets()
140 {
141     const char* root = getenv("ANDROID_ROOT");
142     LOG_ALWAYS_FATAL_IF(root == NULL, "ANDROID_
143
144     String8 path(root);
145     path.appendPath(kSystemAssets);
146
147     return addAssetPath(path, NULL);
148 }

```

该函数中首先获取 Android 的根目录, getenv()是一个 Linux 系统调用, 用户同样可以使用以下终端命令获取该值, 如下:

```

keyd:tools keyd$ ./adb shell
root@android:/ # echo $ANDROID_ROOT
/system
root@android:/ #

```

获得根目录后, 再与 kSystemAssets 路径进行组合, 该变量的定义如下:

```

50 static const char* kSystemAssets = "framework/framework-res.apk";

```

所以最终获得的路径文件名称为/system/framework/framework-res.apk, 这正是 Framework 对应的资源文件。看到这里你可能会产生一个想法: “是否可以通过更改系统资源的路径达到切换系统主题的目标呢?” 这个后面再讨论。

接前面, 看完 init()方法, 再来看另外一个方法 ensureSystemAssets()。该方法实际上仅在 Framework 启动时调用, 因为 mSystem 是一个 static 变量, 该变量在 Zygote 启动时已经赋值, 以后所有的应用程序运行时该值都不为空, 所以, 该方法如同虚设。

因为应用程序中 Resources 对象内部的 AssetManager 对象除了包含应用程序本身的资源路径外, 还包含了 Framework 的资源路径, 这就是为什么应用程序仅使用本地 Resources 对象就可以访问系统的资源的原因。比如, 可以使用以下代码访问 Framework 中的一个图像资源。

```

Resources res = getResources();
res.getDrawable(android.R.drawable.ic_menu_add);

```

在 AssetManager.cpp 文件中, 当使用 getXXX(int id)访问资源时, 如果 id 值小于 0x1000 0000 时, AssetManager 会认为要访问的是系统资源。因为 aapt 在对系统资源进行编译时, 所有的资源 id 都被编译为小于该值的一个 int 值, 而当访问应用程序对应的资源时, id 值都大于 0x7000 0000。

创建好了 Resources 对象后, 就把该对象缓存到 mActiveResource 变量中, 以便以后继续使用。

这就是访问 Resources 内部的整个流程, 可总结为如图 15-3 所示。

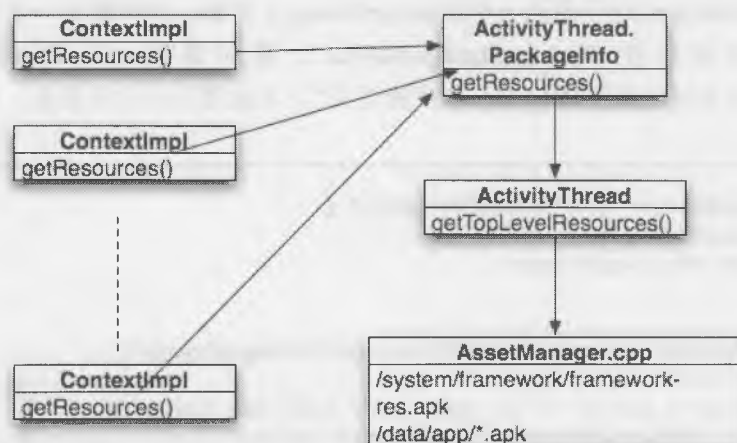


图 15-3 读取 Resources 对象的内部流程

### 15.5.2 通过 PackageManager 获取

该方法主要用于访问其他程序中的资源，其典型的应用就是切换主题，但这种切换一般仅限于一个应用程序内部，而不是整个系统。比如市面上“aHome 桌面”，其工作原理可如图 15-4 所示。

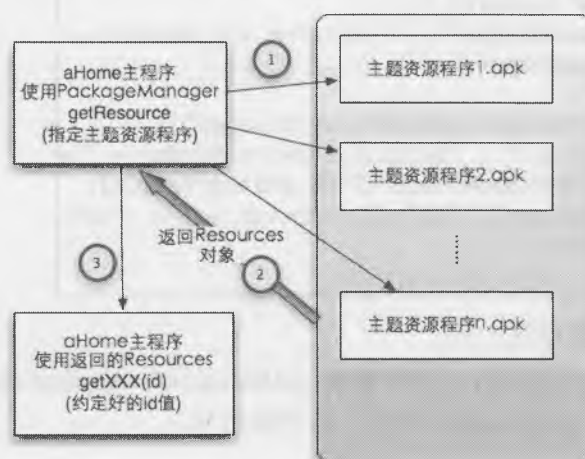


图 15-4 一种可切换主题的桌面程序结构

使用 PackageManager 获取 Resources 对象的代码如下：

```

PackageManager pm = mContext.getPackageManager();
pm.getResourcesForApplication("com.android.haiii.client");
  
```

其中 `getPackageManager()` 用于返回一个 `PackageManager` 对象，该对象是一个本地对象，但是对象内的方法一般是调用远程 `PackageManagerService`。换句话说，`PackageManager` 只是远程 `PackageManagerService` 在本地应用程序的一个“经纪人”，本地要访问远程服务，首先通过该经纪人。其代码如下：

```

236-@Override
237 public PackageManager getPackageManager() {
238     if (mPackageManager != null) {
239         return mPackageManager;
240     }
241
242     IPackageManager pm = ActivityThread.getPackageManager();
243     if (pm != null) {
244         // Doesn't matter if we make more than one instance.
245         return (mPackageManager = new ApplicationPackageManager(this, pm));
246     }
247
248     return null;
249 }

```

`PackageManager` 本身是一个 `abstract` 类，由以上代码可以看出，真正实现这个类的是 `ApplicationPackageManager` 类。该类的构造函数包含了远程服务的一个引用，即 `IPackageManager`，该对象是通过调用 `getPackageManager()` 静态方法获取的，这种获取远程服务的方法和大多数获取远程服务的方法类似，其代码如下：

```

public static IPackageManager getPackageManager() {
    if (sPackageManager != null) {
        //Slog.v("PackageManager", "returning cur default = " + sPackageManager);
        return sPackageManager;
    }
    IBinder b = ServiceManager.getService("package");
    //Slog.v("PackageManager", "default service binder = " + b);
    sPackageManager = IPackageManager.Stub.asInterface(b);
    //Slog.v("PackageManager", "default service = " + sPackageManager);
    return sPackageManager;
}

```

对于这些代码的理解可参照第 5 章。

获得了 `PackageManager` 对象后，接着调用 `getResourceForApplication()` 方法，该方法的代码在 `ContextImpl.ApplicationPackageManager` 子类中，其代码如下：

```

@Override public Resources getResourcesForApplication(
    ApplicationInfo app) throws NameNotFoundException {
    if (app.packageName.equals("system")) {
        return mContext.mMainThread.getSystemContext().getResources();
    }
    Resources r = mContext.mMainThread.getTopLevelResources(
        app.uid == Process.myUid() ? app.sourceDir
        : app.publicSourceDir, mContext.mPackageInfo);
    if (r != null) {
        return r;
    }
    throw new NameNotFoundException("Unable to open " + app.publicSourceDir);
}

```

以上代码内部调用了 `mMainThread.getTopLevelResources()` 方法，这就又回到上一节中通过 `Context` 获取 `Resources` 对象的过程。请注意这里的调用的参数，其含义是：如果目标资源程序和当前程序是同一个 `uid`，那么就使用目标程序的 `sourceDir` 作为路径，否则就使用目标程序的 `publicSourceDir` 目录，该目录可以在目标程序的 `AndroidManifest.xml` 文件中指定。在多数情况下，目标程序和当前程序都不属于同一个 `uid`，因此，多为 `publicSourceDir`，而该值在默认情况下和 `sourceDir` 的值是相同的。

当进入 `mMainThread.getTopLevelResources()` 方法中后，全局 `ActivityThread` 对象就会在 `mActiveResources` 变量中保存一个新的 `Resources` 对象，其键值对应目标应用程序的包名。

以上过程可总结为如图 15-5 所示。

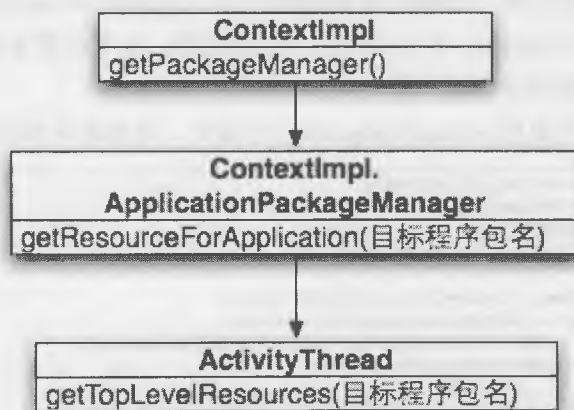


图 15-5 ContextImpl 内部获取 Resources 对象的过程

## 15.6 Framework 资源

了解了 `Resources` 资源的获取流程后，本节再来看 Framework 中的资源是如何获取的，它与普通应



用程序的资源获取过程有何异同，尤其是在代码的实现上有何异同。

在开始之前，请读者先思考这样一个问题：如果系统中不包含任何应用程序，包括系统应用程序和普通应用程序，那么系统还能正常启动吗？启动后的界面会是什么样子？

答案是能够正常启动，启动后的界面上仅有一个系统状态栏。

本节介绍系统资源的加载、读取、添加过程，至于系统资源是如何被编译，即 `appt` 如何编译 `framework/base/core/res/res` 目录下的系统资源，并生成 `framework-res.apk` 的过程请参照本书 18.7 节。

### 15.6.1 加载和读取

系统资源是在 `zygote` 进程启动时被加载的，并且只有当加载了系统资源之后才开始启动其他应用进程，从而实现其他应用进程共享系统资源的目标。该过程的源码请参照 `com/android/internal/os/ZygoteInit.java` 中的 `main()` 函数，该函数的内部过程可分为三步，如图 15-6 所示。

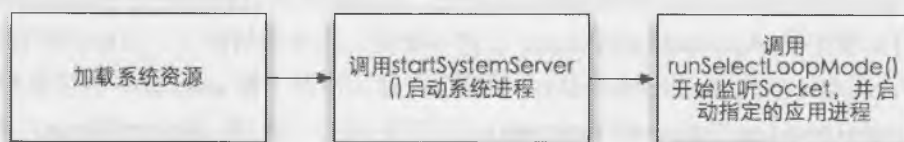


图 15-6 ZygoteInit 的启动过程

启动第一步就是加载系统资源，加载完毕后再调用 `startSystemServer()` 启动系统进程，并最后调用 `runSelectLoopMode()` 开始监听 `Socket`，并启动指定的应用进程，本节主要来介绍加载系统资源的内部过程，关于后面两步的过程请参照本书第 9 章。

加载系统资源具体是通过调用 `preloadResources()` 完成的，该函数内部的关键代码如下：

```

392     mResources = Resources.getSystem();
393     mResources.startPreloading();
394     if (PRELOAD_RESOURCES) {
395         Log.i(TAG, "Preloading resources...");
396
397         long startTime = SystemClock.uptimeMillis();
398         TypedArray ar = mResources.obtainTypedArray(
399             com.android.internal.R.array.preloaded_drawables);
400         int N = preloadDrawables(runtime, ar);
401         Log.i(TAG, "...preloaded " + N + " resources in "
402             + (SystemClock.uptimeMillis() - startTime) + "ms.");
403
404         startTime = SystemClock.uptimeMillis();
405         ar = mResources.obtainTypedArray(
406             com.android.internal.R.array.preloaded_color_state_lists);
407         N = preloadColorStateLists(runtime, ar);
408         Log.i(TAG, "...preloaded " + N + " resources in "
409             + (SystemClock.uptimeMillis() - startTime) + "ms.");
410     }
411     mResources.finishPreloading();
  
```

这段代码的关键有两点。

第一点,创建 Resources 对象 mResources 的方式是调用 Resources.getSystem()函数。虽然 getSystem()函数不是@hide,即应用程序中也可以调用该函数,但一般情况下,应用程序并不需要调用该函数,原因是该函数返回的 Resources 对象仅能访问 Framework 中定义的系统资源。

getSystem()函数内部则调用一个 private 类型的 Resources 构造函数,该函数内部调用 AssetManager 类的静态方法 AssetManager.getSystem()为变量 mAssets 赋值,从而保证了 Resources 类内部 mSystem 变量对应为系统资源, mSystem 这个变量是 static 类型的,笔者觉得这个变量的名称更应该被修改为 sSystem。

从这里也可以看出,zygote 中创建 Resources 对象和普通应用程序的不同,前者使用静态函数 getSystem()创建,后者则使用带有参数的 Resources 构造函数创建,参数间接包含了应用程序资源文件的路径信息。

第二点,有了包含系统资源的 Resources 对象后,接下来则调用两个重要函数 preloadDrawables()和 preloadColorStateLists()装载需要“预装载”的资源。首先来看 preloadDrawables()。

该函数的第二个参数是一个 TypedArray 对象 ar,其来源是 res/values/arrays.xml 中定义的一个 array 数组资源,名称为 preloaded\_drawables,以下是该资源的片段代码:

```
<array name="preloaded_drawables">
  <item>@drawable/sym_def_app_icon</item>
  <item>@drawable/arrow_down_float</item>
```

因此,要想让所有的应用进程共享预装的系统资源,则需要在该文件中声明资源的名称。

接下来看第二个函数 preloadColorStateLists(),该函数的第二个参数也是一个 TypedArray 对象 ar,该对象也是来源于 res/values/arrays.xml 中定义的一个数组资源,名称为 preloaded\_color\_state\_lists,以下是该资源的片段代码:

```
<array name="preloaded_color_state_lists">
  <item>@color/hint_foreground_dark</item>
  <item>@color/hint_foreground_light</item>
```

上面介绍了加载资源的来源,接着,在 Resources 类中相关的资源读取函数中则需要将读取到的资源缓冲起来,以便以后调用,为了这个目的,Resources 类中定义了四个静态变量,如以下代码所示:

```
66 private static final LongSparseArray<Drawable.ConstantState> sPreloadedDrawables
67     = new LongSparseArray<Drawable.ConstantState>();
68 private static final SparseArray<ColorStateList> mPreloadedColorStateLists
69     = new SparseArray<ColorStateList>();
70 private static final LongSparseArray<Drawable.ConstantState> sPreloadedColorDrawables
71     = new LongSparseArray<Drawable.ConstantState>();
72 private static boolean mPreloaded;
```

其中前三个都是类表变量,并且是 static 的,正是这个 static 属性导致 Resources 类在被应用程序进程创建新的 Resources 对象时,保存了 zygote 进程中所预装的资源,这点很关键。

由于 zygote 进程在从 framework-res.apk 中装载资源的实现方式和普通进程基本相同,那么又如何

区别是 `zygote` 还是普通进程呢？这就是变量 `mPreloaded` 的作用，该变量在函数 `startPreloading()` 中被置为 `true`，并在函数 `finishPreloading()` 中被置为 `false`，而 `startPreloading()` 和 `finishPreloading()` 正是在 `ZygoteInit.java` 的 `preloadResources()` 中被调用，这就区别了 `zygote` 调用和普通进程调用。

最后，在 `Resources` 的具体资源读取方法中，会判断 `mPreloaded` 是否为 `true`，如果是，则同时把读取到的资源存贮到三个 `static` 中的相关的列表中，否则，把读取到的资源放到非静态列表 `mPreloadedDrawables`、`mReloadedColorStateLists` 等相关变量中，这些变量的作用范围是调用者所在的进程。

比如，在 `Resources` 内中的 `loadDrawable()` 函数中，就会判断变量 `mPreloading` 是否为 `true`，如以下代码片段所示：

```

1748         if (mPreloading) {
1749             if (isColorDrawable) {
1750                 sPreloadedColorDrawables.put(key, cs);
1751             } else {
1752                 sPreloadedDrawables.put(key, cs);
1753             }
1754         } else {
1755             synchronized (mTmpValue) {
1756                 //Log.i(TAG, "Saving cached drawable @ #
1757                 | //      Integer.toHexString(key.intValue
1758                 //      + " in " + this + ": " + cs);
1759                 if (isColorDrawable) {
1760                     mColorDrawableCache.put(key, new Wea
1761                 } else {
1762                     mDrawableCache.put(key, new WeakRef
1763                 }
1764             }
1765         }

```

## 15.6.2 添加

上一节介绍的 `Framework` 资源加载，仅仅加载的是在 `res/values/arrays.xml` 中预定义的资源值，而事实上 `Framework` 包含了更多的资源，`zygote` 所加载的仅仅是一小部分资源，这些资源在 `Framework` 的设计者看来，被认为是多个应用进程可能会共享的资源。对于那些非“预装载”的系统资源则认为不会被缓冲到静态列表变量中，在这种情况下，多个应用进程如果需要一个非预装载的资源，则会在各自进程中保持一个资源的缓冲。

本节主要介绍如何添加一个系统资源，至于该资源是否为“预装载”，则仅取决于是否将该资源的名称放到 `res/values/arrays.xml` 中。

系统资源按照被公开的方式可分为私有资源 (`private`) 和公开资源 (`public`)，关于私有和公开的详细信息请参照本书第 18.8 小节。总的来讲，只要放到 `res` 目录下的资源都会被 `aapt` 工具编译，而所谓

的私有资源是指仅能在 Framework 内部访问的资源，公开资源是指使用 SDK 的应用程序也能访问的系统资源。

假设要添加一个字符串资源，如以下代码所示：

```
<string name="cus_str">Customer string</string>
```

那么，可以直接把这段字符串添加到 res/values/string.xml 中，然后重新编译 Framework，编译完毕后，在 com.android.internal.R.java 文件中，就会包含该字符串的声明，然后在 Framework 的源码中就可以直接引用该资源，如以下代码所示：

```
TextView tv;
tv.setText(com.android.internal.R.string.cus_str);
```

这里请注意，cus\_str 被存放到了 com.android.internal.R 文件中，而不是 android.R 文件中，这点很关键，因为这正是私有资源和公开资源的区别所在，使用 SDK 开发普通应用程序时，当要引用系统资源时，只能引用 android.R 文件，该文件的内容可认为是 com.android.internal.R 的一个子集。如前所述，res 目录下的资源总是会被 aapt 编译到一个 R.java 文件中，这个文件就是 com.android.internal.R 文件，而 android.R 文件则是来源于 res/values/public.xml 中定义的资源。

同学们可以查看 frameworks/base/core/res/values/public.xml 文件的具体内容，它为所有需要公开到 SDK 中的资源进行 id 值的预先定义，这些 id 值在不同的 Android 版本中总会保持一致，从而保证不同 Android 版本的资源兼容性。

因此，如果想让前面的 cus\_str 字符串资源也公开到 SDK 中，则需要在 public.xml 中声明该字符串，声明时需注意，该文件比较长，其内容排序方式如图 15-7 所示。

version 1	.....	version n
type = attr id=0x01010000		type = attr id=...
type = id id=0x01020000		type = id d=...
type = style id=0x01030000		type = style d=...
type = string id=0x01040000		type = string d=...
type = dimen 0x01050000		type = dimen d=...
type = color 0x01060000		type = color d=...
type = array 0x01070000		type = array d=...
type = drawable 0x01080000		type = drawable d=...
type = layout 0x01090000		type = layout d=...
type = anim 0x010a0000		type = anim d=...

图 15-7 public.xml 文件内容的排序方式

为新资源指定 id 值时，必须考虑两个问题：

第一，不能与已有的 id 值冲突。

第二，尽量避免与未来的 id 值冲突。由于 Android 处于不同升级的开发中，新版本可能会引入新的资源，其可能被分配到的 id 会在原来的基础上逐渐递增，因此，当添加新的资源时，需要考虑到这一点。

本例中，就可以给 public.xml 中添加以下代码：

```
<!-- =====
Resources By myself.
===== -->
<public type="string" name="cus_str" id="0x0104f000" />
```

id 值的含义是，01 代表这是一个 Framework 资源，04 代表这是一个 String 类型的资源，f000 是该资源的编号，之所以从 f 开始，是因为 Framework 内部的资源是从 0 开始的，防止以后递增时与我们自定义的资源值冲突。

### 15.6.3 实现真正主题切换的两种思路

在之前的讨论中，同学们已经了解可以为某个 Activity 或者 Application 类在 AndroidManifest.xml 文件中指定所使用的 Theme，Theme 本质上仅仅是一个 style 而已，即一组属性值而已。

前面所讲的 Theme 有两个局限性：

第一，Theme 的作用域仅仅是指定的 Activity 或者 Application 而已，而无法影响其他应用程序的 Theme。

第二，Theme 中所包含的属性的作用仅仅是影响界面上的一少部分元素而已。

那么，什么才是“真正的主题”，本节先来为此做一个定义。所谓真正的主题是指，所有影响界面效果元素的集合。比如，状态栏的背景色、系统通知图标的样式，以及所有界面控件的显示元素，比如 TextView、Button 等的背景色，字体大小，等等，换句话说，真正的主题应该是 Framework 中所有系统资源的集合。而所谓的“主题切换”是指提供另一套系统资源，进一步讲，“切换过程”必须是可逆的，并且不影响当前系统运行。

因此，从用户角度来看，“主题切换”的场景可描述为：给用户一些主题 APK 程序包，然后用户可以在系统设置中应用不同的主题程序，应用时，系统不需要重启，应用后，当前系统内部所调用的 Framework 资源就被替换为了新的主题资源。

前面小节介绍了系统资源的添加以及访问过程，要实现上面所述的用户场景应该也不是件难事，笔者设计了两种主题切换的思路，与同学们一起讨论，当然，可能还存在更好的思路。

第一种思路是在 C++ 的层面修改源码，并使用 JNI 在 Java 层提供一个 API 接口。具体来讲，C++ 层面的 AssetManager 在初始化时，是将 framework-res.apk 作为资源搜索的路径，因此，可以在 C++ 层面增加一个函数，只要替换 AssetManager 中的路径即可，并把这个函数以 JNI 的方式封装到 Java 层面，

然后在用户界面上设计一个“设置”页面，用户可以选择不同的主题文件，底层只需要使用新的主题文件路径替换 `framework-res.apk` 路径即可。

使用这种思路需要注意以下问题。

第一，由于 `zygote` 进程中使用静态变量保存预装的系统资源，当新的资源被应用前必须清除 `Resources` 类中的静态资源，这就导致所有其他的应用程序都需要重新缓冲系统资源，这就等于 `zygote` 没有预装任何资源，这一定程度上浪费了系统内容。

第二，由于即将使用新的系统资源，因此，需要将每一个应用进程中 `Resources` 对象内部的非静态资源缓存清楚，当然，这个也是可以实现的。

第三，不能在 Eclipse 环境下编译资源生成 APK 资源包，而必须在命令行下使用 `aapt` 手工编译，原因是 Eclipse 环境下默认编译资源时，资源的 id 都会被赋值为 `0x07` 开头的值，而系统资源必须以 `0x01` 开头。

第四，在新的主题资源包中，必须使用原始资源文件中的 `public.xml` 文件，这样可以保证资源 id 的一致性。

第五，新的资源包中的资源名称和数目必须和原始的内容相同，然而一般的主题资源可能不会包含全部的资源文件，因此，为了保证这一点，可以做一个复制脚本，仅当新的资源文件不存在时才把原始的资源文件复制过来，从而保证每一个资源文件都存在。

另一种思路是不在 C++ 层面做任何修改，而是仅仅修改 Java 层面，基本原理是基于系统资源和非系统资源的 id 值差别。前面讲过，系统资源的 id 值都是以 `0x01` 开头，而一般的应用程序的资源值是以 `0x07` 开头，基于这个特性，可以修改 `Resources` 源码，修改的目标是，当要获取的资源 id 是 `0x01` 开头时，则从另外的主题资源文件中利用 `PackageManager` 重新获取，这就在系统资源和真正的主题资源文件中建立了一一对应关系，达到替换系统主题的目的。

使用这种思路需要注意以下问题。

第一，主题资源文件可以使用 Eclipse 进行编译，就像编译一个普通的 APK 一样，并且自定义的系统资源可以只是一小部分，而不必是全部。当 `Resources` 在读取主题资源文件时，如果发现没有该资源，应该继续从原有的系统资源中读取。这就要求在修改 `Resources` 源码时必须区分是第一个读取系统资源，还是因为没有找到而重新读取系统资源，处理不好的话会进入死循环。

第二，当切换主题资源时，必须清空 `Resources` 类中的静态资源缓冲，这就导致多个应用之间无法共享预装的系统资源，这个问题理论上从 Java 层很难做到，笔者也没有什么好办法。一种折中的做法是，在系统重启后，能够在 `zygote` 进程中装载新的主题资源，重新达到多进程共享系统资源的目的。

第三，制作主题资源 APK 时，资源名称应该和原始的资源名称一致，id 值可以一致，也可以不一致。如果一致的话，则需要在主题资源包下使用 `public.xml`，并且手工定义 id 值，这种做法的好处是，在修改 `Resources` 源码时，可以直接将系统资源前面的 `0x01` 替换为 `0x07`，然后访问主题资源。如果不使用 `public.xml`，则在修改 `Resources` 源码时，必须先通过 id 获取原始的资源名称，则通过资源名称在新的资源文件中获取 id，然后再使用新的 id 从资源文件中获取资源，这种资源转换关系如图 15-8 所示。

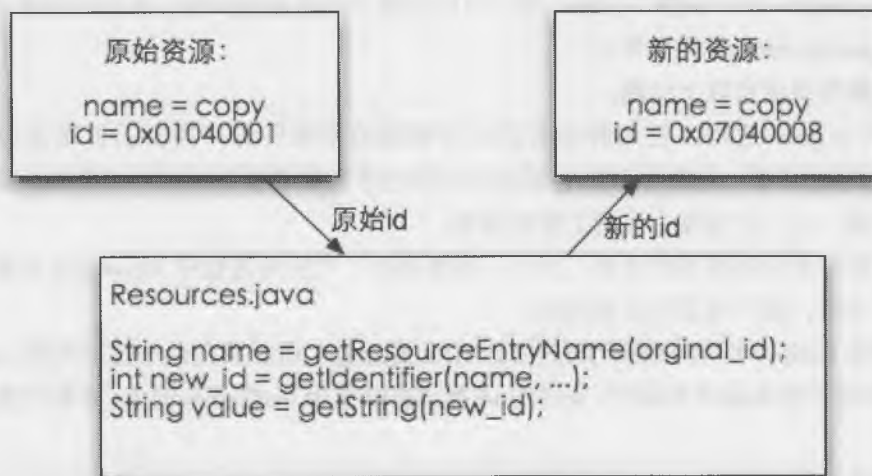


图 15-8 根据原始 id 从新资源文件中获得 String 资源

以上两种思路在实现时,会有一些细节问题,比如,简单清空 `Resources` 中的资源缓存未必就会使得新的资源立即生效,原因是,有些资源值并没有被缓存,比如一些 `dimen`、`color`、`String` 等资源,这就导致那些还没有被 `destroy` 掉的 `Activity` 的界面不会立即应用新的主题资源,解决的办法就是在应用新的资源后调用 `AmS` 销毁所有 `Activity`,以便那些 `Activity` 重新被打开后再次读取资源,这才能使新资源全部生效,包括那些有缓存的或者无缓存的。





## 第 16 章 程序包管理 (Package Manager Service)

程序包管理主要包含三部分内容。

- 提供一个能够根据 `intent` 匹配到具体的 `Activity`、`Provider`、`Service`。即当应用程序调用 `startActivity(intent)` 时，能够把参数中指定的 `intent` 转换成一个具体的包含了程序包名称及具体 `Component` 名称的信息，以便 Java 类加载器加载具体的 `Component`。
- 进行权限检查。即当应用程序调用某个需要一定权限的函数调用时，系统能够判断调用者是否具备该权限，从而保证系统的安全。
- 提供安装、删除应用程序的接口。

完成以上三部分功能的代码主要在 `PackageManagerService` 类中，本章以后简称 `PmS`，该类的代码长度超过了 1 万行，但是其内部逻辑的复杂程度远远小于 `View` 系统，大部分代码仅仅是执行一些事物性的处理，比如解析 XML 文件，进行 Hash 表的添加、查找等。因此本书仅仅介绍一些重要的逻辑框架，以及重要的函数的作用，而对于一些功能明确、逻辑简单的函数不作详细分析，有兴趣的读者可自行研究。

16.1

### 包管理概述

Framework 中的包管理整体框架如图 16-1 所示。

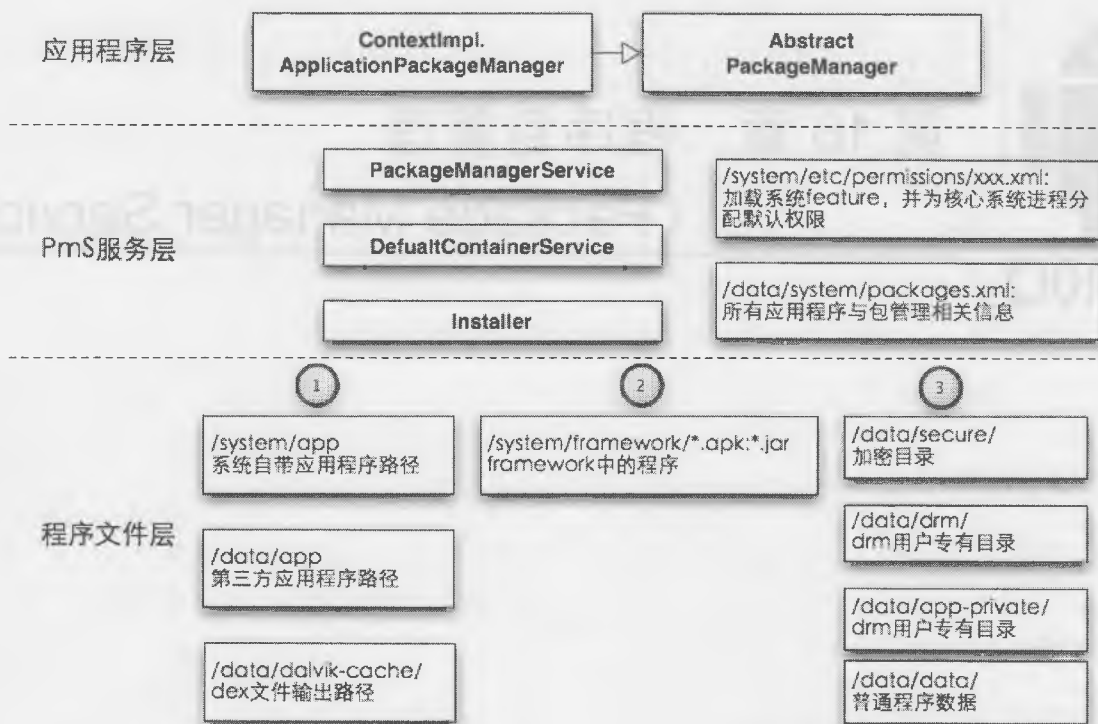


图 16-1 包管理服务的程序结构

该框架可以分为三层，分别是应用程序层、PmS 服务层及数据文件层。

### 1. 应用程序层

应用程序需要使用包管理服务时，调用 ContextImpl 类的 getPackageManager() 函数返回一个 PackageManager 对象，然后调用该对象所提供的各种 API 接口。读者们可能会问，为什么获得 PmS 服务不像其他服务那样通过调用 getSystemService(“package”) 获得呢？事实上，getPackageManager() 类内部获取的过程和 getSystemService() 的过程基本相似，都是从 ServiceManager 获得指定名称的 IBinder 对象，此处名称是“package”。之所以还单独提供一个 getPackageManager() 函数，或许仅仅是为了方便，无本质区别。

### 2. PmS 服务层

和 AmS、WmS 等其他系统服务一样，包管理服务运行于 SystemServer 进程。PmS 服务运行时，使用了两个目录下的 XML 文件保存相关的包管理信息。

第一个目录是“/system/etc/permissions”，如其名称所指，该目录下的所有 XML 文件用于 permission 的管理，具体包含两件事情。第一个是定义系统中都包含了哪些 feature，应用程序可以在 AndroidManifest.xml 中使用 <use-feature> 标签声明程序都需要哪些 feature，比如：

```
<use-feature android:name="android.hardware.wifi" android:required="true" />
```

该标签的作用仅仅是“声明”或者“建议”设备是否具备指定的 feature，而不会影响程序的安装；required 的作用用于表明，如果没有该 feature 的话，该程序不能被运行。

设备都包含哪些 feature 就是在此目录下声明的，因此，如果用户给系统添加了 GPS 或者蓝牙功能，则必须在此目录下声明相关的 XML 文件，以便 Framework 知悉。

该目录下另外还有一个 platform.xml 文件，该文件中为一些特别的 uid 和 gid 分配了一些默认的权限。给 uid 分配权限使用 <assign-permission> 标签，给 gid 分配权限使用 <permission> 标签，举例如下：

```
<assign-permission name="android.permission.WRITE_EXTERNAL_STORAGE"
uid="shell" />
<permission name="android.permission.BLUETOOTH_ADMIN" >
  <group gid="net_bt_admin" />
</permission>
```

从这点上看，这两个标签的风格有点不一致。

第二个目录是“/data/system/packages.xml”，该文件保存了所有安装程序的基本包信息，有点像系统的“注册表”，比如程序的包名称是什么，安装包路径在哪里，程序都使用了哪些系统权限，等等。该文件的详细信息见后面小节。

PmS 在启动时，会从这两个目录中解析相关的 XML 文件，从而建立一个庞大的包信息树，应用程序可以间接从这个信息树中查询所有所需的程序包信息。

除了 PmS 服务外，还有两个辅助系统服务用于程序安装。一个是 DefaultContainerService，该服务主要用于把安装程序复制到程序目录中；另一个是 Installer 服务，该服务实际上并不是一个 Binder，而是一个 Socket 客户端，PmS 直接和该 Socket 客户端交互。Socket 的服务端主要完成程序文件的解压工作及数据目录创建，比如从 APK 文件中提取出 dex 文件，删除 dalvik-cache 目录下的 dex 文件，创建程序专属的数据目录等。

### 3. 数据文件层

就像所有操作系统一样，Android 中的程序也由相关的程序文件组成，这些程序文件可以分为三个部分。

第一部分，程序文件。所有的系统程序保存在 /system/app 目录下，所有的第三方应用程序保存在 /data/app 目录下。对于非系统程序，在安装前，程序文件可以保存在任意的地方，但安装后，PmS 会把 APK 文件存放到 /data/app 目录下。它与原始的 APK 文件的唯一区别是文件的名称不同，原始文件可以任意命名，而该目录下的文件名称是以包名进行命名，并自动增加一个“-x”后缀，比如 com.android.haiiii.debugjar-1.apk。当同样一个程序第二次安装时，后面的数字 1 会变成数字 2，而当第三次再安装时，又会变成数字 1，有点像“乒乓”机制。

/data/dalvik-cache 目录保存了程序中执行代码。一个 APK 实际上是一个 Jar 压缩类型的文件，压缩包中包含了各种资源文件、资源索引文件、AndroidManifest 文件及程序文件，当应用程序运行前，PmS 会从 APK 文件中提取出代码文件，也就是所谓的 dex 文件，并将该文件存储在该目录下，以便以后能

够快速运行该程序。比如：

```
data@app@com.android.haiii.debugjar-1.apk@classes.dex。
```

第二部分，framework 库文件。这些库文件存在于/system/framework 目录下，库文件类型是 APK 或者 Jar，系统开机后，dalvik 虚拟机会加载这些库文件，而在 PmS 启动时，如果这些 Jar 或者 APK 文件还没有被转换为 dex 文件，则 PmS 会将这些库文件转换为 dex 文件，并保存到/data/dalvik-cache 目录下。

第三部分，应用程序所使用的数据文件。应用程序可以使用三种数据保存方式，分别为参数存储、数据库存储、文件存储。这三种存储方式对应的数据文件一般都保存到/data/data/xxx/目录下，xxx 代表程序的包名。除了/data/data 目录外，在 Android 2.2 版本后，系统还提供了额外三个目录，其中/data/secure 用于保存加密数据，Framework 将提供一种加密的类接口，所有的加密文件将存放到该目录下。/data/drm 目录主要用于存放包含有数字版权的数据文件，同样，Framework 层将提供一种与 Drm 相关的类接口用于间接访问该目录中的文件。/data/app-private 目录目前和/data/drm 的作用相同，Framework 中似乎没有使用/data/drm，而临时使用/data/app-private 代替。

## 16.2 packages.xml 文件格式

packages.xml 在/data/system 目录下，该文件记录了系统中所有应用程序的包管理相关信息，PmS 将根据这些信息进行包管理的各种操作。总的来讲，该文件中包含的信息类型如表 16-1 所示。

表 16-1 packages.xml 文件不同标签的意义

标签名称	所包含的值举例
last-platform-version	internal="9", external="9"
permission-trees	暂不使用
<permissions> <item /> </permissions>	<item name="android.permission.PERSISTENT_ACTIVITY" package="android" protection="1" />
<package> <sigs> <cert /> </sigs> <perms> <item /> </perms> </package>	<package name="com.android.soundrecorder" codePath="/system/app/SoundRecorder.apk" nativeLibraryPath="/data/data/com.android.soundrecorder/lib" flags="1" ft="12dabe97198" it="12d805b2cd8" ut="12dabe97198" version="9" userId="10006"> <sigs count="1"> <cert index="0" key="3082... " /> </sigs> </package>

(续表)

标签名称	所包含的值举例
disabled-components	<item name="com.android.email.service.MailService"/>
preferred-activities	偏好设置, 比如当一个 intent 对应了两个 Activity 时, 系统会弹出一个对话框提示用户要执行哪个 Activity, 并且提供一个复选框, 用户可以勾选该复选框, 从而使得以后每次都执行相同的 Activity, 而这个信息就保存在该标签中
<shared-user> <sigs> <cert /> </sigs> <perms> <item /> </perms> </shared-user>	共享 id 本身只是一个名称而已, 这个名称也必须对应一个实实在在的, 具体的 Linux 用户 id。该标签正是描述一个共享 id 所对应的签名信息和权限信息, 和 package 标签十分相似, 所不同的是该标签中不包含 codePath 等属性
cleaning-package	程序已经删除, 但是对应的程序数据目录还没有被删除, 因为默认卸载时并不会清除数据目录
update-package	

下面分别介绍不同的标签。

### 16.2.1 last-platform-version 标签

该标签记录了系统最后一次的版本号, 一般和相应的 SDK 版本号相同, 比如 foryo 对应的 8。标签中包含两个属性, 一个是 internal, 另一个是 external, internal 是内部存储区上的程序被更新前的系统版本号, external 是外部存储区程序更新前的系统版本号, 这两个号一般相同。

所谓的“程序更新”, 不是指应用程序有了新版本的更新, 而是指系统升级后, 需要重新为所有已安装的程序设置访问权限, 即重新建立包管理信息所需的 XML 文件。

### 16.2.2 permissions 标签

该标签保存了系统中所有的权限列表, 包含 Framework 定义的权限以及应用程序自定义的权限, Framework 中定义权限是在 android/framework/base/core/res/rs/AndroidManifest.xml 中。每个 permission 都由一个 item 标签标识, item 标签中分别包含三个属性。

- name: 权限名称。系统权限一般都以 android.permission. 开头, 应用程序中自定义权限一般都以所属的包名开头, 权限名称必须全局唯一。
- package: 权限所在的包名, Framework 对应的包名为 android。

- **protection**: 保护级别。一共有 4 种保护级别, 分别为普通的、危险的、签名的、签名或系统的, 关于四种保护级别的作用参见笔者的另一本书《Android 应用程序开发》。

### 16.2.3 cert 标签

**cert** 代表 **certification**, 即“证书”的意思, 一个应用程序可以包含多个证书。**cert** 包含两个属性, 分别是 **index** 和 **key**。

- **index**: 在 **PmS** 内部保存了所有的证书, 每个证书对应一个索引, **index** 表示的就是索引。
- **key**: 该属性不是必需的, 如果 **index** 对应的证书已经存在, 则该 **key** 子标签可以没有, 而如果 **index** 是首次出现, 则 **key** 的值必须指定, 其值为 2390 个 16 进制值, 来源于对 **APK** 进行签名的证书文件。
- **cert** 标签揭示了一个问题, 即通过修改 **packages.xml** 文件中 **package** 标签中的 **cert** 子标签值, 可以任意改变应用程序的签名, 从而使得应用程序具备系统程序的能力。

### 16.2.4 sigs 标签

**sigs** 代表 **signature**, 即“签名”的意思, 一个应用程序只能有一个签名, 但一个签名中可以包含多个证书。签名标签只有一个属性 **count**, 即该签名中包含多少个证书, **sigs** 标签中必须包含 **count** 个 **cert** 子标签, 用于表示具体的证书。

### 16.2.5 perms 标签

**perms** 代表 **permissions**, 即权限的意思, 一个应用程序可以申请多个权限, **perms** 标签包含了所有这些权限列表。**perms** 标签中可以包含 **item** 子标签, 用于表示每一个具体的权限。注意区分 **perms** 标签和 **permissions** 标签, **perms** 标签用于表示一个具体应用程序所使用的权限列表, 该标签一般被包含在 **package** 标签或者 **shared-user** 标签中, 而 **permissions** 标签却是表示系统中所有定义的权限列表, 该列表是全局的。

### 16.2.6 package 标签

一个 **package** 标签中包含了一个应用程序包的相关信息, **package** 标签的属性包括如下几种。

- **name**: 程序包名称, 比如 **com.haiiii.android.Foo**。
- **codePath**: 程序包所在路径, 比如 **/system/app/Mms.apk**、**/data/app/com.haiiii.android.Foo-1.apk**。

- **nativeLibraryPath**: 该程序所使用的 native 库文件路径。在一般情况下 native 库文件会被安装到程序数据文件路径下的 lib 子目录中, 比如/data/data/com.haiii.android.Foo/lib。
- **flags**: 用于标识应用程序类型, 这些类型在 ApplicationInfo.java 中有定义, 包括 FLAG\_SYSTEM、FLAG\_DEBUGGABLE、FLAG\_PERSISTENT 等。
- **It**、**ut**: 分别代表 “首次安装的时间 install time” 和 “最后升级时间 update time”。
- **ft**、**ts**: 这两个标签的含义是相同的, 代表最后一次修改该记录的时间, 一般情况下, 这个时间等于 ut 标签值。
- **installStatus** 属性: 该属性仅在 packages-backup.xml 文件中存在, 其值一般为 false, 当值为 true 时, 该属性一般不会再写到该文件中。

package 标签中可以包含 sigs 和 perms 子标签, 分别表示该应用程序所使用的签名信息及权限信息, 比如:

```
<package name="com.android.haiii.debugjar" codePath="/data/app/com.an
<sigs count="1">
<cert index="4" />
</sigs>
<perms>
<item name="android.permission.READ_USER_DICTIONARY" />
</perms>
</package>
```

- **userId** 和 **sharedUserId**: 前者代表应用程序对应的 Linux 用户 id 值, 后者代表该应用程序共享的 Linux 用户 id, 这两个属性是互斥的, 即只能有一个属性。当存在 sharedUserId 时, 该应用程序将以共享 id 的身份运行, 但这并不是说所有共享 id 的程序运行在同一个进程中。
- **installer** 属性: 安装器的名称, 一般是指应用程序调用 PackageManager 的 installPackage() 函数时, 参数 installerPackageName 的值。

### 16.2.7 shared-user 标签

系统为每一个应用程序分配一个 Linux 用户 id, 一些 native 进程的用户 id 从 1000 开始, 比如 shell 进程、log 进程、Phone 进程等, Java 应用程序对应的用户 id 从 10000 开始。从这个角度来讲, 一个应用程序实际上就是一个用户 id, 当然, 应用程序申请使用共享用户 id 例外。

shared-user 标签定义了共享用户 id 对应的签名和权限, 它和 package 标签的含义本质是相同的。shared-user 的属性包括 name 和 userId, 其意义和 package 标签的属性相同, 包含的子标签也和 package 相同, 包括 sigs 和 perms 标签。



## 16.3 包管理服务的启动过程

PmS 本身是一个 Service，它与 WmS、AmS 等重要系统服务运行在同一个进程——系统进程中。当 SystemServer 服务启动时，其初始化函数中会启动各种具体的服务进程，包括 AmS、WmS 等，其中也包含 PackageManagerService 服务。

PmS 服务是从其静态 main() 函数中创建的，如以下代码所示：

```
693 public static final IPackageManager main(Context context, boolean factoryTest) {
694     PackageManagerService m = new PackageManagerService(context, factoryTest);
695     ServiceManager.addService("package", m);
696     return m;
697 }
```

main() 函数中创建一个 PmS 的实例，然后把该服务添加到 ServiceManager 中。应用程序可以使用 Context 类的 getSystemService() 调用获得 PmS 服务的 Binder 接口，并调用 PmS 所提供的相应服务。

PmS 的启动过程实际上就是该类的构造函数所包含的各种初始化过程。在介绍构造函数内部执行流程前，首先需要了解一下各主要功能类之间的关系，因为启动过程实际上是读取相关 XML 文件中的信息，并把这些信息存放到相关的类成员变量之中。

### 16.3.1 各主要功能类的关系

包管理服务相关的主要类作为 PmS 的内部类被定义，其关系参见附图 9 所示。由下往上看，PmS 的内部类 Settings 基本上包含了包管理所需的全部信息，该类主要包含几类变量：

#### 1. 包属性信息

- File mSettingsFilename;：配置文件名称，指的就是 packages.xml 文件。
- File mBackupSettingFilename;：配置文件可以有一个 backup 文件，该 backUp 文件用于系统意外关机后和原始配置文件进行对比，以检查系统的完整性。
- File mPackageListFilename;：指的就是 packages.list 文件，保存了所有应用程序列表，该文件中每一行对应一个应用程序，比如 com.android.haiii.debugjar 10032 1 /data/data/com.android.haiii.debugjar。

在该记录中，第一项表示应用程序包名称；第二项代表该应用程序的 Linux 用户 id，如果应用程序使用的是共享 id，则指的是共享 id 对应的用户 id；第三项数字 1 代表应用程序可以被 debug，0 代表不能被 debug；第四项代表应用程序数据文件的目录。

- Hashmap<String, PackageSetting> mPackages：在 PmS 启动后，该变量将被填充为包管理信息，这些信息来源于 packages.xml。注意该变量和 PmS 中定义的 mPackages 变量的区别，前者是从 packages.xml 读取的记录信息，而后者则是直接扫描程序目录下所有 APK 文件而生成的信息，并且前者的类型是 PackageSetting，而后者的类型是 PackageParser.Package。

- `HashMap<String, PackageSetting> mDisabledSysPackages`: 该变量将保存那些没有通过标准卸载方法删除的程序列表。如果使用标准卸载方法, PmS 在卸载程序后, 会清除该程序在 `Packages.xml` 中对应的记录, 而如果直接通过 `adb` 或者其他文件操作删除 APK 文件, 那么 `packages.xml` 中对应的记录将不会被自动清除。每次系统启动时, PmS 会检查 `packages.xml` 文件, 并检查相应程序目录下的文件, 从而判断是否被意外删除。如果删除, 则把相应的包信息存放到 `mDisabledSysPackages` 列表中, 该变量名称中的 `Sys` 并不是指系统程序, 而是指所有应用程序。
- `int mInternalSdkPlatform`: 保存的就是 `packages.xml` 中 `last-platform-version` 标签中的值。
- `int mExternalSdkPlatform`: 与上同。

## 2. 用户 Id 相关信息

- `HashMap<String, SharedUserSetting> mSharedUsers`: 该变量保存了所有共享 id 的信息, 来源于 `packages.xml` 中的 `<shared-user>` 标签。
- `ArrayList<Object> mUserIds`: 所用用户 id。
- `SparseArray<Object> mOtherUserIds`: 目前暂未使用。
- `ArrayList<PendingPackage> mPendingPackages`: 当 PmS 解析 `packages.xml` 文件时, 如果发现某个 `package` 标签中使用的是 `sharedUserId`, 则暂时把该 `package` 添加到 `mPendingPackages` 列表中, 直到最后解析完毕 `shared-user` 标签后, 再完善原来的 `package` 标签中的相关信息。

## 3. 权限管理相关信息

`ArrayList<Signature> mPastSignatures`: 保存了所有的签名。

- `HashMap<String, BasePermission> mPermissions`: 保存了所有的权限, 该 Map 是从 `permission` 名称到 `permission` 信息的一个映射。
- `HashMap<String, BasePermission> mPermissionTrees`: 对应 `packages.xml` 文件中的 `permission-tree` 标签, 目前一直为空, 没有被使用。

## 4. 删除信息

`ArrayList<String> mPackagesToBeCleaned`: 保存的是 `packages.xml` 中的 `cleaning-package` 标签中包含的 `package` 列表, 该列表来源于那些已经被卸载的应用程序, 但其所包含的数据目录由于保存在外部存储区而没有被删除。因为 PmS 卸载程序时, 如果该程序的数据保存在外部存储空间中, 则其数据目录默认不会被删除。

在 PmS 类中, 包含以下重要变量。

- `HashMap<String, PackageParser.Package> mPackages`: 保存所有程序包信息, 来源于扫描程序目录下所有程序文件。
- `final Settings mSettings`: 该变量正是上面所介绍的 PmS 内部类 `Settings`。

- `HashMap<String, FeatureInfo> mAvailableFeatures`: 一个 feature 本质上只是一段字符串描述, 比如 “android.hardware.wifi”。
- `int[] mGlobalGids`: 系统中所有 Linux 用户 id。
- `SparseArray<HashSert<>> mSystemPermissions`: 系统中所有的权限名称。
- `HashMap<String packageName, String path> mSharedLibraries`: 系统所依赖的共享 Java 库, 其值来源于 platform.xml 中 library 标签中定义的值, 比如:

```
<library name="android.test.runner"
        file="/system/framework/android.test.runner.jar" />
<library name="javax.obex"
        file="/system/framework/javax.obex.jar"/>
```

- `ActivityIntentResolver mActivities`;
- `ActivityIntentResolver mReceivers`;
- `ServiceIntentResolver mServices`。

以上三个变量用于进行 Intent-filter 的匹配, 并分别匹配到 Activity、Receiver、Service 对象。在 PmS 初始化时会遍历程序目录下的全部的程序, 并从其包含的 AndroidManifest.xml 文件中提取出所有的 intent-filter 数据, 并将其保存到以上三个变量中。系统运行时, 应用程序调用 PackageManager 的 queryIntentXXX() 函数时, 其内部正是通过以上三个变量查询相关的目标对象信息的。

### 16.3.2 PmS 主体启动过程

主体启动过程就是指 PmS 构造函数内部的启动流程, 参见附图 10 所示。。

- ① 创建一个 PmS.Settings 对象, 在该对象的构造函数中, 将给成员变量静态赋值, 这些变量包括:
  - `mSettingsFilename`, 赋值为 /data/system/packages.xml。
  - `mBackupSettingsFileName`, 赋值为 /data/system/pckages-backup.xml。
  - `mPackageListFileName`, 赋值为 /data/system/packages.list。
- ② 调用 `mSettings.addSharedUserLP()` 添加四个共享用户 id。
- ③ 创建一个 Installer 对象, 该对象将辅助程序的安装, 比如完成从 APK 文件提取出 dex 文件, 删除 dex 文件等操作。Installer 对象内部其实包含一个 LocalSocket 对象, 所有 Installer 提供的 API 接口内部其实会通过该 LocalSocket 发送一系列参数值给远程的 SocketServer 对象。创建好 Installer 对象后, 调用一次 ping() 函数, 确保 Socket 的服务端准备就绪。
- ④ 给以下几个数据文件路径静态赋值, 包括:
  - `mAppDataDir`, 代表程序的数据目录, 其值为 /data/data。
  - `mSecureAppDataDir`, 所谓的解密数据区, 目前该区域并没有被使用, 其值为 /data/secure/data。

- `mDrmAppPrivateInstallDir`, 所谓的 DRM 数据区, 其值为 `/data/app-private`, 该目录暂时也未使用。

以上后面两个目录的设计思想是, 从 Framework 层面分别提供 Secure 和 Dm 相关的类接口, 前者将提供加密的数据访问, 后者提供支持数字版权管理 (Digital Right Management) 的文件存储, 但这只是设计思路, 当前并未真正实现。

⑤ 调用 `readPermission()` 函数, 从 `/system/etc/permissions` 目录下读取全部 XML 文件, 这些文件主要定义了两部分系统属性。第一是系统中所有的 feature, 第二是为一些 native 系统进程分配一些特定的权限, 读取出的属性值将保存到 `mSettings.mPermissions` 变量中。该函数是 PmS 中的一个重要功能函数, 其内部流程见后面小节。

⑥ 调用 `mSettings` 对象的 `readLP()` 函数从 `/data/system/packages.xml` 文件中读取所有应用程序中和包管理相关的信息, 这些信息将保存到 `mSettings.mPackages` 变量中。该函数的内部执行流程将在后面小节中介绍。

⑦ 对 Java 系统中的库文件进行 dex 提取 (转换)。在 Android 系统中, Java 源码编译后的 Class 文件不能直接被执行, Android 中的编译器会将程序中的多个 Class 文件转换成一个 dex 文件。转换后的文件和原始的 Class 文件相比有几点优势, 比如节省了字节码, 优化了内存分配, 重新组织函数映射表等。而 APK 或者 Jar 文件本身是一个 zip 压缩包, 其内部包含了真正的 dex 文件, 因此, 在程序执行前, 必须先从这些 Jar/APK 文件中提取出相应的 dex 文件。当然并不是说系统每次开机都会进行 dex 提取, 它会调用 `dalvik.system.DexFile.isDexOptNeeded()` 函数进行判断, 该函数内部会判断 dex 是否存在, 并且内部版本号和原始的 APK/Jar 是否相同。如果是, 就不进行 dex 提取, 否则, 系统认为原来的 dex 文件已经过期, 于是就会重新提取。

Java 系统中的库文件包含以下三个部分。

- Java Boot 路径下的所有文件, Boot 路径是指在 Terminal 下调用 `getProp("java.boot.class.path")` 获取的路径, 获取的路径以冒号为分隔符。
- 共享库路径: 该路径是在 `platform.xml` 中使用 `library` 标签定义的 Jar 文件。
- Frameworks 目录下的所有 APK 和 Jar 文件, 除了 `framework-res.apk` 之外, 因为该文件内部不包含代码, 所以不存在 dex 文件。

⑧ 为三个程序目录分别创建一个 `FileObserver`, `FileObserver` 对象内部会检测目录中添加、删除文件的事件, 并当事件发生时执行相应的操作。比如, 当目录中添加文件时, 就会调用 `scanPackageLI(file, ...)` 函数扫描添加的文件。注意该函数的参数, PmS 中存在另一个同名函数, 其第一个参数是 `Package` 类型, 而次参数是 `File` 类型, 为了区别, 以后分别简写为 `scanPackageLI(file)` 和 `scanPackageLI(package)`。

这三个目录分别如下。

- `/system/frameworks`: 该目录保存了 Framework 内核相关的程序。
- `/system/app`: 系统程序, 一般是指 Android 中自带的程序。
- `/vendor/app`: 第三程序, 一般是指厂商开发的自定义程序。

为什么没有为 `/data/app` 添加 `FileObserver` 呢? 是的, 的确应该添加, 在后面步骤中。

⑨ 调用 `scanDirLI()` 扫描（解析程序中的 `AndroidManifest.xml`）以上三个目录中的所有程序文件，并将扫描结果保存到 `PmS` 中的 `mPackages` 变量中。该函数的内部流程见后面小节。

⑩ 删除已经不存在程序对应的数据记录。`mPackages` 保存了以上三个目录下程序文件的列表，`mSettings.mPackages` 中保存了安装后的数据记录，因此，如果 `mPackages.contain()` 函数返回 `false`，则意味着以上三个目录下的某个程序已经被手工删除，于是调用 `mInstaller.remove()` 删除对应的记录。

⑪ 清除没有安装成功的数据记录。调用 `mSettings` 的 `getListOfIncompleteInstallPackages()` 函数获取没有安装成功的程序包列表，然后使用 `for()` 循环，调用 `cleanupInstallFailedPackage()` 逐个清除这些记录。所谓“没有安装成功”，就是指 `packages-backup` 中的 `package` 标签中属性 `installStatus` 值为 `false` 的情况，详情请参照后面关于程序安装小节的介绍。

⑫ 为以下两个第三方案程序目录分别添加 `FileObserver`，并调用 `scanDirLI()` 解析目录下的所有应用程序，解析结果将保存到 `mPackages` 变量中。这两个目录如下。

- `/data/app`：普通应用程序目录。
- `/data/app-private`：该目录目前并没有被使用。

⑬ 调用 `deleteTempPackageFiles()` 删除 `/data/app` 目录下的以 `vmdl` 开头及以 `.tmp` 结尾的文件。这种直接删除的做法可能有问题，当应用程序以 `vmdl` 开头时，可能会直接被删除。

⑭ 如果系统版本升级，调用 `updatePermissionsLP()` 重新为应用程序赋予权限。为什么版本升级后需要重新赋予权限呢？因为升级后可能会定义一些新的权限，为了保证已经安装过的程序保持原有权限，则需要重新赋予新的权限。比如原来程序是共享某个用户 `id`，系统升级后，共享 `id` 的权限发生了变化，则需要重新给原来的程序赋予新的权限。

⑮ 调用 `mSettings.writeLP()` 将 `mSettings.mPackages` 中的数据值重新写入 `packages.xml` 文件中。

以上就是 `PmS` 启动的主体过程，后面小节分别介绍该过程中所调用到的重要函数内部详情。

### 16.3.3 readPermission()内部过程

该函数的作用和名称稍有差异，从名称上看，该函数似乎是读取所有应用程序中关于权限的信息，但事实上却不是。该函数仅仅读取了系统中所定义的 `feature` 列表，以及给系统一些 `native` 进程分配的权限信息，而真正读取系统中所有权限列表及各应用程序所使用的权限列表却在 `mSetting.readLP()` 函数中。`readPermission()` 流程如图 16-2 所示。

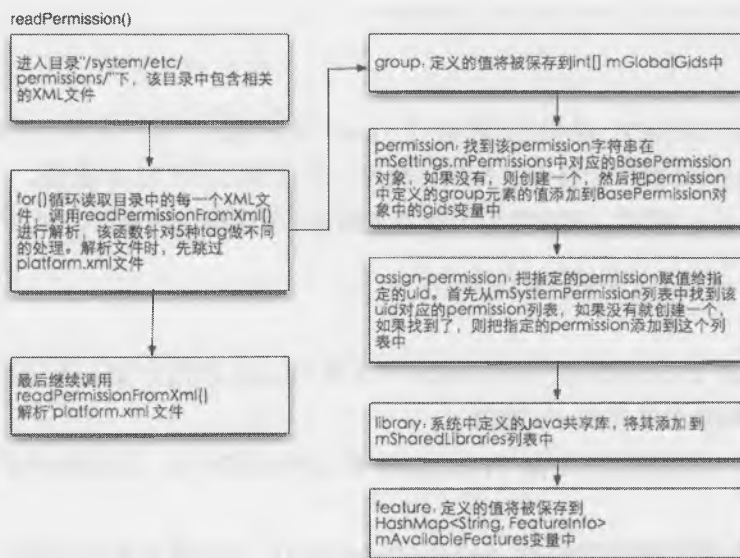


图 16-2 readPermission()的内部执行流程

① 确保/system/etc/permissions 目录的存在。

② 使用 for()循环, 读取该目录下的每一个 XML 文件, 并调用 readPermissionFromXml()函数逐个解析每一个 XML 文件, for()循环读取中跳过 platform.xml 文件。在一般情况下, 除了 platform.xml 文件外, 其他 XML 文件仅仅定义了一些系统的 feature, 包括 features.xml、handleheld\_core\_hardware.xml、android.hardware.wifi.xml 等, 平台设计人员可以在该目录下添加任意命名的 XML 文件描述平台所拥有的 feature。这些 feature 的名称原理上是任意的, 但是 Android 的 Market 会解析应用程序中所包含的 feature, 因此, feature 命名建议遵守共同的约定。比如, 以下为几个常见的 feature 名称。

- <feature name="android.hardware.wifi" />。
- <feature name="android.hardware.camera" />。
- <feature name="android.hardware.location" />。
- <feature name="android.hardware.location.network" />。
- <feature name="android.hardware.sensor.compass" />。
- <feature name="android.hardware.sensor.accelerometer" />。
- <feature name="android.hardware.bluetooth" />。
- <feature name="android.hardware.touchscreen" />。
- <feature name="android.hardware.microphone" />。

③ 最后解析 platform.xml 文件, 该文件使用 3 个标签定义了 3 部分内容, 分别如下。

- <permission>标签, 该标签用于给指定的 groupId 分配指定的 permission。它和 assign-permission 标签十分类似, 只是后者用于给指定的 userId 分配指定的 permission, 比如:

```
<permission name="android.permission.INTERNET" >
  <group gid="inet" />
</permission>
```

以上标签把 INTERNET 权限分配给了 inet 用户组，inet 本身只是一个字符串，在 readPermissionFromXml() 函数中会将该字符串转换为一个具体的用户组 id 数值。

- <assign-permission> 标签，该标签用于给指定的 uid 分配指定的权限，如以下代码所示：

```
<assign-permission
  name="android.permission.READ_CALENDAR"
  uid="shell" />
```

以上代码把 READ\_CALENDAR 权限分配给了名称为 shell 的用户 id，该 uid 是一个字符串，之后的代码将把该字符串转换为具体的 uid 数值。

- <library> 标签，该标签定义了除 Framework 库之外的共享库，应用程序可以假定该平台下包含这些共享库。

解析以上 XML 文件的函数是 readPermissionFromXml()，该函数内部可以解析 5 种类型标签，其执行过程如下。

(1) 解析 group 标签，该标签中包含 gid 属性，其值将被保存到 mGlobalGids[] 变量中。注意区分该 group 标签和 permission 标签中的 group 子标签，两者不是一回事。

(2) 处理 permission 标签，该 permission 将被保存到 mSettings.mPermissions 变量中，其子标签 group 的值将被保存到 mPermissions 中的成员变量 gids。group 中子标签使用字符串表示，系统中调用 Process.getGidForName(String) 将字符串转换为一个 int 型数值。

(3) 处理 assign-permission 标签，将指定的 permission 值添加到 mSystemPermission 列表中。该变量类型是 SparseArray<HashSet<String>>，即从 int 到 HashSet 的映射表，int 对应的是 uid，HashSet<String> 是该 uid 所具备的权限。

(4) 处理 library 标签，将该标签中定义的共享库文件添加到 mSharedLibraries 列表中。

(5) 处理 feature 标签，将该标签中定义的值保存到变量 mAvailableFeatures 列表中。该变量的类型是 HashMap<String, FeatureInfo>，String 代表 feature 的名称，FeatureInfo 是该 feature 包含的信息，除了名称 (name) 之外，还有 flags、reqGlesVersion。

执行完以上步骤后，PmS 就已经了解了系统中都包含哪些 feature，以及一些特定的 uid 都包含哪些默认的权限。

### 16.3.4 mSettings.readLP()

该函数将从 packages.xml 文件中读取所有的安装包信息，了解该函数内部流程之前需要先了解 packages.xml 和 packages-backup.xml 文件（以下简称 backup.xml）的关系。

在正常情况下，系统中只有 packages.xml 文件，而没有 backup 文件。当进行程序安装或者卸载时，



会将 package 文件重命名为 backup 文件, 然后把 mSettings.mPackages 中的数据序列化到该 backup 文件中。一旦正常写入则会把 backup 文件再命名回 packages.xml 文件, 但写入的过程中如果因为掉电或者其他什么原因导致系统崩溃, 则 backup 文件将一直存在, 直到下次系统开始初始化 PmS 时, 才重新读取该文件。

该函数的内部流程如图 16-3 所示。

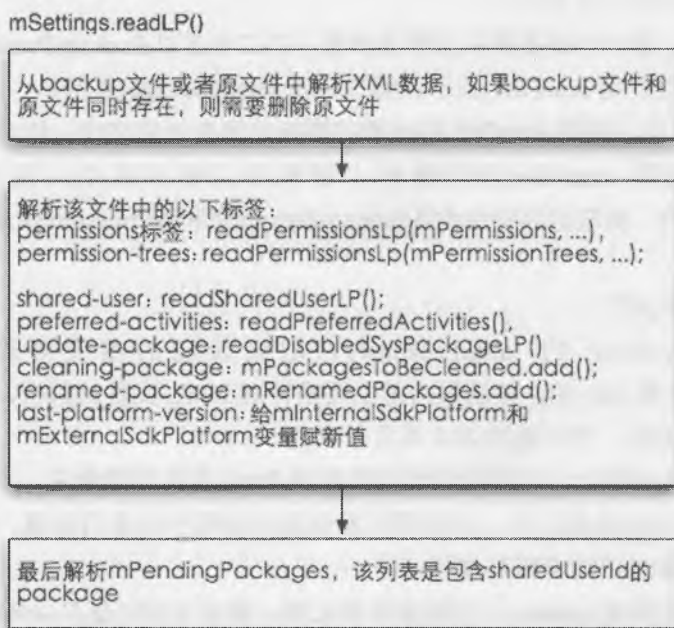


图 16-3 readLP()函数内部流程

① 判断 backup 文件是否存在。如果存在, 则意味着出现了异常情况, 那么调用 reportSettingProblem() 进行错误记录, 并删除原来的 packages.xml 文件, 然后从 backup 文件中读取相关的包管理信息。在正常情况下, 会从 package.xml 文件中读取包管理信息, 但 package.xml 也有可能不存在, 这种情况概率非常非常小, 只有当创建 package.xml 文件时系统突然掉电或者死机时才出现, 如果出现这种情况, readLP() 将返回 false。

② 解析 XML 文件中包含的各种标签, 这些标签的含义见前面小节。不同的标签调用不同的函数进行解析, 比如 permissions 标签调用 readPermissionsLp() 进行解析。解析的过程中, 如果某 package 是以 sharedUserId 运行的, 则暂时把该 Package 记录存放到 mPendingPackages 列表中, 以便解析完所有的 sharedUserId 标签后再把 sharedUserId 标签中的信息复制到之前 packages 记录中。

③ 处理 mPendingPackages 列表中对应的 package 信息。

readLP() 执行完毕后, mSettings.mPackages 变量中将包含所有 packages.xml 中记录的安装包信息。

### 16.3.5 scanPackageLI()内部过程

scanPackageLI()函数的作用从名称上看似乎仅仅是程序包的扫描,而其实际的作用还包括对程序的安装。程序安装可以分为两个过程,第一过程是把原始的 APK 文件复制到相应的程序目录下,第二过程则是为程序创建相应的数据目录及提取 dex 文件,并修改系统包管理信息等,而 scanPackageLI()其内部正是完成了程序安装的第二个过程。

该函数有两个重载,第一个函数是以文件为参数,第二个是以 PackageParser.Package 为参数。为叙述方便,以后第一个函数使用 scan(file)表示,第二个以 scan(package)表示。

在 PmS 的构造函数中,调用 scanDirLI()函数扫描指定目录下的文件,该函数中使用 for()循环对目录下的每一个文件调用 scan(file)进行解析。解析的目标有两个,第一是解析安装包中的 AndroidManifest.xml 文件,提取出其中包含的 intent-filter 信息和 permission 信息;第二是将安装包中的 dex 文件提取出来。

scan(file)函数分两步执行。

① 调用 PackageParser 的 parsePackage()获取安装包的相关信息,并将信息保存到 PackageParser.Package 变量 pkg 中。该函数内部主要是针对 XML 文件的解析,没有什么特别的逻辑,因此,其具体过程不再分析,有兴趣的读者可自行研究。

② 调用 scan(package)将上一步解析的结果转换到 PmS 的内部变量中,并且如果程序包合法,则创建该程序包所需要的各种数据文件,该步骤可看做是安装程序的最后步骤。该函数的代码超过 800 行,内部逻辑稍有些复杂,下面分析其过程。

③ 如果解析的包名称是 android,则需要单独处理,因为有些信息是 android 包特有的。

④ 处理 uses-library 标签,该标签对应的值将被保存到全局变量 mSharedLibraries 中,换句话说, mSharedLibraries 变量保存了所有应用程序所使用到的共享库。

⑤ 处理 uses-feature 标签,该标签对应的值将被保存到全局变量 mAvailableFeatures 列表中。

⑥ 处理 shared-userId,如果程序包声称要使用一个共享 userId,则把该 sharedUseId 保存到 mSettings.mSharedUserId 列表中。

⑦ 处理 originalPackage 的情况,这只有当程序是系统程序时才会执行,对于第三方应用程序,不能使用 android:originalPackage 指定程序的原始包,该属性仅仅是为了后向兼容。

⑧ 调用 verifySignatureLP()进行确保程序签名的正确性。

⑨ 如果调用 scan(package)时,参数 scanMode 包含 SCAN\_NEW\_INSTALL 标签,则需要确保该程序包中的 provider 不与系统中已有的 provider 冲突,即新的 provider 名称不能和已有的 provider 名称相同。

⑩ 如果该程序需要其他程序中定义的权限,则需要把该权限赋给该应用程序。

⑪ 截至这里,系统已经认为这个 APK 包是完全合法的,因此接下来就要创建程序执行时所需的各种文件。首先,为该应用程序创建数据目录,其中 android 包对应的数据目录是/data/system,该目录下一般保存的是 Framework 内核使用的数据文件。应用程序的数据目录是在/data/data 下。

⑩ 接着从程序包中提取出 dex 文件，并保存到/data/dalvik-cache 目录下。

⑪ 在 synchronized (mPackages) 同步域中完成以下操作。

(1) 把 pkg 保存到 mPacakges 列表中。

(2) 提取出该程序中的所有 Provider 对象，并保存到 PmS 的全局变量 mProviders 列表中。

(3) 提取出所有的 Service 对象，保存到 mServices 列表中。

(4) 提取出所有的 Receiver 对象，保存到 mReceivers 列表中。

(5) 提取出所有的 Activity 对象，并保存到 mActivities 列表中。

(6) 提取出程序中定义的 permissionGroup 信息，并保存到 mPermissionGroups 列表中。

(7) 提取出程序中定义的所有 permission，保存到 mSettings.mPermission 列表中。

(8) 提取出所有的 Instrumentation 对象，并保存到 mInstrumentations 列表中。一般的应用程序都不会包含 Instrumentation 对象，该对象仅在使用 Android UnitTest 中存在。

(9) 提取出所有的 Broadcast 对象，并保存到 mProtectedBroadcasts 列表中。

至此，scan(pkg)执行完毕。

### 16.3.6 mSettings.writeLP()

该函数的作用是把 mSettings.mPackages 的数据分别写入 packages.xml 和 packages.list 文件中，其流程如下。

① 检查 packages.xml 和 packages-backup.xml 是否同时存在，如果同时存在，则意味着出现了异常情况，此时需要先删除原来的 packages.xml 文件。在正常情况下，只有 packages.xml 文件存在，此时该文件重命名为 packages-backup.xml，总之，本步骤就是要保证目录中只有一个 packages-backup.xml 文件。

② 创建一个新的 packages.xml 文件，并把 mSettings 中保存的包管理信息序列化到该文件中。

③ 删除老的 packages-backup.xml 文件。

④ 重新生成 packages.list。首先新建一个 packages.list.tmp 临时文件，并将 mSettings.mPackages 中的信息写入到该临时文件，该临时文件的类型为 JournalFile 类，该类的构造函数中指定真正的文件和临时文件，当文件操作完毕后会自动删除该临时文件。

## 16.4 应用程序的安装和卸载

安装及卸载程序的操作都是由 PmS 完成，安装程序的过程包括在程序目录下创建以包名称命名的程序文件、创建程序数据目录，以及把程序信息保存到相关的配置文件 packages.xml 中，卸载过程则是相反的一个操作。

### 16.4.1 各主要功能类关系

在安装和卸载操作过程中，各主要功能类的相互关系如图 16-4 所示。

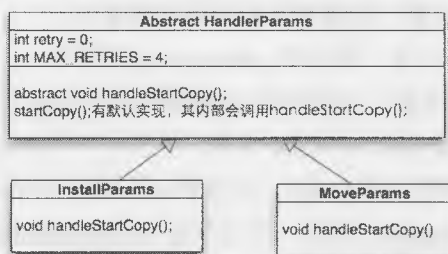


图 16-4 安装、卸载操作的功能类关系

在 PmS 内部存在图中所示的三个类。HandlerParams 是一个 abstract 类，该类有两个实现，分别是 InstallParams 和 MoveParams。HandlerParams 虚基类的作用是进行程序文件的复制，比如当用户指定一个 Hello.apk 文件后，该类将把该 APK 文件复制到程序目录/data/app 中，复制后的文件名称是 Hello 的包名称，比如 com.haiiii.android.Hello.apk。

HandlerParams 虚基类的两个实现类中，InstallParams 实现安装过程的复制，即全新的安装；MoveParams 实现移动复制，比如把已经安装好的程序从内部存储移动到外部存储或者相反。

在 InstallParams 类的 handleStartCopy()函数中，可以把应用程序安装到内部存储区，也可以安装到外部存储区，PmS 中定义了一个虚基类 InstallArgs 类用于完成不同类型的安装，如图 16-5 所示。

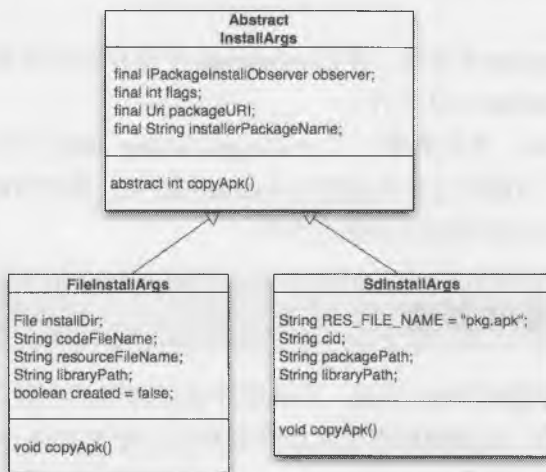


图 16-5 安装过程相关的类关系

InstallArgs 中包含一个虚方法 copyApk(), 该方法用于完成程序文件的复制。InstallArgs 有两个实现, 分别是 FileInstallArgs 和 SdInstallArgs, 这两个类分别用于实现将程序文件复制到内部存储区或外部存储区 (一般为 SD 卡)。

InstallArgs 中包含四个重要变量。

- observer: 该监听器由应用程序提供, 当 PmS 完成程序文件复制后, 可以回调该 observer 的相关方法, 以便应用程序可以在程序安装完毕后做点什么。
- flags: 一些辅助标识。
- packageURI: 该变量保存安装程序的路径。事实上, 安装程序可以是一个具体的文件, 也可以是一个 ContentProvider 对象, 比如 content://...。
- installerPackageName: 该字符串保存了安装器的名称, 该名称由应用程序指定, 可以任意。

而真正完成文件复制的工作并不是在 PmS 中, 而是在一个叫做 IMediaContainerService 的服务中, IMediaContainerService 仅仅定义了一个 IBinder 接口, 实现该 IBinder 的类是 DefaultContainerService, 该类在 framework/base/packages/DefaultContainerService 包中, 该服务以一个安装包的方式存在。有读者可能要问了, 既然安装服务是以安装包存在的, 那么该安装包本身又是如何被安装的呢? 这就是系统程序与第三方便序的区别, 对于系统程序而言, 不需要安装, 更确切地说是不需要程序文件的拷贝, 只需要创建相关数据文件即可, 只有第三方便序才需要进行程序文件复制。所以系统程序都保存在 /system/app 目录下, 并且程序文件的名称并不是包名称, 而第三方应用程序的程序文件都保存在 /data/app 目录下, 并且以包名称命名以防止名称冲突。

### 16.4.2 应用程序安装过程

应用程序安装过程首先从调用 PackageManager 类的 installPackage() 函数开始, 该函数会间接调用到 PmS 的 installPackage() 函数中。安装过程是异步的, installPackage() 函数内部仅仅是发一个异步消息, 名称为 INIT\_COPY, 意思就是“启动复制”, 由该名称可以看出安装过程首先是文件的复制。应用程序安装过程如图 16-6 所示。

在 INIT\_COPY 的消息处理代码中, 首先判断 mBound 变量是否为 true。true 意味着已经绑定 MCS (Media Container Service) 服务, 于是把安装信息保存到 mPendingInstalls 列表中, 并且如果 mPendingInstalls 列表只有一个安装信息时, 则立即发送 MCS\_BOUND 空消息, 意思就是“服务已经绑定了”。

如果 mBound 为 false, 则调用 connectToService() 绑定 MCS 服务, 然后把安装信息添加到 mPendingInstalls 列表中。大家可能会问, 为什么绑定后不发送“服务已经绑定了”的消息呢? 因为绑定服务是异步的, 因此, 此时仅仅是成功执行绑定服务命令, 而服务暂时还没有绑定, 所以应该在绑定服务的 onServiceConnected() 回调中发送 MCS\_BOUND 消息。事实就是这样, MCS 服务对应的绑定回调对象是变量 mDefContainerConn, 对象内部的 onServiceConnected() 函数中发送一个 MCS\_BOUND 消

息, 消息的 obj 参数值为 MCS 服务的 Binder 引用。

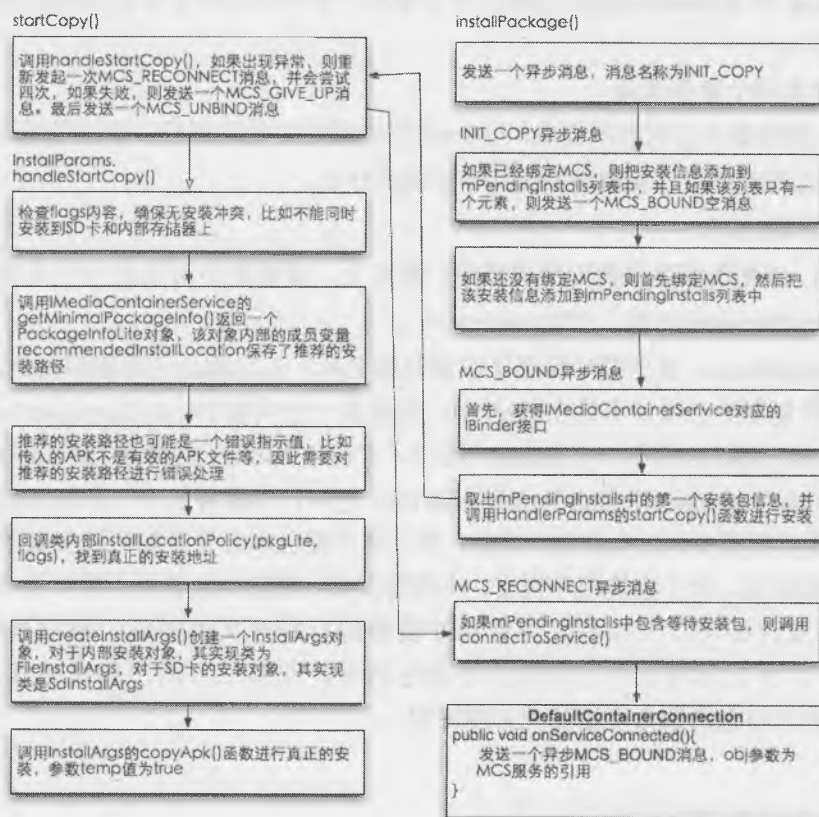


图 16-6 应用程序安装过程

在 MCS\_BOUND 消息处理中, 首先确保 MCS 服务是存在的, 并赋值给全局变量 mContainerServer, 然后从 mPendingInstalls 列表中取出最早的一个安装信息, 安装信息用 HandlerParams 类表示。该类是一个虚拟类, 其具体实现要么是 InstallParams, 要么是 MoveParams, 而调用 installPackages() 启动的程序安装则对应的是 InstallParams 对象, 该对象是在 installPackages() 函数中创建的, 其代码如下:

```

4568     Message msg = mHandler.obtainMessage(INIT_COPY);
4569     msg.obj = new InstallParams(packageURI, observer, flags,
4570                               installerPackageName);
4571     mHandler.sendMessage(msg);
  
```

最后调用 params 的 startCopy() 函数进行程序文件复制。InstallParams 类中采用了 retry 机制, 即如果没有复制成功, 就再复制一次, 连续尝试 4 次, 而进行复制是调用 handleStartCopy() 函数完成的。下面来看 InstallParams 类中的 handleStartCopy() 实现。

- ① 检查 flags 标识, 确保参数没有冲突, 比如不能同时安装到 SD 卡和内部存储区中。



② 以 packageURI 和 flags 为参数, 调用 MCS 服务的 getMinimalPackageInfo(), 询问 MCS 服务是否能够安装 packageURI 指定的程序。该函数返回值为 PackageInfoLite 类对象, 该类中的 loc 变量代表可以安装到什么位置, loc 是一个 int 类型。当然, 如果 MCS 不支持对 packageURI 的安装, 则会给 loc 变量中赋一个错误代码, 比如存储空间不够用了、安装程序已经安装过了, 等等。

③ 如果 MCS 认为可以安装 packageURI 指定的程序, 则回调 installLocationPolicy()。该函数的作用是对安装路径进行某种策略调整, 并返回最终应该安装的路径, 函数参数是上一步返回的 PackageInfoLite 对象和 flags 变量。

④ 此时已经确保 packageURI 是一个有效的安装程序, 并且得知了该程序应该安装到什么地方, 因此接下来就开始真正的文件复制。PmS 针对内部文件复制和外部文件复制分别提供了 FileInstallArgs 和 SdInstallArgs 两个类, 本步会根据上一步获得的 loc 类型, 调用 createInstallArgs() 选择相应的 InstallArgs 类。如果是内部存储区安装, 则创建 FileInstallArgs 对象。

⑤ 调用 InstallArgs 类的 copyApk() 函数进行文件复制。

下面继续来看 FileInstallArgs, 即内部存储区安装中的 copyApk() 函数执行过程, 其流程如图 16-7 所示。

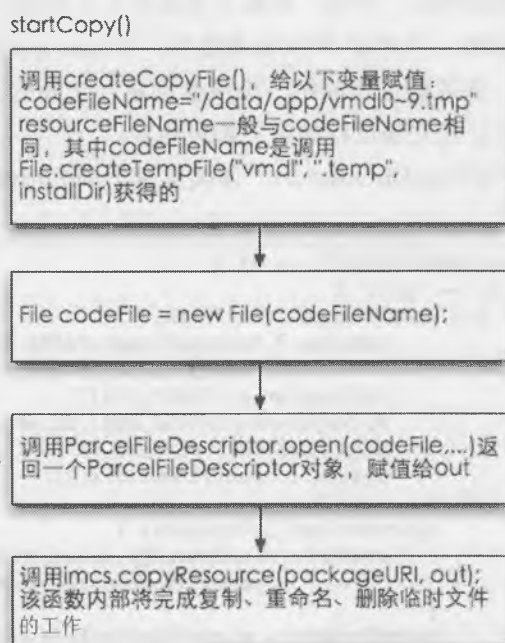


图 16-7 程序文件的复制过程

(1) 调用 createCopyFile() 创建一个临时文件, 临时文件名称为 vmdl0~9.tmp, 比如 vmdl1631123653.tmp。注意该函数内部是使用 File.createTempFile() 创建该临时文件的, 参数分别代表临时文件的前缀、后缀及路径。



(2) 以临时文件的路径构造一个 File 对象。为什么此处要先创建一个临时文件，然后再构造一个 File，而不是直接构造一个 File 对象呢？仅仅是为了效率，临时文件有特定的生命期，内部文件系统会为临时文件分配特定内存以提高效率。

(3) 以 codeFile 为参数构造出一个 ParcelFileDescriptor 对象。这个对象有点意思，请问 ParcelFileDescriptor 和普通的 FileDescriptor 有何区别呢？正如其名称所示，Parcel 意味着可以跨进程访问。为什么要这样呢？因为从根本上讲，复制程序文件的工作并不是 PmS 所在的进程完成的，而是 MCS 服务，对应的类是 DefaultContainerService，该服务运行于其他进程，并且在 MCS 服务中，要完成程序文件的复制、重命名，以及删除工作，因此，它需要一个可以跨进程的文件句柄。

(4) 调用 imcs 的 copyResource() 进行复制、重命名及删除工作，具体是指把 packageURI 所指定的程序文件内容复制到临时文件 vmdl0~9.tmp 中，然后将该 tmp 文件复制为以包命名的 APK 文件，最后删除该临时文件。关于该过程的详细信息参照 framework/base/packages/DefaultContainer 包下的相关文件。

至此，程序安装过程的第一阶段结束，此时/data/app 目录下多了一个以包命名的 APK 文件，并且包名称后面附加“-1”或“-2”。程序安装的下一阶段是把该 APK 中包含的包信息提取出来存放到 packages.xml 文件及 mSettings.mPackages 对象中。从该描述中大家可以看出，这个工作似乎和 PmS 在启动时调用 scanDirLI() 的过程比较相似，没错，实际上完成这个工作的代码也就是那段代码。

前面介绍 PmS 启动过程时讲过，在 PmS 的构造函数中，为/data/app 添加了一个 FileObserver 对象，用于监控该目录下文件的变化，而安装过程的第一阶段会创建一个新的 APK 包，因此，FileObserver 会对此作出响应，其执行的代码正是安装过程的第二阶段内容。

PmS 中定义的 FileObserver 是 AppDirObserver，该类重载了 onEvent() 函数，其中 ADD\_EVENT 消息的执行代码如下：

```

4511         if ((event & ADD_EVENTS) != 0) {
4512             if (p == null) {
4513                 p = scanPackageLI(fullPath,
4514                     (mIsRom ? PackageParser.PARSE_IS_SYSTEM
4515                     | PackageParser.PARSE_IS_SYSTEM_DIR: 0) |
4516                     PackageParser.PARSE_CHATTY |
4517                     PackageParser.PARSE_MUST_BE_APK,
4518                     SCAN_MONITOR | SCAN_NO_PATHS | SCAN_UPDATE_TIME,
4519                     System.currentTimeMillis());
4520             if (p != null) {
4521                 synchronized (mPackages) {
4522                     updatePermissionsLP(p.packageName, p,
4523                         p.permissions.size() > 0, false, false);
4524                 }
4525                 addedPackage = p.applicationInfo.packageName;
4526                 addedUid = p.applicationInfo.uid;
4527             }
4528         }

```

代码中首先调用 scanPackageLI(file) 函数，该函数的执行过程参照前面小节，其作用就是从新添加的 APK 文件中提取出包管理信息。然后调用 updatePermissionsLP() 进行权限提取。

至此，安装程序就结束了。在以往的开发调试时，读者会碰到这样一种情况，比如，调试系统程序

时,一般直接使用 `adb push` 命令把新的安装包存放到 `/system/app` 目录下,其效果似乎等同于 `adb install`。原因是系统程序的名称使用的是程序名称,而不是包名称,所以,两者没有什么分别,但对于第三方程序, `/data/app` 下的程序必须以包名称命名,因此,必须使用 `adb install` 进行安装,而不能直接使用 `adb push`。

### 16.4.3 应用程序的卸载过程

`PackageManager` 中提供了 `deletePackage()` 函数用于卸载程序,该函数通过 IPC 调用到 `PmS` 的 `deletePackage()` 函数,继而调用到 `deletePackageX()`。源码中该函数的注释很明确,说明了该函数的作用及执行过程。当然,就算没有这段说明,我们也能够猜测到该函数的过程,因为基本上所有的操作系统的卸载过程都是安装过程的逆操作。

删除过程的流程如图 16-8 所示。

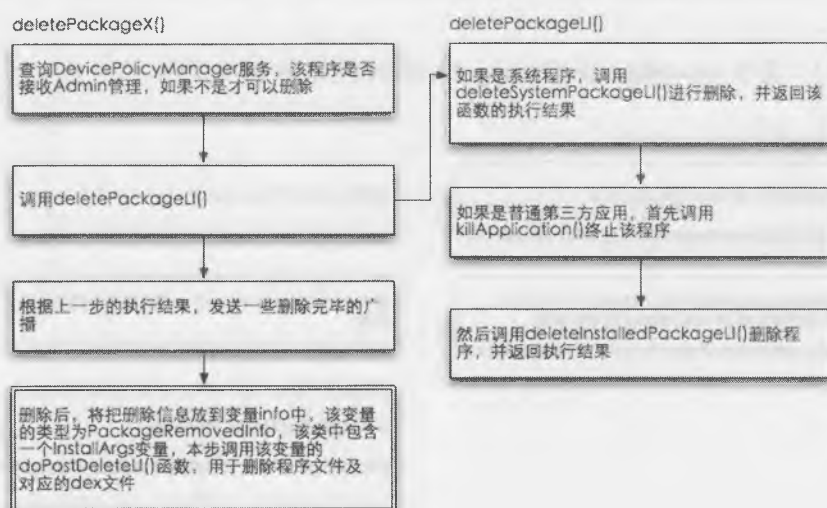


图 16-8 程序删除过程

① 检查该程序是否有 Admin 管理。该机制是在 Android 2.2 版本中引入的,其意义是告诉系统,该程序是接受 Admin 管理的,是被强制安装的。这就好比当你入职一家新公司后,公司的 IT 部门告诉你其中必须要安装哪些应用软件,这些应用软件将接受 Amdin 的远程管理,因此,这些软件是不能被删除的。Framework 中新添加了一个 `DevicePolicyManagerService` 服务,本步调用该服务的 `packageHasActiveAdmins()` 函数查询指定的程序是否接受 Amdin 的管理,如果是,则 `PmS` 组织卸载该程序。关于 DPM 服务的细节可参照 `frameworks/base/services/` 路径下的 `DevicePolicyManagerService` 类。

② 调用 `deletePackageLI()` 开始删除程序,该函数中将主要删除以下内容。

- 程序文件,即 `/data/app` 目录下相应的 APK 文件。

- PmS 中的 mSettings.mPackages 变量及 mPackages 变量中，以及其他相关变量中关于该程序的包管理信息。

具体删除过程如下。

- (1) 如果被删除的程序是系统程序则调用 `deleteSystemPackageLI()`，并返回其执行结果。
- (2) 对于普通的第三程序，则首先调用 `killApplication()` 终止该程序的运行。
- (3) 然后调用 `deleteInstalledPackageLI()` 从程序目录下删除对应的 APK 文件，并更新 mSettings 中对应的程序信息。

③ 发送一个广播消息，消息内容可能是 `Intent.ACTION_PACKAGE_REMOVE`，也可能是 `ACTION_UID_REMOVE`，这取决于上一步的执行结果。同时，如果是系统程序的升级，还会发送一个 `ACTION_PACKAGE_ADDED` 消息和 `ACTION_PACKAGE_REPLACED` 消息。

④ 此时，上一步的返回信息中将包含一个 `InstallArgs` 对象，于是调用该变量的 `doPostDeleteLI()` 函数，进行“善后”处理。所谓的善后实际上就是删除程序文件本身，以及在 `dalvik-cache` 下创建的 dex 文件。

关于以上第 (3) 步中 `deleteInstalledPackageLI()` 的流程如图 16-9 所示。

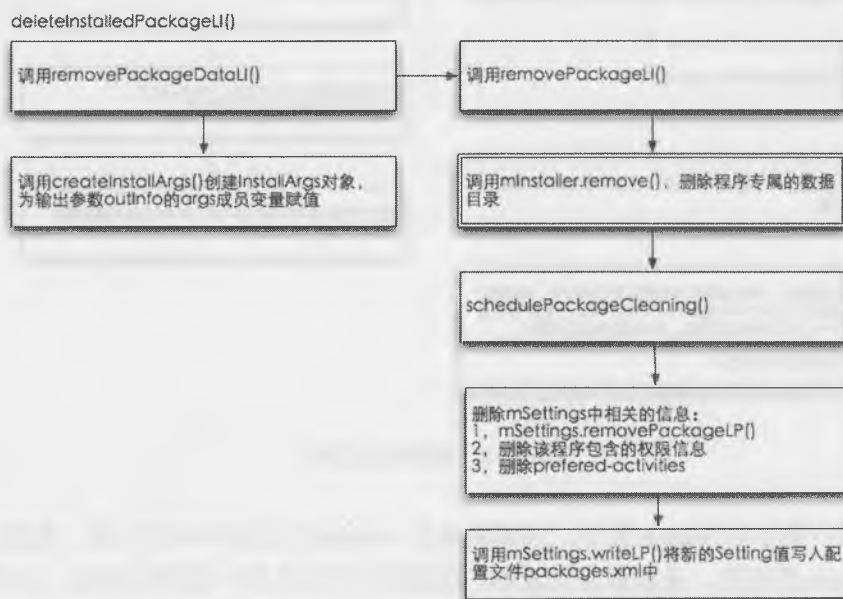


图 16-9 `deleteInstalledPackageLI()` 内部执行过程

- ① 调用 `removePackageDataLI()`，删除 mPackages 变量中保存的包信息，以及全局变量 mActivities、mServices、mProviders 等中保存的和该程序相关的记录信息。
- ② 调用 `mInstaller.remove()`，删除程序专属的数据目录，即 `/data/data/` 目录下的文件。
- ③ 调用 `schedulePackageCleaning()`，启动“清理外部存储器”的服务，其内部仅仅是调用了 `AmS`

的 `startService()`。参数 `Intent` 的 `action` 字段值为 `ACTION_CLEAN_EXTERNAL_STORAGE`, `component` 字段的值为 `com.android.defcontainer.DefaultContainerService`, 至于 `DefaultContainerService` 内部如何处理与该程序相关的外部存储区数据则是 DCS 的事情。从逻辑上讲, DCS 服务必须要知道 PmS 希望清理哪个应用程序对应的外部数据, 然而 `Intent` 中却没有包含这个包信息, 怎么办呢? 事实上, 在 DCS 中处理 `ACTION_CLEAN_EXTERNAL_STORAGE` 消息时, 会回调 PmS 服务的 `nextPackageToClean()` 函数, 该函数会返回 `mSettings.mPacakgeToBeCleaned` 列表中最早的一个元素, 从而使 DCS 知道应该清理哪个包。

④ 删除 `mSettings` 中和该程序相关的信息, 具体包含三个:

- 调用 `mSettings.removePackageLP()` 删除 `mSettings.mPackages` 中相关的信息。
- 删除 `mSettings` 中该程序定义的权限信息。
- 删除 `preferred-activities` 对应的信息。

⑤ 此时 `mSettings` 中已经完全剔除了被删除程序相关的信息, 因此需要把这些新的数据重新写入到 `packages.xml` 文件中, 因此, 调用 `mSettings.writeLP()` 完成此操作。

## 16.5 intent 匹配框架

`intent` 匹配主要解决应用程序中没有明确指定 `Component` 名称的情况。PmS 在初始化时, 会从所有应用程序的 `AndroidManifest.xml` 文件中读取 `intent-filter` 的值, 然后建立一个内部数据结构, 以后应用程序可以通过 `PackageManager` 提供的 `queryXXX()` 方法分别查询不同 `Component` 的包信息。

本节仅以 `Activity` 的匹配过程为例说明匹配架构设计的思路, 其他类型的匹配与此类似, 有兴趣的读者可自行研究。

### 16.5.1 主要功能类的关系

为完成 `intent` 的匹配, 系统中定义了一些类或者子类, 这些类的关系如图 16-10 所示。

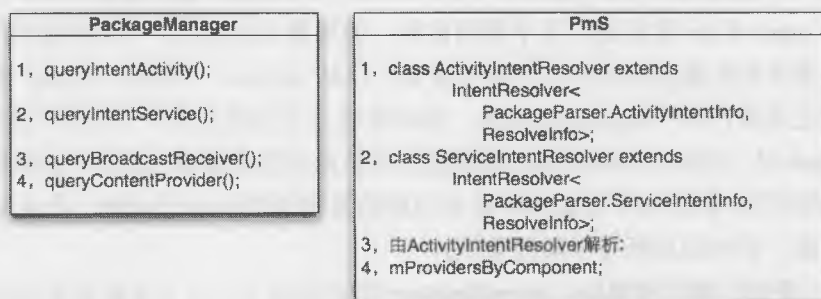


图 16-10 用于 `intent` 匹配的相关函数和类

首先, PackageManager 类中定义了 4 个 query 函数, 分别用于匹配 Framework 定义的四中不同的 Component, 分别是 Activity、Service、Receiver 及 ContentProvider。

在 PmS 内部, 分别定义了四个变量, PmS 使用这四个变量分别保存不同类型 Component 的 intent 匹配相关的数据信息。其中所有匹配到 Activity 的 intent-filter 信息保存在 ActivityIntentResolver 内部类对象中; 所有匹配到 Service 的 intent-filter 信息保存在 ServiceIntentResolver 内部类对象中; Receiver 对应的 intent-filter 信息同样保存在 ActivityIntentResolver 类对象中; 最后, PmS 中并没有为 ContentProvider 对应的 intent-filter 信息单独定义内部类, 而是仅仅保存到了内部变量 mProviderByComponent 中。

同时, PmS 内部提供了四个对应的 query 函数, 这些 query 函数和 PackageManager 定义的 query() 函数一一对应。这些函数内部执行的操作正是从以上四个数据类中提取出匹配到的信息, 并将这些信息按照 PackageManager 中定义的返回值类型重新组合, 并返回。

PmS 内部类 ActivityIntentResolver 的基类是 IntentResolver, 该类内部定义了一些核心数据变量, 在 PmS 启动时调用的 scanDirLI() 函数内部, 会填充这些数据变量, 关于该过程可参照前面小节。下面主要来看 IntentResolver 类中的重要数据变量的意义, 如图 16-11 所示。

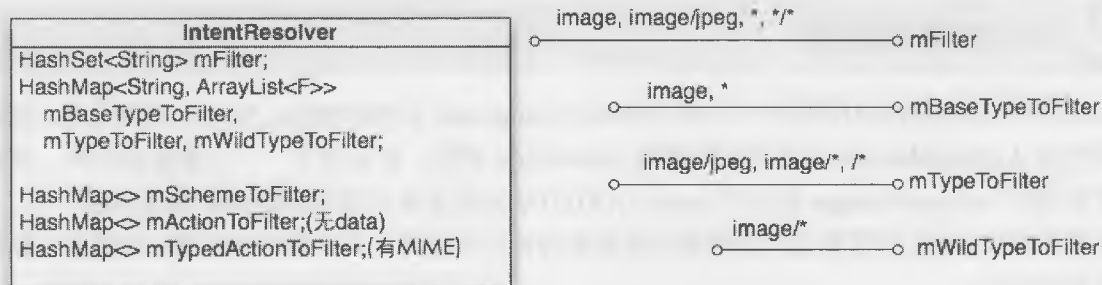


图 16-11 用于 intent 匹配的重要变量

在分析该类内部数据前, 读者可以先考虑一下, 如果让你来设计 intent 匹配的实现, 你会在程序内部使用何种数据结构保存 intent-filter 的信息呢? 答案可能有两种。

第一种, 定义一个 intent-filter 的数据类, 内部包含所有可能的字段, 然后当传入一个具体的 intent 时, 逐个和每一个 intent-filter 数据进行某个逻辑比较, 如果满足就返回, 否则继续比较下一个。

第二种, 按照基本的匹配类型定义几个数据变量, 比如 action、scheme、mime 等, 每个数据变量中保存所有只要满足本条件的 Component 信息, 其结果就是不同的变量中会包含相同的 Component 信息。当传入一个 intent 时, 先将该 intent-filter 分解成这些基本的匹配条件, 比如可分解成 action、scheme、mime 等几种有限的字段, 然后逐个用这些字段在前面的数据变量中进行匹配, 并最终找到不同数据变量匹配结果中的交集, 即为满足所有字段的条件。

在 Android 中, 采用了第二种思路。IntentResolver 类内部定义了几个有限的数据变量, 这些变量保存了 intent-filter 不同的字段可能包含的 Component 对象。首先是和 MIME 类型相关的四个变量, 分别

为 mFilter、mBaseTypeToFilter、mTypeToFilter 及 mWildTypeToFilter，这四个数据变量中包含的内容如图 16-11 右方所示。mFilter 包含了所有包含 MIME 类型的目标对象，mBaseTypeToFilter 是 mFilter 的一个子集，mTypeToFilter 也是 mFilter 的一个子集，mWildTypeToFilter 是 mTypeToFilter 的一个子集。

接着定义了和 scheme 及 action 字段相关的数据变量，分别为 mSchemeToFilter 和 mActionToFilter。另外还额外定义了一个 mTypedActionToFilter，它也是一个和 action 字段相关的数据变量，它和 mActionToFilter 的区别在于，它是同时包含 Action 和 MIME 字段的，而前者没有 data 字段。

有了以上数据变量后，剩下的就仅仅是进行匹配操作了。

### 16.5.2 主体调用过程

下面以 ActivityIntentResolver 中的匹配过程为例说明主体匹配过程，其流程如图 16-12 所示。

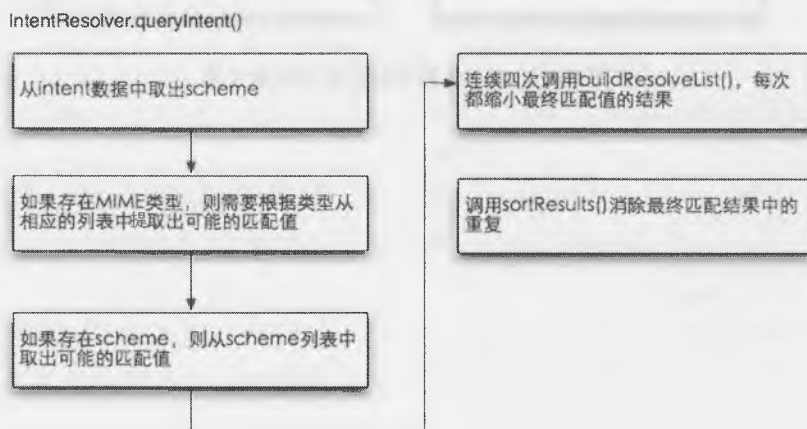


图 16-12 intent 匹配过程

由于 ActivityIntentResolver 的基类是 IntentResolver，因此以上流程实际上来自于 IntentResolver 内部的 queryIntent() 函数。

- ① 从传入的 Intent 数据中提取出 scheme，比如 http://xxx 中的 scheme 就是 http。
- ② 查看 Intent 数据中是否包含 MIME 类型。如果包含的话，则根据类型是否包含通配符及是否包含子类型，分别从前面所提到的四个数据变量中找到可能的匹配值。
- ③ 如果存在 scheme，也从 scheme 中提取出可能匹配值。
- ④ 分别针对 intent 中的每个字段，从数据变量中寻找到所有可能匹配到的结果的交集，代码中连续调用 buildResolveList() 函数进行匹配。每次调用结束后，参数 finalList 保存的都是本次匹配后的可能结果，所以，连续调用过程中 finalList 中包含的结果是一个递减的过程。同时，四次调用 buildResolveList() 的次序可以不分先后。

⑤ 调用 `sortResults()` 消除匹配结果中的重复, 并返回 `finalList()`。

函数 `buildResolveList()` 内部的核心是调用 `filter` 的 `match()` 函数, 而 `filter` 的类型是一个模板类型, 实际运行时, 其类型为 `PackageParser` 的内部类 `ActivityInfoIntentInfo` 或者 `ServiceIntentInfo`, 其继承关系如图 16-13 所示。`match()` 函数最终是执行 `IntentFilter` 类中的 `match()` 函数。

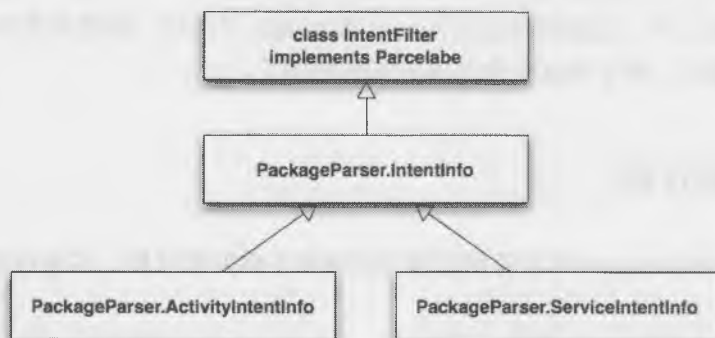


图 16-13 intent 相关的数据类继承关系





## 第 17 章 输入法框架

输入法框架（Input Method Framework）简称 IMF，该框架定义了一套接口，允许系统安装不同的输入法，输入法开发商应该基于该框架开发具体的输入法，从而使用户可以在不同的输入法之间进行切换。

IMF 本身并不复杂，该框架的核心思想有两点，第一是采用 Service 的方式运行具体的输入法，第二是在 Service 中创建输入法窗口，并把输入的内容传递到编辑框中。

抛开 IMF 本身的各种定义，输入法的本质仅仅是创建一个特别的系统级窗口，就像状态栏窗口或者系统对话框一样。所不同的仅仅是该窗口包含了一个虚拟键盘，并且处理虚拟按键的 onClick() 事件，然后把虚拟的字符传递到客户程序的编辑框中。为此，编辑框（EditText）需要实现某种特殊的接口，以便接收字符。

本章开始前，先约定以下几个缩写。

- IMF：输入法框架（Input Method Framework），这可不是“国际货币基金组织”。
- IM：输入法（Input Method）。
- IMS：输入法服务（Input Method Service），一般是指一个具体的输入法对应的服务。
- IMMS：输入法服务管理器（Input Method Manager Service），属于系统进程的一部分，系统中只有一个该服务的实例。
- IMM：输入法管理器（Input Method Manager），每个客户进程中包含一个该实例。
- IME：（Input Method Engine），泛指一个具体的输入法，包括其内部的 IMS 和各种其他 Binder 对象。

## 17.1 输入法框架组成概述

输入法框架组成如图 17-1 所示。

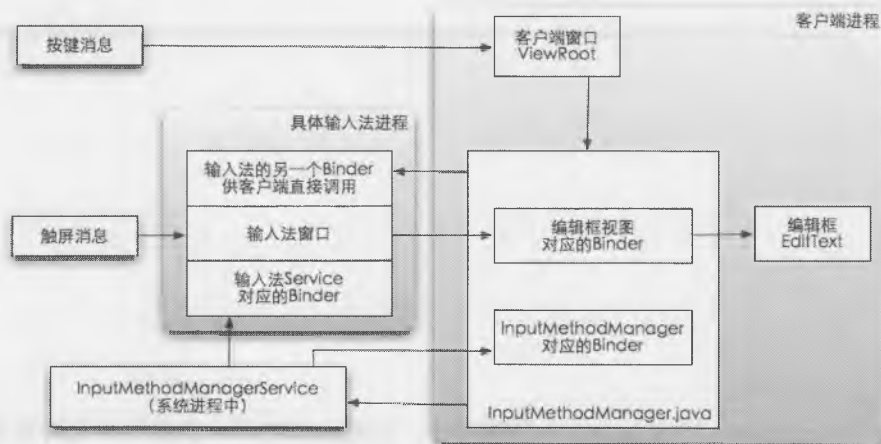


图 17-1 输入法框架组成

首先，在每个客户端进程中，都存在一个输入法管理器（Input Method Manager），简称 IMM，程序员可通过该管理器访问输入法管理服务（Input Method Manager Service），简称 IMMS。

IMM 中存在两个 Binder 对象。一个是编辑框视图对应的 Binder 对象，具体的输入法进程将调用该 Binder 对象把虚拟按键消息传递给对应的编辑框（EditText）。另一个 Binder 对象将用于 IMMS 访问客户端进程。为什么 IMMS 还需要访问 IMM 呢？因为 IMMS 需要把具体某个输入法服务包含的 Binder 对象传递给客户端进程，从而使得客户端进程之后就可以直接访问输入法服务。

具体的某个输入法中也存在两个 Binder 对象。一个是输入法服务对应的 Binder 对象，IMMS 将通过该对象去控制输入法的状态，比如显示、隐藏输入法；另一个是专门供客户端调用的 Binder 对象，该对象主要用于在输入的过程中，客户端向输入法传递一些事件，最常见的就是客户端把按键消息通过该 Binder 对象传递到输入法窗口中。

当有输入法窗口显示时，消息的处理流程大致如下，按键消息和触屏消息稍微有所不同。

对于按键消息，首先会传递到客户端窗口对应的 ViewRoot 对象中，由 ViewRoot 判断是否有输入法窗口。如果有的话则通过 IMM 传递到输入法进程中的“客户端专用”Binder 对象，该对象内部会处理按键消息，并把按键消息转变为一个虚拟按键消息，然后再把该虚拟按键消息传递到编辑框对应的 Binder 对象中，该 Binder 对象再调用 EditText 的相应方法改变显示的文字。

对于触摸屏消息，当该触摸位置在输入法窗口中时，该消息自然是直接传递到了输入法窗口中。该窗口内部会把触摸消息转变为一个虚拟按键消息，并调用编辑框对应的 Binder 对象把这个虚拟按键消息传递到对应的编辑框，从而改变编辑框中的文字。

以上过程存有两个问题值得思考，分别如下：

- 输入法窗口如何获得客户端中编辑框对应的 Binder 对象？
- 客户端又如何获得输入法服务中的“客户端专用”的 Binder 对象？

这两个问题的答案将在下节分析。

## 17.2 输入法中各 Binder 对象的创建过程

上一节大致介绍了 IMF 中存在的各种 Binder 对象，以及它们的作用，本节就来具体分析各 Binder 对象的创建过程，以及在源码中对应的类文件。

### 17.2.1 InputConnection

InputConnection 本身只是一个 interface 接口，它定义了输入法对象（编辑框）应该提供的函数接口，这些函数将被输入法调用，而调用则通过 Binder 实现。该 Binder 相关的类文件信息如下：

- InputConnection.java：纯粹的 interface，定义了所有的 API。
- InputConnectionWrapper：InputConnection 接口的包装，相当于一个空壳子。
- EditableInputConnection.java：这是该 interface 的具体实现。
- IInputContext.aidl：定义该 Binder 的 aidl 文件名称，该名称显得和接口不协调，或者应该定义为 IInputConnection.aidl 更合适。
- IInputContextWrapper：该 Binder 内部函数的具体实现。
- ControlledInputConnectionWrapper：这是 IMM 内部的一个子类。

以上几个类的关系如图 17-2 所示。

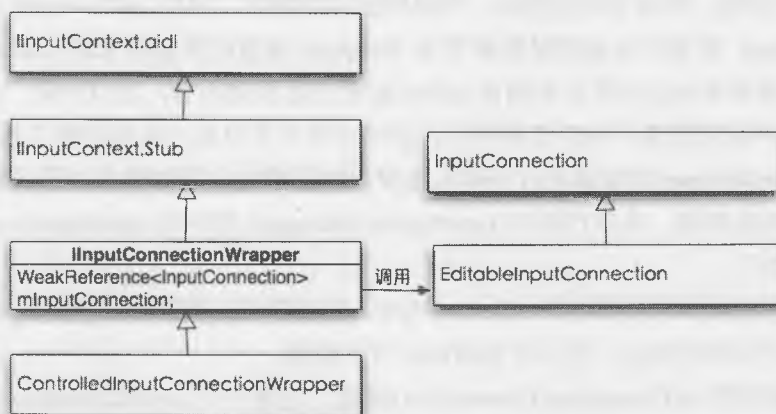


图 17-2 输入对象的接口类关系

当进行 Binder 调用时, `IInputConnectionWrapper` (以下简称 `Wrapper`) 会调用其内部的 `InputConnection` 对象的相应方法, 并通过一个远程回调, 将执行结果返回给调用者。比如, `Wrapper` 中的 `getTextAfterCursor()` 函数, 其源码如下:

```
89 public void getTextAfterCursor(int length, int flags, int seq, IInputContextCallback callback) {
90     dispatchMessage(ObtainMessage(IISC.DO_GET_TEXT_AFTER_CURSOR, length, flags, seq, callback));
91 }
```

远程对象调用该 Binder 时, 会传递过来一个 `IInputContextCallback` 类型的参数, 名称为 `callback`。因为该 `callback` 是 IPC 调用的参数, 所以它必须实现 `parcelable` 接口的类, 查看该类的源码可知, 它是一个 Binder 对象。

在 `Wrapper` 类内部采用了异步处理机制, 即先向内部线程发送一个消息, 然后再异步去处理。此处发送了一个 `DO_GET_TEXT_AFTER_CURSOR` 消息, 该消息的处理函数如以下代码所示:

```
185 void executeMessage(Message msg) {
186     switch (msg.what) {
187         case DO_GET_TEXT_AFTER_CURSOR: {
188             SomeArgs args = (SomeArgs)msg.obj;
189             try {
190                 InputConnection ic = mInputConnection.get();
191                 if (ic == null || !isActive()) {
192                     Log.w(TAG, "getTextAfterCursor on inactive InputConn");
193                     args.callback.setTextAfterCursor(null, args.seq);
194                     return;
195                 }
196                 args.callback.setTextAfterCursor(ic.getTextAfterCursor(
197                     msg.arg1, msg.arg2), args.seq);
198             } catch (RemoteException e) {
199                 Log.w(TAG, "Got RemoteException calling setTextAfterCurs");
200             }
201             return;
202         }
```

以上代码中, 首先获取该 `Wrapper` 内部包含的 `InputConnection` 对象, 然后调用该对象的 `getTextAfterCursor()` 方法, 等该方法返回后, 将返回值作为参数, 调用 `callback` 的 `setTextAfterCursor()` 方法。这里的 `callback` 变量正是远程对象调用该 `Wrapper` 函数时传递进来的 `callback`, 而它也是一个 Binder 对象, 从而使得 `Wrapper` 可以使用该 `callback` 通知远程调用者, 这就叫做“远程回调”。远程回调在多 Binder 架构中使用较多, IMF 多处使用, 后续小节将不再对“远程回调”机制作具体介绍。

`IInputConnectionWrapper` 对象是在打开输入法窗口时创建的, 因为输入法窗口需要该 Binder 对象, 以便和编辑框进行消息传递。具体代码在 `InputMethodManager` 类中的 `startInputInner()` 方法中, 该方法内部执行流程大致如下。

① 获取当前正在服务的视图对象 `mServedView`, 正在服务的视图指的就是获取了焦点 (focus) 的视图, 任何视图都可以获得焦点, 但只有 `EditText` 可以编辑。

② 回调焦点视图的 `onCreateInputConnection()` 函数, 创建一个 `InputConnection` 对象, 并赋值给变量 `ic`, 而在 `TextView` 的该回调函数正创建了 `EditableInputConnection`, 如以下代码所示:

```

4627         if (mText instanceof Editable) {
4628             InputConnection ic = new EditableInputConnection(this);
4629             outAttrs.initialSelStart = getSelectionStart();
4630             outAttrs.initialSelEnd = getSelectionEnd();
4631             outAttrs.initialCapsMode = ic.getCursorCapsMode(mInputType);
4632             return ic;

```

③ 用该 ic 构造出一个 Wrapper 对象。

④ 以新建的 Wrapper 对象作为参数，调用 IMMS 的 startInput() 函数，而在 IMMS 中，则会把该 Wrapper 对象传递到具体的输入法服务中，从而使输入法服务知道该客户端编辑框对应的 Binder。

## 17.2.2 IInputMethodClient

IInputMethodClient 本身是一个 aidl 文件，它会被 Aidl 编译器编译为一个 Binder 本地代理类。该 Binder 的作用是向 IMMS 提供一个接口，以便 IMMS 向客户端程序传递一些和输入法相关的信息，并进行一定的控制。

- IInputMethodClient.aidl：该 Binder 的声明文件。
- IMM 中的 mClient 变量：该 Binder 的一个具体实现。

该 Binder 对象是在 IMM 对象构造时创建的，一个应用程序仅有一个 IMM 对象，因此也就只有一个该 Binder 对象。从 IMM 内部远程调用 IMMS 的函数时，该 client 都将作为函数的一个参数传递给 IMMS，从而使 IMMS 知道该调用来自于哪个应用程序的 IMM。

那么这个 client 对象是什么时候被添加到 IMMS 中的呢？这就要从创建客户窗口说起了。前面章节讲过，客户端要向 WmS 申请添加窗口，具体是通过 WmS 中的子类 Session 完成的，WmS 为每一个客户进程分配一个 Session 对象，客户进程中要添加窗口都通过该 Session 对象。这个 Session 对象存在于 ViewRoot 类中的 sWindowSession 变量中，它是一个静态变量，从而保证每个客户进程只有一个该对象。

而在第一次添加窗口时，sWindowSession 的值为空，因此需要请求 WmS 先创建一个新的 Session 对象。请求的代码在 ViewRoot 中的 getWindowSession() 函数中，如以下代码所示：

```

225 public static IWindowSession getWindowSession(Looper mainLooper) {
226     synchronized (mStaticInit) {
227         if (!mInitialized) {
228             try {
229                 InputMethodManager imm = InputMethodManager.getInstance(mainLooper);
230                 sWindowSession = IWindowManager.Stub.asInterface(
231                     ServiceManager.getService("window"))
232                     .openSession(imm.getClient(), imm.getInputContext());
233                 mInitialized = true;
234             } catch (RemoteException e) {
235             }
236         }
237         return sWindowSession;
238     }
239 }

```

而 `getWindowSession()` 函数又是在 `ViewRoot` 的构造函数中调用的, 因此, 在 `ViewRoot` 对象创建时, 就会申请一个 `Session` 对象。在申请函数中, 具体是调用了 `WmS` 提供的 `openSession()` 方法, 该方法有两个参数。其中第一个参数是调用 `imm.getClient()` 产生, 而这正是返回了 `IMM` 中 `client` 变量; 第二个参数则是 `IInputContext` 对象, 即上一节中所讲的 `IInputConnectionWrapper` 对象。`getInputContext()` 这个函数名称很容易让人误解, 表面上看似乎是会返回一个 `Context` 对象, 实际则不是, 请注意甄别。

在 `WmS` 端的 `openSession()` 函数中, 则会使用该 `client` 变量创建一个新的 `Session` 对象, 如下代码所示:

```
8129 Session session = new Session(client, inputContext);
8130 return session;
```

然后, 在 `Session` 类的构造函数中, 则把 `client` 和 `inputContext` 变量告诉给了 `IMMS`, 从而使 `IMMS` 知道该客户进程的存在, 并按照 `client` 和 `inputContext` 为该客户进程建档, 如下代码所示:

```
5609 mInputMethodManager.addClient(client, inputContext,
5610                               mUid, mPid);
```

在 `IMMS` 中的 `addClient()` 函数中的关键代码如下:

```
582 synchronized (mMethodMap) {
583     mClients.put(client.asBinder(), new ClientState(client,
584     inputContext, uid, pid));
585 }
```

该函数的“建档”正是建立了一个 `ClientState` 内部对象, `mClients` 是一个 `HashMap` 类型的列表。

至此, 在 `WmS` 的撮合下, `IMMS` 建立了和客户端 `IMM` 的联系。接下来, 具体的输入法又需要 `IMMS` 的撮合, 和 `IMM` 建立联系, 这个关系如图 17-3 所示。

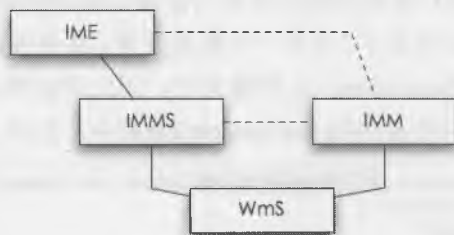


图 17-3 输入法相关的各服务模块关系

### 17.2.3 InputMethodSession

`InputMethodSession` 本身只是一个 `interface`, 它定义了 `IMM` 可以直接访问输入法本身的函数接口, 相关的类文件描述如下。

- InputMethodSession.java: 仅仅为一个 interface。
- AbstractInputMethodService.AbstractInputMethodSessionImpl: 该接口的一个虚拟实现。
- InputMethodService.InputMethodSessionImpl: 该接口的一个完成实现。
- IInputMethodSession.aidl: 定义了一个 Binder。
- IInputMethodSessionWrapper.java: 该 Binder 的具体实现。

其类关系如图 17-4 所示。

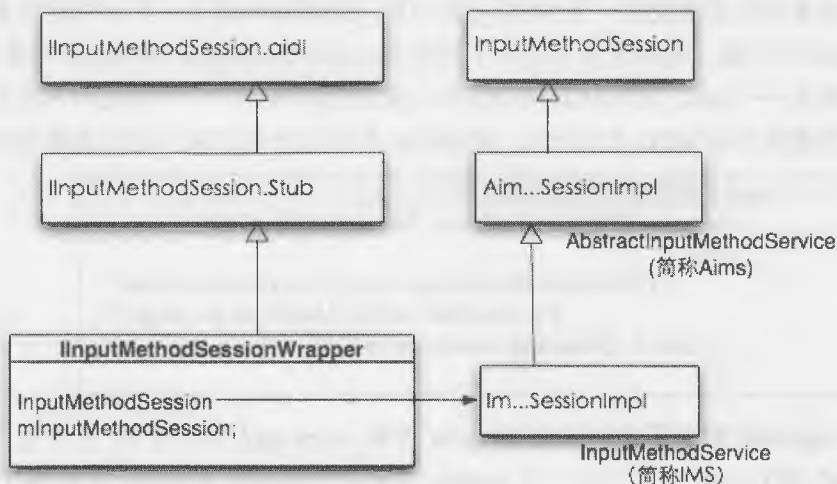


图 17-4 输入法服务端相关类继承和调用关系

图 17-4 中，IInputMethodSessionWrapper 是真正的 Binder 实现类，该类中包含了一个类型为 InputMethodSession 接口的变量 mInputMethodSession，该变量将会被赋值为真正实现了该接口的 InputMethodSessionImpl 类。这是一个子类，在 InputMethodService 类中定义。

下面分析该 Binder 对象是如何被创建的。

该 Binder 对象是 IME 向客户端提供的，只要用户没有切换输入法，那么该 Binder 对象对于固定客户端来讲就是不变的，无论客户端是否显示着输入法窗口，因此该 Binder 对象是在启动该输入法时创建的。

启动输入法时，会调用到 IMMS 的 startInput() 函数，该函数则会间接调用 startInputInnerLocked()，该函数内部会使用标准的 bindService() 方法启动输入法服务，如以下代码所示：

```

785 mCurIntent = new Intent(InputMethod.SERVICE_INTERFACE);
786 mCurIntent.setComponent(info.getComponent());
787 mCurIntent.putExtra(Intent.EXTRA_CLIENT_LABEL,
788     com.android.internal.R.string.input_method_binding_label);
789 mCurIntent.putExtra(Intent.EXTRA_CLIENT_INTENT, PendingIntent.getActivity(
790     mContext, 0, new Intent(Settings.ACTION_INPUT_METHOD_SETTINGS), 0));
791 if (mContext.bindService(mCurIntent, this, Context.BIND_AUTO_CREATE)) {
  
```



按照一般的 `bindService()` 的过程, 远程 Service 会启动, 并且回调调用者实现的 `ServiceConnect` 接口, 这就是为什么 IMMS 实现了 `ServiceConnect` 接口的原因, 如以下代码所示:

```
90 public class InputMethodManagerService extends IInputMethodManager.Stub
91     implements ServiceConnection, Handler.Callback {
```

在 IMMS 的 `ServiceConnection` 接口函数 `onServiceConnected()` 实现中, 做了两件事情。第一是发送一个 `MSG_ATTACH_TOKEN` 的消息, 该消息的作用是告知输入法服务该输入法窗口在 `WmS` 对应的 `token` 对象, 因为这个 `token` 是在 IMMS 中创建的, 名称为 `mCurToken`; 第二是判断当前是否有获得焦点的客户窗口, 如果有的话就发送一个 `MSG_CREATE_SESSION` 消息, 其作用是请求输入法服务返回 `Session` 对应的 `Binder` 对象, 直到这里才进入了创建 `IInputMethodSessionWrapper` 对象的实质性阶段。

关于第一个消息——`MSG_ATTACH_TOKEN` 这里暂且不谈, 下一节综述输入法启动过程将对此有详细介绍, 这里直接来分析 `MSG_CREATE_SESSION` 的消息处理过程。处理该消息的代码如下:

```
1358         case MSG_CREATE_SESSION:
1359             args = (HandlerCaller.SomeArgs)msg.obj;
1360             try {
1361                 ((IInputMethod)args.arg1).createSession(
1362                     (IInputMethodCallback)args.arg2);
1363             } catch (RemoteException e) {
1364             }
```

以上代码中 `args.arg1` 对应了输入法的 `Wrapper` 对象, `args.arg2` 有点特别, 这里使用的是远程回调的概念, 即 `args.arg2` 是 IMMS 中定义的一个 `Binder` 对象。当远程的 `Wrapper` 对象执行完 `createSession()` 方法时, 需要再回调到 IMMS 中的 `IInputMethodCallback` 接口, 而接口对象则是在创建 `MSG_CREATE_SESSION` 消息时创建的, 如以下代码所示:

```
740         executeOrSendMessage(mCurMethod, mCaller.obtainMessage00(
741             MSG_CREATE_SESSION, mCurMethod,
742             new MethodCallback(mCurMethod)));
```

即该接口的实现是 `MethodCallback` 类, 该类中的 `sessionCreate()` 函数正是被回调的函数, 该函数内部则调用了 IMMS 中的 `onSessionCreated()` 函数, 如以下代码所示:

```
445-     public void sessionCreated(IInputMethodSession session)
446         onSessionCreated(mMethod, session);
447     }
```

该函数中参数 `session` 正是在 IMMS 中创建的 `IInputMethodSessionWrapper` 对象, 下面接着分析 IMMS 中的 `createSession()` 是如何创建该 `Session` 对象的。这就要从 `IInputMethodWrapper` 中的 `createSession()` 函数说开了, 因为 IMMS 中的远程调用正是调用到该函数中。

当然, 该 `Wrapper` 内部首先采用的也是异步机制, 发送一个 `DO_CREATE_SESSION` 消息, 该消息的处理代码如下:

```
163         case DO_CREATE_SESSION: {
164             inputMethod.createSession(new InputMethodSessionCallbackWrapper(
165                 mCaller.mContext, (IInputMethodCallback)msg.obj));
```

在该代码中, msg.obj 正是 IMMS 中的远程 Callback 对象, InputMethodSessionCallbackWrapper 只是封装了该 callback 对象的一个普通类而已。这里可能让人混淆的是它的名称后缀也是“Wrapper”, 但它却不是一个 Binder 类, 因此下面可以称之为一个“假 Wrapper”。inputMethod 变量正是 InputMethod 接口的实现类, 对应 InputMethodService 中的 InputMethodImpl 类, 因此这里继续调用该类的 createSession() 函数, 实际上会调用到该类的父类 AbstractInputMethodImpl 的同名函数, 其代码如下:

```
626 public void createSession(SessionCallback callback) {
63     callback.sessionCreated(onCreateInputMethodSessionInterface());
64 }
```

这里采用了回调机制创建一个 AbstractInputMethodSessionImpl 类, 然后调用 callback 的 sessionCreated 把这个类返回给调用者。onCreateInputMethodSessionInterface() 函数在 InputMethodService 中进行了重载, 如以下代码所示:

```
688 public AbstractInputMethodSessionImpl onCreateInputMethodSessionInterface() {
689     return new InputMethodSessionImpl();
690 }
```

当然此时创建的仅仅是 InputMethodSession 接口的一个实现而已, 而并不是一个 Binder。因此接下来需要将这个接口封装成一个 Binder 对象返回给 IMMS, 而这正是上面 callback.sessionCreated() 要做的事情, callback 就是那个“假 Wrapper 对象”, 如以下代码所示:

```
84 public void sessionCreated(InputMethodSession session) {
85     try {
86         if (session != null) {
87             IInputMethodSessionWrapper wrap =
88                 new IInputMethodSessionWrapper(mContext, session);
89             mCb.sessionCreated(wrap);
90         } else {
91             mCb.sessionCreated(null);
92         }
93     }
```

在该代码中, 正是利用了 session 这个接口变量构造了一个 IInputMethodSessionWrapper 对象, 而这个 Wrapper 却是一个“真 Wrapper”。它是一个 Binder 对象, 创建好 Binder 对象后就调用 mCb 这个“远程回调”对象的 sessionCrated() 方法, 这就又回到 IMMS 中去了。

至此, IMS 中便成功地创建了 Session 对应的 Binder 对象, 并将其告知给 IMMS 中, 而在 IMMS 中还需要把这个 Binder 对象告知给客户端的 IMM, 从而使得以后客户端窗口可以直接经由 IMM 内部调用到 IMS 中的 Session Binder 对象。

#### 17.2.4 InputMethod

从代码的结构上看, InputMethod 十分类似 InputMethodSession, 尽管两者的功能完全不同。InputMethod 仅仅是一个 interface, 它定义了具体的输入法必须提供的 API 接口, 当然, 至于这些 API

如何被调用，以及调用后的返回结果如何传递给调用者它并不关心与之相关的类文件如下。

- InputMethod.java: interface 类。
- AbstractInputMethodService.AbstractInputMethodImpl: 该类是 InputMethod 的一个虚拟实现类。
- InputMethodService.InputMethodImpl: 该类是上面虚拟类的实现类。
- InputMethod.aidl: 该 Binder 对象的定义文件。
- IInputMethodWrapper: 该 Binder 的实现文件。

这里需要澄清一下 Service 与 Binder 的关系。在过去的应用程序开发时，大家都知道一个 Service 一般都会有一个 Binder 对象与之对应，但这个 Binder 对象不是凭空产生的，系统并没有自动为所定义的 Service 对象产生一个 Binder 对象。Service 的本质仅是一个 Context 类，和底层线程没有任何关系，而一个 Binder 对象一定会在底层创建一个线程。Service 类的特别之处在于，它内部定义了一个 onBind() 方法，程序可以重载该方法，并返回一个 Binder 对象，AmS 内部会把这个 Binder 对象返回给调用者。而 aidl 则会产生一个 Binder 接口，实现者需要实现其中的函数定义，从而完成真正的 IPC 调用。从这个角度来看，Service 仅仅充当了一个“标准邮递员”的角色，即它按照 AmS 所定义的接口标准传递一个 Binder 对象到调用者，仅此而已。另外，从系统权限的角度来看，如果 ServiceManager 允许普通的应用程序来注册服务，那么 Service 类就没有存在的必要了。因为普通应用可以直接注册所定义的服务，然后调用者再从 ServiceManager 中 getService() 即可。然而，出于安全的考虑，AmS 需要对客户端的 Service 进行某种管理，包括启动和关闭，因此，定义了 Service 类，调用者必须在 AmS 的监督下获取 Service 所提供的 Binder 对象。

以上几个类的关系如图 17-5 所示。

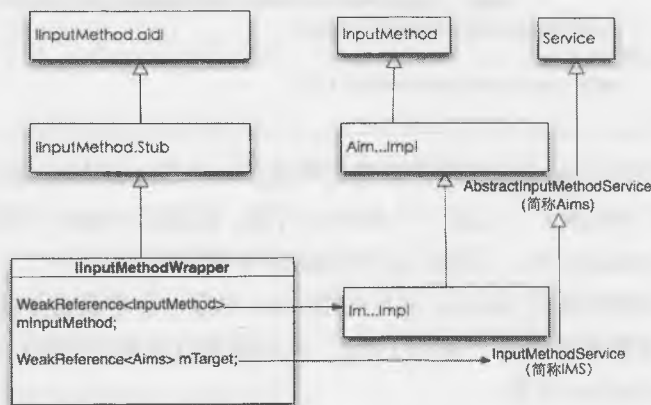


图 17-5 输入法模块内部的类关系

图 17-5 中，IInputMethodWrapper 类是真正的 Binder 类，该类中包含两个重要变量。一个是 mInputMethod，它指向真正实现了 InputMethod 接口的类 InputMethodImpl，该类是一个子类，在 InputMethodService 类中定义；另一个是 mTarget，其类型是 AbstractInputMethodService，该类的父类是

Service 类，实际运行时，该变量是一个 InputMethodService 对象，而该类的父类则是 AbstractInputMethodService。

一个输入法对象对应一个 IInputMethodWrapper 对象，而且该输入法对象在系统启动后任何一个用户窗口获得焦点后就会启动，因此在一般情况下，应用程序调用输入法时都不会执行创建该对象的动作，而有一种情况例外，那就是切换到新的输入法。同时，该 Binder 对象仅被 IMMS 调用，而不直接被 IMM 或者客户窗口调用。

该 Binder 对象的创建过程比较简单。在 IMMS 中的 startInputInnerLocked() 函数中，调用 bindService() 启动指定的输入法，而在输入法的 AbstractInputMethodService 的 onBind() 方法中创建了一个 IInputMethodWrapper 对象，如以下代码所示：

```

184     final public IBinder onBind(Intent intent) {
185         if (mInputMethod == null) {
186             mInputMethod = onCreateInputMethodInterface();
187         }
188         return new IInputMethodWrapper(this, mInputMethod);
189     }

```

### 17.3 输入法主要操作过程

上一节介绍了 IMF 中相关的 Binder 类的作用及其自身的创建过程，由于各 Binder 的创建过程彼此有内在的联系，因此上一节分离式介绍显得有点纷繁庞杂，所以本节将从其内在联系的角度分析各模块，并且从输入法操作的角度来分析输入法整体的内在逻辑。

输入法操作分为三个部分，分别为切换输入法、启动输入法和显示输入法。读者注意区分“启动”和“显示”。启动是指启动输入法服务（IMS），启动后 IMS 就已经知道了客户端的 Binder 对象，客户端也知道了 IMS 中的 Session 类 Binder 对象。而“显示”仅仅是指把输入法窗口显示出来，以便用户可以通过触摸屏上的虚拟键盘向客户窗口传递字符，但事实上该窗口是否显示出来并不会影响 IMS 和客户端的通信。比如，对于有的键盘 Android 设备，往往不显示虚拟键盘，此时用户可以直接通过按键输入字符，表面上似乎是直接把按键消息送入编辑框，实际上还是经过了输入法。

因此，本节中第一小节将介绍输入法启动过程中的各模块内在调用关系，后面三个小节介绍三种不同的输入法操作，为便于理解源码，最后一个小节对源码中相关类中的主要变量作一个语义上解释和总结。

#### 17.3.1 输入法相关模块的启动过程

前面说过，输入法实际上是由各 Binder 对象组成的，每个 Binder 对象即是输入法的模块，各 Binder 对象本身有其自身内部创建过程，同时各 Binder 对象彼此之间又有一定的创建先后顺序。

为了叙述的方便，先对各 Binder 对象进行称呼的约定。

- Session Binder 或者 Session: 指的是 IMS 中为客户端创建的 Binder 对象, 以便客户端可以直接和 IMS 进行通信。
  - Connection Binder 或者 Connection: 指的是客户端创建的 Binder 对象, IMS 将调用该 Binder 对象向客户端传递输入消息。
  - InputMethod Binder 或者输入法 Binder: 这是 IMS 对应的 Binder 对象。
- 总的创建顺序如图 17-6 所示。



图 17-6 输入法的启动流程

在图 17-6 中, 启动输入法是在客户窗口变成当前交互窗口时开始执行的, 关于其细节, 下面小节将有介绍。显示输入法则当某个具有编辑属性的视图获得焦点后或者程序员调用 IMM 的 `showSoftInput()` 时开始执行。

### 17.3.2 切换输入法

在介绍输入法操作之前, 先来介绍输入法是如何切换的, 因为这关系到 IMM 及 IMMS 中的几个重要变量的赋值, 而这几个变量在后面的输入法操作中很重要。

输入法是以 APK 文件的形式安装到系统中的，安装后，在某个输入框中，用户可以长按编辑框，弹出一个窗口，窗口中的最后一项就是选择输入法选项，用户可点击进去选择不同的输入法，选择后，系统会把当前的输入法存在 Setting 中。下面来分析这个过程在代码中是如何实现的。

首先来看长按编辑框的操作。这在 View 类的 performLongClick() 函数中，其内部实现则会回调到 TextView 类的 onCreateContextMenu()，该过程属于标准的 View 操作过程，详情参见第 13 章。

而在 onCreateContextMenu() 函数中，则会判断视图对象是否能够输入文字，如果可以的话则给选项菜单中增加一个切换输入法的选项，如以下代码所示：

```
7484         if (isInputMethodTarget()) {
7485             menu.add(1, ID_SWITCH_INPUT_METHOD, 0, com.android.inter
7486                 setOnMenuItemClickListener(handler);
7487             added = true;
7488         }
```

代码中，isInputMethodTarget() 的代码如下：

```
7509     public boolean isInputMethodTarget() {
7510         InputMethodManager imm = InputMethodManager.peekInstance();
7511         return imm != null && imm.isActive(this);
7512     }
```

代码中，imm.isActive(this) 的参数 this 指的就是当前的 View 对象。isActive() 用于判断当前输入法是否和该 View 对象绑定在一起，输入法只能和具有输入属性的视图绑定在一起。

当用户选择“切换输入法后”，则会调用 IMM 的 showInputMethodPicker() 显示一个输入法列表的菜单框，如以下代码所示。

```
7596         case ID_SWITCH_INPUT_METHOD:
7597             InputMethodManager imm = InputMethodManager.peekInstance();
7598             if (imm != null) {
7599                 imm.showInputMethodPicker();
7600             }
7601             return true;
```

showInputMethodPicker() 实际上则转而调用 IMMS 的 showInputMethodPickerFromClient() 函数，换句话说，用户所看到的输入法列表窗口实际上是一个系统级别的对话框，是由 IMMS 创建的。在 IMMS 内部，该函数是异步执行的，即先发送一个 MSG\_SHOW\_IM\_PICKER 消息，消息的具体执行函数是 showInputMethodMenu()，该函数的执行过程如下。

① 获取当前正在使用的输入法名称。从 Setting Provider 中读取名称为 Settings.Secure.DEFAULT\_INPUT\_METHOD 的字段，该字段的返回值类型是一个字符串，比如 com.android.inputmethod.pinyin/.PinyinIME，它是用输入法的包名加输入法 Service 的类名共同组成的，读取后存放到临时变量 lastInputMethodId 中。

② 获取所有使能的输入法名称列表。从 Setting Provider 中读取名称为 Settings.Secure.ENABLED\_INPUT\_METHOD 的字段，该字段的返回值也是一个 String 类型，该 String

是用“冒号”对每一个输入法名称进行分隔。这里需要注意，“使能”的输入法并不等于已经安装的输入法，比如，系统中可以安装韩文、日文、泰文输入法，但如果你是中国用户可能只希望开启谷歌输入法和搜狗输入法，并在这两个输入法间进行切换，这就是所谓的“使能”。要使能一个输入法，可以在系统设置界面中进行操作，如图 17-7 所示。



图 17-7 在系统设置中使能输入法

而在 IMMS 系统初始化时，会首先读取所有已经安装的输入法列表，并将其保存到 `mMethodMap` 列表中，该列表的键（Key）是输入法名称，其值（Value）是一个 `InputMethodInfo` 类。该步骤中，正是从 `Setting` 字段获取使能名称列表，并根据这个列表从 `mMethodMap` 中找到相应的 `InputMethodInfo`。

③ 创建一个 `AlertDialog`，该对话框中使用一个单选列表作为对话框的内容。具体是调用 `AlertDialog` 的 `setSingleChoiceItems(mItems, ...)` 函数，该函数的第一个参数类型为 `CharSequence[]`，即代表每一个单选的名称，它正是通过第二步中获取的 `InputMethodInfo` 列表获得的。

④ 该 `AlertDialog` 的单选消息响应处理中则调用 `setInputMethodLocked()`，该函数中会把新选的输入法名称保存进 `Setting Provider` 中，并给变量 `mCurMethodId` 赋值新选的输入法名称。然后再发送一个输入法改变的广播消息，名称为 `ACTION_INPUT_METHOD_CHANGED`。最后调用 `unbindCurrentClientLocked()` 断开和上一个输入法连接的客户端，该函数的处理过程如下。

(1) 判断当前是否有输入法启动，即 `mCurMethod` 是否为空。如果有，则发送 `MSG_UNBIND_INPUT` 消息，通知该输入法关闭。消息处理中，远程调用 `IMS` 的 `unbindInput()` 函数。这里注意，调用后，并没有把变量 `mCurMethod` 置空。

(2) 发送 `MSG_UNBIND_METHOD`，通知和当前输入法有连接的客户端断开连接。消息处理中，远程回调 `IMM` 的 `onUnbindMethod()` 方法。

### 17.3.3 启动输入法

在源码中有多处类似于 `startInputXXX` 的函数，因此首先需要澄清“启动”的语义，它与这些包含 `start` 名称的函数的关系如何呢？

从 `IME` 的角度来看，“启动输入法”的语义包括以下两点，即启动 `IMS` 所在 `Service`，并要求 `IME`



创建 Session Binder。关于这个过程，上一小节已经介绍。

从用户的角度来讲，“启动输入法”的语义是指当用户点击一个编辑框时，界面上会显示一个输入法窗口，但这个过程和程序内部的启动过程有所不同。

而从程序员的角度来讲，程序员不能直接调用 IMMS 及 IMS，而只能通过 IMM 控制输入法的操作，所以 IMM 所提供的 start 函数即是所谓的“启动”。

在 IMM 中和 start 名称相关的重要函数有两个，一个是 public 型的 restartInput()，另一个是 private 型的 startInputInner()，但却没有 public startInput() 函数。除了这两个函数外，IMM 还提供了 showXXX() 和 hideXXX() 函数，用于显示和隐藏输入法窗口。从这点可以看出，IMM 给调用者的概念是：“我不需要你启动，不过你可以让我重新启动，当然你也可以显示或者隐藏输入法窗口。”那么，为什么输入法不需要程序员来启动呢？因为，Android 认为，启动输入法应该是自动完成的，即当某个视图（比如编辑框）获得焦点时就应该自动启动。如果该视图是可编辑的，那么同时显示输入法窗口，而如果不是可编辑的，则暂时隐藏输入法窗口。showXXX() 和 hideXXX() 给程序提供了手工显示、隐藏输入法的开关。

那么，系统是如何自动启动输入法的呢？这就得从 WmS 切换到新窗口开始说起。用户启动一个新的 Activity 时，或者退出当前 Activity，或者关闭一个系统窗口时，等等，都会导致 WmS 进行窗口切换，一旦切换完成，就有一个新的窗口成为交互窗口。此时 WmS 通过 IPC 回调，通知客户端相应的客户端“发生了焦点窗口改变”onWindowFocusChanged()，这里 Focus 的含义确切地讲并不是和用户交互的窗口，而只是指能和输入法进行交互的窗口。比如，状态栏窗口是可以和用户交互的，但是却不能获得按键消息，因此，它永远不可能成为一个 Focus 窗口，如果这个函数名称改为 onInputWindowFocusChanged()，大家可能更容易理解。

而一旦客户窗口获得该消息后，就开始启动输入法的过程，整体过程参见附图 11，以下为了叙述方便，截取相关部分分别描述。下面先从客户窗口获得 Focus 后的处理开始，如图 17-8 所示。

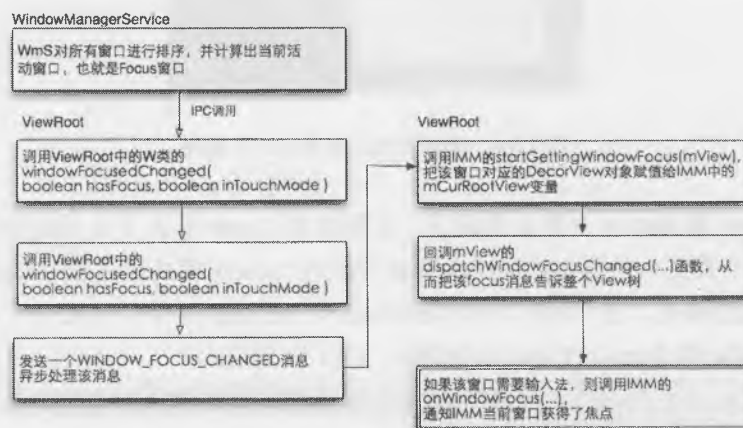


图 17-8 编辑框获得 Focus 后的处理流程

客户端窗口对应的是一个 `ViewRoot` 对象，该对象内部包含一个 `mView` 变量，该变量一般是 `PhoneWindow.DecorView` 类型或者其他自定义的 `ViewGroup` 类型。`WmS` 调用过来时，首先经过 `W` 子类这个 `Binder` 对象获取调用，其相应的函数内部再调用了 `ViewRoot` 中的 `windowFocusedChanged()` 方法，该方法发送一个 `FOCUS` 消息。这种处理是 `W` 子类中所有函数的处理方式，从而使得 `WmS` 不必等待客户窗口的执行过程，而可以立即返回以便处理其他事物。

在 `ViewRoot` 中处理 `WINDOW_FOCUS_CHANGED` 消息的函数中主要做了三件事情。

① 把新的窗口对应的根 `View` 视图对象传递给 `IMM`，毕竟一个应用程序可以有多个窗口，而只有一个 `IMM`，`IMM` 必须知道当前哪个窗口正在和输入法交互，`IMM` 中使用 `mCurRootView` 变量保存这个 `View` 对象。

② 通知窗口中的每一个子 `View`，好比说：“孩子们，我们家族的窗口重新获得或者失去焦点了”，以便孩子们能够对此做出反应。在默认的操作中，如果某个子 `View` 在该 `View` 树中拥有当前焦点，并且该 `View` 具备编辑属性，那么它就会成为 `IMM` 中变量 `mServedView` 的新值。

注意区别窗口的 `Focus` 和 `View` 树中某个 `View` 的 `Focus` 两者的区别。`WindowFocus` 是指该窗口是否和输入法进行交互，而 `ViewFocus` 是指和按键的交互，尤其是当用户按上下左右键时，获得焦点的 `View` 一般会高亮起来。在以往的经验中大家是否发现，如果创建如图 17-9 所示的一个窗口，该窗口包含一个 `TextView`、一个 `Button` 和一个 `EditView`，当窗口显示时，编辑框会自动高亮起来，你是否觉得奇怪，为什么高亮的不是那个 `Button`？



图 17-9 编辑框默认获得焦点并高亮

原因是，`TextView` 和 `Button` 在默认情况下，其属性在 `Touch` 模式下是不能获得 `Focus` 的，而窗口在启动时，内部会挨个找到一个窗口，而 `EditText` 无论在 `Touch` 模式还是非 `Touch` 模式下都可以获得 `Focus`，它在初始时就会高亮。在 `View` 类中的 `onWindowFocused()` 函数默认处理中，正是利用了这个属性，判断是否调用 `IMM` 的 `focusIn(View this)` 函数，如以下代码所示：

```
3949         } else if (imm != null && (mPrivateFlags & FOCUSED) != 0) {
3950             imm.focusIn(this);
```

调用 `focusIn()` 后，将会导致向 `IMM` 中发送一个 `CHECK_FOCUS` 消息，而该消息会导致调用

checkFocus()函数, 该函数的作用见第 ③ 步。

③ 判断该窗口是否需要输入法。这个有点意思, 什么叫做“是否需要输入法”? 它的意思是, 如果一个窗口中没有可编辑的视图, 那么该窗口就是不可输入的。从程序的角度来讲, 如果一个窗口的所有视图都是非 FOCUSABLE 的, 那么该窗口就是不可输入的。该步骤中, 判断是否需要输入法的代码如下:

```
1959         mLastWasImTarget = WindowManager.LayoutParams
1960             .mayUseInputMethod(mWindowAttributes.flags);
```

而在 mayUseInputMethod() 内部则判断 mWindowAttributes 变量中是否包含 FLAG\_NOT\_FOCUSABLE 等相关属性。mLastWasImTarget 变量的含义是“该窗口是否可以作为输入法(IM)的目标”。

如果该窗口是可输入的, 那么该步骤中则调用 IMM 的 onWindowFocus(...), 该函数内部的执行流程如下。

(1) 调用 focusInLocked()函数。该函数的名称有点晦涩, 不明含义, 但其内部实际上做了两件事情, 一个是把参数中的 View 赋值给 mNextServedView 变量, 另一个是判断 ViewRoot 中是否有 CHECK\_FOCUS 消息, 如果没有则发送一个, 这会异步调用 checkFocus()函数。

(2) 调用 checkFocus()函数, 该函数中会先检查 served 视图是否变化。比如, 如果用户在同一个 TextView 上连续点击时, 也会执行到 checkFocus()函数, 但是由于视图没变化, 因此该函数会立即返回, 检查的条件是对比 mServedView 和 mNextServedView。当第一次调用 checkFocus()函数时, 这两个变量值不相同, 而执行后, mServedView 被赋值为 mNextServedView。除了这个条件外, 还利用了变量 mNextServedNeedStart, 如以下代码所示:

```
1084         synchronized (mH) {
1085             if (mServedView == mNextServedView && !mNextServedNeedsStart) {
1086                 return;
1087             }
```

该变量在 IMM 对象初始化时赋值为 true。当 IMM 和一个新的输入法绑定时, IMMS 会回调 IMM 变量 mClient 中的 setActive()函数, 该函数中将该变量置为 false, 并在该程序没有退出的情况下或者没有切换输入法的情况下会始终保持为 false。所以, 该变量代表了是否需要等待输入法重启, 只是这个名称有点晦涩, 没有确切代表这个意思。

如果焦点视图发生了变化, 那么该函数会调用 startInputInner()。

(3) 调用 IMMS 的 windowGainFocus(...)函数, 该函数的参数除了包含新的窗口信息外, 还包含了窗口是否获得焦点、视图是否可编辑等信息, 从而使得 IMMS 一方面可以更新内部变量 mCurClient 和 mCurFocusedWindow, 另一方面, 更新完毕后还可以判断是否为焦点视图显示输入法窗口。

接下来分析 startInputInner()函数的调用过程。

在 IMM 中, 能导致调用到该 private 函数的情况有四种, 如图 17-10 所示。

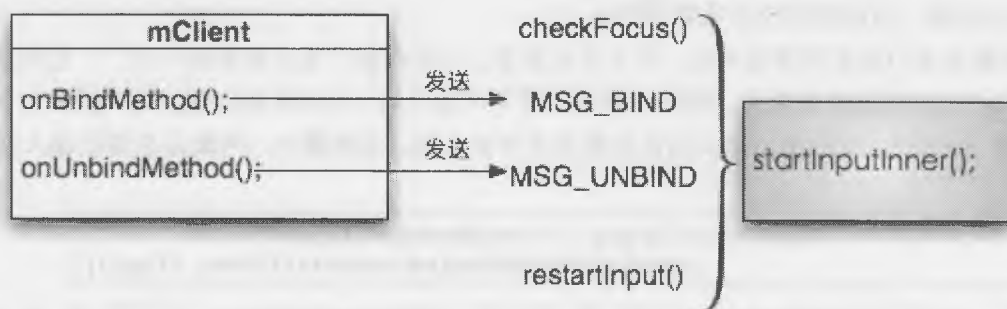


图 17-10 导致调用 startInputInner()的四种情况

其中, checkFocus()虽然是 public 类型,但却被标记为@hide, SDK 中看不到该函数,该函数只能被 Framework 内部及 IMM 内部的其他函数调用;两个 MSG 消息是 IMMS 的 IPC 回调 mClient 对象中的 onBind()和 onunbind()时产生的,这两个函数如果改名为 onSessionBind()和 onSessionUnbind()大家则更容易记住其作用;restartInput()可以被程序员调用,系统内部没有直接调用该函数,一般是当编辑框的内容在输入法之外被修改,此时应该通知输入法重新启动,以便输入法获知编辑框中的新的内容。

startInputInner()函数完成的任务主要包含三大部分,如图 17-11 所示。

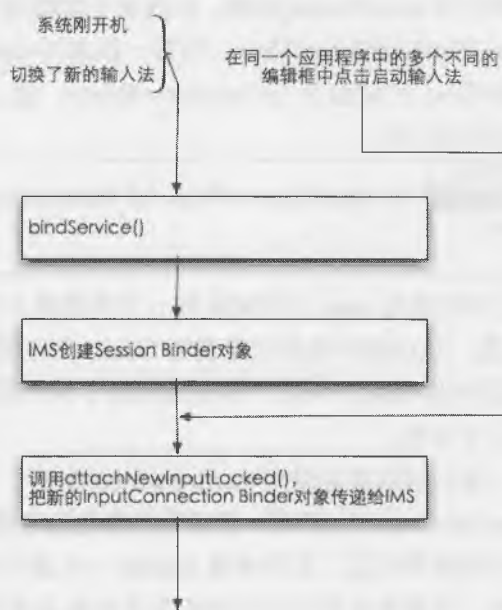


图 17-11 startInputInner()中完成的三个主要任务

这三大任务主要是:

第一,在 IMSS 中调用 bindService(),启动 IMS 中包含了 InputMethodService 对象。

第二，在 IMMS 的 `onServiceConnected()` 回调中，调用 IMS 的 `createSession()` 方法要求 IMS 创建一个 Session Binder 对象，随后 IMMS 会把这个 Binder 对象传递到客户进程中的 IMM 对象中。

第三，在 IMMS 中调用 `attachNewInputLocked()`。正如该函数名称所示，这是一个新的 Input 对象，即一个新的编辑框。因为每个不同的编辑框都有各自不同的 InputConnection Binder 对象，因此，每次当用户从一个编辑框切换到另一个编辑框时，都需要把新的 Binder 对象传递到输入法中，在该函数的参数中包含了新的编辑框对应的 Binder 对象 `mCurInputConext`。

以上三个任务包含在 `startInputInner()` 函数中，但并不是只要调用该函数三个任务就都会被执行，因为没有这个必要，所以该函数内部会根据一些重要的变量值决定是否执行相应的任务。在多数情况下，只有第三个任务被执行，在所有可能的操作中，包含了三种情况，具体如下。

- 系统刚开机，并启动第一个窗口时，此时由于输入法没有启动，因此需要根据 Setting 中保存的输入法名称启动相应的 IMS。IMMS 是如何知道输入法还没有启动的呢？这是通过三个变量完成的，如以下代码所示：

```
735         if (mHaveConnection) {
736             if (mCurMethod != null) {
737                 if (!cs.sessionRequested) {
```

变量 `mHaveConnection` 代表是否已经调用 `bindService()` 启动了 IMS，`mCurMethod` 是 IMS 的 Binder 对象，而 `cs.sessionRequested` 代表了是否已经请求 IMS 创建 Session Binder 对象。当这三个条件都满足时，已经调用过了 `bindService()`，并且该 Service 也已经启动了，但却还没有来得及请求 IMS 创建 Session Binder。事实上这种情况很难发生，除非程序中连续两次调用 `startInputInner()`，而如果这种情况出现，IMMS 的 843 行有一个 bug，这个地方是在 `onServicedConnected()` 回调函数中。即准备请求 IMS 创建 Session Binder 对象，在发出这个 `MSG_CREATE_SESSION` 消息之前，需要先给变量 `mCurClient` 的 `sessionRequested` 变量赋值为 `true`，但源码中却没有。

- 在使用的过程中，用户切换了新的输入法，此时，应用窗口被重新激活，都必须重新绑定新的 IMS。
- 在多数情况下，用户的操作仅仅是在不同的编辑框上来回切换，此时只需要把新的 InputConnection Binder 告诉给 IMS 即可。这里值得思考的是，即使在同一个界面中，点击该界面不同的编辑框也会导致重新创建一个 InputConnection Binder 对象。这显然有点浪费，除非创建 Binder 的代价非常小，因此框架中的这个地方有优化的空间。

关于 `startInputInner()` 中 3 个主要任务的内部流程细节参照附图 11，图中有详细的描述。

#### 17.3.4 显示输入法

上一节介绍了启动输入法的过程，而显示输入法比较简单，仅仅是告知 IMS 中显示输入法窗口而已。显示输入法的时机有三个。

- 当应用窗口成为当前交互窗口时，此时经过各种调用后会调用到 IMMS 的 `windowGainedFocus()` 函数，具体过程参见附图 11。
  - 当程序员主动调用 IMM 的 `showSoftInput()` 函数时。
  - 当点击一个编辑框时，系统会自动弹出输入法窗口。
- 其中第二种和第三种简要过程对比如图 17-12 所示。

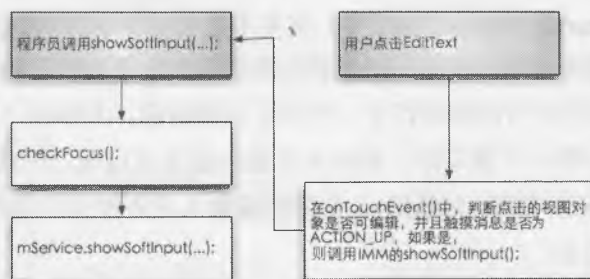


图 17-12 两种显示输入法的方式

无论以上哪种情况，其归根结底都是通过 IMMS 调用 IMS 中的 `showSoftInput()` 函数，因此，本节仅介绍第二种。

当客户端要显示输入法窗口时，会调用 IMM 的 `showSoftInput()` 函数，该函数中则会调用 IMMS 的同名函数。在 IMMS 中采用的是异步机制，即先发送一个 `MSG_SHOW_SOFT_INPUT` 的消息，发送消息时，调用了 IMMS 的内部函数 `executeOrSendMessage()`，该函数的参数使用 `mCaller.obtainMessageIOO()` 函数进行创建。`mCaller` 是一个 `HandlerCaller` 类，`obtainMessageIOO` 中的 `IOO` 代表了消息参数的类型，`I` 为 `int` 类型，`O` 为 `Object` 类型，两个 `O` 代表两个 `Object` 类型，与此相似的还有 `OO`, `OOO` 等。由于 `Handler` 类中定义的消息参数只能有四个，分别是 `int what`、`int arg1`、`int arg2` 和 `Object obj`，因此，这种 `IOO` 或者 `OOO` 类型实际上是把所有的 `Object` 参数封装到一个 `SomeArgs` 子类中，并将该 `SomeArgs` 对象作为 `Handler` 的 `obj` 变量。

处理 `MSG_SHOW_SOFT_INPUT` 消息的代码如下。

```

1334     case MSG_SHOW_SOFT_INPUT:
1335         args = (HandlerCaller.SomeArgs)msg.obj;
1336         try {
1337             ((IInputMethod)args.arg1).showSoftInput(msg.arg1,
1338                 (ResultReceiver)args.arg2);
1339         } catch (RemoteException e) {
1340             }
1341         return true;
  
```

参数 `args.arg1` 来自于 IMMS 中的 `mCurMethod` 变量，它是 `IInputMethod` 类型，指的就是当前输入法服务的 `Binder` 对象；`msg.arg1` 是一个 `int` 型的 `flags`，它从 `showFlags()` 函数中获取，该函数内部根据 `mShowForced` 变量返回不同的值；`args.arg2` 参数来源于 IMM 调用时使用的一个 `Parcelable` 对象，从而可以把执行结果通过该对象返回到 IMM 中。

接下来就会执行到当前 IMS 中的同名函数 showSoftInput() 中, IMS 具体对应了 IInputMethodWrapper 类。该函数的执行也采用了异步机制, 即先发送一个 DO\_SHOW\_SOFT\_INPUT 的消息, 消息的处理代码如下。

```
175     case DO_SHOW_SOFT_INPUT:
176         inputMethod.showSoftInput(msg.arg1, (ResultReceiver)msg.ob
177         return;
```

其中 inputMethod 变量正是该 Wrapper 的内部变量 mInputMethod, 它是 InputMethod 接口的一个实现类而已, 并不是 Binder 对象, 仅仅是一个功能类。该变量是在该 Wrapper 对象构造函数中赋值的, 而该 Wrapper 对象又是在 InputMethodService 的父类 AbstractInputMethodService 中的 onBind() 函数中创建的, 如以下代码所示。

```
184     final public IBinder onBind(Intent intent) {
185         if (mInputMethod == null) {
186             mInputMethod = onCreateInputMethodInterface();
187         }
188         return new IInputMethodWrapper(this, mInputMethod);
189     }
```

其中 mInputMethod 则是通过回调 onCreateInputMethodInterface() 进行创建的, 而该回调函数则是在 InputMethodService 类中实现的, 如以下代码所示。

```
680     public AbstractInputMethodImpl onCreateInputMethodInterface() {
681         return new InputMethodImpl();
682     }
```

所以说, Wrapper 中处理 DO\_SHOW\_SOFT\_INPUT 消息的代码中, inputMethod 变量是 InputMethodImpl 类的一个实例。以上描述听来有点像绕口令, 读者可参照图 17-13 进行对照理解。

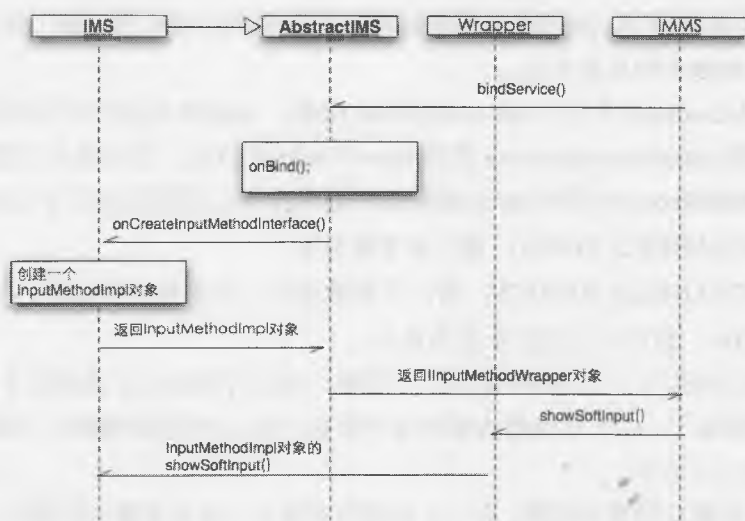


图 17-13 IMS 和 IMMS 模块的通信过程



图 17-13 中, InputMethodService(IMS) 的父类是 AbstractIMS, 而父类中定义了很多 onXXX() 回调函数, 实例类中需要重载的仅仅是这些回调而已。这种设计模式与普通的面向对象有所不同, 普通的面向对象中, 如果 A 是父类, B 是子类, B 基本上可以全部重载 A 的方法, 以实现一个具体的功能。而在操作系统设计中, 面向对象的回调模式揭示了目标架构的一个本质: “不是我们在调用 SDK, 而是操作系统在调用我们”。该模式的一般架构如图 17-14 所示。

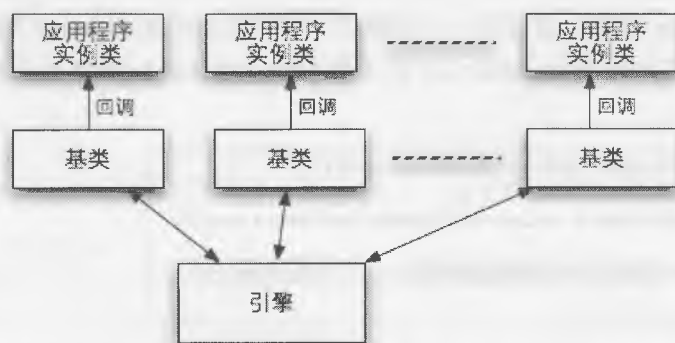


图 17-14 通用操作系统中的回调方式

该模式的特点有三: 第一是架构中存在一个引擎, 从程序上看它是一段代码, 这段代码不能被应用程序调用、重载和执行, 而是在后台自动执行的; 第二是基类中定义了一些标准的回调函数, 具体的实例类一般只需要重载这些回调函数即可; 第三是基类中的一些非 public 函数要实现很多功能, 以便和调度引擎进行配合, 所以, 基类一般显得比较庞大, 比如 View 类的代码有近一万行。

这种设计模式在 Android 中很多地方都在使用, 包括 IMF 中 IMS 端的设计, 还有 Activity 架构的设计, 还有 View 类的架构设计。应用程序开发时表面上是在调用 SDK 的 API, 而实际上则是操作系统通过回调调用程序员所提供的具体实现。

继续来看 InputMethodImpl 类中的 showSoftInput() 函数, 该函数内部分两步执行。

- ① 调用其所在的 InputMethodService 类的 showWindow() 方法, 显示输入法窗口。
- ② 调用参数 ResultReceiver 类的 send() 函数返回显示结果, 结果包括以下几项。
  - RESULT\_UNCHANGED\_SHOW: 窗口本来就显示。
  - RESULT\_UNCHANGED\_HIDDEN: 窗口不能被显示, 本来也没有显示。
  - RESULT\_SHOW: 窗口从没有显示变为显示。

输入法窗口实际上对应了一个 Window 类, 上面第一步执行的结果正是给这个 Window 类内部添加视图, 并把它显示到屏幕上。由于该函数内部涉及更多关于窗口内视图的操作, 因此单独放在一节进行介绍, 详情参见第 17.4.1 小节。

以上介绍了输入法窗口的显示过程, 有一个问题值得思考, 那就是输入法窗口应该显示在屏幕上的什么位置呢? 从 IMMS 的接口来看, 显示窗口时要么调用该类中的 windowGainFocus(), 要么调用

showSoftInput(), 但这两个函数的参数中都没有包含焦点视图的位置信息。这意味着, Android 中的输入法窗口位置不能根据编辑框的位置自动改变, 而只能在固定区域上显示。这与 PC 上的输入法有很大不同, PC 的输入法窗口可以根据当前光标位置动态改变, 而 Android 中却是固定的, 一般只能显示在屏幕的下半部分, 如图 17-15 所示。

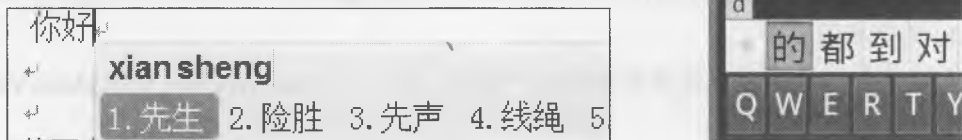


图 17-15 PC 输入法和 Android 输入法的窗口对比

不过大家在 HTC 品牌的 Android 手机上却发现, 其自带的中文 TouchInput 输入法的备选窗口却可以根据编辑框的位置动态改变, 如图 17-16 所示的输入备选窗口。它的原理有可能是 IMS 先利用了 InputConnection 的 performPrivateCommand() 函数向客户端发送一个请求获取编辑框位置的命令, 然后客户端再调用 IMS 中 InputMethodSession 的 appPrivateCommand() 函数传递编辑框的位置信息, 从而根据编辑框的位置动态改变输入法备选窗口的位置。



图 17-16 备选输入词组窗口位置动态变化的情况

### 17.3.5 输入法操作过程中的重要变量总结

为了便于阅读源码, 本小节将输入法操作过程中相关的一些重要状态变量的语义做一个集中的、简要的说明。

#### 1. IMM 中的重要变量

在以上操作中, 相关的几个变量的含义描述如下。

- mCurId: String 类型, 当前正在使用的, 并且是已经启动了输入法名称。

- **mCurMethodId**: 当前应该使用的输入法名称, 不过该输入法可能还没有启动。在 `startInputLocked()` 函数中, 会判断该变量是否和 `mCurId` 相等, 如果相等, 说明输入法没有改变, 如果不相等, 说明输入法已经改变, 但新的输入法还没有启动。
- **mCurMethod**: `Binder` 类型, 表示远程正在连接的 `IMS`。
- **mHaveConnection**: 是否已经启动了 `mCurId` 中指定的输入法, 启动后 `mCurMethodId` 的值将和 `mCurId` 的值相同。
- **mCurRootView**: 当前输入法所连接的客户窗口, 是一个 `PhoneWindow` 中的 `DecorView` 对象。

## 2. IMMS 中的重要变量

- **mCurClient**: 当前客户窗口所在进程中的 `IMM` 对象。
- **mCurFocusedWindow**: 当前焦点窗口, 该窗口正在和输入法进行交互。
- **mCurInputContext**: `Binder` 对象, 和当前输入法连接的客户视图对应的 `Binder` 对象。

## 3. IMS 中的重要变量

- **mWindowAdded**: 输入法窗口是否已经添加。
- **mWindowCreated**: 作用同 `mWindowAdded`。
- **mWindowVisible/mWindowWasVisible**: 当前/上一次的窗口显示状态。
- **mShowInputRequested**: 是否请求显示输入法窗口。
- **mInputStarted**: 目标编辑框包含的 `Connection Binder` 是否已经传递到 `IMS` 中。
- **mInShowWindow**: 表明当前是否正在执行显示窗口的操作, 当调用 `showWindow()` 时赋值为 `true`, 调用完毕后即为 `false`。
- **mIsInputViewShow**: 输入区是否已经显示。
- **mShowInputForced**: 当调用 `showSoftInput()` 时, 如果参数的 `flag` 中包含 `SHOW_FORCED`, 则改变量为 `true`。

## 17.4

## 输入法窗口内部的显示过程

前面介绍了输入法的整个启动过程, 一旦输入法启动后, 无论是后台自动的还是程序调用的, 都会通知输入法显示其所包含的输入法窗口, 以便用户输入文字。对于 `IMS` 来讲, 显示一个输入法窗口就如同普通应用程序显示自己的界面视图一样, 只是这个界面包含了一个虚拟键盘而已。

本节包含三个小节。第一小节中介绍 `IMS` 内部是如何构造输入法界面。由于 `IMS` 内部已经定义了默认的输入法界面组成, 因此对于一般的输入法而言, 只需基于这个标准界面框架设计自己的输入法, 但 `IMF` 同时允许输入法厂商完全自定义输入法, 只是厂商需要编写更多的代码而已。后面两个小节分别介绍编写一个简单的输入法和完全自定义输入法的实现思路, 以供大家参考。

### 17.4.1 IMS 中的 showWindow()的内部执行过程

该函数本身属于 IMS 的函数,不能被远程调用。IMMS 首先调用 Wrapper 中的 showSoftInput()函数,然后再调到 IMS 中子类 InputMethodServiceImpl 的同名函数,该过程可参见第 17.3.4 节,之后才会调用到 IMS 的 showWindow()函数真正开始显示输入法窗口。本节主要介绍 IMS 中的 showWindow()函数的内部执行过程。

接下来先来看如何从 IMS 中的子类 InputMethodServiceImpl 中的 showSoftInput()调用到 showWindow()函数,该过程如图 17-17 所示。

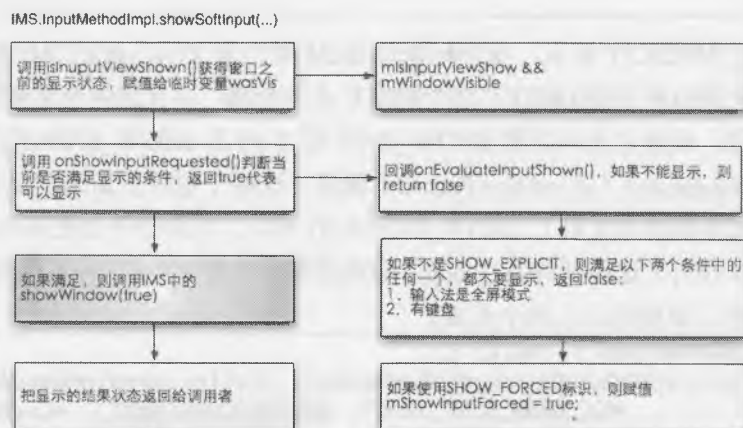


图 17-17 showSoftInput()函数内部执行过程

① 调用函数 isInputViewShown()判断当前窗口是否已经显示。该函数中则利用 mIsInputViewShown 变量和 mWindowVisible 变量,这两个变量的值基本总是相同,从这点上也可以看出,IMS 的源码中,内部变量的定义不是很精简,有些变量的意义有重复。

② 调用 onShowInputRequested()判断是否真的需要显示输入法窗口,该函数的内部逻辑如下。

(1) 调用 onEvaluateInputShown()函数中已经判断过系统是否没有按键,如以下代码所示。该段代码的语义是,如果系统没有键盘或者键盘被隐藏了,那么就onEvaluateInputShown()函数中已经判断过系统是否没有按键,如以下代码所示。该段代码的语义是,如果系统没有键盘或者键盘被隐藏了,那么就

```

989 public boolean onEvaluateInputViewShown() {
990     Configuration config = getResources().getConfiguration();
991     return config.keyboard == Configuration.KEYBOARD_NOKEYS
992         || config.hardwareKeyboardHidden == Configuration.KEYBOARDHIDDEN_YES;
993 }
  
```

(2) 判断非 SHOW\_EXPLICIT 标识。在一般情况下,该步骤的语义是,如果调用者没有明确提出显示输入法窗口,那么,当有键盘或者输入法为全屏模式时,就不要显示输入法。这种定义似乎也是合理的,既然用户没有明确要求显示输入法,而且还有键盘,那么显示一个虚拟的输入法窗口当然有点多余。

显示标识的传递过程大致如下。

无论是 EditText 的点击还是程序员主动调用 showSoftInput(), 一般都会调用到 IMM 的 showSoftInput()。此时假设参数的标识中要么是 SHOW\_FORCE 要么是 0, IMM 中会调用 IMMS 的 showSoftInput(), 继而调用 IMMS 的 showCurrentInputLoced(), 该函数的关键代码如下。

```

1043     if ((flags&InputMethodManager.SHOW_IMPLICIT) == 0) {
1044         mShowExplicitlyRequested = true;
1045     }
1046     if ((flags&InputMethodManager.SHOW_FORCED) != 0) {
1047         mShowExplicitlyRequested = true;
1048         mShowForced = true;
1049     }

```

这里注意 SHOW\_IMPLICIT 标识, 这个标识是 IMM 类中定义的, 而在 IMS 中判断时使用的却是 InputMethod 类定义的 SHOW\_EXPLICIT。这个的确有点不合理, 并且这两个常量的值竟然是相同的, 都是 1, 这更让人迷惑, 此处完全可以把 SHOW\_IMPLICIT 改为 SHOW\_EXPLICIT。此处的代码会给变量 mShowExplicitlyRequested (即 mShowForced) 赋值, 这两个值将重新改变显示标识。另外, 从该段代码中可以看出, 如果调用者使用了 SHOW\_EXPLICIT 标识, 反倒是不会把 mShowExplicitlyRequested 置为 true。从这里可以看出, 这段代码存在 bug, 应该把第一个条件中的 == 0 改为 != 0。

在接下来的处理中, 要使用以上两个变量。

```

1056     if (mCurMethod != null) {
1057         executeOrSendMessage(mCurMethod, mCaller.obtainMessageI00(
1058             MSG_SHOW_SOFT_INPUT, getImeShowFlags(), mCurMethod,
1059             resultReceiver));

```

此处调用 getImeShowFlags() 重新获得显示标识, 而该函数中正是利用了上面两个变量, 如以下代码所示。

```

631     private int getImeShowFlags() {
632         int flags = 0;
633         if (mShowForced) {
634             flags |= InputMethod.SHOW_FORCED
635                 | InputMethod.SHOW_EXPLICIT;
636         } else if (mShowExplicitlyRequested) {
637             flags |= InputMethod.SHOW_EXPLICIT;
638         }
639         return flags;

```

从该代码可以看出, 当用户使用 SHOW\_FORCED 时, 会自动包含 SHOW\_EXPLICIT 标识。

(3) 如果是 SHOW\_FORCED 标识, 则给变量 mShowInputForced 赋值为 true。

以上三小步虽然经过了让人有点迷惑的判断, 但代码的目标很简单, 即判断有没有必要显示输入法窗口, 而不显示的条件一般情况下只有一个, 那就是用户没有明确指定, 并且设备包含键盘。至于全屏模式的处理可以暂时忽略。

③ 如果有必要显示输入法, 则调用 IMS 中的 showWindow(true) 函数。

④ 返回结果包含这么几个意思：如果窗口本来就显示着，那么就返回 `UNCHANGED_SHOW`；如果本来没显示，而该调用导致显示出来，则返回 `SHOW`；如果本来没有显示，而且发现也有必要显示，则返回 `UNCHANGED_HIDDEN`，即依然保持隐藏状态。

下面分析以上第 ③ 步中 `showWindow()` 函数的具体执行过程。

为了理解 `showWindow()` 内部代码思路，首先对几个主要全局变量的含义做个解释。

- `mInputStarted`: `boolean` 型。前面讲过，当用户在多个编辑框中点击切换时，都会调用到 IMMS 的 `attachNewInputLocked()` 函数，其语义是告诉 IMS 有了新的输入框，而在 IMS 中，则会执行 `startInput()` 函数，它会间接调用 `doStartInput()` 函数，该函数中就会把 `mInputStarted` 变量置为 `true`，意思是开始输入。在 IMS 中的另一个对称函数 `doFinishInput()` 中会把该变量重新置为 `false`，而 `doFinishInput()` 是当输入法窗口从 `WmS` 中被删除时才会执行到的，而对于隐藏操作则不会将该变量置为 `false`。
- `mInputViewStarted`: 指输入法窗口是否显示在屏幕上，当调用 `hideWindow()` 时，该变量即被置为 `false`。因此，该变量的名称如果改为 `mInputViewShowStarted`，大家可能会更容易理解。
- `mWindowVisible`: 该变量的值与 `mInputViewStarted` 的值完全一致。这两个变量的赋值总是成对出现，从这点上看，这个变量有点多余，但是就算是成对出现，赋值的先后却不同。事实上，`mInputViewStarted` 有另一个作用。在显示输入法窗口之前，该变量的值为 `false`，程序中会判断该值，如果是 `false`，则会回调 `onStartInputView()` 函数，输入法设计者可以重载 `onStartInputView()` 函数，以便在输入法窗口显示之前可以做点什么，回调执行完后，则立即给该变量赋值为 `true`。
- `mCandidatesViewStarted`: 该变量和 `mInputViewStarted` 的值基本上保持一致，而且作用有点类似，它给输入法设计者提供了一个机会，以便在备选窗口显示之前可以做点什么。之所以没有把该变量和 `mInputViewStarted` 合并成一个变量，是因为输入法窗口的基本组成有三个，包含三个视图，分别是输入区、候选区及提取区（Extracting），其中提取区仅在输入法全屏时才出现。它是一个大的编辑框，该编辑框的内容将作为目标编辑框的内容，因此叫做“提取”，如图 17-18 所示。
- `mShowInputRequested`: 当调用显示函数显示窗口时，会先把该变量置为 `true`，直到调用隐藏窗口时，才被置为 `false`。因此可以认为，从窗口改变再从窗口显示到隐藏的过程中保持 `true`，它与 `mInputViewStarted` 的值基本保持一致，其作用仅仅是为了避免重复执行显示操作。

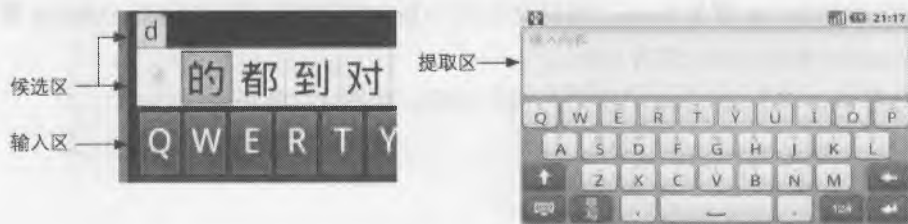


图 17-18 输入法窗口的不同区域名称

showWindow()内部尽管包含了多重调用,但其作用可以用一句话概括:回调 IMS 实例类,分别获取输入区、候选区及提取区视图,并把这些视图添加到输入法窗口中,然后调用 mWindow.show()显示该窗口。

showWindow()函数内部流程如图 17-19 所示。

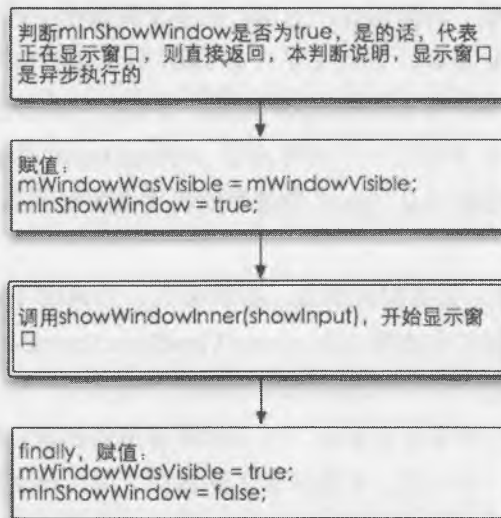


图 17-19 showWindow()函数内部执行流程

① 判断窗口是否正在显示。尽管可能还没有显示出来,但正在进行显示的操作,变量 mInShowWindow 是在该函数中被赋值为 true 的,而当该函数执行完成时则将其置为 false。这就确保了该函数不会被异步连续调用,不过该变量的作用域虽然是全局的,但却只在该函数中使用,并且该函数似乎也没有被异步调用,因此,该变量有点多余。

② 给两个重要变量赋值,其中 mWindowWasVisible 中的 Was 代表了这个变量的含义,即过去是否是显示的。

③ 调用 showWindowInner(showInput), 开始显示窗口。

④ 无论以上执行结果如何,都将认为窗口已经处于显示状态,所以将变量 mWindowVisible 置为 true,并将 mInShowWindow 置为 false。源码此处有个 bug,应当是把 mWindowVisible 置为 true,而源码中却是将 mWindowWasVisible 置为 true。

下面继续分析 showWindowInner()函数的执行过程,如图 17-20 所示。



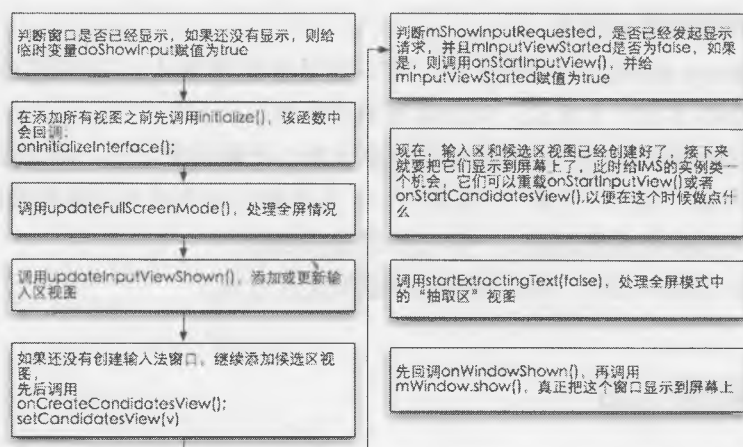


图 17-20 showWindowInner()函数内部执行流程

(1) 判断窗口是否已经显示。判断的条件是还没有调用过该函数 (`mShowInputRequested` 为 `false`), 并且有客户端编辑框和该输入法建立了联系 (`mInputStarted` 为 `true`), 并且该函数的调用参数为 `true`, 同时满足这三个条件的话, 意味着窗口还没有显示在屏幕上。

(2) 在继续执行之前, 先回调一下 `onInitializedInterface()`, IMS 实例可以重载该函数以便在窗口显示之前做点什么。

(3) 调用 `updateFullScreenMode()` 处理全屏情况。由于标准的输入法窗口中包含了不同的视图对象, 而全屏模式和普通模式中使用不同的视图, 因此, 该函数内部正是要根据是否全屏而选择使用不同的视图。

(4) 调用 `updateInputViewShow()` 处理输入区视图的显示状况。该函数内部及其调用虽然表面上有点复杂, 但其作用却很简单, 即判断是否需要显示输入区视图。当设备没有物理键盘, 或者物理键盘被隐藏时, 则必须显示输入区视图, 然后回调 `onCreateInputView()` 创建一个 `View` 对象, 并调用 `setInputView(v)` 将该对象添加到输入区视图对应的 `ViewGroup` 中。

(5) 接下来创建候选区视图, 这也是通过回调方式实现。回调 `onCreateCandidatesView()` 来创建一个 `View` 对象, 并调用 `setCandidatesView(v)` 把这个视图加入到候选区对应的 `ViewGroup` 中。

(6) 此时 `mShowInputRequested` 已经被置为 `true`, 因此将 `mInputViewStarted` 置为 `true`, 并且回调 `onStartInputView()` 函数, 给 IMS 实例一个机会, 意思是“输入区视图已经创建好了, 接下来就要显示出来了”。当然如果设备有物理键盘并且没有隐藏, 那么是不需要输入区的, 则会执行以下第 (7) 步。

(7) 如果第 (6) 步中不显示输入区窗口, 那么则必须显示出候选区代替, 而这里则是回调 `onStartCandidatesView()` 给 IMS 实例一个机会, 意思是“候选区视图已经创建好了, 接下来就要显示出来了”。

(8) 调用 `startExtractingText()` 处理“抽取区”的视图。抽取区只有在全屏模式下才会显示出来, 本步骤主要是设置抽取区中编辑框的初始字符串及提示字符串, 同时设置抽取区中的按钮的显示文字,

关于抽取区的详细内容参见第 17.4 节。

(9) 调用 `mWindow.show()` 方法把该窗口着实地显示到屏幕上，在调用该函数之前，先回调 `onWindowShown()` 给 IMS 实例一个机会，以便此时做点什么。

至此，输入法窗口就显示出来了，用户就可以在该窗口上输入文字了。本节仅介绍了 IMF 框架中所设计的输入法窗口的显示过程，注意，仅仅是窗口，至于窗口内部要显示何种视图对象则完全取决于输入法本身，输入法可以像处理普通窗口一样去处理这些视图。

下面一节将详细介绍输入法窗口内部视图的创建和交互过程。

## 17.4.2 标准布局的 IMS

首先来解释什么是“标准布局”。

在 IMF 框架中，本质上定义了一组 Binder 接口，所有的操作都是在这些 Binder 之间进行，至于 Binder 内部如何实现预先定义的 Binder 接口则完全是各 Binder 内部的事情。对于输入法厂商而言，仅有三个 Binder 接口可以自定义，如表 17-1 所示。

表 17-1 可自定义的 Binder 对象

Binder 名称	意 义
<code>IInputContext</code>	Interface 类型的 interface，客户端编辑框应该包含一个该类型的 Binder 对象，该对象将被 IMS 调用。该对象在 IMF 中被 <code>IInputConnectionWrapper</code> 类实现，本身已经不能重新定义，但是可以重载编辑框的 <code>onCreateInputConnection()</code> 返回一个自定义的 <code>InputConnection()</code> 功能接口，达到自定义目的
<code>IInputMethod</code>	Interface 类型的 interface，IMS 中的 Service 对象启动后应该返回一个类的 Binder 对象，该对象将被 IMMS 调用
<code>IInputMethodSession</code>	Interface 类型的 interface，当 IMMS 请求 IMS 创建一个 Session Binder 时，IMS 应该返回一个该类型的 Binder 对象，该对象将被客户端直接调用

无论输入法如何编写，只要能够实现以上三个 Binder 接口，则该输入法就能够在 IMF 框架中正常运行，哪怕该输入法不包含任何窗口。

然而，一般的输入法中却都包含了输入窗口，并且窗口中包含了相似的视图布局，比如 Google 中文输入法、拉丁输入法、搜狗输入法等，它们的窗口都包含了输入区、候选区。因此，为了简化这种具有相似布局的输入法设计，IMF 中提供了一种标准的 Binder 接口实现，这种标准的实现就称之为标准的输入法，或称之为“标准布局”的输入法。

从代码的角度来讲，标准输入法的实现文件主要包括以下几项。

- `AbstractInputMethodService`: 该类基于 Service，本身是一个 abstract 类型。
- `InpputMethodService.java`: 该类是 `AbstractInputMethodService` 的具体实现，简称 IMS。
- `IInputMethodWrapper`: 该类是 `IInputMethod` Binder 接口的一个具体实现。
- `IInputMethodSessionWrapper`: 该类是 `IInputMethodSession` Binder 接口的具体实现。

- EditableInputConnection: 该类实现了 InputConnection 接口。

对于具体的输入法，应该基于 InputMethodService 类实现一个实例类，并在实例类中重载一些回调函数，通过这些回调函数向标准布局添加自定义的视图。

以上代码的运行过程如下。对于一个基于该标准布局实现的输入法而言，比如 Google 拼音输入法，其所在的源文件名称为 PinyinIME.java，该类基于 InputMethodService，并且在该输入法的 AndroidManifest.xml 文件中要声明该 Service 能够响应一个指定的 action，如以下代码所示：

```
37 <service android:name=".PinyinIME"
38         android:label="@string/ime_name"
39         android:permission="android.permission.BIND_INPUT_METHOD">
40     <intent-filter>
41         <action android:name="android.view.InputMethod" />
42     </intent-filter>
43     <meta-data android:name="android.view.im" android:resource="@xml/method" />
44 </service>
```

因而，当 IMMS 使用约定的 action.view.InputMethod 查询输入法列表时便可以找到该输入法，当 IMMS 中调用 bindService() 时，对应的 Service 对象就会启动。由于 PinyinIME 基于 InputMethodService 类，因此，会首先执行到 InputMethodService 类的 onCreate() 函数中。该函数中主要创建了 SoftInputWindow 类对象，并赋值给 mWindow 变量，而 SoftInputWindow 的父类是 Dialog 类，即输入法窗口本质上是一个 Dialog。

然后调用 initView() 函数，该函数内部则会为该 Dialog 对象添加一个视图布局，布局文件名称为 com.android.internal.R.layout.input\_method。该布局结构如下，其中 FL 和 LL 分别代表 FrameLayout 和 LinearLayout。

```
LL vertical
LL mFullScreenArea, vertical
    FL mExtratFrame, gone
    FL mCandidatesFrame, invisible
FL mInputFrame, gone
```

以上布局包含了“抽取区”，对应 IMS 内部变量 mFullScreenArea；“候选区”，对应变量 mCandidatesFrame；“输入区”，对应变量 mInputFrame。该布局仅仅定义了一个视图框架，至于给 FrameLayout 中显示何种具体的视图，则由 IMS 通过回调输入法实例类的相应回调函数获得。换句话说，IMS 的实例类只需要重载这些实例类提供不同的视图即可，这也就是“标准布局”的含义。具体的回调函数如表 17-2 所示。

表 17-2 标准布局输入法中需重载的回调函数

回调函数名称	作用
onCreateInputView()	创建“输入区”视图
onCreateExtracTextView()	创建“提取区”视图
onCreateCandidatesView()	创建“候选区”视图

以上三个回调函数中，除了“提取区”外，其他两个默认都返回为空。也就是说 IMS 实例类必须

重载这两个函数并提供具体的视图，而提取区默认会返回一个如图 17-21 所示效果的视图，该视图包含一个编辑框和一个按钮。

该视图有如下特点：

- 编辑框中的提示文字和原始编辑框中的提示文字相同。
- 编辑框中的显示的文字内容和原始编辑框中的显示文字相同。
- 如果目标窗口中包含有下一个编辑框，则按钮显示的文字为“下一个”。如果是最后一个编辑框，则按钮显示的文字为“完成”。并且，有时按钮显示的文字是“发送”，比如在撰写短信时，如图 17-22 所示。

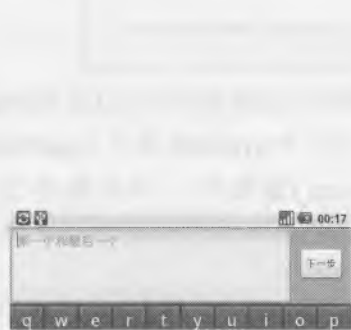


图 17-21 输入法全屏后的界面视图

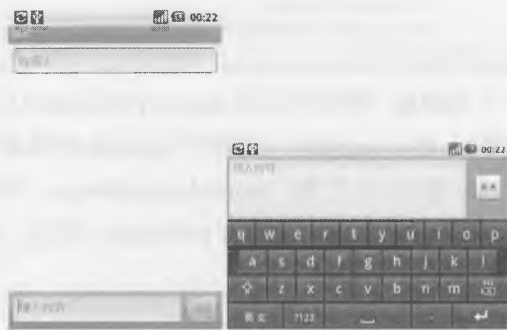


图 17-22 输入法窗口中的按钮文字

下面来分析这些特点是如何实现的。

首先，该视图的定义文件为 `input_method_extract_view.xml`，其内容如下：

```
LL horizontal
    android.inputmethodservice.ExtractEditText, width=0, weight = 1
    FL, id=inputExtractAccessories
        ExtractButton, id=inputExtractAction
```

该视图包含了一个 `ExtractEditText`，其父类是 `EditText`，并包含了一个按钮 `ExtractButton`，其父类是 `Button`。

当切换到全屏时，会执行 `IMS` 中的 `updateFullScreenMode()`，继而调用 `setExtraView()`，继而再调用 `startExtractText()`，该函数中就会从原始的编辑框中提取出提示文字和已输入的文字值，如下代码所示：

```
2050      mExtractedText = ic == null? null
2051          : ic.getExtractedText(req, InputConnection.GET_EXTRACTED_TEXT_MONITOR);

2049      eet.setInputType(inputType);
2050      eet.setHint(ei.hintText);
2051      if (mExtractedText != null) {
2052          eet.setEnabled(true);
2053          eet.setExtractedText(mExtractedText);
```

`eet` 即为 `ExtractEditText` 视图，以上代码分别调用该视图的 `setHint()` 和 `setExtractedText()` 设置提示文

字和已输入的文字。其中提示文字的来源是 `ei.hintText`，`ei` 为 `EditorInfo` 类型，该类型是一个 `Pacelable` 数据类，其所包含的信息是由原始编辑框传递过来的；`mExtractedText` 是一个 `ExtracText` 类型，该类也是一个 `Pacelable` 数据类，它包含了原始编辑框中的原始文字。这就是为什么“提取区”会显示和原始编辑框相同的文字内容的原因。

那么，按钮上的文字又是如何实现的呢？这也是在 `startExtracingText()` 函数中实现的，该函数中会调用 `onUpdateExtractingView(ei)`，参数 `ei` 为 `EditorInfo` 类型，该函数内部会从 `ei` 中提取出 `imeOptions` 变量，并根据该变量显示不同的文字，如以下代码所示。

```

1991         if (mExtractAction != null) {
1992             if (ei.actionLabel != null) {
1993                 mExtractAction.setText(ei.actionLabel);
1994             } else {
1995                 mExtractAction.setText(getTextForImeAction(ei.imeOptions));
1996             }
1997             mExtractAction.setOnClickListener(mActionClickListener);

```

其中，`actionLabel` 的值是在原始编辑框中使用 `android:imeActionLabel` 进行指定，如果用户指定了该值，则按钮上将显示这个自定义值，如果用户没有指定，则调用 `getTextForImeAction(ei.imeOptions)` 获取默认的字符串，参数 `imeOptions` 可以使用 `android:imeOptions` 在原始编辑框中进行指定。`getTextForImeAction()` 的代码如下。

```

1923-    public CharSequence getTextForImeAction(int imeOptions) {
1924        switch (imeOptions & EditorInfo.IME_MASK_ACTION) {
1925            case EditorInfo.IME_ACTION_NONE:
1926                return null;
1927            case EditorInfo.IME_ACTION_GO:
1928                return getText(com.android.internal.R.string.ime_action_go);
1929            case EditorInfo.IME_ACTION_SEARCH:
1930                return getText(com.android.internal.R.string.ime_action_search);
1931            case EditorInfo.IME_ACTION_SEND:
1932                return getText(com.android.internal.R.string.ime_action_send);
1933            case EditorInfo.IME_ACTION_NEXT:
1934                return getText(com.android.internal.R.string.ime_action_next);
1935            case EditorInfo.IME_ACTION_DONE:
1936                return getText(com.android.internal.R.string.ime_action_done);
1937            default:
1938                return getText(com.android.internal.R.string.ime_action_default);
1939        }
1940    }

```

代码中不同的 `case` 对应了 `android:imeOptions` 中不同的值，这就是说，应用程序可以使用该属性为原始编辑框指定 `Action` 类型。

当原始编辑框对应的 `EditorInfo` 没有指定 `imeActionLabel`，也没有指定 `imeOptions` 属性，并且 `inputType` 属性值为 `none`，此时输入法中“提取区”的按钮就不会显示，如以下代码所示。

```

1985        final boolean hasAction = ei.actionLabel != null || (
1986            (ei.imeOptions & EditorInfo.IME_MASK_ACTION) != EditorInfo.IME_ACTION_NONE &&
1987            (ei.imeOptions & EditorInfo.IME_FLAG_NO_ACCESSORY_ACTION) == 0 &&
1988            ei.inputType != InputType.TYPE_NULL);

```

这里注意区分 `EditorInfo` 类和 `TextView` 中的 `InputContentType` 类，前者的目的主要是为了在 `IMS`

和编辑框之间传递编辑框的属性信息，它实现了 `Pacelable` 接口；而後者的目的仅仅是 `TextView` 中为了保存该视图和输入法相关的信息。从功能上讲，这两个类相似，`EditorInfo` 是用户跨进程交换的，而 `InputContentType` 仅仅是线程内的数据类。

而在 `IMS` 中使用的全都是 `EditorInfo` 类，该类内部的变量值是 `IMM` 中回调 `TextView` 的 `onCreateInputConnection()` 时，根据 `TextView` 内部的 `InputContentType` 变量中的相关值产生的，尤其是 `imeOptions` 变量。并且当程序员没有给某个 `TextVeiw` 指定 `imeOptions` 时，`EditorInfo` 中的 `imeOptions` 会尝试自己确定该属性，如以下代码所示。

```

4604         if ((outAttrs.imeOptions&EditorInfo.IME_MASK_ACTION)
4605             == EditorInfo.IME_ACTION_UNSPECIFIED) {
4606             if (focusSearch(FOCUS_DOWN) != null) {
4607                 // An action has not been set, but the enter key will me
4608                 // the next focus, so set the action to that.
4609                 outAttrs.imeOptions |= EditorInfo.IME_ACTION_NEXT;
4610             } else {
4611                 // An action has not been set, and there is no focus to
4612                 // to, so let's just supply a "done" action.
4613                 outAttrs.imeOptions |= EditorInfo.IME_ACTION_DONE;
4614             }

```

代码中，调用 `focusSearch(FOCUS_DOWN)` 查询当前窗口是否有下一个编辑框。如果有的话则把 `imeOptions` 置为 `ACTION_NEXT`，这将使得输入法窗口中的按钮的值显示为“下一个”；如果没有下一个编辑框，则把 `imeOptions` 设为 `ACTION_DONE`，对应字符串“完成”。这就是为什么输入法窗口中“提取区”按钮上的字符串能够如此“聪明”的原因。

接下来继续分析当这个按钮按下后的执行过程。

当点击按钮后，首先当然回调的是输入法窗口内部定义的回调函数，这在 `IMS` 中对应 `mActionClickListener` 变量，该变量的定义如下。

```

293     final View.OnClickListener mActionClickListener = new View.OnClickListener() {
294     public void onClick(View v) {
295         final EditorInfo ei = getCurrentInputEditorInfo();
296         final InputConnection ic = getCurrentInputConnection();
297         if (ei != null && ic != null) {
298             if (ei.actionId != 0) {
299                 ic.performEditorAction(ei.actionId);
300             } else if ((ei.imeOptions&EditorInfo.IME_MASK_ACTION)
301                 != EditorInfo.IME_ACTION_NONE) {
302                 ic.performEditorAction(ei.imeOptions&EditorInfo.IME_MASK_ACTION);
303             }
304         }
305     }
306 };

```

该代码中，首先获得当前编辑框对应 `EditorInfo` 对象，上面讲过 `EditorInfo` 是客户编辑框传递过来的。然后判断客户编辑框是否指定了 `actionId`，如果指定，则回调 `performEditorAction()` 执行该 `actionId` 对应的操作；如果没有指定，则把 `imeOptions` 转换成一 `actionId`，并回调 `performEditorAction()`。

`performEditorAction()` 函数是在 `InputConnetion` 接口中定义的，实现该接口的是客户端的 `EditableInputConnection`，该类中该函数又直接回调对应的编辑框视图的 `onEditorAction()` 函数。



TextView 的 onEditorAction() 函数中, 首先判断该视图是否存在 inputAction 回调, 如果存在的话则先回调, 并且只有当回调中没有处理该 action 时才继续往下执行, 如以下代码所示。

```

3176     final InputContentType ict = mInputContentType;
3177     if (ict != null) {
3178         if (ict.onEditorActionListener != null) {
3179             if (ict.onEditorActionListener.onEditorAction(this,
3180                 actionCode, null)) {
3181                 return;

```

当程序员希望自己处理输入法窗口中的 mExtractAction 按钮消息时, 则可以调用 TextView 的 setOnEditorActionListener() 为 mInputContentType 的 onEditorActionListener 变量赋值。如果程序员没有特别处理该消息, 那么就会进行默认的处理。在短信程序中, 当全屏模式输入消息时, 可以直接点击输入法中的发送按钮将消息发送出去, 其原理正是使用 setOnEditorActionListener() 添加自定义的处理, 这与原始视图中的“发送”按钮没有任何关系。

默认处理仅仅是把焦点窗口移向下一个编辑框, 如果不存在下一个编辑框, 则隐藏输入法。移动焦点分两步执行, 首先调用 focusSearch() 寻找下一个编辑视图, 然后调用 requestFocus() 使该视图获得焦点, 下一个编辑框获得焦点后, 又会开始一次新的输入法显示过程, 关于该过程的细节参照附图 12 所示。输入法相关的调用过程大致如下: 调用 requestFocus() 会导致调用 handleGainFocusInternal(), 继而调用 onFocusedChanged(), 该函数中会判断是否获得焦点, 并且该视图所在的窗口是否是当前交互窗口。如果是, 则调用 IMM 的 focusIn() 函数, 而在 focusIn() 函数中会调用 scheduleCheckFocusLocked(), 该函数则会发送一个异步消息 CHECK\_FOCUS, 这就会导致调用 IMM 中的 checkFocus() 函数, 这就又回到第 17.3.3 小节启动输入法的过程。

大家可能会注意到以下现象, 当在不同输入类型的编辑框中点击切换时, 输入法的输入区会根据编辑框的类型自动切换不同的输入法面板。比如联系人程序中新建联系人时, 姓名输入框和电话号码输入框。这种自动切换是如何实现的呢?

其实这个过程很简单, 如上所述, 无论是用户主动在编辑框上切换还是点击提取区中的“下一个”按钮, 都会导致执行 IMM 中的 checkFocus(), 继而调用到 IMMS 中的 startInput(), 继而再调用到 attachNewInputLocked()。该函数会发送一个异步消息 MSG\_START\_INPUT 或者 MSG\_RESTART\_INPUT, 两个消息的参数中都包含 mCurAttribute 变量, 该变量的类型是 EditorInfo, 该类内部则包含了原始编辑框中的输入类型。而在 IMS 中的 startInput() 调用过程中则调用了 IMS 中的 doStartInput(), 该函数中会回调具体 IMS 实例的 onStartInputView(EditorInfo, ...), 在一般的输入法设计中, 会重载该函数, 并根据参数 EditorInfo 所包含的 inputType 值切换不同的输入法面板。该调用过程如图 17-23 所示。



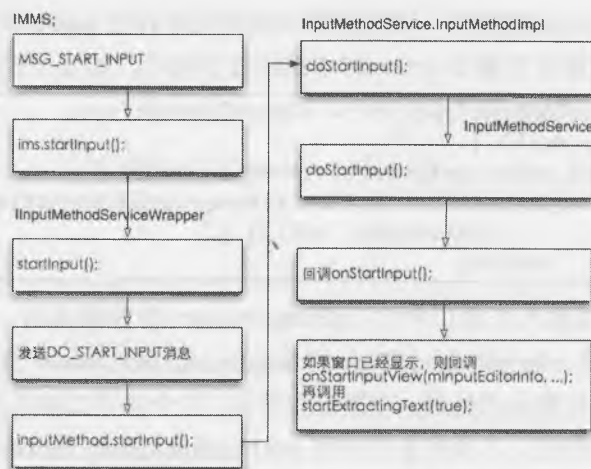


图 17-23 IMMS 中使用异步消息传递参数给 IMM 的过程

### 17.4.3 自定义布局的 IMS

什么是自定义输入法？简单的讲，如果你编写的输入法不是基于 `InputMethodService` 或者 `AbstractInputMethodService` 类，而是完全重新实现 `IInputMethodService.aidl` 中定义的 Binder 接口，那么就可以认为这是一个自定义输入法。

前面几节介绍过，IMF 的本质是几个 Binder 在交互信息，至于如何实现这些 Binder，IMF 本身并没有规定。上节介绍的 `InputMethodService` 只是为了方便设计而提供的一种实现实例，因此从严格意义上讲，实现一个自定义输入法意味着要实现这些 Binder 接口，具体包括以下三个。

- **InputMethod 接口**：当输入法所在的 Service 被启动时，必须在其 `onBind()` 函数中返回一个该 Binder 接口，在标准输入法中，返回的是 `IInputMethodWrapper` 类对象。
- **InputMethodSession 接口**：当 IMMS 调用输入法所在 Service 的 `createSession()` 方法时，必须返回一个该 Binder 接口，在标准输入法中，返回的是 `IInputMethodSessionWrapper` 对象。
- **InputContext 接口**：当 IMM 调用 IMMS 的 `startInput()` 时，参数中需要包含该 Binder 对象，而该 Binder 对象是在 IMM 中创建的，名称为 `ControlledInputConnectionWrapper`。这个 Binder 对象本身是不能自定义的，但该 Binder 对象是通过一个 `InputConnection` 接口构造的，而这个 `InputConnection` 接口的实现却可以自定义。在标准输入法中，该接口的实现类为 `EditableInputConnection` 类。

到目前为止，国内主流的输入法包括 Android 自带的 Google 拼音输入法、HTC 的中文 Touch 手写输入法、搜狗拼音输入法，而这些输入法全部属于标准输入法。

那么为什么要实现自定义的输入法呢？

如果你对标准输入法窗口的那种“输入区”、“候选区”、“提取区”的布局不满意，并想设计一种该

布局无法满足的布局，那么可以考虑实现自定义输入法。比如，设想一种没有候选区，或者候选区会嵌套到输入区中的布局，如图 17-24 所示。

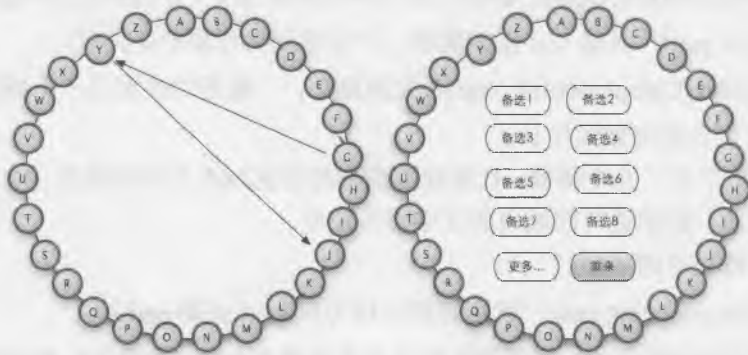


图 17-24 一种可能的输入法窗口

该输入法的设想是，当用户输入中文时，可以在目标词语对应的首字母处画一个路径。比如输入“葛永娇”时，只需要在 G、Y、J 三个字母处画一个路径，然后在屏幕的中央就会出现备选词语，这种输入法超越了标准输入法的框架，因此可以考虑为自定义输入法。

如果你对标准输入法中的那种调用逻辑不满意，想设计一种更为简单的输入法，比如，该输入法可能只是为了输入某种特定的数据。

如果你想设计一种自定义的视图，类似于 `EditView`，但是却不是 `EditView`，比如，该视图可以接收的字符，并自动把字符转换为图片。

总之，凡是你能设想的，标准输入法中没有实现的，都可以考虑实现自定义输入法。

## 17.5 向编辑框传递字符

在输入法窗口显示后，用户可以点击虚拟按键，就可以把相应的字符传递到目标编辑框中。该过程比较简单，对于 IMS 来讲，根据点击的虚拟按键产生相应的字符，并且在候选区中显示相应的文本，这仅仅是 IMS 内部的视图交互，就像普通应用中的视图交互一样。接着 IMS 需要调用 `InputConnection Binder` 对象的 API 把这些字符传递给客户端。

`InputConnection` 接口中定义的函数可以按照传递字符的方式分为三类，分别为读取、插入、替换。第一类，获取编辑框中已有的字符。获取又分为以相对光标位置和使用绝对位置进行获取。

- `getTextBeforeCursor(n, flags)`: 获取当前光标前面的 `n` 个字符。
- `getTextAfterCursor(n, flags)`: 获取当前光标后的 `n` 个字符。
- `getExtractedText(request, flags)`: 获取当前输入框的所有文字。
- `setSelection(start, end)`: 设置绝对获取位置，从 `start` 处到 `end` 处。

- `getSelectedText(flags)`: 获取绝对区域的文字, 该区域由 `setSelection()` 指定。

第二类, 向编辑框中添加或者删除字符。

- `deleteSurroundingText(left, right)`: 删除当前光标处前面 `left` 个字符和后面 `right` 个字符。
- `commitText(text, pos)`: 传递 `text` 给编辑框, 并且移动光标到 `Pos` 位置。
- `commitCompletion(CompletionInfo text)`: 完成提交, 一般是切换到另一个编辑框之前调用, 从而使得客户端可以在此时做点什么。

表面上看, 该类似乎少了几个函数, 比如没有删除绝对区域内文字的函数。实际上, 这可以使用第三类中的替换函数完成, 即把绝对区域内的文字替换成空。

第三类, 替换编辑框中的字符。

- `setComposingRegion(start, end)`: 设置替换区域为从 `start` 处到 `end` 处。
- `setComposingText(text, pos)`: 把替换区域的文字替换为 `text`, 并将光标移动到 `pos` 处。
- `finishComposingText()`: 可以重复设置替换区域并进行替换, 完成所有这些后, 则可调用该函数通知客户端的编辑框, 从而使得客户端可以做点什么。

除了以上三大类外, 还有一个函数 `getCursorCapsMode(reqModes)`。该函数用于获取当前光标处的大写模式, 以便当用户移动光标时, 输入法能够根据当前大小写模式自动显示不同的输入法面板。比如“ABc”, 当光标移动到 B 的后面时, 输入法上显示的字符都会变成大写, 而当移动到 c 后面时, 则又会自动变成小写。这种自动切换正是调用该函数获取光标所在位置的大小写模式, 然后根据该模式进行切换完成的。

以上介绍了传递文字所使用的 API 接口, 客户端中如何实现这些 API 则完全是自定义的事情, 在一般情况下 `TextView` 视图所实现的这些 API 更多的是 `TextView` 类本身所包含的文字编辑 API, 因此, 请参照 `TextVeiw` 内部的文字编辑的实现, 本章不再赘述。

## 17.6 输入法相关源码清单

输入法相关的源码文件分布在四个地方, 不同的地方本身就意味着其代码在整体架构中的角色不同。第一部分, `Package android.inputmethodservice`, 该包主要被一个具体的输入法实现所使用。

- `AbstractInputMethodService.java`, 基于 `Service`, 并且内部实现了 `InputMethod` 和 `InputMethodSession` 接口, 但只是 `abstract` 级别的实现, 还需要具体的实现, 这在 `IMS` 中完成。
- `ExtractButton.java`。
- `ExtractEditText.java`。
- `IInputMethodSessionWrapper.java`, 实现了 `IInputMethodSession.Stub`, 并且把 `IPC` 调用转换为 `InputMethodSession` 接口的调用。
- `IInputMethodWrapper.java`, 实现了 `IInputMethod.Stub`, 并且把 `IPC` 调用转换为 `InputMethod` 接口的调用, 使用时, 首先通过构造函数传递一个 `InputMethod` 接口对象。

- InputMethodService.java, 基础类, 该类基于 AbstractInputMethodService, 具体的输入法必须基于该类。该类内部有两个内联类, 实现了 AbstractInputMethodService 中的两个 abstract 内联类, 分别为 InputMethod 接口和 InputMethodSession 接口。同时, 因为 AbstractInputMethodService 类虚拟实现了 KeyEvent.Callback 接口, 因此, 在 IMS 中也具体实现了这些接口。
- Keyboard.java, 该类是一个功能类, 其作用是把键盘矩形区中的按键位置转换为按键值。
- KeyboardView.java。
- SoftInputWindow.java。
- Keyboard.java 只是一个数据类, 能够从特定的 XML 文件中读取并产生一个 Keyboard 数据, 该数据可以作为 KeyboardView 的内部成员数据变量。Keyboard 要求 XML 文件中描述的排列必须是以固定 row 宽排列的按键, 因为 Keyboard 内部正是根据“固定”计算出该按键的位置。

KeyboardView 是一个 extends View 的类, 该视图内部有一个 Listener 成员, 包含 onKey()、onClick() 等接口, 这些接口实际上都是通过 KeyboardView 中的 onTouchEvent() 产生的。也就是说, KeyboardView 首先把触摸消息转变为接口消息, 然后 LatinIME 实现该接口消息, 从而完成类似于实际键盘的按键消息处理。该类中有一个方法 getKeyIndices(touchX, touchY, null);, 该方法正是通过点击位置映射出对应的按键值。

在 Latin 输入法中:

- Keyboard: 对应 LatinKeyboard。
- KeyboardView: 对应 LatinKeyboardView。

第二部分, Package android.view.inputmethod 该包主要给应用程序使用。

- BaseInputConnection.java, 实现了 InputConnection 接口。
- CompletionInfo.java。
- EditorInfo.java。
- ExtractedText.java。
- ExtractedTextRequest.java, 实现了 Parcelable 接口。
- InputBinding.java, 实现了 Parcelable 接口。
- InputConnection.java, 核心 interface。
- InputConnectionWrapper.java, 这个 Wrapper 的结构和前面的完全不同, 前面的可以把 IPC 调用转换为本地 interface 调用, 而此处仅仅是一个 Proxy 而已。
- InputMethod.java, 核心 interface。
- InputMethodInfo.java, 实现了 Parcelable 接口。
- InputMethodManager.java, IMM。
- InputMethodSession.java, 核心 interface。
- CompletionInfo.aidl, 仅仅是一个 Parcelable。

- EditorInfo.aidl。
- ExtractedText.aidl。
- ExtractedTextRequest.aidl。
- InputBinding.aidl。
- InputMethodInfo.aidl。

第三部分，Package com.android.internal.view，该包定义了一些内部类，这些类将作为 IMF 引擎所使用的基础类。

- IInputConnectionWrapper.java，实现了 IInputContext.Stub，从这个角度来看，该文件的命名有些混淆，把 context 和 connection 统一起来不是更好？
- InputBindResult.java，实现了 Parcelable。
- IInputConnectionWrapper.java，实现了 InputConnection 接口而已，文件命名时，如果前面有多余一个 I，则意味着是一个 Binder，否则只是一个 Wrapper 而已，即把所有的 abstract 方法进行空实现。
- IInputConnectionCallback.adil。
- IInputContext.aidl。
- IInputContextCallback.adil。
- IInputMethod.aidl。
- IInputMethodCallback.aidl。
- IInputMethodClient.aidl。
- IInputMethodManager.aidl，这个实际上对应 IMMS 的实现。
- IInputMethodSession.aidl。
- InputBindResult.adil，这是一个 Parcelable 实现而已。

第四部分，就是 IMMS，它属于系统服务的一部分，不再赘述。



## 第 4 部分 编译篇

第 18 章 Android 编译系统

第 19 章 编译自己的 Rom



## 第 18 章 Android 编译系统

在第 1 章中讲过，make 脚本的基本格式为：

目标(target)：条件(prerequisite)  
(tab 键) 命令

比如：

```
hello: hello.c
    gcc hello.c -o hello.bin
.PHONY he
he: hello.c
    gcc hello.c -o hello.bin
```

在以上脚本中，hello 为编译的目标，hello.c 是编译的条件，即只要 hello.c 有变化，那么当调用 make hello 时就会执行 gcc hello.c -o hello.bin 命令。.PHONY 关键字用于声明一个目标，并且第一个.PHONY 声明的目标将作为默认的目标，即当用户调用 make 命令时不指定任何参数，make 程序将默认执行目标为 he 对应的命令。

- 这是一段多么简单的脚本啊，Android 的编译系统理论上也是基于这个脚本模型进行构建的，只是 Android 源码中包含了太多的子项目，于是 Android 的脚本工程师对以上这个脚本模型进行了一定的封装，从而便于多个项目的编译，这就像从一个 Hello world 程序扩展成为一个复杂的应用程序一样。本章就来剖析 Android 的编译内部是如何进行封装的，以及其中所包含的各种辅助脚本文件之间如何相互作用。但大家心中要保持一个清醒的认识，那就是所有这些封装在被 make 程序解析时，所产生的脚本文件的结构依然是这个简单的脚本模型。



## 18.1 Android 源码文件结构

在分析编译系统之前，先来看一下 Android 的源码结构，这将有助于理解后面的编译辅助脚本如何根据这种文件结构进行特别的设计。源码结构如表 18-1 所示。

表 18-1 Android 源码根文件夹结构

文件夹名称	意 义
bionic	Android 中所使用的是标准 C 库源文件，该项目将被编译成静态库文件，这些库文件仅在编译其他 C 程序时使用，不会被输出到最终设备中
bootable	二次引导程序源码，对应 NAND Flash 的二次引导分区中的内容，这部分代码不会被包含在最终的 system.img 或 boot.img 文件中
build	编译系统中枢，该目录下的各种 make 脚本和 shell 脚本共同组成了 Android 的编译环境，也是本章介绍的重点
cts	兼容测试代码，基于 Android 系统的手机如果需要 Google 的认证，则必须经过 Google 的兼容性测试，兼容性测试的目的是为了保证该设备具有标准的 SDK API 接口
dalvik	Android 中的 Java 虚拟机相关的源码，本书仅介绍虚拟机相关模块的作用，而不介绍虚拟机内部工作原理，详见本书第 10 章。虚拟机内部设计相关的文档请参考该 dalvik/docs 目录下的说明文档
development	一些开发 Android 工程所使用的相关配置或者文件，比如一些 classpath 文件
device	不同设备相关的编译脚本文件，一个设备一般就是指一种型号的手机，比如 Nexus One、Nexus S、HTC magic 等
external	Android 系统所依赖的一些外部库文件，比如 sqlite、opencore 等，这些文件大部分都是 C/C++ 代码，也有少量的 Java 库
frameworks	Framework 的内核源码，主要由 Java 文件组成
hardware	Android 定义的硬件抽象层（HAL）相关的头文件
libcore	Dalvik 虚拟机所依赖的 Java 库，比如 ArrayList，HashMap 等，这些库和 Dalvik 共同组成 Dalvik 的 Java 运行环境
ndk	Native Development Kit，即编译 NDK 所需的相关文件
packages	Android 中的一些系统应用程序，比如 Contacts、Music Player 等
prebuilt	编译所需的程序文件，主要包含不同平台的 ARM 编译器
sdk	编译 SDK 所需的相关文件
system	Android 底层的 Linux 所需的一些系统工具程序，这些程序主要运行于 adb、logcat 等
tools	一些其他辅助工具，目前只有一个名称为 tradefederation 的 Java 项目
vendor	和 device 目录中的程序共同定义一个设备所包含的私有文件，它与 device 目录的区别在于，device 主要定义了不同设备所需的特有文件，但却是开源的，而 vendor 中包含的文件一般都是目标文件，是不开源的

## 18.2 从调用 make 命令开始说起

为了分析 Android 内部编译系统的结构，我们先从如何使用该编译系统开始说起。

### 18.2.1 编译命令

Android 编译系统被调用的主要方式分为三种，如下。

第一种，编译整个 Android 系统，编译完毕后，将在 out 目录的相关子目录下生成 system.img 文件及 boot.img 文件。调用的方法如以下代码所示。

```
cd path/to/android/root
. ./build/envsetup.sh
make PRODUCT-full_crespo-eng
```

在以上代码中，首先进入到 android 工程的目录，path/to/android/root 仅仅是一种自然语言的描述，而不是一个真实的具体目录，这种表示方式在 Linux 世界中经常使用。

第二句以一种 source 的方式，调用 envsetup.sh 设置编译环境，其作用仅仅是定义了几个快速编译函数而已，比如 mm 和 mmm，如果不调用这些特别的编译函数，则可以不执行该脚本。所谓的 source 方式就是命令的开头使用一个“.”，这个点不是一个简单的符号，它在 Shell 脚本环境中实际上对应的是一个内部程序。

第三句中，PRODUCT 是一个固定字符串，不能改变，中隔线后面紧跟一个产品的名称，最后再跟一个该产品的子分类。那么，什么是一个产品（product）呢？产品的子分类又是什么意思呢？

第二种，编译某个子工程，子工程可以是一个 APK，或者是 Java 运行时所需的一个 Jar 库，也可以是一个 Linux 中的静态库、动态库或可执行程序。调用的方法如以下代码所示。

```
make libbz
或者 mmm external/bzip2
```

第一句中 libbz 代表一个子工程，那么这个子工程的名称是在哪里定义的呢？它是某个文件夹名称吗？与第一句具有相同作用的编译脚本是使用 mmm 脚本函数，这个函数是在 envsetup.sh 中定义的，参数指向子工程所在的目录，注意这里的三个 m 组成。与三个 m 相似的还有两个 m，mm 是编译当前目录下的所有子项目，而 mmm 是编译指定目录中的子项目。

第三种，编译 sdk，如以下代码所示。

```
make sdk
```

### 18.2.2 编译结构猜想

前面讲过，无论 Android 的编译系统经过何种抽象，最终都将产生一个标准的 make 所需的脚本文件，这个脚本文件的格式必须如以下格式所示。

```
目标(target): 条件(prerequisite)
      (tab 键) 命令
```

那么, Android 系统是如何将上一节所讲的几种命令调用转换为标准的 make 脚本的? 在分析之前, 大家可以先猜想一下最终生成的编译脚本文件, 如以下代码所示。

```
=====第一部分=====
Contacts: x1.java, x2.java
    javac x1.java x2.java
Music: y1.java, y2.java
    javac y1.java y2.java
其他各种 Java 子项目

=====第二部分=====
libbz: x1.c x2.c
    gcc x1 x2
opencore: y1.c y2.c
    gcc y1 y2
其他各种 C/C++子项目

=====第三部分=====
sdk: Contacts, opencore
    `mkdir out'
    cp Contacts.apk out/.../
    其他相关的复制命令
.PHONEY sdk
.PHONEY 各种子项目名称
```

为什么说这是最基本的呢?

首先, 既然可以直接 make Contacts, 那么 Contacts 应该是一个 make 脚本中的一个 target, 并且该 target 依赖于 javac 编译器进行编译。

其次, 既然可以直接 make libbz, 那么 libbz 应该也是一个 target, 并且该 target 依赖于 gcc 编译器, 并且这个 gcc 编译器并不是 PC 上的, 而是 arm 的 gcc 编译器。

最后, 既然可以 make sdk, 说明 sdk 可能也是一个 target, 并且 sdk 最终包含了一个 image 文件, 因此, sdk 对应的执行的命令中应该包含各种 mkdir 和 cp 等命令用于进行文件复制。

做了以上假设后, 读者们就自然会提出以下这些问题。

第一, 子工程中都包含哪些源文件, 这在 Android 编译系统内部是如何得到的? 难道要手工列出所有源文件名称吗? 如果采用自动搜索的方式可能会更智能一点, 但这些都是如何实现的呢? 比如 Contacts 工程中包含各种 Java 源码, 这是在哪里指定的呢?

第二, 每个子项目都需要包含不同的编译器, 比如 Java 需要的是 javac, C/C++需要的是 gcc 或者 g++, 那么这些编译器又是如何动态指定的呢?

第三, 以上代码仅仅列出了一个示例子工程, 那么 Android 编译系统怎么知道整个 Android 工程都包含哪些子工程呢? 它是如何去搜索的? 当然最笨的办法就是使用一个列表文件, 在上面写下所有的子工程列表, 但 Android 编译系统是这么做的吗?

第四, 对于 sdk 的编译或者整个系统的编译, out 目录下的路径是如何指定的, 最后又是如何被转换为相关的 image 文件的呢?

第五, 对于 `make PRODUCT-full_crespo-eng` 命令, 到底什么是一个 product, 我们都可以使用哪些 product 名称, 它们的区别都在哪里? 如何删除或者增加一个新的 product?

第六, 其他你觉得神奇的地方都可以先写在下面的空白处。

本章后面的小节将会以这些问题为指引, 逐个回答它们。

### 18.3 编译所需脚本文件之间的协同关系

Android 编译系统是通过各种.mk 文件和各种 shell 脚本文件共同定义了一个编译框架, 这个框架基于基本的 make 概念。换句话说, Android 的编译系统不是重新定义了编译脚本, 而是建立一个编译框架, 以便于给该框架中添加新的子项目。打个比方, make 解释器本身就像是 C 语言的语法, 而 Android 的编译系统就像是 libc 库, libc 库并没有定义新的 C 语言, 而只是基于 C 语言写了一些库文件, 从而方便应用程序的开发。

Android 编译系统中定义这个框架的源码在 `./build/` 目录下, 了解该编译系统的本质实际上是分析这个脚本文件之间的系统关系。

#### 18.3.1 编译系统内部功能模块图

系统内部的功能模块如图 18-1 所示。

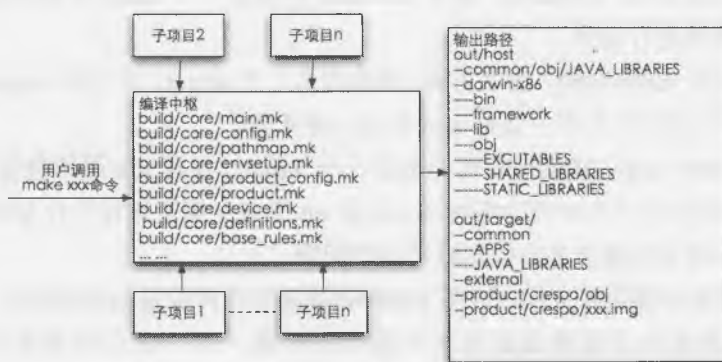


图 18-1 Android 编译系统功能模块图

整个编译系统主要由三部分组成, 分别为编译中枢、子项目及输出路径。

编译中枢中主要包含各种.mk 文件, 这些文件将遍历所有的子项目, 并生成所有的 target, 从而当用户调用 `make xxx` 命令时, 中枢能够知道应该执行什么命令以产生指定的 target。

子项目是在调用 `make xxx` 命令前就保存好的, 每个子项目中都必须包含一个 `Android.mk` 文件, 这个文件的名称是固定的, 不能改变。该文件中将描述该项目中包含哪些源文件, 并且指定该项目的输出

目标是何种类型，比如可以是一个 Jar 包、可执行程序、APK 包等。编译中枢内部定义了一些变量，各子项目只需要在各自的 `Android.mk` 中给这些变量进行不同的赋值即可，相比 `make` 自身的语法而言，这种赋值方法则更简单。

输出路径包含了编译过程所保存的各种临时文件，当然输出路径本身也是在编译中枢中定义的，默认定义为 `out` 目录。该目录下主要包含两个子目录，分别是 `host` 和 `target`，`host` 对应的是 PC 上所需的各种工具，这些工具将参与编译最终所需的目标程序；`target` 就是最终的输出目标。当然，SDK 本身就是在 PC 上执行的，因此，它的输出路径在 `out/host` 目录中。

以下代码来自于子项目 `packages/apps/Camera` 中的 `Android.mk` 文件。

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_SRC_FILES := $(call all-java-files-under, src)

LOCAL_PACKAGE_NAME := Camera
LOCAL_SDK_VERSION := current

LOCAL_PROGUARD_FLAG_FILES := proguard.flags

include $(BUILD_PACKAGE)
```

如前所述，`Android.mk` 文件中只需要对编译中枢所定义的一些变量赋值即可，这些变量一般包括以下几个。

- **LOCAL\_PATH**: 指定该子项目的绝对路径。一般可以通过直接调用编译中枢定义的函数 `my-dir` 获取，而不需要指定一个显示路径，比如 `./packages/apps/Camera`。
- **LOCAL\_MODULE\_TAGS**: 指定当前子项目所属的标签。标签是中枢定义的，各子项目只能指定自己属于哪个标签，而不能新建标签，常见的标签包括 `user`、`eng`、`userdebug`、`optional`。编译中枢会将所有的子项目按照标签进行归类，不同的 `product` 可以指定自己所使用的标签，从而中枢会把和该标签相同的子项目编译到该 `product` 中，关于 `product` 的定义见后面小节。
- **LOCAL\_SRC\_FILES**: 指定该项目所包含的所有源文件，系统提供了相应的函数，可以直接调用，如代码中所示。但是，该函数只能寻找到所有的 Java 文件，而不能寻找到 `aidl` 文件，因此，如果源码中包含 `aidl` 文件，还需要手工在 `Android.mk` 文件中说明，可参考 `packages/apps/Music` 子项目中的 `Android.mk` 文件。这也就回答了前面所提出的问题，即如何指定项目中包含的源文件。
- **LOCAL\_PACKAGE\_NAME**: 指定该项目的名称，该名称将作为 `make xxx` 中的 `xxx`，即子项目不是按照目录区分的，而是按照该变量指定的名称进行区分，至于该名称如何被转换到 `make` 所需要的 `target` 将在后面介绍。该变量仅适应于 Java 项目，对于 C/C++ 项目，则使用变量 `LOCAL_MODULE`，两者的意义是相同的，即作为 `make xxx` 中的 `xxx`。

- **BUILD\_PACKAGE**: 该变量可理解为一个宏, 它不需要被赋值, 因为中枢已经给该变量赋值了。与该变量相关的还有 **BUILD\_HOST\_EXECUTABLE**、**BUILD\_HOST\_STATIC\_LIBRARY** 等其他几个变量, 如其名称所示, 这些变量内部被定义为具体的执行命令, 其结果是编译不同的目标对象, 比如 APK 对象、静态库对象、动态库对象等。本例中 **include** 了 **BUILD\_PACKAGE**, 意思就是告诉中枢“请把我编译成一个 APK”, 那么, 这些不同变量具体对应的执行命令是什么, 是在哪里定义的呢? 将在后面小节中介绍。

以上即为 Android 编译系统的大致框架, 下面继续从编译脚本源码的角度来分析这个架构是如何实现的。由于 Build 系统中包含的各种脚本文件内部注释比较丰富, 因此, 本章不再完全分析代码的整个执行过程, 而仅以前面提出的架构性问题为指引来分析其中相关的代码。

### 18.3.2 脚本文件的包含关系

第 1 章的 **make** 小节中讲过, 在调用 **make** 时, 系统默认从当前目录下的 **Makefile** 文件中读取编译脚本。Android 的 **make** 系统也不例外, 因此, 接下来分析该 **Makefile** 的执行内容。

该文件的内容很简单, 它相当于一个入口, 然后引用 **build/core/main.mk** 文件, **main.mk** 中开始真正的编译中枢运作。该文件的内部执行流程如附图 13 所示。

① 使用 **.PHONY** 定义默认的 **target** 名称为 **droid**。这就解决了当用户直接调用 **make** 命令而不包含任何参数的情况, 这里仅仅定义了默认的 **target** 名称, 而还没有定义该 **target** 对应的执行命令, 具体的执行命令可以在后面继续定义。

② 包含 **config.mk** 文件, 该文件中将扫描整个编译系统中定义的各种 **product** 名称, 并进行必要的系统配置。具体如下。

(1) 定义一部分路径变量, 比如 **SRC\_HEADERS** 等, 具体可以参照源码。

(2) 包含 **pathmap.mk**, 该文件也定义了一些路径变量, 即把一些绝对路径定义为变量, 以便于其他脚本文件使用, 比如 **frameworks-base:frameworks/base/include**, 这种定义使用冒号分隔名称和绝对路径。

(3) 定义各种编译命令宏。比如 **BUILD\_EXECUTABLE** 对应是编译可执行文件, 该宏最后会被展开为真正的 **gcc/g++** 等命令, 只不过这里使用相应的 **make** 脚本进行了封装, 比如 **BUILD\_EXECUTABLE** 的实现是在 **executable.mk** 文件中。这就是为什么每个子项目中的 **Android.mk** 文件可以直接使用 **include** 包含相应的命令宏的原因。

(4) 包含 **buildspec.mk**。

(5) 包含 **envsetup.mk**。注意这里是 **.mk** 类型, 而不是 **envsetup.sh**。正如其名称所指, 该文件将完成环境的配置, 主要包括扫描系统中都包含哪些 **product**。具体如下。

① 包含 **product\_config.mk**。该文件会进而包含 **product.mk**, 该文件中定义了读取 Android 根目录下的两个文件夹, 分别是 **device** 和 **vendor**, 并从这两个目录下搜索所有的 **AndroidProduct.mk** 文件, 如下代码所示。

```

25define _find-android-products-files
26$(shell test -d device && find device -maxdepth 6 -name AndroidProducts.mk) \
27 $(shell test -d vendor && find vendor -maxdepth 6 -name AndroidProducts.mk) \
28 $(SRC_TARGET_DIR)/product/AndroidProducts.mk
29endif

```

然后,在 `product_config.mk` 中调用该脚本函数,并据此添加不同的 `product`。关于什么是一个 `product` 的具体定义及实现详见后面小节。

② 扫描完 `product` 后,就需要根据当前主机系统的类型及 `product` 的类型,定义相应的环境变量,比如设置目标系统的类型,设置目标输出路径等。

(6) 包含不同的 `product` 中的 `BoardConfig.mk` 文件。该文件中会定义不同 `product` 中包含的一些特殊属性,这些属性将被添加到最终目标的系统属性中。

(7) 设置各种工具对应的命令宏,比如 `AAPT` 命令宏对应的 `aapt` 工具。

至此, `config.mk` 的工作就结束了。

③ 确保主机系统满足必要的编译条件。比如 `Android` 源码只能在 `Linux` 系统和苹果 (`Darwin`) 上进行编译,并且 `Android2.3` 版本以后要求主机必须是 64 位系统,等等。

④ 检查 `Java` 及 `Javac` 版本的兼容性, `Android 2.3` 版本以后只能使用 `Java1.6` 版本进行编译。

⑤ 包含 `definitions.mk` 文件。正如其名称所指,该文件中包含了一些系统编译过程所需的各种脚本函数的定义,相当于一个函数库。

⑥ 包含 `dex_preopt.mk`。`preopt` 的意思是 `pre-optimazition`,即事先优化。事先优化的意思是, `Dalvik` 虚拟机中执行的是 `dex` 文件,不同的 `CPU` 会略有差异,为了最大化 `Dalvik` 的性能, `dex` 在执行前可以先根据 `CPU` 的特性做适当的优化,当然,优化后的 `dex` 就仅适应于特定的 `CPU`,而不是特定的操作系统。因此,对于普通应用程序而言,一般不进行 `dex` 优化,本步中包含的 `dex_preopt.mk` 文件可以设置 `dex` 优化相关的配置。

⑦ 检查 `product` 类型 (`variant`) 的合法性。`variant` 的类型只能是四种,即 `user`、`userdebug`、`eng`、`tests`,如果用户指定了一个非法的 `variant` 名称,编译系统将终止编译。

⑧ 检查 `product` 中的 `PRODUCT_TAGS` 是否包含 `dalvik.gc.type-precise`。如是,则需要给 `ADDITIONAL_BUILD_PROPERTIES` 变量中添加相应的值。

⑨ 如果 `product` 中没有包含 `apns-conf.xml` 文件,则使用默认的 `apns-conf.xml` 文件。该文件的作用是定义世界各国移动网络服务商的服务器配置信息,也就是大家常说的中国移动的彩信和 `WAP` 配置。

⑩ 定义 `SDK` 编译默认是 `tiny` 模式所包含的子项目。`Android` 源码中有很多子项目,而 `SDK` 并不会包含所有这些子项目,同样,有一种 `product` 的名称叫做 `tiny`,源码注解中说,它是用于调试系统内核所定义的一种 `product`。无论是 `SDK` 模式还是 `tiny` 模式,都仅包含了一部分子项目,而具体包含哪些子项目正是在这里进行配置的。

⑪ 以上步骤完成后,编译中枢已经掌握了系统中所有的 `product` 名称,并定义了各种所需要的脚本函数,接下来就要根据用户指定的 `product` 名称选择所包含的子项目。源码中调用 `build/tools/findleaves.py` 脚本产生变量 `subdir_makefiles` 的内容,如以下代码所示。



```
subdir_makefiles := \
$(shell build/tools/findleaves.py --prune=out
--prune=.repo --prune=.git $(subdirs) Android.mk)
```

在以上脚本中, `--prune` 用于排除搜索目录, 变量 `subdirs` 的值为 `$(TOP)`, 也就是 Android 根目录。因此, 以上脚本的意思就是: 从 Android 根目录下寻找所有子目录中的 `Android.mk` 文件, 但是要排除 `out` 目录、`.repo` 目录及 `.git` 目录。

获得 `subdir_makefiles` 后, 源码中再使用 `include` 命令包含这些 `Android.mk` 文件, 如下代码所示:

```
503 include $(subdir_makefiles)
```

试想, 假设每一个 `Android.mk` 文件最终被展开的格式是一个标准的 `make` 目标, 那么 `include` 后, 就会产生前面所设想的 `make` 脚本文件, 如图 18-2 所示。

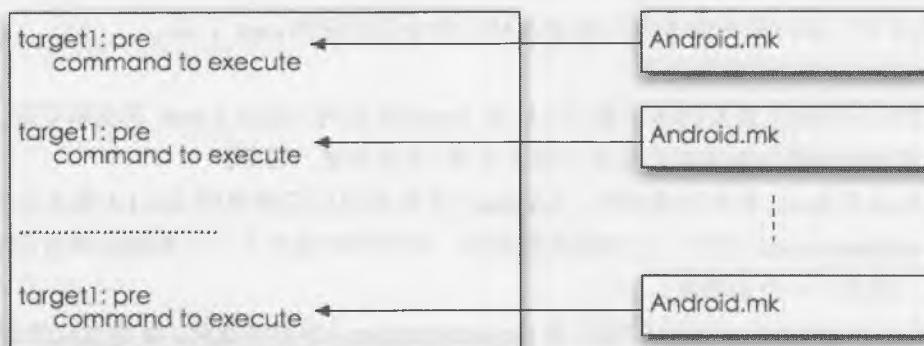


图 18-2 主 make 文件和子项目 make 文件的关系

⑫ 如果是全编译模式 (全编译模式就是非 SDK 模式), 则忽略 `frameworks/policies/base/PolicyConfig.mk` 文件的存在性。注意忽略命令是 `-include`, 即 `include` 前面加一个减号。

⑬ 根据 `variant` 的值选择相应 `tag` 对应的子项目。前面讲过, 用户可以在 `make` 命令后指定 `product` 的名称为子类型 (`variant`) 的类别, 其格式为 `PRODUCT-productName-variant`, 这里的 `variant` 对应的就是子项目的 `tag`。编译中枢内部是在 `base_rules.mk` 中进行 `tag` 值合法性检查的, 如下代码所示。

```
68 ifeq ($(filter-out user debug eng tests optional samples shell_ash shell_mksh,$(LOCAL_MODULE_TAGS)),)
69 $(warning unusual tags $(LOCAL_MODULE_TAGS) on $(LOCAL_MODULE) at $(LOCAL_PATH))
70 endif
```

检查的 `tag` 类别如下, 只有这些 `tag` 值才真正有意义。否则, 该子项目在编译后不会被添加到任何的 `product` 中。

- `user`: 设备厂商私有的项目, 这些项目一般依赖于不同的硬件平台。
- `optional`: 可选项目, 当需要每一个 `product` 中都包含该项目时, 需要使用该 `tag`, 同时需要把该项目添加到 `product` 中的 `PRODUCT_PACKAGES` 变量中, 关于这点参见后面小节。

- debug、eng、tests: 仅仅是两个 tag 名称而已, 对应用户调用 make 命令参数中的 variant 值, 也就是说当 variant 为 debug 时, 该产品中将包含所有 tag 为 debug 的子项目。
- samples: 仅仅是一个 tag, 暂不与任何 variant 对应, 是一些独立的子项目。
- shell\_ash、shell\_mksh: 暂不清楚。

总之, tag 和 variant 并不完全是一一对应的, 其关系可总结为图 18-3 所示。

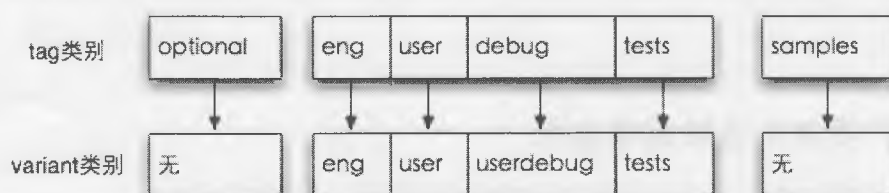


图 18-3 不同 variant 所对应的 tag 类别

14 包含 Makefile 文件, 该文件内部定义了一些编译过程中所需的各种临时目标对象。

15 最后, 定义其他 target, 这些 target 多数并不属于某个子项目, 而是整个 Android 源码可以生成的目标, 包括以下几种。

- files: 生成编译过程中所需的各种临时文件, 包含各种中间所需的目标文件。
- checkbuild: 编译所有子项目, 这是 Android 开发组内部用于调试的一个目标。
- ramdisk: 生成 ramdisk.img。
- systemtarball: 生成 system.img 的压缩文件。
- boottarball: 生成 boot.img 的压缩文件。
- userdataimage: 生成 userdata image 文件。
- bootimage: 生成 boot image 文件。
- droidcore、droid: 两者含义基本相同, 即编译整个 Android 系统。当用户调用 make 而没有指定 target 时, 默认的 target 是 droid。
- apps\_only: 在默认情况下, 该 target 并不存在, 因为它仅在 TARGET\_BUILD\_APPS 变量被定义的情况下才存在, 而这个变量默认并没有被定义。
- droid: 依赖于 droidcore 和 dist\_libraries 两个目标, 在默认情况下即产生该目标。
- tests: 等价于 droidcore。
- all\_modules: 所有子项目, 与 checkbuild 对应的目标相同。
- docs: 生成 SDK 所需要的文档。
- sdk: 生成 SDK。

- **findbugs:** 是一个子项目, 该工具主要用于自动搜索源码中的语法的正确性, 并不是逻辑的正确性, 而是程序编码风格的正确性。
- **clean:** 当用户调用 `make clean` 时, `clean` 本身也被当成了一个 `target` 处理。
- **modules:** 一些其他的子项目, 主要是包含各种用于测试的子项目。
- **showcommands:** 用于显示可用的编译命令参数。

以上就是编译中枢的全部执行过程, 后面小节将根据工程实践中遇到的常见问题再分别介绍一些编译细节。

### 18.3.3 从子项目中提取编译目标

上一节中讲到, 每个子项目中都有一个 `Android.mk`, 编译中枢会读取这个文件并生成子项目的编译目标, 那么, 这个编译目标具体是如何生成的呢?

首先, 在 `Android.mk` 中, 通过给变量 `LOCAL_MODULE` 或 `LOCAL_PACKAGE_NAME` 赋值指定该项目的名称, 然后使用 `include` 命令包含需要编译的目标类型, 比如 `include $(BUILD_PACKAGE)`, 而不同的目标类型本质上都是一个 `.mk` 文件, 这在 `config.mk` 中有定义, 如以下源码所示。

```
CLEAR_VARS:= $(BUILD_SYSTEM)/clear_vars.mk
BUILD_HOST_STATIC_LIBRARY:= $(BUILD_SYSTEM)/host_static_library.mk
BUILD_HOST_SHARED_LIBRARY:= $(BUILD_SYSTEM)/host_shared_library.mk
BUILD_STATIC_LIBRARY:= $(BUILD_SYSTEM)/static_library.mk
BUILD_RAW_STATIC_LIBRARY := $(BUILD_SYSTEM)/raw_static_library.mk
BUILD_SHARED_LIBRARY:= $(BUILD_SYSTEM)/shared_library.mk
BUILD_EXECUTABLE:= $(BUILD_SYSTEM)/executable.mk
BUILD_RAW_EXECUTABLE:= $(BUILD_SYSTEM)/raw_executable.mk
BUILD_HOST_EXECUTABLE:= $(BUILD_SYSTEM)/host_executable.mk
BUILD_PACKAGE:= $(BUILD_SYSTEM)/package.mk
BUILD_HOST_PREBUILT:= $(BUILD_SYSTEM)/host_prebuilt.mk
BUILD_PREBUILT:= $(BUILD_SYSTEM)/prebuilt.mk
BUILD_MULTI_PREBUILT:= $(BUILD_SYSTEM)/multi_prebuilt.mk
BUILD_JAVA_LIBRARY:= $(BUILD_SYSTEM)/java_library.mk
BUILD_STATIC_JAVA_LIBRARY:= $(BUILD_SYSTEM)/static_java_library.mk
BUILD_HOST_JAVA_LIBRARY:= $(BUILD_SYSTEM)/host_java_library.mk
BUILD_DROIDDOC:= $(BUILD_SYSTEM)/droiddoc.mk
BUILD_COPY_HEADERS := $(BUILD_SYSTEM)/copy_headers.mk
BUILD_KEY_CHAR_MAP := $(BUILD_SYSTEM)/key_char_map.mk
```

以上 `.mk` 文件的引用关系如图 18-4 所示。

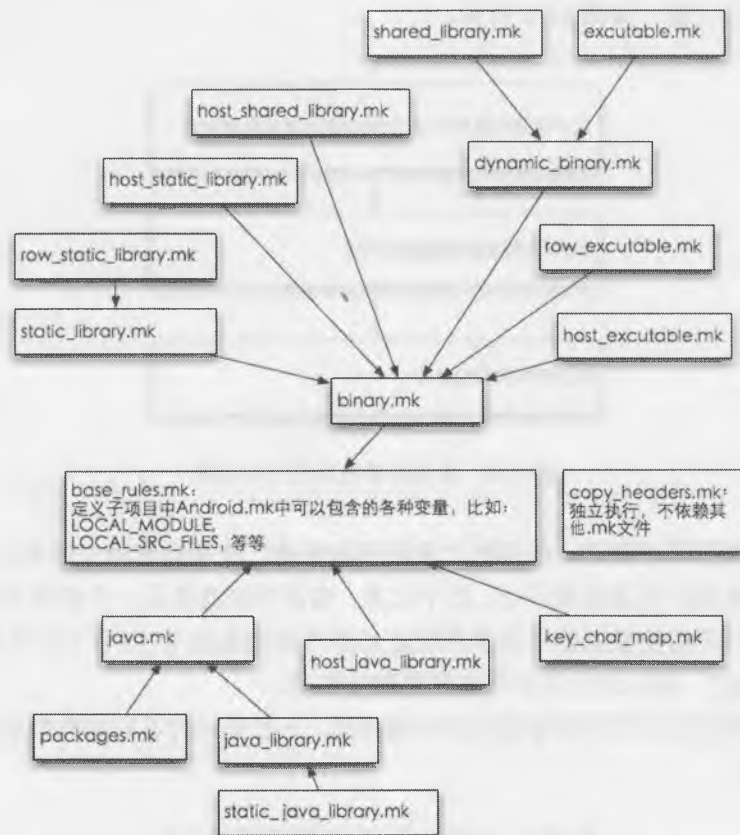


图 18-4 不同编译目标对应的脚本文件关系

从该图可以看出，各种目标类型最终都会引用到 `base_rules.mk`，而在该文件的最后有以下一段代码：

```
558 ALL_MODULES += $(LOCAL_MODULE)
```

这就把子项目中定义的目标添加到了 `ALL_MODULES` 变量中，从而使编译中枢获知了子项目的目标。

#### 18.3.4 生成编译规则

上一节讲到，`config.mk` 中定义了各种编译规则使用的宏，比如 `BUILD_PACKAGES`。子项目中可以通过 `include` 命令包含不同的宏，从而告诉编译中枢该项目被输出的目标类型。读者都知道，无论是编译 `.java` 文件还是编译 `.c/cpp` 文件，最终都必须调用相应的编译器进行编译，那么，上面这些编译规则定义的宏又是如何被展开为具体的编译命令的呢？

本节就以 `BUILD_PACKAGES` 为例说明该过程，其他编译规则的展开过程与此类似，不再赘述。

该文件主要包含三部分功能，如图 18-5 所示。

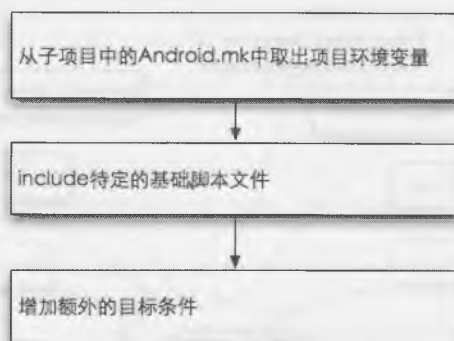


图 18-5 生成编译规则的三个步骤

第一部分，取出项目环境变量。所谓的“项目环境变量”就是指编译中枢中定义的一些特定变量，子项目只需要给这些特定的变量赋值即可。打个比方，编译中枢就像是一个操作系统，而每一个子项目就像是应用程序，项目环境变量就像是操作系统定义的“回调函数”，每个子项目只需要在 Android.mk 中实现这些“回调函数”，即给这些项目环境变量赋值即可。

编译中枢中为子项目定义的环境变量大部分都相同，不过也针对不同的类型定义了不同的变量，这些变量如表 18-2 所示。

表 18-2 packages.mk 中定义的环境变量

编译中枢中定义的 项目环境变量	Android.mk 中 赋值的环境变量
LOCAL_PACKAGE_NAME	相同
LOCAL_MODULE_SUFFIX	COMMON_ANDROID_PACKAGE_SUFFIX
LOCAL_MODULE	LOCAL_PACKAGE_NAME
LOCAL_MANIFEST_FILE	固定值，AndroidManifest.xml
LOCAL_MODULE_CLASS	APPS
LOCAL_MODULE_TAGS	相同，如果没有定义，则默认使用 optional
LOCAL_AAPT_FLAGS	相同，如果没有定义，默认使用\$(LOCAL_AAPT_FLAGS) -z
LOCAL_ASSET_DIR	相同，如果没有指定，默认使用\$(LOCAL_PATH)/assets
LOCAL_RESOURCE_DIR	相同，如果没有指定，默认使用:= \$(LOCAL_PATH)/res
LOCAL_AIDL_INCLUDES	相同，但同时包含 \$(FRAMEWORKS_BASE_JAVA_SRC_DIRS)

关于这些变量的赋值需要注意一点，在装载 assets 及 res 资源时，除了当前项目中的 asset 和 res 子目录外，packages.mk 还从另外两个目录下寻找该项目的资源，分别是

PRODUCT\_PACKAGE\_OVERLAYS 和 DEVICE\_PACKAGE\_OVERLAYS。这就给 OEM 厂商提供了修改 Android 源码的一种架构，它们可以扩展原有的项目，并在这两个厂商目录中创建一个和原项目文件夹相同的文件夹，并将相关 res 或者 assets 目录放到里面。这样的话，项目在编译时会自动将这两个目录中的资源文件包含到项目中，而不需要直接修改原项目中的资源文件。

第二部分，包含特定的基础脚本文件。对于 BUILD\_PACKAGES 而言，就是 java.mk，该文件中定义了如何编译 Java 源文件。当然，java.mk 中调用了 definitions.mk 文件，该文件中真正定义了各种使用 Java、javac、gcc 等编译器的调用规则。

第三部分，增加额外的目标条件。对于 BUILD\_PACKAGES 而言，额外的目标条件主要包含以下目标条件。

(1) 增加 R.java。因为几乎所有的项目源码中都包含一些资源文件，这些资源文件必须首先被 aapt 编译成 R.java，之后其他的 Java 源码才能顺利被编译，因此，需要首先添加该目标条件。脚本中对应的源码如下。

```
195$(R_file_stamp): PRIVATE_RESOURCE_PUBLICS_OUTPUT := \
196     $(intermediates.COMMON)/public_resources.xml
197$(R_file_stamp): PRIVATE_PROGUARD_OPTIONS_FILE := $(proguard_options_file)
198$(R_file_stamp): $(all_res_assets) $(full_android_manifest) $(AAPT) | $(ACP)
199     @echo "target R.java/Manifest.java: $(PRIVATE_MODULE) ($@)"
200     @rm -f $@
201     $(create-resource-java-files)
```

(2) 增加 JNI 的目标条件，因为有些项目会在内部包含一些 JNI 文件。

(3) 增加对签名文件 (4) 依赖条件，因为 Java 项目最后都需要被签名。

(4) 增加对 AAPT 及 ZIPALIGN 的条件依赖。AAPT 是用于进行 APK 打包的，ZIPALIGN 是用于对打包后的文件进行内部存储空间优化以提高装载性能的。

至此，BUILD\_PACKAGES 对应的编译规则就转换完毕了，最后展开的 make 脚本文件中完整地描述了这个项目的 target 名称，以及该 target 所依赖的各种条件目标，以及执行的命令。

### 18.3.5 设置编译输出目录

前面曾经提出过这个问题，编译中枢怎么知道应该把编译目录放到 out 目录中呢？并且该目录中又包含各种子目录，那么，这种目录规则是在哪里被定义的呢？

答案就是 envsetup.mk 文件，如以下源码所示。

```
161# -----
162# figure out the output directories
163
164if [ -n "$(strip $(OUT_DIR))" ]
165OUT_DIR := $(TOPDIR)out
166endif
```

该段代码从 Android 的根目录定义开始，首先定义总的输出目录为 `out`，然后再依次定义调试输出目录 `debug`，只不过这个目录似乎只有 Android 内部调试工程师在使用。

接着定义 `HOST` 和 `TARGET` 的子目录，分别为 `host`、`target`，这就是大家经常在 `out` 目录中看到的 `host` 和 `target`。再接下来的定义就很简单了，可对照 `out` 目录和该脚本中的源码。

关于这种目录结构，有的读者就要问了，是不是可以任意修改呢？理论上讲的确可以任意修改，只是编译中枢并没有被设计成百分之百的变量型结构。所谓的变量型结构是指，任何信息都保存在变量中，其他地方需要这个信息时必须通过变量，而不能硬编码，这种结构具有非常好的灵活性。Android 编译中枢虽然大多时候都是变量型结构，不过也有少数引用是直接编码，比如这个总的输出目录的名称“`out`”，在 `main.mk` 文件调用 `findleaves.py` 脚本时，参数中就直接使用了 `out`，而并没有通过引用变量 `OUT_DIR`。因此，大家在修改这些输出路径时可能会碰到一些引用错误的情况，所以不建议对这个输出目录做修改，除非有特别的需求。

### 18.3.6 生成最终的 Image 文件

有些读者会问，最终的 Image 文件似乎是将相关的子项目输出进行一定格式的打包，那么这种打包的规则到底是什么呢？

的确是这样的，最终生成的各种 Image 文件都仅仅是对相应的目标进行压缩或者格式转换，源码中首先在 `main.mk` 中将这生成 Image 的操作也定义成一个 `make` 的 `target`，如以下代码所示：

```
677.PHONY: ramdisk
678ramdisk: $(INSTALLED_RAMDISK_TARGET)
679
680.PHONY: systemtarball
681systemtarball: $(INSTALLED_SYSTEMTARBALL_TARGET)
682
683.PHONY: boottarball
684boottarball: $(INSTALLED_BOOTTARBALL_TARGET)
685
686.PHONY: userdataimage
687userdataimage: $(INSTALLED_USERDATAIMAGE_TARGET)
688
689.PHONY: userdatatarball
690userdatatarball: $(INSTALLED_USERDATATARBALL_TARGET)
691
692.PHONY: bootimage
693bootimage: $(INSTALLED_BOOTIMAGE_TARGET)
```

这些 `target` 一般不需要用户直接在调用 `make` 时指定，而大多时候用于其他 `target` 的依赖条件。比如当 `make sdk` 时，`sdk` 这个 `target` 就会依赖 `ramdisk`、`systemtarball` 等目标，换句话说，这些 `target` 本身就是 `sdk` 工作的一部分。

而真正产生这些 Image 目标的命令是在对应的命令宏中被定义的，比如 `systemtarball` 对应的命令宏是 `INSTALLED_SYSTEMTARBALL_TARGET`，这个命令是在 `./build/core/Makefile` 文件中被定义的，如以



下代码所示，而 Makefile 文件是在 main.mk 中调用 include 包含进去的，而且是在定义这些 target 之前。

```
788 system_tar := $(PRODUCT_OUT)/system.tar
789 INSTALLED_SYSTEMTARBALL_TARGET := $(system_tar).$(SYSTEM_TARBALL_FORMAT)
790 $(INSTALLED_SYSTEMTARBALL_TARGET): PRIVATE_SYSTEM_TAR := $(system_tar)
791 $(INSTALLED_SYSTEMTARBALL_TARGET): $(FS_GET_STATS) $(INTERNAL_SYSTEMIMAGE_FILES)
792     $(call build-systemtarball-target)
```

从该段代码中可以看出，该命令宏本身也有一些依赖条件，其中最后一个依赖条件是 INTERNAL\_SYSTEMIMAGE\_FILES 变量。该变量实际上就是 out 目录中包含的 system 目录下所有文件，而该条件的执行命令是调用 build-systemtarball-target 函数，该函数就将 system 目录下的内容打包为一个压缩文件。

不过需要注意的是，这里仅仅是产生 system 目录的压缩文件，而并不是产生 system.img。事实上，system.img 文件是一种 ext4 文件格式的映像文件，它的目标依赖关系如下。

首先，在 Makefile 中定义了一个内部目标 BUILD\_SYSTEMIMAGE，这个目标依赖的条件就是 system 目录下的所有文件，如以下代码所示。

```
738 $(BUILD_SYSTEMIMAGE): $(INTERNAL_SYSTEMIMAGE_FILES) $(INTERNAL_USERIMAGES_DEPS)
739     $(call build-systemimage-target,$@)
```

而该目标的执行命令是调用 build-systemimage-target 函数，该函数内部又分别针对是否使用 EXT 模式有两个定义。在 Android 2.2 版本以前，system.img 的文件系统都是 yaffs，而从 Android 2.3 开始，system.img 开始使用 ext4 文件系统。该函数最终调用到了 build-userimage-ext-target 函数，该函数内部调用了 MKEXTUSERIMG 命令宏，而该宏是在 config.mk 中定义的，如以下代码所示。

```
1 MKEXTUSERIMG := $(HOST_OUT_EXECUTABLES)/mkuserimg.sh
```

而 mkuserimg.sh 脚本中则是调用 make\_ext4fs 工具将 system 目录转换成了 ext4 文件系统的影响文件，关于如何使用 make\_ext4fs 工具产生 system.img 的详细内容参见第 19 章。

关于其他 Image 文件的产生过程思路与 system.img 相似。

## 18.4 如何增加一个 product

前面讲过，编译 Android 源码时，调用 make 命令的参数格式为 PRODUCT-productName-variant，其中 productName 是产品的名称，variant 是产品的类型。类型只能是有限的几种，分别为 eng、userdebug、user、tests，而产品的名称则是任意的，但这些产品名称必须在编译开始之前事先定义好。那么到底什么是一个产品？其代码层面的意义是什么？如何添加一个产品，本节就来回答这些问题。

### 18.4.1 什么是一个 product

关于 product 的意义可如图 18-6 所示。

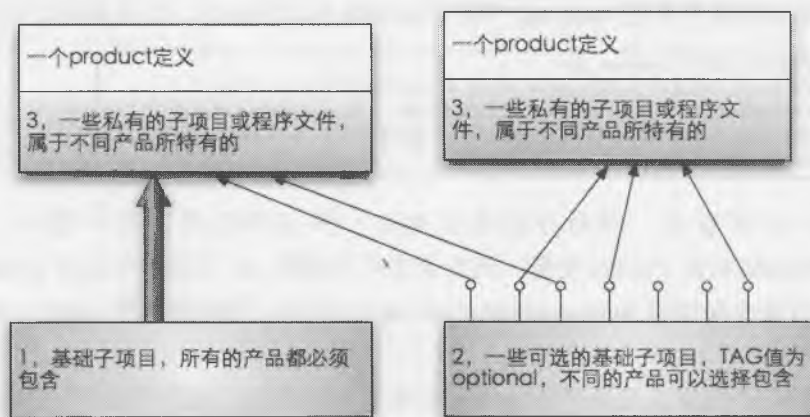


图 18-6 product 概念的定义

Android 源码中的子项目从 product 的角度来讲可以分为三个部分。

第一部分，基础子项目，这些项目是所有 product 都应该包含的，因为它是 Android 系统运行所必需的项目，比如 `adbd`，所有的 Android 系统都必须包含该项目，否则 `adb` 将无法工作；再比如 `Binder` 驱动，如果没有 `Binder` 驱动，Android 系统将无法运行。

第二部分，可选的基础子项目。这里所讲的“基础”是指 Android 源码中自带的，“可选的”是指它并不是 Android 系统运行所必需的，比如一些 APK 程序，基本上全部是可选的，其 `LOCAL_MODULE_TAG` 都是 `optional`，就算没有这些 APK 项目，Android 系统也可以正常启动并运行，因此它是可选的。

第三部分，一些私有的子项目或者文件。这些子项目是 Android 源码中所没有的，而是设备上提供的一些特有的项目或者文件。

在 Android 的编译系统中，可以创建一些新的 `make` 脚本文件，并在这些脚本文件中描述一个 product。描述的信息中应该指出该 product 所包含的子项目名称，从而使得编译中枢对这些子项目进行编译，并最终生成相应的 `system.img` 及其他各种输出目标。这些新的脚本文件的名称和所在路径必须按照编译中枢的约定进行定义。

编译中枢中定义了一组和 product 相关的变量，每一个 product 的本质是给这组变量赋不同的值，用于代表一种 product。因此，product 的本质实际上是一组信息组合，这种组合的目标是告诉编译中枢该 product 都包含哪些子项目。

那么，这组变量是什么呢？

这组变量是在 `product.mk` 文件中定义的，如以下源码所示。

```

62 _product_var_list := \
63     PRODUCT_NAME \
64     PRODUCT_MODEL \
65     PRODUCT_LOCALES \
66     PRODUCT_PACKAGES \
67     PRODUCT_DEVICE \
68     PRODUCT_MANUFACTURER \
69     PRODUCT_BRAND \
70     PRODUCT_PROPERTY_OVERRIDES \
71     PRODUCT_CHARACTERISTICS \
72     PRODUCT_COPY_FILES \
73     PRODUCT_OTA_PUBLIC_KEYS \
74     PRODUCT_PACKAGE_OVERLAYS \
75     DEVICE_PACKAGE_OVERLAYS \
76     PRODUCT_CONTRIBUTORS_FILE \
77     PRODUCT_TAGS \
78     PRODUCT_SDK_ADDON_NAME \
79     PRODUCT_SDK_ADDON_COPY_FILES \
80     PRODUCT_SDK_ADDON_COPY_MODULES \
81     PRODUCT_SDK_ADDON_DOC_MODULE \
82     PRODUCT_DEFAULT_WIFI_CHANNELS

```

为了有一个感性的认识,读者可以先来看看 Android 源码都自带了哪些 product。查看的方法是在 make 脚本中调用 product.mk 中定义的 dump\_products 函数。具体的讲,可以在 Android 根目录下的 Makefile 文件添加该脚本函数调用,如以下代码所示。

```

### DO NOT EDIT THIS FILE ###
include build/core/main.mk

$(call dump-products)
### DO NOT EDIT THIS FILE ###

```

然后在命令行下调用 make -n 命令, -n 参数的含义是仅仅打印出执行的命令,而不真正执行命令。从输出的信息中,读者可以看到源码中已经包含的 product 名称,以及该 product 中以上这组变量的值。举例来讲,full\_crespo 对应的部分变量值如下。

```

==== device/samsung/crespo/full_crespo.mk ====
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_NAME := full_crespo
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_MODEL := Full Android on Crespo
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_LOCALES := en_US ...
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_PACKAGES := ...
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_DEVICE := crespo
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_MANUFACTURER :=
PRODUCTS.device/samsung/crespo/full_crespo.mk.PRODUCT_BRAND := Android

```

由于篇幅原因,以上省略了部分信息,建议读者实际操作一下,看看输出结果。

如果只想查看某个特定 product 的信息,可以使用 dump-products 函数,如以下代码所示:

```

### DO NOT EDIT THIS FILE ###

```

```
include build/core/main.mk

$(call dump-product,device/samsung/crespo/full_crespo.mk)

### DO NOT EDIT THIS FILE ###
```

那么，定义这些 **product** 的脚本文件都在哪里呢？理论上讲，这些脚本文件可以放在任何目录下，只要这些脚本文件能被编译中枢包含进去，并且这些脚本文件对这组变量进行了赋值即可。当然，如果你不确定这些脚本文件是否会被编译中枢自动包含进去，则可以在根目录下的 **Makfile** 文件中手动使用 **include** 命令包含自定义的脚本文件，不过这种做法会增加调试的复杂性。因为你必须十分清楚每一个变量的确切含义，并且要处理这些变量和编译中枢之间的关系，因为每一个 **product** 中都会定义这些变量，如何将不同 **product** 中定义的变量进行复合，都需要特别的处理。

实际上，这些 **product** 的定义文件都有指定的目录，编译中枢已经事先约定好了这种目录结构，从而简化 **product** 的定义。编译中枢先后读取了三个目录寻找 **product** 定义的脚本文件，这三个目录分别如下。

- **build/target/product/**目录下的 **AndroidProducts.mk**。该文件是源码中自带的，其内部又分别包含了七个 **product** 定义的文件，如以下源码所示。

```
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/core.mk \
    $(LOCAL_DIR)/generic.mk \
    $(LOCAL_DIR)/generic_x86.mk \
    $(LOCAL_DIR)/full.mk \
    $(LOCAL_DIR)/full_x86.mk \
    $(LOCAL_DIR)/sdk.mk \
    $(LOCAL_DIR)/sim.mk
```

而在以上七个文件的任何一个中，都对 **product** 相关的那组变量进行了赋值。这里请注意，**PRODUCT\_MAKEFILES** 变量是编译中枢中定义的，中枢中正是使用该变量对不同 **product** 中定义的相关变量值进行了复合。

- **device** 目录下 6 层目录深度中的所有 **AndroidProducts.mk**。
- **vendor** 目录下 6 层目录深度中的所有 **AndroidProducts.mk**。

因此，如果需要定义一个新的 **product**，建议放到 **device** 或者 **vendor** 目录中。

在不同的 **product** 定义文件中，需要指定都包含哪些子项目。其中基础类项目不需要指定；对于 **optional** 类项目，需要添加到 **PRODUCT\_PACKAGES** 变量中，否则该可选类项目不会被包含到该 **product** 中；对于那些私有的项目，也需要添加到该变量中；对于那些私有的文件，可以添加到 **PRODUCT\_COPY\_FILES** 变量中，从而告诉编译中枢将指定文件复制到特定的路径下。该变量的赋值格式为 **src:des**，即用一个冒号分隔源文件和目标文件路径，比如：

```
PRODUCT_COPY_FILES += \
    vendor/samsung/crespo/proprietary/gps.conf:system/etc/gps.conf
```

## 18.4.2 如何增加一个 product

上一节了解了 product 的具体含义，本节就来实际尝试添加一个 product。

添加过程可分为以下几步。

① 根据公司名称新建一个包含脚本文件的文件夹。假设我们的公司名称为 haiii，这个产品名称定义为 k，那么可以在 device 目录下执行以下命令，创建相应的文件夹。

```
cd path/to/android
cd vendor
mkdir -p haiii/k
```

② 在 k 目录下新建一个 AndroidProducts.mk 文件。注意，这个文件的名称是固定的，不能修改，并且该文件的内容应该仅仅是为变量 PRODUCT\_MAKEFILES 赋值，如以下代码所示。

```
#device/haiii/k/AndroidProducts.mk
PRODUCT_MAKEFILES := \
    $(LOCAL_DIR)/k.mk
```

③ 新建一个上一步中指定的 mk 脚本文件，并在该文件中对 product 相关的变量赋值，最简单的情况如以下代码所示。

```
#k.mk
PRODUCT_PACKAGES := \
    helloMake

PRODUCT_NAME := full_k
PRODUCT_DEVICE := k
PRODUCT_BRAND := Android
```

④ 在以上目录中新建一个 BoardConfig.mk 文件，名称必须是固定的，因为编译中枢需要该文件，哪怕该文件的内容为空。由于本例仅仅是为了演示如何新建一个 product，而该 product 并没有真正的 Linux 内核，也没有相应的音频库，也没有 camera 库，因此，可以在 BoardConfig.mk 文件中对相关的变量做以下赋值，否则不能正确编译。

```
TARGET_CPU_ABI := armeabi-v7a
TARGET_NO_BOOTLOADER := true
TARGET_NO_KERNEL := true

#no real audio, so use generic
BOARD_USES_GENERIC_AUDIO := true

# no hardware camera
USE_CAMERA_STUB := true
```

下面来验证该 product 的有效性。

首先修改 Android 根目录下的 Makefile 的内容如下：

```
### DO NOT EDIT THIS FILE ###
include build/core/main.mk

$(call dump-product,device/haiii/k/k.mk)
### DO NOT EDIT THIS FILE ###
```

然后就可以使用 `make PRODUCT-full_k-eng`, 编译完毕后的输出中将包含名称为 `k` 的 `product` 的相关变量值, 如以下代码所示。

```
==== device/haiii/k/k.mk ====
PRODUCTS.device/haiii/k/k.mk.PRODUCT_NAME := full_k
PRODUCTS.device/haiii/k/k.mk.PRODUCT_MODEL :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_LOCALES :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_PACKAGES := helloMake
PRODUCTS.device/haiii/k/k.mk.PRODUCT_DEVICE := k
PRODUCTS.device/haiii/k/k.mk.PRODUCT_MANUFACTURER :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_BRAND := Android
PRODUCTS.device/haiii/k/k.mk.PRODUCT_PROPERTY_OVERRIDES :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_CHARACTERISTICS :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_COPY_FILES :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_OTA_PUBLIC_KEYS :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_PACKAGE_OVERLAYS :=
PRODUCTS.device/haiii/k/k.mk.DEVICE_PACKAGE_OVERLAYS :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_CONTRIBUTORS_FILE :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_TAGS :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_SDK_ADDON_NAME :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_SDK_ADDON_COPY_FILES :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_SDK_ADDON_COPY_MODULES :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_SDK_ADDON_DOC_MODULE :=
PRODUCTS.device/haiii/k/k.mk.PRODUCT_DEFAULT_WIFI_CHANNELS :=
```

以上就完成了定义一个最简单的 `product` 模型, 在实际应用中, 特定的 `product` 一般还需要包含一些私有的子项目或者文件, 关于这点读者可以参考 `crespo` 中的 `AndroidProducts.mk` 文件中相关变量的赋值。

## 18.5 如何增加一个项目

所谓增加一个项目的意思是, 增加一个子项目, 以便于 `Android` 源码编译后, 编译目标中能够包含该项目。比如, 可以增加一个 `C/C++` 工具程序, 从而可以在目标系统的 `shell` 命令行下使用该工具。再比如, 可以增加一个 `APK` 程序, 从而在设备出厂时就自带该程序。

从编译中枢的角度来讲, 增加项目就是要告诉编译中枢一些信息, 这些信息包括:

- 这个项目的 `target` 名称是什么?
- 这个项目的输出类型是什么? 比如 `APK`、`Jar` 包等。
- 这个项目对应的 `variant` 是什么?

有了这些信息后, 编译中枢就知道如何编译这个项目, 并将输出的目标打包到不同的 `variant` 产品

分类中。

本节将首先介绍不同的项目类别，然后介绍两种常见的项目添加方法。

### 18.5.1 项目类别和项目路径

编译中枢中已经定义的项目类别是在 `config.mk` 中定义的，常见的类别如表 18-3 所示。

表 18-3 编译系统内定义的不同编译目标

项目类别对应的命令宏	意 义
<code>BUILD_HOST_STATIC_LIBRARY</code>	主机上的二进制静态库
<code>BUILD_HOST_SHARED_LIBRARY</code>	主机上的二进制动态库
<code>BUILD_HOST_EXECUTABLE</code>	主机上的可执行程序
<code>BUILD_STATIC_LIBRARY</code>	目标设备上的二进制静态库
<code>BUILD_SHARED_LIBRARY</code>	目标设备上的二进制动态库
<code>BUILD_EXECUTABLE</code>	目标设备上的二进制可执行程序
<code>BUILD_PACKAGE</code>	APK 程序
<code>BUILD_JAVA_LIBRARY</code>	目标设备上的共享 Java 库
<code>BUILD_STATIC_JAVA_LIBRARY</code>	目标设备上的静态 Java 库
<code>BUILD_HOST_JAVA_LIBRARY</code>	主机上的 Java 共享库

不同的项目类别中会包含一些特定内部变量，具体可参考源码中相应项目下的 `Android.mk` 文件。

关于项目的路径，前面讲过，编译中枢会遍历 Android 根目录下的所有子目录，除了 `out` 和 `.git`、`.repo` 目录外，并读取所有子目录下的 `Android.mk` 文件。因此，用户自定义的项目可以放到 Android 根目录下的任意目录中，只要在该目录中包含相应的 `Android.mk` 文件即可。尽管理论上是任意的，但为了代码结构的可阅读性，不同的项目一般应该按照用途的不同放到不同的子目录中，常见的情况如下。

- `external` 目录：包含一些二进制的可执行程序或者库文件。
- `packages`：包含一些 APK 程序。

以上两个目录多用于保存 Android 系统所使用的项目，而对于设备商的一些自定义项目，一般可以放到 `vendor` 或者 `device` 目录下。

而如果要更改 Framework 内核的源码，以及添加一些自定义的 Java 类，这就不属于添加一个子项目。从编译中枢的角度来看，并不是说只要添加一个 Java 类中枢就会自动将这个类打包到最后的 `framework.jar` 文件中，而需要做一些额外的工作，关于这点，参见后面小节中如何编译 Framework 一节。



## 18.5.2 添加一个 C 项目

本节就来介绍如何添加一个 C 项目。

首先，需要选择一个子目录，本例读者可以选择 `external` 目录。

接着，需要给该项目定义一个 `target` 名称，这个 `target` 名称必须是全局唯一的，这里可以定义为 `helloMake`。

该工程很简单，只包含一个 `main.c` 文件，其内容如下。

```
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"
int main(){
    makePrintf("hello make");
    return 0;
}
```

`hello.h` 的内容如下。

```
#include <stdio.h>
#include <stdlib.h>

#ifdef _HELLO_H
#define _HELLO_H
void makePrintf(char *str)
{
    printf("%s", str);
}
#endif
```

添加这两个文件后，编写一个 `Android.mk` 文件，内容如下。

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
    main.c
LOCAL_C_INCLUDES += $(LOCAL_PATH)
LOCAL_MODULE := helloMake
LOCAL_MODULE_TAGS := eng
include $(BUILD_EXECUTABLE)
```

该 `mk` 文件中需要注意以下几点。

第一，所有的 C/C++ 源文件必须手动添加，因为编译中枢中并没有提供一个能够自动从当前目录下寻找所有 C/C++ 文件列表的函数。

第二，由于 C 语言中会使用到 `include` 命令，因此，必须在 `mk` 文件中指定头文件的路径，当然仅仅需要指定项目中包含的特别路径，而不需要指定系统头文件路径。因为系统路径在编译中枢中已经设置好了，具体是在 `config.mk` 文件中设定的，如以下代码所示。如果要添加一些特别的系统路径，可以

更改 SRC\_HEADERS 的值。

```

26 SRC_HEADERS := \
27   $(TOPDIR)system/core/include \
28   $(TOPDIR)hardware/libhardware/include \
29   $(TOPDIR)hardware/libhardware_legacy/include \
30   $(TOPDIR)hardware/ril/include \
31   $(TOPDIR)dalvik/libnativehelper/include \
32   $(TOPDIR)frameworks/base/include \
33   $(TOPDIR)frameworks/base/opengl/include \
34   $(TOPDIR)frameworks/base/native/include \
35   $(TOPDIR)external/skia/include

```

第三，该项目使用的 TAG 值为 eng，因此，仅当用户指定的产品类别是 eng 时才会包含该项目。如果要让所有类别的产品都包含该项目，可以将该 TAG 值设为 optional，并在相应的 product 的配置文件中的 PRODUCT\_PACKAGES 变量中包含该项目即可。

然后就可以调用 make PRODUCT-full\_crespo-eng 来编译整个 Android 工程，编译完毕后，可以将新的 system.img 刷入设备，然后启动 ./adb shell，在命令行下，直接调用 helloMake，程序将输出“hello make”。

### 18.5.3 添加一个 APK 项目

添加一个 APK 项目可能是很多应用工程师经常遇到的问题，因此本节就来介绍如何给 Android 源码中添加一个 APK 项目。该过程的原理和添加 C 项目相同，只是 APK 项目中相关的编译变量名称和 C 项目稍有不同。

首先，假设已经在 Eclipse 下新建了一个名称为 HelloApk 的 Android 应用项目，下面要做的就是将这个项目移植到 Android 的源码中。

前面曾讲过，编译中枢会搜索 Android 根目录下除 out 目录外的所有子目录，并找出 Android.mk 文件作为一个子项目。因此，理论上讲可以将 HelloApk 项目放在任何目录下，只不过为了目录的清晰，厂商一般习惯于将自定义的项目放到 vendor 下的相应目录中，因此，本例就将 HelloApk 放到 vendor/haiiii/k/apps 目录下。

然后，删除 HelloApk 目录下的 bin 和 gen 目录，因为这两个目录是 Eclipse 编译的输出目录，而其对 Android 编译系统而言是多余的，它甚至会认为这是两个源码目录。从这点上看，编译中枢似乎需要改进一下，它应该能够自动排出 APK 项目下的 bin 和 gen 目录。

接下来就需要新建一个 Android.mk 文件，并将其放到 HelloApk 目录下，该 Android.mk 的内容如下代码所示。

```

LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_SRC_FILES := $(call all-java-files-under, src)
LOCAL_PACKAGE_NAME := HelloApk

```

```
LOCAL_SDK_VERSION := current
LOCAL_CERTIFICATE := platform
LOCAL_PROGUARD_FLAG_FILES := proguard.cfg
include $(BUILD_PACKAGE)
```

以上 mk 文件说明如下。

LOCAL\_MODULE\_TAGS 用于指定该项目的标签值，对应用类子项目而言，只能指定为三种情况，分别是 optional、tests、eng，这与 C 项目是相同的，本例中使用的是 optional。因此，必须在 device/haiiii/k/k.mk 文件中包含该项目，如以下代码所示，否则 k 产品将不会包含该项目。

```
PRODUCT_PACKAGES := \
    helloMake \
    HelloApk
```

LOCAL\_SRC\_FILE 用于指定该项目中的 Java 源码，编译中枢中提供了一个函数，可以直接遍历指定目录下的所有 Java 文件，从而不必手工列出所有的 Java 文件列表，而 C 项目中却没有类似功能的脚本函数。

LOCAL\_PACKAGE\_NAME 用于描述 APK 项目的名称，而 C 项目中用的是 LOCAL\_MODULE\_NAME，注意区分。

LOCAL\_SDK\_VERSION 用于指定该 APK 项目所需的 SDK 版本，current 代表了当前 Android 源码的版本。

LOCAL\_CERTIFICATE 用于指定该 APK 项目将使用何种签名文件签名最后生成的 apk 文件。系统中一共包含四种签名文件，分别如下。

- platform，Framework 源码最后将生成一个 Jar 包，该 Jar 包默认使用该类型签名。
- shared，一些系统应用程序使用该类型签名，比如 Contacts 等。
- user，一些私有项目使用该签名。
- tests，调试过程中一般使用该签名。

至于选择何种签名取决于项目的需求，一般当需要和特定的程序共享数据库资源时，两个项目必须拥有相同的签名。

LOCAL\_PROGUARD\_FLAG\_FILES，该变量的作用暂不详，不过从实验的结果来看，应该赋值为 Eclipse 项目中的 proguard.cfg 文件。

最后，使用 include 包含 BUILD\_PACKAGE 命令宏告诉编译中枢该项目的目标是一个 APK 程序。

以上介绍的是以源码的方式添加一个项目，有些时候可能还需要以一个 APK 文件的方式添加一个项目，在这种情况下，本质上添加的不是一个项目，而仅仅是一个文件。前面讲过，一个 product 中包含的项目分为三种，其中最后一种就是私有的项目或者文件，而这种方式实际上就是私有的文件，因此，需要在 product 相关的脚本文件中声明该 product 中包含该 apk 文件。本例中就应该在 k.mk 文件中添加以下代码，其中赋值的源路径是主机系统上的路径，而目标路径是设备上的路径，两者使用冒号分隔。

```
PRODUCT_COPY_FILES := \
    vendor/haiiii/k/helloapk/helloapk.apk:/system/app
```

该段代码仅仅是将该 HelloApk.apk 复制到最终设备的/system/app 目录下。有些读者可能会问，如果想给该 APK 进行签名怎么办？对不起，编译中枢中没有单独提供这个功能，也就是说，在把 HelloApk.apk 复制到源码中前，应该事先给该 APK 进行签名。不过如果开发中真的有这种需求，可以尝试增加一个 BUILD\_PACKAGE\_WITHAPK 的命令宏，就像 BUILD\_PACKAGE 一样。用户可以在 Android.mk 文件中使用 include 命令包含该宏，然后在该命令宏的定义中手工实现对 APK 的签名，关于如何签名可参照后面小节。

本小节遗留四个问题请大家思考。

- 如果该 HelloApk 中包含 JNI 代码，又如何将这些 JNI 代码在 Android.mk 文件声明？
- 如果该 HelloApk 中包含 aidl 类型文件，又如何在 Android.mk 文件中声明？
- 如果该 HelloApk 依赖一些特别的 Jar 包，如何在 Android.mk 中指定？
- 本例中对 LOCAL\_SRC\_FILES 赋值时，调用了一个脚本函数，函数的参数 src 代表了 HelloApk 目录下的 src 目录，这似乎告诉编译中枢仅仅从 src 目录下寻找 Java 源码，可为什么还要删除 HelloApk 目录下原来的 gen 和 bin 目录呢？

这四个问题实际上牵扯到一个 APK 的整体编译过程，将在下一节中详述。

## 18.6 APK 编译过程

编译 APK 的脚本是在 package.mk 文件中定义的，同时大家可以使用以下命令查看该脚本的具体执行过程。

```
make Contacts -n > ~/Desktop/mmm.txt
```

该命令以 Contacts 工程为例，把编译 Contacts 工程的编译过程输出到了桌面的 mmm.txt 文本文件中，大家可以对照 package.mk 文件的定义和 mmm.txt 的输出了解编译 APK 的全过程。

### 18.6.1 总体编译过程概述

APK 的总体编译过程如图 18-7 所示。

首先，从一个 APK 项目的文件组成来讲，源文件分别包括：

- 资源文件。
- aidl 源文件。
- Java 源文件。
- Java 静态库。
- Java 共享库。

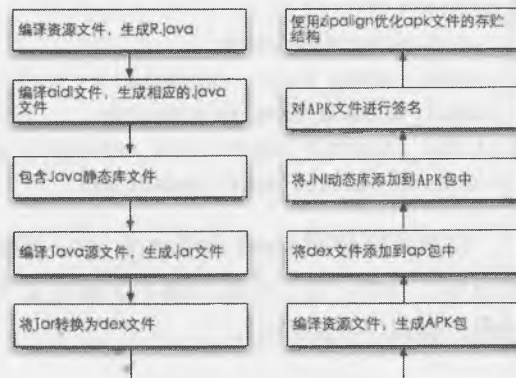


图 18-7 apk 应用程序的编译过程

- 所需的 JNI 动态库。

而 APK 的编译过程也就是对这些源文件的逐个编译过程。另外，以上编译必须遵守一定的顺序，比如，Java 源文件必须在资源文件编译后方可进行。因为 Java 源文件一般会引用到资源文件对应的 R.java，该文件是资源文件编译后的输出结果。

本节将以 HelloApk 项目为例，说明一个 APK 的编译过程，本例中相关的主要源文件清单如下：

```
vendor/haiii/k/apps/HelloApk/Android.mk:
vendor/haiii/k/apps/HelloApk/src/com/haiii/android/helloapk/Welcome.java
vendor/haiii/k/apps/HelloApk/Foo.jar (FooClass.java)
vendor/haiii/k/apps/HelloApk/src/com/haiii/android/helloapk/IHelloApk.aidl
vendor/haiii/k/apps/HelloApk/src/com/haiii/android/helloapk/ApkService.java
```

在以上文件中，Android.mk 是该项目的 make 脚本文件，Welcome.java 是该 APK 项目中的 Activity 类，Foo.jar 是该项目引用的一个静态库，该库中包含一个 FooClass 类。IHelloApk.aidl 文件定义了一个 IBinder 接口，ApkService 是该 IBinder 接口的具体实现。其中 Android.mk 的文件内容如下：

```
LOCAL_PATH:= $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_NO_STANDARD_LIBRARIES := false
LOCAL_SRC_FILES := $(call all-java-files-under, src) \
    src/com/haiii/android/helloapk/IHelloApk.aidl
LOCAL_STATIC_JAVA_LIBRARIES := FooJar

LOCAL_PACKAGE_NAME := HelloApk
#LOCAL_SDK_VERSION := current
LOCAL_CERTIFICATE := platform
LOCAL_PROGUARD_FLAG_FILES := proguard.cfg

include $(BUILD_PACKAGE)

include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_PACKAGE_NAME := AllMyJar
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := FooJar:Foo.jar
include $(BUILD_MULTI_PREBUILT)
```

读者可以使用 `make HelloApk -n > ~/Desktop/helloapk.txt` 编译 HelloApk 项目，并对照输出的文本文件内容。当然，除了从执行脚本的输出文件中查看外，package.mk 中也清晰地给出了编译命令的执行过程，如以下代码所示。

了解了该过程后，大家以后就可以在纯命令行下从任意 res 目录编译出 R.java 文件。

### 18.6.3 编译 aidl 文件

首先需要在 Android.mk 文件中包含项目中的 aidl 文件，如以下代码所示。

```
LOCAL_SRC_FILES := $(call all-java-files-under, src) \
src/com/haiiii/android/helloapk/HelloApk.aidl
```

aidl 文件的路径格式相对于项目的根路径。

编译 aidl 的命令如以下代码所示。

```
out/host/darwin-x86/bin/aidl
-dout/target/common/obj/APPS/HelloApk_intermediates/src/src/com/haiiii/android/helloapk/HelloApk.P
-b
-Ivendor/haiiii/k/apps/HelloApk
-Ivendor/haiiii/k/apps/HelloApk/src -Iframeworks/base/core/java ....
vendor/haiiii/k/apps/HelloApk/src/com/haiiii/android/helloapk/HelloApk.aidl
out/target/common/obj/APPS/HelloApk_intermediates/src/src/com/haiiii/android/helloapk/HelloApk.java
```

aidl 命令的格式是 aidl OPTIONS INPUT [OUTPUT]，其中 OPTIONS 选项中各项参数含义如下。

- -d: 生成依赖文件，关于依赖文件的含义暂不详。
- -b: 如果 aidl 是一个 Parcelable 类，则终止编译。
- -I: 指定该 aidl 文件中 import 对象的搜索路径，该路径一般是 Framework 中的相关路径，因为应用类 APK 中定义的 aidl 文件一般会 import 一些 Framework 中定义的类，此处使用省略号省略了一些其他系统路径。

最后两个参数分别为 aidl 源文件路径和目录路径，目标文件默认就是 aidl 的同名文件，只不过类型是 Java。

### 18.6.4 包含 Java 静态库

有些时候，应用类 APK 需要使用一些静态 Jar 包，因为 Jar 包的开发者并没有提供源码，此时可以把这些 Jar 包复制到项目的根目录下，然后在 Android.mk 中声明。

分析 build/core/java.mk 源码后得知，编译中枢中定义了 LOCAL\_STATIC\_JAVA\_LIBRARIES 这个变量，并且该变量在中枢中以递增方式赋值，如以下代码所示。

```

334 $(LOCAL_BUILT_MODULE): $(a1)_res_assets) $(jni_shared_libraries) $(full_android_manifest)
335     @echo "target Package: $(PRIVATE_MODULE) ($@)"
336     $(create-empty-package)
337     $(add-assets-to-package)
338     $(add-jni-shared-libraries-to-package)
339     $(add-jni-shared-libs-to-package)
340 aapt
341     $(add-full-classes-jar-to-package)
342     $(add-dex-to-package)
343     $(sign-package)
344     @# Alignment must happen after all other zip operations.
345     $(align-package)
346     $(LOCAL_DEX_PREOPT),true)
347     $(hide) rm -f $(patsubst %.apk,%.odex,$@)
348     $(call dexpreopt-one-file,$@,$(patsubst %.apk,%.odex,$@))
349     $(call dexpreopt-remove-classes.dex,$@)
350

```

读者可以对所有以上信息进行综合对比。

## 18.6.2 生成 R.java

Java 源码在编译前必须首先获得 R.java，而 R.java 是编译系统调用 aapt 工具从 res 目录下的资源文件编译而成的，具体命令如下。

```

out/host/darwin-x86/bin/aapt package -z -m
-J out/target/common/obj/APPS/HelloApk_intermediates/src
-M vendor/haiiii/k/apps/HelloApk/AndroidManifest.xml
-P out/target/common/obj/APPS/HelloApk_intermediates/public_resources.xml
-S vendor/haiiii/k/apps/HelloApk/res
-I out/target/common/obj/APPS/framework-res_intermediates/package-export.apk
-G out/target/common/obj/APPS/HelloApk_intermediates/proguard_options
--min-sdk-version AOSP --target-sdk-version AOSP --version-code 9 --version-name AOSP

```

aapt 命令中的参数含义如下。

- -J: 指定 R.java 的输出目录。
- -M: 指定 AndroidManifest.xml 的位置。
- -P: 指定 public\_resources.xml 的输出目录，因为在编译后，同时会输出该文件描述资源的相关属性。
- -S: 指定 res 目录的路径。
- -I: 指定该项目所引用到的其他资源，因为有些应用类 APK 程序会调用到 Framework 中的资源，因此，这里使用的就是 Framework 中的 package-export.apk。
- -G: 指定 proguard 选项的输出文件名称。

读者也可以在 Android 的根目录下手工执行该命令，执行完后，将在 HelloApk\_intermediates/src/com/android/haiiii/HelloApk 目录下生成相应的 R.java 文件。



```

38 Foo ($ (LOCAL_PROTOC_OPTIMIZE_TYPE),micro)
39 LOCAL_STATIC_JAVA_LIBRARIES += libprotobuf-java-2.3.0-micro
40
41 LOCAL_STATIC_JAVA_LIBRARIES += libprotobuf-java-2.3.0-lite
42

```

于是，笔者起初猜测，是不是只需要给这个变量赋值就可以了？如以下代码所示：

```
LOCAL_STATIC_JAVA_LIBRARIES := Foo.jar
```

然而，编译后系统却提示错误，从错误的信息来看，编译中枢把 Foo.jar 看成了一个项目名称，而并没有当成是一个 Jar 包。因此，笔者猜测该变量的含义应该是 HelloApk 所引用的一个静态 Java 库的项目，于是再次尝试给 Android.mk 中添加一个新的 target，并且该 target 的目标仅仅是进行 Jar 包文件的复制。可是，如何告诉编译中枢进行复制呢？此时查看 config.mk 中定义的命令宏中有两个似乎是这个作用，一个是 BUILD\_PREBUILD，另一个是 BUILD\_MULTI\_PREBUILD。因此逐个尝试，最后得出的结论是使用后者，在相应的 Android.mk 文件中增加以下代码。

```

include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_PACKAGE_NAME := AllMyJar
LOCAL_PREBUILT_STATIC_JAVA_LIBRARIES := FooJar:Foo.jar
include $(BUILD_MULTI_PREBUILD)

```

这段代码中主要是给 LOCAL\_PREBUILT\_STATIC\_JAVA\_LIBRARIES 变量赋值，赋值格式类似于中枢中 PRODUCT\_COPY\_FILES 变量，即用冒号进行分隔，冒号之前是该静态库的目标名称，可以任意起名，但应该在整个编译系统中唯一，冒号后面是对应要复制到 Jar 包的名称。

从编译的过程来看，静态库实际上被解压到了 HelloApk 项目中的 src 目录下，如以下代码所示。

```

for f in out/target/common/obj/JAVA_LIBRARIES/FooJar_intermediates/javalib.jar;
do if [ ! -f $f ]; then echo Missing file $f; exit 1; fi;
unzip -qo $f -d out/target/common/obj/APPS/HelloApk_intermediates/classes;
(cd out/target/common/obj/APPS/HelloApk_intermediates/classes && rm -rf META-INF);
done

```

即使用 unzip 解压出 Foo.jar 中内容到 HelloApk\_intermediates/classes 中。

有些读者可能要问了，如果要使用动态库应该怎么办？编译中枢在 java.mk 中定义了一个变量 LOCAL\_JAVA\_LIBRARIES，表面上来看，它对应的就是动态库，如以下代码所示。

```

31 ifneq ($(LOCAL_NO_STANDARD_LIBRARIES),true)
32 LOCAL_JAVA_LIBRARIES := core core-junit ext framework $(LOCAL_JAVA_LIBRARIES)
33 endif

```

所以，似乎只要在 Android.mk 中给该变量赋值为动态库的名称即可，有兴趣的读者可以尝试一下。另外，也可以在 AndroidManifest.xml 中使用 use-library 标签选择需要包含的系统动态库。

### 18.6.5 编译 Java 源文件生成 Jar 包

完成了以上步骤后，所有需要的 Java 源文件或者库文件对应类文件已经就绪，因此可以对 Java 源码进行编译了，编译的命令如下。

```
javac -J-Xmx512M -target 1.5 -Xmaxerrs 9999999 -encoding ascii
-bootclasspath out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes.jar
-classpath out/target/common/obj/JAVA_LIBRARIES/core_intermediates/classes.jar
:out/target/common/obj/JAVA_LIBRARIES/core-junit_intermediates/classes.jar
:out/target/common/obj/JAVA_LIBRARIES/ext_intermediates/classes.jar
:out/target/common/obj/JAVA_LIBRARIES/framework_intermediates/classes.jar
:out/target/common/obj/JAVA_LIBRARIES/JavaLib_intermediates/javilib.jar
-g -extdirs ""
-d out/target/common/obj/APPS/HelloApk_intermediates/classes
\@out/target/common/obj/APPS/HelloApk_intermediates//java-source-list-uniq ||
( rm -rf out/target/common/obj/APPS/HelloApk_intermediates/classes ; exit 41 )
```

该命令的参数意义如下。

- **-bootclasspath**: 指定 Java 运行库 core.jar，这就像 C 程序中的系统库文件一样。
- **-classpath**: 指定静态库，其中包括 core.jar、ext.jar、framework.jar，以及 Foo.jar，多个库文件以冒号分隔。
- **-g**: 标志产生所有的调试所需信息。
- **-d**: 指明 class 的输出路径，此处是 HelloApk\_intermediates/classes 目录。

最后一个参数是要编译的源文件列表，这个列表正是该项目中的所有的 Java 源文件列表，以及 Foo.jar 中解压出的 class 文件。

关于以上命令的使用有个疑惑，既然 java-source-list-uniq 列表中已经包含 Foo.jar 中的 Class 文件，那么为什么 -classpath 中还要再次包含 Foo.jar 所在的路径？

以上命令结束后，HelloApk\_intermediates/classes 目录下就包含了所有的 Class 文件，但这些都是分散的文件，因此需要把这些文件压缩为一个 Jar 包，具体的命令如下。

```
jar -cf out/target/common/obj/APPS/HelloApk_intermediates/classes-full-debug.jar
-C out/target/common/obj/APPS/HelloApk_intermediates/classes
```

该命令中参数的意义如下。

- **-cf**: c 的意义是创建一个新的 Jar 包，f 的意义是指定 Jar 包的文件名称，此处的名称是 classes-full-debug.jar。
- **-C**: 指定要创建的 jar 包所包含的 Class 文件的路径，此处是 Classes 目录，因为前面的 javac 命令已经把全部的 Java 源码一并输出到了该目录中。

创建好 Jar 包后，编译系统又连续调用了四次 **acp** 命令。**acp** 命令和 **cp** 命令作用类似，即进行文件复制 (copy) 的操作，这里 a 的意思是 Advanced，即“高级复制命令”，该命令可以将源文件附加到目

标文件中，如以下代码所示。

```
echo Copying: out/target/common/obj/APPS/HelloApk_intermediates/classes-jarjar.jar
out/host/darwin-x86/bin/acp
    out/target/common/obj/APPS/HelloApk_intermediates/classes-full-debug.jar
    out/target/common/obj/APPS/HelloApk_intermediates/classes-jarjar.jar

mkdir -p out/target/common/obj/APPS/HelloApk_intermediates/emma_out/lib/
out/host/darwin-x86/bin/acp
    -fpt out/target/common/obj/APPS/HelloApk_intermediates/classes-jarjar.jar
    out/target/common/obj/APPS/HelloApk_intermediates/emma_out/lib/classes-jarjar.jar

echo Copying: out/target/common/obj/APPS/HelloApk_intermediates/classes.jar
out/host/darwin-x86/bin/acp
    out/target/common/obj/APPS/HelloApk_intermediates/emma_out/lib/classes-jarjar.jar
out/target/common/obj/APPS/HelloApk_intermediates/classes.jar

echo Copying: out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes.jar
out/host/darwin-x86/bin/acp
    out/target/common/obj/APPS/HelloApk_intermediates/classes.jar
    out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes.jar
```

其中第二次 `acp` 命令中的参数 `-fpt` 的含义是：

`f` 代表强制删除已有的文件，`p` 代表保持文件的原有读写模式，`t` 代表保持文件的原来时间戳。

这四次 `acp` 命令执行的结果是对上面产生的 `classes-full-debug.jar` 进行了四份复制，复制后的结果如下。

```
HelloApk_intermediates/classes-jarjar.jar
HelloApk_intermediates/emma_out/lib/classes-jarjar.jar
HelloApk_intermediates/classes.jar
HelloApk_intermediates/noproguard.classes.jar
```

四份文件的内容完全一样，为什么要进行这四份复制呢？这个没分析明白，因为从后续的编译过程来看，似乎也没有这个必要。

### 18.6.6 将 Jar 包转换为 dex 文件

上一节中编译出了 `noproguard.classes.jar` 文件，接下来需要继续把该 Jar 包转换为 `dex` 格式的文件，因为 Dalvik 虚拟机需要的是 `dex` 格式文件。编译中枢中使用 `dx` 工具用于将 Jar 转换为 `dex` 文件，如下代码所示。

```
out/host/darwin-x86/bin/dx -JXms16M -JXmx1536M
--dex
--output=out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes-with-local.dex
out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes.jar
```

`dx` 工具的参数意义如下。

- `--dex`: 告知 `dx` 产生一个 `dex` 格式的文件。
- `--output`: 指定输出的文件名称, 此处为 `noproguard.classes-with-local.dex`。

最后一个参数为要转换的 Jar 包名称, 这里正是上一节中所产生的 `noproguard.classes.jar` 文件。

转换完后, 本步并没有完成。从实际的过程分析可知, 编译中枢中又使用 `acp` 命令将该 `dex` 文件重新复制了一份, 并命名为 `noproguard.classes.dex`, 如以下代码所示。

```
out/host/darwin-x86/bin/acp
out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes-with-local.dex
out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes.dex
```

### 18.6.7 编译资源文件生成 APK 包

前面小节执行完毕后, 已经产生了 Dalvik 虚拟机所需的 `dex` 文件, 但 Android 中是以 `APK` 的方式进行程序发布的, `APK` 文件本质上只是一个 `zip` 文件, 该文件中包含的内容如下:

- `resources.arsc`: 该文件是一个二进制数据文件, 其中包含了所有资源文件的 `id` 值和路径的映射关系。这就像一个词典, 应用程序执行可以通过资源的 `id` 值找到该资源对应的文件路径。
- `res` 目录: 该目录原封不动地复制了开发时的 `res` 目录, 只是原始的 `XML` 文件以二进制的方式进行保存。
- `classes.dex`: 这就是前面小节执行后产生的 `dex` 文件, 只是名称不同。
- `AndroidManifest.xml`: 这是源文件的二进制格式文件。

从这个结构上可以看出, `classes.dex` 是一个独立的文件, 编译中枢编译 `APK` 的过程也的确如此。它首先使用 `aapt` 工具将资源文件打包为一个 `APK` 文件, 然后再将 `classes.dex` 添加到该 `APK` 包中。

本节介绍的正是该过程中产生 `APK` 的过程, 具体执行的命令如下。

```
out/host/darwin-x86/bin/aapt package -u -z
-c en_US,...,mdpi,nodpi
-M vendor/haiii/k/apps/HelloApk/AndroidManifest.xml
-S vendor/haiii/k/apps/HelloApk/res
-I out/target/common/obj/APPS/framework-res_intermediates/package-export.apk
--min-sdk-version AOSP --target-sdk-version AOSP
--product default --version-code 9 --version-name AOSP
-F out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
```

读者应该还记得, 从资源文件中产生 `R.java` 使用的也是 `aapt` 命令。那么, 该命令如何区分是要产生 `R.java` 还是要产生 `apk` 文件? 这只需要对比一下两者具体使用的差别就可以看出来。

当需要产生 `R.java` 时, 一般需要 `-m` 和 `-J` 参数, 并在 `-J` 后面指定 `R.java` 的输出路径, 一般应该指定到 `src` 目录下, 如以下代码所示。 `-m` 的含义是告诉 `aapt` 使用 `-J` 后面的路径作为 `R.java` 输出路径, 这个语法有点别扭, 因为直接使用 `-J` 似乎更简单, 何必还要再提供一个 `-m` 参数。

```
out/host/darwin-x86/bin/aapt package -z -m
```

```
-J out/target/common/obj/APPS/HelloApk_intermediates/src
```

而当要产生 APK 时,则需要使用-F 参数,并在后面指定 APK 的输出路径。

-u 参数的意思是如果目标 APK 文件存在,则更新包中的内容。

该命令执行完毕后,就在/HelloApk\_intermediates/目录下产生了名为 package.apk 的 APK 文件。此时,该文件中除了没有 dex 文件,其他的就和标准的 APK 文件完全相同。

### 18.6.8 将 dex 文件添加到 APK 包中

由于 APK 包本身就是一个 zip 压缩包,因此,将 dex 文件添加进去的方法很简单,这就跟给一个 zip 压缩包中添加一个新文件的过程是一样的。

编译中枢中执行的添加命令如下。

```
cp out/target/common/obj/APPS/HelloApk_intermediates/noproguard.classes.dex
out/target/common/obj/APPS/HelloApk_intermediates//classes.dex &&
out/host/darwin-x86/bin/aapt add -k
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
out/target/common/obj/APPS/HelloApk_intermediates//classes.dex
&& rm -f out/target/common/obj/APPS/HelloApk_intermediates//classes.dex
```

该命令由三个命令组成,命令之间用&&符号连接。第一个命令是简单的 cp 命令,即把上面转换后的 dex 文件重新命名为 classes.dex。

第二个命令是调用 aapt add 命令。add 只是 aapt 的一个参数,意思是给已有的 APK 中添加新文件,这也就是本节的主要目标。参数-k 注解中说是“Junk path of file(s) added”,实际意义不详。

最后一个命令是删除 classes.dex 文件,因为已经将它添加到 package.apk 中了,所以不再需要了。

### 18.6.9 添加 JNI 所需的动态库文件

有些时候,APK 程序可能需要一些自定义的 JNI 库的支持,比如当 Java 中调用了一些 C 语言实现的函数时,因此,需要把这些 JNI 动态库包含到 APK 包中。

假设需要的动态库文件名称为 Foo.so,则首先将 Foo.so 复制到 HelloApk 目录下,然后在 Android.mk 中声明该库,LOCAL\_JNI\_SHARED\_LIBRARIES := Foo 。

变量 LOCAL\_JNI\_SHARED\_LIBRARIES 是在 package.mk 脚本中被使用的,如以下代码所示:

```
288 jni_shared_libraries := \
289     $(addprefix $(my_prefix)OUT_INTERMEDIATE_LIBRARIES)/, \
290     $(addsuffix $(so_suffix), \
291     $(LOCAL_JNI_SHARED_LIBRARIES)))
```

该脚本给该变量一次添加上前缀和后缀,前缀对应的是一个路径,后缀对应的是 so,这就是为什

么该变量赋值时只需要写上 so 的文件名称即可。在 package.mk 的后面会判断 jni\_shared\_libraries 变量是否为空，如果不为空，则将这些库添加到 APK 中的 lib 目录下，如以下代码所示：

```
338 ifeq ($(jni_shared_libraries),)
339     $(add-jni-shared-libs-to-package)
340 endif
```

其中函数 add-jni-shared-libs-to-package 是在 definitions.mk 中定义的，如以下代码所示：

```
1420 define add-jni-shared-libs-to-package
1421 $(hide) rm -rf $(dir $@)lib
1422 $(hide) mkdir -p $(dir $@)lib/$(TARGET_CPU_ABI)
1423 $(hide) cp $(PRIVATE_JNI_SHARED_LIBRARIES) $(dir $@)lib/$(TARGET_CPU_ABI)
1424 $(hide) (cd $(dir $@) && zip -r $(notdir $@) lib)
1425 $(hide) rm -rf $(dir $@)lib
1426 endef
```

该函数实际执行时展开的命令如下：

```
rm -rf out/target/product/generic/obj/APPS/HelloApk_intermediates/lib
mkdir -p out/target/product/generic/obj/APPS/HelloApk_intermediates/lib/armeabi
cp out/target/product/generic/obj/lib/Foo.so
    out/target/product/generic/obj/APPS/HelloApk_intermediates/lib/armeabi
(cd out/target/product/generic/obj/APPS/HelloApk_intermediates/
    && zip -r package.apk lib)
rm -rf out/target/product/generic/obj/APPS/HelloApk_intermediates/lib
```

即先删除 lib 目录，然后新建一个 lib/armeabi 目录，其中 armeabi 来自于变量 TARGET\_CPU\_ABI，该变量的值正是 make 一个 product 时信息输出中的 TARGET\_ARCH\_VARIANT 变量的值。

接着将 Foo.so 复制到 lib/armeabi 目录中，然后再使用 zip 命令将该 lib 目录压缩到 package.apk 中，最后删除 lib 目录。该命令中需注意，此时 Foo.so 的路径是在 out 目录的 lib 目录下，而并不是 HelloApk\_intermediates 目录，那么编译中枢是如何将 Foo.so 复制到 out 中的 lib 目录中的呢？这就是 Android.mk 中需要做的额外工作，大家如何直接使用前面提到的 Android.mk 编译 HelloApk 项目。系统会提示找不到 Foo.jar 文件，所以在编译之前需要首先将 Foo.so 复制到 out 的 lib 目录下，这个与前面提到的复制静态 Java 库文件的过程类似。因此，可以给 Android.mk 文件最后添加以下代码：

```
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := AllMySo
LOCAL_MODULE_CLASS := SHARED_LIBRARIES
LOCAL_PREBUILT_LIBS := Foo:Foo.so
include $(BUILD_MULTI_PREBUILT)
```

该代码中主要使用了 LOCAL\_MODULE\_CLASS 声明目标为共享库，然后在 LOCAL\_PREBUILT\_LIBS 中定义了要复制的对象，最后调用 BUILD\_MULTI\_PREBUILT 进行编译。以上变量是在 multi\_prebuilt.mk 脚本中定义的，有兴趣的读者可进一步查看其转换逻辑。

以上介绍的是以 so 的方式添加 JNI 动态库，如果想以源码的方式添加又如何呢？这个实际上更简

单，可以先在 HelloApk 中新建一个文件夹，名称任意，然后要把 C 源码放到该目录下，并在该目录下新建一个 Android.mk 文件即可，内容就和编译不同的动态库一样，最后在 HelloApk 目录下的 Android.mk 文件调用 include 包含 JNI 目录下的 Android.mk 文件即可。当然，因为是以源码的方式编译 JNI，因此 HelloApk 目录下的 Android.mk 中，就不需要再使用 BUILD\_MULTI\_PREBUILT 预编译所需的动态库了。有兴趣的读者可以参照 Android 源码下 packages/inputmethods/PinyinIME 项目，该项目使用的正是源码方式的 JNI 编译。

### 18.6.10 对 APK 文件进行签名

需要澄清一下，签名的目的不是加密，而是认证。所以，第一，签名后的程序依然可以提取出程序文件；第二，apk 可以被多次签名，后来的签名会覆盖之前的签名。

签名操作的执行命令如下：

```
mv out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.unsigned

java -jar out/host/darwin-x86/framework/signapk.jar
build/target/product/security/platform.x509.pem
build/target/product/security/platform.pk8
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.unsigned
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.signed
```

以上代码首先把前面产生的 package.apk 重命名为.unsigned，然后执行 signapk.jar 程序。由于该程序是一个 Java 程序，所以使用 Java 命令执行该程序。后续参数指定了.pem 文件和 pk8 文件，这两个文件是认证所需的文件，源码中目前提供了四对认证文件，分别为 platform、shared、user、tests，关于 pk8 和 pem 文件的更多信息，请参考 build/target/product/security/README 自述文件，该文件中介绍了如何从 pk8 产生 pem 文件。

### 18.6.11 使用 zipalign 优化 APK 内部存储

所谓的“内部存储优化”是指，为了提高 APK 程序的加载速度，从而对 APK 中相关的数据进行边界对齐。因为从底层 NAND Flash 的角度来讲，读取 NAND 时，是以一个扇区进行读取的，因此，如果相关的数据能够在同一个扇区中，肯定会提高读取速度。zipalign 的作用正是将 APK 包中的不同类型的数据文件进行边界对齐。

zipalign 的用法如下：

```
mv out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.unaligned
out/host/darwin-x86/bin/zipalign
```



```
-f 4
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.unaligned
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.aligned

mv out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk.aligned
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
```

该段代码首先将 `package.apk` 重命名为 `.unaligned`，然后调用 `zipalign` 命令。参数的意义如下。

- `-f` 意思是如果输出文件存在，则强制覆盖。
- 数字 4 的意思是指定对齐位，4 代表 32 位对齐。
- 后面两个参数分别是输入文件和输出文件。

最后，将 `package.apk.aligned` 重命名为 `package.apk`。至此，这个 APK 文件就是最终产品中的 APK 文件了，编译中枢最后执行的命令就仅仅是把这个 `package.apk` 复制到 `system/app` 目录下，并重命名为 `HelloApk.apk`，如以下代码所示：

```
out/host/darwin-x86/bin/acp -fpt
out/target/product/generic/obj/APPS/HelloApk_intermediates/package.apk
out/target/product/generic/system/app/HelloApk.apk
```

## 18.7 Framework 的编译

这里所讲的“Framework 的编译”是指根目录下的 Framework 子目录的编译，Framework 编译后将输出重要的 Jar 包文件，这些文件包括：

- `framework.jar`，该文件包含了 Framework 中包含的 Java 文件编译后的 Class 文件，当然，这些 Class 文件已经被转换为了 dex 格式。
- `core.jar`，包含了 `dalvik` 虚拟机运行时所需的 Java 运行库文件，比如 `String` 类。
- `ext.jar`，包含了一些扩展的类库文件。
- `framework-res.apk`，包含了 Framework 中所使用到的各种资源文件。

本节就来介绍以上目标的编译过程。

### 18.7.1 总体编译过程

本质上讲，Framework 也是一个 APK，其编译过程和 APK 的编译过程比较类似，只是在编译的工程中同时生成了一些特别的 Jar 包。读者可以在 `android` 根目录下执行 `make framework -n > ~/Desktop/framework.txt` 命令，将 Framework 的中间编译过程输出到桌面的 `framework.txt` 文件中，从而可以具体查看编译的细节。

Framework 的总体编译过程如图 18-8 所示。

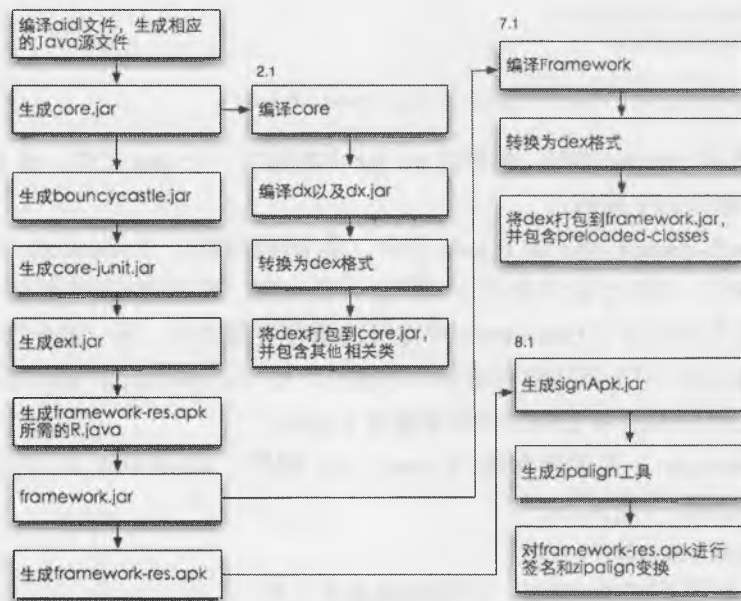


图 18-8 framework.jar 的总体编译过程

① 编译 aidl 文件。这些 aidl 文件是在 framework/base 目录下的 Android.mk 文件中定义的，该文件中定义了两个变量，一个是 LOCAL\_SRC\_FILES，另一个是 aidl\_files，其中前者包含了 Framework 中所需要使用的所有 aidl 文件，而后者仅仅是那些将被放到 SDK 中的 aidl 文件。

② 生成 core.jar。这里所谓的“core”是指 Dalvik 虚拟机运行时所需的 Java 库，并不是 framework/core 目录下文件，这点从命名上很容易让人产生混淆。

(1) core 目标是在 Android 根目录下的 libcore 目录中的 JavaLibrary.mk 文件中定义的，编译 core 后会生成相应的 Jar 文件。

(2) 如果 dx 工具还没有被编译，则需要编译 dx 工具和 dx.jar 工具，这两个工具用于将 Jar 文件转换为 dex 文件。

(3) 将 core 目标对应的 Jar 文件转换为 dex 文件。

(4) 将 dex 打包到 core.jar 中，同时给 core.jar 包含一些其他文件。

从以上过程来看，尽管 core.jar 表面上是一个 Jar 文件，但内部的主要内容却是 core 目标编译后的 dex 文件。换句话说，系统运行或者加载 core.jar 时，本质上加载的是其内部的 dex 文件。

③ 生成 bouncycastle.jar。目前暂不清楚该 Jar 包的作用，不过该 Jar 的编译过程和 core.jar 类似，都是先编译出 Jar 包，然后再进行 dex 转换，最后再添加一个额外的信息，从而生成最终的 Jar 包。

④ 生成 core-junit.jar。该库用于 core 目标的单元测试。

⑤ ext.jar。ext 目标对应的源码路径是在 framework/base/Android.mk 中定义的，如以下代码所示：

```
ext_dirs := \
    ../../external/nist-sip/java \
```

```

../../external/apache-http/src \
../../external/tagsoup/src

ext_src_files := $(call all-java-files-under,$(ext_dirs))

```

ext.jar 的编译过程和 core.jar 类似，最终的 ext.jar 中将包含一个 dex 文件，该 dex 文件是在 ext\_dirs 目录下所有 Java 源码编译后生成的。

⑥ 生成 framework-res.apk 所需的 R.java 文件。本节前面指出，Framework 本质上也是一个 APK 程序，只是这个 APK 内容比较多而已，并且，该 APK 最终被分成两部分输出。一部分是 framework-res.apk，该文件包含了 Framework 中定义的所有资源文件；另一部分是一个 dex 文件，这个 dex 文件包含了 Framework 中所有的源码编译后的输出。而 framework.jar 编译时需要资源文件对应的 R.java，因此需要首先使用 aapt 将资源文件编译输出 R.java。

⑦ 编译 framework.jar。该过程和编译 core、ext 相同，framework.jar 包含的 Java 源码是在 framework/base/Android.mk 中定义的。

⑧ 生成 framework-res.apk。

(1) 如果 signApk 工具还没有编译，则需要编译该工具。

(2) 如果 zipalign 工具还没有编译，则需要编译该工具。

(3) 使用 aapt 工具编译出 framework-res.apk，然后再使用 signApk 对其进行签名，签名时使用的签名文件为 platform，最后再使用 zipalign 对其进行压缩存储对齐。

## 18.7.2 framework/core/ext 三个 Jar 文件的区别

Framework 在编译时会产生三个 jar 包文件，分别为 Framework、core、ext。有些读者对这三个 Jar 包的作用不甚了解，因此，本节从功能及所包含的内容上对其进行说明。

首先，从 Linux 的角度来看，Dalvik 虚拟机仅仅是一个普通的 Linux 应用程序而已，这就是为什么 Dalvik 虚拟机也是用 C/C++ 语言编写的原因。而这个应用程序的功能就是能够读取 dex 文件，并将 dex 文件中的内容转换为程序代码并执行这些程序，这些 dex 文件是用 Java 语言编写的，并在使用 Javac 编译器编译后产生。在编译 Java 源码时，基本上都需要使用到一些内部类，比如 String 类、Runtime 类等，对 Java 应用程序而言，这些类不需要自己实现，而是调用系统内部类即可。这些类本身也是由 Java 语言编译，只不过可能会调用一些 JNI 实现相关的底层函数，core.jar 文件正是这些系统内部 Java 类的实现。

Framework 只是另一个 Dalvik 虚拟机的类库，该类库包含的就是 Android 中定义的功能类，这就是为什么它的名字叫做“框架”的原因。因为它定义的是 Android 框架的内涵，从纯 Java 程序的角度来看，读者完全可以不使用 framework.jar 库，而仅仅使用 core.jar 库，这样编译出来的 dex 程序也同样可以在 Dalvik 虚拟机上执行，只是这种程序无法访问 Framework 中定义的功能，比如窗口管理。

ext 只是一个扩展类库，里面的内容也很少，一般可以把一些第三方的库放到该库中。

这三个 Jar 包对应的源码都是在 framework/base/Android.mk 文件中定义的，详情参照该文件。

## 18.8 编译 android.jar

android.jar 是 framework.jar 的一个子集，android.jar 文件仅作为 SDK 的一部分，其路径是：/android-sdk-mac\_86/platforms/android-7/android.jar，其中 android-7 代表的是 SDK 的版本号。

开发 Android 应用程序时，一般需要指定该应用程序所使用的版本号，Eclipse 就会自动包含该版本对应的 android.jar 文件，该文件就是 Eclipse 中大家常看到的 android 包，如图 18-9 所示。

那么，既然 android.jar 是 Framework 的一个子集，那么为什么不直接使用 framework.jar 呢？另外，生成这个子集的规则是什么，即都把哪些内容放到了 android.jar 中，哪些没有放进去？

如图 18-10 所示，Framework 中的源文件包含三个部分，分别为 aidl 源文件、res 目录下的资源文件，以及 Java 文件。而这些源文件中都包含一些所谓的“公有”部分和私有部分，“公有”的作用是为了保证应用程序的兼容性，而“私有”部分是 Framework 内部使用的一些文件，这些文件由于是私有的，因此，在不同的版本上会发生较大变化。

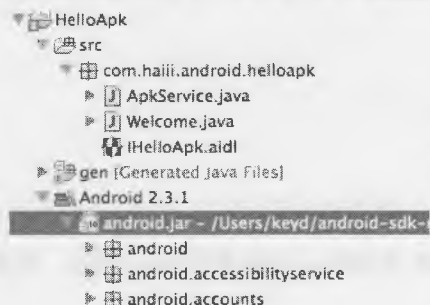


图 18-9 Eclipse 环境下包含的 android.jar

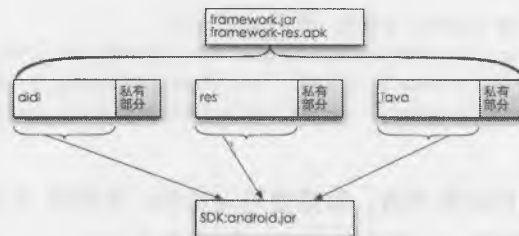


图 18-10 Framework 源程序组成

“公有”部分的兼容性是后向兼容，即较早的应用程序可以在最新的 SDK 中运行。公有部分的内容会被提取出来放到 android.jar 中，应用程序基于 SDK 开发的程序可以使用“公有”部分中的任何类或者资源。较新版本中的 android.jar 会包含较老版本的全部“公有”部分，从而保证兼容性。

后面小节将分别针对这三类源码中的“公有”部分和“私有”部分进行讨论。

### 18.8.1 资源文件

资源文件是使用 aapt 工具进行编译的，aapt 会为每一个资源定义一个 id 值，这个 id 值默认是由 aapt 工具内部自动产生的。不过应用程序可以事先给资源指定 id 值，这是通过在 res/values 目录下新建一个 public.xml 文件实现的。

为了具体说明 public.xml 的作用，本节使用一个具体的例子，基于前面的 HelloApk 项目，并假设

res/values 目录下有一个 attrs.xml 文件，内容如下：

```
<resources>
  <attr name="checkboxStyle_2" format="reference" />
  <attr name="checkboxStyle_3" format="reference" />
  <attr name="checkboxStyle_4" format="reference" />
</resources>
```

首先，在 Android 根目录下运行以下 aapt 命令：

```
aapt package -x -z -m -J out/target/common/obj/APPS/HelloApk_intermediates/src
-M vendor/haiii/k/apps/HelloApk/AndroidManifest.xml
-P out/target/common/obj/APPS/HelloApk_intermediates/public_resources.xml
-S vendor/haiii/k/apps/HelloApk/res
-I out/target/common/obj/APPS/framework-res_intermediates/package-export.apk
-G out/target/common/obj/APPS/HelloApk_intermediates/proguard_options
--min-sdk-version AOSP --target-sdk-version AOSP --version-code 9 --version-name AOSP
```

该命令中要特别说明的是 -x 参数，其意义是告诉 aapt 产生非应用（non-application）的 id 值。执行完命令后，可以到 out/target/common/obj/APPS/HelloApk\_intermediates 目录中查看输出的 R.java 的值，该文件中关于 attrs 资源的 id 值输出如下：

```
public static final class attr {
    public static final int checkboxStyle_2=0x02010000;
    public static final int checkboxStyle_3=0x02010001;
    public static final int checkboxStyle_4=0x02010002;
```

从输出的结果来看，如果使用 -x 参数，资源的 id 值是以 0x02 开始的，而如果没有 -x 参数，则资源值以 0x07 开始。0x02010000 这个值的意义如下。

- 02：代表这是普通的非应用资源值。系统中还有两种类型，一种是 01，另一种是 07，其中 01 的意思是 Framework 资源，07 代表普通应用类资源。
- 01：代表资源的类型，01 的意思是 attr 资源，02 代表 drawable，03 代表 layout 资源，04 代表 string 资源。
- 0000：后面的四个数字代表 id 的编号，从 0 开始，依次增加。

读者此时还可以查看 HelloApk\_intermediates 目录下的输出文件 public\_resources.xml，该文件则使用 XML 格式定义了资源，如以下代码所示：

```
<resources>

  <!-- PRIVATE SECTION. These resources have not been declared public.
    You can make them public by moving these lines into a file in res/values. -->

  <!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/attrs.xml:2 -->
  <public type="attr" name="checkboxStyle_2" id="0x02010000" />
  <!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/attrs.xml:3 -->
  <public type="attr" name="checkboxStyle_3" id="0x02010001" />
```

```

<!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/attrs.xml:4 -->
<public type="attr" name="checkboxStyle_4" id="0x02010002" />

<!-- Declared at res/drawable-ldpi/icon.png:0 -->
<!-- Declared at res/drawable-mdpi/icon.png:0 -->
<!-- Declared at res/drawable-hdpi/icon.png:0 -->
<public type="drawable" name="icon" id="0x02020000" />

<!-- Declared at res/layout/main.xml:0 -->
<public type="layout" name="main" id="0x02030000" />

<!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/strings.xml:3 -->
<public type="string" name="hello" id="0x02040000" />
<!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/strings.xml:4 -->
<public type="string" name="app_name" id="0x02040001" />

</resources>

```

从这段代码中可以看出，aapt 内部似乎隐藏着某种玄机。PRIVATE SECTION 的意思是什么呢？这正是“私有”资源的痕迹。在默认情况下，HelloApk 中的资源都将被编译为私有资源，意思是说，该资源包中的 R.java 将不包含任何资源的定义。可是大家刚才明明看到 R.java 中包含了所有这些私有资源，这是怎么回事呢？

这是因为我们并没有定义私有资源的输出路径，因此在默认情况下私有资源对应 R.java 就输出到了 HelloApk\_intermediates/src/com/haiii/android/helloapk/目录下。

读者可以尝试在 HelloApk/res/values 目录下增加一个 public.xml 文件，内容如下：

```

<resources>
  <public type="attr" name="checkboxStyle_2" id="0x02010000" />
  <public type="attr" name="checkboxStyle_3" id="0x02010002" />
</resources>

```

该文件中将 attrs.xml 中的两个 attr 设置为公开类型，并且分配了默认的 id 值，然后再执行和前面相同的编译命令。执行完毕后查看 public\_resources.xml 文件的输出及 R.java 的输出，此时可以发现，在 public\_resources.xml 中多了一个 PUBLIC SECTION 的说明，结果如下：

```

<resources>

  <!-- PUBLIC SECTION. These resources have been declared public.
    Changes to these definitions will break binary compatibility. -->

  <public type="attr" name="checkboxStyle_2" id="0x02010000" />
  <public type="attr" name="checkboxStyle_3" id="0x02010002" />

  <!-- PRIVATE SECTION. These resources have not been declared public.
    You can make them public by moving these lines into a file in res/values. -->
  <!-- Declared at vendor/haiii/k/apps/HelloApk/res/values/attrs.xml:4 -->
  <public type="attr" name="checkboxStyle_4" id="0x02010001" />

```

此时，public\_resources.xml 中的内容有以下两个特点：

- public.xml 中定义的两个 attr 资源被放到了 PUBLIC SECTION 说明之后，并且这两个资源的 id 值正是 public.xml 中定义的 id 值。
- PRIVATE SECTION 说明后包含了所有其他项目中资源的定义。

下面再看输出的 R.java 文件，和没有 public.xml 文件的区别仅仅在于 public.xml 中定义的两个 attr 的 id 值是预先设置好的，除此之外，没有其他差别。

到目前为止，我们似乎还没有看到“公开”和“私有”的差别在哪里，不过已经看到了 PRIVATE 和 PUBLIC 两个 SECTION 的说明。而我们期望的是，私有的资源和公开的资源能够分别对应两个 R.java，那样才是有意义的。

对照 framework/base/core/res/res/values/public.xml 后发现，该文件中使用了一个特别的参数 private-symbols，事实上，正是这个参数指定了私有资源的输出路径。由于前面没有为该变量赋值，导致在默认情况下私有资源和公开资源被输出到了同一个 R.java 中，因此，给 HelloApk/res/values/public.xml 文件也增加该参数的赋值，如下代码所示。

```
<resources>
  <private-symbols package="com.haiii.internal" />
  <public type="attr" name="checkboxStyle_2" id="0x02010000" />
  <public type="attr" name="checkboxStyle_3" id="0x02010002" />
</resources>
```

然后重新执行以上的 aapt 编译命令，执行完毕后再次查看 public\_resources.xml 的值和 R.java 的值。此时发现 public\_resources.xml 的值没有变化，而 src 目录下多了 com/haiii/internal 目录，这个目录正是 private-symbols 参数的值。总的 src 目录下现在有了两个 R.java，分别是：

```
localhost:HelloApk_intermediates keyd$ pwd
/Users/keyd/android22/out/target/common/obj/APPS/HelloApk_intermediates
localhost:HelloApk_intermediates keyd$ find . -name R.java
./src/com/haiii/android/helloapk/R.java
./src/com/haiii/internal/R.java
```

而这两个 R.java 的内容分别如下：

```
package com.haiii.android.helloapk;
public final class R {
    public static final class attr {
        public static final int checkboxStyle_2=0x02010000;
        public static final int checkboxStyle_3=0x02010002;
    }
    public static final class drawable {
    }
    public static final class layout {
    }
    public static final class string {
    }
}
```



```

package com.haiiii.internal;
public final class R {
    public static final class attr {
        public static final int checkboxStyle_2=0x02010000;
        public static final int checkboxStyle_3=0x02010002;
        public static final int checkboxStyle_4=0x02010001;
    }
    public static final class drawable {
        public static final int icon=0x02020000;
    }
    public static final class layout {
        public static final int main=0x02030000;
    }
    public static final class string {
        public static final int app_name=0x02040001;
        public static final int hello=0x02040000;
    }
}

```

从 R.java 的输出信息可以看出,公开的 R.java 中仅包含了 public 类型资源的定义,而私有的 R.java 中包含了所有资源的定义。Framework 生成 android.jar 时正是把 frameworks/res/res/values/public.xml 中定义 public 类型的资源放到了 android.jar 中,因此,如果想给 Framework 增加新资源,并且希望该新资源能够被放到 SDK 中,那么就在 frameworks 路径下的 public.xml 中声明该资源。

### 18.8.2 aidl 文件

Framework 中包含的 aidl 是在 frameworks/base/Android.mk 中定义的,该文件定义了两处 aidl 文件列表。

第一处是给 LOCAL\_SRC\_FILES 变量使用 += 赋值符号进行赋值,该变量将包含 framework.jar 目标中所有的源文件,包括 aidl 文件和 Java 文件。

第二处是给 aidl\_files 变量使用 := 赋值符号进行赋值,该变量仅仅包含 android.jar 目标中所有的 aidl 文件。

因此,当给 Framework 中添加新的 aidl 文件时,需要考虑该文件是否要公开到 SDK 中。如果需要,则必须把该文件路径同时添加到以上两个变量中;如果不需要公开到 SDK 中,则只需把该文件路径添加到 LOCAL\_SRC\_FILES 变量中。

### 18.8.3 Java 文件

Framework 中包含的 Java 文件列表是在 frameworks/base/Android.mk 中定义的,如以下代码所示:

```
LOCAL_SRC_FILES := $(call find-other-java-files,$(FRAMEWORKS_BASE_SUBDIRS))
```

这段脚本使用 `find-other-java-files` 脚本函数，从 `FRAMEWORKS_BASE_SUBDIRS` 路径下寻找所有的 Java 源文件，而这个路径是在 `build/core/pathmap.mk` 中定义的，如以下代码所示。

```

79 FRAMEWORKS_BASE_SUBDIRS := \
80   $(addsuffix /java, \
81     core \
82     graphics \
83     location \
84     media \
85     drm \
86     opengl \
87     sax \
88     telephony \
89     wifi \
90     vpn \
91     keystore \
92     voip \
93   )

```

因此，如果要给 Framework 中添加新的 Java 文件，要考虑新的文件是否在以上路径中。如果在，那么仅仅添加新的 Java 文件即可，而不需要做额外的修改。而如果新的 Java 文件路径不在以上路径中，则需要修改这段脚本，使其包含新的 Java 文件所在路径。

以上讲述的仅仅是给目标 `framework.jar` 中添加 Java 文件的方式，而如果同时希望新建的 Java 类能够公开到 SDK 的 `android.jar` 包中，又当如何呢？

要回答这个问题就需要了解 `android.jar` 包中包含的 Java 类的来源。通过实验，使用 `make sdk` 命令获取中间的编译过程的输出，可知，`android.jar` 中 Java 类是通过 `javadoc` 工具获得的，如以下代码所示。

```

LD_LIBRARY_PATH=out/host/darwin-x86/lib \
javadoc \

\out/target/product/generic/obj/JAVA_LIBRARIES/api-stubs_intermediates/droiddoc-src-list \
-J-Xmx768m \
-J-Djava.library.path=out/host/darwin-x86/lib \
\
-quiet \
-doclet DroidDoc \
-docletpath

out/host/darwin-x86/framework/clearsilver.jar:out/host/darwin-x86/framework/droiddoc.jar
:out/host/darwin-x86/framework/apicheck.jar \
-templatedir build/tools/droiddoc/templates-sdk \
-templatedir build/tools/droiddoc/templates \
-htmldir frameworks/base/docs/html -htmldir out/target/common/docs/gen \
\
-classpath

out/target/common/obj/JAVA_LIBRARIES/bouncycastle_intermediates/classes.jar...
-sourcepath frameworks/base/core/java: ...
-d out/target/common/docs/api-stubs \
-hdf page.build OPENMASTER-eng.keyd.20110318.100834 -hdf page.now "18 Mar 2011 10:08" \

```

```

-knowntags ./frameworks/base/docs/knownTags.txt
-since ./frameworks/base/api/1.xml 1
-since ./frameworks/base/api/2.xml 2
-since ./frameworks/base/api/3.xml 3
-since ./frameworks/base/api/4.xml 4
-since ./frameworks/base/api/5.xml 5
-since ./frameworks/base/api/6.xml 6
-since ./frameworks/base/api/7.xml 7
-since ./frameworks/base/api/8.xml 8
-since ./frameworks/base/api/9.xml 9
-werror -hide 13
-overview frameworks/base/core/java/overview.html
-hdf sdk.version 2.3
-hdf sdk.rel.id 1
-stubs out/target/common/obj/JAVA_LIBRARIES/android_stubs_current_intermediates/src
-apixml out/target/common/obj/PACKAGING/public_api.xml
-nodocs
.....

```

这段代码的关键包含两点：

- 参数 `since` 的意义是包含所有之前 SDK 版本中的 API 定义文件，作用是为了检查各个版本 API 的兼容性。
- 参数 `-apixml` 后面紧跟当前 Framework 版本对应的 API 的输出路径，输出文件名称为 `public_api.xml`，该文件的内容对应了 `android.jar` 中公开的 Java 类，其产生的过程是由 `javadoc` 工具完成。该工具中将所有非 `@hide` 类型的类、方法、变量进行导出，生成 `public_api.xml`，换句话说，“公开”的规则就是在 Java 源码中不要使用 `@hide` 修饰符。

产生 `public_api.xml` 后，编译系统同时会将这个文件复制到 `frameworks/base/api` 目录下，并重名为 `current.xml`，如下代码所示。

```

out/host/darwin-x86/bin/acp
out/target/common/obj/PACKAGING/public_api.xml
frameworks/base/api/current.xml

```

由于 `public_api.xml` 仅会被自动生成一次，因此，当修改 Framework 源码后，该文件不会自动重新编译，这就是为什么有些读者添加了新的 API 后却不能在新的 SDK 中看到这种变化的原因。重新编译 `public_api.xml` 的方法是使用 `make update-api` 命令，该编译命令会重新调用 `javadoc` 工具扫描所有 Framework 中的 Java 源码，并产生新的 `public_api.xml`。

以上就是 `android.jar` 中 Java 文件添加的规则。

## 18.9 编译 adt 插件

Android Development Toolkit (ADT) 是一个开发 Android 应用程序的工具组，该工具组可以单独使用，也可以被集成到 Eclipse 中，读者可以从 `android.com` 上下载不同的 adt 版本。

除了使用官方的 `adt`，也可以编译自己的 `adt` 插件，编译的方法是在 `Android` 根目录下的三个文件中共同说明，分别如下：

- `sdk/eclipse/README_WINDOWS.txt`。
- `sdk/eclipse/scripts/build_plugins.sh`。
- `sdk/eclipse/scripts/build_server.sh`。

所不幸的是，截至目前为止，只能在 `Linux` 主机系统下编译 `ADT` 插件，而不能直接在 `Mac` 上编译。因此，本节以笔者在 `Linux` 的编译过程进行说明。有些读者可能会疑惑，在 `Linux` 上编译的 `adt` 是不是仅仅适用于 `Linux` 下的 `Eclipse` 而不能适用于 `Mac` 或者 `Windows` 上的 `Eclipse` 呢？当然不是，因为 `Eclipse` 本身是一个 `Java` 程序，`ADT` 插件是以一个 `Jar` 的方式存在的，所以，所有的操作系统都是相同的，这也就是 `Java` 程序的一个优势。

编译 `adt` 的方法很简单，如以下命令所示。

```
$ cd sdk/eclipse/scripts
$ ECLIPSE_PATH="$HOME/eclipse" ./build_server.sh
```

第一句首先进入编译目录，第二句中调用 `build_server.sh` 脚本，调用之前同时设置参数 `ECLIPSE_PATH` 环境变量，使其指向本机中 `eclipse` 的安装路径。笔者的 `eclipse` 在 `HOME` 路径下，因此直接指向此处。

编译完毕后，会在 `eclipse` 安装路径下的 `plugins` 目录下生成相应的 `zip` 文件。

## 18.10 总结

`Android` 编译系统是一个有着丰厚积累的仓库，其中包含了 `make` 脚本、`shell` 脚本、`python` 脚本等众多脚本文件，要彻底了解每一行代码的作用和意义是一件稍烦琐，也没有必要的事情，读者在了解了其大致框架后，再根据实际项目的需求，修改局部脚本代码即可。

另外，`Android` 的源码中包含了很多额外的工具，一般会将其放到 `tools` 目录下。读者可以使用 `find -name "tools"` 查看 `Android` 系统中的所有工具，这些工具的功能十分广泛，欢迎有兴趣的读者去分析一些本书所没有涉及的工具的使用，并与同伴们分享。



## 第 19 章 编译自己的 Rom

编译自己的 ROM 也就是大家熟知的“刷机”，刷机所需的 ROM 文件主要包含两部分，一部分是 Linux 内核 ROM 文件，另一部分是 Android 所需的 ROM 文件。读者可以更改这两部分 ROM 的源码文件，然后生成相应的 ROM 文件，并将 ROM 文件写入手机的存储器上，从而可以检验自己更改源码后的效果。本章就来介绍如何完成这一过程。

### 19.1 嵌入式系统的内存地址空间

要了解各种 ROM 文件的基本工作原理，需要首先了解嵌入式系统的内部地址空间。

所有的 CPU 都包含一组地址总线 and 数据总线，地址总线用于输出 CPU 想要访问的地址信息，数据总线用于传输 CPU 需要读或者写的数据。与 CPU 连接的器件一般包括外部存储器、输入/输出 (I/O) 设备等，从 CPU 的观点上看，外部存储器和 I/O 设备没有什么区分，CPU 总是从指定的地址进行数据交换。而从设计师的角度来看，会为存储器及 I/O 设备分配不同的地址，从而完成所希望的任务。

当系统上电时，CPU 会从固定的地址开始执行程序，这是由 CPU 内部的硬件逻辑保证的。在一般情况下，这个固定的地址会在 CPU 出厂时内置一小段代码，代码长度一般在 1KB 左右，其作用主要是从其他地方读取用户编写的代码，也就是大家常说的引导程序 (Boot Loader)，这个过程就称之为“引导过程”，因为该过程仅仅是把用户的程序装载到相应的内存地址，而用户的程序还没有真正开始执行。所谓的“其他地方”一般包括 USB 端口、SPI 穿行端口、NAND Flash 接口等，不同的 CPU 会根据自身的硬件结构配置不同的引导方式。一般的 CPU 会包含几个引脚，硬件工程师可以配置这几个引脚的高、低电平状态，从而使得 CPU 在上电后会根据这几个引脚的状态选择不同的引导方式。

Android 设备的地址空间如图 19-1 所示。

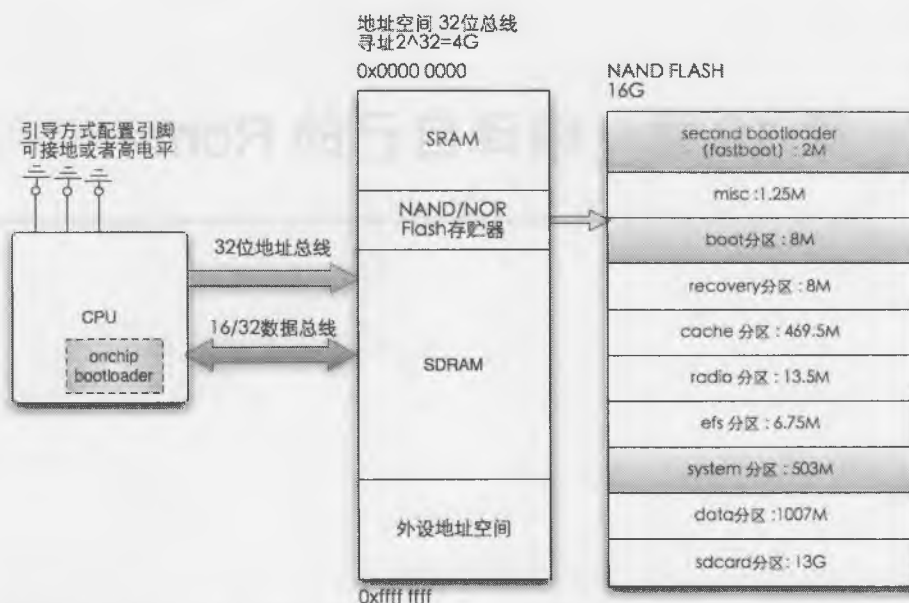


图 19-1 Android 设备的系统内存示意图

CPU 采用 ARM 内核，地址总线为 32 位，可寻址的空间为 4GB，也就是说，该硬件系统能支持的最大系统内存为 4GB。这 4GB 空间中一般包含以下内容。

- **SRAM:** 异步存储器。这种存储器就是早期嵌入式系统中所使用的存储器，其特点是每一个地址都对应一个数据，CPU 可以直接访问这些地址。比如 C 代码中 `*(addr) = 0x5555`，当 `addr` 的值为 `0x001f 0000` 时，`0x5555` 这个值就会被写入到这个地址中。这种存储器一般容量有限，比如一般使用的有 512KB、1MB、2MB 等，再大的话成本就比较高。
- **NAND Flash 存储器:** 这就是大家所熟知的 U 盘。NAND Flash 采用了一种独特的技术，内部可以存储超大容量的数据，比如现在的 U 盘动辄 2GB、4GB 等，Nexus S 上使用的 NAND Flash 容量是 16GB。不熟悉硬件的读者可能觉得疑惑，不是说系统的内存空间最多为 4GB 吗，16GB 的空间如何寻址呢？事实上，NAND Flash 本身所占用的系统地址很少很少，一般可能只占用 4 个地址。CPU 访问 NAND 时，会经过一系列时钟周期，在这个周期中会首先通过数据总线告诉 NAND 需要访问的地址，对于 16GB 的 NAND 而言，会占用两个时钟周期传递地址，第一个传递高 32 位，第二个传递低 32 位。打个比方，假设 CPU 是邮递员，地址总线是信封上写的地址，数据总线传递的是信封的内容，系统中包含 4GB 个邮递地址，而 NAND 仅占用 4 个地址。当用户想访问 NAND 中的某个地址 `0x1234` 时，首先在信封上写下 NAND 的 4 个地址，邮递员只负责把信件传递给 NAND，而 NAND 内部会打开信件，并从信件中读取需要访问的具体数据的地址 `0x1234`，然后再把读出的数据传递给 CPU。NAND Flash 是近 10 年来才出现的技术，而在早

期的硬件中是不存在的，因此在当时这种存储器件被称之为 Memory Technology Device，即“使用了存储技术的设备”，也就是后来 Linux 中所统称的 MTD。

- **SDRAM**: 同步动态 RAM，其特点是访问速度快，内存容量相比 SRAM 会大一些，并且也是一个地址对应一个数据，CPU 可以直接访问。对程序员来讲，它和 SRAM 没有什么区别。
- **外设地址空间**: 硬件系统中不能只有 CPU 和内存，因为大多数硬件系统都需要和用户进行交互，比如按键、显示器、USB 接口等，CPU 访问这些硬件设备也是通过地址总线 and 数据总线。对 CPU 来讲，它并不区分具体地址上对应的是 SDRAM 还是接口设备，CPU 只是执行程序，包括从指定的地址中读取或者写入数据。在嵌入式设备中，一般采用的是统一总线结构，即所有的外部设备共用一个地址空间。这与 PC 的地址结构有所不同，对于 PC 而言，CPU 的速度实在是太快了，考虑到访问内存和普通外部设备的速度的巨大差异，因此提出了“南桥”和“北桥”的概念。其本质是指，访问高速设备时使用一组系统总线，比如内存，而访问低速设备时使用另一组系统总线，比如鼠标、键盘、打印机等。这种结构的设计需要 CPU 内部的特别支持，因为 CPU 内部必须设计出多组总线，并且总线之间要考虑总线仲裁，因为 CPU 内核在某一时刻只能从其中一组总线中读取数据。“南”和“北”的概念来源于“上北下南”，因为在基于 Intel CPU 电路板设计中，一般 CPU 在中间，电路板上会布局一些高速器件，比如内存等，下面会布局一些低速器件，从而形成了所谓的南桥和北桥。

当 CPU 上电后，会首先从 CPU 内部的一小段引导程序中执行，因为这段程序是在 CPU 内部，所以也称之为片上 (OnChip) 引导程序。这段程序是 CPU 厂商固化的，不仅程序员，就连硬件工程师也无法修改，除非一些大厂商跟 CPU 厂商协商，从而可以自定义这段引导程序的内容。

片上引导程序的作用非常简单，它的目的就是固定地址装载程序。请注意是“固定”地址，这个地址是在电路板设计之前就固定了的，因此，硬件工程师如果想从 NAND Flash 上读取程序，就必须把 NAND Flash 所占用的地址固定到片上引导程序所预设的地址上，这个地址一般可以通过 CPU 的手册获得。所幸的是在当前基于 ARM 芯片的设计中，CPU 都预留了专有的 NAND Flash 接口，从而使得硬件工程师都不需要特别的地址译码就可以保证 NAND 满足引导程序所需要的地址。

NS 中 NAND Flash 内部的地址分配如图 19-1 所示，图中按地址段进行划分，不同地址段有不同的名称，具体如下。

- **Second Bootloader**: 该分区大小为 2MB，如其名称所示，其作用是进行第二次引导。读者这里要问了，片上引导程序为什么不直接把需要执行的程序加载到内存呢？原因如下，因为片上引导程序很简单，一般只有 1KB 大小，CPU 的设计者并不希望这段代码能够完成任何“伟大”的任务，这段代码仅仅能够读取 NAND 中的数据，并且是按照扇区的方式进行读取。换句话说，如果你的 NAND 被格式化为 FAT32 分区，并且把要执行的程序以文件的方式存放在该 NAND 上，那么对不起，CPU 内部的引导程序无法识别该文件，因为这需要 FAT32 文件系统驱动的支持，而驱动本身就是软件。这部分软件大家可以想象，1KB 的代码肯定是实现不了的，当然，



这段代码也无法从以太网上下载程序，因为它不包含以太网的驱动。而在像 Linux 这样复杂的系统开发中，为了开发的便利，程序员希望系统能够直接从某个 Ext 文件系统中读取系统程序并执行，或者从以太网上读取系统程序并执行，因为这样做非常方便，操作系统开发者只需要把编译好的系统映像文件放在已有的文件系统中即可。而这就是 Second Bootloader 的作用，其设计的思想是，设计一个只有接口驱动的。没有任何应用的最小系统，然后借助于该最小系统去加载真正的操作系统。二次引导程序可以识别 Ext 文件系统，可以驱动以太网口，具有 USB 接口驱动，因为该段程序没有任何具体的应用程序，所以占用的空间相对 Linux 内核而言比较小，不超过 2MB。这段代码一般需要通过 JTAG 仿真器进行固化，而一旦固化后，一般就不需要改变了。JTAG 仿真器是一种硬件调试工具，其价格一般较高，JTAG 的全称是 Joint Test Action Group，它是一个组织，也是一种 CPU 接口规范。该规范要求兼容的 CPU 都需要留出 14 个引脚，这些引脚在 CPU 内部与其他引脚有一定的逻辑关系，可以通过向这 14 根引脚输入不同的内容来模拟其他引脚上的状态，从而达到测试或者控制 CPU 内部运行状态的目的，其最直接的作用就是可以通过 JTAG 引脚来单步调试 CPU 的运行，并且可以通过仿真器向 CPU 内部加载程序。如果这 2MB 内的内容损坏，那就悲剧了。

由于二次引导程序的作用是通用的，因此有一些爱好者及社区开发了一些开源的代码，fastboot 就是其中一种。该段代码可以读取 Ext 分区中的 update.zip 来更新系统，也可以通过 USB 连接 PC，并将 PC 提供的映像文件写入指定的分区。

片上引导程序运行后，首先加载二次引导程序。这里有一个问题，要把这些内容加载到哪里呢？这个地址一般是在编译、链接时指定的，它取决于二次引导程序的开发者，理论上是任意的，但也要受限与系统中的物理内存地址。一般情况会把这段程序加载到 SDRAM 中运行，并且加载到的地址不能和最终要加载的 Linux 运行地址冲突。假设 Linux 内核在编译时指定的运行地址从 0x4000 0000 开始到 0xf000 000，那么二次引导程序的加载地址应该在 0x4000 0000 之前。

- misc: 不详。
- boot: 这就是读者常说的 Linux 内核，空间为 8MB。
- recovery: 保存和 boot 分区中原始的内容，所以大小也是 8MB，当用户恢复出厂模式时，系统会用这部分内容覆盖 boot 分区。
- cache: 不详。
- Radio: 无线通讯模块。手机内部至少包含两个处理器，一个是应用程序处理器，运行像 Linux 这样的系统，给用户提供各种应用的界面等，另一个是基带处理器，用于提供底层的无线通讯，比如电话、短信、3GB 上网等。而基带处理器实际上也是一个嵌入式系统，也需要可执行文件，Radio 分区的内容正是该子系统所需要的执行代码，与 boot 分区有点技术相似性。
- efs: 不详。

- **system**: 这正是大家熟悉的 Android 内核文件, 其中包含 Android 所需的各种驱动库、应用程序等, 该分区大小为 503MB。
- **data**: 保存各种应用程序所需要的 data 文件, 大小约为 1GB
- **sdcard**: 这个名称有点让人误解, 表面上看是 SD 卡, 而实际上依然是 NAND Flash, 该名称仅仅是为了和以前版本兼容, 因为在之前的 Android 手机上, 内部使用 SD 卡进行内存扩展, 该分区大小约 13GB。

## 19.2 各种映像 (Image) 文件的作用

上节概述了系统的地址空间, 以及 NAND Flash 内部地址空间中的各个分区, 本节就来介绍这些分区中对应的映像文件。

首先, 需要明确一点, Image 文件并不是什么标准格式的文件。有些读者看到 system.img、boot.img 等文件名称时, 起初都以为 .img 是某种特定格式的文件, 像 doc、.zip 等, 但事实上, .img 文件并没有专有的格式规范。

Image 文件最早的语义是映像文件, 比如可以把一段内存中的数据导出为一个映像文件, 映像文件的内容就是内存中的数据, 反过来, 也可以把这个映像文件导入到内存中。再后来, 除了内存的映像外, 可以把 ROM 或者 Flash 期间中的数据保存为一个映像文件, 或者把一个映像文件写入到 ROM 或者 Flash 期间中, 这一般用于所谓的“刷机”或者“烧写”。因此, 在嵌入式领域, 映像文件一般是指保存了存储器数据的文件, 在实际操作中, 为了能够完成导入或者导出的目的, 往往需要在映像文件中记录少量的地址信息。比如, 一般的映像文件头部会使用两个 int 数据分别保存对应的内存地址和数据长度, 从而当导入时知道将这些数据导入到内存的什么地方。不同的映像文件头部会根据具体应用定义不同的信息, 这完全取决于具体的应用, 但思路是一致的, 而这种文件也就称之为映像文件, 至于该文件的名称和扩展名则完全没有标准规范, 但在一般情况下, 大家习惯使用 .img 这个扩展名来表示这种文件。

在 Android 中, 常见的映像文件包括 boot.img、zImage、ramdisk.img、system.img, 下面就来具体分析这些映像文件的结构和作用。

首先来看 boot.img。该文件内部保存的是底层 Linux 对应的映像文件, 其内部格式在 bootimg.h 文件中有定义, 该文件的路径是 ./system/core/mkbootimg/bootimg.h, 该文件中清楚地定义了 boot.img 的内部构成。为了叙述方便, 下面以图形化展示其内部组成, 如图 19-2 所示。

boot.img 内部由四部分组成。第一部分是头信息, 头信息所占据的大小为一个 Page, 在 NS 的 NAND Flash 中, 一个 Page 的大小是 4096。

第二部分是 kernel 数据, 所占据的大小为 n 个 Page, n 的值可以由头信息中相应的数值计算出来。

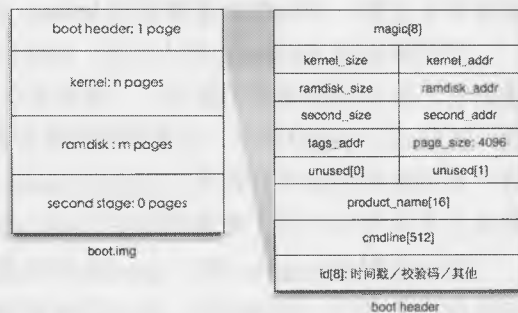


图 19-2 boot.img 的内部结构

第三部分是 ramdisk 数据, 大小为  $m$  个 Page,  $m$  的值也可以由头信息中相应的数值计算出来。

第四部分是 sencond stage 数据, 在目前的系统中, 该结构虽然在 bootimg.h 中有所一定, 但实际的 boot.img 中对应的数据长度却始终是 0。

头信息中的数据含义如下。

- magic[8]: 映像文件的魔数, 为固定的值 “ANDROID!”, 魔数的作用是给文件系统提供一个信息, 从而使得文件系统可以通过这个魔数判断该文件是什么类型的文件。
- kernel\_size: Linux 内核程序的大小。
- kernel\_addr: 内核程序的起始地址, 引导程序将把 kernel 程序装载到该地址指定的位置上。
- ramdisk\_size、ram\_disk\_addr: ramdisk 程序的大小和起始地址, 和 kernel\_size 类似。
- tags\_addr: 暂不详。
- page\_size: 定义 NAND Flash 的页大小, NS 中使用的是 4096。
- unused[2]: 这两个值暂不使用, 为以后扩展预留两个地址位。
- product\_name[16]: 产品名称, 可以自由指定, 长度不能超过 16。
- cmdline[512]: 命令行, 内核启动后会执行该命令行对应的命令。
- id[8]: 可以保存一些额外信息到这里, 比如时间戳等。

头信息一般不会占满一个 Page, 剩余的空间用 0x0000 或者 0xffff 填充。

boot.img 中包含了两个重要 img 文件, 分别为 kernel 和 ramdisk.img, 下面来看这两个 img 文件的作用, 首先来看 kernel。

大家都知道, 所有源程序首先要经过编译器将其转换为机器码, 一般是 .obj 类型的文件, 然后再经过连接器把机器码中的相对地址转换为绝对地址, 此时这就是一个可执行程序了。该执行程序需要引导程序将其装载到连接时所指定的绝对地址中, 方可执行。这段程序如果很小的话, 我们一般称之为嵌入式程序, 如果这段程序很大的话, 比如 Linux, 情况似乎复杂了一点, 但本质却是相同的。Linux 内核本身也是程序, 这段程序在编译及连接后会产生一个可执行程序, 为了便于装载这个程序, 一般对其进行一定的压缩, 比如经过 zip 压缩以减少文件本身的大小, 压缩后的文件就是大家所熟知的 zImage。如果没有人介意, 可以将其重命名为 kernel, 没有扩展名, 这就是 boot.img 中 kernel 分区保存的文件。

下面再来看 ramdisk。如上所述, Linux 编译后的程序尽管可以执行了, 但对于 Linux 这样复杂的系统而言, 为了应用开发的便利, 一般会提供一些常用的应用程序, 比如 cp 命令, mv 命令、chown 命令等。当 Linux 内核启动后, 会读取根目录中的 init.rc 文件, 而该文件内部使用了一些诸如 cp、mv、chown 等一系列常用的命令行程序。因此在 Linux 启动后、读取 init.rc 之前, 会装载一个小型的文件系统, 并且该文件系统中包含了常见的这些 Linux 命令, 而这个文件系统就是 ramdisk.img 的作用。

为什么叫做 ramdisk 呢? ramdisk 的本意是“内存盘”, 即把一段内存当作一个磁盘用, 其优点是访问速度会非常快, 包括读和写, 缺点是容量有限, 因为系统内存是十分紧张的资源。并且如果这个 ramdisk 在初始化后只读取, 而不写入, 那么实现起来将非常简单。在这种情况下, 文件可以在连续地址上存储, 所以不需要跨页索引表, 并且如果 ramdisk 中的文件没有文件夹层次, 就更简单了。嵌入式 Linux 中正

是将 ramdisk 设计为拥有这样特点的文件系统，该文件系统中没有文件夹，并且是只读的。ramdisk.img 保存的就是一些常用的 Linux 命令程序文件，Linux 内核中包含一个非常简单的读取 ramdisk 中文件的驱动，这个驱动简单的都没有必要将其称为文件驱动。可以设想，只需要在 ramdisk 的头部建立一个文件名称表，表中包含文件的起始地址和长度，从而就可以读取 ramdisk 中的文件了。

ramdisk 被加载后，会一直驻留在系统中。

最后再来看 system.img 文件。

和 ramdisk.img 不同，system.img 是一个包含了文件系统的映像文件，该文件系统的类型是 yaffs，yaffs 只是一个名称，就像 FAT16、FAT32、NTFS 一样。yaffs 文件系统是 Linux 的作者开发的一个小型文件系统，最初主要用于基于 NAND Flash 器件的文件系统。Android 中提供了两个工具，分别是 yaffs 和 unyaffs 程序，通过 yaffs 工具可以将 PC 上的某个文件夹中的内容转换为一个 yaffs 文件系统，通过 unyaffs 可以从 yaffs 文件系统中提取出所包含的文件。

在嵌入式开发过程中，开发人员习惯说“把某个文件夹压缩为 ramdisk.img”或者“将某个文件夹压缩成一个 yaffs 文件系统”。从严格意义上讲，ramdisk.img 的确是经过压缩而成，而对于 system.img 而言应该说“转换为 yaffs 文件系统”，因为 yaffs 的作用没有任何压缩的成分。比如当某个文件夹 folder 目录下包含一些文件或者文件夹时，这些文件在 PC 的文件系统中除了占用具体的文件存储空间外，还占用一些额外的文件分配表空间，文件系统将根据这些分配表信息读取具体的文件。yaffs 转换的本质是创建一个 yaffs 文件系统，然后将 folder 中的文件写入到 yaffs 文件系统中，最后再将该文件系统的所有信息保存为一个文件，这就是 system.img。

当 Android 启动时，会挂载 NAND Flash 中的 system 分区，并指定挂载所使用的文件系统类型为 yaffs，从而使得 yaffs 文件系统驱动会被加载，来读取 system 分区中的文件。关于这一点有些读者可能会有一个疑惑：为什么在 system.img 被加载后还可以给该 yaffs 文件系统中添加新文件呢？大家都知道，任何一个文件系统都是有大小的，比如 PC 文件系统的大小就是磁盘的大小，而当把 folder 目录转换为一个 yaffs 文件系统后，该 yaffs 文件系统的大小就应该是固定的。换句话说，当进行 yaffs 转换时，yaffs 工具会根据 folder 目录中的文件大小创建一个文件系统，并且会设置该文件系统的大小，然后再把这些文件写入到这个文件系统。当 system.img 被加载后，system 对应的 yaffs 文件系统大小就应该是在 PC 上进行转换时所指定的大小，而这个大小仅能保存 folder 中的目录。那么，又如何向 system 分区继续添加新文件呢？会不会发生空间不够的情况呢？

事实上，文件系统只能计算出已经占用的空间大小，文件系统本身无法知道可用空间的大小，这个大小是 NAND Flash 驱动读取的，并且驱动会把这个大小报告给文件系统。文件系统在每次启动时都需要调用 NAND Flash 的驱动读取总的空间大小，并且把这个大小存入文件系统的临时变量中，这个值并不会被写入到文件分配信息表中，这就是为什么 system 分区被加载后还能继续添加新文件的原因。该过程可如图 19-3 所示。

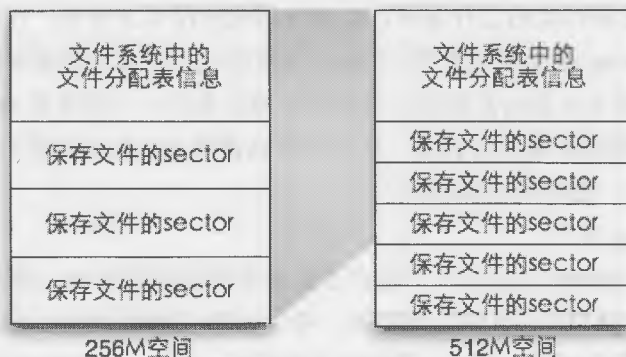


图 19-3 不同大小的文件系统内部结构示意图

在如图-19-3 中, 假设 folder 目录经过 yaffs 转换后需要占用的总磁盘空间为 256MB, 那么可以把这个 256MB 大小的 system.img 原封不动地写入到 Android 设备中 512MB 的 system 分区中, 剩下的 256MB 空间中还可以继续添加新的文件, 而不会对 yaffs 文件系统造成任何影响。

## 19.3 编译 Nexus S ( NS ) 的 Image 文件

本节开始介绍如何生成 NS 所需的各种 image 文件。

### 19.3.1 编译 Linux Kernel

NS 中使用的 kernel 地址为 `git://android.git.kernel.org/kernel/samsung.git`, 可以使用 `git clone` 命令将该 Git 库下载到本地, 如以下命令所示。

```
git clone git://android.git.kernel.org/kernel/samsung.git kernel
```

命令中最后的参数 `kernel` 是本地的文件夹名称, 可以将该 `kernel` 目录放到 `android` 根目录下。下载后, 本地仓库中默认的远程仓库名称为 `origin`, 并且会默认在本地创建一个新的 `branch`, 名称为 `android-samsung-2.6.35`, 可以使用 `git branch` 命令查看本地分支。

以后当需要和远程仓库同步时, 可使用 `git pull` 命令, 如下:

```
git pull origin
```

本章以 Mac 系统为例介绍编译过程, 笔者所使用的 Mac 系统版本为 10.6.6。在 Mac 上编译 kernel 前, 需要做一点额外的系统配置, 如下:

- ① 安装 `libelf` 库, `sudo port install libelf`。
- ② 创建一个软链接, `sudo ln -s /opt/local/include/libelf /usr/include/libelf`, 因为用户路径默认为 `/usr/include`, 该软链接可以解决 `libelf` 库的引用地址。

③ Mac 系统默认没有 `elf.h` 文件, 该文件在 `kernel` 编译时需要, 因此下载一个该文件, 并赋值到 `/usr/include` 目录下。

④ 安装 `gsed` 工具, `sudo port install gsed`。

⑤ `/usr/include` 目录下默认并没有 `malloc.h`, 但是在 Mac 的系统目录却存在, 因此, 在 `/usr/include` 目录下创建一个 `malloc.h`, 文件的内容就一句话: `include <sys/malloc.h>`, 以解决 `malloc.h` 文件引用的地址的问题。

好了, Mac 下的配置就完成了, 接下来就可以编译 `kernel` 了。编译 `kernel` 的方法和普通 Linux 编译的过程一样, 需要指定所使用的编译器, 并且指定 `kernel` 所使用的配置文件, 幸运的是在下载 `kernel` 中已经存在了 NS 所需的配置文件。以下代码是笔者所使用的一个脚本文件, 名称为 `mymake.sh`, 大家可以把这个文件放到自己的 `kernel` 目录中, 并使用 `chmod a+x` 命令赋予其可执行的属性。

```
export PATH=/Users/keyd/android22/prebuilt/darwin-x86/toolchain/arm-eabi-4.4.3/bin/:$PATH
make ARCH=arm clean
make ARCH=arm herring_defconfig
make -j4 ARCH=arm CROSS_COMPILE=arm-eabi-
```

笔者的 Android 源码路径为 `/Users/keyd/android22`, 大家需要把这个地址替换成自己的 Android 源码路径。在 Mac 上所使用的 ARM 编译器为 `arm-eabi-4.4.3`, 因此需要把这个路径包含在 `PATH` 中。

`herring_defconfig` 就是专门针对 NS 的 `kernel` 配置文件, `herring` 是 NS 的内部产品名称。

最后的 `make` 命令中包含了对 `CROSS_COMPILE` 变量的赋值。

执行完以上命令后, `kernel` 就开始编译了。Linux 编译所需的时间显然小于 Android 的编译, 一般几分钟就完成了, 完成后会在 `kernel` 的 `arch/arm/boot/` 目录下产生一个 `zImage` 文件, 该文件就是 Linux 内核对应的映像文件。

### 19.3.2 提取 NS 的私有驱动文件

上一节介绍了 Linux 内核的编译, 本来应该接着介绍编译 Android 的映像文件, 但 Android 的映像文件却依赖一些设备的私有文件, 因此本节先来介绍什么是私有文件, 如何得到这些私有文件。

对 Android 硬件系统而言, 一般至少包含三个 CPU 内核, 分别为:

- 基带处理器内核, 主要完成无线基带的数字信号处理, 比如语音的数字调制、解调等。
- 应用处理器内核, 主要实现用户操作系统, 给用户提供一个图形界面, 执行用户所使用的各种应用软件。
- 多媒体处理器内核, 主要完成音视频编解码, 比如 `mpeg`、`H.263` 等不同编码格式, 该处理器一般也使用数字信号处理器。

在目前的芯片技术中, 一般会将多媒体内核和应用内核做到一个芯片中。常见的 TI 公司的 OMAP 芯片, 内部就包含一个 ARM 内核和一个 DSP 内核, 再比如高通公司的 MSM 处理器, 内部也包含一个



ARM 内核与多个 DSP 内核。

对于设备厂商而言，出于两种原因，他们往往会隐藏一些特殊功能代码的源码，而仅提供目标码。

- 技术原因。如前所述，当一颗芯片中包含 ARM 和 DSP 内核时，DSP 所执行的程序必须由 DSP 的编译器进行编译。而 ARM 执行的程序必须由 ARM 编译器编译，由于不同处理器所使用的 DSP 类型不同，就导致编译器不同，因此 Android 源码中无法提供不同的 DSP 编译器。既然没有编译器，那么就算得到 DSP 所需要的源码，对于工程师来讲也无法使用。
- 技术保护。对于一些包含了专有算法的程序，其价值量大，算法的开发者不希望这些代码开源，甚至设备商也不能得到这些程序的源码。比如基带处理中包含的各种通信算法、DSP 处理器中包含的多媒体编解码算法，以及 GPS 信号处理的算法。

出于以上两种原因，这些私有文件一般都是由原始的技术提供商进行开发，并将目标码提供给设备商，因此，Android 源码中无法包含这些私有文件。

那么，如何得到这些私有文件呢？方法很简单，就是直接从手机中读取，但问题是我们怎么知道该设备都有哪些文件呢？为了解决这个问题，Android 中的工程师针对不同的设备提供了相应的脚本文件，该脚本文件中包含了指定设备所需要的全部私有文件列表，在编译 Android 前，需要将手机连接到电脑，然后执行该脚本，从而获取这些私有文件。

对于 NS 而言，该脚本的路径是 `/device/samsung/crespo/extract-files.sh`，路径中 `device` 是 android 根目录下的 `device` 目录，`samsung` 为设备商名称，`crespo` 为设备名称，NS 的内部名称是 `crespo`。注意和前面所讲的 `herring` 区分，`crespo` 是 NS 的设备名称，而 `herring` 仅仅是 NS 所使用的 Linux 内核的内部名称。

执行该脚本后，会在 android 的根目录下创建一个 `vendor/samsung/crespo/` 目录，该目录中的 `proprietary` 子目录中就包含了所有的私有文件。`extract-files.sh` 并不是 Android 程序员手工编写的，而是通过 `device/common/generate-blob-scripts.sh` 脚本生成的。

### 19.3.3 编译 system.img 文件

获得了私有文件后，接下来就可以编译 `system.img` 了。编译命令很简单，如以下代码所示。

```
./build/envsetup.sh
make -j4 PRODUCT-full_crespo-eng
```

第一句先进行编译环境配置，该命令是以 `source` 的方式运行的，即命令为一个点，`source` 方式运行后，脚本所创建的环境变量在脚本执行完毕后依然有效。

第二句中 `full_crespo` 是产品名称，该名称在 `device/samsung/crespo/full_crespo.mk` 文件中定义；`eng` 是标签名称。产品名称可以无限多，只要有相应的定义文件即可，而标签的名称只有三种，分别为 `user`、`userdebug` 及 `eng`。`eng` 的意思是 `engineer`，即工程模式。源码中不同的模块都包含了一个 `LOCAL_MODULE_TAG` 变量，该变量的值为标签名称的一种，选择不同标签编译时，就会选择相应的



模块。对于 eng 模式而言，编译出的 system.img 中将包含一些调试所用到的工具。

-j4 指明使用 PC 的几个内核进行编译，仅仅是为了提高编译速度。如果你的电脑是 4 核，那么就可以使用 -j8，数字为实际内核乘以 2。

好了，执行完以上命令后，就开始等待吧，该过程会持续 1 个多小时，在这个过程中请大家思考一个问题：上一节所编译出的 Linux 内核 zImage 为什么没有参与 system.img 的编译？

事实上，在 /device/samsung/crespo 目录下存在一个名称为 kernel 的文件，该文件正是 Android 中为 crespo 产品自带的 Linux 内核文件，该文件的内容就是 zImage，只不过不是我们编译的 zImage。

Android 编译完毕后，产生 out/target/product/crespo 目录，该目录中包含了所有所需的 image 文件，比如 ramdisk.img、boot.img 及 system.img。

除了这些 image 文件外，还存在一个 root 目录和 system 目录，root 目录中的内容正是 ramdisk.img 的内容，system 目录的内容正是 system.img 的内容。

前面讲过 system.img 是使用 yaffs 文件系统的映像，换句话说，当 Android 编译完毕后，你想给该目录中添加一些特别的程序，则可以把该程序放到 system 目录下，然后重新将 system 目录转换为 yaffs 文件系统的映像，转换的方法是使用 yaffs 工具。所不幸的是，在 Android2.3 版本中，system.img 不再采用 yaffs 文件系统，而是采用 Linux 中的 ext4 文件系统，因此，下面分别列出制作 yaffs 和 ext4 映像的过程。对于 yaffs，代码如下：

```
PATH=/Users/keyd/android/android22/out/host/darwin-x86/bin:$PATH
mkyaffs2image system s.img
```

mkyaffs2image 是 Android 自带的 yaffs 转换工具，该工具在 out/host 目录下，该命令的第一个参数是要转换的目录，第二个参数是转换后的 image 文件名称。

而对于 ext4，代码如下：

```
PATH=/Users/keyd/android/android22/out/host/darwin-x86/bin:$PATH
make_ext4fs -s -l 536870912 -a system s.img system
```

其中参数 -l 后面数字的 16 进制是 0x20000000，即 16G。-a 参数指定挂载点，此处将挂载到设备的根目录下的 system 目录上。后面三个参数分别为输出的文件名称和要转换的目录名称。-s 参数很重要，该参数的意义是告诉 make\_ext4fs 工具去掉空数据，如果不加该参数，image 文件的大小将等同于 16GB 的实际大小，

make\_ext4fs 也是 Android 自带的工具。

### 19.3.4 创建 ramdisk.img

ramdisk.img 是编译完 Android 后产生的，ramdisk.img 中包含的程序仅仅依赖于 bionic 库，以及 Linux 内核的相关头文件及库文件，而不需要 Linux Kernel 的源码，因此 Android 源码中就可以直接编译出 ramdisk.img。ramdisk.img 的输出目录和 system.img 目录相同。

前面讲过, `ramdisk.img` 是一种没有文件系统的映像文件, 因此, 制作 `ramdisk` 的过程非常简单, Android 源码中提供了 `mkbootfs` 工具用于将 `boot` 目录中的程序打包成 `ramdisk.img`, 语法如下:

```
cd /Users/keyd/android/android22/out/target/product/crespo
mkbootfs boot | gzip > ramdisk2.img
```

在以上命令中, `boot` 目录的内容就是 `ramdisk.img` 文件中包含的内容。 `mkbootfs` 这个名称容易让大家误解, 该名称表面上似乎是要产生 `boot.img`, 而实际上仅仅是产生 `boot.img` 中包含的 `ramdisk.img` 而已。

### 19.3.5 创建 boot.img 文件

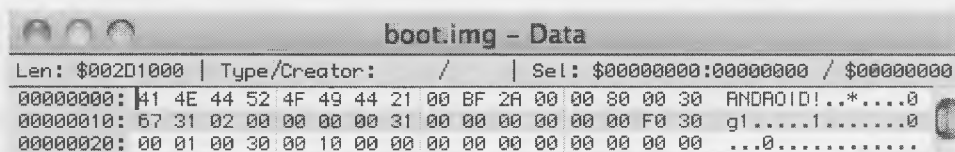
创建 `boot.img` 有两种方法, 第一种是使用 Android 源码中提供的 `mkbootimg` 工具, 如以下代码所示:

```
PATH=/Users/keyd/android/android22/out/host/darwin-x86/bin:$PATH
mkbootimg --kernel zImage --ramdisk ramdisk.img --cmdline 'console=ttyFIQ0 no_console_suspend'
--base 0x30000000 --pagesize 4096 -o boot2.img
```

该命令中 `--kernel` 选项用于指定 Linux kernel 编译后输出的 `zImage` 文件路径, `--ramdisk` 用于指定 `ramdisk.img` 的文件路径, `--pagesize` 用于指定实际 NAND Flash 中的扇区 (Sector) 大小, `--o` 选项用于指定输出的 `boot.img` 文件名称, `--cmdline` 用于指定一些系统启动时的命令, 其中 `no_console_suspend` 标识将赋予命令程序 `root` 权限。

注意 `--base` 选项指定 Kernel 被装载到的内存起始地址, 这个值在 NS 中必须指定为 `0x3000 0000`, 如果不指定该地址, `mkbootimg` 默认会使用 `0x1000 0000`, 这就会导致不能正确启动系统。

读者们可以用二进制文件查看器查看 `boot.img` 的头信息, 如图 19-4 所示。



boot.img - Data		
Len: \$002D1000	Type/Creator: /	Sel: \$00000000:00000000 / \$00000000
00000000: 41 4E 44 52 4F 49 44 21 00 BF 2A 00 00 00 00 30	ANDROID!..*....0	
00000010: 57 31 02 00 00 00 00 31 00 00 00 00 00 F0 30	g1.....1.....0	
00000020: 00 01 00 30 00 10 00 00 00 00 00 00 00 00 00	...0.....	

图 19-4 boot.img 文件的头信息

创建 `boot.img` 的另一种方法是直接把 Linux kernel 编译后产生的 `zImage` 文件重命名为 `kernel`, 并用该文件覆盖 `device/samsung/crespo/kernel` 文件, 然后重新编译 Android 源码。

## 19.4 使用 fastboot 写入 Image 文件

前面介绍了产生各种 `image` 文件的方法, 本节就来介绍如何把这些 `image` 文件写入 NS 内部的 NAND Flash 分区。

NS 在出厂时已经给二次引导分区中通过 JTAG 仿真器写入二次引导程序, 名称为 fastboot, 从而使得每次系统开机后, 首先从 CPU 的片上引导程序开始执行, 然后把二次引导程序装入内存, 然后开始执行二次引导程序。此时, 如果用户同时按下音量键和开机键, 那么二次引导程序就会开始进入 image 写入等待状态。注意是音量调大按键, 而不是音量调小按键。

Android 源码中同时也提供了一个 fastboot 工具, 该工具在 PC 端执行, 其作用是和 NS 设备中的二次引导程序通信, 用户可以使用该 fastboot 工具将前面所编译的 image 文件写入到 NS 设备的不同分区中。

NS 在重写 Flash 分区之前, 需要先对设备进行解锁。所谓的解锁是指, 为了保护系统, 设备在出厂时禁止向 Flash 分区写入新的系统, 除非用户完全确认自己在进行解锁。因此, 解锁需要特别的命令, 解锁只需要执行一次, 如以下代码所示:

```
fastboot oem unlock
```

解锁后, NS 启动时界面下方将出现一把锁的图像。不过要注意, 解锁后手机上的原有数据将全部删除, 因此解锁前先备份好已有的数据。

fastboot 的命令格式如以下代码所示:

```
fastboot flash partition_name imageFile
```

参数 partition\_name 必须为固定名称的分区, 其值可以是 boot、system。imageFile 就是对应的 image 文件名称。

除了写入 image 文件外, fastboot 还提供了清空分区的命令, 如下:

```
fastboot erase partition_name
```

常用的情况是需要清空 data 分区, 于是可以调用:

```
fastboot erase userdata
```

好了, 读者可以使用该工具将前面所产生的 boot.img 和 system.img 写入 Flash 的相应分区中, 然后重新开机, 等待运行结果吧。以下是笔者重启后的运行结果, 从“设置->关于手机”中可以查看所使用的 Linux 内核版本, 如图 19-5 所示。



图 19-5 作者手机上的版本信息

## 19.5 最后验证

按照以上过程启动后发现, 系统存在两个问题。

- 触摸按键中除了“Home”键外, 其他按键均操作无效, 并且按后屏幕会闪动一下, 出现一次顿挫。
- 音量按键和电源按键均无反应。

为了解决这两个问题，首先就需要查看 logcat，根据 logcat 的输出信息，最后找到了解决的办法。

### 19.5.1 解决触摸按键问题

当按下触摸按键后，logcat 输出：

```
W/KeyCharacterMap( 800): No keyboard for id 0
W/KeyCharacterMap( 814): Can't open keycharmap file
E/KeyCharacterMap( 814): Can't find any keycharmaps (also tried
/system/usr/keychars/qwerty.kcm.bin)
```

通过 find 和 grep 命令，找到以上出错信息是在 base/libs/ui/KeyCharacterMap.cpp 文件的 load() 函数中。从该函数中可以分析出，系统准备要装载一个文件，其路径为：

```
snprintf(path, sizeof(path), "%s/usr/keychars/%s.kcm.bin", root, tmpfn);
```

而这个文件的名称 tmpfn 是通过 getprop() 获得的：

```
sprintf(propName, "hw.keyboards.%u.devname", id);
err = property_get(propName, dev, "");
if (err > 0) {
    // replace all the spaces with underscores
    strcpy(tmpfn, dev);
```

所以，属性名称为 hw.keyboards.0.devname，此时大家可以通过 adb shell 进入设备的控制台，然后调用 getprop hw.keyboards.0.devname 查看是否有属性值。测试结果果然是没有该属性，于是可以先手工调用 setprop hw.keyboards.0.devname cypress-touchkey 设置该属性。为什么属性值为 cypress-touchkey 呢？因为我们在 /system/usr/keychars 目录下就只发现了这个触摸按键映射文件。设置完该属性后，再尝试按触摸按键，果然正确。

所以，可以把这个设置属性的命令添加到 init.herring.rc 文件中，如以下代码所示：

```
on boot
    mount debugfs /sys/kernel/debug /sys/kernel/debug
    setprop hw.keyboards.0.devname cypress-touchkey
```

从而，每次系统开机后都自动具备该属性。

修改完毕后，需要重新执行 make -j8 PRODUCT-full\_crespo-eng，然后再使用 fastboot 重新烧写 boot.img 即可。

### 19.5.2 解决音量和电源键

当按音量或者电源按键后，logcat 如下输出：

```
W/KeyCharacterMap( 828): Can't open keycharmap file
```

```
W/KeyCharacterMap( 828): Error loading keycharmap file
'/system/usr/keychars/herring-keypad.kcm.bin'. hw.keyboards.196609.devname='herring-keypad'
```

从输出信息可以看出 `hw.keyboards.196609.devname` 的属性是存在的, 即为 `herring-keypad`, 然而在 `/system/usr/keychars` 目录下却找不到 `herring-keypad.kcm.bin` 文件, 而只有一个名称为 `s3c-keypad.kcm.bin` 的文件。显然, 这只是名称上的错误, 可是 `s3c-keypad` 这个文件是哪里来的呢?

查看 Android 本目录下的 `/device/samsung/crespo` 目录, 发现有几个 `s3c-keypad` 相关的文件, 分别为 `s3c-keypad.kcm` 和 `s3c-keypad.kl`。因此, 尝试将这两个文件重新命名为 `herring-keypad`, 并且将该目录下的 `Android.mk` 和 `device.mk` 文件中的文件引用都修改为 `herring-keypad`, 具体包括:

```
./device/samsung/crespo/Android.mk:LOCAL_SRC_FILES := s3c-keypad.kcm
./device/samsung/crespo/device.mk:
device/samsung/crespo/s3c-keypad.kl:system/usr/keylayout/s3c-keypad.kl \
./device/samsung/crespo/device.mk: s3c-keypad.kcm \
```

修改完毕后, 重新调用 `make -j8 PRODUCT-full_crespo-eng` 编译 Android, 然后将生成的 `system.img` 写入到 Flash 中并重新启动。

启动后发现, 音量按键和电源键依然不起作用, 然而查看 `logcat` 的输出中却并无出错信息, 那么 Android 内核到底收到按键消息了吗? 为了验证这一点, 可以写一个最简单的 Activity, 并且重载 `dispatchKeyEvent()` 函数, 然后尝试捕获按键消息, 如以下代码所示:

```
public boolean dispatchKeyEvent(KeyEvent event) {
    int keyCode = event.getKeyCode();
    int code = event.getScanCode();
    int action = event.getAction();
```

启动该 Activity 后, 当按下音量键或者电源键后, 这段代码都被执行到了, 可是 `keyCode` 的值总是 0, 而 `code` 的值却对应了不同的按键, 于是猜测是按键码映射错误。

打开 `/device/samsung/crespo` 目录下的 `herry-keypad.kl` 文件发现三个键的定义如下:

```
key 42    VOLUME_UP      WAKE
key 58    VOLUME_DOWN    WAKE
key 10    POWER          WAKE
```

而这三个值正是 `event.getScanCode()` 获得值, 然而这三个值在 `qwerty.kl` 中的定义却如下:

```
key 115   VOLUME_UP      WAKE
key 114   VOLUME_DOWN    WAKE
key 116   POWER          WAKE
```

因此, 不妨尝试改变一下看看, 即将 `herring-keypad.kl` 中关于这三个按键的值定义为 `qwerty.kl` 中定义的值, 重新编译后再测试, 一切正常。这说明源码由于 `s3c` 和 `herring` 产品的差别导致源码名称定义尚不规范, 并且发生了按键码值定义错误。

### 19.5.3 WIFI 问题

这个问题笔者没有深究,在 Google 上搜索了一下,据说是 WIFI 的驱动文件 `bcm4329.ko` 本身并没有被放在 Linux Kernel 编译后的 `zImage` 文件中。尽管该文件的确被编译了出来,但编译后放在了 `kernel/drivers/net/wireless/bcm4329/bcm4329.ko` 路径下,而 Android 在运行时会从 `/system/modules` 目录下加载该问题,因此只需要将该文件放到 `/system/modules` 目录下即可,如以下代码所示:

```
./adb push /Users/keyd/android22/kernel/drivers/net/wireless/bcm4329/bcm4329.ko  
/system/modules/
```

至此,就完成了自定义 ROM 的编译。但是,由于 AOSP 源码默认并不带 Google 服务,因此对于很多使用 Gmail 的朋友来讲有点遗憾,在编译出的系统中暂时无法安装 Google 服务,比如 Gmail、Gtalk、Google Maps 等,尤其对于那些将手机通讯录保存在 Gmail 中的朋友。不过默认编译出的 Email 程序却可以连接 Google 的 Gmail 服务器,并且可以同步 Gmail 上的联系人。配置服务器连接时,需要使用如图 19-6 所示的配置。



图 19-6 配置 Gmail 账户

在该配置中,主要的参数如下:

- 域名\用户名: 填写完整的 Gmail 邮箱。
- 服务器: `m.google.com`, 而不是默认的 `gmail.com`。
- SSL 连接: 使用 SSL 连接, 并且接受所有的 SSL 证书。

配置完成后, Gmail 的联系人就可以同步到本地了。

除了以上问题外, AOSP 版本中的谷歌输入法也有一个问题, 就是候选框中的单词不能切换, 这直接导致该输入法无法使用。因此, 在编译时, 大家可以先从 `product` 的配置文件中删除 `PinyinIME` 这个项目, 即从 `build/target/product/full_base.mk` 中的 `PRODUCT_PACKAGES` 变量中删除 `PinyinIME`。

### 19.5.4 安装 Google Mobile Service (GMS)

GMS 是指 Google 开发一些程序, 比如 Gmail、Google Map 等。由于 AOSP 版本默认并不包含这些服务的源码, 因此, 这些服务只能以 APK 包的形式进行安装, 而在安装的过程中有很多读者都发现安装后不能直接使用, 查看 logcat 时一般会提示某个权限没有被定义, 或者是某个包不存在。

从原理上讲, GMS 的 APK 包和普通 APK 包没有本质区别, 导致不能运行的原因有三个。

第一, GMS 包中需要一个特别权限, 而这个权限必须在 Framework 中被定义。在程序安装一章中介绍过, Framework 中定义的 Feature 及权限是在 /system/etc/ 目录下的相关 XML 文件中定义的, 因此, 解决这个问题的方法就是在该目录下添加一个新的文件, 并声明相关的权限即可。

第二, GMS 包需要一个特别的 Jar 包库, 这个库在 Framework 中是没有的。比如 Google Map 就需要一个地图库, 因此, 只需要把这个 Jar 包复制到 /system/framework 目录下即可。

第三, 很多 GMS 都需要后台运行一个特别的 Service, 而这个 Service 并不包含在 GMS 的 APK 包中, 而需要单独安装 Google 的一个 APK 包。因此, 在安装这些 APK 前, 首先安装 GMS 所需的服务包即可。

以上三点中除第一个我们可以编写外, 其他两点所需的程序包都必须从出厂的 NS 手机中复制出来, 表 19-1 列出以上三点所需的文件列表, 本书附件中包含这些程序文件。

表 19-1 GMS 所包含的文件清单

文件名称	目标位置
ns/permissions/com.google.android.maps.xml	/system/etc/permissions/
ns/permissions/features.xml	
ns/app/*.apk	/system/app
ns/framework/ com.google.android.maps.jar	/system/frameworks/

将这些文件复制到手机中有三种方法。

第一种是直接使用 adb push 命令将这些文件复制到相应的目录下。

第二种是将这些文件放到 out 目录下最终生成的 system 目录中的相应目录下, 然后重新调用 make PRODUCT-product\_name-variate, 编译完成后 system.img 中就包含了这些相应的文件, 或者使用前面介绍的 make\_ext4fs 命令仅将 system 目录转换为 system.img 即可。

最后一种方法是把这些文件放到 vendor 目录下, 并修改 vendor 目录下的相应 mk 脚本文件, 然后给变量 PRODUCT\_COPY\_FILES 添加这些文件即可。比如 cresso 对应的 mk 文件是 vendor/samsung/cresso/device-vendor-blogs.mk, 可以在该目录下新建一个 vendor/samsung/cresso/gms 目录, 再把上面所讲的 GMS 需要的文件放到该目录下, 然后修改 device-vendor-blogs.mk, 并给变量 PRODUCT\_COPY\_FILES 添加复制的信息即可。

安装完毕后, 系统中将包含 Gmail、Google Maps、Google Market 等各种 GMS 服务, 然后读者就可以通过 Market 程序去下载 Google 拼音输入法, 以及任何其他有趣的应用。

至此, 我们编译出的 ROM 就是一个完整的 Android 手机了。



## 第5部分

# 硬件驱动篇

### 第20章 基于TI OMAP处理器的 Techshine 开发板介绍



## 第 20 章 基于 TI OMAP 处理器的 Techshine 开发板介绍

本章之前主要介绍了 Android 的软件系统，并在第 19 章介绍了基于 Nexus S 手机的内核编译，这对于纯软件研究已经足够，然而，当我们有以下梦想时，这些就不够了。

- 想设计具有某种特殊硬件接口的手机？
- 想把一款 Android 手机和某个玩具连接起来？
- 想扩展硬件系统的内存容量？
- 想用 Android 手机去控制一个飞机，从而能从高处俯视地面？
- .....

因此，本章向大家介绍一款基于 TI OMAP 处理器的开发板 Techv-35xx，该开发板是一款硬件开源的产品，大家可以得到开发板的全部原理图及所使用的芯片资料，有了这些资料后，再加上该开发板的硬件扩展接口，就可以给该电路板扩展各种外围设备，或者和外部电路板连接，从而实现各种梦想中的硬件模型。

Techv-35xx 开发板由北京精仪达盛科技有限公司开发，该开发板基于美国得州仪器(TI)公司的 OMAP3530 处理器进行设计，该处理器内部包含一颗 600MHz 的 ARM Cortex-A8 内核和一颗 430MHz 的 DSP 内核，目前该处理器被用于很多的 Android 手机中，比如 LG、Motorola 等。

北京精仪达盛科技是 TI 在中国的第三方合作伙伴，并且也是 TI 的大学合作伙伴，本章后续内容由达盛科技提供，这些内容是在 Android Donut 版本上进行实验的，Donut 是 Android 1.6 的版本号，有点老了，不过大家可以先基于该版本进行试验，然后再尝试将最新的 GingerBread 版本移植到开发板上，有任何问题可以登录达盛科技公司的官方 BBS 论坛或与达盛客服人员联系，当然，如果不是开发板相关的特定问题，也欢迎与笔者一起探讨。

笔者在完成本书的写作后，计划使用该开发板进行 Android 更底层的研究，比如 Binder 内部实现、SurfaceFlinger 内部机制等，同时，笔者也梦想着能够设计一款硬件开源的手机，使得电子爱好者基于该开源手机扩充一些外部硬件功能，达到物尽其用的目标。

## 20.1 Techv-35XX 开发板概述

Techv-35XX 开发板包含了丰富的外设接口，其功能框图如图 20-1 所示，具体包括 CPU 外设接口中的网口、TV\_OUT 接口、音频输入输出接口、USB、SD/MMC 接口、串口、JTAG 接口、TFT 屏接口、触摸屏接口、按键接口和总线接口。

Techv-35XX 开发板不仅可以单独使用，也可以配置在北京精仪达盛科技有限公司开发的 EL-ARM-DSP-IV、EL-ARM-860 实验系统上使用。

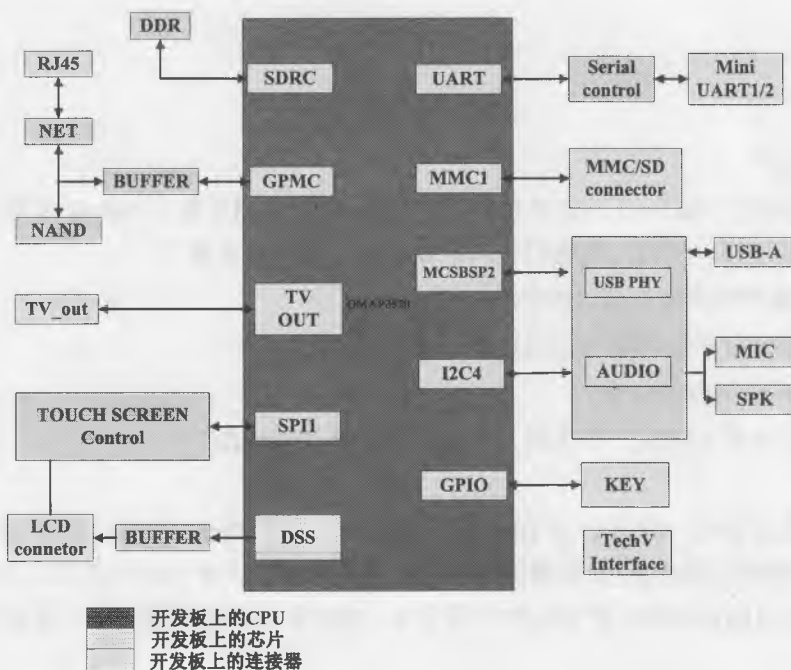


图 20-1 Techv-35XX 开发板结构框图

Techv-35XX 开发板的硬件规格如下：

处理器

- OMAP3530 处理器（完全兼容 OMAP3503 处理器接口）
- 600-MHz ARM Cortex™-A8 Core
- 430-MHz TMS320C64x+™ DSP Core
- ARM 内部集成 16kB I-Cache, 16kB D-Cache, 256kB L2 存储器

- 片上存储 64kB SRAM, 112kB ROM

存储器

- 128MByte 32 位 DDR SDRAM, 166MHz
- 256MByte 16 位 NAND Flash

音频/视频接口

- 1 路输入, 1 路输出;
- TV 输出;

液晶/触摸屏接口

- 8 寸 TFT/3.5 寸;
- 4 线触摸屏

传输接口

- 主从 USB 接口;
- SD/MMC 卡接口;
- 标准 RJ-45 10M/100M 以太网接口;
- 两个标准串口;

输入接口

- 8 个输入按键;
- 3.3V JTAG 仿真接口;

## 20.2 交叉编译环境配置

编译 Techv-35XX Bootload, 首先要搭建基于 Android 的 TechV-3530 平台 Bootload 交叉编译环境, 本章中使用的 Linux 的版本为 Ubuntu 10.04, 下文中简称为 Ubuntu。

### ① 安装交叉编译工具

插入光盘, Ubuntu 默认把光盘挂载到/media/cdrom 目录下, 找到 arm-2007q3-51-arm-none-linux-gnueabi-i686.tar.bz2 这个文件。把这个文件复制到/opt 下, 然后在 Linux 下打开终端输入:

```
cd /opt
sudo tar xvjf arm-2007q3-51-arm-none-linux-gnueabi-i686.tar.bz2
```

### ② 添加环境变量

以上工具安装完成后, 还需要使用如下命令把它们添加到环境变量中:

```
export PATH=/opt/arm-2007q3/bin:/home/u-boot/tools:$PATH
```

注意：用户可把它写入用户目录的.bashrc 文件中，那么系统启动的时候自动完成环境变量的添加，查看路径可以使用 echo \$PATH 命令。如果没有添加环境变量，请在编译之前执行一下如上命令。

### ③ nfs 的设置

#### (1) 安装 Ubuntu nfs 服务

```
apt-get install nfs-kernel-server
```

#### (2) 修改 nfs 配置文件

```
vi /etc/exports
```

在文件中添加 Ubuntu nfs 的目录格式如下

```
/home/omap3evm_nfs *(rw, sync, no_subtree_check)
```

存盘退出

#### (3) 在根目录下建立 nfs 的目录

```
mkdir /home/omap3evm_nfs
```

修改该目录的权限

```
chmod 777 -R /home/omap3evm_nfs
```

#### (4) 重新启动 Ubuntu nfs

```
./etc/init.d/nfs-kernel-server restart
```

#### (5) 测试 nfs

```
mount 192.168.3.111:/home/omap3evm_nfs /mnt (192.168.3.111 为你本电脑的 IP)
```

```
cd /mnt (此时查看 mnt 目录内的内容就会和/home/omap3evm_nfs 的内容一样)
```

### ④ tftp 的安装和设置

#### (1) 步安装 tftp

```
sudo apt-get install tftp tftpd openbsd-inetd
```

#### (2) 在根目录下创建文件夹 tftpboot

```
cd /
```

```
sudo mkdir tftpboot 建立文件夹
```

```
sudo chmod 777 tftpboot 更改文件夹权限
```

(3) `sudo gedit /etc/inetd.conf` 修改成如下

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /tftpboot
```

(4) 在终端上执行如下指令

```
sudo /etc/init.d/openbsd-inetd reload
sudo /etc/init.d/xinetd restart
sudo in.tftpd -l /tftpboot
```

(5) 执行 `gedit /etc/xinetd.d/tftp` 命令, 将该文件修改如下 (如果没有 `tftp` 文件就创建它)。

```
Service tftp
{
    disable          =no
    socket_type      =dgram
    protocol         =udp
    wait            =yes
    user             =root
    server           =/usr/sbin/in.tftpd
    server_args      =-s /tftpboot -c
    source           = 11
    cps              = 100 2
}
```

(6) 执行 `sudo gedit /etc/default/tftpd-hpa`, 将该文件修改为:

```
RUN_DAEMON="no"
OPTIONS="-s /tftpboot -c -p -U tftpd"
```

(7) 在 `tftpboot` 文件夹下新建测试文件 `aaa`

```
cd /tftpboot
sudo touch aaa
sudo chmod 777 aaa
```

(8) 开始测试 `tftp` 服务

```
cd /home
sudo tftp 192.168.1.111 (192.168.3.111 为你本电脑的 IP)
get /tftpboot/aaa
```

如果没有出现错误代码且在 `home` 目录下出现 `aaa` 文件则证明 `tftp` 服务建立成功

**注意:** 1. 如果出现 `permission denied` 错误, 则是操作者权限不够, 需要提升权限, 执行 `su root`, 输入密码后就可以正常进行 `tftp` 传输操作了。  
2. 如果出现 `Access violation` 错误, 则是文件权限没有解开, 执行 `chmod 777 文件名`, 将要操作的文件操作权限全解开就可以了。

## 20.3 x-loader 编译

CPU 从外接的块设备中装载 x-loader 程序到片内存储器，并跳转到 x-loader 运行。此加载程序可以支持 NandFlash、MMC/SD 器件的加载功能。X-loader 程序运行后，能将片外存储器进行配置，然后将 UBoot 程序加载到片外存储器中，最后跳转到 UBoot 运行。

所以，可以理解：1. x-loader 程序运行在片内存储器，所以其程序较小，功能不可能过于复杂；  
2. 用来加载并运行更大、更复杂的加载程序 UBoot。

找到光盘中的 x-load.tar.bz2，复制到 Linux 下的/home 目录下，在终端下进入/home 目录，输入：sudo tar xvf x-load.tar.bz2

在/home 目录下建一个文件夹命名为 linux\_host，然后找到光盘内的 signGP 复制进入文件夹。

编译步骤：

在打开终端，进入/home/x-load 目录；

在终端依次输入下面的命令：

```
export PATH=/opt/arm-2007q3/bin:/home/u-boot/tools:$PATH
make distclean
make omap3evm_config
make
```

等待编译结束，生成 x-load.bin；

注意：x-load.bin 不能够直接烧写到 FLASH 中，在烧写之前需要进行转化。

转化步骤：

把编译得到的 x-load.bin 复制到 Linux 的/home/linux\_host/目录下。

打开 Linux 下的终端，输入下面的指令：

```
sudo ./home/linux_host/signGP /opt/linux_host/x-load.bin
```

执行完后在/opt/linux\_host/目录下会生成 x-load.bin.ift；

说明：x-load.bin.ift 为烧写 FLASH 时需要的 x-load 文件。

## 20.4 u-boot 编译

UBoot 程序是一个通用启动程序，可以跨平台编译运行，它支持多种通用器件如：DDR 存储器、NandFlash、网络芯片等，可以使用串口、以太网接口、USB 端口等通讯手段收发数据，还支持声卡、显卡等通用设备。UBoot 的灵活、外设的丰富、开源、通用等特点使它被越来越普遍地运用在嵌入式



系统中。

找到光盘中的 u-boot.tar.bz2，复制到 Linux 下的/home 目录下，在终端下进入/home 目录，输入：

```
sudo tar xvfz u-boot.tar.bz2
```

编译步骤：

在虚拟机里面打开终端，进入/home/u-boot 目录；

在终端依次输入下面的命令：

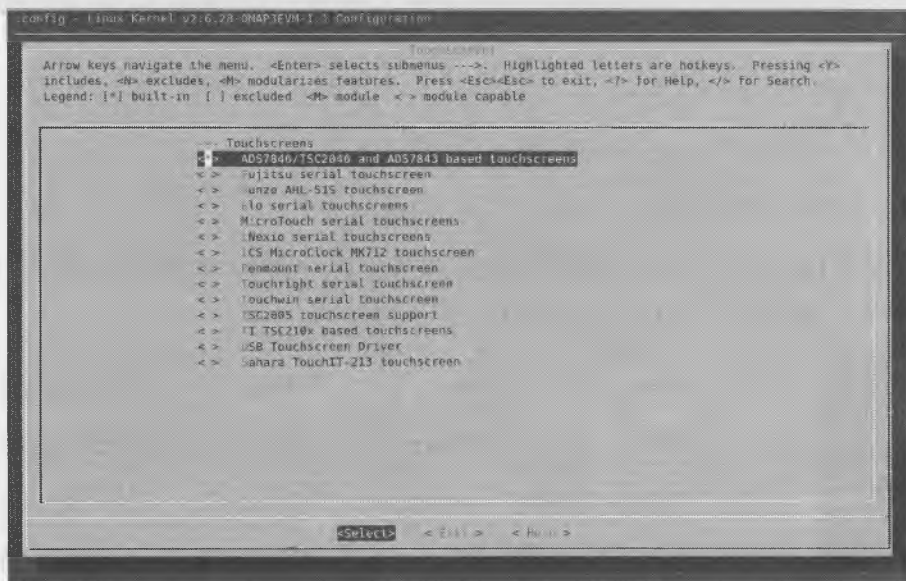
```
export PATH=/opt/arm-2007q3/bin:/home/u-boot/tools:$PATH
make distclean
make omap3evm_config
make
```

等待编译结束，编译结束后会在 u-boot 目录下生成一个 u-boot.bin 文件。

## 20.5 Techv-35XX Linux 驱动和内核配置及编译

Linux 内核编译，首先要进行配置，然后才能进行编译，如果设置不正确的话，编译出来的内核可能不能用。

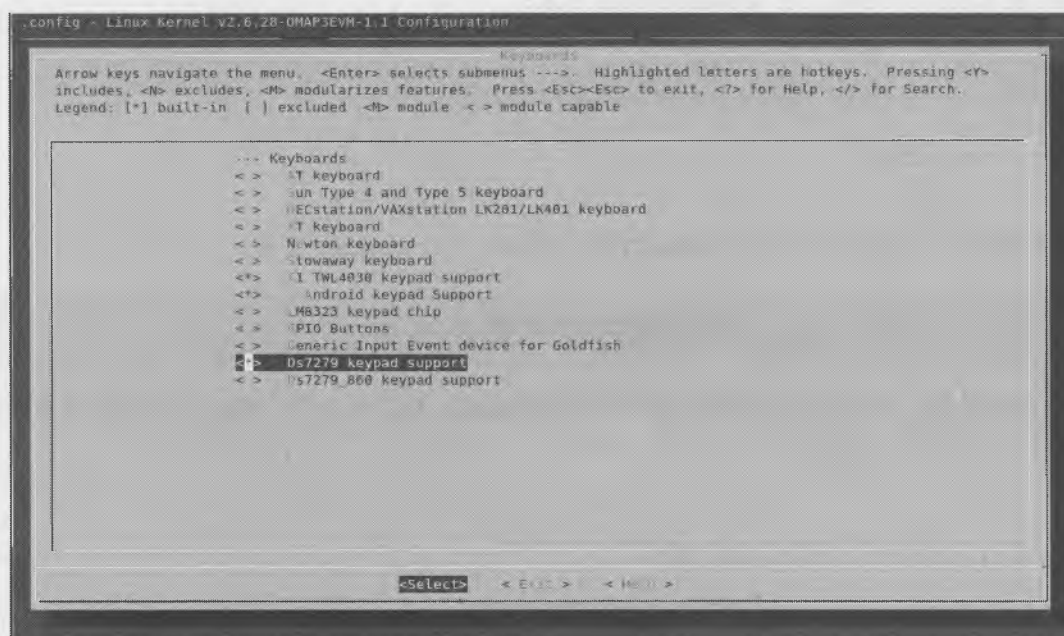
### 20.5.1 Touchscreen 驱动配置



```
Device Drivers --->
Input device support --->
```

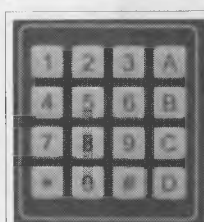
```
[*] Touchscreens --->
<*> ADS7846/TSC2046 and ADS7843 based touchscreens
```

## 20.5.2 KeyBoard 驱动配置



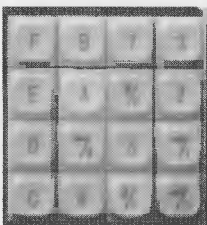
```
Device Drivers --->
Input device support --->
[*] Keyboards --->
<*> Android keypad Support
<*> Ds7279 keypad support
< > Ds7279_860 keypad support
```

如上选择是支持 EL-ARM-DSP-IV 的键盘驱动；如果是 EL-ARM-860 实验箱，请把 Ds7279 keypad support 这一项不选；然后选上 <\*>Ds7279\_860 keypad support 这一项。



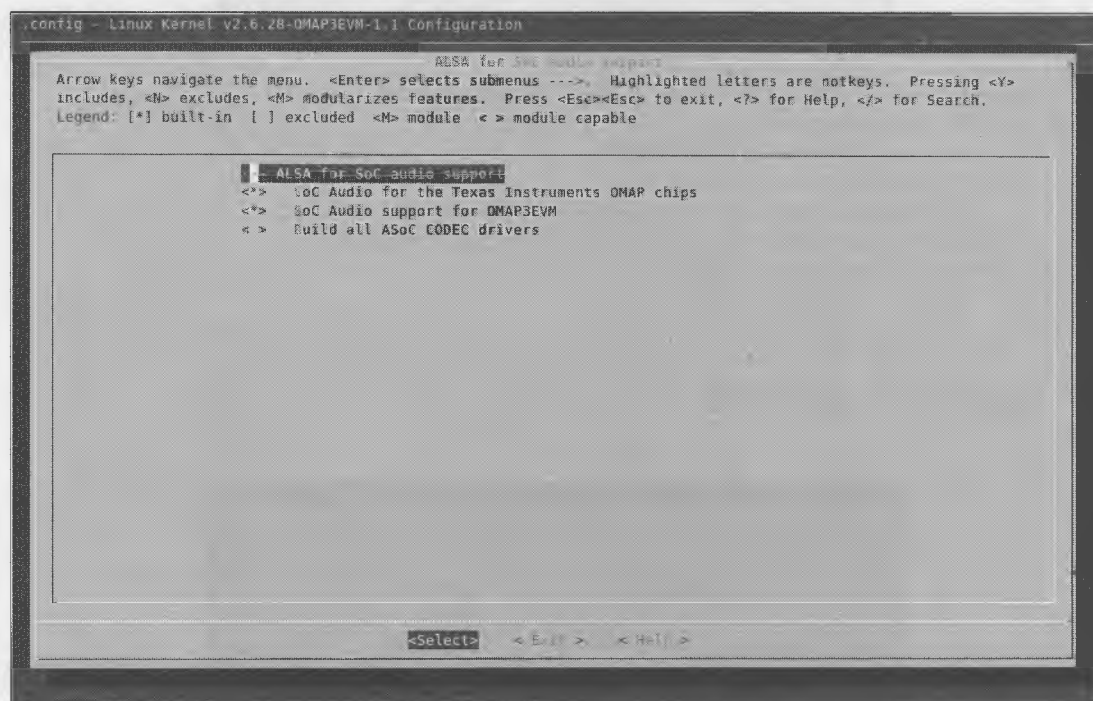
Menu	up	Home	Not Defined
Left	Enter	Right	Not Defined
Volume Down	Down	Volume Up	Not Defined
Not Defined	Back	Not Defined	Not Defined

EL-ARM-DSP-IV 的键盘及功能定义：

	Not Defined	Not Defined	Not Defined	Not Defined
	Not Defined	Menu	Up	Home
	Not Defined	Left	Enter	Right
	Back	Volume Down	Down	Volume Up

EL-ARM-860 实验箱的键盘及功能定义。

### 20.5.3 Audio 驱动配置

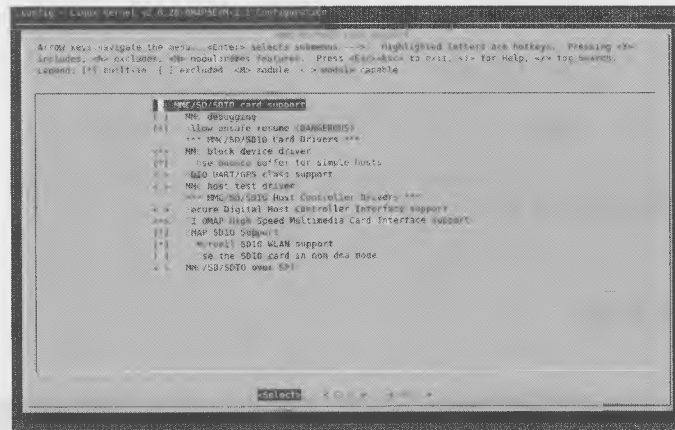


```

Device Drivers --->
<*> Sound card support --->
    <*> Advanced Linux Sound Architecture --->
        [*] Support old ALSA API
        [*] Verbose procfs contents
    <*> ALSA for SoC audio support --->
        <*> SoC Audio for the Texas Instruments OMAP chips
        <*> SoC Audio support for OMAP3EVM

```

## 20.5.4 4MMC/SD 驱动配置



Device Drivers ---&gt;

&lt;\*&gt; MMC/SD/SDIO card support ---&gt;

[\*] Allow unsafe resume (DANGEROUS)

&lt;\*&gt; MMC block device driver

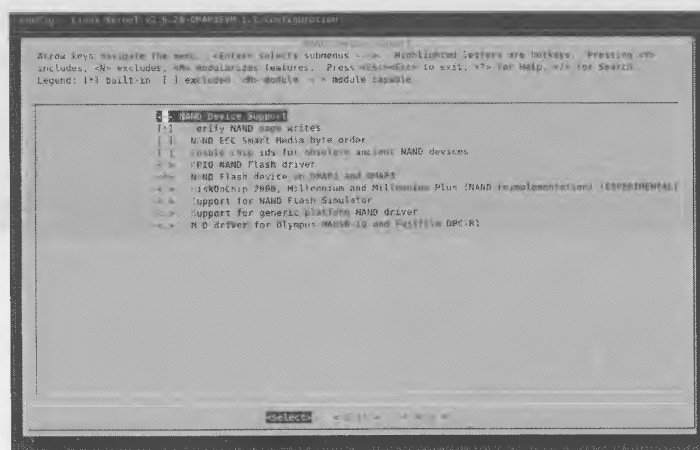
[\*] Use bounce buffer for simple hosts

&lt;\*&gt; TI OMAP High Speed Multimedia Card Interface support

[\*] OMAP SDIO Support

[\*] Marvell SDIO WLAN support

## 20.5.5 NandFlash 驱动配置



Device Drivers ---&gt;

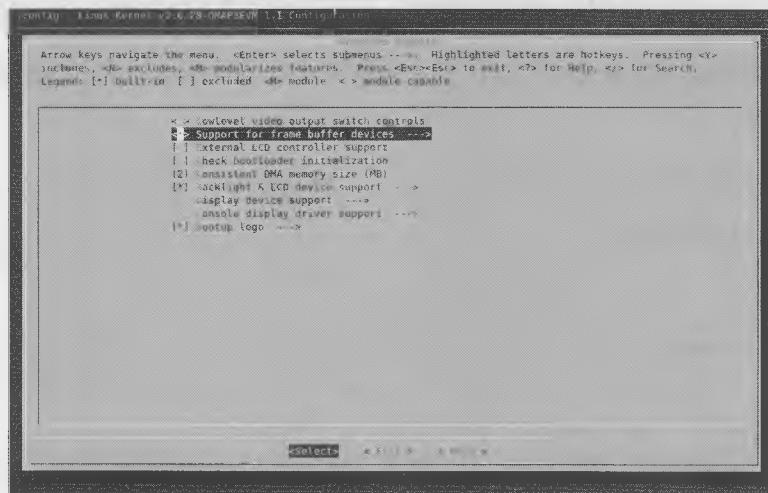
&lt;\*&gt; Memory Technology Device (MTD) support ---&gt;

```

<*> NAND Device Support --->
[*] Verify NAND page writes
<*> NAND Flash device on OMAP2 and OMAP3

```

### 20.5.6 LCD 驱动配置



```

Device Drivers --->
Graphics support --->
<*> Support for frame buffer devices --->
    <*> OMAP frame buffer support (EXPERIMENTAL)
    (2) Consistent DMA memory size (MB)
    [*] Backlight & LCD device support --->
        <*> Backlight support for OMAP3EVM VGA LCD Panel
    Console display driver support --->
        <*> Framebuffer Console support
    [*] Bootup logo --->

```

### 20.5.7 内核编译

找到光盘中的 linux-2.6.28-omap.tar.bz2, 复制到 Linux 下的/home 目录下, 在终端下进入/home 目录, 输入:

```
sudo tar xvzf linux-2.6.28-omap_android.tar.bz2
```

打开终端, 进入/home/linux-2.6.28-omap 目录;

在终端输入下面的命令:

```
export PATH=/opt/arm-2007q3/bin:/home/u-boot/tools:$PATH
make omap3_arm_dsp_iv_android_defconfig #使用 ARM-DSP-IV 默认的配置编译
```

```
(make omap3_arm_860_defconfig #使用 ARM-860 实验箱默认的配置编译)
#如果需要选择编译选项, 使用下面的命令
make menuconfig
make uImage
```

等待编译结束, 编译结束后会在 arch/arm/boot 目录下生成 uImage。

## 20.6 Techv-35XX Android 驱动编写

本节以 Android 下的键盘驱动为例, 通过这章的学习可以了解到 Android 驱动的编写过程。

### ① 键盘驱动注册

```
module_init(Kbd7279_Init);

module_exit(Kbd7279_Exit);
```

### ② 键盘初始化

```
void Setup_Kbd7279(void)
{
    int i,result;
    omap_cfg_reg(OMAP3_KBD_GPIO);
    if (gpio_request(OMAP3_KBD_GPIO, "kbd7279 IRQ") < 0)
        printk(KERN_ERR "Failed to request GPIO%d for kbd IRQ\n");
    gpio_direction_input(OMAP3_KBD_GPIO);
    set_irq_type(OMAP_GPIO_IRQ(OMAP3_KBD_GPIO),
        IRQ_TYPE_EDGE_FALLING); //IRQ_TYPE_EDGE_RISING
    enable_irq(gpio_to_irq(OMAP3_KBD_GPIO));
    if(result = request_irq(OMAP_GPIO_IRQ(OMAP3_KBD_GPIO), &Kbd7279_ISR, 0,
        "Ds7279",
        NULL))
    {
        printk(KERN_INFO "[FAILED: Cannot register Kbd7279_Interrupt!]\n");
        return -EBUSY;
    }
    else
        printk("[OK]\n");
    send_byte(cmd_reset);
    setcsl;
    printk("HD7279 setup complete!          --- > [OK]\n");
    udelay(10);
    omap_7279key = input_allocate_device();
    if (!omap_7279key) {
        kfree(omap_7279key);
        return -ENOMEM;
    }

    /* setup input device */
    omap_7279key->name= "Ds7279 Keyboard";
    set_bit(EV_KEY, omap_7279key->evbit); //EV_KEY
    for(i=0;i<10;i++)
        set_bit(omap3evm_keymap[i]& KEYNUM_MASK,omap_7279key->keybit);
```



```

input_register_device(omap_7279key);
printk("\nRegistering Input Device          --- > [OK]\n");
write7279(decode1+7,0x1);
write7279(decode1+6,0x2);
}

```

### ③ 键盘中断处理

```

static irqreturn_t Kbd7279_ISR(int irq,void* dev_id)
{
    int i;
    static char function = 1;
    disable_irq(gpio_to_irq(OMAP3_KBD_GPIO));
    for(i=0;i<100;i++);
    KeyValue = read7279();
    switch(KeyValue)
    {
        case 0x03:
            KeyValue=0x0d;
            break;
        case 0x02:
            KeyValue=0x0c;
            break;
        case 0x01:
            KeyValue=0x0b;
            break;
        case 0x00:
            KeyValue=0x0a;
            input_report_key(omap_7279key,omap3evm_keymap[10]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[10]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x08:
            KeyValue=0x03;
            input_report_key(omap_7279key,omap3evm_keymap[5]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[5]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x09:
            KeyValue=0x06;
            input_report_key(omap_7279key,omap3evm_keymap[1]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[1]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
    }
}

```



```

        case 0x0a:
            KeyValue=0x09;
            input_report_key(omap_7279key,omap3evm_keymap[8]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[8]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x0b:
            KeyValue=0x0e;
            input_report_key(omap_7279key,"MENU",omap3evm_keymap[1]&
(KEYNUM_MASK | KEY_PERSISTENT));//KEY(3, 3,
KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,"MENU",0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x10:
            KeyValue=0x02;
            input_report_key(omap_7279key,omap3evm_keymap[3]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[3]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x11:
            KeyValue=0x05;
            input_report_key(omap_7279key,omap3evm_keymap[7]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[7]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x12:
            KeyValue=0x08;
            input_report_key(omap_7279key,omap3evm_keymap[4]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[4]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x13:
            KeyValue=0x00;
            input_report_key(omap_7279key,omap3evm_keymap[6]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);

```

```

        input_report_key(omap_7279key,omap3evm_keymap[6]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
        input_sync(omap_7279key);
        break;
        case 0x18:
            KeyValue=0x01;
            input_report_key(omap_7279key,omap3evm_keymap[0]& (KEYNUM_MASK |
KEY_PERSISTENT),16);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[0]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x19:
            KeyValue=0x04;
            input_report_key(omap_7279key,omap3evm_keymap[2]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[2]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x1a:
            KeyValue=0x07;
            input_report_key(omap_7279key,omap3evm_keymap[9]& (KEYNUM_MASK |
KEY_PERSISTENT),1);//KEY(3, 3, KEY_KBDILLUMDOWN)
            udelay(10);
            input_report_key(omap_7279key,omap3evm_keymap[9]& (KEYNUM_MASK |
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
            input_sync(omap_7279key);
            break;
        case 0x1b:
            KeyValue=0x0f;
            if(!function) function = 1;
            else function = 0;
            break;
        default:
            break;
    }
    udelay(10);
    input_report_key(omap_7279key,KEY(1,          0,          KEY_HOME)&          (KEYNUM_MASK
KEY_PERSISTENT),0);//KEY(3, 1, KEY_UP)
    input_sync(omap_7279key);
    write7279(decodel+0,KeyValue%16);
    write7279(decodel+1,KeyValue/16);
    if(!function) write7279(disable,0x7f);
    else write7279(disable,0xff);
    kbd_buf = (unsigned char)KeyValue;
    enable_irq(gpio_to_irq(OMAP3_KBD_GPIO));
    return IRQ_HANDLED;
}

```

EL-ARM-DSP-IV 的键盘驱动文件 Ds7279.c 在内核 kernel 的/drivers/input/keyboard/下。EL-ARM-860 的键盘驱动文件 Ds7279\_860.c 在内核 kernel 的/drivers/input/keyboard/下。

#### ④ 编译

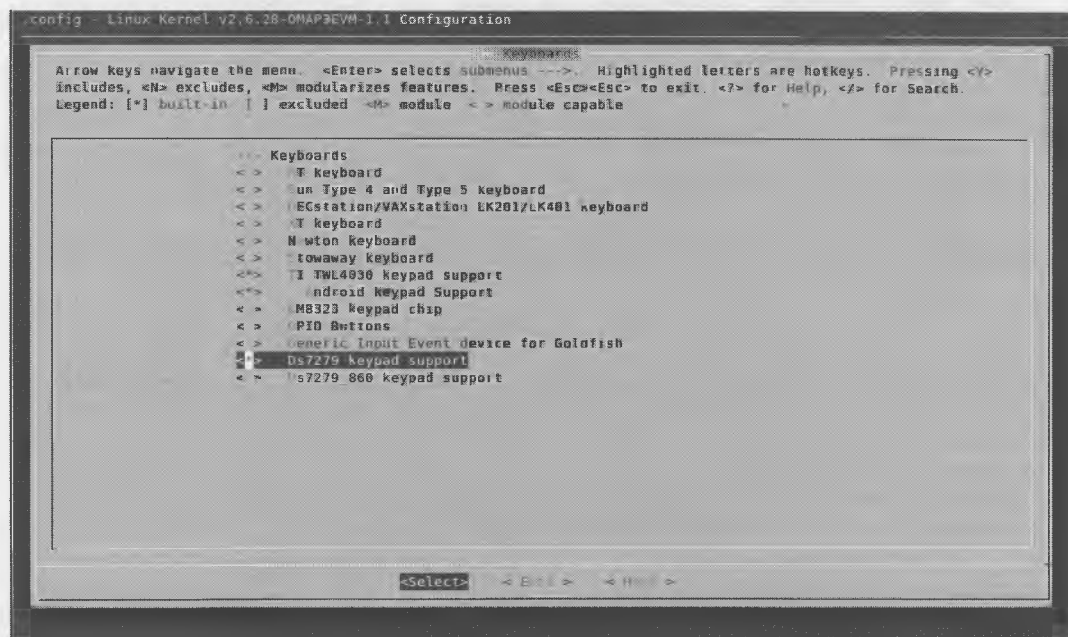
编写好驱动后，文件复制到内核源码包中，这里就是把 Ds727.c 复制到/drivers/input/keyboard/目录下。然后修改/drivers/input/keyboard/的 Makefile，在最后一行添加如下代码：

```
obj-$(CONFIG_KEYBOARD_MAPLE)      += maple_keyb.o
obj-$(CONFIG_KEYBOARD_BFIN)      += bf54x-keys.o
obj-$(CONFIG_KEYBOARD_SH_KEYSC)  += sh_keysc.o
obj-$(CONFIG_KEYBOARD_Ds7279)    += Ds7279.o
```

修改/drivers/input/keyboard/下的 Kconfig，添加如下代码：

```
config KEYBOARD_Ds7279
    tristate "Ds7279 keypad support"
    help
        To compile this driver as a module, choose M here: the
        module will be called Ds7279.
endif
```

在 make menuconfig 配置选项时如下选择，在编译内核时就会编译进内核，启动内核时就会自动加载键盘的驱动。



Device Drivers --->

```
Input device support --->
[*] Keyboards --->
    <*> Android keypad Support
    <*> Ds7279 keypad support
    < > Ds7279_860 keypad support
```

## 20.7 Techv-35XX Android 开发环境建立

### ① 基本环境的建立

在 ubuntu 下，需要一些特别的软件包，可使用以下命令进行安装：

```
sudo apt-get install git-core gnupg flex bison gperf libssl-dev libbsd0-dev
libwxgtk2.6-dev build-essential zip curl libncurses5-dev zlib1g-dev
```

如果在编译过程中还缺少其他的命令和软件，可以根据提示用以下命令来安装：

```
sudo apt-get install xxx (xxx 代表缺少的软件包名)
```

### ② 安装 JDK5

JDK-1.5 安装包，可以直接去 sun 官方主页下载，也可以使用我们在光盘中提供的 JDK-1.5 的安装包 jdk-1\_5\_0\_18-linux-i586.bin。

在/home 目录下建一个 java-develop 文件，把 jdk-1\_5\_0\_18-linux-i586.bin 复制到这个目录下；  
打开终端，进入 jdk 目录

```
cd /home/java-develop
```

更改文件权限为可执行

```
sudo chmod 777 jdk-1_5_0_18-linux-i586.bin
```

解压文件

```
sudo ./jdk-1_5_0_18-linux-i586.bin
```

yes/no 选择 yes，执行完之后边可以在 java-develop 目录下面看到文件夹 jdk1.5.0\_18，以 root 身份打开并编辑 profile 文件

```
sudo gedit /etc/bash.bashrc
```

在 bash.bashrc 文件最后添加以下内容：

```
export JAVA_HOME=/home /java-develop/jdk1.5.0_18
export ANDROID_JAVA_HOME=$JAVA_HOME
export PATH=$JAVA_HOME/bin:$PATH
```

保存并关闭，接着在终端下执行以下命令：

```
source /etc/bash.bashrc
```

在终端输入 `java -version` 将会显示 Java 版本的相关信息，表明 JDK 安装成功。

### ③ 安装 Android 源码包

我们在光盘中已经提 Android 的源码包，文件名为 `Android_donut.tar.gz`；将这个文件复制到 Ubuntu 主目录下运行解压指令 `sudo tar xvzf Android_donut.tar.gz` 这样就可以得到 Android 的源码目录了。

## 20.8 编译 Android Donut

Android 的编译非常简单，在终端中进入到 `Android_donut` 的目录下运行 `make` 即可以完成代码的编译，为了同时编译出配套的 Android SDK，可以使用 `make sdk` 进行编译：

```
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
administrator@ubuntu:/media/new/Android_Donut3$ make
=====
PLATFORM_VERSION_CODENAME=Techsheine_OMAP3530
PLATFORM_VERSION=Techsheine_OMAP3530
TARGET_PRODUCT=generic
TARGET_BUILD_VARIANT=eng
TARGET_SIMULATOR=
TARGET_BUILD_TYPE=release
TARGET_ARCH=arm
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=Donut
=====
*** Build configuration changed: "generic-eng-sdk-{" -> "generic-eng-{"
*** Forcing "make installclean"...
```

编译时间会很长，编译结果放在了 `android_donut/out` 下，其中 `out/host` 是主机上运行的一些程序，比如说编译出来的 `sdk` 就在 `android_donut/out/host/linux-x86/sdk` 下，这些程序需要用来帮助调试，因此使用下面的命令设置环境变量：

```
sudo gedit /etc/bash.bashrc
```

在 `bash.bashrc` 中增加以下内容：

```
export
ANDROID_SDK_ROOT=/home/administrator/Android_donut/out/host/linux-x86/sdk/android-sdk_eng.admi
nistrator_linux-x86
export PATH=$PATH$ANDROID_SDK_ROOT/tools
```

保存后运行 `source /etc/bash.bashrc` 让设置生效，重新开一个终端就能使用 `android`、`emulator` 等命令了。

另外，`android_donut/out/target/product/generic` 下放着所有用于板子上运行的程序，下面会介绍如何使用这些程序。

## 20.9

## Android 根文件系统的制作

进入到 `android_donut/out/target/product/generic` 目录下，把 `system` 这个目录复制到开发板中 `root` 目录，覆盖原有的 `system` 目录。步骤如下：

打开终端进入到 `android_donut/out/target/product/generic/root` 目录下修改文件属性，输入如下命令：

```
sudo chmod -R 777 *
```

修改 `root` 文件夹下的 `init.rc`，注释掉以下几行：

```
mount rootfs rootfs / ro remount
mount yaffs2 mtd@system /system
mount yaffs2 mtd@system /system ro remount
mount yaffs2 mtd@userdata /data nosuid nodev
mount yaffs2 mtd@cache /cache nosuid nodev
service flash_recovery /system/etc/install-recovery.sh
oneshot
```

在终端下进入到 `Android_donut/out/target/product/generic/root`（注意必须进入这个文件夹下），输入如下命令把文件系统打包：

```
tar xvzf android_rootfs.tar.gz *
```

`android_rootfs.tar.gz` 这个文件就可以安装到板子上了。

**注意：**这里只是简单介绍一个可用根文件系统的制作方式，所以制作出来的根文件系统在板子上还不能使用 SD 卡和 Audio，如果要使用这两个设备请使用我们已经制作好的 `android_rootfs.tar.gz`。

## 20.10

## 相关 Image 文件的烧写

本节是要烧写前面编译生成的 `x-load.bin.ift`、`u-boot.bin`，这些文件已经复制到 `tftpboot` 目录下。

电缆线连接方式，Techv-3530 的串口电缆线一头连接到 PC，一头连接到 P9。用网线把 Techv-3530 的网口和 PC 的网络连接起来。

### ① 烧写 x-load

在超级终端下依次输入下面的指令：

```
tftpboot 0x81600000 x-load.bin.ift
nand unlock
nand erase 0 20000
nand ecc hw
nand write 0x81600000 0 20000
```

## ② 烧写 u-boot

在超级终端下依次输入下面的指令

```
tftpboot 0x81600000 u-boot.bin
nand unlock
nand erase 80000 40000
nand ecc sw
nand write 0x81600000 80000 40000
```

等待烧写完，然后输入以下的指令设置 u-boot

```
setenv ipaddr 192.168.3.157      (板子 IP, 根据实际情况设)
setenv serverip 192.168.3.158   (虚拟机的 IP, 根据实际情况设)
setenv netmask 255.255.255.0
setenv bootdelay 6
setenv ethaddr 00:50:c2:7e:8A:1D
setenv gatewayip 192.168.3.1    (网关, 根据实际情况设)

setenv get_kernel 'nand read 0x80000000 280000 200000'
setenv get_initrd 'nand read 0x81600000 780000 1000000'
setenv bootargs_rd 'mem=128M console=ttyS0,115200n8 root=/dev/ram0 initrd=0x81600000,40M
ramdisk_size=40960'

setenv bootargs_nfs 'mem=128M console=ttyS0,115200n8 noinitrd rw root=/dev/nfs
nfsroot=/home/omap3evm_nfs,nolock'

setenv bootargs_mmc 'mem=128M console=ttyS0,115200n8 noinitrd rw root=/dev/mmcblk0p2 init=/init'

setenv addip 'setenv bootargs $(bootargs)
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname):eth0:on'

setenv boot_mmc 'run get_kernel; setenv bootargs $(bootargs_mmc); run addip; bootm 0x80000000'

setenv boot_nfs 'run get_kernel; setenv bootargs $(bootargs_nfs); run addip; bootm
0x80000000'

setenv bootcmd run boot_mmc
saveenv      (保存设置好的参数)
```

等待烧写完成，断电，并将 Techv-3530 的串口电缆线从 P9 换到 P3。

## ③ kernel 烧写

要求需要烧写前面编译生成的 uImage 已经复制到/tftpboot 目录下。

在超级终端下依次输入下面的指令

```
tftpboot 0x81600000 uImage
nand unlock
nand erase 280000 200000
nand ecc sw
nand write 0x81600000 280000 200000
```



等待烧写完成，断电。

## 20.11 Android 根文件系统安装

Android 的根文件系统是安装在 SD 卡内的，首先要把 Sd 卡分区和格式化，然后再把 Android 的根文件系统安装到 SD 卡

### ① 首先确定 SD 卡的挂载点

用 SD 卡插入到读卡器，把读卡器插到 PC，在终端下输入 `dmesg` 命令

```
administrator@ubuntu:~$ dmesg
[ 65.779748] sd 2:0:0:0: [sdd] Write Protect is off
[ 65.779754] sd 2:0:0:0: [sdd] Mode Sense: 4b 00 00 08
[ 65.779759] sd 2:0:0:0: [sdd] Assuming drive cache: write through
[ 65.784496] sd 2:0:0:0: [sdd] Assuming drive cache: write through
[ 65.784503] sdd: sddl
[ 65.788762] sd 2:0:0:0: [sdd] Assuming drive cache: write through
[ 65.788770] sd 2:0:0:0: [sdd] Attached SCSI removable disk
```

### ② 卡插一般会自动挂载，先要卸载 SD 卡

```
administrator@ubuntu:~$ umount /dev/sdd1

administrator@ubuntu:~$ sudo fdisk /dev/sdd

WARNING: DOS-compatible mode is deprecated. It's strongly recommended to
        switch off the mode (command 'c') and change display units to
        sectors (command 'u').

Command (m for help):p ↵

Disk /dev/sdd: 1967 MB, 1967128576 bytes
255 heads, 63 sectors/track, 239 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x006c3357

   Device Boot      Start         End      Blocks   Id  System
/dev/sdd1    *           1          240     1920992+   c   W95 FAT32 (LBA)
Partition 1 has different physical/logical endings:
     phys=(238, 254, 63) logical=(239, 39, 56)

Command (m for help):d ↵
Selected partition 1

Command (m for help):n ↵
Command action
   e   extended
```

```

p primary partition (1-4)
p ↓
Partition number (1-4): 1 ↓
First cylinder (1-239, default 1): ↓
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-239, default 239): 128M ↓

Command (m for help): t ↓
Selected partition 1
Hex code (type L to list codes): c ↓
Changed system type of partition 1 to c (W95 FAT32 (LBA))

Command (m for help): a ↓
Partition number (1-4): 1 ↓

Command (m for help): n ↓
Command action
e extended
p primary partition (1-4)
p ↓
Partition number (1-4): 2 ↓
First cylinder (129-239, default 129): ↓
Using default value 129
Last cylinder, +cylinders or +size{K,M,G} (129-239, default 239): ↓
Using default value 239

Command (m for help): p ↓

Disk /dev/sdd: 1967 MB, 1967128576 bytes
255 heads, 63 sectors/track, 239 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x006c3357

   Device Boot      Start         End      Blocks   Id  System
/dev/sdd1    *           1         128     1028128+    c  W95 FAT32 (LBA)
/dev/sdd2             129        239      891607+   83   Linux

Command (m for help): w ↓
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
administrator@ubuntu:~$

```

### 3 格式化 SD 卡分区

```
administrator@ubuntu:~$ sudo mkfs.msos -F 32 /dev/sdd1 -n disk1
administrator@ubuntu:~$ sudo mkfs.ext3 -L disk2 /dev/sdd2
```

#### ④ 安装 Android 根文件到 disk2

把 SD 卡插入电脑，找到提供的 android\_rootfs.tar.gz 或者是第 20.4 节制作的根文件系统文件，把这个文件复制到/home 下，在终端下进入到/home 这个目录，输入如下命令：

```
sudo tar xvfz android_rootfs.tar.gz -C /media/disk2
```

通过以上几步的操作，把 SD 卡插入到 Techv-OMAP3530 板，给实验箱上电，等待一会系统启动，启动后液晶屏上显示系统界面。

- EL-ARM-DSP-IV 实验箱上的开关设置：SW4：1、2、4 ON；
- S5：2 ON；S4：2、3 ON；其他开关全部 OFF；
- EL-ARM-860 实验箱开关设置：SW4：1 OFF，2 ON，其他开关 OFF。

**附录：** 以下是 Techv-35XX 开发板附带光盘内容说明。

文件夹 image：已经编译好的 Bootload 和内核 kernel 烧写文件。

文件 Android\_donut.tar.gz：Android donut 源码包。

文件 android\_rootfs.tar.gz：已经制作好的 Android 根文件系统。

文件 arm-2007q3-51-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2：Bootload 和 kernel 交叉编译器。

文件 x-load.tar.bz2：x-load 源码包。

文件 u-boot.tar.bz2：u-boot 源码包。

文件 linux-2.6.28-omap\_android.tar.gz：kernel 源码包。

文件 jdk-1\_5\_0\_18-linux-i586.bin：JDK5 安装包。

文件 eclipse-SDK-3.6-linux-gtk.tar.gz：eclipse 安装包。